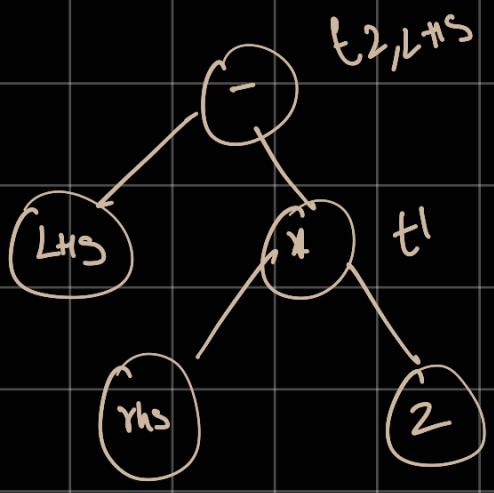
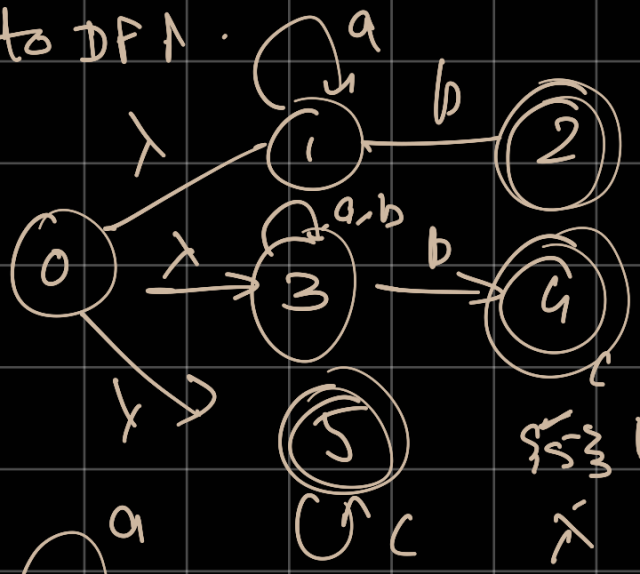
lhs = 0
rhs = 2
to :- lhs <= rhs
if to goto L1
goto L2
L1 : t1 = 2 * rhs
t2 = lhs - t1
lhs = t2
L2 : next

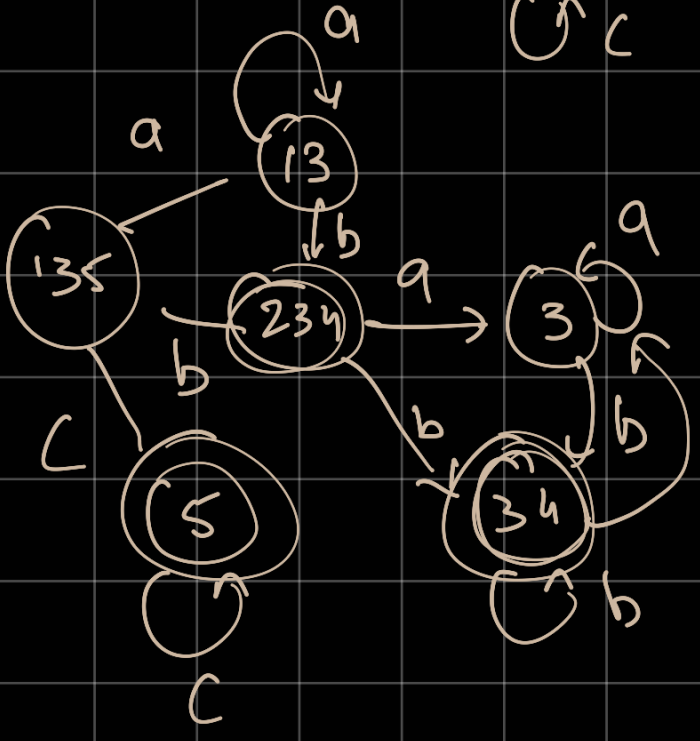


LHS = LHS - (rhs * 2)
= LHS - t1

NFA to DFA :



λ(0) : {1, 3, 5}
λ(1) = {1}
λ(2) = {2}



	λ	a	b
{5}	{1,3,5}	{1,3,5}	{1,3}	{2,3,4}
λ	{1,3}	{1,3}	{1,3}	{2,3,4}
λ	{2,3,4}	{2,3,4}	{3}	{3,4}
{c}	{5}	{5}	λ	λ
λ	{3}	{3}	{3}	{3,4}
λ	{3,4}	{3,4}	{3}	{3,4}

Unit 2 numericals!

