Grades

Grades

- **Final grade** = min(100, Weighted average)
- **Exam**: **50%**
  - **Must pass**: Exam grade < 60 ➔ Final grade = Exam grade
  - **Max grade: 100**
  - Format: Same as last year (but **no bonus questions**)
- **Exercises**: **60%** - Bonuses **accumulate** up to **10%**
  - Ex 0: 2.5%
  - Ex 1: 5%
  - Ex 2: 7.5%
  - Ex 3: 12.5%
  - Ex 4: 12.5%
  - Theoretical Ex: 10%
  - Ex 5*: 10%

*Inform Orr if you want to do it. If you do, then you will get your final grade after Moed B (even if you decide not to submit.)

Compilation

0368-3133 2014/15a

Lecture 12

Data Flow Analysis & Optimizations

Noam Rinetzky

Optimization points

source code → Front end → IR → Code generator → target code

User
profile program
change algorithm

Compiler
intraprocedural IR
Interprocedural IR
IR optimizations

Compiler
register allocation
instruction selection
peephole transformations

now

Program Analysis

- Reasons about the **behavior** of a program
- An analysis is **sound** if it only asserts an correct facts about a program
- An analysis is **precise** if it asserts all correct facts (of interests)
- Sound analysis allows for **semantic-preserving optimizations**
  - "More precise" analyses are "more useful": may enable more optimizations

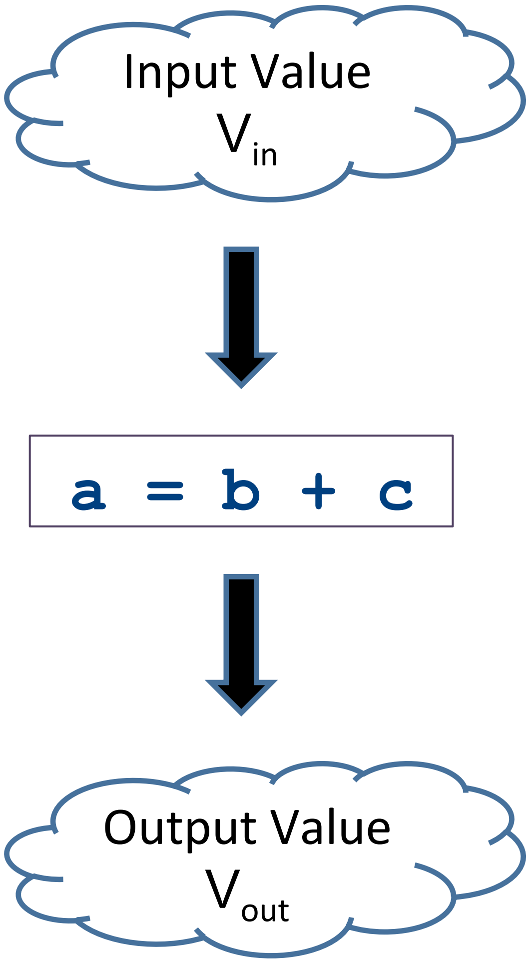
Examples

- Available expressions, allows:
  - Common sub-expressions elimination
  - Copy propagation
- Constant propagation, allows:
  - Constant folding
- Liveness analysis
  - Dead-code elimination
  - Register allocation

Local vs. global optimizations

- An optimization is local if it works on just a single basic block
- An optimization is global if it works on an entire control-flow graph of a procedure
- An optimization is interprocedural if it works across the control-flow graphs of multiple procedure
  - We won't talk about this in this course

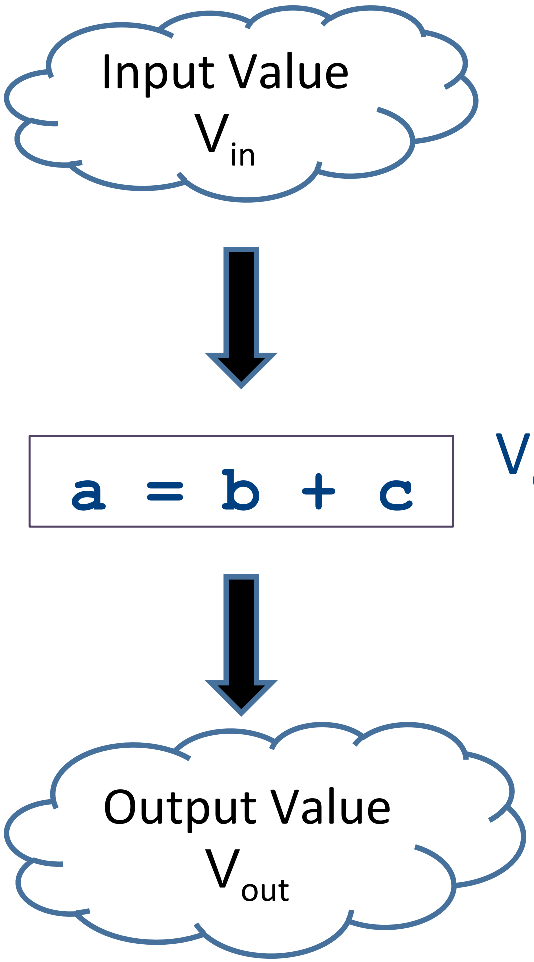
Formalizing local analyses



Transfer Function

$$V_{out} = f_{\mathtt{a=b+c}}(V_{in})$$

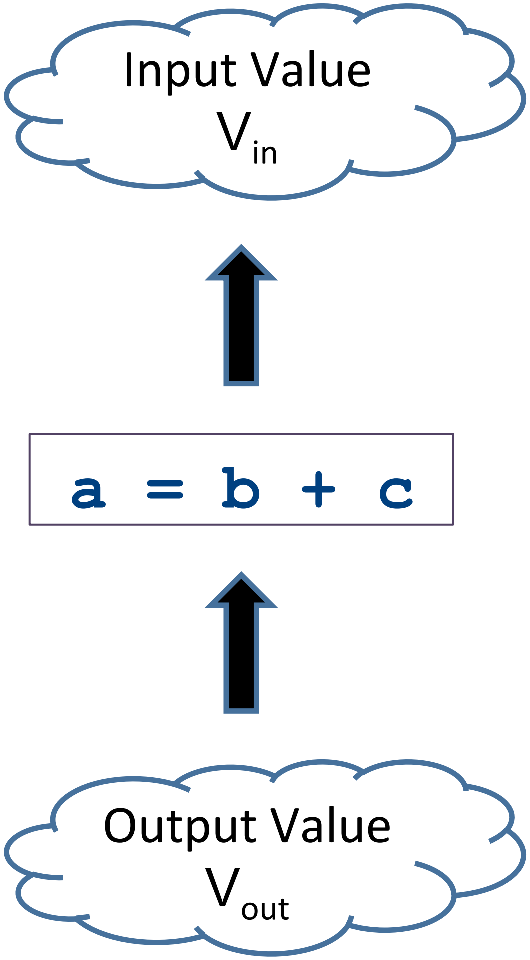
Available Expressions



$V_{out} = (V_{in} \setminus \{e \mid e \text{ contains } \mathbf{a}\}) \cup \{a=b+c\}$

Expressions of the forms
a=… and x=…**a**…

Live Variables



$$V_{in} = (V_{out} \setminus \{\mathbf{a}\}) \cup \{\mathbf{b},\mathbf{c}\}$$

Information for a local analysis

- What direction are we going?
  - Sometimes forward (available expressions)
  - Sometimes backward (liveness analysis)
- How do we update information after processing a statement?
  - What are the new semantics?
  - What information do we know initially?

Formalizing local analyses

- Define an analysis of a basic block as a quadruple (D, V, F, I) where
  - **D** is a direction (forwards or backwards)
  - **V** is a set of values the program can have at any point
  - **F** is a family of transfer functions defining the meaning of any expression as a function $f : \mathbf{V} \rightarrow \mathbf{V}$
  - I is the initial information at the top (or bottom) of a basic block

Available Expressions

- **Direction:** Forward
- **Values:** Sets of expressions assigned to variables
- **Transfer functions:** Given a set of variable assignments V and statement a = b + c:
  - Remove from V any expression containing a as a subexpression
  - Add to V the expression a = b + c
  - Formally: $V_{out} = (V_{in} \setminus \{e \mid e \text{ contains } \mathbf{a}\}) \cup \{a = b + c\}$
- **Initial value:** Empty set of expressions

Liveness Analysis

- **Direction:** Backward
- **Values:** Sets of variables
- **Transfer functions:** Given a set of variable assignments V and statement a = b + c:
- Remove a from V (any previous value of a is now dead.)
- Add b and c to V (any previous value of b or c is now live.)
- Formally: $V_{in} = (V_{out} \setminus \{\mathtt{a}\}) \cup \{\mathtt{b},\mathtt{c}\}$
- **Initial value:** Depends on semantics of language
  - E.g., function arguments and return values (pushes)
  - Result of local analysis of other blocks as part of a global analysis

Running local analyses

- Given an analysis **(D, V, F, I)** for a basic block
- Assume that **D** is "forward;" analogous for the reverse case
- Initially, set OUT[**entry**] to **I**
- For each statement **s**, in order:
  - Set IN[**s**] to OUT[**prev**], where **prev** is the previous statement
  - Set OUT[**s**] to f_s(IN[**s**]), where f_s is the transfer function for statement **s**

Global Optimizations

High-level goals

- Generalize analysis mechanism
  – Reuse common ingredients for many analyses
  – Reuse proofs of correctness
- Generalize from basic blocks to entire CFGs
  – Go from local optimizations to global optimizations

Global analysis

- A global analysis is an analysis that works on a control-flow graph as a whole
- Substantially more powerful than a local analysis
  - (Why?)
- Substantially more complicated than a local analysis
  - (Why?)

Local vs. global analysis

- Many of the optimizations from local analysis can still be applied globally
  - Common sub-expression elimination
  - Copy propagation
  - Dead code elimination
- Certain optimizations are possible in global analysis that aren't possible locally:
  - e.g. code motion: Moving code from one basic block into another to avoid computing values unnecessarily
- Example global optimizations:
  - Global constant propagation
  - Partial redundancy elimination

Loop invariant code motion example

```
while (t < 120) {
  z = z + x - y;
}
```

value of expression x – y is not changed by loop body

```
w = x - y;
while (t < 120) {
  z = z + w;
}
```

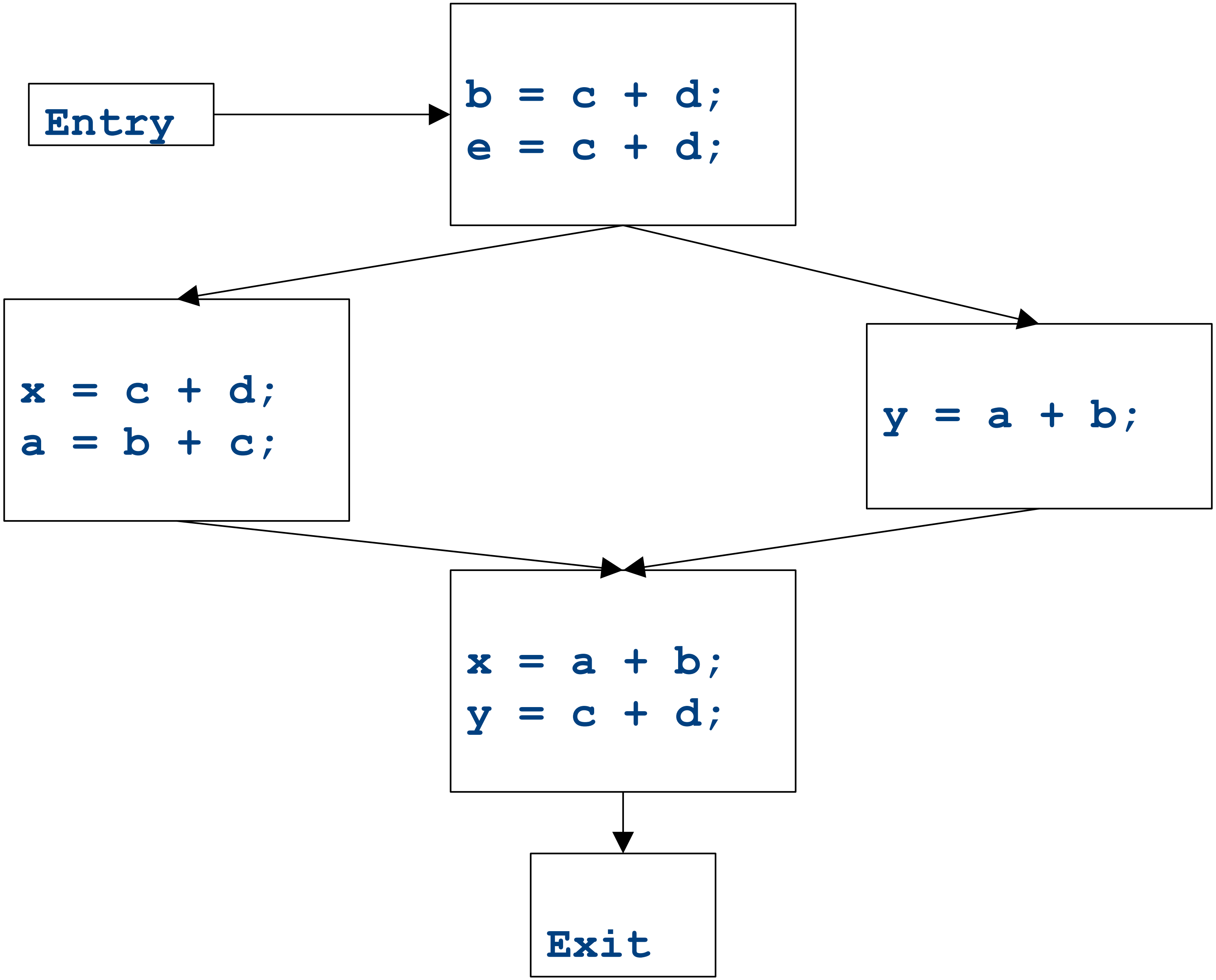
Why global analysis is hard

- Need to be able to handle multiple predecessors/successors for a basic block
- Need to be able to handle multiple paths through the control-flow graph, and may need to iterate multiple times to compute the final value (but the analysis still needs to terminate!)
- Need to be able to assign each basic block a reasonable default value for before we've analyzed it

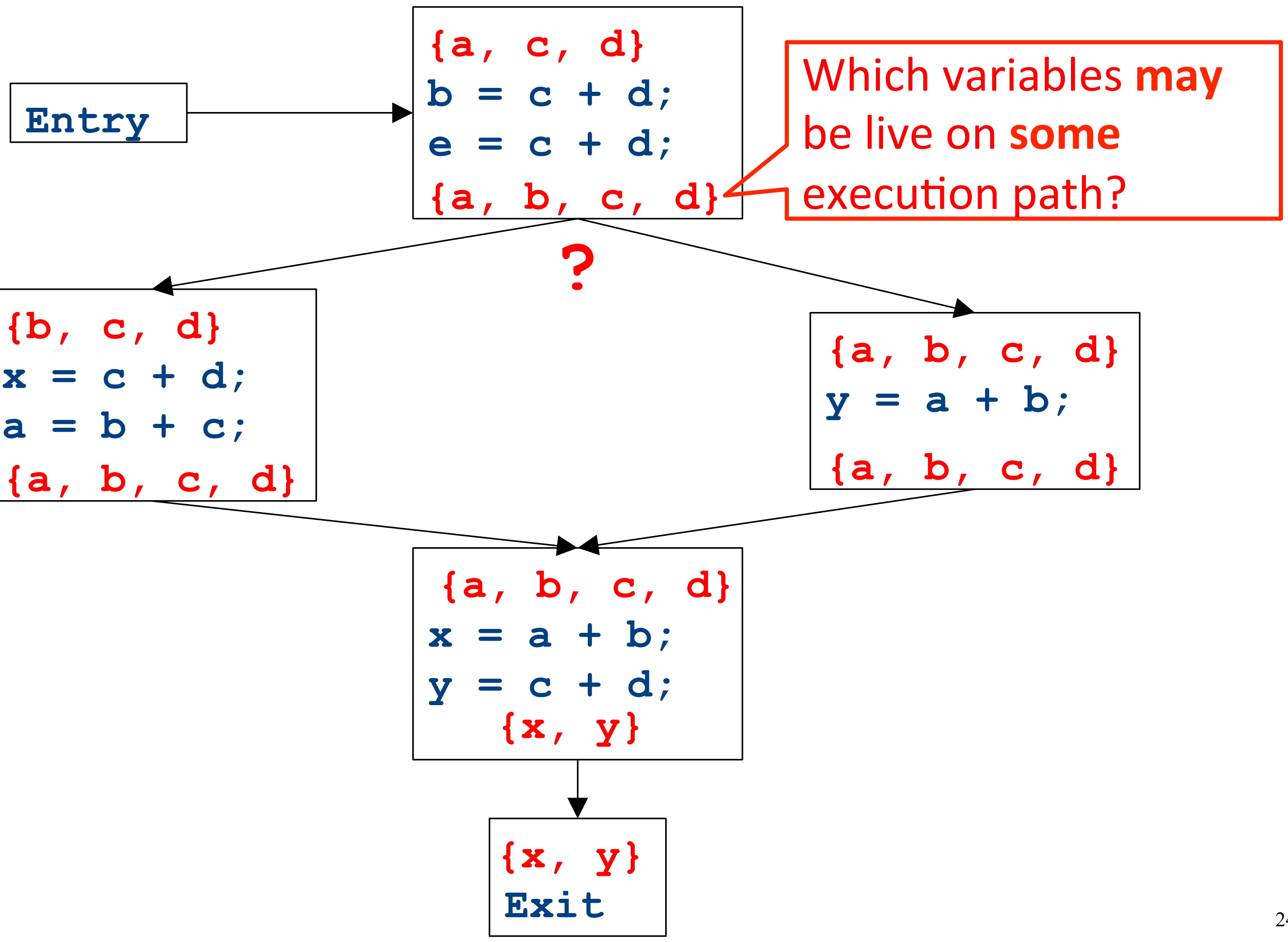
Global dead code elimination

- Local dead code elimination needed to know what variables were live on exit from a basic block
- This information can only be computed as part of a global analysis
- How do we modify our liveness analysis to handle a CFG?

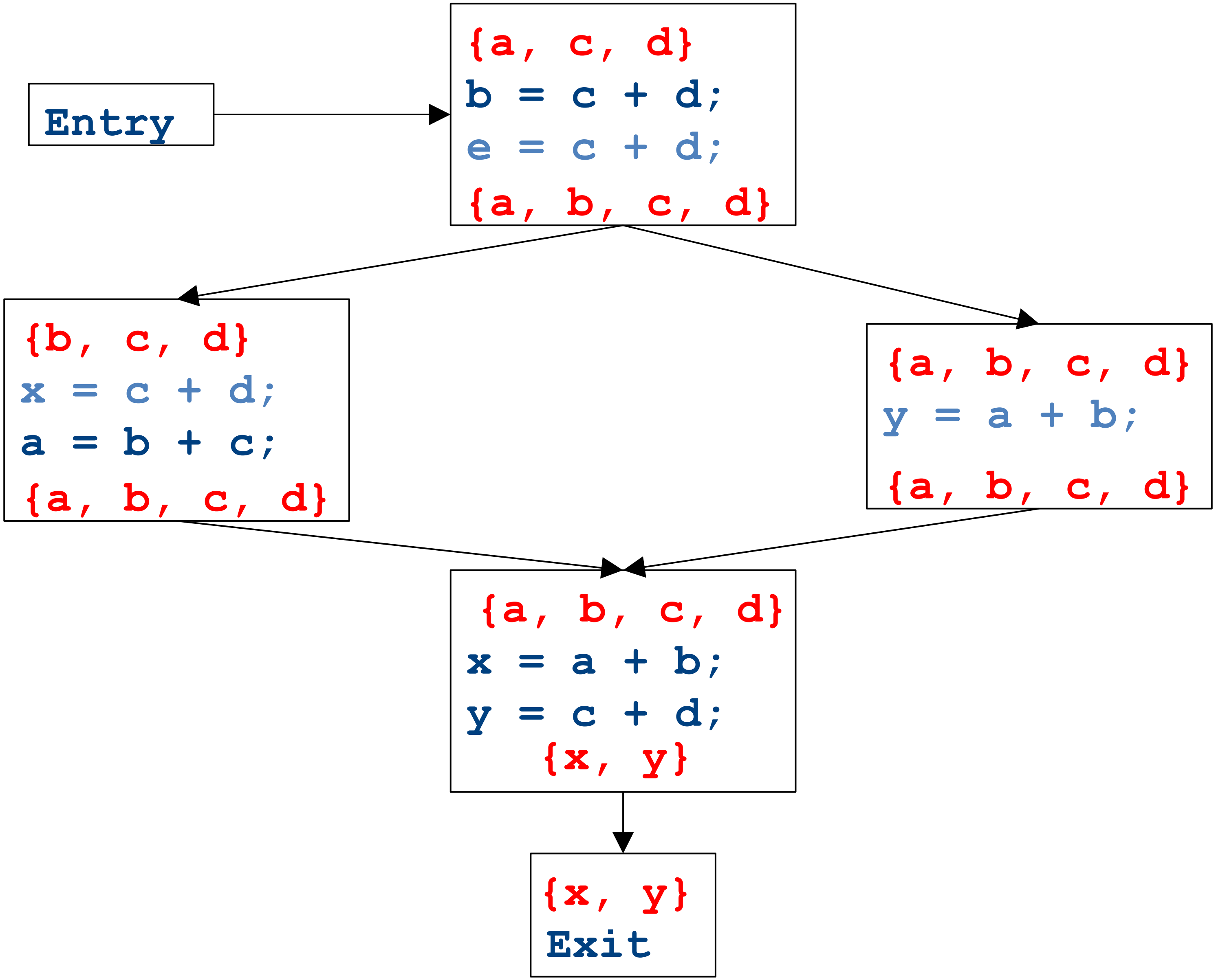
CFGs without loops



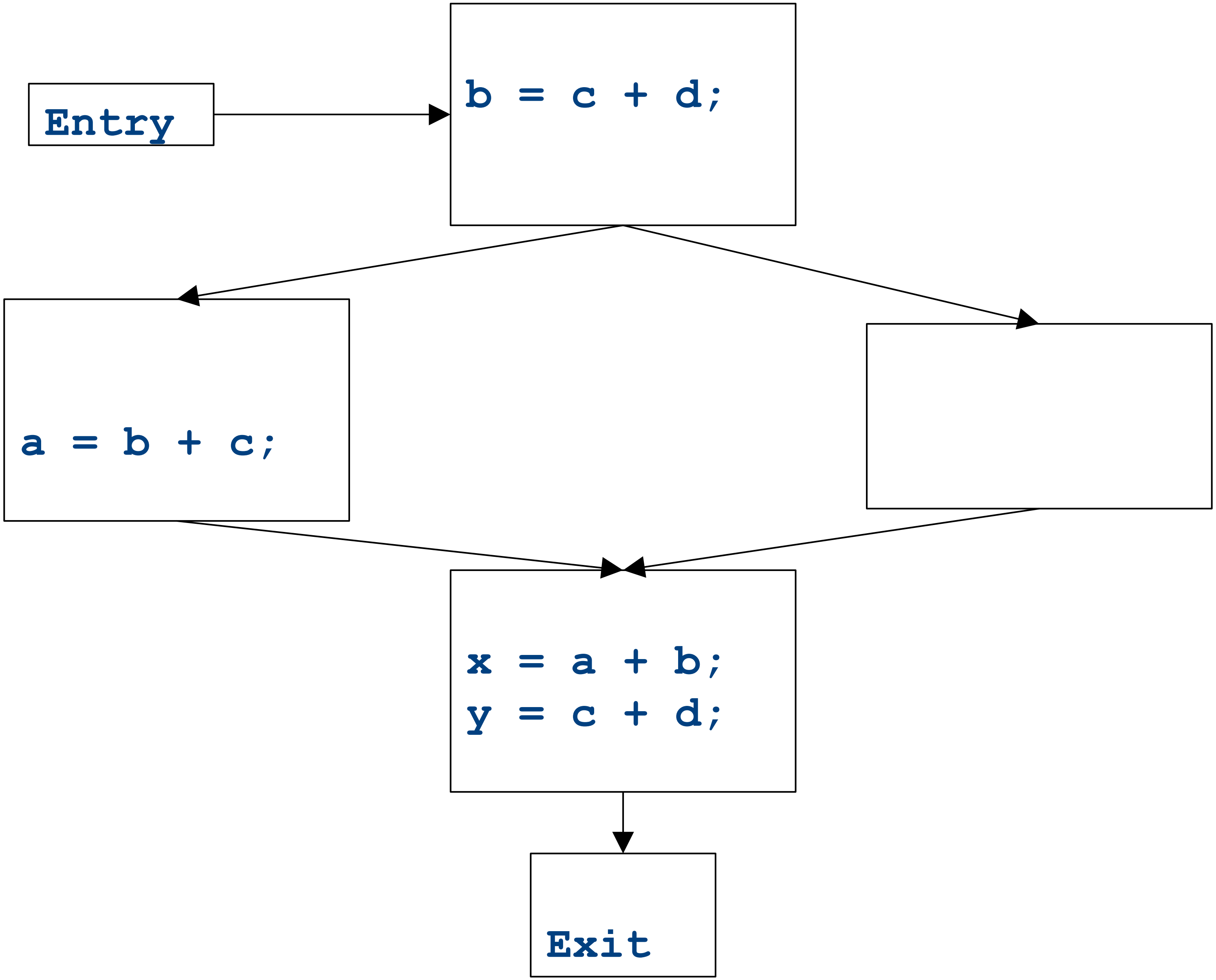
CFGs without loops



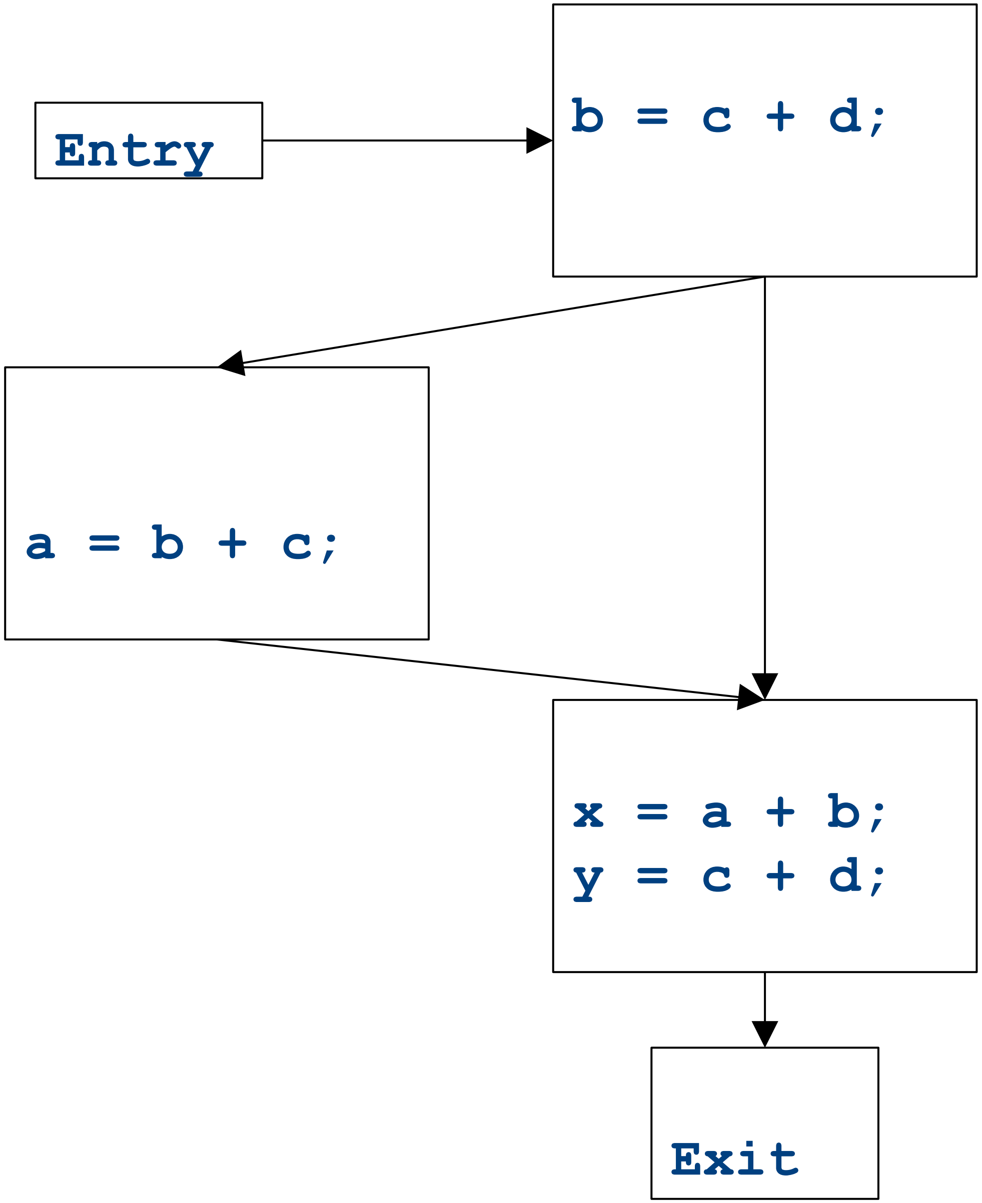
CFGs without loops



CFGs without loops



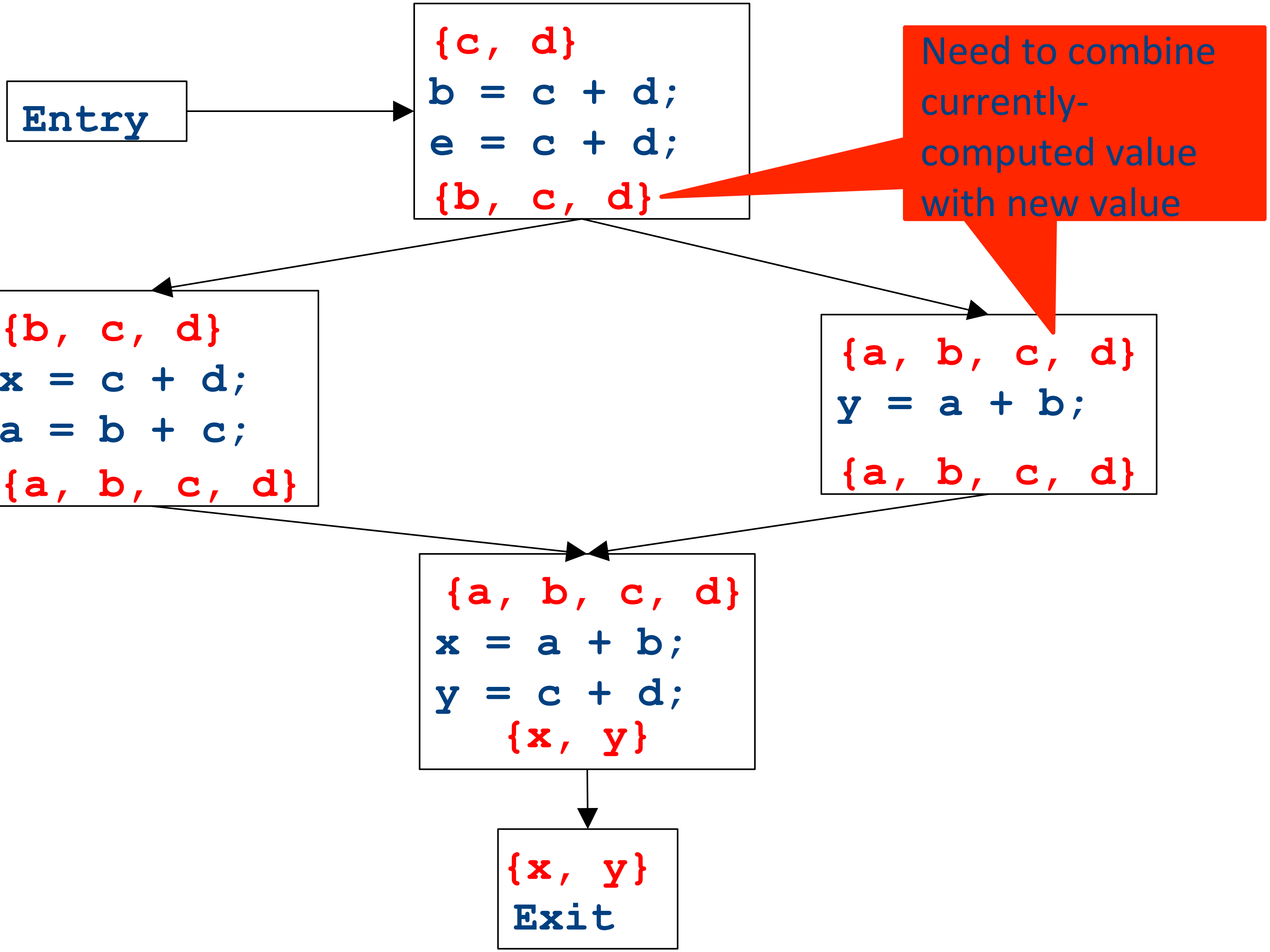
CFGs without loops



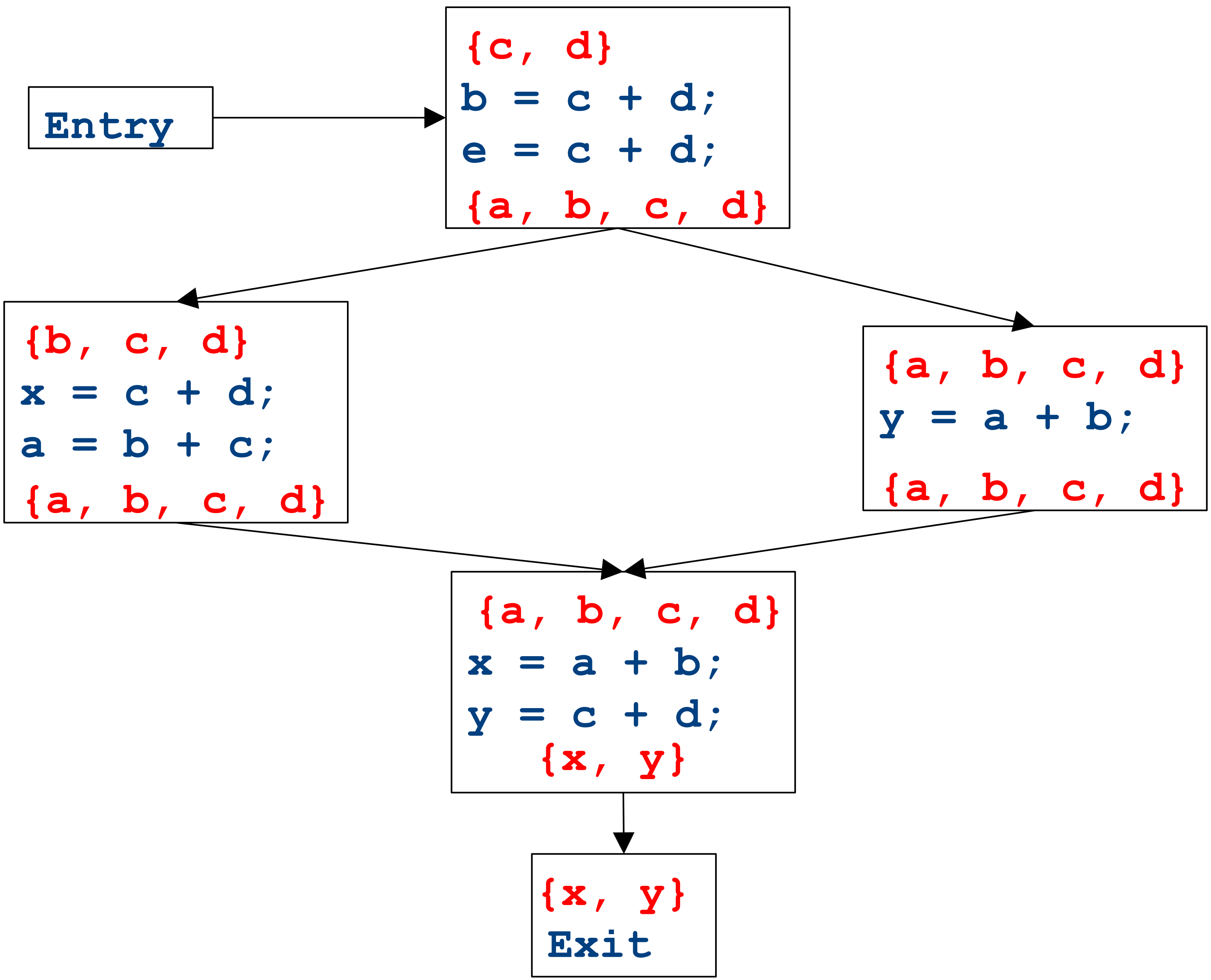
Major changes – part 1

- In a local analysis, each statement has exactly one predecessor
- In a global analysis, each statement may have **multiple** predecessors
- A global analysis must have some means of **combining information** from all predecessors of a basic block

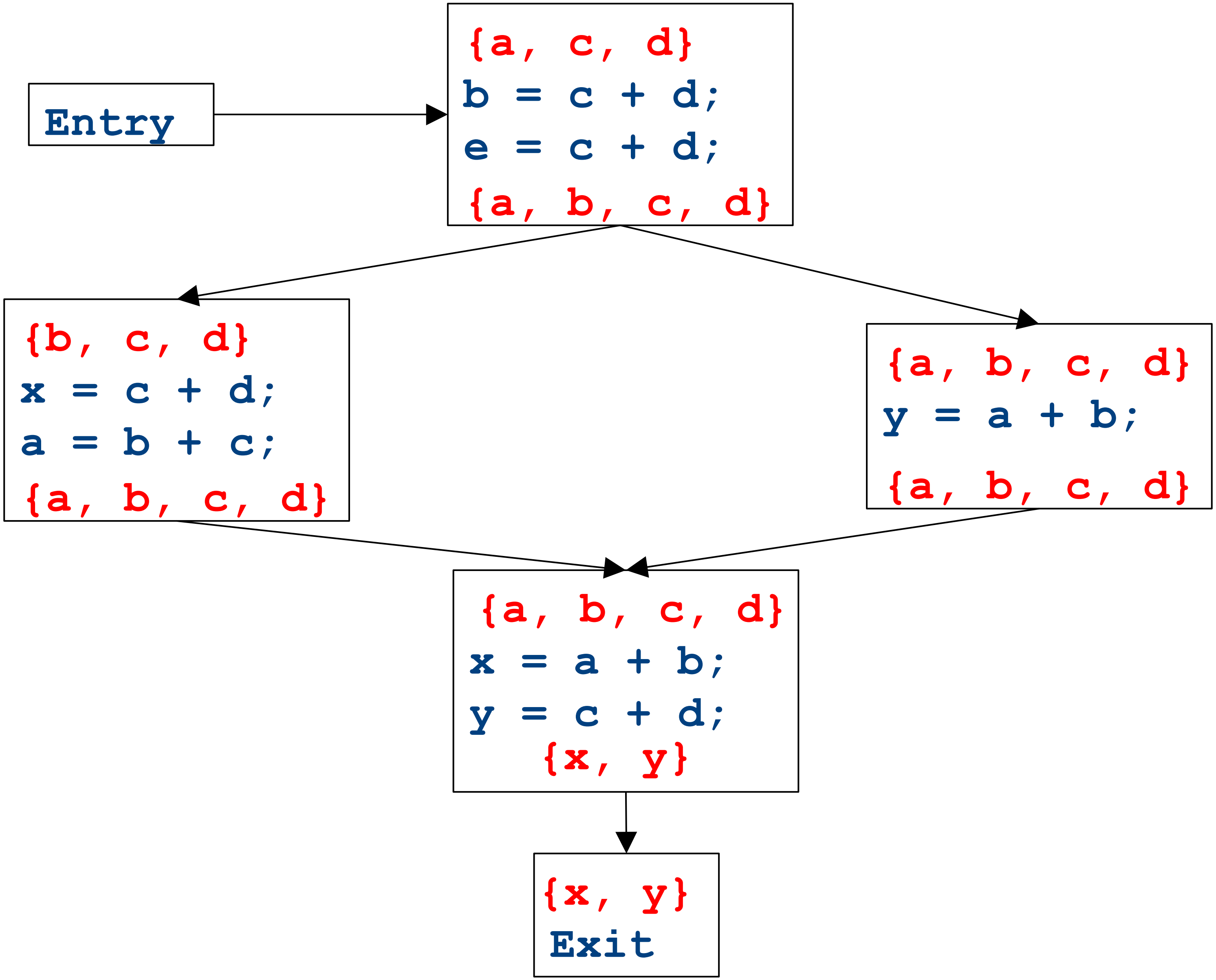
CFGs without loops



CFGs without loops



CFGs without loops

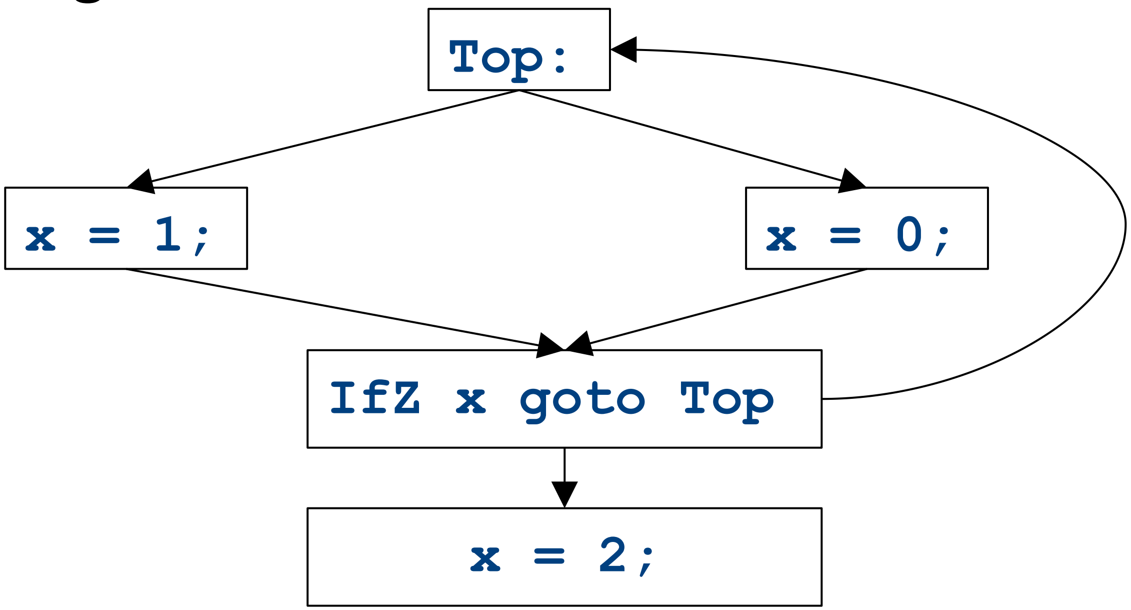


Major changes – part 2

- In a local analysis, there is only one possible path through a basic block
- In a global analysis, there may be **many** paths through a CFG
- May need to recompute values multiple times as more information becomes available
- Need to be careful when doing this not to loop infinitely!
  - (More on that later)

CFGs with loops

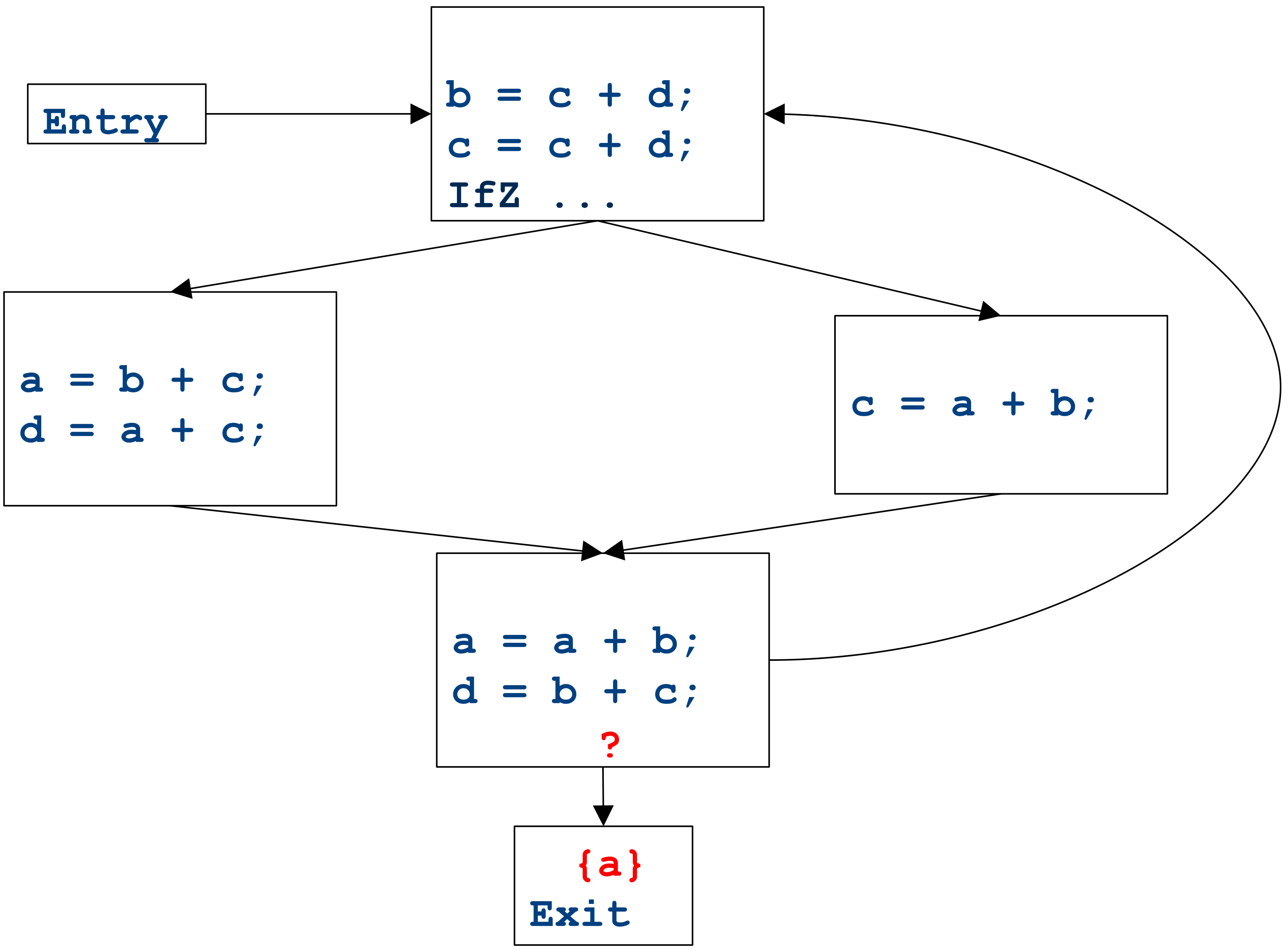
- Up to this point, we've considered loop-free CFGs, which have only finitely many possible paths
- When we add loops into the picture, this is no longer true
- Not all possible loops in a CFG can be realized in the actual program



CFGs with loops

- Up to this point, we've considered loop-free CFGs, which have only finitely many possible paths
- When we add loops into the picture, this is no longer true
- Not all possible loops in a CFG can be realized in the actual program
- **Sound approximation:** Assume that every possible path through the CFG corresponds to a valid execution
  - Includes all realizable paths, but some additional paths as well
  - May make our analysis less precise (but still sound)
  - Makes the analysis feasible; we'll see how later

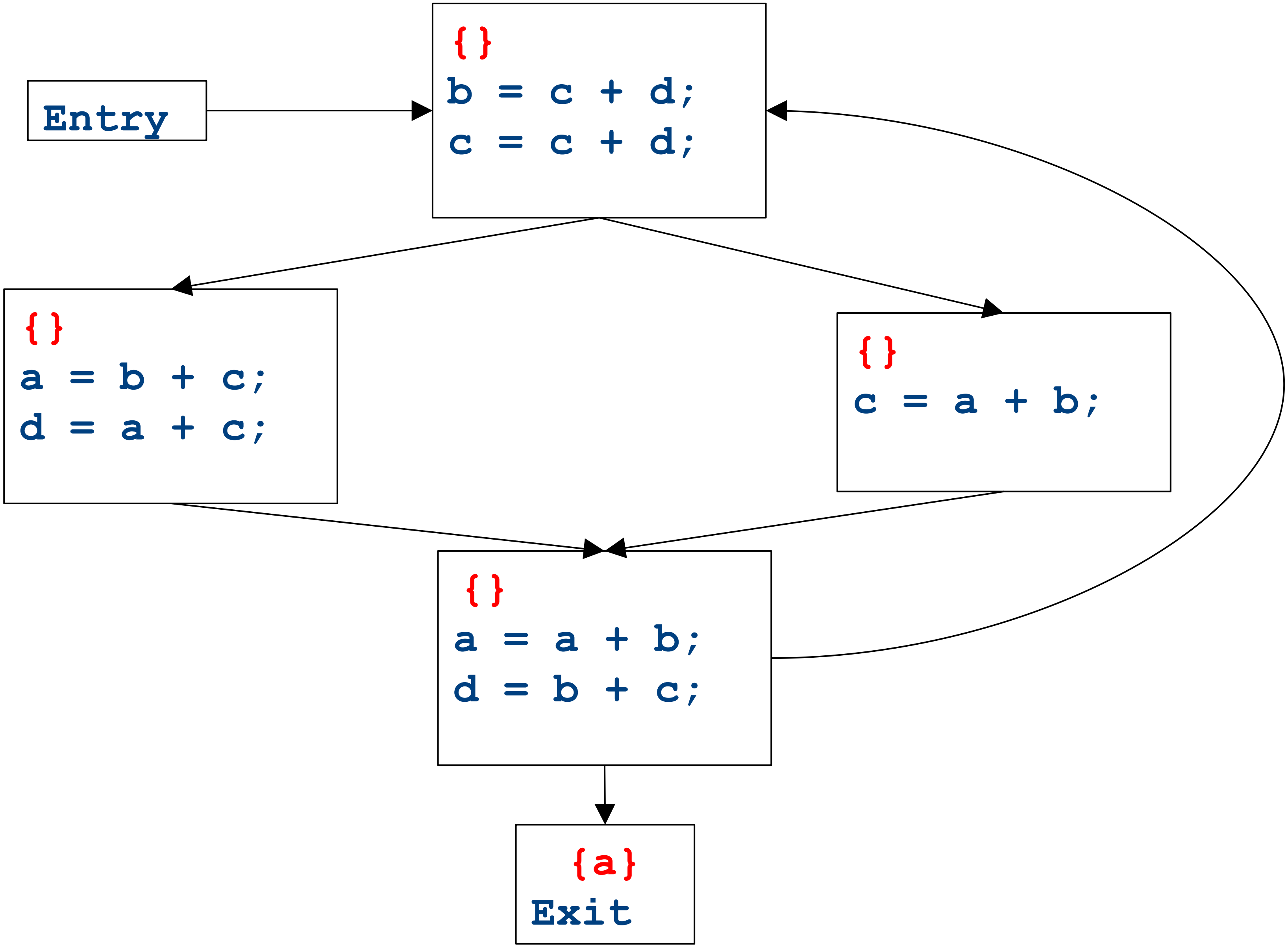
CFGs with loops



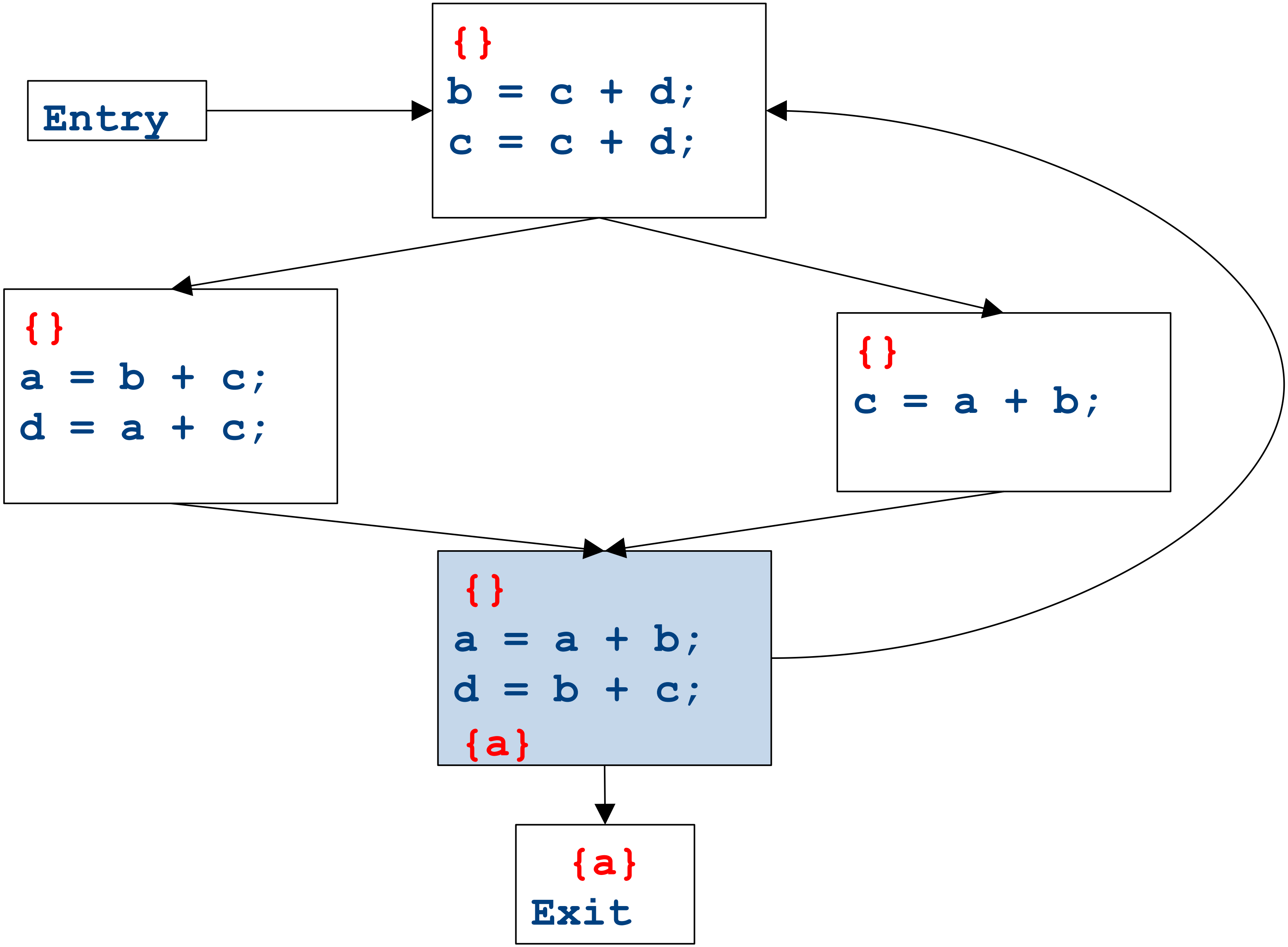
Major changes – part 3

- In a local analysis, there is always a well defined “first” statement to begin processing
- In a global analysis with loops, every basic block might depend on every other basic block
- To fix this, we need to assign initial values to all of the blocks in the CFG

