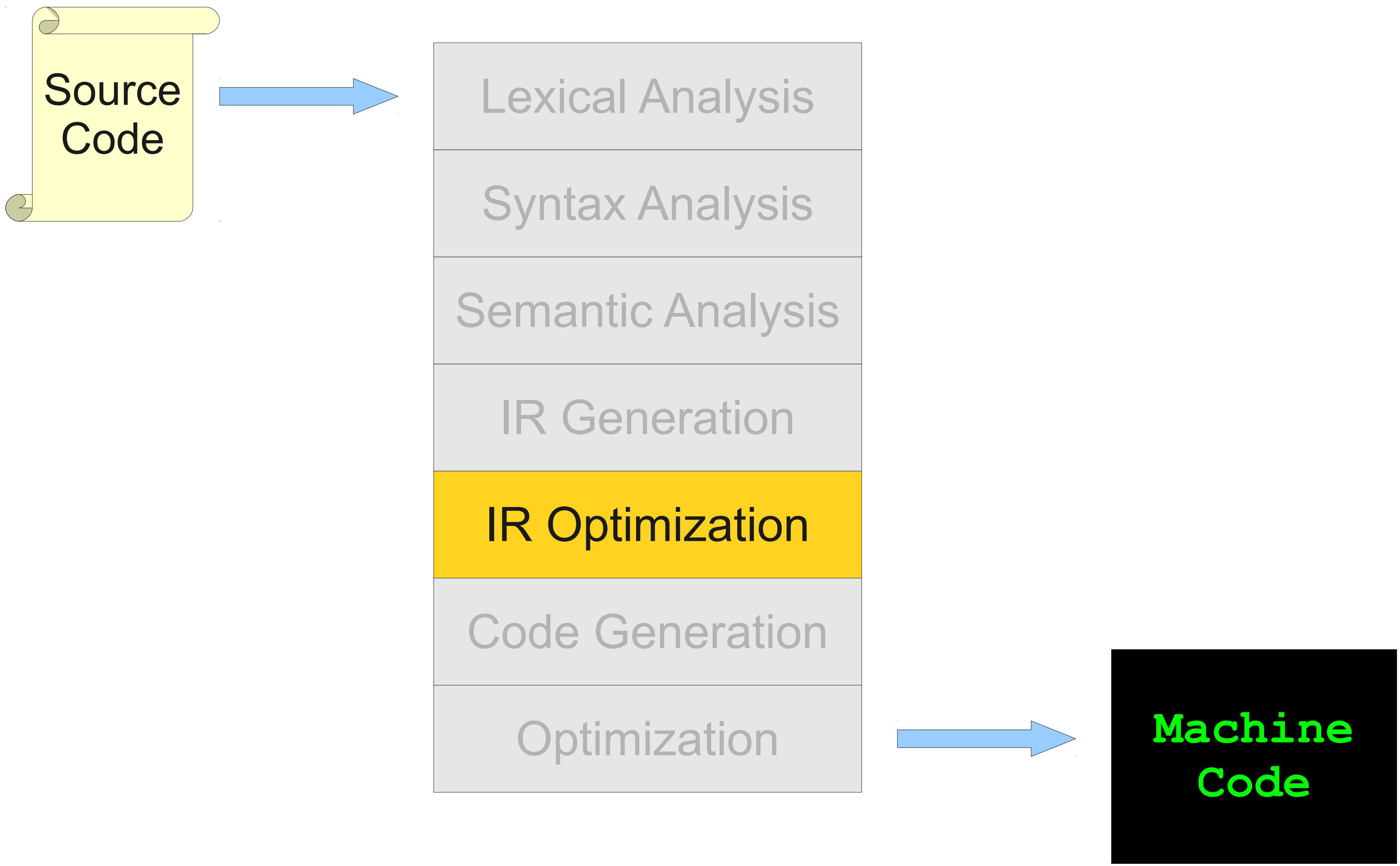
Global Optimization, Part II

Announcements

- Programming Project 4 due **Wednesday, August 10** at 11:59PM.
  - OH all this week and Sunday.
  - Ask questions via email!
  - Ask questions via Piazzza!
- Programming Assignment 2 grades/feedback available on Paperless.
  - Mean: **91/100**
  - Median: **96/100**
  - Stdev: **15**

