Module - 5

SDD – Inherited attributes – Synthesized attributes – Evaluating SDD – Application of SDT

Define SDD:- SYNTAX DIRECTED DEFINITION

SDD is a CFG together with attributes and rules. Attributes are associated with grammar symbols and rules are associated with productions.

Eg:- If X is a symbol and a is one of its attributes, then X.a denote the value of a particular parse tree node.

Two kinds of attributes

(i) Synthesized attribute

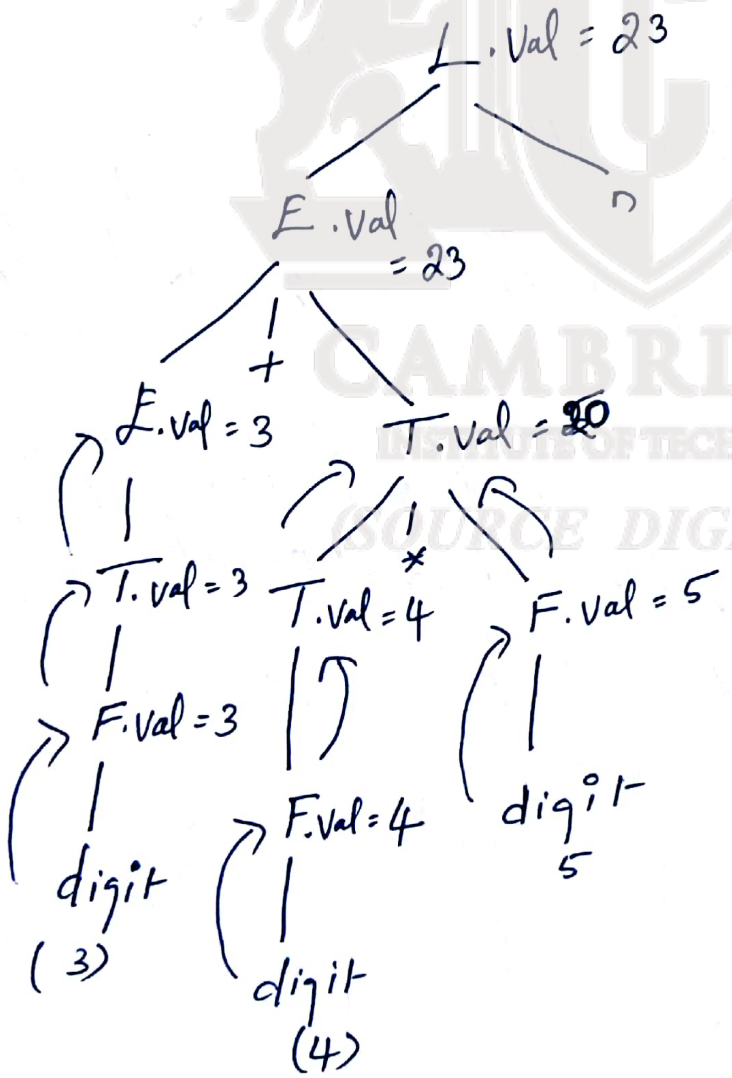
(ii) Inherited attribute.

Synthesized attribute at node N – defined only in terms of attribute values at the children of N and at N itself.

Inherited attribute at node N – defined only in terms of attribute values at N's parent, N itself and N's siblings.

SDD of a Simple desk Calculator. 3+4*5n

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



Val - synthesized attribute

1. Write annotated parse tree for the following
   (i) (4+3)*(5+5)n (ii) 1*2*3*(4+5)n
2. Give SDD for simple desk Calculator.

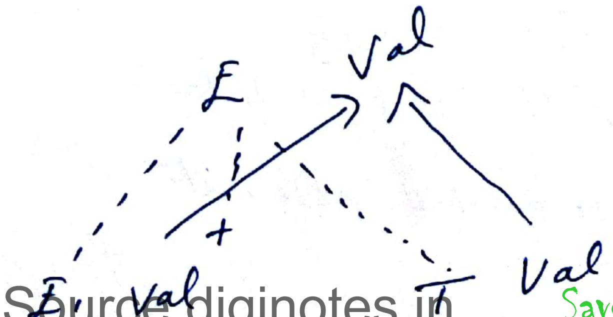
Dependency Graphs:-

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

Dependency graphs - Useful tool for determining an evaluation order for the attribute instances in a given parse tree.

