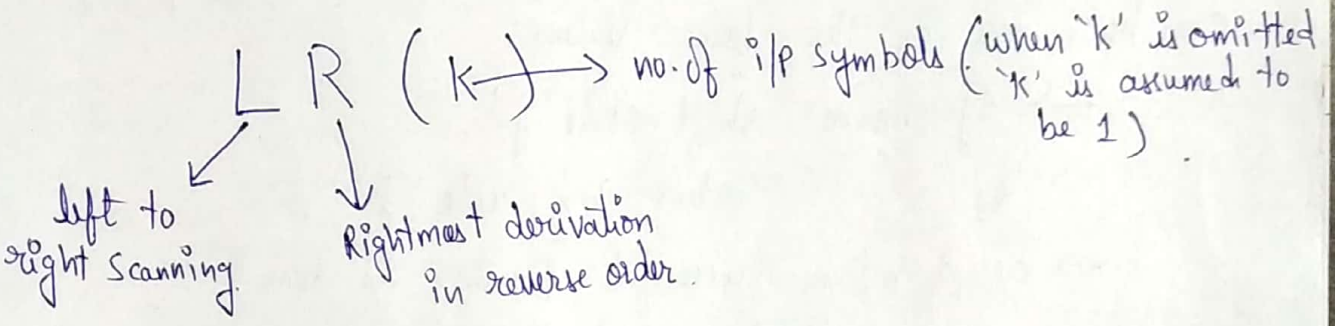
UNIT-III : Syntax Analysis (Part-II)

* Introduction to LR Parsing :

LR Parsing is the most efficient method of bottom up Parsing which can be used to parse the large class of context-free grammars. This method is also called LR(k) Parsing.



→ LR Parsers are widely used for the following reasons :

(1) LR Parsers can be constructed to recognize most of the Programming languages for which context-free grammar can be written.

(2) The class of grammar that can be parsed by LR Parser is a superset of class of grammars that can be parsed using Predictive Parsers.

(3) LR Parser works using non-backtracking shift reduce technique yet it is efficient one.

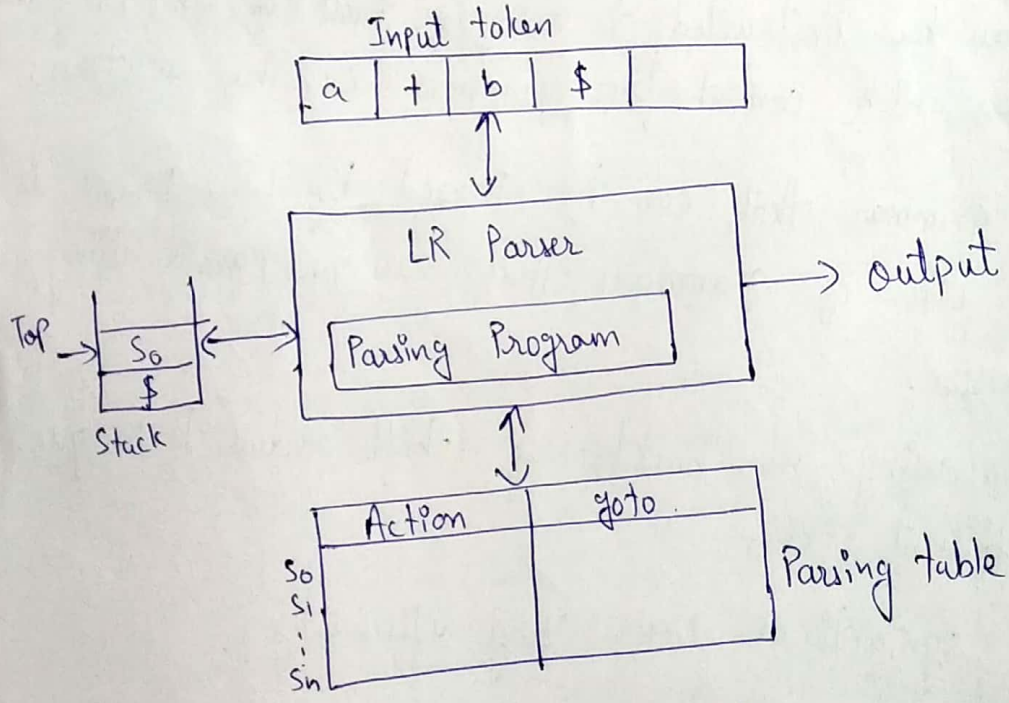
LR Parsers detect syntactical errors very efficiently.

* Structure of LR Parsers.

LR Parsers consists of input buffer for storing the input string, a stack for storing the grammar symbols, output & a Parsing table which has two parts : action & goto. There is one Parsing Program which reads the input symbol one at a time from

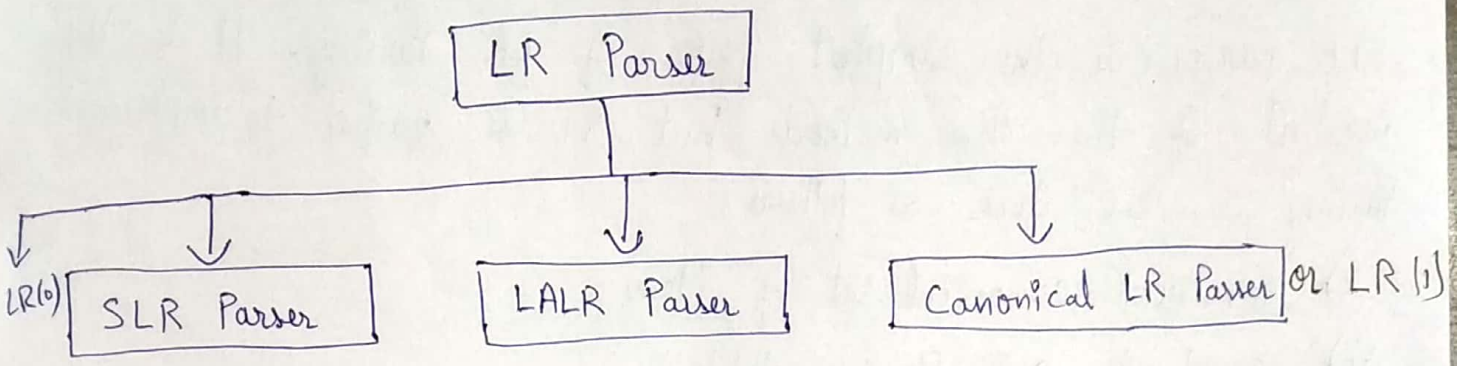
the input buffer. This Parsing Program works as follows:

- It initializes the stack with start symbol & invokes Scanner to get next token.
- It determines `s_j' the state currently on the top of the stack & a_i the current input symbol.
- It consults the Parsing table for the actions $[s_j, a_i]$ which can have one of the four values:
  - → s_j means shift state `j'.
  - → r_j " reduce by rule `j'.
  - → accept means successful Parsing is done
  - → error indicates syntactical error.



Structure Of LR Parser

*Types of LR Parsers :-



SLR Parser	LALR Parser	Canonical LR Parser (CLR)
(1) SLR Parser is the smallest in size.	(1) LALR & SLR have the same size.	(1) LR Parser (or) CLR Parser is largest in size.
(2) It is an easiest method based on FOLLOW function.	(2) This method is applicable to wider class than SLR.	(2) This method is most Powerful than SLR & LALR.
(3) Error detection is not immediate in SLR.	(3) Error detection is not immediate in LALR.	(3) Immediate error detection is done by LR Parser.
(4) It requires less time & space complexity.	(4) The time & space complexity is more in LALR but efficient methods exist for constructing LALR Parsers directly.	(5) The time & space complexity is more for canonical LR Parser.

Power :- SLR(1) ≤ LALR(1) ≤ CLR(1) (or) LR(1).

Size :- (SLR = LALR) < CLR .

* (a) SLR Parser (or) Simple LR.

SLR Parser is the simplest form of LR Parsing. It is the weakest of the three methods but it is easiest to implement. Parsing can be done as follows:

i) Construct canonical set of items
ii) Construct SLR Parsing table
iii) Parsing of input string.

→ A grammar for which SLR Parser can be constructed is called SLR grammar.

(Let $S \rightarrow \epsilon$ be a Production)

* Augmented grammar: If a gramar 'G' is having start symbol 'S' then augmented grammar is a new grammar G' in which S' is a new start symbol such that $S' \rightarrow S$. The Purpose of this grammar is to indicate the acceptance of input.

* Kernel items: It is collection of items $S' \rightarrow .S$ & all the items whose dots are not at the leftmost end of RHS of the rule.

* Non-Kernel items: The collection of all the items in which • are at the left end of RHS of the rule.

