

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 ^{the S/P} or Syntax analysis is a process which takes string "w" and produce either a parse tree or generates the syntactic 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

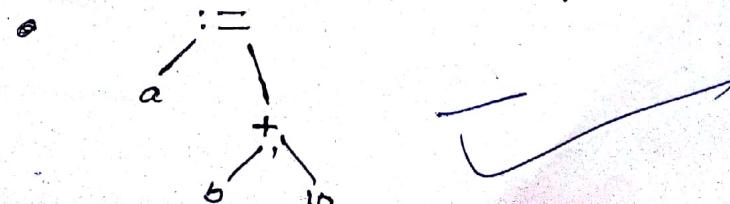
B - Identifier

+ - Assignment Operator

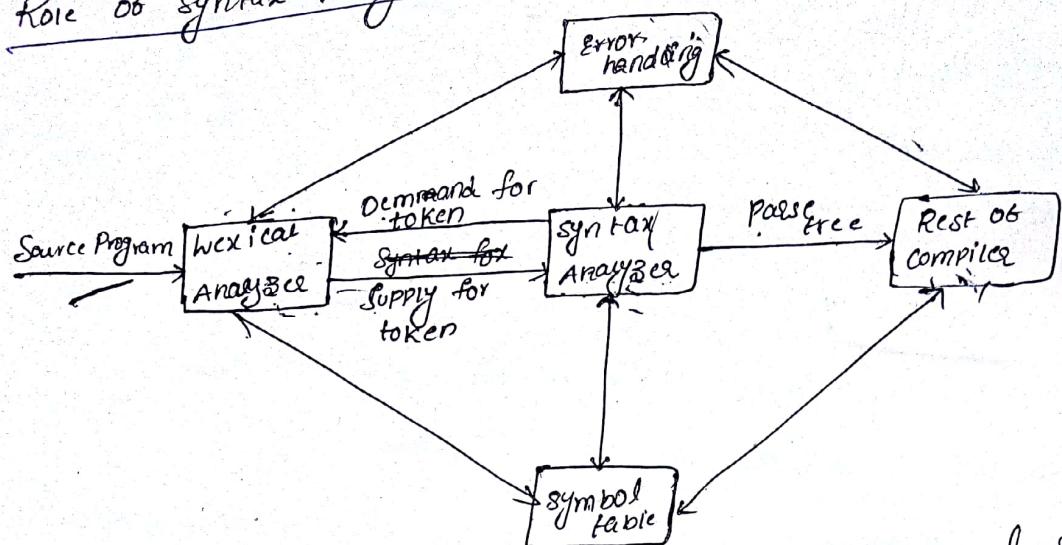
10 - number (or) Constant.

- 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 lexical analyzer work together that means.

- When the parser requires string of tokens it invokes lexical analyzer.
- In turn the lexical analyzer 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.

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

$S \rightarrow$ start symbol.

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

~~$a^n b^n$~~

minimum string = ab

$S \rightarrow \underline{a^n b^n}$

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

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

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 '

from '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 { ϵ , $*$, +, /}.

→ 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
beginning 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
beginning with a start symbol.

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

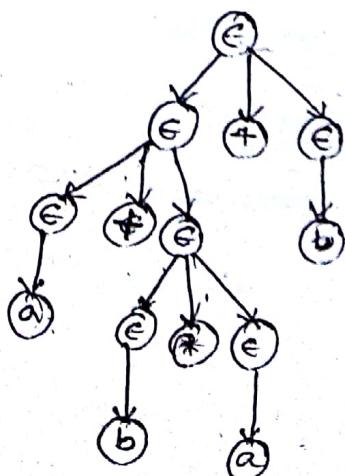
$$\begin{aligned}\epsilon &\rightarrow \epsilon + \epsilon \\ \epsilon &\rightarrow \epsilon - \epsilon \\ \epsilon &\rightarrow \epsilon * \epsilon \\ \epsilon &\rightarrow \epsilon / \epsilon \\ \epsilon &\rightarrow a/b\end{aligned}$$

Obtain 1. Left Most Derivation 2. Right Most Derivation
3. Parse tree. for the input string a+b * a+b.

(a) Left Most Derivation:-

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

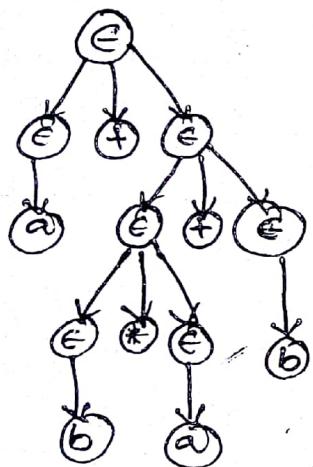
Parse tree:-



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

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

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.

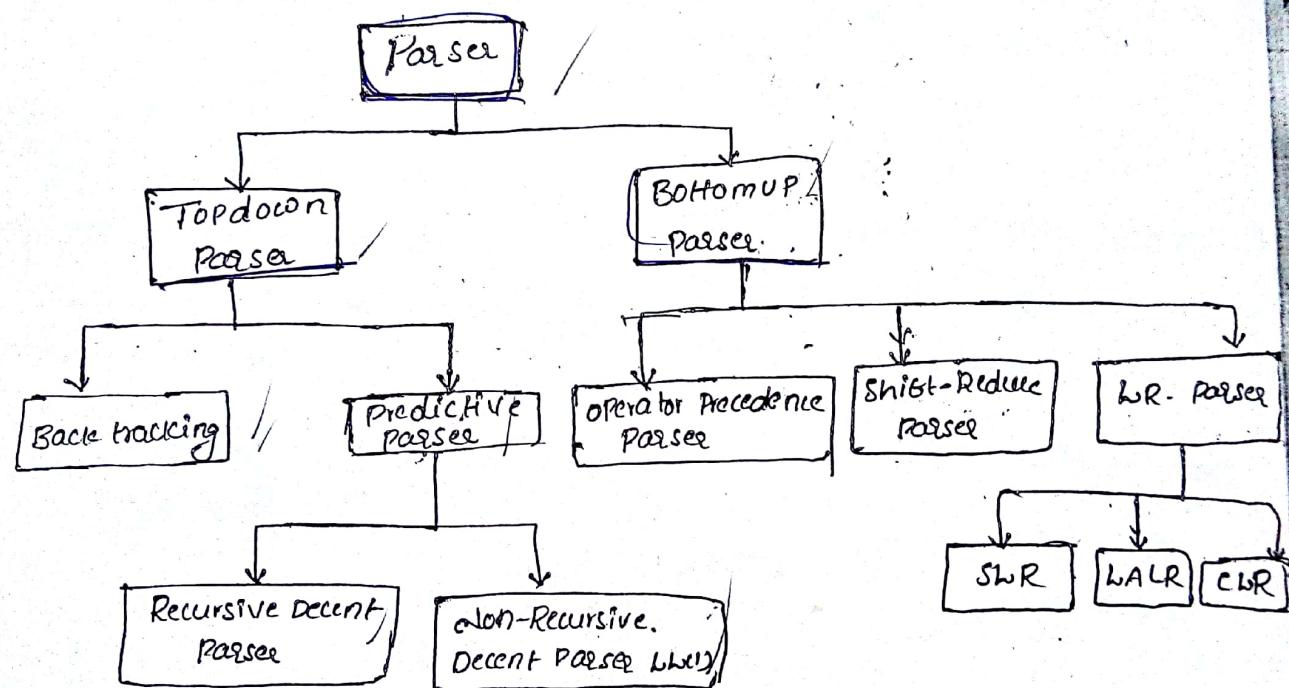
1. Back tracking.

