Date Day

▷ SEMANTIC ANALYSIS

→ This step of compiler takes an Abstract Syntax Tree generated by the syntax analysis phase as input.

→ It performs type checking, scope resolution, and validates semantic consistency

→ Core Functions:

① Type checking
② Scope Resolution
③ Checking for correct function calls — verifying that the number and type of args passed to a function or method match its definition
④ Checking for declaration err
⑤ Handling type coercion / casting ← semantic analyzer can perform implicit type conversions.

→ Role of Symbol Table
↳ Heavily used during semantic analysis
↳ perform quick lookups to find an identifier's attri. whenever it is refrenced in the code

Date _____ Day _____

- **Syntax Directed Definition**

↳ A high level, declerative specification that attaches semantic rules to the productions of a context-free gramar.

Production		Semantic Rule (attribute)
$E \to E \# T$	⟶	$E.val = E.val * T.val$
$E \to T$	⟶	$E.val = T.val$
$T \to T \& F$	⟶	$T.val = T.val - F.val$
$T \to F$	⟶	$T.val = F.val$
$F \to digit$	⟶	$F.val = digit.val$

↳ Har production k saath uska semantic rule attach hai

$N \to L$	$N.val = L.val$
$L \to LB$	$L.val = L.val + B.val$
$L \to B$	$L.val = B.val$
$B \to 1$	$B.val = 1$
$B \to 0$	$B.val = 0$

→ If X is a symbol & "a" is its attr. then the value is denoted by X.a

→ In a parse tree, attr. are stored in nodes, much like data fields in an object.

Date _____ Day _____

• Attributes of SSD

↳ Attr hold the semantic information that we need to compute and check

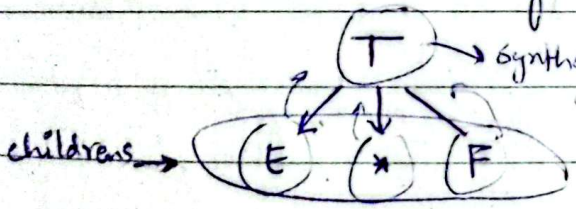
↳ They can be of various data types
- Numbers
- Types (int, float...)
- Table References (pointer to symbol table entry)
- String (variable names)
- code (sequences of intermediate instr)

↳ Two key types of Attr based on how their values flow through the parse tree

(1) Synthesized Attributes

on LHS of production

↳ An attribute for a nonterminal A, whose value is calculated from its children



Synthesized attr, its values is calc from E * F

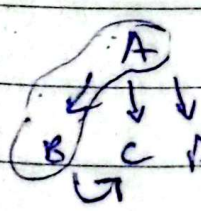
$T \rightarrow E * F$

{ T.val = E.val
T.val = F.val }

(2) Inherited Attributes

on RHS of production

↳ An attribute for a nonterminal A, whose value is either calculated from its parent or siblings



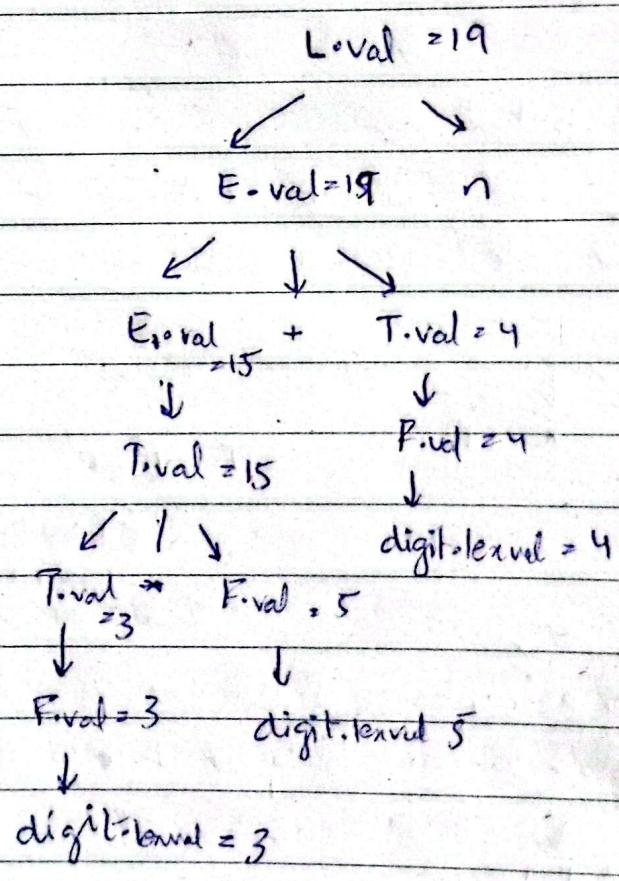
$A \rightarrow BCD$ { C.val = A.val, B.val = A.val, C.val = D.val }

↑ Semantic rules

Date _____ Semantic Rules Day _____

$L \to E_n$	$L.val = E.val$
$E \to E_1 + T$	$E.val = E_1.val + T.val$
$E \to T$	$E.val = T.val$
$T \to T_1 * F$	$T.val = T_1.val \times F.val$
$T \to F$	$T.val = F.val$
$F \to (E)$	$F.val = E.val$
$F \to digit$	$F.val = digit.lexval$

