LALR (Lookahead LR)

$S' \to S$
$S \to aSb$
$S \to \epsilon$

construct LR(1) item set

closure $([S' \to .S, \$]) = \{ [S' \to .S, \$],$
$[S \to .aSb, \$],$
$[S \to ., \$] \}$ — I_0

$GOTO(I_0, S) = closure([S' \to S., \$])$
$= \{ [S' \to S., \$] \}$ — I_1

$GOTO(I_0, a) = closure([S \to a.Sb, \$])$
$= \{ [S \to a.Sb, \$],$
$[S \to .aSb, b]$
$[S \to ., b] \}$ — I_2

$GOTO(I_2, S) = closure([S \to aS.b, \$])$
$= \{ [S \to aS.b, \$] \}$ — I_3

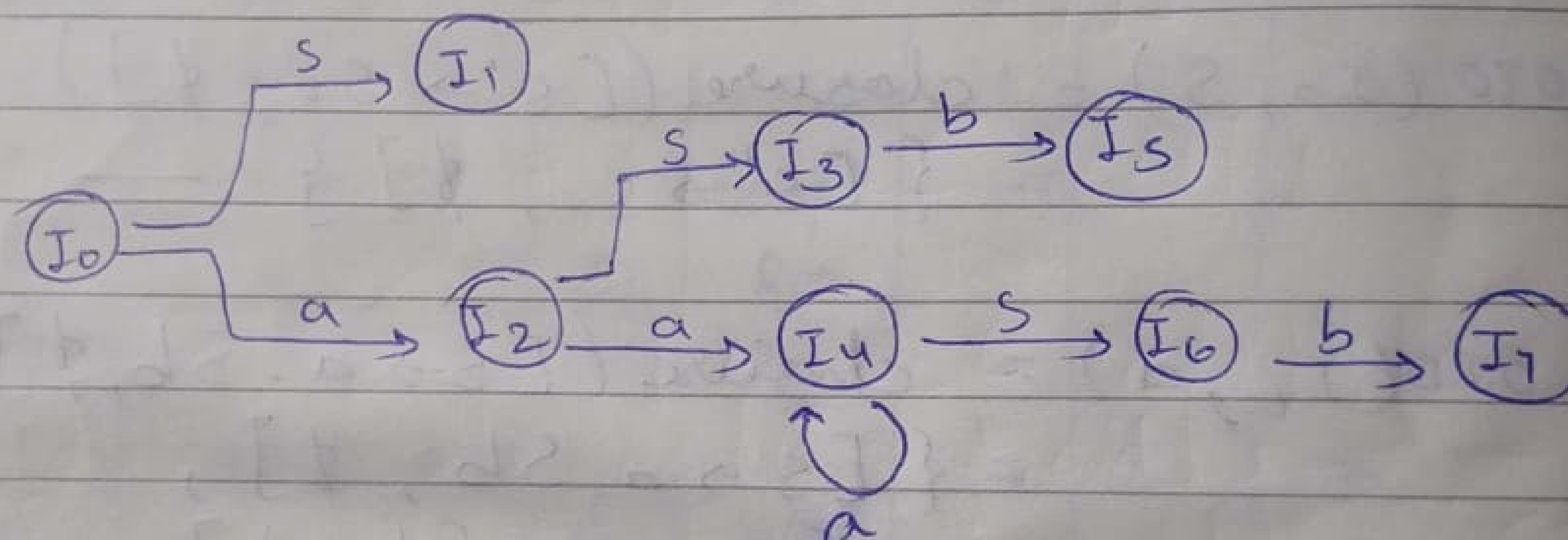
$GOTO(I_2, a) = closure([S \to a.Sb, b])$
$= \{ [S \to a.Sb, b]$
$[S \to .aSb, b]$
$[S \to ., b] \}$ — I_4

$GOTO(I_3, b) = closure([S \to aSb., \$])$
$= \{ [S \to aSb., \$] \}$ — I_5

$GOTO(I_4, S) = closure([S \to aS.b,\ b])$
$= \{[S \to aS.b,\ b]\}$ — I_6

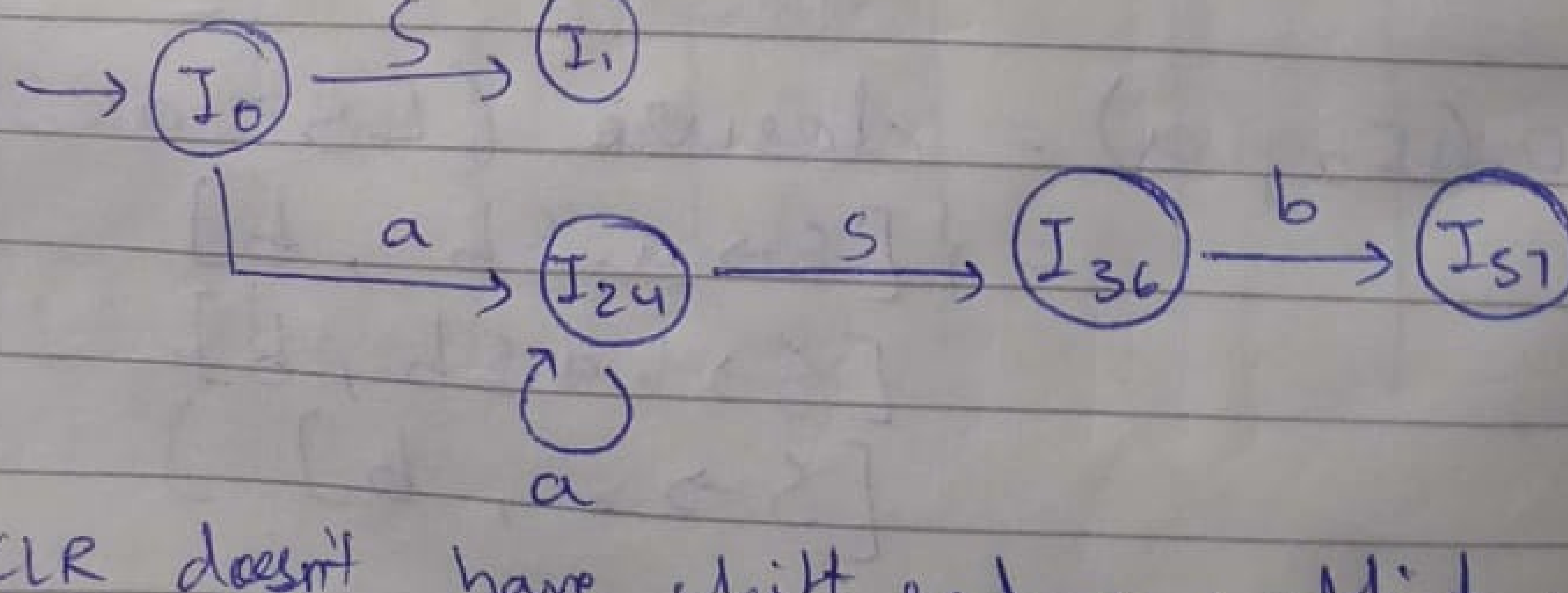
$GOTO(I_4, a) = closure([S \to a.Sb,\ b])$
$= \{[S \to a.Sb,\ b]$
$[S \to .aSb,\ b]$
$[S \to .,\ b]\}$ ← I_4

$GOTO(I_6, b) = closure([S \to aSb.,\ b])$
$= \{[S \to aSb.,\ b]\}$ — I_7



2 & 4 are same only lookaheads are differen[t]
5 & 7 are merged as I_{57}
3 & 6 are merged

DFA after merging states :-



If CLR doesn't have shift reduce conflict LALR will also not have SR conflict
but if CLR ~~may be~~ doesn't have RR conflict LALR may or maynot have RR conflict.
Reduce reduce

	ACTION			GOTO
	a	b	$	S
I_0	S_{24}		R_2	1
I_1			accept	
I_{24}	S_{24}	R_2		36
I_{36}		S_{57}		
I_{57}		R_1	R_1	

