

**IT-302**

**COMPILER DESIGN**

**LAB FILE**

**DELHI TECHNOLOGICAL UNIVERSITY**



Submitted to:

Ms. Anita Thakur

Submitted by:

Varun Kumar  
(2K19/IT/140)

# INDEX

S.No	Topics	Date	Sign
1	Write a program to implement a lexical analyzer for the given source code stored in the input file.	11/01/22	
2	Write a program to convert a NFA to DFA using subset construction.	18/01/22	
3	Write a program for creating tokens for the high-level program.	25/01/22	
4	Write a program to find the FIRST SET for given production of the grammar.	08/02/22	
5	Write a program to find the FOLLOW SET for given production of the grammar.	22/02/22	
6	Write a program to implement NFA with Epsilon move to DFA.	22/02/22	
7	Write a program for the given grammar G of a fictitious language L and parse the input string $w=id+id*id$ , using Recursive Descent Parsing Algorithm.	08/03/22	
8	Write a program to convert left recursive grammar to its equivalent right recursive grammar.	29/03/22	
9	Write a program to left factor the grammar.	29/03/22	
10	Write a program to parse the string $id=num-num$ using LL(1) predictive parser.	05/04/22	
11	Write a program to obtain output using lex and yacc.	05/04/22	
12	Write a program to generate the intermediate representation (IR code) using Syntax Directed Translation. The input will be 4-address code and output generated will be 3-address code.	12/04/22	

# Program 1

**Objective :** Write a program to Implement a lexical analyzer for a given source code stored in an input file.

**Theory :**

Lexical analysis is the first phase of a compiler. It takes 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

**Code :**

```
#include<bits/stdc++.h>
using namespace std;

int isKeyword(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() {

    char ch, buffer[18], operators[] = "+-*/%=<>", other[] = ",;(){ }[]\":";
    ifstream fin("input.txt");
```

```
int i, j = 0;
int keyword = 0;
int identifier = 0;
int number = 0;
int operatr = 0;
int literal = 0;
int others = 0;
int str_check = 0;
```

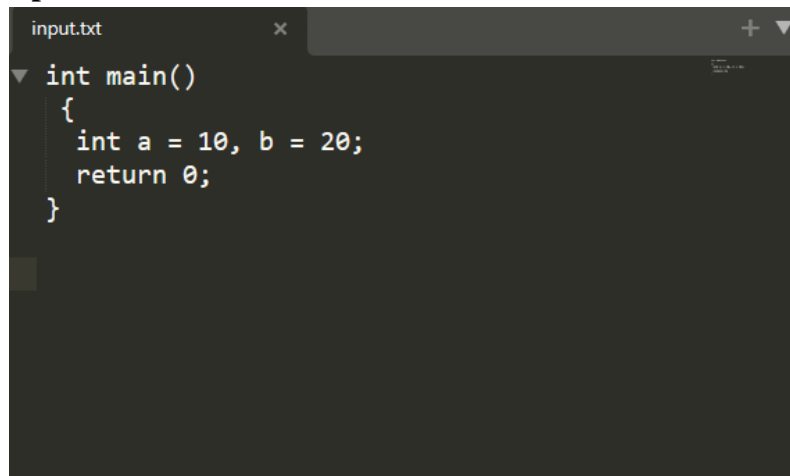
```
if (!fin.is_open()) {
    cout << "Not able to read file\n";
    exit(0);
}
while (!fin.eof()) {
    ch = fin.get();
    bool operato = 0;
    for (i = 0; i < 8; ++i) {
        if (ch == operators[i])
        {
            operato = 1;
            operatr++;
        }
    }
    for (i = 0; i < 12; ++i) {
        if (ch == other[i])
        {
            operato = 1;
            others++;
        }
    }
    if (isalnum(ch)) {
        buffer[j++] = ch;
    }
    else if (ch == "\\") {
        char charctr;
        j++;
        literal++;
        while (!fin.eof()) {
            charctr = fin.get();
            if (charctr == "\\")
```

```

        break;
    }
}
else if ((ch == ' ' || ch == '\n' || operato) && (j != 0)) {
    buffer[j] = '\0';
    if (isKeyword(buffer) == 1)
    {
        keyword++;
    }
    else if (buffer[0] >= '0' and buffer[0] <= '9')
    {
        number++;
    }
    else
    {
        identifier++;
    }
    j = 0;
}
}
cout << "\n";
cout << "Separators - " << others << endl;
cout << "Identifiers - " << identifier << endl;
cout << "Keywords - " << keyword << endl;
cout << "Operators - " << operatr << endl;
cout << "Numbers - " << number << endl;
cout << "String Literals - " << literal << endl;
fin.close();
return 0;
}

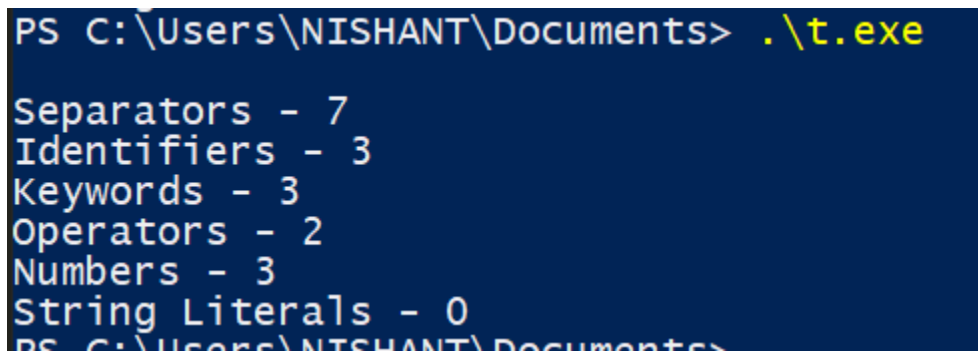
```

### Input :

A screenshot of a code editor window titled 'input.txt'. The code is written in C and defines a 'main' function. Inside the function, two integer variables 'a' and 'b' are declared and initialized with values 10 and 20 respectively. The function then returns 0. The code is as follows:

```
int main()
{
    int a = 10, b = 20;
    return 0;
}
```

### Output :

A screenshot of a Windows command prompt window. The prompt shows the directory 'C:\Users\NISHANT\Documents' and the command '.\t.exe' has been executed. The output of the program is displayed as follows:

```
PS C:\Users\NISHANT\Documents> .\t.exe

Separators - 7
Identifiers - 3
Keywords - 3
Operators - 2
Numbers - 3
String Literals - 0
PS C:\Users\NISHANT\Documents>
```

### Result and Discussion:

Lexical analyzer scans the input file 'input\_code.txt' and classifies all lexeme into tokens and we get :

- 7 -Separators
- 3 –Identifiers
- 3 – Keywords
- 2 - Operators
- 3 – Numbers
- 0 – String literals

## Program 2

**OBJECTIVE :** Convert nfa to dfa using the subset construction method.

**Theory :**

**Non-deterministic finite automata(NFA)** is a finite automata where for some cases when a specific input is given to the current state, the machine goes to multiple states or more than 1 state. It can contain  $\epsilon$  moves. It can be represented as  $M = \{ Q, \Sigma, \delta, q_0, F \}$ .

**Deterministic Finite Automata (DFA)** : DFA is a finite automata where, for all cases, when a single input is given to a single state, the machine goes to a single state, i.e., all the moves of the machine can be uniquely determined by the present state and the present input symbol.

Converting NFA to its equivalent DFA. In NFA, when a specific input is given to the current state, the machine goes to multiple states. It can have zero, one or more than one move on a given input symbol. On the other hand, in DFA, when a specific input is given to the current state, the machine goes to only one state. DFA has only one move on a given input symbol.

Let,  $M = (Q, \Sigma, \delta, q_0, F)$  is an NFA which accepts the language  $L(M)$ . There should be equivalent DFA denoted by  $M' = (Q', \Sigma', q'_0, \delta', F')$  such that  $L(M) = L(M')$ .

**Algorithm :**

**Step 1:** Initially  $Q' = \phi$

**Step 2:** Add  $q_0$  of NFA to  $Q'$ . Then find the transitions from this start state.

**Step 3:** In  $Q'$ , find the possible set of states for each input symbol. If this set of states is not in  $Q'$ , then add it to  $Q'$ .

