Assignment - 3

Q1 Prepare operator Precedence table for grammar given

bexpr → bexpr or bterm | bterm
bterm → bterm and bfactor | bfactor
bfactor → not bfactor | (bexpr) | true | false

Parse : not (true or false and true)

	FIRSTOP+	LASTOP+
bexpr	not, (, true, false, and bexpr, or, bterm, bfactor	false, true,), not, bfactor bterm, or, and
bterm	not, (, true, false, bterm, and, bfactor	false, true,), not bfactor, and
bfactor	not, (, true, false	false, true,), bfactor, not

	or	and	not	(	)	true	false	$
or	·>	<·	<·	<·	·>	<·	<·	·>
and	·>	·>	<·	<·	·>	<·	<·	·>
not	·>	·>	<·	<·	·>	<·	<·	·>
(	<·	<·	<·	<·	=·	<·	<·	—
)	·>	·>	—	—	·>	—	—	·>
true	·>	·>	—	—	·>	—	—	·>
false	·>	·>	—	—	·>	—	—	·>
$	<·	<·	<·	<·	—	<·	<·	ACCEPT

(i) or bterm
and bfactor
not bfactor
(bexpr

(ii) bexpr or
bterm and
bexpr)

(iii) (bexpr)

(iv) $ <· F(bexpr)
(v) L(bexpr) ·> $

Parsing : not (true or false and true)

Stack	Relation	Input	Move/Action
$	<·	not (true or false and true) $	Shift not
$ not	<·	(true or false and true) $	Shift (
$ not (	<·	true or false and true) $	Shift true
$ not (true	·>	or false and true) $	Reduce bfactor → true
$ not (bfactor	<·	or false and true) $	Shift or
$ not (bfactor or	<·	false and true) $	Shift false
$ not (bfactor or false	·>	and true) $	Reduce bfactor → false
$ not (bfactor or bfactor	<·	and true) $	Shift and
$ not (bfactor ... ...or bfactor and	<·	true) $	Shift true
$ not (bfactor or bfactor and true	·>	) $	Reduce bfactor → true
$ not (bfactor or bfactor and bfactor	·>	) $	Reduce bfactor → bfactor and bfactor
$ not (bfactor or bterm	·>	) $	Reduce bexpr → bexpr or bterm
$ not (bexpr	≐	) $	Shift)
$ not (bexpr)	·>	$	Reduce (bfactor) → bfactor
$ not bfactor	·>	$	Reduce not bfactor → not bfactor
$ bfactor	ACCEPT	$	

The following string is successfully parsed.

Q2 Prepare Operator precedence table for grammar.

$S \rightarrow xAy \mid xBy \mid xAz$

$A \rightarrow aS \mid q$

$B \rightarrow q$

	FIRSTOP+	LASTOP+
S	x	y, z
A	a, q	q, S, a, y, z
B	q	q

	x	y	z	a	q	$
x	-	$\doteq$	$\doteq$	$<\cdot$	$<\cdot$	-
y	-	$\cdot>$	-	-	-	$\cdot>$
z	-	$\cdot>$	-	-	-	$\cdot>$
a	$<\cdot$	$\cdot>$	-	-	-	-
q	-	$\cdot>$	-	-	-	-
$	$<\cdot$	-	-	-	-	-

(i) (ii) (iii)

(i)	(ii)	(iii)
xA	Ay	xAy
xB	By	xBy
xA	Az	xAz
aS		

VISION

Q8 Construct CLR parsing table :

