→ Syntax Directed Translation combines the both the Grammar and Semantic Rules.

SDT = CGF + Semantic Rule.

In SDT every non-terminals can get zero(0) or more attributes.

In Semantic Rule attributes may be string or numbers or memory locatn or etc. The semantic actions are enclosed within curly braces.

The most general approach to syntax Directed Translation is to construct a parse tree or a Syntax Tree, and then to compute the values of attributes at the nodes of the tree by visiting the nodes of a tree.

→ Syntax Directed Definitions (SDD).

SDD = CFG + Attributes + Rules

① Synthesized
② Inherited.

The Synthesized attribute for a Non-terminal 'A' at a parse tree node N is defined by a semantic Rule associated with the production at N.

A synthesized attribute at Node N is defined only in terms of attribute values at the children of N and at N itself.

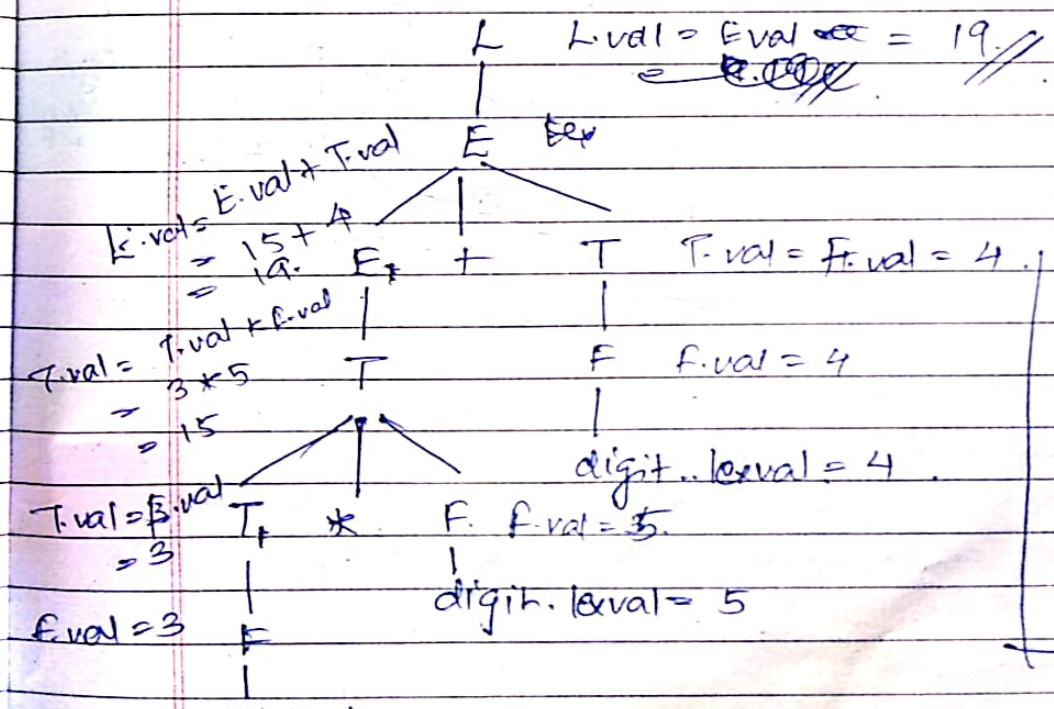
Date / /20

An Inherited attribute for a Non Terminal B at a parse tree node N is defined by a Semantic Rule associated with the production at the parent of N.

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

Production	Semantic Rule
$L \rightarrow E\$$	{ L.val = E.val }
$E \rightarrow E_1 + T$	{ E.val = E_1.val + T.val }
$E \rightarrow T$	{ E.val = T.val }
$T \rightarrow T_1 * F$	{ T.val = T_1.val * F.val }
$T \rightarrow F$	{ T.val = F.val }
$F \rightarrow (E)$	{ F.val = E.val }
$F \rightarrow digit$	{ F.val = digit.lexval }

3 * 5 + 4



annotated Parse Tree.

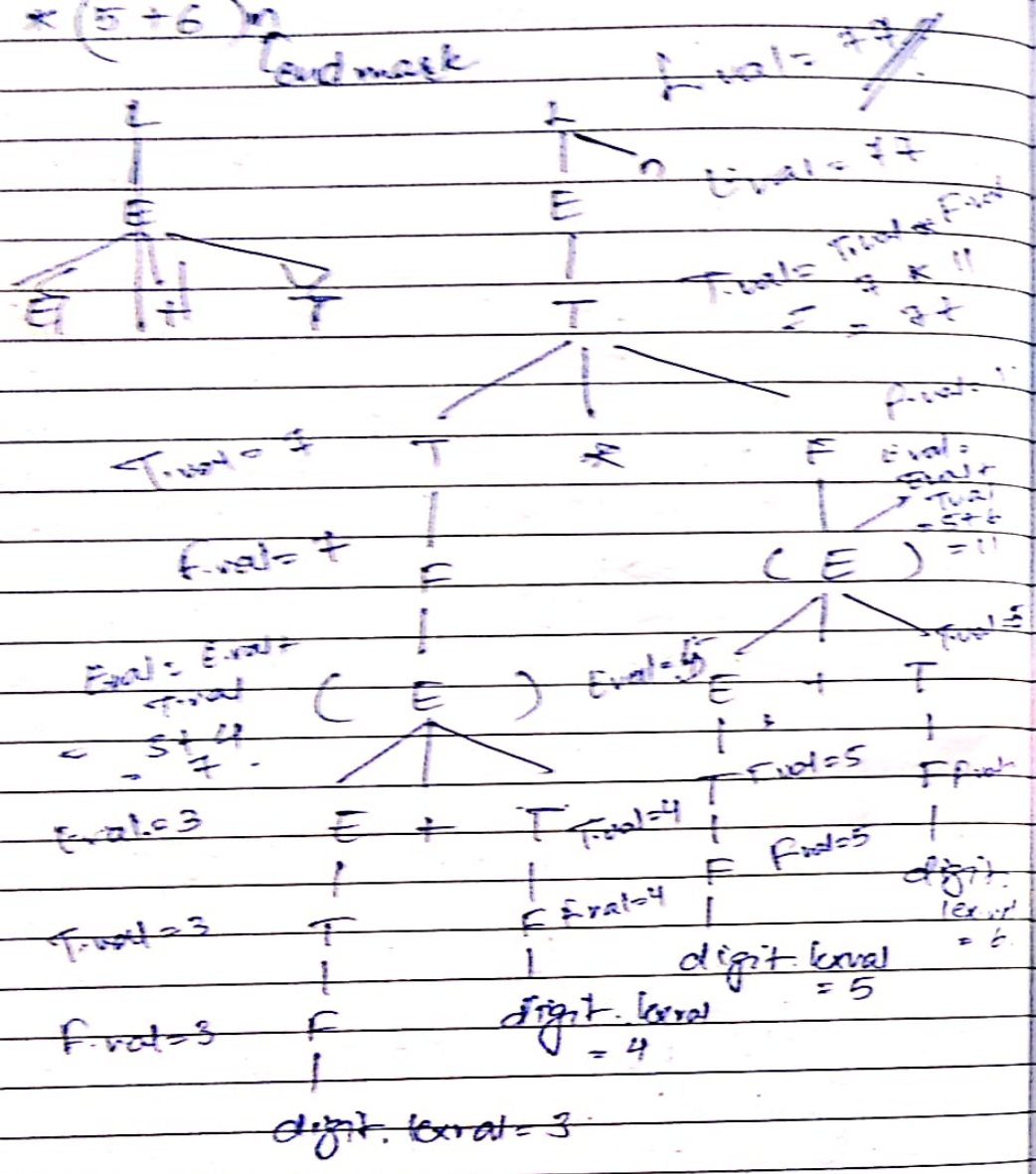
⇓

Parse Tree + attribute + values.

(And also Dependency Graph)

(5) Construct the annotated Tree.

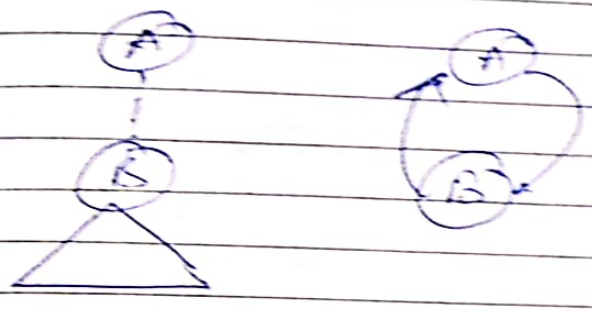
(3+4) * (5+6)n Endmark



Inherited Attribute.

