LECTURE NOTES

COMPILER DESIGN
UNIT-3

Name Of The Programme	**B.Tech-CSE**
Name Of The Subject	**Compiler Design**
Regulations	**R-20**
Year and Semester	**3rd Year, 2nd Semester**
Faculty Name	**Mr.Siva Kumar Ronanki**

COLLEGE OF ENGINEERING
(AUTONOMOUS)

GAYATRI VIDYA PARISHAD COLLEGE OF ENGINEERING (Autonomous)
Approved by AICTE, New Delhi and Affiliated to Andhra University
Re-accredited by NAAC with "A" Grade with a CGPA of 3.47/4.00
Madhurawada, Visakhapatnam - 530 048.

BOTTOM-UP PARSING

Bottom-up parsing corresponds to the construction of a parse tree for an input string beginning at the leaves (the bottom nodes) and working up towards the root (the top node). It involves ‑reducing an input string _w‘ to the Start Symbol of the grammar. in each reduction step, a perticular substring matching the right side of the production is replaced by symbol on theleft of that production and it is the Right most derivation. For example consider the following Grammar.

E->E+T/T

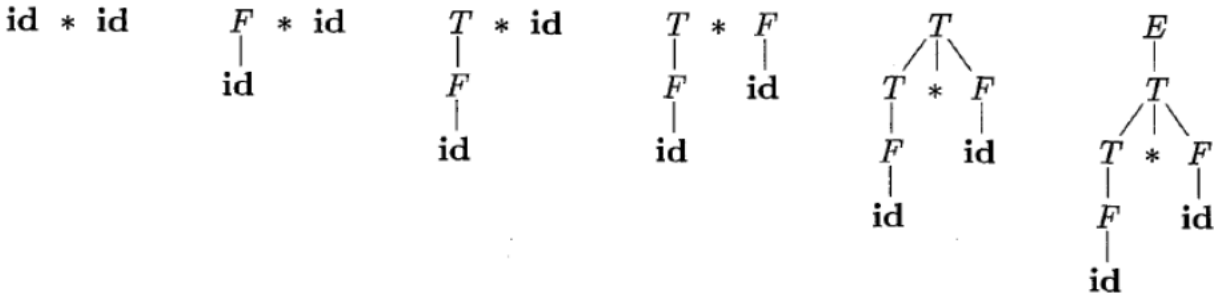
T->T*F/f

F->(E)/id

Bottom up parsing of the input string **“id * id “**is as follows:

INPUT STRING	SUB STRING	REDUCING PRODUCTION
id*id	Id	F->id
F*id	T	F->T
T*id	Id	F->id
T*F	*	T->T*F
T	T*F	E->T
E		Start symbol. Hence, the input String is accepted

Parse Tree representation is as follows:



A Bottom-up Parse tree for the input String “id*id”

Bottom up parsing is classified in to 1. Shift-Reduce Parsing, 2. Operator Precedence parsing , and 3. L R Parsing

i. SLR(1)
ii. CLR (1)
iii. LALR(1)

SHIFT-REDUCE PARSING:

Shift-reduce parsing is a form of bottom-up parsing in which a stack holds grammar symbols and an input buffer holds the rest of the string to be parsed, We use $ to mark the bottom of the stack

and also the right end of the input. And it makes use of the process of shift and reduce actions to accept the input string. Here, the parse tree is Constructed bottom up from the leaf nodes towards the root node.

When we are parsing the given input string, if the match occurs the parser takes the reduce action otherwise it will go for shift action. And it can accept ambiguous grammars also.

For example, consider the below grammar to accept the input string –id * id–, using S-R parser

E->E+T/T

T->T*F/F

F->(E)/id

Actions of the Shift-reduce parser using Stack implementation

STACK	INPUT	ACTION
$	Id*id$	Shift
$id	*id$	Reduce with F→ d
$F	*id$	Reduce with T→ F
$T	*id$	Shift
$T*	id$	Shift
$T*id	$	Reduce with F →id
$T*F	$	Reduce with T →T*F
$T	$	Reduce with E →T
$E	$	Accept

OPERATOR PRECEDENCE PARSING:

Operator precedence grammar is kinds of shift reduce parsing method that can be applied to a small class of operator grammars. And it can process ambiguous grammars also.

- An operator grammar has two important characteristics:
  1. There are no € productions.
  2. No production would have two adjacent non terminals.

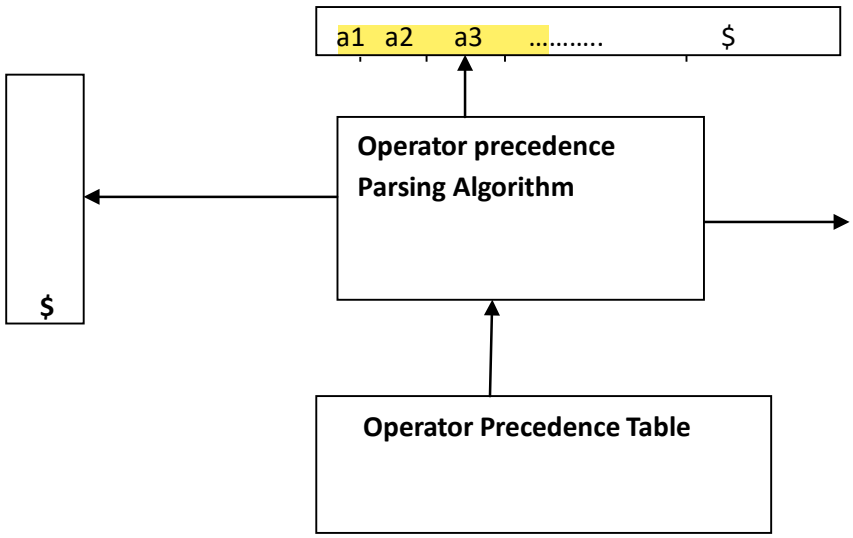
Two main Challenges in the operator precedence parsing are:

1. Identification of Correct handles in the reduction step, such that the given input should bereduced to starting symbol of the grammar.
2. Identification of which production to use for reducing in the reduction steps, such that weshould correctly reduce the given input to the starting symbol of the grammar.

