UNIT - 5

SYNTAX - DIRECTED TRANSLATION

CONTENTS

→ Syntax directed dfinations
→ Evaluatn orders for SDD's
→ Applicatn of SDT
→ SDT schemes.

Syntax Directed Dfinations:

A syntax directed dfination in a context free grammers with attributes & rules. Attributes are associated with grammer symbols & rules with productions. If 'x' is a symbol, 'a' is one of attributes then we write X.a to denote value of a at a particular parse tree node labelled X

* Attributes may be of many kinds: numbers, types, tables, references, strings, etc...

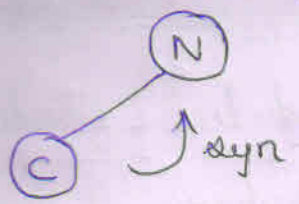
* 2 types of attributes: 1) Synthesized attr
2) Inherited attr

* Synthesized attr: A synth attr for a nonterminal A at a parse tree node N is dfined by a semantic rules associated with the production at N. Note that the production must have A as its head. A synthesized attr at node N is dfined only in terms of attribute values at the children of N & at N itself.

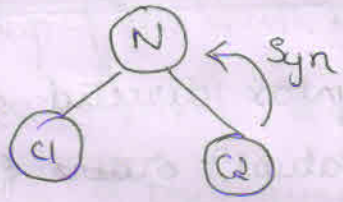
* Inherited attr: A inherited attr for a nonterminal B at a parse tree node N is defined by a semantic rule associated with the producth at the parent of N. Note that the producth must have B as a symbol in its body. An inherited attr at node N is defined only

in terms of attr values at N's parent, N itself & N's sibling

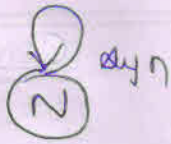
NOTE: ① Synthesised ⟨N → Node under consideration, C → child⟩



Case(i) Single child

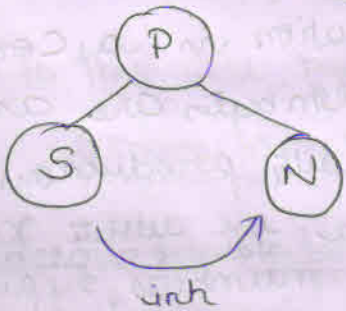


Case(ii) Rightmost child

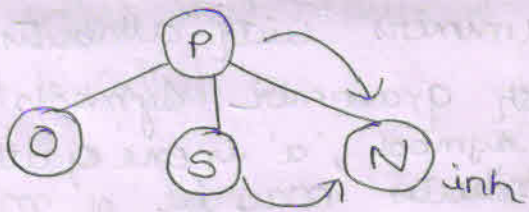


Case(iii) No child ⇒ itself.

② Inherited ⟨P → parent, N → node under consideration, S → Sibling, O → operator.⟩



Case(i) Sibling



Case (2): inherited from both parent & sibling.

③ Terminals can have Synthesized attributes but not inherited attributes.

* Attr of terminals have lexical value that are supplied by lexical analyser.

* Types of SDD: ① S Attributed SDD
② L Attributed SDD.

① S-Attributed SDD

* A SDD that involves only synthesised attr then it is called S-attributed SDD

* In S-attributed SDD, each rule computes

an attribute for the i^{th} terminal at the head of a production from attribute taken from body of production.

* S-attributed SDD can be implemented naturally in conjunction with an LR parser / bottom up parser.

* Annotated parse tree

A parse tree showing the value(s) of attribute(s) is called an annotated parse tree.

* It is used in bottom-up parser
* Order of evaluation is postorder traversal.
* Example of S-attributed SDD.

Production	Semantic rules
1) $L \to En$	$L.val = E.val$
2) $E \to E1 + T$	$E.val = E1.val + T.val$
3) $E \to T$	$E.val = T.val$
4) $T \to T_1 * F$	$T.val = T_1.val * F.val$
5) $T \to F$	$T.val = F.val$
6) $F \to (E)$	$F.val = E.val$
7) $F \to digit$	$F.val = digit.lexval$

(2) L-Attributed SDD

* Example of mixed attributes / L-attributed SDD

Production	Semantic rules
1) $T \to FT'$	$T'.inh = F.val$ $T.val = T'.syn$
2) $T' \to *FT'$	$T_1'.inh = T'.inh * F.val$ $T.syn = T_1'.syn$
3) $T' \to \epsilon$	$T'.syn = T'.inh$
4) $F \to digit$	$F.val = digit.lexval$

* A SDD which has both synthesised & inherited attributes is called as L-attributed SDD.
* It is used in top down parsers.
* Order of evaluation is topological sorting.

Evaluating Orders For SDD's

* A dependency graph is used to determine the order of computation of attributes.
* While an annotated parse trees shows the values of attributes, a dependency graph helps us to determine how those values can be computed.

DEPENDENCY GRAPHS

A dependency graph predicts the flow of information among the attribute instances in a particular parse tree. An edge from one attribute instance to an other means that the value of first is needed to compute the second.

1) For each parse tree node, say node X, the dependency graph has a node for each attribute associated with X.

2) If a semantic rule (defines) associated with a product 'p' defines the value of synthesized attribute A.b in terms of value of X.c then the dependency graph has an edge from X.c to A.b

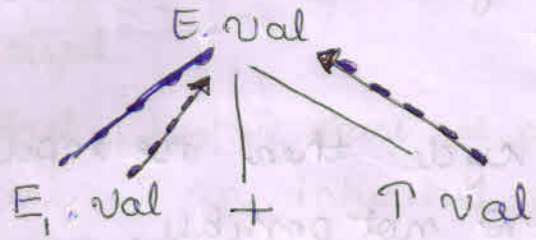
3) If a semantic rule associated with a product 'p' defines the value of inherited attribute B.c in terms of value of X.a then the

dependency graph has edge from X.c to B.c

Eg1 :

Production	Semantic Rule
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$

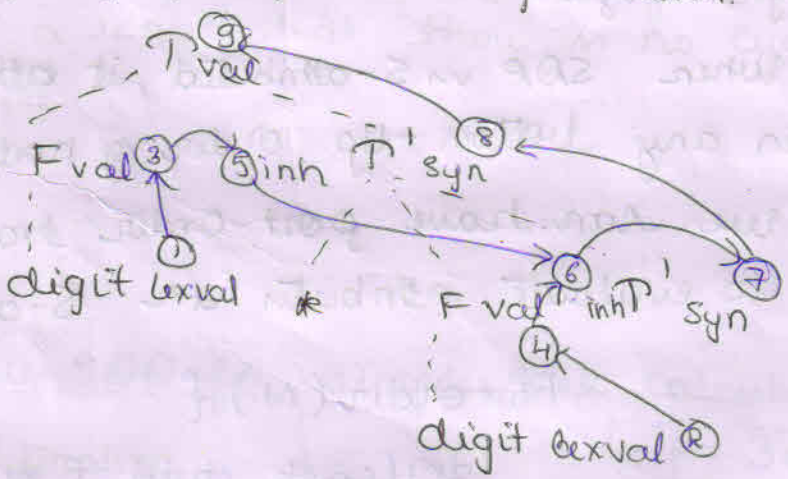
fig: E.val is synthesized from E_1.val & T.val



Eg2 :

Production	Semantic Rule
$T \rightarrow FT'$	$T'.inh = F.val$
$T' \rightarrow *FT'$	$T.val = T'.syn$
$T' \rightarrow \epsilon$	$T'.inh = T'.inh * F.val$
$F \rightarrow digit$	$T.syn = T'.syn$
	$T'.syn = T'.inh$
	$F.val = digit.lexval$

fig: Dependency graph for above production.



Ordering the evaluation of attributes

* If the dependency graph has an edge from node M to N, then the attribute corresponding to M must be evaluated before attribute of N.

* Thus the only allowable orders of evaluation are those sequences of nodes $N_1, N_2, \ldots N_5$ if if there is an edge of the dependency graph from N_i to N_j then $i < j$.

* Such an ordering is called a topological sorting of a graph

* If there is any cycle then no topological sorts, i.e., evaluation of SDD not possible.

* Eg: For dependency graph for (Eg 2) in previous page
topological sorting: 1, 2, 3, 4, 5, 6, 7, 8, 9

(or)

1, 3, 5, 2, 4, 6, 7, 8, 9

S-Attributed Definations

→ An SDD is S-attributed if every attribute is synthesized.

→ When SDD is S-attributed, it attributes evaluated in any bottom-up order of nodes of parse trees.

→ we can have post-order traversal of parse tree to evaluate attributes in S-attributed defination.

```
Postorder(N) {
    for(each child C of N, from the left)
        postorder(C);
    evaluate the attributes associated with
    Node N;
}
```

→ S-Attributed definations can be implemented during bottom up parsing without the need to explicitly create parse trees.

L-Attributed Definations

* A SDD is L-attributed if the edges in dependency graph goes from Left to Right but not from Right to left.
* More precisely, each attribute must be either
  - → Synthesized.
  - → Inherited, but if there is a production $A \to X_1 X_2 \cdots X_n$, & there is an inherited attribute $X_i.a$ Computed by a rule associated with this production then the rule may only use:
    (a) Inherited attributes associated with the head A.
    (b) Either inherited or synthesized attr associated with the occurance of symbols $X_1, X_2 \cdots, X_{i-1}$ located to the left of X_i.
    (c) Inherited/Synthesized attr associated with the occurance of X_i itself but only in such a way that there is no cycle in the graph.

PROBLEMS

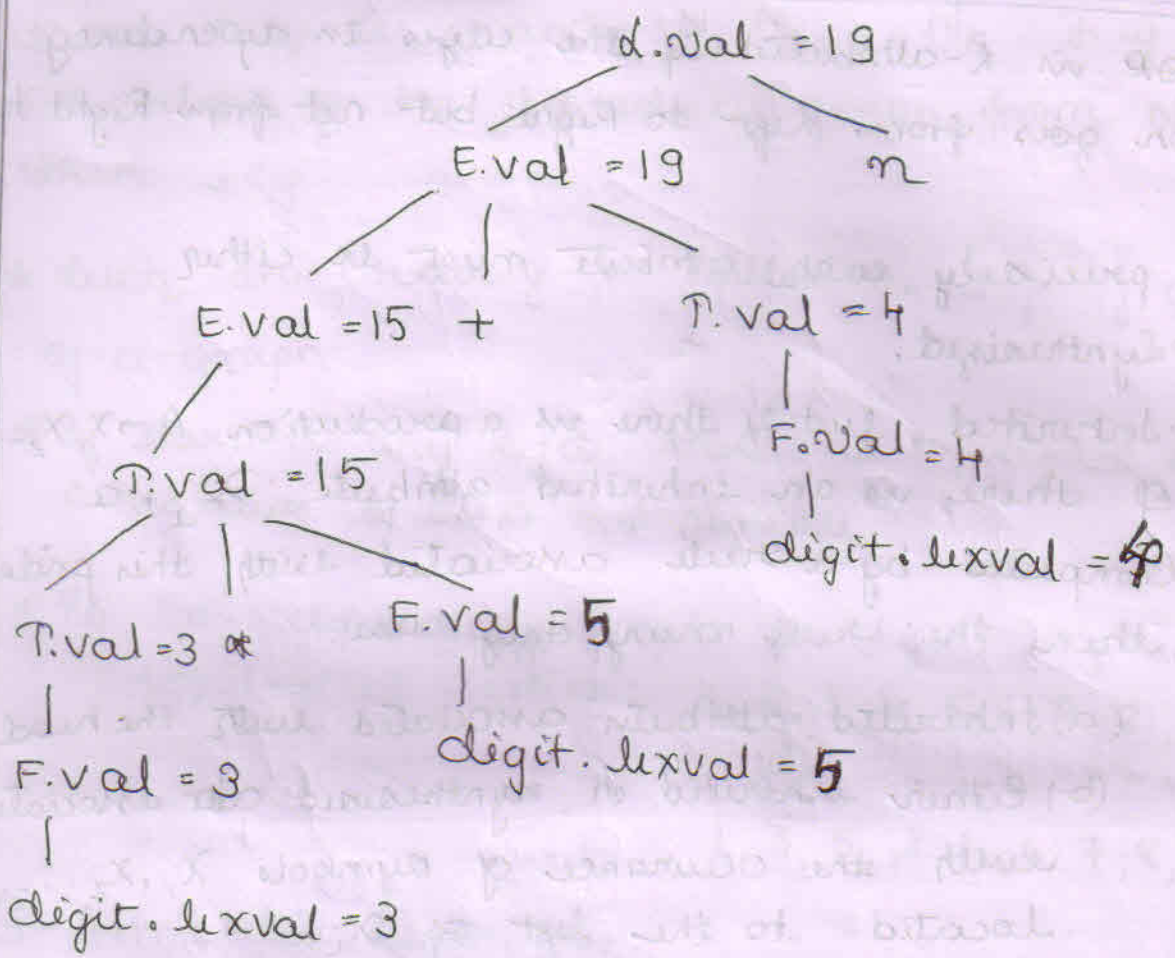
(5.1) Write a SDD for simple desk calculator

Sol: SDD difination

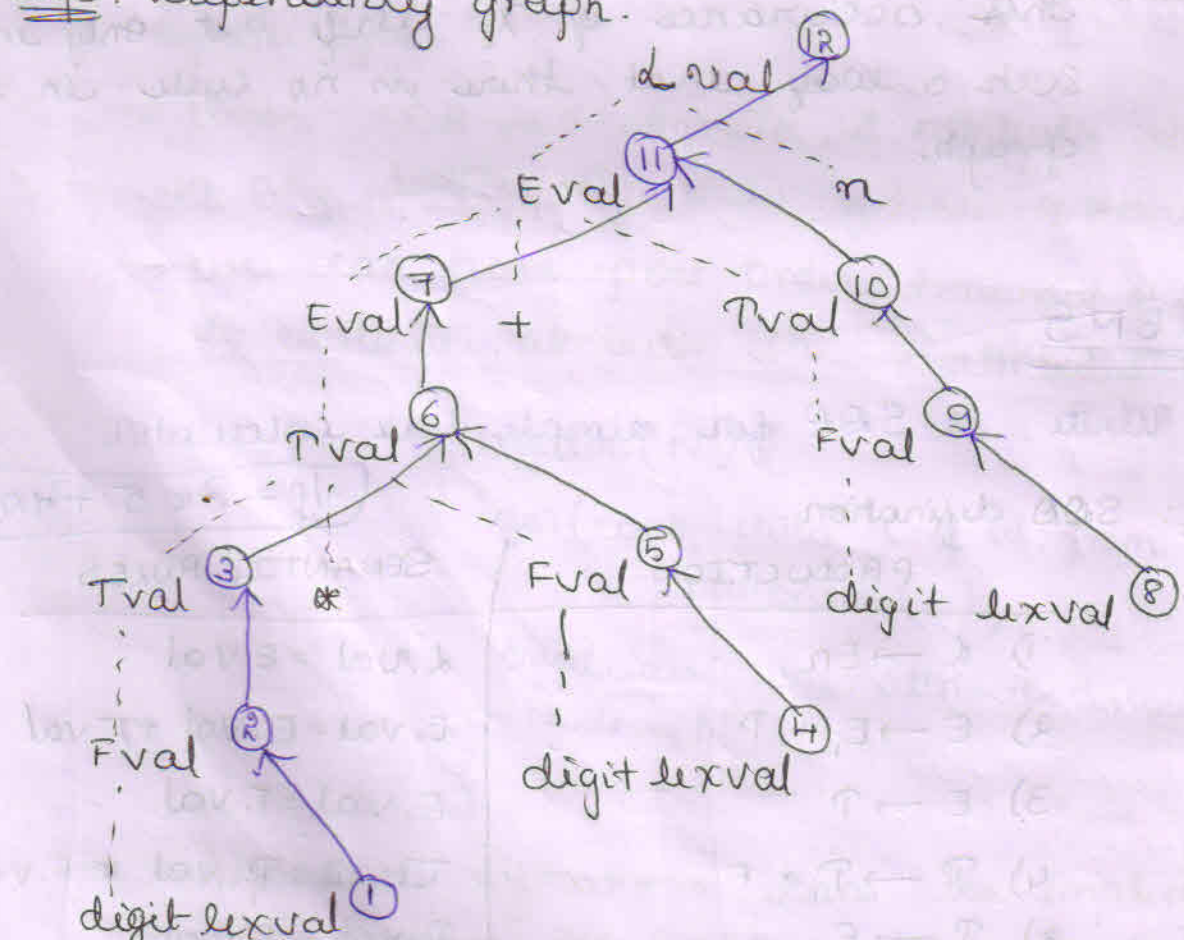
i/p = 3 * 5 + 4 n

PRODUCTION	SEMANTIC RULES
1) $L \to En$	L.val = E.val
2) $E \to E_1 + T$	E.val = E_1.val + T.val
3) $E \to T$	E.val = T.val
4) $T \to T_1 * F$	T.val = T_1.val * F.val
5) $T \to F$	T.val = F.val
6) $F \to (E)$	F.val = E.val
7) $F \to$ digit	F.val = digit.lexval

Step2: Annotated Parse tree



Step3: Dependency graph.



