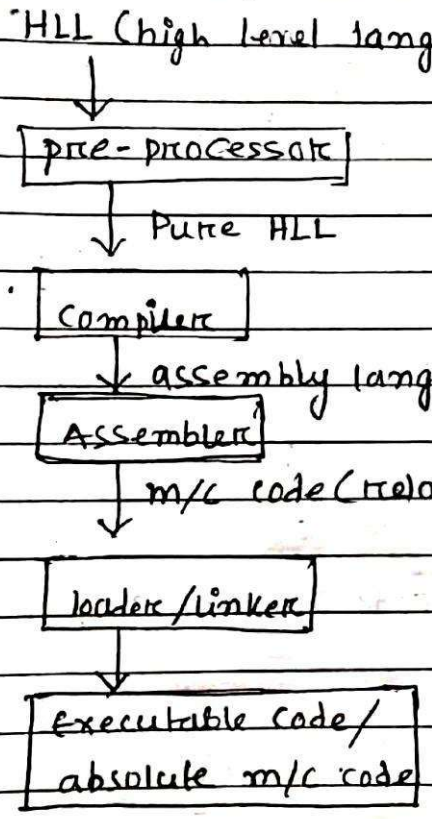
INTRODUCTION

Compiler Design

- **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.



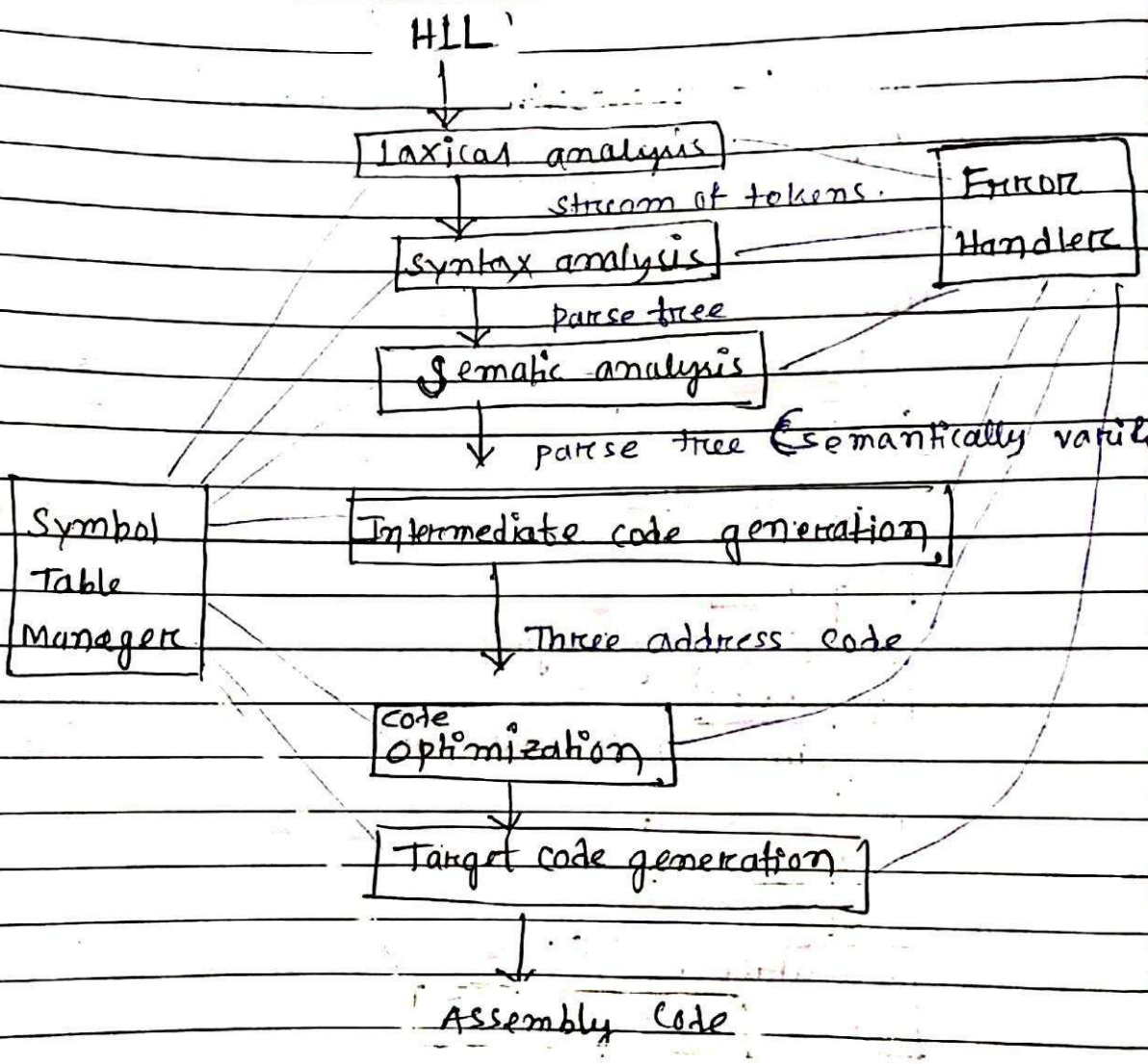
(<#inclu> - file inclusion
<#define> - Macro expansion
→ (If any language contain this type of line it called HLL)

