

**NETAJI SUBHASH UNIVERSITY OF TECHNOLOGY**  
**SECTOR-3, DWARKA,**  
**NEW DELHI-110078**



**Principle of Compiler and Construction**  
**Practical File**  
**COCSC14**  
**Semester-5**

**Name: Aman Kumar**

**Roll No.: 2020UC01539**

**Branch & Section: COE – 1**

## INDEX

| S. No. | Experiment   | Page No. | Signature |
|--------|--|----------|-----------|
| 1.     | (Tokenizing) Use C programming to extract tokens from a given source code.                   | 3-5      |           |
| 2.     | (Tokenizing) Use LEX to extract tokens from a given source code.                             | 6-7      |           |
| 3.     | Implement YACC for Subset of Fortran 'DO loop' program                                       | 8-9      |           |
| 4.     | Create a digital Calculator using LEX and YACC tools   | 10-11    |           |
| 5.     | Generate the three-address code for a simple arithmetic expression using LEX and YACC tools. | 12-15    |           |
| 6.     | Write a program to remove left recursion from a grammar.                                     | 16-18    |           |
| 7.     | Write a program to remove left factoring from a given grammar.                               | 19-20    |           |
| 8.     | Write a program to find first and follow of the grammar symbols from a given grammar.        | 21-26    |           |

# Experiment 1

## Tokenization using C

**AIM:** (Tokenizing) Use C programming to extract tokens from a given source code.

### THEORY:

Lexical Analysis is the first phase of the compiler also known as a scanner. It converts the High level input program into a sequence of Tokens.

- Lexical Analysis can be implemented with the Deterministic finite Automata.
- The output is a sequence of tokens that is sent to the parser for syntax analysis

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. 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

### Code:

```
#include <stdbool.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
bool delimiter(char *s)
{
    if (!strcmp(s, " ") || !strcmp(s, "+") || !strcmp(s, "-") || !strcmp(s, "*") ||
    !strcmp(s, "/") || !strcmp(s, ",") || !strcmp(s, ";") || !strcmp(s, ">") || !strcmp(s,
    "<") || !strcmp(s, "=") || !strcmp(s, "(") || !strcmp(s, ")") ||
    !strcmp(s, "[") || !strcmp(s, "]") || !strcmp(s, "{") || !strcmp(s, "}"))
    {
        return true;
    }
    return false;
}
bool keyword(char *s)
{
    if (!strcmp(s, "if") || !strcmp(s, "else") || !strcmp(s, "while") || !strcmp(s,
    "do") || !strcmp(s, "break") || !strcmp(s, "continue") || !strcmp(s, "int") ||
    !strcmp(s, "double") || !strcmp(s, "float") ||
    !strcmp(s, "return") || !strcmp(s, "char") || !strcmp(s, "case") || !strcmp(s,
    "char") || !strcmp(s, "sizeof") || !strcmp(s, "long") || !strcmp(s, "short") ||
    !strcmp(s, "typedef") || !strcmp(s, "switch") ||
    !strcmp(s, "unsigned") || !strcmp(s, "void") || !strcmp(s, "static") ||
    !strcmp(s, "struct") || !strcmp(s, "goto"))
    {
        return true;
    }
    return false;
}
bool operator(char ch)
{
    if (ch == '+' || ch == '-' || ch == '*' || ch == '/' || ch == '>' || ch == '<' ||
    ch == '=')
```

```

    {
        return true;
    }
    return false;
}

bool identifier(char *s)
{
    if (s[0] == '0' || s[0] == '1' || s[0] == '2' || s[0] == '3' || s[0] == '4' ||
s[0] == '5' || s[0] == '6' || s[0] == '7' || s[0] == '8' || s[0] == '9')
    {
        return false;
    }
    return true;
}

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

int main()
{
    printf("Enter the file name : ");
    char filename[30];
    scanf("%[^\\n]%c", filename);
    FILE *myfile = fopen("input.txt", "r");
    if (NULL == myfile)
    {
        printf("File can't be opened \\n");
    }
    char codeline[50];
    int line_number = 1;
    while (fgets(codeline, 50, myfile) != NULL)
    {
        printf("Line : %d\\n", line_number);
        char *s;
        char *rest = codeline;
        while ((s = strtok_r(rest, " ", &rest)))
        {
            if (s == "")
            {
                continue;
            }
        }
    }
}

