

*Language Processor*

**S. B. Jain Institute of Technology, Management & Research, Nagpur**

**Department of Computer Science & Engineering**

**Session 2022-23**



**(Language Processor)**

**(BECSE402P)**

# **LAB MANUAL**

**Year: IV**

**Semester: VII**

*Department of Computer Science & Engineering, S.B.J.I.T.M.R., Nagpur*

## **Practical No. 1**

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

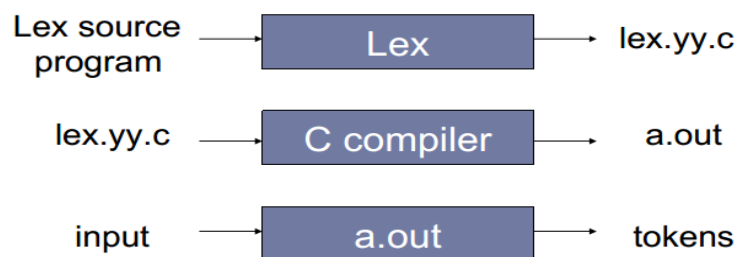
**INLAB**

**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.

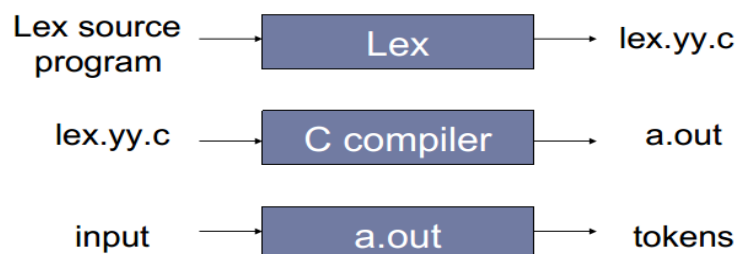
**LEX Skeleton:** LEX skeleton is given below:

```
% {  
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.

yylen

Number of char in text most recently match.

yyval

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 fl.lex
  - lex fl.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

**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:****DISCUSSION AND VIVA VOCE:**

Q 1: What is the difference between token and lexeme?

Q 2: What is lexical analyzer?

Q 3: Which compiler is used for lexical analyzer?

Q 4: What is the output of Lexical analyzer?

Q 5: What is LEX source Program?

**REFERENCE:**

- Lab Manual of Compiler Design (Institute of Aeronautical Engineering, Dundigal, Hyderabad)
- [https://en.wikipedia.org/wiki/Lex\\_\(software\)](https://en.wikipedia.org/wiki/Lex_(software)).

