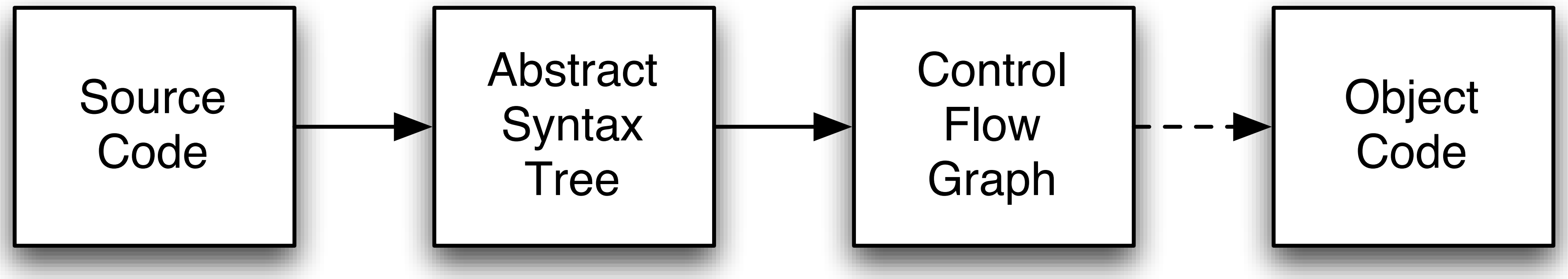
CMSC 631 — Program Analysis and Understanding

Data Flow Analysis

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

Compiler Structure



- Source code parsed to produce AST
- AST transformed to CFG
- Data flow analysis operates on control flow graph (and other intermediate representations)
  - Symbolic execution often uses the AST

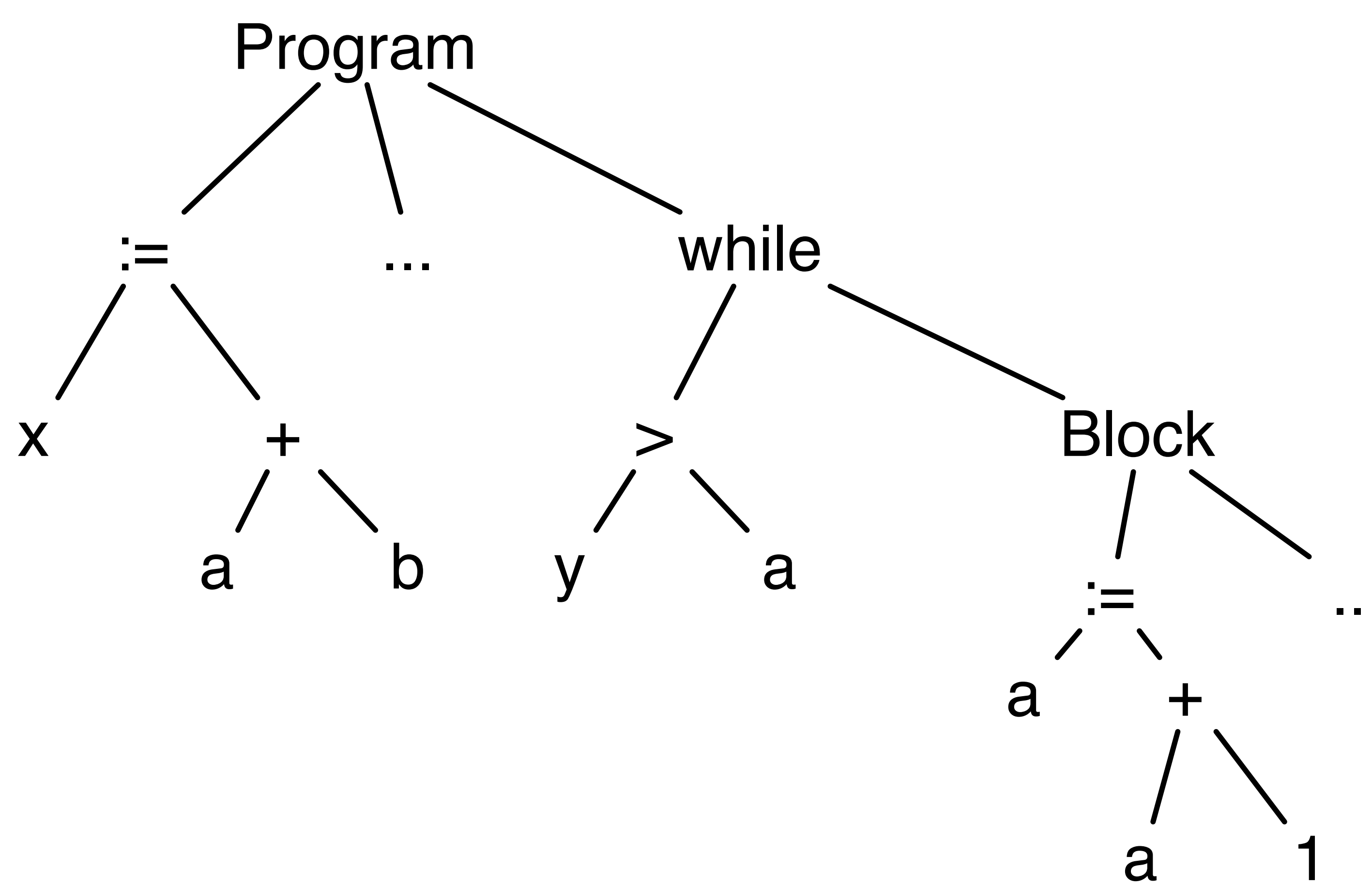
Abstract Syntax Tree (AST)

- Programs are written in text
  - I.e., sequences of characters
  - Awkward to work with

- First step: Convert to structured representation
  - Use lexer (like flex) to recognize *tokens*
    - Sequences of characters that make words in the language
  - Use parser (like bison) to group words structurally
    - And, often, to produce AST

Abstract Syntax Tree Example

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



ASTs

- ASTs are *abstract*
  - They don't contain all information in the program
    - E.g., spacing, comments, brackets, parentheses
  - Any ambiguity has been resolved
    - E.g., a + b + c produces the same AST as (a + b) + c

- For more info, see CMSC 430
  - In this class, we will generally begin at the AST level

Disadvantages of ASTs

- AST has many similar forms
  - E.g., for, while, repeat...until
  - E.g., if, ?:, switch
- Expressions in AST may be complex, nested
  - (42 * y) + (z > 5 ? 12 * z : z + 20)
- Want simpler representation for analysis
  - ...at least, for dataflow analysis

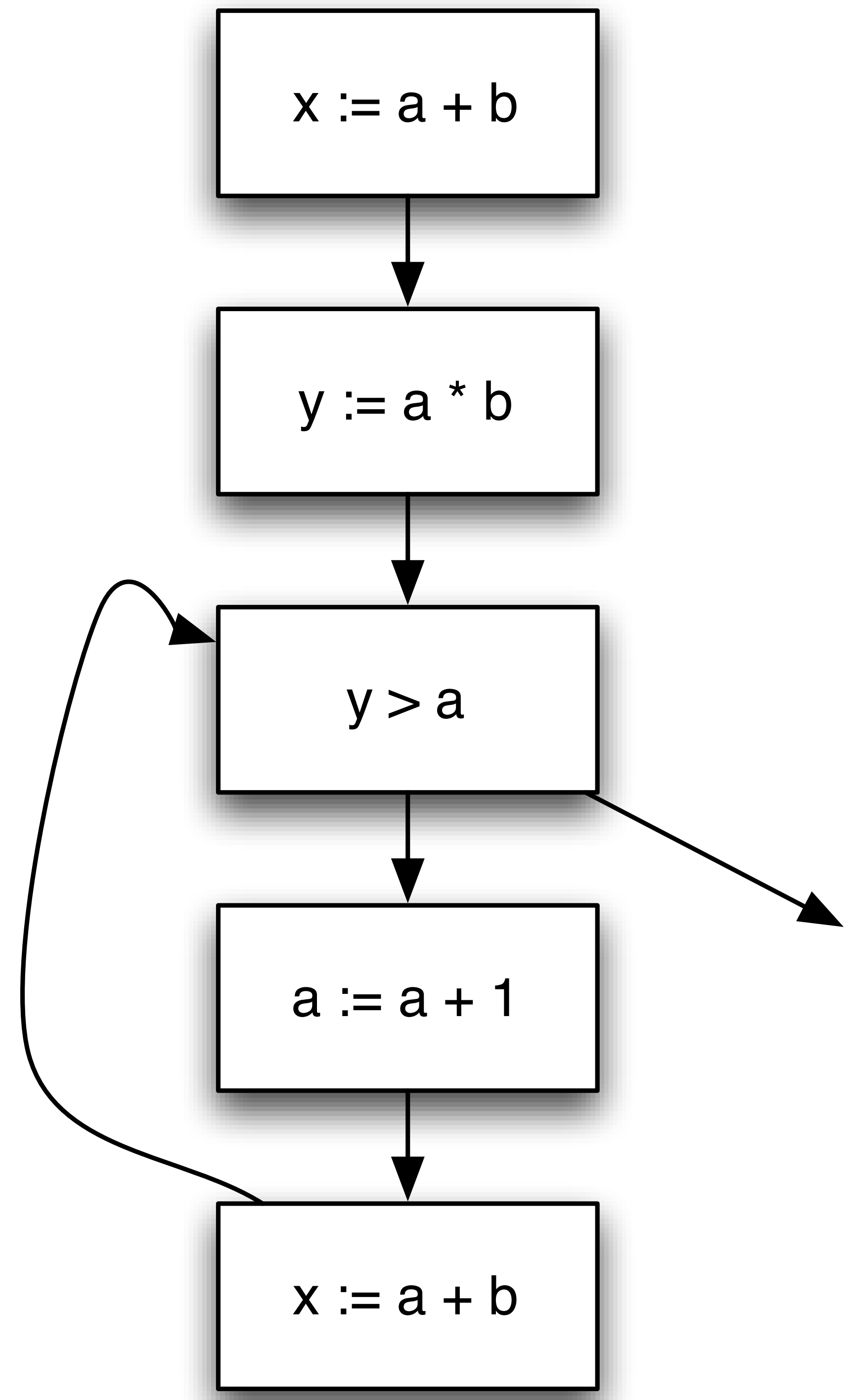
Control-Flow Graph (CFG)

- A directed graph where
  - Each node represents a statement
  - Edges represent control flow

- Statements may be
  - Assignments x := y op z or x := op z
  - Copy statements x := y
  - Branches goto L or if x relop y goto L
  - etc.

Control-Flow Graph Example

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

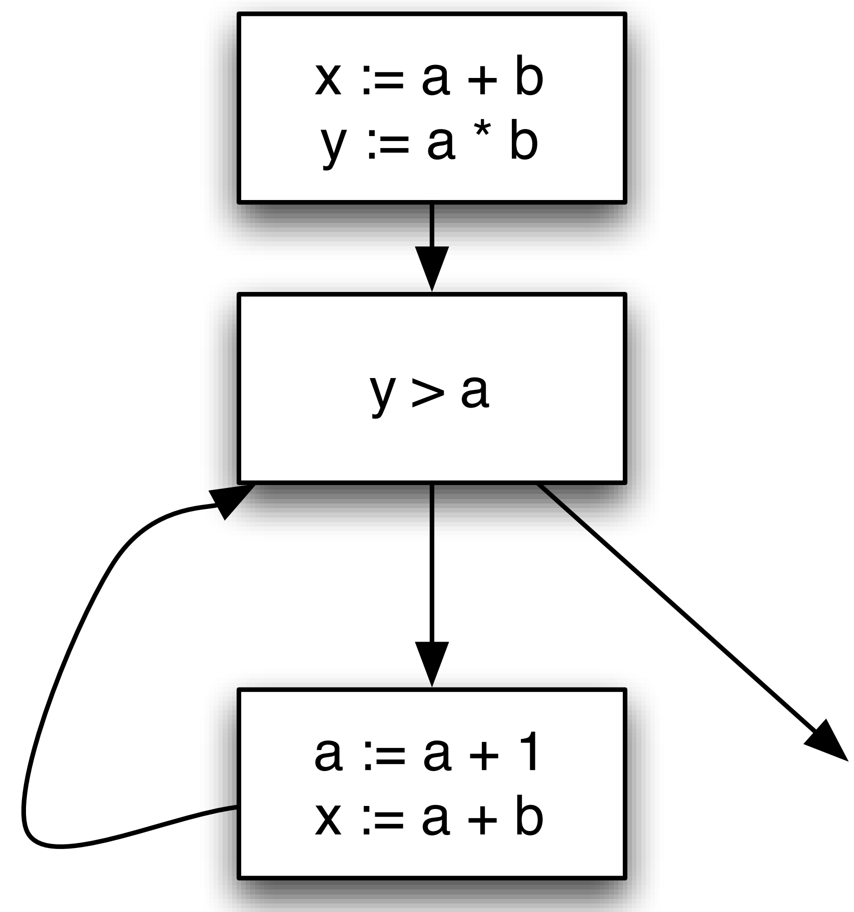


Variations on CFGs

- We usually don’t include declarations (e.g., int x;)
  - 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 w/Basic Blocks

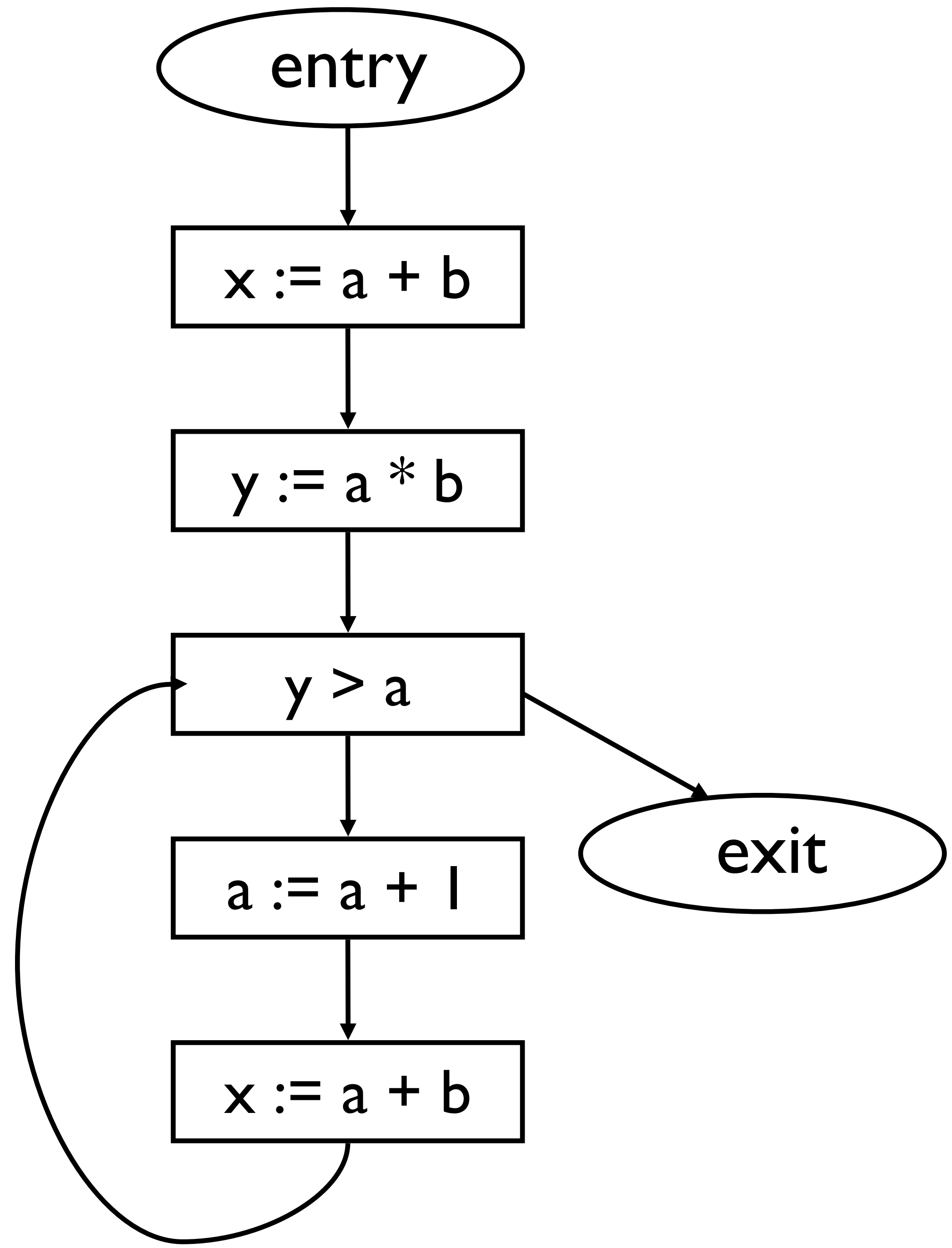
```
x := a + b;
y := a * b;
while (y > a + b) {
   a := a + 1;
   x := a + b
}
```



- Can lead to more efficient implementations
- But more complicated to explain, so...
  - We'll use single-statement blocks in lecture today

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 without 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
- So for AST,
  - Easier to report errors and other messages
  - Easier to explain to programmer
  - Easier to unparse to produce readable code

Analysis: Available Expressions

Analysis: 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 every path to p

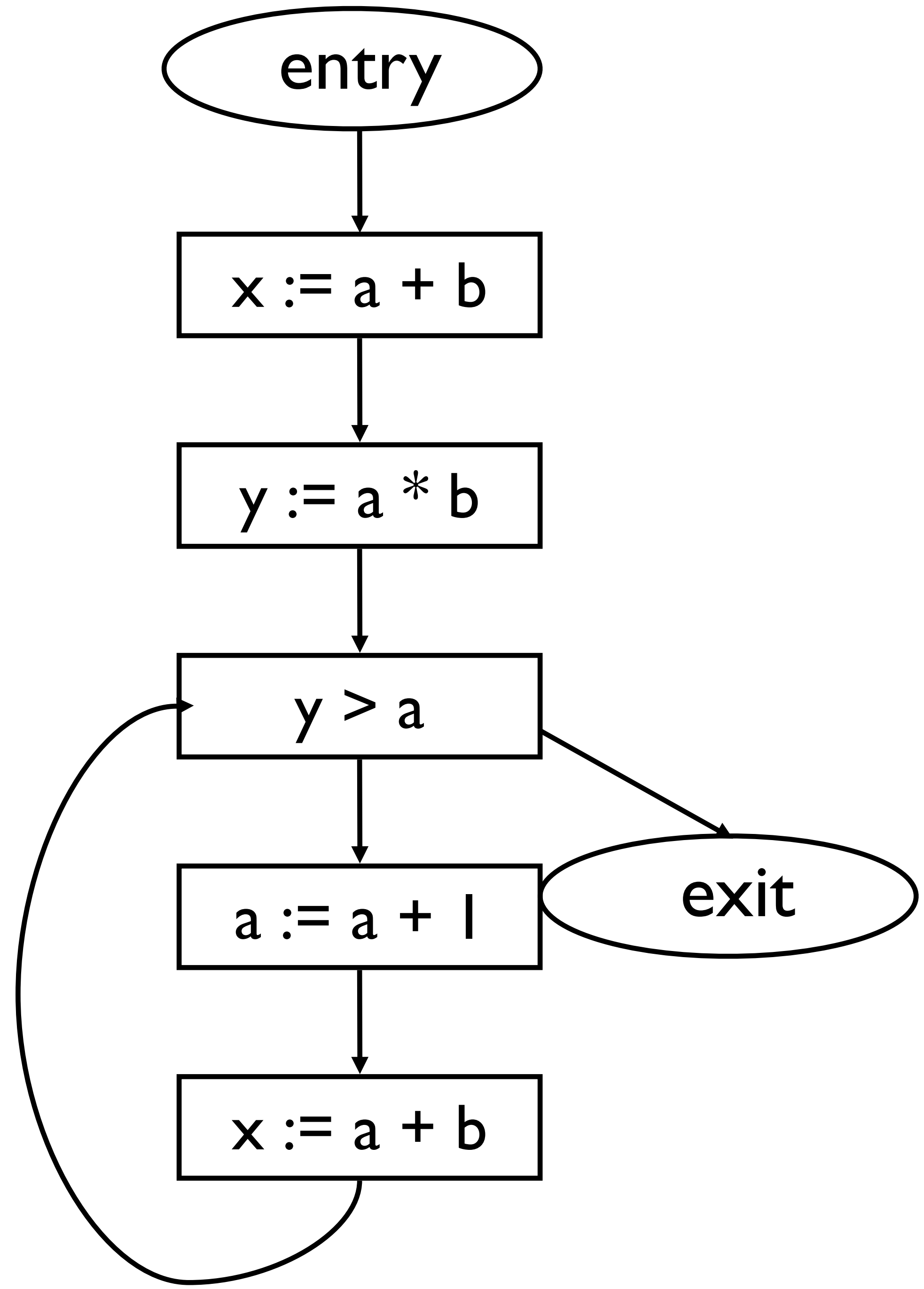
Analysis: Available Expressions

- 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 every path to p

- Optimization
  - If an expression is available, need not be recomputed
    - (At least, if it's still in a register somewhere)

