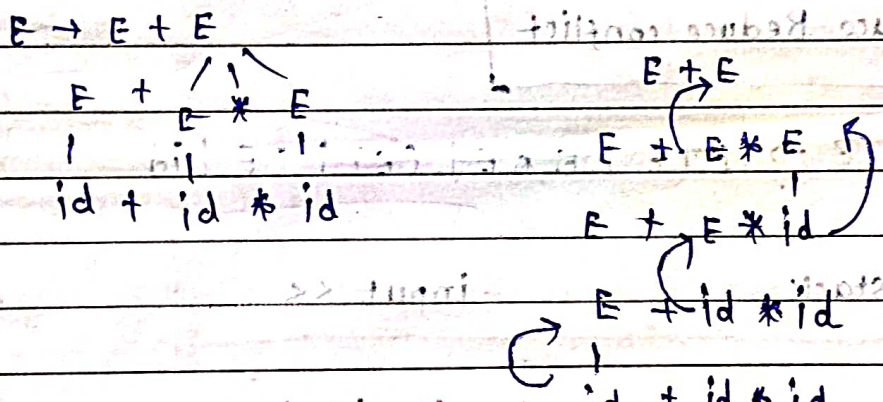
Bottom Up Parsing (Right-to left in Reverse Order)

- SRP
- operator precedence
- LR Parser
  - SLR
  - CLR
  - LALR

$E \rightarrow E+E \mid E*E \mid (E) \mid -E \mid id$

string = id + id * id



SHP (Shift Reduce Parcer) :-

Handle

id + id * id

E + id * id

Handle → Terminal to Non Terminal

E + E * id

Handle

E + E * E

Handle pruning

E + E

Handle Pruning

E

Handle pruning → Right side value Replaces with left side

E * E → E

(ii) Shift :- The next input symbol is shifted on to the top of the stack

(iii) Reduce :- The parser replaces the handle within a stack with a non-terminal.

(iii) Accept :- The parser announce successful completion of parsing.

(iv) Error :- The parser discovers that a syntax error has occured and cause calls an error recovery routine.

Conflict in SRP :-

→ Shift Reduce conflict

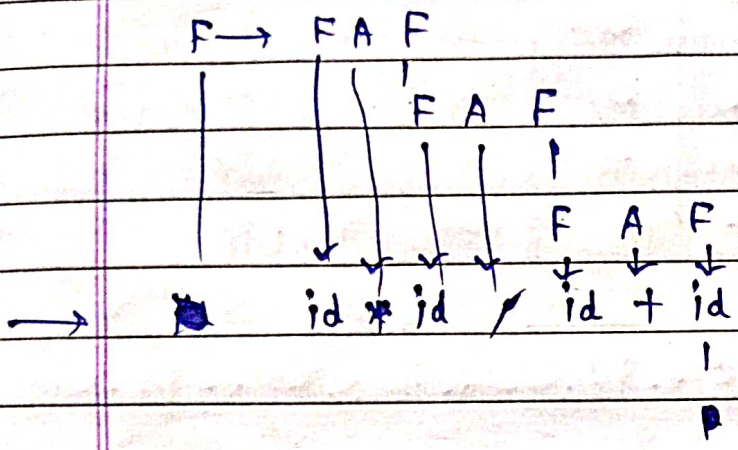
→ Reduce Reduce conflict

G: E → E+E | E*E | (E) | -E | id

Stack	input <<	Action
$	id+id*id $	
$id	+id*id $	shift id
$E	+id*id $	Reduce E→id
$E+	id*id $	shift '+'
$E+id	*id $	shift id
$E+E	*id $	Reduce E→id
$E+E*	id $	shift '*'
$E+E*id	$	shift id
$E+E*E	$	Reduce E→id
$E+E	$	(Here, Reduce Reduce Conflict occurs.) Reduce E→E+E
$E	$	Accept

$F \rightarrow FAF \mid (F) \mid id$ String :- id * id / id + id

$A \rightarrow + / - / * / /$



→ $F \rightarrow F+F \mid F*F \mid F-F \mid F/F \mid (F) \mid id$

F * F
F / F
F + F

F * F / F + F

F * F / F + id

F * F / id + id

F * id / id + id

id * id / id + id

F * id / id + id

F * F / id + id

F * F / F + id

F * F / F + F

F * F / F

F * F

F

H.P.

Operating Precedence Parser

Grammer g is said to be operator precedence if it has following properties:

(1) No production or the right hand side is null.

(2) There should not be any production rule parsing two adjacent non-terminals at the right hand side.

Ex. $E \rightarrow EAE \mid (E) \mid -E \mid id$

$A \rightarrow + \mid - \mid * \mid /$

Precedence Table:

	+	-	*	/	id	(	)	$
+	$\cdot>$	$\cdot>$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	$\cdot>$	$\cdot>$
-	$\cdot>$	$\cdot>$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	$\cdot>$	$\cdot>$
*	$\cdot>$	$\cdot>$	$\cdot>$	$\cdot>$	$<\cdot$	$<\cdot$	$\cdot>$	$\cdot>$
/	$\cdot>$	$\cdot>$	$\cdot>$	$\cdot>$	$<\cdot$	$<\cdot$	$\cdot>$	$\cdot>$
id	$\cdot>$	$\cdot>$	$\cdot>$	$\cdot>$			$\cdot>$	$\cdot>$
(	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	$=$	
)	$\cdot>$	$\cdot>$	$\cdot>$	$\cdot>$			$\cdot>$	$\cdot>$
$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$		

$ has lowest precedence & id has highest precedence

(+, -, *, /) same

2. id + id * id

step 1:

	id	+	*	$
id	0	⋗	⋗	⋗
+	⋖	⋗	⋖	⋗
*	⋖	⋗	⋗	⋗
$	⋖	⋖	⋖	A Accept

→ don't write

step 2: Converted string : $ ⋖ id ⋗ + ⋖ id ⋗ * ⋖ id ⋗ $

step 3: If non-terminal comes shift before

i. E → $E

If LHS ⋖ RHS then shift else reduce.

stack	input	action
$	id + id * id $	$ ⋖ id, shift id
$ id	+ id * id $	id ⋗ +, reduce E→id
$ E	+ id * id $	$ ⋖ +, shift '+'
$ E +	id * id $	+ ⋖ id, shift id
$ E + id	* id $	id ⋗ *, reduce E→id
$ E + E	* id $	+ ⋖ *, shift '*'
$ E + E *	id $	* ⋖ id, shift id
$ E + E * id	$	id ⋗ $, reduce E→id
$ E + E * E	$	* ⋗ $, reduce E→E*E

$ E + E	$	+ > $, reduce E → E + E
$ E	$	Accepted

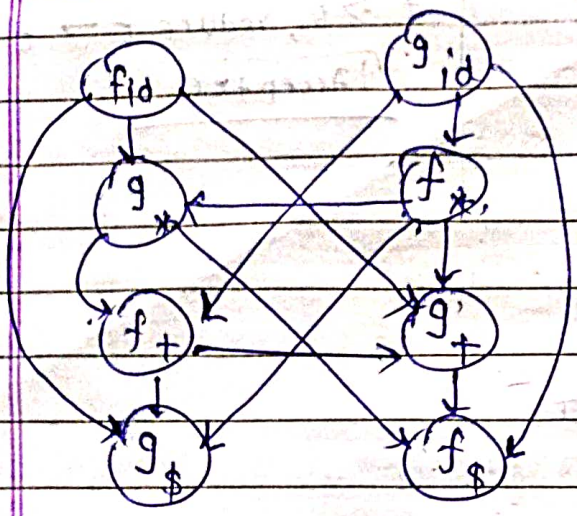
Advantages

- simple to implement

Disadvantage:-

- +, −, etc. such type of unary operators are difficult to handle by operator precedence parser.
- This is applicable for small class of grammers e.g. (E) form previous question.

step-4 Precedence function graph.



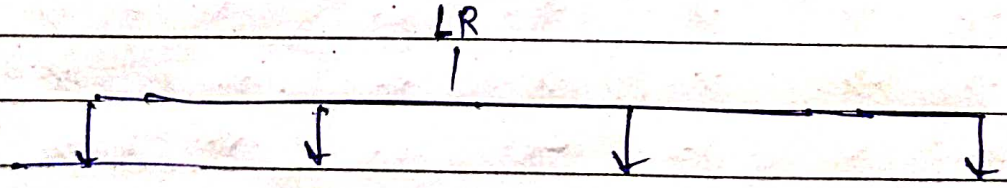
f \ g	id	+	*	$
id		:>	:>	:>
+	<:	:>	<:	:>
*	<:	:>	:>	:>
$	<:	<:	<:	

step-5 function table

	id	*	+	$
f	4	4	2	0
g	5	3	1	0

(Maximum path)

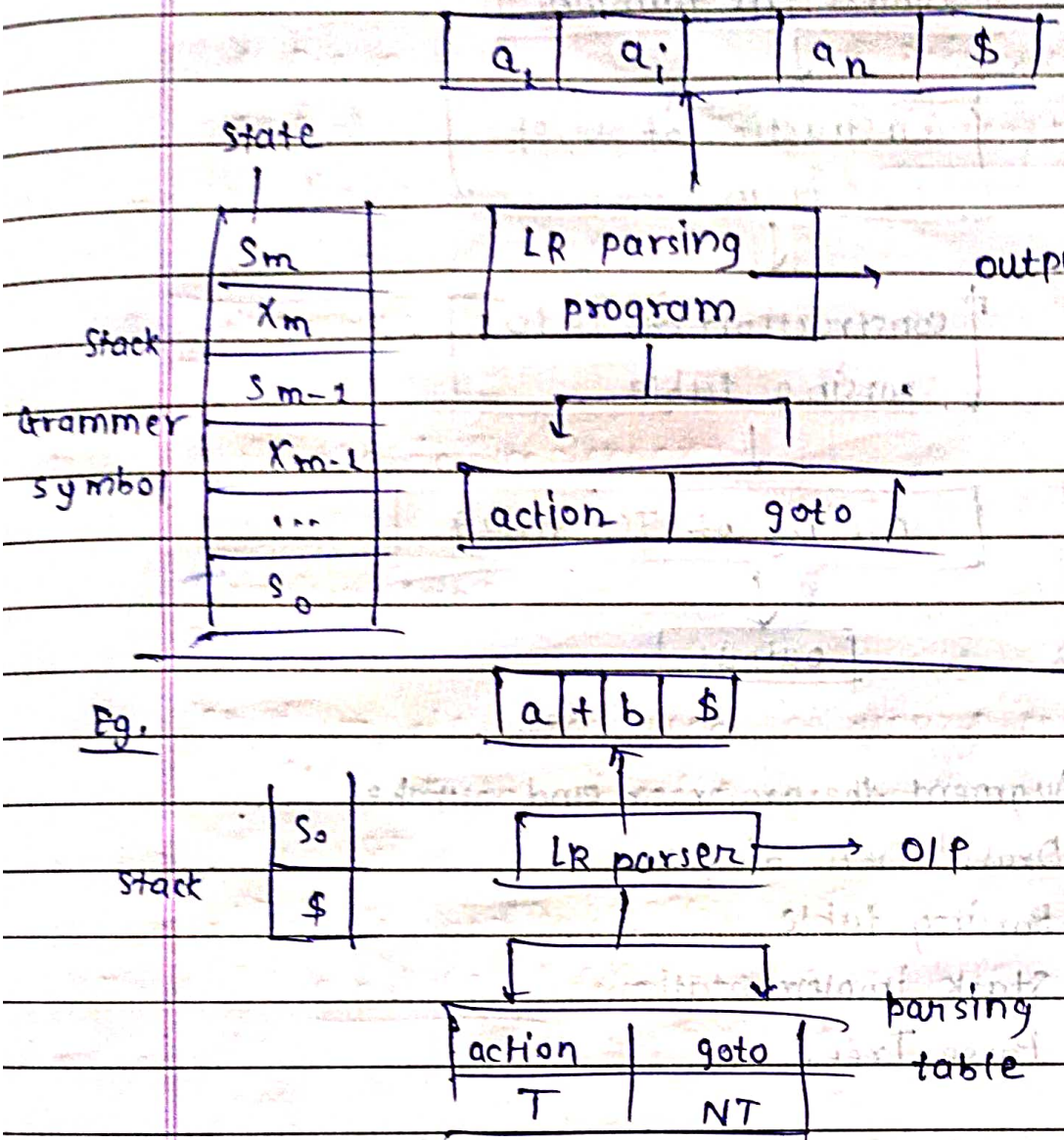
LR parser :-



Model of LR parser :-

L stands for left-to-right scanning.
R stands for right most derivative in reverse order,
K is number of input symbol used to predict the input string
input buffer is used for storing the input streum

(continue...)



Eg.

(continue...) grammer symbol and output.

→ parsing table consist of two parts:-

1. Action
2. Goto

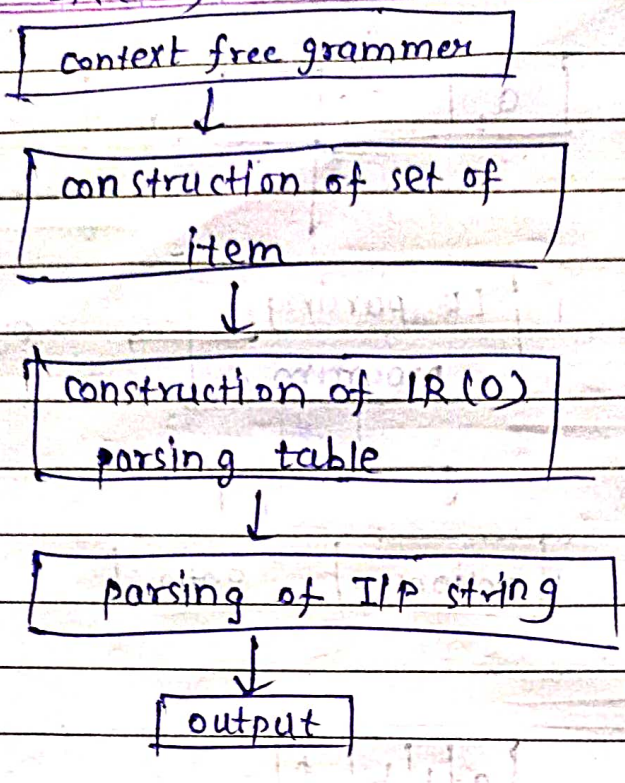
LR(0)

LR(1)

CLR

LALR

LR(0)



step-1: Augment the grammer and number it.
step-2: Draw DFA diagram
step-3: Parsing table
step-4: Stack implementation
step-5: Parse Tree.

