HARVARD
School of Engineering
and Applied Sciences

Dataflow analysis

CS252r Spring 2011

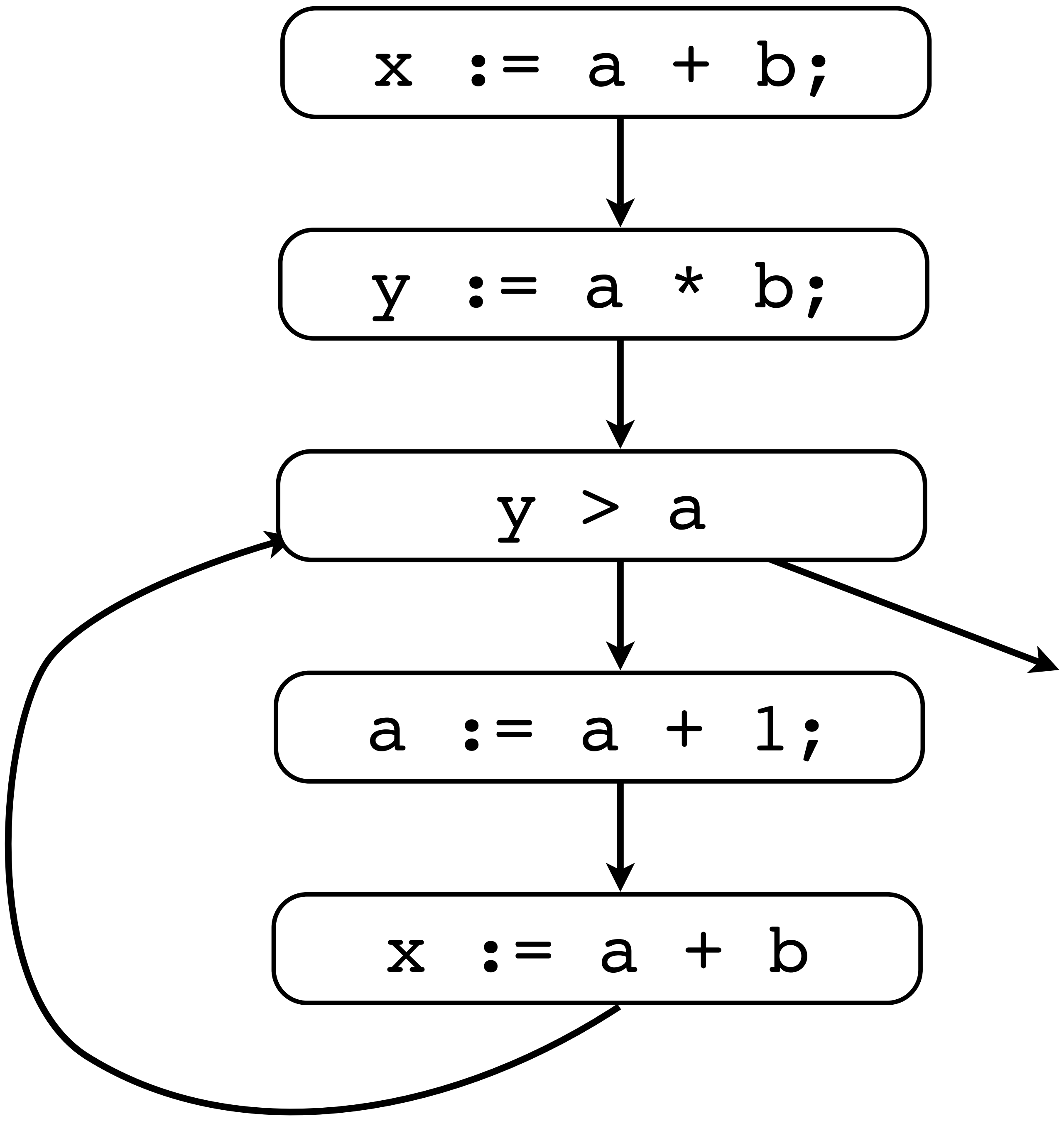
(Based on lecture notes by Jeff Foster)

Control flow graph

- A **control flow graph** is a representation of a program that makes certain analyses (including dataflow analyses) easier
- A directed graph where
  - Each node represents a statement
  - Edges represent control flow
- Statements may be
  - Assignments: x := y or x := y op z or x := op y
  - Branches: goto L or if b then goto L
  - etc.

Control-flow graph example

```
x := a + b;
y := a * b;
while (y > a) {
  a := a + 1;
  x := a + b
}
```