Data Flow Facts

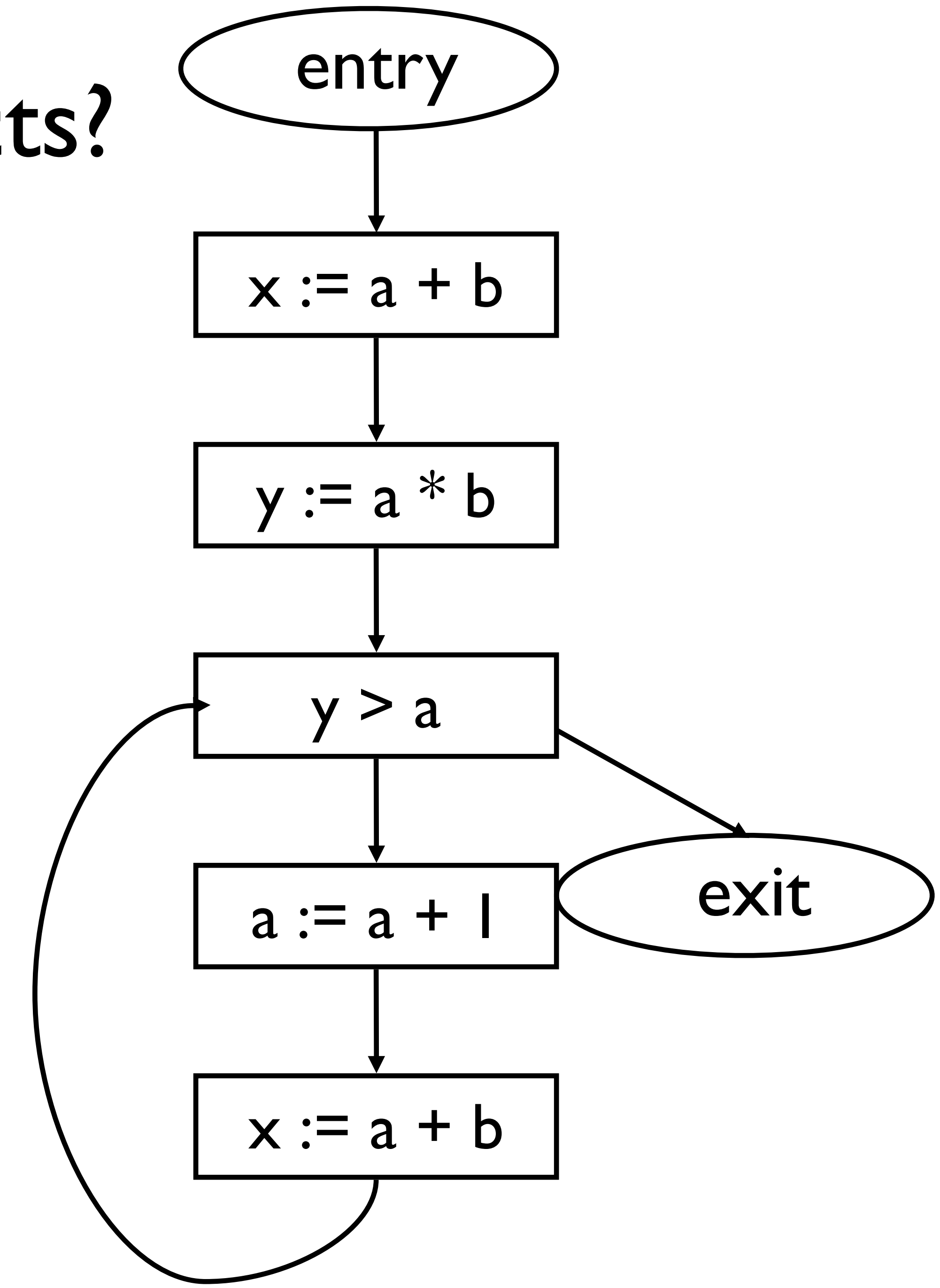
- Is expression e available?
- Facts:
  - a + b is available
  - a * b is available
  - a + 1 is available



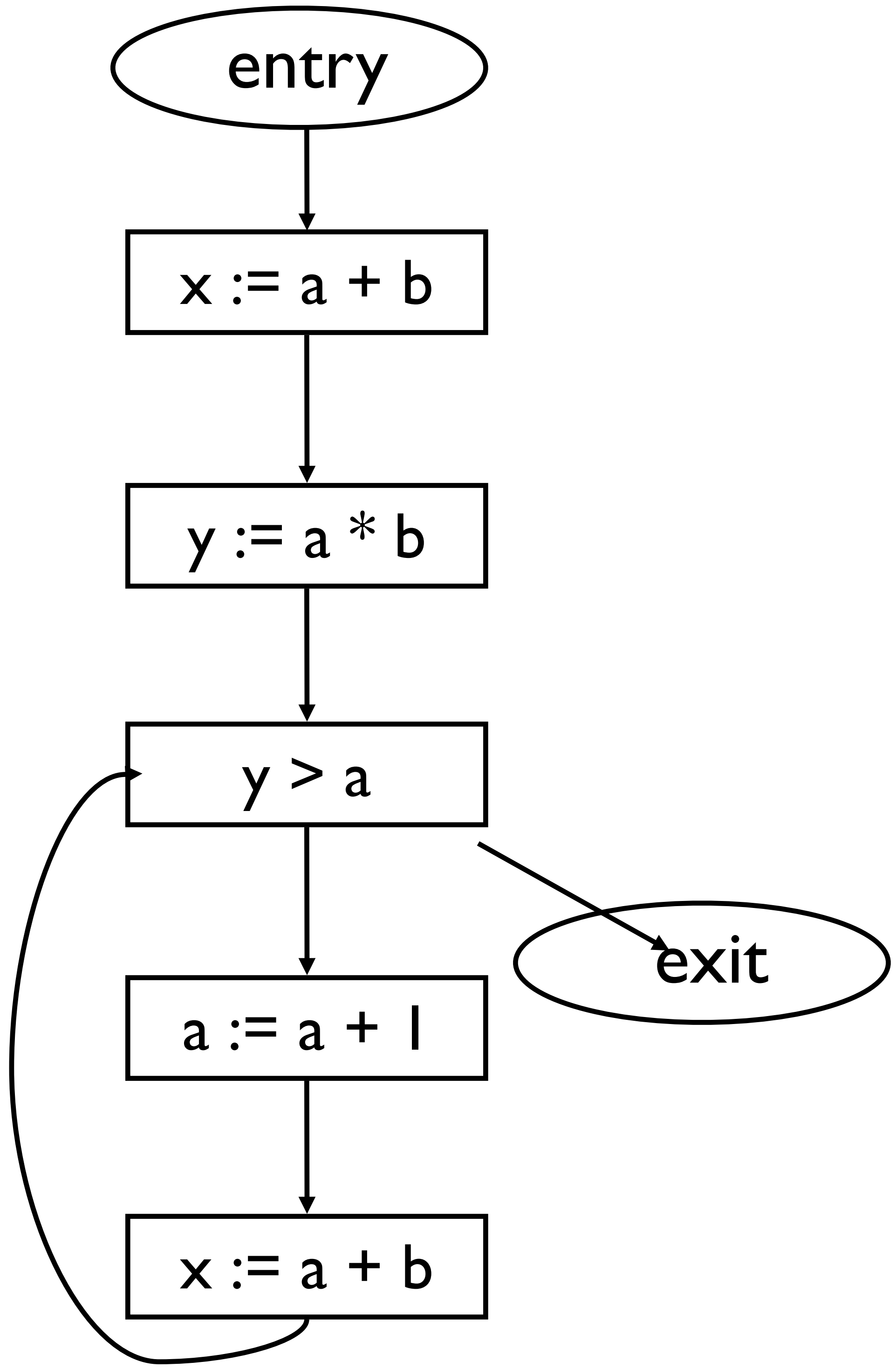
Gen and Kill

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

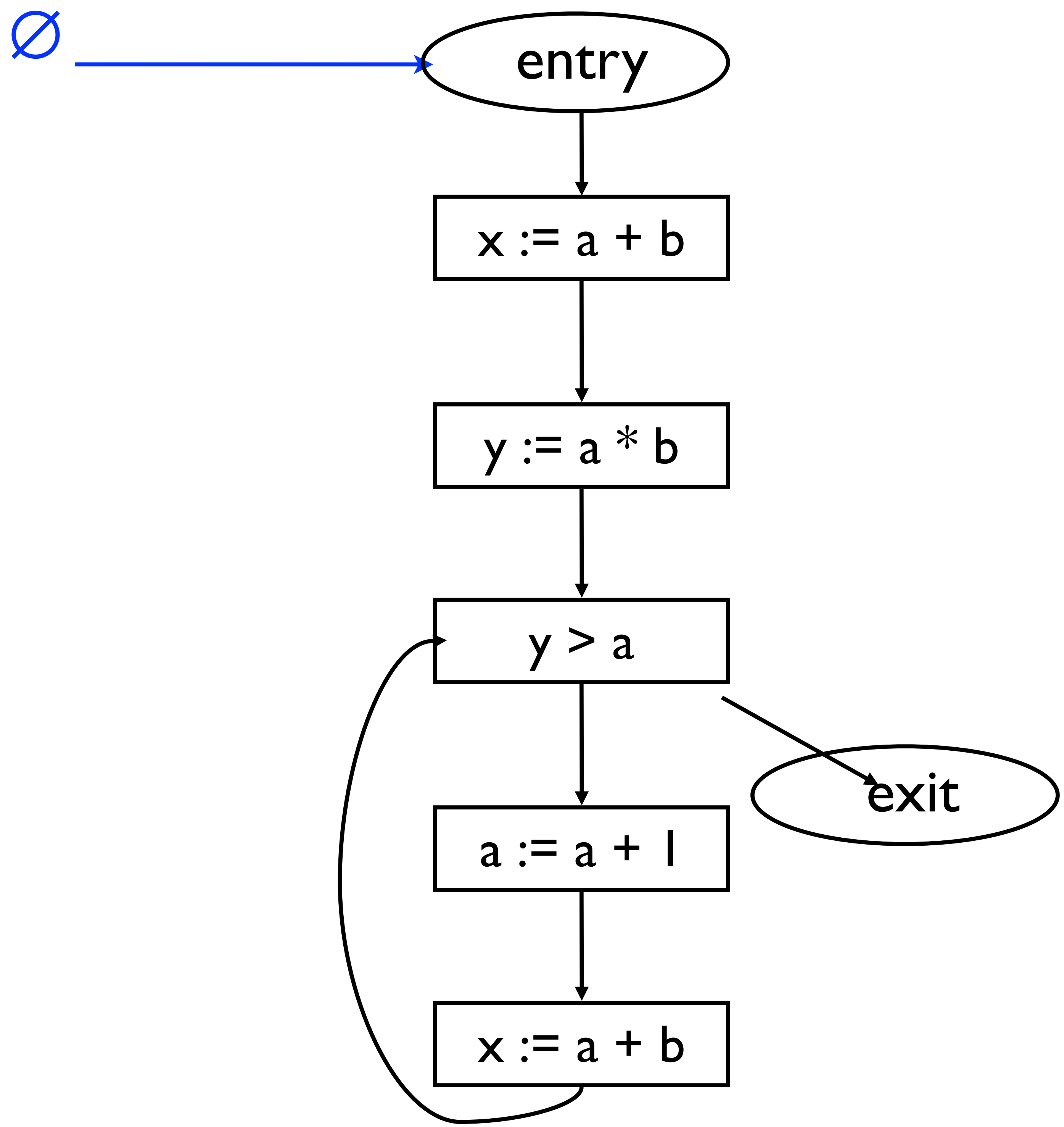
Stmt	Gen	Kill
x := a + b	a + b	
y := a * b	a * b	
a := a + 1		a + 1, a + b, a * b



