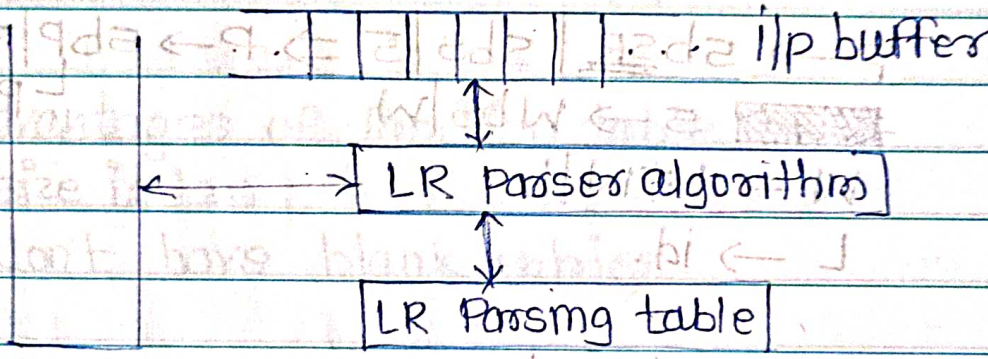
2] LR Parsers :- (Bottom Up Parser)

are of 4 types.

1] LR(0)
2] SLR(1) (Simple LR)
3] LALR(1) (Lookahead LR)
4] CLR(1) (Canonical LR)

Least powerful ↓ most powerful.



For all LR parsing LR parsing algorithm, i/p buffer & stack are same but there is only change in LR parsing tables.

LR(0) :- In order to construct the parsing table of LR(0) & SLR(1) we use something called canonical collections of LR(0) items, & for LALR(1) & CLR(1) canonical collection of LR(1) items.

LR items :- are nothing but grammar rules with a special marker 'dot' at some position on the RHS of the production.

ex. production is $A \rightarrow XYZ$ then it has the following four possible items

$A \rightarrow \cdot XYZ$

$A \rightarrow X \cdot YZ$

$A \rightarrow XY \cdot Z$

$A \rightarrow XYZ \cdot$ .... [once we see everything then we can reduce XYZ to A]

The position of the dot in the preceding LR items tells how far we have seen in parsing the production. Everything to the left of the dot has already been seen & has been pushed onto the parsing stack. Symbols to the right of the dot indicate symbols that are yet to be seen.

How to construct LR(0) parsing table :-

ex $S \rightarrow AA$

$A \rightarrow aA \mid b$

First construct augmented grammar by adding production

$S' \rightarrow S$

$S \rightarrow AA$

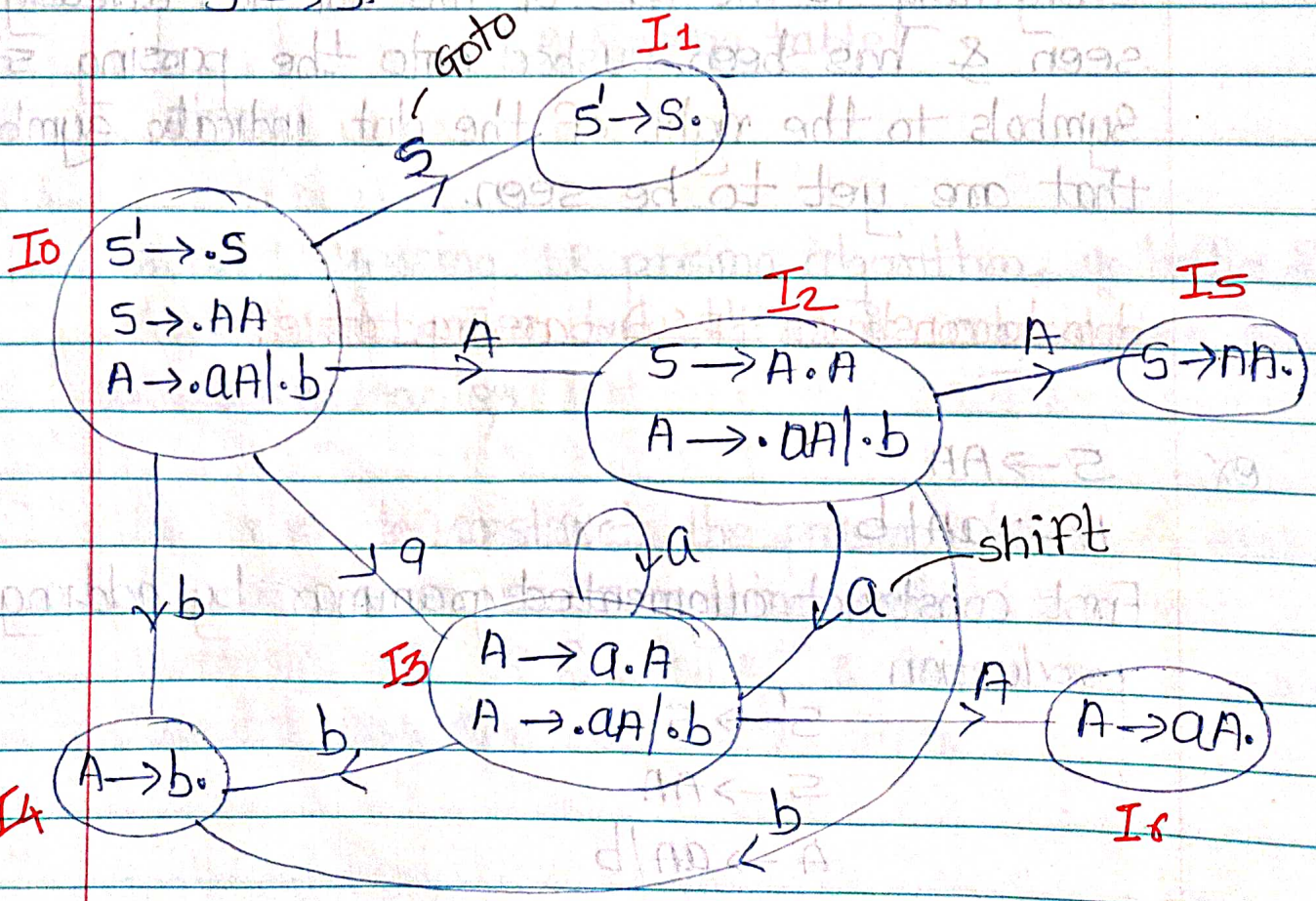
$A \rightarrow aA \mid b$

Two functions are important for construction of LR(0) parsing table.

1] closure :- means whenever there is • (dot) to the left of variable (Non-terminal) you have to add all the production of that variable with • in the begining.

2] Goto :- It is like DFA. move the • to next symbol

Process always start with LR item for 1st production $S' \rightarrow S.$



on variable we are having goto, & on symbol or terminal we are having shift

Explanation: $S' \rightarrow S$

↓ LR item

$S' \rightarrow .S$ ↓ apply closure (S)

$S \rightarrow .AA$ apply closure (A)

$A \rightarrow .aA | .b$

↓

Apply goto $S \Rightarrow S' \rightarrow S.$

goto $A \rightarrow S \rightarrow A.A$ ↓

$A \rightarrow .aA | .b$ apply closure A

goto a

$A \rightarrow a.A$ ↓

$A \rightarrow .aA | .b$ closure A.

goto on b.

$A \rightarrow b.$

& so on.

Assign state a name.

Out of given state I1, I4, I5, & I6 are final states means . at Left most side of RHS. & state I1 i.e $S' \rightarrow S.$ is aceeptence state. final state are nothing but reduction rule.

Simply assign Nos to productions

$S \rightarrow AA$ — 1 i.e I5

$A \rightarrow aA$ — 2 i.e I6

$A \rightarrow b$ 3 i.e I4.

blank entries in parsing table represent errors

	Action			Goto	
states	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		

reduce moves are placed in entire row

$r_{<int>}$ → reduce the terminal to LHS by applying production rule no <int>

ex r_3 → apply reduction rule no 3 i.e A→b

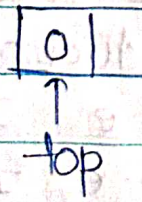
ex: $S' \rightarrow S$

$S \rightarrow AA$

$A \rightarrow aA|b$

string aabb$

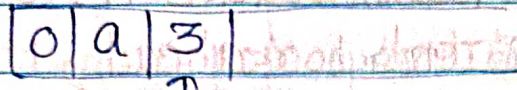
Initial top of stack contain initial state i.e 0



stack.