Operator precedence parser consists of:

1. An input buffer that contains string to be parsed followed by a$, a symbol used toindicate the ending of input.
2. A stack containing a sequence of grammar symbols with a $ at the bottom of the stack.
3. An operator precedence relation table O, containing the precedence ralations between the pair of terminal. There are three kinds of precedence relations will exist between the pair of terminal pair _a‘ and _b‘ as follows:

4. The relation a<•b implies that he terminal _a‘ has lower precedence than terminal _b‘.
5. The relation a•>b implies that he terminal _a‘ has higher precedence than terminal _b‘.
6. The relation a=•b implies that he terminal _a‘ has lower precedence than terminal _b‘.
7. An operator precedence parsing program takes an input string and determines whether it conforms to the grammar specifications. It uses an operator precedence parse table and stack to arrive at the decision.



Example, If the grammar is

E->E+E

E->E-E

E->E*E

E->E/E

E->E^E

E->-E

E->(E)

E->id, Construct operator precedence table and accept input string **“ id+id*id”**

The precedence relations between the operators are
(id) > (^) > (* /) > (+ -) > $, „^“ operator is Right Associative and reaming all operatorsare e Left Associative

	+	-	*	/	^	id	(	)	$
+	•>	•>	<•	<•	<•	<•	<•	•>	•>
-	•>	•>	<•	<•	<•	<•	<•	•>	•>
*	•>	•>	•>	•>	<•	<•	<•	•>	•>
/	•>	•>	•>	•>	<•	<•	<•	•>	•>
^	•>	•>	•>	•>	<•	<•	<•	•>	•>
Id	•>	•>	•>	•>	•>	**Err**	**Err**	•>	•>
(	<•	<•	<•	<•	<•	<•	<•	=	**Err**
)	•>	•>	•>	•>	•>	**Err**	**Err**	•>	•>
$	<•	<•	<•	<•	<•	<•	<•	**Err**	**Err**

The intention of the precedence relations is to delimit the handle of the given input String with <•marking the left end of the Handle and •> marking the right end of the handle.

Parsing Action:

To locate the handle following steps are followed:

1. Add $ symbol at the both ends of the given input string.
2. Scan the input string from left to right until the right most •> is encountered.
3. Scan towards left over all the equal precedence‘s until the first <• precedence isencountered.
4. Every thing between <• and •> is a handle.
5. $ on S means parsing is success.

Example, Explain the parsing Actions of the OPParser for the input string is **“id*id”** and thegrammar is:

E->E+E

E->E*E

E->id

1. $ <• id •> *<• id•> $

The first handle is _id‘ and match for the _id _in the grammar is E->id .So, id is replaced with the Non terminal E. the given input string can be written as

2. $ <• E •> *<• id•> $

The parser will not consider the Non terminal as an input. So, they are notconsidered in the input string. So , the string becomes.

3. $ <• *<• id•> $

The next handle is _id‘ and match for the _id _in the grammar is E->id .So, id is replaced with the Non terminal E. the given input string can be written as

4. $ <• *<• E•> $

The parser will not consider the Non terminal as an input. So, they are notconsidered in the input string. So, the string becomes

5. $ <• * •> $

The next handle is _*‘ and match for the _ _in the grammar is E->E * E .So, id is replaced with the Non terminal E. the given input string can be written as

6. **$ E $**

The parser will not consider the Non terminal as an input. So, they are not considered in the input string. So, the string becomes

7. **$ $**

$ On $ means parsing successful

Operator Parsing Algorithm:

The operator precedence Parser parsing program determines the action of the parser dependingon

1. _a‘ is top most symbol on the Stack
2. _b‘ is the current input symbol

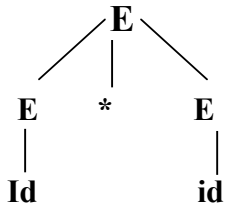
There are 3 conditions for _a‘ and _b‘ that are important for the parsing program

1. a=b=$, the parsing is successful
2. a <• b or a = b, the parser shifts the input symbol on to the stack and advances theinput pointer to the next input symbol.
3. a •> b, parser performs the reduce action. The parser pops out elements one byone from the stack until we find the current top of the stack element has lowerprecedence than the most recently popped out terminal.

Example, the sequence of actions taken by the parser using the stack for the input string −id * id

— and corresponding Parse Tree are as under.

STACK	INPUT	OPERATIONS
$	id * id $	$ <• id, shift _id‘ in to stack
$ id	*id $	id •> *, reduce _id‘ using E-> id
$E	*id $	$ <• *, shift _*‘ in to stack
$E*	id$	* <• id , shift _id‘ in to Stack
$E*id	$	id •> $, reduce _id‘ using E->id
$E*E	$	*•> $, reduce _*‘ using E->E*E
$E	$	$=$=$, so parsing is successful



Advantages and Disadvantages of Operator Precedence Parsing:

The following are the advantages of operator precedence parsing

1. It is simple and easy to implement parsing technique.
2. The operator precedence parser can be constructed by hand

after understanding thegrammar. It is simple to debug.

The following are the disadvantages of operator precedence parsing:

1. It is difficult to handle the operator like _-_which can be either unary or binary and hencedifferent precedence‘s and associativities.
2. It can parse only a small class of grammar.
3. New addition or deletion of the rules requires the parser to be re written.
4. Too many error entries in the parsing tables.

LR Parsing:

Most prevalent type of bottom up parsing is LR (k) parsing. Where, L is left to right scan of the given input string, R is Right Most derivation in reverse and K is no of input symbols as the Look ahead.

- It is the most general non back tracking shift reduce parsing method
- The class of grammars that can be parsed using the LR methods is a proper superset ofthe class of grammars that can be parsed with predictive parsers.
- An LR parser can detect a syntactic error as soon as it is possible to do so, on a left toright scan of the input.

LR Parser Consists of

- An input buffer that contains the string to be parsed followed by a $ Symbol, used toindicate end of input.
- A stack containing a sequence of grammar symbols with a $ at the bottom of the stack,which initially contains the Initial state of the parsing table on top of $.
- A parsing table (M), it is a two dimensional array M[state, terminal or Non terminal] andit contains two parts.

1. ACTION Part

The ACTION part of the table is a two dimensional array indexed by state and theinput 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.

- 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 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.

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.