→ 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.

Compilers 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~~ 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

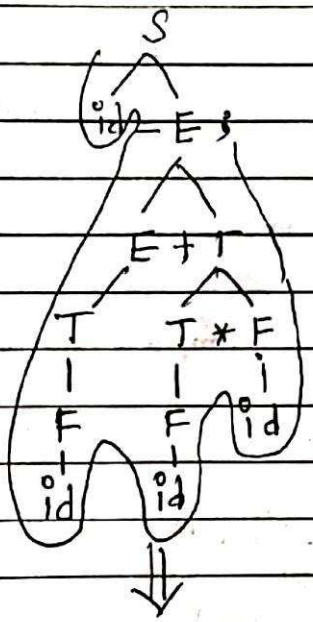
⇓

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

⇓

$t_1 = b * c$

$t_2 = a + t_1$

$x = t_2$

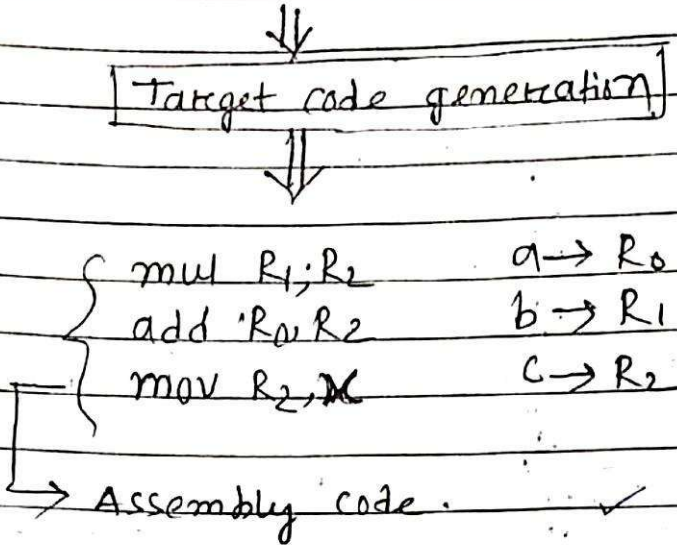
⇓

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 we have tool called 'LANCE'.

• Lexical analyser:

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

(Lexemes, Lexemes)

→ Lexical analyser removing the comments and our White spaces.
→ main function of lexical analyser is converting Lexemes into tokens.
→ Show the errors. (If getting any error).

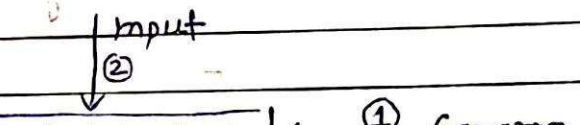
Ex: How many token is there in this particular lines ==

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

② 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 → Terminal
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) (RMD)

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 =

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

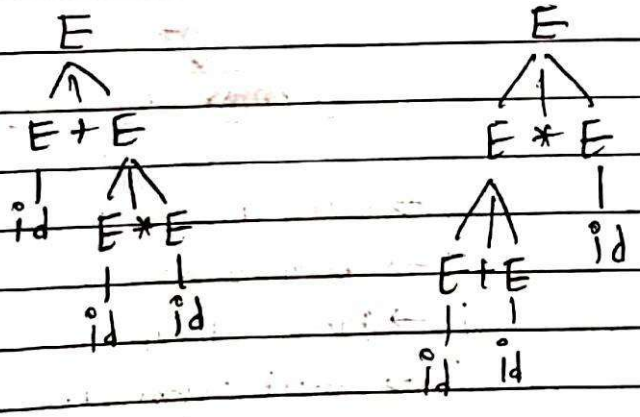
LMD =

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

RMD =

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

- Derived using parse tree -



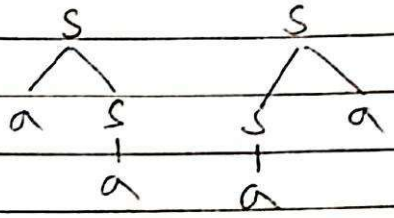
→ To the given grammer and string if we get more than one LMD or RMD (or) parse tree then the grammer is called ambiguous grammar.

