

# **Compiler Design**

**GATE CSE NOTES**



-7085148007 ·

NAME: Rakesh Nama. STD.: \_\_\_\_\_ SEC.: \_\_\_\_\_ ROLL NO.: \_\_\_\_\_ SUB: Compiler Design.

# INTRODUCTION

## Compiler Design

→ ([www.gatenotes.in](http://www.gatenotes.in))

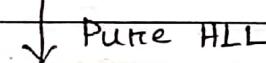
- Introduction of and various phases of compiler Design :-

→ The main aim of compiler design is to convert a pure high level language into low level language.

HLL (high level language)

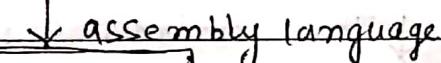


[pre-processor]



Pure HLL

[Compiler]



assembly language

[Assembler]



m/c code (relocatable)

[loader/linker]



[executable code/  
absolute m/c code]

{`#include`} - file inclusion

{`#define`} - macro expansion

(If any language contain this type of line, it called HLL)

→ Pure HLL means program not contain any '#' line.

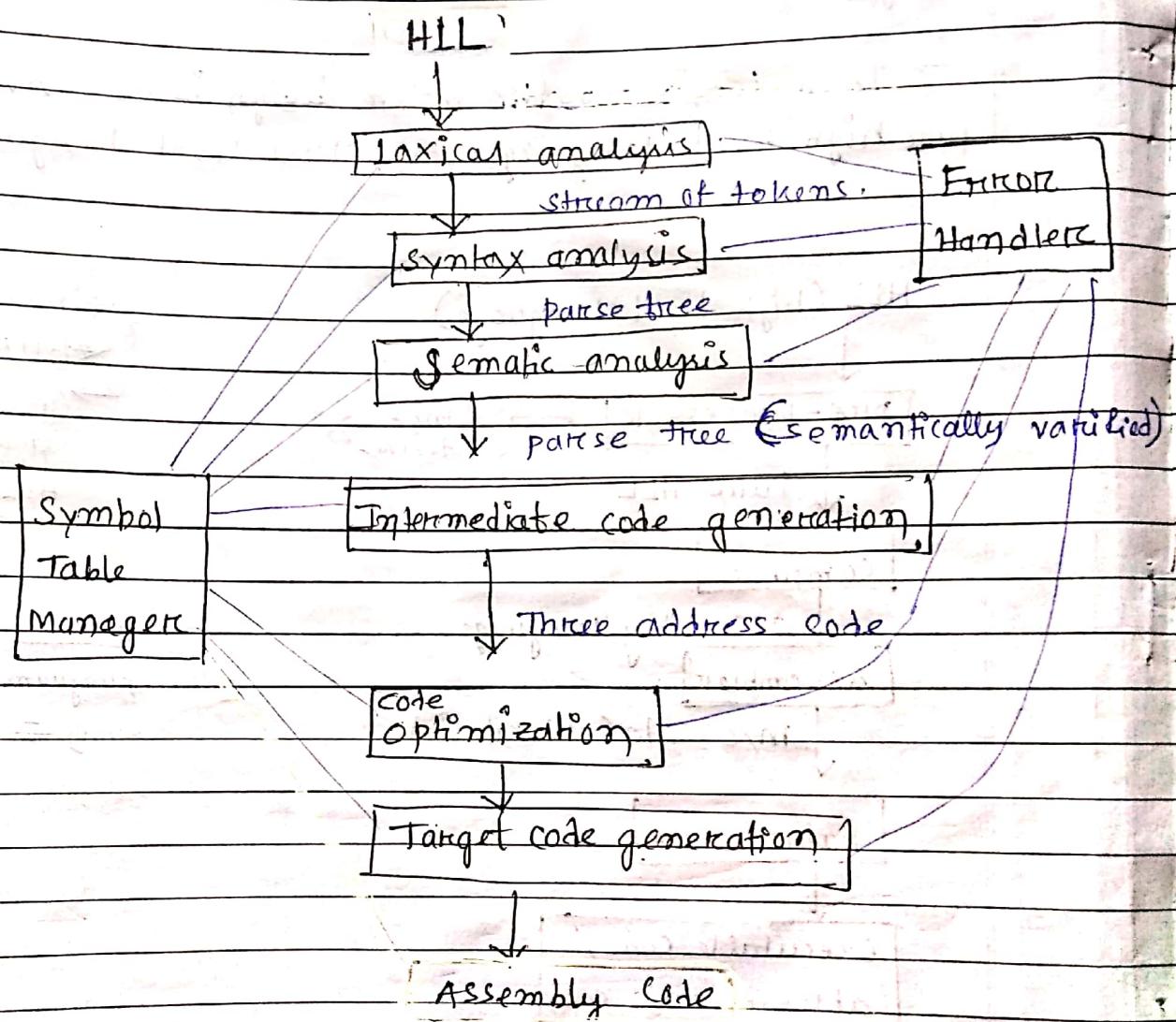
→ preprocessor : is responsible for

1. Macro expansion
2. File inclusion

→ loader : is responsible for

1. Allocation
2. Re-allocation
3. Linking
4. Loading

## Compiler phases



Lex and yacc: are tools used in unix operating system for compiler design. first compiler is FORTRAN. It took 18 years to build it.

### Lexical analysis:

HLL text or source text is broken into tokens.

ex: If (A>B) → 10 tokens.  
 $a = 10;$

$$a = 10;$$

Example of all the phases of compiler —

$x = a + b * c ; \rightarrow$  source program.



[Lexical analyser]



$id = id + id * id \rightarrow (id = identifier)$



heart of the compiler

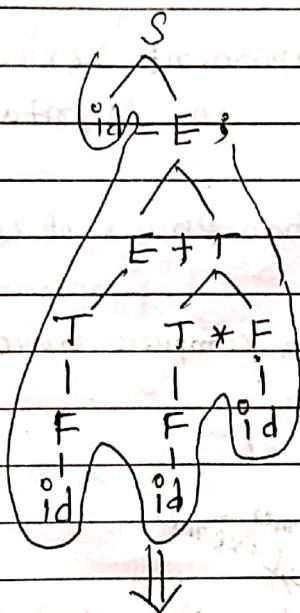
[Syntax analyser]

$S \rightarrow id = E ;$  (CFGc)

$E \rightarrow E + T / T$

$T \rightarrow T * F / F$

$F \rightarrow id$



[Semantic analyser]

(parse tree semantically verified)

[ICGc]



$t_1 = b * c$

$t_2 = a + t_1$

$n = t_2$



[Code optimization]

$t_1 = b * c$

$n = a + t_1$

↓  
 Target code generation]

↓  
 { mul R<sub>1</sub>; R<sub>2</sub>      a → R<sub>0</sub>  
   add R<sub>0</sub>; R<sub>2</sub>      b → R<sub>1</sub>  
   mov R<sub>2</sub>, x      c → R<sub>2</sub>

Assembly code.      ✓ Backend.

→ Lex (tool) used to implement 'lexical analysis'.

→ yacc (tool) " " " Syntax analyser".

→ Practically we have two phases of compiler — frontend and Backend.

→ To do any project on compiler have tool called 'LANCE'.

Lexical analyser:

```

    int max(x,y)
    int x,y;
    /* find max of x and y */
    {
        return (x>y ? x:y);
    }
  
```

Lexemes = tokens

→ Lexical analyser removing the comments and all white spaces.

→ main function of lexical analyser is converting Lexemes into token.

→ Show the errors. (If getting any error).

**Ex:** How many token is there in this particular lines =

① `int/ max/(/x,y/)/  
int/ y/;/  
/* find max of x and y */  
{/  
return/(y>x)?y:x/;/  
}/  
→ 25 (tokens)`

② `printf(" %d Hai ", &w);`

→ 8 (tokens are there)

→ Question can come from Lexical analyser is  
(How many token is there and what are the  
responsibility)

• Grammars:

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

$V \rightarrow$  variable

$T \rightarrow$  Terminal

$P \rightarrow$  Production

$S \rightarrow$  start symbol.

input

②

Syntax analysis

①

Grammar

**Ex:**  $E \rightarrow E + E$  here,

$/E * E/$

$/id/$

$$V = \{E\}$$

$$T = \{+, *, id\}$$

start symbol = E.