→ Parsing

- parsers take input of a grammar G & string w. produce a parse tree if G can generate w.

→ first & follow

- first of NT is the set of symbols that appears as the first symbol in a string that derives from the NT

first of T is the T itself.

first (t) = {t} if t is terminal

if $X \to \lambda$ first (x) = {λ}

if $X \Rightarrow Y_1 Y_2 Y_3$

first(x) = first (Y_1)

if $\lambda \in$ first(Y_1)

first(x) = {first(Y_1) − λ} ∪ {first(Y_2)}

if $\lambda \in$ first(Y_i) ∀ $1 \le i \le n$ then add λ to first(x)

Q) $E \to TE'$

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

$T \to FT'$

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

$F \to id \mid num \mid (E)$

find first set of NT "T"

first (T) = first(F) ∪ {*, λ}
- {id, num, (, *, λ}

- follow:

The NT should vanish if the symbol that follows NT is same as the LHS the symbol that follows it in the prod rule / in some

follow(S) = {$}

if A → αBβ & λ ∈ first(β)

follow(B) = {first(β) - λ} ∪ follow(A)

if A → αB

follow(B) = follow(A)

Q) E → TE'

E' → + TE' | λ

T → FT'

T → * FT' | λ

F → id | num | (E)

find follow of all NT

first(E') {+, λ}

follow(E) = {$}

follow(T) = {first(E') - λ} ∪ follow(E) = {+} ∪ ....

follow(E) = follow(E)

follow(T) = {+} ∪ follow(E')

follow(E') = follow(E')

Q) S → UVW

U → v | WvU | λ

V → w | xU | λ

W → y | z

follow(U) = {first(V) − λ} ∪ {first(W)}
= {w, x, y, z}

follow(w) = first(V) = {x}

follow(U) = follow(V)

follow(V) = first(W) = {y, z}

follow(W) = follow(S) = $

S	{$}
U	{w, x, y, z}
V	{y, z}
W	{v, $}

Q) S → A B C D E

A → a | ϵ

B → b | ϵ

C → c

D → d | ϵ

E → e | ϵ

	first	follow
S	{a, b, c}	{$}
A	{a, ϵ}	{b, c}
B	{b, ϵ}	{c}
C	{c}	{d, e, $}
D	{d, ϵ}	{e, $}
E	{e, ϵ}	{$}

follow(S) = {$}

follow(A) = {b, c} follow(B) = {c}

follow(C) = {d, e} ∪ follow(S) = {d, e, $}

follow(D) = {e, $} follow(E) = {$}

e) $S \to ACB \mid CbB \mid Ba$

$A \to da \mid BC$

$B \to g \mid \epsilon$

$C \to h \mid \epsilon$

	first	follow
S	{d, g, h, ε, b, a}	{$}
A	{d, g, h, ε}	{g, h, $}
B	{g, ε}	{h, a, g, $}
C	{h, ε}	{g, h, b, $}

first(A) C B C

first(S) = {d, g, h, λ} ∪ {h, λ} ∪ {g, λ} ∪ {h, λ} ∪ {b} ∪ {g, λ} ∪ {a}

= {d, g, h, λ, b, a}

first(A) = {d} ∪ {g, h, λ} = {d, g, h, λ}

first(B) = {g, λ} first(C) = {h, λ}

A → BC ~~follow(B) = {h} ∪ follow(A)~~ → {g, h, $}

~~follow(C) = follow(A)~~

~~= {g, h, $}~~

S → ABC | CbB | Ba follow(S) = {$}

follow(A) = {g} ∪ {h} ∪ follow(S) = {g, h, $}

follow(B) = {h, $} = {h, $, a}

follow(C) = {b} = {h, $, a, g}

~~follow(B) = {$}~~

~~follow(B) = {a}~~

→ = {g, h, $, b}

→ LL(1)