The parsing is classified in to

1. LR (0)
2. Simple LR (1)
3. Canonical LR (1)
4. Look ahead LR (1)

LR(0) PARSER:

The LR parser is an efficient bottom-up syntax analysis technique that can be used for a large class of context-free grammar. This technique is also called LR(0) parsing.
L stands for the left to right scanning
R stands for rightmost derivation in reverse
0 stands for no. of input symbols of lookahead.

Augmented grammar :

If G is a grammar with starting symbol S, then G' (augmented grammar for G) is a grammar with a new starting symbol S' and productions S'-> .S . The purpose of this new starting production is to indicate to the parser when it should stop parsing. The ' . ' before S indicates the left side of ' . ' has been read by a compiler and the right side of ' . ' is yet to be read by a compiler.

Steps for constructing the LR parsing table :

1. Writing augmented grammar
2. LR(0) collection of items to be found
3. Defining 2 functions: goto(list of non-terminals) and action(list of terminals) in the parsing table.

Q. Construct an LR parsing table for the given context-free grammar –

S–>AA
A–>aA|b

Solution :

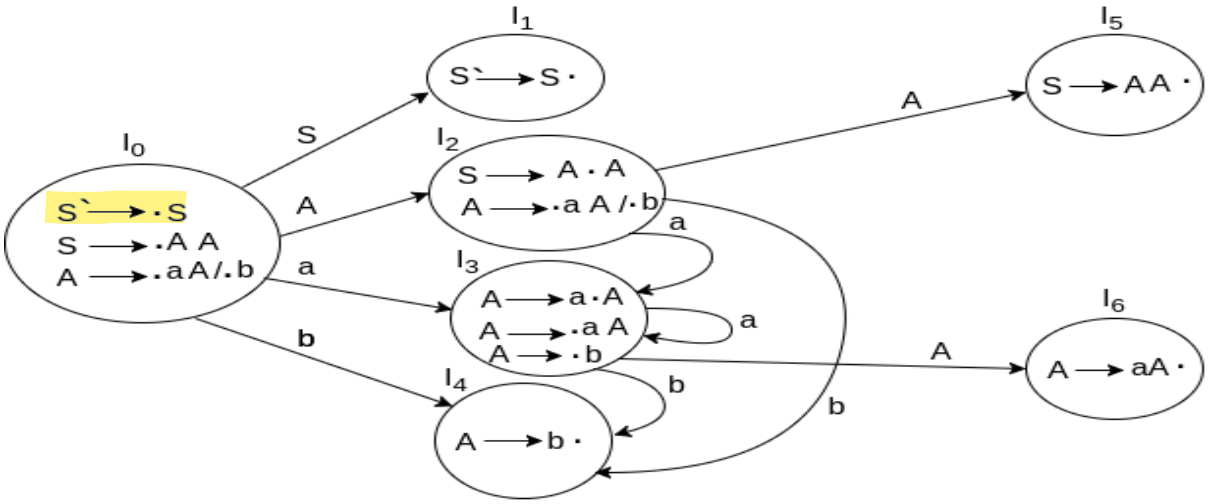
STEP 1- Find augmented grammar –
The augmented grammar of the given grammar is:-

S'–>.S [0th production]
S–>.AA [1st production]
A–>.aA [2nd production]
A–>.b [3rd production]

STEP 2 – Find LR(0) collection of items
Below is the figure showing the LR(0) collection of items.

Drawing DFA:

The DFA contains the 7 states I0 to I6.



Productions are numbered as follows:

1. S → AA ... (1)
2. A → aA ... (2)
3. A → b ... (3)

- I1 contains the final item which drives(S` → S•), so action {I1, $} = Accept.
- I4 contains the final item which drives A → b• and that production corresponds to the production number 3 so write it as r3 in the entire row.
- I5 contains the final item which drives S → AA• and that production corresponds to the production number 1 so write it as r1 in the entire row.
- I6 contains the final item which drives A → aA• and that production corresponds to the production number 2 so write it as r2 in the entire row.

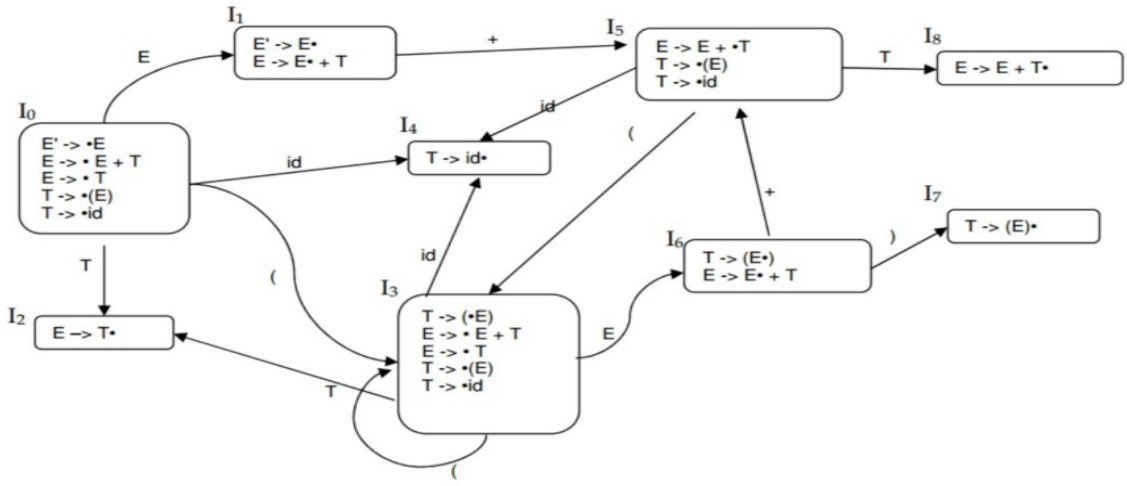
Ex2. Construct an LR parsing table for the given context-free grammar

E→E+T
E→T
T→(E)
T→id

STEP 1- Find augmented grammar –
The augmented grammar of the given grammar is:-
E'→E [0th production]
E→.E+T [1st production]
E→.T [2nd production]
T→.(E) [3rd production]
T→.id [4th production]

STEP 2 – Find LR(0) collection of items
Below is the figure showing the LR(0) collection of items.