Where We Are


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

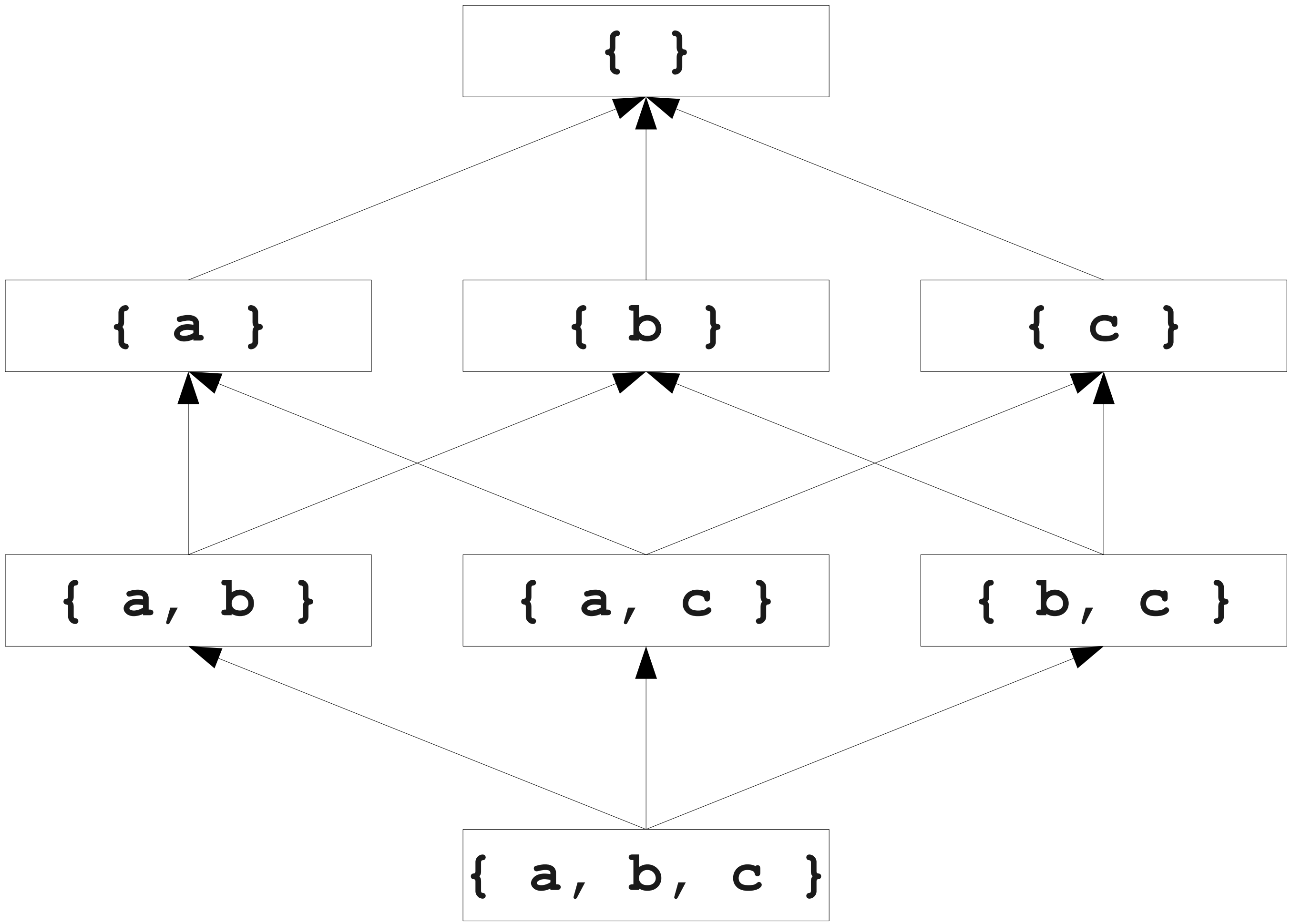
Review: Why Global Analysis is Hard

- 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.

