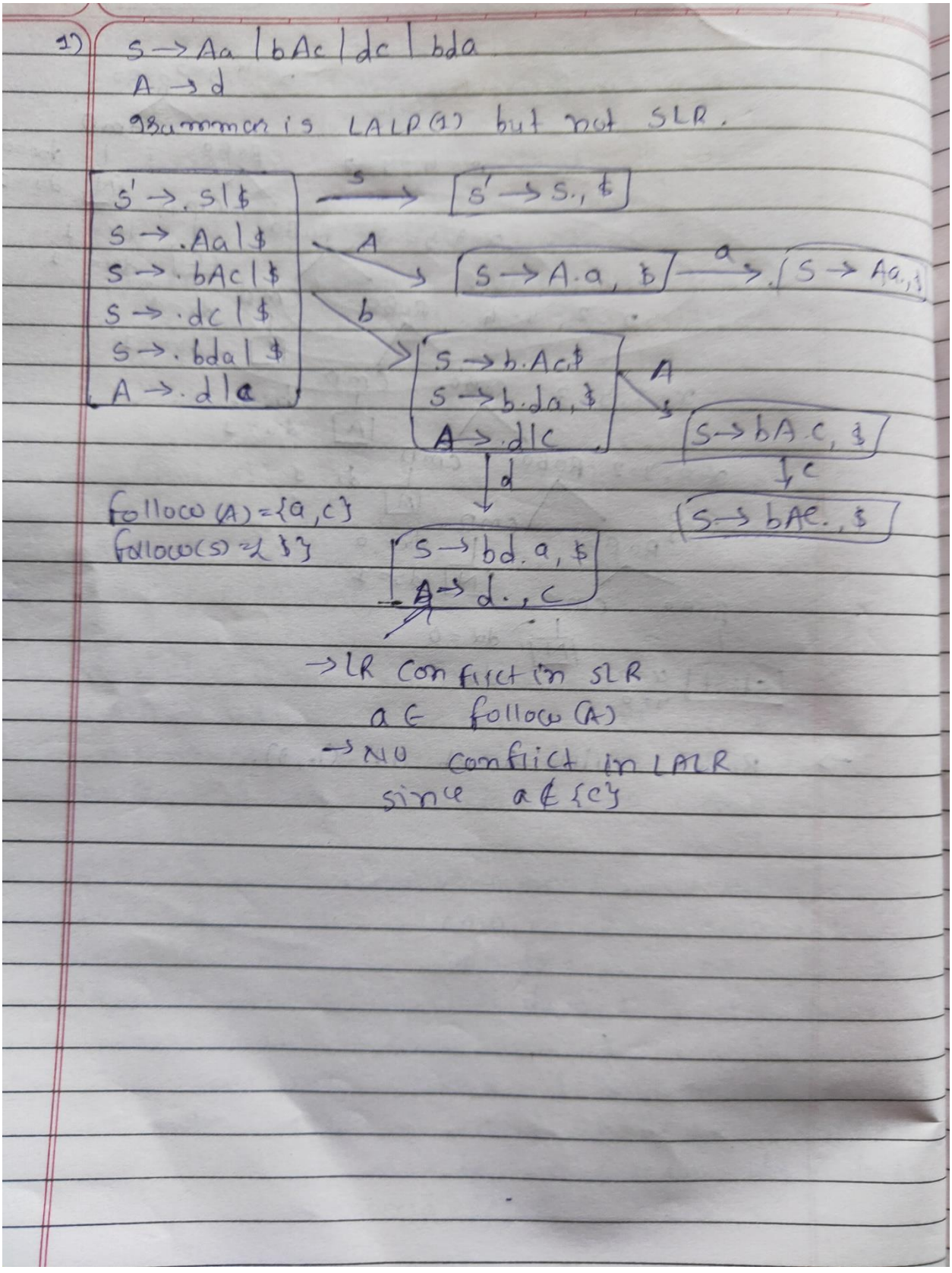
Assignment – 2



(2.) Is the following grammar suitable for LL(1) Parsing? If not make it suitable for LL(1) Parsing. compute FIRST and FOLLOW sets. Generate the Parsing table.