① $E \rightarrow BB$

$B \rightarrow cB / d$

string :: ccdd $

Step 1: $E' \rightarrow E \rightarrow 0$

$E \rightarrow BB \rightarrow 1$

$B \rightarrow cB \rightarrow 2$

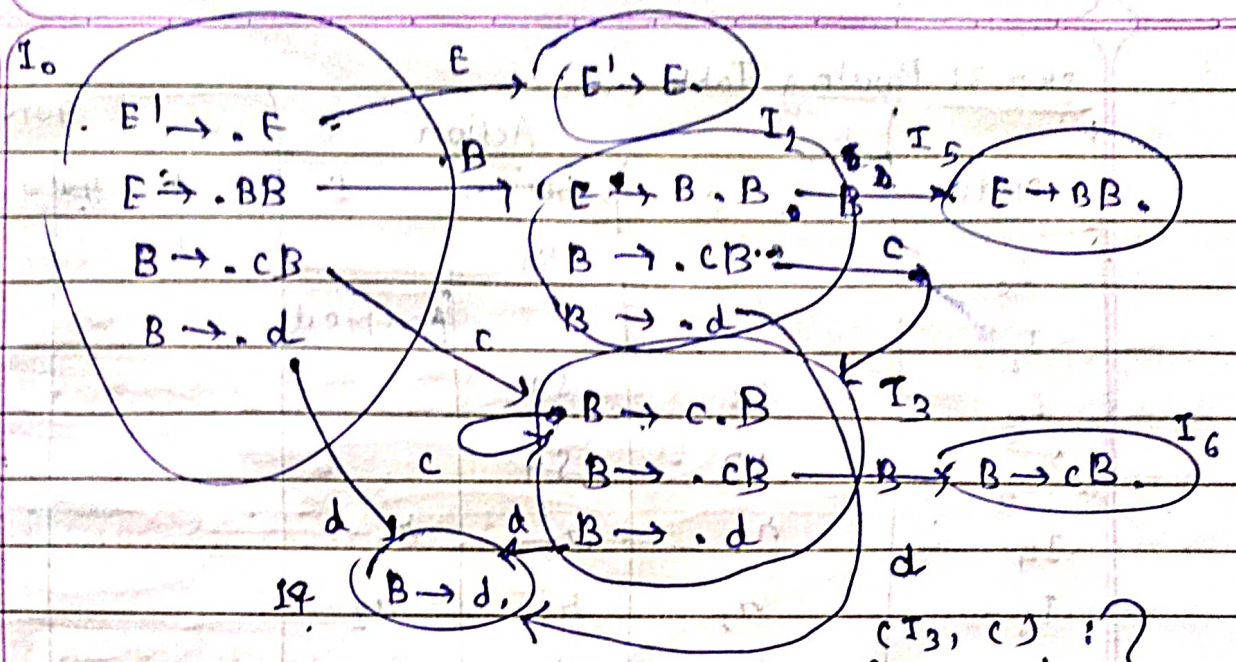
$B \rightarrow d \rightarrow 3$

Step 2: $E' \rightarrow .E \rightarrow 0$

$E \rightarrow .BB \rightarrow 1$

$B \rightarrow .cB \rightarrow 2$

$B \rightarrow .d \rightarrow 3$



I_1

Date
Page

I = closure { $E' \to .E$ }

I_0 :

{

$E' \to .E$

$E \to .BB$

$B \to .cB$

$B \to .d$

}

I_1 = goto (I_0, E) :

{

$E' \to E.$

}

I_2 = goto (I_0, B) :

{

$E \to B.B$

$B \to .cB$

$B \to .d$

}

I_3 = goto (I_0, c) :

{

$B \to c.B$

$B \to .cB$

$B \to .d$

}

I_4 = goto (I_0, d) :

{

$B \to d.$

}

I_5 = goto (I_2, B) :

{

$E \to BB.$

}

I_6 = goto (I_3, B) :

{

$B \to cB.$

}

(I_3, c) :
(I_2, c) :

Write down to 3 times

Step 3: Parsing Table

State	Action			goto	
	c	d	$	E	B
I_0	s3	s4		1	2
I_1			Accepted		
I_2	s3	s4			5
I_3	s3	s4			6
I_4	r_3	r_3	r_3		
I_5	r_1	r_1	r_1		
I_6	r_2	r_2	r_2		

Reduce (r) shift (s)

Step 4: Stack Implementation

Stack	Input	action
$ 0	ccdd$	shift c → s_3
$ 0 C_3	cdd$	shift c → s_3
$ 0 $C_3 C_3$	dd$	shift d → s_4
$ 0 $C_3 C_3 d_4$	d$	Reduce B → d
$ 0 $C_3 C_3 B_6$	d$	Reduce B → cB
$ 0 $C_3 B_6$	d$	Reduce B → cB
$ 0 B_2	d$	shift d → s_4
$ 0 $B_2^2 d_4$	$	Reduce B → d
$ 0 $B_2 B_5$	$	Reduce E → BB
$ 0 E 1	$	Accepted

