

# **Government College of Engineering, Jalgaon**

**(An Autonomous Institute of Govt of Maharashtra)**

**Department of Computer Engineering**



**Lab Manual For CO456 Compiler Construction Lab**

**L. Y. B. Tech.(Computer)**

**Semester: ODD**



# **Government College of Engineering, Jalgaon**

## **(An Autonomous Institute of Govt. of Maharashtra)**

**Name of Student:**.....

**PRN:**.....

**Date of Performance:**.....

**Date of Completion:**.....

### **Experiment No 1**

**Title:** Design a lexical analyzer for given language and the lexical analyzer should ignore redundant spaces, tabs and new lines

#### **Theory:**

Designing a lexical analyzer for a given language involves the identification and extraction of meaningful tokens from the source code while ignoring redundant spaces, tabs, and new lines. Here's the theory for designing such a lexical analyzer:

##### **Language Definition:**

Before designing a lexical analyzer, you need to have a clear understanding of the language's syntax and the types of tokens it supports. This includes keywords, identifiers, literals, operators, and special symbols.

##### **Character Stream:**

The lexical analyzer reads the source code as a character stream. It processes each character in the source code one at a time.

##### **Tokenization:**

Tokenization is the process of breaking the character stream into tokens. Tokens are the smallest meaningful units in the source code. These tokens are classified into different categories, such as keywords, identifiers, literals, operators, and special symbols.

##### **Handling Whitespace:**

To ignore redundant spaces, tabs, and new lines, the lexical analyzer can skip over these characters as they are encountered in the character stream. You can implement this by:

- Defining a set of whitespace characters (e.g., space, tab, newline).
- When a whitespace character is encountered, continue reading characters until a non-whitespace character is found.

##### **Token Classification:**

The lexical analyzer identifies and classifies tokens based on language-specific rules. For example:

- Identifiers: A sequence of letters and digits, often starting with a letter.
- Keywords: Reserved words with special meanings in the language.
- Literals: Constants, such as integers, floating-point numbers, and strings.
- Operators: Arithmetic operators (+, -, \*, /), comparison operators (==, !=), assignment operators (=), etc.
- Special Symbols: Parentheses, braces, semicolons, commas, etc.

### Lexical Rules:

Define lexical rules for recognizing tokens. These rules are usually expressed using regular expressions or finite automata.

- For example, a regular expression for identifying integers in C might be: `^[0-9]+``.
- Regular expressions can be used to match identifiers, keywords, and other token types.

### Error Handling:

The lexical analyzer should handle errors gracefully. When it encounters characters that don't match any expected tokens, it should report an error.

### Token Output:

As tokens are recognized, they can be output along with their corresponding values (lexemes). The output can be in the form of a token stream, which is passed to the parser for further processing.

### Implementation:

The lexical analyzer can be implemented using programming languages like C, C++, Python, or tools like Lex (in combination with Yacc/Bison). The choice of tools and programming language depends on the specific requirements of the language being analyzed.

In summary, a lexical analyzer for a given language identifies and extracts meaningful tokens while ignoring redundant spaces, tabs, and new lines by defining rules for tokenization and handling whitespace characters. It is a crucial component in the compilation process of programming languages.

### Program:

```
/*
Assignment name: Design a lexical analyzer for given language and the lexical analyzer should ignore
redundant spaces, tabs and new lines.
*/
#include<stdio.h>
#include<stdlib.h>
#include<ctype.h>
#include<string.h>
void keyw(char *p);
int i=0,id=0,kw=0,num=0,op=0;
char keys[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 main()
{
    char ch,str[25],seps[15]=" \t\n,;(){}[]#\"<> ",oper[]="!%^&*-+=~|.<>/?";
    int j;
    char fname[50];
    FILE *f1;
    printf("enter file path (drive:\\fold\\filename)\\n");
    scanf("%s",fname);
    f1 = fopen(fname,"r");
    if(f1==NULL)
```

```

{
    printf("file not found");
    exit(0);
}
while((ch=fgetc(f1))!=EOF)
{
    for(j=0;j<=14;j++)
    {
        if(ch==oper[j])
        {
            printf("%c is an operator\n",ch);
            op++;
            str[i]='\0';
            keyw(str);
        }
    }
    for(j=0;j<=14;j++)
    {
        if(i==-1)
            break;
        if(ch==seps[j])
        {
            if(ch=='#')
            {
                while(ch!='>')
                {
                    printf("%c",ch);
                    ch=fgetc(f1);
                }
                printf("%c is a header file\n",ch);
                i=-1;
                break;
            }
            if(ch=="")
            {
                do
                {
                    ch=fgetc(f1);
                    printf("%c",ch);
                }while(ch!="");
                printf("\b is an argument\n");
                i=-1;
                break;
            }
            str[i]='\0';
            keyw(str);
        }
    }
    if(i!=-1)
    {
        str[i]=ch;
        i++;
    }
    else

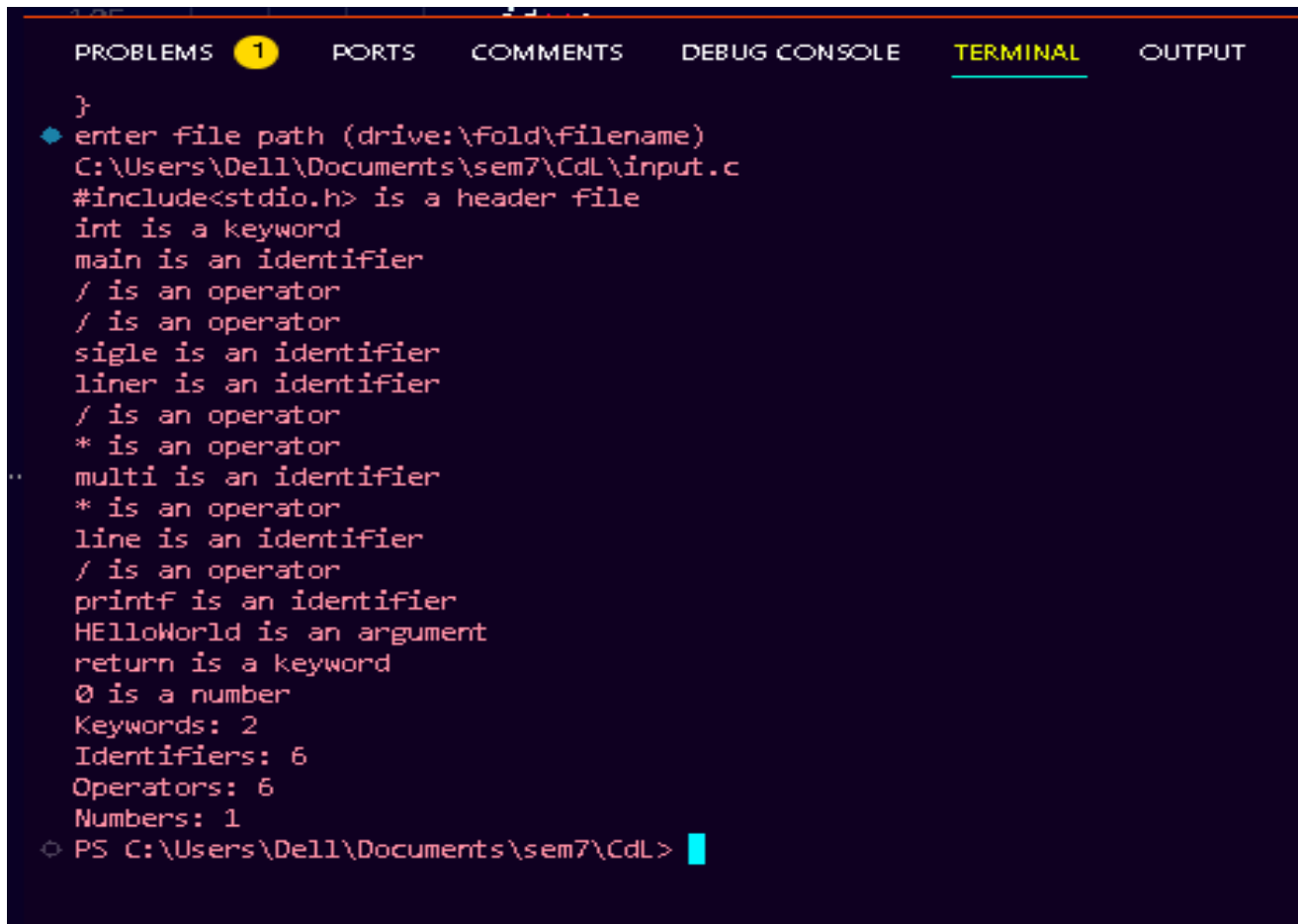
```

```

        i=0;
    }
    printf("Keywords: %d\nIdentifiers: %d\nOperators: %d\nNumbers: %d\n",kw,id,op,num);
}
void keyw(char *p)
{
    int k,flag=0;
    for(k=0;k<=31;k++)
    {
        if(strcmp(keys[k],p)==0)
        {
            printf("%s is a keyword\n",p);
            kw++;
            flag=1;
            break;
        }
    }
    if(flag==0)
    {
        if(isdigit(p[0]))
        {
            printf("%s is a number\n",p);
            num++;
        }
        else
        {
            if(p[0]!='\0')
            {
                printf("%s is an identifier\n",p);
                id++;
            }
        }
    }
    i=-1;
}

```

## OUTPUT:



```

}
◆ enter file path (drive:\Fold\filename)
C:\Users\Dell\Documents\sem7\CdL\input.c
#include<stdio.h> is a header file
int is a keyword
main is an identifier
/ is an operator
/ is an operator
sigle is an identifier
liner is an identifier
/ is an operator
* is an operator
multi is an identifier
* is an operator
line is an identifier
/ is an operator
printf is an identifier
HelloWorld is an argument
return is a keyword
0 is a number
Keywords: 2
Identifiers: 6
Operators: 6
Numbers: 1
○ PS C:\Users\Dell\Documents\sem7\CdL>
```

Conclusion:

---

---

---

Name and Sign of Course Teacher  
Mrs. Archana Chitte





# Government College of Engineering, Jalgaon

## (An Autonomous Institute of Govt. of Maharashtra)

Name of Student:.....

