Data Flow Analysis

Lecture 21

Data Flow Analysis

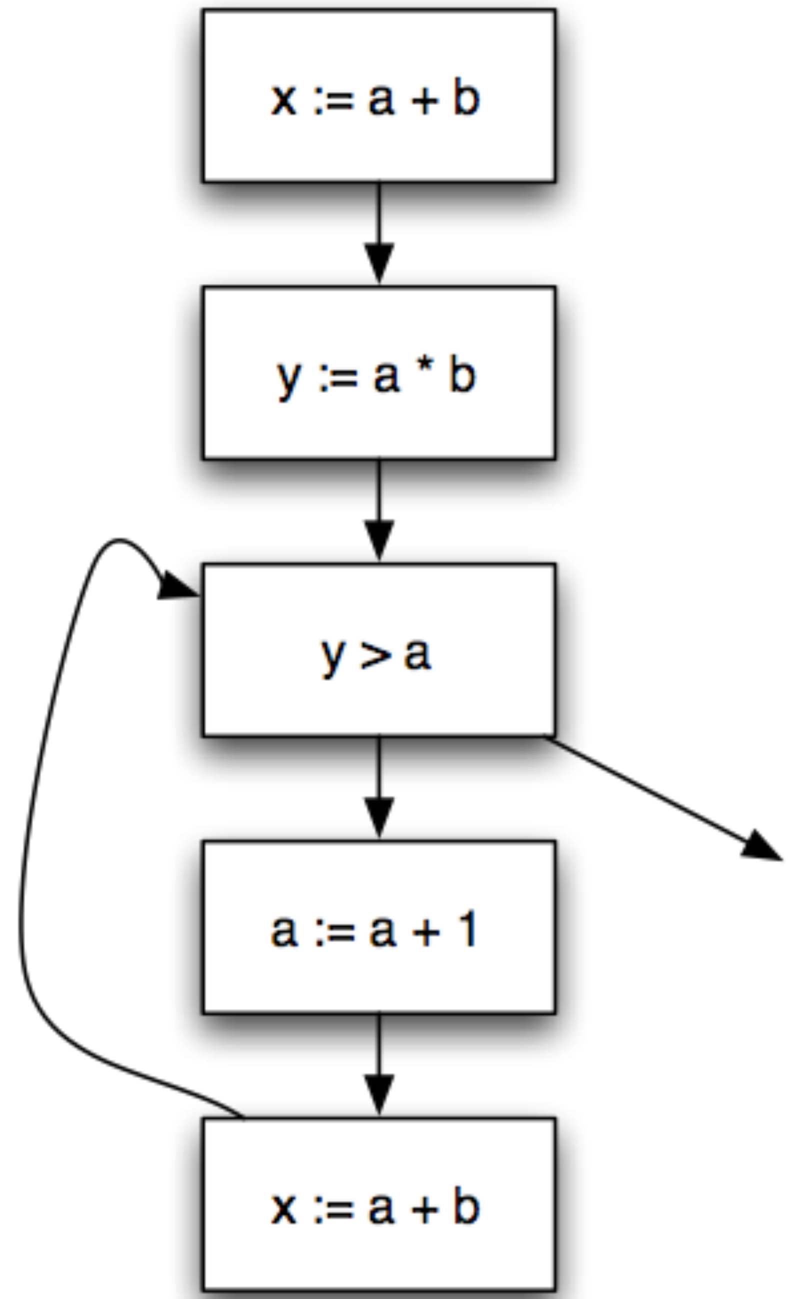
- A framework for proving facts about programs
- Reasons about lots of little facts
- Little or no interaction between facts
  - Works best on properties about *how* program computes
- Based on all paths through program
  - Including infeasible paths

Available Expressions

- An expression e is available at program point p if
  - e is computed on every path to p, and
  - the value of e has not changed since the last time e is computed on p
- Optimization
  - If an expression is available, need not be recomputed
    - (At least, if it's still in a register somewhere)