Step4: Topological Ordering : (1) (2) (3) (4) (5) (6) (7) (8) (9) (10) (11) (12)

Write the SDD & construct annotated parse tree, dependency graph.

a) $(3+4)*(5+6)n$

b) $1*2*3*(4+5)n$

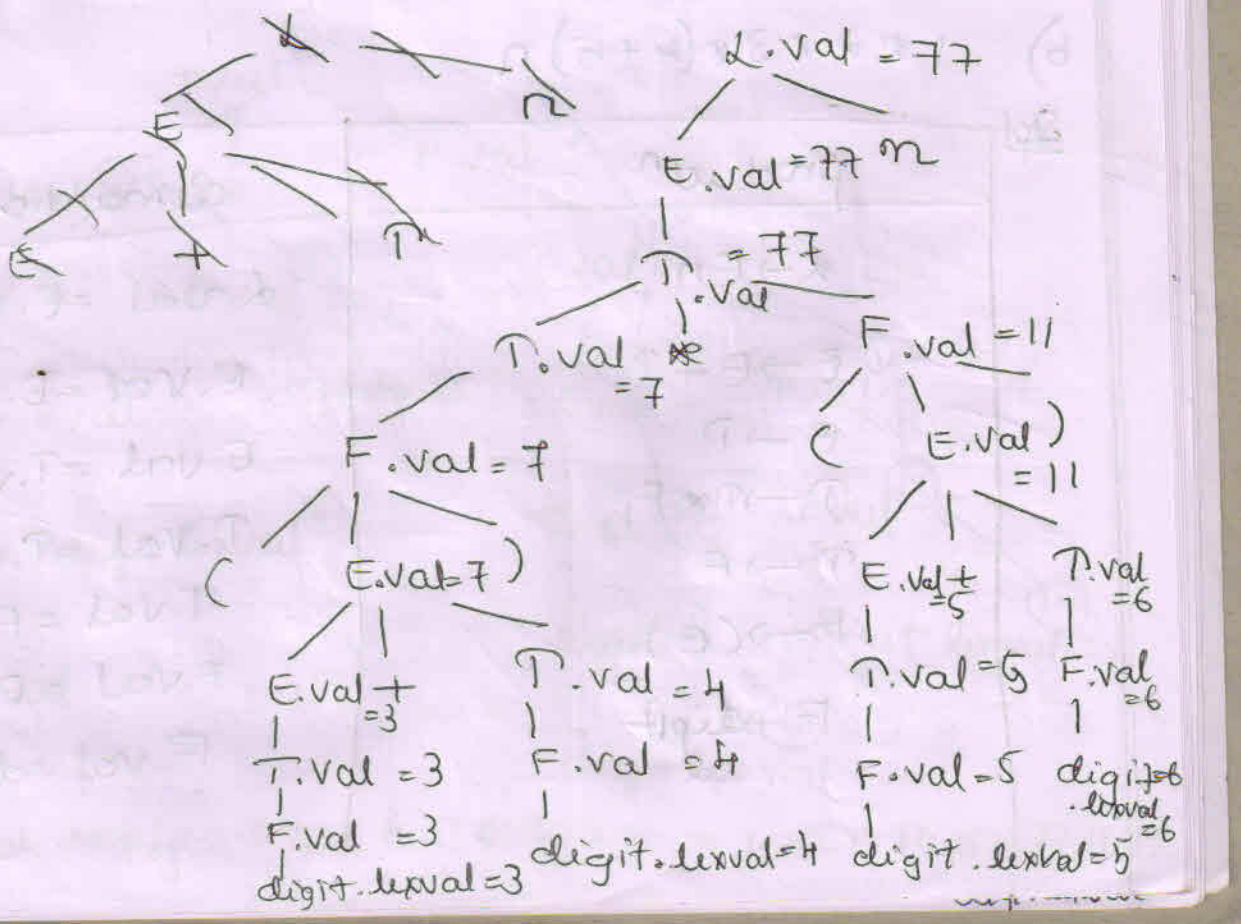
c) $(9+8*(7+6)+5)*4n$

a) $(3+4)*(5+6)n$

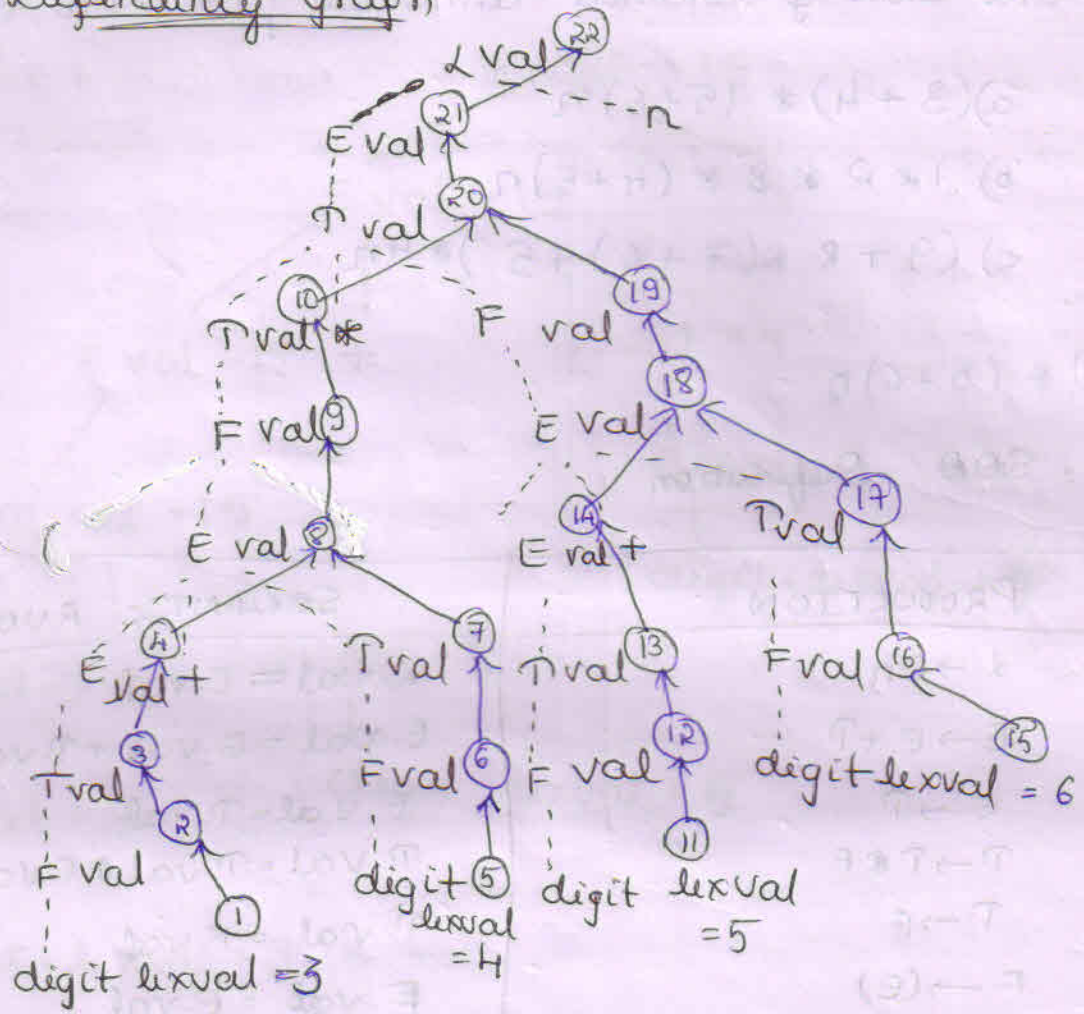
Step 1: SDD Defination

PRODUCTION	SEMANTIC RULES
$L \to En$	$L.val = E.val$
$E \to E+T$	$E.val = E.val + T.val$
$E \to T$	$E.val = T.val$
$T \to T*F$	$T.val = T.val * F.val$
$T \to F$	$T.val = F.val$
$F \to (E)$	$F.val = E.val$
$F \to digit$	$F.val = digit.lexval$

Step 2: Annotated Parse tree



Step3: Dependency Graph



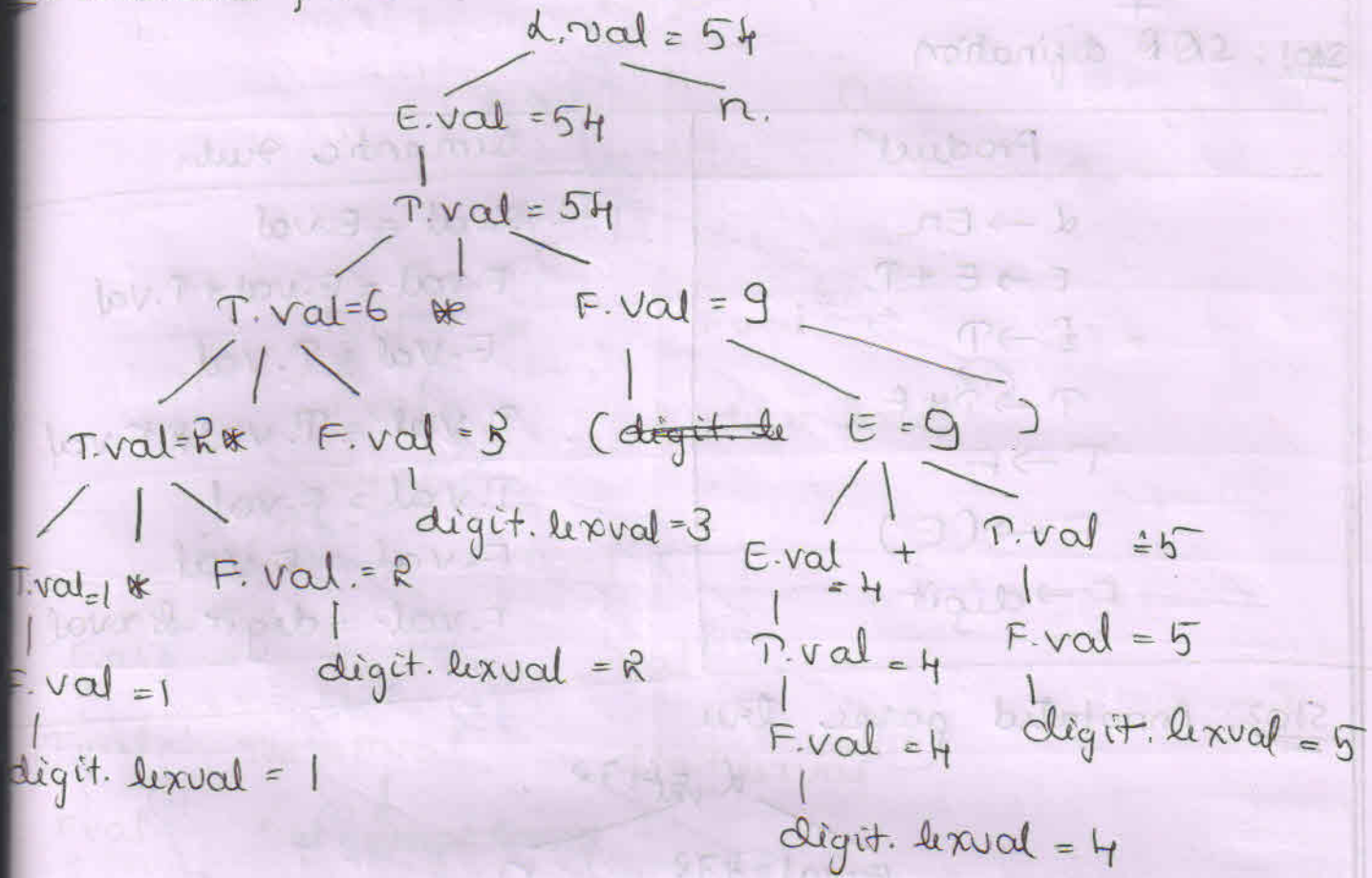
Step4 = Topological sorting : (1) (2) (3) (4) (5) (6) (7) (8) (9) (10) (11) (12) (13) (14) (15) (16) (17) (18) (19) (20) (21) (22).

b) 1 * 2 * 3 * (4 + 5) n

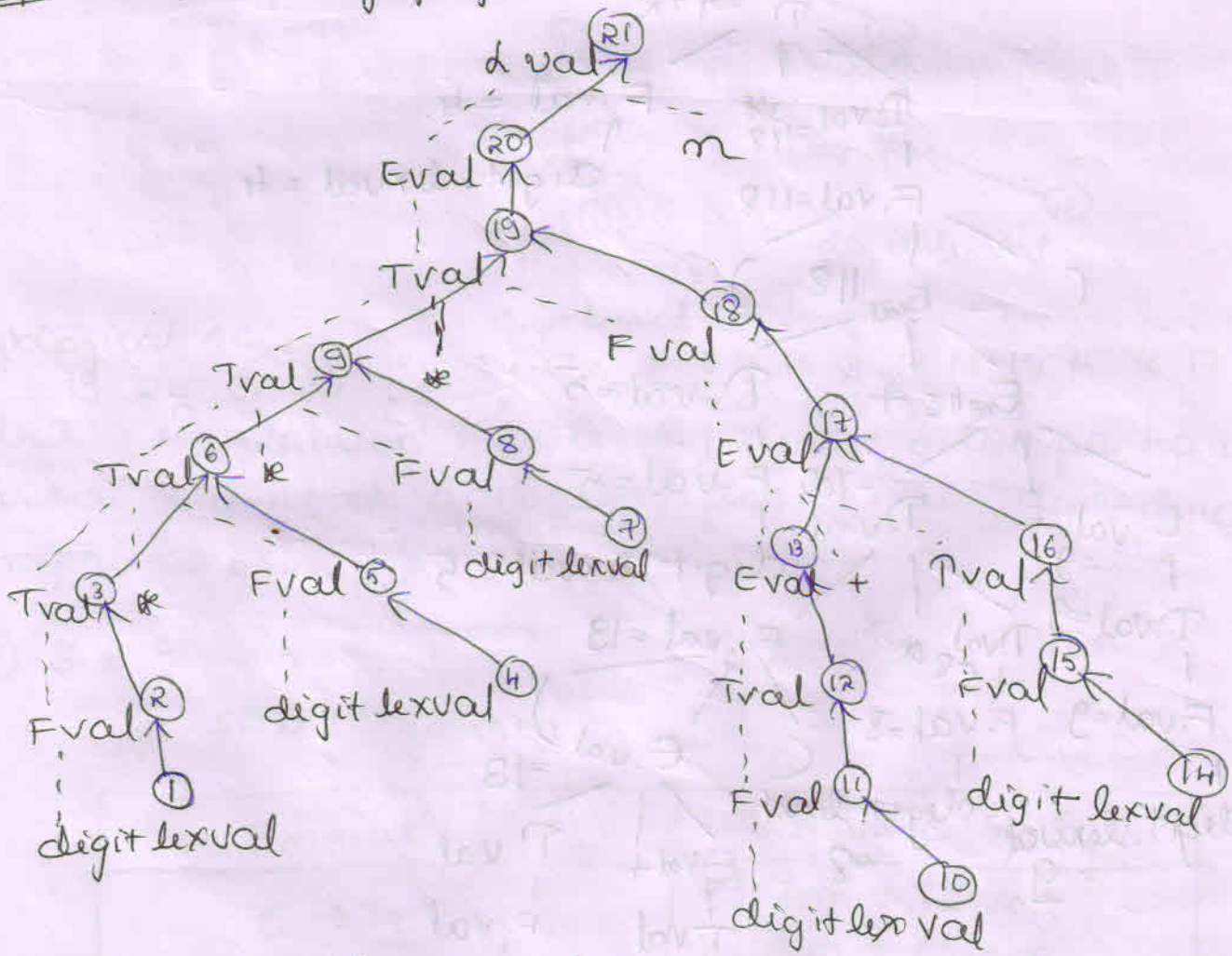
Sol:

Production	Semantic rules
$L \to En$	L.val = E.val
$E \to E + T$	E.val = E.val + T.val
$E \to T$	E.val = T.val
$T \to T * F$	T.val = T.val * F.val
$T \to F$	T.val = F.val
$F \to (E)$	F.val = E.val
$F \to digit$	F.val = digit.lexval

