



**Session: 2021-22(Odd)**

<b>Session:</b>	<b>2021-22 (Odd)</b>
<b>Subject:</b>	<b>Language Processor</b>
<b>Year:</b>	<b>4<sup>th</sup></b>
<b>Semester:</b>	<b>7<sup>th</sup></b>
<b>Name of Student:</b>	<b>Srividya Avadhani (CS18054)</b>
<b>Batch:</b>	<b>B3</b>

## **Practical No. 1**

**Aim:** Implementation of LEX (Compiler Writing Tool) and demonstration of some sample program.

**INLAB**

**AIM:** Implementation of LEX (Compiler Writing Tool) and demonstration of some sample program.

**CODE:**

**Aim of Program 1:** LEX program to check whether number is positive or negative?

**Sample code:**

```
%{  
}%  
%%  
\+?[0-9]+ {printf ("no. is Positive")
```

```
-[0-9]+ { printf ("no. is Negative");
```

```
%%
```

```
void main()
```

```
{
```

```
    printf("Enter any Decimal No.");
```

```
    yylex();
```

```
}
```

```
yywrap()
```

```
{
```

```
}
```

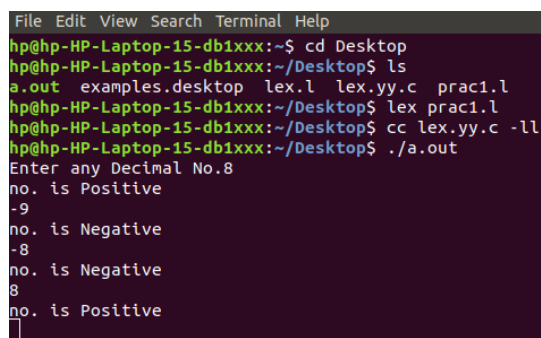
**OUTPUT:**

Enter any Decimal No. 5

no. is Positive

Enter any Decimal No. -632

no. is Negative



```
File Edit View Search Terminal Help  
hp@hp-HP-Laptop-15-db1xxx:~$ cd Desktop  
hp@hp-HP-Laptop-15-db1xxx:~/Desktop$ ls  
a.out examples.desktop lex.l lex.yy.c prac1.l  
hp@hp-HP-Laptop-15-db1xxx:~/Desktop$ lex prac1.l  
hp@hp-HP-Laptop-15-db1xxx:~/Desktop$ cc lex.yy.c -ll  
hp@hp-HP-Laptop-15-db1xxx:~/Desktop$ ./a.out  
Enter any Decimal No.8  
no. is Positive  
-9  
no. is Negative  
-8  
no. is Negative  
8  
no. is Positive  
□
```

**Aim of Program 2:** LEX program to count the space, numbers, small and capital letters, new line & words in given input.

**Sample Code:**

```
%{
    int nw=0,sl=0,cl=0,sc=0,d=0,wd=0;
}%

%%
    \n {nw++;}
    [a-z] {sl++;}
    [A-Z] {cl++;}
    [0-9] {d++;}
    [ ] {sp++;}
    [\t\n]+ {w++;}
%%

main()
{
    printf("Enter String");
    yylex();
    printf("No of new line &nw",nw);

    printf("No of small letter &sl",sl);
    printf("No of capital letter &cl",cl);
    printf("No of spaces &sc",sc);
    printf("No of digits &d",d);
    printf("No of words &wd",wd);

}
yywrap()
{
}
```

**Output:**

```
Enter String: hello I am student 123
No of new line 1
No of small letter 14
No of capital letter 1
No of spaces 4
No of digits 3
No of words 4
```

CONCLUSION: Hence, we done implementation of LEX (Compiler Writing Tool) and demonstration of some programs

## **Practical No. 2**

**Aim:** Write a LEX Program to check input is number, alphanumeric word and Alphabetic word.

**INLAB**

**AIM:** Write a LEX Program to check input is number, alphanumeric word and Alphabetic word.

**CODE:**

```
% {
#include<stdio.h>
#include<string.h>
int al=0,aln=0,n=0,ln=0;
% }
%%
([a-z]|[A-Z])*    {al++;}
(" ")*.(\\n)    {}
[0-9]*    {n++;}
("\\n")*(" ")*("\\n")    {ln++;}
([a-z]|[A-Z]|[0-9])*    {aln++;}
%%
main()
{
yyin=fopen("inp.txt","r");
yylex();
printf("Alphanumeric:%d\\n alphabets:%d\\n numbers:%d\\n line:%d\\n",aln,al,n,ln);
return 0;
}
int yywrap()
{
}
```

**OUTPUT: EXPECTED**

```
File Edit View Search Terminal Help
hp@hp-HP-Laptop-15-db1xxx:~$ cd Desktop
hp@hp-HP-Laptop-15-db1xxx:~/Desktop$ cd Lppractical
hp@hp-HP-Laptop-15-db1xxx:~/Desktop/Lppractical$ lex prac2.l
hp@hp-HP-Laptop-15-db1xxx:~/Desktop/Lppractical$ cc lex.yy.c -ll
hp@hp-HP-Laptop-15-db1xxx:~/Desktop/Lppractical$ ./a.out
1st 2nd Third 4ourth

1st      is a Alphanumeric word
2nd      is a Alphanumeric word
Third    is a Alphabetic Word
4ourth   is a Alphanumeric word

```

CONCLUSION: Hence, we executed LEX Program to check input is number, alphanumeric word and Alphabetic word.

**Practical No. 3**

**Aim:** Implementation of YACC (Compiler Writing Tool) and demonstration of some sample program



**INLAB**

**AIM:** Implementation of YACC (Compiler Writing Tool) and demonstration of some sample programs.

**CODE:**

**Sample programs for demonstration:**

**Aim of Sample Program 1:** YACC program to recognize string aab, ab {a<sup>n</sup>b}

**Sample Code:**

**Lex Code**

```
% {  
    #include "y.tab.h"  
  
% }  
  
%%  
  
[a] {return A;}  
[b] {return B;}  
  
%%
```

**YACC CODE**

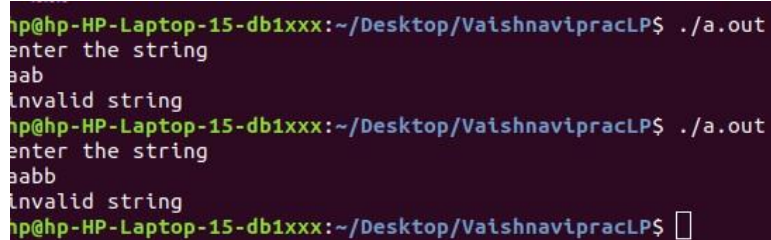
```
% {  
#include <stdio.h>  
  
% }  
  
%token A B  
  
%%  
  
S: T B  
|  
;  
  
T: T A  
|  
;  
  
%%  
  
main()  
{  
    printf("Enter string");
```

```
if(yyparse()==0)
printf("Its valid string");
}
yyerror( )
{
printf("It's not valid string"); }
```

