Shift Reduce Parser:

* Top down parser requires a grammar which do not have left recursion. But bottom up parsers can be built for variety of grammars even if the grammar has left recursion. It is more general & efficient. BU parsers handle a large class of grammars.

* In short, the process of obtaining the start symbol (root node) from string of terminals (leaves) is called the bottom up parser.

Informal defn of Handle:

A substring that matches the RHS of a production & replacing it with equivalent non-terminal on LHS of the production to get one step in RMD in reverse is called the handle.

Ex: $E \rightarrow E+T \mid T$

$T \rightarrow T * F \mid F$

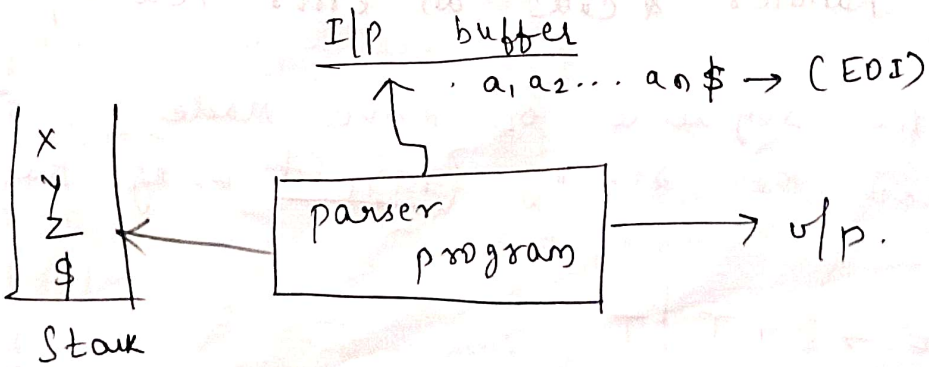
$F \rightarrow (E) \mid id$

Right sentential form in reverse	handle	Reduction prod^n
id * id	id	$F \rightarrow id$
F * id	F	$T \rightarrow F$
T * id	id	$F \rightarrow id$
T * F	T * F	$T \rightarrow T * F$
T	T	$E \rightarrow T$
E		

SR parser : It is an efficient way of implementing bottom up parser using explicit stack. The stack contains grammar symbols & i/p buffer holds the string to be parsed.

During parsing, always the handle will appear on top of the stack just before it is identified as the handle.

Block diagram:



	Stack	Input
Initial configuration	$S	w$
Final	$S	$

↑ start symbol of grammar.

The various actions performed by SR parser

* The parser keeps shifting the i/p symbols on to the stack till the handle β appears on top of the stack.
* The handle β is then reduced to appropriate LHS of the production.
* The above two steps are repeatedly performed till stack contains the start symbol & i/p is empty or an error is encounted.
* Shift (push): i/p symbol is shifted on to the stack top.
* Reduce (pop): By knowing the app. handle on the stack, parser reduces this to the LHS of the app. prodn.
* Accept: parser announces the successfull completion of parsing.
* Error: parser discovers an error & the app. error msg is displayed on the screen with the help of error handler & calls an error recovery routine.

Ex: Show the sequence of moves made by the SR parser for the string (id)+id using the following grammar:

$E \to E+T \mid T$

$T \to T*F \mid F$

$F \to (E) \mid id.$

(1)

Stack	I/p Buffer	Parsing Action
$	(id)+id$	Shift
$(	id)+id$	"
$(id	)+id$	Reduce F→id
$(F	)+id$	T→F
$(T	)+id$	E→T
$(E	)+id$	Shift
$(E)	+id$	R: F→(E)
$F	+id$	T→F
$T	+id$	E→T
$E	+id$	Shift
$E+	id$	"
$E+id	$	Reduce F→id
$E+F	$	T→F
$E+T	$	E→E+T
$E	$	Accept

(2) S→0S0 | 1S1 | 2 String : 10201

Stack	I/p buffer	Parsing Action
$	10201$	Shift
$1	0201$	"
$10	201$	"
$102	01$	R: S→2
$10S	01$	Shift
$10S0	1$	R: S→0S0
$1S	1$	Shift
$1S1	$	Re: S→1S1
$S	$	Accept

Conflicts in SRP:

① Shift reduce conflict: (SR)

whether to shift the next I/p symbol or reduce the current handle, with TOS.

② Reduce Reduce Conflict

which of several reductions to apply

Ex ①

$S \to AB$
$A \to 0S \mid 1S$
$B \to 0S1 \mid 1S$
string : 0S1S1

$S \to AB$
$A \to 0S \mid 1S$
$B \to 0S \mid 1S$
string $\to$ 0S1S.

Stack	I/p buffer	PA
$	0S1S1 $	shift
$0	S1S1 $	"
$0S	1S1$	SR conflict
$0S1 ?		
$0S A	1S1 $	shift
$ A1	S1 $	"
$ A1$S	1$	"
$ A1S1	$	Reduce
$ AB	$	"
$S	$	Accept

Ex 2) string 0S1S

Stack	I/p buffer	Parsing Action
$	0S1S$	shift
$0	S1S$	"
$0S	1S$	Reduce
$A	1S$	shift
$A1	S$	"
$A1S	$	R-R conflict
$AB	$	Reduce
$S	$	Accept
A		

LR Parsers :

- LR(0) → Least powerful
- SLR(1) → Simple LR
- LALR(1) → Look Ahead LR
- CLR(1) → Canonical LR. → Most powerful.

