Lecture 6

Foundations of Data Flow Analysis

I. Meet operator

II. Transfer functions

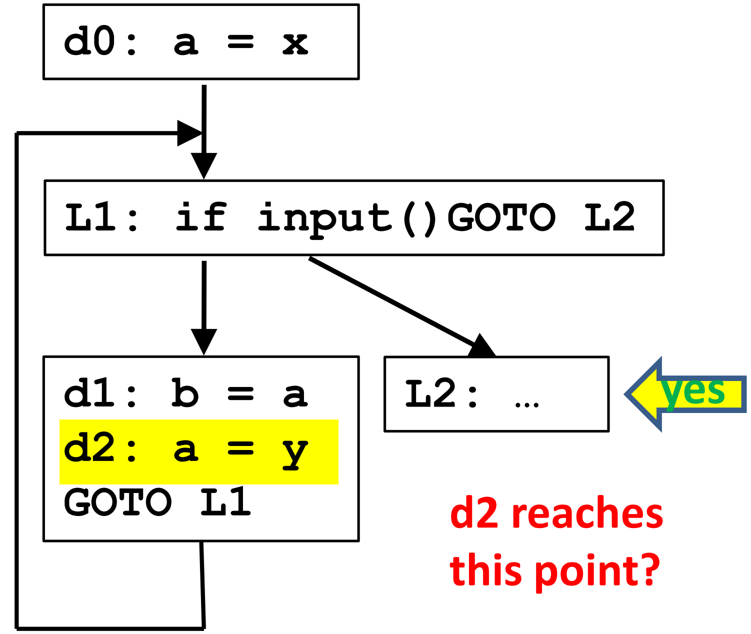
III. Correctness, Precision, Convergence

IV. Efficiency

ALSU 9.3

Review: Reaching Definitions

- A definition **`d`** **reaches** a point *p* if
  - **there exists** a path from the point immediately following *d* to *p* such that *d* is not killed (overwritten) along that path.
- **A basic block b can**
  - generate new definitions: **Gen[b]**
    - set of definitions in b that reach end of b
  - propagate incoming definitions: **in**[b] - **Kill[b]**,
    - where Kill[b]= set of defs killed by defs in b



d2 reaches this point?

- **Forward analysis**
  - **transfer function** for block b:

    out[b] = Gen[b] U (in[b]-Kill[b])
- **meet** operator:

  in[b] = out[p_1] U out[p_2] U ... U out[p_n], where

  p_1, ..., p_n are all the predecessors of b

Review: Live Variable Analysis

- A variable **v** is **live** at point *p* if
  - the value of **v** is used along some path in the flow graph starting at *p*.
- **A basic block b can**
  - generate live variables: **Use[b]**
    - set of locally exposed uses in b
  - propagate incoming live variables: **out**[b] - **Def[b]**,
    - where Def[b]= set of variables defined in b.b.

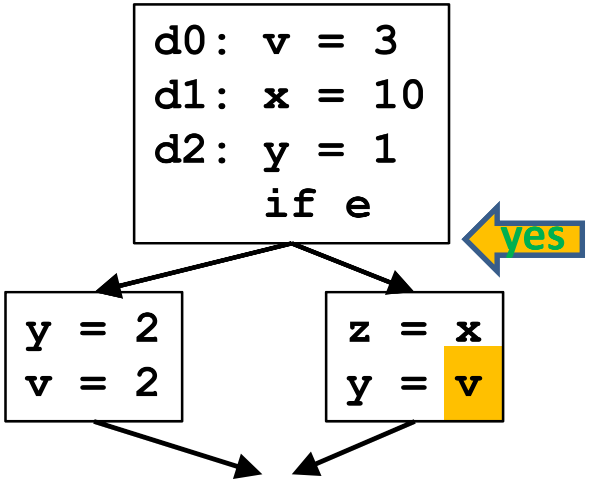
- **Backward analysis**
  - **transfer function** for block b:

    in[b] = Use[b] U (out[b]-Def[b])
- **meet** operator:

  out[b] = in[s_1] U in[s_2] U ... U in[s_n], where

  s_1, ..., s_n are all successors of b

v live at this point?