$(id + id * id)$  generate this string using given rule (LMD)  
Left most derivation = LMD    Right most derivation = RMD

$$E \Rightarrow E + E$$

$$\Rightarrow id + E$$

$$\Rightarrow id + E * E$$

$$\Rightarrow id + id * E$$

$$\Rightarrow id + id * id$$

$$E \Rightarrow E + E$$

$$\Rightarrow E + E * E$$

$$\Rightarrow E + E * id$$

$$\Rightarrow E + id * id$$

$$\Rightarrow id + id * id$$

LMD =

RMD =

$$E \Rightarrow E * E$$

$$\Rightarrow E + E * E$$

$$\Rightarrow id + E * E$$

$$\Rightarrow id + id * E$$

$$\Rightarrow id + id * id$$

$$E \Rightarrow E * E$$

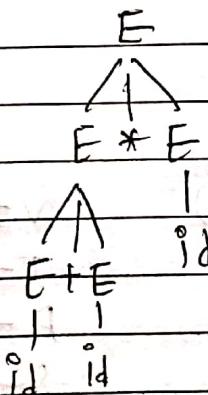
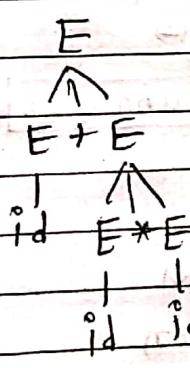
$$\Rightarrow E * id$$

$$\Rightarrow E + E * id$$

$$\Rightarrow E + id * id$$

$$\Rightarrow id + id * id$$

• Derived using parse tree



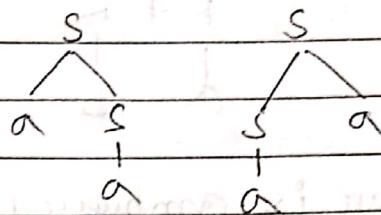
→ To the given grammar and string if we get more than one LMD or RMD and parse tree then the grammar is called ambiguous grammar.

→ ambiguity problem is undecidable.  
(because no algo to solve the problem)

• Find given grammars are ambiguous or not:

$$\text{① } S \rightarrow aS / Sa / a$$

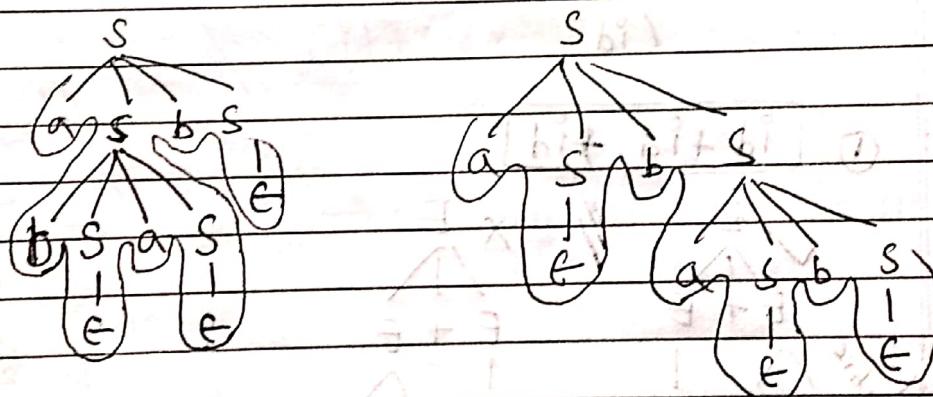
$$\rightarrow w = aa$$



→ This grammar is ambiguous.

$$\text{② } S \rightarrow aSbS / bSaS / e$$

$$\rightarrow w = abab$$



(abab)

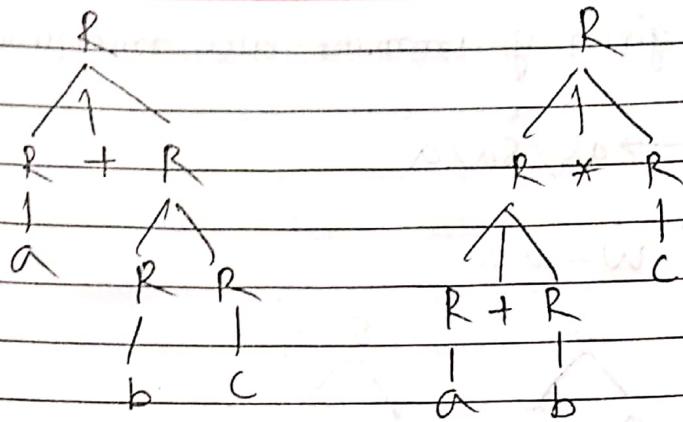
(abab)

→ To the same string given grammar and string we get more than one parse tree, so that the given grammar is ambiguous.

③

$$R \rightarrow R + R / RR / R^* / a / b / c$$

$$\rightarrow w = a + bc$$



→ This grammar is ambiguous grammar.

- Ambiguous grammar and making them unambiguous.

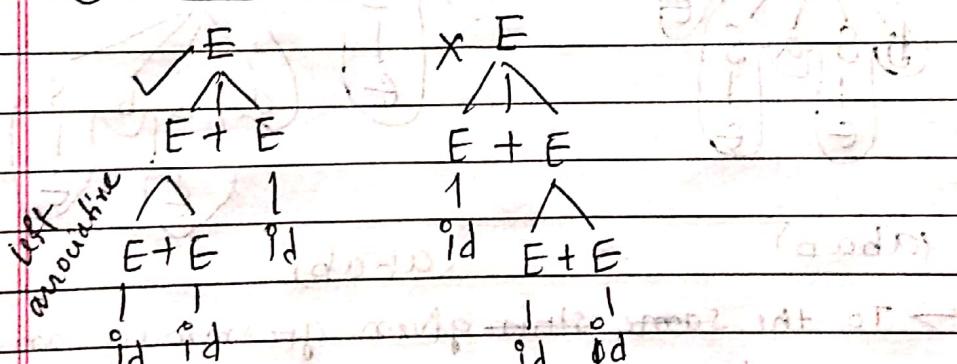
(Ex-1)

$$E \rightarrow E+E$$

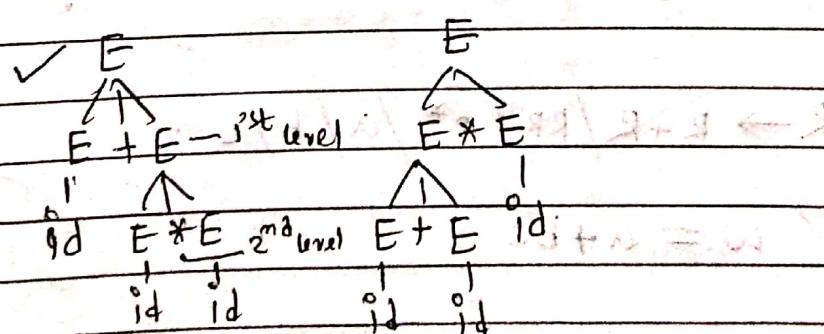
$$/ E * E$$

$$/ id$$

①  $id + id \neq id$



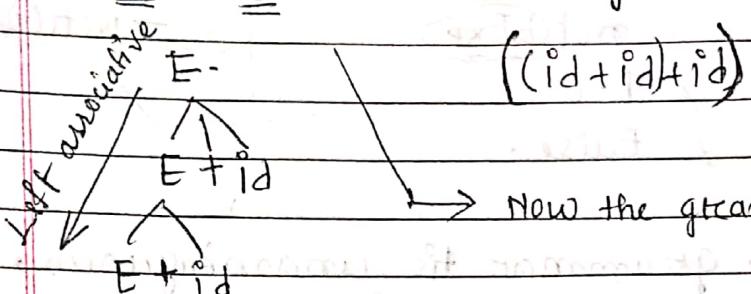
②  $id + id * id$



~~Left recursive are left associative.~~

~~De-erasing or~~

①  $E \rightarrow E + id / id$  (This grammar is left recursive)

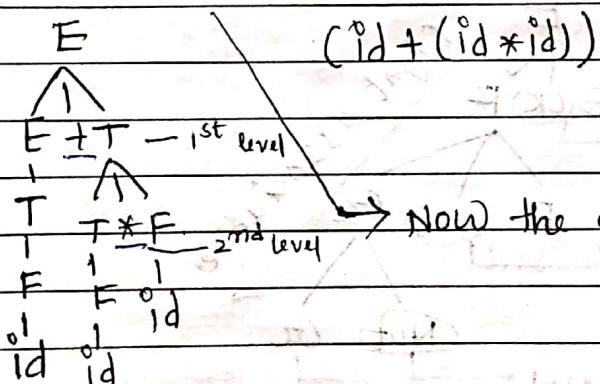


Now the grammar is unambiguous.