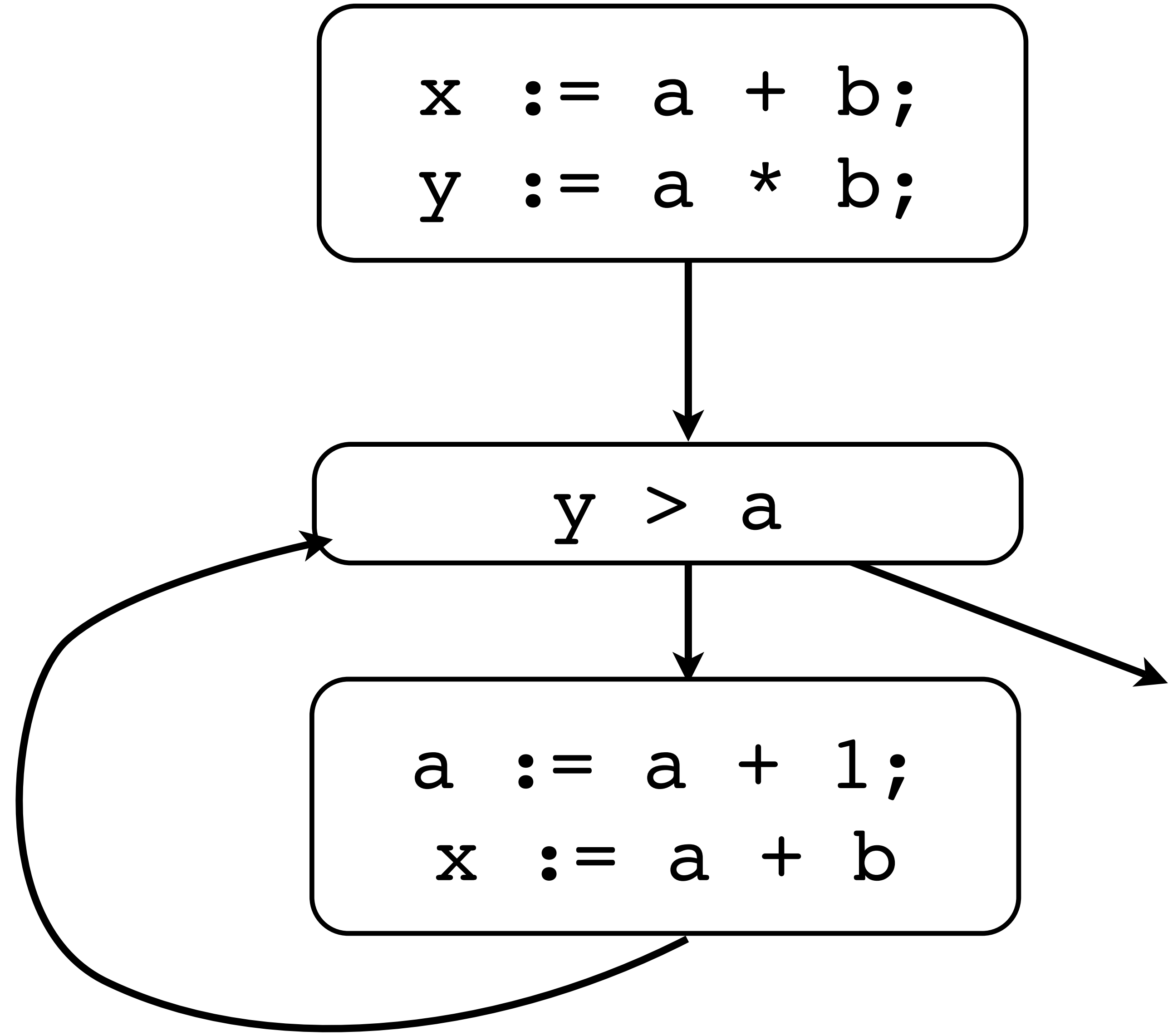


Variations on CFGs

- Usually don't include declarations (e.g., int x;) in the CFG
  - But there's usually something in the implementation
- May want a unique entry and exit node
  - Won't matter for the examples we give
- May group statements into **basic blocks**
  - A sequence of instructions with no branches into or out of the block

Control-flow graph with basic blocks

```
x := a + b;
y := a * b;
while (y > a) {
   a := a + 1;
   x := a + b
}
```

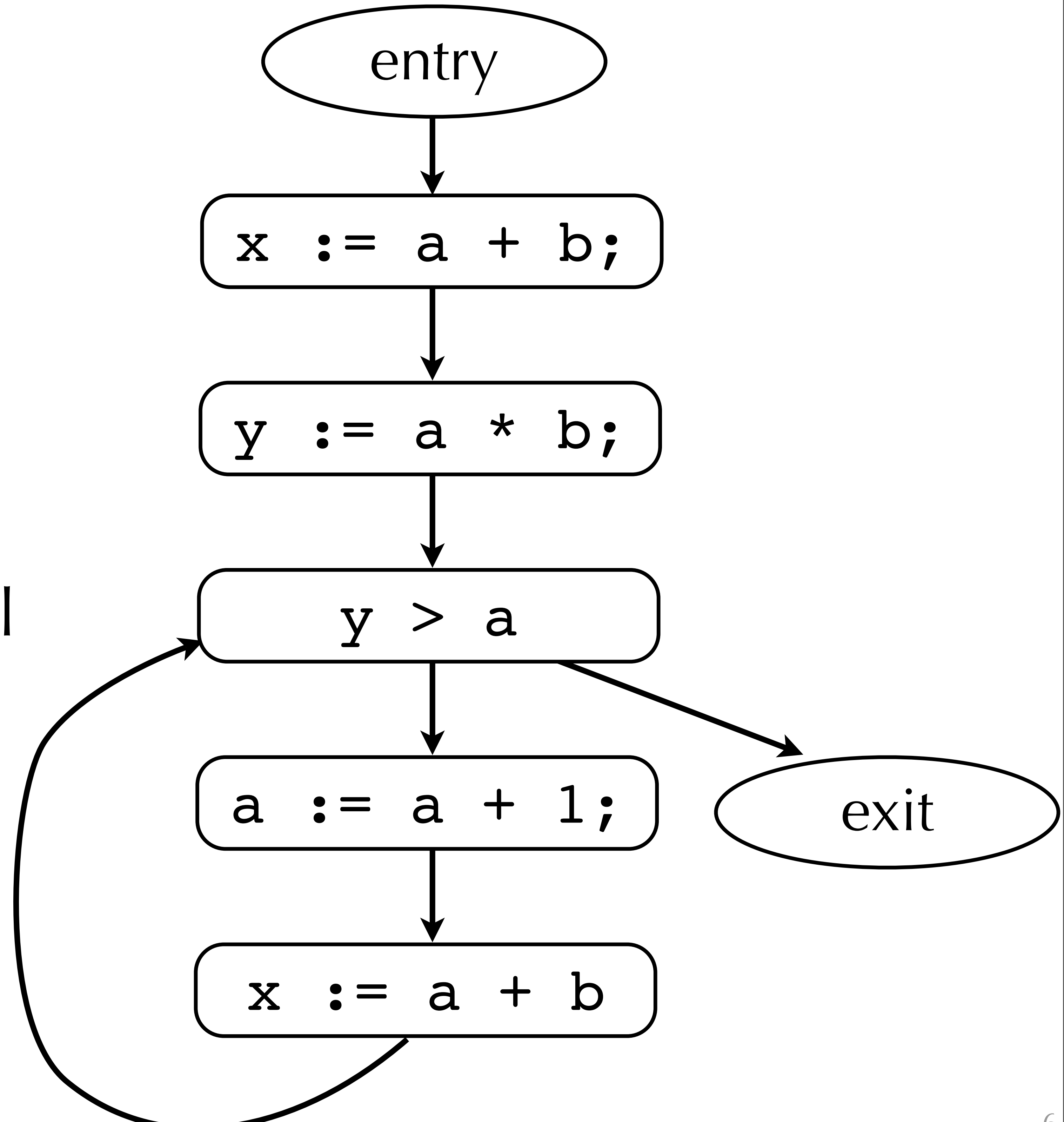


- Can lead to more efficient implementations
- More complicated to explain, so for the meantime we'll use single statement blocks

Graph example with entry and exit

```
x := a + b;
y := a * b;
while (y > a) {
   a := a + 1;
   x := a + b
}
```

- All nodes without a normal predecessor should be pointed to by entry
- All nodes with a successor should point to exit



CFG vs AST

- CFGs are much simpler than ASTs
- Fewer forms, less redundancy, only simple expressions
- But AST is a more faithful representation
  - CFGs introduce temporaries
  - Lose block structure of program
- ASTs are
  - Easier to report error + other messages
  - Easier to explain to programmer
  - Easier to unparse to produce readable code