LR(0) items → LR(1) items.

Item def^n :

$S \to \cdot AA$ → no I/p has been seen.

$S \to A \cdot A$ → only A has been seen.

$S \to AA \cdot$ → meaning everything (i/p) in the RHS is seen so that it can be reduced to S.

Any production with • in RHS is called item.

Why imp ? ∵ in BUP, the main decision is to : when to reduce a terminal / string of terminals with LHS variable to reach start symbol.

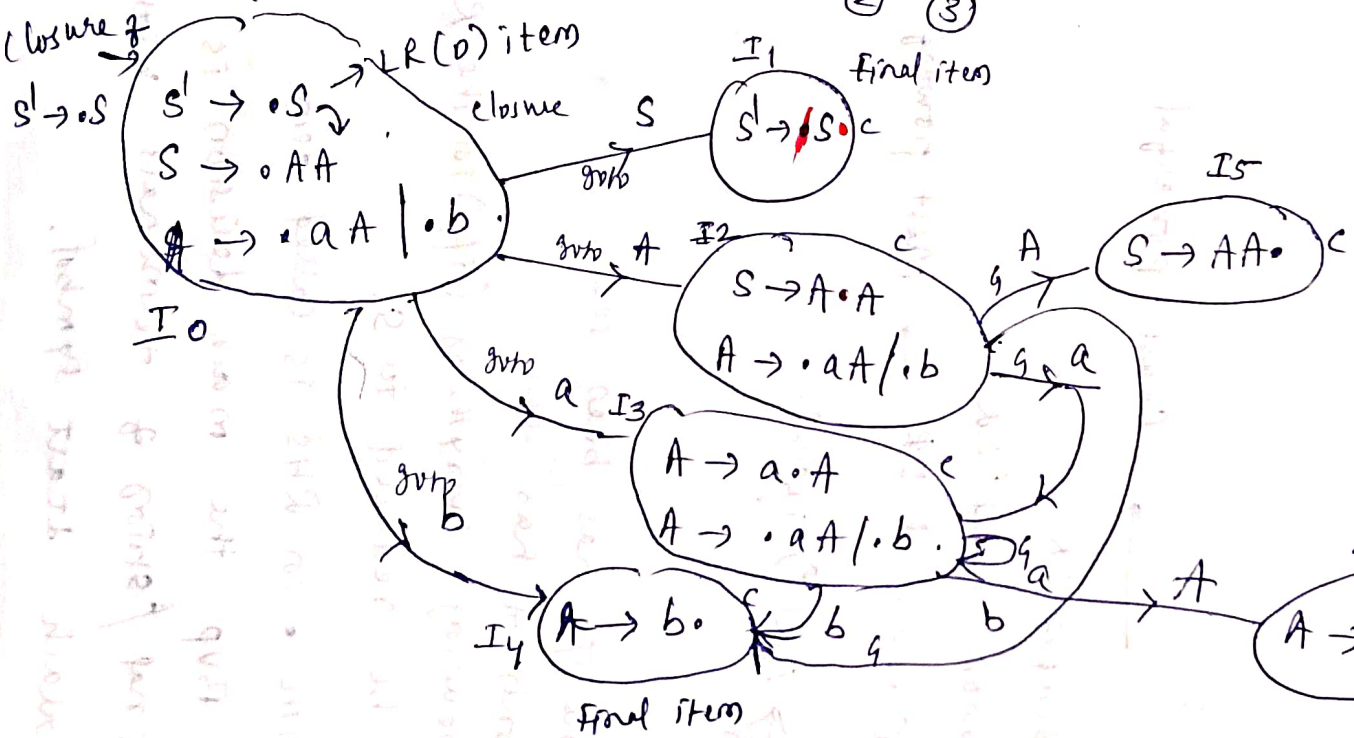
Ex 1] $S \to AA$
$A \to aA \mid b$

* whenever a grammar is given add one more grammar $S' \to S$ which prod^n 1 is known as augmented grammar.

i.e
$S' \to S$ ①
$S \to AA$
$A \to aA \mid b$
② ③

* Item → ~~Ex prod^n~~ &.

(closure of $S' \to .S$) → LR(0) item



Action (terminals) | GOTO (variable)

	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	r1	r1		
6	r2	r2	r2		

① no of rows equal to no of states.

② Shift move : transition of one state to other on a terminal.

③ Goto : ——— Variable

④ Reduce : Any state with final item → write reduce with the corresponding production no. in the entire row.

closure: whenever there is a • to the left of any variable, then add all productions of that variable.