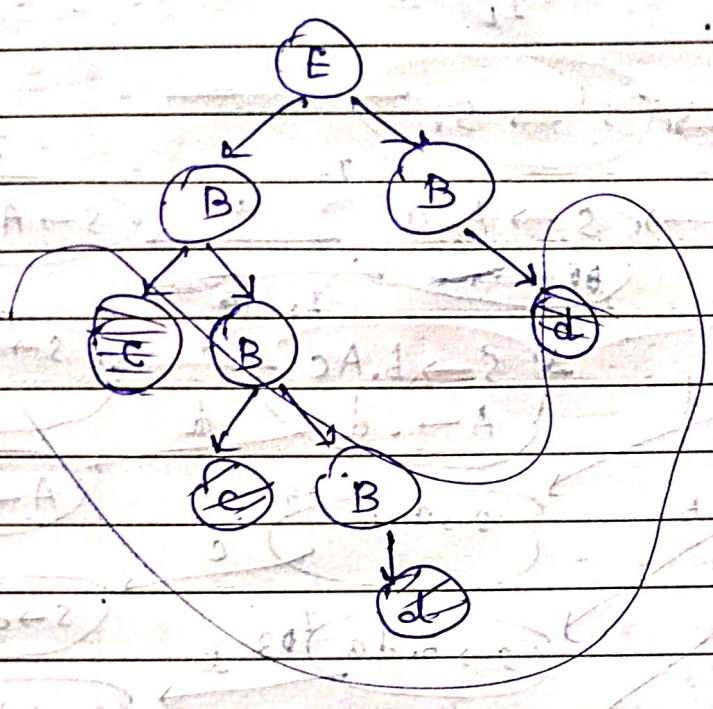
$I_3 \to B \to 6$

$I_0 \to B \to 2$

$I_2 \to B \to 5$

$I_0 \to E$

Step 5 :- Parse Tree



(*) S → Aa | bAc | dc | bda

A → d

Step 1 :- S' → S → 0

S → Aa → 1

S → bAc → 2

S → dc → 3

S → bda → 4

A → d 5

Step 2 :- S' → . S → 0

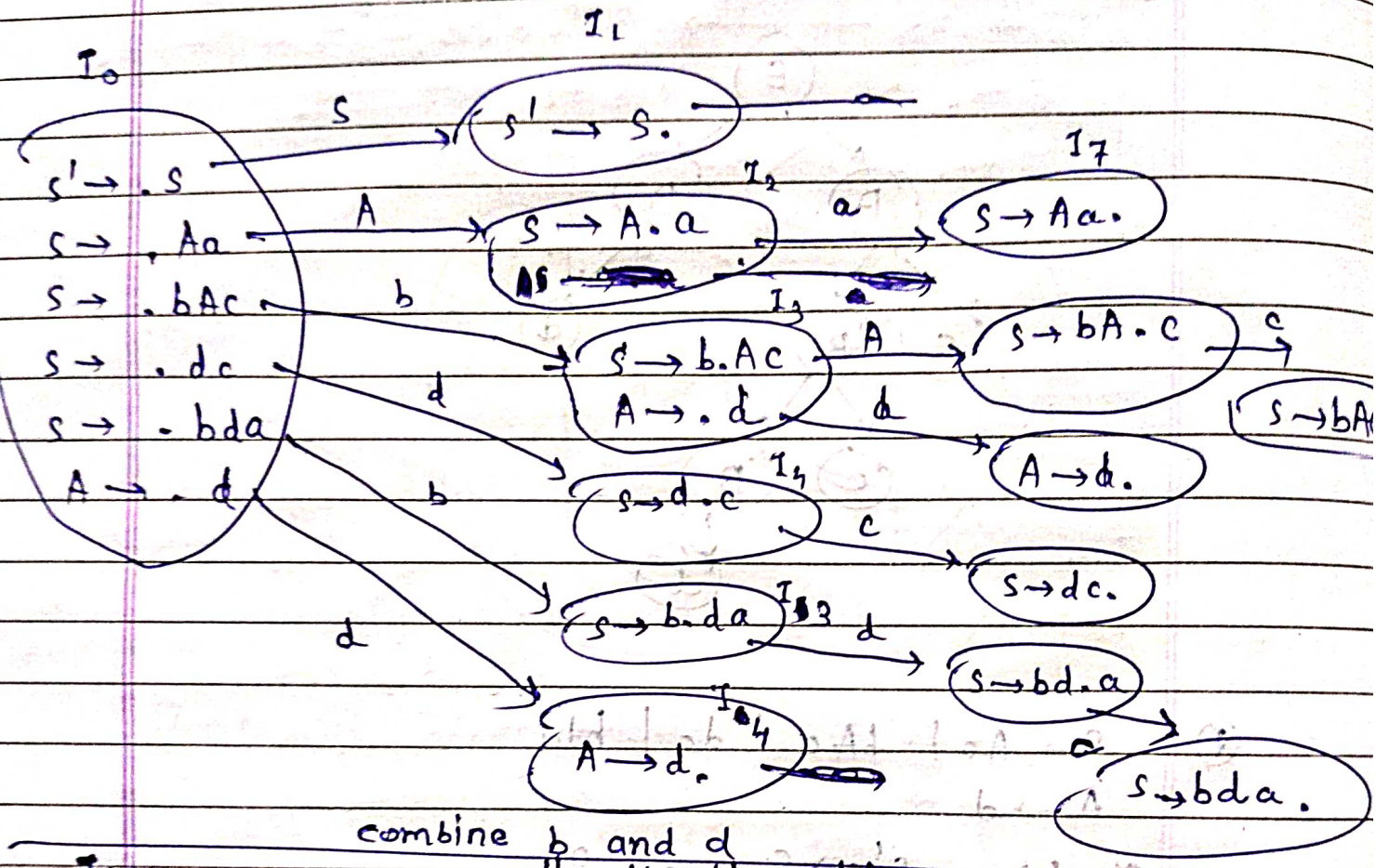
S → . Aa → 1

S → . bAc → 2

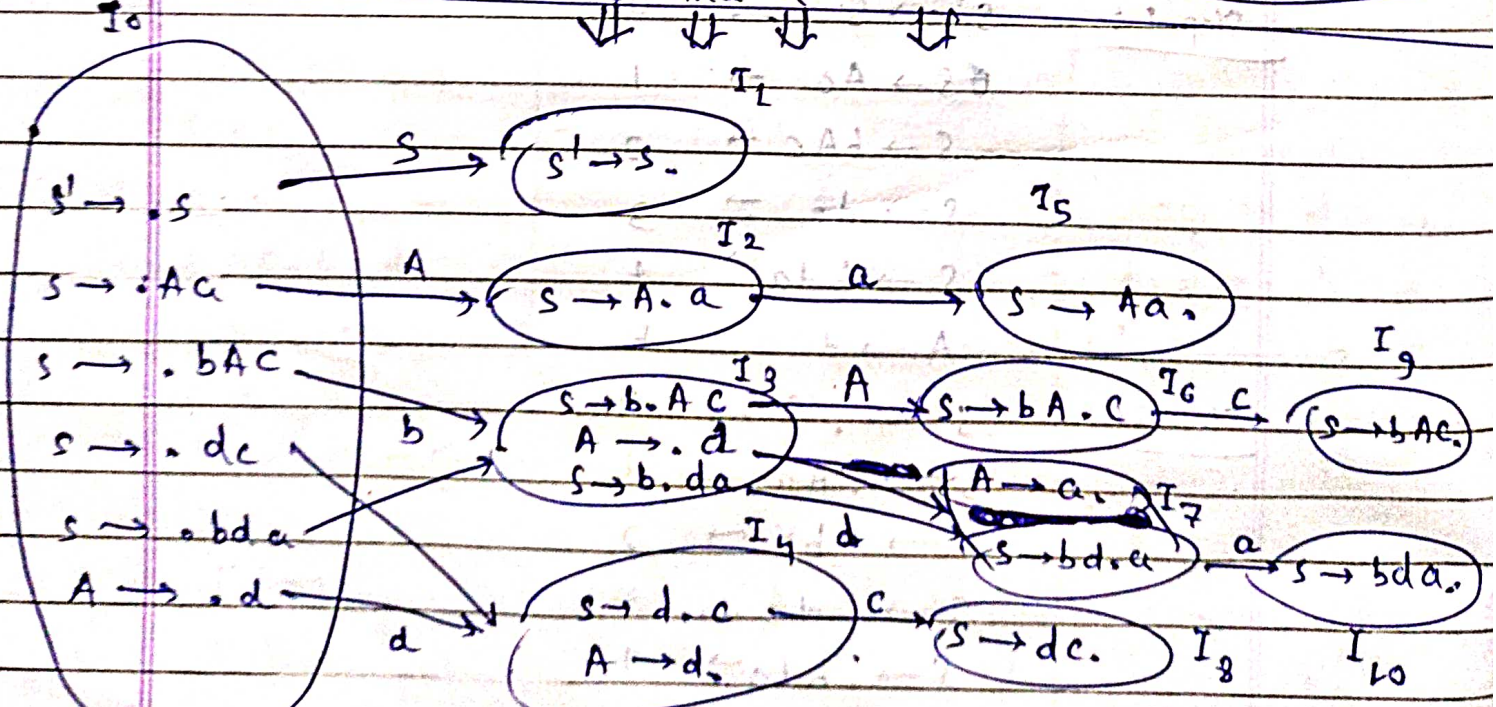
S → . dc → 3

S → . bda → 4

A → . d → 5

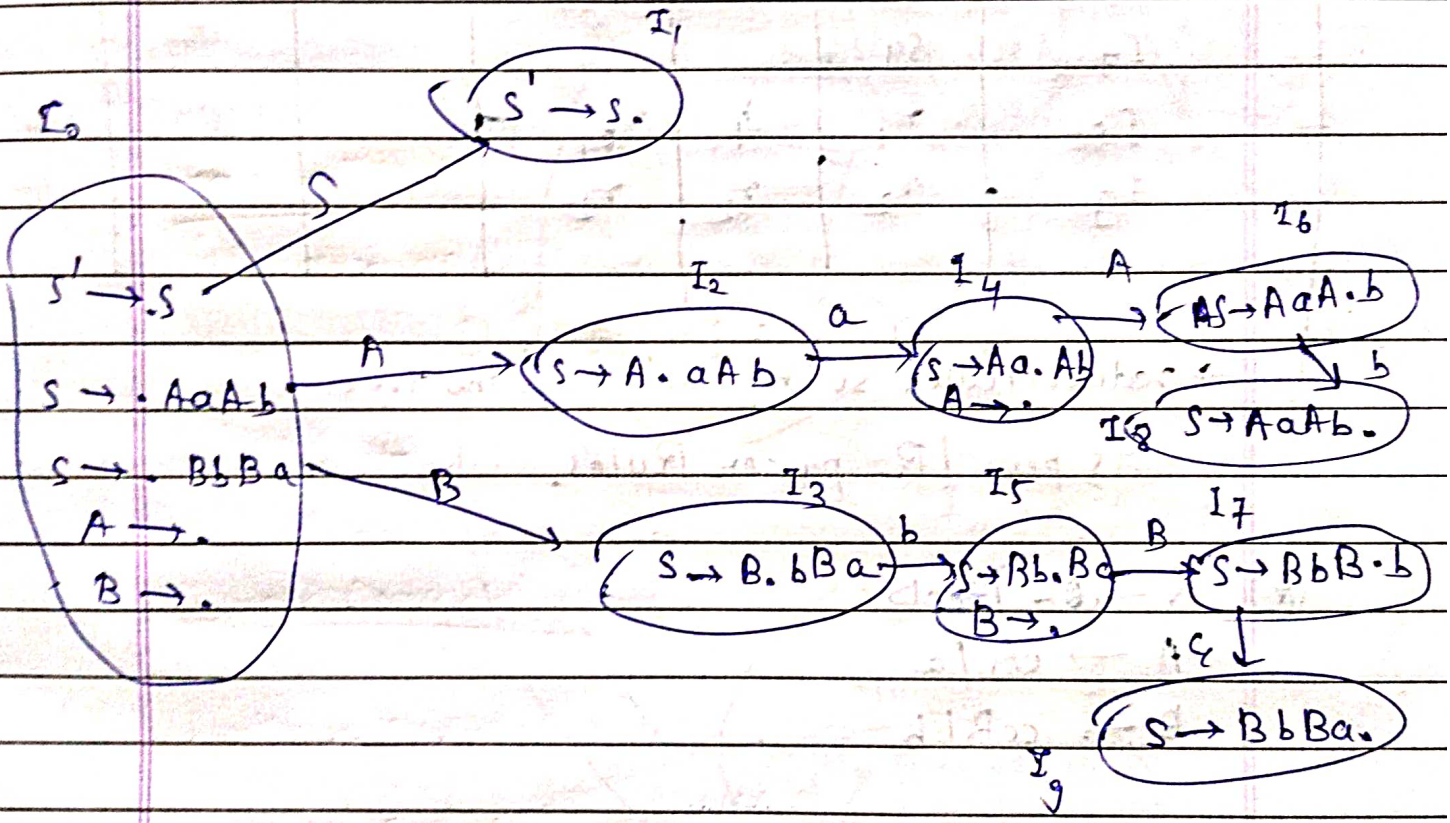


combine b and d



Parsing table

State	Action					goto	
	a	b	c	d	$	S	A
I_0		s_3		s_4		1	2
I_1							
I_2	s_5						
I_3			s_7				6
I_4	r_5	r_5	s_7/r_5	r_5	r_5		
I_5	r_1	r_1	r_1	r_1	r_1		
I_6			s_9				
I_7	r_5	r_5	r_5	r_5	r_5		
I_8	r_3	r_3	r_3	r_3	r_3		
I_9	r_2	r_2	r_2	r_2	r_2		
I_{10}	r_4	r_4	r_4	r_4	r_4		


I_0
$S' \to .S$
$S \to .AaAb$
$S \to .BbBa$
$A \to .$
$B \to .$
S
I_1
$S' \to S.$
A
I_2
$S \to A.aAb$
B
I_3
$S \to B.bBa$
a
I_4
$S \to Aa.Ab$
$A \to .$
b
I_5
$S \to Bb.Ba$
$B \to .$
A
I_6
$S \to AaA.b$
B
I_7
$S \to BbB.b$
b
I_8
$S \to AaAb.$
b
I_9
$S \to BbBa.$

	Action			Goto		
	a	b	$	S	A	B
→ I_0	s_3/r_4	r_3/r_4	r_3/r_4	1	2	3
I_1			Accept			
I_2	s_4					
I_3	[illegible]	s_5			6	
I_4	r_3	r_3	r_3			
I_5	r_1	r_4	r_4			7
I_6		s_8				
I_7	s_9					
I_8	r_1	r_1	r_1			
I_9	r_2	r_2	r_2			

→ reduce/redue so we can't pass the tree.

As per LR parser Rules.

(*) S → cA / ccB

A → cA / a

B → ccB / b

Step 1 :- 0 S' → S

1 S → cA

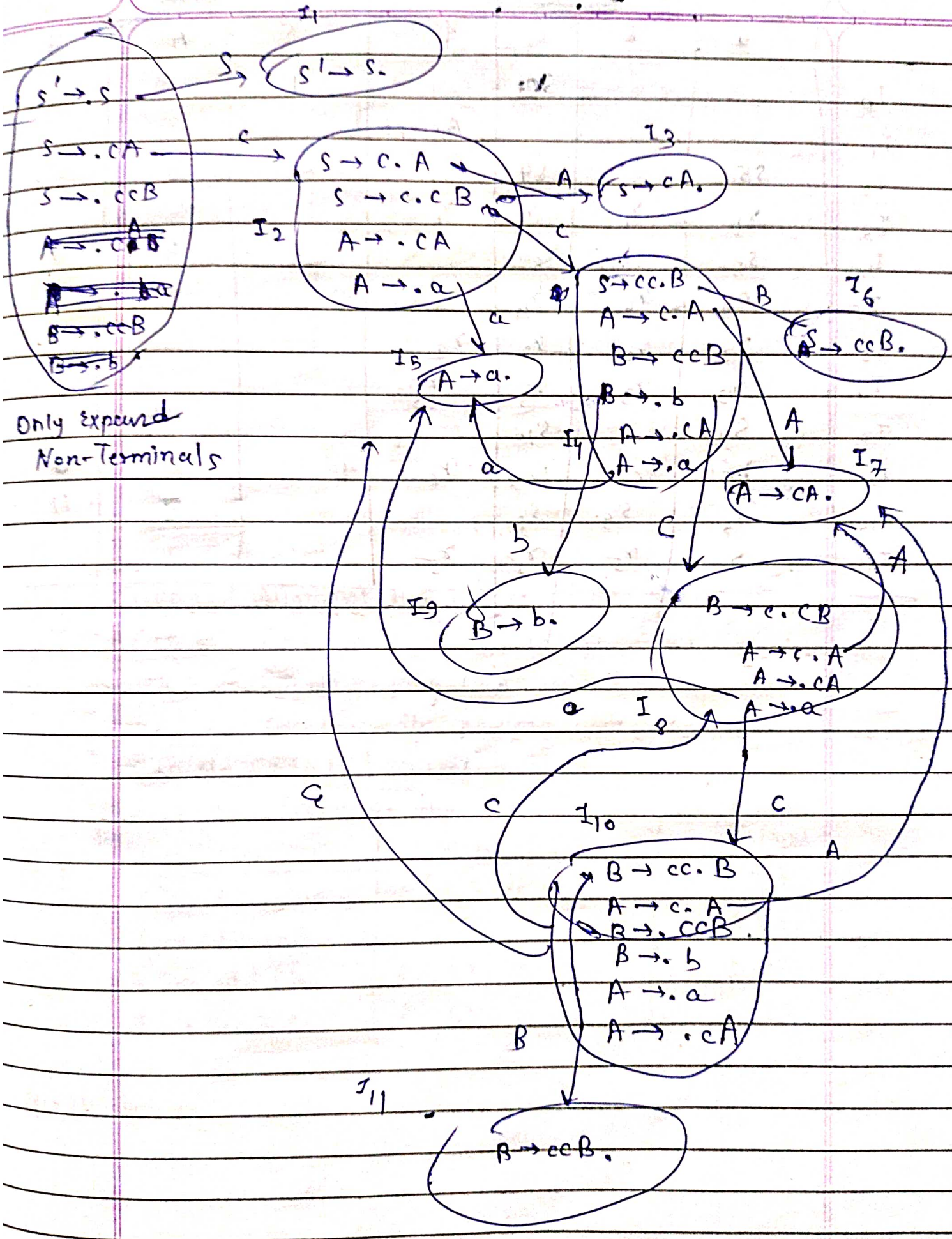
2 S → ccB

3 A → cA

4 A → a

5 B → ccB

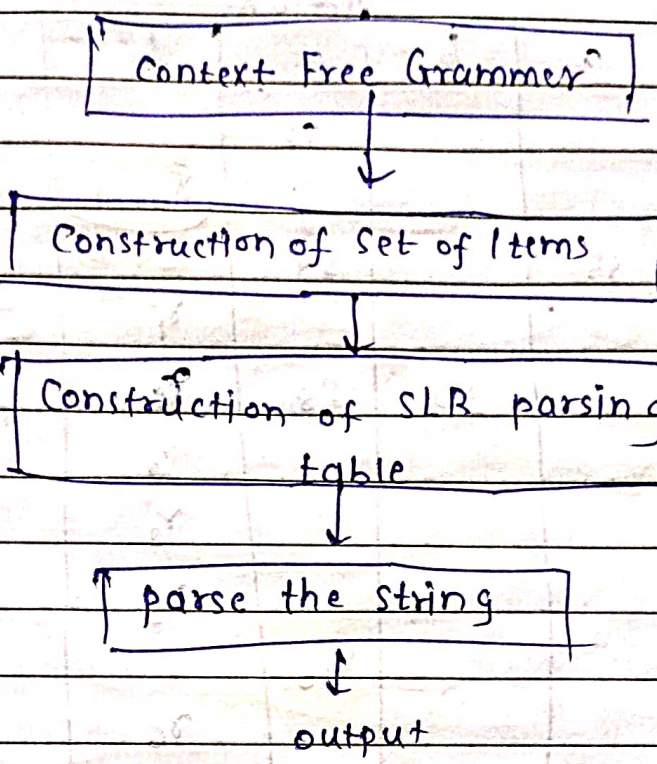
6 B → b


I1
S'→ S.
S'→ .S
S → .CA
S → .CcB
A → .Cα
A → .b
B → .CcB
B → .b
S
C
I2
S → C. A
S → C. c B
A → . CA
A → . a
A
I3
S → CA.
c
S → Cc. B
A → C. A
B → CcB
B → . b
A → . CA
A → . a
I4
B
I6
S → CcB.
a
I5
A → a.
Only expand Non-Terminals
A
I7
A → CA.
b
C
A
I9
B → b.
B → C. CB
A → C. A
A → . CA
A → . a
I8
a
C
C
I10
A
B → Cc. B
A → C. A
B → . CcB
B → . b
A → . a
A → . cA
B
I11
B → CcB.

	Action				Goto		
	a	b	c	$	S	A	B
I_0			S2		1		
I_1				Accept			
I_2	S5		S4			3	
I_3	r_1	r_1	r_1	r_1			
I_4	S_5	S_7	S_8			7	6
I_5	r_4	r_4	r_4	r_4			
I_6	r_2	r_2	r_2	r_2			
I_7	r_3	r_3	r_3	r_3			
I_8	S_5		S_{10}			7	
I_9	r_6	r_6	r_6	r_6			10
I_{10}	S_5	S_9	S_8			7	11
I_{11}	r_5	r_5	r_5	r_5			

SLR (simple LR-parser)

Date
Page



step 1:- Augment the grammer and number it.
step 2:- Find the first and follow.
step 3:- Construct the DFA.
step 4:- Construct the parsing table.
step 5:- Stack Implementation,
step 6:- Generate the parse Tree.

Step 1:- done

Step 2:-

	first	follow
$S' \rightarrow S$	{c}	{$}
$S \rightarrow cA \mid ccB$	{c}	{$}
$A \rightarrow cA \mid a$	{c, a}	{$}
$B \rightarrow ccB \mid b$	{c, b}	{$}

Step 3:- done

Step 4:-

	Action				Goto		
	a	b	c	$	S	A	b
I_0			S2		1		
I_1				Accept			
I_2	S5		S4			3	
I_3				r_1			
I_4	S5	S9	S8			7	6
I_5				r_4			
I_6				r_2			
I_7				r_3			
I_8			S10			7	
I_9				r_6			
I_{10}			S8			7	11
I_{11}				r_5			

Note:- We have to entry for reduce only in follow of particular state, of production.

String ccccb$

(1) $S \to cAd$

$A \to ab \mid e$

Step 1

0 $S' \to S$

1 $S \to cAd$

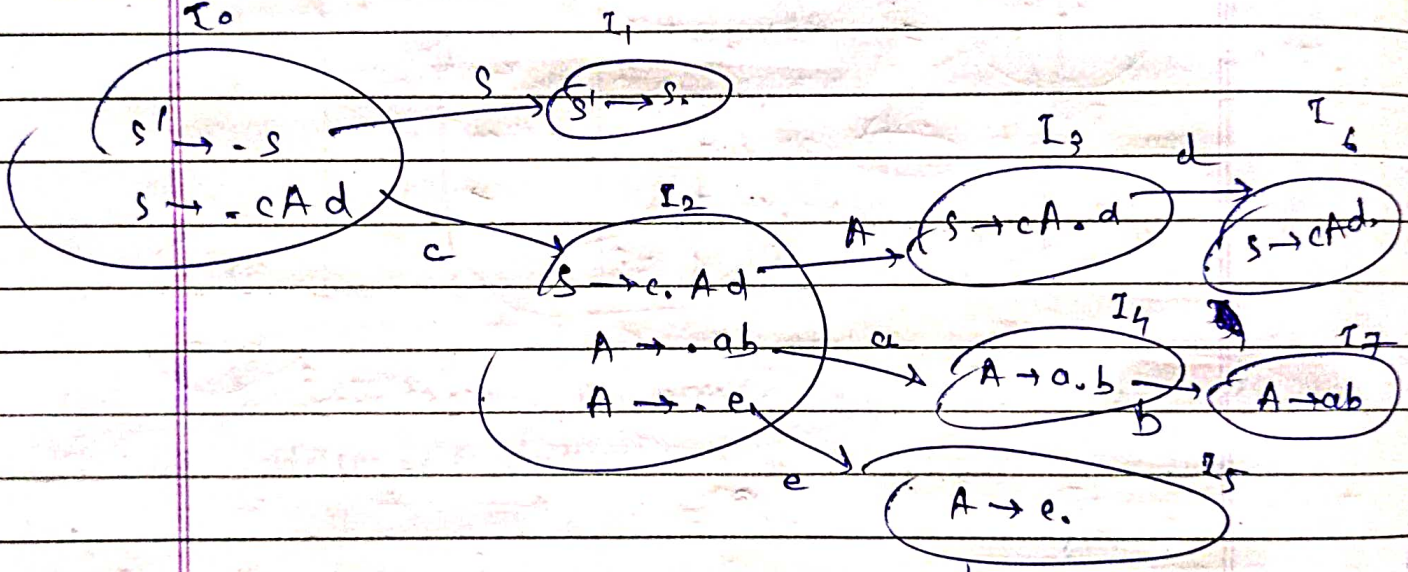
2 $A \to ab$

3 $A \to e$

Step 2 :

	first	follow
$S' \to S$	$\{c\}$	$\{\$\}$
$S \to cAd$	$\{c\}$	$\{d\}$
$A \to ab \mid e$	$\{a, e\}$	$\{\$\}$