Dataflow 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

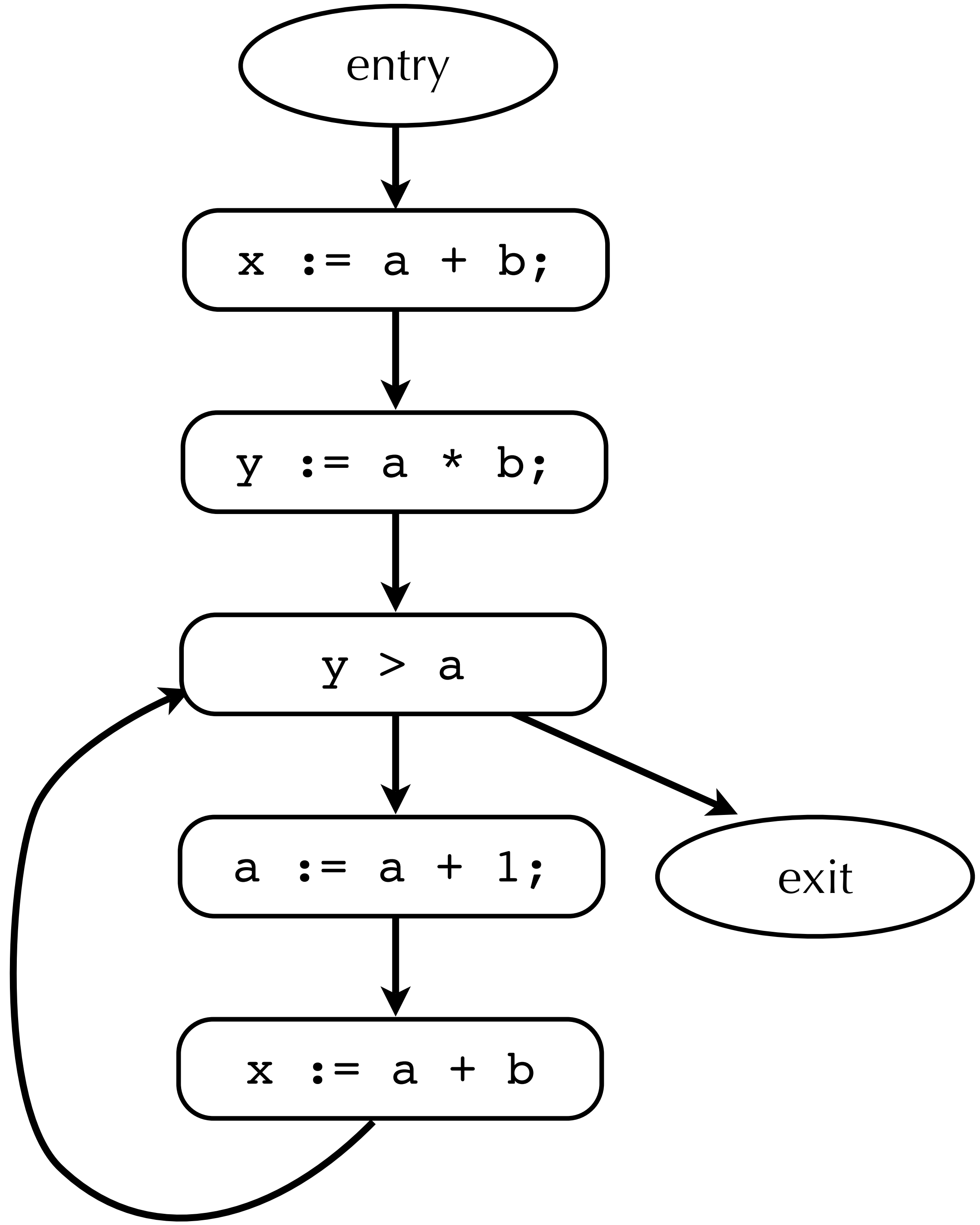
- Let's consider some dataflow analyses

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 was computed on the paths to p
- Available expressions can be used to optimize code
  - If an expression is available, don't need to recompute it (provided it is stored in a register somewhere)

Data flow facts

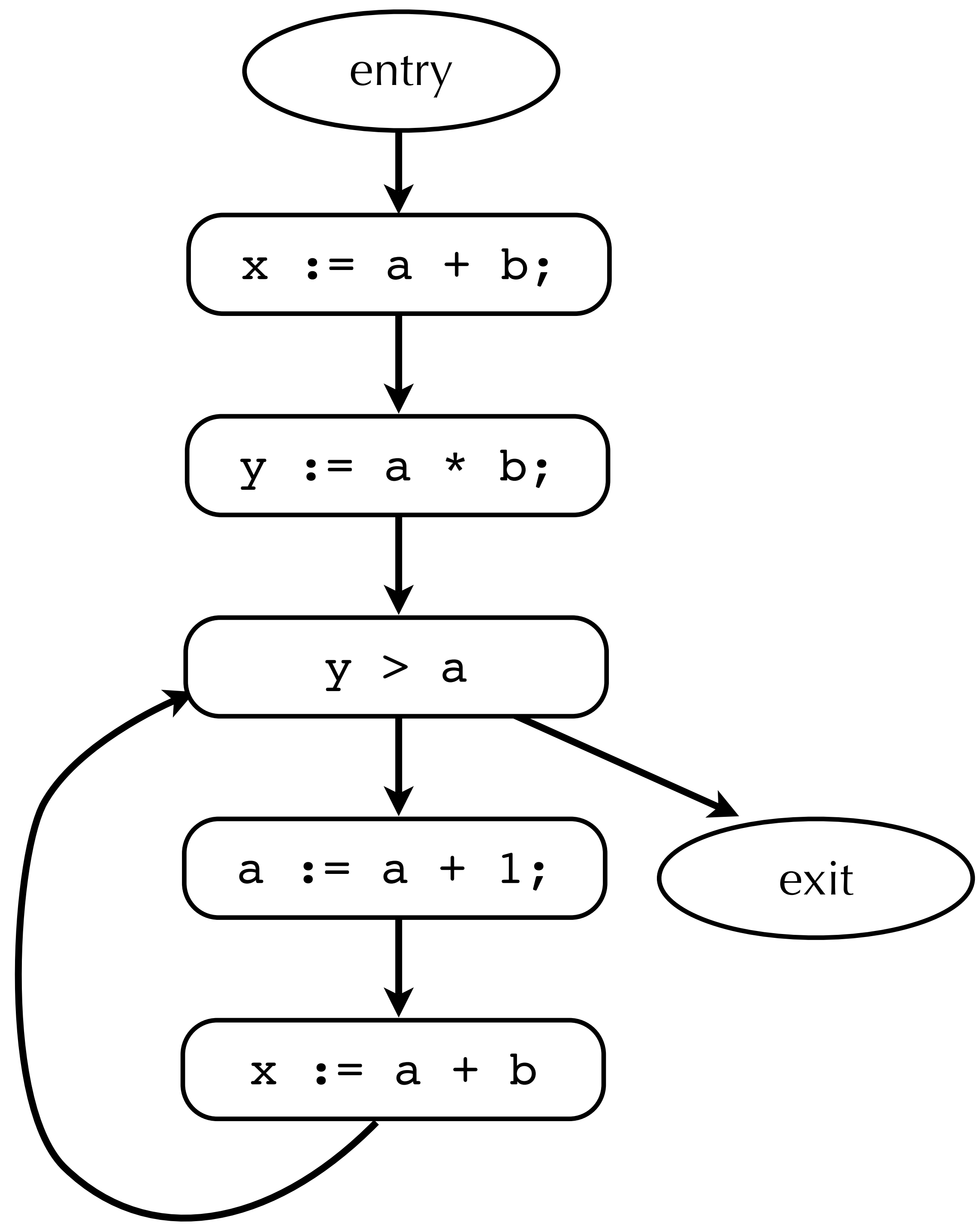
- Is expression e available?
- Facts
  - "a + b is available"
  - "a * b is available"
  - "a + 1 is available"
- For each program point, we will compute which facts hold.



