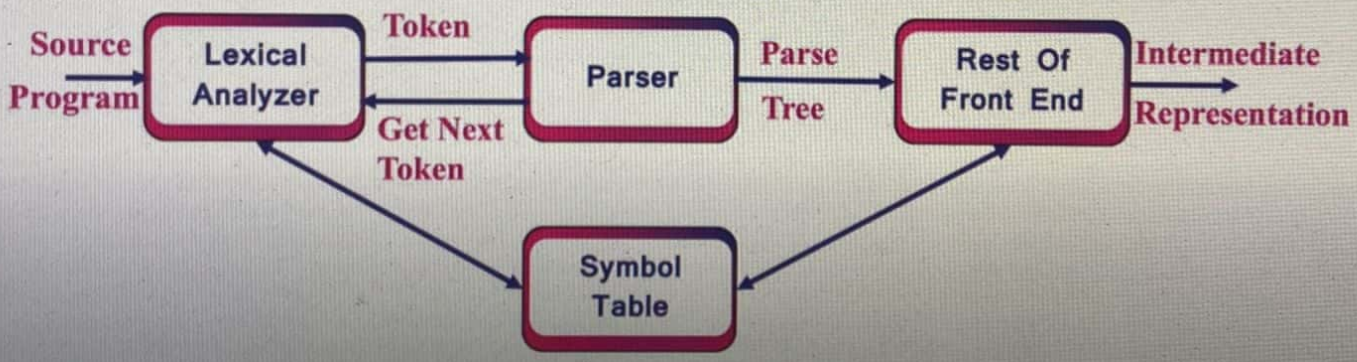
ROLE OF PARSER

- Tokens is given as input to the Parser
- Parser checks the Input/Token Stream with the Grammar of Language
- Context Free Grammar is used to define the Grammar of Language
- Constructs intermediate representation called Parse Tree/Syntax Tree for valid Tokens
- Detects and Reports Syntax Errors for Invalid Tokens
- Recovers from commonly occurring errors if possible

ROLE OF PARSER

- **Two main function**
  - Constructs parse tree for correct token streams
  - Detects and reports syntax error for incorrect token streams



SYNTAX ERROR HANDLING

Error Handler

- The process of **Error Reporting** and **Error Recovery** as a continuous logical entity in syntax analyzer called **Error Handling**
- Should report the error with proper information to correct it
- It is a continuous process until complete source program is error free

- **Goals/Functions of Error Handler in Parser**
  - Report the presence of errors **clearly** and **accurately**
  - Recover from each error quickly to identify and rectify the subsequent errors
  - Add minimal overhead (slow down) to the processing of correct programs

PROGRAMMING ERRORS IN DIFFERENT LEVELS

2. Syntax Error

- Errors in Structure Formation or Invalid Statement
- Tokens violates grammatical rules
- Syntax error does not allow program compilation
- Does not allow the program to run
- **Example**
  - Missing Parenthesis ( ) , { } , [ ]
  - Missing semicolon x = y + z ;
  - Undeclared Variable int a ; a = b + c ;
  - Incorrect format of looping and Selection statements

PROGRAMMING ERRORS IN DIFFERENT LEVELS

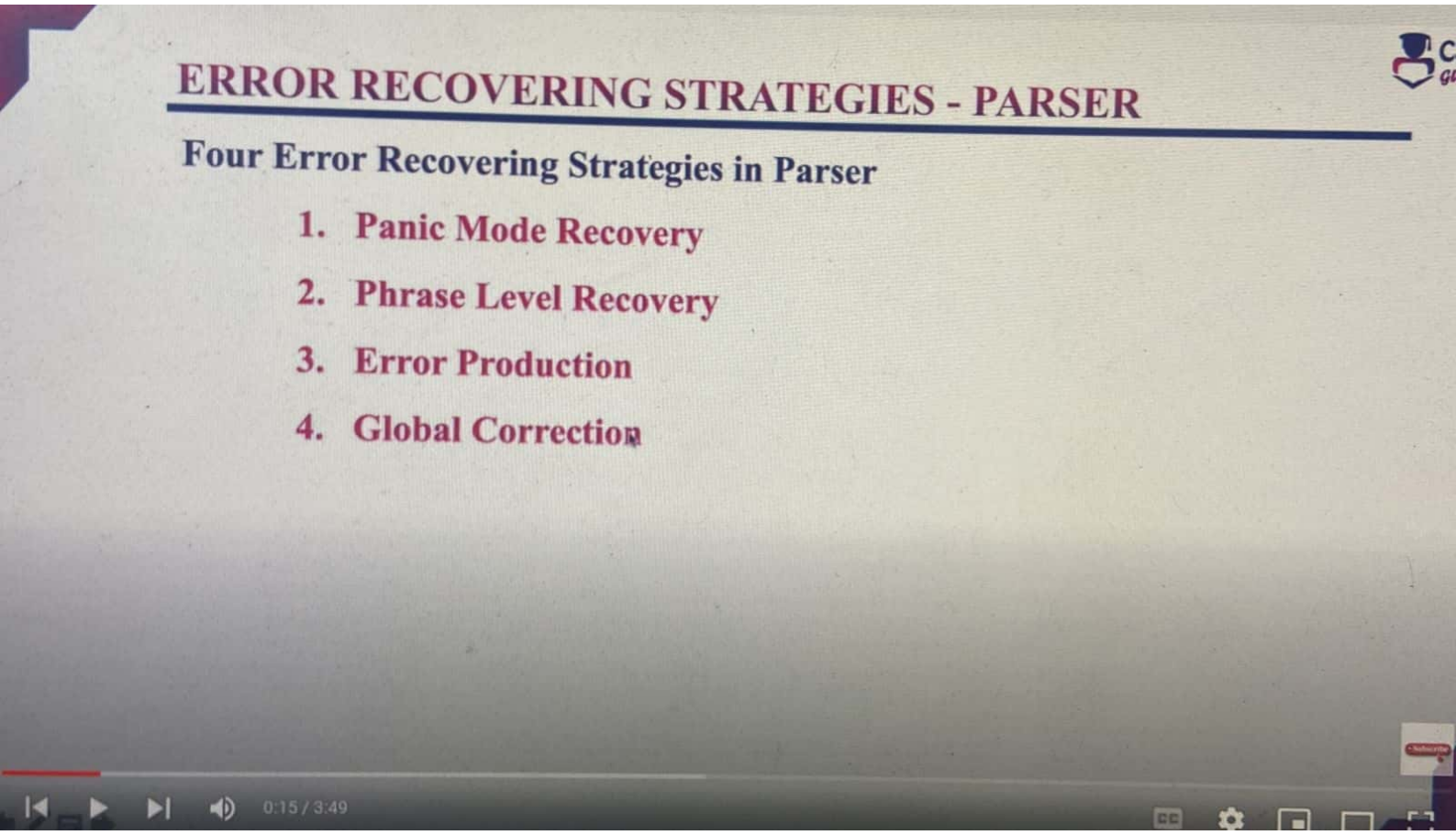
3. Semantic Errors

- Error violates meaning of program statement or incorrect code
- Error is syntactically correct but does not work correctly
- Difficult to detect and correct
- **Example**
  - Incompatible Operands
  - Type Mismatch Errors
  - Dangling else
  - Array index bound

PROGRAMMING ERRORS IN DIFFERENT LEVELS

4. Logical Errors

- Program is error free but related to logical mistakes
- Logical errors are difficult to detect by compiler
- It allows the program to run but will get incorrect result
- Resolved by verifying logical path of the program
- Type of Runtime Error
- **Example**
  - Incorrect reasoning
  - Indefinite Recursive Call
  - Infinite Loops
  - Null Reference



ERROR RECOVERING STRATEGIES

1. Panic Mode Recovery

- Parser discards the input symbol one at a time until designated set of **Synchronized Tokens** is found
- Synchronizing tokens are usually delimiters like semicolon or }
- Very simple recovering strategy
- **Drawbacks**
  - Corrects single error
  - May leads to additional errors

ERROR RECOVERING STRATEGIES

2. Phrase Level Recovery

- Parser performs local correction on inputs
  - Replace the incorrect character by a correct one
- It may replace prefix of remaining input by some string
  - Replace comma by semicolon
  - Delete/Insert semicolon
- Used in Error repairing compilers
- **Drawback:**
  - Difficulty in assessing the actual error
  - It may lead to an infinite loop

ERROR RECOVERING STRATEGIES

3. Error Production

- Anticipate common errors & implement grammar rules
- Construct the parser to detect these anticipated errors
- The parser then implements appropriate error diagnostics
- YACC compiler uses this strategy

ERROR RECOVERING STRATEGIES

4. Global Correction

- Compilers that make possible few changes to process incorrect input string
- Algorithms that implements a global least cost correction
- Given an incorrect input string S and grammar G, the algorithm constructs parse tree with small number of transformations to correct the input
- Costly strategy with respect to time and space
- So its only theoretical interest

CONTEXT FREE GRAMMAR

- The **input** to the Syntax Analyzer is **Token Stream**
- **Syntax**-Structure of Programming Language
- Syntax Analysis is the process of verifying the **Syntactical Structure** of the Token Stream
- If so the Parser will construct the **Parse Tree**
- The Syntax of the source language is described by **Context-Free Grammar(CFG)**
- The Grammar of a language describes **directly Syntactic Structure** and **indirectly Semantic Structure**

CONTEXT FREE GRAMMAR- DEFINITION

The Context Free Grammar(CFG) is a collection of four-tuple (V,T,P,S)

- **V is a set of Non-Terminals**
  - Defines finite set of non-terminal symbols or variables
  - Represented in uppercase letters S A B E
- **T is a set of Terminals**
  - Used to form language constructs
  - Represented in lowercase letters a b c
- **P is Production Rule**
  - Built production rules using non-terminal and terminal symbols
- **S is Start Symbol**
  - Derivation should always start with start symbol

Suggested: Compiler Design

CONTEXT FREE GRAMMAR - EXAMPLE

G = ({S},{a,b}, P, S)

Where P = {

S -> a S a,

S -> b S b

S -> ε

}

S → a S a
→ ab S b a

CSE GURU

DERIVATION

- The **Syntax** of the Source Language is specified by the **Grammar**
- Grammar is based on **Grammatical rules** with **Terminal** and **Non-Terminals**

$G = \{V, T, P, S\}$

- Terminals represents **Tokens**
- To verify a Tokens, **derive a string** of terminals using Grammatical rules
- If derivation exist, then token is **valid**
- The process of showing the input string verified with grammar is called **Derivation**

DERIVATION

- **Steps**

1. The Derivation starts with a **Start Symbol**
2. Replace a **Non-Terminal** by a **Terminal**
3. Continue step 2 until the complete derivation contains **Terminal** strings

- **Types of Derivation**

1. **Left-most Derivation**
   - Always choose the left-most non-terminal to apply the production rule
2. **Right-most Derivation**
   - Always choose the Right-most non-terminal to apply the production rule

LEFT-MOST DERIVATION - EXAMPLE

- **Example :** Let G be a Context free Grammar for which production rules are given

  S → aS

  S → aSbS

  S → ε

Derive the string **aaabaab** using above grammar

Left-Most Derivation

$S \xrightarrow{lm} aS$ $\quad S \to aS$

$\xrightarrow{lm} aaS$ $\quad S \to aS$

$\xrightarrow{lm} aaaSbS$ $\quad S \to aSbS$

$\to aaabS$ $\quad S \to \varepsilon$

$\to aaabaSbS$ $\quad S \to aSbS$

$\to aaabaaSbS$ $\quad S \to aS$

$\to aaabaabS$ $\quad S \to \varepsilon$

$\to aaabaab$ $\quad S \to \varepsilon$

7:14 / 18:24

RIGHT-MOST DERIVATION - EXAMPLE

- **Example :** Let G be a Context free Grammar for which production rules are given

  S → aS

  S → aSbS

  S → ε

Derive the string **aaabaab** using above grammar

Right-Most Derivation

S → aS S → aS
→ aaSbS S → aSbS
→ aaSbaS S → aS
→ aaSbaaSbS S → aSbS
→ aaSbaaSb S → ε
→ aaSbaab S → ε
→ aaaSbaab S → aS
→ aaabaab S → ε

PARSE TREE

- Parse tree is the **hierarchical representation** of the Grammatical derivation of the string
- It shows **Pictorially** how the grammar **derives the string** in the language
- Made up of nodes and branches with **Parent-Child** relationship
- Each node is either a **Root, Branch or a Leaf node**
- Every **Terminal** string generated by the Grammar has at least one **Parse Tree**

PARSE TREE - PROPERTIES

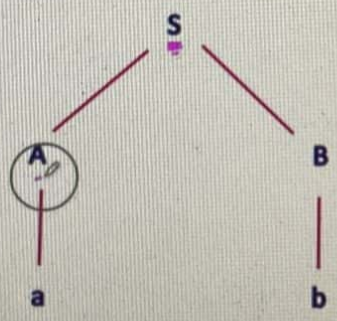
1. **Root** represents **Start Symbol**
2. **Interior node** represents **Non-Terminals**
3. **Leaf Nodes** represents **Terminals**
4. For each **Non-Terminal** apply the **Production Rule** exist in the grammar