Annotated parse trees show the values of attributes. A dependency graph helps us determine how those values are computed.

Eg:- $E \rightarrow E_1 + T$ $\qquad E.val = E_1.val + T.val$

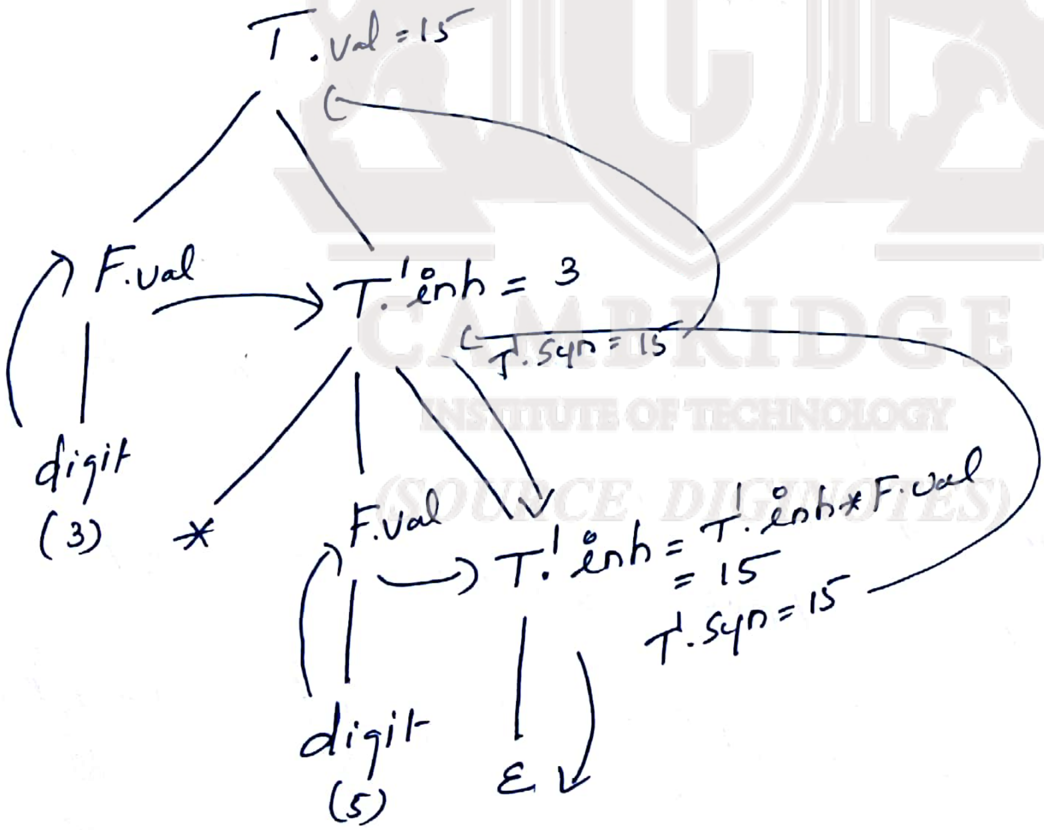


Example for Inherited attributes:-

SDD for the following Grammar

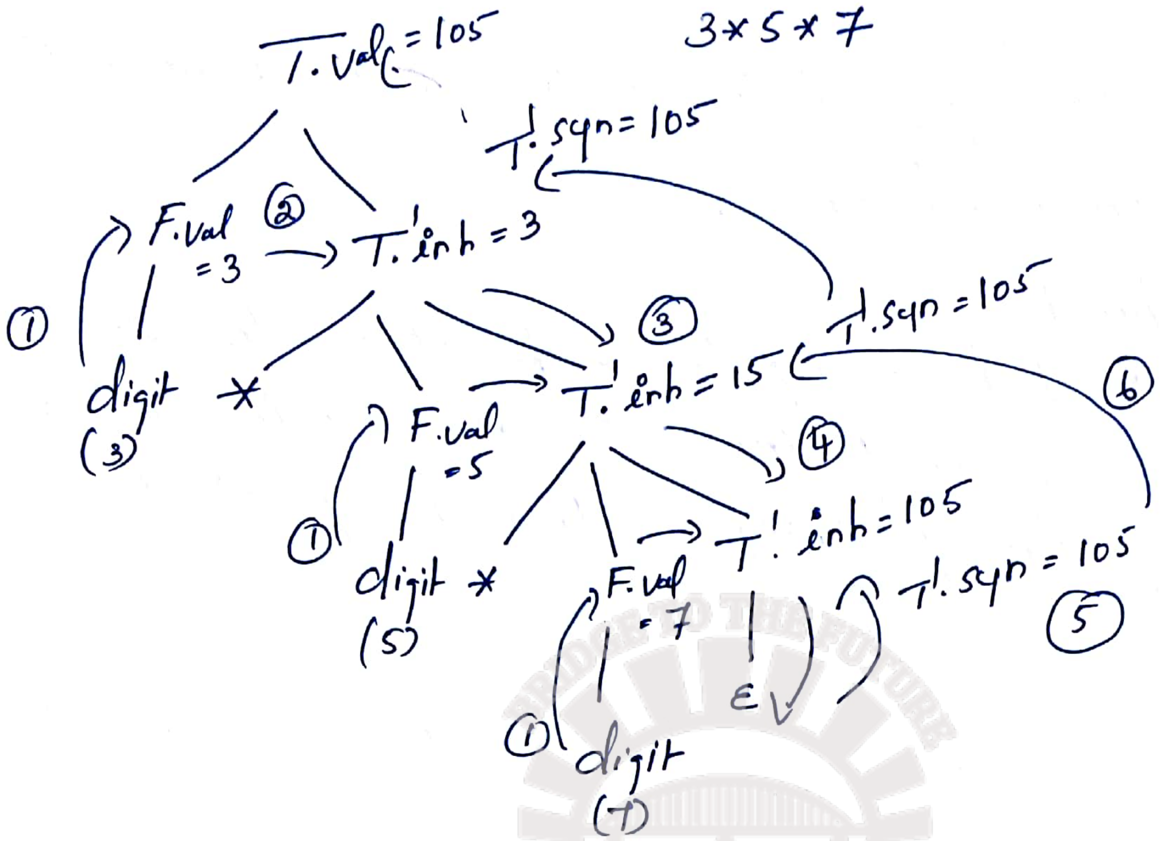
Production	Semantic Rules
$T \rightarrow FT'$	$T.val = T'.syn$ — (5) $T'.inh = F.val$ — (2)
$T' \rightarrow *FT_1'$	$T_1'.inh = T'.inh * F.val$ — (3) $T'.syn = T_1'.syn$
$T' \rightarrow \varepsilon$	$T'.syn = T'.inh$ — (4)
$F \rightarrow digit$	$F.val = digit.lexval$. (1)

(i) $3 * 5$ (ii) $3 * 5 * 7$



Syn, Val - Synthesized attribute