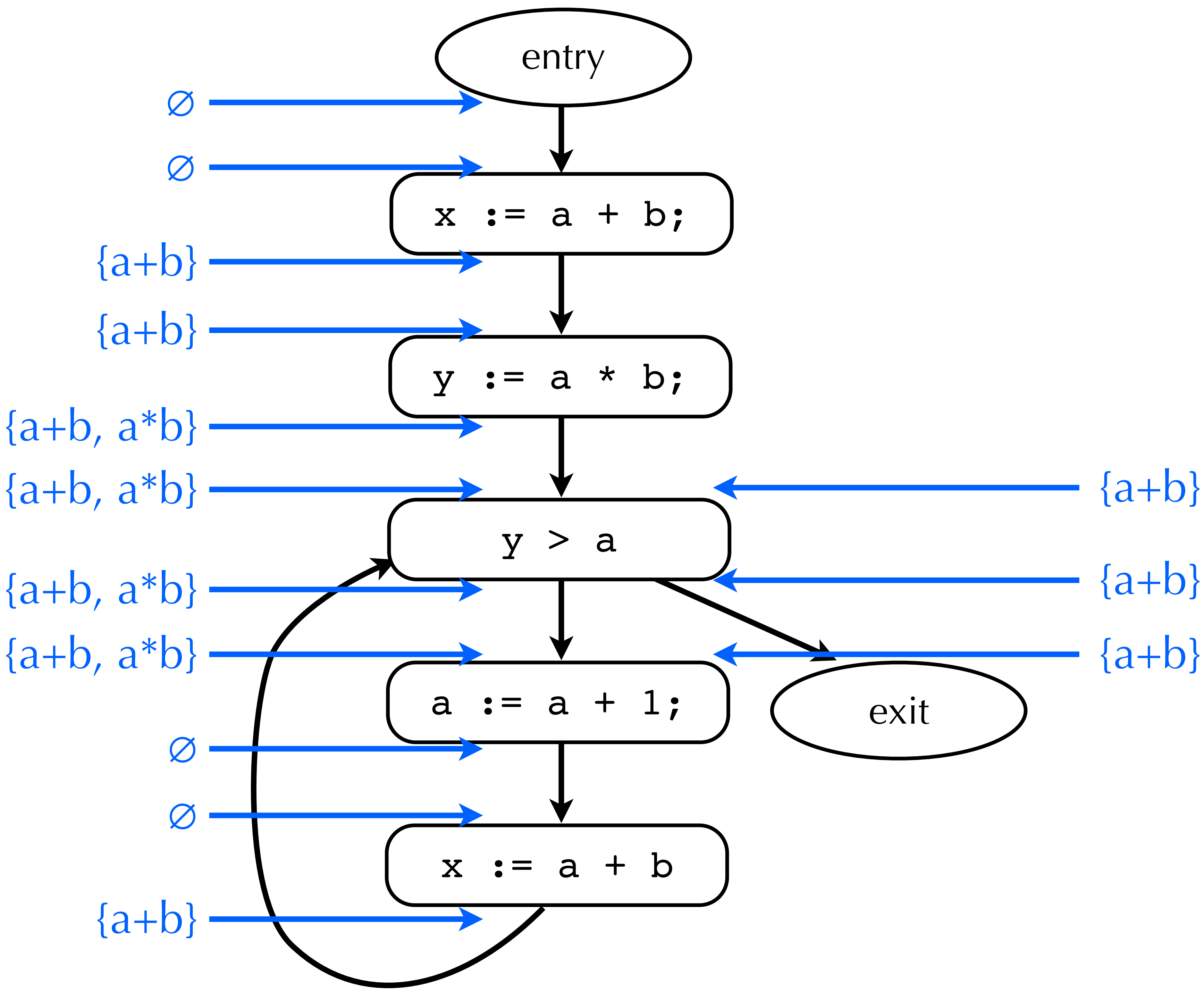
Gen and Kill

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

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



Computing available expressions



Terminology

- A **join point** is a program point where two or more branches meet
- Available expressions is a **forward must** analysis
  - Forward = Data flow from in to out
  - Must = At join points, only keep facts that hold on all paths that are joined

Data flow equations

- Let s be a statement
  - succs(s) = { immediate successor stmts of s }
  - preds(s) = { immediate predecessor stmts of s }
  - In(s) = program point just before executing s
  - Out(s) = program point just after executing s

- $\text{In}(s) = \bigcap_{s' \in \text{preds}(s)} \text{Out}(s')$
- $\text{Out}(s) = \text{Gen}(s) \cup (\text{In}(S) - \text{Kill}(s))$