$S \rightarrow AB$
$A \rightarrow Ca \mid \epsilon$
$B \rightarrow BaAC \mid c$
$C \rightarrow b \mid \epsilon$

→ above grammar is not suitable for LL(1) Parsing due to left recursion.

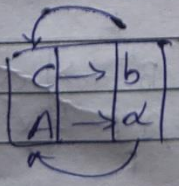
⇝ step 1: Remove left recursion

$S \rightarrow AB$
$A \rightarrow Ca \mid \epsilon$
~~$B \rightarrow CB'$~~ $B \rightarrow CB'$
$B' \rightarrow aACB' \mid \epsilon$
$C \rightarrow b \mid \epsilon$

⇝ step 2: compute FIRST and Follow of non terminals

⇒ FIRST(C)

$C \rightarrow b$



Rule 1
add α to FIRST(A)

$C \rightarrow \epsilon$ $C \rightarrow \epsilon$
$A \rightarrow \epsilon$

FIRST(C) = { b, ε }

⇒ FIRST(B)

B → cB'

B	→	C	B'
A	→	α	

Rule 1
add α to FIRST(A)

FIRST(B) = {c}

⇒ FIRST(A)

A → Ca

A	→	C	a
A	→	Y_1	Y_2

Rule 3
First(A) = First(Y_2)

FIRST(A) = FIRST(C) = {b, ∈}

A → ∈

A	→	∈
A	→	α

FIRST(A) = {b, ∈}

⇒ FIRST(B')

B' → aACB'

B'	→	a	ACB'
A	→	α	

Rule 1
add α to First(A)

FIRST(B') = {a}

B' → ∈

Rule 1 add α to First(A)

FIRST(B') = {a, ∈}

=> FIRST (S)

S → AB

S	→	A	B
A	→	Y_1	Y_2

Rule 3
FIRST(A) = FIRST(Y_1)

FIRST(S) = FIRST(A) = {b, ϵ}

~~S → B~~ Note: there is ϵ is given so if we put ϵ like S → ϵ B than we also have to add ~~FIRST~~ FIRST of B. if ϵ is not these in A then we will not add B to it.

S → B

S	→	B
A	→	Y_1

Rule 3
FIRST(A) = FIRST(Y_1)

FIRST(S) = FIRST(B) = {c}

∴ FIRST(S) = {b, c, ϵ}

=> Note: follow of start symbol is $
follow ~~of~~ never contain ϵ

so,

=> follow(S) = {$}

=> follow(A)

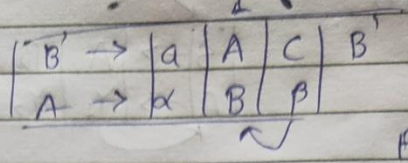
S → AB

S	→	A	B	
A	→	B	B	Rule 2

follow(B) = FIRST(β)

follow(A) = FIRST(B) = {c}

$B' \rightarrow aACB'$

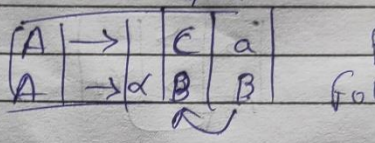


Rule 2
Follow(B) = first(β)

Follow(A) = FIRST(C) = {b}
⇒ we don't Put ε in follow.
So follow(A) = {b, a, $}

follow(C)

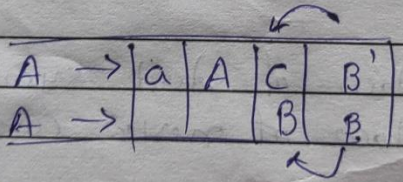
$A \rightarrow Ca$



Rule 2
Follow(B) = First(β)

follow(C) = {a}

$B' \rightarrow aACB'$

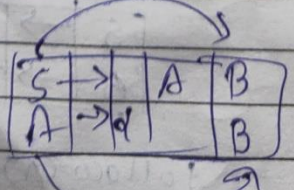


Rule 2

follow(C) = FIRST(B') = {a}
⇒ we don't Put ε in follow.
follow(C) = {a, $}

follow(B)

$S \rightarrow AB$



Rule 3
follow(B) = follow(A)

follow(B) = {$}

FOREVER Page Date: / /

Follow (B')