Review: Data Flow Analysis Framework

	Reaching Definitions	**Live Variables**
Domain	Sets of definitions	Sets of variables
Direction	forward: out[b] = f_b(in[b]) in[b] = $\wedge$ out[pred(b)]	backward: in[b] = f_b(out[b]) out[b] = $\wedge$ in[succ(b)]
Transfer function	$f_b(x) = Gen_b \cup (x - Kill_b)$	$f_b(x) = Use_b \cup (x - Def_b)$
Meet Operation ($\wedge$)	$\cup$	$\cup$
Boundary Condition	out[entry] = $\emptyset$	in[exit] = $\emptyset$
Initial interior points	out[b] = $\emptyset$	in[b] = $\emptyset$

Other Data Flow Analysis problems fit into this general framework, e.g., Available Expressions & Constant Propagation (Lecture 7)

A Unified Framework

- **Data flow problems are defined by**
  - Domain of values: V
  - Meet operator ($V \wedge V \rightarrow V$), initial value
  - A set of transfer functions ($V \rightarrow V$)

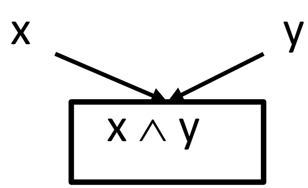
- **Usefulness of unified framework**
  - To answer questions such as
    correctness, precision, convergence, speed of convergence
    for a family of problems
    - If meet operators and transfer functions have properties X, then we know Y about the above.
  - Reuse code

Overview: A Check List for Data Flow Problems

- **Semi-lattice**
  - set of values
  - meet operator
  - top, bottom
  - finite descending chain?
- **Transfer functions**
  - function of each basic block
  - monotone
  - distributive?
- **Algorithm**
  - initialization step (entry/exit, other nodes)
  - visit order: rPostOrder
  - depth of the graph

I. Meet Operator

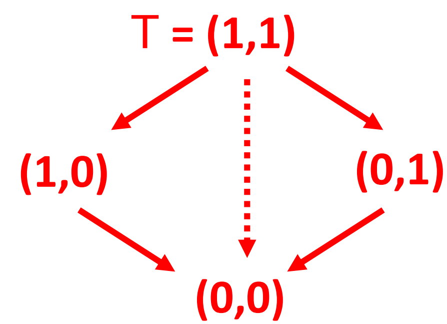
- **Properties of the meet operator**
  - commutative: $x \wedge y = y \wedge x$



  - idempotent: $x \wedge x = x$
  - associative: $x \wedge (y \wedge z) = (x \wedge y) \wedge z$
  - there is a Top element T such that $x \wedge T = x$

- **Meet operator defines a partial ordering on values**
  - $x \leq y$ if and only if $x \wedge y = x$
    - Transitivity: if $x \leq y$ and $y \leq z$ then $x \leq z$
    - Antisymmetry: if $x \leq y$ and $y \leq x$ then $x = y$
    - Reflexivity: $x \leq x$

Partial Order

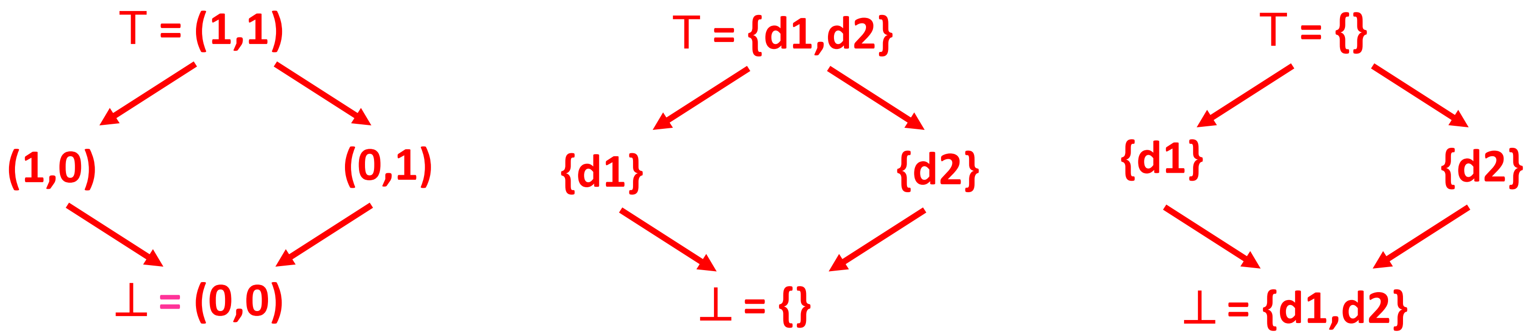


Meet Operator:
Elementwise-min

Note: $x < y$ is depicted as $y \rightarrow x$ in diagram

Note: typically show only minimal (i.e., transitively reduced) set of edges.
[not the dashed edge above]