Q: Annotated parse tree



Step 3 : Dependency graph



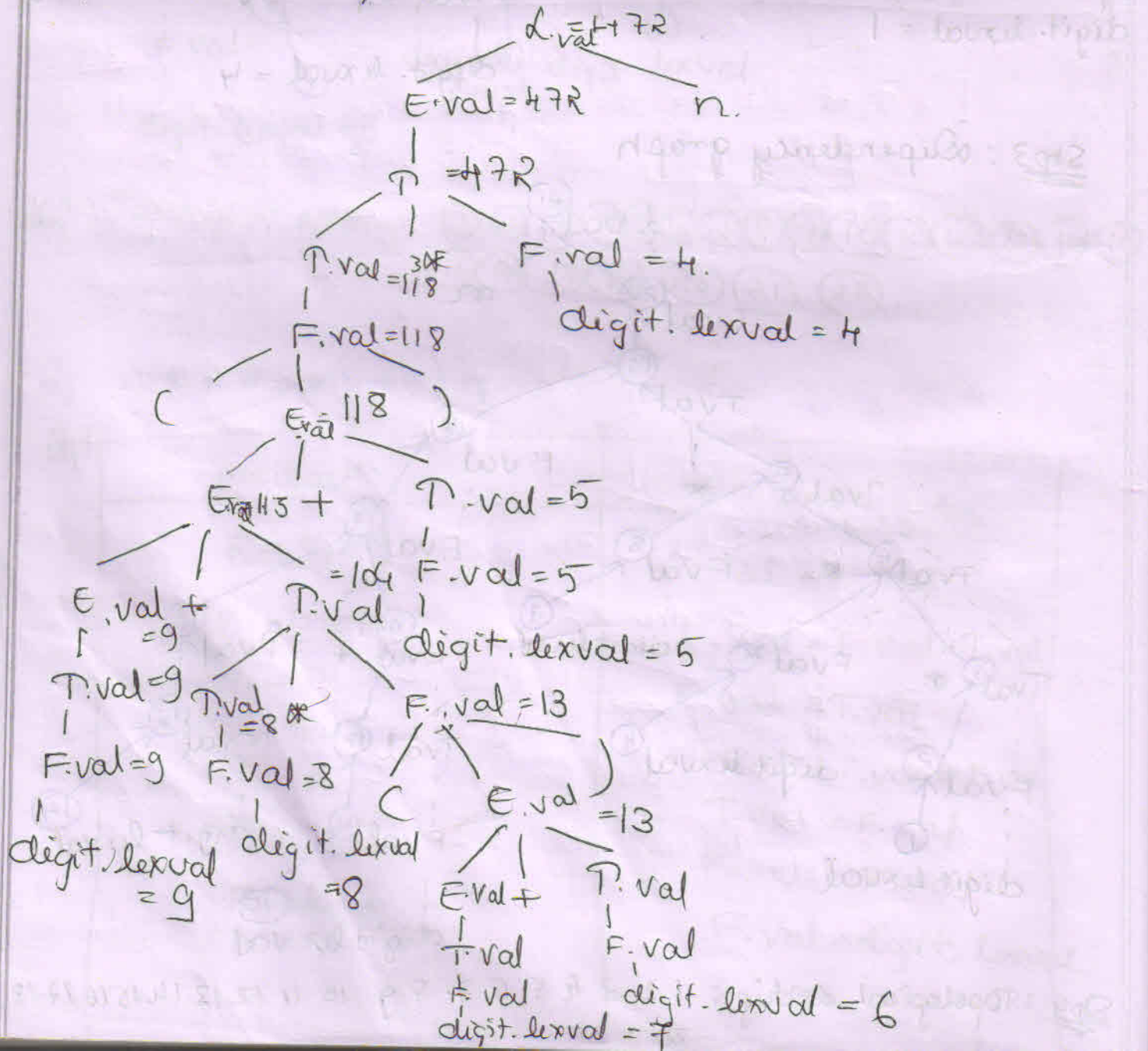
Step 4 : Topological sorting : 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21

c) (9 + 8 * (7+6) +5) * 4n

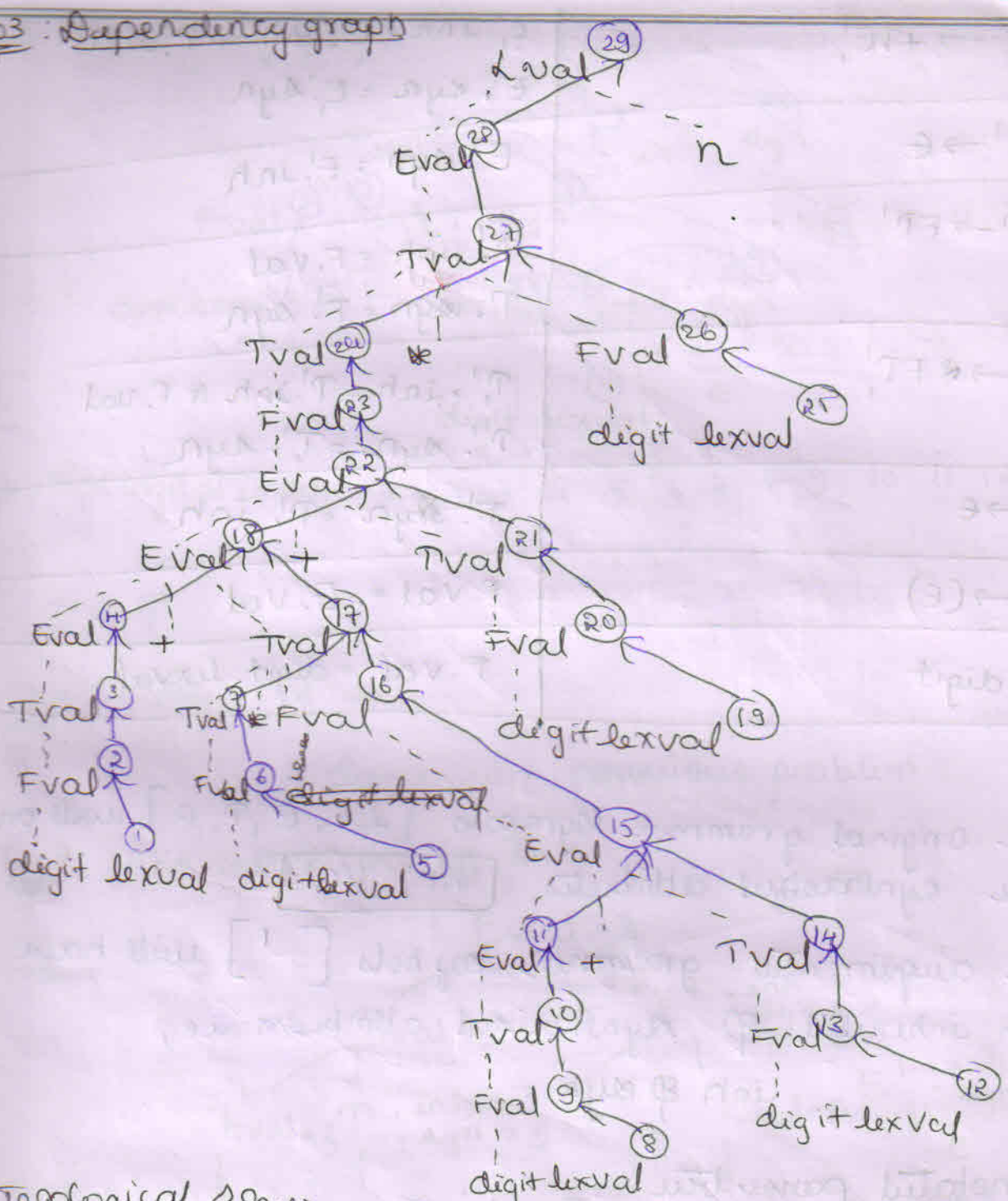
Step1: SDD defination

Production	Semantic rules
$L \to En$	L.val = E.val
$E \to E+T$	E.val = E.val + T.val
$E \to T$	E.val = T.val
$T \to T*F$	T.val = T.val * F.val
$T \to F$	T.val = F.val
$F \to (E)$	F.val = E.val
$F \to digit$	F.val = digit.lexval

Step2: Annotated parse tree



Step3 : Dependency graph



Topological sorting : 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29

I Write L-attributed SDD (or) write a SDD for top down parser & construct annotated parse tree, dependency graph for given i/p.

(1) 3 * 5

Step1 : SDD Defination

Production	Semantic Rules
$E \rightarrow TE'$	$E'.inh = T.val$ $E.val = E'.syn$

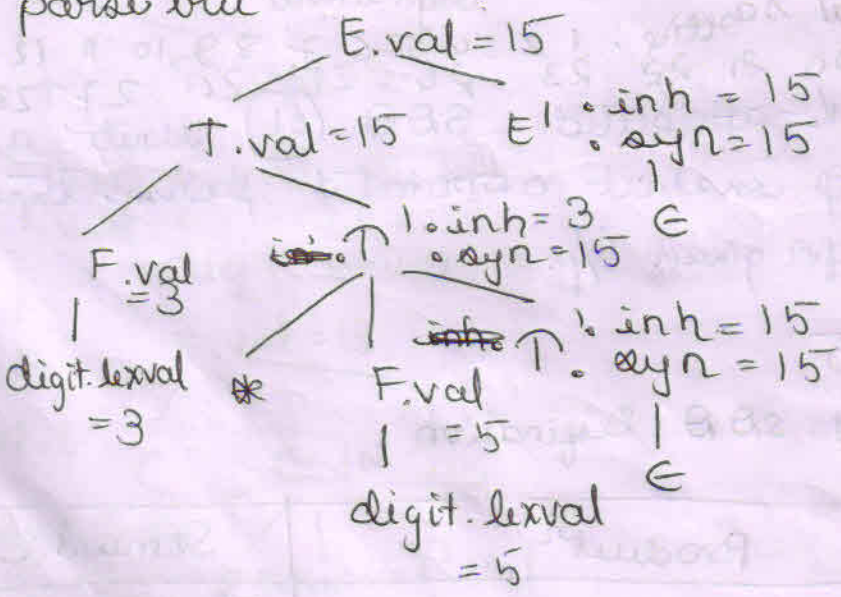
$E' \rightarrow +TE'_1$	$E'_1.inh = E'.inh + T.inh$ $E'.syn = E'_1.syn$
$E' \rightarrow \epsilon$	$E'.syn = E'.inh$
$T \rightarrow FT'$	$T'.inh = F.val$ $T.val = T'.syn$
$T' \rightarrow *FT'_1$	$T'_1.inh = T'.inh * F.val$ $T'.syn = T'_1.syn$
$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow digit$	$F.val = digit.lexval$

NOTE

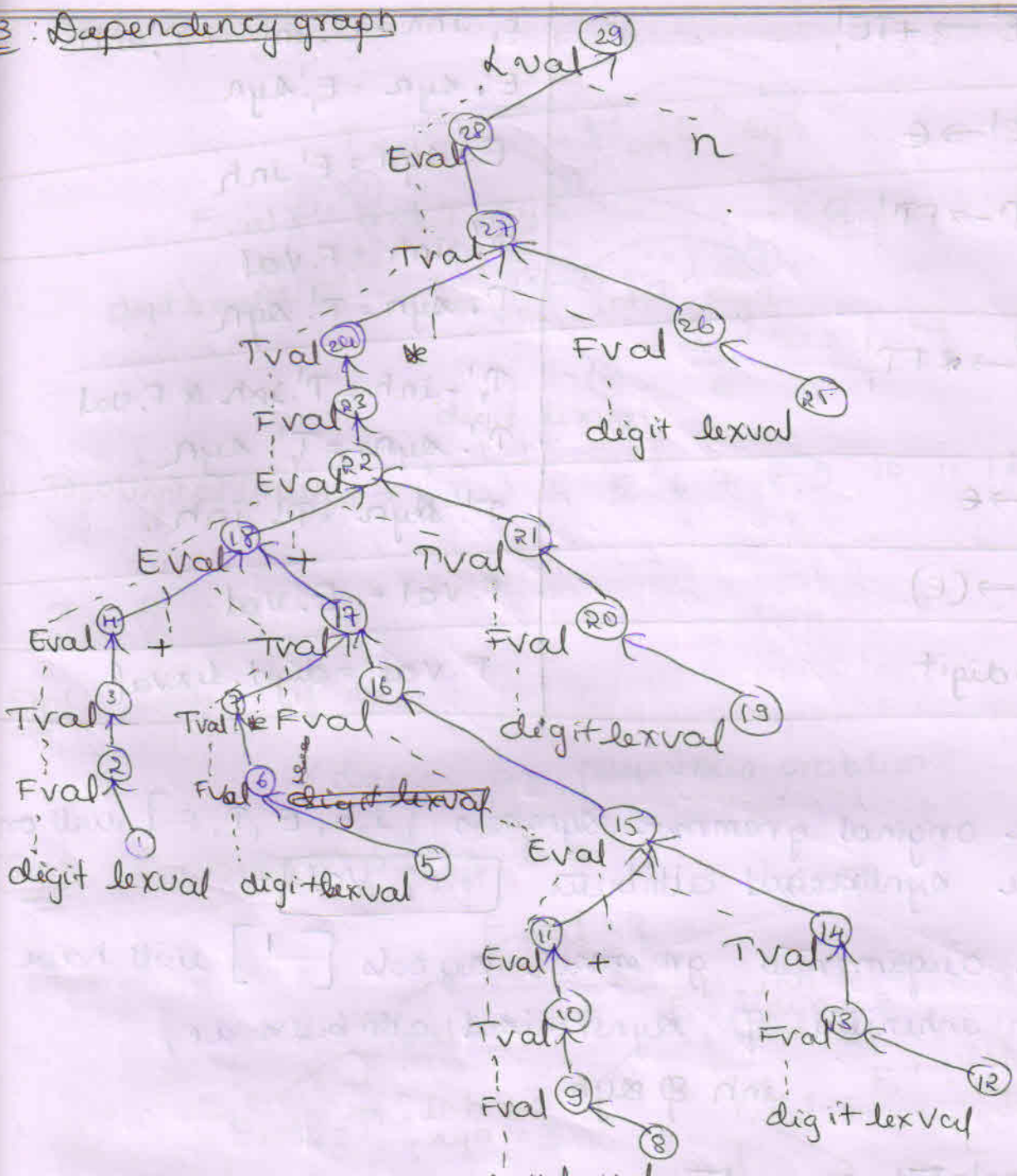
① The original grammar symbols [i.e., E, T, F] will only have synthesized attributes [i.e., val]

② The augmented grammar symbols ['] will have both inherited & synthesized attributes i.e., inh & syn

Step2: Annotated parse tree



03 : Dependency graph



Topological sorting : 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29

Write L-attributed SDD (or) write a SDD for top down parser & construct annotated parse tree, dependency graph for given i/p.

(1) 3 * 5

Sol : SDD Defination

Production	Semantic Rules.
$E \rightarrow TE'$	$E'.inh = T.val$ $E.syn = E'.syn$

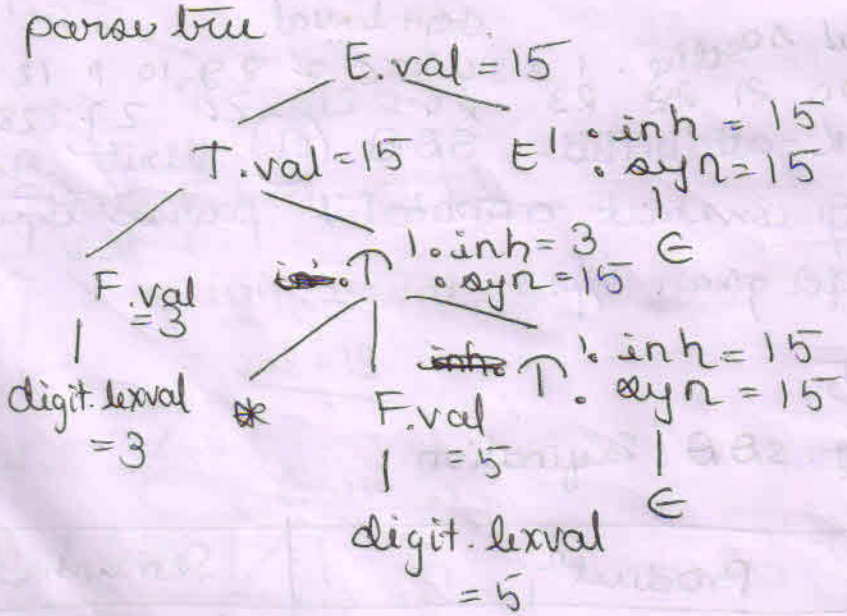
$E' \rightarrow +TE'_1$	$E'_1.inh = E'.inh + T.inh$ $E'.syn = E'_1.syn$
$E' \rightarrow \epsilon$	$E'.syn = E'.inh$
$T \rightarrow FT'$	$T'.inh = F.val$ $T.syn = T'.syn$
$T' \rightarrow *FT'_1$	$T'_1.inh = T'.inh * F.val$ $T'.syn = T'_1.syn$
$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow digit$	$F.val = digit.lexval$

NOTE

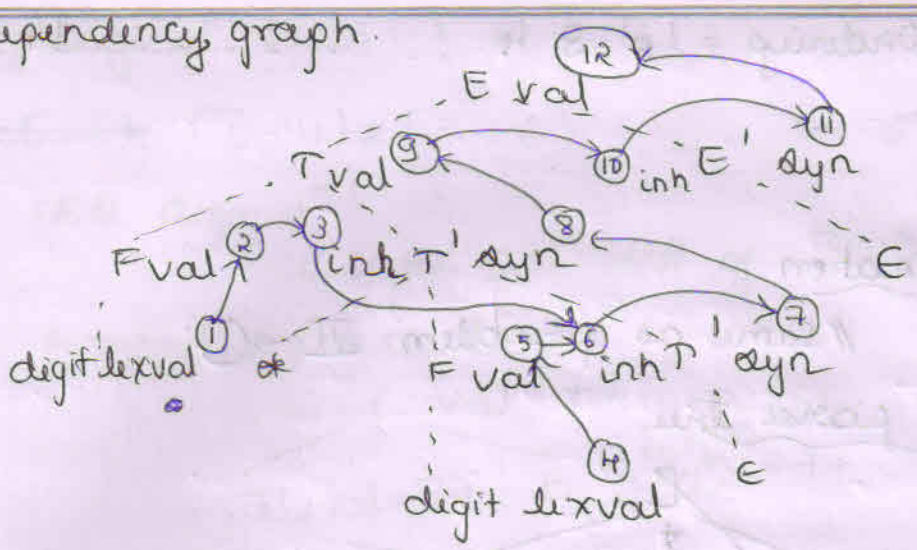
① The original grammer symbols [i.e., E, T, F] will only have synthesized attributes [i.e., val]

② The augumented grammer symbols ['] will have both inherited & synthesized attributes i.e., inh & syn

Step2: Annotated parse tree



Step3: Dependency graph.