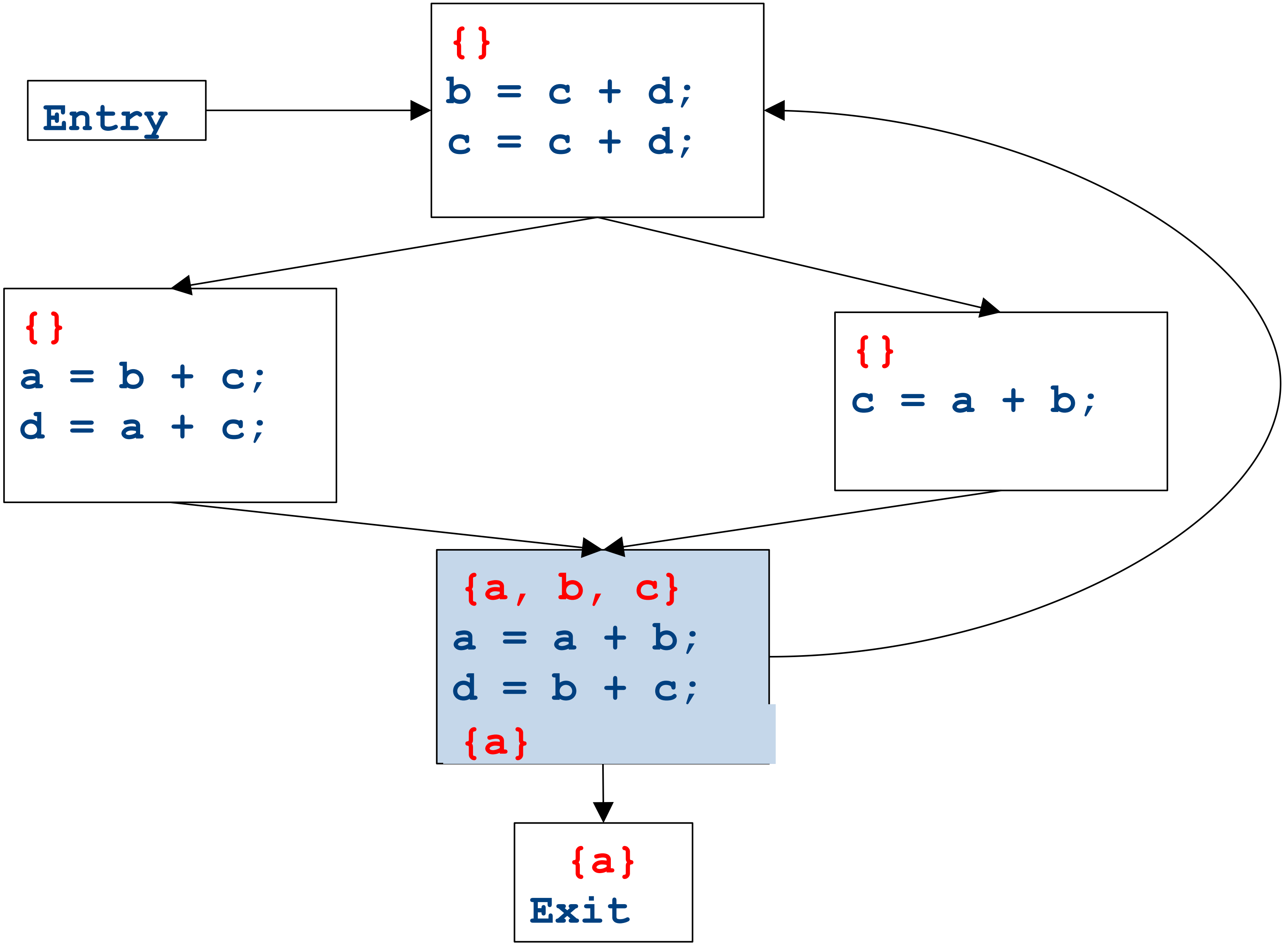
CFGs with loops - initialization



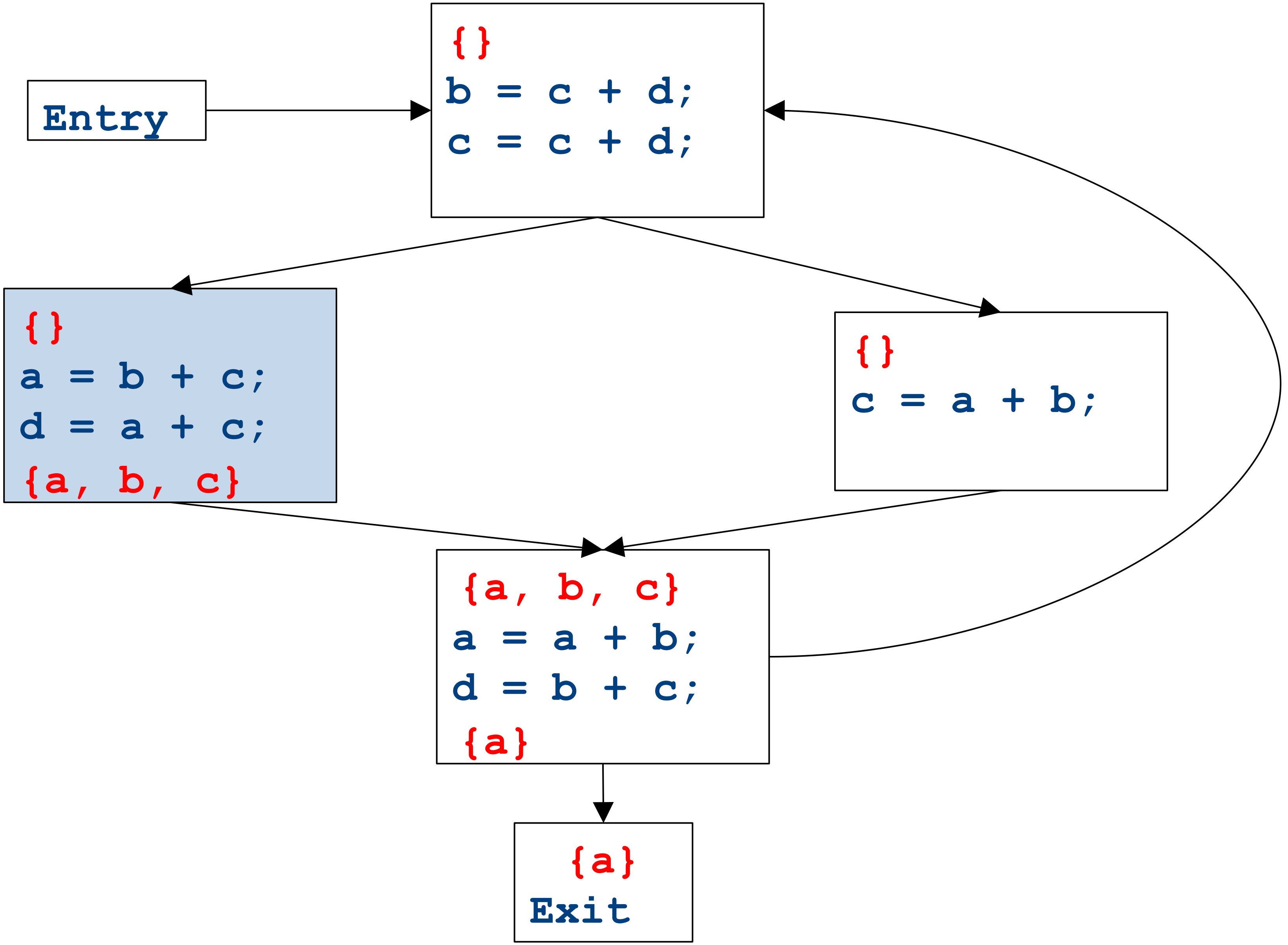
CFGs with loops - iteration



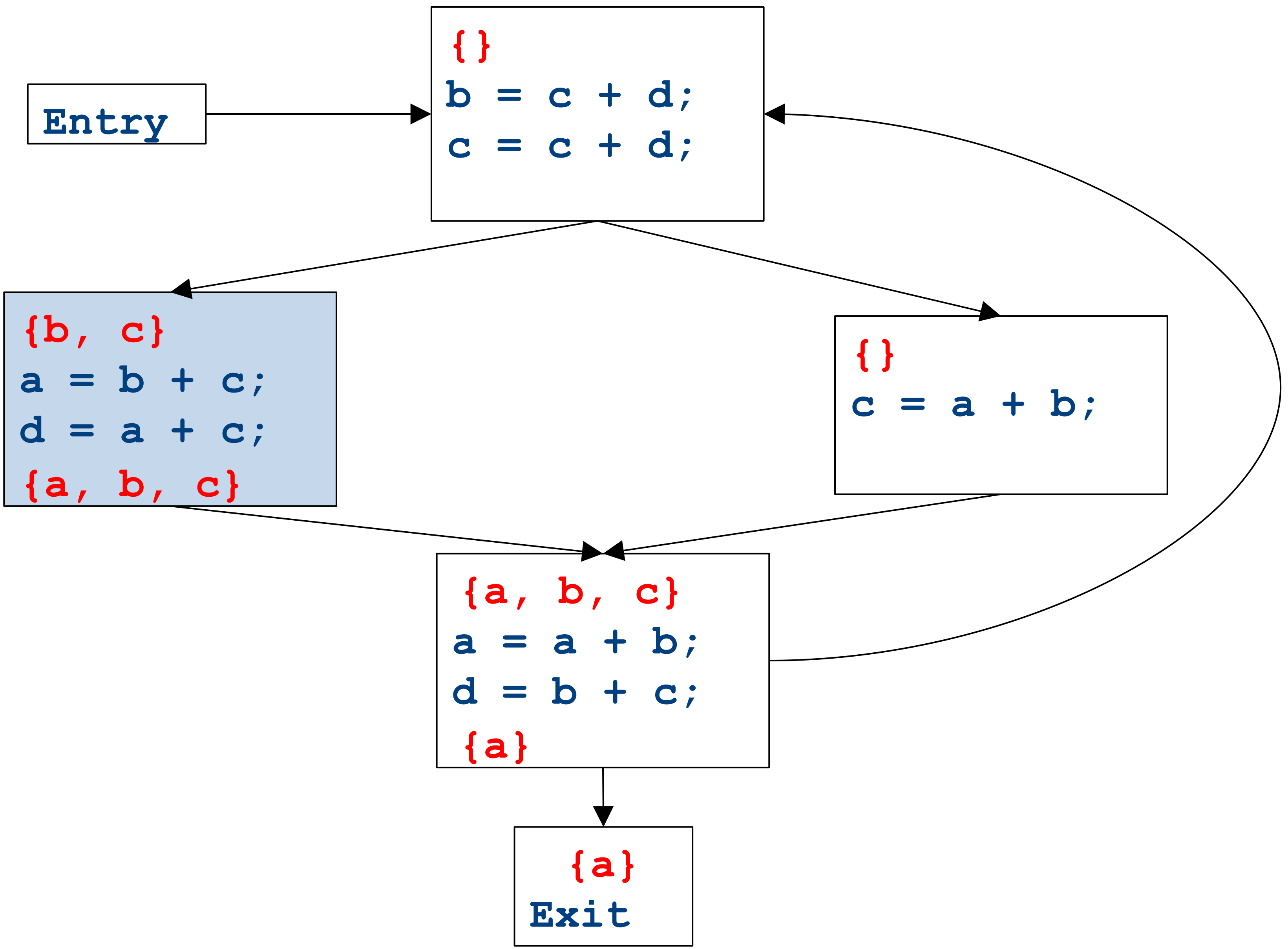
CFGs with loops - iteration



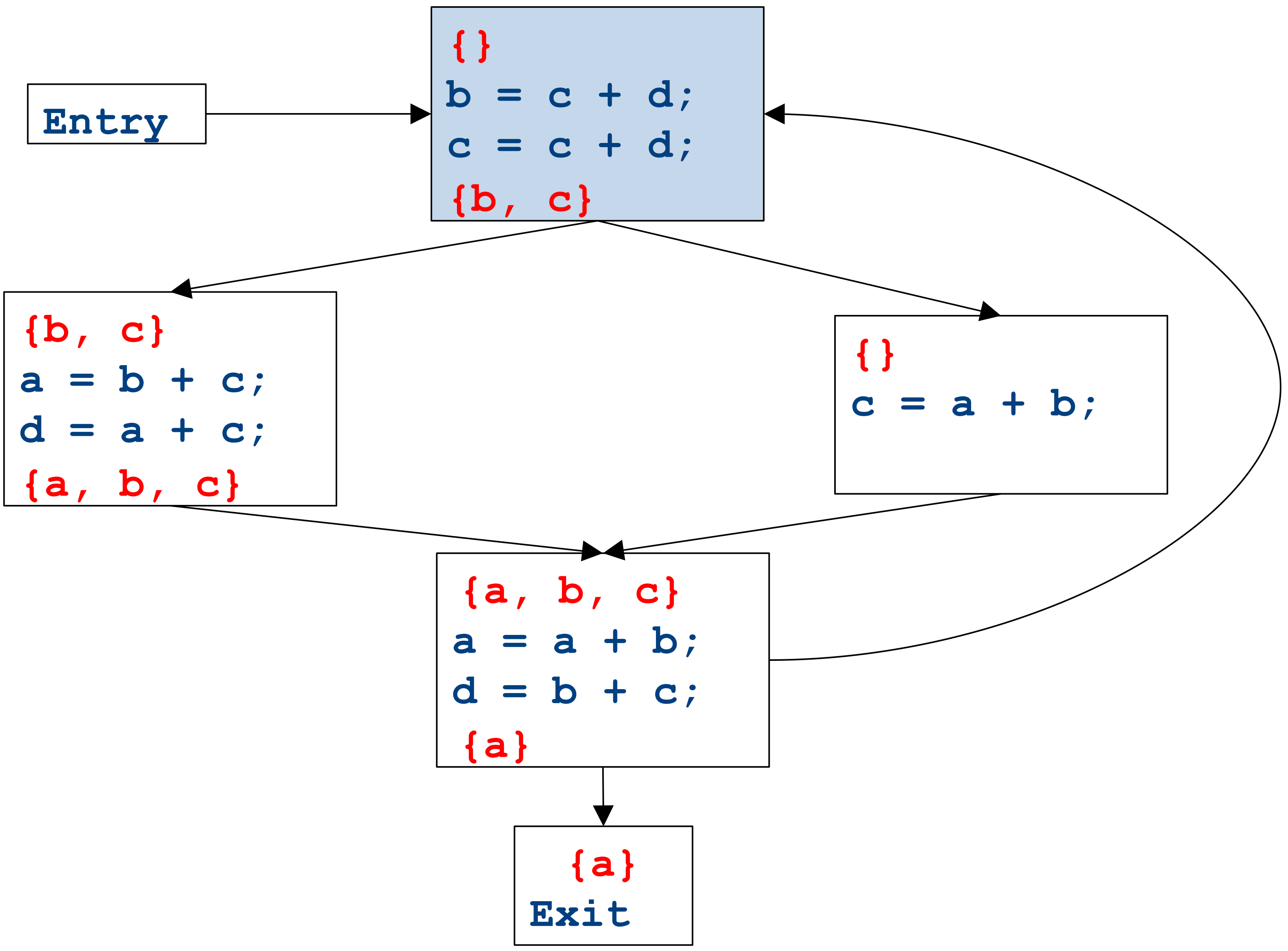
CFGs with loops - iteration



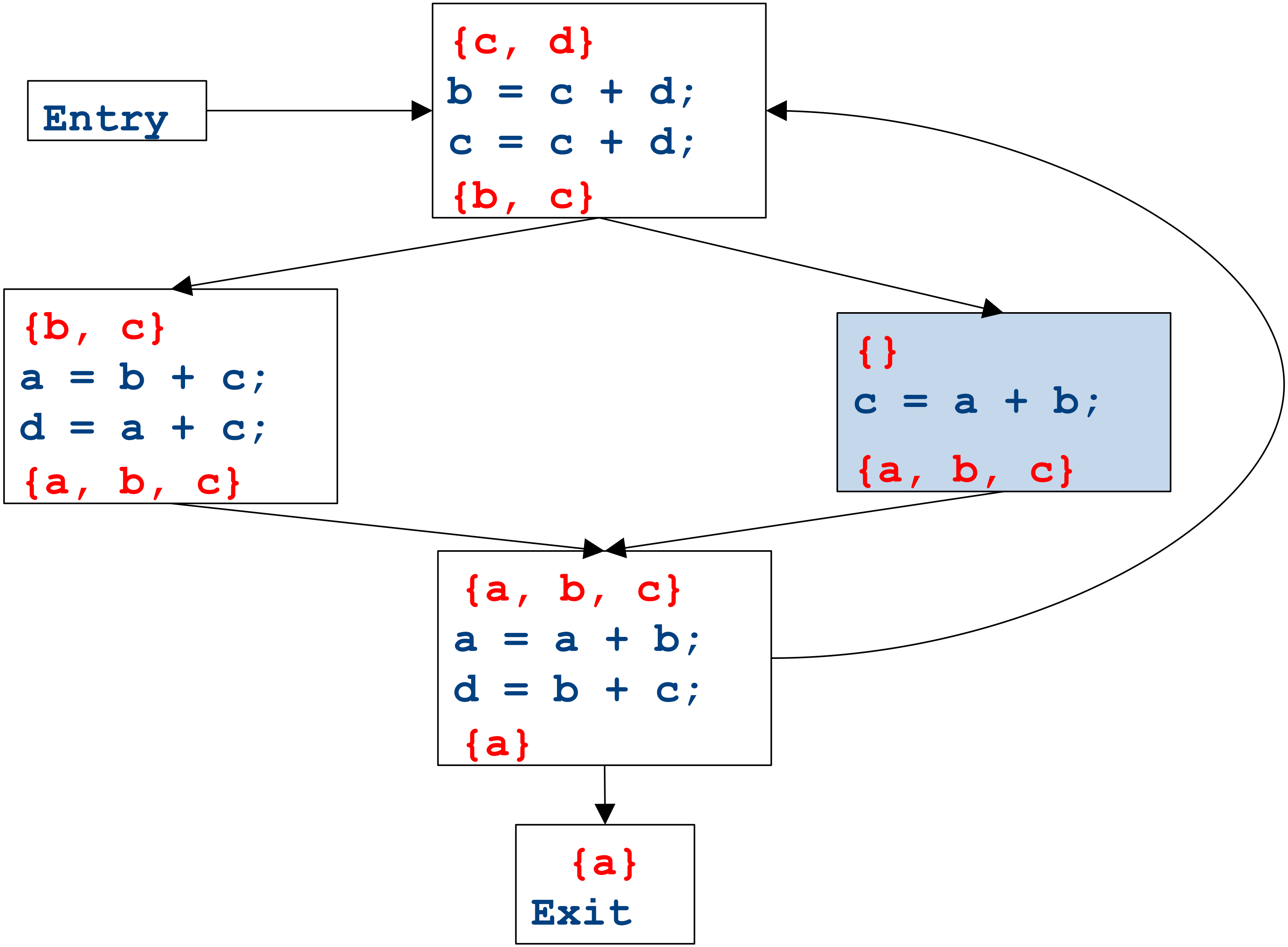
CFGs with loops - iteration



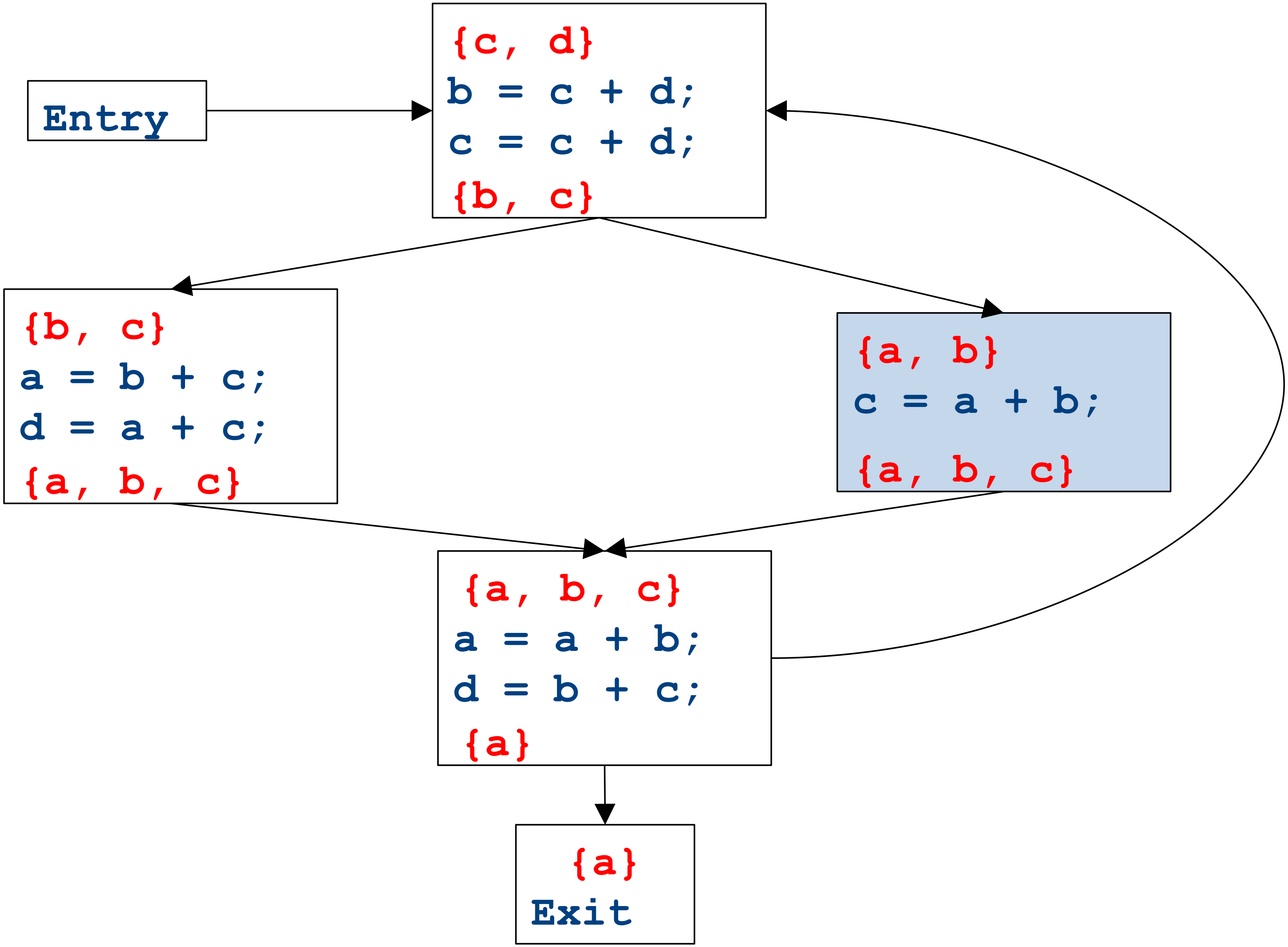
CFGs with loops - iteration



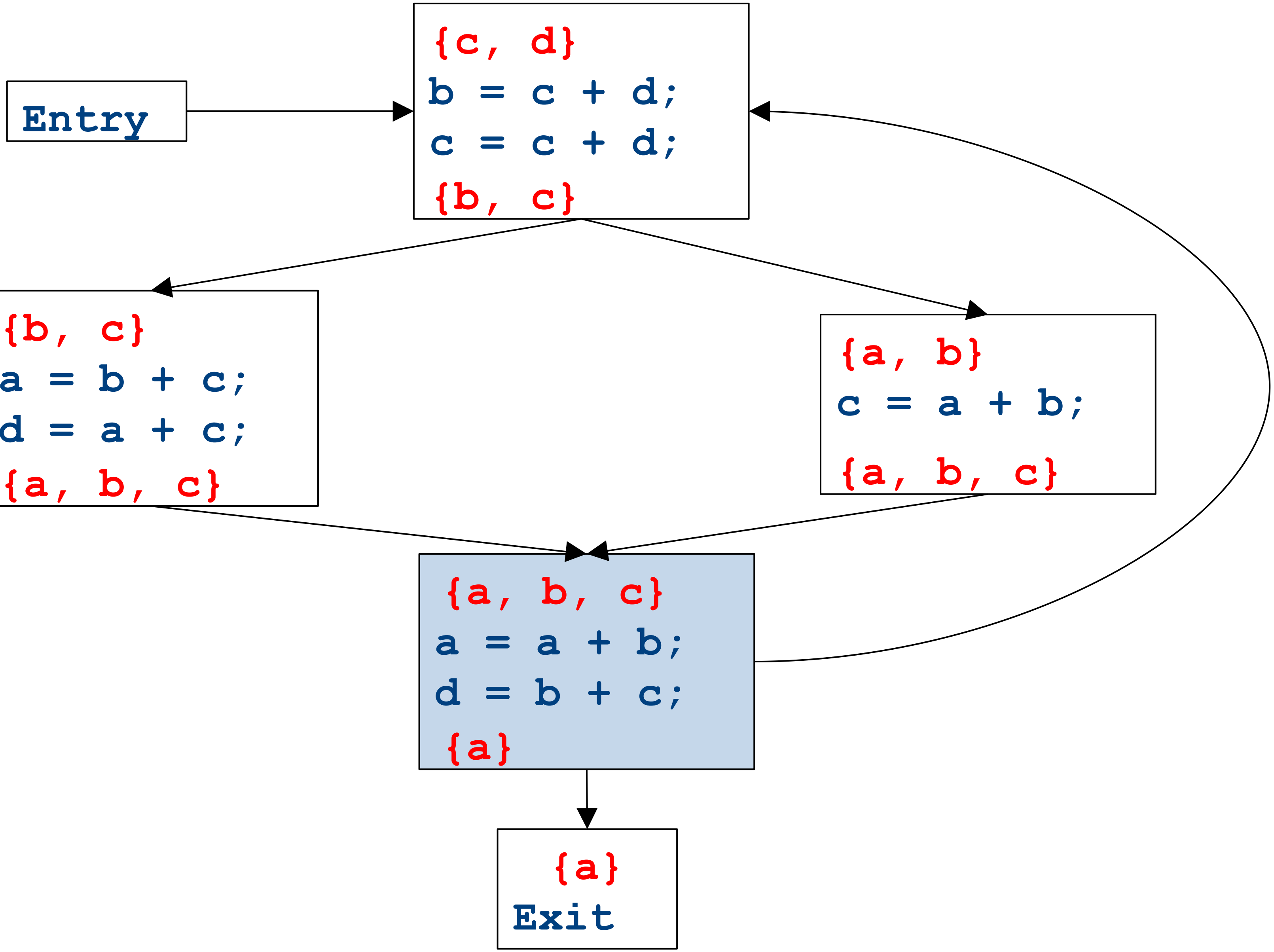
CFGs with loops - iteration



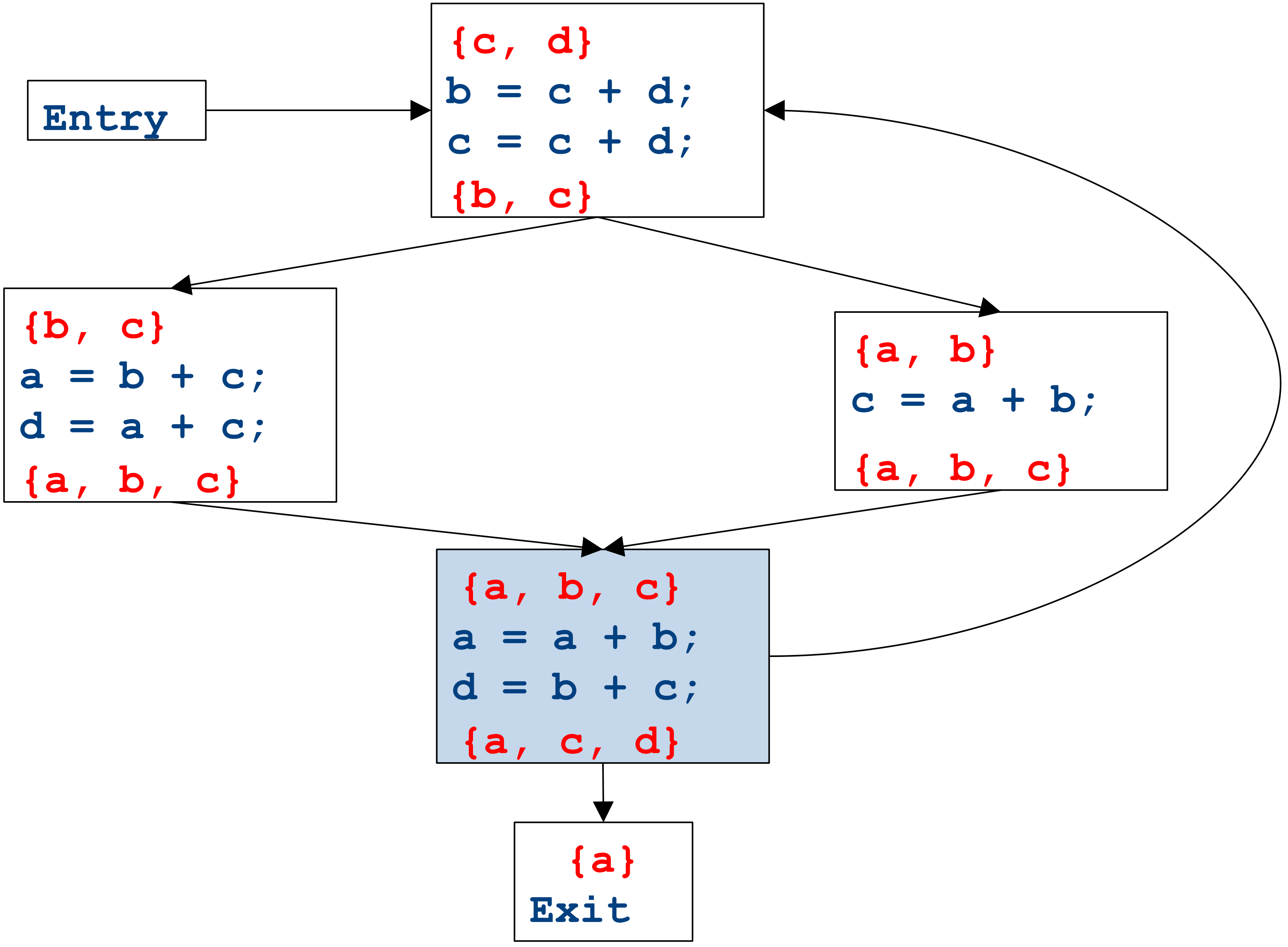
CFGs with loops - iteration



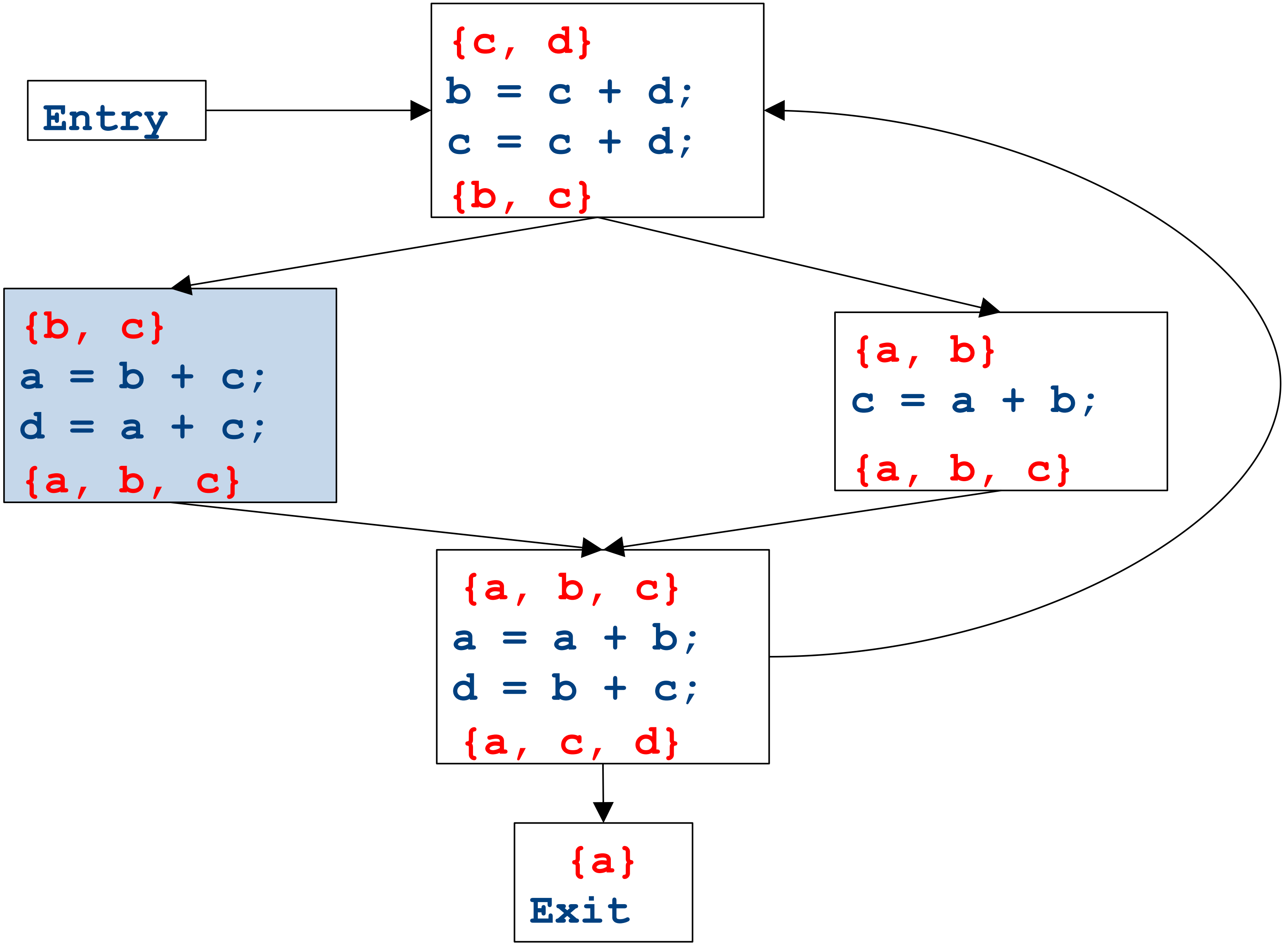
CFGs with loops - iteration



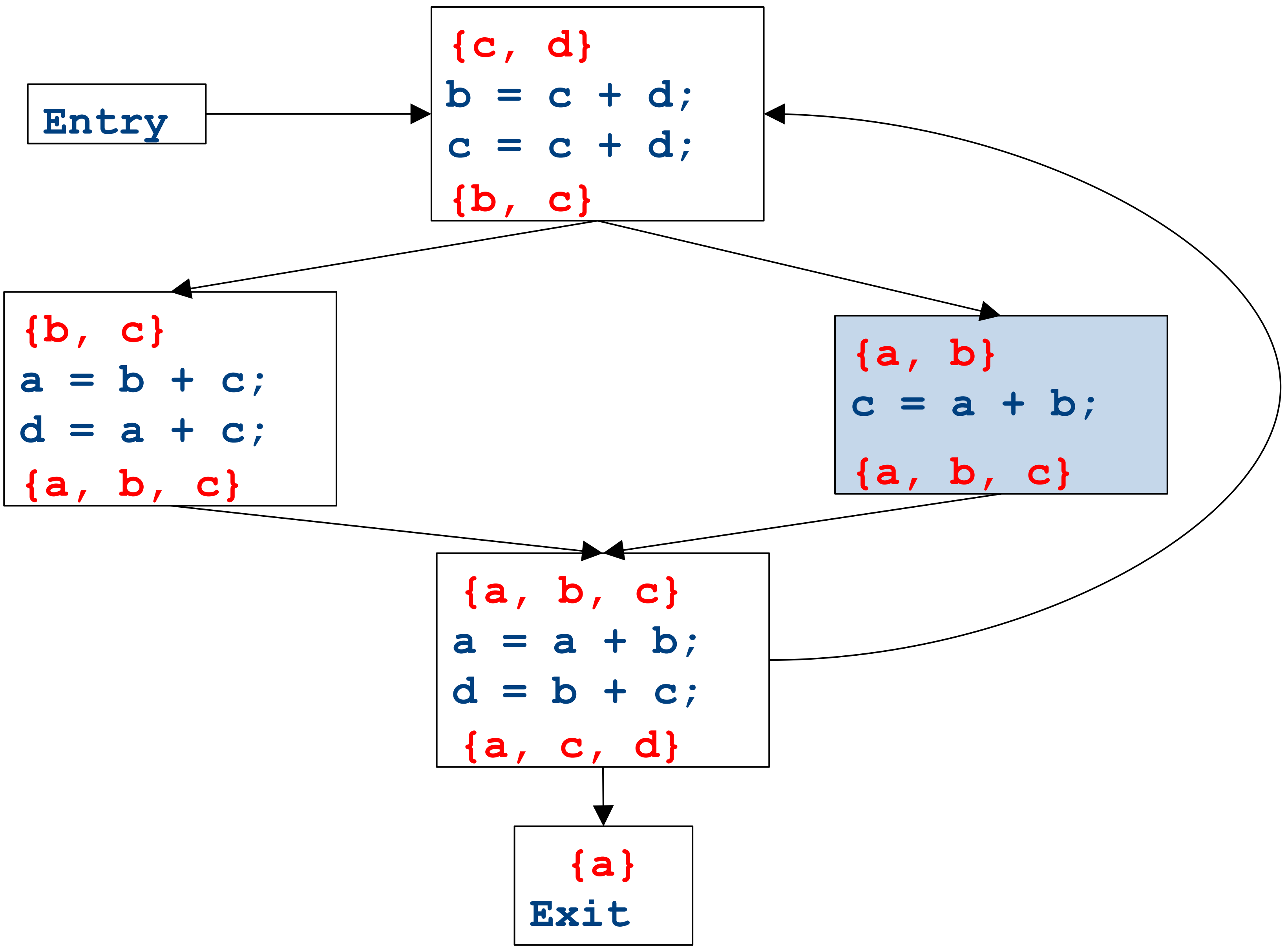
CFGs with loops - iteration



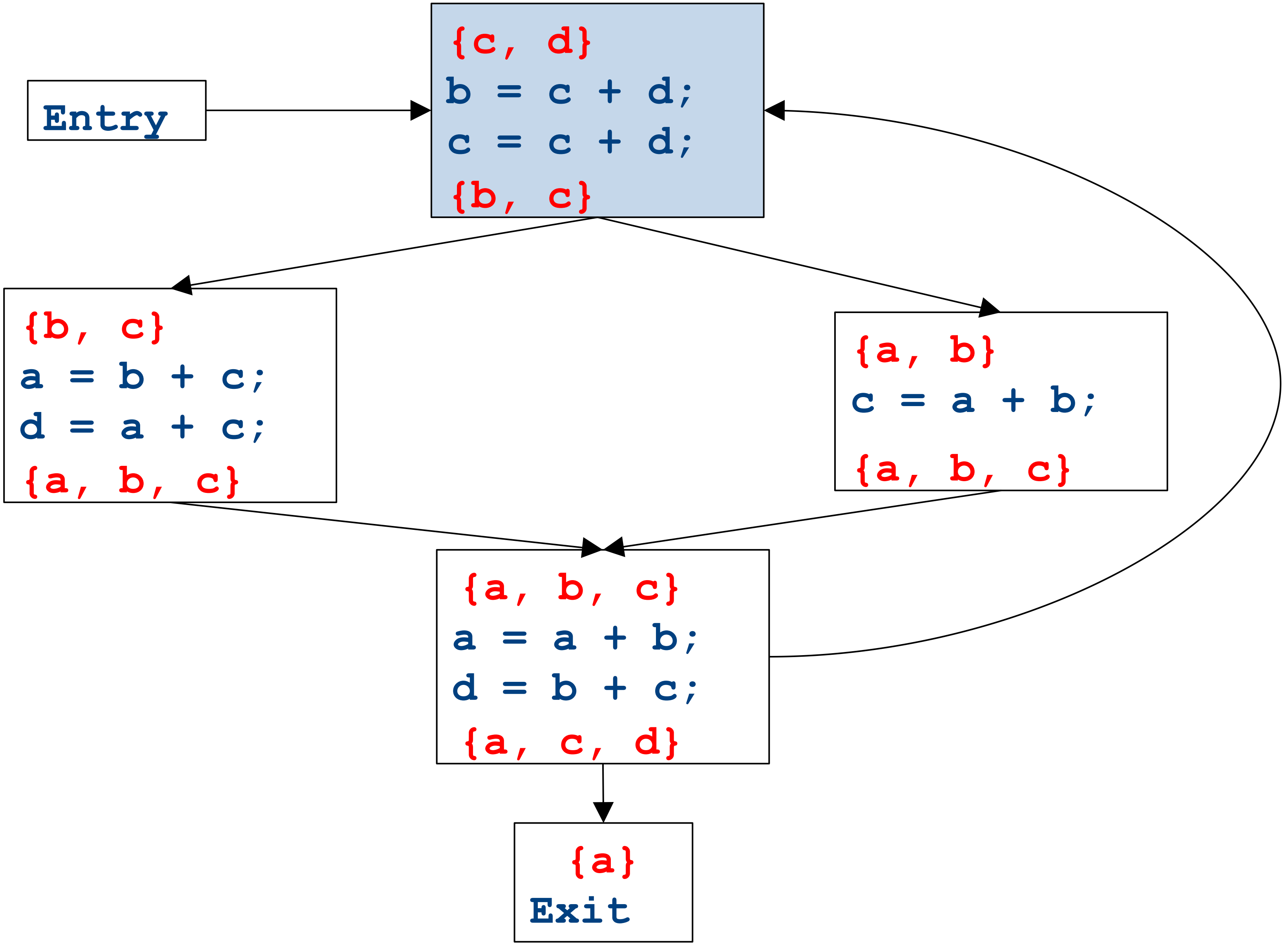
CFGs with loops - iteration



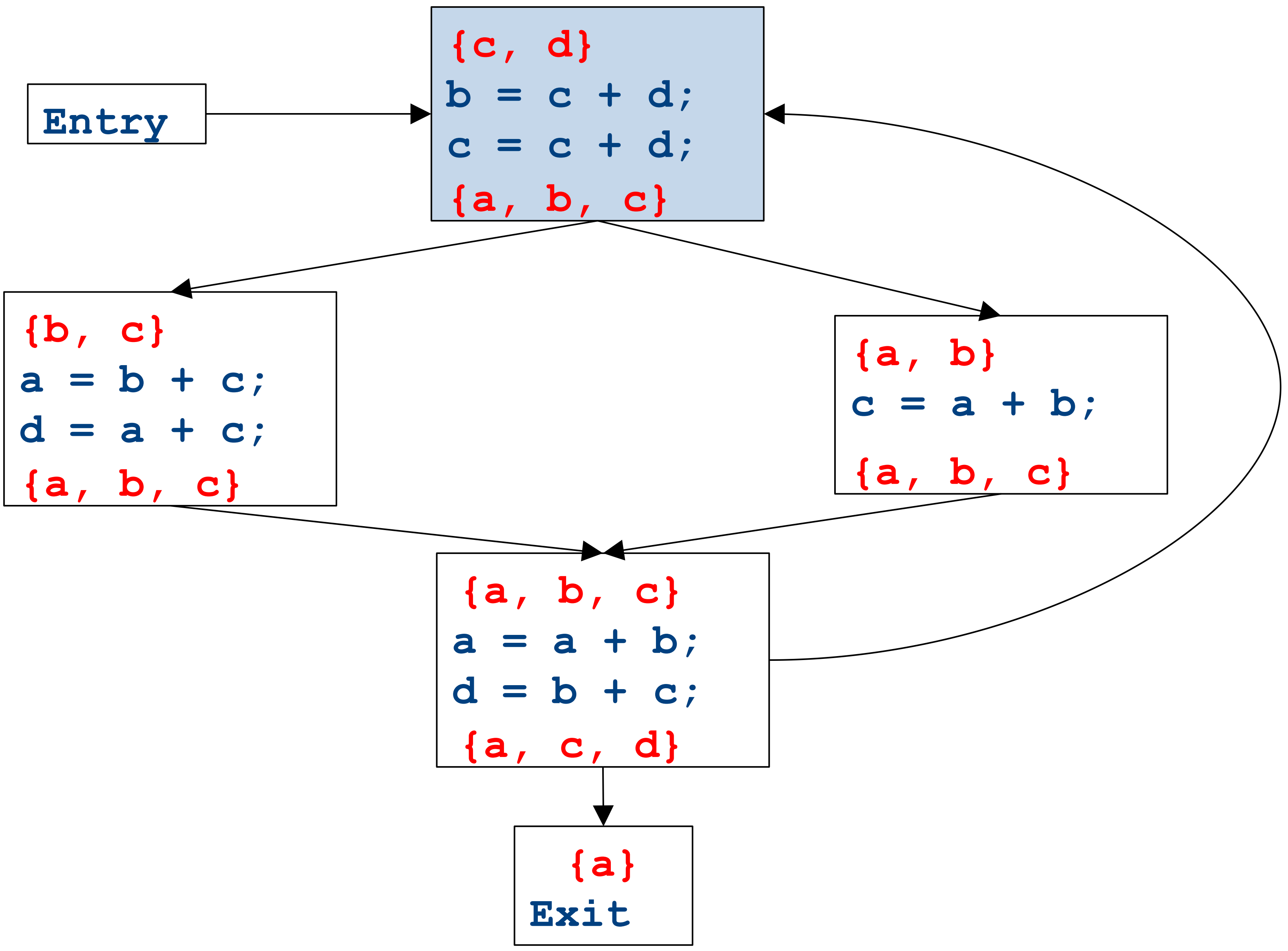
CFGs with loops - iteration



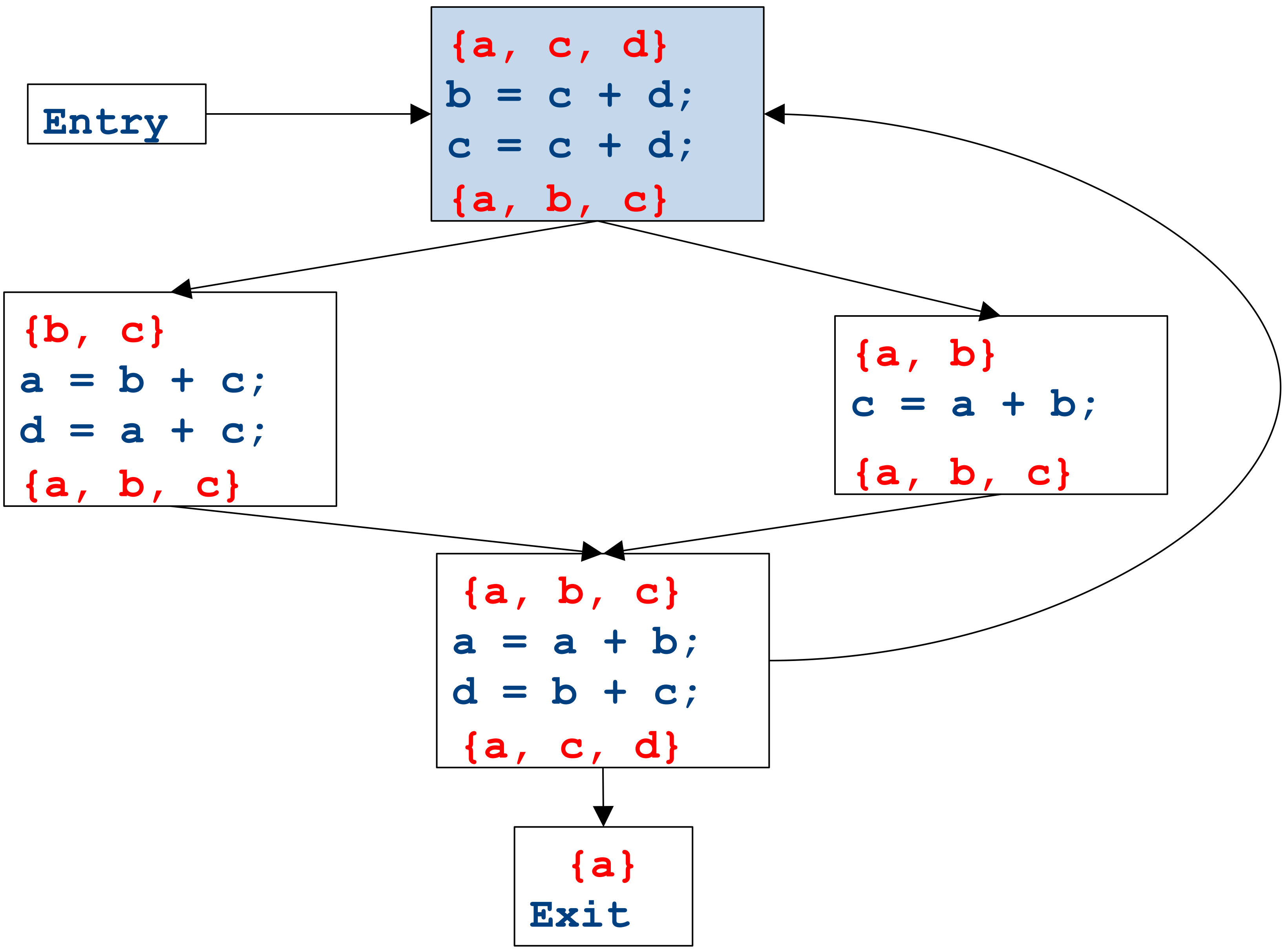
CFGs with loops - iteration



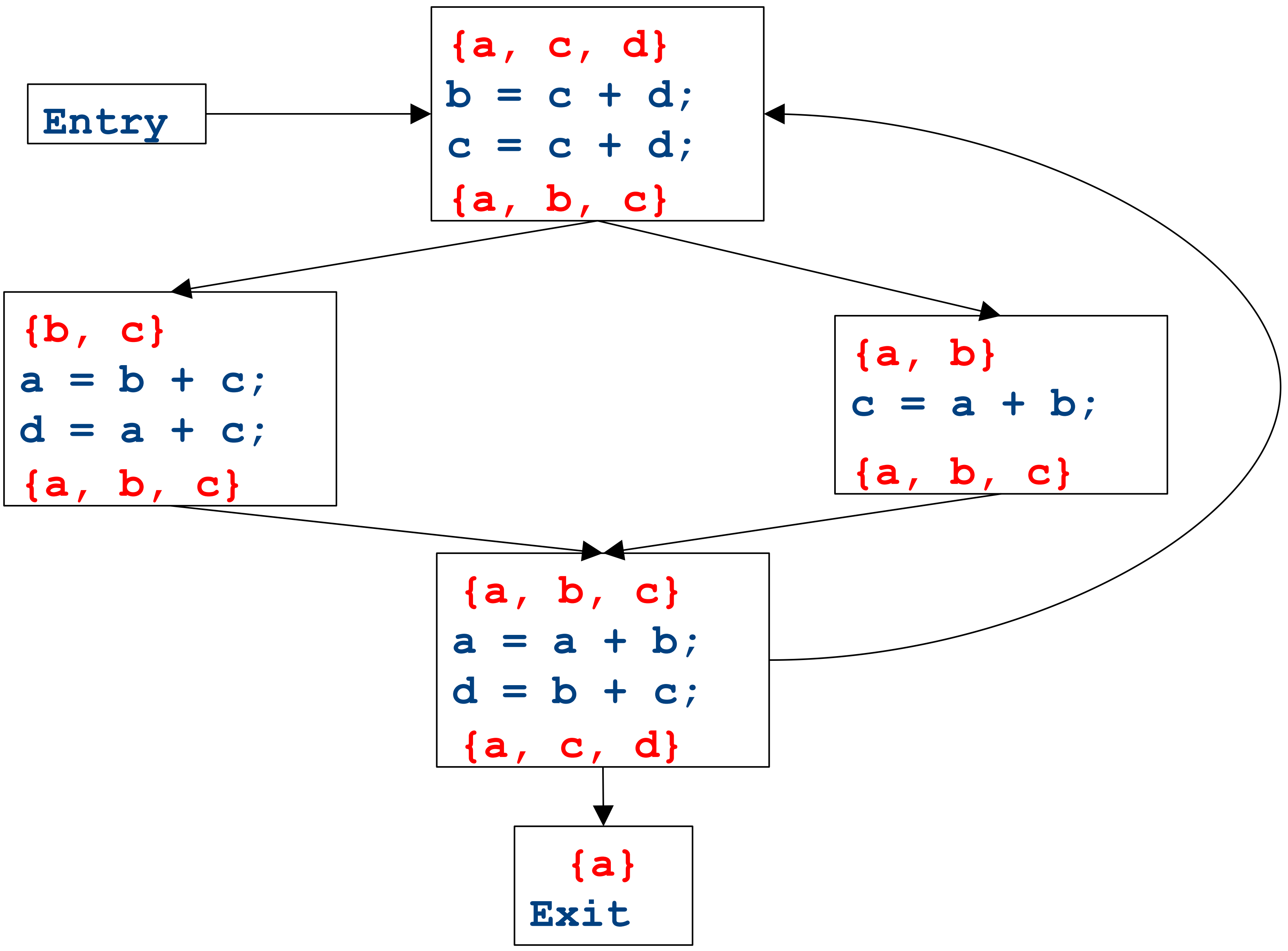
CFGs with loops - iteration



CFGs with loops - iteration



CFGs with loops - iteration



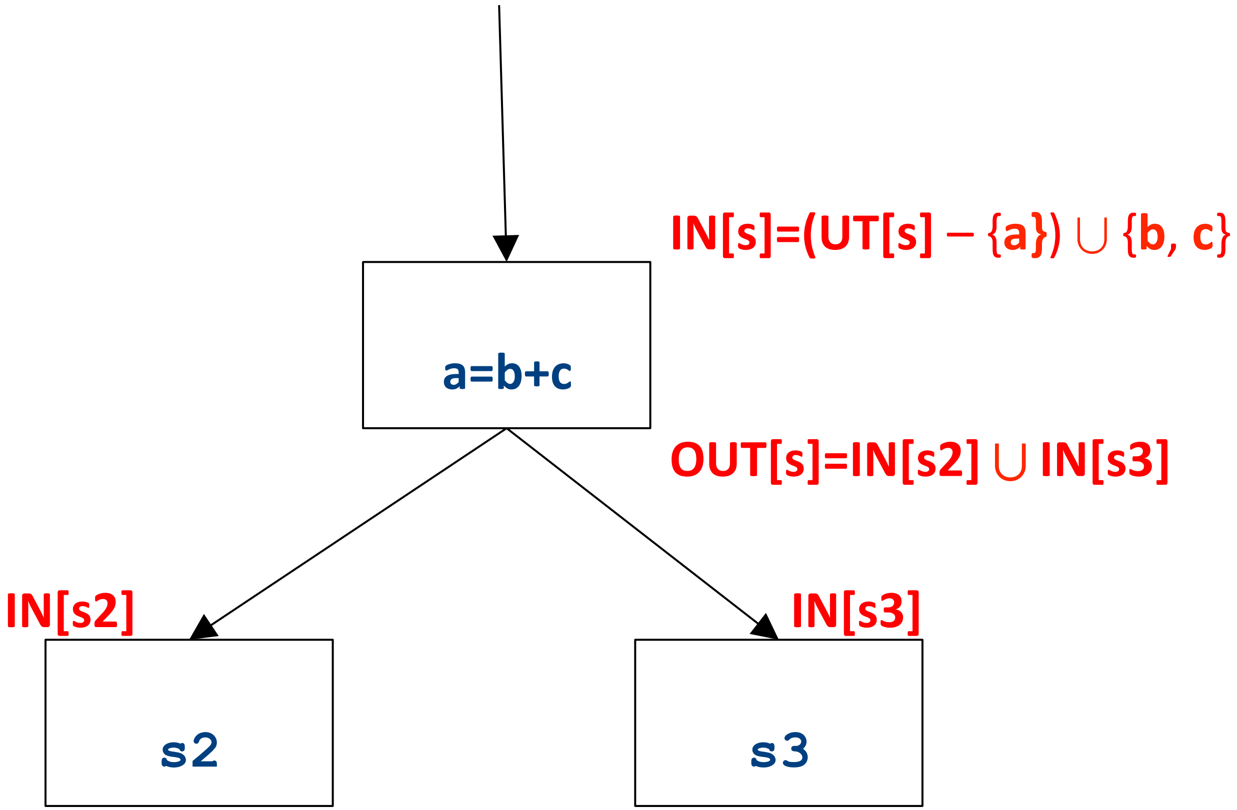
Summary of differences

- Need to be able to handle multiple predecessors/successors for a basic block
- Need to be able to handle multiple paths through the control-flow graph, and may need to iterate multiple times to compute the final value
  - But the analysis still needs to terminate!
- Need to be able to assign each basic block a reasonable default value for before we've analyzed it

Global liveness analysis

- Initially, set IN[**s**] = { } for each statement **s**
- Set IN[**exit**] to the set of variables known to be live on exit (language-specific knowledge)
- Repeat until no changes occur:
  - For each statement **s** of the form **a = b + c**, in any order you'd like:
    - Set OUT[**s**] to set union of IN[**p**] for each successor **p** of **s**
    - Set IN[**s**] to (OUT[**s**] – **a**) ∪ {**b**, **c**}.
- Yet another fixed-point iteration!

Global liveness analysis



Why does this work?

- To show correctness, we need to show that
  - The algorithm eventually terminates, and
  - When it terminates, it has a sound answer
- Termination argument:
  - Once a variable is discovered to be live during some point of the analysis, it always stays live
  - Only finitely many variables and finitely many places where a variable can become live
- Soundness argument (sketch):
  - Each individual rule, applied to some set, correctly updates liveness in that set
  - When computing the union of the set of live variables, a variable is only live if it was live on some path leaving the statement

Abstract Interpretation

- Theoretical foundations of program analysis

- Cousot and Cousot 1977

- Abstract meaning of programs
  - Executed at compile time

Another view of local optimization

- In local optimization, we want to reason about some property of the runtime behavior of the program
- Could we run the program and just watch what happens?
- **Idea:** Redefine the semantics of our programming language to give us information about our analysis

Properties of local analysis

- The only way to find out what a program will actually do is to run it
- Problems:
  - The program might not terminate
  - The program might have some behavior we didn't see when we ran it on a particular input
- However, this is not a problem inside a basic block
  - Basic blocks contain no loops
  - There is only one path through the basic block

Assigning new semantics

- Example: Available Expressions
- Redefine the statement **a = b + c to** mean "**a now holds the value of b + c,** and any variable holding the value **a is** now invalid"
- Run the program assuming these new semantics
- Treat the optimizer as an interpreter for these new semantics

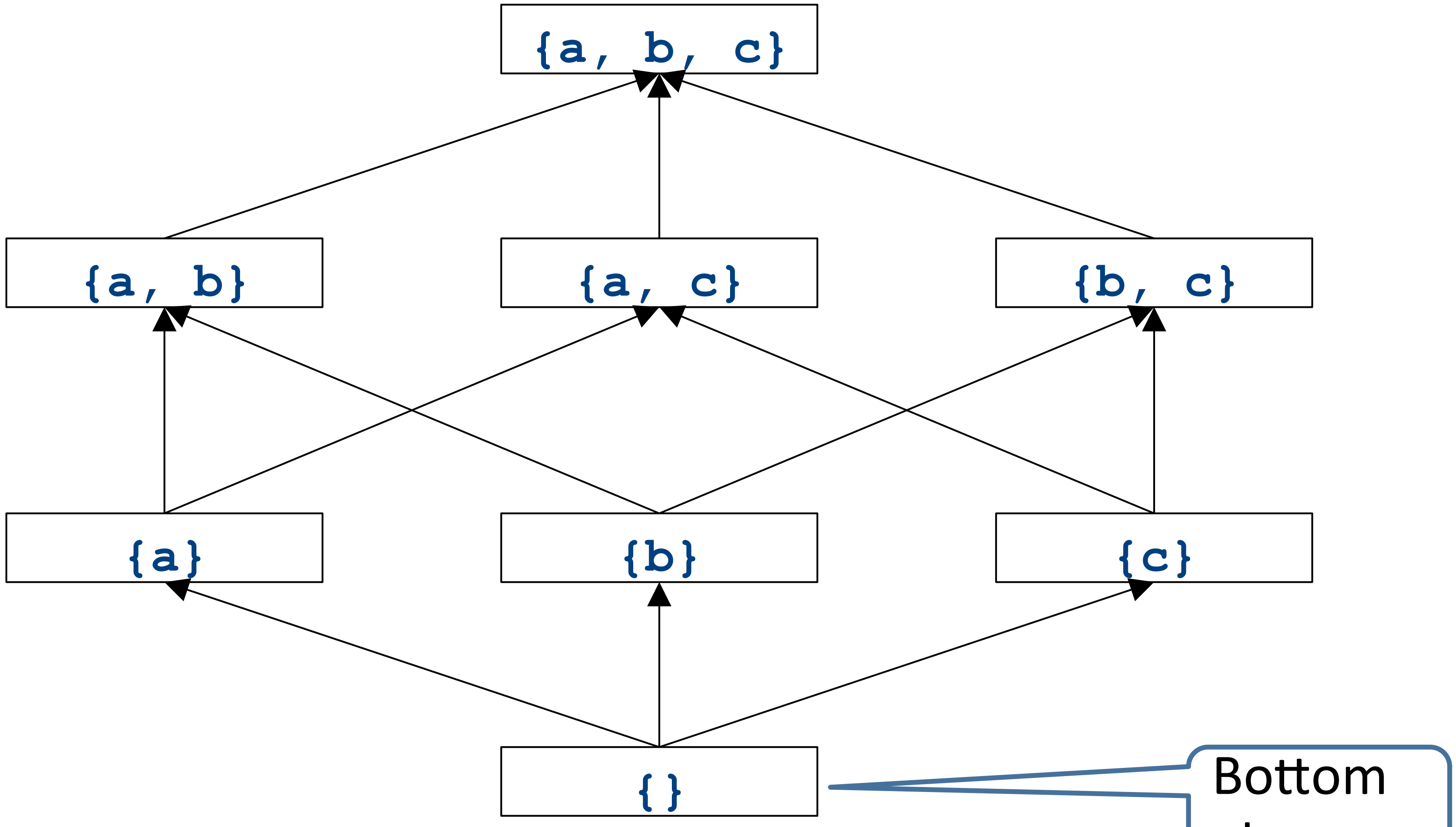
Theory to the rescue

- Building up all of the machinery to design this analysis was tricky
- The key ideas, however, are mostly independent of the analysis:
  - We need to be able to compute functions describing the behavior of each statement
  - We need to be able to merge several subcomputations together
  - We need an initial value for all of the basic blocks
- There is a beautiful formalism that captures many of these properties

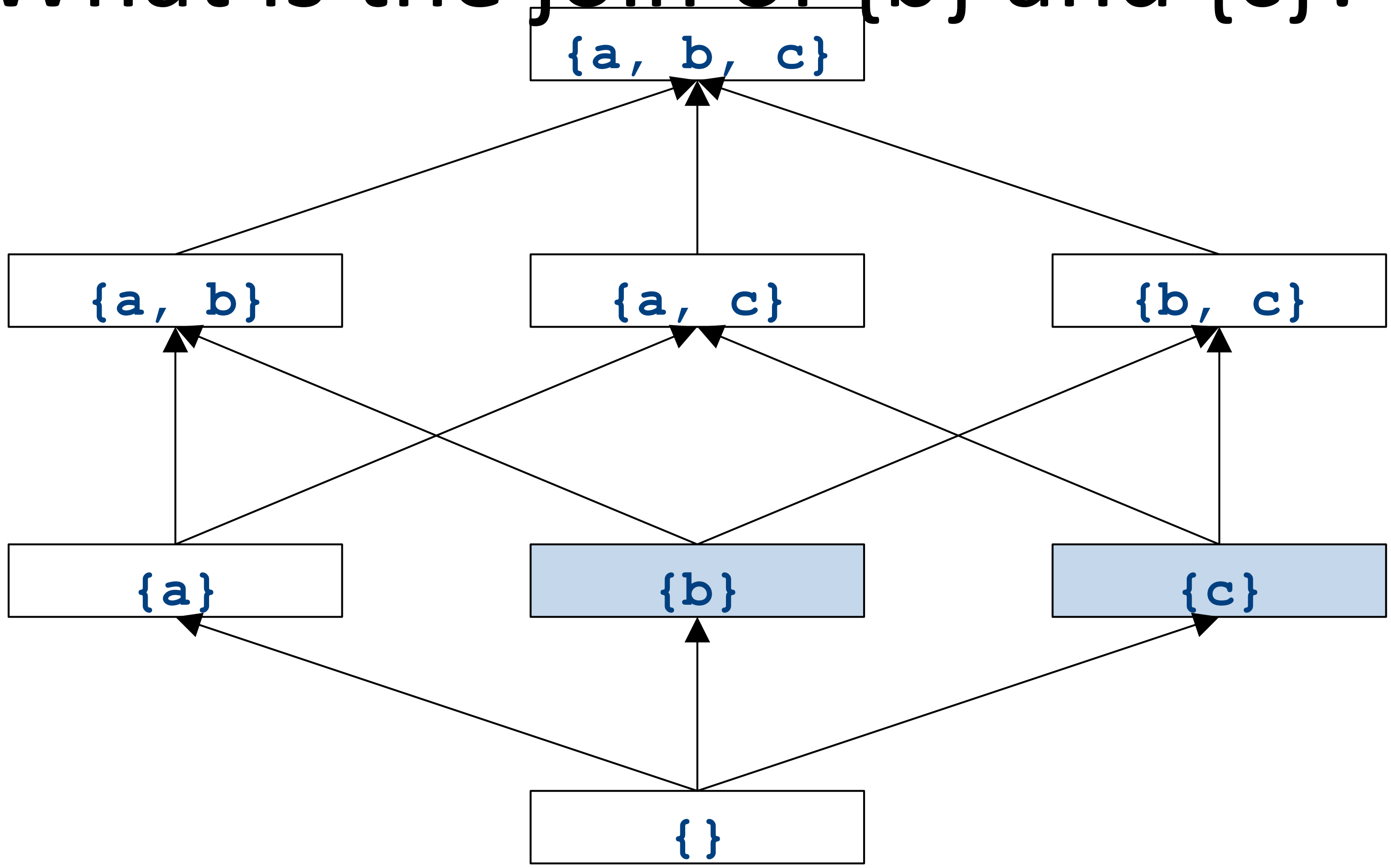
Join semilattices

- A join semilattice is a ordering defined on a set of elements
- Any two elements have some join that is the smallest element larger than both elements
- There is a unique bottom element, which is smaller than all other elements
- Intuitively:
  - The join of two elements represents combining information from two elements by an overapproximation
- The bottom element represents “no information yet” or “the least conservative possible answer”

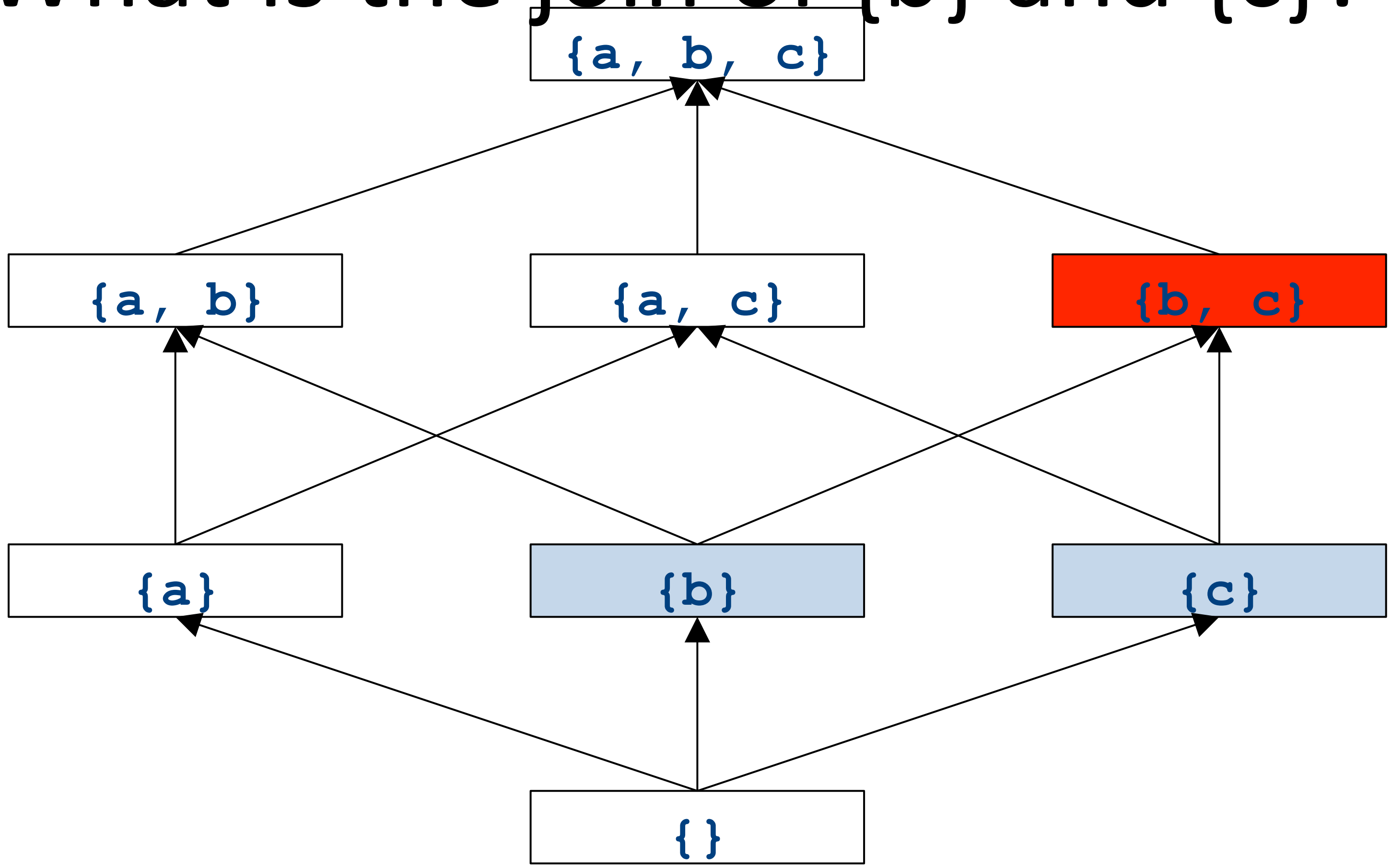
Join semilattice for liveness



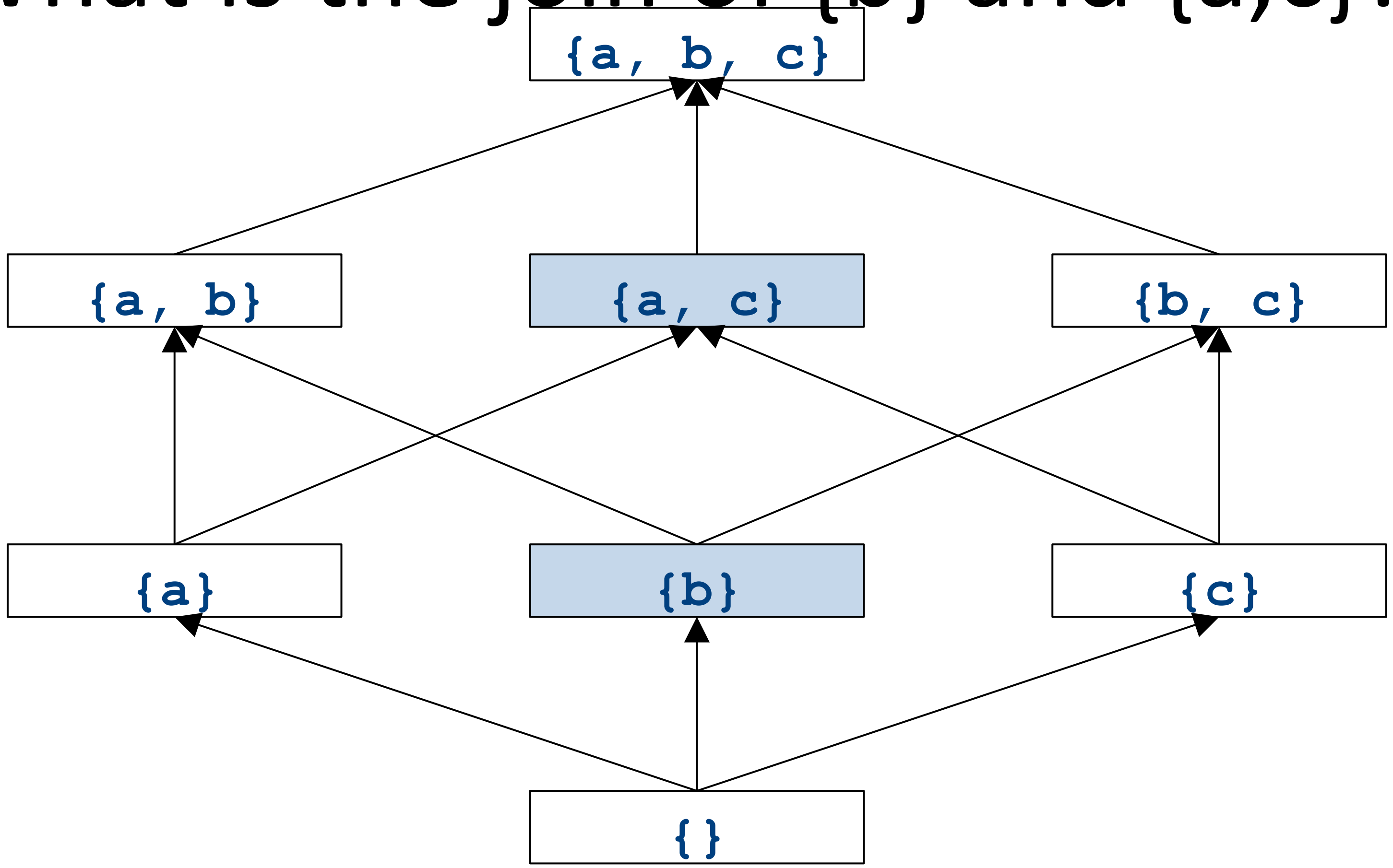
What is the join of {b} and {c}?



