S. B. JAIN INSTITUTE OF TECHNOLOGY, MANAGEMENT & RESEARCH, NAGPUR.

(An Autonomous Institute, Affiliated to RTMNU, Nagpur)

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

To become a center for quality education in the field of Computer Science & Engineering and to create competent professionals.

Session: 2021-22(Odd)

Session:	**2021-22 (Odd)**
Subject:	**Language Processor**
Year:	**4th**
Semester:	**7th**
Name of Student:	**Jaspreet Kaur Saggu (CS18045)**
Batch:	**B2**

Practical No. 1

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

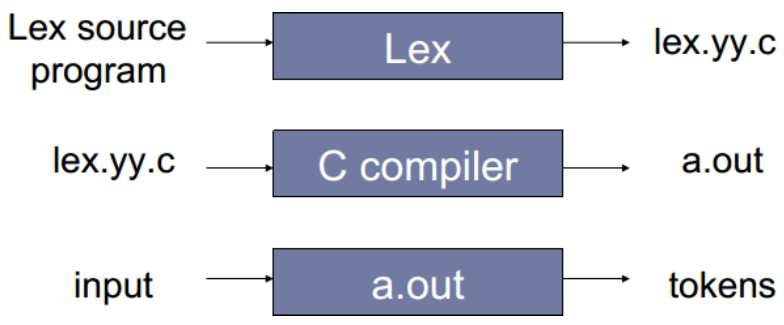
INLAB

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

OBJECTIVE / EXPECTED LEARNING OUTCOME:

- To illustrate the use of Compiler writing Tool (LEX)
- To understand, how to demonstrate the LEX program.

Diagrammatic Representation of LEX:



INLAB

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

OBJECTIVE/EXPECTED LEARNING OUTCOME:

- To illustrate the use of Compiler writing Tool (LEX)
- To understand, how to demonstrate the LEX program.

THEORY:

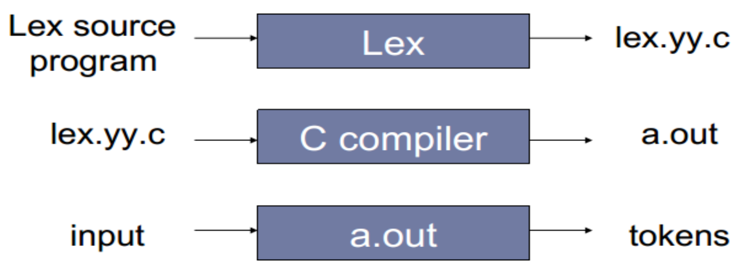
LEX

- It stands for LEXical Analyzer Generator.
- LEX is a tool for generating lexical analyzer or scanners.
- Scanners are programs that recognize lexical patterns in text. These lexical patterns are defined in a particular syntax called regular expression.

LEX Skeleton: LEX skeleton is given below:

```
%{
        Declaration Section
   }%

%%
Rule Section
%%

User Code (C Language)
```

Diagrammatic Representation of LEX:



LEX Functions:

yylex( ):

It is use to invoke lexer to start analysis.

yywrap( ):

It is called when EOF is encounter, indicate end of parsing by lexical analyser.

yymore( ):

It append next string match to current content of yytext.

yyless ( )

It removes from yytext first n char.

LEX Variables:

yytext

Text match most recently is stored.

yyleng

Number of char in text most recently match.

yylval

Associated val of current token.

yyin

This points to current file parsed by lexer.

yyout

This points to location where output of lexer will be written.

Procedure to Execute LEX Program:

These are the following sequence of commands to be executed to run a LEX programs:

- vi f1.lex
- lex f1.lex
- cc lex.yy.c
- ./a.out

CODE:

Sample programs for demonstration:

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

Sample code:

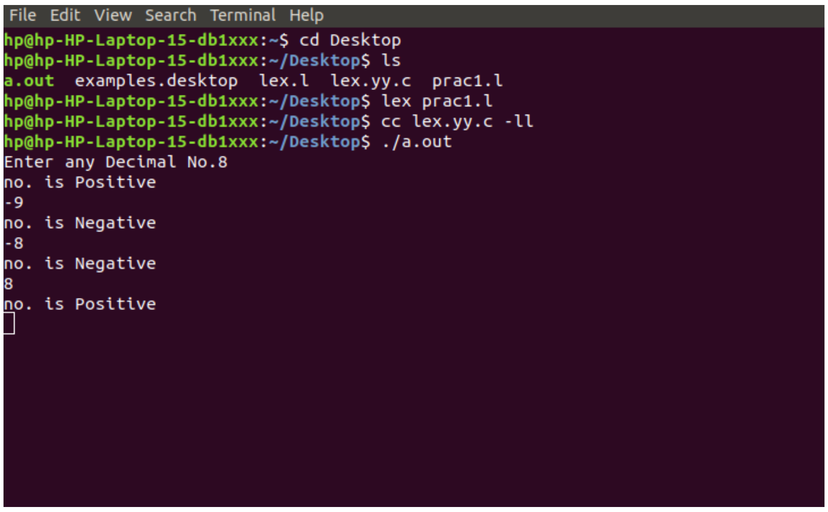
```
%{
}%
```

```
%%
\+?[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
```



```
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 Sample 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 sample program.

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.

OBJECTIVE/EXPECTED LEARNING OUTCOME:

- To build LEX Program using LEX programming Tool.
- To execute LEX Program using LEX programming Tool.

INLAB

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

OBJECTIVE/EXPECTED LEARNING OUTCOME:

- To build LEX Program using LEX programming Tool.
- To execute LEX Program using LEX programming Tool.

THEORY:

This program is just a simulation of a finite machine which is the predecessor of the compiler designing. Basically, Language Processor (LP) is developed from the deterministic finite automation and non-deterministic finite automation techniques in combination with the concept of the regular expressions, context free grammars, etc. So, this program is highly useful to understand the way a machine changes its states to finally attain its final state, thus deciding whether the string is valid under the prescribed grammar conditions designed or not.

This program simulates a finite automata that accepts only C identified keywords. Identified keywords in C means those keywords that cannot be used as variable names in declaration part of a program. We simulate the requirement in the following way mentioned here.

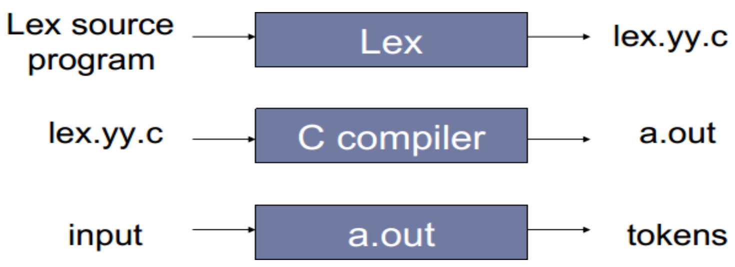
LEX

- It stands for Lexical Analyzer Generator.
- LEX is a tool for generating lexical analyzer or scanners.
- Scanners are programs that recognize lexical patterns in text. These lexical patterns are defined in a particular syntax called regular expression.

LEX Skeleton: LEX skeleton is given below:

