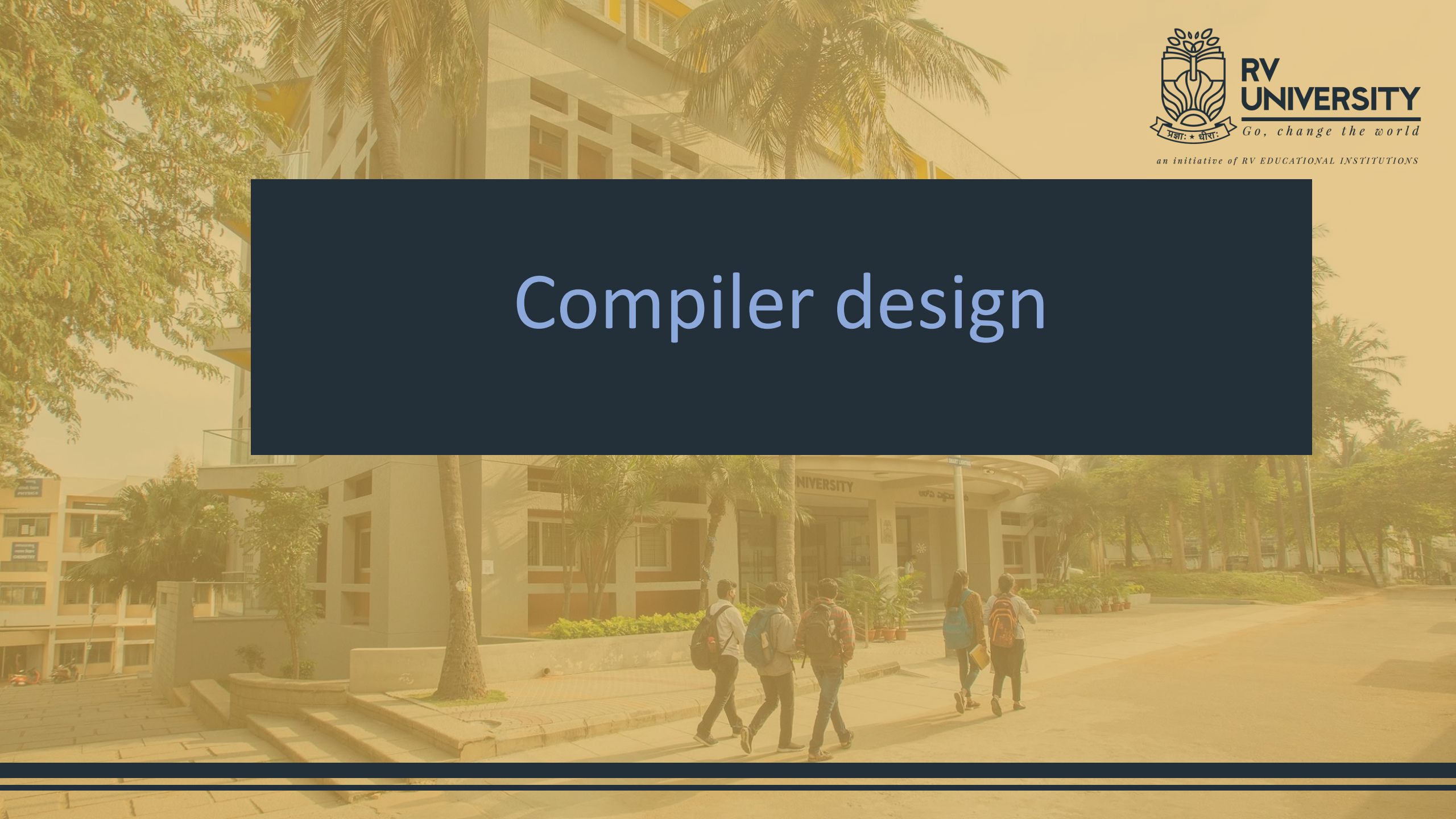
RV UNIVERSITY

Go, change the world

an initiative of RV EDUCATIONAL INSTITUTIONS

NEW-AGE GLOBAL UNIVERSITY FOR LIBERAL EDUCATION



Compiler design

Syntax Analysis

- The Role of the Syntax Analyzer

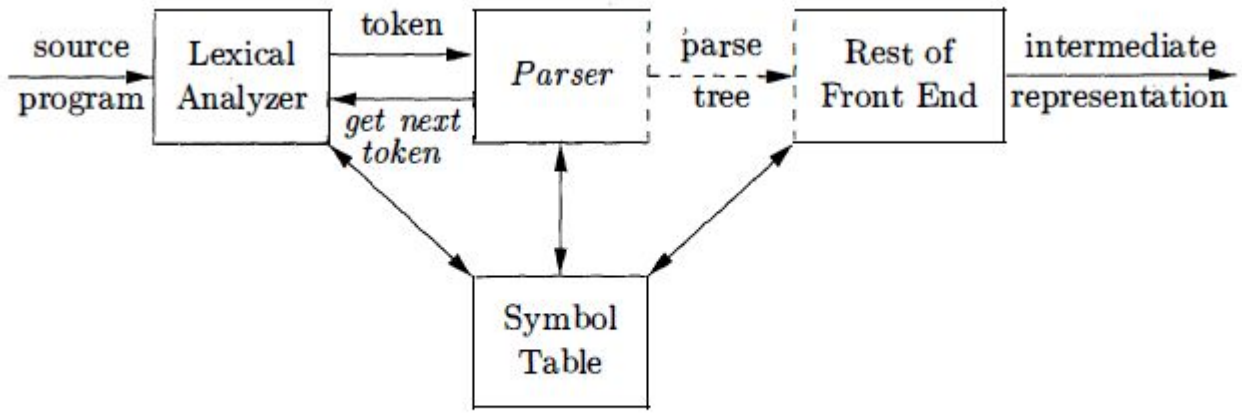


Figure 4.1: Position of parser in compiler model

Context free grammars

- Terminals
- Nonterminals
- Start symbol
- Productions

expression -> expression + term
expression -> expression – term
expression -> term
term -> term * factor
term -> term / factor
term -> factor
factor -> (expression)
factor -> **id**

Derivations

- Productions are treated as rewriting rules to generate a string
- Rightmost and leftmost derivations - E -> E + E
  | E * E
  | -E
  | (E)
  | id

Derivations for −(id+id) •

E => -E => -(E) => -(E+E) => -(id+E)=>-(id+id)

E => -E => -(E) => -(E+E) => -(E+id)=>-(id+id)

Parse Trees

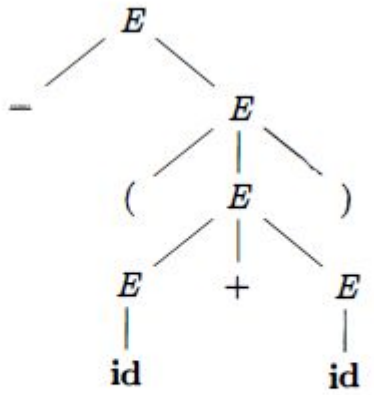


Figure 4.3: Parse tree for $-(\mathbf{id} + \mathbf{id})$

Ambiguity:

An ambiguous grammar is one that produces more than one leftmost derivation or more than one rightmost derivation for the same sentence.

Eg: id + id * id

$$
\begin{aligned}
E &\Rightarrow E + E \\
&\Rightarrow \mathbf{id} + E \\
&\Rightarrow \mathbf{id} + E * E \\
&\Rightarrow \mathbf{id} + \mathbf{id} * E \\
&\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}
\end{aligned}
\qquad
\begin{aligned}
E &\Rightarrow E * E \\
&\Rightarrow E + E * E \\
&\Rightarrow \mathbf{id} + E * E \\
&\Rightarrow \mathbf{id} + \mathbf{id} * E \\
&\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}
\end{aligned}
$$

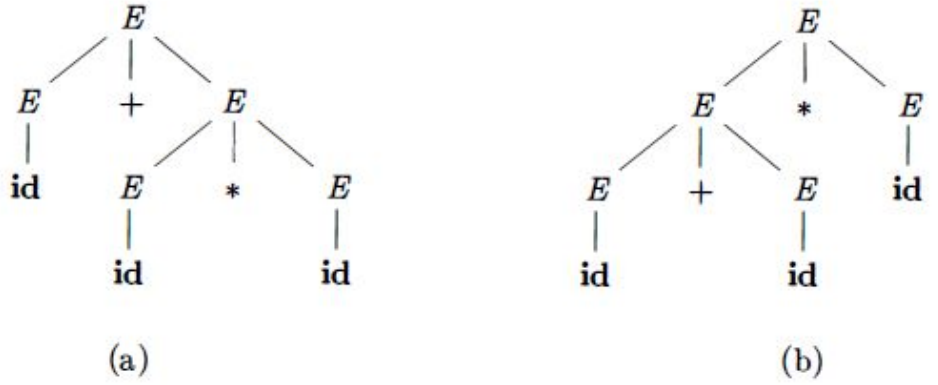
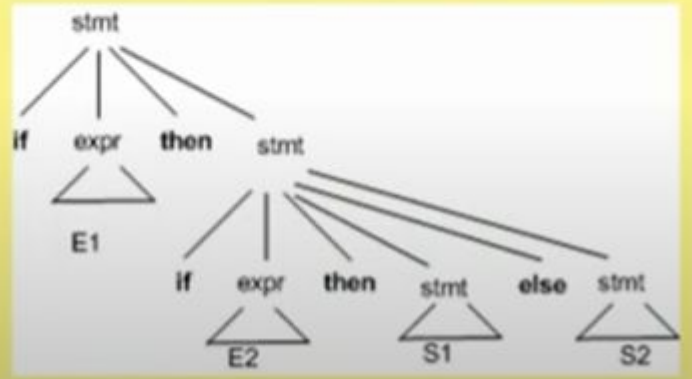
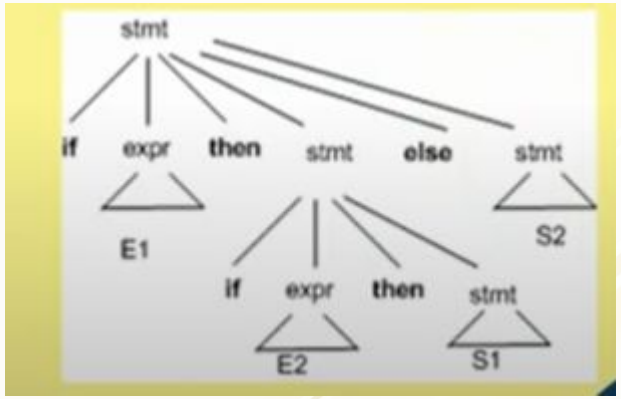


Figure 4.5: Two parse trees for **id+id*id**

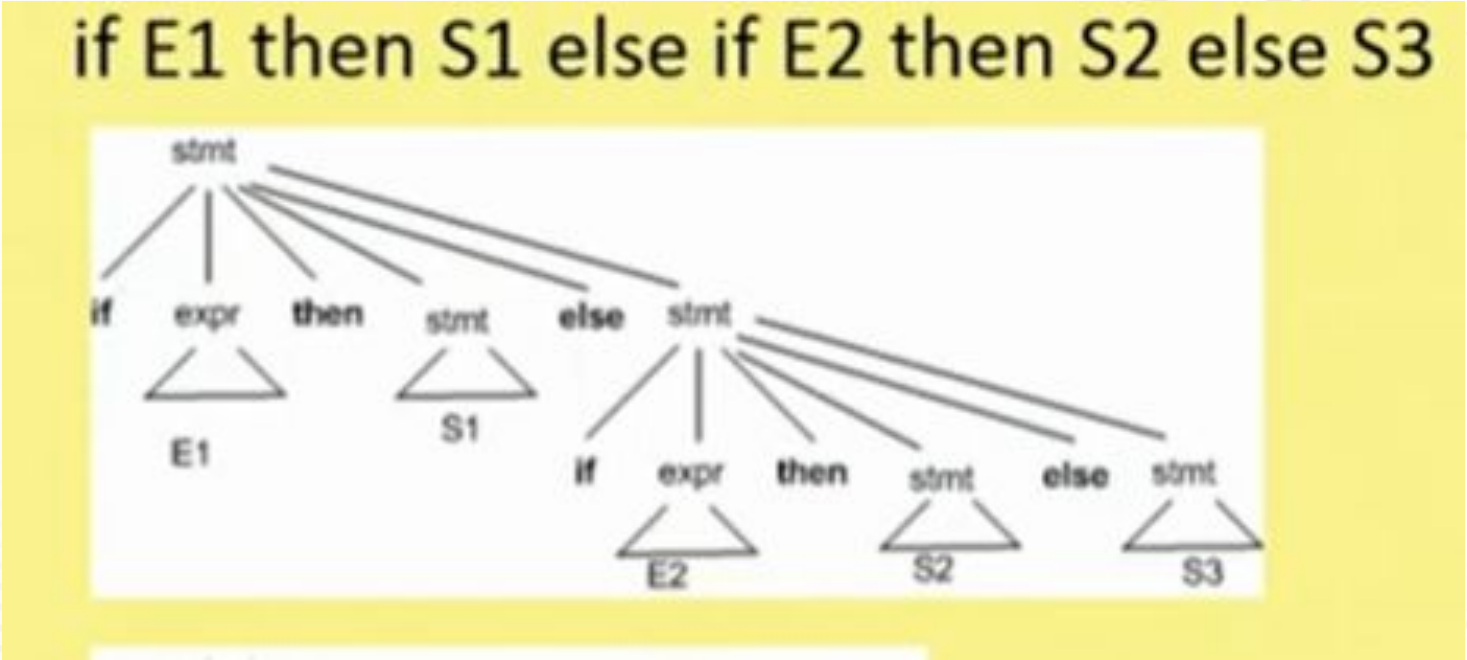
if E1 then if E2 then S1 else S2

stmt ⟶ **If** expr **then** stmt
| **If** expr **then** stmt **else** stmt
| **other**

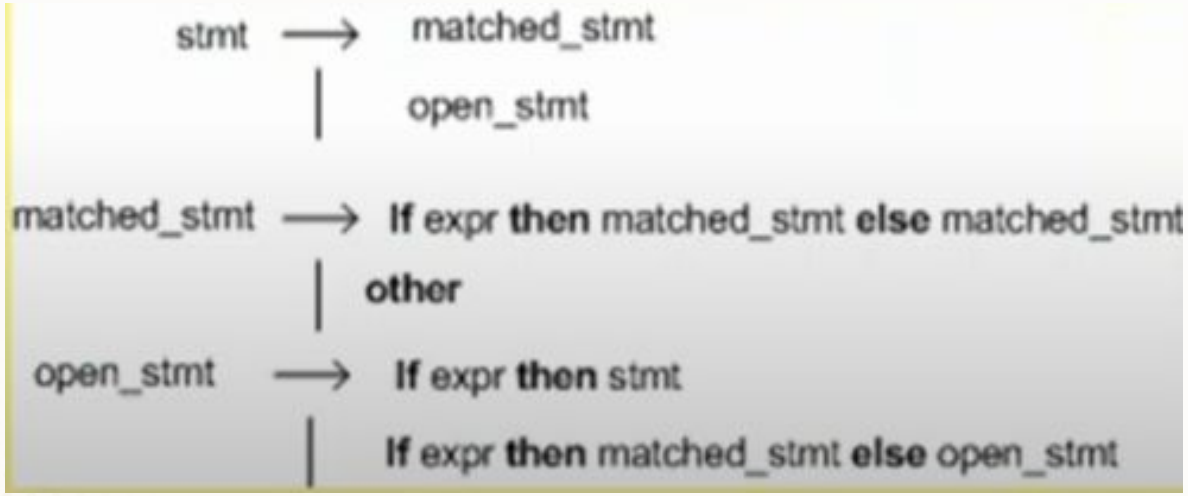




Elimination of Ambiguity



A statement appearing between **then** **and else** should be matched statement



Elimination of left recursion

A grammar is left recursive if it has a non-terminal A such that there is a derivation A=> A a

Top down parsing methods cant handle left-recursive grammars

A simple rule for direct left recursion elimination:

For a rule like: . A -> Aαβ

We may replace it with

A-> β A’

A´ -> α A’| ε

Left recursion elimination

There are cases like following

S -> Aa | b

A -> Ac |

Sd | ε

Left recursion elimination algorithm:

Arrange the nonterminals in some order A1,A2,..., An.

For (each i from 1 to n)

{ For (each j from 1 to i-1)

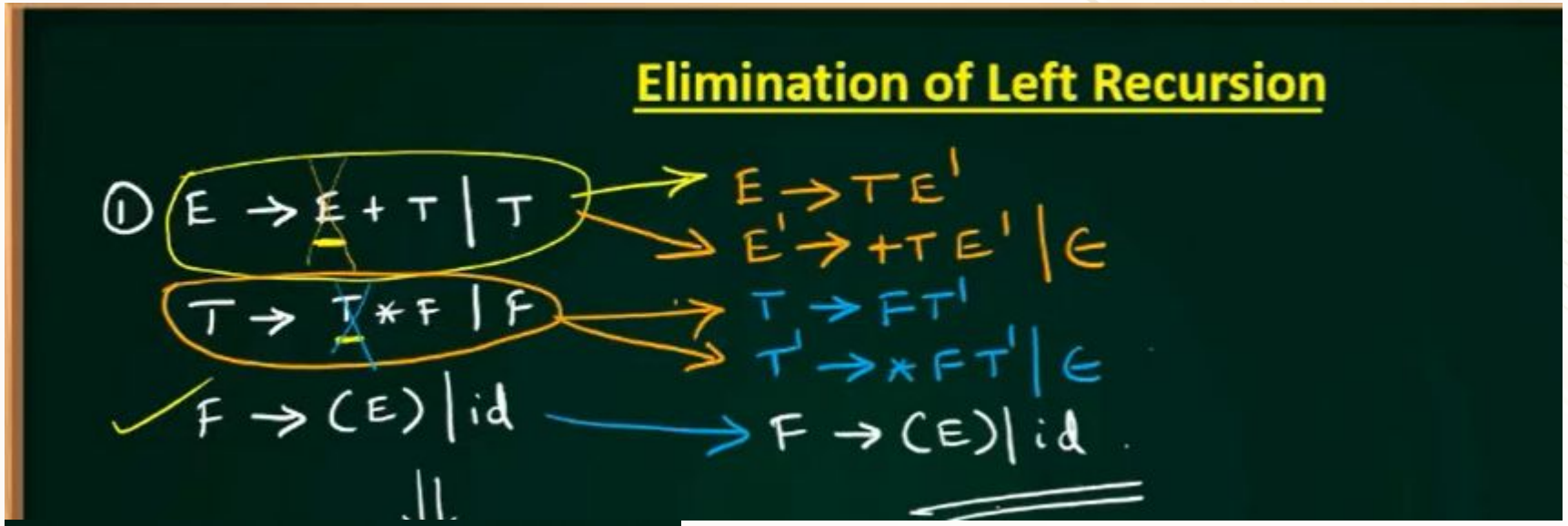
Replace each production of the form Ai-> Aj γ by the production Ai -> $\delta_1 \gamma$ | $\delta_2 \gamma$ | | $\delta_k \gamma$ where Aj-> δ_1 | δ_2 | ... | δ_k are all current Aj productions