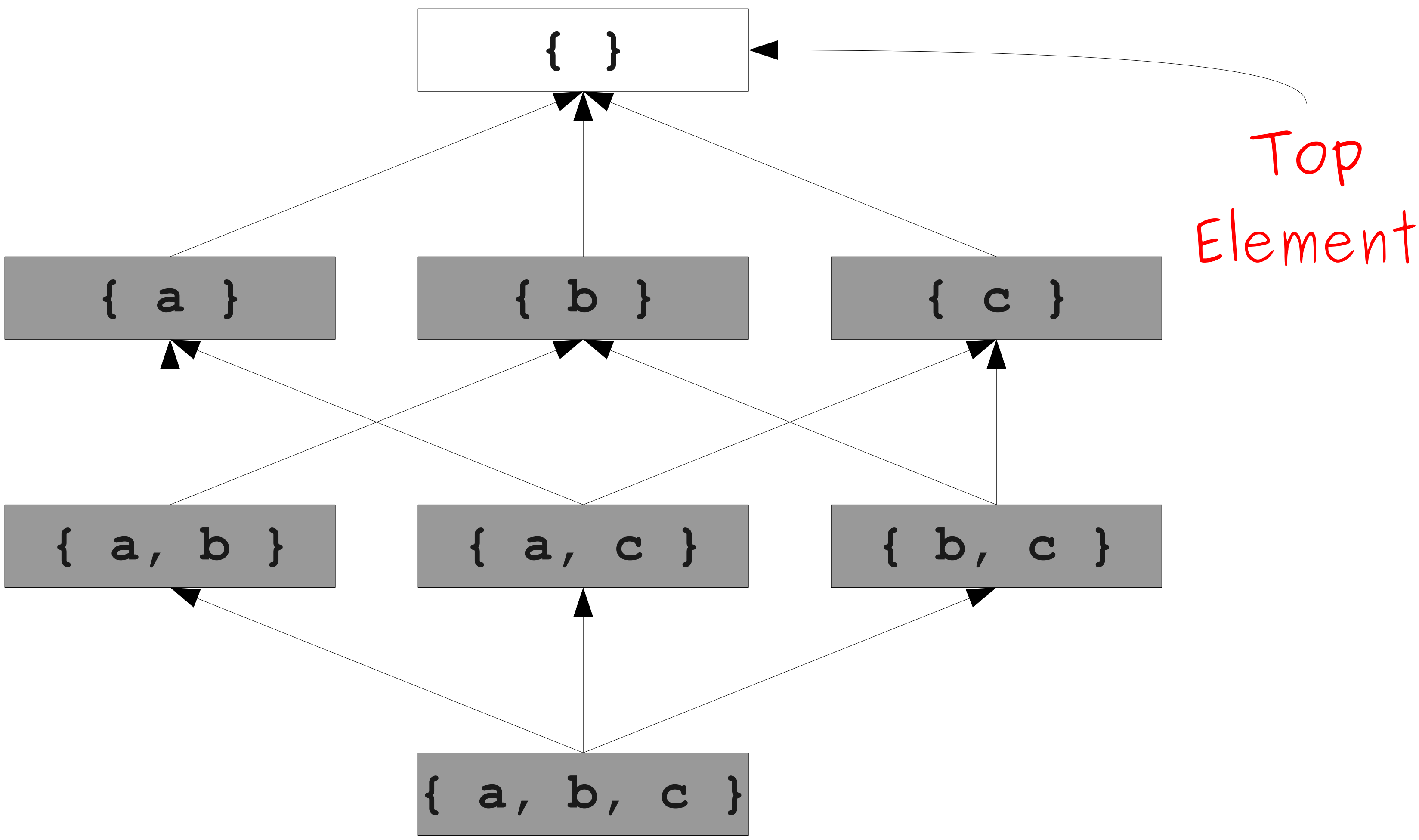
Review: Meet Semilattices

- A **meet semilattice** is a ordering defined on a set of elements.
- Any two elements have some **meet** that is the largest element smaller than both elements.
- There is a unique **top element**, which is at least as large as any other element.
- Intuitively:
  - The meet of two elements represents combining information from two elements.
  - The top element element represents “no information yet.”

