CMSC 430
Introduction to Compilers
Fall 2016

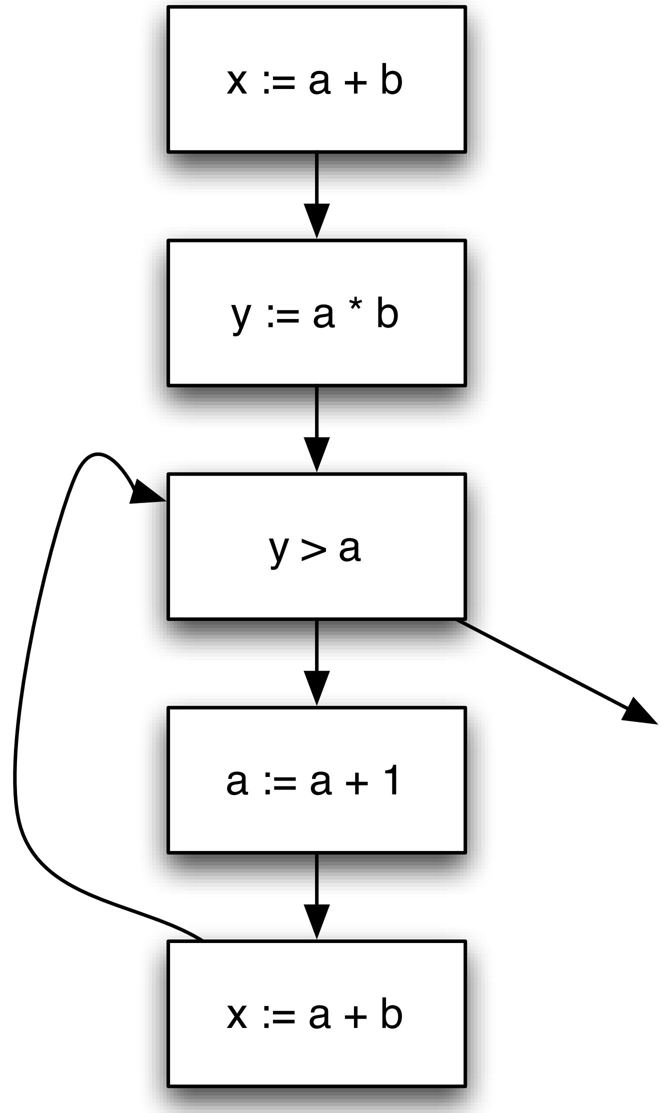
Data Flow Analysis

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
- Operates on control-flow graphs, typically

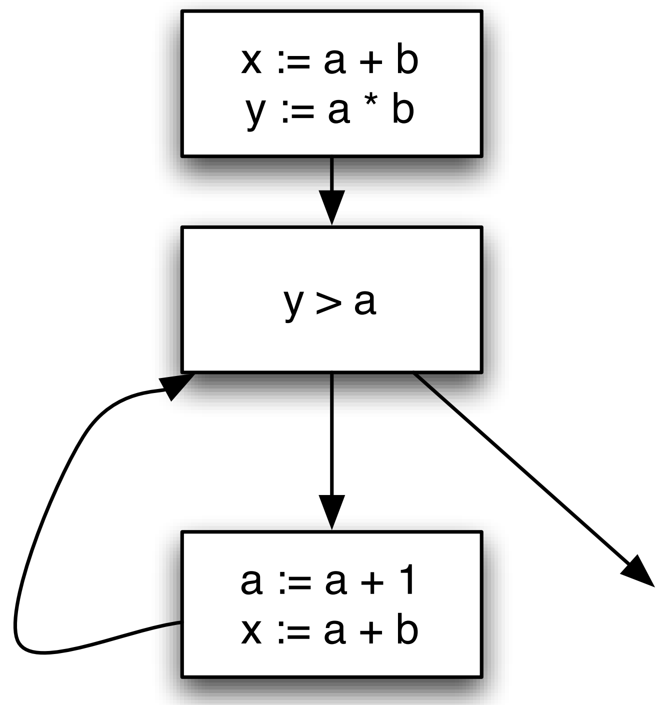
Control-Flow Graph Example

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



Control-Flow Graph w/Basic Blocks

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

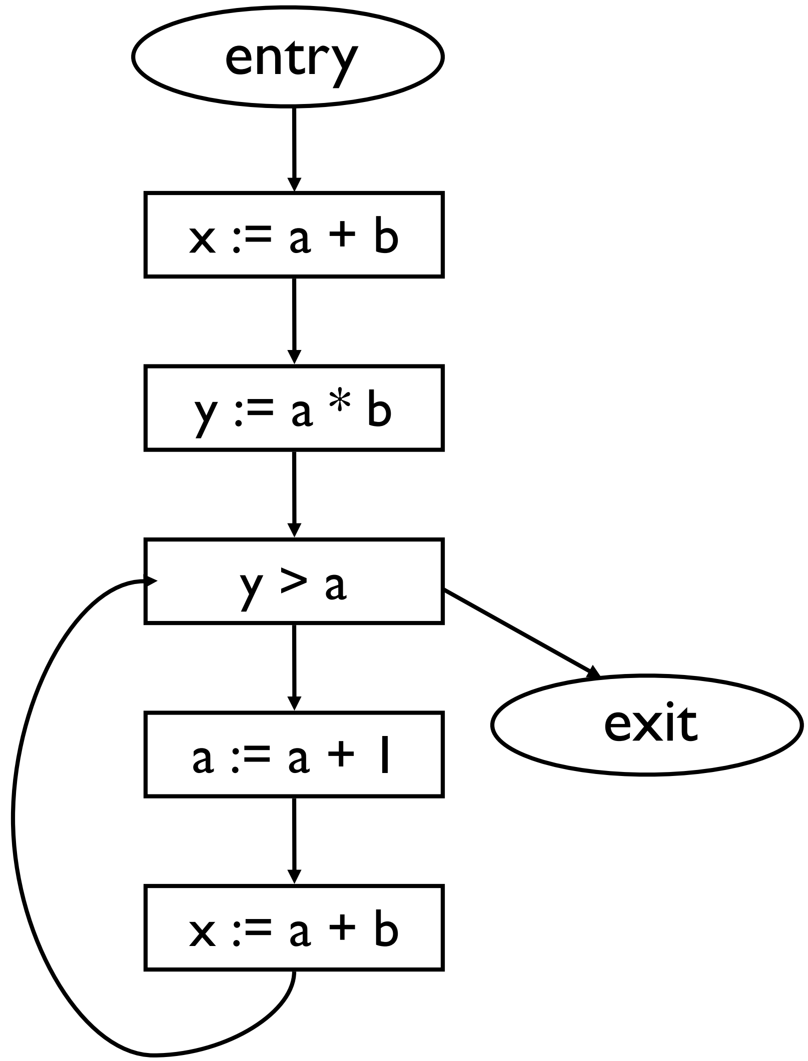


- Can lead to more efficient implementations
- But more complicated to explain, so...
  - We'll use single-statement blocks in lecture today

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



Notes on Entry and Exit

- Typically, we perform data flow analysis on a function body
- Functions usually have
  - A unique entry point
  - Multiple exit points
- So in practice, there can be multiple exit nodes in the CFG
  - For the rest of these slides, we'll assume there's only one
  - In practice, just treat all exit nodes the same way as if there's only one exit node

Available Expressions

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

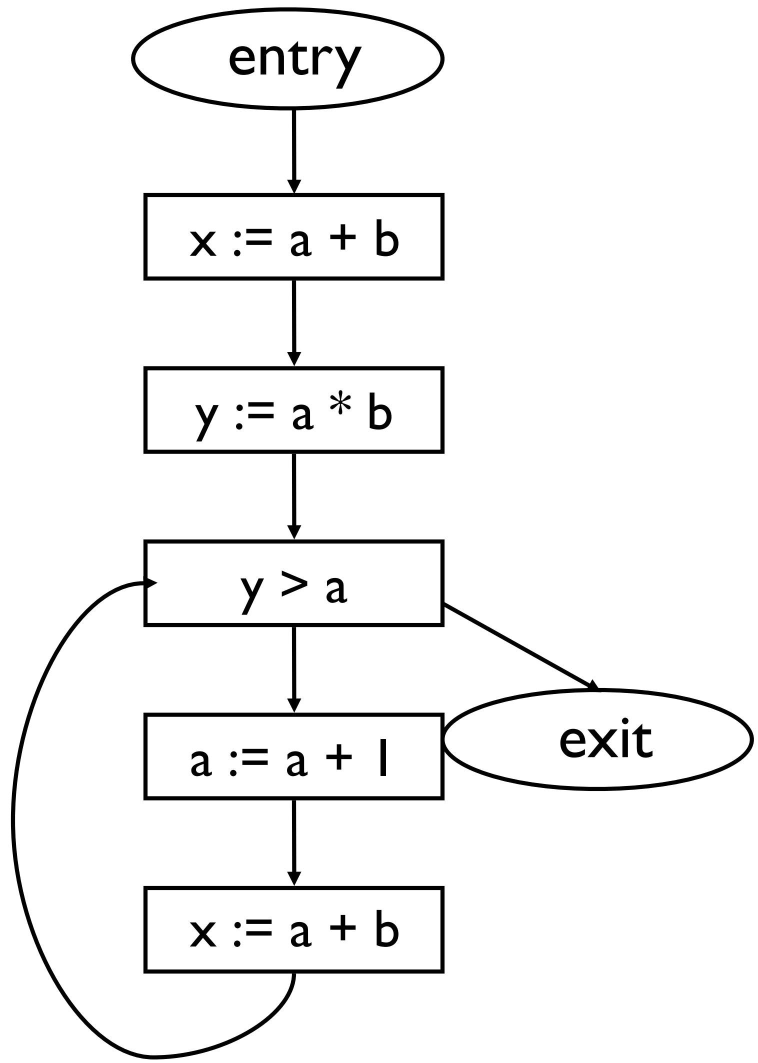
Available Expressions

- 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

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

Data Flow Facts

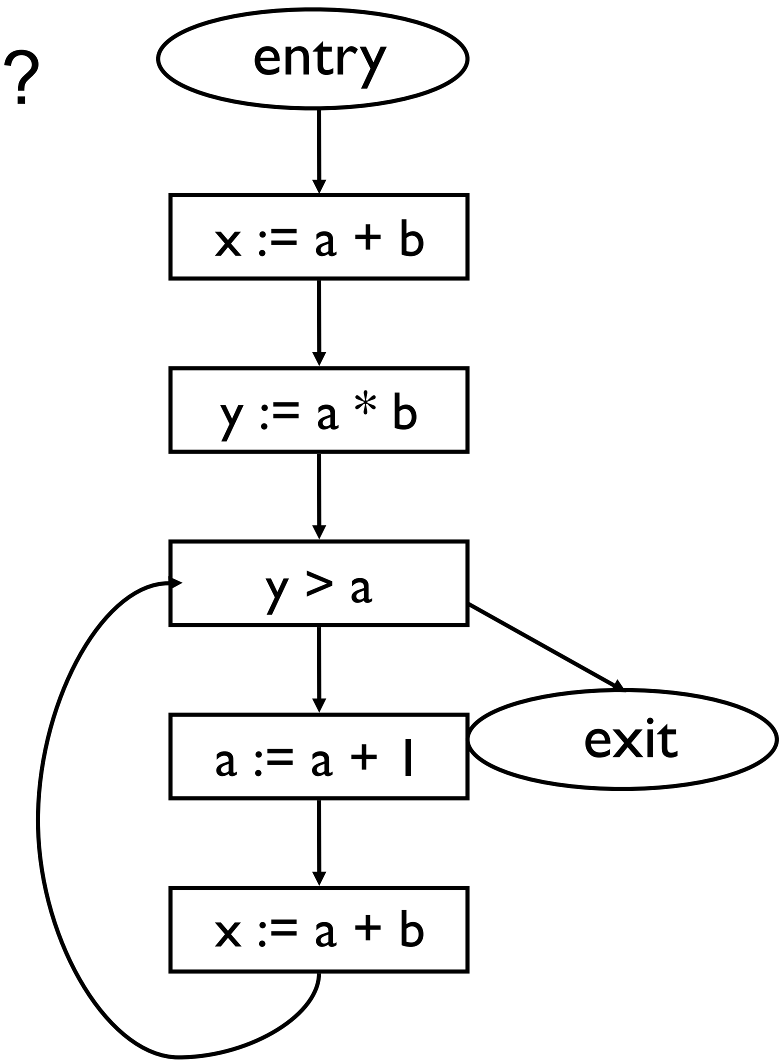
- Is expression e available?
- Facts:
  - a + b is available
  - a * b is available
  - a + 1 is available



