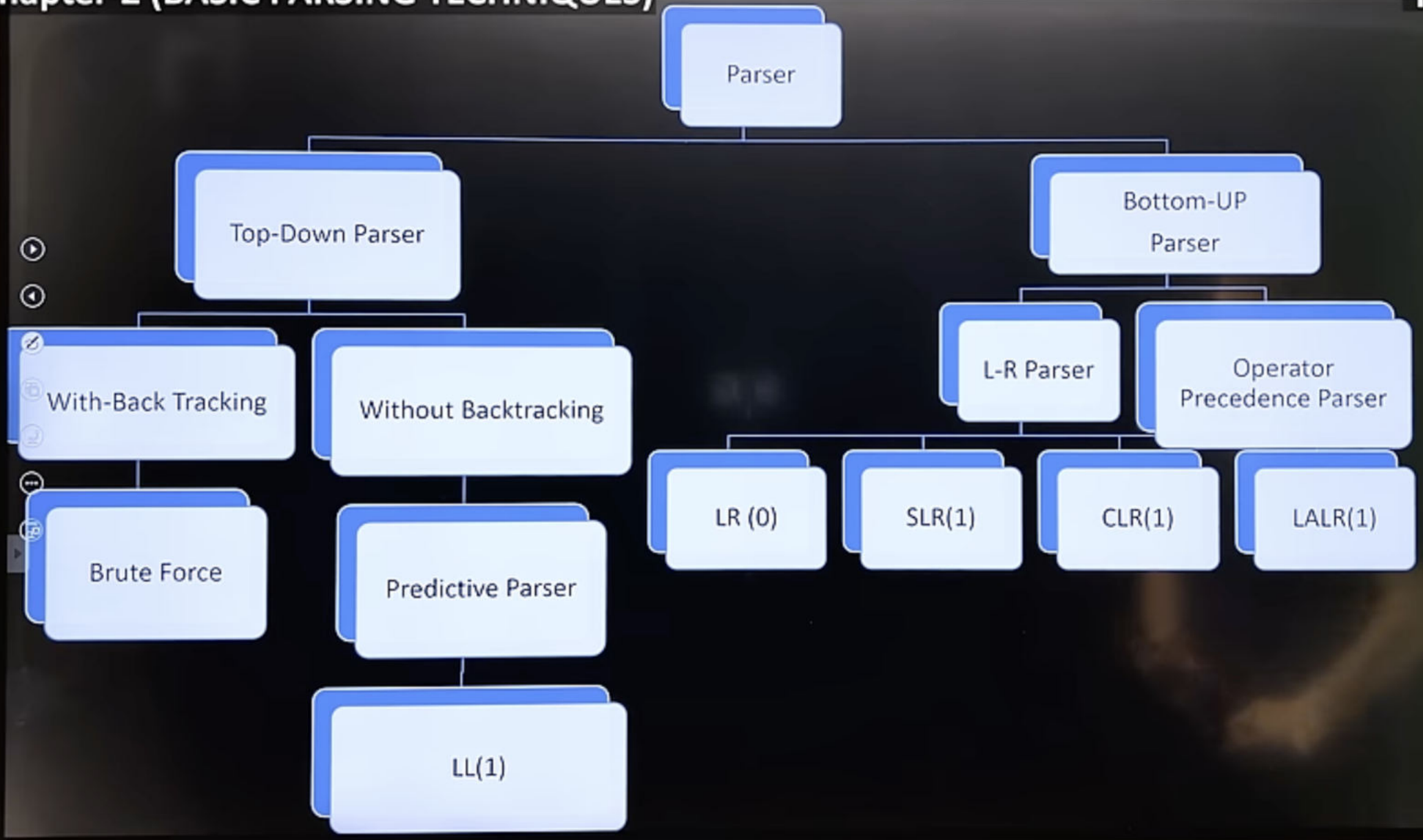
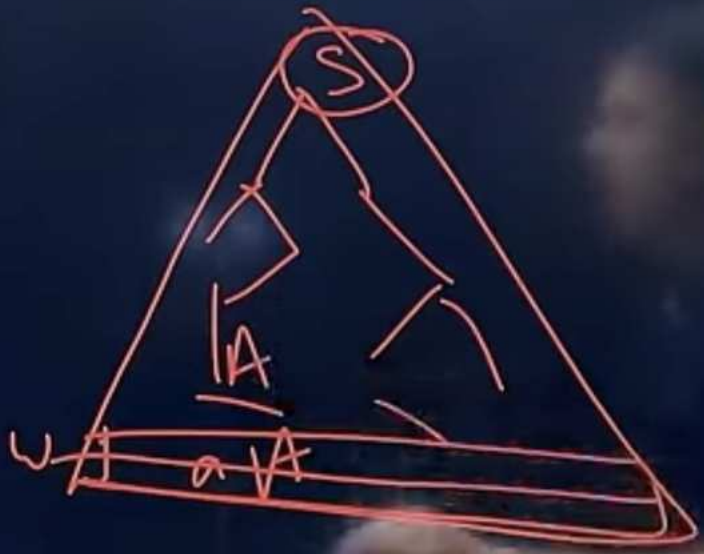

Parser
Top-Down Parser
Bottom-UP Parser
With-Back Tracking
Without Backtracking
L-R Parser
Operator Precedence Parser
Brute Force
Predictive Parser
LL(1)
LR (0)
SLR(1)
CLR(1)
LALR(1)

Top down parsing

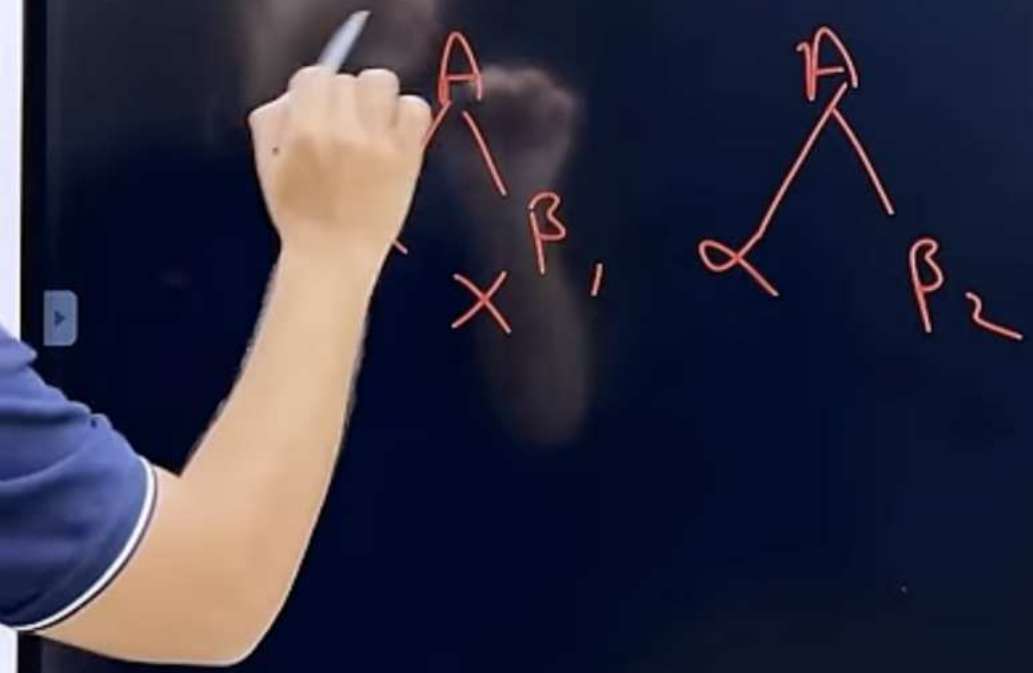
- The process of construction of parse tree, starting from root and process to children, is known as TOP down parsing, i.e. getting the i/p string by with a start symbol of the grammar is top down parsing.

$A \rightarrow aA / \in$



Non-Deterministic Grammar

- The grammar with common prefix is known as Non-Deterministic Grammar.
  - $A \rightarrow \alpha\beta_1 / \alpha\beta_2$



- **Left Factoring**: - The process of conversion of Non-Deterministic grammar into deterministic grammar is known as Left-Factoring.

- $A \rightarrow \alpha\beta_1 / \alpha\beta_2$
  - $A \rightarrow \alpha\beta$
  - $A \rightarrow \beta_1 / \beta_2$

$A \longrightarrow \alpha\beta_1 / \alpha\beta_2$

$A \longrightarrow \alpha\beta$

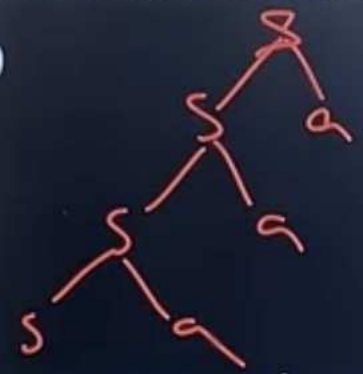
$\beta \longrightarrow \alpha_1 | \alpha_2$

Recursive production: - the production which has same variable both at left- and right-hand side of production is known as recursive production.

S→aSb

S→aS

S→Sa



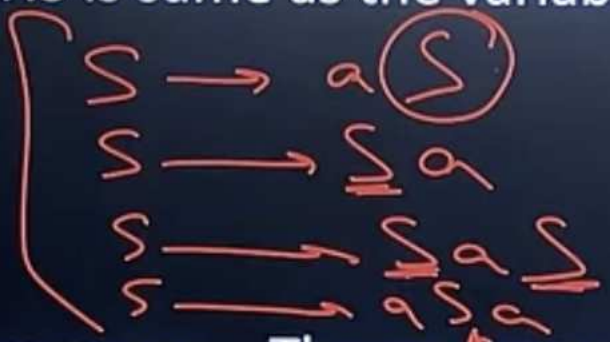
Recursive grammar: - the grammar which contains at least one recursive production is known as recursive grammar.

S→aS / a

S→Sa / a

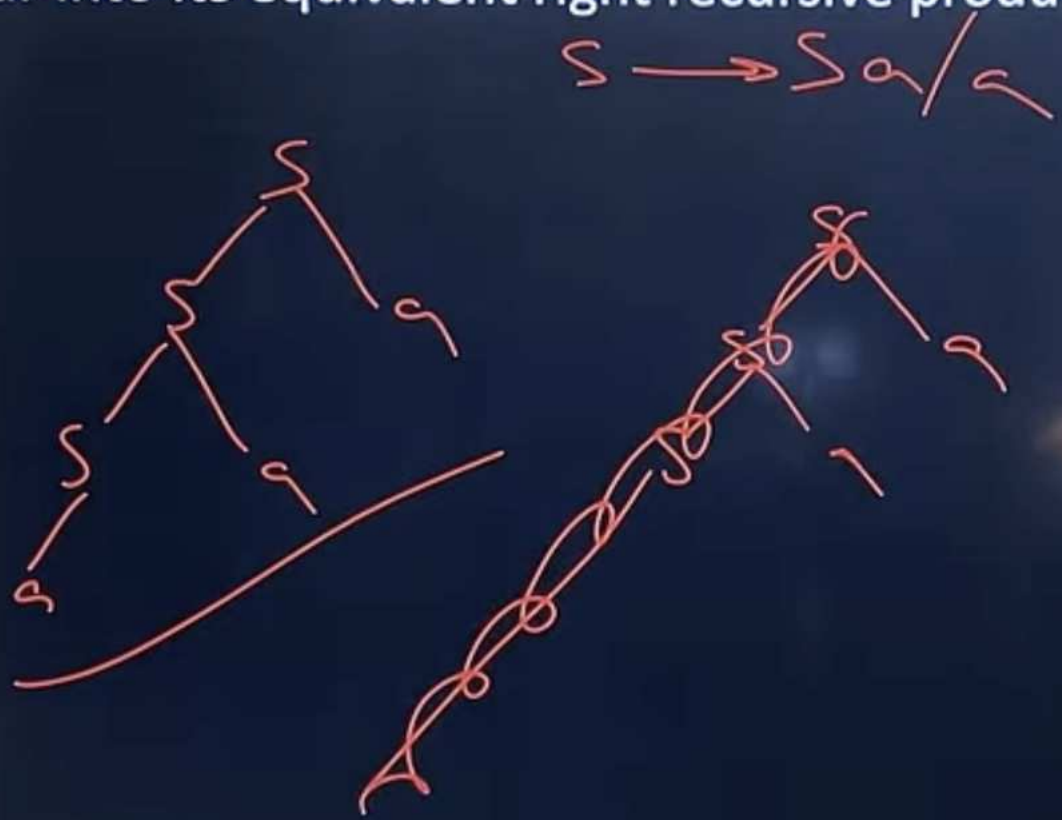
S→aSb / ab

Left Recursive Grammar: - The grammar G is said to be left recursive, if the Left most variable of RHS is same as the variable at LHS.



Right Recursive Grammar: - The grammar G is said to be right recursive, if the right most variable of RHS is same as the variable at LHS.

- If a CFG contains left recursion then the compiler may go to infinite loop, hence to avoid the looping of the compiler, we need to convert the left recursive grammar into its equivalent right recursive production.



Chapter-2 (BASIC PARSING TECHNIQUES)

Non-Deterministic Grammar

$$A \rightarrow A\alpha / \beta_1 / \beta_2 \text{ -------} / \beta_n$$

$$A \rightarrow \beta_1 A' / \beta_2 A' / - - - / \beta_n A'$$

$$A' \rightarrow \alpha A' / \epsilon$$

$$A \rightarrow A\alpha_1 / A\alpha_2 / \text{----------} A\alpha_n / \beta$$

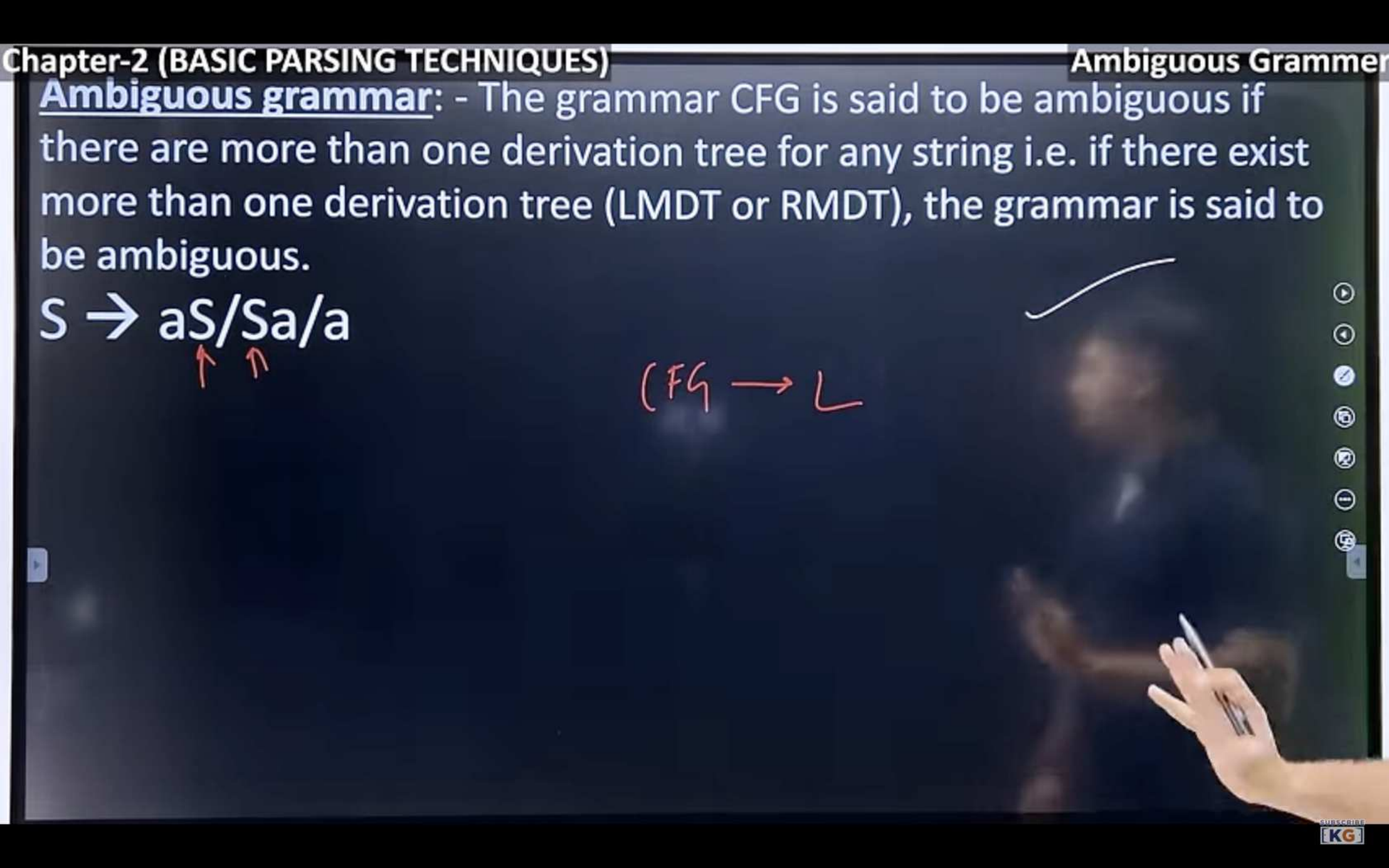
$$A \longrightarrow \beta A'$$

$$A' \longrightarrow \alpha_1 A' / \alpha_2 A' / \alpha_3 A' / \text{——} / \in$$

Ambiguous grammar: - The grammar CFG is said to be ambiguous if there are more than one derivation tree for any string i.e. if there exist more than one derivation tree (LMDT or RMDT), the grammar is said to be ambiguous.