Step 3 :



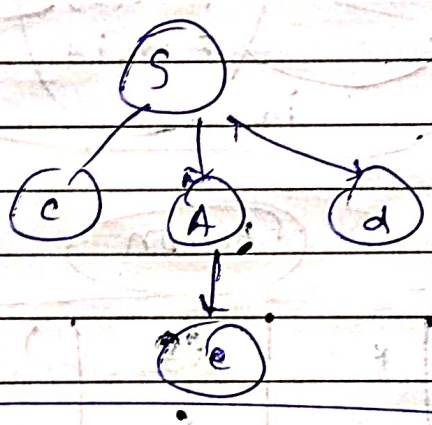
Step 4 :

	Action						goto	
	a	b	c	d	e	$	S	A
I_0			S_2				1	
I_1						Accept		
I_2	S_4				S_5			3
I_3				S_6				
I_4		S_7						
I_5				r_3				
I_6						r_1		
I_7				r_2				

Step 5 ced string

Stack	Input	Action
$ 0	ced $	Shift $c \to s_2$
$ 0 c_2	ed $	Shift $e \to s_5$
$ 0 $c_2 e_5$	d $	reduce $A \to e$
$ 0 $c_2 A_3$	d $	Shift $d \to s_6$
$ 0 $c_2 A_3 d_6$	$	reduce $S \to cAd$
$ 0 S_1	$	Accepted

Step 6



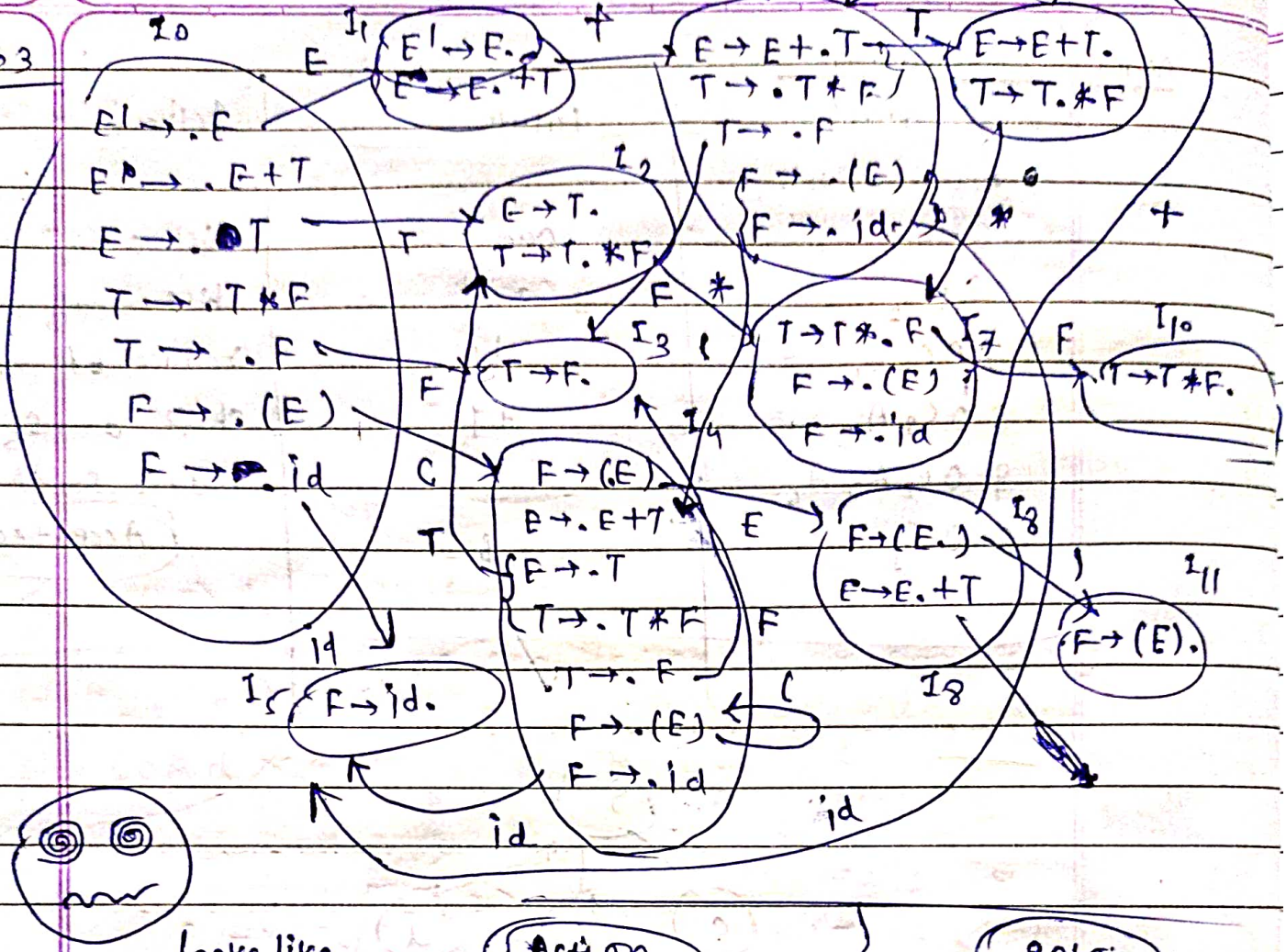
(*) $E' \to E$ (1) $E \to E + T \mid T$ (2) (3) $T \to T * F \mid F$ (4) $F \to (E) \mid id$ (5) (6)

Step 1 ←

Step 2

	first	follow
E'	{ c, id }	{ $ }
E	{ c, id }	{ +, $,) }
T	{ c, id }	{ +, $,), * }
F	{ c, id }	{ +, $,), * }

Step 3



Looks like

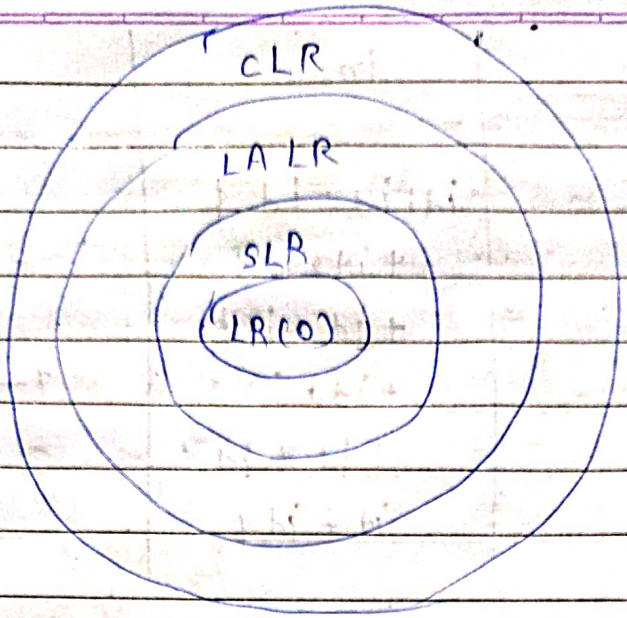
Step 4

	Action						Goto		
	id	+	*	(	)	$	E	T	F
I_0	S_5			S_4			1	2	3
I_1		S_6				Accept			
I_2		r_2	S_7		r_2	r_2			
I_3		r_4	r_4		r_4	r_4			
I_4	S_5			S_4			8	2	3
I_5		r_6	r_6		r_6	r_6			
I_6	S_5			S_4				9	3
→ I_7	S_5			S_4					10
I_8		S_6			S_{11}				
I_9		r_1	S_7		r_1	r_1			
I_{10}		r_3	r_3		r_3	r_3			
I_{11}		r_5	r_5		r_5	r_5			

Date
Page

E 'id + id * id

stack	Input	Action
$0	id+id*id $	
$ 0 'id 5	+id*id$	E → 'id
$ 0 F 3	+id*id$	
$ 0 T 2	+id*id$	T → F
$ 0 E 1	+id*id$	
$ 0 E 1 +	id*id$	



SLR ⇒ 1 LookAhead

Because we are finding follow of string

so, SLR(1)
LR(1) } it is called or known

$E \to (E) \mid T$

$E' \to .E, \$$

$E \to .(E), \$ \longrightarrow E \to (.E), \$$

$E \to .T, \$$

$E \to .(E),)$

$E \to .T,)$

LALR

CLR
. $/)

4. Syntex Directed Translation

Date
Page

Syntex Direction Defination :-
(SDD) = CFG + Semantics rules

→ Attributeds are accoiated with grammer symbols
→ Symentic rules are accociated with production.
→ Attributes may be numbers, strings, references, data types,.

X.a = value at node X. X, Y → symbol
Y.b = value at node Y. a, b → Attribute

Table 4.1

Production	Sementic Rules
Ans L → En	print (E.val) →
E → E1 + T	E.val = E1.val + T.val
E → T	E.val = T.val
T → T1 * F	T.val = T1.val X F.val
T → F	T.val = F.val
F → (E)	F.val := E.val
F → digit	F.val := digit.lexval

Q. Syntex Directed Defination of simple desk calculation
or
Q. SDD evalution of given expresion.
or
Q. Annotated parse tree for given expression.

→

Note:- • The token digit has a synthesized attribute lexval whose value is assumed to be supplied by lexical analyser.
• The Rule accociated with the production L → En for the Starting non-terminal L is just a procedure that prints as a output value of the arithmetic expression generated by E.

Types Of Attributes :-

1) Synthesized Attribute → (It will takes value from it's child)
2) Inherited Attribute →
3) If a node will takes value from it's Parent or it's sibblings, then it is called as Inherited Attribute

Ex. $W \rightarrow XYZ$

$Y.i := W.i$

~~$W.i := Y.i$~~

$Y.i = Z.i$

$Y.i = X.i$

$W \rightarrow XYZ$

Ex. $W.S := X.S$

$W.S := Y.S$

$W.S := Z.S$

~~$Y.S := W.S$~~

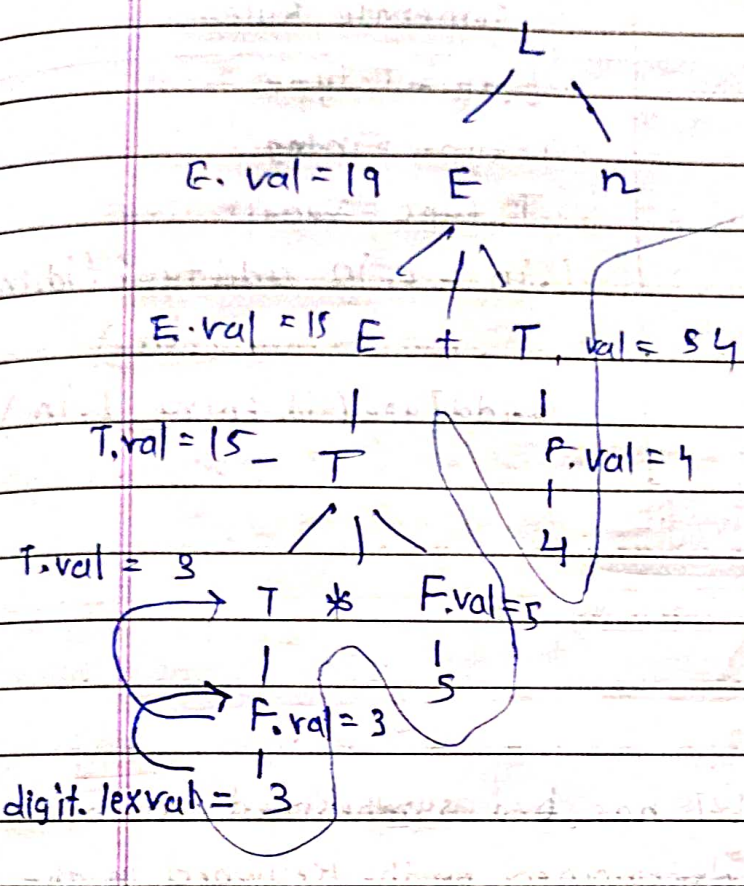
$X, Y, Z, W \rightarrow$ nodes

$S \rightarrow$ Synthesized Attribute

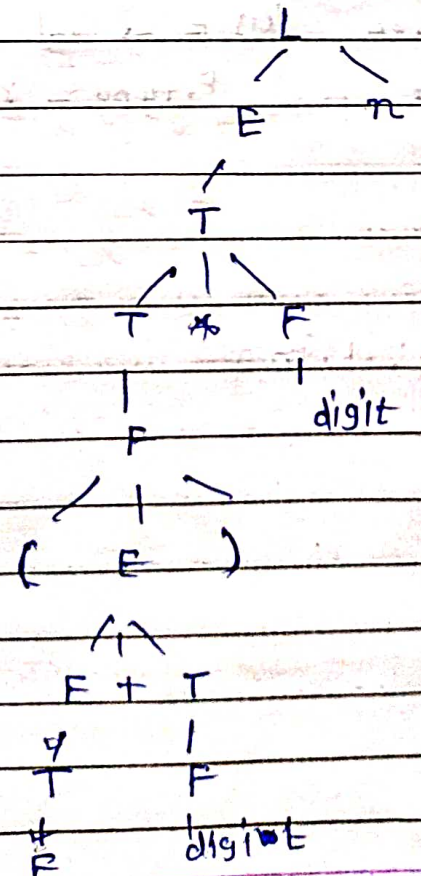
Synthesized Attribute

Parse Tree

Expression :- 3 * 5 + 4n (using Table 4.1)



(*) (4 * 7 + 1) * 2



Inherited Attribute

Production	Semantic Rules
$D \rightarrow TL$	$L.in = T.type$
$T \rightarrow int$	$T.type = integer$
$T \rightarrow real$	$T.type = real$
$L \rightarrow L, id$	$L.in = L.in$ addtype (id.entry, L.in)
$L \rightarrow id$	addType (id.entry, L.in)

Note :- In the nonterminal T has a synthesized attribute type, whose value is determined by the keyword in the declaration then the symentic rule will be

Rules :- Eg. keyword is Integer.

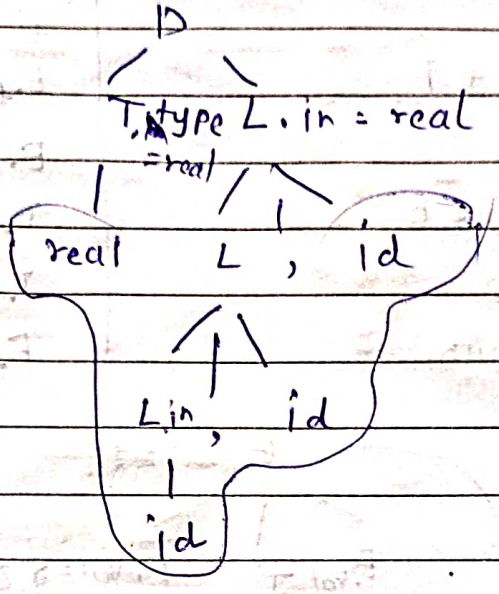
T.type = integer

(1) $F \rightarrow real$
F.type = real

(2) $D \rightarrow TL$
L.In := T.type

(3) $L \rightarrow id$
addType (id.entry; L.in)

Eg. real id, id, id



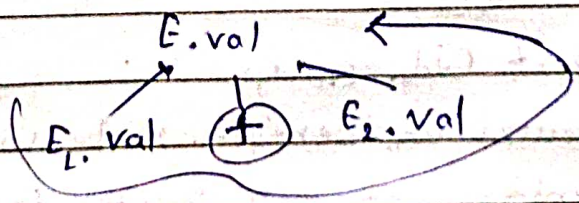
Dependency Graph :-

advantages :-

flow of the information among the Attributes in a parse tree it determines.

Determines the evaluation model, while we make the annotated parse tree it shows the value of attributes which is ~~determined~~ computed by dependency graph. ~~Dependency Graph~~.

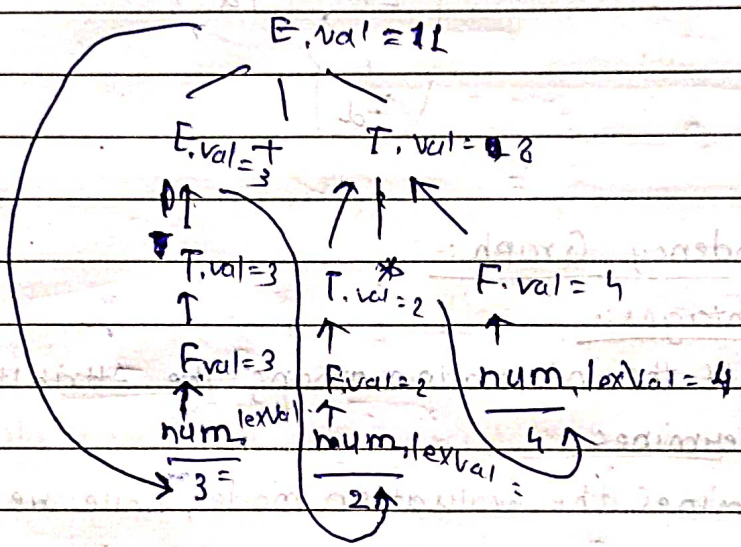
Production	Sementic rule
$E \rightarrow E_1 + E_2$	$E.val = E_1.val + E_2.val$



Top-Down Approach & left to Right

3 + 2 * 4

Production	Semantic rule
$E \rightarrow E + T$	E.val = E.val + T.val
$E \rightarrow T$	E.val = T.val
$T \rightarrow T * F$	T.val = T.val * F.val
$T \rightarrow F$	T.val = F.val
$F \rightarrow num$	F.val = num.lexVal



Construction of Syntax Tree :-

(1) mknode (op, left, right)
It creates an operator node with label op and two fields containing pointers to left and right.
Eg.

(2) mkleaf (id, entry)
It creates an identifier node with label id and a field containing entry, a pointer to the symbol table entry for the identifier.

Eg. p_1 = mkleaf (id, entry - x)

(3) mkleaf (num, val)

Creates a number node with label num and a field containing val, the value of the number.

Eg. 7

P_2 = mkleaf (num, 7)

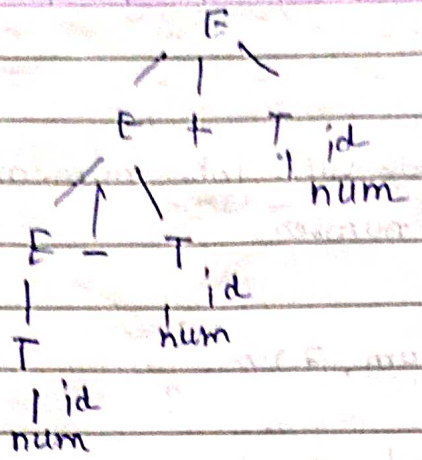
Eg. x+7 mknode ('+', P_1, P_2)

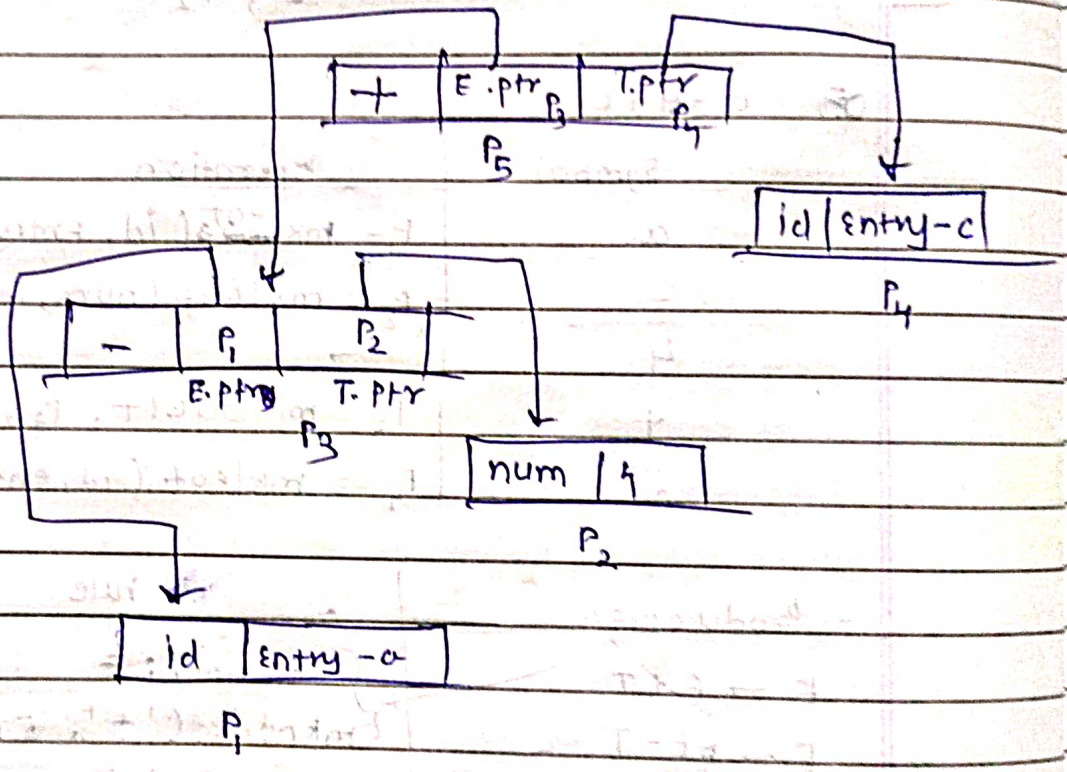
(4) a-4+c

Symbol	operation
a	P_1 = mkleaf (id, entry-a);
-	P_2 = mkleaf (num, 4); → P_3 = mknode(-, P_1, P_2)
4	
+	P_5 = mknode (+, P_3, P_4);
c	P_4 = mkleaf (id, entry-c);

Production	rule
E → E+T	E.nptr := mknode ('+', E.nptr, T.nptr)
E → E-T	E.nptr = mknode ('-', E.nptr, T.nptr)
E → T	E.nptr = T.nptr
T → num	T.nptr = mkleaf (num, num.val)
T → id	T.nptr = mkleaf (id, entry-id)

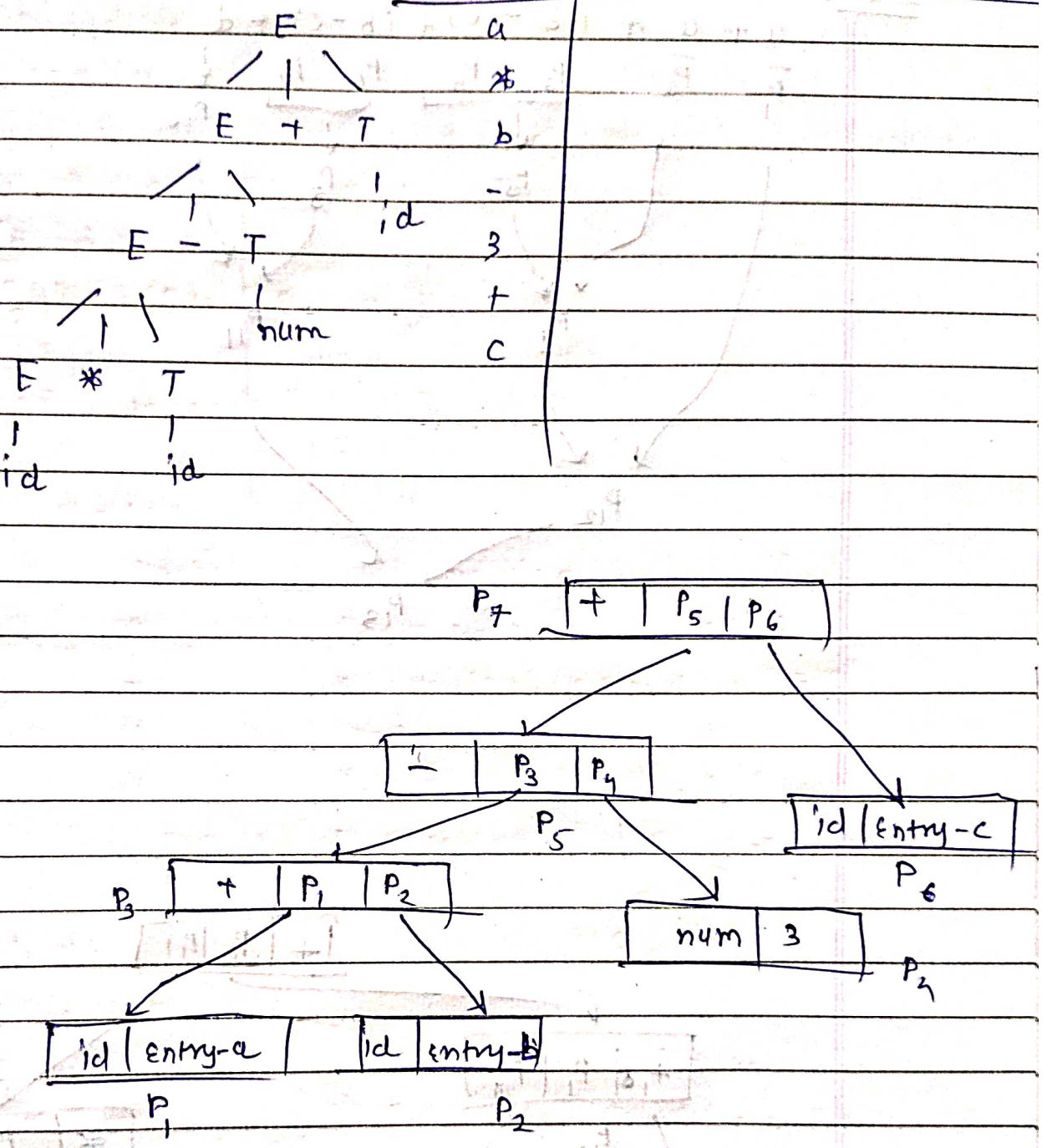
Eg. a - 4 + c





Eg. a + b - 3 + c

Productions	
$E \to E + T$	E.nptr = mknode(+, E.nptr, T.nptr)
$E \to E - T$	" - "
$E \to E * T$	" * "
$E \to T$	E.nptr = T.nptr
$T \to num$	T.nptr = mkleaf(num, num.val)
$T \to id$	T.nptr = mkleaf(id, entry-id)


Symbol
Operation
a
*
b
-
3
+
c
E
E + T
E - T
E * T
id
num
id
id

P7
+ | P5 | P6
- | P3 | P4
P5
id | entry-c
P6
P3
+ | P1 | P2
num | 3
P4
id | entry-a
P1
id | entry-b
P2

Eg.

$a + a * (b - c) + (b - c) * d$

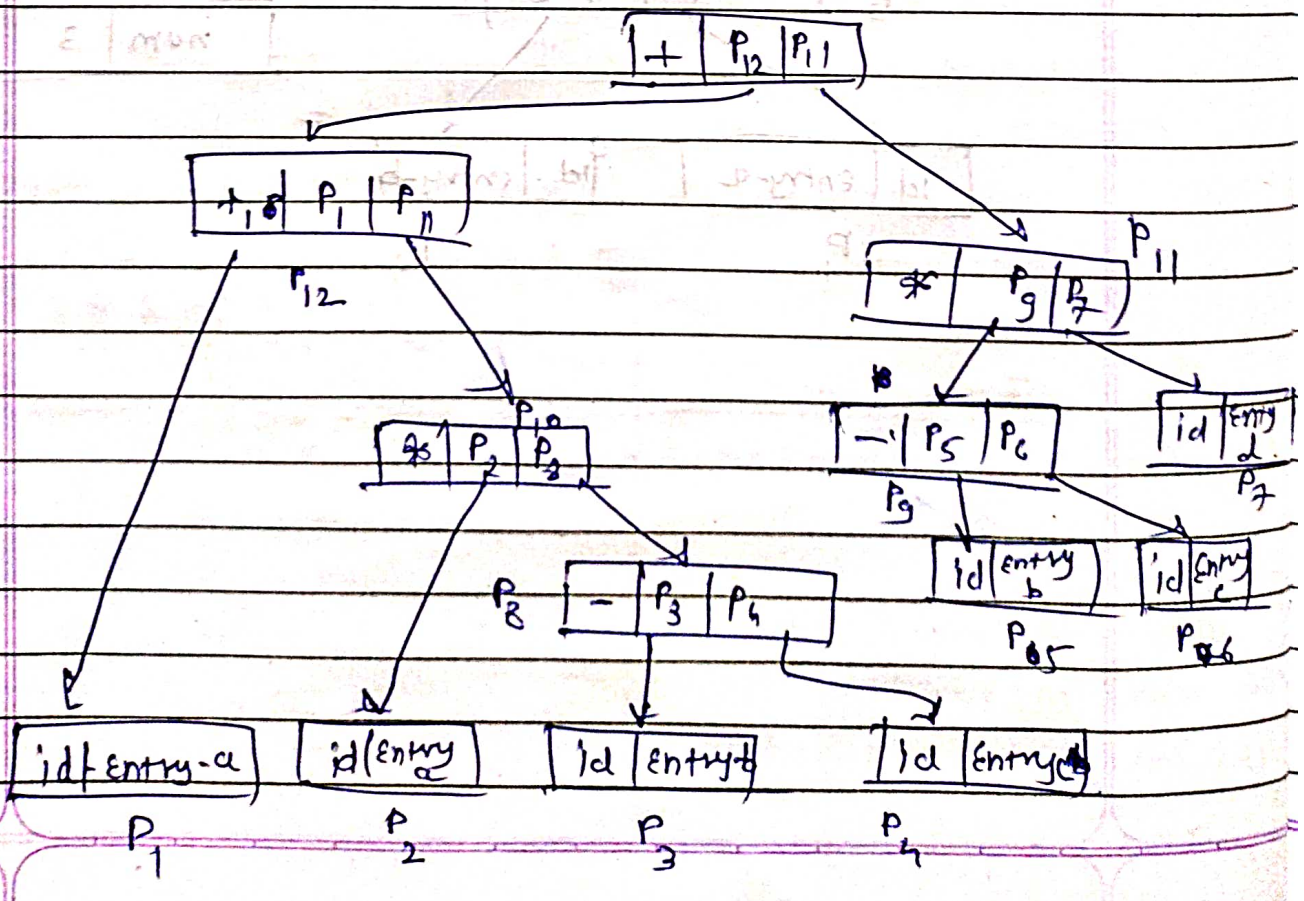
$\overline{P_1}$ $\overline{P_2}$ P_3 P_4 P_6 P_7 P_{10}

P_5 P_8

P_9 P_{11}

P_{12}

P_{13}



Productions

$E \rightarrow E + T$

$E \rightarrow E - T$

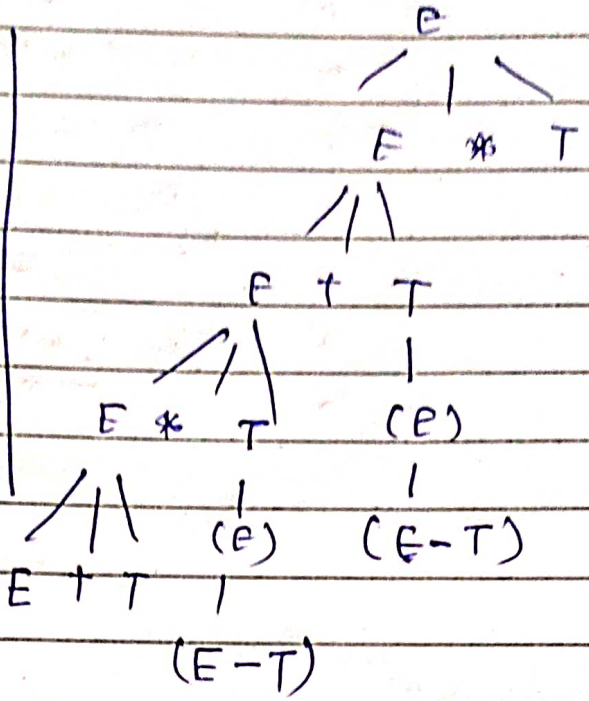
$E \rightarrow E * T$

$E \rightarrow T$

$T \rightarrow (E)$

$T \rightarrow num$

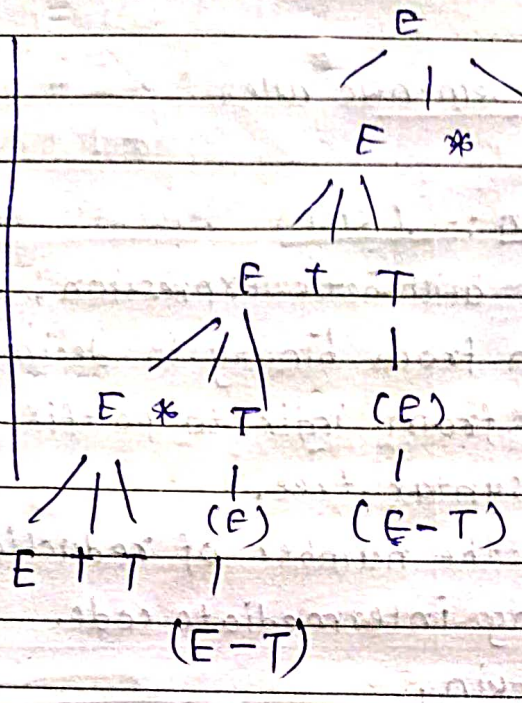
$T \rightarrow id.$



pdf contd... ↗
part (2)

Date
Page

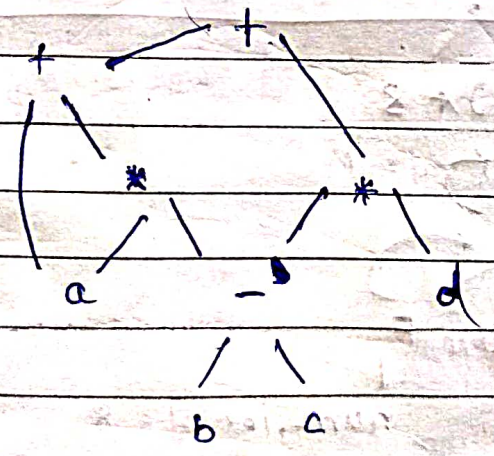
Productions
$E \to E + T$
$E \to E - T$
$E \to E * T$
$E \to T$
$T \to (E)$
$T \to num$
$T \to id$



Directed Acyclic Graph for Expressions:- (DAG)

Leaf node:- It may be an identifier or a variable
INTERNAL node (parent node) :- operators

a + a * (b - c) + (b - c) * d



Syntex Directed Translation (SDT)

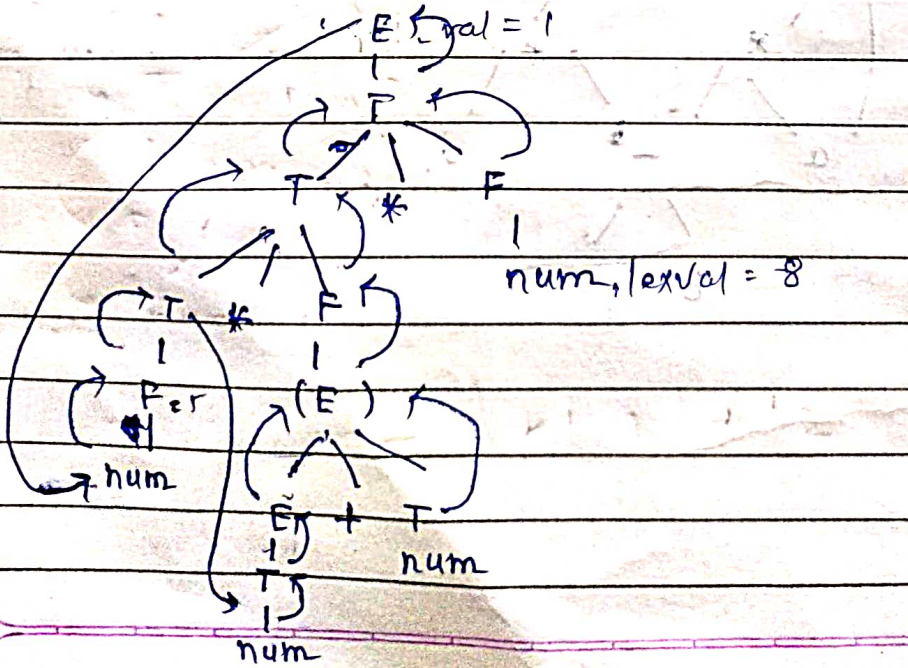
CFG + sementic rules

Application :- (SDT)

- Executing arithmetic expression,
- Conversion from binary to decimal.
- coversion from infix to postfix or prefix.
- Creating syntax tree.
- Counting the number of reductions.
- Generating intermediate code.
- Type checking

	grammer	rules
⊛	$E \to E+T$	{ E.val := E.val + T.val }
	$E \to T$	{ E.val := T.val }
	$T \to T*F$	{ T.val := T.val X F.val }
	$T \to F$	{ T.val := F.val }
	$F \to (E)$	{ F.val := E.val }
	$F \to num$	{ F.val := num.lexval }

String : 5 * (6 + 7) * 8



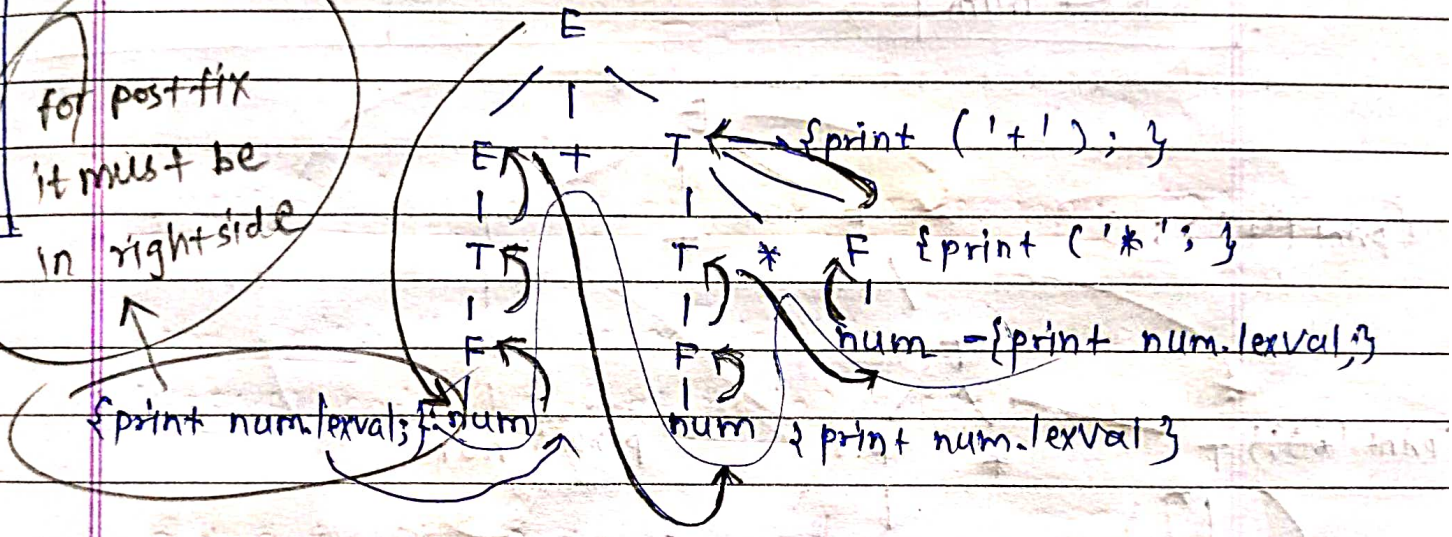
SDT for Infix to Postfix

	Semantic Actions
(contid.....) $E \to E+T$	{print ('+');}
$E \to T$	{print ('-');}
$T \to T*F$	{print ('*');}
$T \to F$	-
$F \to (E)$	-
$F \to num$	{print (num.lexval);}
$E \to E-T$	{print ('-');}

String :- 1+2*3

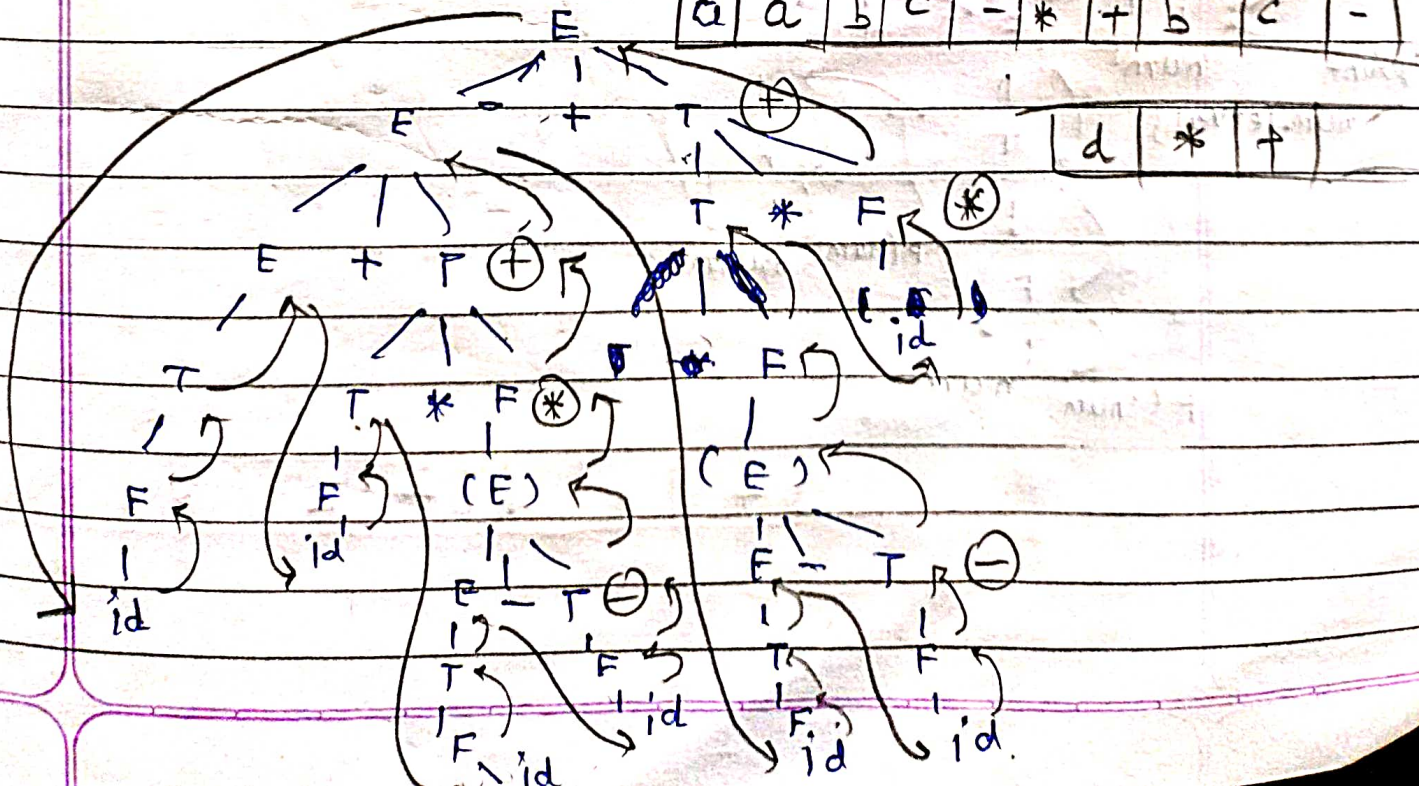
1	2	3	*	+

for postfix it must be in right side



String :- a+a*(b-c)+(b-c)*d

a	a	b	c	-	*	+	b	c	-	d	*	+



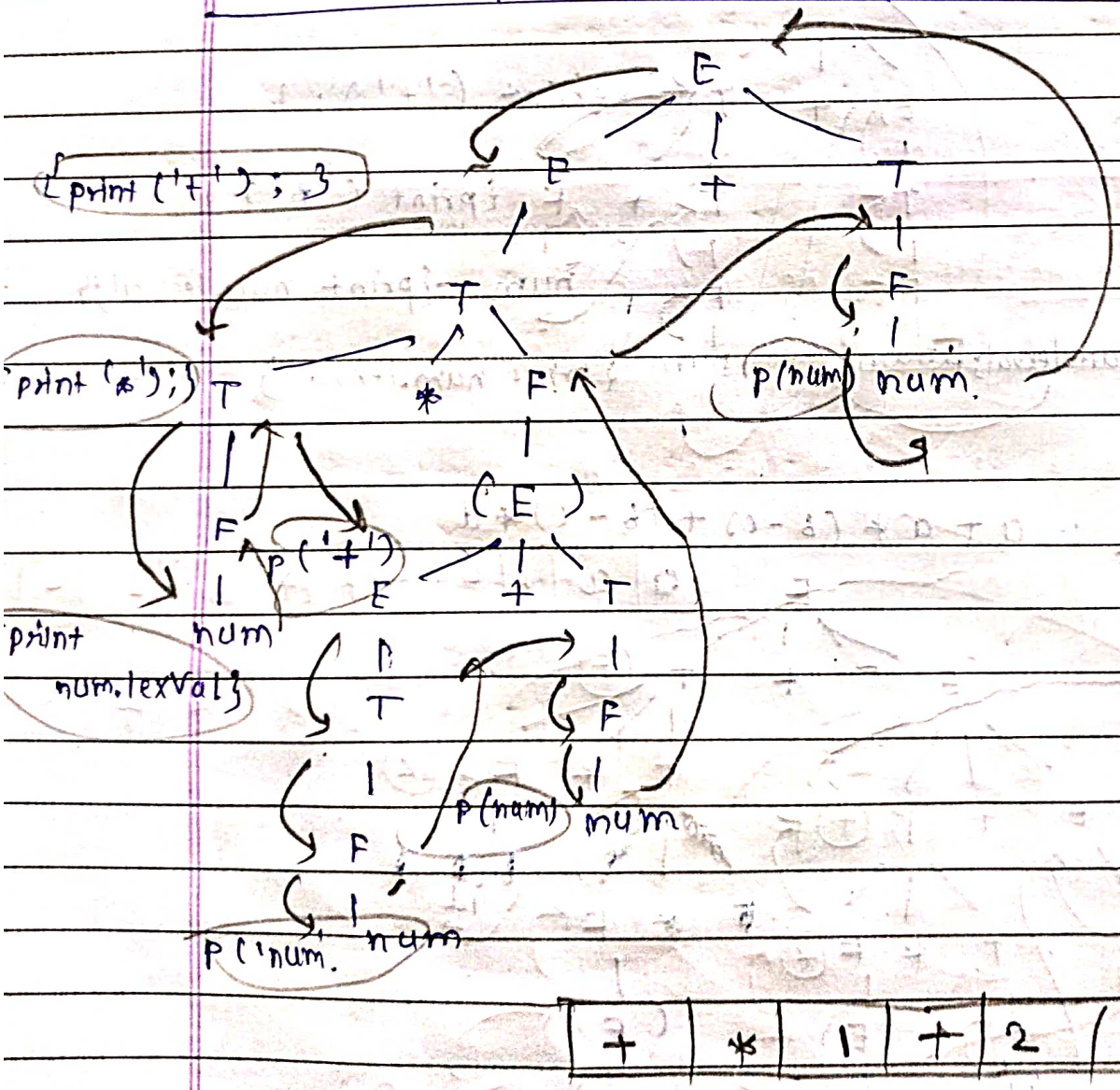
SDT for infix to prefix

Q. 1 * (2+3) +5

1 *(+23) +5

(*1+23)+5 ⇒ +*1+235

Grammer:	Semantics rules	Semantic Actions
E → E + T	E.val := E.val + T.val	{print('+');}
E → T		
T → T * F		
T → F		
F → (E)		
F → num		



+	*	1	+	2	3	5

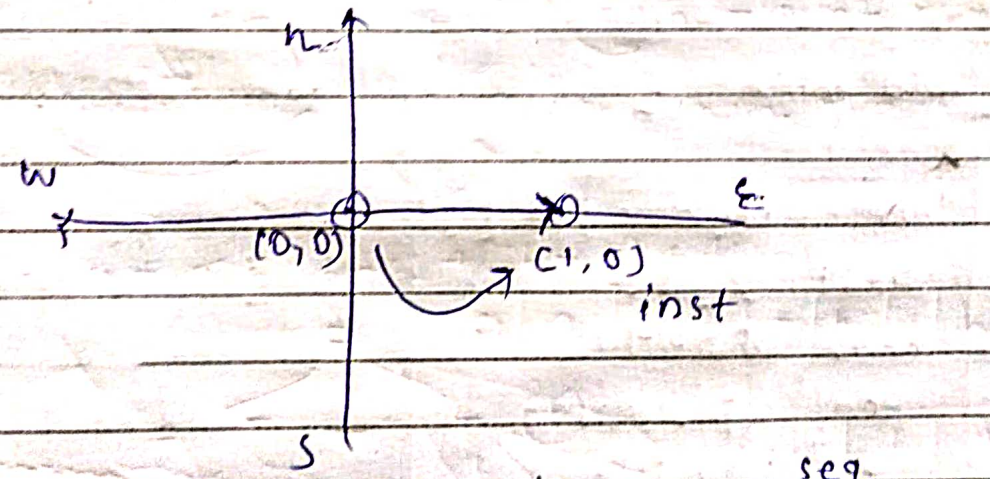
Navigation

(*) Suppose A robot can be instructed to move 1 step East, North, West or South from it's current position.

seq. → seq . instr / begin

instr → E / W / N / S

Grammer	Semantic Rules
seq. → begin	seq.x := 0 seq.y := 0
seq. → seq instr	seq.x = seq.x + instr.dx seq.y = seq.y + instr.dy
instr → E	instr.dx := 1 instr.dy := 0
instr → W	instr.dx := −1 instr.dy := 0
instr → N	instr.dx := 0 instr.dy := 1
instr → S	instr.dx := 0 instr.dy := −1



String :-

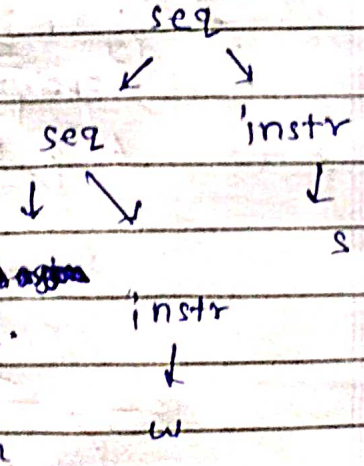
begin W S

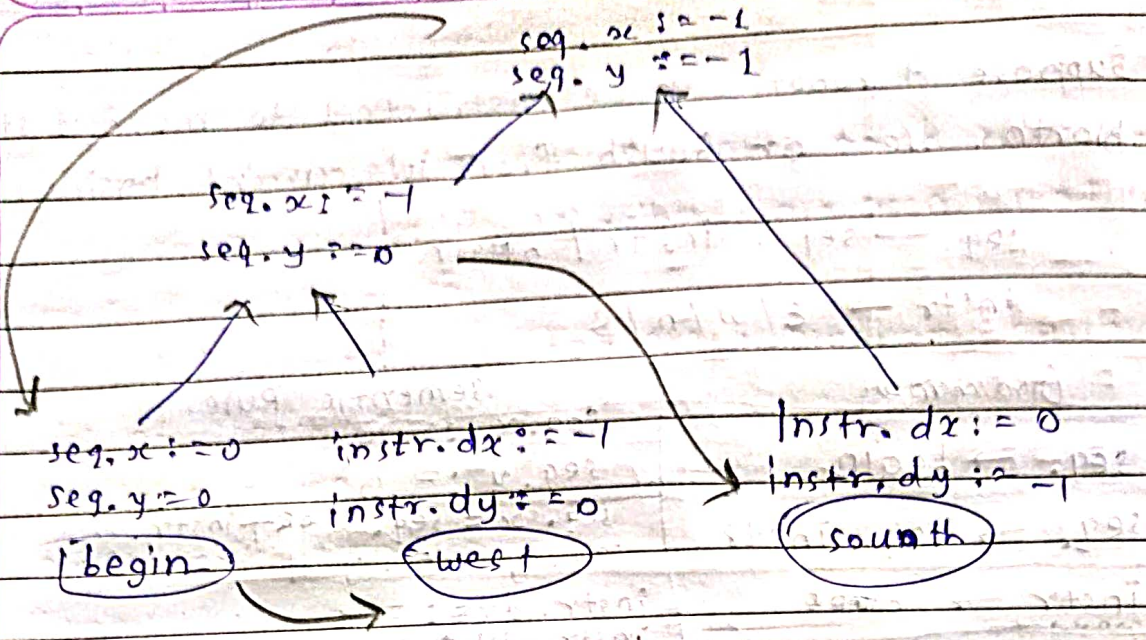
↓ ↓

(0,0) (−1,0) (−1,−1)

seq.x=0 instr.dx=−1 → 0

seq.y=0 instr.dy=0 → −1




seq. x := -1
seq. y := -1
seq. x := -1
seq. y := 0
seq. x := 0
seq. y := 0
begin
instr. dx := -1
instr. dy := 0
west
instr. dx := 0
instr. dy := -1
south

Symbol Table

- Symbol Table is an important data-structure created and maintained by compilers in order to store information about the occurence of various entities such as variable names, function names, objects, classes, interfaces, etc.

- Symbol Table is used by both the analysis & synthesis part of compiler.

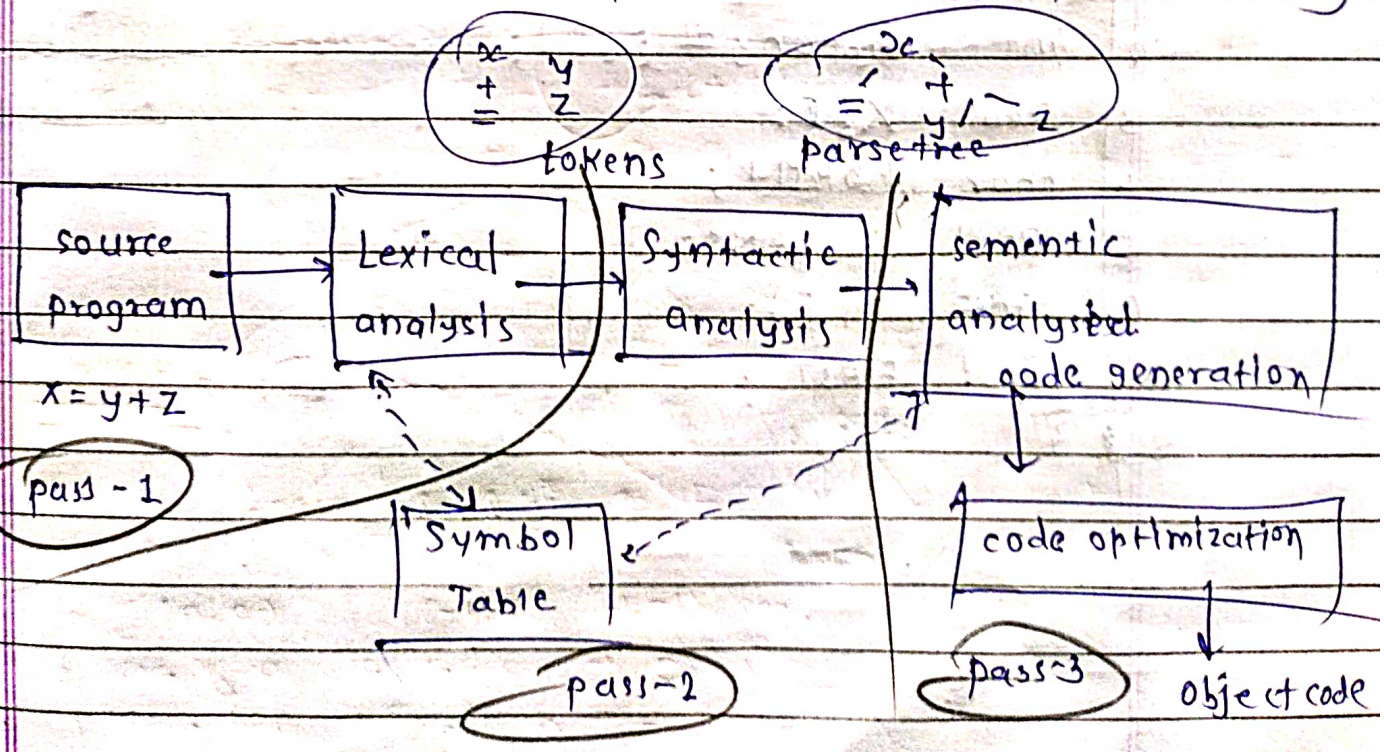
Purpouse of the Symbol Table:

(1) To store the names of all entities in a structured form at one place

(2) To verify if a variable has been declared

(3) To implement type checking by verifying assignments and expressions in the source code are semantically correct.

(4) To determine the scope of a name. (Scope Resolution)



Symbol table Context

Typical view of symbol table

- A symbol table is a series of rows each row containing a list of attribute values that are associated with the particular variable.
- It is not compulsory that every program has every attribute

List of Attributes :-

1. Variable name
2. object code address
3. Type
4. Dimention or no. of parameters for a procedure
5. ~~Dimentt~~ Source line no. at which the variable is declared
6. " " " " " is refereneed
7. Link field for listing in alphabetical order.
8.

Eg.

```
1. void main() {
2   int company#[][];
3   Real X3D;
4   char FORM1;
5   Real B, ANS;
6   int M[];
7   Real FIRST;
8 }
```

Typical view of symbol Table:

it is require for semantic type checking

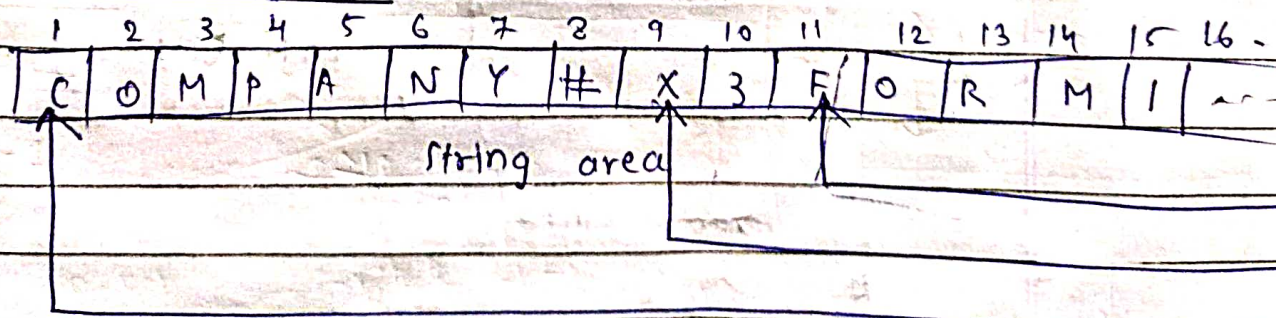
pointer	Variable Name	Address	Type	Dimension	Line Declared	Line References
③	company#	40	int	2	2	-
⑦	x_3	42	Real	0	3	-
④	FORM1	32	char	0	4	-
②	B	45	Real	0	5	-
①	ANS	60	Real	0	5	-
⑥	M	61	str	1	6	-
⑤	FIRST	62	Real	0	7	-

Ordered list Alphabetical order

Name any you want is different

Cross-Reference Listing :- Alphabatical order we have to arrange in variable name & pointer column is not there.

String Descriptor :-



Date
Page

Vearíable name

Position	length
1	5
9	2
11	5
:	:

(6) Remaining all, as it is as previous one :

insert() — insert (a, int)
lookup() — lookup (a)
Delete() — delete (a)

Scope Management -
↳ for identify variable at locally or globally.

(*)

```
int a;
void f1()
{
    int b;
    {
        int c;
    } int d;
      int e;
}
```

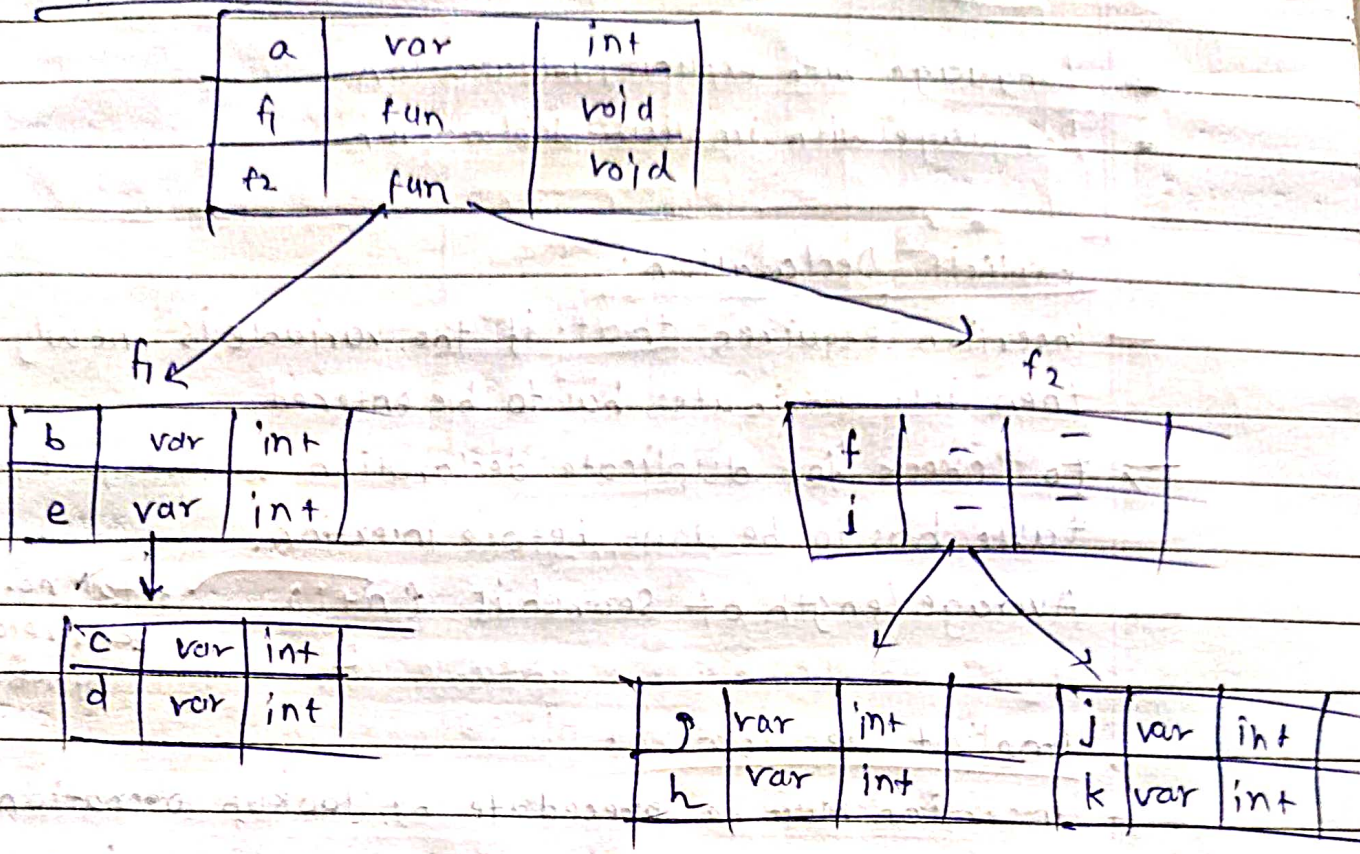
```
void f2()
{
    int f;
    {
        int g;
        int h;
    }
    int i;
}
```

```
{
    int j;
    int k;
}
```

~~var int~~

1
2

Symbol Table Entry



Symbol Table Organization for non-block structured Languages :-

(1) List → unordered symbol table
ordered symbol table
(2) Binary Search Tree (BST)
(3) HashTable

(1) Unordered Symbol Table

- Language with explicit declaration
- Language with implicit declaration

Explicit Declaration:-

→ Insertion requires that if the variable is newly define then its attributes has to be entered

→ To check for duplicate declaration a lookup of entire table has to be done before insertion.

→ Average length of search is $\frac{(n+1)}{2}$ (n: total no. of variables / records in symbol table)

Implicit Declaration:

→ insertion must be precedate by lookup operation if the lookup fails variable attributes must be inserted. because they are not present.

→

Ordered Symbol Table

→ Variables are alphabatically order.

→ Insertion requires to lookup where the attribute should be placed.

(2) Tree structure Symbol Table

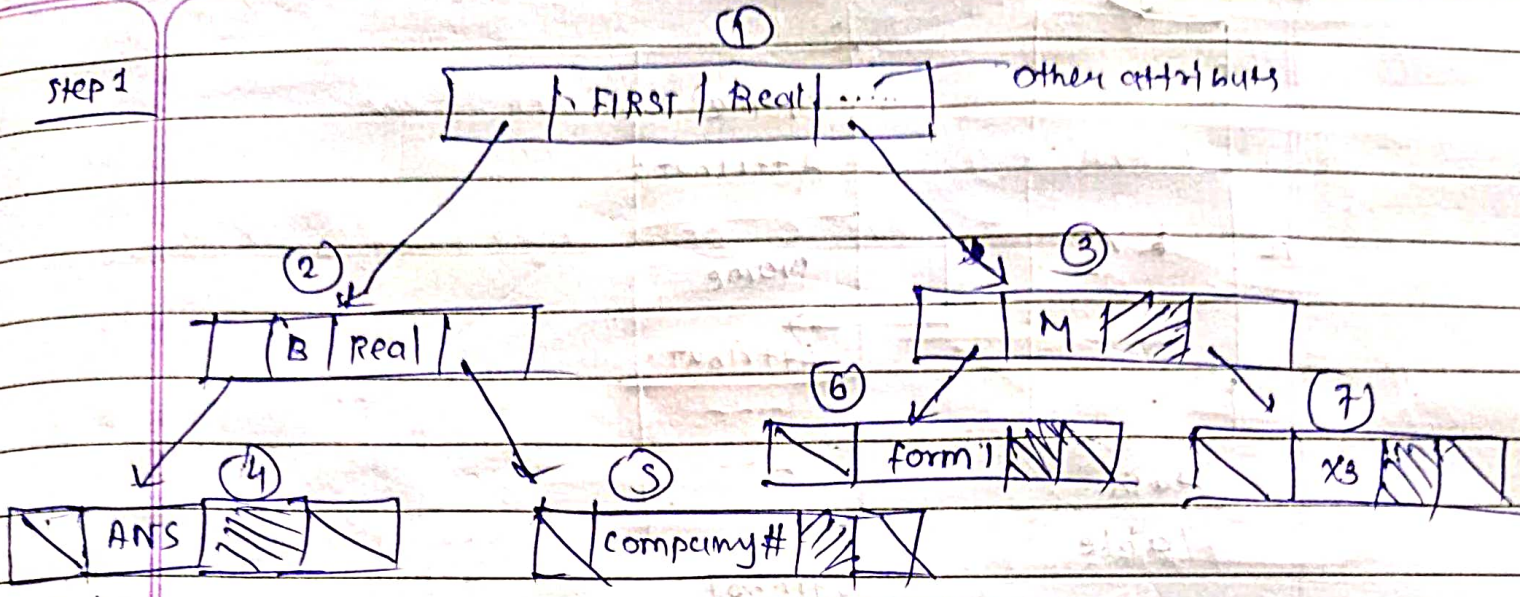
Variable:-

First, B, ANS, company11, M, FORM1, & x3

Step1: Logical Relationship

step 2: physical Representation

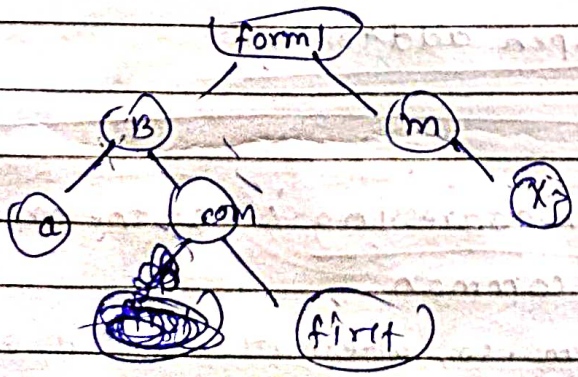
Step 1

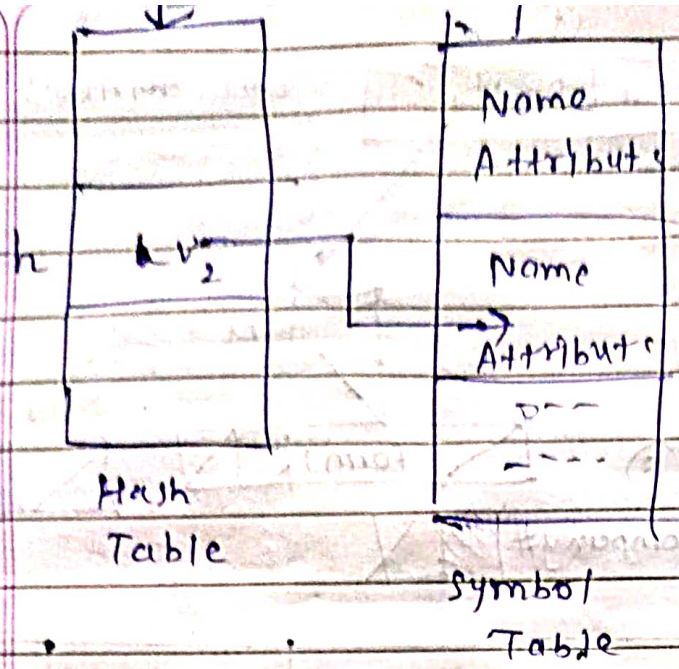


Step 2

Table position	Name field	Type of Attributes	Other Attributes	Left Pointer	Right Pointer
			add dimen Tine d. Pointer		
1	first	Real	40 2 2 3	2	3
2	B	Real	11	4	3
3	M	int		6	7
4	ANS	Real	like	0	0
5	company#	int	wise	0	0
6	form1	char	as per	0	0
7	X3	Real	previous table.	0	0

Ex. FORM1, B, COMPANY#, ANS, M, X3, and FIRST





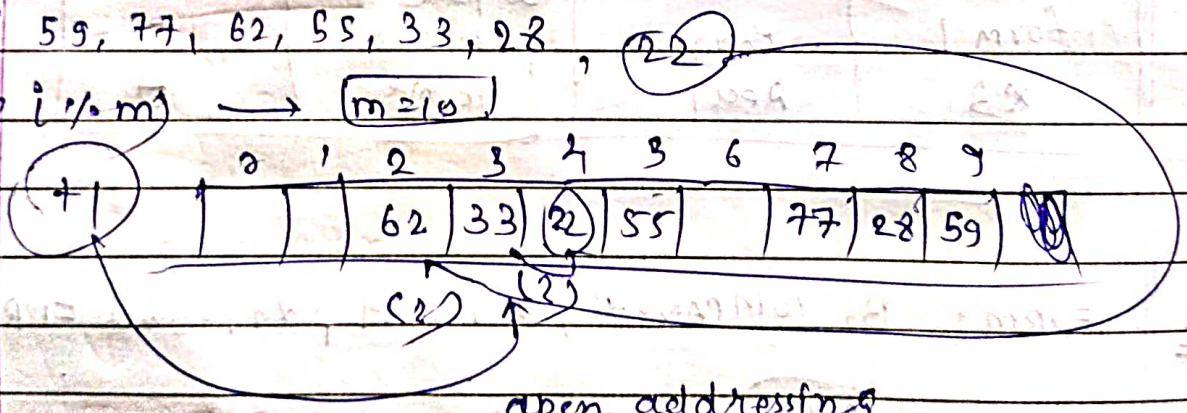
→ HashTable Technique is suitable for searching. &

Hash function :- hash function is the mapping b/w an item and the place in the hash table where that item resides.

→ The hash function takes a collection item & returns an integer in the range of slot names from 0 to m-1

Ex. 59, 77, 62, 55, 33, 28, 22

(i % m) → m=10



open addressing

Disadvantage of open addressing is as the array fields collesion become more common.

Table size is a problem. so, alternative sol^n is saperate chaining.

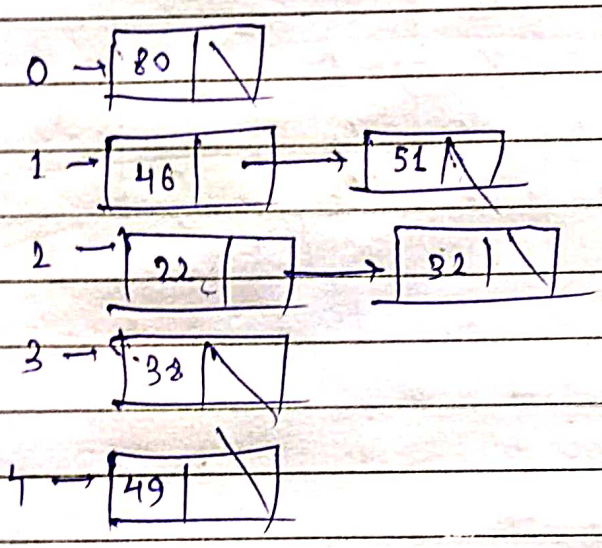
Saperate Chaining

→ The Hash Table is an array of pointers to linked list which is called as buckets.

→ Collesions are addressed by adding a new identifier into a linked list.

Eg. 22, 38, 32, 43, 46, 49, 51, ~~53~~, 56, 71, ~~73~~, ~~79~~, 80

m = 5



Done