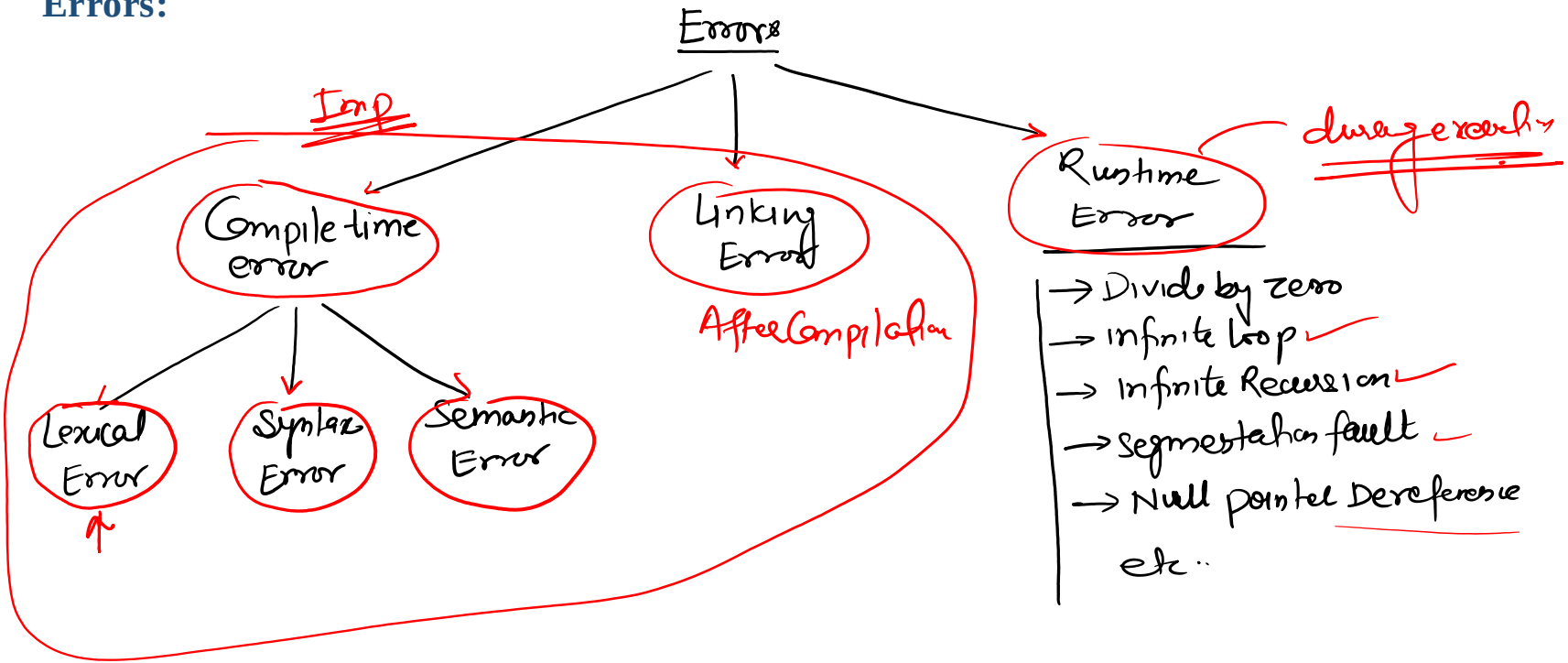
Today Class Topics

Syntax Analysis

-Errors
-Context free grammar(CFG)
-Parse tree or derivation tree
-Ambiguous and Unambiguous grammar
-Left recursion and how to remove left recursion
-Left factoring and left factored grammar

Errors:


Errors
Compile time error
Linking Error
Runtime Error
Lexical Error
Syntax Error
Semantic Error
→ Divide by zero
→ Infinite loop
→ Infinite Recursion
→ Segmentation fault
→ Null pointer Dereference
etc..

Lexical Error: if the Scanned group of characters are not matching with any of rules (pattern) of the Tokens

Lexical Analyzer produce an error.

- Unterminated Comment:
  Ex: /* Comment , /* Comment */ ✓ (lexical)
- Unmatched string: "abcd, abcd", "abcd" (lexical error)
- Invalid Identifier: Ex: int 1a, 2a; (lexical error)
- Appearance of illegal characters:
  Ex: int a = $10; (lexical error)

```
Void main()
{ int a, b, c;
  c = a + b;
}@
```

- Exceeding the length of identifier: 32 characters

$L(L+D)^*$

$(L,-)(L+D+-)^*$

Ex: printf("Delhi is a beautiful city"); LE

Ex: int a = 10.26;
a = a + 5;
b = b * a; /* Multiply b with a */ ⇒ lexical error
printf("%d", b);

Ex3 int a = 09; // octal no. (0 — 7) Lexical error
(09: octal)

Ex4 int a = 0b12; // (0 or 1) Allowed ⇒ lexical error
(0b: Binary No.)

Ex5 int a = 0x9AGH; // not Allowed ⇒ lexical error
(0x: Hexadecimal No.)

(0 — 9 A B C D E F)
0 — 9, 10, 11, 12, 13, 14, 15
A B C D E F

Syntax Error: (Pattern)

1) missing (, &, ; etc
2) missing operator ab+c;
3) missspelled keyword Ex:- fro(i=0; i<n; i++)

ab + c

int ab;

int a, b;

op1 @ op2

Syntax error

(or)

for (i=0, i<n, i++) Syntax error

a = b + cd *

Ex: While(1 Syntax error
{ printf("Hi");
}

Ex:- ef();

Ex:- while();

Ex:- for(, ,);

Syntax error

for(; ;);

Ex:- iit x, y; Syntax error

Ex:- main()
{ { }
{ }
{ }

Syntax error

Ex:-
```
void main()
{
  int x, y;
  x = 2;
  y = 3;
}
```

Ex
```
a = b + cd *;
```

(Semantic Analysis)

Semantic Error : Type mismatch

1) ~~In~~Compatible type of operands
2) ~~Undeclared variable~~
3) Unmatch Actual & formal parameter

```
int a, b, c;
```

Symbol Table

SN	Vname	Vartype	Value
1	a	int	
2	b	int	
3	c	int	

```
c = a + b
```

Ex-1

```
void main()
{
    int a, c;
    float b;
    c = a + b;
}
```

int, float

Semantic error

Ex-2

```
int Sum;
Shm = 15;
```

SN	Vname	Vartype
1	Sum	int

3

```
void main()
{
    int x;
    x = y;
}
```

4

```
while(1);
```

Linking errors

```
While(1);
```

function calling

Ex'

```
int Sum;
Shm = 15;
```

Ex:

Consider line number 3 of the following C-program.

```
int main() {                    /*Line 1 */
    int I, N;                   /*Line 2 */
    fro (I=0, I<N, I++);            /*Line 3 */
}
```

Identify the compiler's response about this line while creating the object module:

A. No compilation error
B. Only a lexical error
C. Only syntactic errors
D. Both lexical and syntactic errors

n = 2

0 < n

fro(i=0, i<n, i++);

⇒ fro(0, 1, 0); linking

⇒ fro(i=0, i<n, i++);

