

## UNIT-II

### Syntax Analysis :-

\* Introduction.

\* Role of Syntax Analysis.

### Introduction :-

It is the second phase of the compilation.

It checks for the syntax of language.

- Syntax analyzer takes the tokens from the lexical analyzer and groups them in some programming structure called "syntax tree (or) parse tree".
- If any syntax cannot be recognized then the syntax error will be generated.

### Definition:-

- A passing over syntax analysis is a process which takes string "w" and produce either a parse tree (or) generates the semantic error.
- It is also called "parser".

Ex:- consider the source program statement  $a := b + 10$

Here the lexical analyzer reads the above statement and broken it into the set of tokens like 'A' is identifier

A - is identifier

$:=$  - Assignment Operator

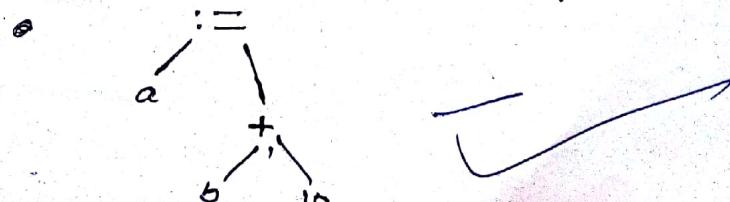
B - Identifier

$+$  - Assignment Operator

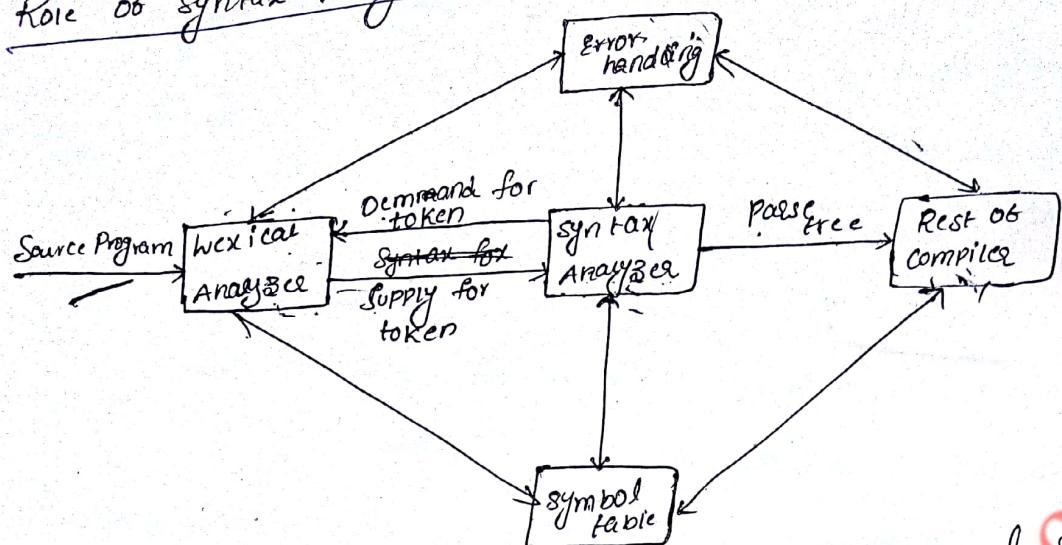
10 - number (or) Constant.

- Now the syntax analyzer connect the above tokens from lexical analyzer and arrange them into a structure is called "Parse tree (or) Syntax tree".

$$\Rightarrow a := b + 10.$$



## Role of Syntax Analyzer : (or) Parser :-



In the process of compilation the parser and lexicae analyse work together that means.

- When the parser requires string of tokens it invokes lexical analyzer.
- In turn the lexical Analyze supply tokens to Syntax Analyzer.

→ The parser collects sufficient number of tokens and build a parse tree.

→ It finds syntactical errors at the time of construction of Parse tree.

→ These errors are recovered by Error handler.

## Context free Grammar :-

\* Introduction \* Derivation and Parse tree \* Ambiguous Grammars

### Introduction:-

A context free Grammar "G" is a four tuples like

$$G = (V, T, P, S)$$

$V \rightarrow$  set of non terminal symbols.

$T \rightarrow$  set of terminal symbols.

$P \rightarrow$  set of Production rules of the form

$\alpha, \beta$  where  $\alpha \in V^*$ ,  $\beta \in T^*$

WWW.KVRSOFTWARES.BLOGSPOT.COM

$s \rightarrow$  start symbol.

Ex:- Let the language  $W = a^n b^n, n \geq 1$

$s \rightarrow a^n b^n$

minimum string = ab

$s \rightarrow a^n b^n$

$\rightarrow a^{n-1} b^{n-1} b$

$\rightarrow a a^{n-2} b^{n-2} b b$

P:  $s \rightarrow a^i s b^j$

$s \rightarrow a^i s b^j$

$s \rightarrow a^i b^j$

From the above  $G = (V, T, P, S)$  where is a context free grammar.

Where  $V \rightarrow$  set of {S}

$T \rightarrow$  set of {a, b}

$P \rightarrow$  {asb, ab}

$S \rightarrow$  start symbol {s}

→ Derivation and Parse tree :-

Derivation from 'S' means generation of a string 'W'

Form 'S'. For constructing derivation two things are

Important.

1. Choice of non-terminal. Form several others.
2. choice of rule from production rules for corresponding non-terminal.

→ Definition of Derivation Tree.

Let  $G = (V, T, P, S)$  be a context free grammar.

The Derivation tree is a tree which can be constructed.

by following properties.

1. The root node has label 'S'.
2. Every vertex can be derived from {VUTS}

3. If there exist a vertex 'A' with children  $x_1, x_2, x_3, \dots, x_n$   
then there should be a production rule.

$$A \rightarrow x_1 x_2 x_3 \dots x_n$$

4. The leaf nodes are from set T, and internal nodes are from set 'V'.

### → 1. Left Most Derivation (LMD)

In left most derivation the left most non terminal is replaced by a terminal (or) non-terminal in each step begining with a start symbol.

### → Right Most Derivation (RMD):-

In Right most derivation the Right most non terminal is replaced by a terminal (or) non-terminal in each step begining with a start symbol.