$S \rightarrow aS/Sa/a$

CFG → L



Brute force technique

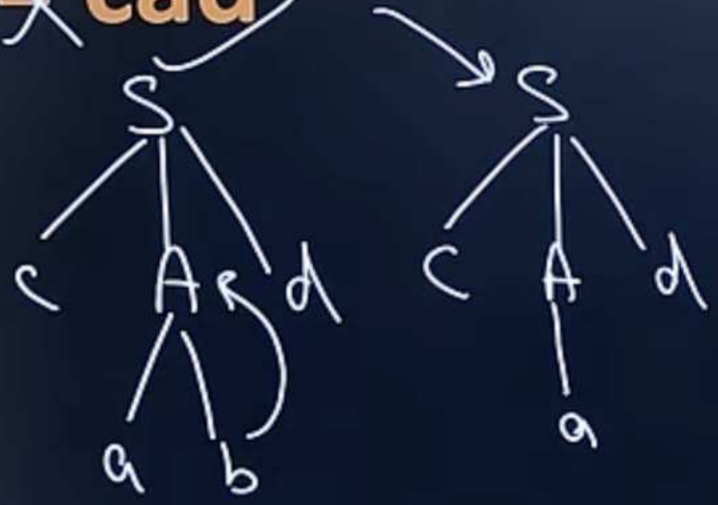
- Whenever a non- terminal is expanding first time, then go with the first alternative and compare with the i/p string. if does not matches, go for the second alternative and compare with i/p string, if does not matches go with the 3rd alternative and continue with each and every alternative.

- if the matching occurs for at least one alternative, then the parsing is successful, otherwise parsing fail.

$S \rightarrow cAd$

$A \rightarrow ab \mid a$

$w_1 = cad$



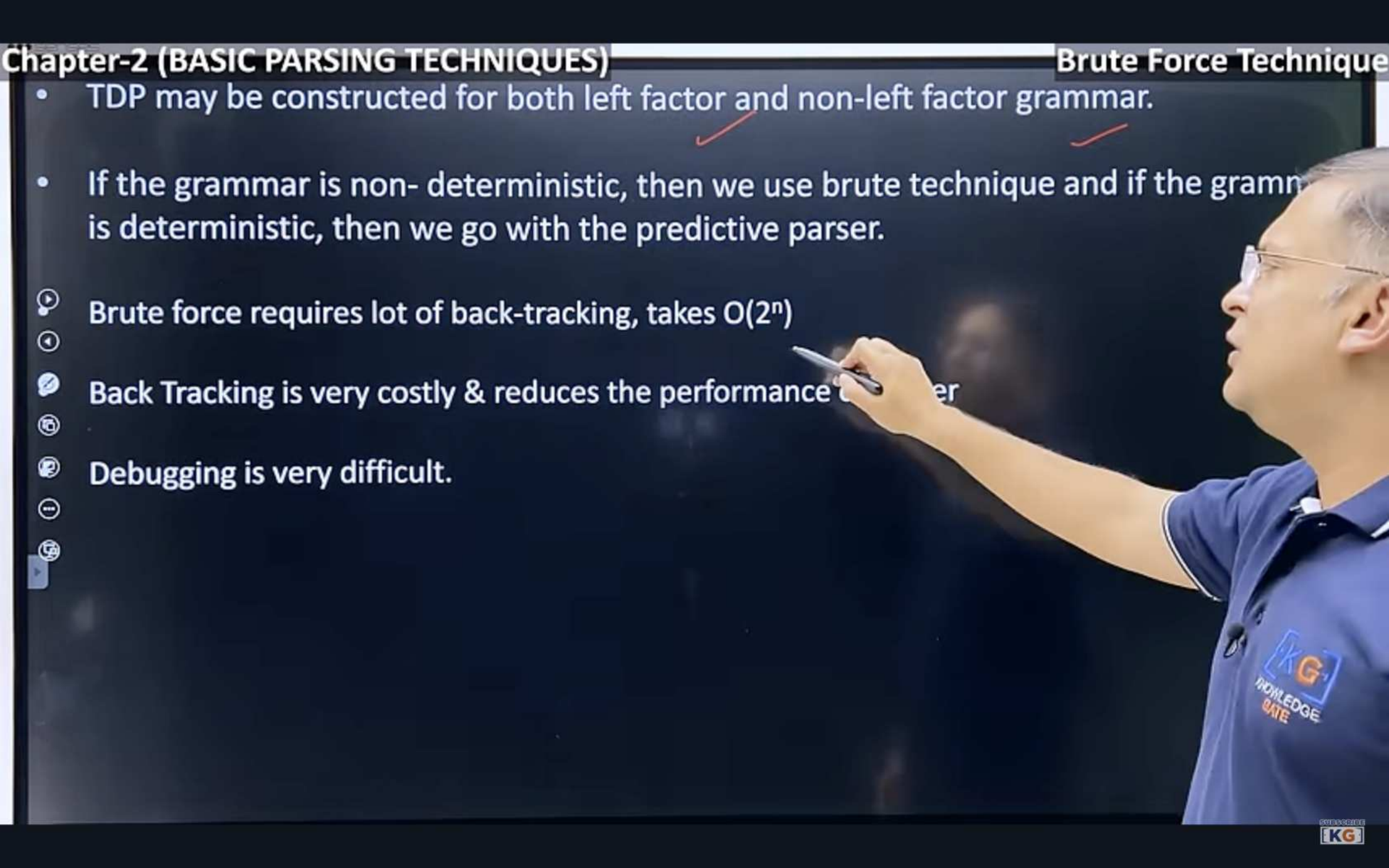
cabd

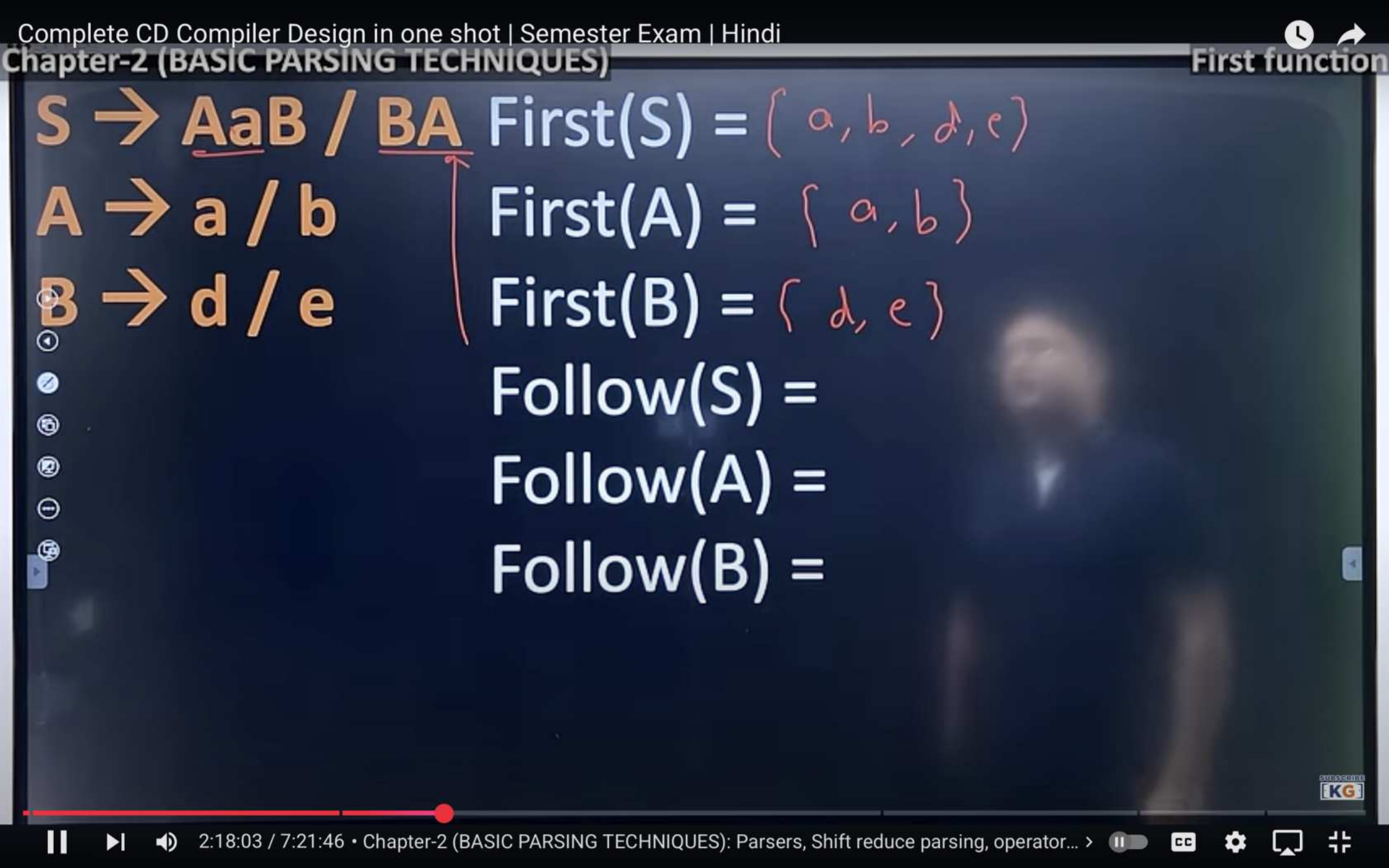
cad

$w_2 = cada$

Brute Force Technique

- TDP may be constructed for both left factor and non-left factor grammar.
- If the grammar is non- deterministic, then we use brute technique and if the gramm is deterministic, then we go with the predictive parser.
- Brute force requires lot of back-tracking, takes $O(2^n)$
- Back Tracking is very costly & reduces the performance er
- Debugging is very difficult.




Chapter-2 (BASIC PARSING TECHNIQUES)
First function
S → AaB / BA
A → a / b
B → d / e
First(S) = { a, b, d, e }
First(A) = { a, b }
First(B) = { d, e }
Follow(S) =
Follow(A) =
Follow(B) =

- First(α) is a set of all terminals that may be in beginning in any sentential form, derived from α
- FIRST(α) is calculated for both Terminals and Non-Terminals
- **if** α is a terminal, then
  - **First(α) = {α}**
- **if** α is a string of terminal e.g. abc
  - **First(abc) = {a}**

$F(a) = a$

- if α is a non - terminal, defined by α → ε, then
  - First(α) = {ε}

- if α is a non - terminal, defined by α → β, β ∈ T
  - First(α) = {β}

- if α is a non - terminal, defined by $\alpha \rightarrow X_1 X_2 X_3$, then
  - **First(α) = First(X_1)** iff $X_1 \rightarrow ! \in$
  - **First(α) = [First(X_1) U First(X_2)] - $\in$** iff $X_1 \rightarrow \in$ && iff $X_2 \rightarrow ! \in$
  - **First(α) = [First(X_1) U First(X_2) U First(X_3)] - $\in$** iff $X_1 \rightarrow \in$ && $X_2 \rightarrow \in$ && iff $X_3 \rightarrow ! \in$
  - **First(α) = First(X_1) U First(X_2) U First(X_2)** iff $X_1 \rightarrow \in$ && $X_2 \rightarrow \in$ && iff $X_3 \rightarrow \in$

$S \rightarrow AaBb \;/\; BbAa$

$A \rightarrow \varepsilon$

$B \rightarrow \varepsilon$

Fol (S) = { $ }

" (A) = { a }

" (B) = { b }

Follow

Follow(A) is the set of all terminals that may follow to the right of (A) in any form of sentential Grammar.

Rules:

1) if A is the start symbol then Follow(A) = {$}

2) if $A \rightarrow \alpha A \beta$, $\beta \rightarrow ! \in$

Follow(A) = First(β)

3) if $S \rightarrow \alpha A$

Follow(A) = Follow(S)

4) $S \rightarrow \alpha A \beta$, where $\beta \rightarrow \in$

Follow(A) = First(β) U Follow(S) - ∈

E → TE'

E' → +TE' / ∈

T → FT'

T' → *FT' / ∈

F → (E) / id

id	+	id	*	id	$

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

F(E) = { (, id }
" (E') = { +, ∈ }
" (T) = { (, id }
" (T') = { *, ∈ }
" (F) = { (, id }

Fol(E) = { $,) }
" (E') = { $,) }
" (T) = { +, $,) }
" (T') = { +, $,) }
" (F) = { *, +, $,) }

E → TE'

E' → +TE' / ∈

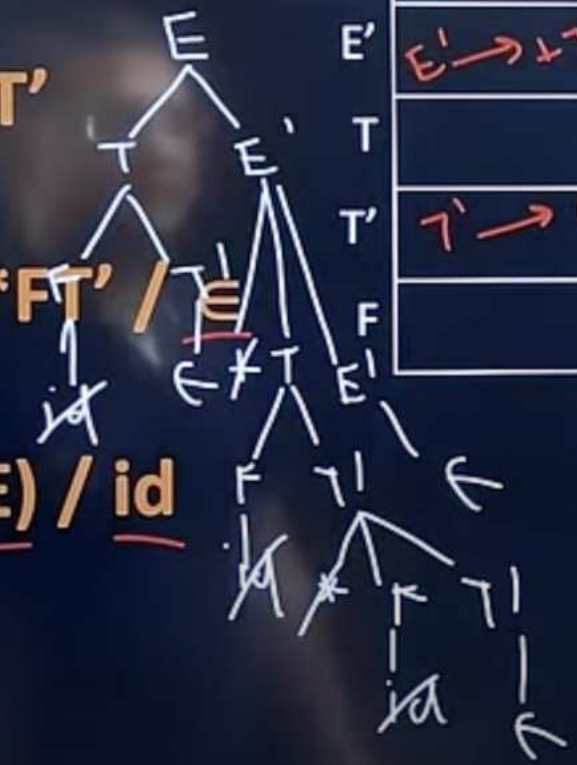
T → FT'

T' → *FT' / ∈

F → (E) / id

id	+	id	*	id	$

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



Q Consider a given Grammar LL(1) grammar design Parsing table and perform complete parsing table?