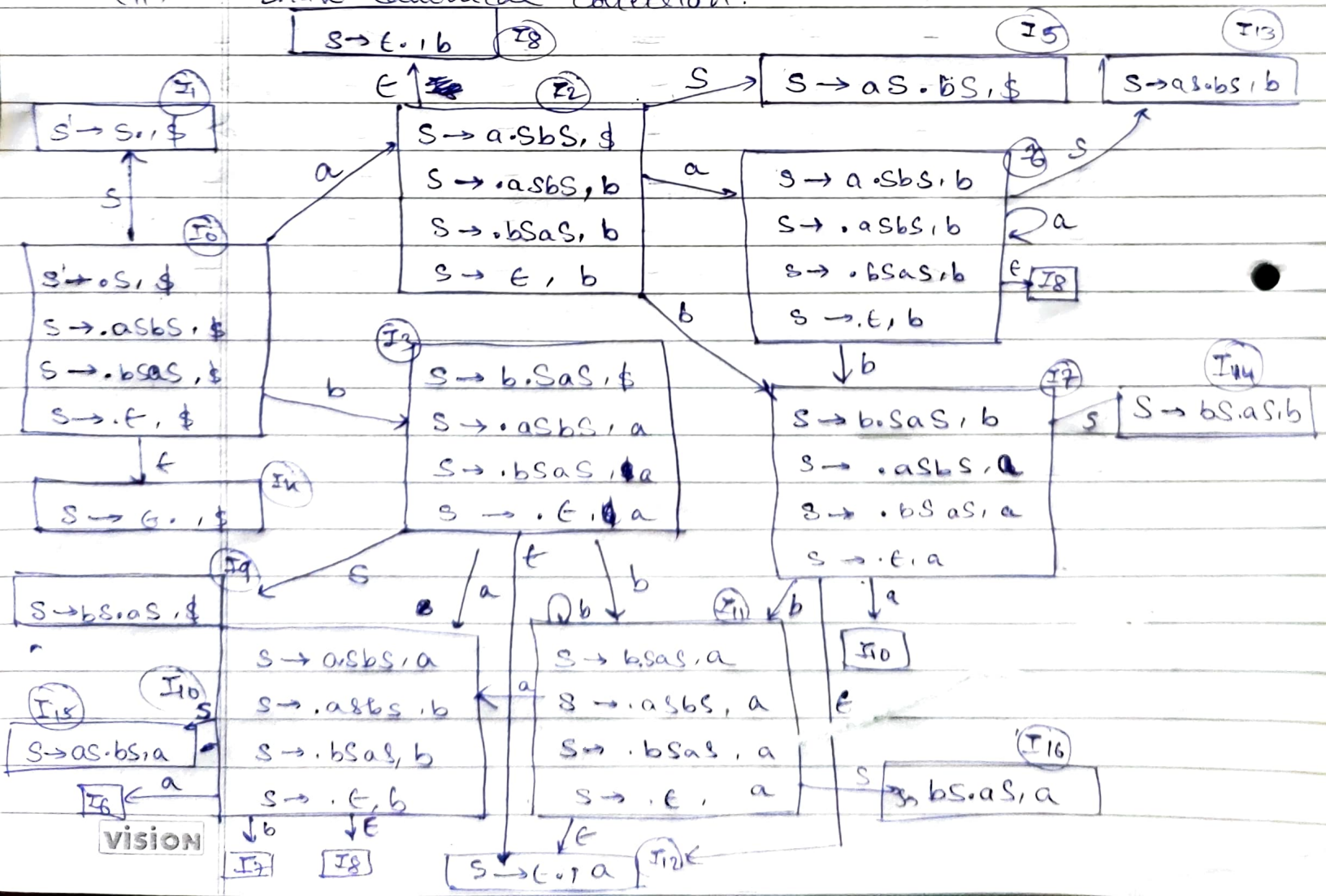
$S \rightarrow aSbS \mid bSaS \mid \epsilon$