Drawing DFA:

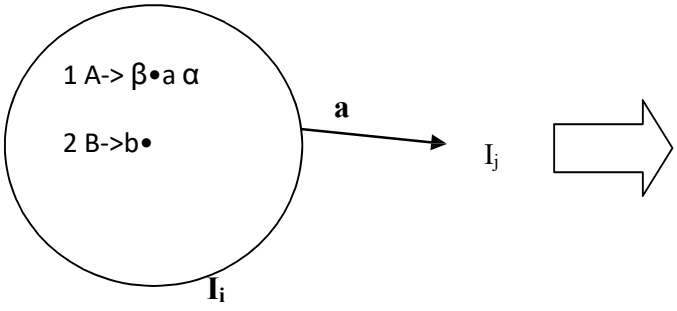


LR(0) Table

State	Action					Goto	
	Id	(	)	+	$	E	T
0	S4	S3				1	2
1				S5	Accept		
2	r2	r2	r2	r2	r2		
3	S4	S3				6	2
4	r4	r4	r4	r4	r4		
5	S4	S3					8
6			S7	S5			
7	r3	r3	r3	r3	r3		
8	r1	r1	r1	r1	r1		

Shift-Reduce Conflict in LR (0) Parsing: Shift Reduce Conflict in the LR (0) parsingoccurs when a state has

1. A Reduced item of the form A → α• and
2. An incomplete item of the form A.→β•aα as shown below:

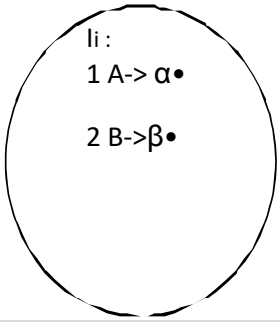


States	Action		GOTO	
	a	$	A	B
Ii	**Sj/r2**	**r2**		
Ij				

Reduce - Reduce Conflict in LR (0) Parsing:

Reduce- Reduce Conflict in the LR (1) parsing occurs when a state has two or morereduced items of the form

1. **A → α•**
2. **B → β• as shown below:**



I_i

States	Action		GOTO	
	a	$	A	B
I_i	Sj/r2	r_2		
I_j				

SLR-PARSER:

SLR is simple LR. It is the smallest class of grammar having few number of states. SLR is very easy to construct and is similar to LR parsing. The only difference between SLR parser and LR(0) parser is that in LR(0) parsing table, there's a chance of 'shift reduced' conflict because we are entering 'reduce' corresponding to all terminal states. We can solve this problem by entering 'reduce' corresponding to FOLLOW of LHS of production in the terminating state. This is called SLR(1) collection of items

Steps for constructing the SLR parsing table :

1. Writing augmented grammar
2. LR(0) collection of items to be found
3. Find FOLLOW of LHS of production
4. Defining 2 functions:goto[list of terminals] and action[list of non-terminals] in the parsing table.

EXAMPLE – Construct LR parsing table for the given context-free **grammar**

S–>AA

A–>aA|b

Solution:

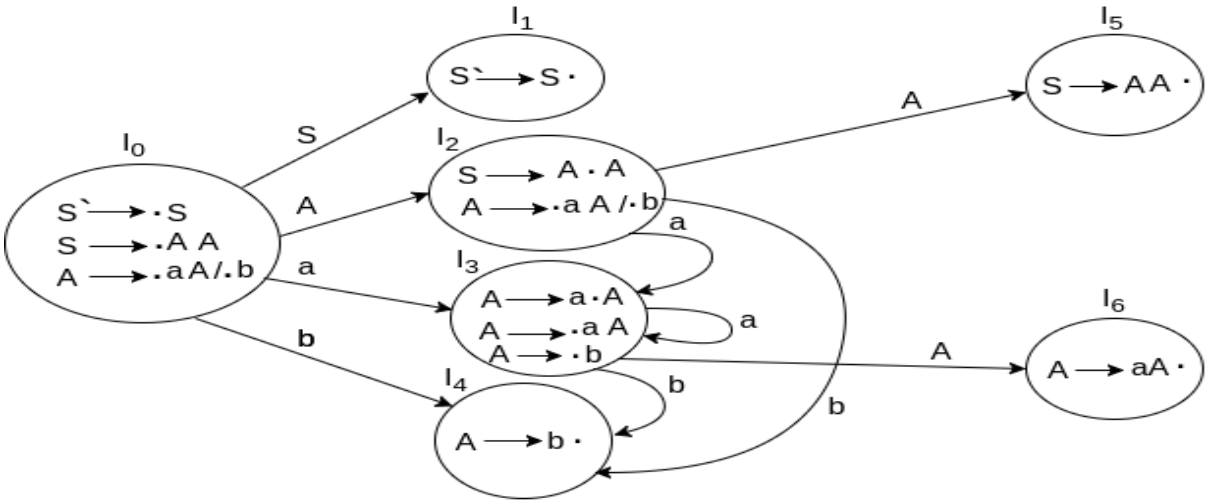
STEP1 – Find augmented grammar

The augmented grammar of the given grammar is:-

S'–>.S [0th production]
S–>.AA [1st production]
A–>.aA [2nd production]
A–>.b [3rd production]

STEP2 – Find LR(0) collection of items

Below is the figure showing the LR(0) collection of items.



STEP3 –
Find FOLLOW of LHS of production
FOLLOW(S)=$
FOLLOW(A)=a,b,$

STEP 4-
Defining 2 functions: goto[list of non-terminals] and action[list of terminals] in the parsing table. Below is the SLR parsing table.

	ACTION			GOTO	
	a	b	$	A	S
0	S3	S4		2	1
1			accept		
2	S3	S4		5	
3	S3	S4		6	
4	R3	R3	R3		
5			R1		
6	R2	R2	R2		

- $ is by default a nonterminal that takes accepting state.
- 0,1,2,3,4,5,6 denotes I0,I1,I2,I3,I4,I5,I6
- I0 gives A in I2, so 2 is added to the A column and 0 rows.
- I0 gives S in I1,so 1 is added to the S column and 1 row.
- similarly 5 is written in A column and 2 row, 6 is written in A column and 3 row.
- I0 gives a in I3 .so S3(shift 3) is added to a column and 0 row.
- I0 gives b in I4 .so S4(shift 4) is added to the b column and 0 row.
- Similarly, S3(shift 3) is added on a column and 2,3 row ,S4(shift 4) is added on b column and 2,3 rows.
- I4 is reduced state as ‘ . ‘ is at the end. I4 is the 3rd production of grammar(A–>.b).LHS of this production is A. FOLLOW(A)=a,b,$. write r3(reduced 3) in the columns of a,b,$ and 4th row