②  $E \rightarrow E + T / T$

$T \rightarrow T * F / F$

$F \rightarrow id$



(id + (id \* id))

Now the grammar is unambiguous.

$$(2^{(3^2)}) = 2 \uparrow 3 \uparrow 2$$

precedence:  $\uparrow > \uparrow > + . : \quad$   
 3rd level    2nd level    1st level

③  $E \rightarrow E + T / T$

hence '+' and '\*' is left associative,

$T \rightarrow T * F / F$

$F \rightarrow G \uparrow F / G$

$G \rightarrow id$

$\uparrow$  is right associative.

(4) boolean expression,

convert

$$(b\text{Exp}) \rightarrow (b\text{Exp}) \text{ OR } b\text{Exp}$$

$$/ (b\text{Exp}) \text{ and } b\text{Exp}$$

$$/ \text{ not } (b\text{Exp})$$

$$/ \text{ True}$$

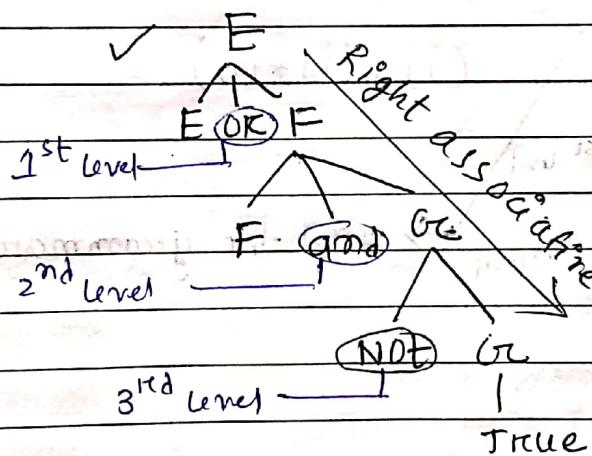
$$/ \text{ False}$$

$$(E) \rightarrow (E) \text{ OR } F / F$$

$$(F) \rightarrow (F) \text{ and } G / G$$

$$(G) \rightarrow \text{Not}(G) / \text{True} / \text{False.}$$

- This grammar is unambiguous, because lower precedence operators closest to the start symbol (1<sup>st</sup> level) and higher precedence operators are least level.
- And this grammar follows the Associativity rule.



(5)  $R \rightarrow R + R$  (convert into Unambiguous grammar)

/ RR

/ R\*

/ a

/ b

/ c

$$E \rightarrow E + T / T$$

$$T \rightarrow T * F / F$$

$$F \rightarrow F * / a / b / c$$

↓ Now this grammar is unambiguous grammar.

⑥ Given grammar -

$$(A) \rightarrow (A) \$B/B \quad (\$, \#, @ \text{ are operators})$$

$$(B) \rightarrow (B) \# C/C$$

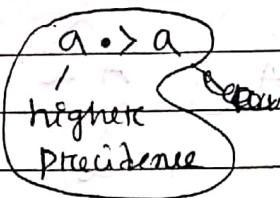
$$(C) \rightarrow @ @ D/D \quad (\text{here, \$, \#, @ are left recursive})$$

$$D \rightarrow d$$

$$\rightarrow \$ \Rightarrow \$$$

$$\# \Rightarrow \#$$

$$@ \Rightarrow @$$



$$\rightarrow \$ < . \# < . @ \quad \text{Operators are evaluated}$$

⑦ Given grammar -

$$E \rightarrow (E) * F \rightarrow \text{here, } * \text{ is left associative. (left recur)}$$

$$/ F + (E) \rightarrow + \text{ is right associative. (right recur)}$$

$$/ F$$

$$F \rightarrow (E) - (E) \rightarrow \text{Associativity is undefined.}$$

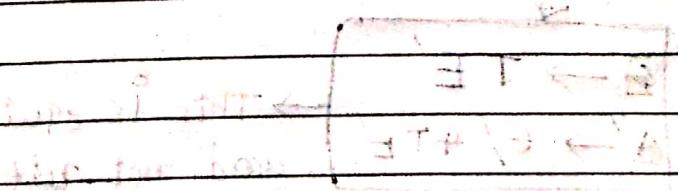
because, here, '-' define left or right associative both.

$\rightarrow$  And This grammar is ambiguous.

$$* \stackrel{\text{def}}{=} + \text{ (elucay).}$$

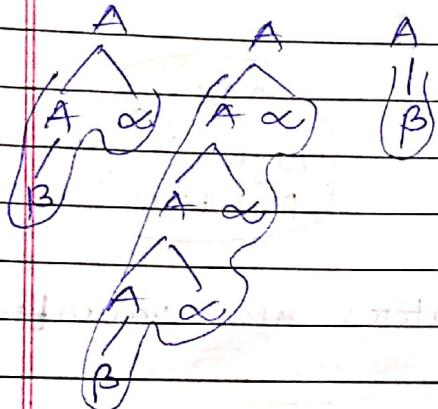
$$(\text{high precedence}) * \Rightarrow *$$

$$+ < . + (\text{high precedence})$$



Recursionleft  
recursion

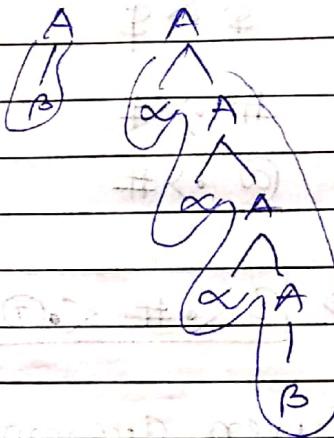
$$(A) \rightarrow A\alpha / B$$



$$L = (\beta\alpha^*)$$

Right  
recursion

$$A \rightarrow \alpha A / B$$



$$L = (\alpha^* B)$$

- $B\alpha^*$

$$A \rightarrow B\alpha^*$$

$$\begin{array}{|c|} \hline A \rightarrow BA' \\ \hline \end{array}$$

$$\begin{array}{|c|} \hline A' \rightarrow \epsilon / \alpha A' \\ \hline \end{array} \Leftrightarrow [A \rightarrow A\alpha / B]$$

$\rightarrow$  To eliminate left recursion we can follow this above rule.

[example] - eliminate the left recursive,

$$\begin{array}{l} E \rightarrow E + T / T \\ (\bar{A}) \Downarrow (\bar{A}) (\bar{\alpha}) (\bar{\beta}) \end{array}$$

$$\rightarrow \begin{array}{l} E \rightarrow TE' \\ E' \rightarrow \epsilon / +TE' \end{array}$$

This is equivalent grammar and not left recursive grammar - atc.

[Example] - Eliminate left recursion from this grammar -

$$\textcircled{1} \quad S \rightarrow \underline{\underline{S_0 S_1 S}} / \underline{\underline{01}}$$

(A)                   (A) (α)                   (B)

$$\Rightarrow \boxed{S \rightarrow \underline{01} S'}$$
  

$$\boxed{S' \rightarrow e / \underline{\underline{0 S_1 S'}}$$

$$\textcircled{2} \quad S \rightarrow (L) / \alpha .$$

$$L \rightarrow \underline{\underline{L_1 S}} / \underline{\underline{S}}$$

(A)                   (A) (α)                   (B)

$$\Rightarrow \boxed{S \rightarrow (L) / \alpha .}$$

$$L \rightarrow \underline{S L'}$$
  

$$\boxed{L' \rightarrow e / , S L'}$$

$$\textcircled{3} \quad A \rightarrow \underline{\underline{A \alpha_1}} / \underline{\underline{A \alpha_2}} / \underline{\underline{A \alpha_3}} / \dots$$
  

$$| \beta_1 | \beta_2 | \beta_3 | \dots$$

$$\Rightarrow \boxed{A \rightarrow \beta_1 A' / \beta_2 A' / \beta_3 A' / \dots}$$
  

$$\boxed{A' \rightarrow \alpha_1 A' / \alpha_2 A' / \alpha_3 A' / \dots}$$

Gi-grammatic



Ambiguous



unambiguous

Gi-grammatic



Left recursive

Right recursive

Gi-grammatic



Deterministic

Non-Deterministic

• Non-Deterministic :

$$A \rightarrow \alpha \beta_1 / \alpha \beta_2 / \alpha \beta_3$$

$\alpha \beta_3$

Left factoring

factoring

$\begin{array}{c} A \\ \nearrow \alpha \\ \beta_1 \end{array}$   
(Back tracking)