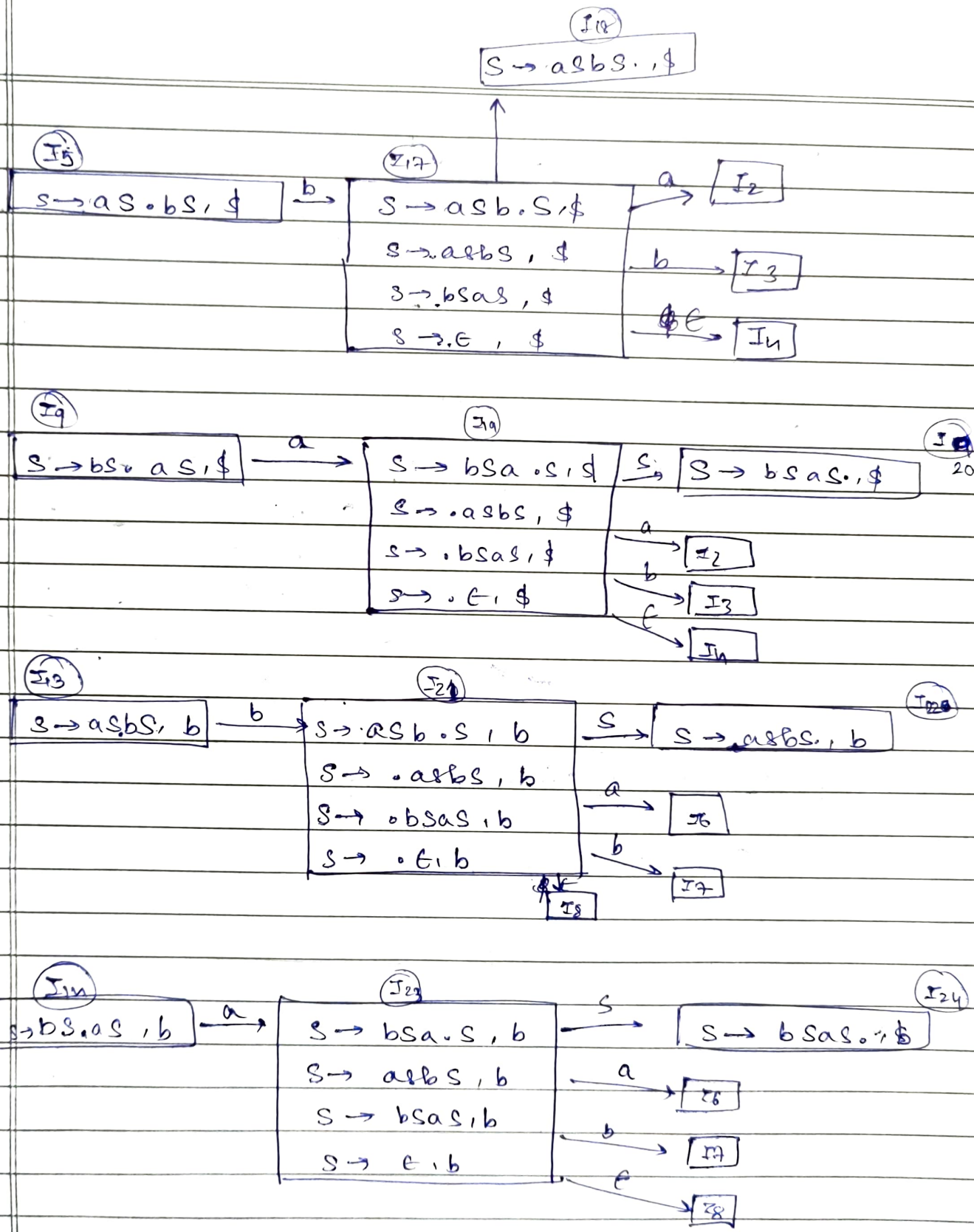
Andalso parse :, "abba".

(i) ~~Augmented grammar & find footsteads~~

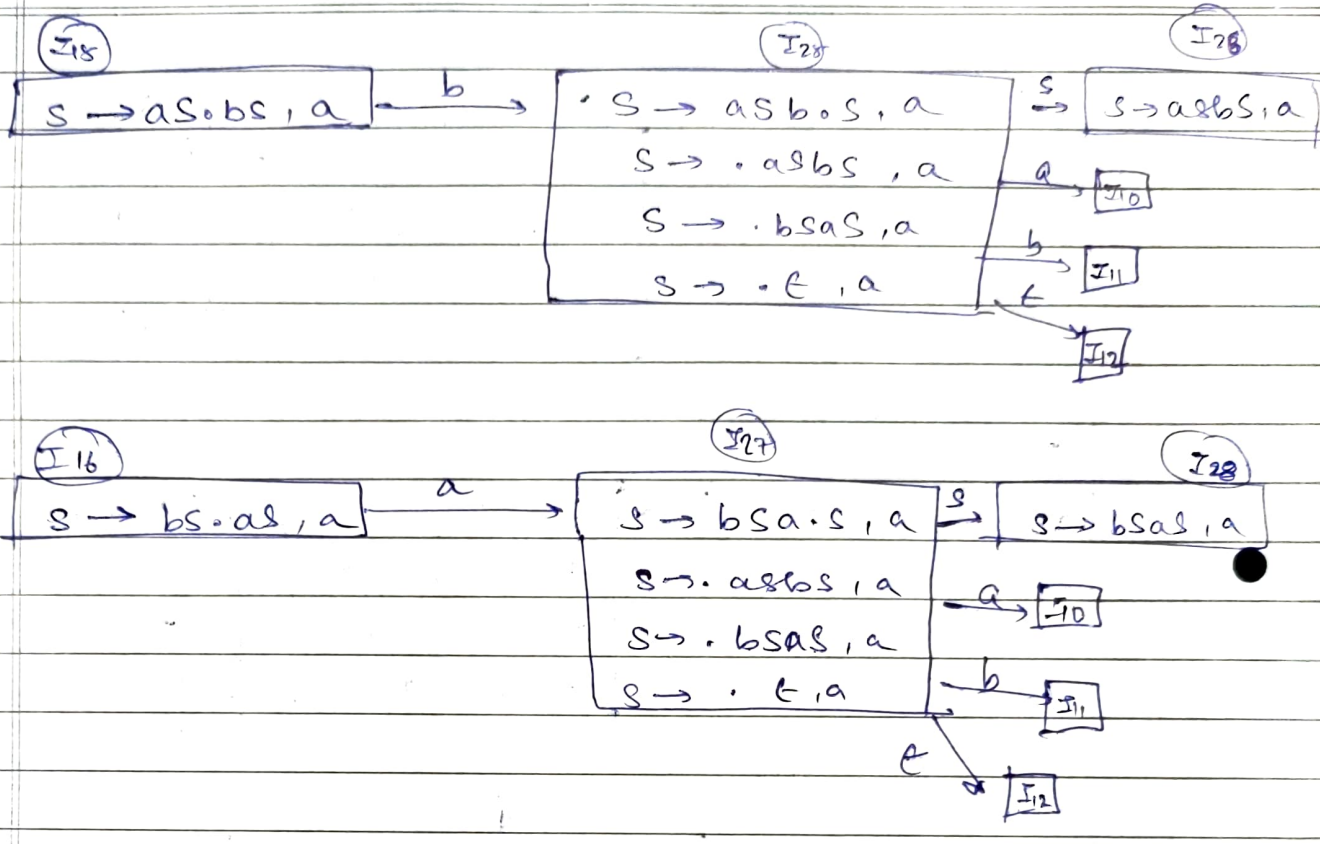
$S' \rightarrow .S, \$$
$S \rightarrow .aSbS$
$S \rightarrow .bSaS$
$S \rightarrow .\epsilon$

