

## UNIT - 5

### SYNTAX - DIRECTED TRANSLATION

#### SYNTAX-DIRECTED DEFINITION (SDD) :

- SDD is a context free grammar (C.F.G) with attributes and rules.
- Attributes are associated with grammar symbol and and Semantic rule are associated with production

Example:

<u>Production</u>	<u>Semantic Rule</u>
$E \rightarrow E_1 + T$	$E.\text{val} = E_1.\text{val} + T.\text{val}$

|      |      |      attributes

- If  $X$  is a symbol and ' $a$ ' is attribute then we write  $X.a$  to denote the value of ' $a$ ' at a particular parse tree labeled  $X$ .
- If we implement nodes of the parse tree by records or objects then the attributes of  $X$  can be implemented by data fields in the record that represents the node of  $X$ .
- Attributes may be any kind: number, table reference, type or string.

#### Inherited and Synthesized Attributes:

##### 1. Synthesized Attribute:

- A synthesized attribute at a parse-tree node is computed from attributes at its children and by itself.

Attribute value of a non-terminal derived from the attribute value of its children or itself called synthesized attribute



- In a Parse tree value of synthesized attribute at a node is computed from the value of attributes at the children of that node and that node itself.
- Terminals have only synthesized attributes.

Example for Synthesized attributes:

SDD for Simple Desk Calculator or Simple Arithmetic Exp.:

- Here, it evaluates expressions terminated by an endmark ( $\sqcup$ ).
- Each non-terminal has a single synthesized attribute called 'val'.
- Digit has synthesized attribute called 'lexval' which is an integral value returned by Lexical Analyzer.

### Production

$$L \rightarrow E_n$$

$$E \rightarrow E_1 + T$$

$$E \rightarrow T$$

$$T \rightarrow T_1 * F$$

$$T \rightarrow F$$

$$F \rightarrow (E)$$

$$F \rightarrow \text{digit}$$

### Semantic Rule

$$L\text{-val} = E\text{-val}$$

$$E\text{-val} = E_1\text{-val} + T\text{-val}$$

$$E\text{-val} = T\text{-val}$$

$$T\text{-val} = T_1\text{-val} * F\text{-val}$$

$$T\text{-val} = F\text{-val}$$

$$F\text{-val} = E\text{-val}$$

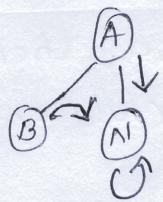
$$F\text{-val} = \text{digit. lexval}$$

SDD for Simple Desk calculator

SDD with only synthesized attribute then it is called S-attributed SDD.

## Inherited attributes:

- An inherited attribute at node  $N$  is defined only in terms of attribute values at  $N$ 's parent,  $N$  itself and  $N$ 's siblings.



- The attribute value of a non-terminal derived from the attribute values of its siblings or from its parent or itself is called inherited attribute.

## Example for Inherited attributes:

### Production

$$T \rightarrow FT'$$

$$T' \rightarrow *FT'_1$$

$$T' \rightarrow \epsilon$$

$$F \rightarrow \text{digit}$$

### Semantic Rule:

$$T'.\text{val} = F.\text{val}$$

$$T'.\text{val} = T'.\text{syn}$$

$$T'.\text{inh} = T'.\text{inh} \times F.\text{val}$$

$$T'.\text{syn} = T'_1.\text{syn}$$

$$T'.\text{syn} = T'.\text{inh}$$

$$F.\text{val} = \text{digit. lexical}$$

- In the above SDD,  $T$  and  $F$  have synthesized attribute called 'val'
- $\text{digit}$  also has a synthesized attribute called 'lexical'.
- $T'$  has two attributes synthesized called syn and inherited called inh.

## Evaluating an SDD at the Nodes of a Parse Tree

### Circular Dependency:

- If the attribute value of a Parent node depends on the attribute value of child node and Vice-Versa, then there exists a circular dependency between the child and parent node.
- In this situation, it is not possible to evaluate the attribute of either parent node or the child node since one value depends on another value.

### Example:

consider the non-terminal A with synthesized attribute A.s and non-terminal B with inherited attribute B.i with following productions and Semantic Rules.

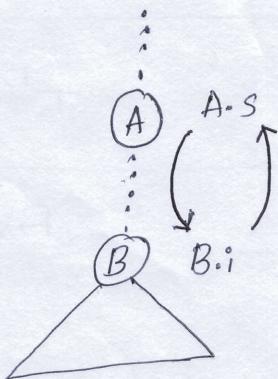
#### Production

$$A \rightarrow B$$

#### Semantic Rule:

$$A.s = B.i ;$$

$$B.i = A.s + 1$$



- The above two Semantic Rules are circular in nature. Note that to compute A.s we require value of B.i and to compute B.i we require the value of A.s.
- So it is impossible to evaluate either the value of A.s or the value of B.i without evaluating others.

## Annotated Parse tree:

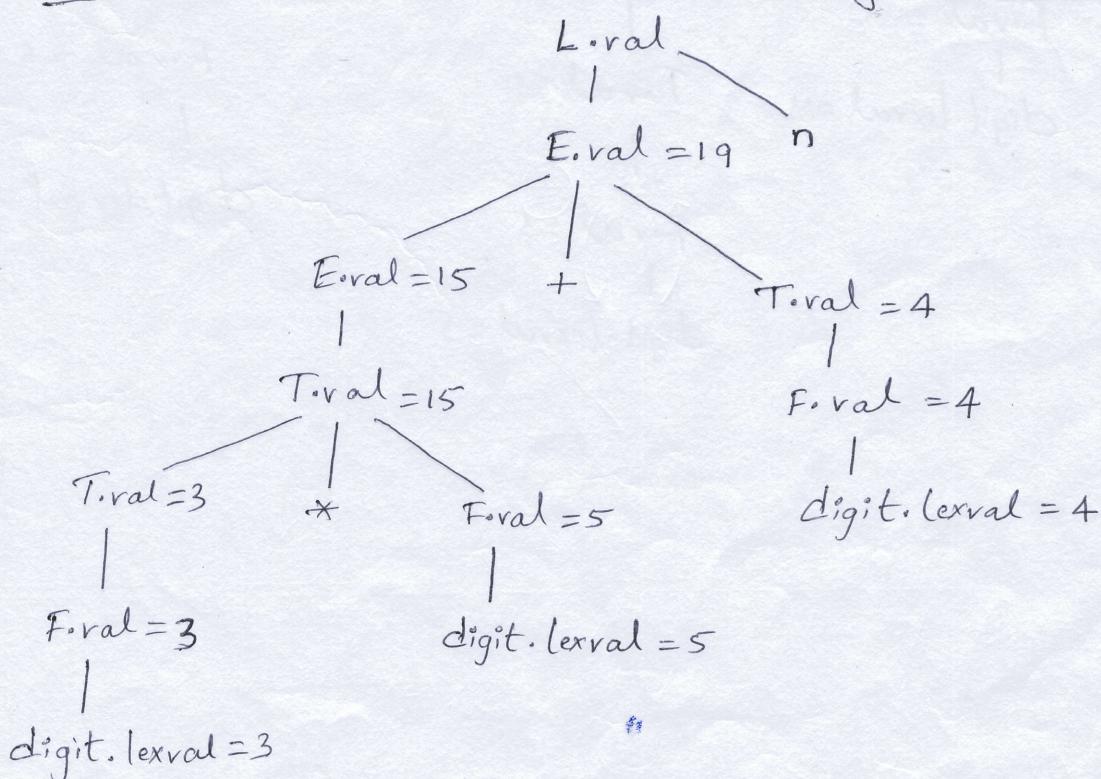
- A parse tree showing the attribute values of each node is called annotated parse tree.
- Before evaluate an attribute at a node of a parse tree we must evaluate all attributes upon which its value depends.
- If attributes are synthesized then first evaluate Val attribute of all the children of a node before evaluate the val attribute of the node itself.

## Problems:

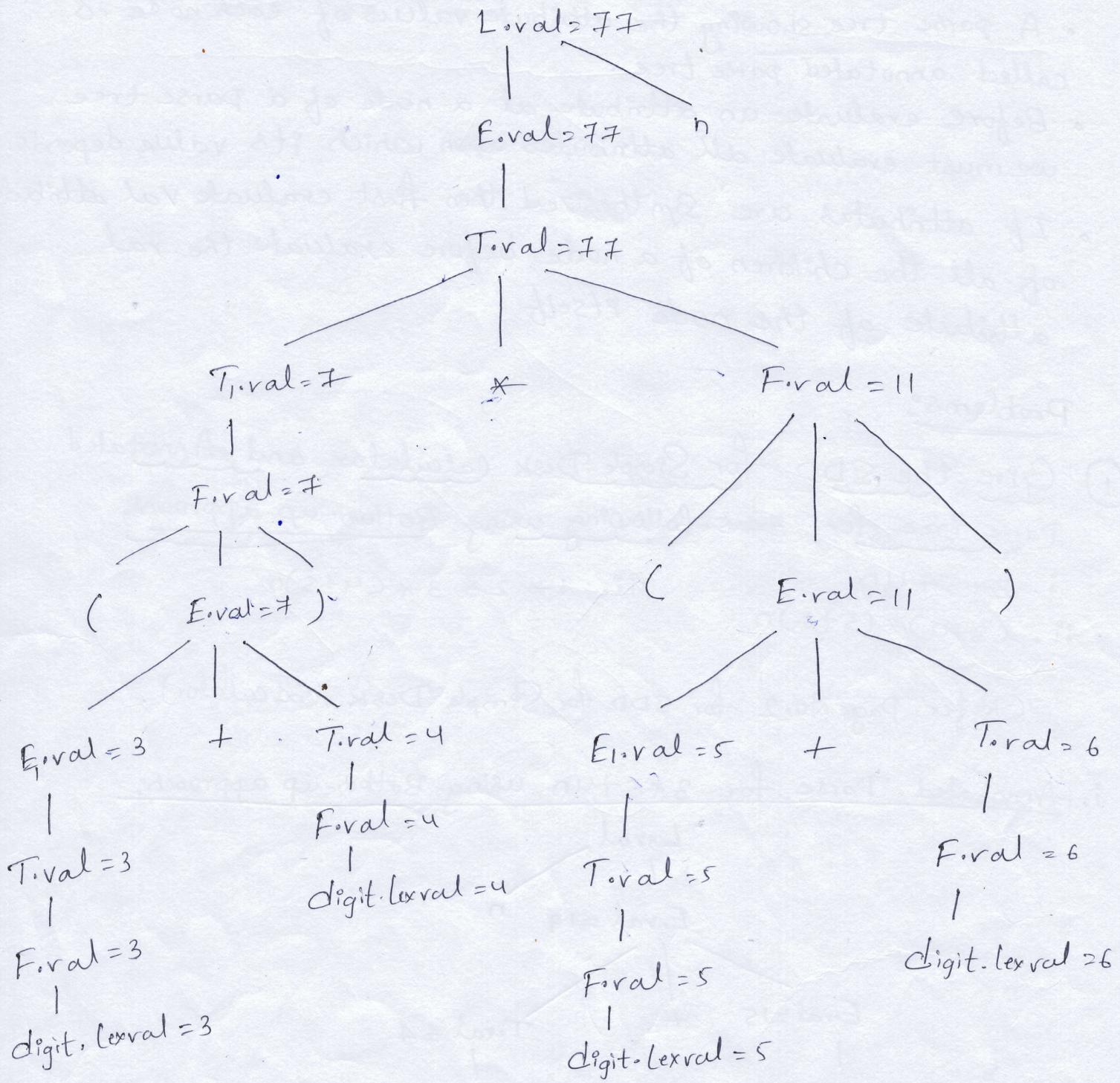
- ① Give the SDD for Simple Desk Calculator and Annotated Parse tree for ~~3\*5~~ following using Bottom-up approach.
- i.  $3 * 5 + 4n$
  - ii.  $(3+4) * (5+6)n$
  - iii.  $1 * 2 * 3 * (4+5)n$

(Refer Page No. 2 for SDD for Simple Desk calculator).