~~Ex:-~~ Consider the grammar given below

$$\begin{aligned} E &\rightarrow E + E \\ E &\rightarrow E - E \\ E &\rightarrow E * E \\ E &\rightarrow E / E \\ E &\rightarrow a \cup b \end{aligned}$$

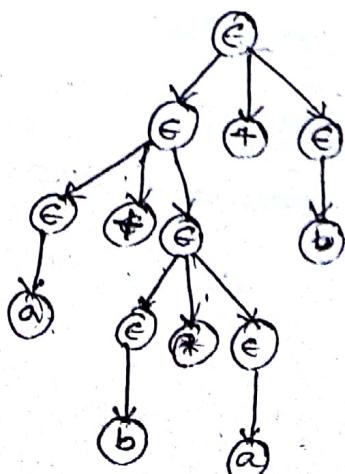
Obtain 1. Left Most Derivation 2. Right Most Derivation

3. Parse tree for the input string a+b \* a+b.

### (a) Left Most Derivation:-

$$\begin{aligned} E &\rightarrow E + E \\ &\rightarrow E + E + E \\ &\rightarrow a + E + E \\ &\rightarrow a + E * E + E \\ &\rightarrow a + b * E + E \\ &\rightarrow a + b * a + E \\ &\rightarrow a + b * a + b \end{aligned}$$

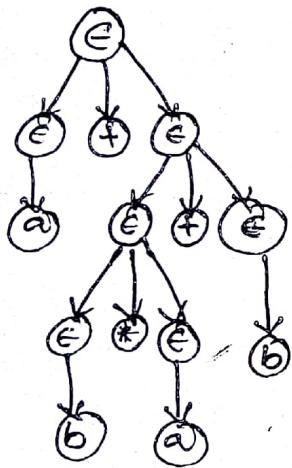
### Parse tree:-



Q. Right Most Derivation:-  $a+b \Rightarrow a+b$ .

$$\begin{aligned}\epsilon &\rightarrow \epsilon + \epsilon \\&\rightarrow \epsilon + \epsilon + \underline{\epsilon} \\&\rightarrow \epsilon + \underline{\epsilon} + b \\&\rightarrow \cancel{\epsilon} + a + b \\&\rightarrow \epsilon + \epsilon * \underline{\epsilon} + b \\&\rightarrow \epsilon + \underline{\epsilon} * a + b \\&\rightarrow \cancel{\epsilon} + b * a + b \\&\rightarrow a + b * a + b\end{aligned}$$

Parse tree:-



Q. consider the grammar given below  
 $S \rightarrow (\omega) \mid a$   
 $\omega \rightarrow S, S \mid s$ .  
Input string is  $(a, (a;a))$ .

→ writing a context-free Grammar:

- \* Lexical Analysis vs Syntax Analysis
- \* Classification of Parsing techniques.
- \* Problems with Parsing Techniques.

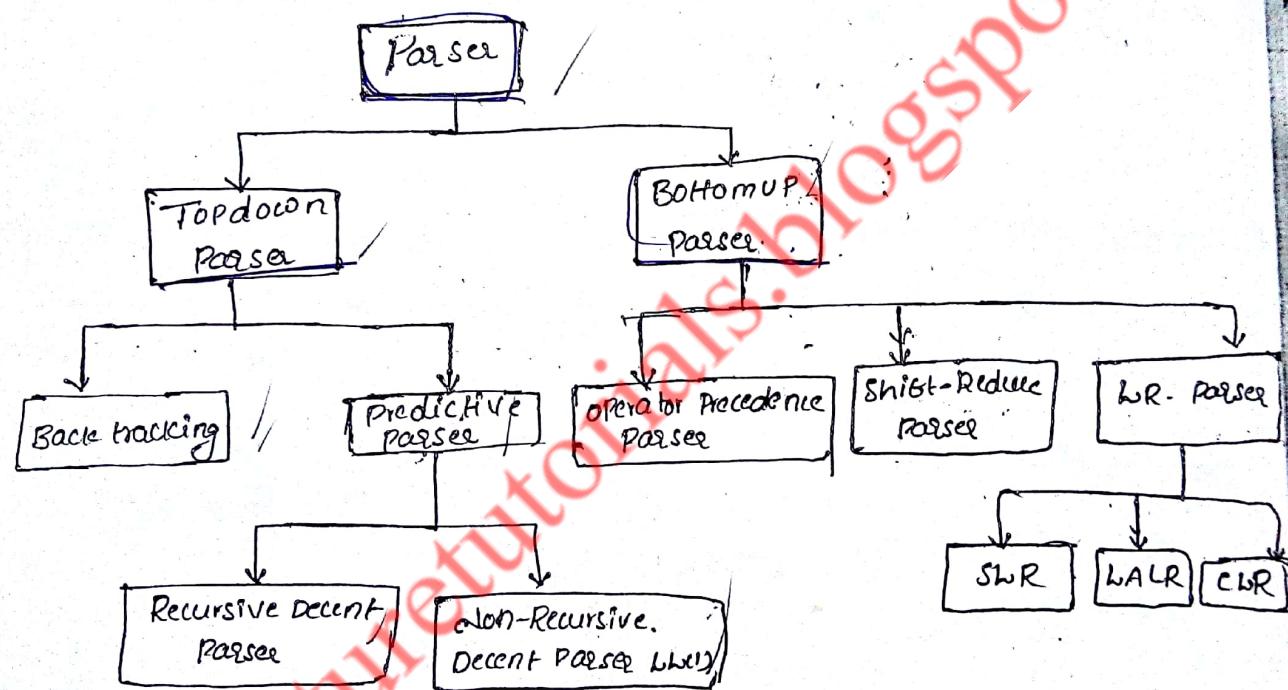
1. Back tracking.

2. Left Recursion.

3. Left factoring.

4. Ambiguity.

→ Classification of Parsing techniques:-



TOP down Parser:-

The process of constructing a syntax tree (or) Parse tree from root node to leaf node is called "TOPDOWN Parser".

TOP down parser classified into two types.

1. Back tracking    2. Predictive Parser.

→ Back tracking :-

Back tracking is a technique in which for expansion.

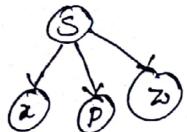
expansion of non terminal symbol we choose one alternative and if some mismatch occurs then we try another alternative if any.

Ex:- consider the grammar  $S \rightarrow xPz$   
 $P \rightarrow ywY$

also we obtain an input string  $xywz$  from the above.

Grammar:

$\rightarrow 1$

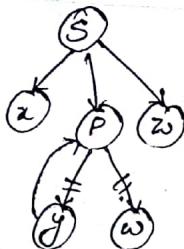


Here we take the first production

$S \rightarrow xPz$ , then the corresponding parse tree is  $S \rightarrow xPz$ .

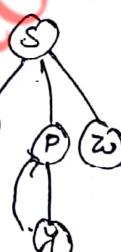
$\rightarrow 2$ . also non-terminal symbol 'P' is replaced by the first alternative of it that is  $P \rightarrow yw$  then the parse tree is

$S \rightarrow xPz$   
 $P \rightarrow xyw$



This string doesn't derived the given input string. So we move backward to 'P' and remove the corresponding branch. Then it and apply another alternative of it. Then the corresponding tree is

$S \rightarrow xPz$   
 $P \rightarrow xyw$

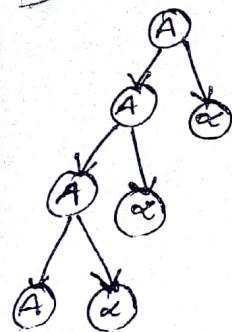


This parsing technique increases lot of overhead in implementation of parse tree. So we need to eliminate the backtracking by modifying the grammar.