- Example

$S \rightarrow AB \mid aaB$

$A \rightarrow a \mid Aa$

$B \rightarrow b$



PARSE TREE- EXAMPLE

- **Example :** Let G be a Context free Grammar for which production rules are given

$$S \rightarrow aS \mid aSbS \mid \varepsilon$$

Derive the string **aaabaab** using above grammar

PARSE TREE

S

PARSE TREE- EXAMPLE

- **Example :** Let G be a Context free Grammar for which production rules are given

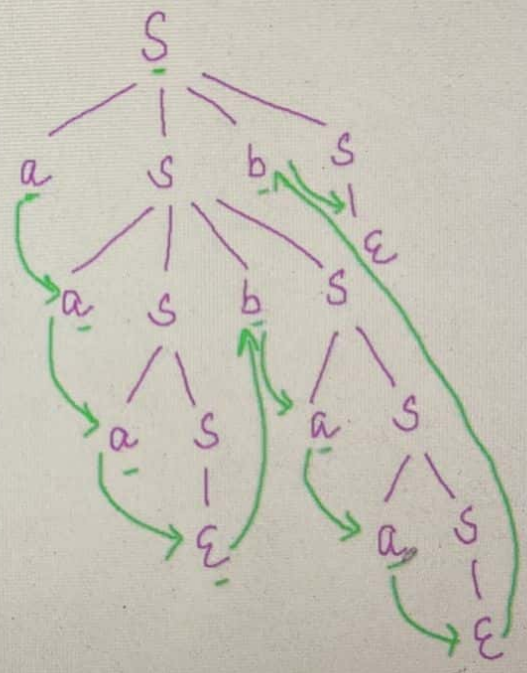
  S → aS

  S → aSbS

  S → ε

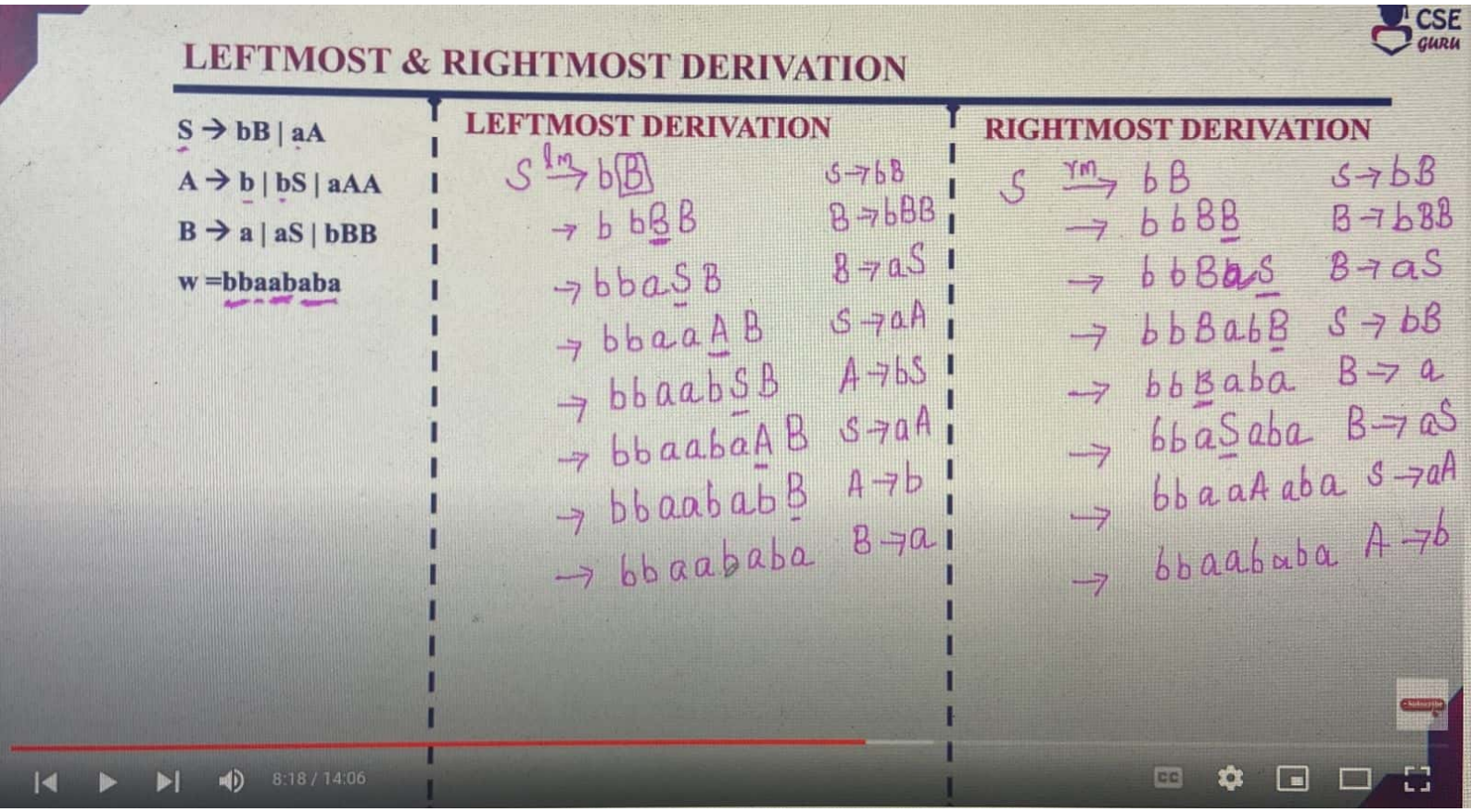
Derive the string **aaabaab** using above grammar

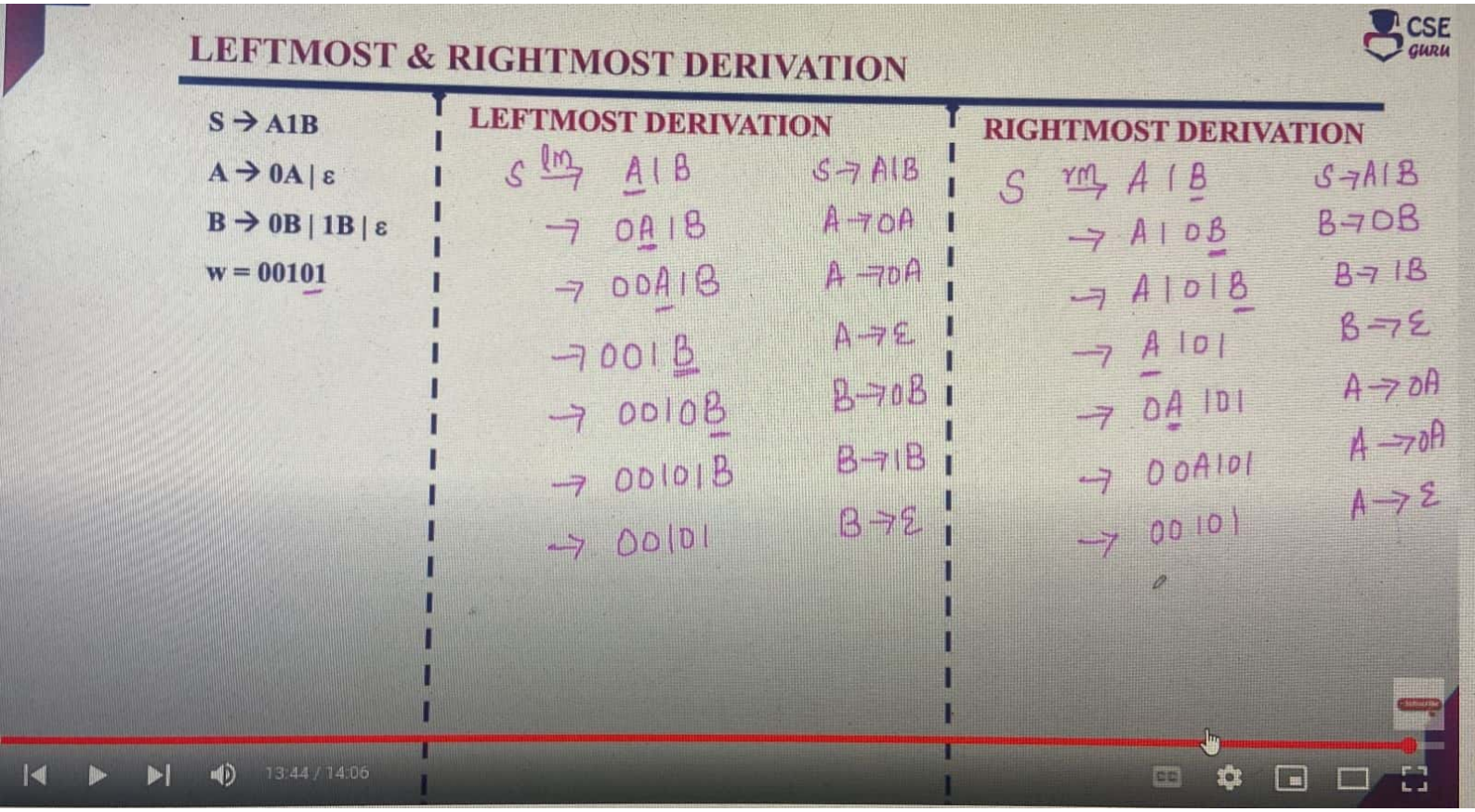
PARSE TREE



PARSE TREE - ADVANTAGE

- Easy to represent the **Hierarchical/Syntactical** structure
- Easy to identify the **Alternate Productions**





LEFT-MOST DERIVATION & PARSE TREE

CSE GURU

E → E + E | E – E | E * E | E / E | a | b

Derive the string **a + b * a + b**

Left-Most Derivation

$E \xrightarrow{lm} E + E$ — $E \to E+E$

$\to a + E$ — $E \to a$

$\to a + E * E$ — $E \to E*E$

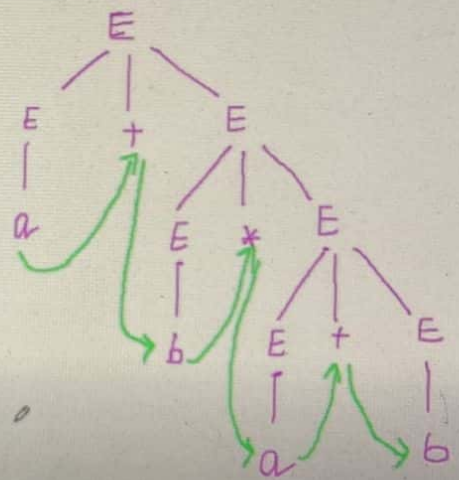
$\to a + b * E$ — $E \to b$

$\to a + b * E + E$ — $E \to E+E$

$\to a + b * a + E$ — $E \to a$

$\to a + b * a + b$ — $E \to b$

PARSE TREE



6:10 / 10:38

RIGHT-MOST DERIVATION & PARSE TREE

E → E + E | E – E | E * E | E / E | a | b

Derive the string **a + b * a + b**

Right-Most Derivation

$E \xrightarrow{rm} E+E$ $\quad E \to E+E$

$\to E+E+E$ $\quad E \to E+E$

$\to E+E+b$ $\quad E \to b$

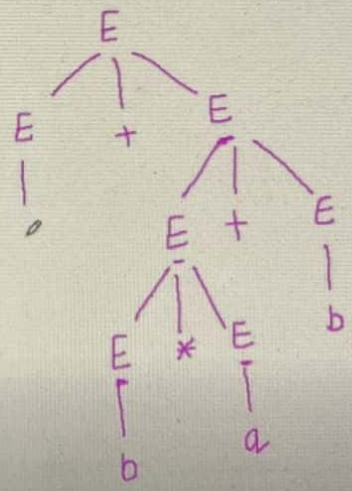
$\to E+E*E+b$ $\quad E \to E*E$

$\to E+E*a+b$ $\quad E \to a$

$\to E+b*a+b$ $\quad E \to b$

$\to a+b*a+b$ $\quad E \to a$

PARSE TREE



LEFT-MOST DERIVATION & PARSE TREE

S → aB | bA
A → a | aS | bAA
B → b | bS | aBB

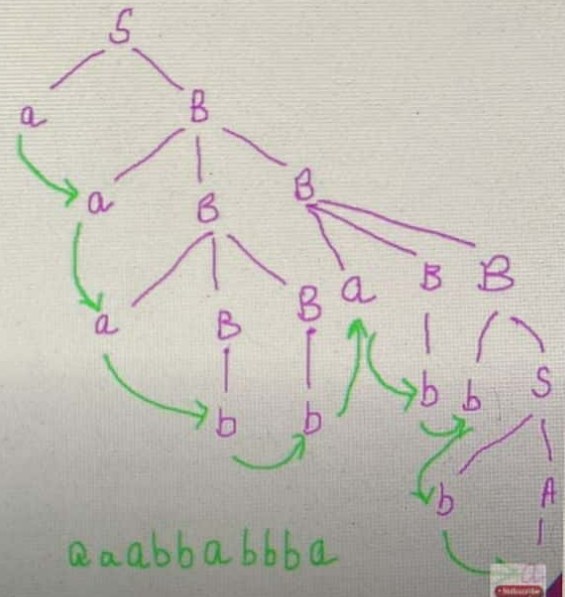
Derive the string **aaabbabbba**

Left-Most Derivation

S → aB
→ aaBB
→ aaaBBB
→ aaabBB
→ aaabbB
→ aaabbaBB
→ aaabbabB
→ aaabbabbS
→ aaabbabbbA
→ aaabbabbba