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.

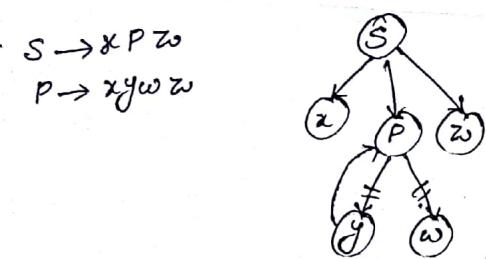
expansion of non terminal symbol we choose one alternative and if some mismatch occurs then we try another alternative if any.

Ex:- consider the grammar $S \rightarrow x P z$
 $P \rightarrow y w y$

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



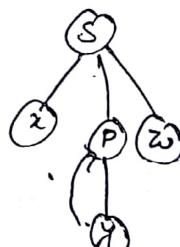
→ 2. also non-terminal symbol 'P' is replaced by the first alternative of it that is $P \rightarrow y w y$ then the parse tree is



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

$$S \rightarrow x P z$$

$$P \rightarrow x y w z$$

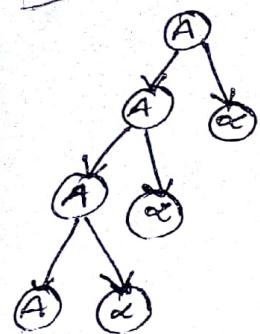


This parsing technique increases lot of overhead in implementation of parse tree. So we need to eliminate the backtracking by modifying the grammar.

→ 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 recursion.

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

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

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

$A = E, \alpha = +T,$

Eliminate left recursion from the given grammar.

$\Rightarrow A^*$
 $A:$ 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 $A = E$
 $\alpha = +T$

$$B = T$$

know we can eliminate left Recursion

$$1. A \rightarrow BA'$$

$$\hookrightarrow E \rightarrow TE'$$

$$3. A' \rightarrow \epsilon$$

$$\hookrightarrow \epsilon' \neq \epsilon$$

$$2. A' \rightarrow \alpha A'$$

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

16

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

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

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

$$\epsilon' \rightarrow \epsilon$$

$$T \rightarrow FT'$$

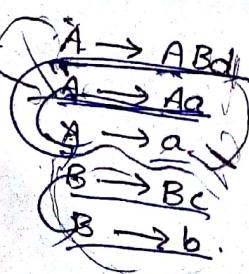
$$T' \rightarrow *FT'$$

$$T \rightarrow \epsilon$$

$$F \rightarrow (\epsilon)$$

$$F \rightarrow id$$

Q) Consider the Grammar



remove left recursion!

The

$$\begin{array}{lll}
 \text{Sol:-} & A \rightarrow ABd & A \rightarrow Aa \\
 & A \rightarrow a' & A \rightarrow a' \\
 & 1. A \rightarrow \alpha A' & 1. A \rightarrow \alpha A' \\
 & 2. A' \rightarrow BdA' & 2. A' \rightarrow \alpha A' \\
 & 3. A' \rightarrow \epsilon & 3. A' \rightarrow \epsilon \\
 & & B \rightarrow Bc \\
 & & B \rightarrow b \\
 & & 1. B \rightarrow bB' \\
 & & 2. B' \rightarrow cB' \\
 & & 3. B' \rightarrow \epsilon
 \end{array}$$

→ Left factoring :-

1. A Grammar may not be suitable for recursive decent Parsing even if there is no left recursion.

2. For example consider the grammar $S \rightarrow iets | ietses | a$

3. A useful method for manipulating the grammar into a form suitable for recursive decent Parsing is left factoring.

* Left factoring :-

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

$$\begin{array}{l}
 1. A \rightarrow \alpha A' \\
 2. A' \rightarrow B_1 | B_2 | B_3 | \dots | B_n
 \end{array}$$

Ex:- consider the Grammar

$$\begin{array}{l}
 S \rightarrow iets | ietses | a \\
 \epsilon \rightarrow b. \text{ do the left factoring on}
 \end{array}$$

above Grammar.

Sol:- consider the Production rule with common Prefix Part.

$S \rightarrow \underline{iets} | \underline{ietses}$ to allow Perform left factoring. Then after the new Productions are α

We can map the Grammar rules with the Rules

$$A \rightarrow \alpha B_1 | \alpha B_2$$

where $A = S$, $\alpha = \underline{iets}$, $B_1 = \epsilon$, $B_2 = \epsilon s$ After left factoring the new Production rules are

$$A \rightarrow \alpha A'$$

$$S \rightarrow iets S'$$

$$\begin{array}{l}
 2. A' \rightarrow B_1 | B_2 \\
 3. \frac{\epsilon S \rightarrow a}{\epsilon \rightarrow b.}
 \end{array}$$

∴ The grammar after left factoring is

$$S \rightarrow i \in \Sigma S'$$

$$S' \rightarrow \epsilon | \epsilon S$$

$$S \rightarrow a$$

$$\epsilon \rightarrow b.$$

Q, To the left factoring in the following Grammar.

$$A \rightarrow aAB | aA | a$$

$$B \rightarrow bB | b.$$

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

$$\text{where } \alpha A = \underline{A}$$

$$\alpha = a$$

$$B_1 = AB$$

$$B_2 = \alpha A$$

$$B_3 = \epsilon$$

After left factoring the new production rule is.