$S \to aSb$ — ①
$S \to \epsilon$ — ②

∵ I_0 contains $[S \to ., \$]$ ∴ R_2 for I_0 on $

H.W. LALR for
$S' \to S$
$S \to CC$
$C \to cC / d$

LR(1) grammar
closure($[S' \to .S]$) = { $[S' \to .S, \$]$,
$[S \to .CC, \$]$,
$[C \to .cC, c/d]$,
$[C \to .d, c/d]$ } — I_0

GOTO(I_0, S) = closure($[S' \to S., \$]$)
= { $[S' \to S., \$]$ } — I_1

GOTO(I_0, C) = closure($[S \to C.C, \$]$)
= { $[S \to C.C, \$]$,
$[C \to .cC, \$]$
$[C \to .d, \$]$ } — I_2

~~GOTO(I_0, d) = closure([C → d.~~
GOTO(I_0, c) = closure($[C \to c.C, c/d]$)
= { $[C \to c.C, c/d]$, $[C \to .cC, c/d]$
$[C \to .d, c/d]$ } — I_3

$GOTO(I_0, d) = closure(\{[C \to d., c/d]\})$
$= \{[C \to d., c/d]\}$ — I_4

$GOTO(I_2, C) = closure([S \to CC., \$])$
$= \{[S \to CC., \$]\}$ — I_5

$GOTO(I_2, c) = closure([C \to c.C, \$]$
$= \{[C \to c.C, \$]$
$[C \to .cC, \$]$
$[C \to .d, \$]\}$ — I_6

$GOTO(I_2, d) = \{[C \to d., \$]\}$ — I_7

$GOTO(I_3, C) = \{[C \to cC., c/d]\}$ — I_8

$GOTO(I_3, c) = \{[C \to c.C, c/d], [C \to .cC, c/d]$
$[C \to .d, c/d]\}$ — I_3

$GOTO(I_3, d) = \{[C \to d., c/d]\}$ — I_4

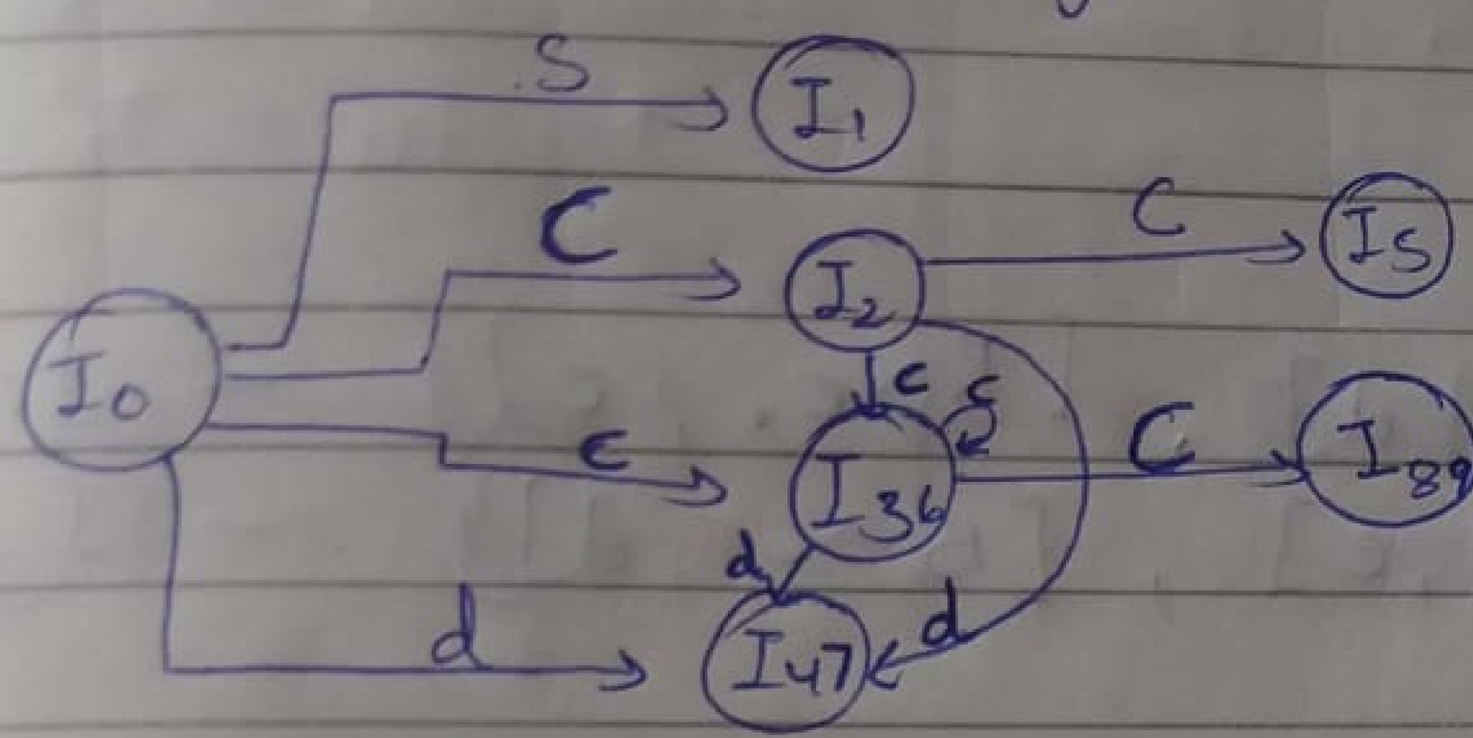
$GOTO(I_6, C) = \{[C \to cC., \$]\}$ — I_9

$GOTO(I_6, c) = \{[C \to c.C, \$],$
$[C \to .cC, \$]$
$[C \to .d, \$]\}$ — I_6

$GOTO(I_6, d) = \{[C \to d., \$]\}$ — I_7

3 & 6 will merge
4 & 7 will merge
8 & 9 will merge

$S \rightarrow CC$ – ①
$C \rightarrow cC$ – ②
$C \rightarrow d$ – ③



states	c	d	$	C	S
0	S36	S47		2	1
1			accept		
2	S36	S47		5	
36	S36	S47		89	
47	R3	R3	R3		
5			R1		
89	R2	R2	R2		

Q. $S' \to S$
$S \to aAd \mid bBd \mid aBe \mid bAe$
$A \to c$
$B \to c$

Two states
$\{[A \to c., d], [B \to c., e]\}$ — I_i
$\{[A \to c., e], [B \to c., d]\}$ — I_j

LALR
After merge
$\{[A \to c., d/e], [B \to c., d/e]\}$ - I_{ij} RR conflict

Semantic Analysis

① Static Semantics
② Dynamic Semantics → if is not handled by compiler
→ int a, b, c;
a = b * c;
↳ if value of a goes out of bound of int
→ overflow of stack

Syntax Directed Definitions (SDD)
- attach attribute to every symbol of the grammar
- attach rules to every production

Productions	Semantic Rules
$S \to E$	$S.val = E.val$ → attribute
$E \to E_1 + T$	$E.val = E_1.val + T.val$
$E \to T$	$E.val = T.val$
$T \to T_1 * F$	$T.val = T_1.val * F.val$
$T \to F$	$T.val = F.val$
$F \to (E)$	$F.val = E.val$
$F \to digit$	$F.val = digit.lexval$

