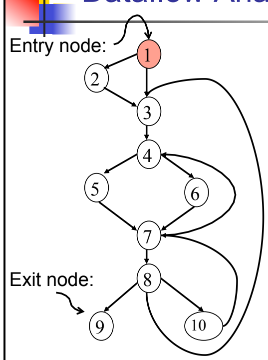
Dataflow Analysis: Dataflow Frameworks

Outline of Today's Class

- Catch up
- 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)) ∪ gen(j)

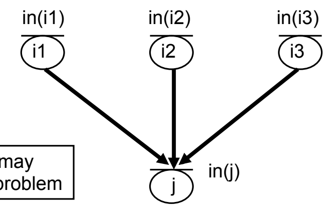
(gen and kill are parameters)

4. Merge operator V

in(j) = V out(i)

i is predecessor of j

Problem 1: Reaching Definitions



Forward, may dataflow problem

What are the primitive dataflow facts?
Definitions, e.g., **(x,1),(y,6)**
Equations act on sets of definitions.

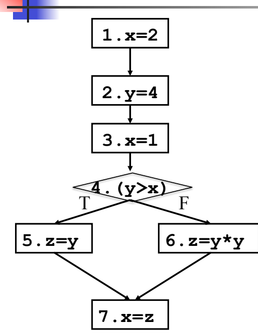
Problem 2. Live Uses of Variables (*Live*)

- 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 may be live <u>on exit</u> 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

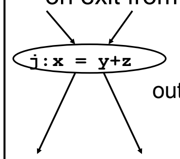


Live Uses of Variables (*Live*)

- Data
  - Primitive facts: variables **`x`**
  - Propagates sets: **`{x,y,z}`**
- Dataflow equations. At **`j: x = y+z`**
  - $kill_{LV}(j)$: **`{x}`**
  - $gen_{LV}(j)$: **`{y,z}`**
- Merge operator: set union $\cup$

Live Uses of Variables (*Live*)

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



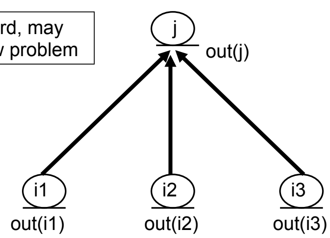
$in_{LV}(j) = (out_{LV}(j) - kill_{LV}(j)) \cup gen_{LV}(j)$

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

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

Problem 2: Live Uses of Variables

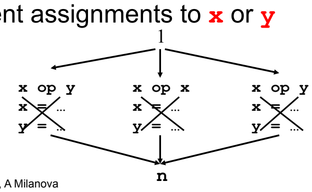
Backward, may dataflow problem



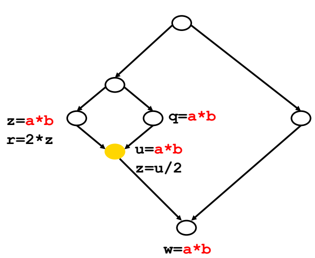
What are the primitive dataflow facts? Variables, e.g., **`x,y,z`**. Equations act on sets of variables.

Problem 3: Available Expressions (*Avail*)

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

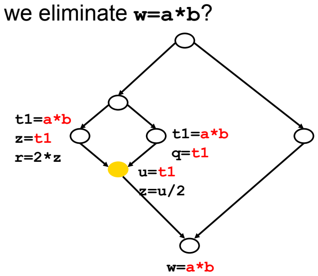


Avail Enables Global Common Subexpressions



Avail Enables Global Common Subexpressions

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



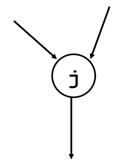
Available Expressions (*Avail*)

- 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)}`

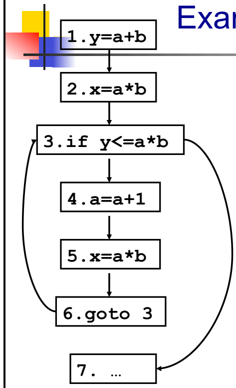
Available Expressions (*Avail*)