```

```

        printf("\t");
        if (delimiter(s))
            printf("%s is a Delimiter", s);
        else if (keyword(s))
            printf("%s is a Keyword", s);
        else if (operator(s))
            printf("%s is a Operator", s);
        else if (identifier(s))
            printf("%s is a Identifier", s);
        else if (integer(s))
            printf("%s is a Integer", s);
        printf("\n");
    }
    line_number++;
}
fclose(myfile);
printf("\n");
return 0;
}

```

### Output:

```

> ./a.out
Enter the file name : input.txt
Line : 1
    # is a Identifier
    include is a Identifier
    " is a Identifier
    iostream is a Identifier
    " is a Identifier
    int is a Keyword
    main is a Identifier
    ( is a Delimiter
    ) is a Delimiter
    {
is a Identifier
Line : 2
    int is a Keyword
    a is a Identifier
    , is a Delimiter
    b is a Identifier
    ;
is a Identifier
Line : 3
    int is a Keyword
    c is a Identifier
    = is a Delimiter
    6 is a Integer
    * is a Delimiter
    a is a Identifier
    + is a Delimiter
    b is a Identifier
    ; is a Delimiter
    return is a Keyword
    0 is a Integer
    ;
is a Identifier
Line : 4
    } is a Delimiter

```

## Experiment 2

### Tokenization using Lex Tool

**AIM:** (Tokenizing) Use LEX to extract tokens from a given source code.

#### **THEORY:**

##### **LEXICAL TOOLS**

- A language for specifying lexical analyzer.
- There is a wide range of tools for construction of lexical analyzers. The majority of these tools are based on regular expressions.
- One of the traditional tools of that kind is lex.

##### **LEX:**

- The lex is used in the manner depicted. A specification of the lexical analyzer is preferred by creating a program lex.l in the lex language.
- Then lex.l is run through the lex compiler to produce a 'c' program lex.yy.c.
- The program lex.yy.c consists of a tabular representation of a transition diagram constructed from the regular expression of lex.l together with a standard routine that uses a table of recognized lexemes.
- Lex.yy.c is run through the 'C' compiler to produce an object program a.out, which is the lexical analyzer that transforms an input stream into sequence of tokens.

##### **Lex Code:**

```
%{
    int commentPresent = 0;
}%

%%
#.* {printf("\n %s is a Preprocessor Directive",yytext);}
if|else|while|do|break|continue|int|double|float|return|char|case|long|short|typedef|s
witch|unsigned|void|static|struct|goto|for { printf("\n Keyword: %s ",yytext);}
"/*" {commentPresent = 1;}
"*/" {commentPresent = 0;}
[a-zA-Z][a-zA-Z0-9]*(\[0-9*\])? {if(!commentPresent) printf("\n Identifier:
%s",yytext);}
"+"|"-"|"*"|"/" {printf("\n Arithmetic Operator: %s ",yytext);}
","|";"|"(")"|"["]" { printf("\n Delimiter:%s",yytext);}
"=" {printf("\n Assignment Operator");}
 "{" {if(!commentPresent) printf("\n Block Begins");}
 "}" {if(!commentPresent) printf("\n Block Ends");}

\".*" {if(!commentPresent) printf("\n %s is a String",yytext);}
[0-9]+ {if(!commentPresent) printf("\n %s is a Number",yytext);}
"<=" | ">=" | "<" | "==" {if(!commentPresent) printf("\n %s is a Relational
Operator",yytext);}
%%

int main(int argc, char **argv) {
    yylex();
    return 0;
}
int yywrap() {
    return 0;
}
```

##### **Output:**

```
> lex checkick.l
> gcc lex.yy.c
> ./a.out
#include<stdio.h>

#include<stdio.h> is a Preprocessor Directive
main(){

Identifier: main
Delimiter:(
Delimiter:)
Block Begins
int a=0;

Keyword: int
Identifier: a
Assignment Operator
0 is a Number
Delimiter;;
}

Block Ends
^C
> Aman Kumar - 2020UC01539
```

## Experiment 3

### Fortran Do Loop Program

**AIM:** Implement YACC for Subset of Fortran 'DO loop' program

#### THEORY:

Syntax Analysis:

- A syntax analyzer or parser takes the input from a lexical analyzer in the form of token streams.
- The parser analyzes the source code (token stream) against the production rules to detect any errors in the code.
- The output of this phase is a parse tree.