Attribute grammar :

bottom up ① Synthesized → $T \to T * F$ value is calculated by children or itself

top down ② Inherited

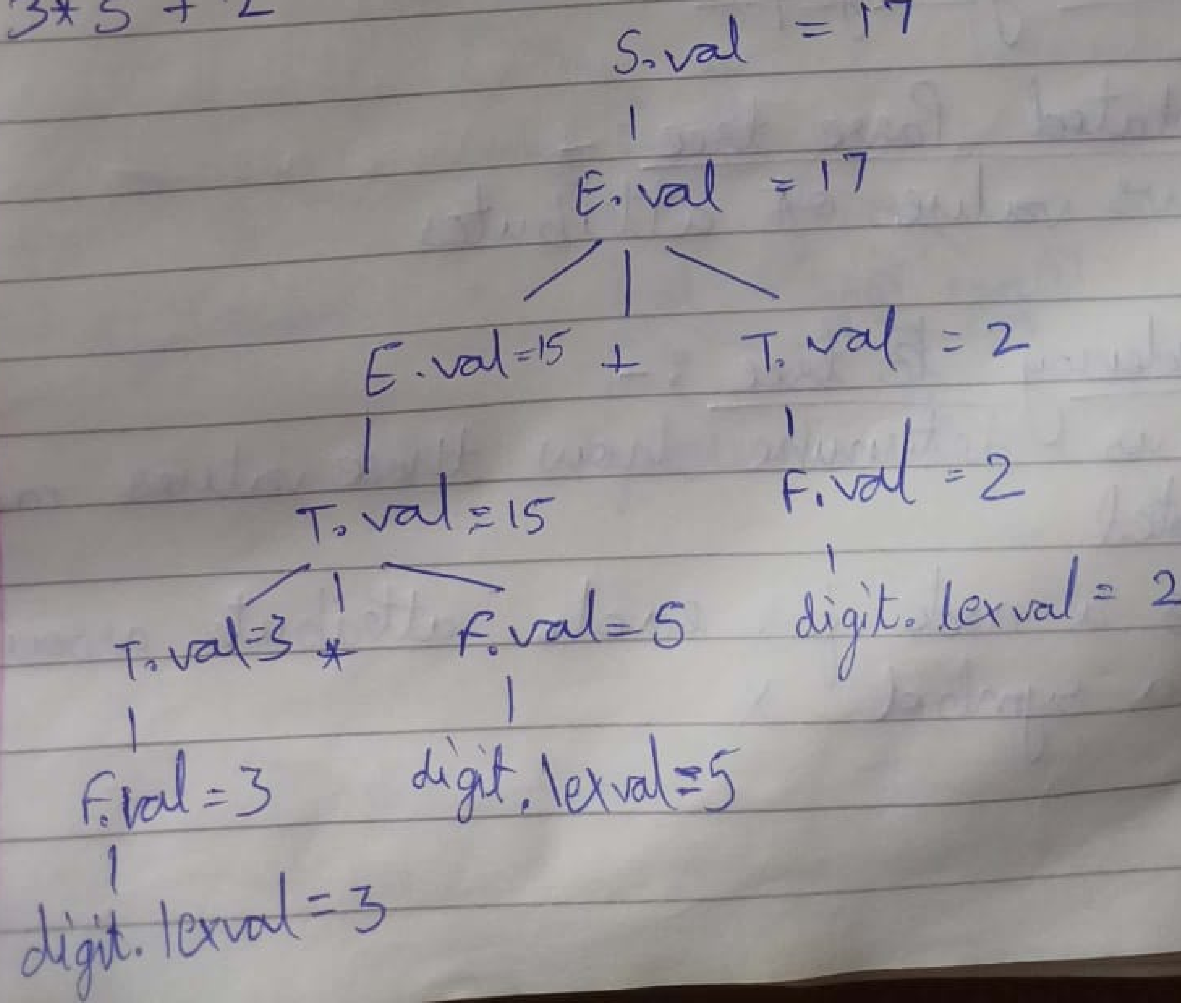
↳ $A \to BC$

$B.val \to A.val * 2$ B.val inherits value from it's parent A.val

$B.val = C.val$ → c is sibling of B still it is inherited

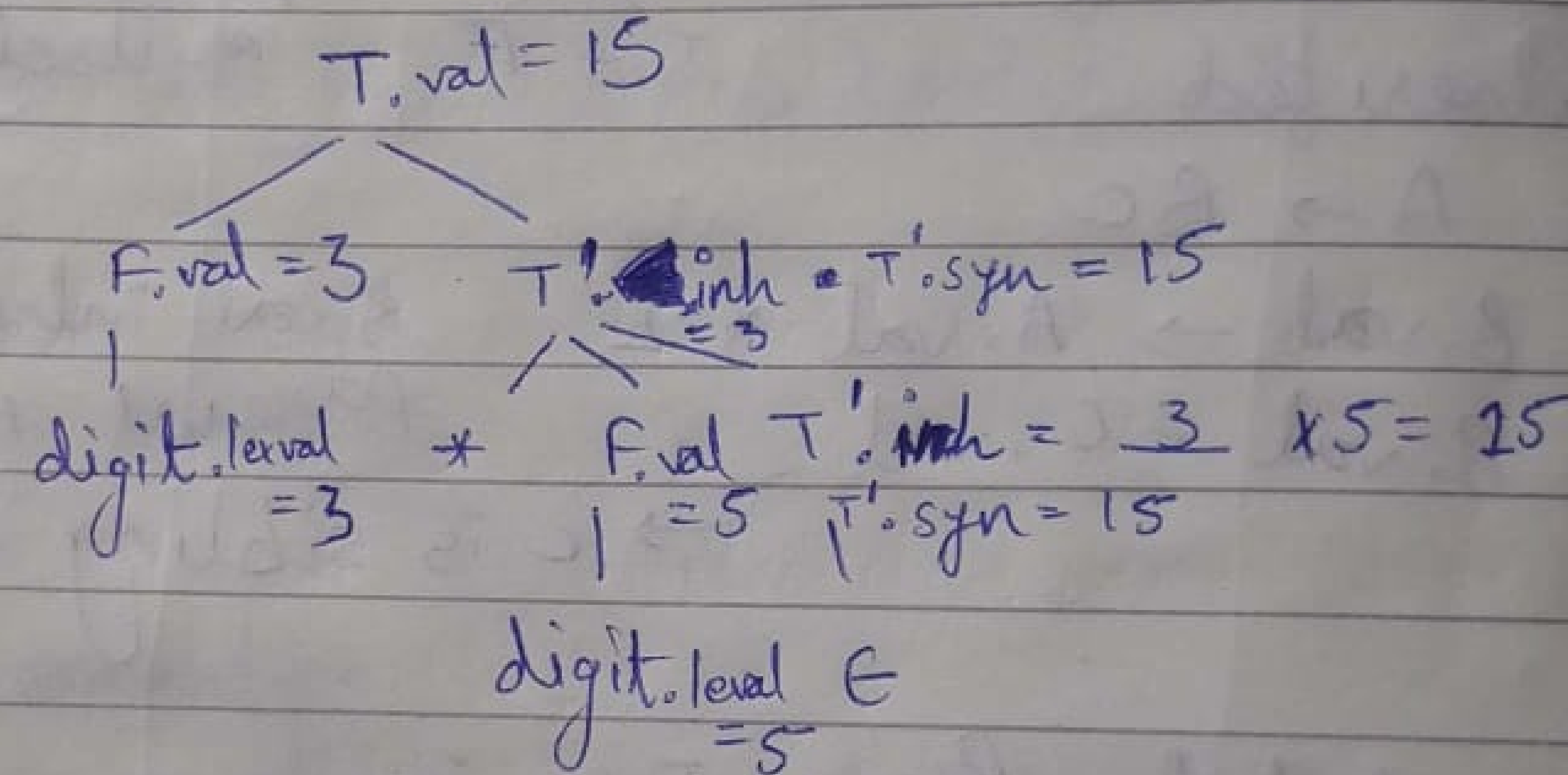
Annotated Parse Tree :

$3 * 5 + 2$



Productions	Semantic Rules
1) $T \rightarrow FT'$	$T'.inh = F.val$ $T.val = T'.syn$
2) $T' \rightarrow * F T_1'$	$T_1'.inh = T'.inh \times F.val$ $T'.syn = T_1'.syn$
3) $T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4) $F \rightarrow digit$	$F.val = digit.lexval$