$$A \rightarrow \alpha A' \quad \text{Q, } A' \rightarrow B_1 | B_2 | B_3 \quad \text{3, } bB | b$$

$$A \rightarrow \alpha A' \quad A' \rightarrow AB | \alpha A | \epsilon. \quad \begin{matrix} A = B \\ \alpha = b \\ B_1 = B \\ B_2 = \epsilon. \end{matrix}$$

∴ the grammar after left factoring is.

$$A \rightarrow \alpha A'$$

$$A' \rightarrow AB | \alpha A | \epsilon.$$

$$A \leftrightarrow \alpha$$

$$B \rightarrow b \cdot B'$$

$$B' \rightarrow B | \epsilon.$$

→ Ambiguity:-

A Grammatical which has more than one left most derivation

cor) Right most derivation cor) 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~~. $s \rightarrow s + s$

$$A \rightarrow Ad$$

$$A \rightarrow B.$$

$$s \rightarrow \underline{s}.$$

(1)

1. LMD:-

$$s \rightarrow s + s$$

$$\rightarrow \underline{s} + s$$

$$\rightarrow \underline{s} + \underline{s} + s$$

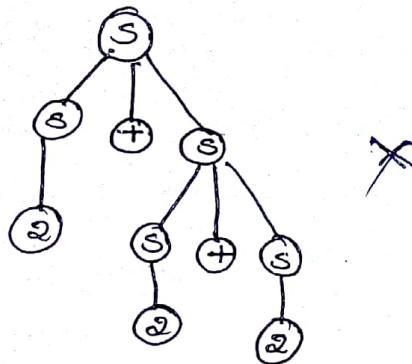
$$\rightarrow 2 + 2 + \underline{S}$$

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

$$A \rightarrow \alpha A$$

$$A \rightarrow B$$

Parse tree :-



✗ ea (doesn't current cal)

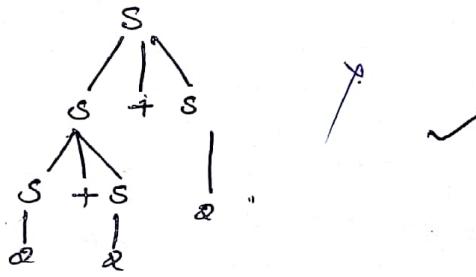
Q, LMD

Draw back :-

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

Q, LMD :-

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



for removing ambiguity

! If the grammar is left associative operators (+, -, *, /, %)

then induce Left Recursion

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

Q, If the grammar is Right associative operator (=, !,)

then induce the Right Recursion.

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

Ex:- Consider the Grammar

$$S \rightarrow S + S$$

$$S \rightarrow S * S$$

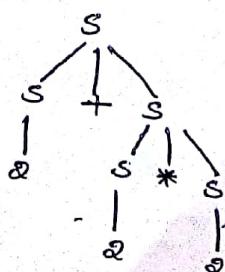
S → 2. And the Sip string is 2 + 2 * 2.

WMD :- S → S + S

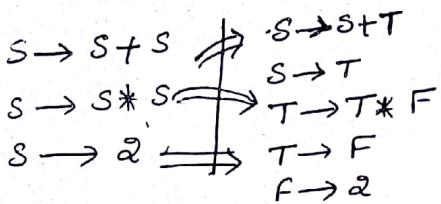
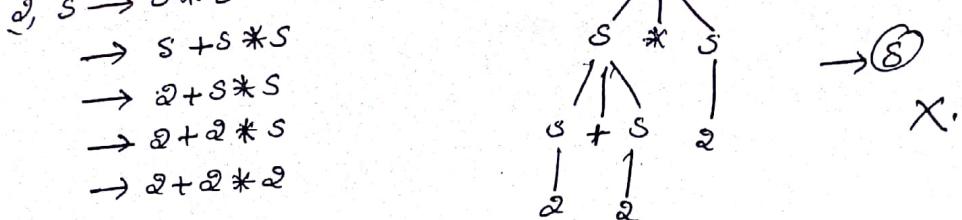
$$\rightarrow 2 + \underline{S}$$

$$\rightarrow 2 + 2 * \underline{S}$$

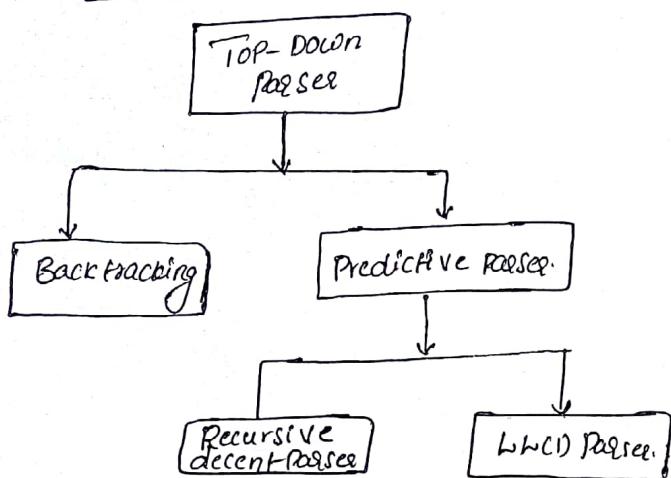
$$\rightarrow 2 + 2 * 2.$$



✓ 8



→ 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 i/p symbol is non-terminal then a call to the procedure corresponding to that non-terminal symbol.
- Step 2 :- If the i/p 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 in to 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.

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

Note :-

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

Sol :- The Given grammar is

$$\begin{array}{l}
 1. E \rightarrow E + T \\
 2. E \rightarrow T \\
 3. T \rightarrow TF \\
 4. T \rightarrow F \\
 5. F \rightarrow F^* / a / b.
 \end{array}$$

The above grammar. Constant left recursion.