YACC:

- YACC (yet another compiler-compiler) is an LALR(1) (LookAhead, Left-to-right, Rightmost derivation producer with 1 lookahead token) parser generator.
  - The YACC input file is divided into three parts.

```
/* definitions */ ....
%%
/* rules */ .... %%
/* auxiliary routines */ ....
```
- The definition part includes information about the tokens used in the syntax definition
- The rules part contains grammar definitions in a modified BNF form. Actions is C code in { } and can be embedded inside (Translation schemes).
- The auxiliary routines part is only C code. It includes function definitions for every function needed in rules part. It can also contain the main() function definition if the parser is going to be run as a program. The main() function must call the function yyparse().

#### Lex Code:

```
%{
    #include "y.tab.h"
    extern int yyval;
}%

%%

do { return (DO); }
= { return (EQU); }
, { return (SEP); }
end { return (END); }
" " { return (WS); }
\n { return (NEW); }
[a-zA-Z]+ { yyval=yytext[0];return(ID); }
[0-9]+ { yyval=atoi(yytext);return(NUM); }
[\\s\\S]* {return (STAT);}

%%

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

Yacc Code:



```

%{
    #include <stdio.h>
    int flag=0;
}%
%token DO EQU SEP ID NUM END NEW STAT WS;
%%
S: DO WS ID EQU E1 SEP E1 SEP E1 WS END WS DO {printf("Accepted!"); flag=1;}
E1: ID | NUM ;
%%
int main() {
    yyparse();
    return 1;
}
yyerror(const char *msg) {
    if(flag==0) {
        printf("Not Accepted\n");
    }
}

```

**Output:**

```

> lex fortranloop.l
> yacc fortranloop.y -d
> gcc lex.yy.c y.tab.c -w
> ./a.out
do i=1,10,1 end do
Accepted!%

```

## Experiment 4

### Calculator using LEX and YACC

**AIM:** Create a digital Calculator using LEX and YACC tools

#### Lex Code:

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

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

%%

yywrap()
{
    return 1;
}
```

#### Yacc Code:

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

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

%%

S: 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, Division,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;
}

```

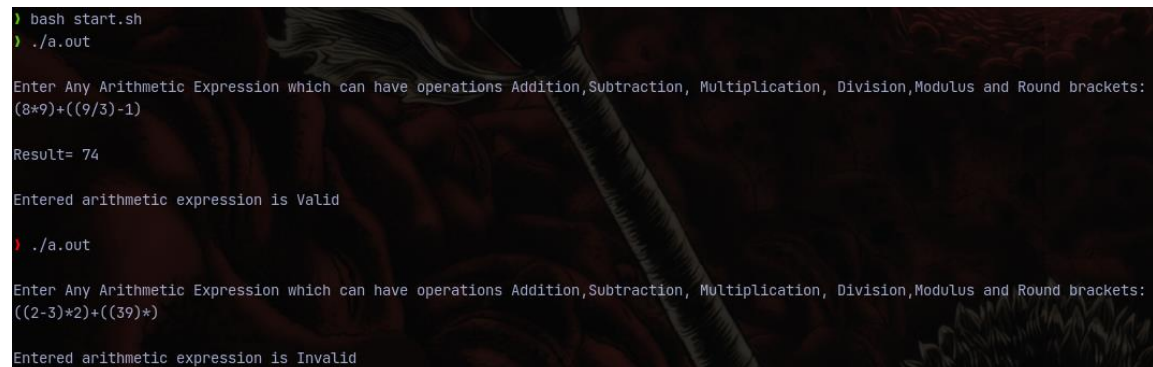
#### Start.sh Code:

```

lex calci.l
yacc calci.y -d
gcc y.tab.c lex.yy.c

```

#### Output:



```

) bash start.sh
) ./a.out

Enter Any Arithmetic Expression which can have operations Addition,Subtraction, Multiplication, Division,Modulus and Round brackets:
(8*9)+((9/3)-1)

Result= 74

Entered arithmetic expression is Valid

) ./a.out

Enter Any Arithmetic Expression which can have operations Addition,Subtraction, Multiplication, Division,Modulus and Round brackets:
((2-3)*2)+((39)*

Entered arithmetic expression is Invalid

```

## Experiment -5

### Intermediate Code Generator

**AIM :** Generate the three address and four address intermediate code source

#### **THEORY:**

**INTERMEDIATE CODE :** If a compiler translates the source language to its target machine language without having the option for generating intermediate code, then for each new machine, a full native compiler is required.

Intermediate code eliminates the need of a new full compiler for every unique machine by keeping the analysis portion the same for all the compilers.

The second part of the compiler, synthesis, is changed according to the target machine.

It becomes easier to apply the source code modifications to improve code performance by applying code optimization techniques on the intermediate code.

#### **THREE ADDRESS CODE:**

Intermediate code generator receives input from its predecessor phase, semantic analyser, in the form of an annotated syntax tree. That syntax tree then can be converted into a linear representation, e.g., postfix notation. Intermediate code tends to be machine independent code. Therefore, code generators assume an unlimited number of memory storage (registers) to generate code.

#### **Lex Code:**