3 * 5



Dependency Graph

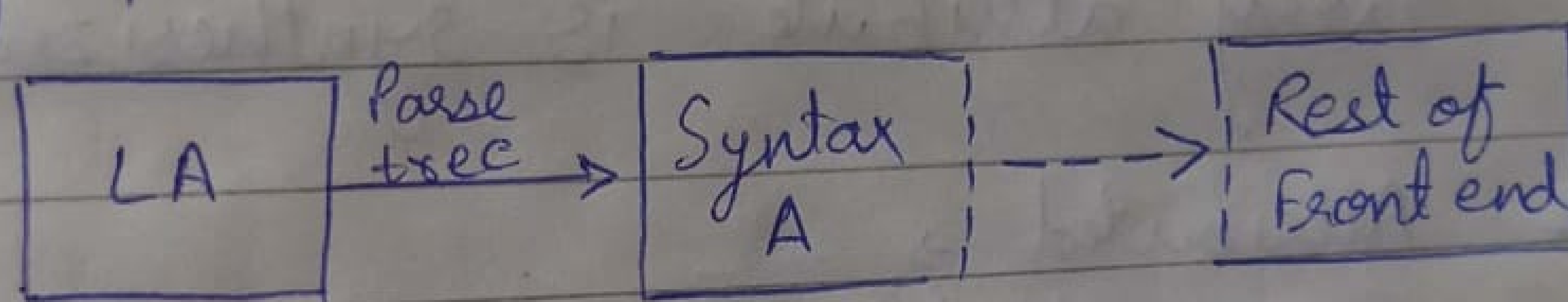
Annotated Parse tree :-
shows values of attributes

Dependency tree :-

* Helps us determine how those values can be compted
* It has a node for each attribute associated with a symbol X

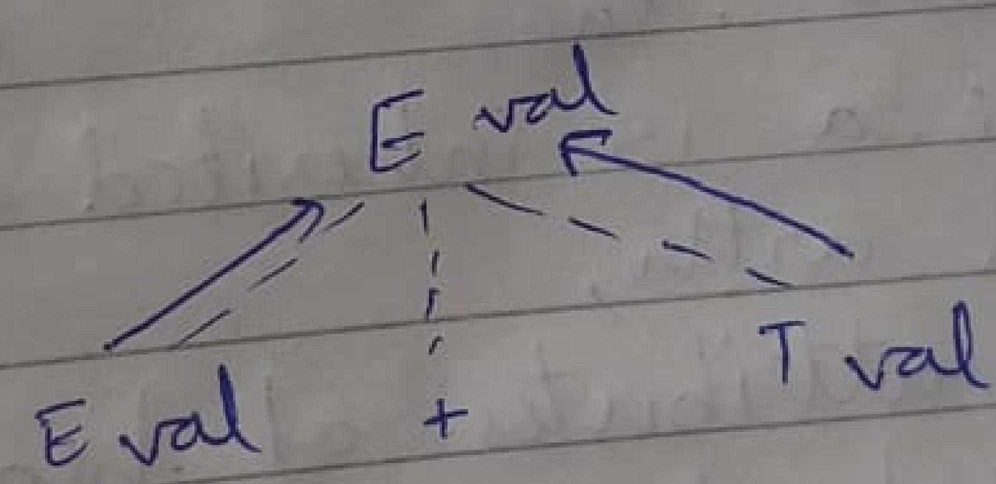
Ordering the Evaluation of Attributes:

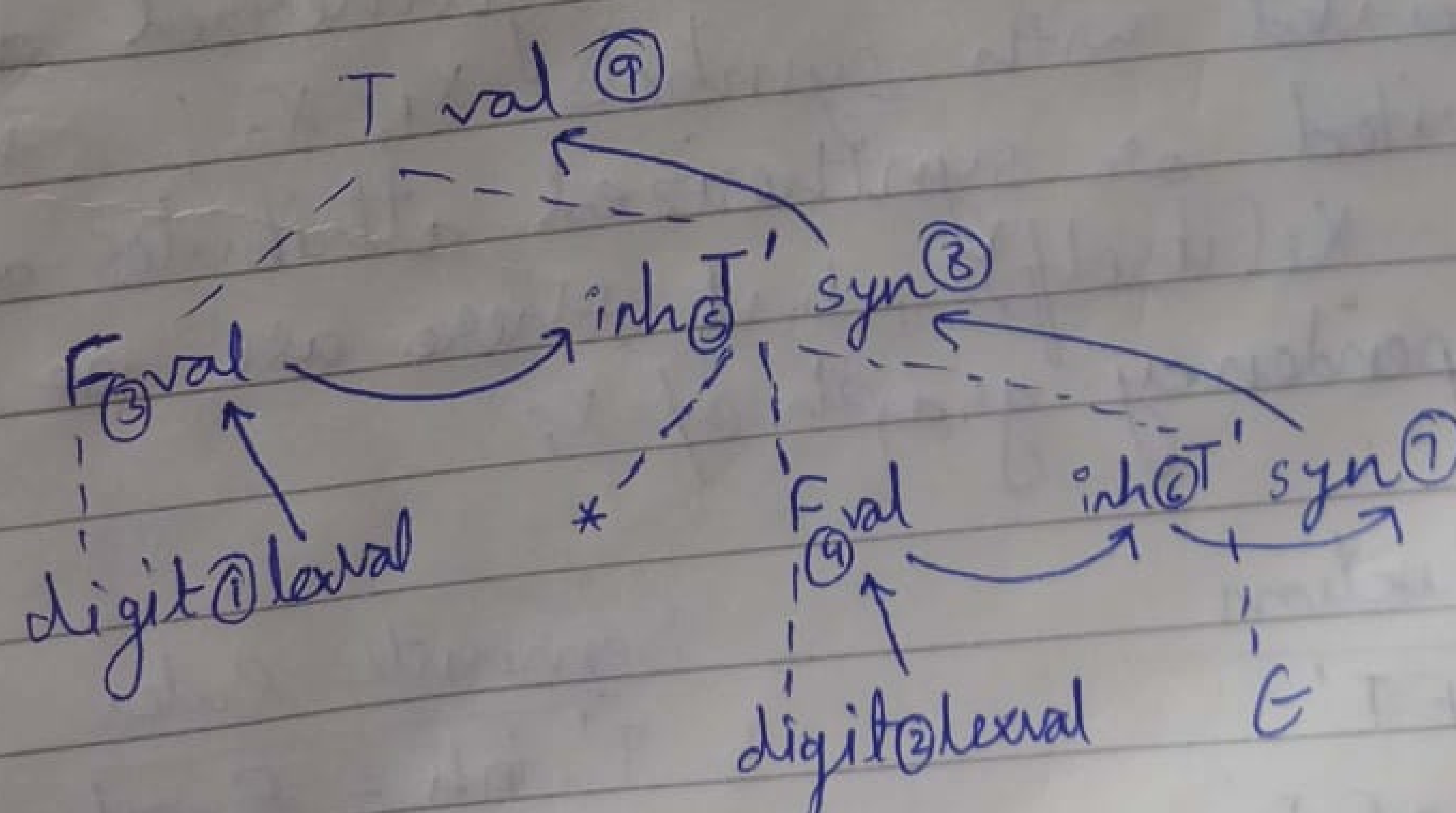
* Allowable orders of evaluation are those sequence of nodes $N_1, N_2, \ldots N_K$ such that if there is an edge of the dependency graph from N_i to N_j, then $i < j$.
* Such ordering is called topological sort of the graph (The graph should be acyclic)



$E.val = E_1.val + T.val$

$E \rightarrow E_1 + T$





Two classes of Syntax Directed Definition (SDD):

* Those classes guarantee an evaluation order.
* They don't permit dependency graph with cycle
* Can be implemented efficiently in connection with top-down / bottom-up parsing.