## **Practical No. 2**

**Aim:** Write a C Program to design and simulate a FA that accepts keywords like char, case, int etc.



**INLAB**

**AIM:** Write a C Program to design and simulate a FA that accepts keywords like char, case, int etc.

**OBJECTIVE/EXPECTED LEARNING OUTCOME:**

- To apply the concept of Finite Automata (FA).
- To Simulate a Finite Automata with compiler of any programming language.

**FLOWCHART:**

**INLAB**

**AIM:** Write a C Program to design and simulate a FA that accepts keywords like char, case, int etc.

**OBJECTIVE/EXPECTED LEARNING OUTCOME:**

- To apply the concept of Finite Automata (FA).
- To Simulate a Finite Automata with compiler of any programming language.

**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.

**ALGORITHM / PROCEDURE:**

**Step 1.** Gather all the information regarding the states and the transitions which need to be done from one state to another state.

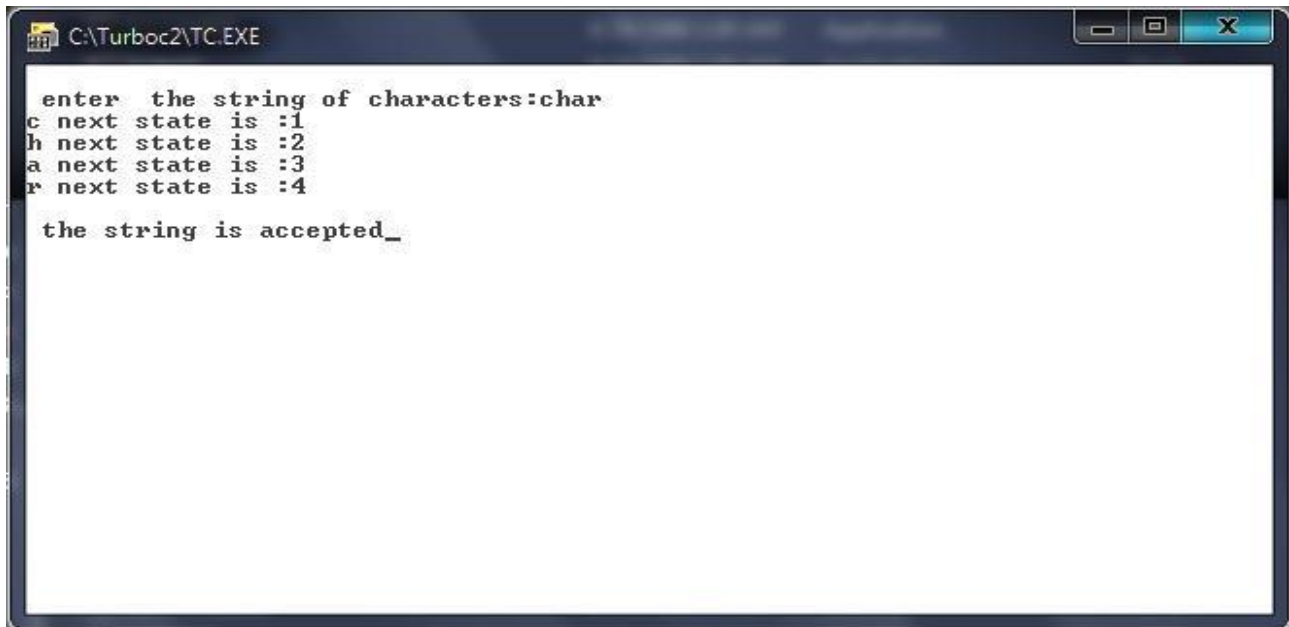
**Step 2.** Consider the case of the keyword, “char”, to move from initial state 0 to 1, we need to check whether the first input variable given is “c” or not and then we move to state 2 if the next variable in sequence is “h”. Similarly, we check up to the last input symbol such that if the State Transition of the last input symbol fetches to a Final State.

**Step 3.** Then, we can say that the input sent by the user is accepted and hence we give a justification to our program.

**Consider only 3 Keywords for program: char, case & int**

**CODE:**

**OUTPUT:**