$\begin{array}{c} A \\ \nearrow \alpha \\ \beta_2 \end{array}$   
(Back tracking)

$\begin{array}{c} A \\ \nearrow \alpha \\ \beta_3 \end{array}$  (accepted)

• Left factoring procedure or eliminating non-determinism :

$$A \rightarrow \alpha A'$$

$$A' \rightarrow \beta_1 / \beta_2 / \beta_3$$

generate  $\alpha \beta_3$

$\begin{array}{c} A \\ \nearrow \alpha \\ A' \\ \nearrow \beta_3 \end{array}$

$\alpha \beta_3$

Example -

$$(i) S \rightarrow i E t S$$

i E t s e s  $\rightarrow$  (Non-deterministic)

/ a

$$E \rightarrow b$$

$$(ii) S \rightarrow i E t S \quad S' / a$$

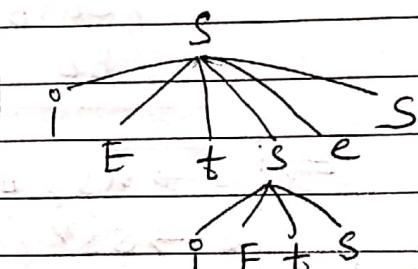
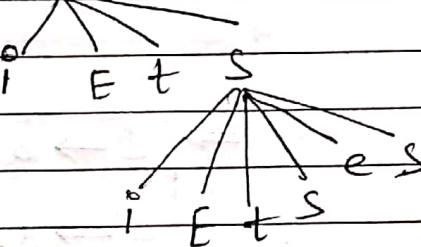
$$S' \rightarrow e / es$$

$E \rightarrow b \rightarrow$  (Deterministic)

(i)

S

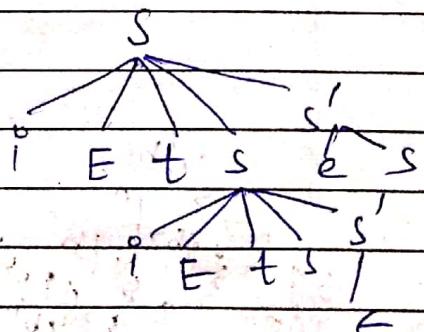
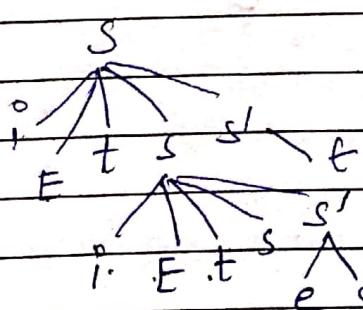
( $w = i E t i E t s e s$ )



(Ambiguous grammar)

$\rightarrow$  for the string we get '2' parse tree from the given grammar.

$$(ii) w = i E t i E t s e s$$



(also ambiguous grammar)

\*\* Eliminating non-determinism or left-factoring doesn't eliminate ambiguity.

[Example]

(Non-deterministic)

$$S \rightarrow @ \underline{as} b S$$

$$/ @ S a s b$$

$$/ @ b b$$

$$/ b$$

Non-deterministic

$$S \rightarrow \underline{b} \underline{as} a a S$$

$$/ \underline{b} \underline{S} S a S b$$

$$/ \underline{b} \underline{S} b$$

$$/ a$$

$\Rightarrow$

$$S \rightarrow a S' / b$$

$$S' \rightarrow \underline{S} b S' /$$

$$/ \underline{S} a S b$$

$$/ b b$$

$\Rightarrow$

$$S \rightarrow b S S' / a$$

$$S' \rightarrow \underline{S} a S / \underline{S} a S b / b$$

$\Rightarrow$

$$S \rightarrow a S' / b$$

$$S' \rightarrow S S'' / b b$$

$$S'' \rightarrow S b S / a S b$$

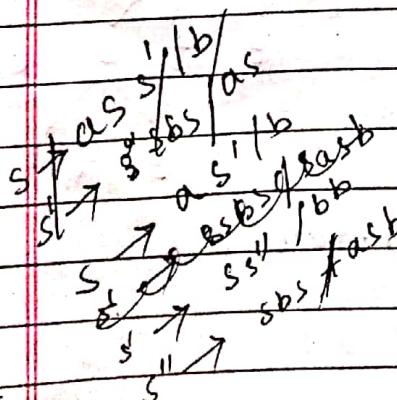
(Deterministic)

$$\Rightarrow S \rightarrow b S S' / a$$

$$S' \rightarrow S a S'' / b$$

$$S'' \rightarrow a S / S b$$

(Deterministic)



# PARSERS

(www.gatenotes.in)

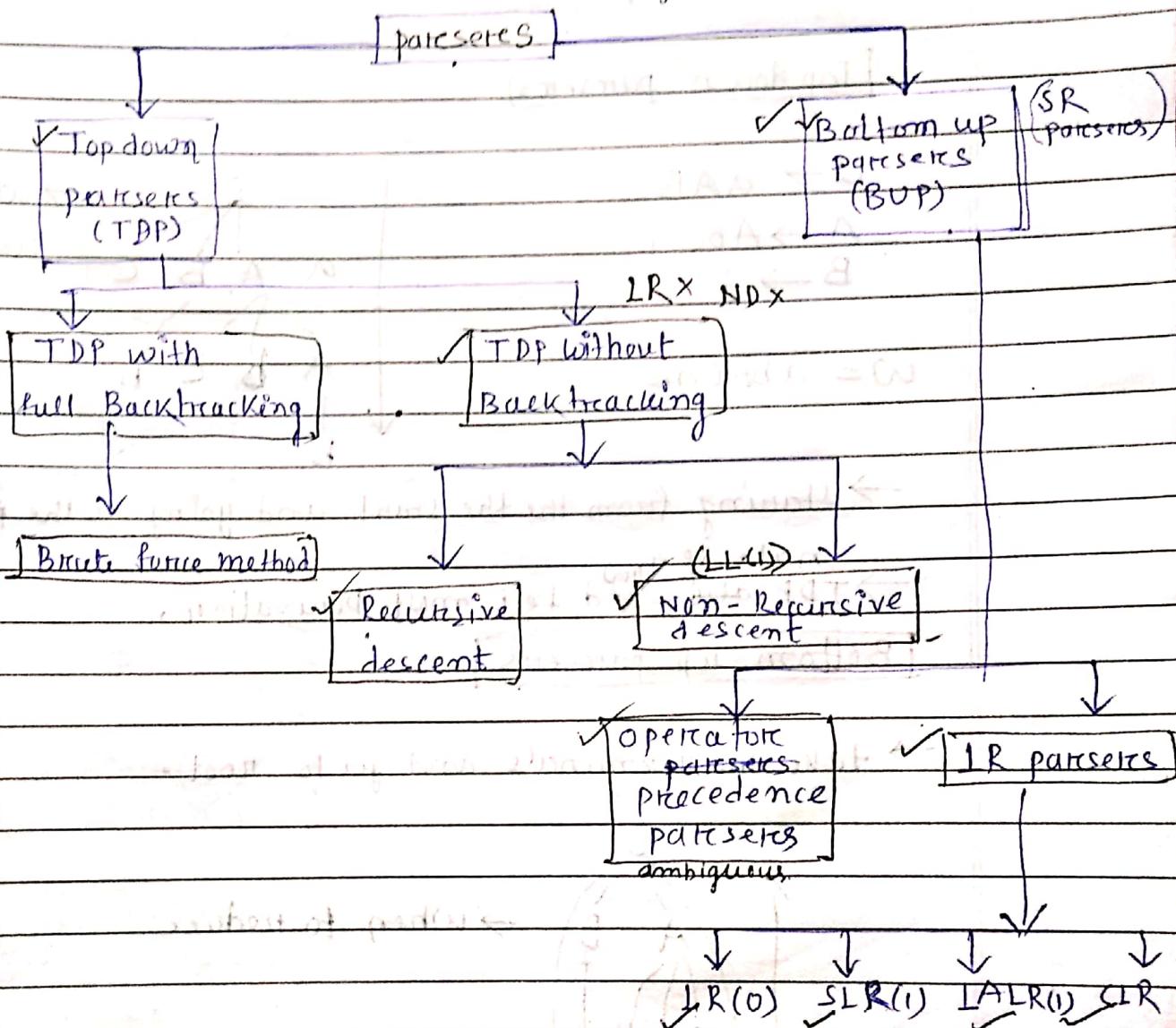
atlantis

Date  
Page

17

- Introduction of parsers : (gatenotes.in)

ambiguous X



→ parsers is syntax Analyser.