S → aB
B → aBB
B → aBB
B → b
B → b
B → aBB
B → b
B → bS
S → bA
A → a

PARSE TREE



RIGHT-MOST DERIVATION & PARSE TREE

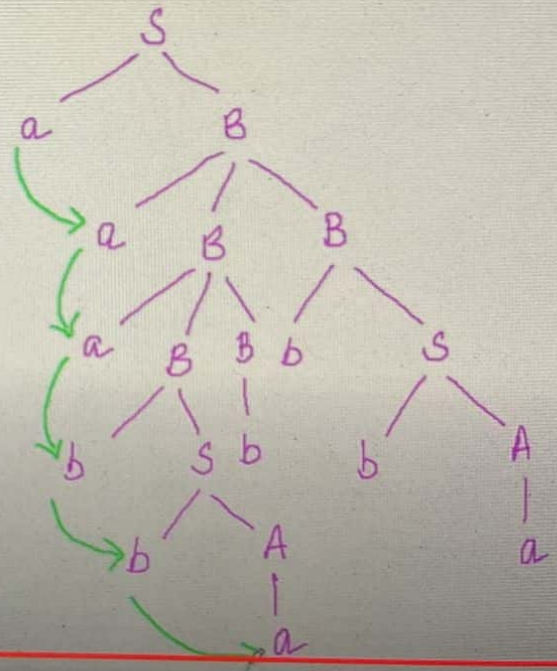
S → aB | bA
A → a | aS | bAA
B → b | bS | aBB

Derive the string **aaabbabbba**

Right-Most Derivation

$S \xrightarrow{RM} aB$ (S → aB)
$\rightarrow aaBB$ (B → aBB)
$\rightarrow aaBbS$ (B → bS)
$\rightarrow aaBbbA$ (S → bA)
$\rightarrow aaBbba$ (A → a)
$\rightarrow aaaBBbba$ (B → aBB)
$\rightarrow aaaBbbba$ (B → b)
$\rightarrow aaabSbbba$ (B → bS)
$\rightarrow aaabbAbbba$ (S → bA)
$\rightarrow aaabbabbba$ (A → a)

PARSE TREE



AMBIGUOUS GRAMMAR

- Grammar **generates more than one parse tree**
- Grammar produce **more than one LMD/RMD**
- Parse tree is **not unique**
- Troubles program compilation
- Eliminate ambiguity by transforming the grammar into **unambiguous**
- **Drawbacks:**
  - Parsing Complexity
  - Affects other phases

AMBIGUOUS GRAMMAR

CSE GURU

- **Conditions for Ambiguity**
  - Grammar Produce more than one Parse Tree

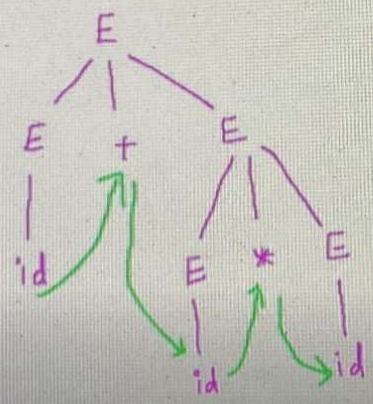
    (OR)
  - Grammar Derives more than one Leftmost Derivation(LMD)

    (OR)
  - Grammar Derives more than one Rightmost Derivation(RMD)

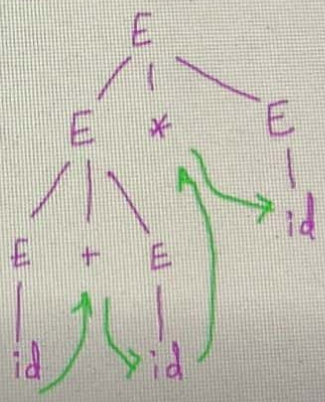
AMBIGUOUS GRAMMAR WITH PARSE TREE - EXAMPLE

E → E + E | E * E | id For the string **id + id * id**

PARSE TREE 1



PARSE TREE 2



AMBIGUOUS GRAMMAR WITH LMD - EXAMPLE

E → E + E | E * E | id For the string **id + id * id**

LMD 1

E → E + E
→ id + E
→ id + E * E
→ id + id * E
→ id + id * id

LMD 2

E → E * E
→ E + E * E
→ id + E * E
→ id + id * E
→ id + id * id

AMBIGUOUS GRAMMAR WITH RMD - EXAMPLE

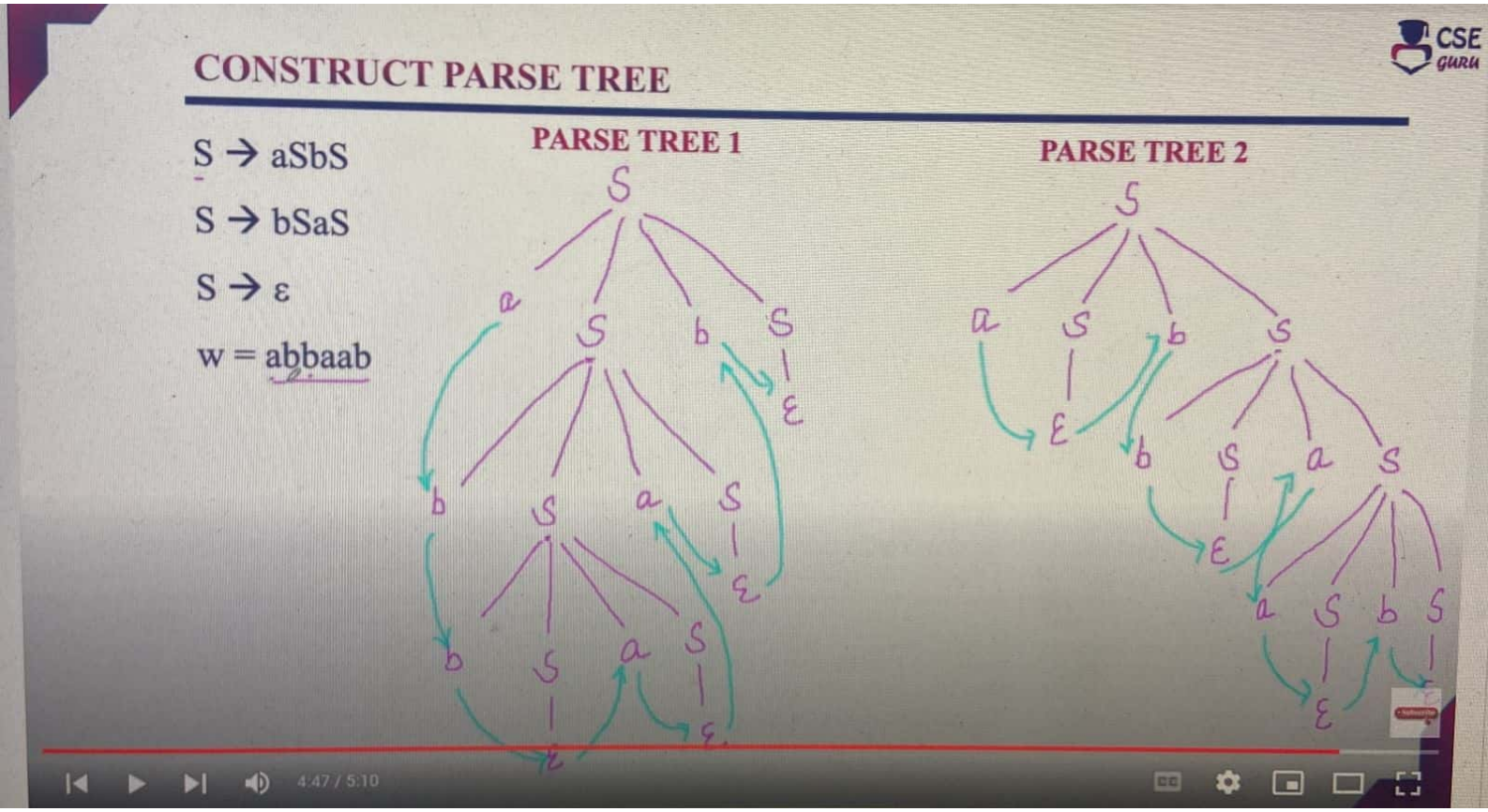
E → E + E | E * E | id · For the string **id + id * id**