E → TE'

E' → +TE' / ∈

T → FT'

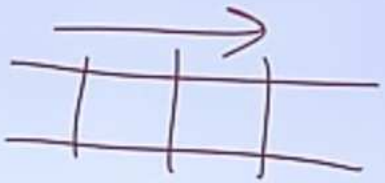
T' → *FT' / ∈

F → (E) / id

w = id+id*id

Stack	i/p	Action
$	id+id*id$	Push E
$ E	id+id*id$	Use Production E→TE' POP E and Push E'T
$ E'T	id+id*id$	Use Production T→FT' POP T and Push T'F
$ E'T'F	id+id*id$	Use Production F→id POP F and Push id
$ E'T'id	id+id*id$	Match Pop id and Increment Look Ahead Pointer
-	-	-
-	-	-
-	-	-
-	-	-
$	$	Accepted

- LL(1)
  - First L means Left to right scanning
  - Second L means Left most derivation
  - 1 means no of look ahead symbol



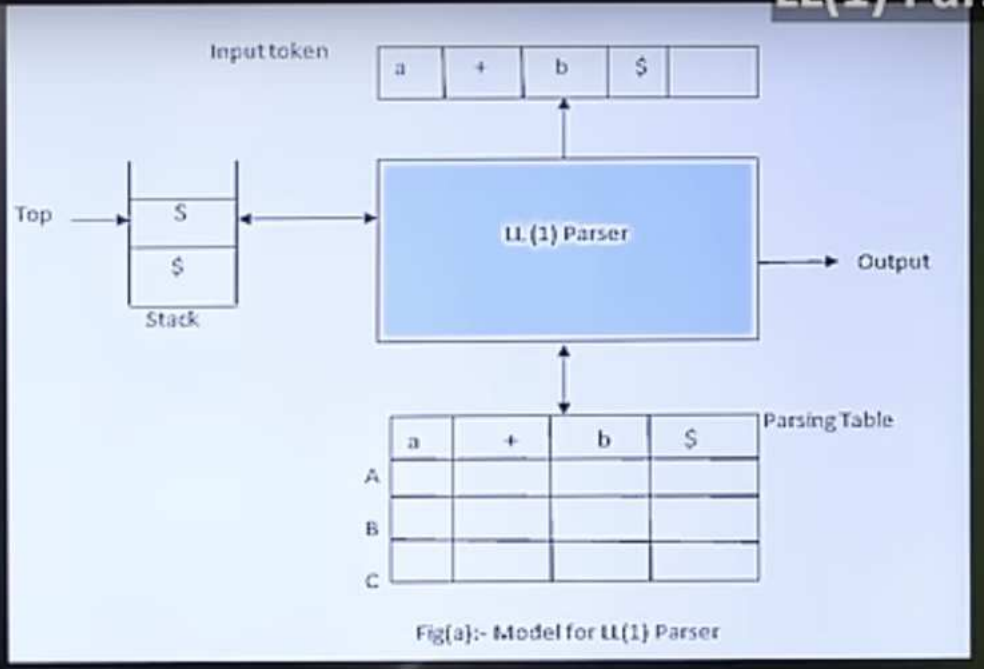
- To predict the required production, to extend the parse tree, LL(1) parser depends on current processing symbol.

- The current processing symbol is called as Look-ahead-symbol.

Chapter-2 (BASIC PARSING TECHNIQUES)

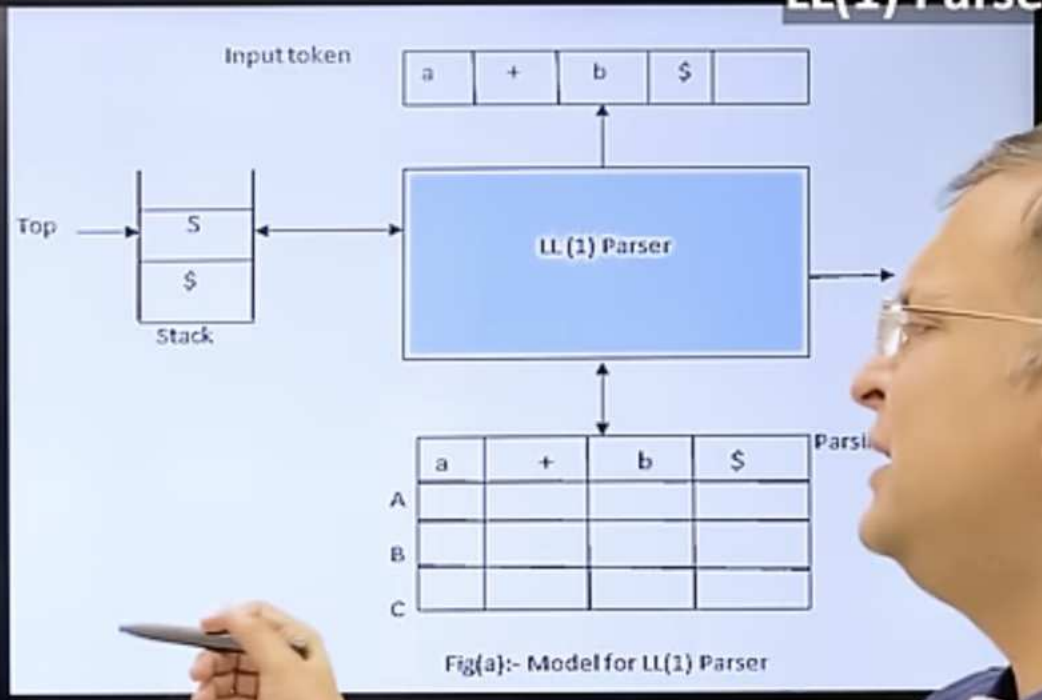
LL(1) Parser

- **Input Buffer**
  - it is divided into finite no of cells and each cell is capable of holding only one i/p symbol.
  - input buffer contains only i/p string at any point of time.
  - the tape header is always pointing only one look ahead symbol and after parsing the current look ahead symbol, the header moves to next cell towards right side.
  - End of the string is recognized by $

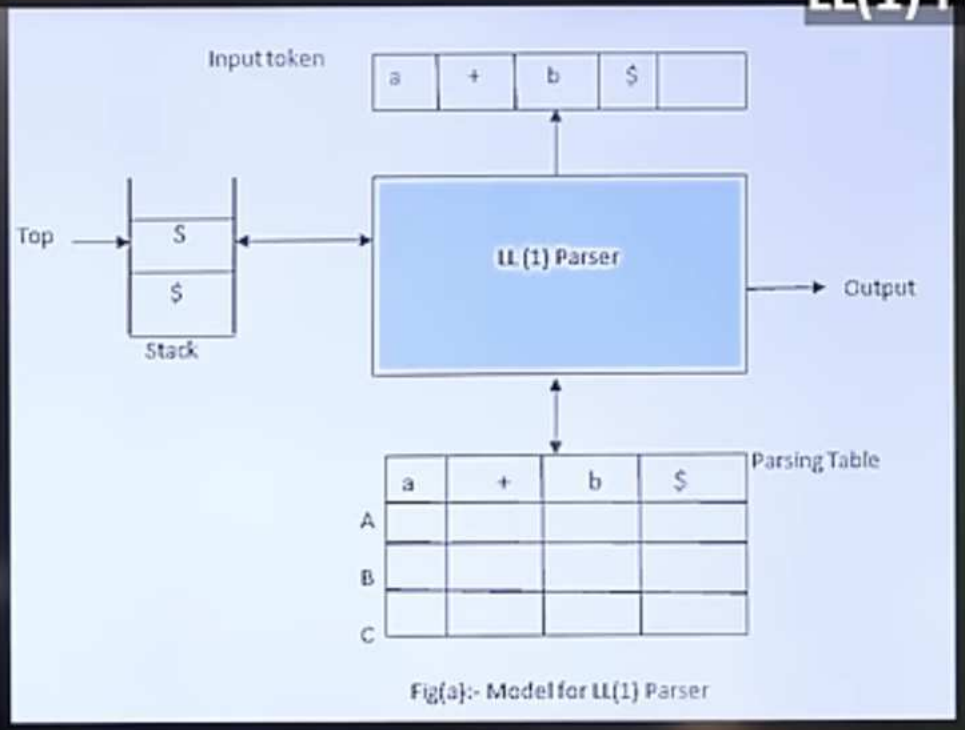


Fig(a):- Model for LL(1) Parser

- **Parse Stack**
  - it contains the grammar symbol, the grammar symbol is pushed into stack or POP from the stack based on the occurrence of matching.
  - if the topmost symbol of the stack is matching with look ahead symbol, then the grammar symbol is POP out from the stack.
  - if the TOP most symbol of the stack is not matching with the look ahead symbol, then the grammar symbol is Pushed into stack.

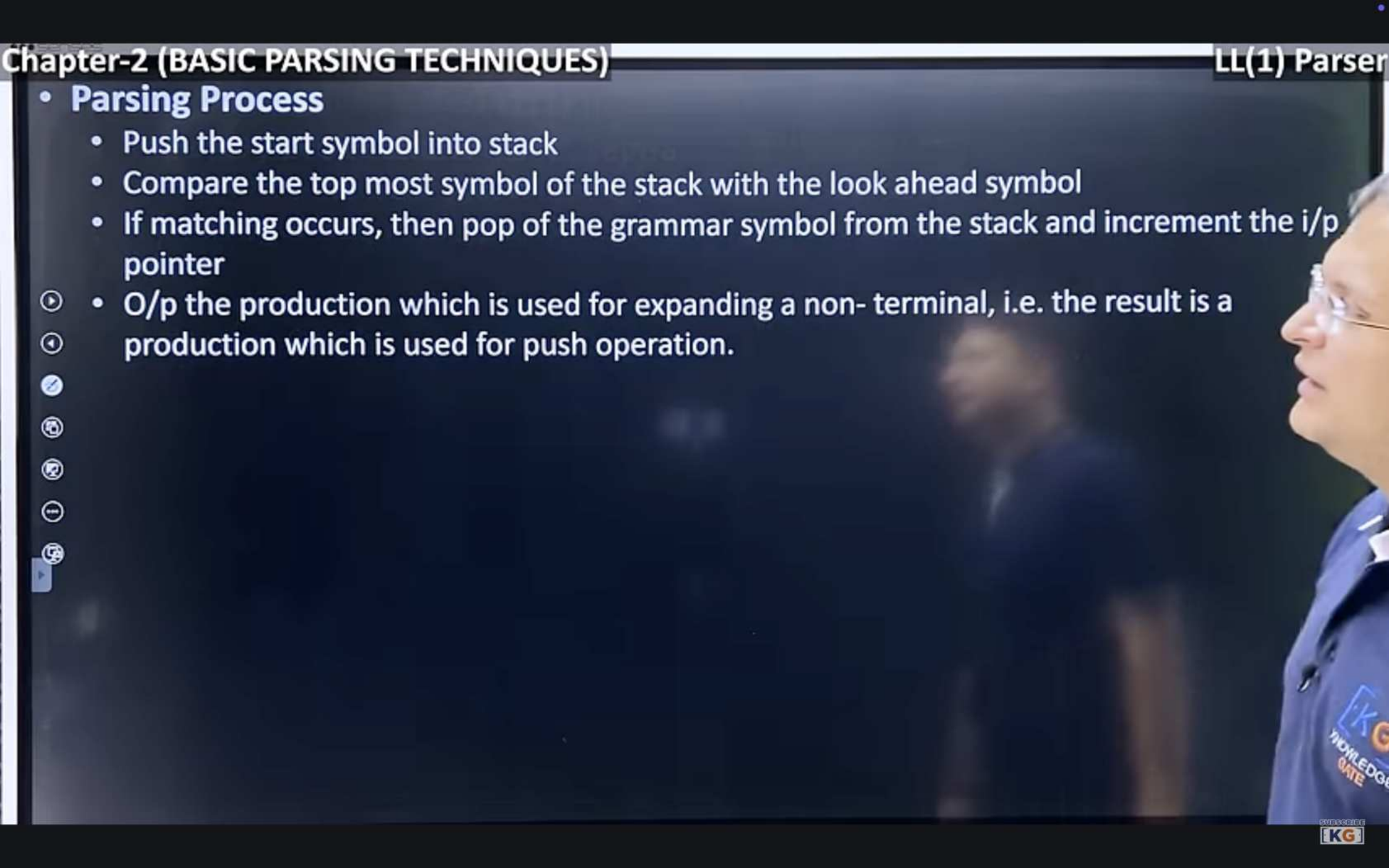


- **Parse table**
  - it is a two-dimensional array of order m*n where m = no of non-terminal and n = no of terminals + 1
  - parse table contains all the production which are used to contain the parse tree for that i/p string.



Fig(a):- Model for LL(1) Parser

- **Parsing Process**
  - Push the start symbol into stack
  - Compare the top most symbol of the stack with the look ahead symbol
  - If matching occurs, then pop of the grammar symbol from the stack and increment the i/p pointer
  - O/p the production which is used for expanding a non- terminal, i.e. the result is a production which is used for push operation.