Partial Order



Meet Operator: Elementwise-min

Meet Operator? **Intersection**

Meet Operator? **Union**

- Top and Bottom elements
  - Top T such that: **$x \wedge T = x$**
  - Bottom ⊥ such that: **$x \wedge \perp = \perp$**
- Values and meet operator in a data flow problem define a semi-lattice:
  - there exists a T, but not necessarily a ⊥.
- x, y are ordered: $x \leq y$ then $x \wedge y = x$
- what if x and y are not ordered?
  - $x \wedge y \leq x$, $x \wedge y \leq y$, and if $w \leq x$, $w \leq y$, then $w \leq x \wedge y$

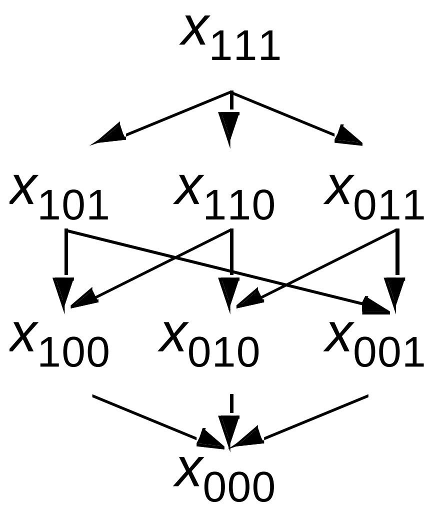
Note: $x < y$ is depicted as $y \rightarrow x$ in diagram

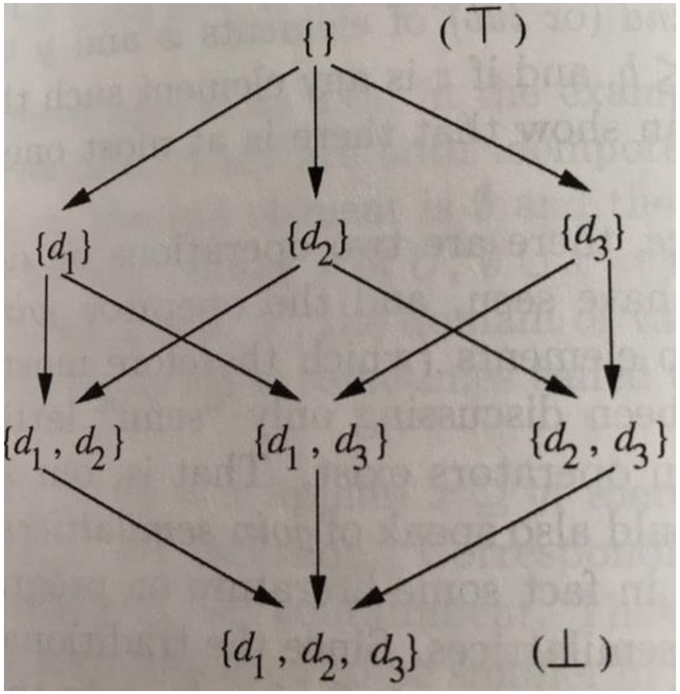
One vs. All Variables/Definitions

- **Lattice for each variable: e.g. intersection**

$$\begin{tikzcd} 1 \arrow[d] \\ 0 \end{tikzcd}$$

- **Lattice for three variables for intersection:**





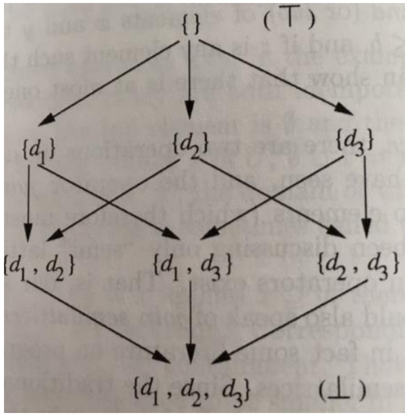
How many variables? **3**
Meet operator? **union**

Carnegie Mellon

Descending Chain

- **Definition**
  - The **height** of a lattice is the largest number of **> relations** that will fit in a descending chain.