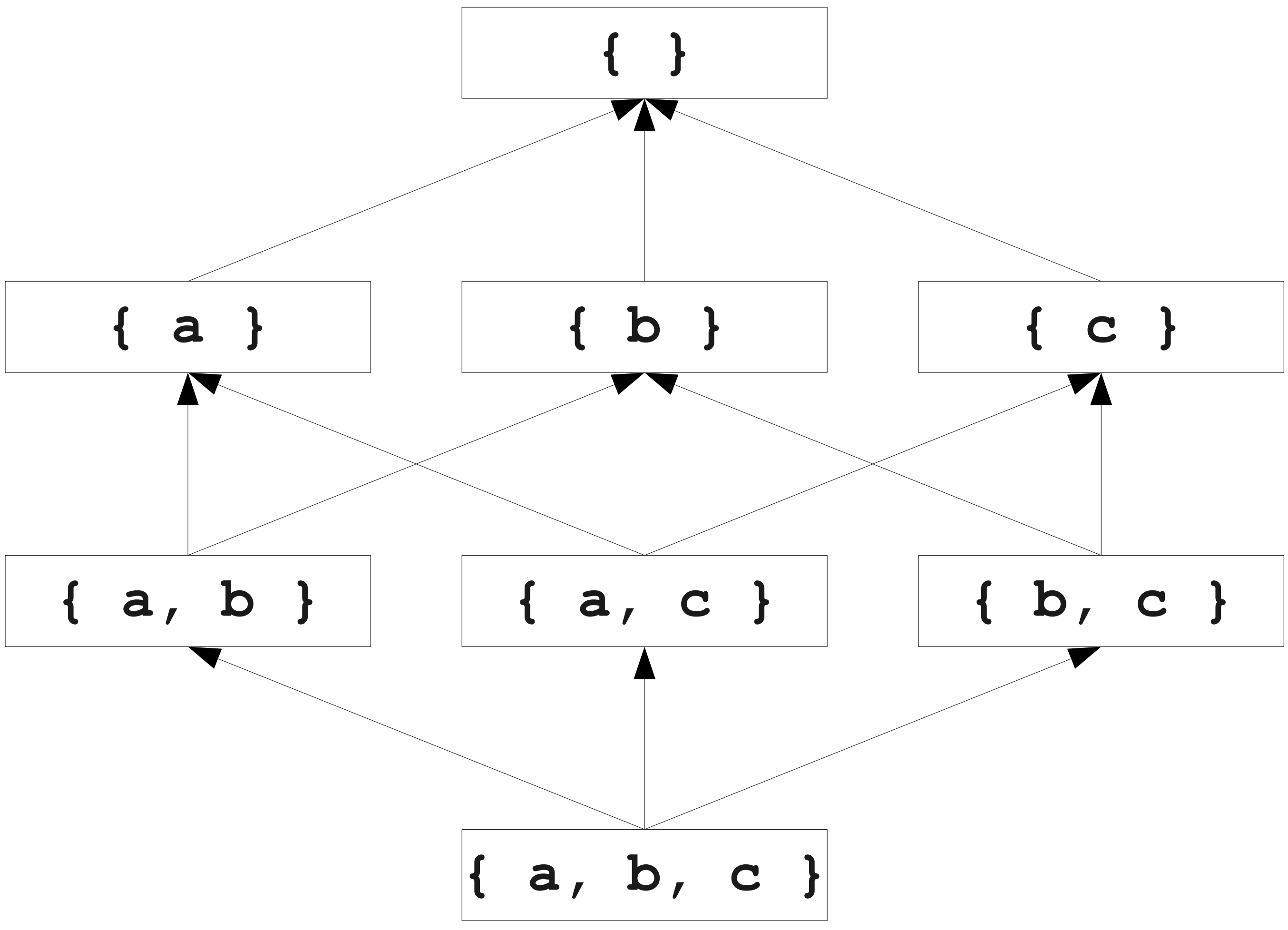
Meet Semilattices for Liveness



Meet Semilattices for Liveness



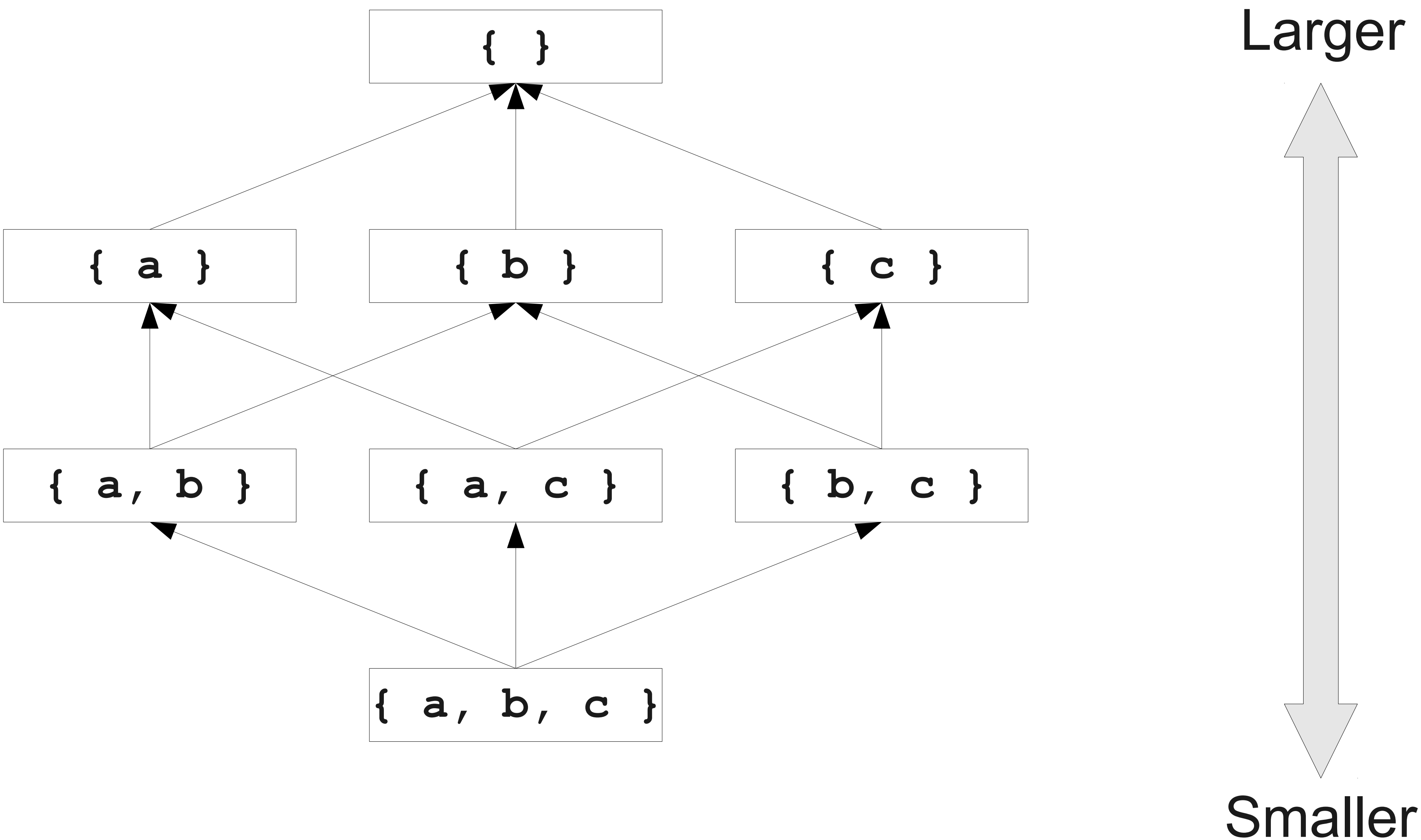
Meet Semilattices for Liveness



Review: Meet Semilattices

- A **meet semilattice** is a pair (D, ∧), where
  - D is a domain of elements.
  - ∧ is a **meet operator** that is
    - **idempotent:** $x \wedge x = x$
    - **commutative:** $x \wedge y = y \wedge x$
    - **associative:** $(x \wedge y) \wedge z = x \wedge (y \wedge z)$
- If $x \wedge y = z$, we say that z is the **meet** or (**greatest lower bound**) of x and y.
- Every meet semilattice has a **top element** denoted $\top$ such that $\top \wedge x = x$ for all x.

Meet Semilattices and Orderings



Review: Orderings on Semilattices

- Every meet semilattice $(D, \wedge)$ induces an ordering relationship $\leq$ over its elements.
- Define $\mathbf{x \leq y}$ **iff** $\mathbf{x \wedge y = x}$

An Example 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}
- Top element is 0:
  - **max**{0, a} = a