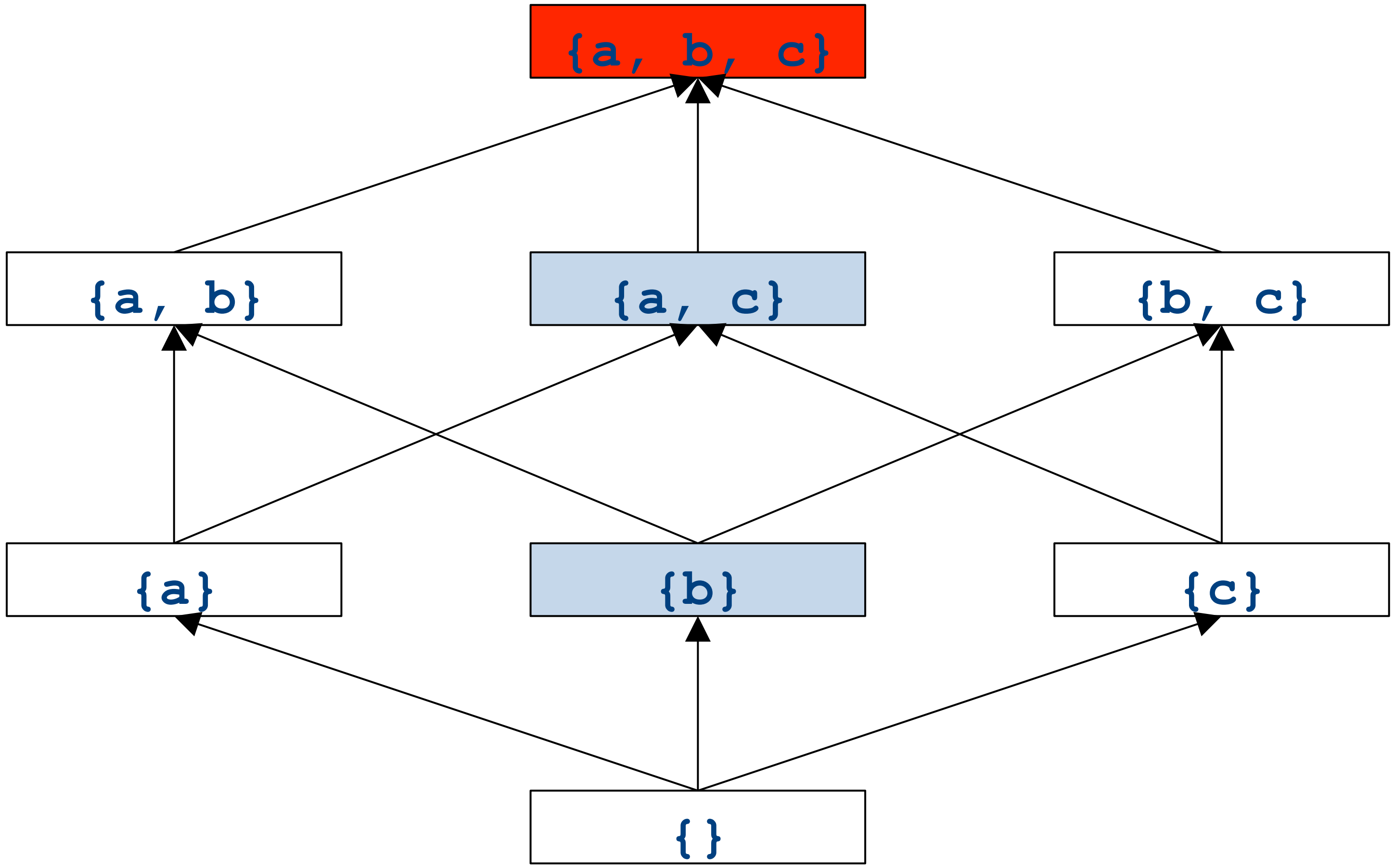
What is the join of {b} and {c}?



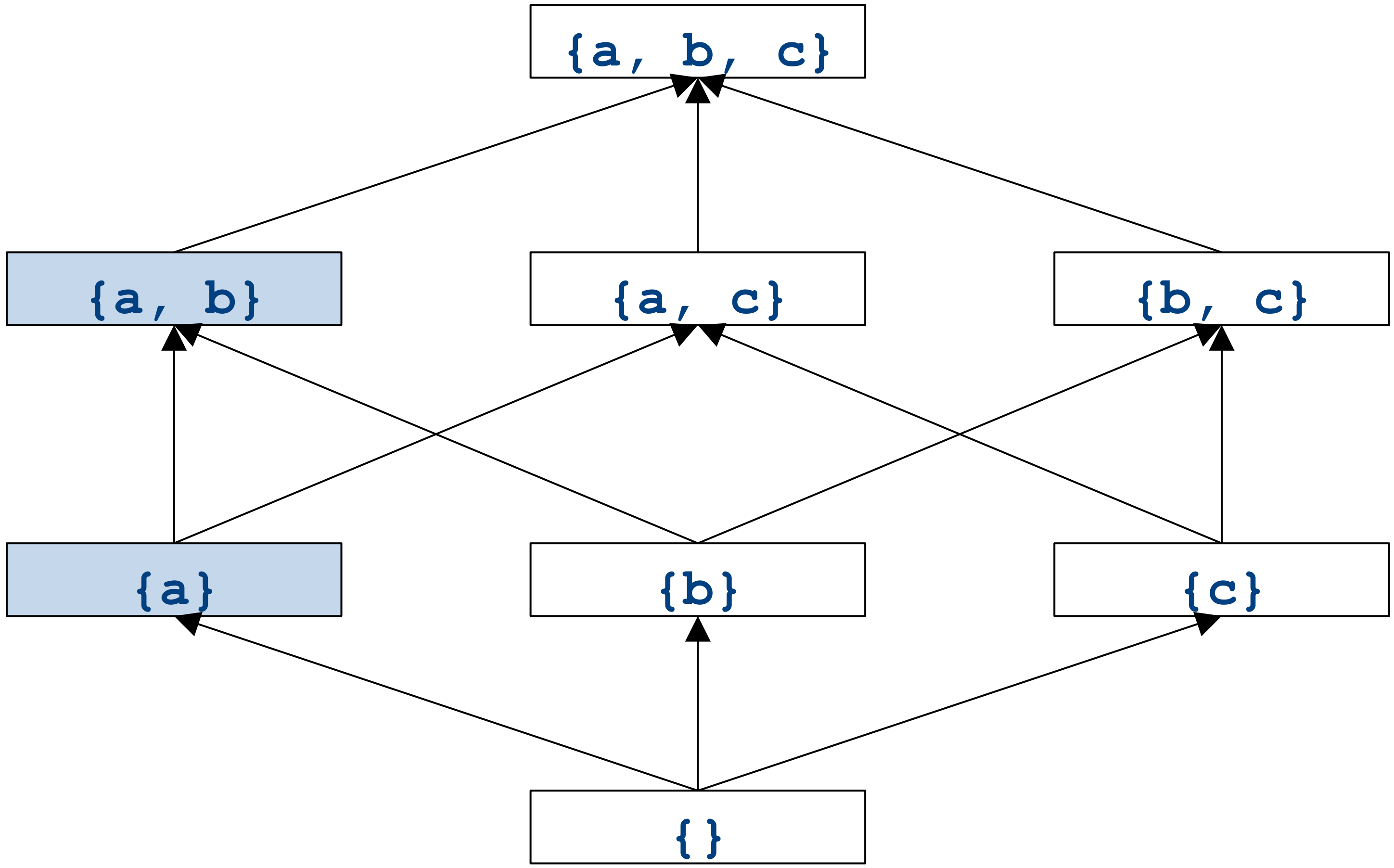
What is the join of {b} and {a,c}?



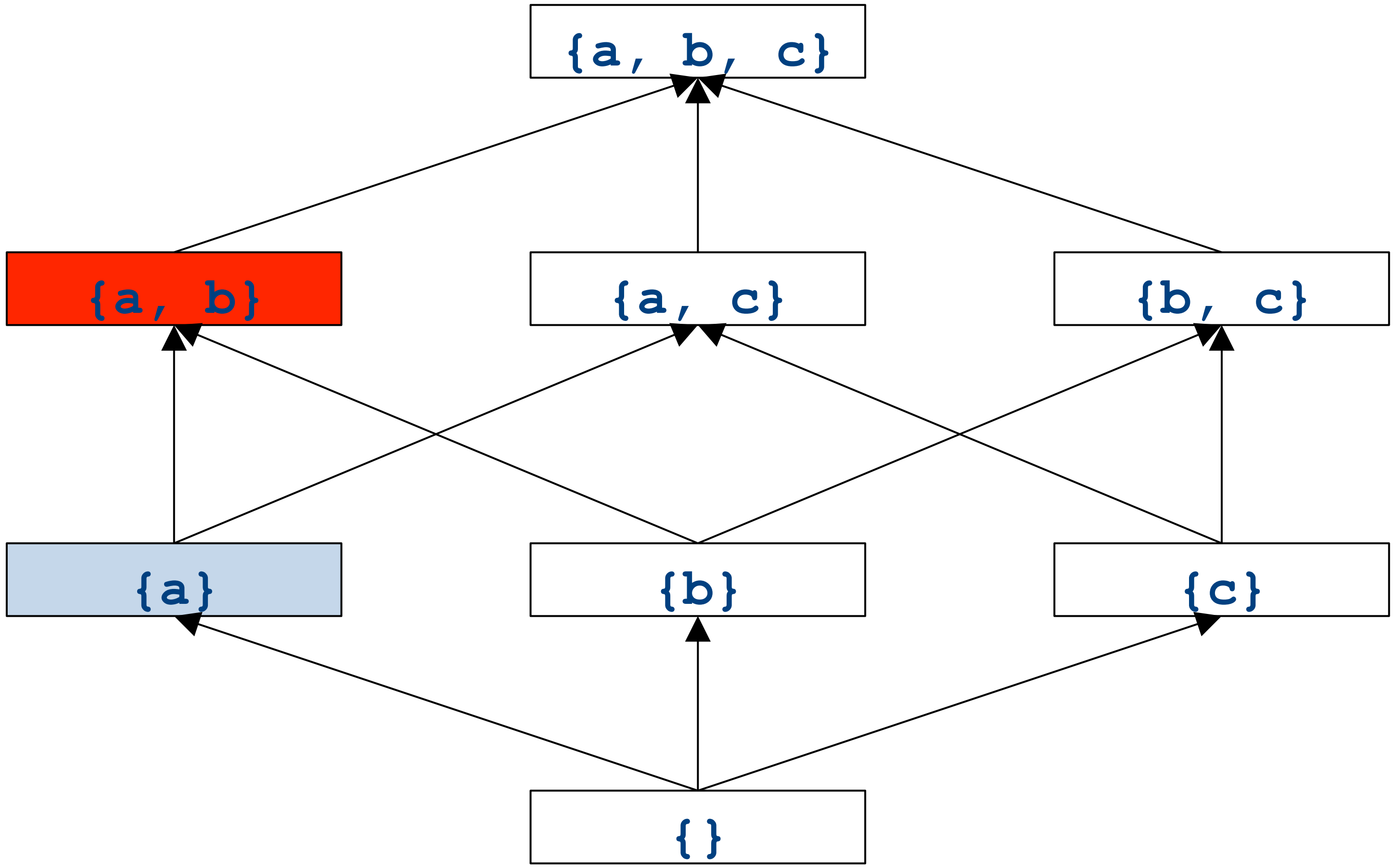
What is the join of {b} and {a,c}?



What is the join of {a} and {a,b}?



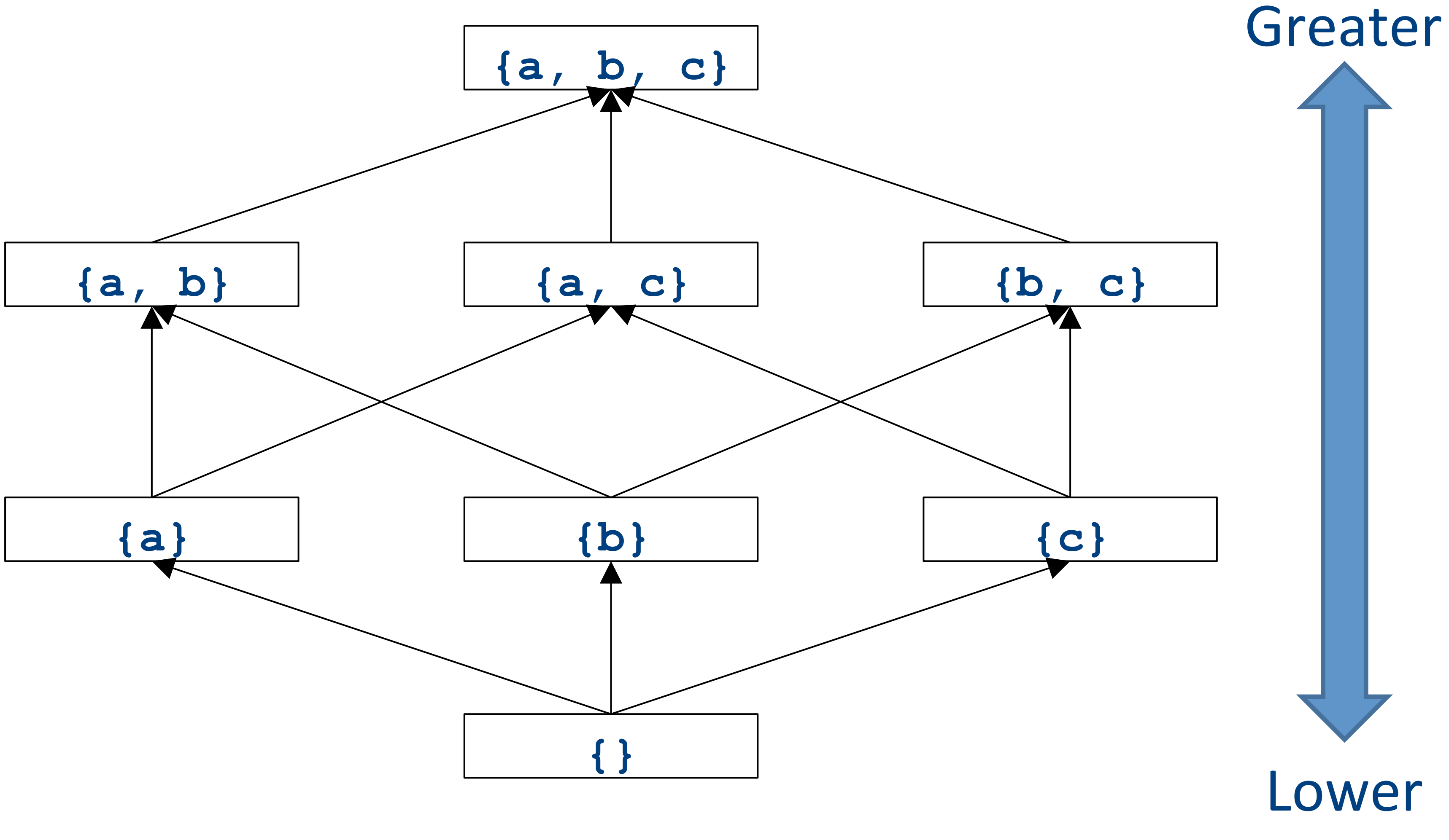
What is the join of {a} and {a,b}?



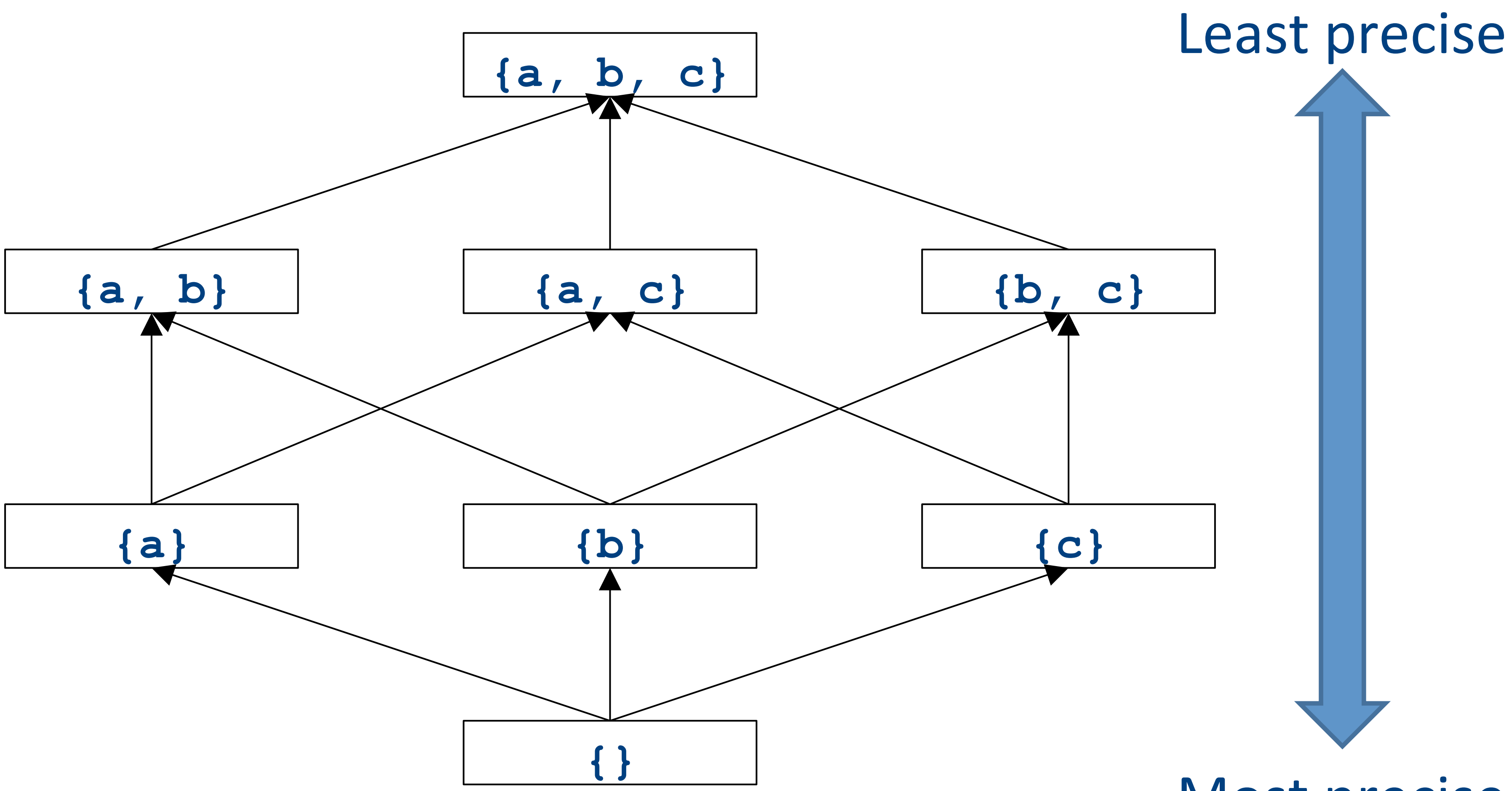
Formal definitions

- A join semilattice is a pair (V, $\sqcup$), where
- V is a domain of elements
- $\sqcup$ is a join operator that is
  - commutative: $x \sqcup y = y \sqcup x$
  - associative: $(x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$
  - idempotent: $x \sqcup x = x$
- If $x \sqcup y = z$, we say that z is the join or (**l**east **u**pper **b**ound) of x and y
- Every join semilattice has a bottom element denoted $\bot$ such that $\bot \sqcup x = x$ for all x

Join semilattices and ordering



Join semilattices and ordering



Join semilattices and orderings

- Every join semilattice (V, $\sqcup$) induces an ordering relationship $\sqsubseteq$ over its elements
- Define $x \sqsubseteq y$ iff $x \sqcup y = y$
- Need to prove
  - Reflexivity: $x \sqsubseteq x$
  - Antisymmetry: If $x \sqsubseteq y$ and $y \sqsubseteq x$, then $x = y$
  - Transitivity: If $x \sqsubseteq y$ and $y \sqsubseteq z$, then $x \sqsubseteq z$

An example join semilattice

- The set of natural numbers and the **max** function
- Idempotent
  - **max**{a, a} = a
- Commutative
  - **max**{a, b} = **max**{b, a}
- Associative
  - **max**{a, **max**{b, c}} = **max**{**max**{a, b}, c}
- Bottom element is 0:
  - **max**{0, a} = a
- What is the ordering over these elements?

A join semilattice for liveness

- Sets of live variables and the set union operation
- Idempotent:
  - $x \cup x = x$
- Commutative:
  - $x \cup y = y \cup x$
- Associative:
  - $(x \cup y) \cup z = x \cup (y \cup z)$
- Bottom element:
  - The empty set: $\emptyset \cup x = x$
- What is the ordering over these elements?

Semilattices and program analysis

- Semilattices naturally solve many of the problems we encounter in global analysis
- How do we combine information from multiple basic blocks?

- What value do we give to basic blocks we haven't seen yet?

- How do we know that the algorithm always terminates?

Semilattices and program analysis

- Semilattices naturally solve many of the problems we encounter in global analysis
- How do we combine information from multiple basic blocks?
  - Take the join of all information from those blocks
- What value do we give to basic blocks we haven't seen yet?
  - Use the bottom element
- How do we know that the algorithm always terminates?
  - Actually, we still don't! More on that later

Semilattices and program analysis

- Semilattices naturally solve many of the problems we encounter in global analysis
- How do we combine information from multiple basic blocks?
  - Take the join of all information from those blocks
- What value do we give to basic blocks we haven't seen yet?
  - Use the bottom element
- How do we know that the algorithm always terminates?
  - Actually, we still don't! More on that later

A general framework

- A global analysis is a tuple (**D, V,** $\sqcup$**, F, I**), where
  - **D** is a direction (forward or backward)
    - The order to visit statements within a basic block, not the order in which to visit the basic blocks
  - **V** is a set of values
  - $\sqcup$ is a join operator over those values
  - **F** is a set of transfer functions $f : \mathbf{V} \rightarrow \mathbf{V}$
  - **I** is an initial value
- The only difference from local analysis is the introduction of the join operator

Running global analyses

- Assume that (**D, V, ⊔, F, I**) is a forward analysis
- Set OUT[**s**] = ⊥ for all statements **s**
- Set OUT[**entry**] = **I**
- Repeat until no values change:
  - For each statement **s** with predecessors $\mathbf{p_1}$, $\mathbf{p_2}$, … , $\mathbf{p_n}$:
    - Set IN[**s**] = OUT[$\mathbf{p_1}$] ⊔ OUT[$\mathbf{p_2}$] ⊔ … ⊔ OUT[$\mathbf{p_n}$]
    - Set OUT[**s**] = f_s (IN[**s**])
- The order of this iteration does not matter
  - This is sometimes called chaotic iteration

For comparison

- Set OUT[**s**] = ⊥ for all statements **s**
- Set OUT[**entry**] = **I**
- Repeat until no values change:
  - For each statement **s** with predecessors $\mathbf{p_1}, \mathbf{p_2}, \ldots, \mathbf{p_n}$:
    - Set IN[**s**] = OUT[$\mathbf{p_1}$] ⊔ OUT[$\mathbf{p_2}$] ⊔ … ⊔ OUT[$\mathbf{p_n}$]
    - Set OUT[**s**] = f_s (IN[**s**])

- Set IN[**s**] = {} for all statements **s**
- Set OUT[**exit**] = the set of variables known to be live on exit
- Repeat until no values change:
  - For each statement **s** of the form **a=b+c**:
    - Set OUT[**s**] = set union of IN[**x**] for each successor **x** of **s**
    - Set IN[**s**] = (OUT[**s**]-{a}) ∪ {b,c}

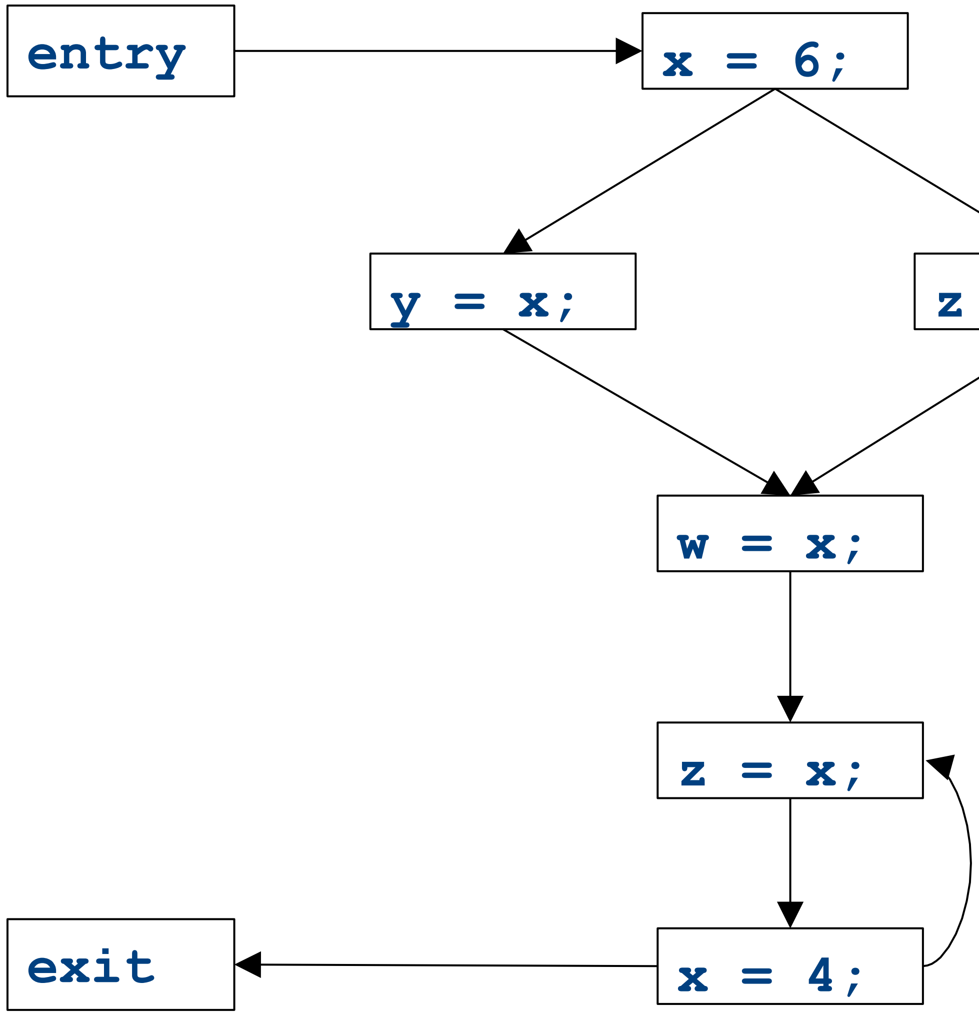
The dataflow framework

- This form of analysis is called the dataflow framework
- Can be used to easily prove an analysis is sound
- With certain restrictions, can be used to prove that an analysis eventually terminates
  - Again, more on that later

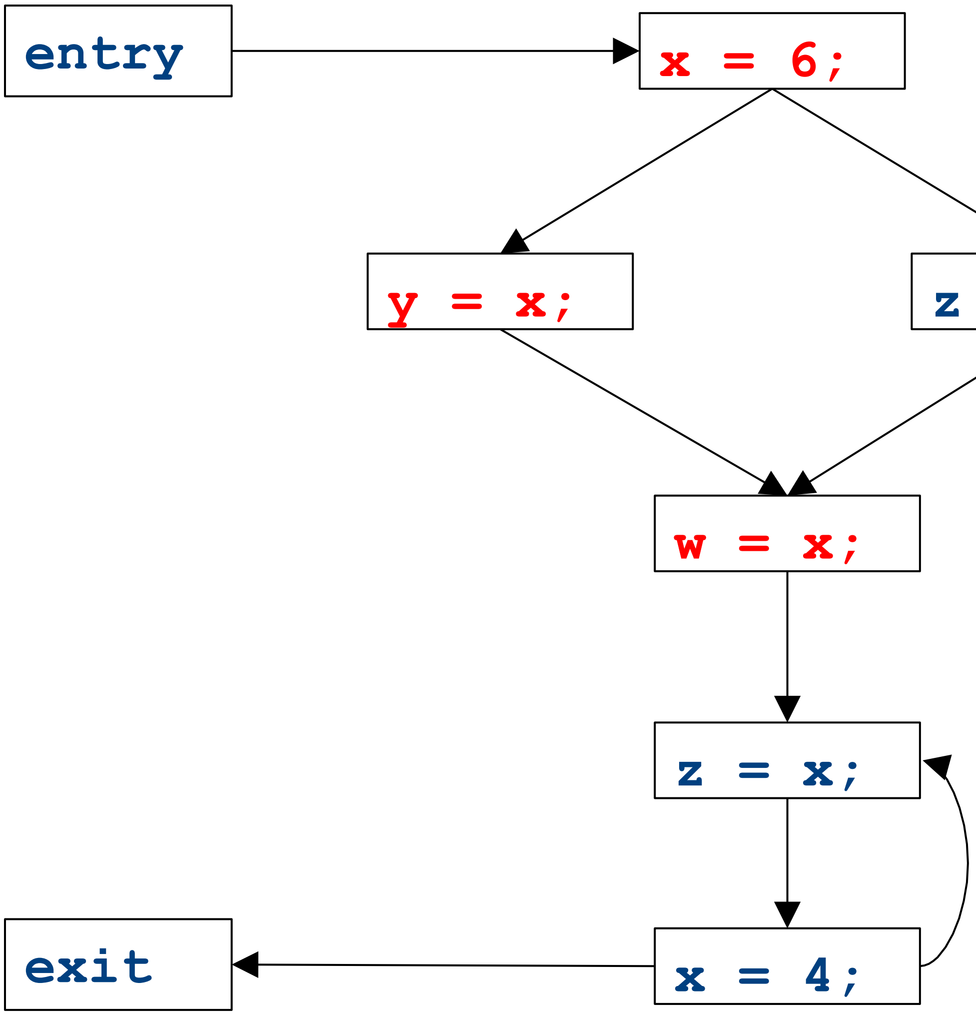
Global constant propagation

- Constant propagation is an optimization that replaces each variable that is known to be a constant value with that constant
- An elegant example of the dataflow framework

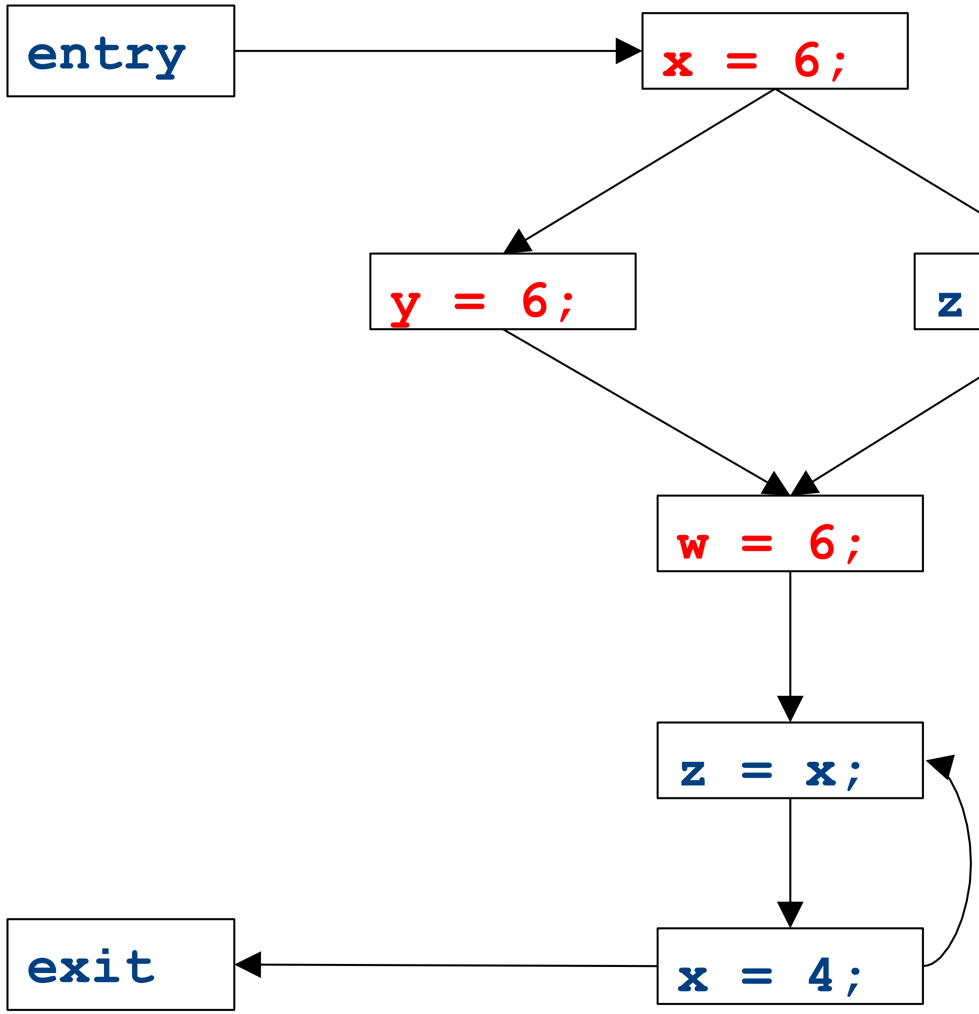
Global constant propagation



Global constant propagation



Global constant propagation



Constant propagation analysis

- In order to do a constant propagation, we need to track what values might be assigned to a variable at each program point
- Every variable will either
  - Never have a value assigned to it,
  - Have a single constant value assigned to it,
  - Have two or more constant values assigned to it, or
  - Have a known non-constant value.
  - Our analysis will propagate this information throughout a CFG to identify locations where a value is constant

Properties of constant propagation

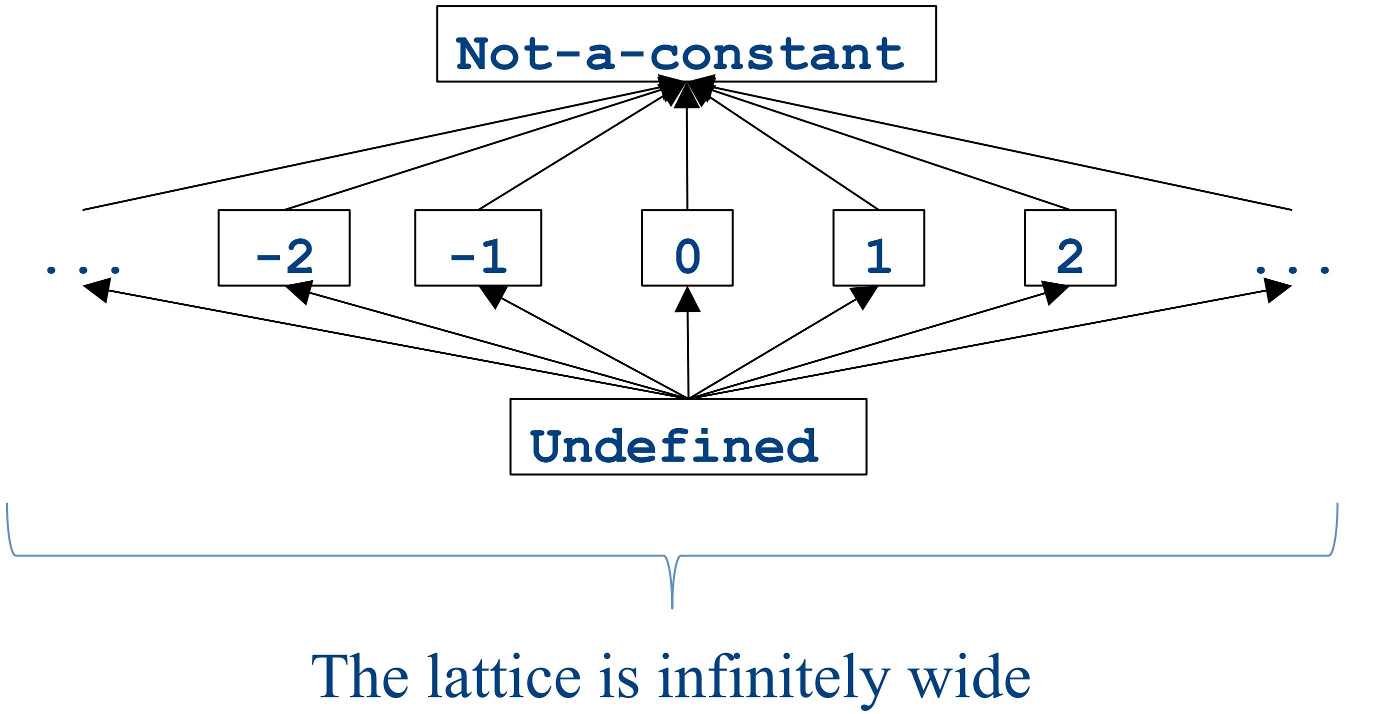
- For now, consider just some single variable **x**
- At each point in the program, we know one of three things about the value of **x**:
  - **x** is definitely not a constant, since it's been assigned two values or assigned a value that we know isn't a constant
  - **x** is definitely a constant and has value **k**
  - We have never seen a value for **x**
- Note that the first and last of these are **not** the same!
  - The first one means that there may be a way for **x** to have multiple values
  - The last one means that **x** never had a value at all

Defining a join operator

- The join of any two different constants is **Not-a-Constant**
  - (If the variable might have two different values on entry to a statement, it cannot be a constant)
- The join of **Not a Constant** and any other value is **Not-a-Constant**
  - (If on some path the value is known not to be a constant, then on entry to a statement its value can't possibly be a constant)
- The join of **Undefined** and any other value is that other value
  - (If **x** has no value on some path and does have a value on some other path, we can just pretend it always had the assigned value)

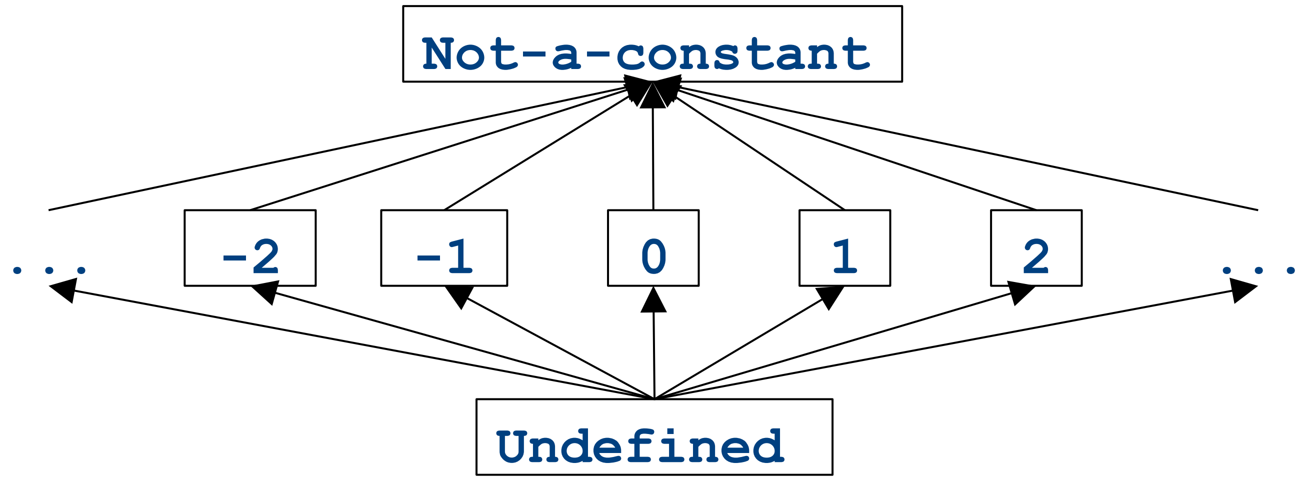
A semilattice for constant propagation

- One possible semilattice for this analysis is shown here (for each variable):



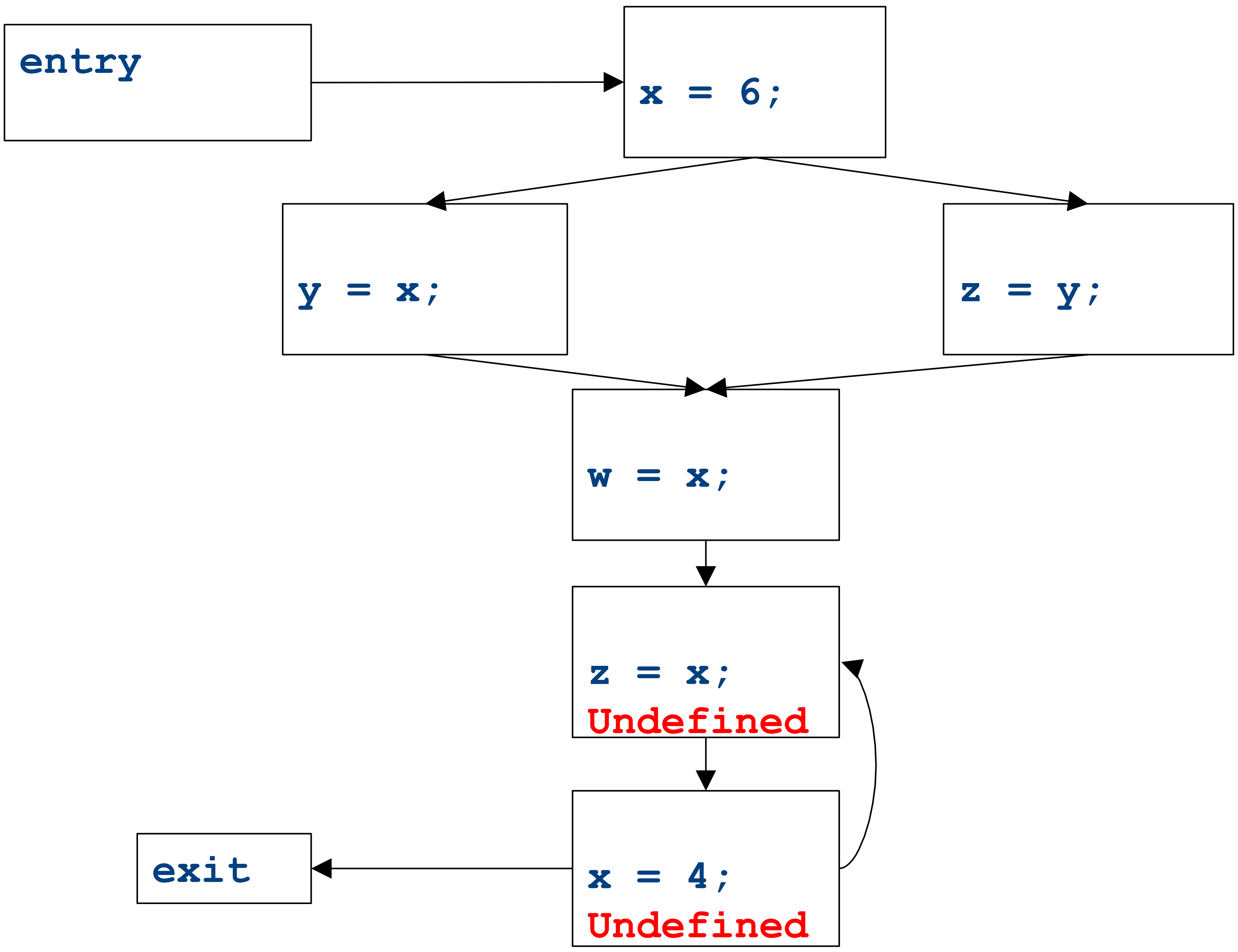
A semilattice for constant propagation

- One possible semilattice for this analysis is shown here (for each variable):

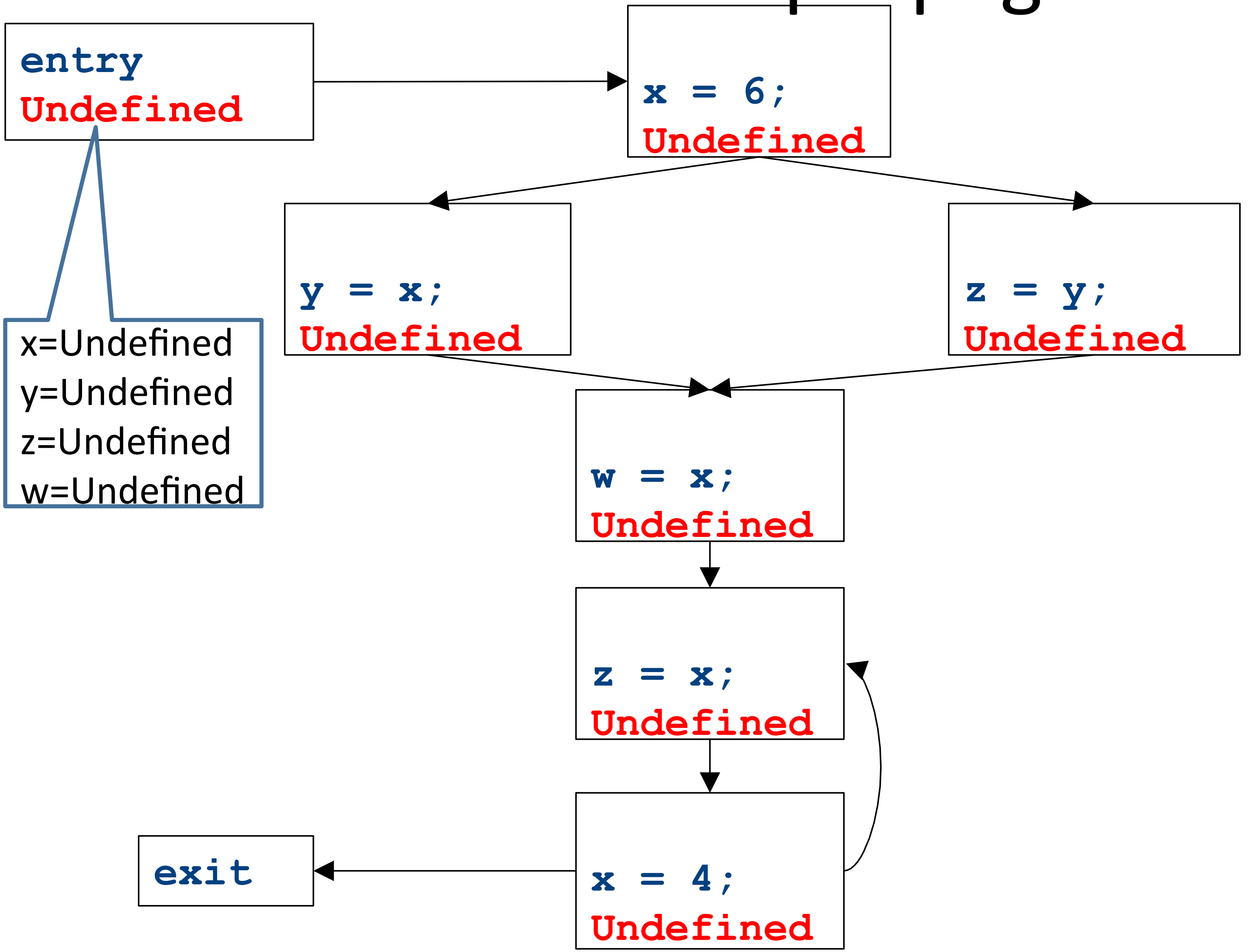


- Note:
  - The join of any two different constants is **Not-a-Constant**
  - The join of **Not a Constant** and any other value is **Not-a-Constant**
  - The join of **Undefined** and any other value is that other value

