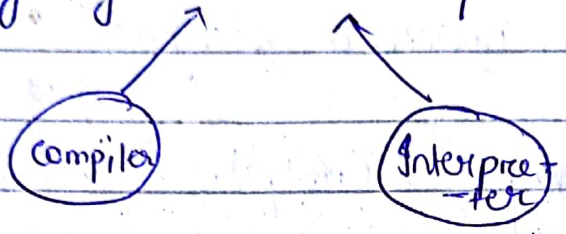
23/7/18

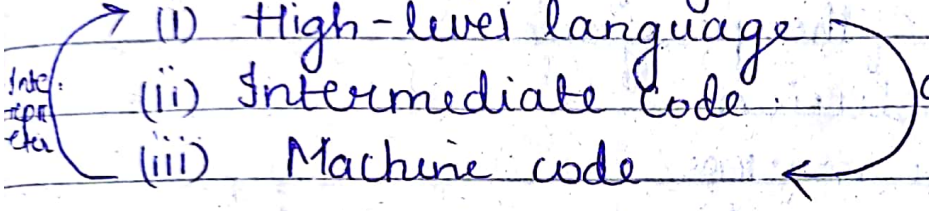
1) Difference between compiler and interpreter.

Faster - compiler.

Language Processors / Translators



→ Translation of a Program



Compiler can convert HLL to Intermediate code like Java byte code and then this code is platform independent (no hardware or OS dependency).

Compiler that convert from one level to the same level (can be HLL to HLL or MC to MC, etc) is known as source to source compile (STSC)

STSC can be used in legacy code (code with different versions).

Cross-compiler - If object code is generated in one OS (windows) then that object code can run in another OS also (Ubuntu).

→ Program after its creation till its execution goes through various stages.

Preprocessing → Compiler → (may) may not be the generation of Assembly lang → linker → loader.

↳ # is used to signify.
↳ Start of the program
→ file inclusion.
→ define macros.
→ pre-processor directives.
→ Eg:- #define A 20-10
#define B 30-10
void main ()
{
int c;
C = A * B
cout << c
}

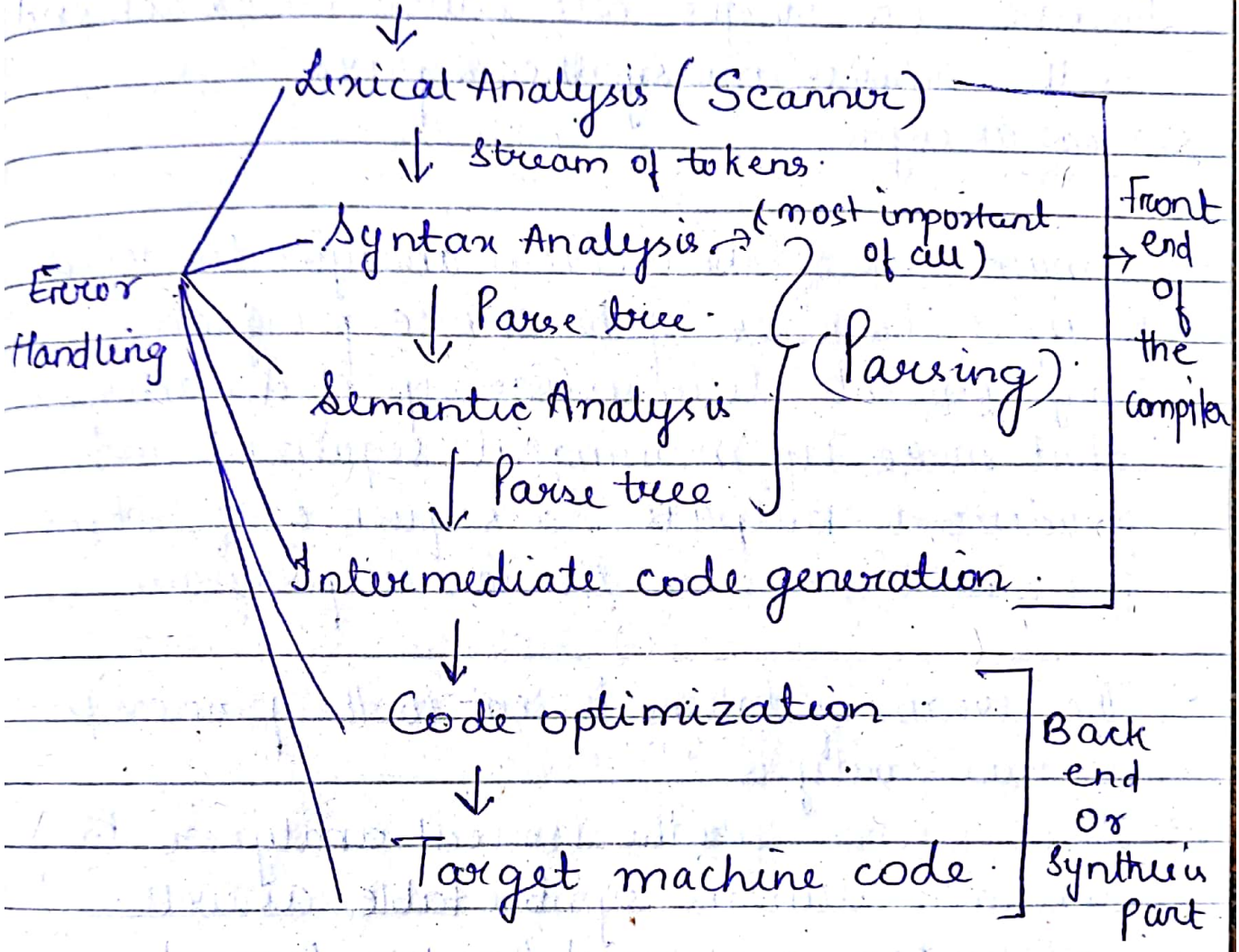
C = 20-10 * 30-10
= -290.

To avoid these errors we use.
( ) parenthesis.

#define A (20-10)
#define B (30-10)

lexicon → dictionary.

Compiler



At each stage there is a mechanism of Error handling and manage Symbol table.

Why generation of ALL ? (It is always not necessary)

→ Easy to generate and debug.

→ It is easy to map from Assembly language to MLL. (as ALL is machine dependent so there is one to one mapping)

24/7/18

The first four stages are called the front end of the compiler or analysis part.

The last two stages are called the back end of the compiler or synthesis part.

Lexical analysis

→ The main task of the lexical analyser to read the input character of the source program, group them into lexemes (group of characters that make up meaningful sequence) and produce as outputs - a sequence of tokens for each lexemes in the source program.

→ The stream of tokens is sent to the parser for syntax analysis.

→ It is common for the lexical analyzer to interact with the symbol table as well.

→ Other task performed by the lexical analyzer are ① removing comments
② removing white spaces.

Symbol table

These are data structures that are used by compilers to hold information about source program constructs. The information

is collected incrementally by the analysis phases and used by the synthesis phases to generate the target code.

Tokens

<token-name, attribute-value>
(this is optional).

(i) Tokens for keywords:

→ tokens have no attribute-value.

→ Eg:, if characters i, f — if → token name for if

Information description/Pattern

<if> → token

→ else characters e, l, s, e.

<else> → token.

* Keywords have keyword as token.
ie, if has token if.
else has token else.

(ii) Tokens for operators

→ Operators also don't have attribute-value but can be there if there is a class of operators

Eg:- + <+>
- <->
= <=>
* <*>

We can write the following also but we usually don't write:

TOKEN Name	Pattern	e.g.;
mul-op	*	*

We can put all operators in a class also, suppose operators.

operators <+, -, *, =, ···>

If we have to send + - then it will be a problem, as which operator to assign, then we use attribute value.

Eg:- <Operators, +>

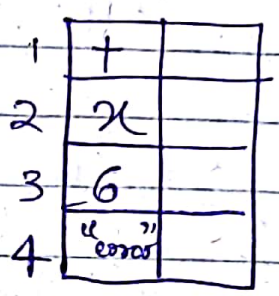
But generally we don't give

attribute value as + or - or etc.
So, here we use symbol table.

So; now our token becomes.

<operators, pointer to symbol table>

<operators, 1>.



(iii) Tokens for constants / literals.

TOKEN	Pattern
① Numbers	0, 1, 1.25 ...
② literals.	" anything between "

① <numbers, pointer to symb. table (PTST)>

<numbers, 3>

② <literals, PTST>

<literals, 4>.

(iii) Tokens for identifiers

TOKEN	Pattern
id.	$l(l+d)^*$

<id, pointer to symbol table>

<id, 2> for x variable.

(iv) Tokens for punctuation symbols.

; , ,

(We can make class of punctuation also, so there we can give att. value)

Token → <;>

no-attribute value for punctuation symbol

Eg:- a = b + c * 60;

<id, 1> <=> <id, 2> <+>

<id, 3> <*> <number, 4>

<;>

1	a
2	b
3	c
4	60
5	
6	

Chap 3, 3.1.1 (Ullman)

Divide the following C++ programs into tokens.

```
float limitedSquare(x) float x;
{
  /* Returns x-squared, but never more
     than 100 */
  return (x <= -10.0 || x >= 10.0)?
                    100 : x * x;
}
```

Sol -

<float> <id,1> <(> <id,2> <)>
<{> <float> <id,3> <;>

<return> <(> <id,4> <<=>
<number,5> <||> <id,6>
<>=> <number,7> <)>
<?> <number,8> <:> <id,9>
<*> <id,10> <;>
<}>

1	Limited squa
2	x
3	x
4	x
5	-10.0
6	x
7	10.0
8	100
9	
10	

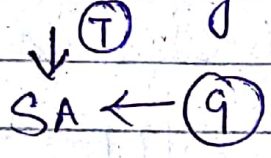
* When operators are coming together you take input chain possible (taking different is not wrong)

* We write x again and again in SYMT in previous example for effective error generation and bug fixing issues. (Later on we will study).

27/1/18

Syntax Analysis.

→ We give grammar as input to the syntax analyzer. (Here we give stream of tokens as input to the syntax analyzer).



$$\begin{cases} E \to E+E \\ \quad | \; E*E \\ \quad | \; id \end{cases}$$

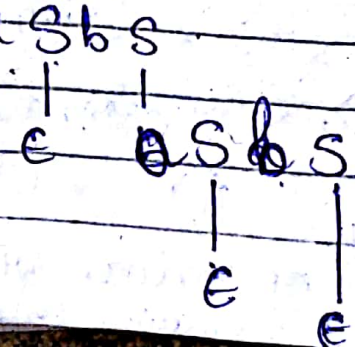
id + id * id and also (Ambiguous) id+id+id.

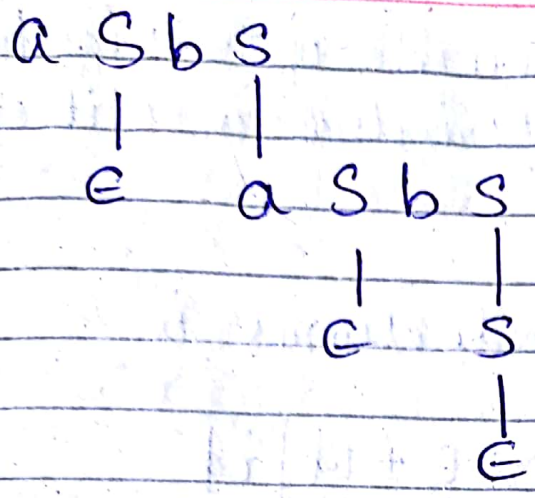
$$S \to aS \mid Sa \mid a$$

w = aa (Ambiguous)

$$S \to aSbS \mid bSaS \mid \epsilon$$

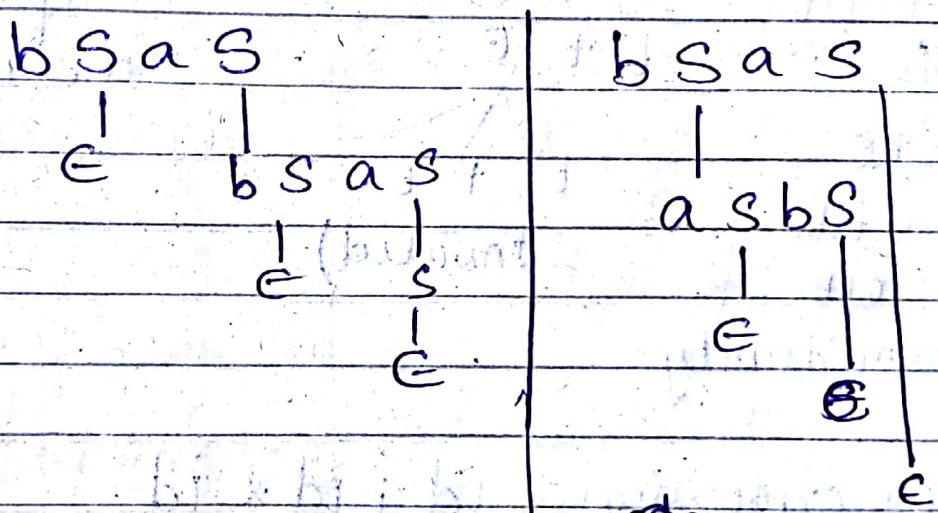
abab





Two parse trees for abab (✓) Ambiguous.

(ii) baba



Two parse trees for baba (Ambiguous).

To which operator the operand is associated is known as associativity.

(id + id) + id — left associativity

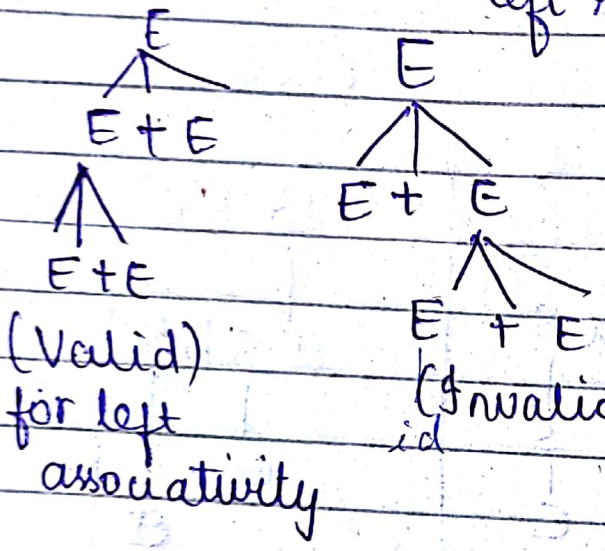
id + (id + id) — right associativity

When the leftmost variable of the righthandside is same as the lefthandside is called left recursion.