### **OUTPUT:**

Enter string aab  
its valid string

Enter string aabb  
It's not valid string



```
hp@hp-HP-Laptop-15-db1xxx:~/Desktop/VaishnavipracLP$ ./a.out
enter the string
aab
invalid string
hp@hp-HP-Laptop-15-db1xxx:~/Desktop/VaishnavipracLP$ ./a.out
enter the string
aabb
invalid string
hp@hp-HP-Laptop-15-db1xxx:~/Desktop/VaishnavipracLP$
```

CONCLUSION: Hence, we have done Implementation of YACC and demonstration of some sample programs

### **Practical No. 4**

**Aim:** Write a YACC program to recognize string bd, bbdd, bbbddd,  
 $\{b^nd^n\}$ .

**INLAB**

**AIM:** Write a YACC program to recognize string bd, bbdd, bbbddd {b<sup>n</sup>d<sup>n</sup>}.

**CODE:**

**LEX Code:**

```
% {  
#include "y.tab.h"  
% }  
%%  
[aA] {return A;}  
[bB] {return B;}  
\n {return NL;}  
. {return yytext[0];}  
%%
```

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

**YACC Code:**

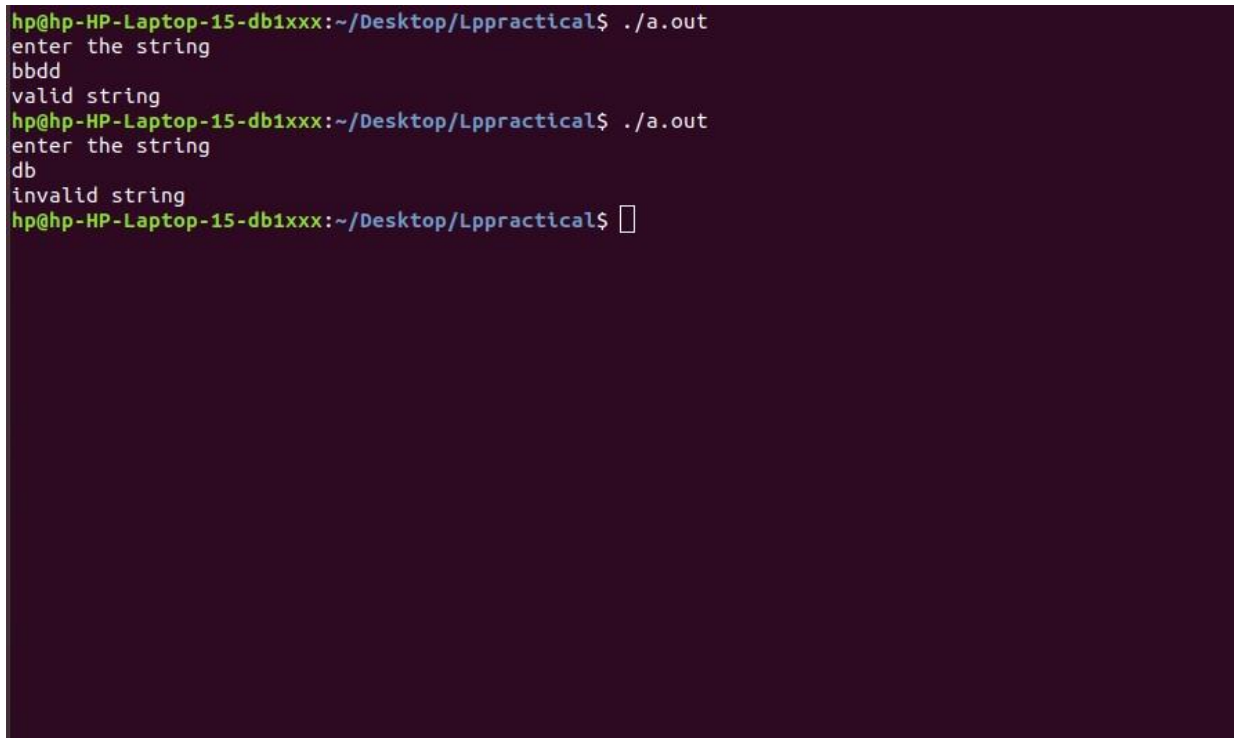
```
% {  
#include<stdio.h>  
#include<stdlib.h>  
% }  
%token A B NL  
%%  
stmt: S NL { printf("valid string\n");  
            exit(0); }  
  
;  
S: A S B |  
;  
%%
```

```
int yyerror(char *msg)  
{  
printf("invalid string\n");
```

```
exit(0);  
}  
main()  
{  
  
printf("enter the string\n");  
yyparse();  
}
```

**OUTPUT: Expected**

1. Enter string bbdd  
Its valid string
2. Enter string db  
It's not valid string



```
hp@hp-HP-Laptop-15-db1xxx:~/Desktop/Lppractical$ ./a.out  
enter the string  
bbdd  
valid string  
hp@hp-HP-Laptop-15-db1xxx:~/Desktop/Lppractical$ ./a.out  
enter the string  
db  
invalid string  
hp@hp-HP-Laptop-15-db1xxx:~/Desktop/Lppractical$
```

**CONCLUSION:** Hence, we executed Write a YACC program to recognize string bd, bbdd, bbbddd, {bndn}.

## **Practical No. 5**

**Aim:** Write a C program to calculate FIRST ( ) of given grammar.

**AIM:** Write a C program to calculate FIRST ( ) of given grammar.

**CODE:**

```
#include<stdio.h>
#include<ctype.h>
void FIRST(char[],char );
void addToResultSet(char[],char);
int numOfProductions;
char productionSet[10][10];
main()
{
    int i;
    char choice;
    char c;
    char result[20];

    printf("How many number of productions ? :");
    scanf(" %d",&numOfProductions);

    for(i=0;i<numOfProductions;i++)//read production string eg: E=E+T
    {
        printf("Enter productions Number %d : ",i+1);
        scanf(" %s",productionSet[i]);
    }
    do
    {
        printf("\n Find the FIRST of :");
        scanf(" %c",&c);
        FIRST(result,c); //Compute FIRST; Get Answer in 'result' array
        printf("\n FIRST(%c)= { ",c);
        for(i=0;result[i]!='\0';i++)
            printf(" %c ",result[i]);    //Display result
        printf("}\n");
        printf("press 'y' to continue : ");
        scanf(" %c",&choice);
    }
    while(choice=='y'||choice=='Y');
}
void FIRST(char* Result,char c)
{
    int i,j,k;
    char subResult[20];
    int foundEpsilon;
    subResult[0]='\0';
    Result[0]='\0';
    if(!isupper(c))
    {
```

```
        addToResultSet(Result,c);
        return ;
    }
    for(i=0;i<numOfProductions;i++)
    {
        if(productionSet[i][0]==c)
        {
            if(productionSet[i][2]=='$') addToResultSet(Result,$');
            else
            {
                j=2;
                while(productionSet[i][j]!='\0')
                {
                    foundEpsilon=0;
                    FIRST(subResult,productionSet[i][j]);
                    for(k=0;subResult[k]!='\0';k++)

                        addToResultSet(Result,subResult[k]);
                    for(k=0;subResult[k]!='\0';k++)
                        if(subResult[k]=='$')
                        {
                            foundEpsilon=1;
                            break;
                        }
                    if(!foundEpsilon)
                        break;
                    j++;
                }
            }
        }
    }
    return ;
}

void addToResultSet(char Result[],char val)
{
    int k;
    for(k=0 ;Result[k]!='\0';k++)
        if(Result[k]==val)
            return;
    Result[k]=val;
    Result[k+1]='\0';
}
```



