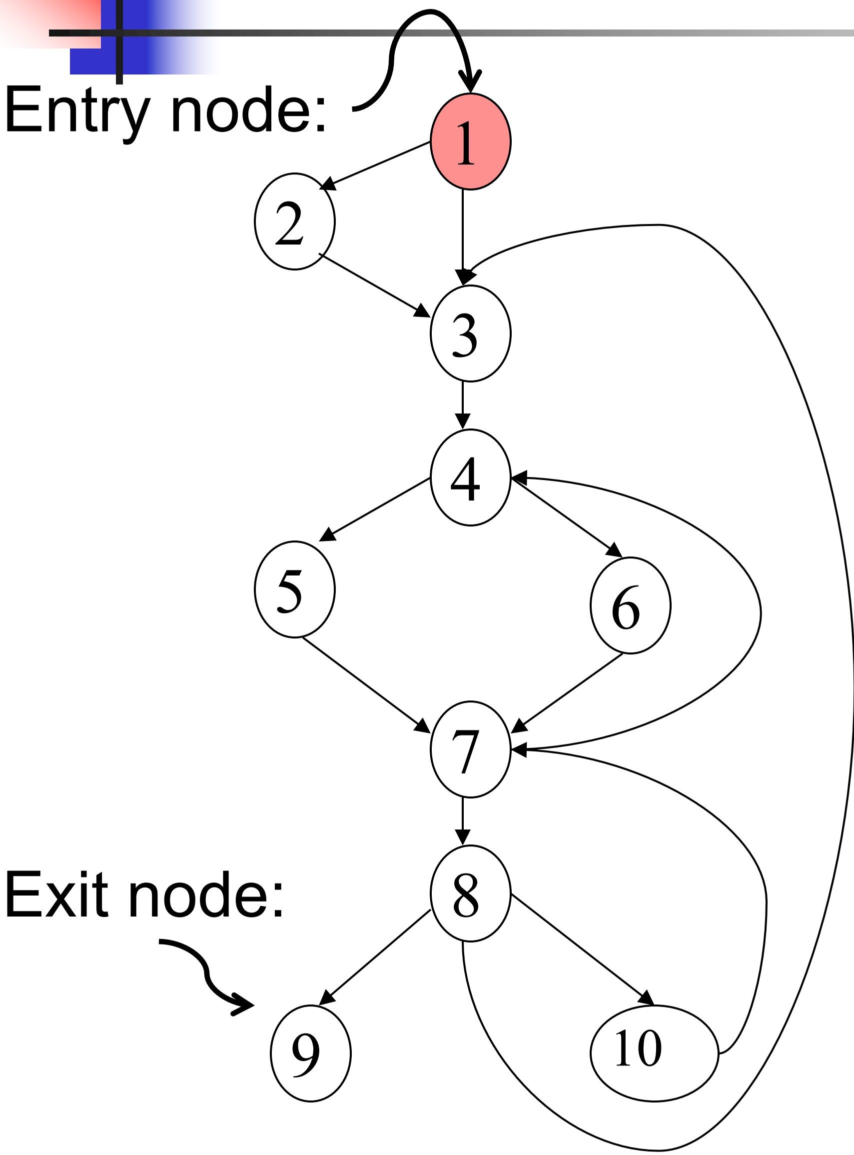
Dataflow Analysis: Dataflow Frameworks

Outline of Today's Class

- Catch up, four classical dataflow problems
- Dataflow frameworks
  - Lattices
  - Transfer functions
  - Worklist algorithm

- Reading:
  - Dragon Book, Chapter 9.2 and 9.3

Dataflow Analysis



1. Control-flow graph (CFG):
   - G = (N, E, 1)
   - Nodes are basic blocks
2. Data
3. Dataflow equations

out(j) = (in(j) – kill(j)) U gen(j)

(gen and kill are parameters)

4. Merge operator V

in(j) = V out(i)

i is predecessor of j

Problem 1. Reaching Definitions (*Reach*)

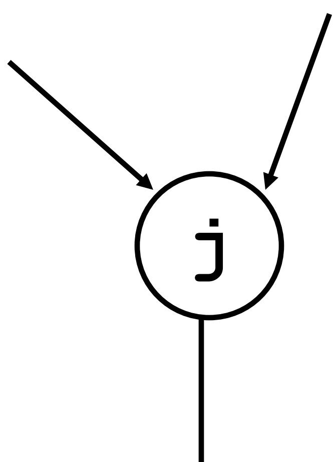
- Problem statement: for each CFG node **`n`**, compute the set of definitions **`(x,k)`** that reach **`n`**

- First, define data (i.e., the dataflow facts) to propagate
  - Primitive dataflow facts are definitions **`(x,k)`**
  - *Reach* propagates sets of definitions, e.g., **`{(i,1),(p,4)}`**

Reaching Definitions (*Reach*)

- Next, define the dataflow equations (i.e., effect of code at node `j` on incoming dataflow facts)

`j: x = y+z` $\left.\begin{array}{l} \\ \\ \end{array}\right\}$ kill(j): all definitions of `(x,_)`
gen(j): this definition of `x`, `(x,j)`

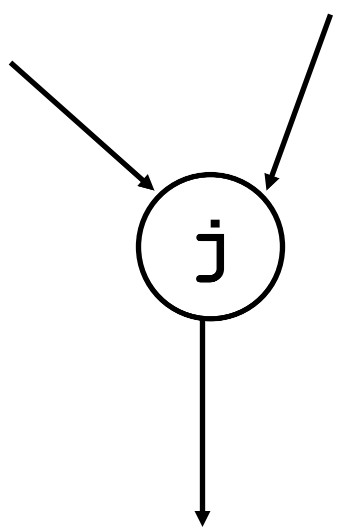


out(j) = (in(j) – kill(j)) U gen(j)

E.g., if in(4) = `{(x,1),(y,2),(x,3)}`
Node 4 is: `x = y+z`
Then out(4) = `{(y,2),(x,4)}`

Reaching Definitions (*Reach*)

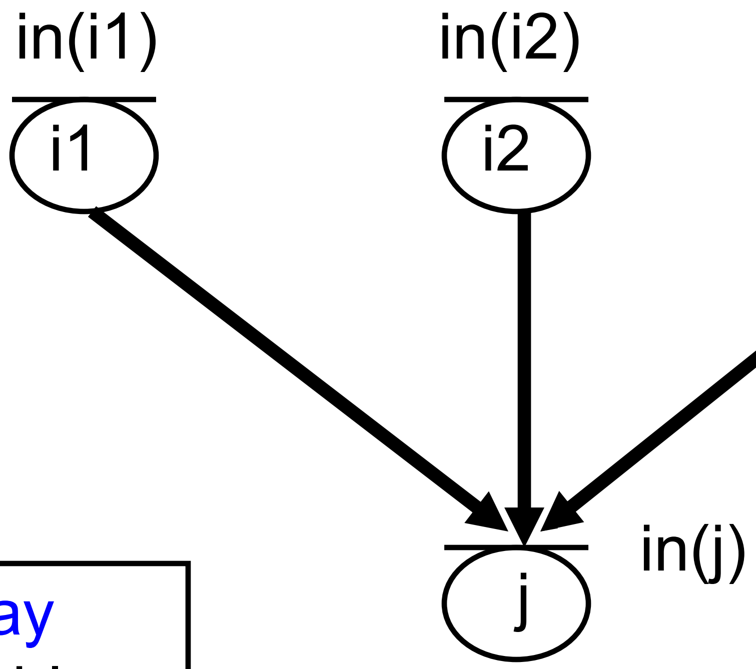
- Next, define the merge operator V (i.e., how to combine data from incoming paths)
- For *Reach*, V is the set union U



in(j) = { U out(i) | i is predecessor of j }

E.g., if out(2) = `{(x,1),(y,2)}` and
out(3) = `{(x,3)}` and
2 and 3 are predecessors of 4
in(4) = `{(x,1),(x,3),(y,2)}`

Reaching Definitions



Forward, may
dataflow problem

Problem 2. Live Uses of Variables (*Live*)

- We say that a variable **`x`** is "live on exit from node **`j`**" if there is a live use of **`x`** on exit from **`j`** (recall the definition of "live use of **`x`** on exit from **`j`**")
- Problem statement: for each node **`n`**, compute the set of variables that are live on exit from **`n`**

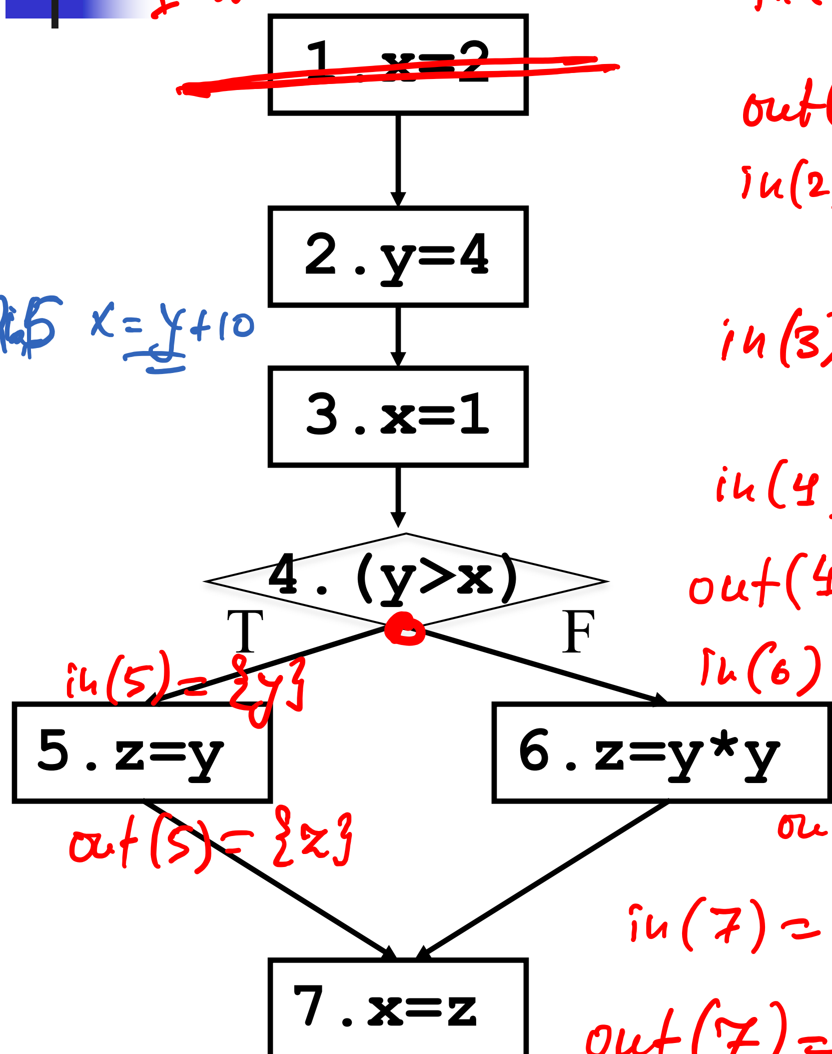
1. x=2; 2. y=4; 3. x=1; if (y>x) then 5. z=y; else 6. z=y*y; 7. x=z;

What variables are live on exit from statement 3? Statement 1?