* Closure operation: For a CFG 'G', if 'I' is the set of items then the function closure(I) can be constructed using following rules:

- Consider 'I' is a set of canonical items & initially every item 'I' is added to closure(I).
- If rule $A \rightarrow \alpha . B\beta$ is a rule in closure(I) & there is another rule for 'B' such as $B \rightarrow \gamma$ then

closure (I) : $A \rightarrow \alpha \cdot B\beta$
$B \rightarrow \cdot\gamma$

This rule has to be applied until no more items can be added to closure (I).

* goto operation :-
If there is a production $A \rightarrow \alpha \cdot B\beta$ then goto $(A \rightarrow \alpha \cdot B\beta, B)$ = $A \rightarrow \alpha B \cdot \beta$. That means simply shifting of • one position ahead over the grammar symbol (may be terminal or non-terminal). The rule $A \rightarrow \alpha \cdot B\beta$ is in 'I' then the same goto function can be written as goto (I, B).

(i) Construction of Canonical set of items :-

- For the grammar 'G' initially add $S' \rightarrow \cdot S$ in the set of item 'C'.
- For each set of items I_i in 'C' & for each grammar symbol X (may be terminal or non-terminal) add closure (I_i, X)

This process should be repeated by applying goto (I_i, X) for each 'X' in I_i such that goto (I_i, X) is not empty & not in 'C'. The set of items has to constructed until no more set of items can be added to 'C'.

(ii) Construction of SLR Parsing table

By considering basic parsing actions such as shift, reduce, accept & error we will fill up the action table. The goto table can be filled up using goto function.

Input : An Augmented grammar G'.

output : SLR Parsing table.

Algorithm :

- Initially construct set of items $C = \{I_0, I_1, I_2, \ldots I_n\}$ where 'C' is a collection of set of LR(0) items for the input grammar G'.
- The parsing actions are based on each Item I_i. The actions are as given below :
  - → If $A \rightarrow \alpha . a\beta$ is in I_i & goto $(I_i, a) = I_j$ then set action $[i, a]$ as "shift j". Note that 'a' must be a terminal symbol.
  - → If there is a rule $A \rightarrow \alpha .$ in I_i then set action $[i, a]$ to "reduce $A \rightarrow \alpha$" for all symbols 'a', where $a \in$ FOLLOW (A). Note that 'A' must not be an augmented grammar s'.
  - → If $S' \rightarrow S$ is in I_i then the entry in the action table action $[i, \$]$ = "accept".
- The goto part of the SLR table can be filled as : The goto transitions for state 'i' is considered for non-terminals only. If goto $(I_i, A) = I_j$ then goto $[I_i, A] = j$.
- All the entries not defined by rule 2 & 3 are considered to be "error".

iii) Parsing the Input using Parsing Table:

Input: The input string 'w' that is to be parsed & Parsing table.

Output: Parse 'w' if $w \in L(G)$ using bottom up. If $w \notin L(G)$ then report syntactical error.

Algorithm:

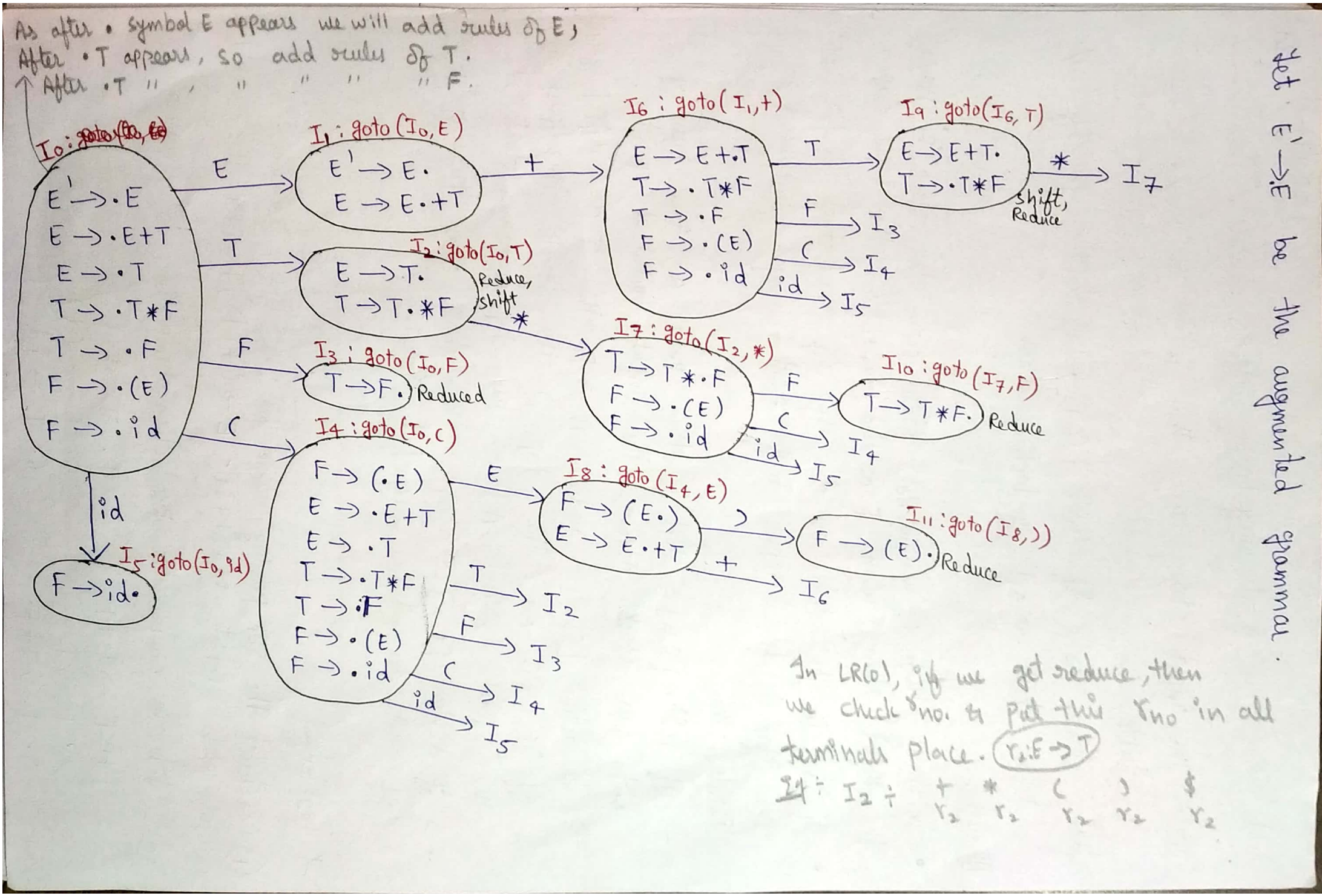
- Initially Push '0' as initial state onto the stack & place the input string with $ as end marker on the input tape.
- If 's' is the state on the top of the stack & 'a' is the symbol from input buffer pointed by a lookahead Pointer then
  - → If action [s,a] = shift 'j' then push 'a', then push 'j' onto the stack. Advance the input lookahead pointer.
  - → If action [s,a] = reduce $A \rightarrow \beta$ then pop $2*|\beta|$ symbols If 'i' is on the top of the stack then push 'A', then push goto [i,A] on the top of the stack.
  - → If action [s,a] = accept then halt the Parsing Process. If indicates the successful Parsing.

* Problems:

(1) Construct the SLR parsing table for the given grammar. Also Parse the input id*id+id. $E \rightarrow E+T \mid T$, $T \rightarrow T*F \mid F$, $F \rightarrow (E) \mid id$.

Sol: Let us number the Production rules in the grammar.

1. $E \rightarrow E+T$
2. $E \rightarrow T$
3. $T \rightarrow T*F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$



- First(E) = First(T) = First(F) = {(, id} (here first & Follow do not do left recursion, left factoring)
- ~~First(E')~~ = ~~{+, ∈}~~
- ~~First(T')~~ = ~~{*, ∈}~~
- Follow(E) = {+, $,)}
- ~~Follow(E')~~ = ~~{$,)}~~
- Follow(T) = {+, *, $,)}
- ~~Follow(T')~~ = ~~{+, $,)}~~
- Follow(F) = {*, +, $,)}

* SLR Parsing Table: (Not LR(0) but it is SLR(1))

states	Action						goto		
	id	+	*	(	)	$	E	T	F
0	S_5			S_4			1	2	3
1		S_6				Accept			
2	LR(0) SLR(1) → r_2	r_2	S_7	r_2	r_2	r_2			
3	LR(0) r_4 SLR(1) → r_4	r_4	r_4	r_4	r_4	r_4			
4	S_5			S_4			8	2	3
5	LR(0) r_6 SLR(1) → r_6	r_6	r_6	r_6	r_6	r_6			
6	S_5			S_4				9	3
7	S_5			S_4					10
8		S_6			S_{11}				
9	LR(0): r_1 SLR(1) → r_1	r_1	S_7	r_1	r_1	r_1			
10	LR(0): r_3 SLR(1): → r_3	r_3	r_3	r_3	r_3	r_3			
11	LR(0): r_5 SLR(1) → r_5	r_5	r_5	r_5	r_5	r_5			

r_1 : E → E+T
r_2 : E → T
r_3 : T → T*F
r_4 : T → F
r_5 : F → (E)
r_6 : F → id

* → For Item 2, E → T. is reduced. check the list. it is numbered r_2. Now Follow(E) = ($,), +) in $,), + columns write r_2.

