UNIT 4 Compiler Design

→ Intermediate rep

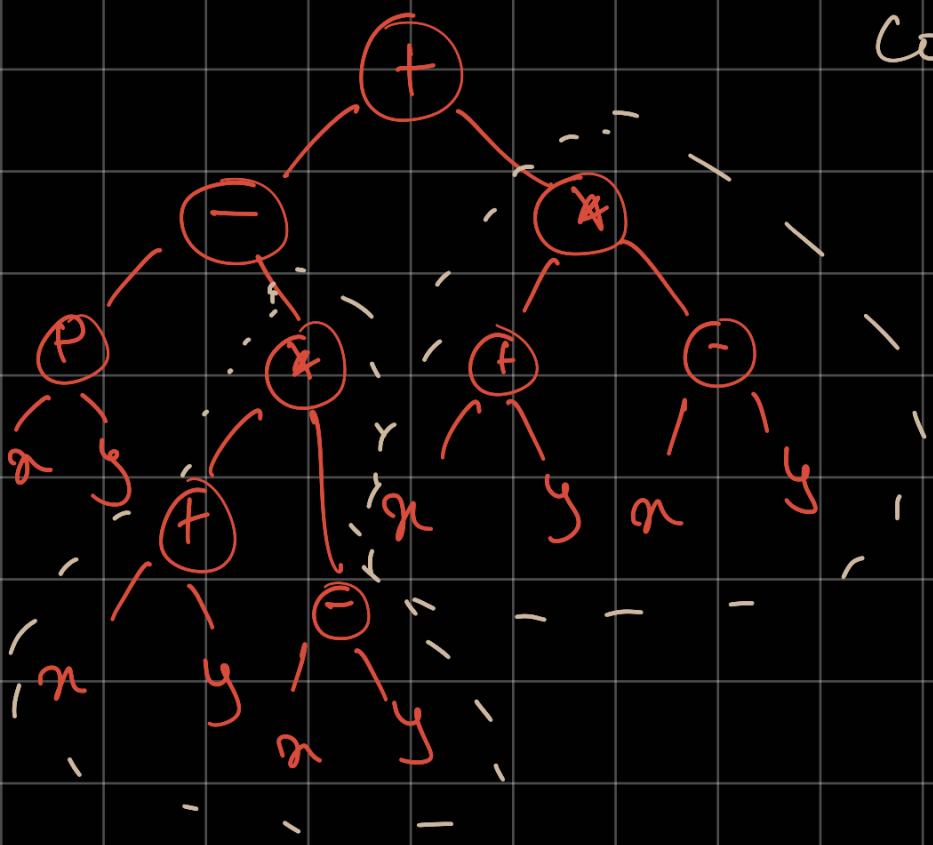
- helps # of backends to be built by retargeting & allows of machine independent code opt for better machine code.
- classification :- high level - low level
  - reps source code (Syntax trees, DAG. Java Byte Code)
  - closer to target (3AC, SSA)
  - language specific - language independent
  - Graphical - linear
- DAGs: Variant of Syntax tree unique node for each value. No cycles
  - Interior nodes are OPS
  - Exterior " " name/id/const
  - helps with opt CSE.
  - The same SDD can build DAG too.
  - Process of making DAG is costly

0) $E \to E+T \mid E-T \mid T$
$T \to T*F \mid T/F \mid F$
$F \to (E) \mid [E] \mid id$