Live Example

~~in(1) = {e}~~

1 Dead code



x = y+10

in(1) = {}

out(1) = {}

in(2) = {}

in(3) = {y}

in(4) = {y, x}

out(4) = {y}

in(5) = {y}

in(6) = {y}

out(5) = {z}

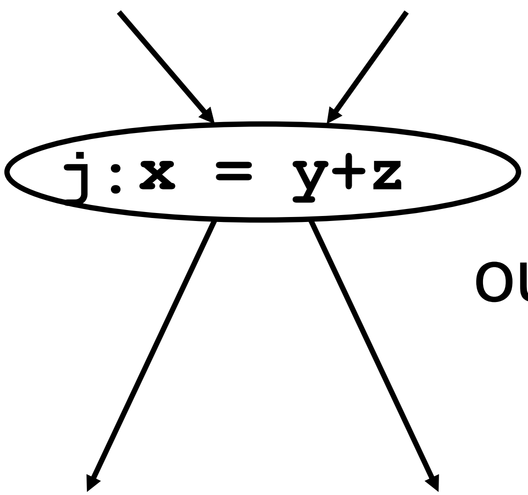
out(6) = {z}

in(7) = {z}

out(7) = {}

Live Uses of Variables (*Live*)

- Problem statement: for each node **n**, compute the set of variables that are live on exit from **n**



$in_{LV}(j) = (out_{LV}(j) - kill_{LV}(j)) \cup gen_{LV}(j)$

$out_{LV}(j) = \{ \cup\, in_{LV}(i) \mid i \text{ is a successor of } j \}$

Q: What are the primitive dataflow facts?
Q: What is $gen_{LV}(j)$?
Q: What is $kill_{LV}(j)$?

Live Uses of Variables (*Live*)

- Data
  - Primitive facts: variables **x**
  - Propagates sets: **{x,y,z}**

$j: \quad x = x + y$

$kill_{LV}(j) = \{x\}$

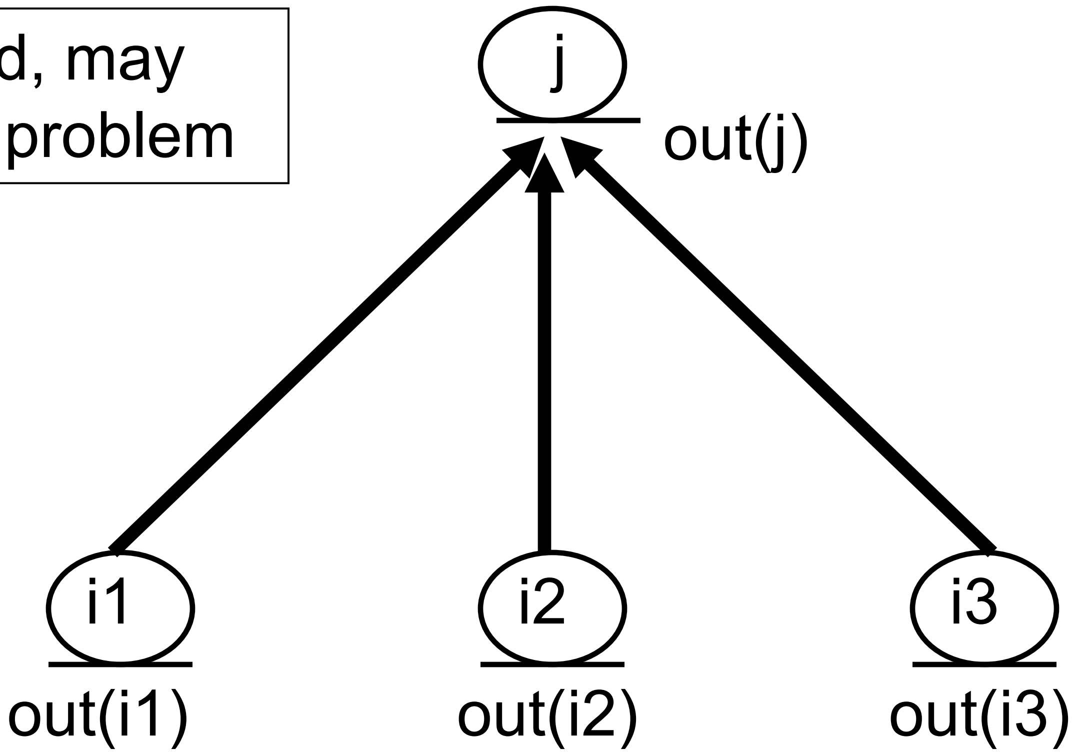
$gen_{LV}(j) = \{x, y\}$

- Dataflow equations. At **j: x = y+z**
  - $kill_{LV}$(j): **{x}**
  - gen_{LV}(j): **{y,z}**

- Merge operator: set union U

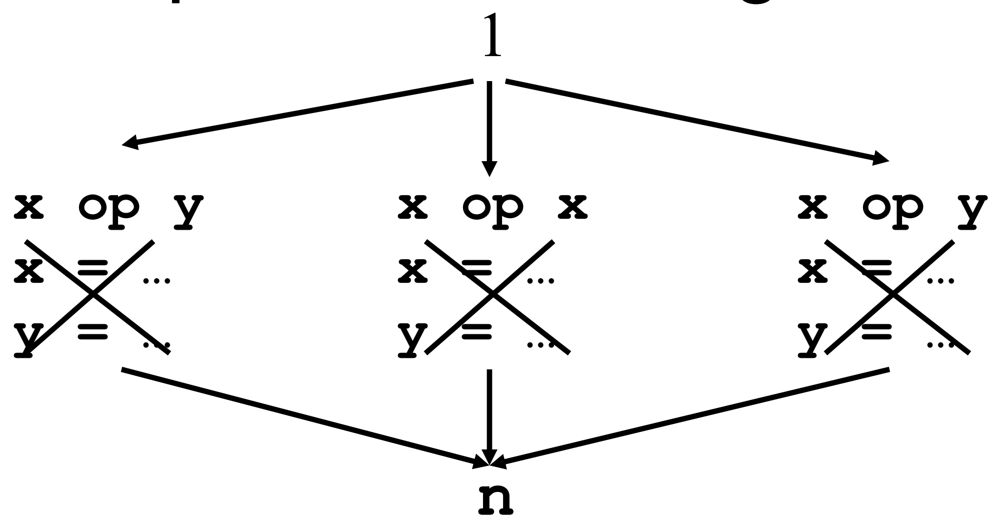
Live Uses of Variables

Backward, may
dataflow problem



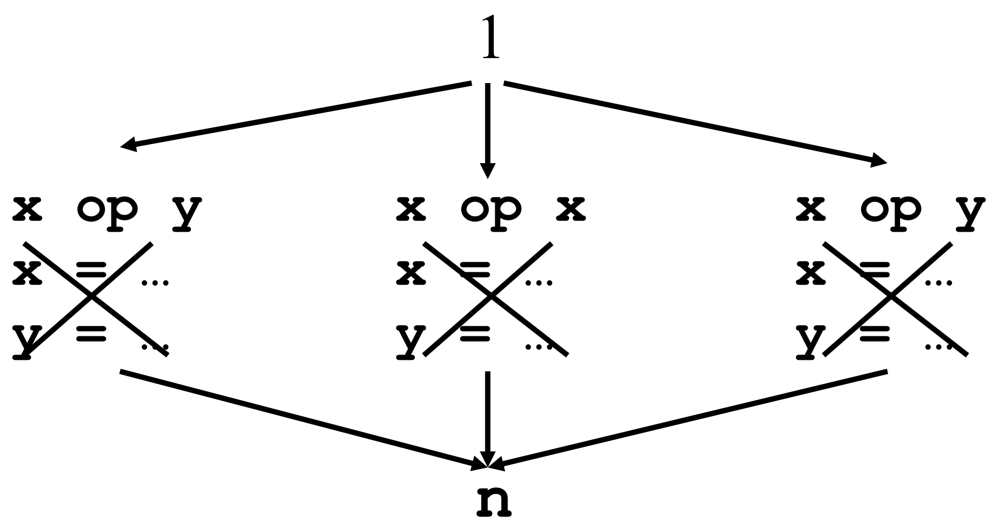
Available Expressions

- An expression `x op y` is available at program point `n` if **every** path from entry to `n` evaluates `x op y`, and there are NO subsequent assignments to `x` or `y` after evaluation and prior to reaching `n`.