fro → not match

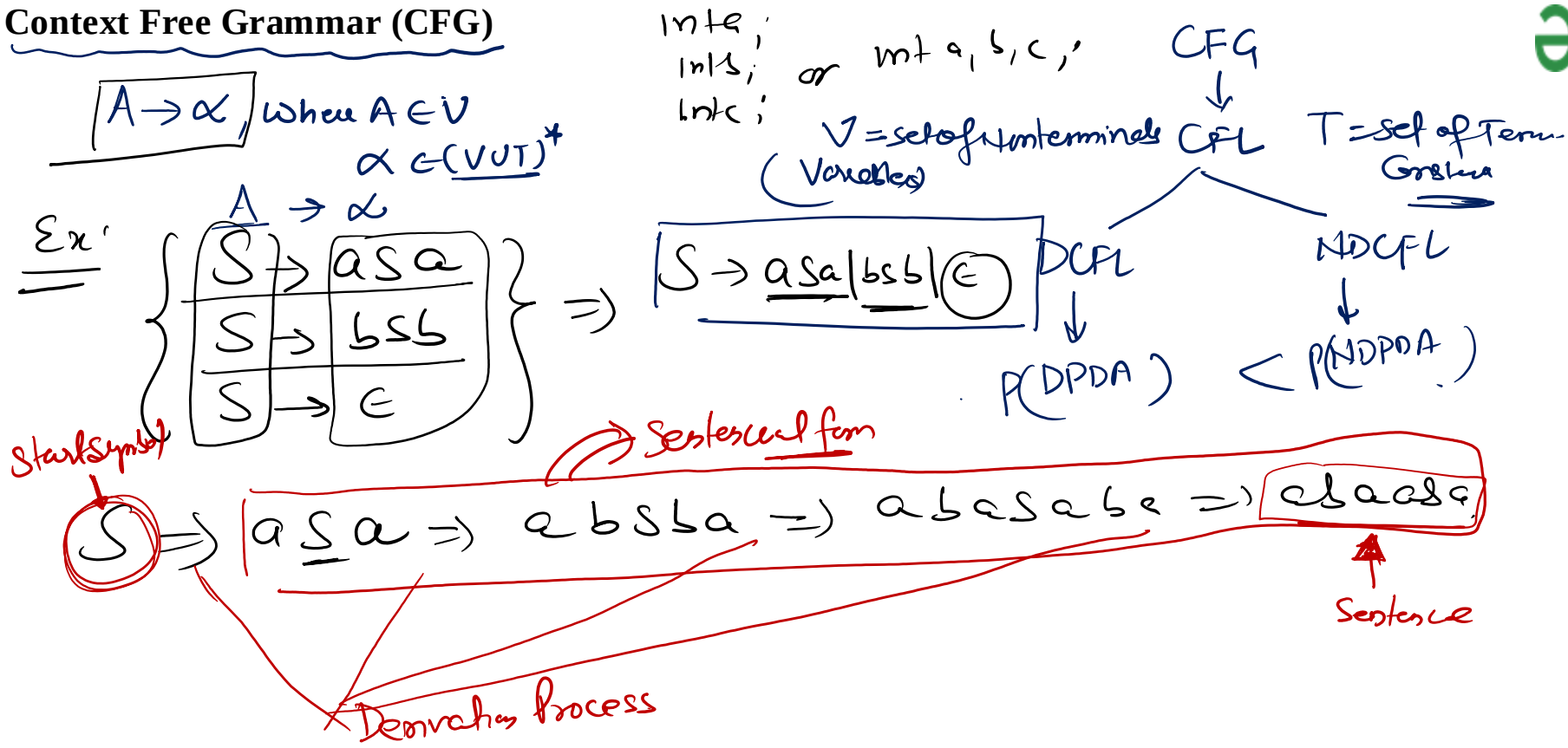
i<n, i++ → not match

fro(i=0; i<n; i++); // Syntax error

for(i=0, i<n, i++); // Syntax error

for(i=0; i<n; i++); // No Syntax error

Context Free Grammar (CFG)



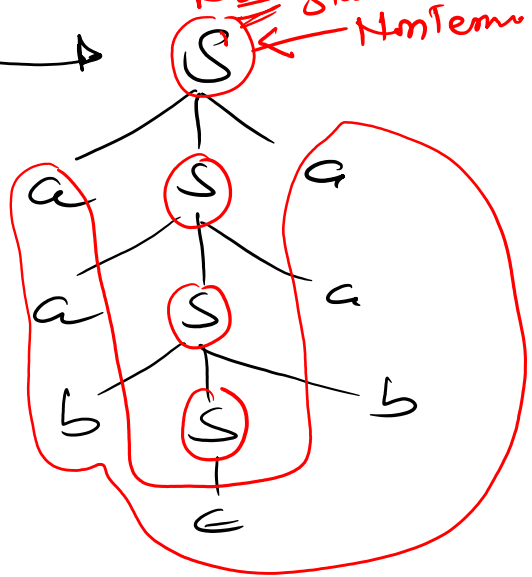
Derivation Tree or Parse Tree : Graphical Representation of Derivation Process

$S \rightarrow aSa \mid bSb \mid \epsilon$

Derivation Process:

$S \Rightarrow aSa \Rightarrow aaSaa \Rightarrow aabSbaa \Rightarrow aabbaa$

RSDT Start Sym = NonTerm



internal node = Non terminal

leaf node = Terminal symbol

Sentence

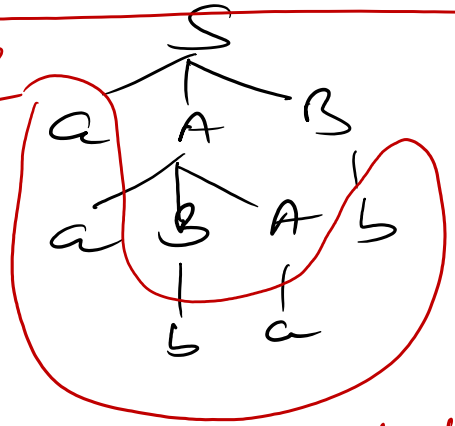
Ex:- G: $S \rightarrow aAB \mid a$, $A \rightarrow aBA \mid a$, $B \rightarrow b$

Construct Parse tree for string | aabaab |

Soln: LMD

$S \Rightarrow aAB \Rightarrow aaBAB \Rightarrow aabAB \Rightarrow aabaB \Rightarrow aabab$

Parse tree



aabab

Derivation Process

LMD

RMD

$S \overset{*}{\Rightarrow} \omega$ (String)

Derivation process

There are two types of derivation process,

1. Left most derivation (LMD)
2. Right most derivation (RMD)

Left most derivation (LMD):

A derivation $S \overset{*}{\Rightarrow} w$ is called a leftmost derivation if we apply a production only to the leftmost variable at every step.

Right most derivation (RMD):

A derivation $S \overset{*}{\Rightarrow} w$ is called a rightmost derivation if we apply a production to the right most variable at every step.

Ex:- $G = \{ S \to AB, A \to aA \mid a, B \to bB \mid b \}$

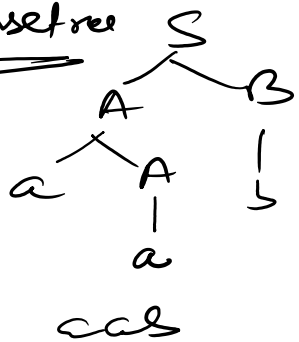
LMD

$\Rightarrow S$
$\Rightarrow AB$
$\Rightarrow aAB$
$\Rightarrow aaB$
$\Rightarrow aab$

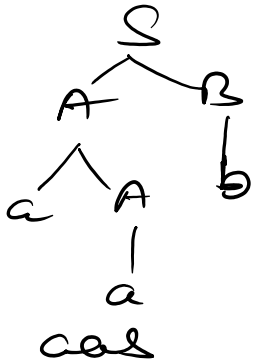
RMD

$\Rightarrow S$
$\Rightarrow AB$
$\Rightarrow Ab$
$\Rightarrow aAb$
$\Rightarrow aab$