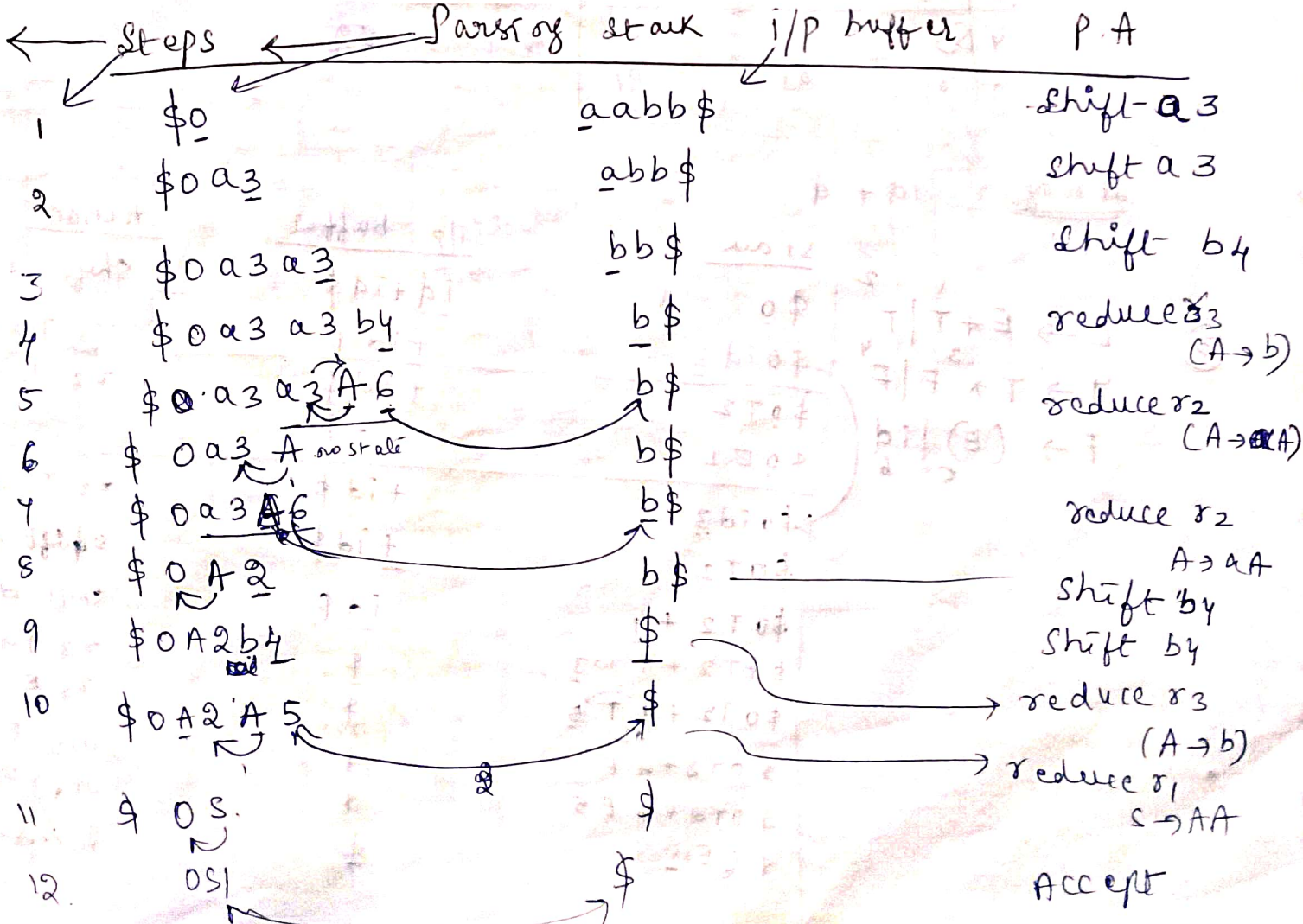
GOTO: In LR(0) item, if we move • towards Right what we will get.

Final item: whenever is • is at the right most side of any production.

$S' \rightarrow S\bullet$ → Accept state

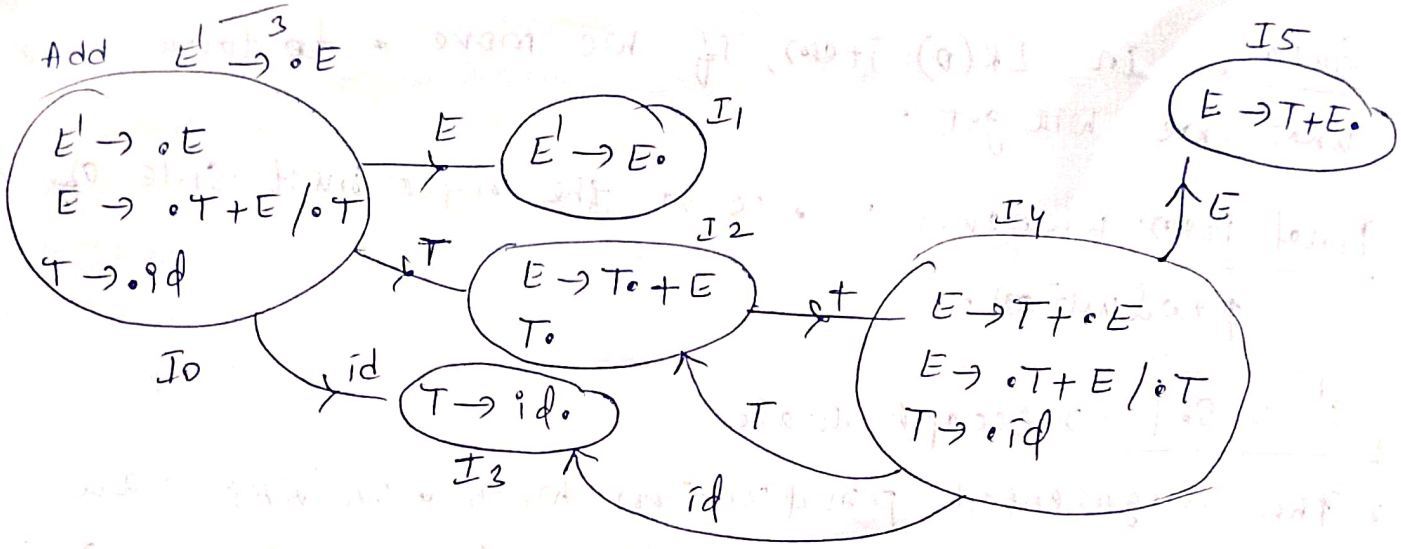
* The augmented production with • on RHS part
* The meaning everything of S (start symbol) has been seen. ∴ this special production is known as Accept state.