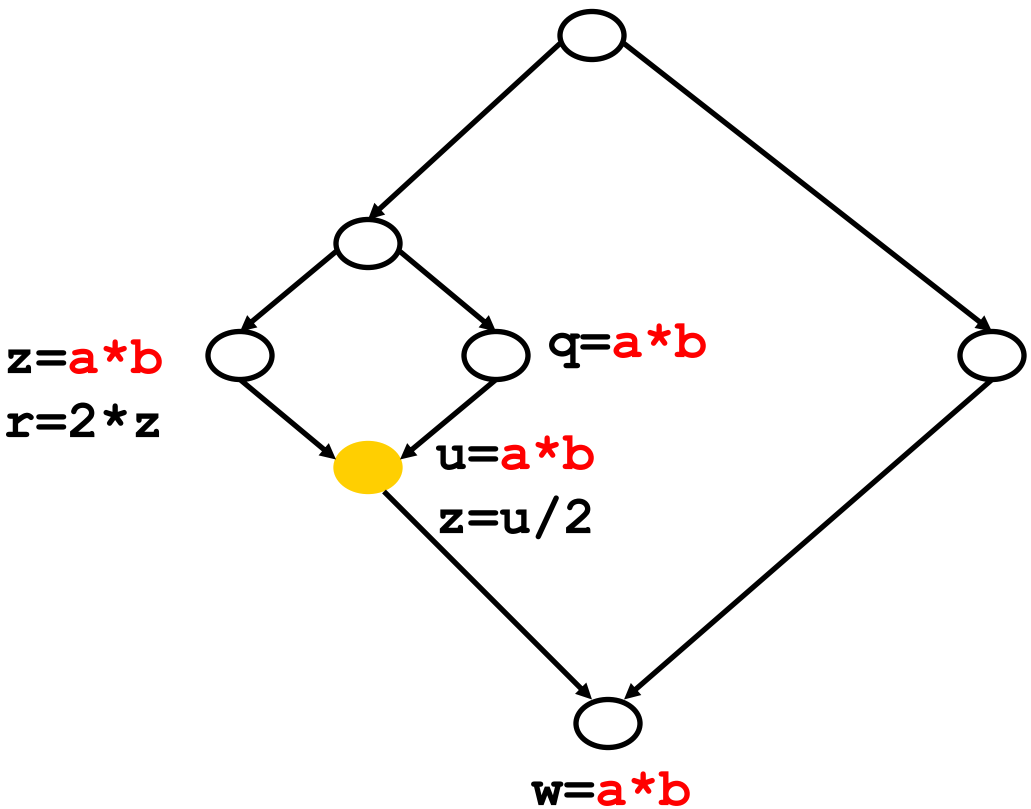


Problem 3. Available Expressions (*Avail*)

- Problem statement: For every node **n**, compute the set of expressions that are available at **n**

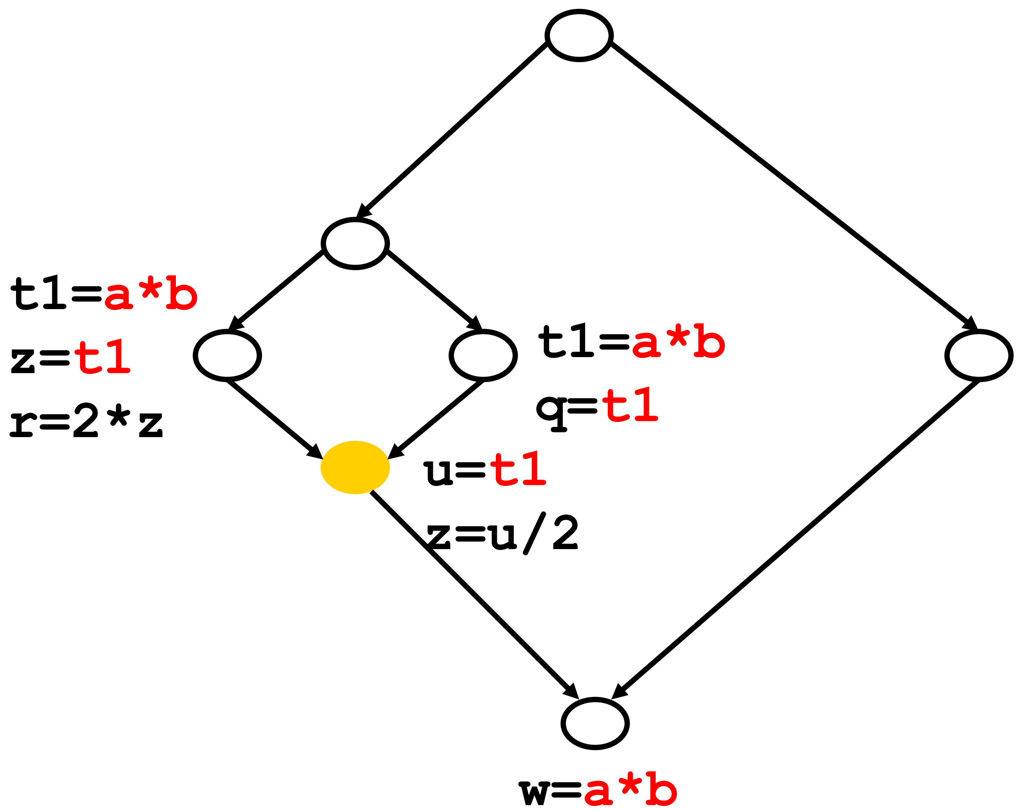


Avail Enables Global Common Subexpression Elimination



Avail Enables Global Common Subexpression Elimination

Can we eliminate **`w=a*b`**?



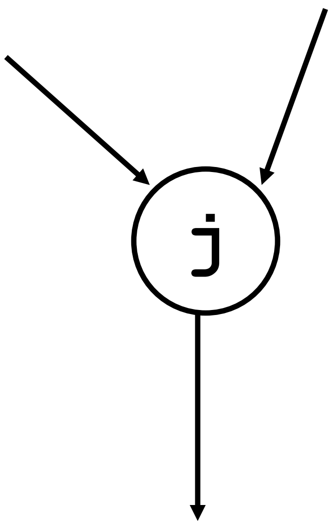
Available Expressions (*Avail*)

- Data?
  - Primitive dataflow facts are expressions, e.g., **`x+y, a*b, a+2`**
  - Analysis propagates sets of expressions, e.g., **`{x+y,a*b}`**
- Dataflow equations at **`j: x = y op z`**?
  - $out_{AE}(j) = (in_{AE}(j) - kill_{AE}(j)) \cup gen_{AE}(j)$
  - $kill_{AE}(j)$: all expressions with operand **`x`**: **`(x op _),(_ op x)`**
  - $gen_{AE}(j)$: new expression: **`{(y op z)}`**

Available Expressions (*Avail*)