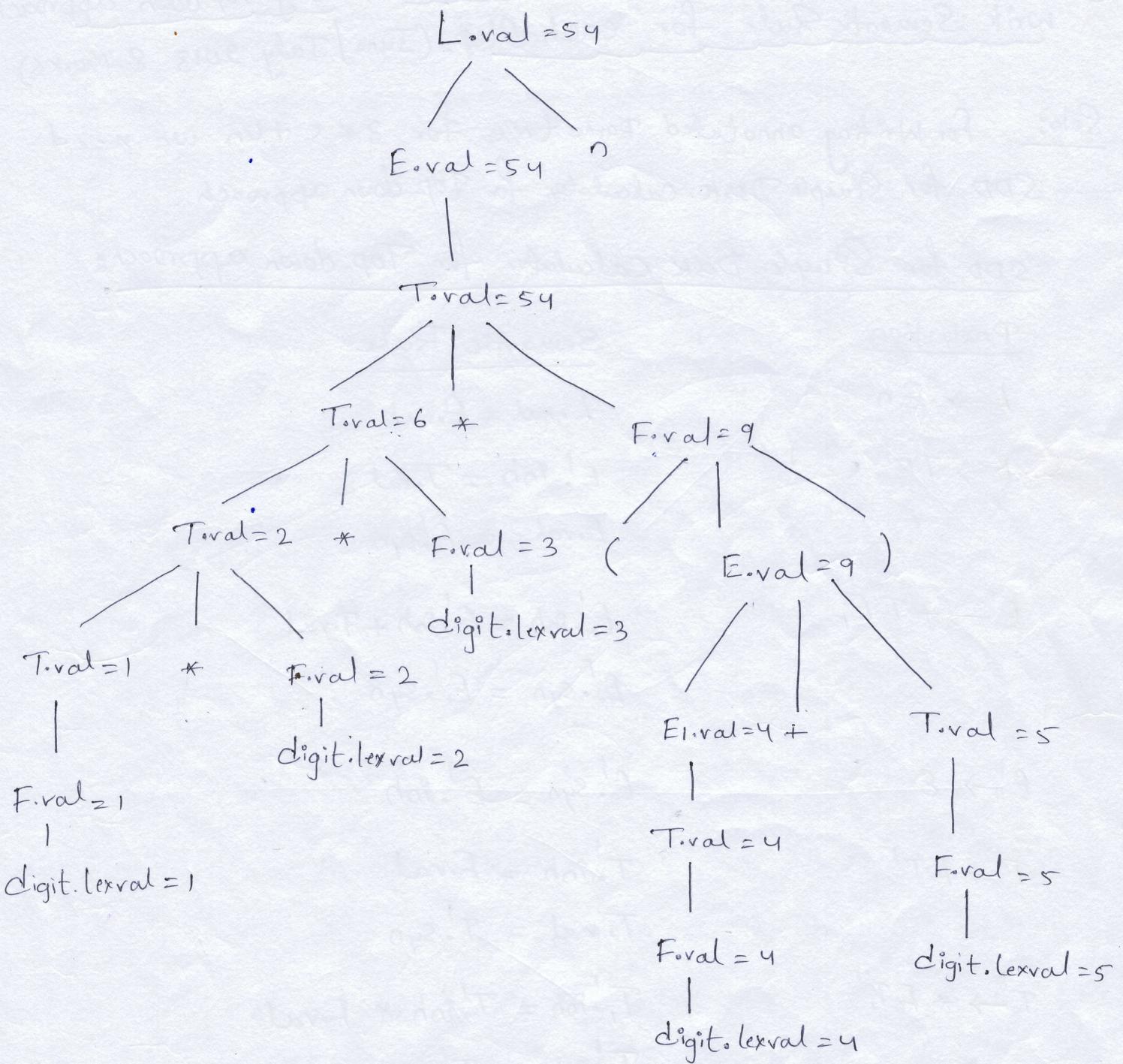
### i. Annotated Parse tree for $3 * 5 + 4n$ using Bottom-up approach.



ii.  $(3+4) * (5+6)$  Annotated Parse tree for Bottom-up approach:



iii) Annotated Parse tree for  $1 * 2 * 3 * (4 + 5) \cdot n$ :



## SDD and Annotated parse tree for Top-down approach:

- ① Write annotated parse tree for  $3 * 5 + 4 n$  using Top down approach.  
 Write Semantic Rule for each Step. (June / July 2013 8-Marks)

Soln: For writing annotated Parse tree for  $3 * 5 + 4 n$  we need SDD for Simple Desk calculator for Top-down approach.

### SDD for Simple Desk calculator for Top-down approach:

#### Production

$$L \rightarrow E_n$$

$$E \rightarrow T E'$$

$$E' \rightarrow + T E_1'$$

$$E' \rightarrow \epsilon$$

$$T \rightarrow F T'$$

$$T' \rightarrow * F T_1'$$

$$T' \rightarrow \epsilon$$

$$F \rightarrow ( E )$$

$$F \rightarrow \text{digit}$$

#### Semantic Rule

$$L.\text{val} = E.\text{val}$$

$$E'.\text{inh} = T.\text{val}$$

$$E.\text{val} = E'.\text{syn}$$

$$E_1'.\text{inh} = E'.\text{inh} + T.\text{val}$$

$$E'.\text{syn} = E_1'.\text{syn}$$

$$E'.\text{syn} = E'.\text{inh}$$

$$T'.\text{inh} = F.\text{val}$$

$$T.\text{val} = T'.\text{syn}$$

$$T_1'.\text{inh} = T'.\text{inh} * F.\text{val}$$

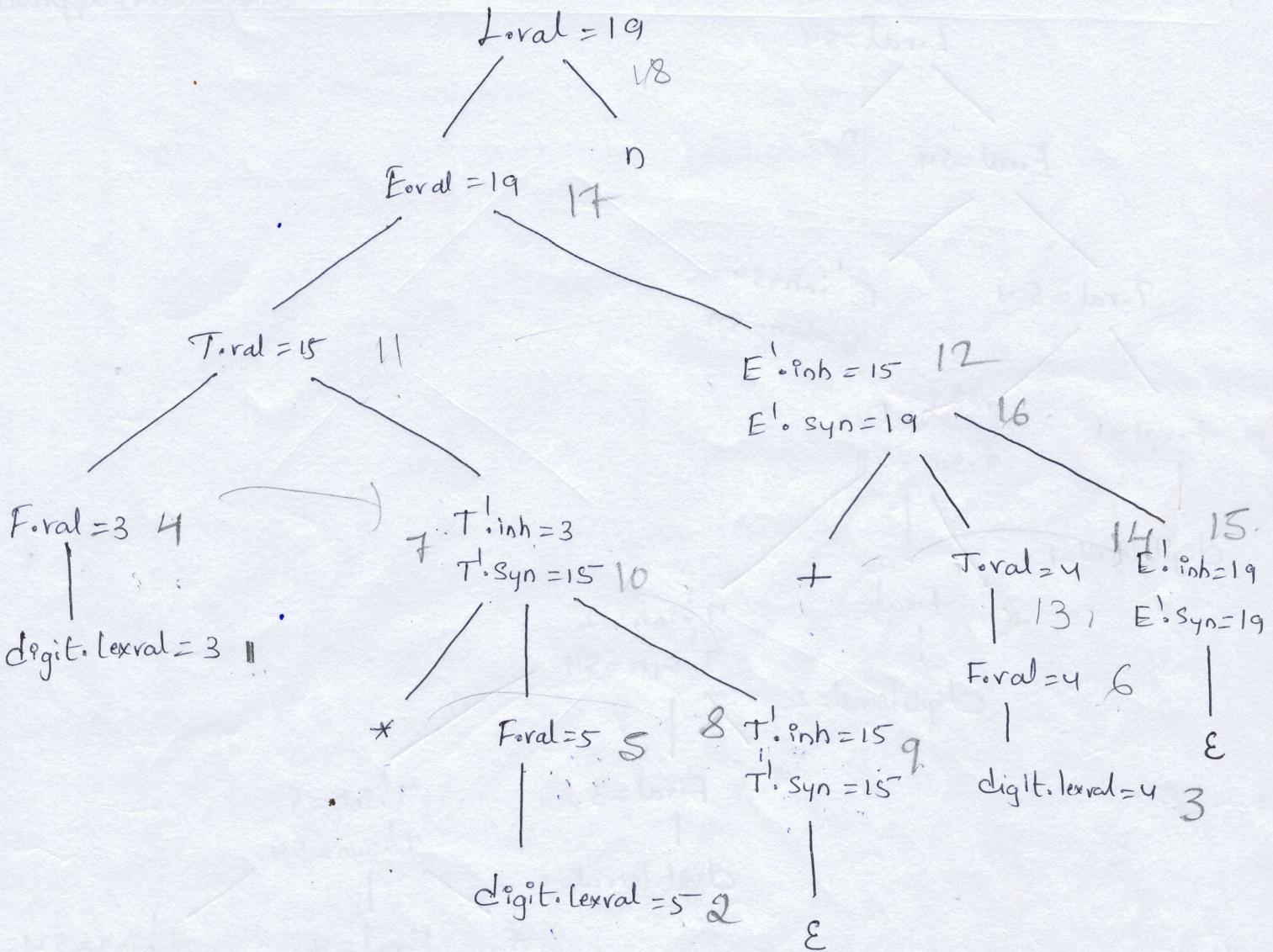
$$T'.\text{syn} = T_1'.\text{syn}$$

$$T'.\text{syn} = T'.\text{inh}$$

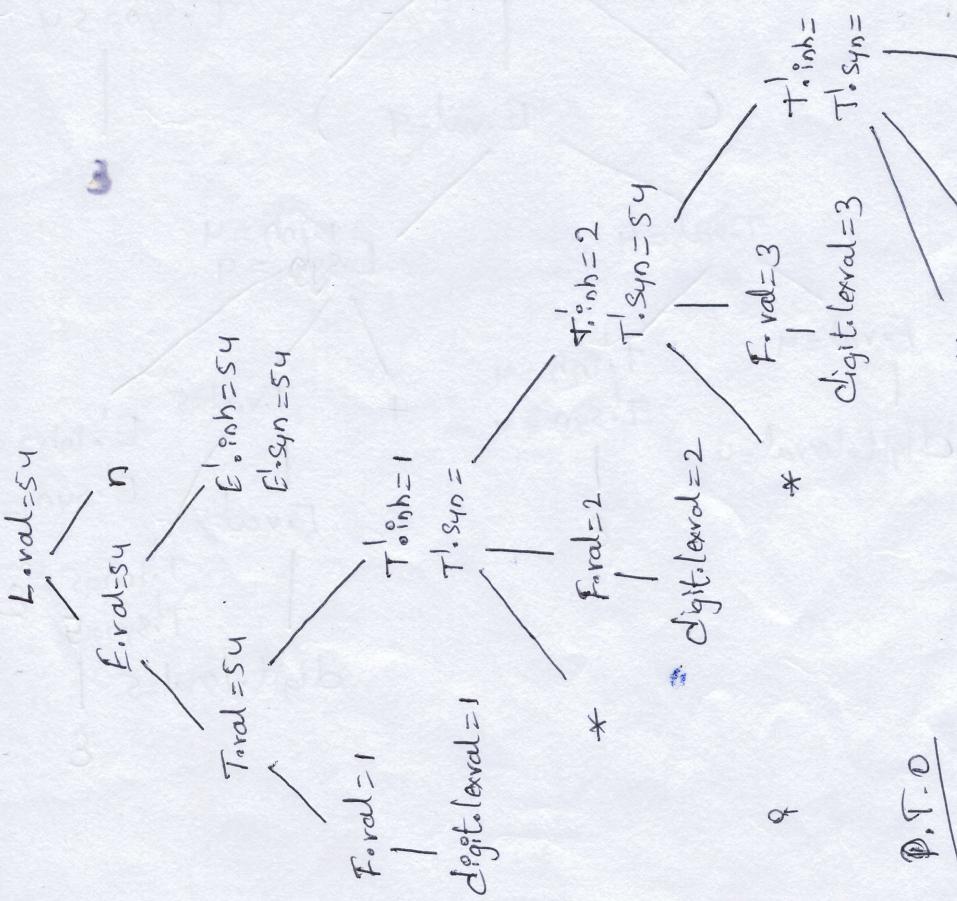
$$F.\text{val} = E.\text{val}$$

$$F.\text{val} = \text{digit}. \text{Lexval}$$

Annotated parse tree for  $3 * 5 + 4 * 1$  for Top-down approach

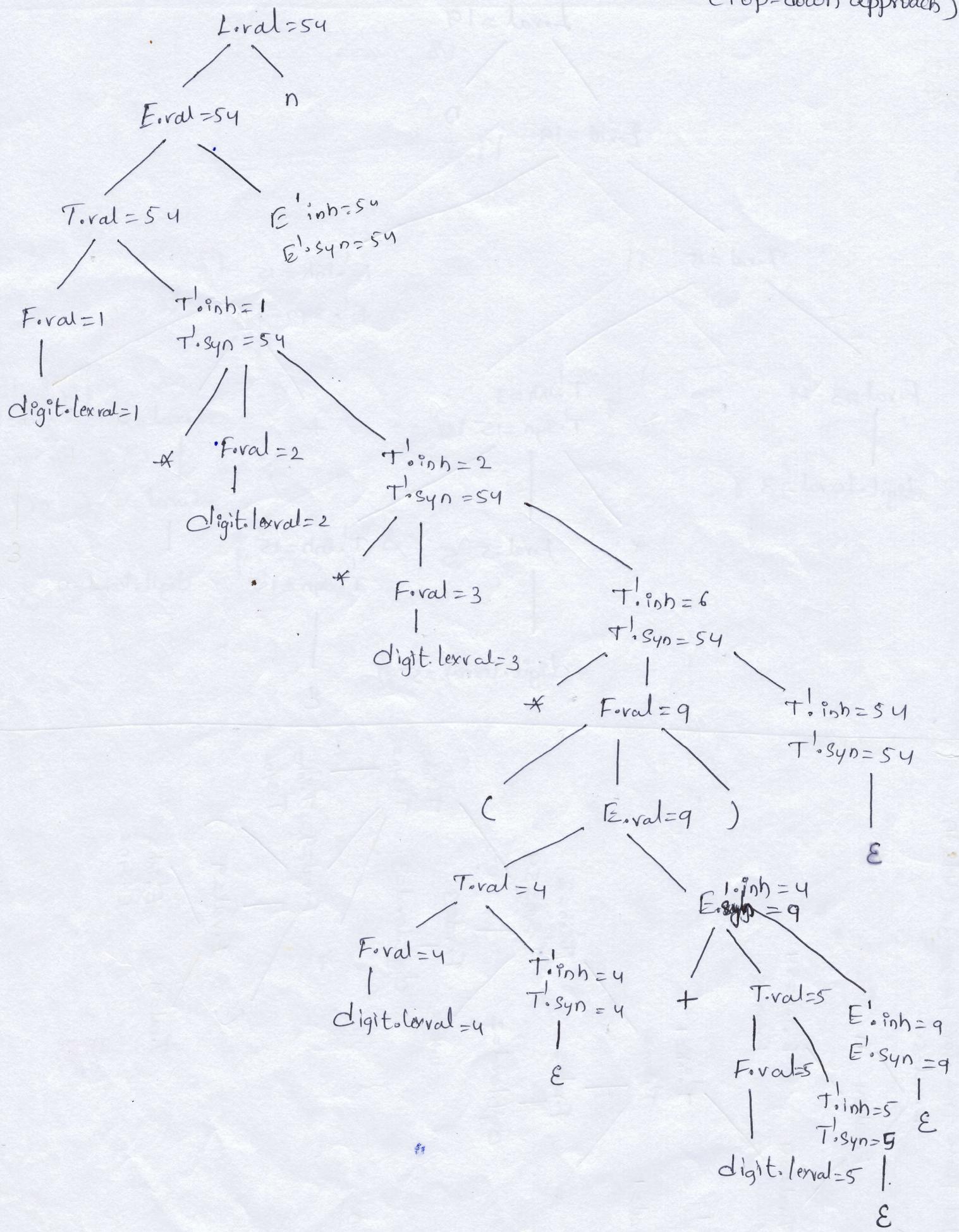


Annotated Parse tree for  $1 * 2 * 3 * (4 + 5) * n$



Annotated Parse tree for  $1 * 2 * 3 * (4 + 5) \cdot n$ .