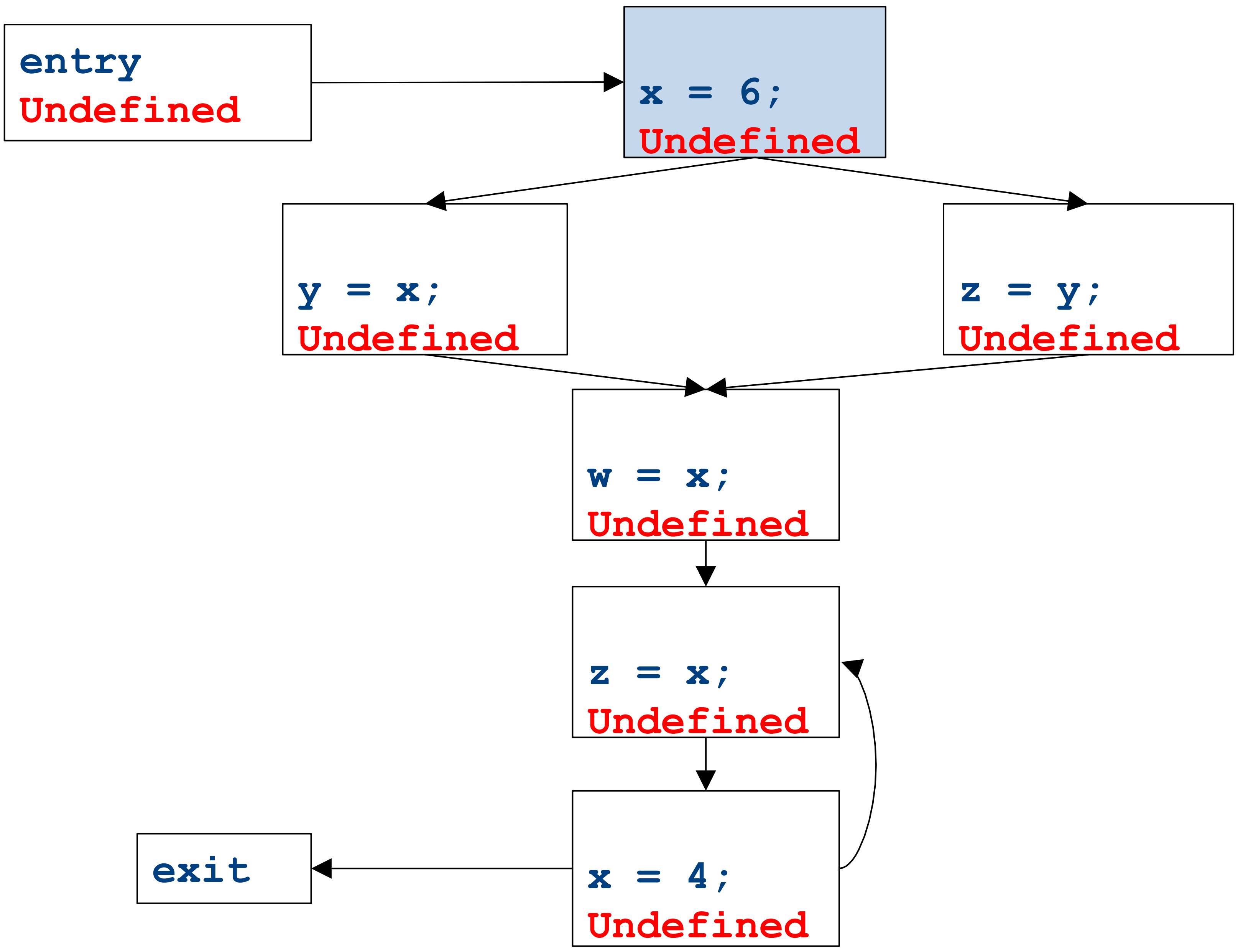
Global constant propagation



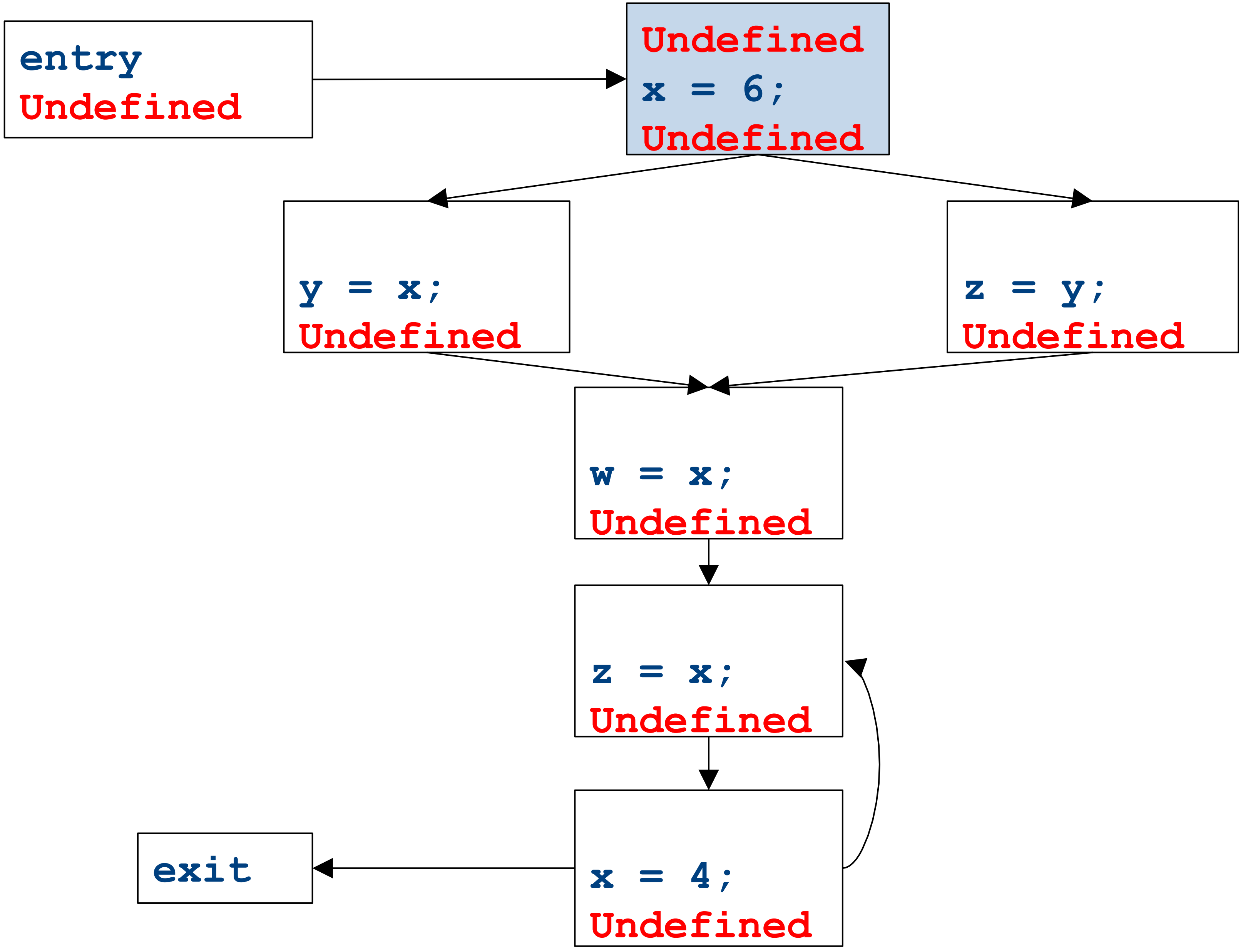
Global constant propagation



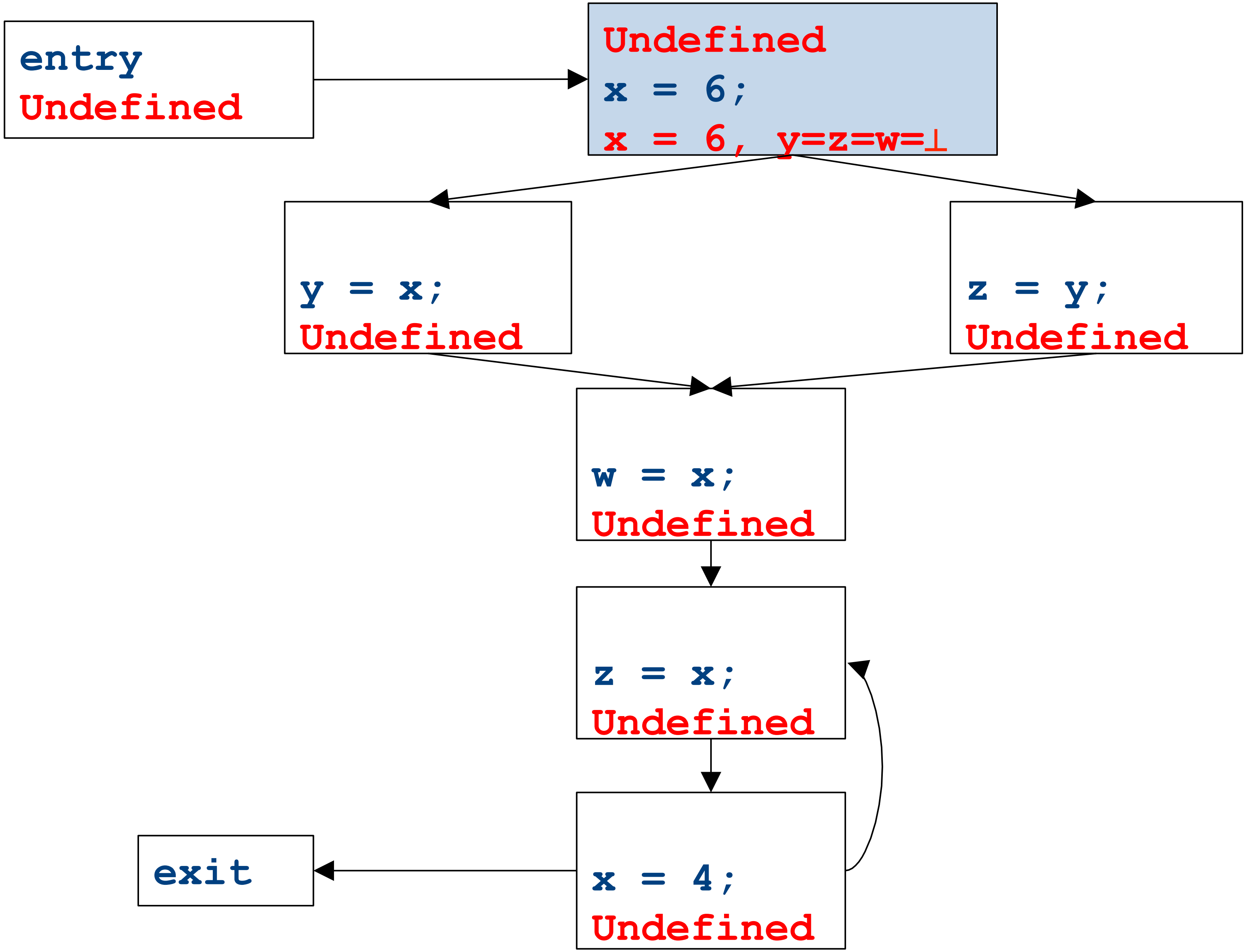
Global constant propagation



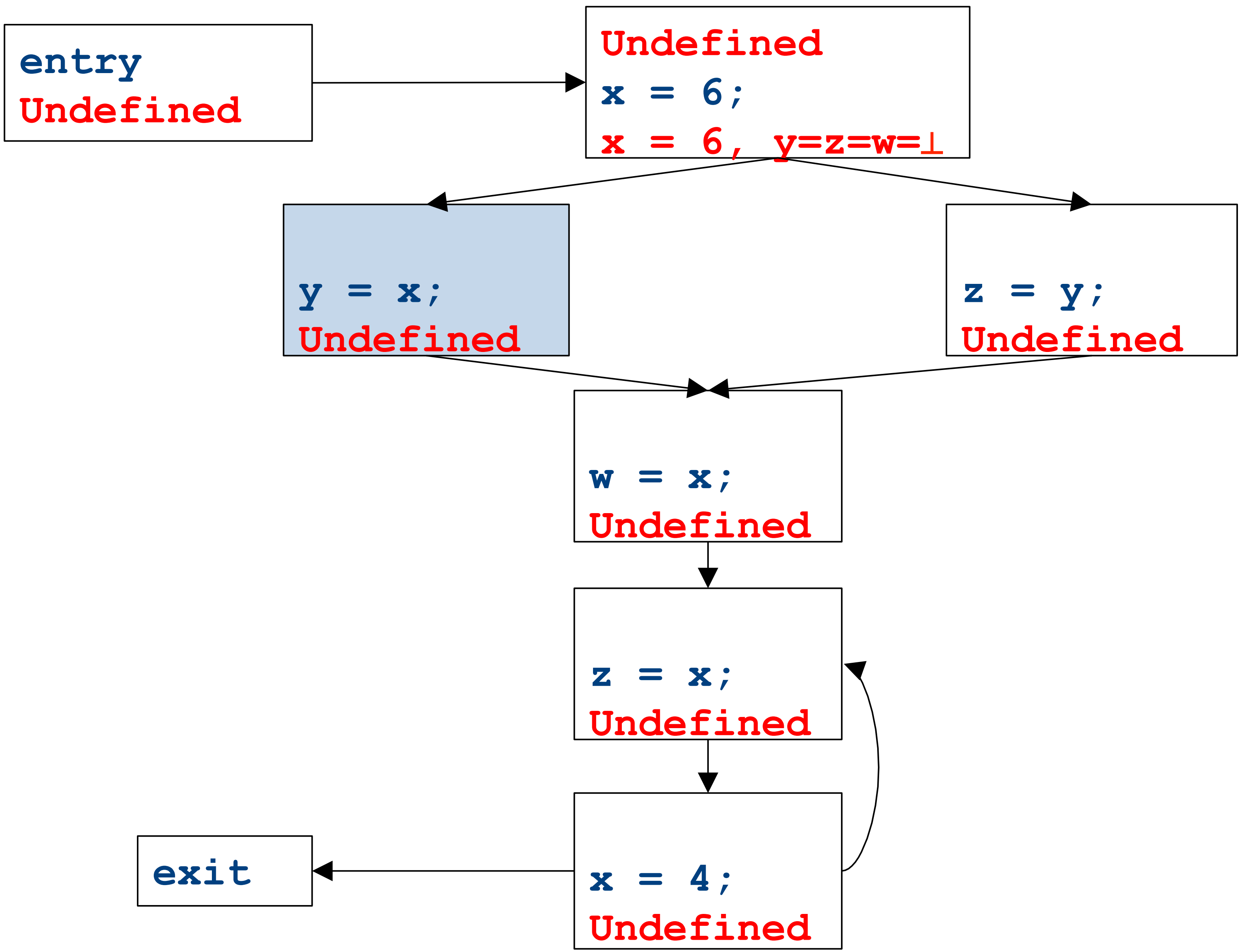
Global constant propagation



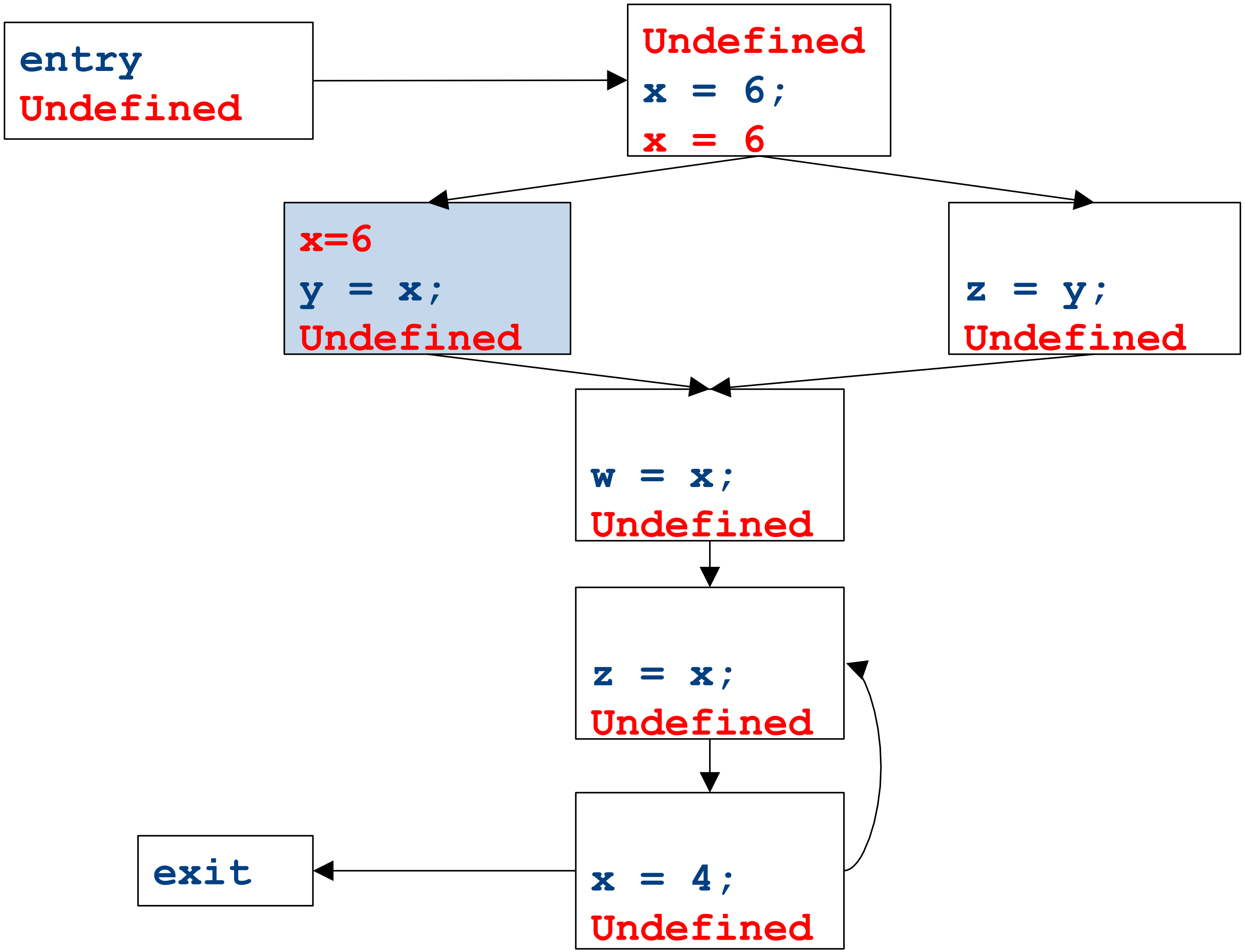
Global constant propagation



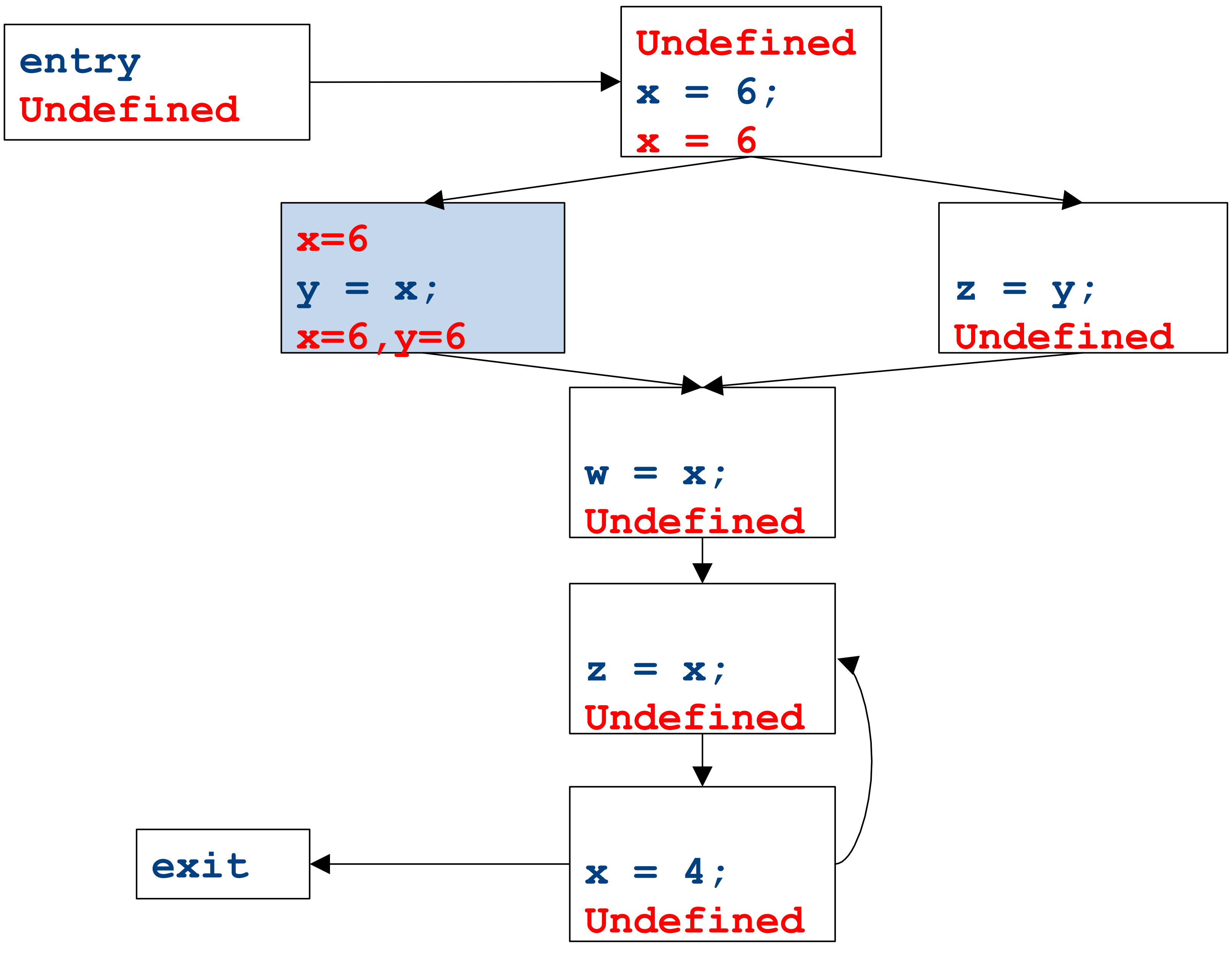
Global constant propagation



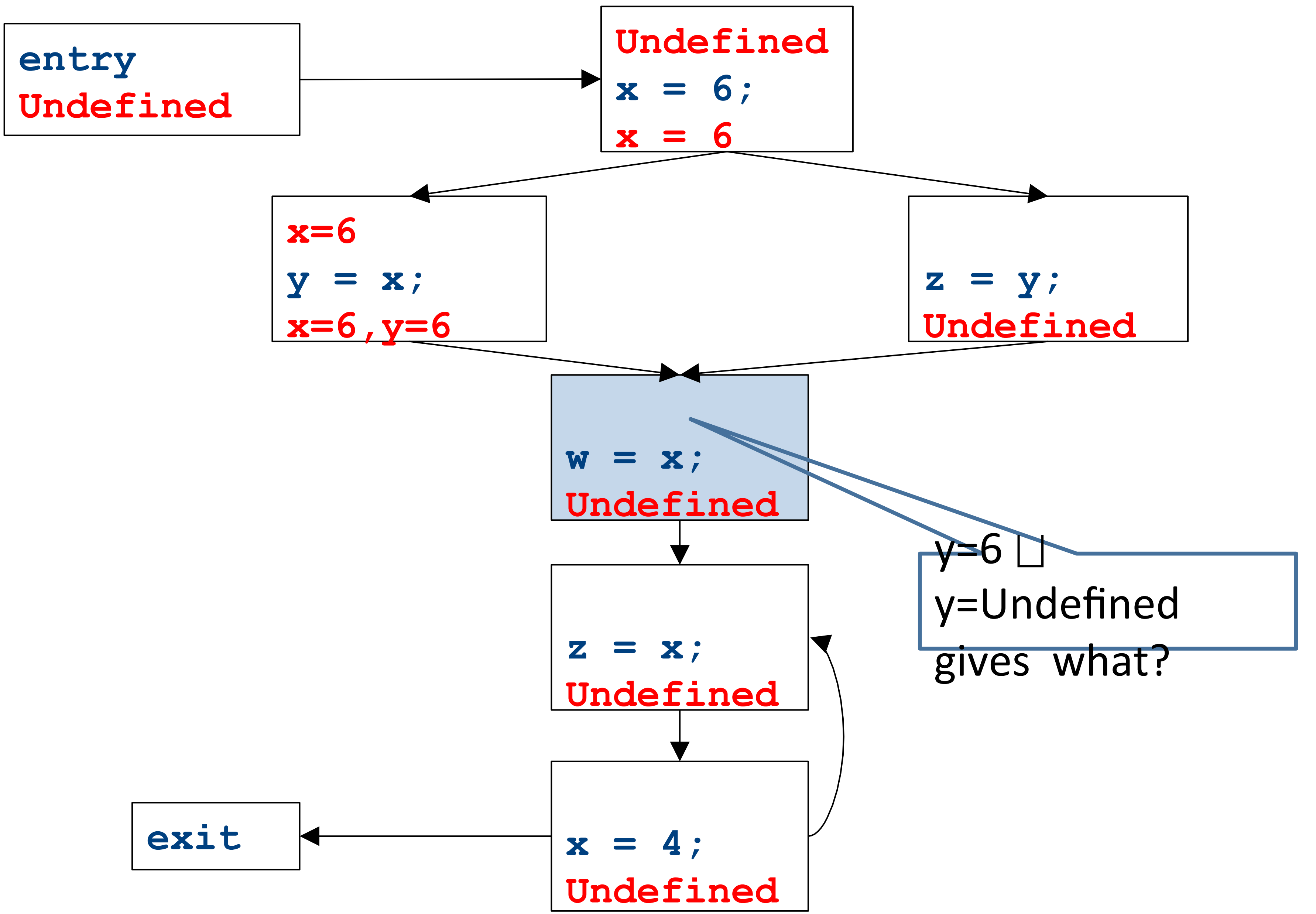
Global constant propagation



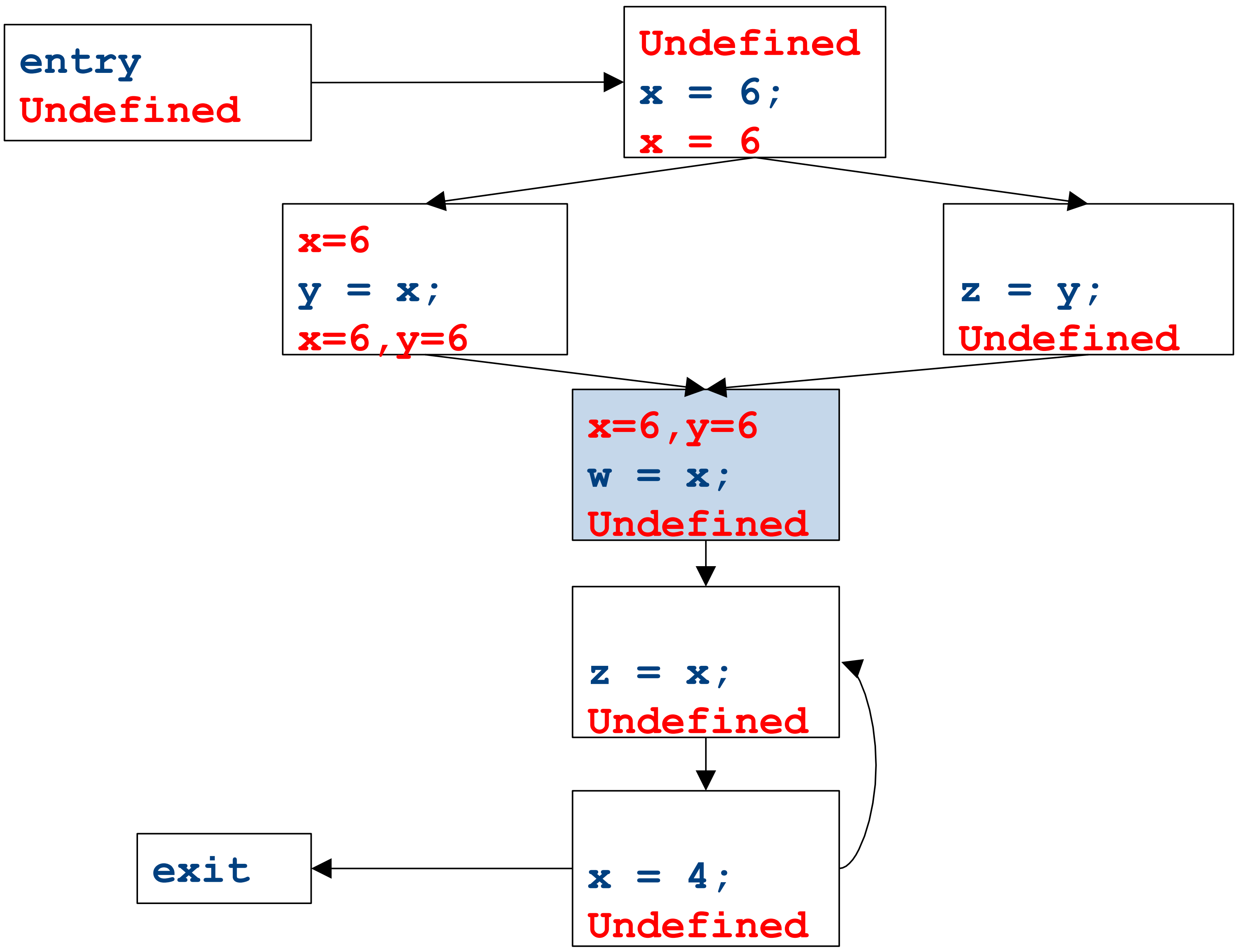
Global constant propagation



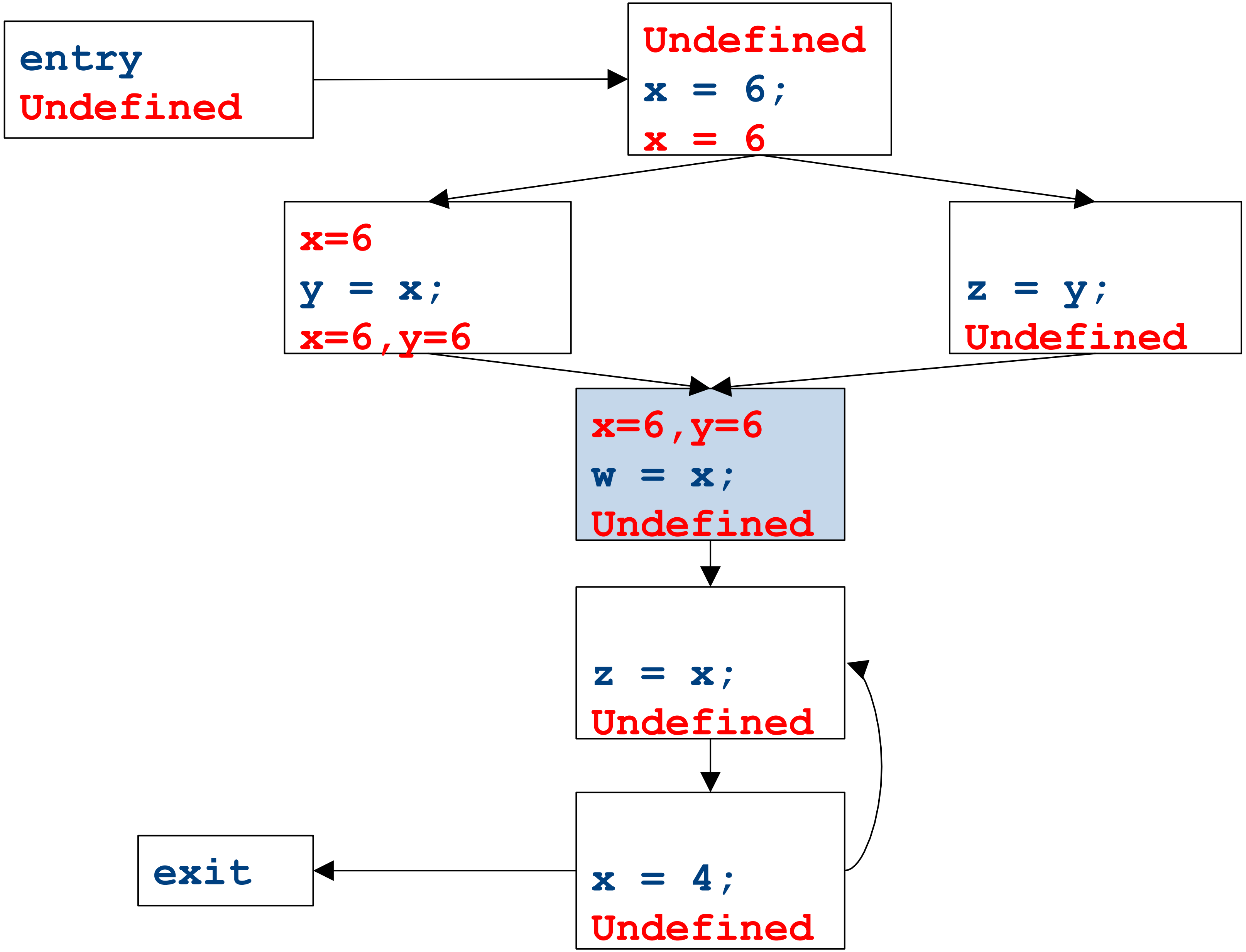
Global constant propagation



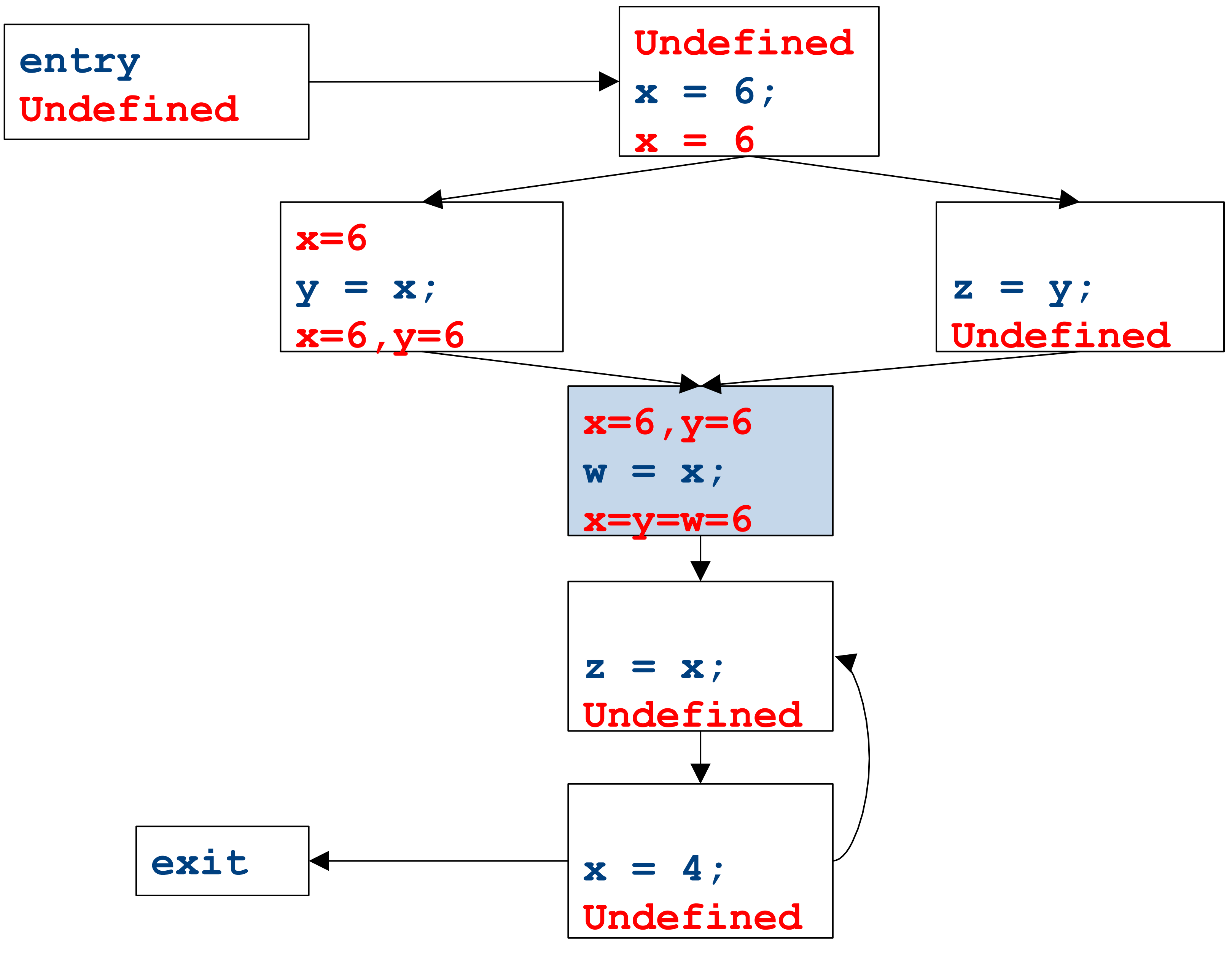
Global constant propagation



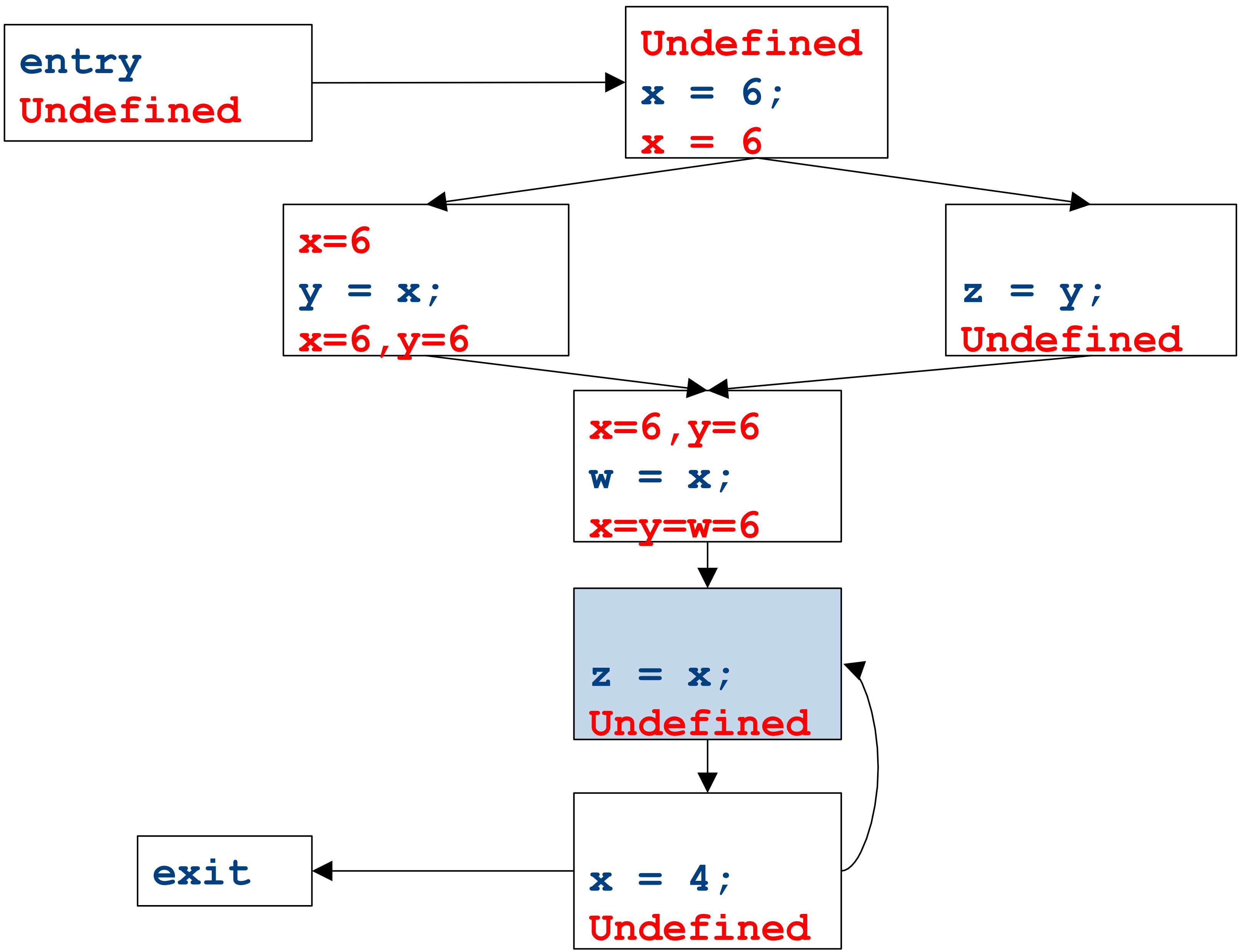
Global constant propagation



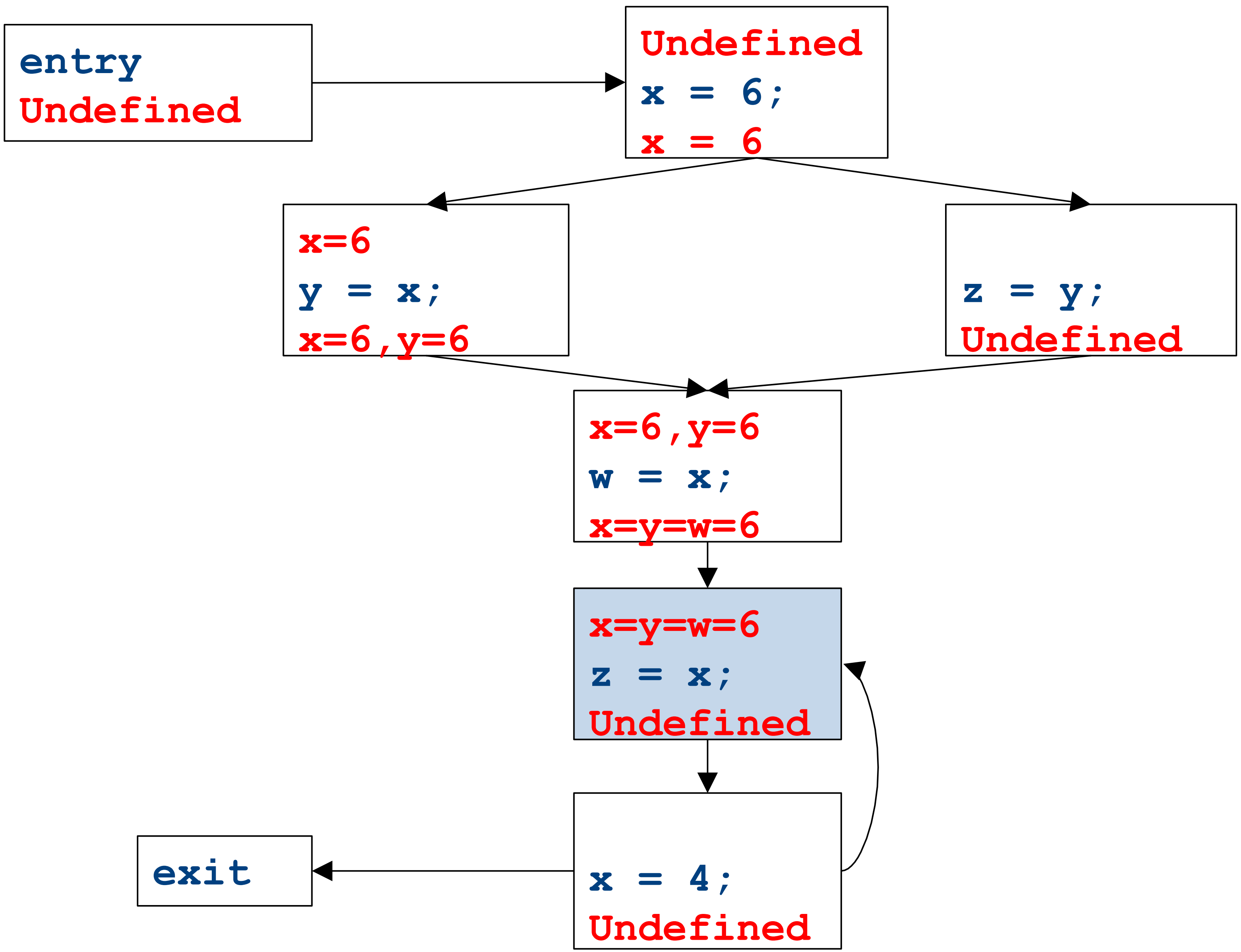
Global constant propagation



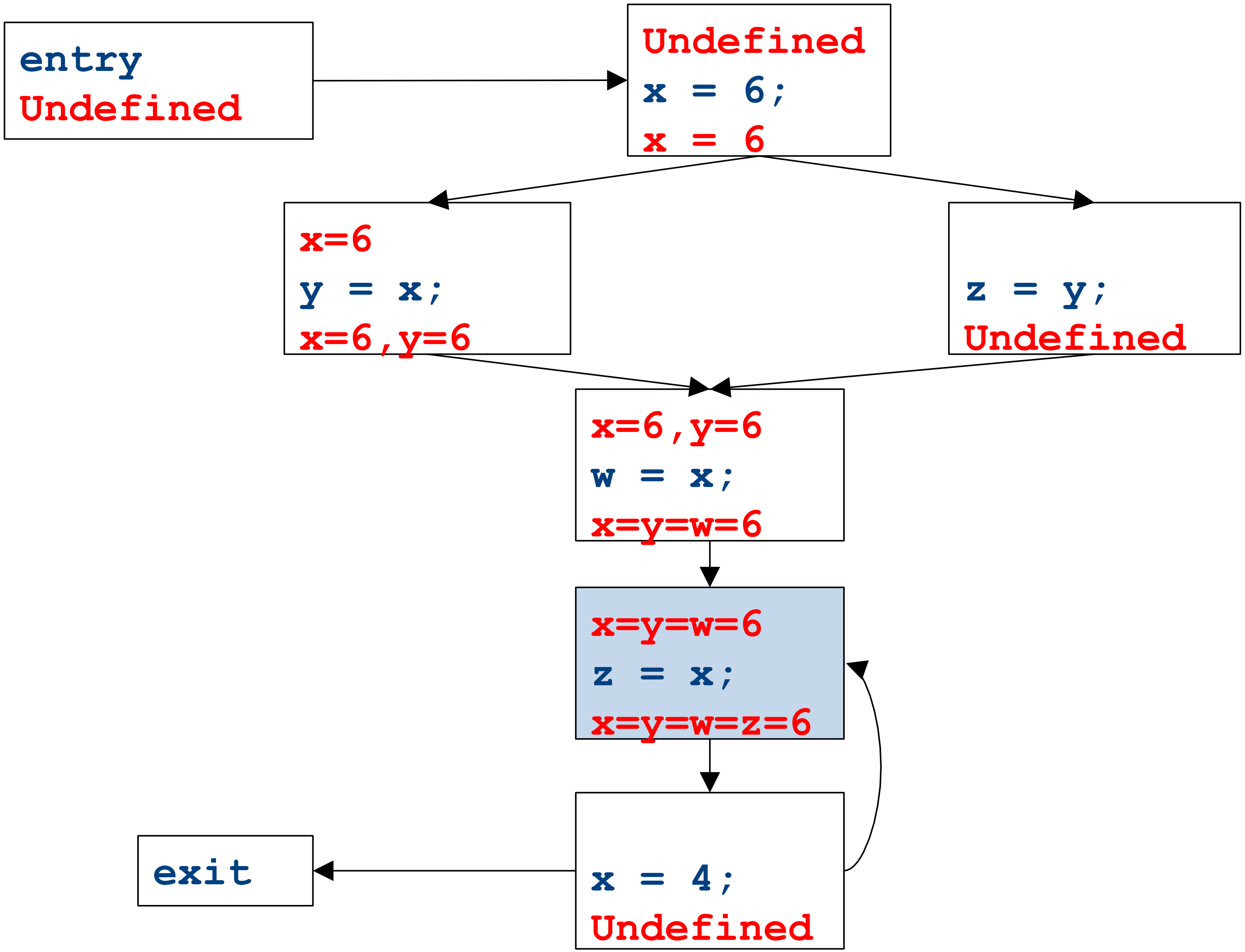
Global constant propagation



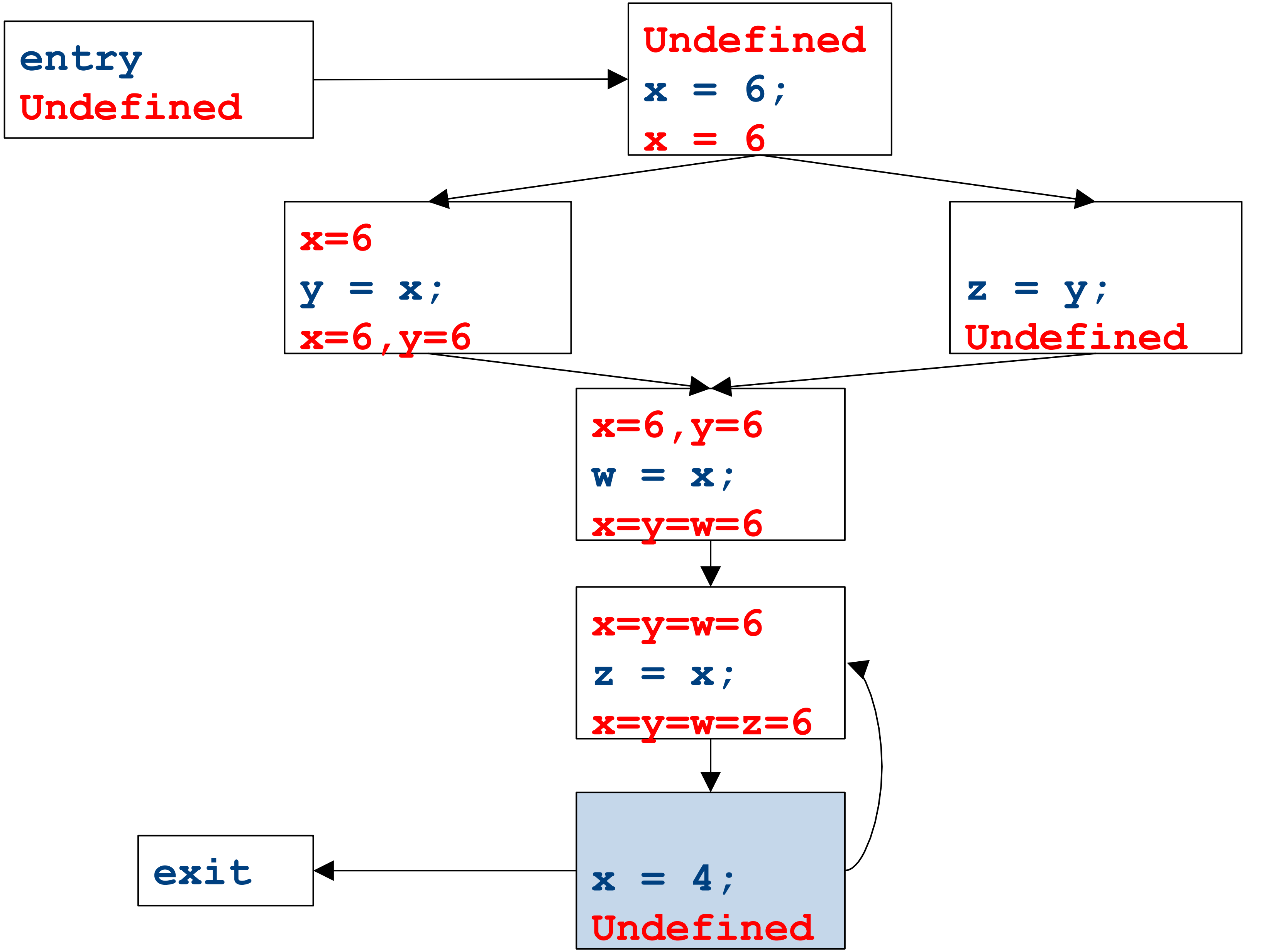
Global constant propagation



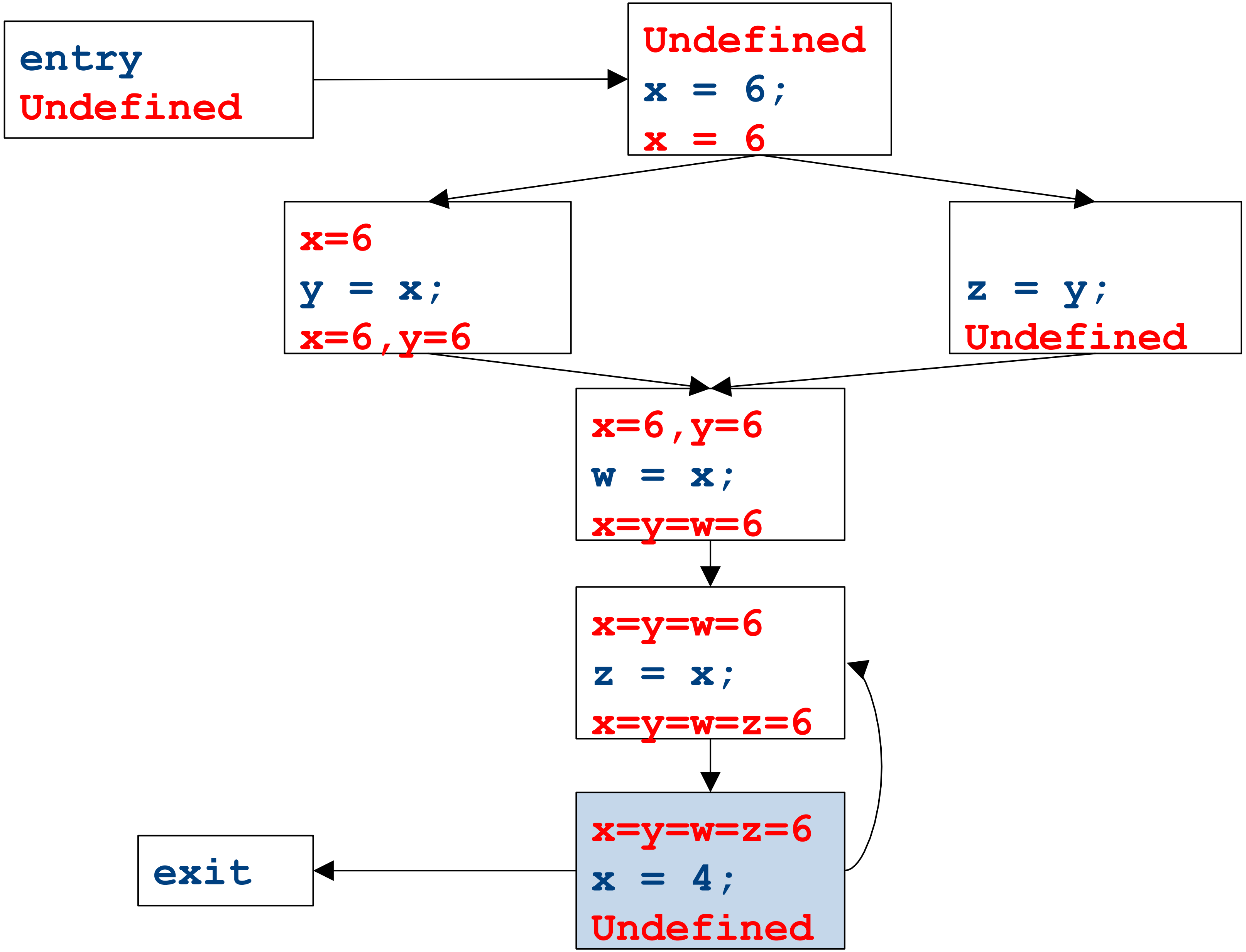
Global constant propagation



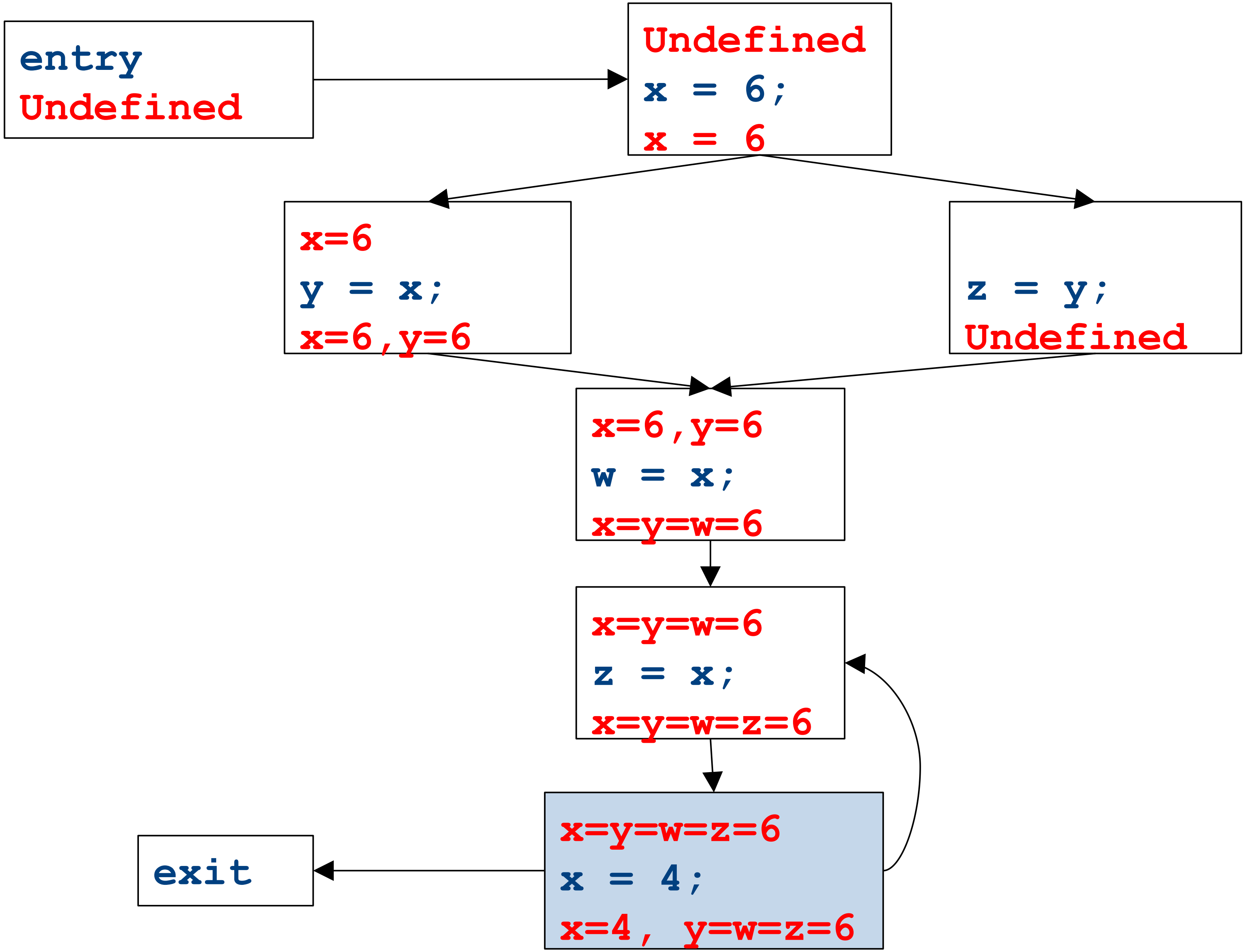
Global constant propagation



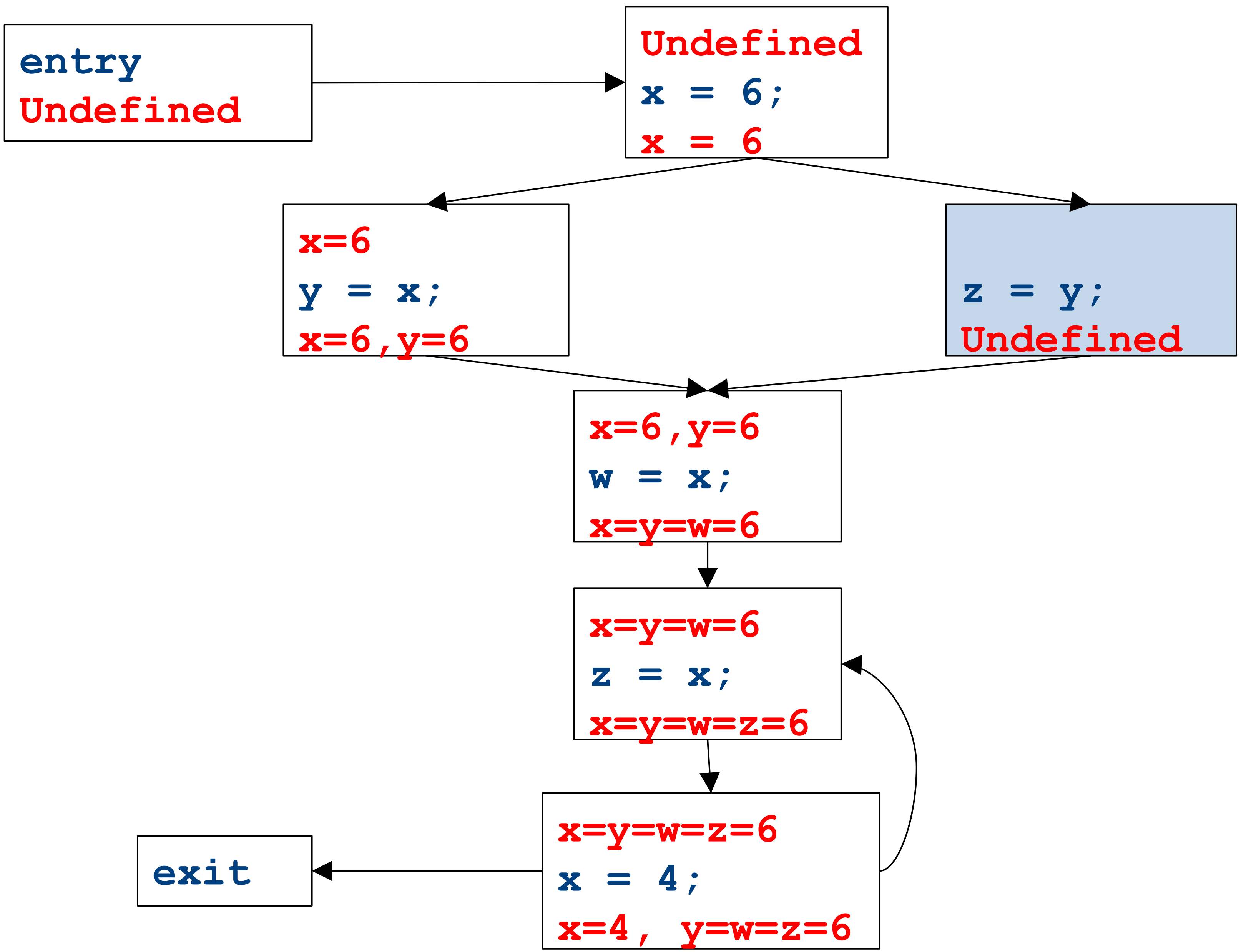
Global constant propagation



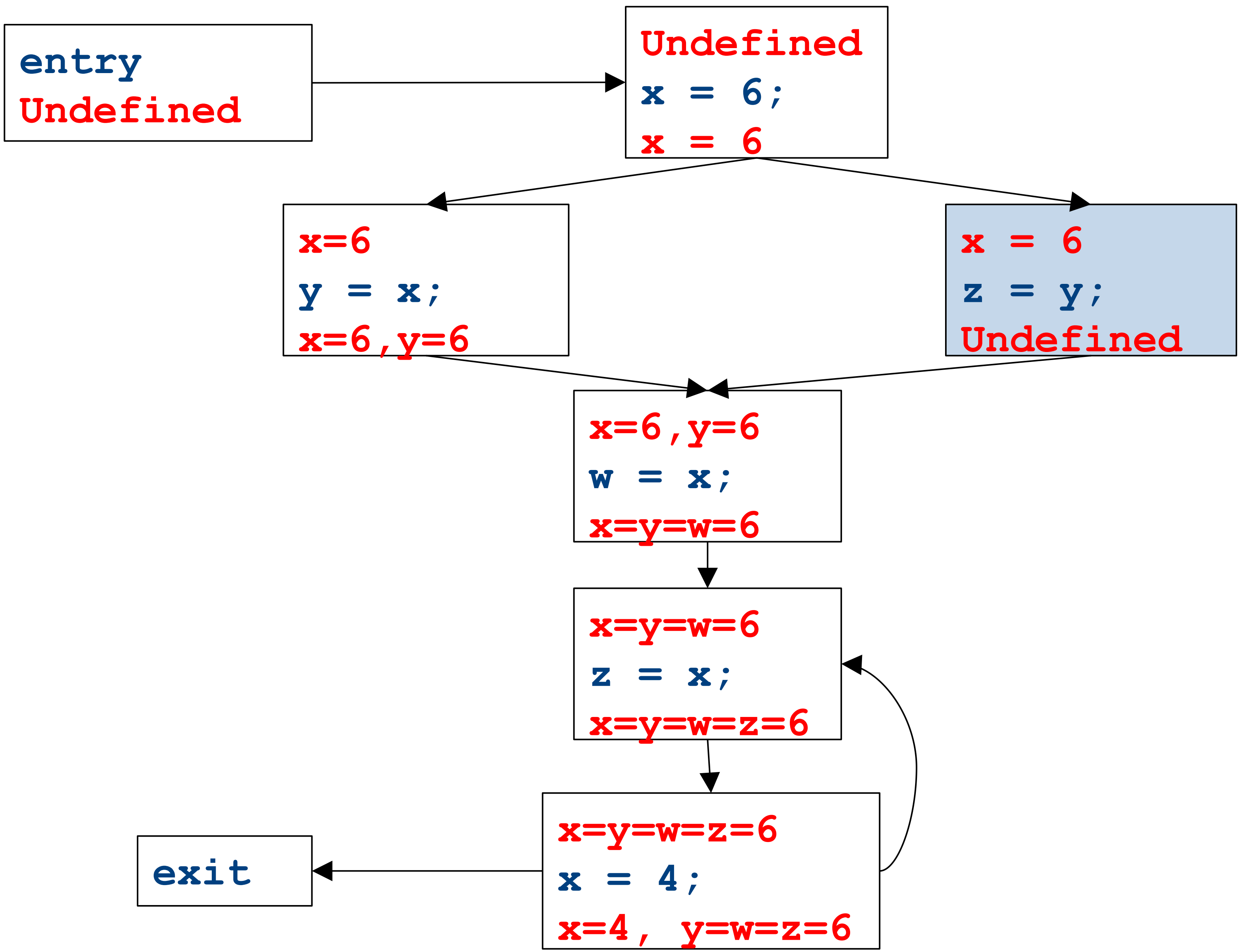
Global constant propagation



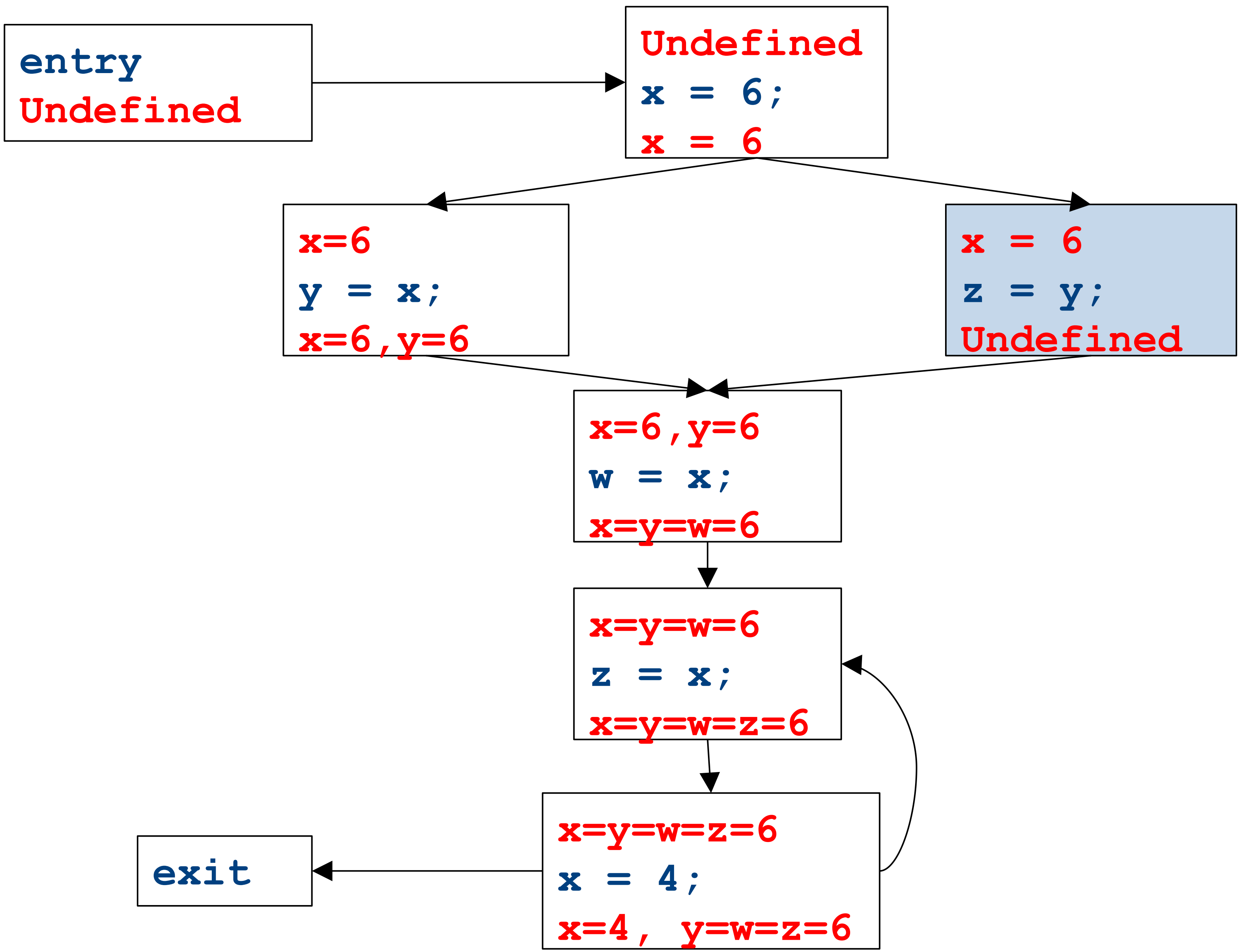
Global constant propagation



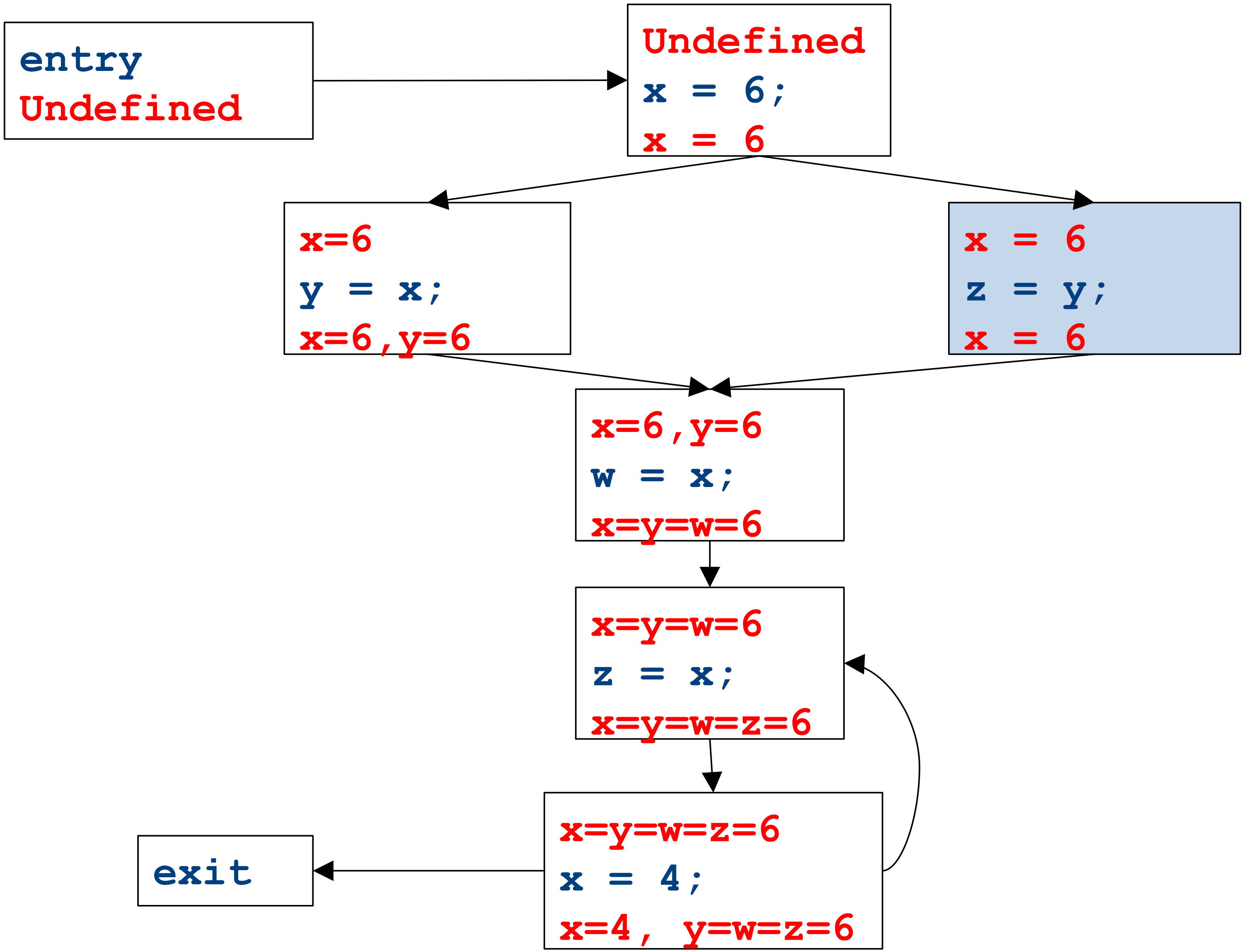
Global constant propagation



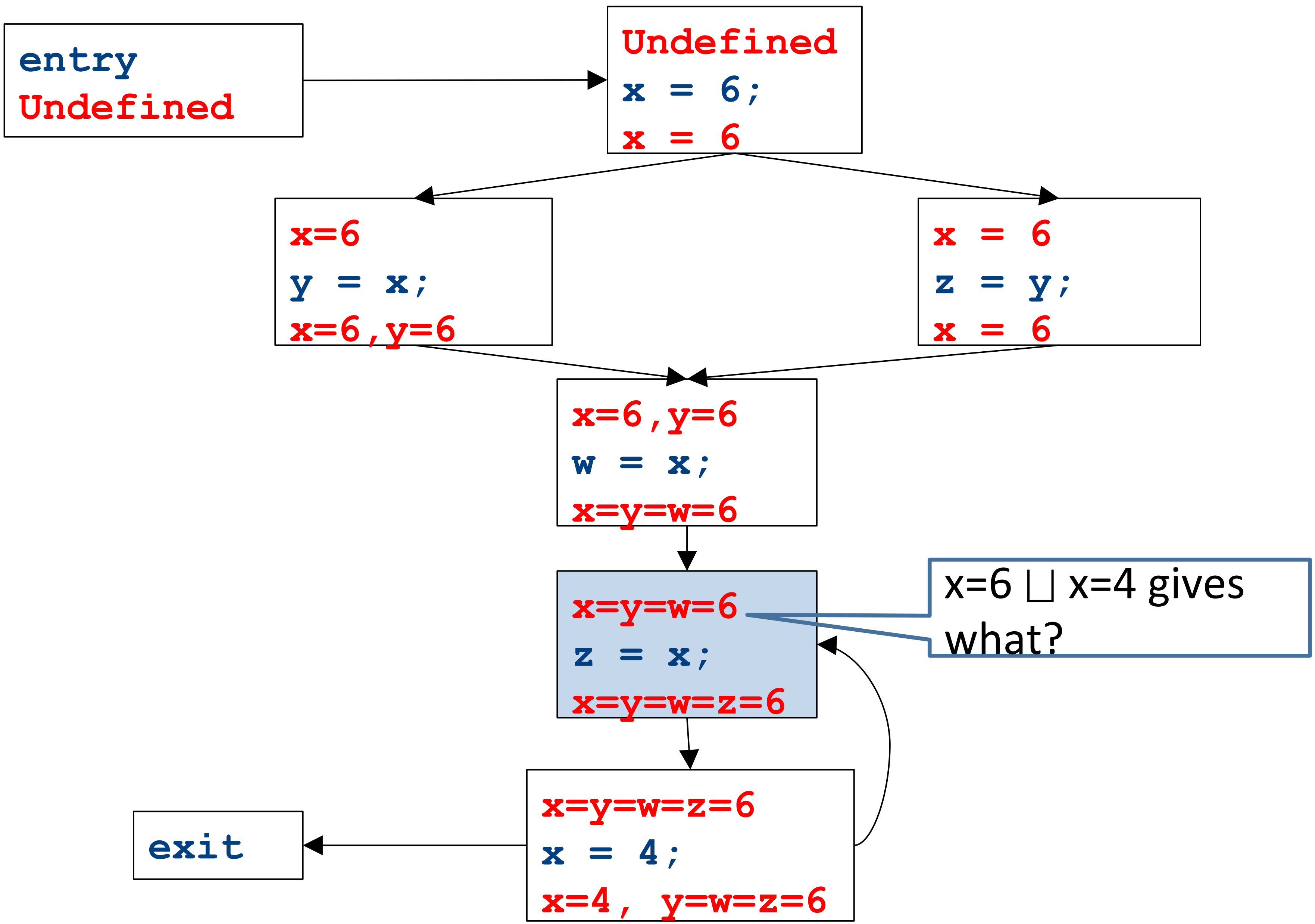
Global constant propagation



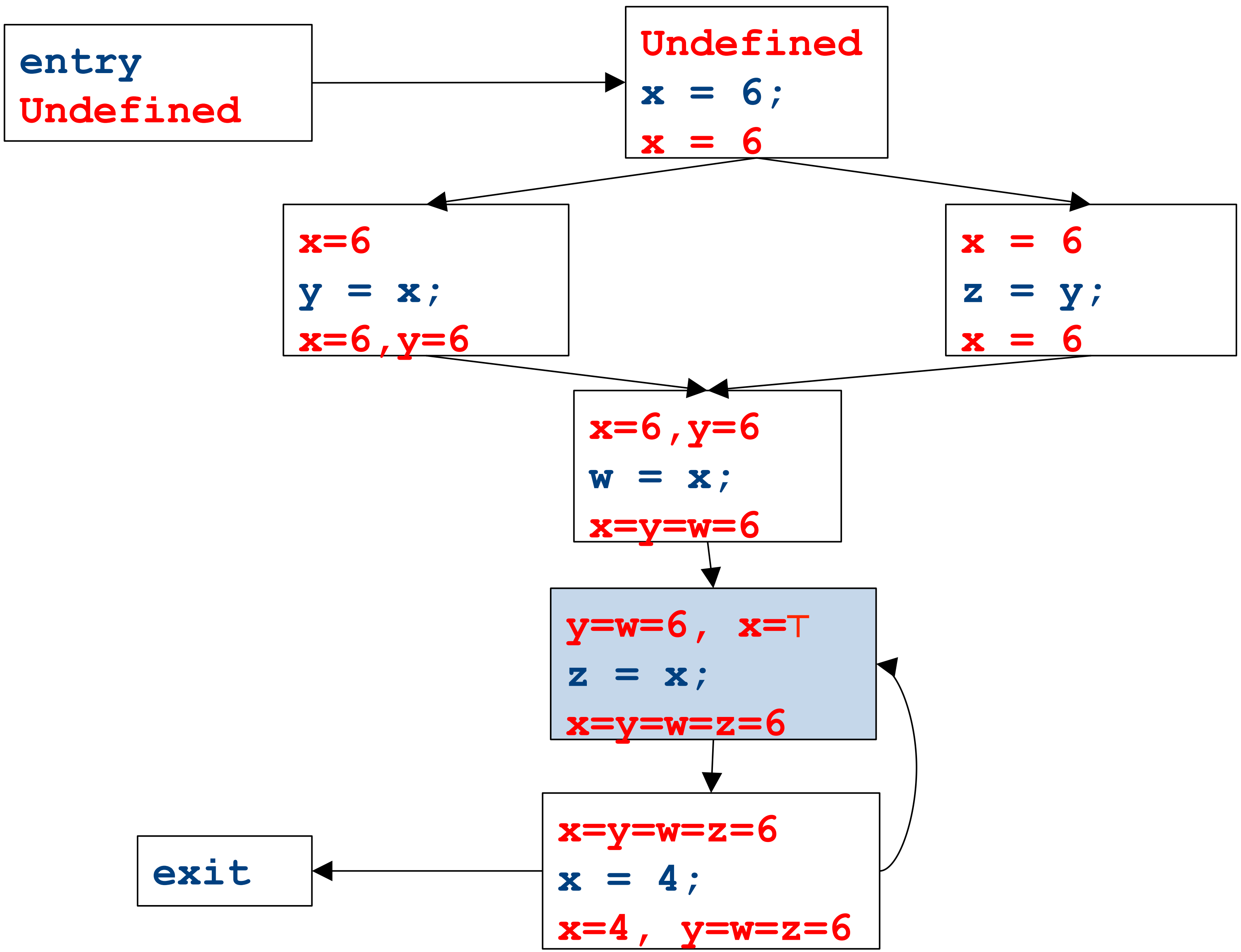
Global constant propagation



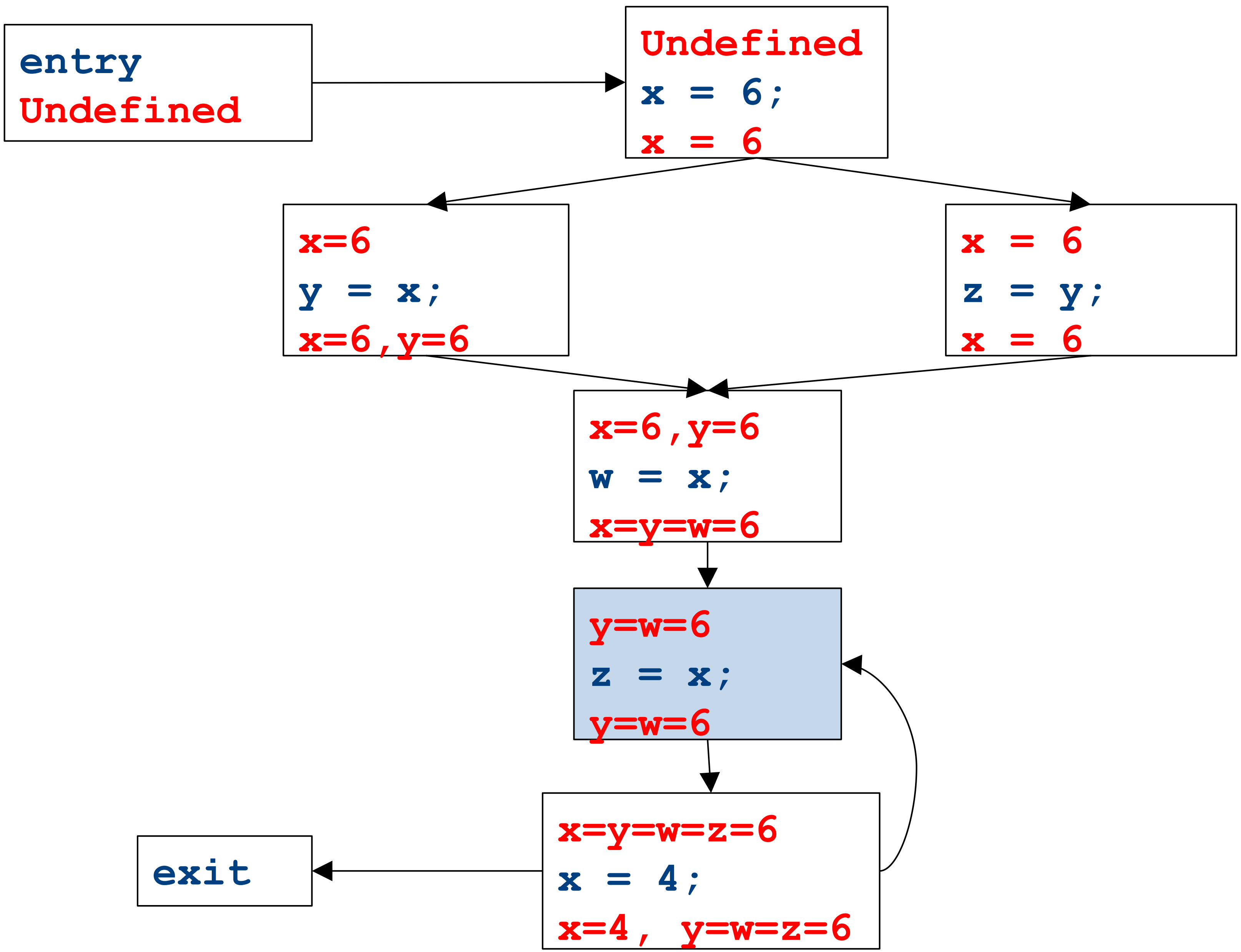
Global constant propagation



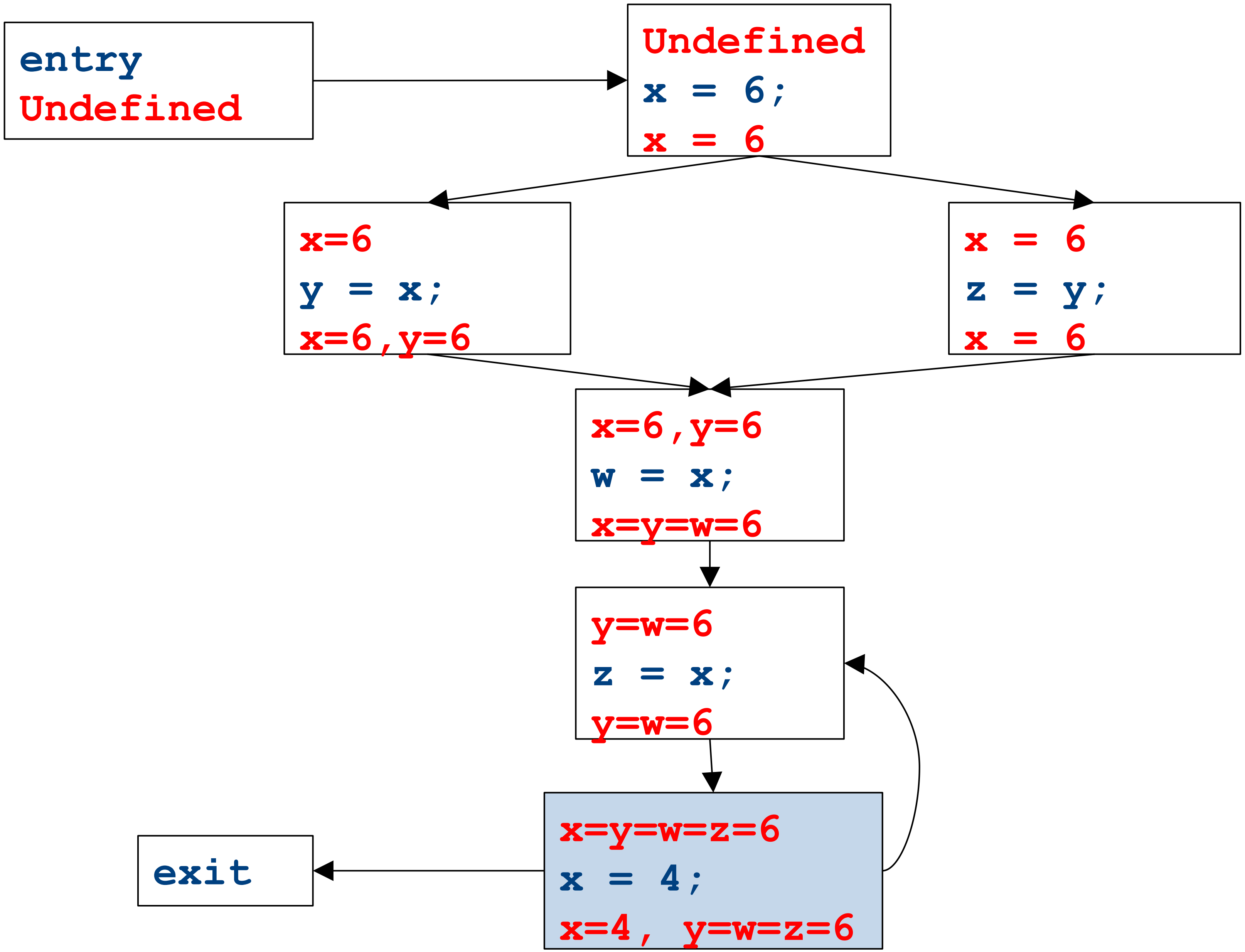
Global constant propagation



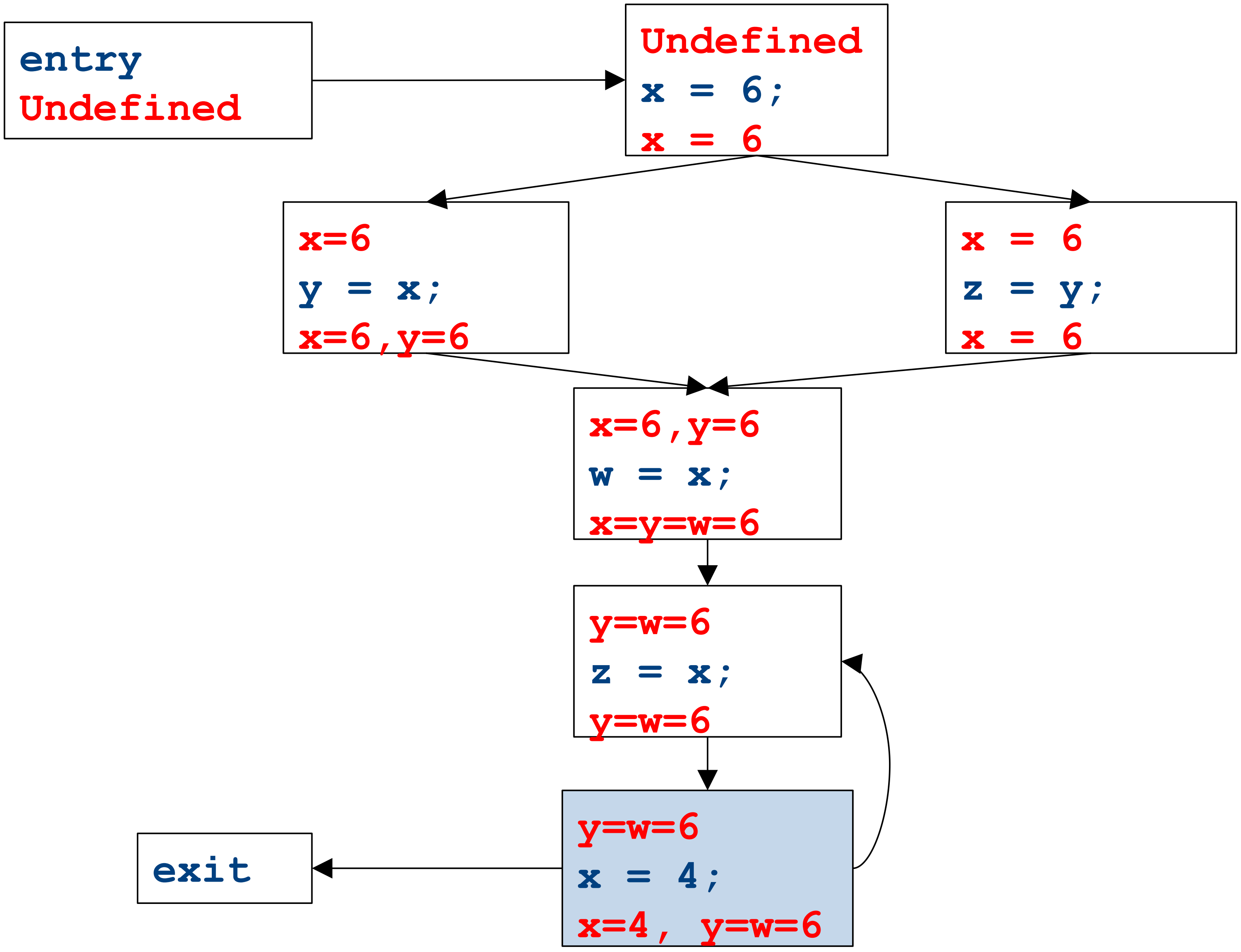
Global constant propagation



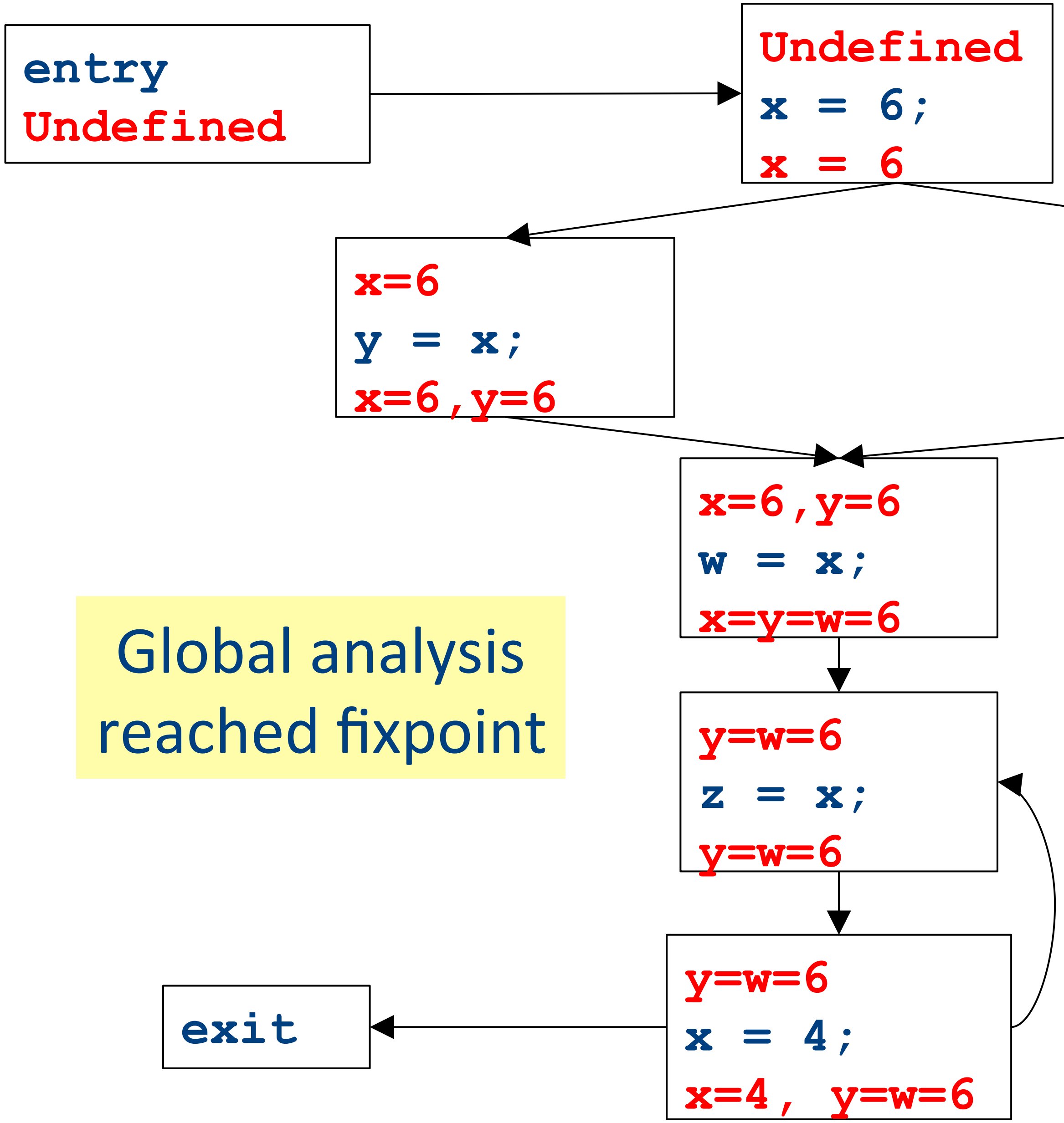
Global constant propagation



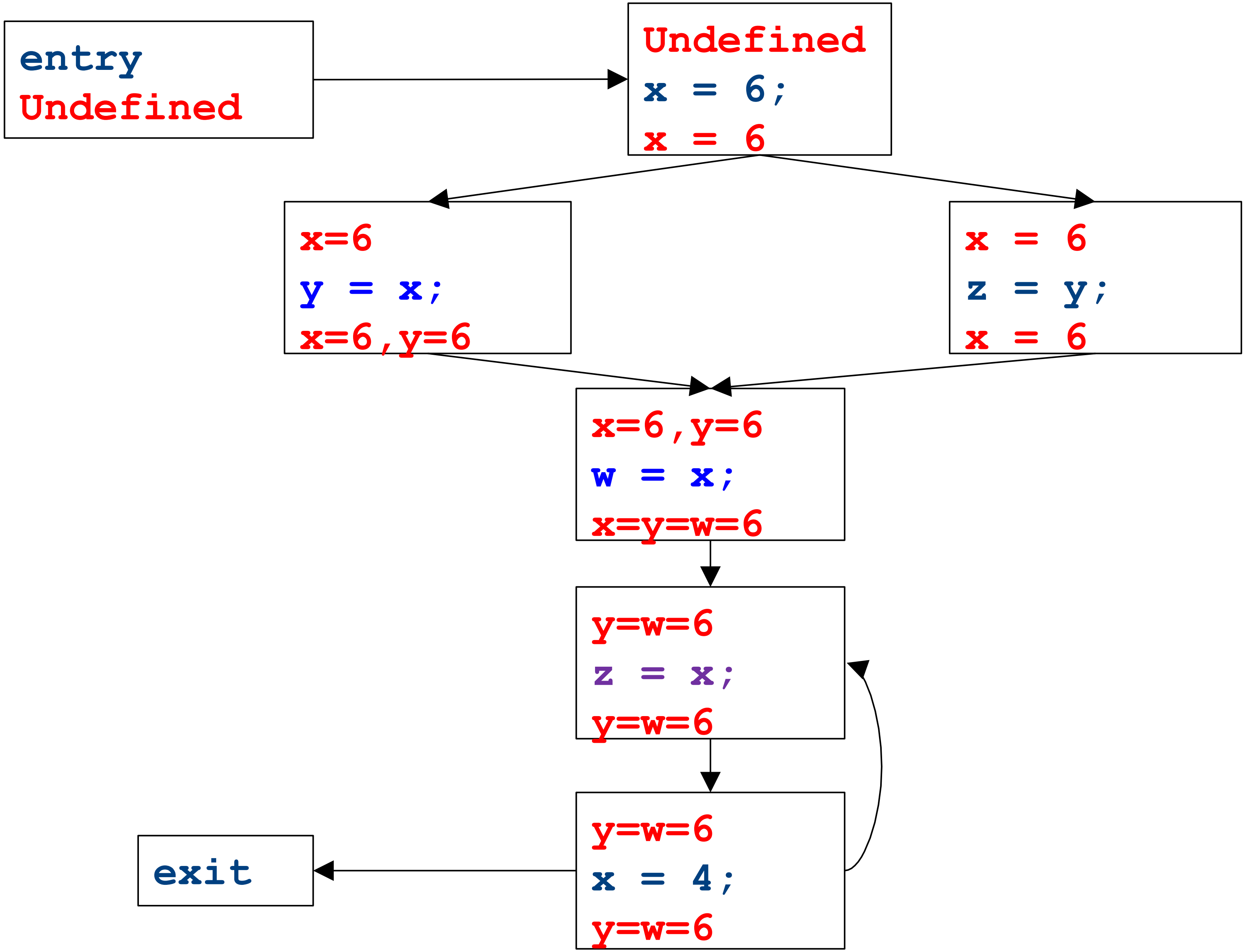
Global constant propagation



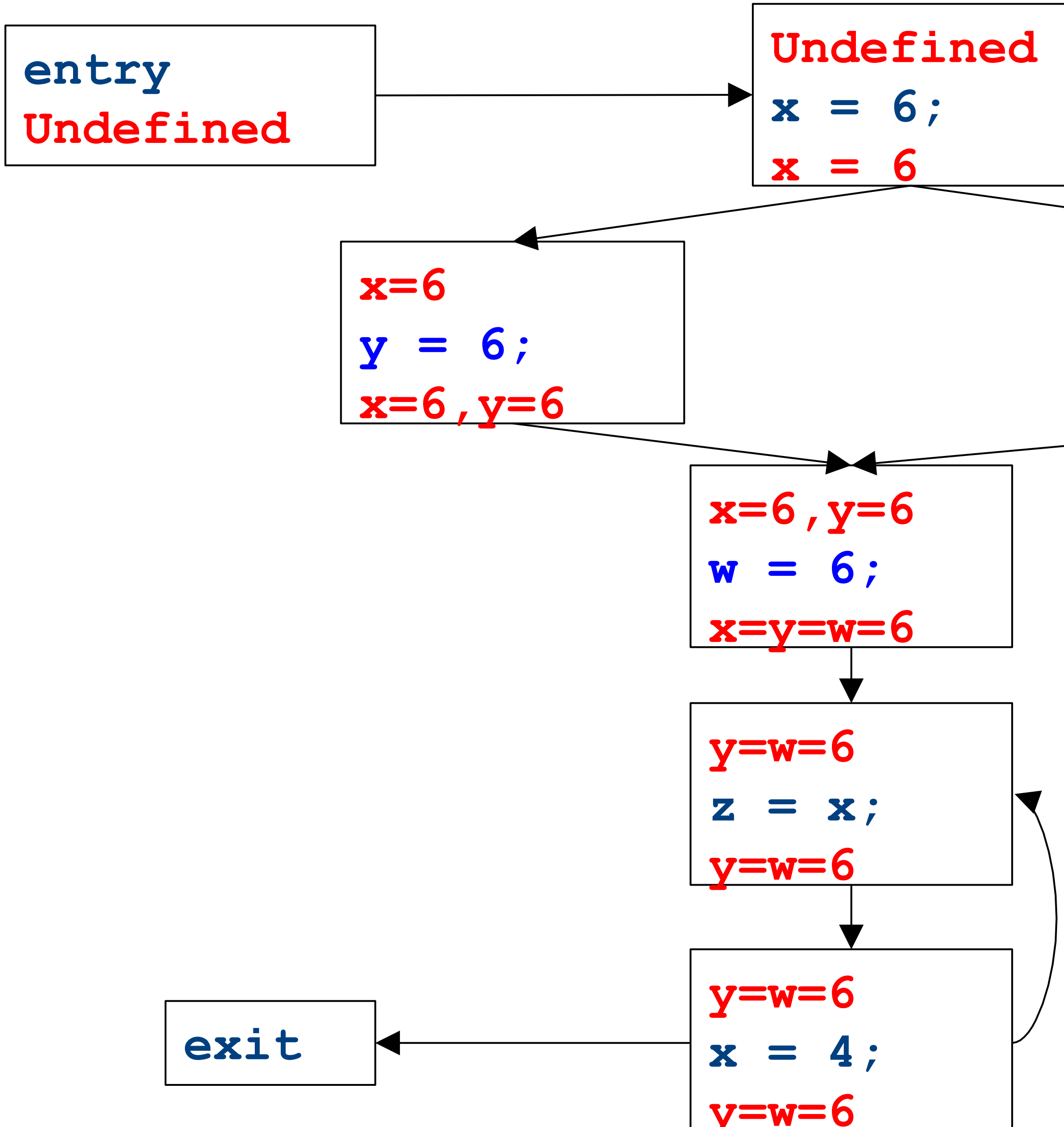
Global constant propagation



Global constant propagation



Global constant propagation



Dataflow for constant propagation

- Direction: **Forward**
- Semilattice: Vars$\longrightarrow$ {Undefined, 0, 1, -1, 2, -2, …, Not-a-Constant}
  - Join mapping for variables point-wise
    $\{x\mapsto 1, y\mapsto 1, z\mapsto 1\} \sqcup \{x\mapsto 1, y\mapsto 2, z\mapsto \text{Not-a-Constant}\} =$
    $\{x\mapsto 1, y\mapsto \text{Not-a-Constant}, z\mapsto \text{Not-a-Constant}\}$
- Transfer functions:
  - $f_{\mathtt{x=k}}(V) = V|_{x\mapsto k}$ *(update V by mapping x to k)*
  - $f_{\mathtt{x=a+b}}(V) = V|_{x\mapsto \text{Not-a-Constant}}$ *(assign Not-a-Constant)*
- Initial value: **x is Undefined**
  - (When might we use some other value?)

Proving termination

- Our algorithm for running these analyses continuously loops until no changes are detected
- Given this, how do we know the analyses will eventually terminate?
  - In general, **we don't**

Terminates?

Liveness Analysis

- A variable is live at a point in a program if later in the program its value will be read before it is written to again

Join semilattice definition

- A join semilattice is a pair (V, $\sqcup$), where
- V is a domain of elements
- $\sqcup$ is a join operator that is
  - commutative: $x \sqcup y = y \sqcup x$
  - associative: $(x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$
  - idempotent: $x \sqcup x = x$
- If $x \sqcup y = z$, we say that z is the join or (**L**east **U**pper **B**ound) of x and y
- Every join semilattice has a bottom element denoted $\bot$ such that $\bot \sqcup x = x$ for all x

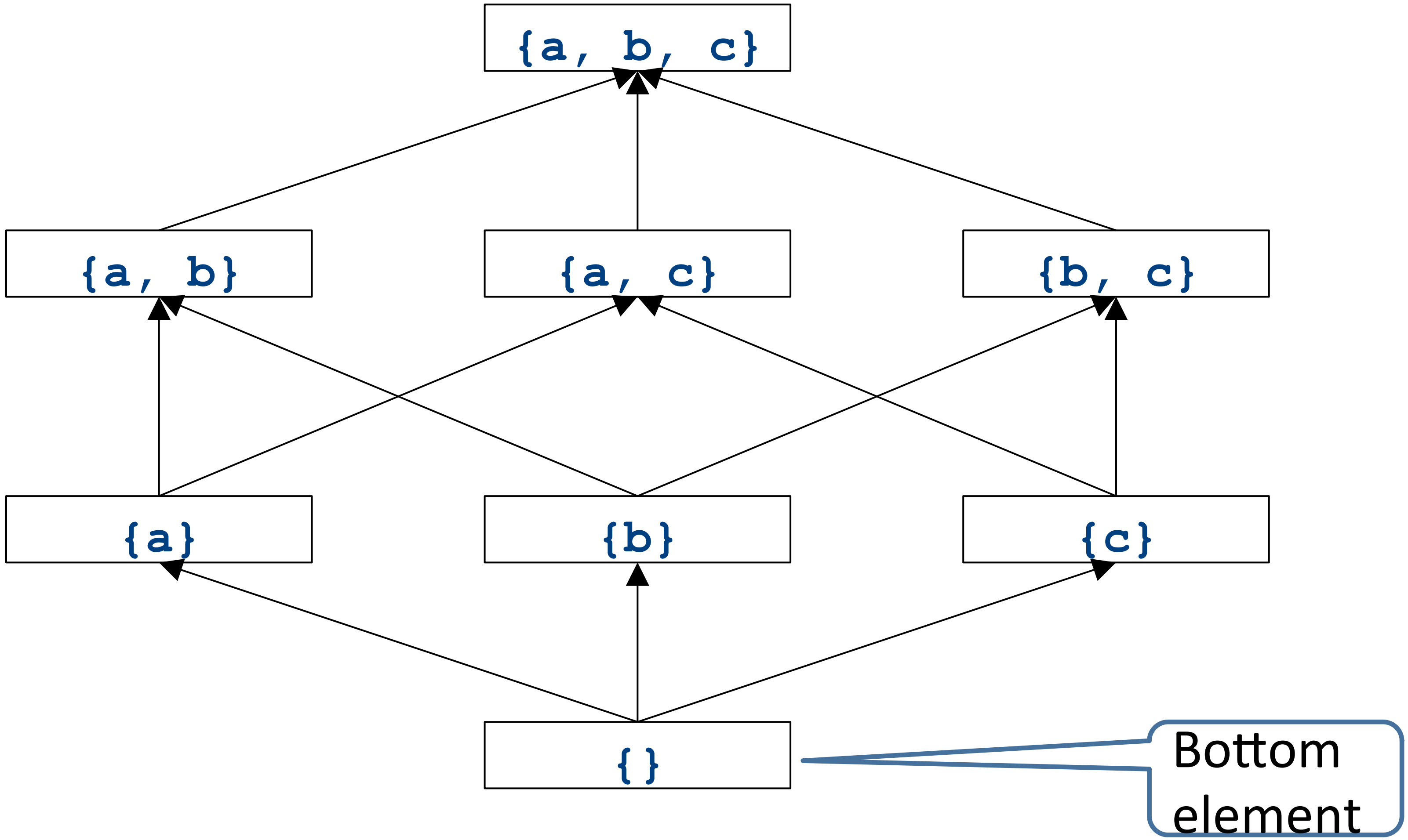
Partial ordering induced by join

- Every join semilattice (V, $\sqcup$) induces an ordering relationship $\sqsubseteq$ over its elements
- Define $x \sqsubseteq y$ iff $x \sqcup y = y$
- Need to prove
  - Reflexivity: $x \sqsubseteq x$
  - Antisymmetry: If $x \sqsubseteq y$ and $y \sqsubseteq x$, then $x = y$
  - Transitivity: If $x \sqsubseteq y$ and $y \sqsubseteq z$, then $x \sqsubseteq z$

A join semilattice for liveness

- Sets of live variables and the set union operation
- Idempotent:
  - $x \cup x = x$
- Commutative:
  - $x \cup y = y \cup x$
- Associative:
  - $(x \cup y) \cup z = x \cup (y \cup z)$
- Bottom element:
  - The empty set: $\emptyset \cup x = x$
- Ordering over elements = subset relation

Join semilattice example for liveness



Dataflow framework

- A global analysis is a tuple (**D, V, ⊔, F, I**), where
  - **D** is a direction (forward or backward)
    - The order to visit statements within a basic block, **NOT** the order in which to visit the basic blocks
  - **V** is a set of values (sometimes called domain)
  - ⊔ is a join operator over those values
  - **F** is a set of transfer functions $f_s : \mathbf{V} \rightarrow \mathbf{V}$ (for every statement s)
  - **I** is an initial value

Running global analyses

- Assume that (**D, V, $\sqcup$, F, I**) is a forward analysis
- For every statement s maintain values before - IN[s] - and after - OUT[s]
- Set OUT[**s**] = $\perp$ for all statements **s**
- Set OUT[**entry**] = **I**
- Repeat until no values change:
  - For each statement **s** with predecessors PRED[s]={$\mathbf{p_1}$, $\mathbf{p_2}$, … , $\mathbf{p_n}$}
    - Set IN[**s**] = OUT[$\mathbf{p_1}$] $\sqcup$ OUT[$\mathbf{p_2}$] $\sqcup$ … $\sqcup$ OUT[$\mathbf{p_n}$]
    - Set OUT[**s**] = f_s(IN[**s**])
- The order of this iteration does not matter
  - Chaotic iteration

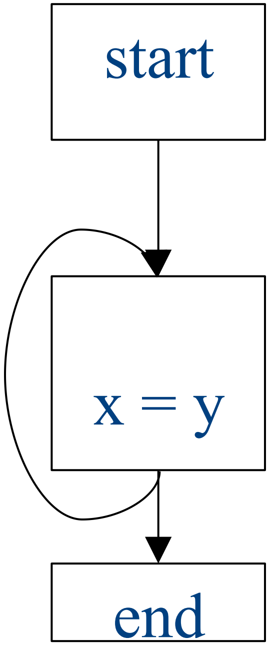
Proving termination

- Our algorithm for running these analyses continuously loops until no changes are detected
- Problem: how do we know the analyses will eventually terminate?

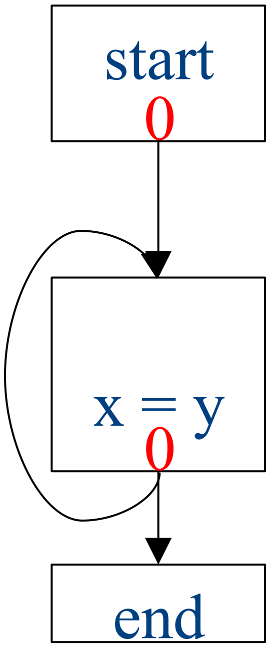
A non-terminating analysis

- The following analysis will loop infinitely on any CFG containing a loop:
- Direction: Forward
- Domain: $\mathbb{N}$
- Join operator: **max**
- Transfer function: $f(n) = n + 1$
- Initial value: 0

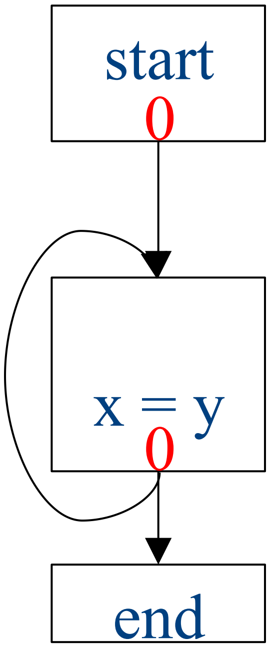
A non-terminating analysis



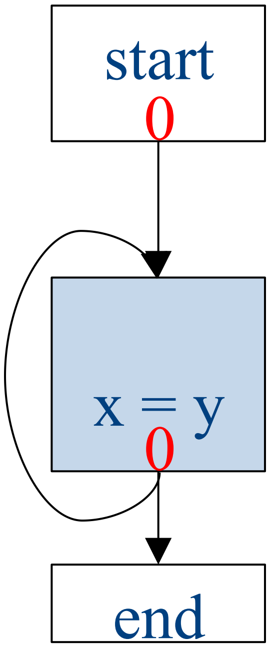
Initialization



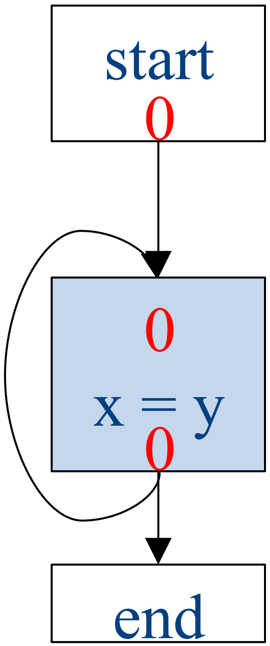
Fixed-point iteration



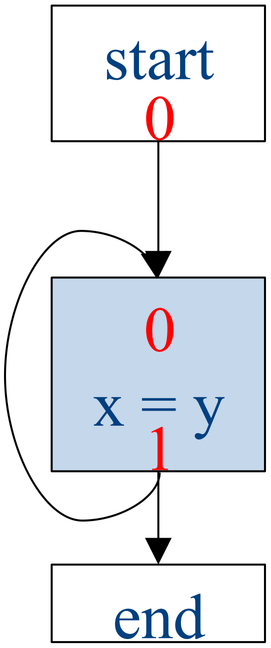
Choose a block



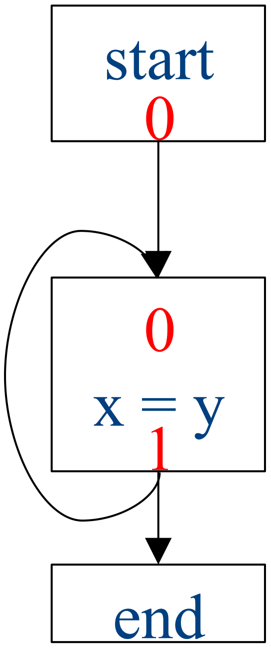
Iteration 1



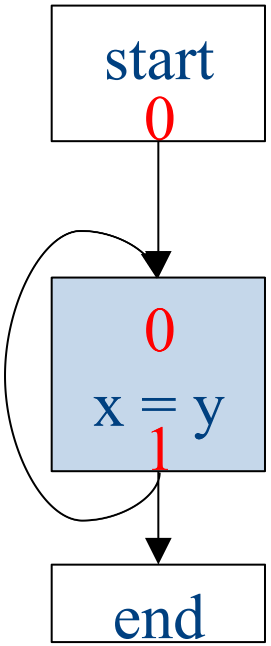
Iteration 1



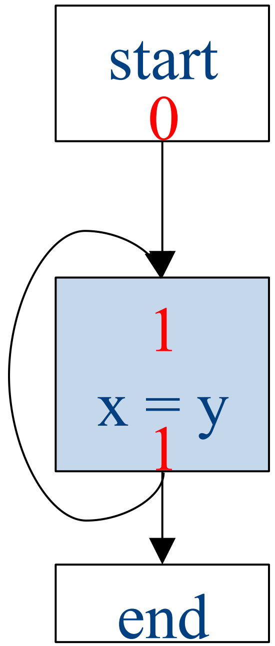
Choose a block



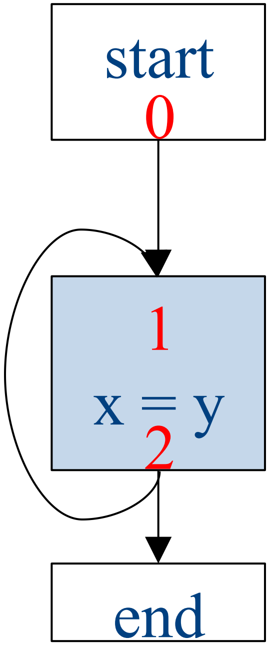
Iteration 2



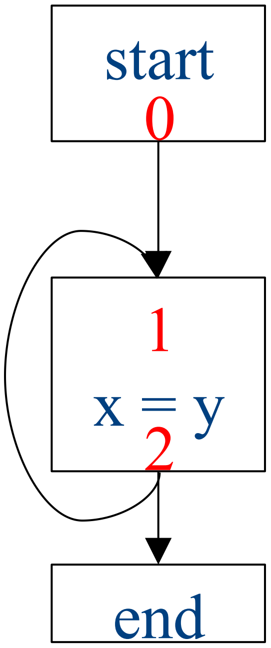
Iteration 2



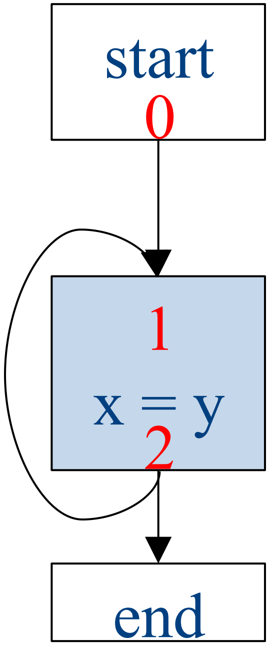
Iteration 2



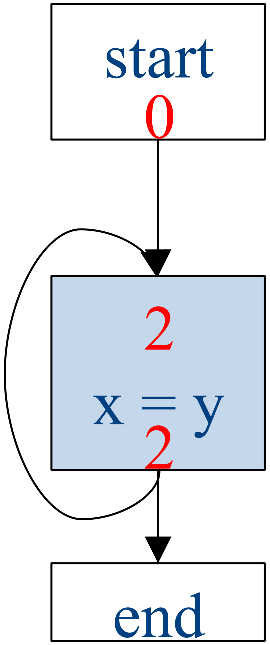
Choose a block



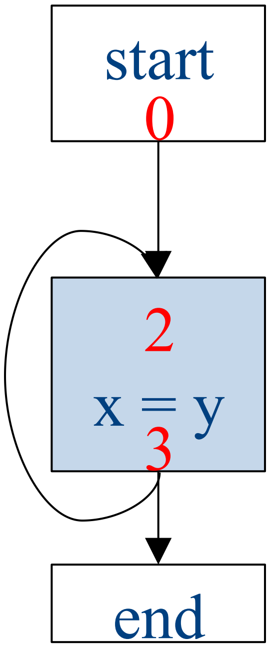
Iteration 3



Iteration 3

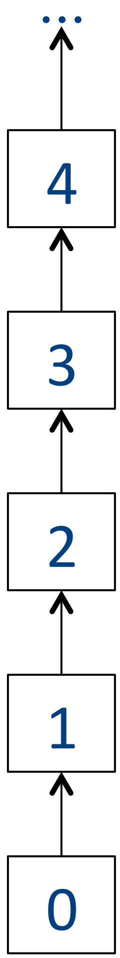


Iteration 3



Why doesn't this terminate?

- Values can increase without bound
- Note that "increase" refers to the lattice ordering, not the ordering on the natural numbers
- The height of a semilattice is the length of the longest increasing sequence in that semilattice
- The dataflow framework is not guaranteed to terminate for semilattices of infinite height
- Note that a semilattice can be infinitely large but have finite height
  - e.g. constant propagation



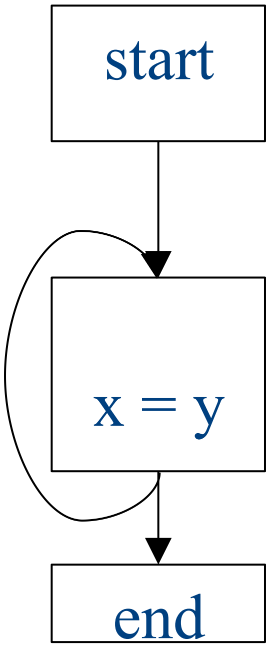
Height of a lattice

- An increasing chain is a sequence of elements $\bot \sqsubset a_1 \sqsubset a_2 \sqsubset \ldots \sqsubset a_k$
  - The length of such a chain is k
- The height of a lattice is the length of the maximal increasing chain
- For liveness with *n* program variables:
  - $\{\} \subset \{v_1\} \subset \{v_1,v_2\} \subset \ldots \subset \{v_1,\ldots,v_n\}$
- For available expressions it is the number of expressions of the form a=b op c
  - For *n* program variables and *m* operator types: $m \cdot n^3$

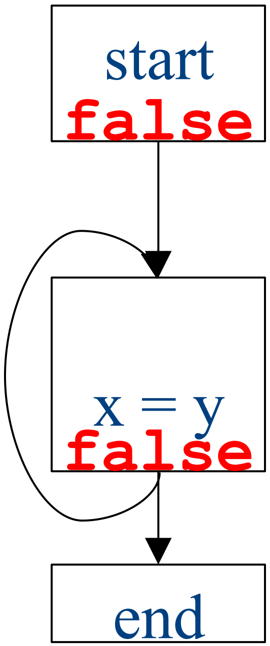
Another non-terminating analysis

- This analysis works on a finite-height semilattice, but will not terminate on certain CFGs:
- Direction: Forward
- Domain: Boolean values **`true`** and **`false`**
- Join operator: Logical OR
- Transfer function: Logical NOT
- Initial value: **`false`**

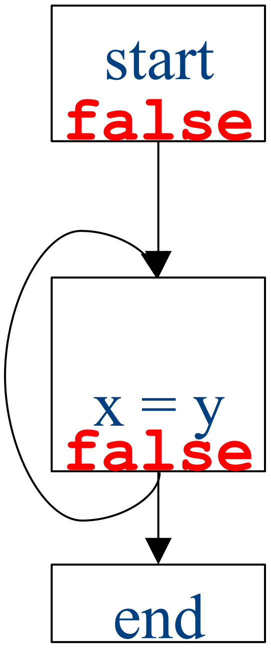
A non-terminating analysis



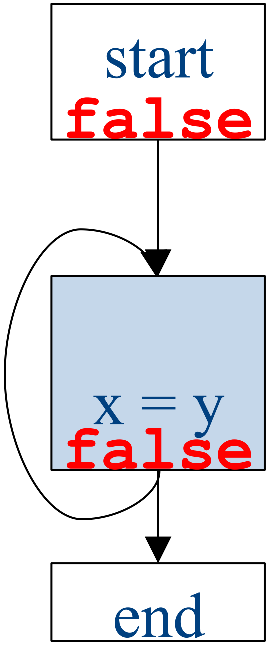
Initialization



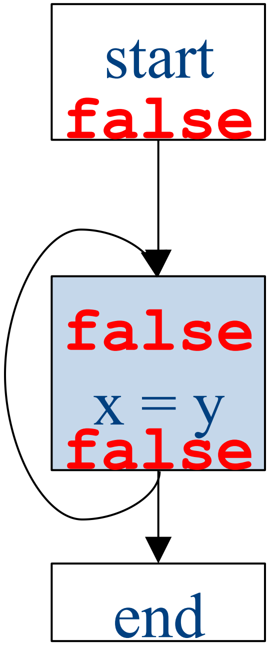
Fixed-point iteration



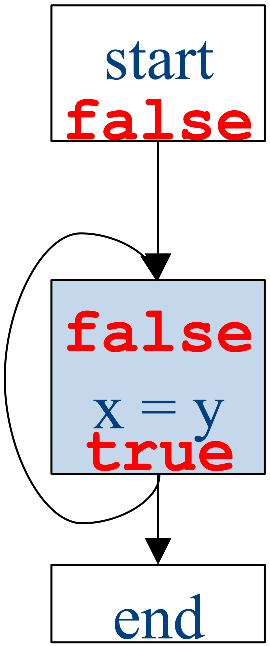
Choose a block



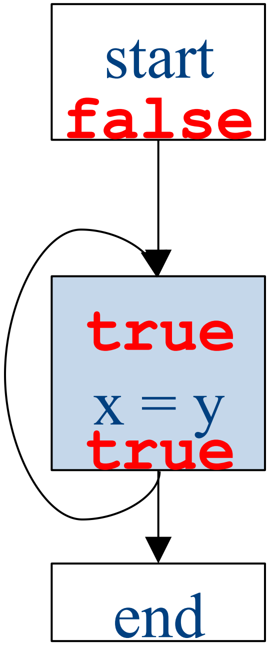
Iteration 1



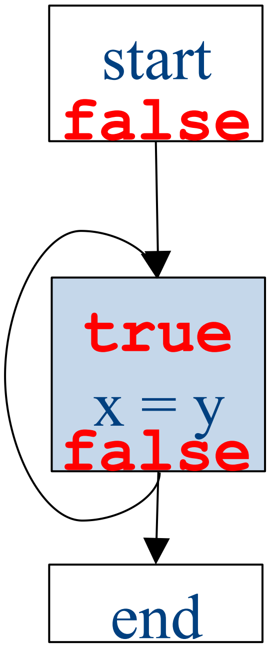
Iteration 1



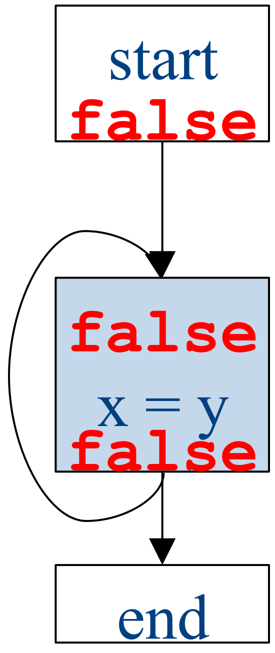
Iteration 2



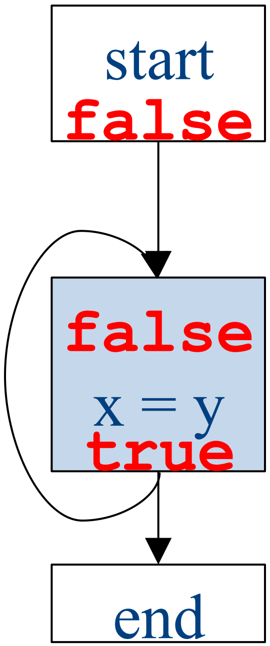
Iteration 2



Iteration 3

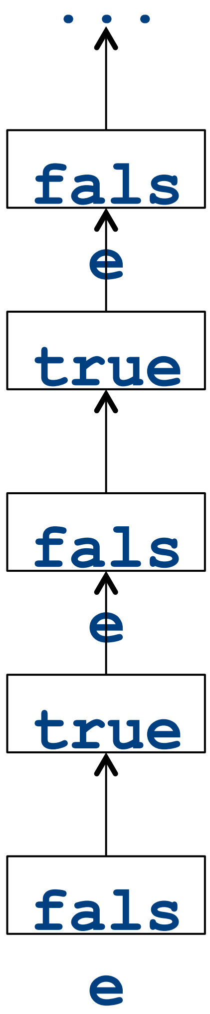


Iteration 3



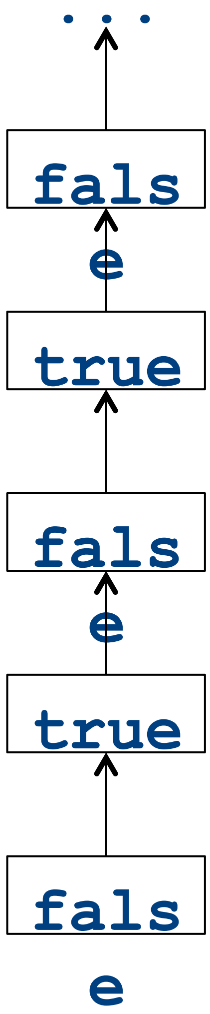
Why doesn't it terminate?

- Values can loop indefinitely
- Intuitively, the join operator keeps pulling values up
- If the transfer function can keep pushing values back down again, then the values might cycle forever



Why doesn't it terminate?

- Values can loop indefinitely
- Intuitively, the join operator keeps pulling values up
- If the transfer function can keep pushing values back down again, then the values might cycle forever
- How can we fix this?



Monotone transfer functions

- A transfer function *f* is monotone iff

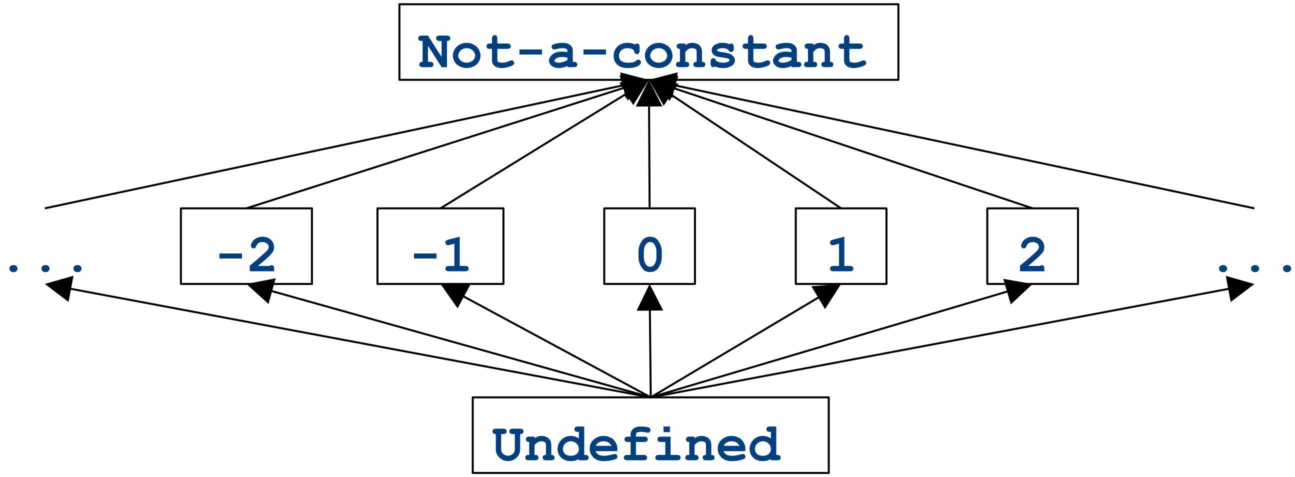
  if $x \sqsubseteq y$, then $f(x) \sqsubseteq f(y)$
- Intuitively, if you know less information about a program point, you can't "gain back" more information about that program point
- Many transfer functions are monotone, including those for liveness and constant propagation
- Note: Monotonicity does **not** mean that $x \sqsubseteq f(x)$
  - (This is a different property called extensivity)

Liveness and monotonicity

- A transfer function f is monotone iff
  if $x \sqsubseteq y$, then $f(x) \sqsubseteq f(y)$
- Recall our transfer function for **a = b + c** is
  - $f_{a = b + c}(V) = (V - \{a\}) \cup \{b, c\}$
- Recall that our join operator is set union and induces an ordering relationship
  $X \sqsubseteq Y$ iff $X \subseteq Y$
- Is this monotone?

Is constant propagation monotone?

- A transfer function *f* is monotone iff
  if $x \sqsubseteq y$, then $f(x) \sqsubseteq f(y)$
- Recall our transfer functions
  - $f_{\mathbf{x=k}}(V) = V|_{x \mapsto k}$ *(update V by mapping x to k)*
  - $f_{\mathbf{x=a+b}}(V) = V|_{x \mapsto \text{Not-a-Constant}}$ *(assign Not-a-Constant)*
- Is this monotone?



The grand result

- Theorem: A dataflow analysis with a **finite-height semilattice** and family of **monotone transfer functions** always terminates
- Proof sketch:
  - The join operator can only bring values up
  - Transfer functions can never lower values back down below where they were in the past (monotonicity)
  - Values cannot increase indefinitely (finite height)

An "optimality" result

- A transfer function f is distributive if
  $$f(a \sqcup b) = f(a) \sqcup f(\mathrm{b})$$
  for every domain elements *a* and *b*
- If all transfer functions are distributive then the fixed-point solution is the solution that would be computed by joining results from all (potentially infinite) control-flow paths
  - Join over all paths
- Optimal if we ignore program conditions

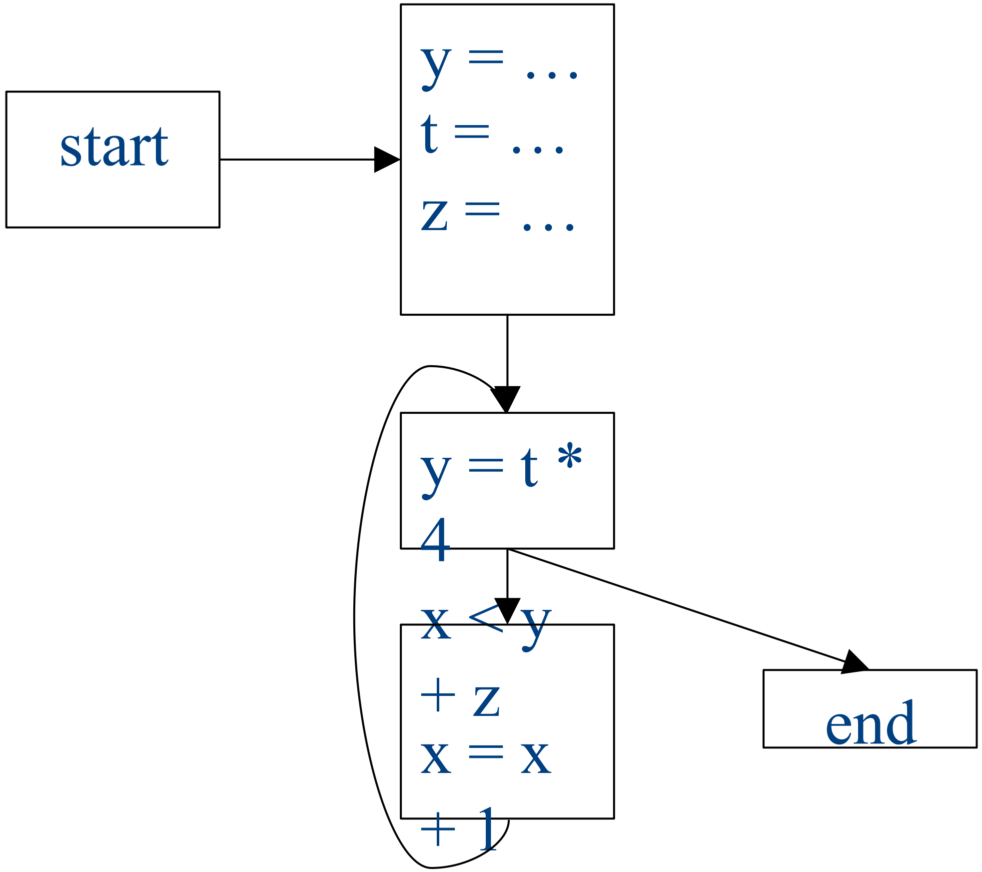
An "optimality" result

- A transfer function *f* is distributive if
  $$f(a \sqcup b) = f(a) \sqcup f(\mathrm{b})$$
  for every domain elements *a* and *b*
- If all transfer functions are distributive then the fixed-point solution is equal to the solution computed by joining results from all (potentially infinite) control-flow paths
  - Join over all paths
- Optimal if we pretend all control-flow paths can be executed by the program
- Which analyses use distributive functions?