1) S- Attributed →
Every attribute is synthesized

2) L- Attributed →
every attribute must be,
i) Synthesized or
ii) Inherited, but with following rules:
If $A \rightarrow X_1 X_2 \ldots X_n$
and $X_i.a$ is inherited and should be associated with,
a) Inherited attributes of head A
b) Either inherited or synthesized attributes associated with symbols $X_1, X_2, \ldots X_{i-1}$
c) Inherited or synthesized attributes associated with X_i (itself), but there are no cycles in a dependency graph of X_i

Eg: ①

Production	Semantic Rules
$T \rightarrow FT'$	$T'.inh = F.val$
$T' \rightarrow *FT_1'$	$T_1'.inh = T'.inh \times F.val$

② $A \rightarrow BC$ ✓ $A.s = B.i$
✗ $B.i = f(C.c, A.s)$

Production	Semantic Rules
1) $D \to TL$	$L.type = T.type$
2) $T \to int$	$T.type = integer$
3) $T \to float$	$T.type = float$
4) $L \to L_1, id$	$L_1.type = L.type$ addType (id.entry, L.type)
5) $L \to id$	addType (id.entry, L.type) ↳ add entry to Symbol Table

Syntax - Directed Translation Schemes

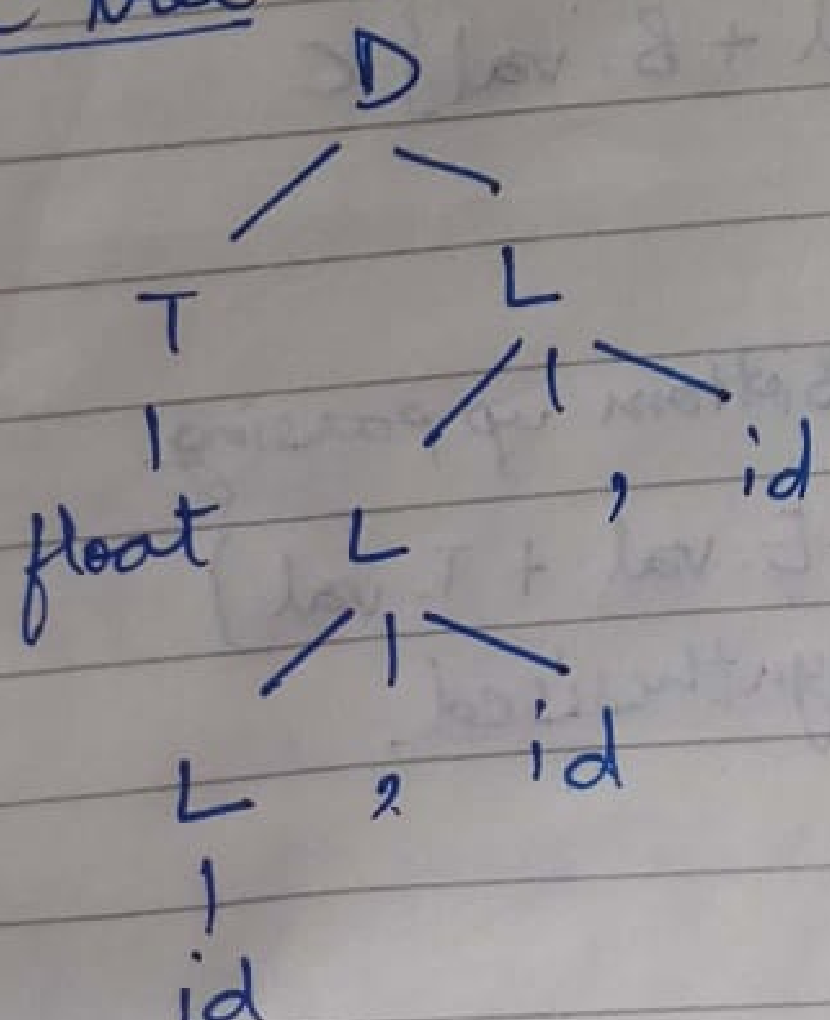
↳ intermediate code generation

* SDT is a CFG with program fragments embedded within the production bodies.
* Program fragments are called Semantic actions.
* They can appear at any ~~given~~ position within a production body.

\# float a, b, c
  T id id id

Q Draw dependency graph for this string.

Parse tree



Postfix Translation Scheme (S-attributed gramm...

$L \to E_n$ { print (E.val); }

$E \to E_1 + T$ { E.val = E_1.val + T.val ; }

$E \to T$ { E.val = T.val ; }

$T \to T_1 \times F$ { T.val = T_1.val × F.val ; }

$T \to F$ { T.val = F.val ; }

$F \to$ digit { F.val = digit.lexval ; }

L-attributed

Ex:

$D \to T$ { L.type = T.type;} L

$T \to$ int { T.type = integer ; }

$T \to$ float { T.type = float ; }

$L \to$ { L_1.type = L.type } L_1, id { enter (id.prt, L.ty...

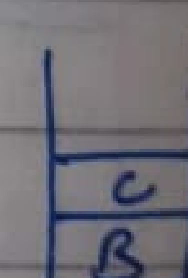
$L \to$ id { enter (id.ptr, L.type); }

Ex: $A \to BC$

$\to B$ { $\underbrace{\text{C.val}}_{\text{inherited}}$ = A.val + B.val } C

$\to aB$

$\to aa$



Bottom up parsing.

$E \to E + T$ { $\underbrace{\text{E.val}}_{\text{synthesised}}$ = E.val + T.val }

X	Y	Z
X_x	Y_y	Z_z

} stack

pointer ↳ if only one attribute is present directly store th... value otherwise use a pointer which store ... the location of attribute

Productions	Actions
$L \to E\$$	{ print (stack[top-1].val); top = top-1; }
$E \to E_1 + T$	{ stack[top-2].val = stack[top-2].val + stack[top].val; top = top-2; }
$E \to T$	
$T \to T_1 * F$	{ stack[top-2].val = stack[top-2].val * stack[top].val; top = top-2; }
$T \to F$	
$F \to (E)$	{ stack[top-2].val = stack[top-1].val; top = top-2; }
$F \to digit$	

Intermediate codes :-

1) Postfix Notation
2) Syntax Tree
3) Three-Address Codes

1> Postfix Notation

If e_1 then x else y

$\Rightarrow e_1\, x\, y\, ?$

If a then if c-d then a*c else a+b else c+d

$\Rightarrow a \underbrace{\underbrace{cd-}_{e_1}\ \underbrace{ac*}_{x}\ \underbrace{ab+?}_{y}}_{x}\ \underbrace{cd+?}_{y}$ with a as e_1

jump if equal to zero to label l_1

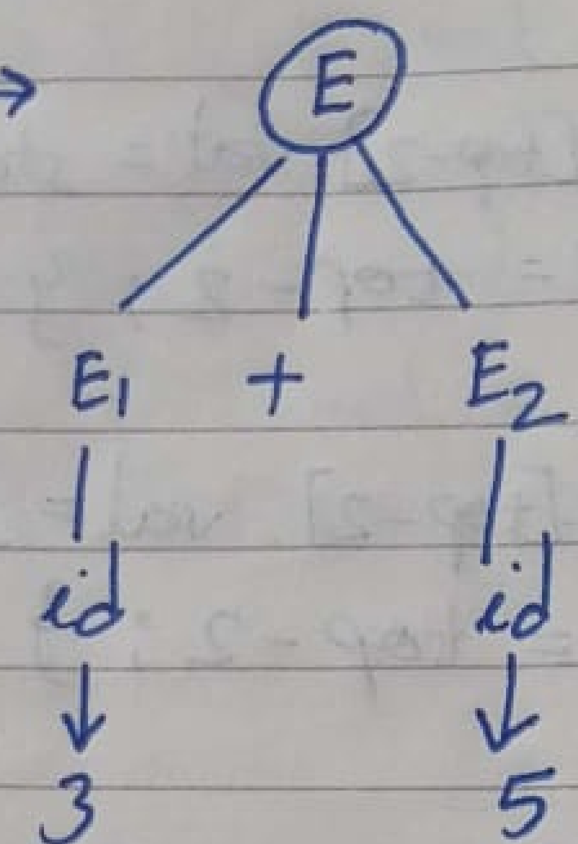
a l_1 jeqz cd- l_2 jeqz ac+ l_3 jump l_2 : ac *

jump l_1 : ab + l_3

SDT

Production	Program Fragments
* $E \rightarrow E_1 \; op \; E_2$	{print op}
$E \rightarrow (E_1)$	{ }
$E \rightarrow id$	{print id}