* → If it is shift operation just write S_{no}. If Item 0 on id it is going to I_5. So write "S_5".

* Parsing input "id * id + id".

Stack	Input	Action	output
$0	id * id + id $	shift s_5	whenever we do Reduce we need to pop out 2 symbols.
$0 id5	* id + id $	Reduce by F → id	
$0 F_3	* id + id $	Reduce by T → F	
$0 T_2	* id + id $	shift s_7	→ when you are shifting no need to pop.
$0 T_2 * 7	id + id $	shift s_5	
$0 T_2 * 7 id5	+ id $	Reduce by F → id	
$0 T_2 * 7 F10	+ id $	Reduce by T → T * F	∴ since there are no conflicts in SLR table. The given grammar is SLR(1).
$0 T_2	+ id $	Reduce by E → T	
$0 E1	+ id $	shift s_6	
$0 E1 + 6	id $	shift s_5	
$0 E1 + 6 id5	$	Reduce by F → id	
$0 E1 + 6 F3	$	Reduce by T → F	
$0 E1 + 6 T9	$	Reduce by E → E + T	
$0 E1	$	accept	

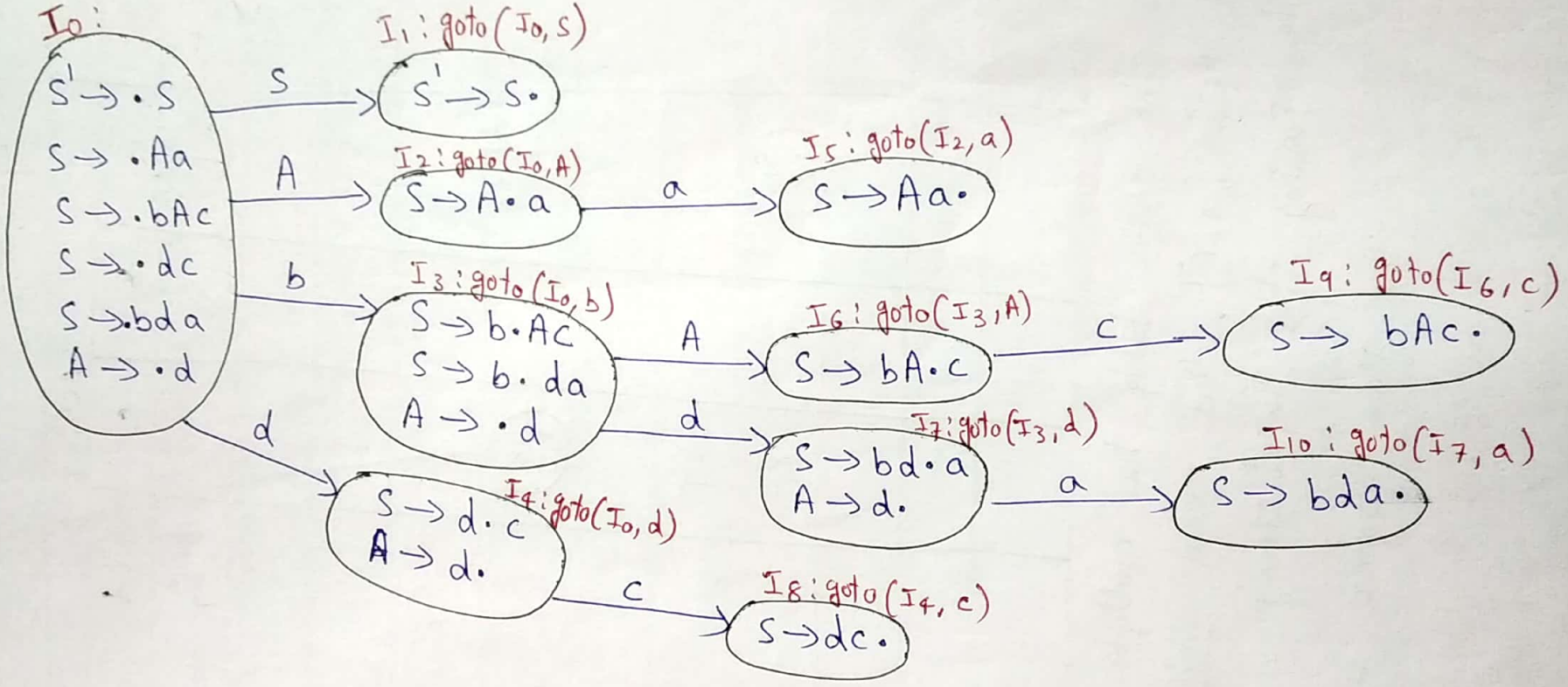
(2) Show that the following grammar is not SLR(1).

S → Aa | bAc | dc | bda

A → d

Sol: Let us number the Production rules in the grammar.

1. S → Aa
2. S → bAc
3. S → dc
4. S → bda
5. A → d



- Follow (S) = {$}
- Follow (A) = {a, c}
- First (S) = {b, d}
- First (A) = {d}

* SLR(1) Parsing table :

Not LR(0)

	Action					goto	
State	a	b	c	d	$	S	A
0		S_3		S_4		1	2
1					Accept		
2	S_5						
3				S_7			6
4	LR(0) r_5 SLR(1) r_5	r_5	S_8 r_5	r_5			
5					r_1		
6			S_9				
7	S_{10} r_5		r_5				
8					r_3		
9					r_2		
10					r_4		

Since the above table has multiple entries in Action [7,a] and Action [4,c], this means shift/reduce conflict will occur while Parsing the input string.

∴ The given grammar is not SLR(1).

LR(0) : in reduce write r_{no} in all terminals of that Item.
SLR(1) : ↓ we write r_{no} in follow " .

↓ only difference.

* More Powerful Parsers :

(a) Canonical LR : (or) LR(K) Parsers

The canonical set of items is the parsing technique in which a lookahead symbol is generated while constructing set of items. Hence the collection of set of items is referred as LR(1). The value 1 in the bracket indicates that there is one lookahead symbol in the set of items.

→ Steps to construct canonical LR :

- Construction of canonical set of items along with the lookahead.
- Building canonical LR Parsing table.
- Parsing the input string using canonical LR Parsing table.

* Construction of canonical set of items along with lookahead :

(1) For the grammar 'G' initially add $S' \rightarrow \cdot S$ in the set of item 'C'.

(2) For each set of items I_i in 'C' & for each grammar Symbol X (may be terminal or non-terminal) add closure (I_i, X). This process should be repeated by applying goto $((I_i, X))$ for each X in I_i such that goto (I_i, X) is not empty & not in 'C'. The set of items has to be constructed until no more set of items can be added to 'C'.

(3) The closure function can be computed as follows :
For each item $A \rightarrow \alpha \cdot X\beta a$ and rule $X \rightarrow \gamma$ & $b \in$ FIRST (βa) such that $X \rightarrow \cdot\gamma$ & 'b' is not in I

then add $X \to .\gamma, b$ to I.

④ Similarly the goto function can be computed as: For each item $[A \to \alpha . X\beta, a]$ is in I & rule $[A \to \alpha X . \beta, a]$ is not in I ~~then add~~ $X \to .\gamma, b$ goto items then add $[A \to \alpha X . \beta, a]$ to goto items.

This process is repeated until no more set of items can be added to the collection 'C'.

* Construction of canonical LR Parsing table:

① Initially construct set of items $C = \{I_0, I_1, I_2, \ldots I_n\}$ where 'C' is a collection of set of LR(1) items for the input grammar G'.

② The parsing actions are based on each item I_i. The actions are as given below:

- If $[A \to \alpha . a\beta, b]$ is in I_i and goto $(I_i, a) = I_j$ then Create an entry in the action table action $[I_i, a]$ = shift j.
- If there is a production $[A \to \alpha ., a]$ in I_i then in the action table action $[I_i, a]$ = reduce by $A \to \alpha$. Here 'A' should not be S'.
- If there is a production $S' \to S., \$$ in I_i then action $[i, \$]$ = accept.

③ The goto Part of the LR table can be filled as: The goto transitions for state 'i' is considered for non-terminals only. If goto $(I_i, A) = I_j$ then goto $[I_i, A] = j$.