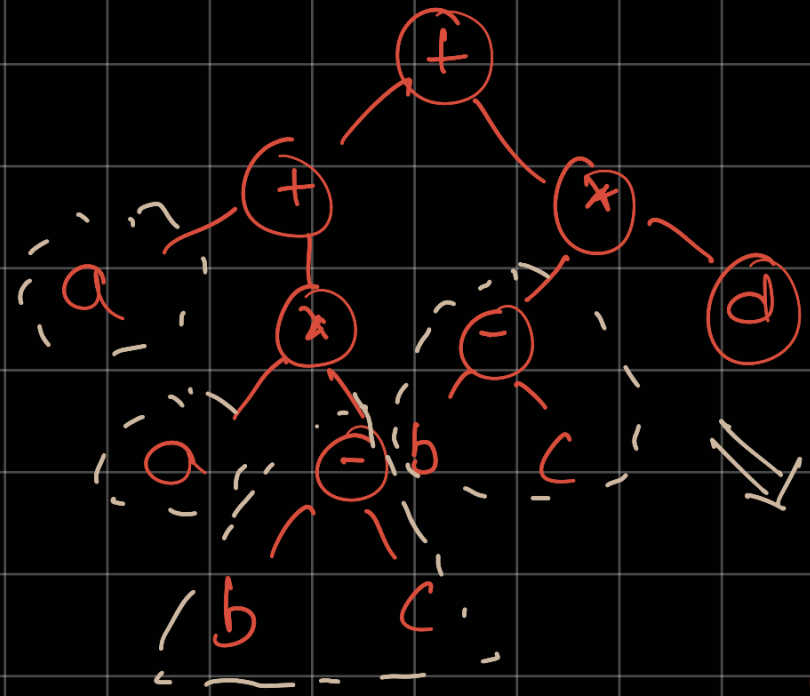
$((x+y)-((x+y)*(x-y)))+((x+y)*(x-y))$

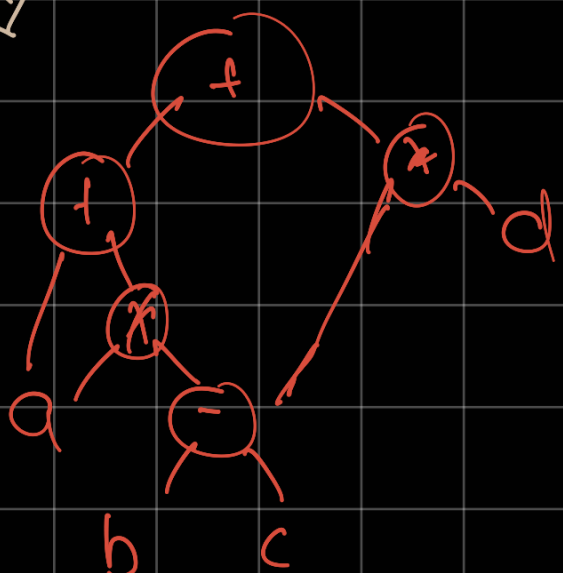
Draw DAG.