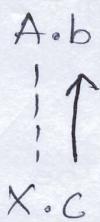
(Top-down approach)



## Evaluation Orders for SDD's:

### Dependency Graphs:

- \* A dependency graph shows the flow of information among the attribute instances in a particular parse tree, an edge from one attribute instances to another means that the value of the first is needed to compute the second.
- \* For each parse tree node, dependency graph has a node for each attribute associated with that node.
- \* If the value of a synthesized attribute  $A.b$  defined in term of  $X.c$ , then the dependency graph has an edge from  $X.c$  to  $A.b$ .



- \* If P defines the value of a inherited attribute  $B.c$  in terms of the value of  $X.a$ , the dependency graph has an edge from  $X.a$  to  $B.c$

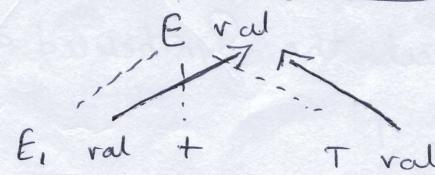
Example:      Production

$$E \rightarrow E_1 + T$$

Semantic Rule

$$E\text{-val} = E_1\text{-val} + T\text{-val}$$

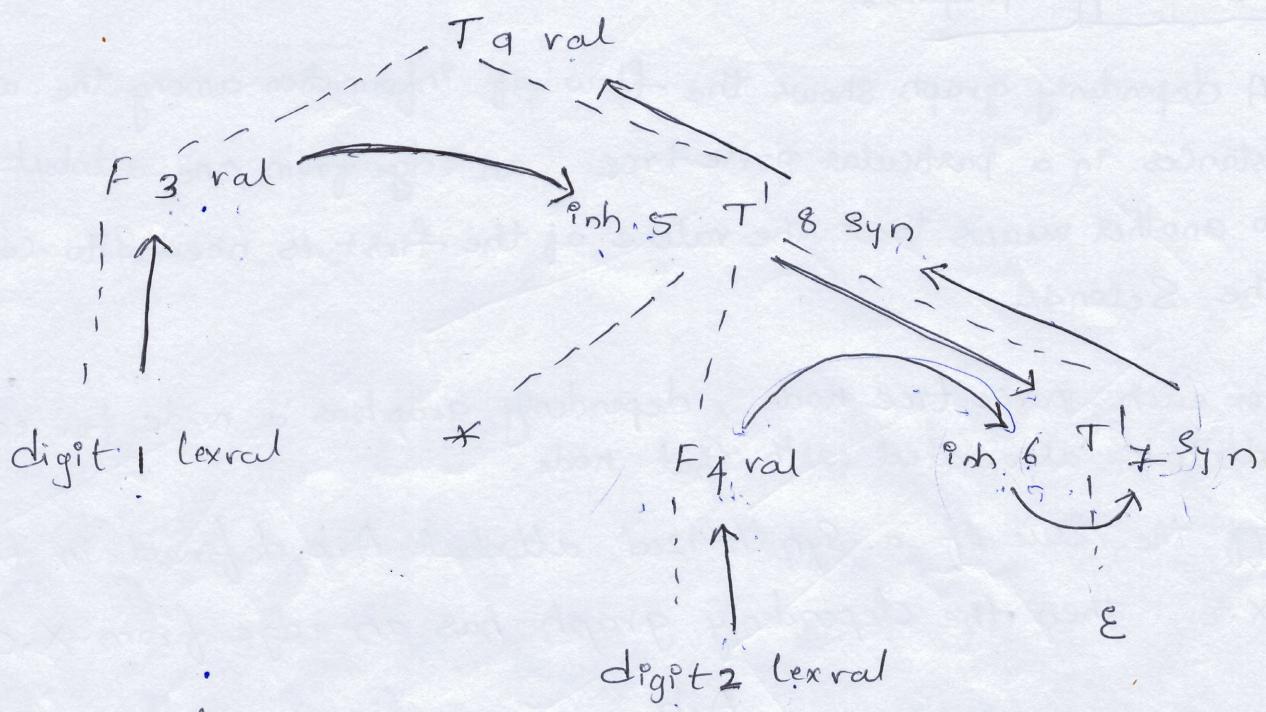
Dependency graph:



Here, dotted line show the parse tree for  $E \rightarrow E_1 + T$  and solid lines shows the dependency edges from  $E_1$  and  $T$  towards  $E$ .

Here,  $E\text{-val}$  synthesized attribute depends on the value of  $E_1\text{-val}$  and  $T\text{-val}$ .

Dependency graph for Annotated parse tree  $3 * 5$  is as below



In above figure:

- \* Node 1 and Node 2 Represent the attribute `lexval` associated with the two leaves labeled `digit`.
- \* Node 3 and Node 4 Represent the attribute `val` associated with the two nodes labeled `F`.
- \* The edges to node 3 from 1 and note 4 from 2 result from Semantic Rule that defines `F.val` in term of `digit.lexval`.
- \* Node 5 and 6 Represents the Inherited attribute `T'.inh` associated with the occurrence of `T'`.
- \* Node 7 and 8 Represents the Synthesized attribute `T'.syn` associated with the occurrence of `T'`.
- \* Edge to node 7 from 6 is due to the Semantic Rule `T'.syn = T'.inh`.
- \* Finally Edge 9 Represents the attribute value `T'.val`.

## L-attributed Definition:

- L-attributed definition is such a SDD in which all attributes are Synthesized attributes or it even contains Inherited attributes with the following rules

Suppose that there is a production

$$A \rightarrow X_1 X_2 X_3 \dots X_n \text{ then}$$

- The Inherited attribute  $X_i.inh$  is associated with the inherited attribute of the head A.
- The Inherited attribute  $X_i.inh$  is associated with either inherited or synthesized attributes of symbols  $X_1 X_2 \dots X_{i-1}$  which are located to the left of  $X_i$ .
- The Inherited attribute  $X_i.inh$  is associated with either inherited or synthesized attribute of  $X_i$  itself.

### Example:

Production	Semantic Rule
$T \rightarrow FT'$	$T'.inh = F.ral$ $T.ral = T'.syn$
$T' \rightarrow *FT_1'$	$T_1.inh = T'.inh * F.ral$ $T'.syn = T_1'.syn$

is a L-attributed SDD.

## Ordering the evaluation of attributes:

- \* The topological sort of the dependency graph decides the evaluation order in a parse tree.
- \* Topological sort of a directed graph is sequence of nodes which gives the order in which the various attributes values can be computed in a parse tree.
- \* Using the dependency graph, we can write the order in which we can evaluate various attribute values in the parse tree

• Topological sort of the dependency graph given in Page 12 (previous page)

Topological sort 1 : 1, 2, 3, 4, 5, 6, 7, 8, 9

— , — 2 : 1, 3, 2, 4, 5, 6, 7, 8, 9

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

4 : 2, 4, 1, 3, 5, 6, 7, 8, 9 and etc

## S-Attributed Definitions:

- \* S-attributed definition is such a syntax directed definition (SDD) in which all attributes are synthesized attributes.
  - consider the following Ex:

Production	Semantic Rule
$F \rightarrow E_n$	$L_{oral} = E_{oral}$
$E \rightarrow E_1 + T$	$E_{oral} = E_1{oral} + T{oral}$
$E \rightarrow T$	$E_{oral} = T{oral}$
$T \rightarrow F$	$T{oral} = F{oral}$
$F \rightarrow \text{digit}$	$F{oral} = \text{digit}.L_{oral}$

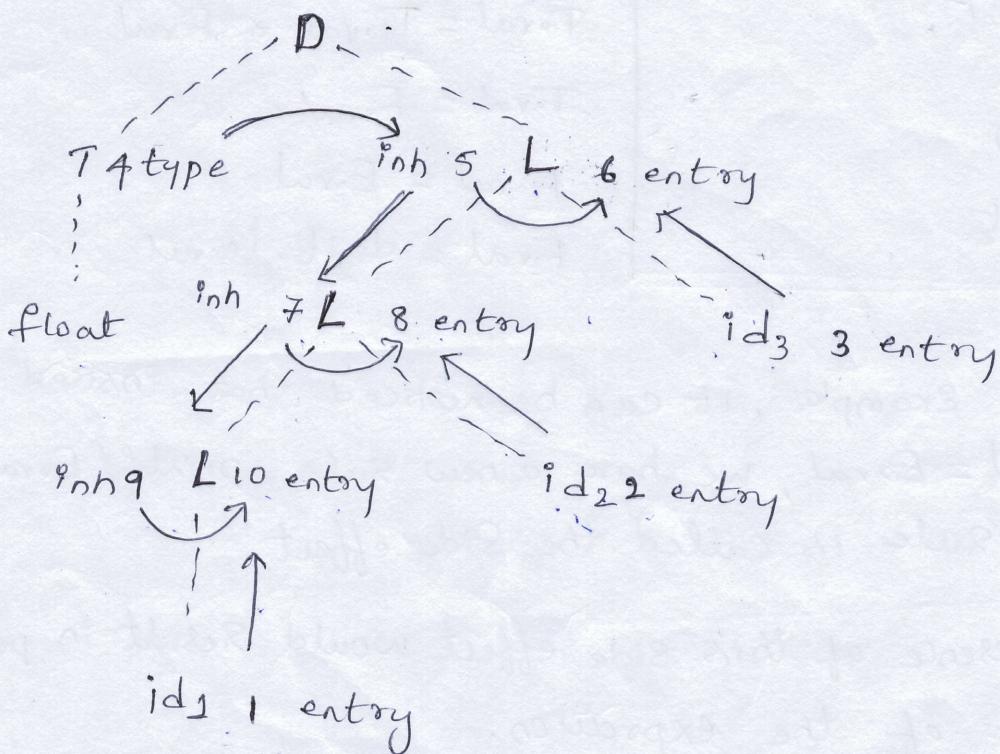
• As notice in this example, all the attributes are synthesized attribute and hence the SDD is called as S-attributed SDD.

## SDD for Simple type Declaration

- ① Write the SDD for Simple type declaration and write Dependency graph for declaration float id<sub>1</sub>, id<sub>2</sub>, id<sub>3</sub>. (8 Marks).

<u>Production</u>	<u>Semantic rule</u>
$D \rightarrow T \ L$	$L.\text{inh} = T.\text{type}$
$T \rightarrow \text{integer}$	$T.\text{type} = \text{integer}$
$T \rightarrow \text{float}$	$T.\text{type} = \text{float}$
$L \rightarrow L_1, \text{id}_1$	$L_1.\text{inh} = \text{float}$ $\text{addType}(\text{id}_1.\text{entry}, L.\text{inh})$
$L \rightarrow \text{id}_d$	$\text{addType}(\text{id}_d.\text{entry}, L.\text{inh})$

Dependency graph for float id<sub>1</sub>, id<sub>2</sub>, id<sub>3</sub>.



## Semantic Rules with controlled Side Effects:

Semantic rules may sometimes be associated with some side effects. Some of the side effects associated with Semantic rules are

(i) printing the final result of evaluation of an exp.

(ii) enter the data-type of a variable into the symbol table

Eg: consider the following grammar

$$L \rightarrow E_n$$

$$E \rightarrow E_1 + T$$

$$E \rightarrow T$$

$$T \rightarrow T_1 * F$$

$$F \rightarrow F$$

$$F \rightarrow (E)$$

$$F \rightarrow \text{digit}$$

The same CFG with side effect is as shown below

$$\text{printf}(E\text{-val})$$

$$E\text{-val} = E_1\text{-val} + T\text{-val}$$

$$E\text{-val} = T\text{-val}$$

$$T\text{-val} = T_1\text{-val} * F\text{-val}$$

$$T\text{-val} = F\text{-val}$$

$$F\text{-val} = E\text{-val}$$

$$F\text{-val} = \text{digit.lexval}$$

\* In above example, it can be noticed that instead of the rule  $L\text{-val} = E\text{-val}$ , we have a new rule  $\text{printf}(E\text{-val})$ .

