* Finding First() and Follow() from given grammar

First(A) contains all terminals present in first place of every string derived by A.
First(terminal) = terminal
First(ϵ) = ϵ

E.g. S → abc | def | ghi
First(S) = {a, d, g}

E.g. S → ABC | ghi | jkl (A, g, j) ⇒ ~~(a, g, j)~~ (a, b, c, g, j)
A → a | b | c (a, b, c)
B → b (b)
D → d (d)

E.g. S → ABC
A → a | b | ϵ
B → c | d | ϵ
C → e | f | ϵ
F(C) = (e, f, ϵ), F(B) = (c, d, ϵ), F(A) = (a, b, ϵ)
F(S) = F(A) = (a, b, ϵ) ∪ F(B) = (c, d) ∪ F(C) = (e, f, ϵ)
= (a, b, c, d, e, f, ϵ)

e.g. $E \to TE'$

$E' \to *TE' \mid \epsilon$

$T \to FT'$

$T' \to \epsilon \mid +FT'$

$F \to id \mid (E$

first (F) = {id, (}, first (T) = first (F) = {id, (},
first (T') = {ϵ, +}, first (E') = {*, ϵ},
first (E) = first (T) = {id, (}

follow (A) contains set of all terminals present immediate in right of A. It never contains ϵ.
follow of start symbol is $.,

e.g. $S \to ACD$

$C \to a \mid b$

Fo (A) = first (C) = {a, b}

Fo (D) = Fo (S) = {$}

e.g. $S \to aSbS \mid bSaS \mid \epsilon$

Fo (S) = {$, b, a}

e.g. $S \to AaAb \mid BbBa$

$A \to \epsilon$

$B \to \epsilon$

Fo (A) = {a, b}, Fo (B) = {b, a}

E.g. $S \to ABC$ (A, B, C: ϵ ϵ ϵ)

$A \to DEF$

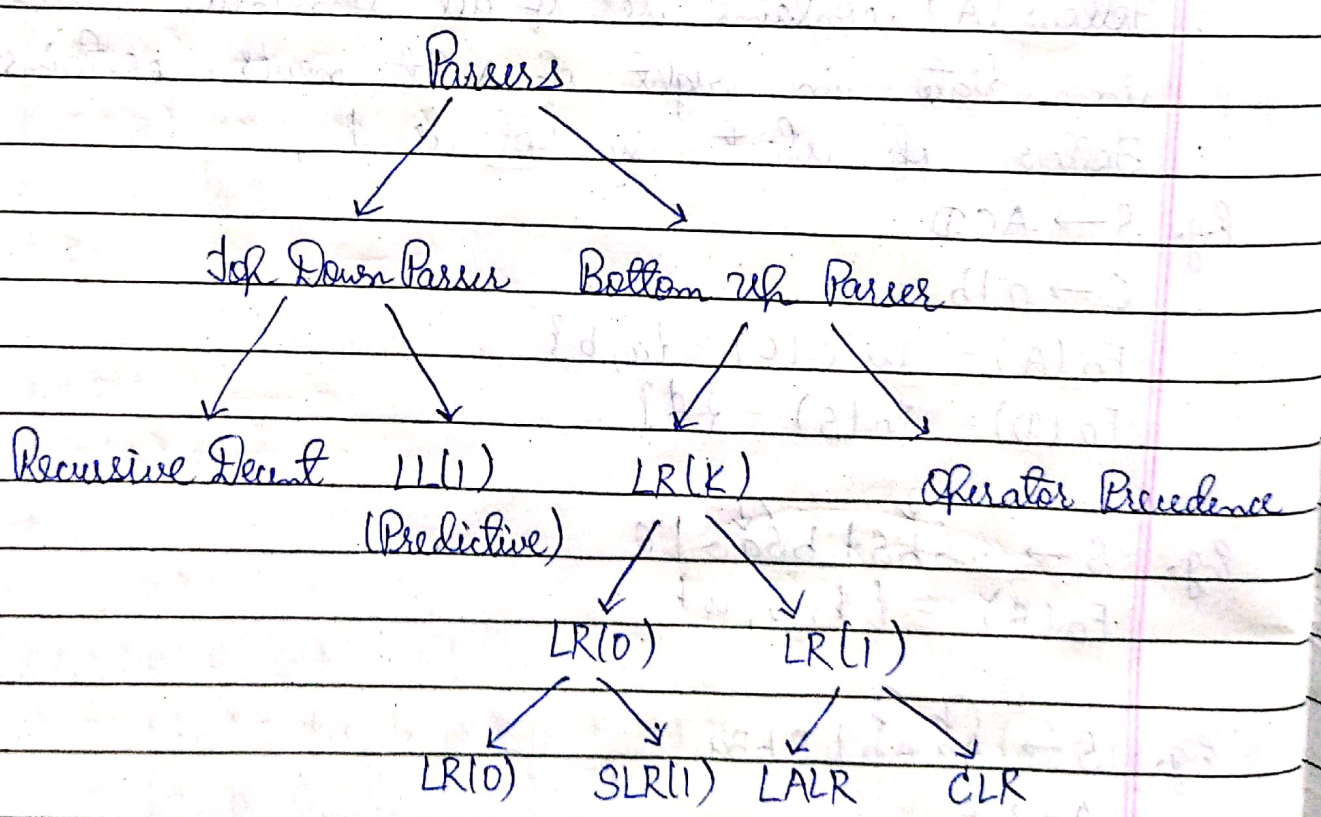
$B \to \epsilon$, $C \to \epsilon$, $D \to \epsilon$, $E \to \epsilon$, $F \to \epsilon$

Fo(A) = first(B) = first(C) = follow(S) = {$}

(first(B) $\downarrow \epsilon$, first(C) $\downarrow \epsilon$)

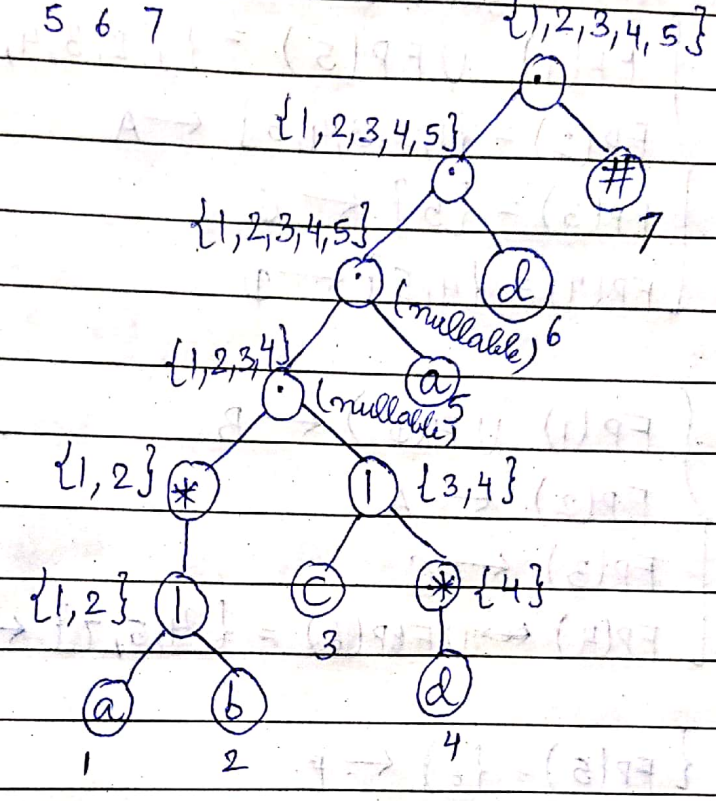
* Passing

It is a process of deriving string from a given grammar.



* RE to DFA using Syntax Tree method

Eg. $(a|b)^*.(c|d^*)^*.ad\#$
$\quad\quad 1\ 2 \quad\quad 3\ 4 \quad 5\ 6\ 7$



Union of

FIRSTPOS(1) = FIRSTPOS(child nodes)

FIRSTPOS(*.*) = FIRSTPOS(child node)

FIRSTPOS(.) = FIRSTPOS(left node)

(OR)

Union of FIRSTPOS(left node) and FIRSTPOS(right node) if left node contains * (that is nullable)

FOLLOWPOS - which symbols can follow the given symbol

FIRSTPOS(root) = {1,2,3,4,5} ← A

FOLLOWPOS(1) = {1,2,3,4,5}

FOLLOWPOS(2) = {1,2,3,4,5}

FOLLOWPOS(3) = {5}

FOLLOWPOS(4) = {4,5}

FOLLOWPOS(5) = {6}

FOLLOWPOS (6) = {7}

FOLLOWPOS (7) = ~~#~~ ∅

same variable

A
- FP(1) ∪ FP(5) = {1,2,3,4,5,6} ← B
- FP(2) = {1,2,3,4,5} ← A
- FP(3) = {5} ← C
- FP(4) = {4,5} ← D

B
- FP(1) ∪ FP(5) ← B
- FP(2) ← A
- FP(3) ← C
- FP(4) ~~←~~ ∪ FP(6) = {4,5,7} ← E

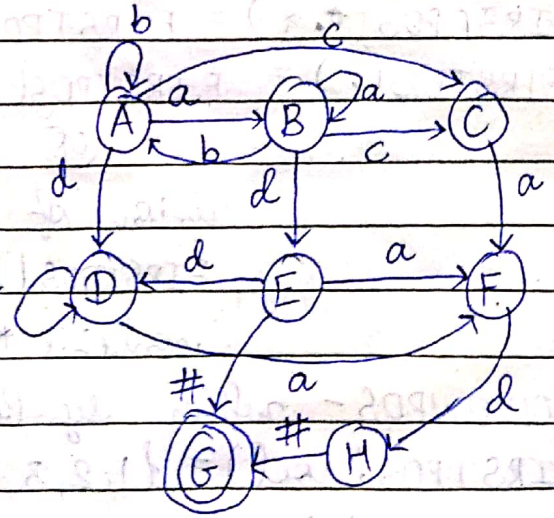
C
- FP(5) = {6} ← F

D
- FP(4) ← D
- FP(5) ← F

E
- FP(4) ← D
- FP(5) ← F
- FP(7) ~~=~~ ~~#~~ ∅ ← G

F
- FP(6) = {7} ← ~~G~~ H

G
- FP(7) ← #



* LL(1) Parser (Top - Down)

(*) How to check whether the string is LL(1) or not

E.g. $S \to (L) \mid a$

$L \to SL'$

$L' \to \epsilon \mid , SL'$

$F(S) = \{(, a\}$ $\quad$ $FO(S) = \{\$,), ,\}$

$F(L) = F(S) = \{(, a\}$ $\quad$ $FO(L) = \{)\}$

$F(L') = \{\epsilon, ,\}$ $\quad$ $FO(L') = \{)\}$

Make Parsing table where rows are non-terminals and columns are terminals and $.

	(	)	a	,	$
S	$S \to (L)$	e	$S \to a$	e	e
L	$L \to SL'$	e	$L \to SL'$	e	e
L'	$L' \to \epsilon$ (FOLLOW)	e	e	$L' \to , SL'$	e

Make entry for every row of the table where terminal's first is non-terminal (column). The entry would be equation whose first is non terminal. If first contains ϵ then use follow of that non-terminal. All other entries will be e. If one cell maximum one entry then the grammar will be accepted by LL(1) Parser.

The above grammar is LL(1).

E.g. $S \rightarrow aSbS \mid bSaS \mid \epsilon$

$F(S) = \{a, b, \epsilon\}$

$FO(S) = \{b, a, \$\}$

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

Here, there are two entries twice for one cell so the above grammar is not LL(1).

* How LL(1) parser works?

$S \rightarrow AA$

$A \rightarrow aA$

$A \rightarrow b$

	a	b	$
S	$S \rightarrow AA$	$S \rightarrow AA$	e
A	$A \rightarrow aA$	$A \rightarrow b$	e

String : abab

Initially, there is first non-terminal on the top of stack followed by $ and string is also followed by $. So at every point, non-terminal at top of stack is replaced with the function present in row of non-terminal of table and column of terminal of string where pointer is. If top of stack and string pointer has same terminal

then stack is popped and string pointer is incremented. If there are no rules present for a given non-terminal in stack according to table then grammar is not LL(1). If top of stack becomes $ and string pointer also becomes $ then string is LL(1).

Note: while replacing non-terminal with rule, all the contents are added in reverse order.

	Stack	a b a b $	Rule
①	$ S		S → AA
	$ AA		A → aA
	$ AAa (pop & increment)		
②	$ AA		A → b
	$ Ab (pop & increment)		
	$ A		A → aA
	$ Aa (pop & increment)		
	$ A		A → b
	$ b (pop & increment)		
	$ (accepted)		

① Check table for rules in row S and column a.
② Check table for rule in row A and column b.

* Conditions for Top-Down Parser
The grammar should not contain:
- Left-Recursion
- ~~Ambiguity~~ (Left-Factoring) Non-Determinism

1. Eliminating Left Recursion

E.g. $E \to E+T \mid E-T \mid T$
$T \to T*F \mid T/F \mid F$
$F \to (E) \mid id$

$A \to A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$

$\Downarrow$

$A \to \beta_1 A' \mid \beta_2 A' \mid \beta_3 A' \mid \dots \mid \beta_n A'$
$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_m A' \mid \epsilon$

$E \to TE'$ $\qquad$ $T \to FT'$
$E' \to +TE' \mid -TE' \mid \epsilon$ $\qquad$ $T' \to *FT' \mid /FT' \mid \epsilon$

E.g. S → aAbA | Sae | Bba | Sdd | bS
A → a
B → b

S → abAS' | Bba S' | bSS'
S' → aeS' | ddS' | ε
A → a
B → b

2. Eliminating Left - factoring

$A \rightarrow \alpha\beta_1 | \alpha\beta_2 | \ldots | \alpha\beta_n$

$\Downarrow$

$A \rightarrow \alpha A'$

$A' \rightarrow \beta_1 | \beta_2 | \ldots | \beta_n$

e.g. $S \rightarrow Aa | Ab | AS$

$S \rightarrow AS'$

$S' \rightarrow a | b | S$

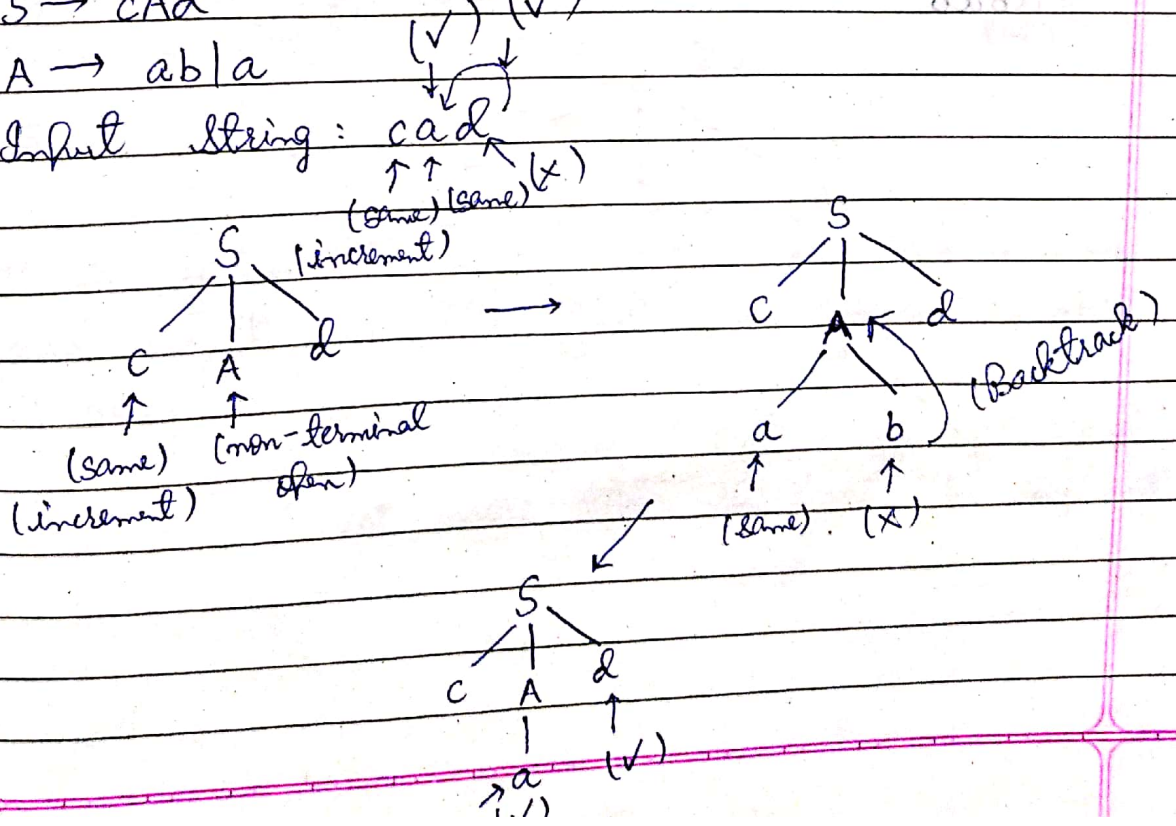
* Recursive Decent Parser (Top - Down)

A Recursive Decent Parser is a top-down parser built from a set of mutually recursive procedures where each such procedure implements one of the non-terminals of the grammar.

e.g. $S \rightarrow cAd$

$A \rightarrow ab | a$

Input string : cad



E.g. $S \rightarrow aSbS \mid bSaS \mid \epsilon$

String : abab

$S \rightarrow aSbS$

$\Rightarrow a\ aSbS\ bS$ × (Backtracking)

$\Rightarrow a\ bSaS\ bS$

$\Rightarrow ab\ aSbS\ aS\ bS$ × (Backtracking)

$\Rightarrow ab\ bSaS\ aS\ bS$ × (Backtracking)

$\Rightarrow ab\ \epsilon\ aSbS$

$\Rightarrow aba\ aSbS\ bS$ × (Backtracking)

$\Rightarrow aba\ bSaS\ bS$ × (Backtracking)

$\Rightarrow aba\ \epsilon\ bS$

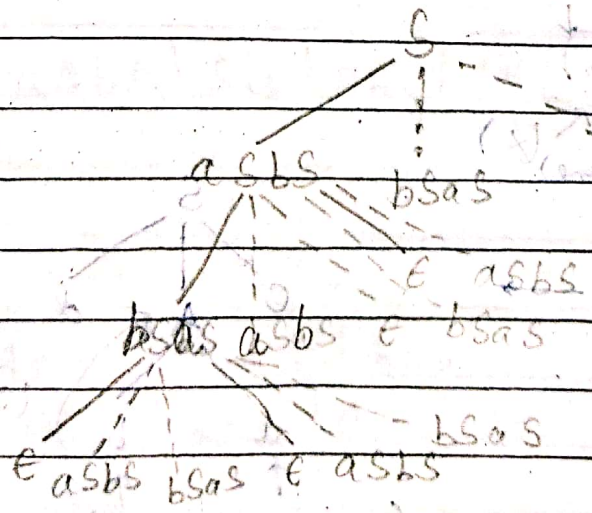
$\Rightarrow abab S$

$\Rightarrow abab aSbS$ × (Backtracking)

$\Rightarrow abab\ bSaS$ × (Backtracking)

$\Rightarrow abab\ \epsilon$

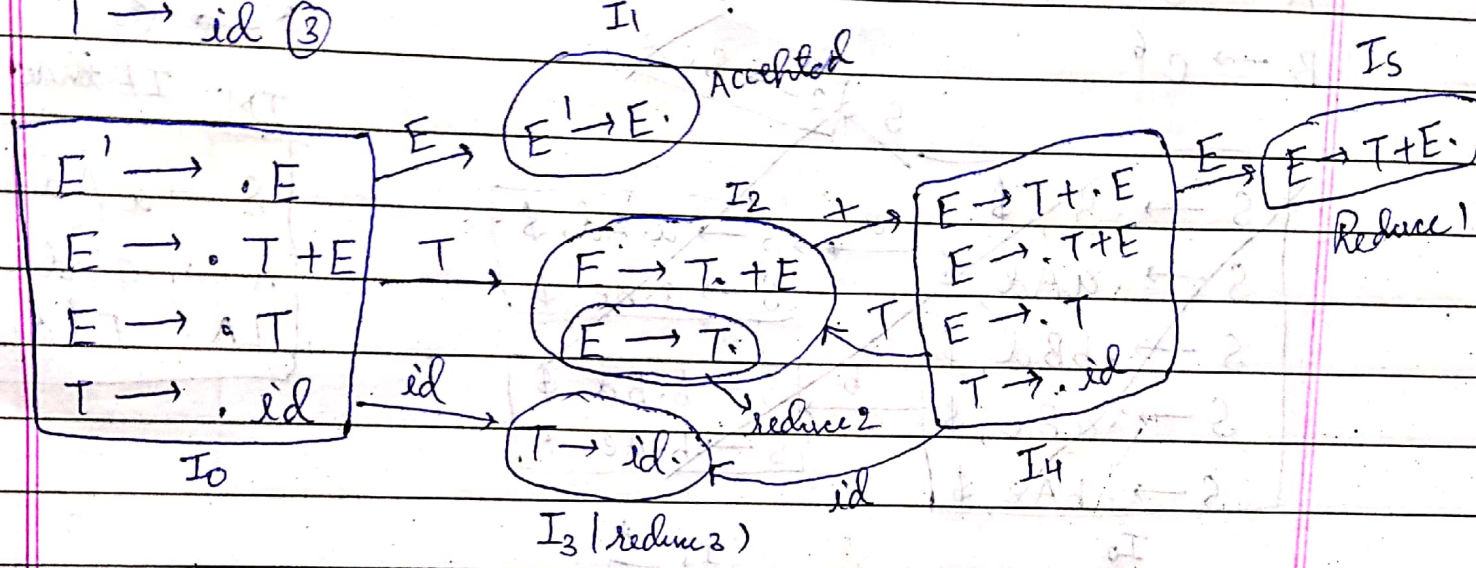
$\Rightarrow abab$



* SLR(1) (Bottom - up)

E.g. $E \to T+E$ (1) | T (2)

$T \to id$ (3)



State	id	+	$	E	T
0	S3			1	2
1			Accepted		
2		S4	r2 FO(E)		
3		r3	r3		
4	S3			5	2
5			r1		

Find follow of LHS of reduce equation

$E \to T.$

$FO(E) = \{\$\}$

$FO(T) = \{\$, +\}$

Write reduce in row having same state number and shift in the row from where equation is going.

SLR(1) ✓ (Not more than one entries in any cell)

* CLR (Bottom - up)

$S \xrightarrow{1} aAd \mid \xrightarrow{2} bBd \mid \xrightarrow{3} aBe \mid \xrightarrow{4} bAe$

$A \xrightarrow{5} c$

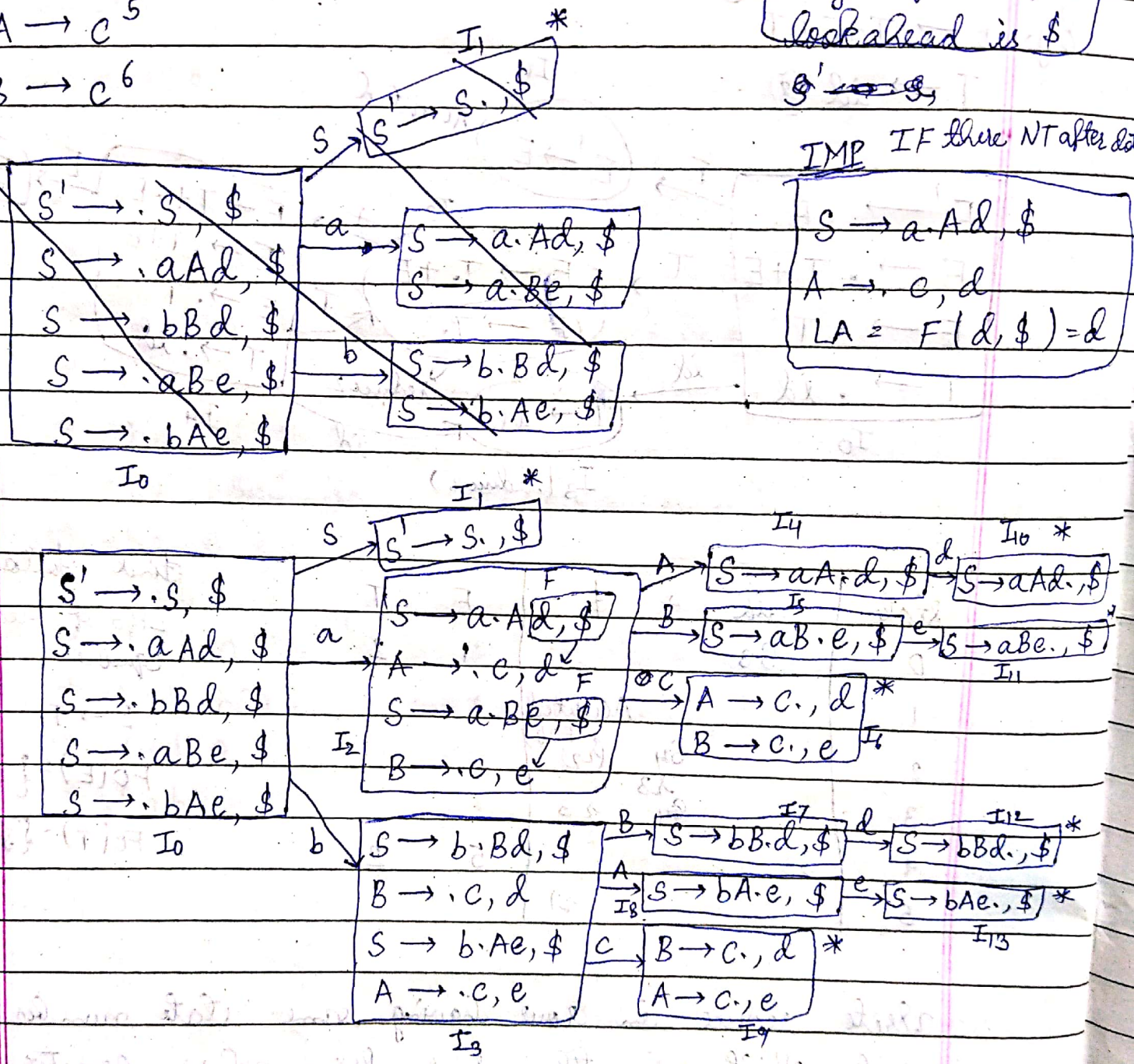
$B \xrightarrow{6} c$

IMP

By default, lookahead is $

$S' \rightarrow S$,

IMP IF there NT after dot



If state is accepted then write reduce in column of its lookahead.

State	a	b	c	d	e	$	A	B
0	S_2	S_3					1	
1						accepted		
2			S_6	r_5	r_6		4	5
3			S_9				8	7
4				S_{10}				
5					S_{11}			
6				r_5	r_6			
7				S_{12}				
8					S_{13}			
9				r_6	r_5			
10						r_1		
11						r_3		
12						r_2		
13						r_4		

* LALR (Bottom-up)

LALR is similar to CLR but in LALR similar states are merged.

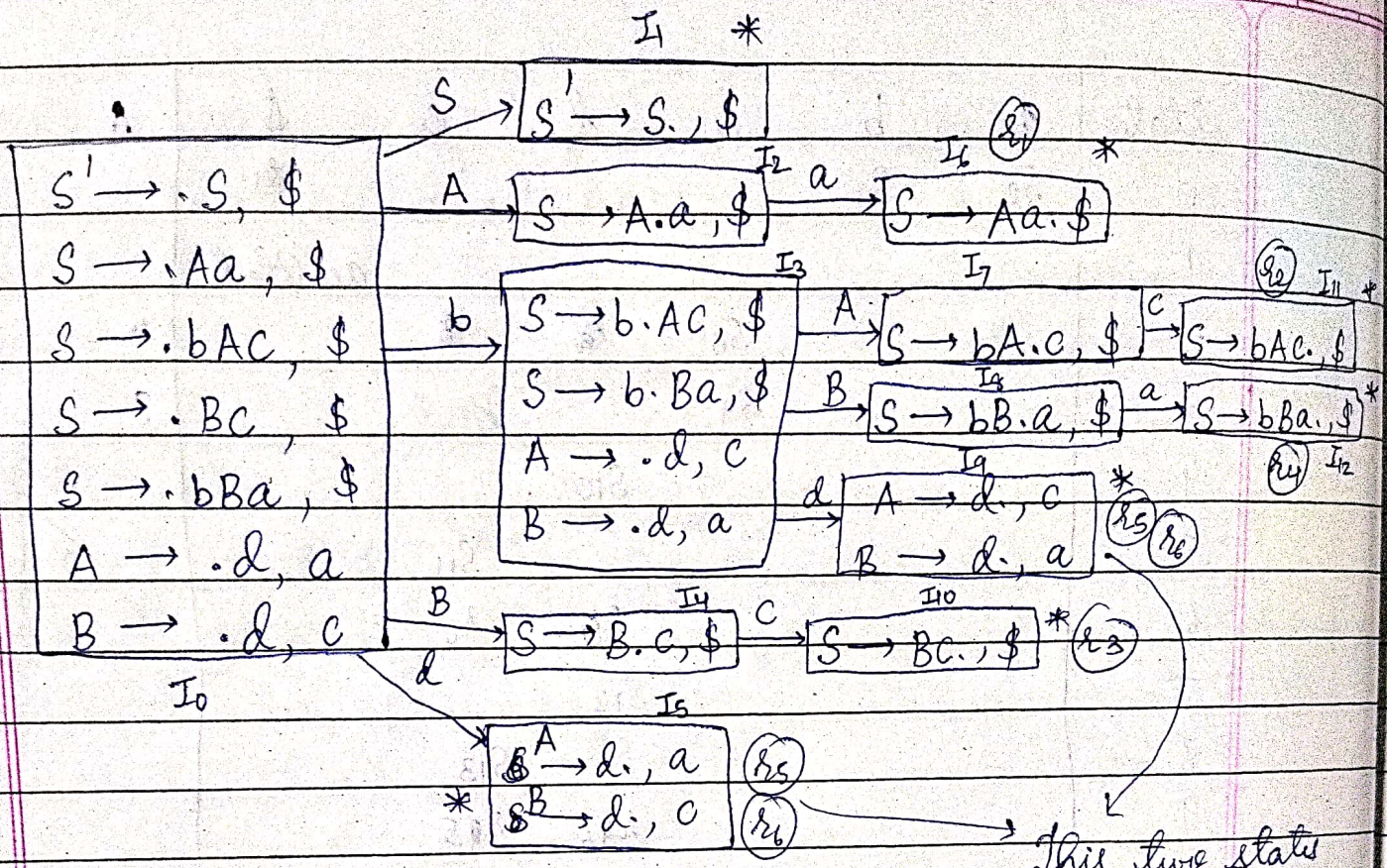
$S \rightarrow \overset{1}{A}a \mid \overset{2}{b}A\overset{3}{c} \mid \overset{4}{B}c \mid bBa$

$A \rightarrow d^5$

$B \rightarrow d^6$

In LALR, two reduces with different look aheads can be merged.

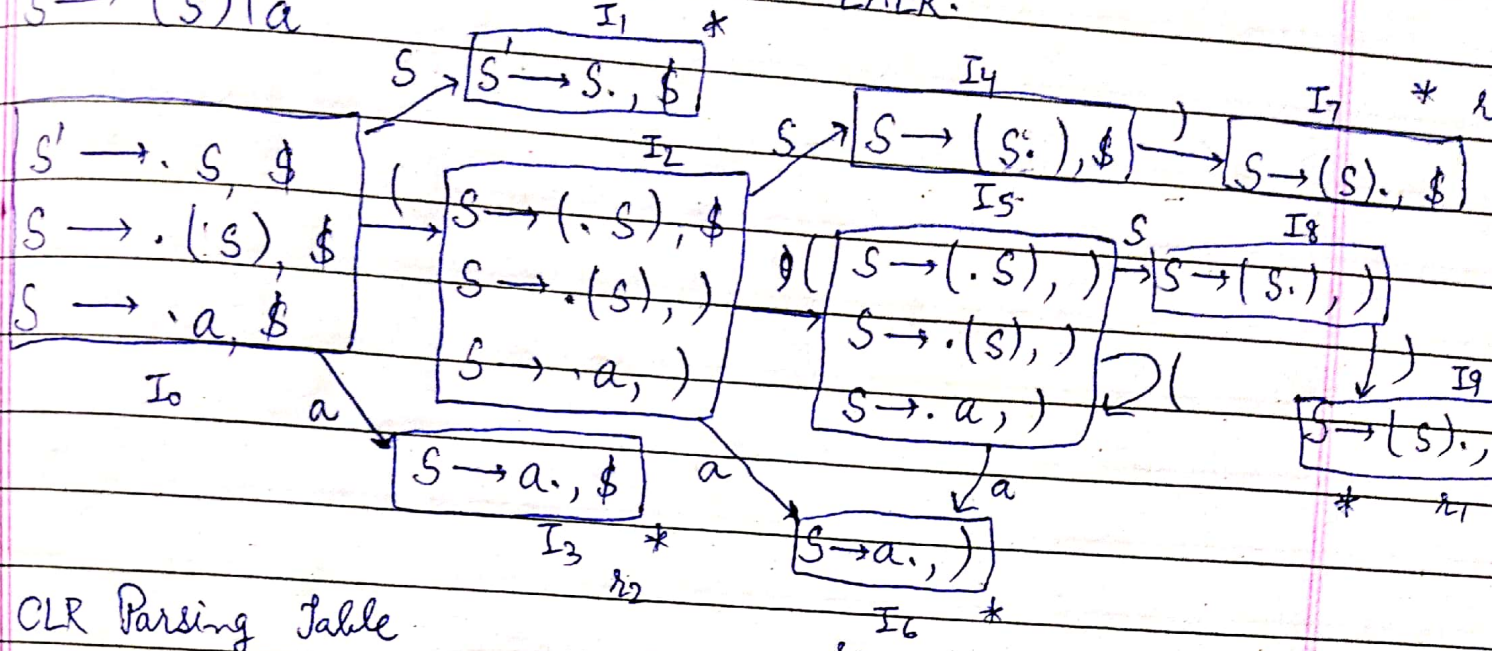
IMP Sometimes, a grammar can be CLR but is not LALR.



This two states cannot be merged as there would be multiple entries in the same column for LALR. So, the above grammar is not LALR but is CLR.

* Difference between CLR and LALR.

$S \rightarrow (S) | a$ (1) (2)



CLR Parsing Table

	(	)	a	$	S
0	S_2		S_3		1
1			accepted		
2	S_5		S_6		4
3				r_2	
4		S_8 7			
5	S_5		S_6		8
6		r_2			
7				r_1	
8		S_9			
9		r_1			

LALR Parsing Table

	(	)	a	$	S
0	S_2		S_3		1
1			accepted		
25	S_5		S_6		48
36		r_2		r_2	
48		S_{79}			
79		r_1		r_1	