```
%{
#include "y.tab.h"
%}

%%

[0-9]+ {yy1val.symbol = (char)(yytext[0]); return NUMBER;}
[a-z] {yy1val.symbol = (char)(yytext[0]); return LETTER;}
. {return yytext[0];}
\n {return 0;}
%%
```

#### **Yacc Code:**

```
%{
#include <stdio.h>
#include <ctype.h>
char addtotable(char, char, char);

int index1=0;
char temp = 'A'-1;

struct expr{
char operand1;
char operand2;
char operator;
char result;
```

```

};

%}

%union{
char symbol;
}

%left '+' '-'
%left '/' '*'

%token <symbol> LETTER NUMBER
%type <symbol> exp

%%
statement: LETTER '=' exp ';' {addtotable((char)$1,(char)$3,'=');};
exp: exp '+' exp {$$ = addtotable((char)$1,(char)$3,'+');}
    | exp '-' exp {$$ = addtotable((char)$1,(char)$3,'-');}
    | exp '/' exp {$$ = addtotable((char)$1,(char)$3,'/');}
    | exp '*' exp {$$ = addtotable((char)$1,(char)$3,'*');}
    | '(' exp ')' {$$ = (char)$2;}
    | NUMBER {$$ = (char)$1;}
    | LETTER {(char)$1};

%%

struct expr arr[20];

void yyerror(char *s){
printf("Error %s",s);

}

char addtotable(char a, char b, char o){
temp++;
arr[index1].operand1=a;
arr[index1].operand2=b;
arr[index1].operator=o;
arr[index1].result = temp;
index1++;
return temp;
}

void threeAdd()
{
int i=0;
char temp='A';
while(i<index1) {
printf("%c:=\t",arr[i].result);
printf("%c\t",arr[i].operand1);
printf("%c\t",arr[i].operator);
printf("%c\t",arr[i].operand2);
i++;
}
}

```

```

temp++;
printf("\n");

}
}
int find(char l){
int i;
for(i=0;i<index1;i++)
{
if(arr[i].result==1) break;
}
return i;
}

void triple(){
int i=0;
char temp='A';
while(i<index1) {
printf("%c\t",arr[i].operator);
if(!isupper(arr[i].operand1))
printf("%c\t",arr[i].operand1);
else{
printf("pointer");
printf("%d\t",find(arr[i].operand1));

}
if(!isupper(arr[i].operand2))
printf("%c\t",arr[i].operand2);
else{
printf("pointer");
printf("%d\t",find(arr[i].operand2));
}
i++;
temp++;
printf("\n");

}
}

int yywrap()
{
return 1;
}

int main()
{
printf("Enter the expression \n");
yyvsparse();
threeAdd();
printf("\n");
triple();
return 0;
}

```

### Output:

```
meister@Meister:~/test_compiler/1
> lex icg.l
> yacc icg.y -d
> gcc lex.yy.c y.tab.c -w
> ./a.out
Enter the expression
a=9*9+b*10/19*c+1-d/10;
A:= 9 * 9
B:= b * 1
C:= B / 1
D:= C * c
E:= A + D
F:= E + 1
G:= d / 1
H:= F - G
I:= a = H