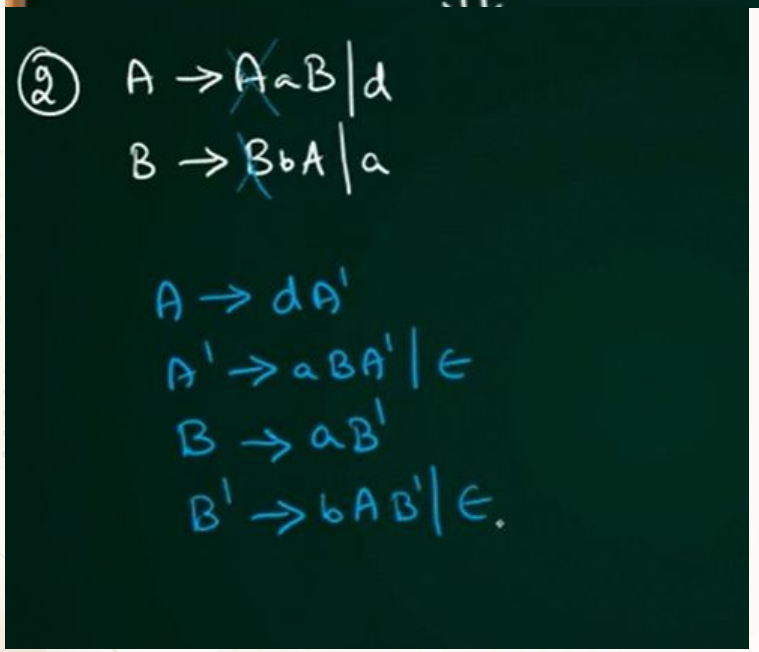
}

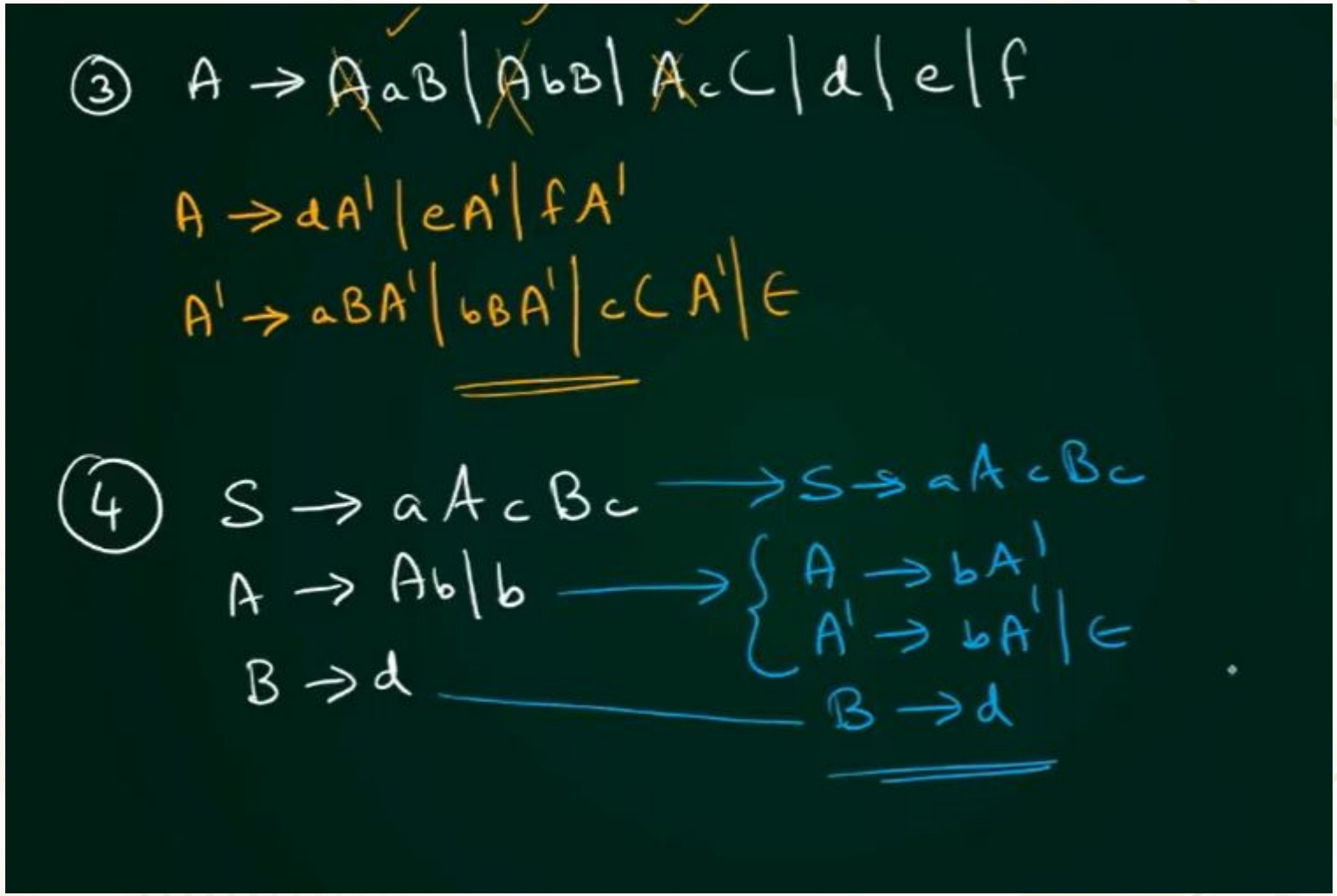
Eliminate left recursion among the Ai-productions

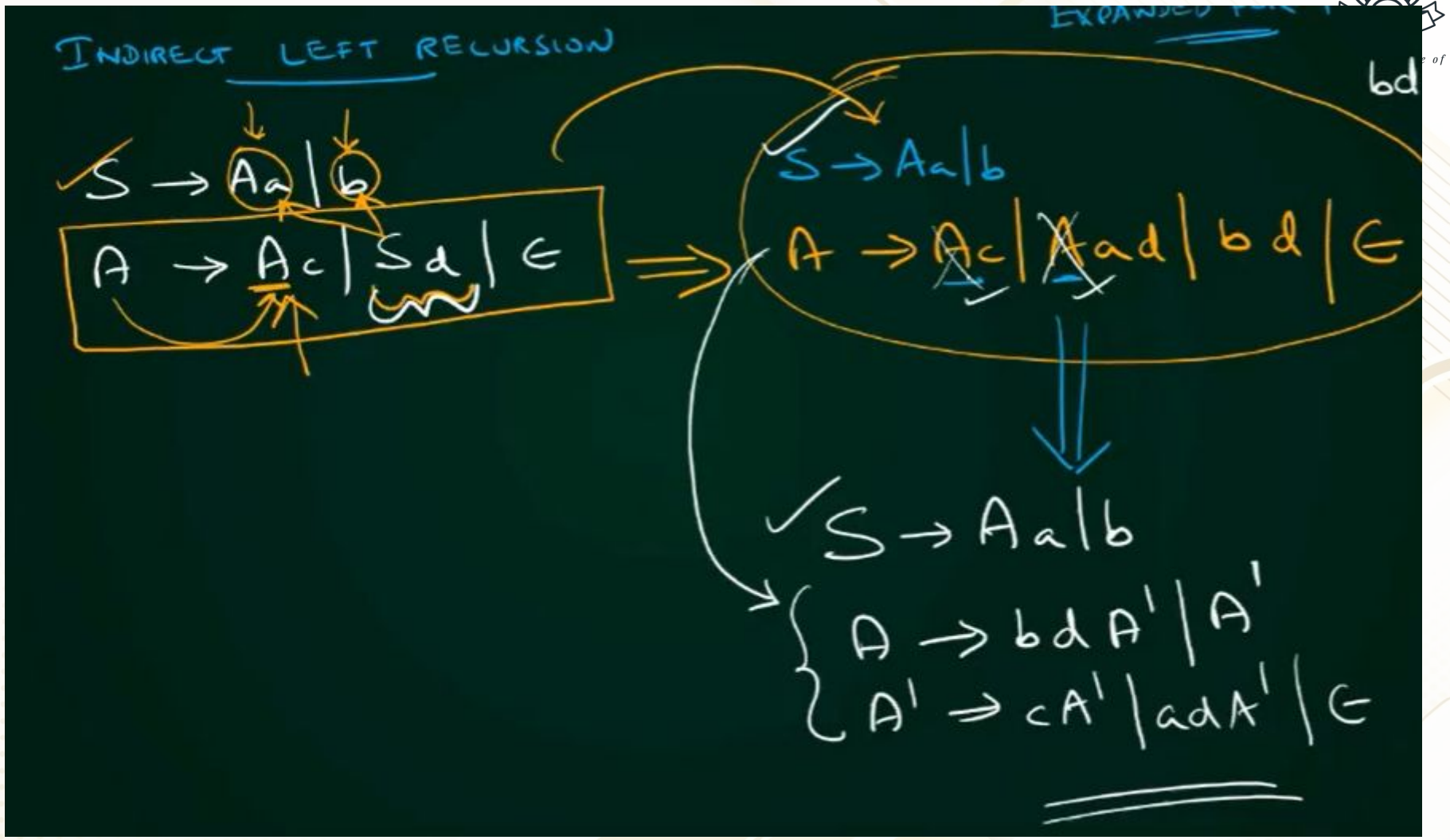
}

Elimination of Left Recursion








INDIRECT LEFT RECURSION
S → Aa | b
A → Ac | Sd | ∈
S → Aa | b
A → Ac | Aad | bd | ∈
bd
S → Aa | b
A → bdA' | A'
A' → cA' | adA' | ∈

Left factoring

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive or top-down parsing.

Consider following grammar:
Stmt -> if expr then stmt else stmt |
if expr then stmt
On seeing input if it is not clear for the parser which production to use .
We can easily perform left factoring:
If we have A-> α β1 | α β2 then we replace it with
A -> α A'
A´-> β1 | β2

Algorithm For each non-terminal A, find the longest prefix a common to two or more of its alternatives.

If a ≠ ε, then replace all of A-productions

A->αβ1| αβ2 | ... | αβí |γ

by •

A-> α A' |γ

A' -> β1| β2 ….βn.

Example: S->iEtS |iEtSeS I a

E-> b

Modifies to

S->iEtSS' | a

S'->eS | ε

E -> b

RV UNIVERSITY
Go, change the world
an initiative of RV EDUCATIONAL INSTITUTIONS

Top down parsing

A Top-down parser tries to create a parse tree from the root towards the leafs scanning input from left to right

It can be also viewed as finding a leftmost derivation for an input string

Example: id+id*id

$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E' \mid \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T' \mid \epsilon \\
F & \rightarrow & (\ E\ ) \mid \mathbf{id}
\end{array}
$$

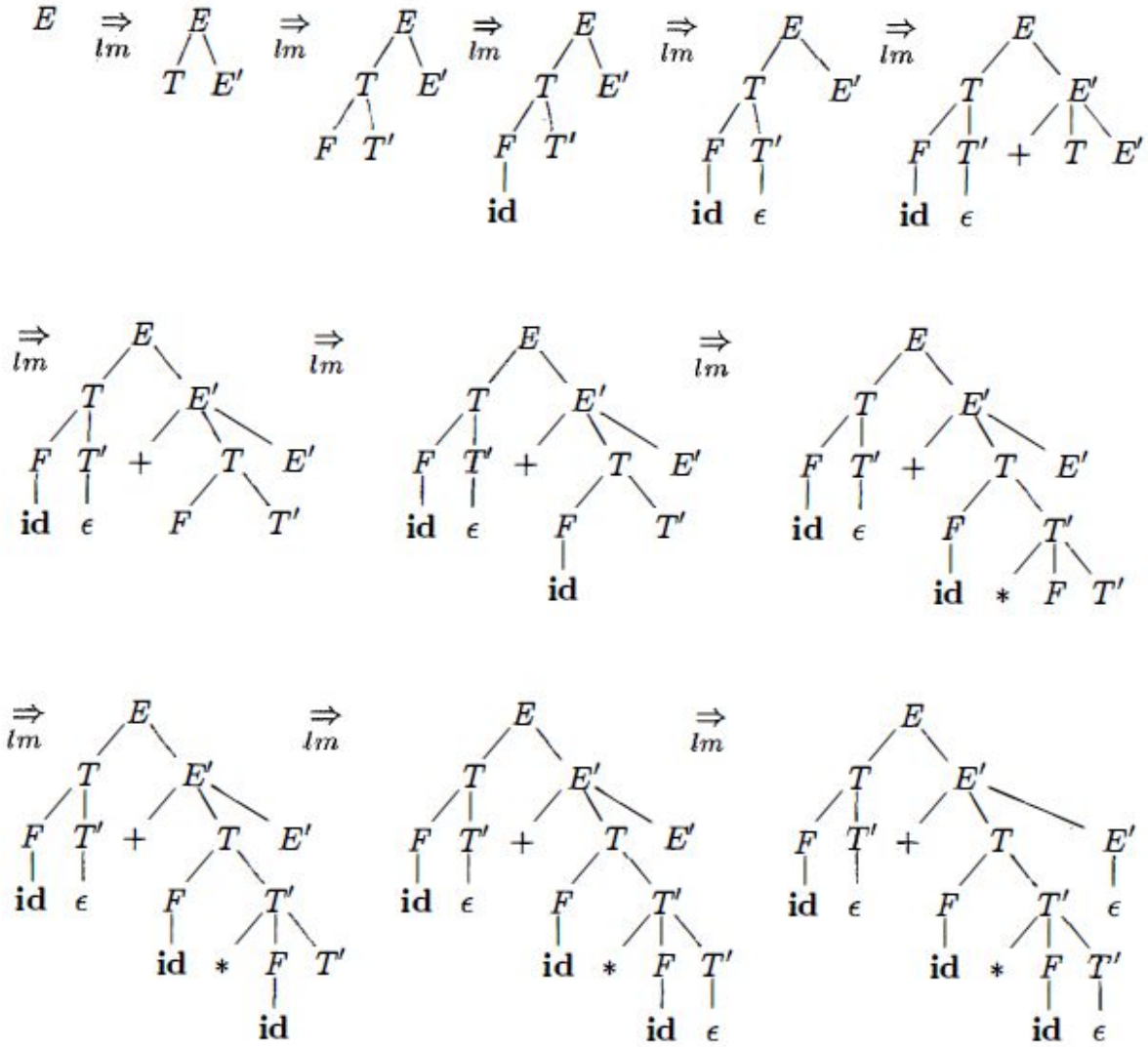


Figure 4.12: Top-down parse for **id + id * id**

Recursive-descent parsing

A recursive-descent parsing program consists of a set of procedures, one for each nonterminal. Execution begins with the procedure for the start symbol, which halts and announces success if its procedure body scans the entire input string.

```
    void A() {
1)      Choose an A-production, A → X1X2 ··· Xk;
2)      for ( i = 1 to k ) {
3)          if ( Xi is a nonterminal )
4)              call procedure Xi();
5)          else if ( Xi equals the current input symbol a )
6)              advance the input to the next symbol;
7)          else /* an error has occurred */;
        }
    }
```

Figure 4.13: A typical procedure for a nonterminal in a top-down parser

- General recursive descent may require backtracking
- The previous code needs to be modified to allow backtracking.
- In general form it cannot choose an appropriate production easily. So we need to try all alternatives If one fails, the input pointer needs to be reset and another alternative has to be tried.
- Recursive descent parsers cannot be used for left-recursive grammars

$$
\begin{array}{lcl}
S & \rightarrow & c\,A\,d \\
A & \rightarrow & a\,b \mid a
\end{array}
$$

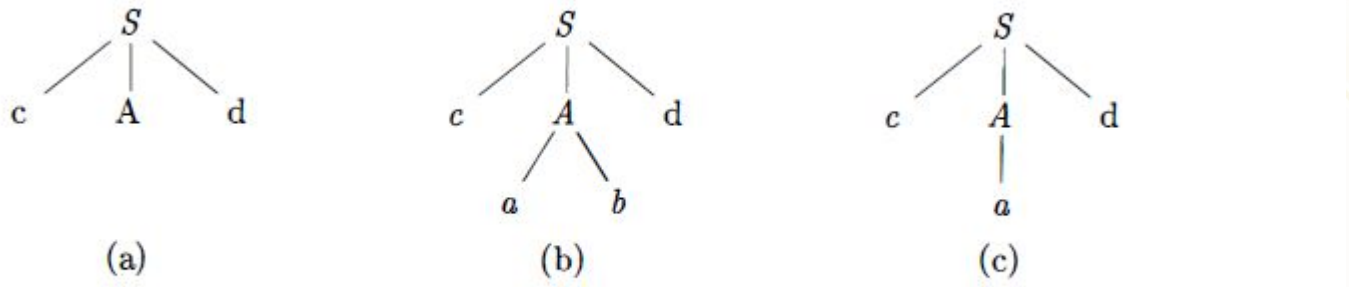


Figure 4.14: Steps in a top-down parse

FIRST & FOLLOW

FIRST(α) IS A SET OF TERMINALS THAT CAN BEGIN THE STRINGS DERIVED FROM α.

FOLLOW(A) FOR A NON TERMINAL 'A' IS A SET OF TERMINALS THAT CAN APPEAR IMMEDIATELY TO THE RIGHT OF A, IN SOME STRING DERIVED WITH THE CORRESPONDING GRAMMAR.

