28/8/18

Operator Precedence Grammar

① You cannot have two non-terminal consecutively in RHS.

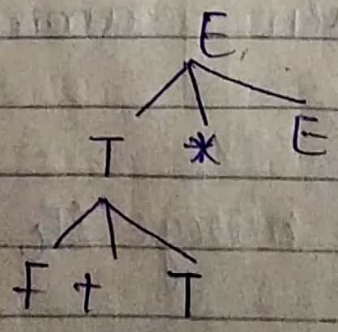
② It cannot have ϵ productions.

$E \rightarrow TF(X)$, $E \rightarrow \epsilon(X)$

Eg:-

$E \rightarrow T * E$
$T \rightarrow F + T$
$F \rightarrow id$

→ The operator that appears on the lower level have higher precedence.



So here, + has more precedence than *.

Associativity of * is Right
" " + is "

Scanned by CamScanner

size of operator precedence grammar
= (# terminals + 1)^2

↑ (Stack) \ → (Input)	id	+	*	$
id	not defined	>	>	>
+	<	>	<	>
*	<	>	>	>
$	<	<	<	accept

This is drawn accordy to bodmas table.

→ precidence of + in stack > + is Input

→ Precedence of $ is the least.
→ Precedence of id is the greatest.

→ Comparison of operator is always depend on grammar.

Q
A → B * A | B
B → B + C | C
C → D / C | D
D → id

Scanned by CamScanner

	*	+	/	id	$
*	>	<	<	<	>
+	>	>	<	<	>
/	>	>	>	<	>
id	>	>	>	nothing	>
$	<	<	<	<	accept

Precedence for this table:-

id > / > + > * > id.

→ Associativity is necessary when we have same operator in the stack & input.

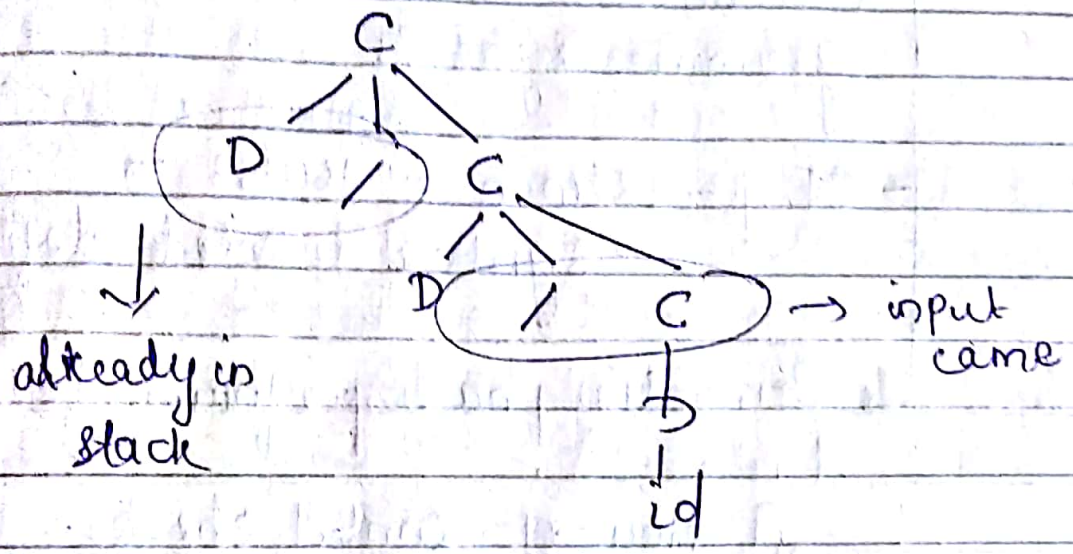
→ When right associative, we have to first shift i.e, we have to give precedence to the incoming operator (input).

→ When left associative, we have to first reduce i.e, we have to give precedence to the operator

Scanned by CamScanner

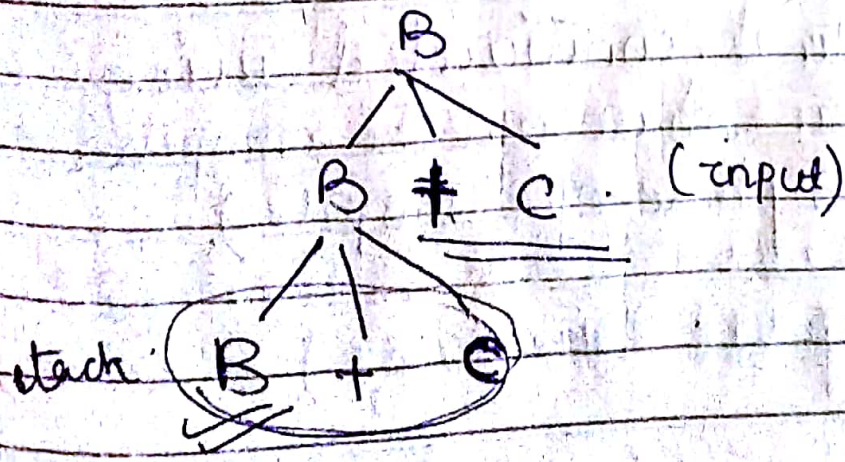
already in stack.

Here / is right associative.



So first we will evaluate input then stack. (first shift then reduce).

Here + is left associative.



So first we will evaluate stack then input. (first reduce then shift).

Scanned by CamScanner

31/8/18

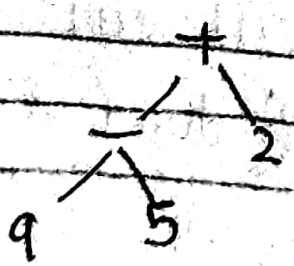
Semantic Analysis

- attributes are attached to the grammar symbols
- CFG + semantic rules = syntax directed definition
- Semantic phase performs following tasks :-
  1. Flow-of-control check.
  2. Type check.
  3. Scope resolution.
  4. Array out of bound check

* Symbol table is constructed using the analysis phase of the compiler (ie, first 3 phase of compiler).

Abstract Syntax Tree (AST)

9 - 5 + 2



Scanned by CamScanner

→ It hides the implementation part & just gives the overview struct.

Semantic errors:-

① Type mismatch.
② Accessing an out of scope variable
③ Formal & Actual parameter mismatch
④ Undeclared variable.
⑤ Multiple declaration of a variable in a scope.

Synthesized Attributes:

S → ABC
attribute/

→ The node can take value only by its children or node.

Inherited attributes:

→ node can take the value from parent, its siblings and the node itself.