Parsing i/p string: aabb$



(2) $E \to \underset{1}{T+E} \mid \underset{2}{T}$

$T \to id$ (3)



	id	*	+	$	E	T
0	S3				1	2
1				Accept		
2	r_2		S4/r_2	r_2		
3	r_3		r_3	r_3		
4	S3				5	2
5	r_1		r_1	r_1		

String : id+id

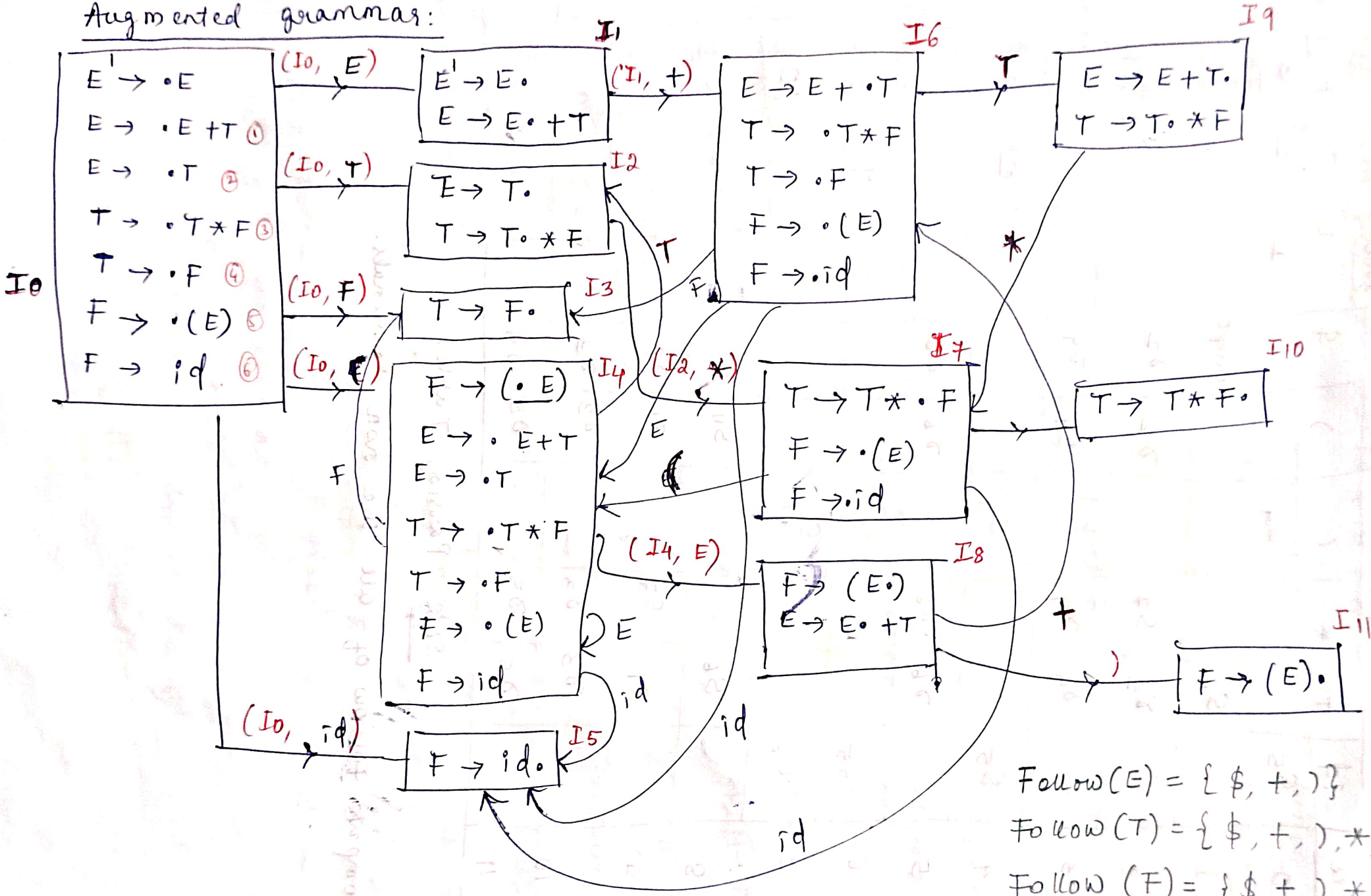
(3) $E \to \underset{1}{E+T} \mid \underset{2}{T}$

$T \to \underset{3}{T*F} \mid \underset{4}{F}$

$F \to \underset{5}{(E)} \mid \underset{6}{id}$

Stack	Ip buffer	Action
$ 0	id+id$	shift id 3
$ 0 id 3	+id$	r_3 T→id
$ 0 T 2	+id$	r_2 E→T
$ 0 E 1	+id$	
$ 0 id 3	+id$	r_3 T→id
$ 0 T 2	+id$	shift +4
$ 0 T 2 + 4	id$	shift id 3
$ 0 T 2 + 4 id 3	$	r_3 T→id
$ 0 T 2 + 4 T 5	$	r_1 E→T+E
$ 0 T 2 + 4 E	$	r_2 E→T
$ 0 T 2 + 4 E 5	$	r_1 S→T+E
$ 0 E 1	$	Accept

