Compiler Design

Gate Notes

INDEX

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

S. No.	Date	Title	Page No.	Teacher's Sign / Remarks
1.		Introduction of CD + Lexical Analysis. (1) ✓		
2.		Parsing Techniques. (2)		
3.		Syntax Directed Translation. (3)		
4.		Intermediate code generation. (4)		
5.		Runtime Environment. (4)		
6.		Code Optimization. (4)		

06/05/2019

INTRODUCTION

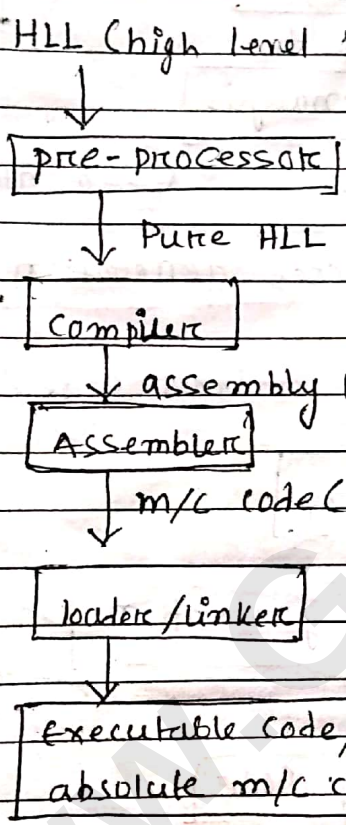
atlantis
Date 06/05/2019
Page 1

Compiler Design

(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.



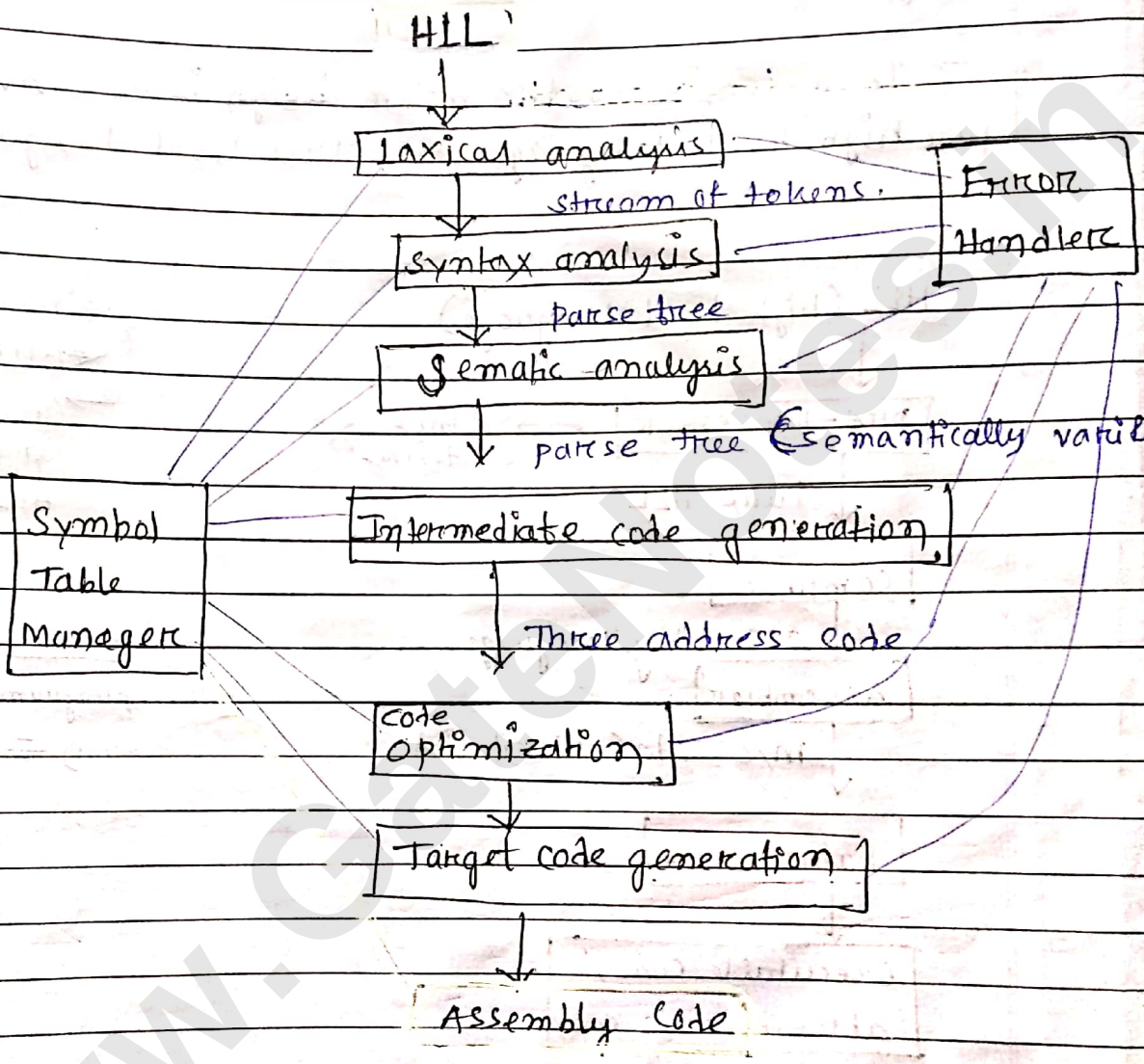
{ <#inclu> – file inclusion
<#define> – Macro expansion
→ (If any language contain this type of line it called HLL)

→ 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 ~~te~~text are source text is broken into tokens.

ex: If (A>B) → 10 tokens.
a = 10;

Example of all the phases of compiler —

$x = a + b * c$; → source program.

⇓

Lexical analyser

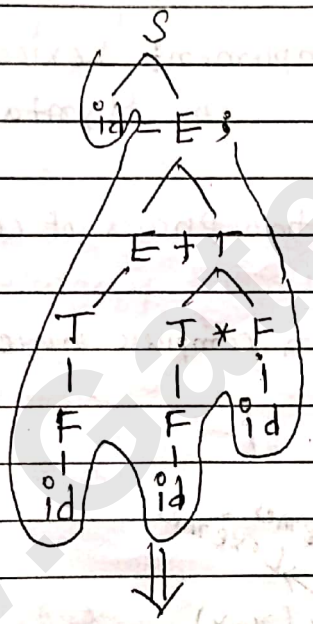
⇓

id = id + id * id → (id = identifier)

⇓

heart of the compiler — Syntax analyser ⇐

$S \rightarrow id = E;$	(CFGs)
$E \rightarrow E+T/T$	
$T \rightarrow T*F/F$	
$F \rightarrow id$	



⇓

Semantic analyser (parse tree semantically verified)

⇓

ICG ✓ frontend

⇓

$t_1 = b * c$

$t_2 = a + t_1$

$x = t_2$

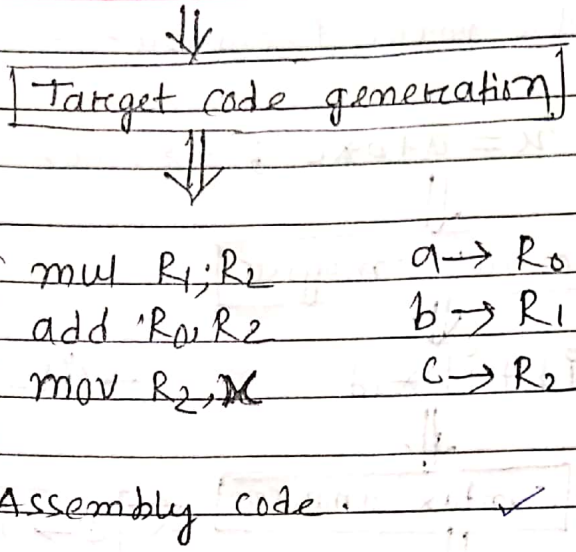
⇓

Code optimization

⇓

$t_1 = b * c$

$x = a + t_1$



→ Lex (tool) used to implement 'Lexical analyser'.
→ 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:

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

→ Lexical analyser removing the comments and ors 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 ==

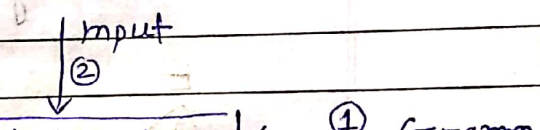
(i) int/ max/(/x,/y/)/
int/ x,/y;/
/* find max of x and y */
{/
return/(x>y?/x:y)/;/
}/

→ 25 (tokens)

(ii) printf/(/"%d Hai",/&x/)/;/

→ 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 → variable
T → Terminals
P → Production
S → start symbol.

Ex: (E) → E+E
/E*E
/id

here,
V = {E}
T = {+, *, id}
start symbol = E.

(id + id * id) generate this string using given rule (LMD)

Left most derivation = (LMD)

$E \Rightarrow E+E$
$\Rightarrow id+E$
$\Rightarrow id+E*E$
$\Rightarrow id+id*E$
$\Rightarrow id+id*id$

Right most derivation =

$E \Rightarrow E+E$
$\Rightarrow E+E*E$
$\Rightarrow E+E*id$
$\Rightarrow E+id*id$
$\Rightarrow id+id*id$

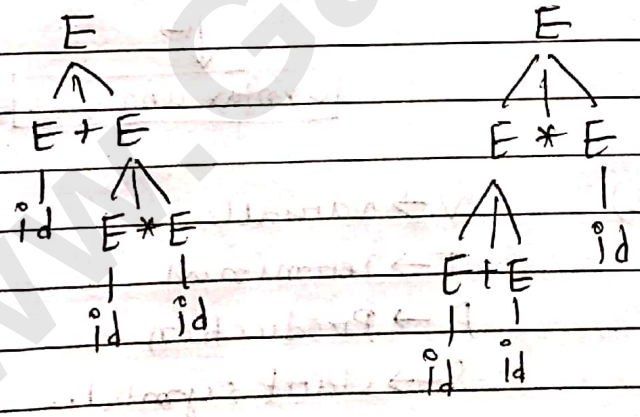
LMD =

$E \Rightarrow E*E$
$\Rightarrow E+E*E$
$\Rightarrow id+E*E$
$\Rightarrow id+id*E$
$\Rightarrow id+id*id$

RMD =

$E \Rightarrow E*E$
$\Rightarrow E*id$
$\Rightarrow E+E*id$
$\Rightarrow E+id*id$
$\Rightarrow id+id*id$

- Derived using parse tree -



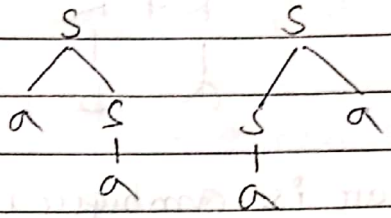
→ To the given grammer and string if we get more than one LMD or RMD and parse tree then the grammer is called ambiguous grammar.

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

• Find given grammars are ambiguous or not :

① $S \rightarrow aS/Sa/a$

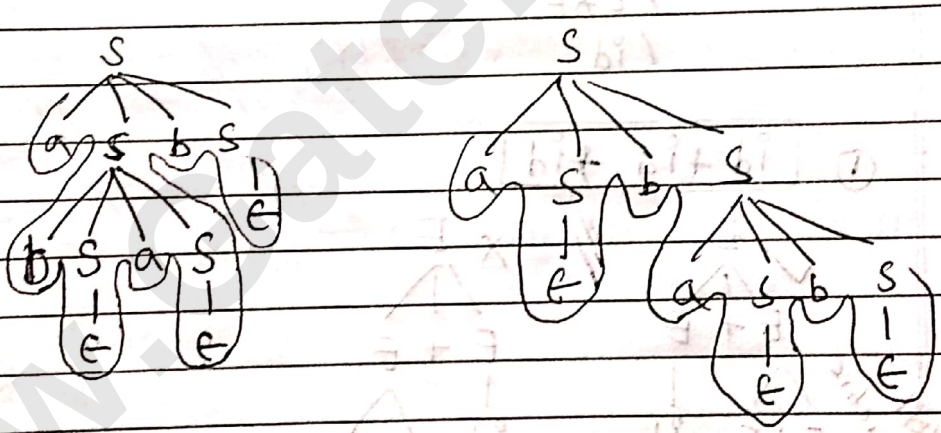
→ $w = aa$



→ This grammar is ambiguous.

② $S \rightarrow aSbS/bSaS/\epsilon$

→ $w = abab$

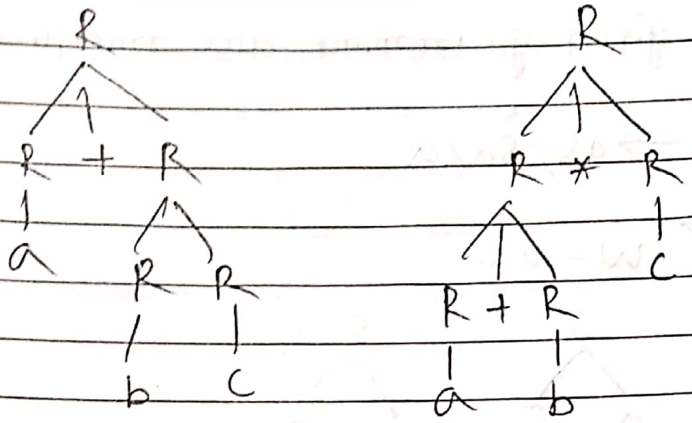


(abab) (abab)

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

③ $R \rightarrow R+R/RR/R^*/a/b/c$

→ $w = a+bc$



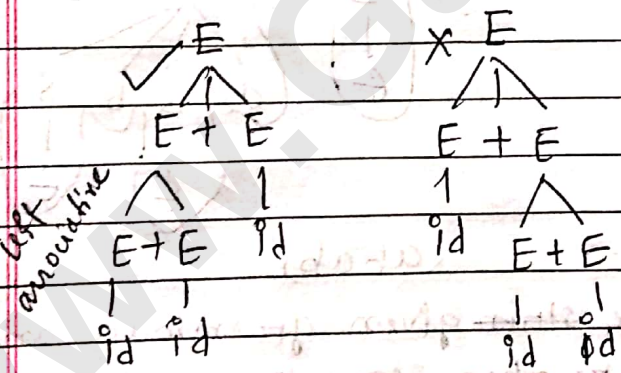
→ this grammar is ambiguous grammar.

• Ambiguous grammar and making them unambiguous:

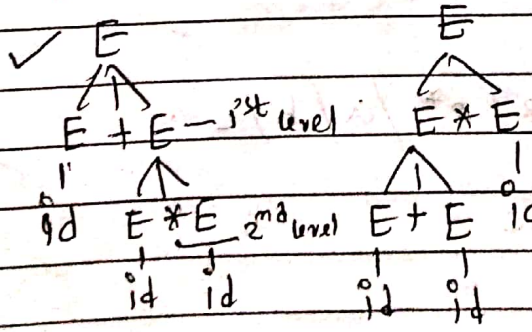
Ex-1

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

① id + id * id

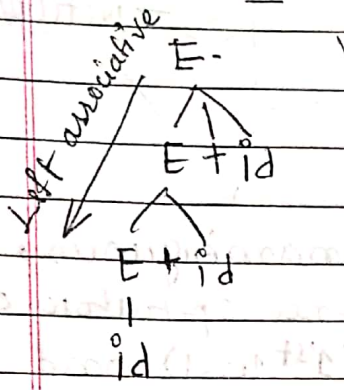


② id + id * id



~~① Let operators are left associative.~~
~~② Let operator is~~

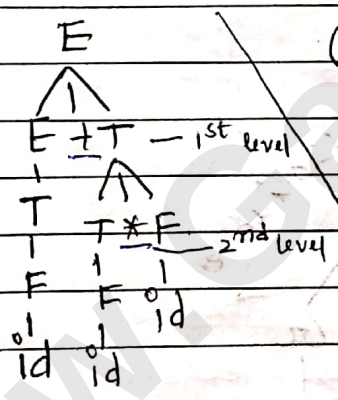
① $E \rightarrow E+id \mid id$ (This grammar is left recursive)



$((id+id)+id)$

→ Now the grammar is unambiguous.

② $E \rightarrow E+T \mid T$
$T \rightarrow T*F \mid F$
$F \rightarrow id$



$(id+(id*id))$

→ Now the grammar is unambiguous.

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

precedence: $\uparrow > * > +$
($\uparrow$: 3rd level, $*$: 2nd level, $+$: 1st level)

③ $E \rightarrow E+T \mid T$
$T \rightarrow T*F \mid F$
$F \rightarrow G \uparrow F \mid G$
$G \rightarrow id$

here, '+' and '*' is left associative,
'↑' is right associative.

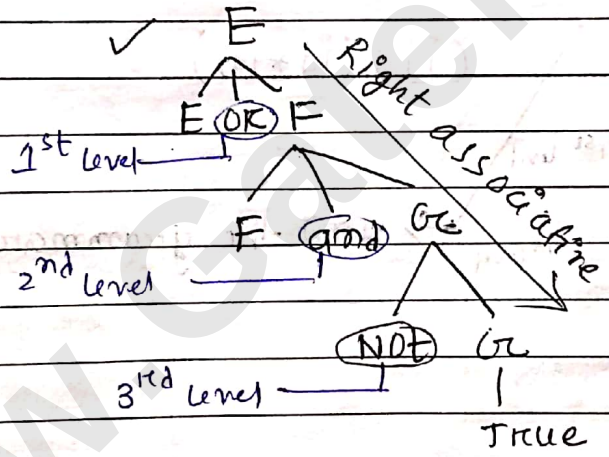
④ boolean expression,

Convert

bExp → bExp OR bExp.	E → E OR F / F
/ bExp and bExp.	F → F and Gr / Gr
/ not bExp	Gr → Not Gr / True / False.
/ True	
/ False.	

→ This grammar is unambiguous, because lowere precedence operator closest to the start symbol (1st level) and higher prece operator are least level.

→ And this grammar follow the Associativity rule.



⑤ R → R+R (convert into Unambiguous grammar)
/ RR
/ R*
/ a
/ b
/ c

E → E + T / T
T → T * F / F
F → F* / a / b / c

↳ Now this gramma is unambiguous grammar.

⑥ Given grammar –

Ⓐ → Ⓐ $B / B (($, #, @ are operators)

Ⓑ → Ⓑ # C / C

Ⓒ → Ⓒ @ D / D (here, $, #, @ are left recursive)

D → d

→ $ ·> $

·>

@ ·> #

a ·> a / higher precidence than

→ $ <· # <· @ Operators are evaluated.

⑦ Given grammar –

E → Ⓔ * F → here, '*' is left associative. (left recur)
/ F + Ⓔ → '+' is Right associative. (Right recur)
/ F

F → Ⓕ − Ⓕ → Associativity is undefined. because, here, '−' define left or Right associative both.
/ id

→ And This grammar is ambiguous.

* ≐ + (equal).

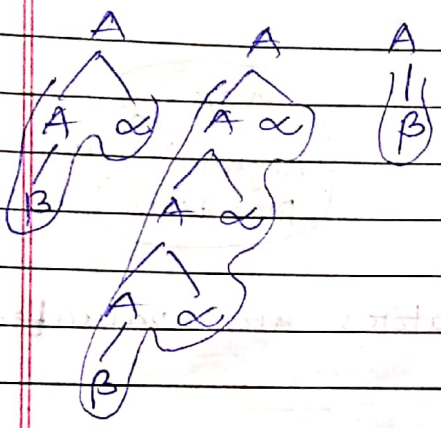
(high precidence) * ·> *

+ <· + (high precidence)

Recursion

Left Recursion

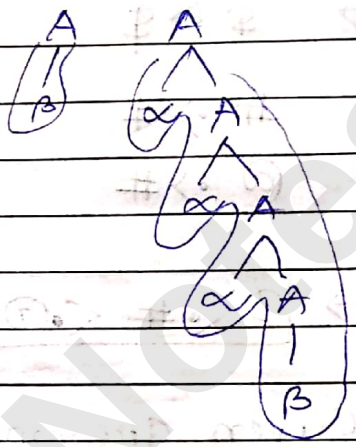
$A \rightarrow A\alpha / \beta$



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

Right Recursion

$A \rightarrow \alpha A / \beta$



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

- $\beta\alpha^*$

$A \rightarrow \beta\alpha^*$

**
$$A \rightarrow \beta A' \qquad A' \rightarrow \epsilon / \alpha A' \iff A \rightarrow A\alpha / \beta.$$

→ To eliminate left recursion we can follow this above rule.

Example — eliminate the left recursive,

$$\underset{(A)}{E} \rightarrow \underset{(A)}{E}\underset{(\alpha)}{+T} / \underset{(\beta)}{T}$$

$$\Updownarrow$$

$$E \rightarrow TE' \qquad E' \rightarrow \epsilon / +TE'$$

→ This is equivalent grammar and not left recursive grammar.

Example- Eliminate left recursion from this grammar-

① $S \rightarrow S0S1S/01$
(A) (A) (α) (β)

$\Rightarrow$ $S \rightarrow 01S'$

$S' \rightarrow \epsilon / 0S1SS'$

② $S \rightarrow (L)/x.$
$L \rightarrow L,S/S.$
(A) (A) (α) (β)

$\Rightarrow$ $S \rightarrow (L)/x.$

$L \rightarrow SL'$

$L' \rightarrow \epsilon / ,SL'$

③ $A \rightarrow A\alpha_1/A\alpha_2/A\alpha_3/\ldots\ldots$
$/\beta_1/\beta_2/\beta_3/\ldots\ldots$

$\Rightarrow$ $A \rightarrow \beta_1A'/\beta_2A'/\beta_3A'/\ldots$

$A' \rightarrow \alpha_1A'/\alpha_2A'/\alpha_3A'/\ldots\ldots$ ε.

Grammar
- Ambiguous ✗
- unambiguous ✓

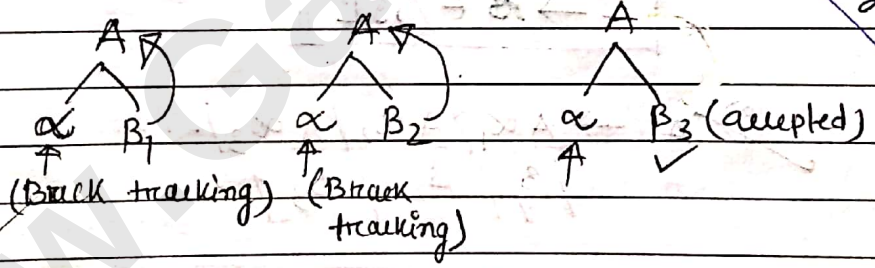
Grammar
- Left recursive ✗
- Right recursive ✓

Grammar
- Deterministic ✓
- Non-Deterministic ✗

• Non-Deterministic :

$A \rightarrow \alpha\beta_1 / \alpha\beta_2 / \alpha\beta_3$ ← Left factoring

$\alpha\beta_3$

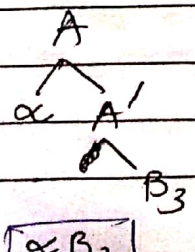


• Left factoring procedure or eliminating non-determinism :

$A \rightarrow \alpha A'$

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

generate $\alpha\beta_3$



$\alpha\beta_3$

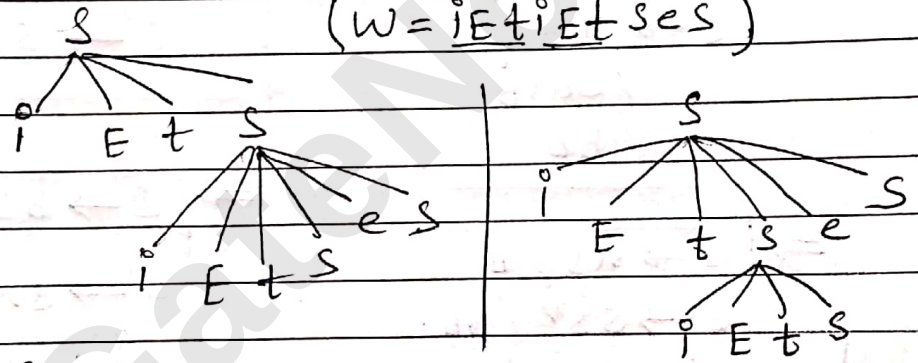
Example –

(i) $S \rightarrow iEtS$
$/\ iEtSeS$ → (Non-Deterministic)
$/\ a$
$E \rightarrow b$

left factoring

(ii) $S \rightarrow iEtSS'/a$
$S' \rightarrow \epsilon/eS$
$E \rightarrow b$ → (Deterministic)

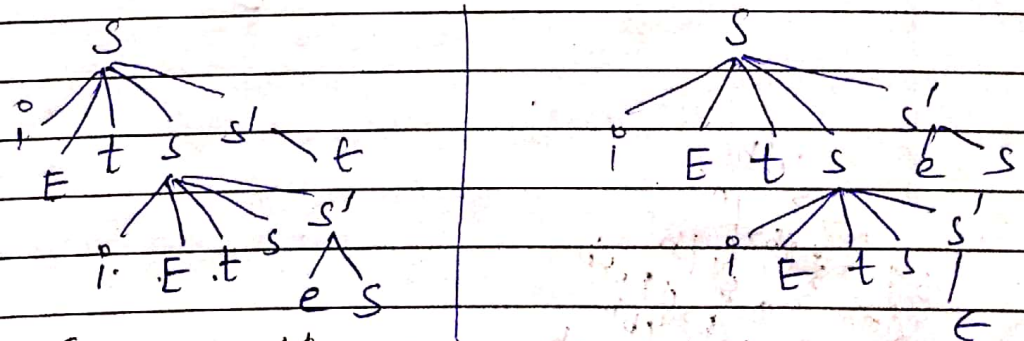
(i) $(W = iEtiEtSeS)$



(ambiguous grammar)

→ for the string we get 2 parse tree from the given grammar.

(ii) $W = iEtiEtSeS$



(also ambiguous grammar)

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

Example –

(Non-Deterministic)

$S \to aSSbS$
$/\ aSaSb$
$/\ abb$
$/\ b$

⇒

$S \to aS' / b$
$S' \to SbSS' / SaSb / bb$

⇒

$S \to aS' / b$
$S' \to SS'' / bb$
$S'' \to SbS / aSb$

(Deterministic)