S → ABCS.

Scanned by CamScanner

Eg; $E \rightarrow E + T$

We write attribute as :-

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

* Here .val → represent the attribute attached with that node

Here we have written E_1 just to differentiate that E can take value from someone else (which is E itself).

S-attributed SDT.

1. uses only synthesized attributes
2. bottom ↑ up parsing.
3. semantic rules are present at the end.

$E.val = E.val \,||\, T.val \,||\, +$

(+ → semantic rules written at the end.)

Scanned by CamScanner

L-attributed SDT

① uses synthesized attributes.
② uses inherited attributes but with the restriction that node cannot take value from its right sibling.
③ Depth-first order and left-to-right parsing

$S \rightarrow ABC$

A can only take value from S
B can only take value from S & A
C " " " " S, A and B

S can take value from ABC

SDD → Syntax directed definition

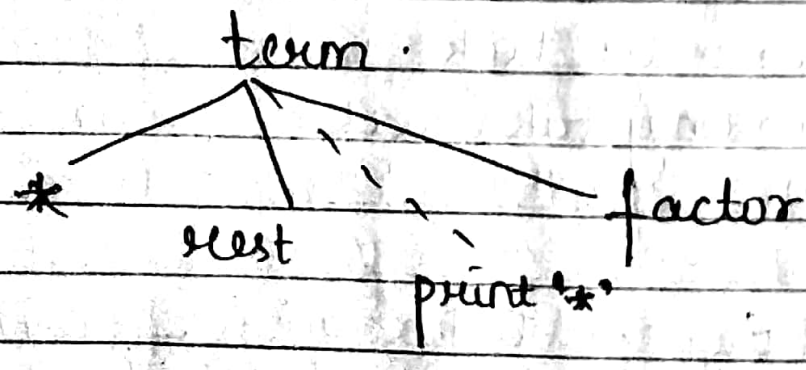
SDT → " " translation.

Scanned by CamScanner

In SDT, semantic rules can come in b/w but in SDD it always come in last.

→ In SDT we have to attach a different node for semantic rule as follow:-

term → * rest { print('*') } factor



→* Till now we have been writing as $E \rightarrow T + E$ but now in SDD and SDT we are writing attributes along with the flow of information in the parse tree.

Type checking

$E \rightarrow E_1 + E_2$ { if (E_1.type == E_2.type) && (E_1.type = int) then

Scanned by CamScanner

E.type = int
else
error (" ");

$E \rightarrow (E_1)$ { E.type = E_1.type }

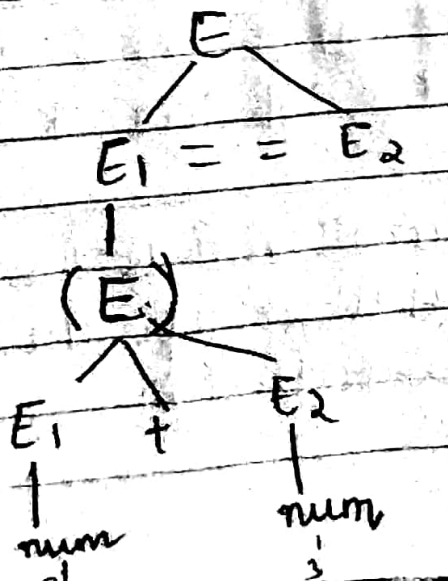
$E \rightarrow E_1 == E_2$ { if (E_1.type == E_2.type)
&& (E_1.type == int | bool)
then E.type = boolean
else error (" "); }

$E \rightarrow$ num { E.type = num.type }

$E \rightarrow$ true { E.type = boolean }

$E \rightarrow$ false { E.type = boolean }.

Eg:- (3+2) == 6



(Something like this will be drawn, not exactly :
→ Info. will flow from bottom to up

Scanned by CamScanner

3/8/18

Tutorial

Ambiguous Grammar.

$E \rightarrow E+E \mid E*E \mid a$

(operator grammar)

Resolve Ambiguity.

There will come a state,

$E \rightarrow E+E\cdot$
$E \rightarrow E\cdot+E$
$E \rightarrow E\cdot*E$

In Normal, there will be S/R conflict for S/R.

but for first Prod → Reduce
and Prod → Shift.] R/S conflict

As + is left associative so we will perform reduce.

Similarly there is a S/R for *.

As * precedence is higher.
so shift will perform first and then reduction.

Scanned by CamScanner

Similary

$E \rightarrow E * E \cdot$

$E \rightarrow E \cdot + E$

$E \rightarrow E \cdot * E$

Here also for + we have a S/R conflict As * precedence is higher than +, we will perform reduce.

For * there is also a conflict, and * left associative so we will perform reduce first.

→ So generally for SLR we have S/R conflict but if we associate operator operation then this can be resolved.

⇒ When we cannot tell from the grammar its associativity so we will use assumption (BODMAS).

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

$S \rightarrow iS \cdot eS$

$S \rightarrow iS \cdot$

Here also we have a S/R conflict.

* We generally give shift more importance than reduce, so we will do shift here.

Scanned by CamScanner

→ This is not an operator grammar, ie, dangling if statement so here Shift ⊙ has higher precedence than reduce.

∴ Because our objective is to take the higher length input so we perform shift operation.

* There are different methods to cope up errors

(1) panic error → as soon as it come leave every thing

(2) Statement error → If at a particular input operator error comes then we leave some input character till the error is gone.