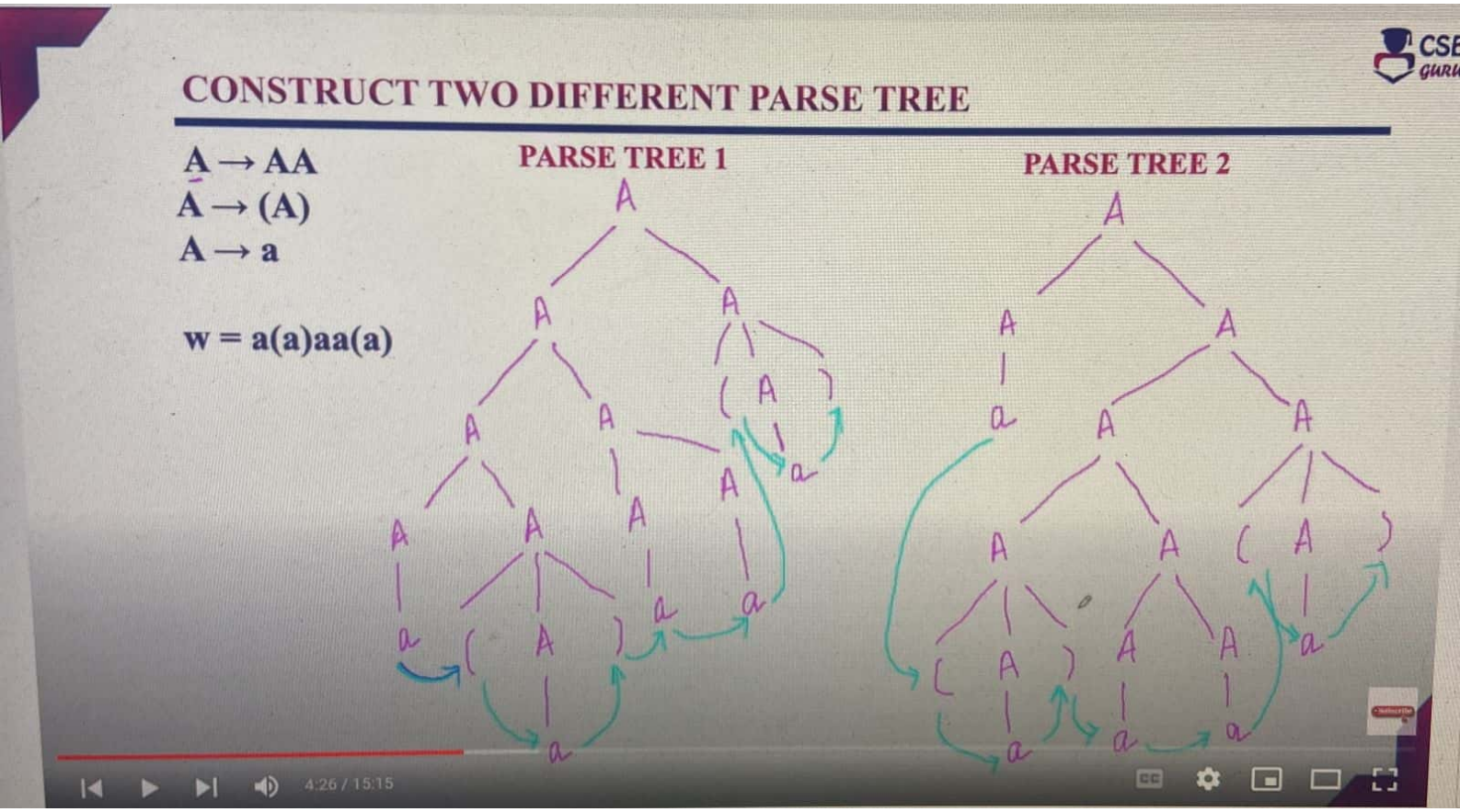
RMD 1

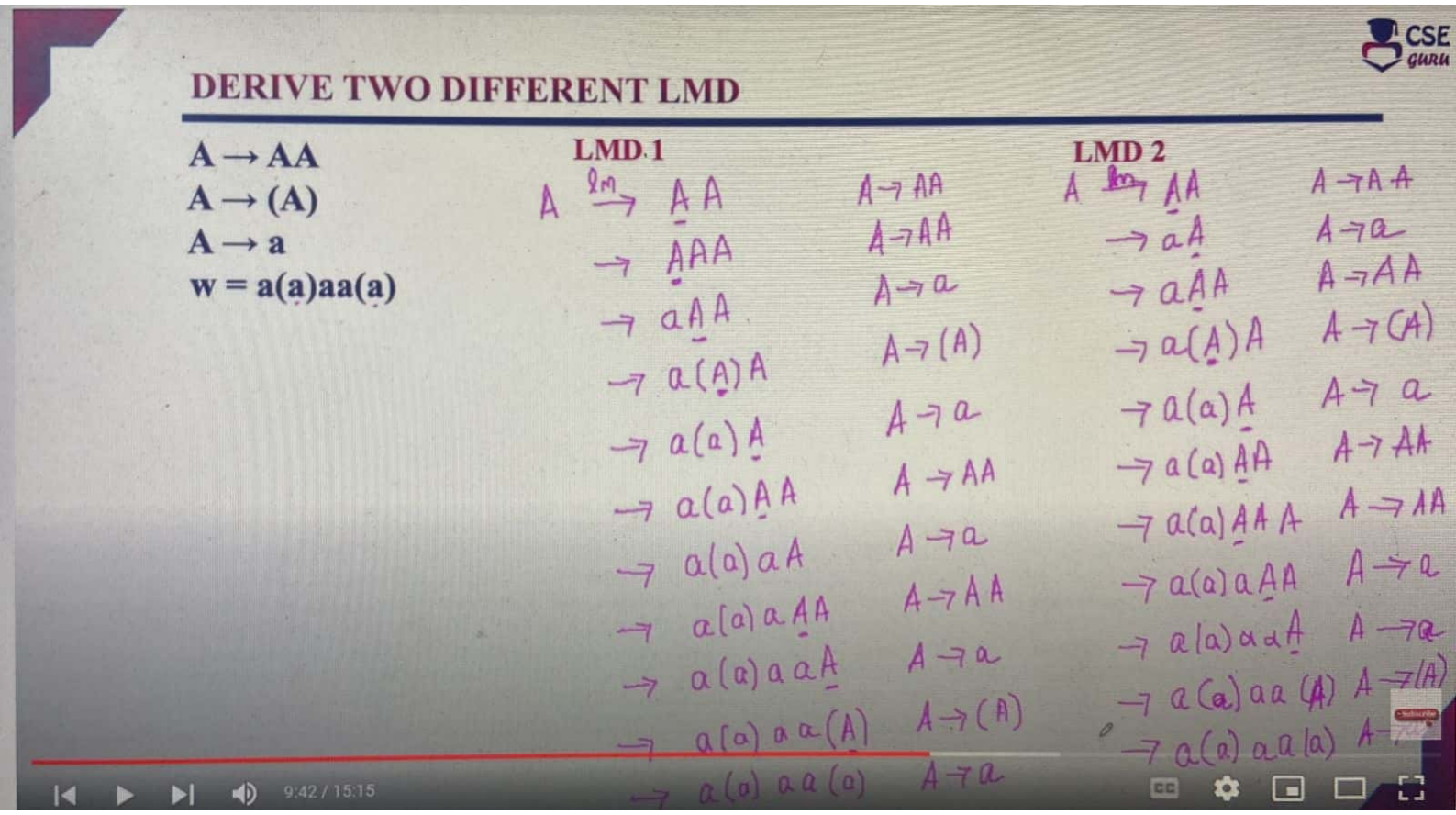
E → E + E
→ E + E * E
→ E + E * id
→ E + id * id
→ id + id * id

RMD 2

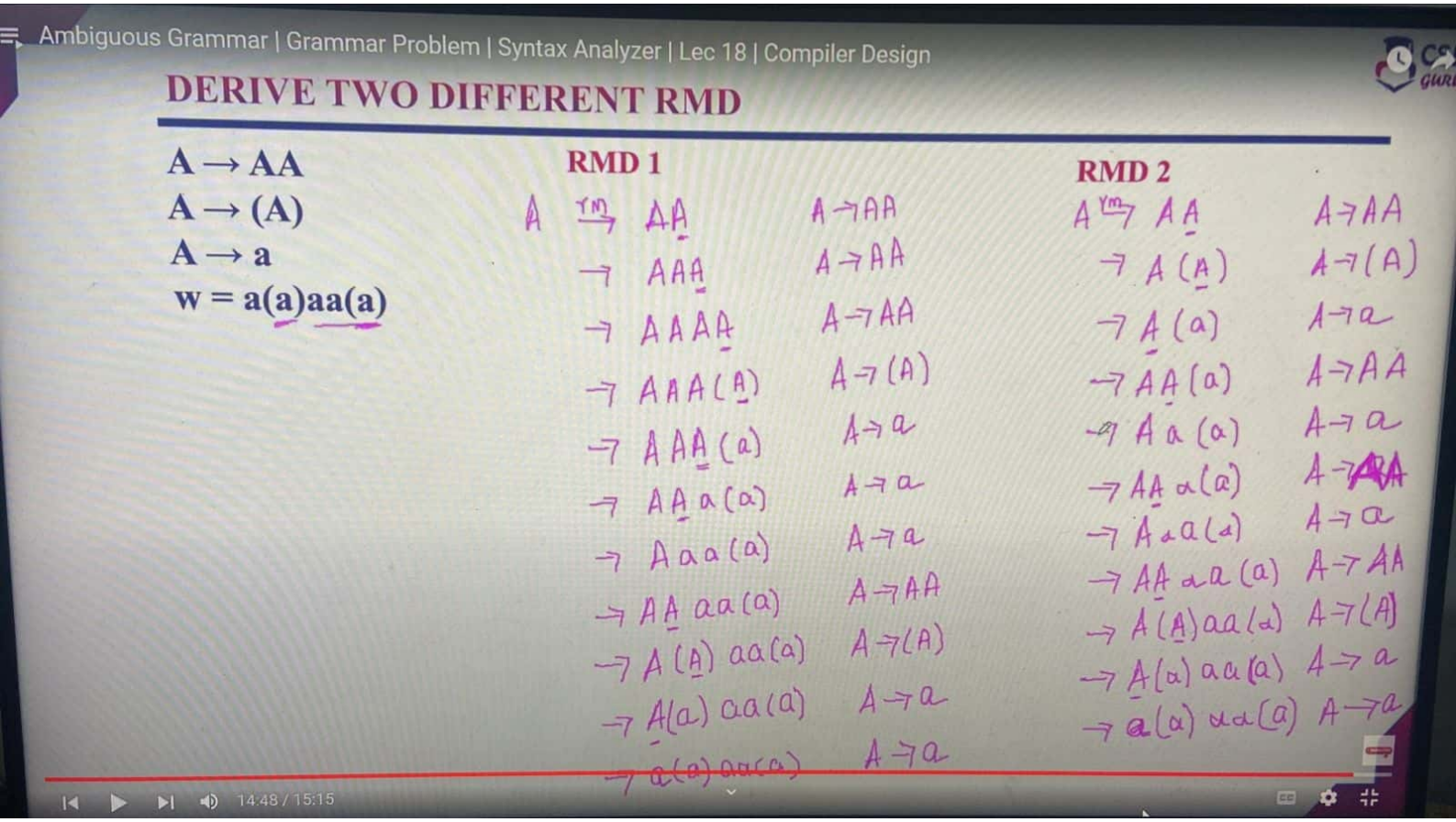
E → E * E
→ E * id
→ E + E * id
→ E + id * id
→ id + id * id







Scanned with CamScanner



GIVEN GRAMMAR

Given the grammar $G = (\{A,S\}, \{a,b, \varepsilon\}, R, S)$

$$R = \{S \rightarrow SS \mid AAA \mid \varepsilon$$
$$A \rightarrow aA \mid Aa \mid b \quad \}$$

a) Show a leftmost derivation of the string abbaaba.

b) Show a rightmost derivation of the string abbaaba.

c) Show the parse tree corresponding to a derivation of abbaaba.

d) Is this grammar unambiguous? Why or why not?