This new rule is called the Side effect.

\* The presence of this side effect would result in printing the value of the expression.

\* The nodes of Syntax tree can be implemented by creating objects, where each object containing two or more fields.

### 1. If the node is leaf Node:

- A constructor function  $\text{leaf}(\text{op}, \text{val})$  creates a leaf object.

Val - holds the lexical value for the leaf

OP - holds the label for the node.

### 2. If the node is an Interior node:

- A constructor function  $\text{Node}(\text{op}, c_1, c_2 \dots c_k)$  creates an interior object.
- $c_1, c_2 \dots c_k$  - indicates children.
- op - holds the label for the node.

### Syntax-tree for S-attributed Def<sup>n</sup> (Bottom-up approach):

① Assuming Suitable SDD, construct Syntax-tree for a-4+e (8marks)

Production	Semantic Rule
$E \rightarrow E_1 + T$	$E.\text{node} = \text{new Node}(+, E_1.\text{node}, T.\text{node})$
$E \rightarrow E_1 - T$	$E.\text{node} = \text{new Node}(-, E_1.\text{node}, T.\text{node})$
$E \rightarrow T$	$E.\text{node} = T.\text{node}$
$T \rightarrow (E)$	$T.\text{node} = E.\text{node}$
$T \rightarrow \text{id}$	$T.\text{node} = \text{new leaf}(\text{id}, \text{id}.\text{entry})$
$T \rightarrow \text{num}$	$T.\text{node} = \text{new leaf}(\text{num}, \text{num}.\text{val})$

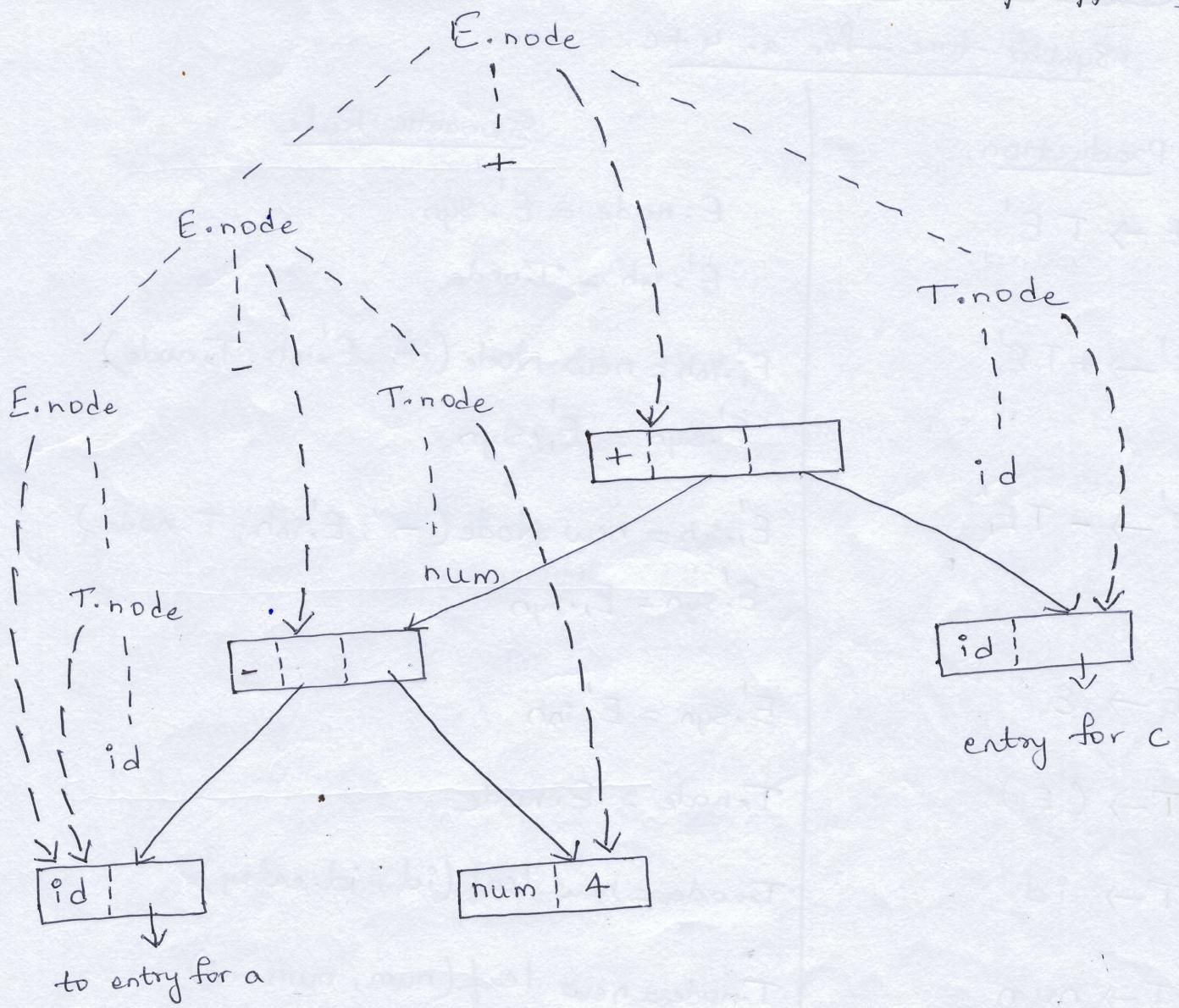
## Application of Syntax-Directed Translation (SDT)

- \* SDT is a C.F.G (Context free Grammar) with embedded Semantic actions. The Semantic actions are the Sequence of steps or program fragments that will be carried out when that production is used in the derivation
- \* Syntax-Directed Translation are used:
  - To build Syntax tree for programming constructs.
  - To translate infix Exp into postfix exp.
  - To evaluate expressions

### 1. Construction of Syntax Tree:

- \* Syntax tree is also called abstract Syntax tree is a compressed form of parse tree which is used to represent language constructs.
  - \* In a Syntax tree of an expression each interior node represents an operator and the children of the node represent the operands of the operator.
  - \* In Syntax tree , All the operators and Keywords appear as interior nodes
- In parse tree , All operators and Keywords appear as leaf node .

Syntax tree for  $a - 4 + c$ . (s-attributed Def<sup>n</sup> or Bottom-up approach)



Steps in the construction of the Syntax tree for  $\underline{a - 4 + c}$

$P_1 = \text{new leaf}(\text{id}, \text{entry}-a);$

$P_2 = \text{new leaf}(\text{num}, 4);$

$P_3 = \text{new Node}('-', P_1, P_2);$

$P_4 = \text{new leaf}(\text{id}, \text{entry}-c);$

$P_5 = \text{new Node}('+', P_3, P_4);$

## Syntax tree for L-attributed Def' (Top-down approach):

### Syntax tree for a-4+e.

#### Production

$$E \rightarrow TE'$$

$$E' \rightarrow +TE'$$

$$E' \rightarrow -TE'$$

$$E' \rightarrow \epsilon$$

$$T \rightarrow (E)$$

$$T \rightarrow id$$

$$T \rightarrow num$$

#### Semantic Rule

$$E.\text{node} = E'.\text{Syn}$$

$$E'.\text{inh} = T.\text{node}$$

$$E'.\text{inh} = \text{new Node}('+' , E'.\text{inh}, T.\text{node})$$

$$E'.\text{Syn} = E_1'.\text{Syn}$$

$$E_1'.\text{inh} = \text{new Node}('-', E'.\text{inh}, T.\text{node})$$

$$E'.\text{Syn} = E_1'.\text{Syn}$$

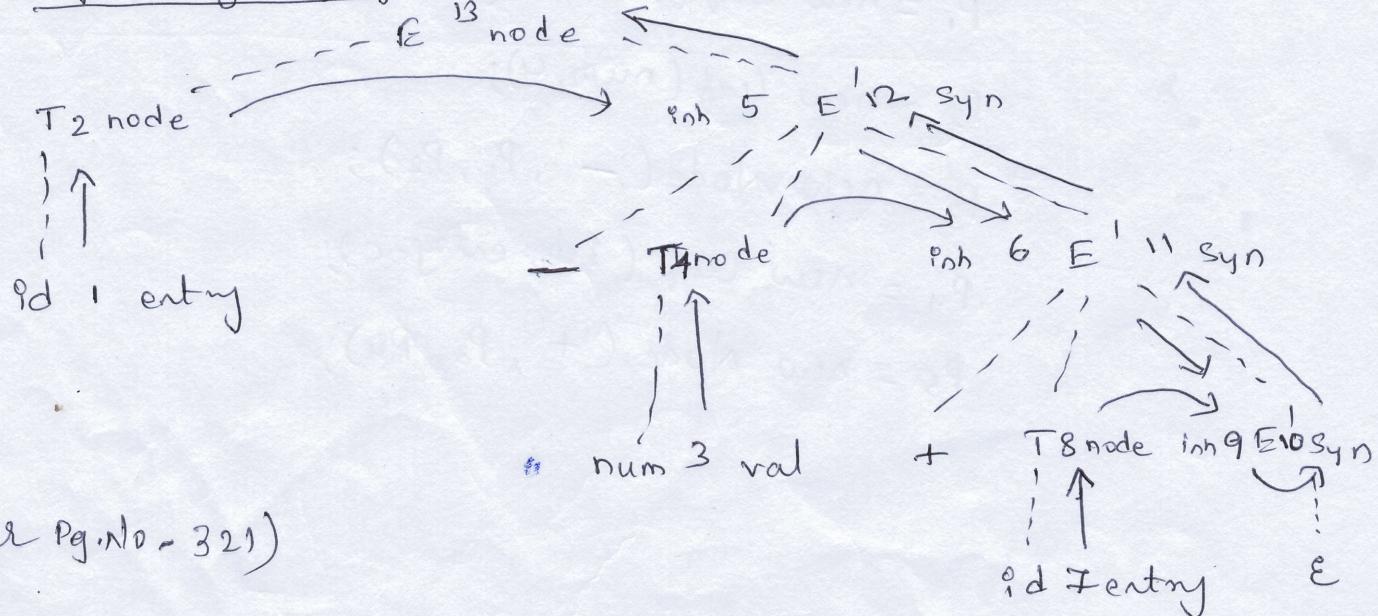
$$E'.\text{Syn} = E'.\text{inh}$$

$$T.\text{node} = E.\text{node}$$

$$T.\text{node} = \text{new leaf}(id, id.\text{entry})$$

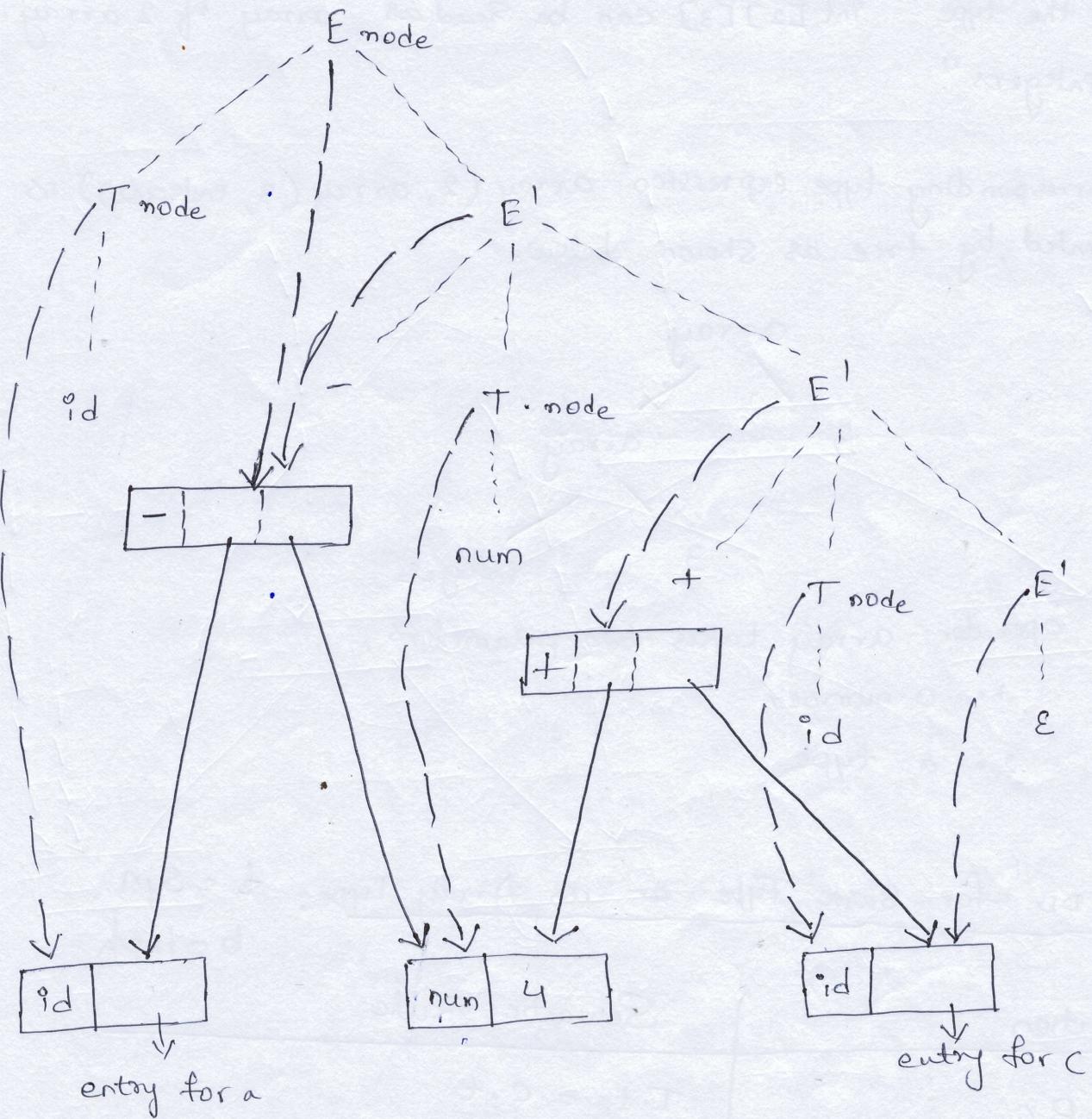
$$T.\text{node} = \text{new leaf}(num, num.\text{val})$$

#### Dependency graph for a-4+e:



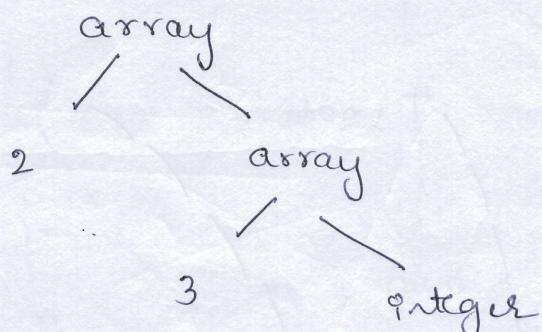
(Refer Pg.No - 321)

Syntax tree for  $a - 4 + c$  (CTOP-down approach)



## The Structure of a Type:

- \* In C, the type `int [2][3]` can be read as "array of 2 array of 3 integers".
- \* The corresponding type expression `array(2, array(3, integer))` is represented by tree as shown below.