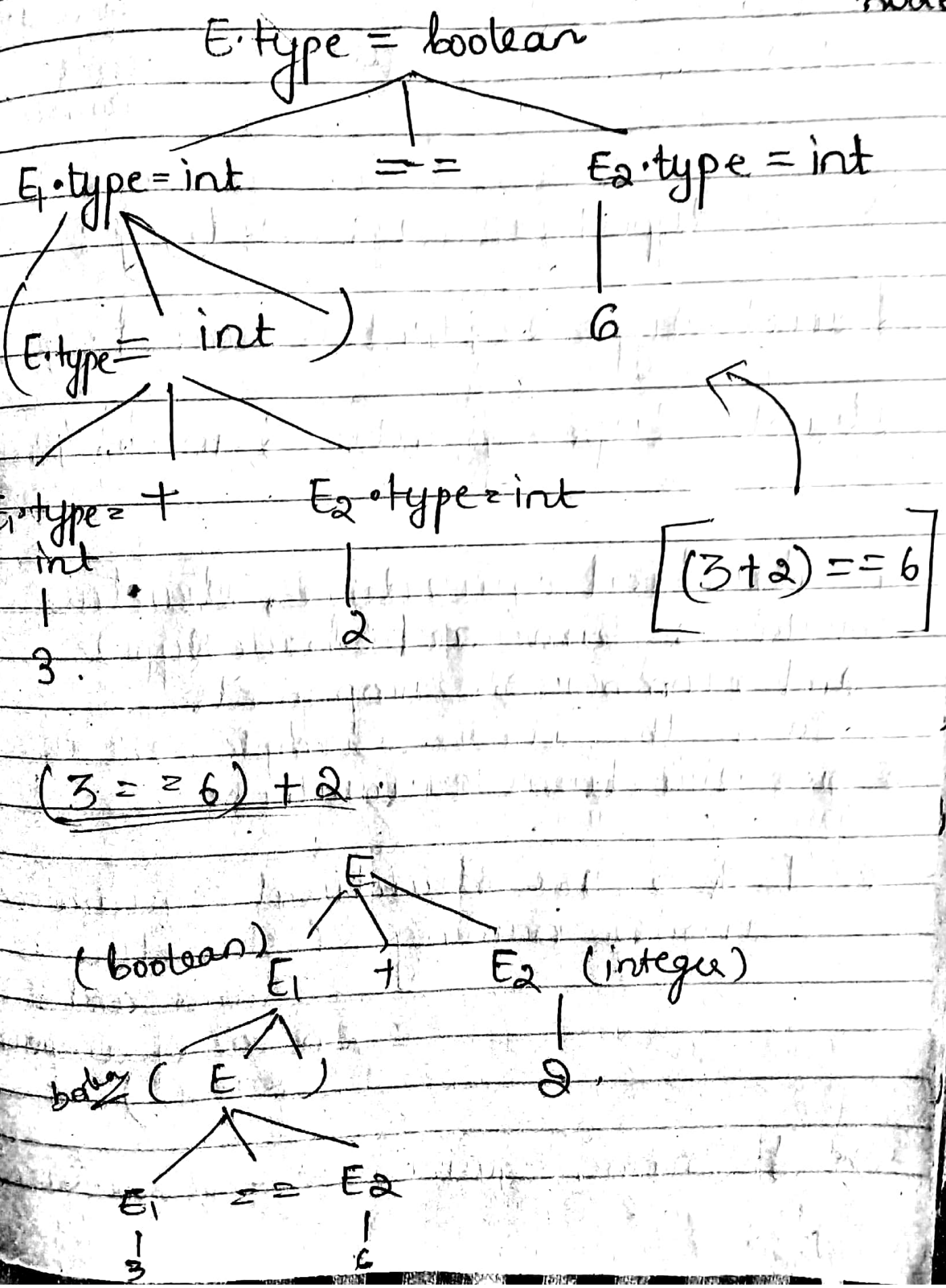
(3) Pop every element from the stack and then start from that particular I/P character and where there is no error.

We will study this part later (Not for exam)

Scanned by CamScanner

Annotated Parse Tree

A parse tree showing attribute value at each node



(3 == 6) + 2

Scanned by CamScanner

Type Equivalence :

- Structural equivalence
- Name equivalence
- Structural equivalence under naming

```
struct type1 { int a ; int b; };
struct type2 { int c ; int d; };
struct type3 { int a ; int b; };
```

→ In structural equivalence, structure should be same (basic type) and number of components.
So in the above example all three are structural equivalence.

→ 1 & 3 are structural equivalence under naming (structure is same and the name of components same).

→ In name equivalence → structure

can be different but name should be same.

Eg:- Struct type 1 {int a ; int b};

array type 1 ← name same.

Type Conversion.

Type coercion ← Implicit

(Eg:- upgrading int to float and then peaform operations).

(changing char to int, there will be no loss of information).

Explicit.

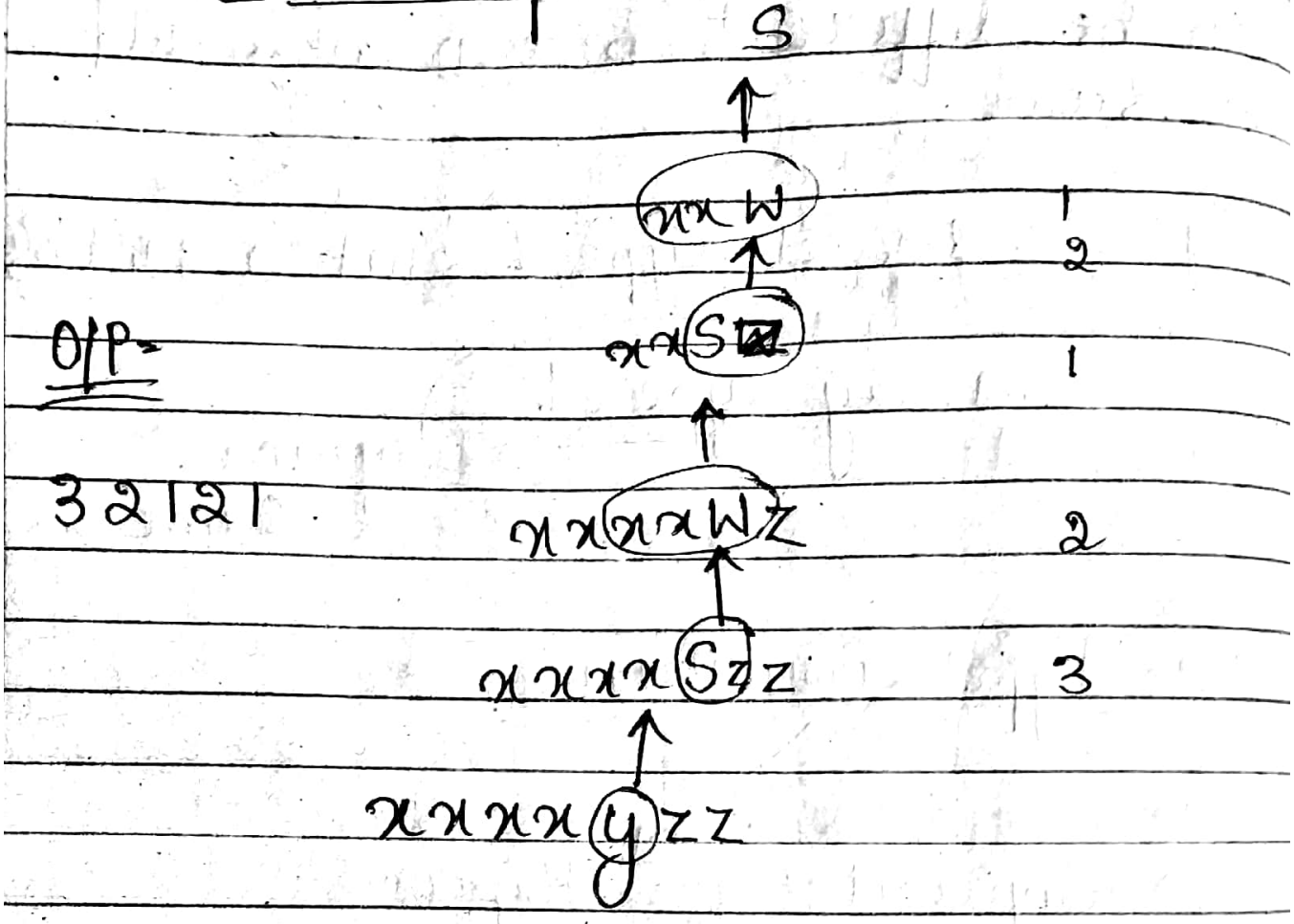
Eg:- changing int to char (there will be loss of information)