```
%{
        Declaration Section
  }%

%%
Rule Section
%%

User Code (C Language)
```

Diagrammatic Representation of LEX:



Lex Functions:

yylex( ):

It is use to invoke lexer to start analysis.

yywrap( ):

It is called when EOF is encounter, indicate end of parsing by lexical analyser.

yymore( ):

It append next string match to current content of yytext.

yyless ( )

It removes from yytext first n char.

Lex Variables:

yytext

Text match most recently is stored.

yyleng

Number of char in text most recently match.

yylval

Associated val of current token.

yyin

This points to current file parsed by lexer.

yyout

This points to location where output of lexer will be written.

Procedure to Execute LEX Program:

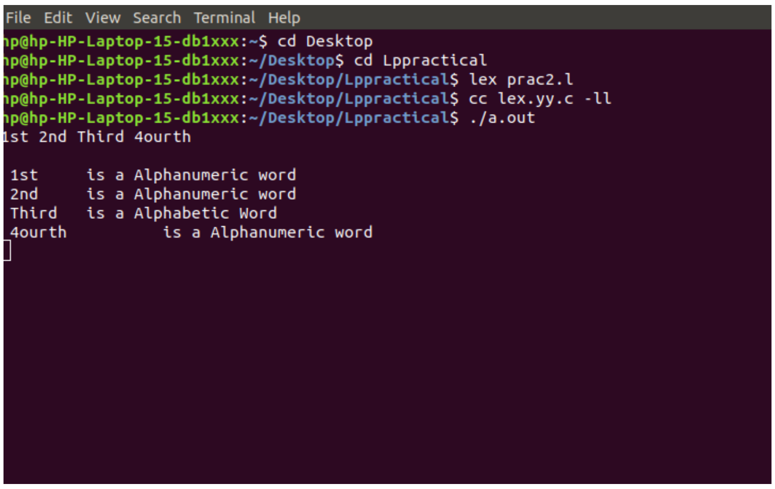
These are the following sequence of commands to be executed to run a LEX programs:

- vi f1.lex
- lex f1.lex
- cc lex.yy.c
- ./a.out

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



```
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) anddemonstration of some sample program

INLAB

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

OBJECTIVE/EXPECTED LEARNING OUTCOME:

- To illustrate the use of Compiler writing Tool (YACC)
- To understand, how to demonstrate the YACC program.

.

INLAB

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

OBJECTIVE/EXPECTED LEARNING OUTCOME:

- To illustrate the use of Compiler writing Tool (YACC)
- To understand, how to demonstrate the YACC program.

THEORY:

Why YACC?

It *is* possible to create a simple parser using Lex alone. by making extensive use of the user-defined states (i.e. start-conditions). However, such a parser quickly becomes un-maintainable, as the number of user-defined states tends to explode.

Once our input file syntax contains complex structures, such as "balanced" brackets, or contains elements which are context-sensitive, we should be considering YACC.

"Context-sensitive" in this case means that a word or symbol can have different interpretations, depending on *where* it appears in the input language. For example in C, the '*' character is used for both multiplication, and to specify indirection (ie to dereference a pointer to a piece of memory). It's meaning is "context-sensitive".

The YACC Specification

Like lex, yacc has it's own specification language. A yacc specification is structured along the same lines as a Lex specification.

```
%{
   /* C declarations and includes */
%}
   /* Yacc token and type declarations */
%%
   /* Yacc Specification
      in the form of grammer rules like this:
   */
   symbol    :    symbols tokens
                      { $$ = my_c_code($1); }
             ;
%%
   /* C language program (the rest) */
```

The Yacc Specification rules are the place where you "glue" the various tokens together that lex has conviniently provided to you.

Each grammar rule defines a symbol in terms of:

- other symbols
- tokens (or terminal symbols) which come from the lexer.

Each rule can have an associated action, which is executed *after* all the component symbols of the rule have been parsed. Actions are basically C-program statements surrounded by curly braces.

YACC FUNCTIONS:

yyparse( )

It is use to call parser.

yyerror ( )

It prints error message when rule for input cannot be found.