→ SR parsers means shift-reduce parsers.

→ TDP without backtracking not accept - Left recursive grammar and also Non-Deterministic grammar.

→ parsers not accept ambiguous grammar.  
only operator precedence parsers accept ambiguous grammar.

- Basic difference between Top-down parsers and Bottom up parsers:

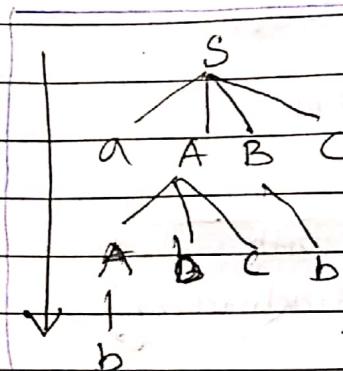
### Top down parsers

$$S \rightarrow aABe$$

$$A \rightarrow Abc/b$$

$$B \rightarrow d$$

$$w = abbcede$$

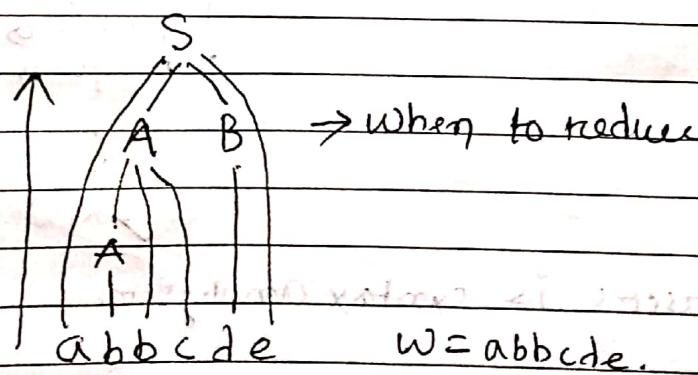


→ what to use.

→ starting from the start and going to the terminal.  
 → TDP also follows left most Derivation.

### Bottom up parsers

→ take the terminals and go to root.



→ when to reduce

$$S \Rightarrow a(A)Be$$

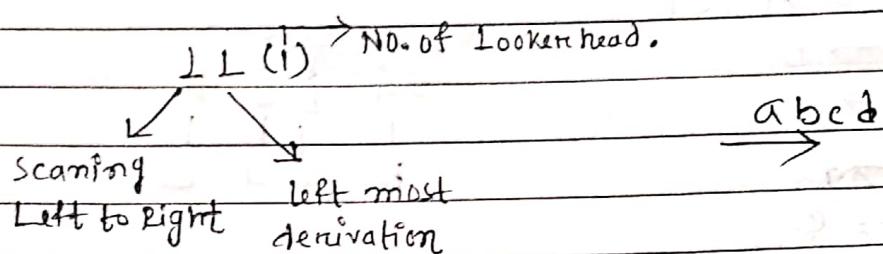
$$\Rightarrow a(A)d e$$

$$\Rightarrow a(Abc)de$$

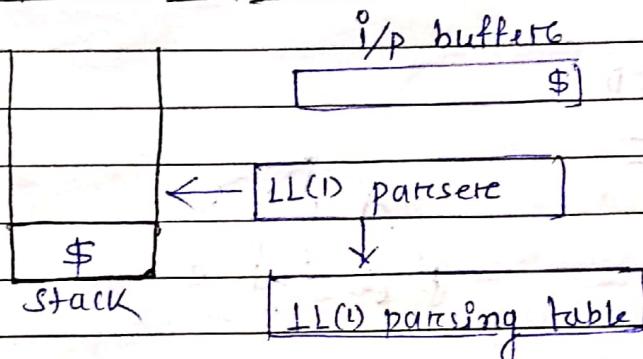
$$\Rightarrow a(b)cde$$

→ BUP follows Right most Derivation.

- LL(1) parsers :



- Components of parsers =

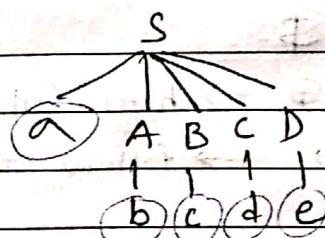


→ '\$' is to guess when stop.

- First()
- Follow()

### First()

Ex:-  $S \rightarrow a A B C D$   
 $A \rightarrow b$   
 $B \rightarrow c$   
 $C \rightarrow d$   
 $D \rightarrow e$



hence, first of S is a.

" " A " b

" " B " c

" " C " d

first of D " e

**[Ex]**

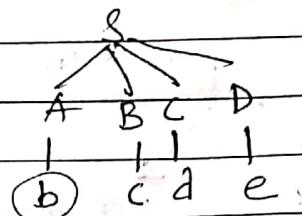
$$S \rightarrow ABCD$$

$$A \rightarrow b$$

$$B \rightarrow c$$

$$C \rightarrow d$$

$$D \rightarrow e$$



→ here first of  $S$  is ' $b$ '.

**[Ex]**

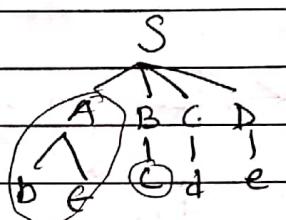
$$S \rightarrow A \overline{BCD}$$

$$A \rightarrow b/e$$

$$B \rightarrow c$$

$$C \rightarrow d$$

$$D \rightarrow e$$



→ here in this case first of  $S$  is ' $c$ '.

**Follow:**

what a terminal <sup>which can</sup> follow a variable in the process of derivation.

 $S \$$ 
 $abcd \$$ 
 $ABCD \$ \rightarrow$  here follow of  $D$  is '\$'.

 $ABD \$ \rightarrow$  follow of  $B$  is ' $d$ '.

$$S \rightarrow ABCD$$

$$A \rightarrow b/e$$

$$B \rightarrow c$$

$$C \rightarrow d$$

$$D \rightarrow e$$

→ follow of  $D$  is follow of ' $S$ '.

→ here in this case follow of  $S$  -  $\{\$\}$

follow of  $A$  is first of  $BCD = 'c'$ .

So, follow of  $A$  is  $\{c\}$ .

follow of B is  $\{d\}$ .

ii) C is  $\{e\}$ .

follow of D is  $\{\$\}$ .

$$A \rightarrow BC$$

- Example - find first and follow in LL(1):

**Ex-1**

	first	follow
6 S $\rightarrow \overset{e}{A} \overset{e}{B} CDF$	$\{a, b, c\}$	$\{\$\}$
1 A $\rightarrow a/e$	$\{a, e\}$	$\{b, c\}$
2 B $\rightarrow b/e$	$\{b, e\}$	$\{c\}$
3 C $\rightarrow c$	$\{c\}$	$\{a, e, \$\}$
4 D $\rightarrow d/e$	$\{d, e\}$	$\{e, \$\}$
5 E $\rightarrow e/e$	$\{e, e\}$	$\{\$\}$

$\rightarrow$  follow of A is first of ABCDF = ~~BCDF~~  $\{b, c\}$ .