inh - inherited attribute.



S-attributed definition:-

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

S-attributed definitions can be implemented during bottom up parsing and a bottom up parse corresponds to postorder traversal.

L-attributed definition:-

Idea behind this is - between attributes associated with a production body, dependency graph edges can go from left to right but not from left to right. Hence the name L-attributed.

More precisely, each attribute must be either

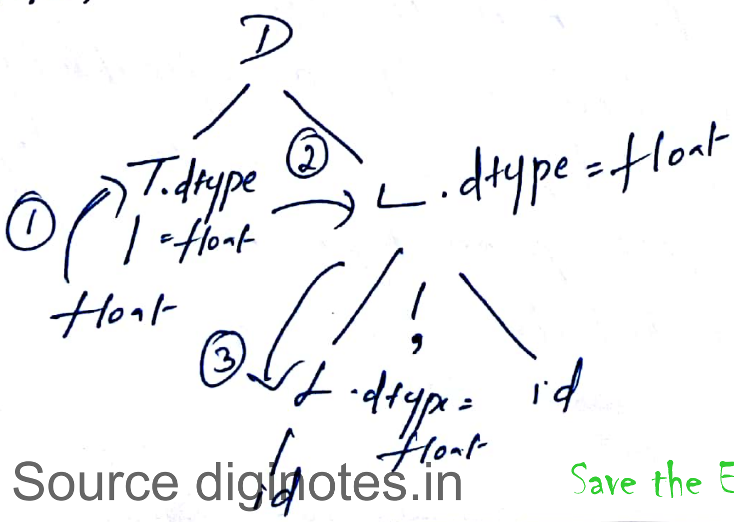
1. Synthesized or
2. Inherited
3. Inherited or Synthesized attributes, in such a way that there are no cycles in a dependency graph formed by attributes.

