Compiler Design Lab

Practical File

Faculty Name: Ms. Garima Gupta

Student's Name: Kshitij Tanwar

Roll No.: 03114812721

Semester: 5

Group: 5 CST1

-

Maharaja Agrasen Institute Of Technology

Sector – 22, Rohini, New Delhi - 110085

INDEX

Name: Kshitij Tanwar

Enrolment Number: 03114812721

Branch: Computer Science and Technology

Group: 5 CST1

S. No.	Experiment	Date of Performance	Date of Checking	Signature	Remark

EXPERIMENT-1

AIM: Practice of LEX/YACC of compiler writing.

THEORY: Lex is a tool or a computer program that generates Lexical Analyzers (converts the stream of characters into tokens). The Lex tool itself is a compiler. The Lex compiler takes the input and transforms that input into input patterns. It is commonly used with YACC(Yet Another Compiler Compiler). It was written by Mike Lesk and Eric Schmidt.

YACC (for "yet another compiler compiler.") is the standard parser generator for the Unix operating system. An open-source program, Yacc generates code for the parser in the O programming language. The acronym is usually rendered in lowercase but is occasionally seen as YACC or Yacc.

SOURCE CODE:

1) LEX Program to count number of vowels and consonants

```
%{
	int vow_count=0;
	int const_count =0;
%}

%%
[aeiouAEIOU] {vow_count++;}
[a-zA-Z] {const_count++;}
%%
int yywrap(){}
int main()
{
	printf("Enter the string of vowels and consonants:");
	yylex();
	printf("Number of vowels are: %d\n", vow_count);
	printf("Number of consonants are: %d\n", const_count);
	return 0;
}
```

OUTPUT:

```
Microsoft Windows [Version 10.0.22621.2283]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Kshitij\OneDrive\Desktop\LF>flex a.l.txt

C:\Users\Kshitij\OneDrive\Desktop\LF>gcc lex.yy.c

C:\Users\Kshitij\OneDrive\Desktop\LF>a.exe
Enter the string of vowels and consonants:KshitijTanwar

[redacted]

Number of vowels are: 4
Number of consonants are: 9
```

2) YACC Program to implement a Calculator and recognize a valid Arithmetic Expression

LEXICAL ANALYZER SOURCE CODE

```
%{

/* Definition section */

#include<stdio.h>

#include "y.tab.h"

extern int yylval;

%}

/* Rule Section */

%%

[0-9]+ {

                yylval=atoi(yytext);

                return NUMBER;

        }
```

```
[\t] ;
[\n] return 0;
. return yytext[0];
%%
int yywrap() {
return 1;
}
```

PARSER SOURCE CODE

```
%{
/* Definition section */
#include<stdio.h>
int flag=0;
%}
%token NUMBER
%left '+' '-'
%left '*' '/' '%'
%left '(' ')'
/* Rule Section */
%%
ArithmeticExpression: E{
          printf("\nResult=%d\n", $$);
          return 0;
          };
E:E'+'E {$$=$1+$3;}
```

```
|E'-'E {$$=$1-$3;}

|E'*'E {$$=$1*$3;}

|E'/'E {$$=$1/$3;}

|E'%'E {$$=$1%$3;}

|'('E')' {$$=$2;}

| NUMBER {$$=$1;}

;

%%

//driver code

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("\nEntered arithmetic expression is Valid\n\n");

}

void yyerror() {

printf("\nEntered arithmetic expression is Invalid\n\n");

flag=1;

}
```

OUTPUT:

```
C:\Users\Kshitij\OneDrive\Desktop\LF\yacc>a.exe

Enter Any Arithmetic Expression which can have operations Addition, Subtraction, Multiplication, Divison, Modulus and Round brackets:

10-5

Result=5

Entered arithmetic expression is Valid

C:\Users\Kshitij\OneDrive\Desktop\LF\yacc>a.exe

Enter Any Arithmetic Expression which can have operations Addition, Subtraction, Multiplication, Divison, Modulus and Round brackets:

10+5-

Entered arithmetic expression is Invalid

C:\Users\Kshitij\OneDrive\Desktop\LF\yacc>a.exe

Enter Any Arithmetic Expression which can have operations Addition, Subtraction, Multiplication, Divison, Modulus and Round brackets:

10/5

Result=2

Entered arithmetic expression is Valid
```

```
C:\Users\Kshitij\OneDrive\Desktop\LF\yacc>a.exe

Enter Any Arithmetic Expression which can have operations Addition, Subtraction, Multiplication, Divison, Modulus and Round brackets:

(2+5)*3

Result=21

Entered arithmetic expression is Valid

C:\Users\Kshitij\OneDrive\Desktop\LF\yacc>a.exe

Enter Any Arithmetic Expression which can have operations Addition, Subtraction, Multiplication, Divison, Modulus and Round brackets:

12%5

Result=2

Entered arithmetic expression is Valid
```

<u>EXPERIMENT -2</u>

AIM: Write a program to check whether a string belong to the grammar or not.