→ left Recursion:-

A left Recursive grammar is a grammar which contains the production rule is like  $A \rightarrow A\alpha$  where  $A \in V$  and  $\alpha \in (V \cup T)^*$ .

If left Recursion is present in the grammar then the top down parser can enter into infinity loop like



\*\*\* elimination of left Recursion :-

To eliminate left Recursion we need to modified the grammar.

Let ' $G$ ' =  $(V, T, P, S)$  be a CFG with the productions rule having left recusssion.

$$\begin{array}{l} A \rightarrow A\alpha \\ A \rightarrow B \end{array}$$

then we eliminate

the left Recursion by rewriting: the production rule has

$$\begin{array}{l} 1. A \rightarrow BA' \\ 2. A' \rightarrow \alpha A' \\ 3. A' \rightarrow \epsilon \end{array}$$

$$\begin{array}{l} A \rightarrow A\alpha \\ A \rightarrow B \end{array}$$

Ex:- consider the grammar

$$\begin{array}{l} E \rightarrow E + T \\ E \rightarrow T B \\ T \rightarrow T * F \\ T \rightarrow F \\ F \rightarrow (E) \\ F \rightarrow id \end{array} \quad \left. \begin{array}{l} A \rightarrow A\alpha \\ A \rightarrow B \end{array} \right\} \quad \begin{array}{l} A = E, \alpha = +T \\ A = T, \alpha = *F \\ A = F \end{array}$$

Eliminate left recursion from the given grammar.

$\therefore A \rightarrow$   
 $A: \text{Step 1: } E \rightarrow E + T \quad T \rightarrow T * E \quad F \rightarrow (E)$   
 $E \rightarrow T \quad T \rightarrow F \quad F \rightarrow id.$   
 we can map this grammar production rule with

the rule  $A \rightarrow A\alpha, A \rightarrow B$ . where  $A = E$

$$\alpha = +T$$

$$B = T$$

Now we can eliminate left Recursion

$$1. A \rightarrow BA' \quad \hookrightarrow E \rightarrow TE'$$

$$3. A' \rightarrow \epsilon \quad \hookrightarrow E' \rightarrow +TE'$$

$$2. A' \rightarrow \alpha A' \quad \hookrightarrow E' \rightarrow +TE'$$

W

After eliminating left recursion the new Production

rules are:

$$\epsilon \rightarrow T\epsilon'$$

$$\epsilon' \rightarrow +TE'$$

$$\epsilon' \rightarrow \epsilon$$

$T \rightarrow FT^*$  similarly for the rules

$$T \rightarrow T^*F$$

$$T \rightarrow F$$

we can map this grammar rule with the rule

$$A \rightarrow Ad$$

$$A \rightarrow B$$

where  $A = T$

$$d = *F$$

$$B = F$$

know we can eliminate left recursion.

1.  $A \rightarrow BA'$   
    $\hookrightarrow T \rightarrow FT'$

2.  $A' \rightarrow dA'$   
    $\hookrightarrow T' \rightarrow *FT'$

3.  $A' \rightarrow \epsilon$   
    $\hookrightarrow T' \rightarrow \epsilon$

Therefore the grammar without left recursion is

$$\epsilon \rightarrow TE'$$

$$\epsilon' \rightarrow +TE'$$

$$\epsilon' \rightarrow \epsilon$$

$$T \rightarrow FT'$$

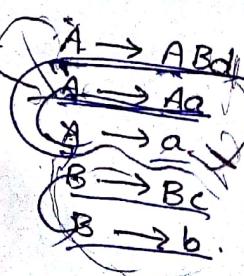
$$T' \rightarrow *FT'$$

$$T \rightarrow \epsilon$$

$$F \rightarrow (\epsilon)$$

$$F \rightarrow id.$$

Q) Consider the Grammar



remove left recursion!

The

$S \rightarrow A \rightarrow ABd$	$A \rightarrow Aa$	$B \rightarrow Bc$
$A \rightarrow a^L$	$A \rightarrow a^I$	$B \rightarrow b$
1. $A \rightarrow aA^I L$	1. $A \rightarrow aA^I /$	1. $B \rightarrow bB^I$
2. $A^I \rightarrow BdA^I$	2. $A^I \rightarrow aA^I$	2. $B^I \rightarrow cB^I$
3. $A^I \rightarrow \epsilon$	3. $A^I \rightarrow \epsilon$	3. $B^I \rightarrow \epsilon$

$\rightarrow$  Left-factoring:



4. left - factoring :-

4. Left-factoring :-

The process of factoring out the common prefix of alternative let  $A \rightarrow \alpha B_1 | \alpha B_2 | \alpha B_3 | \dots | \alpha B_n$  are 'n' number of 'A' Production rules. and ' $\alpha$ ' is not equal to null. after left factoring the grammar will become

1.  $A \rightarrow \alpha A'$

2.  $A' \rightarrow B_1 | B_2 | B_3 | \dots | B_n$

Ex :- consider the Grammar

above Grammae. with common Prefix Part.

Sol: consider the production rule with common factors to allow Person left factoring. Then

We can map this Grammar rules with the Rules

$$A \rightarrow \alpha B_1 \mid \alpha B_2$$

where  $A = S$ ,  $\alpha = i \in S$ ,  $B = E$ ,  $B_2 = ES$

The new production rules are

$$A \rightarrow \alpha B$$

$s \rightarrow i \notin s$

$$2, A' \xrightarrow{\quad} B_1 | B_2 \\ 3, \begin{array}{c} \cancel{S} \xrightarrow{\quad} a \\ \cancel{S} \xrightarrow{\quad} b \end{array}$$

The grammar after least factoring is

$$\begin{aligned} S &\rightarrow i \in \Sigma S' \\ S' &\rightarrow \epsilon | \epsilon S \\ S &\rightarrow a \\ \epsilon &\rightarrow b. \end{aligned}$$

Q, To the left factoring in the following Grammar.

$$\begin{aligned} A &\rightarrow aAB | aA | a \\ B &\rightarrow bB | b. \end{aligned}$$

$$\text{Sol:- } A \rightarrow aAB^{\frac{B_1}{B_2}} | aA^{\frac{B_2}{B_3}} | a.$$

$$\text{where } \begin{array}{l} A = \underline{A} \\ \alpha = a \\ B_1 = AB \\ B_2 = \underline{aA} \\ B_3 = \epsilon \end{array}$$

After left factoring the new production rule is.

$$\begin{array}{lll} A \rightarrow \alpha A' & \text{Q, } A' \rightarrow B_1 | B_2 | B_3 & \text{3, } bB | b \\ A \rightarrow \alpha A' & A' \rightarrow AB | \underline{aA} | \epsilon. & \begin{array}{l} A = B \\ \alpha = b \\ B_1 = B \\ B_2 = \epsilon. \end{array} \end{array}$$

∴ the grammar after left factoring is.

$$\begin{array}{l} A \rightarrow \alpha A' \\ A' \rightarrow AB | \underline{aA} | \epsilon. \end{array}$$

$$\begin{array}{l} A \leftrightarrow \alpha \\ B \rightarrow b.B' \\ B' \rightarrow B | \epsilon. \end{array}$$

→ Ambiguity :-

A Grammal which has more than one left most derivation

(or) Right most derivation (or) Parse tree for the same input string is called "Ambiguous grammar."