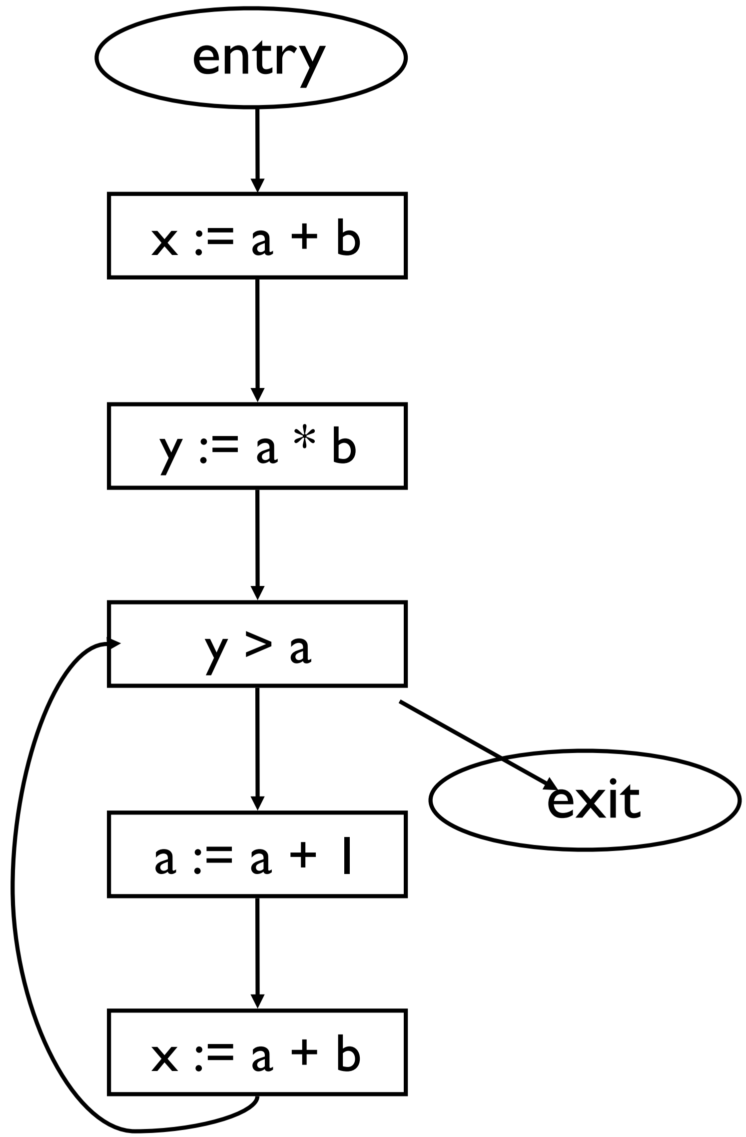
Gen and Kill

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

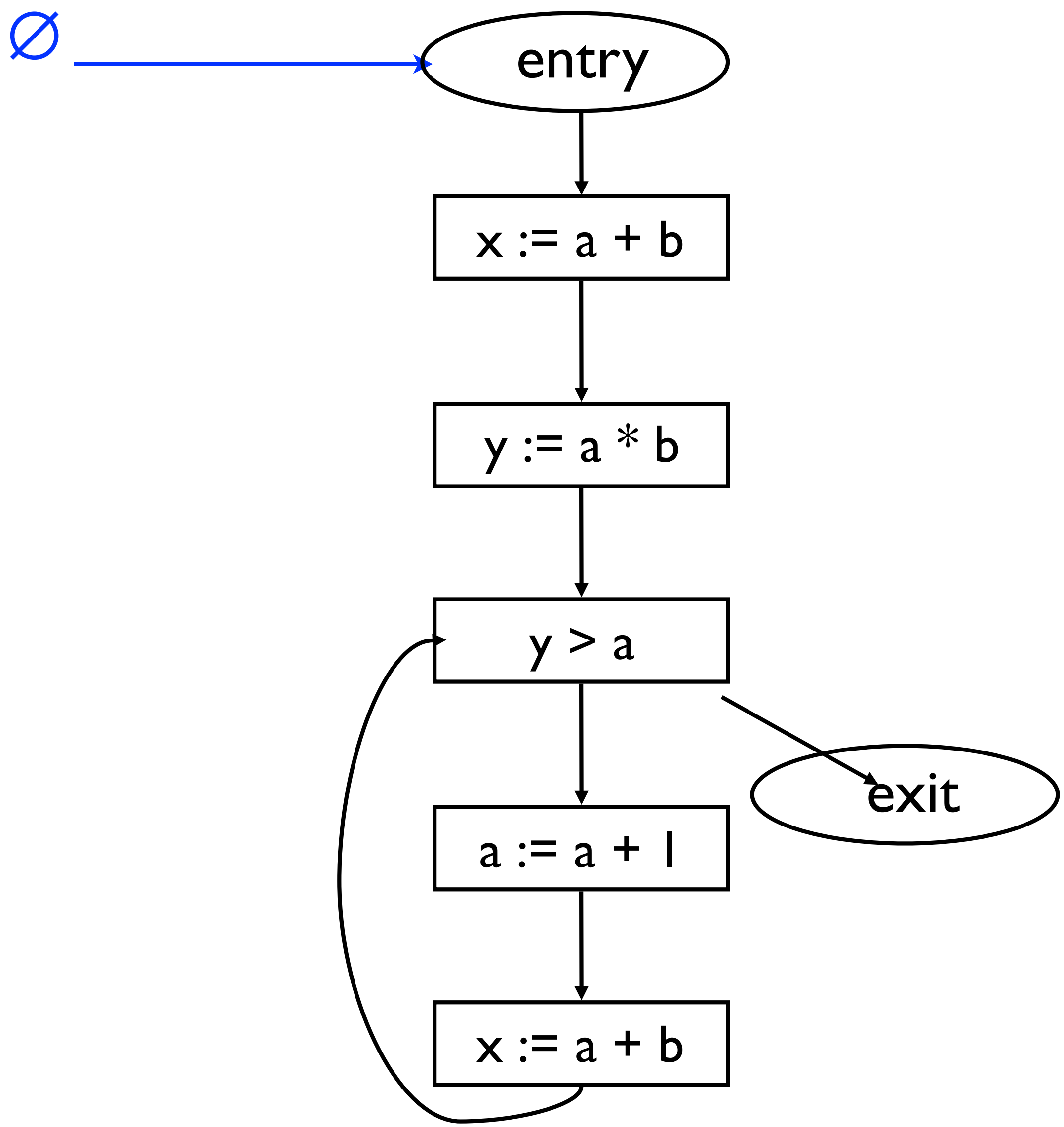
Stmt	Gen	Kill
x := a + b	a + b	
y := a * b	a * b	
a := a + 1		a + 1, a + b, a * b