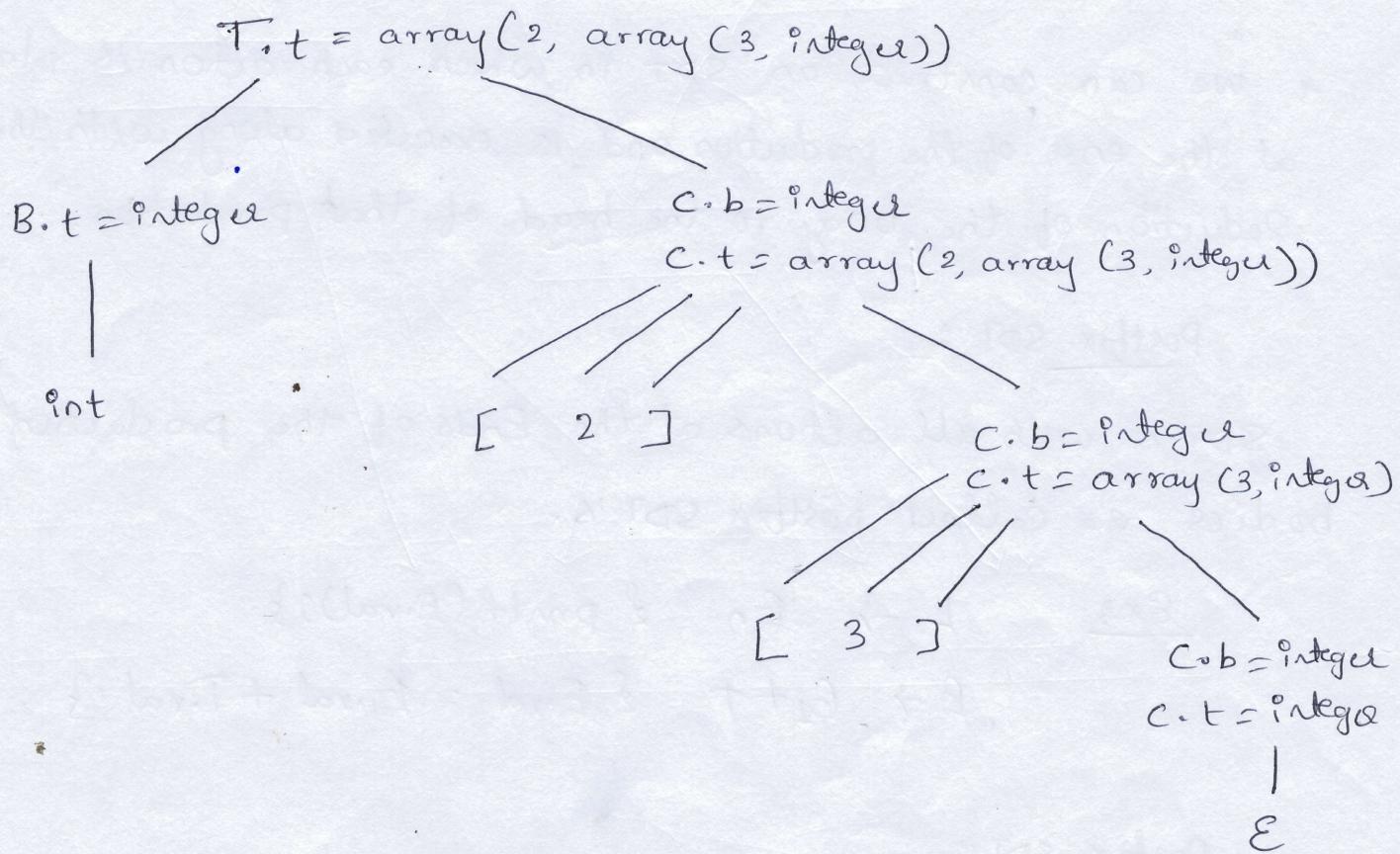
- \* The operator `array` takes two parameters,
  1. a number
  2. a type

SDD for Basic Type or an Array Type: t = Syn  
b = inch

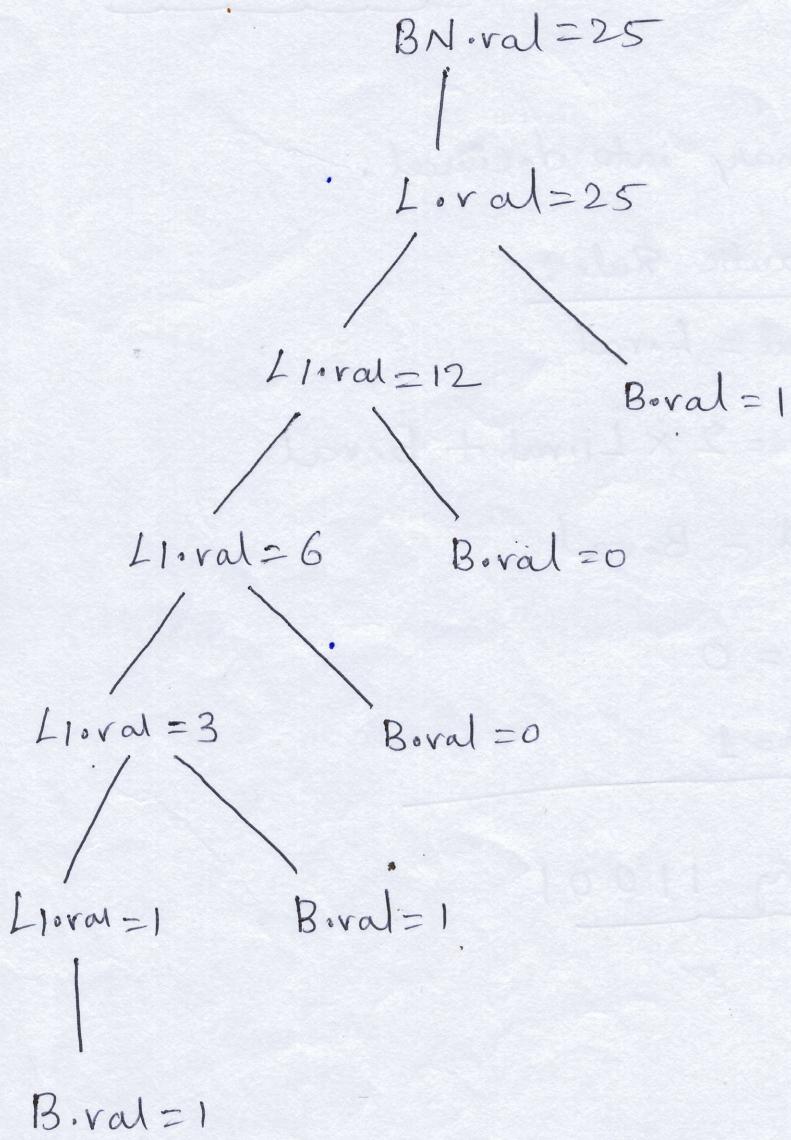
Production	Semantic Rule
$T \rightarrow B.C$	$T.t = C.t$ $C.b = B.t$
$B \rightarrow \text{int}$	$B.t = \text{integer}$
$B \rightarrow \text{float}$	$B.t = \text{float}$
$C \rightarrow [\text{num}] C_1..$	$C.t = \text{array} (\text{num}, C_1.t)$ $C_1.b = C.b$
$C \rightarrow \epsilon$	$C.t = C.b$

- B and T have synthesized attribute t (type) and C has two attributes inherited (b) and synthesized (t).
- In above SPP, non-terminal T generates either a Basic type or Array type.
- Non-terminal B generates one of the basic type int and float.

Annotated parse tree for string int[2][3]



Annotated parse tree for string 11001.



2 <sup>4</sup>	2 <sup>3</sup>	2 <sup>2</sup>	2 <sup>1</sup>	2 <sup>0</sup>
16	8	4	2	1
1	1	0	0	1

$$\text{Result} = 25$$

when we perform reduction  $L \rightarrow L_1 B$   
then we apply Semantic rule

$$L \cdot val = 2 \times L_1 \cdot val + B \cdot val$$

$$L \rightarrow L_1 B \quad L \cdot val = 2 \times L_1 \cdot val + B \cdot val$$

$$L \cdot val = 2 \times 1 + 1$$

$$= 2 + 1$$

$$\underline{\underline{= 3}}$$

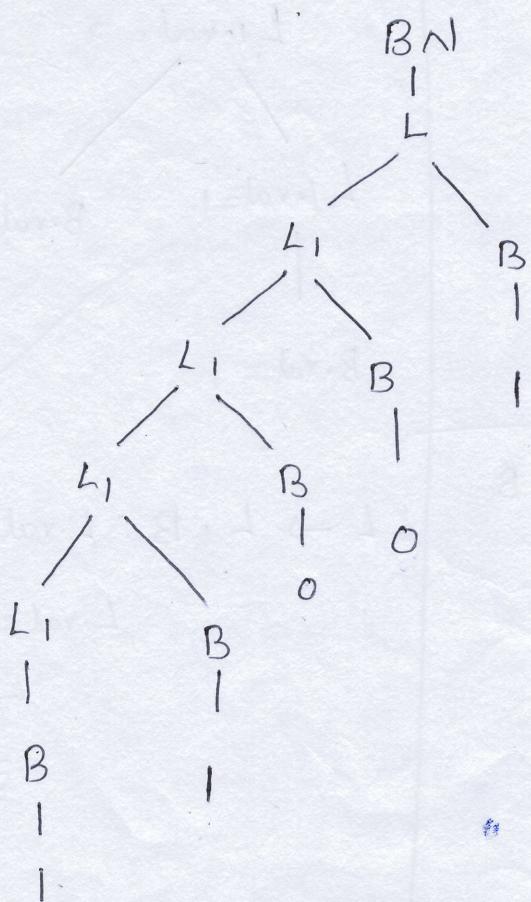
② Write SDD to translate Binary integer number into decimal, construct the parse tree and annotated parse tree for the input string 11001.

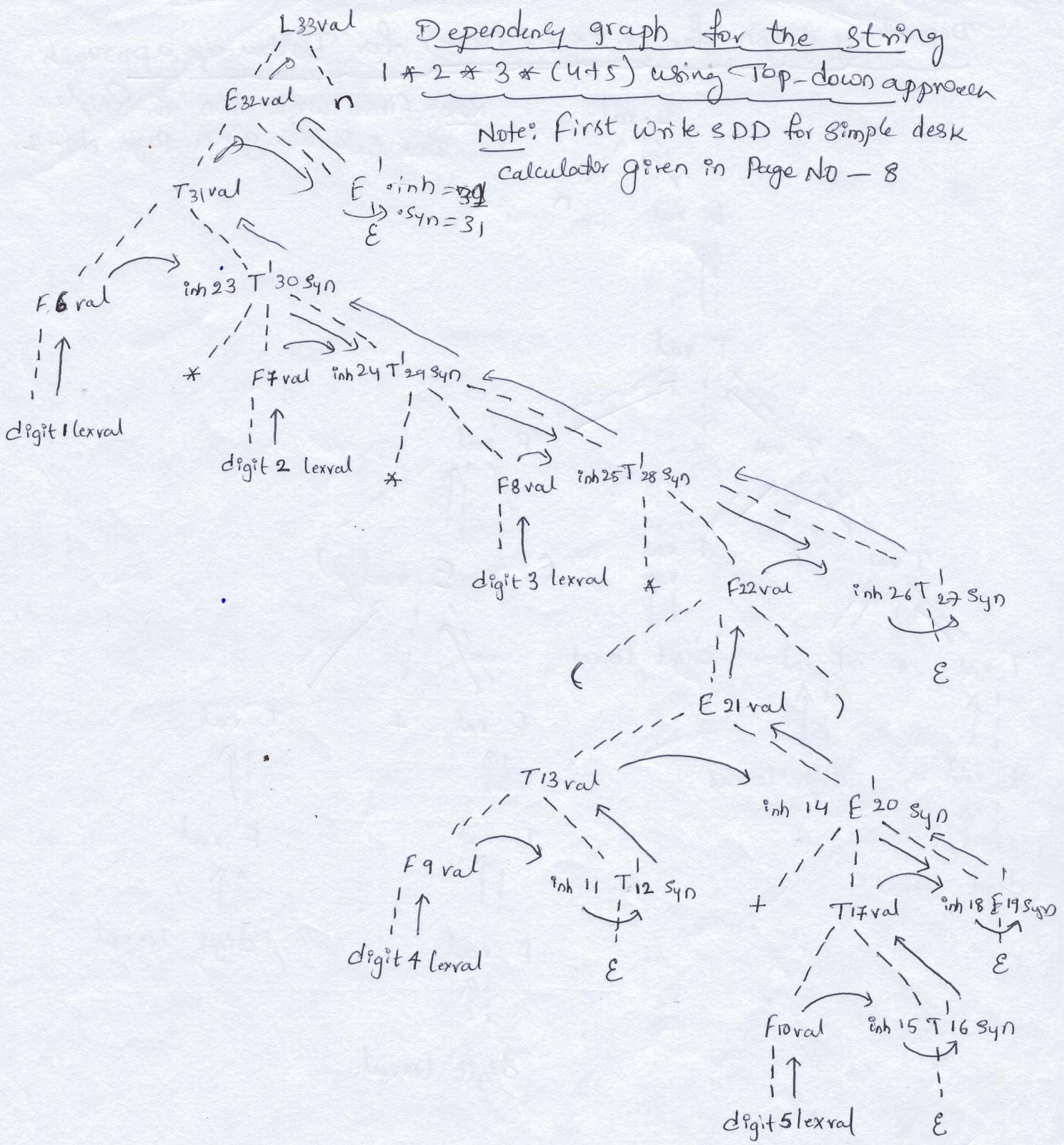
Soln:

SDD to translate Binary into decimal.

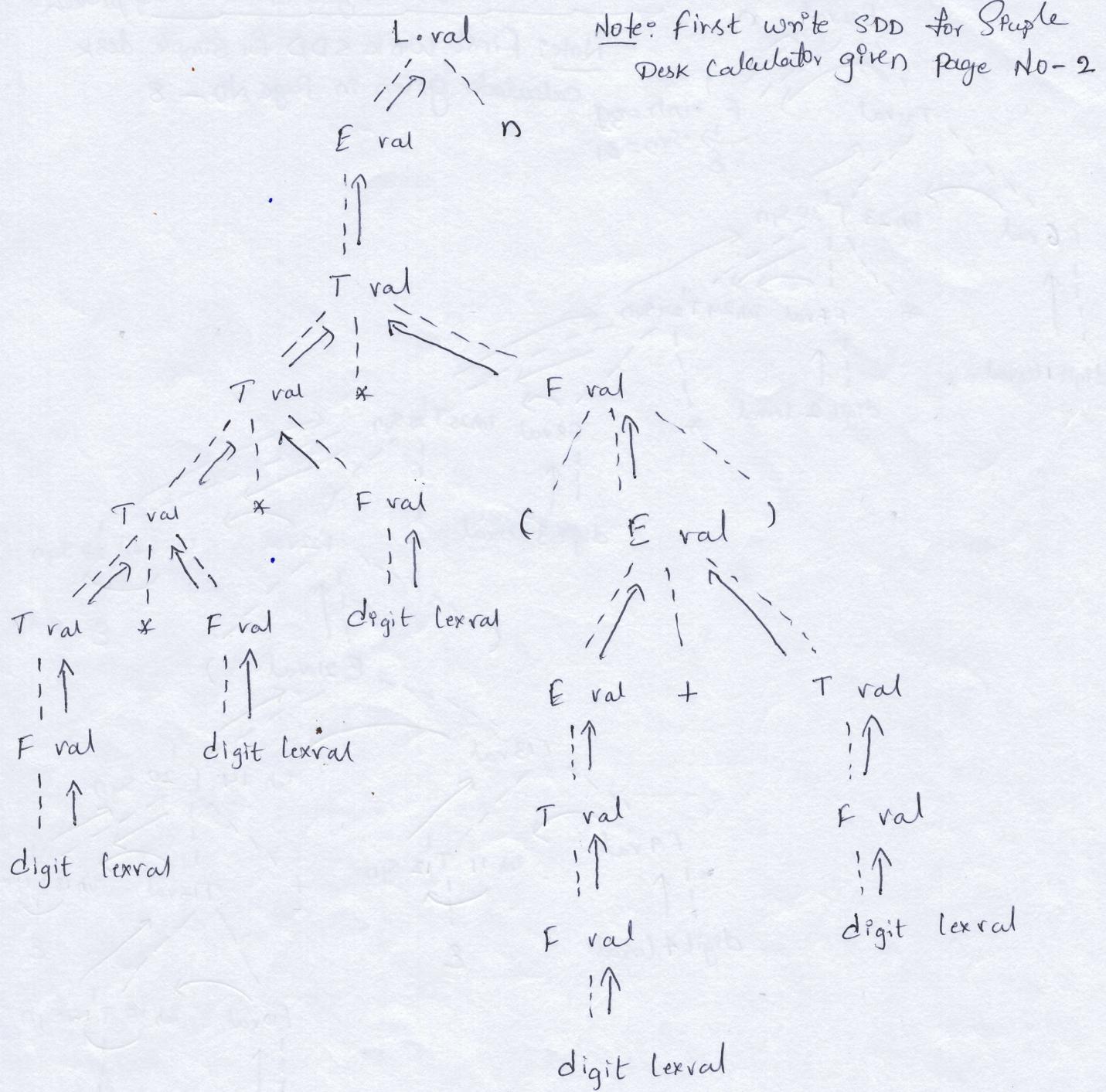
Productions	Semantic Rule :
$BN \rightarrow L$	$BN\cdot val = L\cdot val$
$L \rightarrow L, B$	$L\cdot val = 2 \times L_1\cdot val + B\cdot val$
$L \rightarrow B$	$L\cdot val = B\cdot val$
$B \rightarrow 0$	$B\cdot val = 0$
$B \rightarrow 1$	$B\cdot val = 1$

Parse tree for string 11001





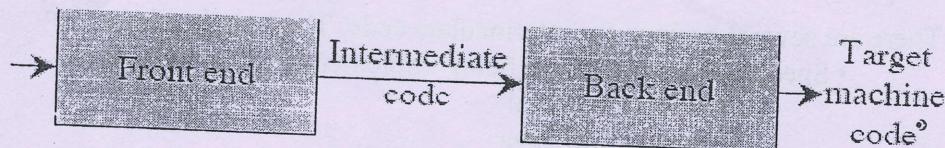
Dependency graph for  $1 * 2 * 3 * (4 + 5)$  for Bottom-up approach



## VTU - Exam Questions

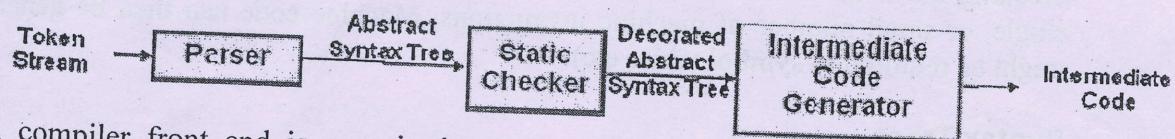
- ① Explain concept of Syntax-Directed definition? (6 Marks)  
Page No-1.
- ② Define Inherited and Synthesized attribute? Give example? (6 Marks)  
Page No-1
- ③ Write SDD that generates either a Basic type or Array type and construct Annotated parse tree for  $\text{int}[2][3]$ . (8 Marks)  
or  
By considering an array type  $\text{int}[3][3]$ . Write SDT with Semantic rule?  
Page No- 22 & 23.  
(Page No-5) & Bottom-up approach.
- ④ Write annotated parse tree for  $3 * 5 + 4 n$  using Top down approach.  
Write semantic rules for each step. (8 Marks)  
and Also write dependency graph?  
Page - 8, 9 & 30
- ⑤ Give the SDD for Simple Desk calculator and draw dependency graph for expression  $1 * 2 * 3 * (4+5)n$ . (10 Marks)  
For Bottom-up approach - Page No- 34  
For Top-down approach - Page No- 33
- ⑥ Explain S-attributed and L-attributed Definition? (6 Marks)  
Page No- 13 & 14
- ⑦ Write the SDD for Simple type declaration and write dependency graph for declaration float id1, id2, id3 (8 Marks)  
or  
Give the SDD to process a Sample variable declaration in C and construct dependency graph for the input float X, Y, Z. (10 Marks)  
Page - 16.
- ⑧ Assuming suitable SDD, construct a Syntax tree for exp  $a - 4 + c$ . (10 Marks)  
Page - 18.
- ⑨ Write SDT to convert Infix-to-Postfix and Infix-to-Prefix and construct parse tree for exp  $2 * 3 + 5$ . (10 Marks)  
Page NO - 25, 26 & 27
- ⑩ Write SDD to translate Binary into decimal, and construct Parse tree and annotated parse tree for string 11001. (8 Marks)  
Page No- 31 & 32
- ⑪ Give a SDT for desktop calculator and Show its parse stack implementation? (8 Marks)  
or  
Explain the parse stack implementation of post fix SDT? (8 Marks)  
Page No- 28 & 29

## Unit 6 : Intermediate Code Generation



In the analysis-synthesis model of a compiler, the front end analyzes a source program and creates an intermediate representation, from which the back end generates target code. This facilitates retargeting: enables attaching a back end for the new machine to an existing front end.

### Logical Structure of a Compiler Front End



A compiler front end is organized as in figure above, where parsing, static checking, and intermediate-code generation are done sequentially; sometimes they can be combined and folded into parsing. All schemes can be implemented by creating a syntax tree and then walking the tree.

### Static Checking

This includes type checking which ensures that operators are applied to compatible operands. It also includes any syntactic checks that remain after parsing like

- flow-of-control checks
  - Ex: Break statement within a loop construct
- Uniqueness checks
  - Labels in case statements
- Name-related checks

### Intermediate Representations

We could translate the source program directly into the target language. However, there are benefits to having an intermediate, machine-independent representation.

- A clear distinction between the machine-independent and machine-dependent parts of the compiler
- Retargeting is facilitated; the implementation of language processors for new machines will require replacing only the back-end
- We could apply machine independent code optimization techniques.

Intermediate representations span the gap between the source and target languages.

- *High Level Representations*
  - closer to the source language
  - easy to generate from an input program
  - code optimizations may not be straightforward.

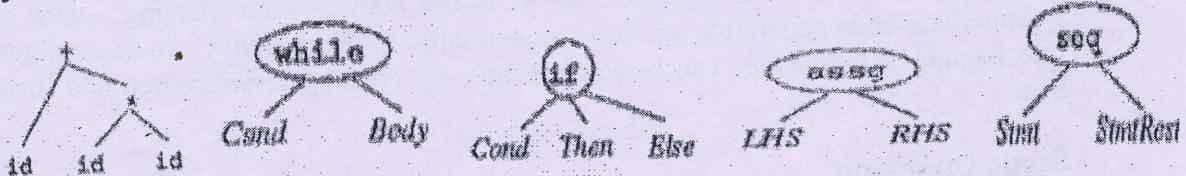
- *Low Level Representations*
  - closer to the target machine
  - Suitable for register allocation and instruction selection.
  - easier for optimizations, final code generation

There are several options for intermediate code. They can be either

- Specific to the language being implemented
  - P-code for Pascal
  - Bytecode for Java
- Language independent:
  - 3-address code

IR can be either an actual language or a group of internal data structures that are shared by the phases of the compiler. C used as intermediate language as it is flexible, compiles into efficient machine code and its compilers are widely available. In all cases, the intermediate code is a linearization of the syntax tree produced during syntax and semantic analysis. It is formed by breaking down the tree structure into sequential instructions, each of which is equivalent to a single, or small number of machine instructions. Machine code can then be generated (access might be required to symbol tables etc).

### Syntax Trees

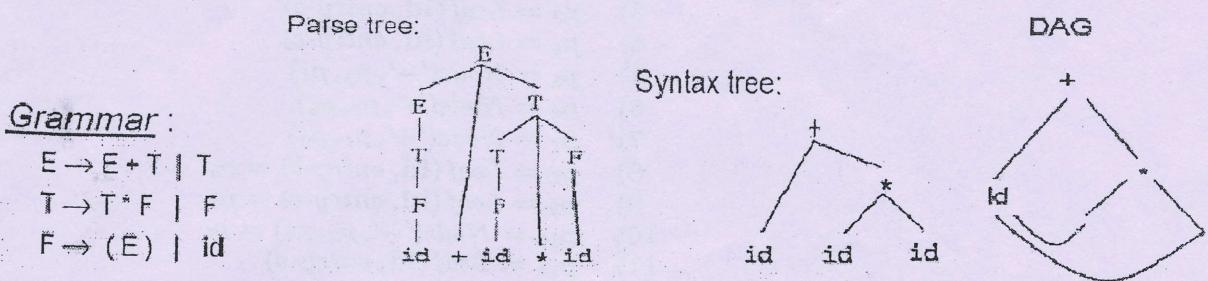


Syntax trees are high level IR. They depict the natural hierarchical structure of the source program. Nodes represent constructs in source program and the children of a node represent meaningful components of the construct. Syntax trees are suited for static type checking.