- Merge operator?
  - For *Avail*, it is set intersection $\bigcap$

$in_{AE}(j) = \{ \bigcap out_{AE}(i) \mid i \text{ is predecessor of } j \}$

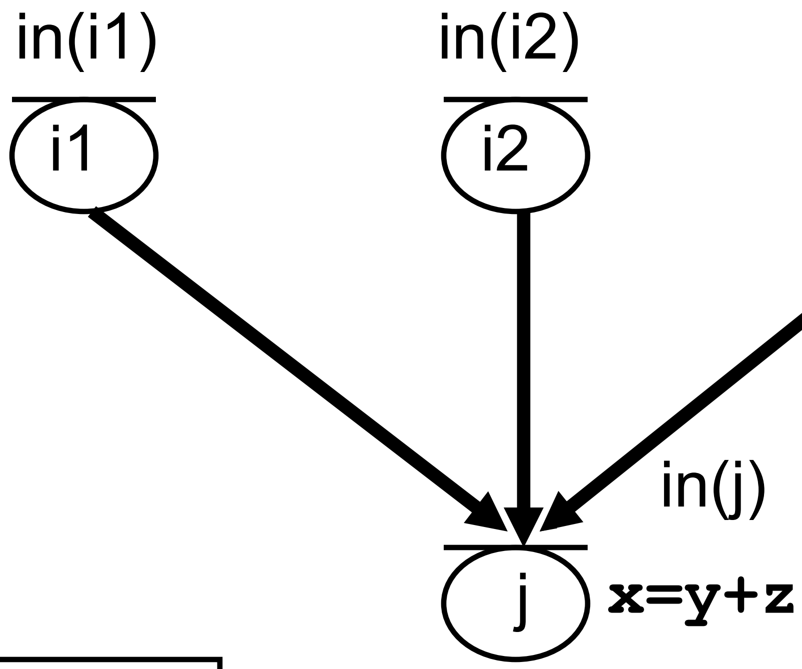


$j: \; x = x \text{ op } y$

$kill_{AE}(j) = (x \text{ op } \_), (\_ \text{ op } x)$

$gen_{AE}(j) = \emptyset$

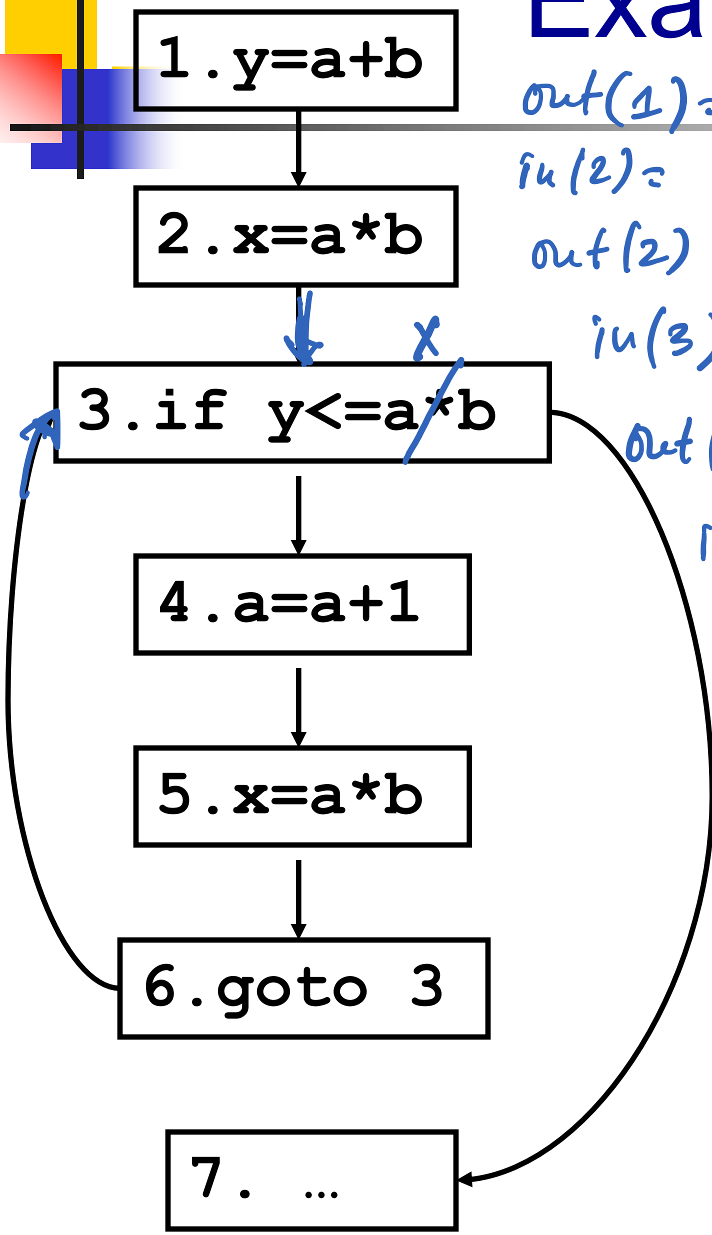
Available Expressions (*Avail*)



Forward, must dataflow problem

$in(1) = \{\}$

Example



$out(1) = \{a+b\}$

$in(2) = \{a+b\}$

$out(2) = \{a+b, a*b\}$

$in(3) = out(2) \cap out(6) = \{a*b\}$

$out(3) = \{a*b\}$

$in(4) = \{a*b\}$

$out(4) = \{\}$

$out(5) = \{a*b\}$

$in(6) = \{a*b\}$

$out(6) = \{a*b\}$

Note on Homework

LV:

$kill_{LV}(B_1) = \{x, y\}$

$gen_{LV}(B_1) = \{x, b\}$

B_1

```
10. x = x + b
11. y = x + 1
12. x = x + y
```

RD

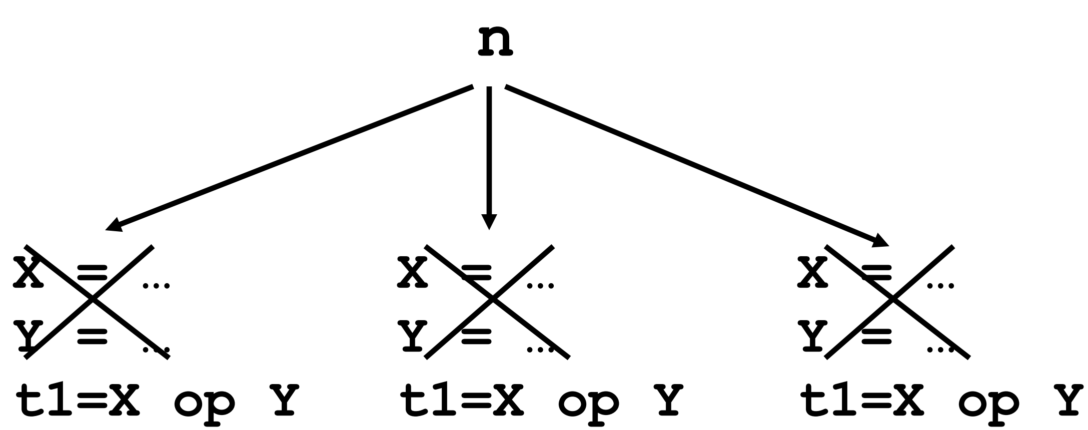
$kill(B_1) = \{$All definitions of y and x

$gen(B_1) = \{(y, 11), (x, 12)\}$

AE:

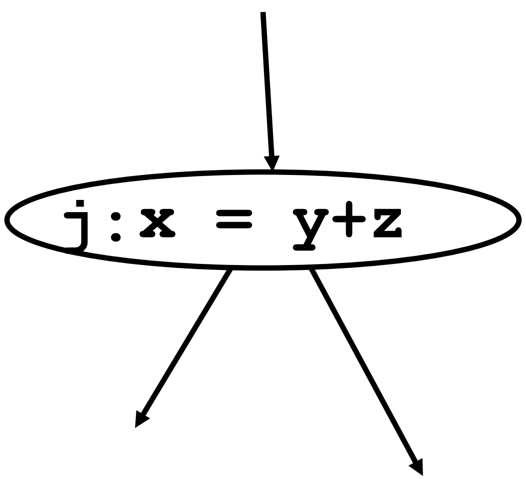
Very Busy Expressions

- An expression **x op y** is very busy at node **n**, if along EVERY path from **n** to the end of the program, we come to a computation of **x op y** BEFORE any redefinition of **x** or **y**.



Problem 4. Very Busy Expressions (*VeryB*)

- Problem Statement: For each node **`n`**, compute the set of expressions that are very busy on exit from **`n`**



Q: What is the data?

Q: What are the equations?

Q: What is $gen_{VB}(i)$?
Q: What is $kill_{VB}(i)$?

Q: What is the merge operator?

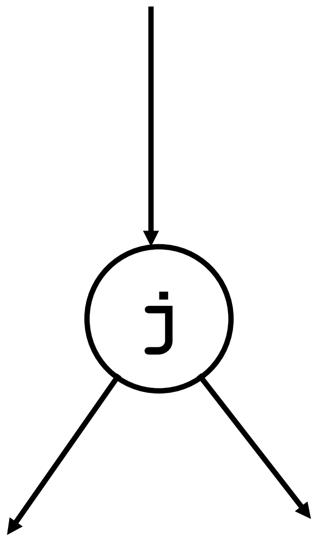
Very Busy Expressions (*VeryB*)

j: x = x+y

- Data?
  - Primitive dataflow facts are expressions, e.g., **`x+y, a*b`**
  - Analysis propagates sets of expressions, e.g., **`{x+y,a*b}`**
- Dataflow equations at **`j: x = y op z`**?
  - in(j) = gen(j) U (out(j) – kill(j))
  - kill(j): all expressions with operand **`x`**: **`(x op _),(_ op x)`**
  - gen(j): new expression: **`{ (y op z) }`**

Very Busy Expressions (*VeryB*)

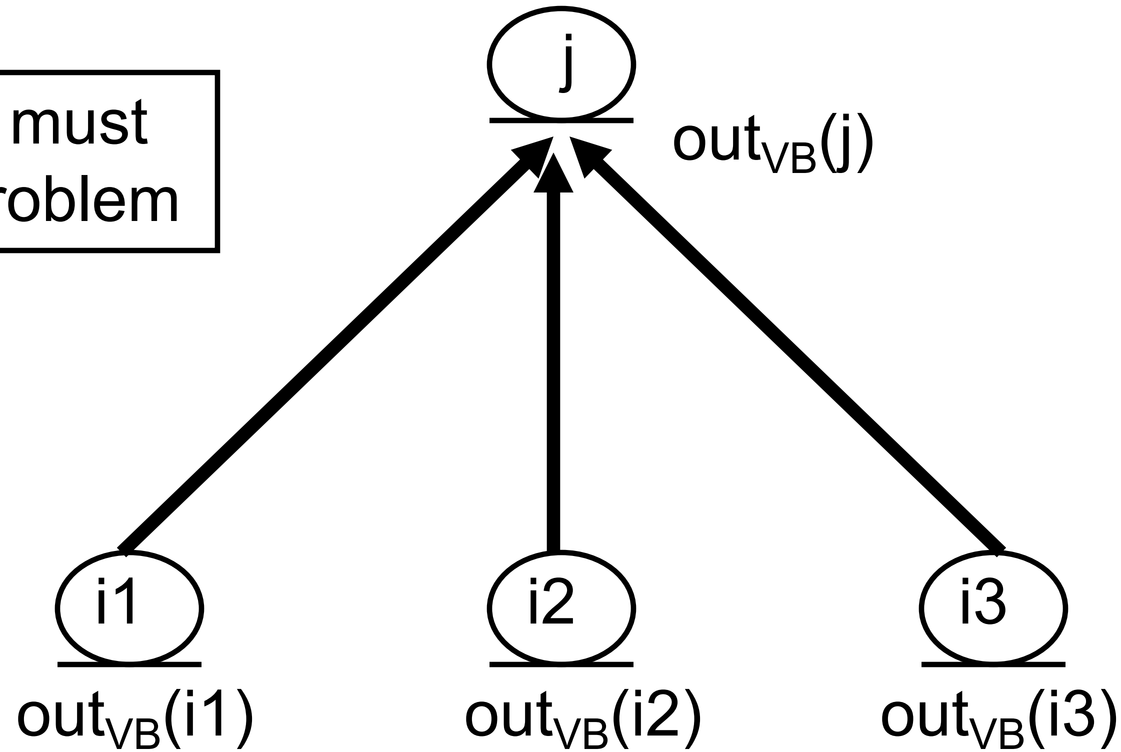
- Merge operator?
  - For *VeryB*, it is set intersection $\bigcap$