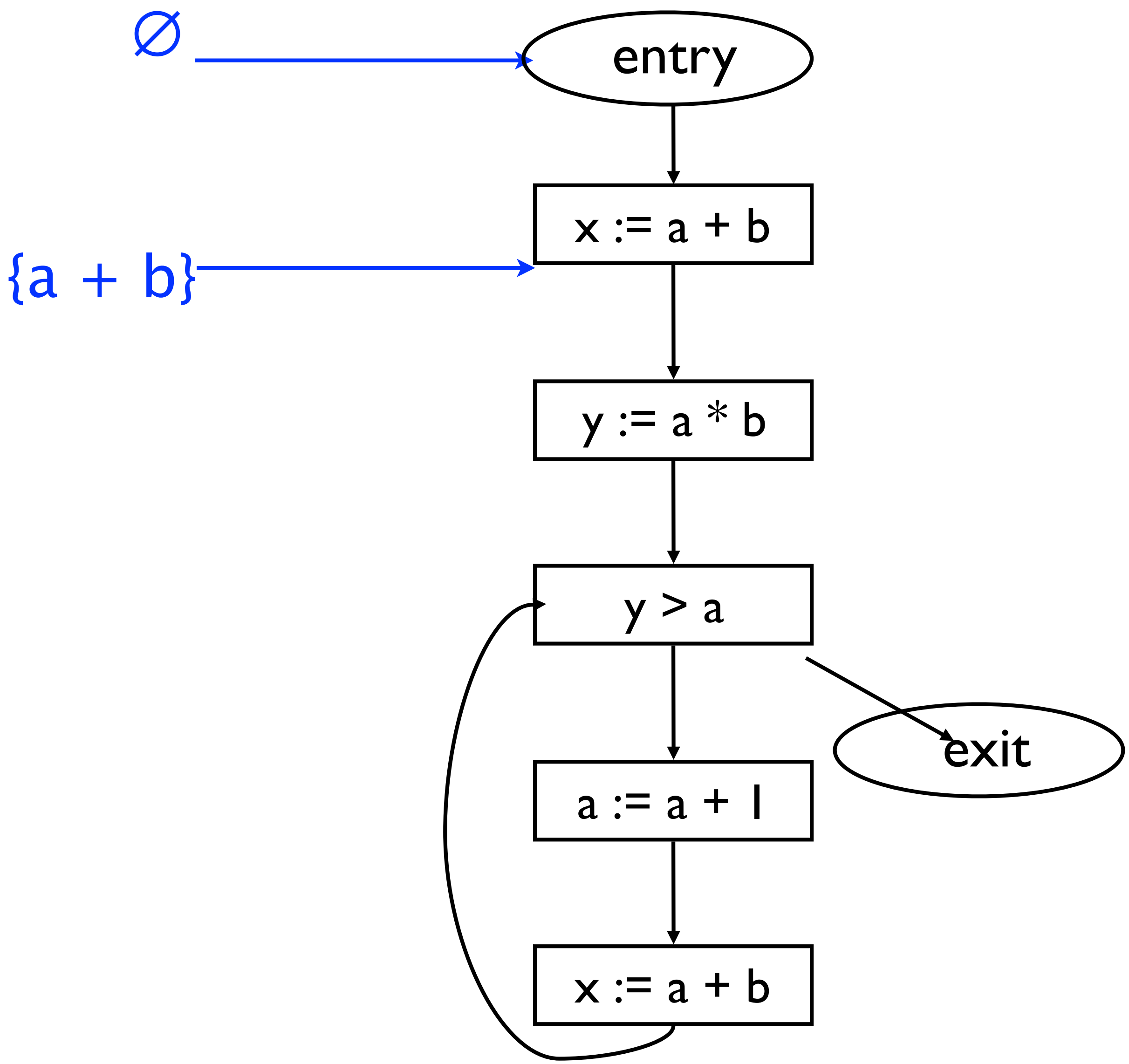
Computing Available Expressions



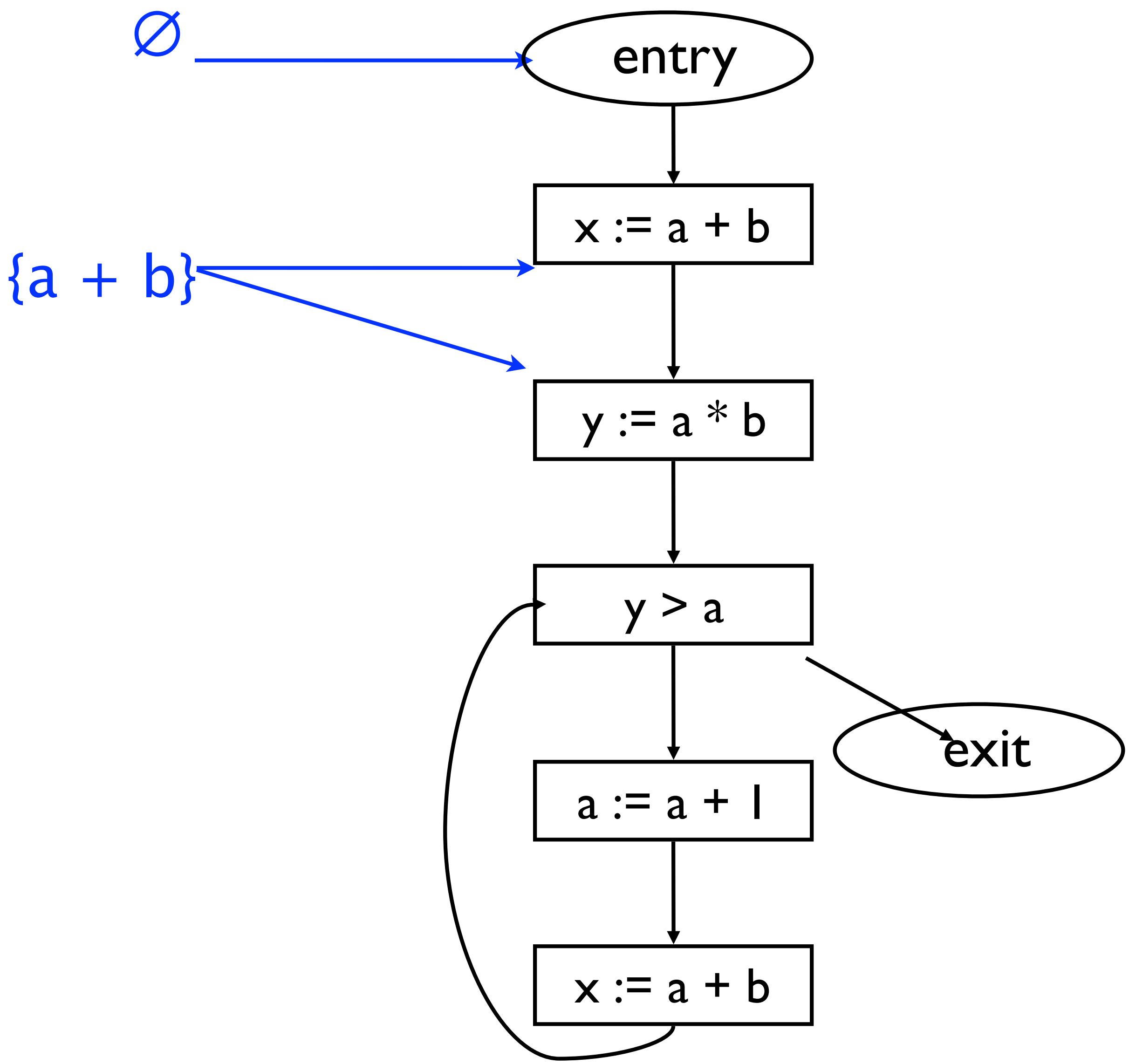
Computing Available Expressions



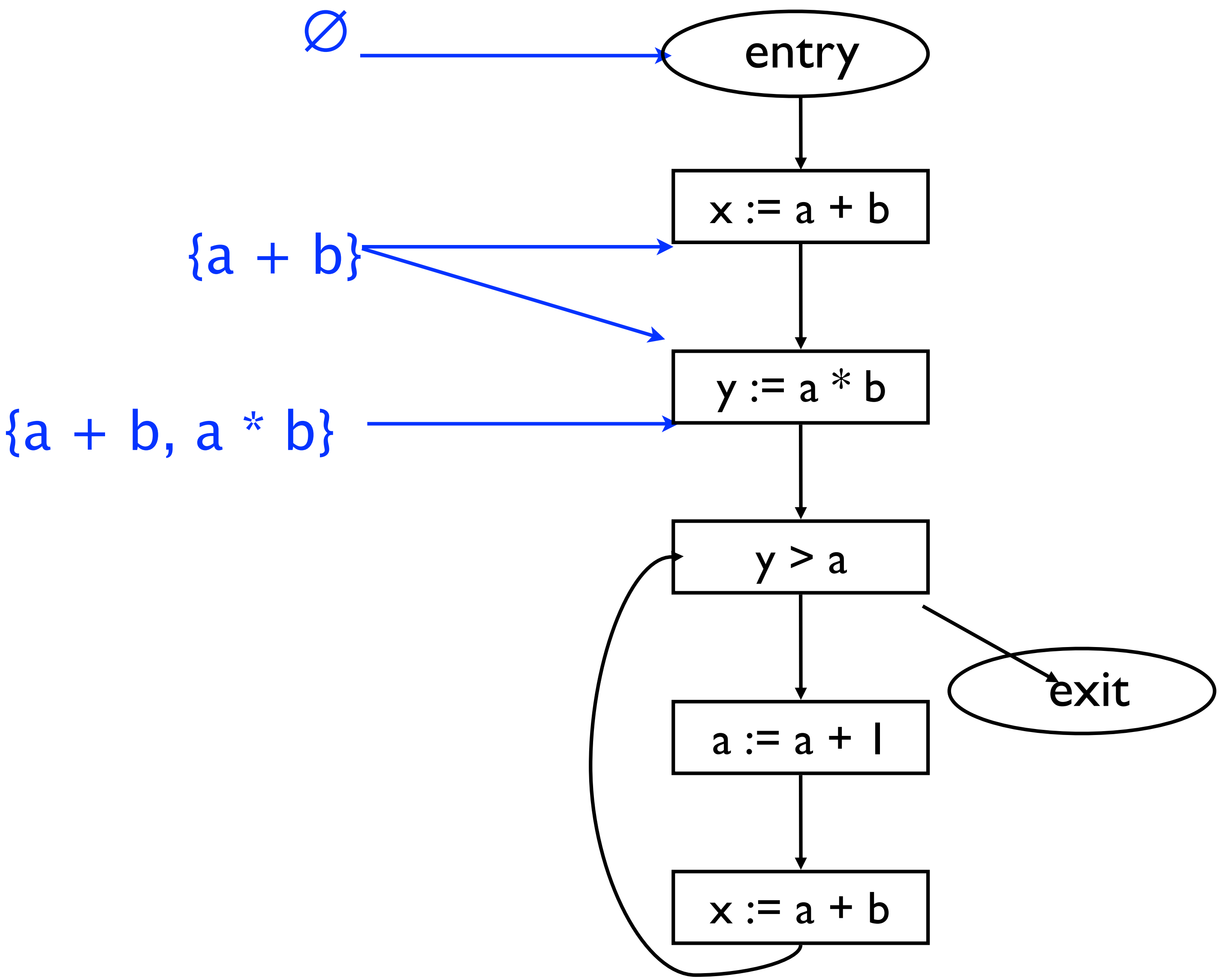
Computing Available Expressions



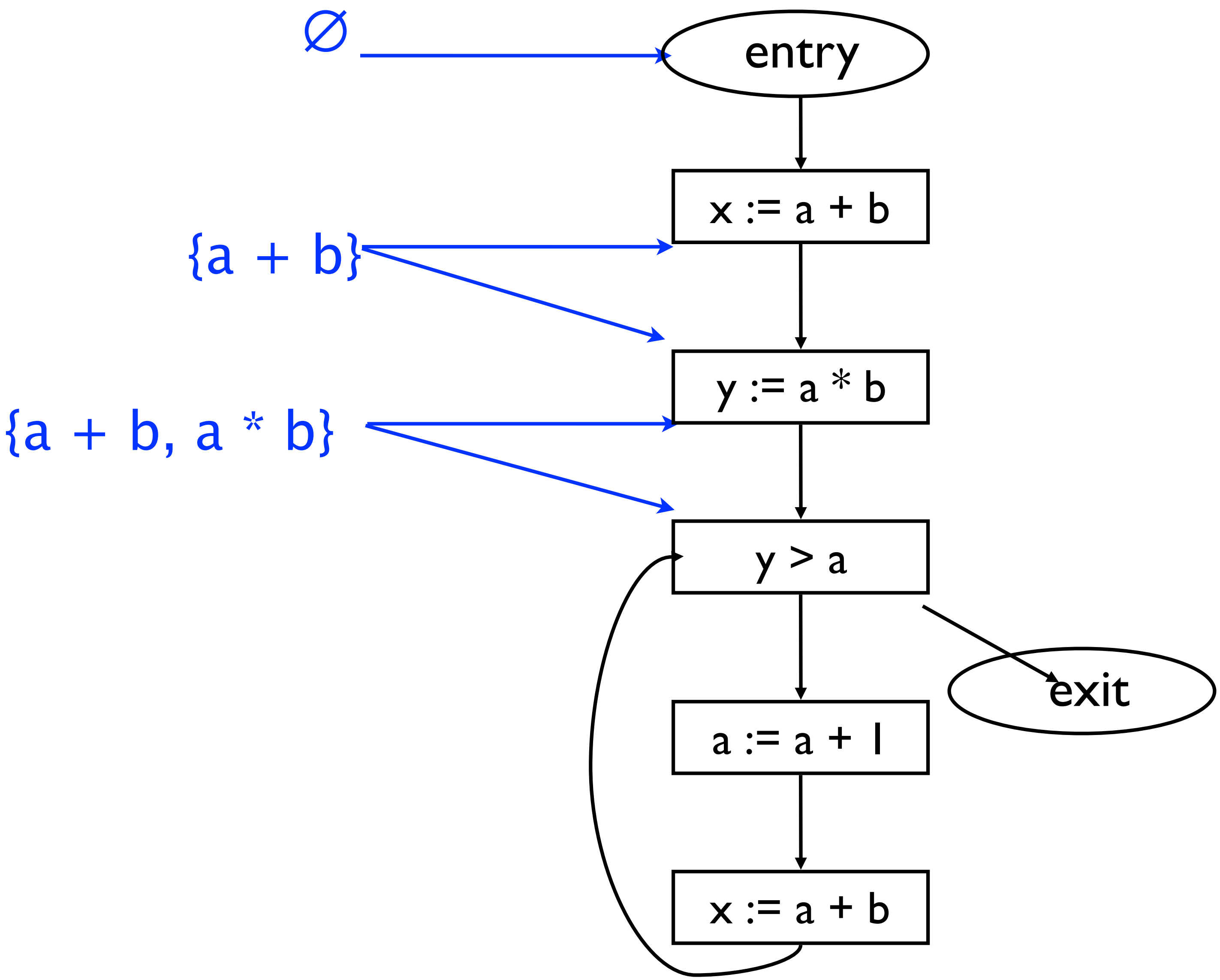
Computing Available Expressions



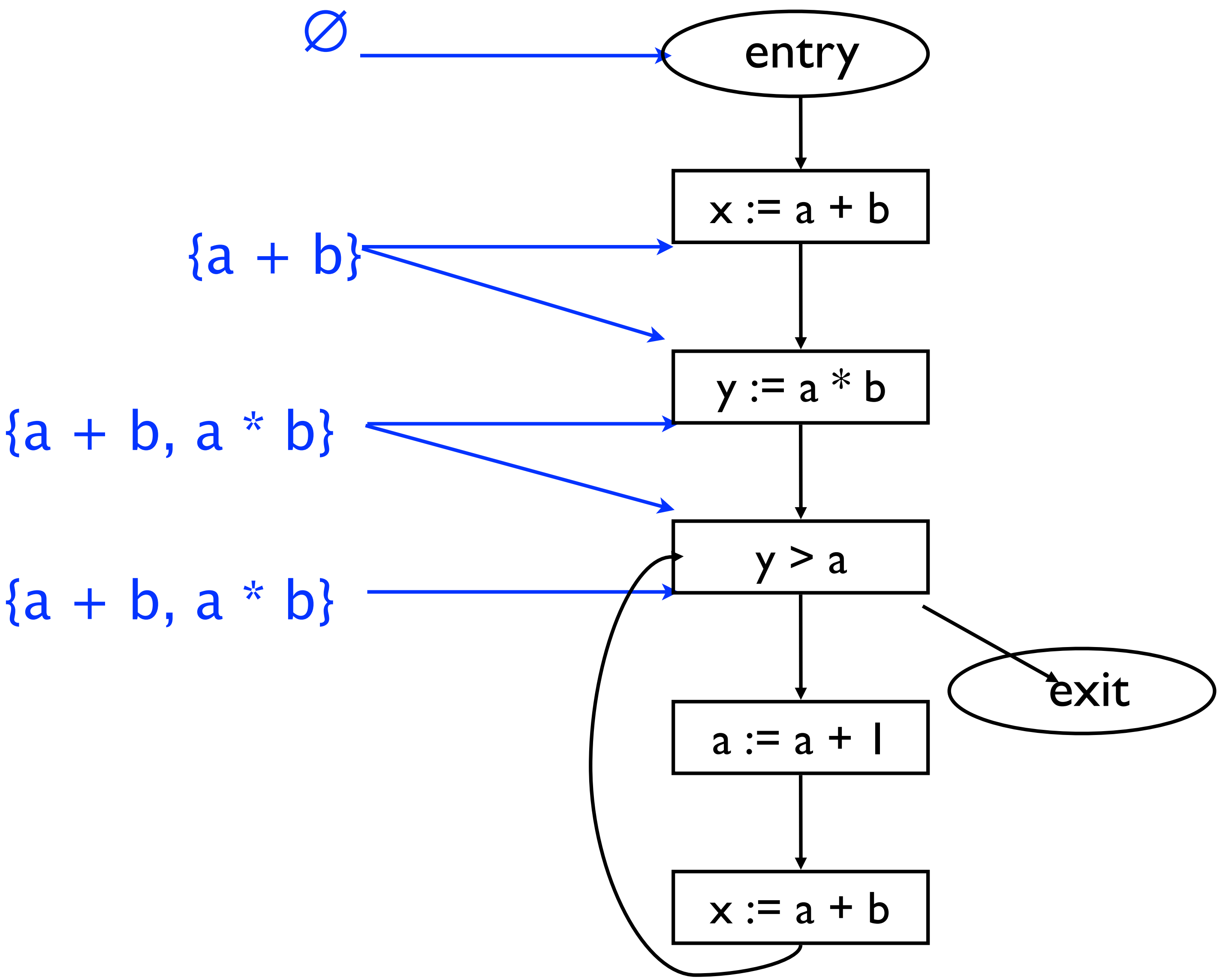
Computing Available Expressions



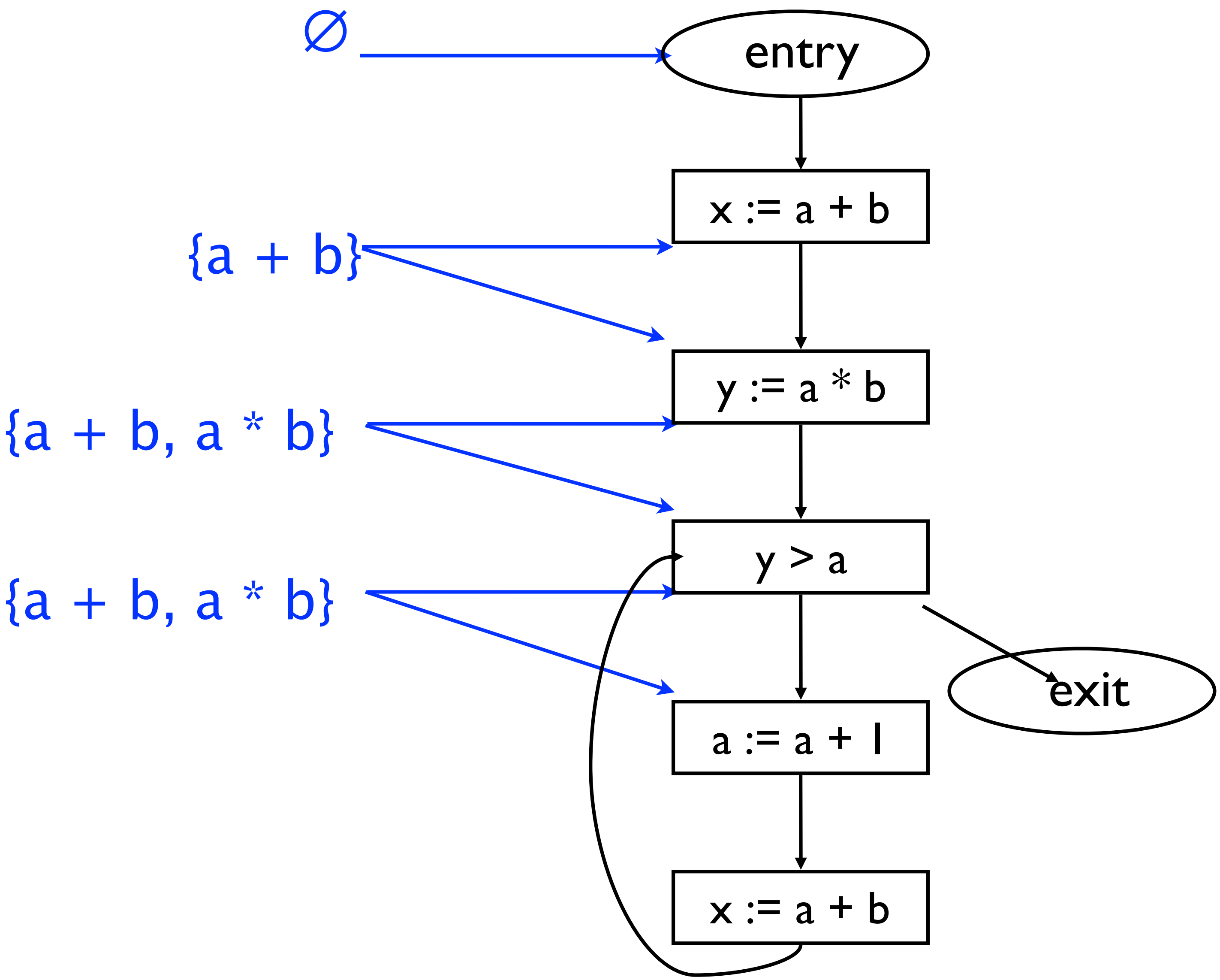
Computing Available Expressions



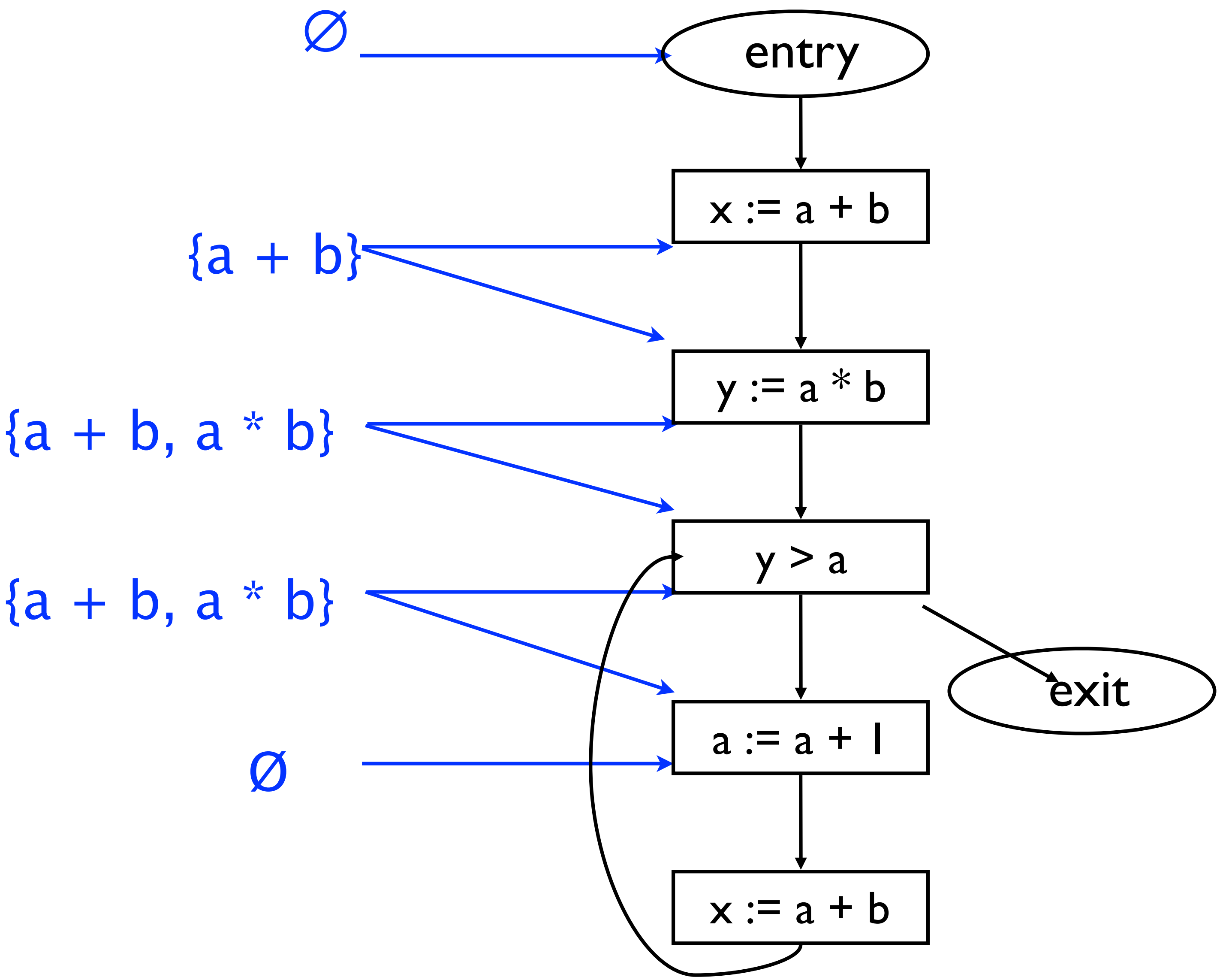
Computing Available Expressions



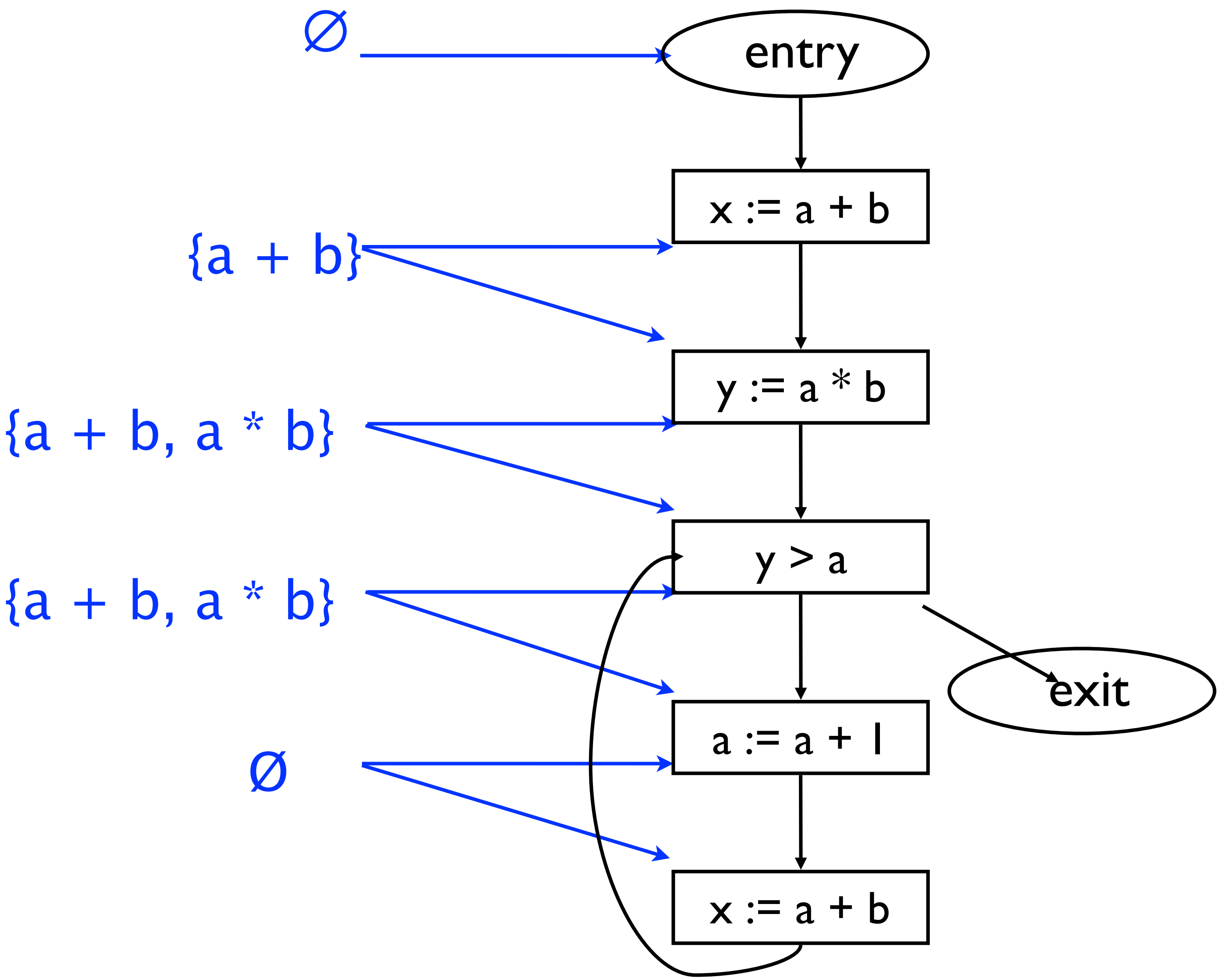
Computing Available Expressions