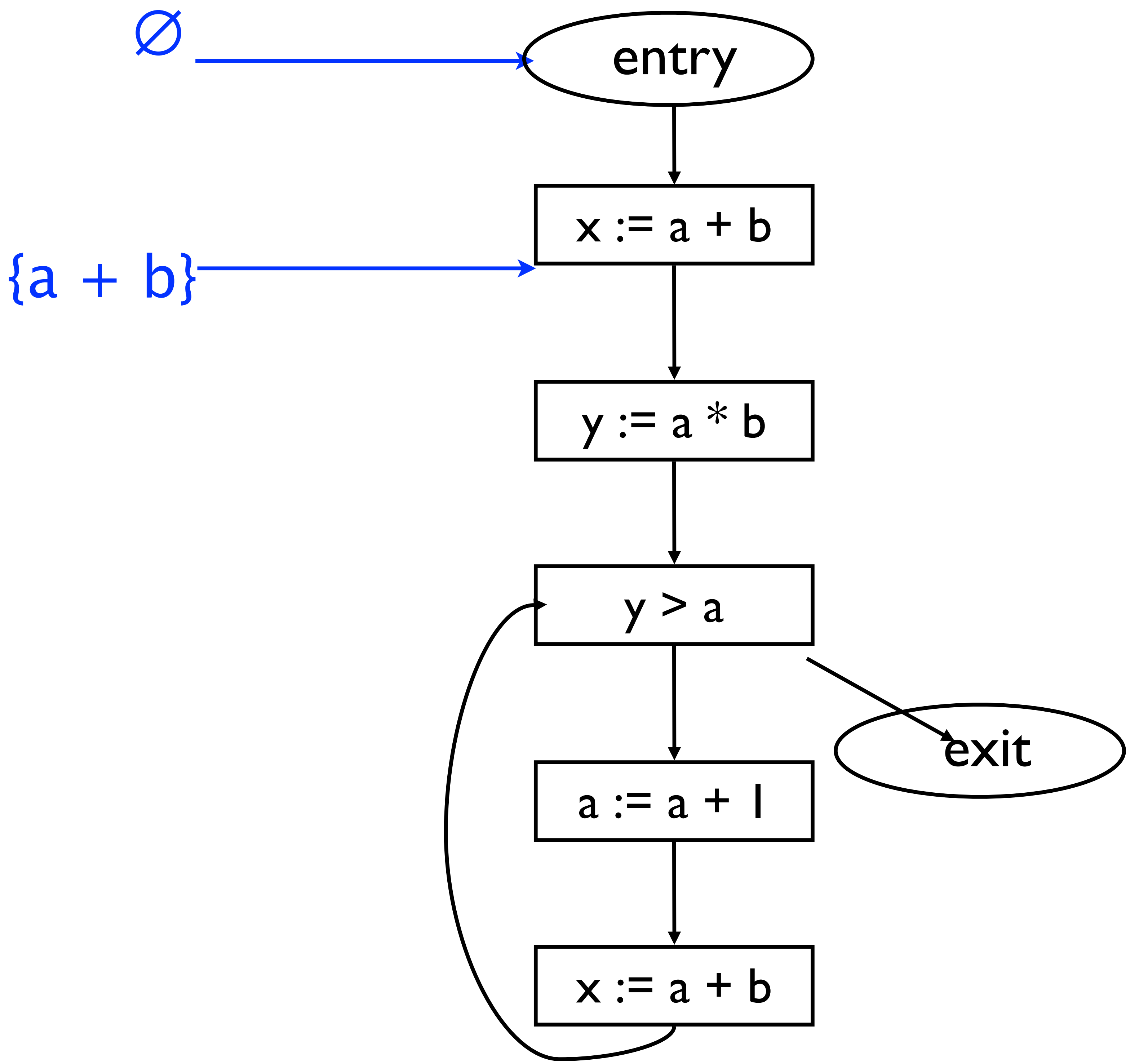
Computing Available Expressions



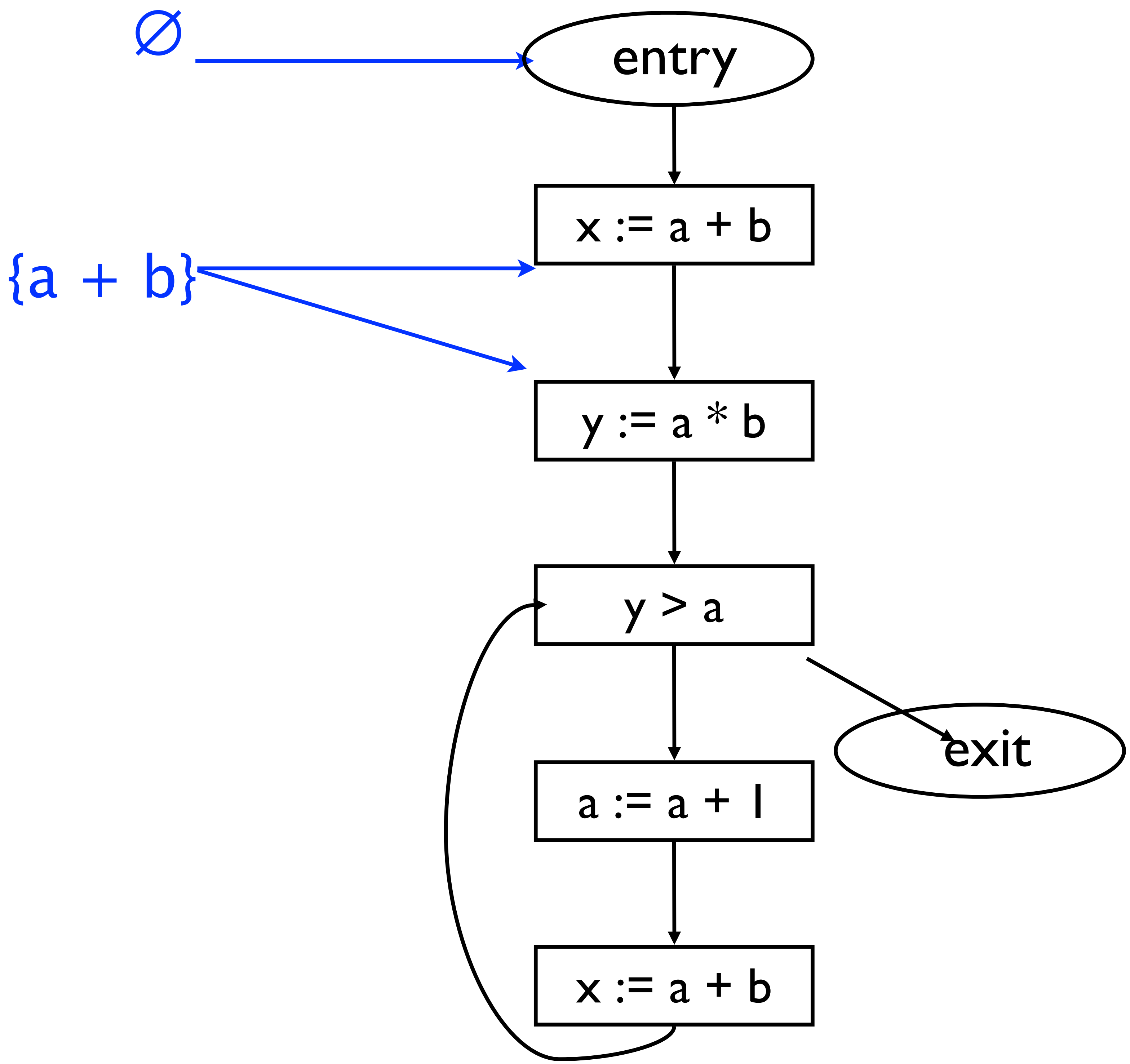
Computing Available Expressions



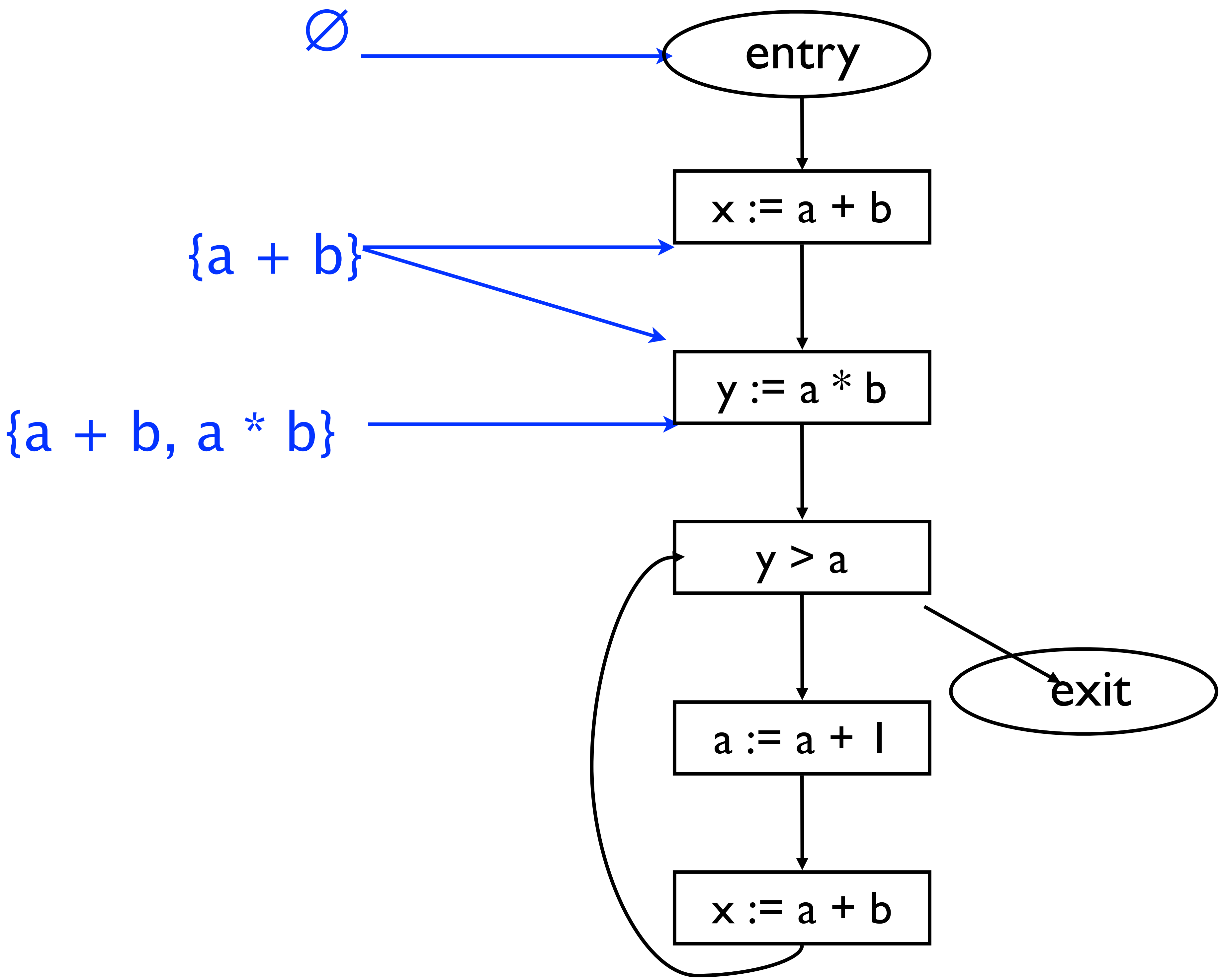
Computing Available Expressions



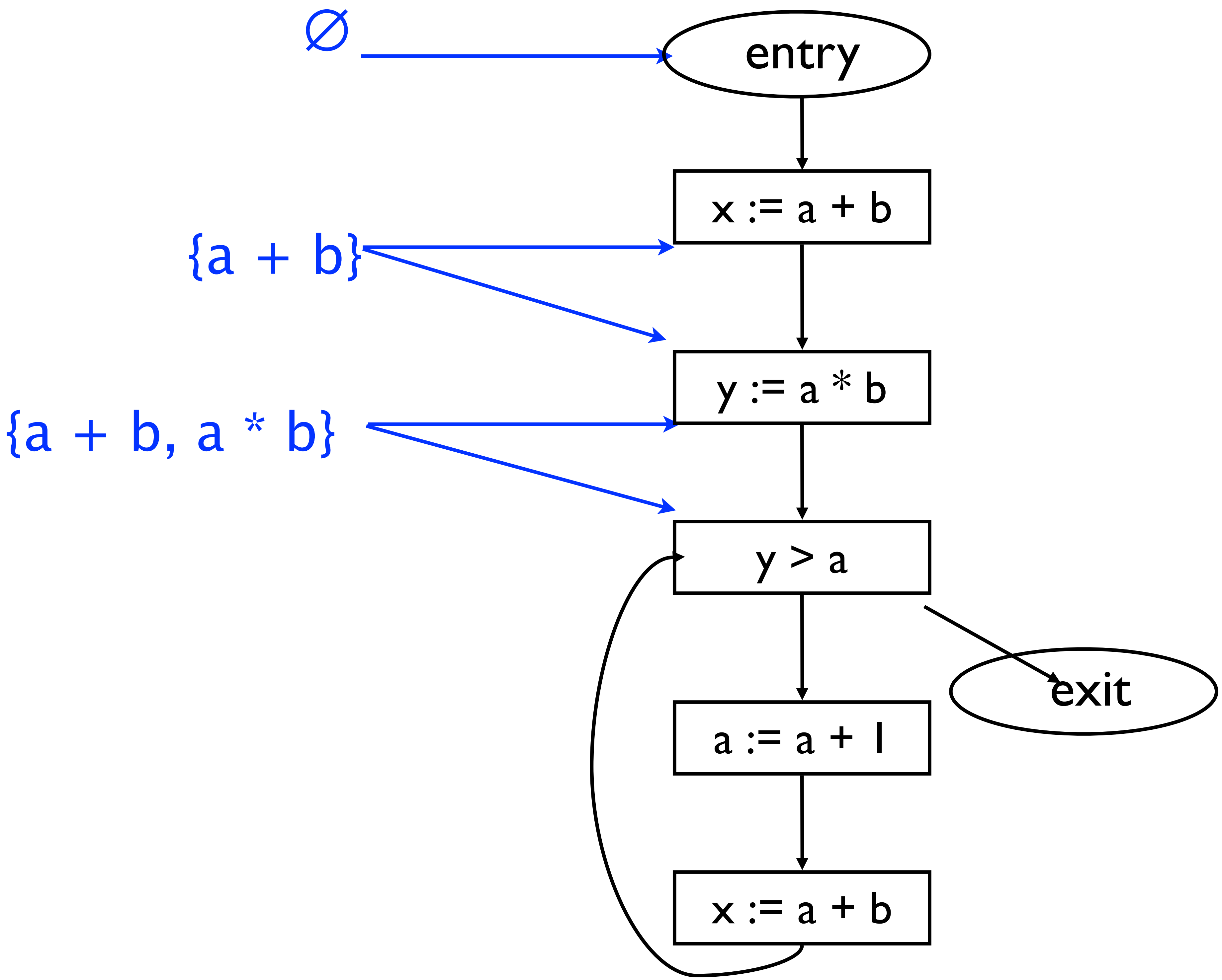
Computing Available Expressions



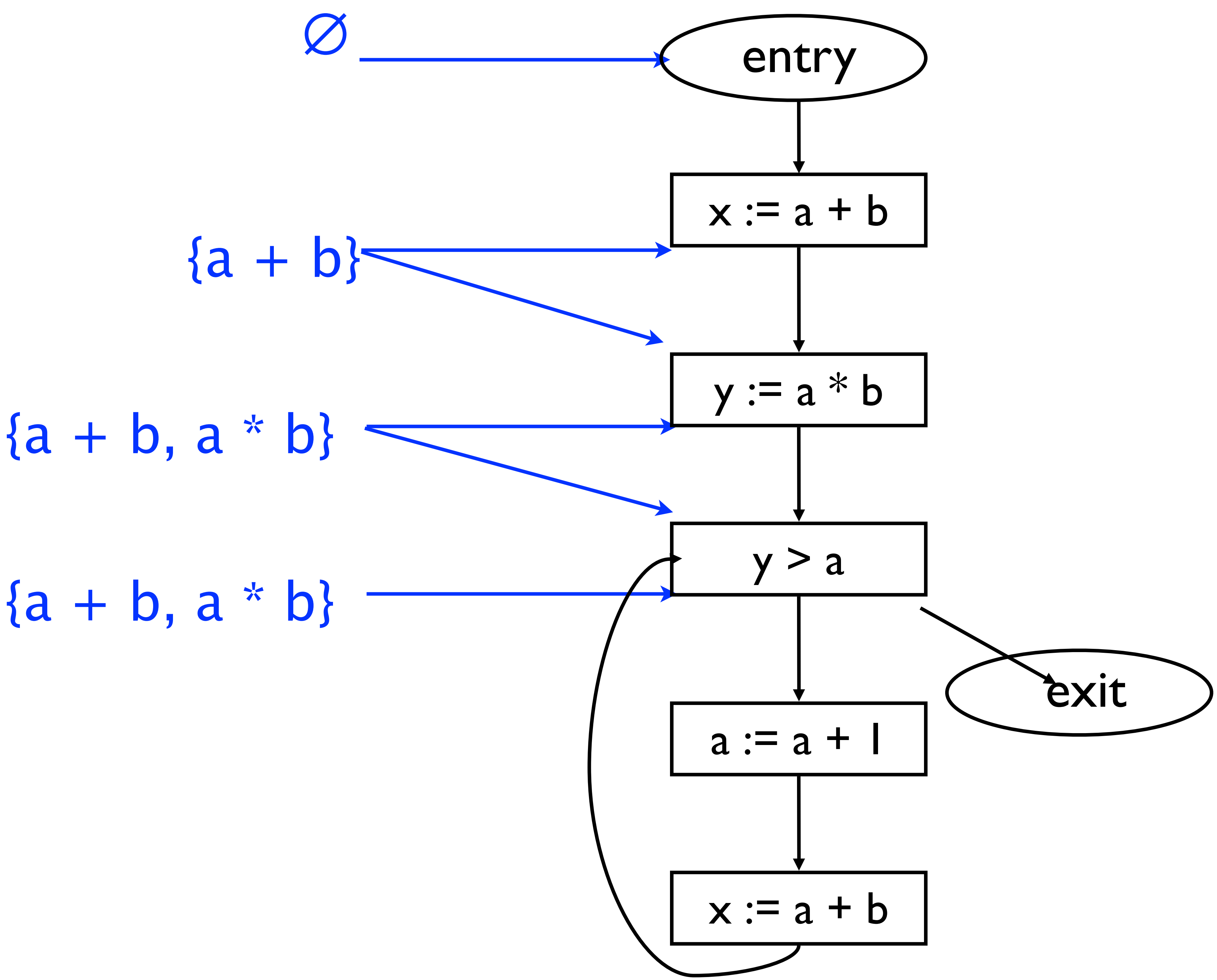
Computing Available Expressions



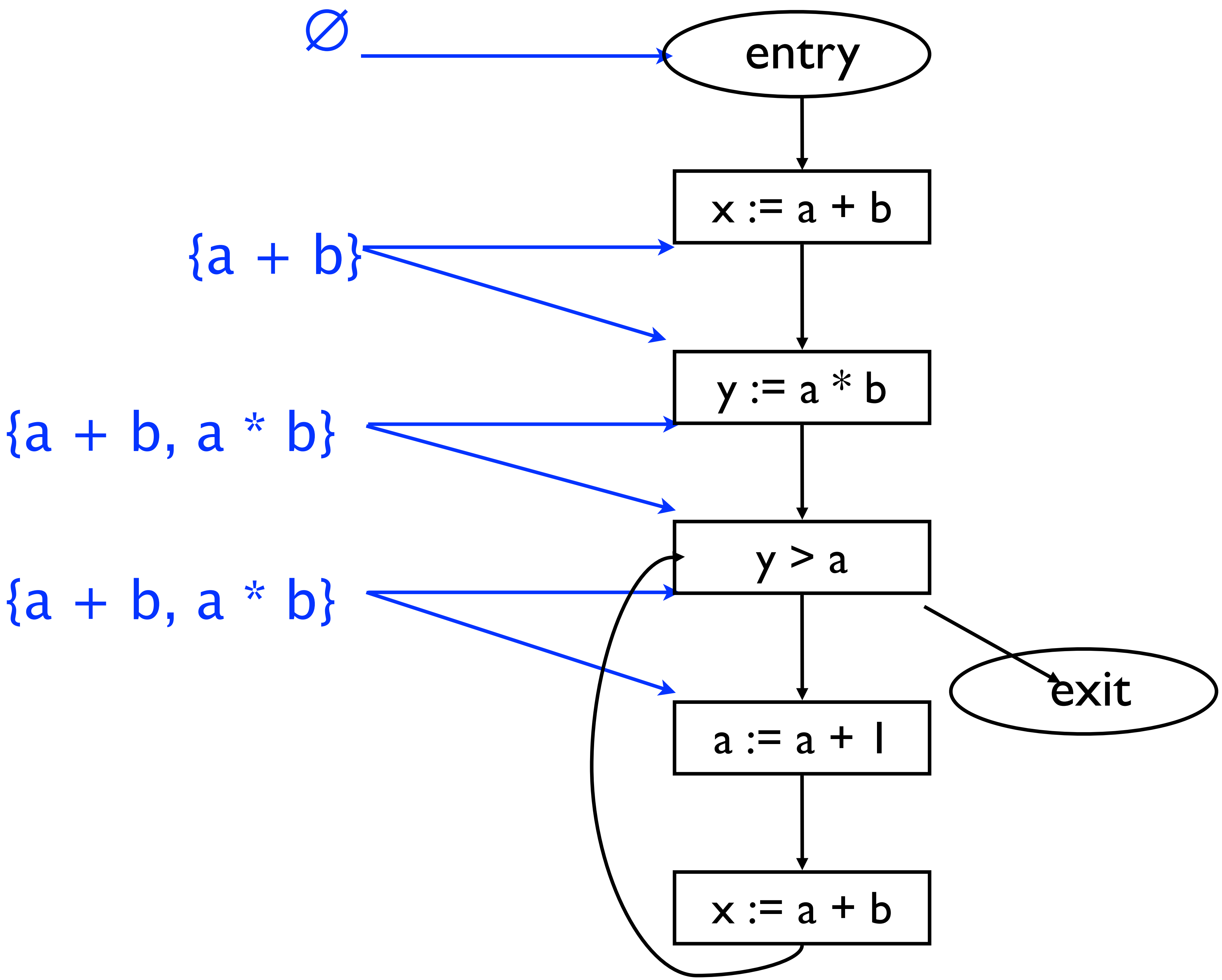
Computing Available Expressions



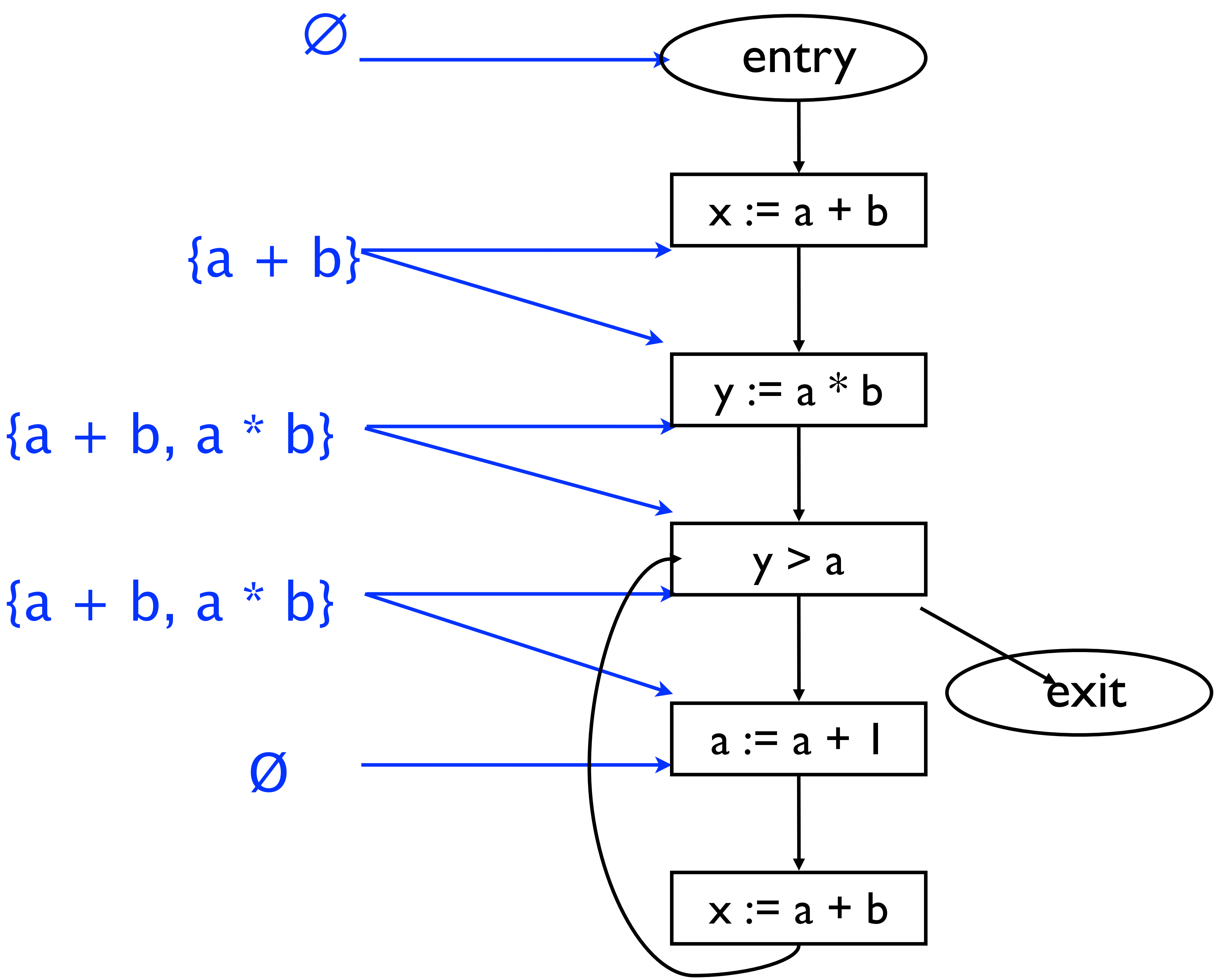
Computing Available Expressions



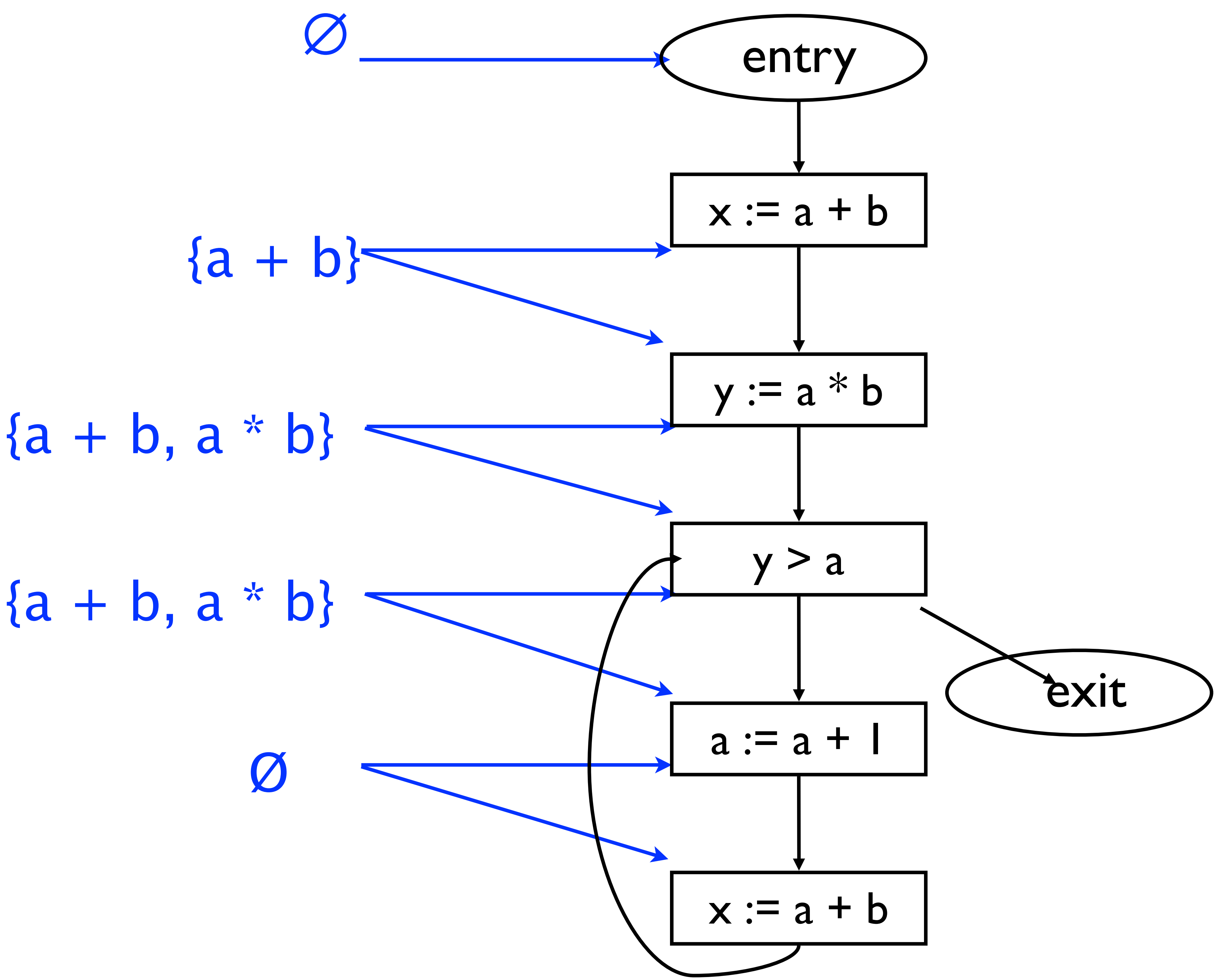
Computing Available Expressions



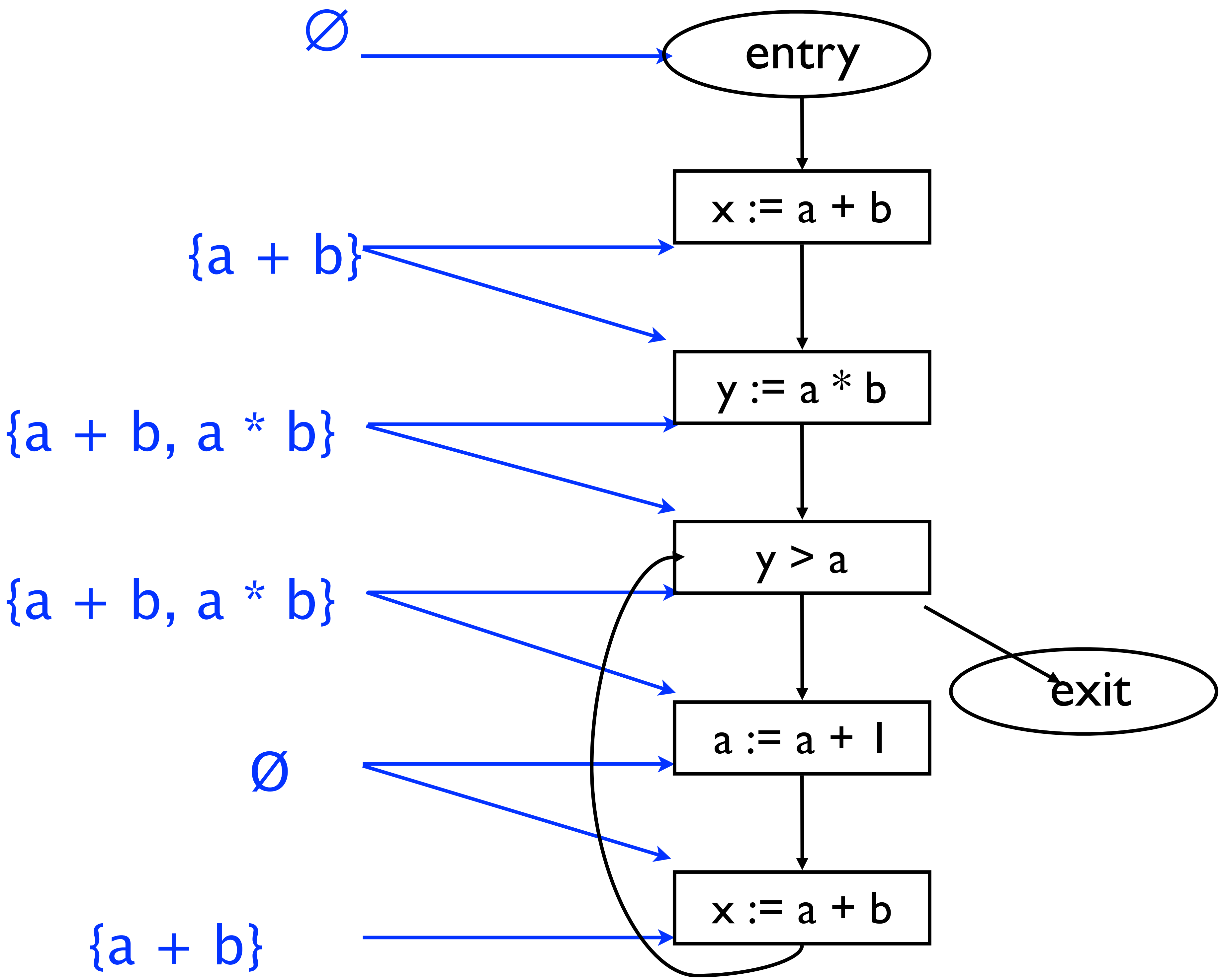
Computing Available Expressions



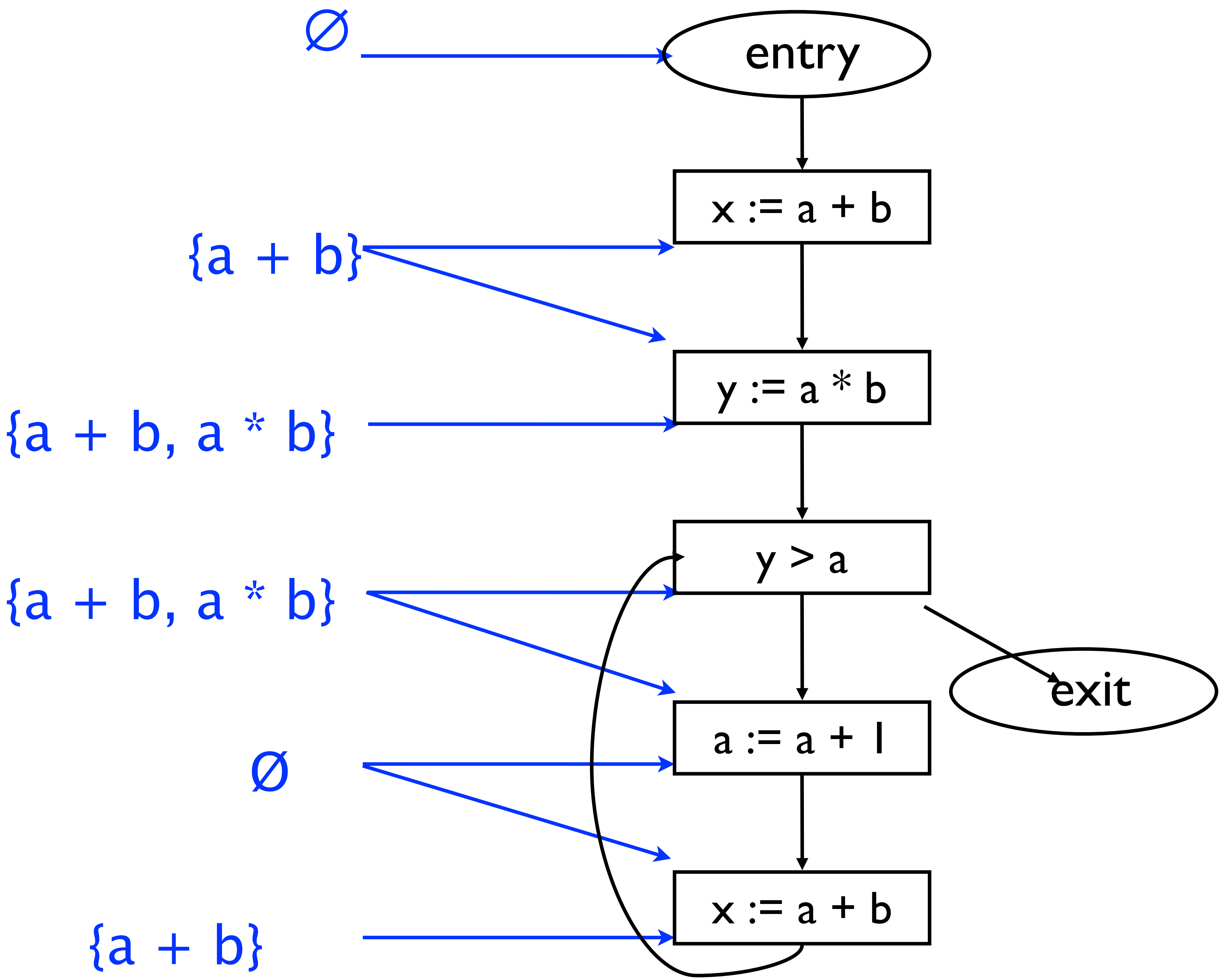
Computing Available Expressions



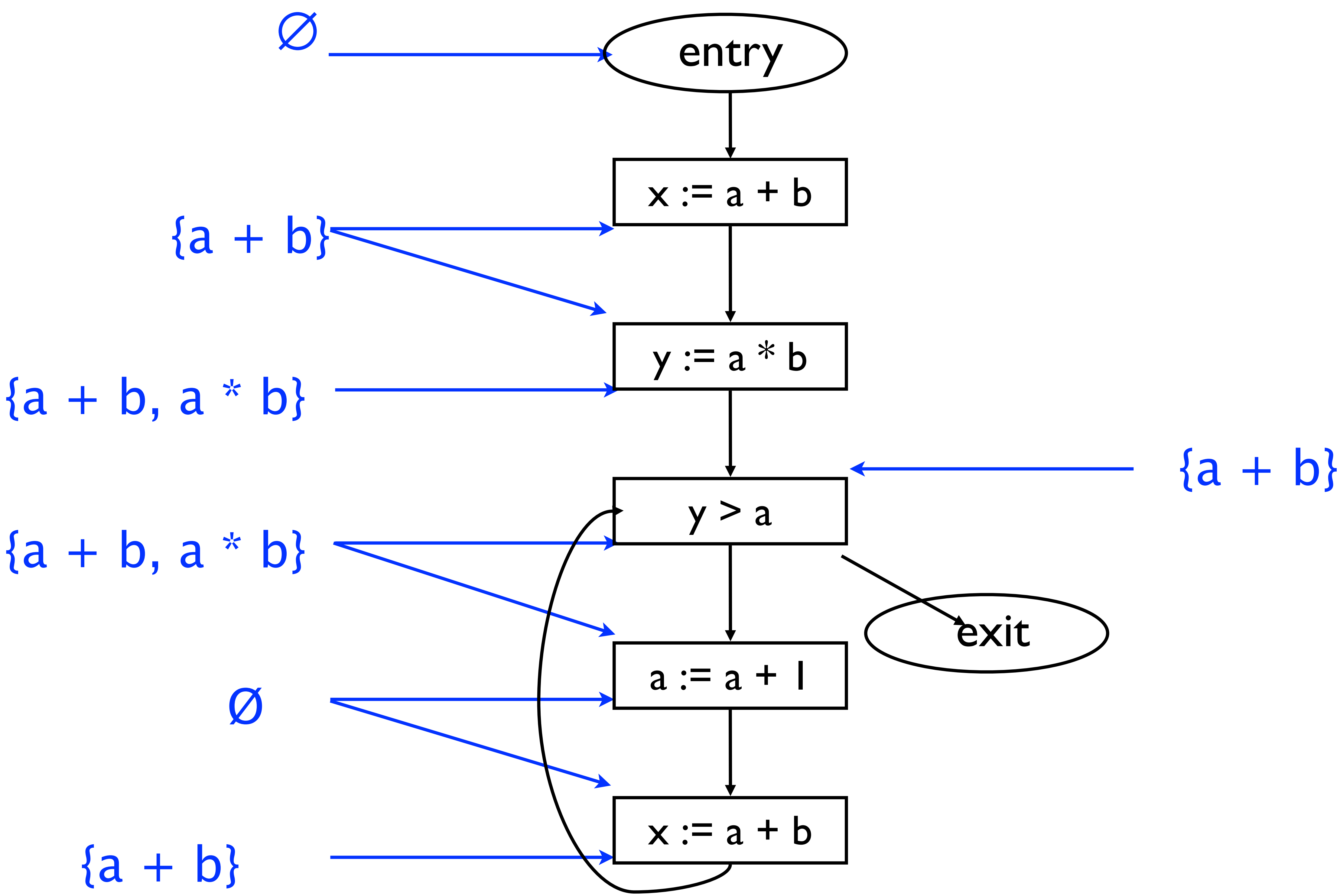
Computing Available Expressions