(ii) Draw canonical collection.




I18
S → aSbS . , $
I5
S → aS . bS , $
b
I17
S → aSb . S , $
S → . aSbS , $
S → . bSaS , $
S → . ∈ , $
a
I2
b
I3
$ ∈
I4
I9
S → bSa . aS , $
a
I19
S → bSa . S , $
S
S → bSaS . , $
I20
a
I2
b
I3
∈
I4
I13
S → aS . bS , b
b
I21
S → aSb . S , b
S → . aSbS , b
S → . bSaS , b
S → . ∈ , b
S
I22
a
I6
b
I7
$
I8
I14
S → bS . aS , b
a
I23
S → bSa . S , b
S → aSbS , b
S → bSaS , b
S → ∈ , b
S
I24
S → bSaS . , $
a
I6
b
I7
∈
I8

VISION



(iii) Number the productions

S → aSbS (1)

S → bSaS (2)

S → ∈ (3)

CLR Parsing Table

	Action				GOTO
State	a	b	c	$	S
I_0	S_2	S_3	S_4		1
I_1				ACCEPT	
I_2	S_6	S_7	S_8		5
I_3	S_{10}	S_{11}	S_{12}		9
I_4				γ_3	
I_5		S_{17}			
I_6	S_6	S_7	S_8		13
I_7	S_{10}	S_{11}	S_{12}		14
I_8		γ_3			
I_9	S_{19}				
I_{10}	S_6	S_7	S_8		15
I_{11}	S_{10}	S_{11}	S_{12}		16
I_{12}	γ_3				
I_{13}		S_{21}			
I_{14}	S_{23}				
I_{15}		S_{25}			
I_{16}	S_{27}				
I_{17}	S_2	S_3	S_4		18
I_{18}				γ_1	
I_{19}	S_2	S_3	S_4		20
I_{20}				γ_2	
I_{21}	S_6	S_7	S_8		22
I_{22}		γ_1			
I_{23}	S_6	S_7	S_8		22

STATE	ACTION				GOTO
	a	b	∈	$	S
I_{24}		γ_2			
I_{25}	S_{10}	S_{11}	S_{12}		26
I_{26}	γ_1				
I_{27}	S_{10}	S_{11}	S_{12}		28
I_{28}	γ_2				

Parse : "abba"

Stack	Input	Action / Move
$0	abba$	Shift 2
$0a2	bba$	Shift 7
$0a2b7	ba$	Shift 11
$0a2b7b11	a$	Shift 10
$0a2b7b11a10	$	

Q4 Check whether given grammar is CLR (1) or not? Also check for LALR (1).

$S \rightarrow aPbSQ \mid a$

$Q \rightarrow t \mid \epsilon$

$P \rightarrow r$

Q5 Augment Grammar & number the productions.