**OUTPUT: Expected**

How much number of productions 3

Enter production number 1    E=aa

Enter production number 2    A=Ba

Enter production number 3    B=c

Find first of:

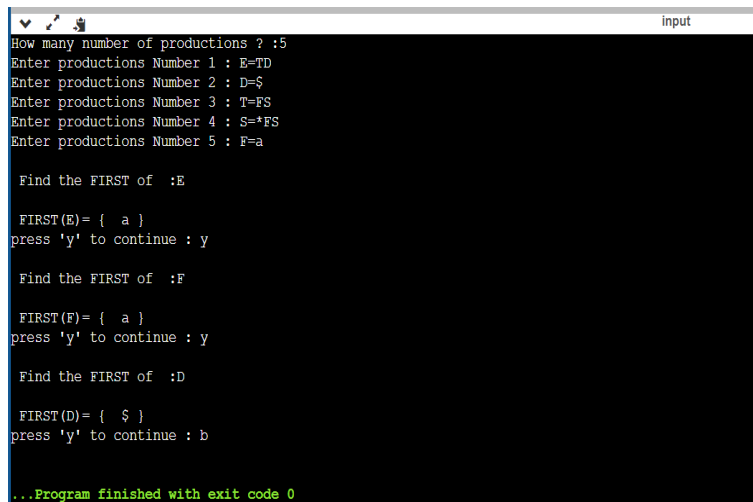
EFirst (E) = a

Press Y/N to continue: Y

Find first of: A

First(A) = c

Press Y/N to continue: N



```
How many number of productions ? :5
Enter productions Number 1 : E=TD
Enter productions Number 2 : D=$
Enter productions Number 3 : T=FS
Enter productions Number 4 : S=*FS
Enter productions Number 5 : F=a

Find the FIRST of :E

FIRST(E)= { a }
press 'y' to continue : y

Find the FIRST of :F

FIRST(F)= { a }
press 'y' to continue : y

Find the FIRST of :D

FIRST(D)= { $ }
press 'y' to continue : b

...Program finished with exit code 0
```

CONCLUSION: Hence, we write a C program to calculate FIRST ( ) of given grammar.

**Practical No. 6**

**Aim:** Write a C program to calculate FOLLOW ( ) of given grammar

**INLAB**

**AIM:** Write a C program to calculate FOLLOW ( ) of given grammar.

**CODE:**

```
#include<stdio.h>
#include<string.h>
int n,m=0,p,i=0,j=0;
char a[10][10],followResult[10];
void follow(char c);
void first(char c);
void addToResult(char);
int main()

{
    int i;
    int choice;
    char c,ch;
    printf("Enter the no.of productions: ");
    scanf("%d", &n);
    printf(" Enter %d productions\nProduction with multiple terms should be give as separate
productions \n", n);
    for(i=0;i<n;i++)
        scanf("%s%c",a[i],&ch);
        // gets(a[i]);
    do
    {
        m=0;
        printf("Find FOLLOW of -->");
        scanf(" %c",&c);
        follow(c);
        printf("FOLLOW(%c) = { ",c);
        for(i=0;i<m;i++)
            printf("%c ",followResult[i]);
        printf(" }\n");
        printf("Do you want to continue(Press 1 to continue... )?");
        scanf("%d%c",&choice,&ch);
    }
    while(choice==1);
}

void follow(char c)
{
    if(a[0][0]==c)addToResult('$');
```

```
for(i=0;i<n;i++)
{
    for(j=2;j<strlen(a[i]);j++)
    {
        if(a[i][j]==c)
        {
            if(a[i][j+1]!='\0')first(a[i][j+1]);
            if(a[i][j+1]=='\0'&& c!=a[i][0])
                follow(a[i][0]);
        }
    }
}
```

```
void first(char c)
{
    int k;
    if(!(isupper(c)))
        //f[m++]=c; addToResult(c);
    for(k=0;k<n;k++)
    {
        if(a[k][0]==c)
        {
            if(a[k][2]=='$') follow(a[i][0]); else
            if(islower(a[k][2]))
                //f[m++]=a[k][2];
                addToResult(a[k][2]);
            else first(a[k][2]);
        }
    }
}
```

```
void addToResult(char c)
{
    int i;
    for( i=0;i<=m;i++)
        if(followResult[i]==c)
            return;
    followResult[m++]=c;
}
}
```