A → FIT is in LL(1) if:

first(F) & first(T) are disjoint

if ε ∈ first(F) first(F) & follow(A) are "

" " ∈ first(t) " (T) " " " " "

- table:

. add the prd to every col of element in first(NT)

. if ε ∈ first(NT) then for each element in follow(NT) add production: (including $)

		first	follow
0)	E → tE'	E {(, id}	), $
	E' → +TE' | ε	E' {+, ε}	), $
	t → FT'	T {(, id}	+,), $
	T' → *FT' | ε	T' {*, ε}	+,), $
	F → (E) | id	F {(, id}	*, +, $

first(F) = fir(T)

= fir(F)

= {(, id}

first(E') = {+, ε}

first(T) = {(, id}

first(T') = {*, ε}

fist(F) = {(, id}

follow(E) = {)}

follow(F) = {*} ∪ follow(T') = {*, +, $}

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

follow(F) = {*} ∪ follow(T)

follow(T') = follow(T) = {*, +, $}
= {+, $}

follow(T) = {+} ∪ follow(E')
= {+, $}

follow(E') = follow(E') = {$}

follow(+) = {+} ∪ follow(E)
= {+, $}

follow(E') = follow(E) = {$}

follow(E) = {$}

	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)		

input = id + id * id $

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

Stack	input buffer	Action
E$	id+id*id$	push +E'
TE'$	id+id*id$	push FT'
FT'E'$	id+id*id$	push id
idT'E'$	id+id*id$	delete id
T'E'$	+id*id$	replace T' with ε
E'$	+id*id$	replace E' → +TE'
+TE'$	+id*id$	delete +
TE'$	id*id$	T → FT'
FT'E'$	id*id$	F → id
idT'E'$	id*id$	delete id
T'E'$	*id$	T' → *FT'
*FT'E'$	*id$	
FT'E'$	id$	
idT'E'$	id$	
T'E'$	$	

E'$ $

$ $ ACCEPT STRING

- LL(k) if LL(E) has more than k entries in any cell it's not LL(k)

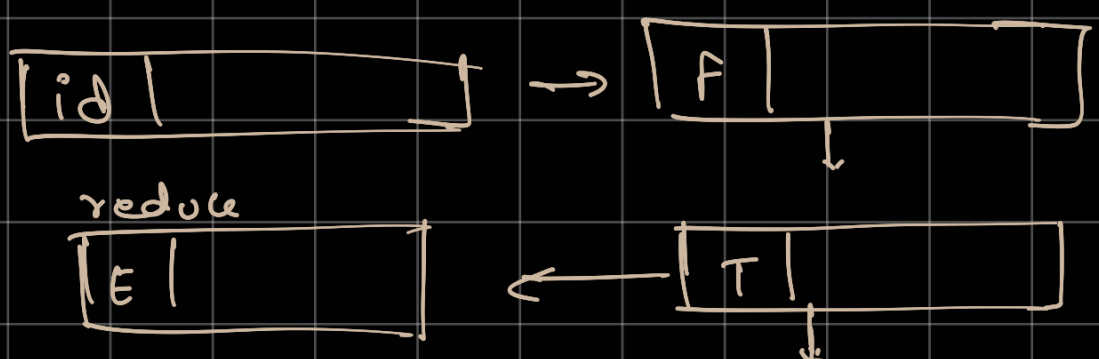
Note: before finding first follow, one needs to left factor it & remove any left recursion

→ BUP: Shift reduce

- No restriction of left factored or no left recursion.
- reads input left to right by give rightmost derivation in reverse : LR parsers
- "Handle pruning" ↔ reduce
- E → E+T | T w = id * id.

  T → T * F | F

  F → id



shift

we had choice to either Shift / reduce. In this case Shift was right, but we had a shift/reduce conflict

Yacc resolves S/r conflict by choosing shift over reduce "greedy" & gives warning

r/r conflict: when 2 RHS are exactly same but 2 diff left hand side.

→ LR(0)

•) $(n_{LR(0)} = n_{SLR} = n_{LALR}) \leq n_{CLR}$

# of nodes for parsing.

•) DFA of LR(0) & SLR(1) is called LR(0)

" " LALR & CLR are called LR(1)

•) "Item" is a production with a .

$A \rightarrow xy.z$

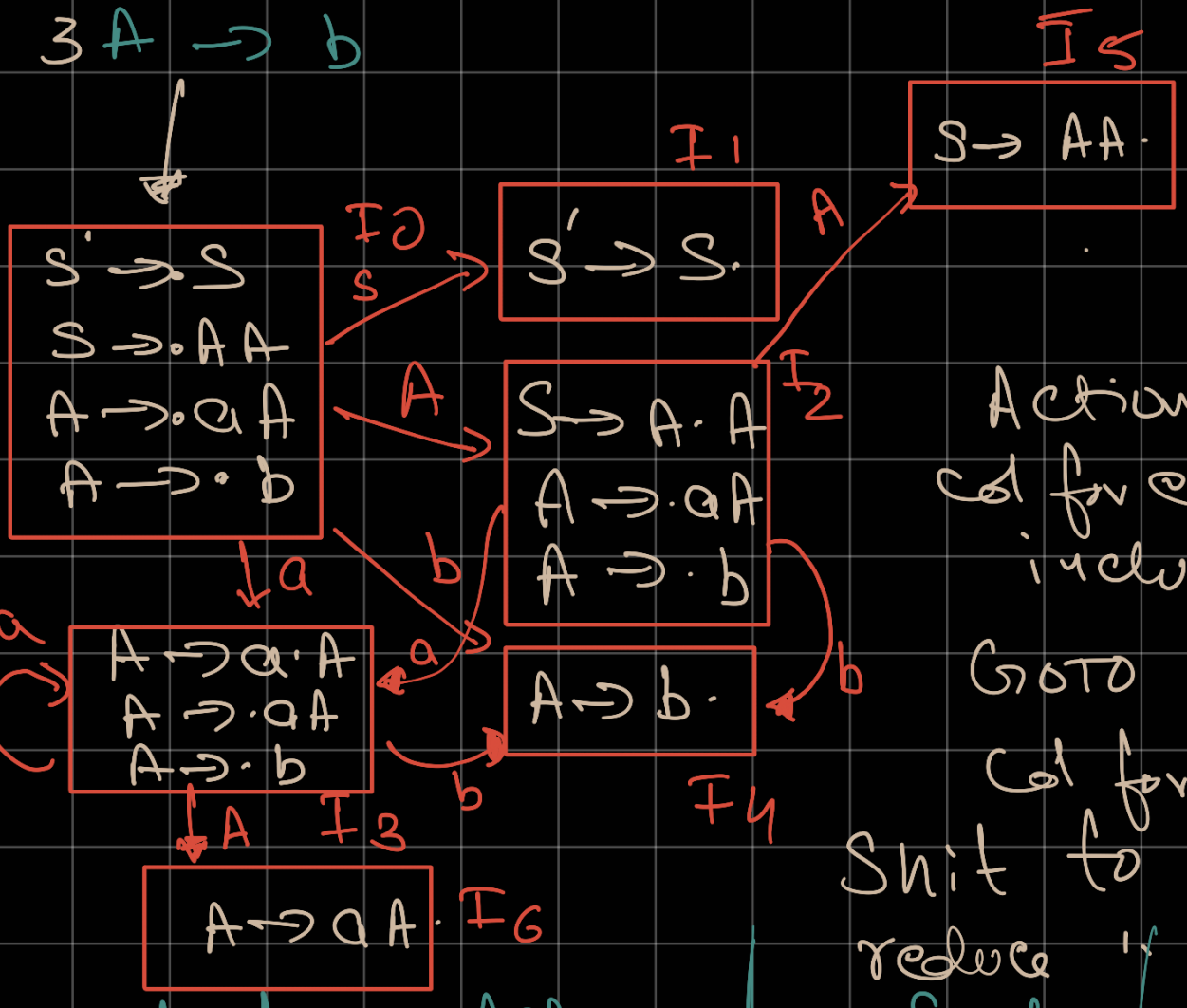
if . is in the end the its called an final item (∴ it's indicates we can perform reduction)

if . is not the first of RHS its called kernel item.

•) Unlike previous parsers, we have a GOTO & ACTION part in LR tables. this table remains same for all parsers only content changes.

ACTION tell whether to Shift or reduce. Any Conflicts will show up in this col.

1 $S \to AA$
2 $A \to aA$
3 $A \to b$



Action has col for each term including $

Goto has a col for every NT

Shift to state: s_x

reduce " : r_x

state	Action			Goto	
	a	b	$	S	A
0	s_3	s_4		1	2
1			accept		
2	s_3	s_4			5
3	s_3	s_4			6
4	r_3	r_3	r_3		
5	r_1	r_1	r_1		
6	r_2	r_2	r_2		

If state was final item in $ it should accept

Please reduce on entire row # is the prod no.

These have final items

reduce # is the # the prod was given befor augumenting the grammar

- parsing a string: "abb"

Stack	I/p	Action
$0	abb$	
$0a3	bb$	
$0a3b4	b$	Pop off 2 times the # Prod RHS where $x \to x$ push LHS
$0a3A6	b$	
$0A2	b$	
$0A2b4	$	
$0A2A5	$	
$0S1	$	
$0S1	$	$ → Accept.

- S/r Conflict:

E → T · + E
E → T ·

- r/r Conflict

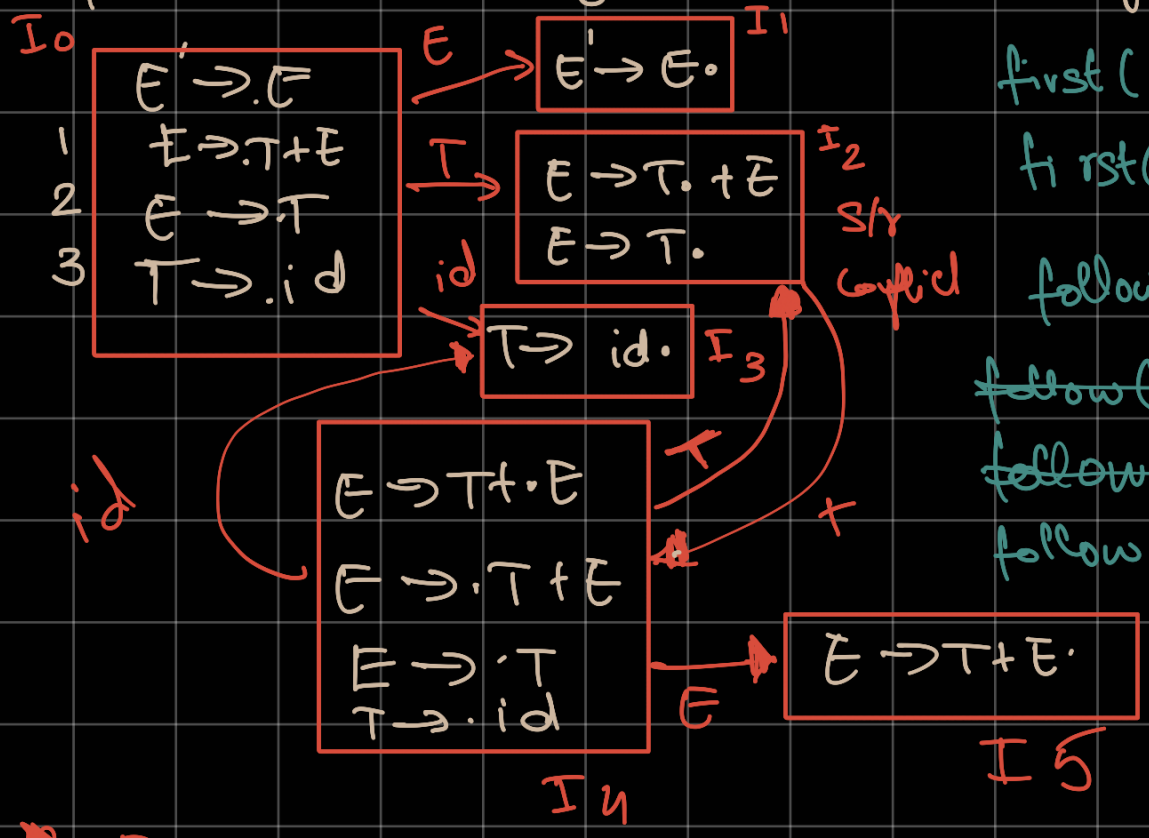
S → a ·
A → a ·

→ 2 diff prod #'s before augumenting

SLR(1)

- These also use the same LR(0)
- diff occurs in the way of assigning reduce in table.

•) Place reduce only in follow(LHS) of production.



first(E) = first(T) = {id}

first(T) = {id}

follow(T) = {+, $}

~~follow(T) = {+}~~

~~follow(E) = follow(E)~~

follow(E) = {$}

Remember to write closure of closures too!!!

State	Action			Goto	
	id	+	$	E	T
0	S_3			1	2
1			Accept		
2		S_4 / r_3	r_2 / r_3		
3					
4	S_3			5	2
5			r_1		

if in LR(0) definetly SLR. If SLR may or may not be in LR(0)

•) Note: SLR only partially handled conflicts.

r/r conflicts still exsist in SLR.

if 2 final items & intersection of follow of LHS of two 2 is not empty

Similarly for S/r if Symbol for Shift as a terminal exists in follow set of LHS of final item.

$B \to \alpha . a$ if follow(A) has a
$A \to \alpha .$ then S/r

LR(1)

•) We have to build lookahead.

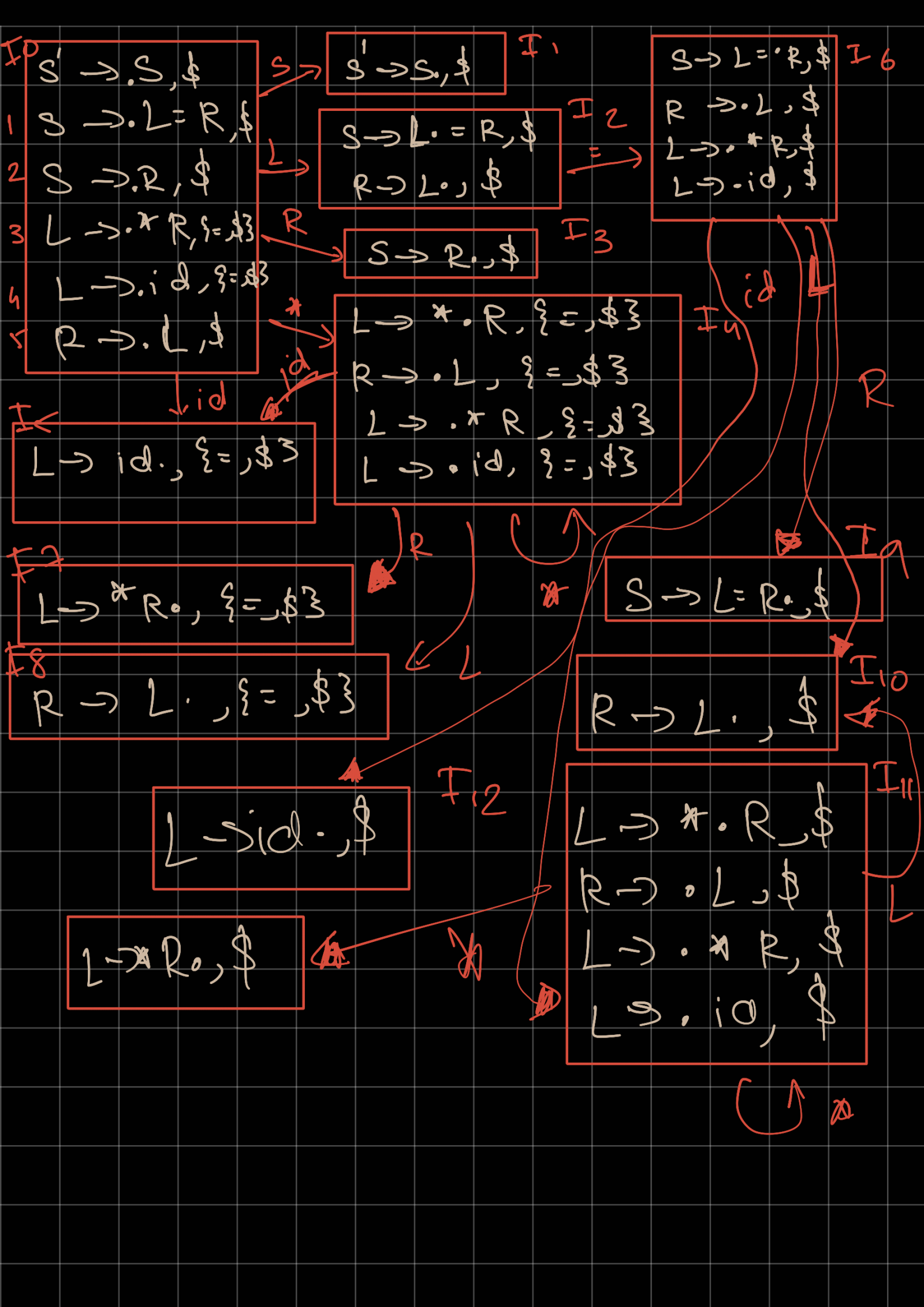
a) $S \to L = R \mid R$
$L \to * R \mid id$
$R \to L$

first(S) = first(L) ∪ first(R)
= {*, id}

first(L) = {*, id}
first(R) = {*, id}

follow(L) = follow(R) = {=, ϵ}
follow(R) = follow(L) = {=, ϵ}
follow(L) = {=}
follow(R) = follow(S) = {ϵ}
follow(S) = {ϵ}

follow(S) = {ϵ}
follow(L) = {=, ϵ}
follow(R) = "


I0
S' → .S, $
S → .L = R, $
S → .R, $
L → .* R, {=, $}
L → .id, {=, $}
R → .L, $
I1
S' → S., $
I2
S → L. = R, $
R → L., $
I3
S → R., $
I4
L → * . R, {=, $}
R → . L, {=, $}
L → . * R, {=, $}
L → . id, {=, $}
I5
L → id., {=, $}
I6
S → L = . R, $
R → . L, $
L → . * R, $
L → . id, $
I7
L → * R., {=, $}
I8
R → L., {=, $}
I9
S → L = R., $
I10
R → L., $
I11
L → * . R, $
R → . L, $
L → . * R, $
L → . id, $
I12
L → id., $
I13
L → * R., $
S
L
R
id
*
=

- Lookahead = first of all term after NT in prod including parents lookahead.