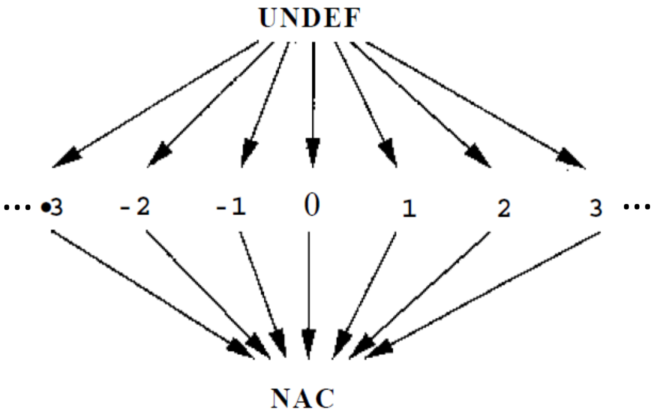
$$x_0 > x_1 > x_2 > \ldots$$



- **Height of values in reaching definitions?**

  Height=n, where n is the number of definitions

- **Important property: finite descending chain**
  - **Can an infinite lattice have a finite descending chain?** yes
  - **Example: Constant Propagation/Folding**
    - To determine if an integer variable is a constant
  - **Domain of values:**
    - undef, ... -1, 0, 1, 2, ..., not-a-constant



II. Transfer Functions

e.g., out[b] = Gen[b] U (in[b]-Kill[b])

- **Basic Properties *f:* V → V**
  - Has an identity function
    - There exists an *f* such that *f* (x) = x, for all x.
  - Closed under composition
    - if $f_1, f_2 \in F$, then $f_1 \cdot f_2 \in F$

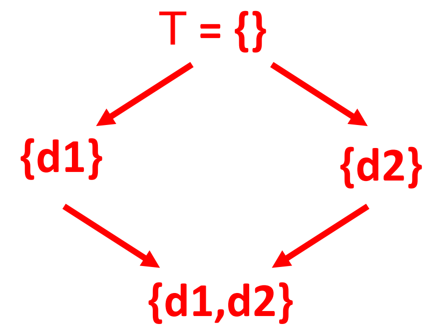
Monotonicity

Transfer function *f*: V → V
e.g., out[b] = Gen[b] U (in[b]-Kill[b])

- A framework ($F, V, \wedge$) is monotone if and only if
  - $x \leq y$ implies $f(x) \leq f(y)$
  - i.e. a "smaller or equal" input to the same function will always give a "smaller or equal" output
- Equivalently, a framework ($F, V, \wedge$) is monotone if and only if
  - $f(x \wedge y) \leq f(x) \wedge f(y)$
  - i.e. merge input, then apply *f* is **small than or equal to** apply the transfer function individually and then merge the result

Example: Reaching Definitions is Monotone

- **Reaching definitions: f(x) = Gen ∪ (x - Kill), ∧ = ∪**
  - Definition 1: x ≤ y implies f(x) ≤ f(y)
    - x ≤ y implies (x - Kill) ≤ (y - Kill)
      implies Gen ∪ (x - Kill) ≤ Gen ∪ (y - Kill)
  - Definition 2: f(x ∧ y) ≤ f(x) ∧ f(y)
    - (Gen ∪ ((x ∪ y) - Kill))
      = (Gen ∪ (x - Kill)) ∪ (Gen ∪ (y - Kill))



Meet Operator: Union

$[x < y \text{ iff } y \rightarrow x]$

Union: y is a subset of x

- **Note: Monotone framework does not mean that f(x) ≤ x**
    - E.g., consider reaching definitions, where d_1 and d_2 define the same variable
    - Then the transfer function f(x) for a basic block that defines only d_1 has

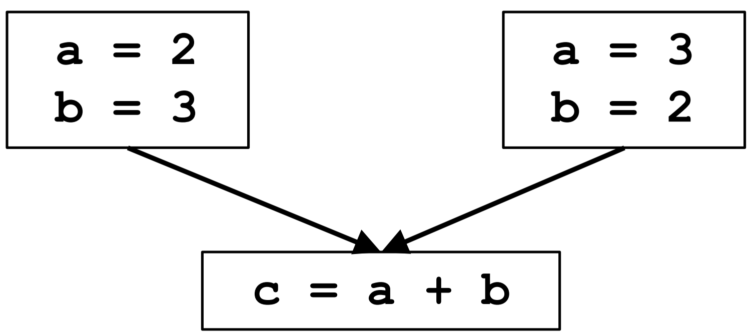
      Gen = $\{d_1\}$ and Kill = $\{d_2\}$
    - Let x= $\{d_2\}$. Then f(x) = $\{d_1\}$ which is unordered w.r.t. x= $\{d_2\}$.