**Step 4:** In DFA, the final state will be all the states which contain  $F$  (final states of NFA)

**CODE:**

```
#include <bits/stdc++.h>
using namespace std;
#define pb push_back
map<char, string> nfazero;
map<char, string> nfaone;
map<char, string> epsilon;
```

```

string depth_first_search(char curr)
{
    string ret = "";
    ret.push_back(curr);
    string neighbors = epsilon[curr];
    for (auto x : neighbors)
    {
        if (x != '-')
            ret += depth_first_search(x);
    }
    return ret;
}

int main()
{
    int n;
    cin >> n;
    int k;
    cin >> k;
    cout << "STATES OF NFA : ";
    for (int i = 0; i < n; i++)
    {
        cout << char('A' + i) << " ";
    }
    cout << "\n\n";
    cout << "GIVEN SYMBOLS FOR NFA: ";
    for (int i = 0; i < k; i++)
    {
        cout << i << " ";
    }
    cout << "\n\nNFA STATE TRANSITION TABLE \n\n";
    cout << "STATES |0 |1 |eps\n";
    cout << "-----+-----+-----+----\n";
    for (int i = 0; i < n; i++)
    {
        string a, b, c;
        cin >> a;
        nfazero.insert({'A' + i, a});
        cin >> b;
        nfaone.insert({'A' + i, b});
        cin >> c;
        epsilon.insert({'A' + i, c});
        cout << char('A' + i) << " " << a << " " << b << " " << c << endl;
    }
    cout << endl;
    map<char, string> closure;
    for (int i = 0; i < n; i++)

```



```

{
    string eps = depth_first_search('A' + i);
    sort(eps.begin(), eps.end());
    closure['A' + i] = eps;
    cout << "e-closure (" << char('A' + i) << ") : " << eps << endl;
}
cout << endl;
string start = closure['A'];
map<string, int> vis;
vector<string> states;
map<string, string> dfa0;
map<string, string> dfa1;
states.pb(start);
vis[start]++;
for (int i = 0; i < states.size(); i++)
{
    string x = states[i];
    string a = "";
    for (auto y : x)
    {
        if (nfazero[y] != "-")
            a += nfazero[y];
    }
    set<char> ans;
    for (auto y : a)
    {
        string gg = closure[y];
        for (auto z : gg)
            ans.insert(z);
    }
    string dfa = "";
    for (auto y : ans)
        dfa += y;
    if (vis[dfa] == 0)
    {
        vis[dfa]++;
        states.pb(dfa);
    }
    dfa0[x] = dfa;
    string b = "";
    for (auto y : x)
    {
        if (nfaone[y] != "-")
            b += nfaone[y];
    }
    set<char> ans1;

```

```

for (auto y : b)
{
    string gg = closure[y];
    for (auto z : gg)
        ans1.insert(z);
}
dfa = "";
for (auto y : ans1)
    dfa += y;
if (vis[dfa] == 0)
{
    vis[dfa]++;
    states.pb(dfa);
}
// cout<<x<<" "<<dfa<<endl;
dfa1[x] = dfa;
}
cout << "****\n";
cout << "\tDFA TRANSITION STATE TABLE \n";
cout << "STATES OF DFA : ";
for (auto x : states)
    cout << x << " ";
cout << "\n\n";
cout << "GIVEN SYMBOLS FOR DFA : ";
for (int i = 0; i < k; i++)
    cout << i << " ";
cout << "\n\n";
cout << "STATES |0 |1 \n";
cout << "-----+-----\n";
for (auto x : states)
{ cout << x << " | " << dfa0[x] << " | " << dfa1[x] << endl; }
return 0;
}

```

**OUTPUT:**

```
PS C:\Users\NISHANT\Documents> .\t.exe
```

```
11
```

```
2
```

```
STATES OF NFA : A B C D E F G H I J K
```

```
GIVEN SYMBOLS FOR NFA: 0 1 eps
```

```
NFA STATE TRANSITION TABLE
```

```
STATES |0 |1 |eps
```

```
-----+-----+-----+-----
```

```
-
```

```
-
```

```
BH
```

```
A - - BH
```

```
-
```

```
-
```

```
CE
```

```
B - - CE
```

```
D
```

```
-
```

```
-
```

```
C D - -
```

```
-
```

```
-
```

```
G
```

```
D - - G
```

```
-
```

```
F
```

```
-
```

```
E - F -
```

```
-
```

```
-
```

```
G
```

```
F - - G
```

```
-
```

```
-
```

```
HB
```

```
G - - HB
```

```
I
```

```
-
```

```
-
```

```
H I - -
```

```
-
```

```
J
```

```
-
```

```
I - J -
```

```
-
```

```
K
```

```
-
```

```

J - K -
-
-
-
K - - -

```

```

e-closure (A) : ABCEH
e-closure (B) : BCE
e-closure (C) : C
e-closure (D) : BCDEGH
e-closure (E) : E
e-closure (F) : BCEFGH
e-closure (G) : BCEGH
e-closure (H) : H
e-closure (I) : I
e-closure (J) : J
e-closure (K) : K

```

\*\*\*\*

#### DFA TRANSITION STATE TABLE

STATES OF DFA : ABCEH BCDEGHI BCEFGH BCEFGHJ BCEFGHK

GIVEN SYMBOLS FOR DFA : 0 1

STATES | 0 | 1

-----+-----+-----

ABCEH	BCDEGHI	BCEFGH
BCDEGHI	BCDEGHI	BCEFGHJ
BCEFGH	BCDEGHI	BCEFGH
BCEFGHJ	BCDEGHI	BCEFGHK
BCEFGHK	BCDEGHI	BCEFGH

PC : C++ | Home : NTCHANET | Documents :

## Program 3

**OBJECTIVE :** WAP for creating tokens for the high-level program  $a = b + c * 30$

### 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.

### Algorithm :

STEP 1. Scan the string character by character

STEP 2. If the symbol encountered is from a-z or A-Z, add the symbol to map with giving a suitable id and increment static row number

STEP 3. If the symbol encountered is one of the operators or any punctuation, push it to the map  
4. Traverse the string path and print respective values from map

### CODE:

```
#include<bits/stdc++.h>
using namespace std;
```

```
bool isPunctuator(char ch) //check if the given character is a punctuator or not
{
    if (ch == ' ' || ch == '+' || ch == '-' || ch == '*' ||
        ch == '/' || ch == ';' || ch == ':' || ch == '>' ||
        ch == '<' || ch == '=' || ch == '(' || ch == ')' ||
        ch == '[' || ch == ']' || ch == '{' || ch == '}' ||
        ch == '&' || ch == '|')
```

```

{
    return true;
}
return false;
}

```

```

bool validIdentifier(char* str) //check if the given identifier is valid or not
{
    if (str[0] == '0' || str[0] == '1' || str[0] == '2' ||
        str[0] == '3' || str[0] == '4' || str[0] == '5' ||
        str[0] == '6' || str[0] == '7' || str[0] == '8' ||
        str[0] == '9' || isPunctuator(str[0]) == true)
    {
        return false;
    } //if first character of string is a digit or a special character, identifier is not valid
    int i, len = strlen(str);
    if (len == 1)
    {
        return true;
    } //if length is one, validation is already completed, hence return true
    else
    {
        for (i = 1 ; i < len ; i++) //identifier cannot contain special characters
        {
            if (isPunctuator(str[i]) == true)
            {
                return false;
            }
        }
    }
    return true;
}

```

```

bool isOperator(char ch) //check if the given character is an operator or not
{
    if (ch == '+' || ch == '-' || ch == '*' ||
        ch == '/' || ch == '>' || ch == '<' ||
        ch == '=' || ch == '|' || ch == '&')
    {
        return true;
    }
}

```

```

    }
    return false;
}

```

bool isNumber(char\* str) //check if the given substring is a number or not

```

{
    int i, len = strlen(str), numOfDecimal = 0;
    if (len == 0)
    {
        return false;
    }
    for (i = 0 ; i < len ; i++)
    {
        if (numOfDecimal > 1 && str[i] == '.')
        {
            return false;
        } else if (numOfDecimal <= 1)
        {
            numOfDecimal++;
        }
        if (str[i] != '0' && str[i] != '1' && str[i] != '2'
            && str[i] != '3' && str[i] != '4' && str[i] != '5'
            && str[i] != '6' && str[i] != '7' && str[i] != '8'
            && str[i] != '9' || (str[i] == '-' && i > 0))
        {
            return false;
        }
    }
    return true;
}