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

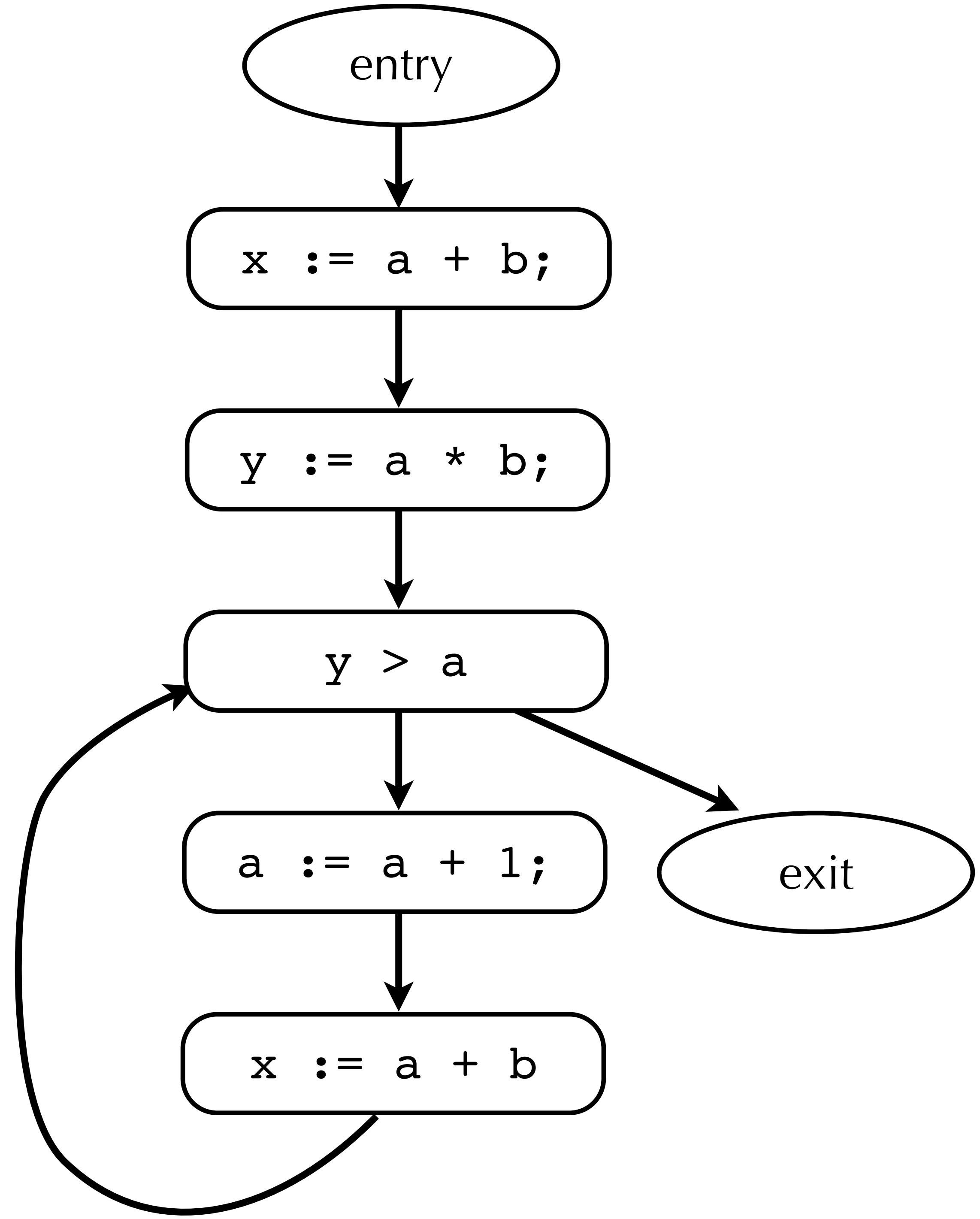
Data flow equations

- Available expressions is a forward must analysis
  - Propagate facts in same direction as control flow
  - Expression is available only if available on all paths
- Liveness is a **backwards may** analysis
  - To know if a variable is live, we need to look at the future uses of it. We propagate facts backwards, from Out to In
  - Variable is live if it is used on some path
- $\text{Out}(s) = \bigcup_{s' \in \text{succs}(s)} \text{In}(s')$
- $\text{In}(s) = \text{Gen}(s) \cup (\text{Out}(S) - \text{Kill}(s))$

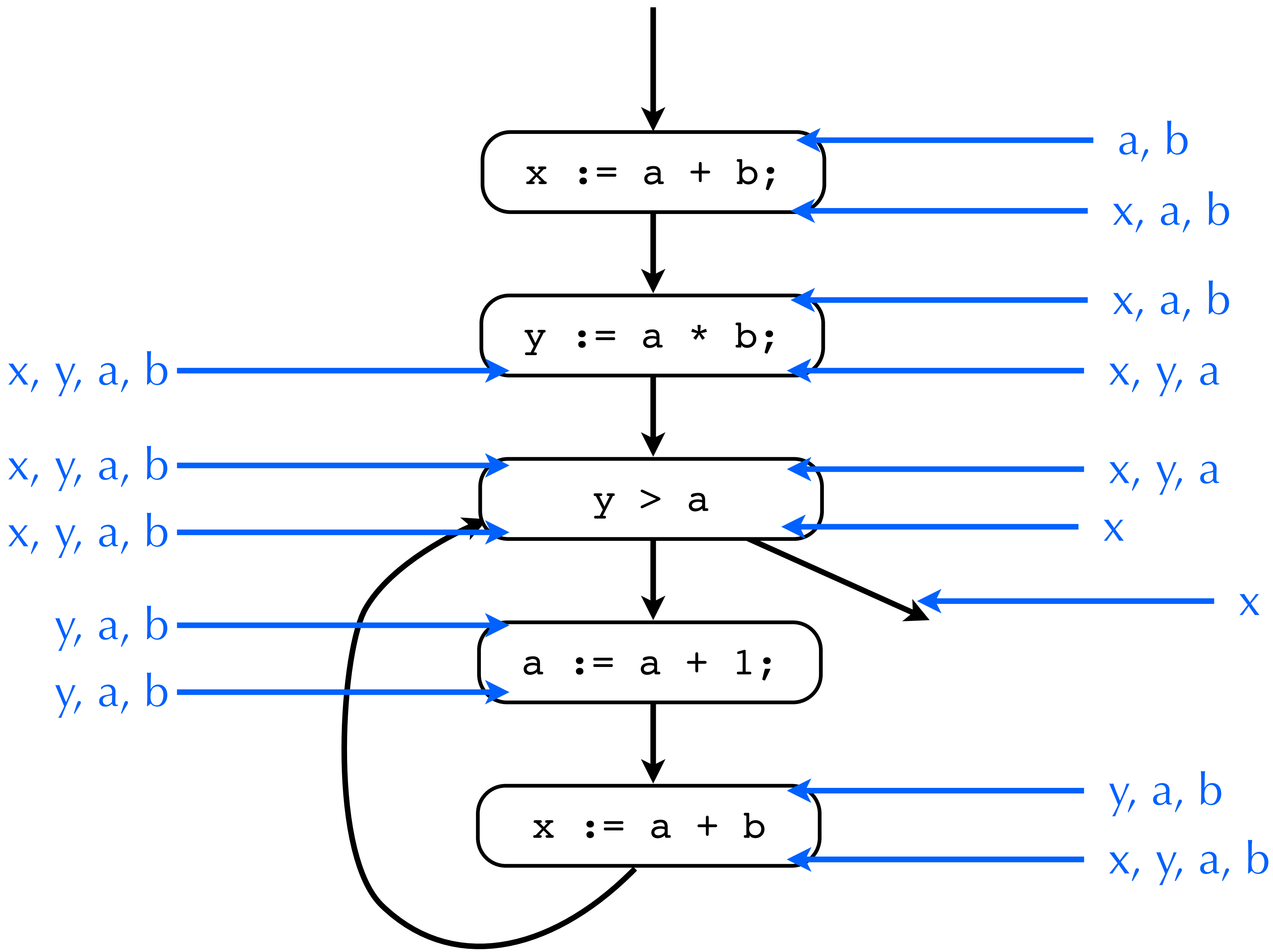
Gen and Kill

- What is the effect of each statement on the 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?
  - May or 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?
  - May or must?