$A \to B$ $\quad A.i = B.i$

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



$T \to FT'$

$T' \to *FT'$

$T' \to \epsilon$

$F \to digit.$

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

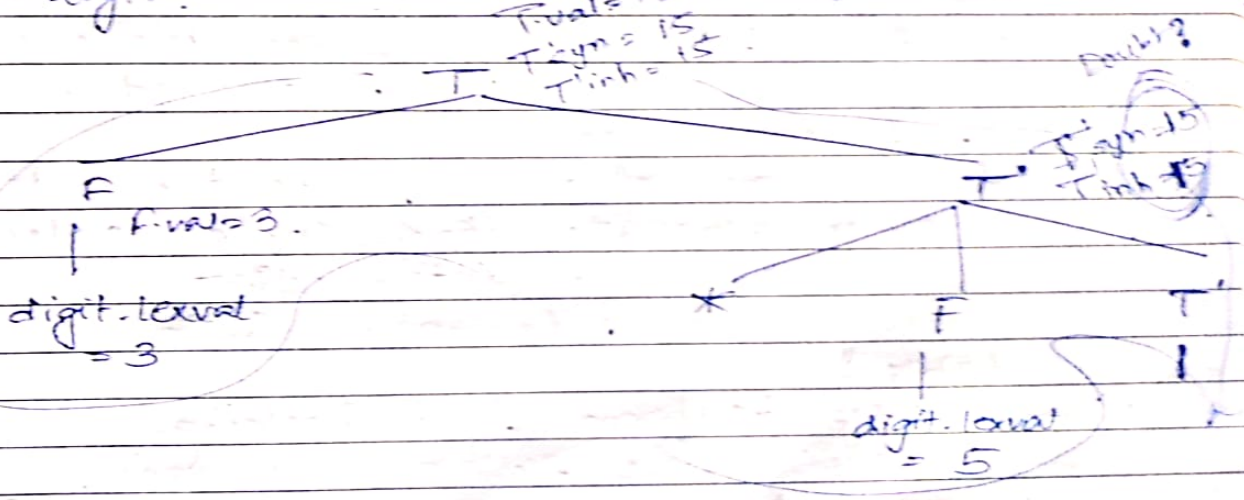
{ $T.val = T'.syn$ }

{ $T'.inh = T'.inh * F.val$ }

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

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

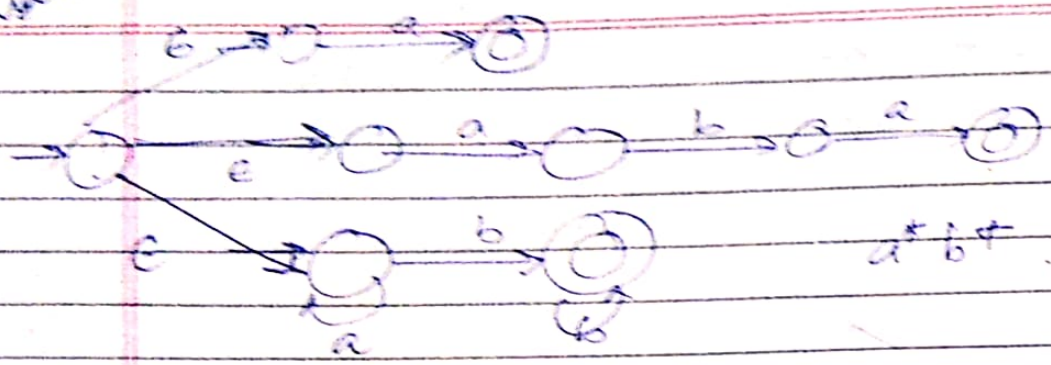
{ $F.val = digit.lexval$ }



Look for examples. $S \to SA \mid (S) \mid a$

Is LL(1) Gram

Refer the textbook for more examples & practise.



abaa

$a^* b^+$

dangling-else ambiguity → 2 diff parse tree for same string.

Look for solutions in email sent.

Hints: Practise the problems solved in textbook.

20/2/20.

* Syntax Directed Translation.

(1) We associate information with the programming language constructs by attaching attributes to grammar symbols.

Values of these attributes are evaluated by the semantic rules associated with the production rules.

Evaluation of these semantic rules:

— may generate intermediate codes.

— may put information into the symbol table.

— may perform type checking.

— may issue error messages.

— may perform some other activities.

— in fact, they may perform almost any activities.

(Q) An attribute may hold almost any ~~kind~~ thing.
— a string, a number, a memory location, a complex record.

Syntax-Directed Defn's and Translation ~~Rules~~ Schemes.

(Q) When we associate semantic rules with productions, we use two notations :-
— Syntax Directed Defn's.
— Translation Schemes.

(A) Syntax-Directed Translation.
— give high level specification for.

- Conceptually with both the syntax directed translation and translation scheme we
— Parse the input token ~~str~~ stream.
— Build the parse tree.
— Traverse the tree to evaluate the semantic rules at the parse tree nodes.

I/p string → parse tree → dependency graph → evaluation order for

i) A SDD is a generalizath of a CFG in which:-
— Each grammar symbol is associated with a set

Eg:- Syn attribute: $E \rightarrow E1 + E2$ { E.val = E1.val + E2.val }
Inh attri: $A \rightarrow XYZ$ { Y.val = 2 * A.val }

(ii) Semantic Rules set up dependencies b/w attributes which can be represented by a dependency graph.
This dependency graph determines the evaluation order of these semantic rules.

Annotated Parse tree.

(1.) A parse tree showing the values of attributes at each node is called an annotated parse tree.

(2.) Values of Attributes in nodes of

In a SDD, each production $A \to \alpha$ is associated with a set of productions.

SDD - Eg :-

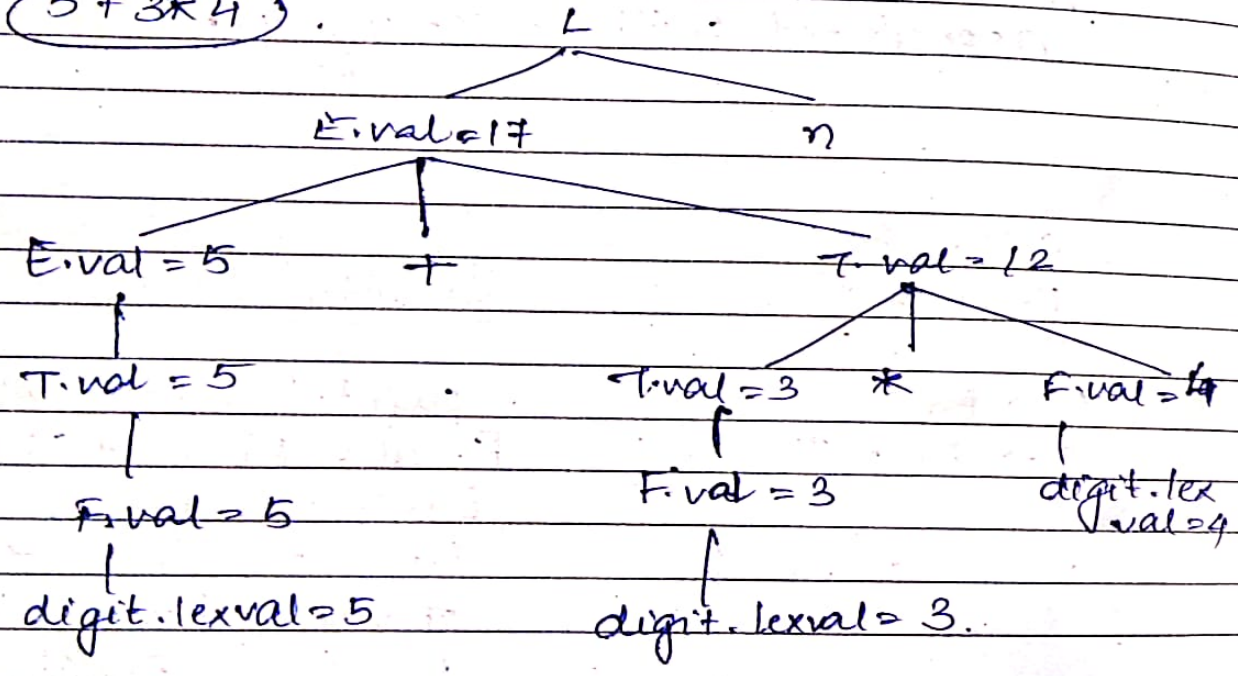
Production	Semantic Rules
$L \to E_n$	Print (E.val)
$E \to E_1 + T$	E.val = E_1.val + T.val
	E.val = T.val
	T.val = T_1.val * F.val
	T.val = F.val
	F.val = E.val
	F.val = digit.lexval.

S-attributed defn:-

- A SDT that uses ~~syntax~~ synthesized attributes exclusively is said to be a S-attributed defn.
- A parse tree for a S-attributed defn can be annotated by evaluating the semantic rules for the attributes at each node, bottom-up from leaves to the root.

Annotated Parse Tree - Eg.