Q: Construct $3 * 5 + 4n$, annoted parse tree



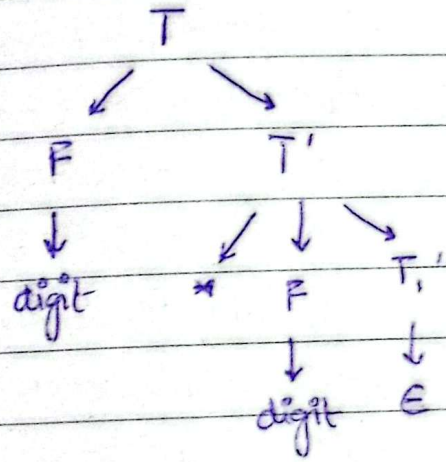
Date _____ Day _____

$T \to FT'$	$T'.inh = F.val$, $T.val = T'.syn$
$T' \to *FT_1'$	$T_1'.inh = T'.inh \times F.val$, $T'.syn = T_1'.syn$
$T' \to \epsilon$	$T'.syn = T'.inh$
$F \to digit$	$F.val = digit.lexval$

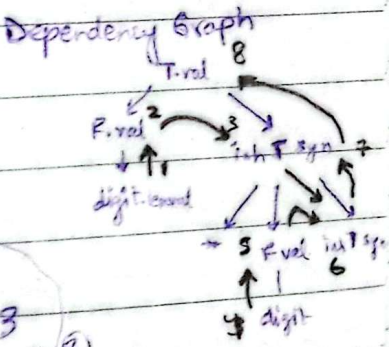
Q: Construct for 3 * 5

A concrete Parse Tree

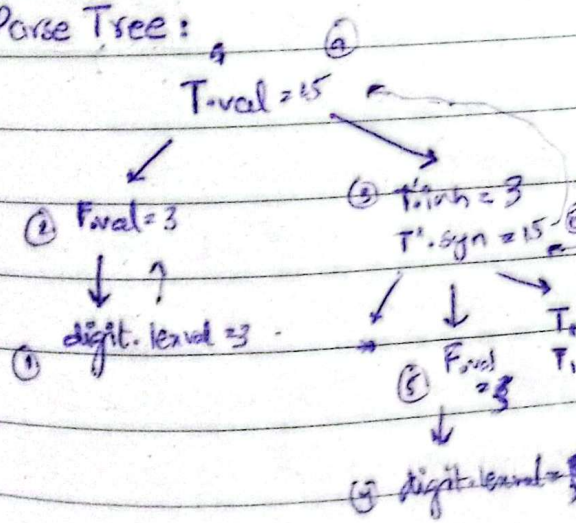
Parse Tree :



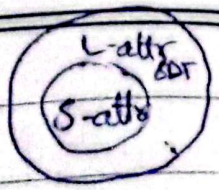
Dependency Graph



Annotated Parse Tree :



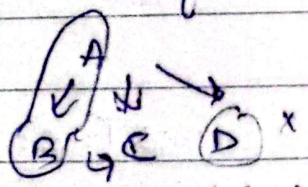
Date _____ Day _____



• Types of SDD

1. S-Attributed
   - ↳ An SDD that uses synthesized attributes is called S-attributed SDD
   - ↳ Information flows from leaves to root
   - ↳ They are guranteed to be non-cyclic
   - ↳ Calculate attribute using Postorder traversal
   - ↳ Fits perfectly with Bottom up Parsing

2. L-Attributed
   - ↳ Both Synthesized & Inherited attributes
   - ↳ The value of inherited attribute can only be obtained from it's parent or left siblings



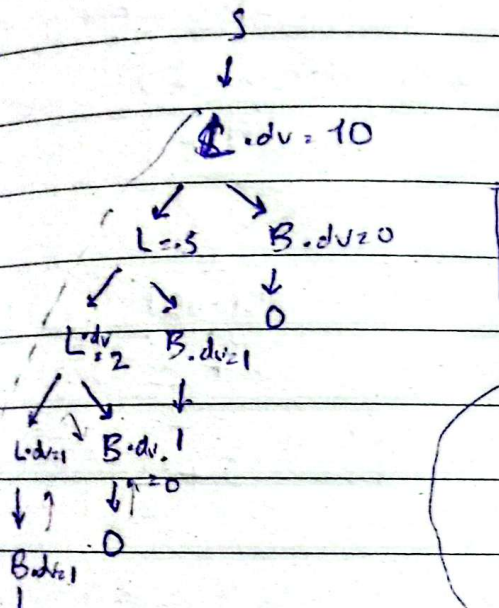
   - ↳ Information can flow downward or left to right
   - ↳ Ideal for top down parsing
   - ↳ The left to right restrictions ensures the dependency graph has no cycles
   - ↳ Attributes are evaluated by traversing parse tree depth first, left to right.

Date Day

• Binary to Decimal SDD

$S \to L$	{ S.dv = L.dv }
$L \to LB$	{ L.dv = 2 * L.dv + B.dv }
$L \to B$	{ L.dv = B.dv }
$B \to 0$	{ B.dv = 0 }
$B \to 1$	{ B.dv = 1 }

String = 1010 (10)



Topological sort:

1. Calculate indegree of all nodes
2. Then choose the node which has indegree = 0
3. Then delete that node from graph
4. Then re-calculate indegrees.
5. Repeat.

→ Agr multiple nodes ho with indegree = 0, choose any one.

• Evaluation Order For SDD's

↳ The evaluation order of an SDD (Eg is any sequence that computes after all their dependencies have been calculated.