④ ~~All the entries not defined by rule 2 and 3 are considered to be "error"~~ All entries not defined by rule 2 & 3 are considered to be "error"

* Problems :

① Construct LR(1) or CLR set of items for the below grammar

$S \rightarrow CC$

$C \rightarrow aC \mid d$

Sol : Add $S' \rightarrow .S, \$$ as the first rule in I_0. The grammar is :

$S' \rightarrow .S, \$$

$S \rightarrow CC$

$C \rightarrow aC \mid d$

Add dot to every Production : $(A \rightarrow \alpha . X\beta, a)$; $A \in S'$, $X \in S$, $a \in \$$

I_0

$S' \rightarrow .S, \$$

$S \rightarrow .CC, \$$

$C \rightarrow .aC, a/d$

$C \rightarrow .d, a/d$

$X \rightarrow \gamma, b$ then add $X \rightarrow .\gamma, b$

$b \in First(\beta a)$

$b \in First(\epsilon \$)$

$b \in First(\$)$

$b = \{\$\}$

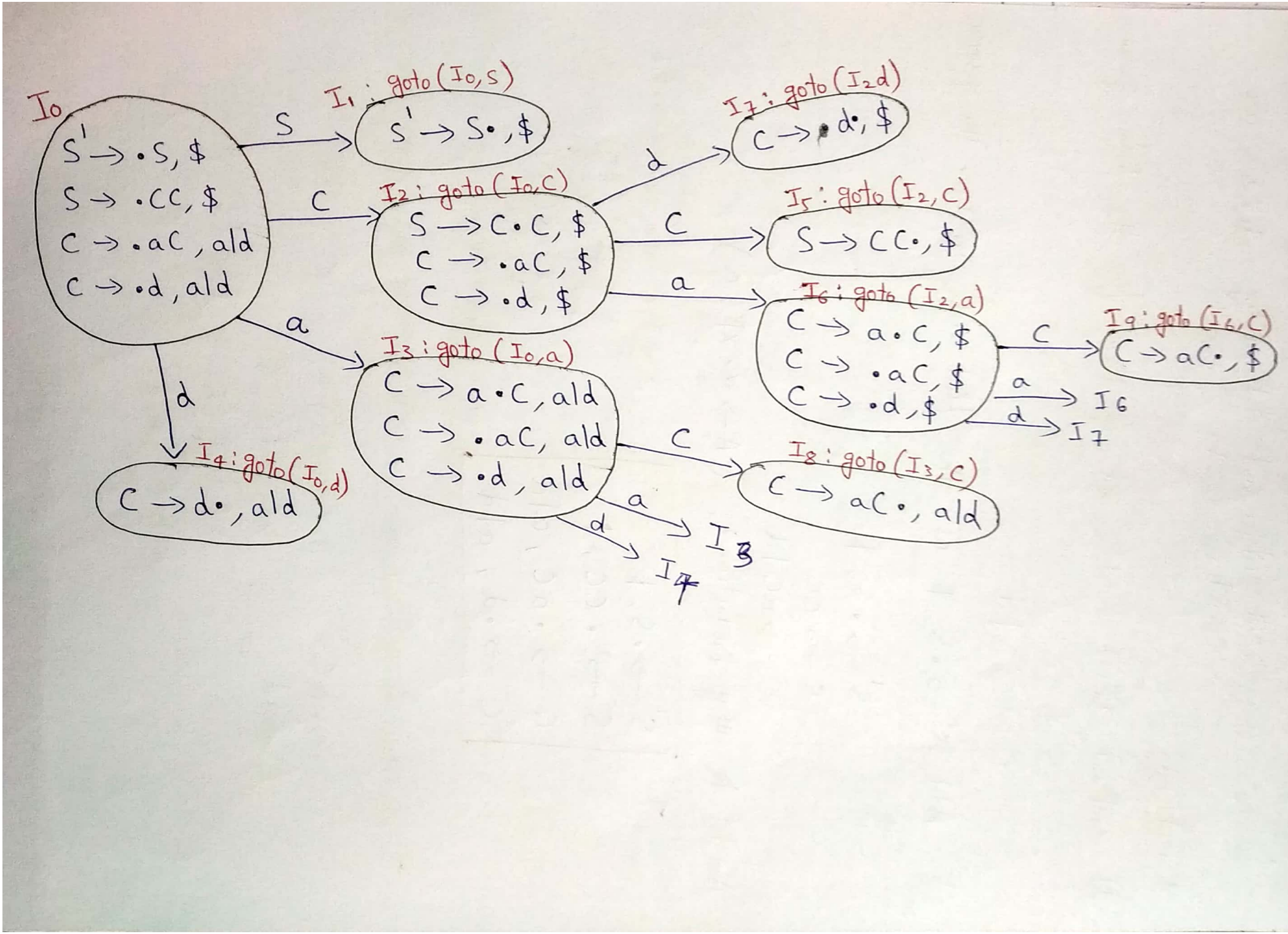
→ If there is a Production $X \rightarrow \gamma, b$ then add $X \rightarrow .\gamma, b$

$C \rightarrow .aC$ $\quad b \in First(\beta a)$

$C \rightarrow .d$ $\quad b \in First(C\$)$

$b \in First(C)$ as $First(C) = \{a, d\}$

$b = \{a, d\}$



* LR(1) Parsing table :-

	action			goto	
	a	d	$	S	C
0	S_3	S_4		1	2
1			Accept		
2	S_6	S_7			5
3	S_3	S_4			8
4	r_3	r_3			
5			r_1		
6	S_6	S_7			
7			r_3		
8	r_2	r_2			
9			r_2		

r_1 : $S \rightarrow CC$

r_2 : $C \rightarrow aC$

r_3 : $C \rightarrow d$

I_0 :- on S it is shifted to I_1
C " " " " I_2
a " " " " I_3
d " " " " I_4

} write Sno in appropriate rows.

I_4 :- reduced check numbering : " r_3 ".

$C \rightarrow d\cdot$, (a/d) => write in a/d " r_3 ".

* Parsing the input using LR(1) Parsing table

Let us Parse for the i/p string "aadd" by using above Parsing table.

stack	Input buffer	action table	goto table	Parsing action
$0	aadd$	action [0,a] = S_3		
$0a3	add $	action [3,a] = S_3		shift
$0a3a3	dd$	action [3,d] = S_4		shift
$0a3a3d4	d$	action [4,d] = r_3	[3,C] = 8	Reduce by C→d
$0a3a3C8	d$	action [8,d] = r_2	[3,C] = 8	Reduce by C→aC
$0a3C8	d$	action [8,d] = r_2	[0,C] = 2	Reduce by C→aC
$0C2	d$	action [2,d] = S7		shift
$0C2d7	$	action [7,$] = r_3	[2,C] = 5	Reduce by C→d
$0C2C5	$	action [5,$] = r_1	[0,S] = 1	Reduce by S→CC
$0S1	$	accept		

Thus the given input string is succesfully Parsed using LR Parser or canonical LR Parser.

(2) show that the following grammar is LR(1).

S → Aa | bAc | Bc | bBa

A → d

B → d.

Sol: We will number out the Production rules in given grammar.

1. S → Aa
2. S → bAc
3. S → Bc
4. S → bBa
5. A → d
6. B → d

Let $S' \rightarrow .S, \$$.

* Constructing Canonical set of LR(1) items :-

* Parsing table :

State	Action					goto		
	a	b	c	d	$	S	A	B
0		S_3		S_5		1	2	4
1					Accept			
2	S_6						7	8
3				S_9				
4			S_{10}					
5	r_5		r_6					
6					r_1			
7			S_{11}					
8	S_{12}							
9	r_6		r_5					
10					r_3			
11					r_2			
12					r_4			

* Parsing the string "bda" using above Parsing table of LR(1) :

stack	Input Buffer	Action
$0	bda$	shift 3
$0b3	da$	shift 9
$0b3d9	a$	Reduce by B → d
$0b3B8	a$	shift 12

Stack	Input Buffer	Action
$ 0b3B8a12	$	Reduce by S → bBa
$ 0S1	$	Accept

The above string 'bda' is accepted. Since in the Parsing table we do not have multiple entries in the same cell, the above grammar is LR(1) or CLR(1).

* LALR :

The algorithm for construction of LALR Parsing table is as follows:

step 1 : Construct the LR(1) set of items.

step 2 : Merge the two states I_i and I_j if the first component (i.e the production rules with dots) are matching & create a new state replacing one of the older state such as $I_{ij} = I_i \cup I_j$.

step 3 : The parsing actions are based on each item I_i. The actions are as given below:

a) If $[A \to \alpha . a\beta, b]$ is in I_i & goto $(I_i, a) = I_j$ then create an entry in the action table action $[I, a]$ = shift 'j'.

b) If there is a production $[A \to \alpha \cdot, a]$ in I_i then in the action table action $[I_i, a]$ = reduce by $A \to \alpha$ Here 'A' should not be S'.

c) If there is a production $S' \to S \cdot, \$$ in I_i then action$[i, \$]$ = accept.

step4 : The goto part of the LR table can be filled as: The goto transitions for state 'i' is considered for non-terminals only. If goto $(I_i, A) = I_j$ then goto $[I_i, A] = j$.

step5 : If the parsing action conflict then the algorithm fails to produce LALR Parser & grammar is not LALR(1). All the entries not defined by rule 3 & 4 are considered to be "error".