$3+5 \longrightarrow$



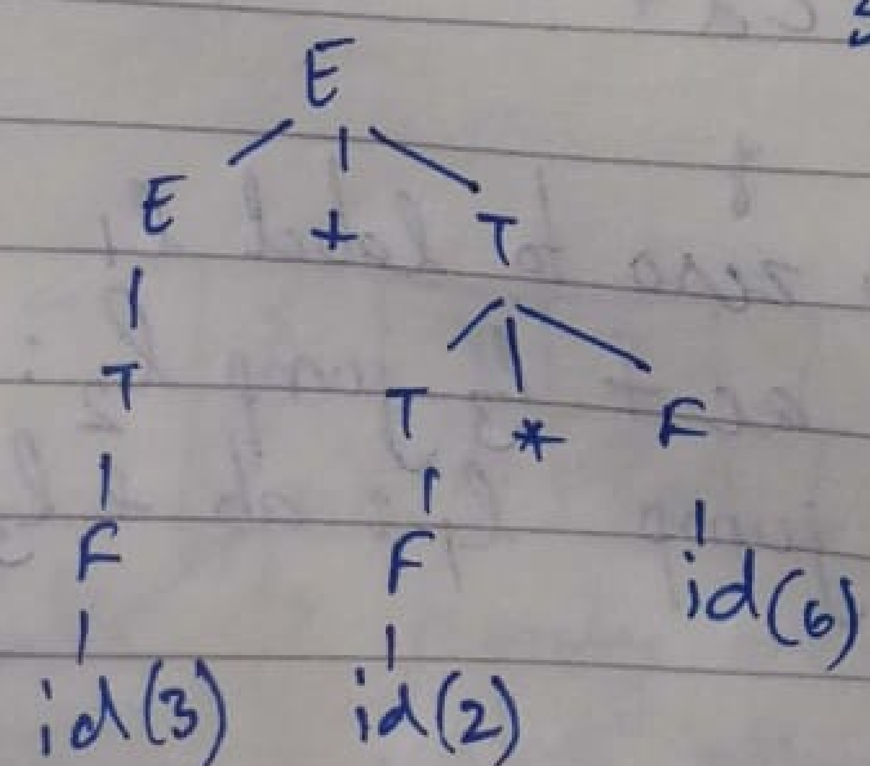
SDT to Postfix Notation

Production	Semantic Action
$E \rightarrow E_1 \; op \; E_2$	$E.code = E_1.code \,\|\|\, E_2.code \,\|\|$ "op"
$E \rightarrow (E_1)$	$E.code = E_1.code$
$E \rightarrow id$	$E.code = id$

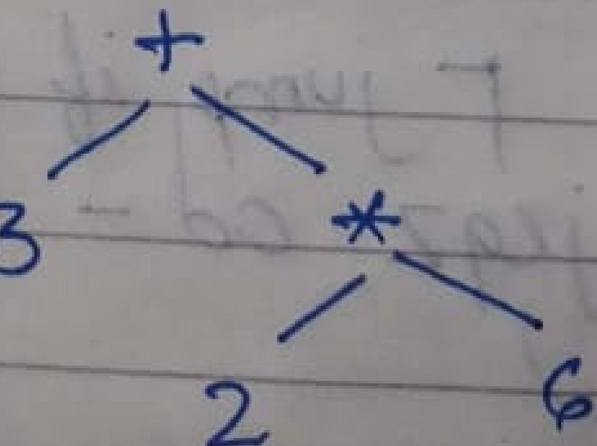
~~Production~~ ~~Program Fragments~~

2) Syntax Tree

$3+2*6$

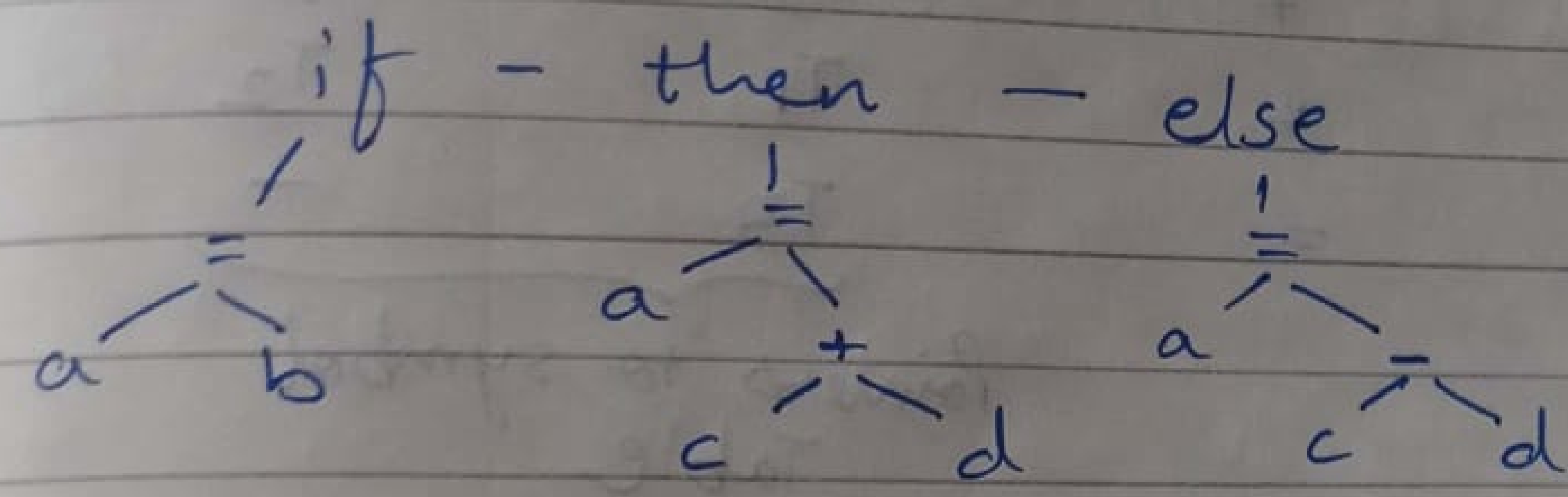


Parse Tree



Syntax Tree

If a=b then a=c+d else a=c-d



SDT scheme for syntax tree construction

Production	Semantic Action
$E \Rightarrow E_1 \text{ op } E_2$	{ E.val = Node (op, E_1.val, E_2.val)}
$E \rightarrow (E_1)$	{ E.val = E_1.val }
$E \rightarrow - E_1$	{ E.val = unary (-, E_1.val)}
$E \rightarrow id$	{ E.val = Leaf (id) }

3) Three-Address Codes

X + Y * Z

$T_1 = Y * Z$
$T_2 = X + T_1$

3 ways to represent Three-Address code in memory

1. Quadruples
2. Triples
3. Indirect Triples.

1) Quadruples

A = -B * C+D

$T_1 = -B$ $\quad T_2 = C + D$
$T_3 = T_1 * T_2$
$A = T_3$

	op	ARG(1)	ARG(2)	ARG(3)
(0)	uminus	B	–	T_1
(1)	+	C	D	T_2
(2)	*	T_1	T_2	T_3
(3)	=	T_3	–	A

ptr → ST

Pointers to symbol Table

2) Triples

	op	ARG(1)	ARG(2)
(1)	uminus	B	–
(2)	+	C	D
(3)	*	(0)	(1)
(4)	=	(2)	A

3) Indirect Triples

statements

Addr

(14)	(0)
(15)	(1)
(16)	(2)
(17)	(3)

Productions	Semantic Rules
$S \to id = E;$	S.code = E.code || gen(top.get(id.lexem) '=' E.address)
$E \to E_1 + E_2$	E.addr = new Temp(), E.code = E_1.code || E_2.code || gen(E.addr '=' E_1.addr '+' E_2.addr)
$\mid - E_1$	E.addr = new Temp() E.code = E_1.code || gen(E.addr '=' 'minus' E_1.addr)
$\mid (E_1)$	E.addr = E_1.addr E.code = E_1.code
$\mid id$	E.addr = top.get(id.lexme) E.code = ' '