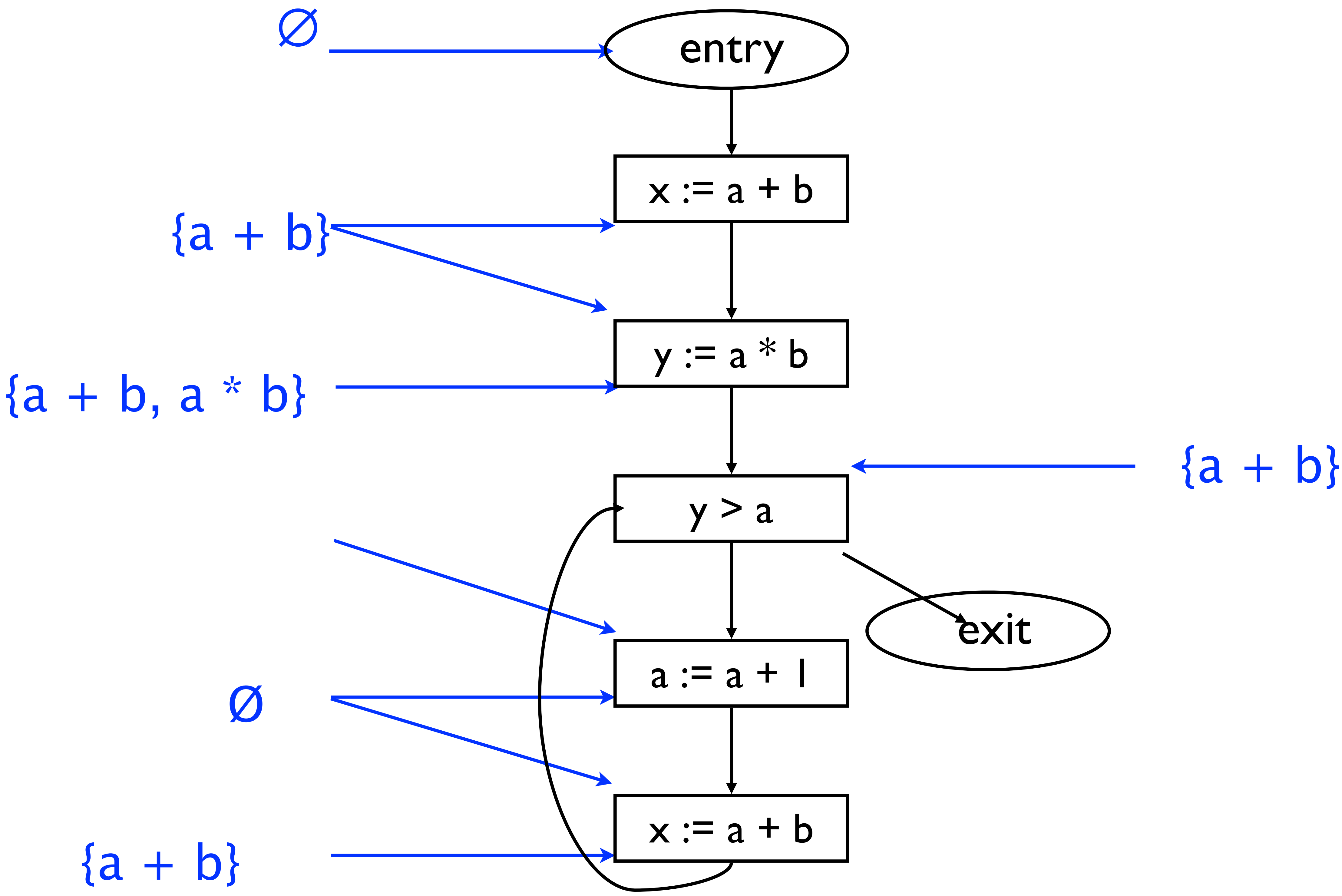
Computing Available Expressions



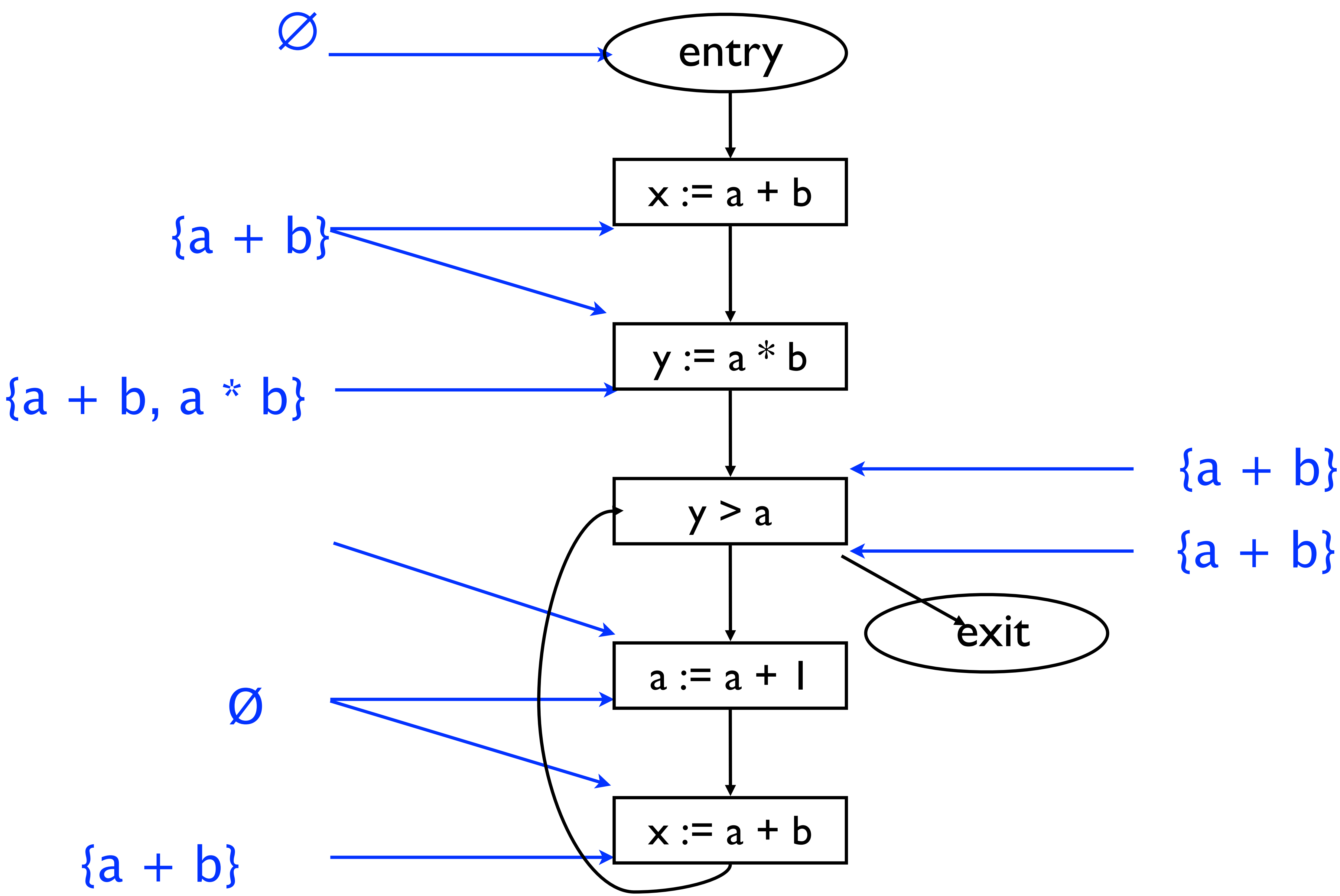
Computing Available Expressions



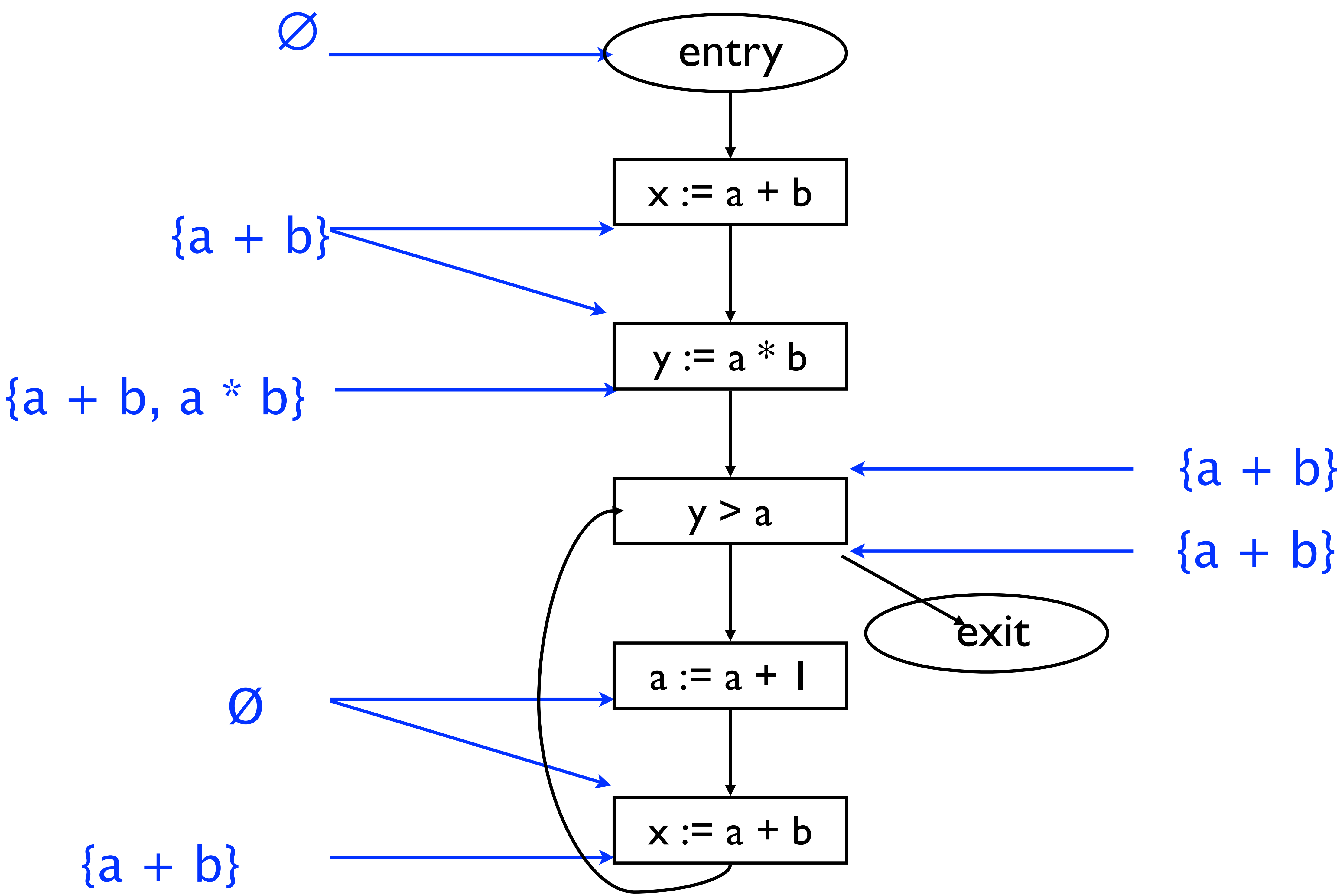
Computing Available Expressions



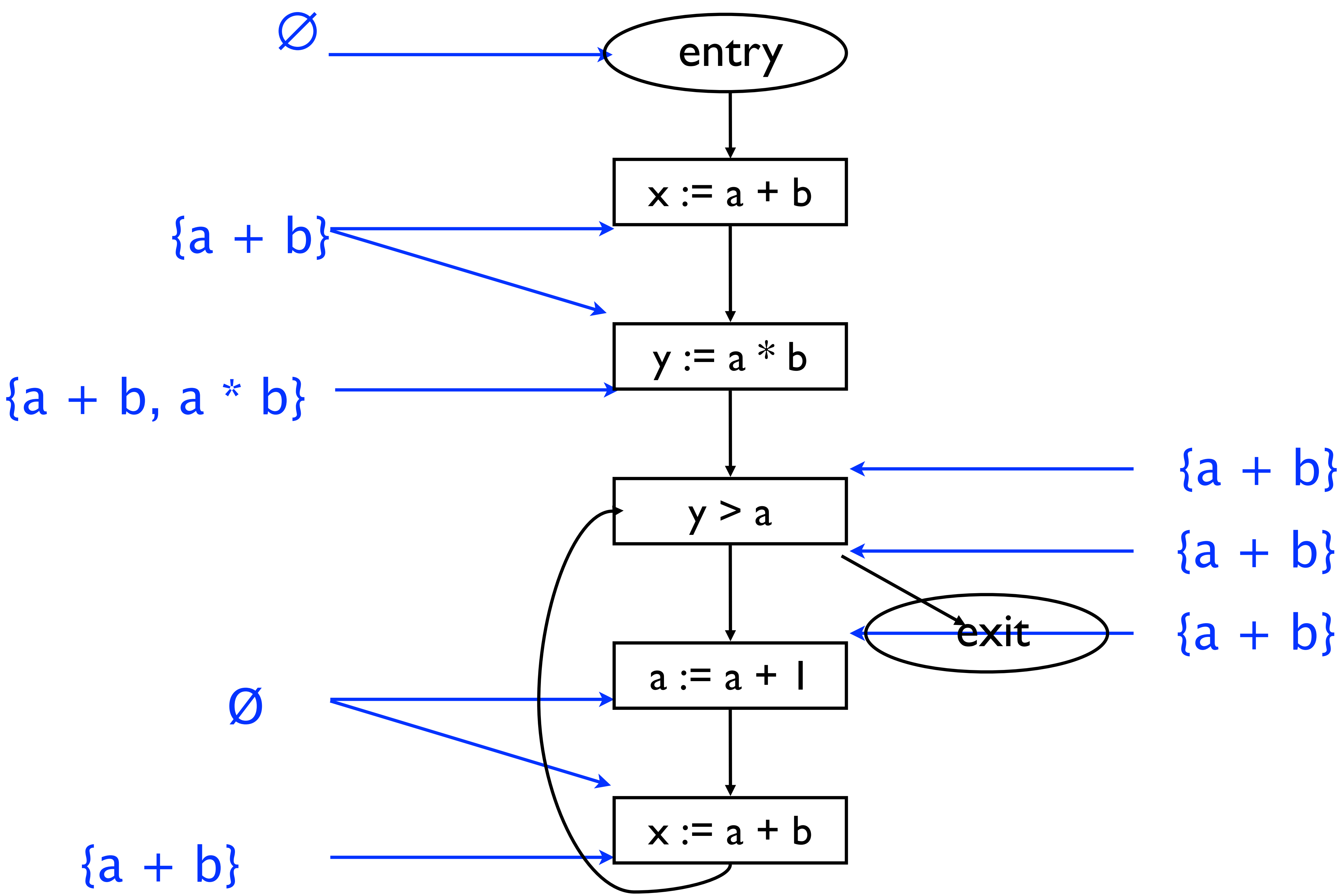
Computing Available Expressions



Computing Available Expressions



Computing Available Expressions



Terminology

- A *join point* is a program point where two branches meet
- Available expressions is a *forward must* problem
  - Forward = Data flow from in to out
  - Must = Property *must* hold on all paths to the join
- Dataflow analysis requires facts that summarize *all paths* to a join point
  - Symbolic execution analyzes each path separately

Data Flow Equations

- Let s be a statement
  - succ(s) = { immediate successor statements of s }
  - pred(s) = { immediate predecessor statements of s}