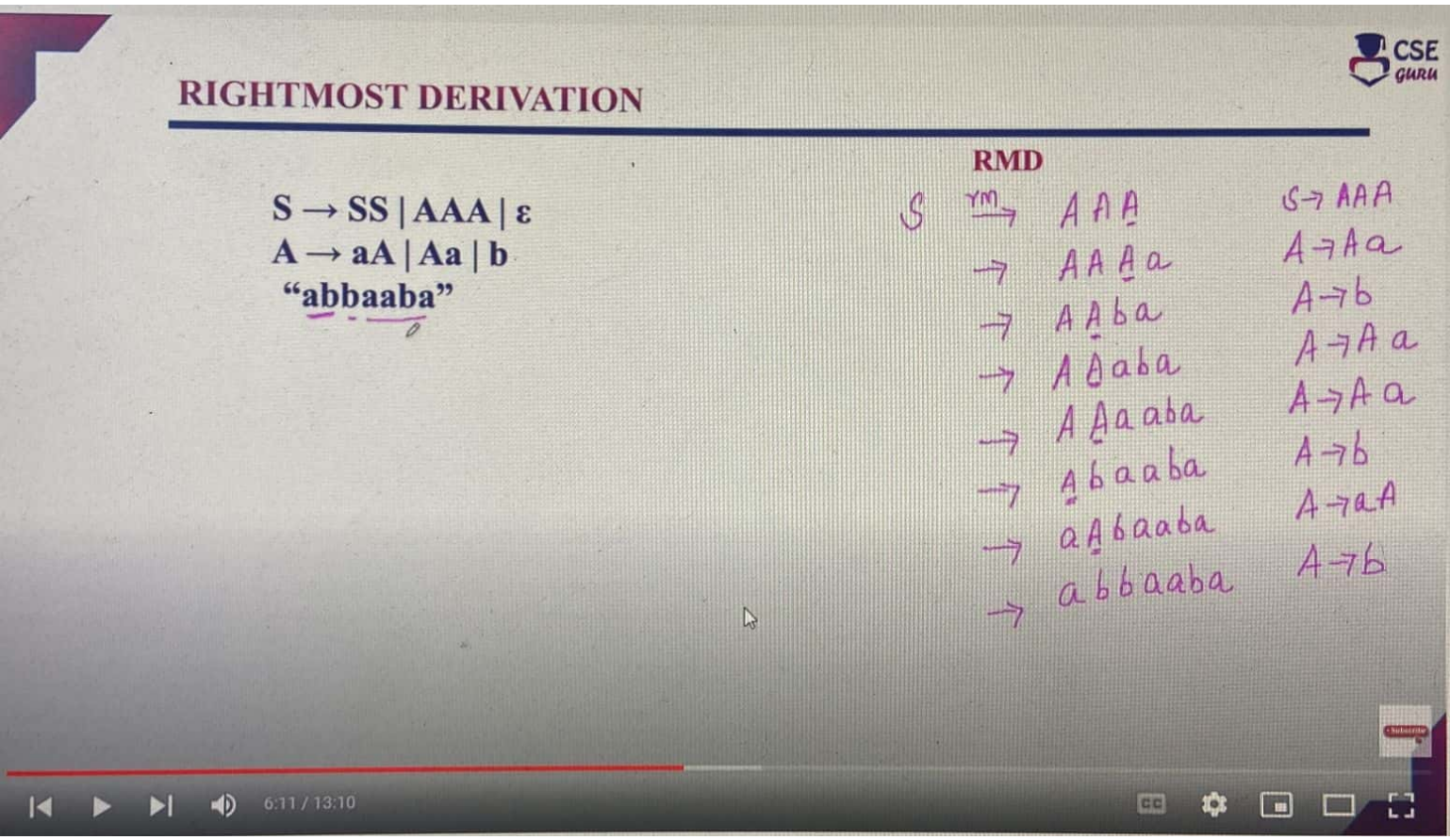
LEFTMOST DERIVATION

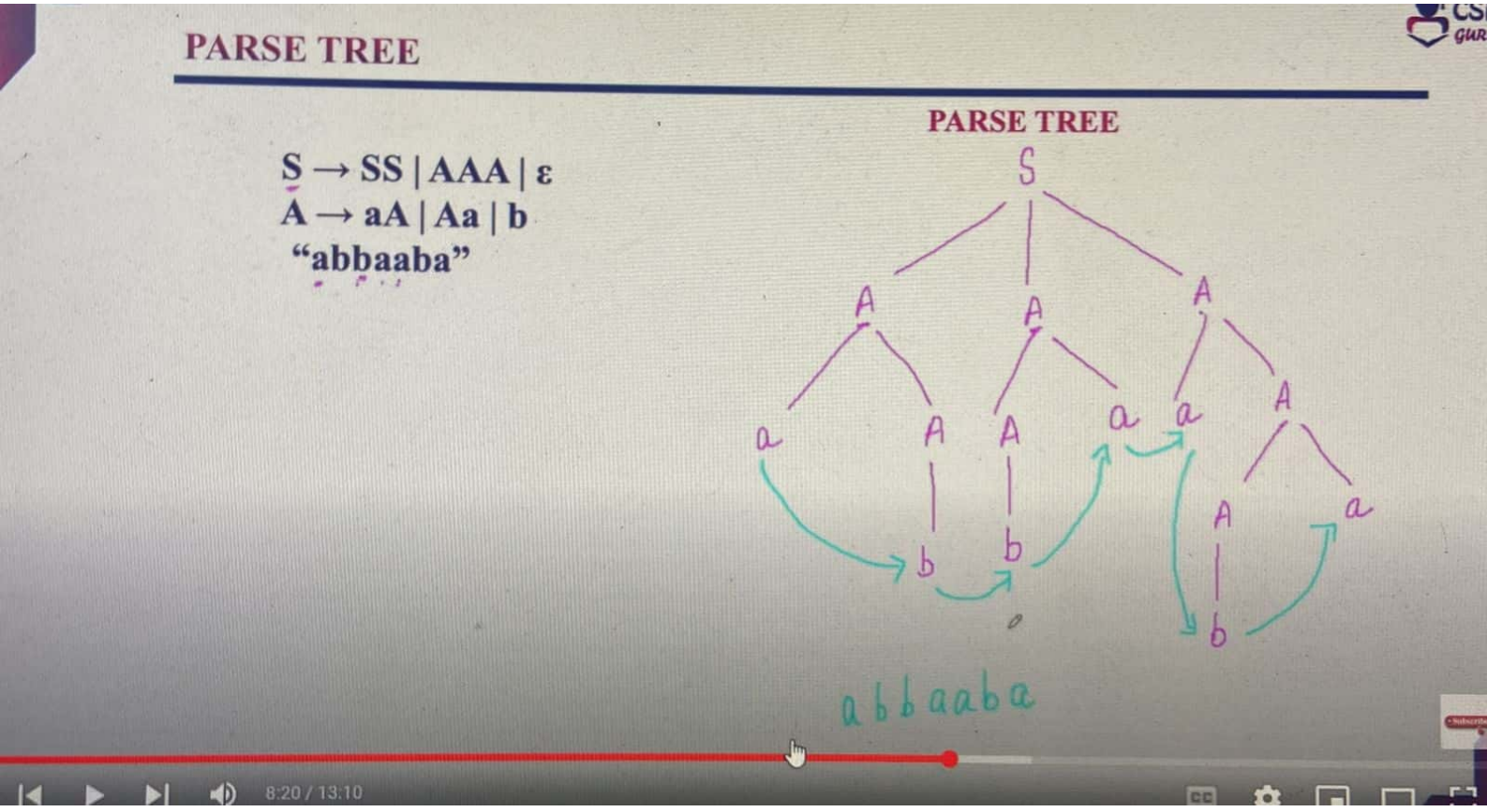
CSE GURU

$S \rightarrow SS \mid AAA \mid \varepsilon$
$A \rightarrow aA \mid Aa \mid b$
"abbaaba"

LMD

$S \xrightarrow{lm} AAA$	$S \rightarrow AAA$
$\rightarrow aAAA$	$A \rightarrow aA$
$\rightarrow abAA$	$A \rightarrow b$
$\rightarrow abbA$	$A \rightarrow b$
$\rightarrow abbaA$	$A \rightarrow aA$
$\rightarrow abbaAa$	$A \rightarrow Aa$
$\rightarrow abbaaAa$	$A \rightarrow aA$
$\rightarrow abbaaba$	$A \rightarrow b$





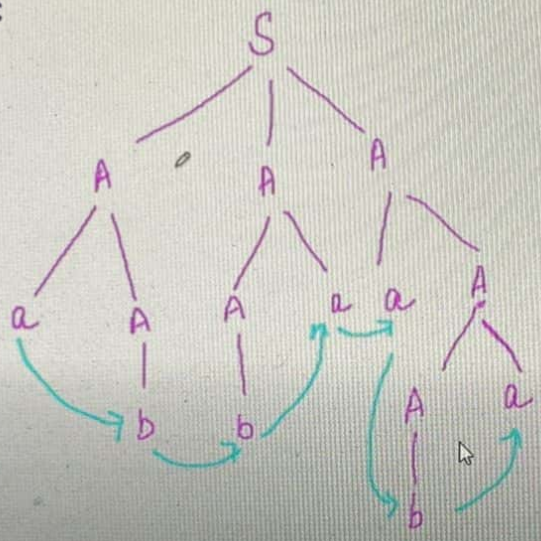
IS THIS GRAMMAR UNAMBIGUOUS? No

$S \rightarrow SS \mid AAA \mid \varepsilon$

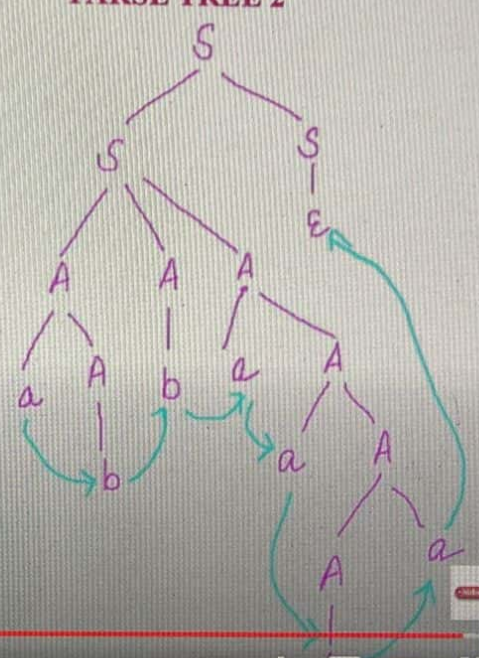
$A \rightarrow aA \mid Aa \mid b$

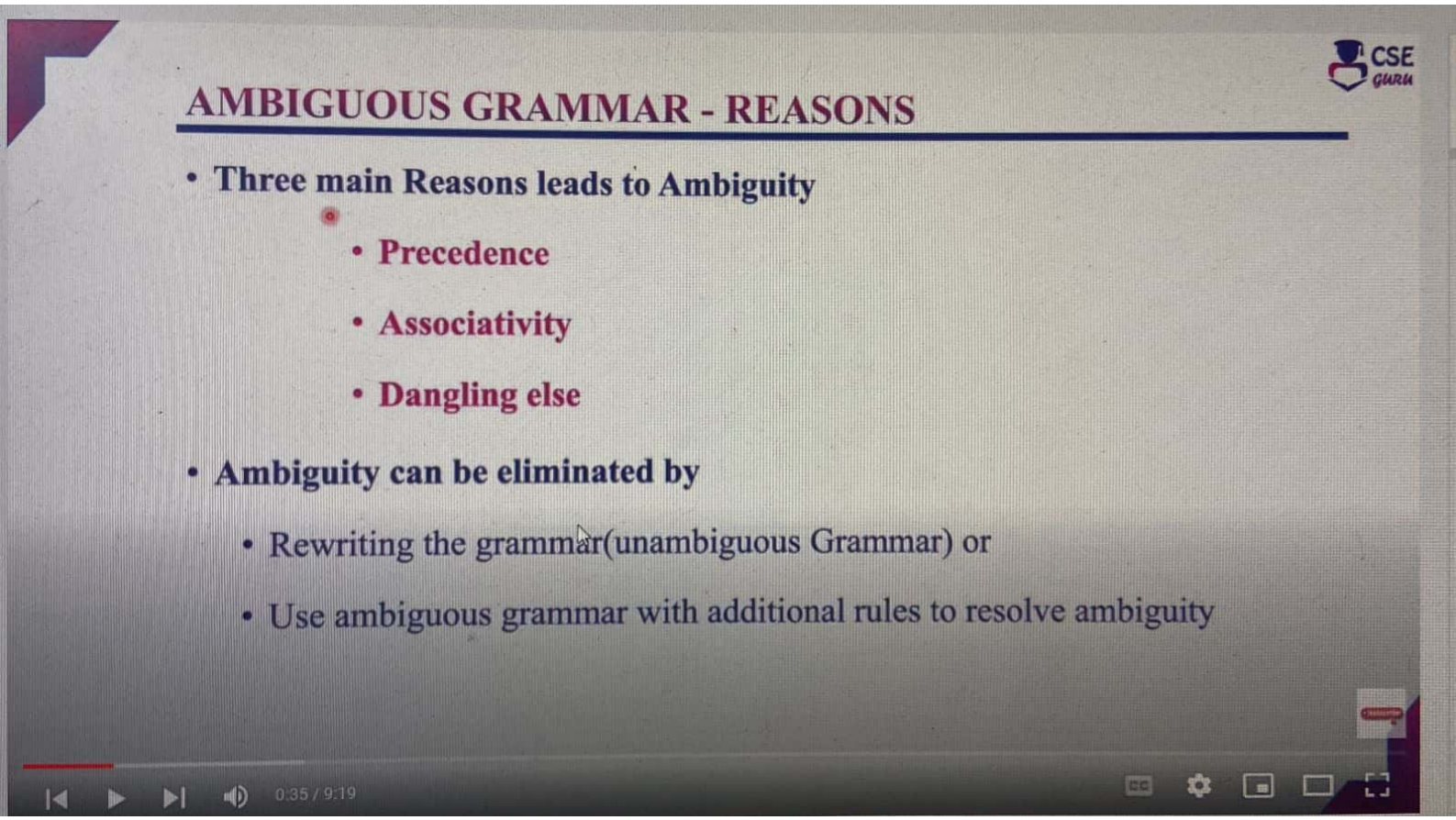
"abbaaba"

PARSE TREE 1



PARSE TREE 2





DANGLING ELSE

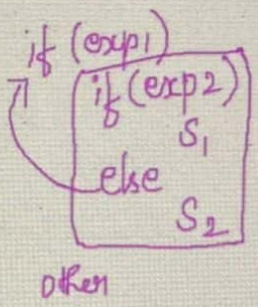
- Consider the following dangling else grammar

  stmt —> if expr then stmt
  | if expr then stmt else stmt
  | other

  - "other" represents any other program statements

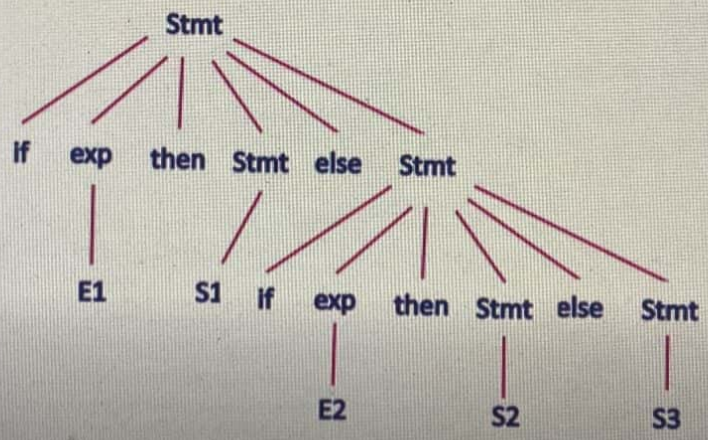
- The compound conditional statement of above grammar is

  If E1 then S1 else if E2 then S2 else S3



PARSE TREE – DANGLING ELSE

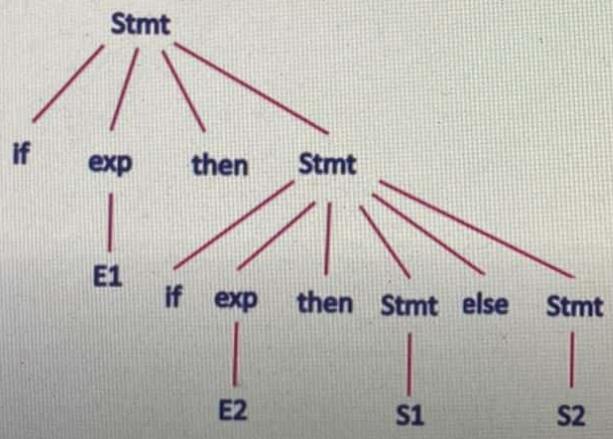
If E1 then S1 else if E2 then S2 else S3