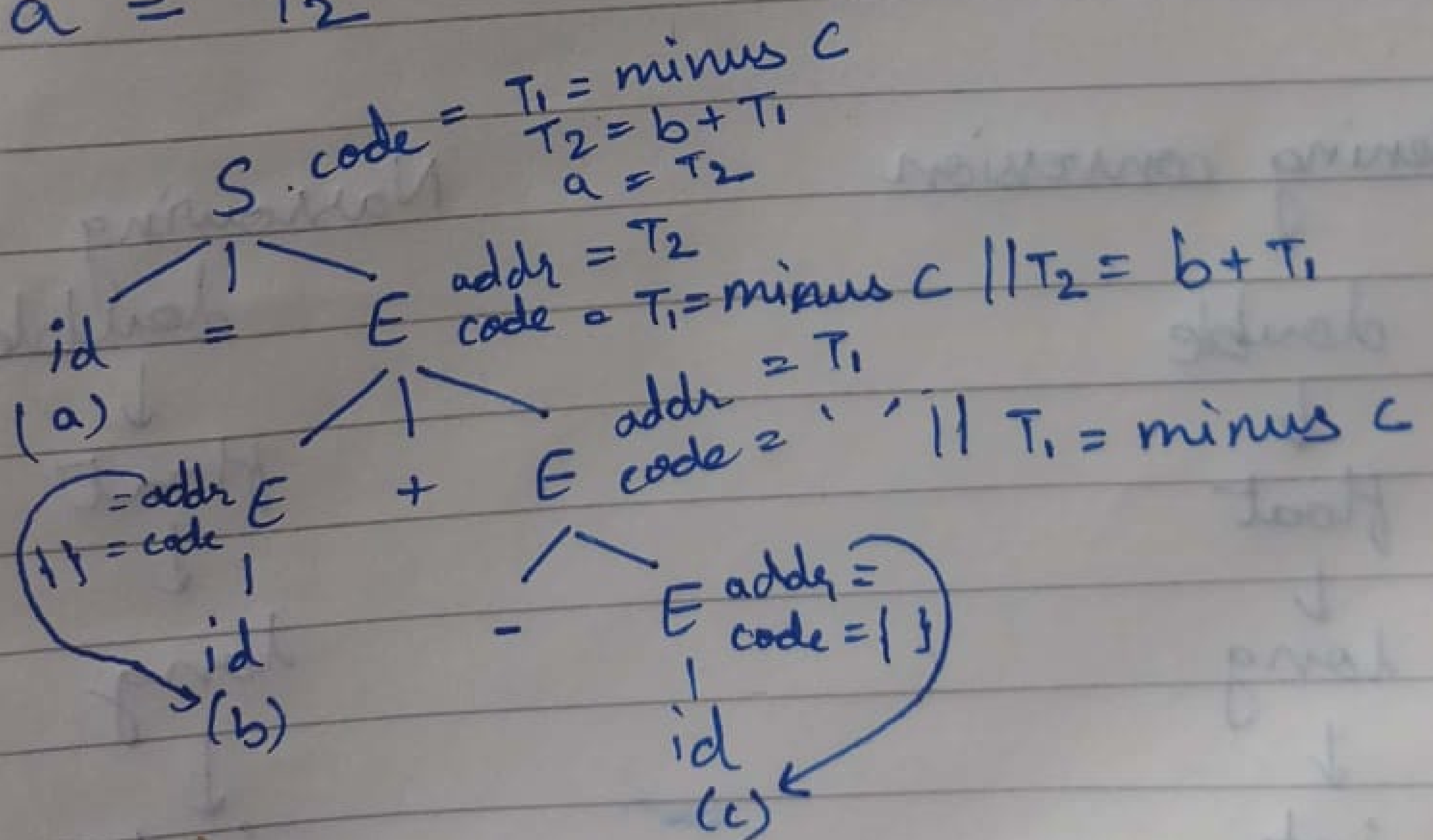
Q $a = b + -c$

T_1 = minus c

$T_2 = b + T_1$

$a = T_2$

→



Type Conversion

```
E = E1 + E2   { E.type = max (E1.type, E2.type);
                a1 = widen (E1.addr, E1.type, E.type);
                a2 = widen (E2.addr, E1.type, E.type);
                E.addr = new Temp ();
E1.code || E2.code || gen (E.addr `=' a1 `+' a2) }
```

```
Addr widen (Addr a, Type E, type w) {
    if (t = w)  return a;
    else if (t = integer and w = float) {
        temp = new Temp ()
        gen (temp `=' `(float)' a);
        return temp;
    }
    else error;
}
```

widening conversion

double
↓
float
↓
long
↓
int
↙ ↘
short   char
↓
byte

Narrowing conversion

double
↓
float
↓
long
↓
int
↙ ↓ ↘
char ↔ short ↔ bytes

now array expression are type converted of 3 addr?

from book

Short - Circuit Codes

if (x < 100 || (x > 200 && x != y)) x = 0;

might be translated to the code

```
if x < 100 goto L2
ifFalse x > 200 goto L1
ifFalse x != y goto L1
L2 : x = 0 ;
L1 :
```

Type

implicit → coersion

explicit → casting

$B \to B_1 || B_2$

$B \to B_1 \&\& B_2$

$B \to (B)$

$B \to E_1$ rep E_2

$E \to id$

Example:

$B \to B_1 || B_2$

E_1 rep E_2 || B_2

$x < 100$ || E_1 rep E_2

$x < 100$ || $x > 200$

$x < 100$ || x

inherited function B_1.true B.false

Backpatching :

nextinstr → variable
M. instr → addr. of instruction that follows M
makelist (i) →
merge (P_1, P_2) →
backpatch (P, i) →
B. truelist → It holds jump instruction
B. falselist →

1) $B \to B_1 \,||\, M B_2$ { backpatch (B_1. falselist, M. instr);
B. truelist = merge (B_1. trulist, B_2. truel
B. falselist = B_2. falselist; }

2) $B \to B_1 \,\&\&\, M B_2$ { backpatch (B_1. truelist, M. instr);
B. falselist = merge (B_1. falselist, B_2. falselist
B. truelist = B_2. truelist; }

3) $B \to \,!\, B_1$ { B. truelist = B_1. falselist;
B. falselist = B_1. truelist; }

4) $B \to (B_1)$ { B. truelist = B_1. truelist;
B. falselist = B_1. falselist; }

5) $B \to E_1$ rel E_2 { B. truelist = makelist (nextinstr);
B. falselist = makelist (nextinstr + 1);
gen ('if' E_1. address relop E_2. address
'goto ___');
gen ('goto ___'); }