PRN:.....

Date of Performance:.....

Date of Completion:.....

### Experiment No 2

**Title:** Write a C program to identify whether a given line is a comment or not.

**Theory:**

To identify whether a given line is a comment or not, you typically need to work with programming or scripting languages that support comments. Here's a general theory for identifying comments in source code:

1. Comments in Programming Languages:

- Comments are non-executable parts of the code that are intended for human readers.
- They provide explanations, documentation, or annotations within the source code.
- Comments are not executed by the compiler or interpreter and are ignored during the compilation or execution of the program.

2. Types of Comments:

- There are different ways to write comments in various programming languages:
  - Single-Line Comments: These are comments that span a single line. They are often preceded by special characters, such as `"/"` in C, C++, Java, and C# or `"#"` in Python.
  - Multi-Line Comments: These comments can span multiple lines and are enclosed within special delimiters, such as `"/" ... "/"` in C, C++, and Java.
  - Documentation Comments: These comments are used to generate documentation for the code. For example, Javadoc comments in Java or Doxygen comments in C++.

3. Comment Identification in Code:

- To identify whether a given line is a comment or not, you need to examine the line and check if it matches the syntax or patterns of comments in the language.
- This typically involves:
  - Looking for comment indicators (e.g., `"/"` or `"#"` for single-line comments, `"/"` and `"/"` for multi-line comments).
  - Ensuring that comments are not within string literals, as they may contain characters that resemble comment indicators.
  - Checking for the position of the comment indicators and the absence of executable code before them.
  - Handling any language-specific rules related to comments.

4. Example in C/C++:

- In C and C++, single-line comments start with `"/"` and continue until the end of the line. Multi-line comments are enclosed within `"/"` and `"/"`.
- A simple program to identify whether a line is a comment in C/C++ might involve reading a line and checking for these patterns at the beginning of the line.

The specific implementation for comment identification will vary depending on the programming language and the tools used. The general theory involves understanding the syntax and rules for comments in the target language and applying these rules to the code being analyzed.

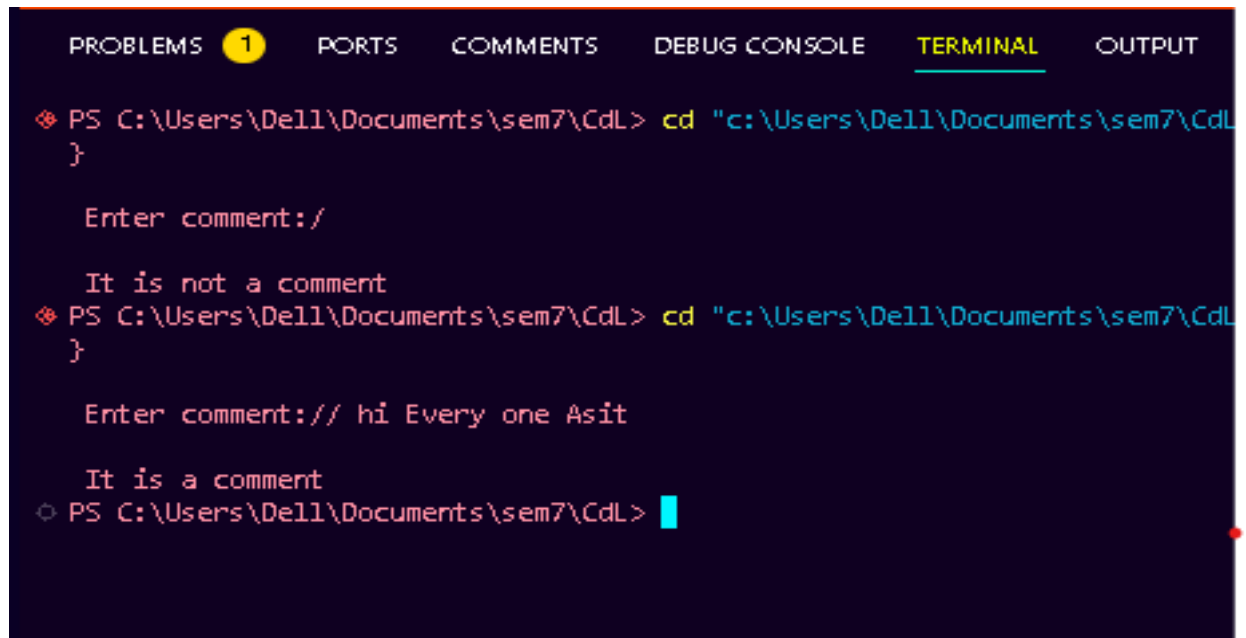
**Program:**

```

#include<stdio.h>
#include<conio.h>
void main() {
    char com[30];
    int i=2,a=0;
    clrscr();
    printf("\n Enter comment:");
    gets(com);
    if(com[0]=='/') {
        if(com[1]=='/')
            printf("\n It is a comment");
        else if(com[1]=='*') {
            for(i=2;i<=30;i++)
            {
                if(com[i]=='*'&&com[i+1]=='/')
                {
                    printf("\n It is a comment");
                    a=1;
                    break; }
            }
        }
        else
            continue; }
    if(a==0)
        printf("\n It is not a comment");
    }
    else
        printf("\n It is not a comment");
    }
    else
        printf("\n It is not a comment");
    getch();
}

```

## OUTPUT:



```
PROBLEMS 1 PORTS COMMENTS DEBUG CONSOLE TERMINAL OUTPUT

❖ PS C:\Users\Dell\Documents\sem7\CdL> cd "c:\Users\Dell\Documents\sem7\CdL
}

Enter comment: /

It is not a comment
❖ PS C:\Users\Dell\Documents\sem7\CdL> cd "c:\Users\Dell\Documents\sem7\CdL
}

Enter comment: // hi Every one Asit

It is a comment
○ PS C:\Users\Dell\Documents\sem7\CdL> 
```

Conclusion:

---

---

---

Name and Sign of Course Teacher  
Mrs. Archana Chitte



# Government College of Engineering, Jalgaon

(An Autonomous Institute of Govt. of Maharashtra)

Name of Student:.....

PRN:.....

Date of Performance:.....

Date of Completion:.....

## Experiment No 3

**Title:** Write a C program to recognize strings under 'a\*', 'a\*b+', 'abb'..

### Theory:

To recognize strings under the regular expressions 'a\*', 'a\*b+', and 'abb', you can create a C program that reads an input string and determines if it matches any of these patterns. Let's discuss the theory behind these regular expressions and provide a sample C program to recognize such strings.

#### 1. Regular Expression 'a':\*

- This regular expression matches strings containing zero or more 'a' characters.
- Example: "a", "aa", "aaa", "", "aaaa", ...

#### 2. Regular Expression 'a\*b':\*

- This regular expression matches strings starting with one or more 'a' characters followed by one or more 'b' characters.
- Example: "ab", "aab", "aaab", "abb", "aaabb", ...

#### 3. Regular Expression 'abb':

- This regular expression matches the exact string "abb."

### C Program to Recognize Strings:

Here's a sample C program that recognizes strings based on the regular expressions 'a\*', 'a\*b+', and 'abb':

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#include<stdlib.h>
void main()
{
char s[20],c;
int state=0,i=0;
clrscr();
printf("\n Enter a string:");
gets(s);
while(s[i]!='\0')
{
switch(state)
{
case 0: c=s[i++];
if(c=='a')
```

```
state=1;
else if(c=='b')
state=2;
else
state=6;
break;
case 1: c=s[i++];
if(c=='a')
state=3; else if(c=='b')
state=4;
else
state=6;
break;
case 2: c=s[i++];
if(c=='a')
state=6;
else if(c=='b')
state=2;
else
state=6;
break;
case 3: c=s[i++];
if(c=='a')
state=3;
else if(c=='b')
state=2;
else
state=6;
break;
case 4: c=s[i++];
if(c=='a')
state=6;
else if(c=='b')
state=5;
else
state=6;
break;
case 5: c=s[i++];
if(c=='a')
state=6;
else if(c=='b')
state=2;
else
state=6;
break;
case 6: printf("\n %s is not recognised.",s);
exit(0);
}
```

```

}
if(state==1)
printf("\n %s is accepted under rule 'a'",s);
else if((state==2)||(state==4))

printf("\n %s is accepted under rule 'a*b+'",s);

else if(state==5)
printf("\n %s is accepted under rule 'abb'",s);
getch();
}

```

## OUTPUT:

```

state = 0;
PS C:\Users\Dell\Documents\sem7\CdL> cd "c:\Users\Dell\Documents\sem7\CdL\
}

Enter a string:abbbb

abbbb is accepted under rule 'a*b+'
PS C:\Users\Dell\Documents\sem7\CdL> cd "c:\Users\Dell\Documents\sem7\CdL\
}

Enter a string:abbdm

abbdm is not recognised.
PS C:\Users\Dell\Documents\sem7\CdL>

```

Conclusion:

---



---



---

Name and Sign of Course Teacher  
Mrs. Archana Chitte





**Government College of Engineering, Jalgaon**  
(An Autonomous Institute of Govt. of Maharashtra)

**Name of Student:**.....

**PRN:**.....

**Date of Performance:**.....

**Date of Completion:**.....

---

**Experiment No 4**

**Title:** Write a C program to simulate lexical analyzer for validating operators...

**Theory:**

Certainly! Let's provide a brief explanation and theory related to writing a Lex program to recognize valid arithmetic expressions, identify identifiers and operators.

Lexical Analysis:

Lexical analysis is the first phase of a compiler that converts the source code into a stream of tokens for further processing. In this phase, the input program is divided into smaller units called tokens, which represent the basic building blocks of the language, such as identifiers, keywords, constants, and operators. Lexical analyzers are typically generated using tools like Flex (fast lexical analyzer generator).

Lex and Flex:

Flex is a popular tool for generating lexical analyzers (lexers or scanners) from a set of regular expressions. These regular expressions define patterns for recognizing tokens in the input source code. Lexers generated by Flex are often used in conjunction with parsers to build compilers and interpreters.

Recognizing Valid Arithmetic Expressions:

To recognize valid arithmetic expressions, a Lex program typically defines regular expressions for operators (e.g., +, -, \*, /), identifiers (e.g., variable names), and constants (e.g., numeric literals). Lexers identify and return tokens representing these elements in the input source code.

Basic Components of a Lex Program:

1. Regular Expressions: Lex programs define regular expressions that describe the patterns of tokens to be recognized. For example, to recognize integer constants, you might use a regular expression like `[0-9]+`.

2. **Actions:** Associated with each regular expression is an action, which is typically written in C. Actions specify what to do when a particular regular expression is matched. For example, when you recognize an identifier, you might store it in a symbol table.
3. **yylex() Function:** The main function in a Flex-generated lexer is `yylex()`. This function scans the input stream and returns tokens based on the regular expressions and associated actions.
4. **Token Output:** When a regular expression is matched, the corresponding action can output the recognized token or perform additional processing. For example, you might return a token like "IDENTIFIER" or "PLUS" when an identifier or plus sign is recognized.
5. **Pattern Rules:** Lex programs define patterns (regular expressions) and associate them with rules. These rules specify what action to take when a pattern is matched. For example:

```
```lex
[0-9]+    { /* Action for recognizing integers */ }
[a-zA-Z][a-zA-Z0-9]* { /* Action for recognizing identifiers */ }
[+*/-]    { /* Action for recognizing operators */ }
```
```

#### Identifiers and Operators:

In the context of your Lex program, "identifiers" are names used for variables, functions, or other user-defined entities in the source code. Operators include symbols like `+`, `-`, `*`, and `/` used for arithmetic operations.

In summary, a Lex program for recognizing valid arithmetic expressions will define regular expressions and actions to tokenize and identify identifiers and operators within the input source code. It serves as the first step in the compilation process, allowing further phases of the compiler to work with these tokens.

#### **Sample Program:**

```
/*
AIM: Validating Operators
```

```

*/
#include<stdio.h>
void main()
{
    char s[5],ch='y',s1[4];
//    printf("\n Enter any operator");
    //gets(s);
    do{
        printf("\n Enter any operator : ");
        scanf("%s",s);
        switch(s[0])
        {
            case '>':
                if(s[1]=='=')
                    printf("\n valid operator : Greater than or equal");
                else
                    printf("\n valid operator : Greater than");
                break;
            case '<':
                if(s[1]=='=')
                    printf("\n valid operator : Less than or equal");
                else
                    printf("\n valid operator : Less than");
                break;
            case '=':
                if(s[1]=='=')
                    printf("\n valid operator : Equal to");
                else
                    printf("\n valid operator : Assignment");
                break;
            case '!':
                if(s[1]=='=')
                    printf("\n valid operator : Not Equal");
                else
                    printf("\n valid operator : Bit Not");
                break;
            case '&':
                if(s[1]=='&')
                    printf("\n valid operator : Logical AND");
                else printf("\n valid operator : Bitwise AND");
                break;
            case '|':
                if(s[1]=='|')
                    printf("\n valid operator : Logical OR");

                else
                    printf("\n valid operator : Bitwise OR");
                break;
            case '+':
                if(s[1]=='+')
                    printf("\n valid operator : unary operator");
                else
                    printf("\n valid operator : Addition");
                break;

```

```

        case '-':
            if(s[1]=='-')
                printf("\n valid operator : unary operator");
            else
                printf("\n valid operator : Substraction");
            break;
        case '*':
            printf("\n valid operator : Multiplication");
            break;
        case '/':
            printf("\n valid operator : Division");
            break;
        case '%':
            printf("\n valid operator : Modulus");
            break;
        default:
            printf("\n Invalid operator !!!!");
            break;
    }
    printf("\n do you want to continue(y:yes,n:No : ");
    scanf("\n%c",&ch);
} while(ch=='Y'||ch=='y');

}

```

## OUTPUT:

```
76
PROBLEMS 1 PORTS COMMENTS DEBUG CONSOLE TERMINAL OUTPUT
PS C:\Users\Dell\Documents\sem7\CdL> cd "c:\Users\Dell\Documents\sem7\CdL
}

Enter any operator : ++

valid operator : unary operator
do you want to continue(y:yes,n:No : y

Enter any operator : --

valid operator : unary operator
do you want to continue(y:yes,n:No : y

Enter any operator : ==

valid operator : Equal to
do you want to continue(y:yes,n:No : y

Enter any operator : =====

valid operator : Assignment
do you want to continue(y:yes,n:No : y

Enter any operator : )+=

Invalid operator !!!!
do you want to continue(y:yes,n:No : n
PS C:\Users\Dell\Documents\sem7\CdL>
```

Conclusion:

---

---

---

Name and Sign of Course Teacher  
Mrs. Archana Chitte



**Government College of Engineering, Jalgaon**  
(An Autonomous Institute of Govt. of Maharashtra)

**Name of Student:**.....

**PRN:**.....

**Date of Performance:**.....

**Date of Completion:**.....

---

**Experiment No 5**

**Title:** Simulate First and Follow of a Grammar.

**Theory:**

- 
- 

Before proceeding, it is highly recommended to be familiar with the basics in Syntax Analysis, LL(1) parsing, and the rules of calculating First and Follow sets of a grammar.

1. [Introduction to Syntax Analysis](#)
2. [Why First and Follow?](#)
3. [FIRST Set in Syntax Analysis](#)
4. [FOLLOW Set in Syntax Analysis](#)

Assuming the reader is familiar with the basics discussed above, let's start discussing how to implement the C program to calculate the first and follow of given grammar.

Example:

Input :

E -> TR

R -> +T R | #

T -> F Y

Y -> \*F Y | #

F -> (E) | i

Output :

First(E) = { (, i, }

First(R) = { +, #, }

First(T) = { (, i, }

First(Y) = { \*, #, }

First(F) = { (, i, }

-----  
Follow(E) = { \$, ), }

Follow(R) = { \$, ), }

Follow(T) = { +, \$, ), }

Follow(Y) = { +, \$, ), }

Follow(F) = { \*, +, \$, ), }

The functions follow and followfirst are both involved in the calculation of the Follow Set of a given Non-Terminal. The follow set of the start symbol will always contain “\$”. Now the calculation of Follow falls under three broad cases :

- If a Non-Terminal on the R.H.S. of any production is followed immediately by a Terminal then it can immediately be included in the Follow set of that Non-Terminal.
- If a Non-Terminal on the R.H.S. of any production is followed immediately by a Non-Terminal, then the First Set of that new Non-Terminal gets included on the follow set of our original Non-Terminal. In case encountered an epsilon i.e. ” # ” then, move on to the next symbol in the production.  
Note: “#” is never included in the Follow set of any Non-Terminal.
- If reached the end of a production while calculating follow, then the Follow set of that non-terminal will include the Follow set of the Non-Terminal on the L.H.S. of that production. This can easily be implemented by recursion.

Assumptions :

1. Epsilon is represented by ‘#’.
2. Productions are of the form A=B, where ‘A’ is a single Non-Terminal and ‘B’ can be any combination of Terminals and Non- Terminals.
3. L.H.S. of the first production rule is the start symbol.
4. Grammar is not left recursive.
5. Each production of a non-terminal is entered on a different line.
6. Only Upper Case letters are Non-Terminals and everything else is a terminal.
7. Do not use ‘!’ or ‘\$’ as they are reserved for special purposes.

Explanation :

Store the grammar on a 2D character array production. findfirst function is for calculating the first of any non terminal. Calculation of first falls under two broad cases :

- If the first symbol in the R.H.S of the production is a Terminal then it can directly be included in the first set.
- If the first symbol in the R.H.S of the production is a Non-Terminal then call the findfirst function again on that Non-Terminal. To handle these cases like Recursion is the best possible solution. Here again, if the First of the new Non-Terminal contains an epsilon then we have to move to the next symbol of the original production which can again be a Terminal or a Non-Terminal.

*Note: For the second case it is very easy to fall prey to an INFINITE LOOP even if the code looks perfect. So it is important to keep track of all the function calls at all times and never call the same function again.*

### Sample Program:

```
//f a given grammar
#include <ctype.h>
#include <stdio.h>
#include <string.h>
```

```
// Functions to calculate Follow
void followfirst(char, int, int);
void follow(char c);
```

```
// Function to calculate First
void findfirst(char, int, int);
```

```
int count, n = 0;
```



```

// Stores the final result
// of the First Sets
char calc_first[10][100];

// Stores the final result
// of the Follow Sets
char calc_follow[10][100];
int m = 0;

// Stores the production rules
char production[10][10];
char f[10], first[10];
int k;
char ck;
int e;

int main(int argc, char    argv)
{
    int jm = 0;
    int km = 0;
    int i, choice;
    char c, ch;
    count = 8;

    // The Input grammar
    strcpy(production[0], "X=TnS");
    strcpy(production[1], "X=Rm");
    strcpy(production[2], "T=q");
    strcpy(production[3], "T=#");
    strcpy(production[4], "S=p");
    strcpy(production[5], "S=#");
    strcpy(production[6], "R=om");
    strcpy(production[7], "R=ST");

    int kay;
    char done[count];
    int ptr = -1;

    // Initializing the calc_first array
    for (k = 0; k < count; k++) {
        for (kay = 0; kay < 100; kay++) {
            calc_first[k][kay] = '!';
        }
    }
    int point1 = 0, point2, xxx;

    for (k = 0; k < count; k++) {
        c = production[k][0];
        point2 = 0;
        xxx = 0;

        // Checking if First of c has
        // already been calculated

```

```

for (kay = 0; kay <= ptr; kay++)
    if (c == done[kay])
        xxx = 1;

if (xxx == 1)
    continue;

// Function call
findfirst(c, 0, 0);
ptr += 1;

// Adding c to the calculated list
done[ptr] = c;
printf("\n First(%c) = { ", c);
calc_first[point1][point2++] = c;