Augmented grammar:



Follow(E) = { $, +,) }

Follow(T) = { $, +,), * }

Follow (F) = { $, +,), * }

State	Action						goto		
	id	+	*	(	)	$	E	T	F
0	S_5			S_4			1	2	3
1		S_6				Accept			
2		γ_2	S_7		γ_2	γ_2			
3		γ_4	γ_4		γ_4	γ_4			
4	S_5			S_4			8	2	3
5		γ_6	γ_6		γ_6	γ_6			
6	S_5			S_4				9	3
7	S_5			S_4					10
8		S_6			S_{11}				
9		γ_1	S_7		γ_1	γ_1			
10		γ_3	γ_3		γ_3	γ_3			
11		γ_5	γ_5		γ_5	γ_5			

SLR Parsing Table.

* Compute FOLLOW of all the non-terminals.
*

i/p String Parsing: w = id * id + id

Stack	Input buffer	Action
$0	id * id + id $	shift id 5
$0 id 5	* id + id $	reduce 6 F→id
$0 F 3	* id + id $	reduce 4 T→F
$0 T 2	* id + id $	shift * 7
$0 T 2 * 7	id + id $	shift id 5
$0 T 2 * 7 id 5	+ id $	reduce 6 F→id
$0 T 2 * 7 F 10	+ id $	reduce 3 T→T*F
$0 T 2	+ id $	reduce 2 E→T
$0 E 1	+ id $	shift + 6
$0 E 1 + 6	id $	shift id 5
$0 E 1 + 6 id 5	$	reduce 6 F→id
$0 E 1 + 6 F 3	$	reduce 4 T→F
$0 E 1 + 6 T 9	$	reduce 1, E→E+T
$0 E 1	$	Accept

Note: In the parsing table for LR(0), the reduce rule for the production is placed in the entire row, across all the terminals whereas
In SLR parsing table, the reduce rule for the production is placed only in the follow set of LHS non-terminal of the reduce production.

(2) $S \to L = R$
$S \to R$
$L \to *R$
$L \to id$
$R \to L$

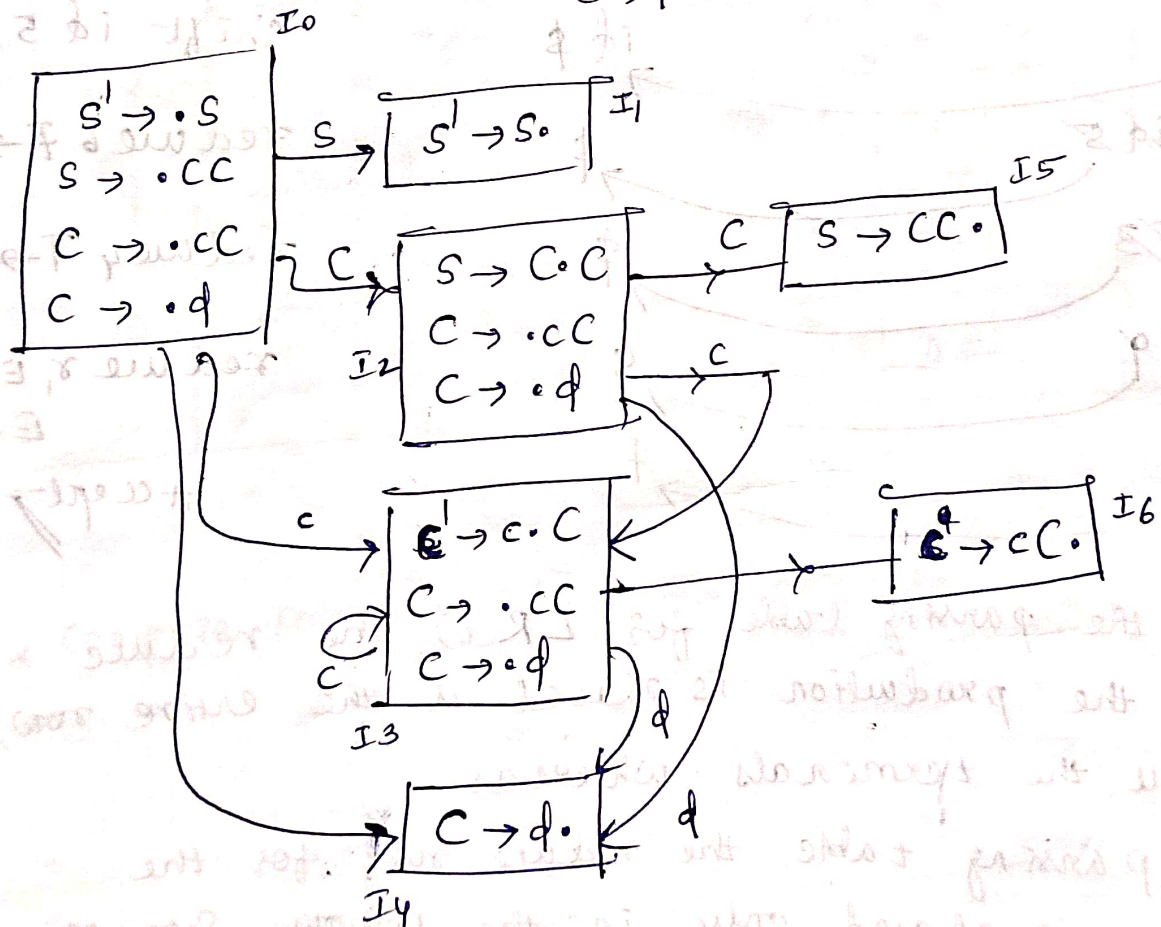
(3) $A \to a \mid (A)$
I/p string: ((a))