Algorithm for finding FIRST

1. IF x IS A TERMINAL, ADD x TO FIRST(x)

   Eg:- FIRST(a) = {a} Eg:- FIRST(aB) = {a}

2. IF x IS ∈, THEN ∈ CAN BE ADDED TO FIRST(x)

   FIRST(∈) = {∈}

3. IF X → A IS A PRODUCTION, THEN FIRST(X) = FIRST(A)

   Eg:- S → A
   A → a
   FIRST(S) = FIRST(A) = {a}

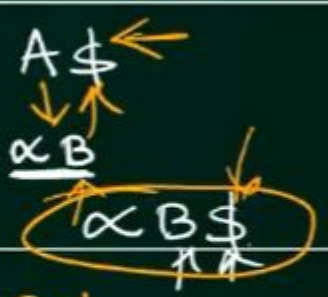
4. IF X → $Y_1 Y_2 Y_3 \cdots Y_n$, THEN FIRST(X) = FIRST(Y_1) {IF FIRST OF Y_1 CONTAINS ∈ THEN FIRST(X) = (FIRST(Y_1) − ∈) ∪ FIRST(Y_2)

Algorithm for finding FOLLOW

1. FOLLOW(S) WILL CONTAIN $ IF S IS A STARTING SYMBOL

2. IF $A \rightarrow \alpha B \beta$ THEN FOLLOW(B) = FIRST(β)
   Ex:- $A \rightarrow aBc$
   FOLLOW(B) = FIRST(c) = c
   Ex:- $A \rightarrow aBC$, $C \rightarrow d$
   FOLLOW(B) = FIRST(C) = {d}

3. IF $A \rightarrow \alpha B$ THEN FOLLOW(B) = FOLLOW(A)



4. IF $A \rightarrow \alpha B \beta$ AND IF FIRST(β) CONTAINS $\in$ THEN
   FOLLOW(B) = FIRST(β) $- \{\in\} \cup$ FOLLOW(A)

[1] Find the FIRST and FOLLOW

	FIRST	FOLLOW
S → A	FIRST(S) = FIRST(A) = {a}	FOLLOW(S) = {$}
A → a	FIRST(A) = {a}	FOLLOW(A) = {$}

[2] Find the FIRST and FOLLOW

	FIRST	FOLLOW
$S \rightarrow AB$	FIRST(S) = FIRST(A) $= \{a, b\}$	FOLLOW(S) = $\{\$\}$
$A \rightarrow a \mid \epsilon$	FIRST(A) = $\{a, \epsilon\}$	FOLLOW(A) = FIRST(B) $= \{b\}$
$B \rightarrow b$	FIRST(B) = $\{b\}$	FOLLOW(B) = FOLLOW(S) $= \{\$\}$

[3] Find the FIRST and FOLLOW

	FIRST	FOLLOW
$S \rightarrow AB$	FIRST(S) = FIRST(A) = {a, b, ∈}	FOLLOW(S) = {$}
$A \rightarrow a \mid \epsilon$	FIRST(A) = {a, ∈}	FOLLOW(A) = FIRST(B) = {b, $}
$B \rightarrow b \mid \epsilon$	FIRST(B) = {b, ∈}	FOLLOW(B) = FOLLOW(S) = {$}

[4] Find the FIRST and FOLLOW

	FIRST	FOLLOW
$S \rightarrow aAB$	FIRST(S) = FIRST(aAB) = {a}	FOLLOW(S) = {$}
$A \rightarrow b$	FIRST(A) = {b}	FOLLOW(A) = FIRST(B) = {c, $}
$B \rightarrow c \mid \epsilon$	FIRST(B) = {c, ∈}	FOLLOW(B) = FOLLOW(S) = {$}

[5] Find the FIRST and FOLLOW

	FIRST	FOLLOW
$S \rightarrow aABb$	FIRST(S) = FIRST(aABb) = {a}	FOLLOW(S) = {$}
$A \rightarrow c \mid \epsilon$	FIRST(A) = {c, ϵ}	FOLLOW(A) = FIRST(B) = {d, b}
$B \rightarrow d \mid \epsilon$	FIRST(B) = {d, ϵ}	FOLLOW(B) = FIRST(b) = {b}

[6] Find the FIRST and FOLLOW

	FIRST	FOLLOW
$S \rightarrow AaAb \mid BbBa$	$FIRST(S) = \{a, b\}$	$FOLLOW(S) = \{\$\}$
$A \rightarrow \epsilon$	$FIRST(A) = \epsilon$	$FOLLOW(A) = \{a, b\}$
$B \rightarrow \epsilon$	$FIRST(B) = \epsilon$	$FOLLOW(B) = \{a, b\}$

$$FIRST(S) = FIRST(\overset{\epsilon}{A}aAb) \cup FIRST(\overset{\epsilon}{B}bBa)$$

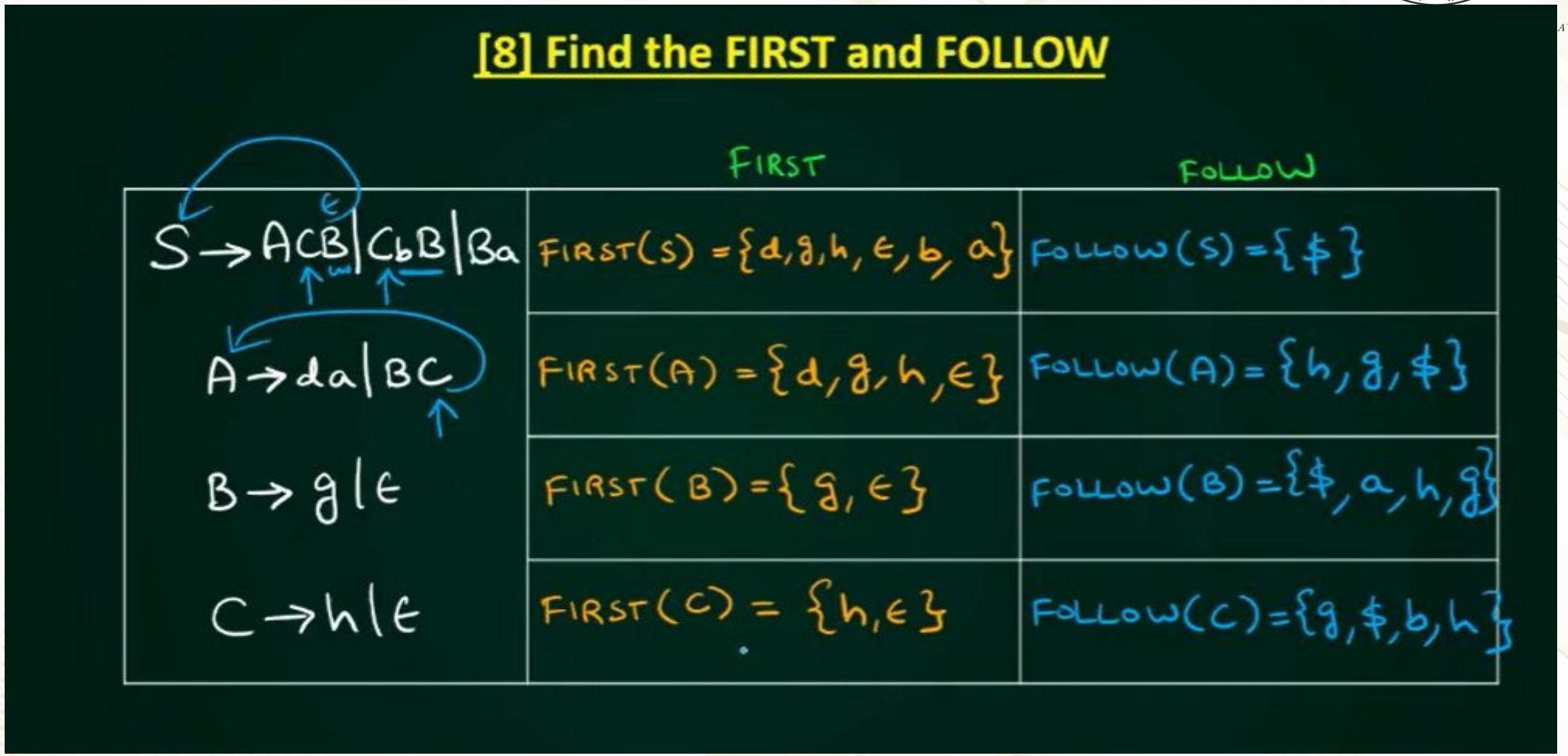
$$\downarrow \qquad \downarrow$$

$$\{a\} \qquad \{b\}$$

[7] Find the FIRST and FOLLOW

	FIRST	FOLLOW
$E \rightarrow TE'$	FIRST(E) = {(, id}	FOLLOW(E) = {$,)}
$E' \rightarrow +TE' \mid \epsilon$	FIRST(E') = {+, ϵ}	FOLLOW(E') = {$,)}
(1) $T \rightarrow FT'$	FIRST(T) = {(, id}	FOLLOW(T) = {+, $,)}
(2) $T' \rightarrow *FT' \mid \epsilon$	FIRST(T') = {*, ϵ}	FOLLOW(T') = {+, $,)}
$F \rightarrow (E) \mid id$	FIRST(F) = {(, id}	FOLLOW(F) = {*, +, $,)}

[8] Find the FIRST and FOLLOW



[9] Find the FIRST and FOLLOW

	FIRST	FOLLOW
$S \rightarrow aBDh$	FIRST(S) = {a}	FOLLOW(S) = {$}
$B \rightarrow cC$	FIRST(B) = {c}	FOLLOW(B) = {g, f, h}
$C \rightarrow bC \mid \epsilon$	FIRST(C) = {b, ∈}	FOLLOW(C) = {g, f, h}
$D \rightarrow EF$	FIRST(D) = {g, f, ∈}	FOLLOW(D) = {h}
$E \rightarrow g \mid \epsilon$	FIRST(E) = {g, ∈}	FOLLOW(E) = {f, h}
$F \rightarrow f \mid \epsilon$	FIRST(F) = {f, ∈}	FOLLOW(F) = {h}

[10] Find the FIRST and FOLLOW

	FIRST	FOLLOW
$S \rightarrow ABCDE$	$FIRST(S) = \{a, b, c\}$	$FOLLOW(S) = \{\$\}$
$A \rightarrow a \mid \epsilon$	$FIRST(A) = \{a, \epsilon\}$	$FOLLOW(A) = \{b, c\}$
$B \rightarrow b \mid \epsilon$	$FIRST(B) = \{b, \epsilon\}$	$FOLLOW(B) = \{c\}$
$C \rightarrow c$	$FIRST(C) = \{c\}$	$FOLLOW(C) = \{d, e, \$\}$
$D \rightarrow d \mid \epsilon$	$FIRST(D) = \{d, \epsilon\}$	$FOLLOW(D) = \{e, \$\}$
$E \rightarrow e \mid \epsilon$	$FIRST(E) = \{e, \epsilon\}$	$FOLLOW(E) = \{\$\}$

LL(1) Parsing Table Construction

STEP-1 : FOR EACH PRODUCTION A->α OF A GRAMMAR DO STEP-2 & STEP-3

STEP-2 : FOR EACH TERMINAL a IN FIRST(α) ADD A->α TO M[A, a]

STEP-3 : IF ϵ IS IN THE FIRST(α), ADD A->α TO M[A, b], FOR EACH SYMBOL b IN FOLLOW(A)

STEP-4 : MAKE EACH UNDEFINED ENTRY OF M TO BE AN ERROR

[1] LL(1) Parsing Table Construction

	FIRST	FOLLOW
① S→(S) ② | ϵ	FIRST((S)) = {(}	FOLLOW(S) = {), $}
	FIRST(ϵ) = {ϵ}	

	(	)	$
S	S→(S)	S→ϵ	S→ϵ

[2] LL(1) Parsing Table Construction

	FIRST	FOLLOW
① $S \to AaAb$ | ② $BbBa$	FIRST $AaAb) = \{a\}$	
	FIRST $(BbBa) = \{b\}$	
③ $A \to \epsilon$	FIRST$(\epsilon) = \epsilon$	FOLLOW$(A) = \{a, b\}$
④ $B \to \epsilon$	FIRST$(\epsilon) = \epsilon$	FOLLOW$(B) = \{a, b\}$

	a	b	$
S	$S \to AaAb$	$S \to BbBa$	
A	$A \to \epsilon$	$A \to \epsilon$	
B	$B \to \epsilon$	$B \to \epsilon$	

[3] LL(1) Parsing Table Construction

	FIRST	FOLLOW
(1) $S \to ABC$	FIRST(ABC) = {a, b, c, ε}	FOLLOW(S) = {$}
(2) $A \to a$ | (3) $A \to \epsilon$	FIRST(a) = {a}	FOLLOW(A) = {b, c, $}
	FIRST(ε) = {ε}	
(4) $B \to b$ | (5) $B \to \epsilon$	FIRST(b) = {b}	FOLLOW(B) = {c, $}
	FIRST(ε) = {ε}	
(6) $C \to c$ | (7) $C \to \epsilon$	FIRST(c) = {c}	FOLLOW(C) = {$}
	FIRST(ε) = {ε}	

	a	b	c	$
S	S→ABC	S→ABC	S→ABC	S→ABC
A	A→a	A→ε	A→ε	A→ε
B		B→b	B→ε	B→ε
C			C→c	C→ε

STRING : a b c

STACK	INPUT
$S	abc$
$CBA	abc$
$CB	bc$
$C	c$
$	$

ACCEPTED

[4] LL(1) Parsing Table Construction

STRING: id + id

(1) $E \to TE'$	FIRST(TE') = {(, id}	
(2) $E' \to +TE'$ | (3) ϵ	FIRST$(+TE')$ = {+} FIRST(ϵ) = {ϵ}	FOLLOW(E') = {$,)}
(4) $T \to FT'$	FIRST(FT') = {(, id}	
(5) $T' \to *FT'$ | (6) ϵ	FIRST$(*FT')$ = {*} FIRST(ϵ) = {ϵ}	FOLLOW(T') = {+, $,)}
(7) $F \to (E)$ | (8) id	FIRST$((E))$ = {(} FIRST(id) = {id}	

STACK: $E

INPUT: id + id $

	id	+	*	(	)	$
E	$E \to TE'$	E	E	$E \to TE'$	E	E
E'	E	$E' \to +TE'$	E	E	$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$	E	E	$T \to FT'$	E	E
T'	E	$T' \to \epsilon$	$T' \to *FT'$	E	$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to id$	E	E	$F \to (E)$	E	E

[5] LL(1) Parsing Table Construction

$S \to iEtSS' \mid a$ (1) (2)	FIRST(iEtSS') = {i}	
	FIRST(a) = {a}	
$S' \to eS \mid \epsilon$ (3) (4)	FIRST(eS) = {e}	FOLLOW(S') = {$, e}
	FIRST(ϵ) = {ϵ}	
$E \to b$ (5)	FIRST(b) = {b}	

	i	a	e	b	t	$
S	$S \to iEtSS'$	$S \to a$				
S'			$S' \to eS$ $S' \to \epsilon$			$S' \to \epsilon$
E				$E \to b$		

THIS IS NOT AN LL(1) GRAMMAR

Properties of LL(1) Parser

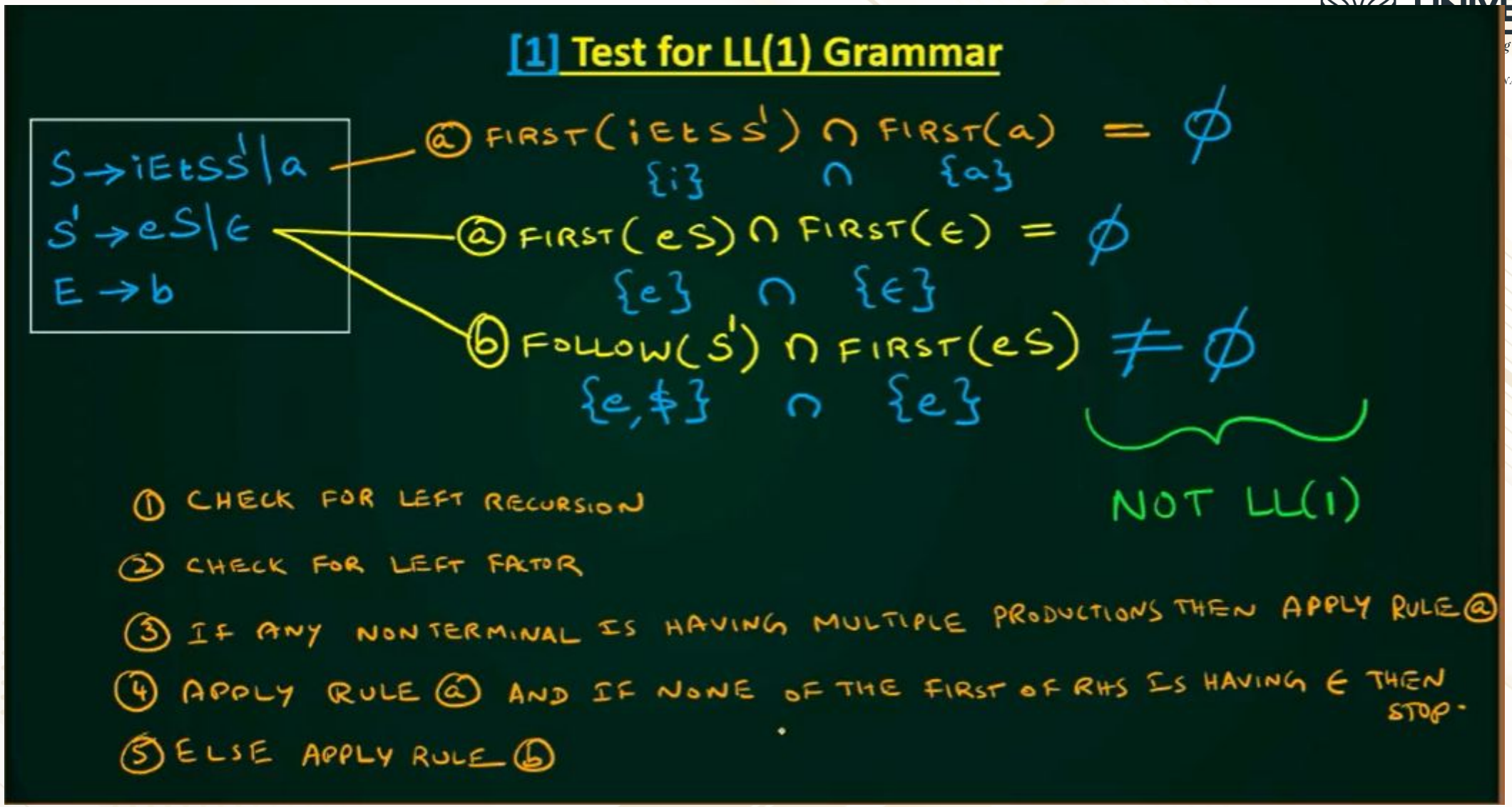
1 : NO AMBIGUOUS OR LEFT RECURSIVE GRAMMAR CAN BE LL(1)