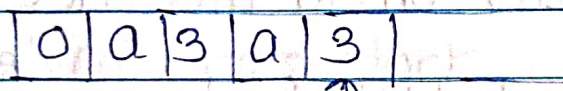
i/p symbol read is a & top of stake is 0

0 on a is S_3 (shift 3).
i.e shift current symbol & 3 on to the stack respectively.



increment lookahead a a b b $

∴ 3 on a S_3.
push a & 3 onto stack.



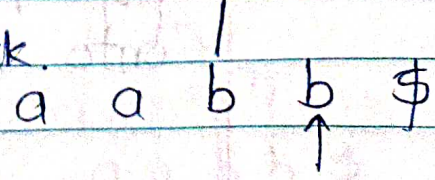
increment lookahead a a b b $

3 on b is S_4 push b & 4 onto stack

Increment lookahead a a b b $

4 on b is r_3 apply reduction rule no 3 for previous symbol of current lookahead.

rule No 3 : A → b.
find length of RHS |b| i.e 1.
then pop 2 elements from stack.



if the size of RHS of reduction production is n then we have to pop 2n elements from stack

Whenever we found reduction, that reduction is not for current lookahead, it is for previous symbol.

0	a	3	a	3	b	4

$A \rightarrow b$ pop 2

then push LHS on to the stack i.e A

0	a	3	a	3	b	4	A

consider the above 2 element of stack

3 on A is 6 (push 6 onto stack)

because top of stack always contain state NO.

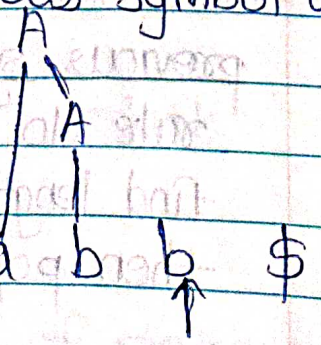
0	a	3	a	3	b	4	A	6

6 on current lookahead b is r2 i.e

$A \rightarrow aAb$ apply for previous symbol of current lookahead i.e →

Length of RHS i.e $|aAb| = 2$

pop 4 elements & push A onto stack.



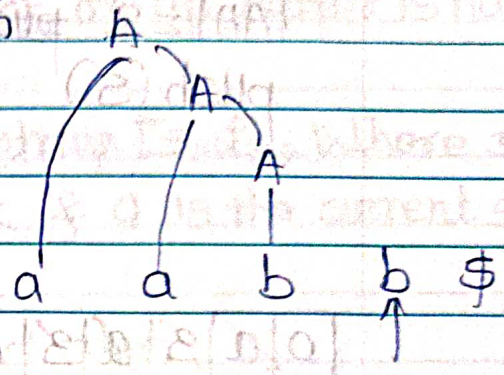
0	a	3	a	3	b	4	A	6	A	6

3 on A 6. push(6)

6 on b (current lookahead) r2.

A → aA for previous symbol of current lookahead

|aA| = 2 = 4 pop operation



0	a	3	a	3	b	4	A	6	A	6	A

push(A) onto stack

0	a	3	a	3	b	4	A	6	A	6	A	2

0 on A is 2 push(2)

2 on b is S4

push(b) then push(4)

Increment lookahead pointer. a a b b $

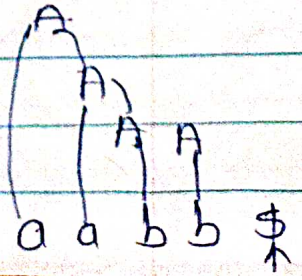
0	a	3	a	3	b	4	A	6	A	6	A	2	b	4

4 on $ is r3

i.e A → b for previous symbol (b) of $

|b| = 1 pop 2

push(A).



0	a	3	a	3	b	4	A	6	A	8	A	2	b	4	A	5

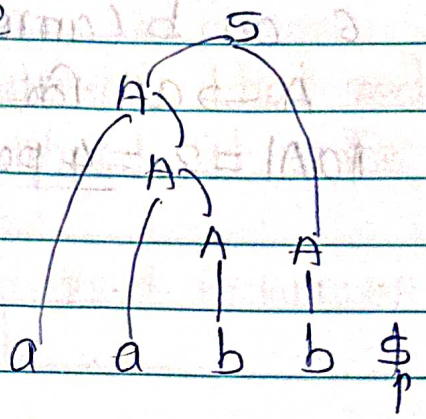
2 on A is 5 push(5)

5 on $ is r1 i.e

$S \rightarrow AA$

|AA| = 2 pop 4.

push(S)



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

0 on S 1 push(1)

1 on $ is accept

LR(0) → zero indicate lookahead we are seeing nothing to take decision of reduction.

In practical we are reading i/p one symbol at a time but reduction decision is not for current symbol. It is always for previous symbol of current symbol being read.

Thats why LR(0)

LR Parsing algorithm:-

1] Initialize stack with S_0, the initial state & w$ in the i/p buffer.

2] Repeat step 3 while the stack or i/p buffer is not empt

3] Consult the parsing table for entries [s, a], where s is the state on top of the stack & a is the current symbol from the i/p token.

If ACTION[s, a] = shift s' then.
- Push symbol a followed by s' on top of the stack
- Increment lookahead pointer

else if ACTION[s, a] = reduce $A \rightarrow \beta$ then.
- Pop $2r$ top symbols from the stack ($r = |\beta|$)
- push A onto the stack.
- push the entry [S.top-1, A] on the stack
- Take the semantic action as desired.

Else if ACTION[s, a] = accept then
- return with success.

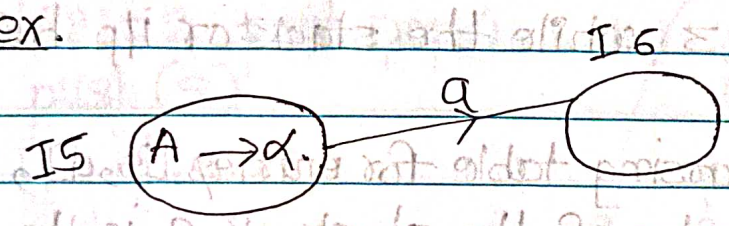
else
- report error.

Disadvantages of LR(0) -

1] Slower in term of error detection.

2] It introduce sr conflicts

ex.

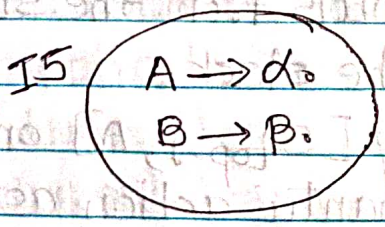


	Action	
	a	b
5	S6/r1	r1

known as sr conflicts

3] It introduce rr conflicts

ex



	a	b	c
5	r1/r2	r1/r2	r1/r2

(rr conflicts

SLR(1):- Difference between LR(0) & SLR(1) in terms of reduction move. Placement of

Consider ex

$S' \rightarrow S$

$S \rightarrow AA$

$A \rightarrow aA/b$

I4, I5, I6 are final states

	a	b	$
:			
4	r3	r3	r3
5			r1
6	r2	r2	r2

State I4 is $A \rightarrow b.$ i.e r3.

place above reduction (r3) in row = 4 & col = Follow(A) or Follow(LHS).

Follow(A) = {a, b, $}

state I5 is $S \rightarrow AA.$ → r1

row = 5 col = Follow(S) i.e {$}

state I6 $A \rightarrow aA.$ — r2.

row = 6, col = Follow(A) i.e {a, b, $}

Disadvantages:-

1] It may have sr or rr conflicts.