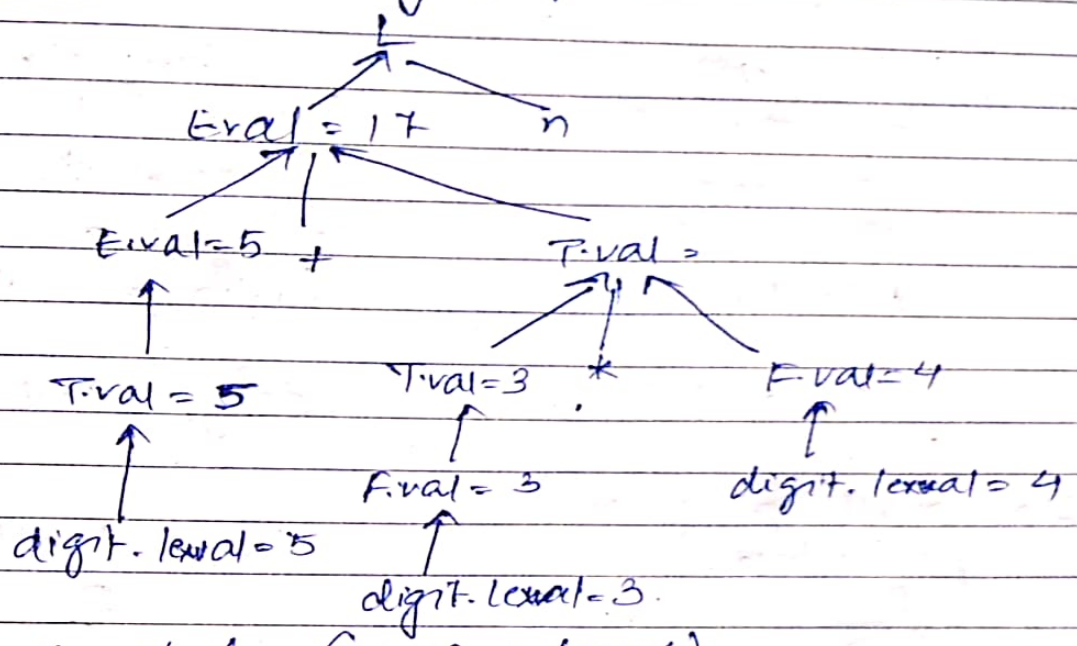
I/P :- ~~5*3+4~~ 5+3*4



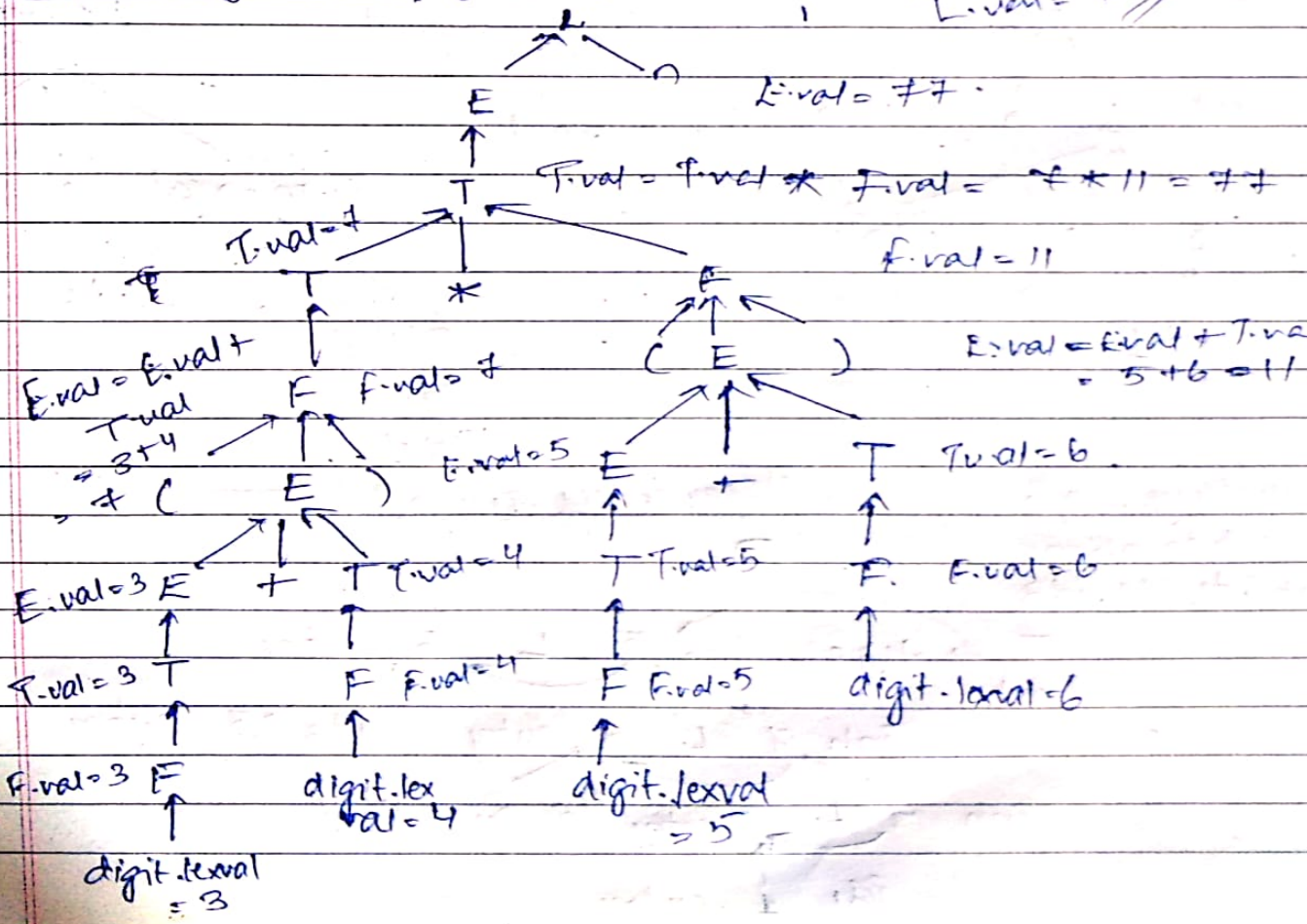
~~Dependency Graph:~~

Date / /20

Ex:- 5+3*4 Dependency Graph.



* Dependency Graph for (3+4) * (5+6)n

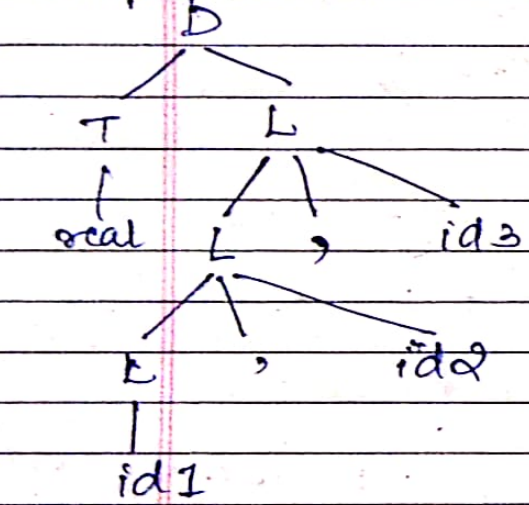


Inherited attributes.

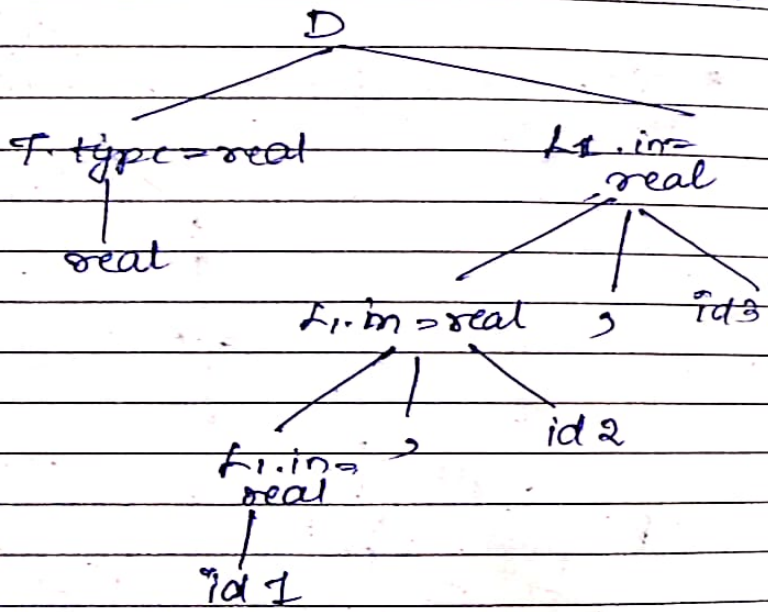
Production	Semantic Rules.
$D \rightarrow TL$	$L.in = T.type$.
$T \rightarrow int$	$T.type =$ integer.
$T \rightarrow real$	$T.type =$ real
$L \rightarrow L_1, id$	$L_1.in = L.in$, addtype (id.entry, L.in)
$L \rightarrow id$	addtype (id.entry, L.in)

4 p : real p, q, r

parse tree.

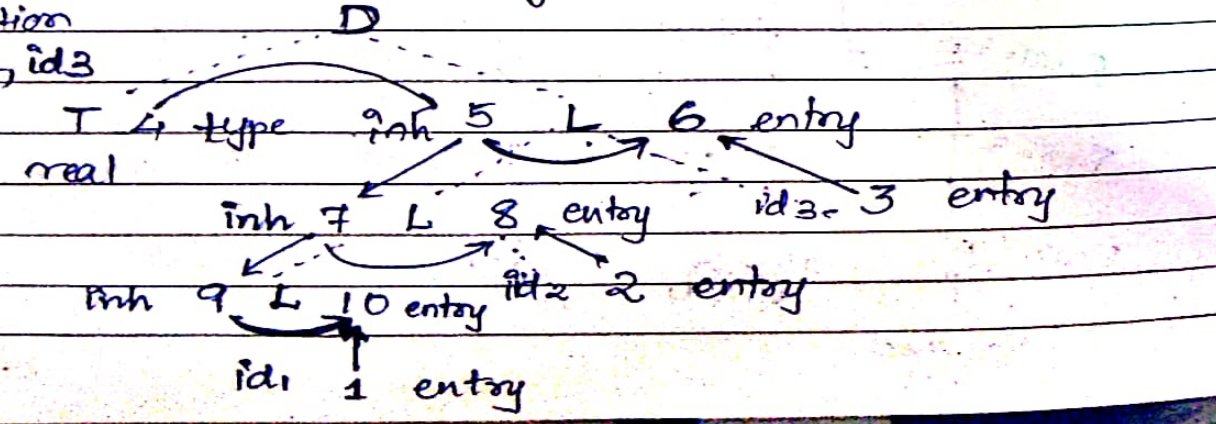


Annotated Parse Tree.



Dependency Graph for a declaration float id1, id2, id3

Dependency Graph



Date / /20

Dependency Graph Construction.

for each node n in the parse tree do
  for each attribute a of the grammar
    symbol at n do
    construct a node in the dependency
    graph for a.

for each node n in the parse tree do
  for each

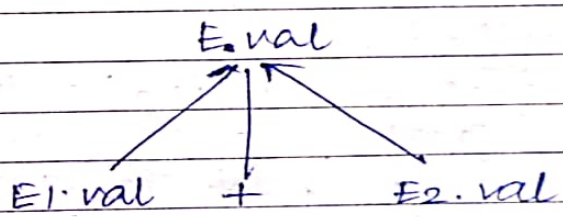
- Example:-

- Production.