$$out_{VB}(j) = \{ \bigcap in_{VB}(i) \mid i \text{ is successor of } j \}$$

Very Busy Expressions

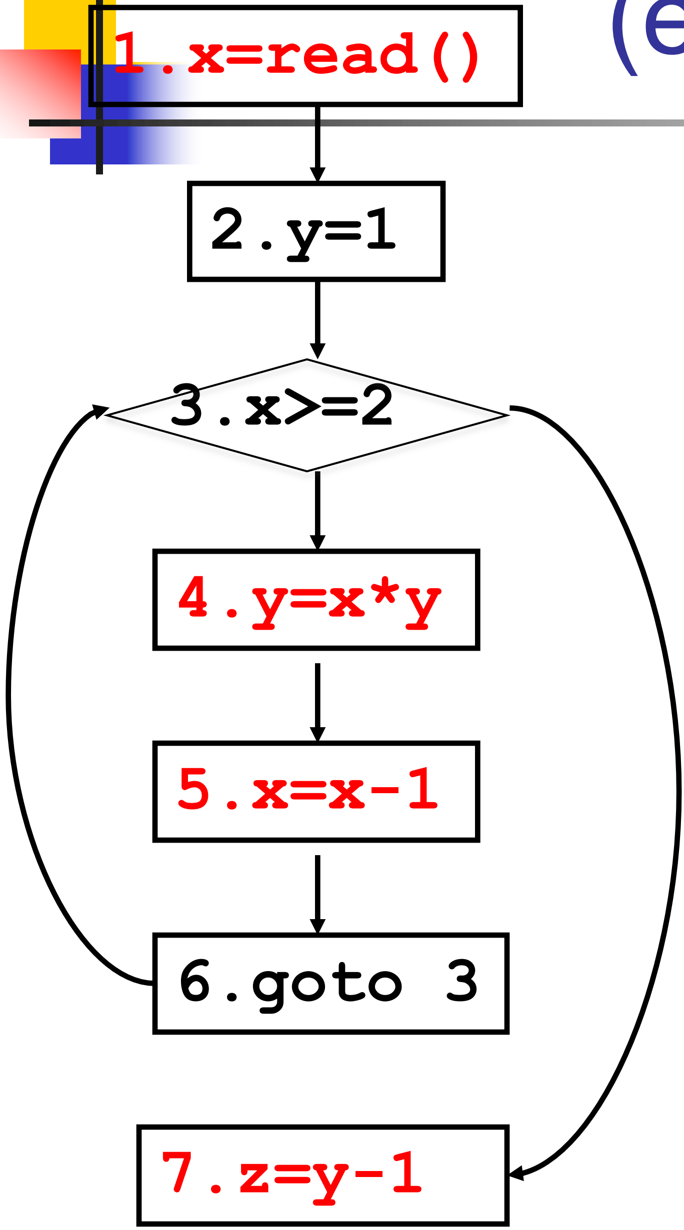
Backward, must dataflow problem



Another Example: Taint Analysis

- A definition **`i: x = … (x,i)`** is tainted if
  - **`i: x = tainted_source()`** is designated as a taint source
    - **`e.g., deviceId=telephony_mgr.getDeviceId();`**
  - or **`i: x = y op z`** and a tainted **`(y,j)`** or a tainted **`(z,k)`** reaches program point **`i`**
- Problem statement: for each node **`n`**, compute the set of tainted definitions that reach **`n`**

Example: Taint Analysis (explicit flow)



Outline of Today's Class

- Catch up
- Dataflow frameworks
  - Lattices
  - Transfer functions
  - Worklist algorithm
- Reading:
  - Dragon Book, Chapter 9.2 and 9.3

Dataflow Problems

	May Problems	*Must* Problems
Forward Problems	Reaching Definitions	Available Expressions
Backward Problems	Live Uses of Variables	Very Busy Expressions

Similarities

- Analyses operate over similar property spaces
- In all cases, analysis operates over a <u>finite</u> set **D** of primitive dataflow facts
  - *Reach*: **D** is the set of <u>all</u> definitions in the program:
    e.g., { `(x,1),(y,2),(x,4),(y,5)` }
  - *Avail* and *VeryB*: **D** is the set of <u>all</u> arithmetic expressions:
    e.g., { `a+b,a*b,a+1` }
  - *Live*: **D** is the set of <u>all</u> variables
    e.g., { `x,y,z` }
- Solution at node `n` is a subset of **D** (e.g., a definition either reaches `n` or it does not reach `n`)

Similarities

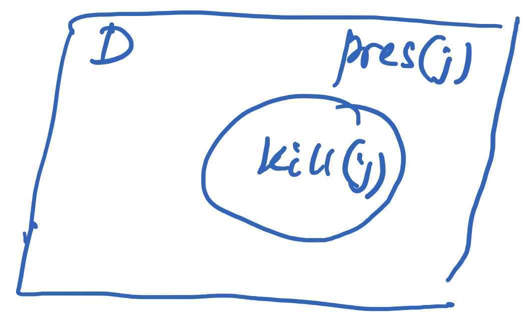
- Dataflow equations have same form (from now on, we'll focus on forward problems):

$out(j) = f_j(in(j))$

out(j) = (in(j) – kill(j)) U gen(j) =

(in(j) $\cap$ pres(j)) U gen(j)

in(j) = { V out(i) | i is predecessor of j }



pres(j) is the complement of kill(j)

- A note: what makes the 4 classical problems special is that sets kill(j)/pres(j) and gen(j) <u>do not depend on in(j)</u>
- Set union and set intersection can be implemented as logical OR and AND respectively

Similarities

- Dataflow equation at node **j** is a transfer function. It takes in(j) as argument and produces out(j) as result:

  - **$out(j) = f_j(in(j))$**

Dataflow Frameworks

- We generalize and study properties of the property space
  - Property space is a lattice
  - Choice of lattice settles merge operator
- We generalize and study properties of the transfer function space
  - Functions are monotone or distributive
- We generalize and study properties of the worklist algorithm that computes a solution

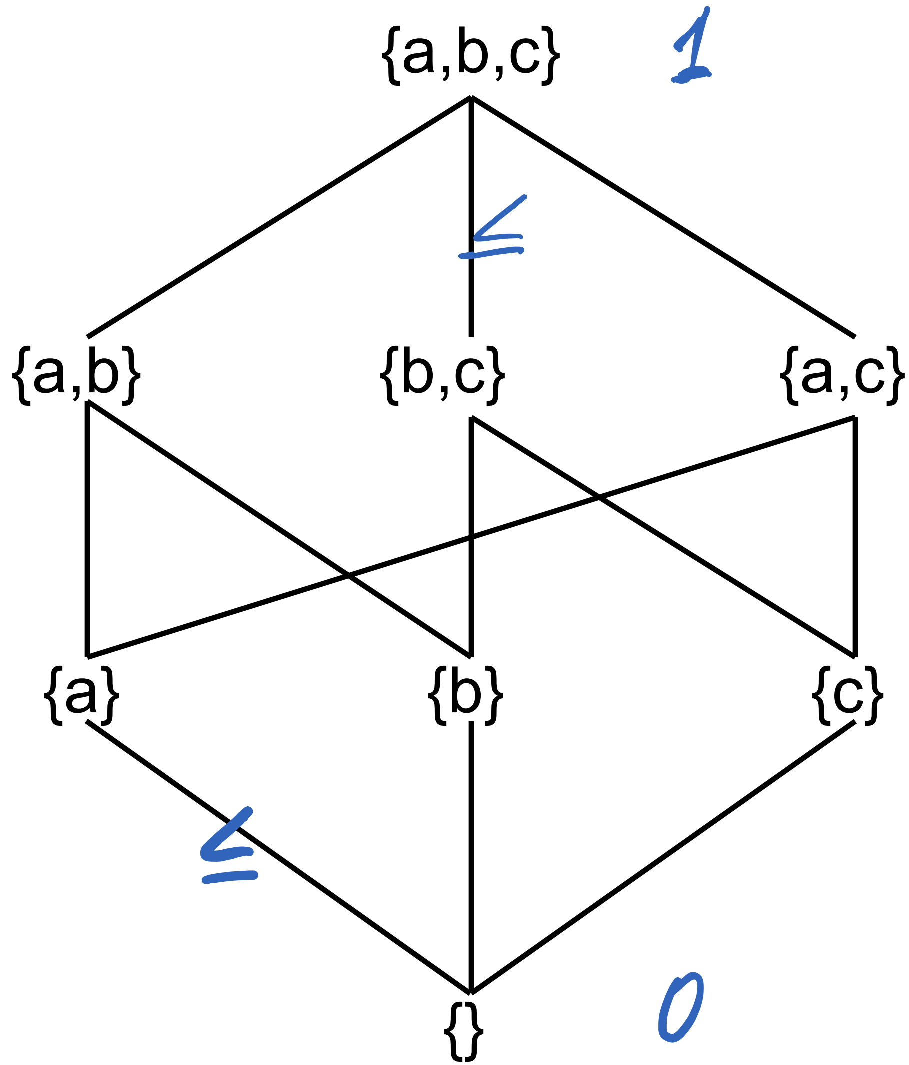
Lattices

- Partial ordering (denoted by **≤** or ⊑)
  - Relation between pairs of elements
  - Reflexive **a ≤ a**
  - Anti-symmetric **a ≤ b** and **b ≤ a** ==> **a = b**
  - Transitive **a ≤ b** and **b ≤ c** ==> **a ≤ c**
- Partially ordered set (poset) (set S, ≤)
  - **0** element **0 ≤ a**, for every **a** in S
  - **1** element **a ≤ 1**, for every **a** in S

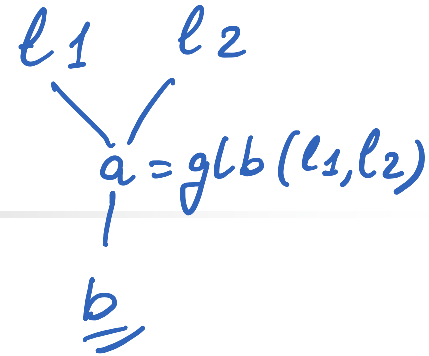
We don't necessarily need 0 or 1 element

Poset Example

D = {a,b,c}
The poset is 2^D, ≤ is set inclusion



Lattice Theory



- Greatest lower bound (glb)

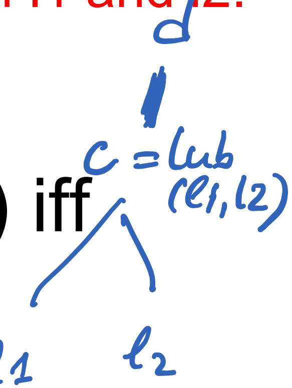
l1, l2 in poset S, **a** in poset S is the **glb(l1,l2)** iff

1) **a ≤ l1** and **a ≤ l2**

2) for any **b** in S, **b ≤ l1, b ≤ l2** implies **b ≤ a**

If glb exists, it is unique. Why? Called ***meet*** (denoted by Λ or ⊓) of l1 and l2.