```

bool isKeyword(char \*str) //check if the given substring is a keyword or not

```

{
    if (!strcmp(str, "if") || !strcmp(str, "else") ||
        !strcmp(str, "while") || !strcmp(str, "do") ||
        !strcmp(str, "break") || !strcmp(str, "continue")
        || !strcmp(str, "int") || !strcmp(str, "double")
        || !strcmp(str, "float") || !strcmp(str, "return")
        || !strcmp(str, "char") || !strcmp(str, "case")
        || !strcmp(str, "long") || !strcmp(str, "short"))
    {
        return true;
    }
    return false;
}

```

```

    || !strcmp(str, "typedef") || !strcmp(str, "switch")
    || !strcmp(str, "unsigned") || !strcmp(str, "void")
    || !strcmp(str, "static") || !strcmp(str, "struct")
    || !strcmp(str, "sizeof") || !strcmp(str, "long")
    || !strcmp(str, "volatile") || !strcmp(str, "typedef")
    || !strcmp(str, "enum") || !strcmp(str, "const")
    || !strcmp(str, "union") || !strcmp(str, "extern")
    || !strcmp(str, "bool"))
{
    return true;
}
else
{
    return false;
}
}

```

char\* subString(char\* realStr, int l, int r) //extract the required substring from the main string

```

{
    int i;

    char* str = (char*) malloc(sizeof(char) * (r - l + 2));

    for (i = l; i <= r; i++)
    {
        str[i - l] = realStr[i];
        str[r - l + 1] = '\0';
    }
    return str;
}

```

void parse(char\* str) //parse the expression

```

{
    int left = 0, right = 0;
    int len = strlen(str);
    while (right <= len && left <= right) {
        if (isPunctuator(str[right]) == false) //if character is a digit or an alphabet
        {
            right++;

```



```

}

if (isPunctuator(str[right]) == true && left == right) //if character is a punctuator
{
    if (isOperator(str[right]) == true)
    {
        std::cout << str[right] << " IS AN OPERATOR\n";
    }
    right++;
    left = right;
} else if (isPunctuator(str[right]) == true && left != right
           || (right == len && left != right)) //check if parsed substring is a keyword or identifier
or number
{
    char* sub = subString(str, left, right - 1); //extract substring

    if (isKeyword(sub) == true)
    {
        cout << sub << " IS A KEYWORD\n";
    }
    else if (isNumber(sub) == true)
    {
        cout << sub << " IS A NUMBER\n";
    }
    else if (validIdentifier(sub) == true
             && isPunctuator(str[right - 1]) == false)
    {
        cout << sub << " IS A VALID IDENTIFIER\n";
    }
    else if (validIdentifier(sub) == false
             && isPunctuator(str[right - 1]) == false)
    {
        cout << sub << " IS NOT A VALID IDENTIFIER\n";
    }

    left = right;
}
}
return;
}

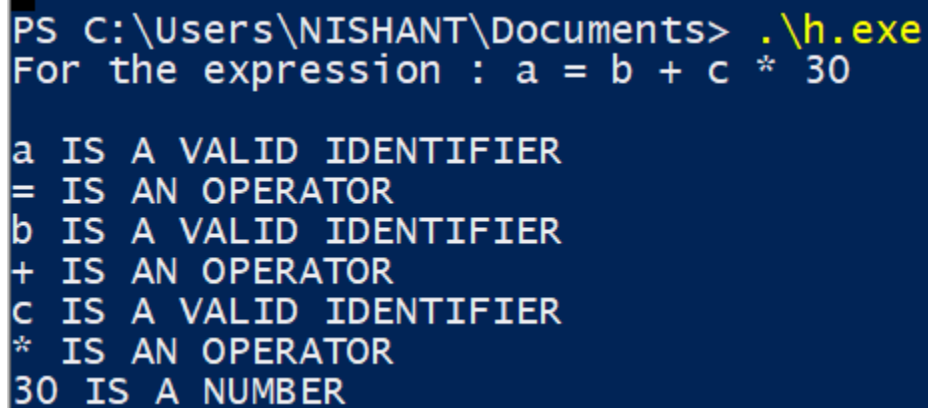
```

```
int main()
{
    char c[100] = "a = b + c * 30";
    cout << "For the expression : ";
    for (int i = 0; i < 14; i++)
        cout << c[i];

    cout << endl;
    cout << endl;

    parse(c);
    return 0;
}
```

#### OUTPUT:



```
PS C:\Users\NISHANT\Documents> .\h.exe
For the expression : a = b + c * 30

a IS A VALID IDENTIFIER
= IS AN OPERATOR
b IS A VALID IDENTIFIER
+ IS AN OPERATOR
c IS A VALID IDENTIFIER
* IS AN OPERATOR
30 IS A NUMBER
```

## Program 4

**OBJECTIVE :** Write a Program to find the FIRST SET for given production of the grammar.

$S \rightarrow ACB / CbB / Ba$

$A \rightarrow da / BC$

$B \rightarrow g / \epsilon$

$C \rightarrow h / \epsilon$

### Theory :

**FIRST ()**– It is a function that gives the set of terminals that begin the strings derived from the production rule.

A symbol  $c$  is in  $FIRST(\alpha)$  if and only if  $\alpha \Rightarrow c\beta$  for some sequence  $\beta$  of grammar symbols.

A terminal symbol  $a$  is in  $FOLLOW(N)$  if and only if there is a derivation from the start symbol  $S$  of the grammar such that  $S \Rightarrow \alpha N \alpha \beta$ , where  $\alpha$  and  $\beta$  are a (possibly empty) sequence of grammar symbols. In other words, a terminal  $c$  is in  $FOLLOW(N)$  if  $c$  can follow  $N$  at some point in a derivation.

### Algorithm :

#### Computation of FIRST

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

$FIRST(\alpha) = \{a \mid \alpha \rightarrow^* a\beta \text{ for some string } \beta\}$

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

If  $X$  is a terminal symbol, then  $FIRST(X) = \{X\}$

If  $X \rightarrow \epsilon$ , then  $FIRST(X) = \{\epsilon\}$

If  $X$  is non-terminal &  $X \rightarrow a \alpha$ , then  $FIRST(X) = \{a\}$

If  $X \rightarrow Y_1, Y_2, Y_3$ , 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  $\epsilon$  then,  $FIRST(X) = FIRST(Y_1, Y_2, Y_3) = FIRST(Y_1)$

(c) If  $FIRST(Y_1)$  contains  $\epsilon$ , then.

$FIRST(X) = FIRST(Y_1, Y_2, Y_3) = FIRST(Y_1) - \{\epsilon\} \cup FIRST(Y_2, Y_3)$

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  $\epsilon$ .

If  $FIRST(Y_2)$  contain  $\epsilon$ , then

$\text{FIRST}(Y_2, Y_3) = \text{FIRST}(Y_2) - \{\epsilon\} \cup \text{FIRST}(Y_3)$

Similarly, this method will be repeated for further Grammar symbols, i.e., for  $Y_4, Y_5, Y_6 \dots$  .  
YK.

### CODE:

```
#include<bits/stdc++.h>
using namespace std;
set<char> ss;
bool dfs(char i, char org, char last, map<char, vector<vector<char>>> &mp) {
    bool rtake = false;
    for (auto r : mp[i]) {
        bool take = true;
        for (auto s : r) {
            if (s == i) break;
            if (!take) break;
            if (!(s >= 'A' && s <= 'Z') && s != 'e') {
                ss.insert(s);
                break;
            }
            else if (s == 'e') {
                if (org == i || i == last)
                    ss.insert(s);
                rtake = true;
                break;
            }
            else {
                take = dfs(s, org, r[r.size() - 1], mp);
                rtake |= take;
            }
        }
    }
    return rtake;
}

int main() {
    int i, j;
    ifstream fin("input.txt");
    string num;
    vector<int> fs;
    vector<vector<int>> a;
    map<char, vector<vector<char>>> mp;
    char start;
    bool flag = 0;
```

```

cout << "Grammar: " << '\n';
while (getline(fin, num)) {
    if (flag == 0) start = num[0], flag = 1;
    cout << num << '\n';
    vector<char> temp;
    char s = num[0];
    for (i = 3; i < num.size(); i++) {
        if (num[i] == '|') {
            mp[s].push_back(temp);
            temp.clear();
        }
        else temp.push_back(num[i]);
    }
    mp[s].push_back(temp);
}
map<char, set<char>> fmp;
for (auto q : mp) {
    ss.clear();
    dfs(q.first, q.first, q.first, mp);
    for (auto g : ss) fmp[q.first].insert(g);
}
cout << '\n';
cout << "FIRST: " << '\n';
for (auto q : fmp) {
    string ans = "";
    ans += q.first;
    ans += " = { ";
    for (char r : q.second) {
        ans += r;
        ans += ' ';
    }
    ans.pop_back();
    ans += "}";
    cout << ans << '\n';
}

return 0;
}