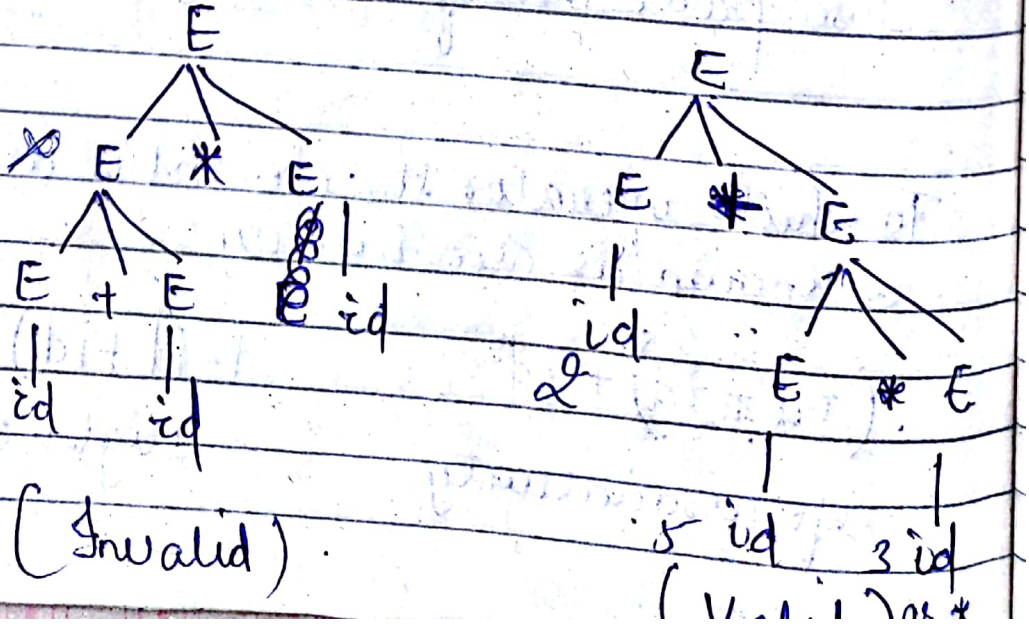
Now our production is

$E \rightarrow E + id \mid id$

left recursion (i.e, tree will only grow on left side and we restrict right side)

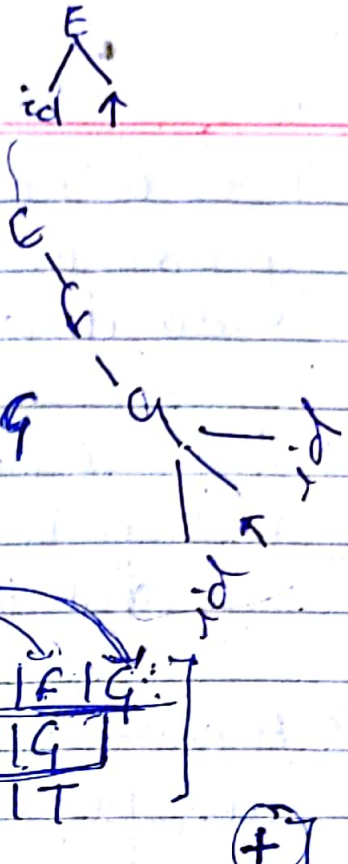


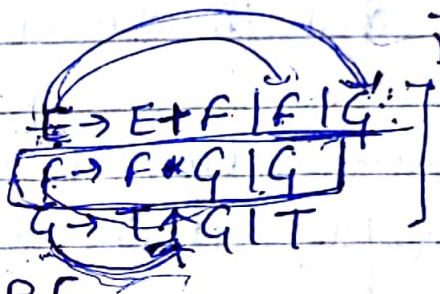
For expression id + id * id



Now our production becomes as after adding exponent (as it is of high property)

$E \rightarrow E+F \mid F$
$F \rightarrow F*G \mid G$
$G \rightarrow T \uparrow G \mid T$
$T \rightarrow id$

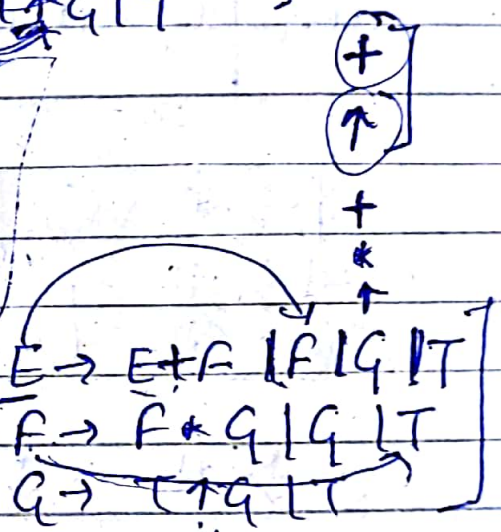




Q BE → BE or BE
| BE and BE
| not BE
| T
| F

(i) id or id and id

$E \rightarrow E \text{ or } F \mid F$
$F \rightarrow F \text{ and } G \mid G$
$G \rightarrow \text{not } G \mid \text{True} \mid \text{false}$



Q $E \mid E*F \mid F+E \mid F$

$E \rightarrow F-F \mid id$

* and + are at same level (so of same precedence)

$- \;\dot{>}\; * \doteq +$

is higher — Higher precedence — same precedence

→ There is no algorithm to find that a grammar is ambiguous. We have to do it in a hit and trial way.

→ There is no algorithm to convert ambigous grammar to unambiguous one. We need to do it manually.

→ Associativity is connected to recursion and precedence is associated with levels.

$:> $
:>
@ :> @
} left associative

@ :> # :> $ (Precedence of @ > # > $)

In eg:- A → A $ B | B
B → B # C | C
C → C @ D | D
D → d.

30/7/18

Recursion

Left-recursion

$A \rightarrow A\alpha \mid \beta$

eg; $E \rightarrow E + id \mid id$

α = $+id$, β = id

Language generated $= \beta\alpha^*$

Right recursion

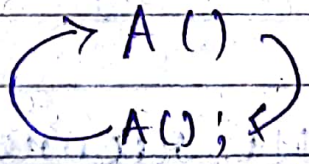
$A \rightarrow \alpha A \mid \beta$

$E \rightarrow id + E \mid id$

α = $id+$, β = id

Lang. generated $= \alpha^* \beta$

→ Top down parser cannot handle left recursion.



α; ← Terminating condition.

As recursion is previous to terminating condition so this is a continous loop.

Eliminating Left Recursion.

Top down parsers doesn't work properly if the grammar is left recursive. Therefore, we want to eliminate the left recursion without changing the

grammar or lang. generated by this grammar.

$$\boxed{\begin{aligned} A &\longrightarrow \beta A' \\ A' &\longrightarrow \epsilon \mid \alpha A' \end{aligned}} \iff \boxed{A \to A\alpha \mid \beta}$$

Eg:- $E \to E+T \mid T$

$E \to \underbrace{E+T}_{\alpha} \mid \underbrace{T}_{\beta}$

$E \to TE'$

$E' \to \epsilon \mid +TE'$ $\Big\}$ → Ans.

E.g.; Eliminate the left recursion from the grammar.

$S \to \underbrace{SOSIS}_{\alpha} \mid \underbrace{01}_{\beta}$

$S \to 01S'$

$S' \to \epsilon \mid OSISS'$ $\Big\}$ → Ans

Eg:- $E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

$E \rightarrow TE'$

$E' \rightarrow \epsilon \mid +TE'$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \epsilon$

$F \rightarrow (E) \mid id$

---

* If Incase our production is like:

$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \beta_n$

Then our new productions will be:-

$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \beta_3 A' \mid \cdots \mid \beta_n A'$

$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \alpha_3 A' \mid \cdots \mid \alpha_m A' \mid \epsilon$

Till now, we have classified grammars as:-

Grammars

- Ambiguous
- Unambiguous (✓)

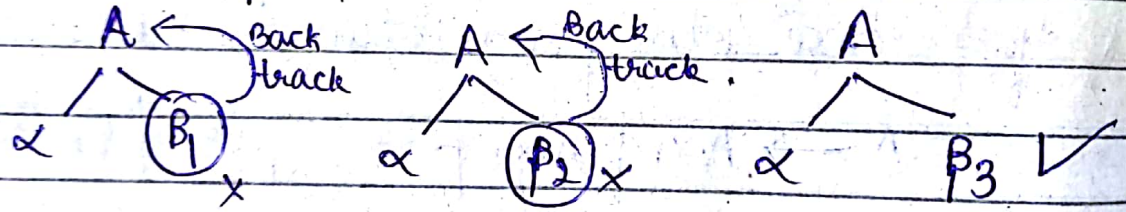
G

- left-recursive (✗)
- right-recursive

G_1

- Deterministic (✓)
- Non-deterministic.

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3$.



The backtracking is happening because of the common prefixes at one or more productions at the RHS.
We are making the decision based on α while we should be making decisions based on β_3, so in such problems, we will postpone the decision making process and it is called left factoring procedure or eliminating non-determinism.

Now our new productions,

$$A \to \alpha A'$$
$$A' \to \beta_1 | \beta_2 | \beta_3 .$$

Q $S \to iEtS | iEtSeS | a$.

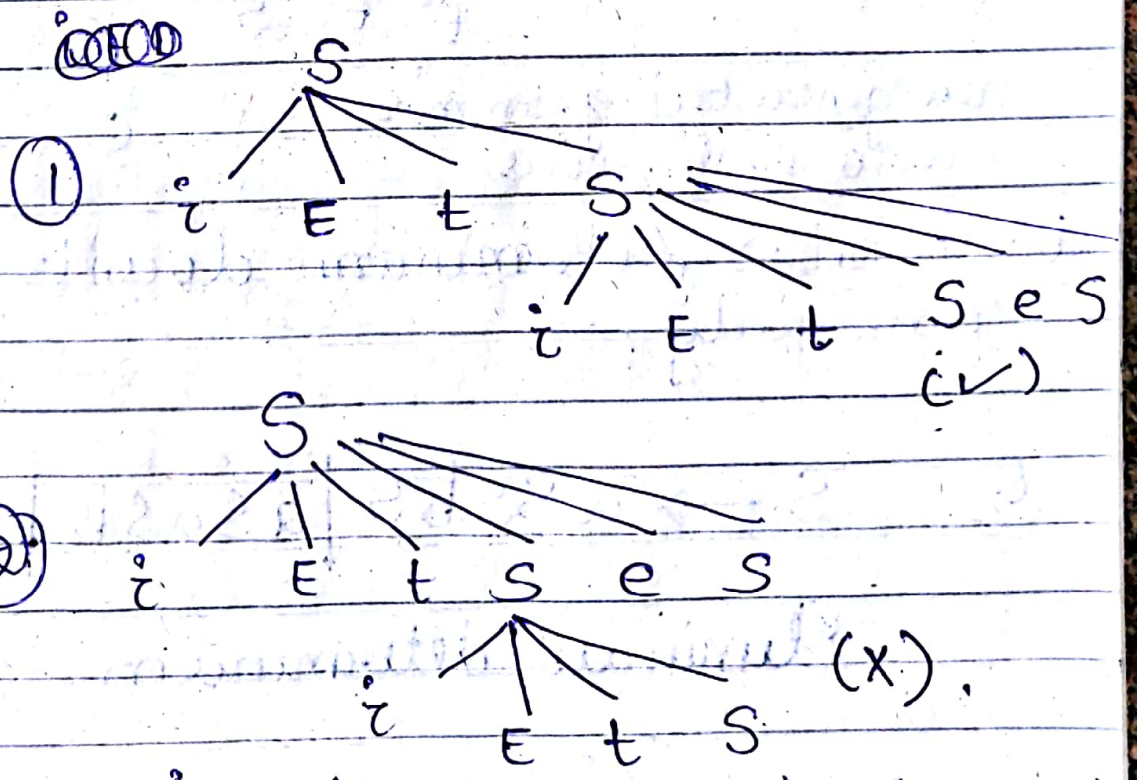
$E \to b$

$S \to iEtSS' | a$.

$S' \to \in | eS$

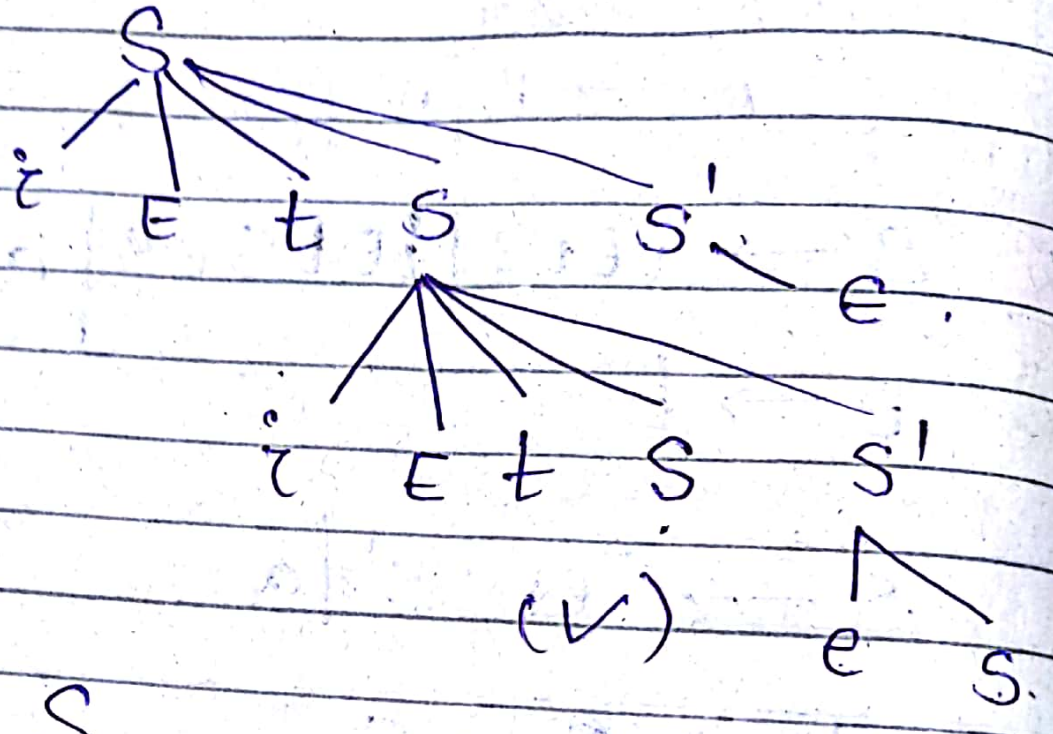
$E \to b$

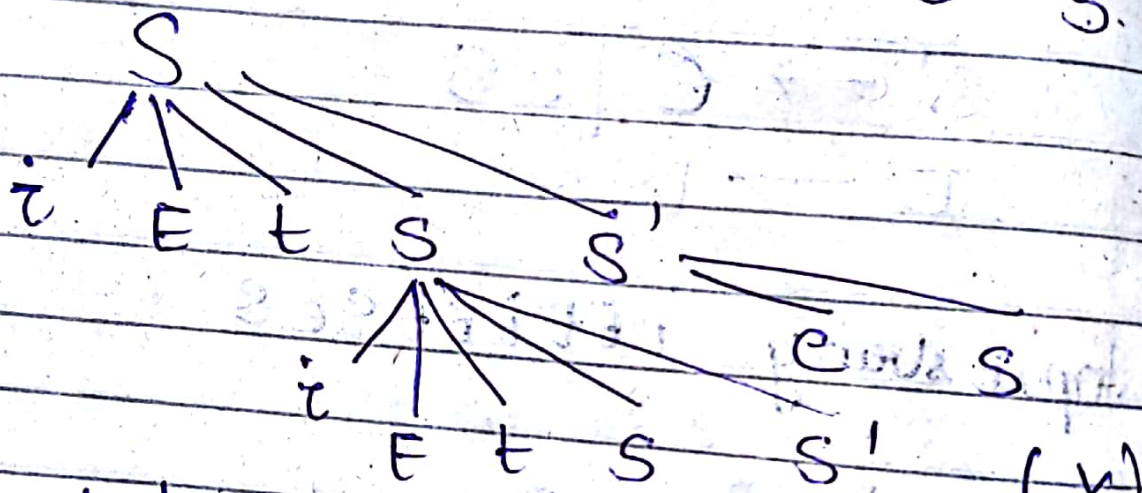
Input string iEtiEtSeS.



Grammar is ambiguous (for original grammar)

for new generated grammar.

① 

② 

The new generated grammar is also ambiguous.

Conclusion → Determinism doesn't eliminate ambiguity.

Q S → aSSbS | aSaSb | abb | b

Eliminate determinism.

① $S \rightarrow aS' \mid b$

$S' \rightarrow SSbS \mid SaSb \mid bb$

② $S' \rightarrow SS'' \mid bb$

③ $S'' \rightarrow SbS \mid aSb$

Final we got 3 productions.

eg:- $A \rightarrow aA$

$B \rightarrow aB$.

Even though we have same prefix, but this grammar is deterministic because we know that from A we have to go at A terminal and from B we have to go to B terminal.

Q $S \rightarrow bSSaaS \mid bSSaSb \mid bSb \mid a$.

① $S \rightarrow bSS' \mid a$

$S' \rightarrow SaaS \mid SaSb \mid b$.