(4) $S \to SS+$
$S \to SS*$
$S \to a$

String: aa*a+

(5) 1. $S \to CC$
2. $C \to cC$
3. $C \to d$

Sol^n Add augmented production
$S' \to S$
$S \to CC$
$C \to cC$
$C \to d$



Follow $S \to \{\$\}$
$C \to \{c, d, \$\}$

	Action			Goto	
	c	d	$	S	C
0	s_3	s_4		1	2
1			Accept		
2	s_3	s_4			5
3	s_3	s_4			6
4	r_3	r_3	r_3		
5			r_1		
6	r_2	r_2	r_2		

SLR Parsing Table:

* show parsing of the below string ccdd.

Stack	Input	Action
0	ccdd $	Shift c3
0 3	cdd $	shift c3
0 3 3	dd$	shift d4
0 3 3 4	d $	r_3 C→d
0 3 3 6	d $	r_2 C→cC
0 3 6	d $	r_2 C→cC
0 2	d $	Shift d4
0 2 4	$	r_3 C→d
0 2 5	$	r_1 S→CC
0 1	$	Accept.

$S \to AA$ (1)

$A \to aA/b$ (2) (3)

$A \to \alpha \cdot B\beta, a/b$

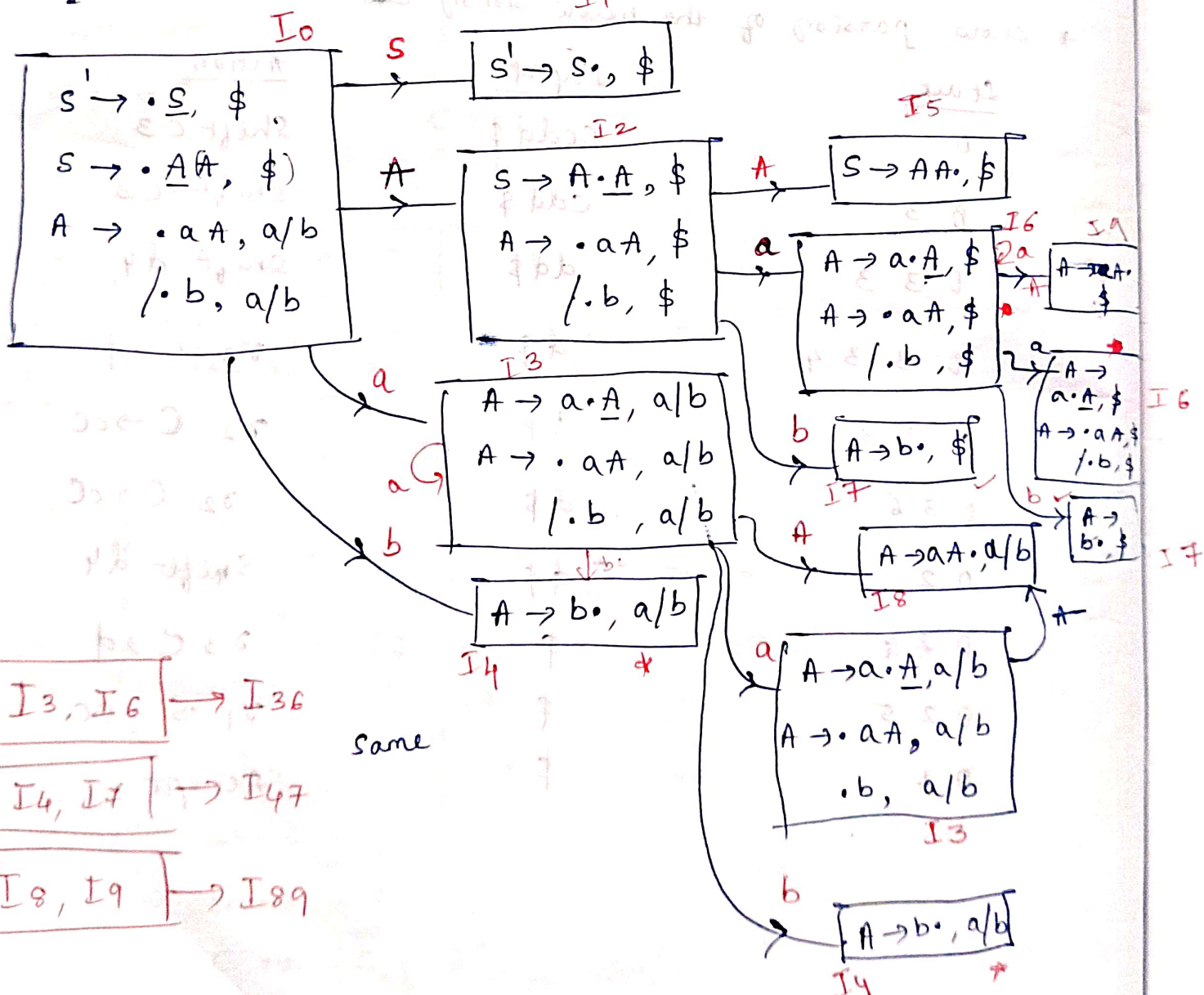
$B \to \cdot\gamma$, find first $(\beta, a/b)$

no β, then find first (a/b) & add

LALR & CLR Parser

* Uses canonical collection of LR(1) items.
* **LR(1) item = LR(0) item + Look ahead**
* Look ahead is used to place reduce move.
* More no. of blank spaces in the parsing table indicates Error detecting capability of a parser.