**OUTPUT: Expected**

Enter the number of productions 3  
Enter 3 productions

E=aEb

A=aAaa

B=cc

Find follow of: E

FOLLOW (E) = b

Do you want to continue (Y/N): Y

Find follow of: A

FOLLOW (A) = a

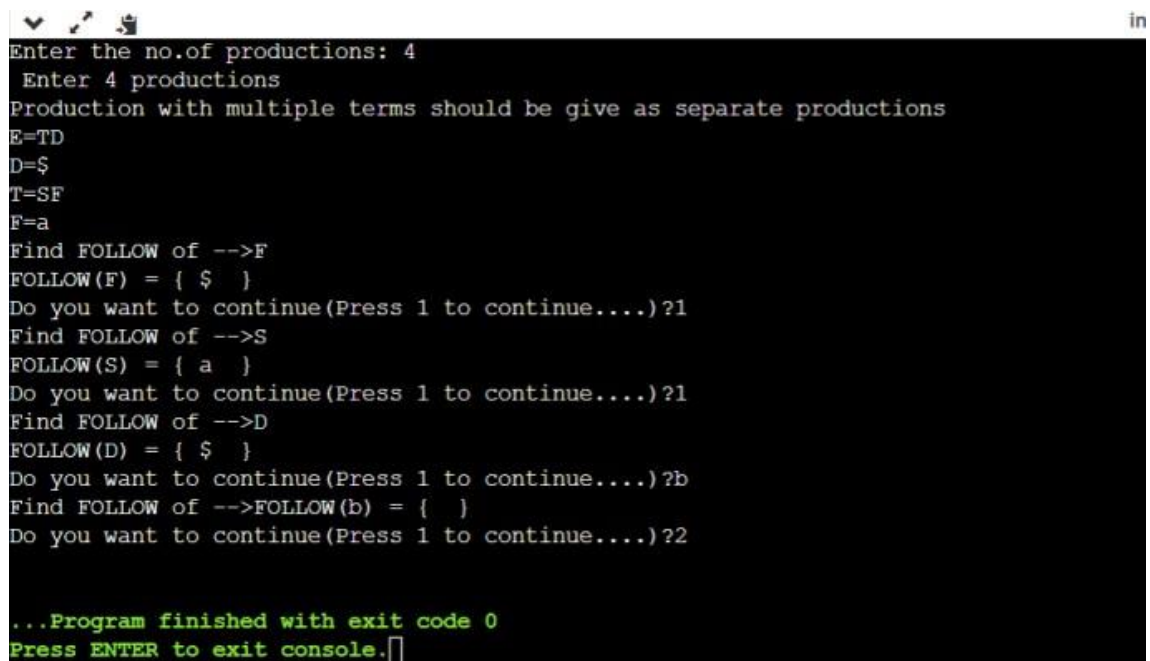
Do you want to continue (Y/N): Y

Find follow of: B

FOLLOW (B) = c

Do you want to continue (Y/N): N

**Conclusion**Hence, we Write a C program for practical

A screenshot of a terminal window showing the execution of a C program. The program prompts the user to enter the number of productions (4) and then the productions themselves: E=TD, D=\$, T=SF, and F=a. It then iteratively calculates the FOLLOW sets for each non-terminal. For F, FOLLOW(F) = { \$ }. For S, FOLLOW(S) = { a }. For D, FOLLOW(D) = { \$ }. For T, FOLLOW(T) = FOLLOW(b) = { }. The program finishes with exit code 0 and prompts the user to press ENTER to exit the console.

```
Enter the no.of productions: 4
Enter 4 productions
Production with multiple terms should be give as separate productions
E=TD
D=$
T=SF
F=a
Find FOLLOW of -->F
FOLLOW(F) = { $ }
Do you want to continue(Press 1 to continue....)?1
Find FOLLOW of -->S
FOLLOW(S) = { a }
Do you want to continue(Press 1 to continue....)?1
Find FOLLOW of -->D
FOLLOW(D) = { $ }
Do you want to continue(Press 1 to continue....)?b
Find FOLLOW of -->FOLLOW(b) = { }
Do you want to continue(Press 1 to continue....)?2

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

**Practical No. 7**

**Aim:** Write a C Program to obtain Predictive Parsing Table i.e. LL (1) for the Context Free Grammar (CFG).

**INLAB**

**AIM:** Write a C Program to obtain Predictive Parsing Table i.e. LL (1) for the Context Free Grammar (CFG).

**CODE:**

```
#include<stdio.h>
#include<string.h>
char prol[7][10]={ "S","A","A","B","B","C","C"};
char pror[7][10]={ "A","Bb","Cd","aB","@","Cc","@" };
char prod[7][10]={ "S->A","A->Bb","A->Cd","B->aB","B->@","C->Cc","C->@" };
char first[7][10]={ "abcd","ab","cd","a@","@","c@","@" };
char follow[7][10]={ "$","$","$","a$","b$","c$","d$"};
char table[5][6][10];
numr(char c)
{
switch(c)
{
case 'S': return 0;
case 'A': return 1;
case 'B': return 2;
case 'C': return 3;
case 'a': return 0;
case 'b': return 1;
case 'c': return 2;
case 'd': return 3;
case '$': return 4;
}
return(2);
}
void main()
{
int i,j,k;
for(i=0;i<5;i++)
for(j=0;j<6;j++)
strcpy(table[i][j]," ");
printf("\nThe following is the predictive parsing table for the following grammar:\n");
for(i=0;i<7;i++)
printf("%s\n",prod[i]);
printf("\nPredictive parsing table is\n");
fflush(stdin);
for(i=0;i<7;i++)
{
k=strlen(first[i]);
for(j=0;j<10;j++)
if(first[i][j]!='@')
strcpy(table[numr(prol[i][0])+1][numr(first[i][j])+1],prod[i]);
}
```

```
}
for(i=0;i<7;i++)
{
if(strlen(pror[i])==1)
{
if(pror[i][0]=='@')
{

k=strlen(follow[i]);
for(j=0;j<k;j++)
strcpy(table[numr(prol[i][0])+1][numr(follow[i][j])+1],prod[i]);
}
}
}
strcpy(table[0][0]," ");
strcpy(table[0][1],"a");
strcpy(table[0][2],"b");
strcpy(table[0][3],"c");
strcpy(table[0][4],"d");
strcpy(table[0][5],"$");
strcpy(table[1][0],"S");
strcpy(table[2][0],"A");
strcpy(table[3][0],"B");
strcpy(table[4][0],"C");
printf("\n.....\n");
for(i=0;i<5;i++)
for(j=0;j<6;j++)
{
printf("%-10s",table[i][j]);
if(j==5)
printf("\n.....\n");
}
}
```

**OUTPUT: Expected**

Enter the CFG:

S->A A-

>BbA->Cd

B->aBB-

>@ C->Cc

C->@

Predictive parsing table is:

NT T	A	b	c	d	\$
S	S->A	S->A	S->A	S->A	
A	A->Bb	A->Bb	A->Cd	A->Cd	
B	B->aB	B->@	B->@		B->@
C	C->@	C->@	C->@		



```
The following is the predictive parsing table for the following grammar:
S->A
A->Bb
A->Cd
B->aB
B->ε
C->Cc
C->ε

Predictive parsing table is
```

	a	b	c	d	\$
S	S->A	S->A	S->A	S->A	
A	A->Bb	A->Bb	A->Cd	A->Cd	
B	B->aB	B->ε	B->ε	B->ε	
C			C->ε	C->ε	C->ε

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

**CONCLUSION:** Hence, we Write a C Program to obtain Predictive Parsing Table i.e. LL (1) for the Context Free Grammar (CFG).

### **Practical No. 8**

**Aim:** Write a C Program to implement Predictive Parser table for a given Grammar and input String

## INLAB

**AIM:** Write a C Program to implement Predictive Parser table for a given Grammar and input String.

### CODE:

```
#include<iostream>
#include<string>
#include<deque>
using namespace std;
int n,n1,n2;
int getPosition(string arr[], string q, int size)
{
    for(int i=0;i<size;i++)
    {
        if(q == arr[i])
            return i;
    }
    return -1;
}
int main()
{
    string prods[10],first[10],follow[10],nonterms[10],terms[10];
    string pp_table[20][20] = { };
    cout<<"Enter the number of productions : ";
    cin>>n;
    cin.ignore();
    cout<<"Enter the productions"<<endl;
    for(int i=0;i<n;i++)
    {
        getline(cin,prods[i]);
        cout<<"Enter first for "<<prods[i].substr(3)<<" : ";
        getline(cin,first[i]);
    }
    cout<<"Enter the number of Terminals : ";
    cin>>n2;
    cin.ignore();
    cout<<"Enter the Terminals"<<endl;
    for(int i=0;i<n2;i++)
    {
        cin>>terms[i];
    }
    terms[n2] = "$";
    n2++;
    cout<<"Enter the number of Non-Terminals : ";
    cin>>n1;
    cin.ignore();
    for(int i=0;i<n1;i++)
```