Eg: 3

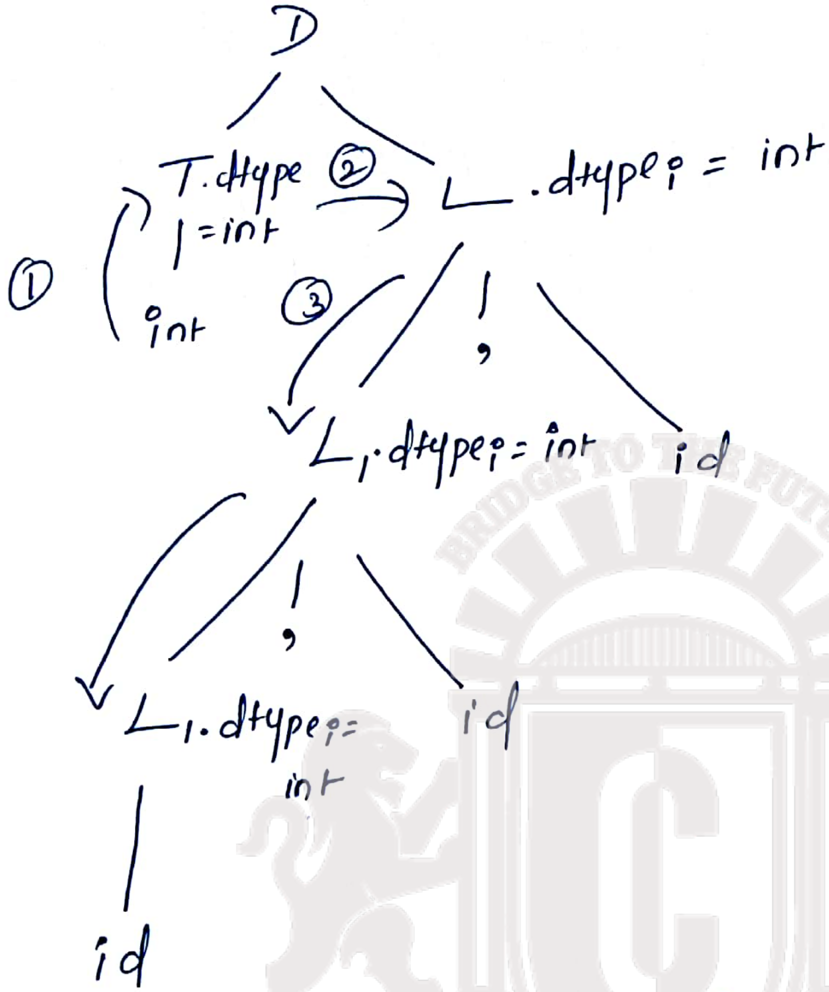
SDD for Declarative Stmts: -

$D \rightarrow T\ L$	$L.dtype_i = T.dtype$	— ②
$T \rightarrow int$	$T.dtype = int$	①
$T \rightarrow float$	$T.dtype = float$	①
$L \rightarrow L_1, id$	$L_1.dtype_i = L.dtype_i$	— ③
	$addType(id.entry, L_1.dtype_i)$	— ④
$L \rightarrow id$	$addType(id.entry, L_1.dtype_i)$	— ⑤

float id_1, id_2;



int id1, id2, id3;