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

Elimination of left Recursion :-

$$\begin{array}{l}
 1. E \rightarrow E + T \\
 | \\
 E \rightarrow T \\
 | \\
 E \rightarrow TE' \\
 | \\
 E' \rightarrow +TE' \\
 | \\
 E' \rightarrow E
 \end{array}$$

$$\begin{array}{l}
 2. T \rightarrow TF \\
 | \\
 T \rightarrow F \\
 | \\
 T \rightarrow FT' \\
 | \\
 T' \rightarrow FT' \\
 | \\
 T' \rightarrow E
 \end{array}
 \quad
 \begin{array}{l}
 3. F \rightarrow F^* \\
 | \\
 F \rightarrow a \\
 | \\
 F \rightarrow aF' \\
 | \\
 F' \rightarrow *F' \\
 | \\
 F' \rightarrow E
 \end{array}$$

$$\begin{array}{l}
 A \rightarrow A\alpha \Rightarrow A \rightarrow BA' \\
 | \\
 A \rightarrow B \quad A' \rightarrow \alpha A' \\
 | \\
 \alpha \rightarrow e
 \end{array}
 \quad
 \begin{array}{l}
 4. F \rightarrow F^* \\
 | \\
 F \rightarrow b \\
 | \\
 F \rightarrow bF' \\
 | \\
 F' \rightarrow *F' \\
 | \\
 F' \rightarrow E
 \end{array}$$

\therefore the resultant Grammar with out left Recursion :-

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)

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

else
 ϵ null;
 ϵ

ϵ

procedure F()

ϵ

 if (lookahead == 'a')
 ϵ

 match ('a');
 F'();

ϵ

 else
 ϵ

 if (lookahead == 'b')
 ϵ

 match ('b');
 F'();

ϵ

ϵ

procedure F'()

ϵ

 if (lookahead == '*')
 ϵ

 match ('*');
 F'();

ϵ

 else
 ϵ

 null;
 ϵ

ϵ

procedure match (char c)

ϵ

 if (lookahead == c)
 ϵ

 lookahead++
 ϵ

ϵ

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

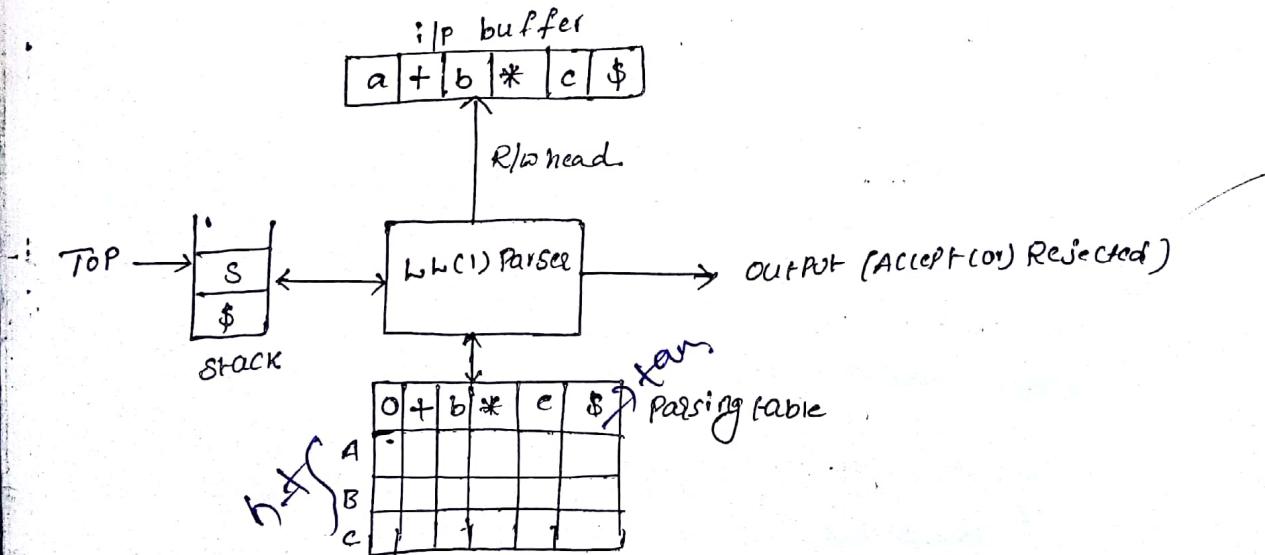
L → Reads the given i/p string from left side to right side.

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^{of} 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 i/p symbol.

Construction of Lw(1) Parser :-

Steps:-

1. computation of FIRST and FOLLOW functions.
2. construction of Lw(1) parser table using FIRST and FOLLOW.
3. construction of Lw(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: Lw. par.

Ex: Compute FIRST function on the following grammar.

$$\begin{aligned} E &\rightarrow E + T \\ E &\rightarrow T \\ T &\rightarrow T * F \\ T &\rightarrow F \\ F &\rightarrow (E) \\ F &\rightarrow id. \end{aligned}$$

The grammar without left recursion is:

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

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

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;

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

where α, B 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 $\beta \in \text{FOLLOW}(B)$.
3. If there is a production rule $A \rightarrow \alpha B \beta$ then $\text{FOLLOW}(B) = \text{FIRST}(\beta) \cup \text{FOLLOW}(A)$ if $\text{FIRST}(\beta)$ contains ϵ .
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. $S \rightarrow Bb/cd$
 $B \rightarrow AB/e$
 $C \rightarrow e/c/e$

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

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

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

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

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

Y

$$S \rightarrow ABCDE$$

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

$$A \rightarrow a = \{ a \}$$

$$A \rightarrow \epsilon$$

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

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

$$B \rightarrow b$$

$$B \rightarrow \epsilon$$

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

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

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

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

$$\begin{array}{l}
 S \rightarrow ABCDE \\
 S \rightarrow ABCDE
 \end{array}$$

$FIRST(S) = \{a\}$

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

$$\begin{aligned}
 \therefore FIRST(S) &= \{a\} \cup \{b\} \cup \{c\} \\
 &= \{a, b, c\}
 \end{aligned}$$

→ Computation of FOLLOW:

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

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

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

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

$$FOLLOW(C) =$$

$$S \rightarrow ABCDE$$

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

$$= \{\epsilon, E\}$$

$$1, S \rightarrow A\underline{BCDE}$$

$$S \rightarrow A\underline{bcde}$$

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

$$2, S \rightarrow A\underline{BCDE}$$

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

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

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

$$= \{\epsilon, C\}$$

$$S \rightarrow ABCDE$$

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

$$= \{C\}$$

$$S \rightarrow AB\underline{CDE}$$

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

$$= \{D, E\}$$

$$① S \rightarrow ABCDE$$

$$S \rightarrow A\underline{B}CD E$$

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

$$② S \rightarrow ABCDE$$

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

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

$$S \rightarrow ABCE$$

→ Construction of NLCI Parse Table:

Algorithm:-

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

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

where ' α ' is a terminal symbol.

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

where ' b ' is a terminal symbol. In FOLLOW(A).

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

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

syntactic errors.