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

Find given grammars are ambiguous or not:

① $S \rightarrow aS/Sa/a$

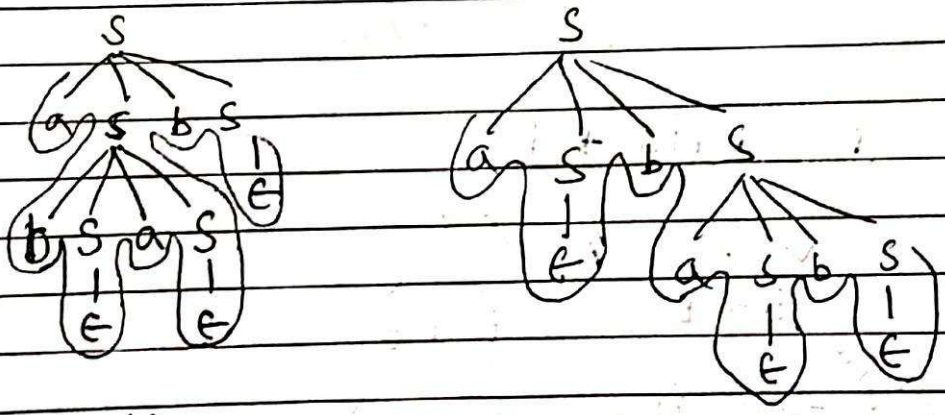
→ $w = aa$



→ This grammar is ambiguous.

② $S \rightarrow aSbS/bSaS/\epsilon$

→ $w = abab$

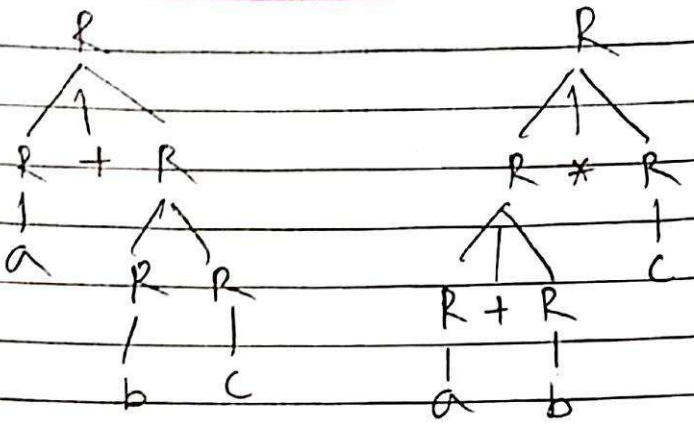


(abab) (abab)

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

③ $R \rightarrow R+R/RR/R^*/a/b/c$

→ $w = a+bc$



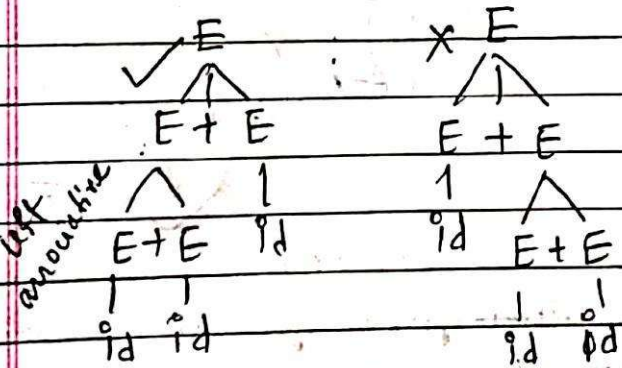
→ this grammar is ambiguous grammar.

• Ambiguous grammar and making them unambiguous:

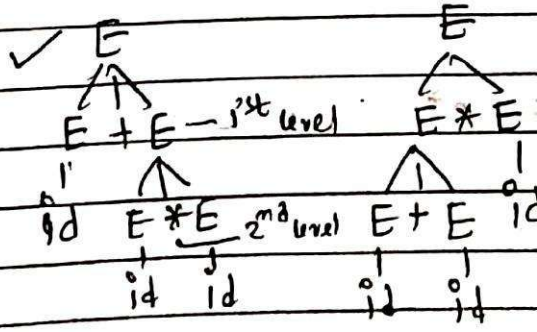
Ex-1

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

① id + id * id

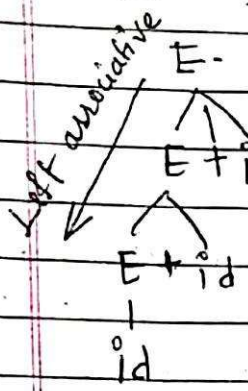


② id + id * id



~~① it's operators are left associative.~~
~~② all operators~~

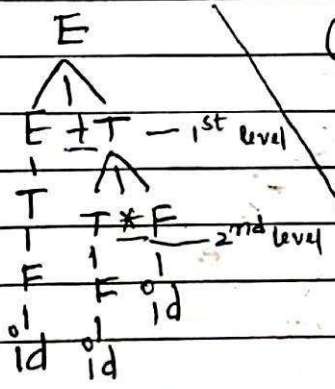
① $E \rightarrow E+id/id$ (This grammar is left recursive)



$((id+id)+id)$

→ 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: $\uparrow > * > +$
($\uparrow$: 2nd level, $*$: 2nd level, $+$: 1st level)

③ $E \rightarrow E+T/T$
$T \rightarrow T*F/F$
$F \rightarrow Gr \uparrow F/Gr$
$Gr \rightarrow id$

here, $+$ and $*$ is left associative,
$\uparrow$ is right associative.

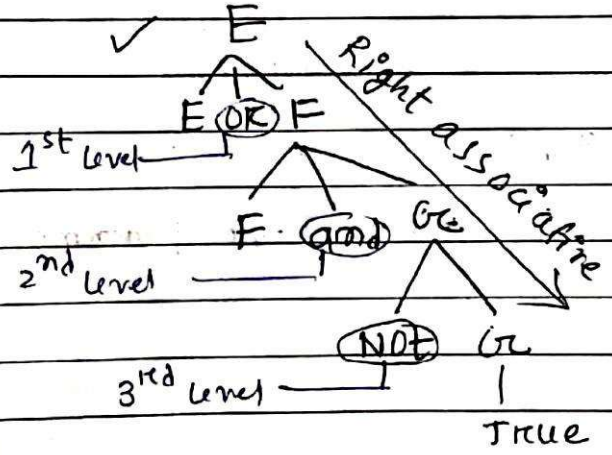
④ boolean expression,

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.

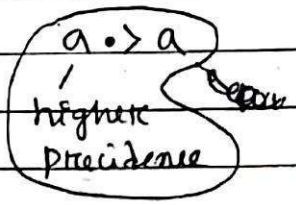
⑥ Given grammar -

A → A $B / B ($, #, @ are operators)
B → B # C / C
C → C @ D / D (here, $, #, @ are left recursive)
D → d

→ $ ·> $

·>

@ ·> #



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

⑦ Given grammar -

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

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

→ ~~And~~ This grammar is ambiguous.

* ≐ + (equal).

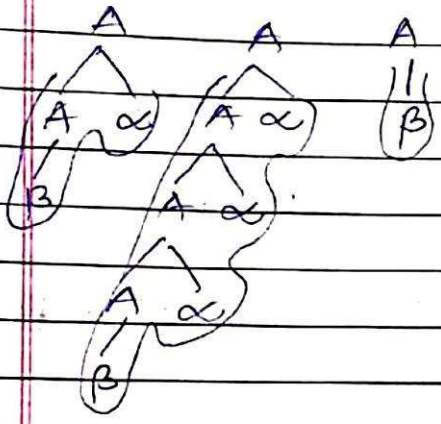
(high precidence) * ·> *

+ <· + (high precidence)

Recursion

Left Recursion

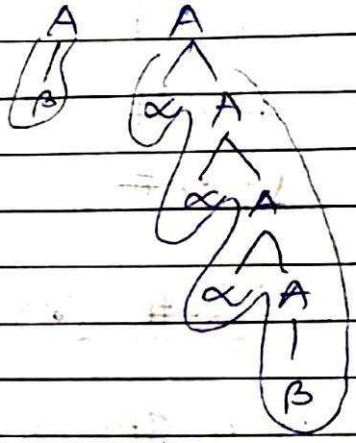
$A \rightarrow A\alpha / \beta$



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

Right Recursion

$A \rightarrow \alpha A / \beta$



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

- $\beta\alpha^*$

$A \rightarrow \beta\alpha^*$.

** $A \rightarrow \beta A'$
$A' \rightarrow \epsilon / \alpha A'$ $\Leftrightarrow$ $A \rightarrow A\alpha / \beta$.

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

Example — eliminate the left recursive,

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

⇕

→ $E \rightarrow TE'$
$E' \rightarrow \epsilon / +TE'$ → This is equivalent grammar and not left recursive grammar.

Example- Eliminate left recursion from this grammar -

① $S \rightarrow S0S1S/01$
$(A) \quad (A)(\alpha) \quad (B)$

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

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

② $S \rightarrow (L)/x.$

$L \rightarrow L,S/S.$
$(A) \quad (A)(\alpha) \quad (\beta)$

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

$L \rightarrow SL'$

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

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

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

$A' \rightarrow \alpha_1A'/\alpha_2A'/\alpha_3A'/\cdots\epsilon.$

Grammar
- Ambiguous ✗
- Unambiguous ✓

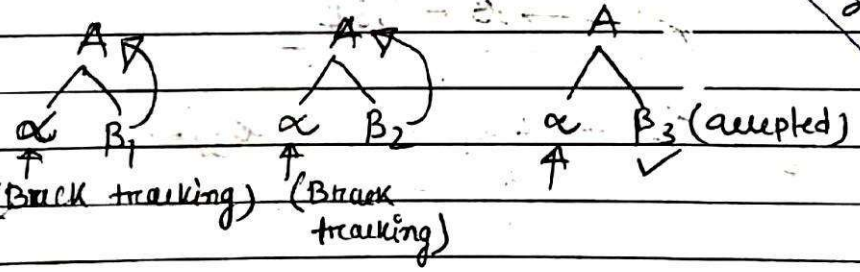
Grammar
- Left recursive ✗
- Right recursive ✓

Grammar
- Deterministic ✓
- Non-Deterministic ✗

• Non-Deterministic :

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

$\alpha\beta_3$

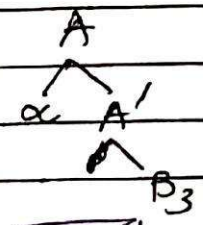


• Left factoring procedure are eliminating non-determinism :

$A \rightarrow \alpha A'$

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

generate $\alpha\beta_3$



$\alpha\beta_3$

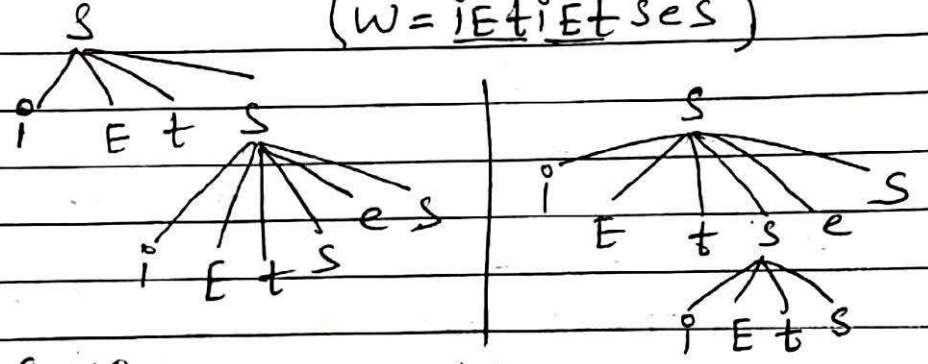
Example –

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

left factoring

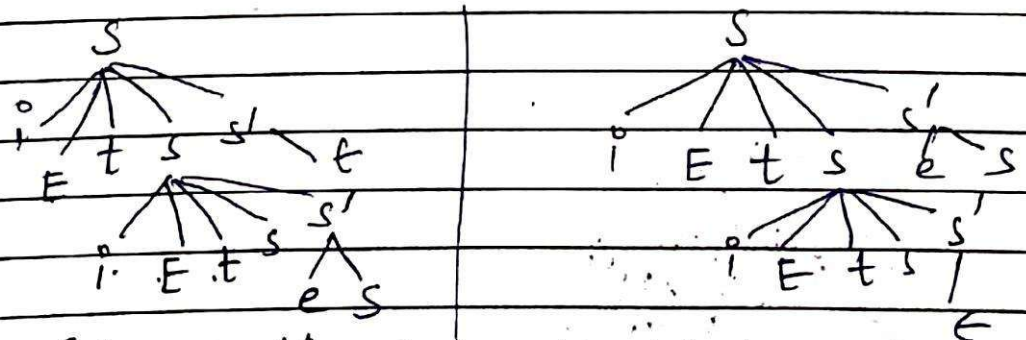
(II) $S \rightarrow iEtSS' / a$
$S' \rightarrow e/eS$
$E \rightarrow b$ → (Deterministic)

(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)

$S \to @SSbS$
$/ @S@SbSb$
$/ @bb$
$/ b$

$\Rightarrow$

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

$\Rightarrow$

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

(Deterministic)

Non-Deterministic

$S \to bSS' / a$
$S' \to @as / @sb / b$

$\Rightarrow$

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

(Deterministic)