E → E1 + E2

Semantic Rule

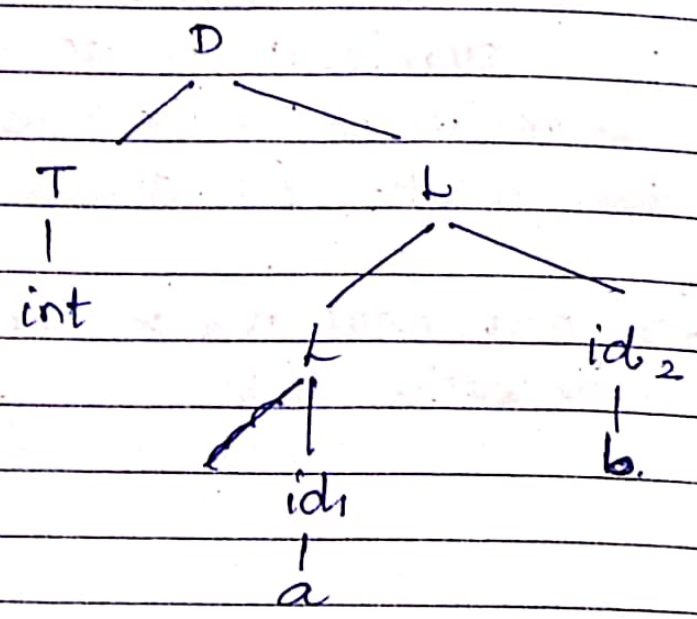
E.val = E1.val + E2.val



= (1/2+1)

should be added to the productions for the 2nd string/ip.

int a, b;
double a = 4, b;



Evaluating Semantic Rules.

- Parse tree methods.
  - At compile time evaluation orders obtained from the topological sort of dependency graph.
  - fails if dependency graph has a cycle.
- Rule Based Methods.
  - Semantic Rules analyzed by hand or specialized tools at compiler construction time.

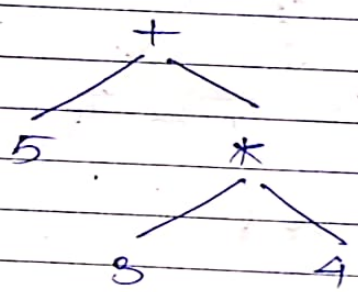
Date / /20

Syntax tree.

- an intermediate representation of the compiler's i/p.
- A condensed form of the parse tree.
- Syntax tree shows the syntatic structure of the program while omitting irrelevant details.
- Operators & keywords are associated with the interior nodes.
- Chains of simple productions are collapsed.
- SDT can be based on syntax tree as well as parse tree.

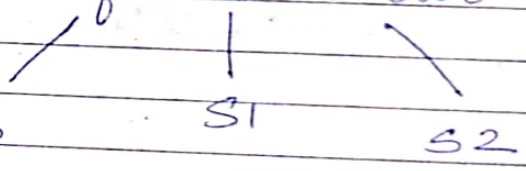
Syntax Tree - Examples

Expression:



- Leaves.

if B then S1 else S2
if - then - else



Stmt:-

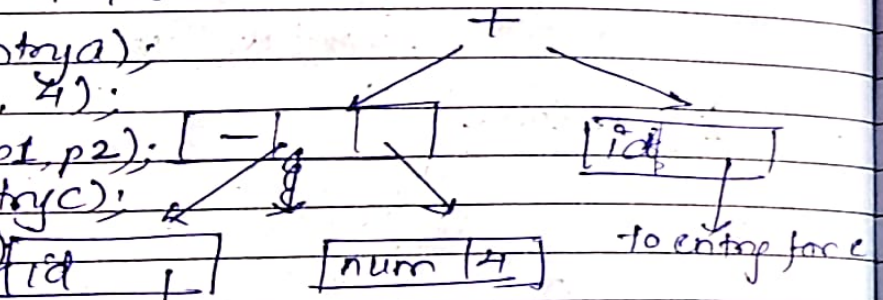
- Node's label indicates what kind of a stmt it is
- Children of a node correspond to

Constructing Syntax Tree for Expressions

- Each node can be implemented as a record with several fields.
- Operator node: one field identifies the operator (called label of the node) & remaining fields contain pointers to operands.
- The nodes may also contain fields to hold the values (pointers to values) of attributes attached to the nodes.

Eg:- $a - 4 + c$

(1) p1 := mkleaf(id, entry a);
p2 := mkleaf(num, 4);
p3 := mknode(-, p1, p2);
p4 := mkleaf(id, entry c);
p5 := mknode(+, p3, p4);



⟹ A Syntax Directed Defn for constructing Syntax Tree.

(1) It uses underlying productions of the grammar to schedule the calls of the funct's mkleaf & mknode to construct the syntax tree.

(2) Employment of the synthesized attribute nptr (pointer) for E and T to keep track of the pointers returned by the function calls.

Date / /20

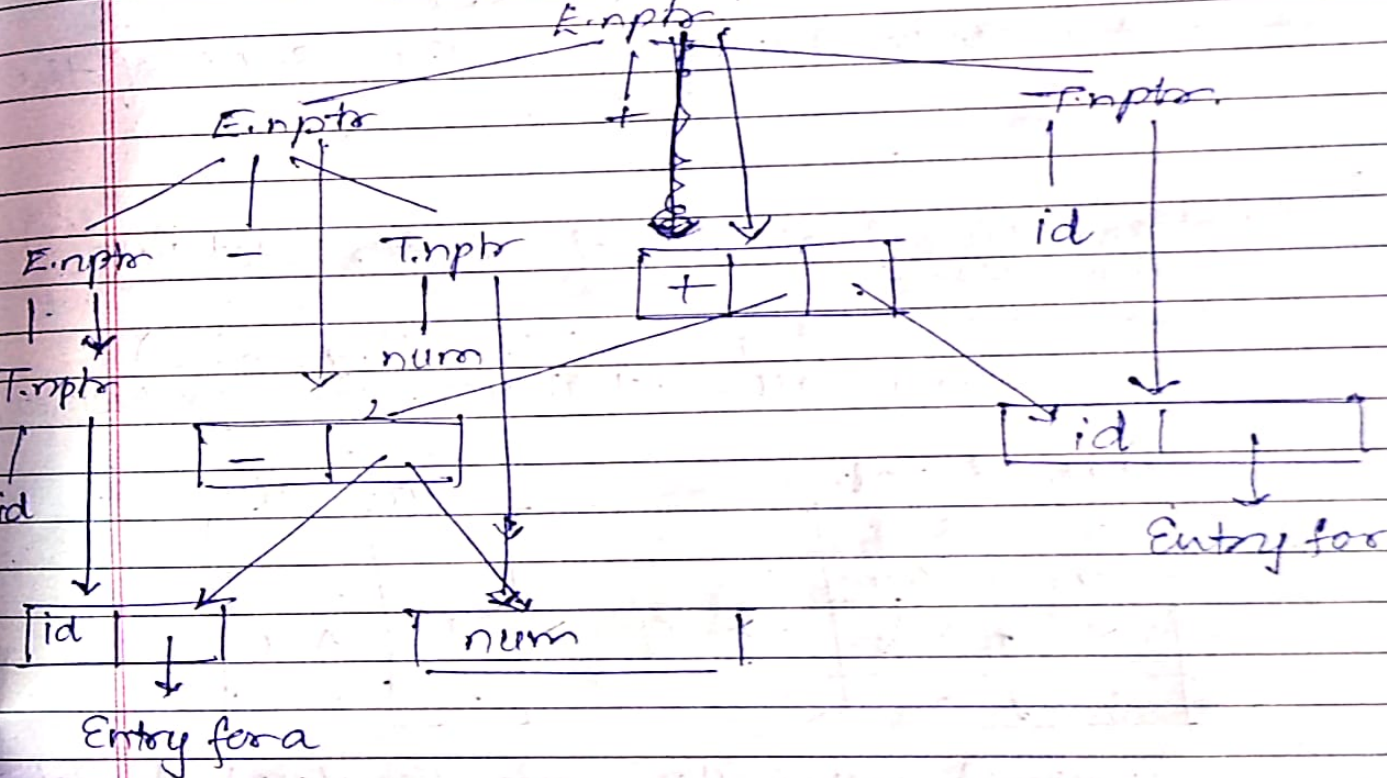
Production Semantic Rule:

$E \rightarrow E_1 + T$ E.nptr = mknode("+", E_1.nptr, T.nptr)

$E \rightarrow E_1 - T$ E.nptr = mknode("-", E_1.nptr, T.nptr)

$T \rightarrow num$

Annotated parse Tree (a-4+c)



S-Attributed Defns.

(i) Syntax-directed defns are used to specify SDT.

(ii) To create a translator for an arbitrary SDD can be difficult.

(iii) We would like to evaluate the semantic rules during parsing. (ie. in a single pass we will parse & we will also evaluate semantic rules during the parsing).

(iv) We will look at two sub-classes of the SDD:

— S-Attributed Defns: only synthesized attributes used in the SDD.

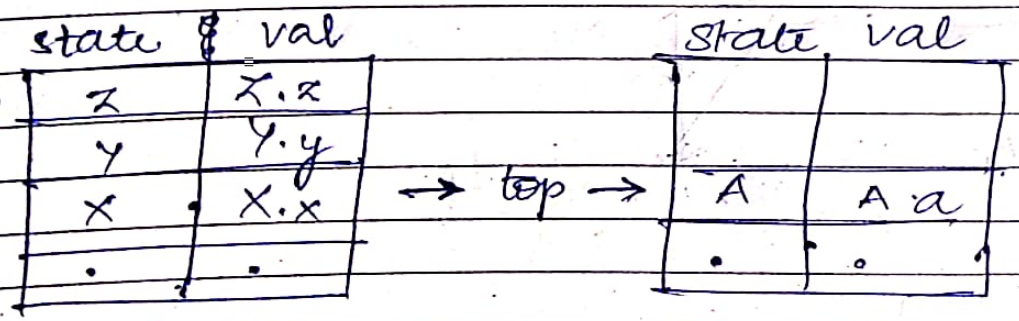
— All actions occurs on the R.H.S of the

Bottom-up Evaluation of S-Attributed Defns.