- Ordering relationship over this lattice:
  - $x \leq y$ iff $x \wedge y = x$ iff **max**{x, y} = x iff x is larger than y.

A 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:
  - $(x \cup y) \cup z = x \cup (y \cup z)$
- Top element:
  - The empty set: $\{ \} \cup x = x$
- Ordering relationship over this lattice:
  - $x \leq y$ iff $x \wedge y = x$ iff $x \cup y = x$ iff $x \supseteq y$.

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?
  - Use the meet of all of those blocks' information.
- What value do we give to basic blocks we haven't seen yet?
  - Use the top 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, ∧, 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.
  - **∧** is a meet operator over those values.
  - **F** is a set of transfer functions $f : V \to V$
  - **I** is an initial value.
- The only difference from local analysis is the introduction of the meet operator.

Running Global Analyses

- Assume that this is a forward analysis.
- Set OUT[**s**] = ⊤ for all statements **s**.
- Set OUT[**begin**] = I.
- Repeat until no values change:
  - For each statement **s** with predecessors $\mathbf{p_1}$, $\mathbf{p}_2$, … , $\mathbf{p_n}$:
    - Set IN[**s**] = $\mathbf{p_1} \wedge \mathbf{p_2} \wedge \ldots \wedge \mathbf{p_n}$
    - Set OUT[**s**] = f_s (IN[**s**])
- The order of this iteration does not matter.

For Comparison

- Set OUT[**s**] = ⊤ for all stmts **s**.
- Set OUT[**begin**] = I.
- Repeat until no values change:
  - For each statement **s** with predecessors $\mathbf{p_1}$, $\mathbf{p}_2$, … , $\mathbf{p_n}$:
    - Set IN[**s**] = $\mathbf{p_1} \wedge \mathbf{p_2} \wedge \ldots \wedge \mathbf{p_n}$
    - Set OUT[**s**] = f_s (IN[**s**])

- Set IN[**s**] = { } for each stmt **s**.
- Set IN[**exit**] to the set of variables known to be live on exit.
- Repeat until no changes occur:
  - For each statement **s** of the form **a** = **b** + **c**
    - Set OUT[**s**] to set union of IN[**p**] for each successor **p** of **s**.
    - Set IN[**s**] to (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.