- **If input(second iteration) ≤ input(first iteration)**
    - result(second iteration) ≤ result(first iteration)

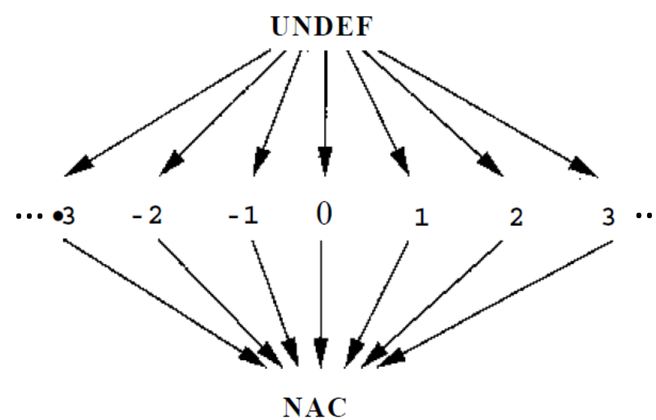
Distributivity

- A framework (F, V, $\wedge$) is **distributive** if and only if
  - $f(x \wedge y) = f(x) \wedge f(y)$
  - i.e., merge input, then apply f is **equal to** apply the transfer function individually then merge result

- Is Reaching Definitions distributive? **yes**
- Is Constant Propagation distributive? **no**



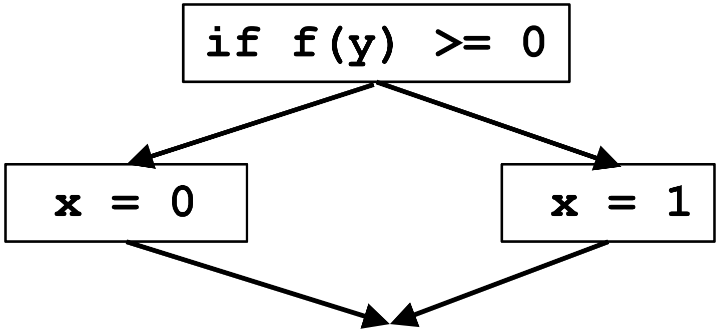
Consider c:
$f(x) \wedge f(y) =$ **5**
$f(x \wedge y) =$ **NAC**



III. Data Flow Analysis

- **Definition**
  - Let $f_1, ..., f_m : \in F$, where f_i is the transfer function for node i
    - $f_p = f_{n_k} \cdot ... \cdot f_{n_1}$, where p is a path through nodes $n_1, ..., n_k$
    - f_p = identify function if p is an empty path
- **Perfect data flow answer:**
  - For each node n:

    $\wedge f_{p_i}(\top)$, for all possibly executed paths p_i in the program reaching n.



If f(y) is always non-negative then right path never taken

- **In general: Determining all possibly executed paths is undecidable**

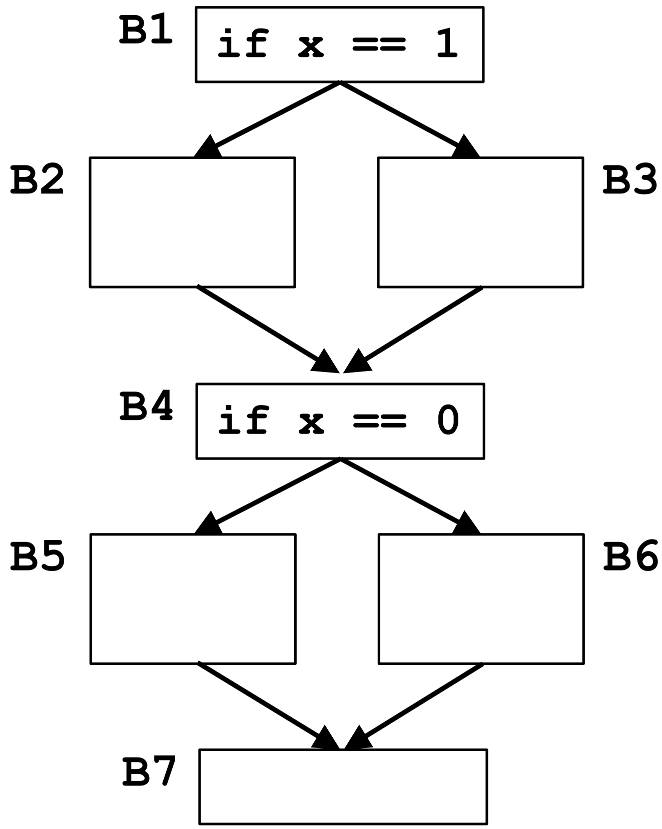
Meet-Over-Paths (MOP)