```

## INPUT -

```
input.txt
S->ACB | CbB | Ba
A->da | BC
B->g | e
C->h | e
|
```

## OUTPUT -

```
PS C:\Users\NISHANT\Documents> .\h.exe
Grammar:
S->ACB | CbB | Ba
A->da | BC
B->g | e
C->h | e

FIRST:
A = {d,e,g,h}
B = {e,g}
C = {e,h}
S = {a,b,d,e,g,h}
```

## Program 5

**OBJECTIVE :** Write a Program to find the FOLLOW SET for given production of the grammar.

$S \rightarrow ACB / CbB / Ba$

$A \rightarrow da / BC$

$B \rightarrow g / \epsilon$

$C \rightarrow h / \epsilon$

**Theory :**

A terminal symbol  $a$  is in **FOLLOW (N)** if and only if there is a derivation from the start symbol  $S$  of the grammar such that  $S \Rightarrow \alpha N a \beta$ , where  $\alpha$  and  $\beta$  are a (possibly empty) sequence of grammar symbols. In other words, a terminal  $c$  is in FOLLOW (N) if  $c$  can follow  $N$  at some point in a derivation.

Benefit of FOLLOW ( )

- It can be used to prove the LL (K) characteristic of grammar.
- It can be used to promote the construction of predictive parsing tables.
- It provides selection information for recursive descent parsers.

**Algorithm :**

Computation of FOLLOW

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

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

Rules to find FOLLOW

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

(a) If FIRST ( $\beta$ ) does not contain  $\epsilon$  then, FOLLOW (B) = {FIRST ( $\beta$ )}

Or

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

$$\text{FOLLOW (B)} = \text{FIRST } (\beta) - \{\epsilon\} \cup \text{FOLLOW (A)}$$

$\therefore$  when  $\beta$  derives  $\epsilon$ , 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;
set<char> ss;
bool dfs(char i, char org, char last, map<char, vector<vector<char>>> &mp) {
    bool rtake = false;
    for (auto r : mp[i]) {
        bool take = true;
        for (auto s : r) {
            if (s == i) break;
            if (!take) break;
            if (!(s >= 'A' && s <= 'Z') && s != 'e') {
                ss.insert(s);
                break;
            }
            else if (s == 'e') {
                if (org == i || i == last)
                    ss.insert(s);
                rtake = true;
                break;
            }
            else {
                take = dfs(s, org, r[r.size() - 1], mp);
                rtake |= take;
            }
        }
    }
    return rtake;
}

int main() {
    int i, j;
    ifstream fin("input.txt");
    string num;
    vector<int> fs;
    vector<vector<int>> a;
    map<char, vector<vector<char>>> mp;
    char start;
    bool flag = 0;
    cout << "Grammar: " << "\n";
    while (getline(fin, num)) {
        if (flag == 0) start = num[0], flag = 1;
        cout << num << "\n";
        vector<char> temp;
        char s = num[0];
        for (i = 3; i < num.size(); i++) {
```



```

        if (num[i] == '|') {
            mp[s].push_back(temp);
            temp.clear();
        }
        else temp.push_back(num[i]);
    }
    mp[s].push_back(temp);
}
map<char, set<char>> fmp;
for (auto q : mp) {
    ss.clear();
    dfs(q.first, q.first, q.first, mp);
    for (auto g : ss) fmp[q.first].insert(g);
}

map<char, set<char>> gmp;
gmp[start].insert('$');
int count = 10;
while (count-->0) {
    for (auto q : mp) {
        for (auto r : q.second) {
            for (i = 0; i < r.size() - 1; i++) {
                if (r[i] >= 'A' && r[i] <= 'Z') {
                    if (!(r[i + 1] >= 'A' && r[i + 1] <= 'Z')) gmp[r[i]].insert(r[i + 1]);
                }
                else {
                    char temp = r[i + 1];
                    int j = i + 1;
                    while (temp >= 'A' && temp <= 'Z') {
                        if (*fmp[temp].begin() == 'e') {
                            for (auto g : fmp[temp]) {
                                if (g == 'e') continue;
                                gmp[r[i]].insert(g);
                            }
                        }
                        j++;
                        if (j < r.size()) {
                            temp = r[j];
                            if (!(temp >= 'A' && temp <= 'Z')) {
                                gmp[r[i]].insert(temp);
                                break;
                            }
                        }
                    }
                }
                else {
                    for (auto g : gmp[q.first]) gmp[r[i]].insert(g);
                    break;
                }
            }
        }
    }
}

```

```

        else {
            for (auto g : fmp[temp]) {
                gmp[r[i]].insert(g);
            }
            break;
        }
    }
}

if (r[r.size() - 1] >= 'A' && r[r.size() - 1] <= 'Z') {
    for (auto g : gmp[q.first]) gmp[r[i]].insert(g);
}
}
}

cout << "\n";
cout << "FOLLOW: " << "\n";
for (auto q : gmp) {
    string ans = "";
    ans += q.first;
    ans += " = { ";
    for (char r : q.second) {
        ans += r;
        ans += ',';
    }
    ans.pop_back();
    ans += " } ";
    cout << ans << "\n";
}
return 0;
}

```

## INPUT -

```
input.txt x
1 S->ACB | CbB | Ba
2 A->da | BC
3 B->g | e
4 C->h | e
5 |
```

## OUTPUT

```
PS C:\Users\NISHANT\Documents> .\h.exe
Grammar:
S->ACB | CbB | Ba
A->da | BC
B->g | e
C->h | e

FOLLOW:
A = {$, g, h}
B = {$, a, g, h}
C = {$, b, g, h}
S = {$}
PS C:\Users\NISHANT\Documents>
```

## Program -6

**AIM:** Write a program to implement NFA with Epsilon move to DFA.

### THEORY:

**Non-deterministic finite automata(NFA)** is a finite automata where for some cases when a specific input is given to the current state, the machine goes to multiple states or more than 1 state. It can contain  $\epsilon$  moves. It can be represented as  $M = \{Q, \Sigma, \delta, q_0, F\}$ .

Where

1.  $Q$ : finite set of states
2.  $\Sigma$ : finite set of the input symbol
3.  $q_0$ : initial state
4.  $F$ : **final** state
5.  $\delta$ : Transition function

**NFA with  $\epsilon$  move:** If any FA contains  $\epsilon$  transaction or move, the finite automaton is called NFA with  $\epsilon$  move.

**$\epsilon$ -closure:**  $\epsilon$ -closure for a given state A means a set of states which can be reached from state A with only  $\epsilon$ (null) move including the state A itself.

**Deterministic Finite Automata (DFA) :** DFA is a finite automata where, for all cases, when a single input is given to a single state, the machine goes to a single state, i.e., all the moves of the machine can be uniquely determined by the present state and the present input symbol.

### ALGORITHM:

**Step 1:** We will take the  $\epsilon$ -closure for the starting state of NFA as a starting state of DFA.

**Step 2:** Find the states for each input symbol that can be traversed from the present. That means the union of transition values and their closures for each state of NFA present in the current state of DFA.

**Step 3:** If we found a new state, take it as the current state and repeat step 2.

**Step 4:** Repeat Step 2 and Step 3 until there is no new state present in the transition table of DFA.

**Step 5:** Mark the states of DFA as a final state which contains the final state of NFA.

### CODE:

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    cout<<"\nINPUT"<<endl;
    int total_states,ways;
    cin>>total_states>>ways;
    vector<vector<vector<char>>>
matri(total_states,vector<vector<char>>>(ways,vector<char>(total_states)));
    map<int,string>arr,epsilon;
    map<char,int>arr1;
    int starting=0;
    char ch;

    for(int i=0;i<total_states;i++) {
```

```

        cin>>ch;
        arr[starting]=ch;
        arr1[ch]=starting;
        starting++;
    }
    for(int i=0;i<total_states;i++) {
        for(int j=0;j<ways;j++) {
            for(int k=0;k<total_states;k++) {
                cin>>matri[i][j][k];
            }
        }
    }
    cout<<"\n\n\nOUTPUT"<<endl;
    cout<<"\nE-NFA table"<<endl;
    for(int i=0;i<total_states;i++) {
        for(int j=0;j<ways;j++) {
            for(int k=0;k<total_states;k++)
                cout<<matri[i][j][k]<<" ";
            cout<<"|";
        }
        cout<<endl;
    }

    for(int i=0;i<total_states;i++) {
        epsilon[i]=arr[i];
        for(int j=0;j<total_states;j++) {
            if(matri[i][ways-1][j]!='0' && epsilon[i]!=to_string(matri[i][ways-1][j]))
                epsilon[i]+=matri[i][ways-1][j];
        }
    }

    for(int i=0;i<total_states;i++) {
        for(int j=1;j<epsilon[i].length();j++) {
            if(epsilon[arr1[epsilon[i][j]]].length()>1) {
                for(int k=1;k<epsilon[arr1[epsilon[i][j]]].length();k++) {
                    if(epsilon[i].find(epsilon[arr1[epsilon[i][j]]][k])== string::npos) {
                        epsilon[i]+=epsilon[arr1[epsilon[i][j]]][k];
                    }
                }
            }
        }
    }

    cout<<"Epsilons"<<endl;
    for(int i=0;i<total_states;i++)
        cout<<arr[i]<<" ---> "<<epsilon[i]<<endl;