  - In(s) = program point just before executing s
  - Out(s) = program point just after executing s

- $In(s) = \bigcap_{s' \in pred(s)} Out(s')$

- $Out(s) = Gen(s) \cup (In(s) - Kill(s))$

Liveness Analysis

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

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

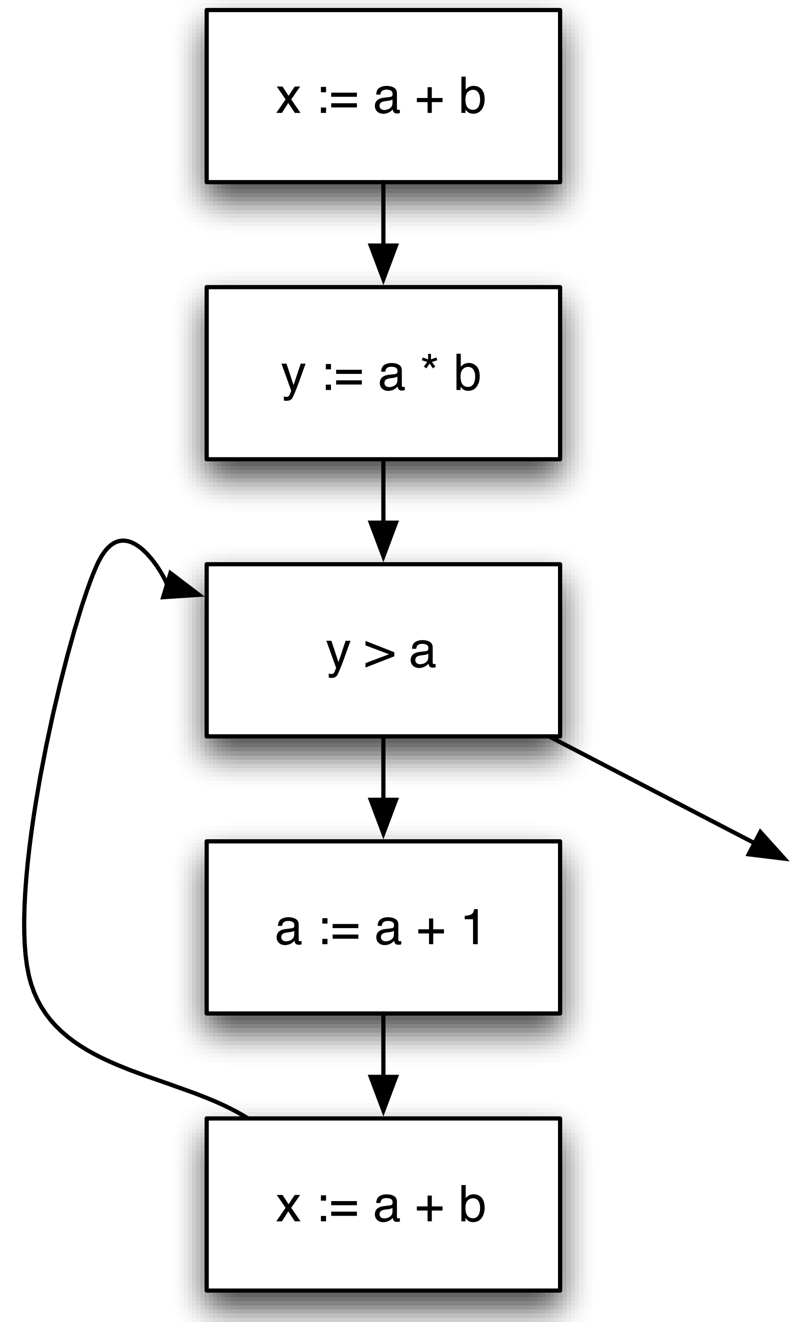
Data Flow Equations

- 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
- $\mathrm{Out}(s) = \bigcup_{s' \in \mathrm{succ}(s)} \mathrm{In}(s')$
- $\mathrm{In}(s) = \mathrm{Gen}(s) \cup (\mathrm{Out}(s) - \mathrm{Kill}(s))$

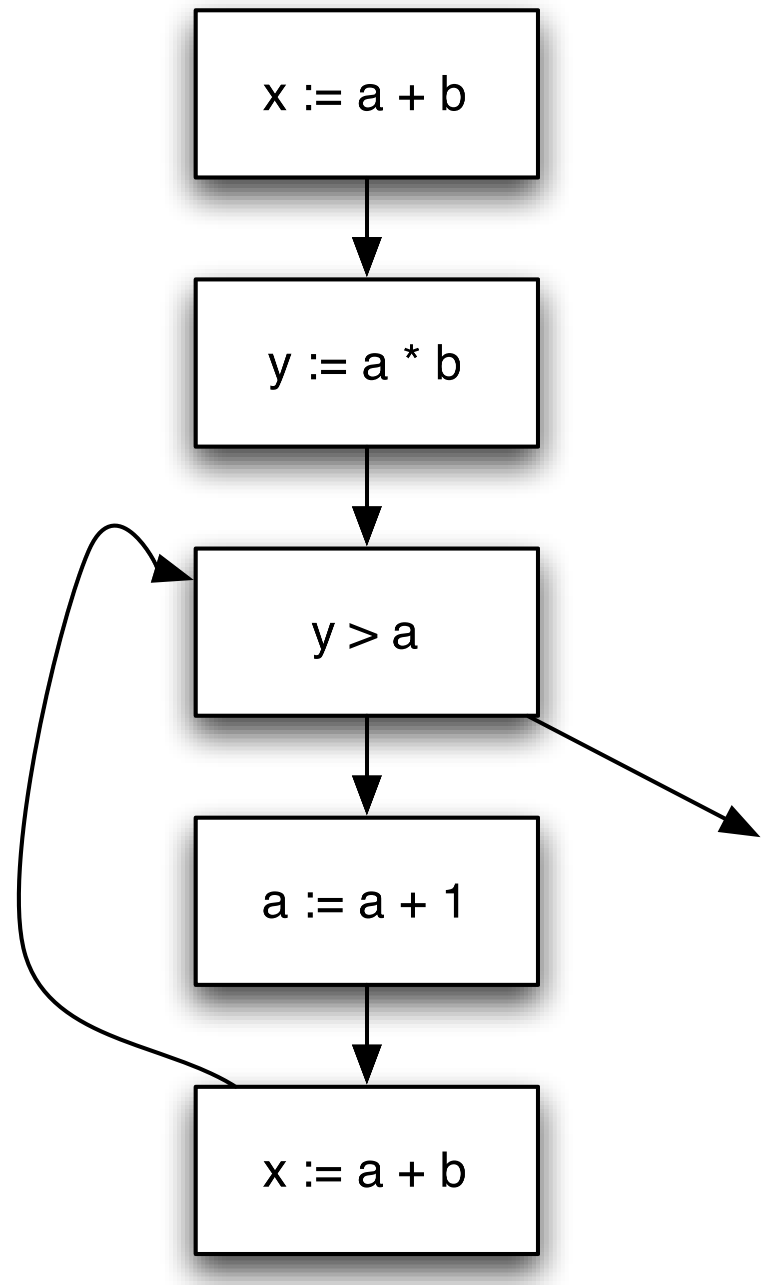
Gen and Kill

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

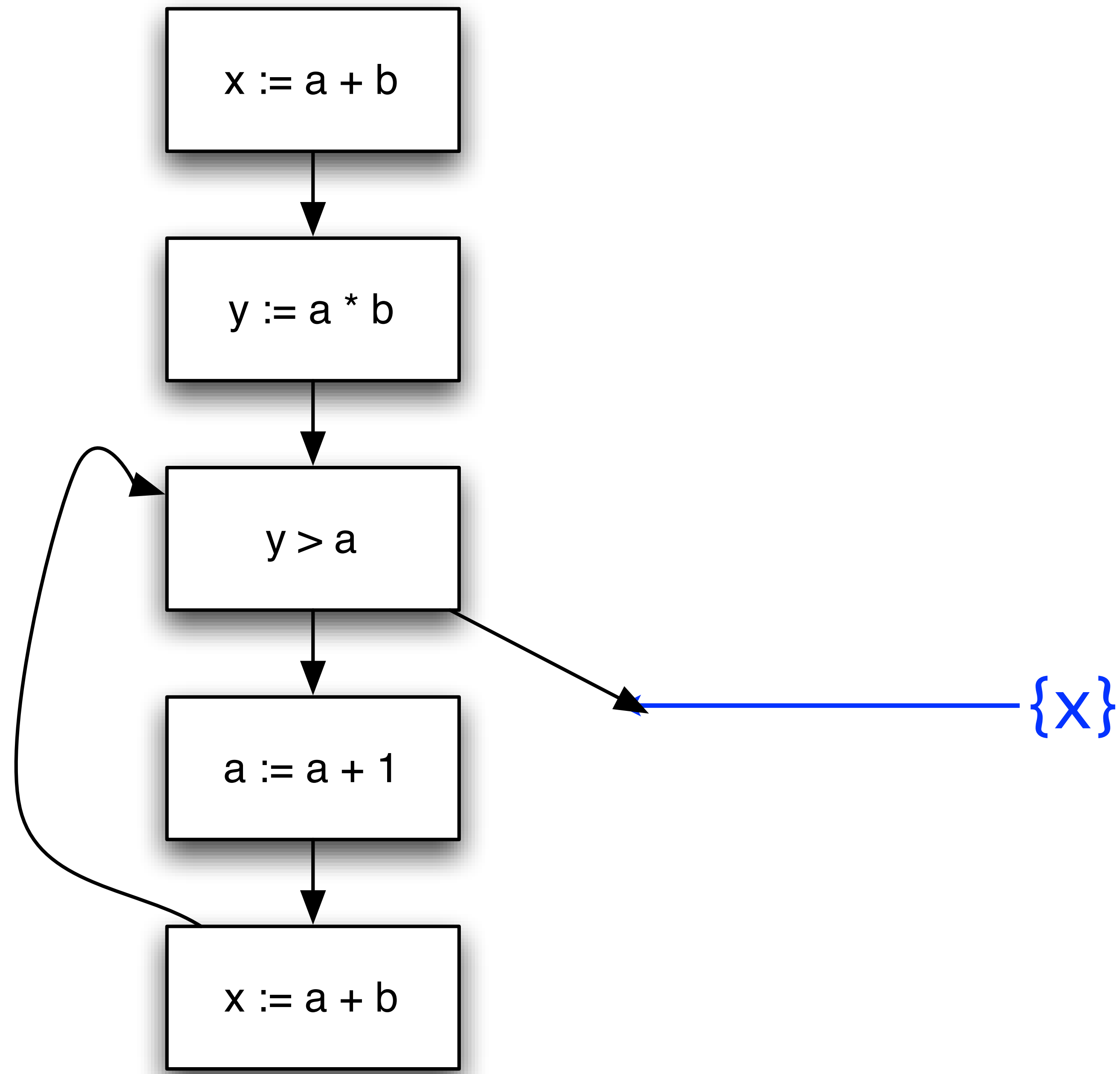
Stmt	Gen	Kill
x := a + b	a, b	x
y := a * b	a, b	y
y > a	a, y	
a := a + 1	a	a