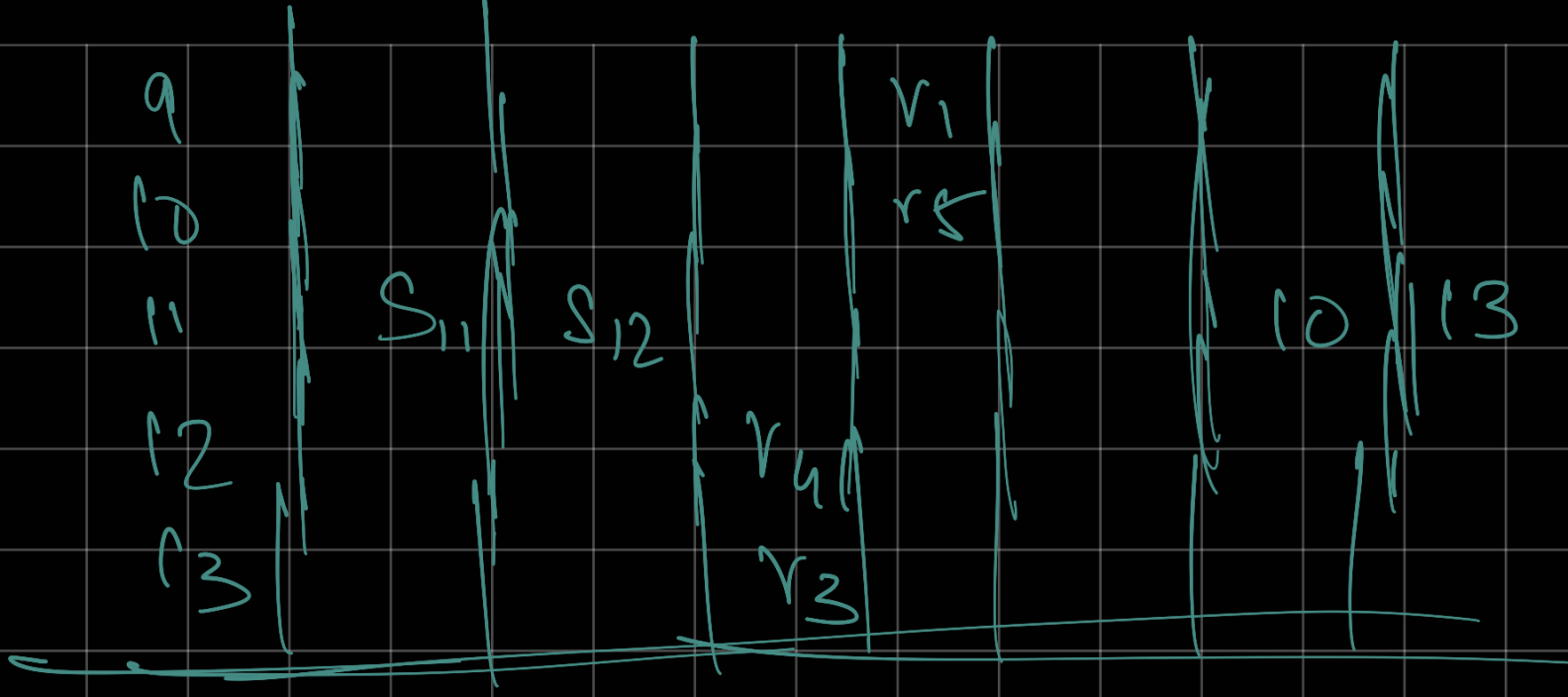
There mistake in slide, above is correct..

As per table, we get accept only in [1, $]. We put reduce in all the lookahead. [CLR table]

-> In LALR we try to merge those item set with same itemset but diff forward lookaheads.

CLR table:

State	Action				GO TO		
	*	id	=	$	S	L	R
0	S_4	S_5			1	2	3
1				accept			
2			S_6	r_5			
3				r_2			
4	S_4	S_5				8	7
5			r_4	r_4			
6	S_{11}	S_{12}				10	9
7			r_3	r_3			
8			r_5	r_5			



for LALR we need to merge states:

I_4 I_{11} rep as 4

I_5 I_{12} as 5

I_7 I_{13} as 7

I_8 I_{10} as 8

$I_4 \in I_{11}$ Could not be merged if $5 \in 12$ did not merge.

If any conflicts just don't merge.

LALV is weaker than CLR

Yacc: is a LALR parser

a) S/r: in CLR

$A \to X \cdot a, \{\dots\}$

$B \to X a \cdot, \{a\}$

r/r in CLR

$A \to X a \cdot \{a, \dots$

$B \to X b \cdot \{a, \dots$

b) LALR can introduce r/r while merging

an r/r exist in LALR only if during merging there are at least 2 (more final items with lookaheads.