Eg:-

S → x x W { print ('1') }
S → y { print ('3') }
W → S z { print ('2') }.

I/P = x x x x y z z .

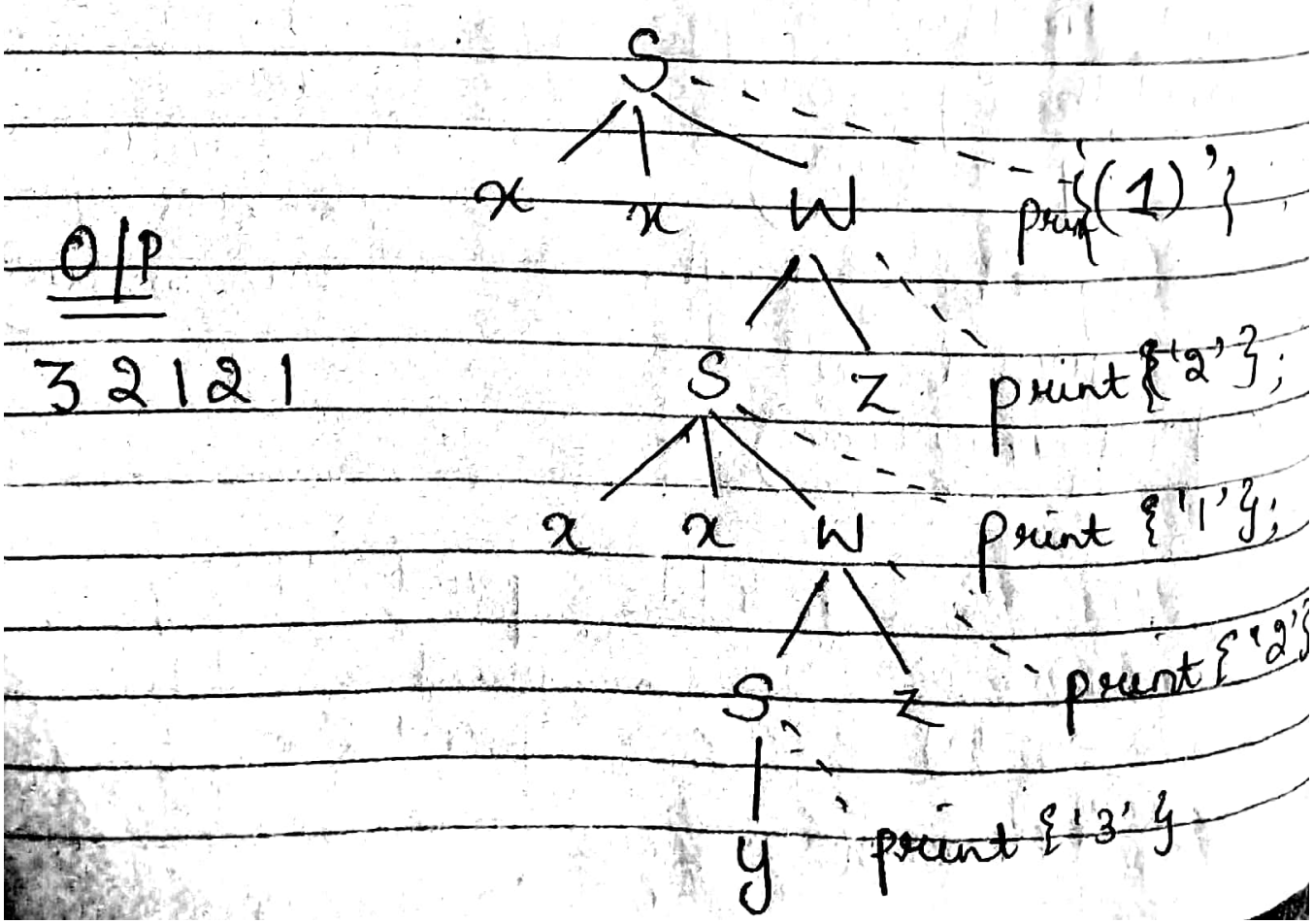
Bottom-Up Parser



O/P:

32121

Top-Down

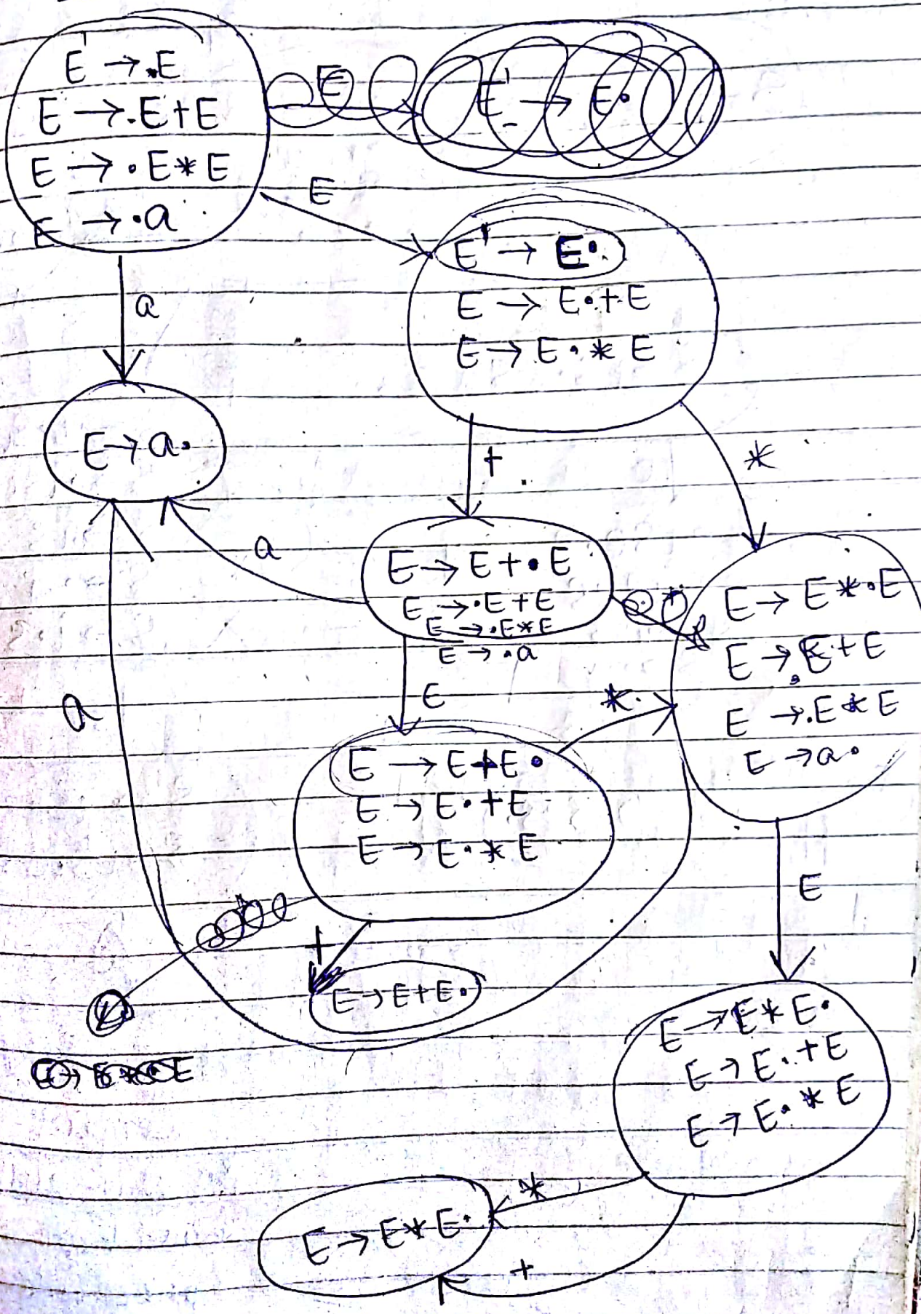


O/P

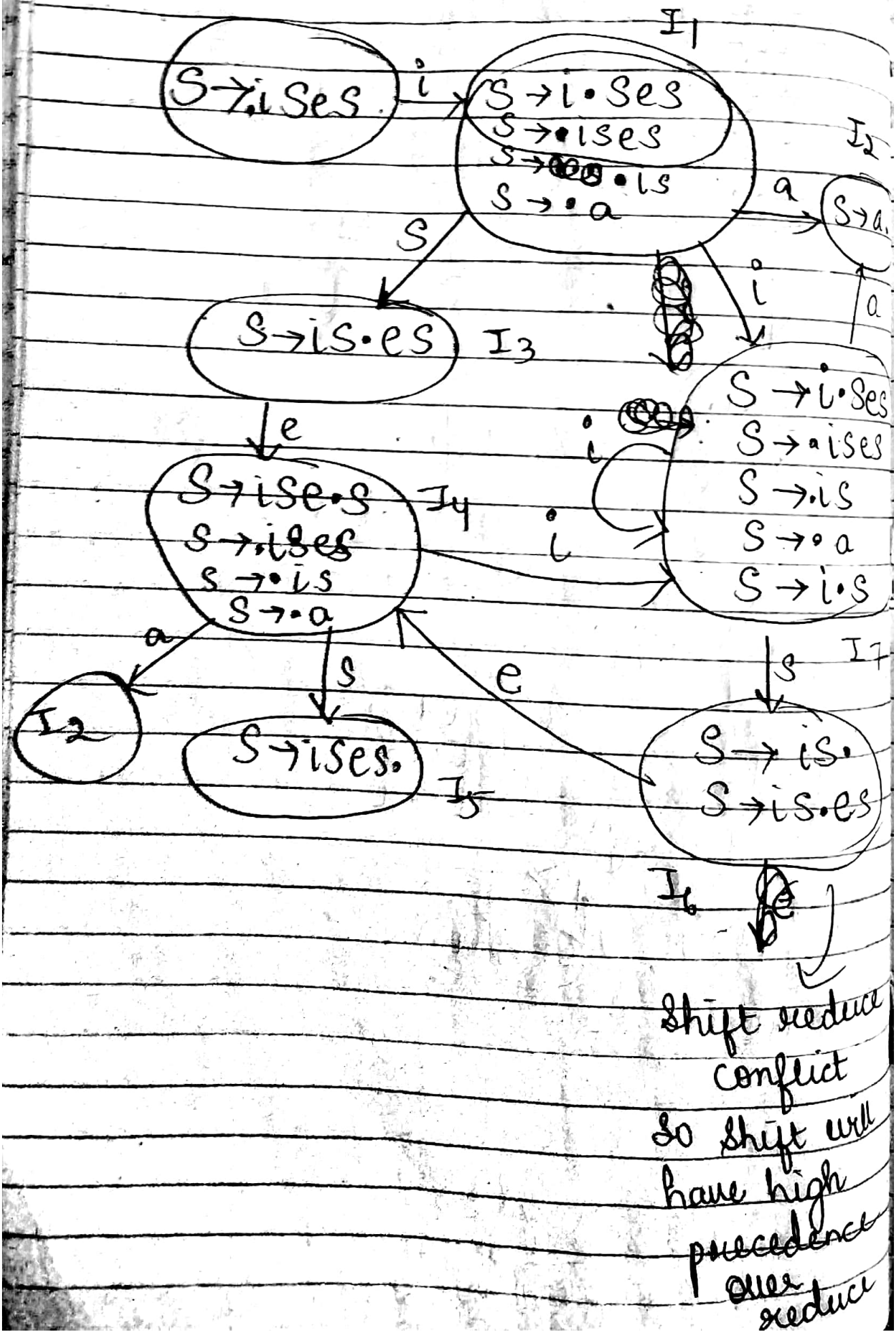
32121

Scanned by CamScanner

Tutorial Ques:



Scanned by CamScanner



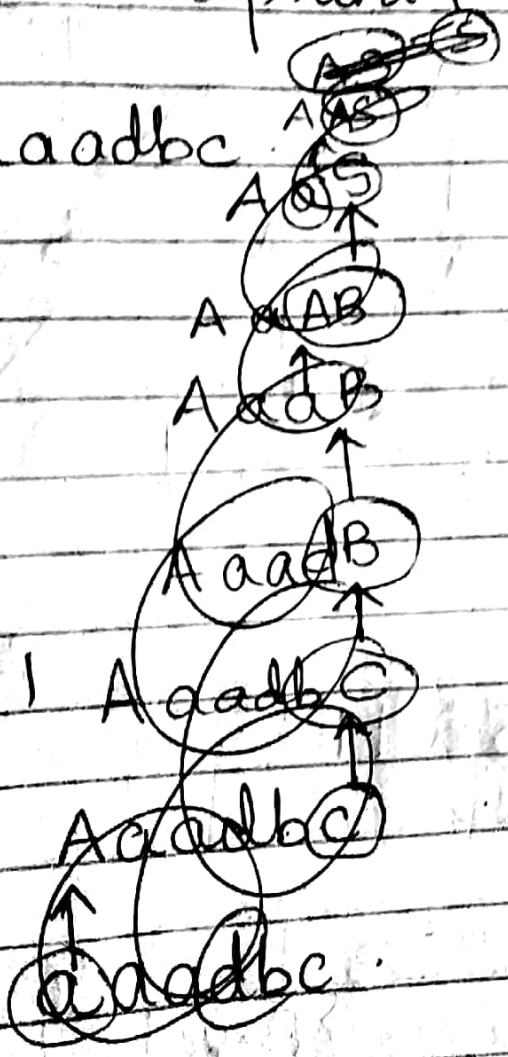
Scanned by CamScanner

6/9/18

Q
$S \rightarrow AS$ { print ("1") }
$S \rightarrow AB$ { print ("2") }
$A \rightarrow a$ { print ("3") }
$B \rightarrow bC$ { print ("4") }
$B \rightarrow dB$ { print ("5") }
$C \rightarrow c$ { print ("6") }.