```

{
    cout<<"Enter Non-Terminal : ";
    getline(cin,nonterms[i]);
    cout<<"Enter follow of "<<nonterms[i]<<" : ";

    getline(cin,follow[i]);
}

cout<<endl;
cout<<"Grammar"<<endl;
for(int i=0;i<n;i++)
{
    cout<<prods[i]<<endl;
}

for(int j=0;j<n;j++)
{
    int row = getPosition(nonterms,prods[j].substr(0,1),n1);
    if(prods[j].at(3)!='#')
    {
        for(int i=0;i<first[j].length();i++)
        {
            int col = getPosition(terms,first[j].substr(i,1),n2);
            pp_table[row][col] = prods[j];
        }
    }
    else
    {
        for(int i=0;i<follow[row].length();i++)
        {
            int col = getPosition(terms,follow[row].substr(i,1),n2);
            pp_table[row][col] = prods[j];
        }
    }
}
//Display Table
for(int j=0;j<n2;j++)
    cout<<"\t"<<terms[j];
cout<<endl;
for(int i=0;i<n1;i++)
{
    cout<<nonterms[i]<<"\t";
    //Display Table
    for(int j=0;j<n2;j++)
    {
        cout<<pp_table[i][j]<<"\t";
    }
    cout<<endl;
}

```

```

}
//Parsing String
char c;
do{
string ip;

deque<string> pp_stack;
pp_stack.push_front("$");
pp_stack.push_front(prods[0].substr(0,1));
cout<<"Enter the string to be parsed : ";
getline(cin,ip);
ip.push_back('$');
cout<<"Stack\tInput\tAction"<<endl;
while(true)
{
    for(int i=0;i<pp_stack.size();i++)
        cout<<pp_stack[i];
    cout<<"\t"<<ip<<"\t";
    int row1 = getPosition(nonterms,pp_stack.front(),n1);
    int row2 = getPosition(terms,pp_stack.front(),n2);
    int column = getPosition(terms,ip.substr(0,1),n2);
    if(row1 != -1 && column != -1)
    {
        string p = pp_table[row1][column];
        if(p.empty())
        {
            cout<<endl<<"String cannot be Parsed."<<endl;
            break;
        }
        pp_stack.pop_front();
        if(p[3] != '#')
        {
            for(int x=p.size()-1;x>2;x--)
            {
                pp_stack.push_front(p.substr(x,1));
            }
        }
        cout<<p;
    }
    else
    {
        if(ip.substr(0,1) == pp_stack.front())
        {
            if(pp_stack.front() == "$")
            {
                cout<<endl<<"String Parsed."<<endl;
                break;
            }
            cout<<"Match "<<ip[0];
            pp_stack.pop_front();
        }
    }
}
}

```

```

        ip = ip.substr(1);
    }
    else
    {
        cout<<endl<<"String cannot be Parsed."<<endl;
        break;
    }
}
cout<<endl;
}
cout<<"Continue?(Y/N) ";
cin>>c;
cin.ignore();
}while(c=='y' || c=='Y');
return 0;
}

```

### OUTPUT: Expected

Enter the Predictive parsing table:

NT T	a	b	c	d	\$
S	S->A	S->A	S->A	S->A	
A	A->Bb	A->Bb	A->Cd	A->Cd	
B	B->aB	B->@	B->@		B->@
C	C->@	C->@	C->@		

Enter the input string to be parsed: a@b

1.  $S \rightarrow A$
2.  $S \rightarrow Bb$
3.  $S \rightarrow aBb$
4.  $S \rightarrow a@b$

```

Enter the number of productions : 5
Enter the productions
S->aXYb
Enter first for aXYb : a
X->c
Enter first for c : c
Y->d
Enter first for d : d
Y->d
Enter first for d : d
Y->d
Enter first for d : d
Y->d
Enter the number of Terminals : 4
Enter the Terminals
a
b
c
d
Enter the number of Non-Terminals : 3
Enter Non-Terminal : S
Enter follow of S : $
Enter Non-Terminal : X
Enter follow of X : bd
Enter Non-Terminal : Y
Enter follow of Y : b

Grammar
S->aXYb
X->c
Y->d
Y->d
Y->d
Y->d

          a      b      c      d      $
S      S->aXYb  X->#    X->c    X->#
X      Y->#
Y      Y->#    Y->d

Enter the string to be parsed : abdc
Stack  Input  Action

```

```

aXYb$  abdc$  Match a
XYb$   bdc$   X->#
Yb$    bdc$   Y->#
b$     bdc$   Match b
$      dc$    Match c
String cannot be Parsed.
Continue?(Y/N) y
Enter the string to be parsed : acbd
Stack  Input  Action
S$     acbd$  S->aXYb
aXYb$  acbd$  Match a
XYb$   cbd$   X->c
cYb$   cbd$   Match c
Yb$    bd$    Y->#
b$     bd$    Match b
$      d$     Match d
String cannot be Parsed.
Continue?(Y/N) y
Enter the string to be parsed : acdb
Stack  Input  Action
S$     acdb$  S->aXYb
aXYb$  acdb$  Match a
XYb$   cdb$   X->c
cYb$   cdb$   Match c
Yb$    db$    Y->d
db$    db$    Match d
b$     b$     Match b
$      $
String Parsed.
Continue?(Y/N) y
Enter the string to be parsed : ab
Stack  Input  Action
S$     ab$    S->aXYb
aXYb$  ab$    Match a
XYb$   b$     X->#
Yb$    b$     Y->#
b$     b$     Match b
$      $
String Parsed.

```

**CONCLUSION:** Hence, we Write a C Program to implement Predictive Parser table for a given Grammar and input String

### **Practical No. 9**

**Aim:** Write a C Program to generate three address codes for Arithmetic expression



## INLAB

**AIM:** Write a C Program to generate three address codes for Arithmetic expression.

### CODE