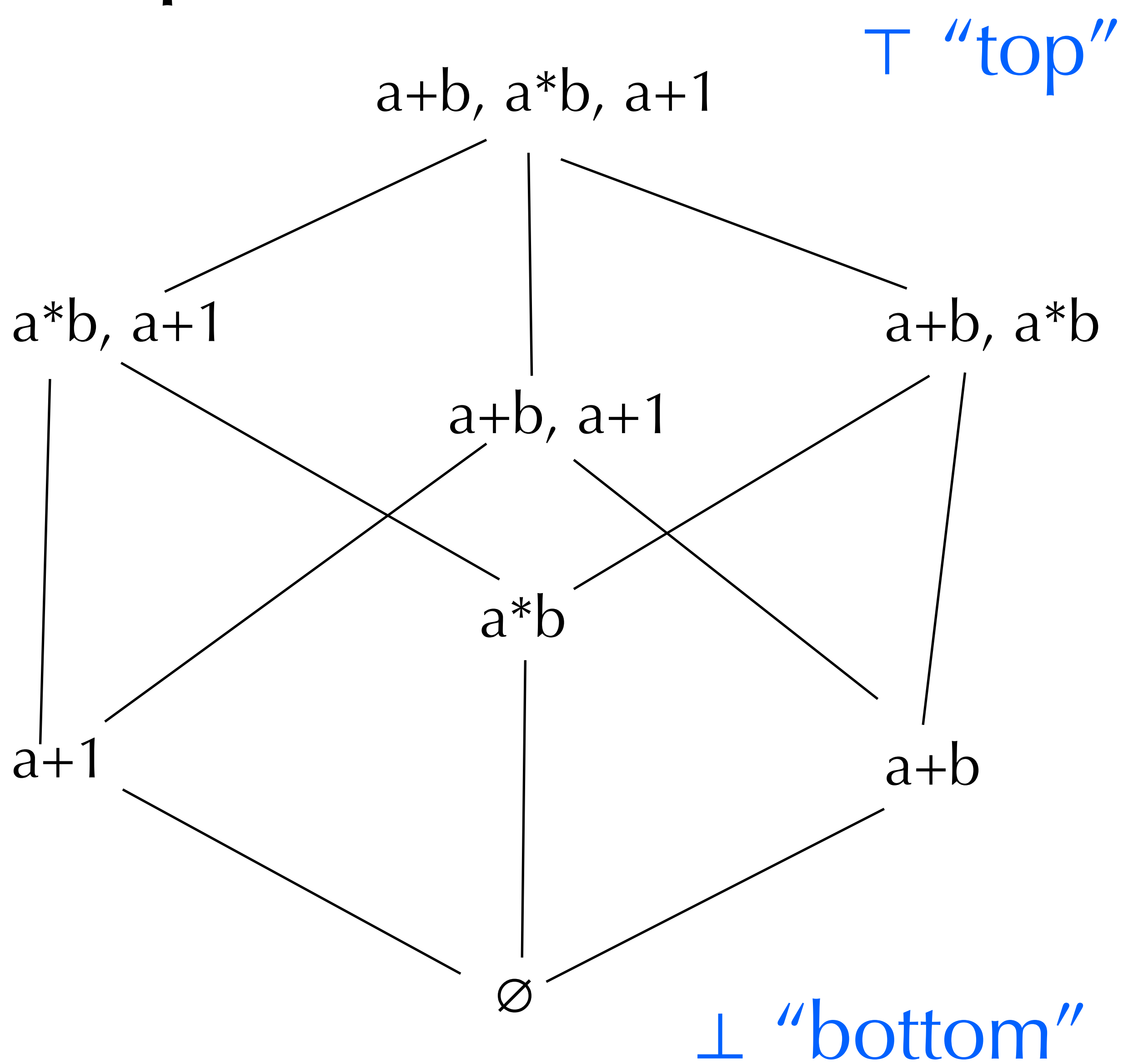
Space of data flow analyses

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

- Most dataflow analyses can be categorized in this way
  - A few don't fit, need bidrectional flow
- Lots of literature on data flow analyses

Data flow facts and lattices

- Typically, data flow facts form lattices
- E.g., available expressions



Partial orders and lattices

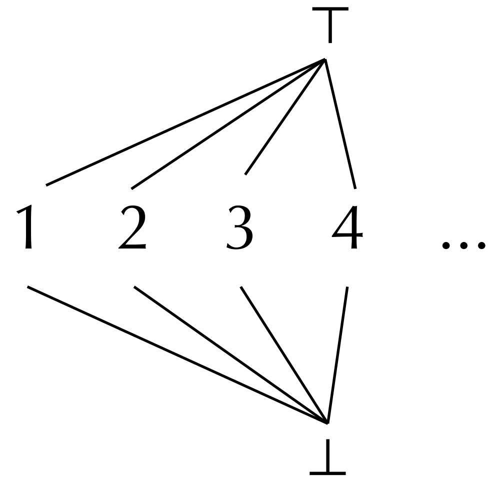
- A **partial order** is a pair $(P, \leq)$ such that
  - $\leq$ is a relation over P ($\leq \; \subseteq P \times P$)
  - $\leq$ is reflexive, anti-symmetric, and transitive
- A partial order is a **lattice** if every two elements of P have a unique least upper bound and greatest lower bound.
  - $\sqcap$ is the meet operator: $x \sqcap y$ is the greatest lower bound of x and y
    - $x \sqcap y \leq x$ and $x \sqcap y \leq y$
    - if $z \leq x$ and $z \leq y$ then $z \leq x \sqcap y$
  - $\sqcup$ is the join operator: $x \sqcup y$ is the least upper bound of x and y
    - $x \leq x \sqcup y$ and $y \leq x \sqcup y$
    - if $x \leq z$ and $y \leq z$ then $x \sqcup y \leq z$
- A join semi-lattice (meet semi-lattice) has only the join (meet) operator defined

Complete lattices

- A partially ordered set is a **complete lattice** if meet and join are defined for all subsets (i.e., not just for all pairs)
- A complete lattice always has a bottom element and a top element
- A finite lattice always has a bottom element and a top element

Useful lattices

- $(2^S, \subseteq)$ forms a lattice for any set S
  - 2^S is powerset of S, the set of all subsets of S.
- If $(S, \leq)$ is a lattice, so is $(S, \geq)$
  - i.e., can "flip" the lattice
- Lattice for constant propagation



Forward must data flow algorithm

```
Out(s) = ⊤ for all statements s
W := { all statements }                    (worklist)
repeat {
    Take s from W
    In(s) := ∩_{s′ ∈ 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
  - if $x \leq y$ then $f(x) \leq f(y)$
- Functions for computing In(s) and Out(s) are monotonic
  - $In(s) := \bigcap_{s' \in pred(s)} Out(s')$
  - $temp := Gen(s) \cup (In(s) - Kill(s))$ A function f_s of In(s)
- Putting them together: $temp := f_s(\bigcap_{s' \in pred(s)} Out(s'))$

Termination

- We know the algorithm terminates
- In each iteration, either W gets smaller, or Out(s) decreases for some s
  - Since function is monotonic
- Lattice has only finite height, so for each s, Out(s) can decrease only finitely often

```
Out(s) = ⊤ for all statements s
W := { all statements }
repeat {
    Take s from W

    In(s) := ∩s′ ∊ pred(s) Out(s′)
    temp := Gen(s) ∪ (In(s) - Kill(s))
    if (temp != Out(s)) {
        Out(s) := temp
        W := W ∪ succ(s)
    }
} until W = ∅
```

Termination

- A **descending chain** in a lattice is a sequence $x_0 < x_1 < \ldots$
- The **height of a lattice** is the length of the longest descending chain in the lattice
- Then, dataflow must terminate in O(nk) time
  - n = # of statements in program
  - k = height of lattice
  - assumes meet operation and transfer function takes O(1) time

Fixpoints

- Dataflow tradition: Start with Top, use meet
  - To do this, we need a meet semilattice with top
    - complete meet semilattice = meets defined for any set
    - finite height ensures termination
  - Computes greatest fixpoint

- Denotational semantics tradition: Start with Bottom, use join
  - Computes least fixpoint

Forward must data flow algorithm

```
Out(s) = ⊤ for all statements s
W := { all statements }                    (worklist)
repeat {
    Take s from W
    In(s) := ∩_{s′ ∈ pred(s)} Out(s′)
    temp := Gen(s) ∪ (In(s) - Kill(s))
    if (temp != Out(s)) {
        Out(s) := temp
        W := W ∪ succ(s)
    }
} until W = ∅
```

Forward data flow again