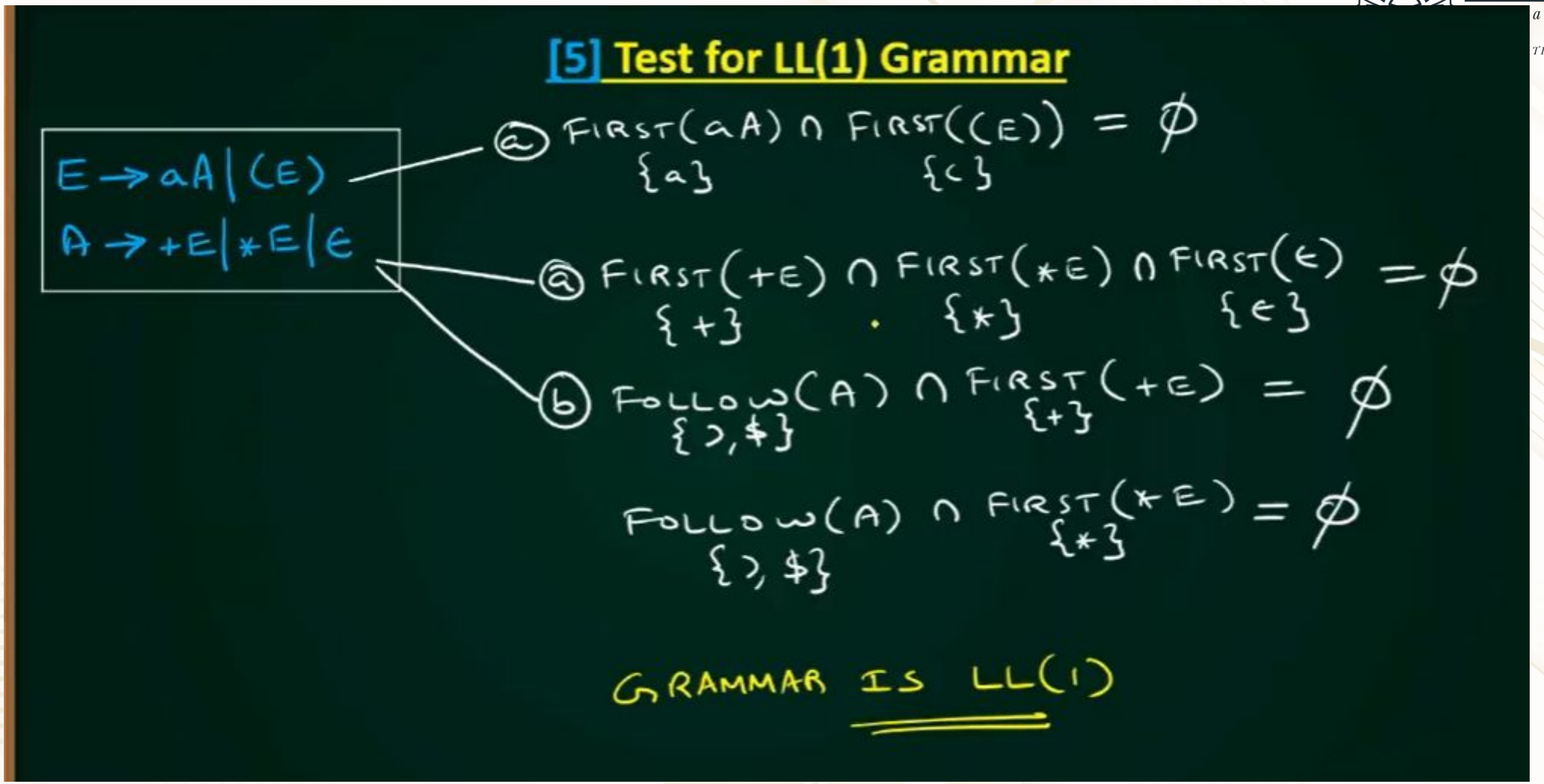
2 : ENTRIES OF PRODUCTIONS IN A FIELD OF THE PARSING TABLE SHOULD BE ATMOST ONE

3 : A GRAMMAR G IS LL(1) IFF THE FOLLOWING CONDITIONS HOLD FOR PRODUCTIONS OF THE FORM A->α | β

(a) FIRST(α) ∩ FIRST(β) = Φ

(b) FIRST(α) ∩ FOLLOW(A) = Φ IF FIRST(β) CONTAINS ϵ
FIRST(β) ∩ FOLLOW(A) = Φ IF FIRST(α) CONTAINS ϵ


[1] Test for LL(1) Grammar
S → iEtSS' | a
S' → eS | ϵ
E → b
ⓐ FIRST(iEtSS') ∩ FIRST(a) = ϕ
{i} ∩ {a}
ⓐ FIRST(eS) ∩ FIRST(ϵ) = ϕ
{e} ∩ {ϵ}
ⓑ FOLLOW(S') ∩ FIRST(eS) ≠ ϕ
{e,$} ∩ {e}
NOT LL(1)
① CHECK FOR LEFT RECURSION
② CHECK FOR LEFT FACTOR
③ IF ANY NONTERMINAL IS HAVING MULTIPLE PRODUCTIONS THEN APPLY RULE ⓐ
④ APPLY RULE ⓐ AND IF NONE OF THE FIRST OF RHS IS HAVING ϵ THEN STOP.
⑤ ELSE APPLY RULE ⓑ


[5] Test for LL(1) Grammar
E → aA | (E)
A → +E | *E | ϵ
a FIRST(aA) ∩ FIRST((E)) = ϕ
{a} {(}
a FIRST(+E) ∩ FIRST(*E) ∩ FIRST(ϵ) = ϕ
{+} {*} {ϵ}
b FOLLOW(A) ∩ FIRST(+E) = ϕ
{), $} {+}
FOLLOW(A) ∩ FIRST(*E) = ϕ
{), $} {*}
GRAMMAR IS LL(1)

LR(0) Items

LR(0) ITEM OF A GRAMMAR G IS A PRODUCTION OF G WITH DOT {.} AT SOME POSITION.

G : •

AN LR(0) ITEM A->X.YZ INDICATE THAT SOME PART OF INPUT STRING DERIVATION IS COMPLETED USING
AND THE REMAINING INPUT SENTENCE IS EXPECTED TO DERIVE USING YZ

STEPS TO COMPUTE LR(0) ITEMS

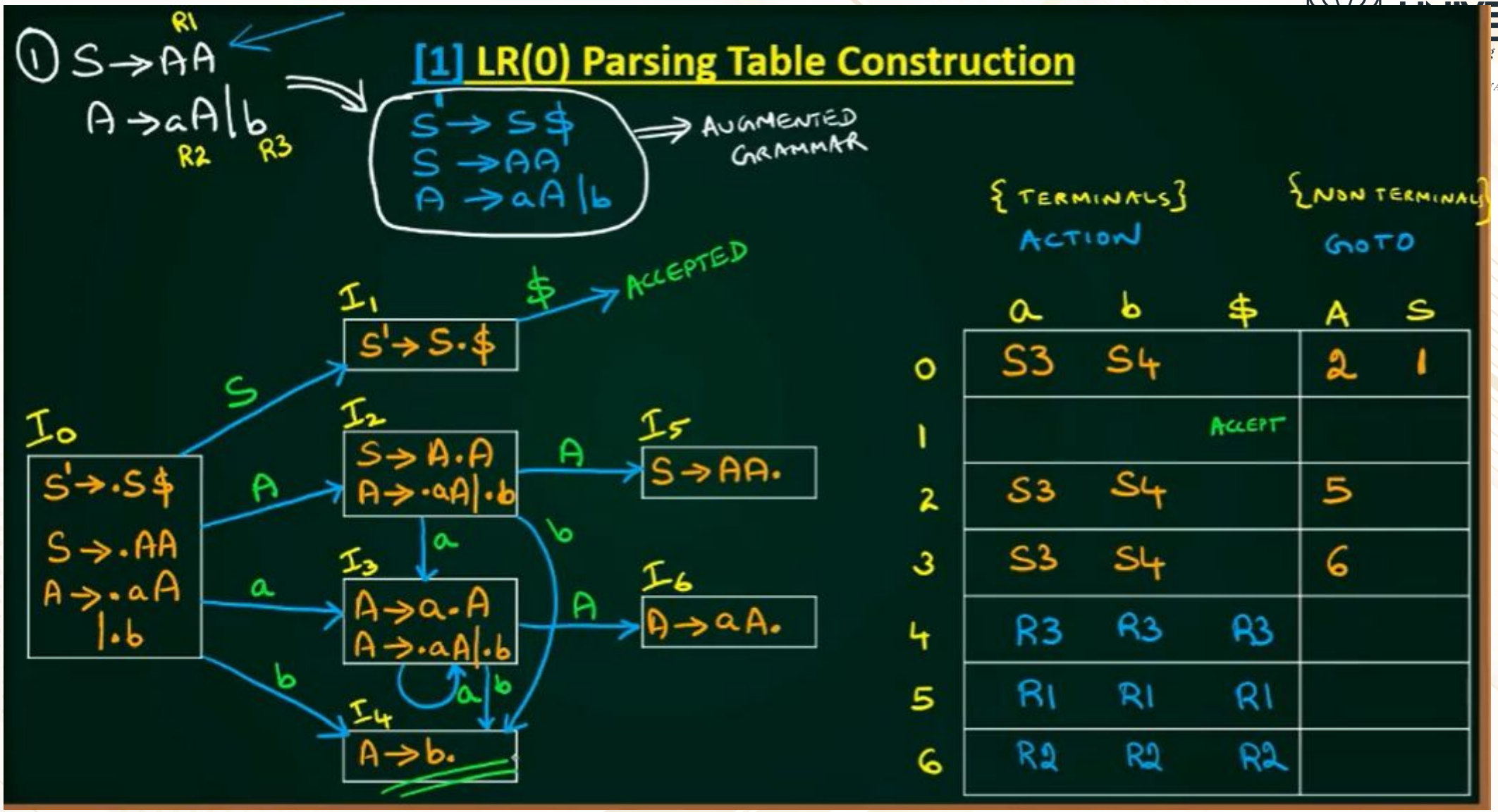
AUGMENT THE GRAMMAR WITH THE PRODUCTION S^{I} ->S$

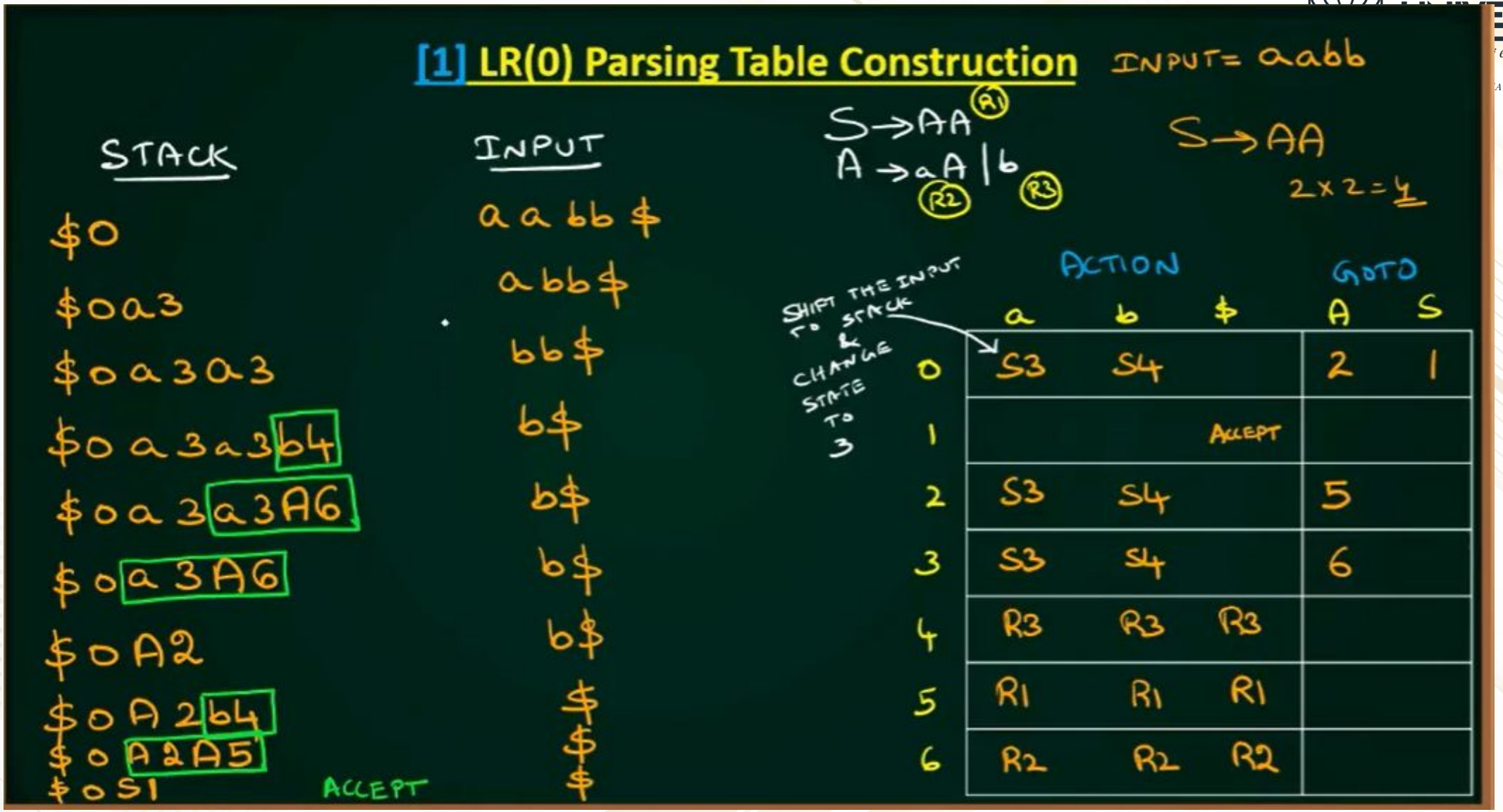
FIND CLOSURE

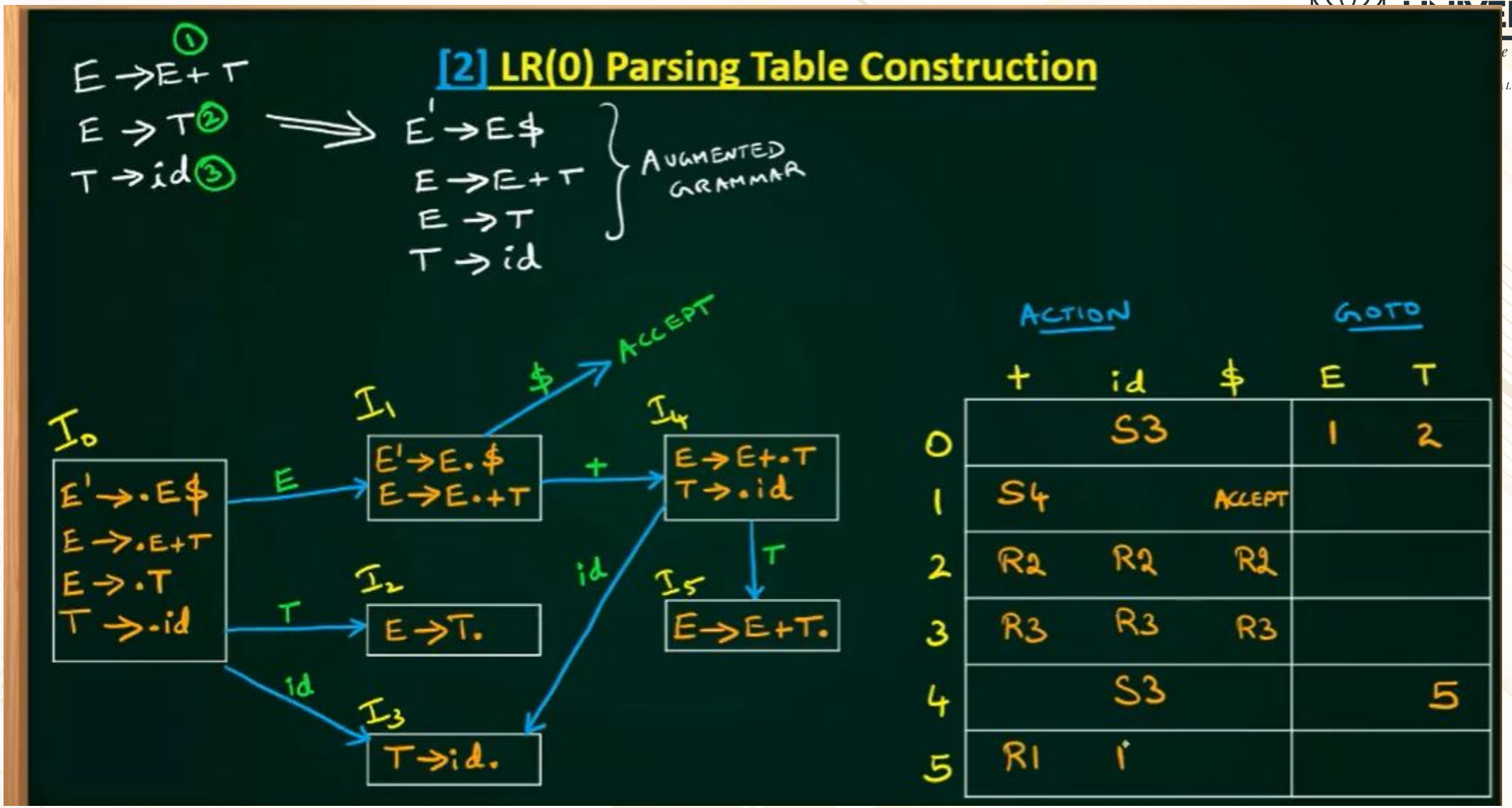
IF A->α.Bβ IS IN ITEM I AND B->α IS A PRODUCTION THEN ADD B->.α TO CLOSURE(I)