// Printing the First Sets of the grammar
for (i = 0 + jm; i < n; i++) {
    int lark = 0, chk = 0;

    for (lark = 0; lark < point2; lark++) {

        if (first[i] == calc_first[point1][lark]) {
            chk = 1;
            break;
        }
    }
    if (chk == 0) {
        printf("%c, ", first[i]);
        calc_first[point1][point2++] = first[i];
    }
}
printf("}\n");
jm = n;
point1++;
}
printf("\n");
printf("-----"
    "\n\n");
char donee[count];
ptr = -1;

// Initializing the calc_follow array
for (k = 0; k < count; k++) {
    for (kay = 0; kay < 100; kay++) {
        calc_follow[k][kay] = '!';
    }
}
point1 = 0;
int land = 0;
for (e = 0; e < count; e++) {
    ck = production[e][0];
    point2 = 0;
    xxx = 0;

```

```

// Checking if Follow of ck
// has already been calculated
for (kay = 0; kay <= ptr; kay++)
    if (ck == donee[kay])
        xxx = 1;

if (xxx == 1)
    continue;
land += 1;

// Function call
follow(ck);
ptr += 1;

// Adding ck to the calculated list
donee[ptr] = ck;
printf(" Follow(%c) = { ", ck);
calc_follow[point1][point2++] = ck;

// Printing the Follow Sets of the grammar
for (i = 0 + km; i < m; i++) {
    int lark = 0, chk = 0;
    for (lark = 0; lark < point2; lark++) {
        if (f[i] == calc_follow[point1][lark]) {
            chk = 1;
            break;
        }
    }
    if (chk == 0) {
        printf("%c, ", f[i]);
        calc_follow[point1][point2++] = f[i];
    }
}
printf(" }\n\n");
km = m;
point1++;
}
}

```

```

void follow(char c)
{

```

```

    int i, j;

```

```

// Adding "$" to the follow

```

```

// set of the start symbol

```

```

if (production[0][0] == c) {

```

```

    f[m++] = '$';

```

```

}

```

```

for (i = 0; i < 10; i++) {

```

```

    for (j = 2; j < 10; j++) {

```

```

        if (production[i][j] == c) {

```

```

            if (production[i][j + 1] != '\0') {

```

```

                // Calculate the first of the next

```

```

        // Non-Terminal in the production
        followfirst(production[i][j + 1], i,
            (j + 2));
    }

    if (production[i][j + 1] == '\0'
        && c != production[i][0]) {
        // Calculate the follow of the
        // Non-Terminal in the L.H.S. of the
        // production
        follow(production[i][0]);
    }
}
}
}

void findfirst(char c, int q1, int q2)
{
    int j;

    // The case where we
    // encounter a Terminal
    if (!(isupper(c))) {
        first[n++] = c;
    }
    for (j = 0; j < count; j++) {
        if (production[j][0] == c) {
            if (production[j][2] == '#') {
                if (production[q1][q2] == '\0')
                    first[n++] = '#';
                else if (production[q1][q2] != '\0'
                    && (q1 != 0 || q2 != 0)) {
                    // Recursion to calculate First of New
                    // Non-Terminal we encounter after
                    // epsilon
                    findfirst(production[q1][q2], q1,
                        (q2 + 1));
                }
                else
                    first[n++] = '#';
            }
            else if (!(isupper(production[j][2]))) {
                first[n++] = production[j][2];
            }
            else {
                // Recursion to calculate First of
                // New Non-Terminal we encounter
                // at the beginning
                findfirst(production[j][2], j, 3);
            }
        }
    }
}
}

```

```

void followfirst(char c, int c1, int c2)
{
    int k;

    // The case where we encounter
    // a Terminal
    if (!(isupper(c)))
        f[m++] = c;
    else {
        int i = 0, j = 1;
        for (i = 0; i < count; i++) {
            if (calc_first[i][0] == c)
                break;
        }

        // Including the First set of the
        // Non-Terminal in the Follow of
        // the original query
        while (calc_first[i][j] != '#') {
            if (calc_first[i][j] != '#') {
                f[m++] = calc_first[i][j];
            }
            else {
                if (production[c1][c2] == '\0') {
                    // Case where we reach the
                    // end of a production
                    follow(production[c1][0]);
                }
                else {
                    // Recursion to the next symbol
                    // in case we encounter a "#"
                    followfirst(production[c1][c2], c1,
                                c2 + 1);
                }
            }
            j++;
        }
    }
}

```

## OUTPUT

```
PROBLEMS 1 PORTS COMMENTS DEBUG CONSOLE TERMINAL OUTPUT
◆ PS C:\Users\Dell\Documents\sem7\CdL> cd "c:\Users\Dell\Documents\sem7\CdL
}

First(X) = { q, n, o, p, #, }
First(T) = { q, #, }
First(S) = { p, #, }
First(R) = { o, p, q, #, }

-----

Follow(X) = { $, }
Follow(T) = { n, m, }
Follow(S) = { $, q, m, }
Follow(R) = { m, }

○ PS C:\Users\Dell\Documents\sem7\CdL> █
```

Conclusion:

---

---

---

Name and Sign of Course Teacher  
Mrs. Archana Chitte

# Government College of Engineering, Jalgaon

(An Autonomous Institute of Govt. of Maharashtra)

Name of Student:.....

PRN:.....

Date of Performance:.....

Date of Completion:.....

## Experiment No 6

**Title:** Write a program for constructing LL (1) parsing. Table

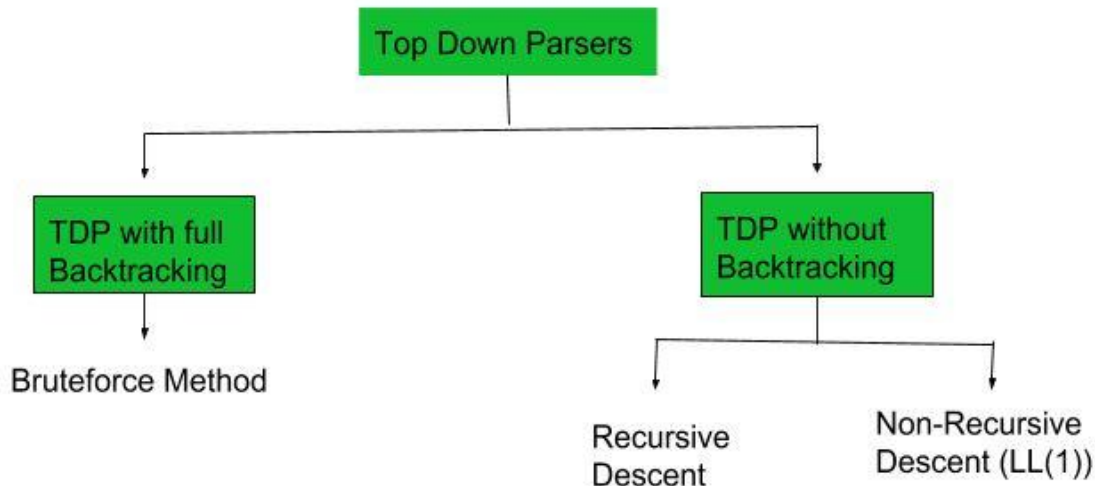
**Theory:**

Prerequisite – [Classification of top-down parsers](#), [FIRST Set](#), [FOLLOW Set](#)

A top-down parser builds the parse tree from the top down, starting with the start non-terminal. There are two types of Top-Down Parsers:

1. Top-Down Parser with Backtracking
2. Top-Down Parsers without Backtracking

Top-Down Parsers without backtracking can further be divided into two parts:



In this article, we are going to discuss Non-Recursive Descent which is also known as LL(1) Parser.

**LL(1) Parsing:** Here the 1st L represents that the scanning of the Input will be done from the Left to Right manner and the second L shows that in this parsing technique, we are going to use the 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.

**Essential conditions to check first are as follows:**

1. The grammar is free from left recursion.
2. The grammar should not be ambiguous.
3. The grammar has to be left factored in so that the grammar is deterministic grammar.

These conditions are necessary but not sufficient for proving a LL(1) parser.

### Algorithm to construct LL(1) Parsing Table:

**Step 1:** First check all the essential conditions mentioned above and go to step 2.

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

1. **First()**: If there is a variable, and from that variable, if we try to drive all the strings then the beginning Terminal Symbol is called the First.
2. **Follow()**: What is the Terminal Symbol which follows a variable in the process of derivation.

**Step 3:** For each production  $A \rightarrow \alpha$ . ( $A$  tends to  $\alpha$ )

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

Now, let's understand with an example.

#### Sample Program:

```
#include<stdio.h>
#include<string.h>
#define TSIZE 128
int table[100][TSIZE];
char terminal[TSIZE];
struct product {
    char str[100];
    int len;
}pro[20];
int no_pro;
char first[26][TSIZE];
char follow[26][TSIZE];
char first_rhs[100][TSIZE];

int isNT(char c) {
    return c >= 'A' && c <= 'Z';
}

// lendo os dados a partir do arquivo txt
void readFromFile() {
    FILE* fptr;
    fptr = fopen("texto.txt", "r");
```



```

char buffer[255];
int i;
int j;
while (fgets(buffer, sizeof(buffer), fptr)) {
    printf("%s", buffer);
    j = 0;
    nonterminal[buffer[0] - 'A'] = 1;
    for (i = 0; i < strlen(buffer) - 1; ++i) {
        if (buffer[i] == '|') {
            ++no_pro;
            pro[no_pro - 1].str[j] = '\0';
            pro[no_pro - 1].len = j;
            pro[no_pro].str[0] = pro[no_pro - 1].str[0];
            pro[no_pro].str[1] = pro[no_pro - 1].str[1];
            pro[no_pro].str[2] = pro[no_pro - 1].str[2];
            j = 3;
        }
        else {
            pro[no_pro].str[j] = buffer[i];
            ++j;
            if (!isNT(buffer[i]) && buffer[i] != '-' && buffer[i] != '>') {
                terminal[buffer[i]] = 1;
            }
        }
    }
    pro[no_pro].len = j;
    ++no_pro;
}

void add_FIRST_A_to_FOLLOW_B(char A, char B) {
    int i;
    for (i = 0; i < TSIZE; ++i) {
        if (i != '^')
            follow[B - 'A'][i] = follow[B - 'A'][i] || first[A - 'A'][i];
    }
}

void add_FOLLOW_A_to_FOLLOW_B(char A, char B) {
    int i;
    for (i = 0; i < TSIZE; ++i) {
        if (i != '^')
            follow[B - 'A'][i] = follow[B - 'A'][i] || follow[A - 'A'][i];
    }
}

void FOLLOW() {
    int t = 0;
    int i, j, k, x;
    while (t++ < no_pro) {
        for (k = 0; k < 26; ++k) {
            if (!nonterminal[k]) continue;
            char nt = k + 'A';
            for (i = 0; i < no_pro; ++i) {
                for (j = 3; j < pro[i].len; ++j) {
                    if (nt == pro[i].str[j]) {
                        for (x = j + 1; x < pro[i].len; ++x) {

```