Common



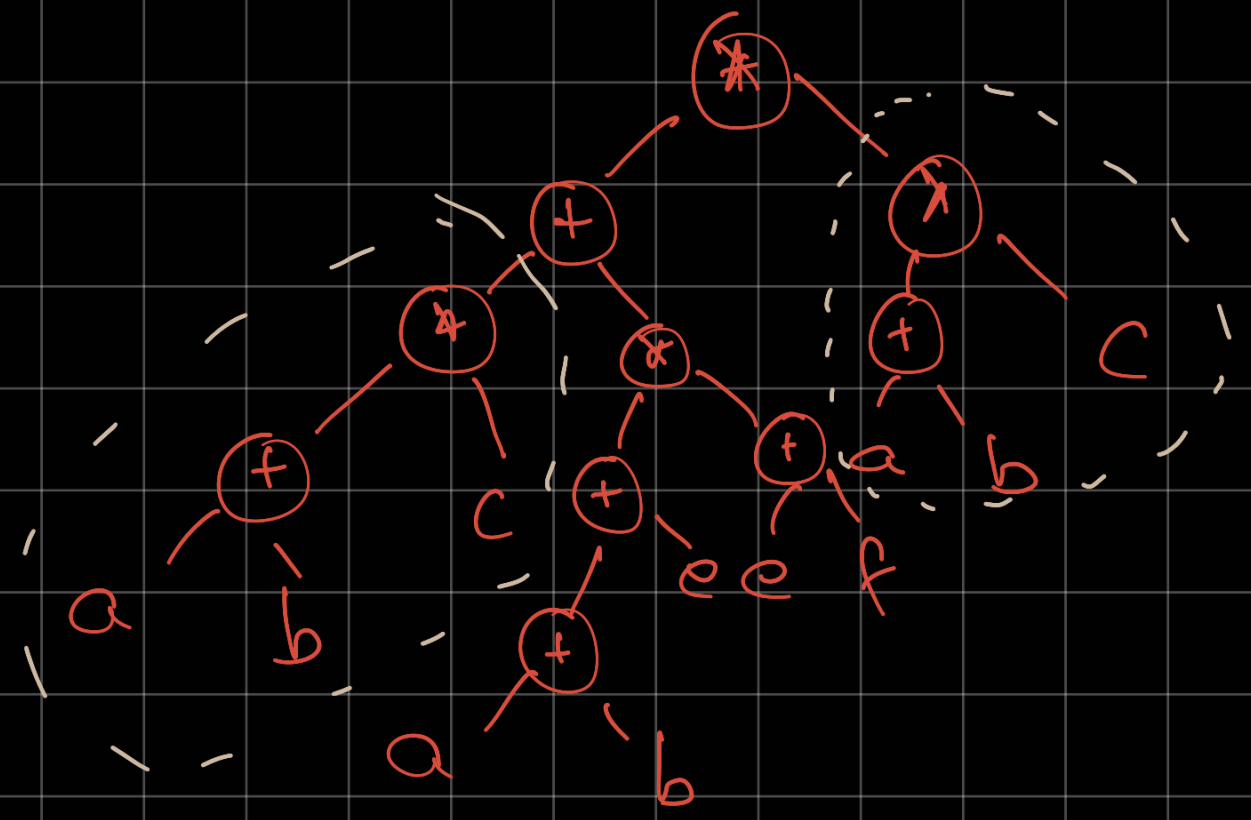
θ) $a + a * (b - c) + (b - c) * d$

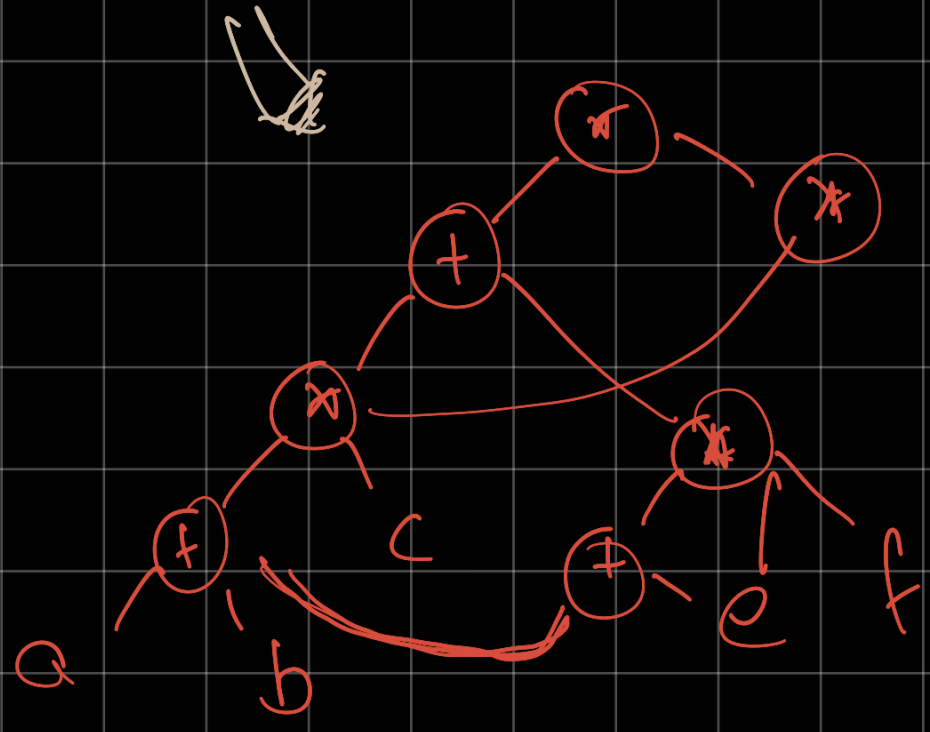




θ)

$[(a+b) * c + [(a+b) * c) * (c+f)] *$
$[(a+b) * c]$





3 AC

- Linearised form of DAG
  - At most one operator on RHS
  - Each ins upto 3 (not strictly)
  - Add can be name const or temps

•>

$x = y \text{ op } z$

$x = y \text{ op}$

$x = y$

goto L

$x = \&y$

$z = *x$

$*x = a$

$x[i] = a$

$y = x[i]$

if x goto L

ifFalse x goto L

if x relop y goto 2

if false x relop y goto L

Procedure Call:

Param a

Param b

...

call (foo, n) → # of parameters into function foo

Q) a + b * c - d || b * c

BODMAS

t1 = a || b

t2 = b * c

t3 = t1 * c

t4 = a + t2

t5 = t4 - t1

Q) x = foo (2 * x + 3, y + 10, g(i), h(3, j))

Param i

t4 = call (g, 1)

param 3

param j

t5 = call (j, 2)

t1 = 2 * x

t2 = t1 + 3

t3 = y + 10

Param t2

" t3

" t4

" t5

t6 = call (foo, 4)

x = t6

Q) alpha =
(65 <= C && C <= 90) ||
(97 <= C && C <= 122)

t1 = 65 <= C

t2 = C <= 90

if false t1 goto L1

if false t2 goto L1

L0 :
alpha = true

goto next

L1 :
t3 = 97 <= C

t4 = C <= 122

if false t3 goto L2

if false t4 goto L2

goto L0

L2 :
alpha = false

next :

a) `x = i + 10;`
`Switch (x) {`
`Case 1: x = x + i; break;`
`Case 2: x = 5;`
`Case 3: x = i;`
`default: x = 0;`
`}`

```
t1 = i + 10
x = t1
if x == 1 goto L1
goto L2
L1 :
    t2 = x * i
    x = t2
    goto next
L2:
    if x == 2 goto L5
    goto L3
L3:
    if x == 3 goto L6
    goto L4
L4:
    x = 0
```

(*) **Array addressing**

-> $A[i] = A + w(i - L_B)$

-) 2D arrays are trickier

2D arrays are given

Contiguous memory, 2-D array needs to be linearized. 2 ways of doing linearization

Row Major (default) Col Major

↳ $A[i][j] = A + w * [N(i - L_r) + (j - L_c)]$

$A(m \times n)$ ↳ # of cols of matrix

Col major

$A[i][j] = A + w * [(i - L_r) + M(j - L_c)]$

$A(m \times n)$ ↳ # of rows

M N

-> if C is 5x5 array: Assume row major

$C[i][j] = C + 4[5(i - 0) + (j - 0)]$

$= C + 4[5 * i + j]$

Representing 3AC

- 3AC can be represented using DS such as:
  - Quadruples (4 fields)
  - Triples (3 field)
  - Indirect triples (Triples ptr to triples)