Parse tree



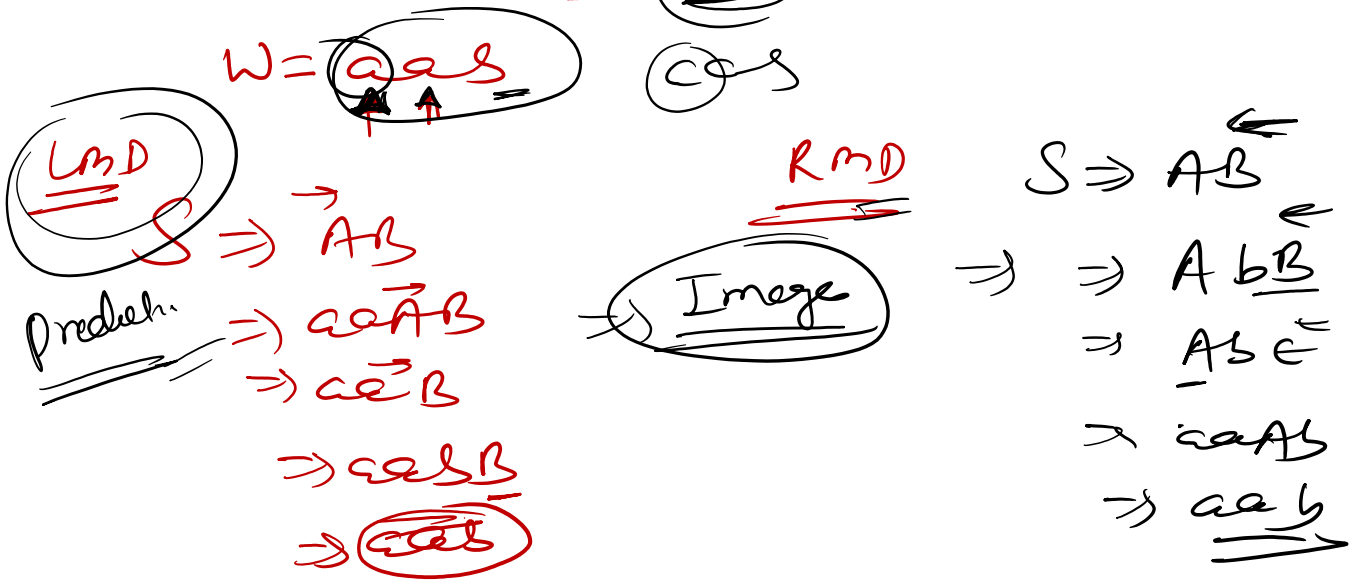
$\cong$



NOTE: LMD & RMD are different but their derivation tree need not be different

Example: Consider a grammar G for the language, L = $\{a^{2n}b^m, m, n \geq 0\}$

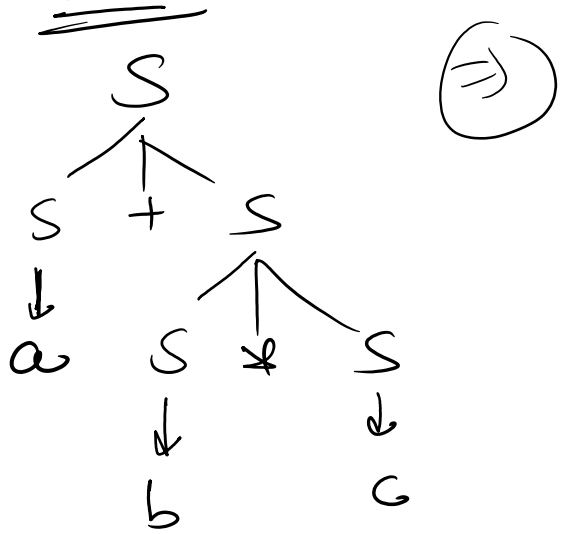
G: S → AB, A → aaA | ε, B → bB | ε, find the LMD and RMD for string w = aab



Example: G: S → S+S | S*S | a | b | c, find the LMD and RMD for string w = a + b * c

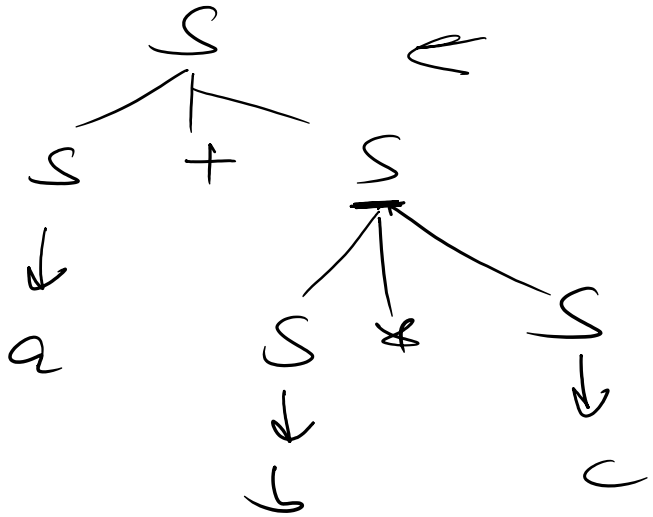
LMDT

a + b * c



⇒ a + b * c

RMDT



a + b * c

Ex: $S \to a \mid AB$, $A \to Ab \mid \in$, $B \to a$

Starting Symbol is S

NOTE: (1) LMD and RMD need not be same

(2) No. of LMD = No. of RMD = No. of parse.

No. of derivations = No. of LMD = No. of RMD = No. of parse tree

$\omega = a$ ✓

LMD

(1) $S \Rightarrow a$

(2) $S \Rightarrow AB \Rightarrow B \Rightarrow a$

RMD

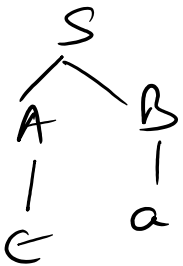
(1) $S \Rightarrow a$

(2) $S \Rightarrow AB \Rightarrow Aa \Rightarrow a$

Parse tree

(1)

(2)



Ambiguous Grammar

A grammar G, is ambiguous grammar if ∃w ∈ L (G), such that w has > one Parse tree either using 2 LMD or 2 RMD i.e., a grammar G is ambiguous if there is more than one Parse Tree or LMD / RMD for a string w ∈ L (G).

Example: G: S → S+S | S*S | a | b | c, Grammar G is ambiguous because there exist two different LMD for a string w = a + b * c

On Ambiguous

G is On Ambiguous if ∀w ∈ L(G)

G ⇒ L(G) = { ——— }

∀w ∈ L(G), w = 1 PT
= 1 LMD
= 1 RMD

w = 1 PT
or
1 DT
or
1 LMD
or
1 RMD

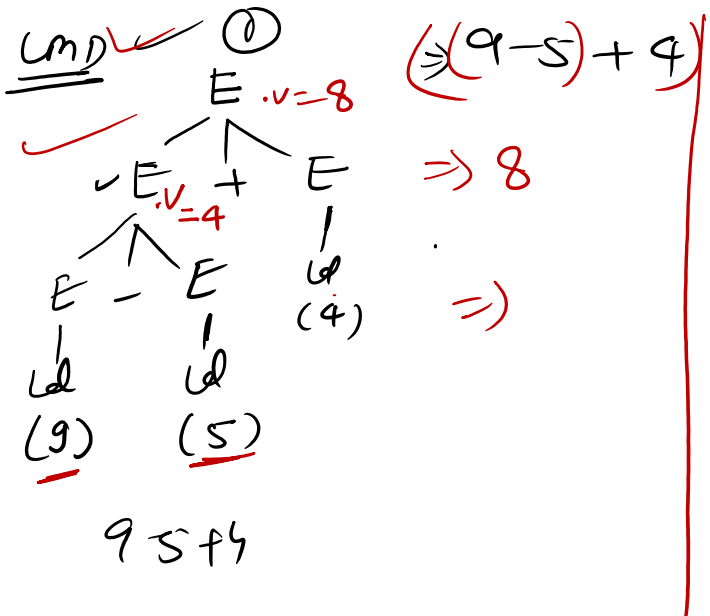
NOTE:

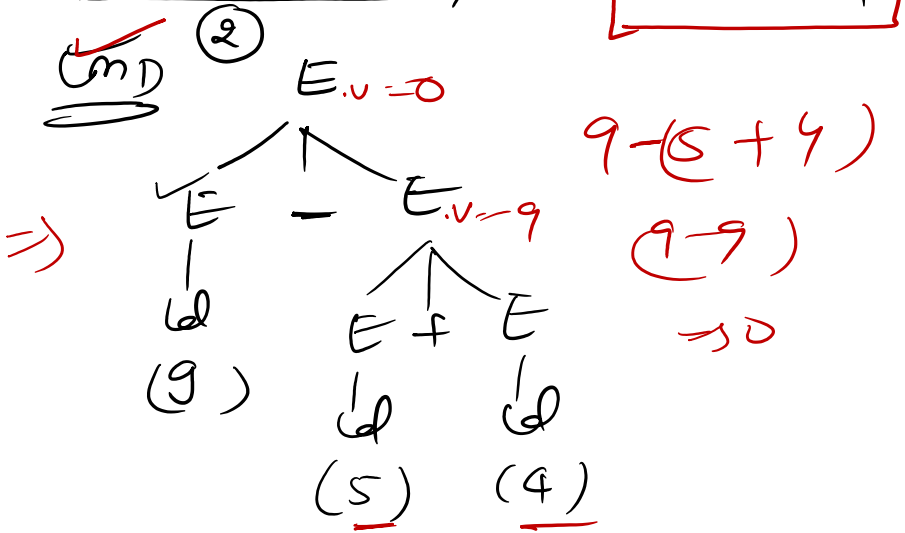
1- A grammar G is unambiguous if there exist exactly one parse tree or LMD / RMD for all the string, w ∈ L (G)

2- If a grammar G is ambiguous, it doesn't mean language (L) is ambiguous.