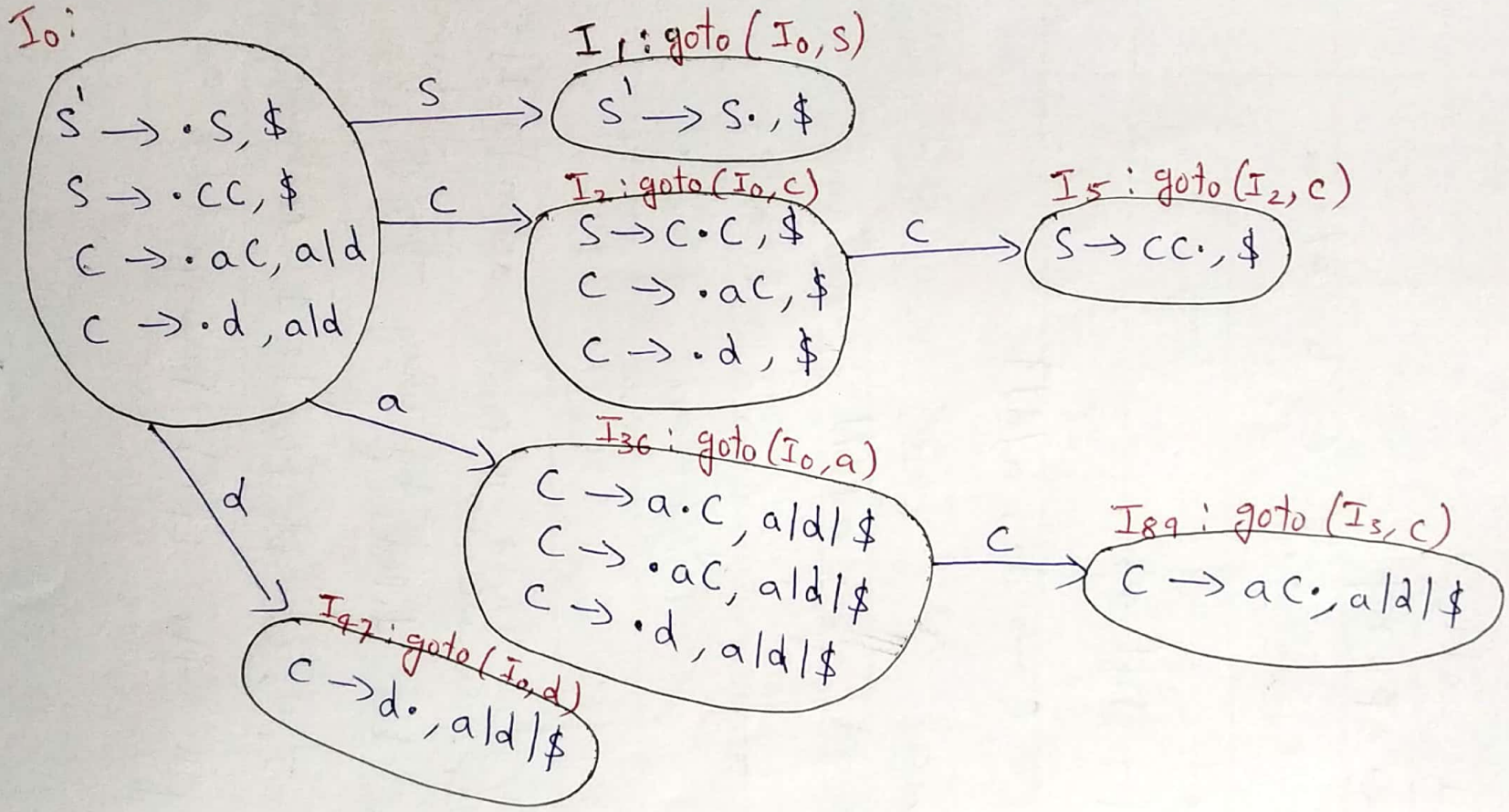
* Problems :

(1) Construct Parsing table for LALR(1) Parser for the below given grammar.

$S \to CC$
$C \to aC$
$C \to d$.

sol : First the set LR(1) items can be constructed as follows with merged states.

$r_1 : S \to CC$
$r_2 : C \to aC$
$r_3 : C \to d$.



Now consider state I_0 there is a match with the rule $[A \to \alpha . a\beta, b]$ and goto $(I_i, a) = I_j$.

$C \to .aC, a/d/\$$ and if the goto is applied on 'a' then we get the state I_{36}. Hence,

in I_0:

$$C \to .d \;, a/d$$

$$\boxed{A \to \alpha . a\beta, b}$$

where $A = C$, $\alpha = \epsilon$, $a = d$, $\beta = \epsilon$, $b = a/d$

goto $(I_0, d) = I_{47}$.

hence action $[0, d]$ = shift 47.

→ For state I_{47} : $C \to d.$, $a/d/\$$

$$A \to \alpha ., a$$

where $A = C$, $\alpha = d$, $a = a/d/\$$.

action $[47, a]$ = reduce by $C \to d$ i.e rule 3

action $[47, d]$ = " " " " "

action $[47, \$]$ = " " " " "

→ $S' \to S.$, $\$$ in I_1. so we will create action $[1, \$]$ = accept.

The goto table can be filled by using the goto functions. In this instance goto $(I_0, S) = I_1$, so goto $[0, S] = 1$. continuing in this manner we can fill the LR(1) Parsing table as follows:

States	Action			goto	
	a	d	$	S	C
0	S_{36}	S_{47}		1	2
1			Accept		
2	S_{36}	S_{47}			5
36	S_{36}	S_{47}			89
47	r_3	r_3	r_3		
5			r_1		
89	r_2	r_2	r_2		

* Parsing string "adad" using LALR Parser :

Stack	Input buffer	Action table	goto table	Parsing action
$0	aadd $	action [0,a] = S36		
$0a36	add $	action [36,a] = S36		shift
$0a36a36	dd $	action [36,d] = S47		shift
$0a36a36d47	d$	action [47,d] = r_{36}	[36,C] = 89	Reduce by C→d
$0a36a36C89	d $	action [89,d] = r_2	[36,C] = 89	Reduce by C→aC
$0a36C89	d$	action [89,d] = r_2	[0,C] = 2	Reduce by C→aC
$0C2	d $	action [2,d] = S47		shift
$0C2d47	$	action [47,$] = r_{36}	[2,C] = 5	Reduce by C→d
$0C2C5	$	action [5,$] = r_1	[0,S] = 1	Reduce by S→CC
$0S1	$	accept		

(2) Show that the following grammar is not LALR(1).

Sol:-

$S \to Aa \mid bAc \mid Bc \mid bBa$

$A \to d$

$B \to d$

Sol:- Let us number the Production rules of the grammar:

$r_1 : S \to Aa$

$r_2 : S \to bAc$

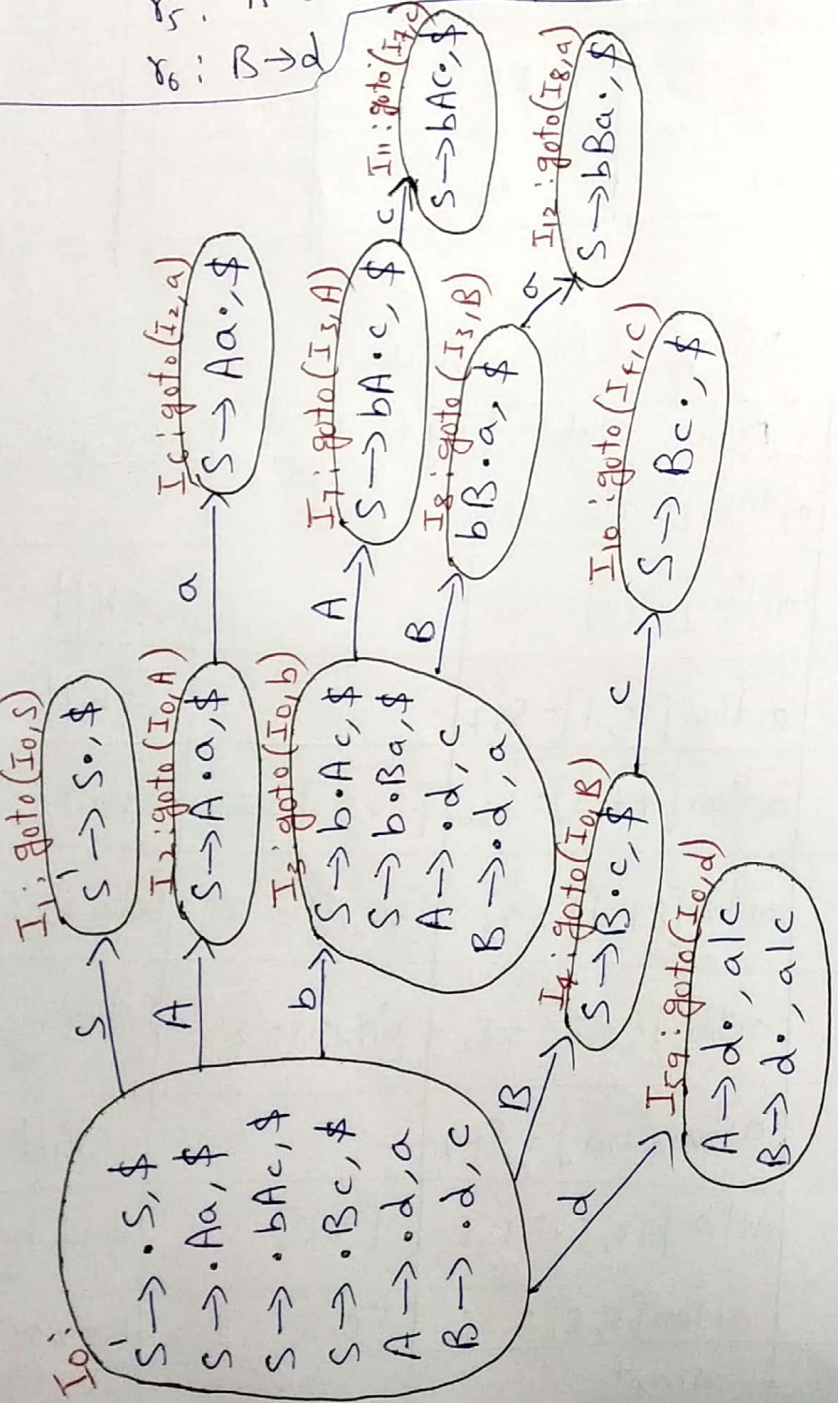
$r_3 : S \to Bc$

$r_4 : S \to bBa$

$r_5 : A \to d$

$r_6 : B \to d$

LALR(1) Set of items are as follows:-



states	Action					goto		
	a	b	c	d	$	S	A	B
0		S_3		S_5		1	2	4
1					Accept			
2	S_6							
3				S_9			7	8
4			S_{10}					
59	γ_5 γ_6		γ_5 γ_6					
6					γ_1			
7			S_{11}					
8	S_{12}							
10					γ_3			
11					γ_2			
12					γ_4			

The Parsing table shows multiple entries in Action [59, a] & Action [59, c]. This is called reduce/reduce conflict. Because of this conflict we cannot parse input.

∴ The given grammar is not LALR(1).

* Handling Ambiguous Grammar:

If all the grammar is ambiguous then it creates the conflicts & we cannot parse the input string with such ambiguous grammar. But for some languages in which arithmetic

expressions are given ambiguous grammar are most compact & Provide more natural specification as compared to equivalent unambiguous grammar.

→ while using ambiguous grammar for Parsing the input string we will use all the disambiguating rules so that each time only one Parse tree will be generated for that specific input. Thus ambiguous grammar can be used in controlled manner for Parsing the input.

* Using Precedence & Associativity to resolve Parsing Action Conflicts

Consider an ambiguous grammar:

$E \rightarrow E+E$

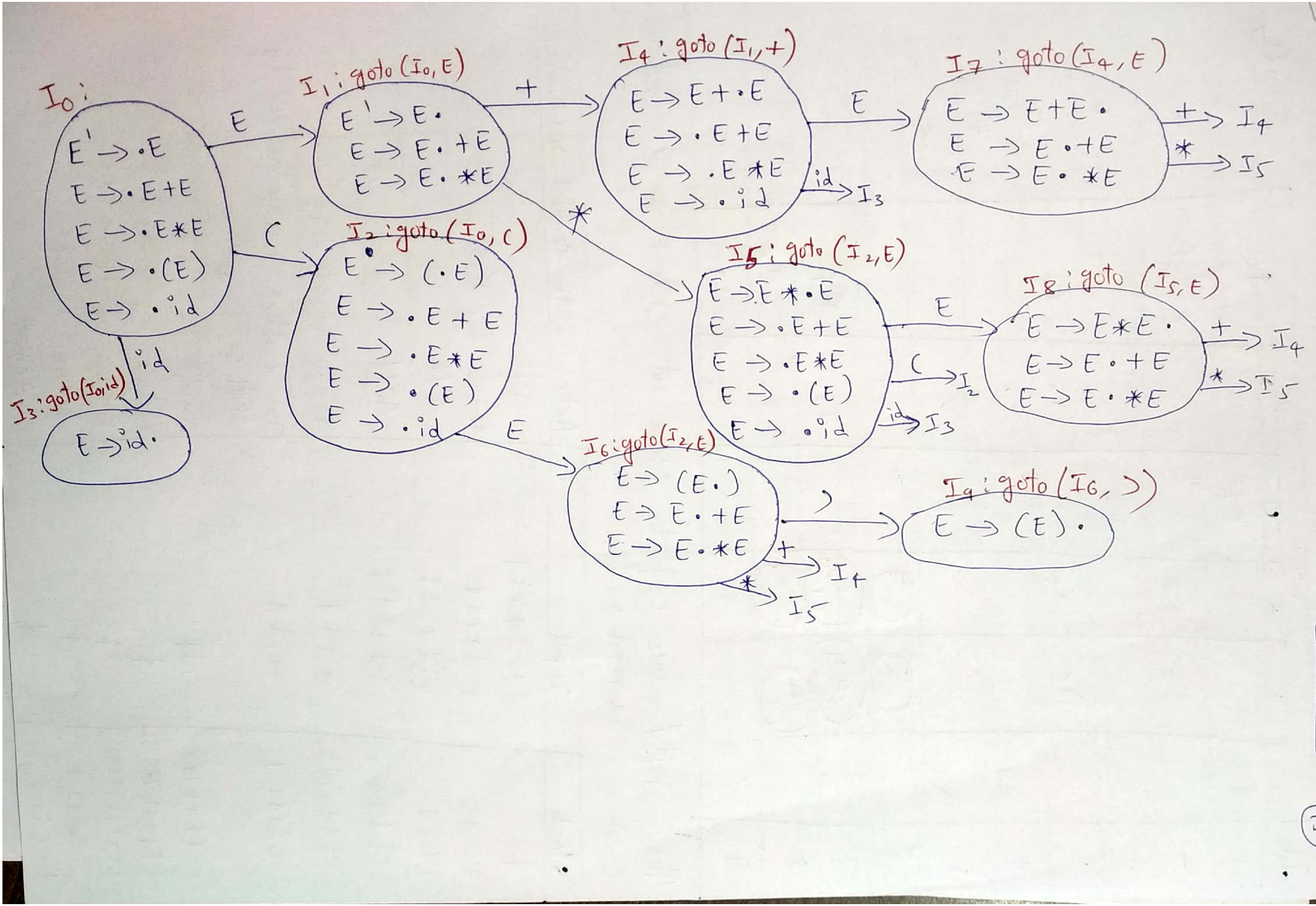
$E \rightarrow E*E$

$E \rightarrow (E)$

$E \rightarrow id$

Now we will build the set of LR(0) items for this grammar.

- Follow(E) = $\{+, *,), \$\}$

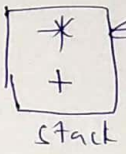


state	action						(goto) E
	id	+	*	(	)	$	
0	S_3			S_2			1
1		S_4	S_5			Accept	
2	S_3			S_2			6
3		r_4	r_4		r_4	r_4	
4	S_3			S_2			7
5	S_3			S_2			8
6		S_4	S_5		S_9		
7		S_4 or r_1	S_5 or r_1		r_1	r_1	
8		S_4 or r_2	S_5 or r_2		r_2	r_2	
9			r_3		r_3	r_3	

The shift/reduce conflicts occur at state 7 & 8. Let us consider id+id*id.