~~Ex:-~~ Consider the grammar which has more than one left most derivation for the input string ~~s+s+s~~ ~~s+s+s~~.

$$\begin{array}{l} A \rightarrow Ad \\ A \rightarrow B. \end{array}$$

$$S \rightarrow S + S$$

11

1. LMD :-

$$\begin{array}{l} S \rightarrow S + S \\ \rightarrow \underline{a} + S \\ \rightarrow \underline{a} + \underline{S} + S \end{array}$$

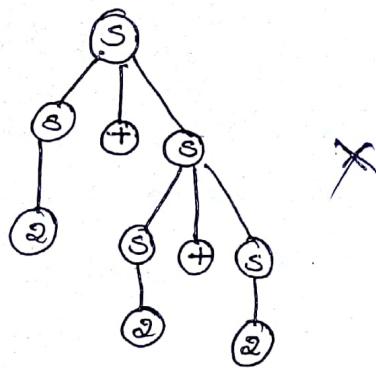
$$\rightarrow 2 + 2 + S$$

$$\rightarrow 2 + 2 + 2.$$

$$A \rightarrow \alpha A$$

$$A \rightarrow B$$

Parse tree :-



ea (doesn't current col)

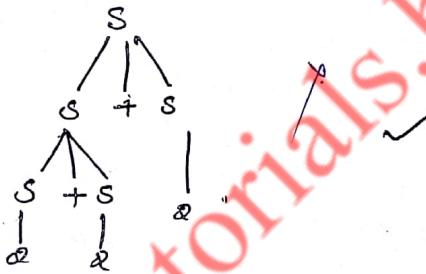
Q, LMD

Draw back :-

The computer may be confused at the time of computing mathematical expressions due to the grammar is Ambiguous.

Q, WMD :-

$$\begin{aligned} S &\rightarrow S + S \\ &\rightarrow S + S + S \\ &\rightarrow 2 + S + S \\ &\rightarrow 2 + 2 + S \\ &\rightarrow 2 + 2 + 2 \end{aligned}$$



for Removing ambiguity

If the grammar as left associative operators (+, -, \*, /, %)  
then induce left Recursion

$$\begin{aligned} A &\rightarrow A\alpha \\ A &\rightarrow B \end{aligned}$$

If the grammar as Right associative operator (=, ↑, )

then induce the Right Recursion.  $A \rightarrow \alpha A$   
 $A \rightarrow B$

Ex:- consider the Grammar

$$S \rightarrow S + S$$

$$S \rightarrow S * S$$

$S \rightarrow 2$ . And the SLP string is  $2 + 2 * 2$ .

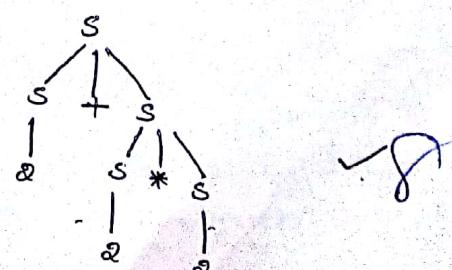
WMD :-  $S \rightarrow S + S$

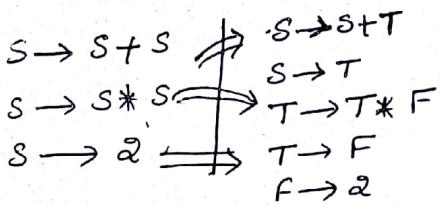
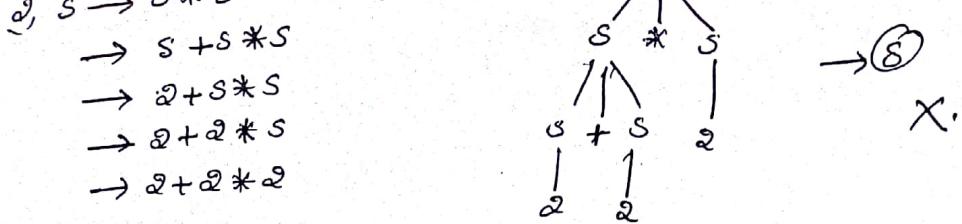
$$\rightarrow 2 + S$$

$$\rightarrow 2 + S * S$$

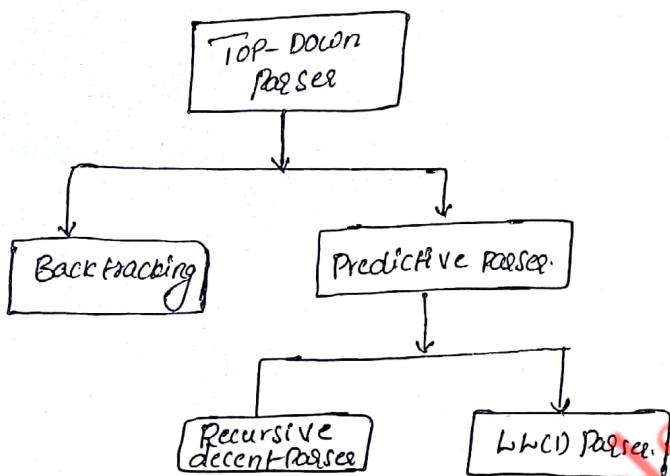
$$\rightarrow 2 + 2 * S$$

$$\rightarrow 2 + 2 * 2$$





$\rightarrow$  Top-down Parser:



Recursive decent parser:

A Parser that uses collection of recursive procedures for parsing the given input string is called "Recursive decent parser".

In this type of parser the CFG is used to build the recursive procedures.

The RHS of the Production rule is directly converted to a program.

For each non-terminal a separate procedure is written and body of the procedure is RHS of the corresponding non-terminal.

Steps for Construction of Recursive decent Parser:

The RHS of the Production rule is directly converted into Program code symbol by symbol.