Data Flow Facts

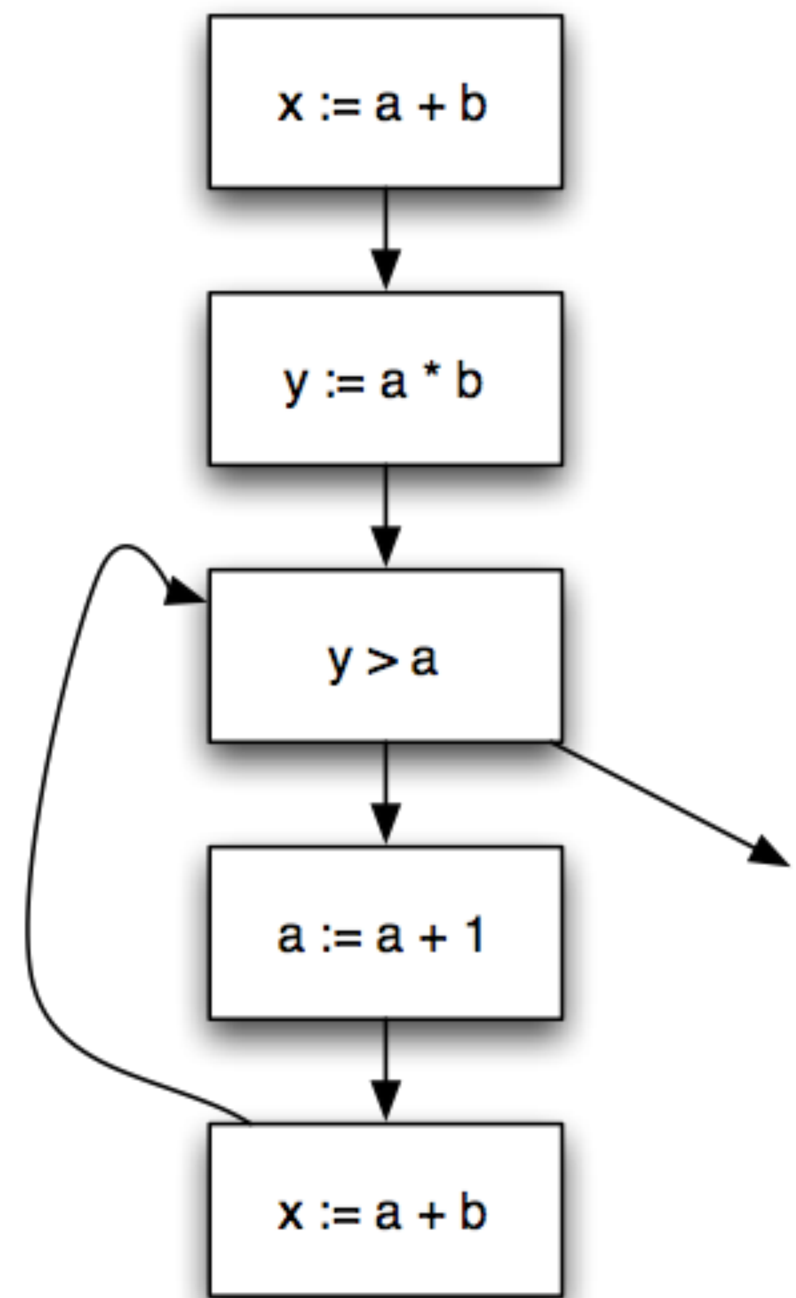
- Is expression e available?
- Facts:
  - a + b is available
  - a * b is available
  - a + 1 is available



