Step:1 First and Follow Table

Production	First	Follow
S→A	{a}	{$}
A→aBA'	{a}	{$}
A'→dA'/ℰ	{d,ℰ}	{$}
B→b	{b}	{d,$}
C→g	{g}	NA

Step 2: Construct parse table using first and follow function

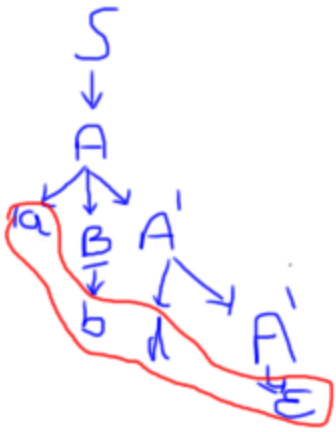
	a	**d**	**b**	**g**	**$**
S	S→A				
A	A→aBA'				
A'		A'→dA'			A'→ℰ
B			B→b		
C				C→g	

Step3: Stack Implementation by using parsing table

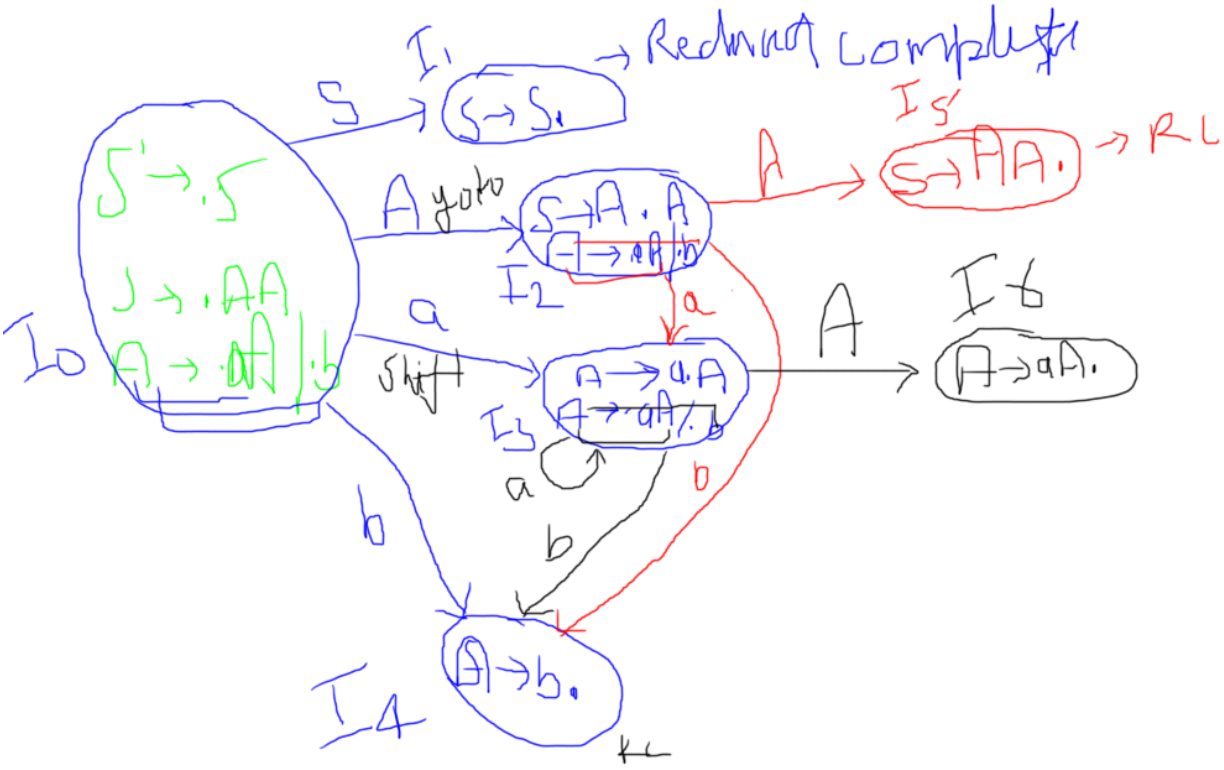
Let take input string is: abd$

Stack	**input sting**	**Production**
S$	abd$	**S→A**
A$	abd$	A→aBA'
aBA'$	abd$	**Pop a**
BA'$	bd$	B→b
bA'$	bd$	**Pop b**
A'$	d$	A'→dA'
dA'$	d$	**Pop d**
A'$	**$**	A'→ℰ
$	**$**	**Accepts**

Step 4: Generate the parse tree using stack implementation following top down approach.



Stack	**input sting**	**Production**
S$	abd$	**S→A**
A$	abd$	A→aBA’
aBA’$	abd$	**Pop a**
BA’$	bd$	B→b
bA’$	bd$	**Pop b**
A’$	d$	A’→dA’
dA’$	d$	**Pop d**
A’$	**$**	A’→ε
$	**$**	**Accepts**



Step 6: LR(0) Table

Sates	Action (Terminal Symbol)			GOTO (Non Terminal Symbol)	
	a	b	$	A	S
I_0	S3	S4		2	1
I_1			Accepted		
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		

Parsing i/p String :

Input string: aabb$

Steps	Parsing Stack	i/p sring	Action
1	$0	aabb$	Shift 3 / Shift a3
2	$0a3	abb$	Shift a3
3	$0a3a3	bb$	Shift b4
4	$0a3a3b4	b$	Reduces 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	$	Reduce1 (S→AA)
10	$0S1	$	Accept

SLR(1) Parsing:

-Simple LR

-works on smallest class of grammar

-few number of states requires

-simple and fast to construction

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

Example: G:

A→ (A)

A→a

Solution:

1. **CFG**
2. **Check ambiguity**
3. **Add the augmented grammmer**
   A'→A
   A→(A)
   A→a
4. **Calculate the canonical collection of LR(0) items**

A’→.A

A→.(A)

A→.a

5- Construct the DFD:

Date:25-09-2020

LALR (1) Parser: / Construct LALR (1) parsing Table

Step 1: Design the clr (1) parser/ parsing table

1 augmented grammar

2. calculate the canonical collection of LR(1) items.

3. Draw diagram (DFA/DFD/TD)

4. Create CLR(1) parsing table

Step2: Design the LALR (1) parser

1. Find the states having the same production, merge both production.
2. Draw diagram DFD/DFA
3. Create LALR paring table.

Example:1 S→AA

A→aA

A→ b

Construct LALR parsing table.

Solution:

1 .

(i) Augmented grammar

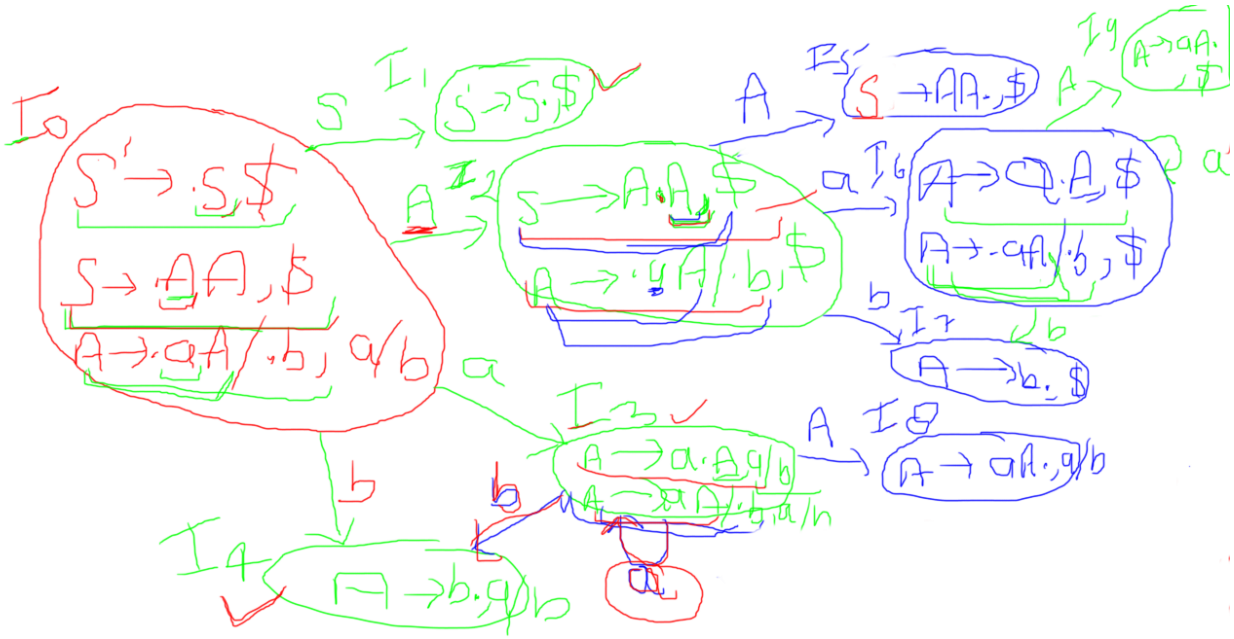
S'→.S
S→.AA ----(i)
A→.aA------ii
A→ .b -------iii

(ii) Calculate the canonical collection of LR(1) items.

LR(1)= LR(0) + Look Ahead items

S'→.S, $
S→.AA, $
A→.aA, a/b
A→ .b , a/b

(iii) Draw the diagram:



Parsing Table:

State	Action				Goto	
	a	b	$		S	A
I0	S3	S4			1	2
I1			Accept			
I2	S6	S7				5
I3	S3	S4				8
I4	R3	R3				
I5			R1			
I6	S6	S7				9
I7			R3			
I8	R2	R2				
I9			R2			

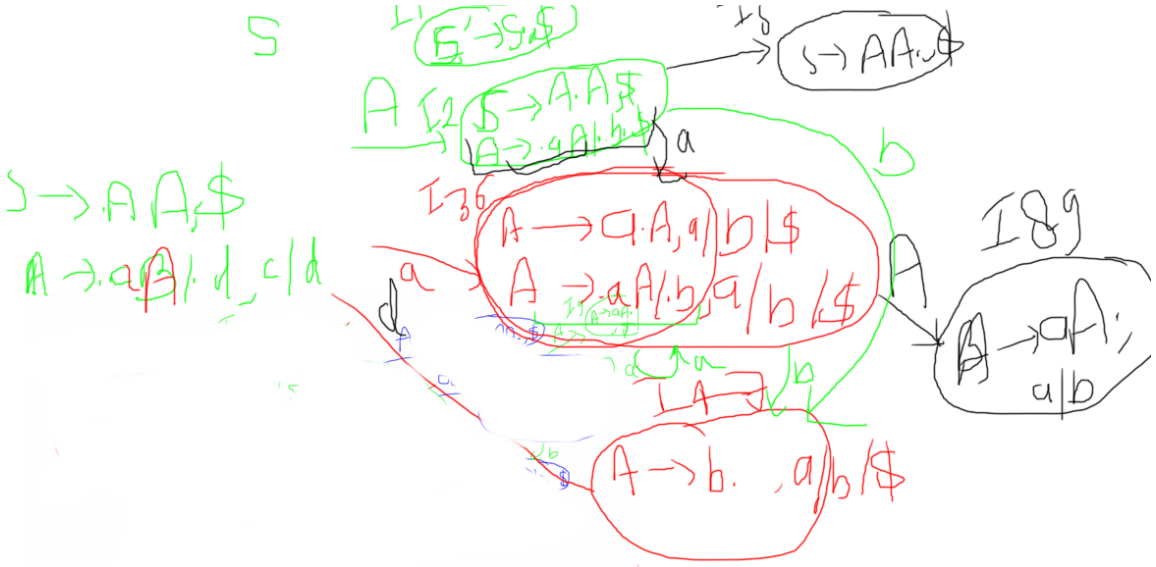
Step 2: (i) find the states having the production & merge

I3,I6 ➔ I36
I4,I7 ➔ I47
I8,I9 ➔I89

Parsing Table of LALR (1)

State	Action				Goto	
	a	b	$		S	A
I0	S36	S47			1	2
I1			Accept			
I2	S36	S47				5
I36	S36	S47				89
I47	R3	R3	R3			
I5			R1			
I89	R2	R2	R2			

DFD of LALR:


S
S → .A A, $
A → .a A | .b
I0
S' → .S, $
S → A.A, $
A → .a A | .b
I1
S → AA., $
I2
A → a.A, $
A → .a A | .b
I3
A → a.A, a/b/$
A → .a A | .b
I4
A → b., a/b/$
I5
A → aA., $
I6
A → a.A, a/b/$
A → .a A | .b
I7
A → b., a/b/$
I8
A → aA., a/b
I9
A → aA., a/b
a
b
A

- **YAAC (yet another compiler compiler)**

Lex – lexical analyzer generator

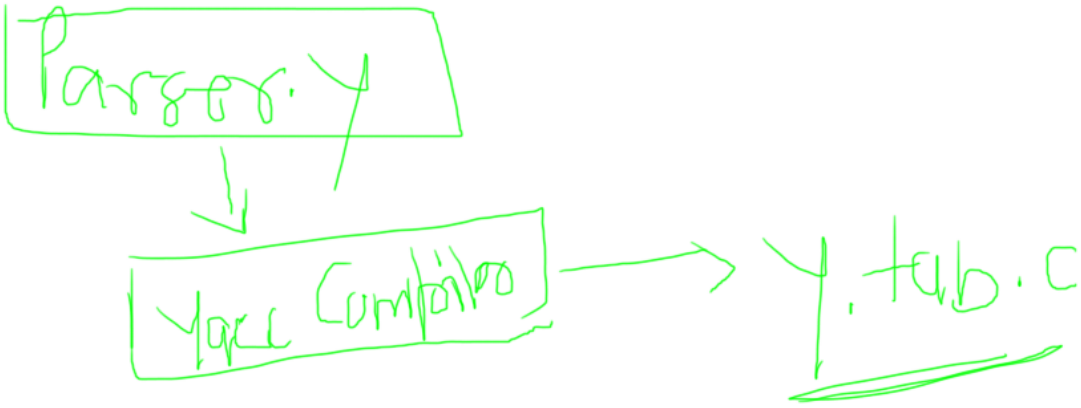
Yaac – Parse generator

Yaac is a tool which generates the LALR parser.

Working of YAAC Tool:

Step 1: YAAC specification

Parser.y

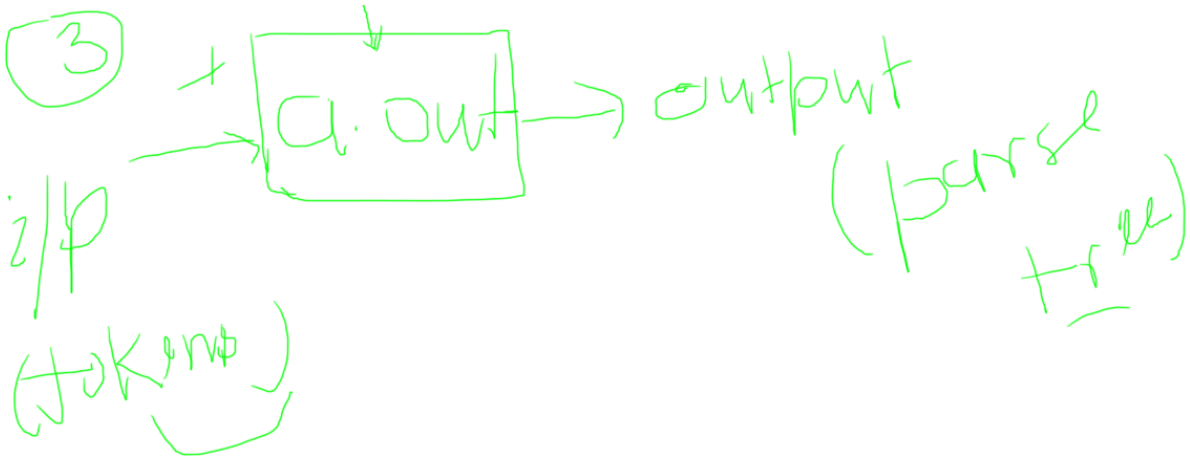


Step 2:

y.tab.c →


y.tab.c
C Compiler
a.out
(syntax Analyzer)

Step 3:


3
+
i/p
(tokens)
a.out
output
(parse tree)