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

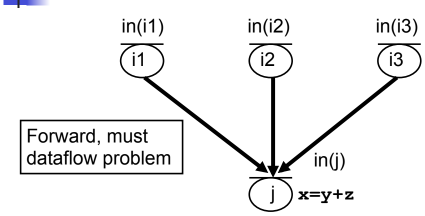
$in_{AE}(j) = \{\bigcap out_{AE}(i) \mid i \text{ is predecessor of } j\}$



Example



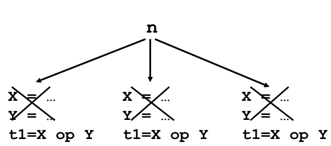
Problem 3: Available Expressions



What are the primitive dataflow facts? Expressions, e.g., `x+y,a*b`. Equations act on sets of expressions.

Problem 4: Very Busy Expressions (*VeryB*)

- 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`.



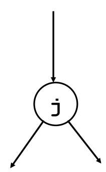
Very Busy Expressions (*VeryB*)

- 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`?
  - $in_{VB}(j) = (out_{VB}(j) - kill_{VB}(j)) \cup gen_{VB}(j)$
  - $kill_{VB}(j)$: all expressions with operand `x`: `(x op _),(_ op x)`
  - $gen_{VB}(j)$: new expression: `{(y op z)}`

Very Busy Expressions (*VeryB*)

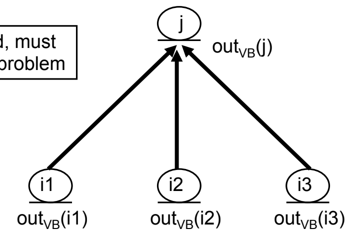
- Merge operator?
  - For *VeryB*, it is set intersection $\bigcap$

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



Very Busy Expressions

Backward, must dataflow problem

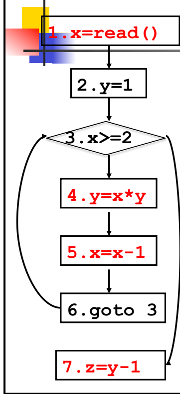


Another Example: Taint Analysis

- A definition **(x,k)** is tainted if **k** is designated as a taint source, or **(x,k)** is computed based on an operand that is tainted.

- Problem statement: for each node **n**, compute the set of tainted definitions that may reach **n**.

Example: Taint Analysis (explicit flow)



Outline of Today's Class

- Catch up
- Dataflow frameworks
- Lattice
- 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 the same form (from now on, we'll focus on forward problems):

out(j) = (in(j) – kill(j)) ∪ gen(j) =
(in(j) ∩ pres(j)) ∪ 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>
- Thus, set union and set intersection can be implemented as logical OR and AND respectively

Similarities

- The dataflow equation at node **j** is a transfer functions. It take in(j) as argument and produces out(j) as result:
  - **out(j) = f_j(in(j))**

Dataflow Frameworks

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

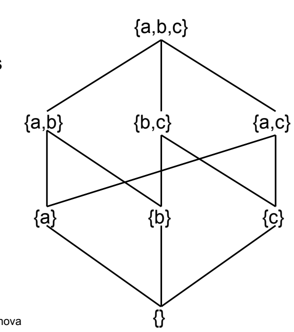
Lattice Theory

- 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 and 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**

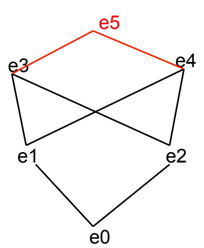
2) for any **d** in S, **d ≥ l1, d ≥ l2** implies **d ≥ c**

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

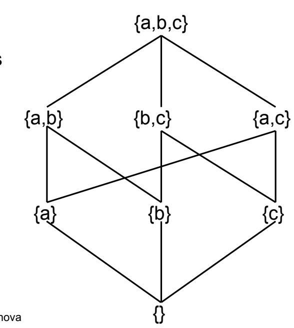


There is no **lub(e3,e4)** in this poset so it is not a lattice.