② $S' \rightarrow SaS'' \mid b$

③ $S'' \rightarrow aS \mid Sb$

Finally, we got 3 productions.

31/7/18.

Parsers

- Top-Down Parsers
  - with full Backtracking
  - without backtracking (They don't need Left recursion)
    - Recursive Descent
    - Non-Recursive Descent
      - Eg: LL(1)
    - Operator Precedence (Only this parsers can handle ambiguous grammar)
- Bottom-Up Parsers (SR parsers (shift reduce))
  - LR Parsers
    - LR(0)
    - SLR(1)

If there is an input string, there will always be a $ sign to mark the end of the input string.

eg; abcde$

{always gives the first char. of string}

There are two functions FIRST() and FOLLOW().

		FIRST()
eg:-	S → aABCD	{a}
	A → b	{b}
	B → c	{c}
	C → d	{d}
	D → e	{e}

Suppose	S → ABCD	{b, c}
	A → b/ϵ	{b, ϵ}

* ϵ will only be written where it is generated.

		FOLLOW()
eg.-	S → ABCD	{$}
	A → b/ϵ	{c}
	B → c	{d}
	C → d	{e}
	D → e	{$}

Follow of A will be first of B and

follow of B is first of C; Follow of C will be first of D; Follow of D will be follow of S.

If $A \to BD$ and A is not the start symbol, then follow of D is follow of A (in every case it won't be $).

* Follow gives the last character of that string; and of non-terminal; the one which follow it in the start production.

Q

	FIRST()	FOLLOW()
$S \to ABCDE$	{a, b, c}	{$}
$A \to a \mid \epsilon$	{a, ε}	{b, c}
$B \to b \mid \epsilon$	{b, ε}	{c}
$C \to c$	{c}	{d, e, $}
$D \to d \mid \epsilon$	{d, ε}	{e, $}
$E \to e \mid \epsilon$	{e, ε}	{$}

Q.

	FIRST()	FOLLOW()
$S \to Bb \mid Cd$	{a, b, c, d}	{$}
$B \to aB \mid \epsilon$	{a, ε}	{b}
$C \to CC \mid \epsilon$	{c, ε}	{d}

Points:-

① In FIRST() we get ∈.
② In FOLLOW() we never get ∈.
③ FOLLOW() of start Symbol is always $.
④ Whenever a variable is at the right hand of the production and there is nothing after it, then its FOLLOW() is the FOLLOW() of left hand side.

Why FIRST() ?

① To eliminate the need of backtrac-king.
② If the compiler would have known in advance that what is the first character of the string produced when a production is applied then by comparing it to the current character or token in the input string, it sees, it can wisely take decisions on which production to apply.

Q Why FOLLOW()?

① If the parser knows what is the terminal that can follow a variable in the process of derivation then may be by the use of look ahead operator it can make a non-terminal to vanish out (ε if needed) to generate the string from the parse tree.

Q

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \epsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \epsilon$

$F \rightarrow id \mid (E)$

	FIRST()	FOLLOW()
E	{id, (}	{$,)}
E'	{+, ε}	{$,)}
T	{id, (}	{+, $,)}
T'	{*, ε}	{+, $,)}
F	{id, (}	{*, $, +,)}

* If our start symbol appears in the right hand side, then we will see that which terminal / variable follows the start symbol.
If terminal is there then no problem, if variable we will see whether it is vanishing or not.

3/8/18

LL(1) Parser.

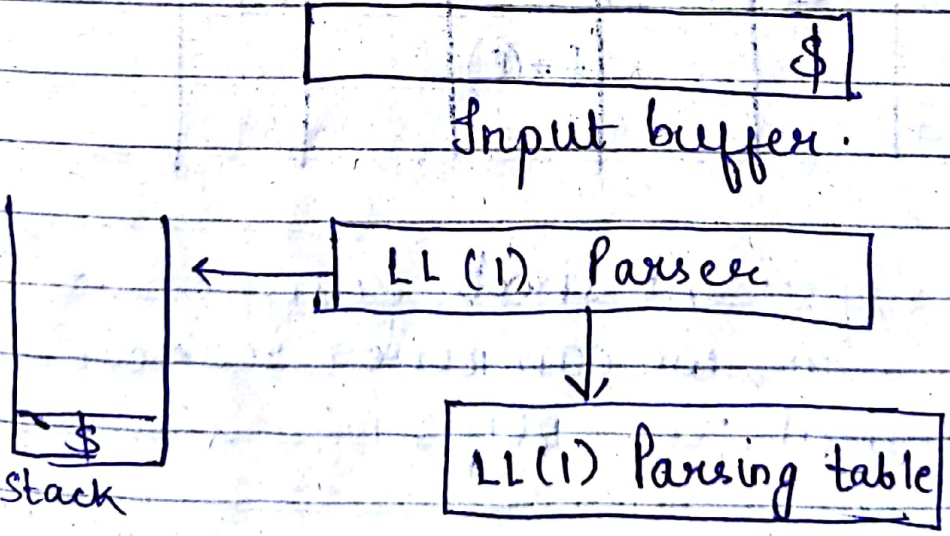
→ It is a top down parser without backtra-cking.

→ 1st L means your input string will be processed from left to right.

→ 2nd L the parser will follow left-most derivation

→ 1→ whenever making a decision in which production is applied to, it totally depends on one input unit (OR) simply no:- of look aheads.

→ What are the components in parsing productions when LL(1) Parser is used?



★LL(1) grammars → when the parsing table is constructed, in any cell there will be not more than one rule.

	FIRST()	FOLLOW()
Q $E \to TE'$	{id, (}	{$,)}
$E' \to +TE' \mid \epsilon$	{+, ε}	{$,)}
$T \to FT'$	{id, (}	{*, +, $,)}
$T' \to *FT' \mid \epsilon$	{*, ε}	{+, $,)}
$F \to id \mid (E)$	{id, (}	{*, +, $,)}

In parsing table, rows will be the left hand side of the productions.
Colums will be the terminal symbols and $ except ε

	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)		

★ Empty cell represent that from that symbol in row, we can never reach the terminal in column.

★ Here, in this table we have seen that, no intersection point have more than one production, so it is LL(1) parser!

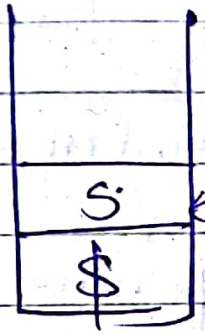
Q $S \to (S) \mid \epsilon$.

	FIRST	FOLLOW
	{(, ∈}	{$,)}.

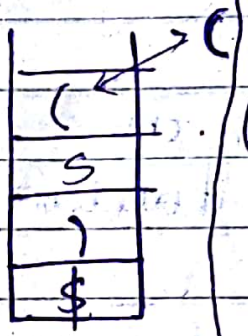
	(	)	$
S	S→(S)	S→∈	S→∈

Input string
(()) $
↑
Look ahead pointer.

(In LL(1) LP is looking at one character at a time and based on this it will make decision.)

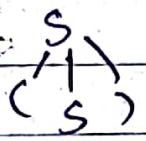
① 

Stack always start with $ then Start symbol.
(no move in forward direction, only applying rule)

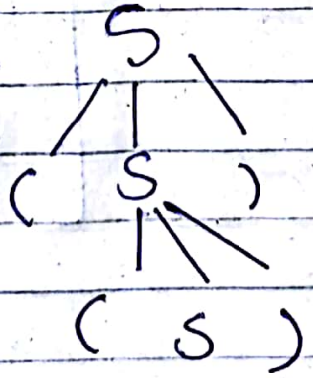
② 

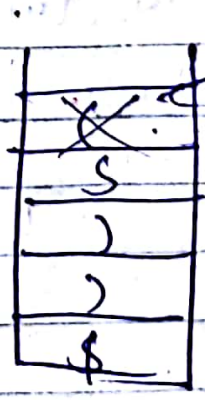
Match then pop. (terminal) no applying rule just move the pointer in forward direction

② and parse tree will be generated.



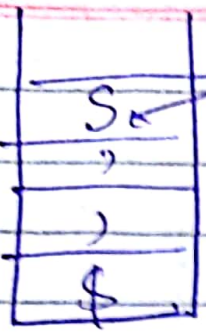
③ 



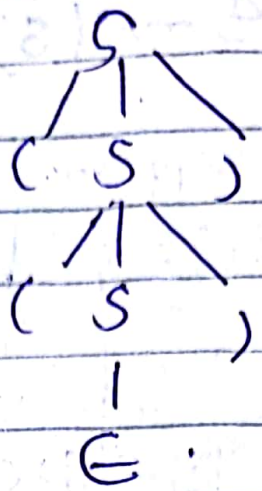
④ 

pointer move in forward direction

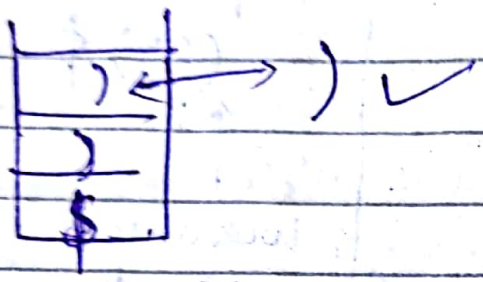
(5)



Apply Rule and generate parse tree

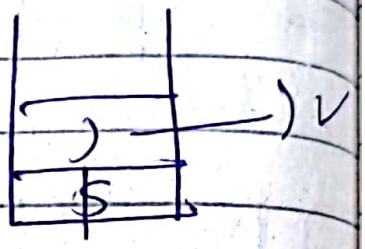


(6)



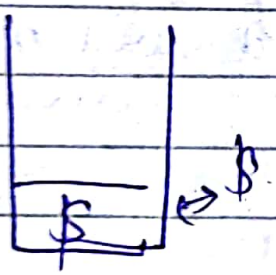
Pointer will move in forward direction.

(7)



Pointer will move in forward direction

(8)



Match, so our string will be accepted.

Note:- Generation of parse tree is just an abstract concept, it really doesn't happen in computer

Q ① $S \rightarrow AaAb \mid BbBa$.

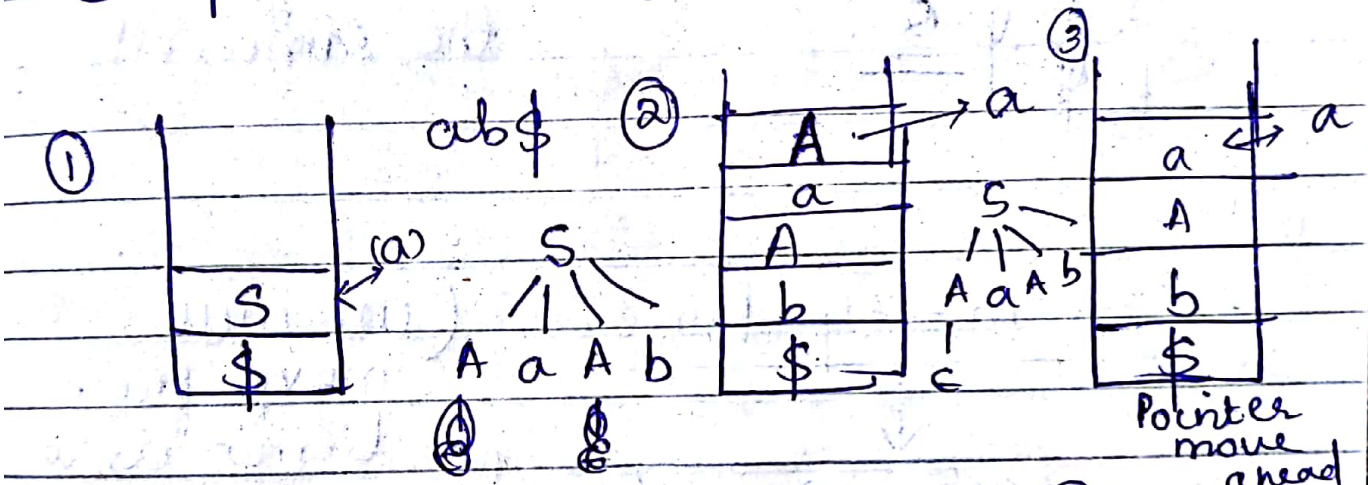
② $A \rightarrow \epsilon$

③ $B \rightarrow \epsilon$.

I/P string ab$.

	FIRST	FOLLOW
①	{a, b}	{$}
②	{ε}	{a, b}
③	{ε}	{b, a}

	a	b	$
S	S→AaAb	S→Bb, Ba	
A	A→[illegible]	A→ε	
B	B→ε	B→ε	



Q $S \rightarrow aSbS \mid bSaS \mid \epsilon$

	First	Follow
S	{a, b}	{b, a, $}

	a	b	$
S	$S \rightarrow aSbS$ $S \rightarrow \epsilon$	$S \rightarrow bSaS$ $S \rightarrow \epsilon$	$S \rightarrow \epsilon$

Not LL(1) grammar as two rules are together in the same cell.

Q.

6 | 8 | 18

Bottom-Up Parsers (We will move from terminals to start symbol)

- Shift / Reduce
  - LR Parsing
    - SLR — Simple LR
    - CLR — Canonical LR
    - LALR — look ahead LR.

LR :- L stands for Scanning input from left to right.

R stands for rightmost derivation in reverse.

They donot do recursion or back-tracking. This is the advantage of LR Parsers.

SLR (Simple LR Parsing)

↳ Considers smallest set of grammar.

↳ fewer number of states, hence smaller table size.

→ Here there is no concept of look ahead symbol.

CLR (Canonical LR)

→ large no: of states, hence larger parsing table.

→ Considers widest set of LR(1) grammar.

→ Here we consider one look-ahead symbol.

LALR.

→ Same number of states as SLR.
→ considers intermediate size of grammar.
→ Here we don't use FIRST() and FOLLOW() instead there are different parsing techniques.

Eg1 $S \rightarrow E\$$
$E \rightarrow id$
$E \rightarrow (T)$
$T \rightarrow T+E$

$S \rightarrow \cdot E\$$

$S \rightarrow \cdot id\$$
$S \rightarrow \cdot (T)\$$

(Anything left to • means they are processed and are in stack and right of • means they are to be processed).

1st state → S_0.

After this we have processed E and have shifted the •.

$S \rightarrow E \cdot \$$

(further will be done later)

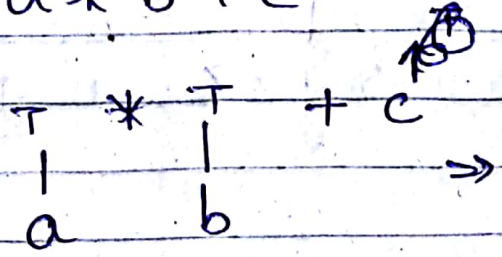
LL	LR
① Uses leftmost derivation.	① Uses rightmost derivation in reverse.
② Top-down.	② Bottom-Up.
③ Pre order traversal of parse tree.	③ Post order traversal of parse tree.
④ Starts with a non-terminal.	④ Starts with handling (or reducing) terminals.
⑤ Ends at the terminal.	⑤ Ends at the start symbol.
⑥ Here initially we have a stack.	⑥ Here it ends with an empty stack.