Execution Procedure for YACC

- vi f1.y
- yacc f1.y
- cc y.tab.c
- ./a.out

Execution Procedure for YACC in Combination with LEX

- vi f1.l
- lex f1.l
- vi   f1.y
- yacc –d f1.y
- cc lex.yy.c y.tab.c -ll
- ./a.out

CODE:

Sample programs for demonstration:

Aim of Sample Program 1: YACC program to recognize string aab, ab {$a^n b$}

Sample Code:

Lex Code

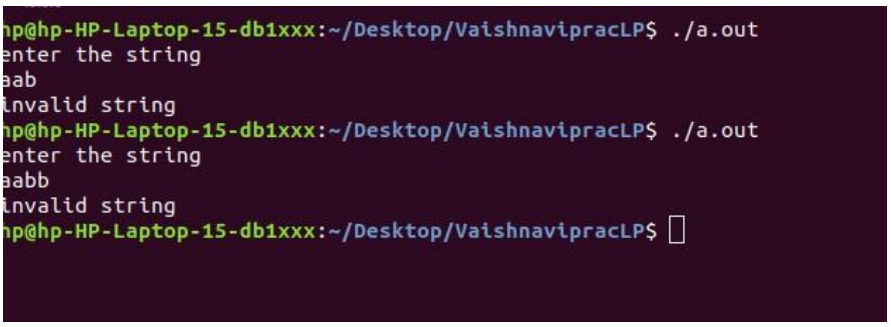
```
%{
    #include"y.tab.h"
%}
%%
[a] {return A;}
[b] {return B;}

%%
```

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 sampleprograms.

Practical No. 4

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

INLAB

AIM: Write a YACC program to recognize string bd, bbdd, bbbddd, {$b^n d^n$}.

OBJECTIVE/EXPECTED LEARNING OUTCOME:

- To build YACC Program using YACC programming Tool.
- To execute YACC Program using YACC programming Tool.

INLAB

AIM: Write a YACC program to recognize string bd, bbdd, bbbddd {$b^n d^n$}.

OBJECTIVE/EXPECTED LEARNING OUTCOME:

- To build YACC Program using YACC programming Tool.
- To execute YACC Program using YACC programming Tool.

THEORY:

YACC FUNCTIONS:

yyparse( )

It is use to call parser.

yyerror ( )

It prints error message when rule for input cannot be found.

Execution Procedure for YACC

- vi f1.y
- yacc f1.y
- cc y.tab.c
- ./a.out

Execution Procedure for YACC in Combination with LEX

- vi f1.l
- lex f1.l
- vi   f1.y
- yacc –d f1.y
- cc lex.yy.c y.tab.c -ll
- ./a.out

ALGORITHM / PROCEDURE:

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:

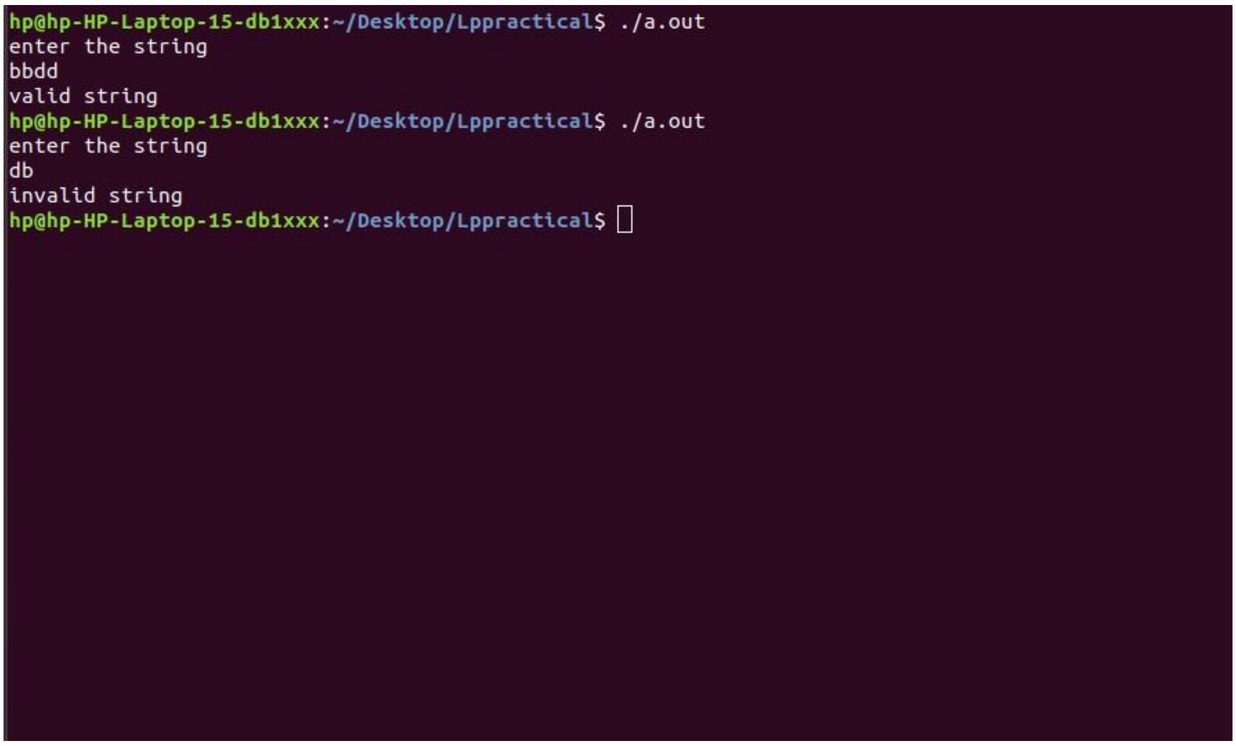
```
%{
#include<stdio.h>
#include<stdlib.h>
%}
%token A B NL
%%
stmt: S NL { printf("valid string\n");
                          exit(0); }
;
S: A S B |
;
%%

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.

OBJECTIVE/EXPECTED LEARNING OUTCOME:

- To summarize the significance of calculation of FIRST ( ) of CFG.
- To build a C program for calculation of FIRST ( ) of CFG.

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

OBJECTIVE/EXPECTED LEARNING OUTCOME:

- To summarize the significance of calculation of FIRST ( ) of CFG.
- To build a C program for calculation of FIRST ( ) of CFG.

THEORY:

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.

ALGORITHM / PROCEDURE:

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:

3. FIRST of any terminal symbol ‗a‘ is the terminal ‗a‘ itself.
4. FIRST of non terminal ‗A‘ = FIRST of ‗@‘, where A -> @ such that @ is a string containing terminals and non-terminals.

FIRST of any entity is given by FIRST (a) for terminal ‘a’ and FIRST (A) for non-terminal ‘A’.

CODE:

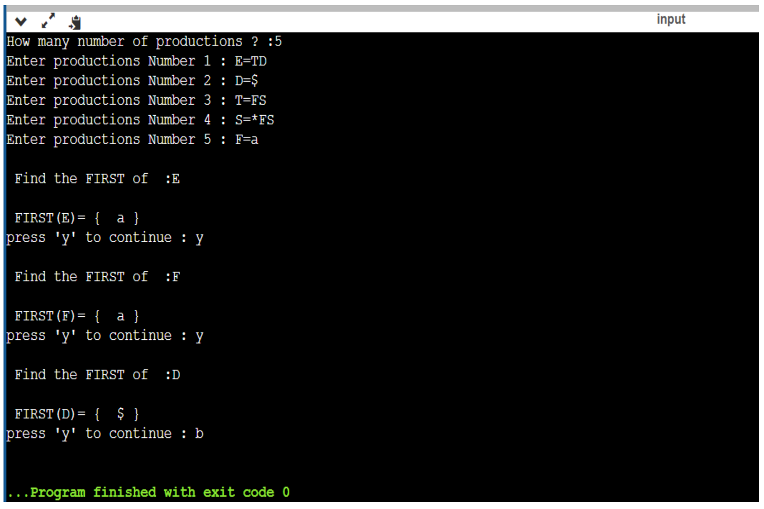
```
#include<stdio.h>
#include<ctype.h>
void FIRST(char[],char );
void addToResultSet(char[],char);
int numOfProductions;
char productionSet[10][10];
main()
{
   int i;
   char choice;
   char c;
   char result[20];
```