- Step 1 :- If the iIP symbol is non-terminal then a call to the procedure corresponding to that non-terminal symbol.
- Step 2 :- If the iIP symbol is terminal then it is matched with the locate symbol from the input.
- Step 3 :- If the production rule has many alternatives then all this alternatives as to be combined into a single body of Procedure.
- Step 4 :- The Parser should be activated by a procedure corresponding to start symbol.

Ex :- construct a recursive decent parser for the following grammar.

$$\begin{array}{l} E \rightarrow E + T \\ \quad \quad | \\ E \rightarrow T \\ \quad \quad | \\ T \rightarrow T F \\ \quad \quad | \\ T \rightarrow F \\ F \rightarrow F^* / a / b \end{array}$$

Note :-

The recursive decent parser is works on a cFG with out left recursion.

Sol :- The Given grammar is

$$\begin{array}{l} S \rightarrow E + T \\ E \rightarrow T \\ T \rightarrow T F \\ T \rightarrow F \\ F \rightarrow F^* / a / b \end{array}$$

The above grammar constant left recursion.

so, before constructing RDP we should eliminate Left Recursion from the given grammar.

Elimination of left Recursion :-

$$\begin{array}{l} 1, E \rightarrow E + T \\ E \rightarrow T \\ \Downarrow \\ E \rightarrow T E' \\ E' \rightarrow + T E' \\ E' \rightarrow E \end{array}$$

$$\begin{array}{l} 2, T \rightarrow T F \\ T \rightarrow F \\ T \rightarrow F T' \\ T' \rightarrow F T' \\ T' \rightarrow E \end{array}$$

$$\begin{array}{l} 3, F \rightarrow F^* \\ F \rightarrow a \\ \Downarrow \\ F \rightarrow a F' \\ F' \rightarrow * F' \\ F' \rightarrow E \end{array}$$

$$\begin{array}{l} A \rightarrow A \alpha \Rightarrow A \rightarrow B A' \\ A \rightarrow B \quad A' \rightarrow \alpha A' \\ \alpha \rightarrow e \\ 4, F \rightarrow F^* \\ F \rightarrow b \\ \Downarrow \\ F \rightarrow b F' \\ F' \rightarrow * F' \\ F' \rightarrow E \end{array}$$

$\therefore$  the resultant Grammar with out left Recursion is

RDP :-

$\epsilon \rightarrow T\epsilon'$  $\epsilon' \rightarrow +T\epsilon'$  $\epsilon' \rightarrow \epsilon$  $T \rightarrow FT'$  $T' \rightarrow FT'$  $T' \rightarrow \epsilon$  $F \rightarrow aF'$  $F' \rightarrow *F'$  $F' \rightarrow \epsilon$  $F \rightarrow bF'$ 

→ Construction of Recursive decent Parser :-

Procedure  $\epsilon()$ 

{

 $T()$ ;     $\epsilon'()$ ;

if (Lookahead == \$)

printf ("In string accepted");

else

printf ("In string rejected");

}

Procedure  $\epsilon E'()$ 

{

if (lookahead == '+')

{

match ('+');

 $T()$ ;         $\epsilon'()$ ;

{

else

{

null;

{

Procedure  $T()$ 

{

 $F()$ ;     $T'()$ ;

{

Procedure  $T'()$ 

{

 $F()$ ;

{

if (true)

{

 $T'()$ ;

{

```
else  
    null;  
}  
  
procedure F()  
{  
    if (lookahead == 'a')  
    {  
        match ('a');  
        F'();  
    }  
    else  
    {  
        if (lookahead == 'b')  
        {  
            match ('b');  
            F'();  
        }  
    }  
}
```

```
procedure F'()  
{  
    if (lookahead == '*')  
    {  
        match ('*');  
        F'();  
    }  
    else  
    {  
        null;  
    }  
}
```

```
procedure match (char c)  
{  
    if (lookahead == c)  
    {  
        lookahead++;  
    }  
}
```

6

## LW(1) Parser

\* Introduction.

\* Model of LW(1) Parser.

\* Construction of LW(1) Parser.

### (a) Introduction:-

\* Top-down parser.

\* Non-Recursive parser.

\* Predictive parser.

\* LW(1) means

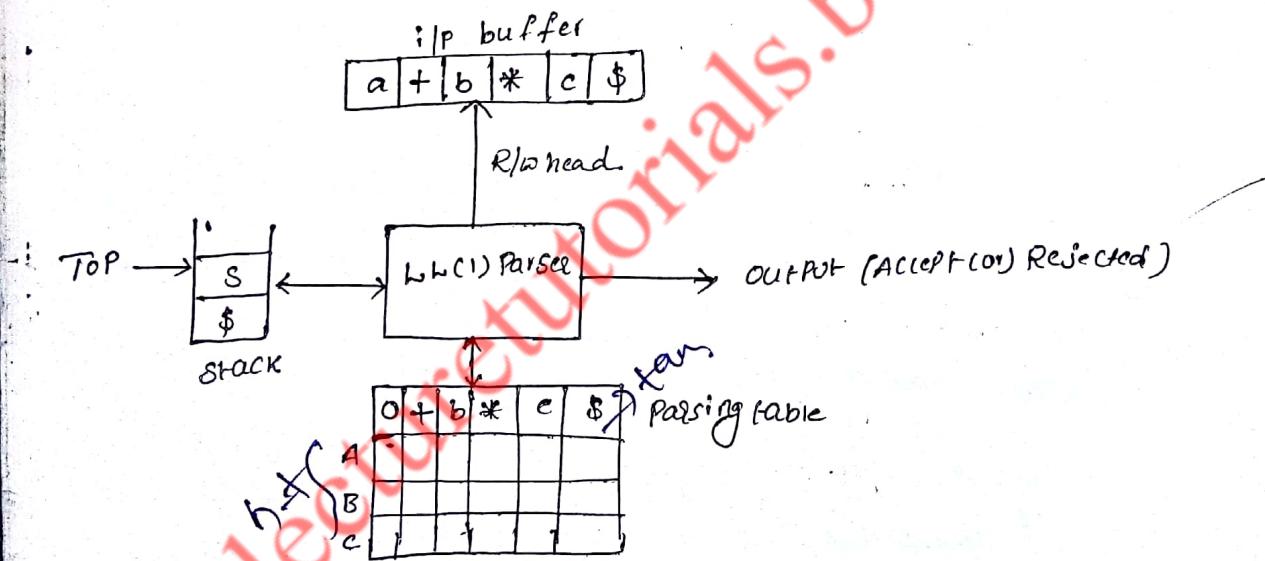
L → Reads the given i/p string from left side to right side.

L → Parse the given i/p string by using "left most derivation".

1 → Reads only one lookahead i/p symbol at a time.