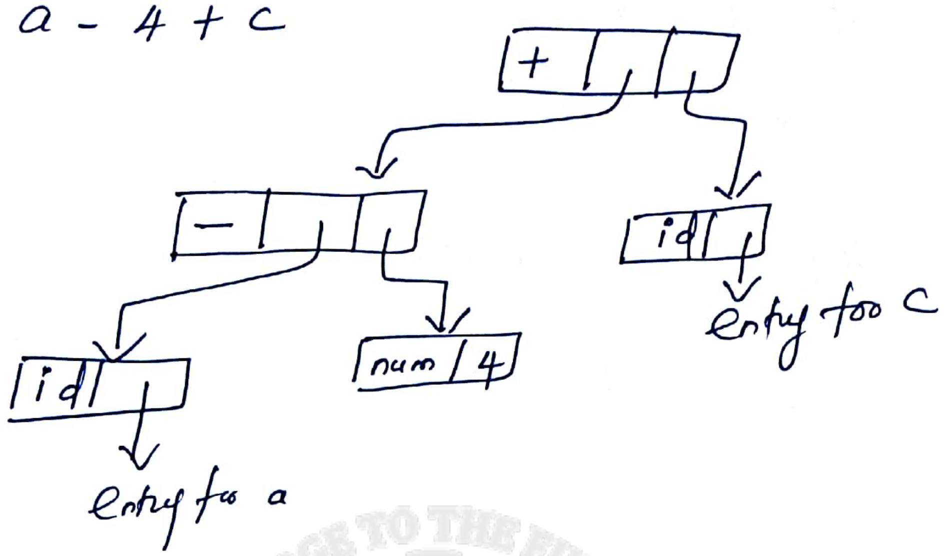


Applns. of SDT — Construction of Syntax tree

Syntax tree → intermediate form / code.

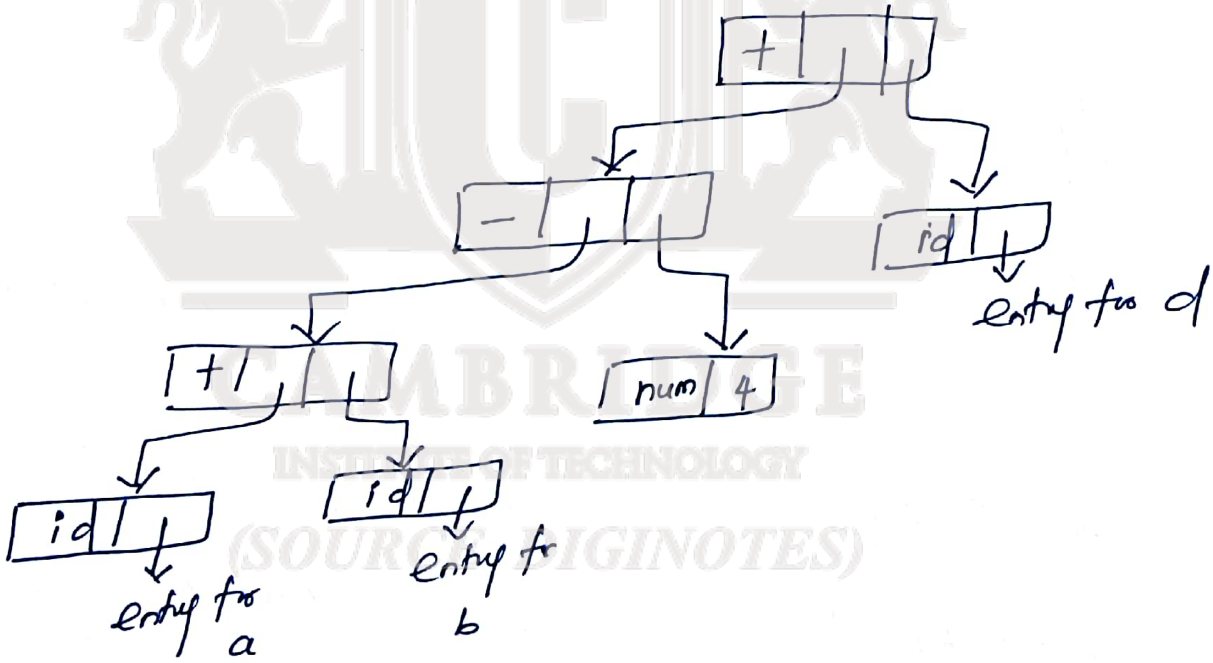
$E \rightarrow E_1 + T$	E.node = new Node('+', E1.node, T.node)
$E \rightarrow E_1 - T$	E.node = new Node('−', E1.node, T.node)
$E \rightarrow T$	E.node = T.node
$T \rightarrow (E)$	T.node = E.node
$T \rightarrow id$	T.node = new leaf(id, id.entry)
$T - num$	T.node = new Leaf(num, num.val)