```
    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.

OBJECTIVE/EXPECTED LEARNING OUTCOME:

- To summarize the significance of calculation of FOLOW ( ) of CFG.
- To build a C program for calculation of FOLLOW ( ) of CFG.

.

INLAB

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

OBJECTIVE/EXPECTED LEARNING OUTCOME:

- To summarize the significance of calculation of FOLOW ( ) of CFG.
- To build a C program for calculation of FOLLOW ( ) of CFG.

THEORY:

FOLLOW function is used to find out the following symbols that are possible from both terminal and non-terminal symbols and these are the symbols that lead the next variables into the operation being done. 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.

ALGORITHM / PROCEDURE:

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

1. FOLLOW of any non-terminal symbol _A‘ is _$‘ iff A is the Starting symbol of the Grammar.
2. FOLLOW of non terminal _B‘ = FIRST of _@‘, where A -> @B & such that _@‘ and _&‘ is a string containing terminals and non-terminals with FIRST (B) not containing EPSILON (e).
3. FOLLOW of non terminal _B‘ = FOLLOW of _A‘, where A -> @B & such that _@‘ and _&‘ is a string containing terminals and non-terminals with FIRST (B) contains EPSILON (e).

FOLLOW of any entity is given by FOLLOW(A) for non-terminal ‘A’.

CODE:

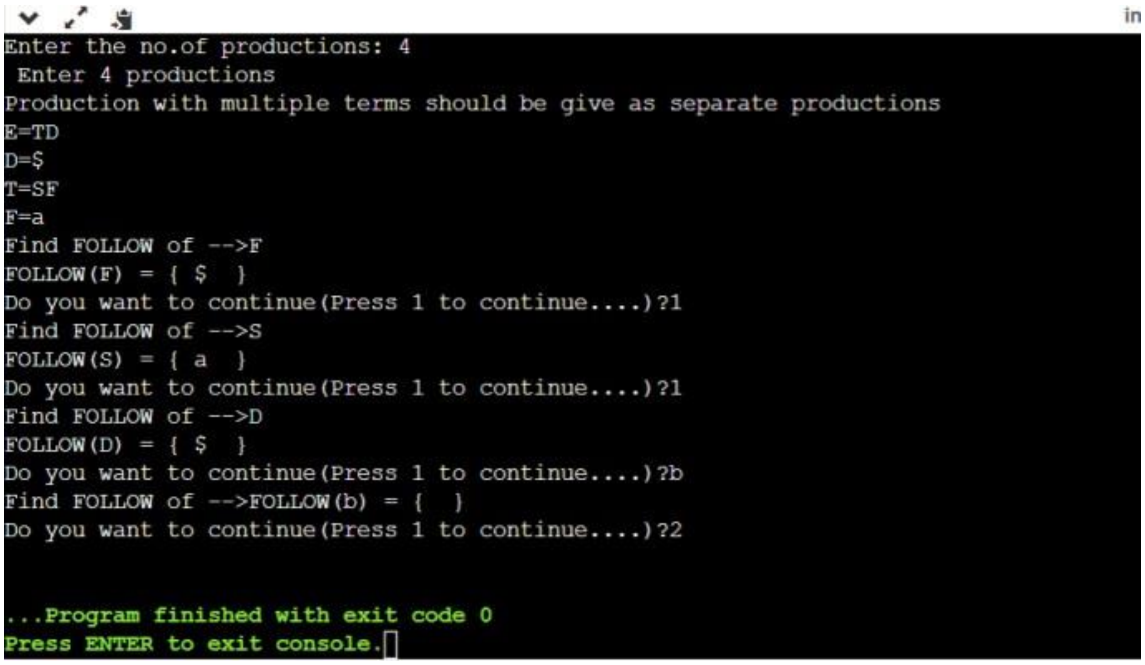
```
#include<stdio.h>
#include<string.h>
int n,m=0,p,i=0,j=0;
char a[10][10],followResult[10];
void follow(char c);
void first(char c);
void addToResult(char);
int main()
```

```
{
 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
```



```
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.
```

Conclusion: Hence, we Write a C program to calculate FOLLOW ( ) of given grammar.