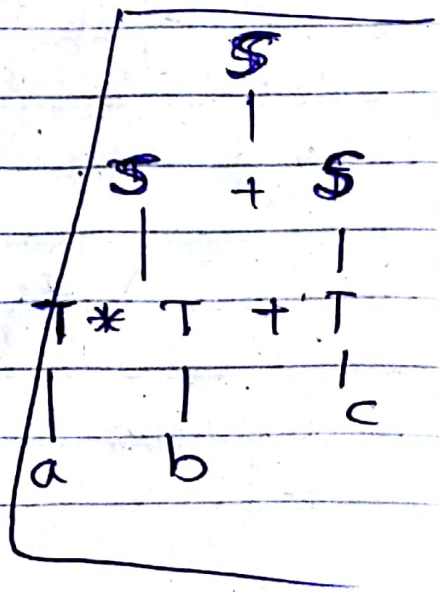
Eg:- LR Parser

$S \rightarrow S*S \mid S+S \mid T$

$T \rightarrow id$

I/P a * b + c



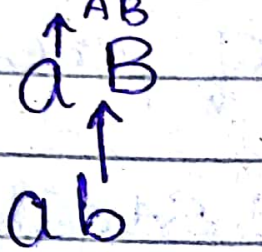


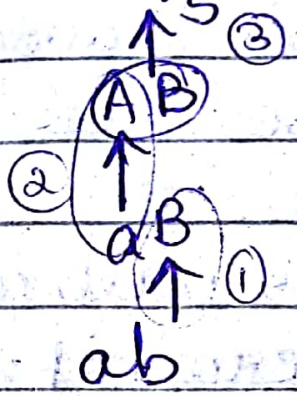
Handle :-

Eg:- $S \to AB$ $A \to a$ $B \to b$.

I/P "ab"

→ You find a match for the input string in the production.

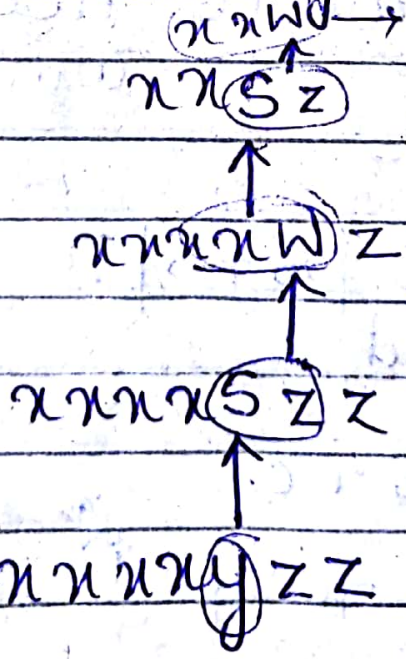




There are 5 handles here.

Eg:- $S \to xxW$ $S \to y$ $W \to Sz$

I/P "xxxyzz"



There are 5 handles.

Def of handle

It is a substring in the sentential form that is matching with complete RHS part of the grammar.

Augmented grammar

E.g. $S \rightarrow E\$$ OR $S' \rightarrow S$, $S \rightarrow E\$$ (Augmented grammar)

Eg: $G \begin{cases} S \rightarrow AB \\ A \rightarrow a \\ B \rightarrow b \end{cases}$ $G' \begin{cases} S' \rightarrow S \\ S \rightarrow AB \\ A \rightarrow a \\ B \rightarrow b \end{cases}$

→ Tell the parser to when to stop parsing and announce the acceptance. This is the purpose of augmented grammar.

→ The Action table can have one of the four forms:
① shift ② reduce ③ accept
④ error

States	Action: Terminals			Goto: Non Terminals		
	a	b	$	S	A	B

S' is ou introdu non-term -inal so we don't write here.

If we introducing • (dot) in the RHS part of the production, this is known as closure operation.

Eg: $S' \rightarrow S$
$S \rightarrow AB$
$A \rightarrow a$
$B \rightarrow b$

$S' \rightarrow \bullet S$
$S \rightarrow \bullet AB$
$A \rightarrow \bullet a$

I_0

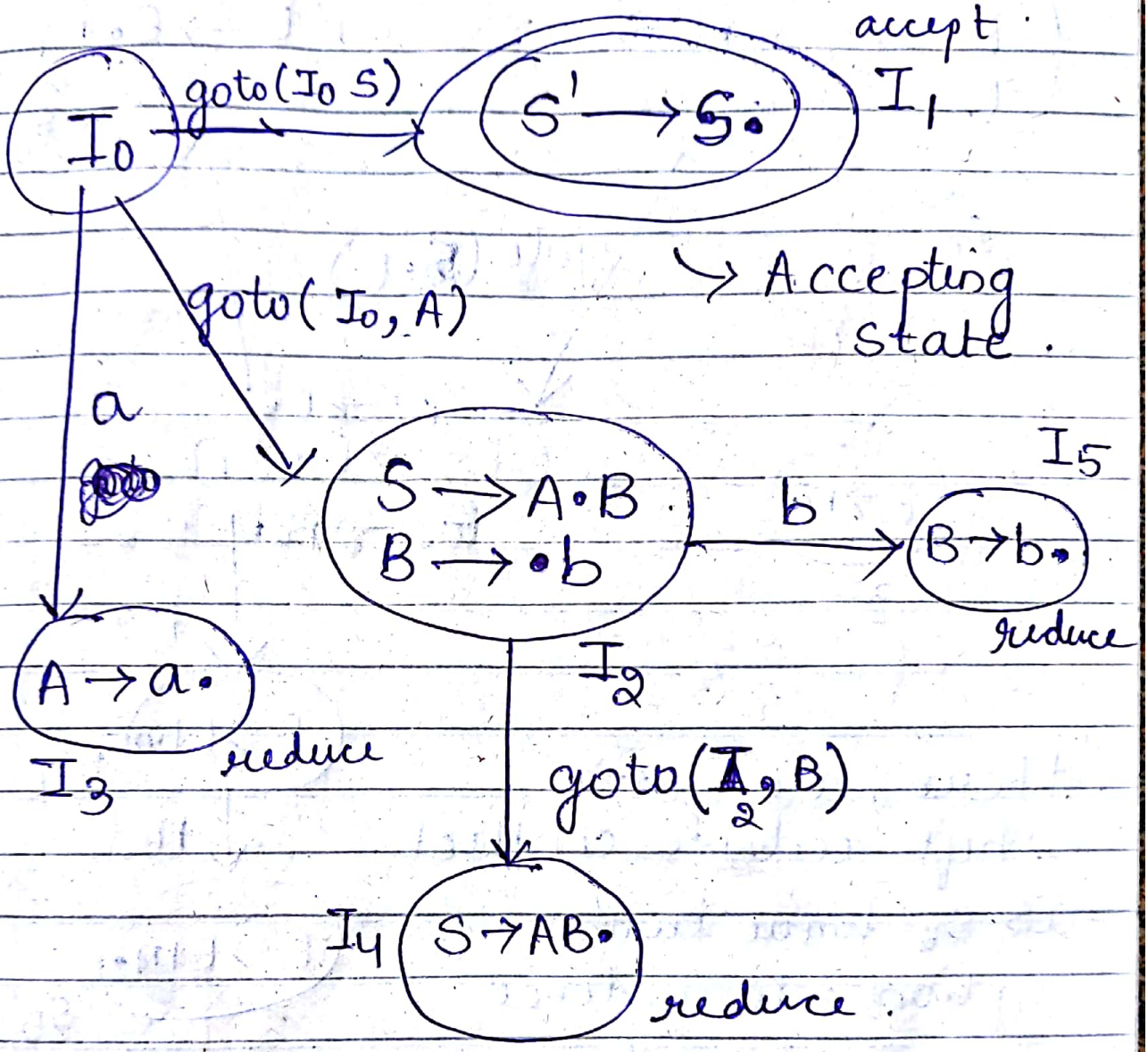
*If we have a non-terminal after • then we will check if it have any production associated with that non-terminal.

goto (1st arg, 2nd arg) = j

Previous state → 1st argument
2nd argument → Symbols it can encounter
j → Next state

→ goto can only have non-terminals.

goto (i, symbol) = j



Eg:-

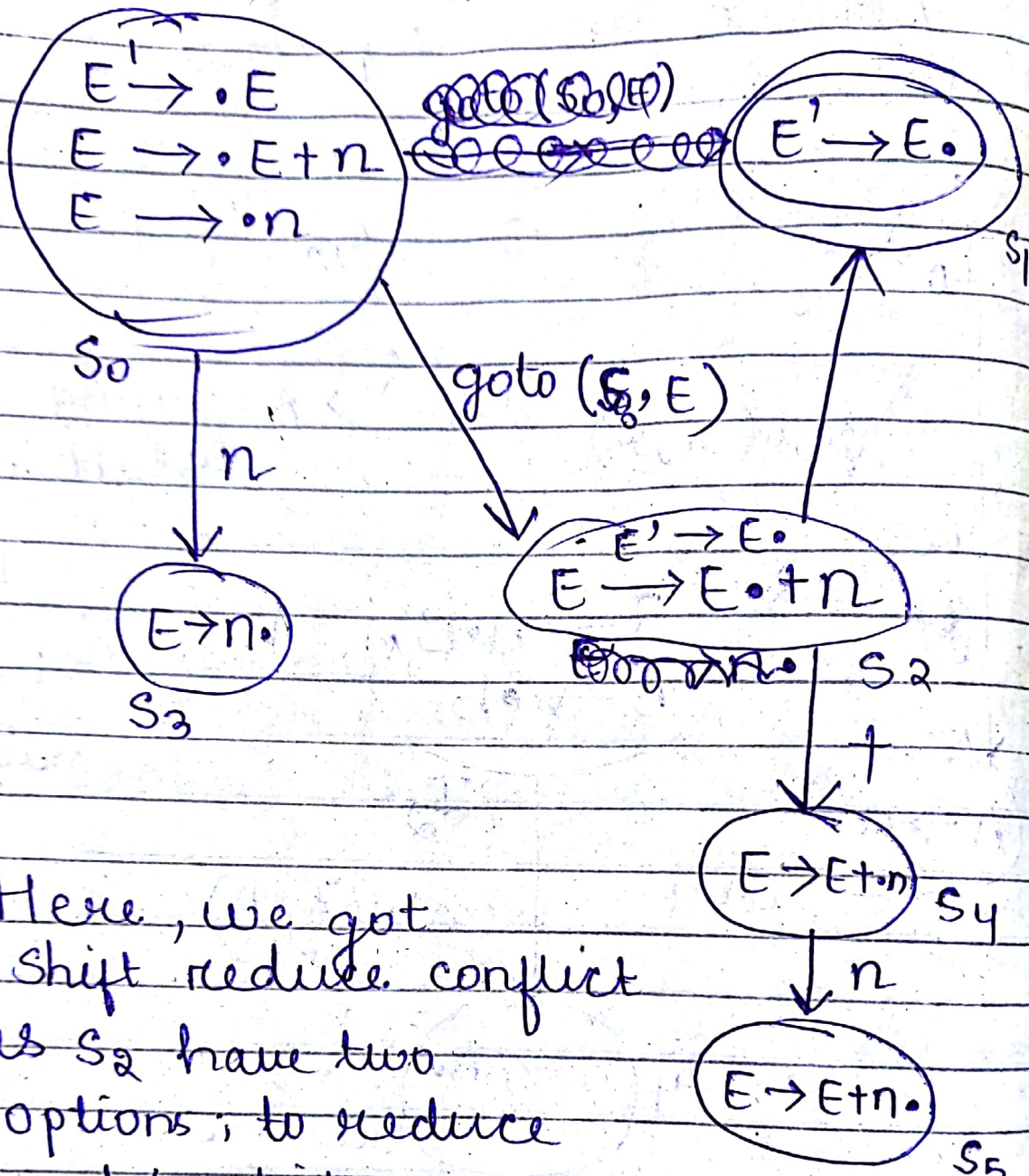
$E \longrightarrow E + n$.

$E \longrightarrow n$.

$E' \longrightarrow E$.

$E \rightarrow E + n$

$E \longrightarrow n$.

