Playlist Link:
https://www.youtube.com/playlist?list=PL1QH9gyQXfguPNDTsnG90W2kBDQpYLDQr

Transition Diagrams:

Relational Ops:

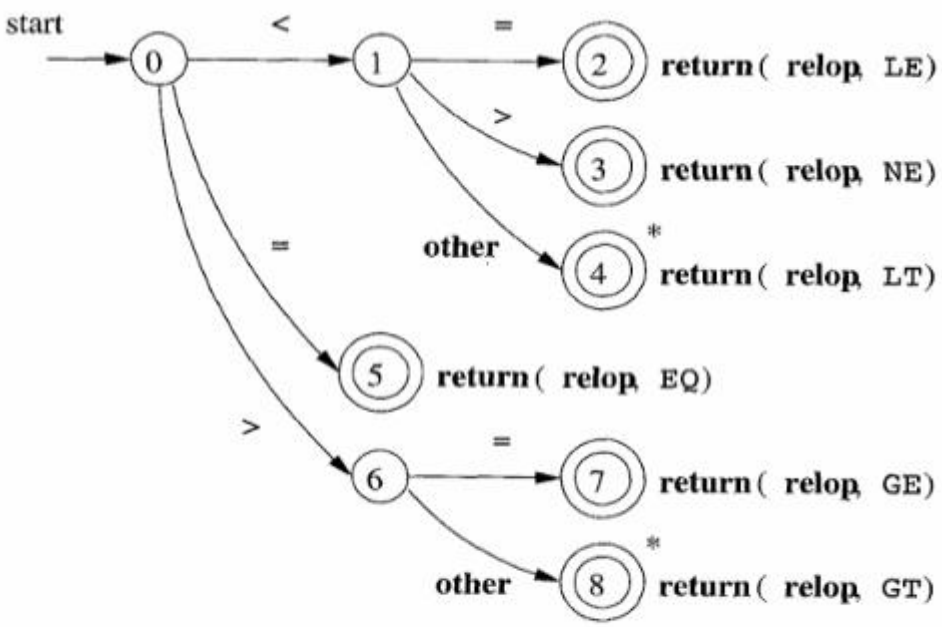
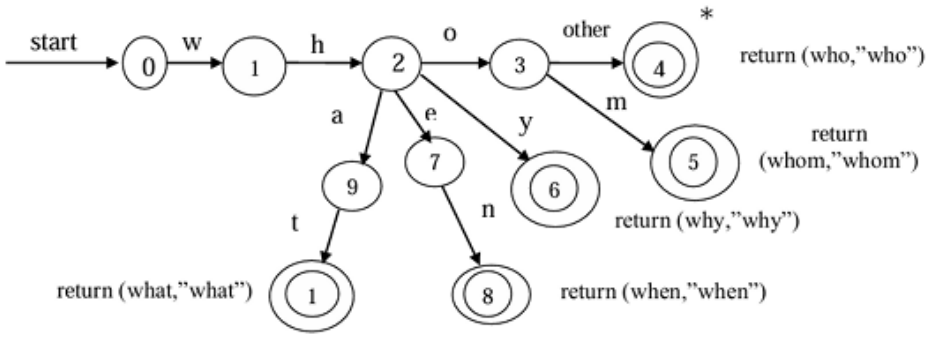
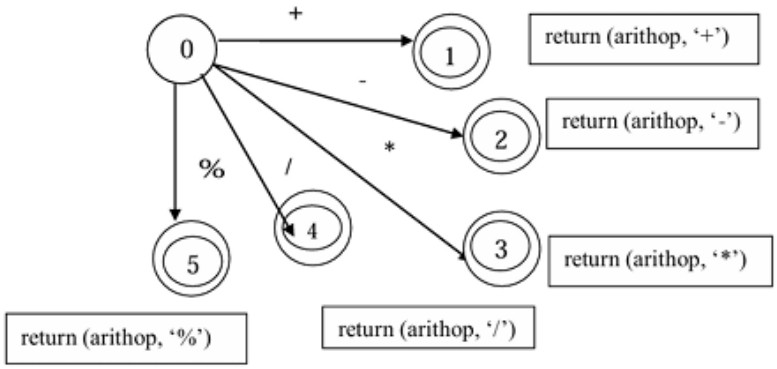


Figure 3.13: Transition diagram for **relop**

Arithmetic Ops:

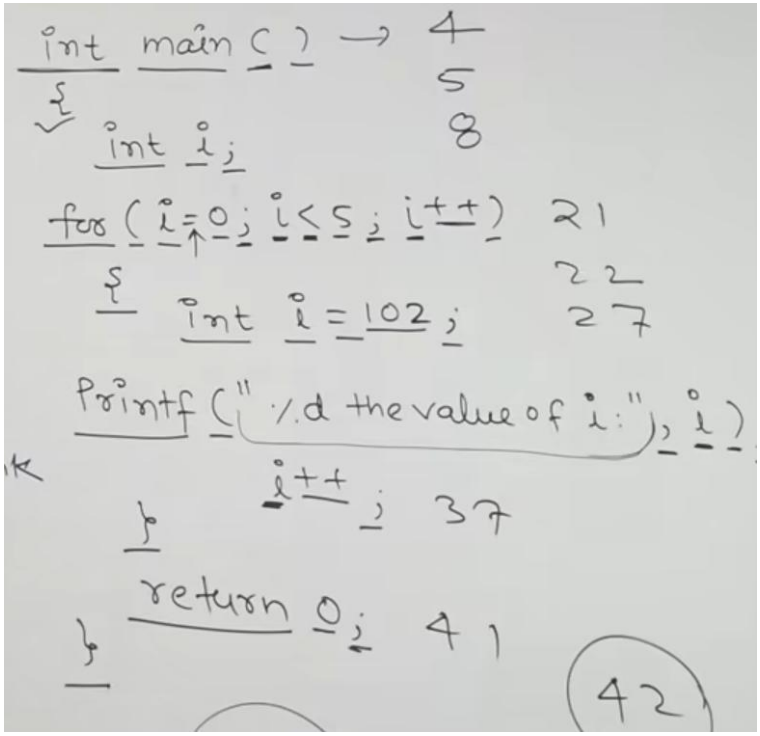


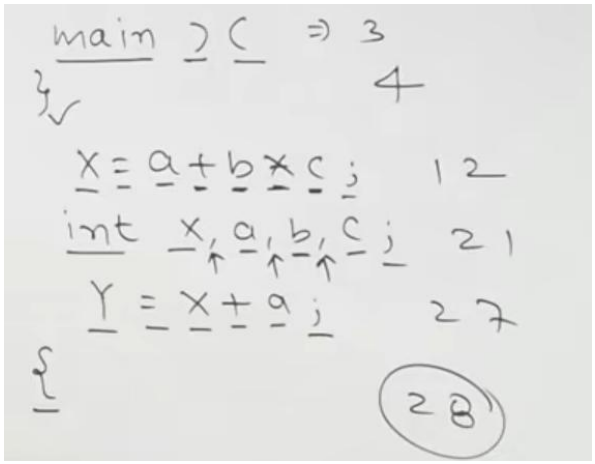
Arithmetic operators

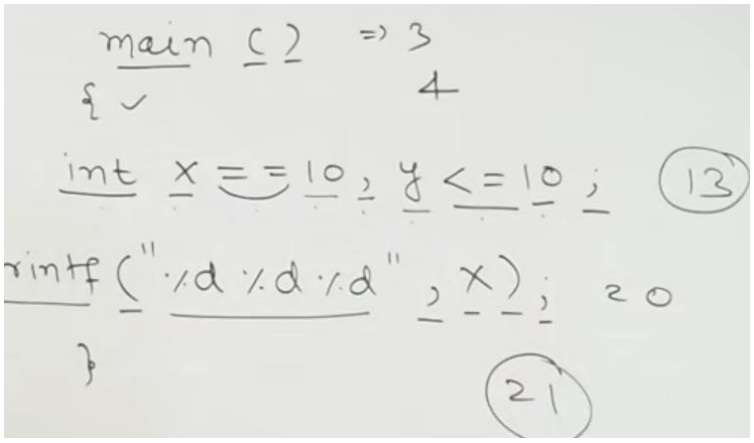


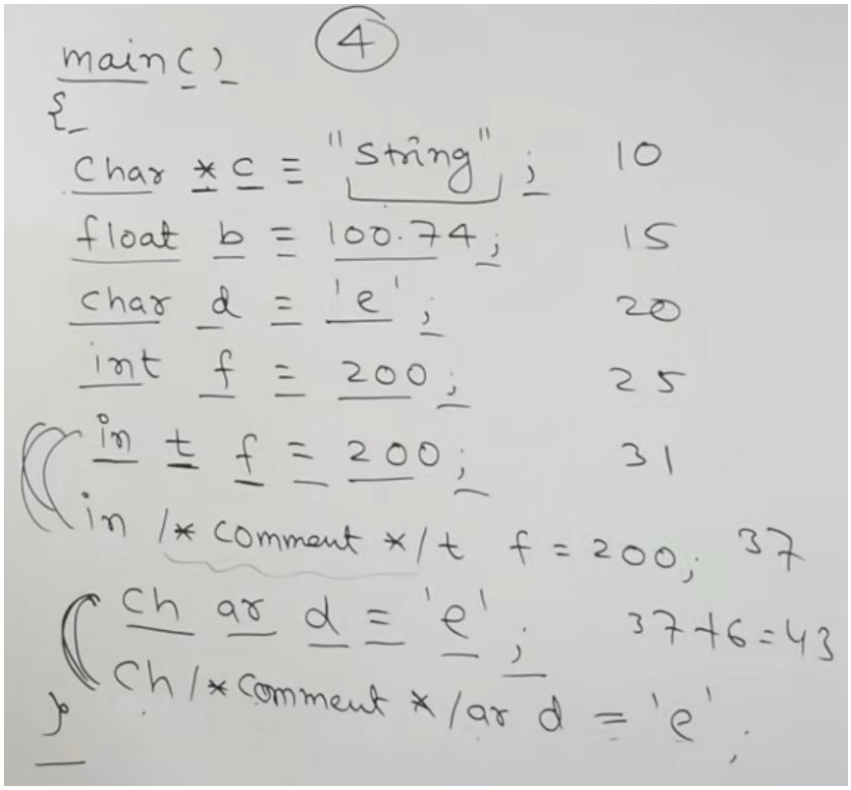
Count Tokens:

Ex 1:

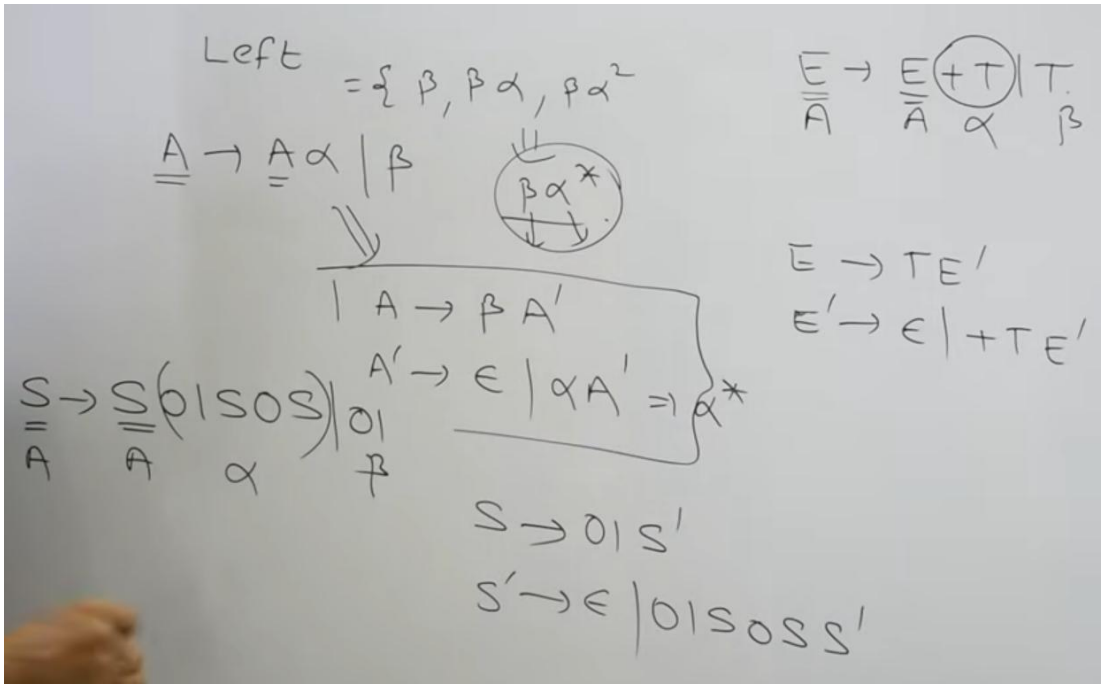




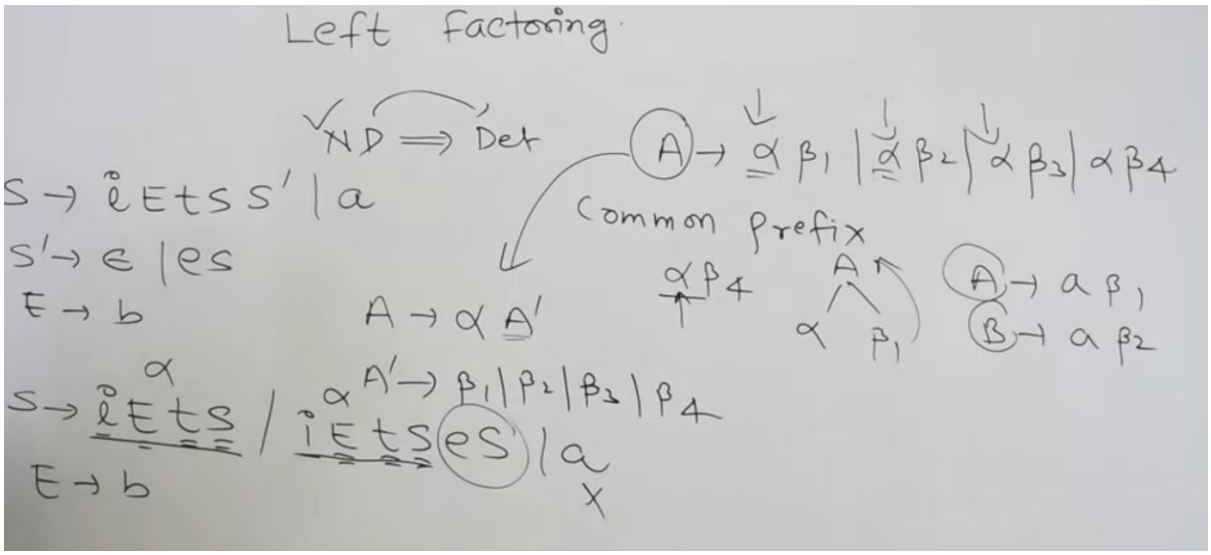


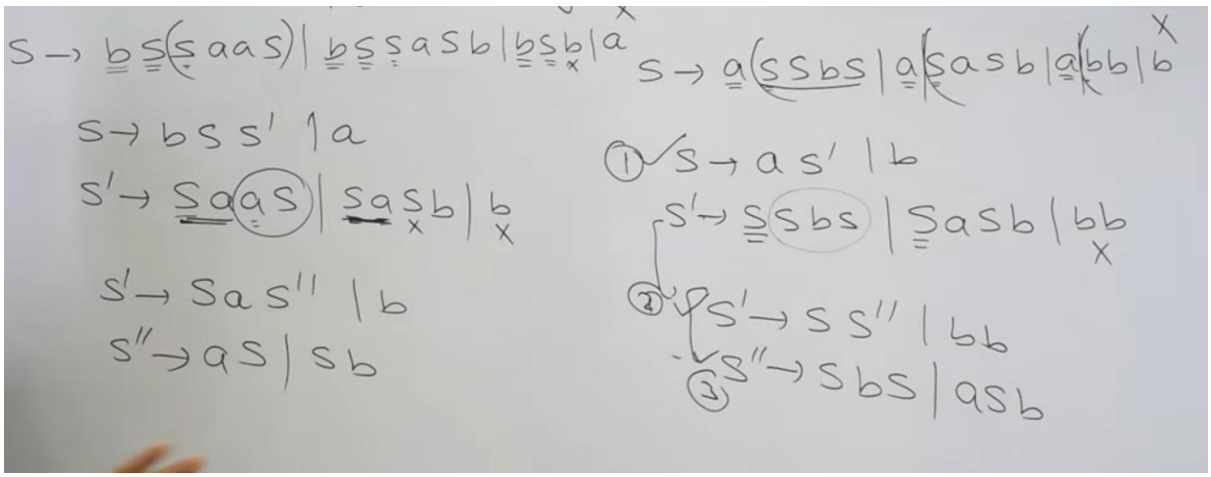


Eliminating Left Recursion:



Eliminating Left Factoring:





$S \to a \mid ab \mid abc \mid abcd$

$S \to aS'$

$S' \to \epsilon \mid b \mid bc \mid bcd$

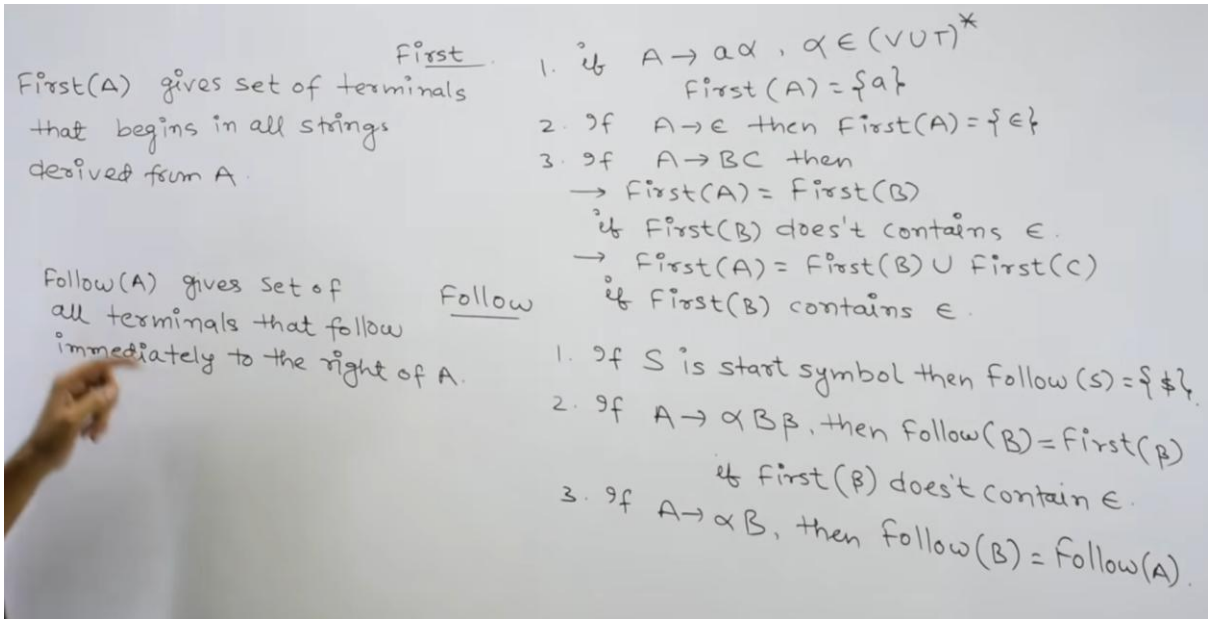
$S' \to \epsilon \mid bS''$

$S'' \to \epsilon \mid c \mid cd$

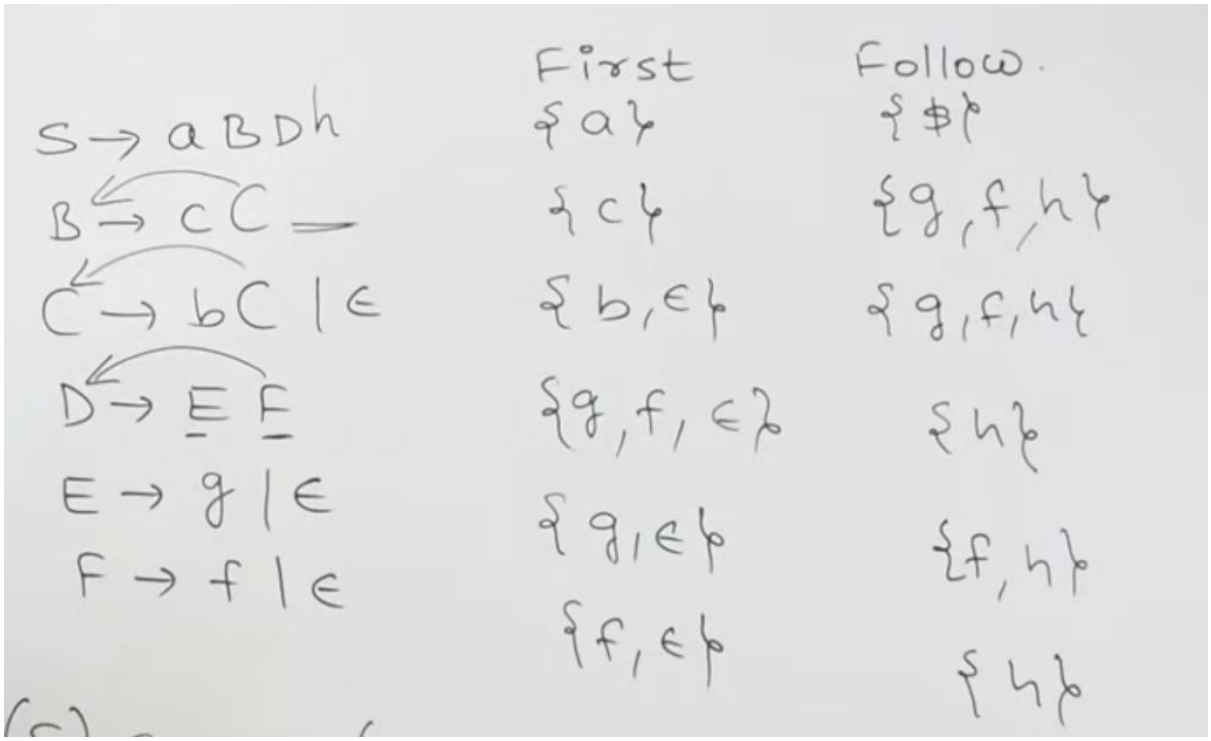
$S'' \to \epsilon \mid cS'''$

$S''' \to \epsilon \mid d$

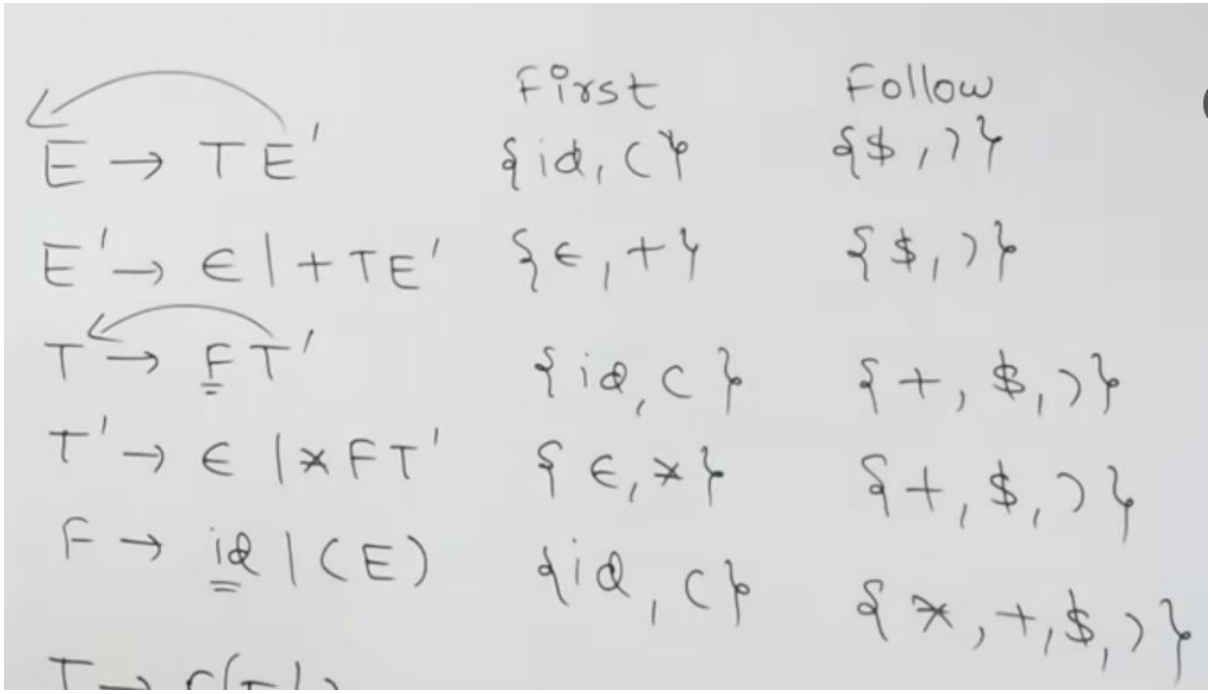
First and Follow Rules:



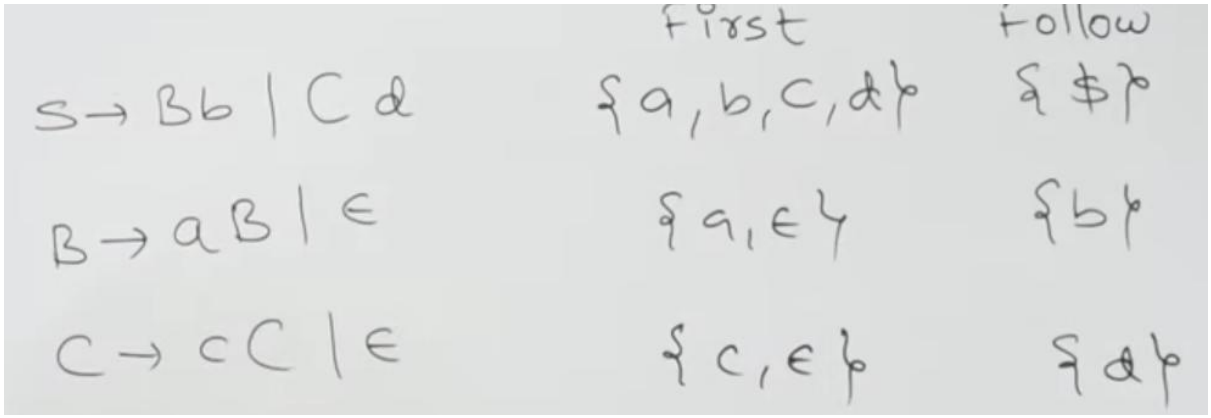
Ex 1:



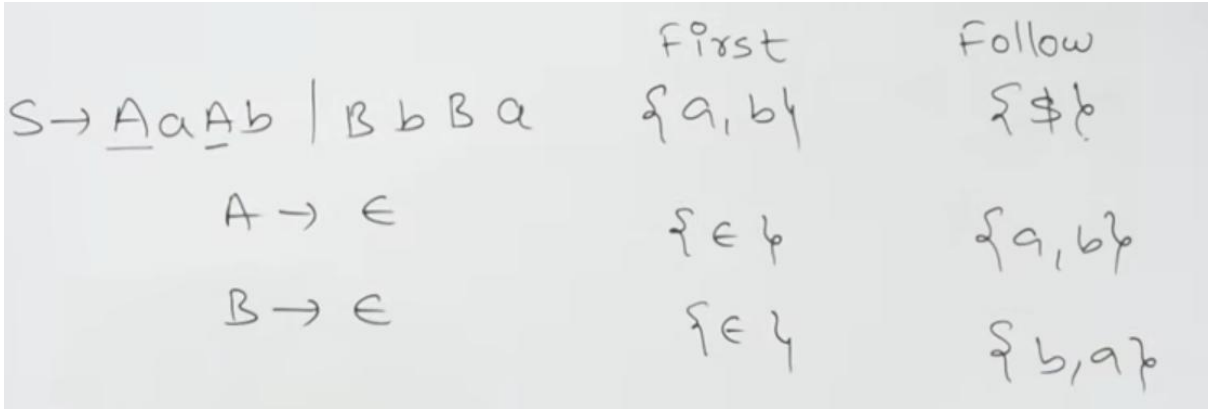
Ex 2:


First
Follow
E → TE'
{id, (}
{$,)}
E' → ∈ | +TE'
{∈, +}
{$,)}
T → FT'
{id, (}
{+, $,)}
T' → ∈ | *FT'
{∈, *}
{+, $,)}
F → id | (E)
{id, (}
{*, +, $,)}

Ex 3:


First
Follow
S → Bb | Cd
{a, b, c, d}
{$}
B → aB | ∈
{a, ∈}
{b}
C → cC | ∈
{c, ∈}
{d}

Ex 4:


First
Follow
S → AaAb | BbBa
{a, b}
{$}
A → ∈
{∈}
{a, b}
B → ∈
{∈}
{b, a}

LL(1):

STEPS:

1. Find first and follow of all

2. Construct a Matrix with Variables on Left and Terminals on Top

3. For each individual broken production A->a, check if the first(a) contains null, if no then fill the production under M[A, first(a)]

4. If yes, fill it in M[A, follow(A)]

5.Now for the stack part, enter the $S (final state).

6. Compare against input string's pointer alphabet, check the production from matrix M and replace that top of the sstack state with the reversed order of the production (because its a Stack). Increment the I/S Ptr.

7. If the top of the stack is a terminal and matyches the input string's pointer, POP.

8. Repeat untilll $ is left.

LR(0):

1. Create a new augmented grammar S' -> S

2. Find Closure of the new AG.

3. Closure is basically adding a . Ex: S'-> .S, and since the . has a State again, add the prodcutions of the state next to . and keep repeating until u can no longer do it.

4. Apply Goto(I0, all possible), for each and construct the DFA state diagram. GOTO is moving the . one step further and finding closure on that new shit.