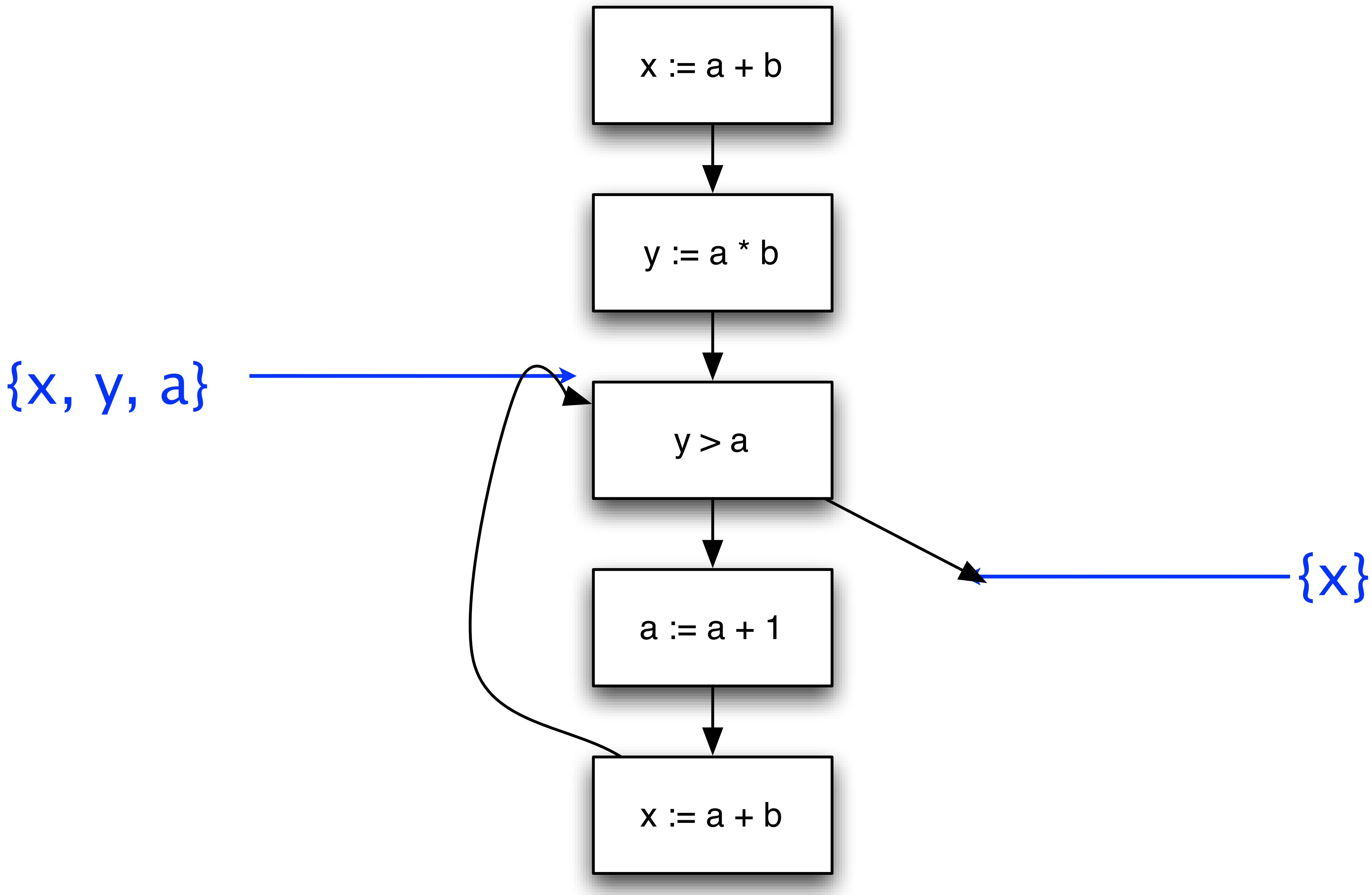
Computing Live Variables



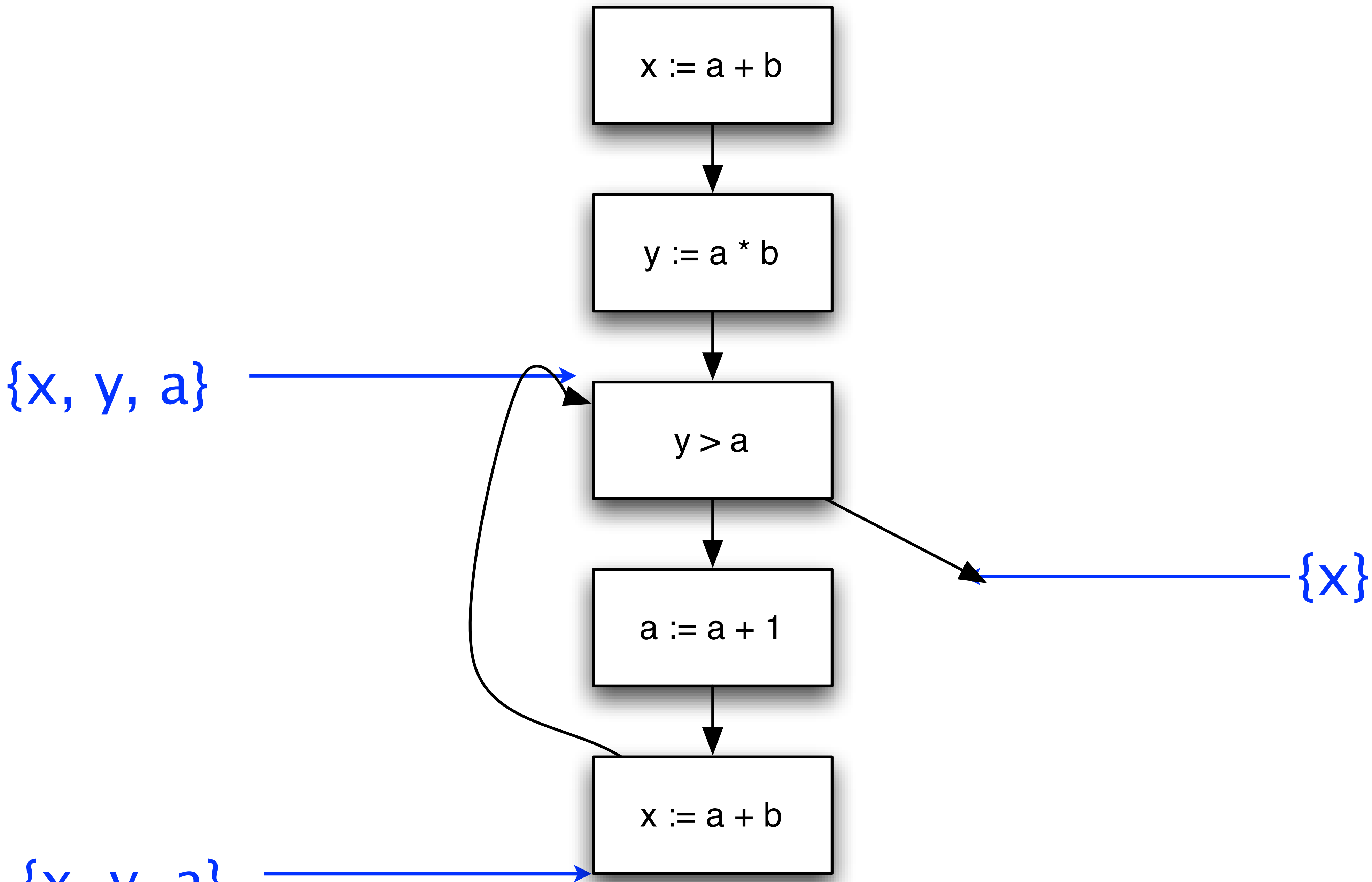
Computing Live Variables



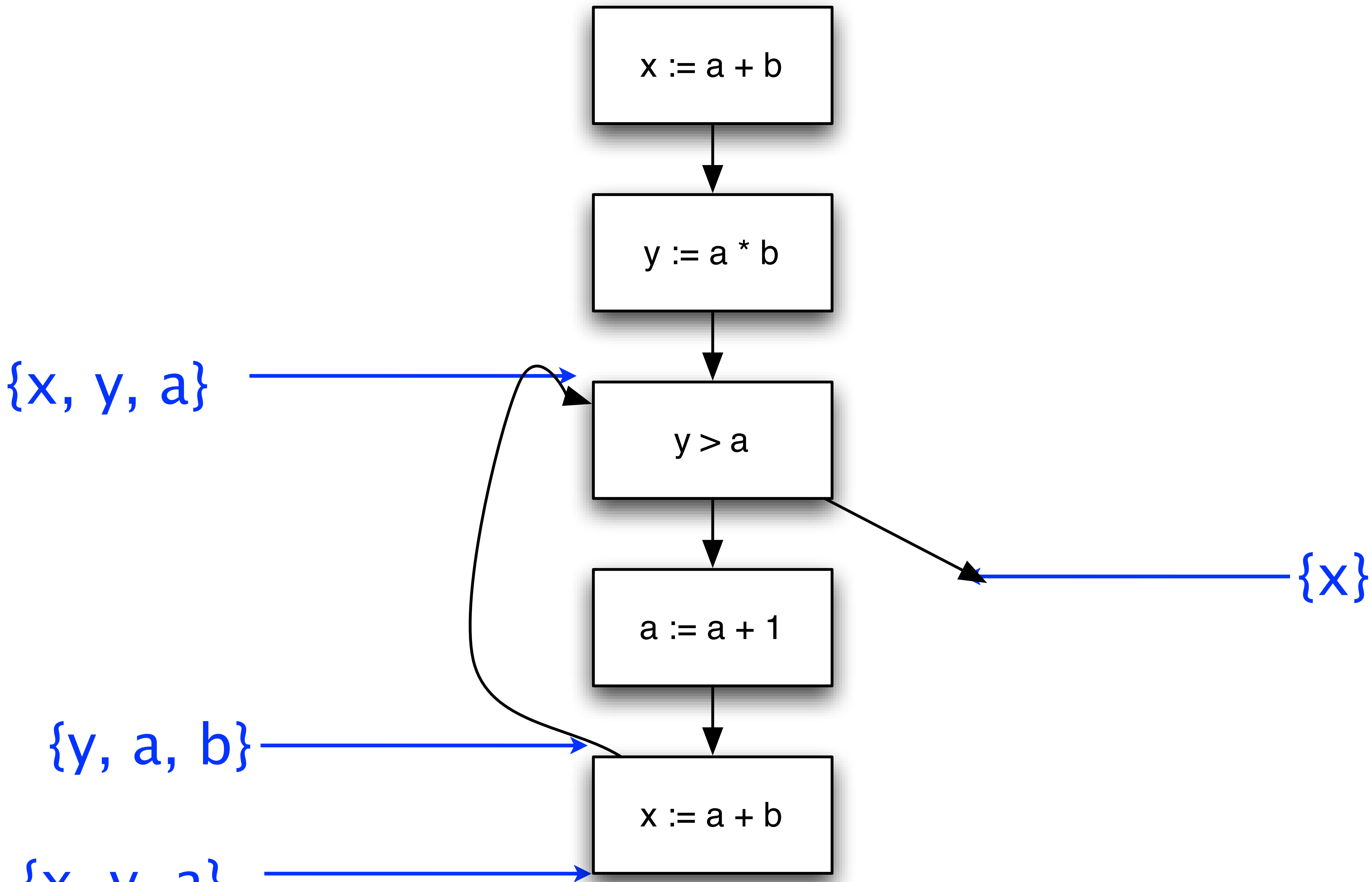
Computing Live Variables



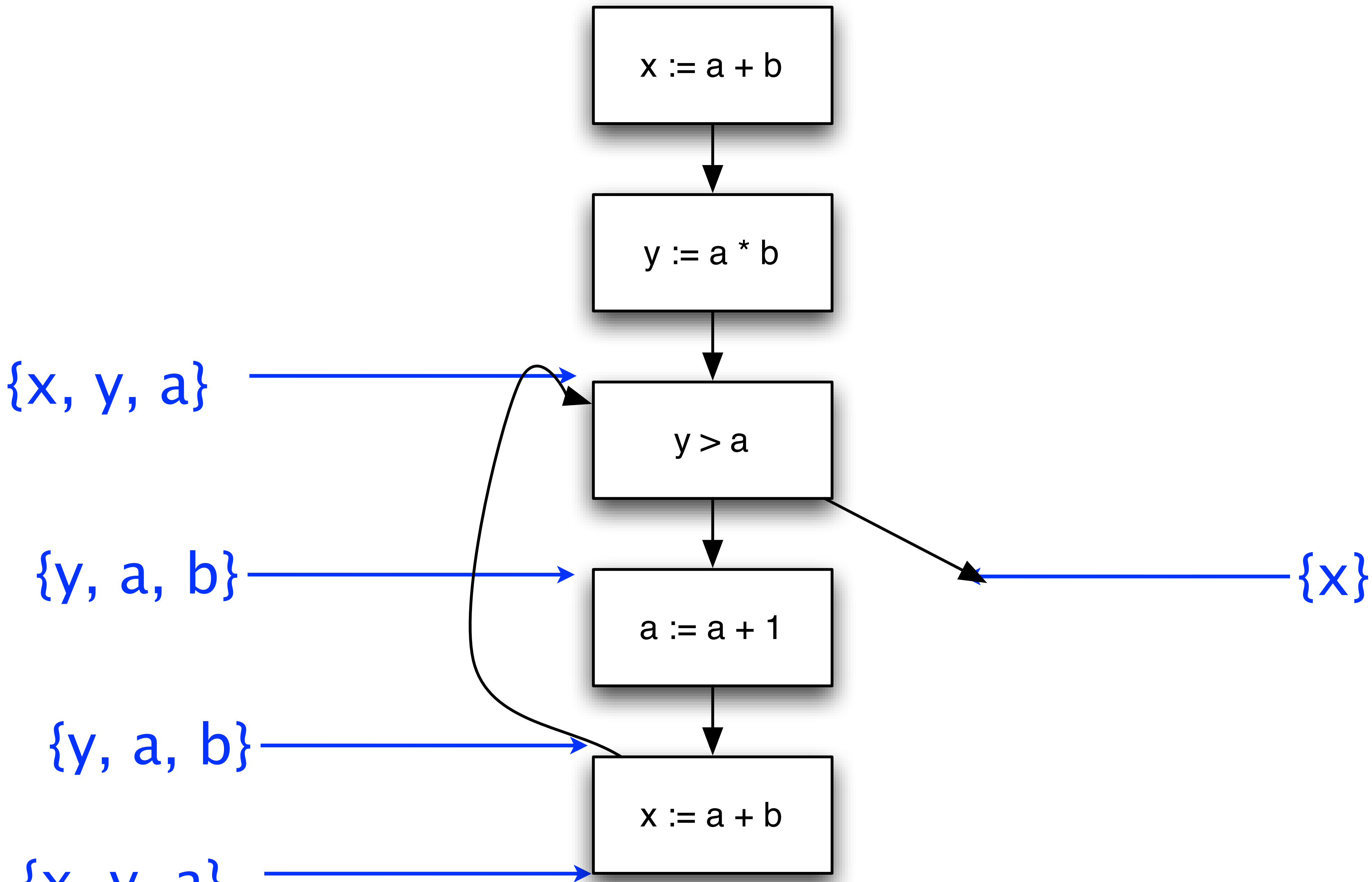
Computing Live Variables



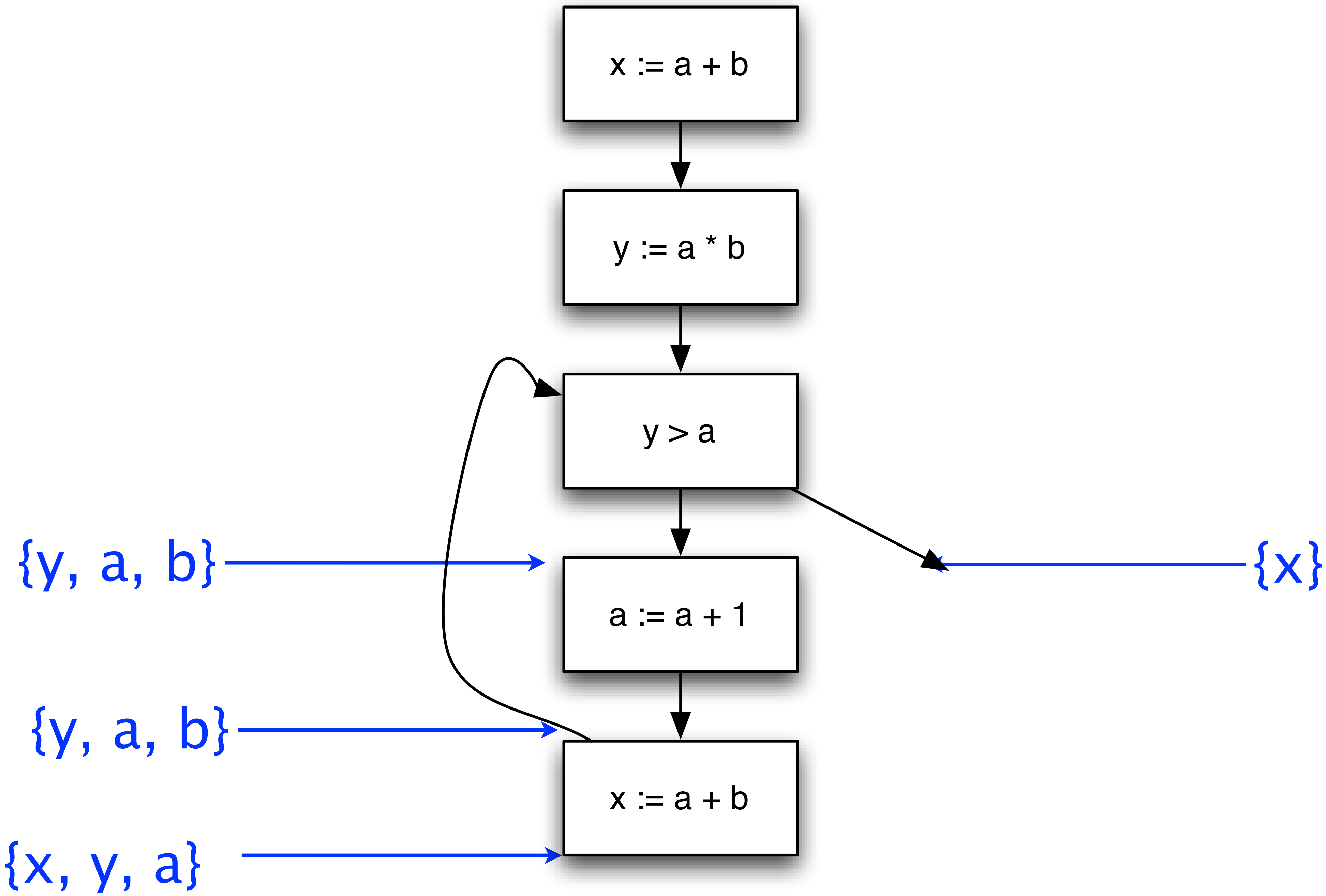
Computing Live Variables



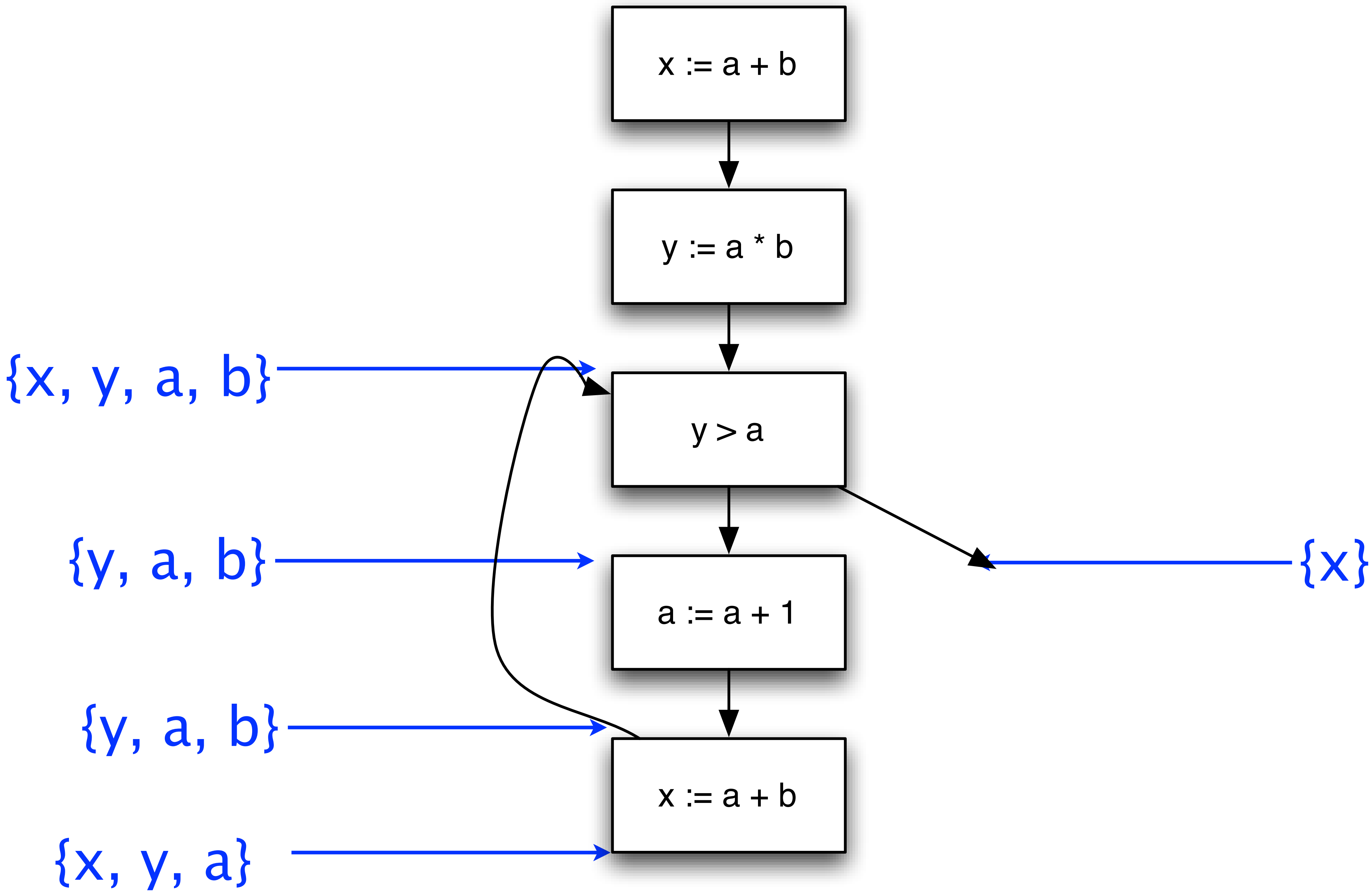
Computing Live Variables



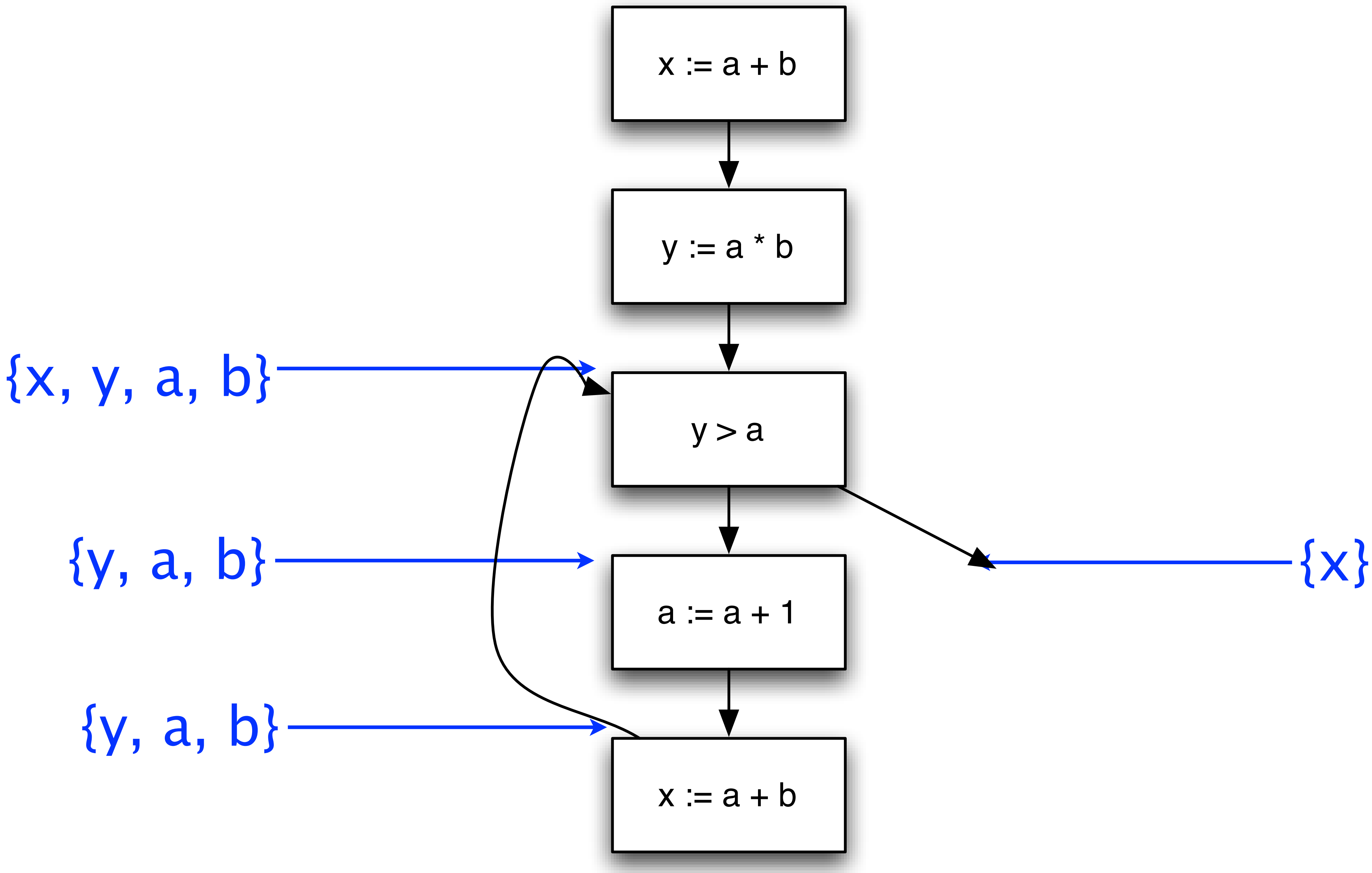
Computing Live Variables



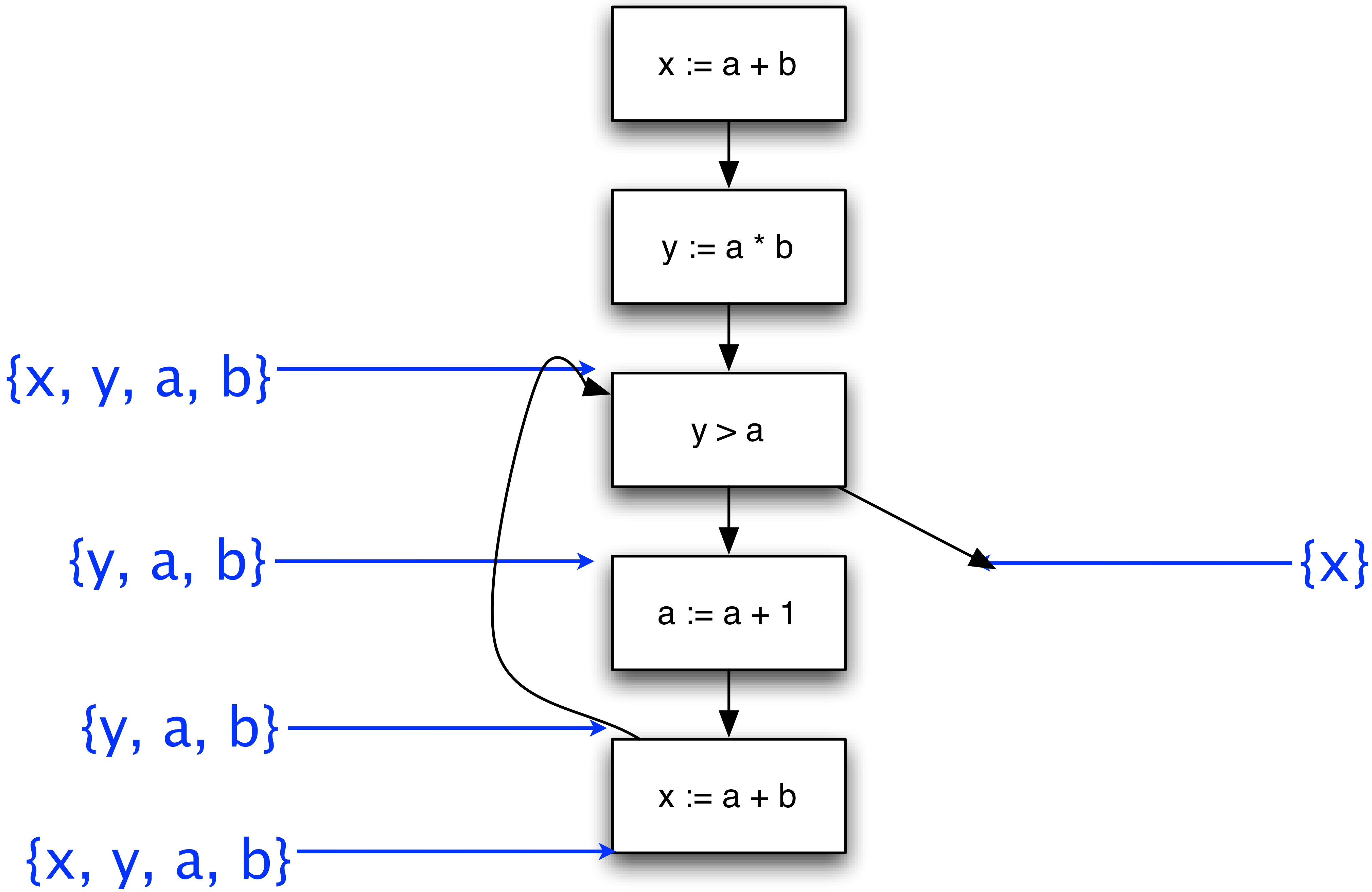
Computing Live Variables



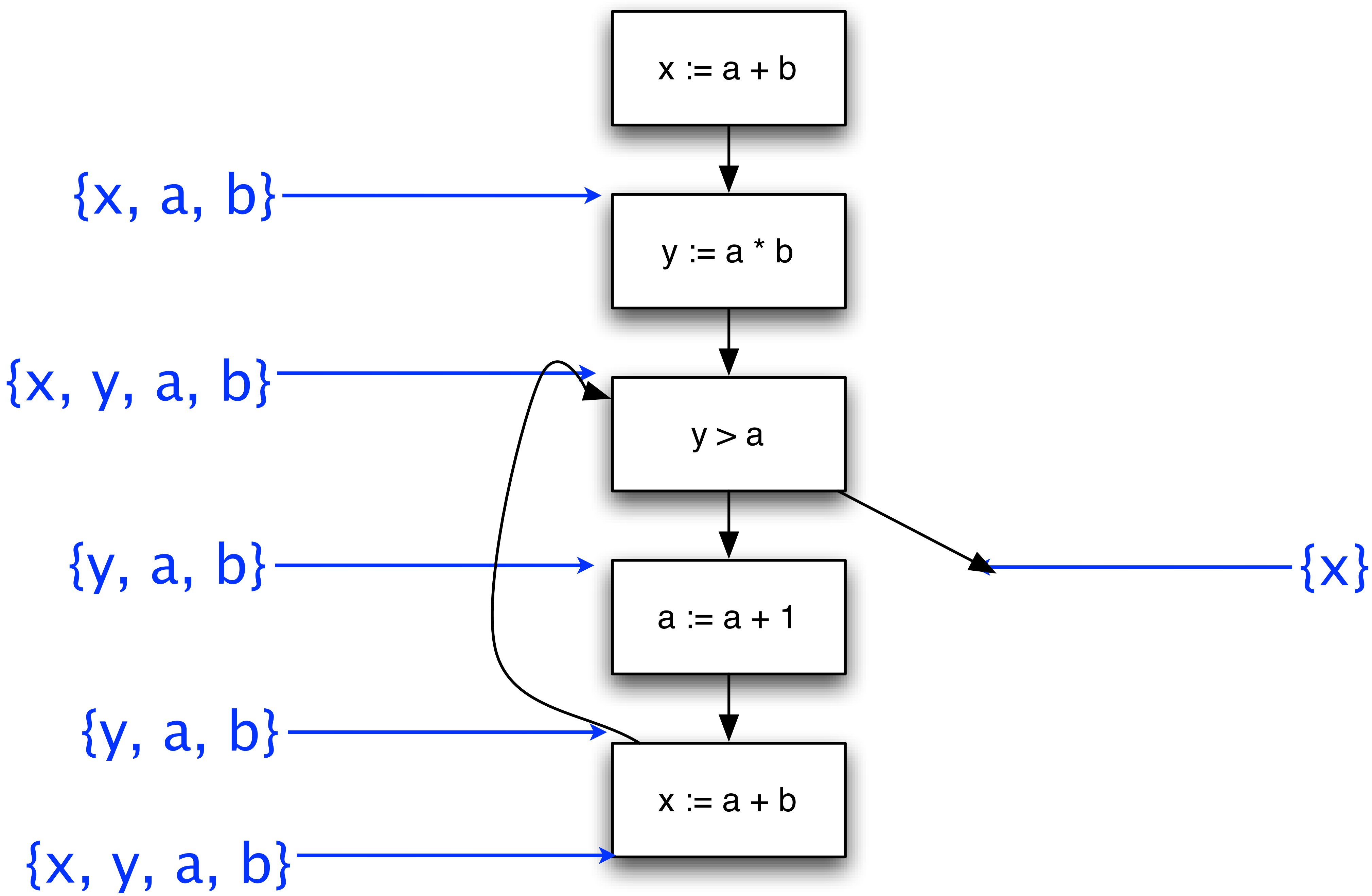
Computing Live Variables



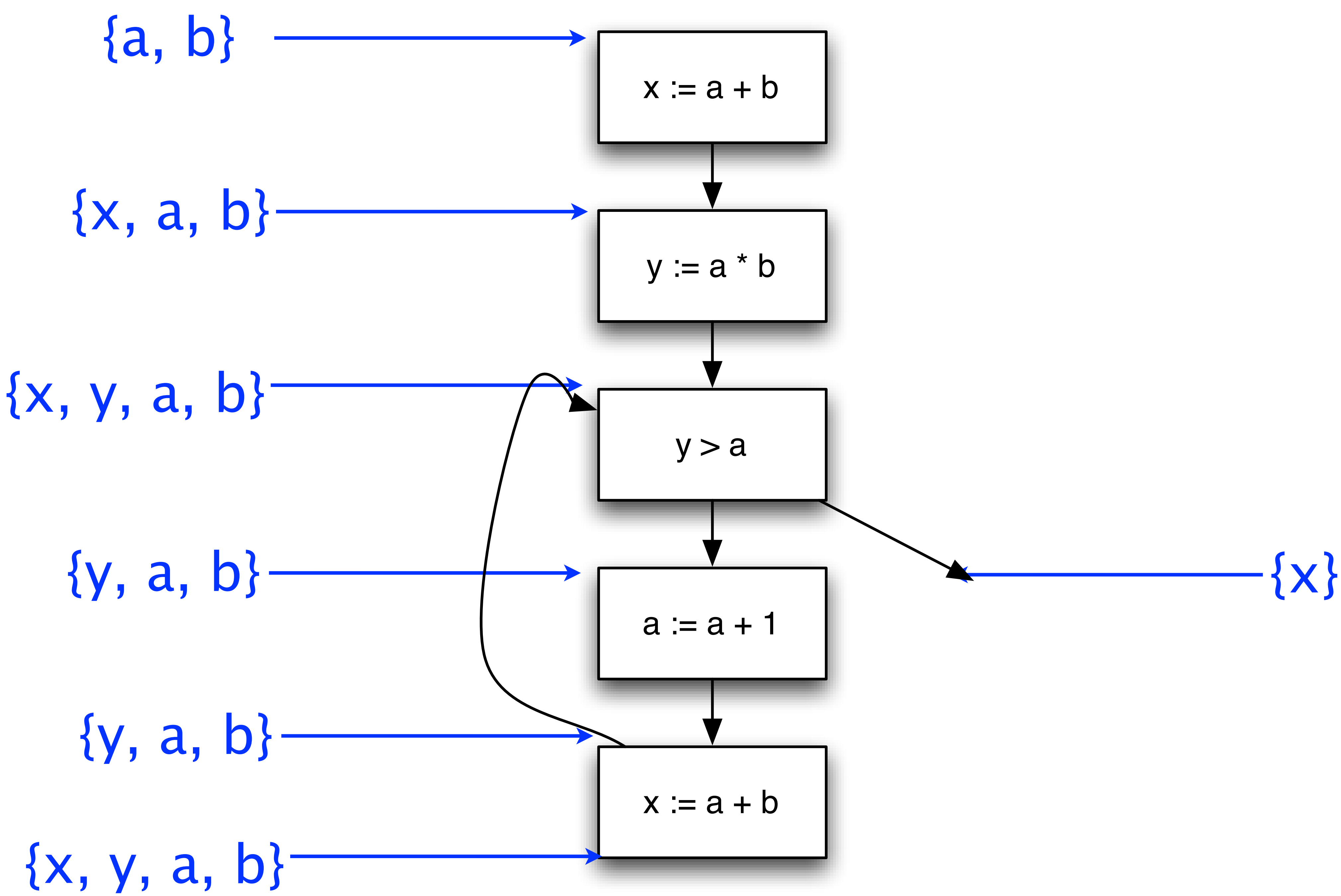
Computing Live Variables



Computing Live Variables



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?

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?

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?
  - 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? forward
  - 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? forward
  - May or must? may

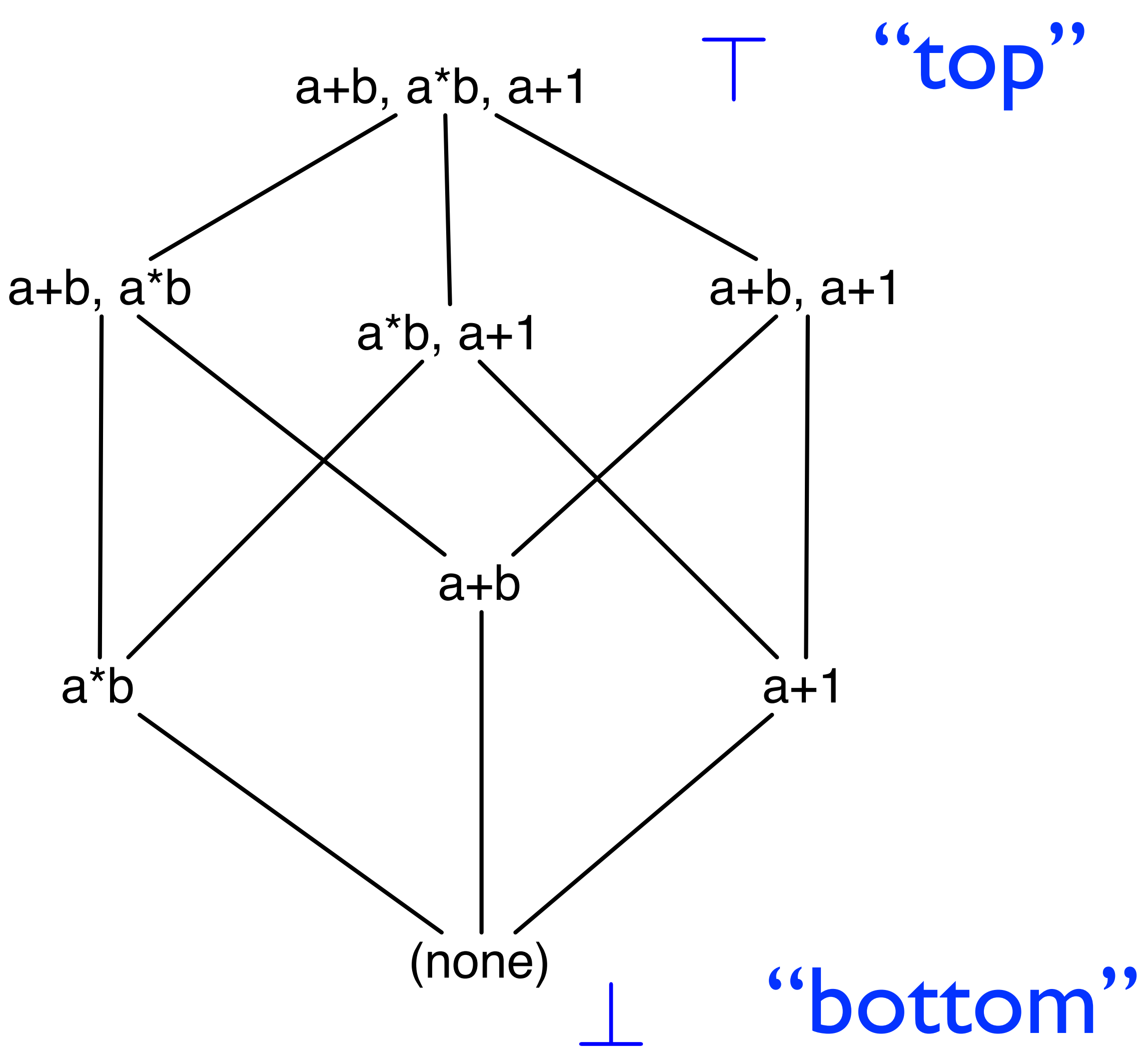
Space of Data Flow Analyses

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

- Most data flow analyses can be classified this way
  - A few don’t fit: bidirectional analysis
- 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

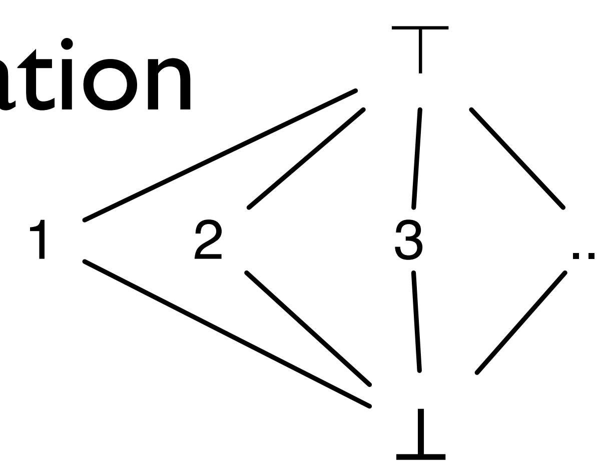
- A partial order is a lattice if $\sqcap$ and $\sqcup$ are defined on any set:
  - $\sqcap$ is the *meet* or *greatest lower bound* operation:
    - $x \sqcap y \leq x$ and $x \sqcap y \leq y$
    - if $z \leq x$ and $z \leq y$, then $z \leq x \sqcap y$
  - $\sqcup$ is the *join* or *least upper bound* 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
  - $x \sqcap \perp = \perp \qquad x \sqcup \perp = x$
  - $x \sqcap \top = x \qquad x \sqcup \top = \top$
- In a lattice, $x \leq y \text{ iff } x \sqcap y = x$
  $x \leq y \text{ iff } x \sqcup y = y$
- A partial order is a *complete lattice* if meet and join are defined on any set $S \subseteq P$

Useful Lattices

- $(2^S, \subseteq)$ forms a lattice for any set S
  - 2^S is the powerset of S (set of all subsets)
- If $(S, \leq)$ is a lattice, so is $(S, \geq)$
  - I.e., lattices can be flipped
- The lattice for constant propagation



Forward Must Data Flow Algorithm

Forward Must Data Flow Algorithm

Out(s) = *Top* for all statements s

Forward Must Data Flow Algorithm

Out(s) = *Top* for all statements s

// Slight acceleration: Could set Out(s) = Gen(s) U (*Top* - Kill(s))

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)

Forward Must Data Flow Algorithm

Out(s) = *Top* for all statements s

// Slight acceleration: Could set Out(s) = Gen(s) U (*Top* - Kill(s))

W := { all statements }   (worklist)

repeat

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