```

        char sc = pro[i].str[x];
        if (isNT(sc)) {
            add_FIRST_A_to_FOLLOW_B(sc, nt);
            if (first[sc - 'A']['^'])
                continue;
        }
        else {
            follow[nt - 'A'][sc] = 1;
        }
        break;
    }
    if (x == pro[i].len)
        add_FOLLOW_A_to_FOLLOW_B(pro[i].str[0], nt);
}
}
}
}
}
void add_FIRST_A_to_FIRST_B(char A, char B) {
    int i;
    for (i = 0; i < TSIZE; ++i) {
        if (i != '^') {
            first[B - 'A'][i] = first[A - 'A'][i] || first[B - 'A'][i];
        }
    }
}
void FIRST() {
    int i, j;
    int t = 0;
    while (t < no_pro) {
        for (i = 0; i < no_pro; ++i) {
            for (j = 3; j < pro[i].len; ++j) {
                char sc = pro[i].str[j];
                if (isNT(sc)) {
                    add_FIRST_A_to_FIRST_B(sc, pro[i].str[0]);
                    if (first[sc - 'A']['^'])
                        continue;
                }
                else {
                    first[pro[i].str[0] - 'A'][sc] = 1;
                }
                break;
            }
            if (j == pro[i].len)
                first[pro[i].str[0] - 'A']['^'] = 1;
        }
        ++t;
    }
}
void add_FIRST_A_to_FIRST_RHS__B(char A, int B) {
    int i;
    for (i = 0; i < TSIZE; ++i) {
        if (i != '^')

```

```

        first_rhs[B][i] = first[A - 'A'][i] || first_rhs[B][i];
    }
}
// Calcula o FIRST(❖) para cada A->❖
void FIRST_RHS() {
    int i, j;
    int t = 0;
    while (t < no_pro) {
        for (i = 0; i < no_pro; ++i) {
            for (j = 3; j < pro[i].len; ++j) {
                char sc = pro[i].str[j];
                if (isNT(sc)) {
                    add_FIRST_A_to_FIRST_RHS__B(sc, i);
                    if (first[sc - 'A']['^'])
                        continue;
                }
                else {
                    first_rhs[i][sc] = 1;
                }
                break;
            }
            if (j == pro[i].len)
                first_rhs[i]['^'] = 1;
        }
        ++t;
    }
}
int main() {
    readFromFile();
    follow[pro[0].str[0] - 'A']['$'] = 1;
    FIRST();
    FOLLOW();
    FIRST_RHS();
    int i, j, k;

    // mostra o first de cada variavel
    printf("\n");
    for (i = 0; i < no_pro; ++i) {
        if (i == 0 || (pro[i - 1].str[0] != pro[i].str[0])) {
            char c = pro[i].str[0];
            printf("FIRST OF %c: ", c);
            for (j = 0; j < TSIZE; ++j) {
                if (first[c - 'A'][j]) {
                    printf("%c ", j);
                }
            }
            printf("\n");
        }
    }

    // mostrar o follow de cada variavel
    printf("\n");
    for (i = 0; i < no_pro; ++i) {

```

```

if (i == 0 || (pro[i - 1].str[0] != pro[i].str[0])) {
    char c = pro[i].str[0];
    printf("FOLLOW OF %c: ", c);
    for (j = 0; j < TSIZE; ++j) {
        if (follow[c - 'A'][j]) {
            printf("%c ", j);
        }
    }
    printf("\n");
}
}

// mostra o first de cada variavel na forma de A->
printf("\n");
for (i = 0; i < no_pro; ++i) {
    printf("FIRST OF %s: ", pro[i].str);
    for (j = 0; j < TSIZE; ++j) {
        if (first_rhs[i][j]) {
            printf("%c ", j);
        }
    }
    printf("\n");
}

terminal['$'] = 1;
terminal['^'] = 0;

//
printf("\n");
printf("\nt\t\t\t\t\tTABELA LL(1) Parser\t\t\t\t\t*\n");
printf("\t-----\n");
printf("%-10s", "");
for (i = 0; i < TSIZE; ++i) {
    if (terminal[i]) printf("%-10c", i);
}
printf("\n");
int p = 0;
for (i = 0; i < no_pro; ++i) {
    if (i != 0 && (pro[i].str[0] != pro[i - 1].str[0]))
        p = p + 1;
    for (j = 0; j < TSIZE; ++j) {
        if (first_rhs[i][j] && j != '^') {
            table[p][j] = i + 1;
        }
        else if (first_rhs[i]['^']) {
            for (k = 0; k < TSIZE; ++k) {
                if (follow[pro[i].str[0] - 'A'][k]) {
                    table[p][k] = i + 1;
                }
            }
        }
    }
}
}

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

```

```

    if (i == 0 || (pro[i - 1].str[0] != pro[i].str[0])) {
        printf("%-10c", pro[i].str[0]);
        for (j = 0; j < TSIZE; ++j) {
            if (table[k][j]) {
                printf("%-10s", pro[table[k][j] - 1].str);
            }
            else if (terminal[j]) {
                printf("%-10s", "");
            }
        }
        ++k;
        printf("\n");
    }
}
}

```

## OUTPUT:

```

PS C:\Users\Dell\Documents\sem7\CdL> cd "c:\Users\Dell\Documents\sem7\CdL\" ;
}
E->TA
A->+TA|^
T->FB
B->*FB|^
F->t|(E)
FIRST OF E: ( t
FIRST OF A: + ^
FIRST OF T: ( t
FIRST OF B: * ^
FIRST OF F: ( t

FOLLOW OF E: $ * +
FOLLOW OF A: $ * +
FOLLOW OF T: $ * +
FOLLOW OF B: $ * +
FOLLOW OF F: $ * +

FIRST OF E->TA: ( t
FIRST OF A->+TA: +
FIRST OF A->^: ^
FIRST OF T->FB: ( t
FIRST OF B->*FB: *
FIRST OF B->^: ^
FIRST OF F->t: t
FIRST OF F->(E: (

***** TABELA LL(1) Parser *****
-----
      $      (      *      +      t
E      E->TA
A      A->^
T      T->FB
B      B->^
F      F->(E      F->t
PS C:\Users\Dell\Documents\sem7\CdL> S

```

Conclusion:

---

---

---

Name and Sign of Course Teacher  
Mrs. Archana Chitte

# Government College of Engineering, Jalgaon

(An Autonomous Institute of Govt. of Maharashtra)

Name of Student:.....

PRN:.....

Date of Performance:.....

Date of Completion:.....

## Experiment No 7

**Title:** Write a C program to implement operator precedence parsing

### Theory:

Operator precedence parsing

Operator precedence grammar is kinds of shift reduce parsing method. It is applied to a small class of operator grammars.

A grammar is said to be operator precedence grammar if it has two properties:

- No R.H.S. of any production has  $a \in$ .
- No two non-terminals are adjacent.

Operator precedence can only established between the terminals of the grammar. It ignores the non-terminal.

There are the three operator precedence relations:

$a > b$  means that terminal "a" has the higher precedence than terminal "b".

$a < b$  means that terminal "a" has the lower precedence than terminal "b".

$a \doteq b$  means that the terminal "a" and "b" both have same precedence.

Precedence table:

|    | +      | *      | (      | )        | id     | \$     |
|----|--------|--------|--------|----------|--------|--------|
| +  | $\geq$ | $\leq$ | $\leq$ | $\geq$   | $\leq$ | $\geq$ |
| *  | $\geq$ | $\geq$ | $\leq$ | $\geq$   | $\leq$ | $\geq$ |
| (  | $\leq$ | $\leq$ | $\leq$ | $\doteq$ | $\leq$ | X      |
| )  | $\geq$ | $\geq$ | X      | $\geq$   | X      | $\geq$ |
| id | $\geq$ | $\geq$ | X      | $\geq$   | X      | $\geq$ |
| \$ | $\leq$ | $\leq$ | $\leq$ | X        | $\leq$ | X      |

### Parsing Action

- Both end of the given input string, add the \$ symbol.
- Now scan the input string from left right until the  $\geq$  is encountered.
- Scan towards left over all the equal precedence until the first left most  $\leq$  is encountered.
- Everything between left most  $\leq$  and right most  $\geq$  is a handle.
- \$ on \$ means parsing is successful.

### Example

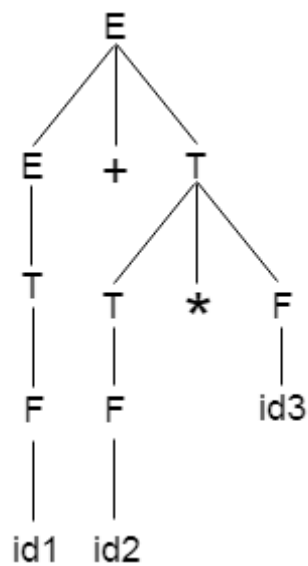
Grammar:

1.  $E \rightarrow E+T/T$
2.  $T \rightarrow T*F/F$
3.  $F \rightarrow id$

Given string:

1.  $w = id + id * id$

Let us consider a parse tree for it as follows:



On the basis of above tree, we can design following operator precedence table:

|    | E               | T               | F               | id              | +                | *                | \$               |
|----|-----------------|-----------------|-----------------|-----------------|------------------|------------------|------------------|
| E  | X               | X               | X               | X               | $\doteq$         | X                | $\triangleright$ |
| T  | X               | X               | X               | X               | $\triangleright$ | $\doteq$         | $\triangleright$ |
| F  | X               | X               | X               | X               | $\triangleright$ | $\triangleright$ | $\triangleright$ |
| id | X               | X               | X               | X               | $\triangleright$ | $\triangleright$ | $\triangleright$ |
| +  | X               | $\doteq$        | $\triangleleft$ | $\triangleleft$ | X                | X                | X                |
| *  | X               | X               | $\doteq$        | $\triangleleft$ | X                | X                | X                |
| \$ | $\triangleleft$ | $\triangleleft$ | $\triangleleft$ | $\triangleleft$ | X                | X                | X                |

Now let us process the string with the help of the above precedence table:

\$ < id1 > + id2 \* id3 \$

\$ < F > + id2 \* id3 \$

\$ < T > + id2 \* id3 \$

\$ < E  $\doteq$  + < id2 > \* id3 \$

\$ < E  $\doteq$  + < F > \* id3 \$

\$ < E  $\doteq$  + < T  $\doteq$  \* < id3 > \$

\$ < E  $\doteq$  + < T  $\doteq$  \*  $\doteq$  F > \$

\$ < E  $\doteq$  +  $\doteq$  T > \$

\$ < E  $\doteq$  +  $\doteq$  T > \$

\$ < E > \$

Accept.



### Sample Program:

```
/*
/*
Aim:operator precedence parsing
*/
#include<stdio.h>
#include<string.h>
void main(){
int i,j,k,n,top=0,col,row;
char stack[20], ip[10], opt[10][10][10],ter[10];
for(i=0;i<10;i++)
{
stack[i]=NULL;
ip[i]=NULL;
for(j=0;j<10;j++)
{
opt[i][j][1] =NULL;
}
}
printf("Enter the no.of terminals :\n");
scanf("%d",&n);
printf("\nEnter the terminals :\n");
scanf("%s",&ter);
printf("\nEnter the table values :\n");
for(i=0;i<n;i++)
{
for(j=0;j<n;j++)
{
printf("Enter the value for %c %c:",ter[i],ter[j]);
scanf("%s",opt[i][j][1]);
}
}
printf("\n      OPERATOR PRECEDENCE TABLE      \n");
for(i=0;i<n;i++)
{
printf("\t%c",ter[i]);
}
printf("\n");
for(i=0;i<n;i++){ printf("\n%c",ter[i]);
for(j=0;j<n;j++){ printf("\t%c",opt[i][j][0]);} }
stack[top]='$';
printf("\nEnter the input string:");
scanf("%s",ip);
i=0;
printf("\nSTACK\t\t\tINPUT STRING\t\t\tACTION\n");
printf("\n%s\t\t\t\t%s\t\t\t",stack,ip);
while(i<=strlen(ip))
{
for(k=0;k<n;k++)
{
if(stack[top]==ter[k])
col=k;
if(ip[i] ==ter[k])
row=k;

```

```

}
if((stack[top]=='$')&&(ip[i]=='$')){
printf("String is accepted\n");
break;}
else if((opt[col][row][0]=='<') ||(opt[col][row][0]=='='))
{ stack[++top]=opt[col][row][0];
stack[++top]=ip[i];
printf("Shift %c",ip[i]);
i++;
}
else{
if(opt[col][row][0]=='>')
{
while(stack[top]!='<'){--top;}
top=top-1;
printf("Reduce");
}
else
{
printf("\nString is not accepted");
break;
}
}
printf("\n");
for(k=0;k<=top;k++)
{
printf("%c",stack[k]);
}
printf("\t\t\t");
for(k=i;k<strlen(ip);k++){
printf("%c",ip[k]);
}
printf("\t\t\t");
}
//getch();
}
/*

```

## OUTPUT:

```
Enter the number of terminals: 5
PS C:\Users\Dell\Documents\sem7\CdL> cd "c:\Users\Dell\Documents\sem7\CdL\"
Enter the number of terminals: 5
Enter the terminals: i+-*$
Enter the table values:
Enter the value for i i: e
Enter the value for i +: >
Enter the value for i -: >
Enter the value for i *: >
Enter the value for i $: >
Enter the value for + i: <
Enter the value for + +: >
Enter the value for + -: >
Enter the value for + *: <
Enter the value for + $: >
Enter the value for - i: <
Enter the value for - +: >
Enter the value for - -: >
Enter the value for - *: <
Enter the value for - $: >
Enter the value for * i: <
Enter the value for * +: >
Enter the value for * -: >
Enter the value for * *: >
Enter the value for * $: >
Enter the value for $ i: <
Enter the value for $ +: <
Enter the value for $ -: <
Enter the value for $ *: <
Enter the value for $ $: a

** OPERATOR PRECEDENCE TABLE **
      i      +      -      *      $
i      e      >      >      >      >
+      <      >      >      <      >
-      <      >      >      <      >
*      <      >      >      >      >
$      <      <      <      <      a

Enter the input string: i+i*i-i$

STACK          INPUT STRING          ACTION

$              i+i*i-i$              Shift i
$<i            +i*i-i$              Reduce
$              +i*i-i$              Shift +
$<+            i*i-i$              Shift i
$<+<i          *i-i$              Reduce
$<+            *i-i$              Shift *
$<+<+          i-i$              Shift i
$<+<+<i        -i$              Reduce
$<+<+          -i$              Reduce
$<+            -i$              Reduce
$              -i$              Shift -
$<-            i$              Shift i
$<-<i          $              Reduce
$<-            $              Reduce
$              $              String is accepted
PS C:\Use
```

Conclusion:

---

---

---

Name and Sign of Course Teacher  
Mrs. Archana Chitte

# Government College of Engineering, Jalgaon

(An Autonomous Institute of Govt. of Maharashtra)

Name of Student:.....

PRN:.....

Date of Performance:.....

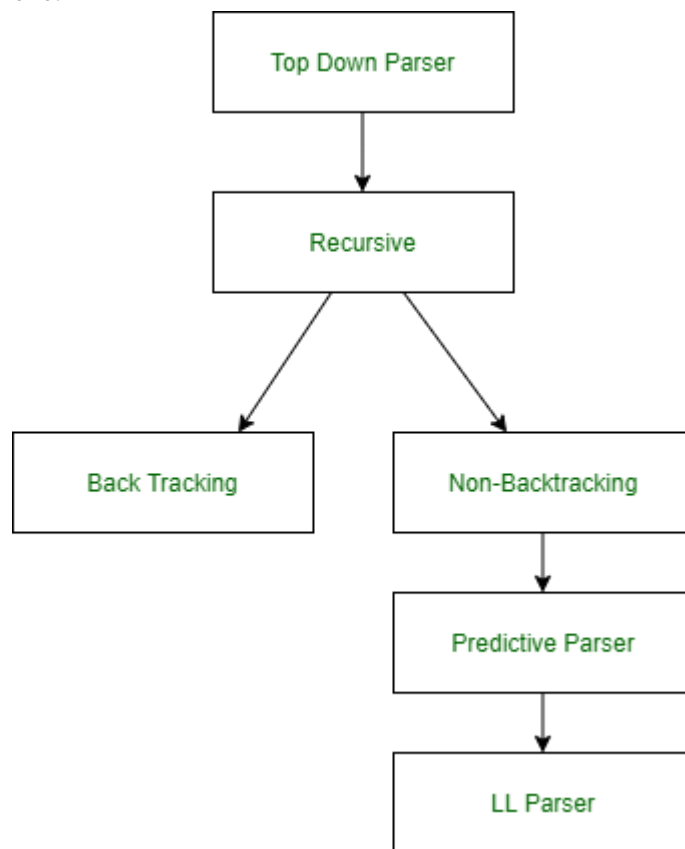
Date of Completion:.....

## Experiment No 8

**Title:** Design of a Predictive parser of given language

**Theory:**

In this, we will cover the overview of Predictive Parser and mainly focus on the role of Predictive Parser. And will also cover the algorithm for the implementation of the Predictive parser algorithm and finally will discuss an example by implementing the algorithm for precedence parsing. Let's discuss it one by one.



### Predictive Parser :

A predictive parser is a recursive descent parser with no backtracking or backup. It is a top-down parser that does not require backtracking. At each step, the choice of the rule to be expanded is made upon the next terminal symbol.

Consider

$A \rightarrow A_1 \mid A_2 \mid \dots \mid A_n$

If the non-terminal is to be further expanded to 'A', the rule is selected based on the current input symbol 'a' only.

### Predictive Parser Algorithm :

1. Make a transition diagram(DFA/NFA) for every rule of grammar.
2. Optimize the DFA by reducing the number of states, yielding the final transition diagram.
3. Simulate the string on the transition diagram to parse a string.
4. If the transition diagram reaches an accept state after the input is consumed, it is parsed.

Consider the following grammar –

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

After removing left recursion, left factoring

$E \rightarrow TT'$

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

$T \rightarrow FT''$

$T'' \rightarrow *FT'' \mid \epsilon$

$F \rightarrow (E) \mid id$

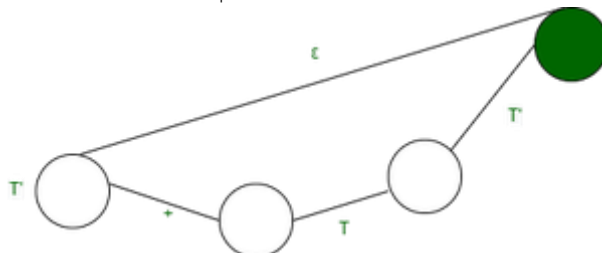
#### STEP 1:

Make a transition diagram(DFA/NFA) for every rule of grammar.

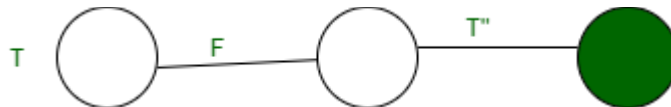
- $E \rightarrow TT'$



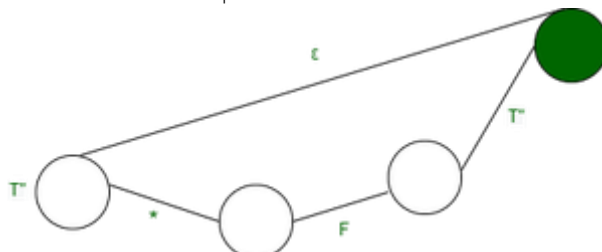
- $T' \rightarrow +TT' \mid \epsilon$



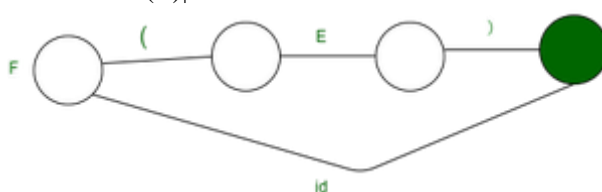
- $T \rightarrow FT''$



- $T'' \rightarrow *FT'' \mid \epsilon$



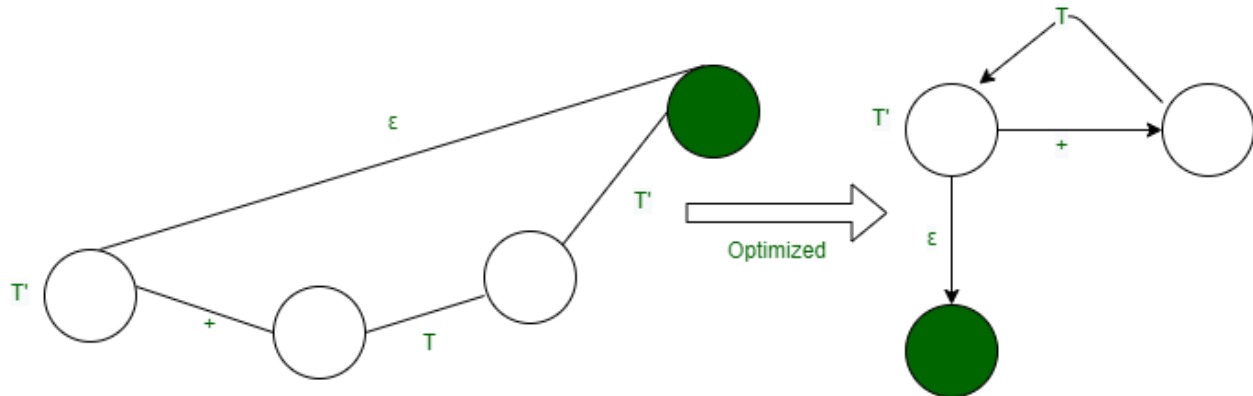
- $F \rightarrow (E) \mid id$



### STEP 2:

Optimize the DFA by decreases the number of states, yielding the final transition diagram.

- $T' \rightarrow +TT' | \epsilon$



### STEP 3:

Simulation on the input string.

Steps involved in the simulation procedure are:

1. Start from the starting state.
2. If a terminal arrives consume it, move to the next state.
3. If a non-terminal arrive go to the state of the DFA of the non-terminal and return on reached up to the final state.
4. Return to actual DFA and Keep doing parsing.
5. If one completes reading the input string completely, you reach a final state, and the string is successfully parsed.

### Source Program:

```
#include<stdio.h>
#include<string.h>
#define TSIZE 128
int table[100][TSIZE];
char terminal[TSIZE];
char nonterminal[26];
struct product {
    char str[100];
    int len;
}pro[20];
int no_pro;
char first[26][TSIZE];
char follow[26][TSIZE];
char first_rhs[100][TSIZE];
int isNT(char c) {
    return c >= 'A' && c <= 'Z';
}
// lendo os dados a partir do arquivo txt
void readFromFile() {
```

```

FILE* fptr;
fptr = fopen("texto.txt", "r");
char buffer[255];
int i;
int j;
while (fgets(buffer, sizeof(buffer), fptr)) {
    printf("%s", buffer);
    j = 0;
    nonterminal[buffer[0] - 'A'] = 1;
    for (i = 0; i < strlen(buffer) - 1; ++i) {
        if (buffer[i] == '|') {
            ++no_pro;
            pro[no_pro - 1].str[j] = '\0';
            pro[no_pro - 1].len = j;
            pro[no_pro].str[0] = pro[no_pro - 1].str[0];
            pro[no_pro].str[1] = pro[no_pro - 1].str[1];
            pro[no_pro].str[2] = pro[no_pro - 1].str[2];
            j = 3;
        }
        else {
            pro[no_pro].str[j] = buffer[i];
            ++j;
            if (!isNT(buffer[i]) && buffer[i] != '-' && buffer[i] != '>') {
                terminal[buffer[i]] = 1;
            }
        }
    }
    pro[no_pro].len = j;
    ++no_pro;
}

void add_FIRST_A_to_FOLLOW_B(char A, char B) {
    int i;
    for (i = 0; i < TSIZE; ++i) {
        if (i != '^')
            follow[B - 'A'][i] = follow[B - 'A'][i] || first[A - 'A'][i];
    }
}

void add_FOLLOW_A_to_FOLLOW_B(char A, char B) {
    int i;
    for (i = 0; i < TSIZE; ++i) {
        if (i != '^')
            follow[B - 'A'][i] = follow[B - 'A'][i] || follow[A - 'A'][i];
    }
}

void FOLLOW() {
    int t = 0;

```



```

int i, j, k, x;
while (t++ < no_pro) {
    for (k = 0; k < 26; ++k) {
        if (!nonterminal[k]) continue;
        char nt = k + 'A';
        for (i = 0; i < no_pro; ++i) {
            for (j = 3; j < pro[i].len; ++j) {
                if (nt == pro[i].str[j]) {
                    for (x = j + 1; x < pro[i].len; ++x) {
                        char sc = pro[i].str[x];
                        if (isNT(sc)) {
                            add_FIRST_A_to_FOLLOW_B(sc, nt);
                            if (first[sc - 'A']['^'])
                                continue;
                        }
                    }
                    else {
                        follow[nt - 'A'][sc] = 1;
                    }
                    break;
                }
            }
            if (x == pro[i].len)
                add_FOLLOW_A_to_FOLLOW_B(pro[i].str[0], nt);
        }
    }
}

void add_FIRST_A_to_FIRST_B(char A, char B) {
    int i;
    for (i = 0; i < TSIZE; ++i) {
        if (i != '^') {
            first[B - 'A'][i] = first[A - 'A'][i] || first[B - 'A'][i];
        }
    }
}

void FIRST() {
    int i, j;
    int t = 0;
    while (t < no_pro) {
        for (i = 0; i < no_pro; ++i) {
            for (j = 3; j < pro[i].len; ++j) {
                char sc = pro[i].str[j];
                if (isNT(sc)) {
                    add_FIRST_A_to_FIRST_B(sc, pro[i].str[0]);
                    if (first[sc - 'A']['^'])
                        continue;
                }
            }
        }
    }
}

```

```

        }
        else {
            first[pro[i].str[0] - 'A'][sc] = 1;
        }
        break;
    }
    if (j == pro[i].len)
        first[pro[i].str[0] - 'A']['^'] = 1;
    }
    ++t;
}
}

void add_FIRST_A_to_FIRST_RHS__B(char A, int B) {
    int i;
    for (i = 0; i < TSIZE; ++i) {
        if (i != '^')
            first_rhs[B][i] = first[A - 'A'][i] || first_rhs[B][i];
    }
}

// Calcula o FIRST(❖) para cada A->❖
void FIRST_RHS() {
    int i, j;
    int t = 0;
    while (t < no_pro) {
        for (i = 0; i < no_pro; ++i) {
            for (j = 3; j < pro[i].len; ++j) {
                char sc = pro[i].str[j];
                if (isNT(sc)) {
                    add_FIRST_A_to_FIRST_RHS__B(sc, i);
                    if (first[sc - 'A']['^'])
                        continue;
                }
                else {
                    first_rhs[i][sc] = 1;
                }
            }
            break;
        }
        if (j == pro[i].len)
            first_rhs[i]['^'] = 1;
    }
    ++t;
}

int main() {
    readFromFile();
    follow[pro[0].str[0] - 'A']['$'] = 1;
    FIRST();
}

```

```

FOLLOW();
FIRST_RHS();
int i, j, k;

```

*// mostra o first de cada variável*

```

printf("\n");
for (i = 0; i < no_pro; ++i) {
    if (i == 0 || (pro[i - 1].str[0] != pro[i].str[0])) {
        char c = pro[i].str[0];
        printf("FIRST OF %c: ", c);
        for (j = 0; j < TSIZE; ++j) {
            if (first[c - 'A'][j]) {
                printf("%c ", j);
            }
        }
        printf("\n");
    }
}

```

```

printf("\n");
for (i = 0; i < no_pro; ++i) {
    if (i == 0 || (pro[i - 1].str[0] != pro[i].str[0])) {
        char c = pro[i].str[0];
        printf("FOLLOW OF %c: ", c);
        for (j = 0; j < TSIZE; ++j) {
            if (follow[c - 'A'][j]) {
                printf("%c ", j);
            }
        }
        printf("\n");
    }
}

```

```

printf("\n");
for (i = 0; i < no_pro; ++i) {
    printf("FIRST OF %s: ", pro[i].str);
    for (j = 0; j < TSIZE; ++j) {
        if (first_rhs[i][j]) {
            printf("%c ", j);
        }
    }
    printf("\n");
}

```