Non-Deterministic

$S \to bSSaaS$
$/\ bSSaSb$
$/\ bSb$
$/\ a$

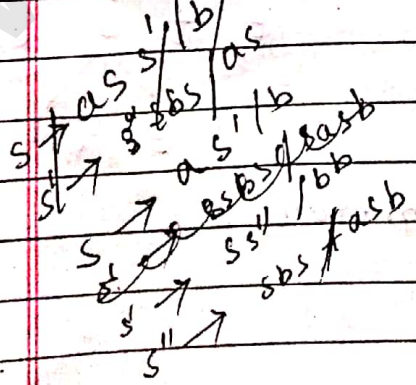
⇒

$S \to bSS' / a$
$S' \to SaaS / SaSb / b$

⇒

$S \to bSS' / a$
$S' \to SaS'' / b$
$S'' \to aS / Sb$

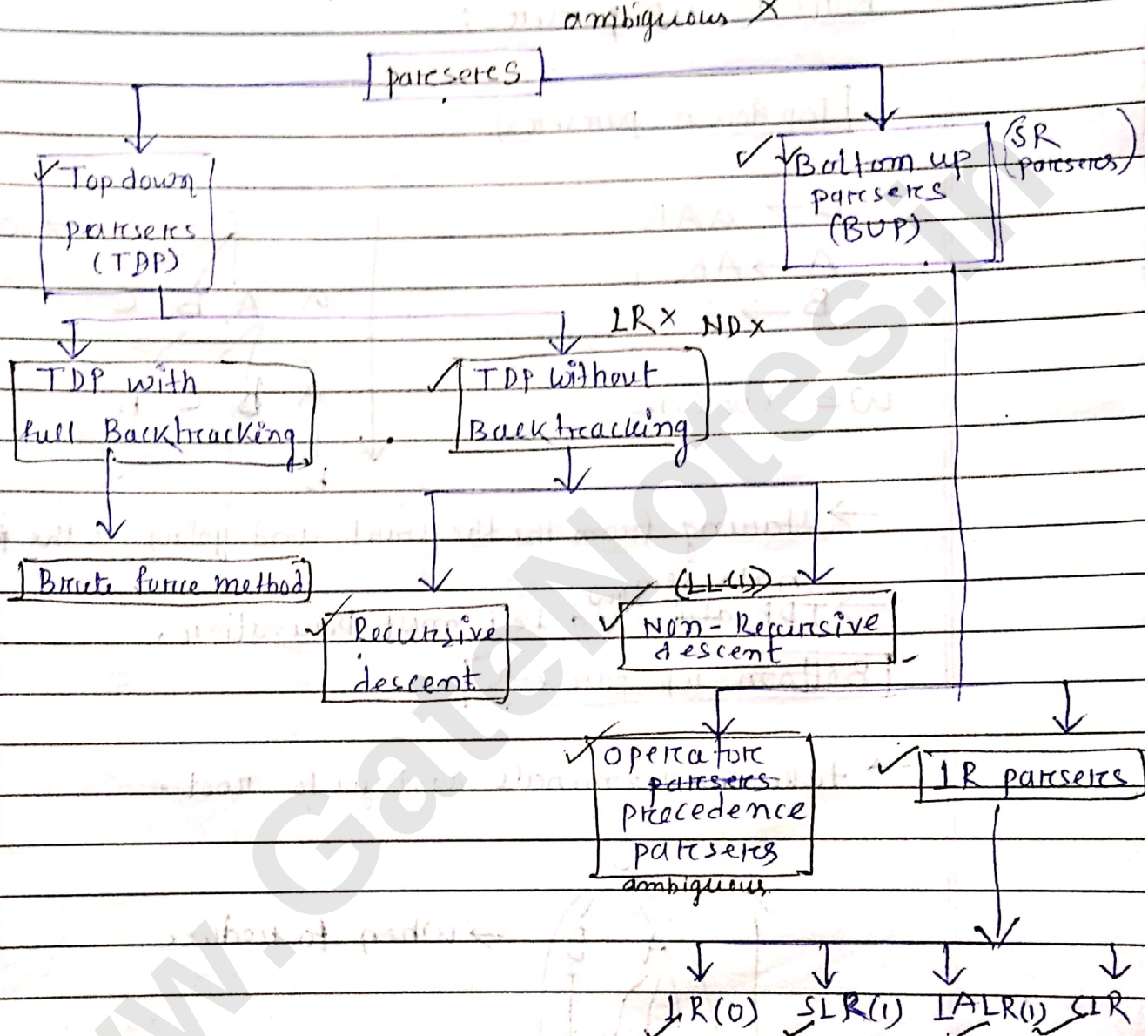
(Deterministic)



PARSERS

(www.gatenotes.in)

- Introduction of parsers : (gatenotes.in)



→ parsers is syntax analyser.

→ SR parsers means shift-reduce parsers.

→ TDP without backtracking not accept - left recursive grammars and also Non-Deterministic grammars.

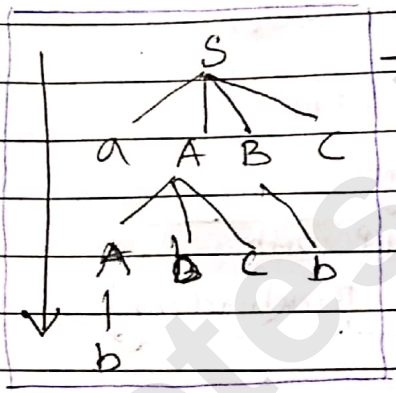
→ parsers not accept ambiguous grammars.
only operator precedence parsers accept ambiguous grammars.

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

Top down parsers

$S \rightarrow aABe$
$A \rightarrow Abc/b$
$B \rightarrow d$

$w = abbcde$



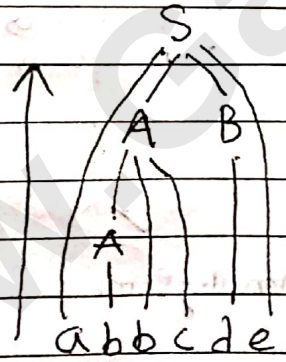
→ what to use.

→ Staring from the the start and going to the ~~bot~~ terminal.

→ TDP ~~also used~~ follow Left most Derivation.

Bottom up parsers

→ take the terminals and go to root.



→ when to reduce

$w = abbcde$.

$S \Rightarrow a\,A\,B\,e$
$\Rightarrow a\,A\,d\,e$
$\Rightarrow a\,A\,b\,c\,d\,e$
$\Rightarrow a\,b\,b\,c\,d\,e$

→ BUp follow Right most Derivation.