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).

OBJECTIVE/EXPECTED LEARNING OUTCOME:

- To demonstrate how to implement Predictive parsing Table (LL(1)).
- To build program for Predictive Parsing Table (LL(1)).

INLAB

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

OBJECTIVE/EXPECTED LEARNING OUTCOME:

- To demonstrate how to implement Predictive parsing Table (LL(1)).
- To build program for Predictive Parsing Table (LL(1)).

THEORY:

Predictive parse table is used in designing compilers for various languages through one of the most prominent and widely used compiler namely predictive parser. We use the parsing technique developed through the predictive to detect whether a given string can be accepted or not. Even the first compiler C, has an in built predictive parser which is operated through the calculation of the first and follow functions. Not only C compiler, many other modern compilers are being developed based on the action sequence generated through a predictive parser.

ALGORITHM / PROCEDURE:

The following program is to represent the moves of a Parser through a Parsing Table. There are some predefined rules to configure the moves of a parser. We need to maintain a stack array for defining the productions and an input buffer array to accept the input string in a sequential manner. The conditions for deciding whether an input operation is syntactically right or not are:

Step 1. If the element in the top of stack is equal to _$' and also the present element in the input buffer is _$', then directly convey that the string is syntactically valid one.

Step 2. If the element in the top of the stack is equal to the present element in the input buffer other that _$', then POP out the top most element of the stack and increment the input buffer's current position, i.e. make the current element as the next element in the sequence.

Step 3. If the element in the top of the stack is not equal to the present element in the input buffer, then, move all the elements of the input buffer in reverse order to the stack.

The program that follows does all the above verifications and finally conveys whether a given input string of a respective Grammar is valid with respect to syntax or not.

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]=='@')
{
```

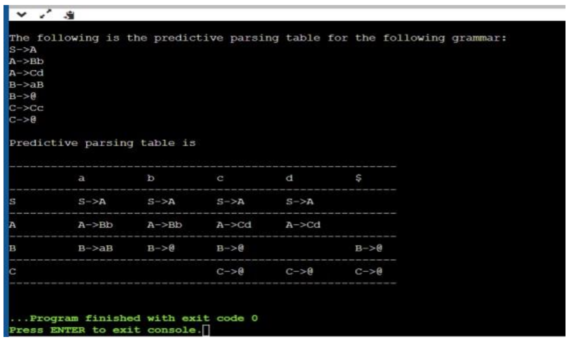
```
k=strlen(follow[i]);
for(j=0;j<k;j++)
strcpy(table[numr(prol[i][0])+1][numr(follow[i][j])+1],prod[i]);
}
}
}
strcpy(table[0][0]," ");
strcpy(table[0][1],"a");
strcpy(table[0][2],"b");
strcpy(table[0][3],"c");
strcpy(table[0][4],"d");
strcpy(table[0][5],"$");
strcpy(table[1][0],"S");
strcpy(table[2][0],"A");
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->@
C->Cc
C->@

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->@      B->@                B->@
-------------------------------------------------------
C                             C->@      C->@      C->@
-------------------------------------------------------


...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 FreeGrammar (CFG).

Practical No. 8

Aim: Write a C Programto 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.

OBJECTIVE/EXPECTED LEARNING OUTCOME:

- To demonstrate how to implement Predictive parsing Table (LL(1)).
- To build program for implementation of Predictive Parsing Table (LL(1)).

INLAB

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

OBJECTIVE/EXPECTED LEARNING OUTCOME:

- To demonstrate how to implement Predictive parsing Table (LL(1)).
- To build program for implementation of Predictive Parsing Table (LL(1)).

THEORY:

Predictive parse table is used in designing compilers for various languages through one of the most prominent and widely used compiler namely predictive parser. We use the parsing technique developed through the predictive to detect whether a given string can be accepted or not. Even the first compiler C, has an in built predictive parser which is operated through the calculation of the first and follow functions. Not only C compiler, many other modern compilers are being developed based on the action sequence generated through a predictive parser.

ALGORITHM / PROCEDURE:

The following program is to represent the moves of a Parser through a Parsing Table. There are some predefined rules to configure the moves of a parser. We need to maintain a stack array for defining the productions and an input buffer array to accept the input string in a sequential manner. The conditions for deciding whether an input operation is syntactically right or not are:

Step 4. If the element in the top of stack is equal to _$‘ and also the present element in the input buffer is _$‘, then directly convey that the string is syntactically valid one.

Step 5. If the element in the top of the stack is equal to the present element in the input buffer other that _$‘, then POP out the top most element of the stack and increment the input buffer‘s current position, i.e. make the current element as the next element in the sequence.

Step 6. If the element in the top of the stack is not equal to the present element in the input buffer, then, move all the elements of the input buffer in reverse order to the stack.

The program that follows does all the above verifications and finally conveys whether a given input string of a respective Grammar is valid with respect to syntax or not.