- I5 is reduced state as ' . ' is at the end. I5 is the 1st production of grammar(S–>.AA). LHS of this production is S.
  FOLLOW(S)=$. write r1(reduced 1) in the column of $ and 5th row
- I6 is a reduced state as ' . ' is at the end. I6 is the 2nd production of grammar(A–>.aA). The LHS of this production is A.
  FOLLOW(A)=a,b,$. write r2(reduced 2) in the columns of a,b,$ and 6th row

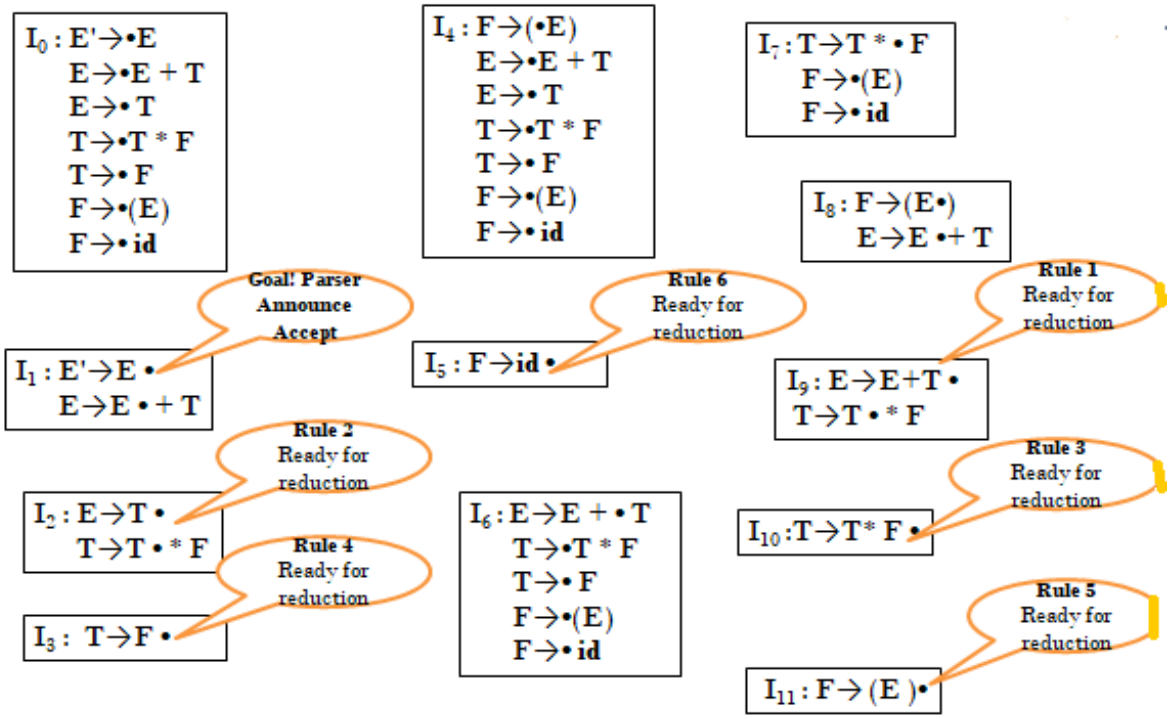
Example: Implement SLR Parser for the given grammar:
1.E→E+T
2.E→T
3.T→T*F
4.T→F
5.F→(E)
6.F→id

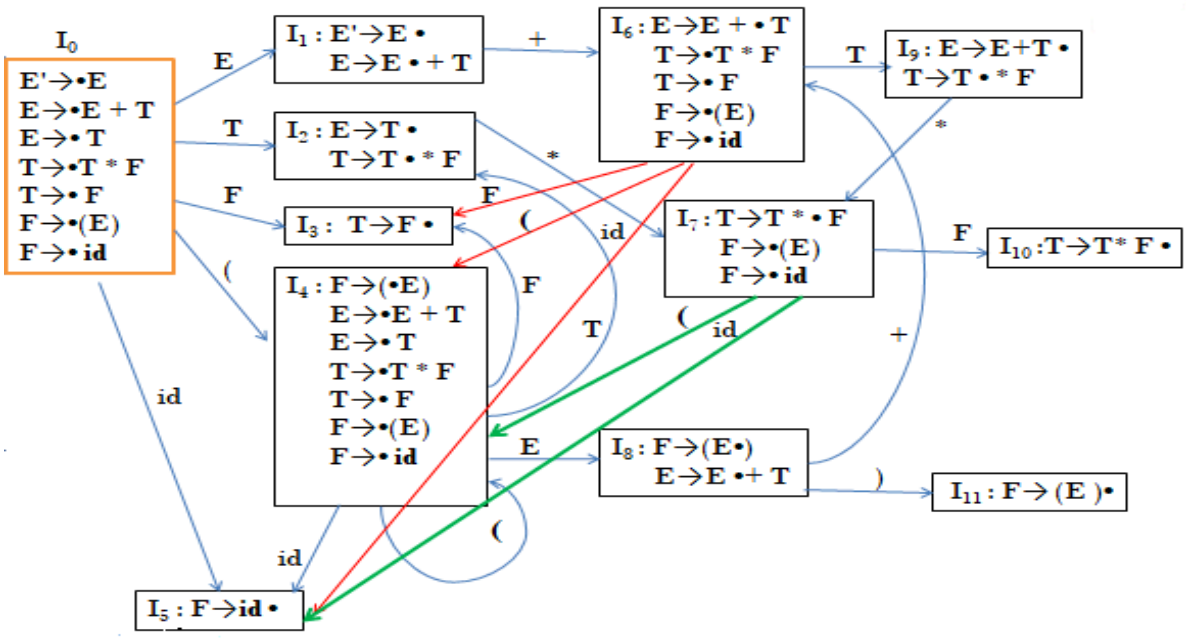
Step 1: Convert given grammar into augmented grammar.

Augmented grammar:

0.E'→E
1.E→E+T
2.E→T
3.T→T*F
4.T→F
5.F→(E)
6.F→id

Step 2 : Find LR (0) items





Step 3 : Construction of Parsing table.
Computation of FOLLOW is required to fill the reduction action in the ACTION part of the table.