Theory: Context Free Grammar is a formal grammar; the syntax structure of a formal language can be described using context-free grammar (CFG) a type of formal grammar. The grammar has 4 tuples: (V, T, P, S).

V - It is the collection of variables or nonterminal symbols.

T - It is a set of terminals.

P - It Is the production rule that consists of both terminals and non-terminals.

S – It is the starting symbol.

A grammar is said to be Context-free grammar if every production is in the form of:

$$G \rightarrow (V \cup T)^* \text{ , where } G \in V$$

And the left-hand side of G here in the example can only be a variable. It cannot be a terminal

But on the right-hand side it can be a variable or terminal, or both combination of variable and terminal.

Above equation states that every production which carry any combination of ‘V’ variable or ‘T’ terminal is said to be a context-free grammar.

SOURCE CODE:

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

int length(string str){
    int i=0;
    while (str[i])
        i++;
    return i;
}


void first(){
    char str[100];
    cout<<"\nThe grammar is as follows --> \nS -> aS\nS -> Sb\nS -> ab\n";
    cout<<"Enter a string --> ";
    cin>>str;
    if(str[0]!='a'){
```

```
        cout<<"String is invalid because of incorrect first character!!";
        exit(0);
    }
    int n=1;
    while(str[n]=='a')
        n++;
    if ( str[n] != 'b'){
        cout<<"String does not belong to grammar!!";
        exit(0);
    }
    n++;
    while (str[n]=='b')
        n++;
    if (str[n] != '\0'){
        cout<<"String does not belong to grammar!!";
        exit(0);
    }
    cout<<"String is Valid!!";
}

void second(){
    char str[100];
    cout<<"\nThe grammar is as follows --> \nS -> aSa\nS -> bSb\nS -> a\nS -> b\n";
    cout<<"Enter a string --> ";
    cin>>str;
    if(str[0]!='a'&& str[0]!='b'){
        cout<<"String is invalid because of incorrect first character!!";
        exit(0);
    }
    if(length(str)%2 ==0){
        cout<<"String is invalid because of Even Length!!";
        exit(0);
    }
    int i = 0;
    int j = length(str)-1;
    while(i<length(str) && j>=0){
      if(str[i] != str[j])
            cout<<"String is Invalid!!";
            i++;
            j--;
    }
    cout<<"String belongs to the Grammar!!";
}
```

```
void third(){
    char str[100];
    cout<<"\nThe grammar is as follows --> \nS -> aSbb\nS -> abb\n";
    cout<<"Enter a string --> ";
    cin>>str;
    if(str[0]!='a'){
        cout<<"String is invalid because of incorrect first character!!";
        exit(0);
    }
    int n=1;
    int c=1;
    while(str[n]=='a'){
        n++;
        c++;
    }
    int j = 1;
    if (str[n] != 'b'){
        cout<<"String does not belong to grammar!!";
    exit(0);
    }
    n++;
    while (str[n]=='b'){
        n++;
        j++;
    }
    if (2*c!=j || str[n] != '\0'){
        cout<<"String does not belong to grammar!!";
        exit(0);
    }
    cout<<"String is Valid!!";
}

void fourth(){
    char str[100];
    cout<<"\nThe grammar is as follows --> \nS -> aSb\nS -> ab\n";
    cout<<"Enter a string --> ";
    cin>>str;
    if(str[0]!='a'){
        cout<<"String is invalid because of incorrect first character!!";
        exit(0);
    }
    int n=1;
    int c=1;
    while(str[n]=='a'){
```

```
        n++;
        c++;
    }
    int j = 1;
    if (str[n] != 'b'){
        cout<<"String does not belong to grammar!!";
        exit(0);
    }
    n++;
    while (str[n]=='b'){
        n++;
        j++;
    }
    if (c!=j || str[n]!='\0'){
        cout<<"String does not belong to grammar!!";
        exit(0);
    }
    cout<<"String is Valid!!";
}

int main()
{
    int a;
    cout<<"1 --> \nS -> aS\nS -> Sb\nS -> ab\nString is of the form - aab\n\n";
    cout<<"2 --> \nS -> aSa\nS -> bSb\nS -> a\nS -> b\nThe Language generated is - All Odd Length
Palindromes\n\n";
    cout<<"3 --> \nS -> aSbb\nS -> abb\nThe Language generated is - anb2n , where n>1\n\n";
    cout<<"4 --> \nS -> aSb\nS -> ab\nThe Language generated is - anbn where n>0\n\n";
    cout<<"5 --> Exit!!\n\n";
    do
    {
        cout<<"Choose one input --> ";
        cin>>a;
        switch(a)
        {
            case 1:
            {
                first();
                break;
            }
            case 2:
            {
                second();
                break;
```

```
            }
            case 3:
            {
                third();
                break;
            }
            case 4:
            {
                fourth();
                break;
            }
            default:
                cout<<"Wrong Input !!";
        }
    }
    while (a!=5);
    return 0;
}
```

OUTPUT:

```
1 -->
S -> aS
S -> Sb
S -> ab
String is of the form - aab

2 -->
S -> aSa
S -> bSb
S -> a
S -> b
The Language generated is - All Odd Length Palindromes

3 -->
S -> aSbb
S -> abb
The Language generated is - anb2n , where n>1

4 -->
S -> aSb
S -> ab
The Language generated is - anbn where n>0

5 --> Exit!!