↳ It is obtain from a topological sort of the dependency graph

Date _____ Day _____

→ Binary to Decimal with Fraction

$S \rightarrow L_1 \cdot L_2 \quad \{ S.dv = L_1.dv + \frac{L_2.dv}{2^{L_2.nb}} \}$

$L \rightarrow L \cdot B \quad \{ L.dv = 2 \times L.dv + B.dv \;\; , \;\; L.nb = L.nb + B.nb \}$

↳ no. of bits

$L \rightarrow B \quad \{ L.dv = B.dv \;\; , \;\; L.nb = B.nb \}$

$B \rightarrow 0 \quad \{ B.dv = 0 \;\; , \;\; B.nb = 1 \}$

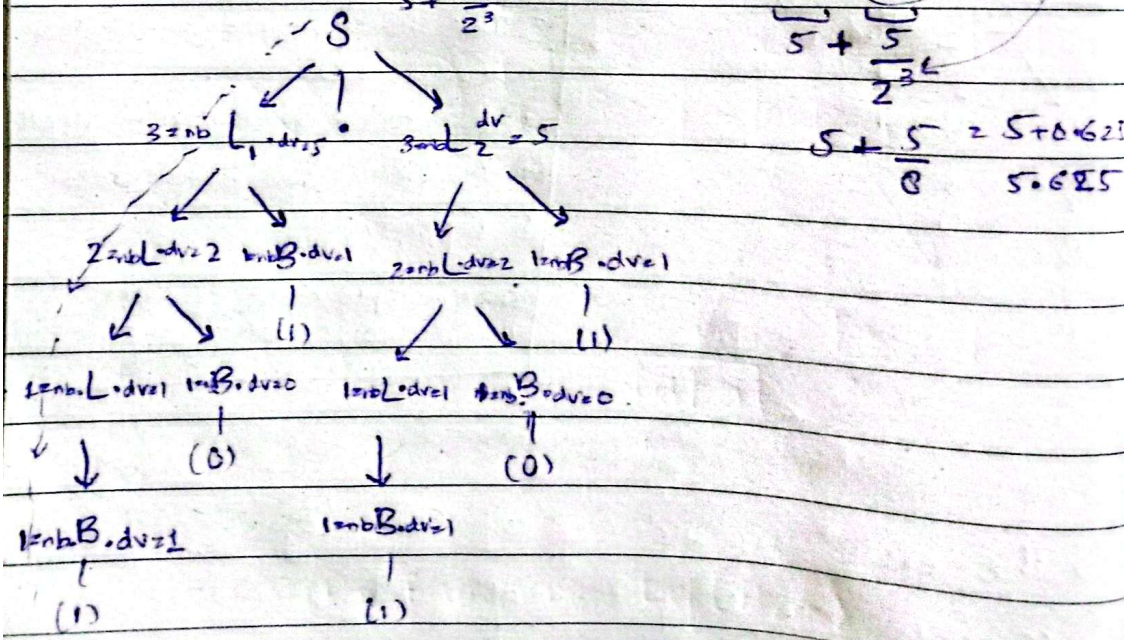
$B \rightarrow 1 \quad \{ B.dv = 1 \;\; , \;\; B.nb = 1 \}$

$5 + \frac{5}{2^3} = 5.625$

$101.101 \rightarrow nb = 3$

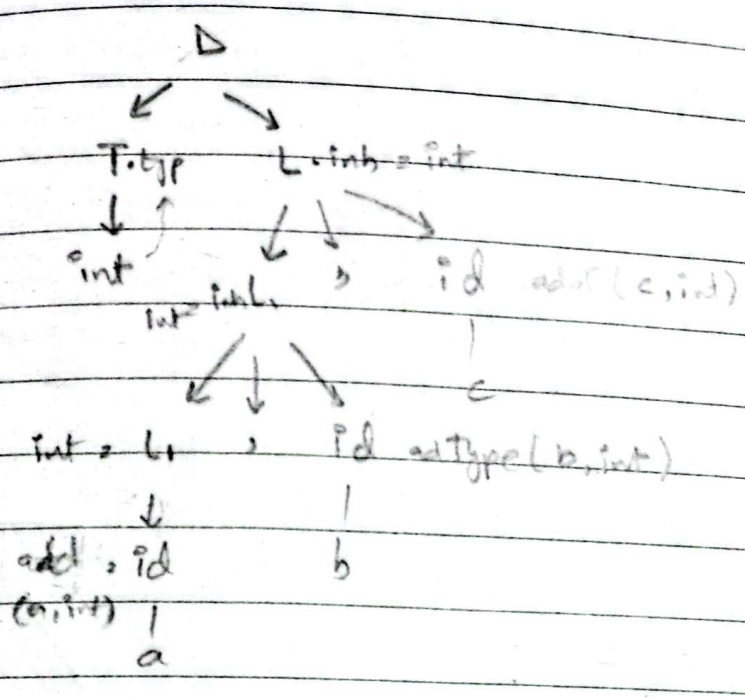
$\underbrace{101}_{5} . \underbrace{101}_{\frac{5}{2^3}}$

$5 + \frac{5}{8} = 5 + 0.625 = 5.625$



Date _____ Day _____

int a, b, c



• SYNTAX TREE CONSTRUCTION:

→ It is a tree representation of source code

→ where node → a variable, operator, numbe..)

→ leaf Node ⇒ variable names, number or other literal value

leaf(op, val)

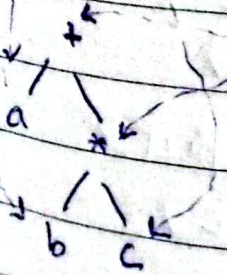
↳ A label indicating the type (eg id for identifier, num for numbers)

↳ exact value

→ Node (op, $c_1, c_2 .. c_n$)