Stack	Input	Action with conflict resolution
$0	id+id*id$	shift
$0id3	+id*id$	Reduce by E→id
$0E1	+id*id$	shift
$0E1+4	id*id$	shift
$0E1+4id3	*id$	Reduce by E→id
$0E1+4E7	*id$	The conflict can be resolved by shift 5
$0E1+4E7*5	id$	shift
$0E1+4E7*5id3	$	Reduce by E→id
$0E1+4E7*5E8	$	Reduce by E→E*E
$0E1+4E7	$	Reduce by E→E+E
$0E1	$	Accept

As * has precedence over '+' we have to perform multiplication operation first. And for that it is necessary to push * on the top of the stack. The stack position will be:



By this we can perform E*E first & then E+E. The Parsing conflict can be resolved by assigning shift operation. Hence action[7,*] = S_5.

state	Action						goto
	id	+	*	(	)	$	E
0	S_3			S_2			1
1		S_4	S_5			Accept	
2	S_3			S_2			
3		r_4	r_4		r_4	r_4	
4	S_3			S_2			
5	S_3			S_2			
6		S_4	S_5		S_9		
7		r_1	S_5		r_1	r_1	
8		r_2	r_2		r_2	r_2	
9			r_3		r_3	r_3	

* Using Dangling Else Ambiguity :

Consider the grammar :

$S \rightarrow iSeS \mid iS \mid a$

where ;seS = if expression then statement else statement

;S = if " " "

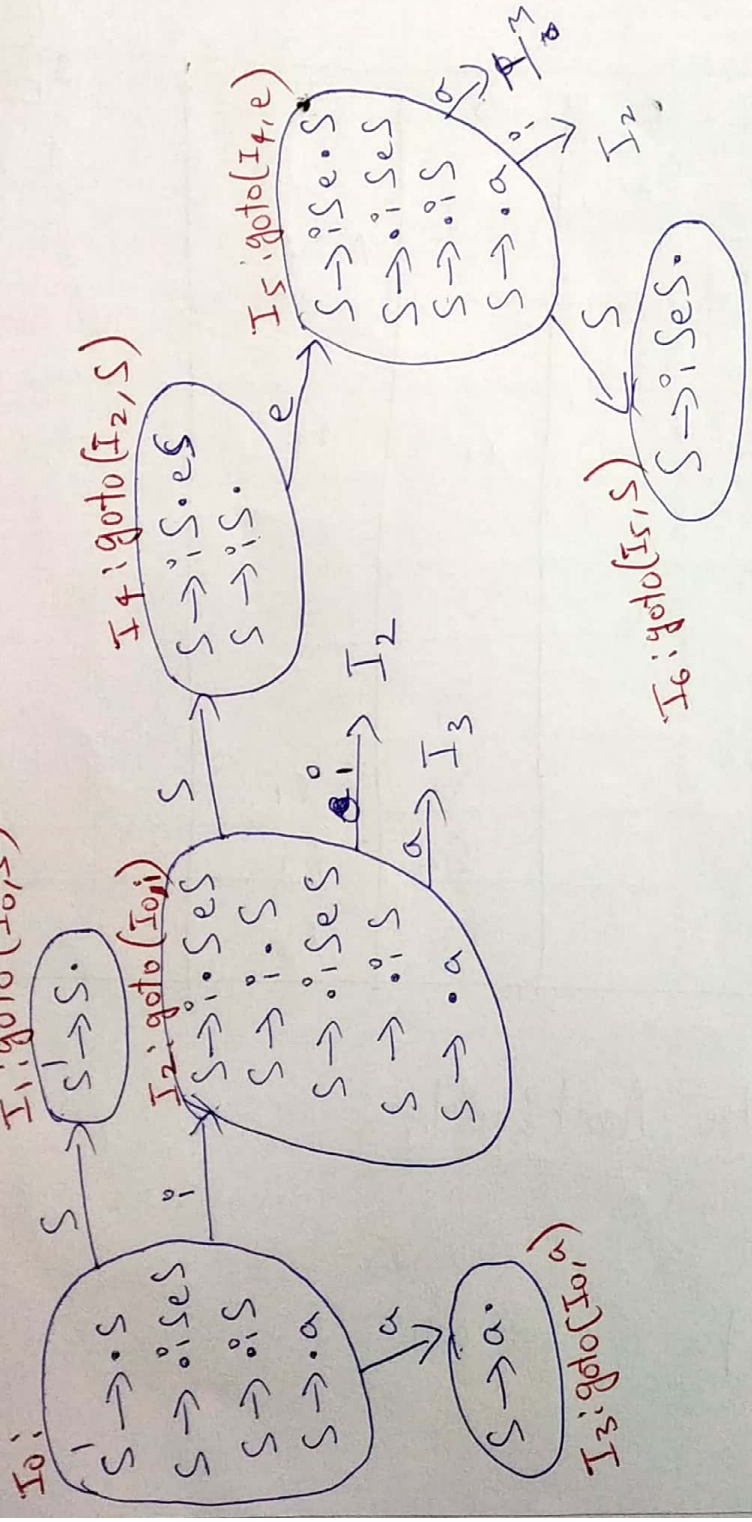
a = all other productions

Let the grammar be:

$S' \rightarrow S$

$S \rightarrow ;seS \mid ;S \mid a$

Let us construct LR(0) set of items as:



First(S) = {;, a}

Follow(S) = {e, $}

State	Action. i	e	a	$	goto S
0	S_2		S_3		1
1				Accept	
2	S_2		S_3		4
3		γ_3		γ_3	
4		S_5 or γ_2		γ_2	
5	S_2		S_3		6
6		γ_1		γ_2	

Action [5, e) has shift/reduce conflict.

Consider the input "iiaea$" for Procesing.

Stack	Input	Action with conflict resolution
$0	iiaea$	shift
$0i2	iaea$	shift
$0i2i2	aea$	shift
$0i2i2a3	ea$	Reduce S → a
$0i2i2S4	ea$	From the conflict we have chosen Reduce S → iS
$0i2S4	ea$	Reduce S → iS
$0S1	a$	Error!!

That means the choice of γ_2 in action [4, e] is not valid. Hence we will try it by choosing the shift action.

Stack	Input	Action with conflict resolution
$0	iiaea$	shift
$0i2	iaea$	shift
$0i2i2	aea$	shift
$0i2i2a3	ea$	Reduce S→a
$0i2i2S4	ea$	From the conflict we have chosen Reduce S→iS
$0i2i2S4e5	a$	shift
$0i2i2S4e5a3	$	Reduce S→a
$0i2i2S4e5S6	$	Reduce S→iSeS
$0i2S4	$	Reduce S→iS
$0S1	$	Accept

Logically also we should choose the shift operation as by shifting the else we can associate it with previous "if expression then statement". Therefore shift / reduce conflict is resolved in favour of shift.

→ After resolving the conflict we get Parsing table for dangling else problem as:

state	Action				goto
	i	e	a	$	S
0	S2		S3		1
1				Accept	
2	S2		S3		4
3		r3		r3	
4		S5		r2	
5	S2		S3		6
6		r1		r2	

* Error Recovery in LR Parser :-

The LR parser is a table driven Parsing method in which the blank entries are treated as errors. when we compile any Program we first get the syntactical errors. These errors are usually denoted by user friendly error messages.

*Example :- $E \to E+E$
$E \to E*E$
$E \to (E)$
$E \to id$.

(we have already constructed LR(0) items for this grammar in handling Ambiguous grammar topic).

state	Action						goto
	id	+	*	(	)	$	E
0	S_3			S_2			1
1		S_4	S_5			Accept	
2	S_3			S_2			6
3		r_4	r_4		r_4	r_4	
4	S_3			S_2			7
5	S_3			S_2			8
6		S_4	S_5		S_9		
7		r_1	S_5		r_1	r_1	
8		r_2	r_2		r_2	r_2	
9			r_3		r_3	r_3	

Following are the rules to fill the table:

(1) If there are entries for r_j in some particular state then fill up all the blank entries r_j only.

(2) If there are shift entries only in some particular state then do not replace blank entries. Keep them as it is.

During the error detection & recovery process we have replaced some blank (error) entries by particular reduction rules. This change means we are postponing the error detection until one or more reductions are done & error will be introduced before any shift move take place.

Consider the set of items generated for obtaining the error messages.

$I_0 : E' \rightarrow \cdot E$

It means that there is no symbol before the dot. And being the initial state the stack is empty. In such a case if + (or) * (or) $ comes in the input string then we say that operand is missing for these operator.

Stack	Input	Error could be
$	+	Missing operand
$	*	missing operand
$	$	Missing operand
$	)	Unbalanced right Parenthesis

I_1 : goto (I_0, E)
$E' \rightarrow E \cdot$

If we ultimately reduce $E \rightarrow id$ then id = missing operator, id (= missing operator, id) = unbalanced right Parenthesis.

Stack	Input	Error could be
$...id	id	Missing operator
$...id	(	" "
$...id	)	unbalanced right Parenthesis

I_6 : goto (I_2, E)
$E \rightarrow (E \cdot)$

If we eventually reduce $E \rightarrow id$ then the rule becomes $E \rightarrow (id \cdot)$. That means after id we expect ')' & if $ comes then the error will be missing right Parenthesis.

Stack	Input	Error could be
$.... (id	id	Missing operator
$.... (id	(	" "
$... (id	$	Missing right Parenthesis

From all these situations, some error messages are:

(1) E1 : These errors are in states I_0, I_2, I_4 & I_5. This indicates that the operand should appear before operator. Hence the error message will be "missing operand".