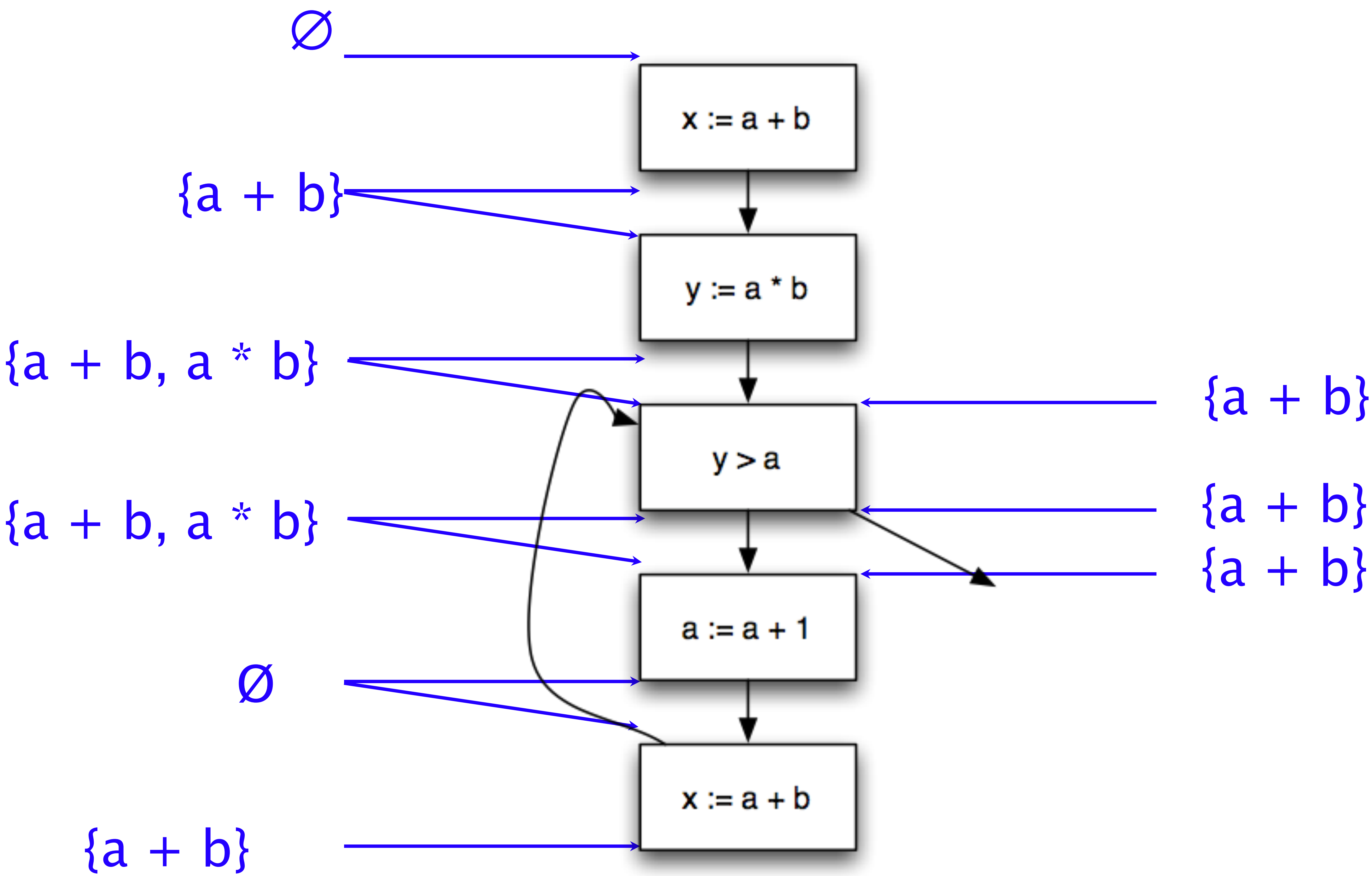
Gen and Kill

- What is the effect of each statement on the set of facts?

Stmt	Gen	Kill
x := a + b	a + b	
y := a * b	a * b	
a := a + 1		a + 1, a + b, a * b



Computing Available Expressions



Terminology

- A *joint point* is a program point where two branches meet

- Available expressions is a *forward must* problem
  - Forward = Data flow from in to out
  - Must = At join point, property must hold on all paths that are joined

Data Flow Equations

- Let s be a statement
  - succ(s) = { immediate successor statements of s }
  - pred(s) = { immediate predecessor statements of s}
  - In(s) = facts at program point just before executing s
  - Out(s) = facts at program point just after executing s
- $\mathrm{In}(s) = \bigcap_{s' \in \mathrm{pred}(s)} \mathrm{Out}(s')$
- $\mathrm{Out}(s) = \mathrm{Gen}(s) \cup (\mathrm{In}(s) - \mathrm{Kill}(s))$
  - Note: These are also called *transfer functions*

Liveness Analysis

- A variable v is *live* at program point p if
  - v will be used on some execution path originating from p…
  - before v is overwritten
- Optimization
  - If a variable is not live, no need to keep it in a register
  - If variable is dead at assignment, can eliminate assignment