```
enter the string of characters:char
c next state is :1
h next state is :2
a next state is :3
r next state is :4

the string is accepted_
```

**CONCLUSION:**

**DISCUSSION AND VIVA VOCE:**

- Q 1: What is Finite Automata?
- Q 2: What is the application of Finite Automata?
- Q 3: List of TUPLES of Finite Automata?
- Q 4: What is Deterministic and non-deterministic FA?
- Q 5: What is the relation between Finite Automata and LEX source Program?

**REFERENCE:**

- <http://www.indiastudychannel.com/resources/150257-Compiler-Design-Lab-Programs-B-Tech-Computer.aspx>
- Book: Introduction To Automata Theory Languages, and Computation by John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman,—2nd ed.

### **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.

**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 grammar 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 conveniently 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:

yyvsparse( )

It is use to call parser.

yyerror ( )

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

#### **Execution Procedure for YACC**

---

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

#### **Execution Procedure for YACC in Combination with LEX**

---

- vi fl.l
  - lex fl.l
  - vi fl.y
  - yacc -d fl.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<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(yparse()==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

---

**CONCLUSION:**

*Department of Computer Science & Engineering, S.B.J.I.T.M.R., Nagpur*



**DISCUSSION AND VIVA VOCE:**

- 1 Q 1: What is the difference between LEX and YACC?
- 1 Q 2: What is Syntax analyzer?
- Q 3: Which compiler is used for Syntax analyzer?
- Q 4: What is the output of Syntax analyzer?
- 1 Q 5: What is YACC source Program?

**REFERENCE:**

- Lab Manual of Compiler Design (Institute of Aeronautical Engineering, Dundigal, Hyderabad)
- [https://luv.asn.au/overheads/lex\\_yacc/yacc.html#yacc](https://luv.asn.au/overheads/lex_yacc/yacc.html#yacc)
- <http://dinosaur.compilertools.net/> (The Lex & Yacc Page)

### **Practical No. 4**

**Aim:** Write a YACC program to recognize string aabb, ab, aaabbb { $a^n b^n$ }.

**INLAB AIM:** Write a YACC program to recognize string aabb, ab, aaabbb { $a^n b^n$ }.

**OBJECTIVE/EXPECTED LEARNING OUTCOME:**

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

**FLOWCHART:**

**INLAB**

**AIM:** Write a YACC program to recognize string aabb, ab, aaabbb {a<sup>n</sup>b<sup>n</sup>}.

**OBJECTIVE/EXPECTED LEARNING OUTCOME:**

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

**THEORY:**

**YACC FUNCTIONS:**

yparse( )

It is use to call parser.

yyerror ( )

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

**Execution Procedure for YACC**

---

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

**Execution Procedure for YACC in Combination with LEX**

---

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

**ALGORITHM / PROCEDURE:**

**CODE:**

**OUTPUT: Expected**

1. Enter string aab  
Its valid string
  2. Enter string aabb  
Its not valid string
- 

**CONCLUSION:**

**DISCUSSION AND VIVA VOCE:**

- 2        What is the application of YACC tool?
- 3        A YACC tool belongs to which operating System?
- 4        What is YACC program input?
- 5        What is YACC program Output?
- 6        What are the different parsing tools for another operating system?

**REFERENCE:**

- Lab Manual of Compiler Design (Institute of Aeronautical Engineering, Dundigal, Hyderabad)
- [https://luv.asn.au/overheads/lex\\_yacc/yacc.html#yacc](https://luv.asn.au/overheads/lex_yacc/yacc.html#yacc)
- <http://dinosaur.compilertools.net/> (The Lex & Yacc Page)

## **Practical No. 5**

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

**INLAB 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.

**FLOWCHART:**

**INLAB**

**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:

1. FIRST of any terminal symbol 'a' is the terminal 'a' itself.
2. FIRST of non terminal 'A' = FIRST of '@', where  $A \rightarrow @$  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:**

**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: E
```



First (E) = a

Press Y/N to continue: Y

Find first of: A

First(A) = c

Press Y/N to continue: N

### **CONCLUSION:**

### **DISCUSSION AND VIVA VOCE:**

Q1:What is the significance to calculate FIRST() of CFG?

Q2:What is the output of FIRST() function?

Q3:Can we calculate FIRST() for left recursive CFG?

Q4:Is it necessary to calculate FIRST() of CFG for parsing technique?

Q5: Calculate FIRST() of the given CFG:

### **REFERENCE:**

- <http://www.indiastudychannel.com/resources/150257-Compiler-Design-Lab-Programs-B-Tech-Computer.aspx>
- Book: Compiler Design by O.G. Kakde, Laxmi Publications, 2006.
- Lab Manual of Compiler Design (Institute of Aeronautical Engineering, Dundigal, Hyderabad)

## **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 FOLLOW ( ) of CFG.
- To build a C program for calculation of FOLLOW ( ) of CFG.

**FLOWCHART:**

**INLAB**

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

**OBJECTIVE/EXPECTED LEARNING OUTCOME:**

- To summarize the significance of calculation of FOLLOW ( ) 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  $A \rightarrow @B$  & 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  $A \rightarrow @B$  & 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:**

**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:**

**DISCUSSION AND VIVA VOCE:**

Q1:What is the significance to calculate FOLLOW() of CFG?

Q2:What is the output of FOLLOW() function?