~~A translator~~

$A \to XYZ \qquad A.a = f(X.x, Y.y, Z.z)$

where all attributes are synthesized.



- Synthesized attributes are evaluated before each reductn.
- Before XYZ is reduced to A, the value of Z.z is in val[top], that of Y.y in

Date / /20

val [top-1] & that of X.x in val [top-2]

Bottom-Up Evaluatn of S-Attributed Defns.

Producn.	Semantic Rules.
L → En	print(val [top-1]
E → E1 + T	val [ntop] = val [top-2] + val [top]
E → T	
T → T1 * F	val [ntop] = val [top-2] * val [top]
T → F	
F → (E)	val [ntop] = val [top-1].
F → digit	

(1) At each shift of digit, we also push digit.lexval into val-stack.

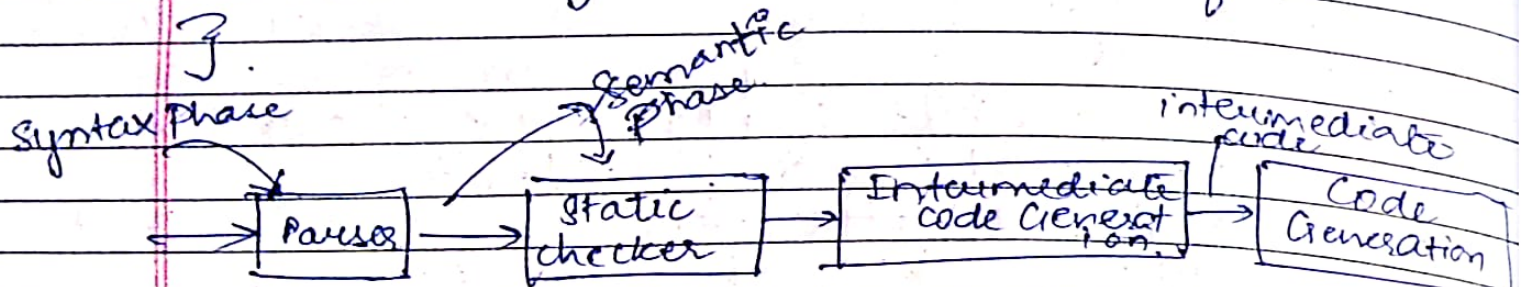
(2) At all other shifts, we do not put anything into val-stack because.

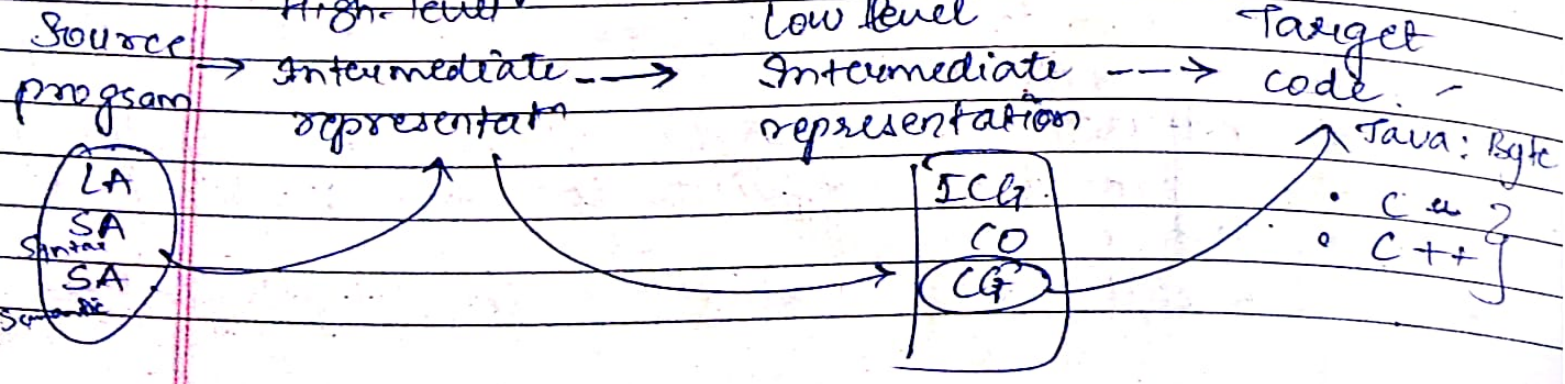
L-attributed Defns.

- When translatn takes place during parsing, order of evaluatn is linked to the order in which nodes of a parse tree are created by parsing method.
- A natural order can be obtained by applying the procedure dfvisit to the root of a parse tree.
- We call this evaluatn order depth first order.

- L-attributed defn is a class of SDD whose attributes can always be evaluated in depth first order (L stands for left since attribute information flows from L to R).

```
dfvisit (node n)
{
    for each child m of n, from left to
    right
    {
      evaluate inherited attributes of m
      dfvisit (m)
    }
    evaluate synthesized attributes of n
}.
```





26/2/20

Date 26/2/20.

CHP 6. Compiler Course.

Outline

* Variants of syntax Trees
* Three-address code.
* Types and declarations.
* Translation of expressions.

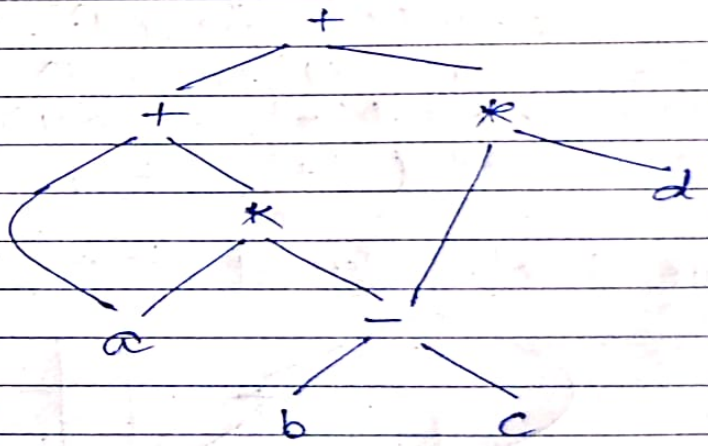
Introductn:-

- Intermediate code is the 'interface b/w front end & back end in a compiler.
- Ideally the details of source lang are confined to

Variants of Syntax Trees

It is sometimes beneficial to create a DAG instead of tree for expressions
This way we can easily show the common sub-exprs & then use that knowledge during code generatn.

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



machine independent:
code.

object code.

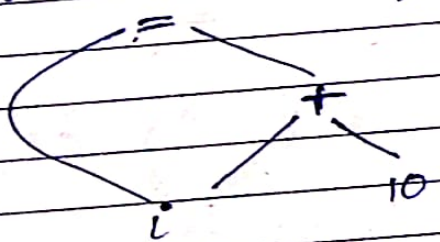
SDD for creating DAG's.

Production	Semantic Rules
$E \rightarrow E1 + T$	E.node = new Node('+', E1.node, T.n
$E \rightarrow E1 - T$	E.node = new Node('-', E1.node, T.n
$E \rightarrow T$	E.node = T.node
$T \rightarrow (E)$	T.node = E.node
$T \rightarrow id$	T.node = new Leaf(id, id.entry
$T \rightarrow num$	T.node = new Leaf(num, num.

Eg: (Order of Evaluation).

1.) p1 = Leaf (id, entry-a)
2.) p2 = Leaf (id, entry-a) = p1
3) p3 = Leaf (id, entry-b)
4) p4 = Leaf (id, entry-c)
5.) p5 = Node('-', p3, p4)
6.) p6 = Node('*', p1, p5)
7.) p7 = Node('+', p1, p6)
8.) p8 = Leaf (id, entry-b) = p3
9.) p9 = Leaf (id, entry-c) = p4
10.) p10 = Node('-', p3, p4) = p5
11.) p11 = Leaf (id, entry-d)
12.) p12 = Node('*', p5, p11)
13.) p13 = Node('+', p7, p12)

Value Number Method for constructing DAG's.



id		
num	10	
+	1	2
3	1	3

To entry for i

Nodes of a DAG for i = i + 10.

allocated in an array.

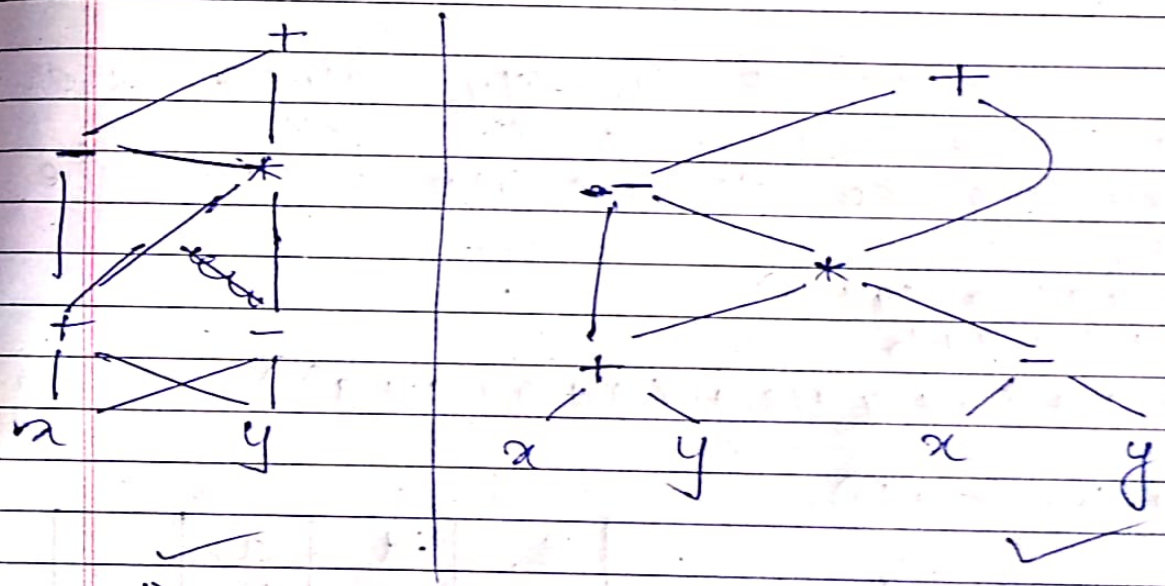
Date / /20

→ Algorithm:-

- Search the array for a node M with label op, left child l and right child r.
- If there is such a node, return the value number M.
- If not create in the array a new node N with label op, left child l, & right child r & return its value
- We may use a hash table.