Properties of Constant Propagation

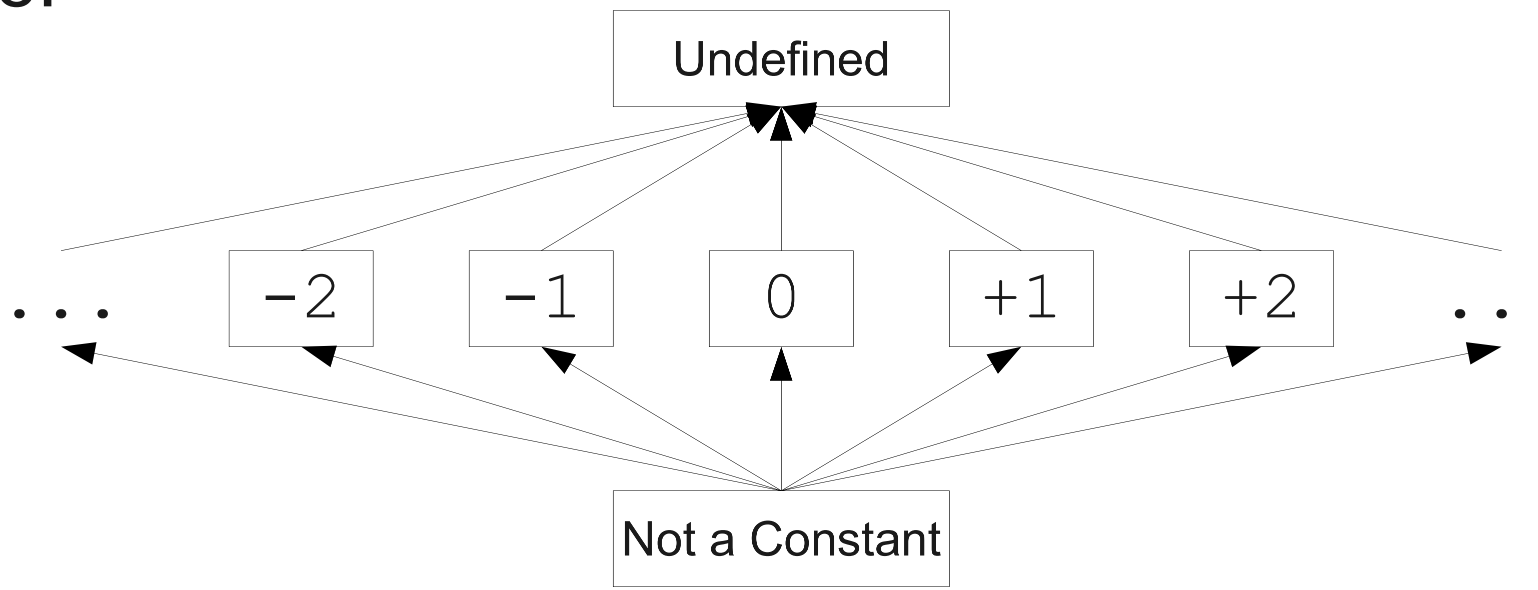
- For now, consider just some single variable **x**.
- At each point in the program, we know one of three things about the value of **x**:
  - **x** is definitely not a constant, since it's been assigned two values or assigned a value that we know isn't a constant.
  - **x** is definitely a constant 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 Meet Operator

- The meet 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.)
- The meet of **Not a Constant** and any other value **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 meet 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.)