Eg:- a - 4 + c



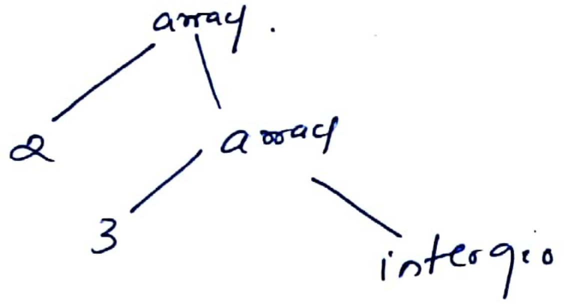
Eg:- 2:-

a + b - 4 + d

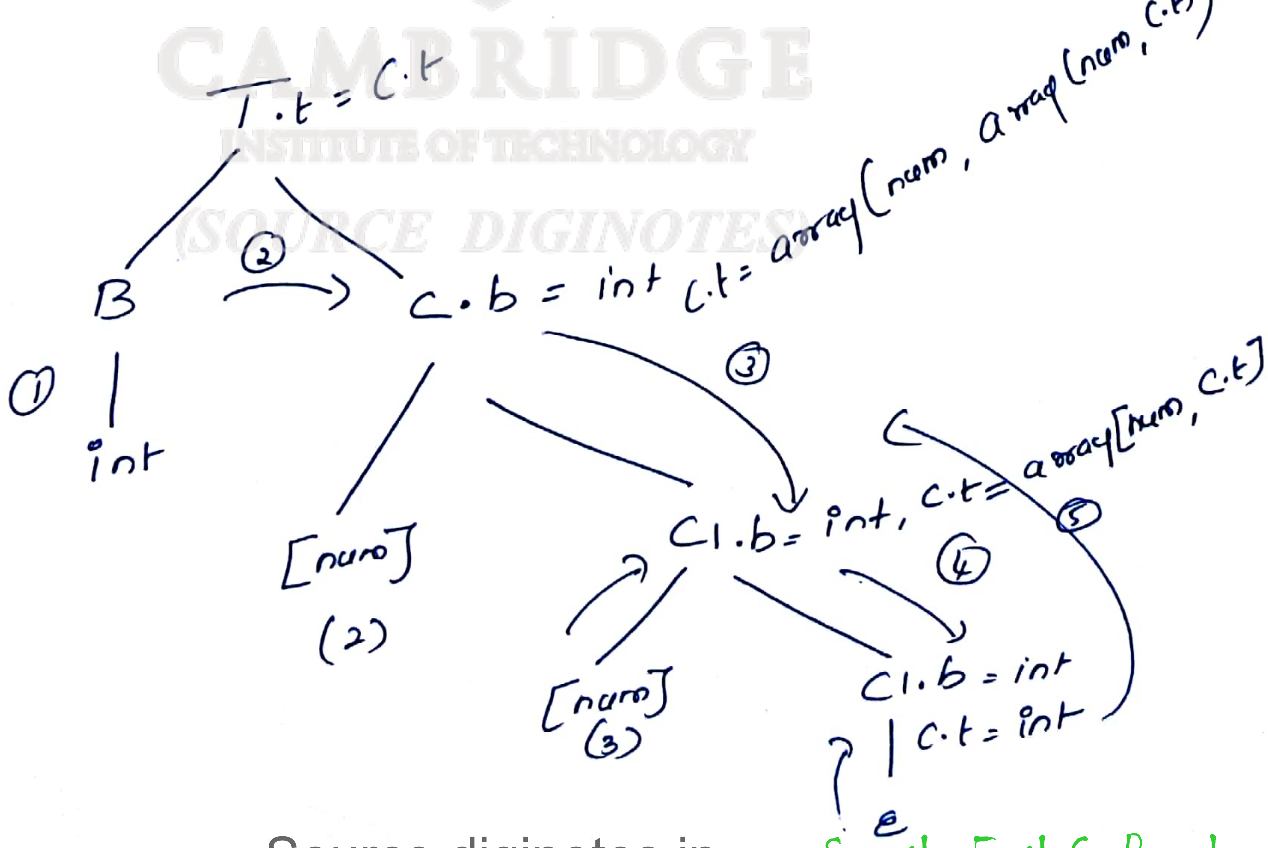


SDD for array datatype:-

int a[2][3]