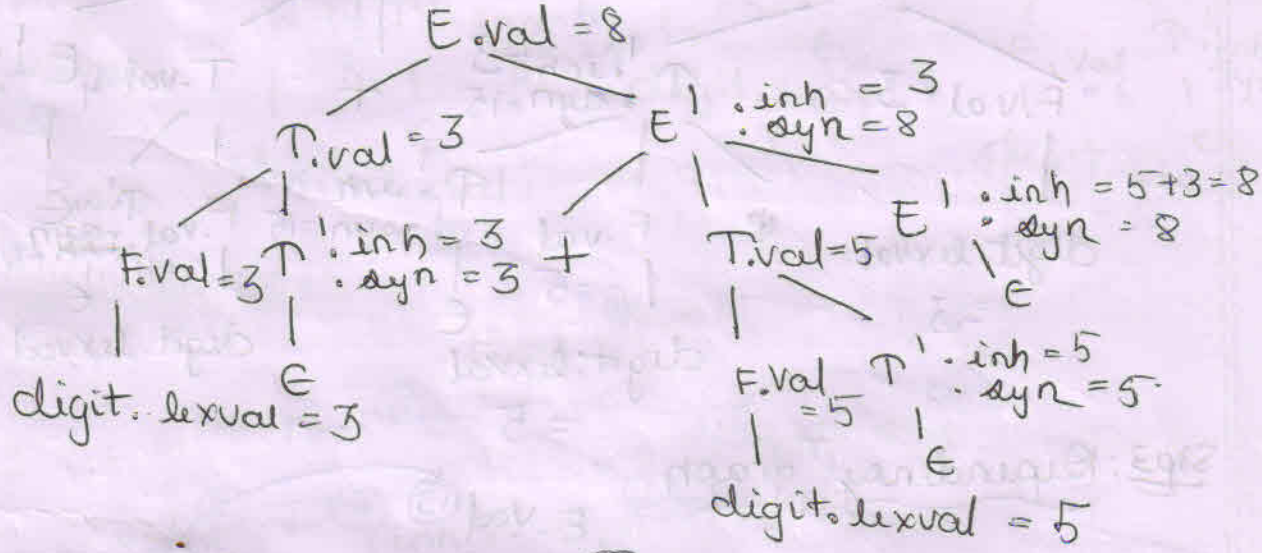
Step4: Topological Order → 1 2 3 4 5 6 7 8 9 10 11 12.

(2) 3 + 5

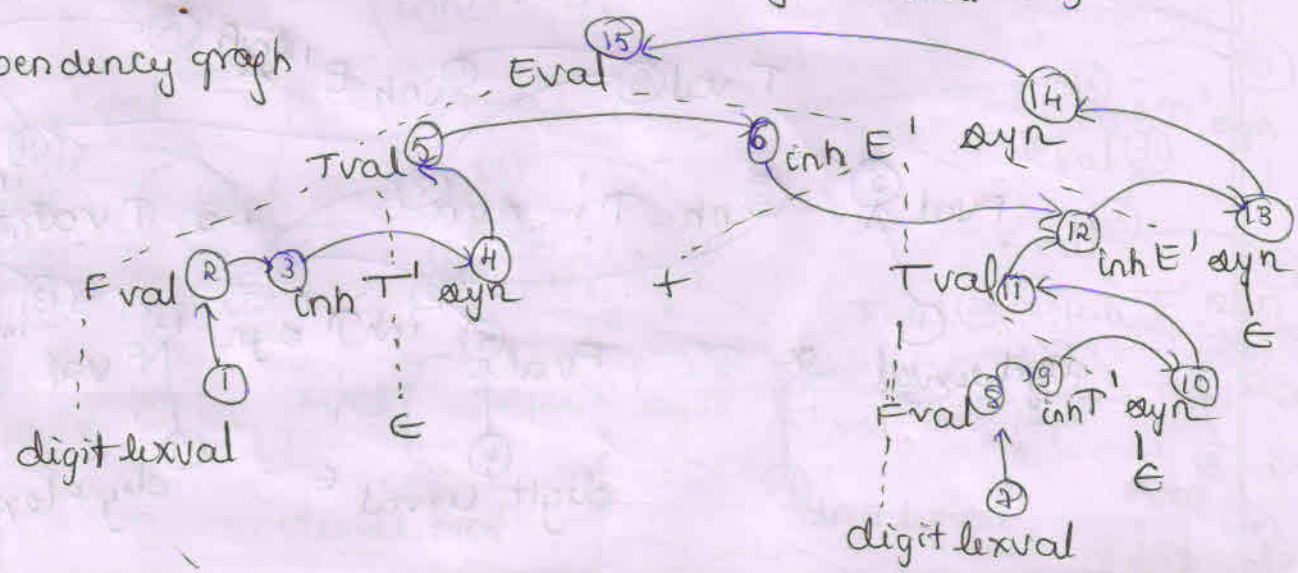
Step1: SDD defination

// Same as previous problem.

Step2: Annotated parse tree



Step3: Dependency graph



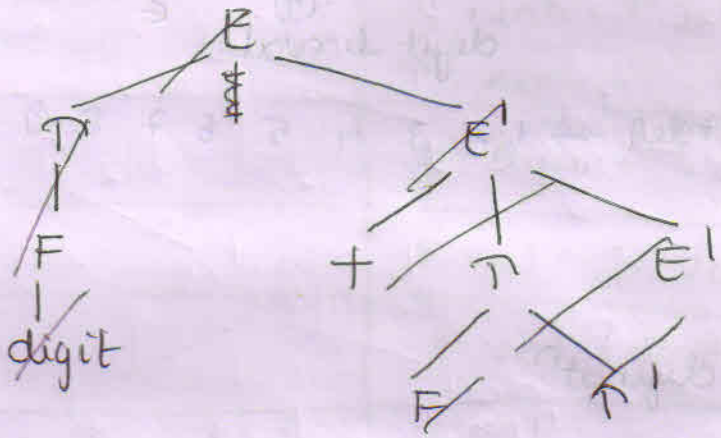
Step4: Topological Ordering = 1 2 3 4 .. .. 14 15

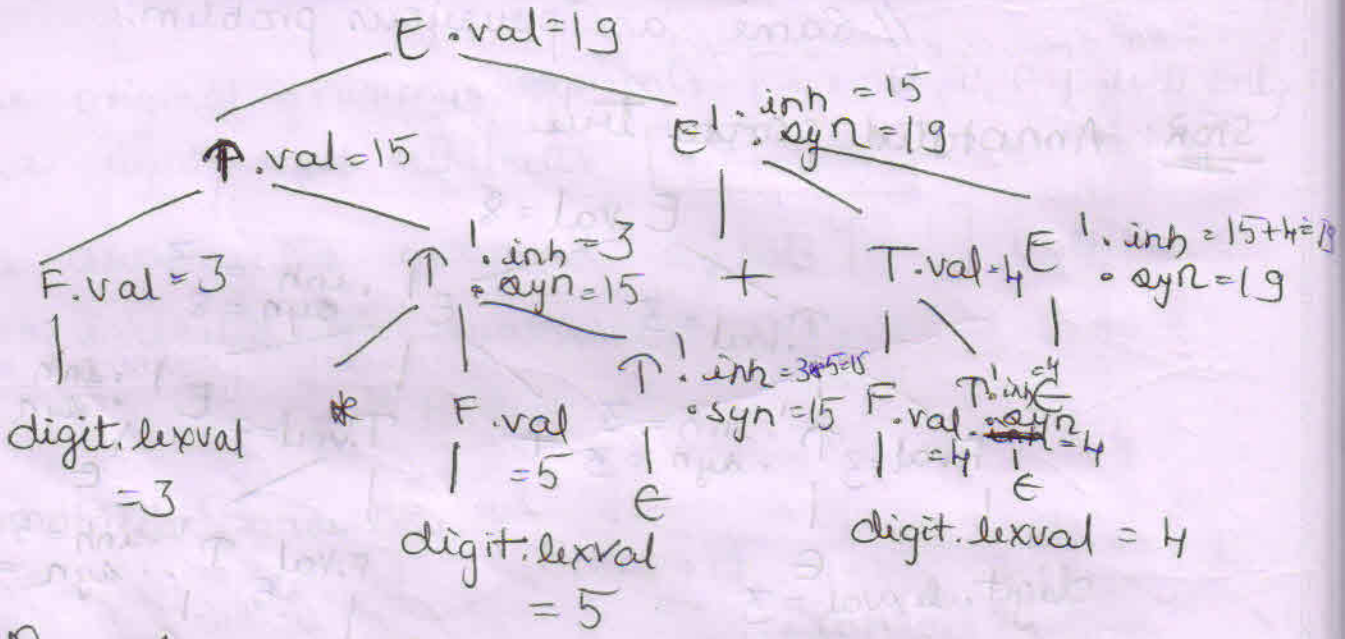
(3) 3 * 5 + 4

Step1: SDD definition

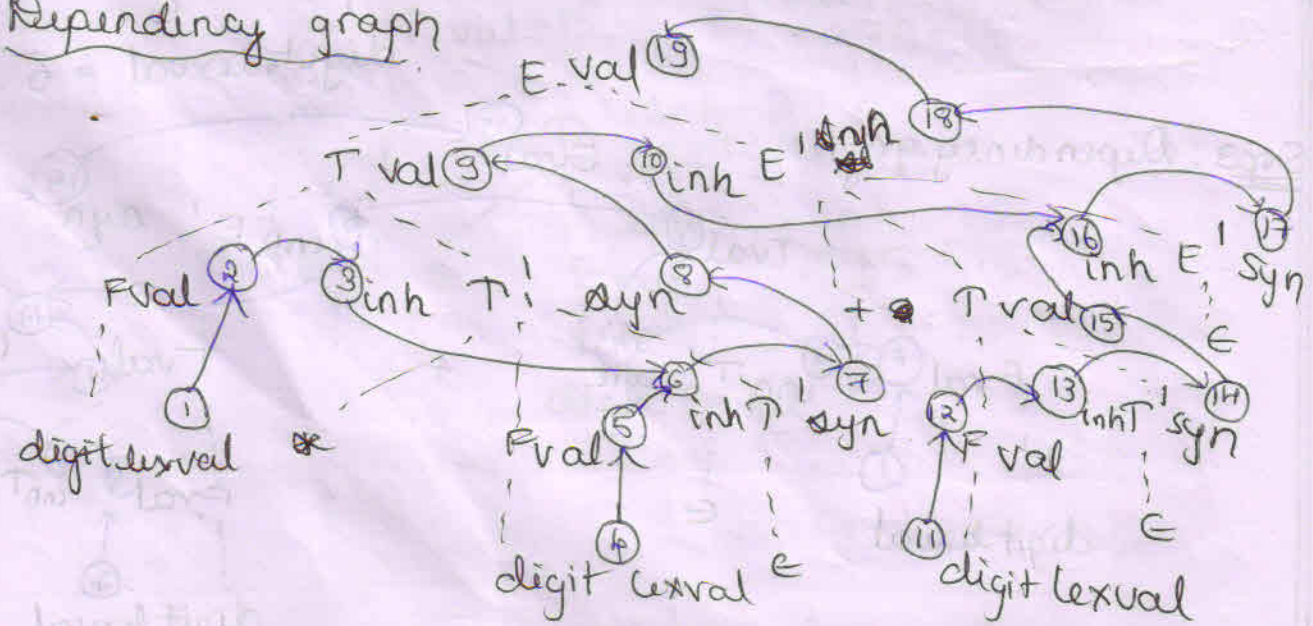
// Same as problem II → (1)

Step2: Annotated parse tree





Step3: Dependency graph

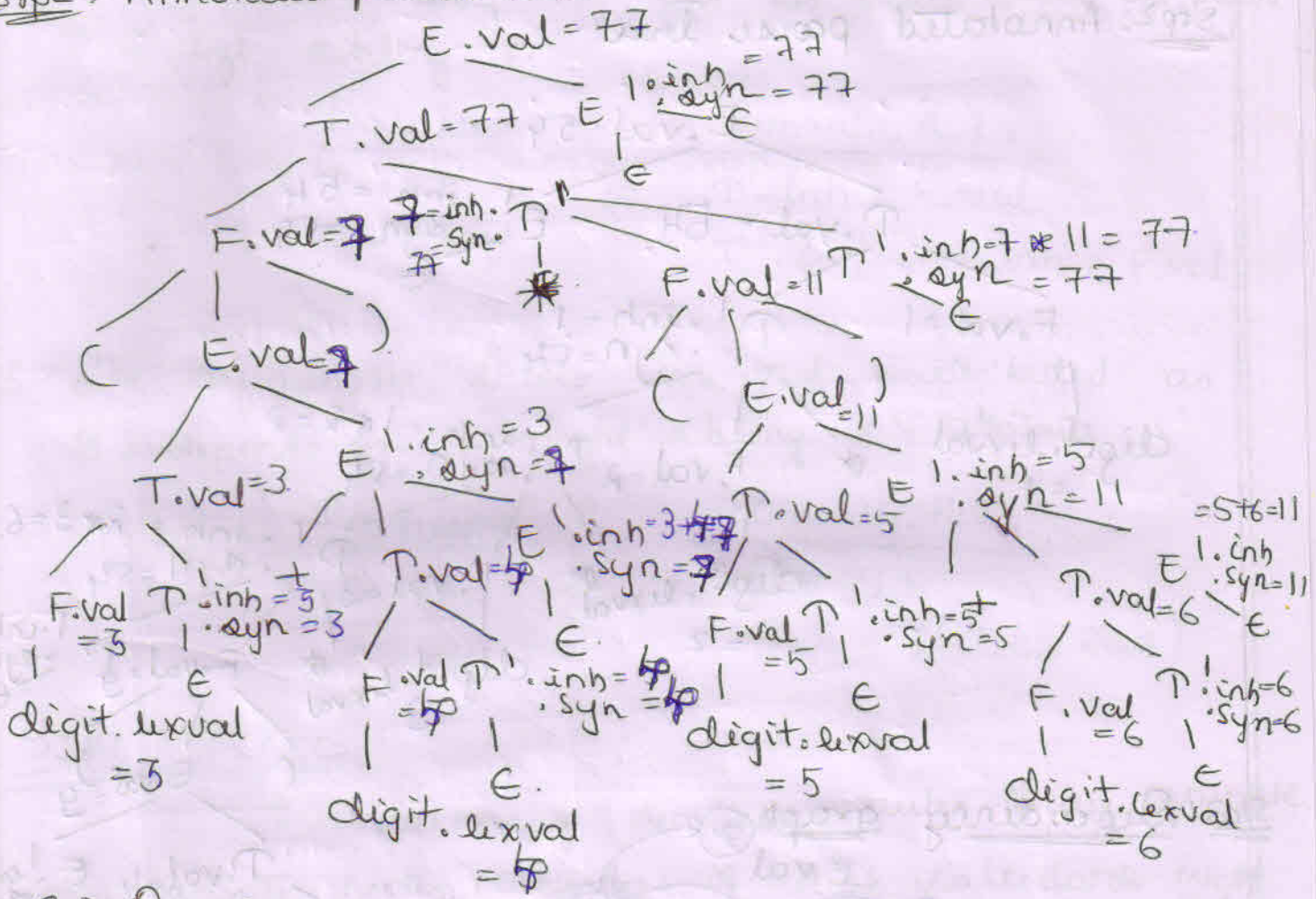


Topological Order: 1 2 3 4 19

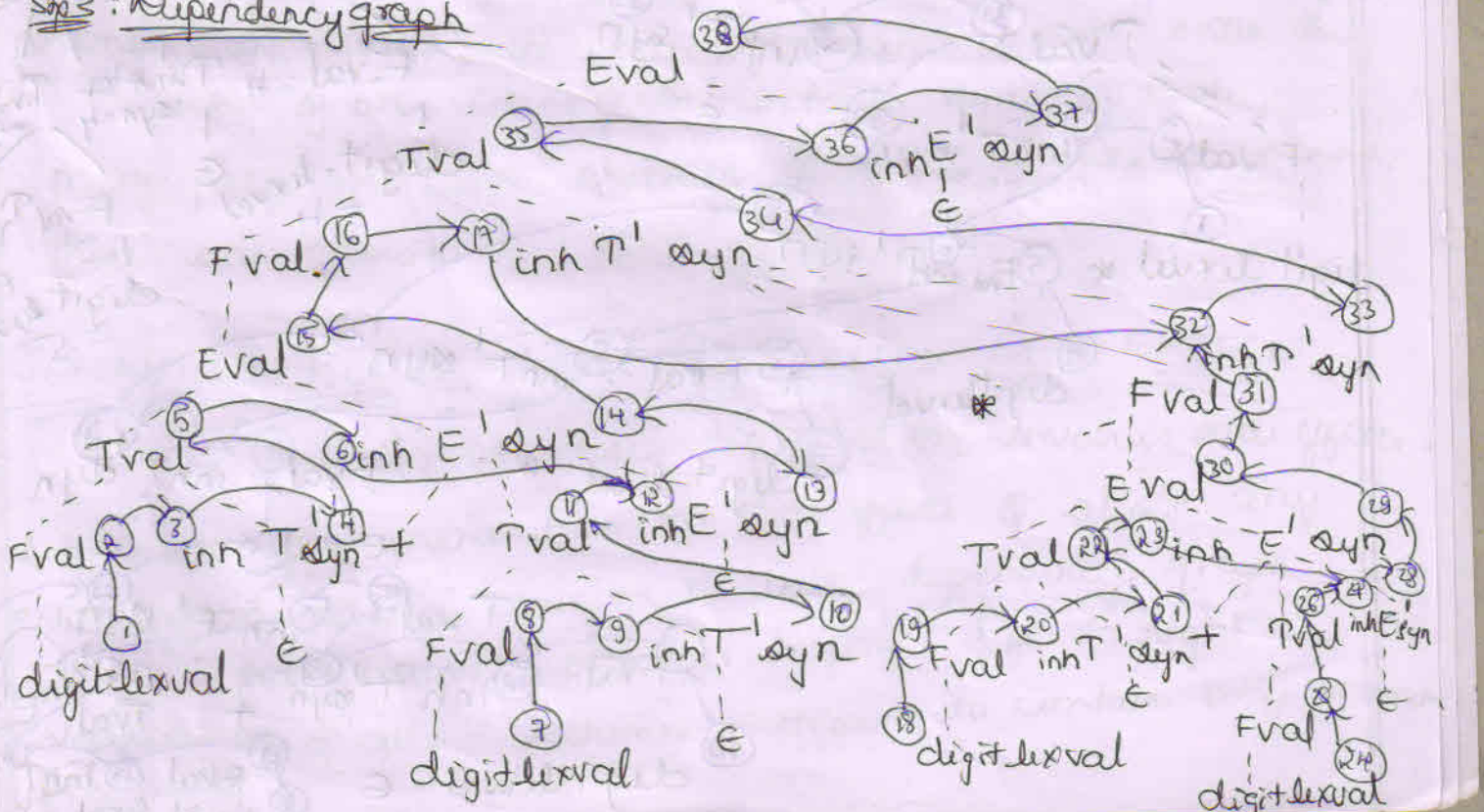
~~3 * 5 + 4~~ (3+4)*(5+6)