$B \rightarrow CB'$

B	→		C	B'	Rule 3
A	→	α		B	

Follow (B') = Follow (B)
= {$}

$B' \rightarrow aACB'$

Rule 3

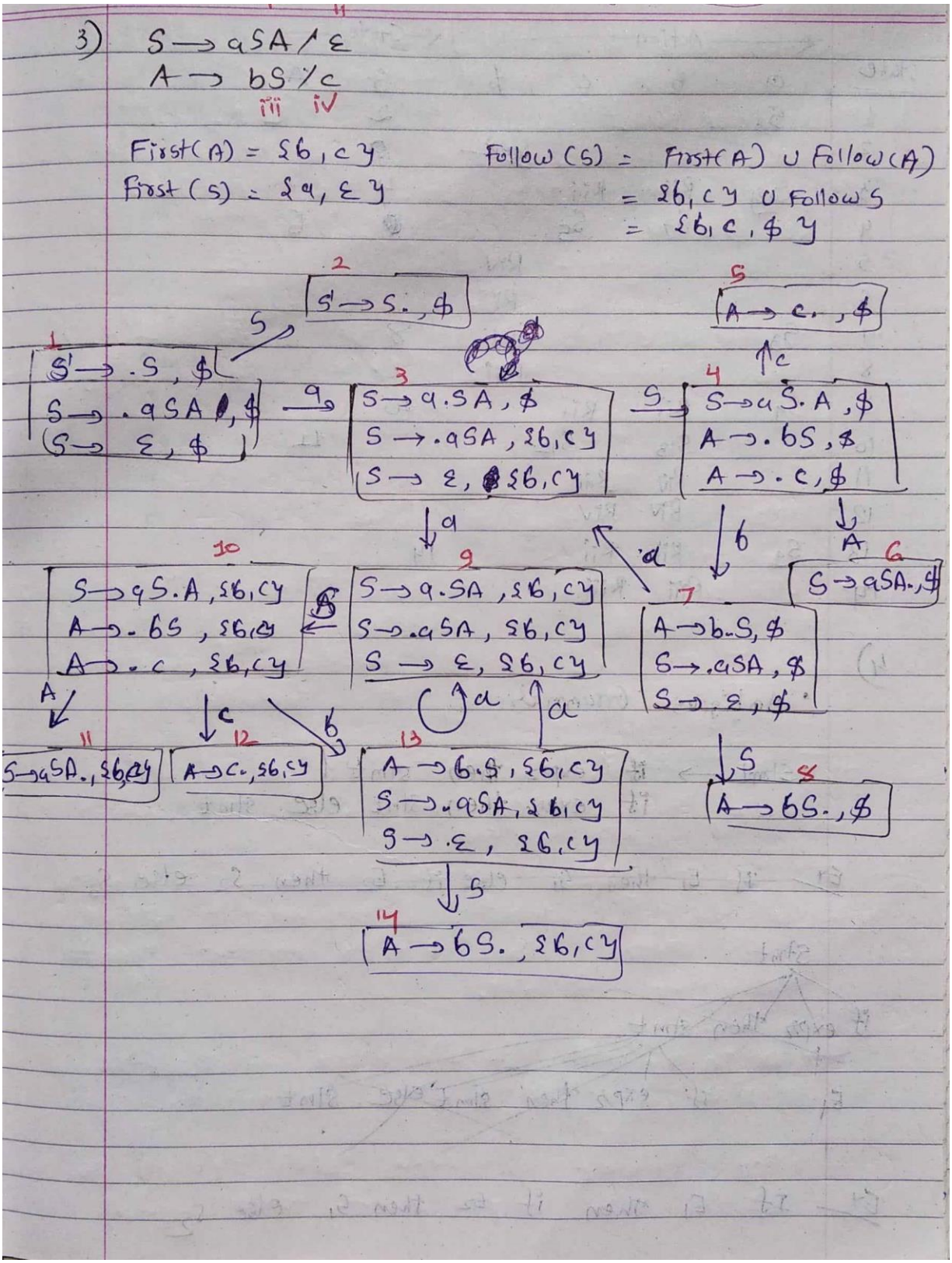
follow (B') = follow (B') = {$}

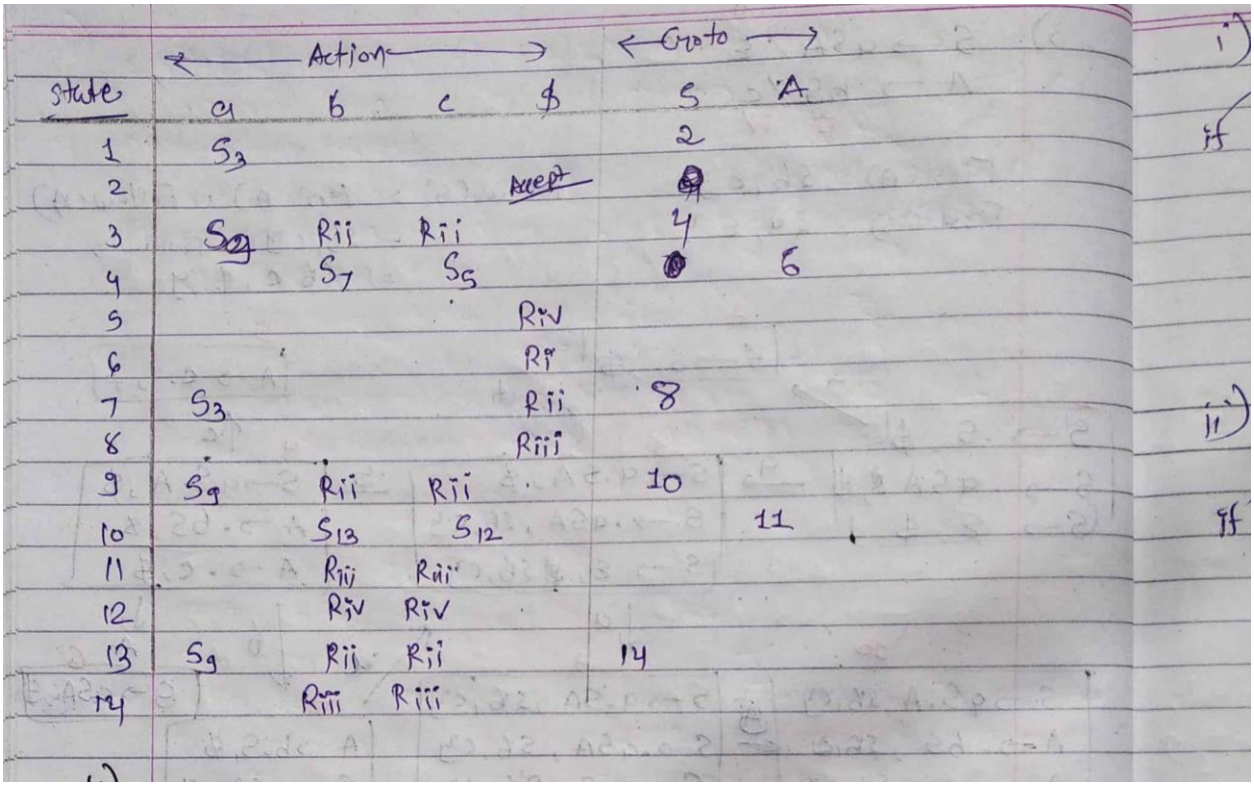
NT	First	Follow
S	{b, C, ϵ}	{$}
A	{b, ϵ}	{b, a, C, $}
B	{C}	{$}
B'	{a, ϵ}	{$}
C	{b, ϵ}	{a, $}

NT	Input symbol			
	a	b	C	$
S		S→AB	S→AB	S→AB
A	A→ϵ	A→ϵ A→Ca	A→ϵ	A→ϵ
B			B→CB'	
B'	B'→aACB'			B'→ϵ
C	C→ϵ	C→b		C→ϵ

Note if ϵ encounter then go to follow

above grammar is not LL(1).