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

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

temp := Gen(s) ∪ (In(s) - Kill(s))

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

temp := Gen(s) ∪ (In(s) - Kill(s))

if (temp != Out(s)) {

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

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

  temp := Gen(s) ∪ (In(s) - Kill(s))

  if (temp != Out(s)) {

    Out(s) := temp

    W := W ∪ succ(s)

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

temp := Gen(s) ∪ (In(s) - Kill(s))

if (temp != Out(s)) {

Out(s) := temp

W := W ∪ succ(s)

}

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
  - $\mathrm{In}(s) := \cap_{s' \in \mathrm{pred}(s)} \mathrm{Out}(s')$
  - $\mathrm{temp} := \mathrm{Gen}(s) \cup (\mathrm{In}(s) - \mathrm{Kill}(s))$
- Putting these two together,
  - temp := $f_s(\sqcap_{s' \in \mathrm{pred}(s)} Out(s'))$

Monotonicity

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

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

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

a function f_s(In(s))

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

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 ∪ succ(s)

  }

until W = ∅

Lattices (P, ≤)

Lattices (P, ≤)

- Available expressions

Lattices (P, $\leq$)

- Available expressions
  - P = sets of expressions

Lattices (P, $\leq$)

- Available expressions
  - P = sets of expressions
  - $S1 \sqcap S2 = S1 \cap S2$

Lattices (P, ≤)

- Available expressions
  - P = sets of expressions
  - S1 ⊓ S2 = S1 ∩ S2
  - *Top* = set of all expressions

Lattices (P, ≤)

- Available expressions
  - P = sets of expressions
  - S1 $\sqcap$ S2 = S1 $\cap$ S2
  - *Top* = set of all expressions
- Reaching Definitions

Lattices (P, ≤)

- Available expressions
  - P = sets of expressions
  - S1 ⊓ S2 = S1 ∩ S2
  - *Top* = set of all expressions
- Reaching Definitions
  - P = set of definitions (assignment statements)

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)
  - $S1 \sqcap S2 = S1 \cup S2$

Lattices (P, ≤)

- 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
  - 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

Lattices (P, ≤), cont'd

- Live variables

Lattices (P, ≤), cont'd

- Live variables
  - P = sets of variables

Lattices (P, ≤), cont'd

- Live variables
  - P = sets of variables
  - S1 $\sqcap$ S2 = S1 $\cup$ S2

Lattices (P, ≤), cont'd

- Live variables
  - P = sets of variables
  - S1 $\sqcap$ S2 = S1 $\cup$ S2
  - Top = empty set

Lattices (P, ≤), cont'd

- Live variables
  - P = sets of variables
  - $S1 \sqcap S2 = S1 \cup S2$
  - Top = empty set

Lattices (P, ≤), cont'd

- Live variables
  - P = sets of variables
  - S1 ⊓ S2 = S1 ∪ S2
  - Top = empty set