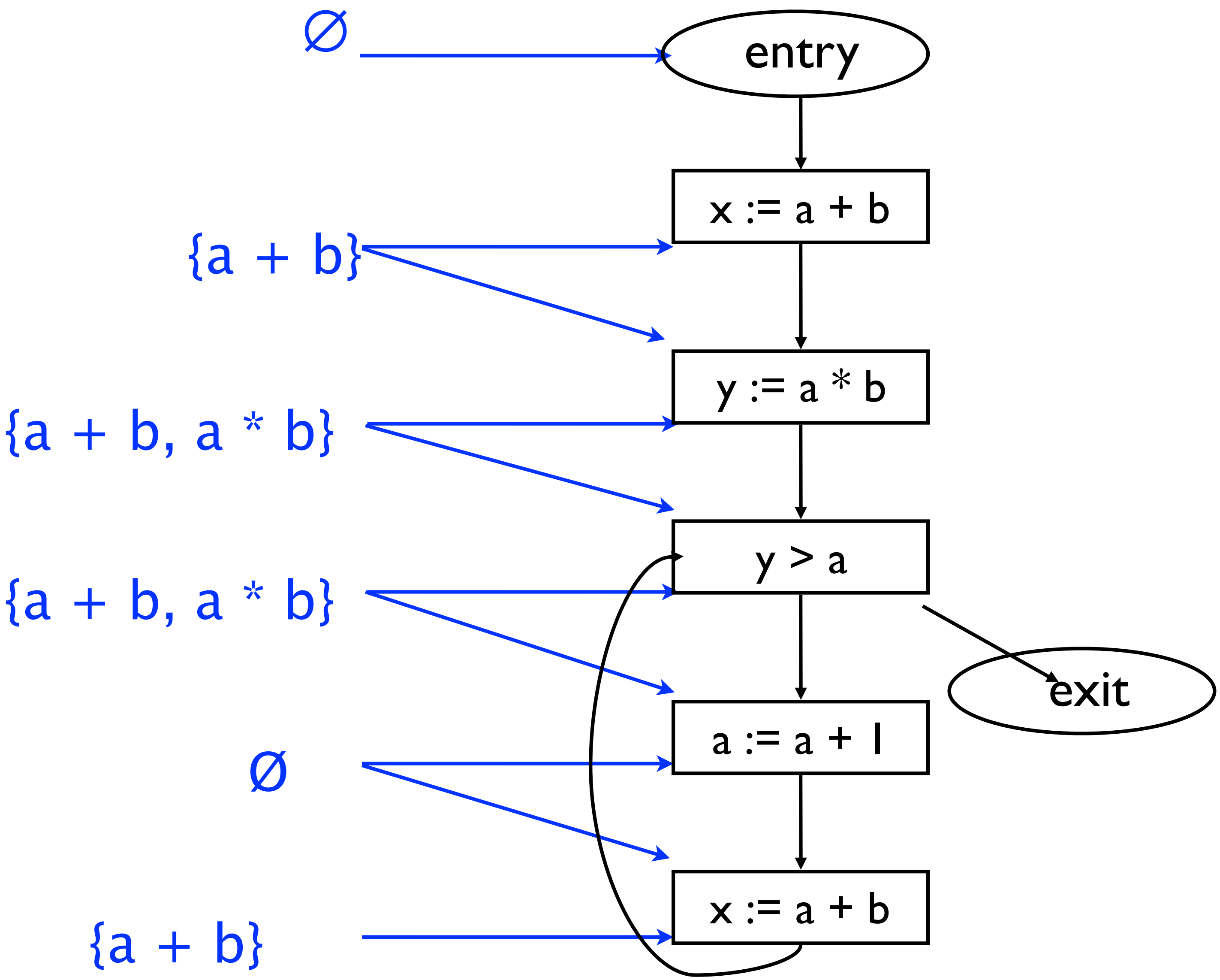
Computing Available Expressions



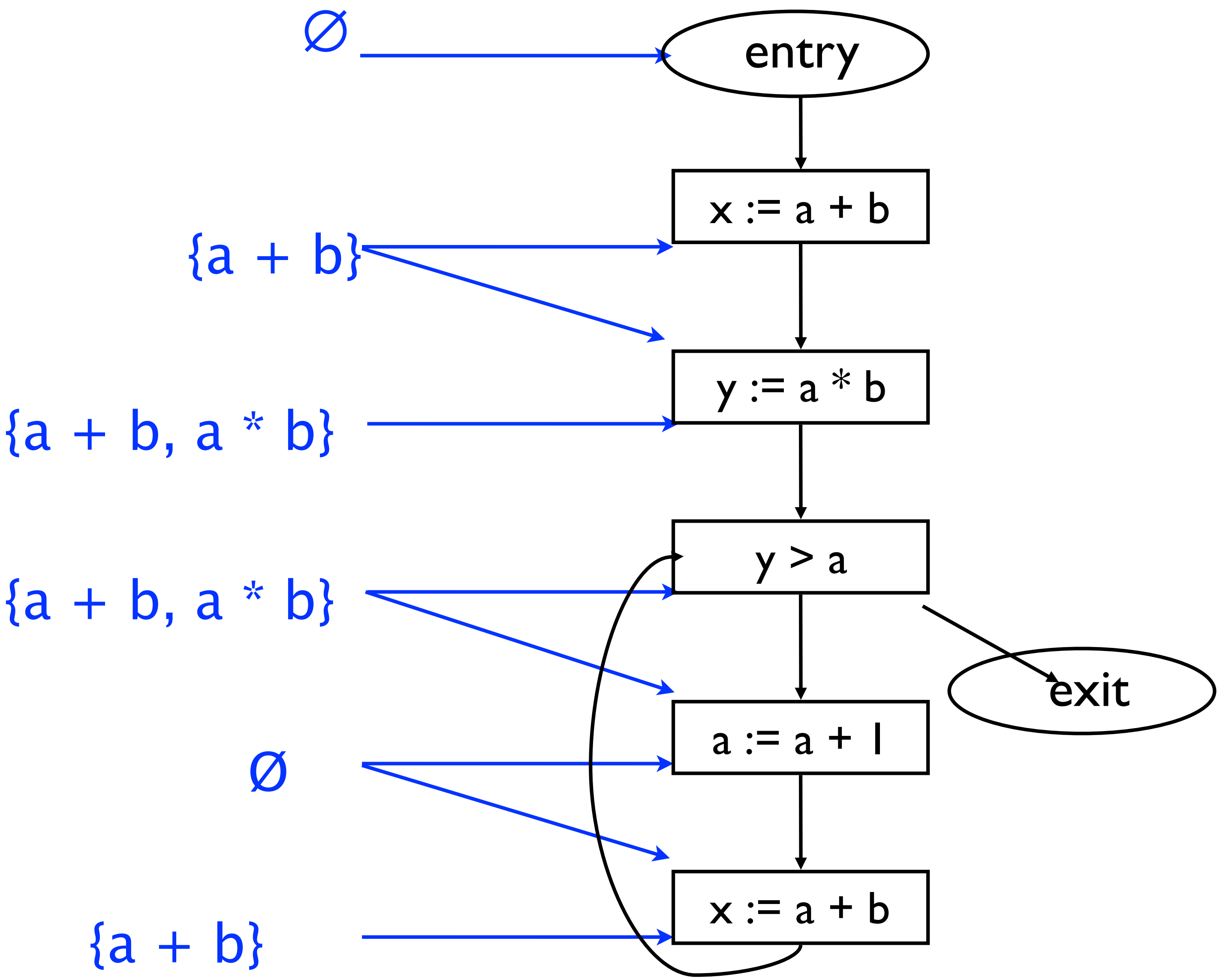
Computing Available Expressions



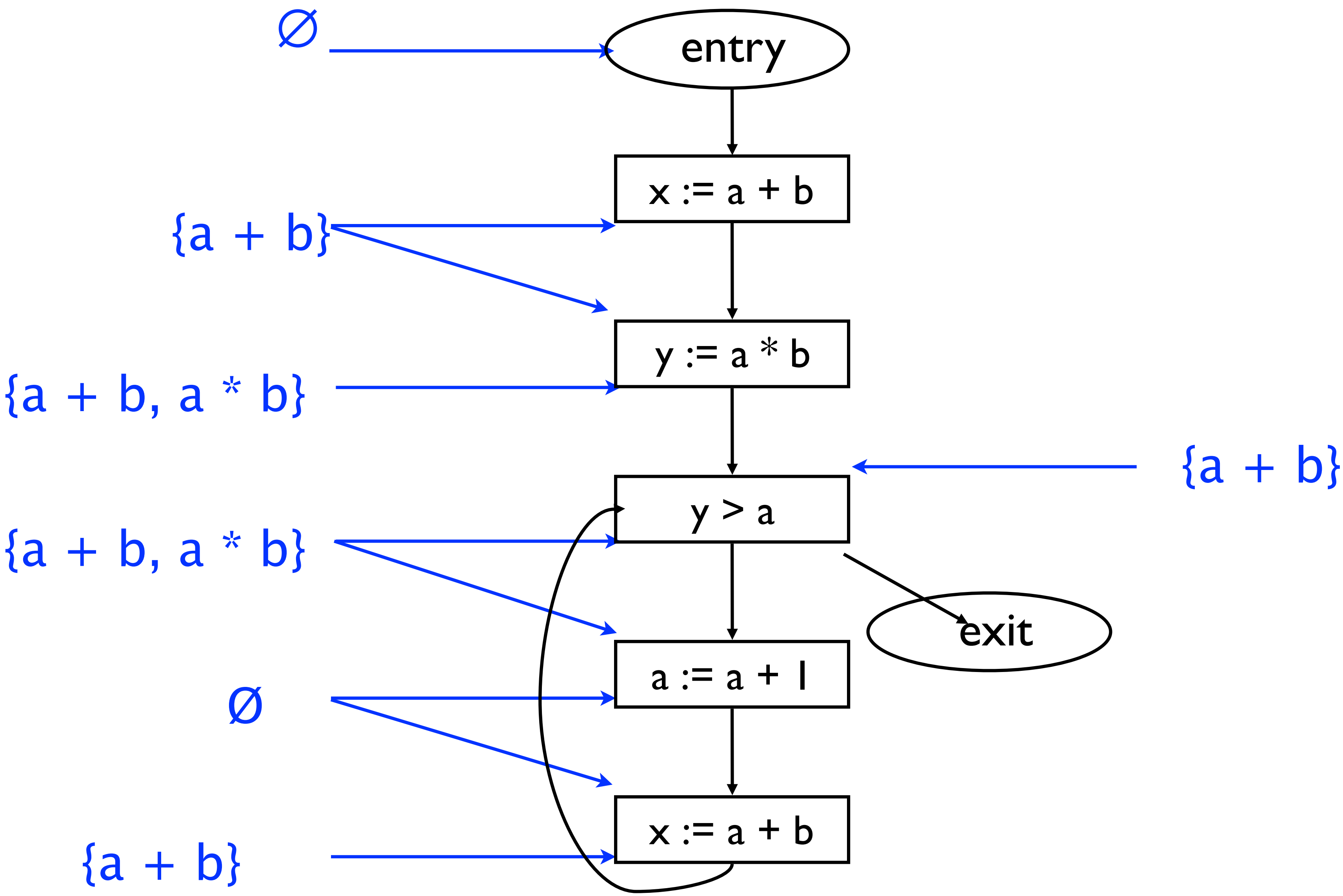
Computing Available Expressions



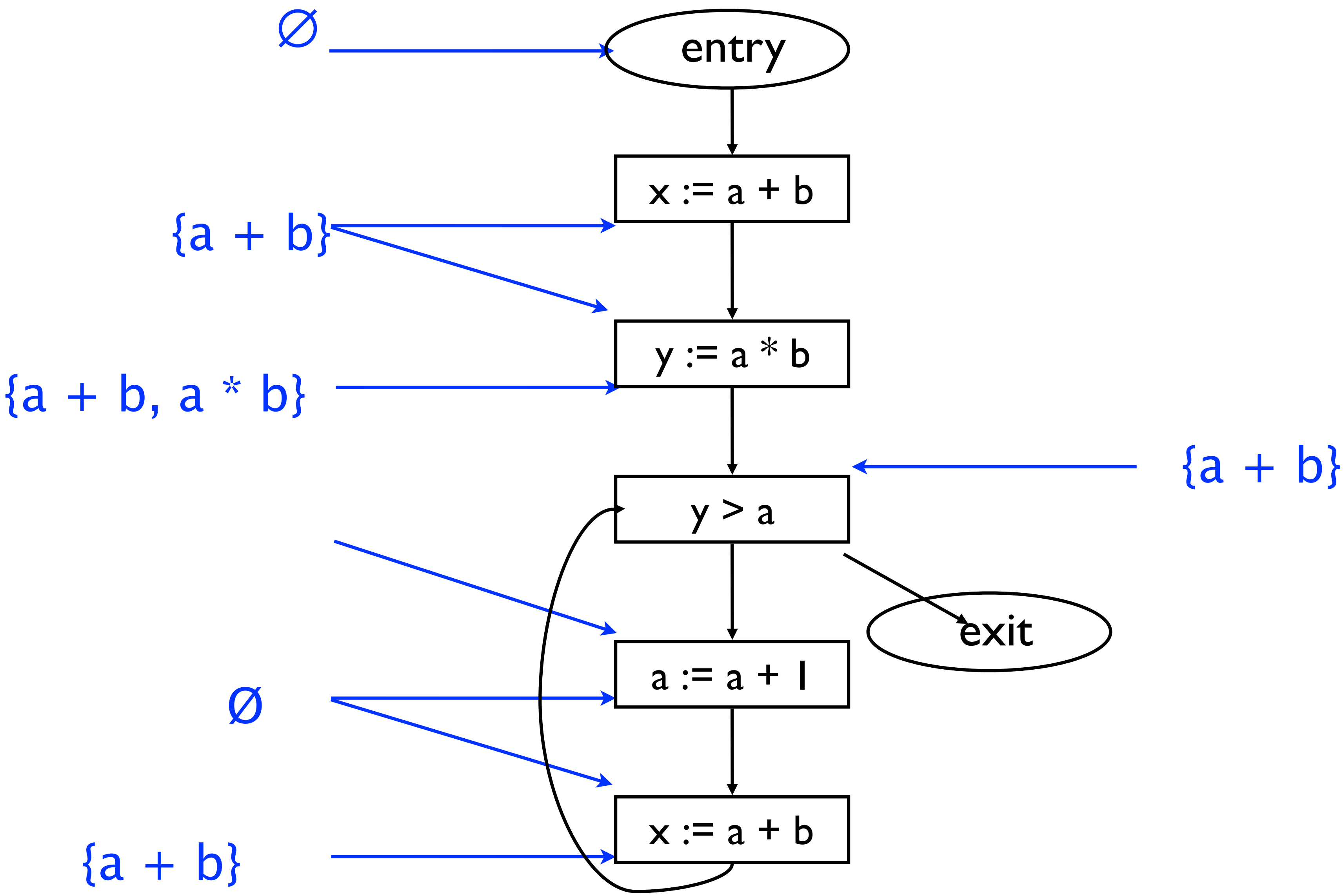
Computing Available Expressions



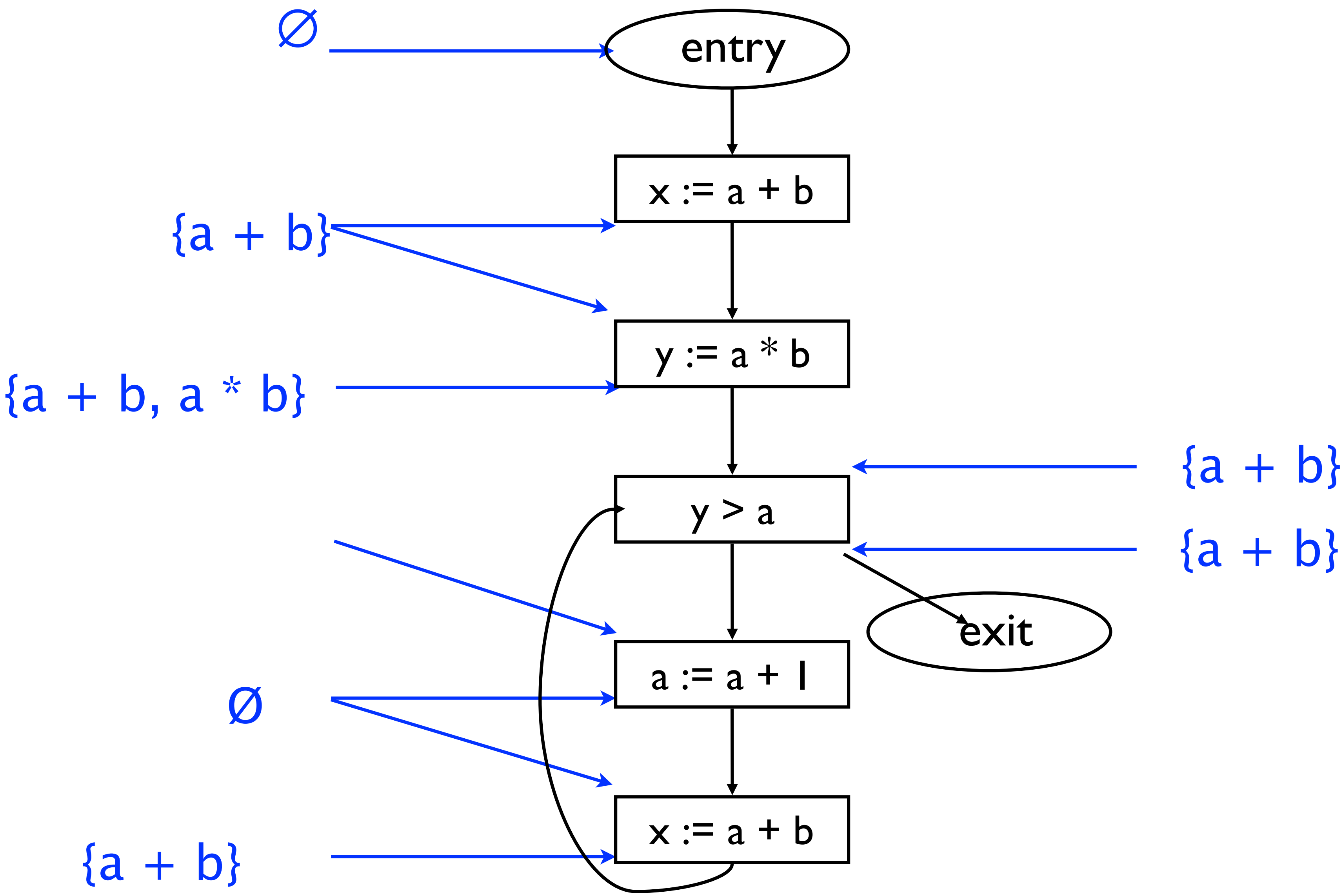
Computing Available Expressions



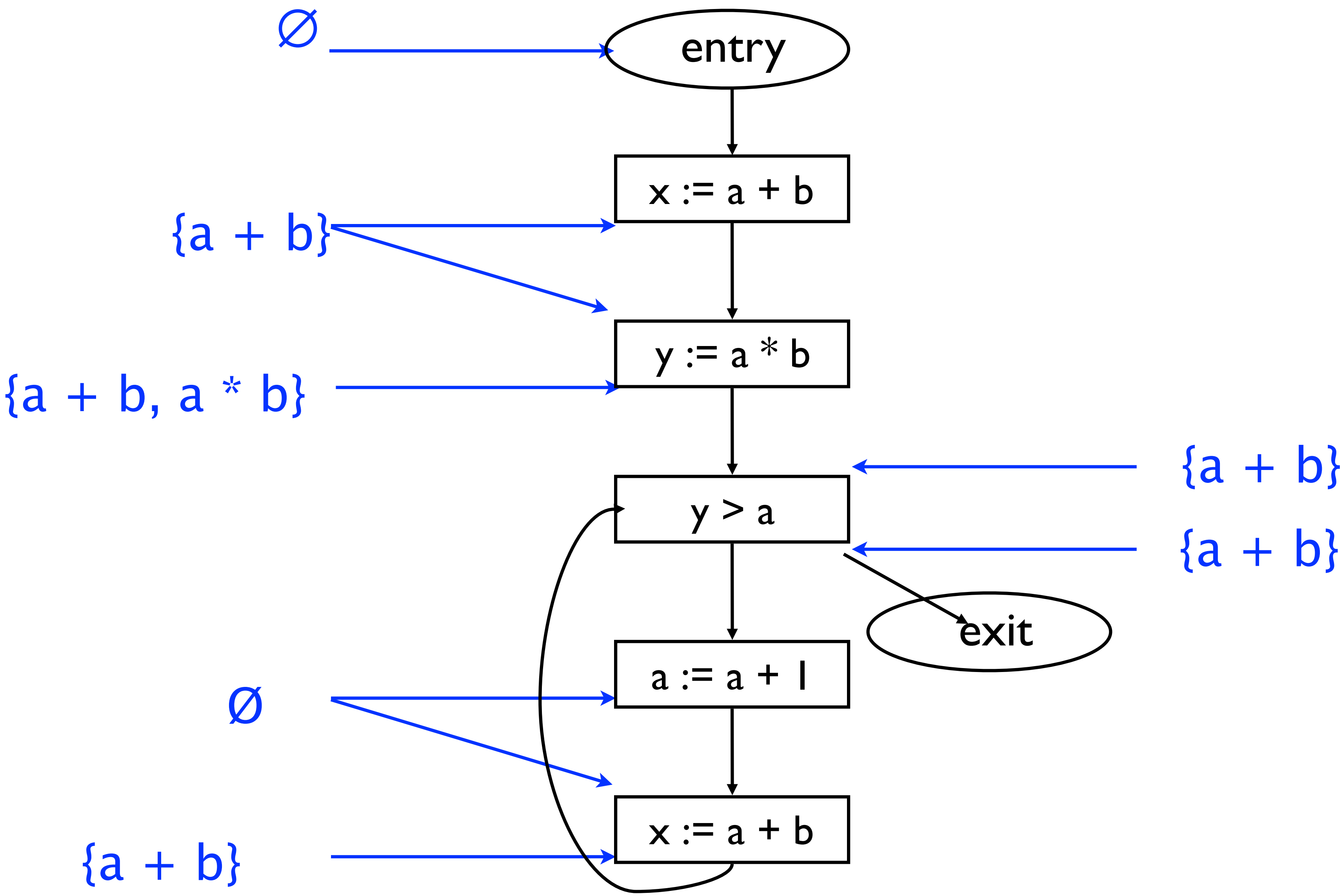
Computing Available Expressions



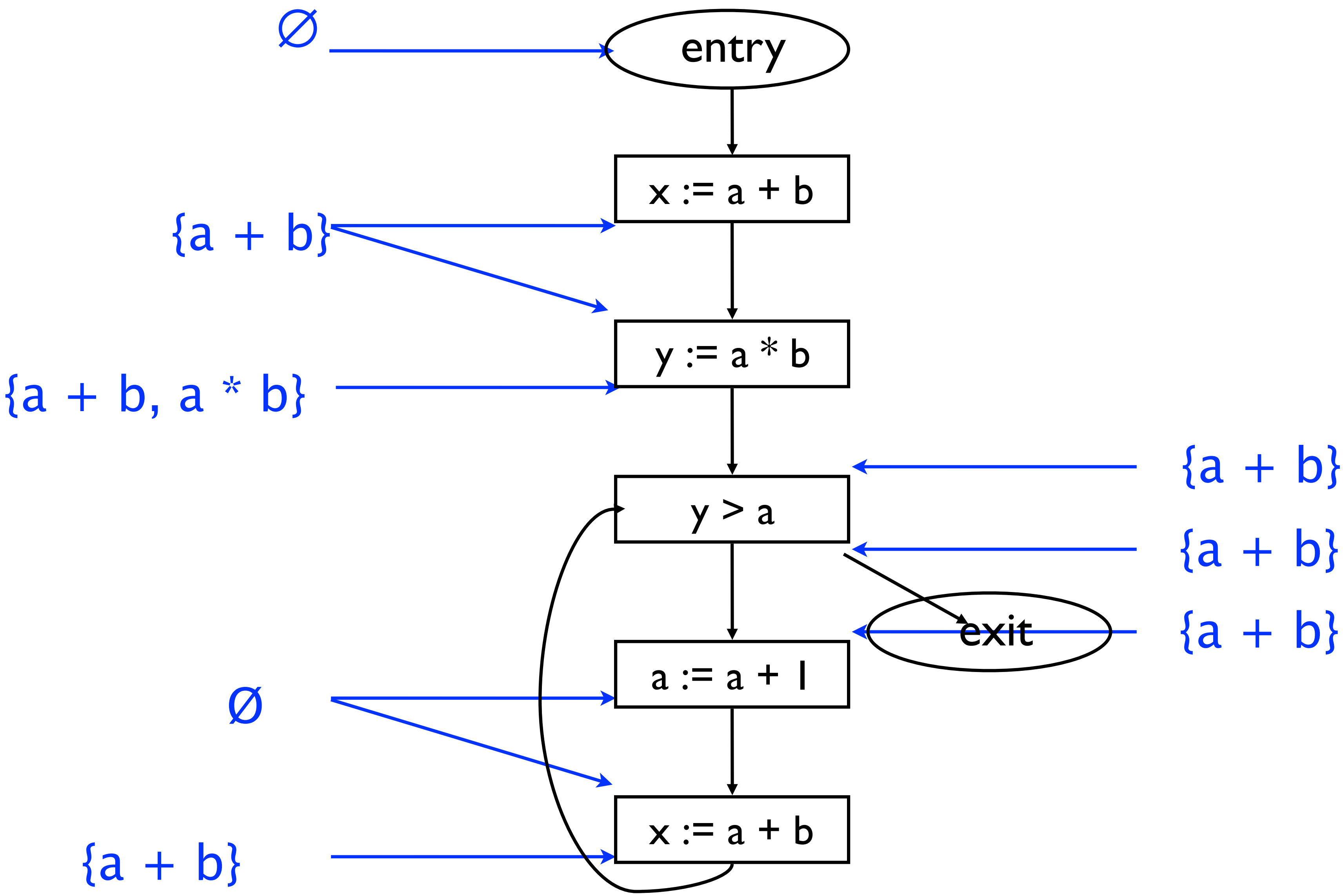
Computing Available Expressions



Computing Available Expressions



Computing Available Expressions