Stp1: SDD definat^n // same as SDD of II → (1)

Stp2: Annotated parse tree



Stp3: Dependency graph



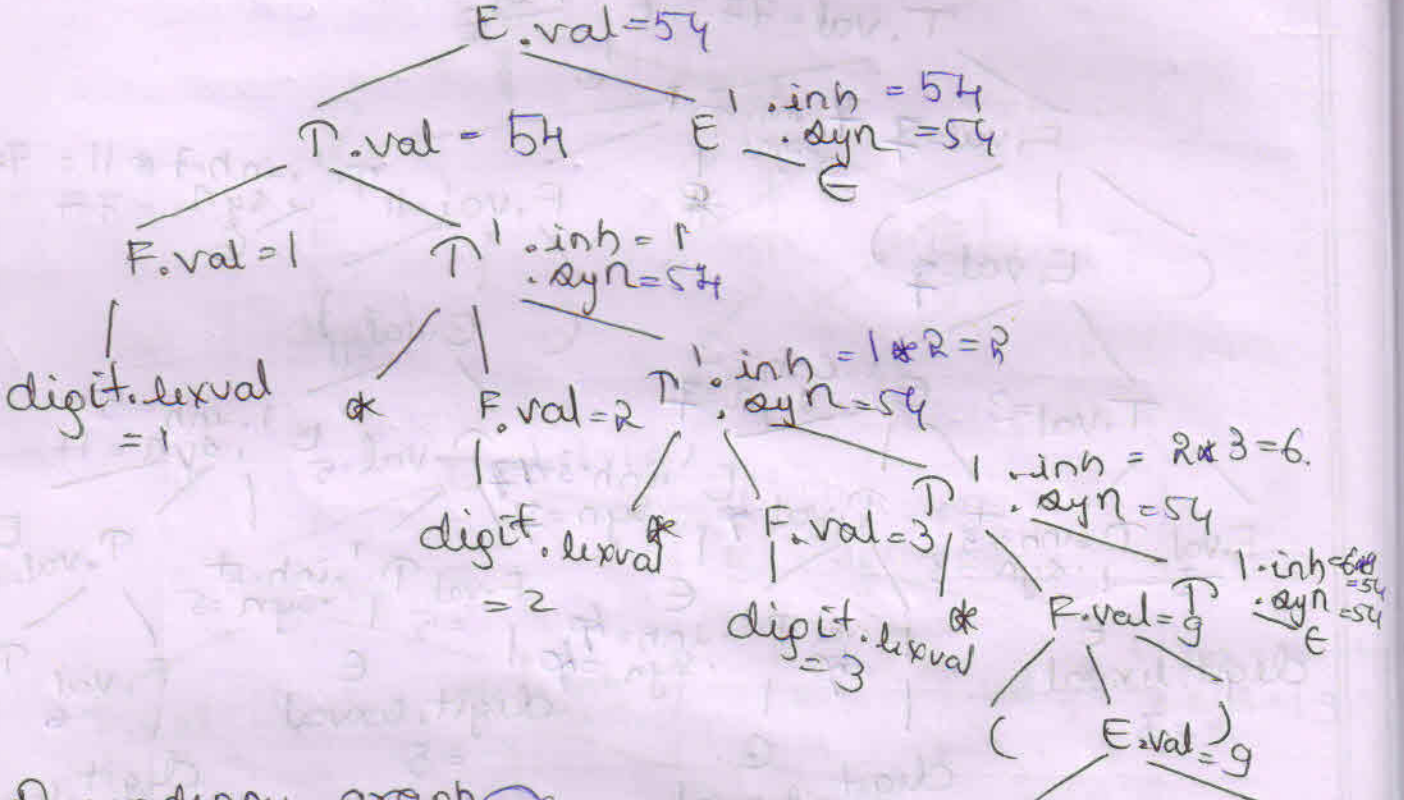
Topological Ordering ① ② ③ ④ ---------- (37) (38)

(5) 1*2*3*(4+5)

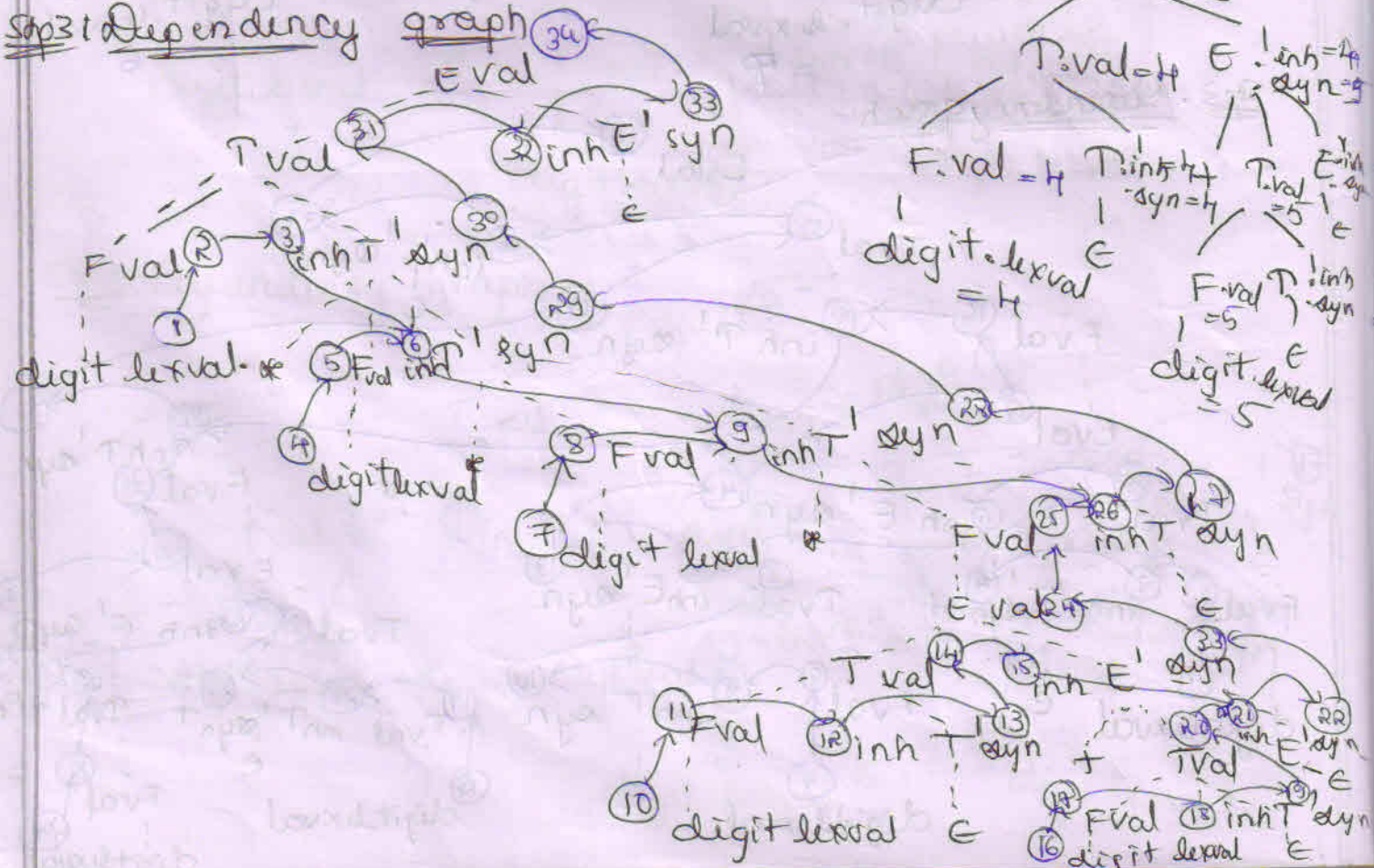
Step1: SDD depinat^n

// same as II → ①

Step2: Annotated parse tree



Step3: Dependency graph



Eg1: The following dynation is L attributed. Here the inherited attribute of T' gets its value from its left sibling F. Similarly T_1' gets its value from its parent T' & left sibling F

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

Eg2: The dynations below are not L-attributed as B.i depends on its right sibling C's attribute.

Production	Semantic Rules
$A \rightarrow BC$	$A.s = B.b$
	$B.i = f(C.c, A.s)$

SIDE EFFECTS

Evaluation of semantic rules may generate intermediate codes, Eg: A desk calculator might print a result, a code generator might enter the type of an identifier into a symbol table, may perform type checking & may issue error msgs. These are known as side effects.

SEMANTIC RULES WITH CONTROLLED SIDE EFFECTS

In practise translation involves side effects. Attribute grammers has no side effects & allow any evaluation order consistent with dependency graph whereas translation schemes impose left to right evaluation & allow scheme actions to contain any program fragment.

Ways to Control Side Effects

1. permit incidental side effects that do not constrain attribute evaluation.
   In other words, permit side effects when attr evaluation is based on any topological sort of the dependency graph produces a correct translation.
2. Impose constraints on allowable evaluation orders so that the same translation is produced for any allowable order.

Write an SDD for simple type declaration

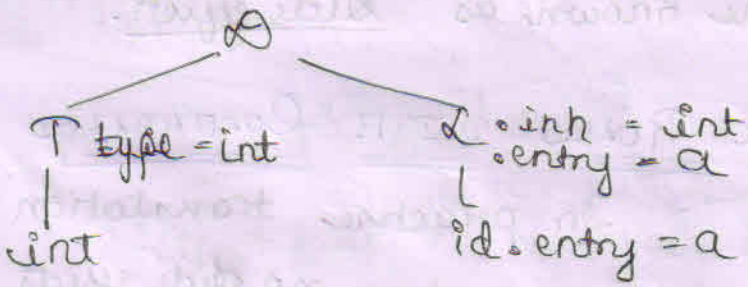
i/p: int a

a)

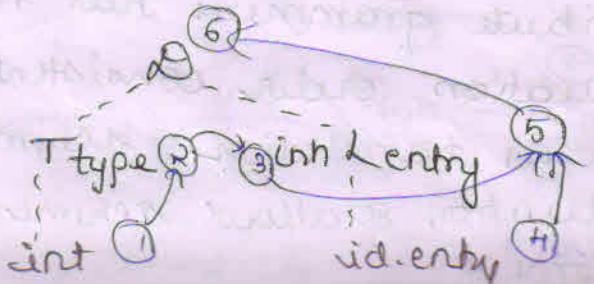
Step1:

Production	Semantic rules
$D \to T\ L$	$L.inh = T.type$
$T \to int$	$T.type = int$
$T \to float$	$T.type = float$
$L \to L_1, id$	$L_1.inh = L.inh$ addtype(id.entry, L.inh)
$L \to id$	addtype(id.entry, L.inh)

Step2: Annotated parse tree



Step3: Dependency graph



Explanation:

Non terminal D represents a declaration, which from production 1, consists of a type T followed by a list L of identifiers. T has one attribute: T.type, which is the type in the declaration D. Nonterminal L has one attribute, which call inh to emphasize that it is an inherited attribute. The purpose of L.inh is to pass the declared type down the list of identifiers, so that it can be the appropriate symbol table entries. Production (2) & (3) each evaluate the synthesized attribute T.type giving it the appropriate value, integer or float. This type is passed to the attribute L.inh in the rule of production 1. Production 4 passes L.inh down the parse tree i.e., the value of L1.inh is computed at a parse tree node by copying the value of L1.inh from the parent of that node, the parent corresponds to the head of production. Production (4) & (5) also have a rule in which a funct^n addtype is called with 2 arguments:

1) id.entry a lexical value that points to a symbol table object.

2) L.inh, the type being assigned to every identifier on the list.

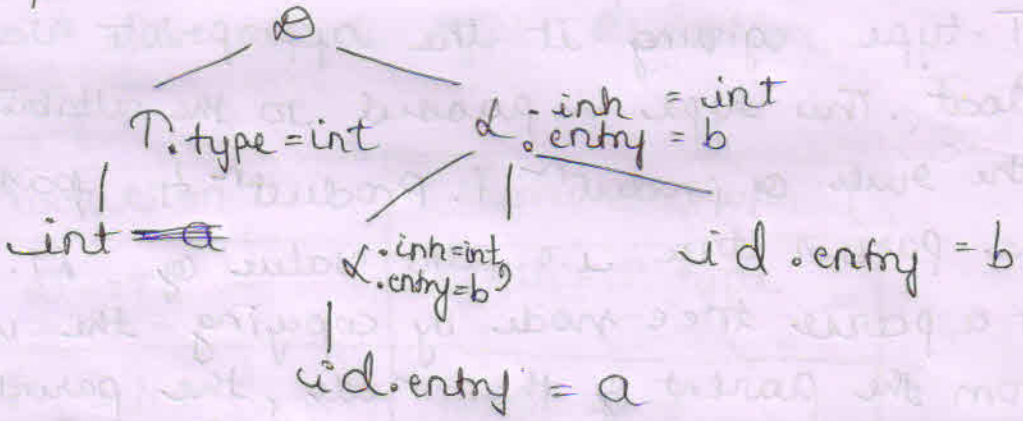
The function addType properly installs the type L.inh as the type of the represented identifier. Note that the side effect, adding the type info to the table, doesnot effect the evaluation order.

(b) int a, b

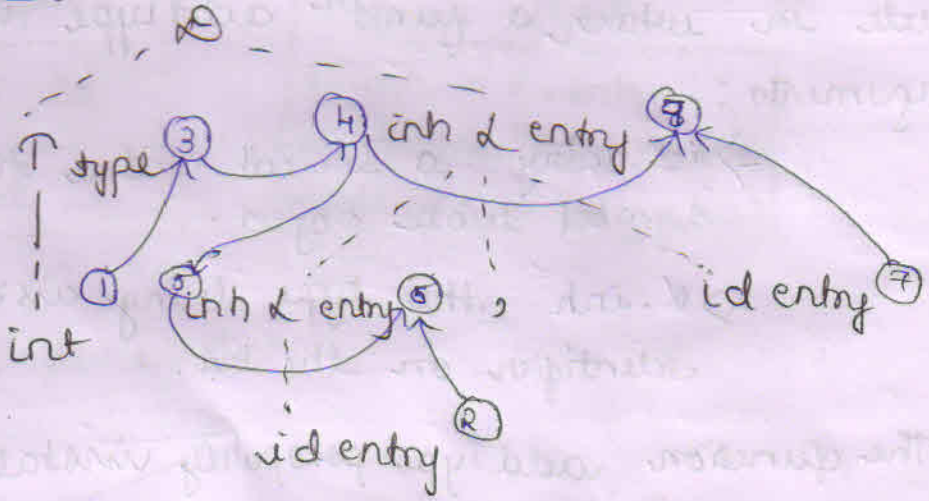
Step1

Production	Semantic rules
$D \rightarrow T\ L$	$L.inh = T.type$
$T \rightarrow int$	$T.type = int$
$T \rightarrow float$	$T.type = float$
$L \rightarrow L, id$	$addtype(id.entry, L_1.inh)$ $L_1.inh = L.inh$
$L \rightarrow id$	$addtype(id.entry, L.inh)$

Step2: Annotated parse tree



Step3: Dependency graph



Step4: Topological order (1) (2) (3) (4) (5) (6) (7) (8) (9)

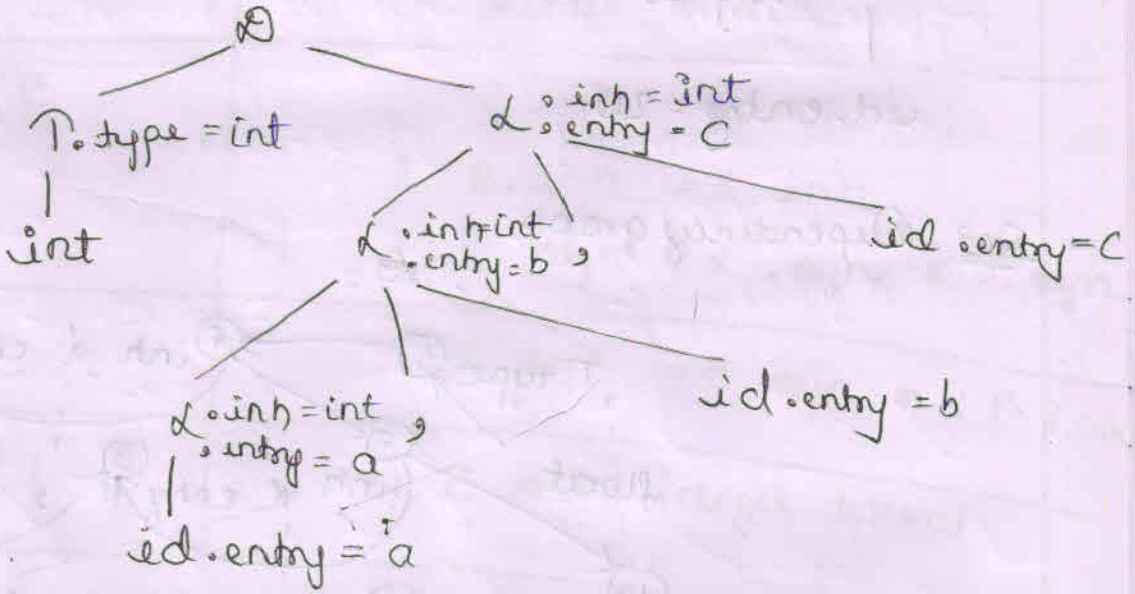
UNIT-5 [continued ....]