```

```

map<int,map<int,string>>>newstatematrix;

int power=pow(2,total_states);
vector<string>arrofnewstates(power);
arrofnewstates[0]=epsilon[0];

map<string,int>newstates;
newstates[epsilon[0]]=1;

vector<char>eachstate;
int start=0,end=1;

while(start<end) {
    for(int x=0;x<ways-1;x++) {
        for(int i=0;i<arrofnewstates[start].length();i++) {
            if(matri[arr1[arrofnewstates[start][i]]][x][0]!='0') {
                for(int j=0;j<total_states;j++) {
                    if(matri[arr1[arrofnewstates[start][i]]][x][j]!='0')
                        eachstate.push_back(matri[arr1[arrofnewstates[start][i]]][x][j]);
                    else break;
                }
            }
        }
    }

    for(int l=0;l<eachstate.size();l++) {
        for(int k=0;k<epsilon[arr1[eachstate[l]]].length();k++)
            if(newstatematrix[start][x].find(epsilon[arr1[eachstate[l]]][k])==string::npos)
                newstatematrix[start][x]+=epsilon[arr1[eachstate[l]]][k];
    }

    eachstate.clear();
    if(newstates[newstatematrix[start][x]]!=1) {
        arrofnewstates[end]=newstatematrix[start][x];
        end++;
    }
    newstates[newstatematrix[start][x]]=1;
}
start++;
}
cout<<"DFA table"<<endl;
for(int i=0;i<newstates.size();i++) {
    cout<<arrofnewstates[i]<<" ---> ";
    for(int j=0;j<ways-1;j++) {
        cout<<newstatematrix[i][j]<<" |";
    }
}

```

```

        cout<<endl;
    }
    cout<<endl;
    return 0;
}

```

### **INPUT / OUTPUT:**

```

INPUT
3 3
A B C
B C 0
A 0 0
B 0 0
0 0 0
B 0 0
C 0 0
C 0 0
C 0 0
0 0 0

OUTPUT

E-NFA table
B C 0 | A 0 0 | B 0 0 |
0 0 0 | B 0 0 | C 0 0 |
C 0 0 | C 0 0 | 0 0 0 |
Epsilons
A ---> ABC
B ---> BC
C ---> C
DFA table
ABC ---> BC | ABC |
BC ---> C | BC |
C ---> C | C |

```

**RESULT:** Hence, we implemented the method to convert an  $\epsilon$ -NFA to DFA.

## Program-7

**AIM:** Write a program for the given grammar G of a fictitious language L and parse the input string  $w=id+id*id$ , using Recursive Descent Parsing Algorithm.

### **THEORY:**

Recursive descent Parser 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 non-terminal entity. This parsing technique recursively parses the input to make a parse tree, which may or may not require back-tracking. But the grammar associated with it (if not left factored) cannot avoid back-tracking.

### **ALGORITHM:**

1. Start from the input symbol
2. For each non terminal say S
  - a. Call Procedure(S)
  - b. for  $i=1$  to  $k$ 
    - i. choose production  $S \rightarrow A_1 A_2 \dots A_k$
    - ii. If  $A_i$  is a non terminal then call Procedure( $A_i$ )
    - iii. else if  $A_i$  is a terminal
      - If  $A_i$  is same as the current input symbol then break from current loop of  $A_i$  and advance the pointer of the current input symbol
      - else display "Syntax Error"

### **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";
}

```

## **INPUT:**

```
cd_program > cd > parser > recursive.txt  
1      i+i*i$
```

## **OUTPUT:**

```
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'=*FT' production is used  
F=i production is used  
T'={ production is used  
E'={ production is used  
Parsing Successful
```

**RESULT:** We successfully parsed the string using recursive descent parsing algorithm. 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.

## Program-8

**AIM:** Write a program to convert left recursive grammar to its equivalent right recursive grammar.

### **THEORY:**

Left recursion is eliminated by converting the grammar into a right recursive grammar.

If we have the left-recursive pair of productions

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

(Left Recursive Grammar)

where  $\beta$  does not begin with an A.

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

$$\text{With } A \rightarrow \beta A' \quad A' \rightarrow \alpha A' / \epsilon$$

(Right Recursive Grammar)

This right recursive grammar functions the same as left recursive grammar.

### **ALGORITHM:**

$$A1 \rightarrow A2 A3$$

$$A2 \rightarrow A3 A1 / b$$

$$A3 \rightarrow A1 A1 / a$$

Where A1, A2, A3 are non terminals and a, b are terminals.

1. Identify the productions which can cause indirect left recursion. In our case,  $A3 \rightarrow A1 A1 / a$
2. Substitute its production at the place the terminal is present in any other production substitute  $A1 \rightarrow A2 A3$  in production of A3.  $A3 \rightarrow A2 A3 A1$ . Now in this production substitute  $A2 \rightarrow A3 A1 / b$  and then replace this by,  $A3 \rightarrow A3 A1 A3 A1 / b A3 A1$
3. Now the new production is converted in form of direct left recursion, solve this by direct left recursion method. Eliminating direct left recursion in the above,
4.  $A3 \rightarrow a \mid b A3 A1 \mid aA' \mid b A3 A1A'$   
 $A' \rightarrow A1 A3 A1 \mid A1 A3 A1A'$

The resulting grammar is then:

$$A1 \rightarrow A2 A3$$

$$A2 \rightarrow A3 A1 \mid b$$

$$A3 \rightarrow a \mid b A3 A1 \mid aA' \mid b A3 A1A'$$

$$A' \rightarrow A1 A3 A1 \mid A1 A3 A1A'$$

### **CODE:**

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

```
using namespace std;
```

```
int main(){  
    cout<<"Enter Productions -\n";  
    int n;  
    cin>> n;
```

```

map<string, vector<string>> p, ans;
for (int i = 0; i < n; i++) {
    string a;
    int j;
    cin>>a>>j;

    for (int j_ = 0; j_ < j; j_++) {
        string b;
        cin>>b;
        p[a].push_back(b);
    }
}

set<string>processed;
for (auto i : p) {
    if (i.first.size() > 1) continue;

    bool ch = true;
    while(ch) {
        ch = false;
        vector<string> v;
        for (auto j : i.second) {
            string f = "";
            f += j[0];

            if (processed.find(f) != processed.end()) {
                ch = true;
                for (auto k : ans[f])
                    v.push_back(k + j.substr(1, j.size() - 1));
            } else v.push_back(j);
        }
        i.second = v;
    }

    processed.insert(i.first);
    vector<string> same, diff;
    for (auto j : i.second) {
        if (j[0] == i.first[0])
            same.push_back(j);
        else diff.push_back(j);
    }

    string x = i.first + "";
    for (auto &j : same) {
        string s = j.substr(1, j.size() - 1);
    }
}

```