FOLLOW(E) = {+,),$ }
FOLLOW(T) ={*,+,) ,$}
FOLLOW(F) ={*,+,) ,$}

STEP 4-
Defining 2 functions: goto[list of non-terminals] and action[list of terminals] in the parsing table. Below is the SLR parsing table.

State	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

1. si means shift and stack state i.

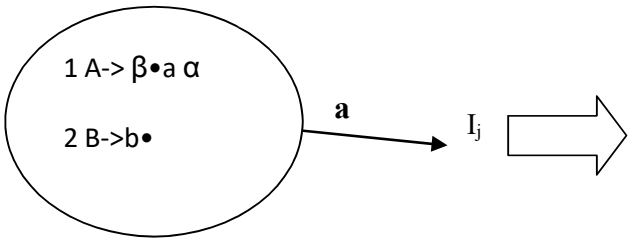
2. rj means reduce by production numbered j.

3. acc means accept.

4. Blank means error.

Shift-Reduce Conflict in SLR (1) Parsing : Shift Reduce Conflict in the LR (1) parsing occurswhen a state has

1. A Reduced item of the form A →α• and Follow(A) includes the terminal value _a'.
2. An incomplete item of the form β →•aα as shown below:

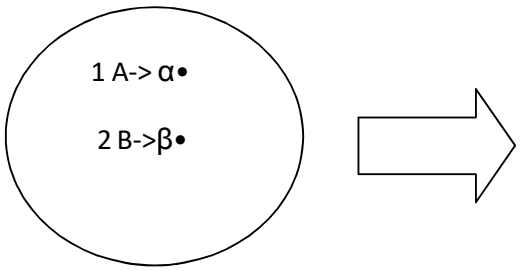


States	Action		GOTO	
	a	$	A	B
I_i	Sj/r2			

Reduce - Reduce Conflict in SLR (1) Parsing

Reduce- Reduce Conflict in the LR (1) parsing occurs when a state has two or morereduced items of the form

1. **A → α•**
2. **B → β• and Follow (A) ∩ Follow(B) ≠ null as shown below:**



States	Action		GOTO	
	a	$	A	B
I_i	r_1/r2			

CLR (1) Parsing

CLR refers to canonical lookahead. CLR parsing use the canonical collection of LR (1) items to build the CLR (1) parsing table. CLR (1) parsing table produces the more number of states as compare to the SLR (1) parsing.

In the CLR (1), we place the reduce node only in the lookahead symbols.

Various steps involved in the CLR (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 (0) items
- Draw a data flow diagram (DFA)
- Construct a CLR (1) parsing table

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 is used to determine that where we place the final item.

The look ahead always add $ symbol for the argument production.

CLR (1) Grammar

1. S → AA
2. A → aA
3. A → b

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

I0 State:

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, $

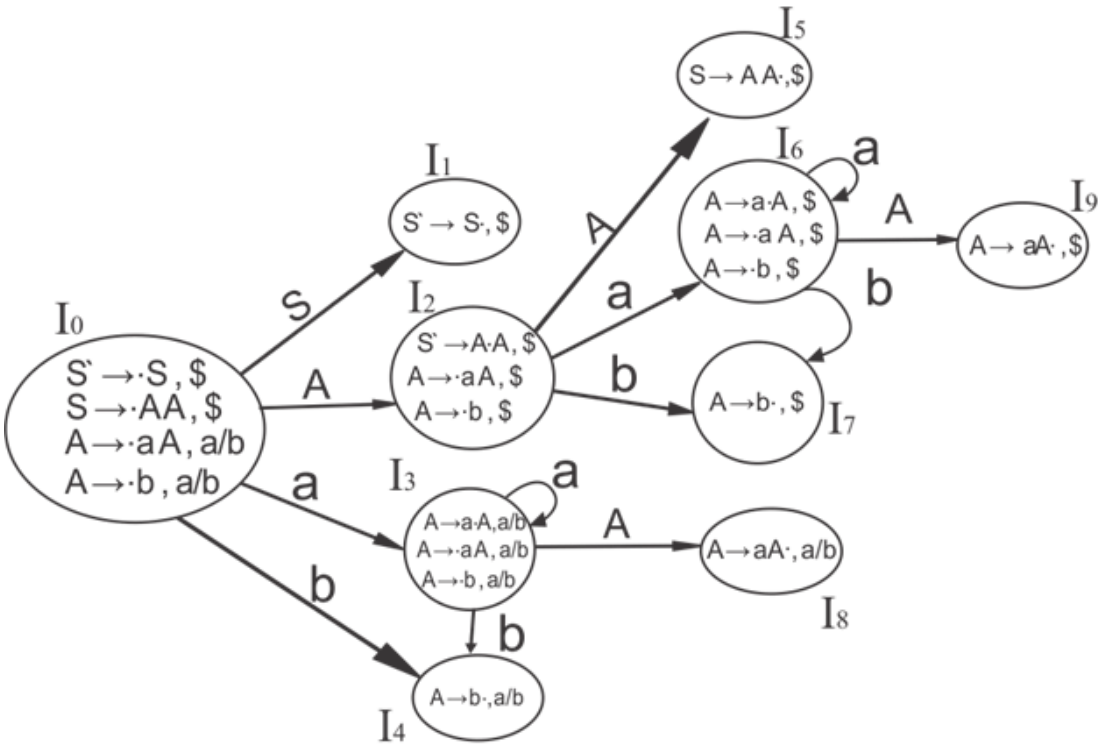
I3= Go to (I0, a) = Closure (A → a•A, a/b)

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:



CLR (1) Parsing table:

States	a	b	S	S A
I_0	S_3	S_4		2
I_1			Accept	
I_2	S_6	S_7		5
I_3	S_3	S_4		8
I_4	R_3	R_3		
I_5			R_1	
I_6	S_6	S_7		9
I_7			R_3	
I_8	R_2	R_2		
I_9			R_2	