(c) float a, b, c. or int a, b, c <Exercise 5.2.2>

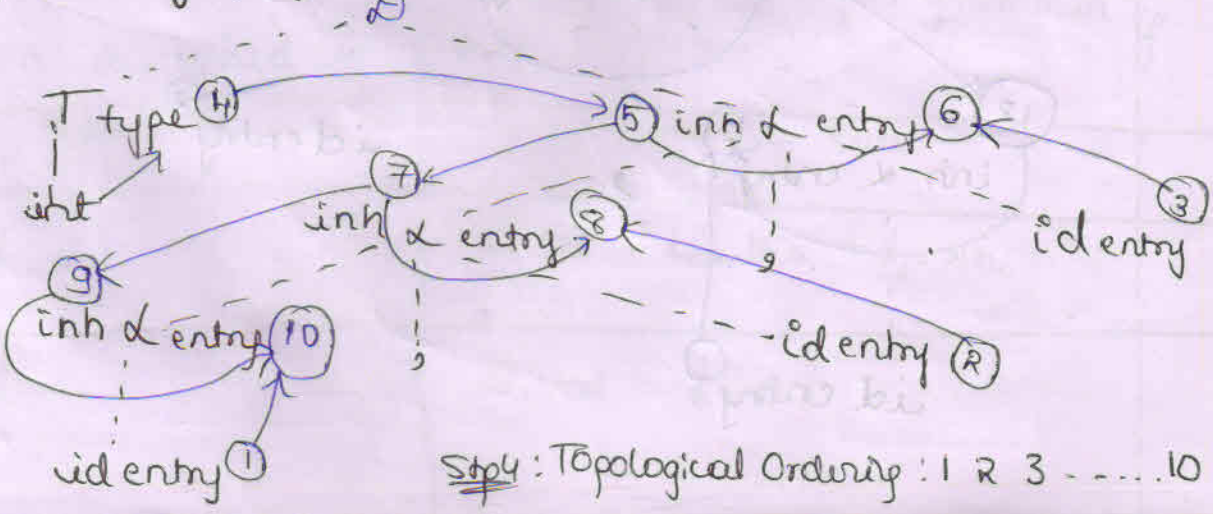
Step1 : SDD defination

Production	SDD
$D \to T\ L$	$L.inh = T.type$
$T \to int$	$T.type = int$
$T \to float$	$T.type = float$
$L \to L_1, id$	$L_1.inh = L.inh$ addtype (id.entry, L.inh)
$L \to id$	addtype (id.entry, L.inh)

Step2 : Annotated parse tree



Step3 : Dependency graph



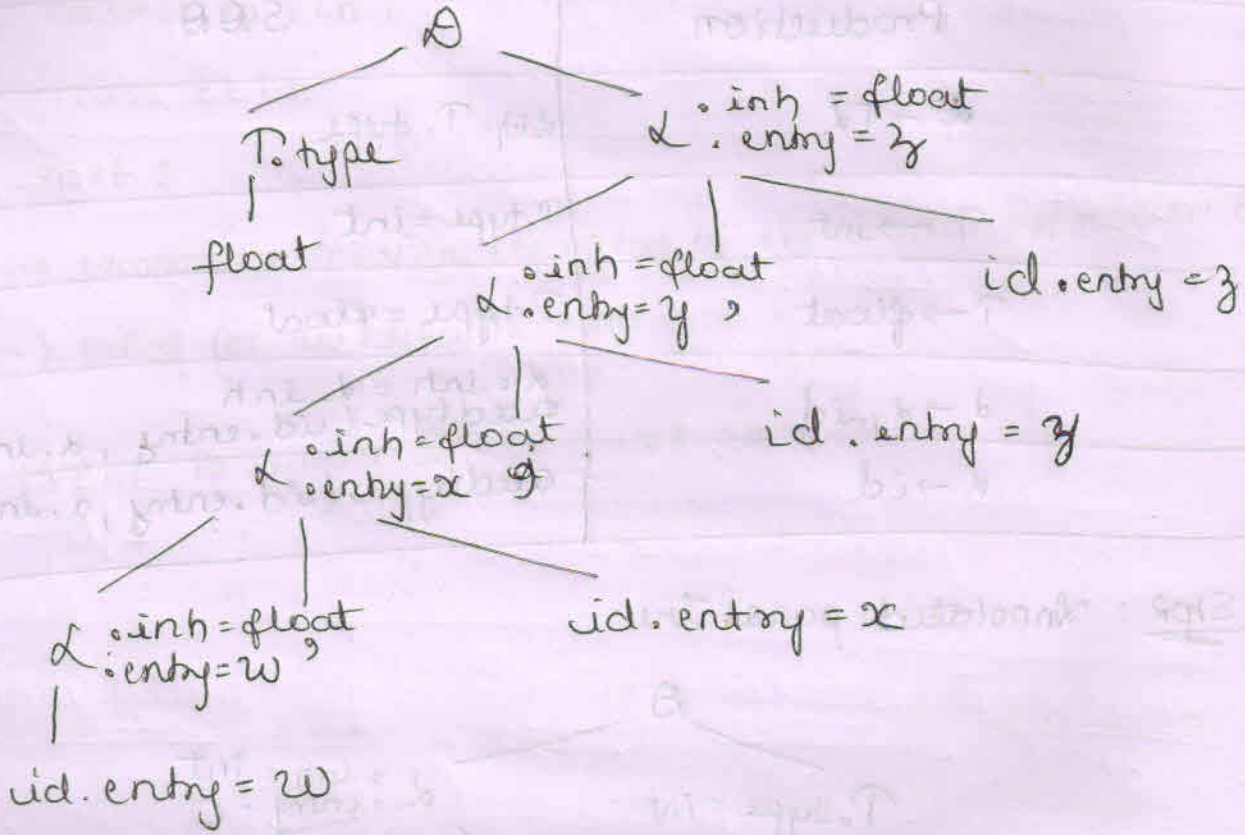
Step4 : Topological Ordering : 1 2 3 10

(d) float w, x, y, z

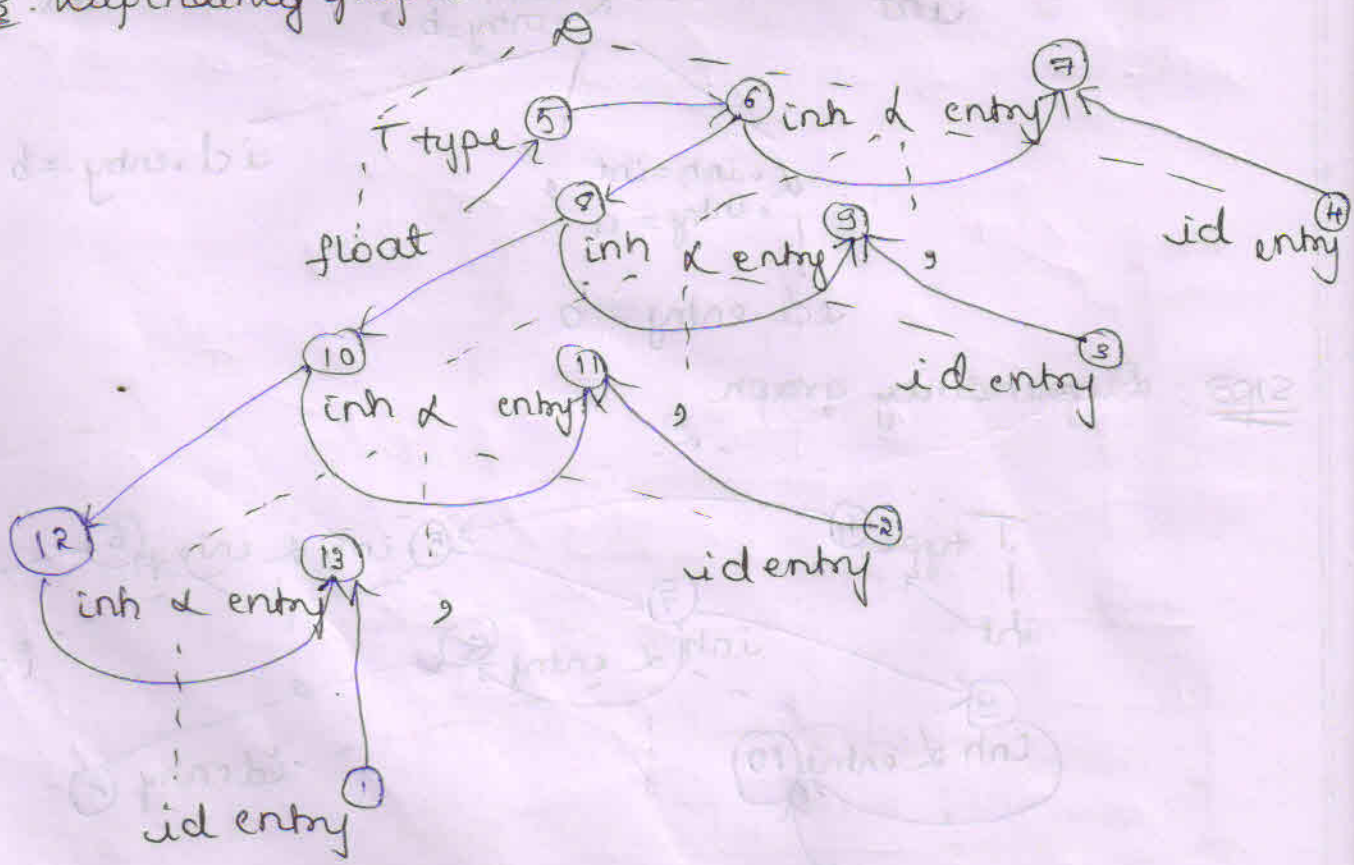
Step1 SDD definat^n

// Same as previous problem.

Step2 : Annotated parse tree



Step3 : Dependency graph



Exercise 5.2.4 This grammar generates binary no with a decimal point

$S \to L . L \mid L$

$L \to L B \mid B$

$B \to 0 \mid 1$

Design an L attributed SDD to compute S.val, the decimal no value of i/p string. For eg, translation of string 101.101 should be the decimal no 5.625.

Soln

Production	SDD
1) $S \to L . L_1$	1. L.inh = 0 2. L_1.inh = -1 3. S.val = L.syn + L_1.syn
2) $S \to L$	1. L.inh = 0 2. S.val = L.syn
3) $L \to L_1 B$	1. L_1.inh = L.inh + 1 2. B.inh = L.inh 3. L.syn = L_1.syn * B.syn
4) $L \to B$	1. L.syn = B.syn * 2 ^ L.inh
5) $B \to 0 \mid 1$	1. B.syn = digit.lexval

Exercise 5.2.5 Design an S-attributed SDD for grammar of translation described in 5.2.4

Production	Semantic Rule.
1) $S \to L . L_1$	S.val = L.lhs + L_1.rhs
2) $S \to L$	S.val = L.lhs

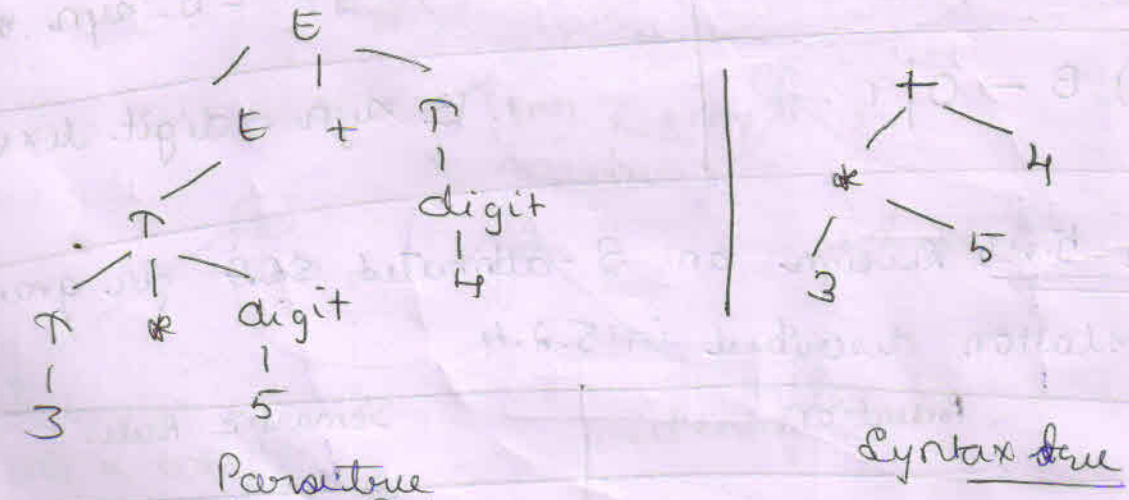
3) $\alpha \rightarrow \alpha_1 B$	1) $\alpha.lhs = \alpha_1.lhs + (2 \uparrow \alpha.lhs\_exponent * B.val)$ 2) $\alpha.rhs = \alpha_1.rhs + (2 \uparrow \alpha.rhs\_exponent * B.val)$ 3) $\alpha.lhs\_exponent = \alpha_1.lhs\_exponent + 1$ 4) $\alpha.rhs\_exponent = \alpha_1.rhs\_exponent + 1$
4) $\alpha \rightarrow B$	1) $\alpha.lhs = 2 \uparrow \alpha.lhs\_exponent * B.val$ 2) $\alpha.rhs = 2 \uparrow \alpha.rhs\_exponent * B.val$ 3) $\alpha.lhs\_exponent = 0$ 4) $\alpha.rhs\_exponent = -1$
5) $B \rightarrow 0 \mid 1$	B.Val = digit.lexval

Application Of Syntax - Directed Translation

1. Construction Of Syntax Tree

SDD's are useful for construct^ of syntax tree.

A syntax tree is condensed form of parse tree



Parse tree

Syntax tree

* Syntax trees are useful for representing programming language constructs like expressions & statements.

* They help compiler design by decoupling parsing from translatⁿ.

* Each node of a syntax tree represent a construct; the children of the node represent the meaningful components of a construct

Eg: A syntax tree node representing an expression E1 + E2 has label + & 2 children representing the sub expression E1 & E2

* Each node is implemented by objects with suitable no of fields; each object will have an Opfield that is the label of node with additional fields as follows:

a) If the node is a leaf, an additional field holds the lexical value for the leaf. This is created by function Leaf (op, val).

b) If the node is an interior node, there are as many fields as the node has children in syntax tree. This is created by function Node(op, C1, C2, ... Ck)

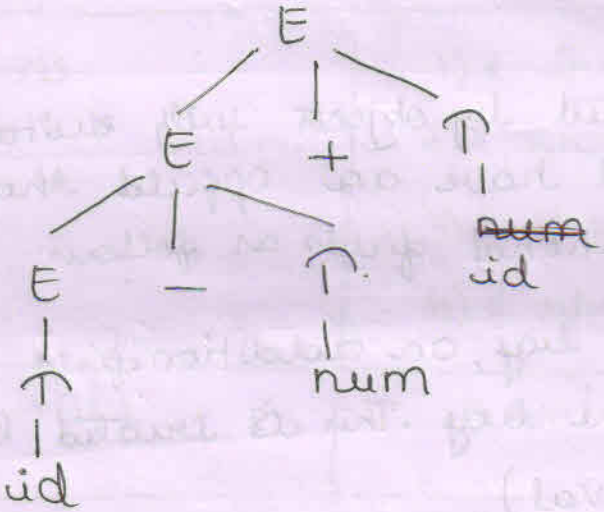
Example: The S-attributed definatⁿ in fig below constructs syntax trees for a simple expn grammar involving only binary operators + & -. As usual these operators are at the same precedence level & are jointly left associative. All nonterminals have one synthesized attr node, which represents a node of the syntax tree.