```

        s += x;
        j = s;
    }

    for (auto &j : diff) {
        if (j[0] == '#')
            j = x;
        else j += x;
    }

    same.push_back("#");
    ans[i.first] = diff;
    ans[x] = same;
}

cout<<"\nAfter removing left recursion-\n";
for (auto i : ans){
    cout<<i.first<<" -> ";
    int n_ = i.second.size();
    for (int j = 0; j < n_-1; ++j)
        cout<<i.second[j]<<" / ";
    cout<<i.second[n_-1]<<"\n";
}
return 0;
}

```

### **INPUT/OUTPUT:**

```

Enter Productions -
1
S 2 S0S1S 01

After removing left recursion-
S -> 01S'
S' -> 0S1SS' / #

```

**RESULT:** 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.

## Program-9

**AIM:** Write a program to left factor the grammar.

### **THEORY:**

Left factoring is a process by which the grammar with common prefixes is transformed to make it useful for Top down parsers.

Grammar with common prefix :

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

Converted into: Grammar with not common prefix:

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

### **ALGORITHM:**

For a Grammar G equivalent left factored grammar can be given as:

1. For each non terminal A find the longest prefix  $\alpha$  common to two or more of its alternatives.
2. If  $\alpha \neq \epsilon$ , i. e., there is a non trivial common prefix, replace all the A productions.

### **CODE:**

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

```
using namespace std;
```

```
int main() {
    cout<<"\nEnter number of productions: ";
    int p, i;
    cin>>p;
    vector<string> pleft(p),pright(p);
    cout<<"Enter productions one by one:\n";
    for(i = 0; i < p; ++i) {
        cout<<"Left of production "<<i + 1<<": ";
        cin>>pleft[i];
        cout<<"Right of production "<<i + 1<<": ";
        cin>>pright[i];
    }
    int j, e = 1;
    for(i = 0; i < p; ++i) {
        for(j = i + 1; j < p; ++j) {
            if(pleft[j] == pleft[i]) {
                int k = 0;
                string com = "";
                while(k < pright[i].length() && k < pright[j].length() && pright[i][k] == pright[j][k]) {
                    com += pright[i][k];
                    ++k;
                }
                if(k == 0)
```

```

        continue;
    char* buffer;
    string comleft = pleft[i];
    if(k == pright[i].length()) {
        pleft[i] += string(itoa(e,buffer,10));
        pleft[j] += string(itoa(e,buffer,10));
        pright[i] = "^";
        pright[j] = pright[j].substr(k,pright[j].length()-k);
    } else if(k == pright[j].length()) {
        pleft[i] += string(itoa(e,buffer,10));
        pleft[j] += string(itoa(e,buffer,10));
        pright[j] = "^";
        pright[i] = pright[i].substr(k,pright[i].length()-k);
    } else {
        pleft[i] += string(itoa(e,buffer,10));
        pleft[j] += string(itoa(e,buffer,10));
        pright[j] = pright[j].substr(k,pright[j].length()-k);
        pright[i] = pright[i].substr(k,pright[i].length()-k);
    }
    int l;
    for(l = j + 1; l < p; ++l) {
        if(comleft == pleft[l] && com == pright[l].substr(0,fmin(k,pright[l].length())) {
            pleft[l] += string(itoa(e,buffer,10));
            pright[l] = pright[l].substr(k,pright[l].length()-k);
        }
    }
    pleft.push_back(comleft);
    pright.push_back(com+pleft[i]);
    ++p;
    ++e;
}
}
}
cout<<"\nNew productions";
for(i = p - 1; i >= 0; --i)
    cout<<"\n"<<pleft[i]<<"->"<<pright[i];
return 0;
}

```

## INPUT/OUTPUT

```
Enter number of productions: 3
Enter productions one by one:
Left of production 1: S
Right of production 1: itaC
Left of production 2: S
Right of production 2: it
Left of production 3: C
Right of production 3: ab

New productions
S->itS1
C->ab
S1->^
S1->aC
```

**RESULT:** 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.



## Program-10

**AIM:** Write a program to parse the string id=num-num using LL(1) predictive parser.

### **THEORY:**

LL(1) parsing is a top-down parsing method in the syntax analysis phase of compiler design. Required components for LL(1) parsing are input string, a stack, parsing table for given grammar, and parser. It is a parser that determines that given string can be generated from a given grammar(or parsing table) or not.

Let given grammar is  $G = (V, T, S, P)$  where V-variable symbol set, T-terminal symbol set, S-start symbol, P- production set.

LL(1) parser is a non recursive decent parser. In LL(1) parser the 1<sup>st</sup> L represents that the scanning of the Input will be done from Left to Right manner and the second L shows that in this parsing technique we are going to use Left most Derivation Tree. And finally, the 1 represents the number of look-ahead, which means how many symbols are you going to see when you want to make a decision.

### **ALGORITHM:**

1. First check for left recursion in the grammar, if there is left recursion in the grammar remove that and go to step 2.
2. Calculate First() and Follow() for all non-terminals.
  - i. 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.
  - ii. Follow(): What is the Terminal Symbol which follows a variable in the process of derivation.
3. For each production  $A \rightarrow \alpha$ . (A tends to alpha)
  - i. Find First( $\alpha$ ) and for each terminal in First( $\alpha$ ), make entry  $A \rightarrow \alpha$  in the table.
  - ii. If First( $\alpha$ ) contains  $\epsilon$  (epsilon) as terminal than, find the Follow(A) and for each terminal in Follow(A), make entry  $A \rightarrow \alpha$  in the table.

If the First( $\alpha$ ) contains  $\epsilon$  and Follow(A) contains \$ as terminal, then make entry  $A \rightarrow \alpha$  in the table for the \$.

### **CODE:**

```
#include <bits/stdc++.h>
using namespace std;
map <string,string> first, follow, rules;
set <string> nt, t;
vector <string> calls_for_nt;
map < pair<string,string> , string > parse_table;
vector <string> stck, cur_str, rule_used;
string to_string(char c) {
    string s="";
    s+=c;
    return s;
}
void set_map(string s) {
```

```

    stringstream ss(s);
    string key,value,temp;
    ss>>key;
    calls_for_nt.push_back(key);
    while(ss>>temp){
        if(temp!="->"&&temp!="|")
            value+=" "+temp;
    }
    rules[key]=value;
    return ;
}

bool is_nterminal(string s) {
    if(find(nt.begin(),nt.end(),s)!=nt.end())
        return true;
    else return false;
}

bool is_terminal(string s) {
    if(find(t.begin(),t.end(),s)!=t.end())
        return true;
    else return false;
}

void set_first(string s) {
    if(first[s].length()!=0)
        return ;
    string temp=rules[s].substr(1,rules[s].length()-1);
    stringstream ss(temp);
    while(ss>>temp) {
        if(is_nterminal(to_string(temp[0]))){
            set_first(to_string(temp[0]));
            first[s]=first[to_string(temp[0])];
        }
        else first[s]+=temp.substr(0,1)+" ";
    }
    return;
}

bool check_dol(string s) {
    for(int i=0;i<s.length();i++)
        if(s[i]=='!')
            return true;
    return false;
}

void set_follow(string s) {
    int flag=0;
    for(auto itr=rules.begin();itr!=rules.end();itr++) {
        stringstream ss(itr->second);
        string temp;

```

```

while(ss>>temp) {
    for(int i=0;i<temp.length();i++) {
        flag=0;
        if(temp[i]==s[0]) {
            if(i+1<temp.length()) {
                if(is_terminal(to_string(temp[i+1]))){
                    follow[s]+="" +to_string(temp[i+1])+"";
                } else {
                    if(check_dol(rules[to_string(temp[i+1]))]){
                        for(int j=0;j<first[to_string(temp[i+1])].length();j++)
                            if(first[to_string(temp[i+1])][j]!='!')
                                follow[s]+="" +to_string(first[to_string(temp[i+1])][j]);
                        follow[s]+=" ";
                    }
                    follow[s]+="" +follow[to_string(temp[i+1])];
                }
            } else follow[s]+="" +follow[itr->first]+" ";
            flag=1;
            break;
        }
    }
    if(flag==1) break;
}
}

string in_rules(string s, string t) {
    if(find(rules[s].begin(),rules[s].end(),t[0])!=rules[s].end()){
        stringstream ss(rules[s]);
        string temp;
        while(ss>>temp)
            if(find(temp.begin(),temp.end(),t[0])!=temp.end())
                return temp;
    } else{
        stringstream ss(rules[s]);
        string temp;
        ss>>temp;
        return temp;
    }
    return t;
}

void set_parse_table(string s) {
    string temp=first[s];
    if(find(temp.begin(),temp.end(),'!')!=temp.end()){
        string for_dol;
        stringstream ss(follow[s]);
        while(ss>>for_dol)

```