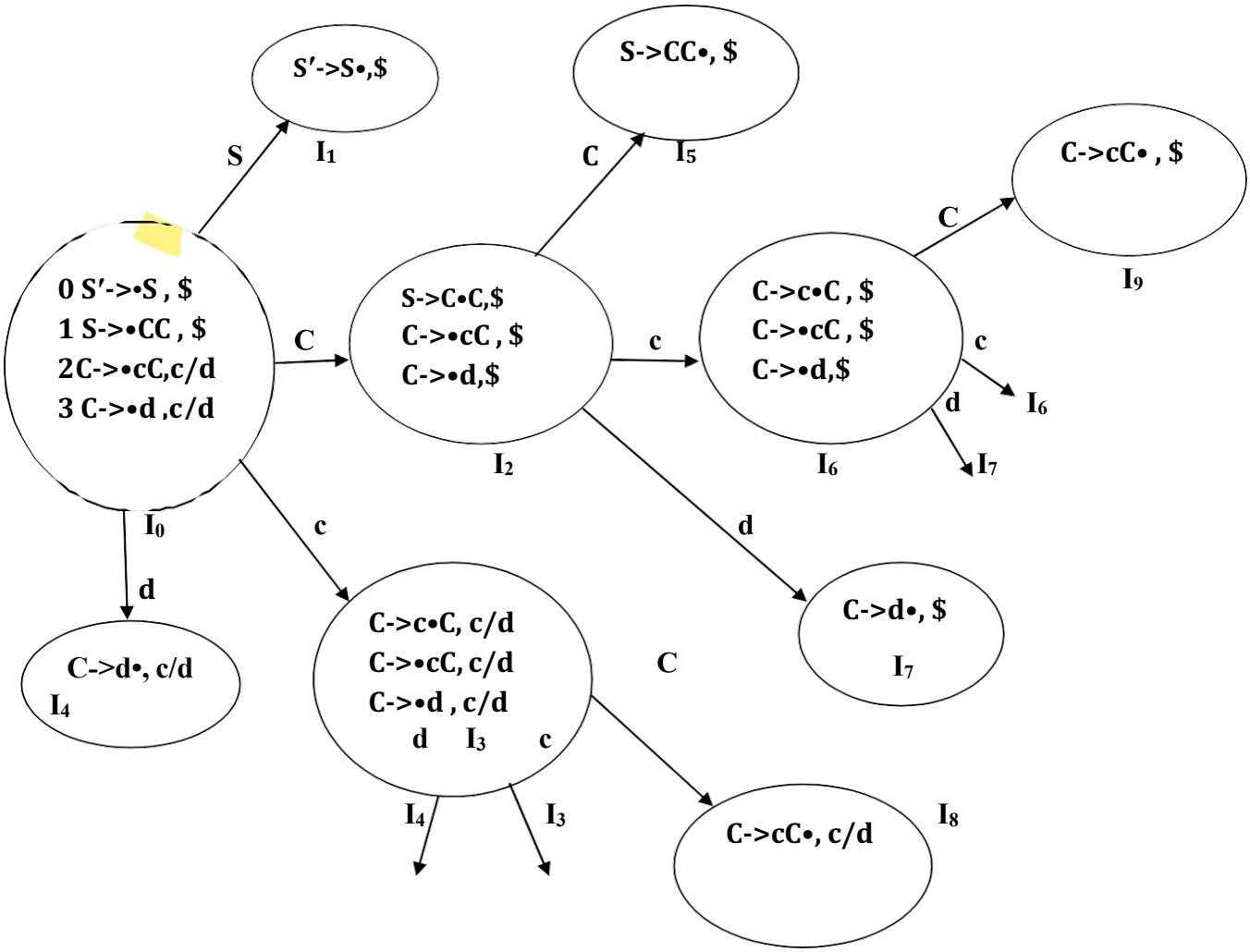
EXAMPLE:

S->CC
C->cC.
C->d

The Canonical collection of LR (1) items can be created as follows:

S′->•S (Augment Production)
S->•CC
C->•cC
C->•d

Drawing the Finite State Machine DFA for the above LR (1) items



Construction of CLR (1) Table

Rule1: if there is an item [A->α•Xβ,b] in I_i and goto(I_i,X) is in I_j then action [I_i][X]= Shiftj, Where X is Terminal.
Rule2: if there is an item [A->α•, b] in I_i and (A≠S`) set action [I_i][b]= reduce along withthe production number.

Rule3: if there is an item [S`->S•, $] in I_i then set action $[I_i][\$]$= Accept.
Rule4: if there is an item [A->α•Xβ,b] in I_i and go to(I_i,X) is in I_j then goto $[I_i][X]$= j,Where X is Non Terminal.

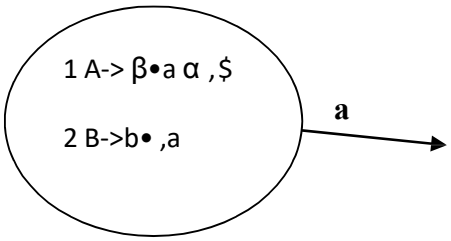
States	ACTION			GOTO	
	c	d	$	S	C
I_0	S_3	S_4		1	2
I_1			ACCEPT		
I_2	S_6	S_7			5
I_3	S_3	S_4			8
I_4	R_3	R_3			5
I_5			R_1		
I_6	S_6	S_7			9
I_7			R_3		
I_8	R_2	R_2			
I_9			R_2		

Conflicts in the CLR (1) Parsing : When multiple entries occur in the table. Then, the situation is said to be a Conflict.

Shift-Reduce Conflict in CLR (1) Parsing

Shift Reduce Conflict in the CLR (1) parsing occurs when a state has

0. A Reduced item of the form A → α•, a and
1. An incomplete item of the form A →β•aα as shown below:

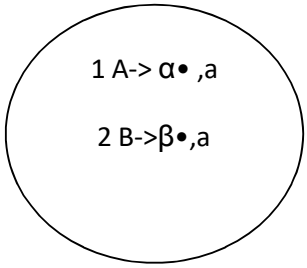


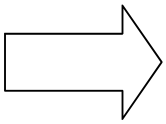
States	Action		GOTO	
	a	$	A	B
I_i	Sj/r2			

Reduce / Reduce Conflict in CLR (1) Parsing

Reduce- Reduce Conflict in the CLR (1) parsing occurs when a state has two or morereduced items of the form

3. **A → α•**
4. **B → β• If two productions in a state (I) reducing on same look ahead symbolas shown below:**





States	Action		GOTO	
	a	$	A	B
I_i	$r_1/r2$			

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

LALR (1) parsing is same as the CLR (1) parsing, only difference in the parsing table.

Example

1.S → AA
2.A → aA
3.A → b

Add Augment Production, insert '•' symbol at the first position for every production in G and also add the lookahead.