**Ex-2**

	first	follow
3 S $\rightarrow \overset{B}{B} b / \overset{C}{a} d$	$\{a, b, c, d\}$	$\{\$\}$
1 B $\rightarrow a \overset{B}{B} / e$	$\{a, e\}$	$\{b\}$
2 C $\rightarrow c \overset{C}{C} / e$	$\{c, e\}$	$\{a\}$

**Ex-3**

	first	follow
5 E $\rightarrow TE'$	$\{id, (\}\)$	$\{\$\}\}$
4 E' $\rightarrow +TE'/e$	$\{+, e\}$	$\{\$\}, \{+\}$
3 T $\rightarrow FT'$	$\{id, (\}\)$	$\{+, \$\}\}$
2 F $\rightarrow *FT'/e$	$\{* , e\}$	<del>del</del> $\{+, \$\}, \{+, +, \$\}\}$
1 F $\rightarrow id / (E)$	$\{id, (\}\)$	<del>del</del> $\{+, +, \$\}\}$

Ex-4:

	first	follow
1 $S \rightarrow A(CB)/CB/Ba$	{d, g, h, t, b, a}	{\\$, }
3 $A \rightarrow d/a/BC$	{d, g, h, t}	{h, g, \\$}
2 $B \rightarrow g/t$	{g, t}	{\\$, a, h, g}
1 $C \rightarrow h/e$	{h, t}	{b, h, g, \\$}

Ex-5:

	first	follow
$S \rightarrow aABb$	{a}	{\\$}
$A \rightarrow c/e$	{c, e}	{d, b, }
$B \rightarrow d/e$	{d, e}	{b}

Ex-6:

	first	follow
$S \rightarrow aBDh$	{a}	{\\$}
$B \rightarrow c/o$	{c}	{g, f, h}
$C \rightarrow b/o/e$	{b, e}	{g, f, b}
$D \rightarrow EF$	{g, f, e}	{h}
$E \rightarrow g/e$	{g, e}	{f, h}
$F \rightarrow f/e$	{f, e}	{h}

 $\overleftarrow{A \rightarrow CAB}$ here follow of  $A^*B$  is = follow of 'A' . $\overleftarrow{A \rightarrow ABCD}$ 

here follow of B is first of 'C' and 'D' .

• Construction of LL(1) parsing table

Example - 1

	first	follow
$E \rightarrow TE'$	{id, \$}	{\$, )}
$E' \rightarrow +TE'/E$	{+, E}	{\$, )}
$T \rightarrow FT'$	{id, \$}	{+, (, ), \$, )}
$T' \rightarrow *FT'/E$	{*, F}	{+, \$, )}
$F \rightarrow id/( E )$	{id, C}	{*, +, \$, )}

→ This LL(1) parser is a top down parser.

→ The main purpose of top down parser is when have two alternative for a variable. find out which production should choose.

		terminals					
		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'		$T' \rightarrow F$	$T' \rightarrow *FT'$			$T' \rightarrow E$	$T' \rightarrow E$
F	$F \rightarrow id$				$F \rightarrow (F)$		

✓ All the 'E' production have to place under follow of 'left hand side'.

W Where should place the production is depends on the first or first of right hand side.

→ To construct parser's tree, this table is used.

- Use of parsing to construct a parser's tree -

[Example - 2]

$(( ))$

$$S \rightarrow (S)/E$$

first  
 $\{C, E\}$

follow  
 $\{\$, )\}$

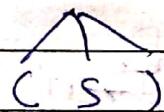
table:

S	(	)	\$
	$S \rightarrow (S)$	$S \rightarrow E$	$S \rightarrow E$

generate:  $(( )) \$$

$\swarrow \uparrow \uparrow \uparrow$

bottom of the stack



|  
E

→ LL(1) parser algo. Start with '\$' in the bottom of the stack and '\$' in the end of the input.

when \$, \$ match then we can say that when we see \$ & \$ then we can say it is successful/unsuccessful matching.

Example-3

	first	follow
$S \rightarrow AaAb / BbBa$	{t, a, b}	{\$}
$A \rightarrow \epsilon$	{\epsilon}	{a, b}
$B \rightarrow \epsilon$	{\epsilon}	{b, a}

table =

	a	b	\$
S	$S \rightarrow AaAb$	$S \rightarrow BbBa$	
A	$A \rightarrow \epsilon$	$A \rightarrow \epsilon$	
B	$B \rightarrow \epsilon$	$B \rightarrow \epsilon$	

→ In every cell of table we got '1' production, so that this grammar is LL(1).

→ Left recursion and Non-Deterministic grammar are not can not be used for LL(1).

$$A \rightarrow \alpha_1 / \alpha_2 / \alpha_3 / \alpha_4 / \dots / \alpha_n$$

A grammar has to be LL(1) when first of  $\alpha_1, \alpha_2, \alpha_3, \dots, \alpha_n$  are mutually exclusive (means no common)

• Test that given grammar is LL(1) or not:

		first	follow
①	$S \rightarrow aSbS$ <del><math>\quad \quad \quad / bSas</math></del>	{a, b, ε}	{b, a, \$}
	/ε		

	a	b	\$
S	$S \rightarrow aSbS$	$S \rightarrow bSas$	
	$S \rightarrow \epsilon$	$S \rightarrow \epsilon$	$S \rightarrow \epsilon$

→ some cell contain more than one production  
∴ therefore given grammar is not LL(1).

		first	follow
②	$S \rightarrow aABb$ ✓	{a}	{\$}
	$A \rightarrow c/\epsilon$	{c, ε}	{a, b}
	$B \rightarrow d/\epsilon$	{d, ε}	{b}

	a	b	c	d	\$
S	$S \rightarrow aABb$				
A		$A \rightarrow \epsilon$	$A \rightarrow \epsilon$	$A \rightarrow \epsilon$	
B		$B \rightarrow \epsilon$		$B \rightarrow d$	

→ each cell contain only '1' production, therefore this given grammar is LL(1).

	first	follow
$S \rightarrow A/a$	{a}	{\$}
$A \rightarrow a$	{a}	{\$}

	a	\$
S	$S \rightarrow A, S \rightarrow a$	
A	$A \rightarrow a$	

→ Cell contain more than '1' production, not LL(1).  
→ also it is a ambiguous grammar.

(4)

		first	follow
✓	$S \rightarrow aB / \epsilon$	{a, $\epsilon^2$ }	{ $\$^2$ }
	$B \rightarrow bC / \epsilon$	{b, $\epsilon^2$ }	{ $\$^2$ }
	$C \rightarrow cS / \epsilon$	{c, $\epsilon^2$ }	{ $\$^2$ }

table

	a	b	c	\$
S	$S \rightarrow aB$			$S \rightarrow \epsilon$
B		$B \rightarrow bC$		$B \rightarrow \epsilon$
C			$C \rightarrow cS$	$C \rightarrow \epsilon$

→ each cell contain single one production.  
therefore LL(1).

(5)

		first	follow
✓	$S \rightarrow AB$	{a, b, $\epsilon^2$ }	{ $\$^2$ }
	$A \rightarrow a / \epsilon$	{a, $\epsilon^2$ }	{b, $\$^2$ }
	$B \rightarrow b / \epsilon$	{b, $\epsilon^2$ }	{ $\$^2$ }

parse table =

	a	b	\$
S	$S \rightarrow AB$	$S \rightarrow AB$	$S \rightarrow AB$
A	$A \rightarrow a$	$A \rightarrow \epsilon$	$A \rightarrow \epsilon$
B		$B \rightarrow b$	$B \rightarrow b$

→ each cell contain single one production,  
therefore this grammar is LL(1).

(6)

		first	follow
✗	$S \rightarrow aSA / \epsilon$	{aS}	{c, $\$^2$ }
	$A \rightarrow c / \epsilon$	{c, $\epsilon^2$ }	{c, $\$^2$ }

	a	c	\$
S	$S \rightarrow aSA$	$S \rightarrow aSA$	$S \rightarrow \epsilon$
A		( $A \rightarrow c$ $A \rightarrow \epsilon$ )	$A \rightarrow \epsilon$

→ one cell contained more than '1' production, so therefore it is not LL(1).

(7)

$S \rightarrow A$	first {a, b, c, d}	follow {b}
$A \rightarrow Bb/Cd$	{a, b, c, d}	{\$}
$B \rightarrow aB/E$	{a, E}	{b}
$C \rightarrow cC/E$	{c, E}	{d}

table

	a	b	c	d	\$
S	$S \rightarrow A$	$S \rightarrow A$	$S \rightarrow A$	$S \rightarrow A$	
A	$A \rightarrow Bb$	$A \rightarrow Bb$	$A \rightarrow Bb$	$A \rightarrow Bb$	
B	$B \rightarrow aB$	$B \rightarrow E$			<del>break</del>
C			$C \rightarrow cC$	$C \rightarrow E$	

→ each cell contains only 1 production so it is LL(1).

(8)

	a	b	\$	first -	follow
X	$S \rightarrow aAa/E$			{a, E}	{\$, a}
	$A \rightarrow abS/CE$			{a, E}	{a}
S	$S \rightarrow aAa$				$S \rightarrow E$
A	$A \rightarrow abS$				

→ not LL(1).