5. After State diagram, construct a Matrix M with the numbers of state on the left and the Headers Action (containing terminals) and Goto (containing Variables (state: S, T, A, B, etc))

6. Now using the state Diagram, for each LR Item 01,2,3, etc. fill the action side with Shifts if I0->I3 is via a, then under M[0,a] fill S3.

7. Fill the Goto side with the state number directly, ex: If I0->I2 is via A, then under M[0,A], fill 2

8. If its the final Item, then fill all the actions, with the no. of production (u can label each individual production as 1,2,3,etc) as R2 etc

Ex: if the state I6 is final item, and it contains A->ab., and it is 3rd production, then fill all actions with R3.

9. Now for the stack shit, fill the stack with 0 .

10. The I/Str ptr is at first position, check the state over the M[0,ptr] if its a Shift, then push the PTR alphabet first and then the Shift Nukmber in matrix. Increment PTR and continue.

11. If its a reduce operation, then u should find the lenghth of the No of Reduce production's and POP those no of elements. Add the Left hand (State) of the production to the stack. And now find the transition of the added state with the Number below it.

12. Repeat until u find Accept.

SLR(1):

1. Same as LR0 until the state diagram is being constructed.

2. While adding the Reduce in Matrix, make sure it gets added only in the positions where the Follow(the reduced production's left State) is present. Not the entire action row for that state.

3. Stack shit same as LR0.

CALR(1):

1. A new LR(0) iTem, it will have the Augmented Grammar + a Look ahead symbol. Ex: S'-> . S, $ Here $ is look ahead symbol, comma ',' is a separator.

2. The look ahead symbol while calculating closure is done as follows.... The first(next elements following the newly added Closure States), will be the new look ahead for that transition.

Ex: S'-> .S, $

Next the follow of the elements to the next of the . and the state i.e First($)

S->.AA, $

now next we need to add A's production (assume A->a | b), here since we have A after .A, we need to calculate first(A$), which is a | b.

A->.ab, a | b.

If there is nothing then add $

3. now continue the same, like that for LR0, but when adding the reduce in actions for final elements, add it to the elements in the LOOK AHEAD of the Final state only.

4. Same stack shit.

LALR(1):

Steps:

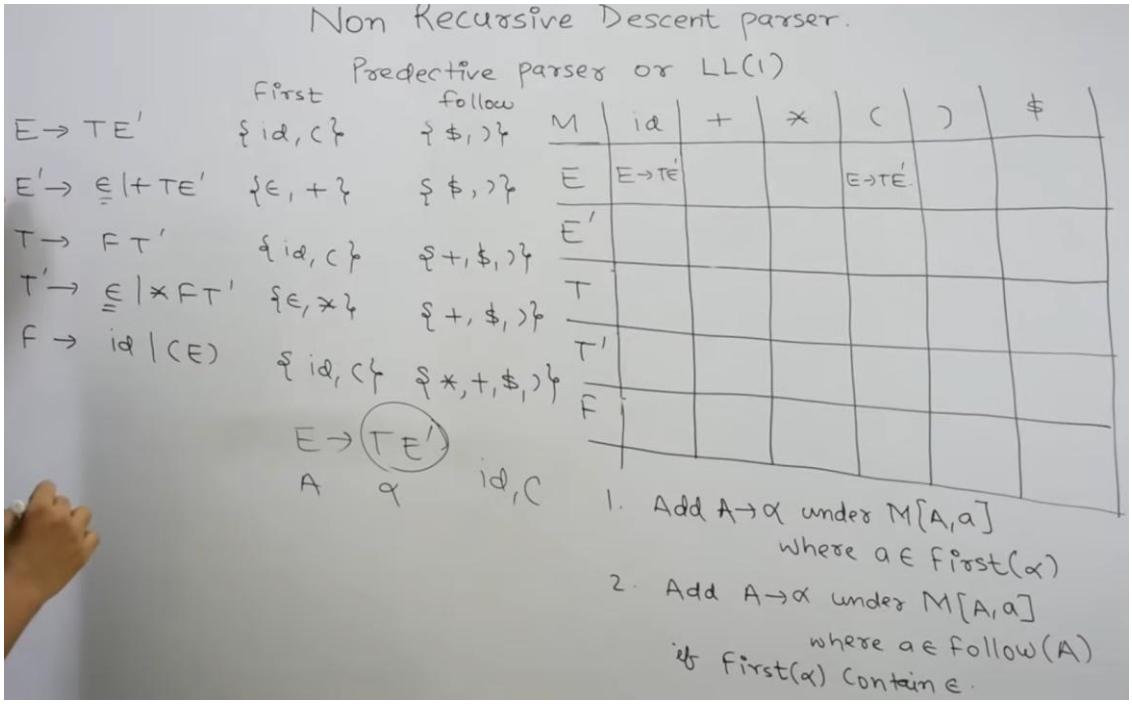
1. Same as CALR 1
2. Merge the states which have the same LR(0) items (ignore Look ahead symbols).
3. Basically if 4^{th} and 6^{th} have same LR0 items, then replace them by a new $4,6^{th}$ state, write r4 r6 as r46 and so on.

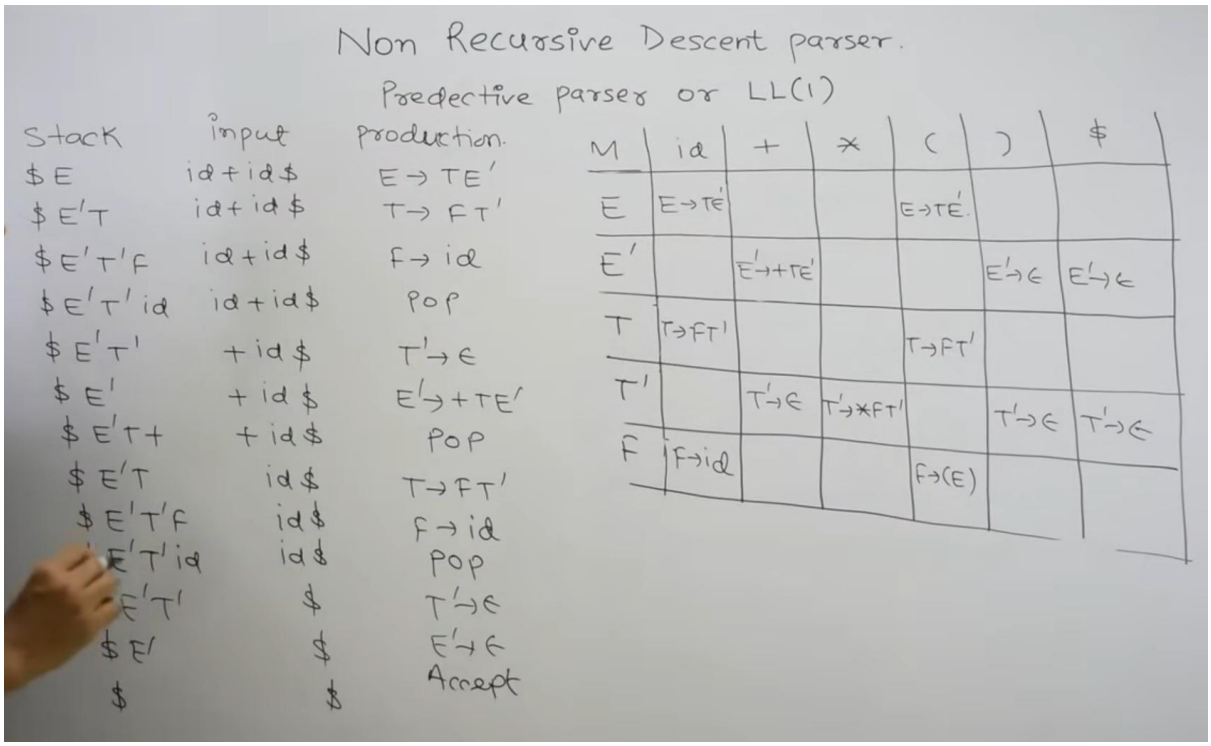
Parsers

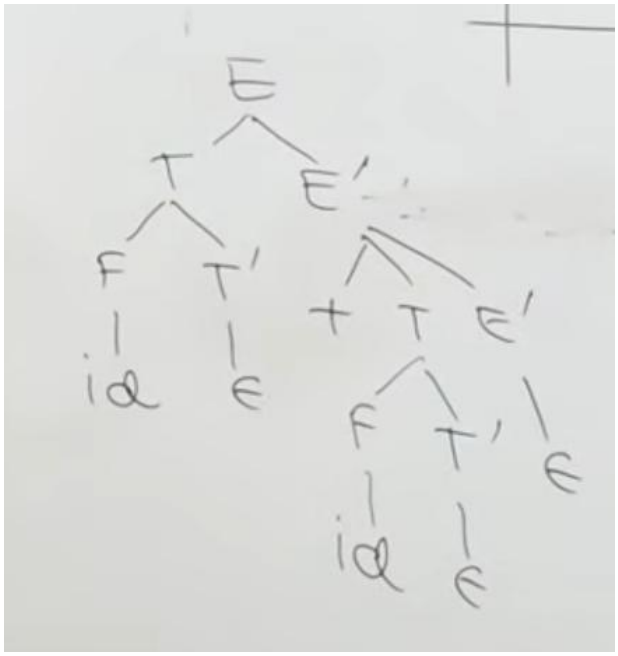
Non Recursive Descent Parser (Top-Down Parser):

LL(1) (Left-right Leftmost 1 by 1 character read):