Q3:Can we calculate FOLLOW() for left recursive CFG?

Q4:Is it necessary to calculate FOLLOW() of CFG for parsing technique?

Q5:Can FOLLOW() function return empty set?

Q6: Calculate FIRST() of the given CFG:

**REFERENCE:**

- <http://www.indiastudychannel.com/resources/150257-Compiler-Design-Lab-Programs-B-Tech-Computer.aspx>
- Book: Compiler Design by O.G. Kakde, Laxmi Publications, 2006.
- Lab Manual of Compiler Design (Institute of Aeronautical Engineering, Dundigal, Hyderabad).

### **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)).

**FLOWCHART:**

**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 than '\$', 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:**

**OUTPUT: Expected**



Enter the CFG:

S->A  
A->Bb  
A->Cd  
B->aB  
B->@  
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->@  |       |      |

### CONCLUSION:

### DISCUSSION AND VIVA VOCE:

- Q1:** What are the application of Predictive Parser (LL(1))?  
**Q2:** What do you mean by LL(1) Parser?  
**Q3:** Differentiate Backtracking and Non-Backtracking Parser?  
**Q4:** What is TOP-DOWN Parser?  
**Q5:** Can left recursive grammar be parsed by LL(1) parser?

### REFERENCE:

- <https://web.cs.wpi.edu/~kal/PLT/PLT4.1.2.html>
- Book: Compiler Design by O.G. Kakde, Laxmi Publications, 2006.
- Lab Manual of Compiler Design (Institute of Aeronautical Engineering, Dundigal, Hyderabad).

## **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.

**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)).

**FLOWCHART:**

**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 than '\$', 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:**

**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.  $S \rightarrow a@b$

**CONCLUSION:**

**DISCUSSION AND VIVA VOCE:**

- Q1:** What are the application of Predictive Parser (LL(1))?  
**Q2:** What do you mean by LL(1) Parser?  
**Q3:** Differentiate Backtracking and Non-Backtracking Parser?  
**Q4:** What is TOP-DOWN Parser?  
**Q5:** Can left recursive grammar be parsed by LL(1) parser?

**REFERENCE:**

- <https://web.cs.wpi.edu/~kal/PLT/PLT4.1.2.html>
- Book: Compiler Design by O.G. Kakde, Laxmi Publications, 2006.
- Lab Manual of Compiler Design (Institute of Aeronautical Engineering, Dundigal, Hyderabad).

## **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:**

## *Language Processor*

- 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.

### **FLOWCHART:**

### **INLAB**

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

*Department of Computer Science & Engineering, S.B.J.I.T.M.R., Nagpur*

**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  |
|---------------------------|---|
| $S \rightarrow id := E$   | $S.code := E.code \parallel gen(id.place ':=' E.place)$   |
| $E \rightarrow E_1 + E_2$ | $E.place := newtemp;$<br>$E.code := E_1.code \parallel E_2.code \parallel$<br>$gen(E.place ':=' E_1.place '+' E_2.place)$ |
| $E \rightarrow E_1 * E_2$ | $E.place := newtemp;$<br>$E.code := E_1.code \parallel E_2.code \parallel$<br>$gen(E.place ':=' E_1.place '*' E_2.place)$ |

**CODE:**

**OUTPUT: Expected**

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



**TAC is:**

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

**CONCLUSION:**

**DISCUSSION AND VIVA VOCE:**

- Q1:** What is Intermediate Code?
- Q2:** What is the need of Intermediate code?
- Q3:** Why Three Address Code (TAC) preferred to use as Intermediate Code?
- Q4:** Which are the optional phases of Compiler?
- Q5:** What is SDTS?

**REFERENCE:**

- **Book:** Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, "Compilers Principles, Techniques and Tools", Pearson Education, 2<sup>nd</sup> edition. 2010.
- **Book:** Compiler Design by O.G. Kakde, Laxmi Publications, 2006.
- Lab Manual of Compiler Design (Institute of Aeronautical Engineering, Dundigal, Hyderabad).
- Language Processors Lab Manual (MIT, Manipal).