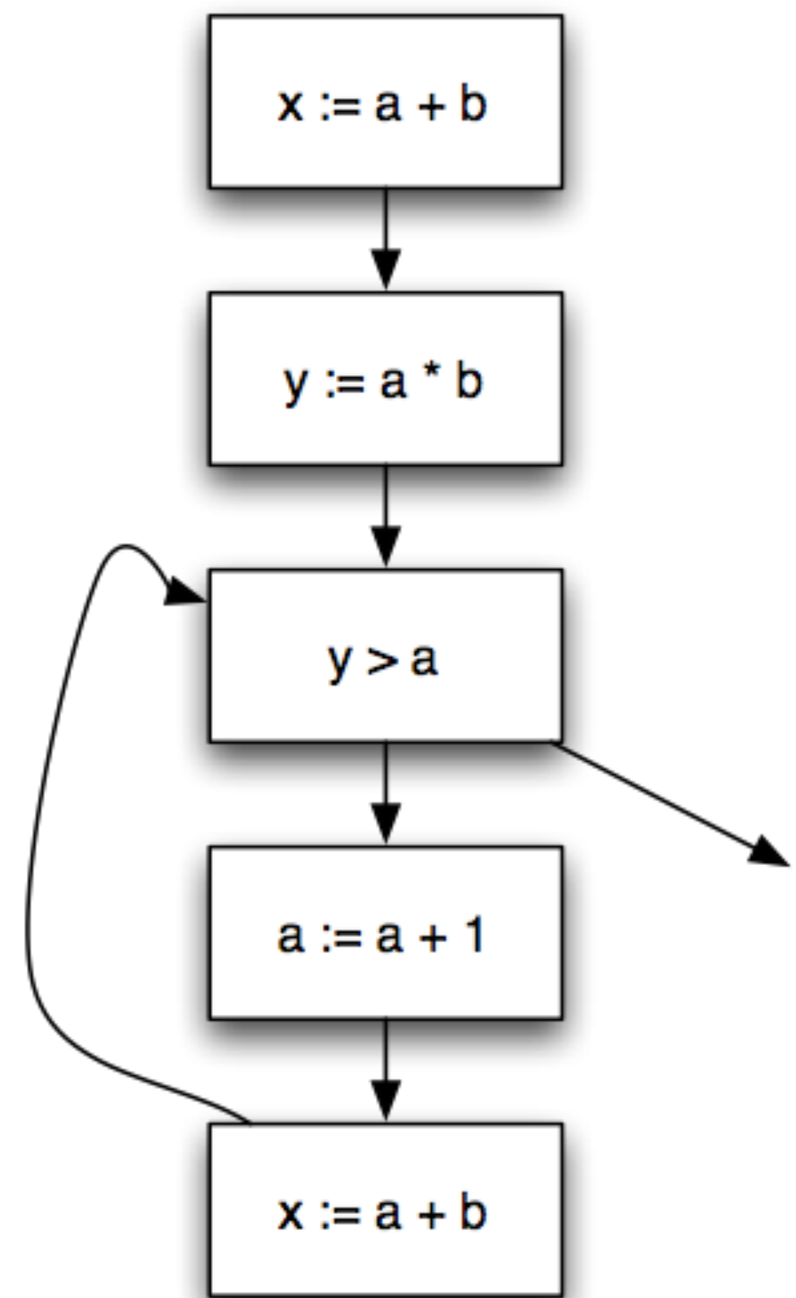
Data Flow Equations

- Available expressions is a forward must analysis
  - Data flow propagate in same dir as CFG edges
  - Expr is available only if available on all paths
- Liveness is a *backward may* problem
  - To know if variable live, need to look at future uses
  - Variable is live if used on some path
- $\mathrm{Out}(s) = \bigcup_{s' \in \mathrm{succ}(s)} \mathrm{In}(s')$
- $\mathrm{In}(s) = \mathrm{Gen}(s) \cup (\mathrm{Out}(s) - \mathrm{Kill}(s))$