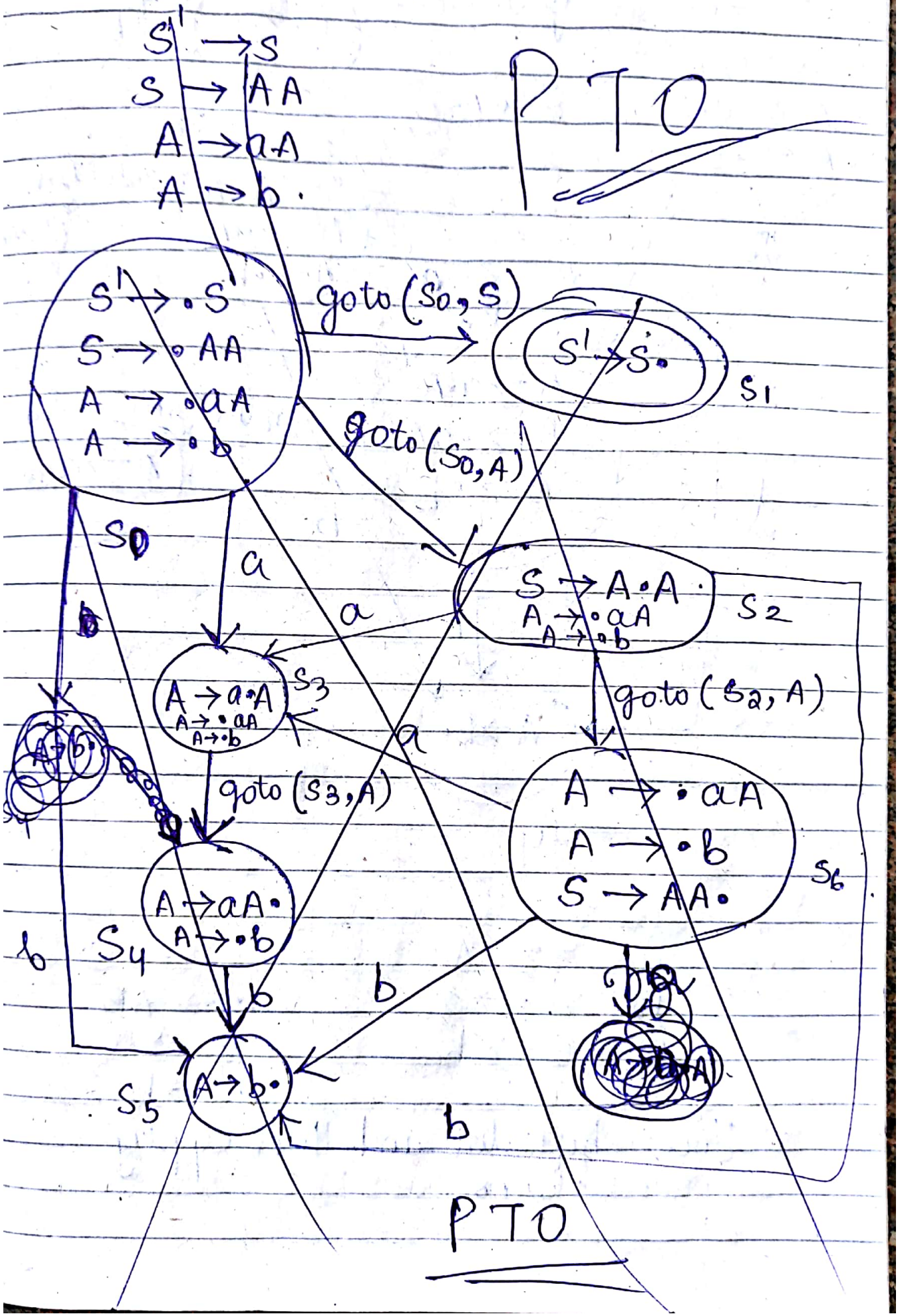


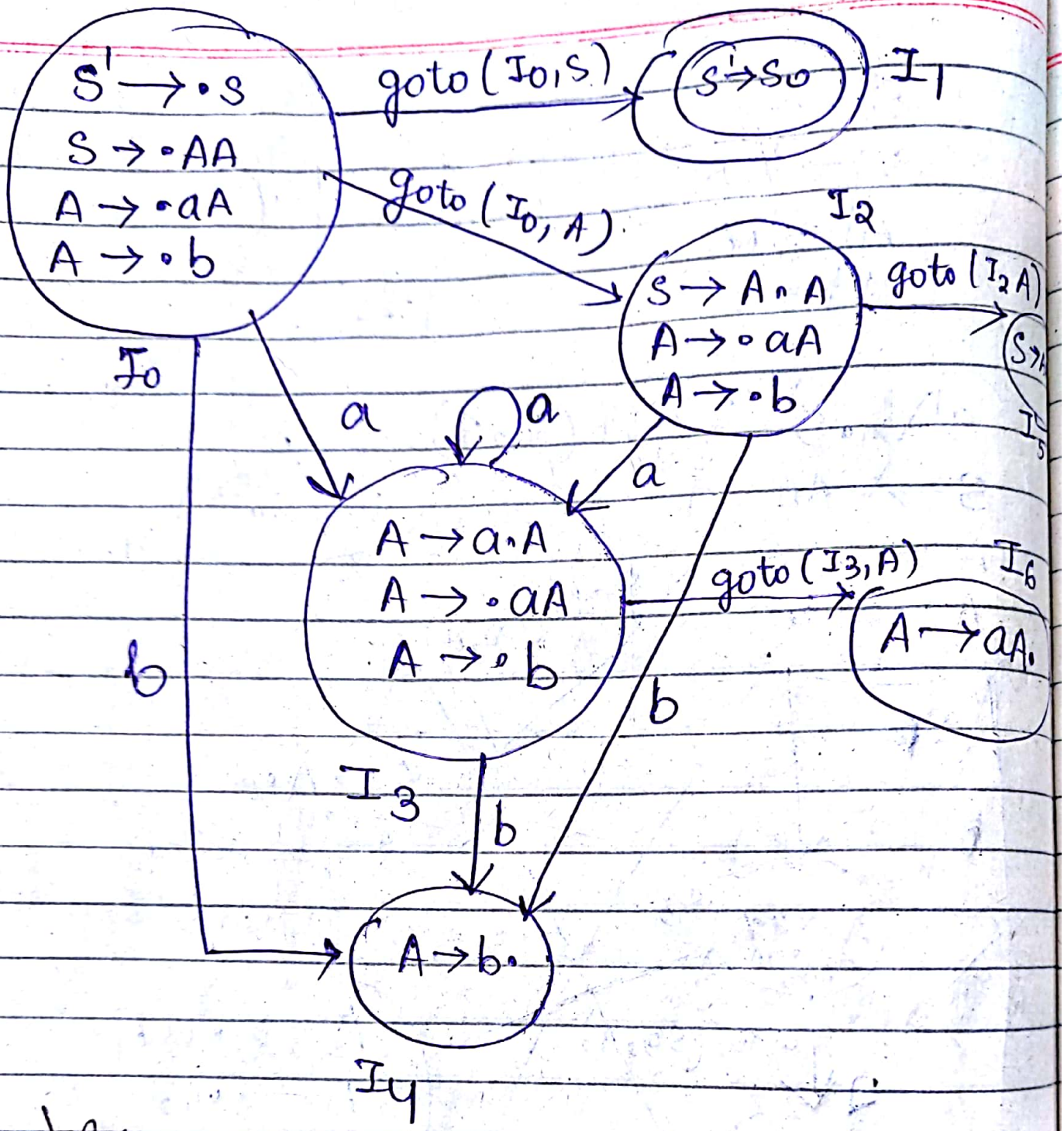
Here, we got shift reduce conflict as S_2 have two options; to reduce and to shift.

→ so there is a conflict that which one to carry.

Eg:-

$P_1 S \rightarrow AA$	$S' \rightarrow S$
$P_2 A \rightarrow aA$	$S \rightarrow AA$
$P_3 A \rightarrow b$	$A \rightarrow aA$
	$A \rightarrow b$



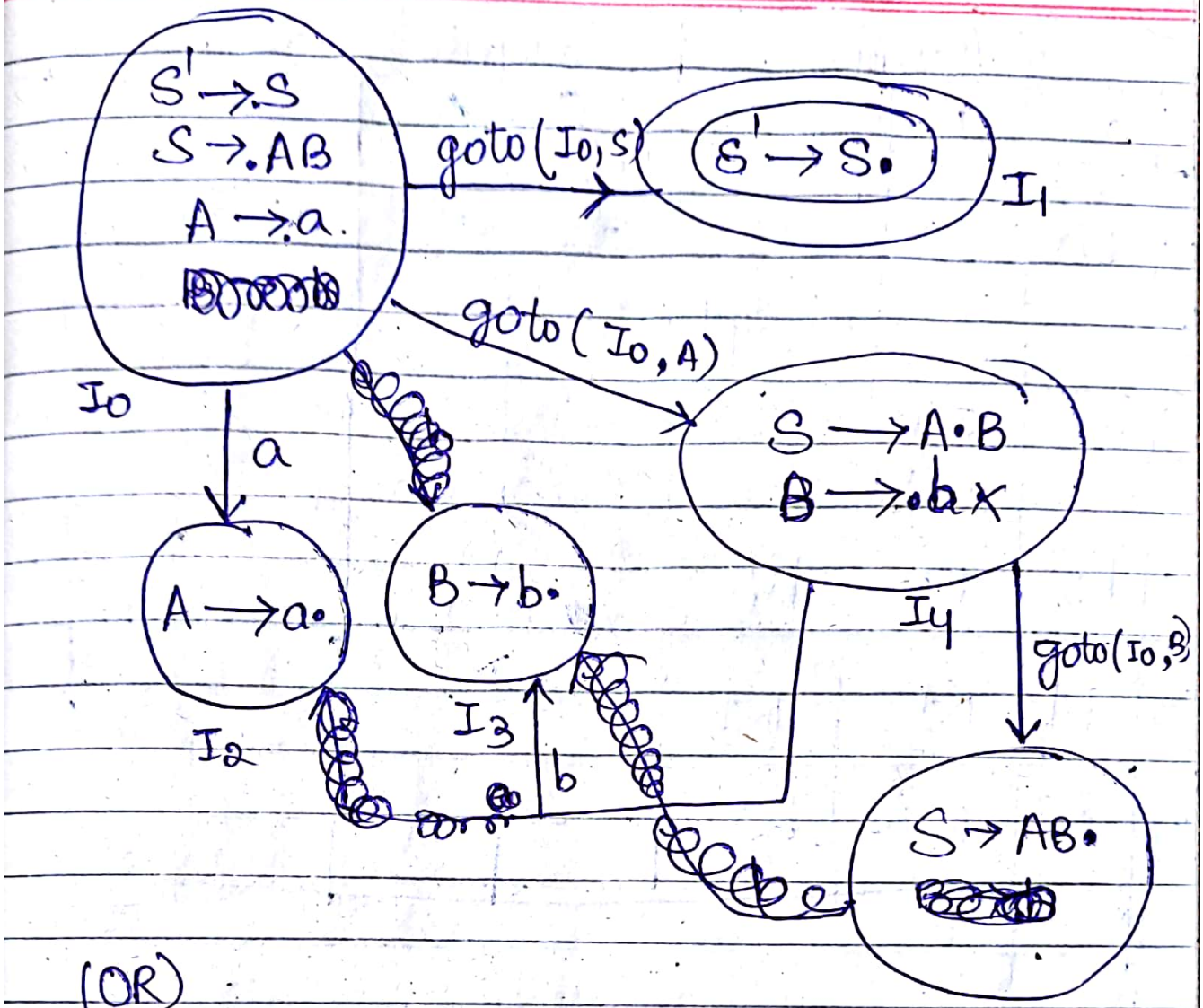


10/8/18

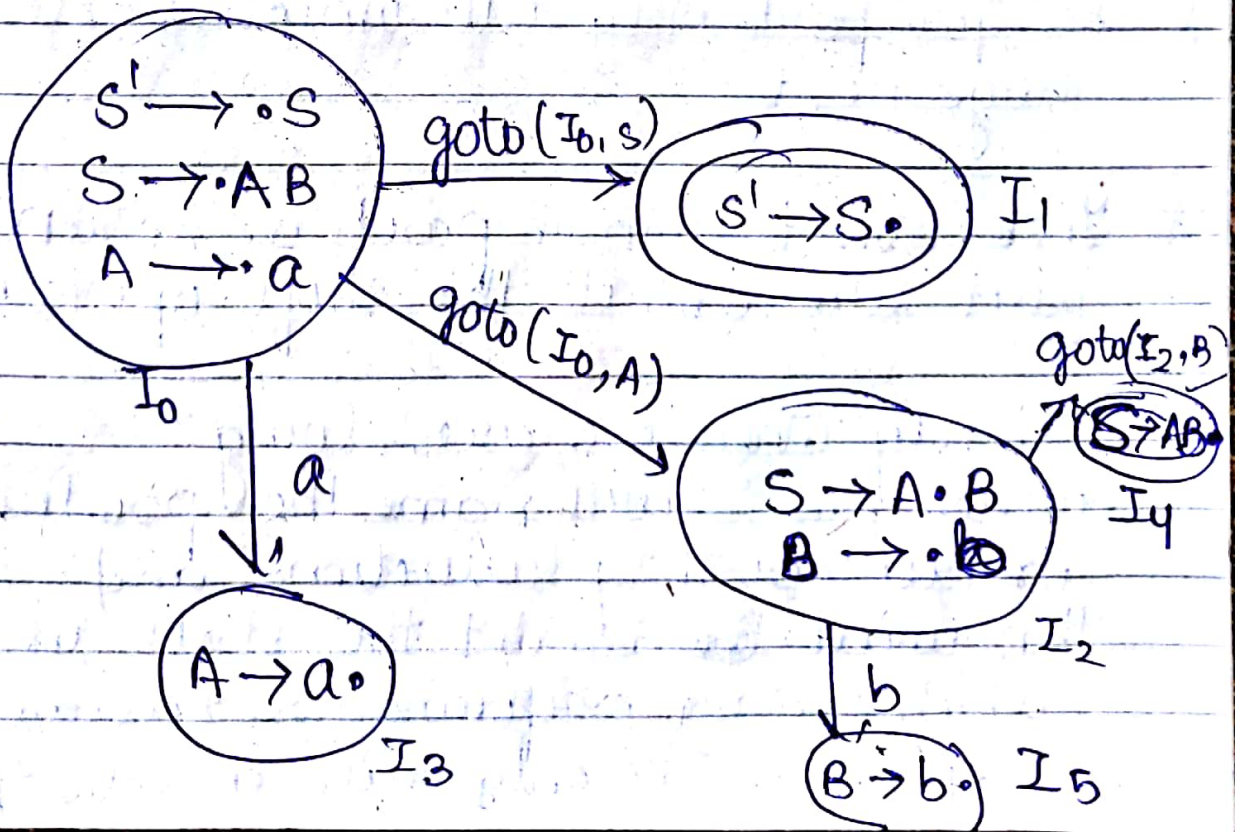
Eg:2 P_1 : $S \to AB$
P_2 $A \to a$
P_3 $B \to b$

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

★ First shift dot and then apply productions.



(OR)



LR(0)

	Action			Goto		
	a	b	$	S	A	B
I_0	Shift 3 (S3)			1	2	
I_1			accept			
I_2		Shift 5 (S5)				4
I_3	R2	R2	R2			
I_4	R1	R1	R1			
I_5	R3	R3	R3			

★ The Goto part only tells which state to go next.

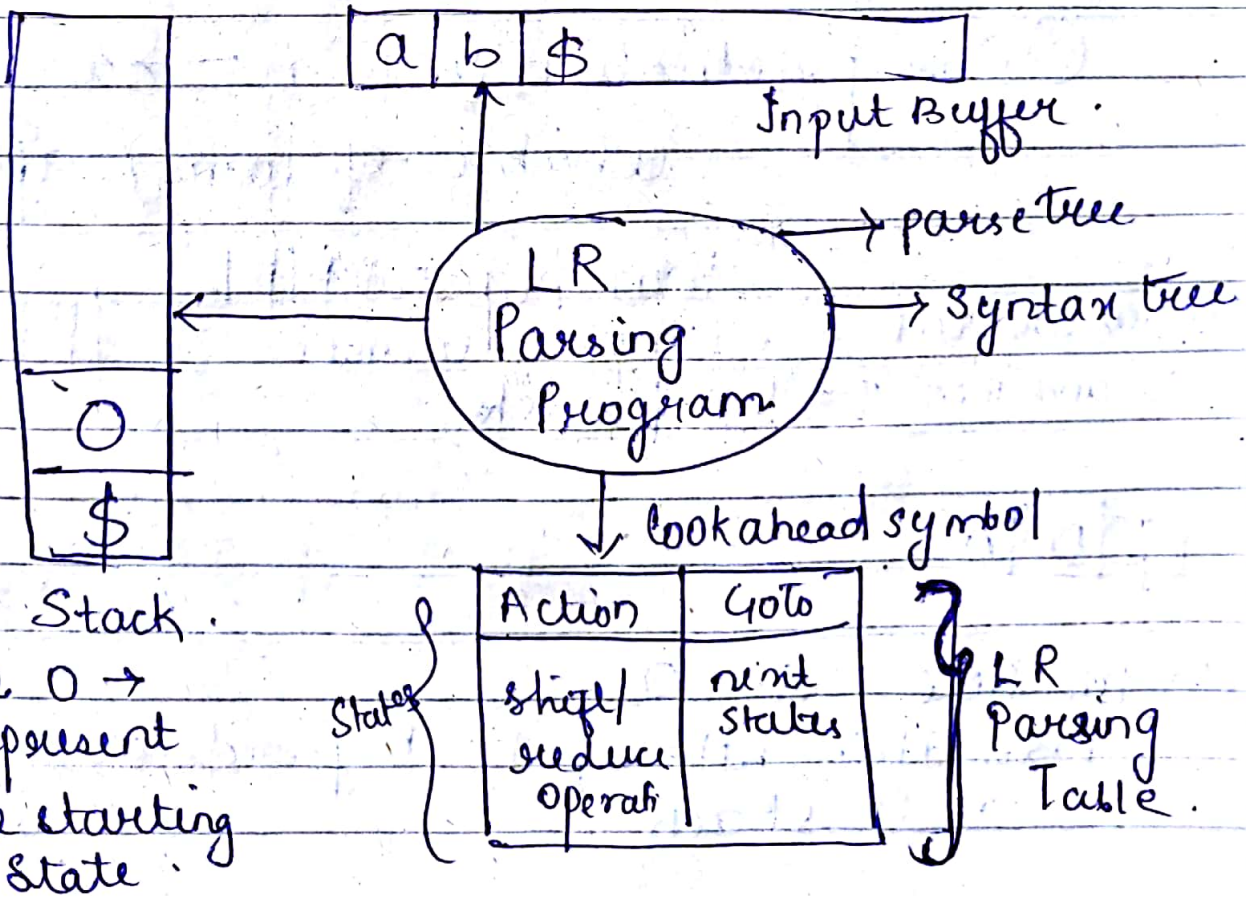
★ The terminal part or action part tells about the shift operation.

★ Finally when we are doing reducing, we will name the productions or number the productions and then write against the state in which we are performing reducing operation (we only write in action part)

Table for previous example (1)

State	Action			Goto	
	a	b	$	S	A
I_0	S3	shift 4 (S4)		1	2
I_1			accept		
I_2	S3	S4			5
I_3	S3	S4			6
I_4	R3	R3	R3		
I_5	R1	R1	R1		
I_6	R2	R2	R2		

Working of this table



Here 0 → represent the starting state.

Stack	Buffer	Action
\$ 0	a b \$	[0, a] = sh

Here we have an action shift 3.

① we move or push the input token to stack and then shift state on to stack.
and then move the pointer of the buffer to next token.

\$ 0 a 3	b \$	[3, b] = reduce 2

① Our production 2 is $A \rightarrow a$

which is of form $A \rightarrow \beta$

then pop = $2 * |\beta|$

$|\beta|$ = length of β

② Now we take a tempory variable S and then S = top of stack.

\$ 0 A

③ For here S = 0

④ Push the LHS of that production on to stack

(5) Then we PUSH goto (S, A)

= goto (0, A)

↓

gives us 2.

So now we will push 2 onto stack.

So now our stack is :

$	0	A	2

* In reduce we do not move the pointer of the buffer; so our buffer looks like

b	$

III

$	0	A	2

b	$

Action — [2, b] = shift 5.