- Quadruples:

Op	Arg1	Arg 2	result

(ptr to Symbol table)

- Problem is that in this we need insert temps into Symbol table as well.

x = -y	-	y	null	x
param x	param	x	null	null
Call foo, 3	call	foo	3	null
x = Call foo, 3	call	foo	3	x
if x goto L	if	x	null	L
goto L	goto	null	null	L
L1:	Label	"	"	L1
x[i] = y	[]=	x	i	y
x = y[i]	=[]	y	i	x

Array base | index | whatever else

- Triples:

Op	Arg1	Arg2

- No temps in Symbol table, instead you can reference the temp using serial # of instr computing the temp. This means the code cannot be moved around

$x[i] = y$

0	[]=	x	i
1	=	0	y

$x = y[i]$

0	=[]	y	i
1	=	x	0

- **Indirect triples**
  - A seperate list of pointers is maintained. This allow the statement to be reorderd.
  - Speed/easiness is same as Quad but lesser speed
  - If any change to code just change the ptr table & code can remain unchanges

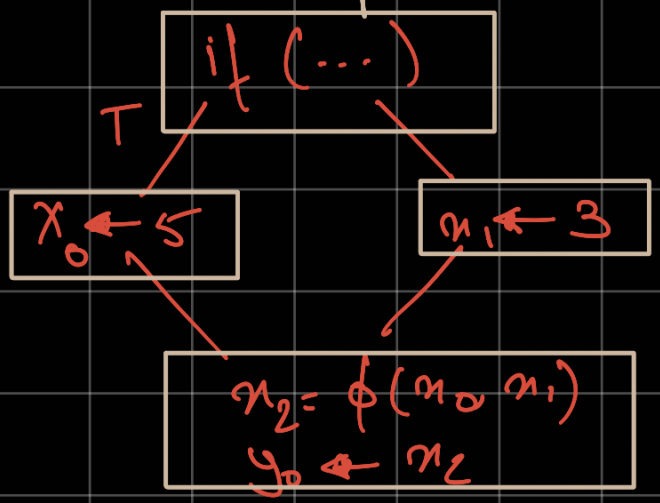
SET Slides for examples

→ **SSA: Single Static Assignment**

- Each var is assigned exactly once. But can be used many times
- Existing vars is split into versions in the IR using subscripts
- Control flow can't be known before hand. hence we have ϕ as a decision maker.

This is needed as after branches we need to decide which version of variable is active then

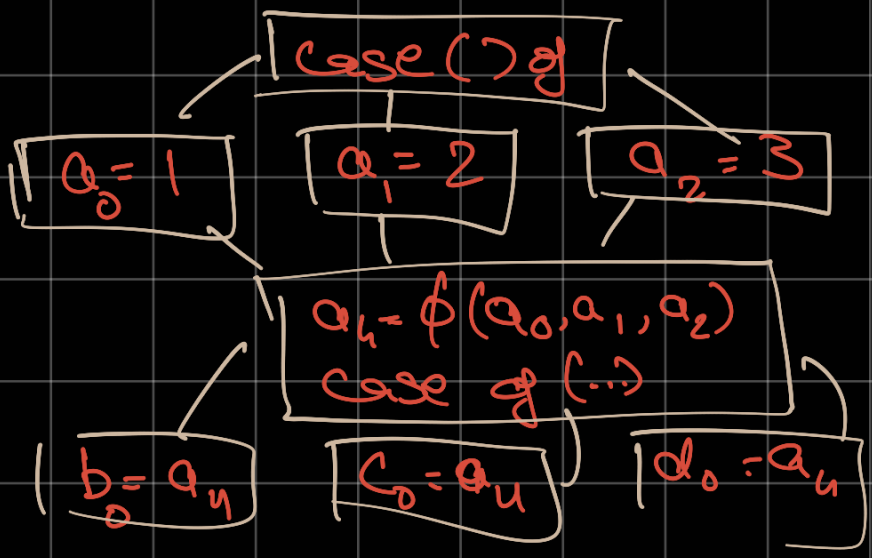
• ex 1:



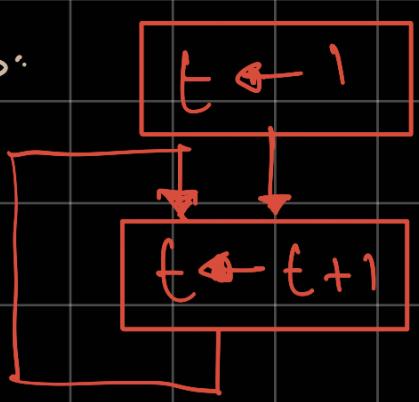
Examples cover all common cases of split joins & ϕ occurences