Choose one input --> 1

The grammar is as follows -->
S -> aS
S -> Sb
S -> ab
```

```
Enter a string --> aabbb
String is Valid!!
Choose one input --> 2

The grammar is as follows -->
S -> aSa
S -> bSb
S -> a
S -> b
Enter a string --> abbba
String belongs to the Grammar!!
Choose one input --> 3

The grammar is as follows -->
S -> aSbb
S -> abb
Enter a string --> aabbbb
String is Valid!!
Choose one input --> 4

The grammar is as follows -->
S -> aSb
S -> ab
Enter a string --> aabb
String is Valid!!
Choose one input --> 5
Wrong Input !!
--------------------------------
Process exited after 39.94 seconds with return value 0
Press any key to continue . . .
```

EXPERIMENT-3

AIM: Write a program to check whether a string includes a keyword or not.

THEORY: We break the given string into tokens and check each word if it is a keyword of C++ language. Keywords are predefined words, we cannot name variables as keywords. There are 32 keywords in C++. We print the count of keywords present in the string.

SOURCE CODE:

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

void simple_tokenizer(string s){
        stringstream ss(s);
        string word;
        map<string,int> m;
        string key[32] =
{"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","typede
f","union","unsigned","void","volatile","while"};

        while(ss >> word){
                for(int i=0;i<32;i++){
                        if(key[i] == word){
                                m[word]++;
                        }
                }
        }

        if(!m.empty()){
                for(auto it:m){
                        cout << it.first << " " << it.second << endl;
                }
        }
        else cout << "No keyword present" << endl;
}

int main(){
        string str;
    cout << "Enter a string: ";
    getline(cin,str);
```

```
    simple_tokenizer(str);
        cout << endl;

        cout<< "Raghav Agarwal    01414812721    5CST1" << endl;
    return 0;
}
```

OUTPUT:

```
Enter a string: Hello char Kshitij Tanwar int void auto int bye int
auto 1
char 1
int 3
void 1


Kshitij Tanwar 03114812721 5CST1
```

EXPERIMENT-4

AIM: Write a program to remove left recursion from the grammar

THEORY:

Elimination of Left Recursion

Left Recursion - The generation is left-recursive if the leftmost symbol on the right side is equivalent to the nonterminal on the left side. For Ex: exp → exp + term.

A grammar that contains a production having left recursion is called a Left-Recursive Grammar. Similarly, if the rightmost symbol on the right side is equal to the left side is called Right-Recursion.

Why to Eliminate Left Recursion?

Consider an example: E -> E+T/T,

The above example will go in an infinite loop because the function E keeps calling itself which causes a problem for a parser to go in an infinite loop which is a never ending process so to avoid this infinite loop problem we do Elimination of left recursion.

Rules to follow to eliminate left recursion

A-->bA'

A'--> # / aA'   //# stands for epsilon

Therefore, solution for above left recursion problem,

E --> TE'
E'--> eps/+TE'

SOURCE CODE:

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

using namespace std;


int main(){

    string ip,op1,op2,temp;
```

```
int sizes[10] = {};
char c;
int n,j,l;
cout<<"Enter the Parent Non-Terminal --> ";
cin>>c;
ip.push_back(c);
op1 += ip + "\'->";
ip += "->";
op2+=ip;
cout<<"Enter the Number of Productions --> ";
cin>>n;
for(int i=0;i<n;i++){
    cout<<"Enter Production "<<i+1<<" -> ";
    cin>>temp;
    sizes[i] = temp.size();
    ip+=temp;
    if(i!=n-1)
        ip += "|";
}
cout<<"Production Rule -> "<<ip<<endl;
for(int i=0,k=3;i<n;i++){
    if(ip[0] == ip[k]){
        cout<<"Production "<<i+1<<" has left recursion."<<endl;
        if(ip[k] != '#'){
         for(l=k+1;l<k+sizes[i];l++)
                op1.push_back(ip[l]);
         k=l+1;
```

```
            op1.push_back(ip[0]);
            op1 += "\'|";
           }
        } else{
           cout<<"Production "<<i+1<<" does not have left recursion."<<endl;
           if(ip[k] != '#'){
            for(j=k;j<k+sizes[i];j++)
                    op2.push_back(ip[j]);
            k=j+1;
            op2.push_back(ip[0]);
            op2 += "\'|";
           }  else{
            op2.push_back(ip[0]);
            op2 += "\'";
           }
        }
    }
    op1 += "#";
    cout<<op2<<endl<<op1<<endl;
    return 0;
}
```

OUTPUT:

```
Enter the Parent Non-Terminal --> E
Enter the Number of Productions --> 2
Enter Production 1 -> E+T
Enter Production 2 -> T
Production Rule -> E->E+T|T
Production 1 has left recursion.
Production 2 does not have left recursion.
E->TE'|
E'->+TE'|#
```

```
Enter the Parent Non-Terminal --> T
Enter the Number of Productions --> 2
Enter Production 1 -> T*F
Enter Production 2 -> F
Production Rule -> T->T*F|F
Production 1 has left recursion.
Production 2 does not have left recursion.
T->FT'|
T'->*FT'|#
```

```
Enter the Parent Non-Terminal --> F
Enter the Number of Productions --> 2
Enter Production 1 -> (E)
Enter Production 2 -> id
Production Rule -> F->(E)|id
Production 1 does not have left recursion.
Production 2 does not have left recursion.
F->(E)F'|idF'|
F'->#
```

EXPERIMENT-5

AIM: Write a program to perform Left Factoring on a Grammar

THEORY:

What is Left Factoring

Left factoring is a grammar transformation that produces a grammar more suitable for predictive or top-down praising. If more than one grammar production rules has a standard prefix string, then the top-down parser cannot choose which of the productions it should take to parse the string in hand.

The process by which the grammar with a common prefix is transformed to make it worthwhile for top-down parsers is known as left factoring.

How to do Left Factoring

Left Factoring transforms the grammar to make it useful for top-down parsers. In this technique, we make one production for each common prefix, and the rest of the derivation is added by new productions.

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$$

Rewrite the given expression without changing the meaning,

$$A \rightarrow \alpha X$$

$$X \rightarrow \beta_1 \mid \beta_2$$

You can Check left Recursion

SOURCE CODE:

```
#include<stdio.h>

#include<string.h>

int main()

{

   char gram[20],part1[20],part2[20],modifiedGram[20],newGram[20],tempGram[20];

   int i,j=0,k=0,l=0,pos;

   printf("Enter Production : A->");

   gets(gram);

   for(i=0;gram[i]!='|';i++,j++)
```

```
        part1[j]=gram[i];

    part1[j]='\0';

    for(j=++i,i=0;gram[j]!='\0';j++,i++)

        part2[i]=gram[j];

    part2[i]='\0';

    for(i=0;i<strlen(part1)||i<strlen(part2);i++){

        if(part1[i]==part2[i]){

            modifiedGram[k]=part1[i];

            k++;

            pos=i+1;

        }
    }

    for(i=pos,j=0;part1[i]!='\0';i++,j++){

        newGram[j]=part1[i];

    }
    newGram[j++]='|';

    for(i=pos;part2[i]!='\0';i++,j++){

        newGram[j]=part2[i];

    }
    modifiedGram[k]='X';

    modifiedGram[++k]='\0';

    newGram[j]='\0';

    printf("\nGrammar Without Left Factoring : : \n");
    printf(" A->%s",modifiedGram);

    printf("\n X->%s\n",newGram);

}
```

OUTPUT:

```
Enter Production : A->bE+acF|bE+f

Grammar Without Left Factoring : :
 A->bE+X
 X->acF|f

Process returned 0 (0x0)   execution time : 1.473 s
Press any key to continue.
```

EXPERIMENT-6

AIM: Write a program to show all the operations of a stack.

THEORY:

What is a Stack?

The Last-In-First-Out (LIFO) concept is used by Stacks, a type of linear data structure. The Queue has two endpoints, but the Stack only has one (front and rear). It only has one pointer, top pointer, which points to the stack's topmost member. When an element is added to the stack, it is always added to the top, and it can only be removed from the stack. To put it another way, a stack is a container that allows insertion and deletion from the end known as the stack's top.

LIFO (Last in First out) Method

According to this method, the piece that was added last will appear first. As an actual illustration, consider a stack of dishes stacked on top of one another. We may claim that the plate we put last comes out first because the plate we put last is on top and we take the plate at the top. Due to the possibility of understating inventory value, LIFO is not a reliable indicator of ending inventory value. Due to increasing COGS, LIFO leads to reduced net income (and taxes). However, under LIFO during inflation, there are fewer inventory write-downs. Results from average cost are in the middle of FIFO and LIFO.

Operations on Stack

- push() to insert an element into the stack
- pop() to remove an element from the stack
- top() Returns the top element of the stack.
- isEmpty() returns true if stack is empty else false.
- size() returns the size of stack.

SOURCE CODE:

```
#include <stdio.h>

#include <stdlib.h>

#define SIZE 4

int top = -1, inp_array[SIZE];
void push();
```

```c
void pop();
void show();

int main()
{
   int choice;

   while (1)
   {
      printf("\nPerform operations on the stack:");
      printf("\n1.Push the element\n2.Pop the element\n3.Show\n4.End");
      printf("\n\nEnter the choice: ");
      scanf("%d", &choice);

      switch (choice)
      {
      case 1:
         push();
         break;
      case 2:
         pop();
         break;
      case 3:
         show();
         break;
      case 4:
         exit(0);

      default:
         printf("\nInvalid choice!!");
      }
   }
}

void push()
{
   int x;
```

```
    if (top == SIZE - 1)
    {
        printf("\nOverflow!!");
    }
    else
    {
        printf("\nEnter the element to be added onto the stack: ");
        scanf("%d", &x);
        top = top + 1;
        inp_array[top] = x;
    }
}

void pop()
{
    if (top == -1)
    {
        printf("\nUnderflow!!");
    }
    else
    {
        printf("\nPopped element: %d", inp_array[top]);
        top = top - 1;
    }
}

void show()
{
    if (top == -1)
    {
        printf("\nUnderflow!!");
    }
    else
    {
        printf("\nElements present in the stack: \n");
        for (int i = top; i >= 0; --i)
            printf("%d\n", inp_array[i]);
    }
```

}

OUTPUT:

```
Enter the size of STACK[MAX=100]:10

         STACK OPERATIONS USING ARRAY
        --------------------------------
         1.PUSH
         2.POP
         3.DISPLAY
         4.EXIT
 Enter the Choice:1
 Enter a value to be pushed:12

 Enter the Choice:1
 Enter a value to be pushed:24

 Enter the Choice:1
 Enter a value to be pushed:98

 Enter the Choice:3

 The elements in STACK

98
24
12
 Press Next Choice
 Enter the Choice:2

         The popped elements is 98
 Enter the Choice:3

 The elements in STACK

24
12
 Press Next Choice
 Enter the Choice:4

         EXIT POINT
```

EXPERIMENT-7

AIM: Write a program to find out the leading of the non-terminals in a grammar.

THEORY:

LEADING
If production is of form $A \rightarrow a\alpha$ or $A \rightarrow Ba\ \alpha$ where B is Non-terminal, and α can be any string, then the first terminal symbol on R.H.S is
Leading(A) = {a}
If production is of form $A \rightarrow B\alpha$, if a is in LEADING (B), then a will also be in LEADING (A).

TRAILING
If production is of form $A \rightarrow \alpha a$ or $A \rightarrow \alpha aB$ where B is Non-terminal, and α can be any string then,
TRAILING (A) = {a}
If production is of form $A \rightarrow \alpha B$. If a is in TRAILING (B), then a will be in TRAILING (A).
LEADING
If production is of form $A \rightarrow a\alpha$ or $A \rightarrow Ba\ \alpha$ where B is Non-terminal, and α can be any string, then the first terminal symbol on R.H.S is
Leading(A) = {a}
If production is of form $A \rightarrow B\alpha$, if a is in LEADING (B), then a will also be in LEADING (A).

TRAILING
If production is of form $A \rightarrow \alpha a$ or $A \rightarrow \alpha aB$ where B is Non-terminal, and α can be any string then,
TRAILING (A) = {a}
If production is of form $A \rightarrow \alpha B$. If a is in TRAILING (B), then a will be in TRAILING (A).
LEADING
If production is of form $A \rightarrow a\alpha$ or $A \rightarrow Ba\ \alpha$ where B is Non-terminal, and α can be any string, then the first terminal symbol on R.H.S is
Leading(A) = {a}
If production is of form $A \rightarrow B\alpha$, if a is in LEADING (B), then a will also be in LEADING (A).

TRAILING
If production is of form $A \rightarrow \alpha a$ or $A \rightarrow \alpha aB$ where B is Non-terminal, and α can be any string then,
TRAILING (A) = {a}
If production is of form $A \rightarrow \alpha B$. If a is in TRAILING (B), then a will be in TRAILING (A).

FIRST function is used to find out the terminal symbols that are possible from both terminal and non-terminal symbols. The application of this function is widely seen in designing the Predictive parser tables that are used in designing compilers for various languages. Even the first compiler C, has an in built predictive parser which is operated through the calculation of the first and follow functions.

FIRST function is applied to both terminal symbols and non-terminal symbols such that its definition has certain rules to be employed. They are as follows:

1)FIRST of any terminal symbol ‘a' is the terminal ‘a' itself.
2)FIRST of non terminal ‘A' = FIRST of ‘@', where A -> @ such that @ is a string containing terminals and non-terminals.

FIRST of any entinty is given by FIRST(a) for terminal ‘a' and FIRST(A) for non-terminal ‘A'. The following program code simulates and computes the FIRST of the Grammar defined by the user at the compile time.

LEADING
If production is of form $A \rightarrow a\alpha$ or $A \rightarrow Ba\ \alpha$ where B is Non-terminal, and α can be any string, then the first terminal symbol on R.H.S is
Leading(A) = {a}
If production is of form $A \rightarrow B\alpha$, if a is in LEADING (B), then a will also be in LEADING (A).

TRAILING
If production is of form $A\rightarrow \alpha a$ or $A \rightarrow \alpha aB$ where B is Non-terminal, and α can be any string then,
TRAILING (A) = {a}
If production is of form $A \rightarrow \alpha B$. If a is in TRAILING (B), then a will be in TRAILING (A)

SOURCE CODE:

```
#include<stdio.h>
#include<conio.h>
char array[10][20],temp[10];
int c,n;
void fun(int,int[]);
int fun2(int i,int j,int p[],int );

void main()
{
  int p[2],i,j;
  printf("Enter the no. of productions :");
  scanf("%d",&n);
  printf("Enter the productions :\n");
  for(i=0;i<n;i++)
```

```
  scanf("%s",array[i]);
  for(i=0;i<n;i++)
  {
   c=-1,p[0]=-1,p[1]=-1;
   fun(i,p);
   printf("First(%c) : [ ",array[i][0]);
   for(j=0;j<=c;j++)
   printf("%c,",temp[j]);
   printf("\b ].\n");
   getch();
  }
}

int fun2(int i,int j,int p[],int key)
{
 int k;
 if(!key)
 {
   for(k=0;k<n;k++)
   if(array[i][j]==array[k][0])
   break;
   p[0]=i;p[1]=j+1;
   fun(k,p);
   return 0;
 }
 else
 {
   for(k=0;k<=c;k++)
    {
   if(array[i][j]==temp[k])
   break;
    }
   if(k>c)return 1;
   else return 0;
 }
}
 void fun(int i,int p[])
 {
 int j,k,key;
 for(j=2;array[i][j] != NULL; j++)
 {
 if(array[i][j-1]=='/')
 {
 if(array[i][j]>= 'A' && array[i][j]<='Z')
 {
 key=0;
 fun2(i,j,p,key);
 }
 else
```

```
{
key = 1;
if(fun2(i,j,p,key))
temp[++c] = array[i][j];
if(array[i][j]== '@'&& p[0]!=-1) //taking '@' as null symbol
{
if(array[p[0]][p[1]]>='A' && array[p[0]][p[1]] <='Z')
{
key=0;
fun2(p[0],p[1],p,key);
}
else
if(array[p[0]][p[1]] != '/'&& array[p[0]][p[1]]!=NULL)
{
if(fun2(p[0],p[1],p,key))
temp[++c]=array[p[0]][p[1]];
}
}
```

OUTPUT:

```
Enter the no. of productions :6
Enter the productions :
S/aBDh
B/cC
C/bC/@
E/g/@
D/E/F
F/f/@
First(S) : [ a ].
First(B) : [ c ].
First(C) : [ b,@ ].
First(E) : [ g,@ ].
First(D) : [ g,@,f ].
First(F) : [ f,@ ].

Process returned 6 (0x6)   execution time : 110.862 s
Press any key to continue.
```

EXPERIMENT-8

AIM: Write a program to Implement Shift Reduce parsing for a String.

THEORY:

Shift-reduce parsing is based on a bottom-up parsing technique, where the parser starts with the input tokens and aims to construct a parse tree by applying a set of grammar rules. The two main actions involved in shift-reduce parsing are “shift” and “reduce.”

1. **Shift:** During the shift operation, the parser reads the next input token and pushes it onto a stack. The parser maintains a buffer to hold the remaining input tokens. The shift operation moves the parser’s current position to the right in the input stream.

2. **Reduce:** The reduce operation applies a grammar rule to the tokens on top of the stack. If the tokens on the stack match the right-hand side of a grammar rule, they are replaced with the corresponding non-terminal symbol from the left-hand side of that rule. The reduce operation moves the parser’s current position to the left in the input stream.

3. **Accept:** If the stack contains the start symbol only and the input buffer is empty at the same time then that action is called accept.

4. **Error:** A situation in which the parser cannot either shift or reduce the symbols, it cannot even perform accept action then it is called an error action.

Shift-reduce parsing continues until it reaches the end of the input stream and constructs a valid parse tree. It employs a parsing table, often generated using techniques like the LR(1) or LALR(1) parsing algorithm, to determine the shift or reduce action to take at each step.

SOURCE CODE:

```
struct grammer{
   char p[20];
   char prod[20];
}g[10];

void main()
{
   int i,stpos,j,k,l,m,o,p,f,r;
   int np,tspos,cr;

   cout<<"\nEnter Number of productions:";
   cin>>np;
```

```
char sc,ts[10];

cout<<"\nEnter productions:\n";
for(i=0;i<np;i++)
{
   cin>>ts;
   strncpy(g[i].p,ts,1);
   strcpy(g[i].prod,&ts[3]);
}

char ip[10];

cout<<"\nEnter Input:";
cin>>ip;

int lip=strlen(ip);

char stack[10];

stpos=0;
i=0;

//moving input
sc=ip[i];
stack[stpos]=sc;
i++;stpos++;

cout<<"\n\nStack\tInput\tAction";
do
{
   r=1;
   while(r!=0)
   {
      cout<<"\n";
      for(p=0;p<stpos;p++)
      {
         cout<<stack[p];
      }
      cout<<"\t";
      for(p=i;p<lip;p++)
      {
         cout<<ip[p];
      }

      if(r==2)
      {
         cout<<"\tReduced";
      }
      else
```

```
    {
      cout<<"\tShifted";
    }
    r=0;