4.S` → •S, $
5.S → •AA, $
6.A → •aA, a/b
7.A → •b, a/b

I0 State:

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•, $

If we analyze then LR (0) items of I3 and I6 are same but they differ only in their lookahead.
I3 = { A → a•A, a/b
A → •aA, a/b
A → •b, a/b }
I6= { A → a•A, $
A → •aA, $
A → •b, $ }
Clearly I3 and I6 are same in their LR (0) items but differ in their lookahead, so we can combine them and called as I36.
I36 = { A → a•A, a/b/$
A → •aA, a/b/$

A → •b, a/b/$ }

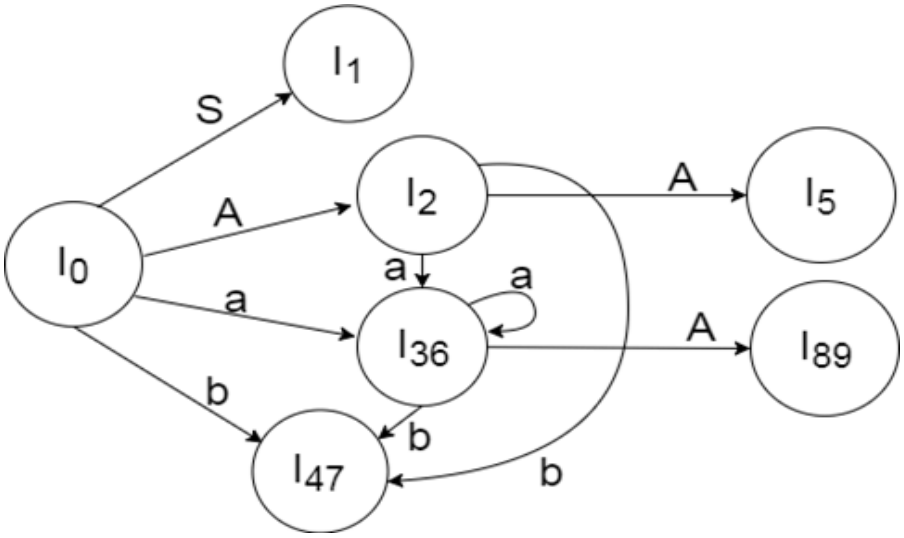
The I4 and I7 are same but they differ only in their look ahead, so we can combine them and called as I47.

I47 = {A → b•, a/b/$}

The I8 and I9 are same but they differ only in their look ahead, so we can combine them and called as I89.

I89 = {A → aA•, a/b/$}

Drawing DFA:



LALR (1) Parsing table:

States	a	b	$	S	A
I_0	S_{36}	S_{47}		12	
I_1		accept			
I_2	S_{36}	S_{47}			5
I_{36}	$S_{36}S_{47}$				89
I_{47}	R_3R_3	R_3			
I_5			R_1		
I_{89}	R_2	R_2	R_2		

YACC

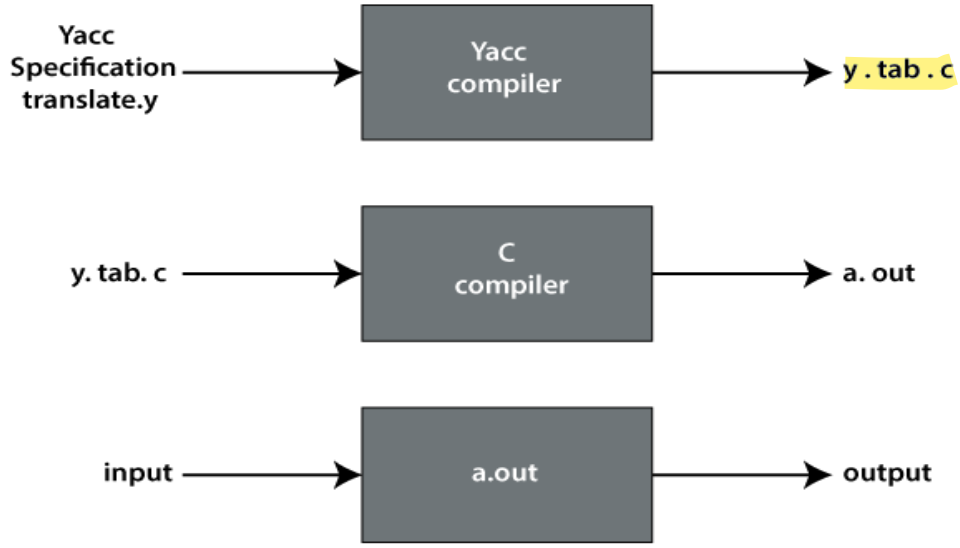
- YACC stands for **Yet Another Compiler Compiler**.
- YACC provides a tool to produce a parser for a given grammar.
- YACC is a program designed to compile a LALR (1) grammar.
- It is used to produce the source code of the syntactic analyzer of the language produced by LALR (1) grammar.
- The input of YACC is the rule or grammar and the output is a C program.

These are some points about YACC:

Input: A CFG- file.y

Output: A parser y.tab.c (yacc)

The basic operational sequence is as follows:



Parts of a YACC Program in Compiler Design

The parts of YACC program are divided into three sections:

```
/* declarations*/
 ....

%%
/*translation  rules */
....
%%

/* auxiliary functions*/
....
```

Declarations: %{

Int a,b;

Const int a=2;

#include<stdio.h>

%}

Translation rules:

Head -> body1/body2

Head: body1{semantic action}

/ body2{ semantic action}

Example of YACC :

E->E+T/T

T->T*F/F

F->(E)/id

The YACC program code for a simple desk calculator is given below:

```
%{
#include <ctype.h>
#include <stdio.h>
int yylex();
%}


%token DIGIT
%left '+' '-'
%left '*' '/'
%left '(' ')'


%%


line :exp ‘\n’ {printf("=%d\n",$$); return 0;};


exp  :exp '+' term {$$ =$1+$3;}
      /term;
Term :term‘*’factor {$$ =$1*$3;}
       /factor;
Factor: ‘(‘ exp ‘)’ {$$ =$2;}
       /DIGIT;

%%


void yylex()
{
  int c;
  c=getchar();
  if(isdigit(c))
    {
     yylval=c-‘0’;
     return DIGIT;
     }
    return c;
}
```