- Very busy expressions

Lattices (P, $\leq$), cont'd

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

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
  - $S1 \sqcap S2 = S1 \cap S2$

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 := $f_s(\sqcap_{s' \in pred(s)} Out(s'))$
  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 := $f_s(\sqcap_{s' \in succ(s)} In(s'))$
  if (temp != In(s)) {
    In(s) := temp
    W := W ∪ pred(s)
  }
until W = ∅

Termination Revisited

- How many times can we apply this step:

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

  if (temp != Out(s)) { ... }

  - Claim: Out(s) only shrinks
    - Proof: Out(s) starts out as *Top*
      - So temp must be $\leq$ than *Top* after first step
    - Assume Out(s′) shrinks for all predecessors s′ of s
    - Then $\sqcap_{s' \in pred(s)} Out(s')$ shrinks
    - Since f_s monotonic, $f_s(\sqcap_{s' \in pred(s)} Out(s'))$ shrinks

Termination Revisited (cont'd)

- A *descending chain* in a lattice is a sequence
  - $x0 \sqsupset x1 \sqsupset x2 \sqsupset \ldots$
- 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 takes $O(1)$ time

Least vs. Greatest 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

Distributive Data Flow Problems

- By monotonicity, we also have

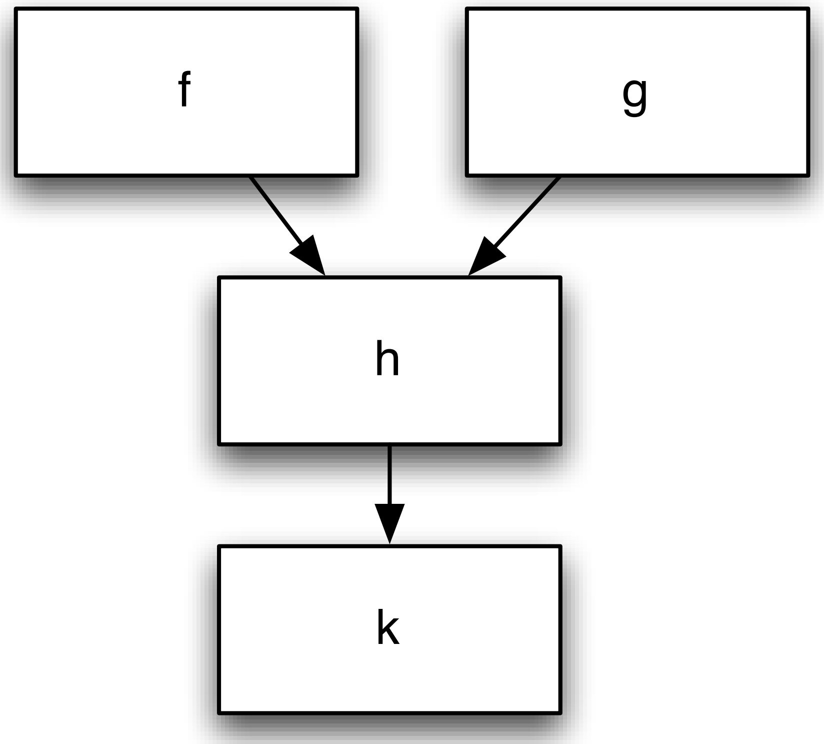
$$f(x \sqcap y) \leq f(x) \sqcap f(y)$$

- A function f is distributive if

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

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 $\{s_1, ..., s_n\}$, let $f_p = f_n;...;f_1$
  - Let path(s) be the set of paths from the entry to s

$$\mathrm{MOP}(s) = \sqcap_{p \in \mathrm{path}(s)} f_p(\top)$$

- If a data flow problem is distributive, then solving the data flow equations in the standard way yields the MOP solution

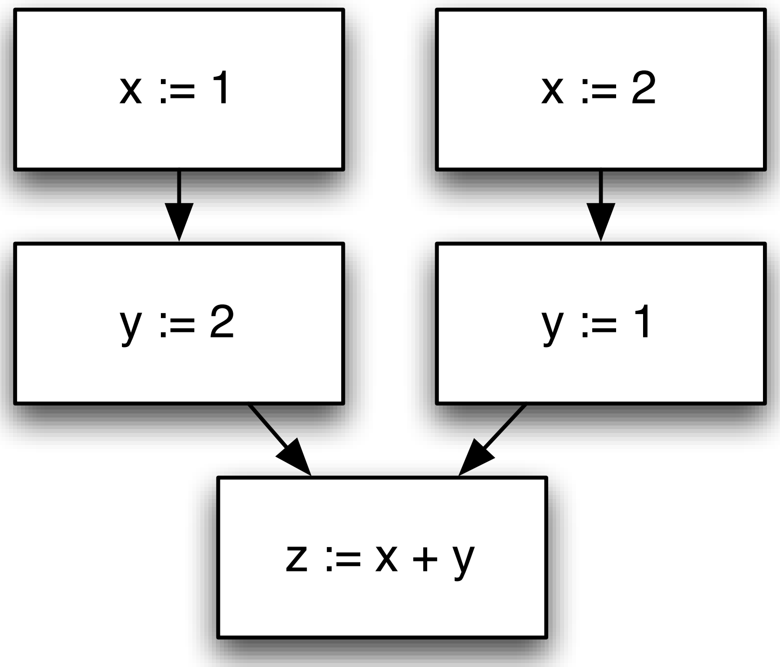
What Problems are Distributive?

- Analyses of *how* the program computes
  - Live variables
  - Available expressions
  - Reaching definitions
  - Very busy expressions

- All Gen/Kill problems are distributive

A Non-Distributive Example

- Constant propagation



- In general, analysis of *what* the program computes in not distributive
  - Symbolic execution more accurate for nondistributive problems

Practical Implementation

- Data flow facts = assertions that are true or false at a program point
- 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 s.t.
  - No statement except the last in a branch
  - There are no branches to any statement in the block except the first

- 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 ~5 statements

Order Matters

- 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 Matters — Cycles

- 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 $O((Q+1)|E|)$
    - Note direction of req't depends on top vs. bottom

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
    - /* x : int */ x := ... /* x : int */

Terminology Review

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

Data Flow Analysis and Functions

- What happens at a function call?
  - Lots of proposed solutions in data flow analysis literature
- In practice, only analyze one procedure at a time
- Consequences
  - Call to function kills all data flow facts
  - May be able to improve depending on language, e.g., function call may not affect locals

More Terminology

- An analysis that models only a single function at a time is *intraprocedural*
- An analysis that takes multiple functions into account is *interprocedural*
- An analysis that takes the whole program into account is...guess?

- Note: *global* analysis means "more than one basic block," but still within a function

Data Flow 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

Data Flow Analysis and Optimization

Data Flow Analysis and Optimization

- Moore's Law: Hardware advances double computing power every 18 months.

Data Flow Analysis and Optimization

- Moore's Law: Hardware advances double computing power every 18 months.

- Proebsting's Law: Compiler advances double computing power every 18 *years*.

Data Flow Analysis and Optimization

- Moore's Law: Hardware advances double computing power every 18 months.

- Proebsting's Law: Compiler advances double computing power every 18 *years*.

  - Not so much bang for the buck!

DF Analysis and Defect Detection

- LCLint - Evans et al. (UVa)
- METAL - Engler et al. (Stanford, now Coverity)
- ESP - Das et al. (MSR)
- FindBugs - Hovemeyer, Pugh (Maryland)
  - For Java. The first three are for C.
- Many other one-shot projects
  - Memory leak detection
  - Security vulnerability checking (tainting, info. leaks)

Appendix: Elimination

- Recall using one transfer function per basic block

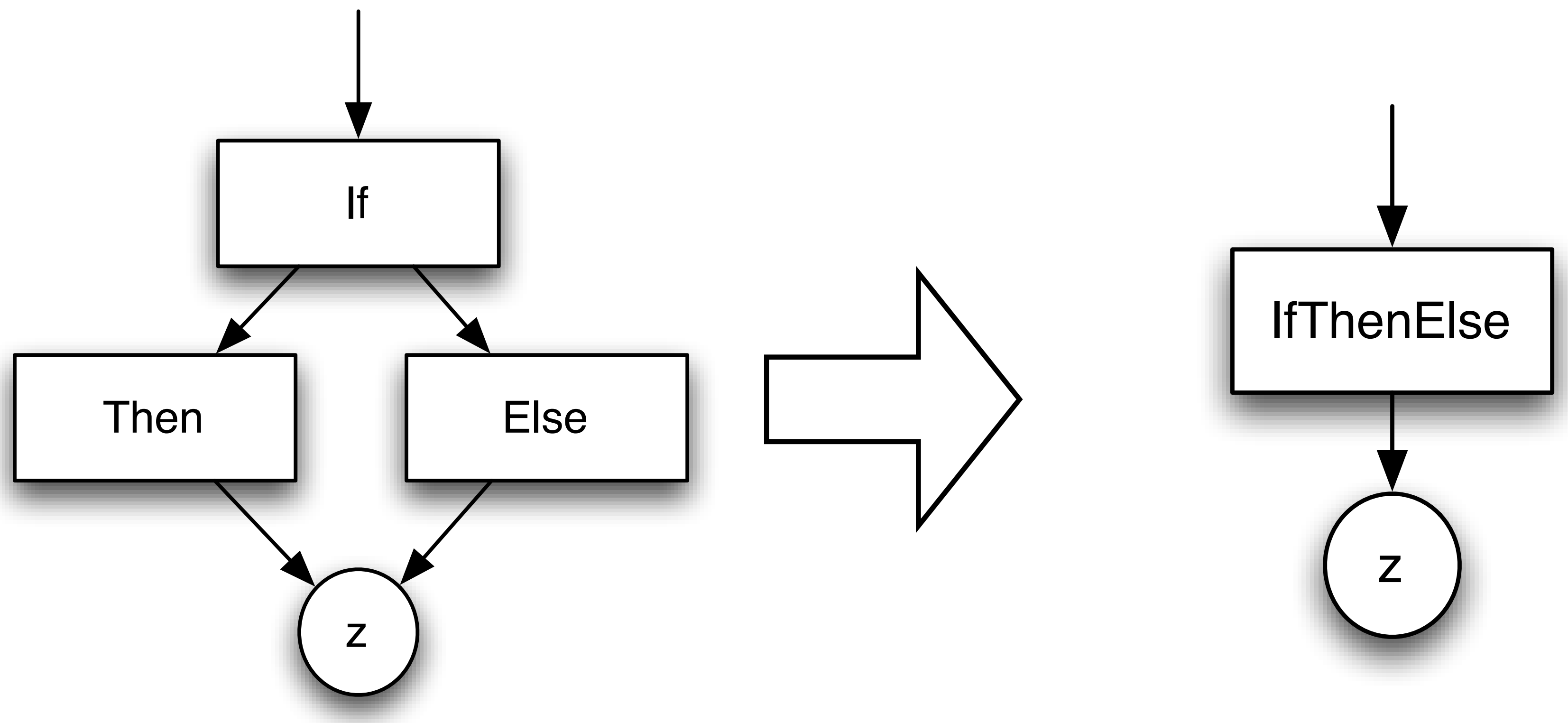
- Why not generalize this idea beyond a basic block?
  - "Collapse" larger constructs into smaller ones, combining data flow equations
  - Eventually program collapsed into a single node!
  - "Expand out" back to original constructs, rebuilding information

Lattices of Functions

- Let $(P, \leq)$ be a lattice
- Let M be the set of monotonic functions on P
- Define $f \leq_f g$ if for all x, $f(x) \leq g(x)$
- Define the function $f \sqcap g$ as
  - $(f \sqcap g)(x) = f(x) \sqcap g(x)$

- Claim: $(M, \leq_f)$ forms a lattice

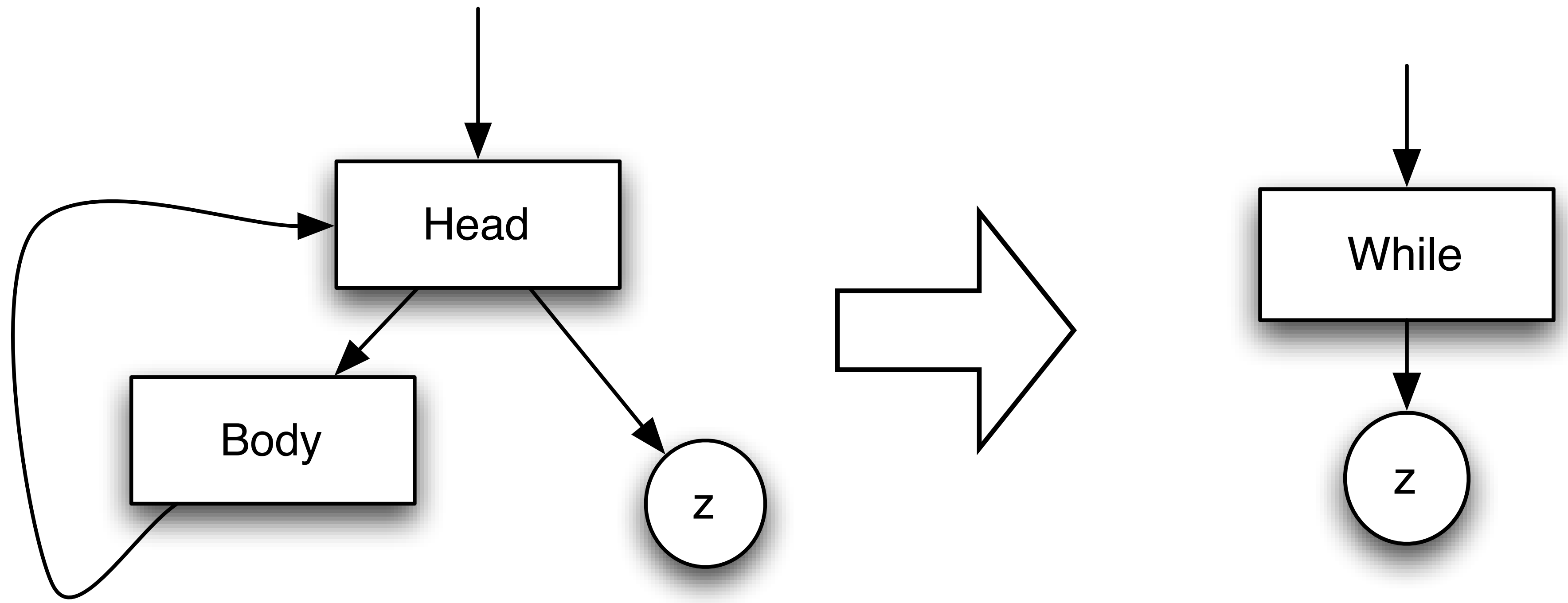
Elimination Methods: Conditionals



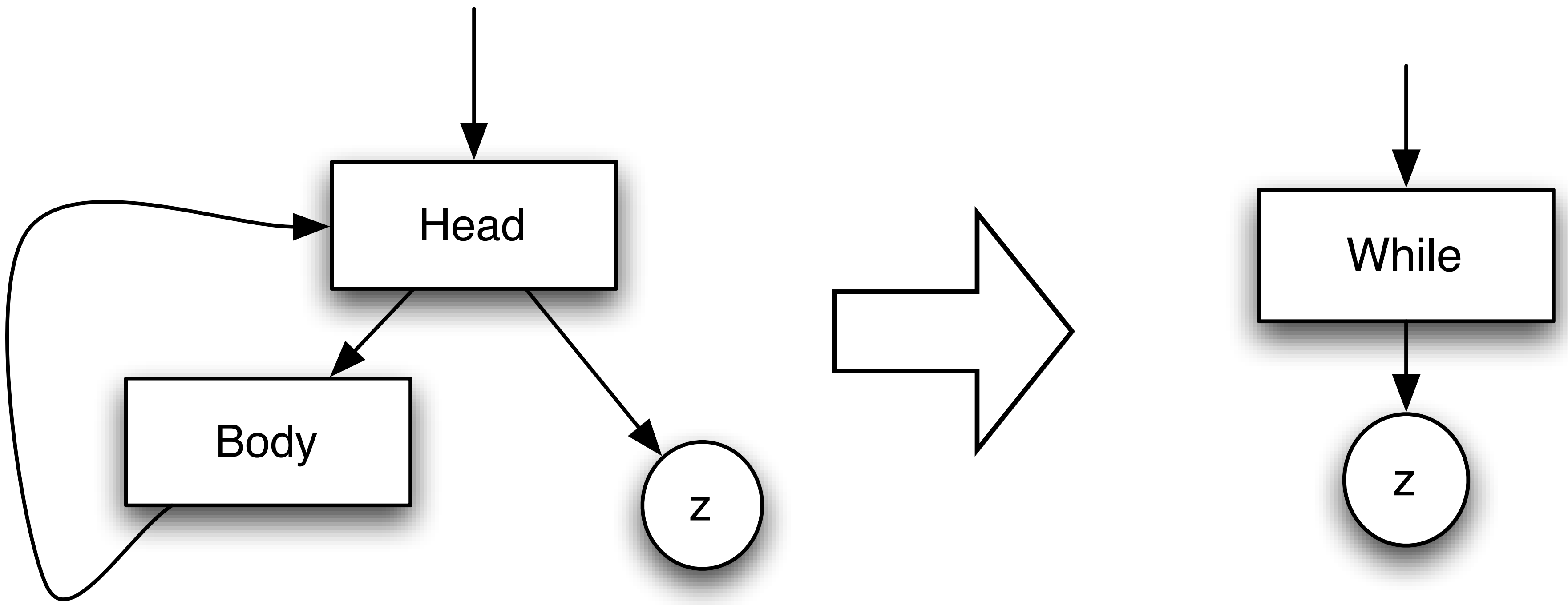
$$f_{\mathrm{ite}} = (f_{\mathrm{then}} \circ f_{\mathrm{if}}) \sqcap (f_{\mathrm{else}} \circ f_{\mathrm{if}})$$

$$\begin{aligned}
\mathrm{Out(if)} &= f_{\mathrm{if}}(\mathrm{In(ite)})) \\
\mathrm{Out(then)} &= (f_{\mathrm{then}} \circ f_{\mathrm{if}})(\mathrm{In(ite)})) \\
\mathrm{Out(else)} &= (f_{\mathrm{else}} \circ f_{\mathrm{if}})(\mathrm{In(ite)}))
\end{aligned}$$

Elimination Methods: Loops



Elimination Methods: Loops



$$
\begin{aligned}
f_{\text{while}} \quad = \quad & f_{\text{head}} \sqcap \\
& f_{\text{head}} \circ f_{\text{body}} \circ f_{\text{head}} \sqcap \\
& f_{\text{head}} \circ f_{\text{body}} \circ f_{\text{head}} \circ f_{\text{body}} \circ f_{\text{head}} \sqcap \cdots
\end{aligned}
$$

Elimination Methods: Loops (cont'd)

- Let $f^i = f \circ f \circ \ldots \circ f$ (i times)
  - $f^0 = id$
- Let
  $$g(j) = \sqcap_{i \in [0..j]} (f_{\text{head}} \circ f_{\text{body}})^i \circ f_{\text{head}}$$
- Need to compute limit as j goes to infinity
  - Does such a thing exist?
- Observe: $g(j+1) \leq g(j)$

Height of Function Lattice

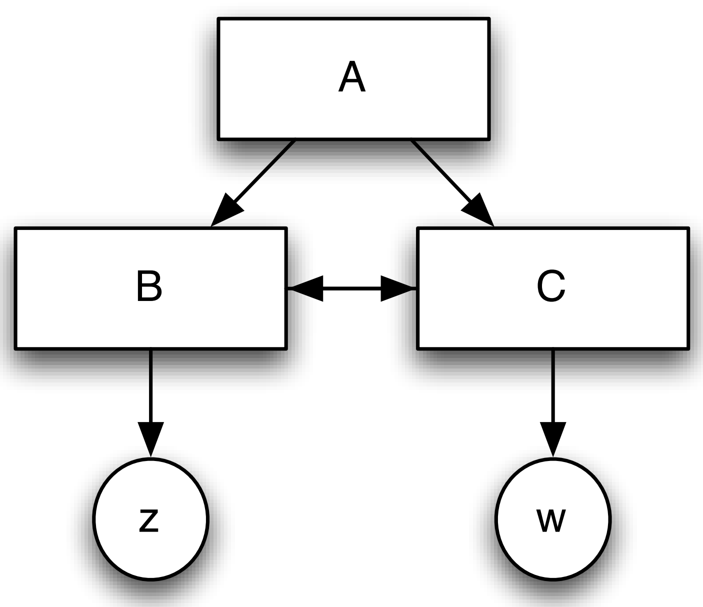
- Assume underlying lattice $(P, \leq)$ has finite height
  - What is height of lattice of monotonic functions?
  - Claim: finite (can you prove this?)

- Therefore, $g(j)$ converges

Non-Reducible Flow Graphs

- Elimination methods usually only applied to *reducible* flow graphs
  - Ones that can be collapsed
  - Standard constructs yield only reducible flow graphs

- Unrestricted goto can yield non-reducible graphs



Comments

- Can also do backwards elimination
  - Not quite as nice (regions are usually single *entry* but often not single *exit*)
- For bit-vector problems, elimination efficient
  - Easy to compose functions, compute meet, etc.
- Elimination originally seemed like it might be faster than iteration
  - Not really the case