It means if there

Ex:- CONSTRUCT

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

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

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

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

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

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

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

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

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

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

$$e \rightarrow TE'$$

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

$$e \rightarrow TE$$

$$E \rightarrow T$$

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

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

$$e' \rightarrow TE$$

$$e' \rightarrow +T$$

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

$$T' \rightarrow * F e$$

$$T' \rightarrow * F$$

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

WJ

$$T^* \rightarrow * FT'$$

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

$$T \rightarrow FT'$$

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

$$T \rightarrow FT'$$

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

$$T \rightarrow FF$$

$$T \rightarrow F$$

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

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

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			$F \rightarrow (E)$		$F \rightarrow id$	

$$\rightarrow E \rightarrow TE'$$

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

$$E' \rightarrow +TE'$$

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

$$3, E' \rightarrow E$$

\Downarrow

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

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

$$4, T \rightarrow FT$$

\Downarrow

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

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

\Downarrow

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

$$6, T' \rightarrow E$$

\Downarrow

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

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

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	$a), a)\$$	accepted. X

- 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 ^{of} 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'	$\epsilon' \rightarrow T\epsilon'$			$\epsilon' \rightarrow \epsilon$		$\epsilon' \rightarrow E$
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 ablank then the iIP symbol 'a' skipped.

* If the entry is 'sync' then the non-terminal at 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' F	id ** id \$	Pop
\$ E' T' F	** id \$	$T' \rightarrow *FT'$
\$ E' T' F	* id \$	Pop
\$ E' T' F	* id \$	syn, Pop
\$ E' T' F	* id \$	$T' \rightarrow *FT'$
\$ E' T' F	* id \$	Pop
\$ E' T' F	id \$	$F \rightarrow id$
\$ E' T' F	id \$	Pop
\$ E' T' F	\$	$T' \rightarrow E$ - 11

end marker \$.

$LR(1)$ items = $LR(0)$ items + lookahead

How to add lookahead with the production?

Case 1: $A \rightarrow \alpha \cdot BC, \alpha$

Suppose this is old production.

Now it precedes B, so we have to write B's production as well

$B \rightarrow \cdot D$ [1st production]

The look ahead of this production is given as we look at previous productions i.e., old whatever is after B, we find first of that value after that is look ahead of 1st production.

assume $Fist(C) = d$,

$B \rightarrow \cdot D, d$

Case 2: Now if the old production

was like $A \rightarrow \alpha \cdot B, \alpha$

here after B there's nothing

after B so lookahead is

$B \rightarrow \cdot D, \alpha$

Case 3: Assume $A \rightarrow a/b$

$A \rightarrow a, \$$

$A \rightarrow b, \$$

Steps to construct CLR parsing table:

1. Writing augmented grammar.
2. LRD collection of items to be found.
3. Defining 2 functions: goto [list of terminals] and action [list of non-terminals]

CLR Parsing.

Ex: $S \rightarrow CC$ String = dd
 $C \rightarrow CC/d$

Step 1: Write the augmented grammar and LRD items.

$S' \rightarrow S^0$

$S \rightarrow \cdot CC^1$

$C \rightarrow \cdot CC^2$

$C \rightarrow \cdot d^3$

$I_0 \rightarrow S^1 \rightarrow \cdot S, \$$

$S^1 \rightarrow \cdot CC, \$$

$\cdot C \rightarrow \cdot CC, C/d$

$C \rightarrow \cdot d, C/d$

$\text{goto}(I_0, \delta) :$

$I_1 \rightarrow S' \rightarrow S, \emptyset$

$\text{goto}(I_0, C) :$

$I_2 \rightarrow S \rightarrow C.C, \emptyset$

$C \rightarrow \cdot C.C, C/d \emptyset$

$C \rightarrow \cdot d, C/d \emptyset$

$\text{goto}(I_0, C) :$

$I_3 \rightarrow C \rightarrow C.C, C/d$

$C \rightarrow \cdot C.C, C/d$

$C \rightarrow \cdot d, C/d$

$\text{goto}(I_0, d) :$

$I_4 \rightarrow C \rightarrow d, C/d$

$\text{goto}(I_0, C) :$

$I_5 \rightarrow S \rightarrow C.C., \emptyset$

$\text{goto}(I_2, C)$

$C \rightarrow C.C, \emptyset$

$I_6 - \left\{ \begin{array}{l} C \rightarrow \cdot C.C, \emptyset \\ C \rightarrow \cdot C.C, \emptyset \end{array} \right.$

$C \rightarrow \cdot d, \emptyset$

goto (I_2, d):

If: $c \rightarrow d, \$$

goto (I_3, c)

$I_8 \rightarrow c \rightarrow cc., \$ / d$

goto (I_3, c)

$c \rightarrow c.c, \$ / d$

$c \rightarrow cc., \$ / d$

$c \rightarrow d, \$ / d$

goto (I_3, d)

$c \rightarrow d, \$ / d \rightarrow I_4$

goto (I_6, c)

$c \rightarrow cc., \$ \rightarrow I_9$

goto (I_6, c)

$c \rightarrow c.c, \$$

$c \rightarrow cc., \$$

$c \rightarrow d, \$$

goto (I_6, d)

$c \rightarrow d, \$ \rightarrow I_f$

Step 8: Construction of CCR Parse table.

	Terminal			Nonterminals	
	Action			goto	
λ_0	c	d	\$	S	C
I_0	s_3	s_4		I_1	I_2
I_1			Accepted		
I_2	s_6	s_7			I_5
I_3	s_3	s_4			I_8
I_4	g_3	g_3			
I_5				g_1	
I_6	s_6	s_7			I_9
I_7				g_3	
I_8	g_2	g_2			
I_9	g_1	g_2	g_2		

Reduce can be find out by
seeing 'o' operator is present on last
of the grammar.

Stack	Input string	Action
\$ 0	d d \$	Shift 4
\$ 0 d 4	d \$	Reduce $C \rightarrow d$ if we have one symbol on right side then pop 2 symbols.
\$ 0 C 2	d \$	Shift 7
\$ 0 C 2 d 7	\$	Reduce $C \rightarrow d$
\$ 0 C 2 C 5	\$	Reduce $S \rightarrow CC$
\$ 0 S 1	\$	Accepted.

3) SLR Parser:

SLR is simple LR. It is the smallest class of grammar having few numbers of states. SLR is very easy to construct and simpler to LR parsing.

Steps to construct SLR parse table:

1. Writing augmented grammar.
2. $\delta(R)^{(0)}$ Collection of items
3. Find Follow of LHS of production.
4. Defining 2 functions: goto and action.