I/P aaadbc

O/P = 33364521 1

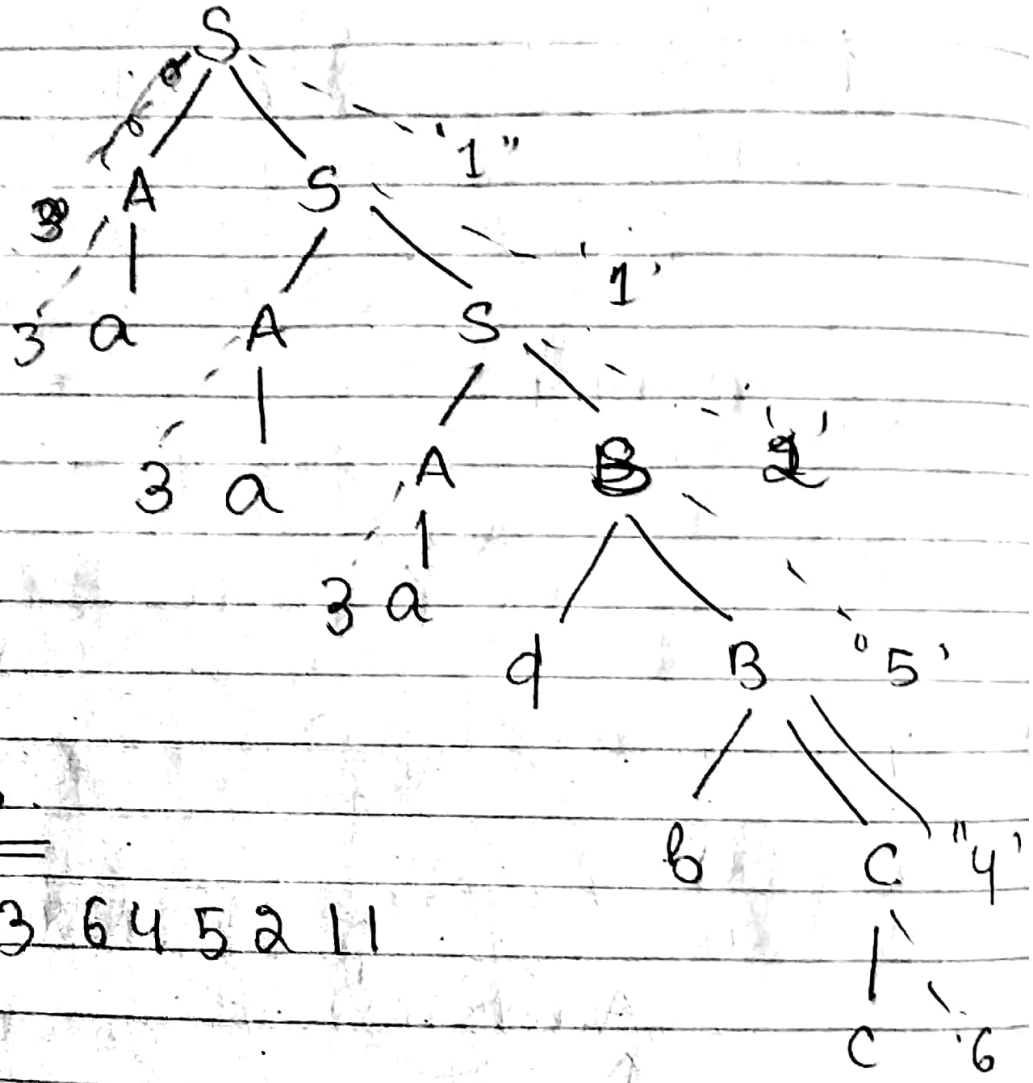


Bottom-Up

S
AS 1
AAS 1
AAAB 2
AAAdB 5
AAAdbC 4
AAAdbc 6
AAadbc 3
Aaadbc 3
aaadbc 3

Scanned by CamScanner

Top down



O/P

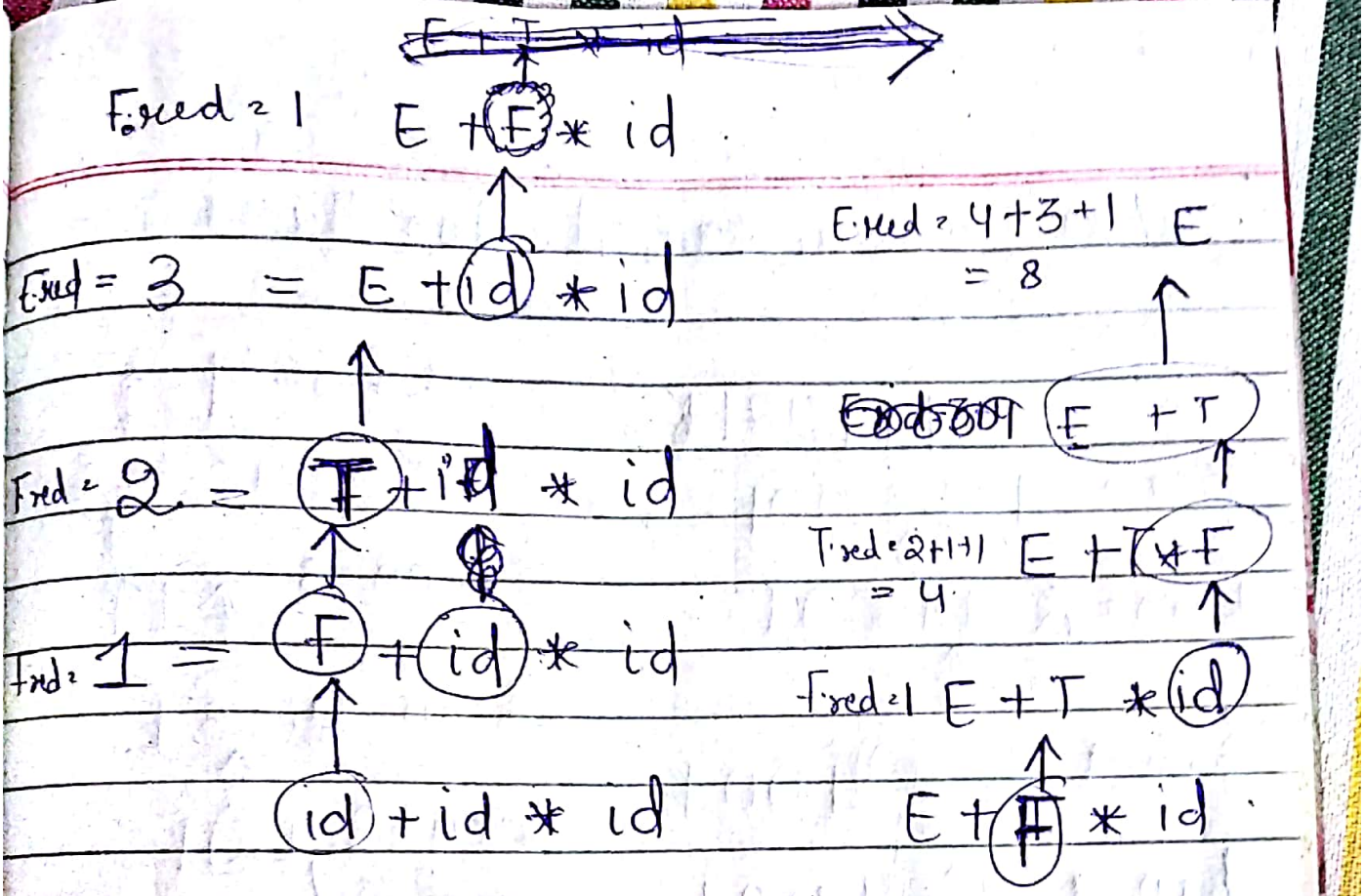
3 3 3 6 4 5 2 1 1

Q

$E \to E+T$	$\{E.red = E.red + T.red + 1\}$
$E \to T$	$\{E.red = T.red + 1\}$
$T \to T*F$	$\{T.red = T.red + F.red + 1\}$
$T \to F$	$\{T.red = F.red + 1\}$
$F \to id$	$\{F.red = 1\}$

Q Calculate the no. of reductions used in the parsing of "id + id * id".

Scanned by CamScanner



We have used 8 reductions.

Now we will perform reduction using semantic rules. → we are getting 8 reductions.

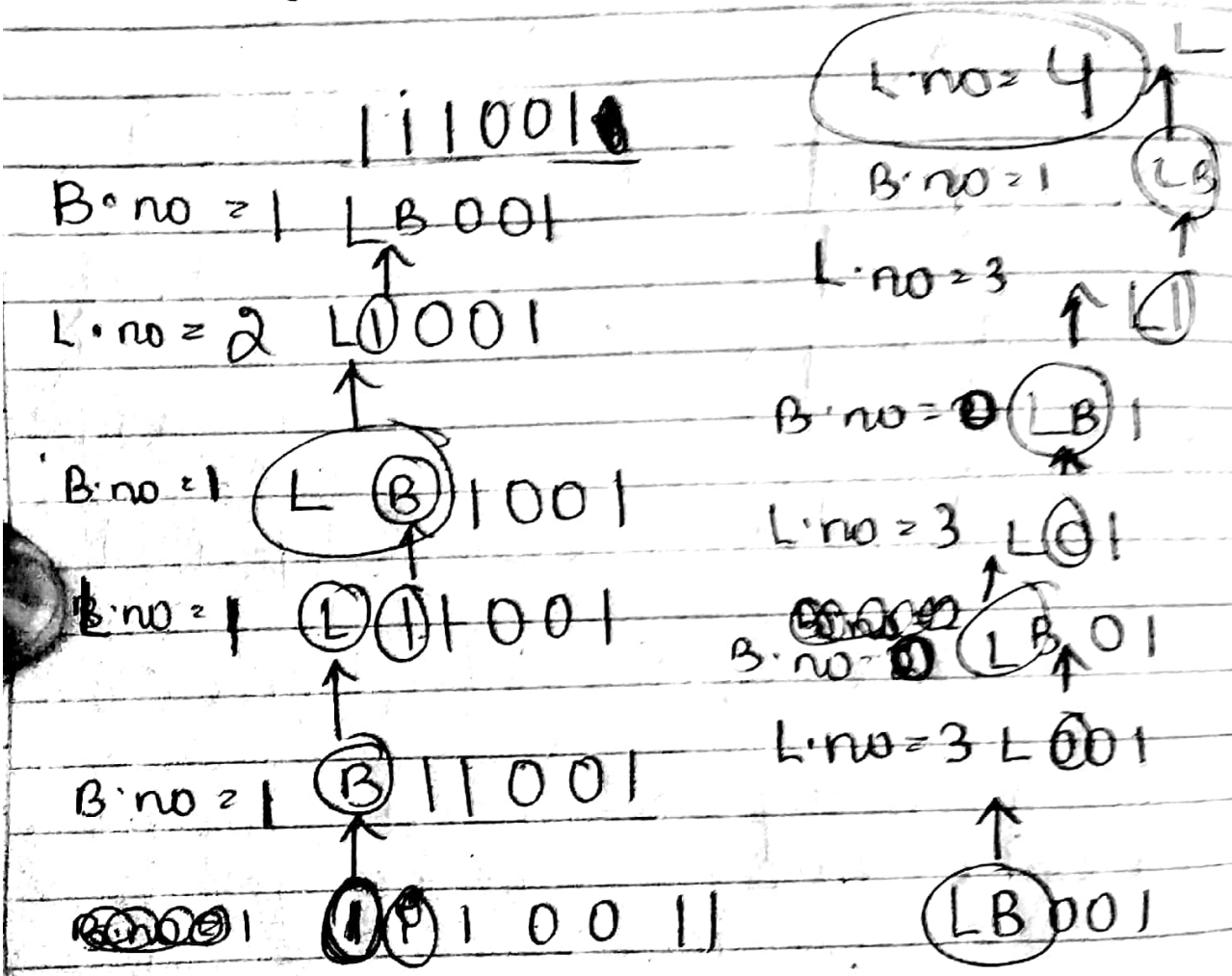
∴ This is S-attributed SDT.