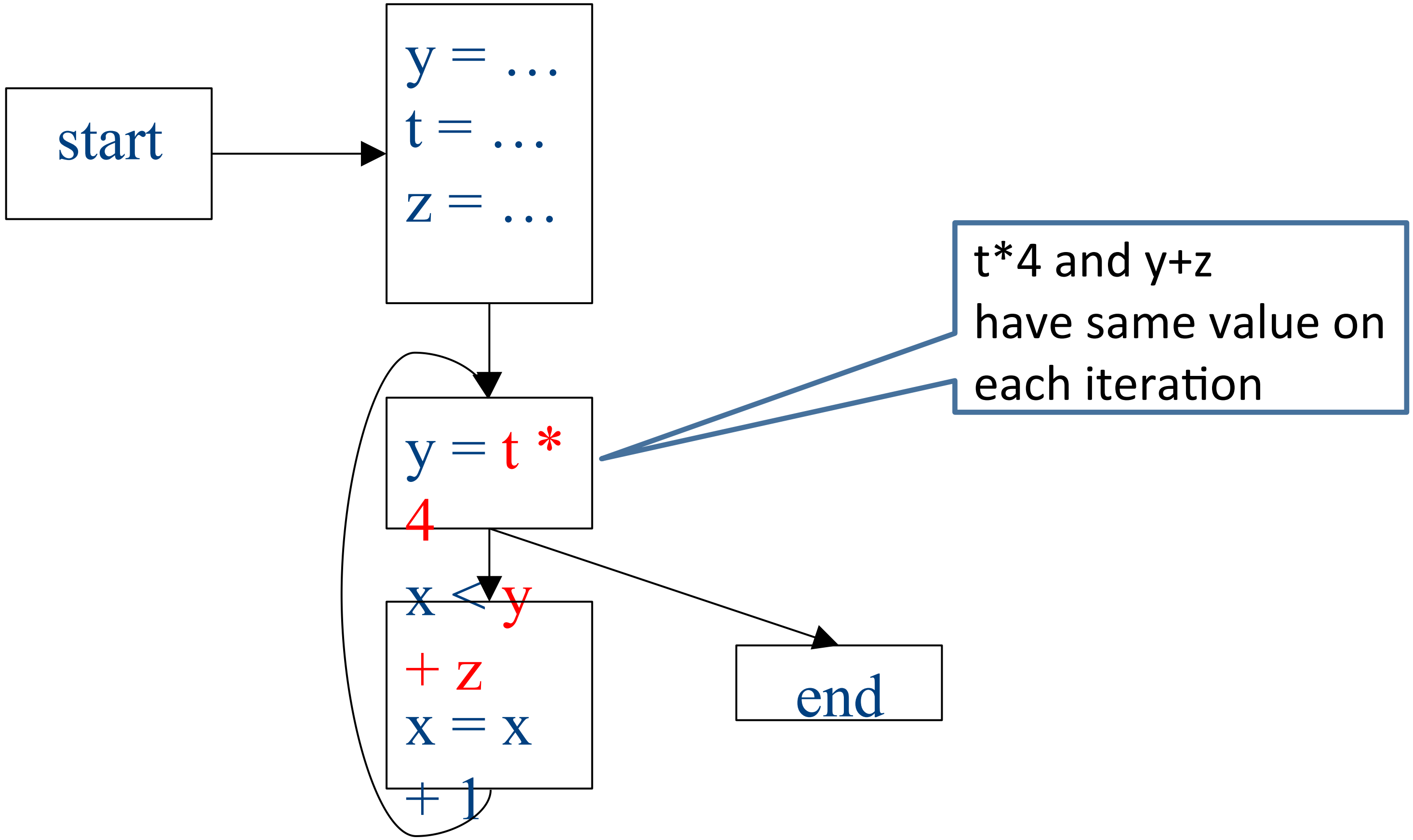
Loop optimizations

- Most of a program's computations are done inside loops
  - Focus optimizations effort on loops
- The optimizations we've seen so far are independent of the control structure
- Some optimizations are specialized to loops
  - Loop-invariant code motion
  - (Strength reduction via induction variables)
- Require another type of analysis to find out where expressions get their values from
  - Reaching definitions
    - (Also useful for improving register allocation)

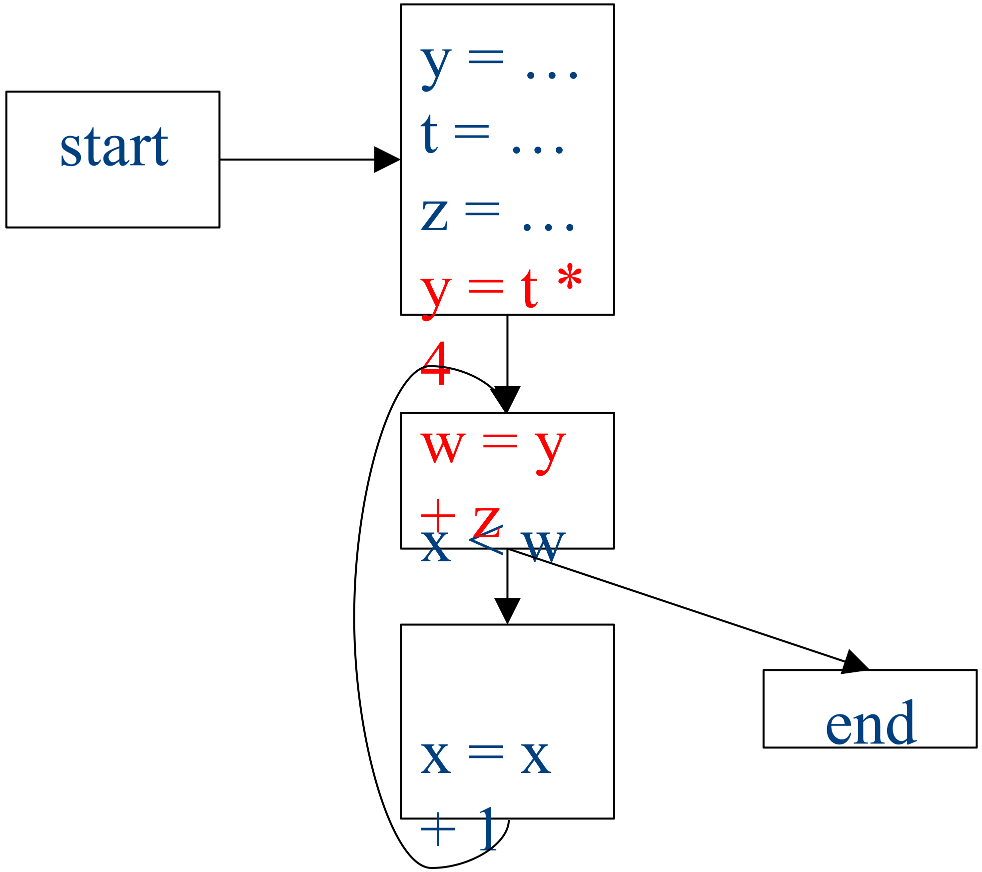
Loop invariant computation



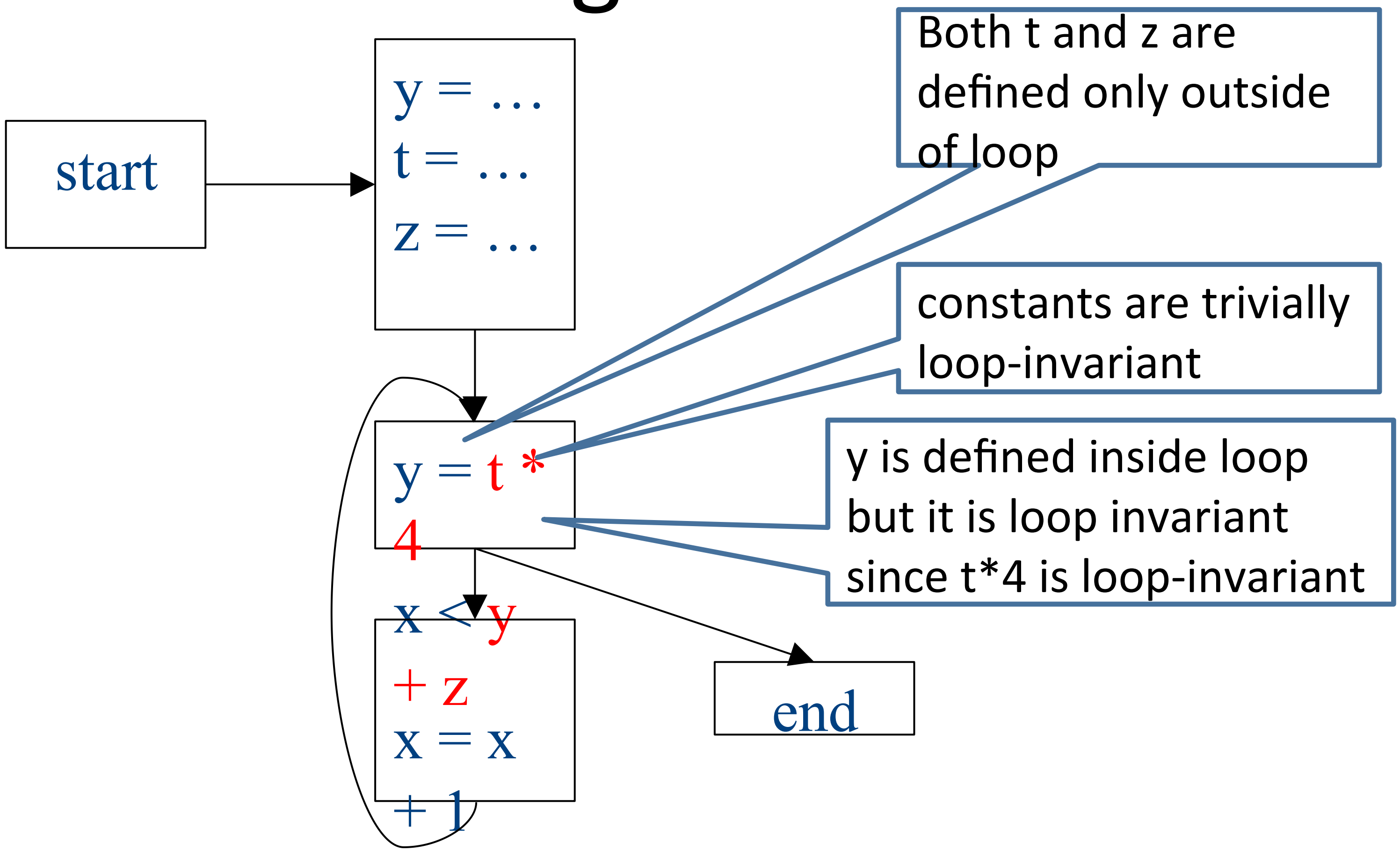
Loop invariant computation



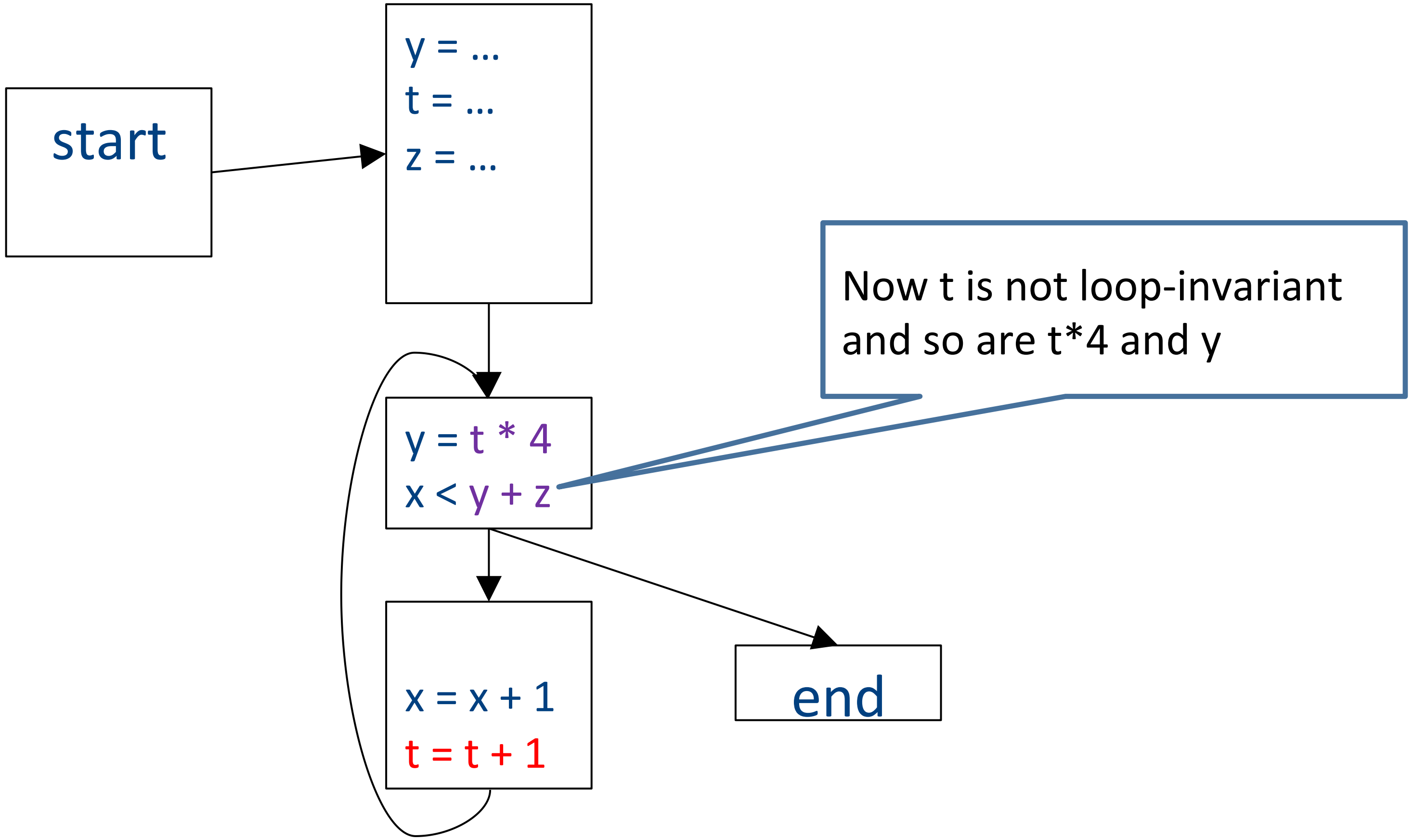
Code hoisting



What reasoning did we use?



What about now?



Loop-invariant code motion

- *d*: t = a_1 op a_2
  - *d* is a program location
- a_1 op a_2 loop-invariant (for a loop *L*) if computes the same value in each iteration
  - Hard to know in general
- Conservative approximation
  - Each a_i is a constant, or
  - All definitions of a_i that reach *d* are outside *L*, or
  - Only one definition of of a_i reaches *d*, and is loop-invariant itself
- Transformation: hoist the loop-invariant code outside of the loop

Reaching definitions analysis

- A definition *d*: t = … reaches a program location if there is a path from the definition to the program location, along which the defined variable is never redefined

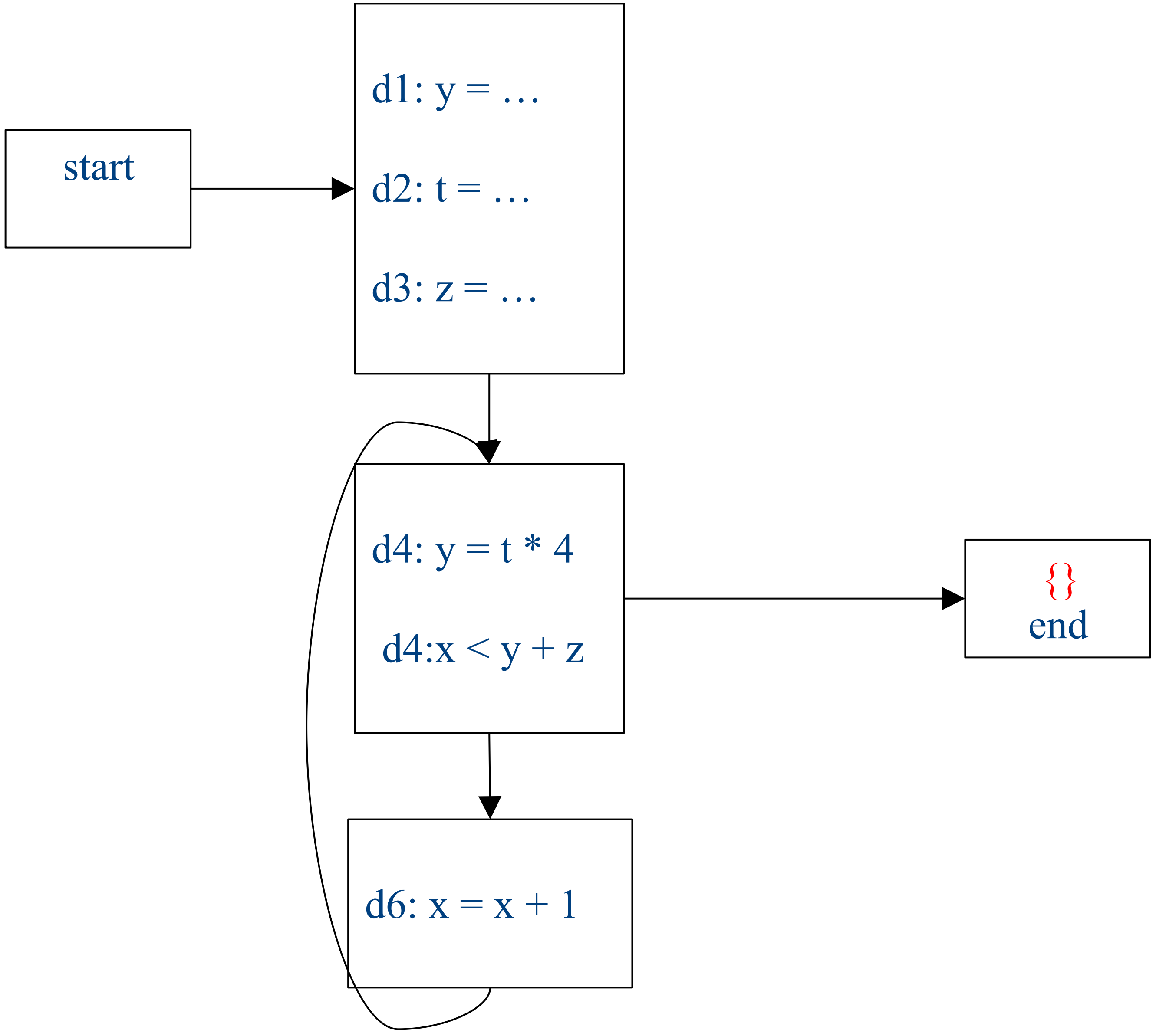
Reaching definitions analysis

- A definition *d*: t = … reaches a program location if there is a path from the definition to the program location, along which the defined variable is never redefined
- Direction: Forward
- Domain: sets of program locations that are definitions `
- Join operator: union
- Transfer function:

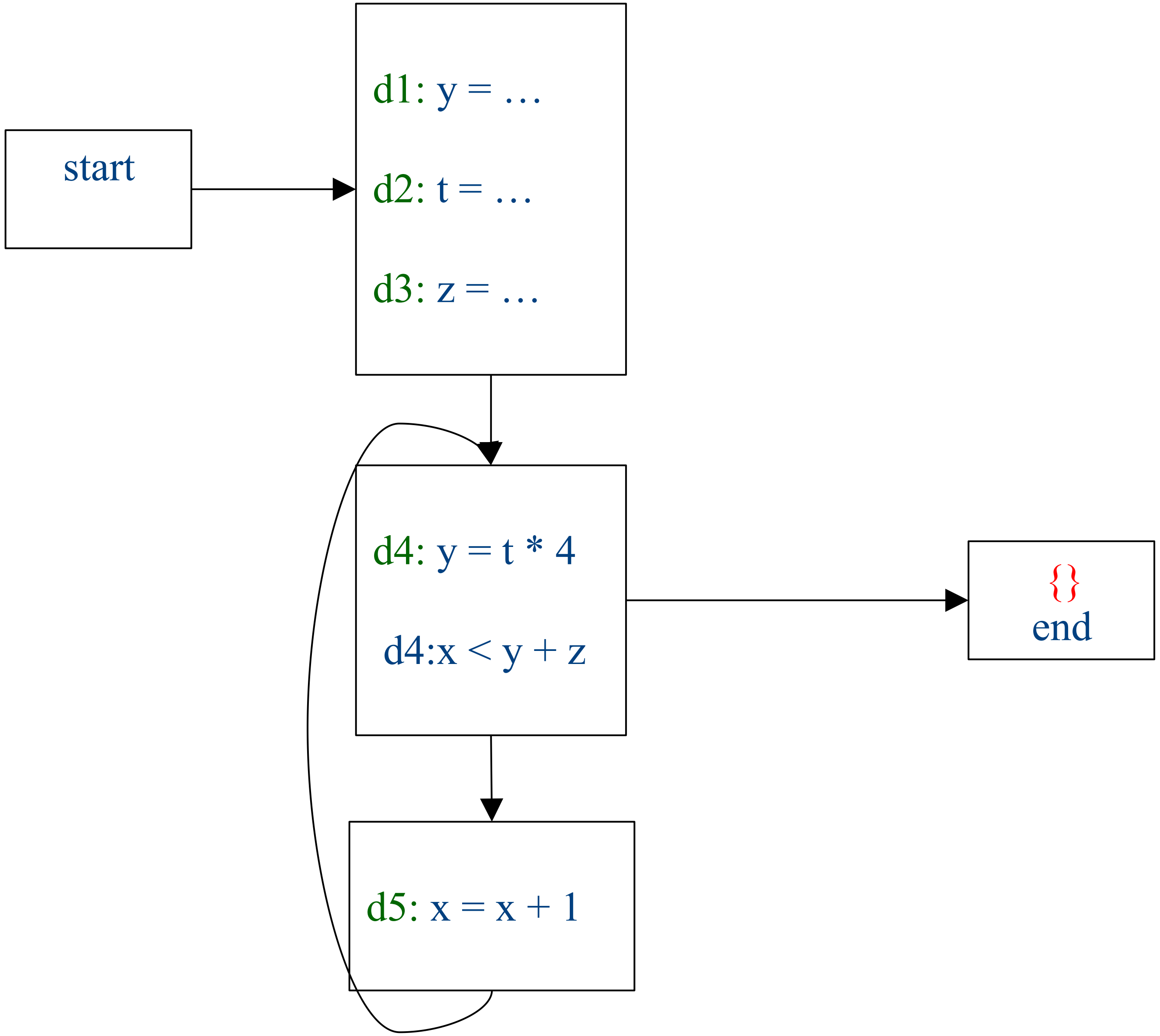
$$f_{d:\ a=b\ op\ c}(RD) = (RD - defs(a)) \cup \{d\}$$
$$f_{d:\ not\text{-}a\text{-}def}(RD) = RD$$

  - Where *defs*(*a*) is the set of locations defining *a* (statements of the form *a*=...)
- Initial value: {}

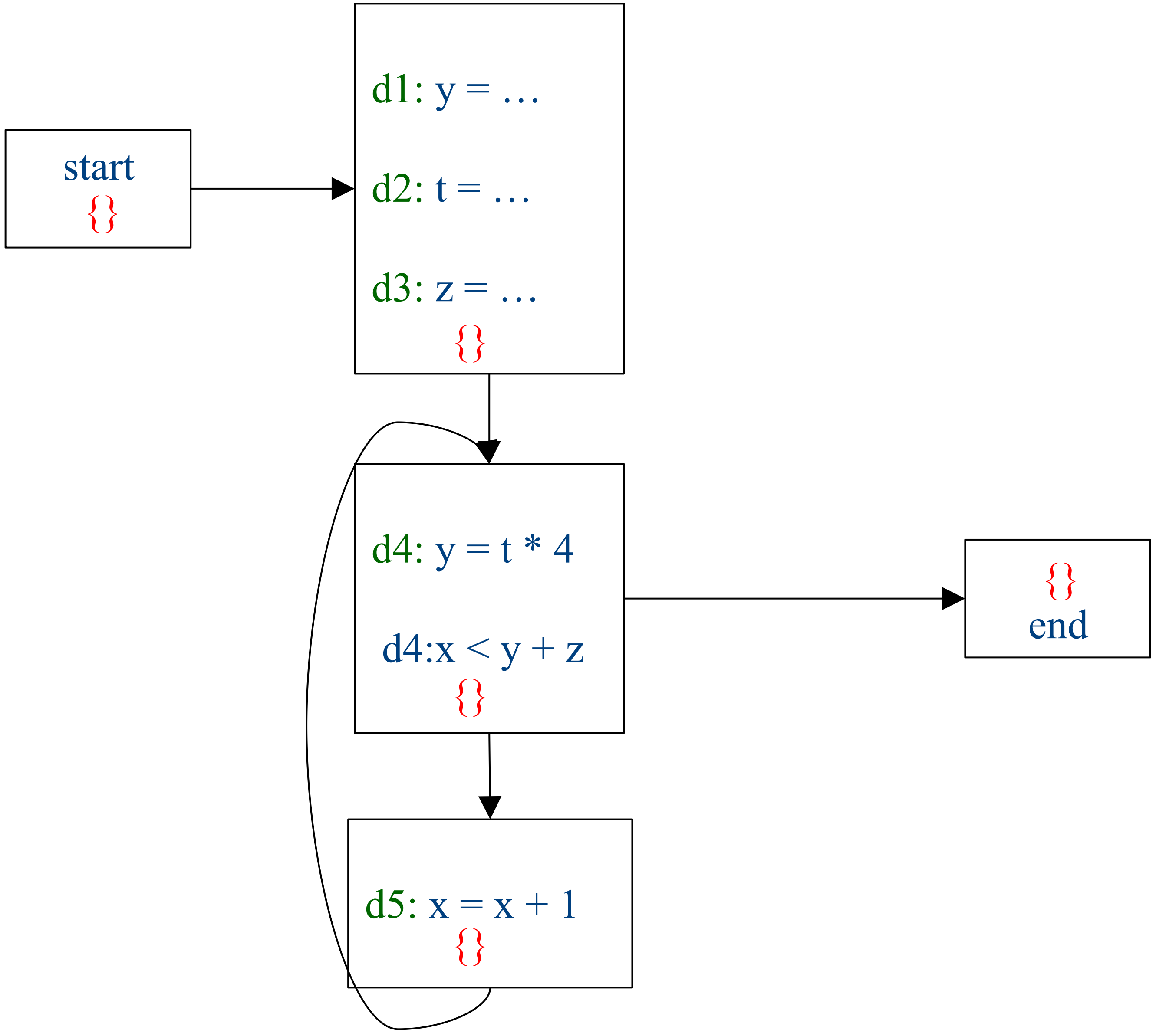
Reaching definitions analysis



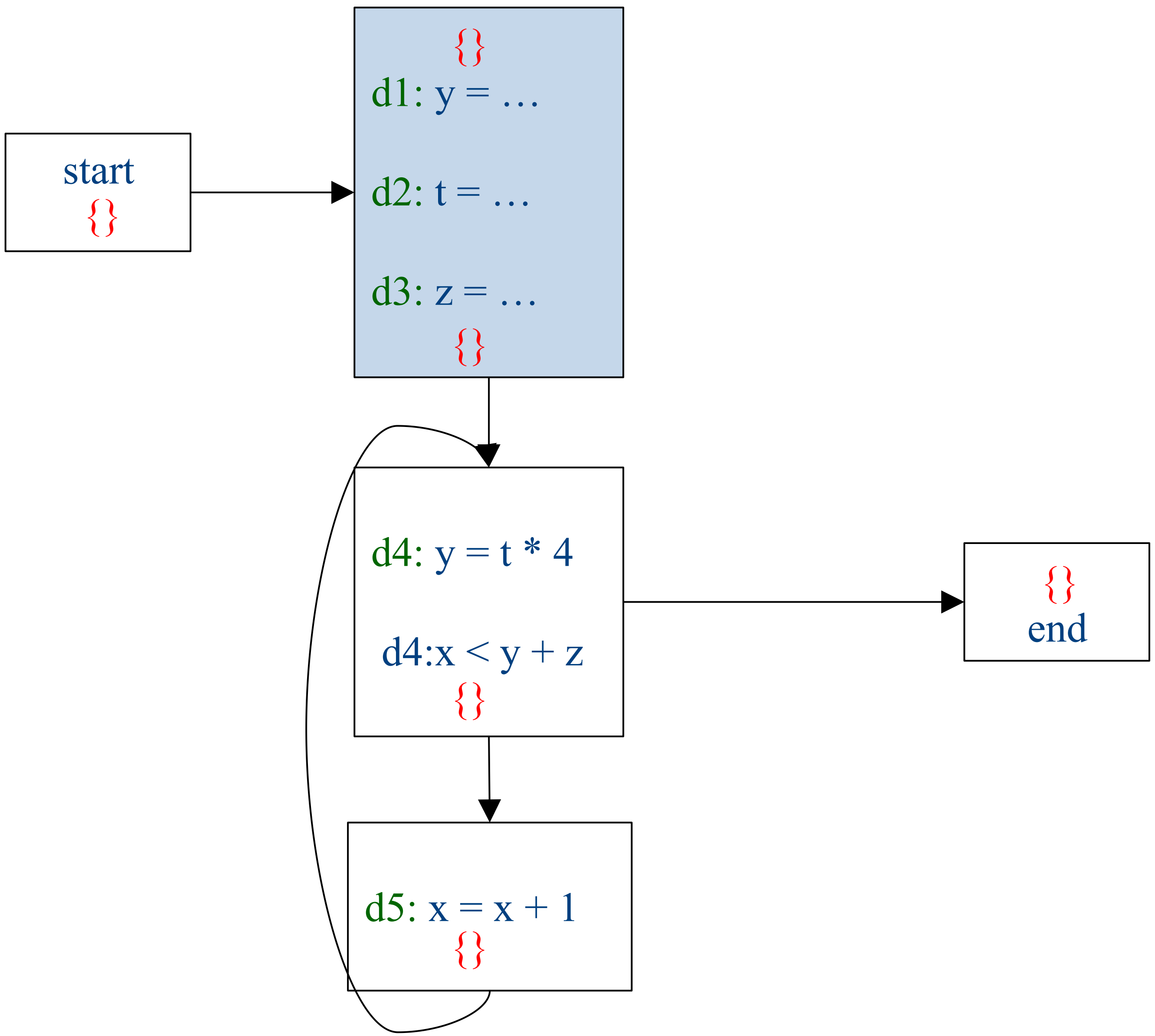
Reaching definitions analysis



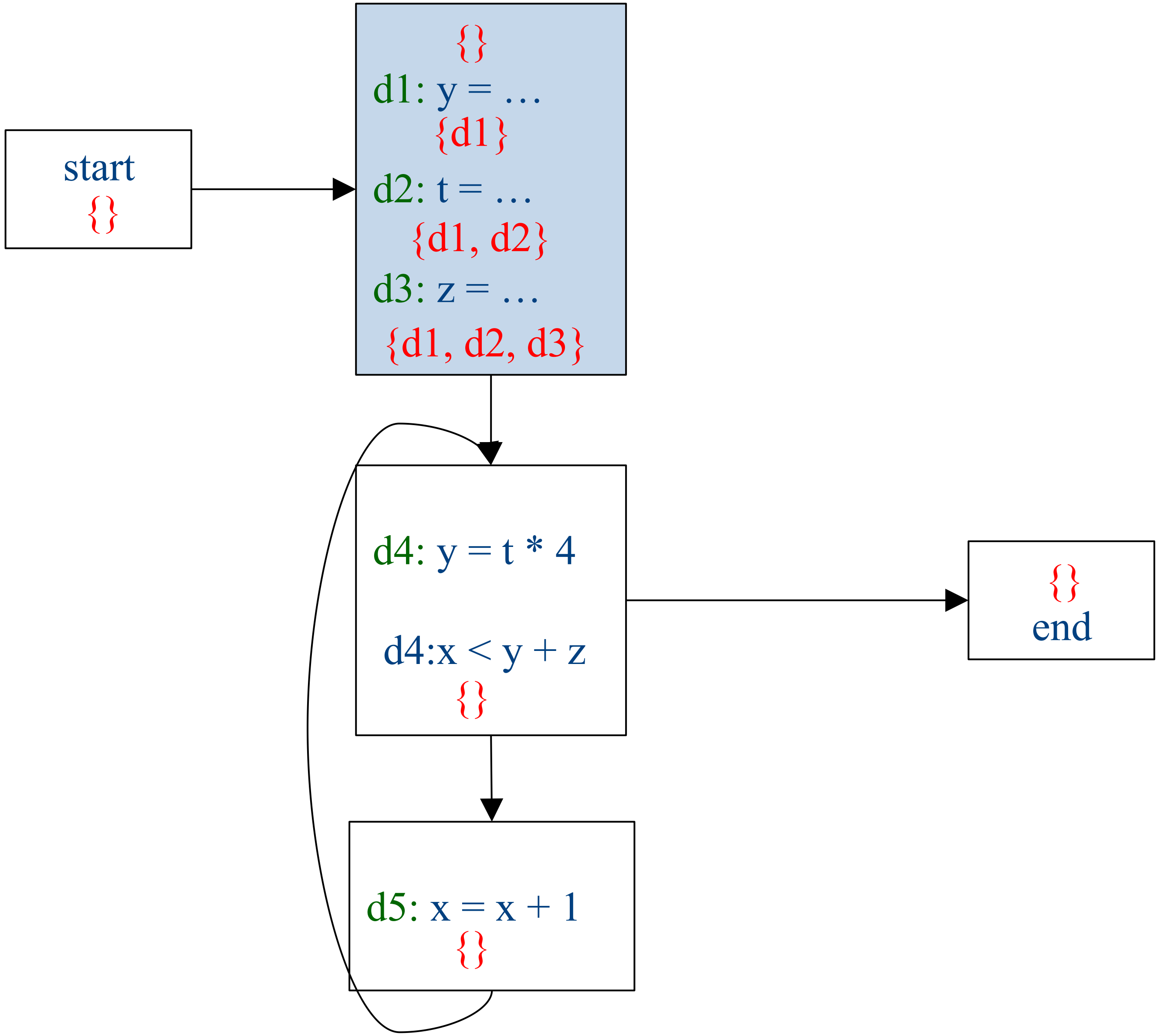
Initialization



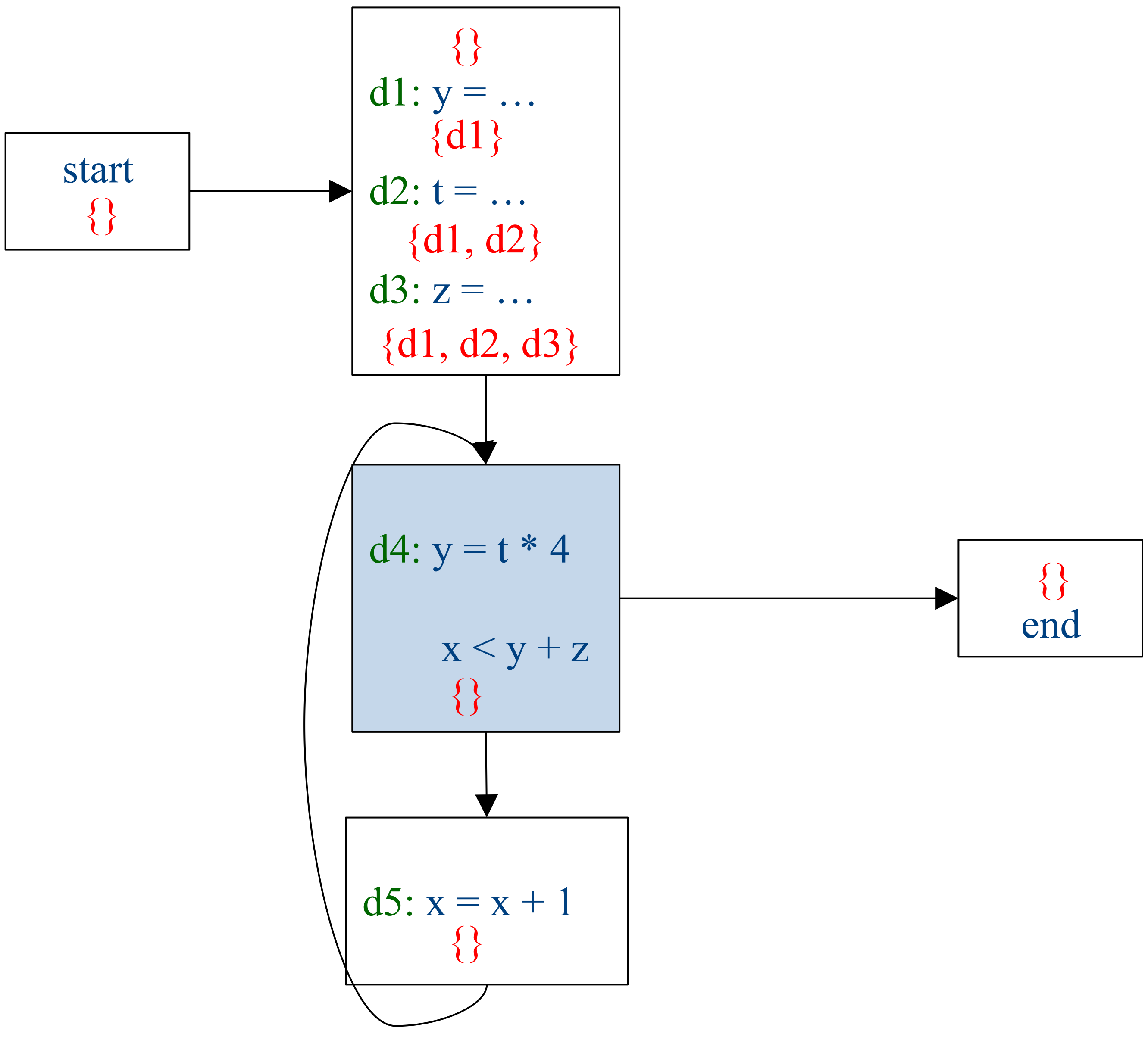
Iteration 1



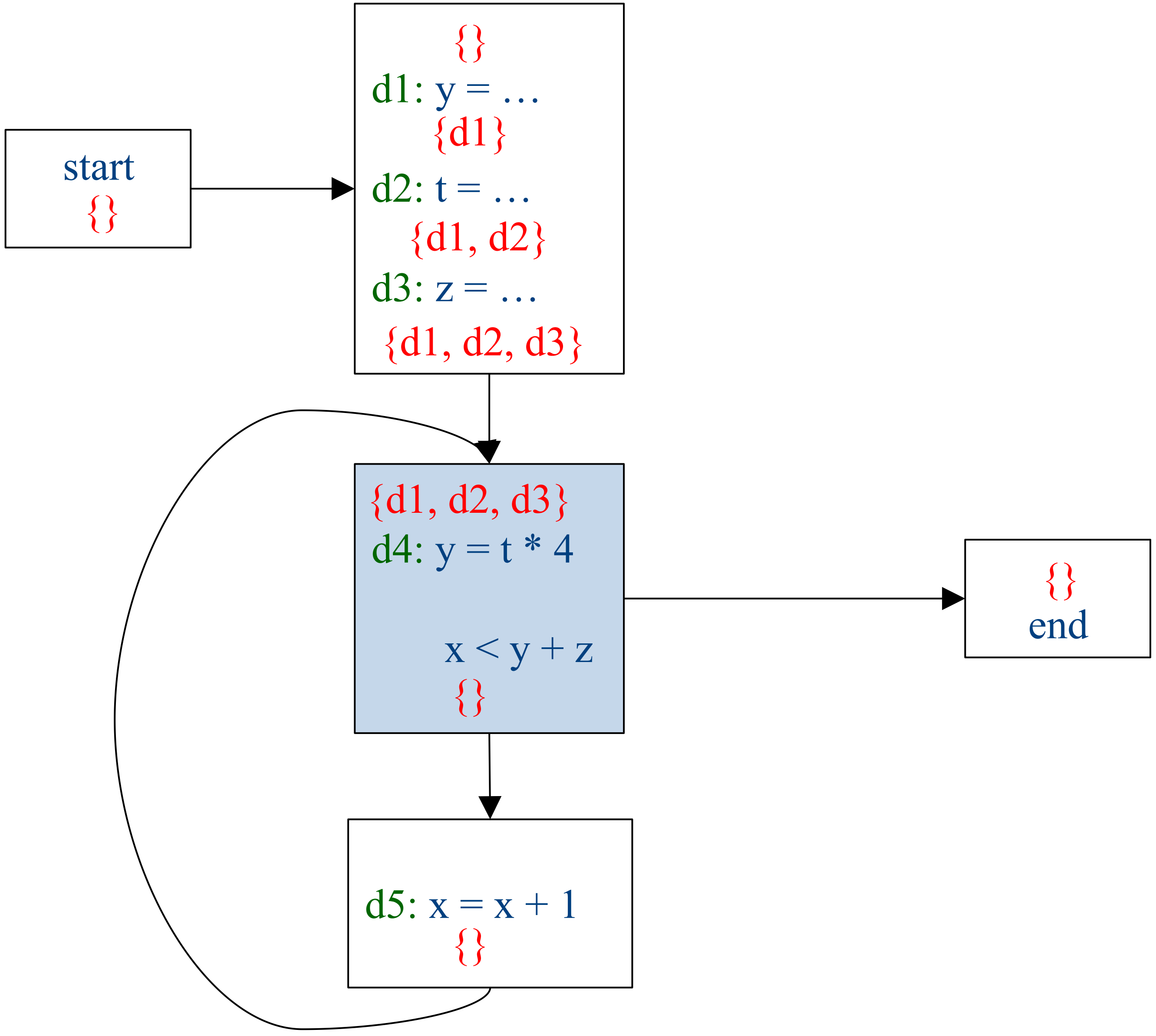
Iteration 1



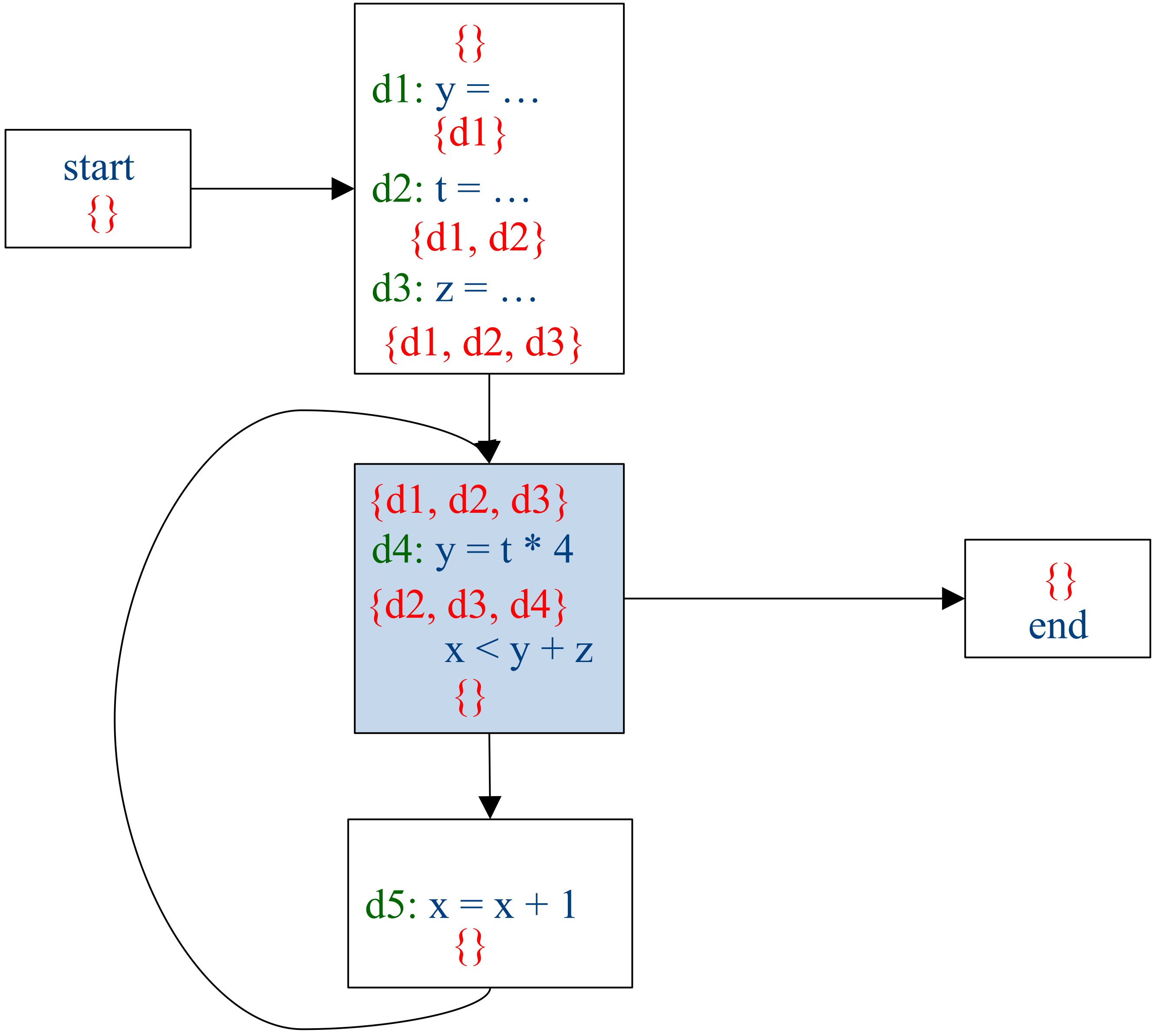
Iteration 2



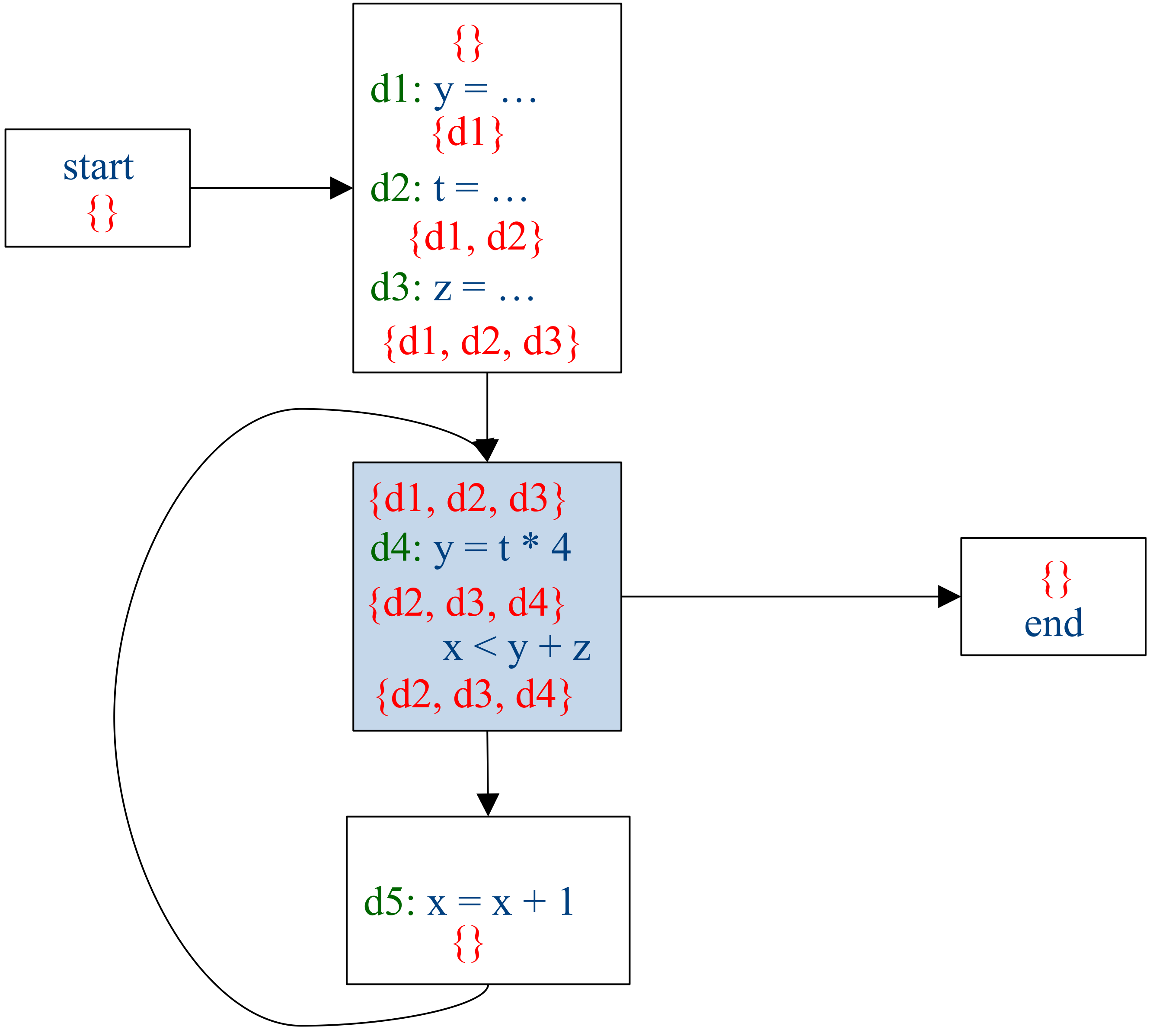
Iteration 2



Iteration 2

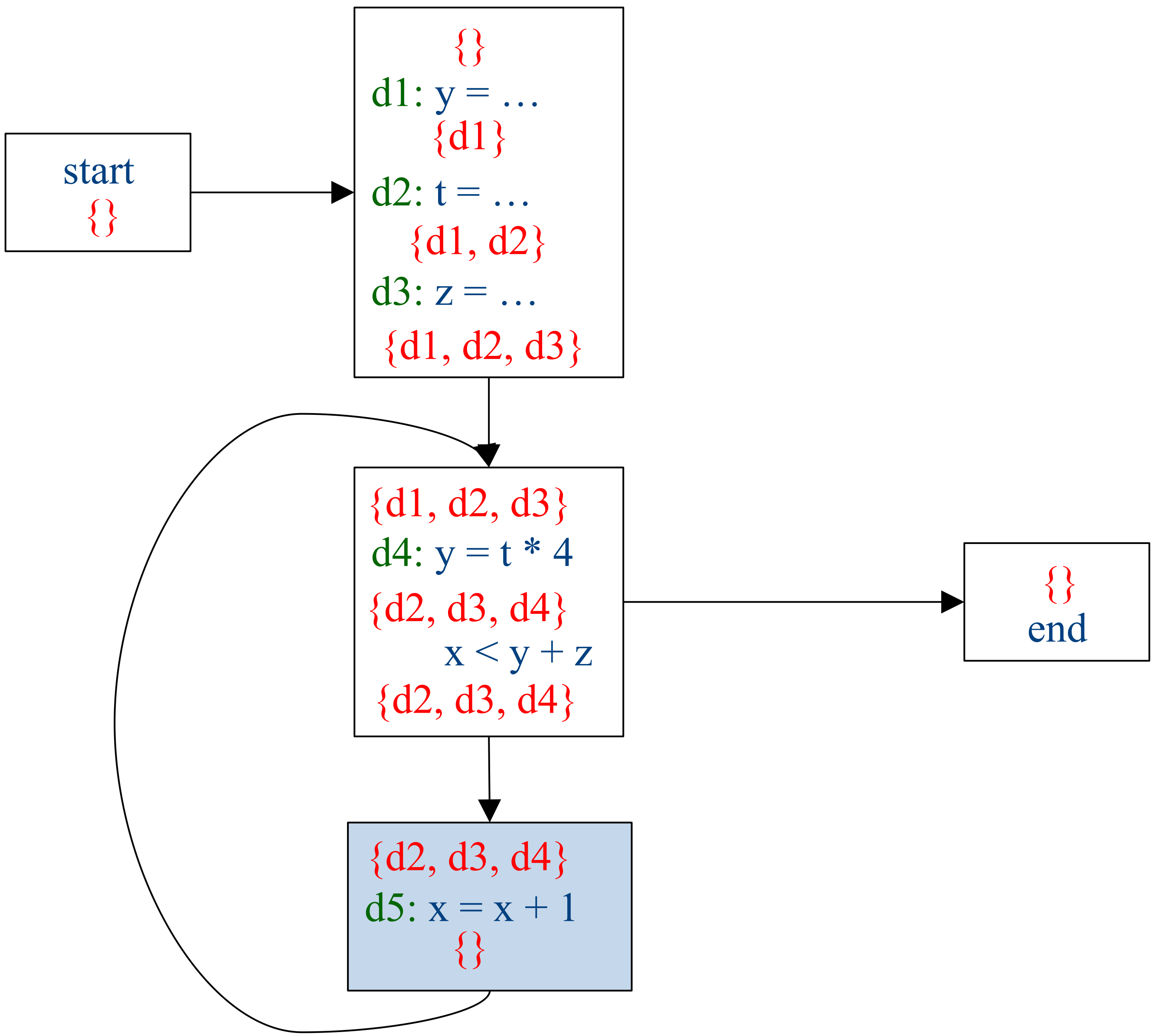

start
{}
{}
d1: y = …
{d1}
d2: t = …
{d1, d2}
d3: z = …
{d1, d2, d3}
{d1, d2, d3}
d4: y = t * 4
{d2, d3, d4}
x < y + z
{}
{}
end
d5: x = x + 1
{}

Iteration 2

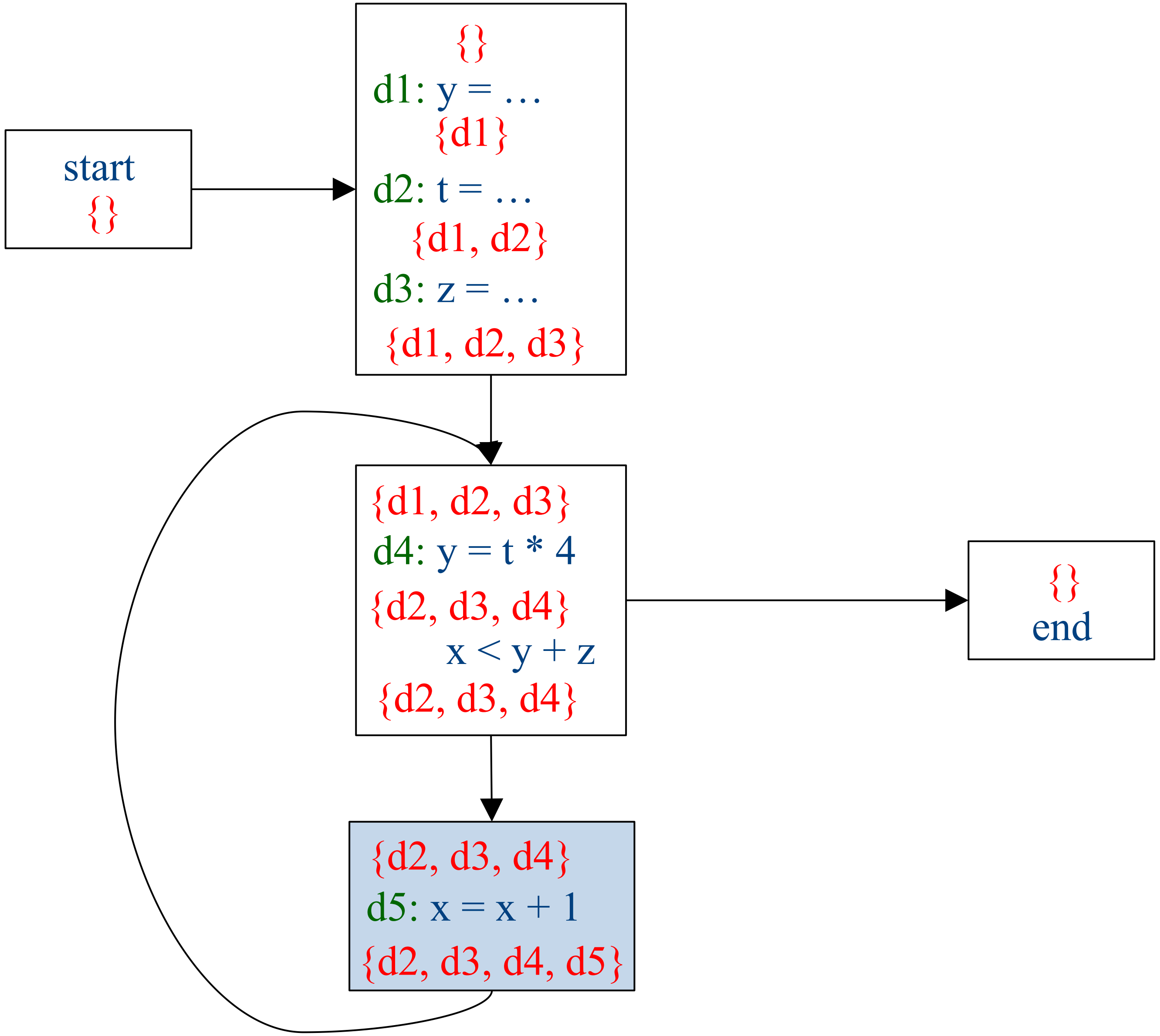


start
{}
{}
d1: y = …
{d1}
d2: t = …
{d1, d2}
d3: z = …
{d1, d2, d3}
{d1, d2, d3}
d4: y = t * 4
{d2, d3, d4}
x < y + z
{d2, d3, d4}
{}
end
d5: x = x + 1
{}

Iteration 3

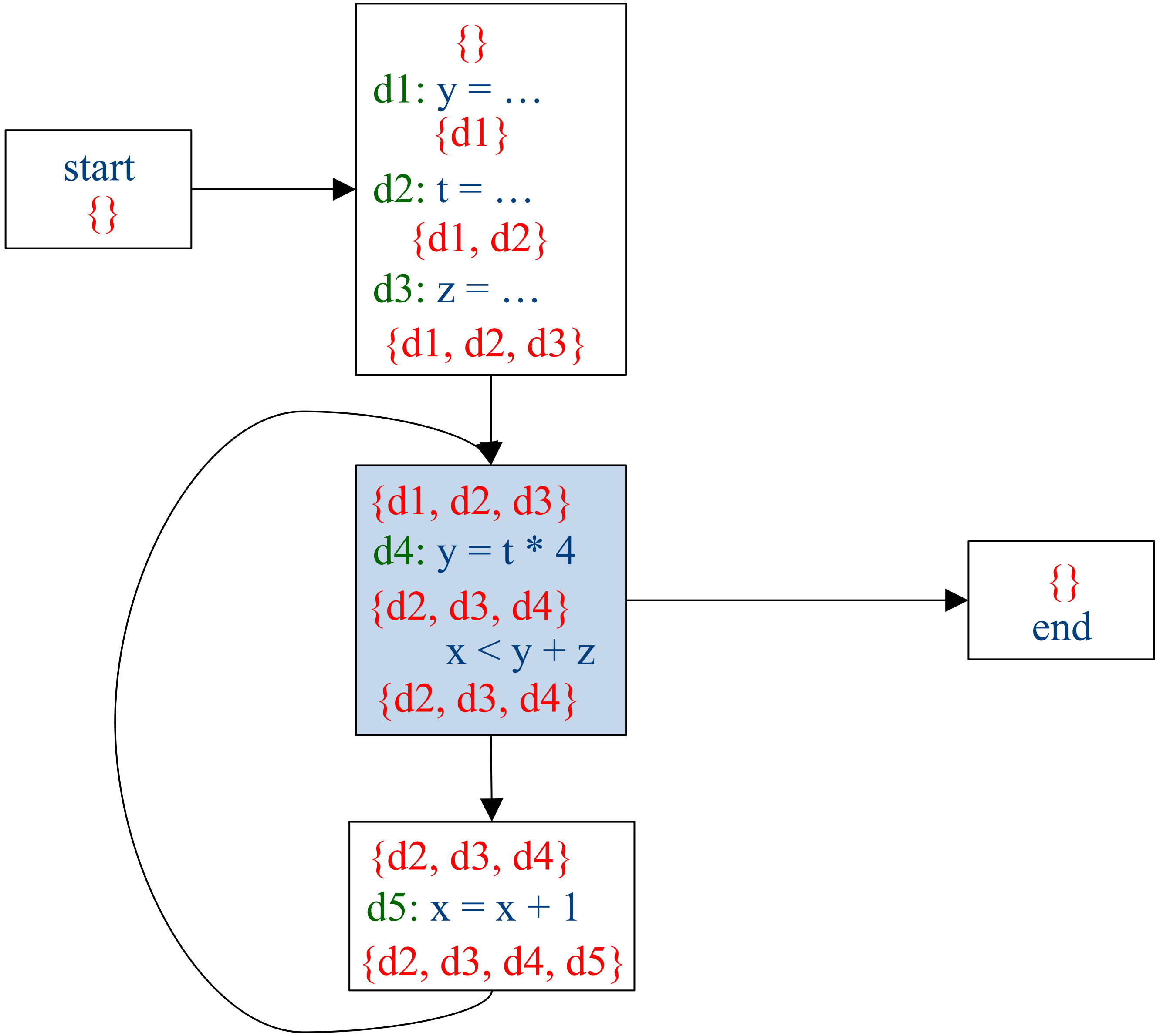

start
{}
{}
d1: y = …
{d1}
d2: t = …
{d1, d2}
d3: z = …
{d1, d2, d3}
{d1, d2, d3}
d4: y = t * 4
{d2, d3, d4}
x < y + z
{d2, d3, d4}
{}
end
{d2, d3, d4}
d5: x = x + 1
{}

Iteration 3



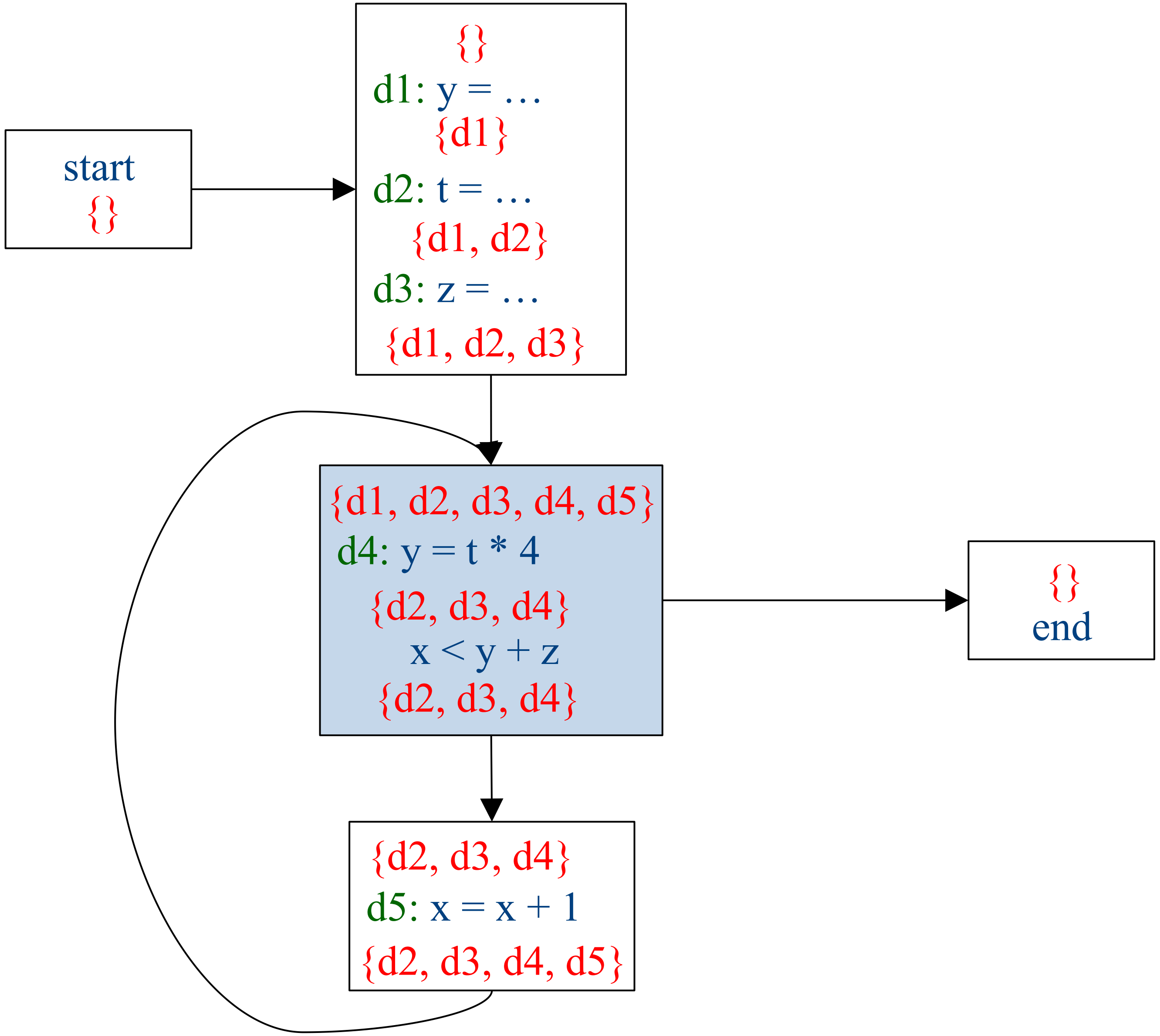
start
{}
{}
d1: y = …
{d1}
d2: t = …
{d1, d2}
d3: z = …
{d1, d2, d3}
{d1, d2, d3}
d4: y = t * 4
{d2, d3, d4}
x < y + z
{d2, d3, d4}
{}
end
{d2, d3, d4}
d5: x = x + 1
{d2, d3, d4, d5}

Iteration 4

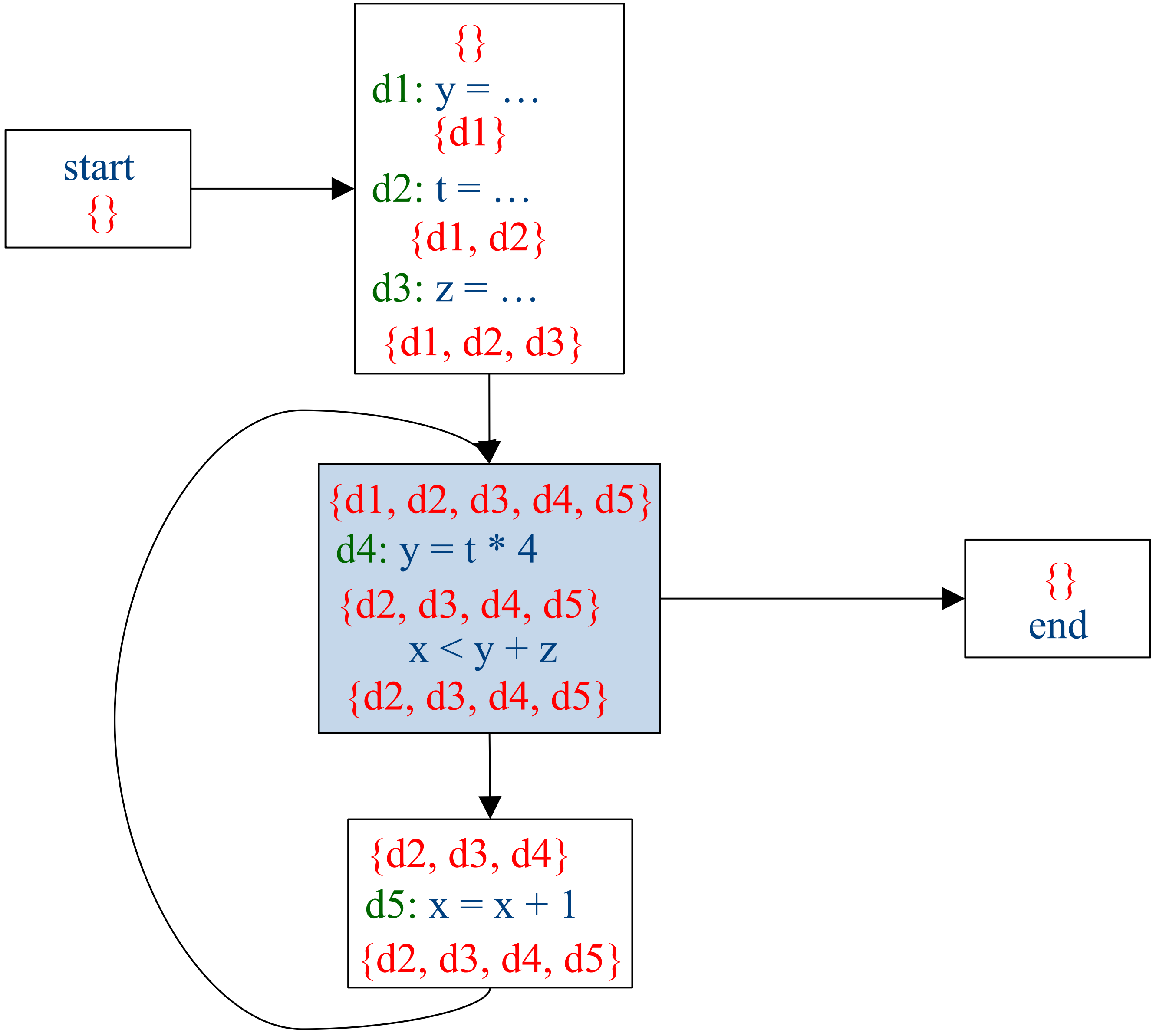


start
{}
{}
d1: y = …
{d1}
d2: t = …
{d1, d2}
d3: z = …
{d1, d2, d3}
{d1, d2, d3}
d4: y = t * 4
{d2, d3, d4}
x < y + z
{d2, d3, d4}
{}
end
{d2, d3, d4}
d5: x = x + 1
{d2, d3, d4, d5}