$E_2: S \rightarrow AS/b$

$A \rightarrow SA/a$

Step 1: Writing writing Augmented Grammar

$S' \rightarrow S$

$S \rightarrow AS$

$S \rightarrow b$

$A \rightarrow SA$

$A \rightarrow a$

Step 2: Calculating Canonical collection
of LR(0) items.

$I_0 \rightarrow S' \rightarrow \cdot S^0$

$S \rightarrow \cdot AS^1$

$S \rightarrow \cdot b^2$

$A \rightarrow \cdot SA^3$

$A \rightarrow \cdot a^4$

goto(I_0, S)

$I_1 \quad S' \rightarrow \cdot S = S' \rightarrow S.$

$A \rightarrow \cdot SA = A \rightarrow S.A$

$A \rightarrow \cdot SA$

$A \rightarrow \cdot a$

(1) present
at first
then it is
final item

$2A \rightarrow \cdot b$

$S \rightarrow \cdot AS$

$S \rightarrow \cdot b$

goto(I_0 , A)

$$S \rightarrow \cdot AS = S \xrightarrow{A} A \cdot S$$

$I_0 - S \rightarrow \cdot AS$

$$S \rightarrow \cdot b$$

$$A \rightarrow \cdot SA$$

$$A \rightarrow \cdot a$$

goto(I_0 , b)

$I_0 - S \rightarrow b \cdot$

goto(I_0 , a)

$I_0 - S \rightarrow a \cdot$

goto(I_1 , A)

$$A \rightarrow SA \cdot$$

$$A \rightarrow S \cdot A$$

$$S \rightarrow A \cdot S$$

$$S \rightarrow \cdot AS$$

$$S \rightarrow \cdot b$$

$$A \rightarrow \cdot SA$$

$$A \rightarrow \cdot a$$

goto(I_1 , S)

$$A \rightarrow \cdot SA$$

$$S \rightarrow \cdot AS$$

$$A \rightarrow \cdot SA$$

$$S \rightarrow \cdot b$$

$$A \rightarrow \cdot a$$

goto(I_1 , a);

$I_4 \rightarrow A \rightarrow a$

goto(I_1 , b)

$I_3 \rightarrow S \rightarrow b$

goto(I_2 , S);

If $S \rightarrow AS$:

$A \rightarrow S \cdot A$

$A \rightarrow \cdot SA$

$A \rightarrow \cdot A$

$S \rightarrow \cdot A S$

$S \rightarrow \cdot b$

goto(I_2 , A)

$I_2 \rightarrow S \rightarrow A \cdot S$

$S \rightarrow \cdot AS$

$S \rightarrow \cdot b$

$A \rightarrow \cdot SA$

$A \rightarrow \cdot a$

goto(I_2 , a)

$A \rightarrow a \cdot - I_4$

goto(I_2 , b)

$A \cdot S \rightarrow b \cdot - I_4$

goto(I₅, S)

$$\begin{array}{l} S \rightarrow AS \\ A \rightarrow S \cdot A \\ A \rightarrow \cdot SA \\ A \rightarrow a \\ S \rightarrow \cdot AS \\ S \rightarrow \cdot b \end{array} \quad \left. \begin{array}{l} \\ \\ \\ \\ \end{array} \right\} I_5$$

goto(I₅, a)

$$A \rightarrow a \cdot \quad I_5$$

goto(I₅, b)

$$S \rightarrow b \cdot \quad I_5$$

goto(I₅, A)

$$\begin{array}{l} S \rightarrow A \cdot S \\ S \rightarrow \cdot AS \\ S \rightarrow \cdot b \\ A \rightarrow \cdot S A \\ A \rightarrow \cdot a \end{array} \quad \left. \begin{array}{l} \\ \\ \\ \end{array} \right\} I_5$$

goto(I₆, A)

$$\begin{array}{l} I_5 \quad A \rightarrow \cdot SA \\ A \rightarrow S A \cdot \quad A \rightarrow \cdot a \\ S \rightarrow A \cdot S \\ S \rightarrow \cdot AS \\ S \rightarrow \cdot b \end{array}$$

goto (I_F , S)

$$\begin{array}{l} A \rightarrow S \cdot A \\ A \rightarrow \cdot S A \\ A \rightarrow a \\ S \rightarrow \cdot A S \\ S \rightarrow b \end{array} \quad \left. \begin{array}{l} \\ \\ \\ \end{array} \right\} I_F$$

goto (I_F , a)

$$A \rightarrow a \cdot \rightarrow I_4$$

goto (I_F , b)

$$S \rightarrow b \cdot \rightarrow I_3$$

goto (I_F , A)

$$\begin{array}{l} A \rightarrow S A \cdot \\ S \rightarrow A \cdot S \end{array} \quad \left. \begin{array}{l} \\ \end{array} \right\} I_5$$

goto (I_F , S)

$$\begin{array}{l} A \rightarrow S \cdot A \\ A \rightarrow \cdot S A \\ A \rightarrow \cdot a \end{array} \quad \left. \begin{array}{l} \\ \\ \end{array} \right\} I_6$$