Chapter-2 (BASIC PARSING TECHNIQUES) LL(1) Parser

- Procedure to constructed LL(1) parser Table:
  - for every production A → α, repeat the following steps
  - add A → α in M[A, α] for every terminal 'α' in first(A).
  - if First(α) contains ε, then add A → ε in M[A, b] for every symbol, b in Follow[A]

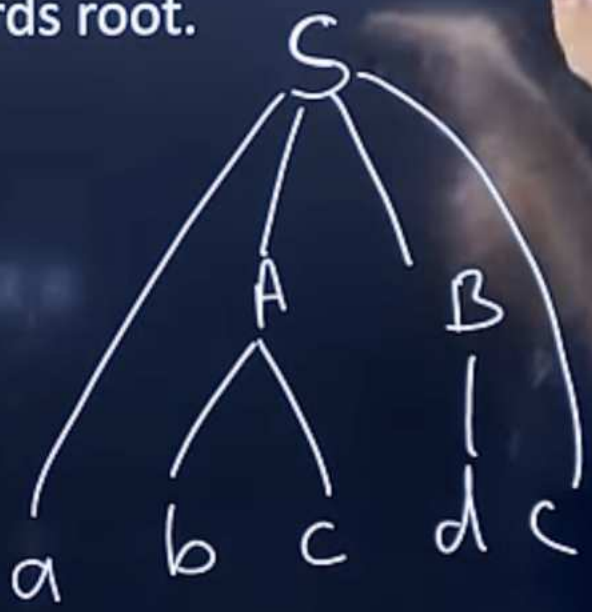
- **LL(1) parsing algorithm**
  - let x is a grammar symbol (stack symbol) and a is the look ahead symbol
  - if x == a == $, then the parsing is successful
  - if x==a! = $, then pop of and increment the i/p pointer
  - if x!= a!=$ and m[x, a] contain the production, x → abc, then replace x by abc in the reverse order and continue the process.
  - out of the production which is used for expanding the non-terminal, i.e. the production which is used for PUSH operation

Bottom Up parser

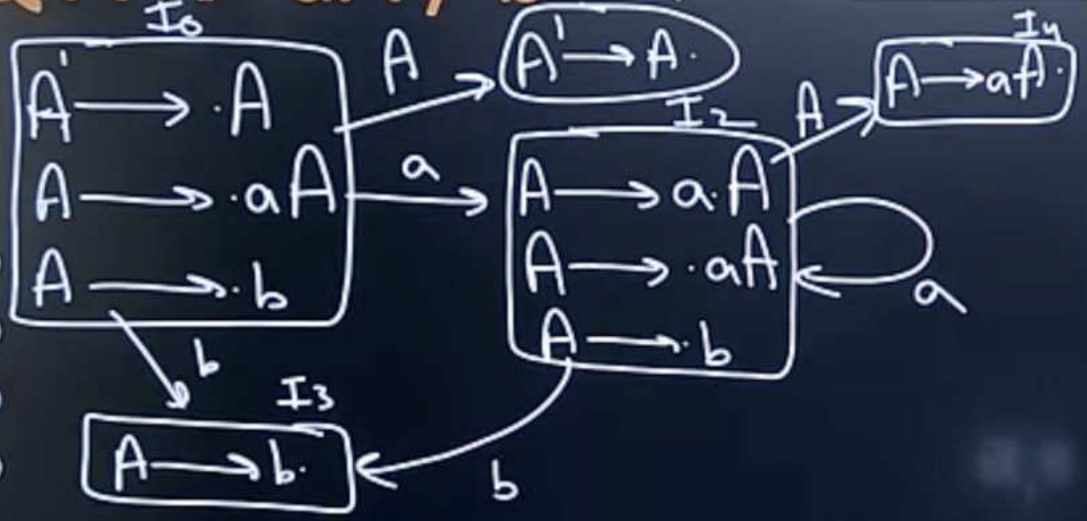
- The process of constructing the parse tree in the Bottom-Up manner, i.e. start from the children & proceeding towards root.

- S → aABc
  A → b / bc
  B → d

- w = abcdc



Q $A \rightarrow aA / b$ (r_1 / r_2)



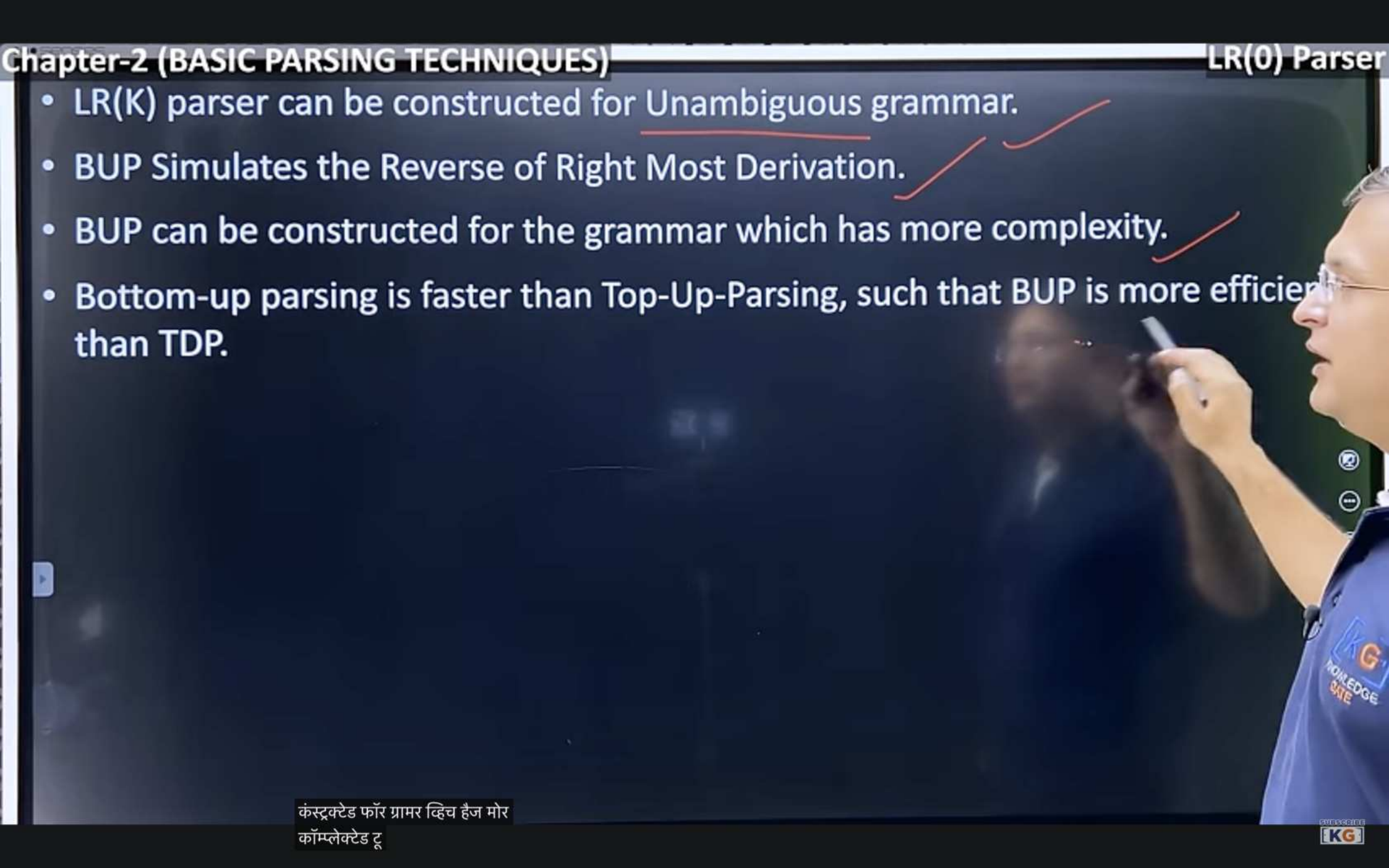
a	a	a	b	$

	Action			Goto
	a	b	$	A
I_0	S_2	S_3		1
I_1			accept	
I_2	S_2	S_3		4
I_3	r_2	r_2	r_2	
I_4	r_1	r_1	r_1	

हमने किया और इसका क्या-क्या मत मतलब है
अब आपको वो पूरा आईडिया

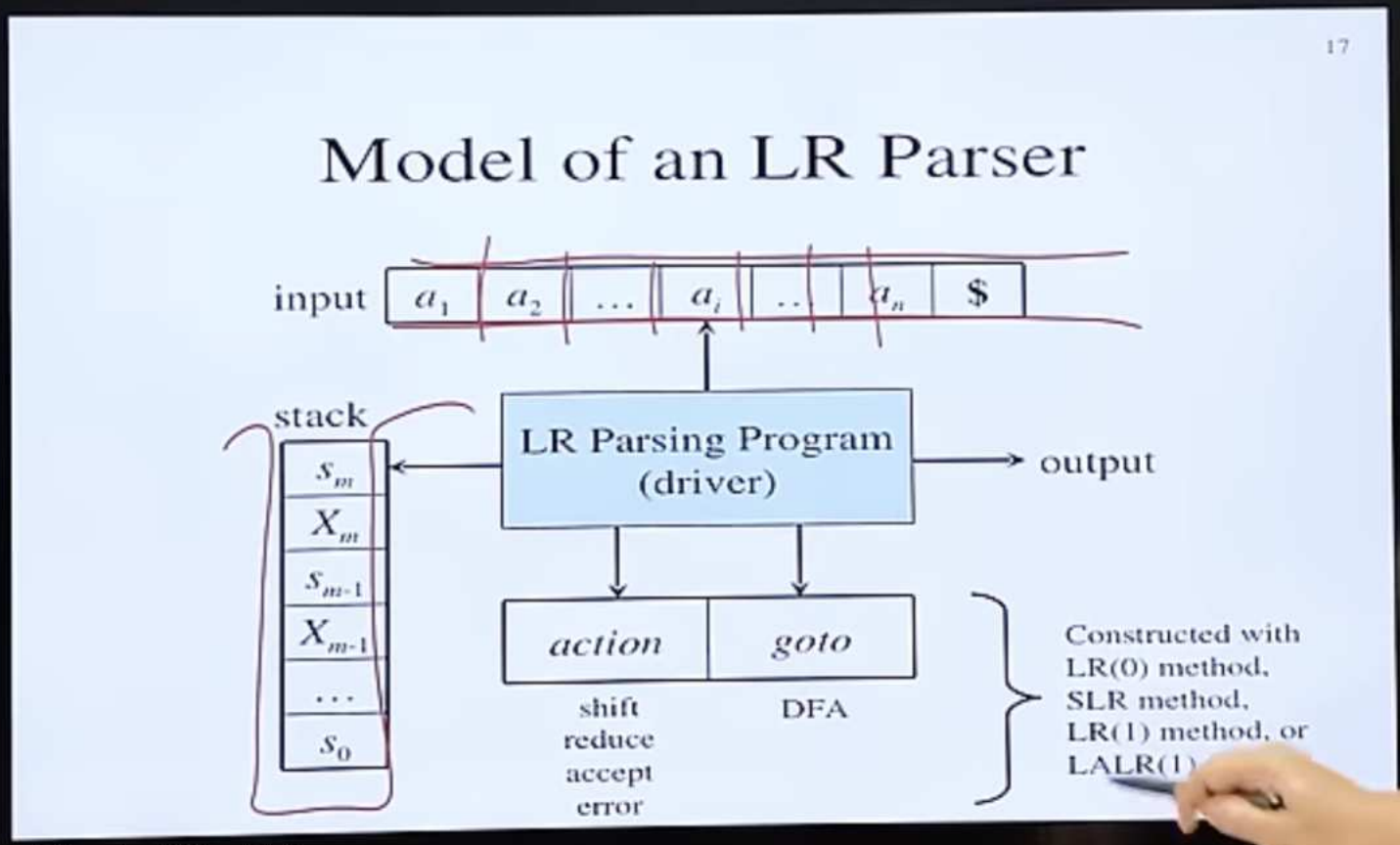
- **Handle**: - Substring of the i/p string that matches with RHS of any production, is called as Handle.
- The process of finding the handle & replacing that handle by it's LHS variable is called Handle Pruning.
- Bottom-Up-Parser is also known as Shit-Reduced Parser.
- BUP can be constructed for both Ambiguous & Unambiguous grammar
  - Ambiguous →OPP
  - Unambiguous → LR(k)

- LR(K) parser can be constructed for Unambiguous grammar.
- BUP Simulates the Reverse of Right Most Derivation.
- BUP can be constructed for the grammar which has more complexity.
- Bottom-up parsing is faster than Top-Up-Parsing, such that BUP is more efficien than TDP.



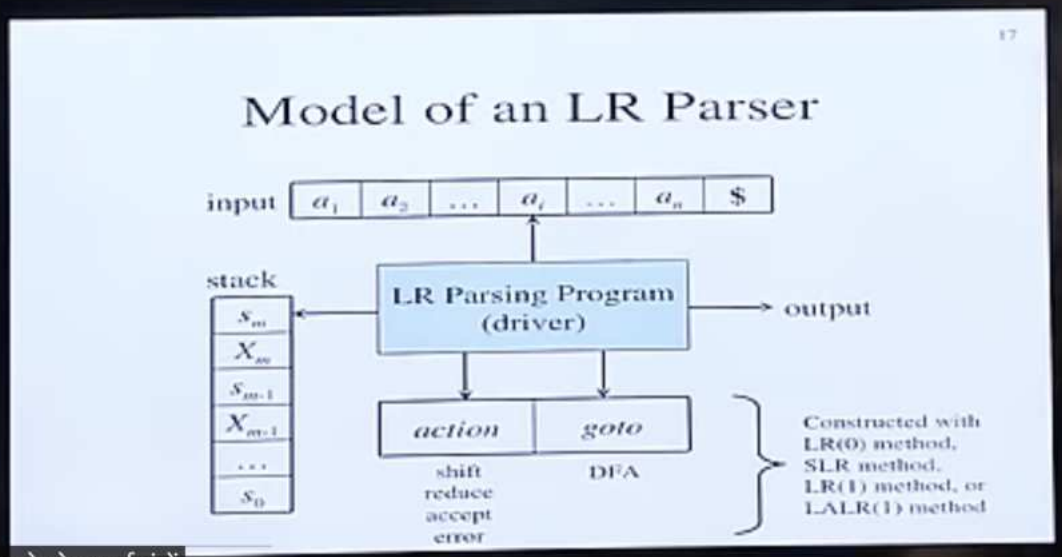
कंस्ट्रक्टेड फॉर ग्रामर व्हिच हैज मोर
कॉम्प्लेक्टेड टू

- Bottom Up Parser consist of three components
  - i/p buffer
  - Parse stack
  - Parse table



पार्सिंग करके नहीं दिखाई कैसे काम आएगा
आपको समझेगा देन व्हाट डू

- **Parse table**: parse table is constructed using terminals, non-terminals & LR(0) items. this parse table consist of two parts:
  - Action
  - Goto
- Action part contains shift & reduced operation over the terminals
- Goto part consists of only Shift operation over the Non-terminals.