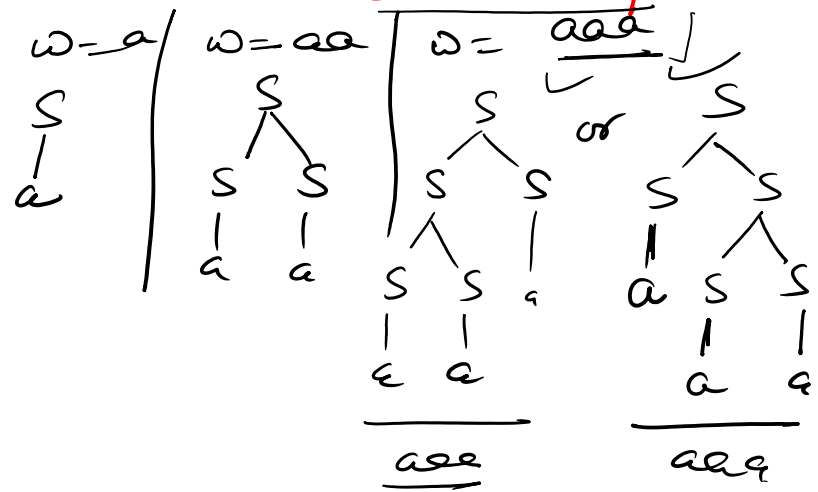
Ambiguity

Example: State whether given grammar is ambiguous or not. G: E → E+E | E-E | id , ω = 9-5+4





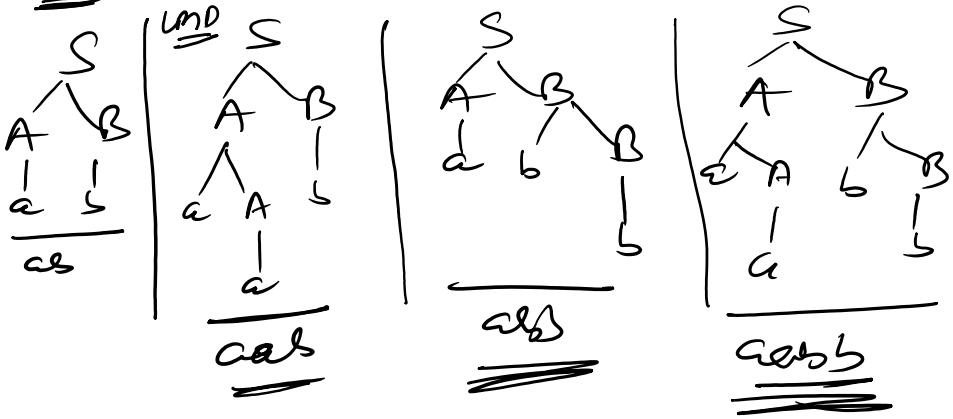
Ex 1) $G = \{ S \rightarrow SS \mid a \}$



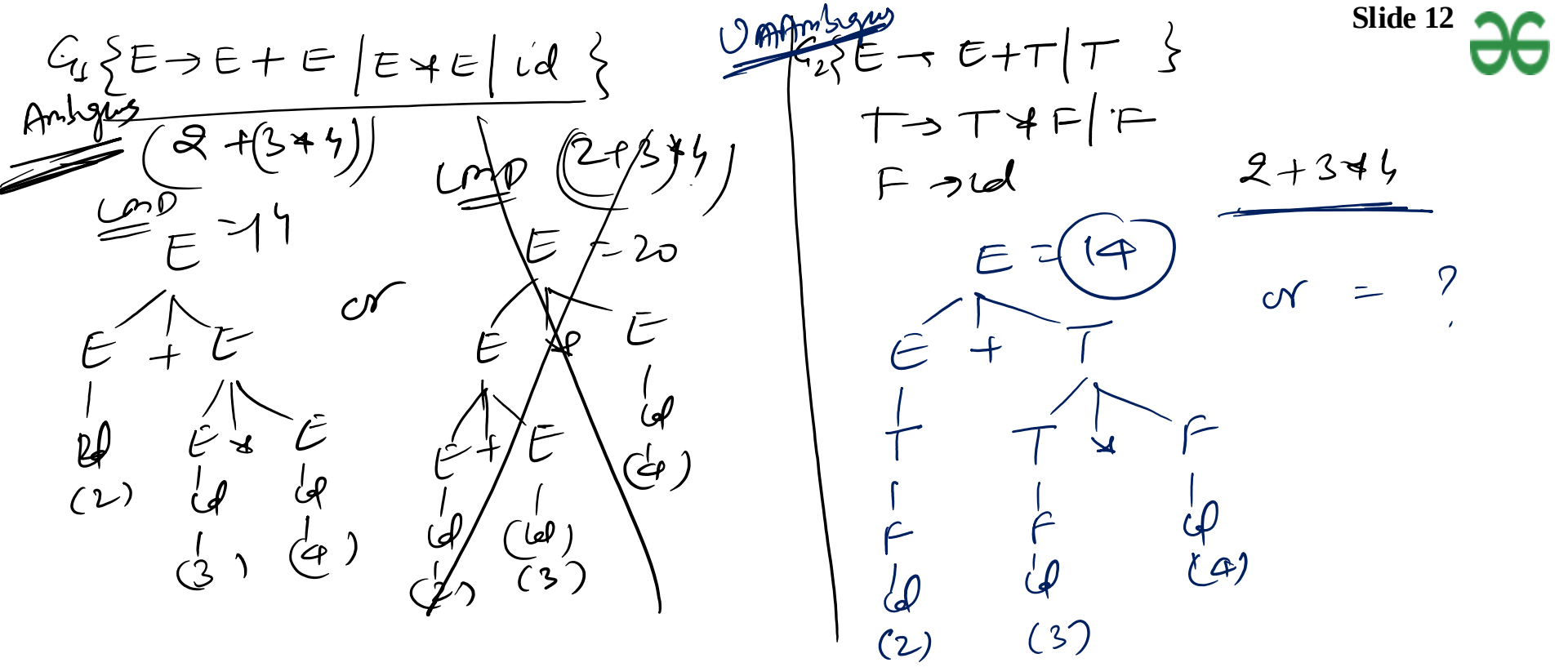
Ambiguous Grammar

Ex 2

$G = \{ S \rightarrow AB, A \rightarrow aA \mid a, B \rightarrow bB \mid b \}$

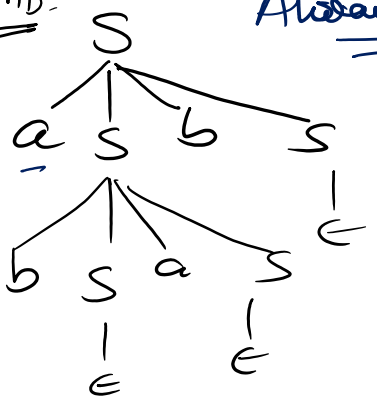


On Ambiguous


$G_1 \{E \to E+E \mid E*E \mid id\}$
Ambiguous
(2+(3*4))
LMD
E = 14
E + E
id E * E
(2) id E
id id
(3) (4)
or
LMD (2+3*4)
E = 20
E * E
E + E id
id id
(2) (3) (4)
Unambiguous
$G_2 \{E \to E+T \mid T\}$
$T \to T*F \mid F$
$F \to id$
2+3*4
E = 14
E + T
T * F
F
id
(2)
(3)
(4)
or = ?

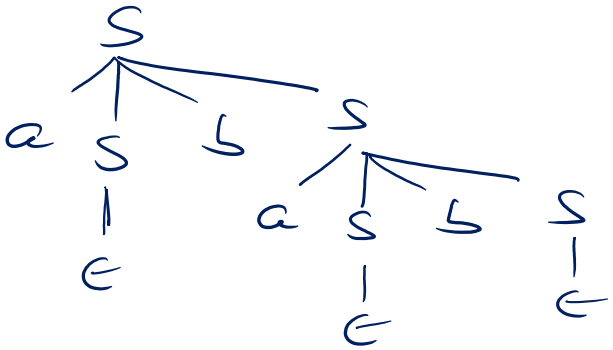
$G = \{ S \to aSbS \mid bSaS \mid \epsilon \}$ Check Ambiguous ?

Sol^{n}: LMD:



$\Rightarrow aSaS$

Always Ambiguous RMD:

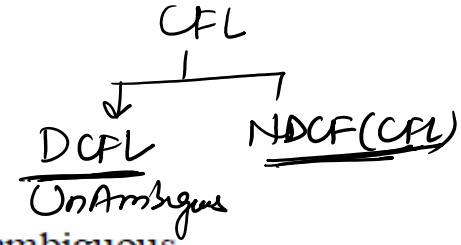


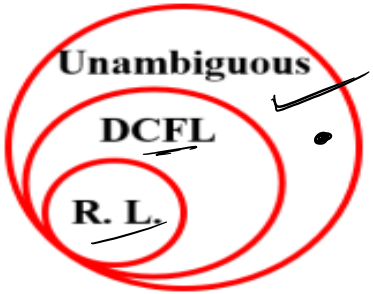
$a \cdot \notin b\,a \notin b \cdot \in$

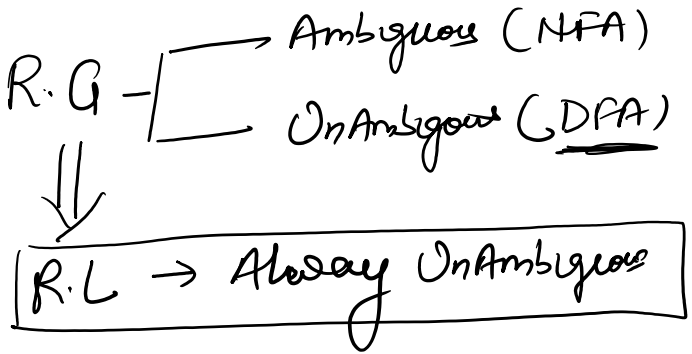
$\Rightarrow aS\,aS$

NOTE:

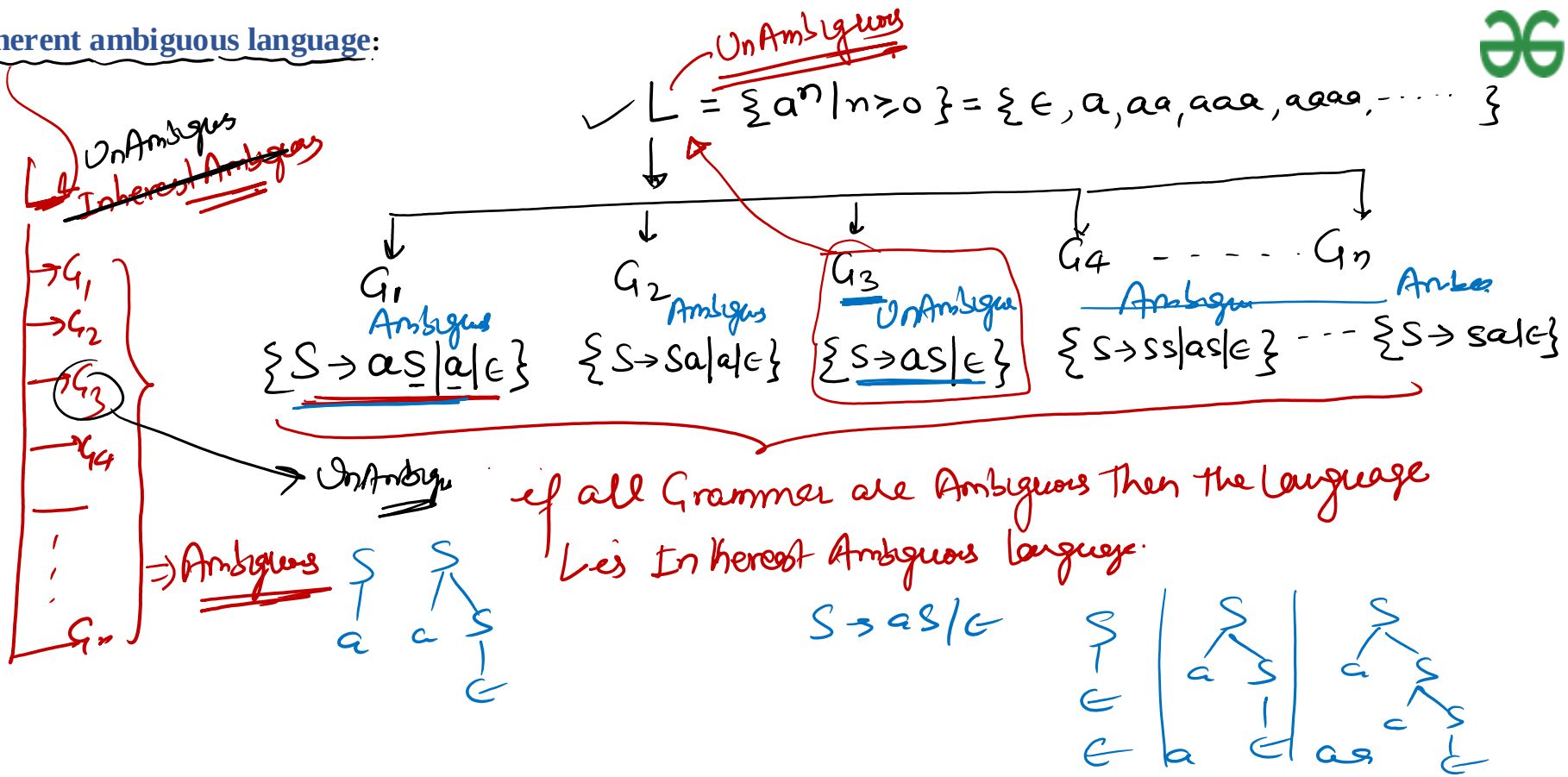
- Regular Grammar:
  - Ambiguous (NFA)
  - Unambiguous (DFA)
- Regular language is always unambiguous.
- DCFL is always unambiguous language
- Ambiguity starts from CFL



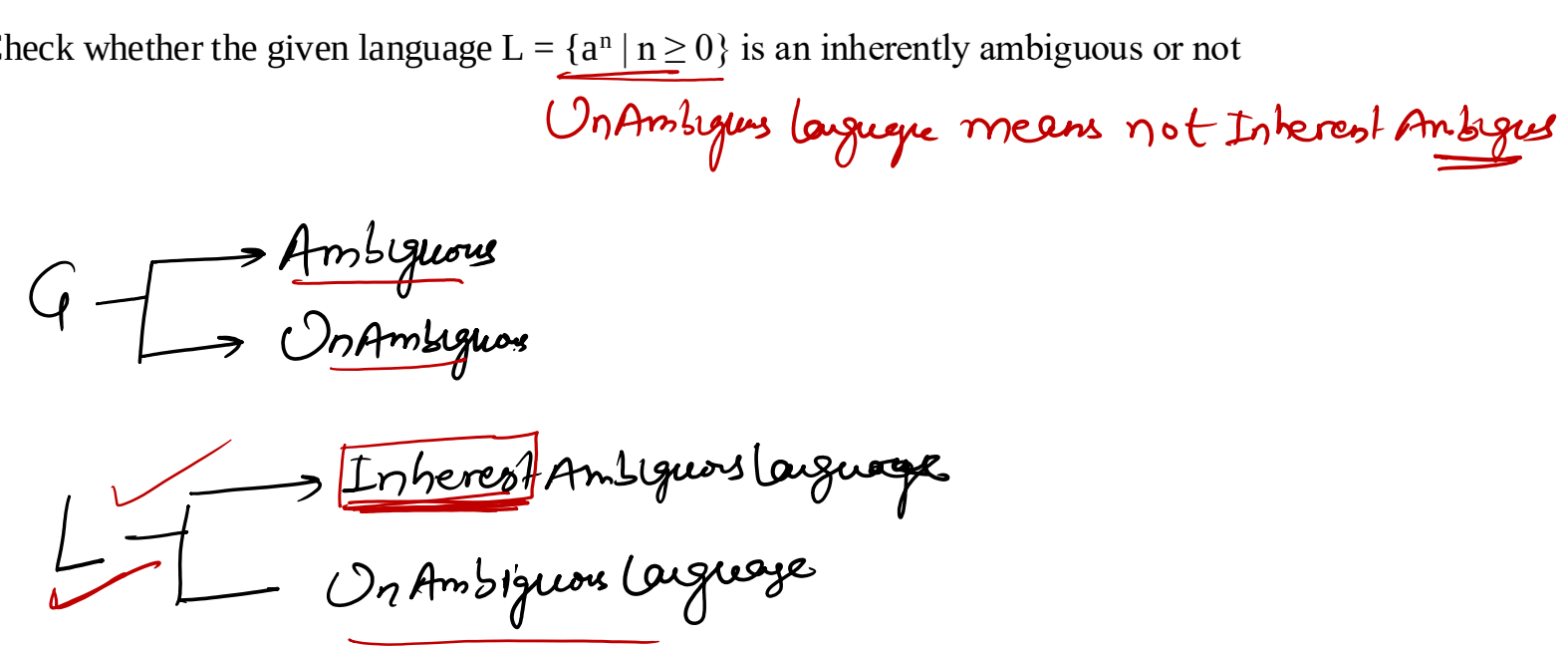




Inherent ambiguous language:



Example: Check whether the given language L = $\{a^n \mid n \geq 0\}$ is an inherently ambiguous or not



Example: Given language L = $\{a^m b^n c^k \mid m, n, k \geq 1$, either m = n or n = k $\}$ is Inherent Ambiguous?