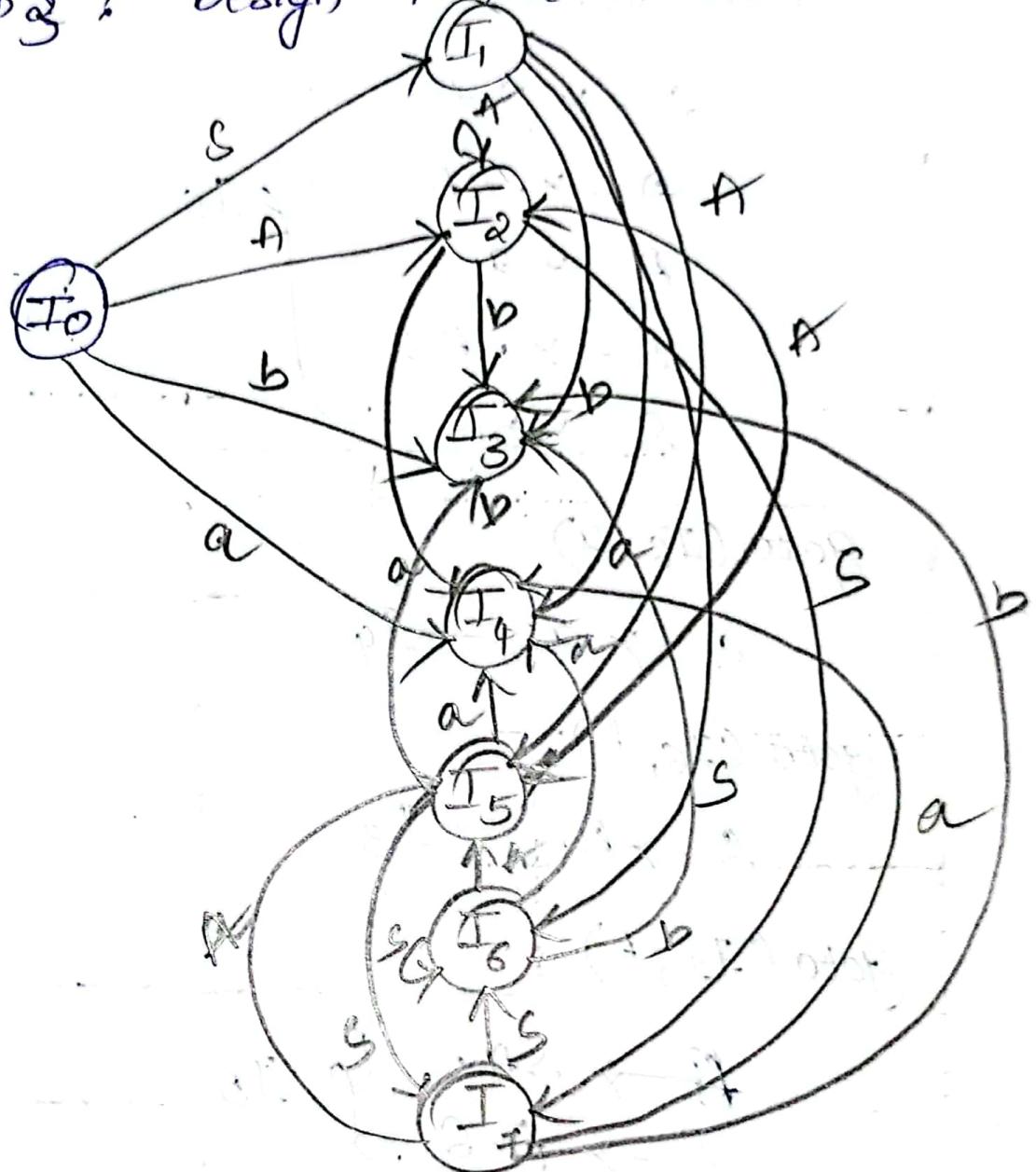
goto (I_F , a)

$$A \rightarrow a \cdot \rightarrow I_4$$

goto (I_F , b)

$$S \rightarrow b \cdot \rightarrow I_3$$

Step 3 : Design Finite Automata



Step 4: $\text{Follow}(S) = \{b, a\}$

$\text{Follow}(A) = \{a, b\}$

$\text{Follow}(S) = \{\$, a, b\}$

$\text{Follow}(A) = \{a, b\}$

Construction of parse table.

	Action			State	
	a	b	\$	AS	AF
I ₀	S ₄	S ₃		I ₁	I ₂
I ₁	S ₄	S ₃	Accepted	I ₅	I ₅
I ₂	S ₄	S ₃		I ₇	I ₂
I ₃	g ₂	g ₂	g ₂		
I ₄	g ₄	g ₄			
I ₅	S ₄ /g ₃	S ₃ /g ₃		I ₇	I ₂
I ₆	S ₄	S ₃		I ₆	I ₅
I ₇	S ₄ /g ₁	S ₃ /g ₁	g ₁	I ₆	I ₅

Step 5: Check whether the string will accept by the parser or not.

String = abab

Stack	Tips	Action
\$ 0	abab\$	Shift 4
\$ 0a\$	bab\$	Reduce A → a
\$. 0A2	bab\$	Shift 3
\$ 0A2b3	ab\$	Reduce $S \rightarrow b$
\$ 0A2S\$	ab\$	Reduce $S \rightarrow AS$
\$ 0S1	ab\$	Shift 4
\$ 0S1a4	b\$	Reduce $A \rightarrow a$
\$ 0S1	b\$	Shift 3
\$ 0S1A5	b\$	Reduce $A \rightarrow SA$
\$ 0A2A2	b\$	Shift 3
\$ 0A2b3	b\$	$S \rightarrow b$
\$ 0A2S\$	\$	Reduce $S \rightarrow AS$
\$ 0S1	\$	Accepted.

↳ LALR Parser

LALR Parser is lookahead LR parser. It is the most powerful parser which can handle large classes of grammar. The size of CLR parsing is quite large as compared to other parsing table. LALR reduces the size of this table. LALR works similar to CLR. It combines the frontier states of CLR parsing table into single state.

LR(1) Item = LR(0) Item + lookahead

Steps for constructing the LALR parsing table:

1. Writing the augmented grammar.
2. LR(1) collection of items to be found.
3. Defining 2 functions i.e. goto and action.

$$Ex: S \rightarrow d = R / R^2$$

$$d \rightarrow * R / id$$

$$R \rightarrow L$$

$$String = id = id$$

Step 1: Augmented Grammar

$$S \rightarrow S$$

$$S \rightarrow \lambda = R / R$$

$$\lambda \rightarrow *R / ?d$$

$$R \rightarrow L$$

Step 2: Calculating LR(0) items.

$$I_0 > S \rightarrow \cdot S, \$ \longrightarrow 0$$

$$S \rightarrow \cdot \lambda = R, \$ \longrightarrow 1$$

$$S \rightarrow \cdot R, \$ \longrightarrow 2$$

$$\lambda \rightarrow \cdot *R, = / \$ \longrightarrow 3$$

$$\lambda \rightarrow \cdot ?d, = / \$ \longrightarrow 4$$

$$R \rightarrow \cdot L, \$ \longrightarrow 5$$

goto(I₀, S)

$$S \rightarrow \lambda \cdot = R, \$$$

$$I_1 > S' \rightarrow S \cdot, \$$$

goto(I₀, L)

$$I_2 > S \rightarrow \lambda \cdot = R, \$$$

$$R \rightarrow L \cdot, \$$$

goto (I_0 , R)

$I_3 \rightarrow S \rightarrow R, \$$

goto (I_0 , *)

$I_4 \rightarrow L \rightarrow *R, =/\$$

$R \rightarrow \cdot L, /\$$

$L \rightarrow \cdot *R, =/\$$

$L \rightarrow \cdot id, =/\$$

goto (I_0 , id)

$I_5 \rightarrow id \rightarrow id, \$$

goto (I_2 , =)