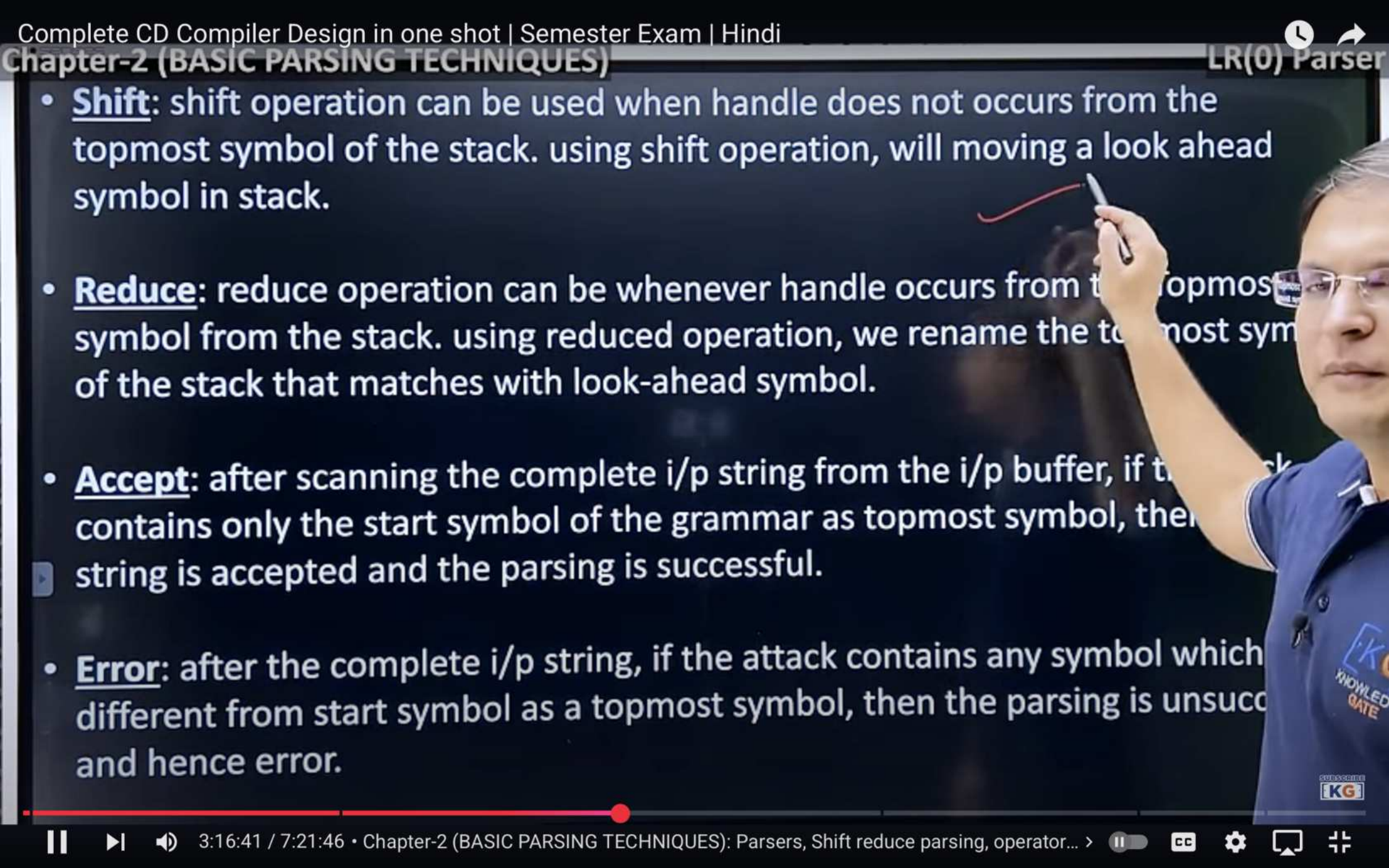
थे नॉन टर्मिनल सो गो टू पार्ट कंटें
शिफ्ट एंट्रीज फॉर

- **Shift**: shift operation can be used when handle does not occurs from the topmost symbol of the stack. using shift operation, will moving a look ahead symbol in stack.

- **Reduce**: reduce operation can be whenever handle occurs from t opmos symbol from the stack. using reduced operation, we rename the to most sym of the stack that matches with look-ahead symbol.

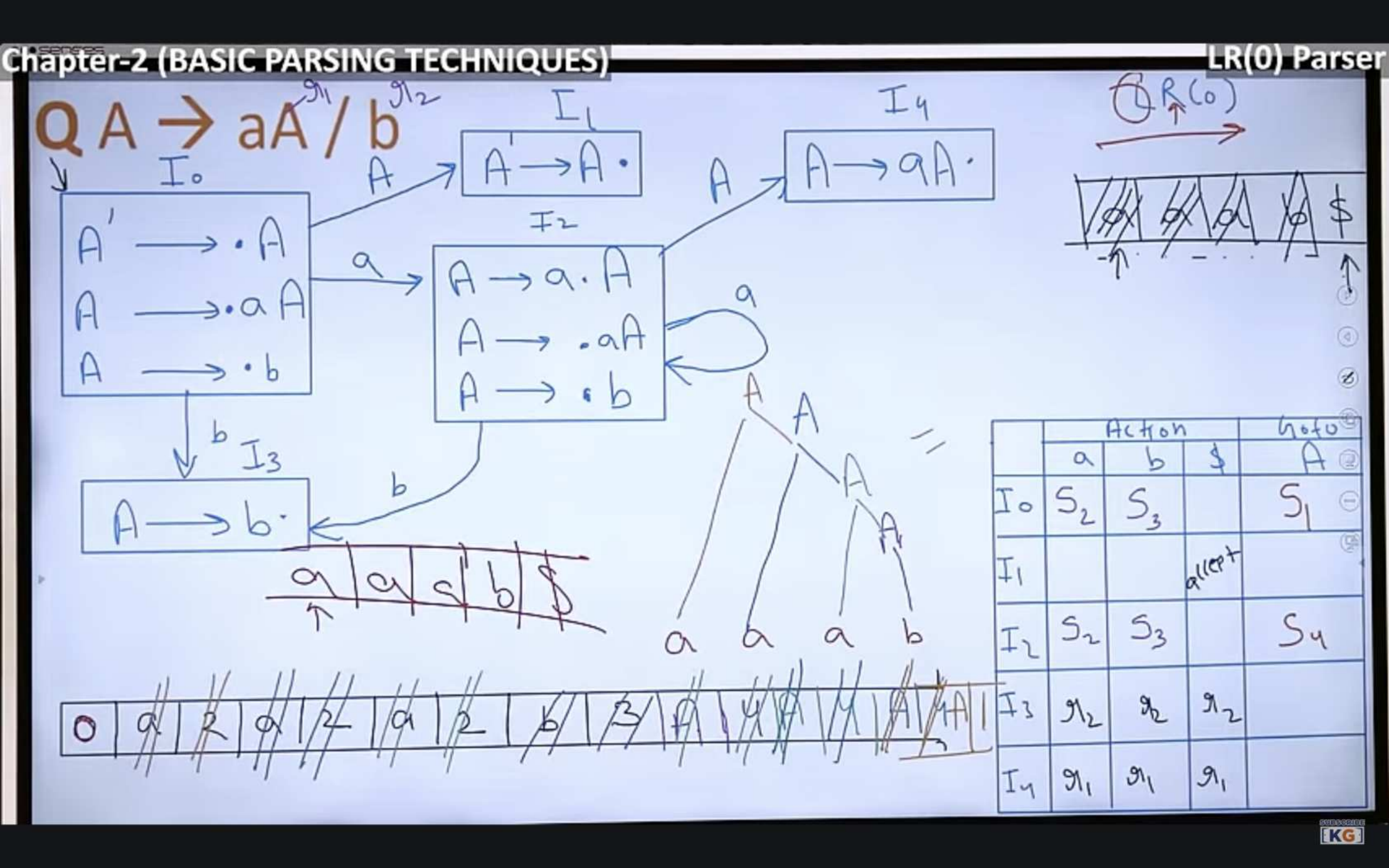
- **Accept**: after scanning the complete i/p string from the i/p buffer, if t k contains only the start symbol of the grammar as topmost symbol, ther string is accepted and the parsing is successful.

- **Error**: after the complete i/p string, if the attack contains any symbol which different from start symbol as a topmost symbol, then the parsing is unsucc and hence error.



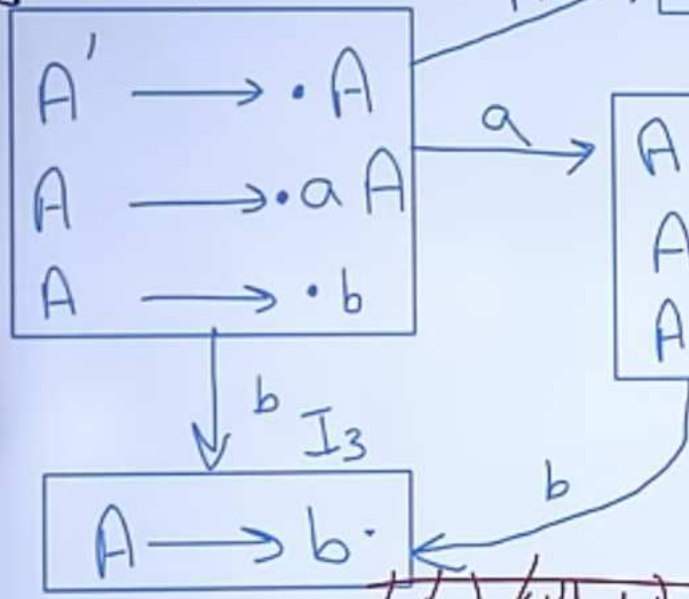
- Operation in shift/reduced passer
  - shift
  - reduced
  - accept
  - error

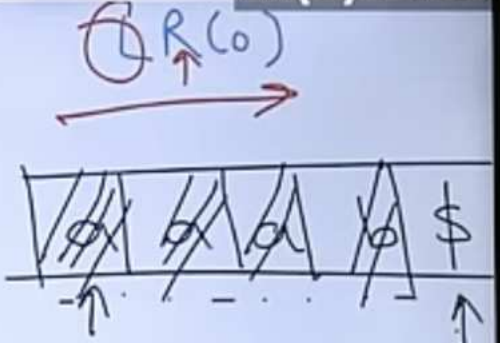
	Action	Goto
I_0	Terminals	Non-Terminals
I_{n-1}	Shift / Reduce	Shift

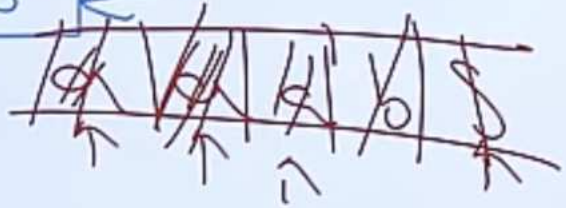


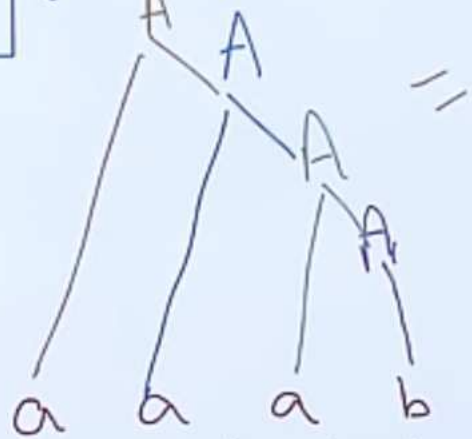
	Action			Goto
	a	b	$	A
I_0	S_2	S_3		S_1
I_1			accept	
I_2	S_2	S_3		S_4
I_3	r_2	r_2	r_2	
I_4	r_1	r_1	r_1	

Q A → aA / b (r1 / r2)

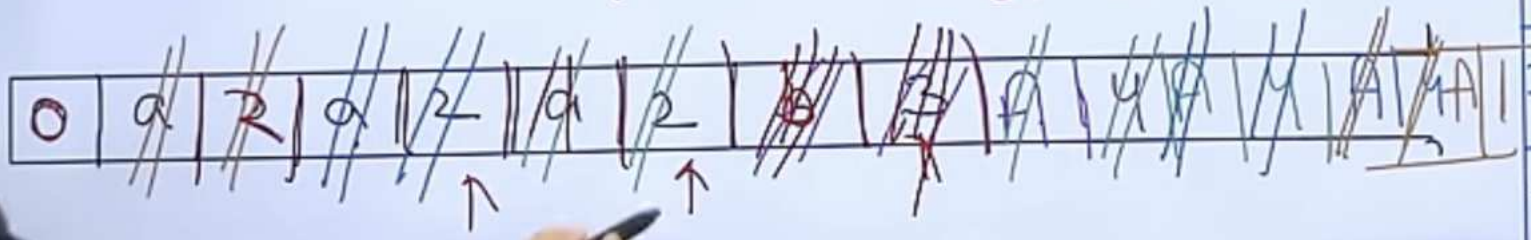


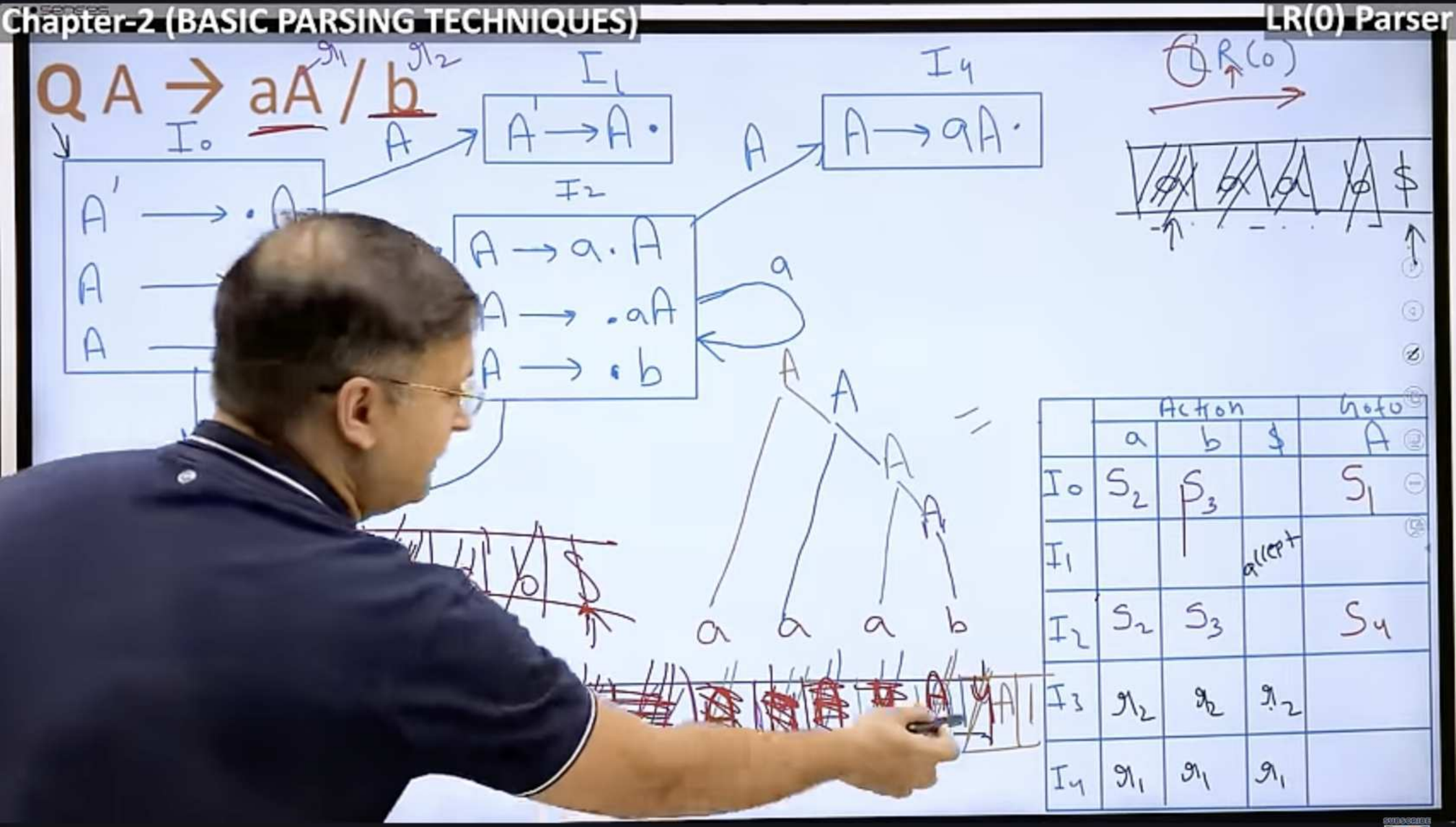






	Action			Goto
	a	b	$	A
I_0	S_2	S_3		S_1
I_1			accept	
I_2	S_2	S_3		S_4
I_3	r_2	r_2	r_2	
I_4	r_1	r_1	r_1	



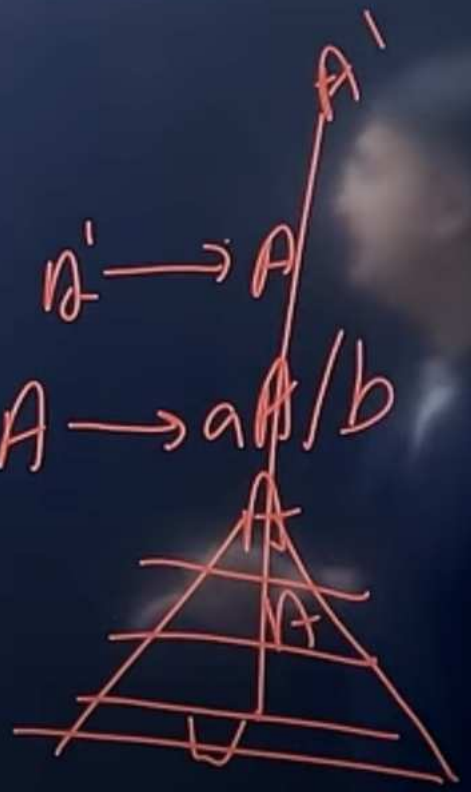

Q A → aA / b
r1 r2
I0
A' → .A
A → .aA
A → .b
A
I1
A' → A.
A
I2
A → a.A
A → .aA
A → .b
a
A
I4
A → aA.
LR(0)
$
a a a b
A A A A
Action
Goto
a
b
$
A
I0 S2 S3 S1
I1 accept
I2 S2 S3 S4
I3 r2 r2 r2
I4 r1 r1 r1

- Procedure for the construction of LR Parser table:
  1. Obtain the augmented grammar for the given grammar
  2. Create the canonical collection of LR items or compiler items.
  3. Draw the DFA & prepare the table based on LR items.