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, ≤ is set inclusion



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)**

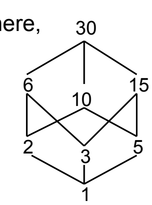
  (N_1 denotes natural numbers starting at 1)

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*

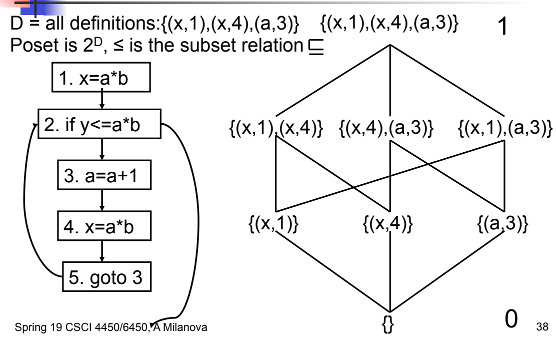
Lattices in Dataflow Analysis

- Lattices define property space
- Lattices entail properties of the standard dataflow analysis solution procedure (the worklist algorithm, which we will study shortly)

Dataflow Lattices: *Reach*

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

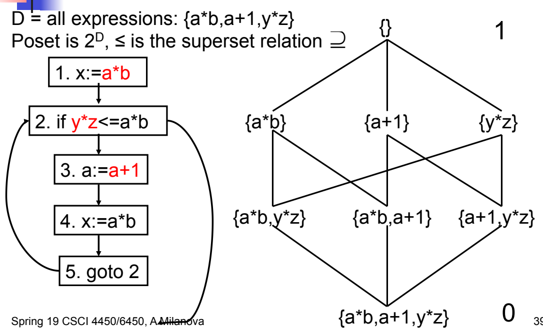
Poset is 2^D, $\leq$ is the subset relation $\sqsubseteq$



Dataflow Lattices: *Avail*

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

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



Dataflow Frameworks

- Equations:

  $in(j) = \bigvee_{i \text{ in } pred(j)} out(i)$ $\qquad$ $out(j) = f_j(in(j))$

  where:
  - in(j), out(j) are elements of a property space
  - f_j is the transfer function associated with node **j**
  - V is the merge operator

Dataflow Frameworks (cont.)

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

     Requires that all ascending chains are finite
- **The merge operator V** 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 is must be to the join of **L**, a *may* problem entails that **≤** is **subset**. Conversely, a *must* problem entails that **≤** is **superset**

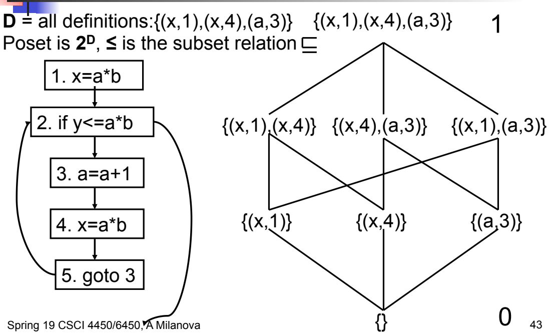
Example: *Reach* Lattice

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

Reach Lattice

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

Poset is $\mathbf{2^D}$, **≤** is the subset relation ⊑



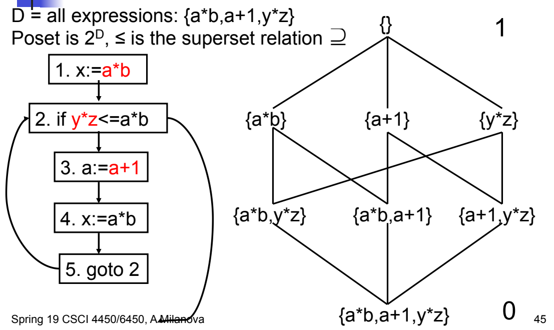
Example: *Avail* Lattice

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

Dataflow Lattices: *Avail*

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

Poset is 2^D, ≤ is the superset relation ⊒



Transfer Functions

- **The transfer functions: f_j: 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_j is **monotone**