Terminology

- A *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}
  - $in(s)$ = program point just before executing s
  - $out(s)$ = program point just after executing s

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

- $out(s) = gen(s) \cup (in(s) - kill(s))$
  - Note: These are also called *transfer functions*

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

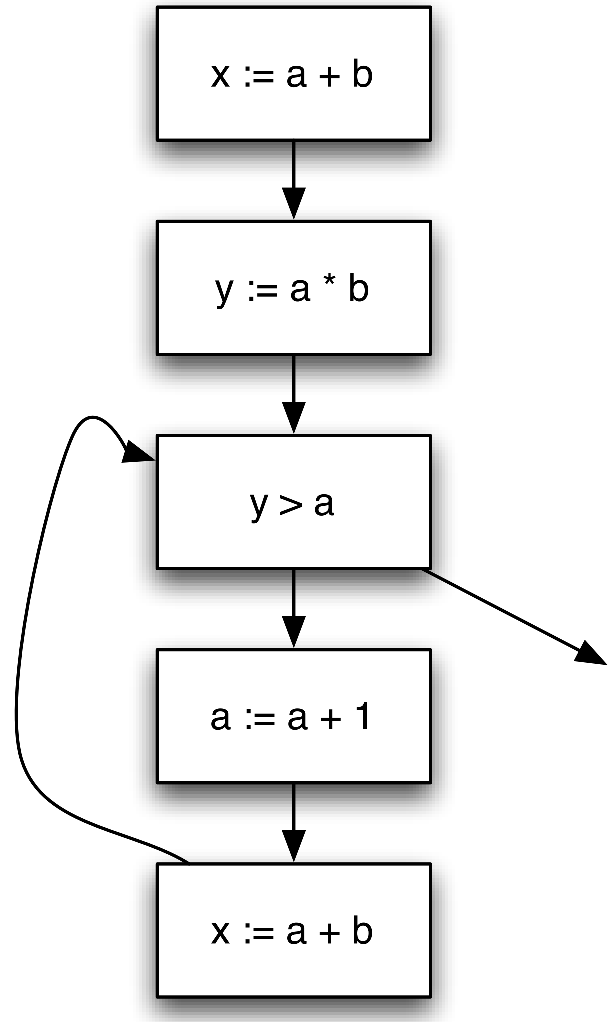
Data Flow Equations

- 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
- $out(s) = \bigcup_{s' \in succ(s)} in(s')$
- $in(s) = gen(s) \cup (out(s) - kill(s))$