(9)

$S \rightarrow (IE)^*$

⑨

		first	follow.
X	$S \rightarrow {}^0 E + {}^0 S' / a$	$\{{}^0, a\}$	$\{{}^0, e\}$
	$S' \rightarrow eS / e$	$\{e, e\}$	$\{{}^0, e\}$
	$E \rightarrow b$	$\{b\}$	$\{t\}$

table

	${}^0$	$t$	$a$	$b$	$e$	${}^0$
$S$	$S \rightarrow {}^0 E + {}^0 S' / a$		$S \rightarrow a$			
$S'$					$(S' \rightarrow eS)$	$S' \rightarrow e$
$E$				$E \rightarrow b$		

→ It is not LL(1).

### • Recursive descent parser =

#### Example

$$E \rightarrow {}^0 E'$$

$E' \rightarrow + {}^0 E' / e$  → each time variable has a function.

$E()$

{

1. If ( $l == {}^0$ )

2. { match ( ${}^0$ );

3.  $E'();$

}

$E'()$

{

1. If ( $l == +$ )

{

2. match (+);

3. match ('0');

4.  $E'();$

}

else

return;

}

match (char t)

{

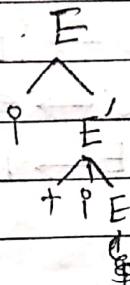
if ( $l == t$ )

$l = getch();$

else

print ("error");

}



$l = getch$   
(incrementing)

main()

{

1. E();

2. if ( $l == '$'$ )

3. print ("parsing success");

}

according to grammar give it is generated.

main() E() E'() E'() ~~E()~~



(Recursion stack)

→ write function for every variable, then call the function using the stack provided by operating system.

- Operator precedence parser: (Bottom up parsers)

- Operator grammar:

✓  $E \rightarrow E+E/E*E/id$  → no two variable are adjacent.

X  $E \rightarrow EAE/id$   
 $A \rightarrow +/*.$   $\Rightarrow E \rightarrow E+E/E*E/id$  ✓

Example: (given grammar are operator grammar or not)

$$① S \rightarrow SAS/a$$

$$A \rightarrow bSb/b$$

→ this grammar is not operator precedence grammar, because two variable are not adjacent.

now going to convert this grammar into operator precedence grammar =

$$S \rightarrow SbSb/SbS/a$$

$$A \rightarrow bSb/b$$

→ operator precedence can parse ambiguous grammar.

- Operation relation table:

→ Using this table we can make parsing =

① example - 8

$$E \rightarrow E+E / E * E / id$$

(id > \* > + > \$)

table =

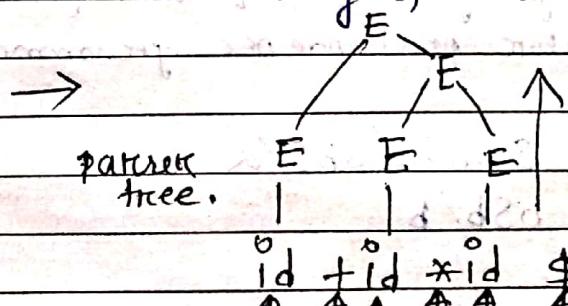
	id	+	*	\$	+ > +
id	-	>	>	>	* > *
+	<.	>	<.	>	
*	<.	>	>	>	
\$	<.	<.	<.	-	

(Operation precedence table)

(operator relation table)

example ① parse (id + id \* id \$) → i/p

using operator precedence table.



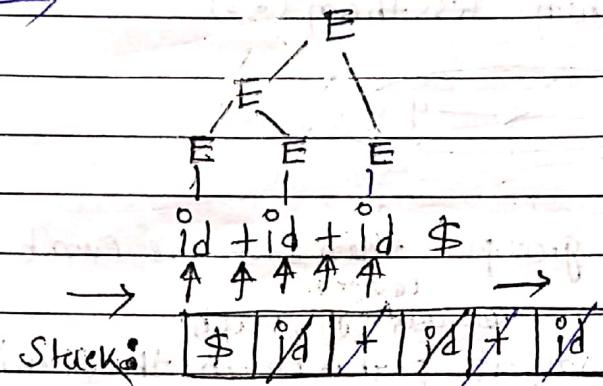
Stack: [\$ | id | + | id | \* | id ]

- When top of the stack less than or equal to input then push i/p in stack.

- When top of the stack are greater than to input then pop the top of the stack no terminal.

## (Example) 2

Construct parse tree of  $id + id \cdot id \$$  using operator precedence table.



→ left most '+' is greater than right most '+'.

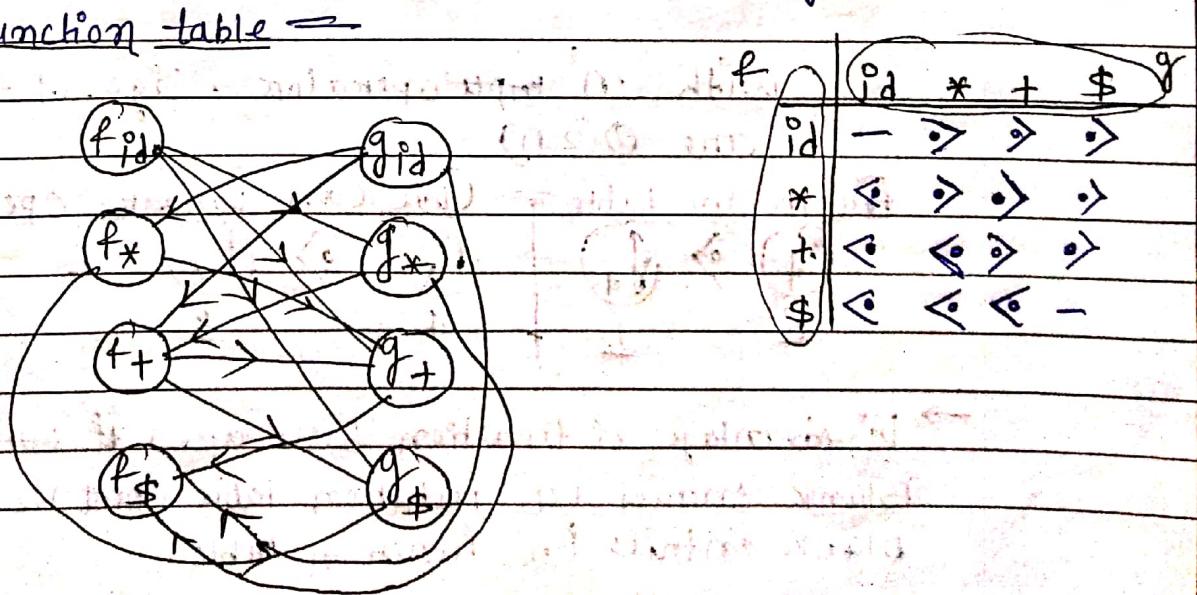
→ we will get one parse tree → whether it is ambiguous.

• Disadvantage of operation relation table =

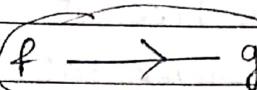
→ if we have ' $i$ ' operators ~~use exist~~ then size of the table are  $i^2 = 16$ .

if we have ' $n$ ' operators then size of table are  $n^2$  or  $O(n^2)$ .

→ To decrease the size of the table, we go for operator function table →



When greater than ( $\Rightarrow$ )



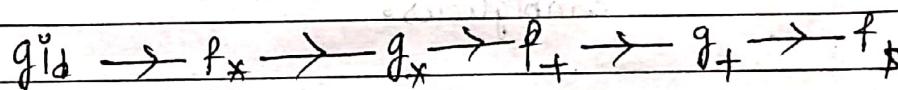
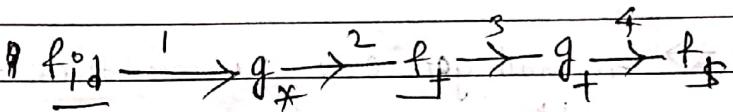
When less than ( $\Leftarrow$ )



any cycle

$\rightarrow$  If 1 in this graph, then stop we can't make function table.

$\rightarrow$  find the longest path from the starting node.



### Function table

	id	+	*	/
f	4	2	4	0
g	5	1	3	0

$\rightarrow$  This table contain same information (like relation table) but, the size of the table less.

with ' $n$ ' input operators, size of the table are  $\Theta(2^n)$

from the table = (We can compare operators)

$f_+ > f_+$	$f_* > f_*$
1	3

$\rightarrow$  Disadvantage of function table are, if when every blank entries in relation table, then we get non-blank entries in function table.