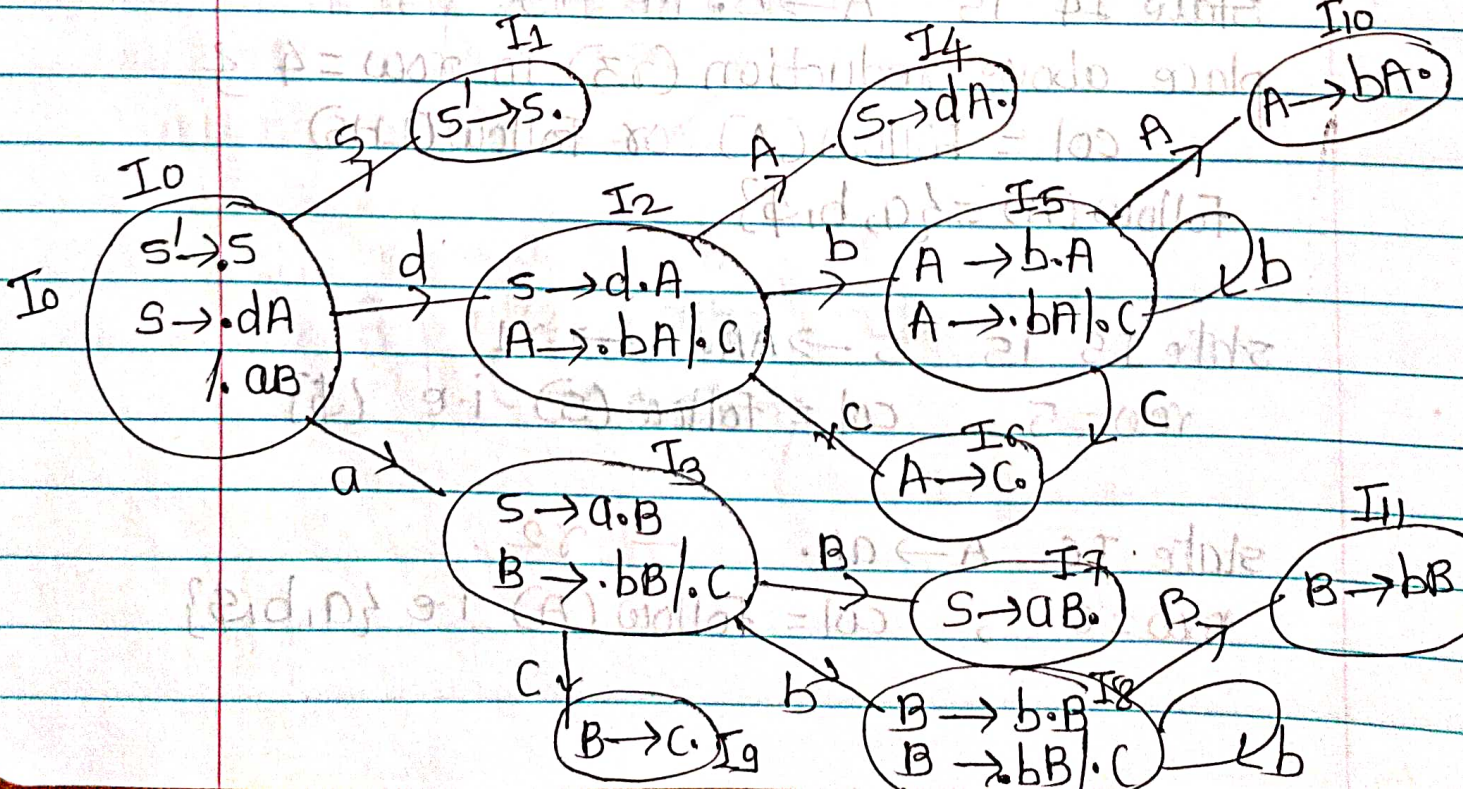
2] SLR(1) is more powerful than LR(0)

ex Identify whether given grammar LL(1), LR(0) or SLR(1).

1)

	row	col.
$S \to dA \mid aB$	S	$S \to dA\{d\}, S \to aB\{a\}$
$A \to bA \mid c$	A	$A \to bA\{b\}, A \to c\{c\}$
$B \to bB \mid e$	B	$B \to bB\{b\}, B \to c\{c\}$

∴ Grammar is LL(1)



There is no SR & RR conflict

∴ Grammar is LR(0).

∴ Thus Grammar is SLR(1)

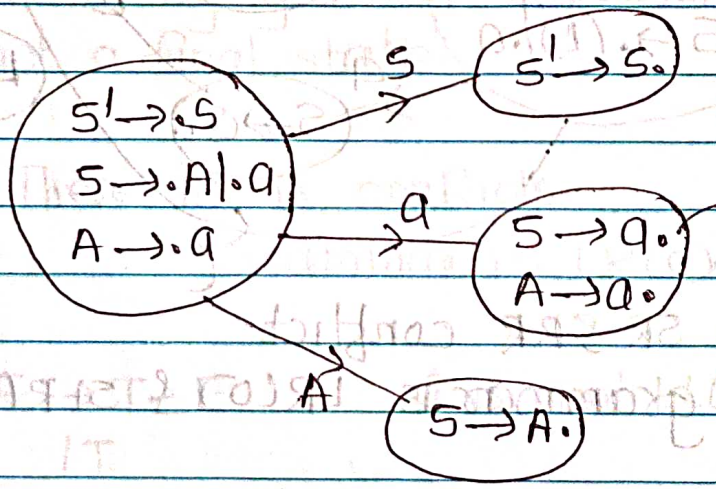
2] $S \to A|a$, $A \to a$

row: A — col: $S \to A$ {a}, $S \to a$ {a}

∴ Not LL(1).

Given grammar is ambiguous.

∴ It is not suitable for any parser.