3) S → aSA / ε
A → bS / c
iii iv
First(A) = {b, c}
First(S) = {a, ε}
Follow(S) = First(A) ∪ Follow(A)
= {b, c} ∪ Follow S
= {b, c, $}
1 S' → .S, $
S → .aSA, $
S → ε, $
2 S' → S., $
3 S → a.SA, $
S → .aSA, {b, c}
S → ε, {b, c}
4 S → aS.A, $
A → .bS, $
A → .c, $
5 A → c., $
6 S → aSA., $
7 A → b.S, $
S → .aSA, $
S → ε, $
8 A → bS., $
9 S → a.SA, {b, c}
S → .aSA, {b, c}
S → ε, {b, c}
10 S → aS.A, {b, c}
A → .bS, {b, c}
A → .c, {b, c}
11 S → aSA., {b, c}
12 A → c., {b, c}
13 A → b.S, {b, c}
S → .aSA, {b, c}
S → ε, {b, c}
14 A → bS., {b, c}
S
A
a
b
c



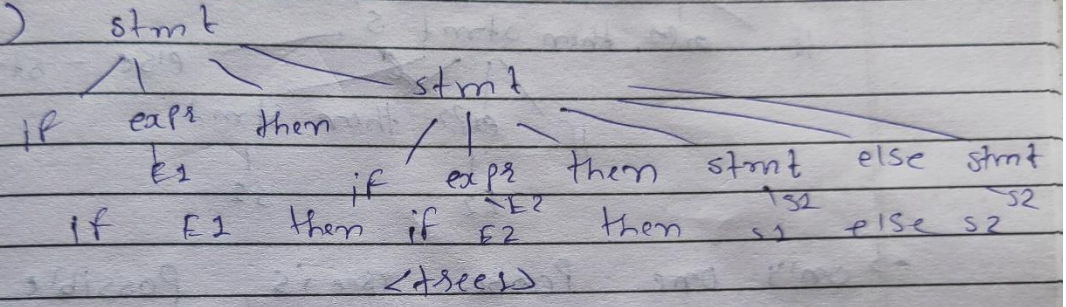
State	Action a	b	c	$	Goto S	A
1	S_3				2	
2				Accept		
3	S_9	Rii	Rii		4	
4		S_7	S_5			6
5				Riv		
6				Ri		
7	S_3			Rii	8	
8				Riii		
9	S_9	Rii	Rii		10	
10		S_{13}	S_{12}			11
11		Riii	Riii			
12		Riv	Riv			
13	S_9	Rii	Rii		14	
14		Riii	Riii			

(h) Write ambiguous and unambiguous production rules for if then else construct. Illustrate Passing using both types of rules by giving an example. Also Explain Left factoring and its use.

stmt → if expr then stmt
| if expr then stmt else stmt

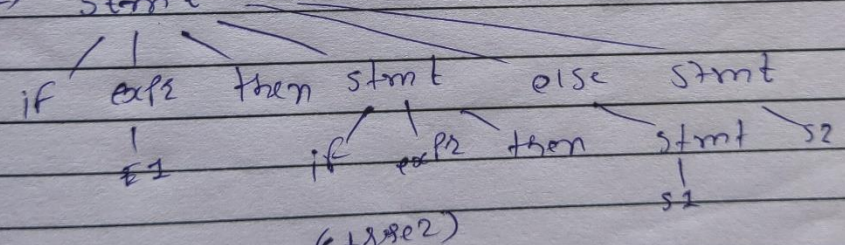
Ex if F_1 then if E_2 then S_1 else S_2

i)



(tree1)

2)



(tree2)

→ Here Given grammar is ambiguous.

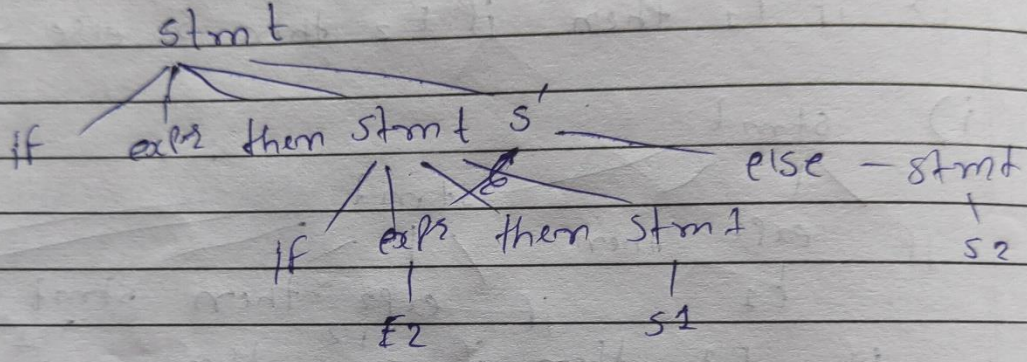
→ Also, first two expression has prefix "if expr then stmt" which makes grammar ambiguous.

