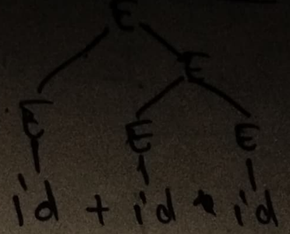
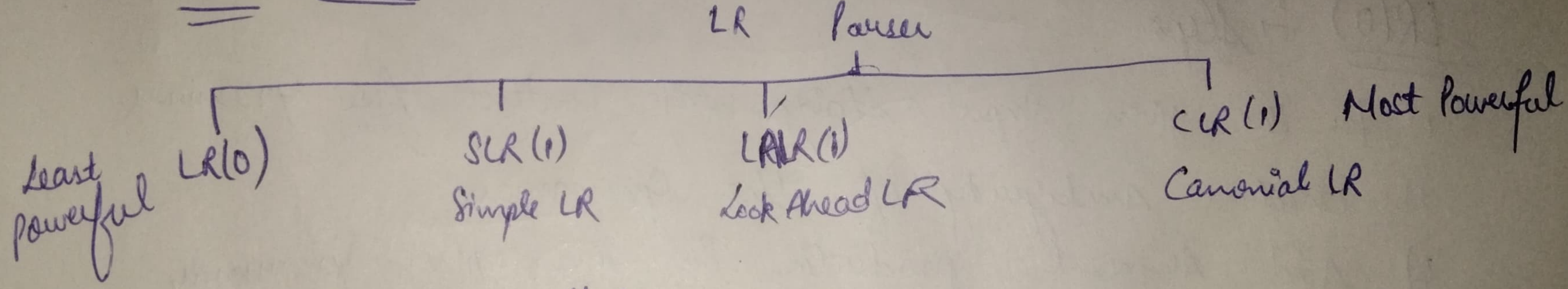


Disadvantage → No. of entries i.e. if we have 4 operators we will have 16 entries
10 operators we will have 100 entries
for N operator → O(n^2)   Size of table will be very big

So to dec. the size of the table, we use operator fn table.

* **LR Parsers** :→



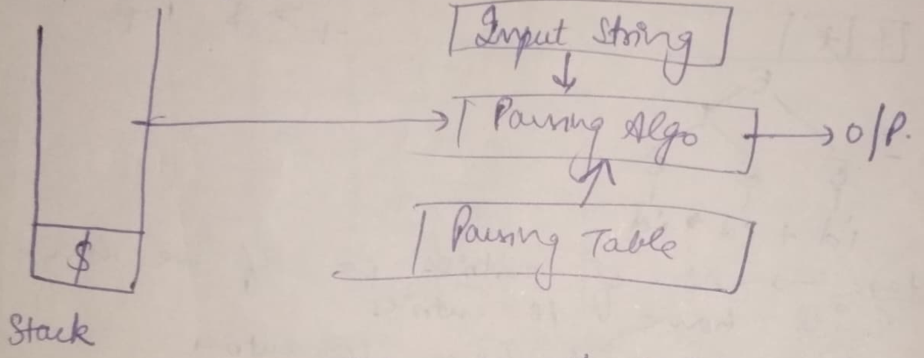
LR → Non recursive Shift reduce bottom up parser
Also known as LR(K)
L → Left to Right Scanning Input stream.
R → Construction of Rightmost derivation in reverse
K → Look Ahead Symbol

① SLR(1) :- Simple LR parser
Works on smallest class of Grammar.
few number of states (small Table required)
Simple and fast construction

② LR(1) → LR parser
Works on complete set of LR(1) grammar.
Large number of states (Large Table Required).
Slow Construction

③ LALR :- Look Ahead LR parser
Works on Intermediate size of Grammar.
No. of states are same as SLR(1).

Structure of LR Parser



- To construct LR(0) and SLR(1) tables we use Canonical Collection of LR(0) Items.
- To construct ~~LAR~~ LALR(1) and CLR(1) tables we use Canonical Collection of LR(1) Items.

LR(0) :- Steps

1) for the given Input string write Content free Grammar.
2) Check Ambiguity of the Grammar.
3) Add Augment production in the given Grammar.
4) Create a Canonical collection of ~~a~~ LR(0) Items
5) Draw a Data flow Diagram.
6) Construct LR(0). Parsing Table.

Example :- $S \to AA$
$A \to aA \mid b$

Soln :- $S \to AA$
$A \to aA$
$A \to b$

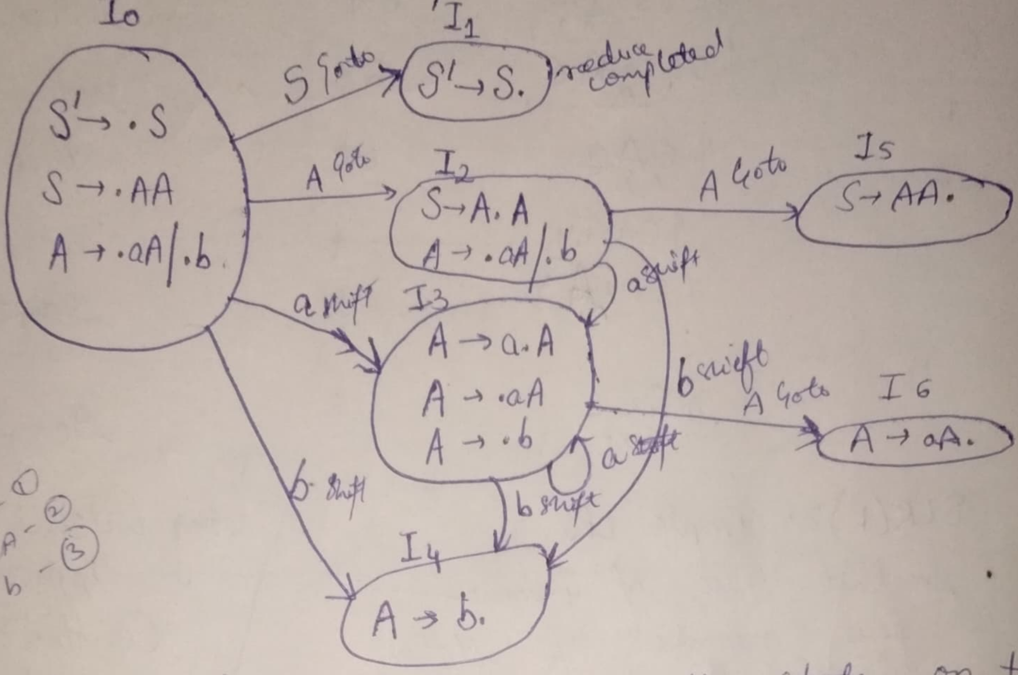
Augmented :- $S' \to S$
$S \to AA$
$A \to aA$
$A \to b$

Create Canonical Collection of LR(0) Items

✱ An LR(0) Item is a production G with dot at some position the right side of production

✱ LR(0) Items is useful to Indicate that how much of the Input has been scanned up to a given point in the process of parsing.

✱ In LR(0), we place the reduce node in entire row



S → AA - ①
A → aA - ②
A → b - ③

- If a state is going to another state on terminal (a,b) It is termed as shift.
- If a state is going to some other state on a variable → goto more
- If a state Contains the final Item in the particular row then write the reduce node completed.

States	Action (terminal)			Go to (non-terminal)	
	a	b	$	A	S
I0	S3	S4		2	1
I1	~~S3~~		accept		
I2	S3	S4		5	
I3	S3	S4		6	
I4	r3	r3	r3		
I5	r1	r1	r1		
I6	r2	r2	r2		

S → AA – ①
A → aA ②
A → b ③ – r3

At I₀ a is S3
so shift 3

Parse I/P String :-

Steps	Parsing Stack	Input	Action
1	$0	aabb$	Shift a 3
2.	$0a3	abb$	Shift a 3
3	$0a3a3	bb$	Shift b 4
4.	$0a3a3b4	b$	reduce r3 (A→b)
5.	$0a3a3A6	b$	reduce r2 | A→aA
6.	$0a3A6	b$	Reduce r2 (A→aA)
7.	$0A2	b$	Shift b4
8.	$0A2b4	$	reduce r3 (A→b)
9.	$0A2A5	$	~~Shift~~ reduce r1 (S→AA)
10.	$0S1	$	Accept

Using Bottom Up Approach Construct Tree

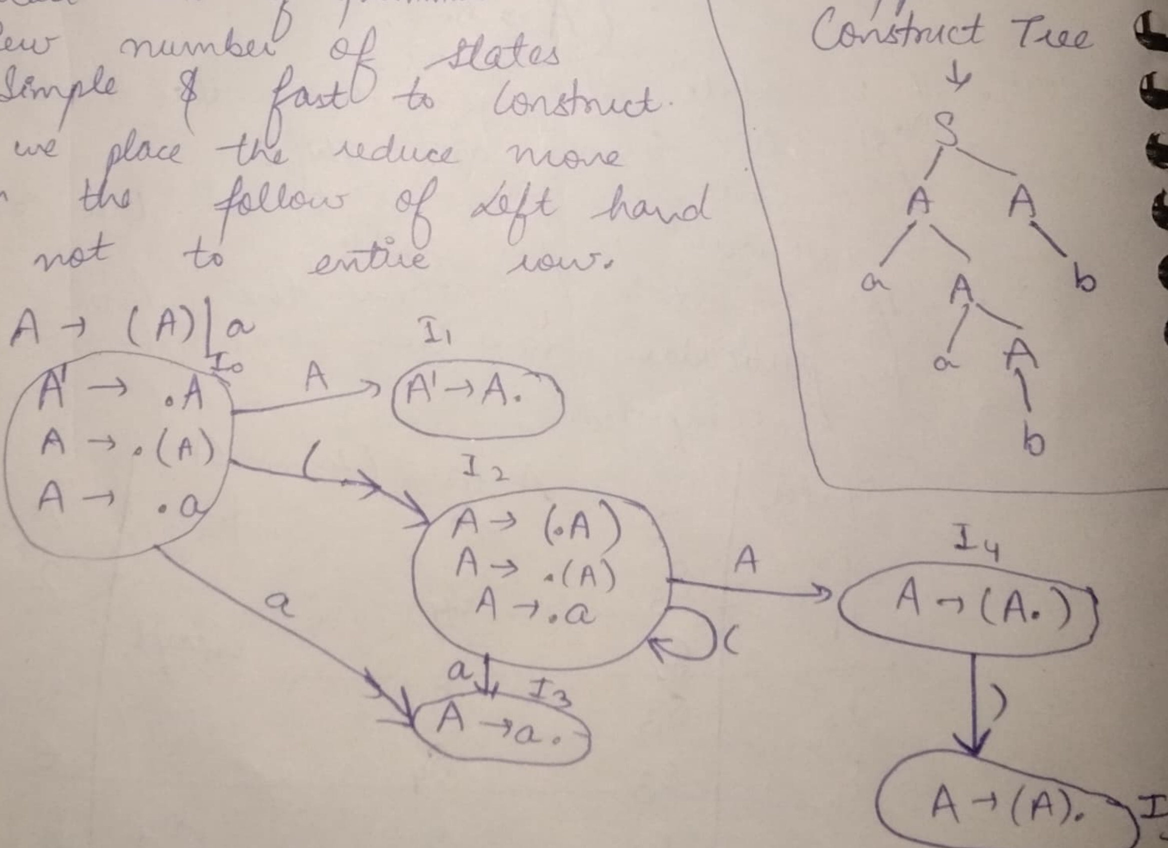
* SLR(1) :- Simple LR

- Smallest Class of Grammar
- few number of states
- Simple & fast to Construct.

* In SLR we place the reduce move only in the follow of Left hand side. not to entire row.

Example :- A → (A) | a

Soln :-



State	Action				Goto
	a	(	)	$	A
I0	S_3	S_2			1
I1				accept	
I2	S_3	S_2			4
I3			r_2	r_2	
I4			S_5		
I5			r_1	r_1	

A → .(A) — ①
A → .a — ②

A → .(A)
follow(A) = {), $}

Input string Parsing → (a)$

Step	Parsing stack	Input	Action
1.	$0	(a)$	shift (2
2.	$0(2	a)$	Shift a3
3.	$0(2a3	)$	reduce (A→a)
4.	$0(2A4	)$	Shift)5
5.	$0(2A4)5	$	reduce (A→(A))
6	$0A1	$	Accept

* **CLR(1)** :- Canocial Collection of LR(1) Items
LR(0) Item + Look ahead.

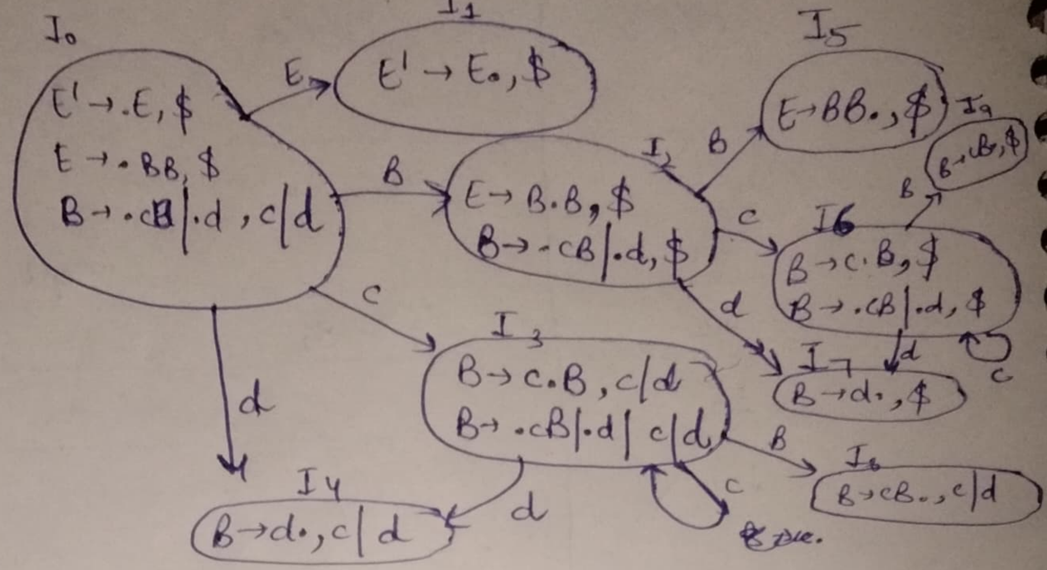
- reduce is written only on Look Ahead.

Example :- E → BB
B → cB|d

Soln :- Augment Grammar & LR(1) Item
E' → .E, $ ← Look ahead
E → .BB, $
B → .cB|.d, c|d

Pg

~~E'→.E~~



write reduce value at look ahead

State	Action			Goto	
	c	d	$	E	B
I0	S3	S4		1	2
I1			Accept		
I2	S3	S7			5
I3	S3	S4			8
I4	r3	r3			
I5			r1		
I6	S6	S7			9
I7			r3		
I8	r2	r2			
I9			r2		

* **LALR(1)** :→ LR(1) items

⇓

LR(0) items + look ahead value

Example
E → BB
B → cB|d

⟶ E'→.E, $
E→.BB, $
B→.cB|.d, c|d

Productions of I_3 & I_6 are same just look ahead Symbols are different.

So Combine I_3 and I_6.

$I_3, I_6 \rightarrow I_{36}$

Similarly, $I_4, I_7 \rightarrow I_{47}$

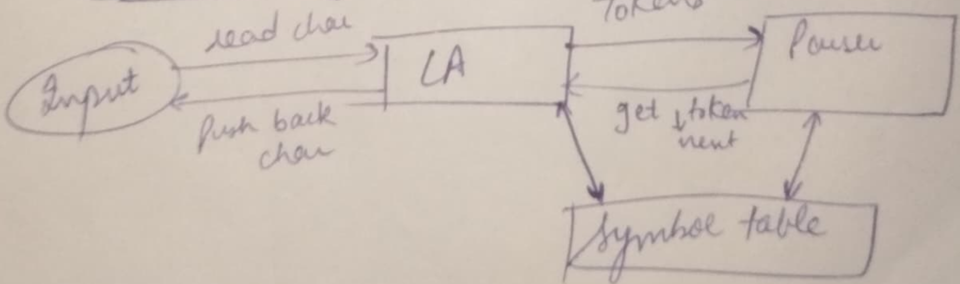
$I_8, I_9 \rightarrow I_{89}$

State	Action			Goto	
	c	d	$	E	B
I_0	S_{36}	S_{47}		1	2
I_1			Accept		
I_2	S_{36}	S_{47}			5
I_{36}	S_{36}	S_{47}			89
I_{47}	r_3	r_3	r_3		
I_5			r_1		
I_{36}	S_{36}	S_{47}			89
I_{47}			r_3		[illegible]
I_{89}	r_2	r_2	r_2		
I_{89}			r_2		

Similar and 1

merge

* Lexical Analyzer :-



* Structure of Lex Program :-

{ declaration } => declaration of Variables

%%

{ Translation Rules } => have the Pattern (Action)

%%

{ Auxilary function } => funcs can be Compiled Separately.