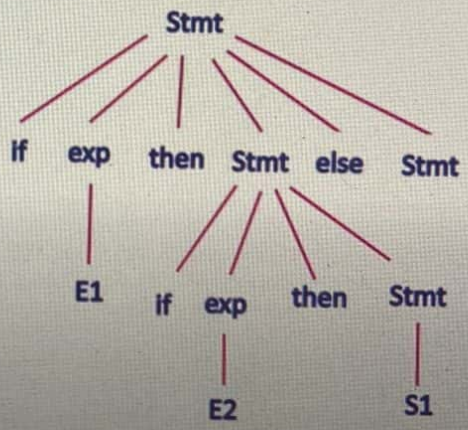
TWO POSSIBLE PARSE TREES – DANGLING ELSE

String: If E1 then if E2 then S1 else S2

PARSE TREE 1



PARSE TREE 2



The First parse tree is preferred with the rule

Match each else statement with closest unmatched then

DANGLING ELSE – UNAMBIGUOUS GRAMMAR

- Rewrite the dangling else grammar as unambiguous grammar
- The grammar will be

Stmt → *matchedstmt*

| *Openstmt*

***Matchedstmt* → if expr then matchedstmt else matchedstmt**

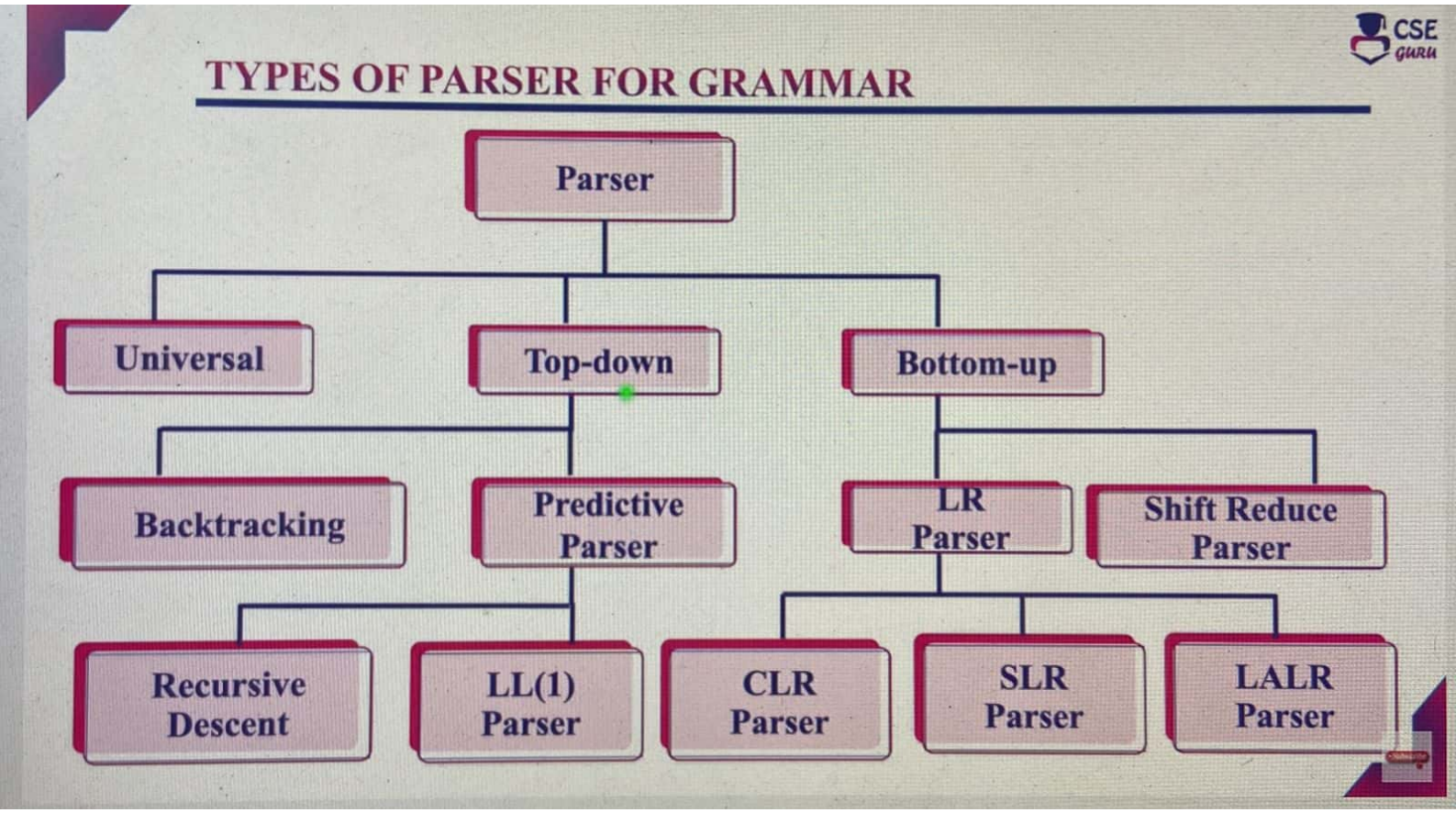
| other

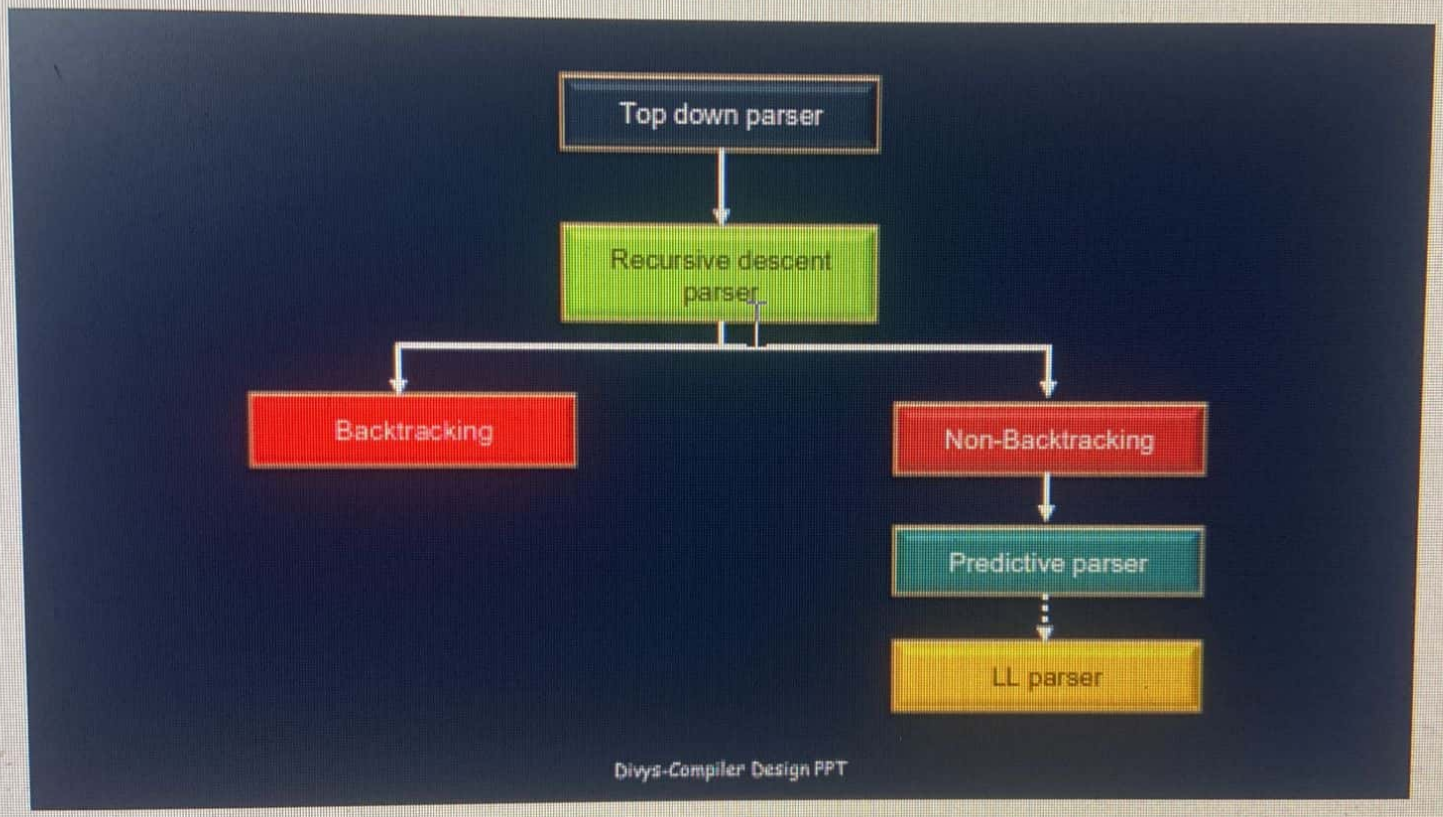
***Openstmt* → if expr then stmt**

| if expr then matchedstmt else openstmt

Matchedstmt

If E1 then if E2 then S1 else S2

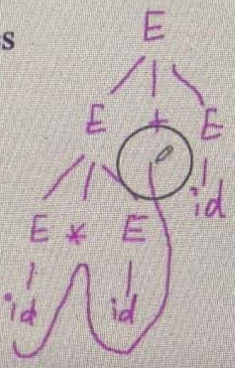




TOP-DOWN PARSER

- Constructs parse tree from **top to bottom** i.e root to the leaves
- Generates string starting from start symbol(Root) and repeatedly applies production rules for non-terminals until gets terminals
- Constructs **parse tree** as the output
- Every step decision is "what is the production to apply and **Expand**"
- Attempts to find **Left Most Derivation (LMD)** of deriving the string
- Creates nodes of parse tree as per **Pre-order**
- Popular Top down parser is **LL(1) Grammar**

$E \rightarrow E+E \mid E*E \mid id$



TOP-DOWN PARSER

Objective

Constructs parse tree starting with root node and proceeds downward to generate a string that matches the given input

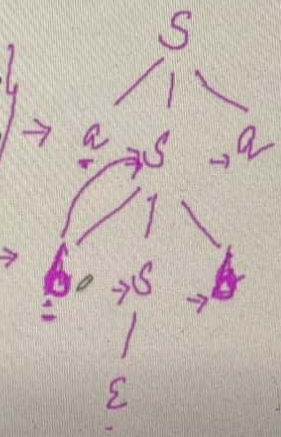
Steps

1. Parsing starts with root node & it is a Non-terminal
2. Apply production rule for Non-terminal and Expand
3. Parser scans input from left to right, one token at a time
4. Apply production rule for Leftmost Non terminal
   i. If derives terminal and match with input string, advance the pointer
   ii. It derives Non-Terminal, apply any one production rule and expand
5. If mismatch, backtrack and apply other production rule and check
6. Procedure continues until matches with complete input string