```
l=strlen(expe);
exp1[0]='\0';
for(i=0;i<l;i++)
{
    if(expe[i]=='+'||expe[i]=='-')
    {
        if(expe[i+2]=='/'||expe[i+2]=='*')
        {
            while (expe[count] != '\0')
            {
                count++;
            }
            j = count - 1;
            for (i = 0; i < count; i++)
            {
                rev[i] = expe[j];
                j--;
            }
            j=l-i-1;
            strncat(exp1,rev,j);
            while (exp1[count] != '\0')
            {
                count++;
            }
            j = count - 1;
            for (i = 0; i < count; i++)
            {
                rev1[i] = exp1[j];
                j--;
            }
            printf("Three address code:\ntemp=%s\ntemp1=temp%c%c\n",exp1,expe[j+1],expe[j]);
            break;
        }
        else
        {
            strncat(exp1,expe,i+2);
            printf("Three address code:\ntemp=%s\ntemp1=temp%c%c\n",exp1,expe[i+2],expe[i+3]);
            break;
        }
    }
    else if(expe[i]=='/'||expe[i]=='*')
    {
        strncat(exp1,expe,i+2);
        printf("Three address code:\ntemp=%s\ntemp1=temp%c%c\n",exp1,expe[i+2],expe[i+3]);
        break;
    }
}
```

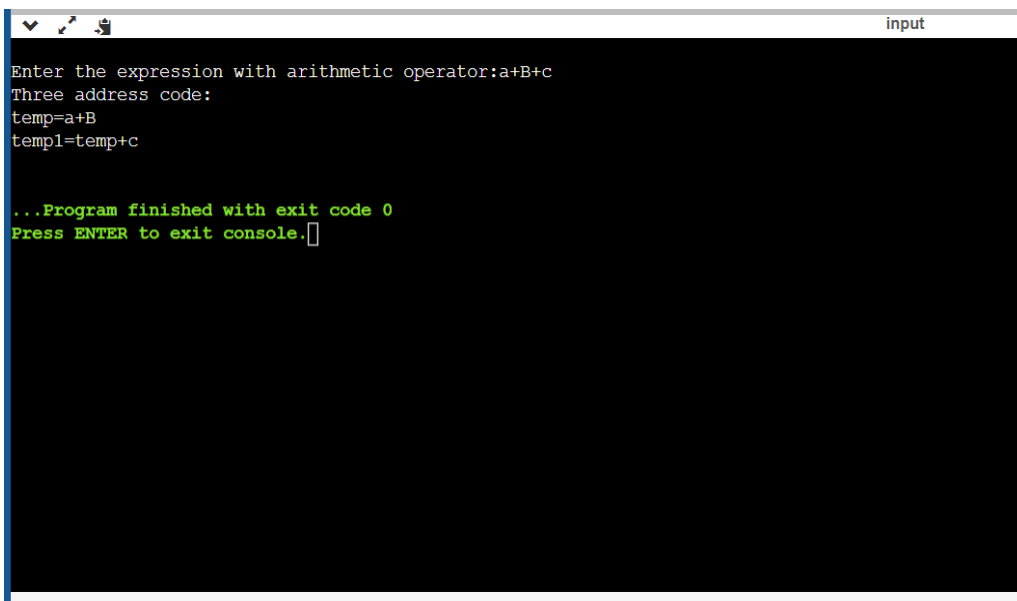
```
}  
}
```

**OUTPUT: Expected**

**Enter the expression:**  $a=a+b*c$

**TAC is:**

1.  $T1 = b * c$
2.  $T2 = a + T1$
3.  $a = T2$



```
input  
Enter the expression with arithmetic operator:a+B+c  
Three address code:  
temp=a+B  
temp1=temp+c  
  
...Program finished with exit code 0  
Press ENTER to exit console.█
```

**CONCLUSION:** Hence, we write a C Program to generate three address codes for Arithmetic expression.

## **Practical No. 10**

**Aim:** Write a C Program to generation of assembly code from the three address code.

## INLAB

**AIM:** Write a C Program to generation of assembly code from the three address code.

### Code:

```
#include<stdio.h>
#include<string.h>
#include<ctype.h>
void main()
{
    char a[20];
    int x;
    int i,j=0,k;
    i=0;
    scanf("%s",a);
    if( strlen(a)==6)
    {
        i=i+3;
        if(islower(a[i]))
            printf("lw $t%d, (%c)\n", j++,a[i]);
        else

        {
            for(i=3;i < strlen(a);i++)
            {
                if(isdigit(a[i]))
                {
                    x= a[i] - '0';
                    k=k*10 +x;
                }
            }
            printf("li $t%d, %d\n", j,k);
        }
        i=i+2;
        if(islower( a[i]))
            printf("lw $t%d, (%c)\n", j++,a[i]);
        else
        {
            for(i=3;i < strlen(a);i++)
            {
                if(isdigit(a[i]))
                {
                    x= a[i] - '0';
                    k=k*10 +x;
                }
            }
            printf("li $t%d, %d\n", j,k);
        }
    }
}
```

```

    }
    i=i-1;
    if(a[i] == '+')
        printf("add $t%d, $t%d, $t%d\n", j,j-1,j-2);
    else if( a[i] == '-')
        printf("sub $t%d, $t%d, $t%d\n", j,j-2,j-1);
    else if( a[i] == '*')
        printf("mul $t%d, $t%d, $t%d\n", j,j-2,j-1);
    else if( a[i] == '/')
        printf("div $t%d, $t%d, $t%d\n", j,j-2,j-1);
    }
    else if(strlen(a)==4)
    {
        i=i+3;
        if( islower(a[i]))
        {
            printf("lw $t%d, %c\n",j,a[i]);
            printf("copy %c, $t%d\n", a[i-3],j);
        }
        else
            printf("li $t%d, %c\n",j,a[i]);
    }
    j=j+1;
}

```

### OUTPUT: Expected

*Enter the Three address Code:*

1.  $a := b + c$
2.  $d := a + e$

*Assembly code:*

```

MOV b, R0
ADD c, R0

MOV R0, a
MOV a, R0
ADD e, R0
MOV R0, d

```



The image shows a terminal window with a dark header bar. The header bar has two tabs: 'input' and 'stdout'. Below the header, the terminal displays the following text:

```
Compiled Successfully. memory: 1548 time: 0 exit code: 0
```

```
lw $t0, (b)
lw $t1, (c)
add $t2, $t1, $t0
```

**CONCLUSION:** Hence, we write a C Program to generation of assembly code from the threeaddress code

### **Practical No. 11**

**Aim:** Write a program to design calculator using LEX and YACC

## INLAB

**AIM:** Write a program to design calculator using LEX and YACC.

### CODE:

#### LEX Code:

```
% {
#include<stdio.h>
#include "y.tab.h"
extern int yylval;
% }
%%
[0-9]+ {
    yylval=atoi(yytext);
    return NUMBER;}

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

#### YACC Code:

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



## **Practical No. 12**

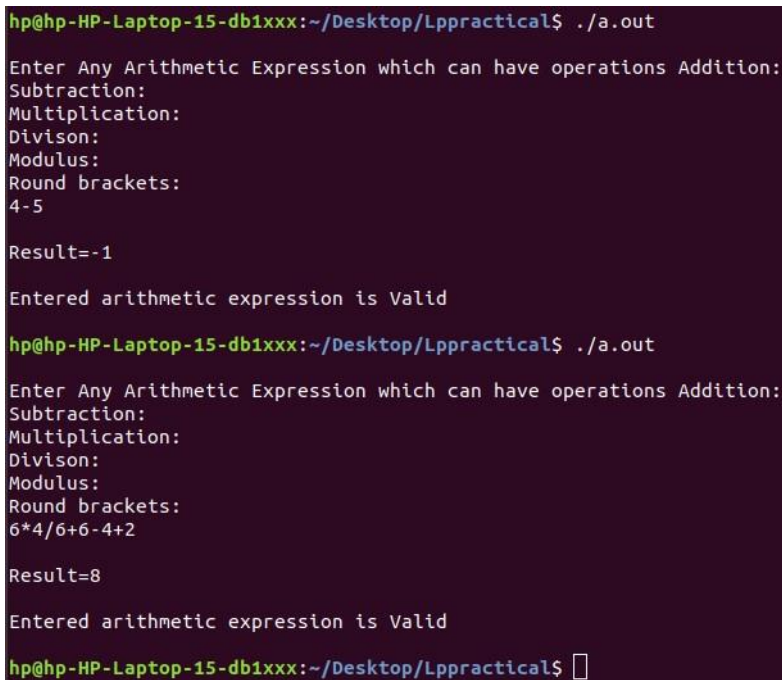
**Aim:** Demonstration of Stanford POS tagger (English Language) (NLP).