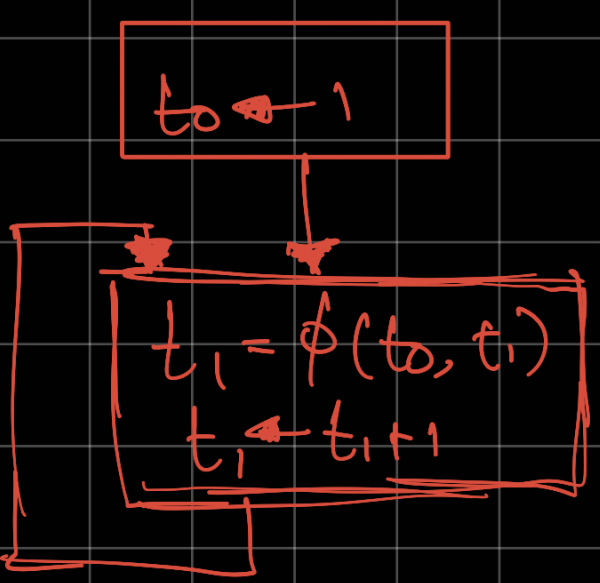
•) ex 2

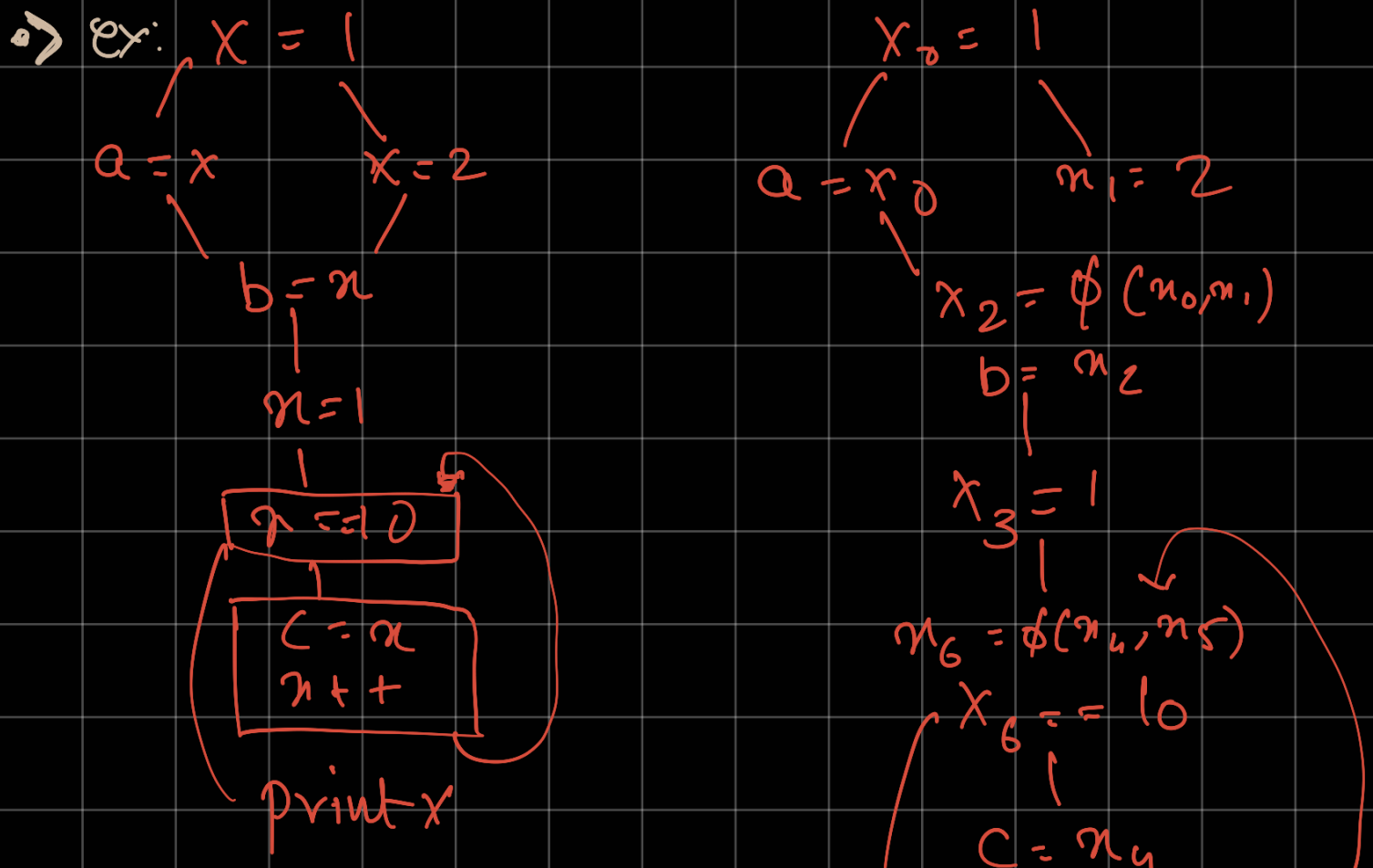
```
case(...) of
  0: a = 1
  1: a = 2
  2: a = 3
case(...) of
  0: b = a
  1: c = a
  2: d = a
```