Example

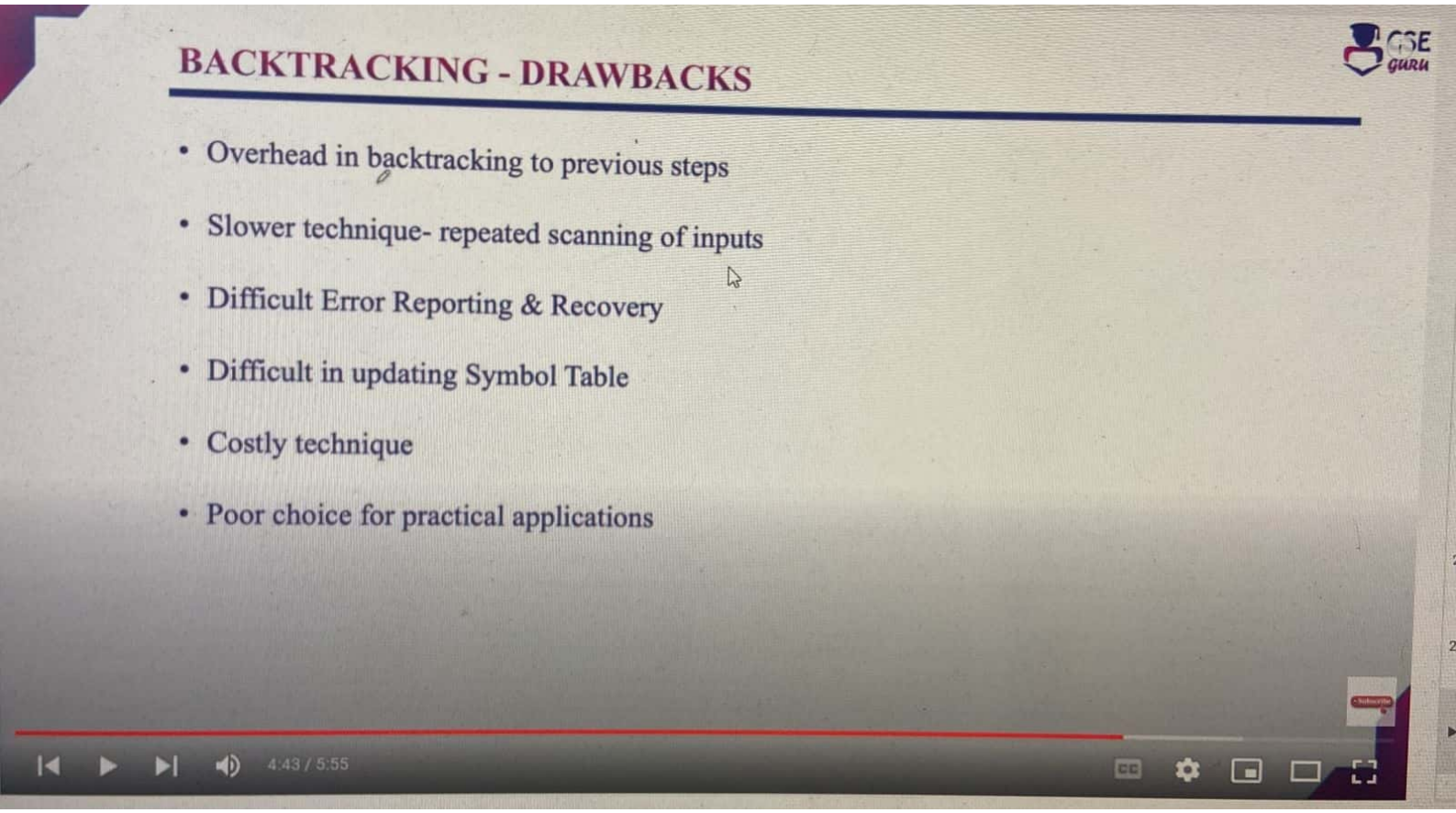
P = { S->aSa,
S->bSb,
S->ε
}

W= abba



BACKTRACKING

- It is a **Top-Down Parsing** Technique
- Technique starts with the **start symbol**
- Apply the **Production rule for the non-terminal** to get the desired string
- During parsing if there is a **mismatch**(wrong derivation), **backtrack** to previous step and apply the next possible production rule and check
- Process continues until get the desired string
- Backtracking is powerful because it is suitable for any kind of grammar



LEFT RECURSION

A Grammar G is left recursive Grammar if the non-terminal A in the derivation is of the form:

$$A \overset{+}{\Rightarrow} A\alpha$$

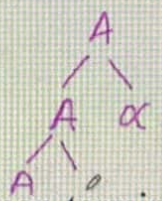
Where α is a string of terminals and non-terminals.

Whenever the first symbol in the **right hand side of the production is same as the left hand side variable**, then the grammar is said to be a **left recursive grammar**.

LEFT RECURSION

A Grammar G is left recursive Grammar if the non-terminal A in the derivation is of the form:

$$A \overset{+}{\Rightarrow} A\alpha$$



Where α is a string of terminals and non-terminals.

Whenever the first symbol in the **right hand side of the production is same as the left hand side variable**, then the grammar is said to be a **left recursive grammar**.

IMMEDIATE LEFT RECURSION

A Grammar is said to be left recursive grammars, if the first symbol in the right hand side of the production is same as the left hand side variable

Example:

$E \rightarrow E+T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

In this grammar, the first two productions

$E \rightarrow E+T$

$T \rightarrow T * F$

INDIRECT LEFT RECURSION

A Grammar is said to be Indirect left recursive grammar, if the first symbol in the right hand side of any of its derivations is same as the left hand side variable

Example:

$E \rightarrow T$

$T \rightarrow F$

$F \rightarrow E+T \mid id$

$E \rightarrow T \rightarrow F \rightarrow E+T$

Here the indirect derivation is

$$E \Rightarrow T \Rightarrow F \Rightarrow E+T$$

The partial derivation E + T contains the first symbol E same as the LHS.

LEFT RECURSION

The left recursive grammar is of the form

$$A \Rightarrow A\alpha \mid \beta$$

First Production

- **left recursive grammar** - first symbol in the right-hand side of the production is same as the left-hand side variable
- A is a non-terminal
- α is a string of terminals and non-terminals

Second Production

- β is a string of terminals and non-terminals

ELIMINATING LEFT RECURSION

CSE GURU

- **Top-down parsers cannot handle left recursion**
- The left recursive grammar is of the form

$$A \Rightarrow A\alpha \mid \beta$$

- Steps to replace left recursive productions as non-left recursive productions

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' \mid \varepsilon$$

$S \rightarrow Sa \mid b$

$S \rightarrow bS'$

$S' \rightarrow aS' \mid \varepsilon$

ELIMINATING LEFT RECURSION

- The left recursive grammar is of the form

$$A \Rightarrow A\alpha_1 \mid A\alpha_2 \ldots \mid \beta_1 \mid \beta_2 \ldots\ldots$$

- Steps to replace left recursive productions as non-left recursive productions

$$\begin{cases} A \rightarrow \beta_1 A' \mid \beta_2 A' \ldots\ldots \\ A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \ldots \mid \varepsilon \end{cases}$$

$S \rightarrow Sa \mid Sb \mid c \mid d$

$A \quad \alpha_1 \quad A \quad \alpha_2 \quad \beta_1 \quad \beta_2$

$$\begin{cases} S \rightarrow cS' \mid dS' \\ S' \rightarrow aS' \mid bS' \mid \varepsilon \end{cases}$$

6:36 / 6:56

ELIMINATING IMMEDIATE LEFT RECURSION

CSE GURU

$E \rightarrow E+T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

① $E \rightarrow E+T \mid T$

$A \quad A \quad \alpha \quad \beta$

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

② $T \rightarrow T*F \mid F$

$A \quad A \quad \alpha \quad \beta$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

$A \rightarrow A\alpha \mid \beta$

$\Downarrow$

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \varepsilon$

Final Grammar

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

$F \rightarrow (E) \mid id$

ELIMINATING INDIRECT LEFT RECURSION

$S \rightarrow Aa \mid b$

$A \rightarrow Ac \mid Sd \mid \varepsilon$

$S \Rightarrow Aa \rightarrow Sda$

$A \rightarrow A\alpha_1 \mid A\alpha_2 \cdots \mid \beta_1 \mid \beta_2 \cdots$

$A \rightarrow \beta_1 A' \mid \beta_2 A' \cdots$

$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \varepsilon$

$A \rightarrow Ac \mid Sd \mid \varepsilon$

$A \rightarrow Ac \mid Aad \mid bd \mid \varepsilon$

$A \rightarrow bdA' \mid \varepsilon A'$

$A' \rightarrow cA' \mid adA' \mid \varepsilon$

Final Grammar

$S \rightarrow Aa \mid b$

$A \rightarrow bdA' \mid A'$

$A' \rightarrow cA' \mid adA' \mid \varepsilon$

5:19 / 5:40

ELIMINATING IMMEDIATE LEFT RECURSION

Consider the following grammar and eliminate left recursion

$S \rightarrow (L) / a$

$L \rightarrow L , S / S$

$L \rightarrow L , S \;/\; S$

$\downarrow \quad \downarrow \quad \downarrow \quad \downarrow$

$A \quad A \quad \alpha \quad \beta$

$L \rightarrow SL'$

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

$A \rightarrow A\alpha / \beta$

$\Downarrow$

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' / \varepsilon$

Final Grammar

$S \rightarrow (L) / a$

$L \rightarrow SL'$

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

ELIMINATING INDIRECT LEFT RECURSION

Consider the following grammar and eliminate left recursion

$X \rightarrow XSb / Sa / b$

$S \rightarrow Sb / Xa / a$

$S \rightarrow Xa \rightarrow Saa$

Eliminated LR from X

$X \rightarrow Sax' | bx'$

$x' \rightarrow Sbx' | \varepsilon$

$S \rightarrow Sb | Sax'a | bx'a | a$

$A \Rightarrow A\alpha_1 | A\alpha_2 ... | \beta_1 | \beta_2 .....$

$\downarrow$

$A \rightarrow \beta_1 A' | \beta_2 A' .....$

$A' \rightarrow \alpha_1 A' | \alpha_2 A' ... | \varepsilon$

① $X \rightarrow XSb / Sa / b$

$X \rightarrow Sax' | bx'$

$x' \rightarrow Sbx' | \varepsilon$

② $S \rightarrow Sb | Sax'a | bx'a | a$

$S \rightarrow bx'as' | as'$

$S' \rightarrow bs' | ax'as' | \varepsilon$

Final Grammar

$X \rightarrow Sax' | bx'$

$x' \rightarrow Sbx' | \varepsilon$

$S \rightarrow bx'as' | as'$

$S' \rightarrow bs' | ax'as' | \varepsilon$

LEFT FACTORING

- Left factoring is a grammar transformation helps to produce suitable grammar for Predictive parsing
- It is the process of converting non-deterministic grammar to Deterministic Grammar
- **Basic Idea:** When two alternative productions are available to expand a particular non-terminal with same prefix, there is a confusion to choose which production to apply for a non-terminal A.
- So rewrite the A-productions in-order to make right choice

LEFT FACTORING

Consider there are two A-productions

$$A \rightarrow \alpha\beta_1 | \alpha\beta_2 \ldots\ldots | r_1 | r_2 \ldots\ldots$$

Which production to apply for the non-terminal A, $\alpha\beta_1$ or $\alpha\beta_2$

Convert the above production as

$$A \rightarrow \alpha A' | r_1 | r_2$$

$$A' \rightarrow \beta_1 | \beta_2 | \ldots\ldots$$

LEFT FACTORING : EXAMPLE

$$\begin{cases} S \rightarrow iEtS \mid iEtSeS \mid a \\ E \rightarrow b \end{cases}$$

$\alpha \quad \beta_1 \quad \alpha \quad \beta_2 \quad \gamma_1$

$S \rightarrow iEtSS' \mid a$

$S' \rightarrow \varepsilon \mid eS$

Final Grammar

$$\begin{cases} S \rightarrow iEtSS' \mid a \\ S' \rightarrow \varepsilon \mid eS \\ E \rightarrow b \end{cases}$$

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \ldots\ldots \mid r_1 \mid r_2 \ldots\ldots$

$$\Downarrow$$

$$\begin{cases} A \rightarrow \alpha A' \mid r_1 \mid r_2 \\ A' \rightarrow \beta_1 \mid \beta_2 \mid \ldots\ldots \end{cases}$$

7:13 / 7:36

LEFT FACTORING : EXAMPLE

$S \rightarrow a / abSb / aA$

$A \rightarrow bS / aAAb$

$S \rightarrow a \mid abSb \mid aA$

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

$S \rightarrow aS'$

$S' \rightarrow \varepsilon \mid bSb \mid A$

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \ldots\ldots \mid r_1 \mid r_2 \ldots\ldots$

$\Downarrow$

$A \rightarrow \alpha A' \mid r_1 \mid r_2$

$A' \rightarrow \beta_1 \mid \beta_2 \mid \ldots\ldots$

Final Grammar

$S \rightarrow aS'$

$S' \rightarrow \varepsilon \mid bSb \mid A$

$A \rightarrow bS \mid aAAb$

LL(1) PARSER

- Top-down parser
- Non-Recursive Parser / Parser without backtracking / Table driven Parser
- Technique predicts the **right alternative** to expand non-terminal
- LL(1)
  - L – Left to Right scan
  - Left most derivation
  - 1 – using one lookahead symbol to making parsing decision
- Implements parsing by explicit stack

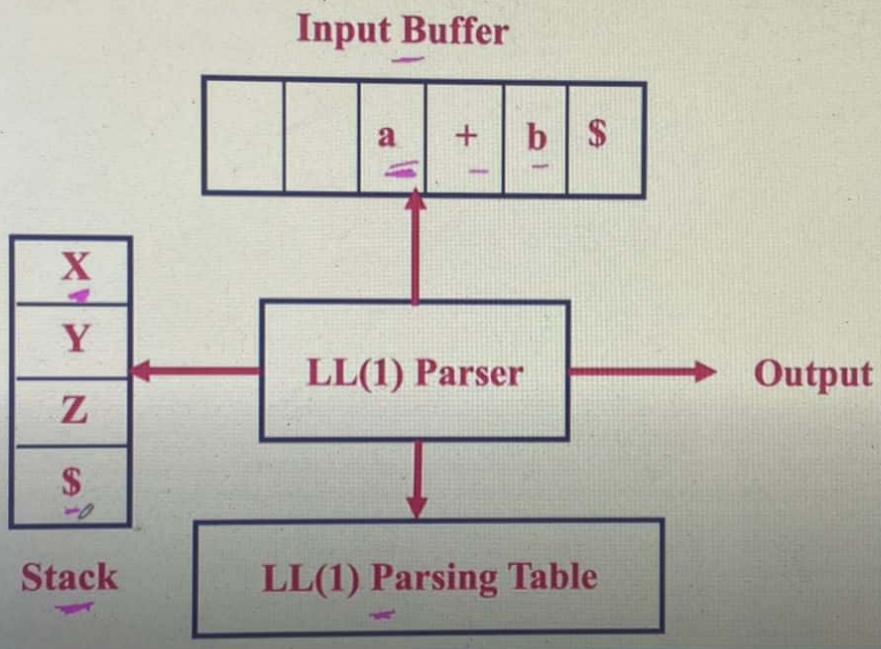
LL(1) PARSER

- Top-down parser
- Non-Recursive Parser / Parser without backtracking / Table driven Parser
- Technique predicts the **right alternative** to expand non-terminal
- LL(1)
  - L – Left to Right scan
  - Left most derivation
  - 1 – using one lookahead symbol to making parsing decision
- Implements parsing by explicit stack