- Least upper bound (lub)



l1, l2 in poset S, **c** in poset S is the **lub(l1,l2)** iff

1) **c ≥ l1** and **c ≥ l2**

2) for any **d** in S, **d ≥ l1, d ≥ l2** implies **d ≥ c**

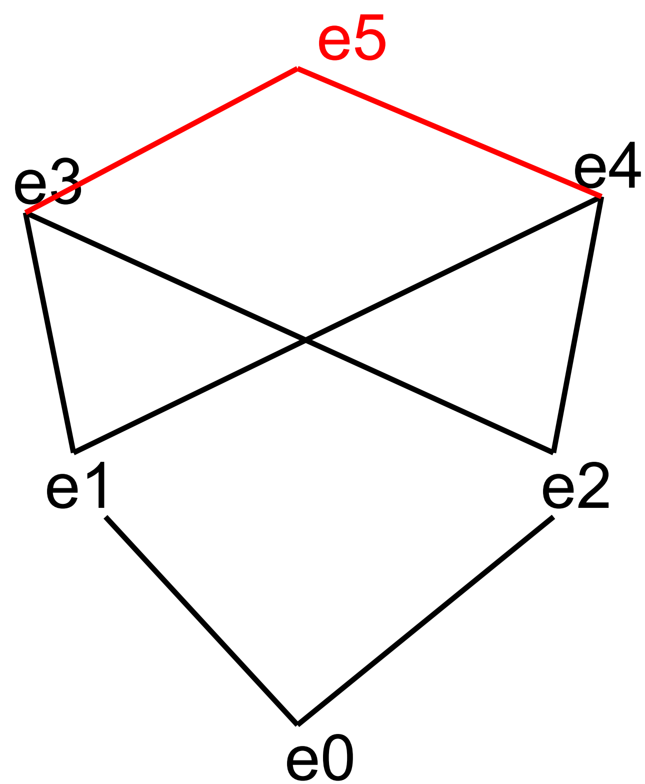
If lub exists, it is unique. Called ***join*** (denoted by V or ⊔) of l1 and l2.

Definition of a Lattice **(L, Λ, V)**

- A lattice **L** is a poset under ≤, such that every pair of elements has a **glb (meet)** and **lub (join)**

- A lattice need not contain a 0 or 1 element
- A finite lattice must contain 0 and 1 elements
- Not every poset is a lattice
- If there is element **a** such that **a ≤ x** for every **x** in **L**, then **a** is the 0 element of **L**
- If there is **a** such that **x ≤ a** for every **x** in **L**, then **a** is the 1 element of **L**

A Poset but Not a Lattice



$e1 \le e3$, $e1 \le e4$

$e2 \le e3$, $e2 \le e4$

There is no **lub(e3,e4)** in this poset so it is not a lattice.

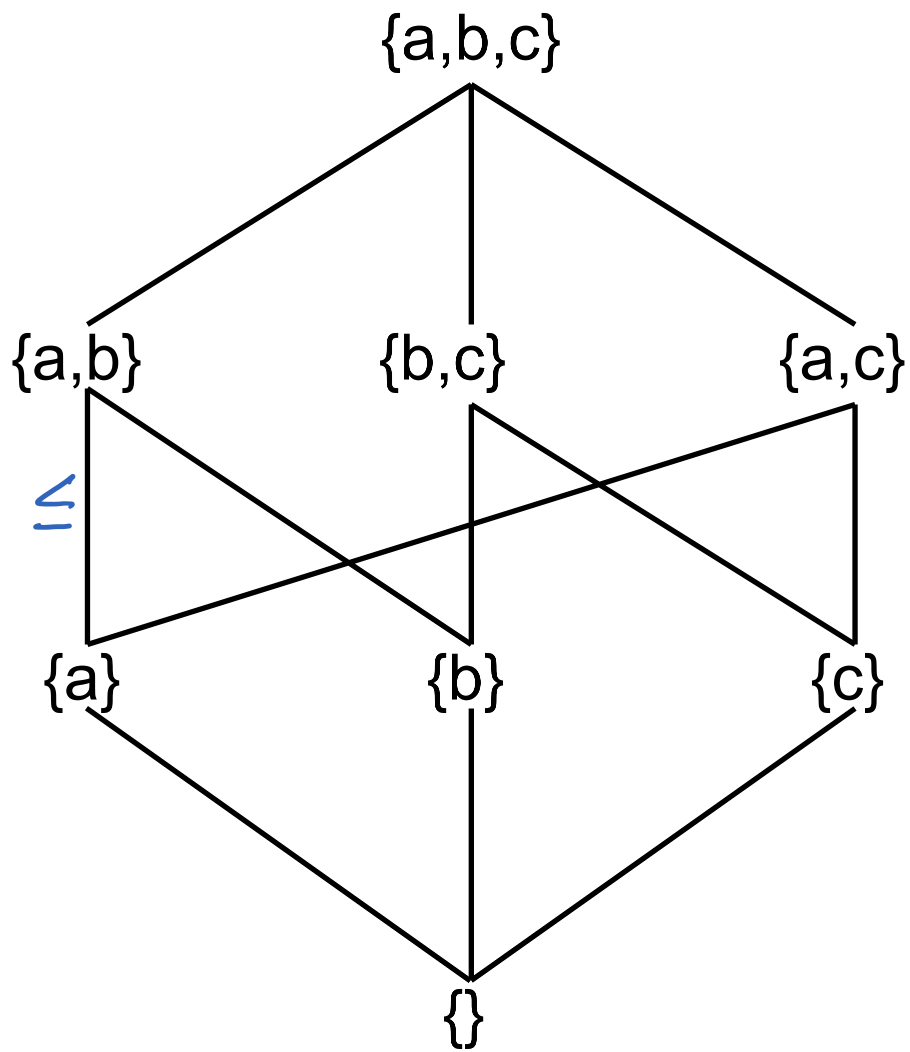
Suppose we add the **lub(e3,e4)**, is it a lattice?

Is This Poset a Lattice

D = {a,b,c}
The poset is 2^D, $\leq$ is set inclusion

$glb(\ell_1, \ell_2) = \ell_1 \cap \ell_2$

$lub(\ell_1, \ell_2) = \ell_1 \cup \ell_2$ $\leq$



Examples of Lattices

- H = (2^D, ∩, U) where D is a finite set
  - **glb(s1,s2)** denoted **s1Λs2**, is set intersection **s1∩s2**
  - **lub(s1,s2)** denoted **s1Vs2**, is set union **s1Us2**
- J = (N_1, gcd, lcm)
  - Partial order is integer divide on N_1
  - **lub(n1,n2)** denoted **n1Vn2** is **lcm(n1,n2)**
  - **glb(n1,n2)** denoted **n1Λn2** is **gcd(n1,n2)**

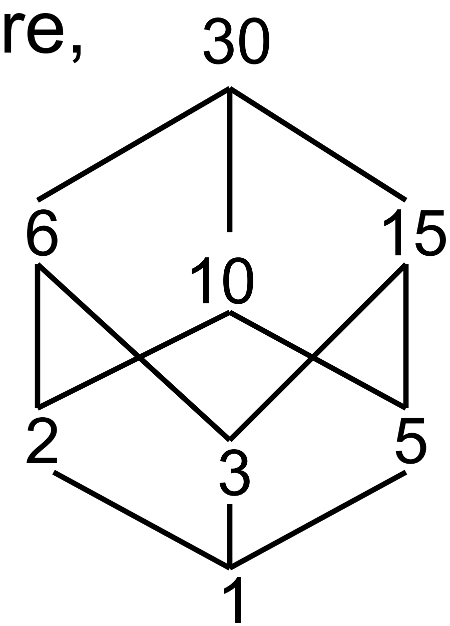
  (N_1 denotes natural numbers starting at 1)

Chain

- A poset C where for every pair of elements **c1**, **c2** in C, either **c1 ≤ c2** or **c2 ≤ c1**.
  - E.g., {} ≤ {a} ≤ {a,b} ≤ {a,b,c}
  - E.g., from the lattice J as shown here,

    1 ≤ 2 ≤ 6 ≤ 30

    1 ≤ 3 ≤ 15 ≤ 30



- A lattice s.t. every ascending chain is finite, is said to satisfy the *Ascending Chain Condition*