\* It constants a "LW(1) Predictive Parsing table!"

### (b) Model of LW(1) parser:-



It contains three data structures like

1. Input buffer. 2. Stack 3. Parsing table.

Input buffer :- LW(1) Parser uses input buffer to store the i/p tokens.

Stack :- LW(1) Parser uses stack to hold the left sentential form.

i.e., the symbols in the R.H.s<sup>of</sup> rule are placed (pushed) into the stack in reverse order that is from right to left.

Parsing table :- It is a two dimensional array contains called rows and columns. Rows represents non-terminals. Columns represent terminals. The table can be represented by  $M[A][B]$ .

where A is a non terminal.  
a is a current input symbol.

## → Construction of LR(1) Parser :-

Steps:-

1. Computation of FIRST and FOLLOW functions.
2. Construction of LR(1) parser table using FIRST and FOLLOW.
3. Construction of LR(1) Parsing algorithm using parser table.

1. Computation of FIRST and FOLLOW functions:-

1. FIRST function: FIRST is a set of Terminal symbols that are FIRST symbols appearing at RHS of Production rule.

Rule for Computing FIRST function:-

1. If the terminal symbol 'a' then  $\text{FIRST}(a) = \{a\}$
2. If there is a rule ' $X \rightarrow e$ ' then  $\text{FIRST}(X) = \{e\}$
3. for the rule  $A \rightarrow x_1 x_2 x_3 \dots x_n$  then  $\text{FIRST}(A) = \{\text{FIRST}(x_1) \cup \text{FIRST}(x_2) \cup \text{FIRST}(x_3) \cup \dots \cup \text{FIRST}(x_n)\}$ .

Note: LR, PAR.

Ex: Compute FIRST function on the following grammar.  $E \rightarrow E + T$   
 $E \rightarrow T$   
 $T \rightarrow T * F$   
 $T \rightarrow F$   
 $F \rightarrow (E)$   
 $F \rightarrow id$ .

The grammar without left recursion is

$E \rightarrow TE'$   
 $E' \rightarrow +TE'$   
 $E' \rightarrow \epsilon$   
 $T \rightarrow FT'$   
 $T' \rightarrow *FT'$   
 $T' \rightarrow \epsilon$   
 $F \rightarrow (E)$   
 $F \rightarrow id$ .

Production rule	FIRST
$E \rightarrow TE'$	$\text{FIRST}(T) = \{\epsilon, id\}$
$E' \rightarrow +TE'/\epsilon$	$\{\epsilon, +\}$
$T \rightarrow FT'$	$\text{FIRST}(F) = \{c, id\}$
$T' \rightarrow *FT'/\epsilon$	$\{\epsilon, *\}$
$F \rightarrow (E)/id$	$\{c, id\}$

3, compute FIRST function for the following grammar  $S \rightarrow (L)/a$   
 $L \rightarrow L, S/L$

$$\begin{array}{l} S \rightarrow (L)/a \\ L \rightarrow L, S/L \\ \downarrow \\ L \rightarrow SL' \\ L' \rightarrow , SL' \\ L' \rightarrow \epsilon \end{array}$$

The Grammar without left recursion is

$$\begin{array}{l} S \rightarrow (L)/a \\ L \rightarrow SL' \\ L' \rightarrow , SL' \\ L' \rightarrow \epsilon \end{array}$$

Production rule	FIRST
$S \rightarrow (L)/a$	$\{\epsilon, a\}$
$L \rightarrow SL'$	$\text{FIRST}(L) = \{\epsilon, a\}$
$L' \rightarrow , SL'   \epsilon$	$\{\epsilon, \epsilon\}$

3, compute FIRST function on the following grammar.

$$S \rightarrow AaAb | BbBa$$

$$A \rightarrow \epsilon$$

$$B \rightarrow \epsilon$$

The given grammar doesn't contain left recursion.

Production rule	FIRST
$S \rightarrow AaAb   BbBa$	$\{a, b\}$
$A \rightarrow \epsilon$	$\{\epsilon\}$
$B \rightarrow \epsilon$	$\{\epsilon\}$

4, compute FIRST grammar

$$A \quad S \rightarrow aAB \quad (1) \quad B \rightarrow bA \quad (2) \quad \epsilon. \quad (3)$$

$$A \rightarrow aA \quad (1) \quad \epsilon$$

$$B \rightarrow bB \quad (2) \quad \epsilon$$

Production rule	FIRST
$S \rightarrow aAB   bA   \epsilon$	$\{a, b, \epsilon\}$
$A \rightarrow aAb   \epsilon$	$\{a, \epsilon\}$
$B \rightarrow bB   \epsilon$	$\{b, \epsilon\}$

## FOLLOW function:-

FOLLOW(A) is a set of terminal symbols that appear immediately to the right of (A). i.e;

$$\text{FOLLOW}(A) = \{ a \mid s \rightarrow \alpha A \beta \}$$

where  $\alpha, \beta$  is grammar symbols.

a. is terminal symbols

## Rules for Computing FOLLOW:-

1. FOLLOW(S) = { \$ } where S is the start symbol.
2. If there is a Production rule  $A \rightarrow \alpha B \beta$  then  $\in$   
 $\text{FOLLOW}(B) = \text{FIRST}(\beta)$  accept ' $\epsilon$ ' in  $\text{FIRST}(\beta)$ .
3. If there is a Production rule  $\cancel{A \rightarrow \alpha B}$   $\text{FOLLOW}(B) = \text{FIRST}(\beta) \setminus \alpha$
4. If there is a Production rule of the form  $A \rightarrow \alpha B$  then  
 $\text{FOLLOW}(B) = \text{FOLLOW}(A)$ .

Ex:- find follow function on the following grammar.  $S \rightarrow Bb|cd$   
 $B \rightarrow AB|\epsilon$   
 $C \rightarrow c|e$

Sol:-  $\text{FOLLOW}(S) = \{ \$ \}$

$$\text{FOLLOW}(B) = \{ b \}$$

$$\text{FOLLOW}(C) = \{ d \}$$

Ex:- compute FIRST and follow functions on the following grammar.

$$\begin{array}{ll} S \rightarrow ABCDE & C \rightarrow c \\ A \rightarrow a|\epsilon & b \rightarrow d|\epsilon \\ B \rightarrow b|\epsilon & E \rightarrow e|\epsilon \end{array}$$