Example =

$P \rightarrow SR/S$  → not operator precedence gram.  
 $R \rightarrow bSR/bS$  because two variable are adjacent.  
 $S \rightarrow WbS/W$   
 $W \rightarrow L * W/L$   
 $L \rightarrow id$ .

$\Rightarrow P \rightarrow SR/S \rightarrow \text{①}$   
 $R \rightarrow bSR/bS \rightarrow \text{②}$   
 $S \rightarrow WbS/W$   
 $W \rightarrow L * W/L$   
 $L \rightarrow id$

$P \rightarrow$	paragraph.
$S \rightarrow$	sentence
$R \rightarrow$	recursive
$b \rightarrow$	Blank
$W \rightarrow$	word
$L \rightarrow$	letter.
$id \rightarrow$	Identifier.

conversion into operator precedence grammar.

$P \rightarrow SbSR/SbS/S$   
 $\Rightarrow P \rightarrow SbP/SbS/S$   
 $S \rightarrow WbS/W$  → 'b' is right associative.  
 $W \rightarrow L * W/L$  → so right most 'b' will get  
 $L \rightarrow id$ . → '\*' also right higher precedence compare  
 to left 'b'.

operator relation table =

	id	*	b	\$
id	-	>	>	>
*	<	<	>	>
b	<	<	<	>
\$	<	<	<	-

→ if grammar are not ambiguous then we can make table from given grammar.

④ table =

	a	c	)	,	\$
a	↑	↑	↑	↑	↑
(	<	<	≡	<	>
)	↑	↑	↑	↑	↑
,	<	<	↑	↑	↓
\$	<	<	↑	↑	↑

(operator relation table)

	a	c	)	,	\$
a	2	0	2	2	0
f	2	0	2	2	0
g	3	3	0	1	0

(operator function table)

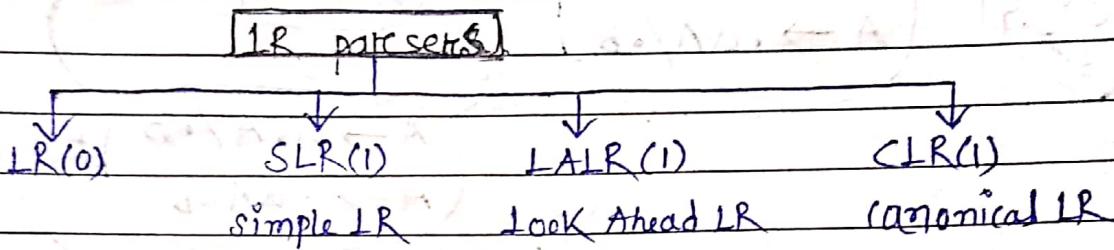
$$f_C = g_C$$

$f_C$

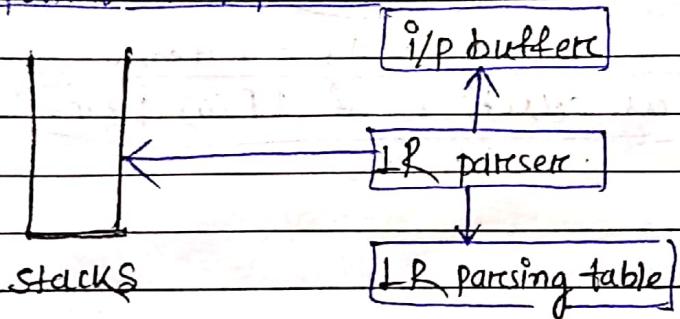
$g_C$

$f_C, g_C \rightarrow$  both w same value.

- LR parsers: (Bottom up parser)



Components of LR parsers =



→ LR(0) items is used to construct the LR(0) and SLR(1) parsers.

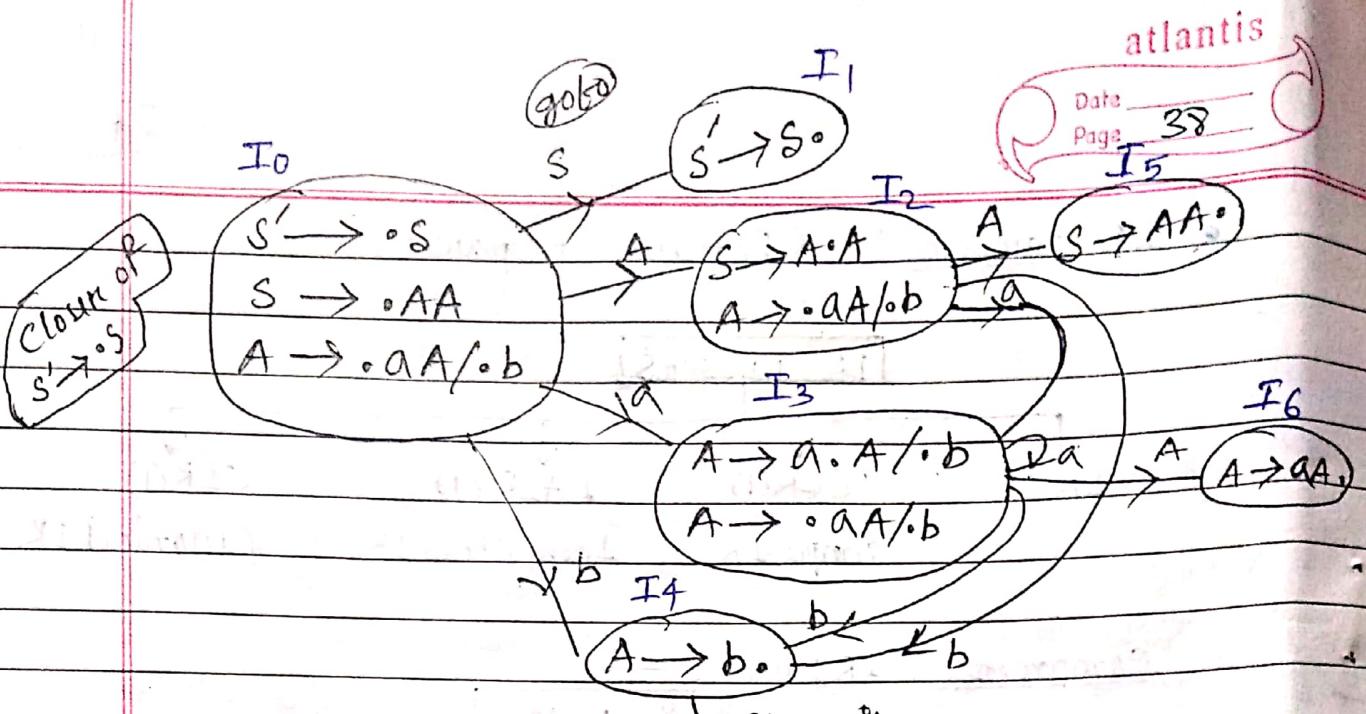
→ LR(1) items is used to construct the LALR(1) and CLR(1) parsers.

Construct LR(0) parsing table =

$S' \rightarrow S$
$S \rightarrow AA \quad ①$
$A \rightarrow aA/b \quad ② \quad ③$

→ add one more production with given grammar.

→ closer: When a dot is left of a variable we have  
 $S' \rightarrow \cdot S$  to add all the production of  
 $S \rightarrow \cdot AA$  dot in the left side of  
 $A \rightarrow \cdot aA/b$



Canonical collection of LR(0) items

Steps —

- (i) Take the grammar add a new production.
- (ii) and start with first front production by dot of the set left of S.
- (iii) then all applied closure.
- (iv) then find out transition (goto move) and again apply closure.

→ some states contain final items.

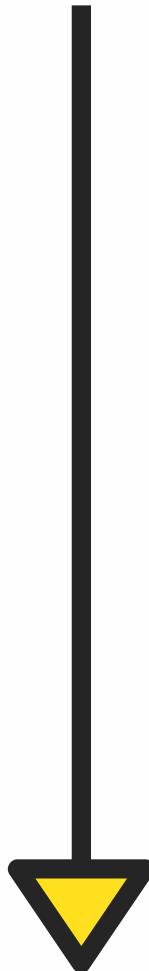
(Any item date at the end is called final item)

• LR(0) parsing table for Canonical collection of LR(0) items —

→ next page —

**TO DOWNLOAD THE COMPLETE PDF**

**CLICK ON THE LINK  
GIVEN BELOW**



[WWW.GATENOES.IN](http://WWW.GATENOES.IN)

**GATE CSE NOTES**