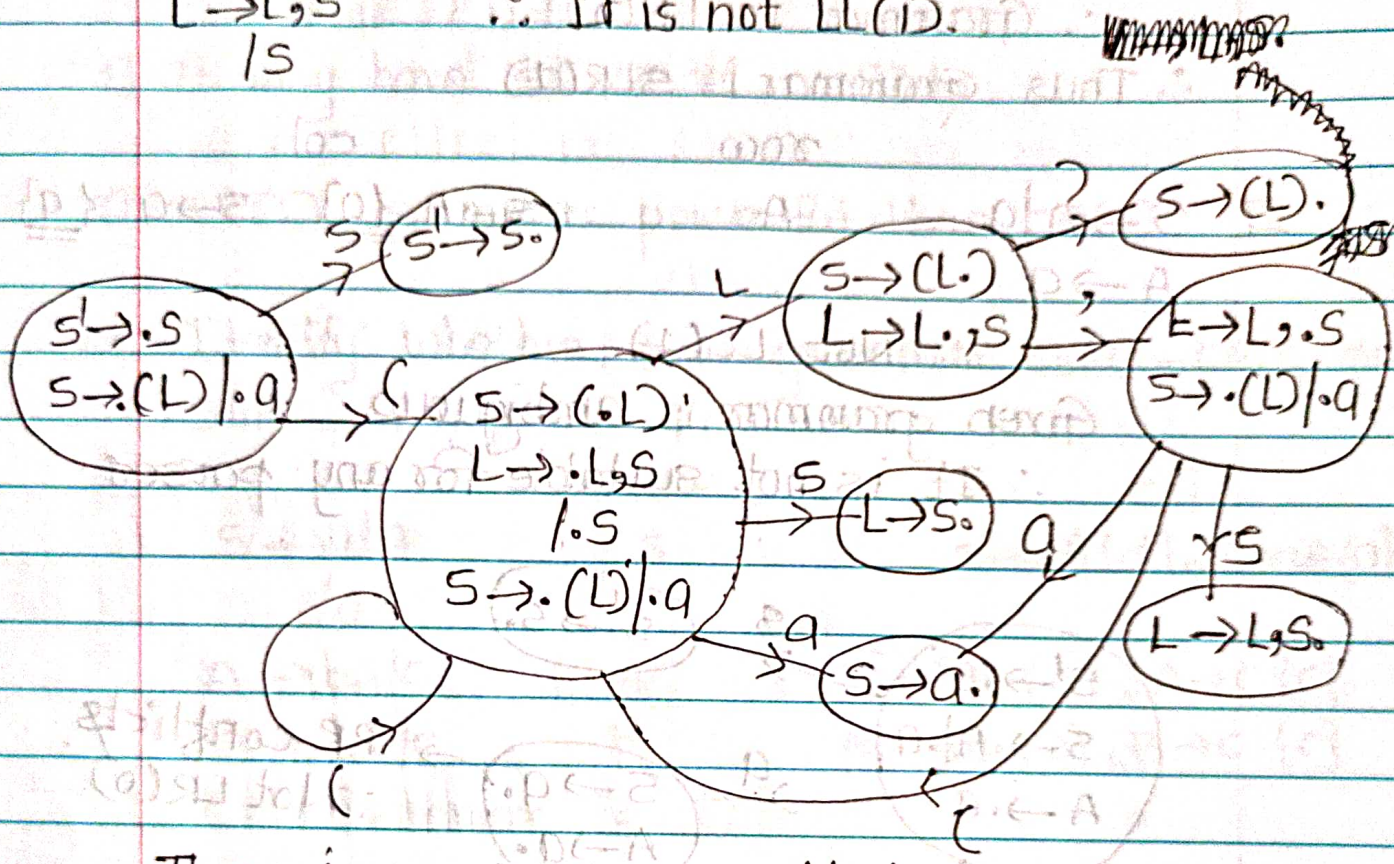


$S \to a.$ Follow(S) = {$}

$A \to a.$ Follow(A) = {$}

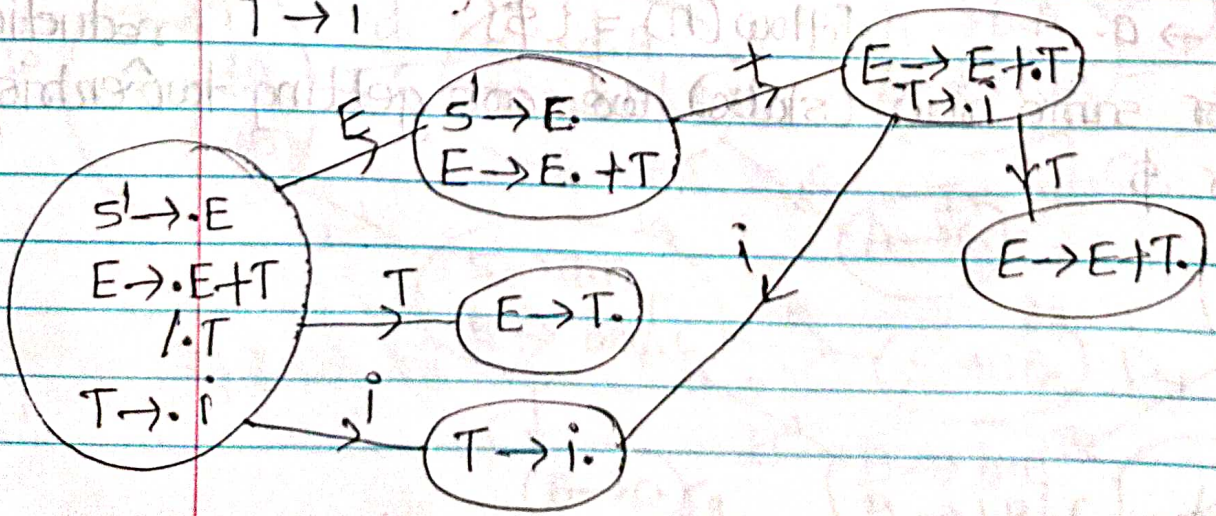
reduction for same row (state) we are getting two entries for $

3] $S \to (L) \mid a$ As it is Left Recursive
$L \to L,S$ ∴ It is not LL(1).
$\mid S$



There is no SR & RR conflict
therefore grammar is LR(0) & SLR(1)

4] $E \to E+T$
$\mid T$
$T \to i$



$S' \to E.$ this is not a final state because follow of $ is $

$\therefore$ $S' \to E.\$$ —$\$$→ $S' \to E\$.$

↓ This is final state

Generally we are not considering $, but $S' \to E.$ is not final state

also $S' \to E.$ This is augmented production added by you. It represent accepting state & not a final state.

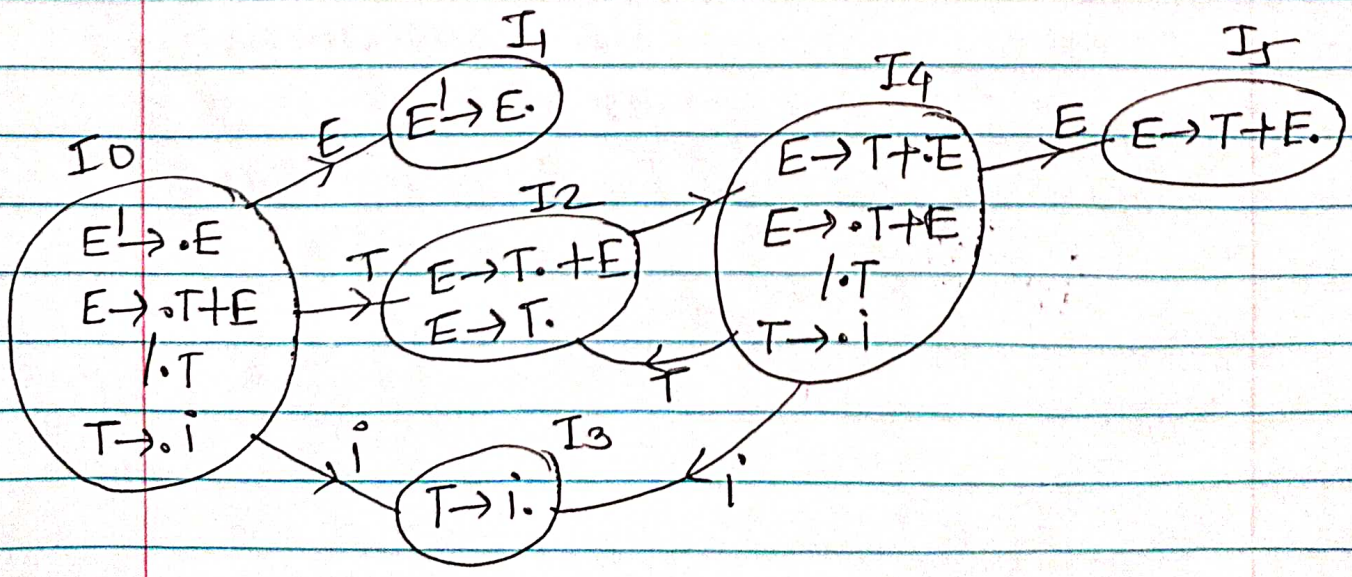
$\therefore$ There is no conflicts.

$\therefore$ Grammar is LR(0) & SLR(1)

5] $E \to T+E$ | T

$T \to i$

G is not LL(1)



state I_2

$E \to T. + E$

$E \to T.$

SR conflict is there.

$\therefore$ Grammar is not LR(0).

$E \to T$

Follow(E) = {$}

$\therefore$ Grammar is SLR(1)