* 9 9
* b 1
/ pointer1 1
* pointer2 c
+ pointer0 pointer3
+ pointer4 1
/ d 1
- pointer5 pointer6
= a pointer7
> Aman Kumar -2020uco1539
```

## Experiment-6

### Removing Left recursion

**Aim:** Write a program to remove left recursion from a grammar.

**Theory:** A Grammar  $G (V, T, P, S)$  is left recursive if it has a production in the form.

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

The above Grammar is left recursive because the left of production is occurring at a first position on the right side of production. It can eliminate left recursion by replacing a pair of production with

$$A \rightarrow \beta A'$$

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

This type of recursion is also called Immediate Left Recursion.

In Left Recursive Grammar, expansion of  $A$  will generate  $A\alpha$ ,  $A\alpha\alpha$ ,  $A\alpha\alpha\alpha$  at each step, causing it to enter an infinite loop

The general form for left recursion is

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$$

can be replaced by

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_n A'$$

$$A \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_m A' \mid \epsilon$$

**Code:**

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;

int main()
{
    int n;
    cout<<"\nEnter number of non terminals: ";
    cin>>n;
    cout<<"\nEnter non terminals one by one: ";
    int i;
    vector<string> nonter(n);
    vector<int> leftrecr(n,0);
    for(i=0;i<n;++i) {
        cout<<"\Non terminal "<<i+1<<" : ";
        cin>>nonter[i];
    }
    vector<vector<string> > prod;
    cout<<"\nEnter 'esp' for null";
    for(i=0;i<n;++i) {
        cout<<"\nNumber of "<<nonter[i]<<" productions: ";
        int k;
        cin>>k;
        int j;
        cout<<"\nOne by one enter all "<<nonter[i]<<" productions";
        vector<string> temp(k);
        for(j=0;j<k;++j) {
            cout<<"\nRHS of production "<<j+1<<" : ";
            string abc;
            cin>>abc;
```



```

        temp[j]=abc;
        if(nonter[i].length()<=abc.length()&&nonter[i].compare(abc.substr(0,nonter
[i].length()))==0)
            leftrecr[i]=1;
    }
    prod.push_back(temp);
}
for(i=0;i<n;++i) {
    cout<<leftrecr[i];
}
for(i=0;i<n;++i) {
    if(leftrecr[i]==0)
        continue;
    int j;
    nonter.push_back(nonter[i]+"");
    vector<string> temp;
    for(j=0;j<prod[i].size();++j) {
        if(nonter[i].length()<=prod[i][j].length()&&nonter[i].compare(prod[i][j].s
ubstr(0,nonter[i].length()))==0) {
            string abc=prod[i][j].substr(nonter[i].length(),prod[i][j].length()-
nonter[i].length()+nonter[i]+"");
            temp.push_back(abc);
            prod[i].erase(prod[i].begin()+j);
            --j;
        }
        else {
            prod[i][j]+=nonter[i]+"";
        }
    }
    temp.push_back("esp");
    prod.push_back(temp);
}
cout<<"\n\n";
cout<<"\nNew set of non-terminals: ";
for(i=0;i<nonter.size();++i)
    cout<<nonter[i]<<" ";
cout<<"\n\nNew set of productions: ";
for(i=0;i<nonter.size();++i) {
    int j;
    for(j=0;j<prod[i].size();++j) {
        cout<<"\n"<<nonter[i]<<" -> "<<prod[i][j];
    }
}
return 0;
}

```

Output:

```
> g++ left-recurr.cpp  
> ./a.out
```

Enter number of non terminals: 1

Enter non terminals one by one:

Non terminal 1 :S

Enter 'esp' for null

Number of S productions: 4

One by one enter all S productions

RHS of production 1: Sa

RHS of production 2: Sb

RHS of production 3: Ab

RHS of production 4: Sc

1

New set of non-terminals: S S'

New set of productions:

S -> AbS'

S' -> aS'

S' -> bS'

S' -> cS'

## Experiment-7

### Removing Left Factoring

**Aim:** Write a program to remove left factoring from a given grammar.

**Theory:** Left factoring is removing the common left factor that appears in two productions of the same non-terminal. It is done to avoid back-tracing by the parser. Suppose the parser has a look-ahead, consider this example:

$A \rightarrow qB \mid qC$

where A, B and C are non-terminals and q is a sentence.

In this case, the parser will be confused as to which of the two productions to choose and it might have to back-trace. After left factoring, the grammar is converted to:

$A \rightarrow qD$

$D \rightarrow B \mid C$

In this case, a parser with a look-ahead will always choose the right production.