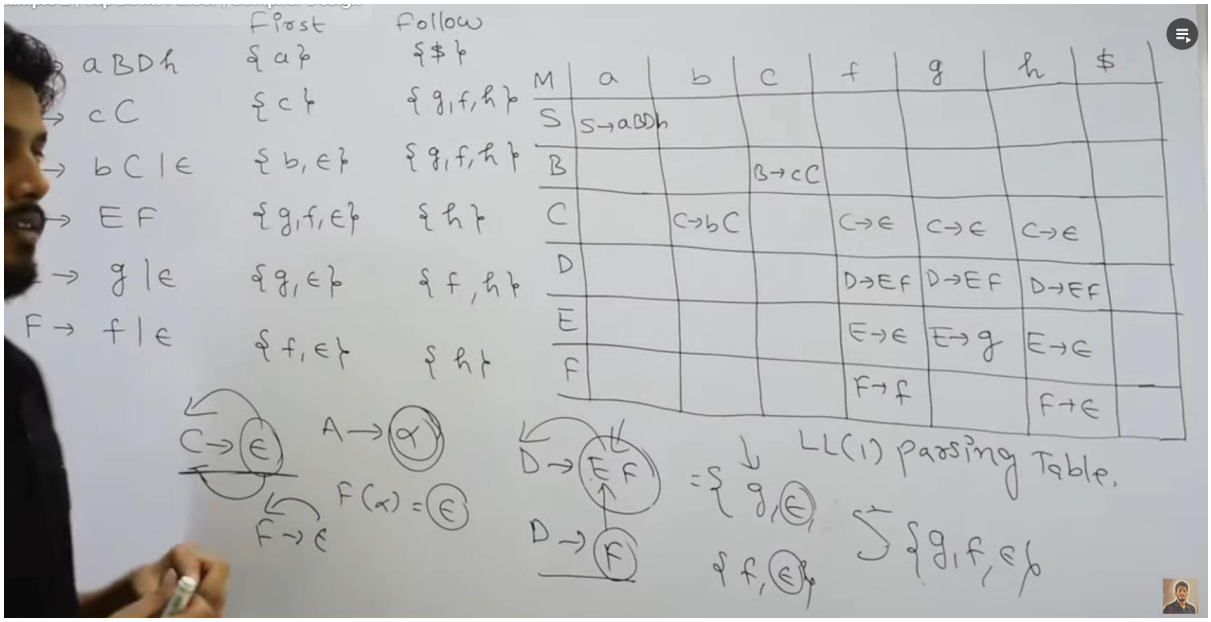
Ex 1:

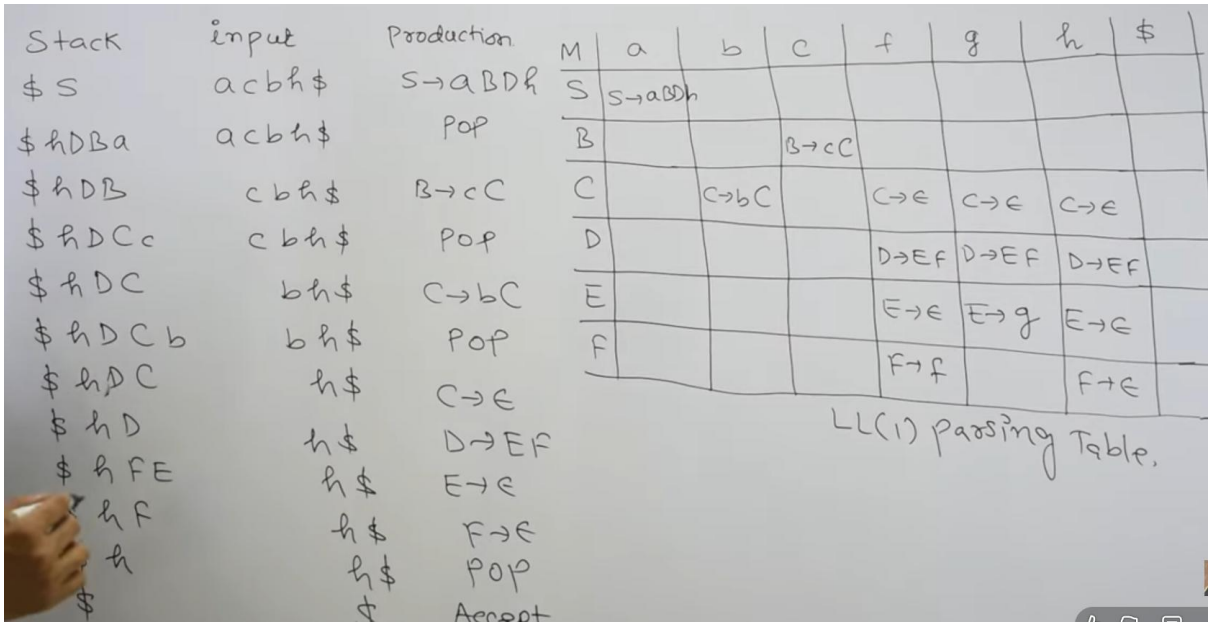





E
T
E'
F
T'
+
T
E'
id
ϵ
F
T'
ϵ
id
ϵ

Ex 2:


First
Follow
aBDh {a} {$}
cC {c} {g,f,h}
bC|ϵ {b,ϵ} {g,f,h}
EF {g,f,ϵ} {h}
g|ϵ {g,ϵ} {f,h}
f|ϵ {f,ϵ} {h}
M a b c f g h $
S S→aBDh
B B→cC
C C→bC C→ϵ C→ϵ C→ϵ
D D→EF D→EF D→EF
E E→ϵ E→g E→ϵ
F F→f F→ϵ
LL(1) parsing Table.
C→ϵ
F→ϵ
A→α
F(α)=ϵ
D→EF
D→F
={g,ϵ}
{f,ϵ}
{g,f,ϵ}

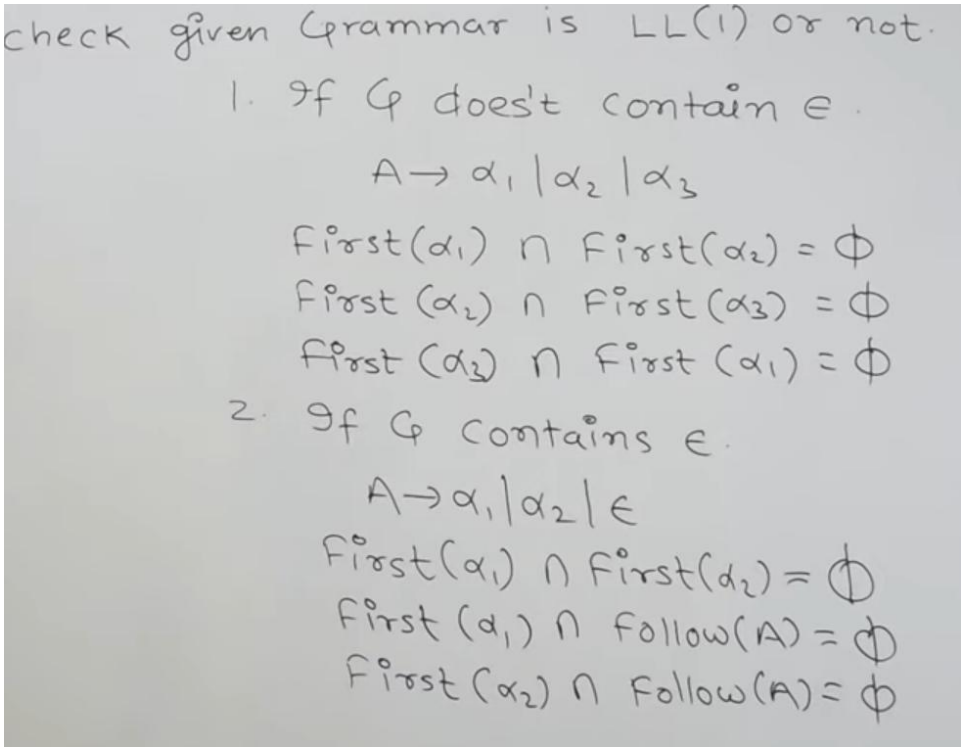


Fast method: Check whether Grammar is LL(1)

LL(1) Grammar: Each cell M[A, a] has **0 or 1** production.

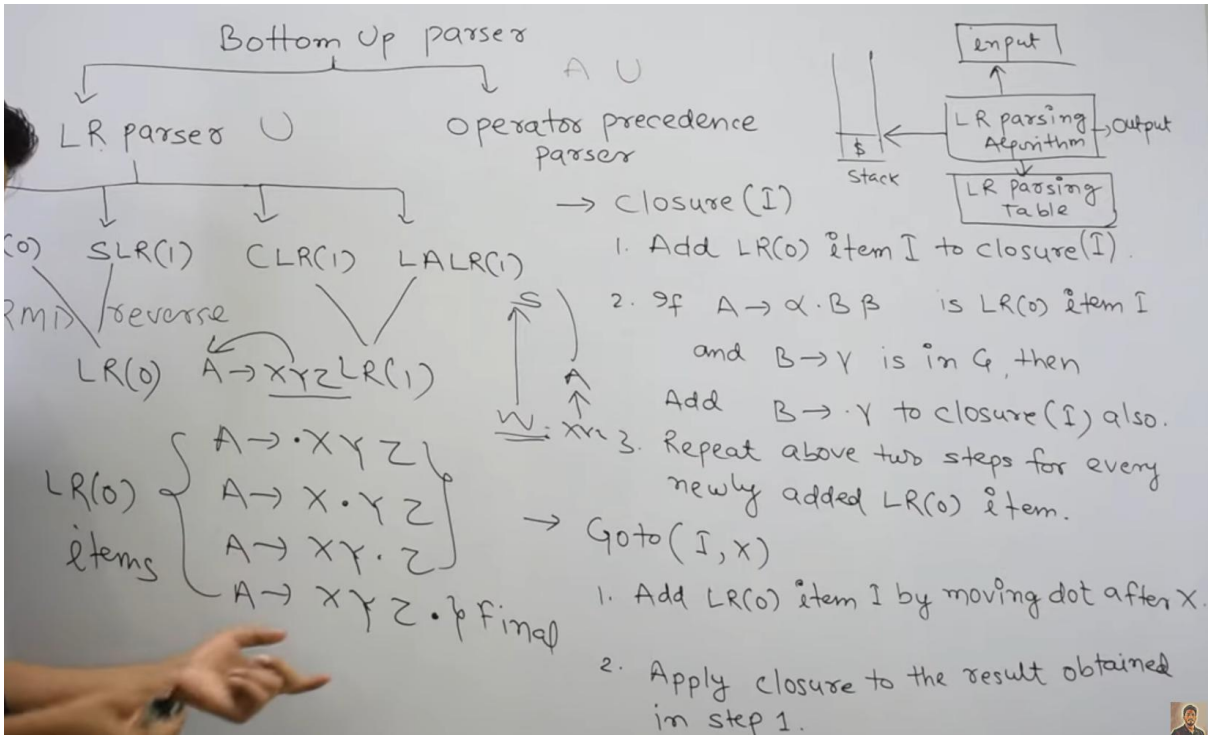
Non-LL(1) Grammar: At least one cell M[A, a] has **2 or more** productions. This is called a **conflict.**