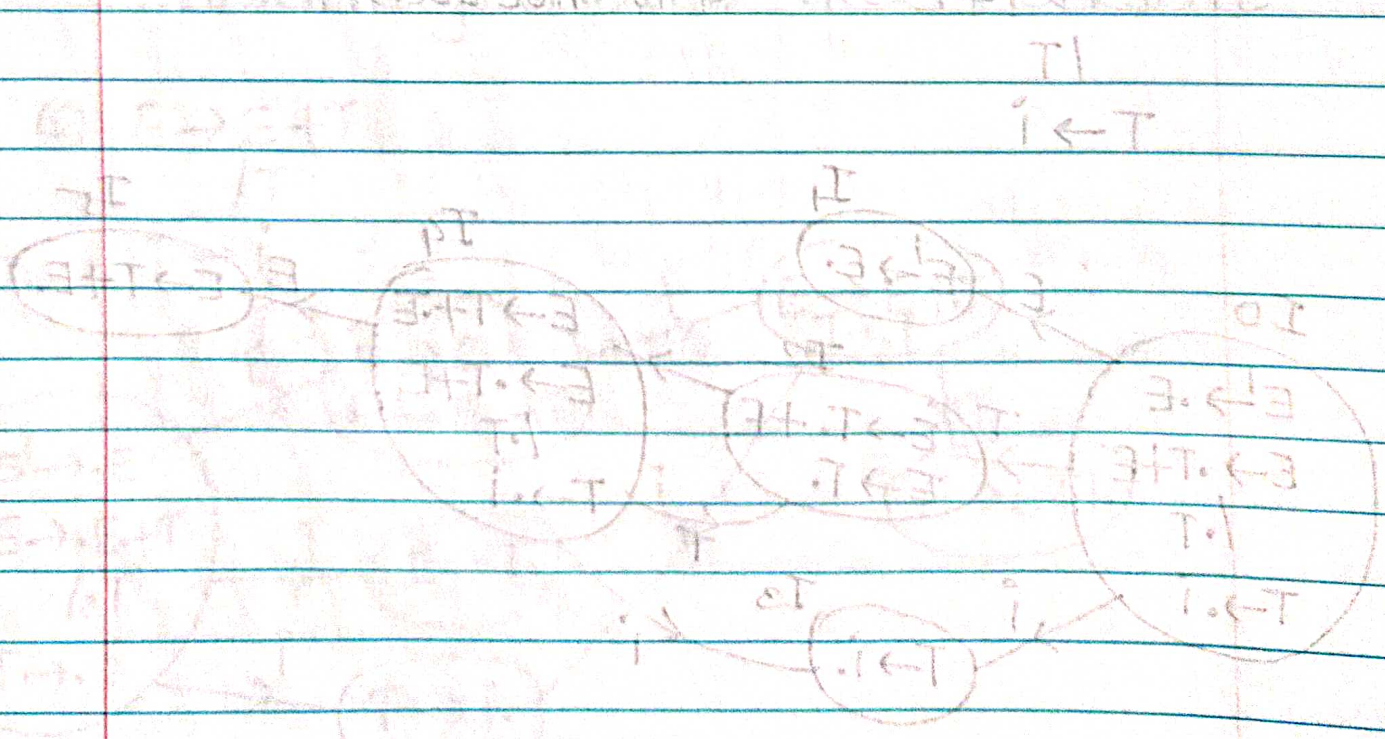
6] $E \to E+T$ / T — Ans

LL(1) ✗

$T \to TF$ / F — LR(0) ✗

SLR(1) ✓

$F \to F* / a / b$



If I have $A \to .E$, $A \to E.$ } then I can write $A \to .$

7] $S \to AaAb$ / $BbBa$

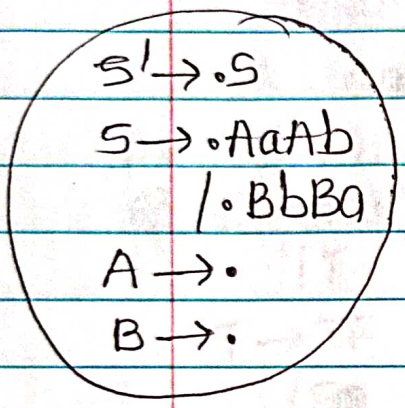
$A \to \epsilon$

$B \to \epsilon$

LL(1) ✗

∴ There are two reduce move in the same state.

∴ Grammar is not LR(0)



for SLR(1)

follow (A) = {a, b}

follow (B) = {a, b}

∴ Grammar is not SLR(1)

8] $S \to AS|b$

$A \to SA|a$

Ans

LL(1) ✗

LR(0) ✗

SLR(1) ✗

} Given grammar Ambiguous G.

9] $S \rightarrow Aa$ LL(1) X

$/bAC$ LR(0) X

$/dc$ SLR(1) X

$/bdg$

$A \rightarrow d$

3] CLR(1) :-

We are not going to use canonical collection of LR(0) item instead we use canonical collection of LR(1) items.

LR(1) items = LR(0) items + lookahead

ex. $S \rightarrow .aA, a/b$

($S \rightarrow .aA$: LR(0) item; a/b : lookahead.)

ex $S \rightarrow AA$

$A \rightarrow aA/b$

Add augmented production

$S' \rightarrow S$

$S \rightarrow AA$

$A \rightarrow aA/b$

LR(1) item for augmented production.

for start symbol lookahead is alway $

$S' \rightarrow .S, \$$

$\Downarrow$ closure(S)

$S \rightarrow .AA, \$$

$\Downarrow$ closure(A)

$A \rightarrow .aA/.b$, ~~follow~~ First(A) i.e {a/b}

i.e $A \rightarrow .aA/.b, a/b \Rightarrow A \rightarrow .aA, a/b$

$/.b, a/b$

Rules to find lookahead

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

⇓

closure (B).

$B \rightarrow .\gamma$, First (β)

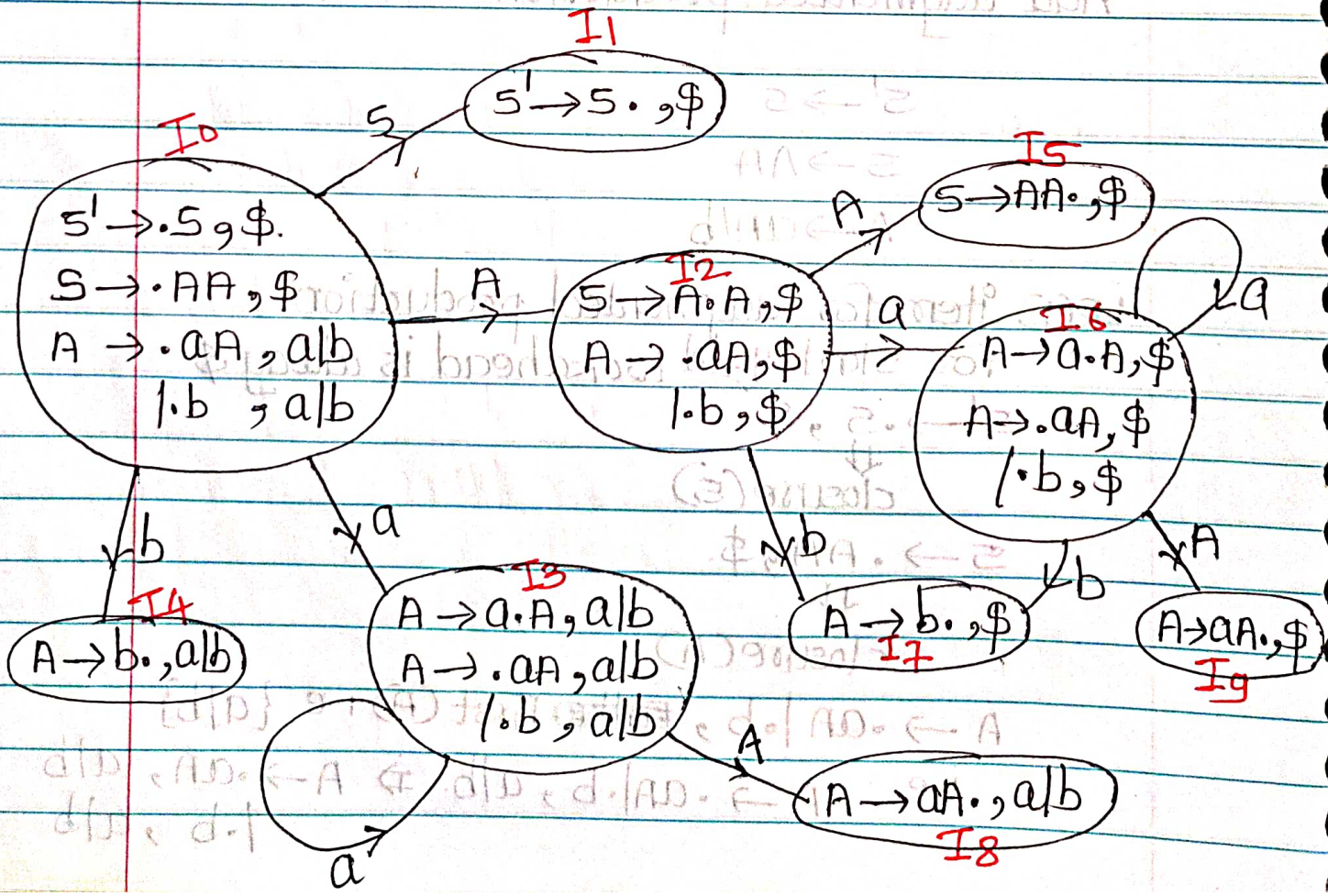
OR

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

⇓

closure (B)

$B \rightarrow .\gamma$, a/b ← as it is, if there is no variable after B.



If i apply transition or goto over a symbol then we are writting the same production rule with the change in position of dot towards one symbol right also we are going to keep lookahead for that production same in next state.

We are going to place the reduce move under the column equal to lookahead symbols.

In CLR(1) we are getting more no. states than in LR(0) & SLR(1) because here we are having some states which contain same LR(0) items but different lookahead.

Parsing table :-

	a	b	$	S	A
0	S3	S4			2
1					
2	S6	S7			5
3	S3	S4			8
4	r3	r3			
5			r1		
6	S6	S7			9
7			r3		
8	r2	r2			
9			r2		

In above parsing table

I3, I6
I4, I7
I8, I9

These pair of states are same in terms of LR(0) items & having different lookahead.