```

        parse_table[{s,for_dol}]=s+"-> !";
    }
    stringstream ss(temp);
    while(ss>>temp){
        if(temp==to_string('!'))
            continue;
        parse_table[{s,temp}]=s+"-> "+in_rules(s,temp);
    }
    return ;
}

void check_str() {
    if(stck[stck.size()-1][0]=='$' && cur_str[cur_str.size()-1][0]=='$')
        return ;
    if(stck[stck.size()-1][0]==cur_str[cur_str.size()-1][0]) {
        stck.push_back(stck[stck.size()-1].substr(1,stck[stck.size()-1].length()-1));
        cur_str.push_back(cur_str[cur_str.size()-1].substr(1,cur_str[cur_str.size()-1].length()-1));
        rule_used.push_back(" ");
        check_str();
    }else if(parse_table[{to_string(stck[stck.size()-1][0]),to_string(cur_str[cur_str.size()-1][0])}].length()!=0){
        stringstream ss(parse_table[{to_string(stck[stck.size()-1][0]),to_string(cur_str[cur_str.size()-1][0])}]);
        string temp;
        ss>>temp;ss>>temp;
        if(temp=="!") {
            stck.push_back(stck[stck.size()-1].substr(1,stck[stck.size()-1].length()-1));
            cur_str.push_back(cur_str[cur_str.size()-1]);
            rule_used.push_back(parse_table[{to_string(stck[stck.size()-1][0]),to_string(cur_str[cur_str.size()-1][0])}]);
        } else {
            stck.push_back(temp+stck[stck.size()-1].substr(1,stck[stck.size()-1].length()-1));
            cur_str.push_back(cur_str[cur_str.size()-1]);
            rule_used.push_back(parse_table[{to_string(stck[stck.size()-1][0]),to_string(cur_str[cur_str.size()-1][0])}]);
        }
        check_str();
    }
    else return;
}

int main() {
    int n;
    cout<<"HOW MANY RULES ARE THERE ? ";
    cin>>n;
    cout<<"ENTER THEM ONE BY ONE : "<<endl;
    string s;

```

```

getline(cin,s);
for(int i=0;i<n;i++) {
    getline(cin,s);
    set_map(s);
}
cout<<"MAP IS : \n";
for(auto itr=rules.begin();itr!=rules.end();itr++)
    cout<<itr->first<<" "<<itr->second<<endl;
for(auto itr=rules.begin();itr!=rules.end();itr++)
    nt.insert(itr->first);
for(auto itr=rules.begin();itr!=rules.end();itr++){
    stringstream ss(itr->second);
    string test;
    while(ss>>test) {
        for(int i=0;i<test.length();i++)
            if(!is_terminal(test.substr(i,1))&&(test.substr(i,1))!="!")
                t.insert(test.substr(i,1));
    }
}
cout<<"NON-TERMINALS : "<<endl;
for(string s:nt)
    cout<<s<<" ";
cout<<"\nTERMINALS: "<<endl;
for(string s:t)
    cout<<s<<" ";
for(string s:nt)
    set_first(s);
follow.clear();
for(string s:calls_for_nt)
    set_follow(s);
for(auto itr=follow.begin();itr!=follow.end();itr++)
    itr->second+=" $";
cout<<"\nALL FIRST'S : "<<endl;
for(auto itr=first.begin();itr!=first.end();itr++)
    cout<<itr->first<<" -> "<<itr->second<<endl;
cout<<"\nALL FOLLOW'S : "<<endl;
for(auto itr=follow.begin();itr!=follow.end();itr++)
    cout<<itr->first<<" -> "<<itr->second<<endl;
for(string s:calls_for_nt)
    set_parse_table(s);
cout<<"\n PARSE TABLE CONTENTS : "<<endl<<endl;
cout<<"NT\t";
for(string s:t)
    cout<<s<<"\t";
cout<<"$\t";
cout<<endl;

```

```

for(int i=0;i<calls_for_nt.size();i++) {
    cout<<calls_for_nt[i]<<"\t";
    for(string ts:t)
        cout<<parse_table[{ calls_for_nt[i],ts}]<<"\t";
    cout<<parse_table[{ calls_for_nt[i],"$"}]<<"\t";
    cout<<endl;
}
stk.push_back(calls_for_nt[0]+"$");
cout<<"\n\nENTER STRING TO CHECK : ";
getline(cin,s);
cur_str.push_back(s+"$");
rule_used.push_back(" ");
check_str();
cout<<"\n\nstack\t\tINPUT\t\tPRODUCTION\n";
for(int i=0;i<stk.size();i++)
    cout<<stk[i]<<"\t"<<cur_str[i]<<"\t"<<rule_used[i]<<endl;
if(stk[stk.size()-1][0]=='$' && cur_str[cur_str.size()-1][0]=='$')
    cout<<"\n\nSTRING IS ACCEPTED !!!";
else
    cout<<"\n\nSTRING IS NOT ACCEPTED !!!";
return 0;
}

```

## INPUT/OUTPUT:

```

HOW MANY RULES ARE THERE ? 4
ENTER THEM ONE BY ONE :
S -> aB
B -> =A
A -> E-E
E -> b
MAP IS :
A E-E
B =A
E b
S aB
NON-TERMINALS :
A B E S
TERMINALS:
- = a b
ALL FIRST'S :
A -> b
B -> =
E -> b
S -> a

ALL FOLLOW'S :
A -> $
B -> $
E -> - $
S -> $

```

```

PARSE TABLE CONTENTS :

NT      -      =      a      b      $
S
B      B-> =A
A      A-> E-E
E      E-> b

ENTER STRING TO CHECK : a=b-b

stack      INPUT      PRODUCTION
S$          a=b-b$
aB$         a=b-b$
B$          =b-b$
=A$         =b-b$
A$          b-b$
E-E$        b-b$      E-> b
b-E$        b-b$
-E$         -b$
E$          b$
b$          b$
$           $

STRING IS ACCEPTED !!!

```

**RESULT:** We have parsed string id = num – num using the LL(1) parser. id was replaced by ‘a’ and num was replaced by ‘b’ for the code. And for grammar  $S \rightarrow aB$ ,  $B \rightarrow =A$ ,  $A \rightarrow E-E$ ,  $E \rightarrow b$  the string was accepted.

# Program-11

**AIM:** Write a program to obtain output using lex and yacc.

## **THEORY:**

Lex is a program that generates a lexical analyzer. It is used with the YACC parser generator. The lexical analyzer is a program that transforms an input stream into a sequence of tokens. It reads the input stream and produces the source code as output by implementing the lexical analyzer in the C program. YACC (yet another compiler-compiler) is an LALR(1) (LookAhead, Left-to-right, Rightmost derivation producer with 1 lookahead token) parser generator. YACC was originally designed for being complemented by Lex.

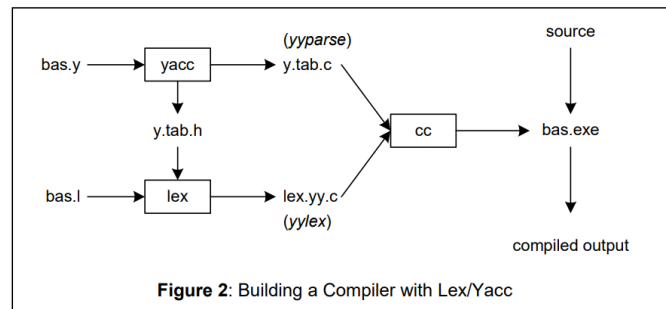


Figure 2: Building a Compiler with Lex/Yacc

## **ALGORITHM:**

1. A Yacc source program has three parts as follows:  
Declarations %% translation rules %% supporting C routines
2. Declarations Section: This section contains entries that:
  - i. Include standard I/O header file.
  - ii. Define global variables.
  - iii. Define the list rule as the place to start processing.
  - iv. Define the tokens used by the parser.
  - v. Define the operators and their precedence.
3. Rules Section: The rules section defines the rules that parse the input stream. Each rule of a grammar production and the associated semantic action.
4. Programs Section: The programs section contains the following subroutines. Because these subroutines are included in this file, it is not necessary to use the yacc library when processing this file.
5. Main- The required main program that calls the `yyparse` subroutine to start the program.
6. `yyerror(s)` -This error-handling subroutine only prints a syntax error message.
7. `yywrap` -The wrap-up subroutine that returns a value of 1 when the end of input occurs. The `calc.lex` file includes statements for standard input and output, as programmer file information if we use the `-d` flag with the yacc command. The `y.tab.h` file contains definitions for the tokens that the parser program uses.
8. `calc.lex` contains the rules to generate these tokens from the input stream.



## **CODE:**

//Implementation of calculator using LEX and YACC

// LEX PART:

```
% {
#include<stdio.h>
#include "y.tab.h"
extern int yylval;
% }

%%

[0-9]+ {
    yylval=atoi(yytext);
    return NUMBER;
}
[\t] ;
[\n] return 0;
. return yytext[0];
%%
```

```
int yywrap()
{
    return 1;
}
```

// YACC PART:

```
% {
    #include<stdio.h>
    int flag=0;
% }
```

```
%token NUMBER
%left '+' '-'
%left '*' '/' '%'
%left '(' ')'
%%
```

```
ArithmeticExpression: E{
    printf("\nResult=%d\n",$$);
    return 0;
};
```

```
E:E'+E {$$=$1+$3;}
|E'-E {$$=$1-$3;}
|E'*E {$$=$1*$3;}
|E'/E {$$=$1/$3;}
|E'%E {$$=$1%$3;}
```

```

|('E')' {$$=$2;}
| NUMBER {$$=$1;}
;
%%

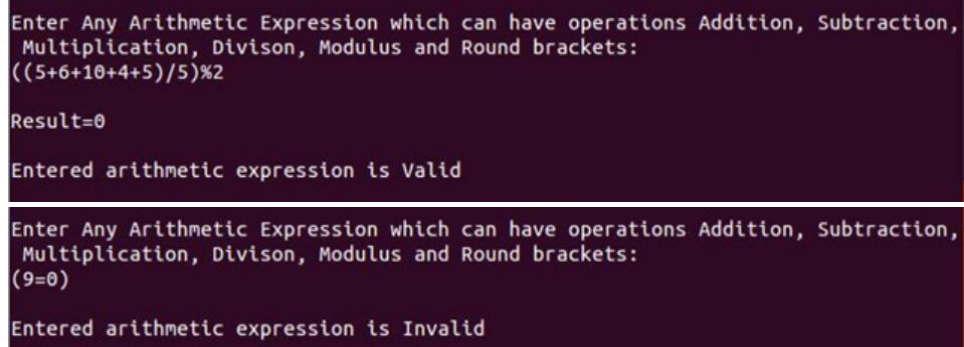
void main()
{
    printf("\nEnter Any Arithmetic Expression which can have operations Addition, Subtraction,
    Multiplication, Divison, Modulus and Round brackets:\n");

    yyparse();
    if(flag==0)
        printf("\nEnter arithmetic expression is Valid\n\n");
    }

void yyerror()
{
    printf("\nEnter arithmetic expression is Invalid\n\n");
    flag=1;
}

```

### **INPUT/OUTPUT:**



```

Enter Any Arithmetic Expression which can have operations Addition, Subtraction,
Multiplication, Divison, Modulus and Round brackets:
((5+6+10+4+5)/5)%2

Result=0

Entered arithmetic expression is Valid

Enter Any Arithmetic Expression which can have operations Addition, Subtraction,
Multiplication, Divison, Modulus and Round brackets:
(9=0)

Entered arithmetic expression is Invalid

```

**RESULT:** Here we implemented a simple calculator using LEX and YACC and got the output for the expression and checked if the arithmetic expression is valid or not and clearly defined all the parts in the code.

## Program-12

**AIM:** Write a program to generate the intermediate representation (IR code) using Syntax Directed Translation. The input will be 4-address code and output generated will be 3-address code.

### THEORY:

In computer science, three-address code (often abbreviated to TAC or 3AC) is a form of representing intermediate code used by compilers to aid in the implementation of code-improving transformations. Each instruction in three-address code can be described as a 4-tuple: (operator, operand1, operand2, result).

Each statement has the general form of: such as:

Result = operand1 operator operand 2 example  $x = y + z$ .

where x, y and z are variables, constants or temporary variables generated by the compiler. op represents any operator, e.g. an arithmetic operator.

Expressions containing more than one fundamental operation, such as:  $p = x + y * z$  are not representable in three-address code as a single instruction. Instead, they are decomposed into an equivalent series of instructions, such as  $t1 = y * z$ ,  $p = x + t1$ .

The term three-address code is still used even if some instructions use more or fewer than two operands. The key features of three address codes are that every instruction implements exactly one fundamental operation, and that the source and destination may refer to any available register. A refinement of three-address code is static single assignment form (SSA).

### ALGORITHM:

1. When three-address code is generated, temporary names are made up for the interior nodes of a syntax tree.
2. The value of nonterminal E on the left side of  $E \rightarrow E1 + E2$  will be computed into a temporary t.
3. The nonterminal E has two attributes:
  - i. E.place: a name to hold the value of E at runtime
  - ii. E.code: the sequence of 3AC statements implementing E.
4. Temporary names are associated for interior nodes of the syntax tree.
5. The function newtemp() returns a fresh temporary name on each invocation. It means that return a sequence of distinct names t1, t2, t3,... in response to successive calls.

### CODE:

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

```
void pm();
void plus();
void div();
int i, ch, j, l;
char ex[10], ex1[10], exp1[10], ex2[10];
void strrev(char str1[])
```

```

{
    int I=0;
    int size=0;
    while(str1[I]!='\0')
    {
        size++;
    }
    int II=0;
    int JJ=size-1;
    while(II<JJ){
        char temp;
        temp = str1[II];
        str1[II] = str1[JJ];
        str1[JJ] = temp;
        II++;
        JJ--;}
}

```

```

int main() {
    while (1) {
        printf("\n 1.Assignment\n 2.Arithmetic\n 3.exit\n ENTER THE CHOICE:");
        scanf("%d", & ch);
        switch (ch) {
            case 1:{
                printf("\n enter the expression with assignment operator:");
                scanf("%s", ex1);
                l = strlen(ex1);
                ex2[0] = '\0';
                i = 0;
                while (ex1[i] != '=') i++;
                strncat(ex2, ex1, i);
                strrev(ex1);
                exp1[0] = '\0';
                strncat(exp1, ex1, l - (i + 1));
                strrev(exp1);
                printf("3 address code:\n temp=%s \n %s=temp\n", exp1, ex2);
                break;
            }
}

```

```

case 2:{
    printf("\n enter the expression with arithmetic operator:");
    scanf("%s", ex);
    strcpy(ex1, ex);
    l = strlen(ex1);
    exp1[0] = '\0';
    for (i = 0; i < l; i++) {

```

```

    if (ex1[i] == '+' || ex1[i] == '-') {
        if (ex1[i + 2] == '/' || ex1[i + 2] == '*') {
            pm();
            break;
        } else {
            plus();
            break;
        }
    } else if (ex1[i] == '/' || ex1[i] == '*') {
        div();
        break;
    }
}
break;
}
case 3: break;
}
break;
}
}
void pm(){
    strrev(exp1);
    j = l - i - 1;
    strncat(exp1, ex1, j);
    strrev(exp1);
    printf("3 address code:\n temp=%s\n temp1=temp%c%c temp\n", exp1, ex1[j + 2], ex1[j]);
}
void div() {
    strncat(exp1, ex1, i + 2);
    printf("3 address code:\n temp=%s\n temp1=temp%c%c\n", exp1, ex1[l + 2], ex1[i + 3]);
}
void plus() {
    strncat(exp1, ex1, i + 2);
    printf("3 address code:\n temp=%s\n temp1=temp%c%c\n", exp1, ex1[l + 2], ex1[i + 3]);
}

```

## **INPUT/OUTPUT:**

```
1.Assignment
2.Arithmetic
3.Exit
Enter the choice:1
Enter the exp with assignment operator: a=b
3 address code:
temp=b
a=temp
1.Assignment
2.Arithmetic
3.Exit
Enter the choice:2
Enter the exp with arithmetic operator: a*b+c
3 address code:
temp =a*b
temp1=temp+c
1.Assignment
2.Arithmetic
3.Exit
Enter the choice:3
```

**RESULT:** We have successfully generated intermediate representation (IR code) using Syntax Directed Translation which is a 3-address code output for 4-address code input.