(2) E2 : This error is in ')' column & from states I_0, I_1, I_2, I_4 & I_5 indicating unbalancing in right parenthesis. Hence error message will be "unbalanced right Parenthesis".

(3) E3 : The operator is expected in this case of error as it is from state I_1 or I_6. The error message will be "missing operator".

• E_4 : This error occurs at state 6 in the $ column. The state 6 expects ')' Parenthesis at the end of expression. Hence the error message will be "missing right Parenthesis"

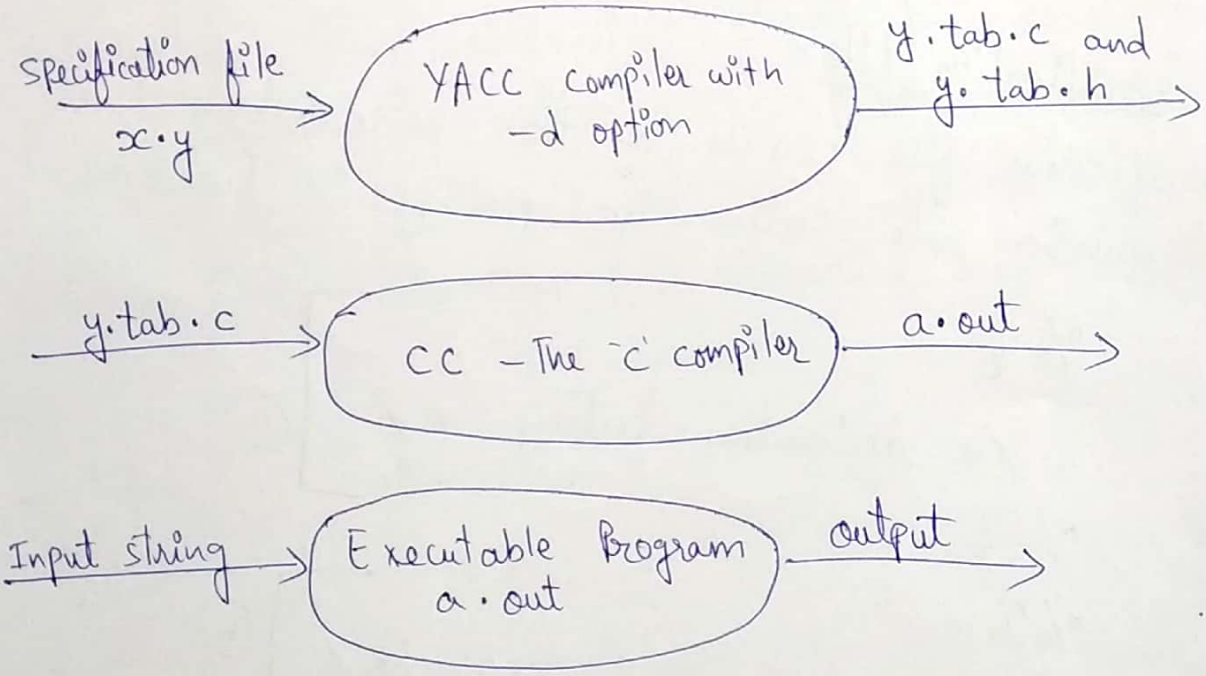
→ Modified table with included error messages :-

state	Action						goto
	id	+	*	(	)	$	E
0	S_3	E_1	E_1	S_2	E_2	E_1	1
1	E_3	S_4	S_5	E_3	E_2	Accept	
2	S_3	E_1	E_1	S_2	E_2	E_1	6
3	r_4	r_4	r_4	r_4	r_4	r_4	
4	S_3	E_1	E_1	S_2	E_2	E_1	7
5	S_3	E_1	E_1	S_2	E_2	E_1	8
6	E_3	S_4	S_5	E_3	S_9	E_4	
7	r_1	r_1	S_5	r_1	r_1	r_1	
8	r_2	r_2	r_2	r_2	r_2	r_2	
9	r_3	r_3	r_3	r_3	r_3	r_3	

While Parsing the input, LR Parser will refer the parsing table if it detects the error entry then respective error message will be reported. This is how we get syntax errors when we compile our Program.

* YACC – Automatic Parser Generator :

YACC is an automatic tool for generating the Parser Program. YACC stands for "Yet Another Compiler Compiler", which is basically the utility available from Unix. Basically YACC is LALR Parser generator. The YACC can report conflicts or ambiguities in the form of error messages. LEX & YACC work together to analyse the program syntactically.



YACC : Parser generator model

YACC specification file need to be written first, for example "x.y" This file is given to the YACC compiler by Unix command as:

```
$ yacc x.y
```

Then it will generate a Parser Program using your YACC specification file. This Parser Program has a standard name as "y.tab.c". This is basically Parser Program in 'C'

generated automatically.

```
$yacc -d xc.y
```

By -d option two files will get generated one is "y.tab.c" & other is "y.tab.h". The header file y.tab.h will store all the tokens & we need not have to create "y.tab.h" explicitly. The generated y.tab.c program will then be compiled by 'C' compiler & generates the executable a.out file. We can test YACC program with the help of some valid & invalid strings.

* YACC specification file :-

YACC specification file consists of three parts: (a) declaration section, (b) translation rule section & (c) supporting 'C' functions.

Eg :-

```
%{
   /* declaration section */        (a)
%}
%%
   /* Translation rule section */   (b)
%%
/* Required 'c' functions */        (c)
```

YACC specification file

(a) Declaration section :- In this section ordinary 'C' declarations can be written. We can also declare grammar tokens in this section. The declaration of tokens should be after %{ %}.

(b) Translation Rule Section: It consists of all the Production rules of context free grammar with corresponding actions.

Ex:

Rule 1	action 1
Rule 2	action 2
⋮	⋮
Rule n	Action n

→ If there are more than one alternatives to a single rule then those alternatives should be seperated by "|" character. The actions are typical 'C' statements. If CFG is:

LHS → alternative 1 | alternative 2 | .... | alternative n

then:

LHS : alternative 1 {action 1}
| " 2 {" 2}
⋮
| alternative n {action n}

(c) 'C' functions section: This section consists of one main function in which the routine yyparse() will be called. And it also consists of required 'C' functions.

*Program : Write a YACC Program that will take arithmetic expression as input & Produce the corresponding Postfix expression as output.

Sol:

```
/* LEx Program to convert Infix expression to Postfix form */
%{
  #include <stdlib.h>
  #include <stdio.h>
  #include "y.tab.h" /* contain token definitions */
%}
%%
/* Valid digit */
[0-9]+ {
  /* convert from character form to integer form */
  yylval.no = atoi(yytext);
  /* Return the appropriate token */
  return(DIGIT);
}
/* Valid Identifier / variable */
[a-z|A-Z][a-z|A-Z|0-9|_]* {
    strcpy(yylval.str, yytext);
    return(ID);
}
/* Operators */
```

```
"+"   { return (PLUS); }
"-"   { return (MINUS); }
"*"   { return (MUL); }
"/"   { return (DIV); }
"^"   { return (EXPO); }
"("   { return (OPEN); }
")"   { return (CLOSE); }
"\n"  { return 0; }

/* Ignore white spaces */
[\t]  ;

/* Ignore remaining characters */
.     ;
%%
```

* YACC Program :

```
/* YACC Program to convert Infix expression to Postfix form */
%{
   #include <stdio.h>
%}
/* The identifier can be a digit (number) or a Variable (str) */
%union
  { int no;
    char str[10];
  }
```

```
/* Declaring the tokens */
%token <no> DIGIT
%token <str> ID
/* + and - are left associative & have the least Precedence */
%left PLUS MINUS
/* then comes * & / which are also left associative */
%left MUL DIV
/* And lastly exponent operator ^ which has higher Precedence
   & is right associative */
%right EXPO
/* Brackets which has the highest Precedence */
%left OPEN CLOSE
%%
STMT : EXPR          { Printf ("\n"); }
EXPR : EXPR PLUS EXPR    { Printf ("+"); }
     | EXPR MINUS EXPR   { Printf ("-"); }
     | EXPR MUL EXPR     { Printf ("*"); }
     | EXPR DIV EXPR     { Printf ("/"); }
     | EXPR EXPO EXPR    { Printf ("^"); }
     | OPEN EXPR CLOSE
     | DIGIT     { Printf ("%d", yyval.no); }
     | ID        { Printf ("%s", yyval.str); }
     ;
%%
```

```
/* user sub-routine section */
int main(void)
{
    Printf("\n");
    yyparse();
    Printf("\n");
    return 0;
}
```

* output :-

```
$ lex postfix.l
$ yacc postfix.y
$ gcc lex.yy.c y.tab.c -ll -ly
$ ./a.out
(2+3)*(3/4+4) - 3^2
23+34/4+*32^-
```