- **Augmented grammar**
  - The grammar which is obtained by addition one more production that generate the start symbol of the grammar, is known as Augmented grammar.
- $S \rightarrow AB \quad A \rightarrow a \quad B \rightarrow b$

$S' \rightarrow S \quad S \rightarrow AB \quad A \rightarrow a \quad B \rightarrow b$

$S' \rightarrow S$



- **LR(0) or Compiler item**
  - The production, which has dot(.) anywhere on RHS is known as LR(0) items.

  - A → abc

  - LR(0) items:
    - A → .abc
    - A → a.bc
    - A → ab.c
    - A → abc. Final / Completed items

- **Canonical Collection:**
  - The set $C = \{I_0, I_1, I_2, I_3, \ldots\ldots\ldots I_N\}$ is known as canonical collection of LR(0) items.

- Function used to generate LR(0) item's:
  - Closure: i/p set of items & o/p also set of items
  - GOTO

- Add everything from i/p to o/p
- If A → α. βB is in closure(I) & β → б is in the grammar G
  - Then add β → .б to the closure(I)
  - A → α. βB
  - β → .б
- Repeat the previous step for every newly added item

- Goto(I, X)
  - Goto(I, X)
  - Goto(A → α.Xβ, X) = A → αX.β

$Goto(I_i, X) = I_j$ (X is terminal)

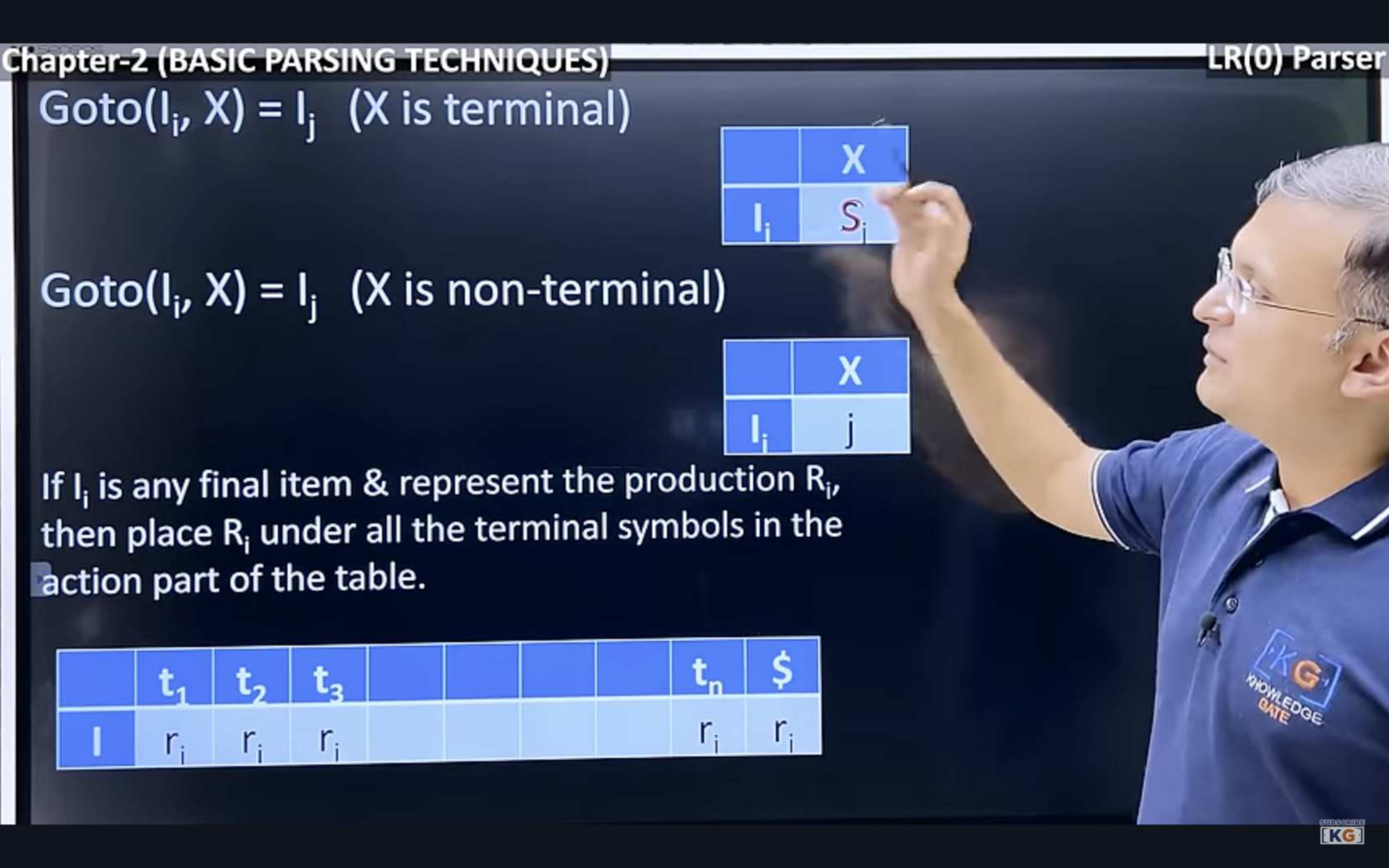
	X
I_i	S_j

$Goto(I_i, X) = I_j$ (X is non-terminal)

	X
I_i	j

If I_i is any final item & represent the production R_i, then place R_i under all the terminal symbols in the action part of the table.

	t_1	t_2	t_3					t_n	$
I	r_i	r_i	r_i					r_i	r_i



$Goto(I_i, X) = I_j$ (X is terminal)

	X
I_i	S_j

$Goto(I_i, X) = I_j$ (X is non-terminal)

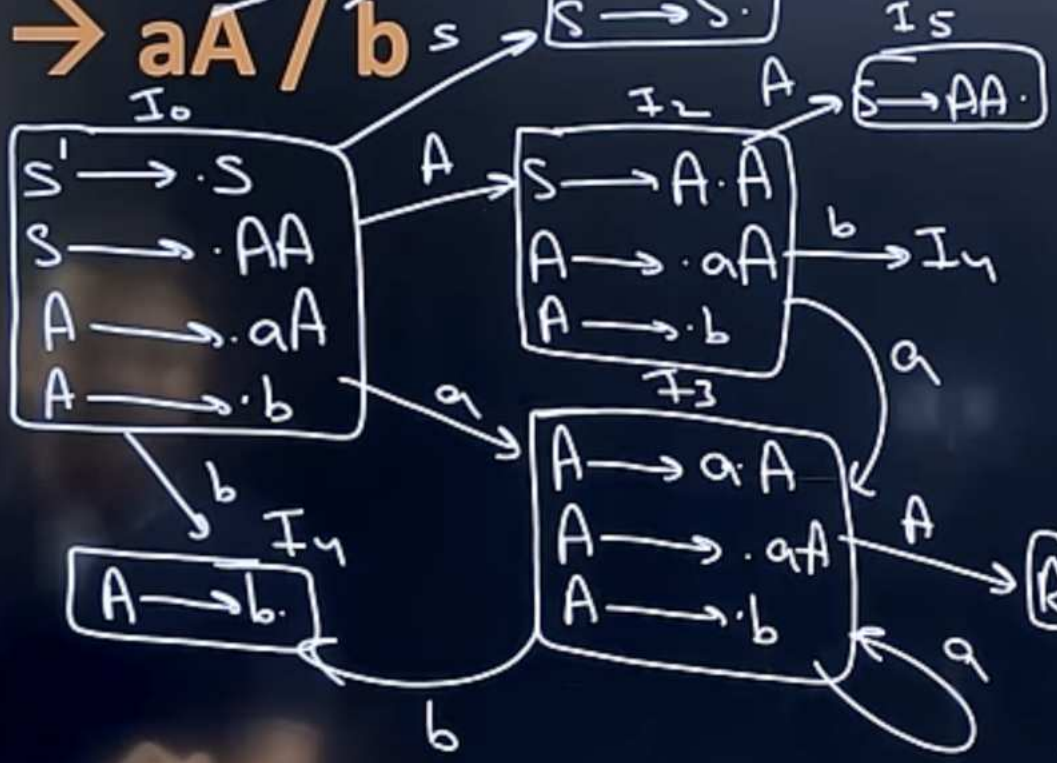
	X
I_i	j

If I_i is any final item & represent the production R_i, then place R_i under all the terminal symbols in the action part of the table.

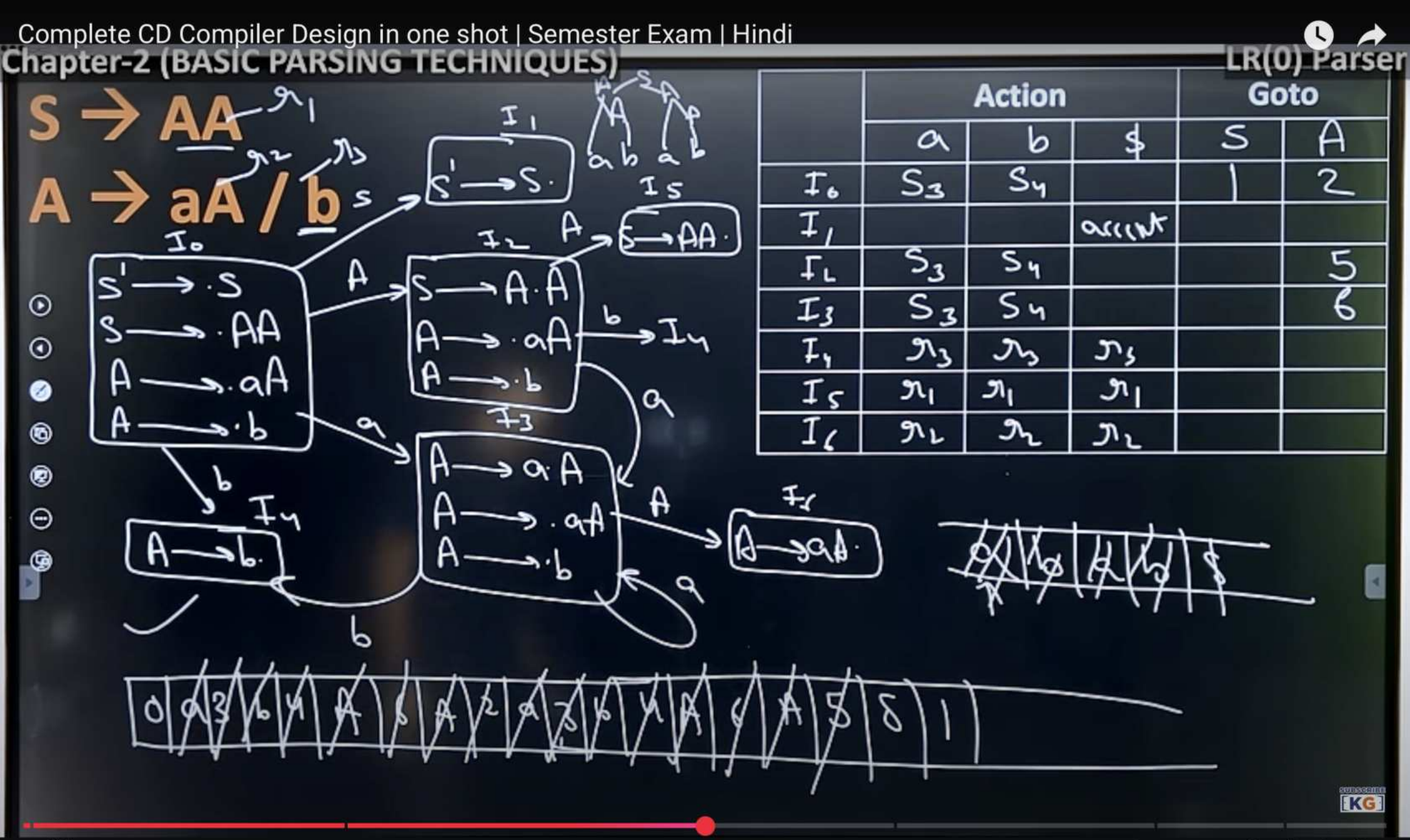
	t_1	t_2	t_3				t_n	$	
I	r_i	r_i	r_i				r_i	r_i	

$S \rightarrow AA$ r_1

$A \rightarrow aA$ r_2 / b r_3



	Action			Goto	
	a	b	$	S	A
I_0	S_3	S_4		1	2
I_1			accept		
I_2	S_3	S_4			5
I_3	S_3	S_4			6
I_4	r_3	r_3	r_3		
I_5	r_1	r_1	r_1		
I_6	r_2	r_2	r_2		


S → AA
A → aA / b
r1
r2
r3
I0
S' → .S
S → .AA
A → .aA
A → .b
I1
S' → S.
I2
S → A.A
A → .aA
A → .b
I3
A → a.A
A → .aA
A → .b
I4
A → b.
I5
S → AA.
I6
A → aA.
S
A
a
b
$
Action
Goto
I0 | S3 | S4 | | 1 | 2
I1 | | | accept | |
I2 | S3 | S4 | | | 5
I3 | S3 | S4 | | | 6
I4 | r3 | r3 | r3 | |
I5 | r1 | r1 | r1 | |
I6 | r2 | r2 | r2 | |

SLR(1)