## INLAB

**AIM:** Demonstration of Stanford POS tagger (English Language) (NLP).

```
|('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;
}
```

## OUTPUT:



```
hp@hp-HP-Laptop-15-db1xxx:~/Desktop/Lppractical$ ./a.out
Enter Any Arithmetic Expression which can have operations Addition:
Subtraction:
Multiplication:
Divison:
Modulus:
Round brackets:
4-5

Result=-1

Entered arithmetic expression is Valid

hp@hp-HP-Laptop-15-db1xxx:~/Desktop/Lppractical$ ./a.out
Enter Any Arithmetic Expression which can have operations Addition:
Subtraction:
Multiplication:
Divison:
Modulus:
Round brackets:
6*4/6+6-4+2

Result=8

Entered arithmetic expression is Valid

hp@hp-HP-Laptop-15-db1xxx:~/Desktop/Lppractical$
```

**CONCLUSION:** Hence, we designed calculator using LEX and YACC.

## **Practical No. 13**

## INLAB

**AIM:** Demonstration of Stanford POS tagger (English Language) (NLP).

### OBJECTIVE/EXPECTED LEARNING OUTCOME:

- To explain the concept of Natural Language programming
- To relate the POS tagger for English language with NLP.

### THEORY:

In corpus linguistics, part-of-speech tagging (POS tagging or POST), also called grammatical tagging or word-category disambiguation, is the process of marking up a word in a text (corpus) as corresponding to a particular part of speech, based on both its definition and its context—i.e., its relationship with adjacent and related words in a phrase, sentence, or paragraph. A simplified form of this is commonly taught to school-age children, in the identification of words as nouns, verbs, adjectives, adverbs, etc.

Once performed by hand, POS tagging is now done in the context of computational linguistics, using algorithms which associate discrete terms, as well as hidden parts of speech, in accordance with a set of descriptive tags. POS-tagging algorithms fall into two distinctive groups: rule-based and stochastic. E. Brill's tagger, one of the first and most widely used English POS-taggers, employs rule-based algorithms.

#### *Principle*

Part-of-speech tagging is harder than just having a list of words and their parts of speech, because some words can represent more than one part of speech at different times, and because some parts of speech are complex or unspoken. This is not rare—in [natural languages](#) (as opposed to many [artificial languages](#)), a large percentage of word-forms are ambiguous. For example, even "dogs", which is usually thought of as just a plural noun, can also be a verb:

The sailor dogs the hatch.

Correct grammatical tagging will reflect that "dogs" is here used as a verb, not as the more common plural noun. Grammatical context is one way to determine this; semantic analysis can also be used to infer that "sailor" and "hatch" implicate "dogs" as 1) in the nautical context and 2) an action applied to the object "hatch" (in this context, "dogs" is a [nautical](#) term meaning "fastens (a watertight door) securely").

Schools commonly teach that there are 9 [parts of speech](#) in English: [noun](#), [verb](#), [article](#), [adjective](#), [preposition](#), [pronoun](#), [adverb](#), [conjunction](#), and [interjection](#). However, there are clearly many more categories and sub-categories. For nouns, the plural, possessive, and singular forms can be distinguished. In many languages words are also marked for their "[case](#)" (role as subject, object,

etc.), [grammatical gender](#), and so on; while verbs are marked for [tense](#), [aspect](#), and other things. Linguists distinguish parts of speech to various fine degrees, reflecting a chosen "tagging system".

In part-of-speech tagging by computer, it is typical to distinguish from 50 to 150 separate parts of speech for English. For example, NN for singular common nouns, NNS for plural common nouns, NP for singular proper nouns (see the [POS tags](#) used in the Brown Corpus). Work on [stochastic methods](#) for tagging [Koine Greek](#) (DeRose 1990) has used over 1,000 parts of speech, and found that about as many words were [ambiguous](#) there as in English. A morphosyntactic descriptor in the case of morphologically rich languages is commonly expressed using very short mnemonics, such as 'Ncmsan for Category=Noun, Type = common, Gender = masculine, Number = singular, Case = accusative, Animate = no.

## English

[English](#) words have been classified into eight or nine parts of speech (this scheme, or slight expansions of it, is still followed in most [dictionaries](#)):

### Noun (names)

a word or lexical item denoting any abstract (abstract noun: e.g. *home*) or concrete entity (concrete noun: e.g. *house*); a person (*police officer, Michael*), place (*coastline, London*), thing (*necktie, television*), idea (*happiness*), or quality (*bravery*). Nouns can also be classified as [count nouns](#) or [non-count nouns](#); some can belong to either category. The most common part of the speech; they are called naming words.

### Pronoun (replaces)

a substitute for a noun or noun phrase (*them, he*). Pronouns make sentences shorter and clearer since they replace nouns.

### Adjective (describes, limits)

a modifier of a noun or pronoun (*big, brave*). Adjectives make the meaning of another word (noun) more precise.

### Verb (states action or being)

a word denoting an action (*walk*), occurrence (*happen*), or state of being (*be*). Without a verb a group of words cannot be a clause or sentence.

### Adverb (describes, limits)

a modifier of an adjective, verb, or other adverb (*very, quite*). Adverbs makes writing more precise.

### Preposition (relates)

a word that relates words to each other in a phrase or sentence and aids in syntactic context (*in, of*). Prepositions show the relationship between a noun or a pronoun with another word in the sentence.

### Conjunction (connects)

a syntactic connector; links words, phrases, or clauses (*and, but*). Conjunctions connect words or group of words

### Interjection (expresses feelings and emotions)

an emotional greeting or exclamation (*Huzzah, Alas*). Interjections express strong feelings

and emotions.

Article (describes, limits)

a grammatical marker of definiteness (*the*) or indefiniteness (*a, an*). The article is not always listed among the parts of speech. It is considered by some grammarians to be a type of adjective<sup>[12]</sup> or sometimes the term '**determiner**' (a broader class) is used.

English words are not generally **marked** as belonging to one part of speech or another; this contrasts with many other European languages, which use **inflection** more extensively, meaning that a given word form can often be identified as belonging to a particular part of speech and having certain additional **grammatical properties**. In English, most words are uninflected, while the inflective endings that exist are mostly ambiguous: *-ed* may mark a verbal past tense, a participle or a fully adjectival form; *-s* may mark a plural noun or a present-tense verb form; *-ing* may mark a participle, **gerund**, or pure adjective or noun. Although *-ly* is a frequent adverb marker, some adverbs (e.g. *tomorrow, fast, very*) do not have that ending, while some words with that ending (e.g. *friendly, ugly*) are not adverbs.