Q-H+C

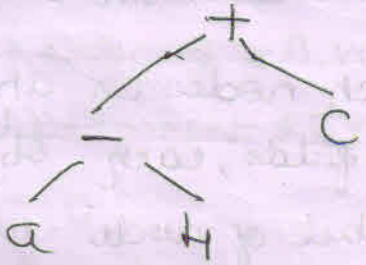
	PRODUCTION	SEMANTIC RULES
1.	$E \rightarrow E_1 + T$	E.node = new Node ('+', E_1.node, T.node)
2.	$E \rightarrow E_1 - T$	E.node = new Node ('-', E_1.node, T.node)
3.	$E \rightarrow T$	E.node = T.node

$T \rightarrow (E)$	T.node = E.node
$T \rightarrow id$	T.node = newleaf(id, id.entry)
$T \rightarrow num$	T.node = newleaf(num, num.val)

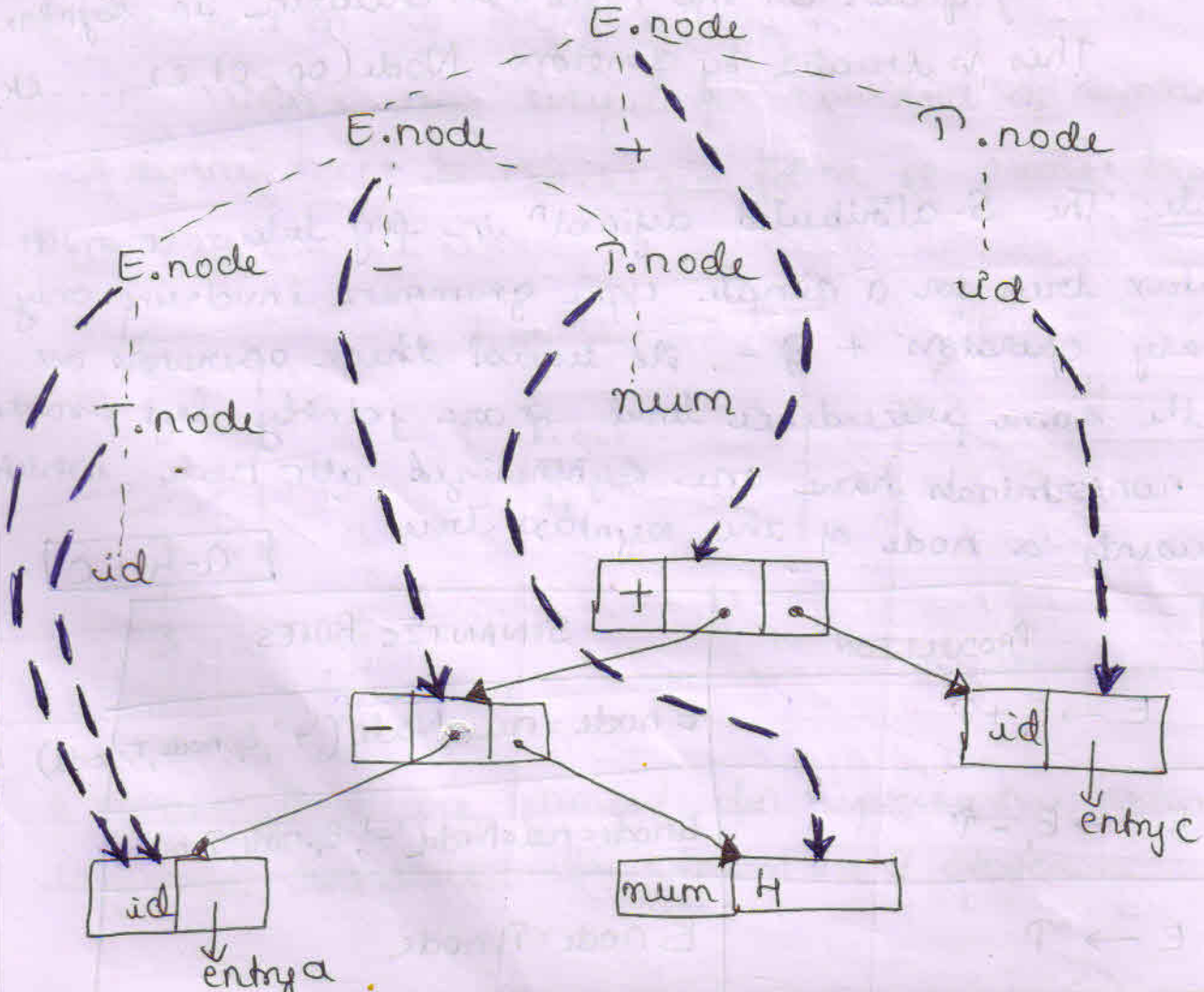
Step2 : Parse tree



Syntax tree



Step3 : Syntax tree for a − 4 + c using above SDD :



Steps in construction of the syntax tree for a-4+c

If the rules are evaluated during a post order traversal of the parse tree, or with reduction during a bottom up parse, then the sequence of steps shown below ends with p_5 pointing to the root of the constructed syntax tree.

1) P_1 = new leaf (id, entry-a)
2) P_2 = new leaf (num, 4)
3) P_3 = newNode ('-', P_1, P_2)
4) P_4 = new leaf (id, entry-c)
5) P_5 = new Node ('+', P_3, P_4)

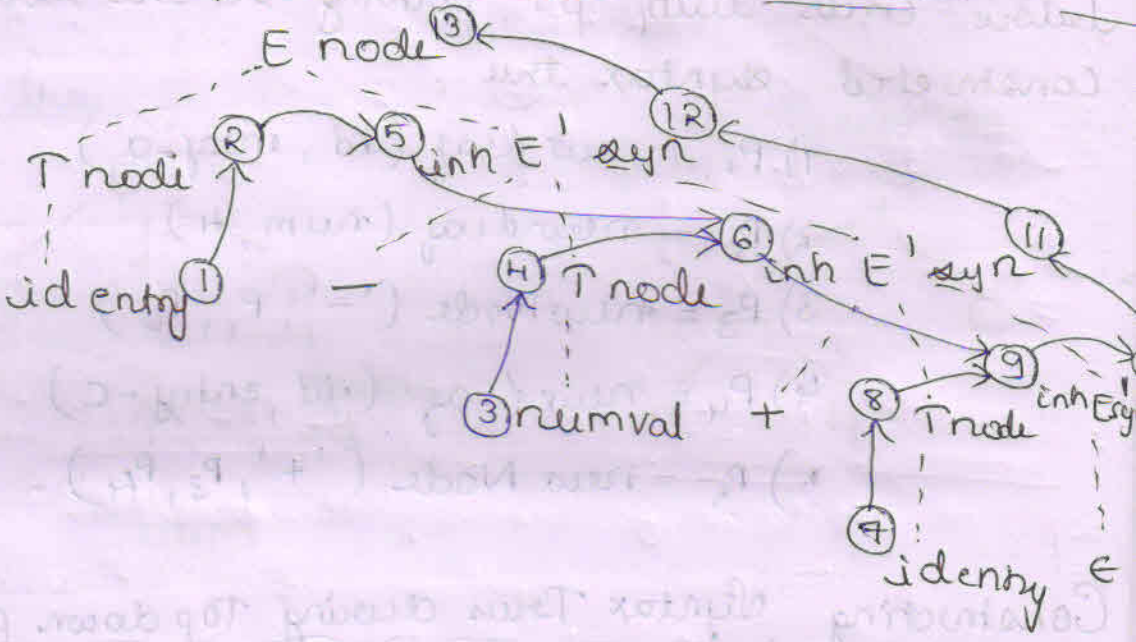
Constructing Syntax Trees during Top down parsing

With a grammar designed for top-down parsing, the syntax trees are constructed, using the same sequence of steps, even though the structure of the parse trees differs significantly from that of syntax trees. The L-attributed definition below performs the same translation as the S attributed definition shown before.

1) $E \to TE'$	E.node = E'.syn E'.inh = T.node
2) $E' \to +TE_1'$	E_1'.inh = ~~E'.inh + T.~~ new Node('+', E'.inh, T.node) E'.syn = E_1'.syn
3) $E' \to -TE_1'$	E'.inh = new Node('-', E'.inh, T.node) E'.syn = E_1'.syn
4) $E' \to \epsilon$	E'.syn = E'.inh
5) $T \to (E)$	T.node = E.node

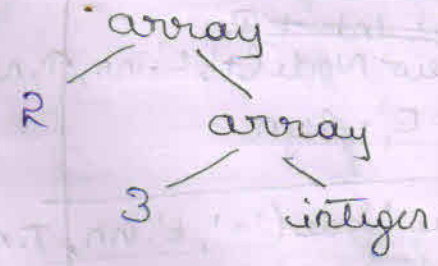
6) T → id	T.node = new Leaf (id, id.entry)
7) T → num	T.node = new Leaf (num, num.val)

Dependency Graph for a-4+c with Latributed SDD



STRUCTURE OF A TYPE

This is an example of how inherited attributes can be used to carry info from one part of the parse tree to another. In C, the type int[2][3] can be read as "array of 2 arrays of 3 integer". The corresponding type expression array(2, array(3, integer)) is represented by the tree as shown below.



product^n	Semantic rules
1) T → BC	T.t = C.t C.b = B.t
2) B → int	B.t = integer
3) B → float	B.t = float
4) C → [num] C_1	C.t = array (num.val, C.t) C_1.b = C.b
5) C → ε	C.t = C.b

→ The non terminals B & T have a synthesized attribute t representing a type

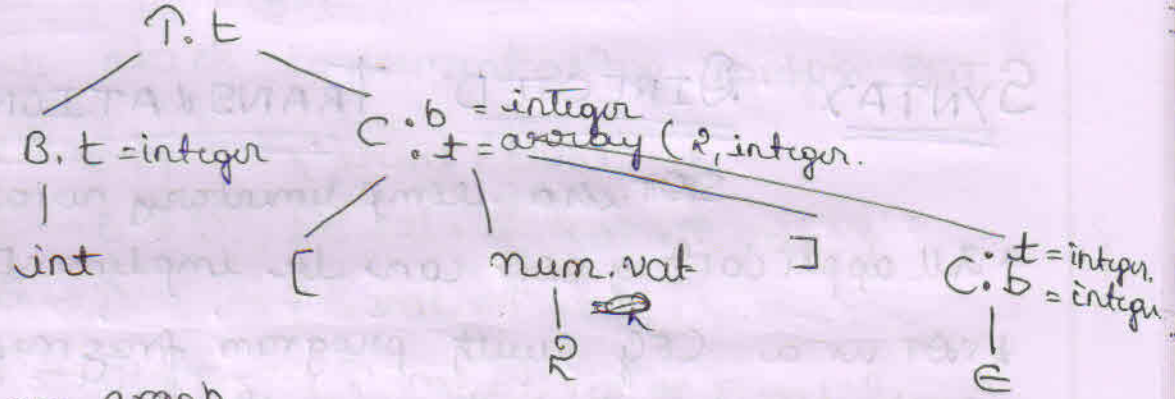
→ The non terminal C has 2 attributes: an inherited attr (b) & a synthesized attr (t).

→ The inherited attribute, b pass a basic type down the tree

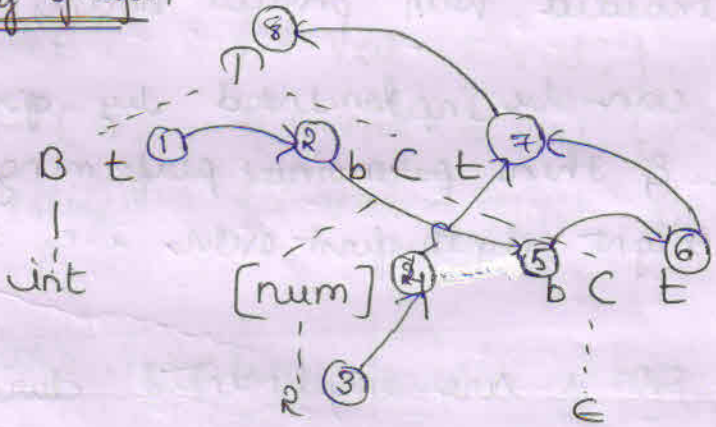
→ The synthesized attribute, t accumulate the result.

→ An annotated parse tree for i/p : int [2]

(i) int [2]

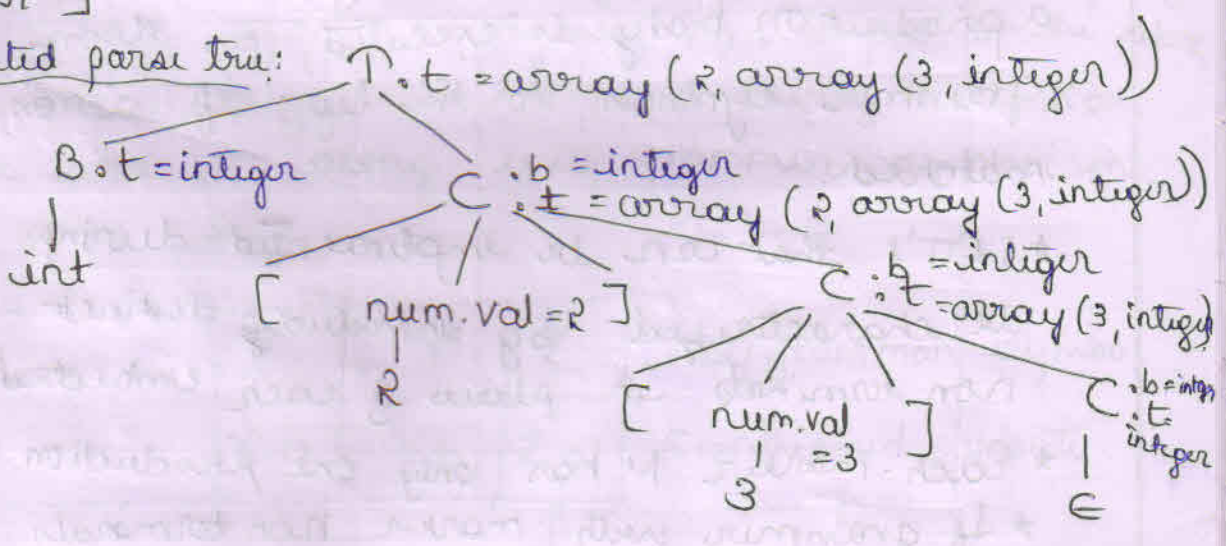


Dependency graph

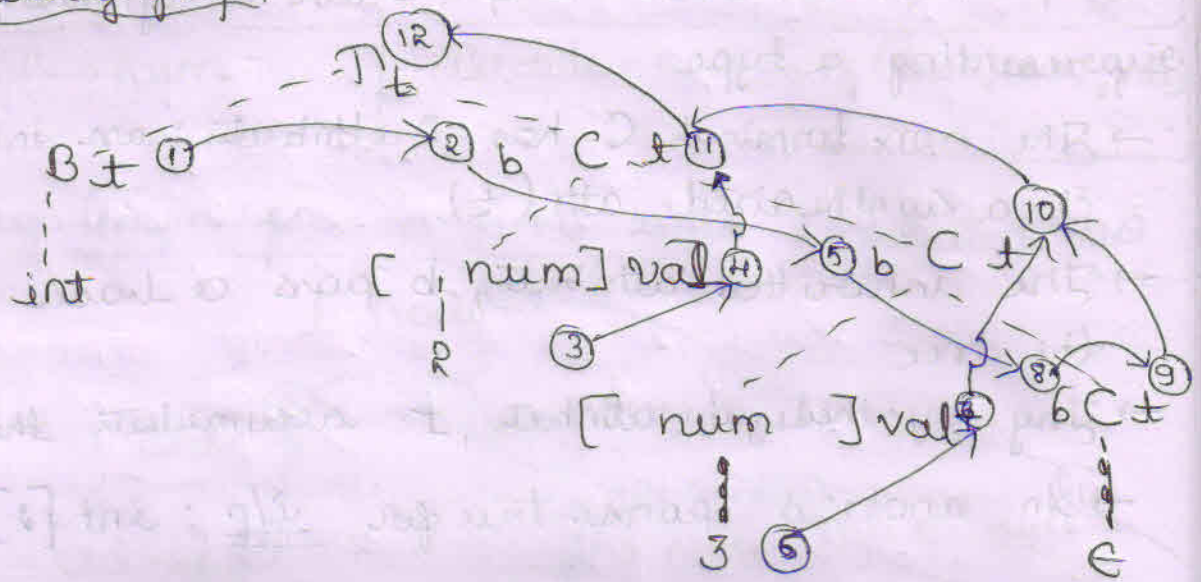


(ii) int [2][3]

Annotated parse tree: T.t = array (2, array (3, integer))



Dependency graph



SYNTAX DIRECTED TRANSLATION SCHEME:

SDT is a complementary notation to SDD.