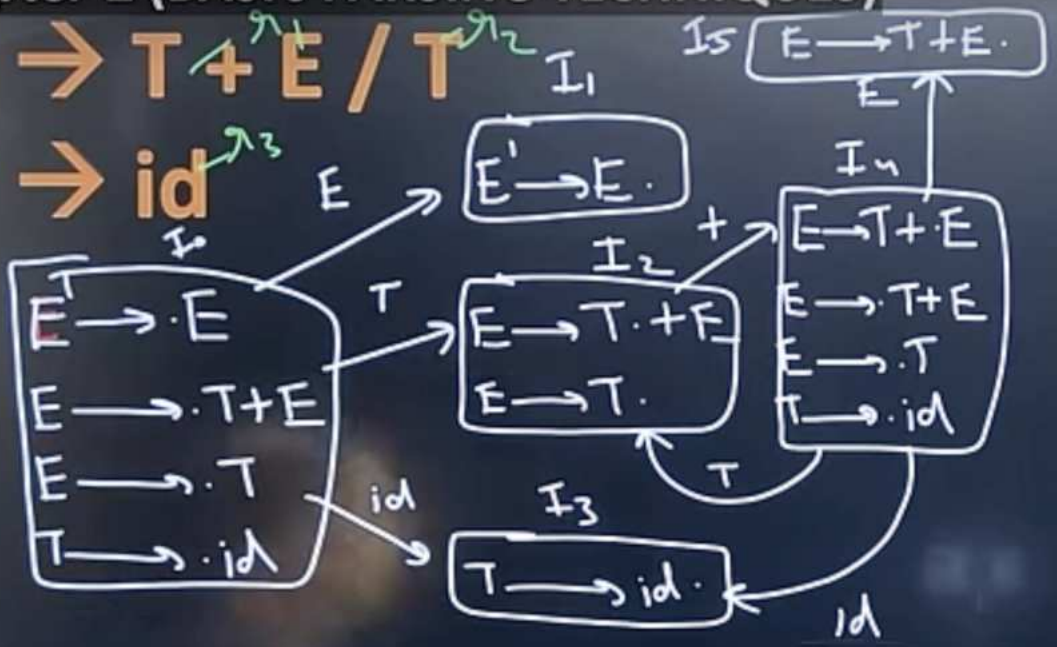
- The procedure for constructing the parse table is similar to LR(0) parse, but there is a restriction in the reducing entries.
- Whenever there is a final item, then placed the reduced entries under the follow symbol of LHS symbol.
- If the SLR(1) parse table is free from multiple entries than the grammar is SLR(1) grammar.
- Every LR(0) grammar is SLR(1), but every SLR(1) grammar need not be LR(0)
- SLR(1) parser is more powerful then LR(0) parser

Chapter-2 (BASIC PARSING TECHNIQUES)

SLR(1) Parser

$E \rightarrow T+E \;(r_1) \;/\; T \;(r_2)$

$T \rightarrow id \;(r_3)$



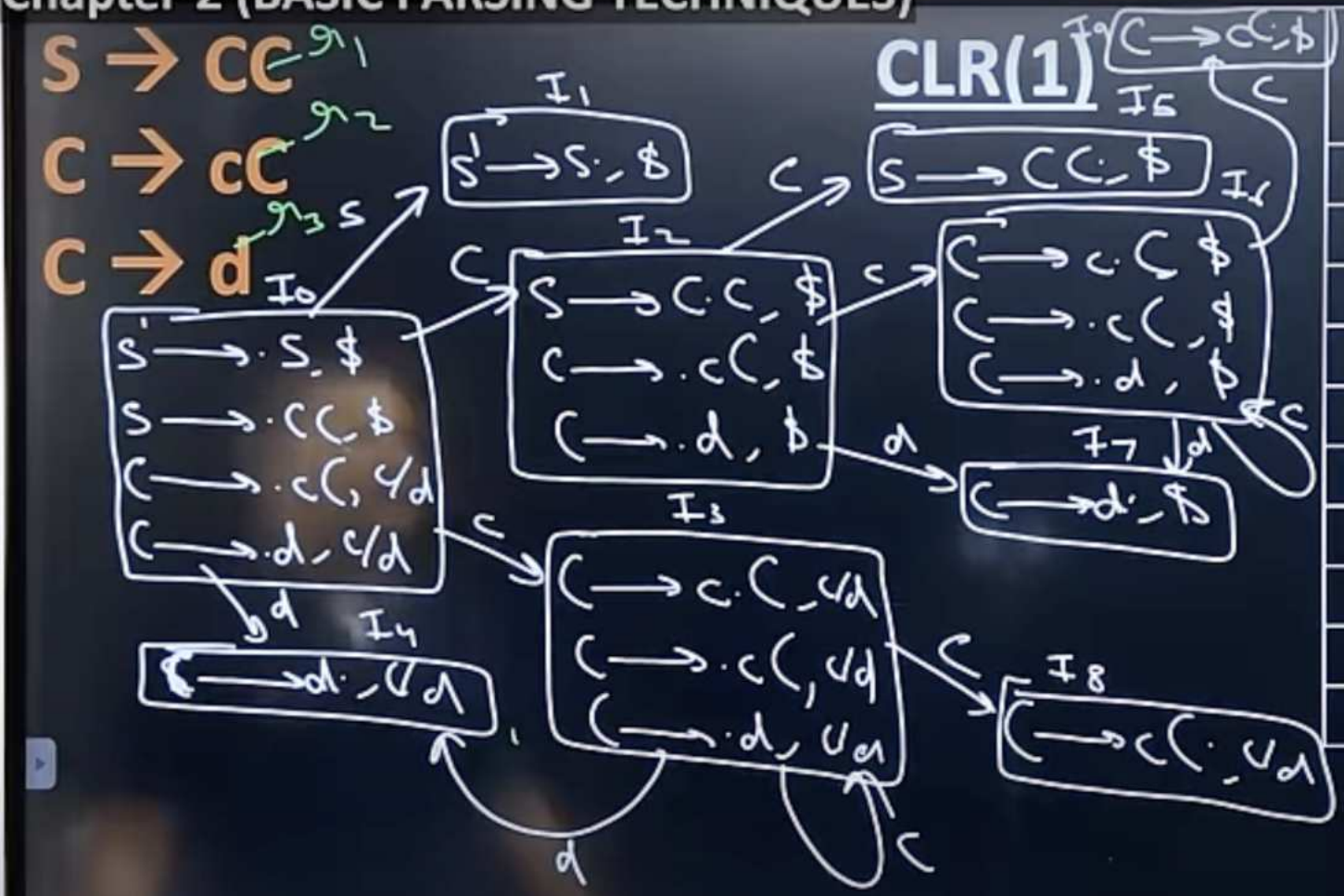
	Action			Goto	
	+	id	$	E	T
I_0		S_3		1	2
I_1			acc		
I_2	S_4		r_2		
I_3	r_3		r_3		
I_4		S_3		5	
I_5			r_1		

$Fol(E) = \{\$\}$

$Fol(T) = \{\$, +\}$

Chapter-2 (BASIC PARSING TECHNIQUES)

CLR(1) Parser



	Action			Goto	
	c	d	$	S	C
I_0	S_3	S_4		1	2
I_1			accept		
I_2	S_6	S_7			5
I_3	S_3	S_4			8
I_4	r_3	r_3			
I_5			r_1		
I_6	S_6	S_7			9
I_7			r_3		
I_8	r_2	r_2			
I_9			r_2		

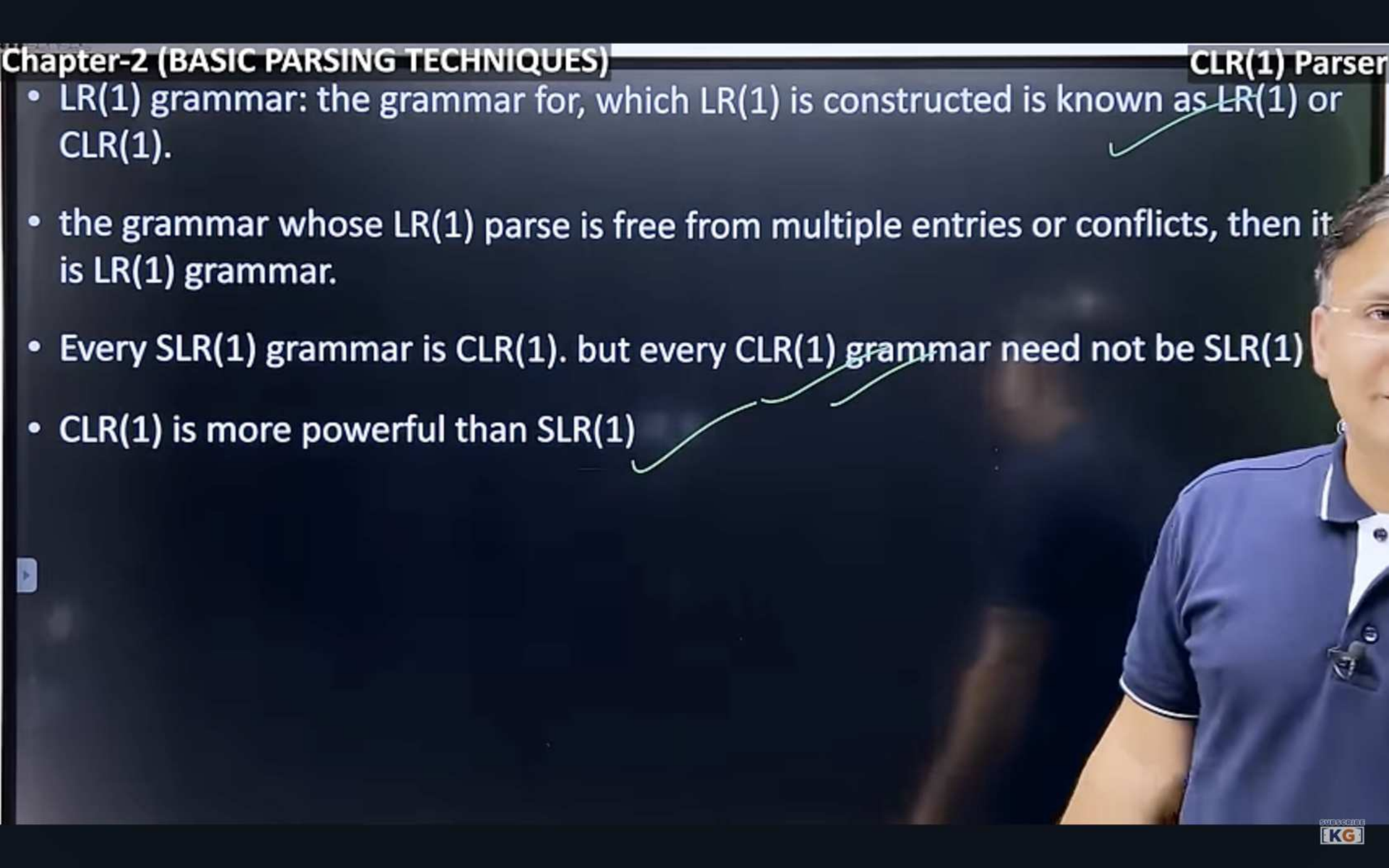
CLR(1)

- LR(1) depends on one look Ahead symbol

- Closure(I):
  - Add everything from i/p to o/p
  - A → α.Bβ, $ is in closure I and β → б is in the grammar G, then add β → .б, first(β, $), to the closure I
  - repeat previous step for every newly added items

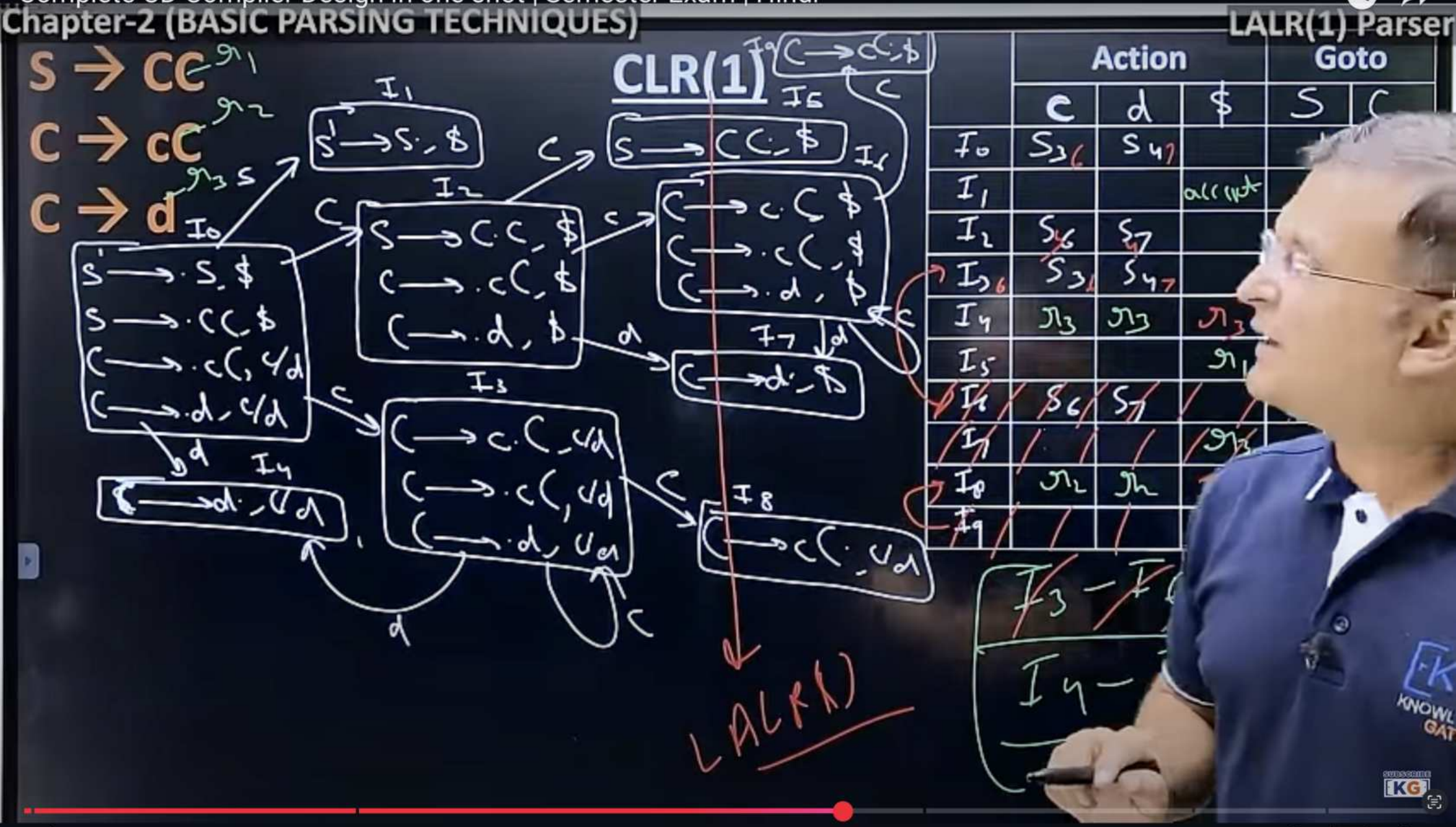
- Goto(I, x):
  - there will not be any change in the goto part while finding the transition.
  - these many be change in the follow or look Ahead part while finding the closure.