Production	Semantic rules	
$T \rightarrow B\ C$	$T.t = C.t$ $C.b = B.t$	(2)
$B \rightarrow int$	$B.t = int$	(1)
$B \rightarrow float$	$B.t = float$	(1)
$C \rightarrow [num]\ C_1$	$C.t = array(num, C_1.t)$ (5) $C_1.b = C.b$ (3)	
$C_1 \rightarrow \varepsilon$	$C.t = C.b$	(4)



Variants of Syntax trees:-

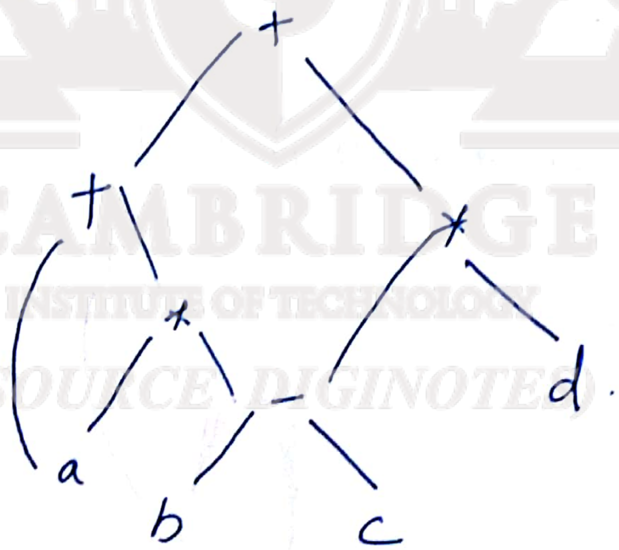
→ DAG is a Variant of Syntax tree.

→ A DAG for an expression identifies the Common Subexpression.

→ Like Syntax trees, DAG has leaves Corresponding to atomic Operands and interior nodes Corresponding to Operators.

→ Difference is that a node N in a DAG has more than one parent if N (Node) represents a Common Subexpression.

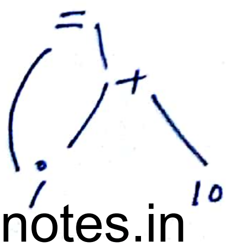
$a + a * (b-c) + (b-c) * d.$



How to store DAG's?

Value Number Method for Constructing DAG's.

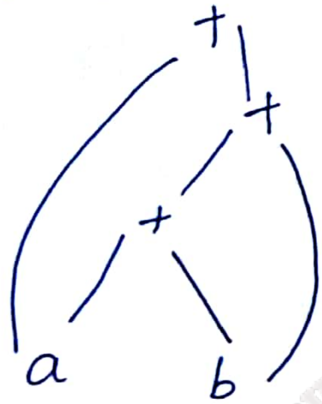
Eg: $i = i + 10$



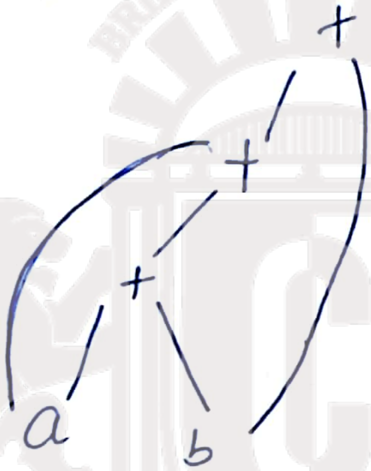
(1)	id		→ entry i
(2)	num	10	
(3)	+	(1)	(2)
(4)		1	3