$S' \rightarrow .S$

$S \rightarrow .aPbSQ$ ①

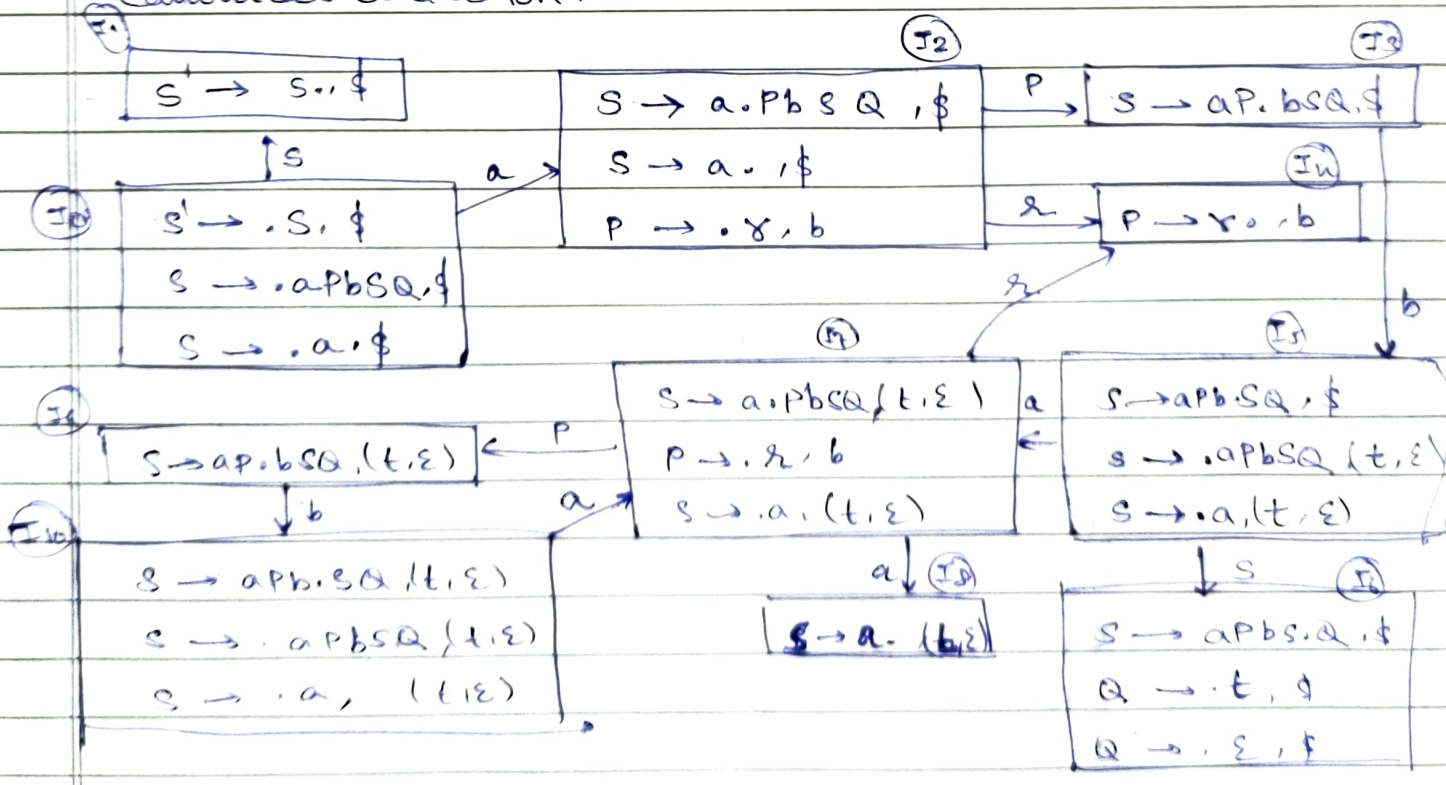
$S \rightarrow .a$ ②

$Q \rightarrow .t$ ③

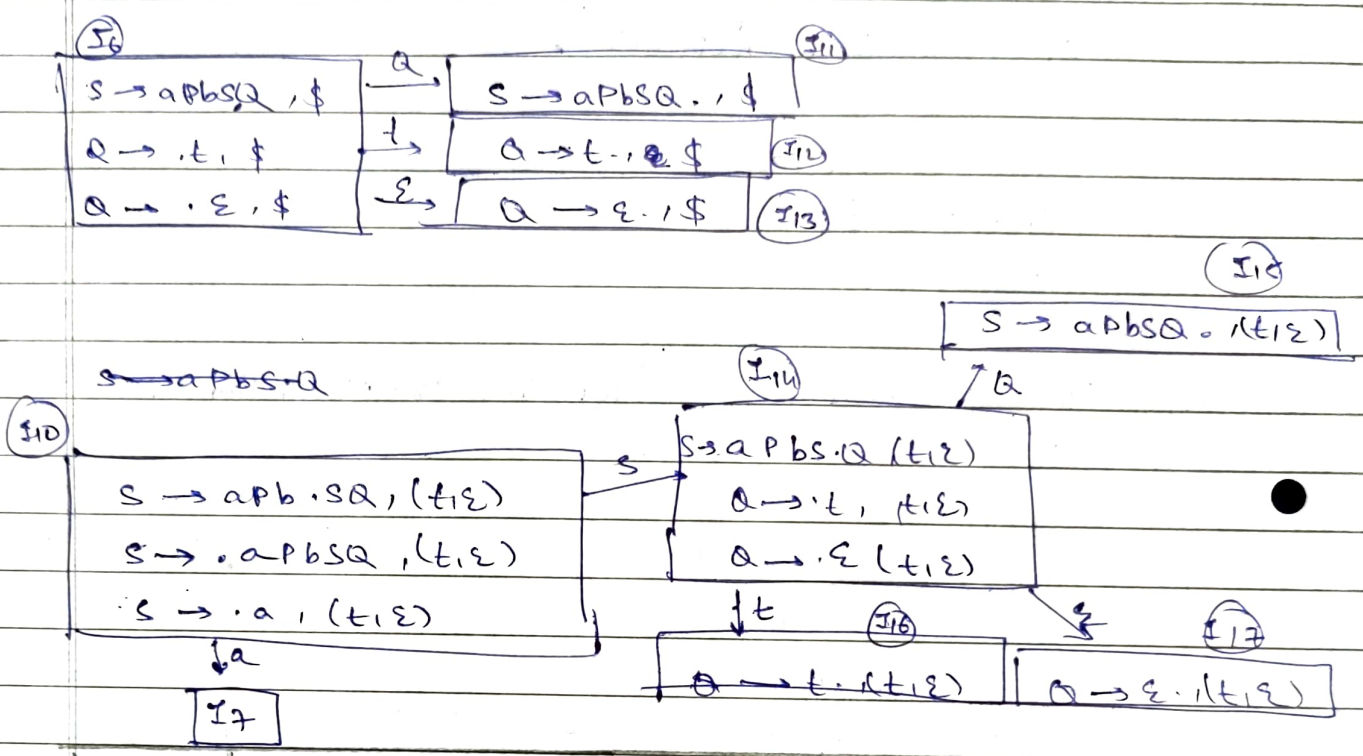
$Q \rightarrow .\epsilon$ ④

$P \rightarrow .r$ ⑤

(i) Canonical collection.



VISION



CLR

STATE	ACTION						GOTO		
	a	b	t	ε	ε	$	S	P	Q
I_0	S_2						1		
I_1						ACCEPT			
I_2				S_4				3	
I_3		S_5							
I_4		γ_5							
I_5	S_7						6		
I_6			S_{12}		S_{13}				11
I_7				S_4					
I_8	S_8							9	
I_9		S_{10}							
I_{10}	S_7						14		
I_{11}						γ_1			
I_{12}						γ_3			
I_{13}						γ_4			
[illegible]									
[illegible]									

VISION

	a	b	t	$\mathcal{R}$	$\mathcal{E}$	$	S	P	Q
I_{14}			S_{16}		S_{17}				15
I_{15}			$\mathcal{R}_1$		$\mathcal{R}_1$				
I_{16}			$\mathcal{R}_3$		$\mathcal{R}_3$				
I_{17}			$\mathcal{R}_4$		$\mathcal{R}_4$				

LALR

Similar production canonical collections:

$I_2 - I_7$

$I_3 - I_9$

$I_5 - I_{10}$

$I_6 - I_{14}$

$I_{11} - I_{15}$

$I_{12} - I_{16}$

$I_{13} - I_{17}$

LALR

	a	b	t	$\mathcal{R}$	$\mathcal{E}$	$	S	P	Q
I_0	S_2						1		
I_1						ACCEPT			
I_{27}				S_4				3	
I_{39}		S_5/S_{10}							
I_4		$\mathcal{R}_5$							
I_{5-10}	S_7						G_{14}		
I_{G14}			S_{16}/S_{12}		S_{17}/S_{13}				11/15
I_8	S_8							9	
I_{11-15}			$\mathcal{R}_1$			$\mathcal{R}_1$			
I_{12-16}			$\mathcal{R}_3$			$\mathcal{R}_3$			
I_{13-17}			$\mathcal{R}_4$			$\mathcal{R}_4$			

Yes, the given grammar is CLR & LALR.

Q5 Explain Error Recovery of LL, operator precedence and LR parsing with examples.

→ (i) For LL(1) Parsers

→ Predictive Parsers have two modes of error recovery:

(a) Panic Mode Recovery

→ It is based on the idea of skipping symbols on the input until a token in a selected set of synchronizing tokens appears.