IV

$	0	A	2	b	5

$

reduce 3

Pop b, 5 s = 2.

push B.

B → b

push goto[2, B) = 4.

LR Parsing Algorithm:

```
token = next_token()
repeat forever
    s = top of stack
    If action[s, token] = "Shift si"
    then
        PUSH token
        PUSH si
        token = next_token()

    else if action[s, token] = "reduce
                                A → β"
    then
        POP 2*|β| symbols
        s = top of stack
        PUSH A
        PUSH goto[s, A]

    else if action[s, token] = "accept"
    then
        return
    else
        error.
```

→ We write reduce under every column of action part; which is an unnecessary overhead so to overcom this we went to further parsers!

→ In goto we only mention the state or only the number of state to which we are going

→ R2 → reduce 2 (reduce by using prod. no 2)

3	B	1	B

Continuation of algo implementation

$	0	A	2	B	4

$

[4, $] - reduce 1

S → AB.

|β| = 2.

pop = 2 * |β|

= 4.

$	0

S = 0

Push Start symbol S

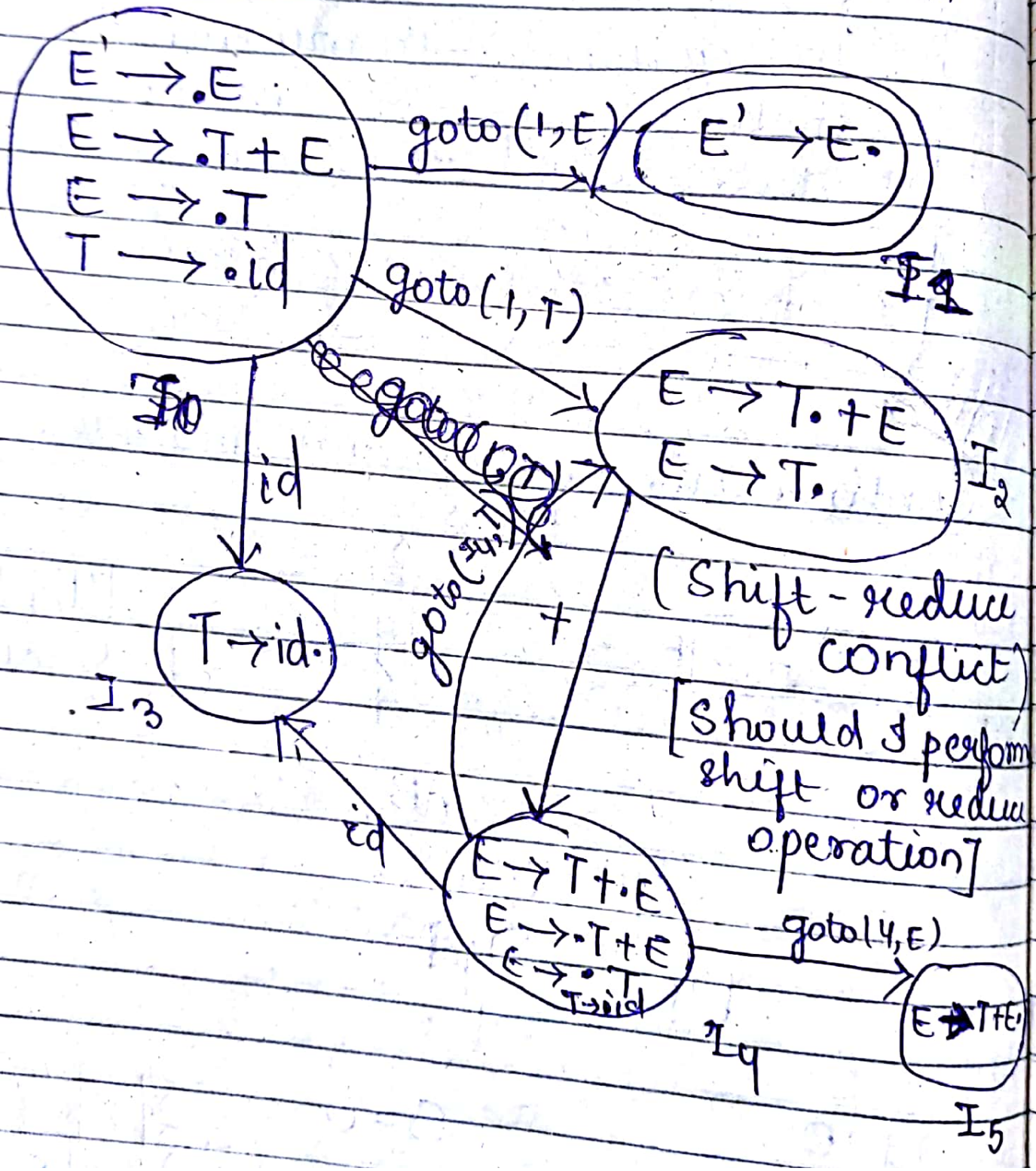
goto [0, S] = 1

$	0	S	1		$

[1, $] = acc

(2) $E \rightarrow T+E \mid T$
$T \rightarrow id$

① $E \rightarrow T+E$
② $E \rightarrow T$
③ $T \rightarrow id$



	Action			Goto	
	+	id	$	E	T
I_0		shift 3		1	2
I_1			accept		
I_2	shift 4/R2	R2	R2		
I_3	R3	R3	R3		
I_4		shift 3		5	2
I_5	R1	R1	R1		

(S4/R2) → shift-reduce conflict.

→ So this grammar cannot be parsed by LR(0) parser. (Or this gramm -ar is not LR(0) grammar)

SLR (Simple LR parser)

For same eg!-

follow of E is {$}
follow of T is {+, $}

	Action			Goto	
	+	id	$	E	T
I_0		S3		1	2
I_1			accept		
I_2	S4		R2		
I_3	R3		R3		
I_4		S3		5	2
I_5			R1		

→ In this SLR parser we place only the reduce action in the follow of the LHS of the production.

Stack	Buffer	Action
$ 0	id + id $	[0, id] = S3
$ 0 id 3	+ id $	[3, +] = R3

$T \rightarrow id$

$s = 0$

$\beta = 1$

$pop = 2 * 1 = 2$

Stack	Buffer	Action
$ 0 T 2	+ id $	[2, +] = S4

$	0	T	2	+	4

id	$

[4, id] = S3

$	0	T	2	+	4	id	3

$

[3, $] = R3

T → id

Pop = 2 * |β|

= 2

$	0	T	2	+	4	T	2

S = 4

Push T

goto (4, T)

$	0	T	2	+	4	T	2

$

[2, $] = R2

E → T

Pop = 2 elements

$	0	T	2	+	4	E	5

$

[5, $] = R1

S = 4

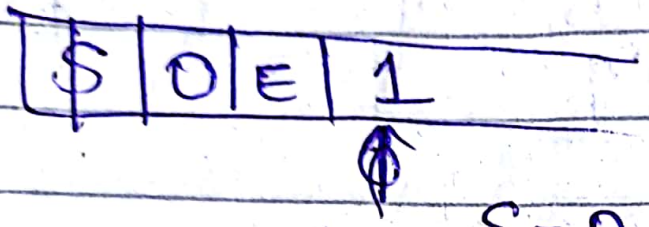
goto (4, T) = 5

$E \rightarrow T+E$

$Pop = 2 * |\beta|$

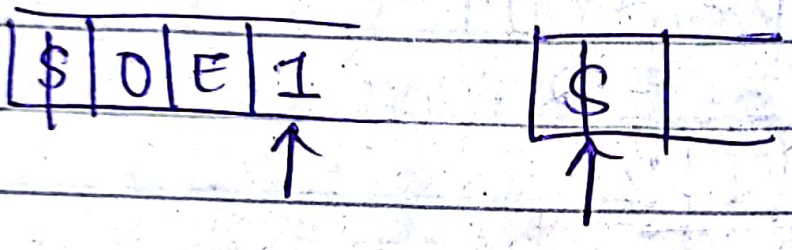
$= 2 * 3$

$= 6$



$S = 0$

$goto(0, E) = 1$



$[1, \$] =$ accept

14/8/18

$S \rightarrow A | a$

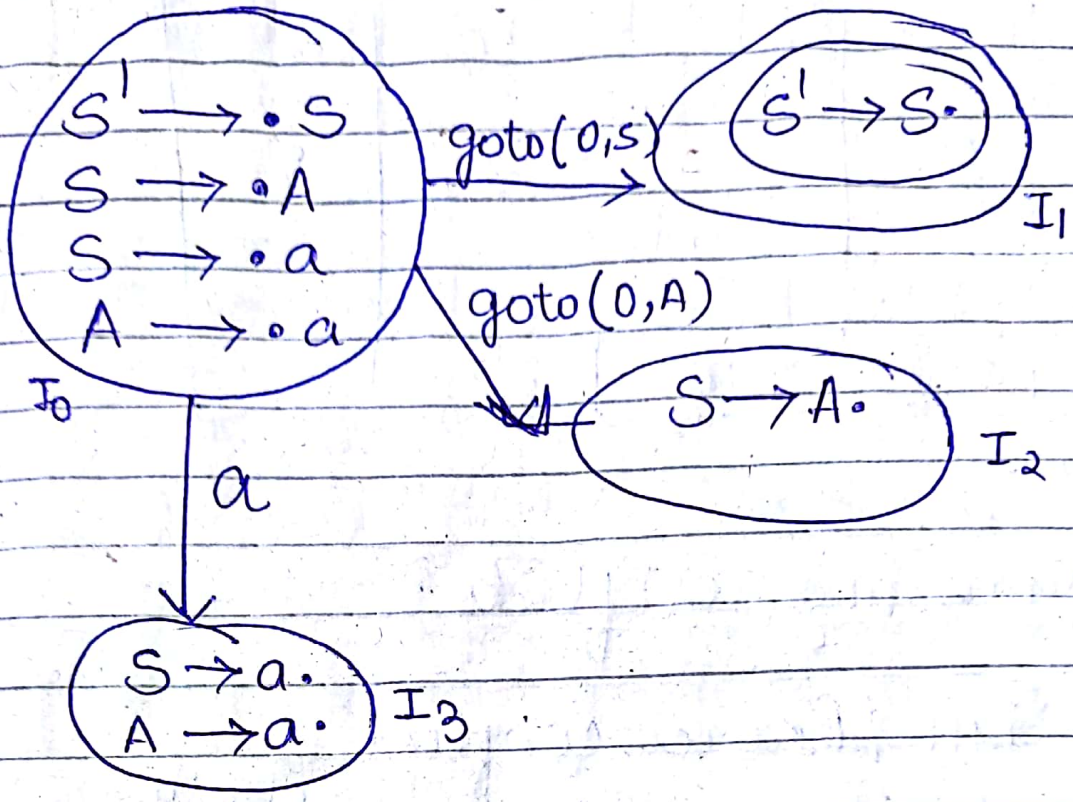
$A \rightarrow a.$

$S' \rightarrow S$

$S \rightarrow A$

$S \rightarrow a$

$A \rightarrow a.$



	Action		GOTO	
	a	$	S	A
I_0	S3		1	2
I_1		accept		
I_2	R1	R1		
I_3	R2/R3	R2/R3		

→ This is not LR(0). Reduce reduce conflict is there

follow of S → $
follow of A is $.

follow (S) ∩ follow (A) ≠ ϕ . .
then there is reduce reduce conflict

LR(0)

	Action		GOTO	
	a	$	S	A
I_0	S_3		1	2
I_1		accept		
I_2	R1	R1		
I_3	R2/R3	R2/R3		

Identifying conflicts :

① Shift / Reduce conflict.

$A \rightarrow \alpha \cdot a\beta$ → shifting terminal.

$B \rightarrow \gamma \cdot$

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

② Reduce / Reduce conflict.

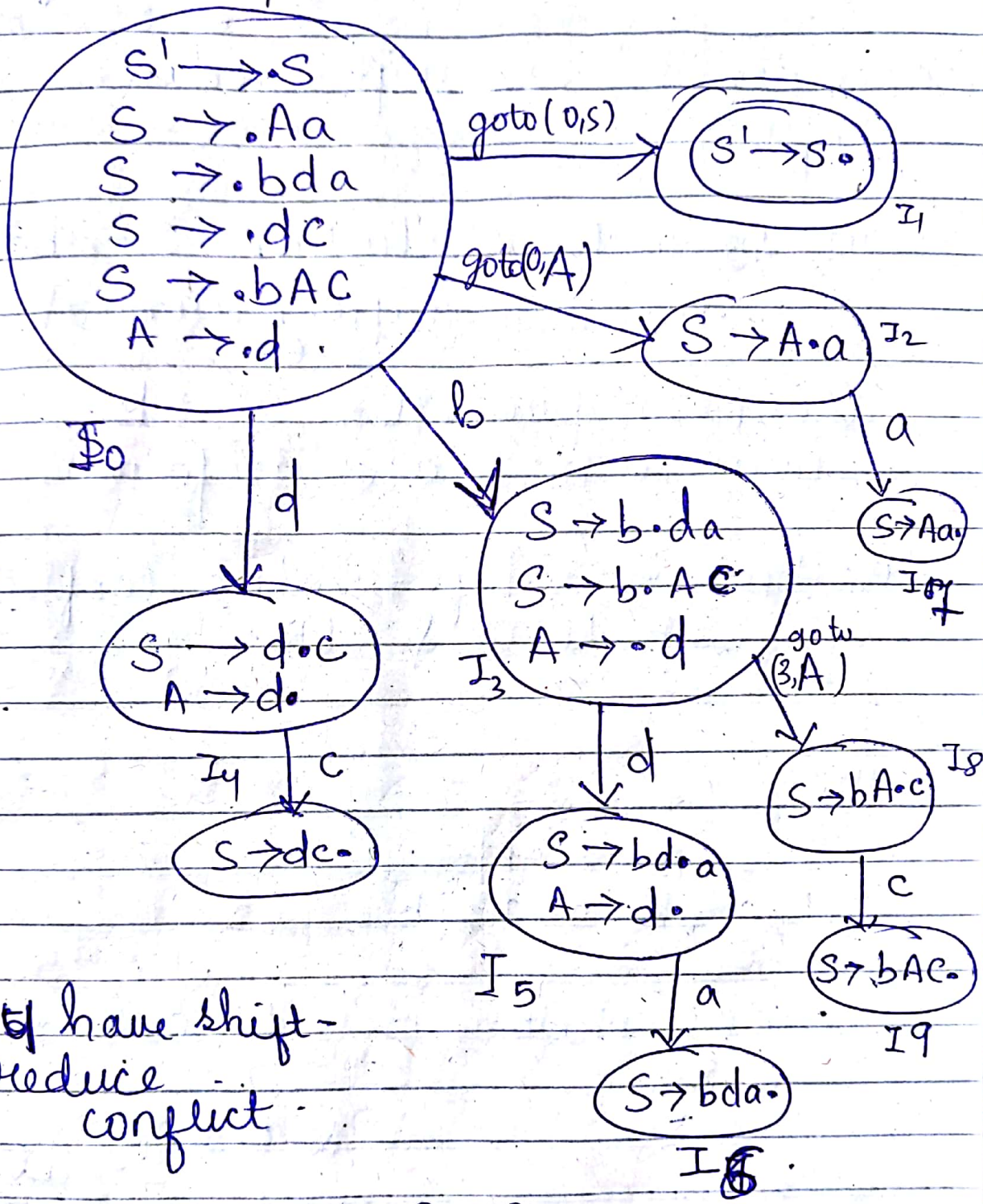
$A \rightarrow \alpha \cdot$

$B \rightarrow \alpha \cdot$

follow(A) ∩ follow(B) ≠ ϕ.

Q. $S \to Aa \mid bda \mid dc \mid bAc$

$A \to d$



→ I5 have shift-reduce conflict.

follow(S) = {$}

(A) = {a, c}

S → d.c → c is our terminal ; The follow of A :)

have c.
Thus there is shift-reduce conflict.