Construct DAG for following

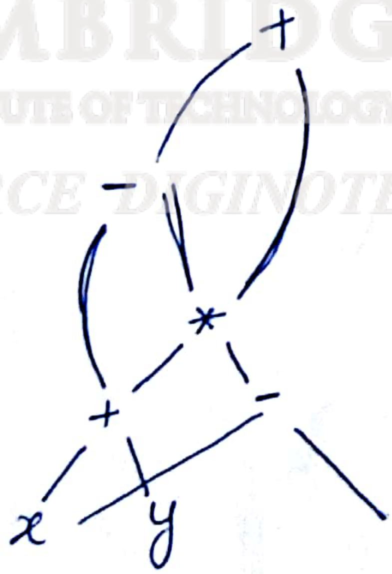
$a+b+(a+b)$



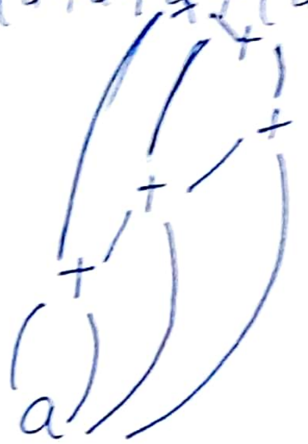
$a+b+a+b$



$((x+y)-(\underbrace{(x+y)*(x-y)}_{1}))+((x+y)*(x-y))$



$a + a + (a + a + a + (a + a + a + a))$



Three address Code (3AC) :-

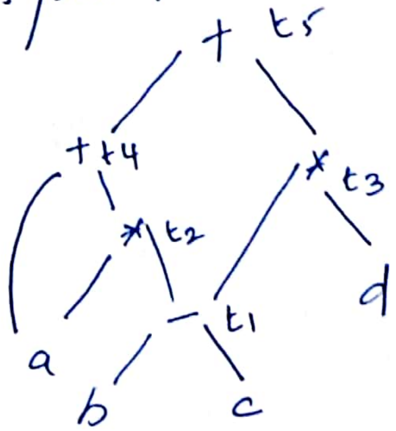
In 3AC, there is atmost one operator on the right side of an instruction.

Eg:- $x + y * z$

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

where t_1 and t_2 are compiler generated temp names.

3AC is a linearised representation of a syntax tree or a DAG in which explicit names correspond to interior nodes of the graph.



$t_1 = b - c$

$t_2 = a * t_1$

$t_3 = t_1 * d$

$t_4 = a + t_2$

$t_5 = t_3 + t_4$

In 3AC, an address can be any one of the following.

1. A name – Source prog. names can appear in 3AC
2. A Constant – Different types of Constants
3. A Compiler generated temporary [ie t_1 t_2 .. t_n] – Compiler Creates a distinct name each time a temporary is needed.

List of Common 3AC forms:-

1. Assignment instruction of form	$\boxed{x = y \text{ op } z}$ op is binary, arithmetic or logical opera
2. Assignment instructions of form	$\boxed{x = \text{op } y}$ op → unary operator
3. Copy instructions.	$x = y$
4. Unconditional jump	goto L
5. Conditional jump	1. if x goto L 2. iffalse x goto L 3. If x relop y goto L
6. Procedure calls & Returns. for p(x_1, x_2 .. x_n)	Param x_1 Param x_2 : Param x_n Call p, n

7. Indexed copy instruction of form — $x = y[i]$, $x[i] = y$

8. Address and pointer assignments — $x = \&y$, $x = *y$, $*x = y$.

Eg:-

do $i = i+1$; while $(a[i] < v)$;

3AC:-

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

Representation of 3AC

1. Quadruples
2. Triples
3. Indirect triples.

Quadruple

OP	arg1	arg2	Result

→ Has four fields which we call op, arg1, arg2 and result.

→ Instr with unary operators donot

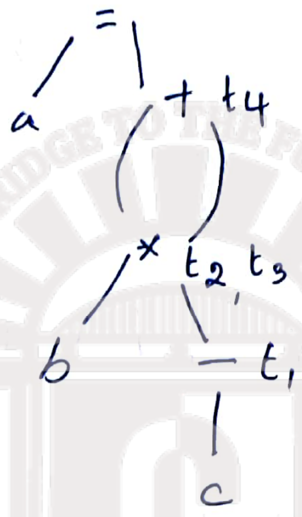
→ Operators like param use neither arg2 nor result.

→ Condnl and Uncondnl. jumps put the target label in result.

→ Copy instrn. '=' is operator.

Eg:- $a = b * -c + b * -c$

DAG:-



t_1 = minus c

$t_2 = b * t_1$

$t_3 = t_2$

$t_4 = t_2 + t_3$

$a = t_4$

	op	arg1	arg2	result
0	minus	c		t_1
1	*	b	t_1	t_3
2	=	t_2		t_3
3	+	t_2	t_3	t_4
4	=	t_4		a