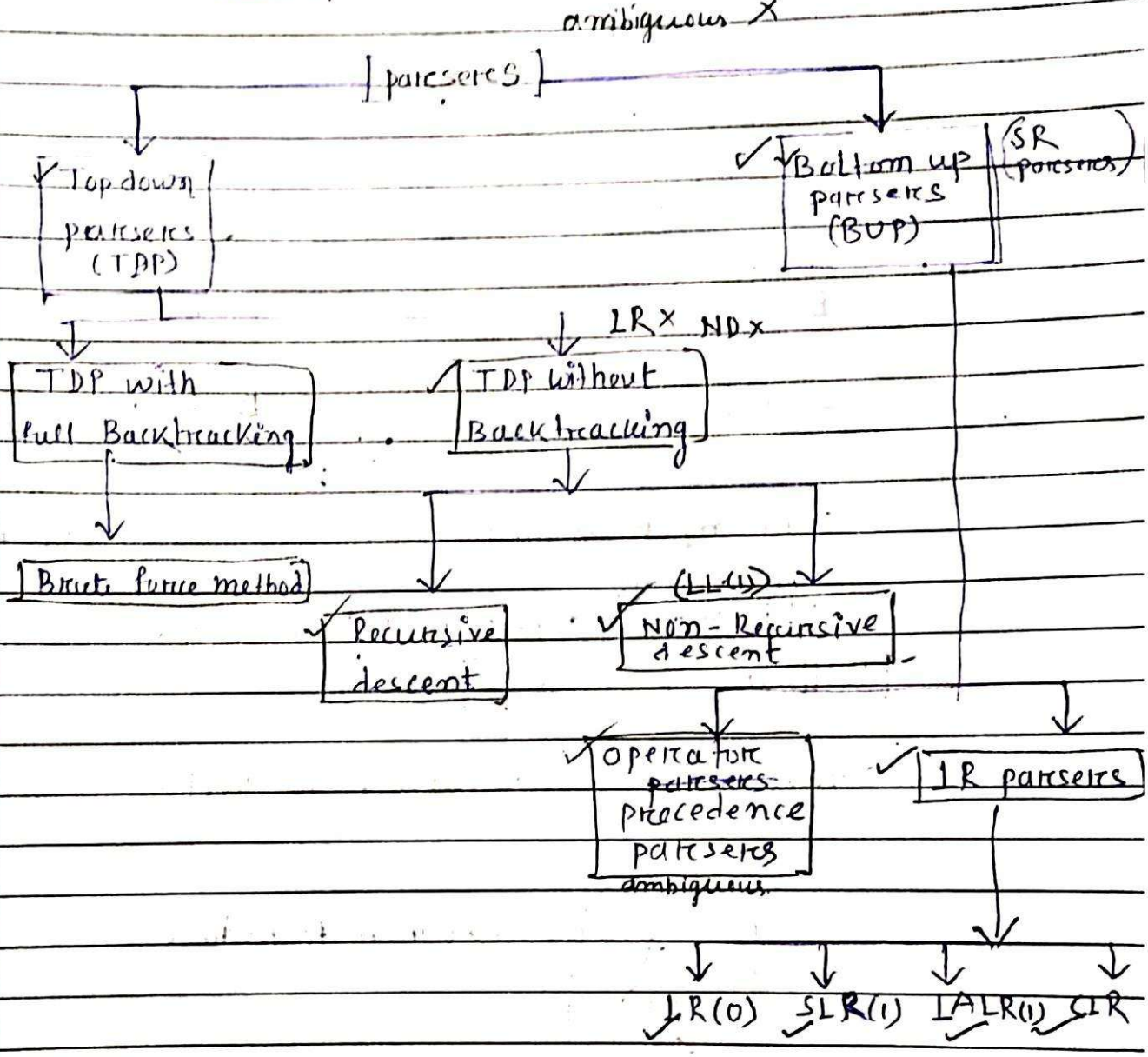
PARSERS

- Introduction of parsers :



→ parsers is syntax analyser.

→ SR parsers means shift-reduce parsers.

→ TDP without backtracking not accept - Left recursive grammar and also Non-Deterministic grammar.

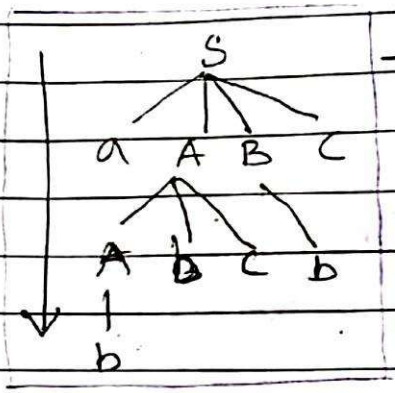
→ parsers not accept ambiguous grammar.
only operator precedence parsers accept ambiguous grammar.

- Basic difference between Top-down parsers and Bottom up parsers:

Top down parsers

$S \rightarrow aABe$
$A \rightarrow Abc/b$
$B \rightarrow d$

$w = abbcde$



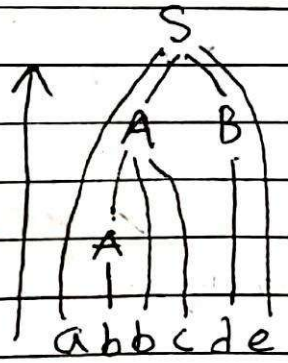
→ what to use.

→ Staring from the the start and going to the ~~bot~~ terminal. follow.

→ TDP ~~also used~~ Left most Derivation.

Bottom up parsers

→ take the terminals and go to root.



→ when to reduce

$w = abbcde$.

$S \Rightarrow a\,A\,B\,e$
$\Rightarrow a\,A\,d\,e$
$\Rightarrow a\,A\,b\,c\,A\,e$
$\Rightarrow a\,b\,b\,c\,d\,e$

→ BUp follow Right most Derivation.

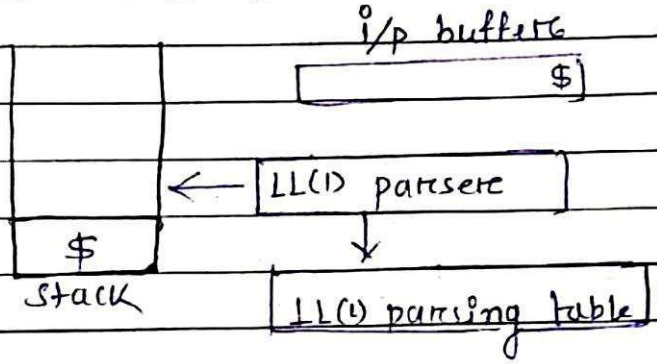
• LL(1) parsers :

LL(1) → No. of Lookerhead.

- Scaning Left to Right
- Left most derivation

abcd ⟹

• Components of parsers =

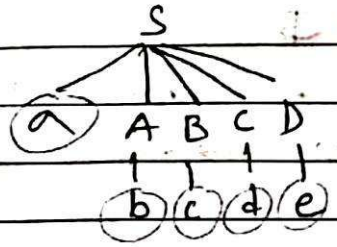


→ '$' is to guess when stop.

- First()
- Follow()

First()

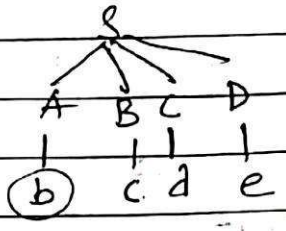
Ex- S → aABCD
A → b
B → c
C → d
D → e



here, first of S is a.
" " A " b
" " B " c
" " C " d
first of D " e

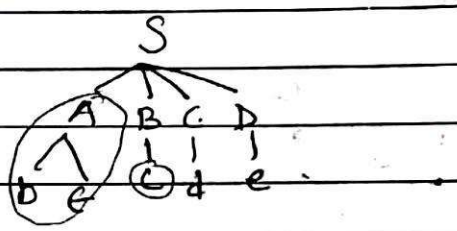
Ex-

$S \rightarrow ABCD$
$A \rightarrow b$
$B \rightarrow c$
$C \rightarrow d$
$D \rightarrow e$



→ here first of S is 'b'.

Ex- $S \rightarrow A BCD$
$A \rightarrow b/\epsilon$
$B \rightarrow c$
$C \rightarrow d$
$D \rightarrow e$



→ here in this case first of S is 'c'.

Follow:

What a terminal which can follow a variable in the process of derivation.

$S\$$
$abcd\$$
$ABCD\$$ → here follow of D is '$'
$ABdD\$$ → follow of B is 'd'.

$S \rightarrow ABCD$
$A \rightarrow b/\epsilon$
$B \rightarrow c$
$C \rightarrow d$
$D \rightarrow e$

→ follow of D is follow of 'S'.

→ here in this case follow of S - {$}
follow of A is first of BCD = 'c'.
So, follow of A is {c}.

follow of B is {d}.
" " C is {e}.
follow of D is {$}.

$A \rightarrow BC$

- Example— find first and follow in LL(1):

Ex-1

	first	follow
6 $S \rightarrow ABCDE$	{a, b, c}	{$}
1 $A \rightarrow a/\epsilon$	{a, ε}	{b, c}
2 $B \rightarrow b/\epsilon$	{b, ε}	{c}
3 $C \rightarrow c$	{c}	{d, e, $}
4 $D \rightarrow d/\epsilon$	{d, ε}	{e, $}
5 $E \rightarrow e/\epsilon$	{e, ε}	{$}

→ follow of A is first of BCDF = {b, c}.

Ex-2

	first	follow
3 $S \rightarrow Bb/Cd$	{a, b, c, d}	{$}
1 $B \rightarrow aB/\epsilon$	{a, ε}	{b}
2 $C \rightarrow cC/\epsilon$	{c, ε}	{d}

Ex-3

	first	follow
5 $E \rightarrow TE'$	{id, (}	{$,)}
4 $E' \rightarrow +TE'/\epsilon$	{+, ε}	{$,)}
3 $T \rightarrow FT'$	{id, (}	{+, $,)}
2 $T' \rightarrow *FT'/\epsilon$	{*, ε}	{+, $,)}
1 $F \rightarrow id/(E)$	{id, (}	{*, +, $,)}

Ex-4:

	first	follow
4 S → ACB / CbB / Ba	{d, g, h, ϵ, b, a}	{$}
3 A → da / BC	{d, g, h, ϵ}	{h, g, $}
2 B → g / ϵ	{g, ϵ}	{$, a, h, g}
1 C → h / ϵ.	{h, ϵ}	{b, h, g, $}

Ex-5:

	first	follow
S → aABb	{a}	{$}
A → c / ϵ	{c, ϵ}	{d, b,}
B → d / ϵ	{d, ϵ}	{b}

Ex-6

	first	follow
S → aBDh	{a}	{$}
B → cC	{c}	{g, f, h}
C → bC / ϵ	{b, ϵ}	{g, f, h}
D → EF	{g, f, ϵ}	{h}
E → g / ϵ.	{g, ϵ}	{f, h}
F → f / ϵ	{f, ϵ}	{h}

A → CAB

here follow of 'A' is = follow of 'A'.

A → ABCD

here follow of B is first of 'C' and 'D'.

• Construction of LL(1) parsing table :

Example – 1

	first	follow
$E \to TE'$	{id, (}	{$,)}
$E' \to +TE' / \epsilon$	{+, ε}	{$,)}
$T \to FT'$	{id, (}	{+, $,)}
$T' \to *FT' / \epsilon$	{*, ε}	{+, $,)}
$F \to id / (E)$	{id, (}	{*, +, $,)}

→ LL(1) parser is a top down parser.

→ The main purpose of top down parser is when have two alternative for a variable, find out which production should choose.

Table

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

↑ variables ↓

→ All the 'ε' production have to place under follow of 'left hand side'.

→ Where should place the production is depends on the first of right hand side.

→ To construct parsers tree, this table used.

• Use of parsing to construct a parsers tree –

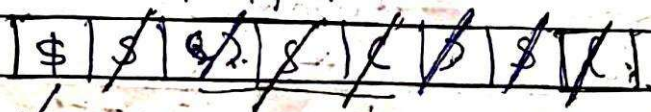
Example – 2 (( ))

	first	follow
$S \rightarrow (S)/\epsilon$	{(, ϵ}	{$,)}

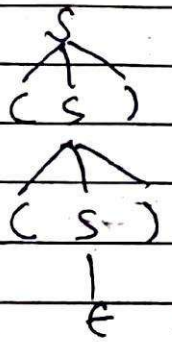
table:

	(	)	$
S	S → (S)	S → ϵ	S → ϵ

generate : (( )) $



bottom of the stack



→ LL(1) parsers algo start with '$' in the bottom of the stack and '$' in the end of the input.

~~when $, $ match then we can say~~ when we see $ & $ then we can say it is successful matching.

Example-3

	First	Follow
$S \rightarrow AaAb / BbBa$	$\{\epsilon, a, b\}$	$\{\$\}$
$A \rightarrow \epsilon$	$\{\epsilon\}$	$\{a, b\}$
$B \rightarrow \epsilon$	$\{\epsilon\}$	$\{b, a\}$

table =

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

→ In every cell of table we got '1' production, so that this grammar is LL(1).

→ Left recursion and Non-Deterministic grammar can not be used for LL(1).

$A \rightarrow \alpha_1 / \alpha_2 / \alpha_3 / \alpha_4 / \cdots \alpha_n$

A grammar has to be LL(1) when First of $\alpha_1, \alpha_2, \alpha_3, \cdots \alpha_n$ are mutualy exclusive (means no common)

• Test that given Grammar is LL(1) or not:

	first	follow
① $S \to aSbS$ / $bSaS$ / ϵ	{a, b, ε}	{b, a, $}

	a	b	$
S	S → aSbS	S → bSaS	
	S → ε	S → ε	S → ε

→ One cell contain more than one production, therefore this grammar is not LL(1).

	first	follow
② $S \to aABb$ (V)	{a}	{$}
$A \to c / \epsilon$	{c, ε}	{a, b}
$B \to d / \epsilon$	{d, ε}	{b}

	a	b	c	d	$
S	S → aABb				
A		A → ε	A → c	A → ε	
B		B → ε		B → d	

→ each cell contain only '1' production, therefore this given grammar is LL(1).

③

	first	follow
$S \to A / a$	{a}	{$}
$A \to a$	{a}	{$}

	a	$
S	S → A, S → a	
A	A → a	

→ Cell contain more than '1' production, not LL(1).
→ also it is a ambiguous Grammar.

④

	First	Follow
✓ $S \to aB / \epsilon$	$\{a, \epsilon\}$	$\{\$\}$
$B \to bC / \epsilon$	$\{b, \epsilon\}$	$\{\$\}$
$C \to cS / \epsilon$	$\{c, \epsilon\}$	$\{\$\}$.

table

	a	b	c	$
S	$S \to aB$			$S \to \epsilon$
B		$B \to bC$		$B \to \epsilon$
C			$C \to cS$	$C \to \epsilon$

→ each cell contain single one production. therefore LL(1).

⑤

	first	follow
✓ $S \to AB$	$\{a, b, \epsilon\}$	$\{\$\}$
$A \to a / \epsilon$	$\{a, \epsilon\}$	$\{b, \$\}$
$B \to b / \epsilon$	$\{b, \epsilon\}$	$\{\$\}$.

Parsere table =

	a	b	$
S	$S \to AB$	$S \to AB$	$S \to AB$
A	$A \to a$	$A \to \epsilon$	$A \to \epsilon$
B		$B \to b$	$B \to b$

→ each cell contain single one Production, therefore this grammar is LL(1).

⑥

	First	Follow
✗ $S \to a(S)A / \epsilon$	$\{a\}$	$\{c, \$\}$
$A \to c / \epsilon$	$\{c, \epsilon\}$	$\{c, \$\}$

	a	c	$
S	$S \to aSA$	$S \to ASA\epsilon$	$S \to \epsilon$
A		$A \to c$, $A \to \epsilon$	$A \to \epsilon$

→ one cell contained more than '1' production, so therefore it is not LL(1).

(7)

	first	follow
$S \to A$	{a, b, c, d}	{$}
$A \to Bb/Cd$	{a, b, c, d}	{$}
$B \to aB/\epsilon$	{a, ε}	{b}
$C \to cC/\epsilon$	{c, ε}	{d}

table

	a	b	c	d	$
S	S→A	S→A	S→A	S→A	
A	A→Bb	A→Bb	A→Cd	A→Cd	
B	B→aB	B→ε			
C			C→cC	C→ε	

→ each cell contain only '1' production so it is LL(1).

(8)

		first	follow
X	$S \to aAa/\epsilon$	{a, ε}	{$, a}
	$A \to abS/\epsilon$	{a, ε}	{a}

	a	b	$	
S	S→aAa, S→ε			S→ε
A	A→abS, A→ε			

→ not LL(1).

(9)

S→(iE)S

(9)

		first	follow
X	$S \to iEtSS'/a$	{i, a}	{$, e}
	$S' \to eS/\epsilon$	{e, ϵ}	{$, e}
	$E \to b$	{b}	{t}

table

	i	t	a	b	e	$
S	S→iEtSS'		S→a			
S'					S'→eS, S'→ϵ	S'→ϵ
E				E→b		

→ It not LL(1).

- Recursive descent parser =

Example

$E \to iE'$

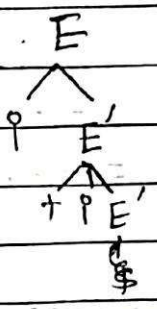
$E' \to +iE'/\epsilon$ → each ~~func~~ variable has a function.

```
E()
{
1. if (l == 'i')
2.   { match ('i');
3.     E'();
     }
}
```

```
E'()
{
1. if (l == '+')
   {
2.   match ('+');
3.   match ('i');
4.   E'();
   }
5. else
6.   return;
7. }
```

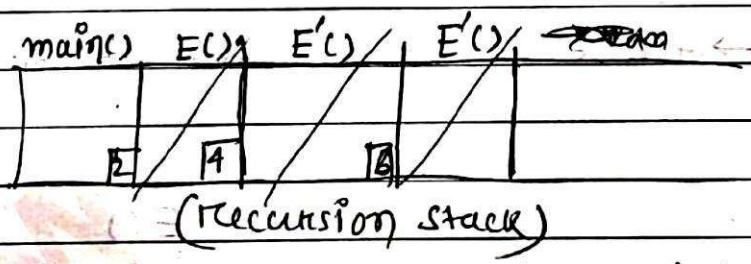
```
match (char t)
{
  if (l == t)
    l = getchar();
  else
    print ("error");
}

main()
{
1. E();
2. if (l == '$')
3. print ("parsing success");
}
```



l = getchar (incrementing)

according to grammar give i+i$ generated.



→ write function for every variable, then call the function using the stack provided by operating system.

- Operator precedence parsers: (Bottom up parsers)

- Operator grammar :-

✓ $E \to E+E/E*E/id$ → no two variable are adjacent.

✗ $E \to EAE/id$, $A \to +/*.$ $\Rightarrow E \to E+E/E*E/id$ ✓

Example: (given grammar are operators grammar or not) precedence

① $S \to SAS/a$
$A \to bSb/b$

⇒ this grammar is not operator precedence grammar, because two variable are not adjacent.

now going to convert this grammar into operator precedence grammar ⇐

$S \to SbSbS/SbS/a$
$A \to bSb/b$

→ operator precendence g can parse ambiguous grammar.

- Operation relation table :
→ using this table we can do parsing ⇐

① example -

$E \rightarrow E+E / E*E / id.$ (id > * > + > $)

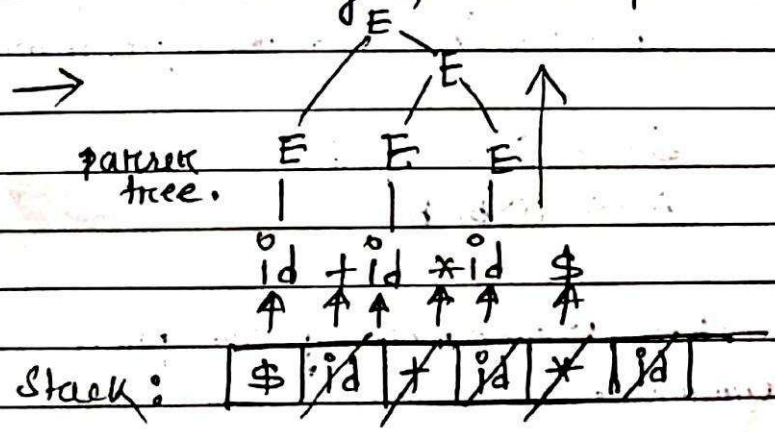
+ > +
* > *

table =

	id	+	*	$
id	—	·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	—

(Operation precedence table)
(Operator relation table)

Example ① parse (id + id * id $) → i/p
using operator precendence table.

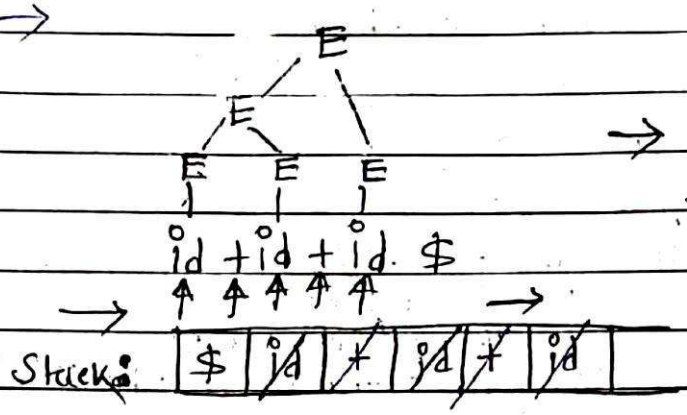


- When top of the stack less than or equal to input then push i/p in stack.
- When top of the stack are greater than to input then pope the top of the stack as terminal.

Example 2

Construct parser tree of id + id + id $ using operator precendence table.

→



→ left most '+' is greater than right most '+'.

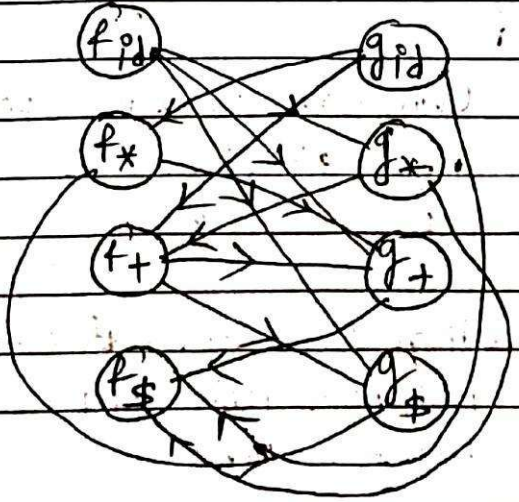
→ We will get one parser tree whether it is ambiguous.

• Disadvantage of operation relation table =

→ if we have '4' operator then size of the table are $4^2 = 16$.

if we have 'n' operator then size of table are n^2 or $O(n^2)$.

→ To decrease the size of the table we go for Operator function table =



f \ g	id	*	+	$
id	—	>	>	>
*	<	>	>	>
+	<	<	>	>
$	<	<	<	—

When greater than (>)

$f \longrightarrow g$

When less than (<)

$f \longleftarrow g$

any cycle

→ If in this graph then stop we can't make function table.

→ find the longest path from the staring node.

$f_{id} \xrightarrow{1} g_* \xrightarrow{2} f_+ \xrightarrow{3} g_+ \xrightarrow{4} f_\$$

$g_{id} \rightarrow f_* \rightarrow g_* \rightarrow f_+ \rightarrow g_+ \rightarrow f_\$$

Function table

	id	+	*	$
f	4	2	4	0
g	5	1	3	0

→ This table contain same information (like relation table) but, the size of the size of table less.

with 'n' input operator, sige of the table are $O(2n)$

From the table = (we can compare operators)

$f_+ \Rightarrow g_+$ (2, 1)

$f_* \Rightarrow f_*$ (4, 3)

→ Disadvantage of function table are, if when even blank entries in relation table, then we get non-blank entries in function table.

example —

- P → (SR)/S → not operator precendence gram. because two variable are adjecent.
- R → bSR/bS
- S → WbS/W
- W → L*W/L
- L → id .

⇒ P → (SR)/S → (I)
R → bSR/bS → (II)
S → WbS/W
W → L*W/L
L → id

P → paragraph.
S → sintence
R → recursive
b → Blank
W → word
L → letter.
id → Identifier.

convertion into operator precendence grammar.

P → Sb(SR)/SbS/S

→ P → SbP/SbS/S
S → WbS/W → 'b' is right associative.
W → L*W/L → 'i' Right most 'b' will get higher precedence compare to Left 'b'.
L → id . → '*' also right associative.

Operator relation table —

	id	*	b	$
id	—	⋗	⋗	⋗
*	⋖	⋖	⋗	⋗
b	⋖	⋖	⋖	⋗
$	⋖	⋖	⋖	—

→ If grammar are not ambiguous then we can make table from given grammar.

(a) table =

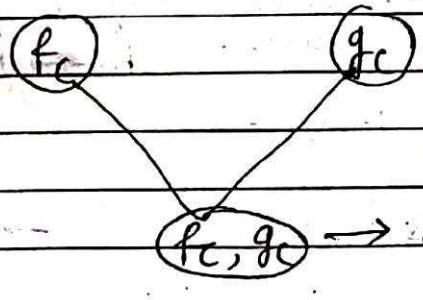
	a	(	)	,	$
a			⋗	⋗	⋗
(	⋖	⋖	≐	⋖	
)			⋗	⋗	⋗
,	⋖	⋖	⋗	⋗	
$	⋖	⋖			

(Operator relation table)

	a	(	)	,	$
f	2	0	2	2	0
g	3	3	0	1	0

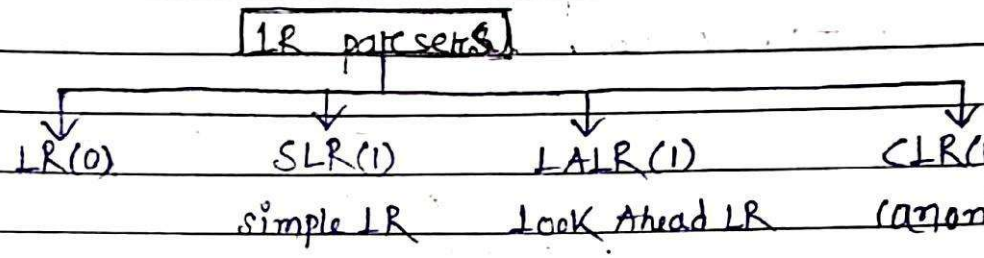
(Operator function table)

f (≐) g
0 0

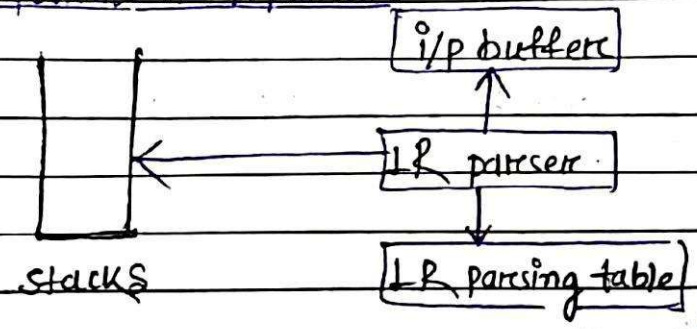


→ both w same value.

LR parsers: (Bottom up parser)



Components of LR parsers =



→ LR(0) items is used to construct the LR(0) and SLR(1) parsers.

→ LR(1) items is used to construct the LALR(1) and CLR(1) parsers.

Construct LR(0) parsing table =

$S' \rightarrow S$

$S \rightarrow AA$ ①

$A \rightarrow \alpha A / b$ ② ③

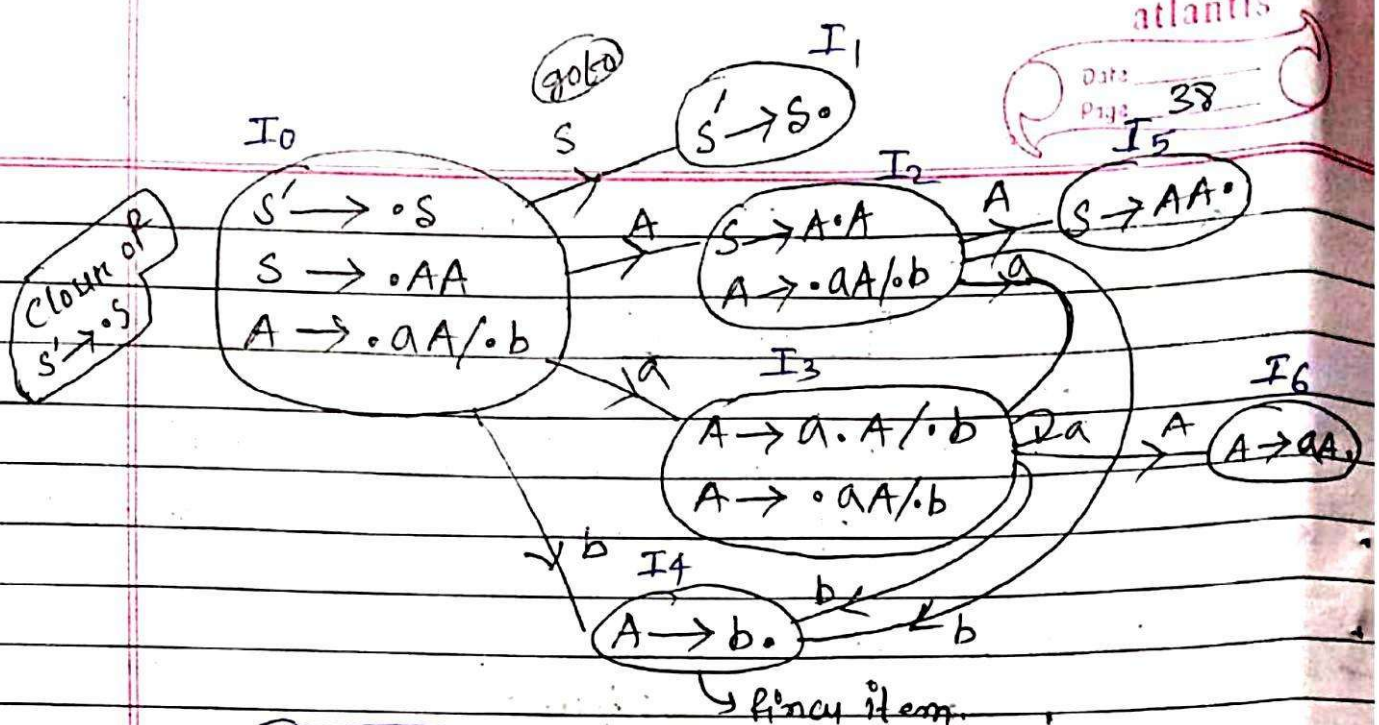
→ add one more production with given grammars.

→ closer: When a dot is left of a variable we have to add all the productions of dot in the left side of

$S' \rightarrow .S$

$S \rightarrow .AA$

$A \rightarrow .\alpha A / .b$



Canonical collection of LR(0) items

Steps —

(I) Take the grammar add a new production.
(II) and start with first ~~first~~ production by dot of the ~~set~~ left of S.
(III) then ~~all~~ applyed closure.
(IV) then find out transation (goto move) and again apply closure.

→ some states contain final items.
(any item dote at the ~~ed~~ end is called final item)

• LR(0) parsing table for canonical colection of LR(0) items —

→ next page —

Table	terminals: action part			variable: goto part	
	a	b	$	A	S
0	S_3	S_4		2	1
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		

r_s → reduced → Blank entries in table corresponding to the error.
s_s → shift

Example: (Use LR(0) parsing table to parsing)

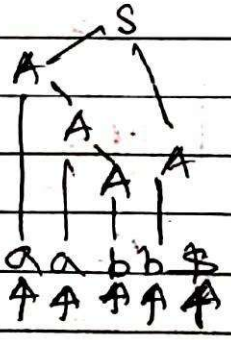
$S' \rightarrow S$

$S \rightarrow AA$ ①

$A \rightarrow aA$ / b ② ③

generate aabb$ using table.

→



aabb

Stack: 0 a 3 a 3 b 4 A 6 A 8 A 2 b 4 A 5 S 1

(accepted).

$A \rightarrow b$ (reduced)

$A \rightarrow aA$ (reduced)

$A \rightarrow aA$ (reduced)

$A \rightarrow b$ "

$A \rightarrow AA$ "

• SLR(1) parsing :-

→ The main difference between LR(0) and SLR(1) is Reduce moves.

→ SLR(1) more powerful than LR(0).

$S' \rightarrow S$

$S \rightarrow AA$ ①
$A \rightarrow aA / b$
② ③

→ grammar.

→ SLR(1) has less number of reduce moves compare to LR(0). (Detect the error faster)

SLR(1) parsing table: (Make the same way like LR(0))

	action			goto	
	a	b	$	A	S
0	S_3	S_4		2	1
1			accept		
2	S_3	S_4		5	
3	S_3	S_4		6	
4	r_3	r_3	r_3		
5			r_1		
6	r_2	r_2	r_2		

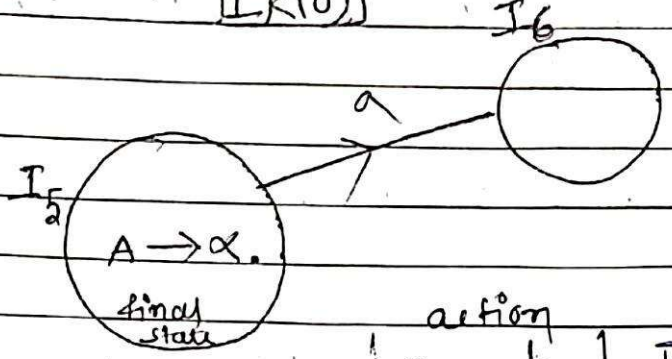
→ Just change in reduce move.

Make final state = (find follow of left L.H.S)

$(S' \rightarrow S.)^{I_1}$ $(S \rightarrow AA.)^{I_5}$ $(A \rightarrow aA.)^{I_6}$ $(A \rightarrow b.)^{I_4}$

$. (a, b, $)

LR(0)

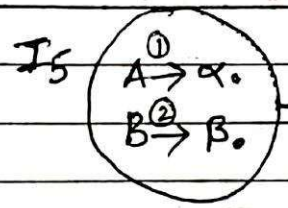


	action		
	a	b	
5	S_6/r_1		r_1

SR conflict

→ called shift reduce conflict.

~~→~~ all grammar are not suitable for LR(0) parsing.



→ one final state contain 2 production.

	a	b	c	
5	r_1/r_2	r_1/r_2	r_1/r_2	

RR conflict

→ called RR conflict.

→ a grammar can't be LR(0) when it is RR conflict and SR conflict.

→ A grammar can't be ~~SLR(1) when it is~~ RR conf

- chances of getting RR, SR for SLR(1) parsing =

(SR)

when follow(A) = {a}.

(RR)

When follow(A) ∩ follow(B) ≠ ∅.

Find out

- Given grammar are LR(0), SLR(1) or not =

(1)

S → dA / aB
A → bA / c
B → bB / c

given grammar are - (i) LL(1) or not.
(ii) LR(0) or not.
(iii) SLR(1) or not.

→

LL(1) or not :

	first	follow
S → dA / aB	{d, a}.	{$}
A → bA / c	{b, c}.	{$}
B → bB / c	{b, c}	{$}

	a	b	c	d	$
S	S → aB			S → dA	
A		A → bA	A → c		
B		B → bB	B → c		

→ each cell contain only '1' production so this grammar is LL(1).

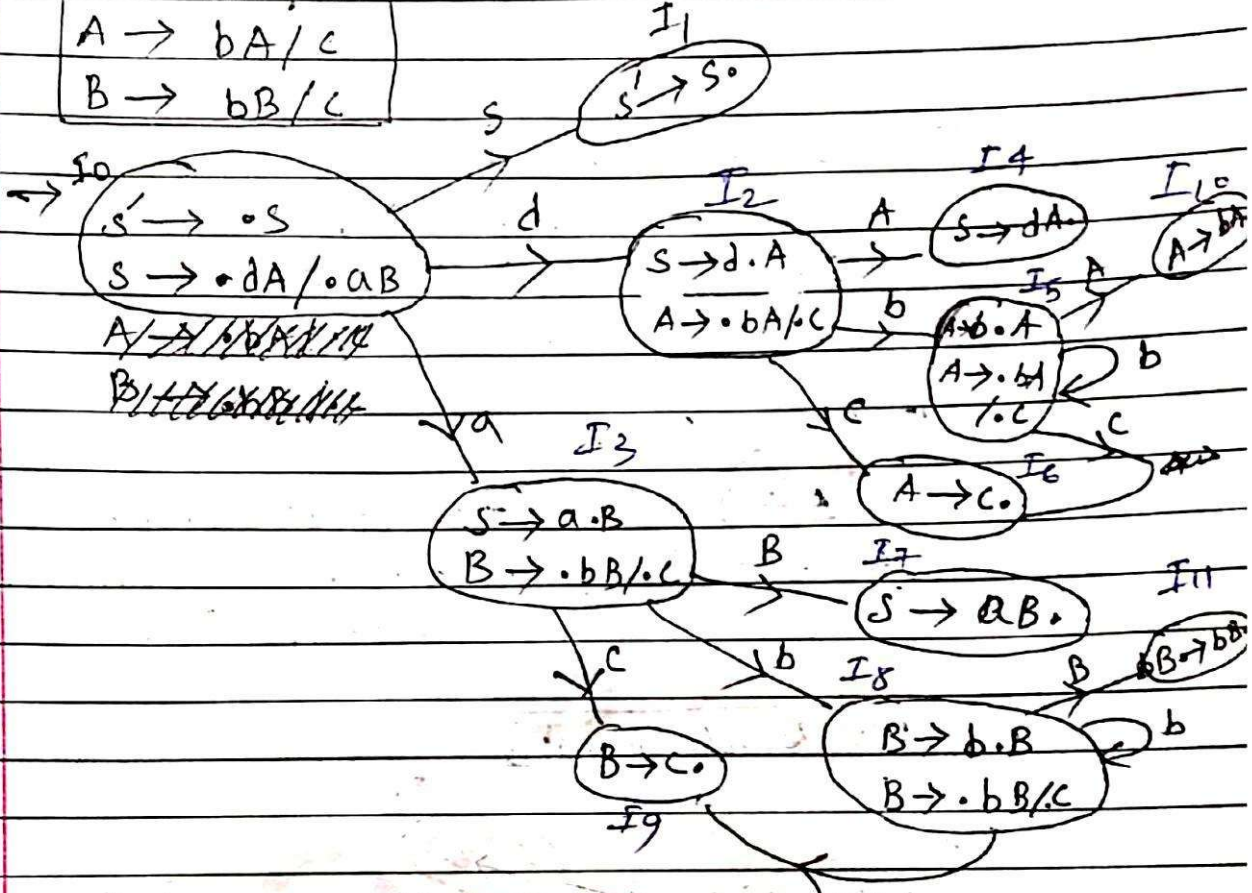
LR(0), SLR(1) or not

S' → S (added)

S → dA / aB

A → bA / c

B → bB / c



→ here, no SR (Shift Reduce) conflict.
(Because each final state Doesn't have any shift)

→ no RR Conflict (Because each final state contain only one ~~RR pro~~ ~~reduce~~ production)

So, this grammar is LR(0).

→ When a grammar is LR(0), then this grammar also SLR(1).

(2)

$S \to A/a$
$A \to a$

check, given grammar are suitable for
- LL(1) or not
- LR(0) " X
- SLR(1) " X

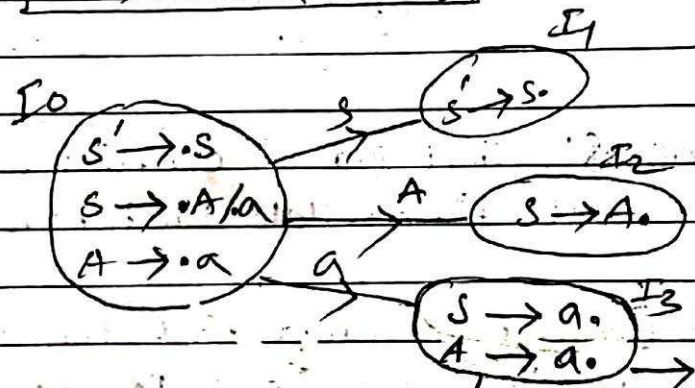
→ Check LL(1) or not

	First	Follow
$S \to A/a$	{a}	{$}
$A \to a$	{a}	{$}

	a	$
S	$S \to A$, $S \to a$	
A	$A \to a$.	

→ not suitable for LL(1) parser because cell contain '2' production.

LR(0) & SLR or not



→ one final state contain '2' reduce move so it is not suitable for LR(0) parsers.

follow of S and A is common '$', so this grammar are not suitable for SLR(0) parsers.

(3) Check given grammar are suitable for (i) LL(1) or not
(ii) LR(0) "
(iii) SLR(1) "

$S \rightarrow (L) / a$
$L \rightarrow L, S / S$

→

LL(1) or not

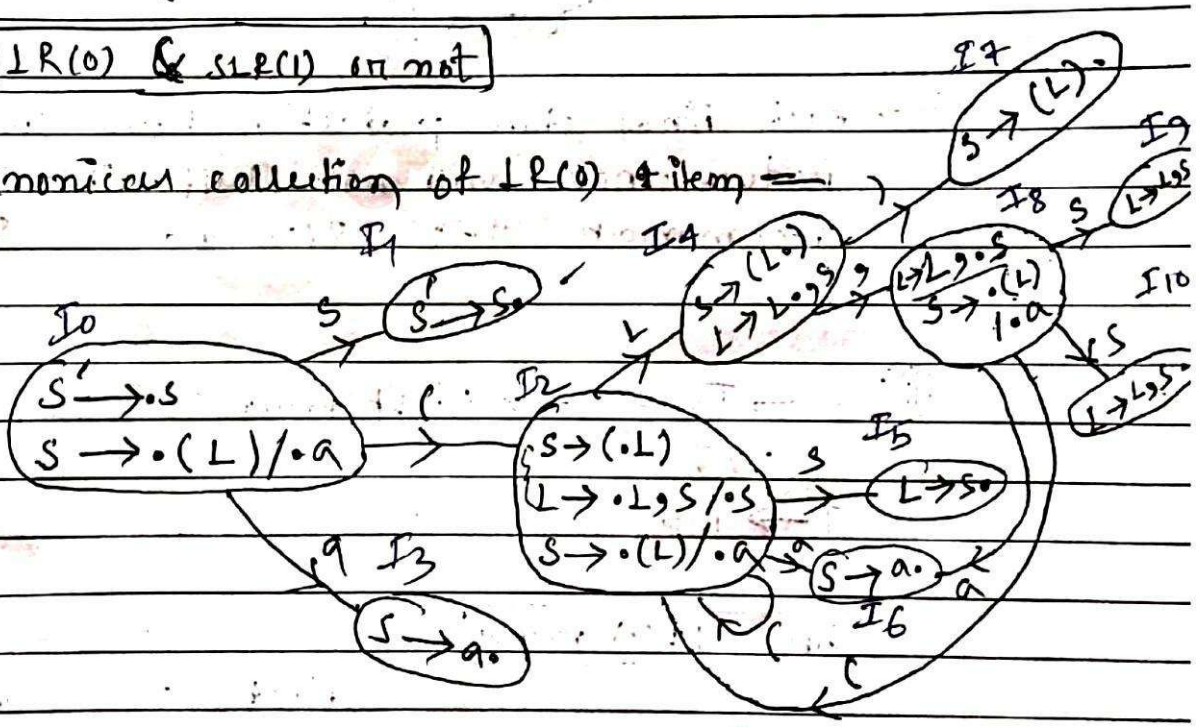
	First	Follow
$S \rightarrow (L) / a$	{(, a}	{$}
$L \rightarrow L, S / S$	{(, a}	{, ,)}

	(	)	a	,	$
S	S→(L)		S→a		
L	L→L,S		L→S		

This grammar are left recursive grammar, any left recursive can't be LL(1).

LR(0) & SLR(1) or not

canonical collection of LR(0) item →



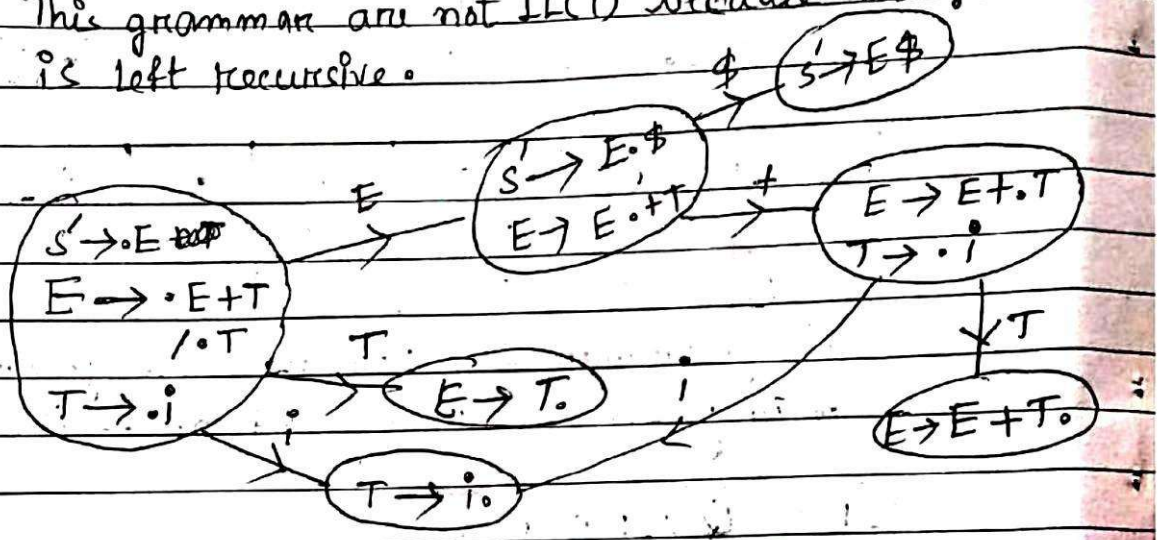
→ each final state contain only one production and it doesn't have any shift . so, this grammar is suitable for LR(0).

→ SLR(1) also (because any are suitable for LR(0) and suitable for SLR(1))

(4) Check given grammars are suitable for LL(1), LR(0), SLR(1) or not.

(i) $E \to E+T$ → left recursive grammar.
$/T$
$T \to i$

⇒ This grammar are not LL(1) because this grammar is left recursive.



⇒ each final state production have single reduce move and not shift. therefore given grammars are LR(0) and also SLR(0).

(ii) $E \to T+E$ ① → Right recursive grammar.
$/T$ ②
$T \to i$ ③

Check LL(1) or not

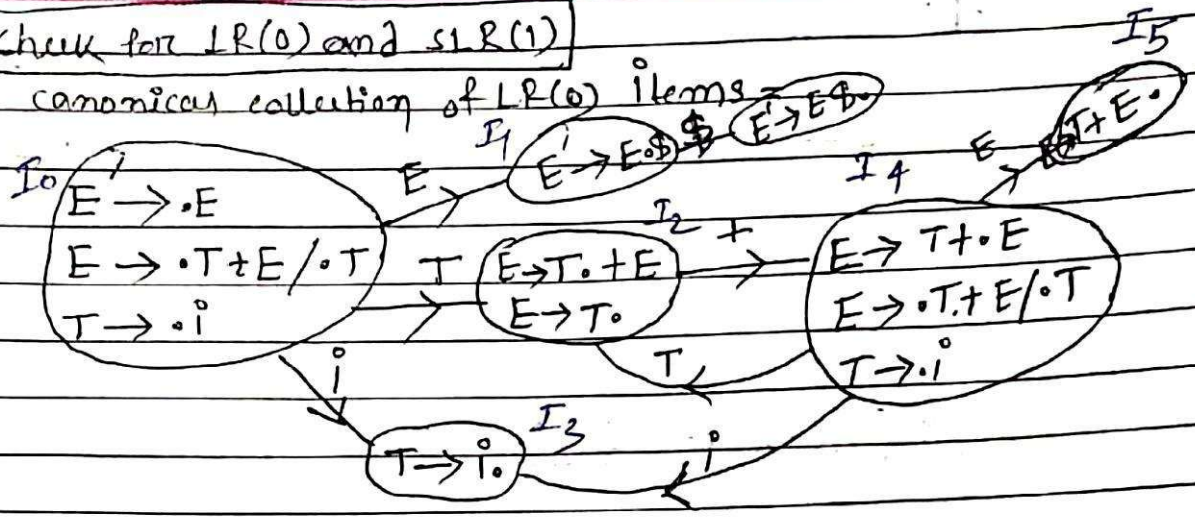
	first	follow
$E \to T+E/T$	{i}	{$}
$T \to i$	{i}	{+, $}

	+	i	$
F		$E \to E+E$ / $E \to T$	
T		$T \to i$	

→ not LL(1).

Check for LR(0) and SLR(1)

canonical collection of LR(0) items =



→ State I_2 contain both reduce move and shift move, means SR conflict. therefore this grammar is not SLR(0).

✓ → But this grammar is SLR(1).

LR(0) table

	i	+	$	E	T	
0	S_3			1	2	
1			accept			→ SR conflict
2		r_2/s_4				
3	r_3	r_3	r_3			
4	S_3			5	2	
5	r_1	r_1	r_1			
6						

SLR(1) table

	i	+	$	E	T	
0	S_3			1	2	follow(E) = {$}
1			accept			" (T) = {+, $}
2		S_4	r_2			
3		r_3	r_3			
4	S_3			5	2	→ no SR conflict
5			r_1			

(1) check given grammar are suitable for (i) LL(1). (ii) LR(0). (iii) SLR(1).

$S \rightarrow dA$
$/aB$
$A \rightarrow bA/c$
$B \rightarrow bB/c$

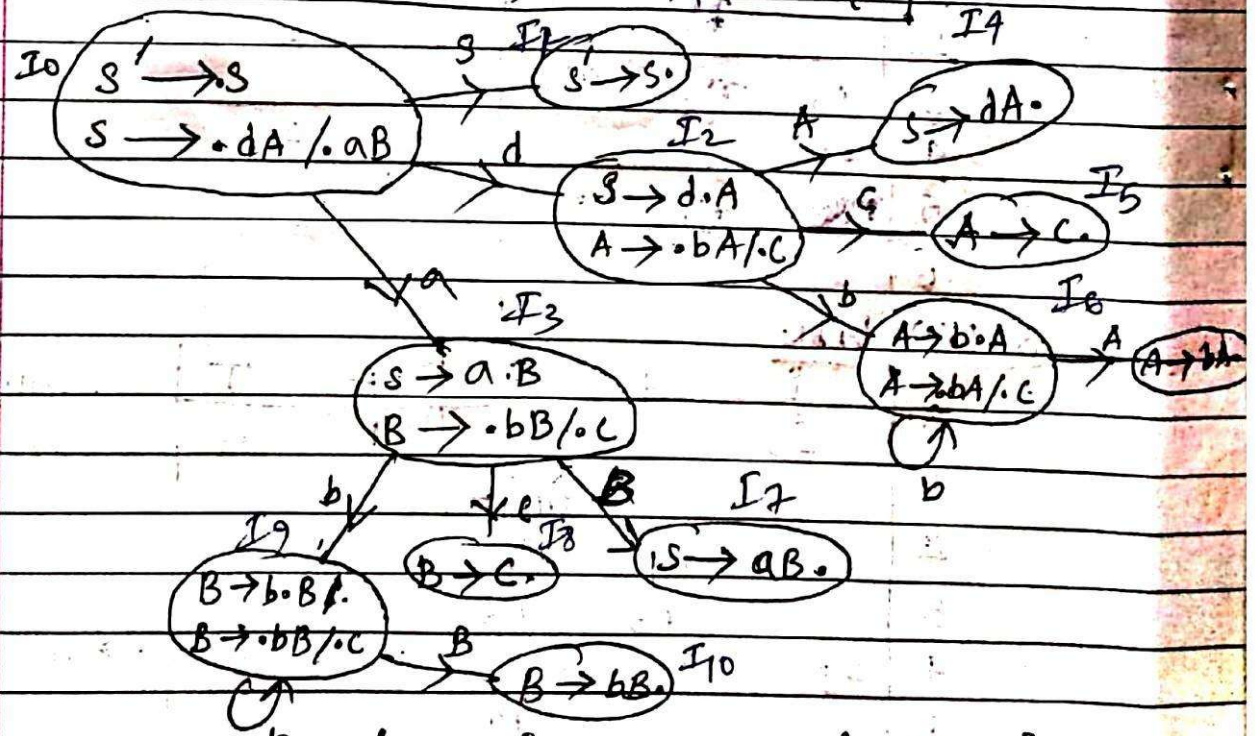
$\Rightarrow$ Check for LL(1)

	first	follow
$S \rightarrow dA/aB$	$\{d,a\}$	$\{\$\}$
$A \rightarrow bA/c$	$\{b,c\}$	$\{\$\}$
$B \rightarrow bB/c$	$\{b,c\}$	$\{\$\}$

	a	b	c	d	$
S	$S \rightarrow aB$			$S \rightarrow dA$	
A		$A \rightarrow bA$	$A \rightarrow c$		
B		$B \rightarrow bB$	$B \rightarrow c$		

$\rightarrow$ given grammar is LL(1).

Check whether LR(0) & SLR, or not



(canonical collection of LR(0) items)

→ Each final state Doesn't have ~~only~~ SR ~~or~~ and RR conflict, so grammar is LR(0).

→ If a grammar is suitable for LR(0) then it is also suitable for SLR(0).

(5) Check given grammar are suitable for ~~LL(1)~~, LR(0) SLR(1) parsers.

		(first)
$E \to E+T$ ① / T ②		$\{a, b\}$
$T \to TF$ ③ / F ④		$\{a, b\}$
$F \to F*$ ⑤ / a ⑥ / b ⑦		$\{a, b\}$

⇒ Check for LL(1) parsers

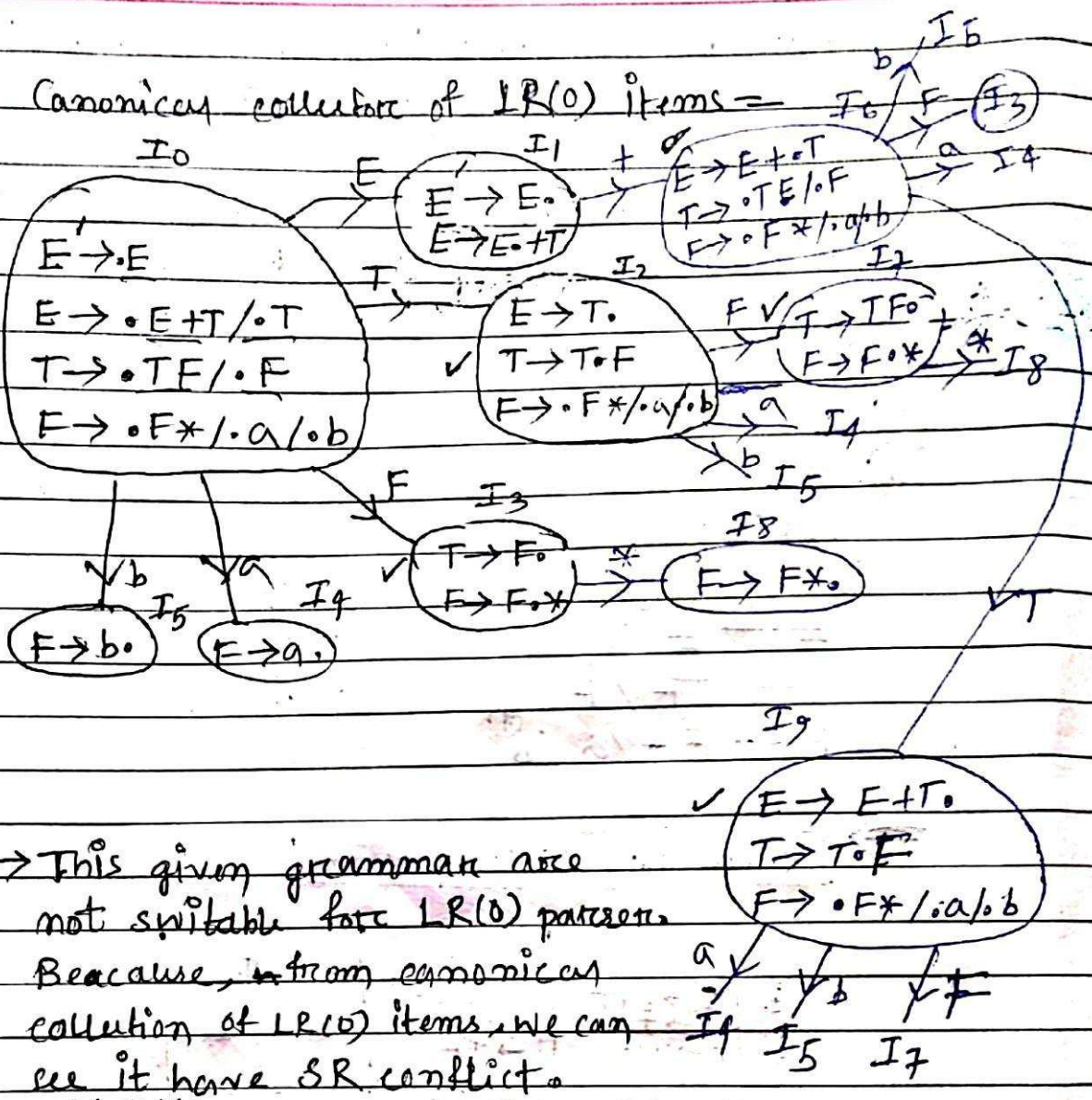
	a	b	$
E	E→E+T E→T	E→E+T E→T	
T	T→TF T→F	T→TF T→F	
F	F→F* F→a	F→F* F→b	

→ left recursive grammar can't be suitable for LL(1) parsers.

Check for LR(0) and SLR(1)

→ next page.

Canonical collection of LR(0) items =



→ This given grammar are not suitable for LR(0) parsers. Because, from canonical collection of LR(0) items, we can see it have SR conflict. (shift and reduce)

Follow(E) = {+, $}
Follow(T) = {+, $, a, b}
Follow(F) = {*, +, $, a, b}

SLR(1) table

	a	b	+	*	$	E	T	F
2	S_4	S_5	r_2		r_2	(not need to find RR, SR conflict)		
3	r_4	r_4	r_4	S_8	r_4			
7	r_3	r_3	r_3	S_8	r_3			
9	S_4	S_5	r_1		r_1			

→ not SR / RR conflict in SLR(1) table so, given grammar are suitable for SLR(1) parsers.

	first	follow
S →	{a, b}	{$}
A →	{ϵ}	{a, b}
B →	{ϵ}	{b, a}

⑥ Given grammar are suitable or not =

S → A a A b / B b B a {a, b}

A → ϵ ③ {ϵ}

B → ϵ ④ {ϵ}

(I) LL(1) ✓

(II) LR(0) ✗

(III) SLR(1) ✗

LL(1) :-

⇒

	a	b	$
S	S → AaAb	S → BbBa	
A	A → ϵ	A → ϵ	
B	B → ϵ	B → ϵ	

Note :-
A → .ϵ }
A → ϵ. } A → . same

→ each cell have only one production. so given grammar are accepted by LL(1) parsers.

Check for LR(0) and SLR(1) :-

I0

S → .S
S → .AaAb / .BbBa
A → .
B → .

→ This grammar not LR(0), because one single state containing two reduce move.

SLR(1) parsing table

	a	b	$
0	r3/r4	r3/r4	

→ Reduce-Reduce conflict, so this grammar are not accepted for SLR(1) parser.

⑦ Check whether given grammar are LL(1) (X), LR(0) (X), SLR(1).

$S \rightarrow AS$ ① / b ②

$A \rightarrow SA$ ③ / a ④

$\{a, b\}$

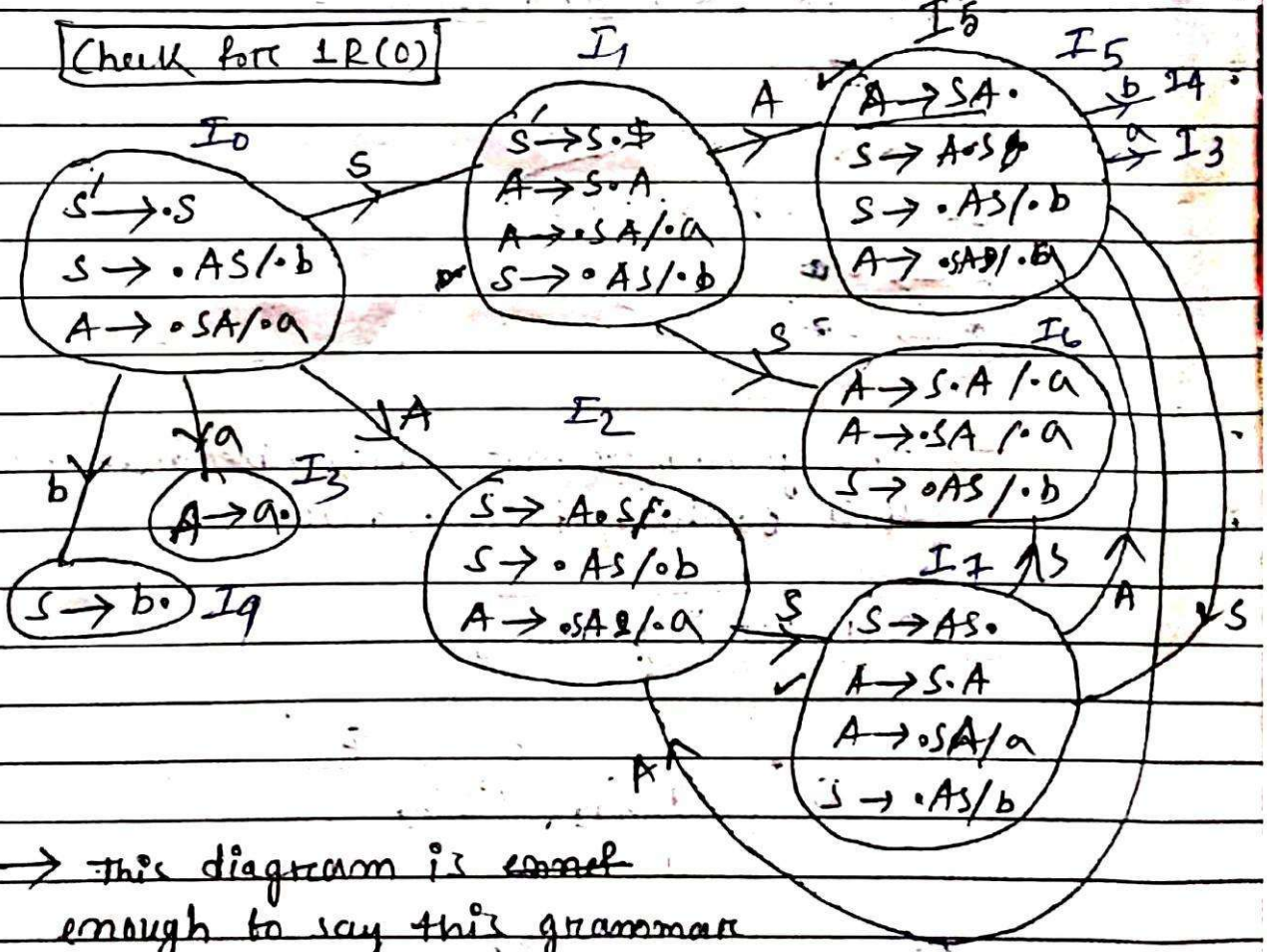
$\{a, b\}$

$\Rightarrow$ **Check for LL(1)**

	a	b	$
S	S → AS	S → AS S → b	
A	A → SA A → a	S → SA	

→ Where more than two production in one cell, so, this grammar can't be LL(1) grammar.

Check for LR(0)



→ this diagram is enough to say this grammar are not LR(0), because (SR conflict).

Check for SLR(1)

SLR(1) parsing table :-

follow(A) = {a, b}
" (S) = {$, a, b}

	a	b	$
5	r3/s3	r3/s4	

→ SR conflict.

→ grammar not accepted by SLR(1) parser.

Given grammar are ambiguous, so that this grammar are not accepted by LL(1), LR(0) and or SLR(1) parsers.

(8) Check given grammar are ~~LL(1)~~, ~~LR(0)~~ and ~~SLR(1)~~ or not.

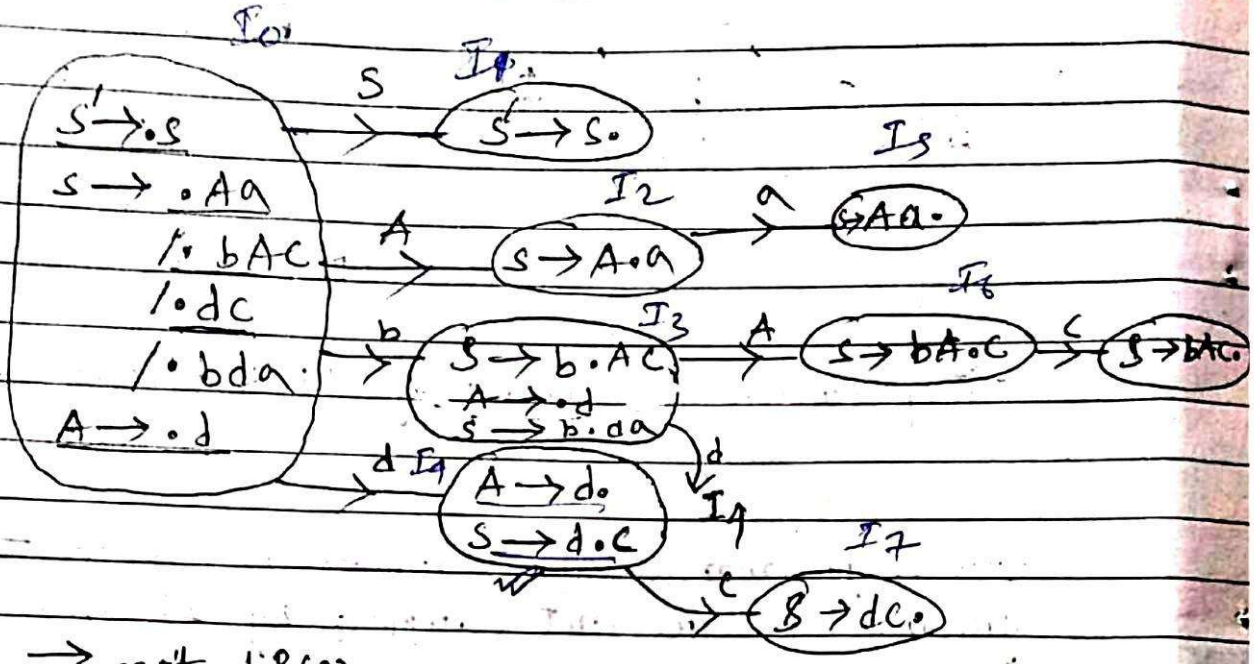
	first
S → Aa ① / bAC ② / dC ③ / bda ④	{d, b}
A → d ⑤	{d}

⇒ Check for LL(1) parser

	a	b	c	d	$
S		S→bAC, S→bda		S→Aa, S→dC	
A				A→d	

→ not LL(1).

Check for LR(0) =



→ not LR(0).
(because SR conflict)

follow (A) = {a, c}

Check for SLR(1) =

SLR(1) table =

	a	b	c	d	$
I_4	r_5		r_5 / S_7		

↳ SR conflict

→ not accepted by SLR(1) parser.

- **CLR(1) and LALR(1) parsers :**

CLR(1) LALR(1)

LR(1) item = LR(0) item + lookahead.

- **Canonical collection of LR(1) items –**

given grammar,

$S \rightarrow AA$

$A \rightarrow aA/b$

(lookahead)

$A \rightarrow \alpha.B\beta, a/b$

$B \rightarrow \alpha.\gamma, c/d$

first of β

$A \rightarrow \alpha.B, a/b$

$B \rightarrow .\gamma, a/b$

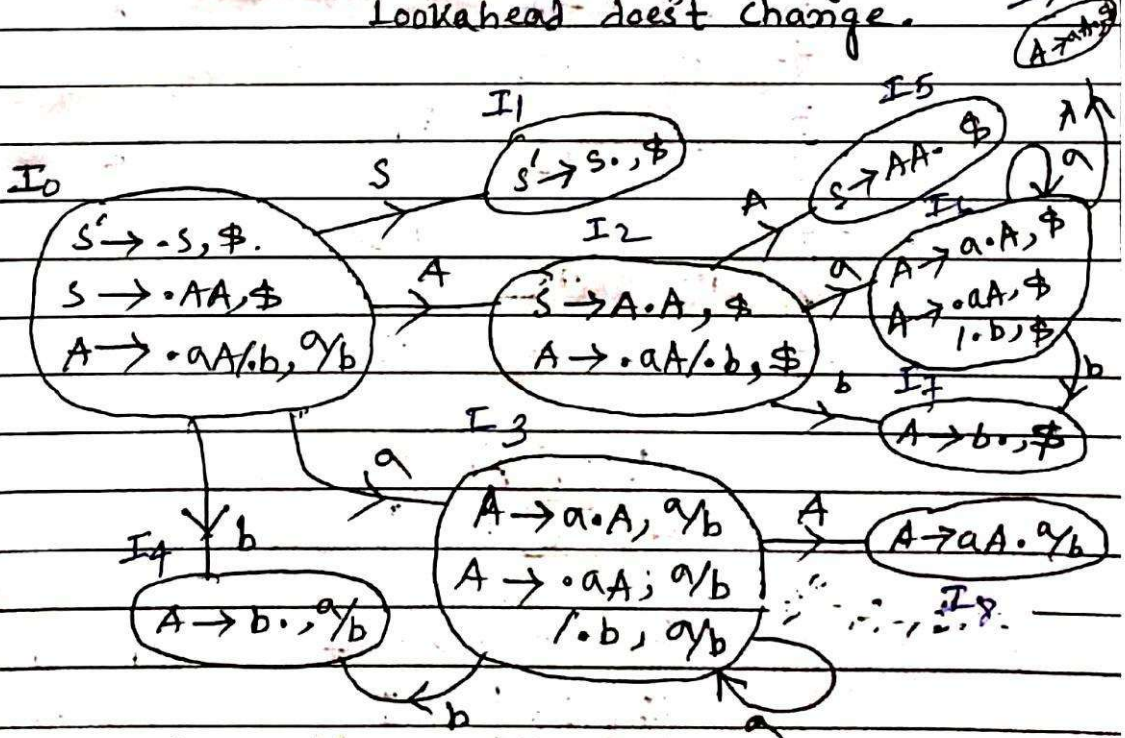
I_0

$S' \rightarrow .S, \$$

$S \rightarrow .AA, \$$

$A \rightarrow .aA/.b, a/b$

→ after transition of any symbole lookahead doesn't change.



→ state may increase in LR(1) compare to LR(0) & parser.

→ number of state increases because of Lookahead.
→ Number of Reduce move decrease because of Lookahead.

→ In diagram I_3 and I_6 both have same LR(0) items but different Lookahead.

→ also I_4 and I_7 both have same LR(0) items but diff Lookahead.

→ I_8 and I_9 " " " " diff Lookahead.

• CLR(1) parsing table (for previous example)

	a	b	$	S	A
0	S_3	S_4			2
1					
2	S_6	S_7			5
3	S_3	S_7			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		

(Merging)

$I_3, I_6 \Rightarrow I_{36}$

$I_4, I_7 \Rightarrow I_{47}$

$I_8, I_9 \Rightarrow I_{89}$

$S \rightarrow AA$ (1), $
$A \rightarrow aA$ (2)
$A \rightarrow b$ (3), a/b

(reduce move according to lookahead)

no. of state LR(0) = SLR(1) = LALR(1) ≤ CLR(1)

- LALR(1) parsing table: ~~by~~

→ LALR(1) table maked from CLR(1) table, by marging '6' state into '3' states.

	a	b	$	S	A
0	S_{36}	S_{47}			2
1					
2	S_{36}	S_{47}			5
(36)	S_{36}	S_{47}			89
(47)	r_3	r_3			
5			r_1		
(36)	S_{36}	S_{47}			89
(47)			r_3		
(89)	r_2	r_2			
(89)			r_2		

(compress similar states)

New table

	a	b	$	S	A
0	S_{36}	S_{47}			2
1					
2	S_{36}	S_{47}			5
36	S_{36}	S_{47}			89
47	r_3	r_3	r_3		
5			r_1		
89	r_2	r_2	r_2		

(no of state, LR(0) = SLR(1) = LALR(1) = < CLR(1))
(7) (7) (7) (10)

- SR conflict and RR conflict =

LR(1) items: $A \to \alpha . a\beta, c/d$ (a) ; $B \to \gamma ., a/\$$ → SR conflict

(single state contain shift and reduce move both)

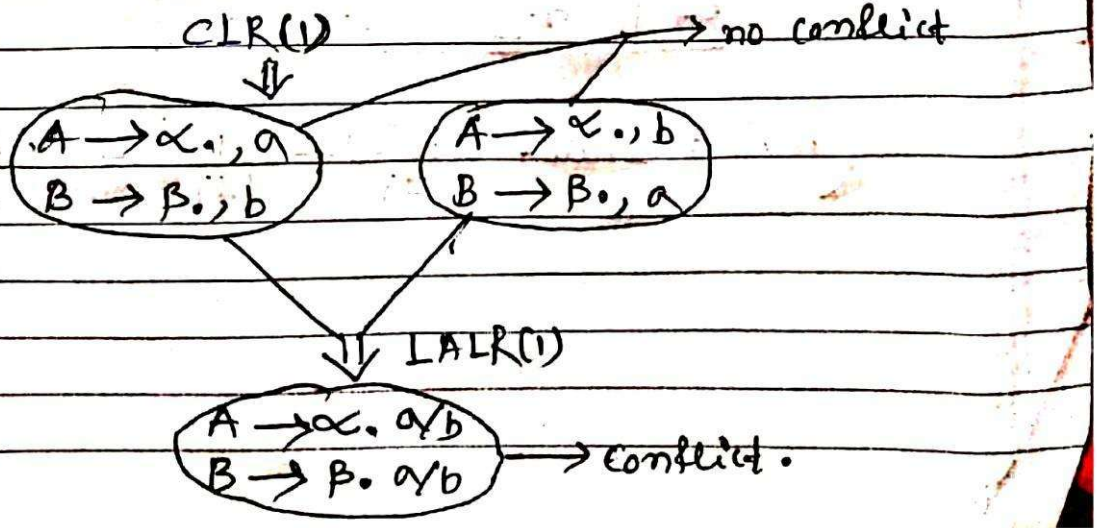
$A \to \alpha ., a$
$B \to \alpha ., a$ → RR conflict

(single state contain Reduce - Reduce move).

→ If 'I' grammar are not ~~write~~ accepted by CLR(1) parser, that can't be accepted by LALR(1) parser. grammar

→ When One grammar are accepted by CLR(1) parser, that grammar may or may' not be accepted by LALR(1) parser.

ex =

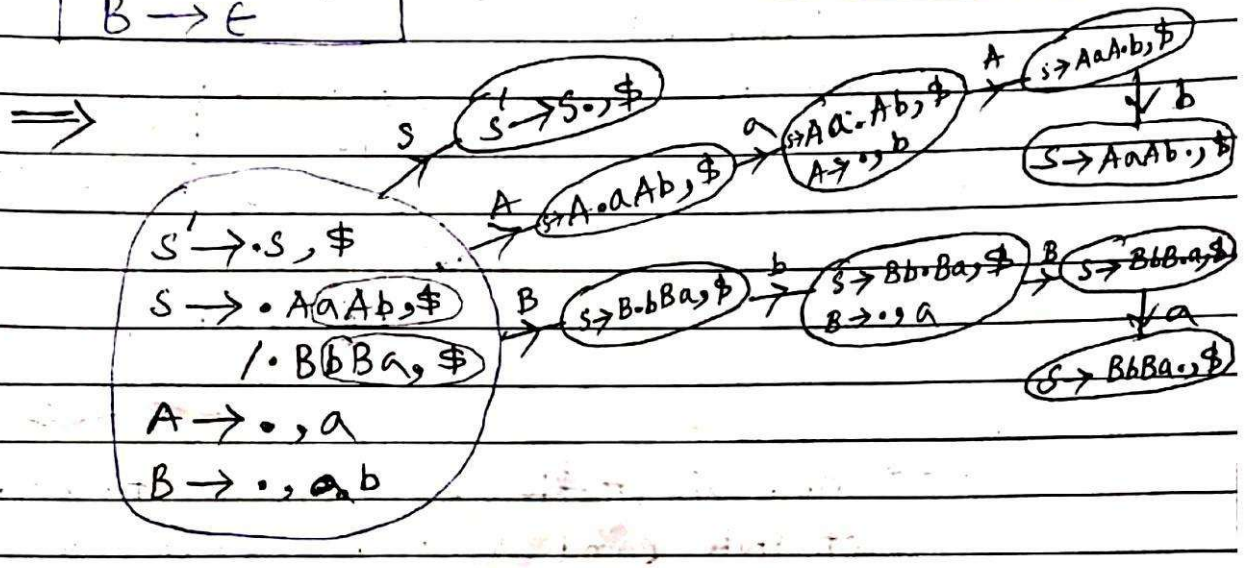


Example =

(1) Check given grammar are accepted by — (i) CLR(1) (ii) LALR(1).

S → AaAb
/ BbBa
A → ϵ
B → ϵ

A → .ϵ → ϵ. → •



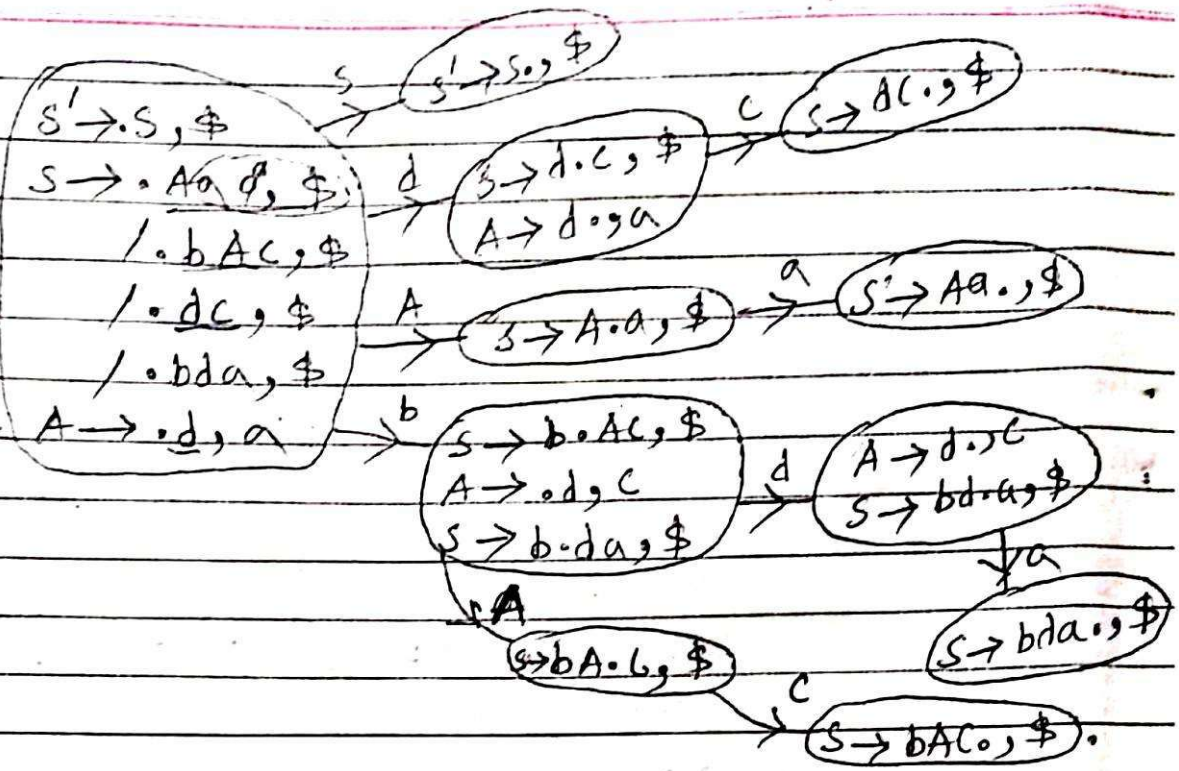
Canonical Collection of LR(1) items, no conflict occure. So, given grammar accepted by CLR(1) parser. And ~~m~~ state marging not possible - so that grammar also accepted by LALR(1) parser.

(2) Given grammar are accepted by — ✓ LR(1) parser.
✗ LALR(1) parser.

S → Aa
/ bAc
/ dc
/ bda
A → d

⇒ canonical collection of LR(1) items —

→ next page.



→ no, SR ~~or RR~~ and RR conflict. And number of state can't be reduce ~~by~~ (merge)

→ so, given grammar are accepted by CLR(1) or LALR(1) both, parser.

(3) Check given grammar are accepted by — ~~CLR(1)~~ and, ~~LALR(1)~~ parser, ~~LL(1)~~, ~~LR(0)~~, ~~SLR(1)~~.

$S \to Aa$ $\{d, b\}$
$\quad / bAa$
$\quad / Bc$
$\quad / bBa$
$A \to d$ $\{d\}$
$B \to d$ $\{d\}$

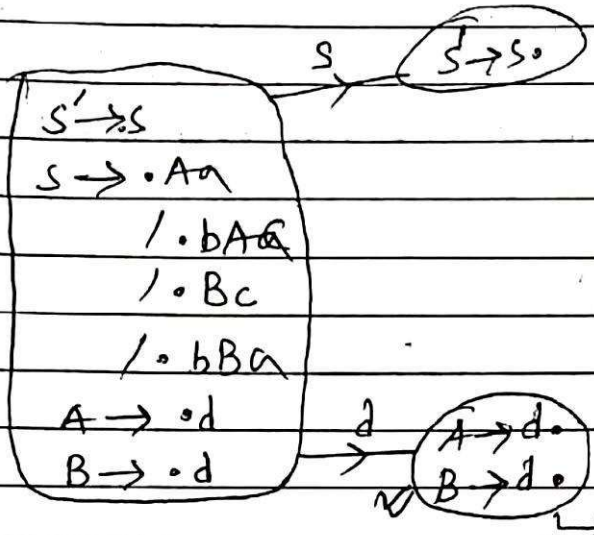
⇒ LL(1):

	a	b	c	d	$
S		S→bAa S→bBa		S→Aa S→Bc	
A				A→d	
B				B→d	

— not accepted by LL(1) parser.

Check for LR(0) = SLR(1) also.

Canonical collection of LR(0) items :—
(not required to draw full diagram) ~~to prove.~~



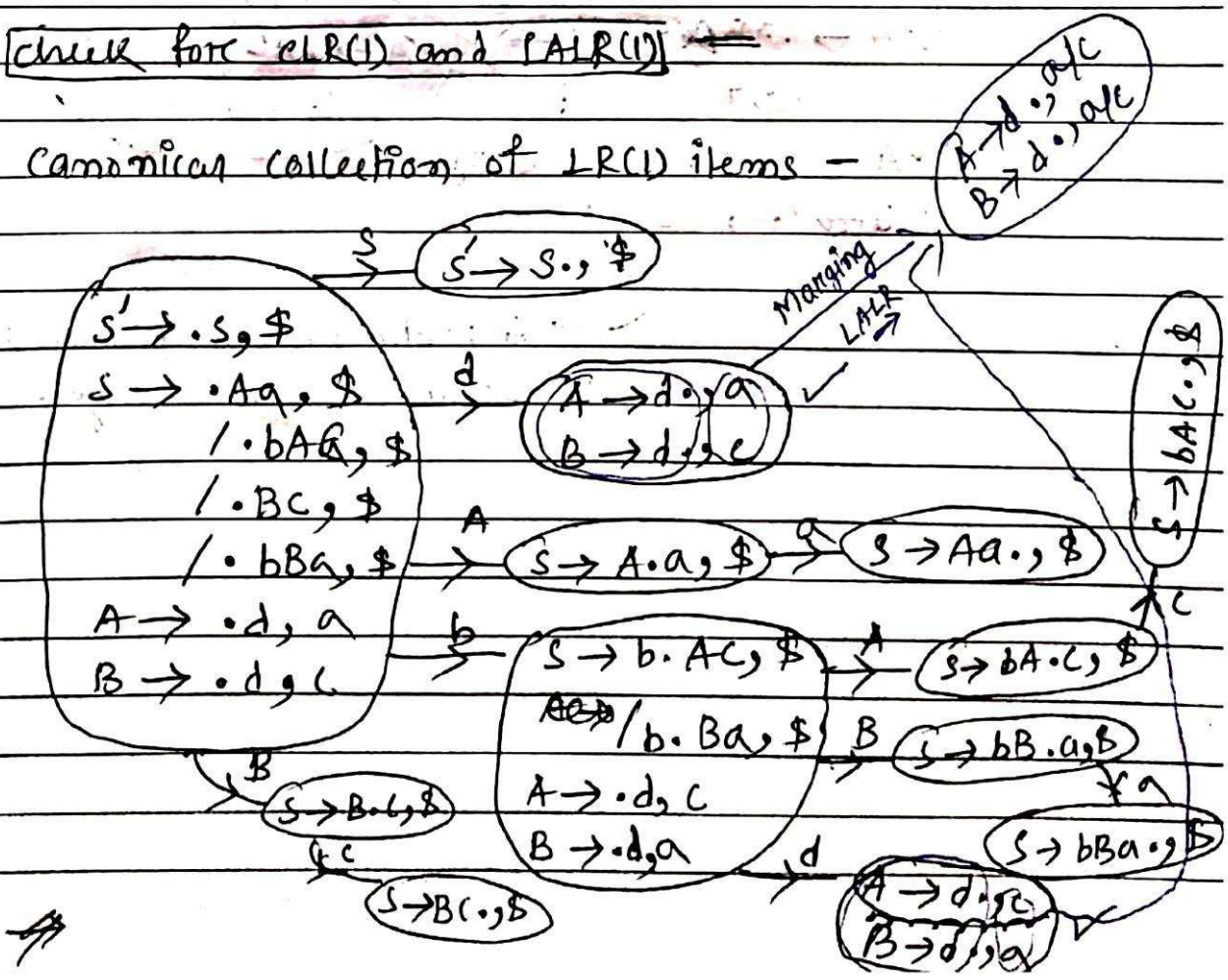
follow(A) = {a, c}
follow(B) = {a, c}

follow

→ not accepted by LR(0). (because RR conflict)
→ also not accepted by SLR(1). (RR conflict).

Check for CLR(1) and LALR(1) —

Canonical collection of LR(1) items —



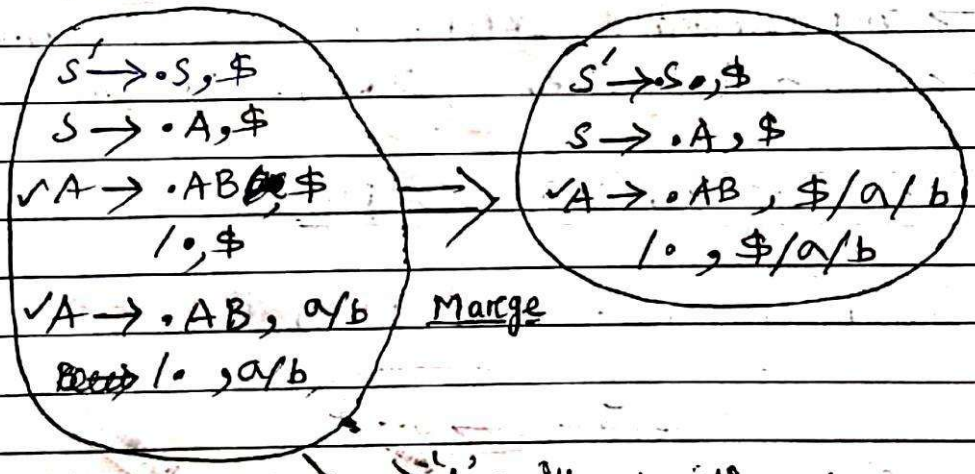
→ given grammar accepted by CLR(1) parsers. (No RR and SR conflict)

→ given grammar are not accepted by LALR(1) (because after marging two state in one state then new state contain ~~two~~ RR conflict.

④ example — ①

S → A
A → AB/ϵ
B → aB/b

Closure



→ 'A' written two time because there Lookahead can change.

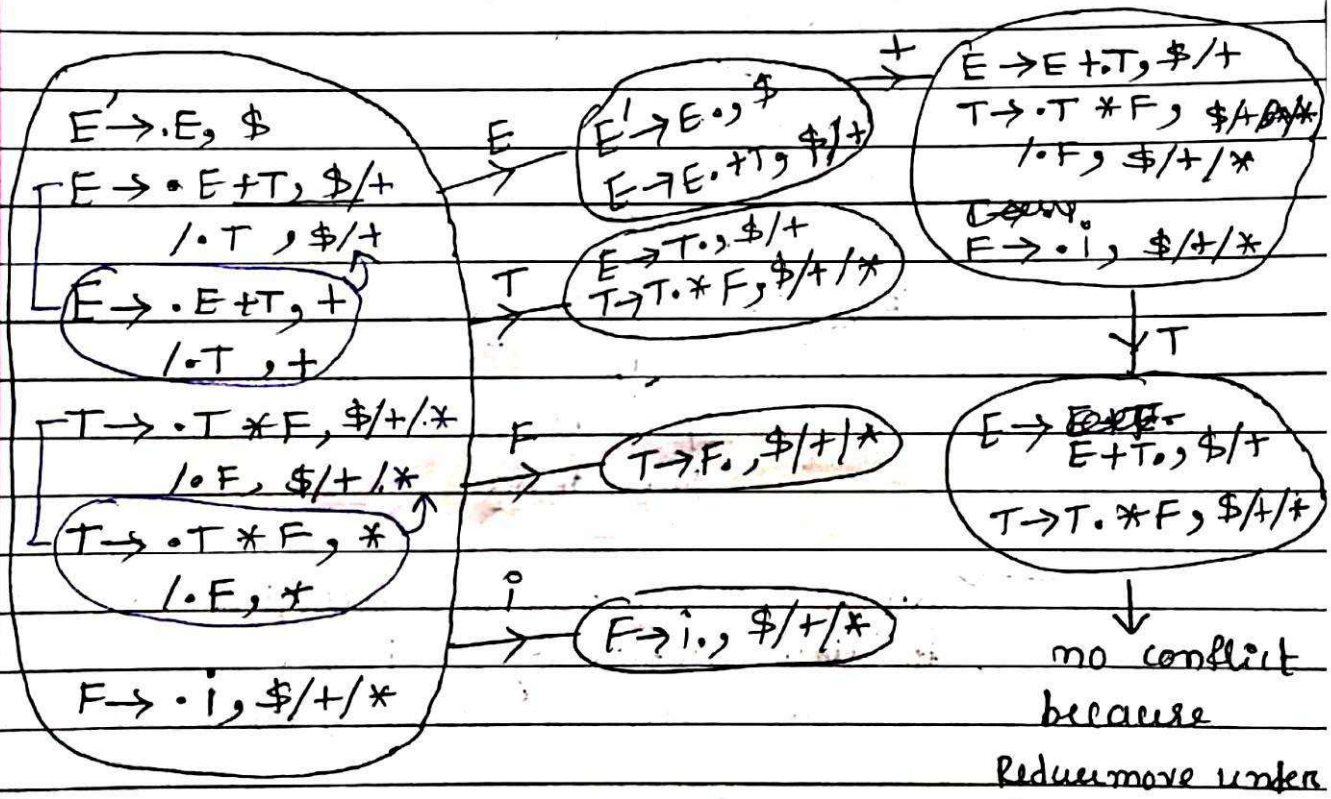
example — (2)

$E \rightarrow E+T \mid T$

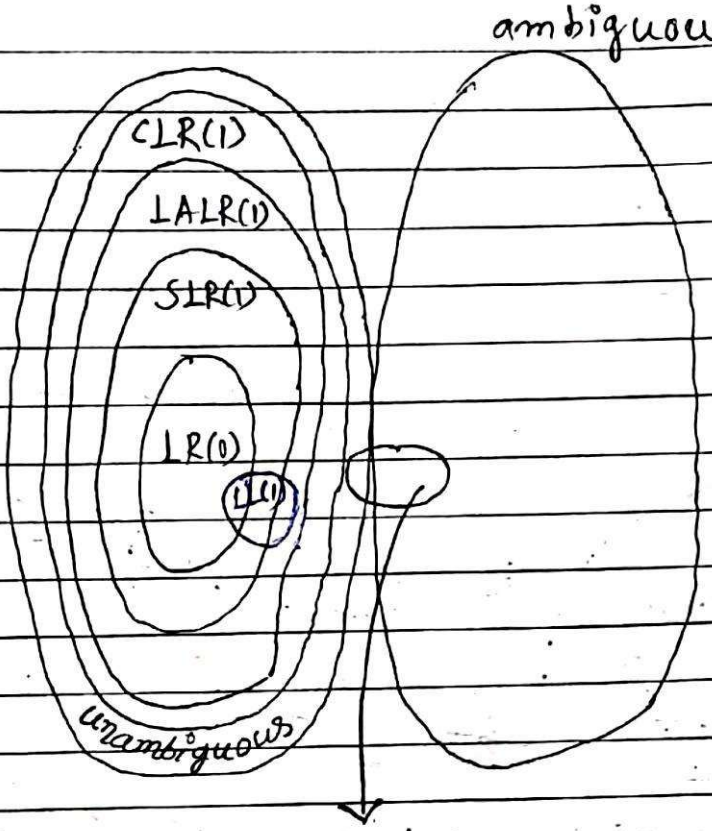
$T \rightarrow T*F \mid F$

$F \rightarrow i$

Canonical collection of LR(1) items —



- Comparison of all parsers —



→ when a grammar are accepted by LL(1), then it also accepted by LALR(1).
(because LL(1) are completly under of LALR(1)).

- Some questions —

Q-1 Consider SLR(1) and LALR(1) table for a CFG —
✗ (a) Goto of both tables may be different.
✓ (b) shift entries are identical in both tables.
✓ (c) Reduce entries in table may be different.
✓ (d) error entries in tables may be different.
(Blank entries)

Q-2] n_1, n_2 and n_3 are number of state of SLR(1), LALR(1) and CLR(1) table respectively. What is the relationship between n_1, n_2 and n_3.

→ n_1 = SLR(1)
n_2 = LALR(1)
n_3 = CLR(1)

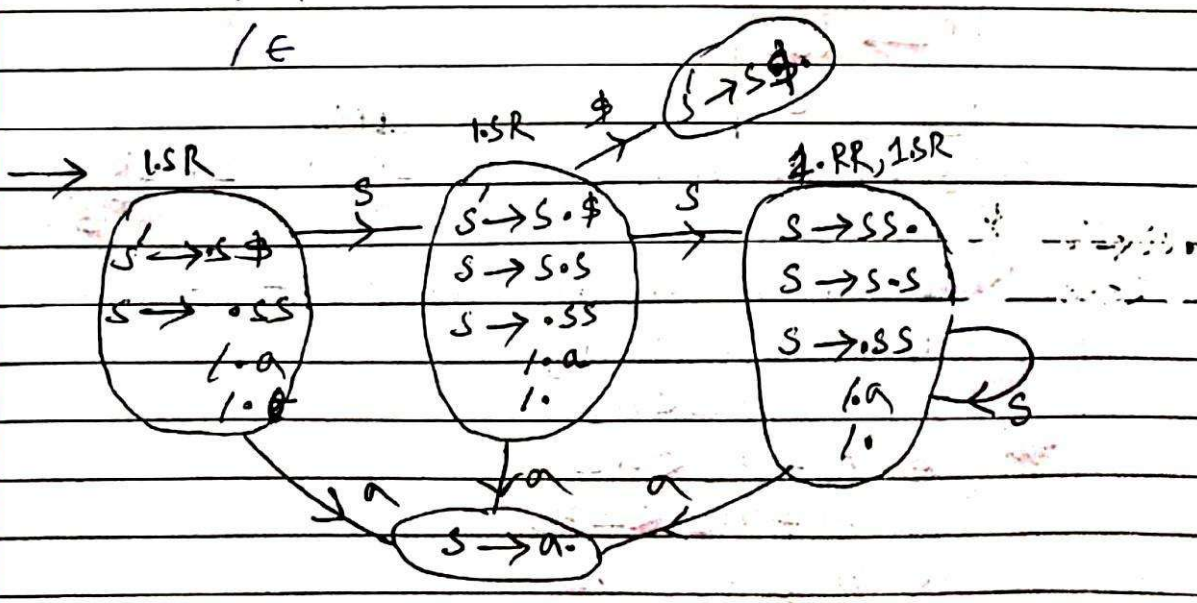
relation –
$n_1 = n_2 \leq n_3$

Q-3] $S \rightarrow CC$
$C \rightarrow eC/d$ → this grammar are accepted by all grammars.

~~a)~~ LL(1)
b) SLR(1) but not LL(1)
c) LALR(1) but not SLR(1)
d) CLR(1) but not LALR(1).

Q-4] Find the number of SR and RR conflicts in dfa with LR(0) items =

$S \rightarrow SS$
$/a$
$/\epsilon$



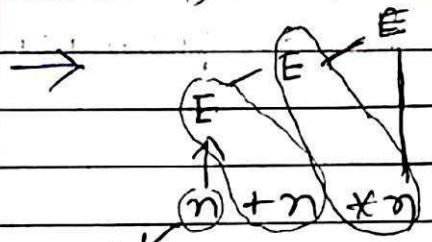
→ here 3 SR move and 1 RR move.

Q-5] given-

$E \to E+n$
$/ E*n$
$/ n$

for the string $n+n*n$, the handles in right sentential form of the reduction are-

a) n, $E+n$, $E+n*n$
b) n, $E+n$, $E+E*n$
c) n, $n+n$, $n+n*n$
d) n; $E+n$, $E*n$.



n, $E+n$, $E*n$

OR

$E \Rightarrow (E*n)$
$\Rightarrow (E+n)*n$
$\Rightarrow (n)+n*n$

Q-6] given grammar =

$S \to (S,)$
$/ a$

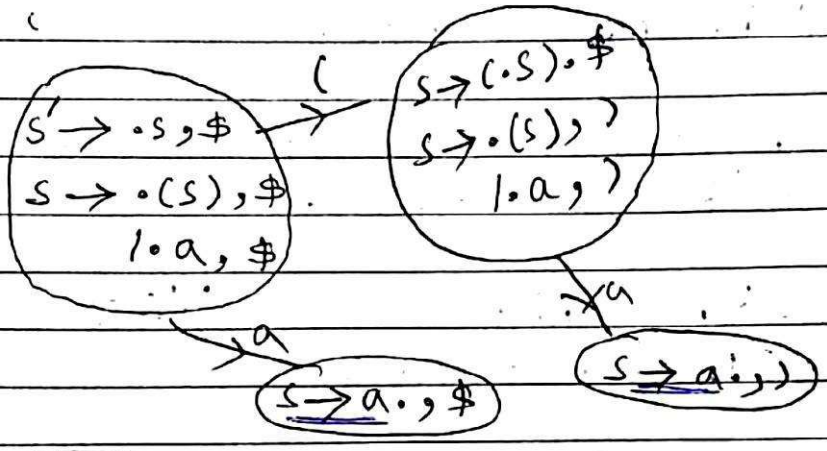
relation between SLR(1), LR(1), LALR(1)
n_1 n_2 n_3
(CLR(1))

here n_1, n_2, n_3 are no. of state of SLR(1), LR(1), LALR(1) respectively.

(a) $n_1 < n_2 < n_3$
(b) $n_1 = n_3 < n_2$
(c) $n_1 = n_2 \neq n_3$
(d) $n_1 \geq n_3 \geq n_2$

$\Rightarrow (n_1 = n_3 \leq n_2)$

canonical collection for LR(1) items:-



→ $n_1 = n_3$ always

→ of LR(1) an no. of state more compare to LR(0).

$n_1 = n_3 < n_2$

- Ambiguous and unambiguous =

Ambiguous

E → E+E
/E * E
/ id

Unambiguous

E → E+T / T
T → T*F / F
F → id.
(left associative)

Adv:

(i) no. of production less.
(ii) natural and easy to understand.
(iii) we can change associativity and precedence.

E
↓
id (1-step)

E
T
F
id (3-step).

→ ambiguous grammar not used in parsing expect operator precedence because of conflict. will be occur more.

→ ambiguous can used, if you can solve the conflicts meaningfully.

SR → shift
RR → first Reduce

} YACC = use table of LALR(1)

r_1/r_2