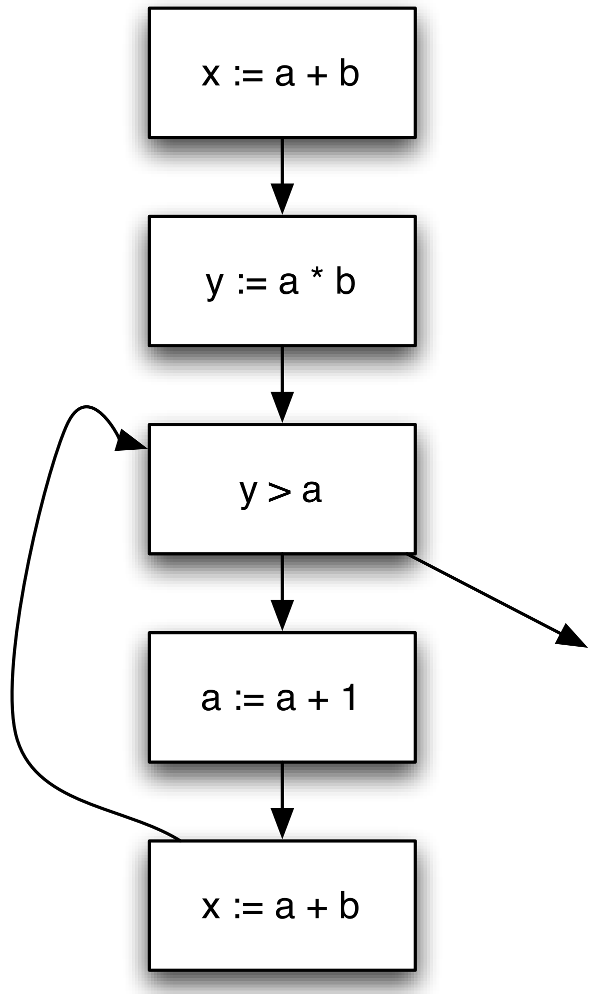
Gen and Kill

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

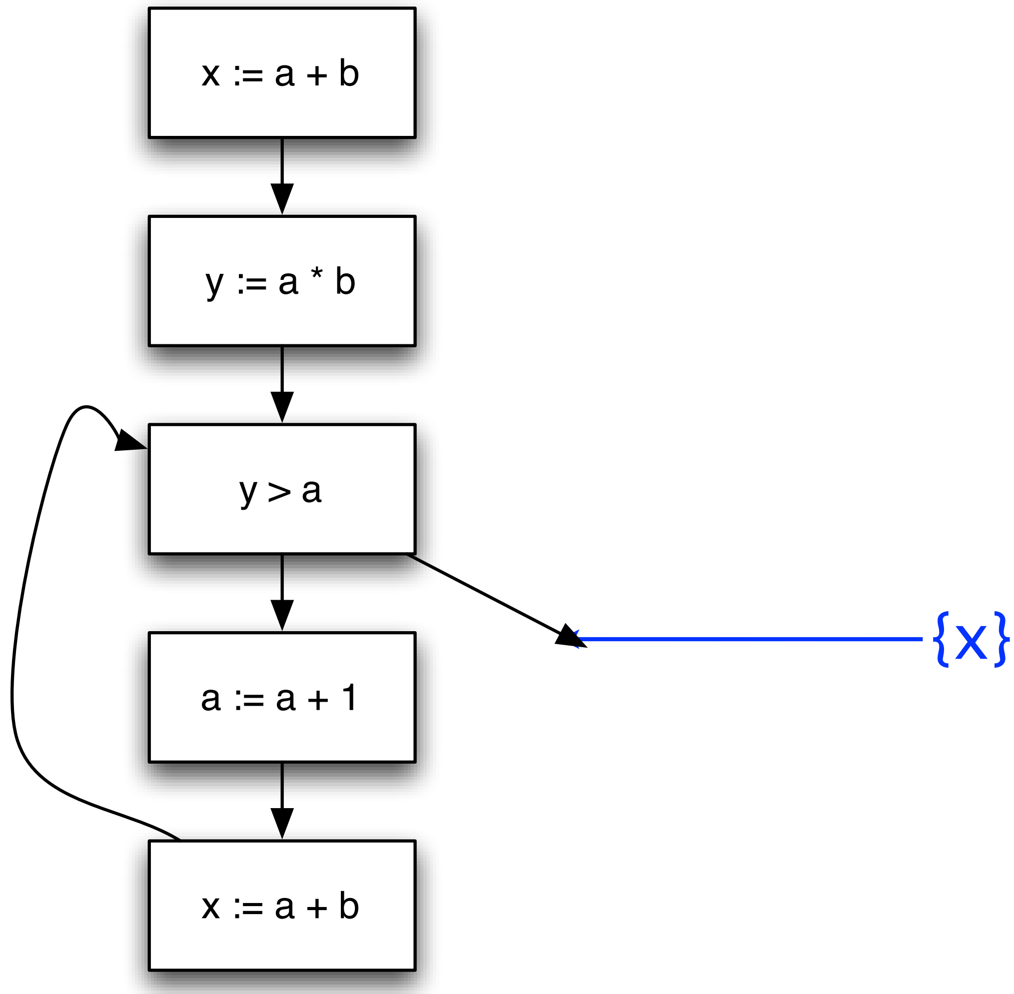
Stmt	Gen	Kill
x := a + b	a, b	x
y := a * b	a, b	y
y > a	a, y	
a := a + 1	a	a