→ I_5 also have a shift-reduce conflict.

In I_5 in the production $S \to bd.$ @
a is our shifting terminal.
Our next production $A \to d.$
now A have a as its follow.

So, there is a shift-reduce conflict for both LR(0) & SLR.

Q HW

$S \to A$
$A \to AB|\epsilon$
$B \to aB|b$

Check whether it is LR(0) and SLR.

Note:-
$S \to ab$
$A \to ac$
LL(1); lookahead is a and is same so it is not LL(1)
LL(2); look ahead = ab and ac so no conflict

CLR (Canonical LR) or LR(1)

→ It has the highest parsing power.

According to the parsing power,

$$LR(0) < SLR(1) = LALR(1) <= CLR$$

Sometimes it can be as:-

$$LR(0) < SLR(1) < LALR(1) <= CLR$$

⇒ When the parsing table for SLR(1) and CLR are same then we need not go to the LALR(1)

⇒ When we tell about the size;

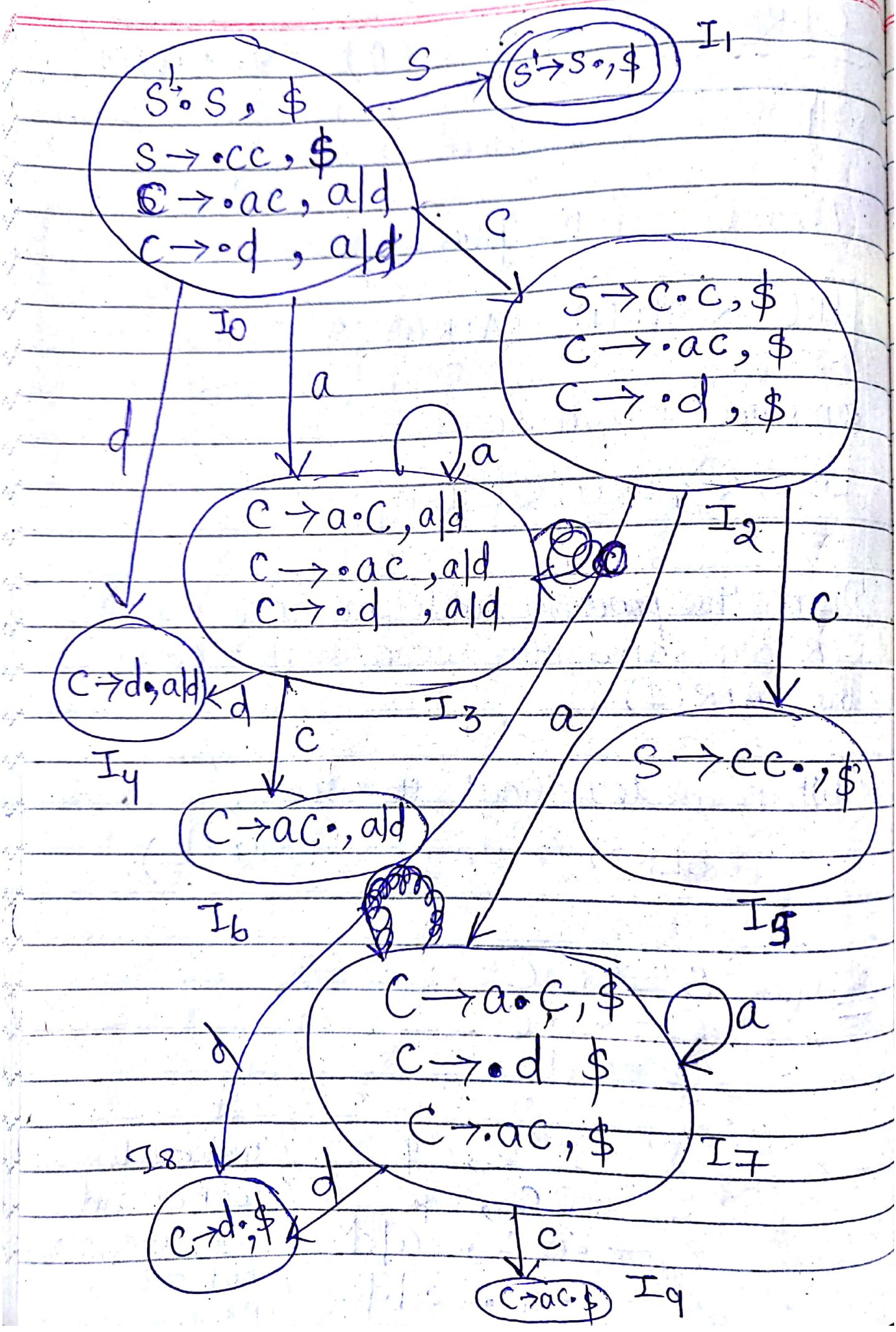
SLR(1) is always = LALR(1)

Q

$S \rightarrow CC$
$C \rightarrow aC$
$C \rightarrow d$

$S' \rightarrow \cdot S, \$$
$S \rightarrow \cdot CC, \$$
$C \rightarrow \cdot aC, a/d$
$C \rightarrow \cdot d, a/d$

} We write lookahead for symbols not a/c to production

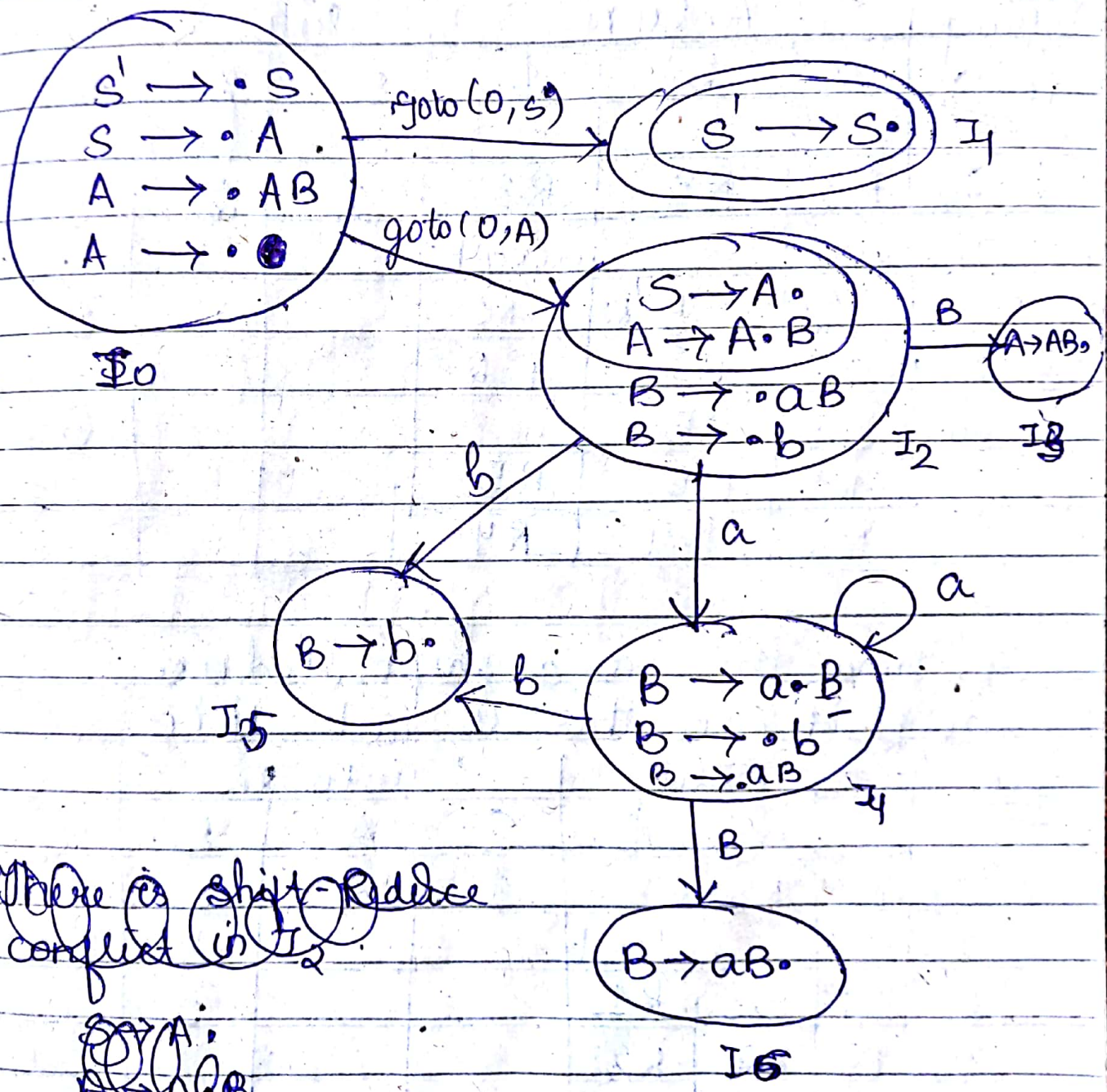


17/8/16

Tutorial

Q. $S' \to S$
$S \to A$
$A \to AB\$$
$A \to \epsilon$
$B \to aB$
$B \to b$

follow $S \to \{\$\}$
follow $A \to \{\$, a, b\}$
follow of $B \to \{\$, a, b\}$.



There is shift-Reduce conflict in I_2

A:
$A \to AB$

follow of B → $

If I write • ∈ or ∈ • it means the same

So $A \to \cdot \epsilon$

$= A \to \epsilon \cdot$

$= A \to \cdot$ (So we need not have to go in another separate state)

ϵ is not a terminal. It is just your empty or null string

LR(0)

	Action			Goto		
	a	b	$	S	A	B
← I_0	R3	R3	R3	1	2	
I_1			accept			
for $A \to \cdot$ (reduction) I_2	S4/R1	S5/R1	R1			3
I_3	R2	R2	R2			
I_4	S4	S5				6
I_5	R5	R5	R5			
I_6	R4	R4	R4			

So, here there is a shift reduce conflict. So this is not LR(0).

SLR

	Action			goto		
	a	b	$	S	A	B
I_0	R3	R3	R3	1	2	
I_1			accept			
I_2	S4	S5	R1			3
I_3	R2	R2	R2			
I_4	S4	S5				6
I_5	R5	R5	R5			
I_6	R4	R4	R5			

Here, shift reduce conflict is being resolved so it is SLR.

Input Strings:

① "a"

② "ab"

① Stack | Buffer | Action

Stack	Buffer	Action
$ 0 ↑	a $ ↑	[0, a] = R3

A → ∈.

2 * |β| = 0.

goto (0, A) = 2

Stack	Buffer	Action
$ 0 A 2 ↑	a $ ↑	[2, a] = S4
$ 0 A 2 a 4 ↑	$ ↑	(error). (4, $) = blank.

② ~~stack~~

② Stack | Buffer | Action

① $ 0 | a b $ | [0, a] = R3

② $ 0 A | a b $ | goto(0, A) = 2

$ 0 A 2 | a b $ | [2, a] = S4

③

$ 0 A 2 a 4 | b $ | [4, b] = S5

$ 0 A 2 a 4 b 5 | $ | [5, $] = R

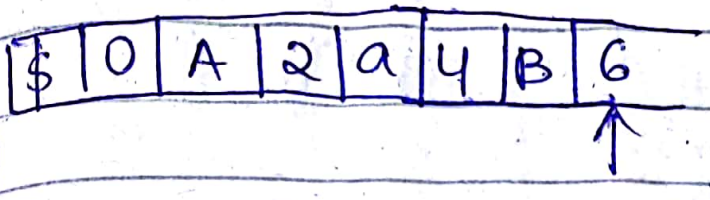
B → b

|β| = 1

Pop 2 * 1 = 2

$ 0 A 2 a 4 B

goto (4, B) = S6



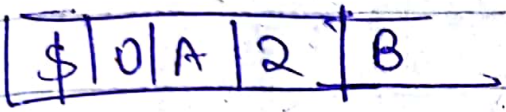
R4.

4)

$B \rightarrow aB$

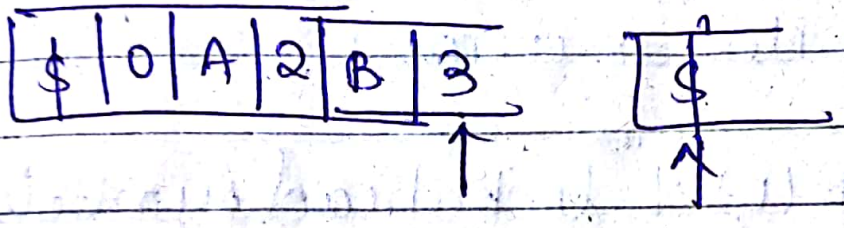
$|B| = 2$

Pop = 4



goto (2, B) = 3.

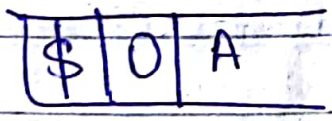
5)



= R2

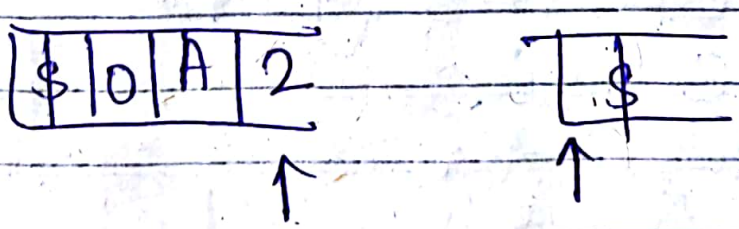
$A \rightarrow AB$

Pop = 4



goto(0, A) = 2

6)



R1.

Pop = 2 ; $S \rightarrow A$; (0, S) = 1

7)

accept

S → bB

B → B | b | ∈

B+ productions

S → bB

B → bB | ∈

(OR)

S → bS | b

20/8/18

⇒ We tell states as items.

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

Two types of lookaheads:-

① propagate ② spontaneous

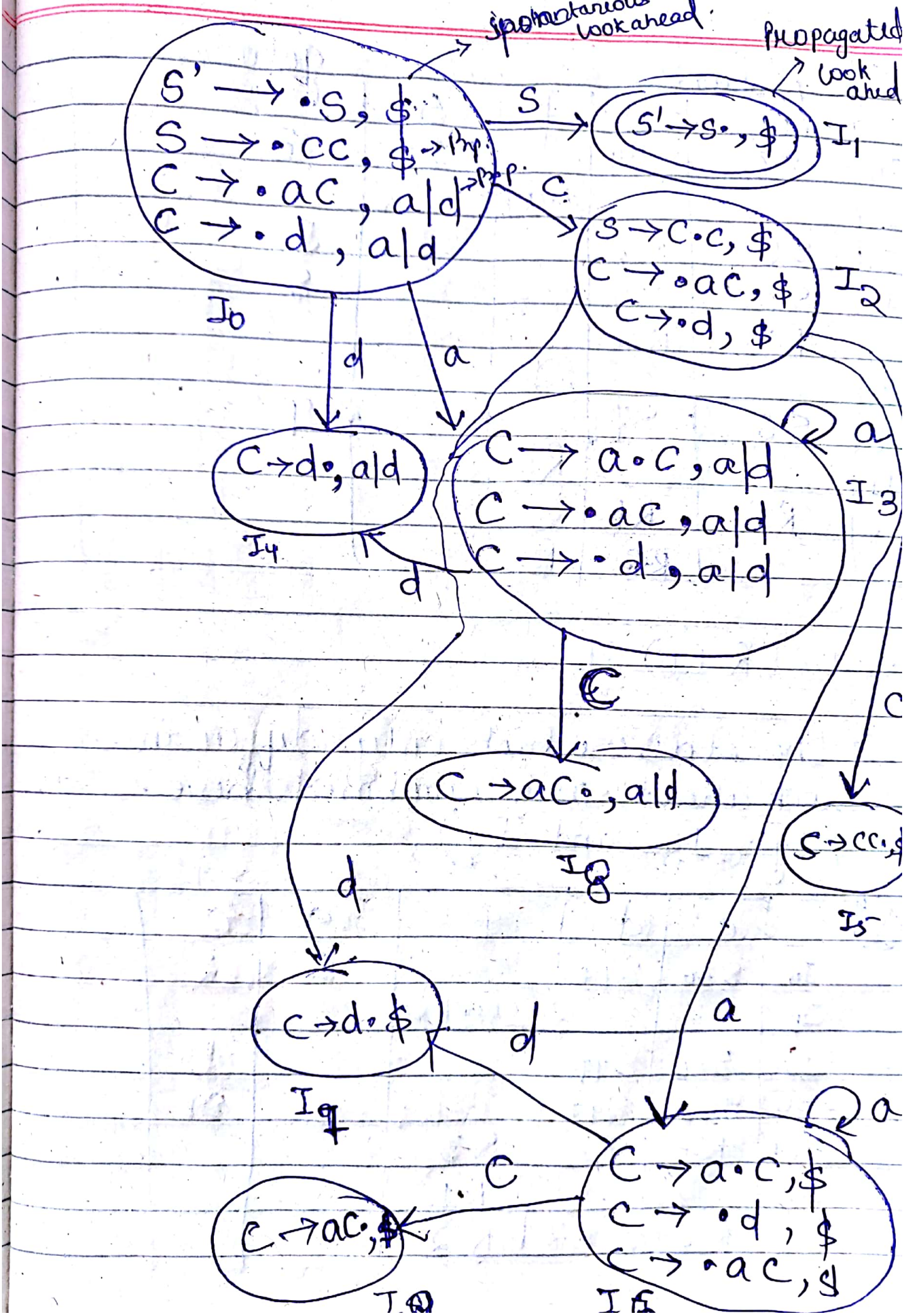
↓ ↓

same as the previous lookahead. new generated.