LL(1) PARSER RESTRICTIONS

- Ambiguity
- Left Recursion
- Left Factoring

LL(1) PARSER – BLOCK DIAGRAM



Data Structures
1. I/p buffer
2. stack
3 parsing table

4:40 / 5:01

FIRST() FUNCTION – LL(1) PARSER

Definition

If S is any string of grammar symbols, then FIRST(S) is the set of **terminal symbols** that begin with the string derived from S.

i.e the beginning terminal symbol in the RHS of the production is FIRST(S)

Example

$S \rightarrow aA \mid bB \mid \varepsilon$

$FIRST(S) = \{a, b, \varepsilon\}$

$S \rightarrow Aab$

$A \rightarrow c \mid d$

$FIRST(S) = FIRST(A)$

$= \{c, d\}$

4:30 / 8:32

FIRST() FUNCTION – RULES

1. **RULE 1: If s is a terminal symbol, then FIRST(s) is {s}**
2. **RULE 2: If S → ε is a production, then add ε in FIRST(S)**
3. **RULE 3: If S is a NT, and S→ABCD.... is a production then**
   - i. **FIRST(S) = FIRST(A)**
   - ii. **If A→ ε , then also include FIRST(S) = FIRST(B) and so on**

$S \xrightarrow{\varepsilon} ABd$
$A \to a \mid \varepsilon$
$B \to c$

$FIRST(S) = FIRST(A) = \{a, \varepsilon\}$
$FIRST(S) = FIRST(B) = \{c\}$
$FIRST(S) = \{a, \varepsilon, c\}$

FIRST() FUNCTION IN LL(1) PARSER

E → TE'
E' → +TE'/ ε
T → FT'
T' → *FT'/ ε
F → (E)/id

FIRST(E) = FIRST(T) = FIRST(F) = {c, id}
FIRST(E') = {+, ε}
FIRST(T) = FIRST(F) = {c, id}
FIRST(T') = {*, ε}
FIRST(F) = {c, id}

1. **RULE 1:** If s is a terminal symbol, then FIRST(s) is {s}
2. **RULE 2:** If S → ε is a production, then add ε in FIRST(S)
3. **RULE 3:** If S is a NT, and S → ABCD.... is a production then
   i. FIRST(S) = FIRST(A)
   ii. If A → ε, then also include FIRST(S) = FIRST(B) and so on

FOLLOW() FUNCTION – LL(1) PARSER

DEFINITION

For any Non-Terminal S, the set of **terminals a** which **appears** immediately to the **right of S** is FOLLOW(S)

$S \rightarrow aAb$

$A \rightarrow e \mid \varepsilon$

$w = ab$

$Follow(A) = \{b\}$

$S \rightarrow aAb$

$\rightarrow$

FOLLOW() FUNCTION – LL(1) PARSER

DEFINITION

For any Non-Terminal S, the set of **terminals a** which **appears** immediately to the **right of S** is FOLLOW(S)

$S \rightarrow iEtS \mid iEtSeS \mid a$

$E \rightarrow b$

FOLLOW() FUNCTION – RULES

1. **RULE 1:** If S is start symbol, then include $ in FOLLOW(S)
2. **RULE 2:** If there is a production $S \rightarrow \alpha B \beta$, then include everything in FIRST(β) except ε in FOLLOW(B)
3. **RULE 3:** If there is a production $S \rightarrow \alpha B$ or $S \rightarrow \alpha B \beta$ where FIRST(β) = ε, then everything in FOLLOW(S) is in FOLLOW(B)

FOLLOW() FUNCTION IN LL(1) PARSER

E → TE'

E' → +TE'/ ε

T → FT'

T' → *FT'/ ε

F → (E)/id

S → AB

FIRST(E') = {+, ε}

FIRST(T') = {*, ε}

FOLLOW(E) = {), $}

FOLLOW(E') = {), $}

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

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

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

1. **RULE 1:** If S is start symbol,
   then include $ in FOLLOW(S)
2. **RULE 2:** If there is a production S→$\alpha B\beta$,
   then include everything in FIRST(β)
   except ε in FOLLOW(B)
3. **RULE 3:** If there is a production S→αB or S→$\alpha B\beta$
   where FIRST(β) = ε,
   then everything in FOLLOW(S) is in FOLLOW(B)

FIRST() & FOLLOW() FUNCTION

CSE GURU

S → (L) / a

L → L,S / S

Grammar After Eliminating Left Recursion

S → (L) | a

L → SL'

L' → ,SL' | ε

	FIRST()	FOLLOW()
S	{(, a}	{FIRST(L') - ε, FOLLOW(L')} {, ,), $}
L	{(, a}	{)}
L'	{, , ε}	{FOLLOW(L)} {)}

COMPUTE FIRST() & FOLLOW()

S → iEtSS' / a

S' → eS / ε

E → b

	FIRST()	FOLLOW()
S	{i, a}	{e, $}
S'	{e, ε}	{e, $}
E	{b}	{t}

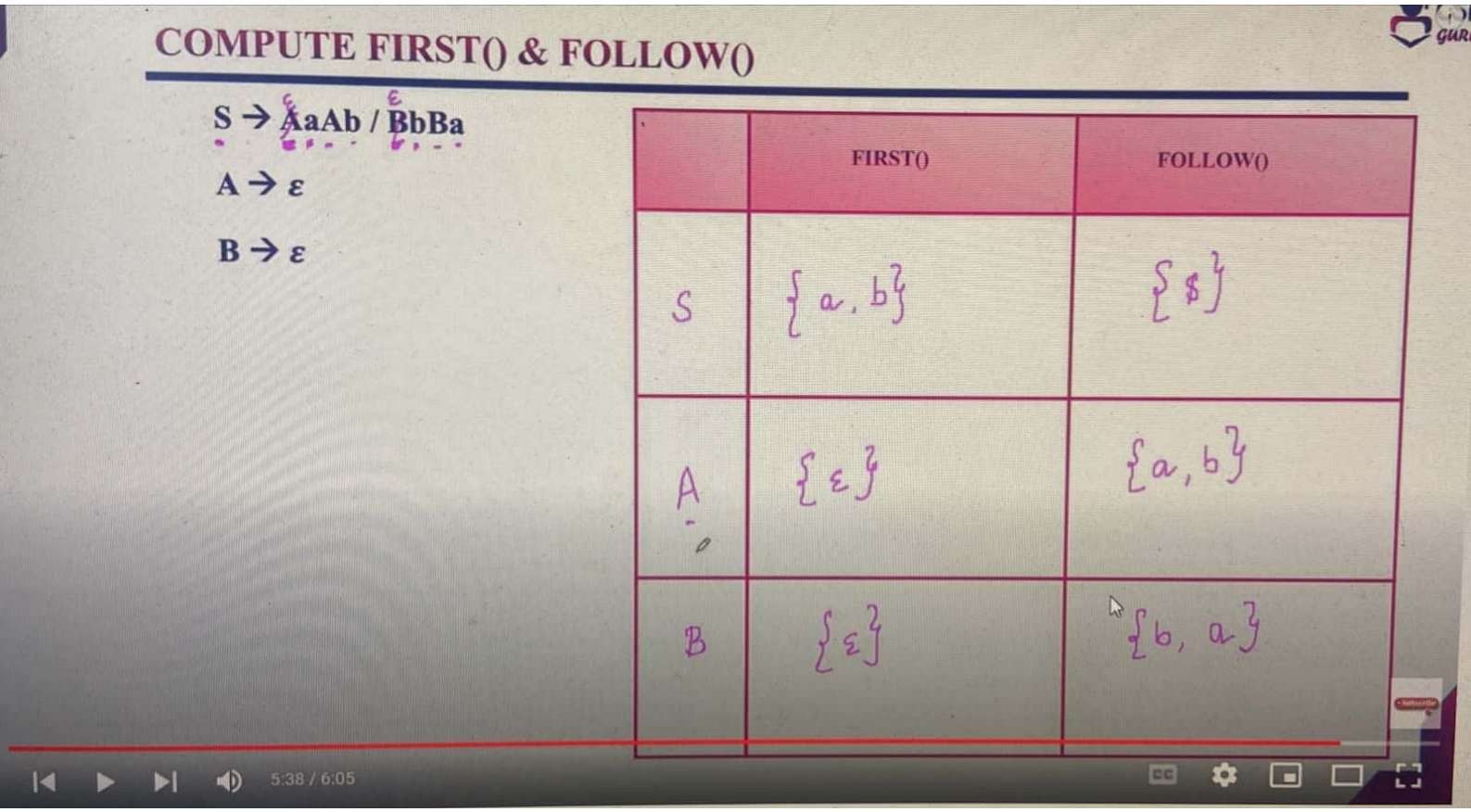
COMPUTE FIRST() & FOLLOW()

S → AaAb / BbBa

A → ε

B → ε

	FIRST()	FOLLOW()
S	{a, b}	{$}
A	{ε}	{a, b}
B	{ε}	{b, a}



CONSTRUCT LL(1) PARSING TABLE

E → TE'

E' → +TE'/ ε

T → FT'

T' → *FT'/ ε

F → (E)/id

	FIRST()	FOLLOW()
E	{(, id}	{), $}
E'	{+, ε}	{), $}
T	{(, id}	{+,), $}
T'	{*, ε}	{+,), $}
F	{(, id}	{+, *,), $}

LL(1) Parsing Table

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

CONSTRUCT LL(1) PARSING TABLE

S → (L) / a

L → SL'

L' → ,SL'/ ε

	FIRSTO	FOLLOWO
S	{ (, a }	{ $, , ,) }
L	{ (, a }	{) }
L'	{ , , ε }	{) }

LL(1) Parsing Table

	a	(	)	,	$
S	S → a	S → (L)			
L	L → SL'	L → SL'			
L'			L' → ε	L' → ,SL'	

L')

PARSING THE INPUT STRING

E → TE'
E' → +TE'/ ε
T → FT'
T' → *FT'/ ε
F → (E)/id

Parse the Input : **id + id * id**

LL(1) Parsing Table

	id	+	*	(	)	$
E	E → TE'			E → TE'		
E'		E' → +TE'			E' → ε	E' → ε
T	T → FT'			T → FT'		
T'		T' → ε	T' → *FT		T' → ε	T' → ε
F	F → id			F → id		

STACK	INPUT	ACTION
$ E	id + id * id $	
$ E'T	id + id * id $	E → TE'
$ E'T'F	id + id * id $	T → FT'
$ E'T'id	id + id * id $	F → id
$ E'T'	+ id * id $	pop id
$ E'	+ id * id $	T' → ε
$ E'T +	+ id * id $	E' → +TE'
$ E'T	id * id $	pop +
$ E'T'F	id * id $	T → FT'
$ E'T'id	id * id $	F → id
$ E'T'	* id $	pop id
$ E'T'F*	* id $	T' → *FT'
$ E'T'F	id $	pop *
$ E'T'id	id $	F → id
$ E'T'	$	pop id
$ E	$	T' → ε
$	$	E' → ε

PARSING THE INPUT STRING: (a , (a , a))

STACK	INPUT	ACTION
$ S	(a,(a,a))$	
$)L(	(a,(a,a))$	S → (L)
$)L	a,(a,a))$	pop (
$)L'S	a,(a,a))$	L → SL'
$)L'a	a,(a,a))$	S → a
$)L'	,(a,a))$	pop a
$)L'S,	,(a,a))$	L' → ,SL'
$)L'S	(a,a))$	pop ,
$)L')L(	(a,a))$	S → (L)
$)L')L	a,a))$	pop (
$)L')L'S	a,a))$	L → SL'
$)L')L'a	a,a))$	S → a
$)L')L'	,a))$	pop a
$)L')L'S,	,a))$	L' → ,SL'

STACK	INPUT	ACTION
$)L')L' S	a))$	pop ,
$)L')L' a	a))$	S → a
$)L')L'	))$	pop a
$)L')	))$	L' → ε
$)L'	)$	pop)
$)	)$	L' → ε
$	$	pop)

LL(1) Parsing Table

	a	(	)	,	$
S	S → a	S → (L)			
L	L → SL'	L → SL'			
L'			L' → ε	L' → ,SL'	