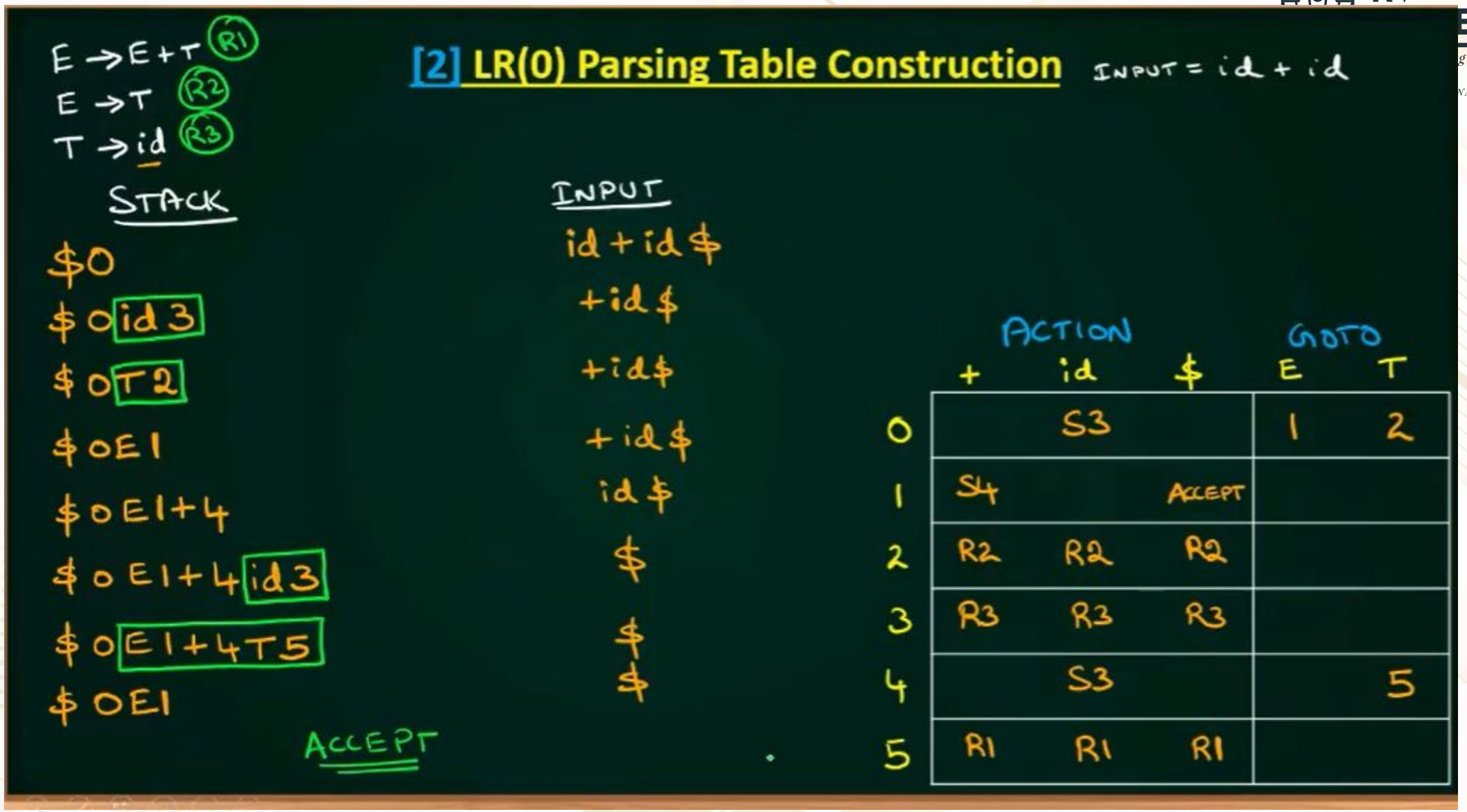
FIND GOTO

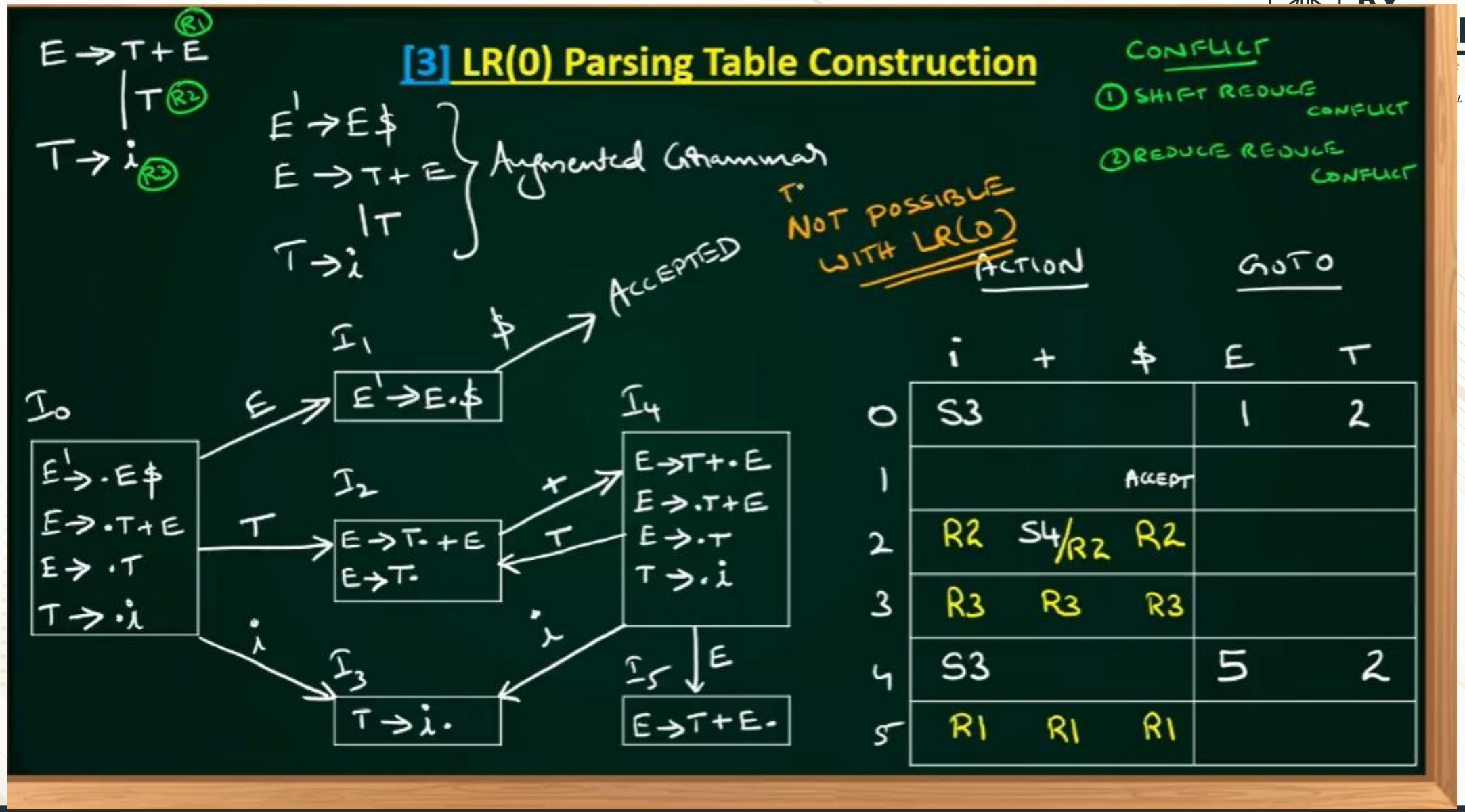
IF A->α.Bβ IS IN I AND A-> AB.β IS IN J, THEN GOTO(I,B) = J

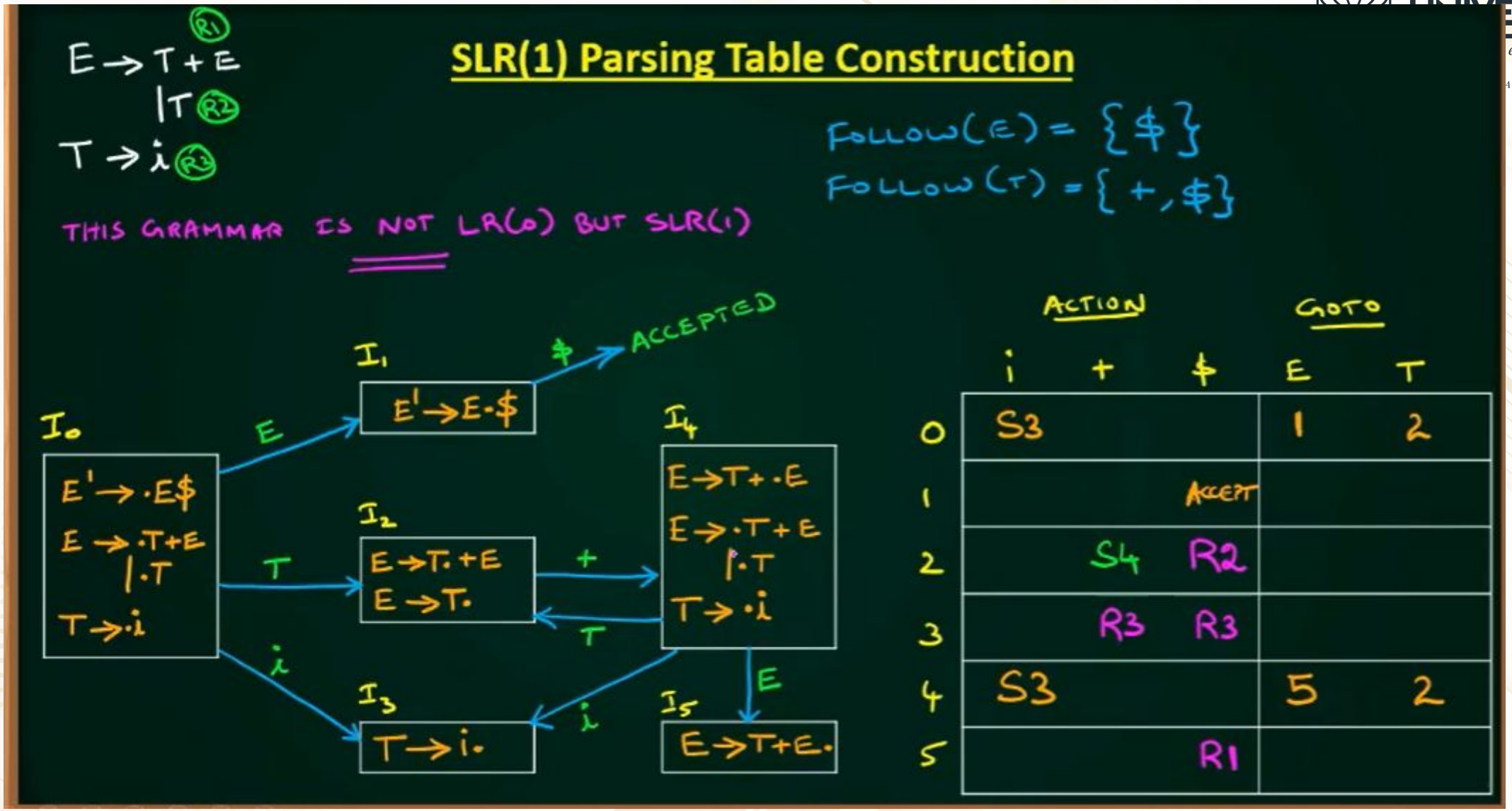

[1] LR(0) Parsing Table Construction
R1
① S→AA
A→aA|b
R2 R3
S→S$
S→AA
A→aA|b
AUGMENTED GRAMMAR
I0
S'→.S$
S→.AA
A→.aA
|.b
I1
S'→S.$
$ ACCEPTED
I2
S→A.A
A→.aA|.b
I3
A→a.A
A→.aA|.b
I4
A→b.
I5
S→AA.
I6
A→aA.
S
A
a
b
{TERMINALS}
ACTION
{NON TERMINALS}
GOTO
a b $ A S
0 S3 S4 2 1
1 ACCEPT
2 S3 S4 5
3 S3 S4 6
4 R3 R3 R3
5 R1 R1 R1
6 R2 R2 R2




[2] LR(0) Parsing Table Construction
E → E + T ①
E → T ②
T → id ③
E' → E $
E → E + T
E → T
T → id
AUGMENTED GRAMMAR
I0
E' → . E $
E → . E + T
E → . T
T → . id
I1
E' → E . $
E → E . + T
I2
E → T .
I3
T → id .
I4
E → E + . T
T → . id
I5
E → E + T .
ACCEPT
ACTION
GOTO
+ id $ E T
0 S3 1 2
1 S4 ACCEPT
2 R2 R2 R2
3 R3 R3 R3
4 S3 5
5 R1


E → E+T R1
E → T R2
T → id R3
[2] LR(0) Parsing Table Construction
INPUT = id + id
STACK
INPUT
$0
id + id $
$0 id 3
+ id $
$0 T 2
+ id $
$0 E 1
+ id $
$0 E 1 + 4
id $
$0 E 1 + 4 id 3
$
$0 E 1 + 4 T 5
$
$0 E 1
$
ACCEPT
ACTION
GOTO
+ id $ E T
0 S3 1 2
1 S4 ACCEPT
2 R2 R2 R2
3 R3 R3 R3
4 S3 5
5 R1 R1 R1


[3] LR(0) Parsing Table Construction
E → T + E (R1)
T (R2)
T → i (R3)
E' → E$
E → T + E
T
T → i
Augmented Grammar
CONFLICT
1 SHIFT REDUCE CONFLICT
2 REDUCE REDUCE CONFLICT
NOT POSSIBLE WITH LR(0)
ACCEPTED
I0
E' → .E$
E → .T + E
E → .T
T → .i
I1
E' → E.$
I2
E → T. + E
E → T.
I3
T → i.
I4
E → T + .E
E → .T + E
E → .T
T → .i
I5
E → T + E.
ACTION
GOTO
i + $ E T
0 S3 1 2
1 ACCEPT
2 R2 S4/R2 R2
3 R3 R3 R3
4 S3 5 2
5 R1 R1 R1

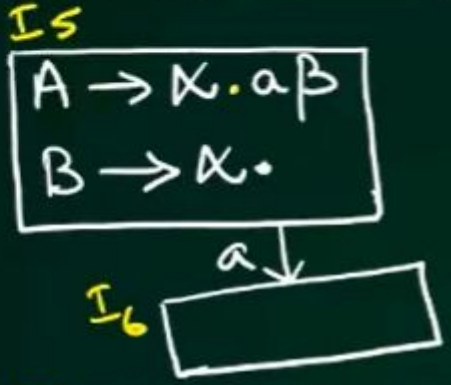

E → T + E (R1)
| T (R2)
T → i (R3)
SLR(1) Parsing Table Construction
FOLLOW(E) = {$}
FOLLOW(T) = {+, $}
THIS GRAMMAR IS NOT LR(0) BUT SLR(1)
I0
E' → .E$
E → .T+E
| .T
T → .i
I1
E' → E.$
$ ACCEPTED
I2
E → T.+E
E → T.
I3
T → i.
I4
E → T+.E
E → .T+E
| .T
T → .i
I5
E → T+E.
E
T
i
+
ACTION
GOTO
i + $ E T
0 S3 1 2
1 ACCEPT
2 S4 R2
3 R3 R3
4 S3 5 2
5 R1

Parser Conflicts in LR(0) Parser

① **SHIFT-REDUCE CONFLICT** OCCURS IF THE LR(0) ITEM CONTAINS PRODUCTIONS OF THE FORM

A->α.aβ and B->α.

R1 R2

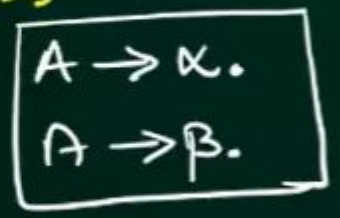


	a	—	—	—	—	—
5	S6/R2					

② **REDUCE-REDUCE CONFLICT** OCCURS IF LR(0) ITEM CONTAINS PRODUCTIONS OF THE FORM

A->α. and B->β.