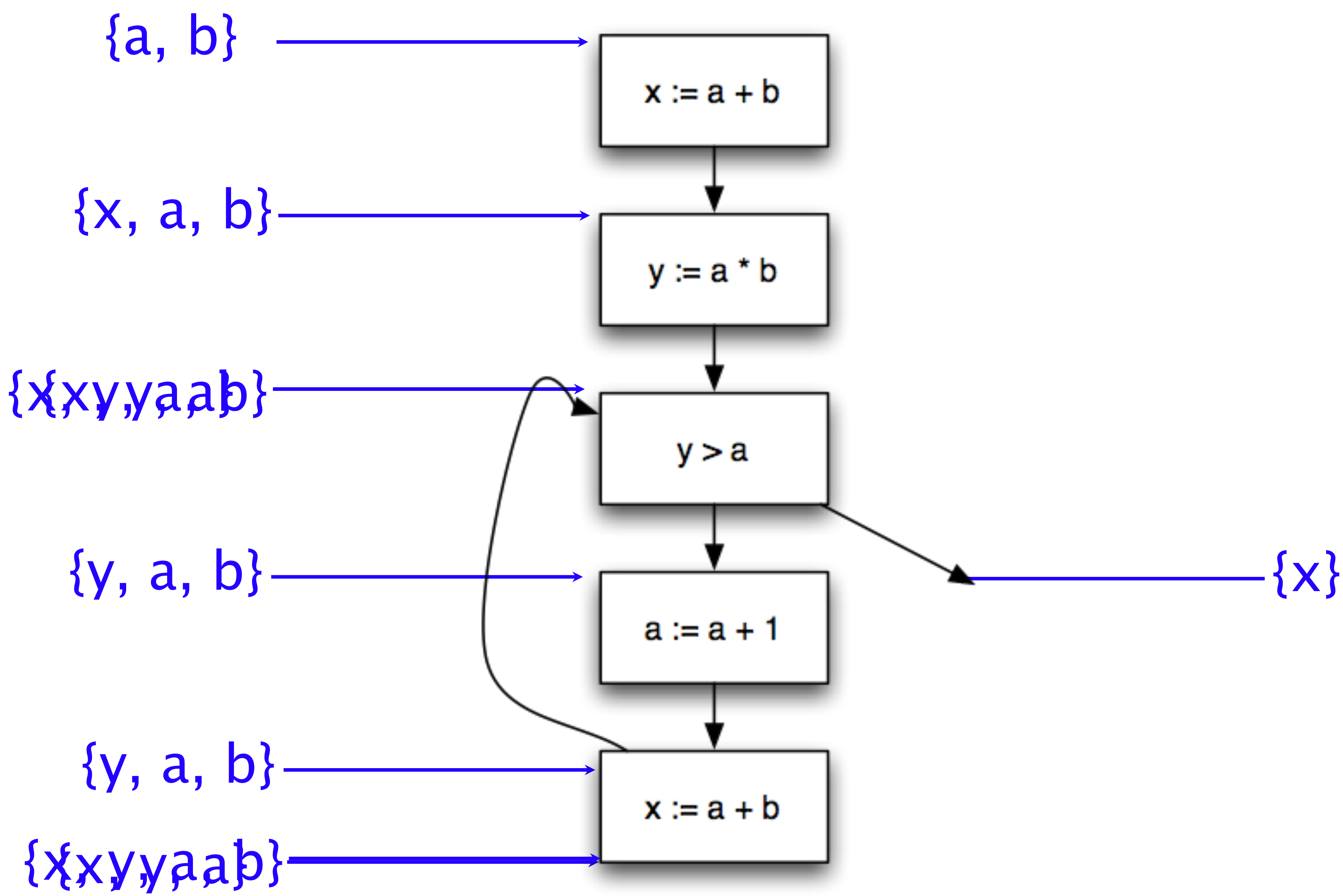
Gen and Kill

- What is the effect of each statement on the set of facts?

Stmt	Gen	Kill
x := a + b	a, b	x
y := a * b	a, b	y
y > a	a, y	
a := a + 1	a	a



Computing Live Variables



Very Busy Expressions

- An expression e is *very busy* at point p if
  - On every path from p, expression e is evaluated before the value of e is changed

- Optimization
  - Can hoist very busy expression computation

- What kind of problem?
  - Forward or backward? backward
  - May or must? must

Reaching Definitions

- A *definition* of a variable v is an assignment to v
- A definition of variable v reaches point p if
  - There is no intervening assignment to v
- Also called def-use information
- What kind of problem?
  - Forward or backward? forward
  - May or must? may