↳ This used to create nodes for operations that combine other nodes

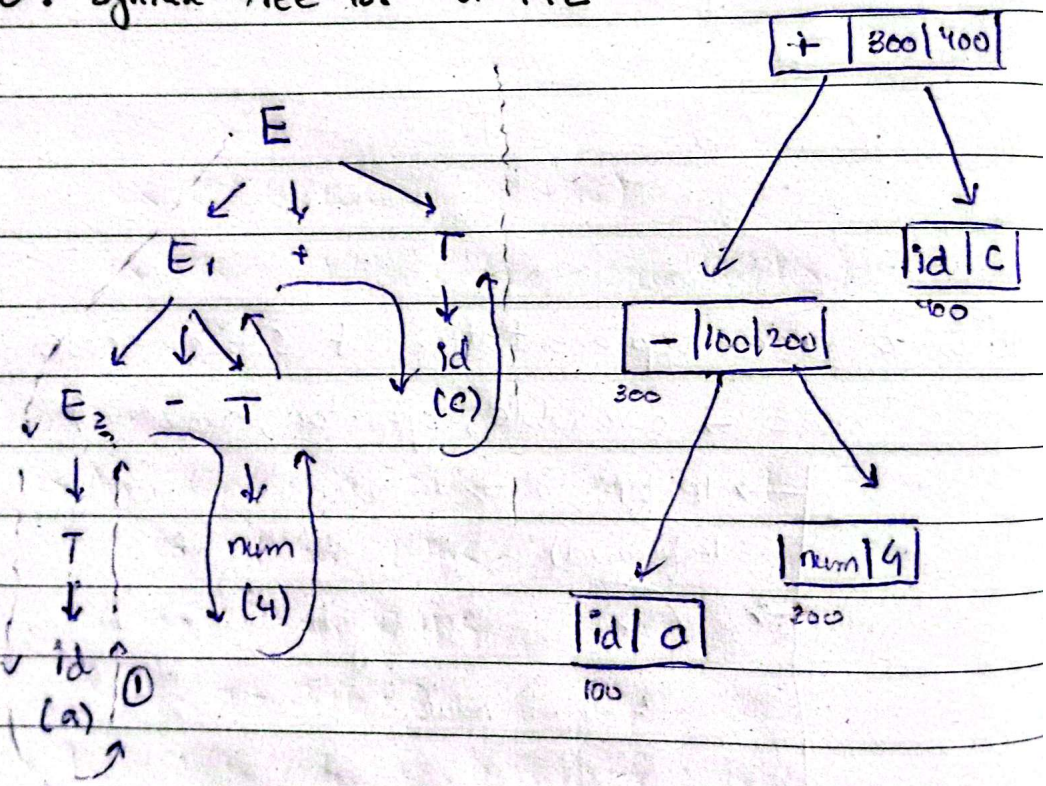
↳ It repr operations like +, -, *, /



Date _____ Day _____

1) $E \to E_1 + T$	E.node = new Node('+', E_1.node, T.node)
$E \to E_1 - T$	E.node = new Node('-', E_1.node, T.node)
$E \to T$	E.node = T.node
$T \to (E)$	T.node = E.node
$T \to id$	T.node = new Leaf(id, id.entry)
$T \to num$	T.node = new Leaf(~~id, id.entry~~ num, num.val)

Q: Syntax tree for a-4+c



(1) p_1 = T.node = new Leaf(id, a)

(2) p_2 = T.node = new Leaf(num, 4)

(3) p_3 = E.node = new Node(-, p_1, p_2)

(4) p_4 = T.node = new Leaf(id, c)

(5) p_5 = E.node = new Node('+', p_3, p_4)

Date Day

• INTERMEDIATE CODE GENERATION

→ In the analysis-synthesis model of a compiler, the frontend of a compiler translates a source program into an independent intermediate code

→ Backend of the compiler uses this intermediate code to generate the target code.

→ Why Inter. Code needed?

↳ If a src language is directly translated to its target machine code, then for each new machine, a full native compiler is required

↳ Inter. code eliminates this

↳ Analysis portion same for all compilers, only the second part of compiler is changed acc to target machine

→ Two types of Representation

① High Level IR

↳ Very close to the src language itself

↳ But for target machine optimization, it is less preferred

② Low Level IR

↳ Close to target machine

↳ Suitable for register & memory allocation

Date _____ Day _____

→ IC can be language specific (eg: Byte Code for Java) or language Independent (eg: three address code)

→ Intermediate Language Types

① Graphical IRs:
- ↳ Abstract Syntax Tree
- ↳ DAGs (Directed Acyclic Graph)
- ↳ Control Flow Graphs

② Linear IRs:
- ↳ Stack based (Postfix)
- ↳ Three Addr. Code (quadruples)

→ Three Address Code

↳ An expression can contain atmost 3 variable
- ↳ 1 on L.H.S
- ↳ 2 on R.H.S

$x = a + b$

↳ Three Addr code is linearized representation of a Syntax tree or DAG

↳ Common Three Address Instructions forms
- $x = y \text{ op } z$
- $x = \text{op } y$
- $x = y$
- if (x relop y) goto L1
- goto L (unconditional jump)
- param x call p return y
- param x1 param x2 call p,2 return y

Backpatching: It is a method used in single pass compiler generation of three-address code where the address for jump statements is temporarily left unspecified due to unknown labels; backpatching later fills these address with correct values once the target labels are specified.

Date _____ Day

- A[i] = x
- y = A[i]
- x = *p
- y = &x

Q: c=0
do
{
if (a<b) then
x++;
else
x--;
c++
} while (c<5)

c = 0
L1:
if (a<b) goto L2
T1 = x-1
x = T1
goto L3
L2:
T2 = x+1
x = T2
L3:
T3 = c+1
c = T3
if (c<5) goto L1:

Q: while (A<C and B>D) do
. if A=1 then C=C+1
else
while A<=D
do A=A+B

1 if (A<C) goto 3
2 goto
3 if (B>D) goto 5
4 goto
5 if A==1 goto
6
7 if A<=D goto 8
8 goto 10
A = A+B
9 goto 6
10

Page No.