- **Err in the conservative direction**
- **Meet-Over-Paths (MOP):**
  - For each node *n:*

    $MOP(n) = \wedge f_{p_i}(T)$, for all paths p_i in data flow graph reaching n

  - a path exists as long there is an edge in the code
  - MOP = Perfect-Solution $\wedge$ Solution-to-Unexecuted-Paths
  - MOP $\leq$ Perfect-Solution
  - Considers more paths than necessary, hence solution is **conservative**
    - Meet = union: Definition may reach; Variable may be live
    - Meet = intersection: Expression is always available even when consider extra paths
  - Considering too few paths (> Perfect-Solution) would not be **safe!**
- **Desirable solution: as close to MOP as possible**

Example: MOP considers more paths than Perfect



Perfect considers only which 2 paths?

B1-B2-B4-B6-B7 (i.e., x=1)
B1-B3-B4-B5-B7 (i.e., x=0)

MOP also considers which other (unexecuted) paths?

B1-B2-B4-B5-B7
B1-B3-B4-B6-B7

What changes if x $\in \{0,1,2\}$ **?**

B1-B3-B4-B6-B7 is also considered by Perfect

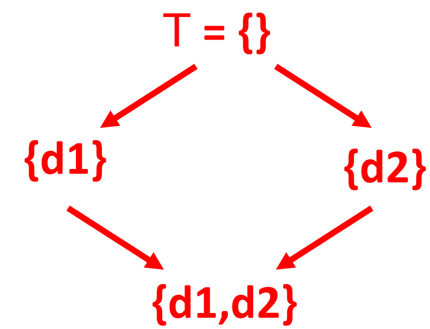
Assume x $\in \{0,1\}$ and B2 & B3 do not update x

Solving Data Flow Equations

- **Framework** (*F, V,* $\wedge$) defines set of equations relating in[b]'s and out[b]'s
- **Any solution satisfying equations = Fixed Point Solution (FP)**
- **Iterative algorithm for forward analysis** (backward analysis case is symmetric)
  - initializes out[b] to T for all b
  - if framework is monotone & algorithm converges, then it computes Maximum Fixed Point (MFP):
    - MFP is the largest of all solutions to equations (in any other solution, the values of IN[b] and OUT[b] are ≤ the corresponding values of the MFP)
- **Properties:**
  - FP ≤ MFP ≤ MOP ≤ Perfect-solution
  - FP, MFP are safe
  - If monotone & converges, then in[b] ≤ MOP[b]

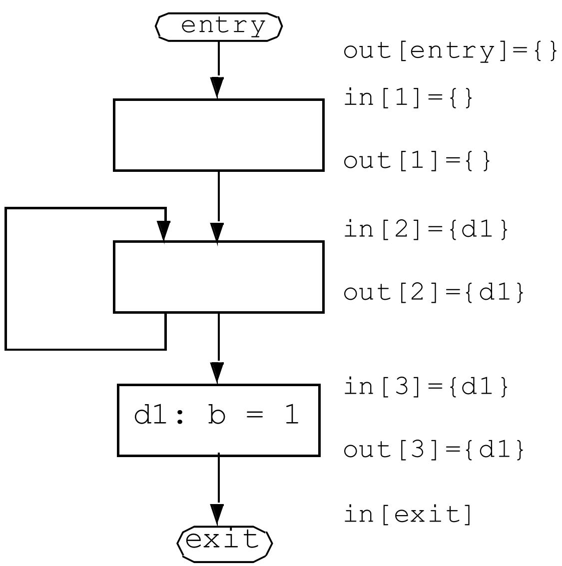
Solving Data Flow Equations

- **Example: Reaching definitions**
  - Values = {subsets of definitions}. Init out[b]= {}
  - Meet operator: in[b] = ∪ out[*p*], for all predecessors *p* of b
  - Transfer functions: out[b] = $gen_b \cup (in[b] - kill_b)$
- **Any solution satisfying equations = Fixed Point Solution (FP)**
- **Iterative algorithm computes** Maximum Fixed Point (MFP):
  – In any other solution, the values of IN[b] and OUT[b] are ≤ the corresponding values of the MFP