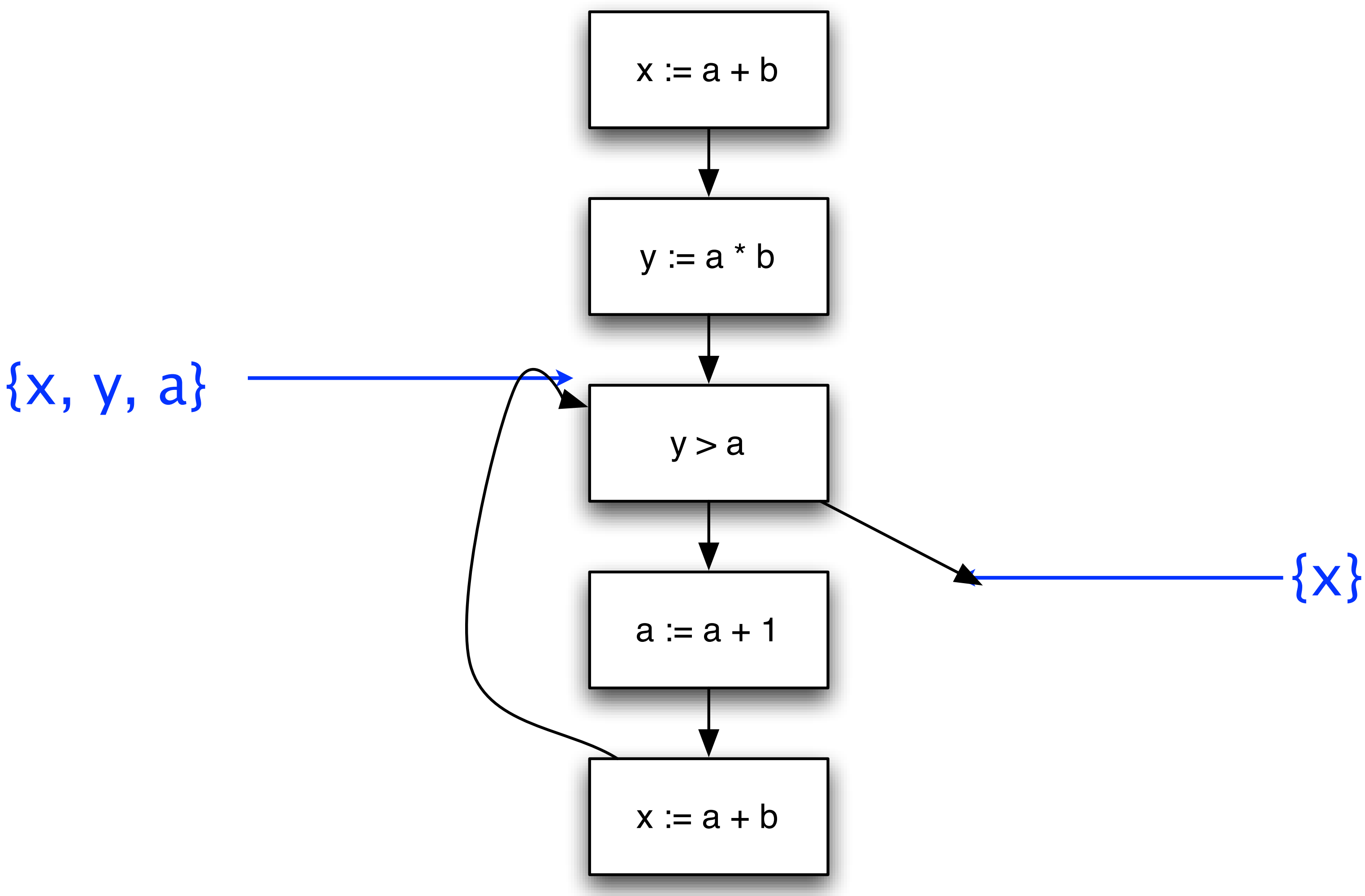
Computing Live Variables



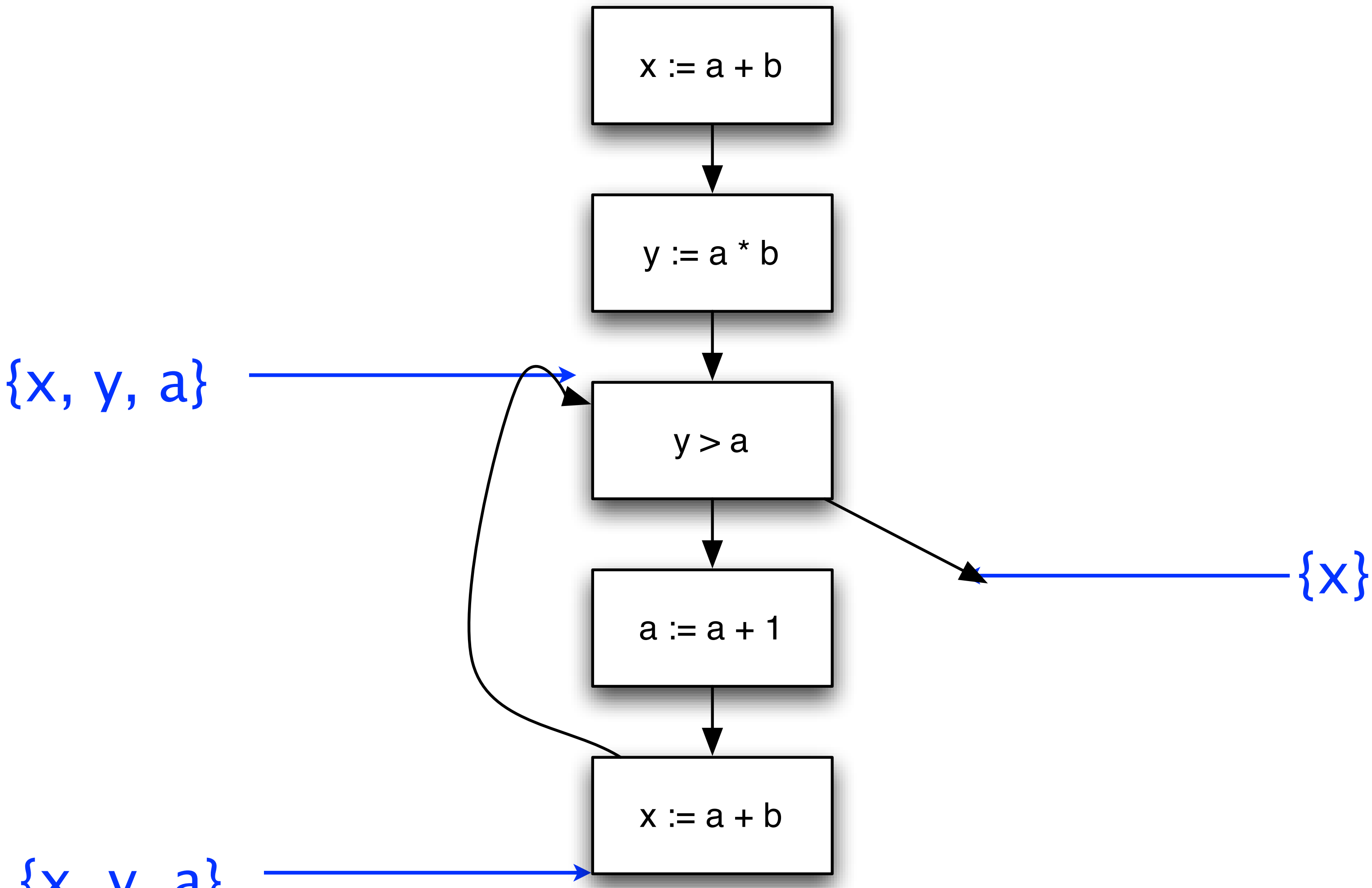
Computing Live Variables



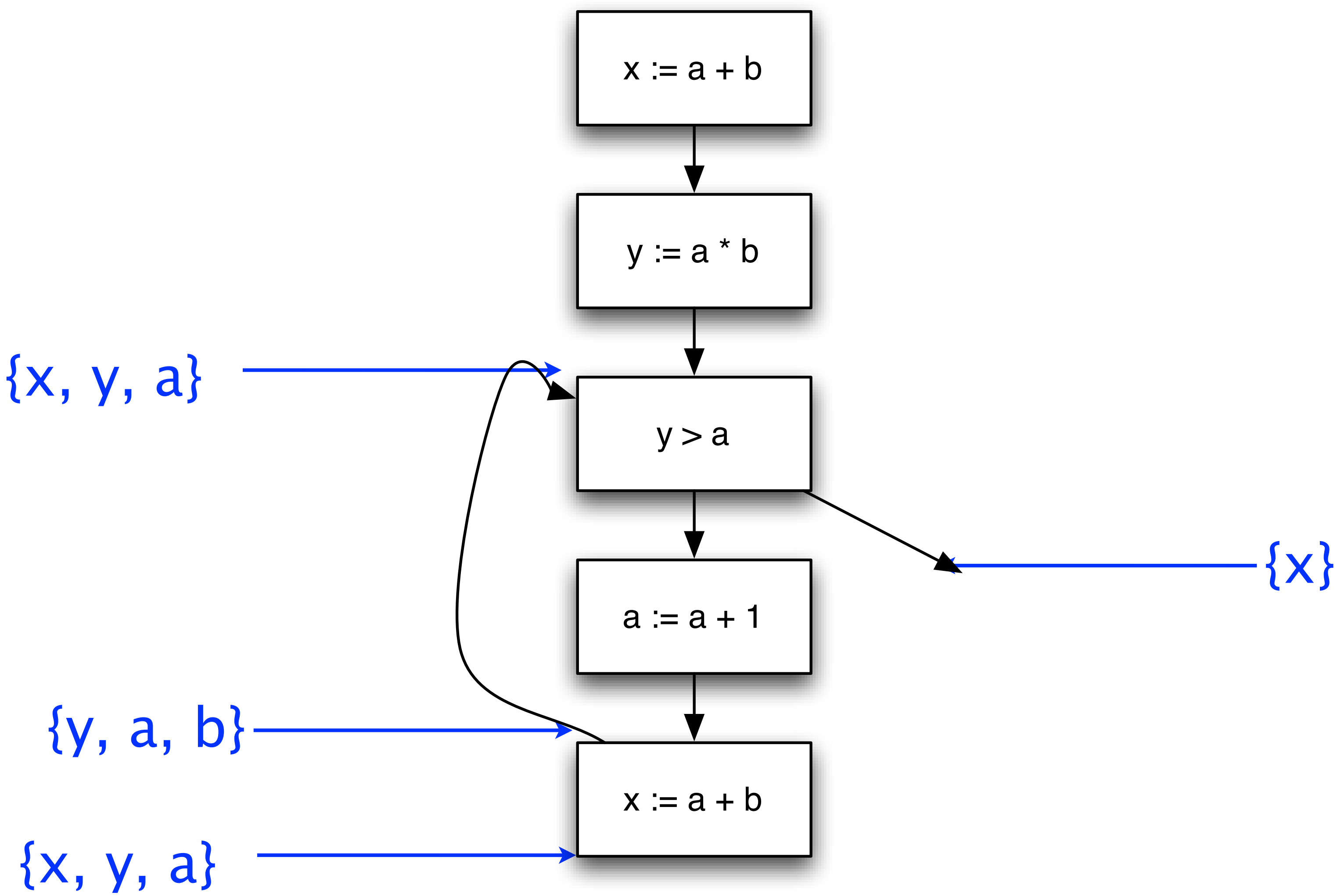
Computing Live Variables



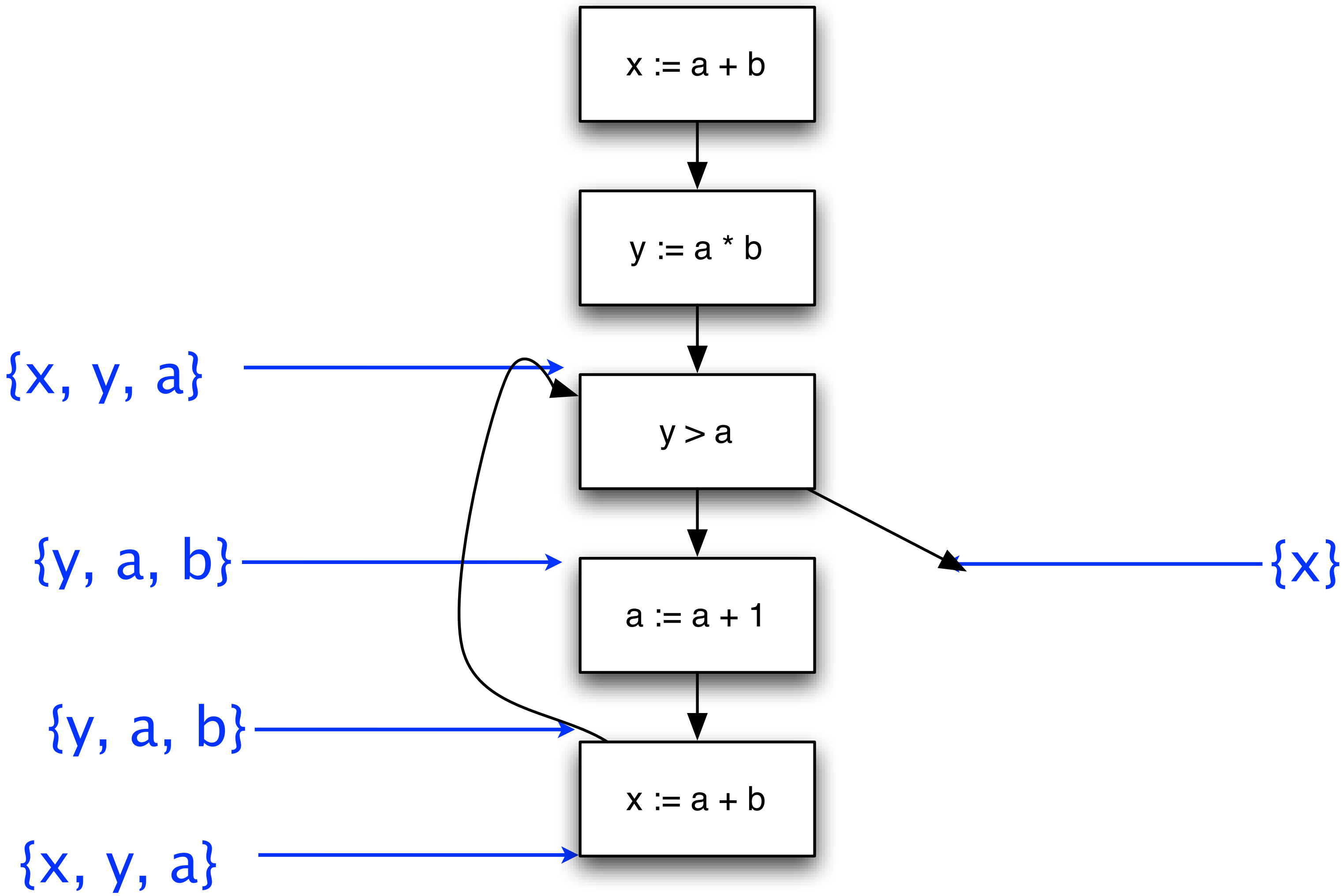
Computing Live Variables



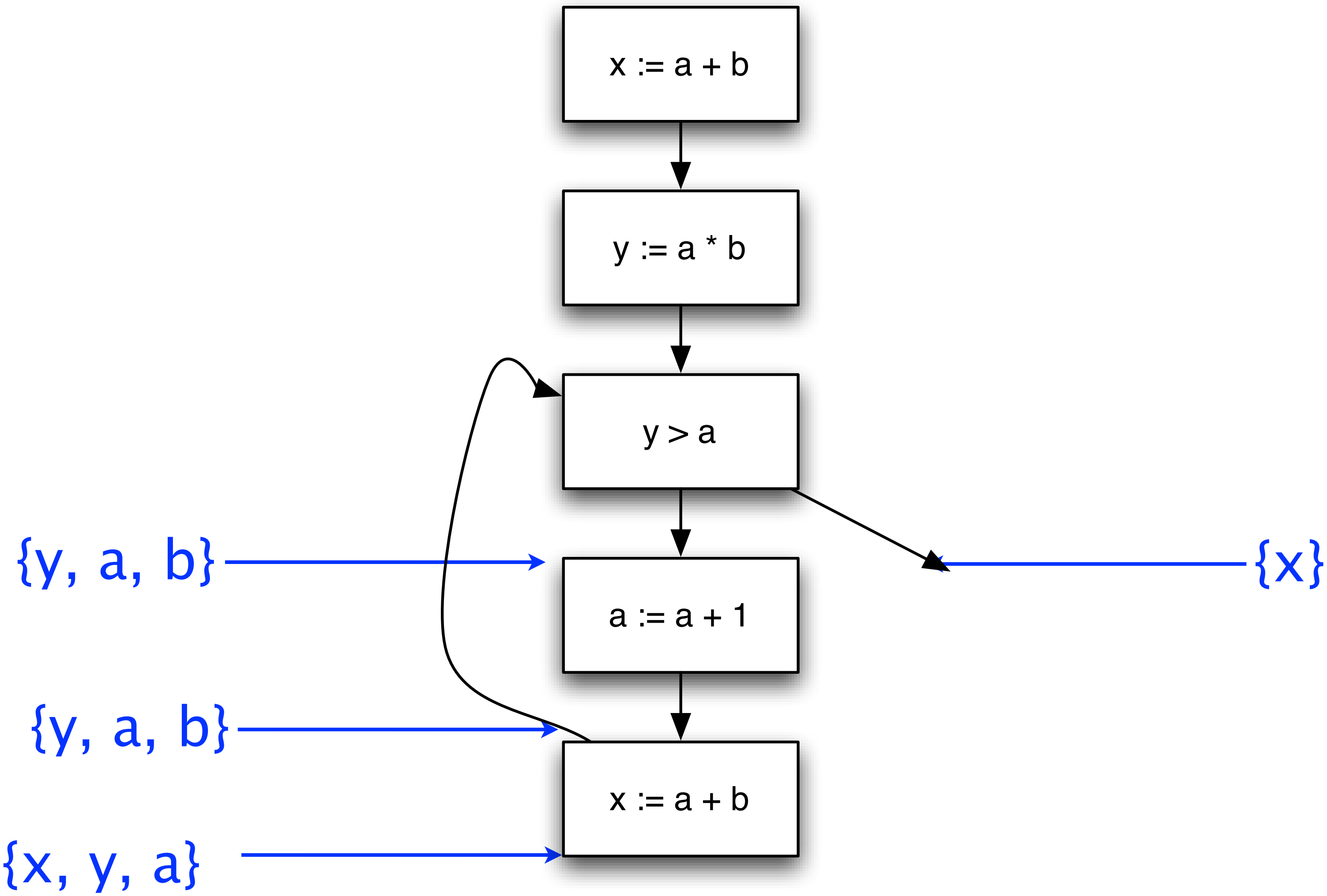
Computing Live Variables



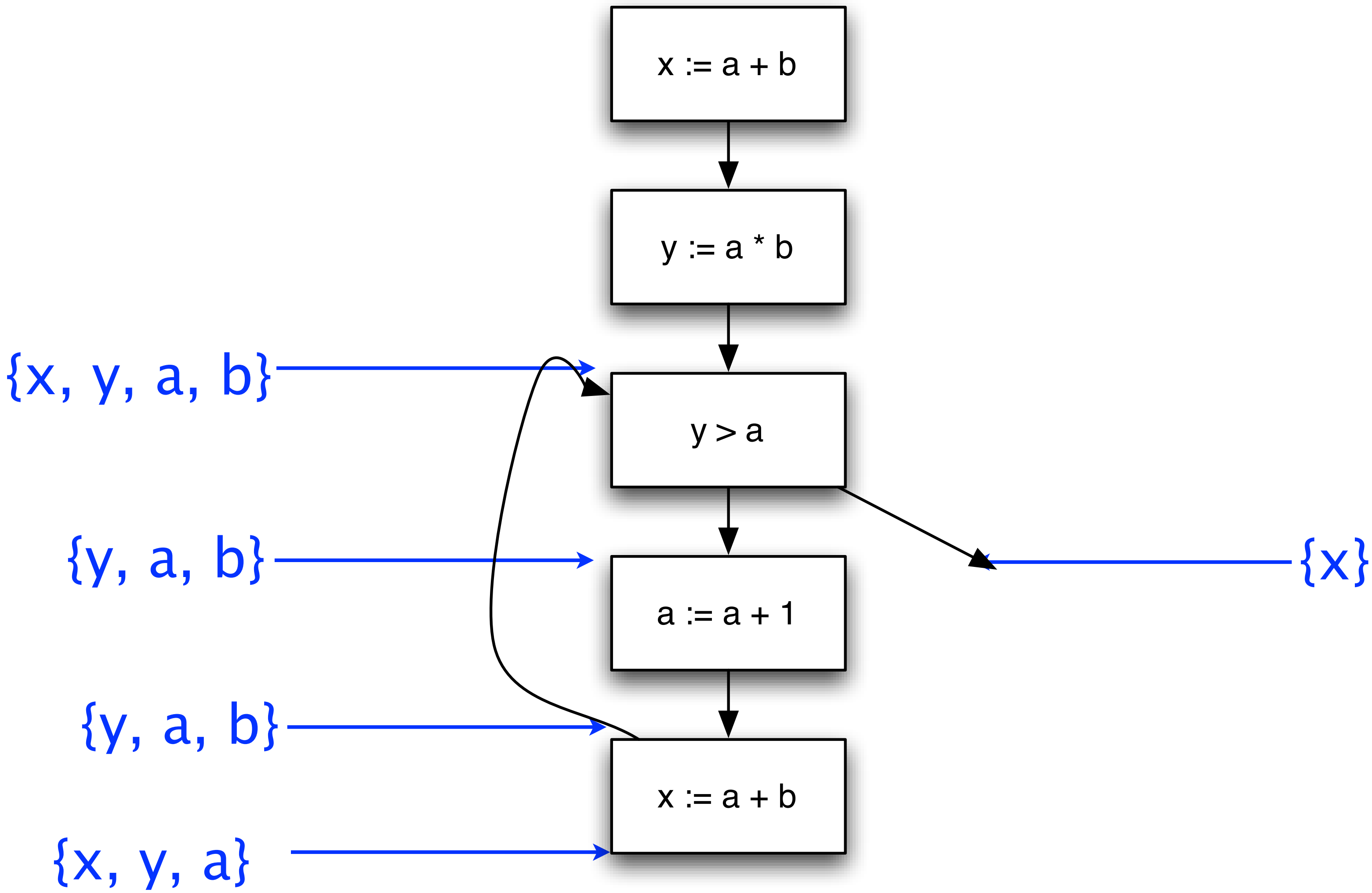
Computing Live Variables



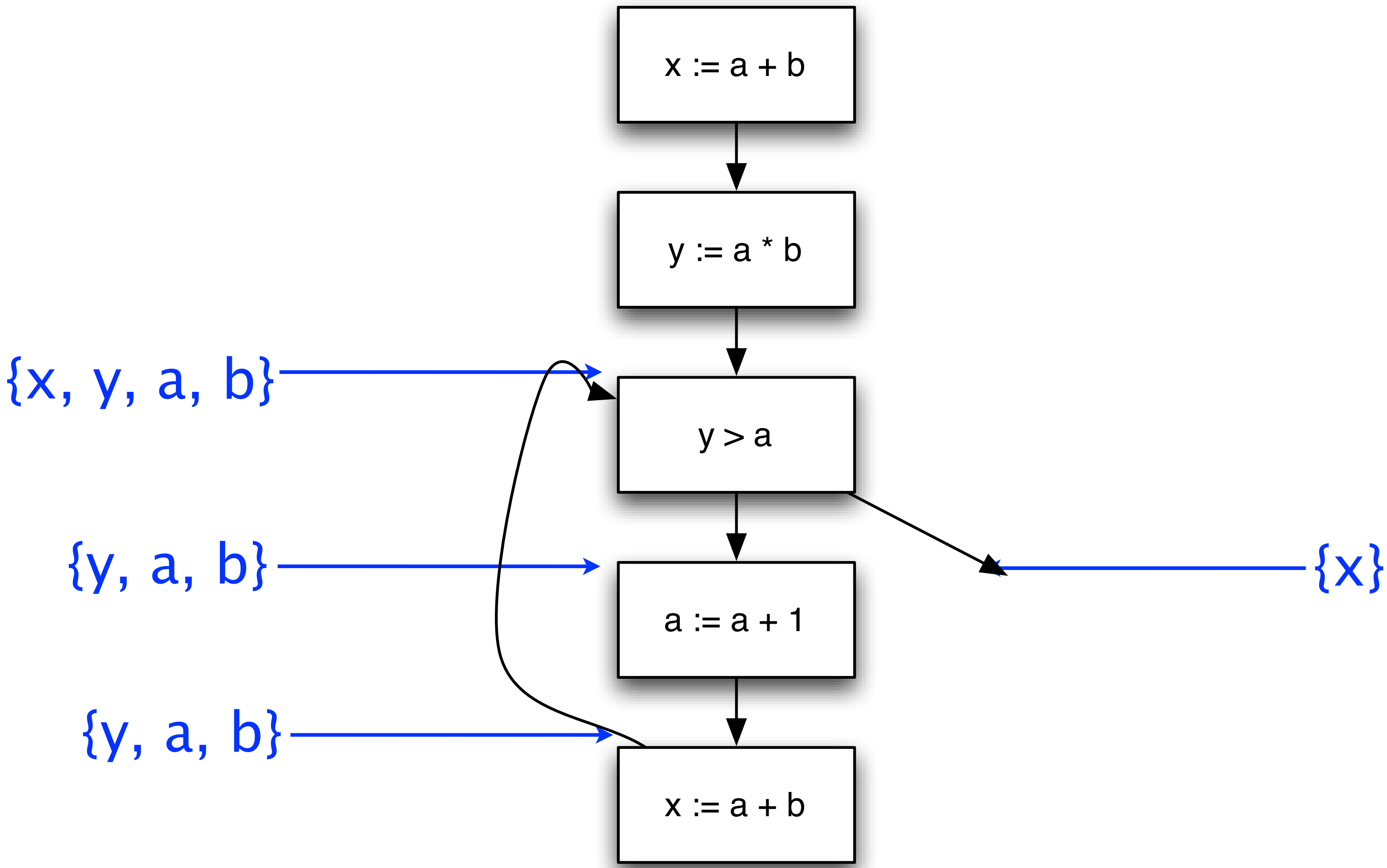
Computing Live Variables



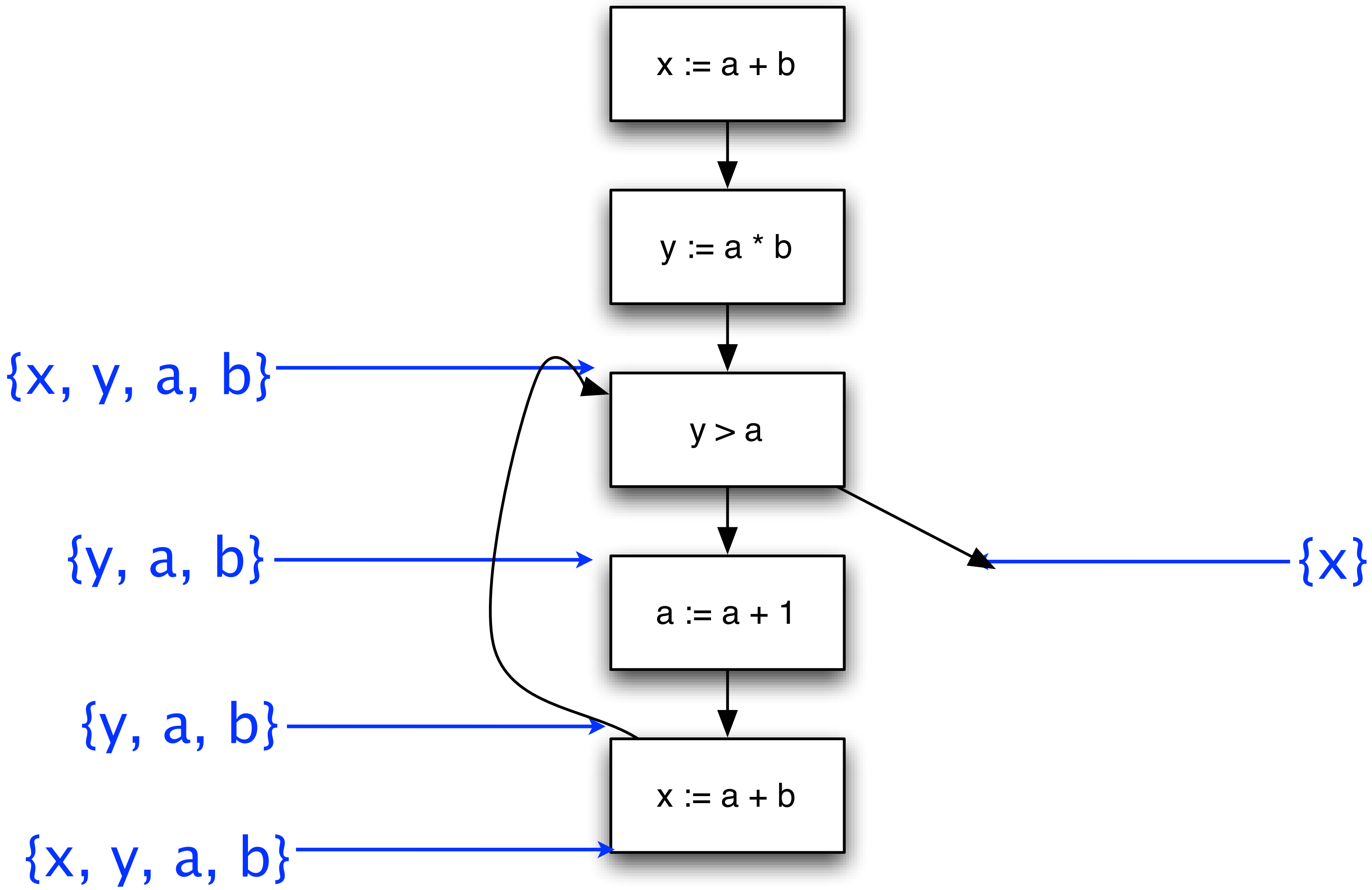
Computing Live Variables



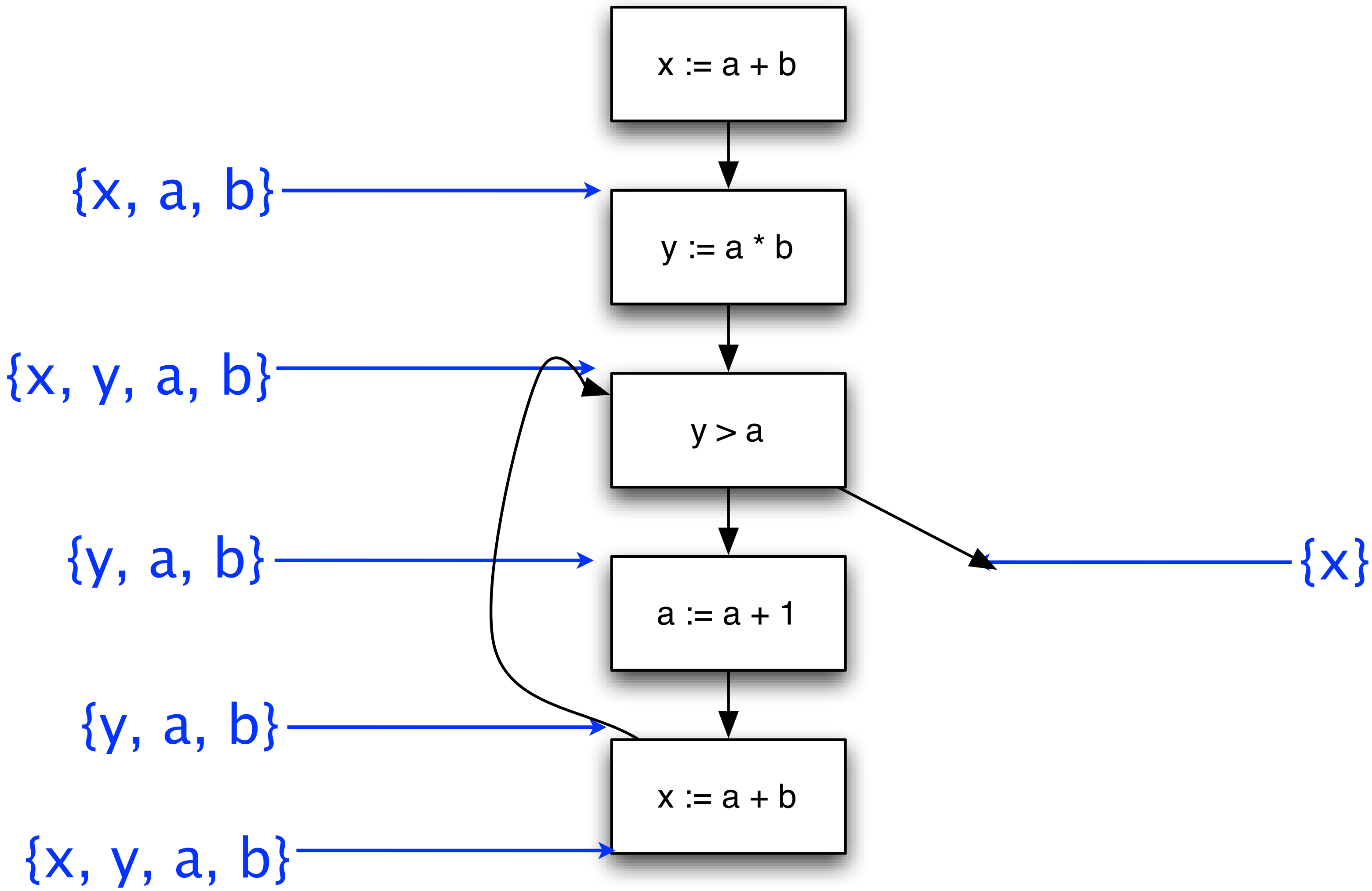
Computing Live Variables



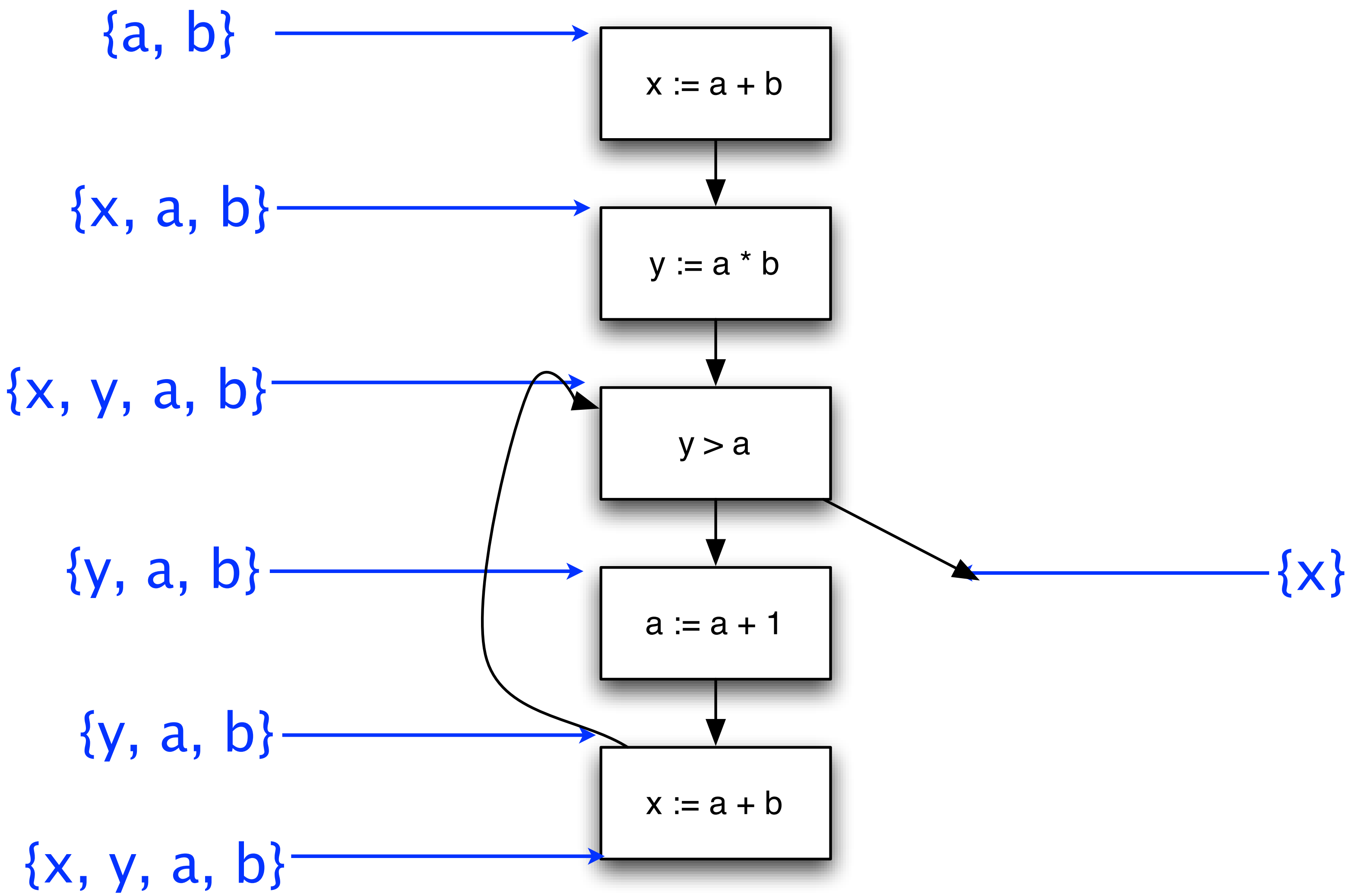
Computing Live Variables



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

Solving data flow equations

- Let's start with forward may analysis
  - Dataflow equations:
    - $in(s) = \bigcup_{s' \in pred(s)} out(s')$
    - $out(s) = gen(s) \cup (in(s) - kill(s))$
- Need algorithm to compute in and out at each stmt
- Key observation: out(s) is *monotonic* in in(s)
  - gen(s) and kill(s) are fixed for a given s
  - If, during our algorithm, in(s) grows, then out(s) grows
  - Furthermore, out(s) and in(s) have max size
- Same with in(s)
  - in terms of out(s') for precedessors s'

Solving data flow equations (cont'd)

- Idea: fixpoint algorithm
  - Set out(entry) to emptyset
    - E.g., we know no definitions reach the entry of the program
  - Initially, assume in(s), out(s) empty everywhere else, also
  - Pick a statement s
    - Compute in(s) from predecessors' out's
    - Compute new out(s) for s
  - Repeat until nothing changes
- Improvement: use a worklist
  - Add statements to worklist if their in(s) might change
  - Fixpoint reached when worklist is empty

Forward May Data Flow Algorithm

```
out(entry) = ∅
for all other statements s
  out(s) = ∅
W = all statements     // worklist
while W not empty
  take s from W
    in(s) = ∪ s′∈pred(s) out(s′)
    temp = gen(s) ∪ (in(s) - kill(s))
    if temp ≠ out(s) then
      out(s) = temp
      W := W ∪ succ(s)
    end
end
```

Generalizing

	May	Must
Forward	$in(s) = \bigcup_{s' \in pred(s)} out(s')$ $out(s) = gen(s) \cup (in(s) - kill(s))$ $out(entry) = \varnothing$ initial out elsewhere = $\varnothing$	$in(s) = \bigcap_{s' \in pred(s)} out(s')$ $out(s) = gen(s) \cup (in(s) - kill(s))$ $out(entry) = \varnothing$ initial out elsewhere = {all facts}
Backward	$out(s) = \bigcup_{s' \in succ(s)} in(s')$ $in(s) = gen(s) \cup (out(s) - kill(s))$ $in(exit) = \varnothing$ initial in elsewhere = $\varnothing$	$out(s) = \bigcap_{s' \in succ(s)} in(s')$ $in(s) = gen(s) \cup (out(s) - kill(s))$ $in(exit) = \varnothing$ initial in elsewhere = {all facts}