R1 R2



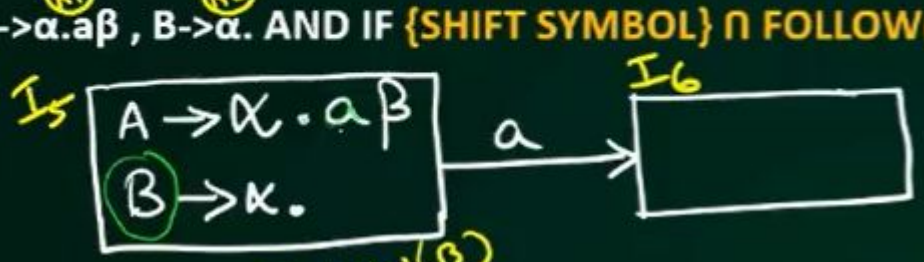
ACTION GOTO

	a	b	c
5	R1/R2	R1/R2	R1/R2

Parser Conflicts in SLR(1) Parser

SHIFT-REDUCE CONFLICT OCCURS IF THE LR(0) ITEM CONTAINS PRODUCTIONS OF THE FORM

A->α.aβ (R1) , B->α. (R2) AND IF {SHIFT SYMBOL} ∩ FOLLOW(B) ≠ Φ



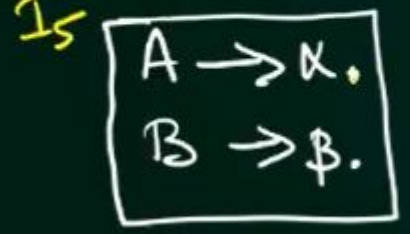
a ∩ Follow(B) {a, b}

Follow(B) = Reduce R2

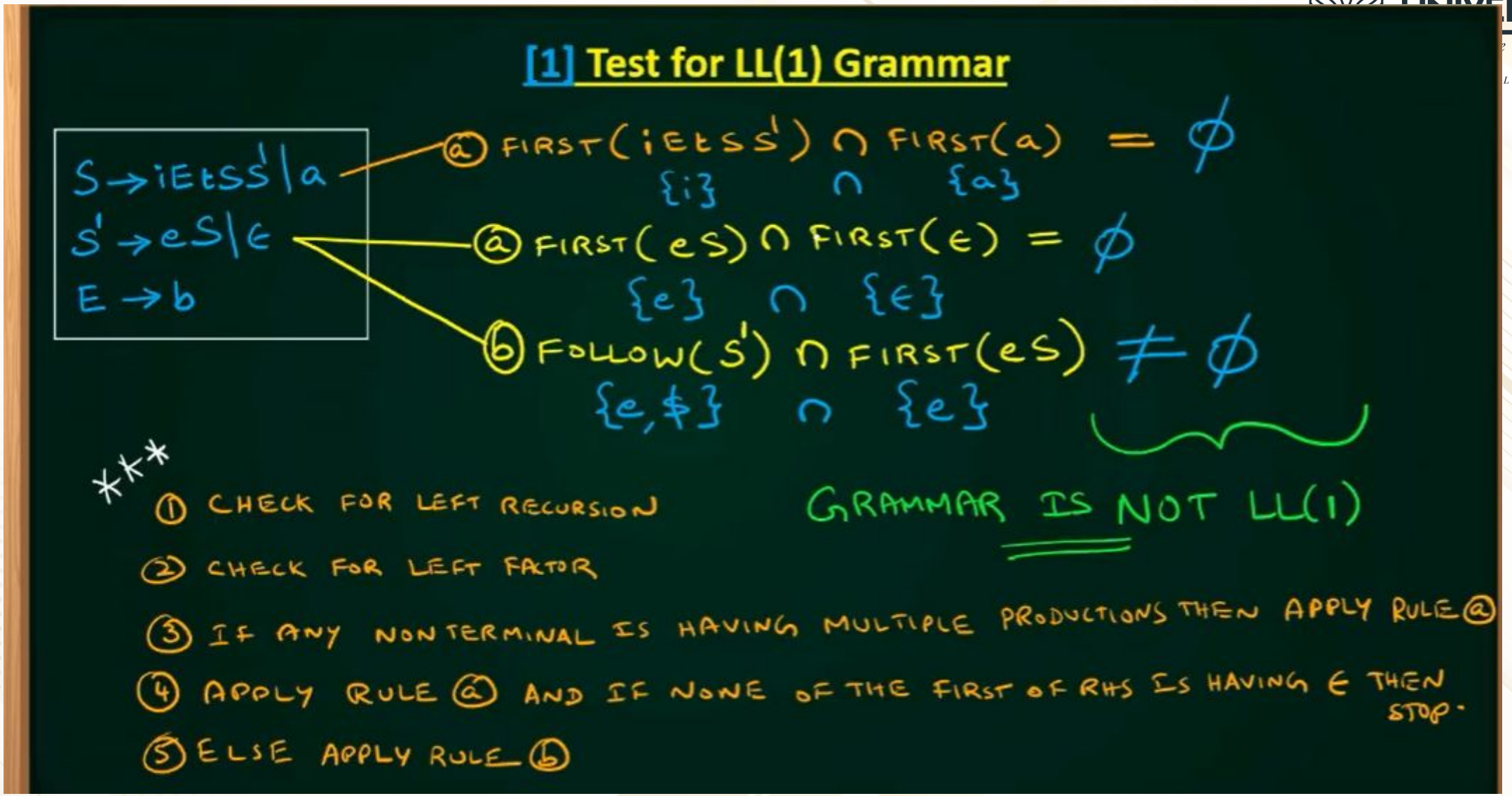
	ACTION			GOTO	
	a	b	$	A	B
5	S6/R2				

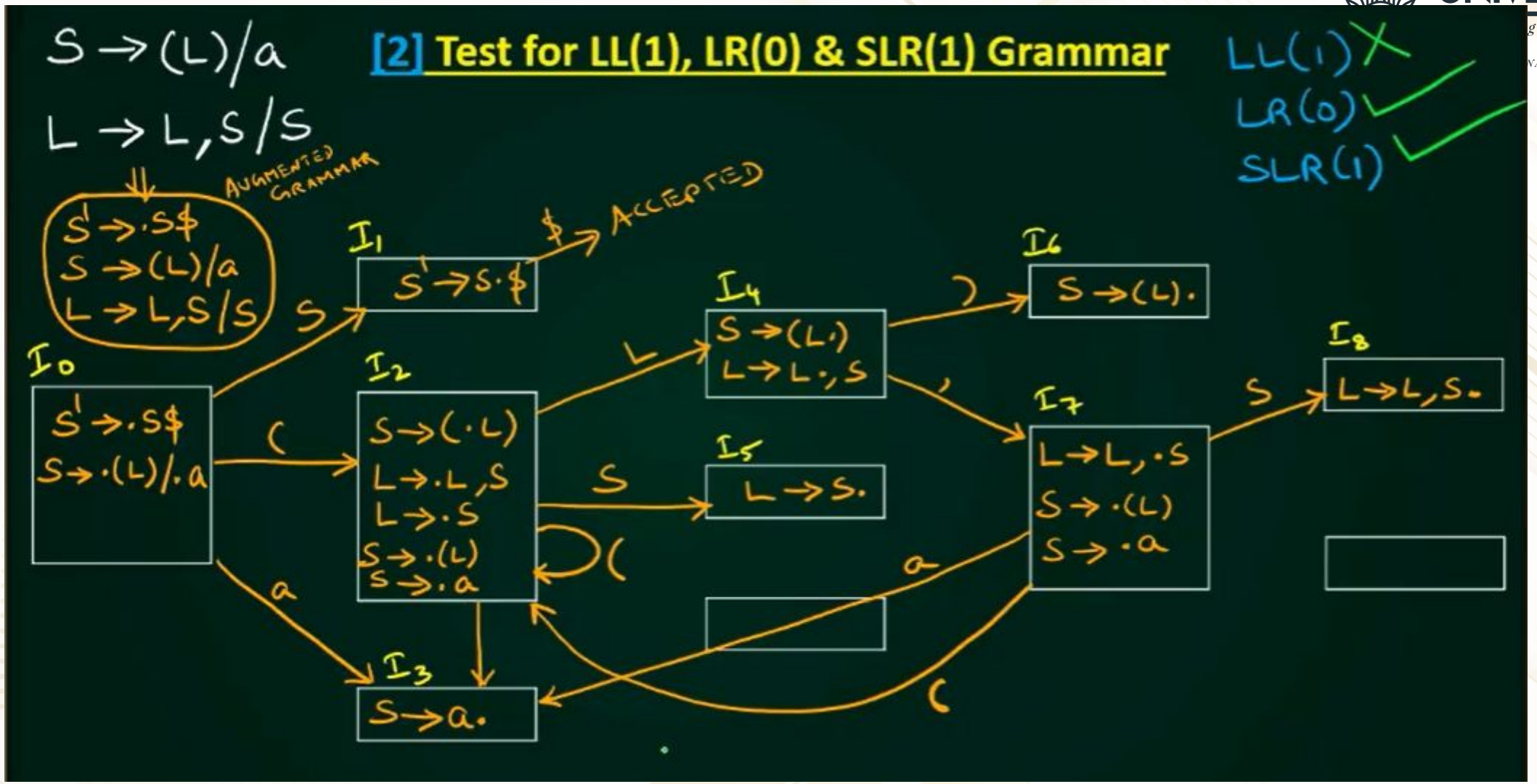
REDUCE-REDUCE CONFLICT OCCURS IF LR(0) ITEM CONTAINS PRODUCTIONS OF THE FORM

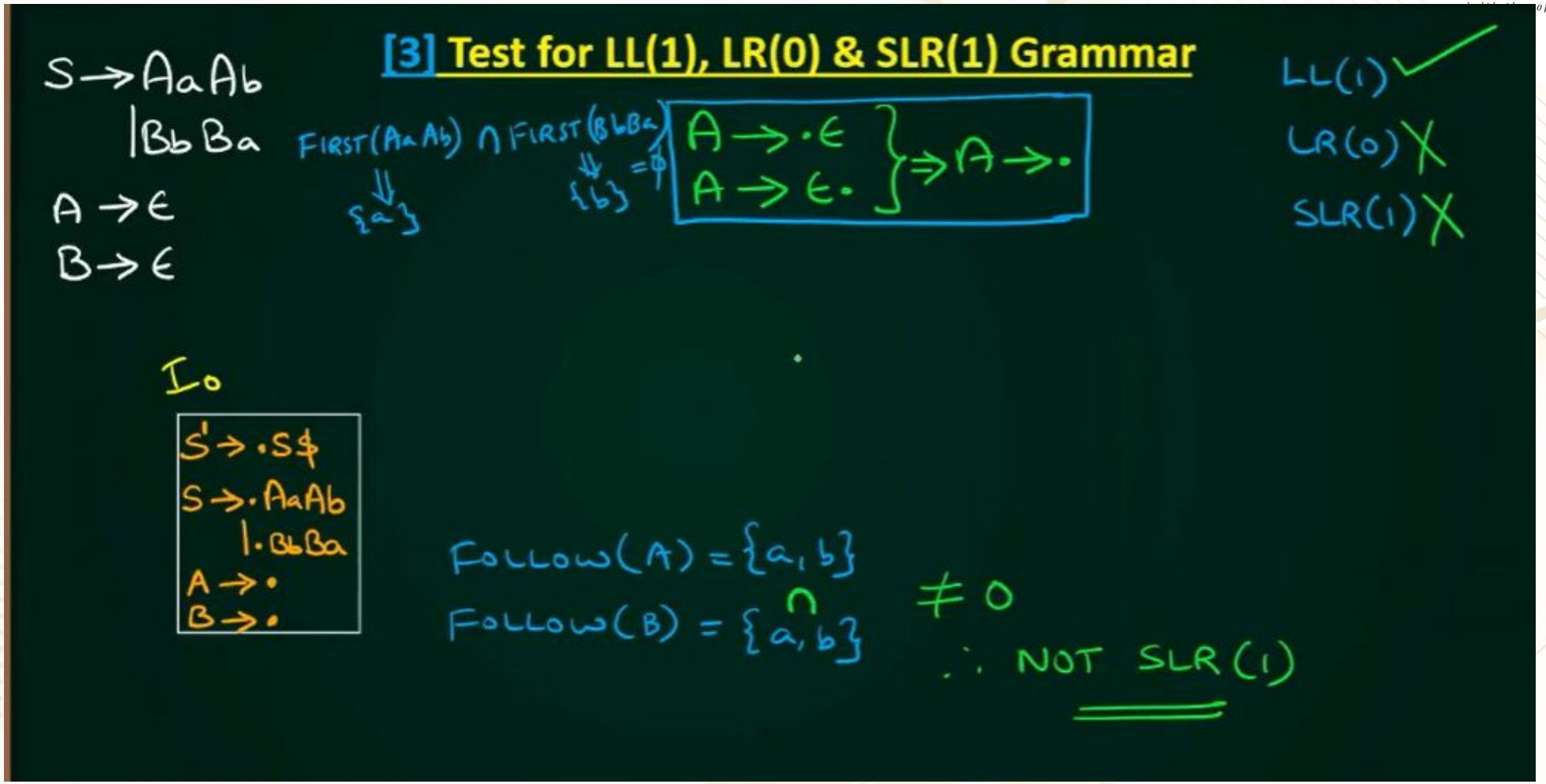
A->α. and B->β. AND IF FOLLOW(A) ∩ FOLLOW(B) ≠ Φ

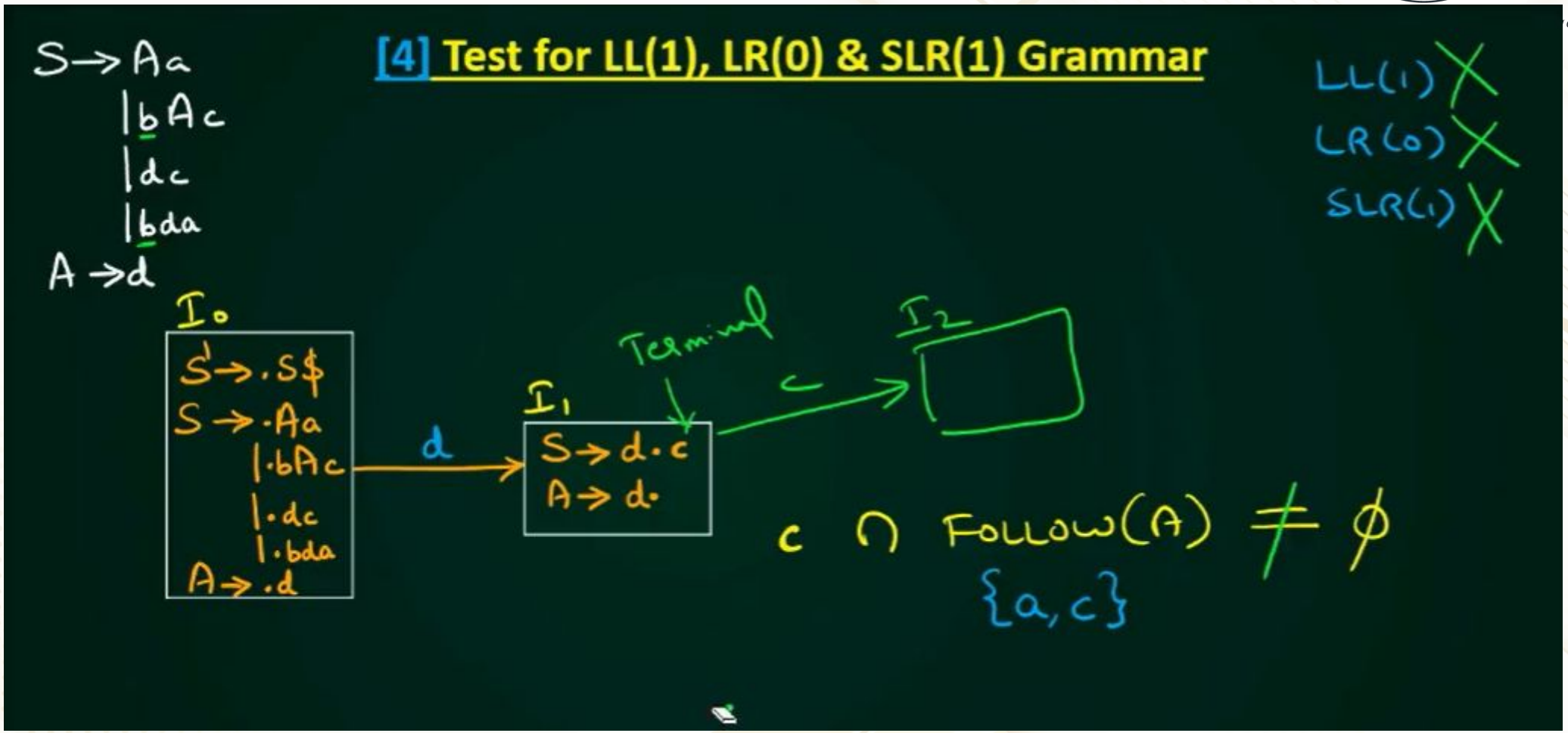


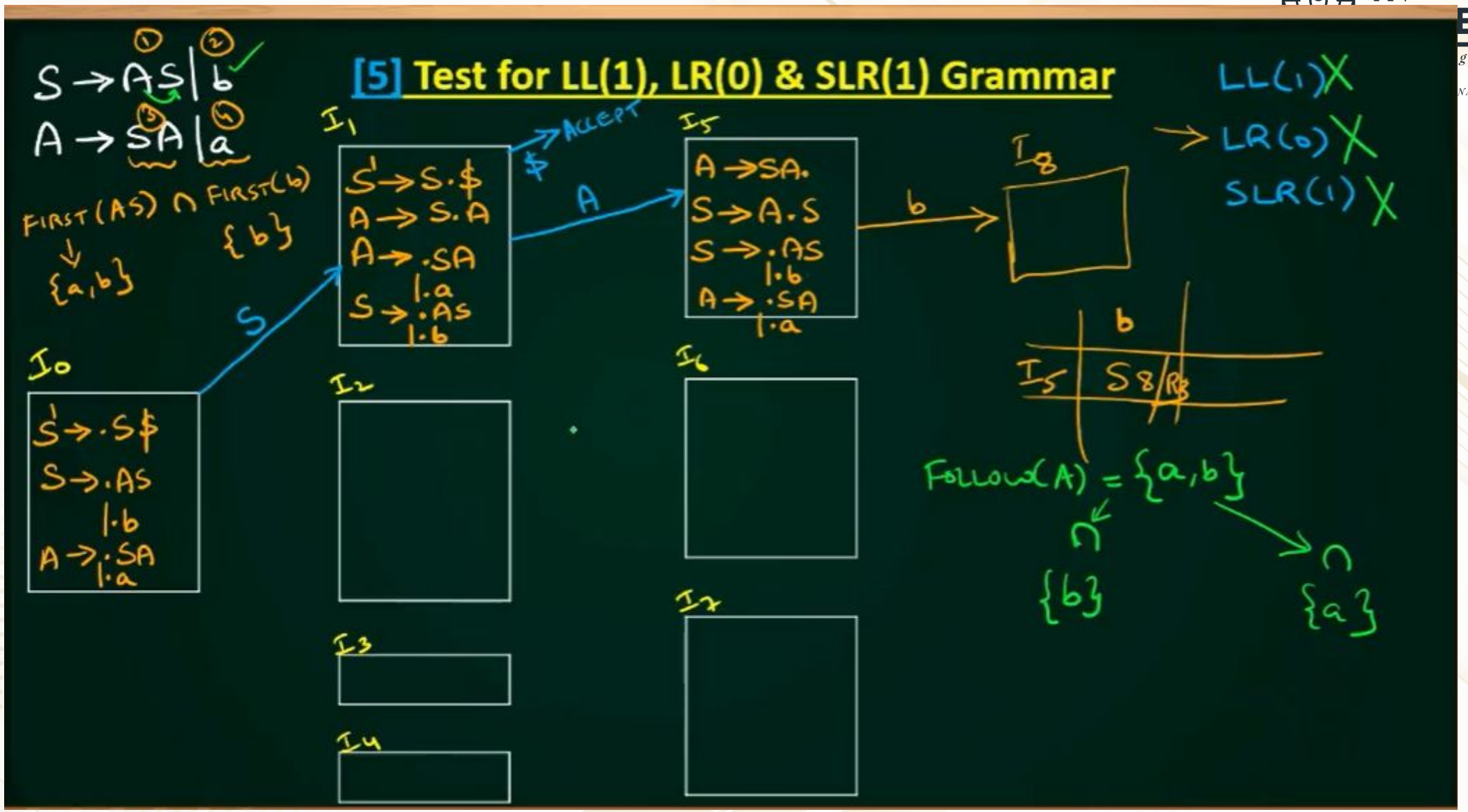
	ACTION			GOTO	
	a	b	$	A	B
5					

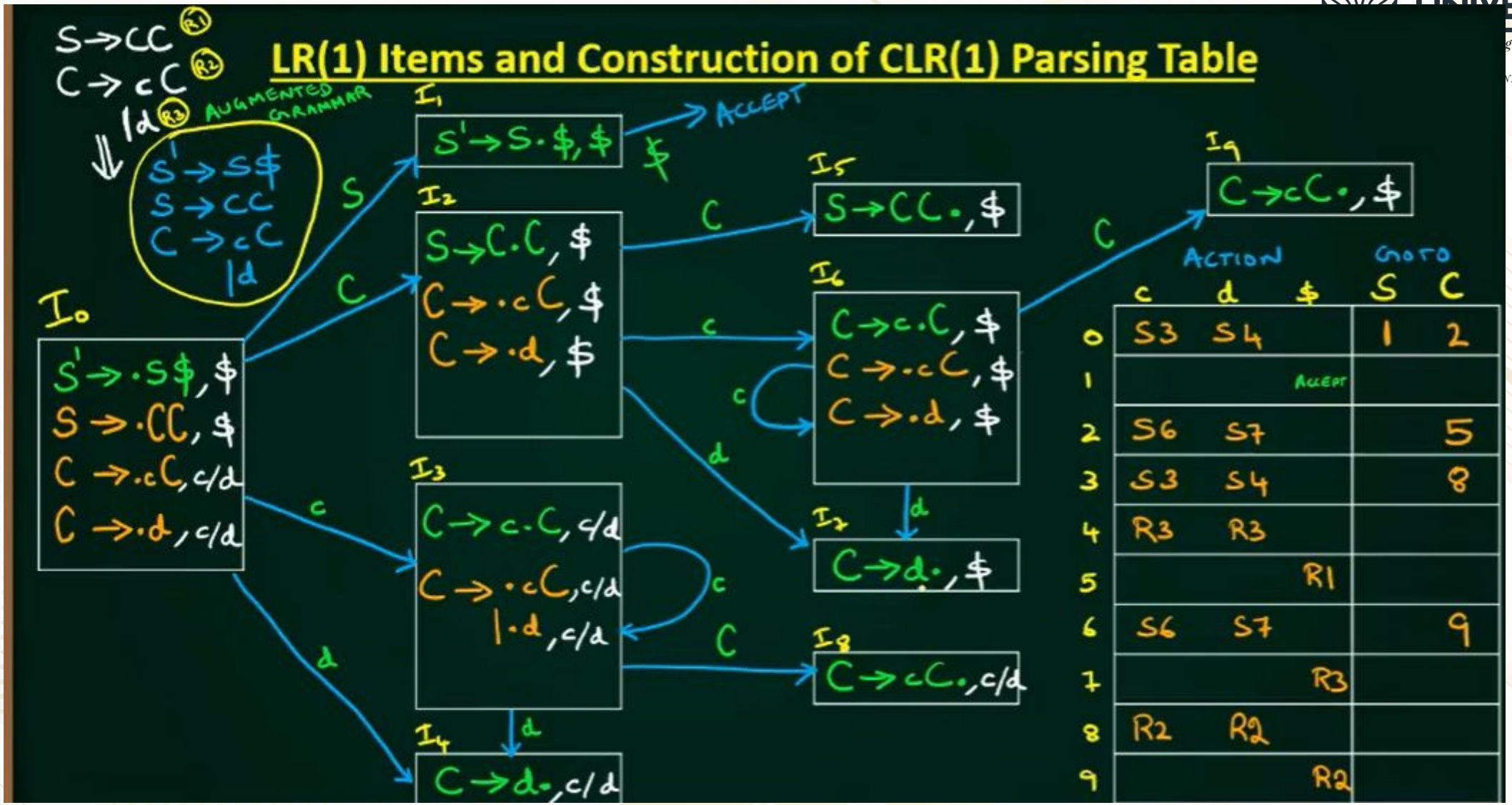

[1] Test for LL(1) Grammar
S → iEtSS' | a
S' → eS | ϵ
E → b
a FIRST(iEtSS') ∩ FIRST(a) = ϕ
{i} ∩ {a}
a FIRST(eS) ∩ FIRST(ϵ) = ϕ
{e} ∩ {ϵ}
b FOLLOW(S') ∩ FIRST(eS) ≠ ϕ
{e,$} ∩ {e}
GRAMMAR IS NOT LL(1)
***
1 CHECK FOR LEFT RECURSION
2 CHECK FOR LEFT FACTOR
3 IF ANY NONTERMINAL IS HAVING MULTIPLE PRODUCTIONS THEN APPLY RULE a
4 APPLY RULE a AND IF NONE OF THE FIRST OF RHS IS HAVING ϵ THEN STOP.
5 ELSE APPLY RULE b