→ We can apply left factoring Technique to convert given grammer into unambiguous.

left factoring

stmt → if expr then stmt s
s' → else stmt / ϵ

Ex if E_1 than if E_2 then S_1 else S_2



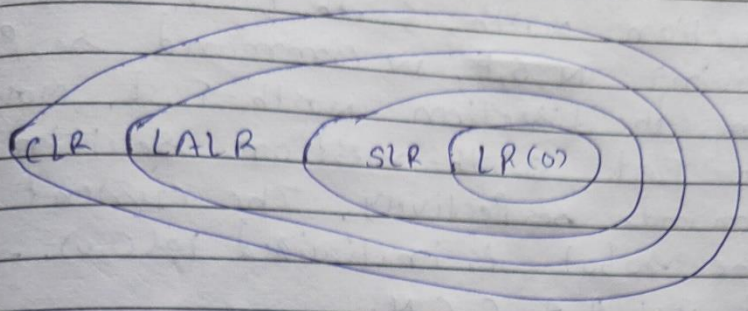
→ only one Passe Tree is Possible for Same Exumple after applying left factoring

(5) Differentiate SLR, Canonical LR and LALR. Also justify the statement "A class of grammar that can be Parsed using LR methods is a Proper subset of the class of grammars that can be Parsed with Predictive Parsers.

SLR	CLR	LALR
→ simple left to right, right most derivation Parsers	Canonical LR Parsers	look ahead LR Parsers
→ Uses LR(0) items	Uses LR(1) items	Uses LR(1) items with core merged into single item
→ don't do any lookahead	Lookahead one symbol	look ahead one symbol
→ are easiest to implement	are difficult to implement	are medium to implement
→ Every SLR(1) grammar is LR(1) and LALR(1)	Every LR(1) grammar is may not SLR(1)	Every LALR(1) grammar is may not SLR(1) but LR(1).
→ is least Powerful	is most Powerful	is intermediate Powerful

Date: / / Page

→ Relation ship between Passes



→ In LL(k) Passing, top-down approach is applied that means stack contains current transition and based on current non-terminal prefix of input string should be inherited.

→ Key difference between LL(k) & LR(k) is, LL(k) uses only current input symbol while LR(k) Passes considers current input symbol as well as next k symbols.