- LR(1) grammar: the grammar for, which LR(1) is constructed is known as LR(1) or CLR(1).
- the grammar whose LR(1) parse is free from multiple entries or conflicts, then it is LR(1) grammar.
- Every SLR(1) grammar is CLR(1). but every CLR(1) grammar need not be SLR(1)
- CLR(1) is more powerful than SLR(1)

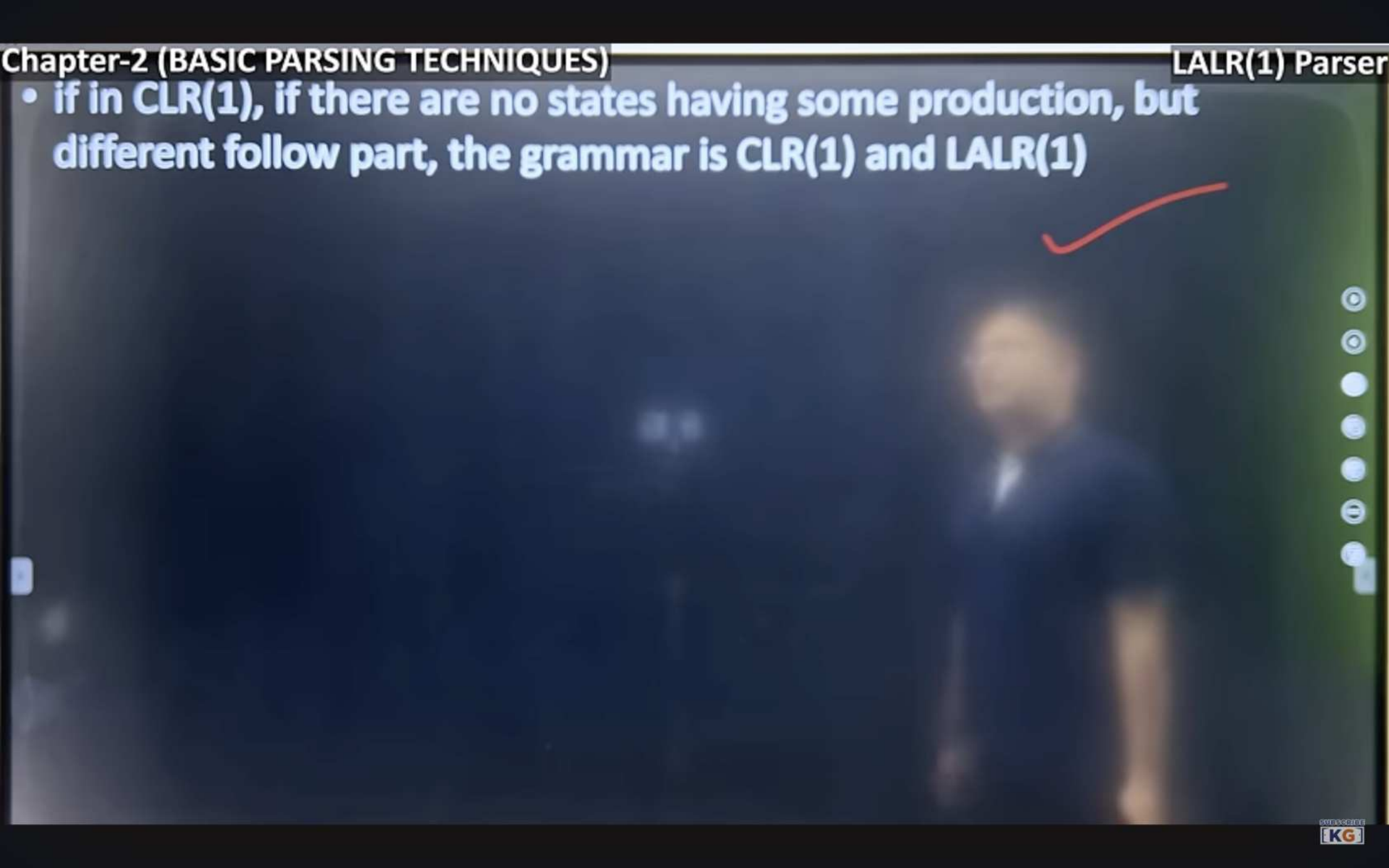


LALR(1)

- In CLR(1) parser, there can ne more than one state, having same production part and different follow part. Now combine those state whose production part is common and follow part is different, it is a single state and then construct the parse table, if the parse table is free from multiple entries, then the grammar is LALR(1).

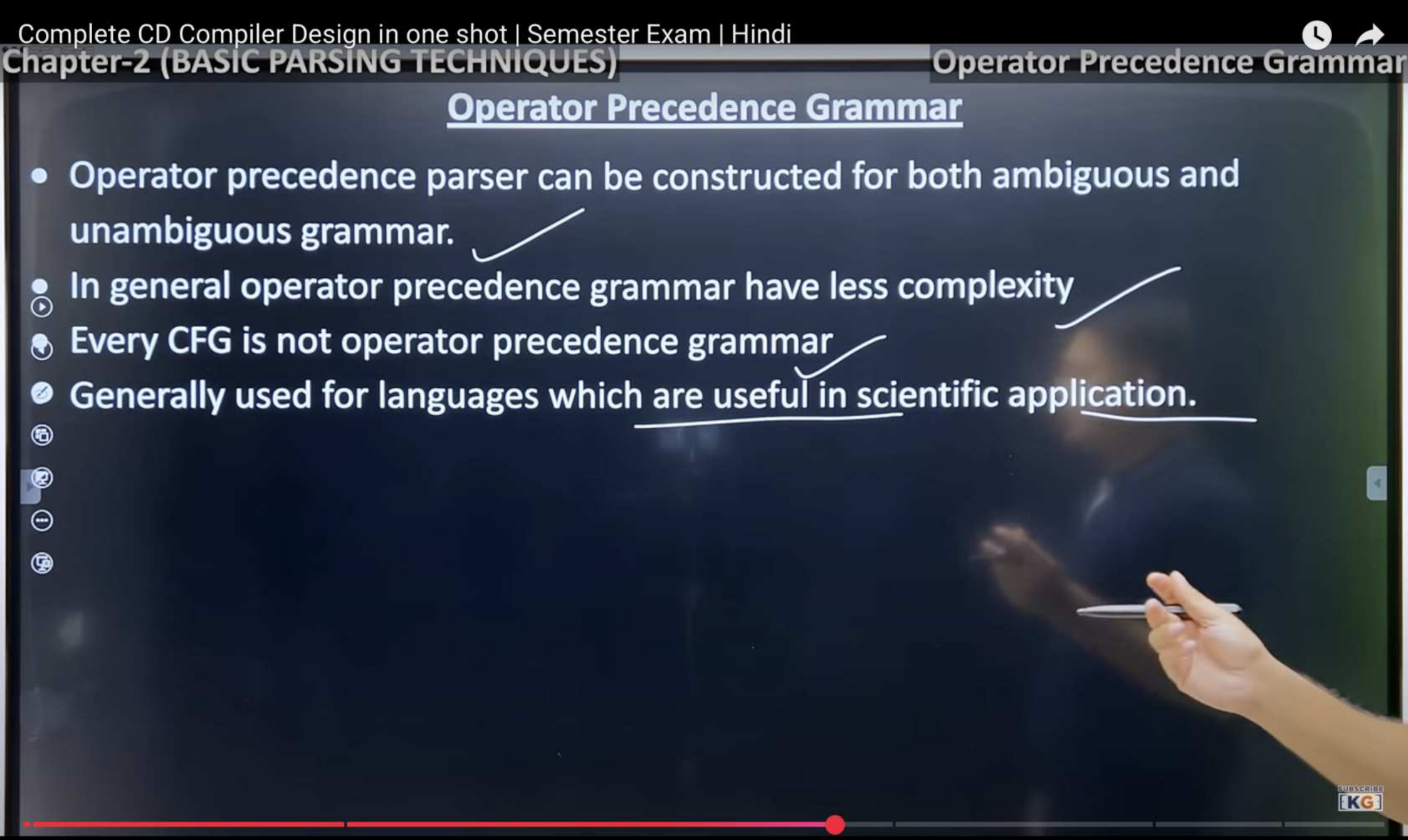

S → CC
C → cC
C → d
CLR(1)
LALR(1)
Action
Goto
c
d
$
S
C
accept

- if in CLR(1), if there are no states having some production, but different follow part, the grammar is CLR(1) and LALR(1)



Operator Precedence Grammar

- Operator precedence parser can be constructed for both ambiguous and unambiguous grammar.
- In general operator precedence grammar have less complexity
- Every CFG is not operator precedence grammar
- Generally used for languages which are useful in scientific application.


Operator Precedence Grammar
Operator precedence parser can be constructed for both ambiguous and unambiguous grammar.
In general operator precedence grammar have less complexity
Every CFG is not operator precedence grammar
Generally used for languages which are useful in scientific application.

- Operator Grammar is a context free grammar that has following properties
  - Does not contain ε production
  - No adjacent non-terminals on RHS of any production.

$\alpha \rightarrow \beta$

ABC

S → AB
A → a
B → b

S → AaB
B → aA / b
A → b
B → a

S → AaB
A → a / ε
B → b

S → AOB / int
O → + / * / -
A → b
B → a

Algorithm for computing precedence function

$ a + b $

a b

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow id$

a + b * c

a * b * c

a ⋗ b

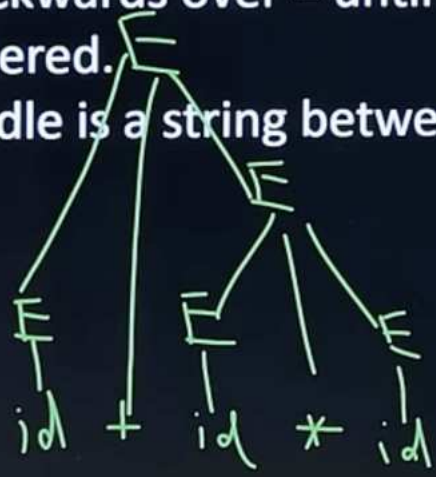
	id	*	+	$
id	—	⋗	⋗	⋗
*	⋖	⋗	⋗	⋗
+	⋖	⋖	⋗	⋗
$	⋖	⋖	⋖	—

(a+b) X c

1. $a \gtrdot b$
2. $a \lessdot b$
3. $a \doteq b$

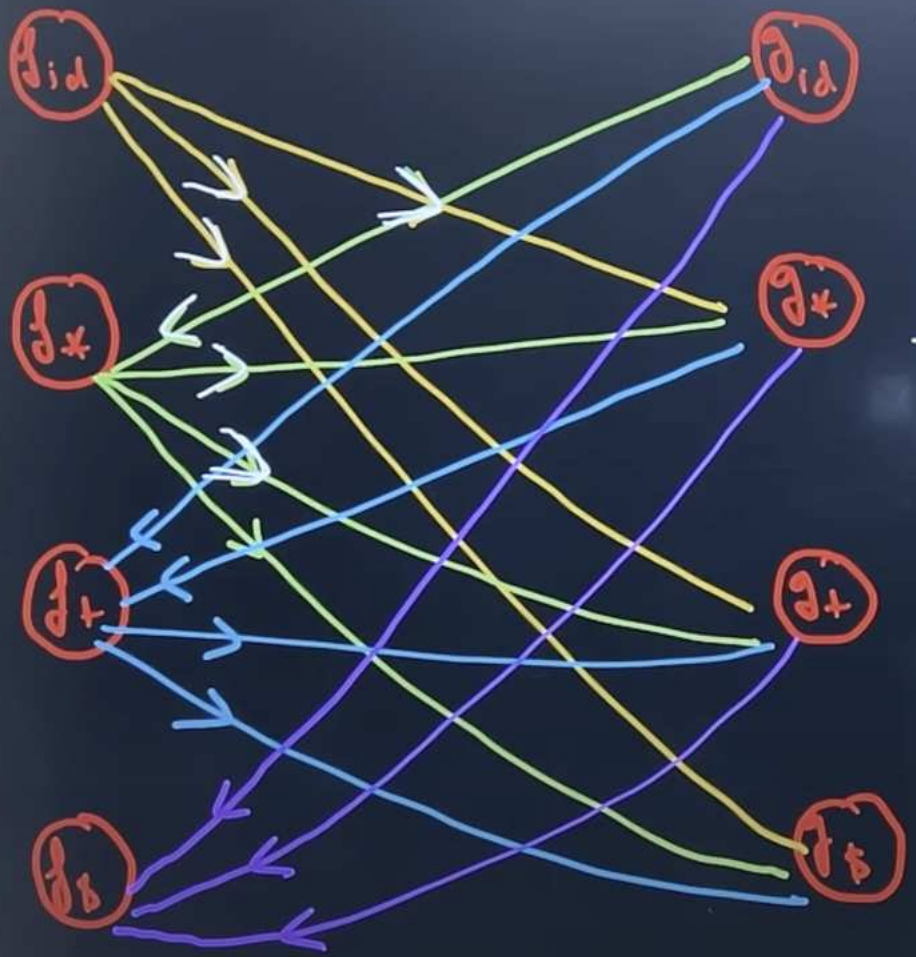
Operator Precedence Grammar

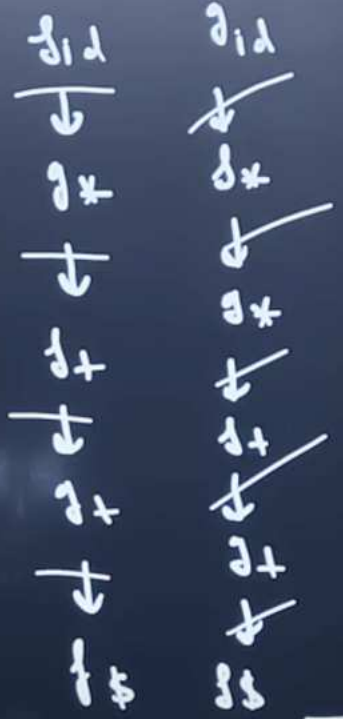
- Now consider the string *id + id * id*
- We will insert $ symbols at the start and end of the input string. We will also insert precedence operator by referring the precedence relation table.
- $ < · *id* · > + < · *id* · > * < *id* · > $
- We will follow following steps to parse the given string :
  - Scan the input from left to right until first · > is encountered.
  - Scan backwards over = until < · is encountered.
  - The handle is a string between < · and · >.



$ < · id · > + < · id · > * < · id · > $	Handle id is obtained between <··>. Reduce this by E → id.
E + < · id · > * < · id · > $	Handle id is obtained between <··>. Reduce this by E → id.
E + E * < · id · > $	Handle id is obtained between <··>. Reduce this by E → id.
E+E*E	Remove all the non-terminals.
+*	Insert $ at the beginning and at the end. Also insert the precedence operators.
$<·+<·*·>$	The * operator is surrounded by <··>. This indicates that * becomes handle. That means, we have to reduce E * E operation first.
$<·+·>$	Now + becomes handle. Hence, we evaluate E + E.
$$	Parsing is done.

Algorithm for computing precedence function





	id	*	+	$
id	−	>	>	>
*	<	>	>	>
+	<	<	>	>
$	<	<	<	−

	id	*	+	$
f	4	4		
g	5	3	1	0

Chapter-2 (BASIC PARSING TECHNIQUES) Operator Precedence Grammar

Aspect	Top-Down Parsing	Bottom-Up Parsing
Direction of Analysis	Begins from the start symbol and works towards the leaves of the syntax tree.	Starts from the leaves (input symbols) and works towards the root of the syntax tree.
Parse Tree Construction	Constructs the parse tree from the top (root) to the bottom (leaves).	Constructs the parse tree from the bottom (leaves) to the top (root).
Type of Grammar Used	Generally uses non-left-recursive grammars.	Can handle a more extensive range of grammars, including left-recursive grammars.
Example Methods	Recursive Descent Parsing, LL(k) Parsing (where k is the number of lookahead tokens).	LR(k) Parsing (including SLR, LALR, CLR), Operator-precedence parsing.
Handling Ambiguity	Less efficient in handling ambiguity and requires backtracking in some cases.	More efficient in handling ambiguous grammars and reduces the need for backtracking.