•) ex 3:







> Note: Make sure to upate version on every assignment.

ex:

```
i = 1
j = 1
k = 0
if k < 100
    return j
if j < 20
    j = i
    k = k + 1
j = k
k = k + 2
```

```
i1 = 1
j1 = 1
k1 = 0
k2 = φ(k1, k3, k4)
j2 = φ(j1, j3, j4)
if k2 < 100
    return j2
if j2 < 20
    j3 = 1
    k3 = k2 + 1
j4 = k2
k4 = k2 + 2
```

→ Control flow graph (CFG)

- helps with MI opt. One block of CFG is code that is executed without any branches
- to build CFG identify leaders:
  - Any ins that is the target of a jump
  - Any " " follow branch ins
  - First ins of code is a leader

each block starts from the leader & has all ins till next leader (not included)

(included)

questions can even further asking to apply SSA on CFG.

a)
```
int add(n,k){
    S=0;
    a=4;
    i=0
    if (k==0)
        b=1
    else
        b=2
    while(i<n){
        S=S+a*b
        i=i+1
    }
    return S;
}
```

```
S=0;          leader
a=4;
i=0;
if False k==0 goto L1
b=1           leader
goto L2
L1:           leader
  b=2
  goto L2
L2: if False i<n goto L3   leader
  t1=a*b      leader
  t2=S+t1
  S=t2
  t3=i+1
  i=t3
  goto L2
L3: return S  leader
```

a) do {
i = i+1
} while(a[i]<v)

⇒

L1: t1 = i+1
i = t1
t2 = i*4
t3 = a[t2]
if t3<v goto L1

doesn't.

Code optimisation

•) A good optimiser must :
- be Semantic preserving
- be fast during compilation
- Speed up program on avg.

•) types of opt :
Control flow analysis: loop opts
Data flow

> Opt is best when programmer does sensible choices

•) opt — MI ← IR i/p
MD
BASED on CFS → global
Local (per block)
Machine code i/p

8 MI opt we are going to see:

1) Constant folding
- Computing constants at compile time.
- even algebraic expressions like: 0*n = 0 & 1*n = n are folded
- Concatenation of strings can be folded if constant

ii) Const propagation

- Substituting value of vars whose const value is known at compile time.

ex: int a = 10; int b = 45 - a/2; return b*(200/a+2);

Propagate a:

int a = 10; int b = 45 - 10/2; return b*(200/10+2);

fold b:

int a = 10; int b = 40; return b*(200/10+2);

Propagate b:

int a = 10; int b = 40; return 40*(200/10+2);

fold result:

int a = 10; int b = 40; return 880;

dead code elim:

return 880;

iii) CSE: Constant Sub exp elim.

iv) Copy propagation

- replacing occurences of targets of direct assignment with value

- Say v = v, we replace all v's with v's. often used as a clean up step

v) dead code elimination (DCE)

- code that is never executed

- does not affect further exec..

The above 5 are semantically preserving

The above is mostly dealt with data flow.

vi) Strength reduction

- replacing costly ops with cheaper ones.

x^2	x * x
x*2	x+x
x*2	x<<1
x/2	x*0.5
x/2	x>>1

vii) Packing temp

- replace distinct temps with 1 temp value if they are not needed to be alive together.

t1 = a+a	t1 = a+a
t2 = t1+b	t1 = t1+b
c = t2 * t2	c = t1 * t1

viii) Loop optimisation (IMPORTANT)

- play imp role in comp perf & making effective use of processing capabilities

•) loop invariant

var whose value is constant inside a loop can be placed outside.

•) Code motion:

Any modification that leads to decreased code in loop:

```
while(i <= limit-2){   =>   t = limit-2
                            while(i<t){
}                           }
```

- loop unwinding / unravelling

reducing # of iterations by replicating body of loop.

We could completely remove loop, but this requires us to know # of iters at compile time

- induction variable.

A variable that changes by a fixed amount in every iteration.