Eg:- S → CC 1

C → aC 2

C → d 3



	Action			goto	
	a	d	$	S	C
I_0	S3	S4		1	2
I_1			accept		
I_2	S6	S7			5
I_3	S3	S4			8
I_4	R3	R3			
I_5			R1		
I_6	S6	S7			9
I_7			R3		
I_8	R2	R2			
I_9		R2	R2		

LALR (1) -

→ The states which only differ in look ahead are combined here. (Eg:- I_8 and I_9).

	a	d	$	S	C
I_0	S36	S47		1	2
I_1			accept		
I_2	S36	S47			5
I_{36}	S36	S47			89
I_{47}	R3	R3	R3		
I_5			R1		
I_{89}	R2	R2	R2		

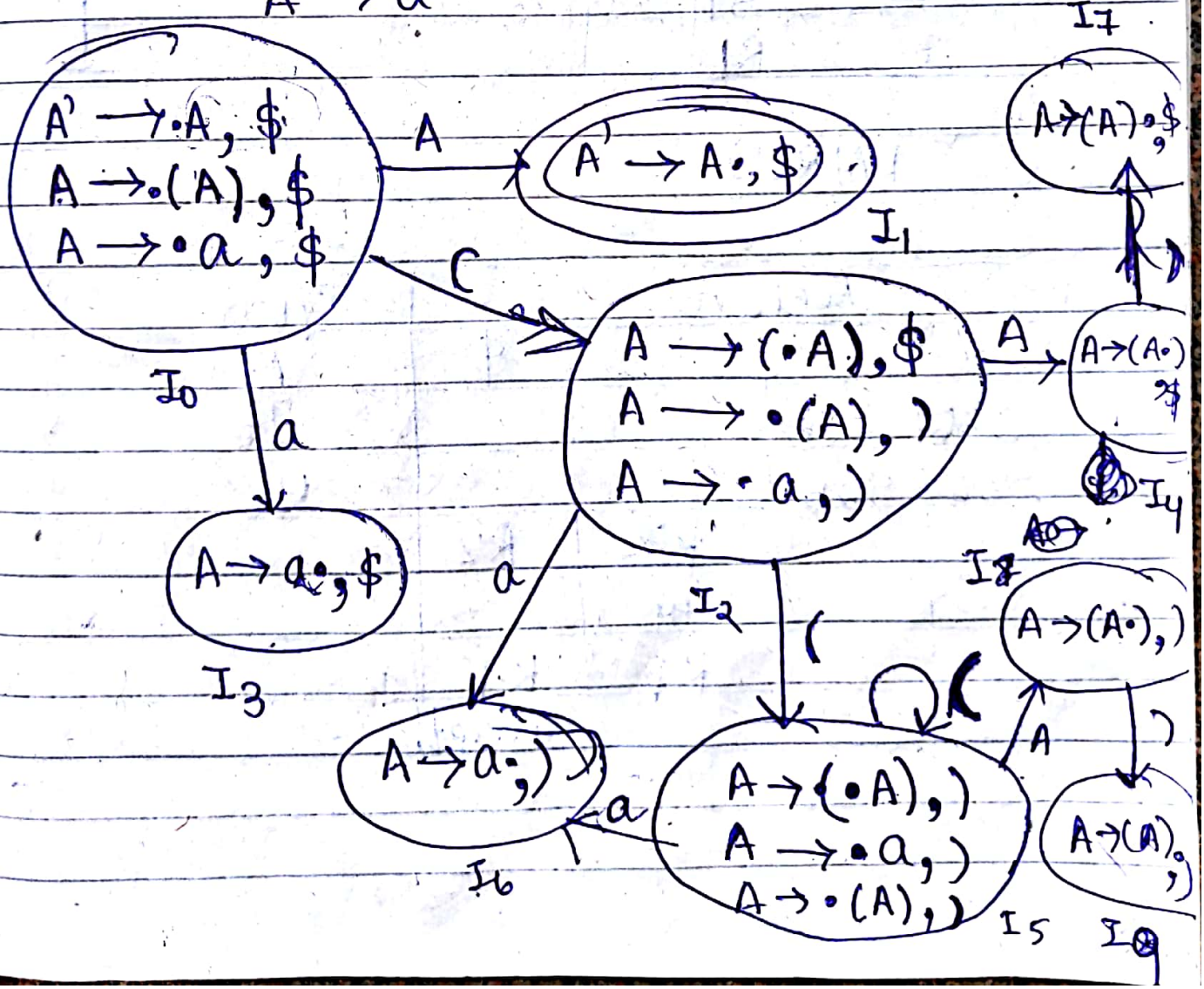
21/8/18

→ If there is no S/R conflict in CLR(1) then there is no S/R conflict in LALR(1) also.

→ If there is no R/R conflict in CLR(1) then there can be R/R conflict in LALR(1)

Q $A \to (A)$
$A \to a$

$A' \to A$
$A \to (A)$
$A \to a$



CLR(1)

	Action				Goto
	a	(	)	$	A
I_0	S3	S2			1
I_1				accept	
I_2	S6	S5			4
I_3				R2	
I_4			S7		
I_5	S6	S5			8
I_6			R2	R2	
I_7				R1	
I_8			S9		
I_9			R1		

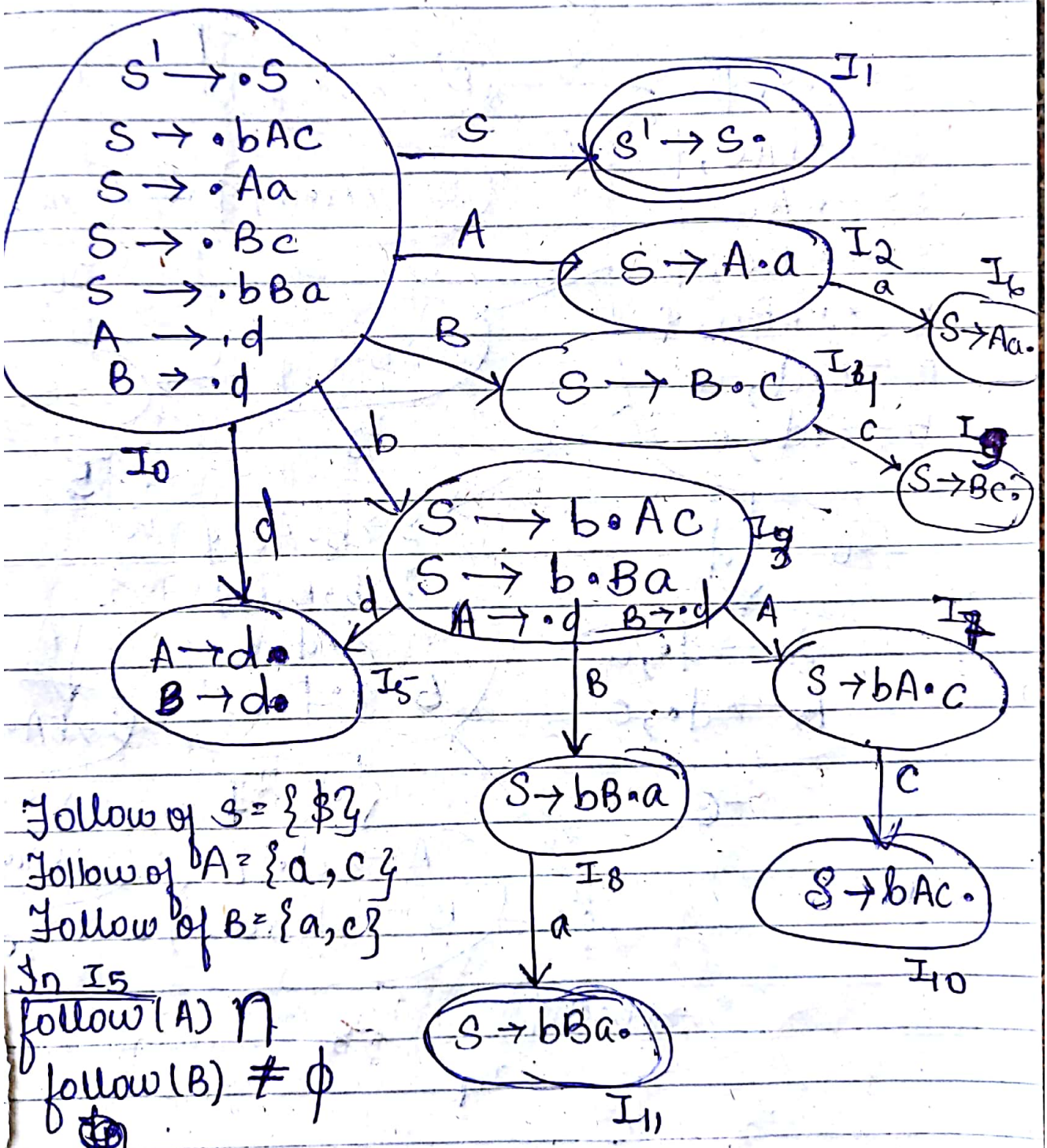
LALR(1)

	Action				Goto
	a	(	)	$	A
I_0	S36	S25			1
I_{25}	S36	S25			48
I_{36}			R2	R2	
I_{48}			S79		
I_{79}			R1	R1	
I_1				accept	

Q $S \to bAc \mid Aa \mid Bc \mid bBa$ (1, 2, 3, 4)

$A \to d$ (r) ; $B \to d$ (c)

For LR(0).



Follow of S = {$}

Follow of A = {a, c}

Follow of B = {a, c}

In I_5

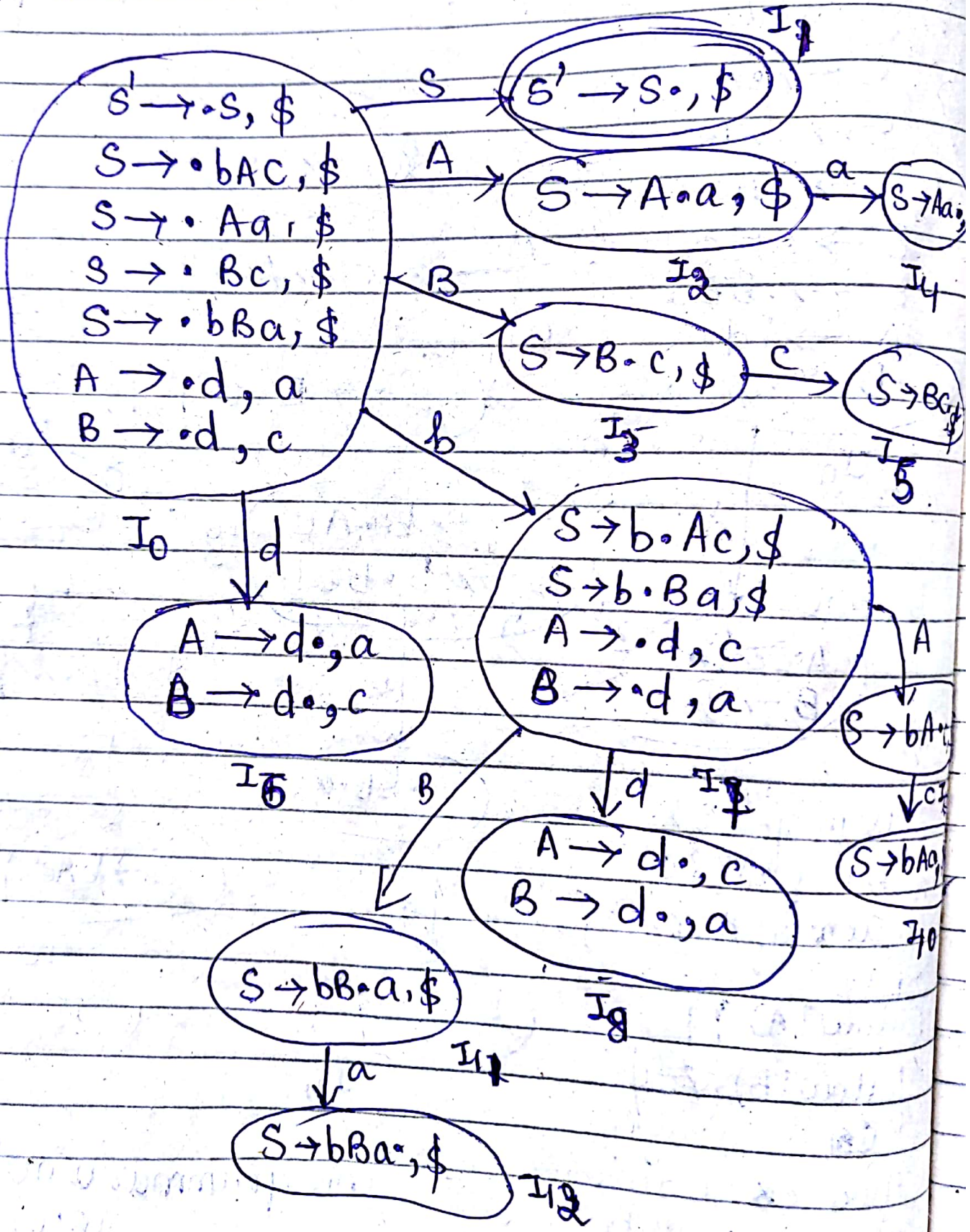
follow(A) ∩ follow(B) ≠ ϕ

There is reduce – reduce conflict in I_5

This grammar is not LR(0)

In SLR, still the reduce reduce conflict will exist, so it is not SLR.

CLR(1) or LR(1)



	Action					Goto		
	a	b	c	d	$	S	A	B
I_0		S7		S6		1	2	3
I_1					accept			
I_2	S4							
I_3			S5					
I_4					R2			
I_5					R3			
I_6	R5		R6					
I_7				S8			9	11
I_8	R6		R5					
I_9			S10					
I_{10}					R1			
I_{11}	S12							
I_{12}					R4			

The given grammar is CLR or LR(1).

In LALR(1) we will merge I_6 and I_8 giving

$A \rightarrow d\cdot, \{a,c\}$
$B \rightarrow d\cdot, \{a,c\}$
I_{68}

Thus, reduce/reduce error in State I_{68} for lookaheads {a,c}.

Thus, this grammar is not LALR(1).