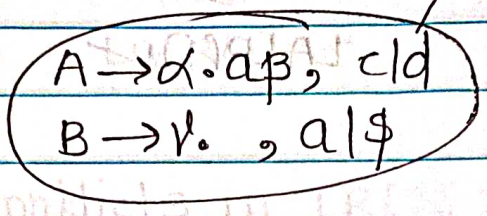
If i merge above pair of states & consider single state ex I3, I6 ⇒ I36, I4, I7 ⇒ I47, I8, I9 ⇒ I89

LALR(1) :- Parsing table is same as CLR(1) parsing table, only those state which are having same LR(0) item & different lookaheads, such states are merge into single state. also merge the behavior of those states

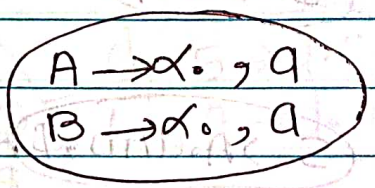
∴ No of states in LALR(1) is exactly equal to the number of states in LR(0) & SLR(1)

Conflicts in CLR(1) :-

1] SR conflicts



2] RR conflicts

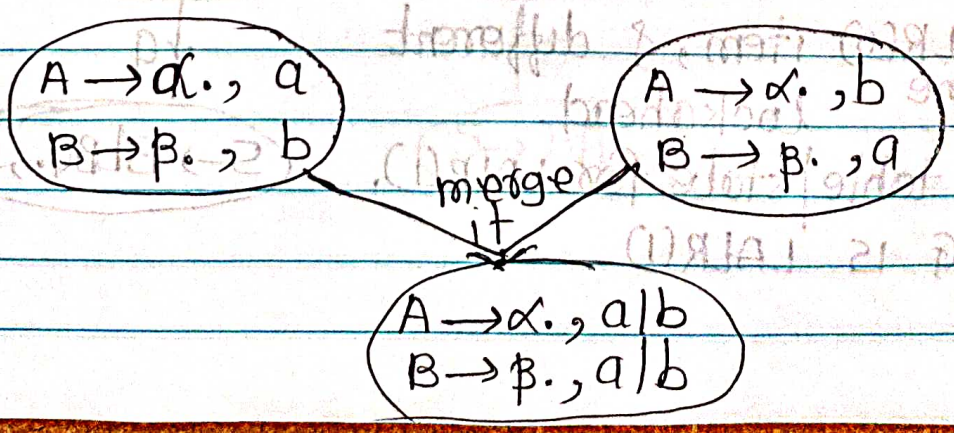


SR, RR conflicts depends on lookaheads.

If grammar is not CLR(1) then it is also not in LALR(1)

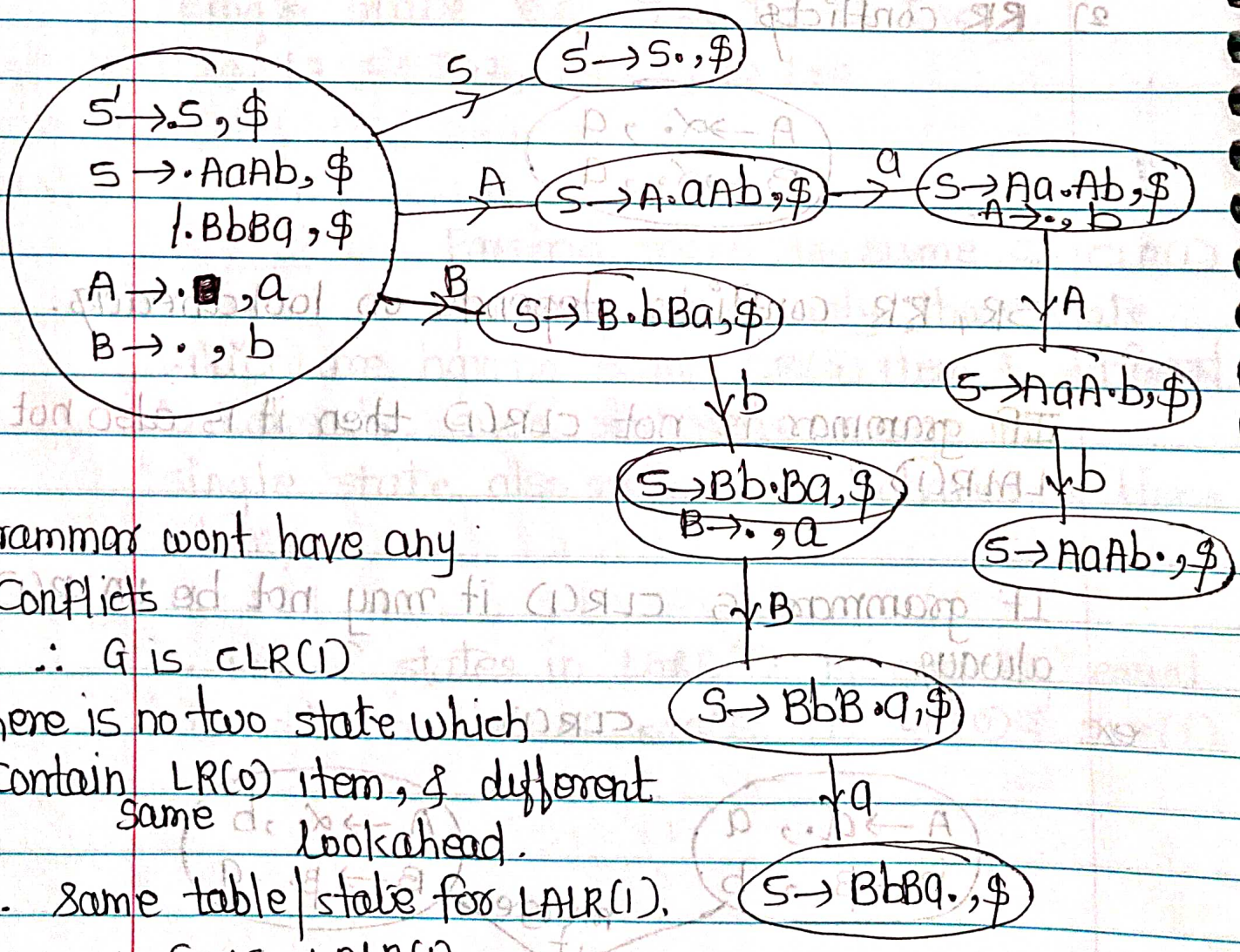
If grammar is CLR(1) it may not be LALR(1) always.

ex CLR(1)



ex $S \to AaAb$ | $BbBa$
$A \to \epsilon$
$B \to \epsilon$

LL(1) ✓
LR(0) ✗
SLR(1) ✗
CLR(1) ✓
LALR(1) ✓



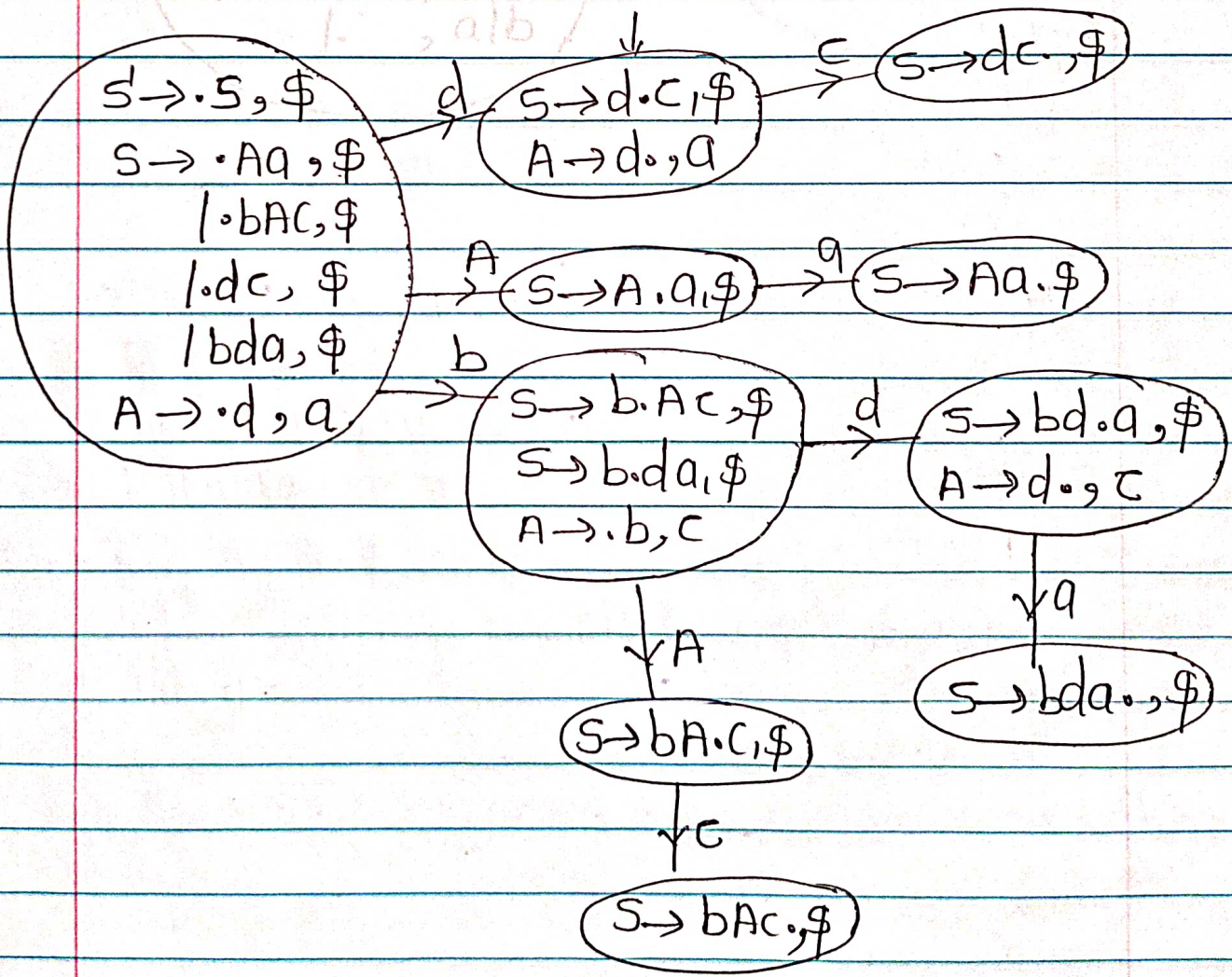
Grammar wont have any conflicts
∴ G is CLR(1)