Q Draw SDT to count the no. of 1's in the grammar :-

$S \to L$	{ B.no = L.no }
$L \to LB$	{ L.no = L.no + B.no }
$L \to B$	{ B.no = B.no + 1 }
$B \to 0$	{ B.no = 0 }
$B \to 1$	{ B.no = 1 }

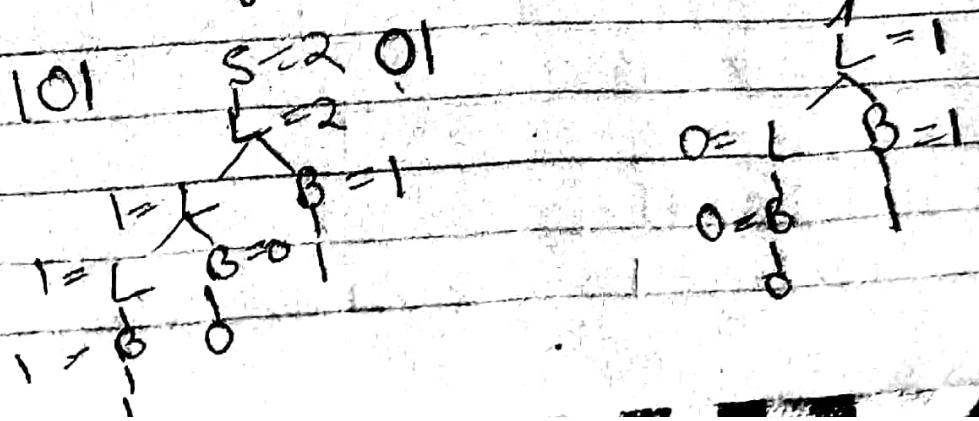
Scanned by CamScanner

Also give an example for the same.



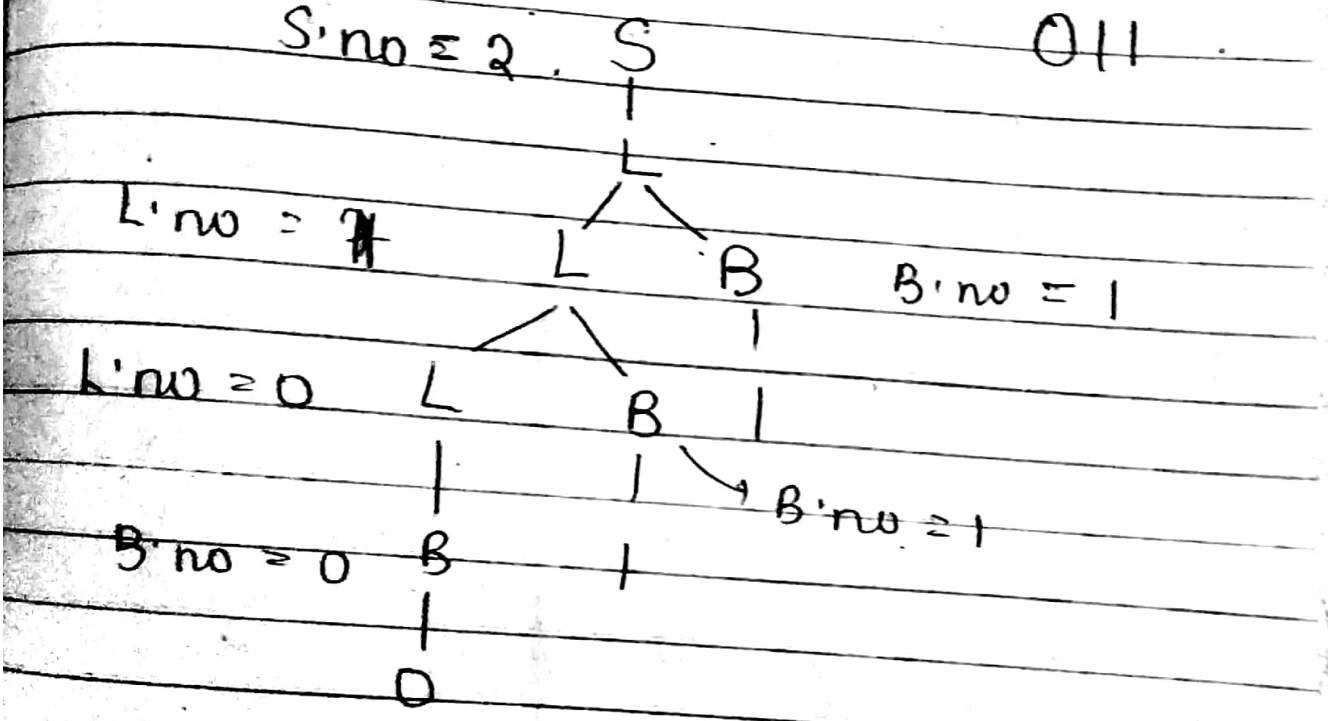
Total number of 1's = 4.

According to annoted parse tree no of 1 = 4.



Scanned by CamScanner

Annoted parse tree can be bottom-up & top-down. But preferably draw top-down.



Scanned by CamScanner