Monotonicity

- **F: L→ L** is monotone if and only if:
  (1) **a,b** in **L**, **f** in **F** then **a ≤ b** ⟹ **f(a) ≤ f(b)**
  or (equivalently):
  (2) **x,y** in **L**, **f** in **F** then **f(x) V f(y) ≤ f(x V y)**
- Theorem: Definitions (1) and (2) are equivalent.
  - Show that (1) implies (2)
  - Show that (2) implies (1)

Distributivity

- **F: L → L** is distributive if and only if **x,y** in **L**, **f** in **F** then **f(x V y) = f(x) V f(y)**
- Every 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 ((in(j) U in'(j)) ∩ pres(j)) U gen(j)** =
((in(j)∩pres(j)) U gen(j)) U ((in'(j)∩pres(j)) U gen(j))

((in(j) U in'(j)) ∩ pres(j)) U gen(j) =
((in(j) ∩ pres(j)) U (in'(j) ∩ pres(j))) U gen(j) =
((in(j)∩pres(j)) U gen(j)) U ((in'(j)∩pres(j)) U gen(j))

Monotone Dataflow Frameworks

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

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

Worklist Algorithm for Forward Dataflow Problems

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

while W ≠ Ø do {
   remove j from W
   out(j) = fj(in(j))                  outReach(j) = inReach(j)∩pres(j)Ugen(j)
   for i in successors(j)
     if out(j) ≰ in(i) then {           if outReach(j) ⊄ inReach (i)
        in(i) = out(j) V in(i)            inReach(i) = outReach(j) U inReach(i)
        W = W U { i }
     }
}
```

Worklist Algorithm for Forward Dataflow Problems (slightly different)

```
/* Initialize to initial values; 1 is entry node of CFG */
in(1) = InitialValue; out(1) = f1(in(1))
for m := 2 to n do in(m) = 0; out(m) = fm(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) = fj(in(j))
   if out(j) changed then
      W = W U { k | k is successor of j }
}
```


Termination Argument

- Why does the algorithm terminate?
- Sketch of proof:

  At each iteration, at least one out(j) changes. Since out(j) in **L**, and **L** satisfies the *Ascending Chain Condition*, out(j) changes at most **O(h)** times where **h** is the height of the lattice **L**

Correctness Argument

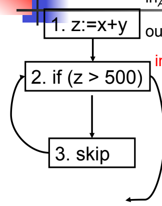
- Theorem: The worklist algorithm computes a solution that satisfies the dataflow equations
- Why?
- Sketch of proof:

  Whenever **j** is processed, algorithms sets out(j) = $\mathbf{f_j}$(in(j)). Whenever out(j) changes, algorithm puts successors on the list, so in(j) = **V** { out(i) }.
  So final solution will satisfy equations.

Precision Argument

- Theorem: The algorithm computes the least solution of the dataflow equations.
  - Historically though, this solution is often called the maximal fixpoint solution (MFP)
  - I.e., For every node **j**, the worklist algorithm computes a solution **MFP(j)** = {in(j),out(j)}, such that every other solution {in'(j),out'(j)} of the dataflow equations is in(j) ≤ in'(j), out(j) ≤ out'(j)

Example



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

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

Many Applications!

- Static debugging
  - Memory errors in C/C++ programs
    - Memory leaks
    - Null pointer dereferences
    - Array-out-of-bound accesses
  - Concurrency errors in shared-memory apps
    - Data-races, atomicity violations, deadlocks
- Information flow (as known as taint analysis)

Many Applications!

- White-box testing: compute coverage
  - Control-flow-based testing
  - Data-flow-based testing
    - Intuitively, test each def-use chain
- Regression testing
  - Analyze changes and select regression tests that actually test changed code

Dataflow Analysis

- Classical technique
- Compared to Hoare logic, it captures state in a more coarse way
- Still relevant, many interesting problems are phrased in dataflow terms

Next Class

- MOP vs MFP solutions
- Two classical non-distributive dataflow analyses:
  - Constant propagation and
  - Points-to analysis