Forward Analysis

May	Must
$out(entry) = \varnothing$	$out(entry) = \varnothing$
for all other statements s	for all other statements s
$out(s) = \varnothing$	$out(s) =$ *all facts*
W = *all statements* *// worklist*	W = *all statements*
while W not empty	while W not empty
take s from W	take s from W
$in(s) = \cup_{s' \in pred(s)} out(s')$	$in(s) = \cap_{s' \in pred(s)} out(s')$
temp = $gen(s) \cup (in(s) - kill(s))$	temp = $gen(s) \cup (in(s) - kill(s))$
if temp $\neq out(s)$ then	if temp $\neq out(s)$ then
$out(s)$ = temp	$out(s)$ = temp
W := W $\cup$ succ(s)	W := W $\cup$ succ(s)
end	end
end	end

Backward Analysis

May

```
in(exit) = ∅
for all other statements s
  in(s) = ∅
W = all statements
while W not empty
  take s from W
    out(s) = ∪_{s′∈succ(s)} in(s′)
    temp = gen(s) ∪ (out(s) - kill(s))
    if temp ≠ in(s) then
      in(s) = temp
      W := W ∪ pred(s)
    end
end
```

Must

```
in(exit) = ∅
for all other statements s
  in(s) = all facts
W = all statements
while W not empty
  take s from W
    out(s) = ∩_{s′∈succ(s)} in(s′)
    temp = gen(s) ∪ (out(s) - kill(s))
    if temp ≠ in(s) then
      in(s) = temp
      W := W ∪ pred(s)
    end
end
```

Practical Implementation

- 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

- Recall 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 some data flow implementations,
  - Compute gen/kill for each basic block as a whole
    - Compose transfer functions
  - Store only in/out for each basic block
  - Typical basic block ~5 statements
    - At least, this used to be the case...

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
  - (Reverse for backward analysis)

- Let Q = max # back edges on cycle-free path
  - Nesting depth
  - Back edge is from node to ancestor in DFS tree

- In common cases, running time can be shown to be $O((Q+1)|E|)$
  - Proportional to structure of CFG rather than lattice

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 */

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 *whole program*

- Note: *global* analysis means "more than one basic block," but still within a function
  - Old terminology from when computers were slow...

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

Proebsting's Law

Proebsting's Law

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

Proebsting's Law

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

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

Proebsting's Law

- 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!

DFA 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)