A var whose value is controlled by loop control var

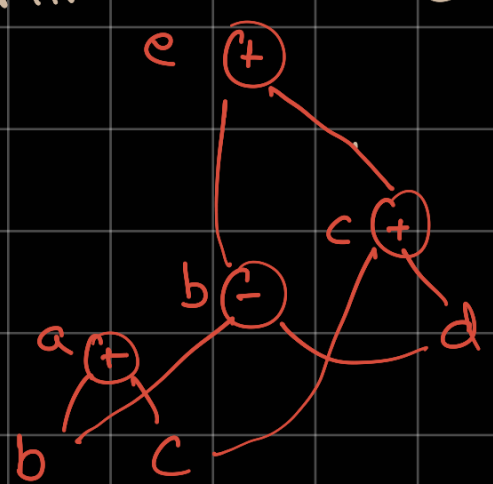
```
for(i=0; i<10; i+=2){
    x = i+3
    a[i] = y-x;
}
```

```
x = -6
for(i=0; i<10; i+=2){
    x = x+6;
    a[i] = y-x;
}
```

→ Local Optimisation (we are looking at control flow opts)

- The same opt can applied within the block

a) `a = b+c`
`b = b-d`
`c = c+d`
`e = b+c`



NO CSE

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

a) a = 10
b = 4 * a
c = i * j + b
d = 15 + a * c
e = i
c = e + j + i * a

TAC:
a = 10
t1 = 4 * a
t2 = i * j
t3 = t2 + b
c = t3
t4 = 15 + a
t5 = t4 * c
d = t5
e = i
t6 = e + j
t7 = i + a
t8 = t6 * t7
c = t8

This entirely is one block, hence DAG can be built & optimised

→ Code opt on CFG

See Slide bro, only examples

→ Next use algorithm:

ins	sym table live	sym table next use	i ins info live	i ins info next use

→ live var analysis

•) What is live analysis:

for a var x at point p, determine if x can be used along some path starting at p

liveness of a var is dependent / base on paths & not nodes

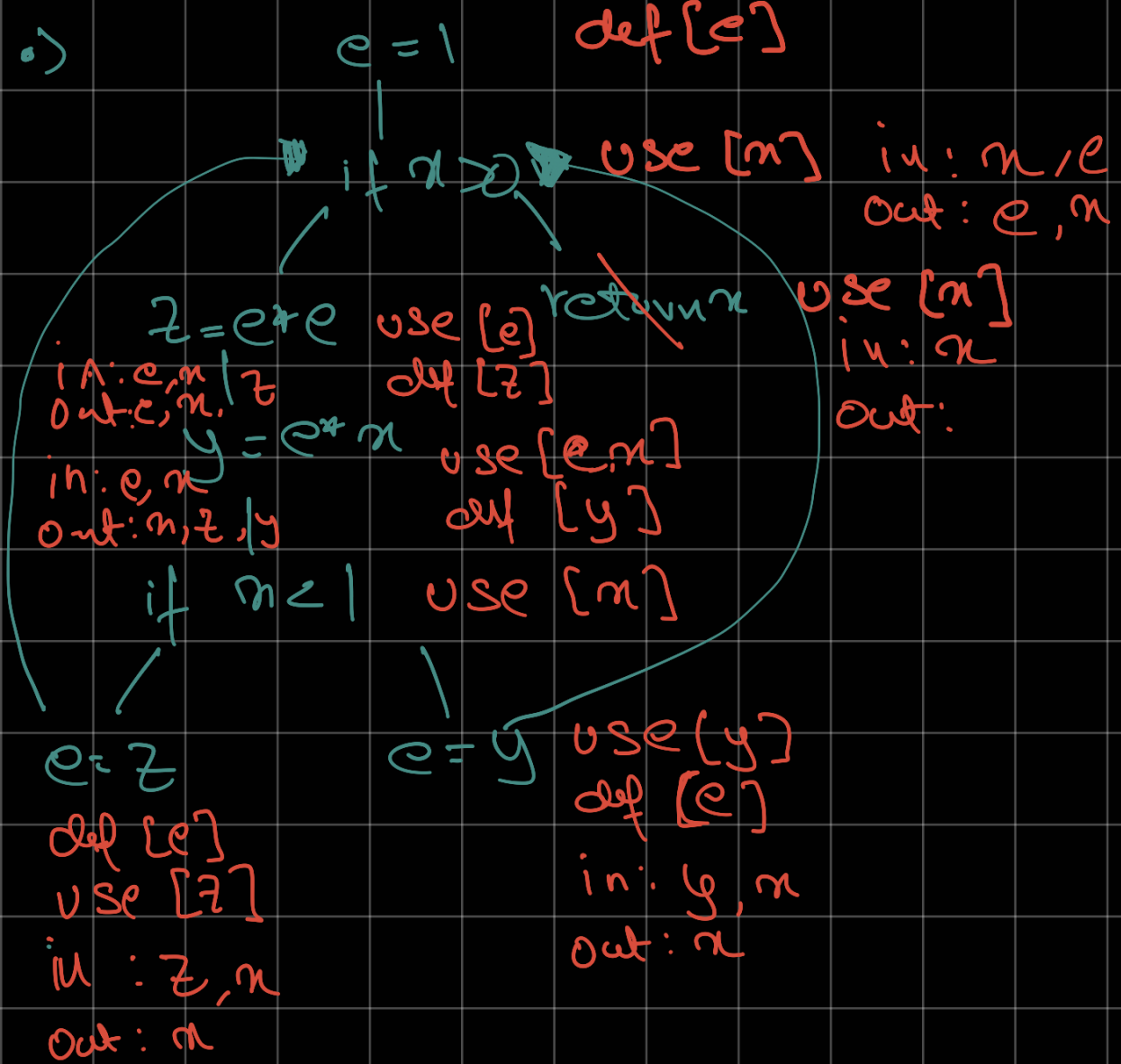
if not live, we can reuse register

-> for live analysis, you need the following:

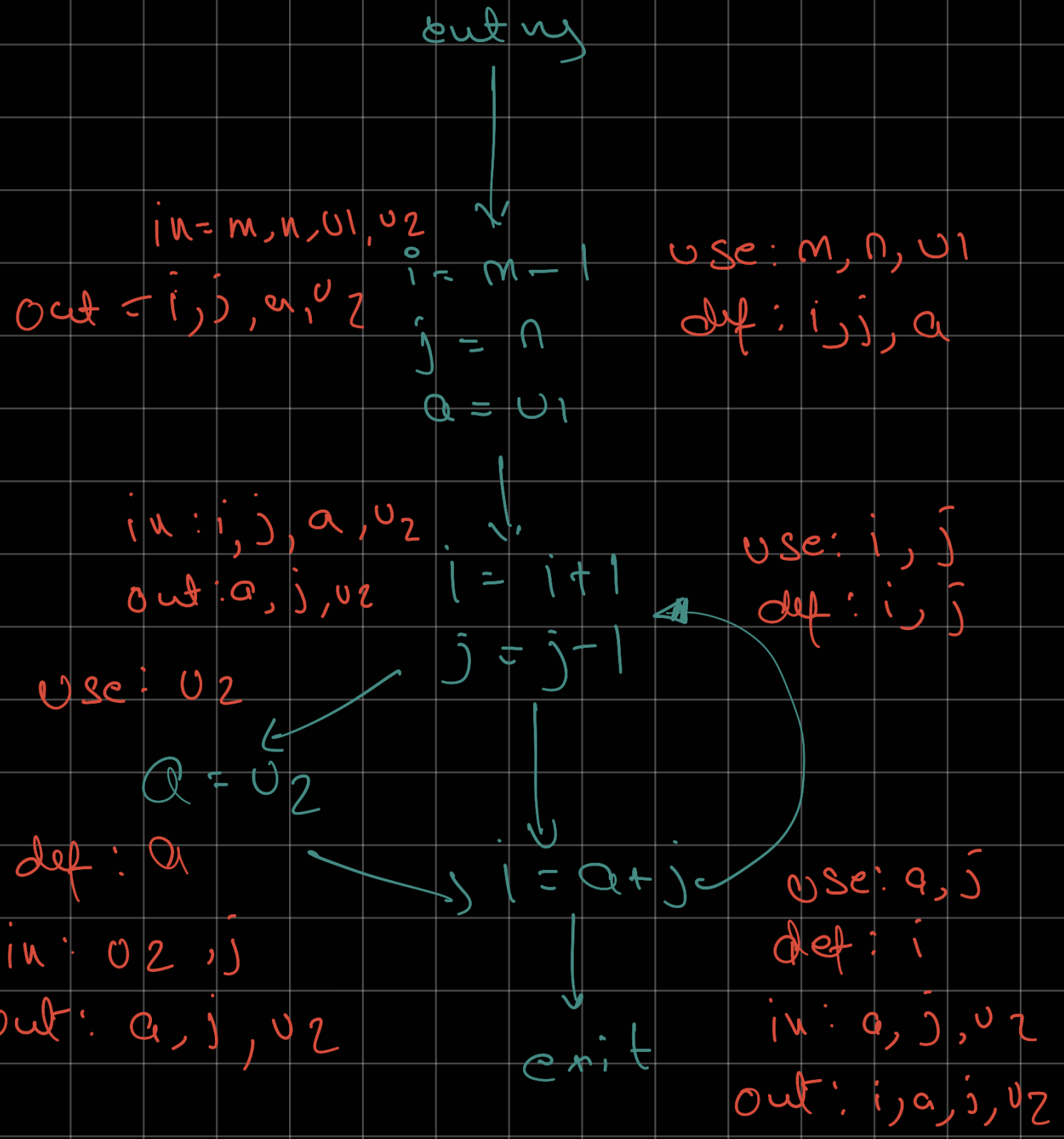
use[n] : vars used by node n
def[n] : " def " " "
in[n] : vars live on entry to node n
out[n] : " " " exit from " "

out[B] = ∪ of all IN[S] where
S is a successor of B
IN[B] = use[B] ∪ (out[B] - def[B])

->



a)


entry
in = m, n, v1, v2
out = i, j, a, v2
i = m - 1
j = n
a = v1
use: m, n, v1
def: i, j, a
in: i, j, a, v2
out: a, j, v2
i = i + 1
j = j - 1
use: i, j
def: i, j
use: v2
a = v2
def: a
in: v2, j
out: a, j, v2
l = a + j
use: a, j
def: l
in: a, j, v2
out: l, a, j, v2
exit