→ Pop items off the parse stack until a synchronization token is at the top of the stack.

→ Read the input symbols until the current lookahead symbol matches the symbol at the top of the stack or until you reach the end of the input.

→ If you are at the end of the input, error recovery failed, otherwise you have recovered.

(b) Phrase Mode Recovery

→ It can be implemented by filling in the blank entries in the LL parsing table with error routines that are called when that entry in the table is accessed.

Examples: $E \rightarrow E+T \mid T$
$T \rightarrow T*F \mid F$
$F \rightarrow (E) \mid id$

Input : id + * (id ...
corrected : id + (id ...
[REMOVED +]

(ii) Operator Precedence Parsers

(a) Error Cases

→ No relation holds between the terminal on the top of the stack and the next input symbol.

→ A handle is found, but there is no production with this handle as a right side.

(b) Error Recovery

→ Each empty is filled with a pointer to an error routine.

→ Decides the popped handle "looks like" which right-hand side. And tries to recover from that situation.

(c) Handling shift/reduce errors

→ To take care of such type of errors we must modify : Stack, Input or Both.

Example: If the input string is "2*+3", the parser encounters an error because it cannot determine the correct way to handle the "+3" part. Error recovery in this context typically involves selecting the correct operator based on the precedence and associativity rules.

(iii) LR Parsing

(a) Panic Mode Recovery

→ We can implement panic-mode error recovery by scanning down the stack until a state s with a goto on a particular nonterminal A is found.

zero or more input symbols are then discarded until a symbol a is found that can legitimately follow A.

(b) Phrase-level Recovery

→ Phrase-level recovery is implemented by examining each error entry in the LR action table and deciding on the basis of language usage the most likely programmer error that would give rise to that error.

Example: $S \rightarrow E$

$E \rightarrow E + E$

$E \rightarrow id$

And the input string is "id + id id +". The parser encounters a shift-reduce conflict when it sees the second id, as it can either shift or reduce the previous "id + id" to E. Error recovery strategies may involve choosing to shift in this context.

Q6 Write SDT rules & Draw parse tree for each.

a) To convert binary numbers to decimals and calculate the value of the number 11011.

→ (i) Start with an accumulator initialised to 0.

(ii) For each binary digit from left to right:

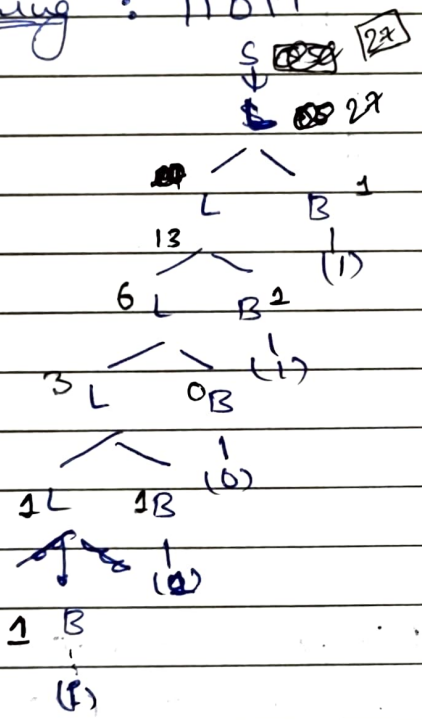
(a) Multiply the accumulator by 2.

(b) Add the value of the current binary digit to the accumulator.

SDT Rules

$S \rightarrow L$	$\{S.dv = L.dv\}$
$L \rightarrow LB$	$\{L.dv = 2 * L.dv + B.dv\}$
$L \rightarrow B$	$\{L.dv = B.dv\}$
$B \rightarrow 0$	$\{B.dv = 0\}$
$B \rightarrow 1$	$\{B.dv = 1\}$

String : 11011



Binary : 11011

Decimal : 27

(b) To convert binary float numbers to decimals & calculate the value of number 11011.0101