* Look ahead is $ always for augmented production.



$I_3, I_6 \to I_{36}$

$I_4, I_7 \to I_{47}$

$I_8, I_9 \to I_{89}$

same

CLR(1) Parsing Table:

	a	b	$	S	A
0	S_3	S_4			2
1					
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			9
7			r_3		
8	r_2	r_2			
9			r_2		

no of states in

$$CLR(1) \geq LR(0) = SLR(1) = LALR(1)$$

LALR(1) Parsing Table → Merge the states with same LR(0) item but diff lookahead.

	a	b	$	S	A
0	S_{36}	S_{47}			2
1					
2	S_6	S_7			5
36	S_{36}	S_{47}			89
47	r_3	r_3	r_3		
89	r_2	r_2	r_2		
5			r_1		

Ex 2]

~~S → DD~~
~~D → cD~~
~~D → d~~

S → L = R | R
L → * R | id
R → L

parse the string id = id

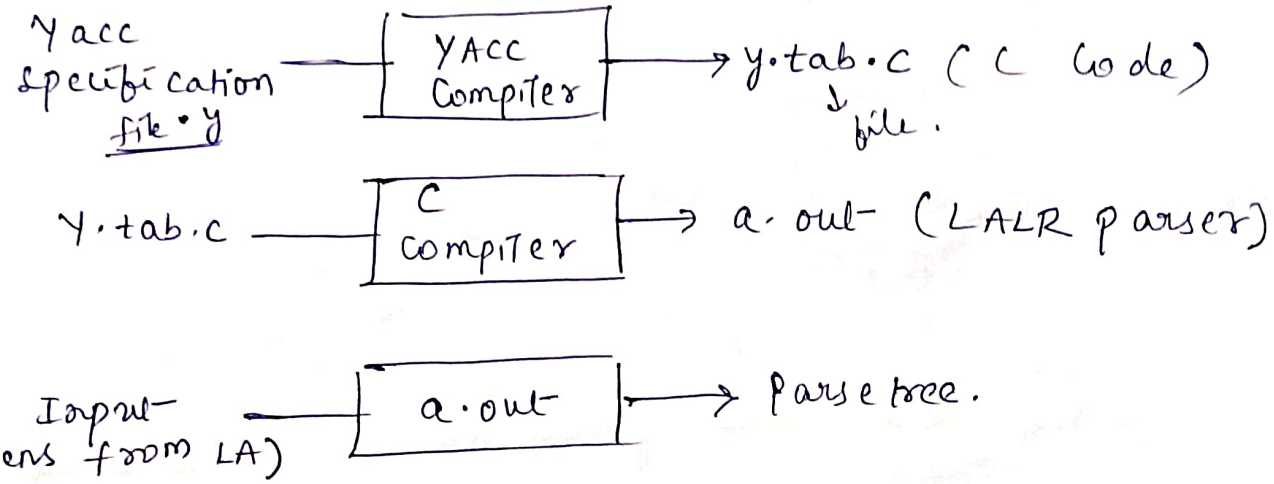
* ~~S → .S~~
~~S → .DD~~
~~D → .cD~~

$S' \to .S, \$$
$S \to .L = R, \$$
$S \to .R, \$$
$L \to .*R, =$
$L \to .id, =$
$R \to .L, \$$

YACC (Yet Another Compiler Compiler)

* Developed by Stephen Johnson.
* It is a tool for generating Look Ahead Left to Right (LALR) parser.
* It takes i/p from the Lexical Analyzer & generates parse tree.
* Syntax Analyzer / parser is the second phase of the compiler which takes i/p as tokens & generates a parse tree.

BLOCK DIAGRAM:



* file.y → contain desired grammar in yacc format.

SYNTAX:

{definitions / declarations}

%%

tokens

%%

{Rules} head : body1 {action} | body2 {action}

%%

{Auxiliary routines / supplementary code (C Code)}

To design program in yacc language:

* yyval - values associated with the tokens are returned by lex in the variable yyval.

* yytext - ptr to the i/p character stream / matched i/p string.

* yywrap - called by lex or yacc when i/p is exhausted/ finished (returns 1 when i/p is finished)

* yyparse() - responsible for parsing to occur. It reads tokens & executes the actions.
  * If it returns 0 means string is accepted.

	id	=	*	$	S	L	R
0	S5		S4		1	2	3
1				Accept			
2		S6		r5			
3				r2			
4	S5		S4			8	7
5		r4		r4			
6	S12		S11			10	9
7		r3		r3			
8		r5		r5			
9				r1			
10				r5			
11	S12		S11				
12				r4			
13				r3			

① S → L=R
② S → R
③ L → *R
④ L → id
⑤ R → L

Stack	I/p string.	Action
$0	id=id$	S5
$0id5	=id$	reduce 4 L→id
$0L2	=id$	S6
$0L2=6	id$	S5 S12
$0L2=6id12	$	r4 L→id
$0L2=6L10		r5 R→L
$0L2=6R9		r1 S→L=R
$0S1		Accept.