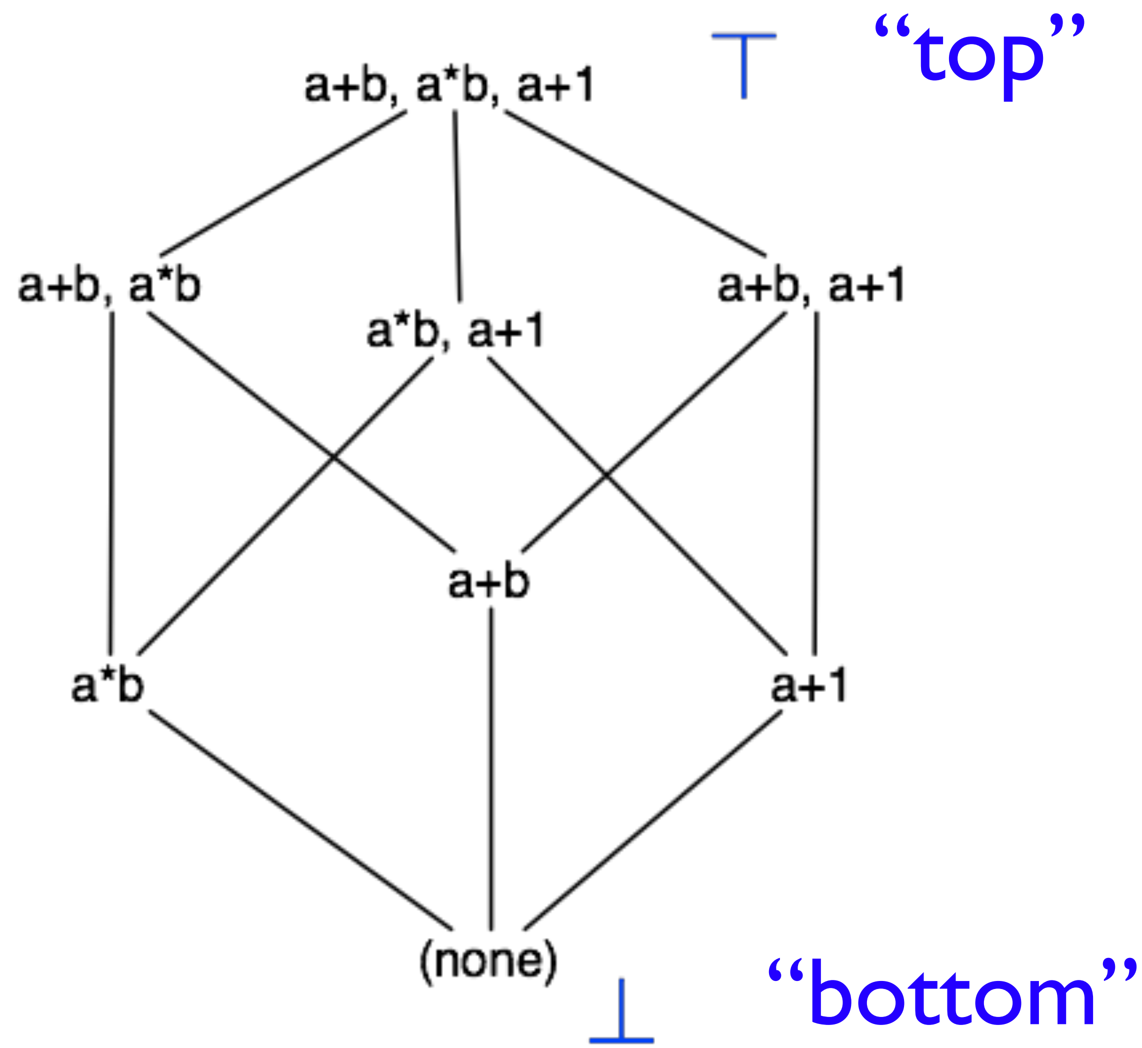
Space of Data Flow Analyses

	May	Must
Forward	Reaching definitions	Available expressions
Backward	Live variables	Very busy expressions

- Most data flow analyses can be classified this way
- Lots of literature on data flow analysis

Data Flow Facts and Lattices

- Typically, data flow facts form a lattice
  - Example: Available expressions



Partial Orders

- A partial order is a pair $(P, \leq)$ such that
  - $\leq \subseteq P \times P$
  - $\leq$ is reflexive: $x \leq x$
  - $\leq$ is anti-symmetric: $x \leq y$ and $y \leq x \Rightarrow x = y$
  - $\leq$ is transitive: $x \leq y$ and $y \leq z \Rightarrow x \leq z$