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;
}
```

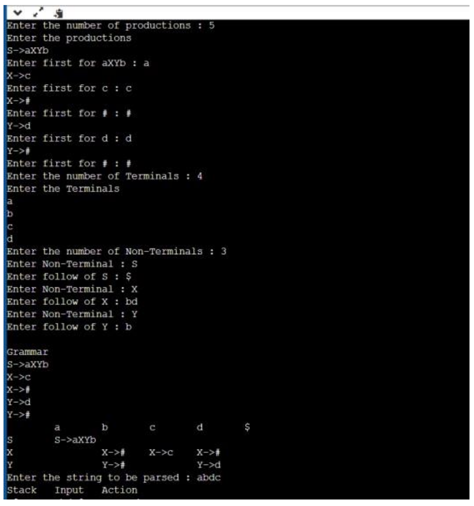
OUTPUT: Expected

Enter the Predictive parsing table:

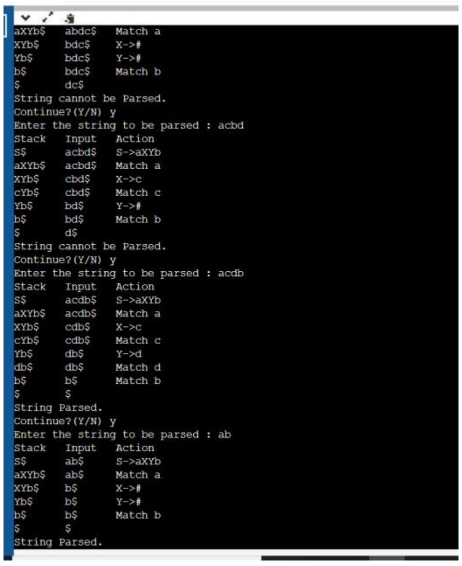
NT T	a	b	c	d	$
S	S->A	S->A	S->A	S->A	
A	A->Bb	A- >Bb	A- >Cd	A->Cd	
B	B->aB	B->@	B->@		B->@
C	C->@	C->@	C->@		

Enter the input string to be parsed: a@b

1. S→ A
2. S→ Bb
3. S→ aBb
4. S→ 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
X->#
Enter first for # : #
Y->d
Enter first for d : d
Y->#
Enter first for # : #
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
X->#
Y->d
Y->#
        a       b       c       d       $
S       S->aXYb
X               X->#    X->c    X->#
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$
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$
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.

OBJECTIVE/EXPECTED LEARNING OUTCOME:

- To demonstrate, how to convert an arithmetic expression into TAC (Three Address Code)
- To build a program for Syntax Directed Translation Scheme (SDTS) of Arithmetic expression to generate TAC.

.

INLAB

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

OBJECTIVE/EXPECTED LEARNING OUTCOME:

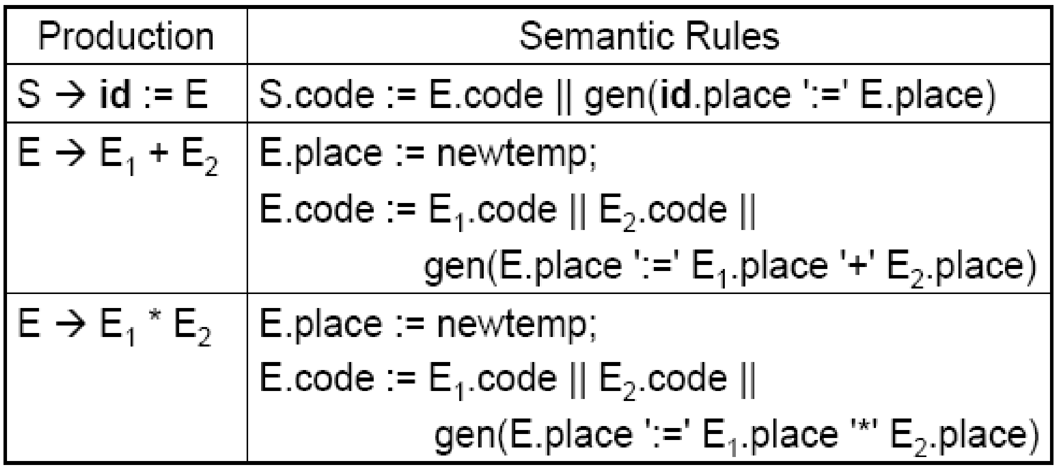
- To demonstrate, how to convert an arithmetic expression into TAC (Three Address Code).
- To build a program for Syntax Directed Translation Scheme (SDTS) of Arithmetic expression to generate TAC.

THEORY:

Generating Three-Address Code:

- Temporary names are made up for the interior nodes of a syntax tree
- The synthesized attribute S.code represents the code for the assignment S
- The non-terminal E has attributes:
  - E.place is the name that holds the value of E
  - E.code is a sequence of three-address statements evaluating E
- The function newtemp() returns a sequence of distinct names
- The function newlabel() returns a sequence of distinct labels

SDTS for Assignment Statement(S) and arithmetic expression (E):



Production	Semantic Rules
S → **id** := E	S.code := E.code || gen(**id**.place ':=' E.place)

E → E_1 + E_2	E.place := newtemp;
	E.code := E_1.code || E_2.code ||
	gen(E.place ':=' E_1.place '+' E_2.place)

E → E_1 * E_2	E.place := newtemp;
	E.code := E_1.code || E_2.code ||
	gen(E.place ':=' E_1.place '*' E_2.place)

CODE:

```
#include<stdio.h>
#include<string.h>
void main()
{
   int i,j,l,count=0;
   char ex[10],expe[10] ,exp1[10],rev[10],rev1[10];
   printf("\nEnter the expression with arithmetic operator:");
   scanf("%s",ex);
   strcpy(expe,ex);
```

```
    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=%c%ctemp\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