There is no two state which contain LR(0) item, & different same lookahead.
∴ Same table/state for LALR(1).
∴ G IS LALR(1)

ex $S \to Aa$ | bAC | dc | bda | $A \to d$

LL(1) X
LR(0) X
SLR(1) X
CLR(1) ✓
LALR(1) ✓

There are conflicts in LR(0) items but not in LR(1)
∴ Grammar is CLR(1) & LALR(1)



ex. $S \to Aa$ | bAc | Bc | bBa

$A \to d$

$B \to d$

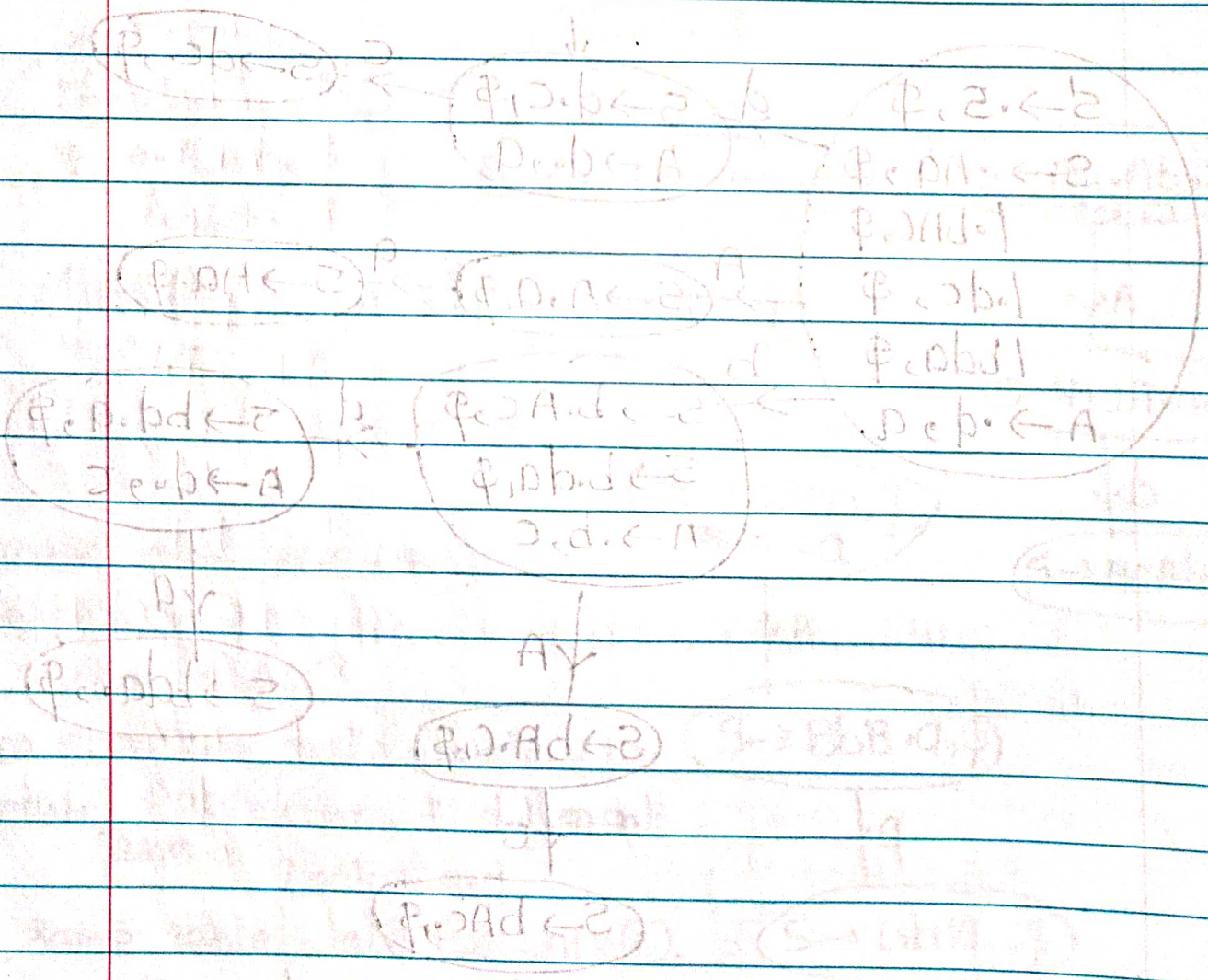
LL(1) ✗

LR(0) ✗

SLR(1) ✗

CLR(1) ✓

LALR(1) ✗

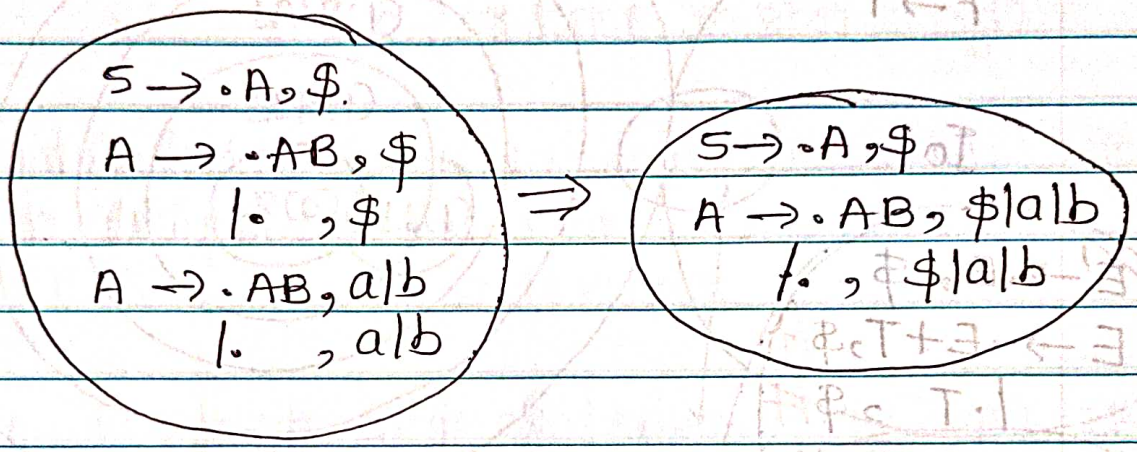


ex $S \to A$

$A \to AB \mid \epsilon$

$B \to aB \mid b$

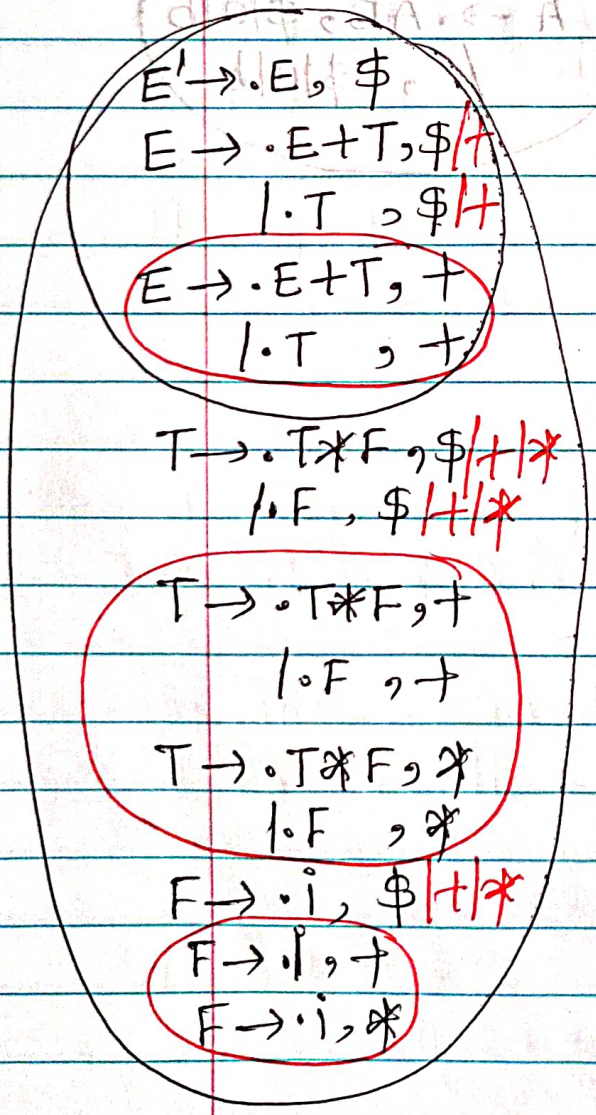
closure $(S \to .A, \$)$



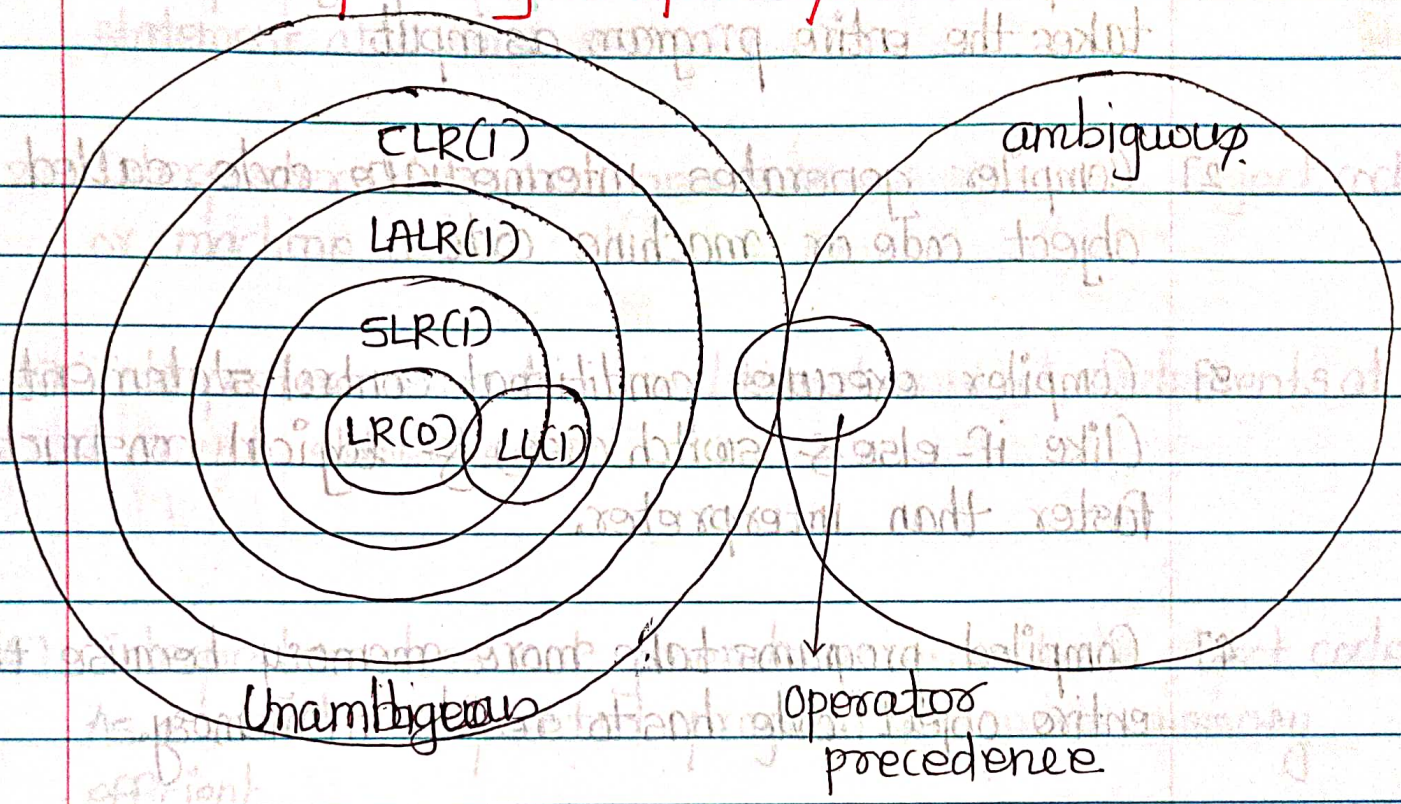
ex. $E \to E+T \mid T$

$T \to T*F \mid F$

$F \to i$



Relationship among LR parsers



every LL(1) grammar is LALR(1)

If grammar is LR(0) then it is also SLR(1), LALR(1) & CLR(1)

If grammar is SLR(1) then it is also LALR(1) & CLR(1)

If grammar is LALR(1) then it is CLR(1)

Compiler

1] Compiler works on the complete program at once. It takes the entire program as input

2] Compiler generates intermediate code, called the object code or machine code.

3] Compiler executes conditional control statement (like if-else & switch case) & logical constructs faster than interpreter.

4] Compiled programs take more memory because the entire object code has to reside in memory.