Lattices

- A partial order is a lattice if $\sqcap$ and $\sqcup$ are defined on any pair of elements:
  - $\sqcap$ is the *meet* or *greatest lower bound* operation:
    - $x \sqcap y \leq x$ and $x \sqcap y \leq y$
    - if $z \leq x$ and $z \leq y$, then $z \leq x \sqcap y$
  - $\sqcup$ is the *join* or *least upper bound* operation:
    - $x \leq x \sqcup y$ and $y \leq x \sqcup y$
    - if $x \leq z$ and $y \leq z$, then $x \sqcup y \leq z$

Lattices (cont'd)

- A finite partial order is a lattice if meet and join exist for every pair of elements
- A lattice has unique elements $\perp$ and $\top$ such that
  - $x \sqcap \perp = \perp$ $\quad$ $x \sqcup \perp = x$
  - $x \sqcap \top = x$ $\quad$ $x \sqcup \top = \top$
- In a lattice,

$$x \leq y \text{ iff } x \sqcap y = x$$
$$x \leq y \text{ iff } x \sqcup y = y$$

Forward Must Data Flow Algorithm

- Out(s) = Top for all statements s
  - // Slight acceleration: Could set Out(s) = Gen(s) ∪(Top - Kill(s))
- W := { all statements } (worklist)

repeat

Take s from W

In(s) := $\cap_{s' \in pred(s)}$ Out(s′)

temp := Gen(s) ∪ (In(s) - Kill(s))

if (temp != Out(s)) {

Out(s) := temp

W := W ∪ succ(s)

}

until W = ∅

Monotonicity

- A function f on a partial order is *monotonic* if

$$x \leq y \Rightarrow f(x) \leq f(y)$$

- Easy to check that operations to compute In and Out are monotonic
  - In(s) := $\cap_{s' \in \text{pred}(s)}$ Out(s')
  - temp := Gen(s) ∪ (In(s) - Kill(s))

- Putting these two together,
  - temp := $f_s(\sqcap_{s' \in \text{pred}(s)} Out(s'))$

Termination

- We know the algorithm terminates because
  - The lattice has finite height
  - The operations to compute In and Out are monotonic
  - On every iteration, we remove a statement from the worklist and/or move down the lattice

Forward Data Flow, Again

- $Out(s) = Top$ for all statements s
- W := { all statements } (worklist)

repeat

  Take s from W

  temp := $f_s(\sqcap_{s' \in pred(s)} Out(s'))$ (f_s monotonic *transfer fn*)

  if (temp != Out(s)) {

    Out(s) := temp

    W := W $\cup$ succ(s)

  }

until W = $\varnothing$

Lattices (P, $\leq$)

- Available expressions
  - P = sets of expressions
  - S1 $\sqcap$ S2 = S1 $\cap$ S2
  - Top = set of all expressions
- Reaching Definitions
  - P = set of definitions (assignment statements)
  - S1 $\sqcap$ S2 = S1 $\cup$ S2
  - Top = empty set

Fixpoints

- We always start with Top
  - Every expression is available, no defns reach this point
  - Most optimistic assumption
  - Strongest possible hypothesis
    - = true of fewest number of states
- Revise as we encounter contradictions
  - Always move down in the lattice (with meet)
- Result: A greatest fixpoint

Lattices (P, ≤), cont'd

- Live variables
  - P = sets of variables
  - $S1 \sqcap S2 = S1 \cup S2$
  - Top = empty set
- Very busy expressions
  - P = set of expressions
  - $S1 \sqcap S2 = S1 \cap S2$
  - Top = set of all expressions

Forward vs. Backward

```
Out(s) = Top  for all s
W := { all statements }
repeat
    Take s from W
    temp := f_s(⊓_{s′ ∈ pred(s)} Out(s′))
    if (temp != Out(s)) {
      Out(s) := temp
      W := W ∪ succ(s)
    }
until W = ∅
```

```
In(s) = Top  for all s
W := { all statements }
repeat
    Take s from W
    temp := f_s(⊓_{s′ ∈ succ(s)} In(s′))
    if (temp != In(s)) {
      In(s) := temp
      W := W ∪ pred(s)
    }
until W = ∅
```

Data Flow Analysis Summary

- Need to determine the information that should be computed at a node
- Need to determine how that information should flow from node to node
  - Backward or Forward
  - Union or Intersection
- Often there is more than one way to solve a problem
  - Can often be solved forward or backward, but usually one way is easier than the other