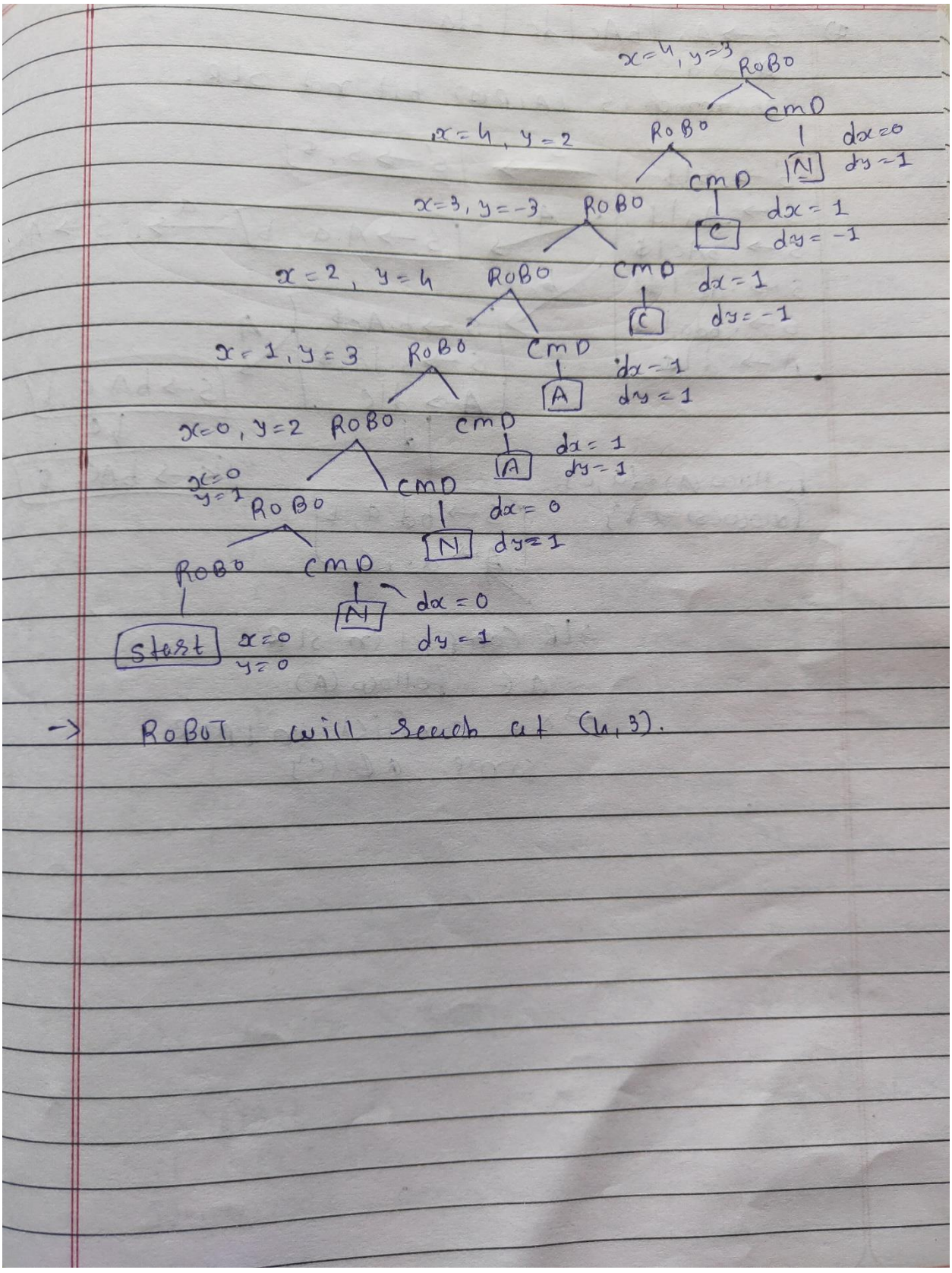
→ so based on algorithm, LR(k) decides weather to shift or reduce operation

→ so, LL(k) provides more strict grammers than LR(k), for ex, LL(k) must be non-left Recursive & left-factored So it proved that LR(k) Passes

(6.) A robot is to be moved to a unit step in a direction specified as a command given to it. The robot moves in the direction North, South, East, West on receiving N, S, E, W command respectively & in the direction North-East, North-West, South-East, South-West on receiving A, B, C, D commands respectively. The current position of the robot is initialized to (0,0). Draw stack for N N A A C C N.

Production	Semantic Rules
ROBO → start	ROBO.x = 0, ROBO.y = 0
ROBO → ROBO, CMD	ROBO.x = ROBO.x + CMD.dx ROBO.y = ROBO.y + CMD.dy
CMD → E	CMD.dx = 1, CMD.dy = 0
CMD → W	CMD.dx = -1, CMD.dy = 0
CMD → N	CMD.dx = 0, CMD.dy = 1
CMD → S	CMD.dx = 0, CMD.dy = -1
CMD → A	CMD.dx = 1, CMD.dy = 1
CMD → B	CMD.dx = -1, CMD.dy = 1
CMD → C	CMD.dx = 1, CMD.dy = -1
CMD → D	CMD.dx = -1, CMD.dy = -1

→ We will use ROBO.x & ROBO.y for current position of ROBOT and ROBO.dx & ROBO.dy is used to denote change in dx & dy by current command (CMD).



→ RoBOT will search at (4, 3).