(8.) Construct the DAG for the expression.

$((x+y) - ((x+y)*(x-y))) + ((x+y)*(x-y))$



Eg:-

p1 = Leaf (id, entry - x)
p2 = Leaf (id, entry - y)
p3 = Node ('+', p1, p2)
p4 = Node ('-', p1, p2)
p5 = Node ('*', p3, p4)
p6 = Node ('-', p3, p5)
p7 = Node ('+', p6, p5)

$((x+y) - ((x+y) * (x-y))) + (((x+y) * (x-y)))$

Eg:-
p1 = leaf (id, entry - x)
p2 = leaf (id, entry - y)
p3 =

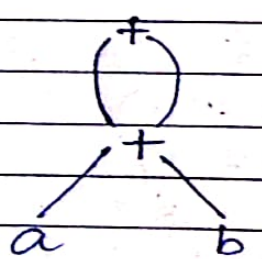
Q. Construct DAG & identify value nos for the subexpressions of the foll exprs, assuming + associates from the left.

(a) a + b + (a + b)

(b) a + b + a + b

(c) a + a + (a + a + a + (a + a + a + a))

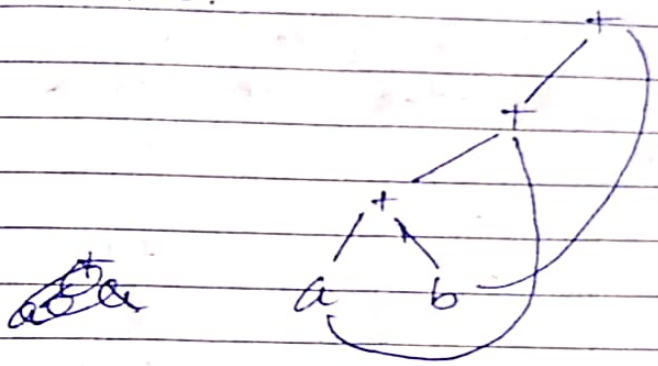
(a) a + b + (a + b)



1.	id	a	
2.	id	b	
3.	+	1	2
4.	+	3	3

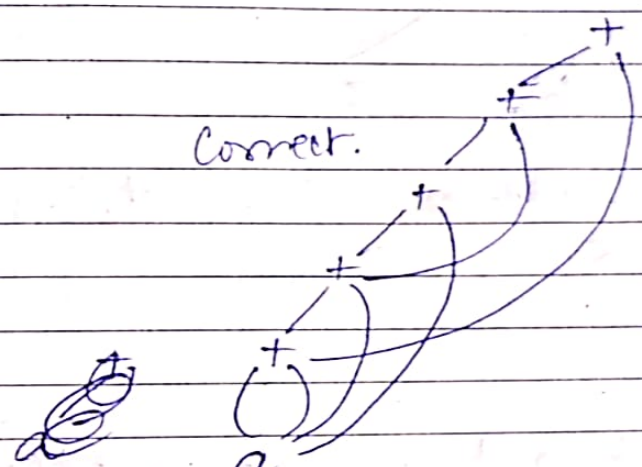
Date / /20

(b) $a+b+a+b$.

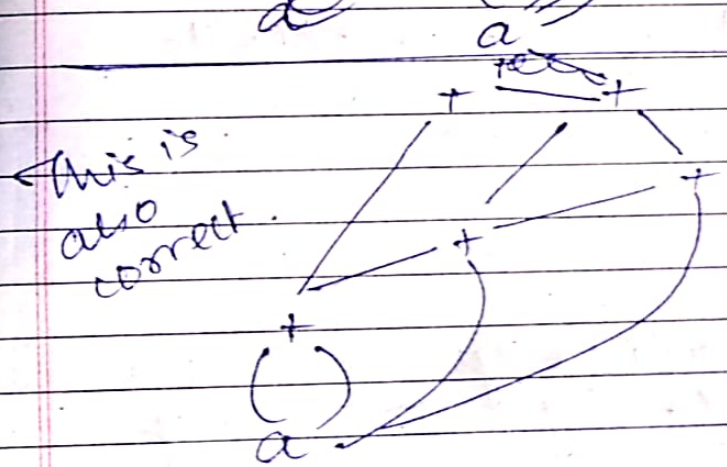


1.	id	a	
2.	id	b.	
3.	+	1	2
4.	+	3	1
5.	+	4	2.

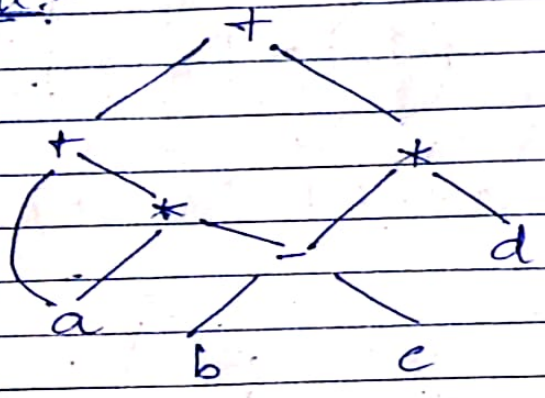
(c) $a+a+(a+a+a+(a+a+a+a))$



1.	id	a	
2.	+	1	1
3.	+	2	1
4.	+	3	1
5.	+	3	4
6.	+	2	5



* Three Address Code:-
- In an 3-address code there is at most one operator at right side of an instruction.
- Example:-



t1 = b - c
t2 = a * t1
t3 = a + t2
t4 = t1 * d
t5 = t3 + t4

Forms of 3-address instructns.
- x = y op z
- x = op y
- x = y
- goto L
- if x goto L and if false x goto L
- if x relop y goto L
- Procedure calls using:
  - param x
  - call p, n
  - y = call p, n
- x = y[i] and x[i] = y
- x = &y and x = *y and *x = y

Example:-
- do i = i+1; while (a[i] < v);

L: t1 = i + 1
i = t1
t2 = i * 8

100: t1 = i + 1
101: i = t1
102: t2 = i * 8

$t3 = a[t2]$
if $t3 < v$ goto L
Symbolic labels.

103: $t3 = a[t2]$
104: if $t3 < v$ goto 100.
Position Numbers.

⇒ Data structures for 3-add codes:

⇒ Quadruples
- Has four fields: op, arg1, arg2 & result.

⇒ Triples
- Temporaries are not used & instead ref to instructn are made.

⇒ Indirect triples
- In addn to triples we use a list of pointers to triples.

24/8/20

Example:-
b * minus c + b * minus c

Three address code.
t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4
a = t5.

Quadruples

op	arg1	arg2	result
minus	c		t1
*	b	t1	t2
minus	c		t3
*	b	t3	t4
+	t2	t4	t5
=	t5		a

Page

Triples

	op	arg1	arg2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

Indirect Triples.

	op
35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)

	op	arg1	arg2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

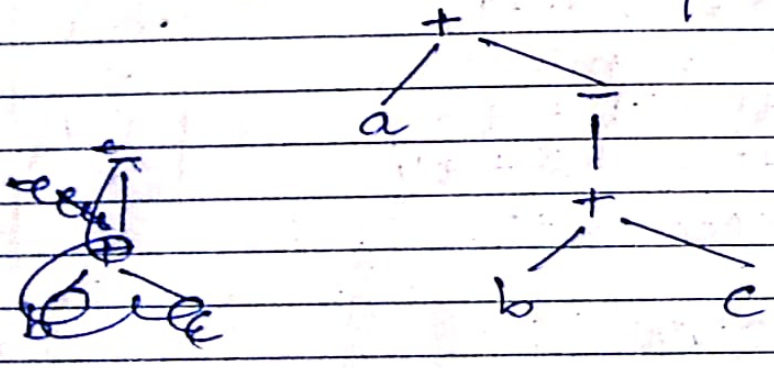
Exercise:-

(1.) Translate the arithmetic expression $a + -(b+c)$ into

(a) Syntax Tree (b) Quadruples

(c) Triples (d) Indirect Triples.

Sol^n (a)



Date / /20

Quadruples.

	op	arg1	arg2	result
0	+	b	c	t1
1	minus	t1		t2
2	+	a	t2	t3

Triples

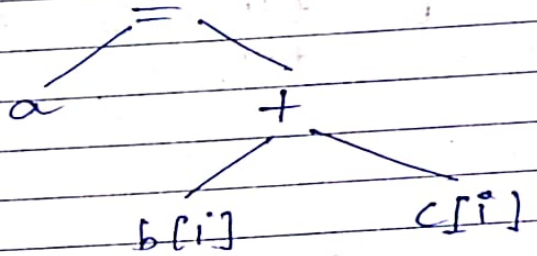
	op	arg1	arg2
0	+	b	c
1	minus	(0)	
2	+	a	(1)

Indirect triples

instruction	
0	(0)
1	(1)
2	(2)

	op	arg1	arg2
0	+	b	c
1	minus	(0)	
2	+	a	(1)

(2) $a = b[i] + c[j]$



Quadruple:

0)	=[]	b	i	t1
1)	=[]	c	j	t2
2)	+	t1	t2	t3
3)	=	t3		a

Triple:

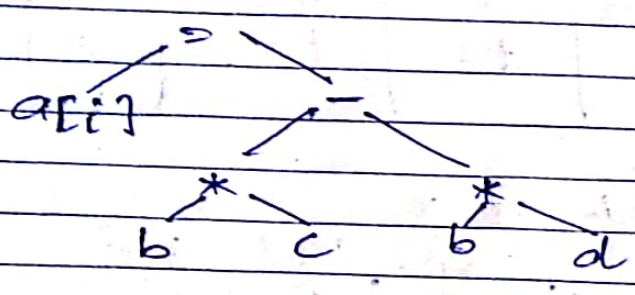
0)	=[]	b	i
1)	=[]	c	j
2)	+	(0)	(1)
3)	=	a	(2)

Indirect triples.