```
Out(s) = ⊤ for all statements 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 = ∅
```

Transfer function for statement s

Which lattice to use?

- Available expressions
  - P = sets of expressions
  - Meet operation $\sqcap$ is set intersection $\cap$
  - $\top$ is set of all expressions
- Reaching definitions
  - P = sets of definitions (assignment statements)
  - Meet operation $\sqcap$ is set union $\cup$
  - $\top$ is empty set
- Monotonic **transfer function** f_s is defined based on gen and kill sets.

Distributive data flow problems

- If f is monotonic, then we have

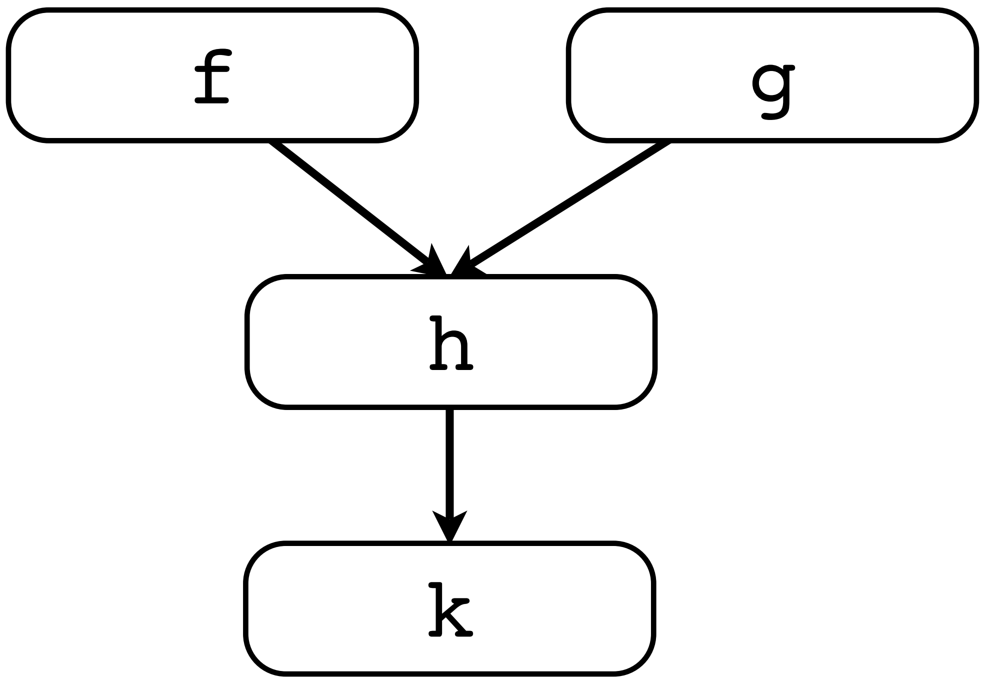
$$f(x \sqcap y) \leq f(x) \sqcap f(y)$$

- If f is **distributive** then we have

$$f(x \sqcap y) = f(x) \sqcap f(y)$$

Benefit of distributivity

- Joins lose no information



- $k(h(f(\top) \sqcap g(\top)))$
  $= k(h(f(\top)) \sqcap h(g(\top)))$
  $= k(h(f(\top))) \sqcap k(h(g(\top)))$

Accuracy of data flow analysis

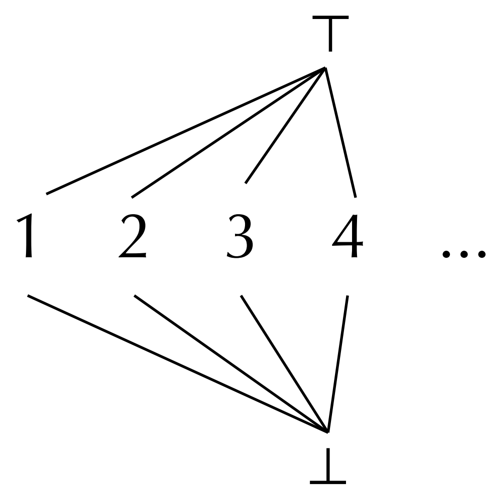
- Ideally we would like to compute the **meet over all paths** (MOP) solution:
  - Let f_s be the transfer function for statement s
  - If p is a path $s_1,\ldots,s_n$, let $f_p = f_{sn};\ldots f_{s1}$
  - Let paths(s) be the set of paths from the entry to s
  - $MOP(s) = \sqcap_{p \in paths(s)} f_p(\top)$
- If the transfer functions are distributive, then solving using the data flow equations in the standard way produces the MOP solution

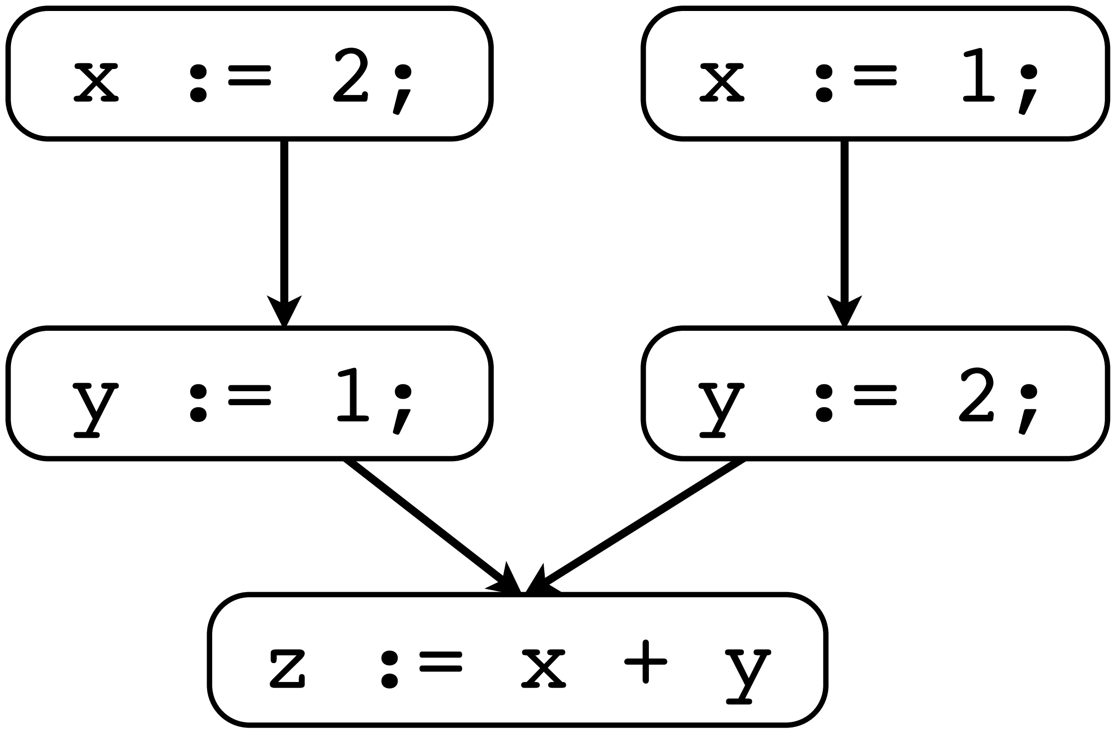
What problems are distributive?

- Analyses of *how* the program computes
  - E.g.,
    - Live variables
    - Available expressions
    - Reaching definitions
    - Very busy expressions
- All Gen/Kill problems are distributive

Non-distributive example

- Constant propagation





- In general, analysis of *what* the program computes is not distributive
- Thm: MOP for In(s) will always be $\sqsubseteq$ iterative dataflow solution

Practical implementation

- Data flow facts are assertions that are true or false at a program point
- Can represent set of facts as bit vector
  - Fact i represented by bit i
  - Intersection=bitwise and, union=bitwise or, etc
- "Only" a constant factor speedup
  - But very useful in practice

Basic blocks

- A **basic block** is a sequence of statements such that
  - No branches to any statement except the first
  - No statement in the block branches except the last

- In practical data flow implementations
  - Compute Gen/Kill for each basic block
    - Compose transfer functions
  - Store only In/Out for each basic block
  - Typical basic block is about 5 statements

Order is important

- Assume forward data flow problem
  - Let G=(V,E) be the CFG
  - Let k be the height of the lattice
- If G acyclic, visit in topological order
  - Visit head before tail of edge
- Running time O(|E|)
  - No matter what size the lattice

Order is important

- If G has cycles, visit in reverse postorder
  - Order from depth-first search
- Let Q = max # back edges on cycle-free path
  - Nesting depth
  - Back edge is from node to ancestor on DFS tree
- Then if $\forall x.\ f(x) \leq x$ (sufficient, but not necessary)
  - Running time is $O((Q + 1)|E|)$

Flow sensitivity

- Data flow analysis is **flow sensitive**
  - The order of statements is taken into account
    - I.e., we keep track of facts per program point
- Alternative: **Flow-insensitive** analysis
  - Analysis the same regardless of statement order
  - Standard example: types describe facts that are true at all program points
    - /*x:int*/   x:=…   /*x:int*/

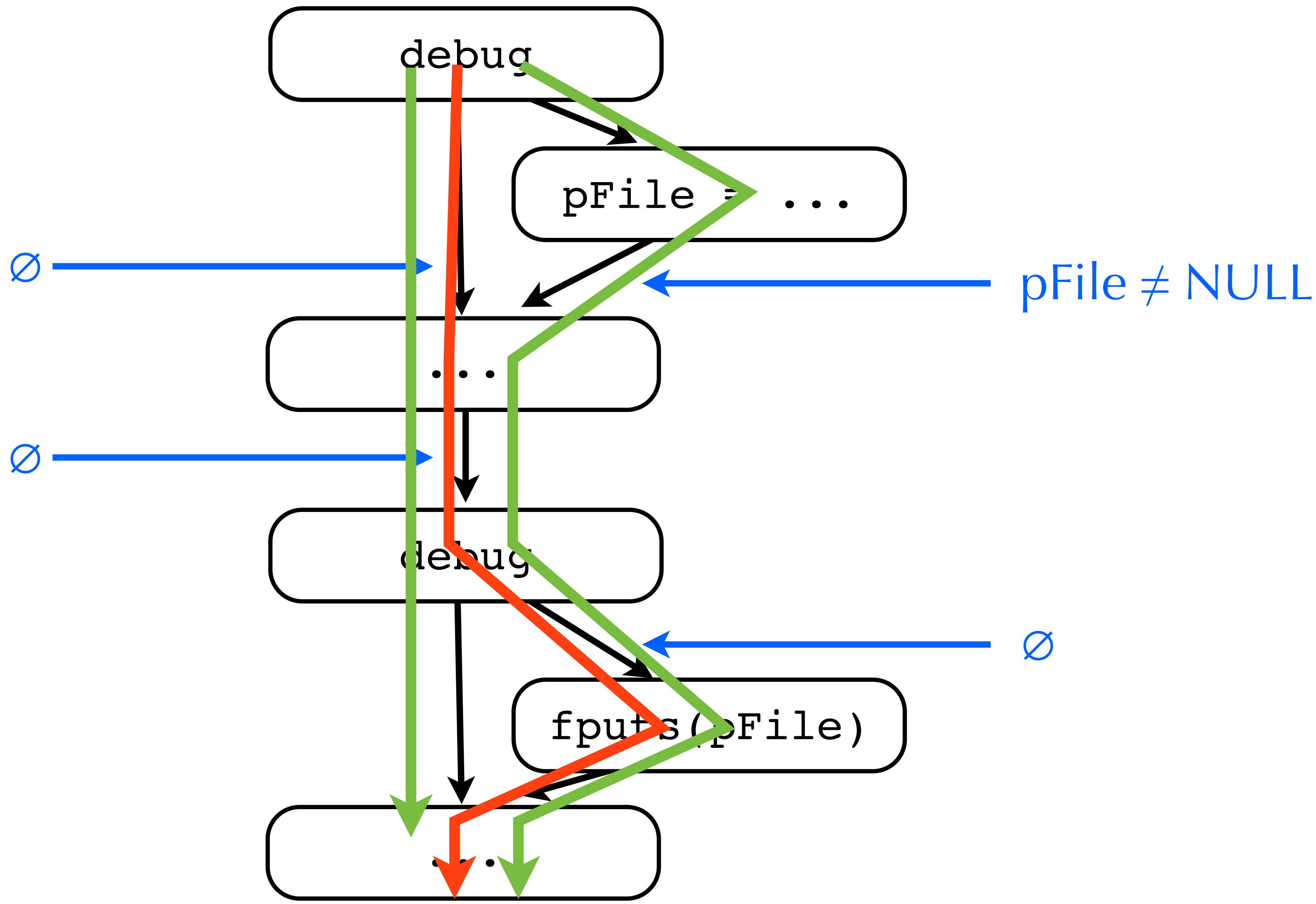
A problem...

- Consider following program