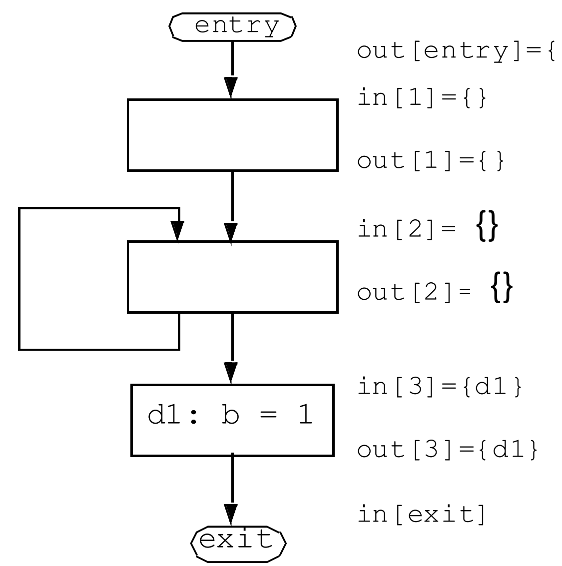


Meet: Union

$[x < y \text{ iff } y \rightarrow x]$



FP ≤ MFP



Carnegie Mellon

Partial Correctness of Algorithm

- **If data flow framework is monotone (i.e., $x \leq y$ implies $f(x) \leq f(y)$) then if the algorithm converges, IN[b] $\leq$ MOP[b]**
- **Proof: Induction on path lengths**
  - Define IN[entry] = OUT[entry]
    and transfer function of entry = Identity function
  - Base case: path of length 0
    - Proper initialization of IN[entry]
  - If true for path of length k, $p_k = (n_1, \ldots, n_k)$, then
    true for path of length k+1: $p_{k+1} = (n_1, \ldots, n_{k+1})$
    - Assume: $IN[n_k] \leq f_{n_{k-1}}(f_{n_{k-2}}(\ldots f_{n_1}(IN[entry])))$
    - $IN[n_{k+1}] = OUT[n_k] \wedge \ldots$

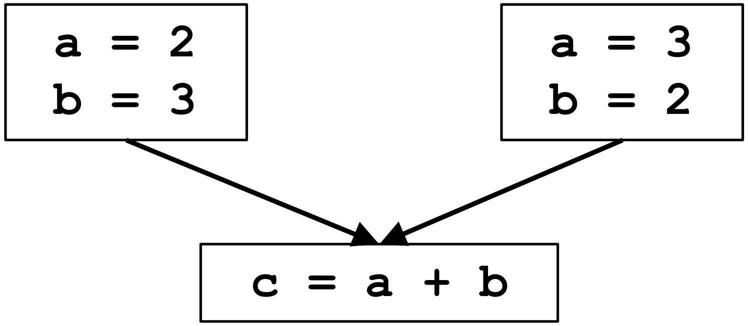
$$\leq OUT[n_k] = f_{n_k}(IN[n_k])$$

$$\leq f_{n_k}(f_{n_{k-1}}(\ldots f_{n_1}(IN[entry])))$$ by inductive assumption & monotonicity

Precision

- **If data flow framework is distributive (i.e., f(x ∧ y) = f(x) ∧ f(y)) then if the algorithm converges, IN[b] = MOP[b]**

- Monotone but not distributive: behaves as if there are additional paths



Additional Property to Guarantee Convergence

- **Data flow framework (monotone) converges if there is a finite descending chain**

- For each variable IN[b], OUT[b], consider the sequence of values set to each variable across iterations:

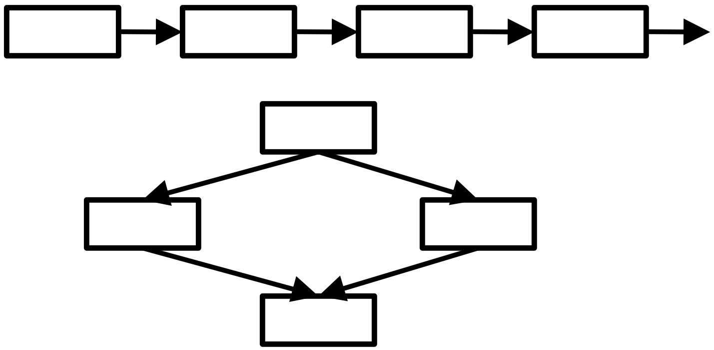
  – if sequence for in[b] is monotonically decreasing
    - sequence for out[b] is monotonically decreasing
      - (out[b] initialized to T)

  – if sequence for out[b] is monotonically decreasing
    - sequence of in[b] is monotonically decreasing