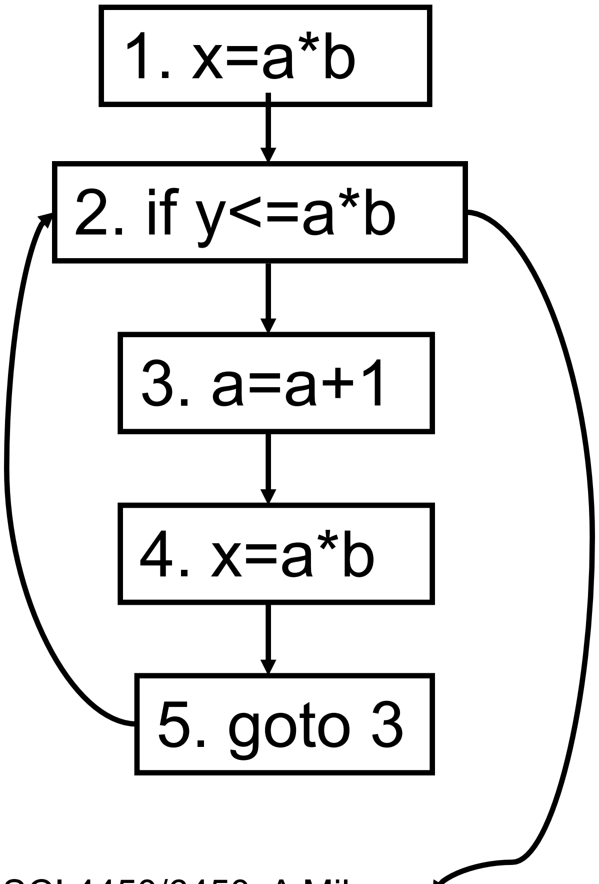
Lattices in Dataflow Analysis

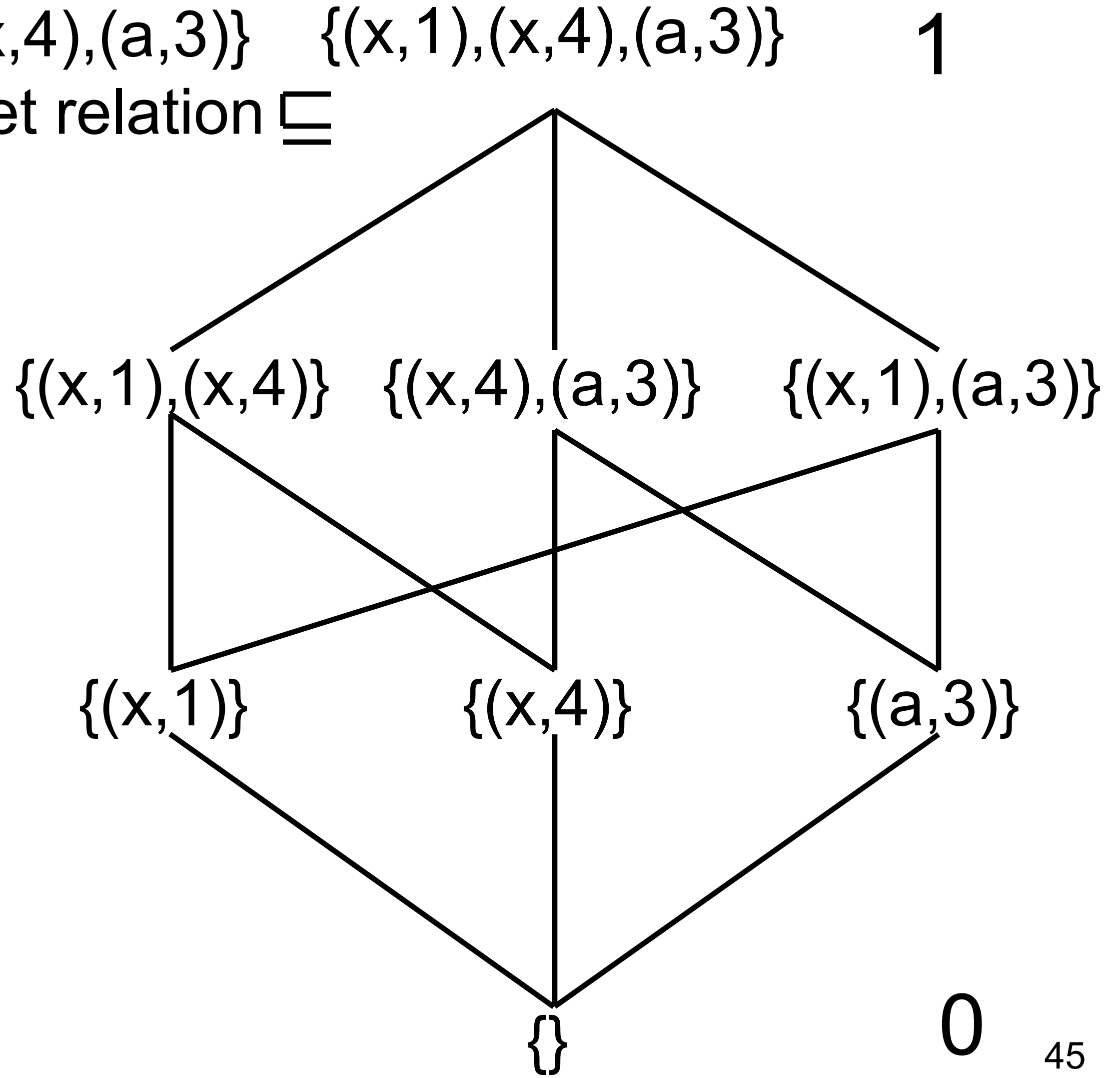
- Lattices define property space
- Lattice properties lead to certain properties of the worklist algorithm (standard way of solving dataflow problems)