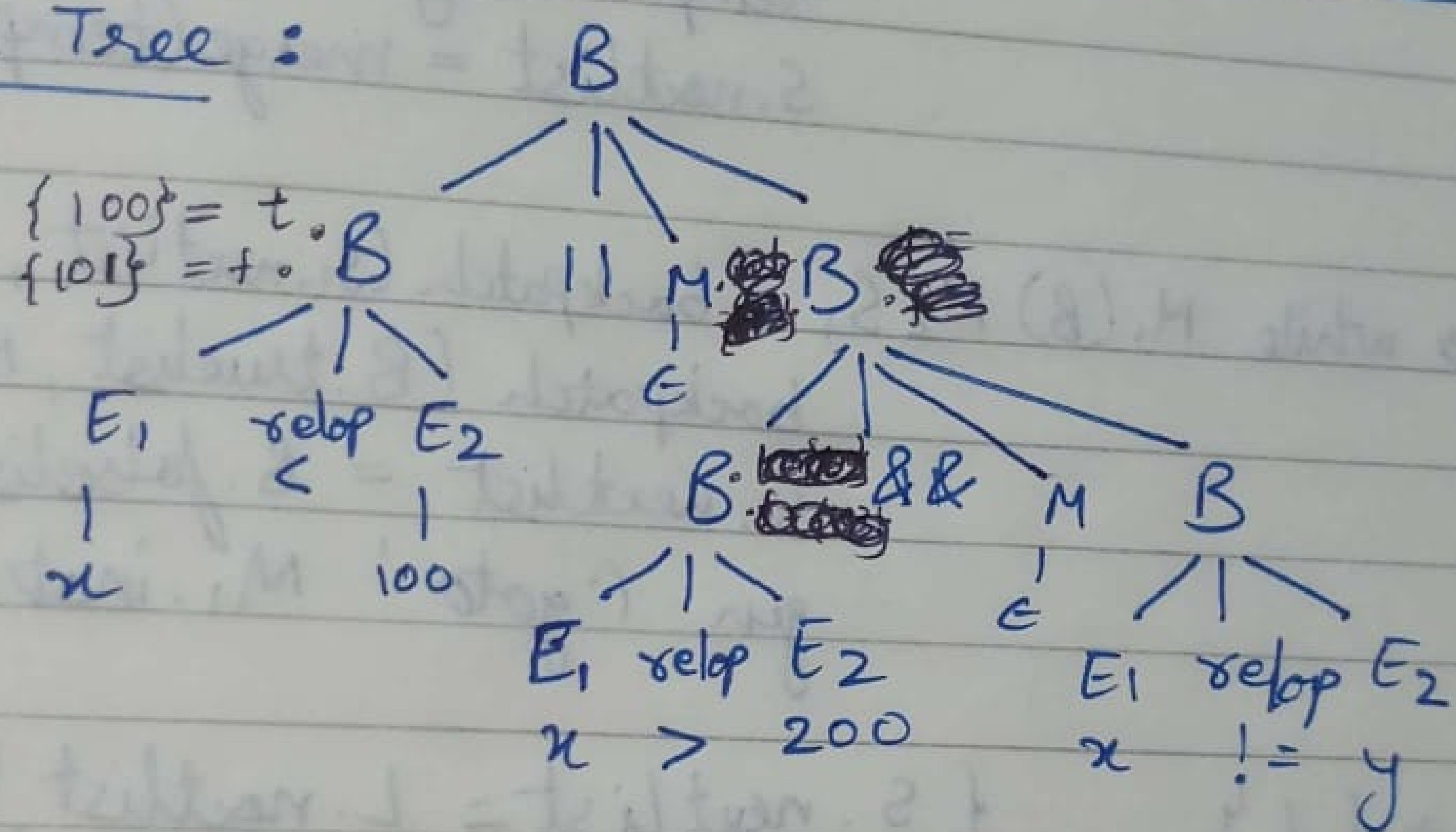
6) $B \to$ true { B. truelist = makelist (nextinstr);
gen ('goto ___'); }

7) B → false { B.falselist = makelist (nextinstr);
gen ('goto ___'); }

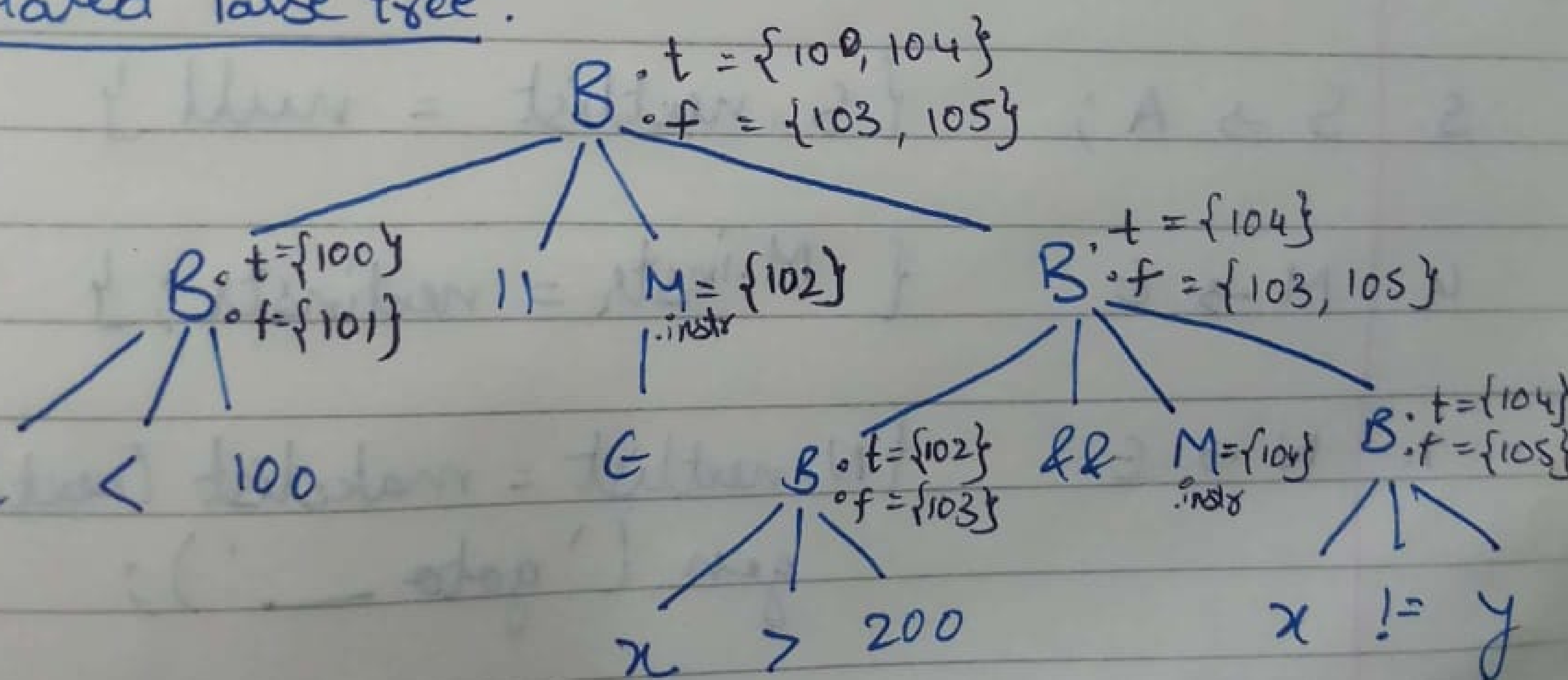
8) M → ∈ { M.instr = nextinstr; }

Annotated Parse tree for $x<100 \;||\; x>200 \;\&\&\; x!=y$

Parse Tree :



Annotated Parse tree :



100 if E.addr relop E2.addr goto ___ (x < 100)
(backpatch (B1.t, M.instr))
101 goto 102
102 if x > 200 goto 104
103 goto ___
104 if x != y goto ___
105 goto ___

Flow control statements

1. $S \rightarrow if\ (B)\ M\ S_1$ { backpatch ($B.truelist$, $M.instr$); $S.nextlist = S_1.nextlist$; }

2. $S \rightarrow if\ (B)\ M_1 S_1\ N\ else\ M_2\ S_2$ { backpatch ($B.truelist$, $M_1.instr$);
backpatch ($B.falselist$, $M_2.instr$);
temp = merge ($S_1.nextlist$, $N.nextlist$);
$S.nextlist$ = merge (temp, $S_2.nextlist$);

3. $S \rightarrow while\ M_1\ (B)\ M_2\ S_1$ { backpatch ($S_1.nextlist$, $M_1.instr$);
backpatch ($B.truelist$, $M_2.instr$);
$S.nextlist = B.falselist$;
gen ('goto' $M_1.instr$); }

4. $S \rightarrow \{L\}$ { $S.nextlist = L.nextlist$ }

5. $S \rightarrow A;$ { $S.nextlist$ = null }

6. $M \rightarrow \epsilon$ { $M.instr$ = nextinstr; }

7. $N \rightarrow \epsilon$ { $N.nextlist$ = makelist (nextinstr);
gen ('goto __');

8. $L \rightarrow L_1 M S$ { backpatch ($L_1.nextlist$, $M.instr$);
$L.nextlist = S.nextlist$; }

9. $L \rightarrow S$ { $L.nextlist = S.nextlist$; }

Scanned by CamScanner

nextlist - holds the list of next instruction

L - generate all statements within 'if' loop.