5] Compile once & run anytime. Compiled program does not need to be compiled every time.

6] Errors are reported after the entire program is checked for syntactical & other errors.

7] Compiler does not allow a program to run until it is completely error-free.

8] Compiled languages are more efficient but difficult to debug.

Interpreter

1] Interpreter program works line-by-line. It takes one statement at a time as input.

2] Interpreter does not generate intermediate object code or machine code.

3] Interpreter execute conditional control statements at a much slower speed.

4] Interpreter does not generate intermediate object code. As a result interpreted program are more memory efficient.

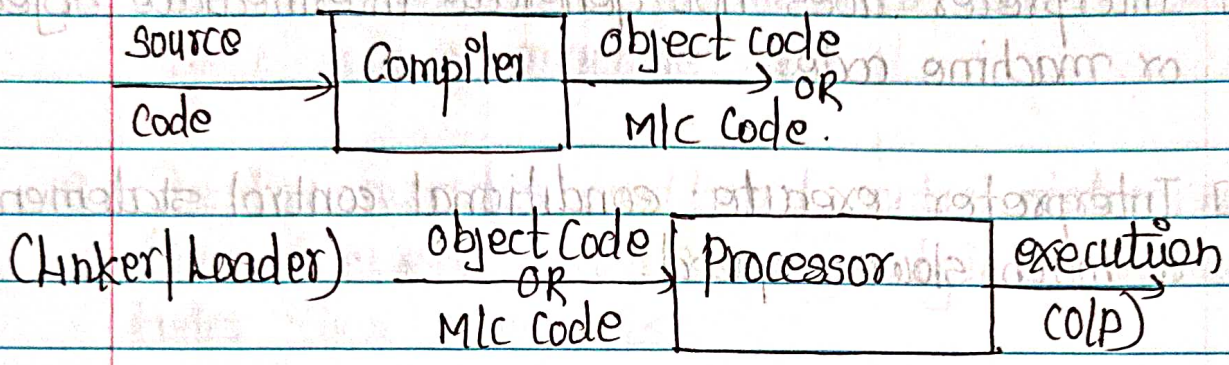
5] Interpreted programs are interpreted line-by-line every time they are run.

6] Error is reported as soon as the first error is encounte-ed. Rest of the program will not be checked until the existing error is removed.

7] Interpreter runs the program from first line & stops execution only if it encounters an error.

8] Interpreted languages are less efficient but easier to debug.

9] Examples of programming languages that use compilers: C, C++, COBOL



9] Examples of programming languages that use interpreter
BASIC, Visual Basic, Python, Ruby, PHP, Perl,
MATLAB, LISP.