Many English words can belong to more than one part of speech. Words like *neigh, break, outlaw, laser, microwave*, and *telephone* might all be either verbs or nouns. In certain circumstances, even words with primarily grammatical functions can be used as verbs or nouns, as in, "We must look to the *hows* and not just the *whys*." The process whereby a word comes to be used as a different part of speech is called **conversion** or zero derivation.

### **Functional classification**

**Linguists** recognize that the above list of eight or nine word classes is drastically simplified.<sup>[13]</sup> For example, "adverb" is to some extent a catch-all class that includes words with many different functions. Some have even argued that the most basic of category distinctions, that of nouns and verbs, is unfounded,<sup>[14]</sup> or not applicable to certain languages.<sup>[15][16]</sup> Modern linguists have proposed many different schemes whereby the words of English or other languages are placed into more specific categories and subcategories based on a more precise understanding of their grammatical functions.

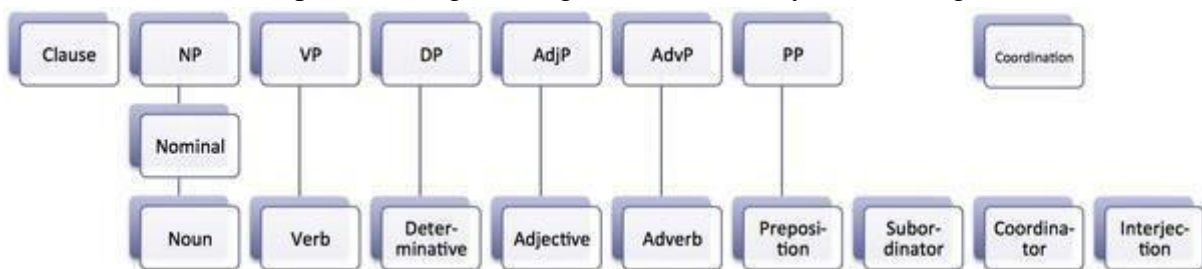
Common lexical categories defined by function may include the following (not all of them will necessarily be applicable in a given language):

- Categories that will usually be **open classes**:
  - **adjectives**
  - **adverbs**
  - **nouns**
  - **verbs** (except auxiliary verbs)
  - **interjections**
- Categories that will usually be **closed classes**:
  - **auxiliary verbs**
  - **clitics**
  - **coverbs**
  - **conjunctions**

- determiners (articles, quantifiers, demonstrative adjectives, and possessive adjectives)
- particles
- measure words or classifiers
- adpositions (prepositions, postpositions, and circumpositions)
- preverbs
- pronouns
- contractions
- cardinal numbers

Within a given category, subgroups of words may be identified based on more precise grammatical properties. For example, verbs may be specified according to the number and type of objects or other complements which they take. This is called subcategorization.

Many modern descriptions of grammar include not only lexical categories or word classes, but also *phrasal categories*, used to classify phrases, in the sense of groups of words that form units having specific grammatical functions. Phrasal categories may include noun phrases (NP), verb phrases (VP) and so on. Lexical and phrasal categories together are called syntactic categories.



A diagram showing some of the posited English syntactic categories

#### ASSIGNMENT FOR STUDENTS:

- **Submit the study report on POS tagging of English and Hindi Language with some examples.**

# **Report**

## **On**

### **POS tagging of English and Hindi Language**

#### **1. INTRODUCTION**

Natural language processing (NLP) is the process of extracting meaningful information from natural language. Part of speech (POS) tagging is considered as the one of the important tool for Natural language processing. Part of speech is a process of assigning a tag to every word in the sentences as a particular part of speech such as Noun, pronoun, adjective, verb, adverb, preposition, conjunction etc. Hindi is a natural language so there is a need to perform natural language processing on Hindi sentence.

#### **2. POS TAGGING OF ENGLISH AND HINDI LANGUAGE**

##### **2.1. POS TAGGING OF HINDI**

A system can be very large to manage means as size and functionality of a system increases, it is very difficult to handle the system. So to manage the system, generally system is divided in subsystems or modules. Modularity defines the degree to which system components can be separated or recombined. As the value of the modularity increases, the system becomes more manageable and easy to handle. The presented system has following modules.

##### **Read and Verify Hindi Text**

The very first module of system reads and verifies Hindi data. It contains a text area in GUI. Here user has to enter his Hindi text. Module reads and verifies this text. Data must be Devanagari Hindi.

##### **Split in Sentences**

This module breaks input Hindi data in individual sentences according to delimiter, which can be “Puranviraam” or “prashanvachak chinha.” In this module input will be Hindi (Devanagari) data and output will be individual Hindi sentence.

##### **Tokenize in Words**

This module breaks input Hindi data in individual words according to delimiter “space.” In this module input will be Hindi (Devanagari) data and output will be individual Hindi words. Output will be displayed in GUI.

##### **Tag Hindi Data**



## ***Language Processor***

This module tags each word of input Hindi (Devanagari) data with tags like pronoun, adverb, date, number, verb, time, etc. Words which are not tagged using corpus matcher or various rules are tagged as “SYM” tag. In this module input will be Devanagari Hindi data and output will be POS tagged Devanagari Hindi data. Output will be displayed in GUI.

### **EXAMPLE:**

#### **Splitting**

Input text to the system:

“नई दिल्ली। सिविल सेवा देने वाले अभ्यर्थियों को इस साल से आयु में दो साल का फायदा मिलेगा। उन्हें दो अतिरिक्त मौके परीक्षा देने के लिए मिलेंगे।

Output of the system:

1. नई दिल्ली।
2. सिविल सेवा देने वाले अभ्यर्थियों को इस साल से आयु में दो साल का फायदा मिलेगा।
3. उन्हें दो अतिरिक्त मौके परीक्षा देने के लिए मिलेंगे।

#### **Tokenization**

Input Text to the system:

“दो साल का फायदा मिलेगा”

Output of the system:

दो, साल, का, फायदा, मिलेगा

#### **POS Tagging**

Input Text to the system:

“सिविल सेवा देने वाले अभ्यर्थियों को इस साल से आयु में दो साल का फायदा मिलेगा।

Output of the system:

सिविल\_NNC सेवा\_NN देने\_VNN वाले\_PREP अभ्यर्थियों\_NN को\_PREP इस\_PRP  
साल\_NN से\_PREP आयु\_NN में\_PREP दो\_QFNUM साल\_NN का\_PREP फायदा\_NN  
मिलेगा\_VFM ।\_PUNC

## 2.2. POS TAGGING OF ENGLISH

### EXAMPLE:

#### **Splitting**

##### **Input text to the system:**

I was the more deceived Ophelia in *Hamlet* by William Shakespeare

##### **Output of the system:**

1. I was the more deceived
2. Ophelia in *Hamlet* by William Shakespeare

#### **Tokenization:**

##### **Input text to the system:**

Open the jar carefully

##### **Output of the system:**

Open,the,jar,carefully

#### **POS Tagging:**

##### **Input text to the system:**

Sita reads a book.

##### **Output of the system:**

Sita\_NN reads\_VB a\_DT book\_NN

**CONCLUSION:** Hence, we have done the study report on POS tagging of English and Hindi Language with some examples.