Dataflow Lattices: *Reach*

D = all definitions:{(x,1),(x,4),(a,3)}

Poset is 2^D, ≤ is the subset relation $\sqsubseteq$

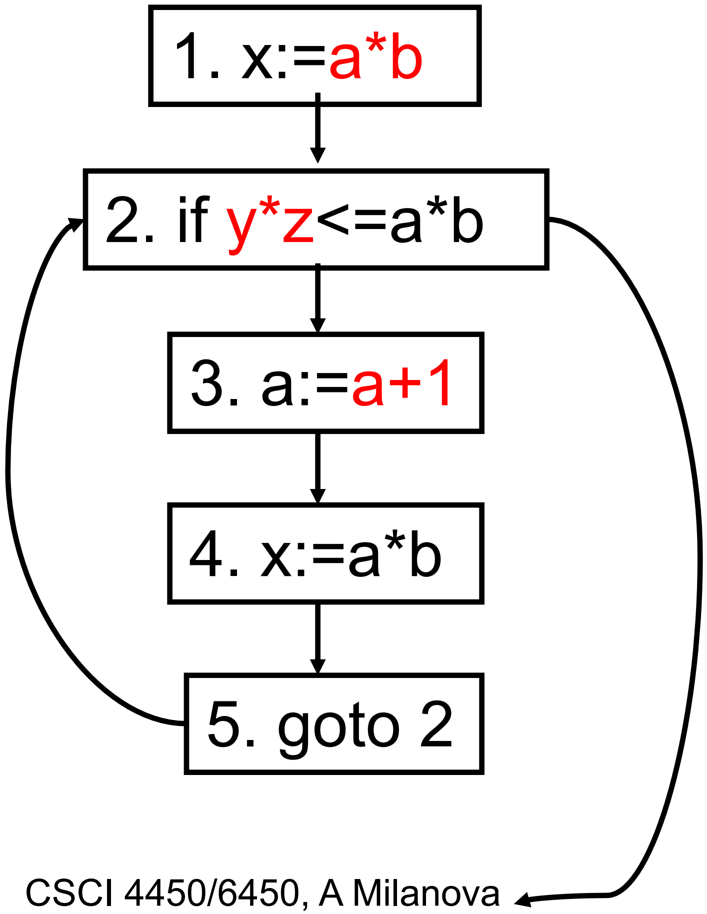


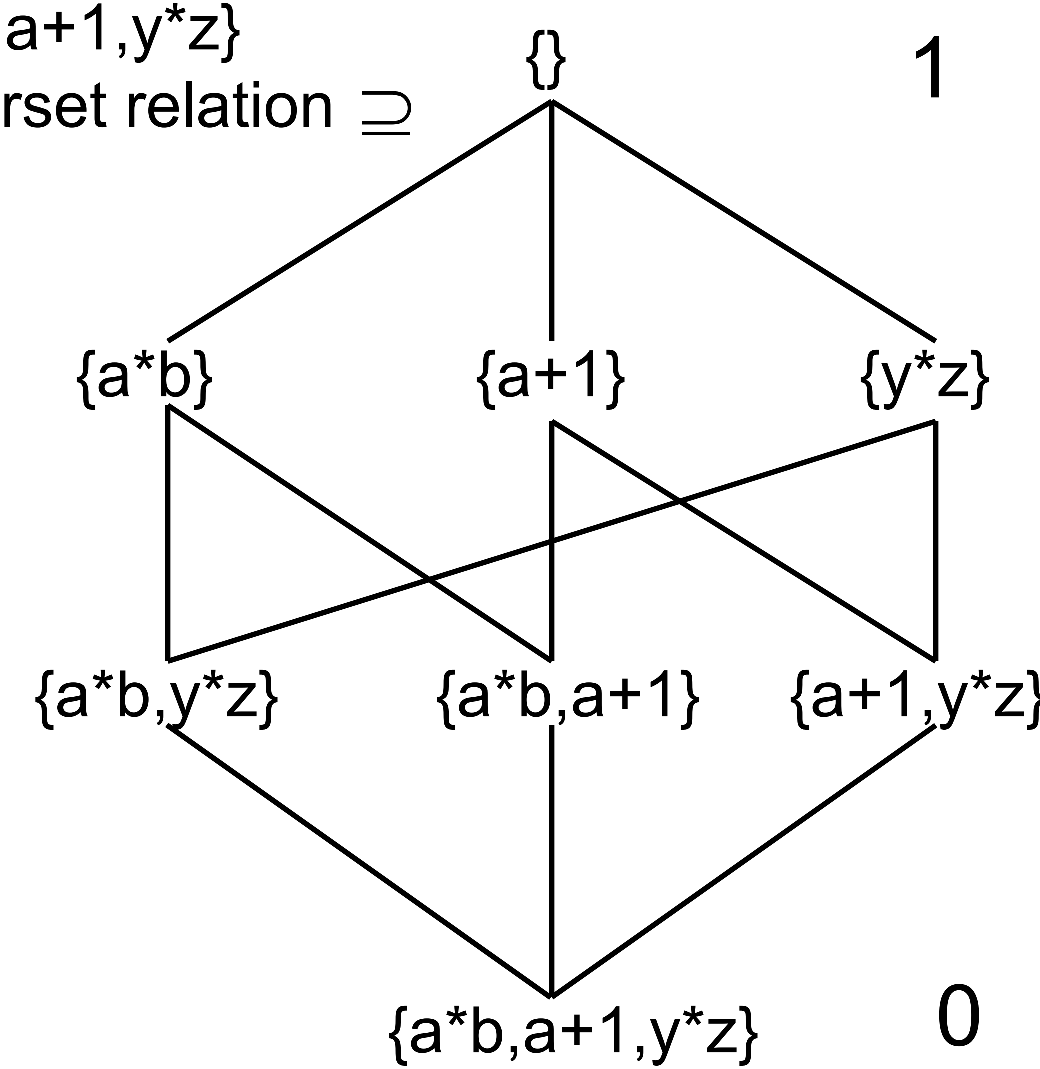


Dataflow Lattices: *Avail*

D = all expressions: {a*b,a+1,y*z}

Poset is 2^D, $\leq$ is the superset relation $\supseteq$





Property Space

- **Property space must be:**
  1. A lattice **L, ≤**
  2. **L** satisfies the *Ascending Chain Condition*

     Requires that all ascending chains are finite

Property Space

- **Merge operator $\vee$** must be the join of **L**
- In dataflow, **L** is often the lattice of the subsets over a finite set of dataflow facts **D**
  - Choose universal set **D** (e.g., all definitions)
  - Choose ordering operation **≤**. Since the merge operator must be the join of **L**, a *may* problem sets **≤** to **subset** and a *must* problem sets **≤** to **superset**

Example: *Reach* Lattice

- Property space is the lattice of the subsets
  - **D** is the set of all definitions in program
  - **≤** is the **subset** operation
    - Thus, join is set union, as needed for *Reach*, which is a *may* problem
  - Lattice has a **0** being **{}**, and a **1** being **D**
  - Lattice satisfies the *Ascending Chain Condition*

Example: *Avail* Lattice

- Property space is the lattice of the subsets
  - **D** is the set of all expressions in the program
  - **≤** is **superset**
    - Thus, join is set intersection, as needed for *Avail*, which is a *must* problem
  - Lattice has a **0** being **D,** and a **1** being **{}**
  - Lattice satisfies *Ascending Chain Condition*

(Monotone) Dataflow Framework

- A problem fits into the dataflow framework if
  - its <u>property space</u> is a lattice **L, ≤** that satisfies the *Ascending Chain Condition*
  - its merge operator V is the join of **L**

  and
  - its <u>transfer function space</u> **F: L→ L** is monotone
- Thus, we can make use of a generic solution procedure, known as the worklist algorithm (also the maximal fixpoint algorithm or the fixpoint iteration algorithm)

Outline of Today's Class

- Catch up
- Dataflow frameworks
  - Lattices
  - Transfer functions
  - Worklist algorithm

- Reading:
  - Dragon Book, Chapter 9.2 and 9.3

Transfer Functions

- **The transfer functions: f: L→ L**. Formally, function space **F** is such that
  1. **F** contains all $\mathbf{f_j}$
  2. **F** contains the identity function **id(x) = x**
  3. **F** is closed under composition
  4. Each **f** must be **monotone**

Monotonicity Property

- **F: L→ L** is monotone if and only if:

  (1) **a,b** in **L**, **f** in **F** then $\mathbf{a \leq b} \Longrightarrow \mathbf{f(a) \leq f(b)}$

  or (equivalently):

  (2) **x,y** in **L**, **f** in **F** then $\mathbf{f(x) \vee f(y) \leq f(x \vee y)}$

- Theorem: Definitions (1) and (2) are equivalent.
  - Show that (1) implies (2)
  - Show that (2) implies (1)

Monotonicity Property

- Show that (1) implies (2)

Distributivity Property

- **F: L → L** is distributive if and only if **x,y** in **L**, **f** in **F** then **f(x) V f(y) = f(x V y)**

- A distributive function is also monotone but not the other way around

- Distributivity is a very nice property!

Monotonicity and Distributivity

- Is classical *Reach* distributive?
  - Yes
  - To show distributivity:

For each **j**: $((X_1 \cup X_2) \cap pres(j)) \cup gen(j) =$

$((X_1 \cap pres(j)) \cup gen(j)) \cup ((X_2 \cap pres(j)) \cup gen(j))$

$((X_1 \cup X_2) \cap pres(j)) \cup gen(j) =$

$((X_1 \cap pres(j)) \cup (X_2 \cap pres(j))) \cup gen(j) =$

$((X_1 \cap pres(j)) \cup gen(j)) \cup ((X_2 \cap pres(j)) \cup gen(j))$

Monotone Dataflow Framework

- A problem fits into the dataflow framework if
  - its <u>property space</u> is a lattice **L, ≤** that satisfies the *Ascending Chain Condition*
  - its merge operator V is the join of **L**

  and
  - its <u>transfer function</u> space **F: L→ L** is monotone
- Thus, we can make use of a generic solution procedure, known as the worklist algorithm.

Worklist Algorithm for Forward Dataflow Problems

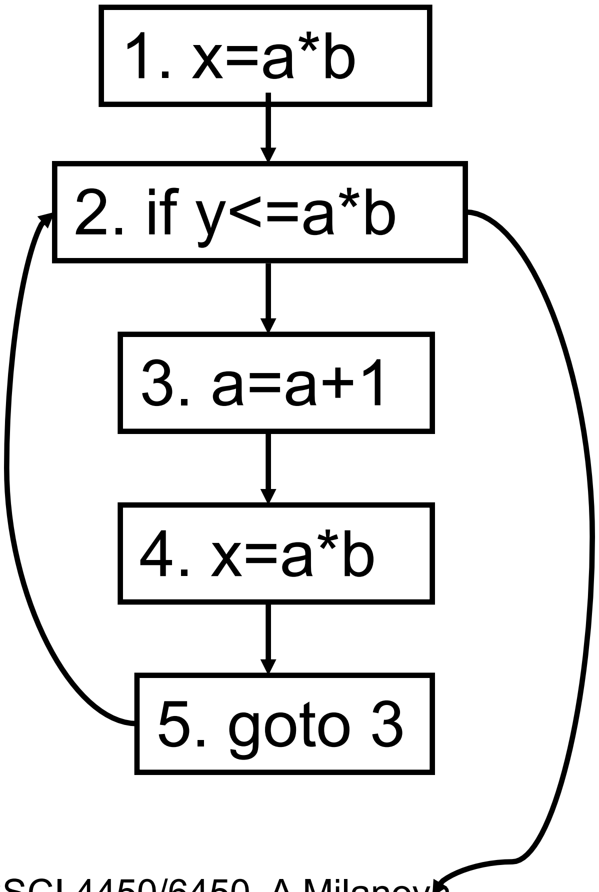
```
/* Initialize to initial values; 1 is entry node of CFG */
in(1) = InitialValue; out(1) = f_1(in(1))
for m = 2 to n do in(m) = 0; out(m) = f_m(0)
W = {2,…,n} /* put every node but 1 on the worklist */

while W ≠ Ø do {
    remove j from W
    in(j) = V { out(i) | i is predecessor of j }
    out(j) = f_j(in(j))
    if out(j) changed then
       W = W U { k | k is successor of j }
}
```

Worklist Algorithm on *Reach*

D = all definitions:{(x,1),(x,4),(a,3)}

Poset is 2^D, ≤ is the subset relation ⊑



Termination Argument

- Why does the algorithm terminate?

- Sketch of argument:

  A node j is placed on the worklist only if the out(i) of a predecessor i changes. Monotonicity of **f** ensures that $in^k(i) \leq in^{k+1}(i)$ and $out^k(i) \leq out^{k+1}(i)$.

  in(i) and out(j) sets and in **L** and **L** satisfies

  the *Ascending Chain Condition*; therefore, there is only a finite number of times each out(i) changes

Correctness Argument

- Theorem: Worklist algorithm computes a solution that satisfies the dataflow equations

- Why?
- Sketch of argument:

Suppose either (1) Vout(i) =/= in(j) or (2) out(j) =/= f_j(in(j))

For (1) to hold we must have “grown” out(i) and not added successor j to worklist or otherwise in(j) would have been recomputed to account for new out(i); This is impossible.

Precision Argument

- Theorem: Worklist algorithm computes the least solution of the dataflow equations.
  - Historically though, this solution is called the **maximal fixpoint solution (MFP)**
  - For every node **j**, worklist algorithm computes a solution **MFP(j)** = {in(j),out(j)}, such that for every solution {in'(j),out'(j)} of the dataflow equations we have in(j) ≤ in'(j) and out(j) ≤ out'(j)

Example

1. z:=x+y

2. if (z > 500)

3. skip

	Solution1	Solution2
$in_{Avail}(1) = \emptyset$	$\emptyset$	$\emptyset$
$out_{Avail}(1) = (in_{Avail}(1) - E_z) \cup \{x+y\}$	{x+y}	{x+y}
$in_{Avail}(2) = out_{Avail}(1)$ **V** $out_{Avail}(3)$	{x+y}	$\emptyset$
$out_{Avail}(2) = in_{Avail}(2)$		
$in_{Avail}(3) = out_{Avail}(2)$	{x+y}	$\emptyset$
$out_{Avail}(3) = in_{Avail}(3)$		

Equivalent to: $in_{Avail}(2) = \{x+y\}$ **V** $in_{Avail}(2)$
and recall that **V** is ∩ (i.e., set intersection).