Day

Date

```
1  if A<C goto 3
2  goto 13  → loop exit
3  if B>D goto 5
4  goto 13  → loop exit
5  if A=1 goto 11        → in loop
6  if A<=D goto 8
7  goto 1   → exit loop
8  A=A+B
9  goto 6
10 C=C+12   → exit else
11 goto 1
12
13
```

- Representation Types of TAC

① Quadruple

$a = -b * c + d$

TAC:

$t_1 = -b$

$t_2 = t_1 * c$

$t_3 = t_2 + d$

$a = t_3$

	Op	Operand1	Operand2	Result
(0)	-	b	∝	t_1
(1)	*	t_1	c	t_2
(2)	+	t_2	d	t_3
(3)	=	t_3	∝	a

Date _____ Day _____

→ This representation takes more space
→ Changing the order of independent instruction doesn't affect result

② Triple

↳ yaha par temp variable mein store krne k bajae direct memory reference krte hai.
↳ Space not wasted

	Op	Operand1	Operand2
(0)	-	b	
(1)	*	(0)	c
2	+	(1)	d
3	=	(2)	

of the stmts

↳ Changing the order affects the result

↳ 0 → + | (1) | d
1 → * | (0) | c
2 → - | b

changed the order,
isko andar -b ka answer aana chahiye tekin humne ordering change krdi to ab uske andar ghalat answer ajayega.

③ Indirect Tuples

↳ Yaha par hum ek list of pointers store krenge.
↳ The pointers will reference to each of the results of the stmts

	Stmt
100	(0)
101	(1)
102	(2)
103	(3)

↳ Here we can reorder the stmts.
↳ But it requires two memory references.

Page No.

Date _____ Day _____

Q: for (i=1 ; i<n ; i++)

x = a+b*c

condition reverse krdi

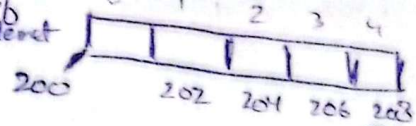
1) i = 1
2) if (i > n) goto 9
3) $t_1 = b * c$
4) $t_2 = a + t_1$
5) $x = t_2$
6) $t_3 = i + 1$
7) $i = t_3$
8) goto 2
9)

Q: switch (i)
{ case1:
$x_1 = a_1 + b_1$;
break;
case2:
$x_2 = a_2 + b_2$;
break;
default
$x_3 = a_3 + b_3$;
break;
}

1) if (i==1) goto 6
2) if (i==2) goto 9
3) $t_1 = a_3 + b_3$
4) $x_3 = t_1$
5) goto 11
6) $t_2 = a_1 + b_1$
7) $x_1 = t_2$
8) goto 11
9) $t_3 = a_2 + b_2$
10) $x_2 = t_2$
11)

Page No. ☐

A[i] = Base Addr + i × size of each element



Date Day

Q: int A[10], B[10]
int x=0, i;

For (i=0; i<10; i++)
x = x + A[i] * B[i]

1) x = 0
2) i = 0
3) t_1 = base Addr A
4) t_2 = base Addr B
5) if (i >= 10) goto 14
6) t_3 = i * 2
7) t_4 = t_1[t_3]
8) t_5 = t_2[t_3]
9) t_6 = t_4 * t_5
10) t_7 = x + t_6
11) x = t_7
12) i = i + 1
13) goto 5
14)

Q x = A[i][j]

t_1 = Addr of A
t_2 = 4 (no. of col)
t_3 = i * t_2
t_4 = t_3 * j
t_5 = t_4 * 2 → size of an element
x = t_1[t_5]

a_{ij} = Base + i × 2 × len + j × 2

A[i][j] = Base + (i × len + j) × 2
for int

Q2 int A[32][32][8]
i = A[i][j][k]
x = A[(i * 32 × 8 + j * 8 + k) × 2]

Date _____ Day _____

→ Directed Acyclic Graphs

↳ Similar to syntax tree but with shared common subexpressions

↳ Key Diff : Nodes can have multiple parents

DAG	Syntax Tree
→ Repr shares common subexpression	→ Replicates common subexpr
→ Single node represents multiple occurence	→ Each occurence gets its own subtree
→ Compact Representation	→ Inefficient Representation

$x = (((\underbrace{(a+a)}_{t_1} + \underbrace{(a+a)}_{t_2}) + \underbrace{((\underbrace{(a+a)}_{t_4} + \underbrace{(a+a)}_{t_5})}_{t_6})}_{t_3})$ t_7

$t_1 = a+a$

$t_2 = a+a$

$t_3 = t_1+t_2$

$t_4 = a+a$

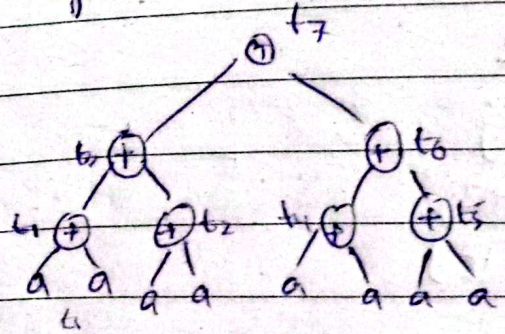
$t_5 = a+a$

$t_6 = t_4+t_5$

$t_7 = t_3+t_6$

$x = t_7$

Syntax Tree:

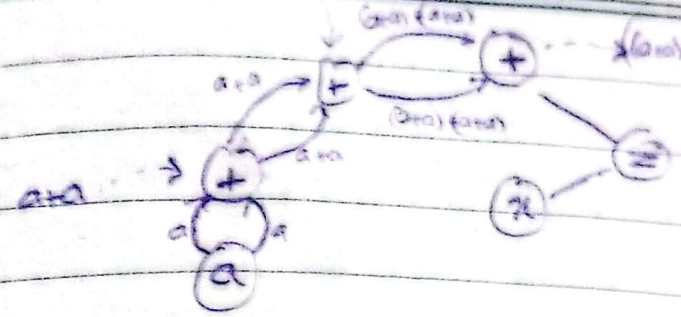


↳ Jab yaha "a" eighth times repeat horaha

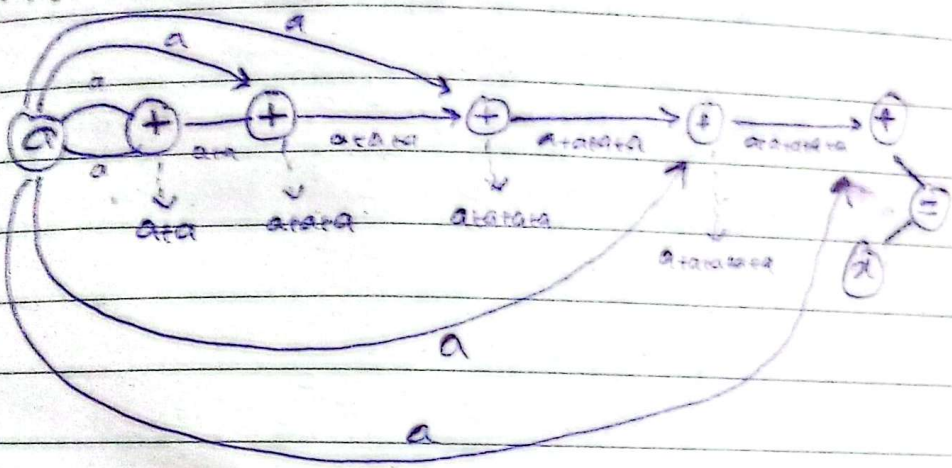
↳ DAG mein sirf ek dafa ayega

Date _____ Day _____

• DAGs :

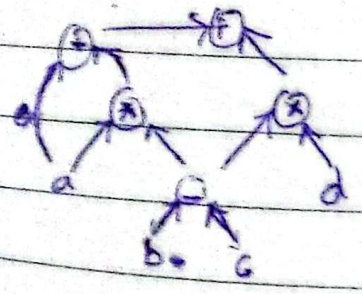


$x = a+a+a+a+a+a$



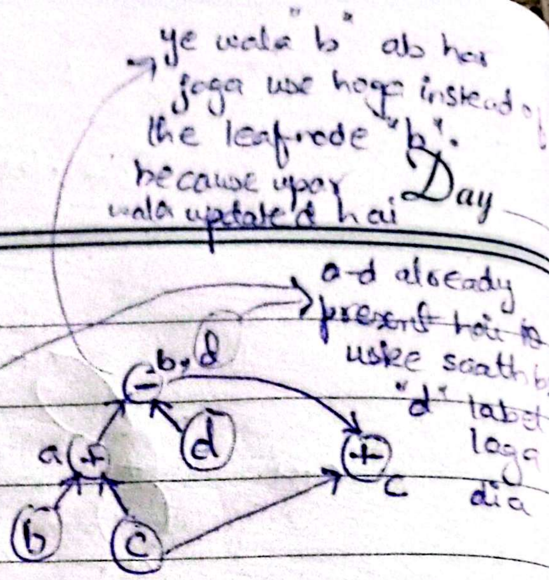
$x = a + a*(b-c) + (b-c)*d$

$t_1 = b-c$
$t_2 = a \times t_1$
$t_3 = d \times t_2$
$t_4 = t_2 + a$
$t_5 = t_4 + t_3$

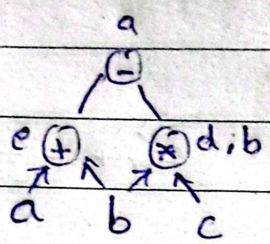


Date ______ Day ______

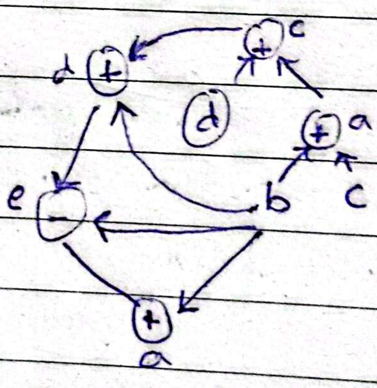
$a = b + c$
$b = a - d$
$c = b + c$
$d = a - d$



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



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



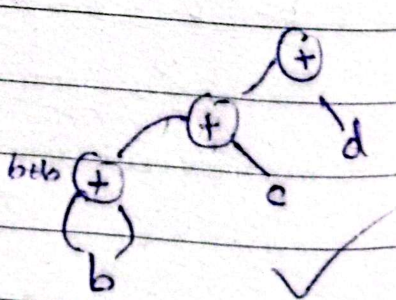
For minimum no. of nodes & edges we 1st simplify the equation by subsituting values.