$\Rightarrow$ Inherent Ambiguity

$L = \underbrace{\{a^m b^m c^k \mid m,k \geq 1\}}_{L_1} \cup \underbrace{\{a^m b^n c^n \mid m,n \geq 1\}}_{L_2}$

L_1: A . B L_2: C . D

G:

$$\begin{cases} S \to AB \mid CD \\ A \to aAb \mid ab \\ B \to cB \mid c \\ C \to aC \mid a \\ D \to bDc \mid bc \end{cases}$$

$L_1 = \{a^m b^m c^k \mid m,k \geq 1\} = \{$

$L_2 = \{a^m b^n c^n \mid m,n \geq 1\}$

$L_1 \cap L_2 = \{a^i b^i c^i \mid i \geq 1\}$

for this type string $\Rightarrow$ $a^n b^n c^n$

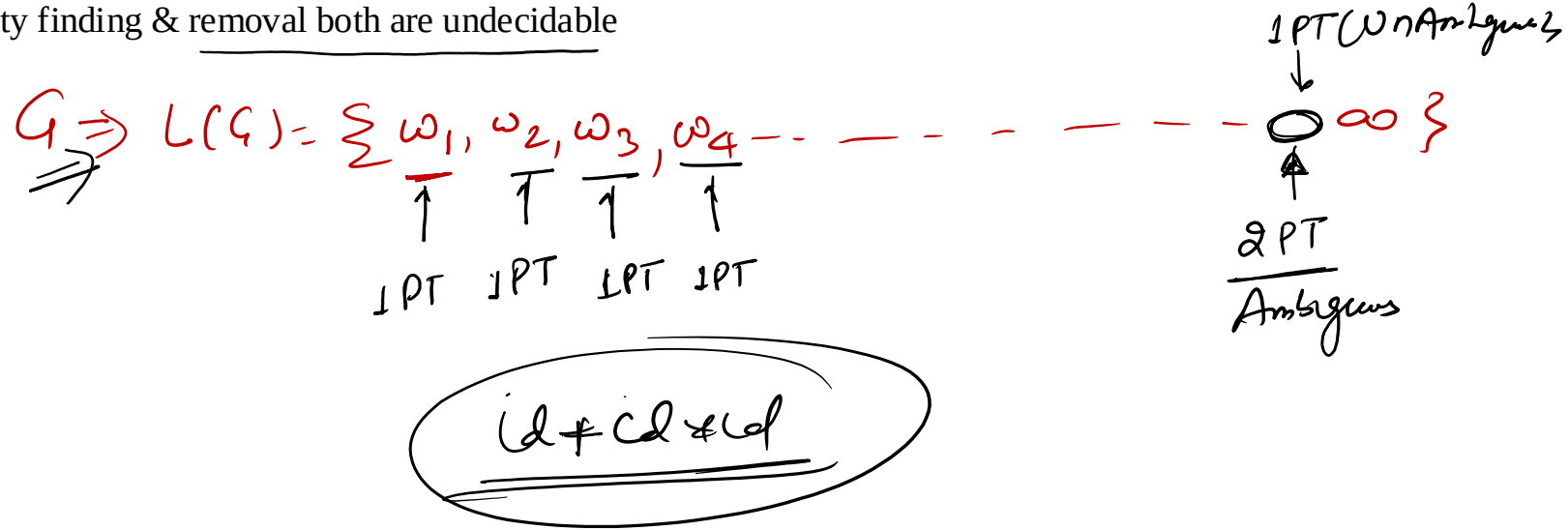
$\downarrow$

$\{$no. of a = no of b$\}$ AND $\{$No of b = No of c$\}$

Note: If grammar is ambiguous then it is not suitable for any kind of parsing technique except backtracking and operator precedence parsing

Remove Ambiguity:

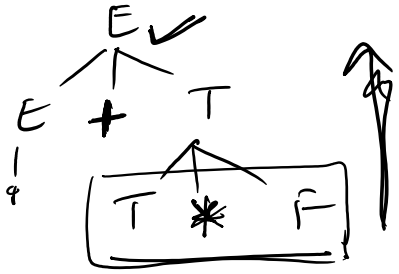
It is not always possible to convert an ambiguous grammar into an unambiguous grammar because ambiguity finding & removal both are undecidable



Remove Ambiguity by precedence & associativity rules:

An ambiguous grammar may be converted into an unambiguous grammar by implementing:

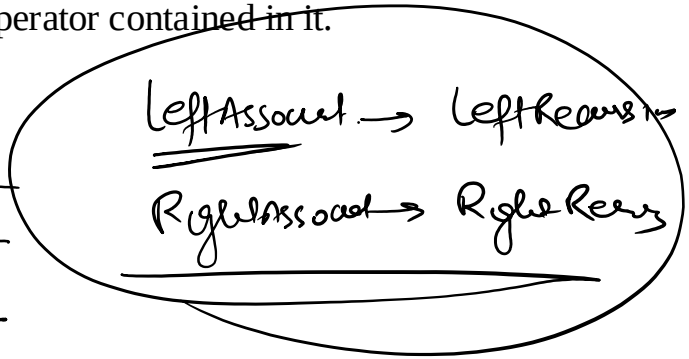
- Precedence Constraints
- Associativity Constraints



These constraints are implemented using the following rules:

Rule 1: The precedence constraint is implemented using the following rules:

- The level at which the production is present defines the priority of the operator contained in it.
- The top level of the production, the lower the priority of the operator.
- The bottom level of production, the higher the priority of the operator.

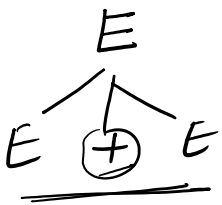


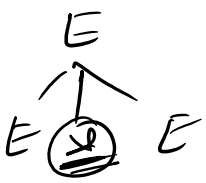
Rule 2: The associativity constraint is implemented using the following rule:

- If the operator is left associative, induce left recursion in its production.

If the operator is right associative, induce right recursion in its production

Example: G: E → E+E | E*E | id. Ambiguous grammar converts into unambiguous grammar





id+id*id

E → E+T | T
T → T*F | F
F → id

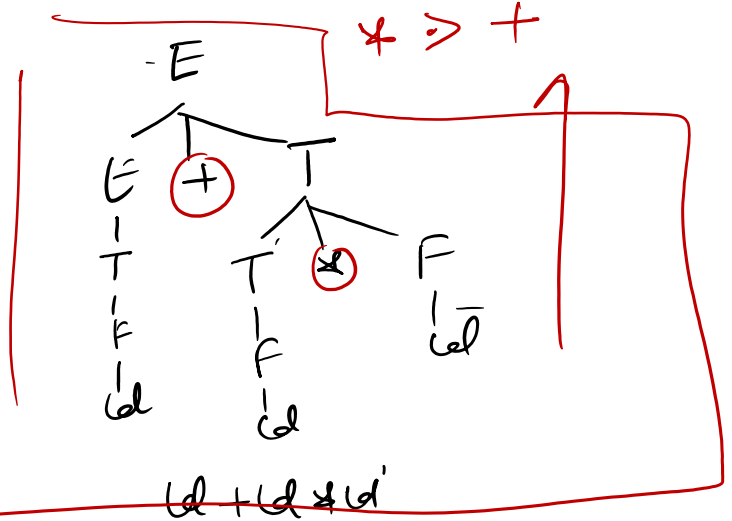
Unambiguous

X

* > +

[+ > +] leftAsso

[* > *] leftAsso



Example: Convert the following ambiguous grammar into unambiguous grammar.

G: E → E+E | E*E | E^E | id

E → E+E

E → E*E

E → E^E

E → id

Unambiguous Grammar

Apply Rule 1 & 2

Starting Symbol, left Assoc

E → E + T | T (+) left associative

T → T * F | F (*) left associative

F → G ↑ F | G (^) Right associative

G → id

The priority order: **id** .> ↑ .> * .> +

↑ .> * .> +

Associativity: + and * are left associative and ^ operator is right associative

↑ .> * .> +

+ .> +

* .> *

↑ <. ↑

Example: Find the precedence and associativity

i)

G1: A → A $ B | B
B → B # C | C
C → D @ C | D
D → d

ii)

G2: E → E * F | F + E | F
F → F − F | id

⇒

Ambiguous Grammar

@ > # > $

@ < @

>

$ > $

① − > [* = +]

② * > *

③ + < +

④ − X −

Q. Consider the grammar defined by the following production rules, with two operators * and +

```
S --> T * P
T --> U | T * U
P --> Q + P | Q
Q --> Id
U --> Id
```

+ > *

+ = Right Associative

* = Left Associative

Which one of the following is TRUE?