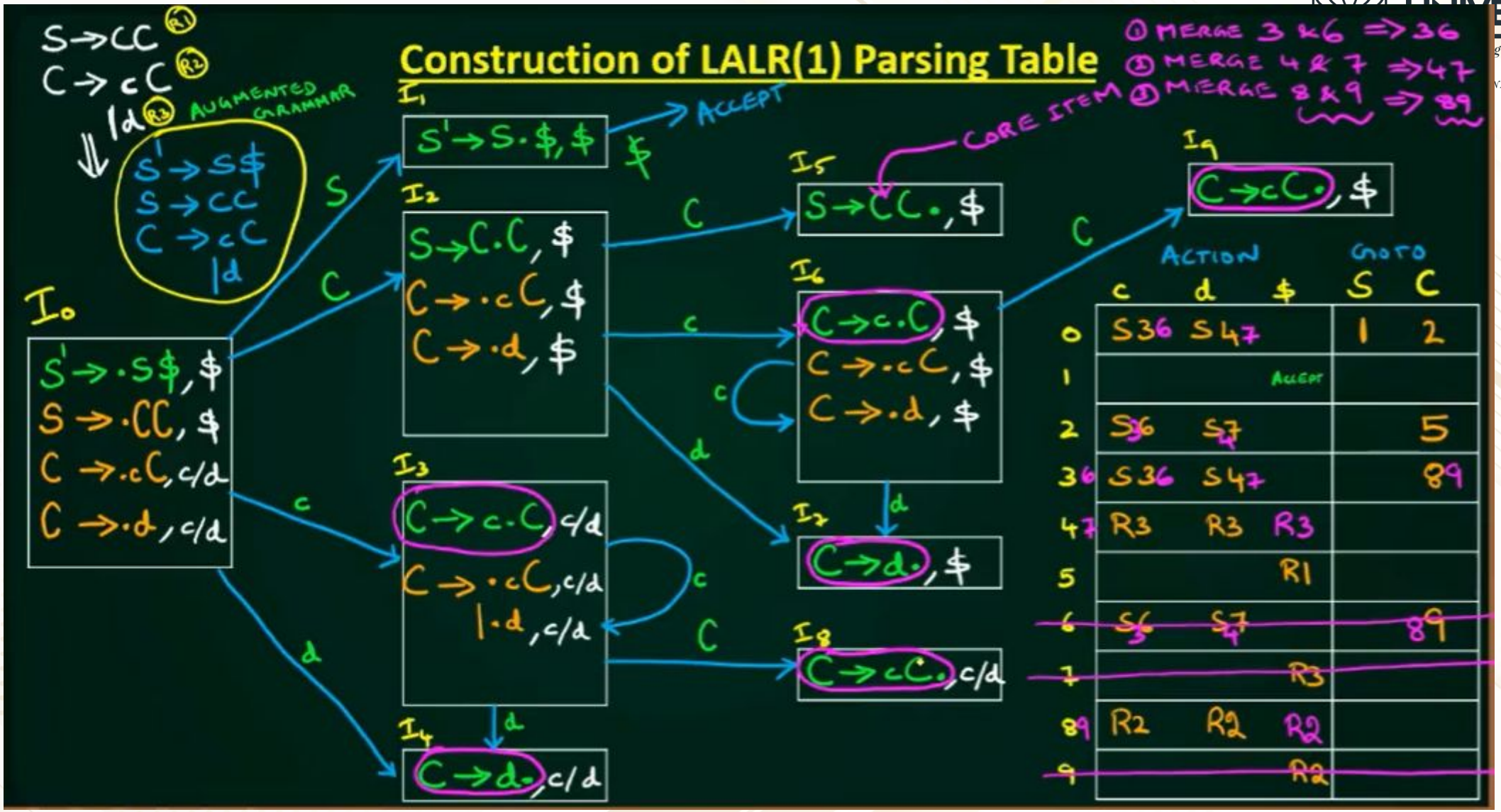

S → (L)/a
L → L,S/S
[2] Test for LL(1), LR(0) & SLR(1) Grammar
LL(1) X
LR(0)
SLR(1)
AUGMENTED GRAMMAR
S' → .S$
S → (L)/a
L → L,S/S
I0
S' → .S$
S → .(L)/.a
I1
S' → S.$
$ → ACCEPTED
I2
S → (.L)
L → .L,S
L → .S
S → .(L)
S → .a
I3
S → a.
I4
S → (L.)
L → L.,S
I5
L → S.
I6
S → (L).
I7
L → L,.S
S → .(L)
S → .a
I8
L → L,S.
S
L
(
a
)
,




S → Aa
|bAc
|dc
|bda
A → d
[4] Test for LL(1), LR(0) & SLR(1) Grammar
LL(1) X
LR(0) X
SLR(1) X
I0
S' → .S$
S → .Aa
|.bAc
|.dc
|.bda
A → .d
d
I1
S → d.c
A → d.
Terminal
c
I2
c ∩ Follow(A) ≠ ϕ
{a, c}


[5] Test for LL(1), LR(0) & SLR(1) Grammar
S → AS | b
A → SA | a
FIRST(AS) ∩ FIRST(b)
{b}
FIRST(AS)
{a,b}
I0
S' → .S$
S → .AS
1.b
A → .SA
1.a
S
I1
S' → S.$
A → S.A
A → .SA
1.a
S → .AS
1.b
$
ACCEPT
A
I5
A → SA.
S → A.S
S → .AS
1.b
A → .SA
1.a
b
I8
LL(1) X
LR(0) X
SLR(1) X
b
I5 | S8/R3
Follow(A) = {a,b}
{b}
{a}
I2
I3
I4
I6
I7

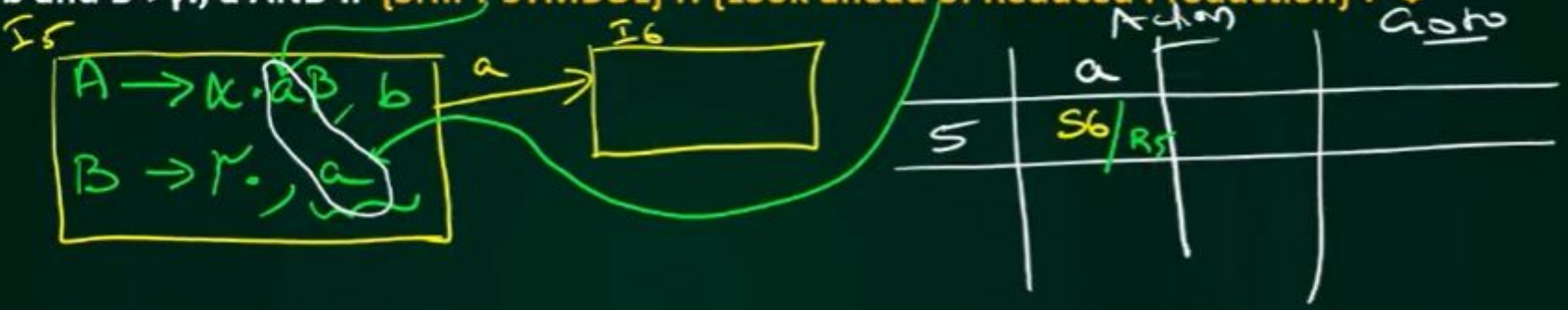

LR(1) Items and Construction of CLR(1) Parsing Table
S→CC (R1)
C→cC (R2)
|d (R3)
AUGMENTED GRAMMAR
S'→S$
S→CC
C→cC
|d
I0
S'→.S$, $
S→.CC, $
C→.cC, c/d
C→.d, c/d
I1
S'→S.$, $
ACCEPT
I2
S→C.C, $
C→.cC, $
C→.d, $
I3
C→c.C, c/d
C→.cC, c/d
|.d, c/d
I4
C→d., c/d
I5
S→CC., $
I6
C→c.C, $
C→.cC, $
C→.d, $
I7
C→d., $
I8
C→cC., c/d
I9
C→cC., $
ACTION
GOTO
c d $ S C
0 S3 S4 1 2
1 ACCEPT
2 S6 S7 5
3 S3 S4 8
4 R3 R3
5 R1
6 S6 S7 9
7 R3
8 R2 R2
9 R2


Construction of LALR(1) Parsing Table
S→CC (R1)
C→cC (R2)
|d (R3)
AUGMENTED GRAMMAR
S'→S$
S→CC
C→cC
|d
I0
S'→.S$, $
S→.CC, $
C→.cC, c/d
C→.d, c/d
I1
S'→S.$, $
ACCEPT
I2
S→C.C, $
C→.cC, $
C→.d, $
I3
C→c.C, c/d
C→.cC, c/d
|.d, c/d
I4
C→d., c/d
I5
S→CC., $
CORE ITEM
I6
C→c.C, $
C→.cC, $
C→.d, $
I7
C→d., $
I8
C→cC., c/d
I9
C→cC., $
① MERGE 3 & 6 ⇒ 36
② MERGE 4 & 7 ⇒ 47
③ MERGE 8 & 9 ⇒ 89
ACTION
GOTO
c d $ S C
0 S36 S47 1 2
1 ACCEPT
2 S36 S47 5
36 S36 S47 89
47 R3 R3 R3
5 R1
6 S36 S47 89
7 R3
89 R2 R2 R2
9 R2

Parser Conflicts in CLR(1) Parser

SHIFT-REDUCE CONFLICT OCCURS IF THE LR(1) ITEM CONTAINS PRODUCTIONS OF THE FORM

A->α.aβ, b and B->γ., a AND IF {SHIFT SYMBOL} ∩ {Look ahead of Reduced Production} ≠ Φ



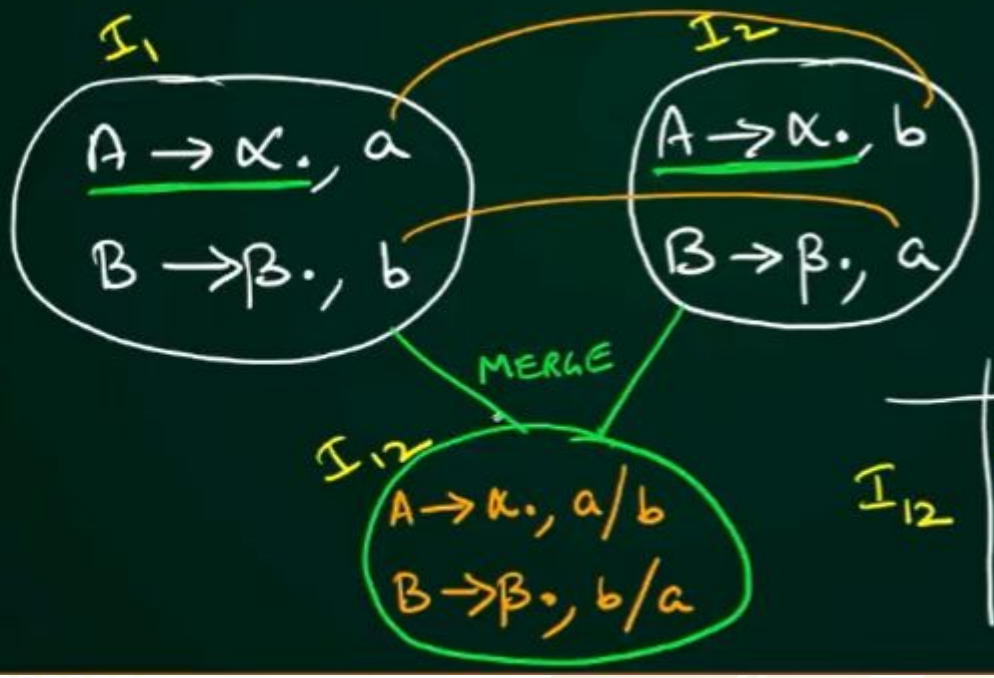
REDUCE-REDUCE CONFLICT OCCURS IF LR(1) ITEM CONTAINS PRODUCTIONS OF THE FORM

A->α., a and B->β., a AND IF {Look ahead of Reduced Prod. 1} ∩ {Look ahead of Reduced Prod. 2} ≠ Φ

Parser Conflicts in LALR(1) Parser

SHIFT-REDUCE CONFLICT DOES NOT OCCUR IN LALR(1) PARSER SINCE IT IS IN CLR(1)

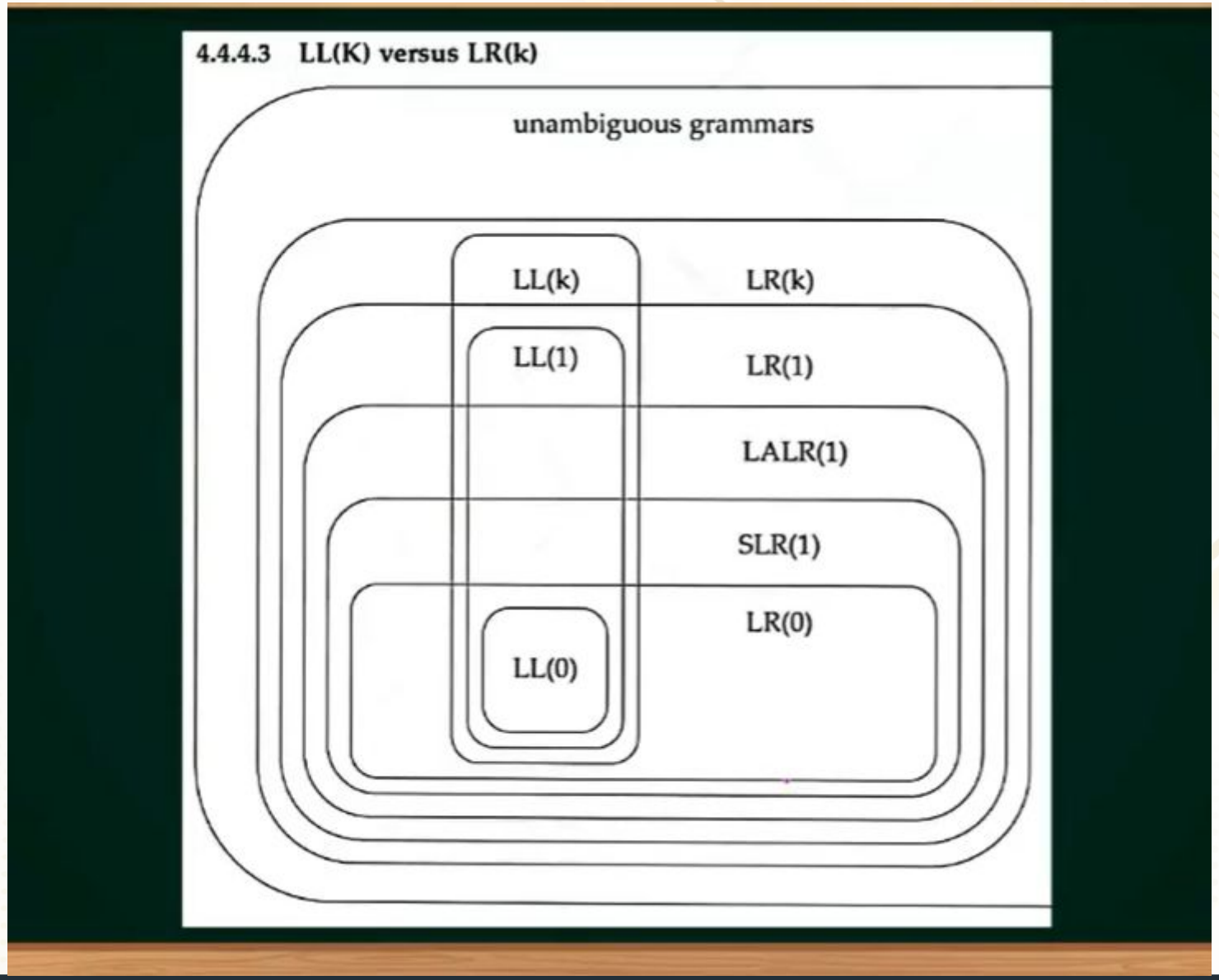
REDUCE-REDUCE CONFLICT OCCURS IF LR(1) ITEM CONTAINS PRODUCTIONS OF THE FORM
A->α., a and B->β., b AND AFTER MERGING IDENTICAL CORE ITEMS
{Look ahead of Reduced Prod. 1} ∩ {Look ahead of Reduced Prod. 2} ≠ Φ

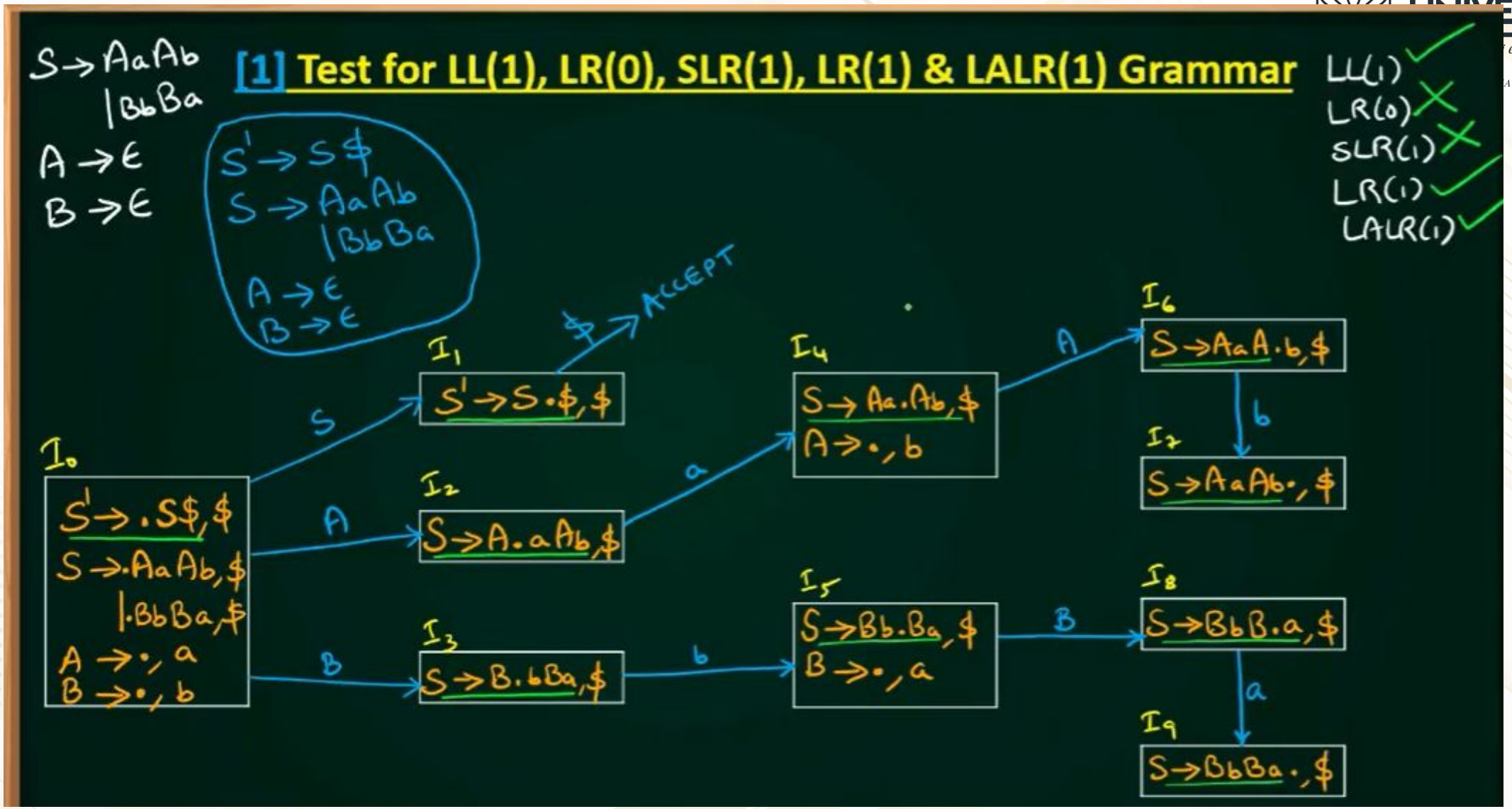


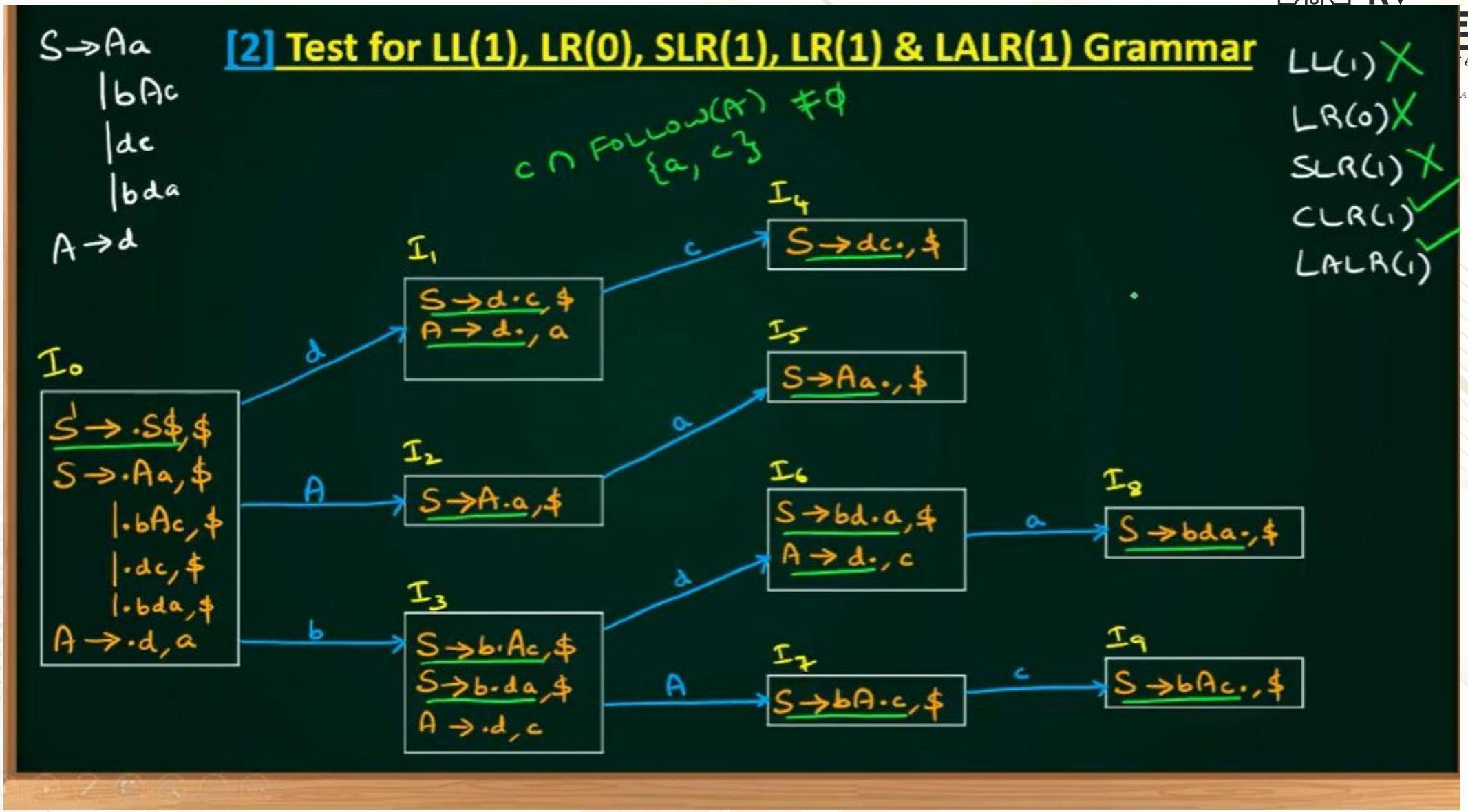
ACTION GOTO

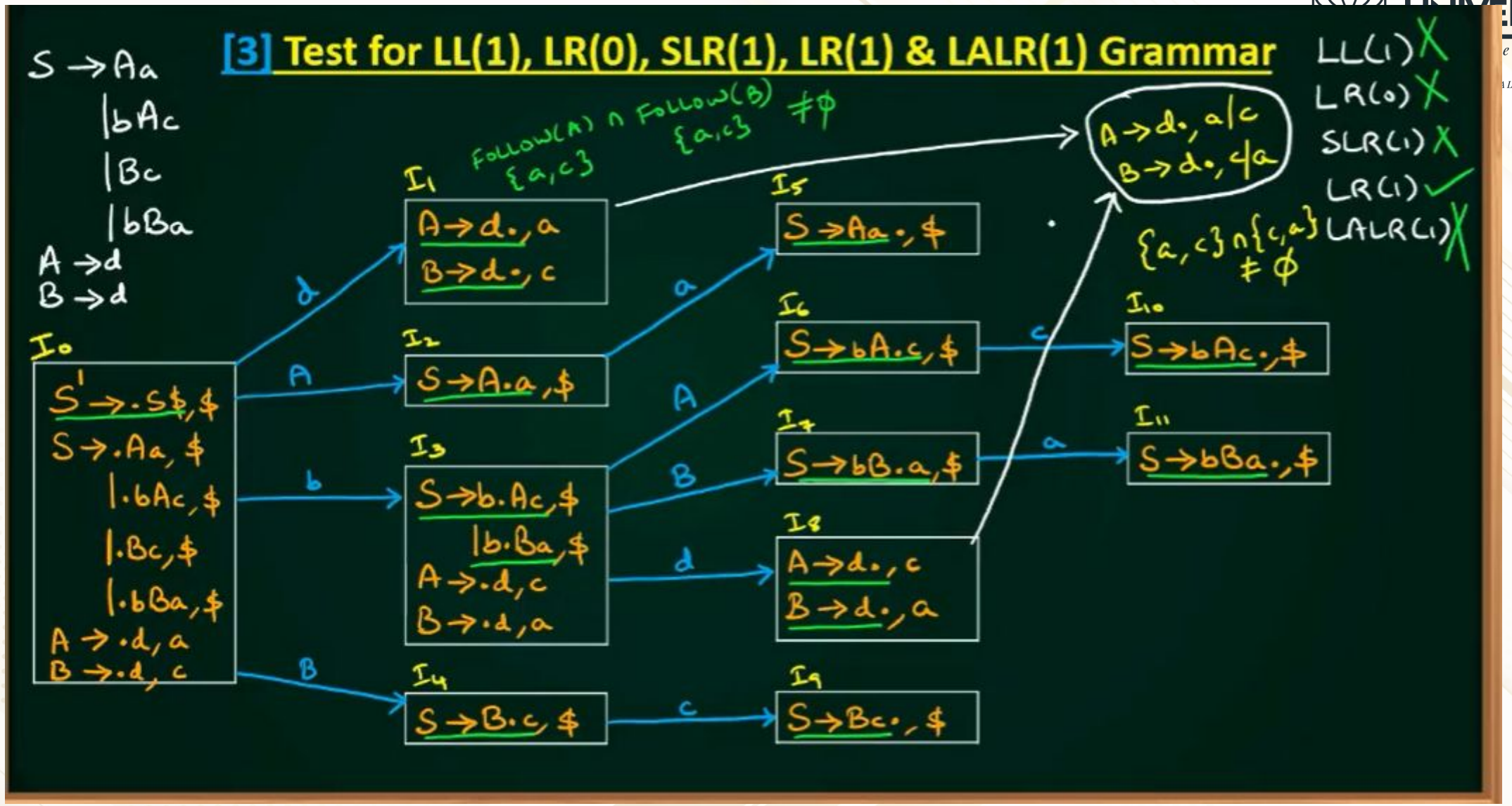
	a	b
I_{12}	R1/R2	R1/R2

4.4.4.3 LL(K) versus LR(k)




S → AaAb
|BbBa
A → ϵ
B → ϵ
[1] Test for LL(1), LR(0), SLR(1), LR(1) & LALR(1) Grammar
LL(1) ✓
LR(0) ✗
SLR(1) ✗
LR(1) ✓
LALR(1) ✓
S' → S $
S → AaAb
|BbBa
A → ϵ
B → ϵ
I0
S' → .S$, $
S → .AaAb, $
|.BbBa, $
A → •, a
B → •, b
S
A
B
I1
S' → S.$, $
$ → ACCEPT
I2
S → A.aAb, $
a
I3
S → B.bBa, $
b
I4
S → Aa.Ab, $
A → •, b
A
I5
S → Bb.Ba, $
B → •, a
B
I6
S → AaA.b, $
b
I7
S → AaAb•, $
I8
S → BbB.a, $
a
I9
S → BbBa•, $


S→Aa
|bAc
|dc
|bda
A→d
[2] Test for LL(1), LR(0), SLR(1), LR(1) & LALR(1) Grammar
LL(1) X
LR(0) X
SLR(1) X
CLR(1) ✓
LALR(1) ✓
c ∩ FOLLOW(A) ≠ φ
{a, c}
I0
S'→.S$, $
S→.Aa, $
|.bAc, $
|.dc, $
|.bda, $
A→.d, a
I1
S→d.c, $
A→d., a
I2
S→A.a, $
I3
S→b.Ac, $
S→b.da, $
A→.d, c
I4
S→dc., $
I5
S→Aa., $
I6
S→bd.a, $
A→d., c
I7
S→bA.c, $
I8
S→bda., $
I9
S→bAc., $
d
A
b
c
a
d
A
a
c


[3] Test for LL(1), LR(0), SLR(1), LR(1) & LALR(1) Grammar
S → Aa | bAc | Bc | bBa
A → d
B → d
LL(1) X
LR(0) X
SLR(1) X
LR(1) ✓
LALR(1) X
FOLLOW(A) ∩ FOLLOW(B) {a,c} ≠ ϕ
{a,c} ∩ {c,a} ≠ ϕ
I0
S'→.S$, $
S→.Aa, $
|.bAc, $
|.Bc, $
|.bBa, $
A→.d, a
B→.d, c
I1
A→d., a
B→d., c
I2
S→A.a, $
I3
S→b.Ac, $
|b.Ba, $
A→.d, c
B→.d, a
I4
S→B.c, $
I5
S→Aa., $
I6
S→bA.c, $
I7
S→bB.a, $
I8
A→d., c
B→d., a
I9
S→Bc., $
I10
S→bAc., $
I11
S→bBa., $
A→d., a|c
B→d., c|a
d
A
b
B
a
A
B
d
c
c
a

Operator Precedence Parser

OPERATOR PRECEDENCE GRAMMAR CAN PARSE SOME OF THE AMBIGUOUS GRAMMARS AND ARE MAINLY USED FOR MATHEMATICAL OPERATORS.

OPERATOR GRAMMAR

EG. E+E | E*E | id

IN AN OPERATOR GRAMMAR TWO VARIABLES(NON TERMINALS) CANNOT BE ADJACENT AND IT SHOULD NOT HAVE ϵ PRODUCTIONS.

E -> E+E | E*E | id

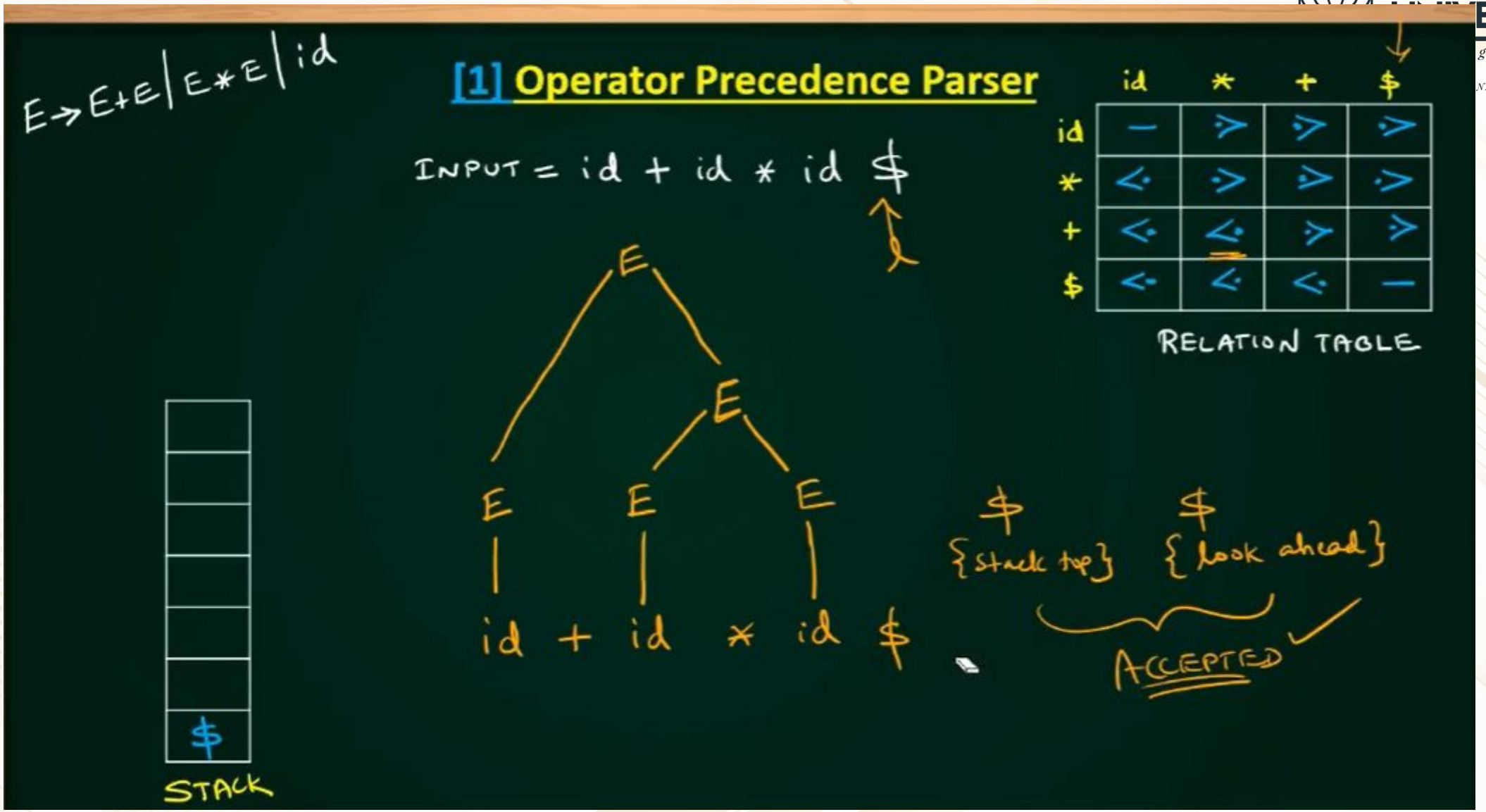
OPERATOR RELATION TABLE

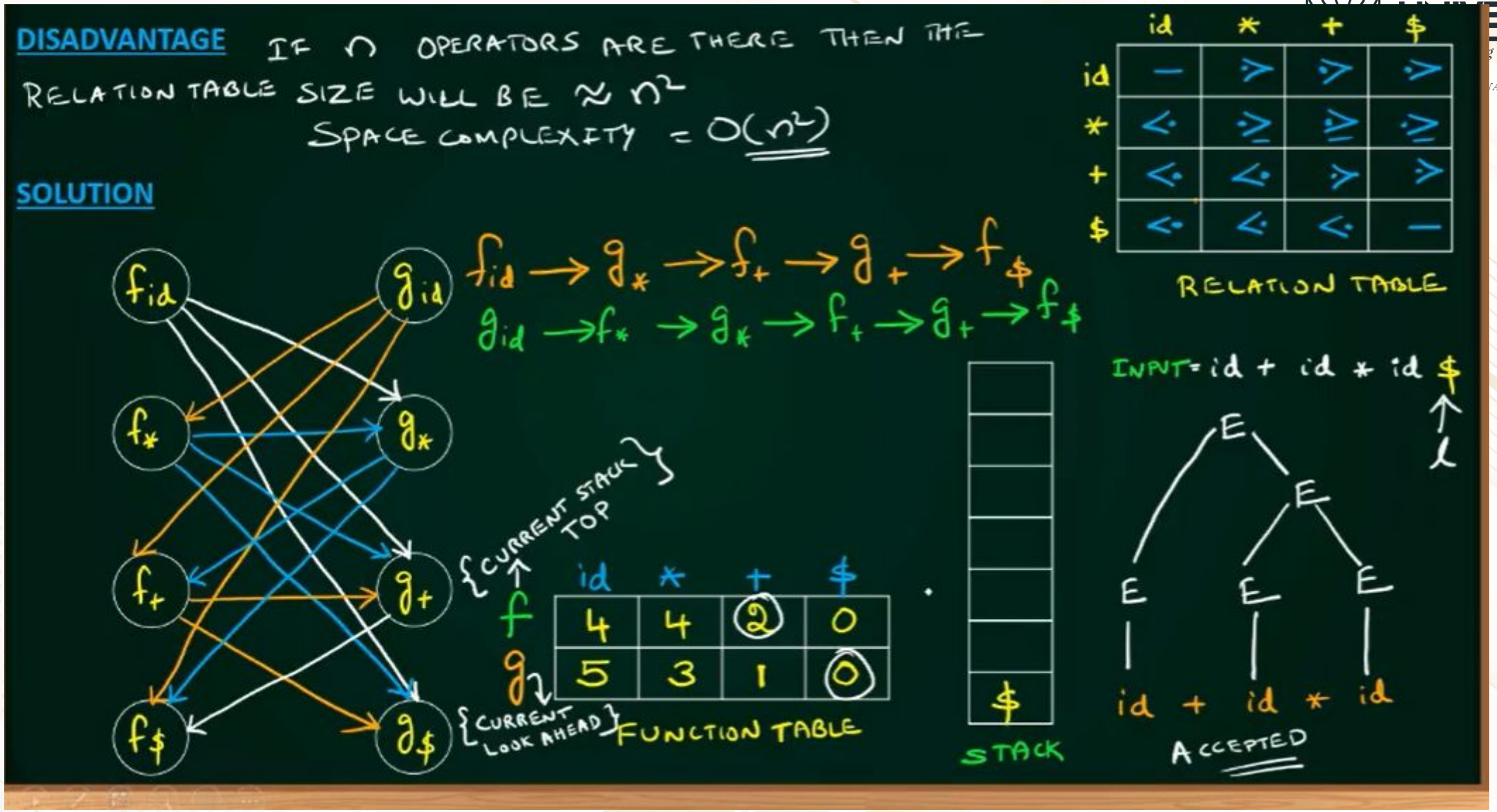
E-> E+E | E*E | id

IN THE OPERATOR RELATION TABLE id WILL BE GIVEN HIGHEST PRECEDENCE AND $ WILL BE GIVEN LEAST PRECEDENCE COMPARED TO ANY OTHER OPERATORS.

OPERATOR RELATION TABLE

	id	*	+	$
id	-	>	>	>
*	<	>	>	>
+	<	<	>	>
$	<	<	<	-


E→E+E|E*E|id
[1] Operator Precedence Parser
INPUT = id + id * id $
id * + $
id
*
+
$
RELATION TABLE
E
E
E
E
id + id * id $
$
{stack top}
$
{look ahead}
ACCEPTED
$
STACK


DISADVANTAGE
IF n OPERATORS ARE THERE THEN THE RELATION TABLE SIZE WILL BE ≈ n²
SPACE COMPLEXITY = O(n²)
SOLUTION
f_id → g_* → f_+ → g_+ → f_$
g_id → f_* → g_* → f_+ → g_+ → f_$
RELATION TABLE
INPUT = id + id * id $
CURRENT STACK TOP
CURRENT LOOK AHEAD
FUNCTION TABLE
STACK
ACCEPTED