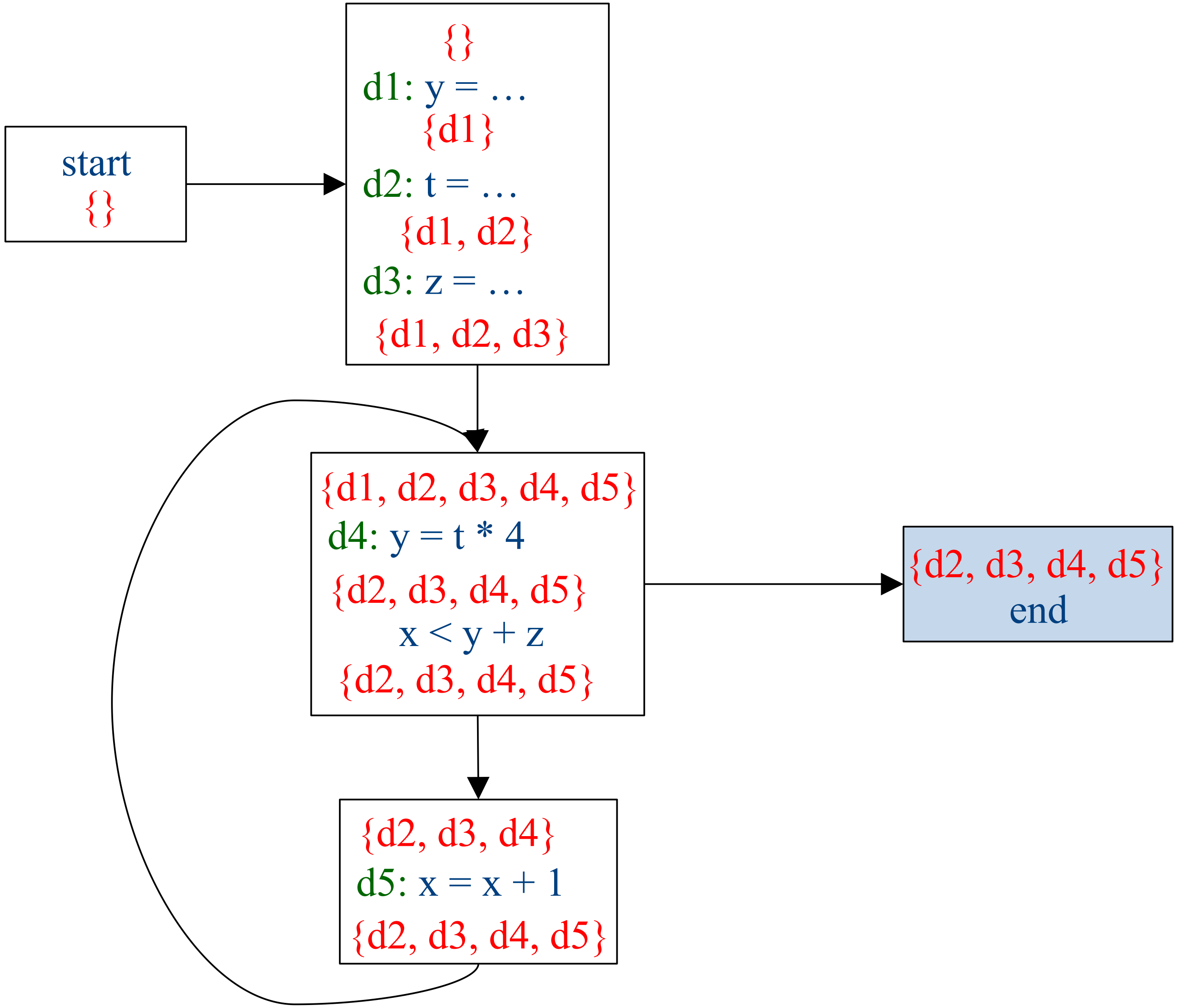
Iteration 4


start
{}
{}
d1: y = …
{d1}
d2: t = …
{d1, d2}
d3: z = …
{d1, d2, d3}
{d1, d2, d3, d4, d5}
d4: y = t * 4
{d2, d3, d4}
x < y + z
{d2, d3, d4}
{}
end
{d2, d3, d4}
d5: x = x + 1
{d2, d3, d4, d5}

Iteration 4

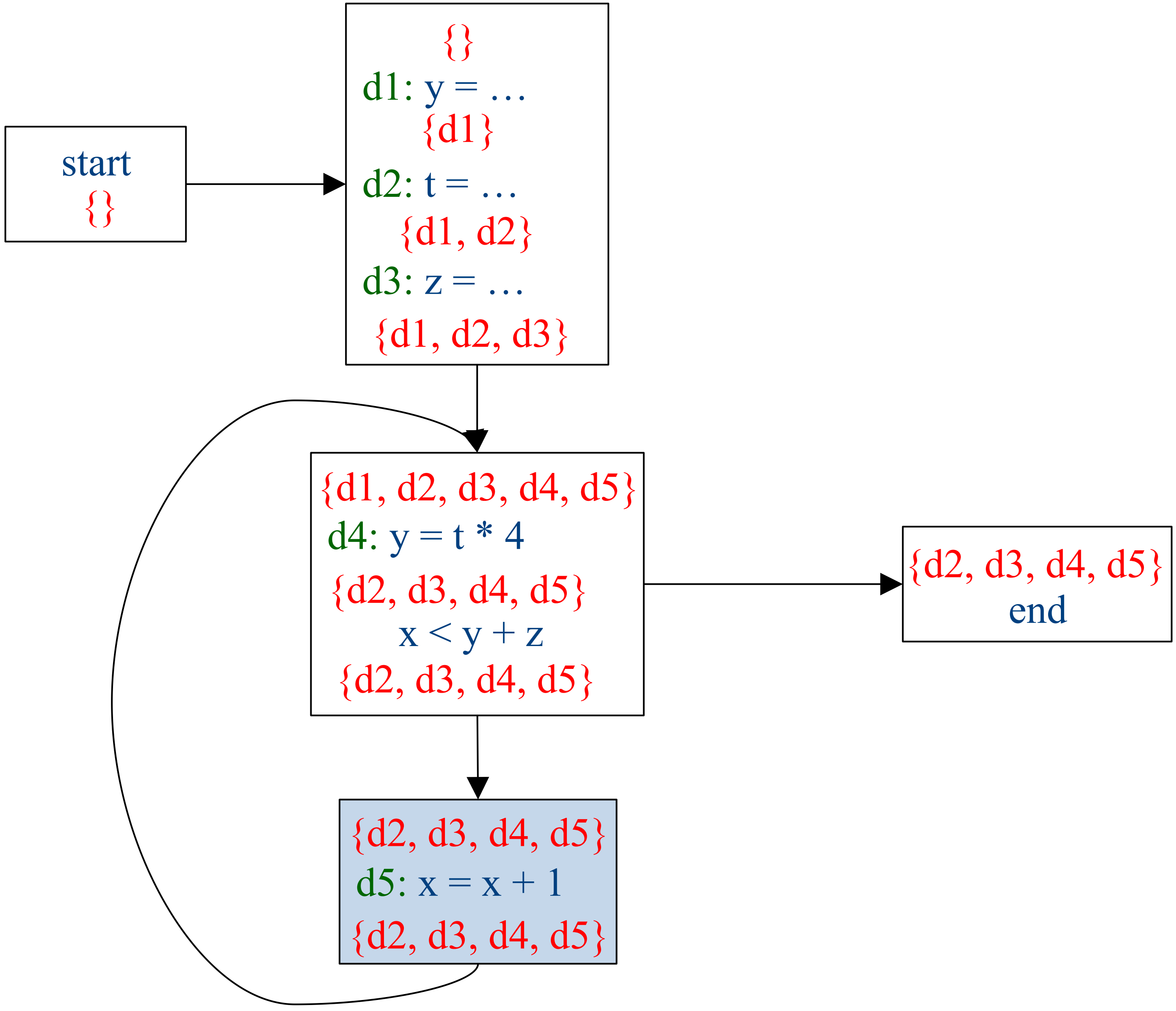

start
{}
{}
d1: y = …
{d1}
d2: t = …
{d1, d2}
d3: z = …
{d1, d2, d3}
{d1, d2, d3, d4, d5}
d4: y = t * 4
{d2, d3, d4, d5}
x < y + z
{d2, d3, d4, d5}
{}
end
{d2, d3, d4}
d5: x = x + 1
{d2, d3, d4, d5}

Iteration 5



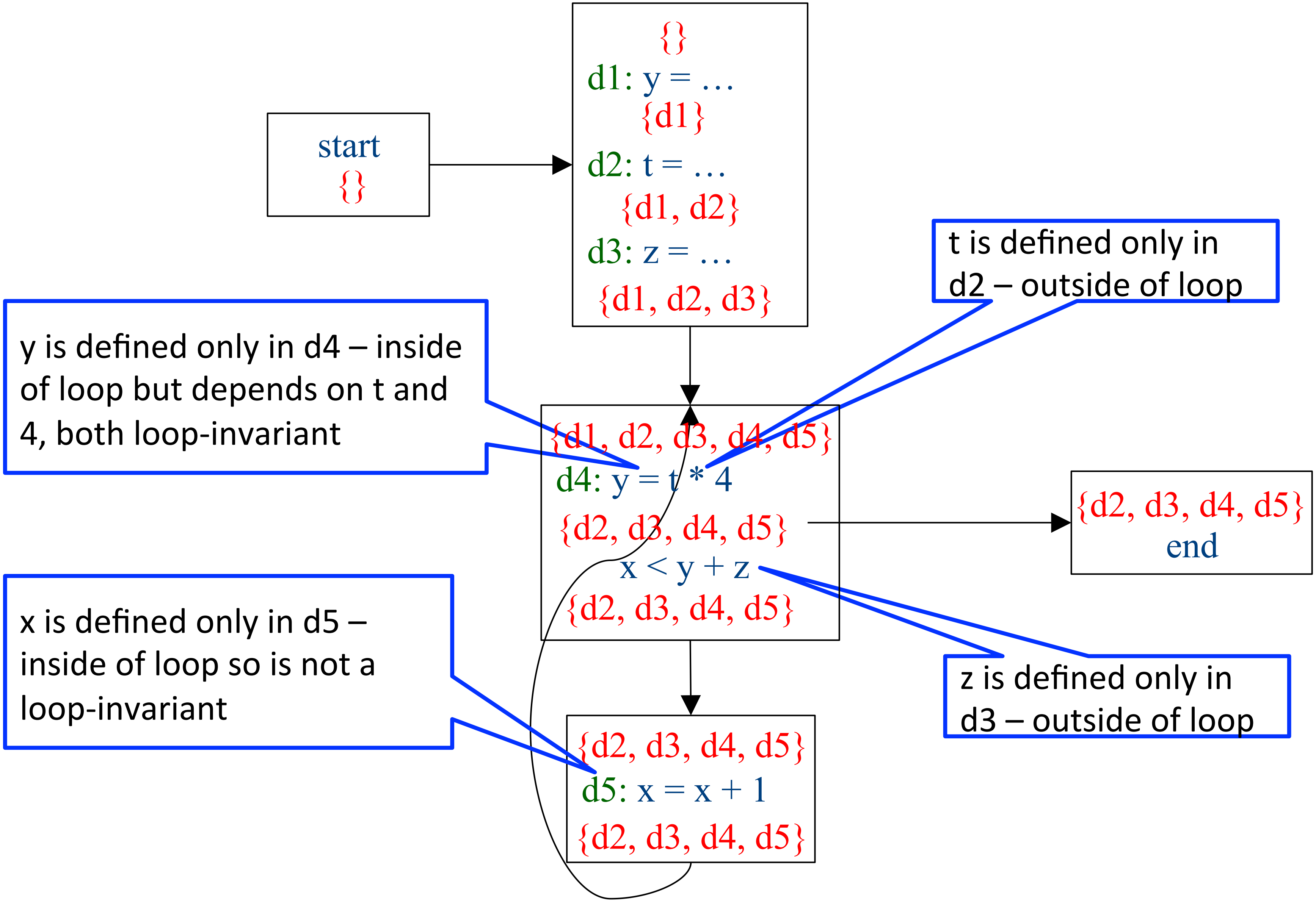
start
{}
{}
d1: y = …
{d1}
d2: t = …
{d1, d2}
d3: z = …
{d1, d2, d3}
{d1, d2, d3, d4, d5}
d4: y = t * 4
{d2, d3, d4, d5}
x < y + z
{d2, d3, d4, d5}
{d2, d3, d4, d5}
end
{d2, d3, d4}
d5: x = x + 1
{d2, d3, d4, d5}

Iteration 6



start
{}
{}
d1: y = …
{d1}
d2: t = …
{d1, d2}
d3: z = …
{d1, d2, d3}
{d1, d2, d3, d4, d5}
d4: y = t * 4
{d2, d3, d4, d5}
x < y + z
{d2, d3, d4, d5}
{d2, d3, d4, d5}
end
{d2, d3, d4, d5}
d5: x = x + 1
{d2, d3, d4, d5}

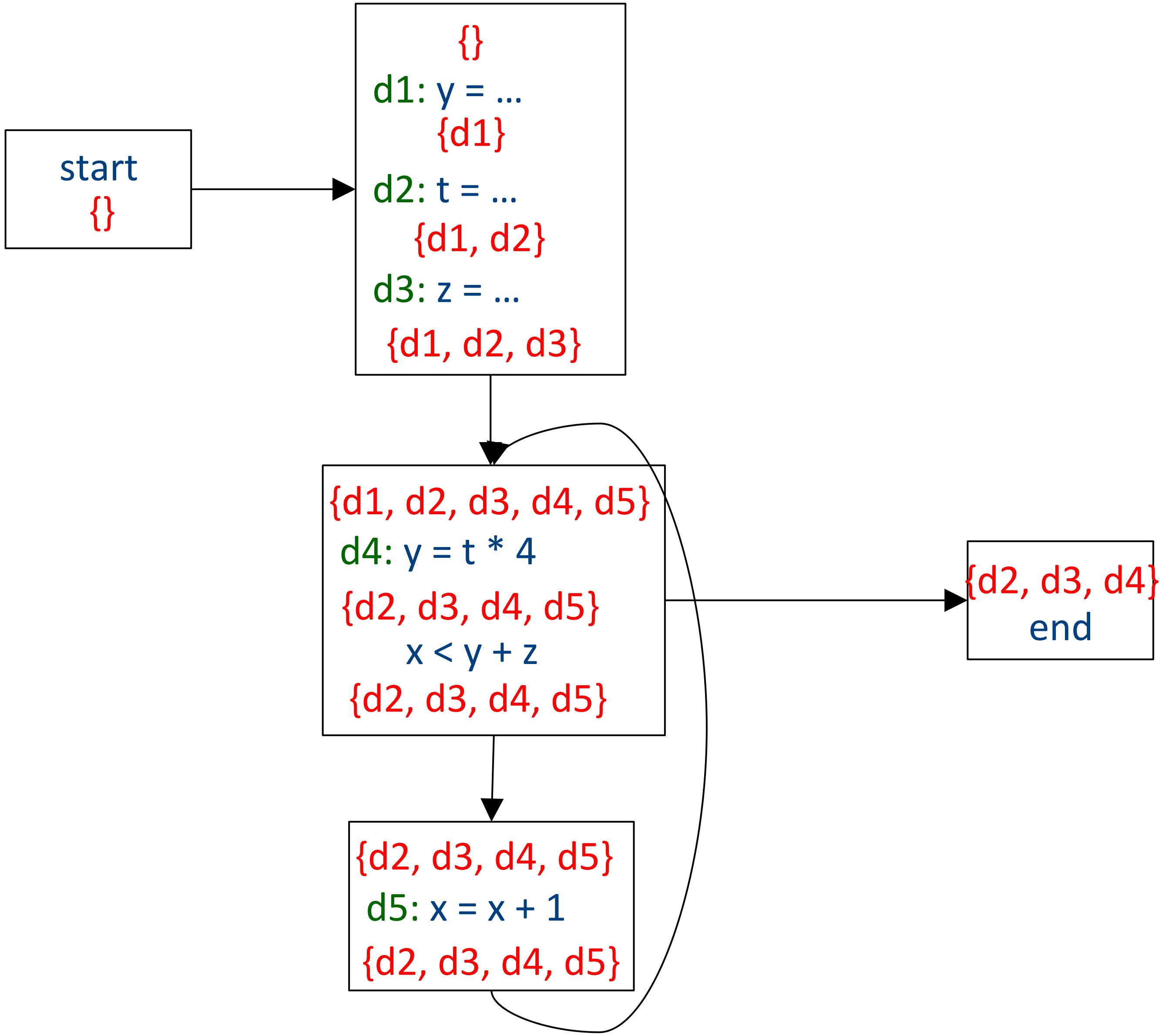
Which expressions are loop invariant?



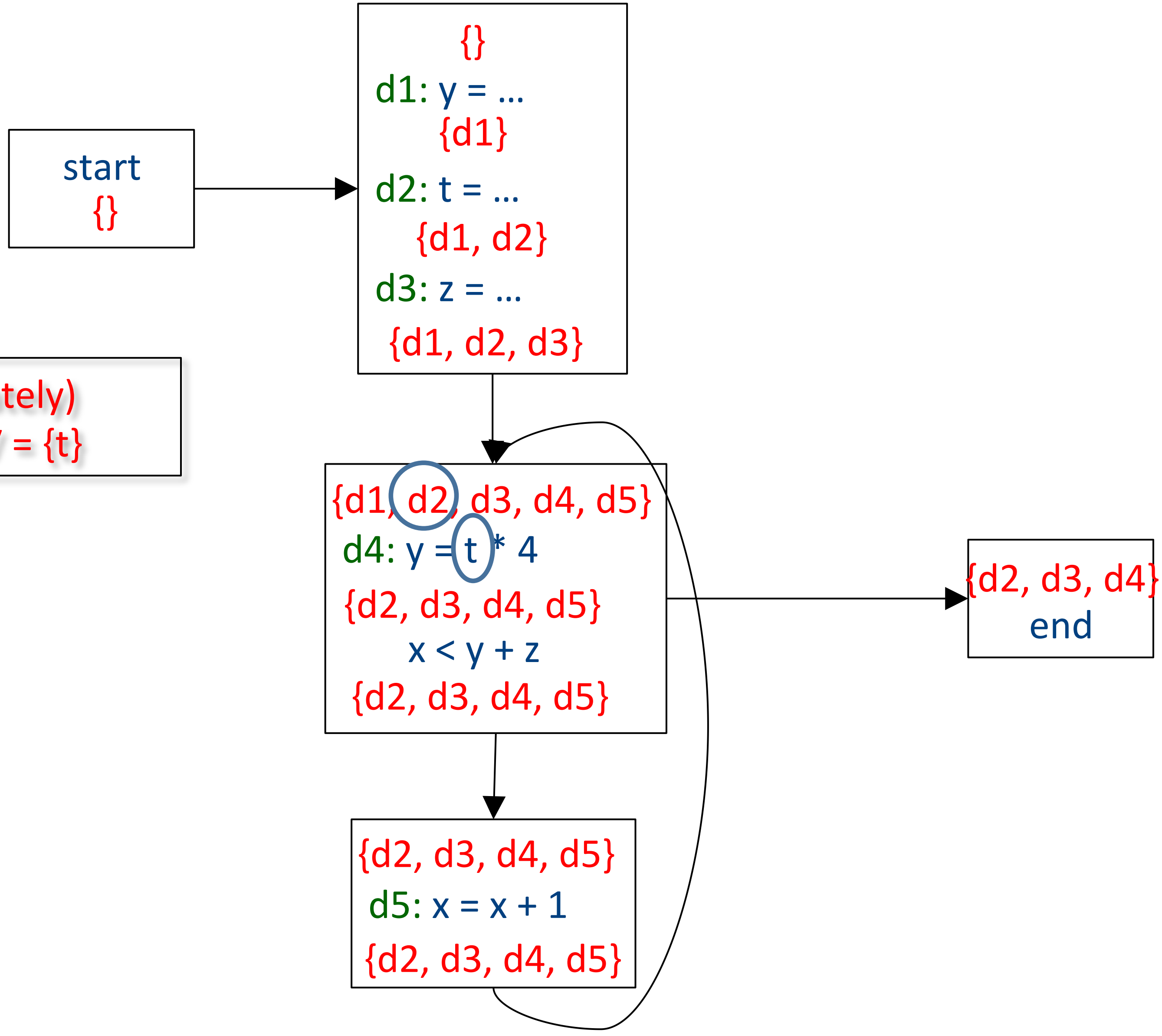
Inferring loop-invariant expressions

- For a statement *s* of the form t = a_1 op a_2
- A variable a_i is immediately loop-invariant if all reaching definitions IN[*s*]={d_1,…,d_k} for a_i are outside of the loop
- LOOP-INV = immediately loop-invariant variables and constants
  LOOP-INV = LOOP-INV ∪ {x | d: x = a_1 op a_2, d is in the loop, and both a_1 and a_2 are in LOOP-INV}
  - Iterate until fixed-point
- An expression is loop-invariant if all operands are loop-invariants

Computing LOOP-INV

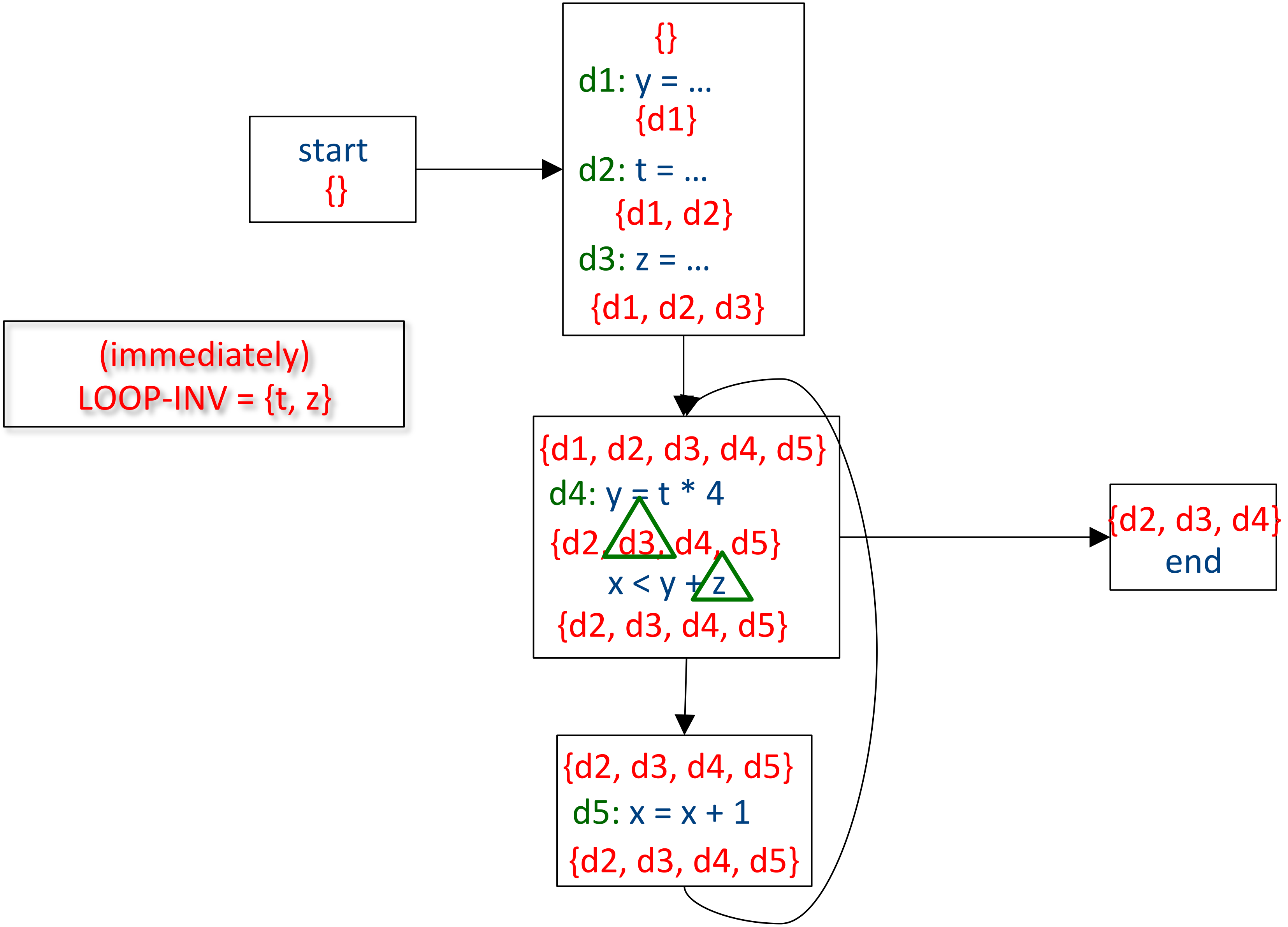


Computing LOOP-INV

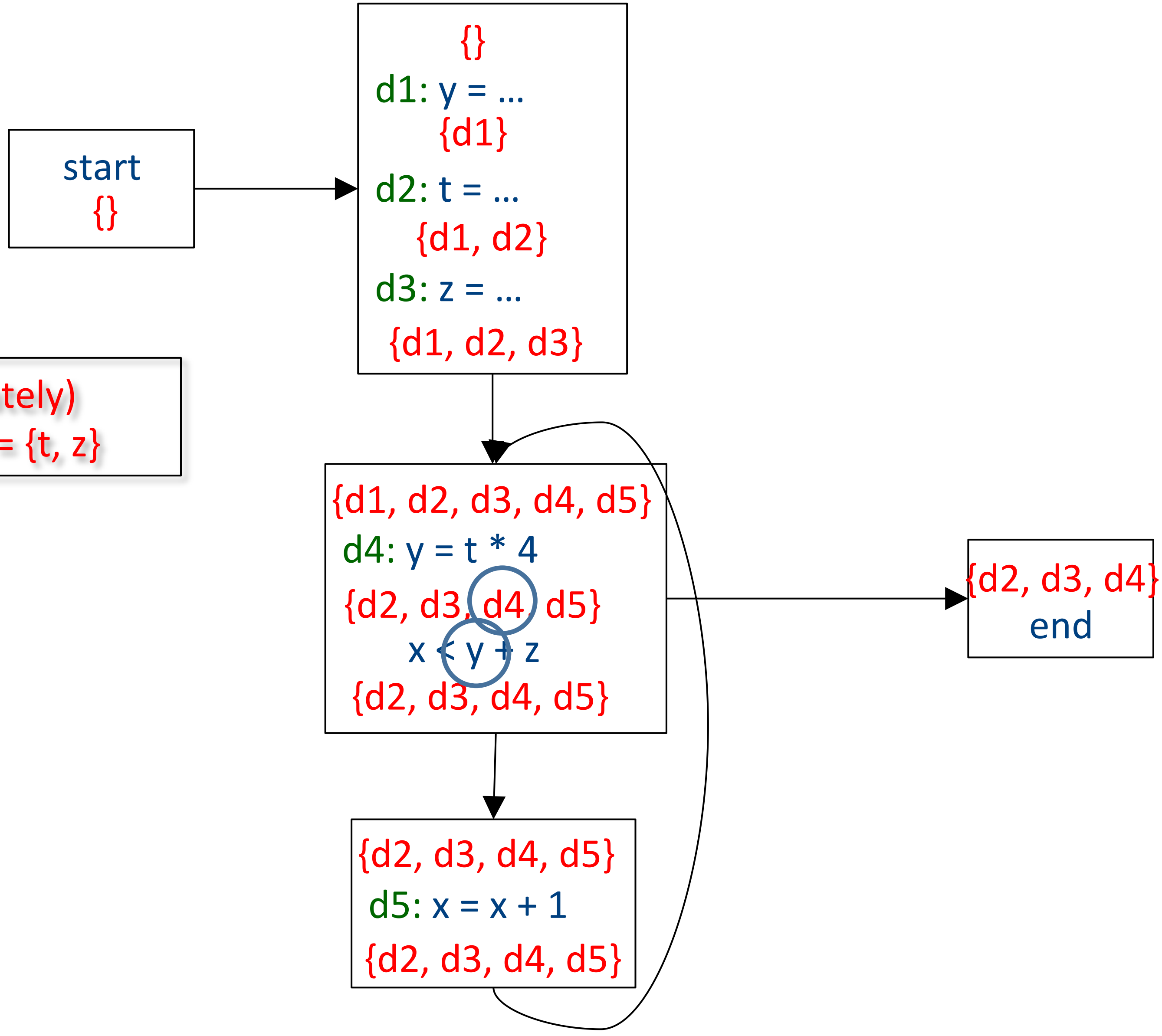


(immediately)
LOOP-INV = {t}

Computing LOOP-INV

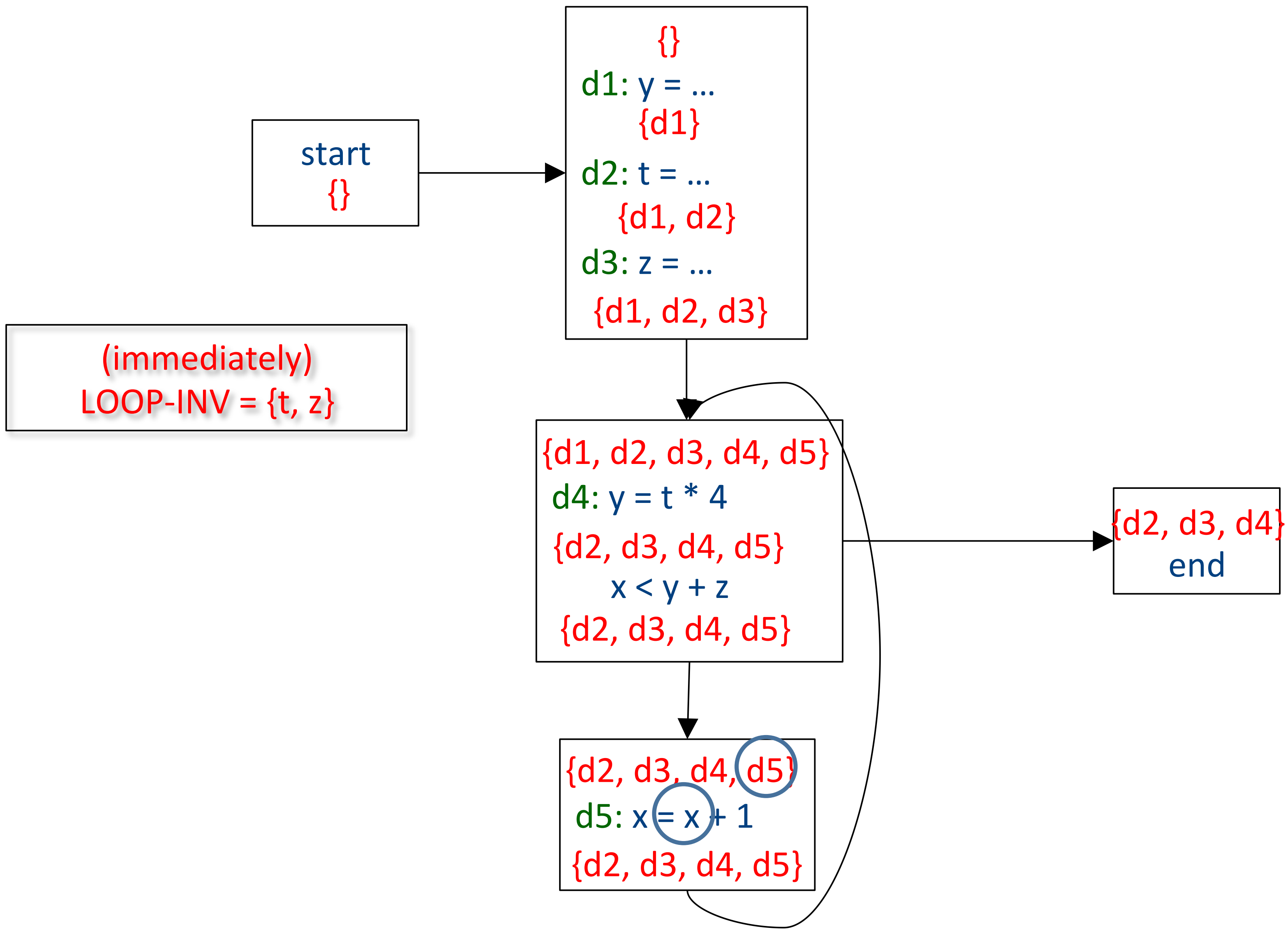


Computing LOOP-INV



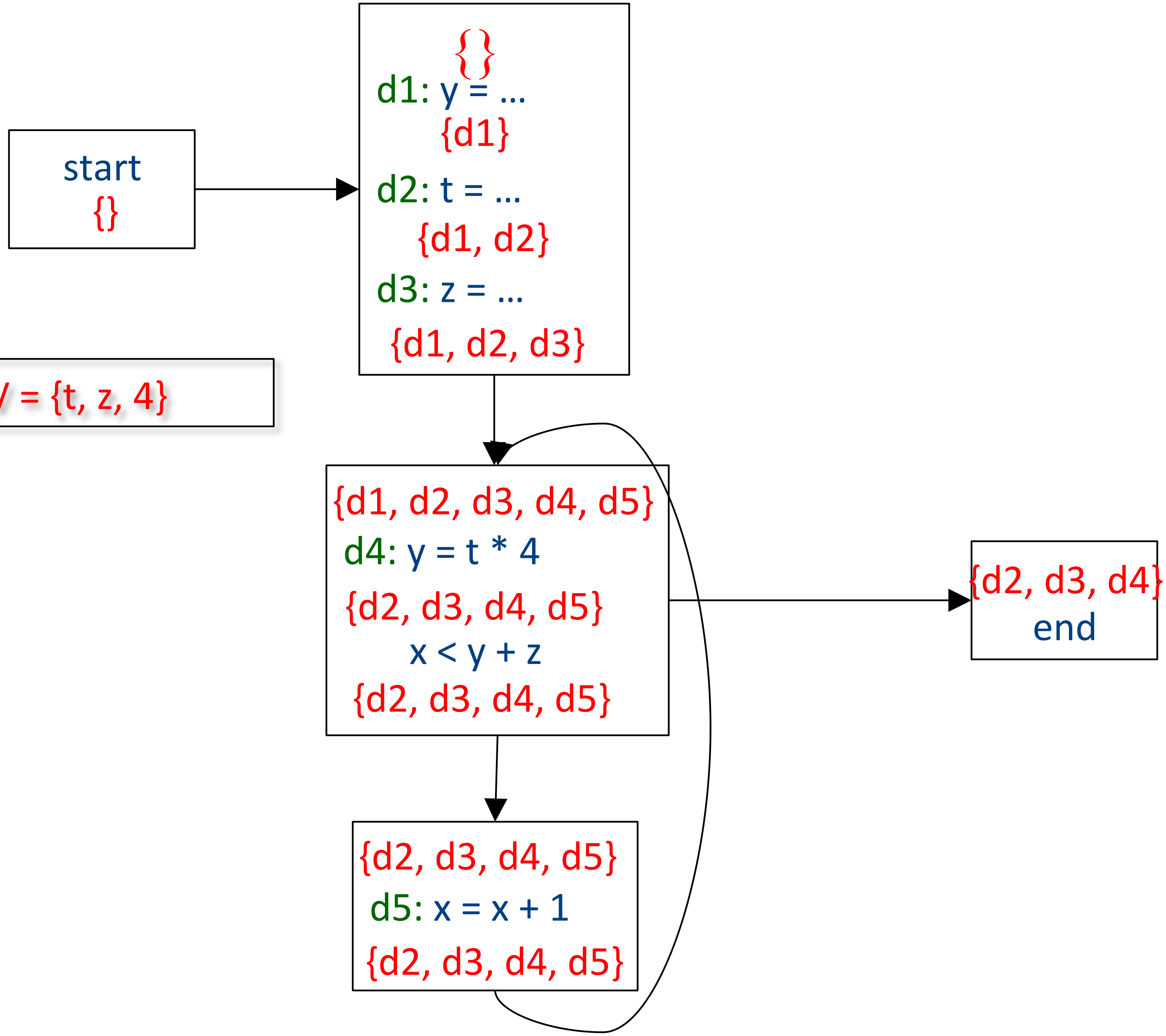
(immediately)
LOOP-INV = {t, z}

Computing LOOP-INV



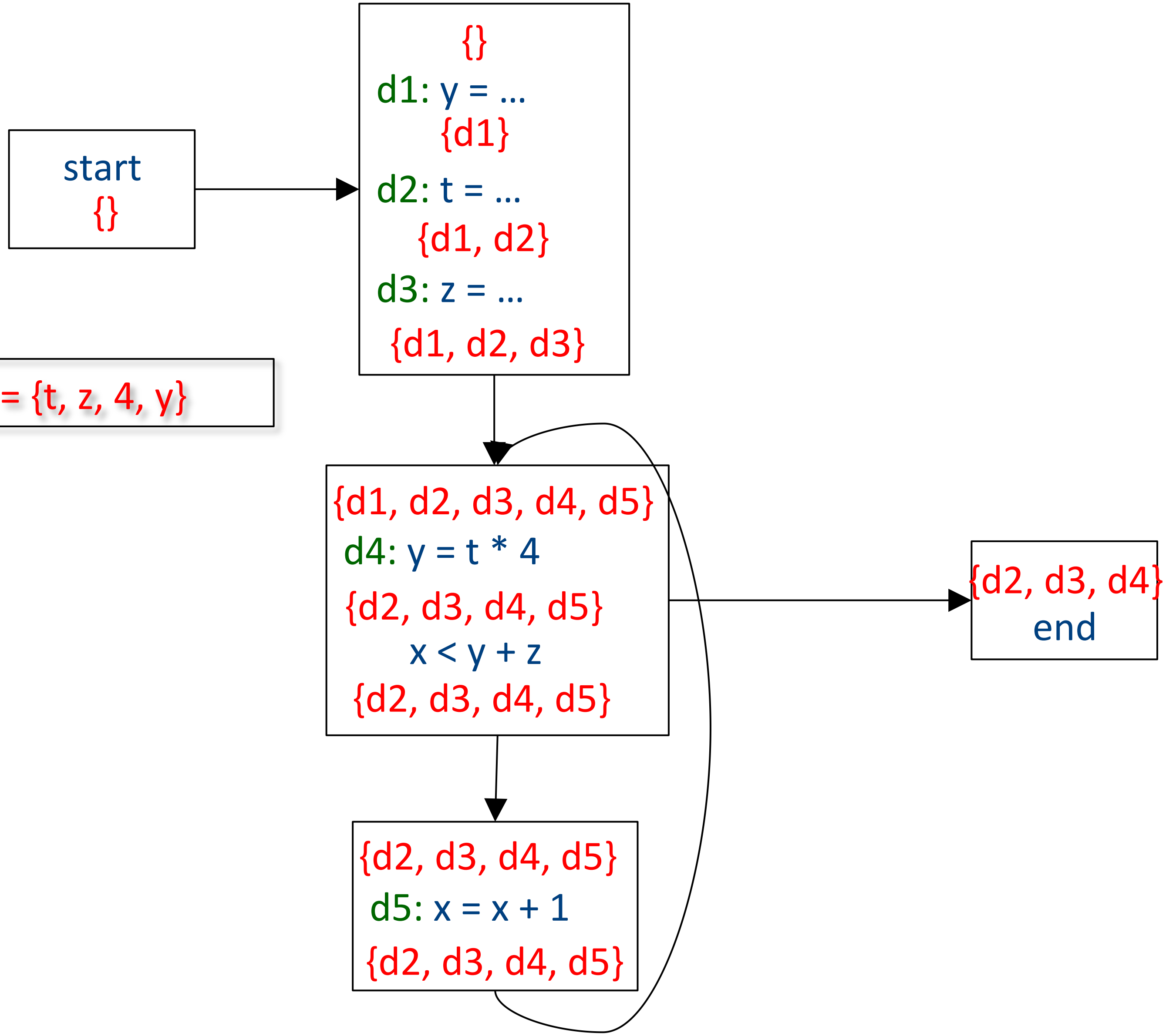
Computing LOOP-INV

LOOP-INV = {t, z, 4}



Computing LOOP-INV

LOOP-INV = {t, z, 4, y}



Induction variables

j is a linear function of the induction variable with multiplier 4

```
while (i < x) {
    j = a + 4 * i
    a[j] = j
    i = i + 1
}
```

i is incremented by a loop-invariant expression on each iteration – this is called an induction variable

Strength-reduction

Prepare initial value

```
j = a + 4 * i
while (i < x) {
    j = j + 4
    a[j] = j
    i = i + 1
}
```

Increment by multiplier

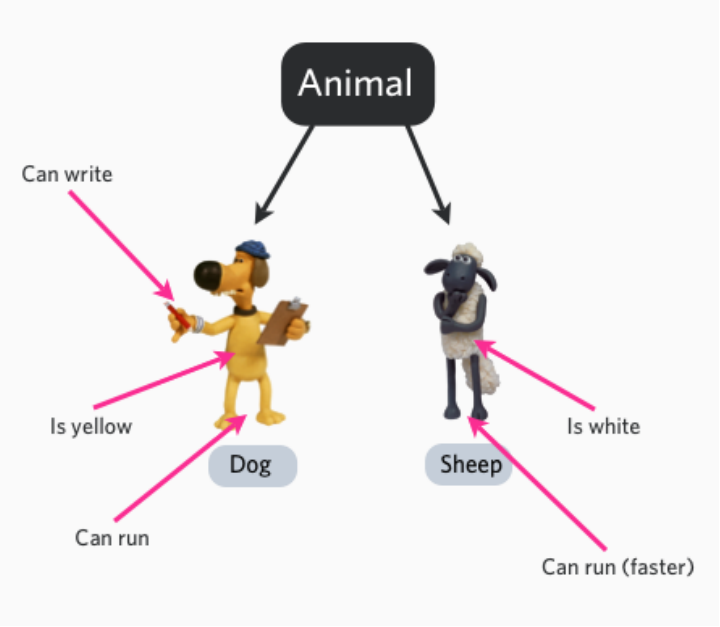
Summary of optimizations

Analysis	**Enabled Optimizations**
Available Expressions	Common-subexpression elimination Copy Propagation
Constant Propagation	Constant folding
Live Variables	Dead code elimination
Reaching Definitions	Loop-invariant code motion

Compilation

0368-3133 2014/15a

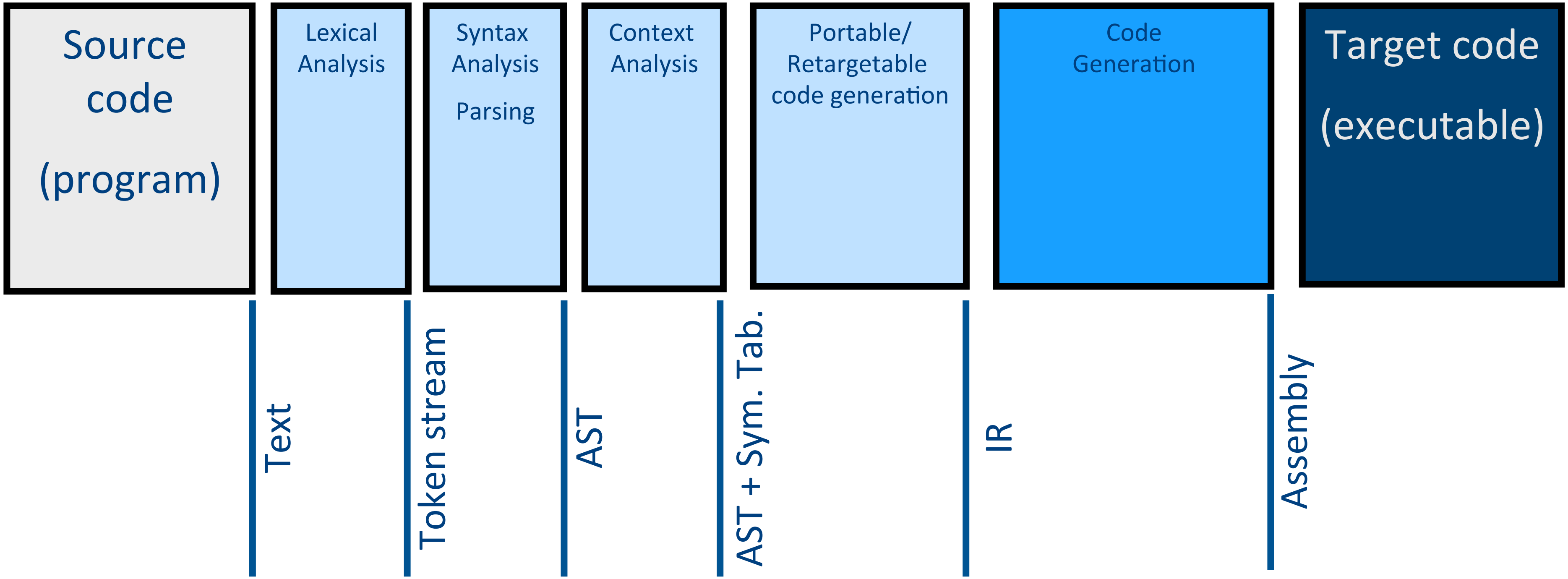
Lecture 12



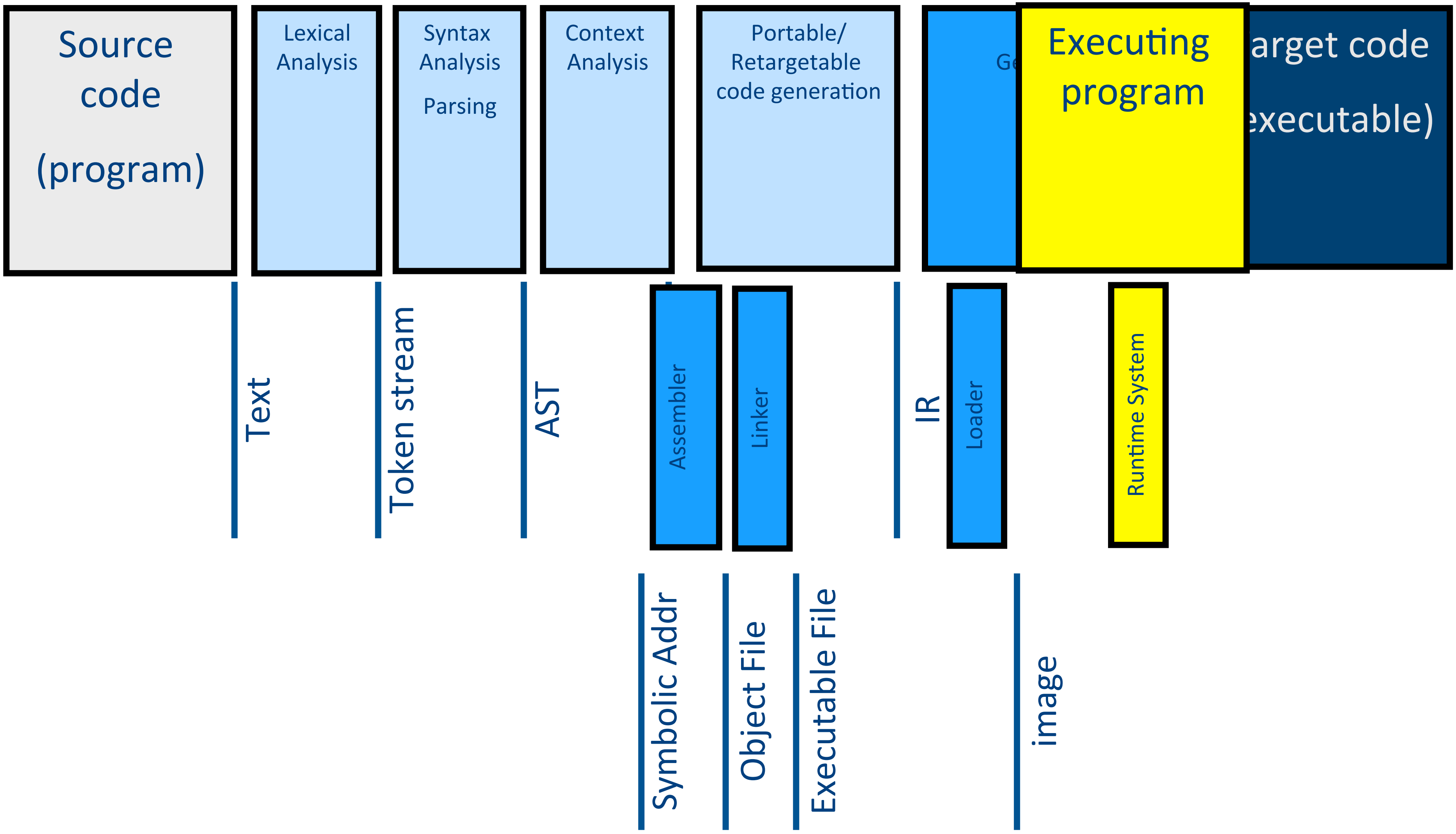
Compiling Object-Oriented Programs

Noam Rinetzky

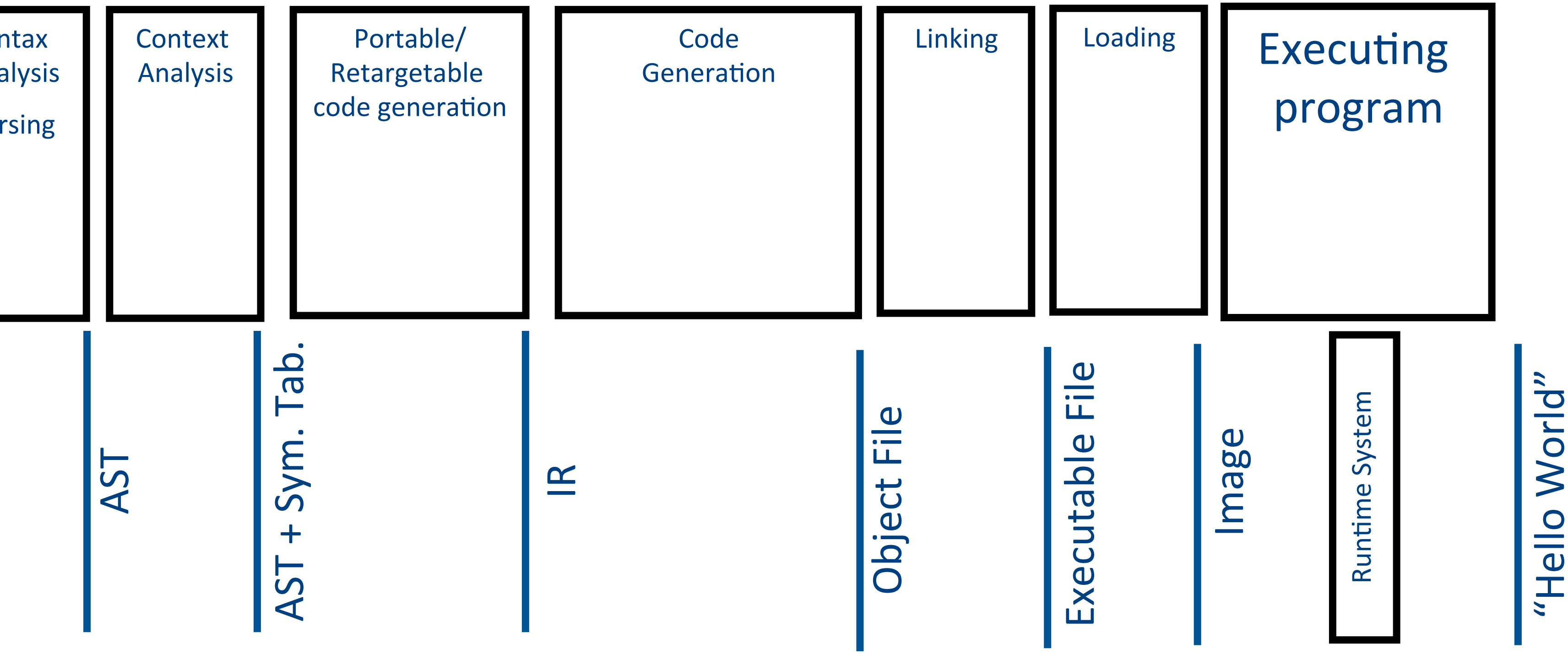
Stages of compilation



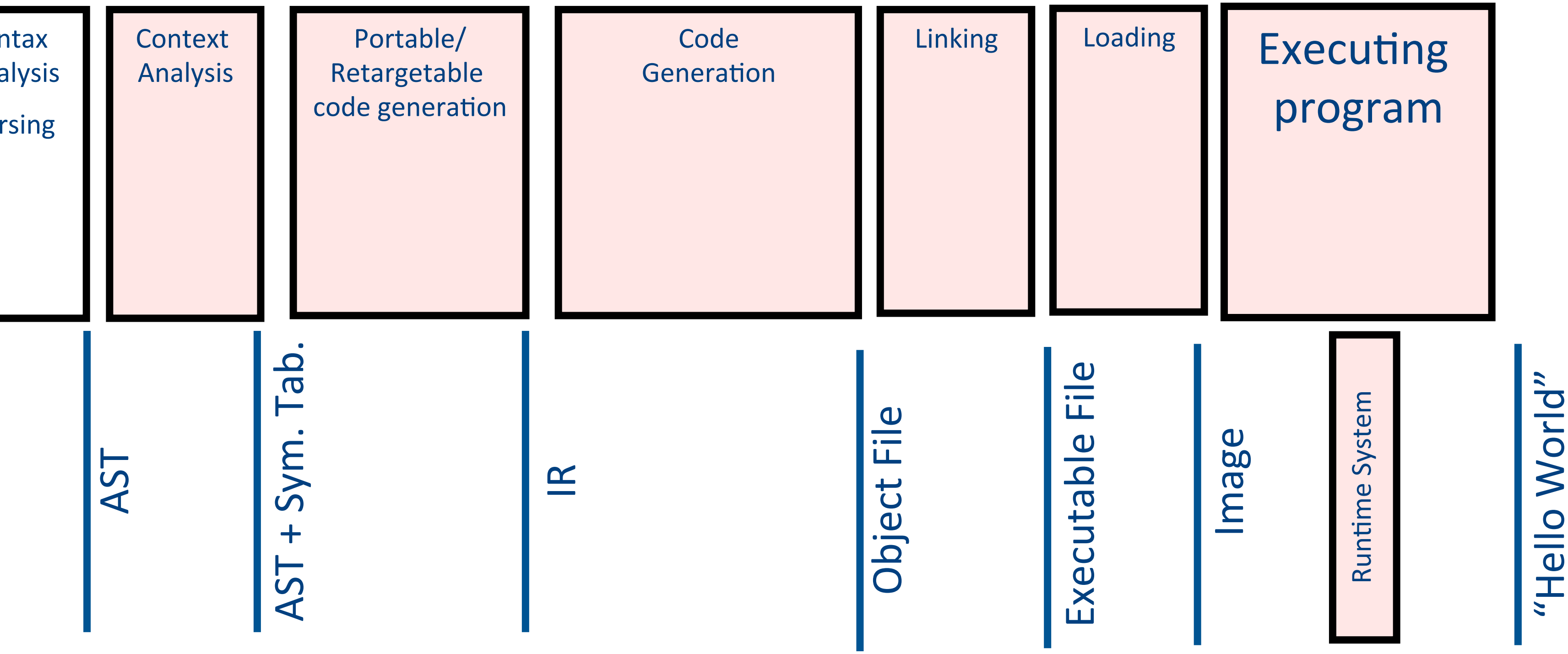
Compilation ➔ Execution



Compilation ➔ Execution



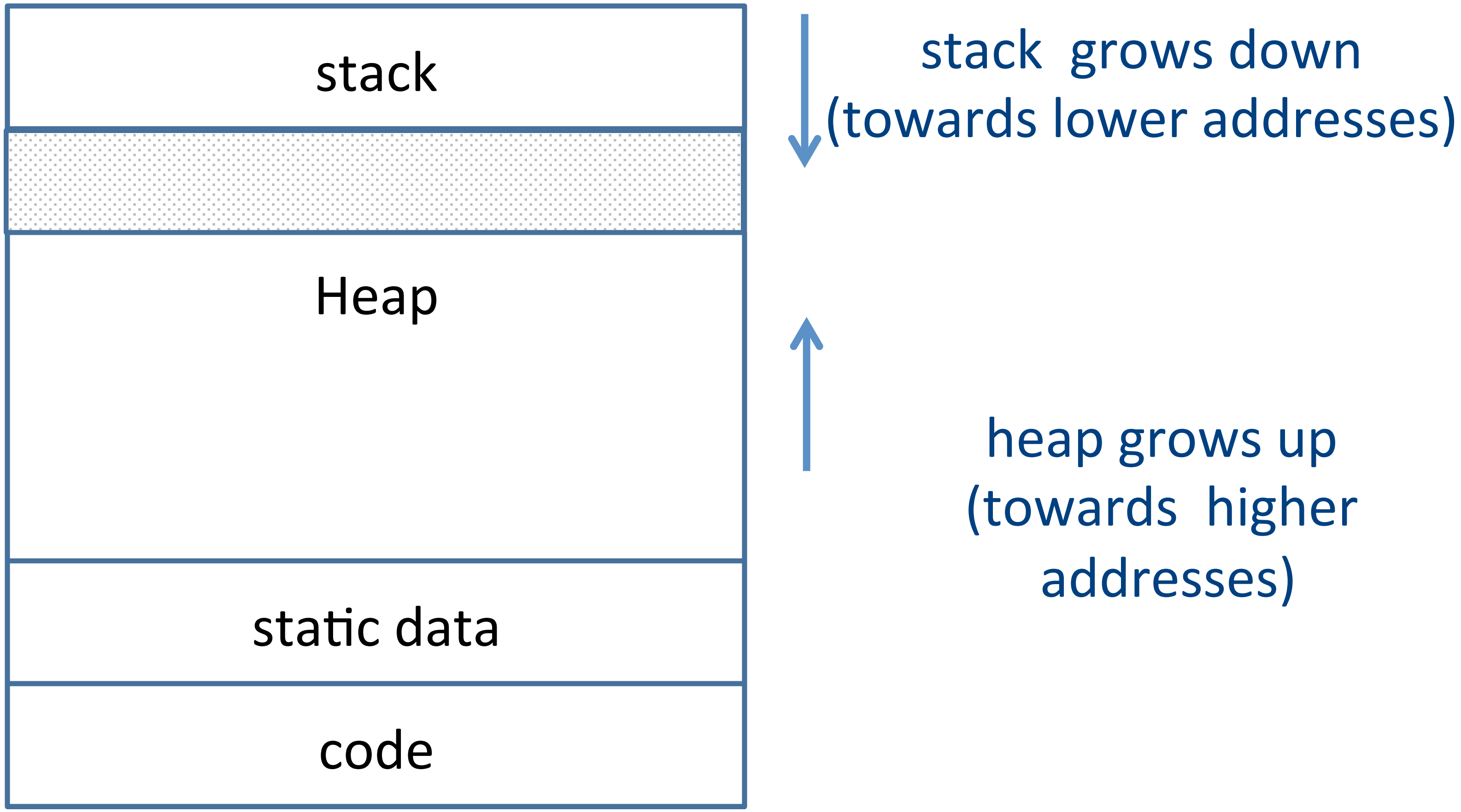
OO: Compilation ➔ Execution



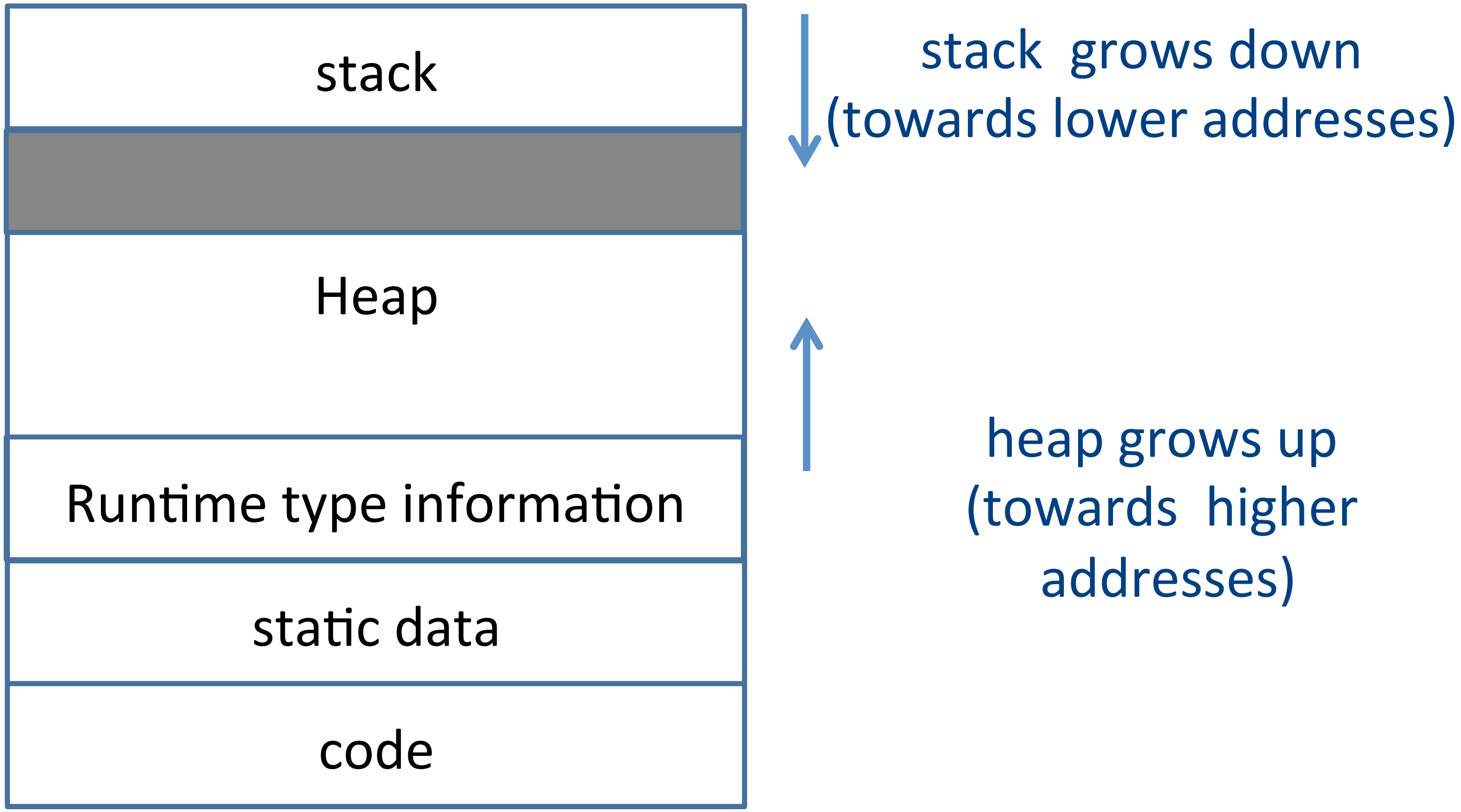
Runtime Environment

- Mediates between the OS and the programming language
- Hides details of the machine from the programmer
  - Ranges from simple support functions all the way to a full-fledged virtual machine
- Handles common tasks
  - Runtime stack (activation records)
  - Memory management
- Runtime type information
  - Method invocation
  - Type conversions