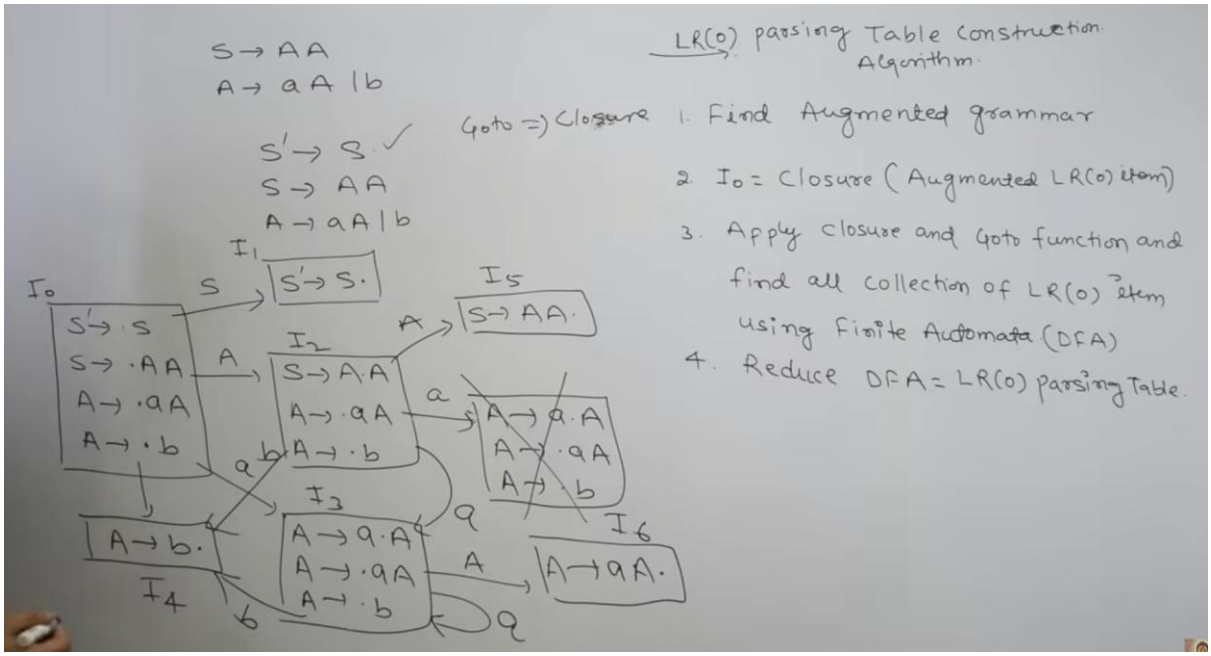
Slow method (faster if only asked if Grammar is LL(1) or not)

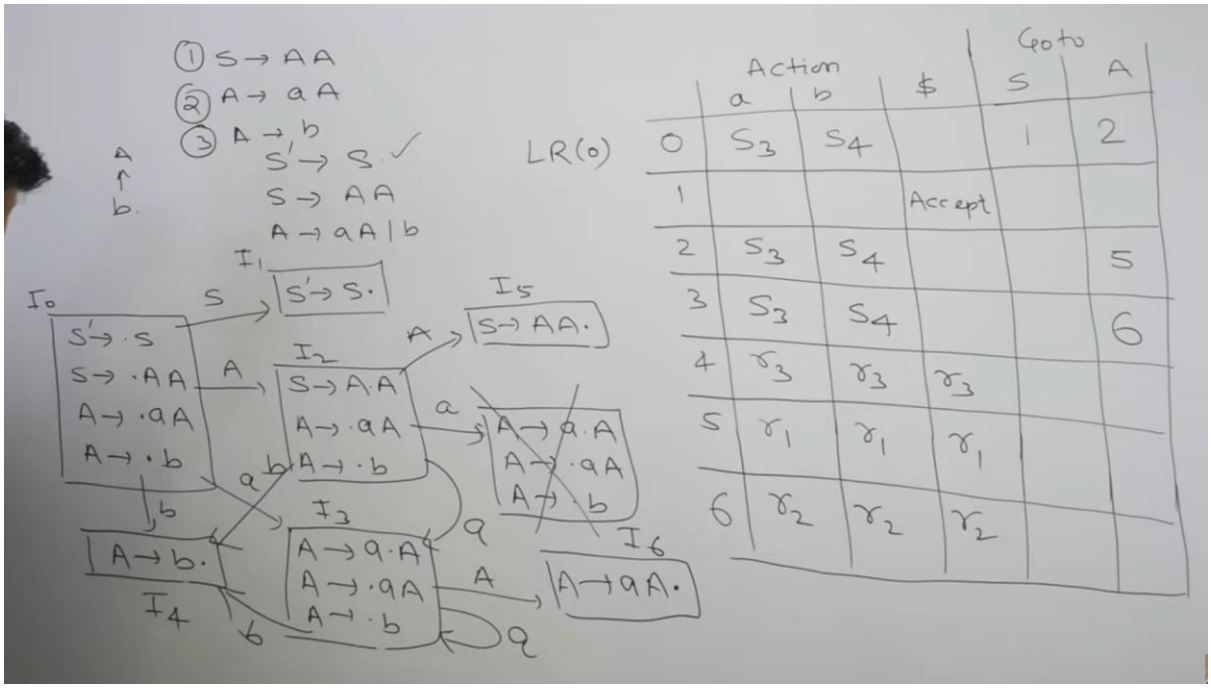


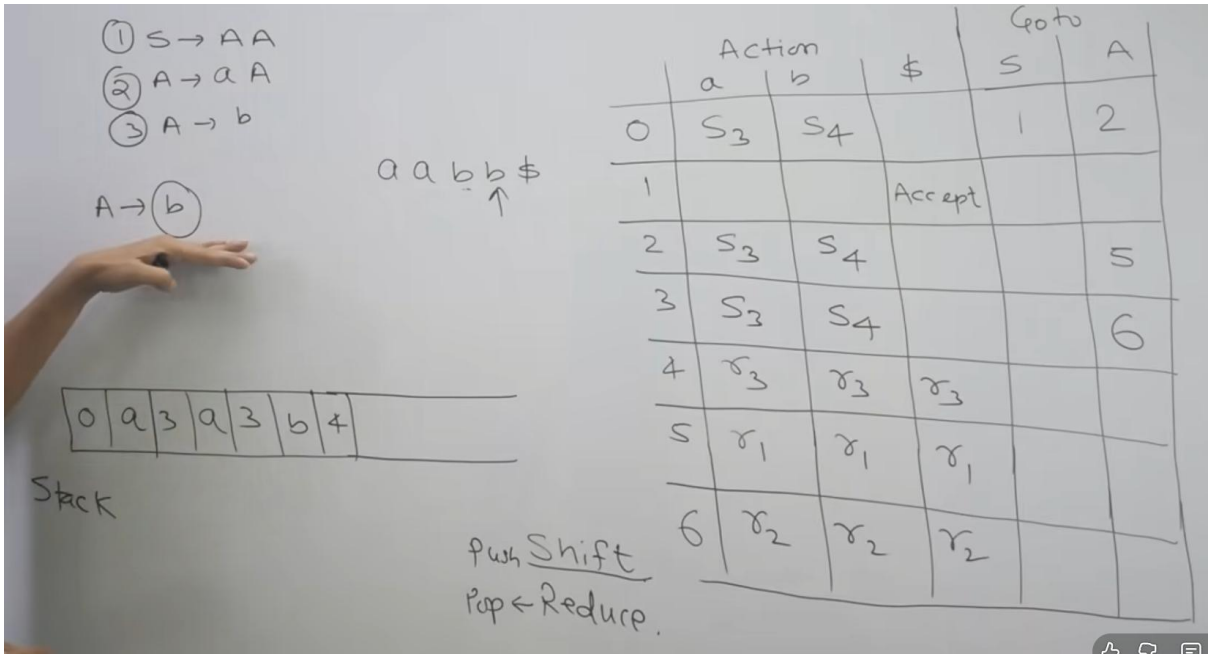
Bottom Up Parsers

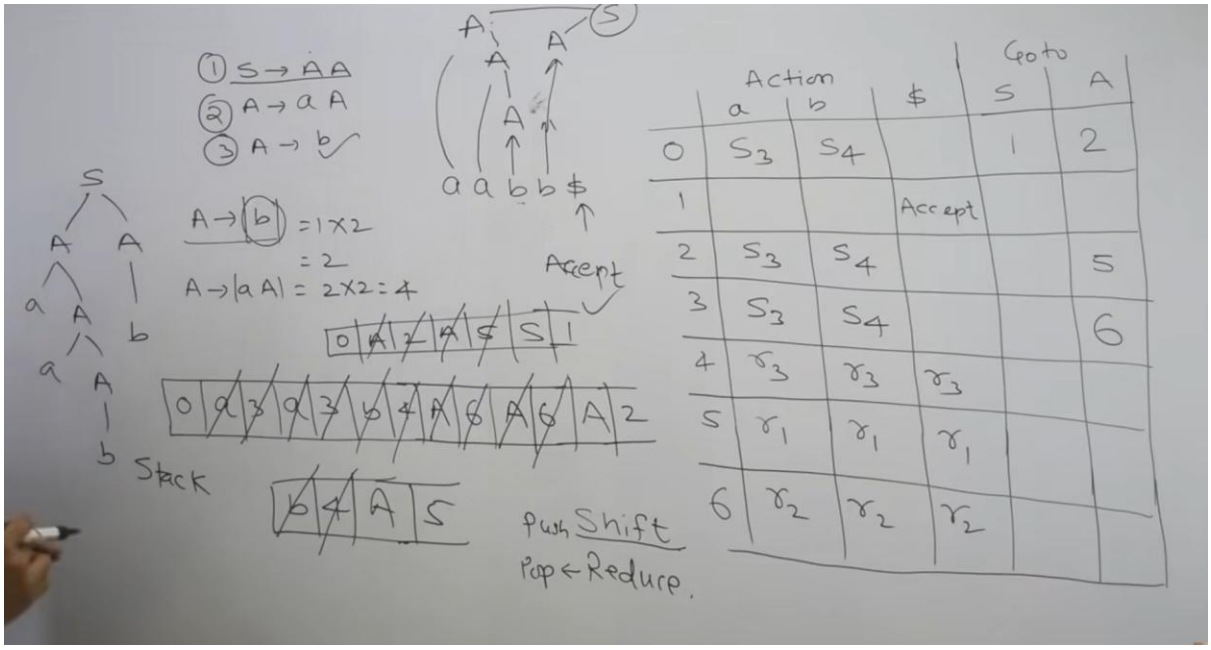
LR(0) (Left to right read, Rightmost Derivation in Reverse order):