- Must be at least one out[b] change to warrant an additional iteration
  – Thus, guaranteed to converge after at most
  (height of lattice) x (number of nodes in flow graph) iterations

IV. Speed of Convergence

- Speed of convergence depends on order of node visits



- Reverse “direction” for backward flow problems

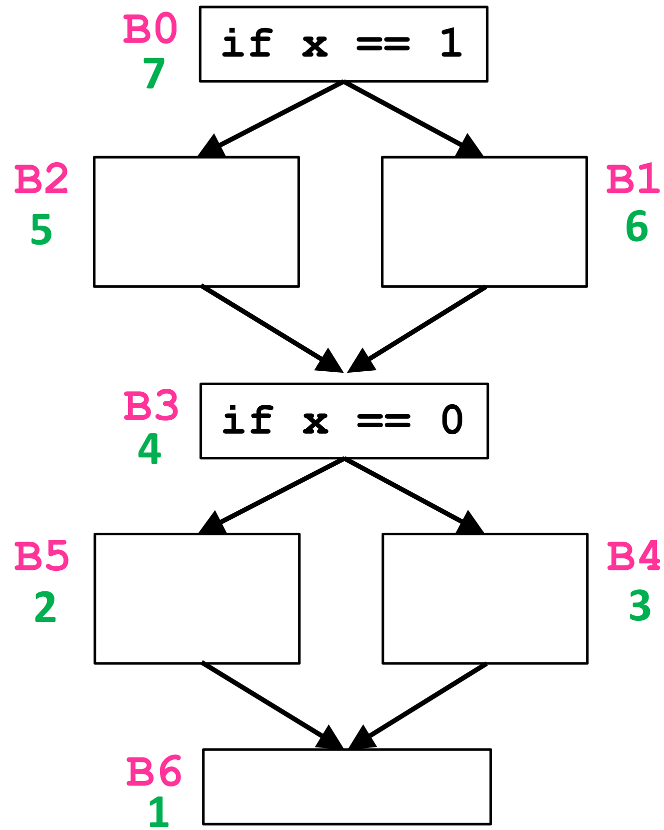
Reverse Postorder

- **Step 1: depth-first post order**

```
main() {
   count = 1;
   Visit(root);
}
Visit(n) {
   for each successor s that
               has not been visited
      Visit(s);
   PostOrder(n) = count;
   count = count+1;
}
```

- **Step 2: reverse order**

```
For each node i
   rPostOrder(i)= NumNodes - PostOrder(i)
```



(order among siblings unimportant)

Depth-First Iterative Algorithm (forward)

```
input: control flow graph CFG = (N, E, Entry, Exit)

/* Initialize */
    out[entry] = init_value
    For all nodes i
       out[i] = ⊤
    Change = True

/* iterate */
    While Change {
       Change = False
       For each node i in rPostOrder {
          in[i] = ∧(out[p]), for all predecessors p of i
          oldout = out[i]
          out[i] = f_i(in[i])
          if oldout ≠ out[i]
             Change = True
       }
    }
```

[ALSU 9.6.7]

Speed of Convergence

- **If cycles do not add information***
  - information can flow in one pass down nodes of increasing order number:
    - e.g., $\underbrace{1 \text{ -> } 4 \text{ -> } 5 \text{ -> } 7}_{\text{first pass}}$ -> 2 -> 6 ...
  - passes determined by number of back edges in the path
    - essentially the nesting depth of the graph
  - Number of iterations = number of back edges in any acyclic path + 2
    - (2 are necessary even for acyclic CFGs)
    - (2 not 1 since need a last pass where nothing changed)

- **What is the depth?**
  - corresponds to depth of intervals for "reducible" graphs
  - in real programs: average of 2.75

* E.g., if a defn *d* in node n_1 reaches a node n_k along a path that contains a cycle (i.e., a repeated node), then the cycle can be removed to form a shorter path from n_1 to n_k such that *d* reaches n_k.

Summary: A Check List for Data Flow Problems

- **Semi-lattice**
  - set of values
  - meet operator
  - top, bottom
  - finite descending chain?
- **Transfer functions**
  - function of each basic block
  - monotone
  - distributive?
- **Algorithm**
  - initialization step (entry/exit, other nodes)
  - visit order: rPostOrder
  - depth of the graph

Today's Class

I. Meet operator

II. Transfer functions

III. Correctness, Precision, Convergence

IV. Efficiency

Wednesday's Class

- Global common subexpression elimination
  - ALSU 9.2.6
- Constant propagation/folding
  - ALSU 9.4