35)	0)
36)	1)
37)	2)
38)	3)

0)	=[]	b	i
1)	=[]	c	j
2)	+	(0)	(1)
3)	=	a	(2)

(3) a[i] = b*c - b*d.



Quadruple :-

0.)	*	b	c	t1
1.)	*	b	d	t2
2.)	-	t1	t2	t3
3.)	[ ]=	a	i	t4
4.)	=	t3		t4.

Triples

0)	*	b	c
1)	*	b	d
2)	-	(0)	(1)
3)	[ ]=	a	i
4)	=	(3)	(2)

Indirect triples.

(0)
(1)
(2)
(3)
(4)

(4) c.x = f(y+1) + 2

Quadruple.

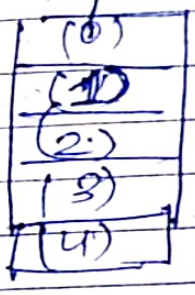
(0.)	+	y	1	t1
(1.)	param	t1		
(2.)	call	f	t1	t2
(3.)	+	t2	2	t3
(4.)	=	t3		x.

Triple.

0)	+	y	1
1.)	param	(0)	
2.)	call	f	(1)
3)	+	(2)	2
4.)	=	x	(3)

Date / /20

Indirect triple.



(4) $x = *p + \&y.$

*

* Static single - Assignment Form.
- Static single - assignment form (SSA) is an intermediate representatn that facilitates certain code optimizatns.
- Two distinctive aspects. distinguish SSA from 3-add code.
- The first is that all assignments in SSA are to variables with distinct names; hence the term single - assignment.

```
p = a + b
q = p - c
  = q * d
```

```
p1 = a + b
q1 = p1 + c
p2 = q1 * d
p3 = e - p2
q2 = p3 + q1
```

→ Types & declaratns:-
→ The app^n's of types can be grouped under checking & translation:
- Type checking uses logical rules to reason about the behaviour of

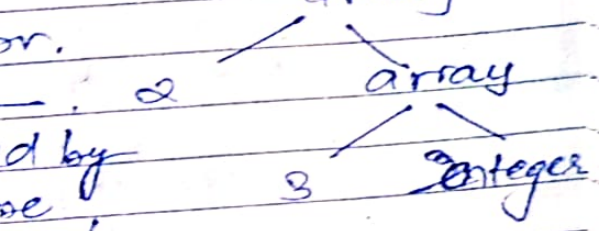
Date / /20

Type Expressions

Eg:- int[2][3]

array(2, array(3, integer))

- A basic type is a type expr.
  A type name is a type expr.
  A type expr can be formed by applying the array type constructor to a number & a type expr.
  A record is a data structure with named field.
  A type expr can be formed by using the type constructor → for functn types.
  If s and t are type exprs, then

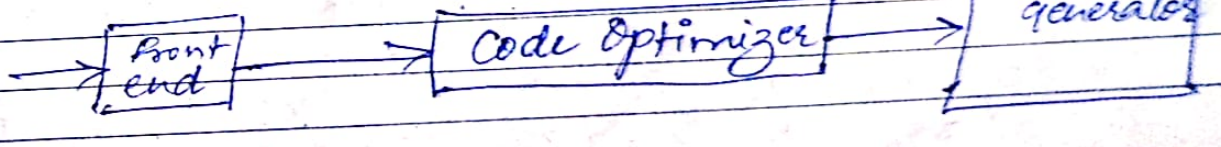


2/3/20

Code Generation

Introduct^n:-

- The final phase of a compiler is code generator.
- It receives an intermediate represent^n (IR) with supplementary informat^n in symbol table.
- Produces a semantically equivalent target program.
- Code generator main tasks:-
  - Instruction selection.
  - Register allocation & assignment.
  - Instruction ordering.



Issues in the Design of Code Generator.

- The most important criterion is that it produces correct code.
- Input to the code generator.
- IR + Symbol table.
- We assume front end produces low-level IR, ie. values of names in it can be directly manipulated by the m/c instruc
- Syntatic

Complexity of mapping:-

- the level of the IR.
- the nature of the instruction-set architecture
- the desired quality of the generated code.

$x = y + z$

```
LD   R0, y
ADD  R0, R0, z
ST   x, R0
```

target op1 op2

$a = b + c$
$d = a + e$

```
LD   R0, b
ADD  R0, R0, c
ST   a, R0
LD   R0, a
ADD  R0, R0, e
ST   d, R0
```

Register allocation:-

- Two subproblems.
  - Register allocation: selecting the set of variables that will reside in registers at each point in the program.
  - Register assignment: selecting specific register that a variable will reside in

Date / /20

- Complications imposed by the h/w architecture
- Eg:- register pairs for multiplication & division.

```
t = a + b          L    R1, a
t = t * c          A    R1, b
T = t/d.           M    R0, c
                   D    R0, d
                   ST   R1, t.

t = a + b          L    R0, a
t = t + c          A    R0, b
T = t/d.           M    R0, c
                   SRDA - R0, 32
                   D    R0, d
                   ST   R0, t.
```

shift Right Double arithmetic

16 bit they have double for carry purpose.

⇒ A simple target m/c model

- Load operations : LD R, x and LD r1, r2
- Store operations : ST x, r
- Computatn op^ns : OP dst, src1, src2
- Unconditional jumps : BR L
- Conditional jumps : B cond r, L like BLTZ, r, L.

Addressing modes.

variable name : x

Indexed address : a(r) like LD R1, a(R2) means.

R1 = contents (a + contents (R2))

integer indexed by a register : like LD R1, 100 (R2)

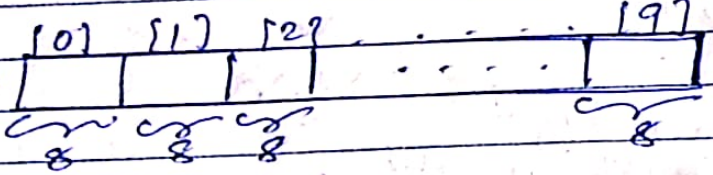
1 byte = 8 bits

Indirect addressing mode: *r and *100(r)
immediate constant addressing mode:
like LD R1, #100.

① b = a[i]

```
LD   R1, i        // R1 = i
MUL  R1, R1, 8    // R1 = R1 * 8
LD   R2, a(R1)    // R2 = contents(a + contents(R1))
ST   b, R2.       // b = R2.
```

8 for byte

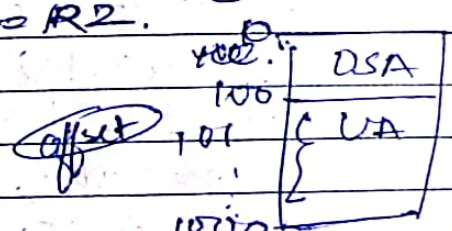


② a[j] = c

```
LD   R1, c        // R1 = c
LD   R2, j        // R2 = j
MUL  R2, R2, 8    // R2 = R2 * 8
ST   a(R2), R1    // contents
```

③ x = *p

```
LD   R1, p        // R1 = p
LD   R2, 0(R1)    // R2 = contents(0 + contents
ST   x, R2        // x = R2.
```



Date / /20

⇒ conditional-jump three-address instructions.

If $x < y$ goto L

```
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
```

⇒ Costs associated with the addressing modes

- LD R0, R1 — cost = 1
- LD R0, M — cost = 2
- LD R1, *100(R2) — cost = 3

⇒ Addresses in the Target code.

- A statically determined area code.
- A ←—— ——→ data area Static
- A dynamically managed area Heap.
- A ←—— ——→ area Stack.

⇒ Exercises:-

(1) Generate code for the foll three-address stmts assuming all variables are stored in memory locations.

a) $x = 1$

```
LD R1, #1
ST x, R1
```

b) $x = a$

```
LD R1, a
ST x, R1
```

(R1) c) $x = a + 1$

```
LD R1, a
ADD R1, R1, #1
ST x, R1
```

d) $x = a + b$.

```
LD R1, a
LD R2, b
ADD R1, R1, R2
ST x, R1
```

e) The two statements

$x = b * c$

$y = a + x$.

e.)
```
LD R1, b
LD R2, c
MUL R1, R1, R2
LD R3, a
ADD R3, R3, R1
ST y, R3.
```

(a) Generate code for foll 3-address stmts assuming a & b are arrays whose elements are 4-byte values.

a.) The four stmts sequence.

```
x = a[i]                LD R1, i
y = b[j]                MUL R1, R1, #4
a[i] = y                LD R2, a(R1)
b[j] = x.               LD R3, j
                        MUL R3, R3, #4
                        LD R4, b(R3)
                        ST a(R1), R4
                        ST b(R3), R2
```

(b) The three-stmt sequence.

```
x = a[i]                LD R1, i
y = b[i]                MUL R1, R1, #4
z = x * y.              LD R2, a(R1)
                        LD R1, b(R1)
                        MUL R1, R2, R1
                        ST z, R1
```

Types:- (1) LAN
(2) MAN
(3) WAN
(4) SAN.

Date / /20

Introduction

Definition of a Distributed System.