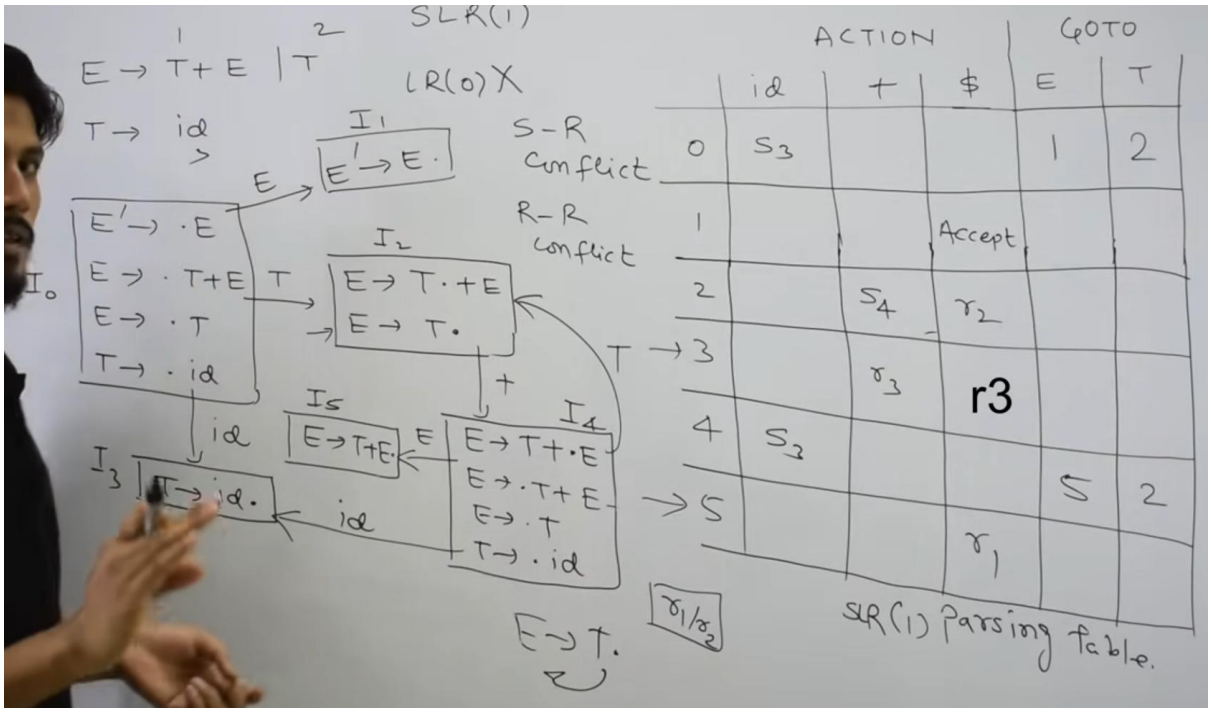

① S → AA
② A → aA
③ A → b
S' → S ✓
S → AA
A → aA | b
LR(0)
I0
S' → ·S
S → ·AA
A → ·aA
A → ·b
I1
S' → S·
I2
S → A·A
A → ·aA
A → ·b
I3
A → a·A
A → ·aA
A → ·b
I4
A → b·
I5
S → AA·
I6
A → aA·
Action
Goto
a
b
$
S
A
0 S3 S4 1 2
1 Accept
2 S3 S4 5
3 S3 S4 6
4 r3 r3 r3
5 r1 r1 r1
6 r2 r2 r2


① S → AA
② A → aA
③ A → b
A → b
a a b b $
0 a 3 a 3 b 4
Stack
Push Shift
Pop ← Reduce.
Action
Goto
a
b
$
S
A
0 S3 S4 1 2
1 Accept
2 S3 S4 5
3 S3 S4 6
4 r3 r3 r3
5 r1 r1 r1
6 r2 r2 r2



SLR(1) Shift LR(0):

If there's an SR or RR conflict (two S/Rs in same cell) then find the Follow of that production and add only in those cells

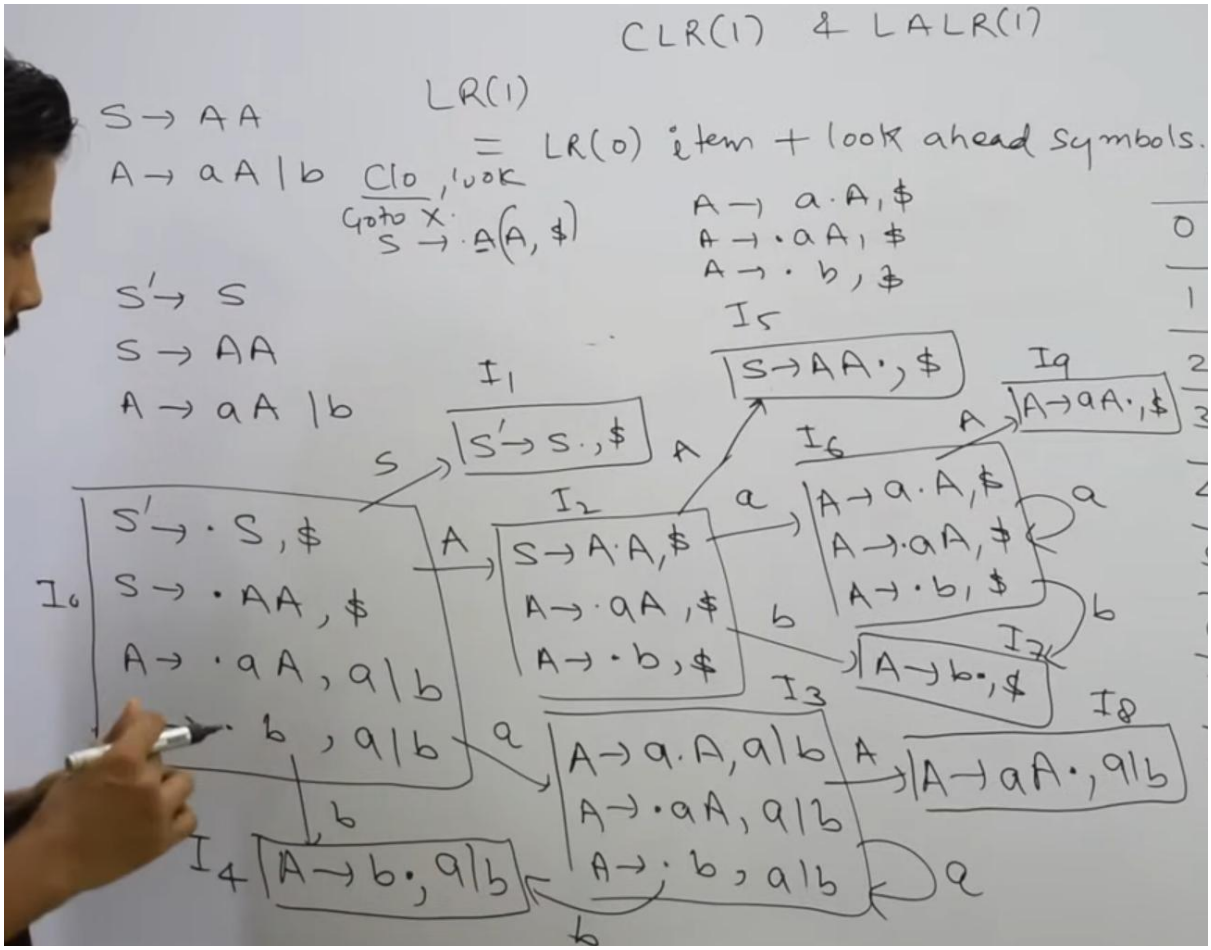


Next popping same as LR(0)

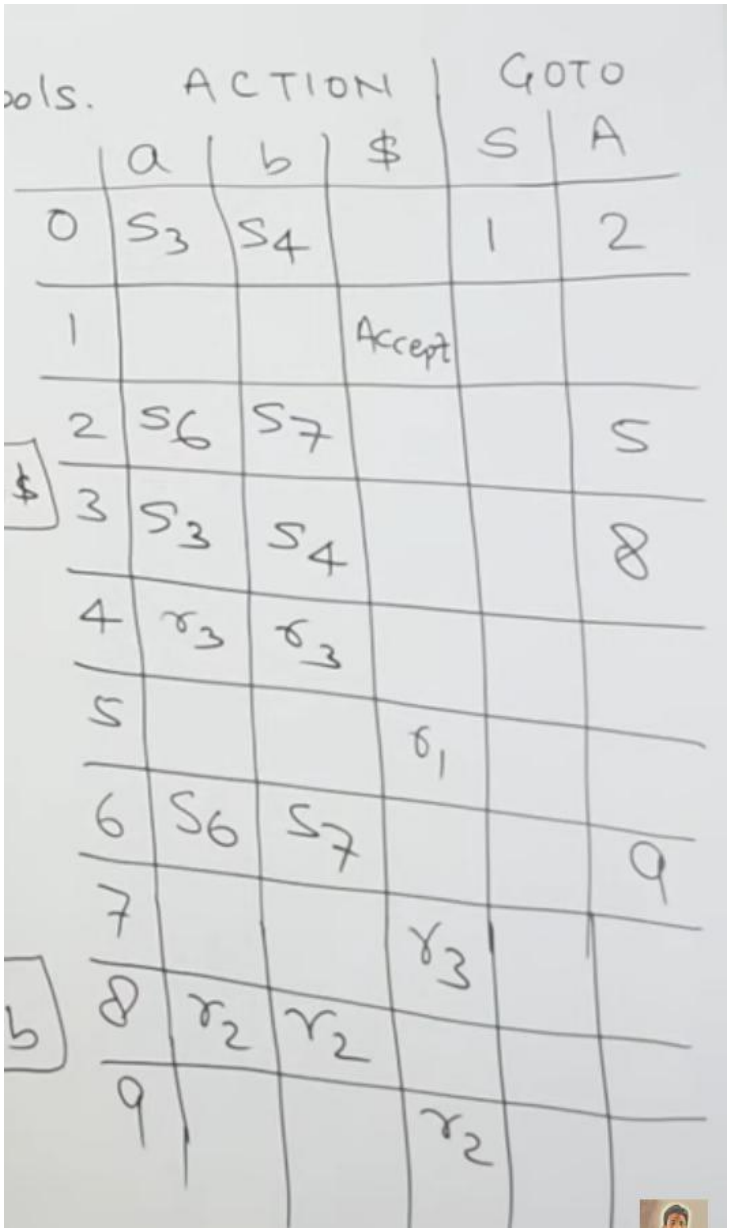
CALR(1) Canonical Form Items:

LR(0) Item + Look Ahead Symbols (Follow of everything ahead of .X)

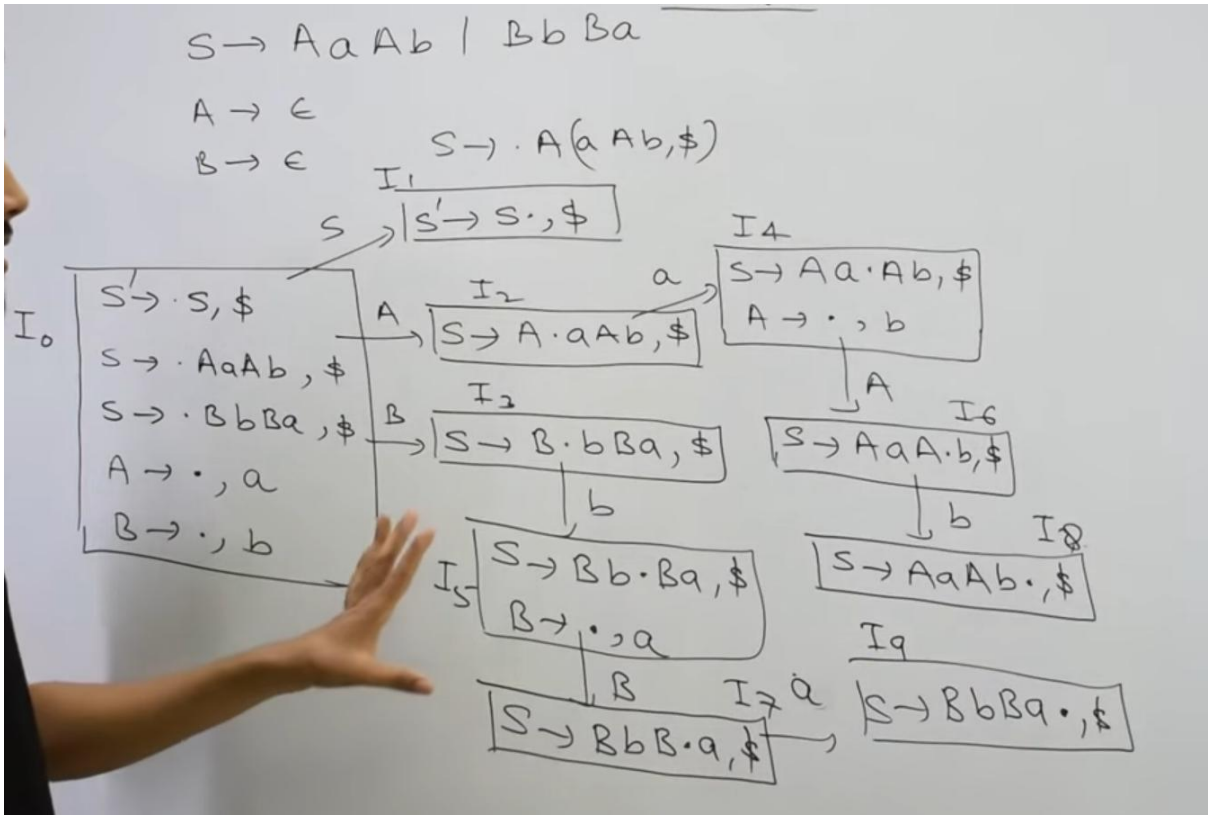
Ex 1: S->.Aa , $

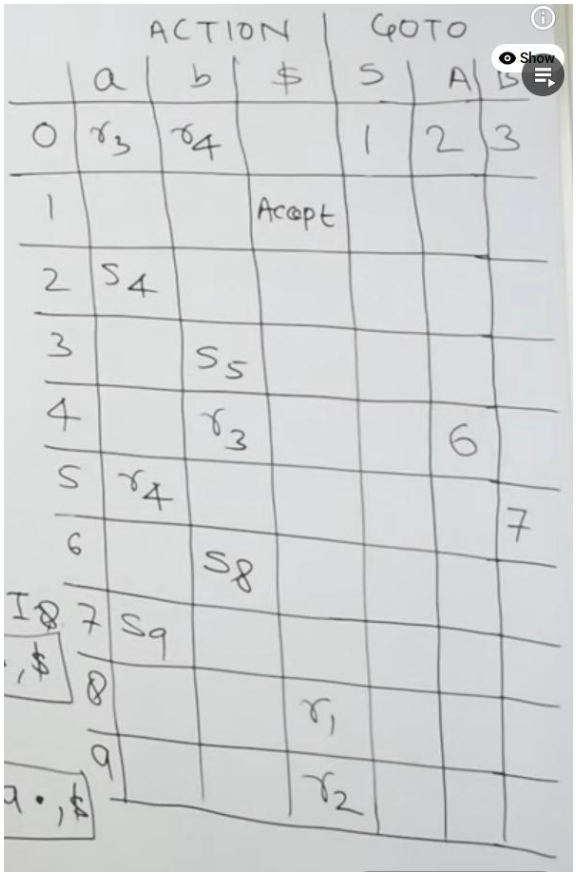


Filling of table same as LR(0), but here the Reduce will be filled in Look Ahead Symbols of Final States (Reduced states)


ols.
ACTION
GOTO
a
b
$
S
A
0
S_3
S_4
1
2
1
Accept
2
S_6
S_7
5
3
S_3
S_4
8
4
r_3
r_3
5
r_1
6
S_6
S_7
9
7
r_3
8
r_2
r_2
9
r_2
$
b

Ex 2:

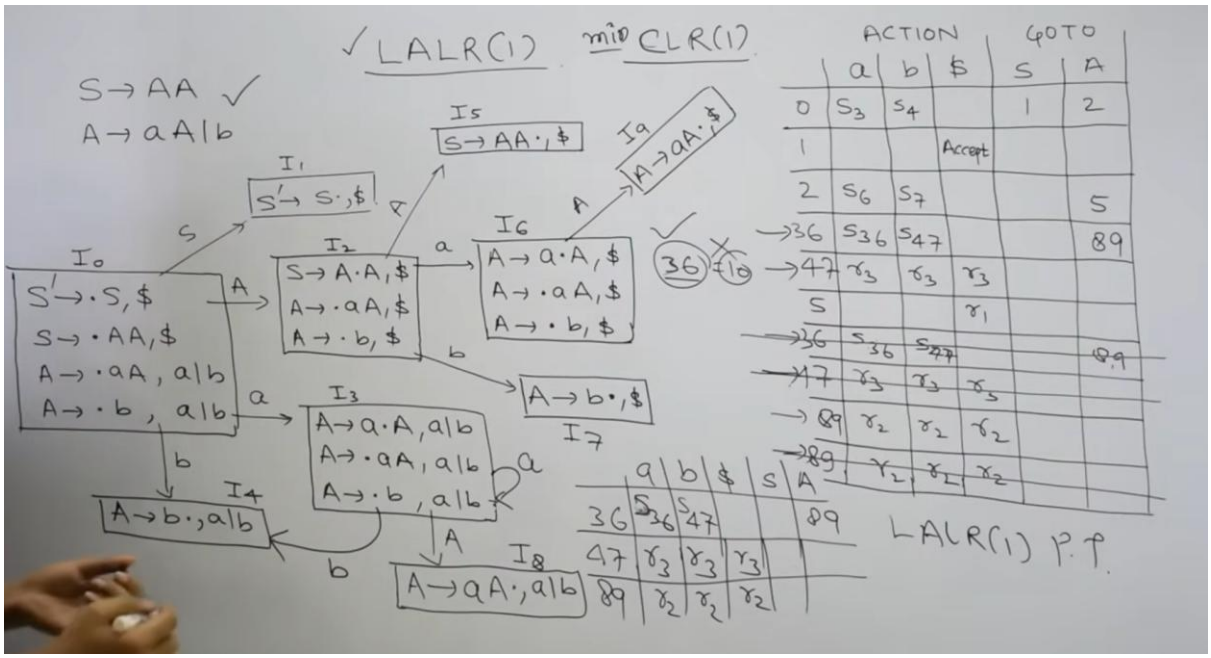

S → AaAb | BbBa
A → ∈
B → ∈
S → · A(aAb,$)
I1
S' → S·,$
I0
S' → ·S,$
S → ·AaAb,$
S → ·BbBa,$
A → ·,a
B → ·,b
I2
S → A·aAb,$
I3
S → B·bBa,$
I4
S → Aa·Ab,$
A → ·,b
I5
S → Bb·Ba,$
B → ·,a
I6
S → AaA·b,$
I7
S → BbB·a,$
I8
S → AaAb·,$
I9
S → BbBa·,$


ACTION
GOTO
a
b
$
S
A
B
0 r3 r4 1 2 3
1 Accpt
2 S4
3 S5
4 r3 6
5 r4 7
6 S8
7 S9
8 r1
9 r2

LALR(1):

Same as CALR(1) steps, merge the states that have the same LR(0) items (look ahead symbols may be different), and then fill the rows with the productions and then eliminate the repeated merge state. If there's no Conflicts (SR, RR) in any cell, then its valid.

Ex 1:



Ex 2: