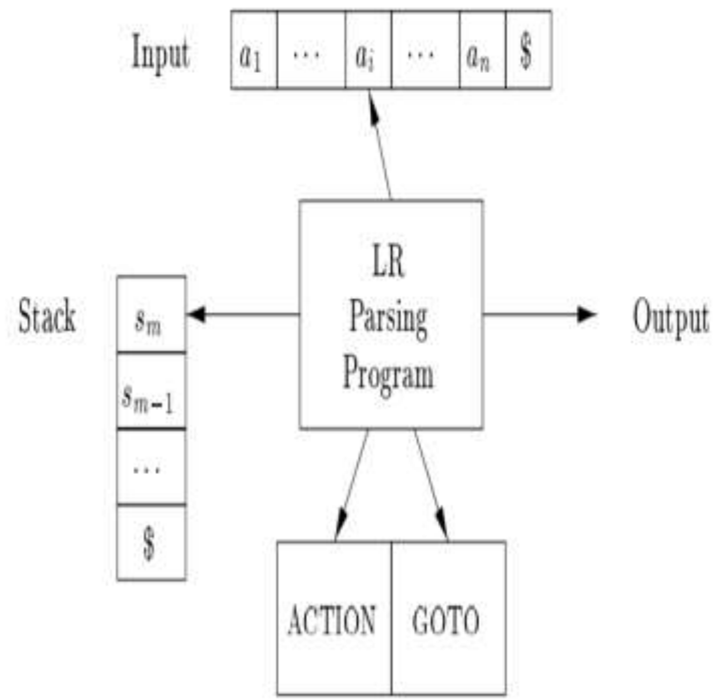
LR Parser

AMRITA VISHWA VIDYAPEETHAM School of Engineering,- Dept .of Computer Science & Engineering

LR Parser

- Type of bottom up parsing
- L is left to right scan of the given input string,
- R is Right Most derivation in reverse
- K is no of input symbols as the Look ahead.

Model of LR Parser



LR Parser Consists of

- An **input buffer** that contains the string to be parsed followed by a $ Symbol, used to indicate end of input.
- The **stack** holds a sequence of states, s_o, s_1, $\cdots$, S_m, where S_m is on the top.
- A **parsing table** (M), it is a two dimensional array M[state, terminal or Non terminal] and it contains two parts.

Types Of LR Parsers

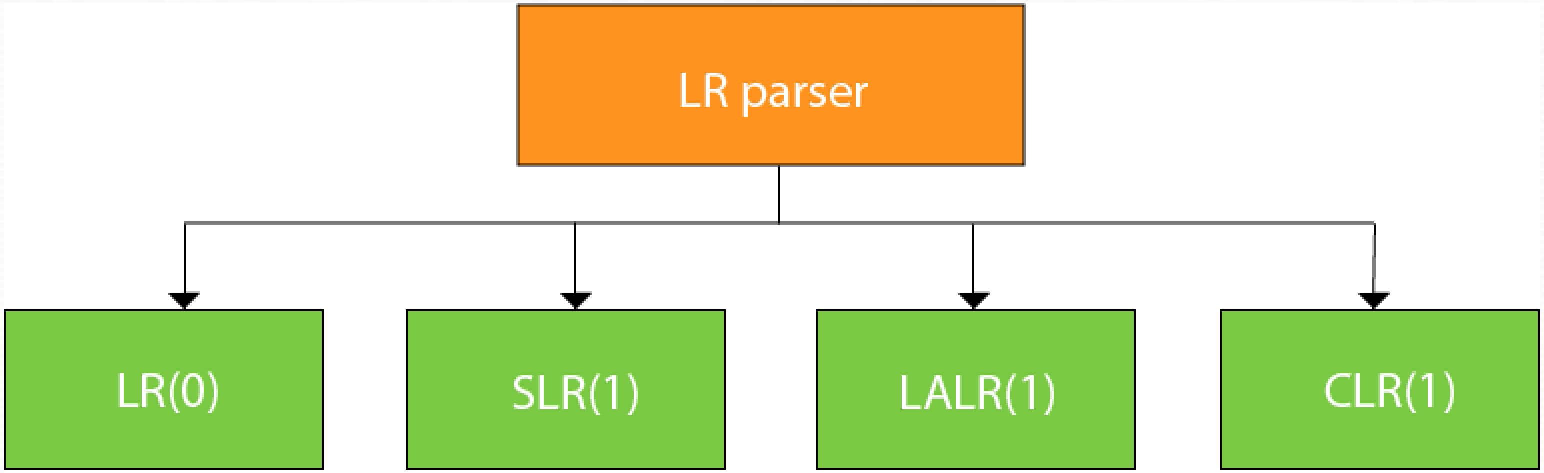


Fig: Types of LR parser

LALR Parser

LALR (1) Parsing:

- LALR refers to the lookahead LR.
- To construct the LALR (1) parsing table, we use the canonical collection of LR (1) items.
- In the LALR (1) parsing, the LR (1) items which have same productions but different look ahead are combined to form a single set of items

Various steps involved in the LALR (1) Parsing:

- For the given input string write a context free grammar
- Check the ambiguity of the grammar
- Add **Augment production** in the given grammar
- Create **Canonical collection of LR (1) items**
- Draw a data flow diagram (DFA)
- Construct a LALR (1) parsing table

Augmented Grammar

- Augmented grammar G` will be generated if we add one more production in the given grammar G.
- It helps the parser to identify when to stop the parsing and announce acceptance of the input.
- Example:

S → AA

A → aA | b

- The Augment grammar G` is represented by

S`→ S

S → AA

A → aA | b

LR (1) item

- LR (1) item is a collection of LR (0) items and a look ahead symbol.
- **LR (1) item = LR (0) item + look ahead**
- The look ahead always add $ symbol for the augment production.
  - $S' \rightarrow .S\ ,\ \$$

Canonical Collection of LR(0) items

- An LR (0) item is a production G with dot at some position on the right side of the 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 the LR (0), we place the reduce node in the entire row.

- For each symbol s (either a token or a nonterminal) that immediately follows a dot in an LR(1) item $[A \rightarrow \alpha \cdot s\beta, t]$ in set I, let Is be the set of all LR(1) items in I where s immediately follows the dot.

Closure Operation

- If ‘I’ is a set of items for a grammar G then closure of I (closure (I)) is set of items constructed from I by 2 rules:
  - Initially, every item in I is added to closure(I).
  - If $A \rightarrow \alpha \bullet B \beta$ is in closure (I) and $B \longrightarrow \gamma$ is a production,

  then add the item $B \longrightarrow \bullet\gamma$ to I, if it is not already in existence,

  we apply this rule until no more new items can be added to closure(I).

Goto Operation

- The function goto can be defined as, if there is a production $A \rightarrow \alpha \bullet B \beta$ then goto ($A \rightarrow \alpha \bullet B \beta$, B) = $A \rightarrow \alpha B \bullet \beta$. That means, simply shifting of “•” one position ahead over grammar symbol.

Example:

$S \rightarrow AA$

$A \rightarrow aA \mid b$

- Add Augment Production and insert '•' symbol at the first position for every production in G

$S` \rightarrow \bullet S$

$S \rightarrow \bullet AA$

$A \rightarrow \bullet aA$

$A \rightarrow \bullet b$

1. $S \rightarrow AA$
2. $A \rightarrow aA$
3. $A \rightarrow b$

Step 1:

- Add Augment Production, insert '•' symbol at the first position for every production in G and also add the lookahead.
    1. S` → •S, $
    2. S → •AA, $
    3. A → •aA, a/b
    4. A → •b, a/b

Add Augment production to the I0 State and Compute the Closure

- **I0 =** Closure (S` → •S) : Add all productions starting with S in to I0 State because "." is followed by the non-terminal. So, the I0 State becomes

I0 = S` → •S, $
S → •AA, $

Add all productions starting with A in modified I0 State because "." is followed by the non-terminal. So, the I0 State becomes.

I0= S` → •S, $
S → •AA, $
A → •aA, a/b
A → •b, a/b

- **I1=** Go to (I0, S) = closure (S` → S•, $) = S` → S•, $
- **I2=** Go to (I0, A) = closure (S → A•A, $)

Add all productions starting with A in I2 State because "." is followed by the non-terminal. So, the I2 State becomes

I2= S → A•A, $
A → •aA, $
A → •b, $

- **I3=** Go to (I0, a) = Closure (A → a•A, a/b)

Add all productions starting with A in I3 State because "." is followed by the non-terminal. So, the I3 State becomes

I3= A → a•A, a/b
    A → •aA, a/b
    A → •b, a/b

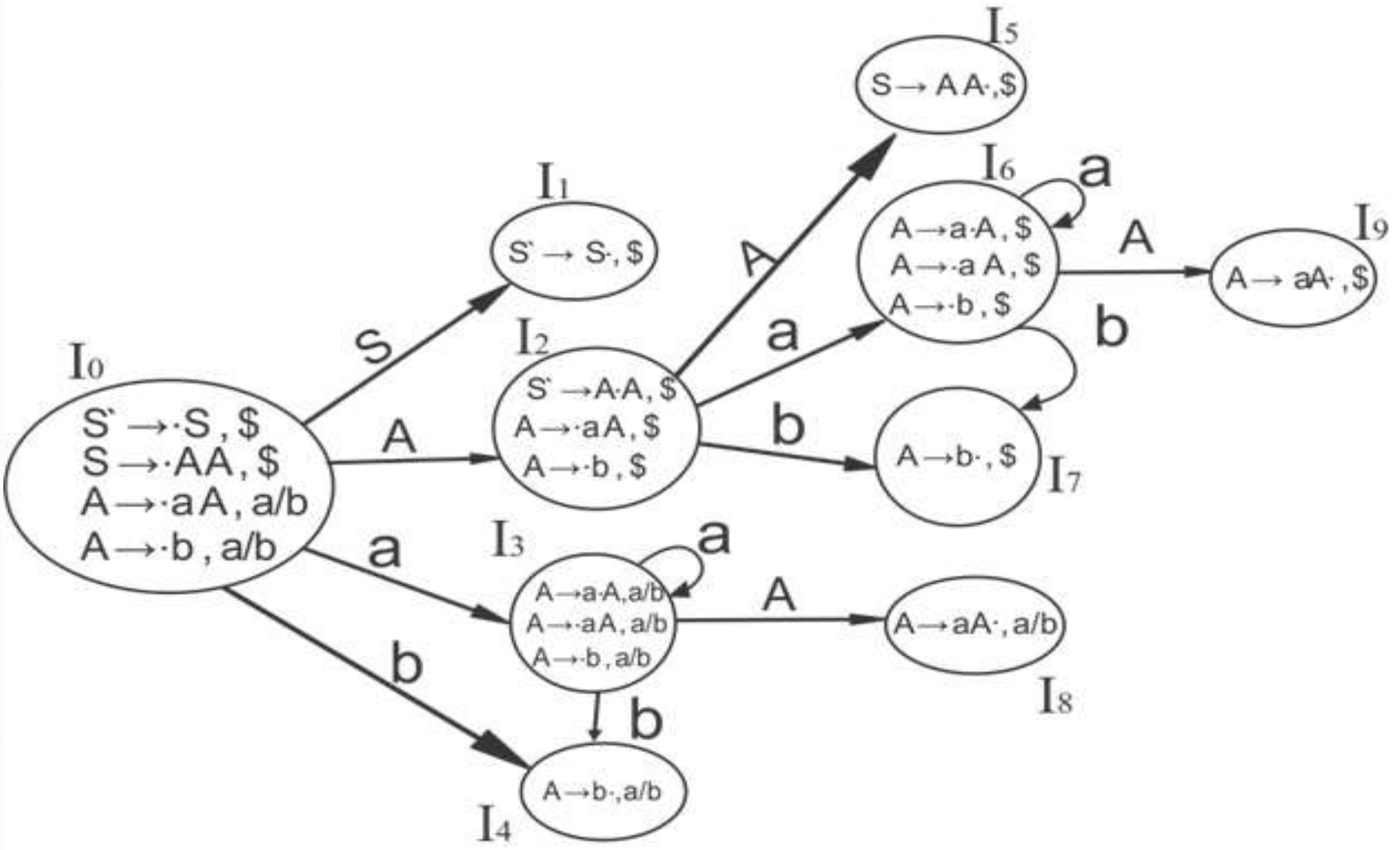
- Go to (I3, a) = Closure (A → a•A, a/b) = (same as I3)
- Go to (I3, b) = Closure (A → b•, a/b) = (same as I4)
- **I4=** Go to (I0, b) = closure (A → b•, a/b) = A → b•, a/b
- **I5=** Go to (I2, A) = Closure (S → AA•, $) =S → AA•, $
- **I6=** Go to (I2, a) = Closure (A → a•A, $)

  Add all productions starting with A in I6 State because "." is followed by the non-terminal. So, the I6 State becomes

I6 = A → a•A, $
    A → •aA, $
    A → •b, $

- Go to (I6, a) = Closure (A → a•A, $) = (same as I6)
  Go to (I6, b) = Closure (A → b•, $) = (same as I7)
- **I7=** Go to (I2, b) = Closure (A → b•, $) = A → b•, $
- **I8=** Go to (I3, A) = Closure (A → aA•, a/b) = A → aA•, a/b
- **I9=** Go to (I6, A) = Closure (A → aA•, $) = A → aA•, $

Drawing DFA:



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

- LALR table construction

- In the diagram, we can see that

- I3 and I6 are similar except their lookaheads.
- I4 and I7 are similar except their lookaheads.
- I8 and I9 are similar except their lookaheads.

- In LALR parsing table construction, we merge these similar states.
- Wherever there is 3 or 6, make it 36(combined form)
- Wherever there is 4 or 7, make it 47(combined form)
- Wherever there is 8 or 9, make it 89(combined form)

Parsing table (M),

- A parsing table (M), it is a two dimensional array M[state, terminal or Non terminal] and it contains two parts.
- 1.ACTION Part
  - The ACTION part of the table is a two dimensional array indexed by state and the input symbol, i.e. ACTION[state][input], An action table entry can have one of following four kinds of values in it. They are:
    - 1.Shift X, where X is a State number.
    - 2.Reduce X, where X is a Production number.
    - 3.Accept, signifying the completion of a successful parse.
    - 4.Error entry.
- 2.GO TO Part
  - The GO TO part of the table is a two dimensional array indexed by state and a Non terminal, i.e. GOTO[state][NonTerminal].
  - A GO TO entry has a state number in the table.

LR Parsing Algirithm.

- A parsing Algorithm uses the current State X, the next input symbol ‗a‘ to consult the entry at action[X][a]. it makes one of the four following actions as given below:
- 1.If the action[X][a]=shift Y, the parser executes a shift of Y and the state X on to the top of the stack and advances the input pointer.
- 2.If the action[X][a]= reduce Y (Y is the production number reduced in the State X), if the production is Y->β, then the parser pops 2*β symbols from the stack and push Y on to the Stack and PUSH goto[s,Y]
- 3.If the action[X][a]= accept, then the parsing is successful and the input string is accepted.
- 4.If the action[X][a]= error, then the parser has discovered an error and calls the error routine.

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()
```

	ACTION			GOTO	
	a	b	$	A	S
0	S36	S47		2	1
1			accept		
2	S36	S47		5	
36	S36	S47		89	
47	R3	R3			
5			R1		
36	S36	S47		89	
47			R3		
89	R2	R2			
89			R2		

- Now we have to remove the unwanted rows
- As we can see, 36 row has same data twice, so we delete 1 row.
- We combine two 47 row into one by combining each value in the single 47 row.
- We combine two 89 row into one by combining each value in the single 89 row.
-

- The final LALR table looks like the below.

	ACTION			GOTO	
	a	b	$	A	S
0	S36	S47		2	1
1			accept		
2	S36	S47		5	
36	S36	S47		89	
47	R3	R3	R3		
5			R1		
89	R2	R2	R2		

Stack	Input	Action
0	aaabb$	Shift 36
0a36	aabb$	Shift 36
0a36a36	abb$	Shift 36
0a36a36a36	bb$	Shift 47
0a36a36a36**b47**	b$	Reduce3, A→b
0a36a36a**36A89**	b$	Reduce2, A→aA
0a36a**36A89**	b$	Reduce2, A→aA
0a36A89	b$	Reduce2, A→aA
0A2	b$	Shift 47
0A2**b47**	$	Reduce3, A→b
0**A2A5**	$	Reduce1, S→AA
0S1	$	Accept

0. $S' \to S$

1. $S \to C\,C$

2. $C \to \mathbf{c}\,C$

3. $C \to \mathbf{d}$

State 0.

[S' → · S, $]
[S → · C C, $]
[C → · c C, c]
[C → · c C, d]
[C → · d, c]
[C → · d, d]
On S goto 1
On C goto 2
On c goto 3
On d goto 4

State 1.

[S' → S ·, $]

State 2.

[S → C · C, $]
[C → · c C, $]
[C → · d, $]
On C goto 5
On c goto 6
On d goto 7

State 3.

[C → c · C, c]
[C → c · C, d]
[C → · c C, c]
[C → · c C, d]
[C → · d, c]
[C → · d, d]
On C goto 8
On c goto 3
On d goto 4

State 4.

[C → d ·, c]
[C → d ·, d]

State 5.

[S → C C ·, $]

State 6.

[C → c · C, $]
[C → · c C, $]
[C → · d, $]
On C goto 9
On c goto 6
On d goto 7

State 7.

[C → d ·, $]

State 8.

[C → c C ·, c]
[C → c C ·, d]

State 9.

[C → c C ·, $]

Merge the states with same LR(0) but with different lookahead

State (3,6).

[C → c · C, c/d/$]
[C → · c C, c/d/$]
[C → · d, c/d/$]
On C goto (8,9)
On c goto (3,6)
On d goto (4,7)

State (4,7).

[C → d ·, c/d/$]

State (8,9).

[C → c C ·, c/d/$]

The LALR(1) parsing table is as follows.

	Actions			Goto	
	c	d	$	S	C
0	s(3,6)	s(4,7)		1	2
1			acc		
2	s(3,6)	s(4,7)			5
(3,6)	s(3,6)	s(4,7)			(8,9)
(4,7)	r3	r3	r3		
5			r1		
(8,9)	r2	r2	r2		