## **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:**

*Department of Computer Science & Engineering, S.B.J.I.T.M.R., Nagpur*

## *Language Processor*

- 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.

### **FLOWCHART:**

### **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:**

**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
```

**CONCLUSION:**

**DISCUSSION AND VIVA VOCE:**

**Q1:** What is assembly code?

**Q2:** What is the difference between syntax tree and DAG?

**Q3:** Why heuristic Dag algorithm is used in code generation phase?

**Q4:** What are the different addressing mode?

**REFERENCE:**

- **Book:** Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, “Compilers Principles, Techniques and Tools”, Pearson Education, 2<sup>nd</sup> edition. 2010.
- **Book:** Compiler Design by O.G. Kakde, Laxmi Publications, 2006.
- Lab Manual of Compiler Design (Institute of Aeronautical Engineering, Dundigal, Hyderabad).

## **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.

**FLOWCHART:**

**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:**

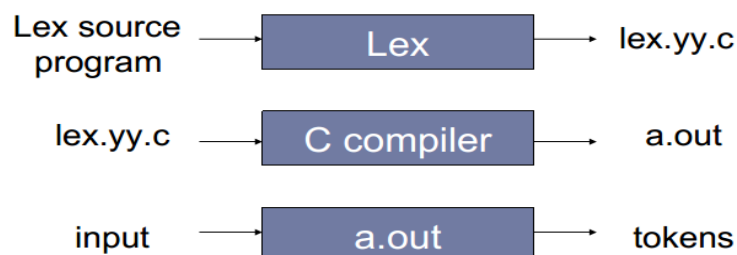
### **LEX**

- 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 used to invoke lexer to start analysis.

`yywrap( ):`

It is called when EOF is encountered, indicating end of parsing by lexical analyser.

`yymore( ):`

It appends 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.

yylen

Number of char in text most recently match.

yyval

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**

Definition 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 fl.y
  - yacc fl.y
  - cc y.tab.c
  - ./a.out
- 

#### **Execution Procedure for YACC in Combination with LEX**

---

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

**CODE:**

**OUTPUT:**

**CONCLUSION:**

**DISCUSSION AND VIVA VOCE:**

- Q1:** What is the difference between token and lexeme?  