A collection of independent computers that appears to its users as a single coherent systems.

Several important aspects :-

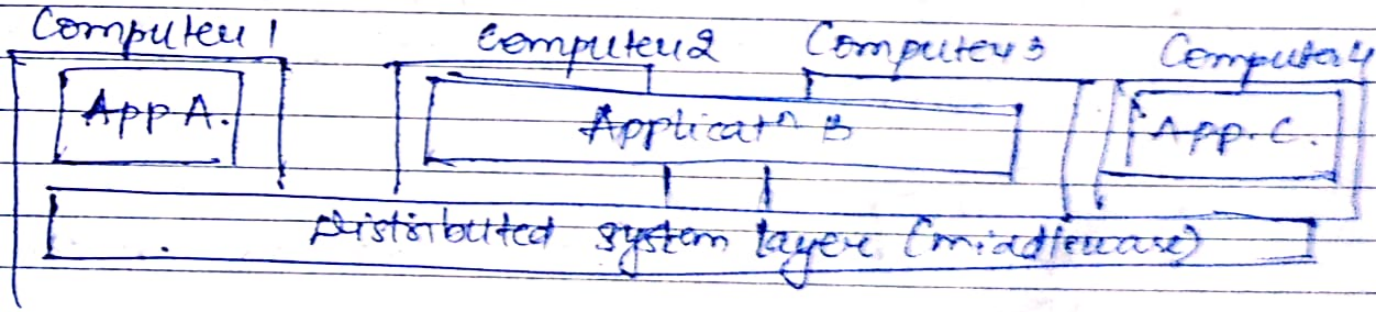
(1) ~~Consider~~ Consists of components that are autonomous.

(2) Users think they are dealing with a single system.

(3) No assumptions made regarding the types of

→ Distributed systems are often organized by means of a layer of s/w :-

- logically placed b/w a higher level layer consisting of users & app^ns & a lower layer consisting of OS and basic communicatn facilities.
- middleware



Goals of Distributed Systems.

- Making resources ~~available~~ accessible.
- Distribution transparency.
- Openness
- Scalability.

Making Resources Accessible.

- Making it easy for users & app^ns to access remote resources.
- Share remote resources in a controlled and efficient manner.

⇒ **Benefits of sharing remote resources.**

- Better economics By sharing expensive resources.
- Easier to collaborate & exchange informat
- Connectivity of the Internet has lead to numerous virtual organizations where geographically dispersed people can work together using groupware.
- Connectivity has enabled electronic commerce
- However, as connectivity & sharing increase....

- Security Problems.
- Eavesdropping or intrusion on communic ath.
- Tracking of communicatn to build up a preference profile of a specific user.

Date / /20

Distribution Transparency:

An important goal - hide the fact that the processes and resources are physically distributed across multiple computers.

Transparent :- A distributed system that presents itself to users

Types of Transparency.

Transparency	Description
Access	Hide differences in data representation & how a resource is accessed.
Location	Hide where a resource is located.
Migration	Hide that a resource may move to another location.
Relocation	Hide that a resource may be moved to another location while in use.
Replication	Hide that a resource is replicated.
Concurrency	Hide that a resource may be shared by several competitive users.
Failure.	Hide the failure and recovery of a resource.

Different forms of transparency in a distributed system (ISO, 1995).

Access Transparency :- hide differences in data representation and the way the resources are accessed.

Eg :- a distributed system may have computer systems that run different OS, each having

Location Transparency: users cannot tell where a resource is physically located in the system. Achieved by assigning only logical names to resources.

Relocation Transparency: resources can be relocated while they are being accessed without the user or app^n noticing anything.

Failure Transparency: a user does not notice that a resource fails to work properly, & that system subsequently recovers from that failure.

Sharing of resources can also be done in a competitive way:

Degree of Transparency: Complete hiding the distribut^n aspects from users is not always a good idea.
Attempting to mask a server failure before trying another one may slow down the system.
Requiring several replicas to be always consistent means a single update op^n may take seconds to complete.
For mobile & embedded devices it may be better to expose distribut^n rather than trying to hide it. Signal transmission is limited by speed of light as well as the speed of

Date / /20

Openness: An open distributed system offers services according to standard rules that describe syntax & semantics of those services. Services are generally specified through interfaces, which are often described in an Interface Defn Lang (IDL).

An interface defn – allows an arbitary process that needs a certain interface to talk to another process that provides that interface. allows two independent parties to build completely diff implementations of those interfaces.
Proper specifications are complete & neutral. completeness & neutrality are imp for interoperability & portability.

Interoperability: characterizes the extent by which two implementations of systems or components from diff manufacturers can co-exist & work together by merely relying on each other's services as specified by a common standard.

Portability: characterizes to what extent an appn developed for a distributed system A can be executed, without modification, on a diff distributed system B that implements

Extensibility:
It should be easy to configure the system out of diff components.
It should be easy to add new components or replace existing ones.

Scalability :- Scalability can be measured against three dimensions :

- Size : be able to easily add more users & resources to a system.
- Geography : be able to handle users & resources that are far apart.
- Administrative : be easy to manage even if it spans many independent administrative organizations.

Scalability Problems :- Consider scaling w.r.t size - we are often confronted with the limitations of centralized services, data & algorithms.

Concept	Example
Centralized service	A single server for all users
Centralized data	A single on-line telephone book
Centralized algo.	Doing routing based on complete information

Eg. of scalability limitations.

Only decentralized algorithms should be used.

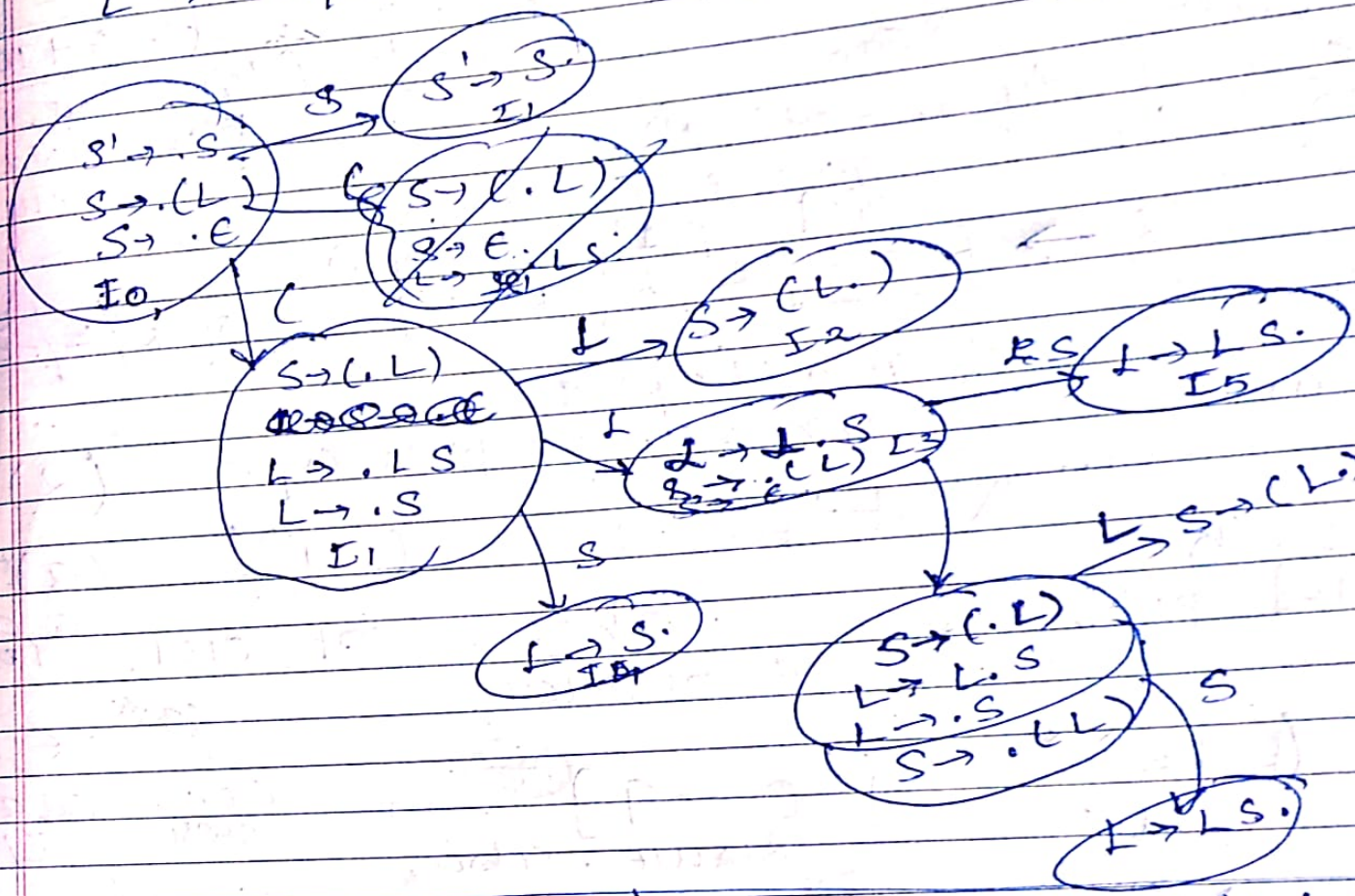
Characteristics of decentralized algorithms :-

- No machine has complete information about the system state.
- M/cs make decisions based only on local information
- failure of one m/c does not ruin the algorithm.
- There is no implicit assumption that global clock exists

→ LANs use synchronous communication. Designing WANs using synchronous communication is much more difficult. Communication in WANs is

Date / /20

S → (L) | ε
L →
S → (L) | ε
L → LS | S.



inherently unreliable, & virtually always point-to-point. LANs use broadcasting.

Security issues:- Many components of a distributed system that reside within a single domain, may not be trusted by users in other domains.