```
FILE *pFile = NULL;
if (debug) {
    pFile = fopen("debuglog.txt", "a")
}
...
if (debug) {
    fputs("foo", pFile);
}
```

- Can pFile be NULL when used for fputs?
- What dataflow analysis could we use to determine if it is?

Path sensitivity



Path sensitivity

- A **path-sensitive** analysis tracks data flow facts depending on the path taken
  - Path often represented by which branches of conditionals taken
- Can reason more accurately about **correlated conditionals** (or **dependent conditionals**) such as in previous example

- How can we make a path sensitive analysi
  - Could do a dataflow analysis where we track facts for each possible path
  - But exponentially many paths make it difficult to scale
- Some research on scalable path sensitive analyses. We will discuss one next week

Terminology review

- Must vs. May
  - (Not always followed in literature)
- Forwards vs. Backwards
- Flow-sensitive vs. Flow-insensitive
- Path-sensitive vs Path-insensitive
- Distributive vs. Non-distributive

Dataflow analysis and the heap

- Data Flow is good at analyzing local variables
  - But what about values stored in the heap?
  - Not modeled in traditional data flow
- In practice: *x := e
  - Assume all data flow facts killed (!)
  - Or, assume write through x may affect any variable whose address has been taken
- In general, hard to analyze pointers