* All applicat^n of SDD can be implemented using SDT.
* SDT is a CFG with program fragments called semantic actions embedded with product^n bodies.
* Any SDT can be implemented by first building a parse tree & then ~~parsing~~ performing the actions in a left to right, depth first order i.e, during preorder traversal.
* Typically SDT's are implemented during parsing without building parse tree. During parsing, an action in a production body is executed as soon as all the grammar symbols to the left of action have been matched.
* SDT's that can be implemented during parsing can be characterized by introducing distinct marker non terminals in place of each embedded action.
* Each marker M has only one production $M \to \epsilon$.
* If grammer with marker non terminals can be parsed by a given method, then SDT can be implemented

during parsing.

Postfix Translation Scheme

The simplest implementation of SDD occurs when we can parse the grammar bottom up & SDD is S-attributed. Here each semantic action can be placed at the end of production & executed along with the reduction of body to the head of production. This type of SDT is called postfix SDT.

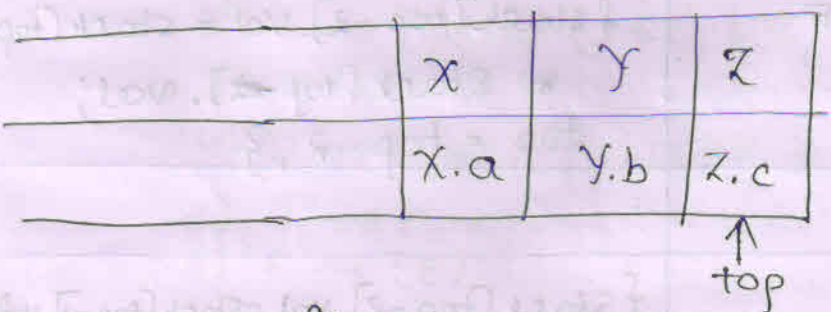
Eg: Postfix SDT for implementing the desk calculator (or)

Parser stack implementation of postfix SDT

$L \to E\,n$	{ print (E.val); }
$E \to E_1 + T$	{ E.val = E_1.val + T.val; }
$E \to T$	{ E.val = T.val; }
$T \to T_1 * F$	{ 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; }

Parser - Stack Implementation of Postfix SDT's:

Postfix SDT's can be implemented during LR parsing. Attribute of each grammar symbol can be put on the stack in place where they can be found during reduction, i.e., place attribute along with grammar symbol in record of stack itself.



fig; parser stack with field for synthesized attr

during parsing.

Postfix Translation Scheme

The simplest implementation of SDD occurs when we can parse the grammar bottom up & SDD is S-attributed. Here each semantic action can be placed at the end of production & executed along with the reduction of body to the head of production. This type of SDT is called postfix SDT.

Eg: Postfix SDT for implementing the desk calculator

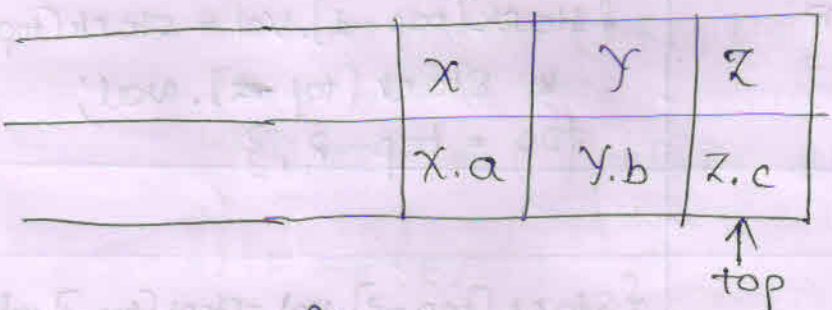
(or)

Parser stack implementation of postfix SDT

$L \to E\,n$	{ print (E.val); }
$E \to E_1 + T$	{ E.val = E_1.val + T.val; }
$E \to T$	{ E.val = T.val; }
$T \to T_1 * F$	{ 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; }

Parser - Stack Implementation of Postfix SDT's:

Postfix SDT's can be implemented during LR parsing. Attribute of each grammar symbol can be put on the stack in place where they can be found during reduction, i.e., place attribute along with grammar symbol in record of stack itself.



fig; parser stack with field for synthesized attr

To implement SDT during LR parsing, add semantic stack parallel to the parsing stack: each symbol (terminal or (non terminal) on the parsing stack stores its value on the semantic stack. It holds terminals attributes & non terminals translations. When the parser is finished, the semantic stack will hold just 1 value the translat^n of the root nonterminal.

Semantic Actions during Parsing

* When shifting,
  - Push the value of the terminal on the semantic stack.
* When reducing,
  - Pop k values from the semantic stack, where k → No of symbols on product^n's RH
  - Push the product^n's value on the semant stack.

* SDT for implementing the desk calculator on a bottom up parsing stack as below.

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

SDT's with actions inside productions

→ Actⁿ can be placed @ any positⁿ in the productⁿ body.

→ Actⁿ is performed immediately after all symbols left to it are parsed.

→ Given $B \to X \{a\} Y$, an action a is done after
a) we have recognized X. (if X is terminal)
or
b) all terminals derived from X (if X is a nonterminal).

→ If bottom up parser is used, then action a is performed as soon as X appears on top of stack.

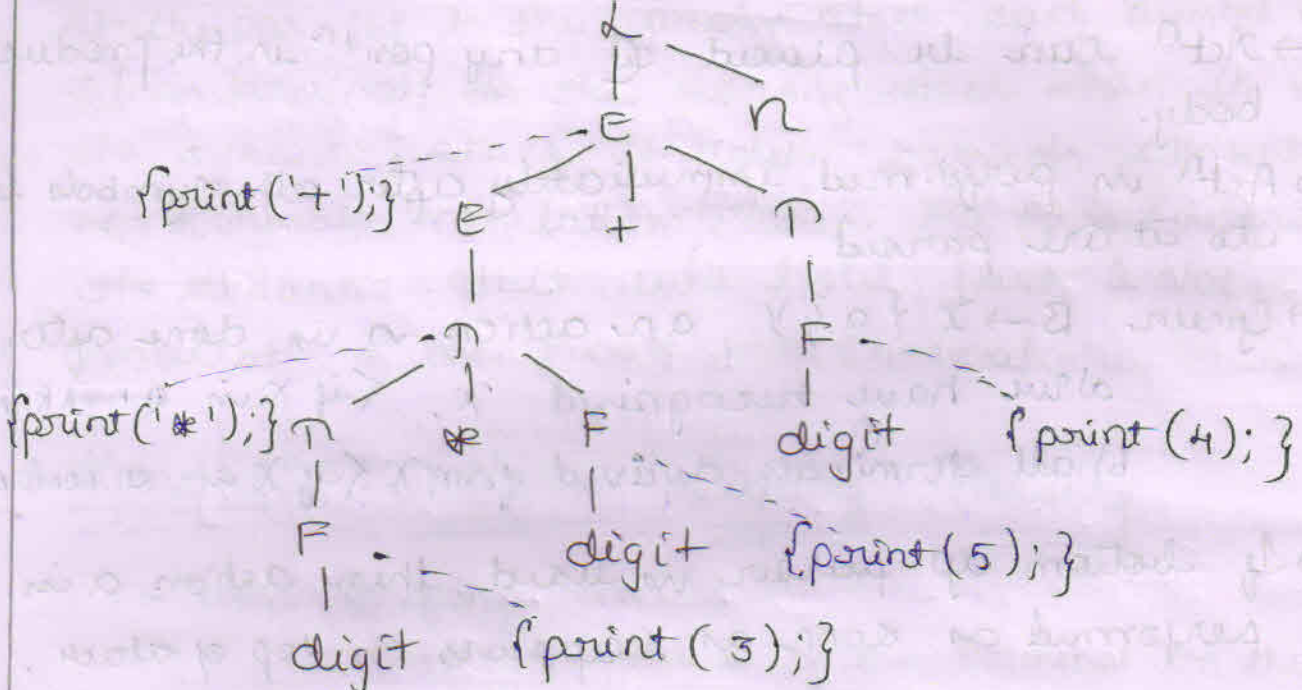
→ If top down parser is used, then action a is performed
a) Just before Y is expanded. (if Y is non terminal)
or
b) Check Y on input. (if Y is a terminal)

→ An SDT can be implemented as follows.
a) Ignoring actions, parse i/p & produce parse tree.
b) Add tional children to node N, for action in α where $A \to \alpha$.
c) Perform preorder traversal of the tree & as soon as a node labeled by an action is visited perform that action.

SDT for infix-to-prefix translation during parsing

1) $L \to En$
2) $E \to \{print('+');\} E_1 + T$
3) $E \to T$
4) $T \to T_1 * F \{print('*');\}$
5) $T \to F$
6) $F \to (E)$
7) $F \to digit \{print(digit.lexval);\}$

Parse tree with action embedded



Eliminating left recursion from SDT's

Grammar with left recursion cannot be parsed during top-down parser. In case of SDT we treat action as terminal symbol in the production then we use the following rule of transforming grammar to non left recursive form.

$$A \to Aa \mid \beta \xrightarrow[\text{transformed to}]{} A \to \beta R, \quad R \to \alpha R \mid \epsilon$$

In both form, A is defined by $\beta(\alpha)^*$

Eg: Given $E \to E_1 + T \{print('+');\}$

$E \to T$

Here $\alpha = +T\{print('+');\}$

& $\beta = T$.

∴ Modified grammar will be

$E \to TR$

$R \to +T\{print('+');\}R$

$R \to \epsilon$

Eg2: $A \rightarrow A_1 Y \{A.a = g(A_1.a, Y.y)\}$

$A \rightarrow X \{A.a = f(X.x)\}$

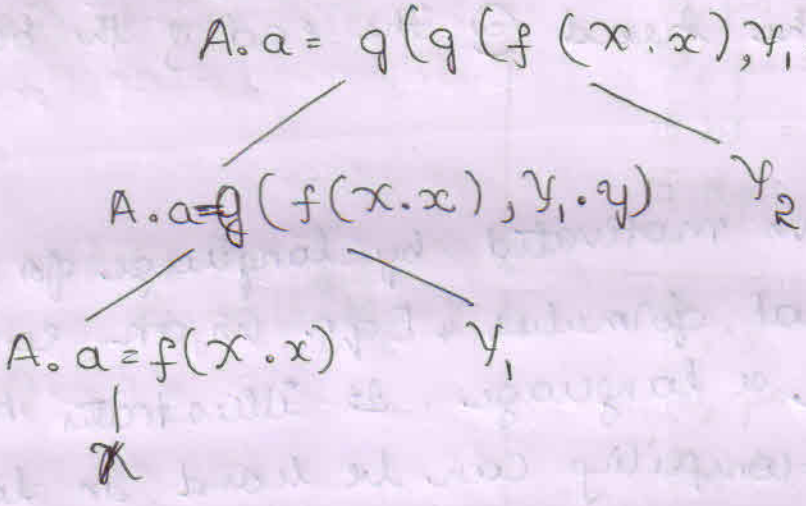
Can be rewritten as,

$A \rightarrow X \{R.i = f(X.x)\}\ R \{A.a = R.s\}$

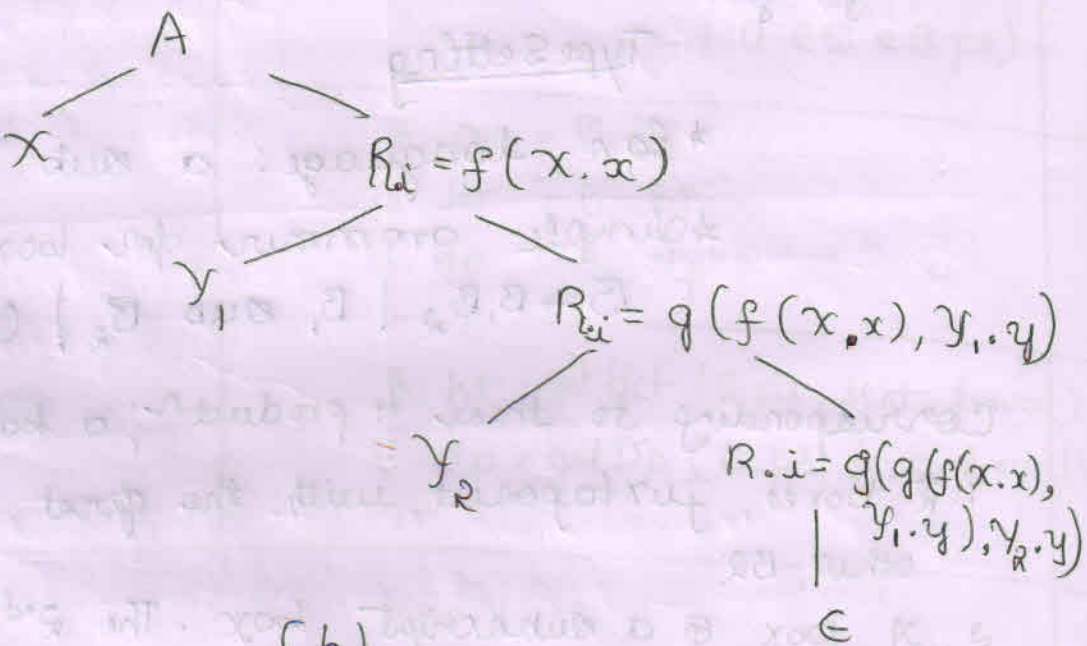
$R \rightarrow Y \{R_1.i = g(R.i, Y.y)\}\ R_1 \{R.s = R_1.s\}$

$R \rightarrow \in \{R.s = R.i\}$

fig; Eliminating left recursion from a postfix SDT



(a)



(b)

SDT's for L-attributed Definations

* It is necessary that the underlying grammar can be parsed top-down.
* Rules to modify L attributed SDD to SDT.
  (i) Embed the actⁿ for computing inherited attr for a non terminal A immediately before A in productⁿ body
  (ii) Place the actⁿ for computing synthesized attr for the head @ the end of the body of productⁿ

Example: This eg is motivated by language for type setting mathematical formulas. Eqn is an early example of such a language. It illustrates how the techniques of compiling can be used in lang processing for applicatⁿ other than programming languages.

TypeSetting

* Eqn language: a sub i sub j ⇒ a_{i_j}
* Simple grammar for boxes

$$B \to B_1 B_2 \mid B_1 \text{ sub } B_2 \mid (B_1) \mid \text{text}$$

Corresponding to these 4 productⁿ, a box can be either
1. A boxes, juxtaposed, with the first, B1, to the left of other, B2.
2. A box of a subscript box. The 2nd box appears in a smaller size, lower & to the right of the 1st box.
3. A parenthesized box, for grouping of boxes & subscripts.
4. A text string, i.e., any string of characters.

fig; Constructing larger boxes from smaller ones.

SDD for Typesetting Boxes

①	$S \rightarrow B$	$B.ps = 10$
②	$B \rightarrow B_1 B_2$	$B_1.ps = B.ps$ $B_2.ps = B.ps$ $B.ht = \max(B_1.ht, B_2.ht)$ $B.dp = \max(B_1.dp, B_2.dp)$
③	$B \rightarrow B_1 \overset{\text{delb}}{} B_2$	$B_1.ps = B.ps$ $B_2.ps = 0.7 * B.ps$ $B.ht = \max(B_1.ht, B_2.ht - 0.25 * B.ps)$ $B.dp = \max(B_1.dp, B_2.dp + 0.25 * B.ps)$
④	$B \rightarrow (B_1)$	$B_1.ps = B.ps$ $B.ht = B_1.ht$ $B.dp = B_1.dp$
⑤	$B \rightarrow$ text	$B.ht = getHit(B.ps, text.lexval)$ $B.dp = getDp(B.ps, text.lexval)$

SDT for typesetting boxes

① $S \to B$	$\{B.ps = 10;\}$
② $B \to B_1$ B_2	$\{B_1.ps = B.ps;\}$ $\{B_2.ps = B.ps;\}$ $\{B.ht = max(B_1.ht, B_2.ht);$ $B.dp = max(B_1.dp, B_2.dp);\}$
③ $B \to B_1$ Sub B_2	$\{B_1.ps = B.ps;\}$ $\{B_2.ps = 0.7 * B.ps;\}$ $\{B.ht = max(B_1.ht, B_2.ht - 0.25 * B.ps);$ $B.dp = max(B_1.dp, B_2.dp + 0.25 * B.ps);\}$
④ $B \to (B_1)$	$\{B_1.ps = B.ps;\}$ $\{B.ht = B_1.ht;$ $B.dp = B_1.dp;\}$
⑤ $B \to$ text	$\{B.ht = getHit(B.ps, text.lexval);$ $B.dp = getDP(B.ps, text.lexval);$

Eg2: 2nd example is about the generatn of intermediate code for a typical programming language construct : a form of while statement.

- SDD for while statements

$S \rightarrow$ while (c) S_1

$\alpha_1 = new();$

$\alpha_2 = new();$

$S_1.next = \alpha_1;$

$C.false = S.next;$

$C.true = \alpha_2;$

$S.code = label \,||\, \alpha_1 \,||\, C.code \,||\, label \,||\, \alpha_2 \,||\, S_1.code$

SDT for while statements

$S \rightarrow$ while ($\{\alpha_1 = new(); \alpha_2 = new();$
$C.false = S.next; C.true = \alpha_2; \}$

C)

$\{ S_1.next = \alpha_1;$

S_1

$\{ S.code = label \,||\, \alpha_1 \,||\, C.code \,||$
$label \,||\, \alpha_2 \,||\, S_1.code; \}$

—x—

Explanation has to be written for this question