$a = (d - b) + b$

$a = b + c$

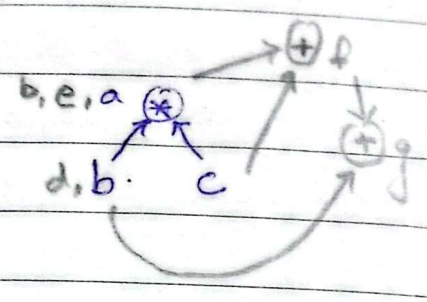
$a = b + a + d$

$a = b + b + c + d$

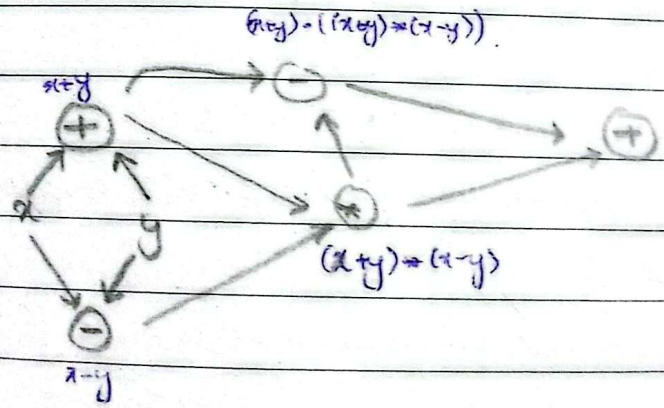


Date Day

a = b*c
d = b
e = d*c
b = e
f = b+c
g = f+d



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



→ Application Of DAG in Compiler

① Code Optimization — DAGs identify & eliminate redundant computations, also known as common sub expression, to make code more efficient

② Code generation — They assist in generating efficent machine code by representing intermediate code & helping with tasks like register allocation

Date _____ Day

• INTERMEDIATE CODE OPTIMIZATION

→ The phase of compiler tends to improve the code by making it consume fewer resources so that faster-running machine code will result.

→ The optimized code should always give same o/p on same i/p as before.

→ Compilation time must be kept reasonable

→ Should increase the speed & performance

→ Types of Code Optimization

① Machine Independent

↳ Performed on intermediate code before target code generation

↳ It improves program efficiency without consider the target machine architecture

↳ It does not involve CPU registers or absolute memory addresses

↳ Examples:

- Constant Folding
- common subexpression elimination
- dead code elimination
- loop optimization

Date _____ Day _____

② Machine-Dependent

↳ This optimization is applied after target code generation

↳ It is specific to the target machine architecture and uses CPU register & memory hierarchy

↳ It focuses on

↳ register allocation ↳ instruction selection

↳ instruction scheduling ↳ addressing modes

↳ Peephole optimization

↳ Redundant LOAD/STORE

↳ Flow of Control Optimization

- Loop Optimization

① First detect loop

② Loop detect krne k liye we do Control Flow Analysis on ~~Contro~~ Program Flow Graph

③ PFG k liye we need to find Basic Blocks

④ A basic block is a sequence of 3-Addr Code stmts where controls enters at the begining & leaves only at the end without any jump stmts

Date _____ Day _____

* Agr normally loop identify hosakta hai then ye sab lambi krne ki need nhi

→ Finding Basic Blocks

i- Inorder to find basic block we need to find "leaders" in TAC

ii- A basic block will start from one leader to the next leader, but wo next wala include nhi hoga.

iii- Finding leader

a) The 1st line of TAC is a leader

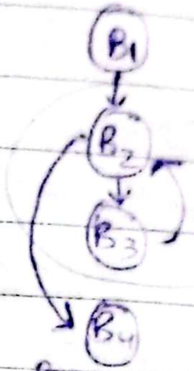
b) Any instruction that is the target of a conditional or unconditional jump is a leader like "if (i>10) goto (3)" → this 3rd line will be leader

c) Any instruction that immediately follows the conditional or ~~and~~ concadiontal jump is a leader.

```
2  if (i>10) goto 34
       c = c+2   → leader
       =
       :
34  a = b+d  → leader
```

Date Day

```
*1  f = 1            B1
 2  i = 2            → point (a)
*3  if (i>n) goto 8  B2   → point (b)
*4  t1 = f*i              → point (c)
 5  f = t1           B3
 6  i = i+1
 7  goto (3)
*8  goto calling fn  B4   → point (c)
```

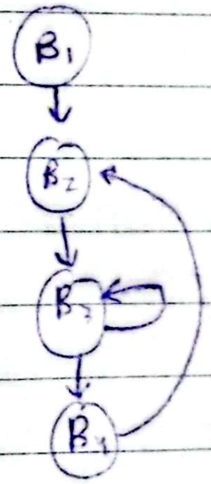


~~Program~~ ~~Control~~ Flow Graph

if there is a cycle means ~~there is a~~ loop

Q:

```
*1.  i = 11                B1
*2.  j = 1                 B2
*3.  t1 = 5 * i            B3
 4.  t2 = t1 + j
 5.  t3 = 4 * t2
 6.  t4 = t3
 7.  a[t4] = -1
 8.  j = j+1
 9.  if (j<=5) goto 3
*10. i = i+1               B4
 11. if (i<5) goto 2
```



Day

Date

→ Techniques of Loop Optimization

(1) Frequency Reduction (Code Motion)

↳ Agr loop mein koi esi stmt ho jiske agr loop se bahir krdein to koi farq na pare

eg:

```
i=0                          i=0
while (i<100){               t = a/b
    A = a/b + i      ⇒       while (i<100)
    i++                          A = t + i
}                                i++
                             }
```

a/b 100 baar evaluate hoga to isko ek baar evaluate krke value use krlenge

→ getting the same output with less no. of iteration

(2) Loop Unrooling

↳ Reducing the no. of times comparasion are made in a loop

↳ eg:

10 comparasions

```
for(i=0; i<10; i++)          for (i=0; i<5; i++)
    printf("Hi");      ⇒         Pf("Hi");
                                 Pf("Hi");
```

ya 5 printf likhdein & i = i+5 krdein ←

or

```
for (i=0; i<10; i+=2)
    Pf("Hi")
    Pf("Hi")
```

5 comparasions

Date _____ Day _____

② Loop Jamming

↳ Combining the bodies of two loops.

```
for (i=0; i<5; i++)
    a = i+5
for (i=0; i<5; i++)
    b = i+5
```