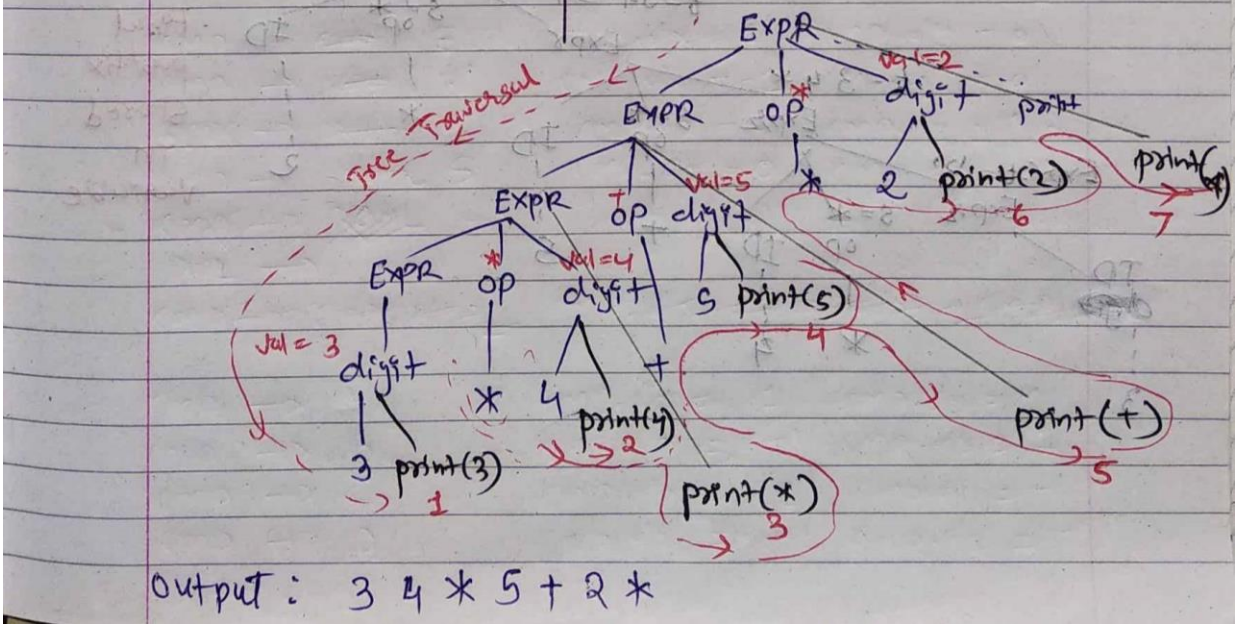
7) infix to postfix

EXPR → EXPR op digit / digit
op → + / - / * / ÷
digit → 0/1/2/3/4/5/6/7/8/9

Method-1: Using Print action

Production	Semantic Rules
EXPR → EXPR op digit	print(OP.val)
EXPR → digit	print (digit.val)
op → +	op.val = '+'
op → -	op.val = '-'
op → *	op.val = '*'
op → ÷	op.val = '÷'
digit → 0/1/2/.../9	digit.val = 1 / digit.val = 2 / ... / digit.val = 9

→ Input: 3*4+5*2



Output: 3 4 * 5 + 2 *

Method-2 : Using store action

Grammer :

$Expr_0 \rightarrow Expr_1\ op\ Term$ $\quad Expr \rightarrow Expr\ op\ ID$

$Expr \rightarrow Term$ $\quad Expr \rightarrow ID$

$Term \rightarrow ID$ $\quad op \rightarrow +/-/*/\div$

$op \rightarrow +/-/*/\div$

e : expression ~~v : value~~ s : sign

Productions	Semantic Rules.
$Expr_0 \rightarrow Expr_1\ op\ ID$	$Expr_0.e = Expr_1.e + ID.val + op.s$
$Expr_0 \rightarrow ID$	$Expr_0.e = ID.val$
$op \rightarrow +$	$op.s = '+'$
$op \rightarrow \div$	$op.s = '\div'$
$op \rightarrow *$	$op.s = '*'$
$op \rightarrow -$	$op.s = '-'$

Input : 3*4+5*2

$t = 34*5+2*$