(A) + is left associative, while * is right associative

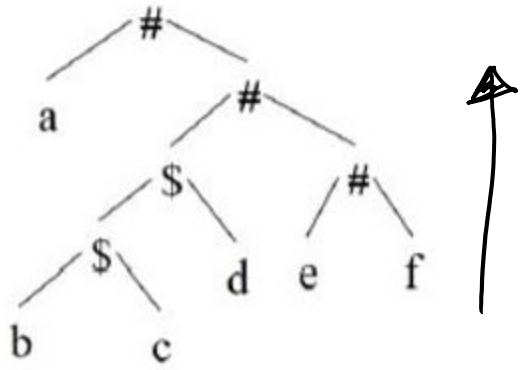
(B) + is right associative, while * is left associative

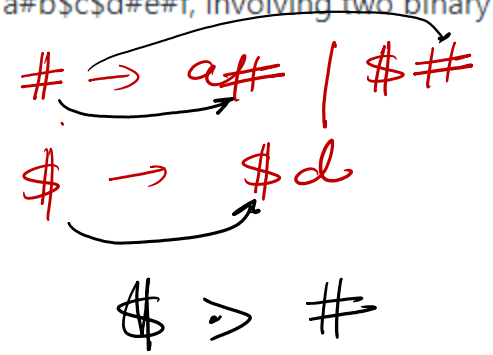
(C) Both + and * are right associative

(D) Both + and * are left associative

(Compiler Design)

Ques. Consider the following parse tree for the expression a#bcd#e#f, involving two binary operators $ and #





#.

Which one of the following is correct for the given parse tree?

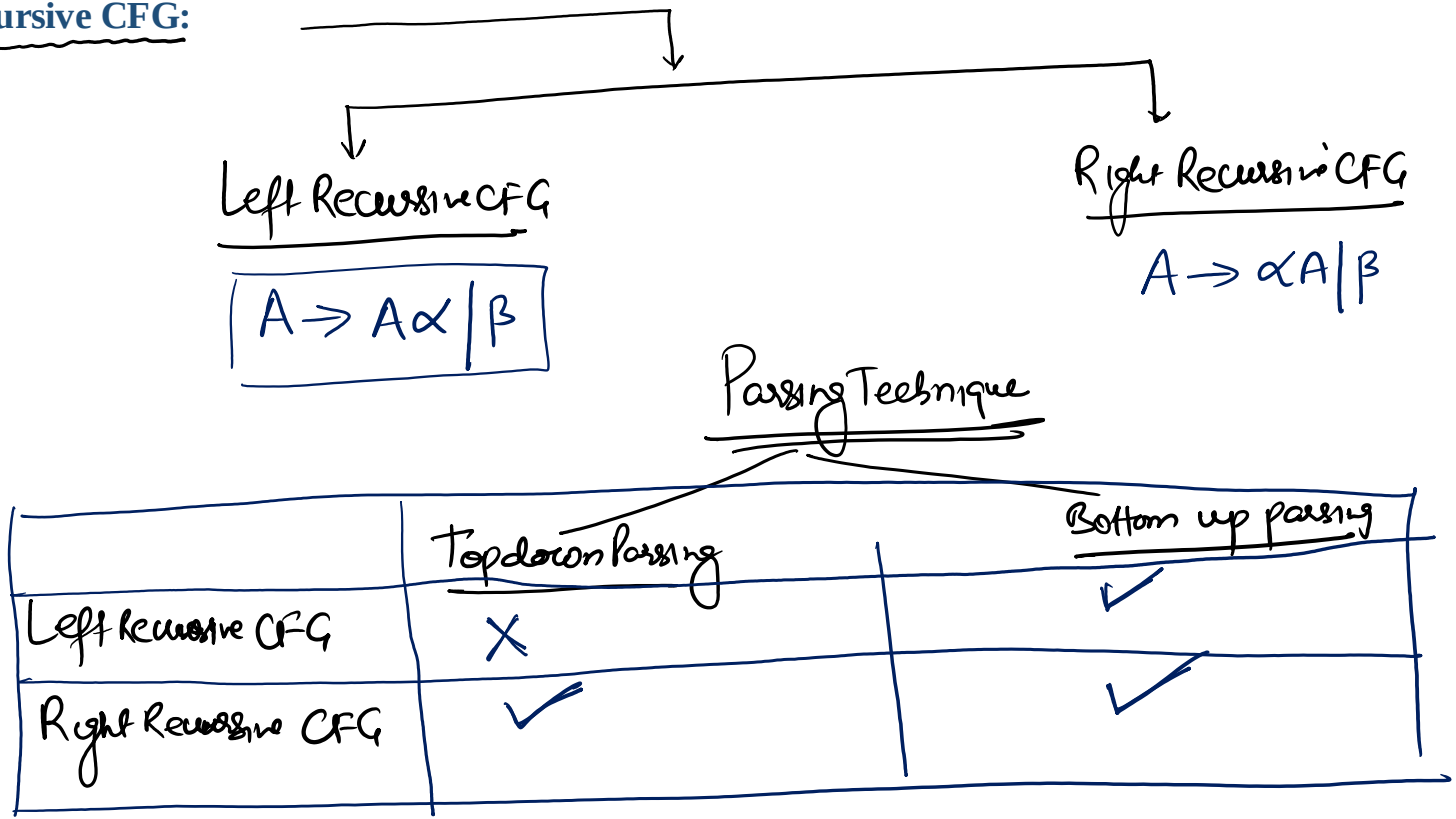
(A) $ has higher precedence and is left associative; # is right associative

(B) # has higher precedence and is left associative; $ is right associative

(C) $ has higher precedence and is left associative; # is left associative

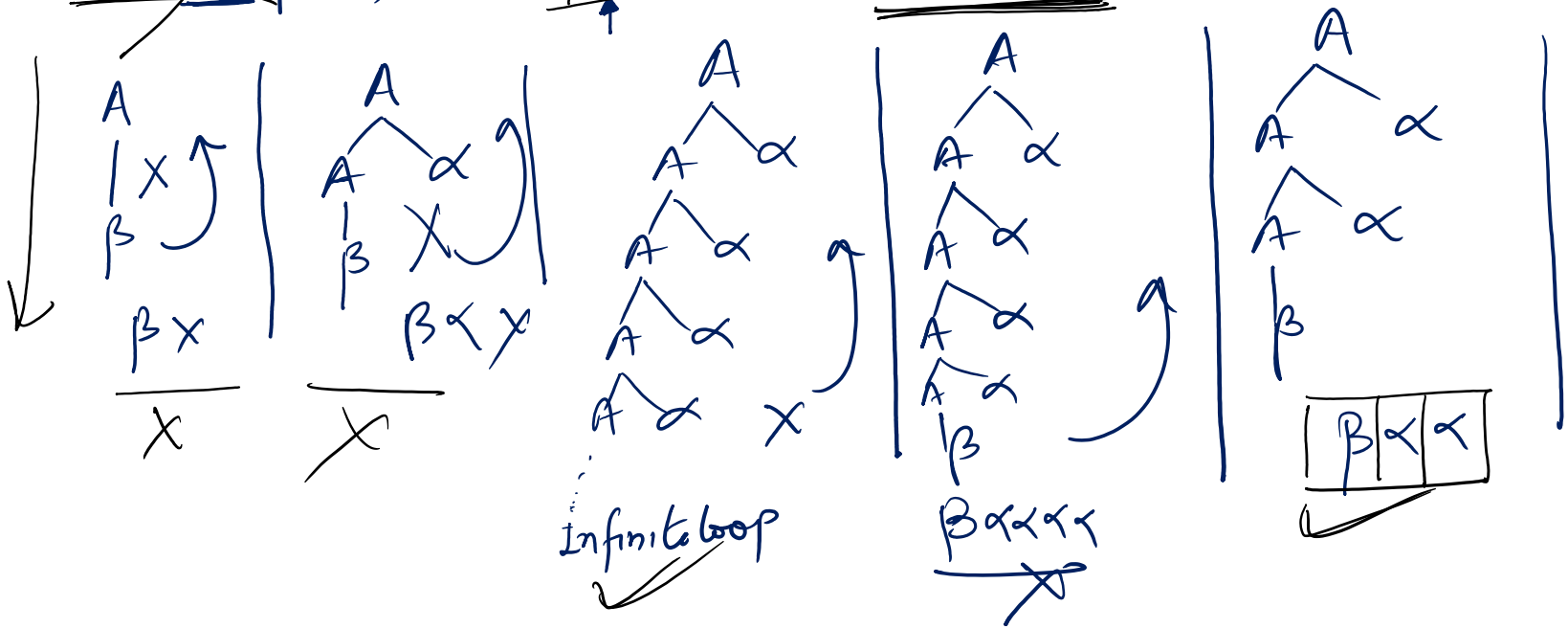
(D) # has higher precedence and is right associative; $ is left associative

Recursive CFG:



→ If Grammar is Left Recursive then it is not suitable for top down Parsing

$A \to A\alpha \mid \beta$, $\omega = \beta\alpha\alpha$ $\quad A \to A\alpha \mid \beta$