⇒

```
for (i=0; i<5; i++)
    a = i+5
    b = i+5
```

↳ both loop runs 5, 5 times ⇒ Reduce that to only 5
total 10 times

→ Optimization of Basic Blocks

① Constant Folding

↳ Replacing the value of expression before compilation is called as constant folding.

x = a + b + 2 * 3 + 4 → this will be computed at compile time

x = a + b + 10

② Constant Propagation

↳ Replacing the value of constant before compile time

x = 8 ⇒ x = 8

y = x/4 y = 8/4 ⇒ y = 2

```
int k=2
if (k) goto L3   ⇒   goto L3
```

condition will always be true

Date _____ Day _____

③ Copy Propagation

↳ if a variable is assigned the value of another variable, all subsequent uses of the copied variable are replaced with original variable as long as the original value doesnot change

eg:

```
x = y          ⇒    x = y
z = x + 1           z = y + 1
```

④ Common Subexpression Elimination

```
t1 = a+b            t1 = a+b
t2 = a+b       ⇒    t2 = t1 * d;
t2 = t2 * d;
```

⑤ Dead Code Elimination

↳ Remove stmts which never executes or they are unreachable

```
i = 0;                    i = 0
if (i == 1);        ⇒
    a = x+5;
```

will never gets executed

⑥ Strength Deduction

↳ Replacing the costly operator by cheaper operator

eg: `y = 2 * x  ⇒  y = x + x`

`B = A * 2` (left shift) ↳ `A << 1`

`y = x^2 = x * x`, `x/2 ⇒ x * 0.5`

Date _____ Day _____

⑦ Algebaic Simplification

eg: a = b*1 ⇒ a = b

c = d+0 c = d

⑧ Function Inlining

↳ Function call replaced by the body, so it saves a lot of time in copying all the params, storing the return addr etc.

```
int multiply(int a, int b){
    return a*b;
}
int main(){
    x = multiply(3, 4);
```

⇒

x = 3*4

→ Peephole Optimization → Machine Dependent

↳ Machine dependent optimization technique that examines a small window ("peephole") of consecutive target instructions & replaces inefficient with more efficent ones.

a) Redundant LOAD / STORE Elimination

a = b+c

d = a+e →

Day

Date

```
LOAD  R0, b      R0 = b
ADD   R0, c      R0 = b+c
STORE a, R0   }  x redundant   a = R0
LOAD  R0, a   }                R0 = a
ADD   R0, e      R0 = R0+e
STORE d, R0      d = R0
```

b) Flow Of Control Optimization

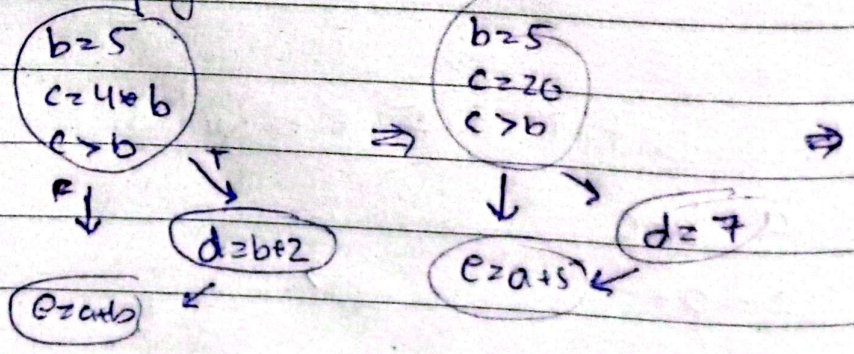
1) Avoid jumps on jumps

```
L1 : jump L2          L1 : jump L3
  =                     =
L2 : jump L3   ⇒   x extra jump
  =                     L3 : =
L3 : =
```

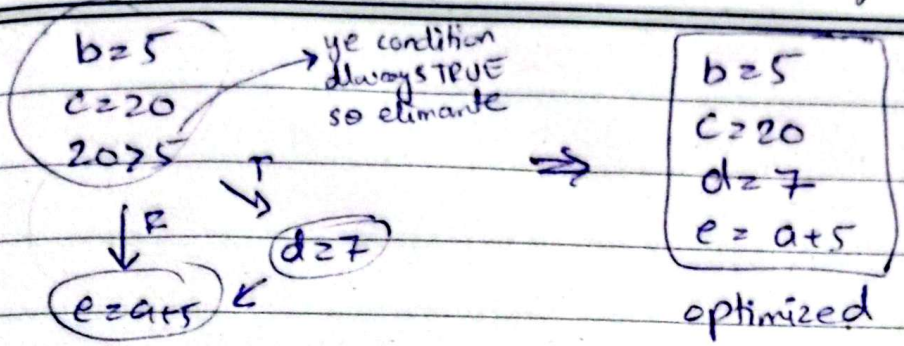
2) Eliminate Dead Code

Constant Propagation

Ex:



Date _____ Day _____



→ Matrix Multiplication Optimization

```
for (i=1 → n)        ⇒ A: elements are accessed across rows; spatial locality is exploited for cache
  for (j=1 → n)      ⇒ B: elements are accessed along columns, unless can hold hold all of B, cache will have problem
    for (k=1 → n) →
      C[i,j] = C[i,j] + A[i,k] + B[k,j]
```

↓

Single element accessed per loop

Optimized

```
for (i=1 → n)
  for (k=1 → n)      j ↓ ↑ k
    for (j=1 → n)
      C[i,j] = C[i,j] + A[i,k] + B[k,j]
```

A → Single element loaded for loop body

B → Elements are accessed in raw mjor

C → Extra loading storing, but accross raws