Y

$$S \rightarrow ABCDE$$

$$\text{FIRST}(S) = \text{FIRST}(A)$$

$$= \{ a \}$$

$$A \rightarrow \underline{a}$$

$$A \rightarrow \underline{\epsilon}$$

$$\text{FIRST}(A) = \{ a \} \cup \{ \epsilon \}$$

$$= \{ a, \epsilon \}$$

$$B \rightarrow \underline{b}$$

$$B \rightarrow \underline{\epsilon}$$

$$\text{FIRST}(B) = \{ b \} \cup \{ \epsilon \}$$

$$= \{ b, \epsilon \}$$

$$\begin{array}{l}
 C \rightarrow c \\
 C \rightarrow \epsilon \\
 \text{FIRST}(C) = \{c\} \cup \{\epsilon\} \\
 = \{c, \epsilon\}
 \end{array}
 \quad
 \begin{array}{l}
 A \rightarrow d \\
 D \rightarrow \epsilon \\
 \text{FIRST}(D) = \{d\} \cup \{\epsilon\} \\
 = \{d, \epsilon\}
 \end{array}$$

$$\begin{array}{l}
 E \rightarrow e \\
 E \rightarrow \epsilon \\
 \text{FIRST}(E) = \{e\} \cup \{\epsilon\} \\
 = \{e, \epsilon\}
 \end{array}$$

$$\begin{array}{l}
 S \rightarrow ABCDE \\
 S \rightarrow ABCDE
 \end{array}
 \quad
 \text{FIRST}(S) = \{a\}$$

$$\begin{array}{lll}
 S \rightarrow ECBDE & S \rightarrow BCDE & S \rightarrow ECDE \\
 S \rightarrow B CDE & S \rightarrow b CDE & S \rightarrow C DE \\
 \text{FIRST}(S) = \text{FIRST}(B) & \text{FIRST}(S) = \{b\} & \text{FIRST}(S) = \text{FIRST}(C) \\
 = \{b, \epsilon\} & & = \{c\}
 \end{array}$$

$$\therefore \text{FIRST}(S) = \{a\} \cup \{b\} \cup \{c\} \\
 = \{a, b, c\}$$

Computation of FOLLOW:

$$\begin{array}{l}
 S \rightarrow ABCDE \\
 A \rightarrow a | \epsilon \\
 B \rightarrow b | \epsilon \\
 C \rightarrow c \\
 D \rightarrow d | \epsilon \\
 E \rightarrow e | \epsilon
 \end{array}$$

$$\text{FOLLOW}(S) = \{\$\}$$

$$\text{FOLLOW}(A) = \{b, c\}$$

$$\text{FOLLOW}(B) = \{c\}$$

$$\text{FOLLOW}(C) =$$

$$S \rightarrow ABCDE$$

$$\text{FOLLOW}(A) = \text{FIRST}(B)$$

$$= \{b, \epsilon\}$$

$$1, S \rightarrow ABCDG$$

$$S \rightarrow AB\underline{c}DE$$

$$\text{FOLLOW}(A) = \{b\}$$

$$2, S \rightarrow A\underline{B}CDE$$

$$S \rightarrow A\underline{E}CDE$$

$$S \rightarrow A\underline{C}DE$$

$$\text{FOLLOW}(A) = \text{FIRST}(C)$$

$$= \{c\}$$

$$S \rightarrow ABCDE$$

$$\text{FOLLOW}(B) = \text{FIRST}(C)$$

$$= \{c\}$$

$$S \rightarrow AB\underline{C}DE$$

$$\text{FOLLOW}(C) = \text{FIRST}(D)$$

$$= \{d, \epsilon\}$$

$$① S \rightarrow ABCDE$$

$$S \rightarrow AB\underline{C}D E$$

$$\text{FOLLOW}(C) = \{d\}$$

$$② S \rightarrow ABCDE$$

$$S \rightarrow ABC\underline{E} E$$

$$S \rightarrow ABCE$$

→ Construction of WLLD Parse Table:

Algorithm:-

Construction for the rule  $A \rightarrow \alpha$  of grammar 'G'.

Step1:- for each ' $\alpha$ ' in FIRST( $\alpha$ ) create entry ' $M[A, \alpha] = A \rightarrow \alpha$ '

where ' $\alpha$ ' is a terminal symbol.

Step2:- for ' $\epsilon$ ' in FIRST( $\alpha$ ) create entry  $M[A, \epsilon] = A \rightarrow \alpha$ .

where ' $\epsilon$ ' is a terminal symbol. In FOLLOW(A).

Step3:- If  $\epsilon$  in FIRST( $\alpha$ ) and  $\$ \in \text{FOLLOW}(A)$  then create entry  
in the table  $M[A, \$] = A \xrightarrow{\epsilon} \alpha$ .

Step4:- All the remaining entries in the table M are made as

syntactic errors.

It means :-

Ex:- construct

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow +TE' \mid e \\ T &\rightarrow FT' \\ T' &\rightarrow *FT' \mid e \\ F &\rightarrow (E) \mid id \end{aligned}$$

$$\rightarrow FIRST(E) = FIRST(T) = \{e, id\}$$

$$FIRST(T) = FIRST(F) = \{c, id\}$$

$$FIRST(E') = \{+, e\}$$

$$FIRST(T') = \{\ast, e\}$$

$$FIRST(F) = \{c, id\}$$

$$FOLLOW(E) = \{\$, \}\}$$

$$FOLLOW(E') = FOLLOW(E) = \{\$, \}\}$$

$$FOLLOW(T) = \{+, \$, \}\}$$

$$FOLLOW(T') = \{+, \$, \}\}$$

$$e \rightarrow TE'$$

$$\begin{aligned} FOLLOW(T) &= FIRST(E') \\ &= \{+, e\} \end{aligned}$$

$$e \rightarrow TE$$

$$E \rightarrow T$$

$$\begin{aligned} FOLLOW(T) &= FOLLOW(E) \\ &= \{\$\}\} \end{aligned}$$

$$\therefore FOLLOW(T) = \{+, \$, \}\}$$

$$e' \rightarrow TE$$

$$E' \rightarrow +T$$

$$\begin{aligned} FOLLOW(T) &= FOLLOW(E') \\ &= \{\$\}\} \end{aligned}$$

$$T' \rightarrow *FT$$

$$T' \rightarrow *F$$

$$\begin{aligned} FOLLOW(F) &= FOLLOW(T') \\ &= \{+, \$, \}\} \end{aligned}$$

WJ

$$\begin{aligned} T^* &\rightarrow *FT' \\ FOLLOW(F) &= FIRST(T') \\ &= \{\ast, e\} \end{aligned}$$