**Q2:** What is lexical analyzer?  
**Q3:** Which compiler is used for lexical analyzer?  
**Q4:** What is the difference between LEX and YACC?  
**Q5:** What is LEX source Program?

**REFERENCE:**

- Lab Manual of Compiler Design (Institute of Aeronautical Engineering, Dundigal, Hyderabad)
- [https://en.wikipedia.org/wiki/Lex\\_\(software\)](https://en.wikipedia.org/wiki/Lex_(software)).
- [https://luv.asn.au/overheads/lex\\_yacc/yacc.html#yacc](https://luv.asn.au/overheads/lex_yacc/yacc.html#yacc)
- <http://dinosaur.compilertools.net/> (The Lex & Yacc Page)

## **Practical No. 12**

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

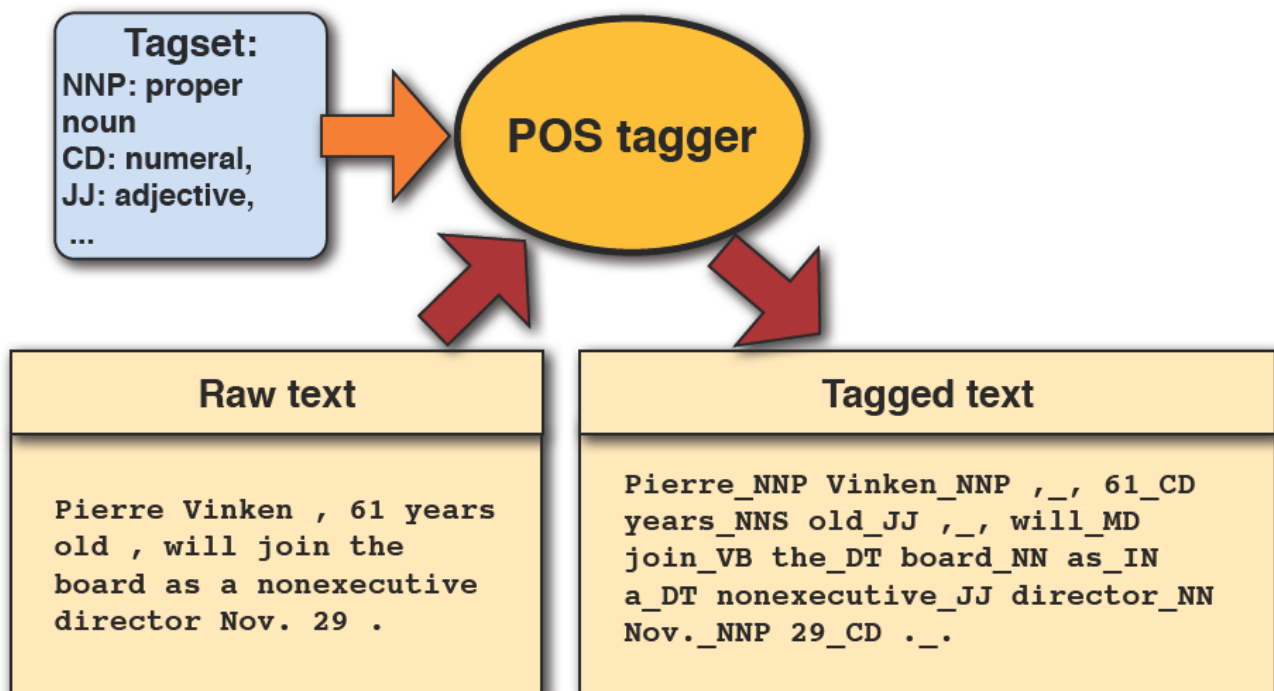
**INLAB AIM:** Demonstration of Stanford POS tagger (English Language) (NLP).

**OBJECTIVE/EXPECTED LEARNING OUTCOME:**

*Department of Computer Science & Engineering, S.B.J.I.T.M.R., Nagpur*

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

## POS tagging



**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 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. 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, or not applicable to certain languages. 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.

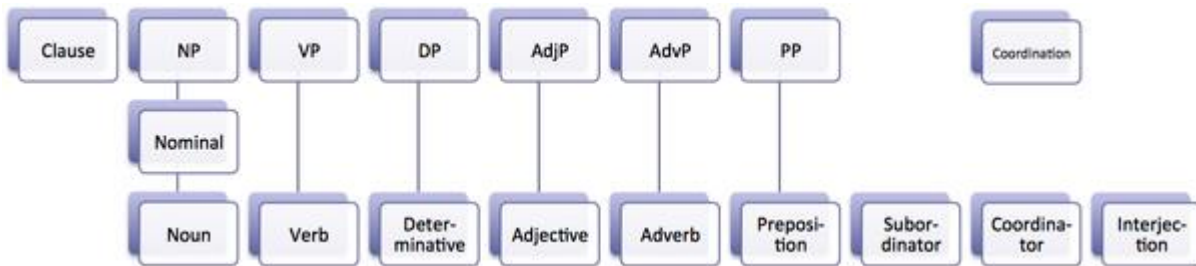
Common lexical categories defined by function may include the following (not all of them will necessarily be applicable in a given language):

- 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

### **CONCLUSION:**

### **DISCUSSION AND VIVA VOCE:**

- Q1:** What is tagging?
- Q2:** What is NLP?
- Q3:** How it is different from LP?
- Q4:** What are the phases of NLP?
- Q5:** How the POS tagger will be helpful to resolve word sense disambiguation?

### **REFERENCE:**

- [https://en.wikipedia.org/wiki/Part-of-speech\\_tagging](https://en.wikipedia.org/wiki/Part-of-speech_tagging)
- Part-of-Speech Tagging examples handout.