```

terminal['$'] = 1;
terminal['^'] = 0;

```

*//*

```
printf("\n");
printf("\n\t\t\t\t\tTABELA Predictive Parser\t\t\t\t\t*\n");
printf("\t-----\n");
printf("%-10s", "");
for (i = 0; i < TSIZE; ++i) {
    if (terminal[i]) printf("%-10c", i);
}
printf("\n");
int p = 0;
for (i = 0; i < no_pro; ++i) {
    if (i != 0 && (pro[i].str[0] != pro[i - 1].str[0]))
        p = p + 1;
    for (j = 0; j < TSIZE; ++j) {
        if (first_rhs[i][j] && j != '^') {
            table[p][j] = i + 1;
        }
        else if (first_rhs[i]['^']) {
            for (k = 0; k < TSIZE; ++k) {
                if (follow[pro[i].str[0] - 'A'][k]) {
                    table[p][k] = i + 1;
                }
            }
        }
    }
}
k = 0;
for (i = 0; i < no_pro; ++i) {
    if (i == 0 || (pro[i - 1].str[0] != pro[i].str[0])) {
        printf("%-10c", pro[i].str[0]);
        for (j = 0; j < TSIZE; ++j) {
            if (table[k][j]) {
                printf("%-10s", pro[table[k][j] - 1].str);
            }
            else if (terminal[j]) {
                printf("%-10s", "");
            }
        }
        ++k;
        printf("\n");
    }
}
```

## OUTPUT:

```
PROBLEMS 1 PORTS COMMENTS DEBUG CONSOLE TERMINAL OUTPUT
◆ PS C:\Users\Dell\Documents\sem7\CdL> cd "c:\Users\Dell\Documents\sem7\CdL\" ; if
}
E->TA
A->+TA|^
T->FB
B->*FB|^
F->t|(E)
FIRST OF E: ( t
FIRST OF A: + ^
FIRST OF T: ( t
FIRST OF B: * ^
FIRST OF F: ( t

FOLLOW OF E: $ * +
FOLLOW OF A: $ * +
FOLLOW OF T: $ * +
FOLLOW OF B: $ * +
FOLLOW OF F: $ * +

FIRST OF E->TA: ( t
FIRST OF A->+TA: +
FIRST OF A->^: ^
FIRST OF T->FB: ( t
FIRST OF B->*FB: *
FIRST OF B->^: ^
FIRST OF F->t: t
FIRST OF F->(E: (

***** TABELA Predictive Parser *****
-----
      $      (      *      +      t
E      E->TA      E->TA      E->TA      E->TA
A      A->^      A->^      A->^      A->^
T      T->FB      T->FB      T->FB      T->FB
B      B->^      B->^      B->^      B->^
F      F->(E      F->t      F->t      F->t
○ PS C:\Users\Dell\Documents\sem7\CdL>
```

Conclusion:

---

---

---

Name and Sign of Course Teacher  
Mrs. Archana Chitte

# Government College of Engineering, Jalgaon

(An Autonomous Institute of Govt. of Maharashtra)

Name of Student:.....

PRN:.....

Date of Performance:.....

Date of Completion:.....

---

## Experiment No 9

**Title:** Write a program to check whether a string belongs to a grammar or not

**Theory:**

**Sample Program:**

```
/*
Aim: To check whether string is accepted by grammar or not.
*/
#include<string.h>
#include<stdio.h>
int main()
{
    char string[20];
    int state=0, count=0;
    printf("\n The string must begin with a and terminate with b");
    printf("\n The Grammar Is:\n");
    printf("\tS->aS \n\tS->b \n\tS->ab\n");
    printf("Enter The String To Be Checked\n");
    gets(string);
    while(string[count]!='\0')
    {
        switch(state)
        {
            case 0: if (string[count]=='a')
                    state=1;
                    else
                    state=3;
                    break;
            case 1: if (string[count]=='a')
                    state=1;
                    else if(string[count]=='b')
                    state=2;
                    else
                    state=3;
                    break;
            case 2: if (string[count]=='b')
                    state=2;
                    else
                    state=3;
                    break;
            default: break;
        }
        count++;
    }
```

```

        if(state==3)
            break;
    }
    if(state==2)
        printf("\nstring is accepted\n");
    else
        printf("\nstring is not accepted\n");
    return(0);
}

```

## OUTPUT

```

}

The string must begin with a and terminate with b
The Grammar Is:
    S->aS
    S->Sb
    S->ab
Enter The String To Be Checked
aabb

string is accepted
PS C:\Users\Dell\Documents\sem7\CdL> cd "c:\Users\Dell\Documents
}

The string must begin with a and terminate with b
The Grammar Is:
    S->aS
    S->Sb
    S->ab
Enter The String To Be Checked
aba

string is not accepted
PS C:\Users\Dell\Documents\sem7\CdL>

```

Conclusion:

---



---



---

Name and Sign of Course Teacher  
Mrs. Archana Chitte



**Government College of Engineering, Jalgaon**  
(An Autonomous Institute of Govt. of Maharashtra)

**Name of Student:**.....

**PRN:**.....

**Date of Performance:**.....

**Date of Completion:**.....

---

**Experiment No 10**

**Title:** Implement Deterministic Finite Automata

**Theory:** Finite Automaton can be classified into two types –

- Deterministic Finite Automaton (DFA)
- Non-deterministic Finite Automaton (NFA / NFA)

**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.

**Formal Definition of a DFA**

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

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

**Graphical Representation of a DFA**

A DFA is represented by digraphs called state diagram.

- The vertices represent the states.
- The arcs labeled with an input alphabet show the transitions.
- The initial state is denoted by an empty single incoming arc.
- The final state is indicated by double circles.

**Example**

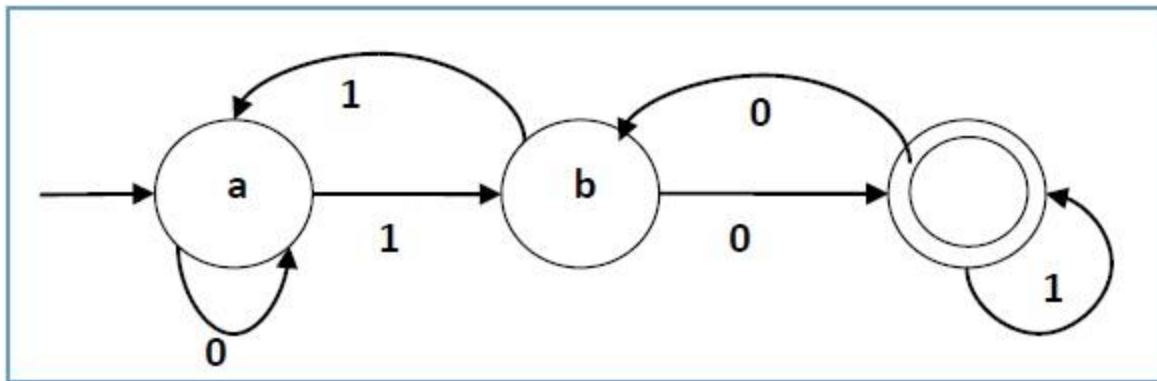
Let a deterministic finite automaton be  $\rightarrow$

- $Q = \{a, b, c\}$ ,
- $\Sigma = \{0, 1\}$ ,
- $q_0 = \{a\}$ ,
- $F = \{c\}$ , and

Transition function  $\delta$  as shown by the following table –

| Present State | Next State for Input 0 | Next State for Input 1 |
|---------------|------------------------|------------------------|
| a             | a                      | b                      |
| b             | c                      | a                      |
| c             | b                      | c                      |

Its graphical representation would be as follows –



### Sample Program:

```

#include <stdio.h>
#include <stdlib.h>

struct node{
    int id_num;
    int st_val;
    struct node *link0;
    struct node *link1;
};
struct node *start, *q, *ptr;
int vst_arr[100], a[10];
int main(){
    int count, i, posi, j;
    char n[10];

    printf("-----\n");
    printf("Enter the number of states in the m/c:");
    scanf("%d",&count);

    q=(struct node *)malloc(sizeof(struct node)*count);

    for(i=0;i<count;i++){
        (q+i)->id_num=i;

        printf("State Machine::%d\n",i);
        printf("Next State if i/p is 0:");
        scanf("%d",&posi);
        (q+i)->link0=(q+posi);

        printf("Next State if i/p is 1:");
        scanf("%d",&posi);
        (q+i)->link1=(q+posi);

        printf("Is the state final state(0/1)?");
        scanf("%d",&(q+i)->st_val);
    }

    printf("Enter the Initial State of the m/c:");
    scanf("%d",&posi);
    start=q+posi;
    printf("-----\n");
    while(1){

```

```

printf("-----\n");
printf("Perform String Check(0/1):");
scanf("%d",&j);
if(j){
    ptr=start;
    printf("Enter the string of inputs:");
    scanf("%s",n);
    posi=0;

    while(n[posi]!='\0'){
        a[posi]=(n[posi]-'0');
        //printf("%c\n",n[posi]);
        //printf("%d",a[posi]);
        posi++;
    }
    i=0;
    printf("The visited States of the m/c are:");
    do{
        vst_arr[i]=ptr->id_num;
        if(a[i]==0){
            ptr=ptr->link0;
        }
        else if(a[i]==1){
            ptr=ptr->link1;
        }
        else{
            printf("iNCORRECT iNPUT\n");
            return;
        }
        printf("[%d]",vst_arr[i]);
        i++;
    }while(i<posi);

    printf("\n");
    printf("Present State:%d\n",ptr->id_num);
    printf("String Status:: ");
    if(ptr->st_val==1)
        printf("String Accepted\n");
    else
        printf("String Not Accepted\n");
    }
    else
        return 0;
}
printf("-----\n");
return 0;
}

```

## OUTPUT :

```
PROBLEMS 1 PORTS COMMENTS DEBUG CONSOLE TERMINAL OUTPUT
Enter the number of states in the m/c:2
State Machine::0
Next State if i/p is 0:1
Next State if i/p is 1:0
Is the state final state(0/1)?1
State Machine::1
Next State if i/p is 0:1
Next State if i/p is 1:0
Is the state final state(0/1)?1
Enter the Initial State of the m/c:0
Perform String Check(0/1):1
Enter the string of inputs:00
The visited States of the m/c are:[0][1]
Present State:1
String Status:: String Accepted
Perform String Check(0/1):1
Enter the string of inputs:001
The visited States of the m/c are:[0][1][1]
Present State:0
String Status:: String Accepted
Perform String Check(0/1):0
PS C:\Users\Dell\Documents\sem7\CdL>
```

Conclusion:

---

---

---

Name and Sign of Course Teacher  
Mrs. Archana Chitte