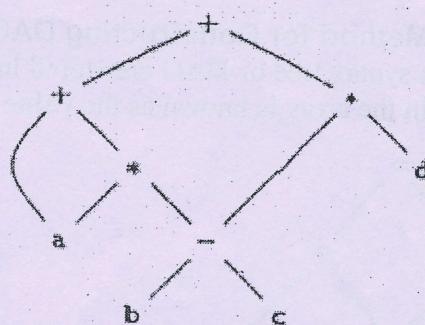
### Variants of Syntax Trees: DAG

- A directed acyclic graph (*DAG*) for an expression identifies the *common sub expressions* (sub expressions that occur more than once) of the expression.
- DAG's can be constructed by using the same techniques that construct syntax trees.
- A DAG has leaves corresponding to atomic operands and interior nodes corresponding to operators. A node N in a DAG has more than one parent if N represents a common sub expression, so a DAG represents expressions concisely.
- It gives clues to compiler about the generating efficient code to evaluate expressions.

**Example 1:** Given the grammar below, for the input string  $\text{id} + \text{id} * \text{id}$ , the parse tree, syntax tree and the DAG are as shown.



**Example 2:** DAG for the expression  $a + a * (b - c) + (b - c) * d$  is shown below.



**Using the SDD to draw syntax tree or DAG for a given expression:-**

- Draw the parse tree
- Perform a post order traversal of the parse tree
- Perform the semantic actions at every node during the traversal
  - Creates a syntax tree if a new node is created each time functions Leaf and Node are called.
  - Constructs a DAG if before creating a new node, these functions check whether an identical node already exists. If yes, the existing node is returned.

SDD to produce Syntax trees or DAG is shown below:

PRODUCTION	SEMANTIC RULES
1) $E \rightarrow E_1 + T$	$E.\text{node} = \text{new Node}( '+', E_1.\text{node}, T.\text{node})$
2) $E \rightarrow E_1 - T$	$E.\text{node} = \text{new Node}( ' - ', E_1.\text{node}, T.\text{node})$
3) $E \rightarrow T$	$E.\text{node} = T.\text{node}$
4) $T \rightarrow ( E )$	$T.\text{node} = E.\text{node}$
5) $T \rightarrow \text{id}$	$T.\text{node} = \text{new Leaf}(\text{id}, \text{id.entry})$
6) $T \rightarrow \text{num}$	$T.\text{node} = \text{new Leaf}(\text{num}, \text{num.val})$

For the expression  $a + a * (b - c) + (b - c) * d$ , steps for constructing the DAG is as below:

- 1)  $p_1 = \text{Leaf}(\text{id}, \text{entry-}a)$
- 2)  $p_2 = \text{Leaf}(\text{id}, \text{entry-}a) = p_1$
- 3)  $p_3 = \text{Leaf}(\text{id}, \text{entry-}b)$
- 4)  $p_4 = \text{Leaf}(\text{id}, \text{entry-}c)$
- 5)  $p_5 = \text{Node}(' - ', p_3, p_4)$
- 6)  $p_6 = \text{Node}('* ', p_1, p_5)$
- 7)  $p_7 = \text{Node}('+ ', p_1, p_6)$
- 8)  $p_8 = \text{Leaf}(\text{id}, \text{entry-}b) = p_3$
- 9)  $p_9 = \text{Leaf}(\text{id}, \text{entry-}c) = p_4$
- 10)  $p_{10} = \text{Node}(' - ', p_3, p_4) = p_5$
- 11)  $p_{11} = \text{Leaf}(\text{id}, \text{entry-}d)$
- 12)  $p_{12} = \text{Node}('* ', p_5, p_{11})$
- 13)  $p_{13} = \text{Node}('+ ', p_7, p_{12})$

### Value-Number Method for Constructing DAGs

Nodes of a syntax tree or DAG are stored in an array of records. The integer index of the record for a node in the array is known as the value number of that node.

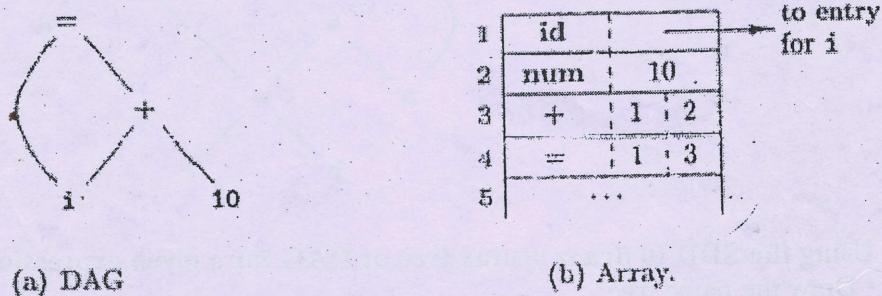


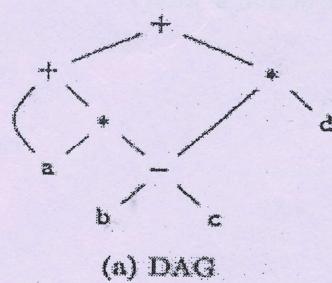
Figure 6.6: Nodes of a DAG for  $i = i + 10$  allocated in an array

The signature of a node is a triple  $\langle op, l, r \rangle$  where  $op$  is the label,  $l$  the value number of its left child, and  $r$  the value number of its right child. The value-number method for constructing the nodes of a DAG uses the signature of a node to check if a node with the same signature already exists in the array. If yes, returns the value number. Otherwise, creates a new node with the given signature.

### Three Address Code

- Three Address Code can range from high- to low-level, depending on the choice of operators. In general, it is a statement containing at most 3 addresses or operands.
- The general form is  $x := y \text{ op } z$ , where “op” is an operator,  $x$  is the result, and  $y$  and  $z$  are operands.  $x, y, z$  are variables, constants, or “temporaries”.
- A three-address instruction consists of at most 3 addresses for each statement. It is a linearized representation of a binary syntax tree.
- Explicit names correspond to interior nodes of the graph.

- E.g. for a looping statement, syntax tree represents components of the statement, whereas three-address code contains labels and jump instructions to represent the flow-of-control as in machine language.



$$\begin{aligned}
 t_1 &= b - c \\
 t_2 &= a * t_1 \\
 t_3 &= a + t_2 \\
 t_4 &= t_1 + d \\
 t_5 &= t_3 + t_4
 \end{aligned}$$

A Three Address Code instruction has at most one operator on the RHS of an instruction; no built-up arithmetic expressions are permitted.

e.g.:  $x + y * z$  can be translated as

$$\begin{aligned}
 t_1 &= y * z \\
 t_2 &= x + t_1
 \end{aligned}$$

where  $t_1$  &  $t_2$  are compiler-generated temporary names.

Since it unravels multi-operator arithmetic expressions and nested control-flow statements, it is useful for target code generation and optimization.

### Addresses and Instructions:

- Three Address Code consists of a sequence of instructions, each instruction may have up to three addresses, prototypically  $t_1 = t_2 \text{ op } t_3$
- Addresses may be one of:
  - A name. Each name is a symbol table index. For convenience, we write the names as the identifier.
  - A constant.
  - A compiler-generated temporary. Each time a temporary address is needed, the compiler generates another name from the stream  $t_1, t_2, t_3$  etc.
    - Temporary names allow for code optimization to easily move instructions
    - At target-code generation time, these names will be allocated to registers or to memory.

### Three Address Code Instructions:



- Symbolic labels will be used by instructions that alter the flow of control. The instruction addresses of labels will be filled in later.

$$L: t_1 = t_2 \text{ op } t_3$$

- Assignment instructions:  $x = y \text{ op } z$ 
  - Includes binary arithmetic and logical operations
- Unary assignments:  $x = \text{op } y$ 
  - Includes unary arithmetic op (-) and logical op (!) and type conversion (integer to floating point number)
- Copy instructions:  $x = y$

- Unconditional jump: goto L
  - L is a symbolic label of an instruction
- Conditional jumps:
  - if  $x$  goto L If  $x$  is true, execute instruction L next
  - ifFalse  $x$  goto L If  $x$  is false, execute instruction L next
- Conditional jumps:
  - if  $x$  relop  $y$  goto L
- Procedure calls. For a procedure call  $p(x_1, \dots, x_n)$ 

```

param  $x_1$ 
...
param  $x_n$ 
call p, n

```
- Function calls :  $y = p(x_1, \dots, x_n)$   $y = \text{call } p, n$ , return  $y$
- Indexed copy instructions:  $x = y[i]$  and  $x[i] = y$ 
  - Left: sets  $x$  to the value in the location  $i$  memory units beyond  $y$
  - Right: sets the contents of the location  $i$  memory units beyond  $x$  to  $y$
- Address and pointer instructions:
  - $x = \&y$  sets the value of  $x$  to be the location (address) of  $y$ .
  - $x = *y$ , presumably  $y$  is a pointer or temporary whose value is a location. The value of  $x$  is set to the contents of that location.
  - $*x = y$  sets the value of the object pointed to by  $x$  to the value of  $y$ .

Example: Given the statement `do i = i+1; while (a[i] < v);`, the TAC can be written as below in two ways, using either symbolic labels or position number of instructions for labels.

L:     $t_1 = i + 1$   
        $i = t_1$   
        $t_2 = i * 8$   
        $t_3 = a[t_2]$   
       if  $t_3 < v$  goto L

(a) Symbolic labels.

100:  $t_1 = i + 1$   
 101:  $i = t_1$   
 102:  $t_2 = i * 8$   
 103:  $t_3 = a[t_2]$   
 104: if  $t_3 < v$  goto 100

(b) Position numbers.

### Three Address Code Representations

*v. imp* Data structures for representation of TAC can be objects or records with fields for operator and operands. Representations include quadruples, triples and indirect triples.

#### 1) Quadruples

- In the quadruple representation, there are four fields for each instruction:  
 $op, arg1, arg2, result$ 
  - Binary operators have the obvious representation
  - Unary operators don't use arg2
  - Operators like param don't use either arg2 or result

- Jumps put the target label into result
- The quadruples in Fig (b) implement the three-address code in (a) for the expression  $a = b * - c + b * - c$

```

t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4
a = t5

```

(a) Three-address code

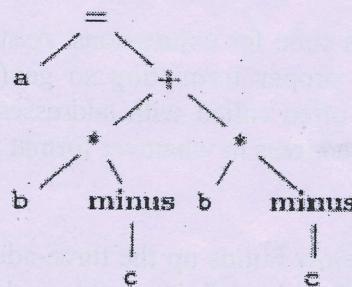
	op	arg <sub>1</sub>	arg <sub>2</sub>	result
0	minus	c		t <sub>1</sub>
1	*	b	t <sub>1</sub>	t <sub>2</sub>
2	minus	c		t <sub>3</sub>
3	*	b	t <sub>3</sub>	t <sub>4</sub>
4	+	t <sub>2</sub>	t <sub>4</sub>	t <sub>5</sub>
5	=	t <sub>5</sub>		a
				...

(b) Quadruples

## 2) Triples

- A triple has only three fields for each instruction: *op*, *arg<sub>1</sub>*, *arg<sub>2</sub>*
- The *result* of an operation  $x \ op \ y$  is referred to by its position.
- Triples are equivalent to signatures of nodes in DAG or syntax trees.
- Triples and DAGs are equivalent representations only for expressions; they are not equivalent for control flow.
- Ternary operations like  $x[i] = y$  requires two entries in the triple structure, similarly for  $x = y[i]$ .
- Moving around an instruction during optimization is a problem

Example: Representations of  $a = b * - c + b * - c$



(a) Syntax tree

	op	arg <sub>1</sub>	arg <sub>2</sub>
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
			...

(b) Triples

### 3) Indirect Triples :-

These consist of a listing of pointers to triples, rather than a listing of the triples themselves. An optimizing compiler can move an instruction by reordering the instruction list, without affecting the triples themselves.

instruction	op	arg <sub>1</sub>	arg <sub>2</sub>
35 (0)	minus	c	
36 (1)	*	b	(0)
37 (2)	minus	c	
38 (3)	*	b	(2)
39 (4)	+	(1)	(3)
40 (5)	=	a	(4)
...			

### Static Single-Assignment Form

Static single-assignment form (SSA) is an intermediate representation that facilitates certain code optimizations. Two distinctive aspects distinguish SSA from three-address code.

- All assignments in SSA are to variables with distinct names; hence *static single assignment*.
- $\Phi$ -FUNCTION

Same variable may be defined in two different control-flow paths. Eg of  $\Phi$  SSA :-

For example,

if ( flag ), x = -1; else x = 1;

y = x \* a;

using  $\Phi$ -function it can be written as

if ( flag ) x<sub>1</sub> = -1; else x<sub>2</sub> = 1;

x<sub>3</sub> =  $\Phi$ (x<sub>1</sub>, x<sub>2</sub>);

y = x<sub>3</sub> \* a;

$$P = a + b$$

$$q = P - c$$

$$P = q * d$$

$$P = e - P$$

$$q = P + q$$

Three address code

$$P_1 = a + b$$

$$q_1 = P_1 - c$$

$$P_2 = q_1 * d$$

$$P_3 = e - P_2$$

$$q_2 = P_3 + q_1$$

- The  $\Phi$ -function returns the value of its argument that corresponds to the control flow path that was taken to get to the assignment statement containing the  $\Phi$ -function.