**Code:**

```
#include <iostream>
#include <math.h>
#include <vector>
#include <string>
#include <stdlib.h>
using namespace std;

int main()
{
    cout<<"\nEnter number of productions: ";
    int p;
    cin>>p;
    vector<string> prodleft(p),prodrigh(p);
    cout<<"\nEnter productions one by one: ";
    int i;
    for(i=0;i<p;++i) {
        cout<<"\nLeft of production "<<i+1<<": ";
        cin>>prodleft[i];
        cout<<"\nRight of production "<<i+1<<": ";
        cin>>prodrigh[i];
    }
    int j;
    int e=1;
    for(i=0;i<p;++i) {
        for(j=i+1;j<p;++j) {
            if(prodleft[j]==prodleft[i]) {
                int k=0;
                string com="";
                while(k<prodrigh[i].length()&&k<prodrigh[j].length()&&prodrigh[i][k]
]==prodrigh[j][k]) {
                    com+=prodrigh[i][k];
                    ++k;
                }
                if(k==0)
                    continue;
                char* buffer;
                string comleft=prodleft[i];
```

```

string es = to_string(e);
if(k==prodright[i].length()) {
    prodleft[i]+=string(es);
    prodleft[j]+=string(es);
    prodright[i]="^";
    prodright[j]=prodright[j].substr(k,prodright[j].length()-k);
}
else if(k==prodright[j].length()) {
    prodleft[i]+=string(es);
    prodleft[j]+=string(es);
    prodright[j]="^";
    prodright[i]=prodright[i].substr(k,prodright[i].length()-k);
}
else {
    prodleft[i]+=string(es);
    prodleft[j]+=string(es);
    prodright[j]=prodright[j].substr(k,prodright[j].length()-k);
    prodright[i]=prodright[i].substr(k,prodright[i].length()-k);
}
int l;
for(l=j+1;l<p;++l) {
    if(comleft==prodleft[l]&&com==prodright[l].substr(0,fmin(k,prodrig
ht[l].length())) {
        prodleft[l]+=string(es);
        prodright[l]=prodright[l].substr(k,prodright[l].length()-k);
    }
}
prodleft.push_back(comleft);
prodright.push_back(com+prodleft[i]);
++p;
++e;
}
}
}
cout<<"\n\nNew productions";
for(i=0;i<p;++i) {
    cout<<"\n"<<prodleft[i]<<"->"<<prodright[i];
}
return 0;
}

```

### Output:

```

> g++ left-factor.cpp
> ./a.out

Enter number of productions: 2

Enter productions one by one:
Left of production 1: A

Right of production 1: aB

Left of production 2: A

Right of production 2: aC

New productions
A1->B
A1->C
A->aA1

```

## Experiment-8

### First and Follow

Aim: Write a program to find first and follow of the grammar symbols from a given grammar.

Theory:

FIRST and FOLLOW are two functions associated with grammar that help us fill in the entries of an M-table.

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 (possible 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 FIRST () and 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.

#### Code:

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

void followfirst(char, int, int);
void follow(char c);
void findfirst(char, int, int);
int count, n = 0;
char calc_first[10][100];
char calc_follow[10][100];
int m = 0;

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;

    strcpy(production[0], "S=(L)");
    strcpy(production[1], "S=a");
    strcpy(production[2], "L=SY");
    strcpy(production[3], "Y=*SY");
    strcpy(production[4], "Y=#");

    int kay;
```

```

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

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;

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

    if (xxx == 1)
        continue;

    findfirst(c, 0, 0);
    ptr += 1;
    done[ptr] = c;
    printf("\n First(%c) = { ", c);
    calc_first[point1][point2++] = c;

    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;
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;

    for(kay = 0; kay <= ptr; kay++)
        if(ck == donee[kay])
            xxx = 1;

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

    // Function call
    follow(ck);
    ptr += 1;
    donee[ptr] = ck;
    printf(" Follow(%c) = { ", ck);
    calc_follow[point1][point2++] = ck;

    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;
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])
            {
                follow(production[i][0]);
            }
        }
    }
}

void findfirst(char c, int q1, int q2)
{
    int j;
    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))
                {
                    findfirst(production[q1][q2], q1, (q2+1));
                }
            }
            else
                first[n++] = '#';
        }
        else if(!isupper(production[j][2]))
        {
            first[n++] = production[j][2];
        }
        else
        {

```



```

        findfirst(production[j][2], j, 3);
    }
}
}

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

    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;
        }
        while(calc_first[i][j] != '!')
        {
            if(calc_first[i][j] != '#')
            {
                f[m++] = calc_first[i][j];
            }
            else
            {
                if(production[c1][c2] == '\\0')
                {
                    follow(production[c1][0]);
                }
                else
                {
                    followfirst(production[c1][c2], c1, c2+1);
                }
            }
            j++;
        }
    }
}
}
}

```

Output:

```
> gcc first-follow.c  
> ./a.out
```

```
First(S) = { (, a, }
```

```
First(L) = { (, a, }
```

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

```
First() = { }
```

```
-----
```

```
Follow(S) = { $, *, ), }
```

```
Follow(L) = { ), }
```

```
Follow(Y) = { ), }
```

```
Follow(*) = { $, ), (, a, =, }
```