$$T \rightarrow FT'$$

$$\begin{aligned} FOLLOW(F) &= FIRST(T') \\ &= \{\ast, e\} \end{aligned}$$

$$T \rightarrow FT'$$

$$\begin{aligned} FOLLOW(T) &= FOLLOW(T') \\ &= \{+, \$, \}\} \end{aligned}$$

$$T \rightarrow FF$$

$$T \rightarrow F$$

$$\begin{aligned} FOLLOW(F) &= FOLLOW(T) \\ &= \{+, \$, \}\} \end{aligned}$$

$$\therefore FOLLOW(F) = \{+, *, \$, \}\}$$

LAL(1) Parse Table :-

	+	*	(	)	id	\$
E			$E \rightarrow TE'$		$E \rightarrow TE'$	
E'	$E' \rightarrow +TE'$			$E' \rightarrow E$		$e' \rightarrow e$
T			$T \rightarrow FT'$		$T \rightarrow FT'$	
T'	$T' \rightarrow E$	$T' \rightarrow *FT'$		$T' \rightarrow e$		$T' \rightarrow e$
F			$F \rightarrow (E)$		$F \rightarrow id$	

$$\begin{aligned} &\rightarrow E \rightarrow TE' \\ &\quad \Downarrow FIRST(TE') = FIRST(T) \\ &\quad = \{c, id\} \end{aligned}$$

$$\begin{aligned} &E' \rightarrow +TE' \\ &\quad \Downarrow FIRST(+TE') = FIRST(+T) \\ &\quad = \{+\} \end{aligned}$$

$$3, E' \rightarrow E$$

$\Downarrow$

$FIRST(E) = \{\epsilon\}$

$\Rightarrow FOLLOW(E') = \{\$, T\}$

$$4, T \rightarrow FT'$$

$\Downarrow$

$FIRST(FT') = FIRST(F)$   
 $= \{c, id\}$

$$5, T' \rightarrow *FT'$$

$\Downarrow$

$FIRST(*FT') = FIRST(*)$   
 $= \{\*\}$

$$6, T' \rightarrow \epsilon$$

$\Downarrow$

$FIRST(\epsilon) = \{\epsilon\}$

$FOLLOW(T') = \{t, b\}$

7, F  $\rightarrow (E)$

$\Downarrow$

$FIRST((E)) = \{c\}$

$$F \rightarrow id$$

$\Downarrow$

$FIRST(id) = \{id\}$

WCD Parsing algorithm :-

consider the i/p string ~~id+id\*id.~~  $((a), a)$

stack	input string	Action
\$ S	$((a), a)\$$	$S \rightarrow C(L)$
\$) L	$((a), a)\$$	POP
\$) L	$(a), a)\$$	$L \rightarrow SL^1$
\$) L	$(a), a)\$$	$S \rightarrow C(L)$
\$) L	$(a), a)\$$	POP
\$) L	$(a), a)\$$	$L \rightarrow SL^1$
\$) L	$a), a)\$$	$S \rightarrow a$
\$) L	$a), a)\$$	POP
\$) L	$a), a)\$$	$L^1 \rightarrow \epsilon$
\$) L	$a), a)\$$	-
\$) L	$a), a)\$$	POP
\$) L	$a), a)\$$	$L^1 \rightarrow SL^1$
\$) L	$a), a)\$$	POP
\$) L	$a), a)\$$	$S \rightarrow a$
\$) L	$a), a)\$$	POP
\$) L	$a), a)\$$	$L^1 \rightarrow \epsilon$
\$) L	$a), a)\$$	-
\$) L	$a), a)\$$	POP
\$) L	$a), a)\$$	$L^1 \rightarrow \epsilon$
\$) L	$a), a)\$$	-
\$) L	$a), a)\$$	POP
\$) L	$a), a)\$$	accepted

- ① construct LL(1) parser table for the following grammar
- $S \rightarrow (L) / a$
- $L \rightarrow L, S / s$  and check the input string (a), a. is replaced or not by the LL(1) parser.

- ② construct LLL(1) parser table for the following grammar.

$$S \rightarrow PETSS'$$

$$S \rightarrow a$$

$$S' \rightarrow eS / \epsilon$$

$$\epsilon \rightarrow b.$$

### Error Recovery in Predictive Parser:-

→ An error is detected during predictive parsing when the terminal on the top of the stack does not match the next input symbol (or).

→ when non-terminal A on the top of the stack a is the next input symbol and parsing table entry  $E M[A, a]$  is empty.

→ the process of reducing number of errors in the parser table is called error recovery.

→ LLL(1) parser uses "Panic mode" error recovery technique.

### "Panic mode Error recovery"

It is based on the idea of skipping symbols on the i/p until a synchronizing token is selected.

Synchronizing token :- It is a set of terminals obtained from follow <sup>of</sup> non-terminal in the given grammar.

$$\text{ex: } \text{FOLLOW}(E) = \{\$, \$\}$$

$$\text{FOLLOW}(E') = \{\$, \}\}$$

$$\text{FOLLOW}(T) = \{+, \$, \}\}$$

$$\text{FOLLOW}(T') = \{+, \$, \}\}$$

$$\text{FOLLOW}(F) = \{*, +, \$, \}\}$$

After applying panic mode error recovery technique modified  
LALR(0) parse table is.

	+	*	(	)	id	\$
E			$\epsilon \rightarrow T\epsilon'$	sync	$\epsilon \rightarrow T\epsilon'$	sync
$E'$	$\epsilon' \rightarrow \epsilon T\epsilon'$			$\epsilon' \rightarrow \epsilon$		$\epsilon' \rightarrow \epsilon$
T	sync		$T \rightarrow FT'$	sync	$T \rightarrow FT'$	sync
$T'$	$T' \rightarrow E$ , $T' \rightarrow F$			$T' \rightarrow E$		$T' \rightarrow E$
F	sync	sync	$F \rightarrow (E)$	sync	$F \rightarrow id$	sync

### Parsing Algorithm:-

\* If the parser look up the entry M [AA] as a blank then the IIP symbol 'a' is skipped.

\* If the entry is 'sync' then the non-terminal at the top of the stack is popped.

\* If the token on the top of the stack doesn't match the IIP symbol then we pop on the token from the stack.

Ex:- consider the IIP string  $+ id * * id$ .

Stack	IIP String	Action
$\$ E$	$+ id * * id \$$	Skipped +
$\$ E'$	$id * * id \$$	$\epsilon \rightarrow T\epsilon'$
$\$ E' T$	$id * * id \$$	$T \rightarrow FT'$
$\$ E' T F$	$id * * id \$$	$F \rightarrow id$
$\$ E' T id$	$id * * id \$$	Pop
$\$ E' T'$	$* * id \$$	$T' \rightarrow * FT'$
$\$ E' T' F$	$* id \$$	Pop
$\$ E' T' id$	$* id \$$	Sync, Pop
$\$ E' T' F$	$* id \$$	$T' \rightarrow * FT'$
$\$ E' T' id$	$id \$$	Pop
$\$ E' T' id$	$id \$$	$F \rightarrow id$
$\$ E' T' id$	$id \$$	Pop
$\$ E' T' id$	$\$$	$T' \rightarrow \epsilon$ - 11