### Translation of Expressions

The goal is to generate 3-address code for expressions. Assume there is a function gen() that given the pieces needed does the proper formatting so gen(x = y + z) will output the corresponding 3-address code. gen() is often called with addresses rather than lexemes like x. The constructor Temp() produces a new address in whatever format gen needs.

### Operations within Expressions

The syntax-directed definition below builds up the three-address code for an assignment statement *S* using attribute *code* for *S* and attributes *addr* and *code* for an expression *E*. Attributes *S.code* and *E.code* denote the three-address code for *S* and *E*, respectively. Attribute *E.addr* denotes the address that will hold the value of *E*.

## UNIT-8

### CODE GENERATION

- \* Code generation is the final stage of a compiler.
- \* The software that performs code generation is called as the code generator.
- \* The code generator accepts as its input an intermediate code and produces the target code as its output.
- \* It can be pictorially represented as shown below.



- \* The code generator has three primary tasks: instruction selection, register allocation and assignment, and instruction ordering.



#### Issues in the Design of a Code Generator:

- \* The issues that play a significant role in the design of a code generator are:
  - 1) Input to the code generator
  - 2) The target program
  - 3) Instruction selection
  - 4) Register Allocation
  - 5) Evaluation Order.

## (i) Input to the Code generator :-

- \* The input to the code generator is the intermediate representation of the source program produced by the front end, along with the information in the symbol table that is used to determine the run-time addresses of the data objects denoted by the names in the Intermediate representation.
- \* Various types of intermediate representations are:
  - Three-address representations such as quadruples, triples, indirect triples.
  - Virtual Machine representations such as bytecodes and stack-Machine code.
  - Graphical representations such as Syntax trees and DAG's.
  - Linear representations such as postfix notation.

## (ii) The Target Program :-

- \* The target program that the code generator is expected to generate depends upon the target machine.
- \* However, this is also an important issue that significantly affects the design of a code generator.
- \* The most common target-machine architectures are RISC (Reduced instruction set computer), CISC (Complex instruction set computer) and stack based.
- \* A RISC machine typically has many registers, three-address

instructions, simple addressing modes and a relatively simple instruction-set architecture.

- \* A CISC machine has few registers, two-address instructions, a variety of addressing modes, several register classes, variable length instructions and instructions with side effects.
- \* In a stack-based machine, operations are done by pushing operands onto a stack and then performing the operations on the operands at the top of the stack.

### iii) Instruction Selection :-

- \* The code generator must map the IR program into a code sequence that can be executed by the target machine.
- \* The complexity of performing this mapping is determined by factors such as
  - the level of the IR
  - the nature of the instruction-set architecture.
  - the desired quality of the generated code.
- \* If the IR is high level, the code generator may translate each IR statement into a sequence of machine instructions using code templates.
- \* Such statement-by-statement code generation, however, often produces poor code that needs further optimization.
- \* The nature of the instruction set of the target machine has a strong effect on the difficulty of instruction selection.

\* If the target machine does not support each data type in a uniform manner, then each exception to the general rule requires special handling.

\* For example, if the target machine has an "increment" instruction (INC), then the three-address statement  $a = a + 1$  may be implemented more efficiently by the single instruction INC a, rather than by a more obvious sequence that loads a into a register, adds one to the register, and then stores the result back into a:

```
LD R0, a      // R0=a  
ADD R0, R0, #1 // R0=R0+1  
ST a, R0      // a=R0
```

#### iv) Register Allocation :-

- \* A key problem in code generation is deciding what values to hold in what registers.
- \* Registers are the fastest computational unit on the target machine, but we usually do not have enough of them to hold all values.
- \* Values not held in registers need to reside in memory.
- \* The instructions involving register operands are invariably shorter and faster than those involving operands in memory, so efficient utilization of registers is particularly important.

\* The use of registers is often subdivided into two subproblems:

1. Register allocation, during which we select the set of variables that will reside in registers at each point in the program.
2. Register assignment, during which we pick the specific register that a variable will reside in.

Eg: Consider the two three-address code sequences given below.

$$\begin{aligned}t &= a + b \\t &= t * c \\t &= t / d\end{aligned}\quad (a)$$

$$\begin{aligned}t &= a + b \\t &= t + c \\t &= t / d\end{aligned}\quad (b)$$

The shortest assembly-code sequences for (a) and (b) is

L R1, a  
A R1, b  
M R0, c  
D R0, d  
ST R1, t  
(a)

L R0, a  
A R0, b  
A R0, c  
SRDA R0, 32  
D R0, d  
ST R1, t  
(b)

-  $R_i$  stands for register  $i$ .

- SRDA stands for Shift-right-Double Arithmetic and SRDA  $R0, 32$  shifts the dividend into  $R_1$  and clears

$R_0$  so all bits equal its sign bit.

- L, ST, A stand for load, store & add respectively.

## (V) Evaluation Order :-

- \* The order in which computations are performed can affect the efficiency of the target code.
- \* The simplest method is to generate code for 3-address statements in the order in which they have been produced.

## → Target Language :-

Simple Target Machine Model :-

- \* The target language refers to that language in which the code generator has to generate the code.
- \* The target language depends upon the target computer.

\* The target computer is assumed to have  $n$  registers

$R_0, R_1, \dots, R_{n-1}$

\* Different types of instructions are :

### 1) Load Operations :-

- \* The instruction  $LD \ dst, \ addr$  loads the value in location  $addr$  into location  $dst$ .
- \* This instruction denotes the assignment  $dst = addr$ .
- \* The most common form of this instruction is  $LD \ r_1, x$  which loads the value in location  $x$  into register  $r_1$ .
- \* An instruction of the form  $LD \ r_1, r_2$  is a register-to-register copy in which the contents of register  $r_2$  are copied into register  $r_1$ .

## 2) Store Operations :-

\* The instruction  $ST \alpha, a$  stores the value in register  $a$  into location  $\alpha$ .

\* This instruction denotes the assignment  $\alpha = a$ .

## 3) Computation instructions :-

\* It is of the form  $OP \ dst, \ src_1, \ src_2$ .

where,  $OP$  is a operator like ADD or SUB

$dst, \ src_1, \ src_2$  are locations, not necessarily distinct

\* The effect of this machine instruction is to apply the operation represented by  $OP$  to the values in locations  $src_1$  and  $src_2$  and place the result of this operation in location  $dst$ .

\* Eg:-  $SUB \ a_1, a_2, a_3$  computes  $a_1 = a_2 - a_3$ .

\* Any value formerly stored in  $a_1$  is lost, but if  $a_1$  is  $a_2$  or  $a_3$ , the old value is read first.

\* Unary Operators that take only one Operand do not have a  $src_2$ .

## 4) Unconditional Jumps :-

\* The instruction  $BR \ L$  Causes control to branch to the machine instruction with the label  $L$ . ( $BR$  stands for branch)

## 5) Conditional Jumps :-

\* It is of the form  $Cond \ a, L$ , where  $a$  is a register,  $L$  is a label and cond stands for any of the common tests on values in the register  $a$ .

Eg:-  $BLTZ \ a, L$  causes a jump to a label  $L$  if the

value in register  $a$  is less than zero, and allows control to pass to the next machine instruction if not.

\* The target machine has a variety of addressing modes as shown below:

- 1) In instructions, a location can be a variable name  $x$  referring to the memory location that is reserved for  $x$ . (i.e l-value of  $x$ ).
- 2) A location can also be an indexed address of the form  $a(r)$ , where  $a$  is a variable and  $r$  is a register.
  - The memory location denoted by  $a(r)$  is computed by taking the l-value of  $a$  and adding it to the value in register  $r$ .
  - eg:  $LD R1, a(R2)$  will set  $R1 = \text{contents}(a + \text{contents}(R2))$  where,  $\text{contents}(x)$  denotes the contents of the register  $x$ .
  - This addressing mode is useful for accessing arrays, where  $a$  is the base address of the array (i.e address of the first element) and  $r$  holds the number of bytes past that address we wish to reach one of the elements of array  $a$ .

- 3) A memory location can be an integer indexed by a register.

eg:  $LD R1, 100(R2)$  will set  $R1 = \text{contents}(100 + \text{contents}(R2))$

i.e., of loading into R1 the value in the memory location obtained by adding 100 to the contents of register R2.

4) we also allow two indirect addressing modes:

\*R means the memory location found in the location represented by the contents of register R and \*100(R) means the memory location found in the location obtained by adding 100 to the contents of R.

Eg.: LD R1, \*100(R2) will set R1 = contents(contents(100 + contents(R2))), i.e., of loading into R1 the value in the memory location stored in the memory location obtained by adding 100 to the contents of register R2.

5) Immediate Constant addressing mode.

\* The constant is prefixed by #.

\* The instruction LD R1, #100 loads the integer 100 into register R1.

\* ADD R1, R1, #100 adds the integer 100 into register R1.

- comments at the end of instructions are preceded by //.

[Generate machine code for the following three-address instruction]

Eg.: 1) The three-address statement  $x=y-z$  can be

implemented by the machine instructions:

LD R1, y                          // R1 = y

LD R2, z                          // R2 = z

SUB R1, R1, R2                // R1 = R1 - R2

ST x, R1                        // x = R1

Eg 2:- Suppose  $a$  is an array whose elements are 8-byte value.

Also assume elements of  $a$  are indexed starting at 0.

We may execute the three - address instruction  $b = a[i]$

by the machine instructions:

LD R1, i //  $R1 = i$

MUL R1, R1, 8 //  $R1 = R1 * 8$  (computes  $8^i$ )

LD R2, a(R1) //  $R2 = \text{contents}(a + \text{contents}(R1))$

(Places in register R2 the value in the  $i^{\text{th}}$  element of  $a$ ).

ST b, R2

Eg 3:- The assignment into the array  $a$  represented by three - address instruction  $a[j] = c$  is implemented by:

LD R1, c //  $R1 = c$

LD R2, j //  $R2 = j$

MUL R2, R2, 8 //  $R2 = R2 * 8$

ST a(R2), R1 //  $\text{contents}(a + \text{contents}(R2)) = R1$

Eg 4:- To implement a simple pointer indirection, such as the three - address statement  $x = *p$ , we can use machine instructions like:

LD R1, P //  $R1 = p$

LD R2, O(R1) //  $R2 = \text{contents}(0 + \text{contents}(R1))$

ST x, R2 //  $x = R2$

Eg 5:- The assignment through a pointer  $*p = y$  is similarly implemented in machine code by:

LD R1, P //  $R1 = p$

LD R2, y //  $R2 = y$

ST O(R1), R2 //  $\text{contents}(0 + \text{contents}(R1)) = R2$

Eg 6:- Consider a conditional-jump three-address instruction

if  $x < y$  goto L

The Machine Code is :

LD R1, x //  $R1 = x$

LD R2, y //  $R2 = y$

SUB R1, R1, R2 //  $R1 = R1 - R2$

BLTZ R1, M // if  $R1 < 0$ . jump to M.

Here, M is the label that represents the first machine instruction generated from the three-address instruction that has label L.

Eg 7:-  $x = 1$

Machine code is , LD R1, #1  
ST x, R1

Eg 8:-  $x = a$

Machine code is , LD R1, a  
ST x, R1

Eg 9:-  $x = a + 1$

Machine code is , LD R1, a  
ADD R1, R1, #1  
ST x, R1

Eg 10:-  $x = a + b$

Machine code is , LD R1, a  
LD R2, b  
ADD R1, R1, R2  
ST x, R1 .

Eg 11 :-

$$\begin{aligned}x &= b * c \\y &= a + x\end{aligned}$$

Machine code is ,

LD R1, b  
LD R2, c  
MUL R1, R1, R2  
ST X, R1  
LD R3, a  
LD R4, X  
ADD R3, R3, R4  
ST Y, R3.

Program and Instruction Costs :-

How are Program and instruction costs assessed? Explain

- \* A program is a collection of instructions. Each instruction has a certain cost associated with it.
- \* For simplicity, we take the cost of an instruction to be one plus the costs associated with the addressing modes of the operands.
- \* This cost corresponds to the length in words of the instruction.
- \* Addressing modes involving registers have zero additional cost, while those involving a memory location or constant in them have an additional cost of one, because such operands have to be stored in the words following the instruction.

Eg:-

- 1) The instruction LD R0, R1 copies the contents of register

into register R0.

- This instruction has a cost of one because no additional memory words are required.

2) The instruction LD R0, M loads the contents of memory location M into register R0.

- The cost is two since the address of memory location M is in the word following the instruction.

3) The instruction LD RI, \*100(R2) loads into register RI the value given by contents(contents(100 + contents(R2))).

- The cost is two because the constant 100 is stored in the word following the instruction.

\* The cost of a program is equal to the sum of costs of the instructions present in the program.

→ Addresses in the Target Code :-

[Explain how addresses can be generated in the target code for all call and return instructions ?

OR

Explain the static allocation and the stack allocation of activation records in the target code.]

\* logical address space was partitioned into four code & data areas : they are :

1) A statically determined area code that holds the executable target code. The size of the target code can be determined at compile time.