$I_6 \rightarrow S \rightarrow L = \cdot R, \$$

$R \rightarrow \cdot L, \$$

$L \rightarrow \cdot *R, \$$

$L \rightarrow \cdot id, \$$

goto (I_4 , R)

$I_7 \rightarrow L \rightarrow *R, =/\$$

$R \not\rightarrow$

goto (I_4 , L)

$R \rightarrow L, /\$ \rightarrow I_8$

goto (I_4 , *)

$L \rightarrow *R, \$ \rightarrow I_4$

goto(I_4 , id)

$\lambda \rightarrow \text{id.}; \& \rightarrow I_5$

goto(I_6 , R)

$\delta \rightarrow \lambda = R.$, $\& \rightarrow I_9$

goto(I_6 , λ)

$R \rightarrow \lambda.$, $\& \rightarrow I_{10}$

goto(I_6 , $*$)

$\lambda \rightarrow * \cdot R,$ $\&$ } I_{10}

$R \rightarrow \cdot L,$ $\&$

$\lambda \rightarrow \cdot * R,$ $\&$

$\lambda \rightarrow \cdot \text{id},$ $\&$

goto(I_6 , id)

$\lambda \rightarrow \text{id.},$ $\& \rightarrow I_8$

goto(I_{10} , R)

$\lambda \rightarrow * R.$, $\& \rightarrow I_{11}$

goto(I_{11} , λ)

$R \rightarrow \lambda.$, $\& \rightarrow I_{10}$

goto(I_{11} , $*$)

$\lambda \rightarrow * R,$ $\& \rightarrow I_{11}$

goto (I_{11} , id)

$\alpha \rightarrow id, \$ \rightarrow T_{12}$

Combine the 8 states grammar

$I_4 \in I_{11} \rightarrow I_{411}$

$I_5 \in I_{12} \rightarrow I_{512}$

$I_7 \in I_{13} \rightarrow I_{713}$

$I_8 \in I_0 \rightarrow I_{810}$

Step 3: Construct LR parse table

	Action	*	\$	S, d	R	goto
0	S_{512}	S_{411}		1	2	3
1			Accepted			
2	S_6		g_5			
3			g_2			
411	S_{512}	S_{411}			810	T_{13}
512		g_4	g_4			
6	S_{512}	411			810	9
713		g_3	g_3			
810		g_5	g_5			
9			g_1			

Stack

\$ 0

Top Storing

id = Id \$

Action

Shift 512

\$ 0 @ Id 512

!= Id \$

Shift Reduce
d → Id

\$ 0 R d 512

= Id \$

Shift 6

\$ 0 L 2 = 6

Id \$

Shift 512

\$ 0 L 2 = 6 Id 512

\$

Reduce 94

C → ;

\$ 0 L 2 = 6 (8) 2

\$

Reduce 95

\$ 0 L 2 = 6 R 9

\$

Reduce 96
S → L = R

\$ 0 S 1

\$

Accepted.

Ex: S → CC

C → aC / b

Storing % bb

Ex: E → E + T / T

T → T * F / F

Storing % id \$ id \$ id

F → (E) / Id



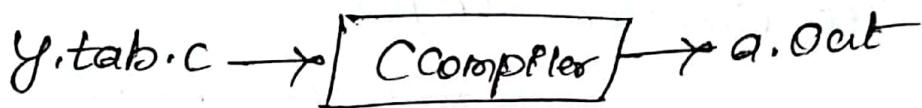
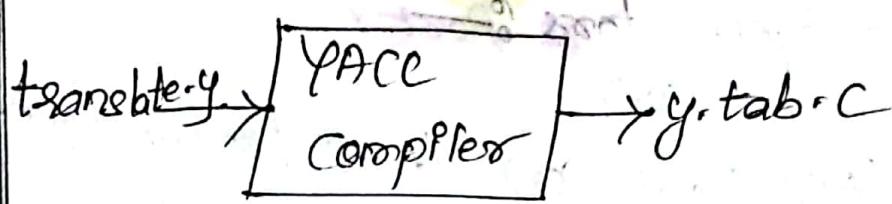
VI) Parser Generators :-

(or)

YACC Tool (or) YACC Parser Generators
(Yet Another Compiler Compiler)

1. YACC provides a tool to produce a parser for a given grammar.
2. YACC is a program designed to compile a LR(0) grammar.
3. It is used to produce the source code of the syntactic analyzer of the language produced by LR(0) grammar.
4. The Input of YACC is the rule (or) grammar and the output is a program.

It is a tool to produce a parser. It accepts input from lexical analyzer(tokens) and produce parsers. It is also called as parser generators.



Structure of YACC tool.

YACC program generally contains

3 sections.

1. declarations
2. translation rules
3. Auxiliary functions.

The Declaration Part:

There are two sections in the declarations part of YACC programs;
in the first section, we put ordinary C declarations, delimited by % and %., and in second section contains only include statement that causes the C preprocessor to include the standard header file <ctype.h>.

The translation Rules Part:

In this part the YACC Specification after the first %Y part, we put the translation rules. Each rules contains of a grammar production and the associated semantic actions.

$\langle \text{heads} \rangle \rightarrow \langle \text{body}_1 \rangle / \langle \text{body}_2 \rangle / \dots / \langle \text{body}_n \rangle$

would be written as in YACC

$\langle \text{heads} \rangle : \langle \text{body}_1 \rangle \{ \langle \text{semantic action} \rangle \}$

$/ \langle \text{body}_2 \rangle \{ \langle \text{semantic action} \rangle \}$

⋮

$/ \langle \text{body}_n \rangle \{ \langle \text{semantic action} \rangle \}$

Ex: $E \rightarrow E + T / T$

expr: expr '+' term { \$1 = \$1 + \$3; }

| term

⋮

The Auxiliary functions :

It is mainly used to define C functions `ypylex()` will be written in auxiliary functions.

YACC Program of Simple Desk Calculator

$E \rightarrow E + T / T$

$T \rightarrow T * F / F$

$F \rightarrow (E) / id$

→ translate.y

1. #

~~#include <ctype.h>~~ — ctype stands
for conversion.

2. #

3. token DIGIT

4. %

line : expr 'n' { printf("%d\n", \$1) }

expr : expr '+' term { \$\$ = \$\$ + \$3 }

| term ;

term : term '*' factor { \$\$ = \$\$ * \$3 }

| factor ;

factor : '(' expr ')' { \$\$ = \$2 }

| DIGIT ;

5. % — lexical analyzer

yylex()

{ int c ;

c = getchar();

if (c.isdigit())

{
 yylval = c - '0';

getchar();

}