    //try reducing
    getch();
    for(k=0;k<stpos;k++)
    {
      f=0;

      for(l=0;l<10;l++)
      {
        ts[l]='\0';
      }

      tspos=0;
      for(l=k;l<stpos;l++) //removing first caharcter
      {
        ts[tspos]=stack[l];
        tspos++;
      }

      //now compare each possibility with production
      for(m=0;m<np;m++)
      {
        cr = strcmp(ts,g[m].prod);

        //if cr is zero then match is found
        if(cr==0)
        {
          for(l=k;l<10;l++) //removing matched part from stack
          {
            stack[l]='\0';
            stpos--;
          }

          stpos=k;

          //concatinate the string
          strcat(stack,g[m].p);
          stpos++;
          r=2;
        }
      }
    }
  }

  //moving input
```

```
        sc=ip[i];
        stack[stpos]=sc;
        i++;stpos++;

    }while(strlen(stack)!=1 && stpos!=lip);

    if(strlen(stack)==1)
    {
        cout<<"\n String Accepted";
    }

    getch();
}
```

OUTPUT:

```
Enter the number of production rules: 4

Enter the production rules (in the form 'left->right'):
E->E+E
E->E*E
E->(E)
E->i

Enter the input string: i*i+i
i       *i+i    Shift i
E       *i+i    Reduce E->i
E*      i+i     Shift *
E*i     +i      Shift i
E*E     +i      Reduce E->i
E       +i      Reduce E->E*E
E+      i       Shift +
E+i             Shift i
E+E             Reduce E->i
E               Reduce E->E+E

Accepted
```

EXPERIMENT-9

AIM: Write a program to find out the FIRST of the Non-terminals in a grammar.

THEORY:

LEADING
If production is of form $A \rightarrow a\alpha$ or $A \rightarrow Ba\ \alpha$ where B is Non-terminal, and α can be any string, then the first terminal symbol on R.H.S is
Leading(A) = {a}
If production is of form $A \rightarrow B\alpha$, if a is in LEADING (B), then a will also be in LEADING (A).

TRAILING
If production is of form $A \rightarrow \alpha a$ or $A \rightarrow \alpha aB$ where B is Non-terminal, and α can be any string then,
TRAILING (A) = {a}
If production is of form $A \rightarrow \alpha B$. If a is in TRAILING (B), then a will be in TRAILING (A).

LEADING
If production is of form $A \rightarrow a\alpha$ or $A \rightarrow Ba\ \alpha$ where B is Non-terminal, and α can be any string, then the first terminal symbol on R.H.S is
Leading(A) = {a}
If production is of form $A \rightarrow B\alpha$, if a is in LEADING (B), then a will also be in LEADING (A).

TRAILING
If production is of form $A \rightarrow \alpha a$ or $A \rightarrow \alpha aB$ where B is Non-terminal, and α can be any string then,
TRAILING (A) = {a}
If production is of form $A \rightarrow \alpha B$. If a is in TRAILING (B), then a will be in TRAILING (A).

LEADING
If production is of form $A \rightarrow a\alpha$ or $A \rightarrow Ba\ \alpha$ where B is Non-terminal, and α can be any string, then the first terminal symbol on R.H.S is
Leading(A) = {a}
If production is of form $A \rightarrow B\alpha$, if a is in LEADING (B), then a will also be in LEADING (A).

TRAILING
If production is of form $A \rightarrow \alpha a$ or $A \rightarrow \alpha aB$ where B is Non-terminal, and α can be any string then,
TRAILING (A) = {a}
If production is of form $A \rightarrow \alpha B$. If a is in TRAILING (B), then a will be in TRAILING (A).

FIRST function is used to find out the terminal symbols that are possible from both terminal and non-terminal symbols. The application of this function is widely seen in designing the Predictive parser tables that are used in designing compilers for various languages. Even the first compiler C, has an in built predictive parser which is operated through the calculation of the first and follow functions.

FIRST function is applied to both terminal symbols and non-terminal symbols such that its definition has certain rules to be employed. They are as follows:

1)FIRST of any terminal symbol ‘a' is the terminal ‘a' itself.
2)FIRST of non terminal ‘A' = FIRST of ‘@', where A -> @ such that @ is a string containing terminals and non-terminals.

FIRST of any entinty is given by FIRST(a) for terminal ‘a' and FIRST(A) for non-terminal ‘A'. The following program code simulates and computes the FIRST of the Grammar defined by the user at the compile time.

LEADING
If production is of form A → aα or A → Ba α where B is Non-terminal, and α can be any string, then the first terminal symbol on R.H.S is
Leading(A) = {a}
If production is of form A → Bα, if a is in LEADING (B), then a will also be in LEADING (A).

TRAILING
If production is of form A→ αa or A → αaB where B is Non-terminal, and α can be any string then,
TRAILING (A) = {a}
If production is of form A → αB. If a is in TRAILING (B), then a will be in TRAILING (A)

SOURCE CODE:

```
#include<stdio.h>
#include<conio.h>
char array[10][20],temp[10];
int c,n;
void fun(int,int[]);
int fun2(int i,int j,int p[],int );

void main()
{
   int p[2],i,j;
   printf("Enter the no. of productions :");
   scanf("%d",&n);
   printf("Enter the productions :\n");
   for(i=0;i<n;i++)
```

```
  scanf("%s",array[i]);
  for(i=0;i<n;i++)
  {
   c=-1,p[0]=-1,p[1]=-1;
   fun(i,p);
   printf("First(%c) : [ ",array[i][0]);
   for(j=0;j<=c;j++)
   printf("%c,",temp[j]);
   printf("\b ].\n");
   getch();
  }
}

int fun2(int i,int j,int p[],int key)
{
 int k;
 if(!key)
 {
   for(k=0;k<n;k++)
   if(array[i][j]==array[k][0])
   break;
   p[0]=i;p[1]=j+1;
   fun(k,p);
   return 0;
 }
 else
 {
   for(k=0;k<=c;k++)
   {
   if(array[i][j]==temp[k])
   break;
   }
   if(k>c)return 1;
   else return 0;
 }
}
 void fun(int i,int p[])
 {
 int j,k,key;
 for(j=2;array[i][j] != NULL; j++)
 {
 if(array[i][j-1]=='/')
 {
 if(array[i][j]>= 'A' && array[i][j]<='Z')
 {
 key=0;
 fun2(i,j,p,key);
 }
 else
```

```
{
key = 1;
if(fun2(i,j,p,key))
temp[++c] = array[i][j];
if(array[i][j]== '@'&& p[0]!=-1) //taking '@' as null symbol
{
if(array[p[0]][p[1]]>='A' && array[p[0]][p[1]] <='Z')
{
key=0;
fun2(p[0],p[1],p,key);
}
else
if(array[p[0]][p[1]] != '/'&& array[p[0]][p[1]]!=NULL)
{
if(fun2(p[0],p[1],p,key))
temp[++c]=array[p[0]][p[1]];
}
}
```

OUTPUT:

```
Enter the no. of productions :6
Enter the productions :
S/aBDh
B/cC
C/bC/@
E/g/@
D/E/F
F/f/@
First(S) : [ a ].
First(B) : [ c ].
First(C) : [ b,@ ].
First(E) : [ g,@ ].
First(D) : [ g,@,f ].
First(F) : [ f,@ ].

Process returned 6 (0x6)   execution time : 110.862 s
Press any key to continue.
```

EXPERIMENT-10

AIM: Write a program to check whether a grammar is operator precedent.

THEORY:

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∈.
- No two non-terminals are adjacent.

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

Precedence table:

	+	*	(	)	id	$
+	⋗	⋖	⋖	⋗	⋖	⋗
*	⋗	⋗	⋖	⋗	⋖	⋗
(	⋖	⋖	⋖	≐	⋖	X
)	⋗	⋗	X	⋗	X	⋗
id	⋗	⋗	X	⋗	X	⋗
$	⋖	⋖	⋖	X	⋖	X

Parsing Action

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

SOURCE CODE:

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

```
// Function to check whether the grammar is operator-precedent
bool isOperatorPrecedent(char grammar[][3], int numRules) {
    for (int i = 0; i < numRules - 1; i++) {
        // Check if the precedence of the current rule is greater than or equal to the next rule
        if (grammar[i][1] >= grammar[i + 1][1]) {
            return false;
        }
        // Check if the associativity of the current rule is the same as the next rule
        if (grammar[i][2] != grammar[i + 1][2]) {
            return false;
        }
    }
    return true;
}

int main() {
    // Maximum number of rules for simplicity
    int maxRules = 10;

    // Example grammar: {operator, precedence, associativity}
    char grammar[maxRules][3];

    printf("Enter the grammar rules in the format (operator precedence associativity), e.g., + 1 L:
\n");

    int numRules;
    printf("Enter the number of rules: ");
    scanf("%d", &numRules);

    if (numRules > maxRules) {
        printf("Exceeded maximum number of rules. Exiting.\n");
        return 1;
    }

    for (int i = 0; i < numRules; i++) {
        printf("Enter rule %d: ", i + 1);
        scanf(" %c %d %c", &grammar[i][0], &grammar[i][1], &grammar[i][2]);
    }

    if (isOperatorPrecedent(grammar, numRules)) {
        printf("The grammar is operator-precedent.\n");
    } else {
        printf("The grammar is not operator-precedent.\n");
    }
```

```
    return 0;
}
```

OUTPUT:

```
Enter the number of rules: 4
Enter rule 1: + 1 L
Enter rule 2: * 2 L
Enter rule 3: - 1 L
Enter rule 4: / 2 L
```

```
The grammar is operator-precedent.
```