Ex 3] $S \to aAd \;(1) \mid bBd \;(2) \mid aBe \;(3) \mid bAe \;(4)$

$A \to c \;(5)$

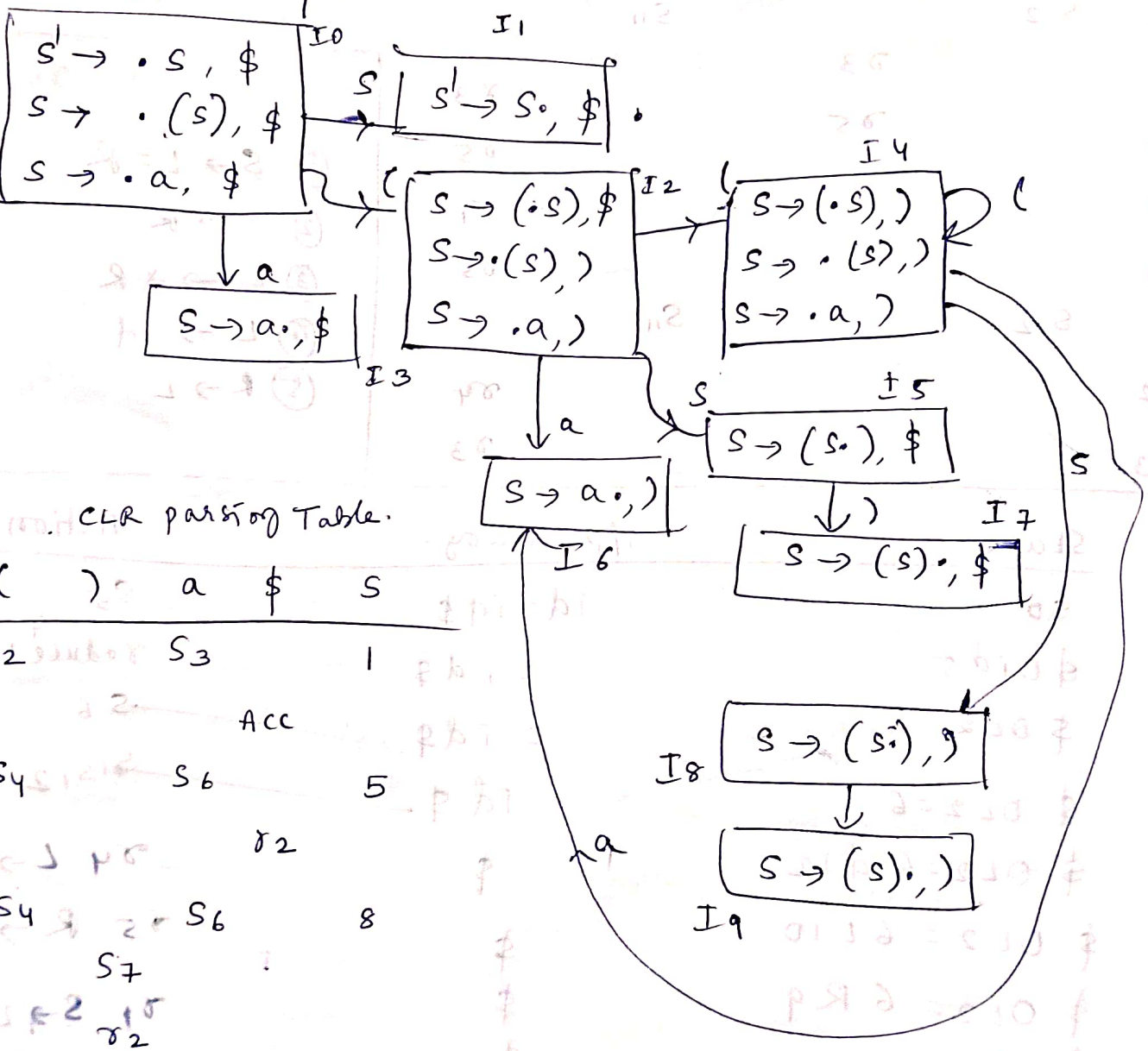
$B \to c \;(6)$

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

$A \to d$

$B \to d$

5] $S \to (S) \mid a$



CLR parsing Table.

	(	)	a	$	S
0	S_2		S_3		1
1				Acc	
2	S_4		S_6		5
3		γ_2			
4	S_4		S_6		8
5		S_7			
6		γ_2			
7				γ_1	
8		S_9			
9		γ_1			

$I_2, I_4 \rightarrow I_{24}$
$I_3, I_6 \rightarrow I_{36}$

$I_5, I_8 \rightarrow I_{58}$
$I_7, I_9 \rightarrow I_{79}$

	(	)	a	$	S
0	S_2		S_3		1
1				Accept	
24	S_4		S_6		S_8
36		γ_2		γ_2	
58		S_{79}			
79		γ_1		γ_1	

* In $\begin{pmatrix} CLR \\ LALR \end{pmatrix}$ parser, SR conflict will not arise.

But RR conflict will ~~not~~ arise in LALR but not in CLR.

i/p string $((a))\$$