Memory Layout



Memory Layout



Object Oriented Programs

- Simula, Smalltalk, Modula 3, C++, Java, C#, Python

- Objects (usually of type called class)
  - Code
  - Data

- Naturally supports Abstract Data Type implementations
- Information hiding
- Evolution & reusability

A Simple Example

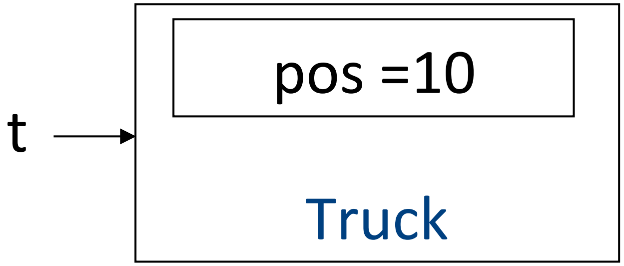
```
class Vehicle extends object {
  int pos = 10;
  void move(int x) {
    position = position + x ;
  }
}

class Truck extends Vehicle {
  void move(int x){
    if (x < 55)
      pos = pos + x;
  }
}

class Car extends Vehicle {
  int passengers = 0;
  void await(vehicle v){
    if (v.pos < pos)
      v.move(pos - v.pos);
    else
      this.move(10);
  }
}
```

```
class main extends object {
  void main() {
    Truck t = new Truck();
    Car c = new Car();
    Vehicle v = c;
    c.move(60);
    v.move(70);
    c.await(t);
  }
}
```

A Simple Example

```
class Vehicle extends object {
  int pos = 10;
  void move(int x) {
    position = position + x ;
  }
}

class Truck extends Vehicle {
  void move(int x){
    if (x < 55)
      pos = pos + x;
  }
}

class Car extends Vehicle {
  int passengers = 0;
  void await(vehicle v){
    if (v.pos < pos)
      v.move(pos - v.pos);
    else
      this.move(10);
  }
}
```

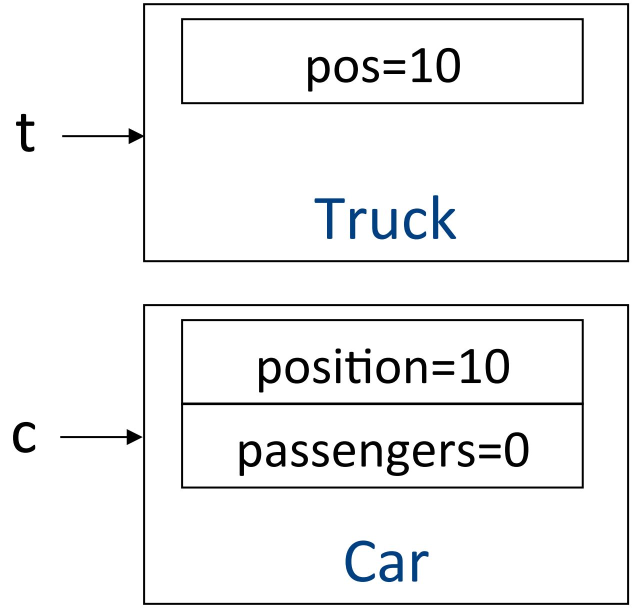
```
class main extends object {
  void main() {
    Truck t = new Truck();
    Car c = new Car();
    Vehicle v = c;
    c.move(60);
    v.move(70);
    c.await(t);
  }
}
```



A Simple Example

```
class Vehicle extends object {
  int pos = 10;
  void move(int x) {
     pos = pos + x ;
  }
}

class Truck extends Vehicle {
  void move(int x){
    if (x < 55)
      pos = pos + x;
  }
}

class Car extends Vehicle {
  int passengers = 0;
  void await(vehicle v){
    if (v.pos < pos)
      v.move(pos - v.pos);
    else
      this.move(10);
  }
}
```

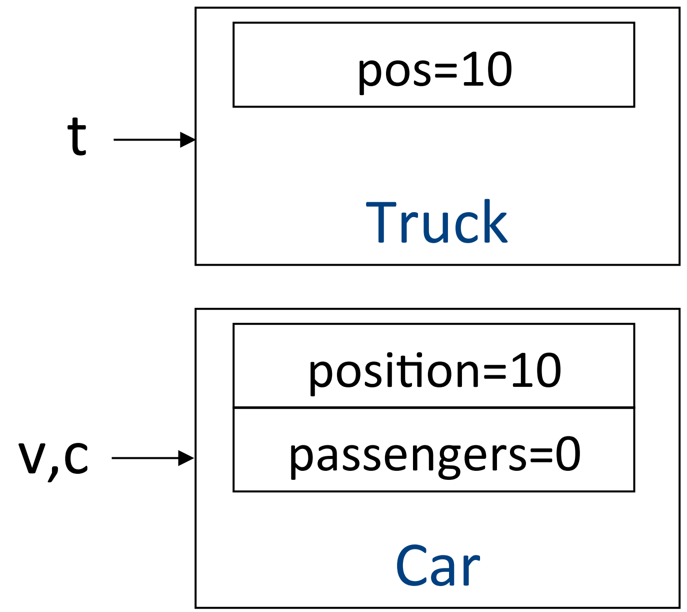
```
class main extends object {
  void main() {
    Truck t = new Truck();
    Car c = new Car();
    Vehicle v = c;
    c.move(60);
    v.move(70);
    c.await(t);
  }
}
```



A Simple Example

```
class Vehicle extends object {
  int pos = 10;
  void move(int x) {
    pos = pos + x ;
  }
}

class Truck extends Vehicle {
  void move(int x){
    if (x < 55)
      pos = pos + x;
  }
}

class Car extends Vehicle {
  int passengers = 0;
  void await(vehicle v){
    if (v.pos < pos)
      v.move(pos - v.pos);
    else
      this.move(10);
  }
}
```

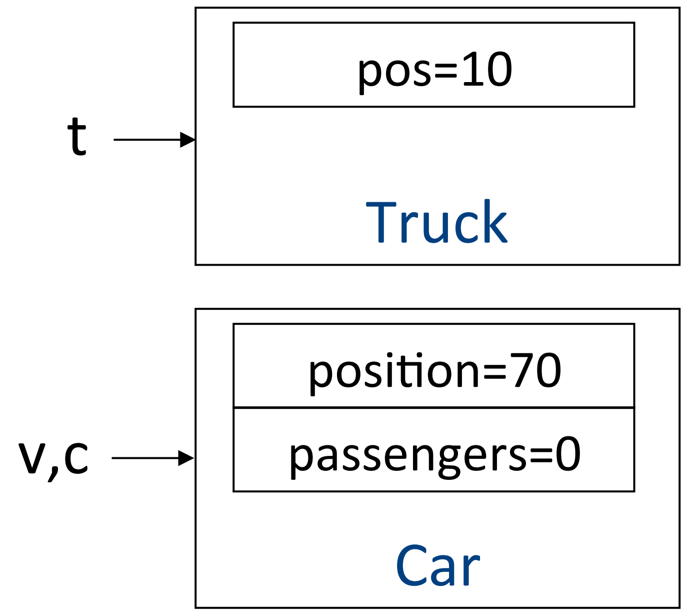
```
class main extends object {
  void main() {
    Truck t = new Truck();
    Car c = new Car();
    Vehicle v = c;
    c.move(60);
    v.move(70);
    c.await(t);
  }
}
```



A Simple Example

```
class Vehicle extends object {
  int pos = 10;
  void move(int x) {
     pos = pos + x;
  }
}

class Truck extends Vehicle {
  void move(int x){
    if (x < 55)
      pos = pos + x;
  }
}

class Car extends Vehicle {
  int passengers = 0;
  void await(vehicle v){
    if (v.pos < pos)
      v.move(pos - v.pos);
    else
      this.move(10);
  }
}
```

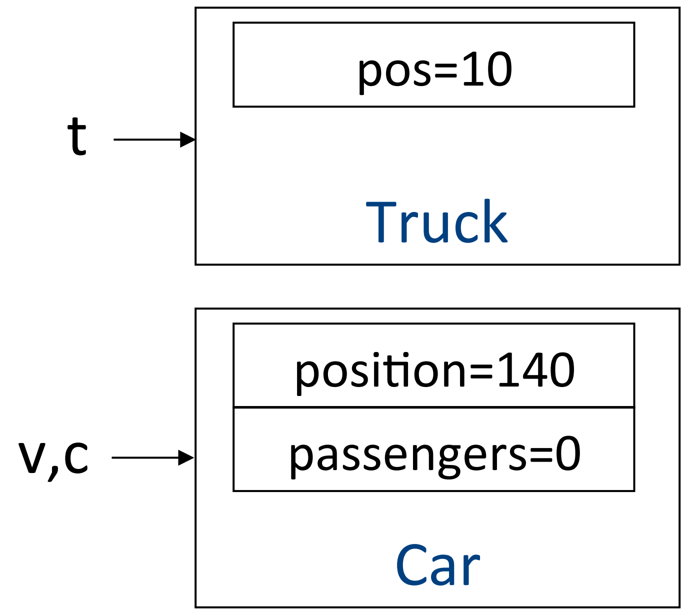
```
class main extends object {
  void main() {
    Truck t = new Truck();
    Car c = new Car();
    Vehicle v = c;
    c.move(60);
    v.move(70);
    c.await(t);
  }
}
```



A Simple Example

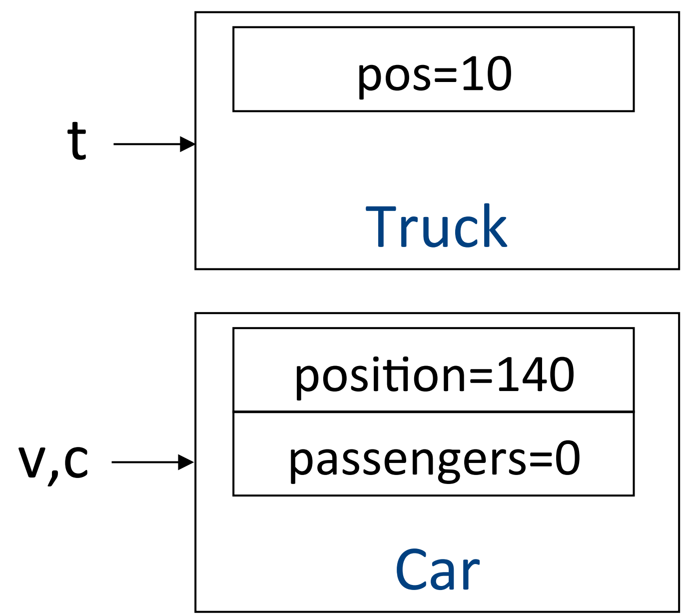
```
class Vehicle extends object {
  int pos = 10;
  void move(int x) {
    position = position + x ;
  }
}

class Truck extends Vehicle {
  void move(int x){
    if (x < 55)
      pos = pos + x;
  }
}

class Car extends Vehicle {
  int passengers = 0;
  void await(vehicle v){
    if (v.pos < pos)
      v.move(pos - v.pos);
    else
      this.move(10);
  }
}
```

```
class main extends object {
  void main() {
    Truck t = new Truck();
    Car c = new Car();
    Vehicle v = c;
    c.move(60);
    v.move(70);
    c.await(t);
  }
}
```



A Simple Example

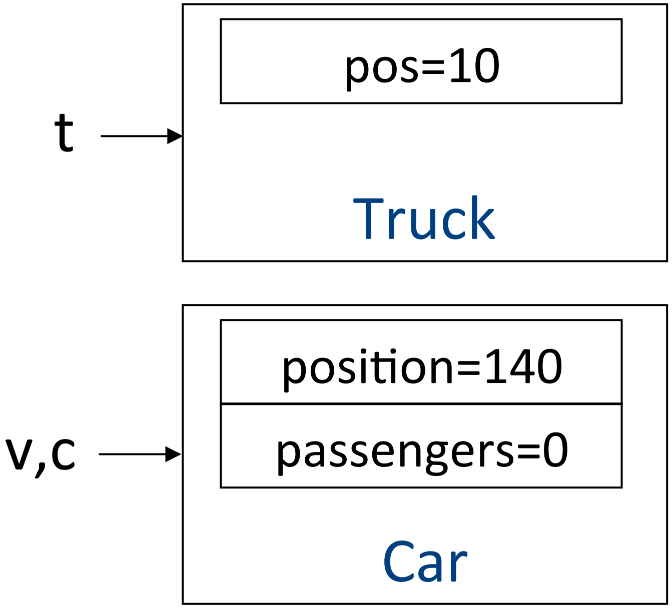
```
class Vehicle extends object {
  int pos = 10;
  void move(int x) {
    position = position + x ;
  }
}

class Truck extends Vehicle {
  void move(int x){
    if (x < 55)
      pos = pos + x;
  }
}

class Car extends Vehicle {
  int passengers = 0;
  void await(vehicle v){
    if (v.pos < pos)
      v.move(pos - v.pos);
    else
      this.move(10);
  }
}

class main extends object {
  void main() {
    Truck t = new Truck();
    Car c = new Car();
    Vehicle v = c;
    c.move(60);
    v.move(70);
    c.await(t);
  }
}
```



A Simple Example

```
class Vehicle extends object {
  int pos = 10;
  void move(int x) {
    position = position + x ;
  }
}

class Truck extends Vehicle {
  void move(int x){
    if (x < 55)
      pos = pos + x;
  }
}

class Car extends Vehicle {
  int passengers = 0;
  void await(vehicle v){
    if (v.pos < pos)
      v.move(pos - v.pos);
    else
      this.move(10);
  }
}
```

```
class main extends object {
  void main() {
    Truck t = new Truck();
    Car c = new Car();
    Vehicle v = c;
    c.move(60);
    v.move(70);
    c.await(t);
  }
}
```



Translation into C (Vehicle)

```
class Vehicle extends object {
  int pos = 10;
  void move(int x) {
    pos = pos + x ;
  }
}
```

```
struct Vehicle {
  int pos;
}
```

Translation into C (Vehicle)

```
class Vehicle extends object {
  int pos = 10;
  void move(int x) {
    pos = pos + x ;
  }
}
```

```
typedef struct Vehicle {
  int pos;
} Ve;
```

Translation into C (Vehicle)

```
class Vehicle extends object {
  int pos = 10;
  void move(int x) {
    pos = pos + x ;
  }
}
```

```
typedef struct Vehicle {
  int pos;
} Ve;

void NewVe(Ve *this){
  this→pos = 10;
}

void moveVe(Ve *this, int x){
  this→pos = this→pos + x;
}
```

Translation into C (Truck)

```
class Truck extends Vehicle {
  void move(int x){
    if (x < 55)
      pos = pos + x;
  }
}
```

```
typedef struct Truck {
  int pos;
} Tr;

void NewTr(Tr *this){
  this→pos = 10;
}

void moveTr(Ve *this, int x){
  if (x<55)
    this→pos = this→pos + x;
}
```

Naïve Translation into C (Car)

```
class Car extends Vehicle {
 int passengers = 0;
 void await(vehicle v){
  if (v.pos < pos)
    v.move(pos - v.pos);
  else
    this.move(10);
 }
}
```

```
typedef struct Car{
  int pos;
  int passengers;
} Ca;

void NewCa (Ca *this){
  this→pos = 10;
  this→passengers = 0;
}

void awaitCa(Ca *this, Ve *v){
  if (v→pos < this→pos)
    moveVe(this→pos - v→pos)
  else
    MoveCa(this, 10)
}
```

Naïve Translation into C (Main)

```
class main extends object {
  void main() {
    Truck t = new Truck();
    Car c = new Car();
    Vehicle v = c;
    c.move(60);
    v.move(70);
    c.await(t);
  }
}
```

```
void mainMa(){
  Tr *t = malloc(sizeof(Tr));
  Ca *c = malloc(sizeof(Ca));
  Ve *v = (Ve*) c;
  moveVe(Ve*) c, 60);
  moveVe(v, 70);
  awaitCa(c,(Ve*) t);
}
```

Naïve Translation into C (Main)

```
class main extends object {
  void main() {
    Truck t = new Truck();
    Car c = new Car();
    Vehicle v = c;
    c.move(60);
    v.move(70);
    c.await(t);
  }
}
```

```
void mainMa(){
  Tr *t = malloc(sizeof(Tr));
  Ca *c = malloc(sizeof(Ca));
  Ve *v = (Ve*) c;
  moveVe(Ve*) c, 60);
  moveVe(v, 70);
  awaitCa(c,(Ve*) t);
}
```

```
void moveCa() ?
```

Naïve Translation into C (Main)

```
class main extends object {
  void main() {
    Truck t = new Truck();
    Car c = new Car();
    Vehicle v = c;
    c.move(60);
    v.move(70);
    c.await(t);
  }
}
```

```
void mainMa(){
  Tr *t = malloc(sizeof(Tr));
  Ca *c = malloc(sizeof(Ca));
  Ve *v = (Ve*) c;
  moveVe(Ve*) c, 60);
  moveVe(v, 70);
  awaitCa(c,(Ve*) t);
}
```

```
void moveCa() ?
```

```
void moveVe(Ve *this, int x){
  this→pos = this→pos + x;
}
```

Compiling Simple Classes

- Fields are handled as records
- Methods have unique names

```
class A {
   field a1;
   field a2;
   method m1() {…}
   method m2(int i) {…}
}
```

Runtime object

a1
a2

Compile-Time Table

m1A
m2A

```
void m2A(classA  *this, int i) {
   // Body of m2 with any object
   // field f as this→f
   …
}
```

Compiling Simple Classes

- Fields are handled as records
- Methods have unique names

```
class A {
   field a1;
   field a2;
   method m1() {…}
   method m2(int i) {…}
}
```

Runtime object

a1
a2

Compile-Time Table

m1A
m2A

```
void m2_A(classA  *this, int i) {
   // Body of m2 with any object
   // field f as this→f
   …
}
```

```
a.m2(5)
```

```
m2A(a,5)   //m2A(&a,5)
```

Features of OO languages

- Inheritance
- Method overriding
- Polymorphism
- Dynamic binding

Handling Single Inheritance

- Simple type extension
- Type checking module checks consistency
- Use prefixing to assign fields in a consistent way

```
class A {
    field a1;
    field a2;
    method m1() {…}
    method m2() {…}
}
```

```
class B extends A {
    field b1;
    method m3() {…}
}
```

Method Overriding

- Redefines functionality
  - More specific
  - Can access additional fields

```
class A {
    field a1;
    field a2;
    method m1() {…}
    method m2() {…}
}
```

```
class B extends A {
    field b1;
    method m2() {
        … b1 …
    }
    method m3() {…}
}
```

Method Overriding

- Redefines functionality
  - More specific
  - Can access additional fields

```
class A {
    field a1;
    field a2;
    method m1() {…}
    method m2() {…}
}
```

m2 is declared and defined

```
class B extends A {
    field a3;
    method m2() {
        … a3 …
    }
    method m3() {…}
}
```

m2 is redefined

Method Overriding

- Redefines functionality
- Affects semantic analysis

```
class A {
    field a1;
    field a2;
    method m1() {…}
    method m2() {…}
}
```

```
class B extends A {
    field a3;
    method m2() {
        … a3 …
    }
    method m3() {…}
}
```

Runtime object

a1
a2

Compile-Time Table

m1A_A
m2A_A

Runtime object

a1
a2
b1

Compile-Time Table

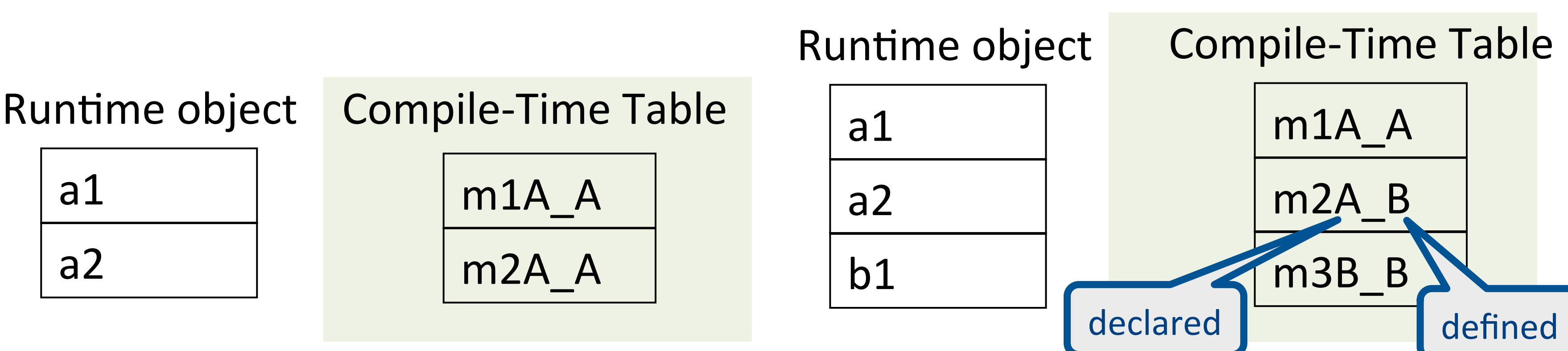
m1A_A
m2A_B
m3B_B

Method Overriding

- Redefines functionality
- Affects semantic analysis

```
class A {
    field a1;
    field a2;
    method m1() {…}
    method m2() {…}
}
```

```
class B extends A {
    field b1;
    method m2() {
        … b1 …
    }
    method m3() {…}
}
```



Method Overriding

```
a.m2(5)  // class(a) = A
```

```
m2A_A(a, 5)
```

```
class A {
    field a1;
    field a2;
    method m1() {…}
    method m2() {…}
}
```

Runtime object

a1
a2

Compile-Time Table

m1A_A
m2A_A

```
b.m2(5)  // class(b) = B
```

```
m2A_B(b, 5)
```

```
class B extends A {
    field b1;
    method m2() {
        … b1 …
    }
    method m3() {…}
}
```

Runtime object

a1
a2
b1

Compile-Time Table

m1A_A
m2A_B
m3B_B

Method Overriding

```
class A {
    field a1;
    field a2;
    method m1() {…}
    method m2() {…}
}
```

```
class B extends A {
    field b1;
    method m2() {
     … b1 …
    }
    method m3() {…}
}
```

```
typedef struct  {
    field a1;
    field a2;
} A;

void m1A_A(A* this){…}
void m2A_A(A* this){…}
```

```
typedef struct  {
    field a1;
    field a2;
    field b1;
} B;

void  m2A_B(B* this) {…}
void  m3B_B(B* this) {…}
```

Runtime object

a1
a2

Compile-Time Table

m1A_A
m2A_A

Runtime object

a1
a2
b1

Compile-Time Table

m1A_A
m2A_B
m3B_B

Method Overriding

a.m2(5) // class(a) = A

m2A_A(a, 5)

b.m2(5) // class(b) = B

m2A_B(b, 5)

```
typedef struct  {
    field a1;
    field a2;
} A;

void m1A_A(A* this){…}
void m2A_A(A* this){…}
```

```
typedef struct  {
    field a1;
    field a2;
    field b1;
} B;

void m2A_B(B* this) {…}
void m3B_B(B* this) {…}
```

Runtime object

a1
a2

Compile-Time Table

m1A_A
m2A_A

Runtime object

a1
a2
b1

Compile-Time Table

m1A_A
m2A_B
m3B_B

Abstract Methods

- Declared separately
  - Defined in child classes
  - E.g., Java abstract classes
    - Abstract classes cannot be instantiated

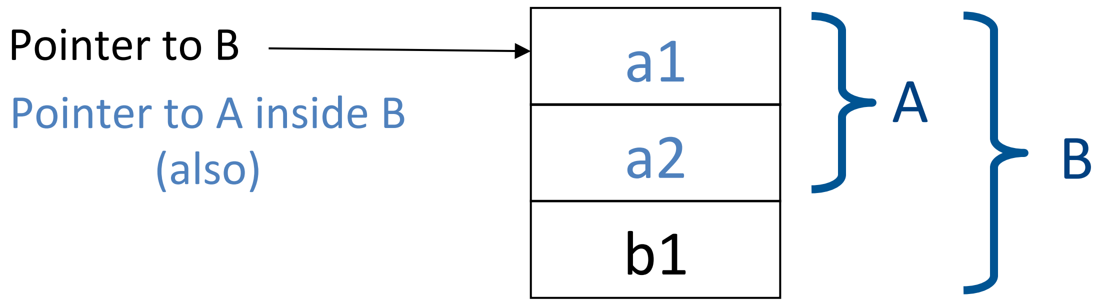
- Handled similarly
- Textbook uses “virtual” for abstract

Handling Polymorphism

- When a class B extends a class A
  - variable of type pointer to A may actually refer to object of type B
- Upcasting from a subclass to a superclass
- Prefixing guarantees validity

```
class B *b = …;

class A *a = b ;
```

```
classA *a = convert_ptr_to_B_to_ptr_A(b) ;
```



Dynamic Binding

- An object ("pointer") o declared to be of class A can actually be ("refer") to a class B
- What does 'o.m()' mean?
  - Static binding
  - Dynamic binding
- Depends on the programming language rules
- How to implement dynamic binding?
  - The invoked function is not known at compile time
  - Need to operate on data of the B and A in consistent way

Conceptual Impl. of Dynamic Binding

```
class A {
    field a1;
    field a2;
    method m1() {…}
    method m2() {…}
}
```

```
class B extends A {
    field b1;
    method m2() {
     … a3 …
    }
    method m3() {…}
}
```

```
typedef struct  {
    field a1;
    field a2;
} A;

void m1A_A(A* this){…}
void m2A_A(A* this){…}
```

```
typedef struct  {
    field a1;
    field a2;
    field b1;
} B;

void m2A_B(B* this) {…}
void m3B_B(B* this) {…}
```

Runtime object

a1
a2

Compile-Time Table

m1A_A
m2A_A

Runtime object

a1
a2
b1

Compile-Time Table

m1A_A
m2A_B
m3B_B

Conceptual Impl. of Dynamic Binding

```
switch(dynamic_type(p)) {
  case Dynamic_class_A: m2_A_A(p, 3);
  case Dynamic_class_B:m2_A_B(convert_ptr_to_A_to_ptr_B(p), 3);
}
```

```
typedef struct  {
    field a1;
    field a2;
} A;

void m1A_A(A* this){…}
void m2A_A(A* this){…}
```

```
typedef struct  {
    field a1;
    field a2;
    field b1;
} B;

void m2A_B(B* this) {…}
void m3B_B(B* this) {…}
```

Runtime object

a1
a2

Compile-Time Table

m1A_A
m2A_A

Runtime object

a1
a2
b1

Compile-Time Table

m1A_A
m2A_B
m3B_B

Conceptual Impl. of Dynamic Binding

?

```
switch(dynamic_type(p)) {
  case Dynamic_class_A: m2_A_A(p, 3);
  case Dynamic_class_B:m2_A_B(convert_ptr_to_A_to_ptr_B(p), 3);
}
```

```
typedef struct  {
    field a1;
    field a2;
} A;

void m1A_A(A* this){…}
void m2A_A(A* this){…}
```

```
typedef struct  {
    field a1;
    field a2;
    field b1;
} B;

void m2A_B(B* this) {…}
void m3B_B(B* this) {…}
```

Runtime object

a1
a2

Compile-Time Table

m1A_A
m2A_A

Runtime object

a1
a2
b1

Compile-Time Table

m1A_A
m2A_B
m3B_B

More efficient implementation

- Apply pointer conversion in sublasses
  - Use dispatch table to invoke functions
  - Similar to table implementation of case

```
void m2A_B(classA *this_A) {
    Class_B *this = convert_ptr_to_A_ptr_to_A_B(this_A);
   …
}
```

More efficient implementation

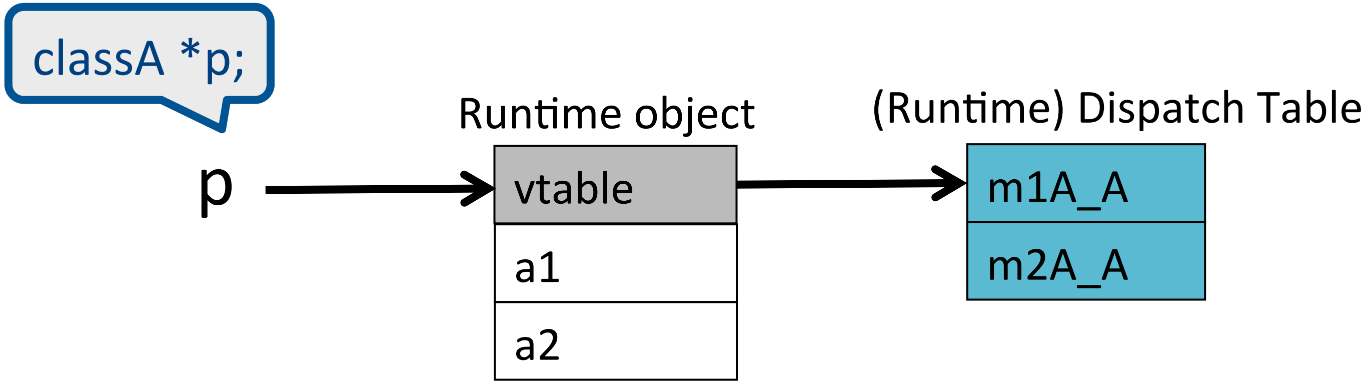
```
typedef struct  {
    field a1;
    field a2;
} A;

void m1A_A(A* this){…}
void m2A_A(A* this, int x){…}
```

```
typedef struct  {
    field a1;
    field a2;
    field b1;
} B;

void m2A_B(A* thisA, int x){
  Class_B *this =
    convert_ptr_to_A_to_ptr_to_B(thisA);
    …
  }

void m3B_B(B* this){…}
```

More efficient implementation

```
typedef struct  {
    field a1;
    field a2;
} A;

void m1A_A(A* this){…}
void m2A_A(A* this, int x){…}
```

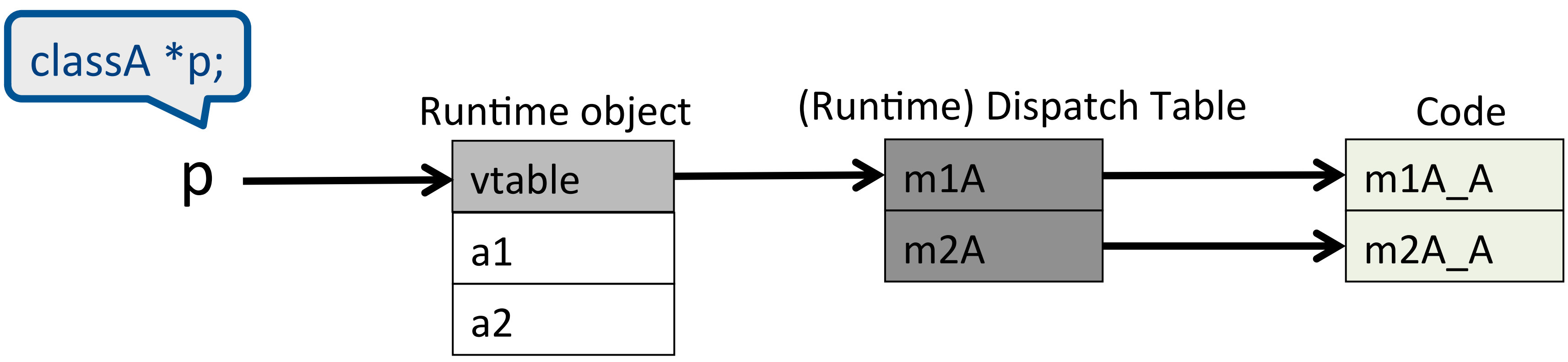
```
typedef struct  {
    field a1;
    field a2;
    field b1;
} B;

void m2A_B(A* thisA, int x){
  Class_B *this =
    convert_ptr_to_A_to_ptr_to_B(thisA);
    …
  }

void m3B_B(B* this){…}
```



More efficient implementation

```
typedef struct  {
    field a1;
    field a2;
} A;

void m1A_A(A* this){…}
void m2A_A(A* this, int x){…}
```

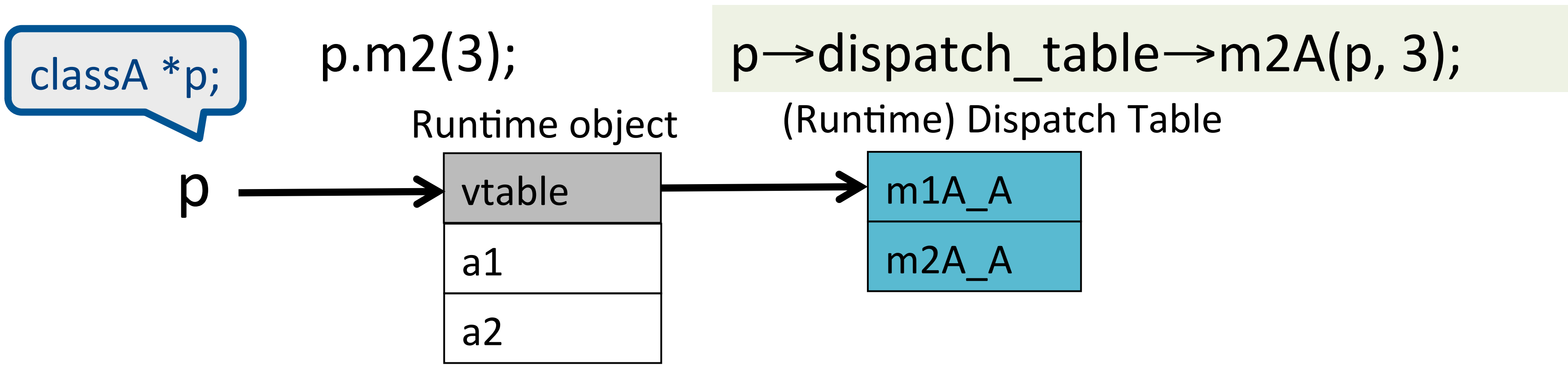
```
typedef struct  {
    field a1;
    field a2;
    field b1;
} B;

void m2A_B(A* thisA, int x){
  Class_B *this =
    convert_ptr_to_A_to_ptr_to_B(thisA);
    …
  }

void m3B_B(B* this){…}
```



More efficient implementation

```
typedef struct  {
    field a1;
    field a2;
} A;

void m1A_A(A* this){…}
void m2A_A(A* this, int x){…}
```

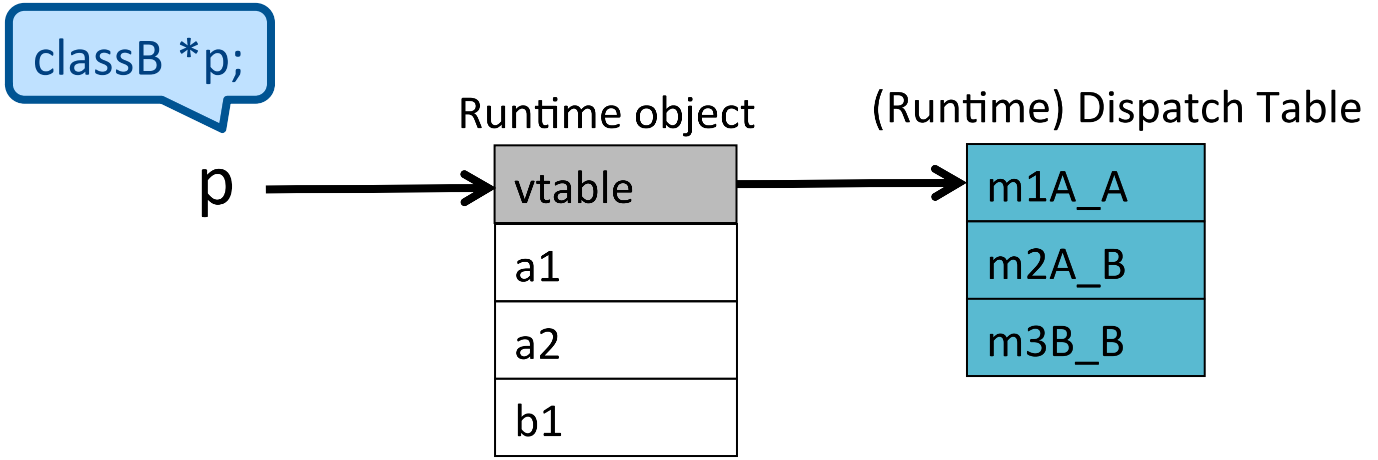
```
typedef struct  {
    field a1;
    field a2;
    field b1;
} B;

void m2A_B(A* thisA, int x){
  Class_B *this =
    convert_ptr_to_A_to_ptr_to_B(thisA);
    …
  }

void m3B_B(B* this){…}
```



More efficient implementation

```
typedef struct  {
    field a1;
    field a2;
} A;

void m1A_A(A* this){…}
void m2A_A(A* this, int x){…}
```

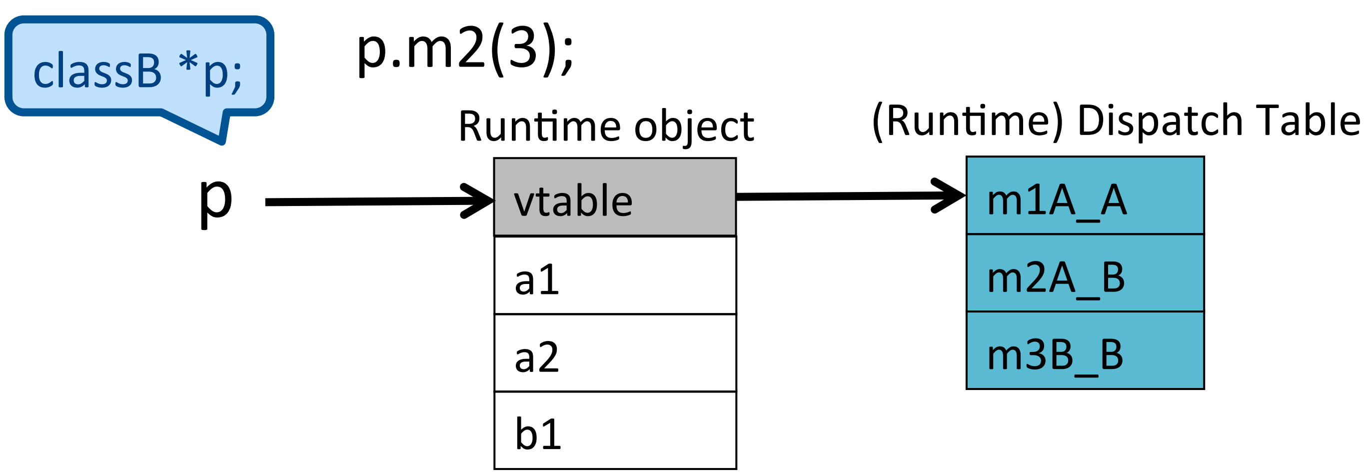
```
typedef struct  {
    field a1;
    field a2;
    field b1;
} B;

void m2A_B(A* thisA, int x){
  Class_B *this =
    convert_ptr_to_A_to_ptr_to_B(thisA);
    …
  }

void m3B_B(B* this){…}
```



More efficient implementation

```
typedef struct  {
    field a1;
    field a2;
} A;

void m1A_A(A* this){…}
void m2A_A(A* this, int x){…}
```

```
typedef struct  {
    field a1;
    field a2;
    field b1;
} B;

void m2A_B(A* thisA, int x){
  Class_B *this =
    convert_ptr_to_A_to_ptr_to_B(thisA);
    …
  }

void m3B_B(B* this){…}
```



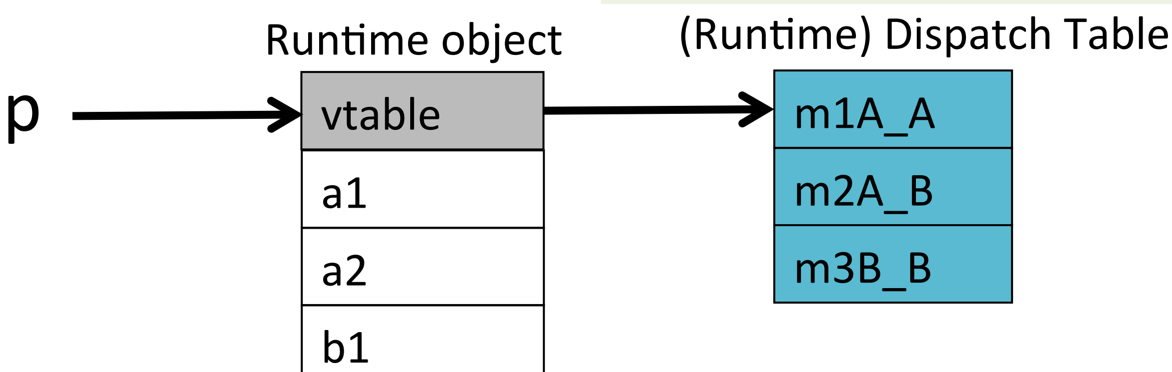
More efficient implementation

```
typedef struct  {
    field a1;
    field a2;
} A;

void m1A_A(A* this){…}
void m2A_A(A* this, int x){…}
```

```
typedef struct  {
    field a1;
    field a2;
    field b1;
} B;

void m2A_B(A* thisA, int x){
  Class_B *this =
    convert_ptr_to_A_to_ptr_to_B(thisA);
    …
  }

void m3B_B(B* this){…}
```

p.m2(3);

p→dispatch_table→m2A(~~p~~, 3);



More efficient implementation

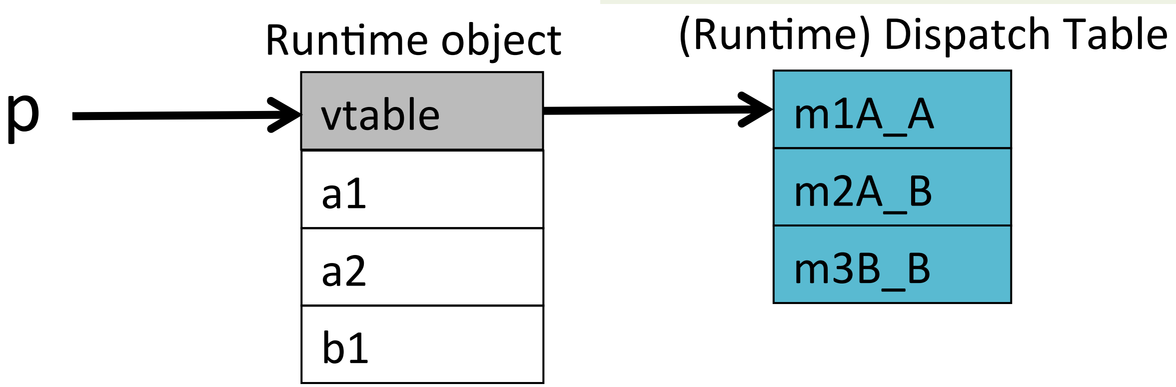
```
typedef struct  {
    field a1;
    field a2;
} A;

void m1A_A(A* this){…}
void m2A_A(A* this, int x){…}
```

```
typedef struct  {
    field a1;
    field a2;
    field b1;
} B;

void m2A_B(A* thisA, int x){
  Class_B *this =
    convert_ptr_to_A_to_ptr_to_B(thisA);
    …
  }

void m3B_B(B* this){…}
```

p.m2(3);

convert_ptr_to_B_to_ptr_to_A(p)

p→dispatch_table→m2A(, 3);



Multiple Inheritance

```
class C {
    field c1;
    field c2;
    method m1(){…}
    method m2(){…}
}
```

```
class D {
    field d1;

    method m3() {…}
    method m4(){…}
}
```

```
class E extends C, D {
    field e1;

    method m2() {…}
    method m4() {…}
    method m5(){…}
}
```

Multiple Inheritance

- Allows unifying behaviors
- But raises semantic difficulties
  - Ambiguity of classes
  - Repeated inheritance
- Hard to implement
  - Semantic analysis
  - Code generation
    - Prefixing no longer work
    - Need to generate code for downcasts
- Hard to use

A simple implementation

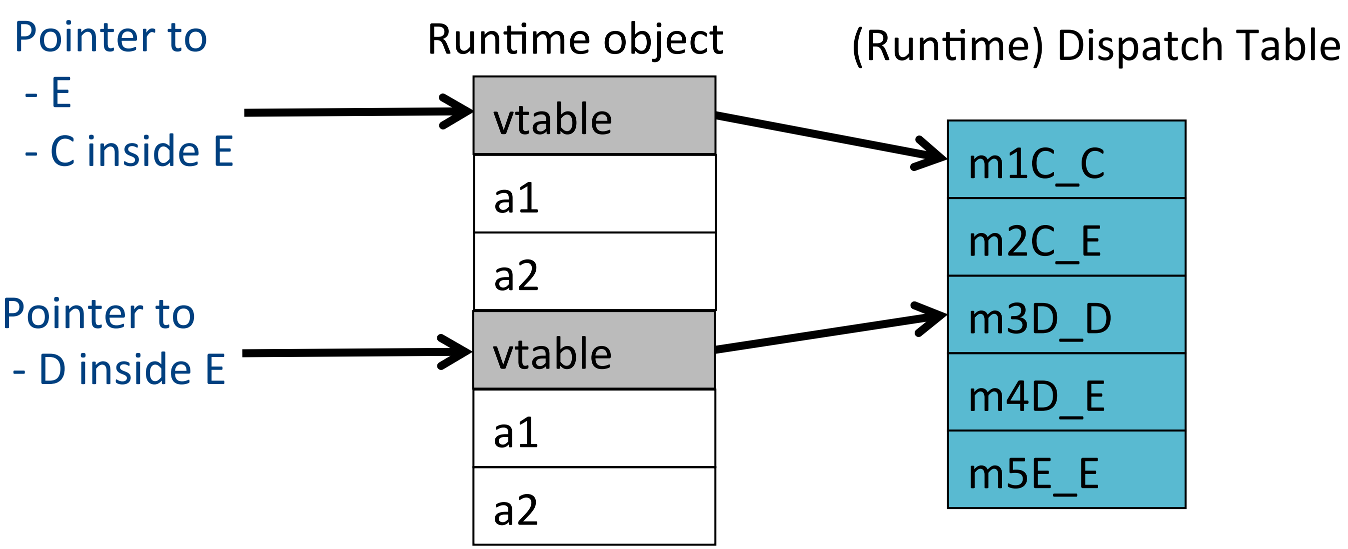
- Merge dispatch tables of superclases
- Generate code for upcasts and downcasts

A simple implementation

```
class C {
    field c1;
    field c2;
    method m1(){…}
    method m2(){…}
}
```

```
class D {
    field d1;

    method m3()  {…}
    method m4(){…}
}
```

```
class E extends C, D {
    field e1;

    method m2()  {…}
    method m4()  {…}
    method m5(){…}
}
```



Downcasting (E→C,D)

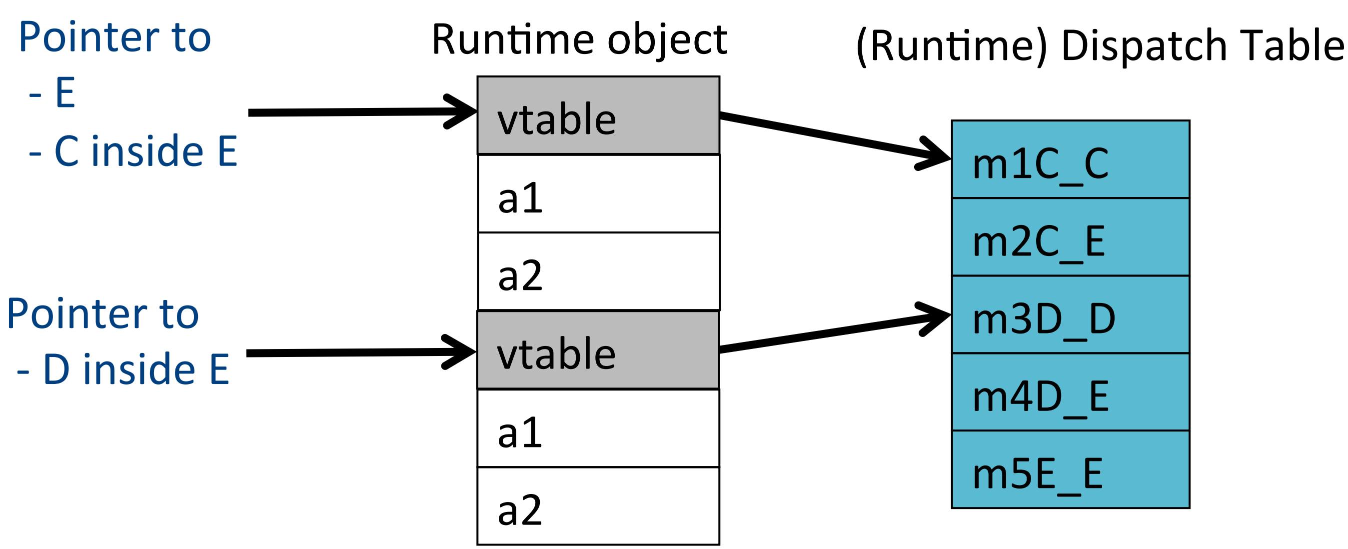
```
class C {
    field c1;
    field c2;
    method m1(){…}
    method m2(){…}
}
```

```
class D {
    field d1;

    method m3()  {…}
    method m4(){…}
}
```

```
class E extends C, D {
    field e1;

    method m2()  {…}
    method m4()  {…}
    method m5(){…}
}
```

convert_ptr_to_E_to_ptr_to_C(e) = e;

convert_ptr_to_E_to_ptr_to_D(e) = e + sizeof(C);



Upcasting (C,D→E)

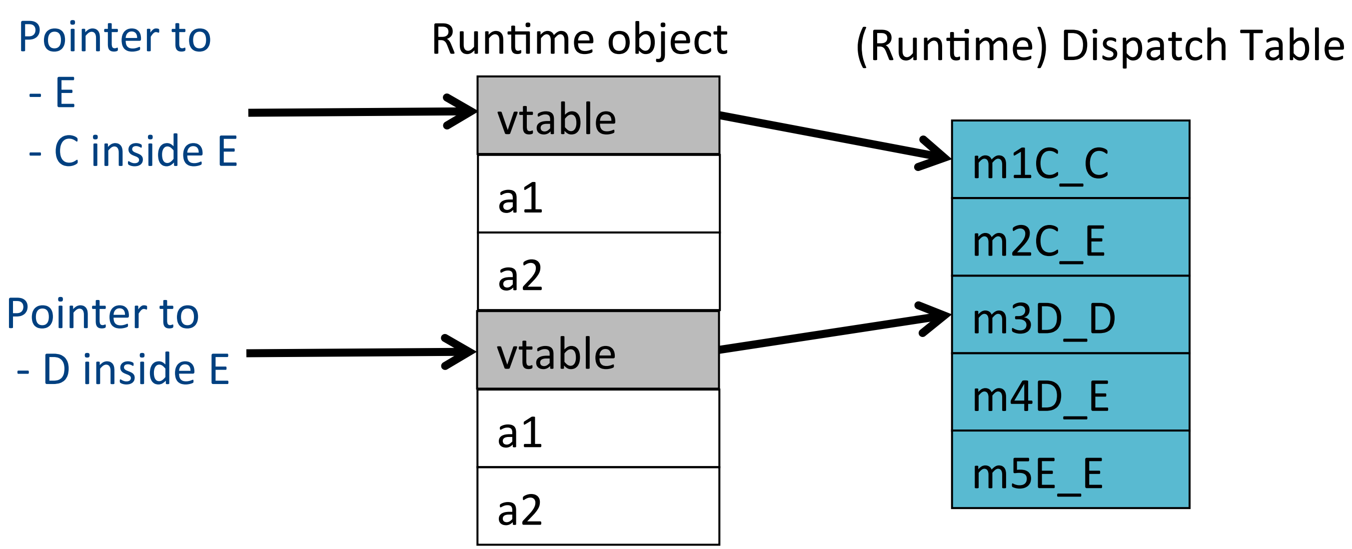
```
class C {
    field c1;
    field c2;
    method m1(){…}
    method m2(){…}
}
```

```
class D {
    field d1;

    method m3() {…}
    method m4(){…}
}
```

```
class E extends C, D {
    field e1;

    method m2() {…}
    method m4() {…}
    method m5(){…}
}
```

convert_ptr_to_C_to_ptr_to_E(c) = c;

convert_ptr_to_D_to_ptr_to_E(d) = d - sizeof(C);



Multiple Inheritance

```
class A{
    field a1;
    field a2;
    method m1(){…}
    method m3(){…}
}
```

```
class C extends A {
    field c1;
    field c2;
    method m1(){…}
    method m2(){…}
}
```

```
class D extends A {
    field d1;

    method m3(){…}
    method m4(){…}
}
```

```
class E extends C, D {
    field e1;

    method m2() {…}
    method m4() {…}
    method m5(){…}
}
```

Multiple Inheritance

```
class A{
    field a1;
    field a2;
    method m1(){…}
    method m3(){…}
}
```

```
class C extends A {
    field c1;
    field c2;
    method m1(){…}
    method m2(){…}
}
```

```
class D extends A {
    field d1;

    method m3(){…}
    method m4(){…}
}
```

```
class E extends C, D {
    field e1;

    method m2() {…}
    method m4() {…}
    method m5(){…}
}
```

Dependent Multiple Inheritance

```
class A{
    field a1;
    field a2;
    method m1(){…}
    method m3(){…}
}

class C extends A {
    field c1;
    field c2;
    method m1(){…}
    method m2(){…}
}

class D extends A {
    field d1;

    method m3(){…}
    method m4(){…}
}

class E extends C, D {
    field e1;

    method m2() {…}
    method m4() {…}
    method m5(){…}
}
```

Dependent Inheritance

- The simple solution does not work
- The positions of nested fields do not agree

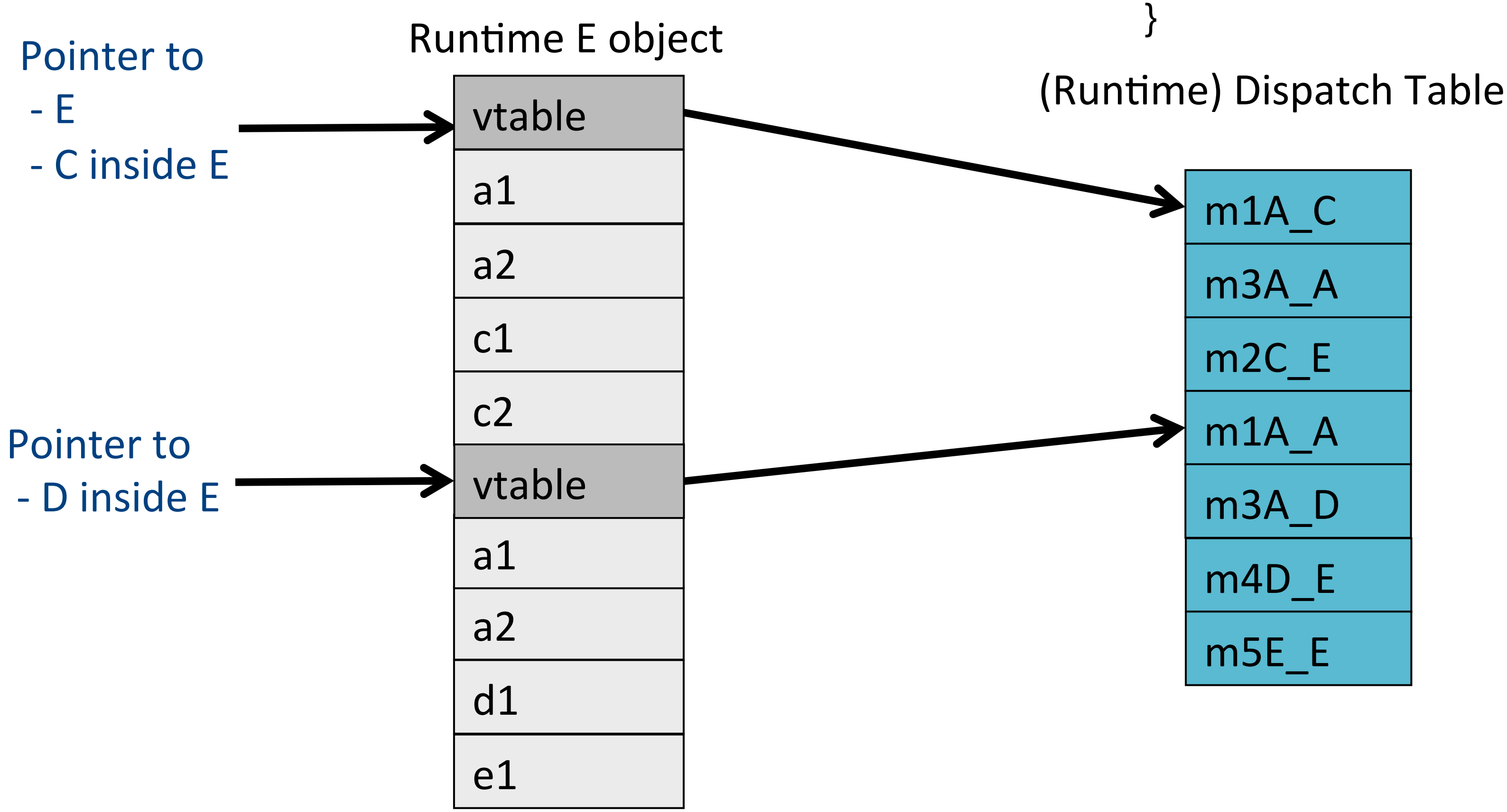
Independent Inheritance

```
class A{
  field a1;
  field a2;
  method m1(){…}
  method m3(){…}
}
```

```
class C
    extends A{
  field c1;
  field c2;
  method m1(){…}
  method m2(){…}
}
```

```
class D
    extends A{
  field d1;

  method m3(){…}
  method m4(){…}
}
```

```
class E
    extends C,D{
  field e1;

  method m2()  {…}
  method m4()  {…}
  method m5(){…}
}
```



Implementation

- Use an index table to access fields
- Access offsets indirectly

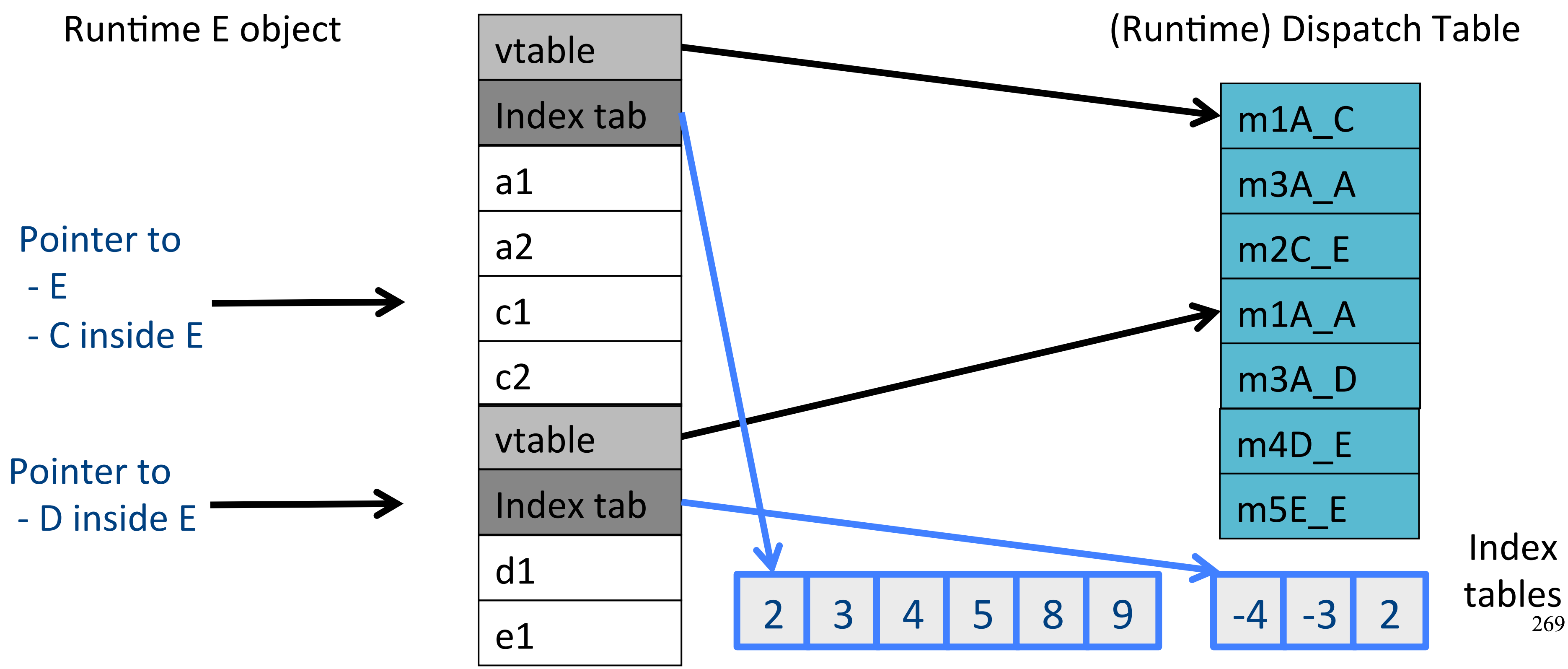
Implementation

```
class A{
  field a1;
  field a2;
  method m1(){…}
  method m3(){…}
}
```

```
class C
    extends A{
  field c1;
  field c2;
  method m1(){…}
  method m2(){…}
}
```

```
class D
    extends A{
  field d1;

  method m3(){…}
  method m4(){…}
}
```

```
class E
    extends C,D{
  field e1;

  method m2() {…}
  method m4() {…}
  method m5(){…}
}
```



Class Descriptors

- Runtime information associated with instances
- Dispatch tables
  - Invoked methods
- Index tables
- Shared between instances of the same class

- Can have more (reflection)

Interface Types

- Java supports limited form of multiple inheritance
- Interface consists of several methods but no fields

```
public interface Comparable {
    public int compare(Comparable o);
}
```

- A class can implement multiple interfaces
  Simpler to implement/understand/use
- Implementation: record with 2 pointers:
  - A separate dispatch table per interface
  - A pointer to the object

Dynamic Class Loading

- Supported by some OO languages (Java)
- At compile time
  - the actual class of a given object at a given program point may not be known
- Some addresses have to be resolved at runtime
- Compiling c.f() when f is dynamically loaded:
  - Fetch the class descriptor d at offset 0 from c
  - Fetch the address of the method-instance f from (constant) f offset at d into p
  - Jump to the routine at address p (saving return address)

Other OO Features

- Information hiding
  - private/public/protected fields
  - Semantic analysis (context handling)
- Testing class membership

Optimizing OO languages

- Hide additional costs
  - Replace dynamic by static binding when possible
  - Eliminate runtime checks
  - Eliminate dead fields

- Simultaneously generate code for multiple classesà
- Code space is an issue

Summary

- OO is a programming/design paradigm
- OO features complicates compilation
  - Semantic analysis
  - Code generation
  - Runtime
  - Memory management
- Understanding compilation of OO can be useful for programmers

The End