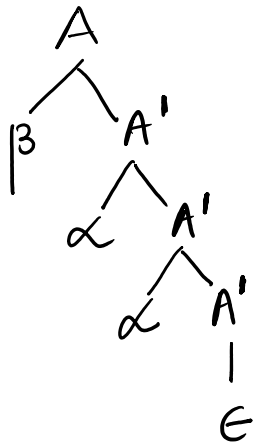
How to Eliminate Direct left Recursion:

$G\{\ A \rightarrow A\alpha \mid \beta\ \}$ — Remove Left Recursion →

non left Recursive Grammar:

$G'\{\ A \rightarrow \beta A'$
$A' \rightarrow \alpha A' \mid \in\ \}$

$\omega = \beta\alpha\alpha\$$



$= \beta\alpha\alpha$

Top down parsing

$\beta\alpha\alpha$

Eliminate Direct Left Recursion:

1) $G : \{ A \to A\alpha \mid \beta \}$ → Non Left Recursive → $G' \begin{cases} A \to \beta A' \\ A' \to \alpha A' \mid \in \end{cases}$

2) $G = \{ A \to A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \dots \mid A\alpha_r \mid \beta_1 \mid \beta_2 \mid \beta_3 \mid \dots \mid \beta_s \}$

⇓ Non Left Recursive Grammar.

$$G = \begin{cases} A \to \beta_1 A' \mid \beta_2 A' \mid \beta_3 A' \mid \dots \mid \beta_s A' \\ A' \to \alpha_1 A' \mid \alpha_2 A' \mid \alpha_3 A' \mid \dots \mid \alpha_r A' \mid \in \end{cases}$$

1) $G = \{ S \to Sa \mid b \mid aa \}$

⇓ Eliminate Left Recursion

$G' = \{ S \to bS' \mid aaS' \}$
$\{ S' \to aS' \mid \epsilon \}$

$A \to A\alpha \mid \beta$

⇓

$A \to \beta A'$

$A' \to \alpha A' \mid \epsilon$

2) $S \to Sb \mid Sab \mid asb \mid b \mid \epsilon$

$\beta_1 \quad \beta_2 \quad \beta_3$

⇓

$S \to asbS' \mid bS' \mid S'$

$S' \to bS' \mid abS' \mid \epsilon$

$\epsilon \cdot S' = S'$

$A \to A\alpha \mid \beta \implies A \to \beta A'$

$A' \to \alpha A' \mid \epsilon$

Ex.- G:

$E \to E+T \mid T$

$T \to T*F \mid F$

$F \to (E) \mid id$

Non Left Recursive Grammar

$E \to TE'$

$E' \to +TE' \mid \epsilon$

$T \to FT'$

$T' \to *FT' \mid \epsilon$

$F \to (E) \mid id$

Top-down parsing

How to Eliminate indirect Left Recursion:

$S \Rightarrow A\alpha$

$A \rightarrow S\beta \mid \gamma$

$S \circlearrowright A \Rightarrow$

Step1: Choose order of productions S, A, B, C

Step2: Choose S, Apply direct left Recursion elimination method if left Recursion exist

Step3: Choose A
Substitute S production in A production
ie) Apply Direct left Recursion eliminate

Step4: Choose B
ie) Substitute S, A
ee) Apply Direct left Recursion

Ex:- $G = \{ S \to Aa S \mid bS \mid a,\ A \to Ab \mid SbA \mid b \}$

Aa S : Indirect Left Recursion

Ab : Direct Left Recursion

SbA : Indirect Left Recursion

$G' = \{ S \to AaS \mid bS \mid a,\ A \to bSbAA' \mid abAA' \mid bA',\ A' \to bA' \mid aSbAA' \mid \in \}$

Soln:

Step1: S, A

Step2: Choose $S \to AaS \mid bS \mid a$, No Direct Left Recursion

Step3: Choose A:

$A \to Ab \mid SbA \mid b$

$A \to Ab \mid AaSbA \mid bSbA \mid abA \mid b$

⇓ Eliminate Direct Left Recursion

$A \to bSbAA' \mid abAA' \mid bA'$

$A' \to bA' \mid aSbAA' \mid \in$

Ex:- $G = \{ S \to Sb \mid Aa \mid b \;,\; A \to Ab \mid Sa \mid a \}$

Sol^{n}:-

$$G' = \begin{cases} S \to Aas' \mid bs' \;,\; A \to Ab \mid Sa \mid a \} \\ s' \to bs' \mid \epsilon \end{cases}$$

$$G'' = \begin{cases} S \to Aas' \mid bs' \;,\; A \to Ab \mid Aas'a \mid bs'a \mid a \} \\ s' \to bs' \mid \epsilon \end{cases}$$

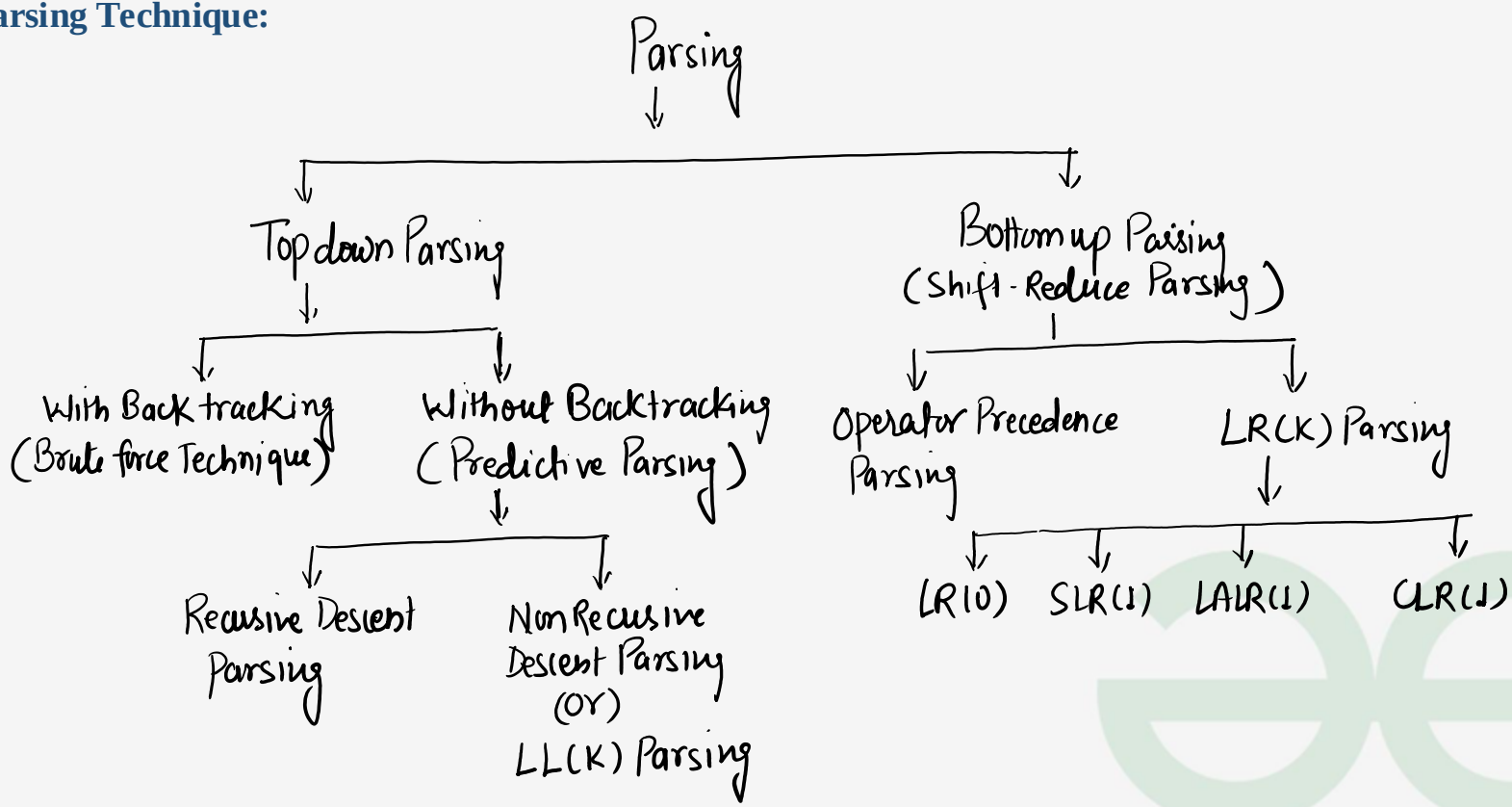
Non left Recursive Grammar

$$G''' = \begin{cases} S \to Aas' \mid bs' \;,\; A \to bs'aA' \mid aA' \\ s' \to bs' \mid \epsilon \qquad\qquad A' \to bA' \mid as'aA' \mid \epsilon \end{cases}$$

Left Factoring:

Compiler Design

Parsing Technique:



Thank You !