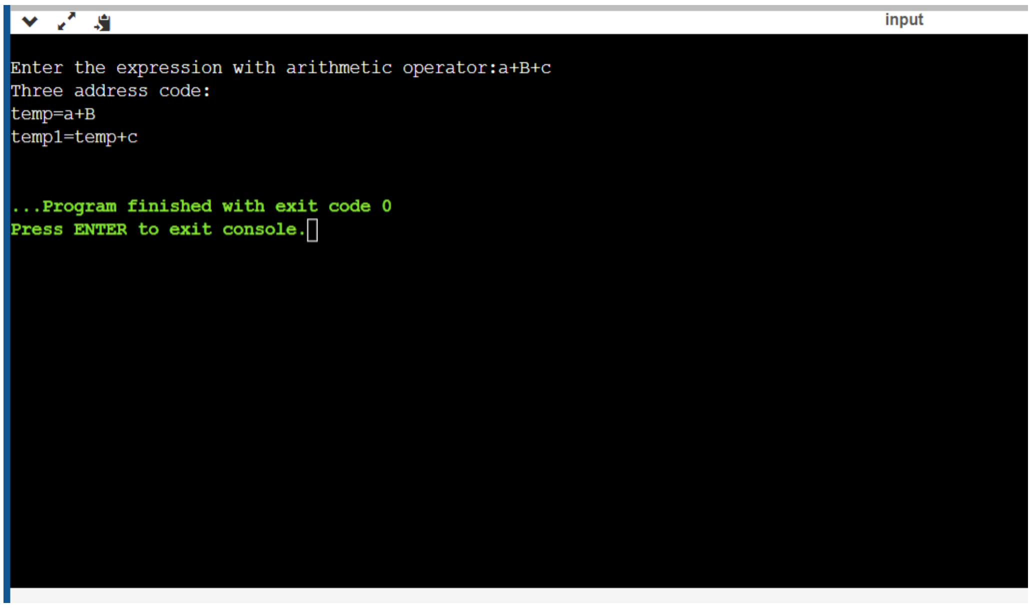
Enter the expression: a=a+b*c

TAC is:

1. T1 = b * c

2. T2 = a + T1

3. a = T2



```
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.

OBJECTIVE/EXPECTED LEARNING OUTCOME:

- To relate the prerequisite of course i.e. Computer Architecture and Organization with the Language Processor.
- To generate assembly code from TAC using some algorithm.

INLAB

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

OBJECTIVE/EXPECTED LEARNING OUTCOME:

- To relate the prerequisite of course i.e. Computer Architecture and Organization with the Language Processor.
- To generate assembly code from TAC using some algorithm.

THEORY:

It is a last phase of any compiler, which is connected directly to the processor or machine. Input for this phase is the optimized three address code. In code generation phase machine dependent optimization will also be performed which is based on General purpose register optimization. For that there are some algorithms and tools are also available like:

- Directed Acyclic Graph
- Heuristic DAG Algorithm
- Labelling Algorithm.
- Simple code generation algorithm.
- Peephole Optimization.

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;
}
```

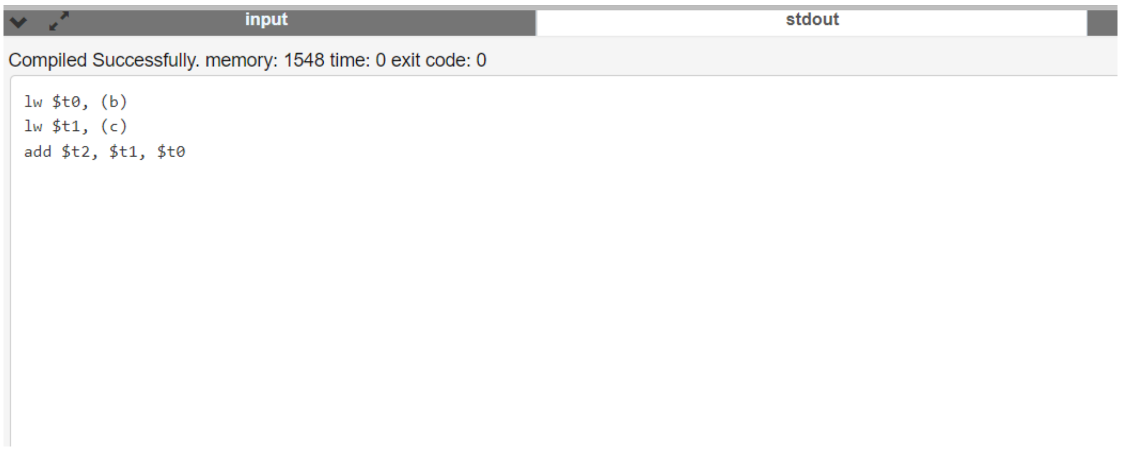
OUTPUT: Expected

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
```



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.

OBJECTIVE/EXPECTED LEARNING OUTCOME:

- To understand the application of LEX and YACC together.
- To write system programs using compiler writing tools.

INLAB

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

OBJECTIVE/EXPECTED LEARNING OUTCOME:

- To understand the application of LEX and YACC together.
- To write system programs using compiler writing tools.

THEORY:

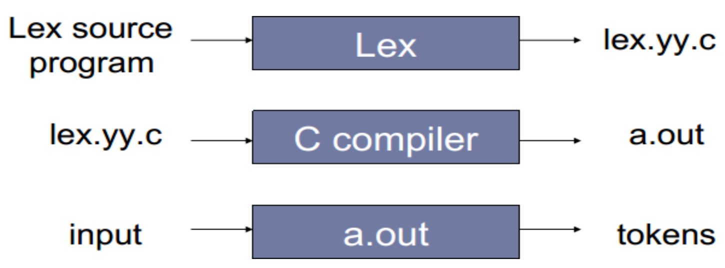
LEX

- It stands for LEXical Analyzer Generator.
- LEX is a tool for generating lexical analyzer or scanners.
- Scanners are programs that recognize lexical patterns in text. These lexical patterns are defined in a particular syntax called regular expression.

LEX Skeleton: LEX skeleton is given below:

```
%{
        Declaration Section
   }%
%%
Rule Section
%%

User Code (C Language)
```

Diagrammatic Representation of LEX :



Lex Functions:

yylex( ):

It is use to invoke lexer to start analysis.

yywrap( ):

It is called when EOF is encounter, indicate end of parsing by lexical analyser.

yymore( ):

It append next string match to current content of yytext.

yyless ( )

It removes from yytext first n char.

Lex Variables:

yytext

Text match most recently is stored.

yyleng

Number of char in text most recently match.

yylval

Associated val of current token.

yyin

This points to current file parsed by lexer.

yyout

This points to location where output of lexer will be written.

YACC:

It stands for yet another compilers compiler. It takes tokens as input generated by LEX by dividing input stream & groups them together logically.

YACC TEMPLATE:

```
%{
   Header Declaration
%}
%Token declaration
%%
  Parsing Rule
%%
C code section
```

YACC has three parts

Defination Section
It set up execution environment in which parser will operate..
Rule section
It contains rules for parser. C code section
It mainly contains main function.

Types of Token in YACC

There are three main types of tokens as
% token (No precedence)
% left (Precedence given to input on left of this section)
% right (Precedence given to input on right of this section)

YACC FUNCTIONS:
Yyparse( )
It is use to call parser.
Yyerror ( )
It prints error message when rule for input cannot be found.

Execution Procedure for YACC

- vi f1.y
- yacc f1.y
- cc y.tab.c
- ./a.out

Execution Procedure for YACC in Combination with LEX

- vi f1.l
- lex f1.l
- vi   f1.y
- yacc –d f1.y
- cc lex.yy.c y.tab.c -ll
- ./a.out

CODE:

LEX Code:

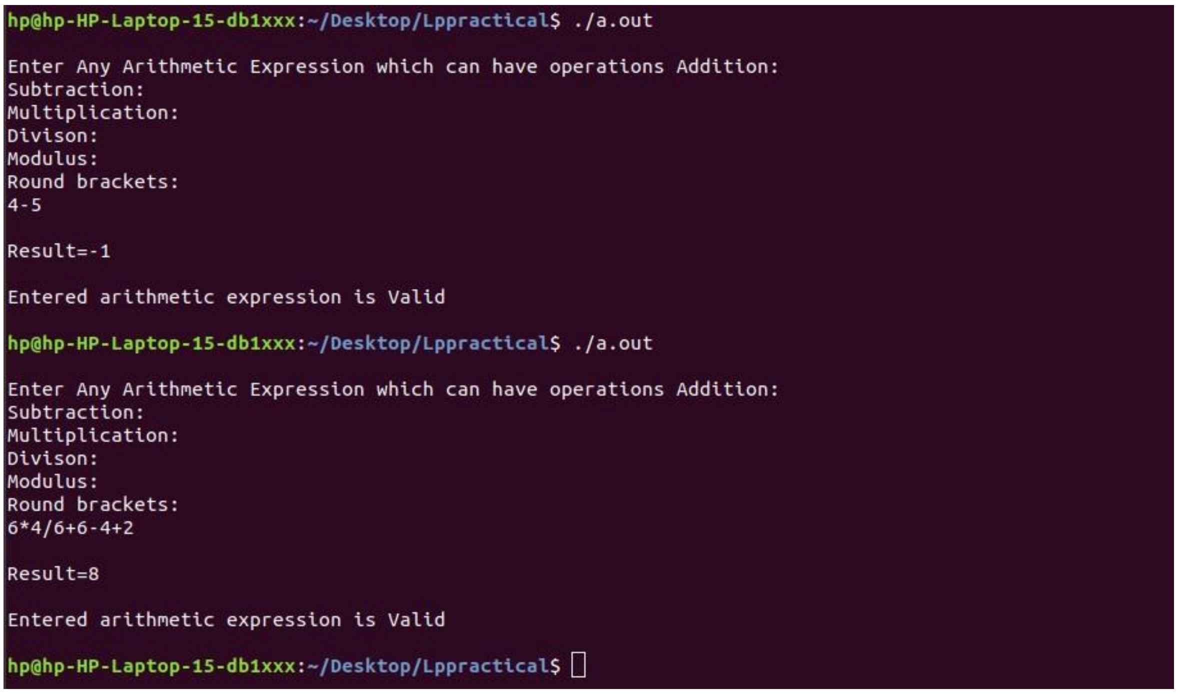
```
%{
#include<stdio.h>
#include "y.tab.h"
extern int yylval;
%}
%%
[0-9]+ {
                yylval=atoi(yytext);
                return NUMBER;
       }
[\t] ;
[\n] return 0;
. return yytext[0];
%%
int yywrap()
{
return 1;
}
```

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 {$$=$1-$3;}
|E'*'E {$$=$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("\nEntered arithmetic expression is Valid\n\n");
}
void yyerror()
{
printf("\nEntered 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. 12

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

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

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[12] 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.[13] 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,[14] or not applicable to certain languages.[15][16] 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.

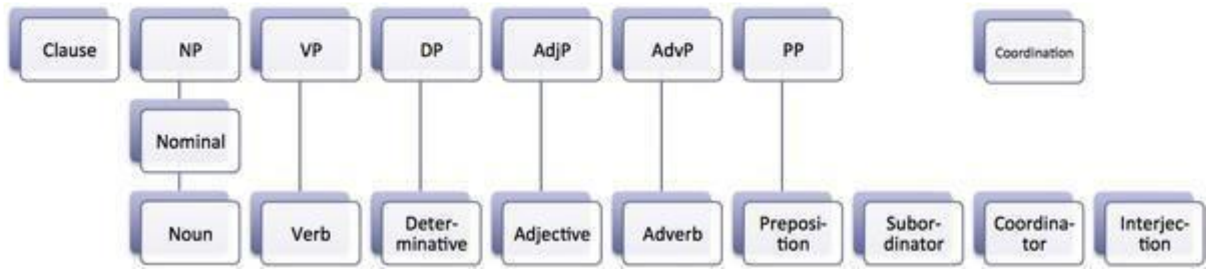
Common lexical categories defined by function may include the following (not all of them will necessarily be applicable in a given language):

- 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 “Puranviram” 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

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

CONLUSION: Hence, we have done the study report on POS tagging of English and Hindi Language with someexamples.