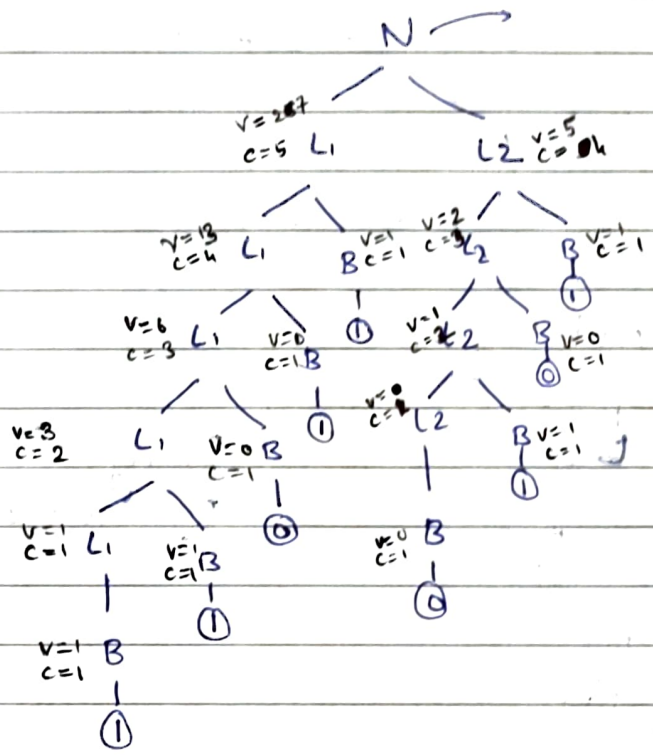
SDT Rules :

$N \rightarrow L_1\ L_2$	$\{N.val = L_1.val + L_2.val / 2^{L_2.c}\}$
$L \rightarrow L_1\ B$	$\{L.val = L_1.val * 2 + B.val, L.c = L_1.c + B.c\}$
$L \rightarrow B$	$\{L.val = B.val, L.c = B.c\}$
$B \rightarrow 0$	$\{B.val = 0;\ B.c = 1\}$
$B \rightarrow 1$	$\{B.val = 1,\ B.c = 1\}$

Here c stands for count & val stands for value

Example : 11011.0101

$V = 27 + \frac{5}{2^4} = 27 + \frac{5}{16} = 27.3125$



Binary : 11011.0101

Decimal : 27.3125

(c) For generating 3A Code.

S → id = E { gen (id.name = E.place) ; }

E → E + T { E.place = newtemp() ;
gen (E.place = E.place + T.place) ; }

E → T { E.place = T.place }

T → T1 * F { T.place = newtemp() ;
gen (T.place = T1.place * F.place) ; }

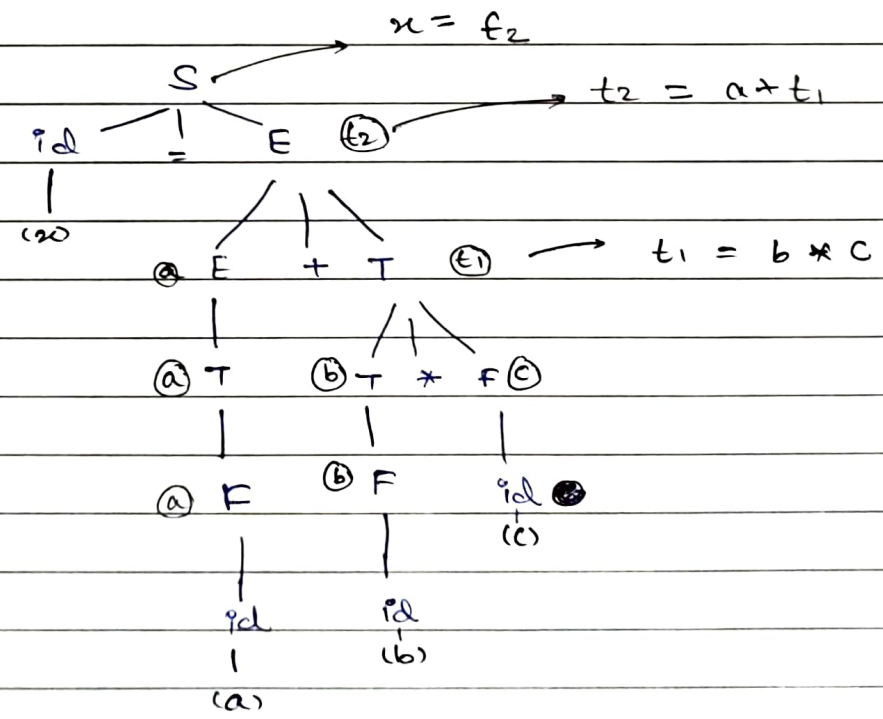
T → F { T.place = F.place }

F → id { F.place = id.name }

SDT Rules :-

Binary → Digit { Binary.value = Digit.value ; }
Binary → Binary Digit { Binary.value = (Binary.value * 2) + Digit.value ; }
Digit → '0' { Digit.value = 0 ; }
Digit → '1' { Digit.value = 1 ; }

Taking string $x = a + b * c$



3A Code :-

$t_1 = b * C$
$t_2 = a * t_1$
$x = t_2$