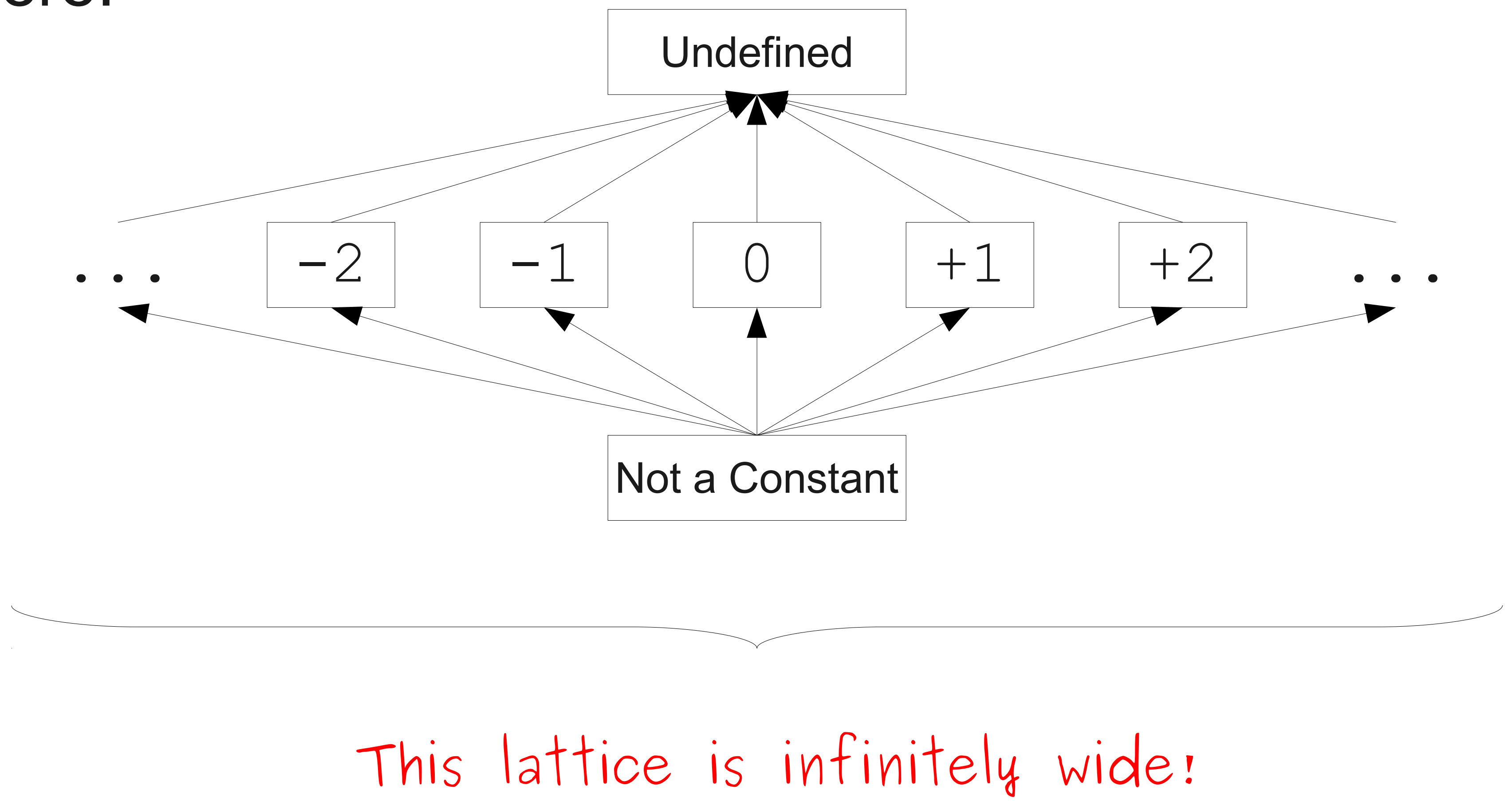
A Semilattice for Constant Propagation

- One possible semilattice for this analysis is shown here:



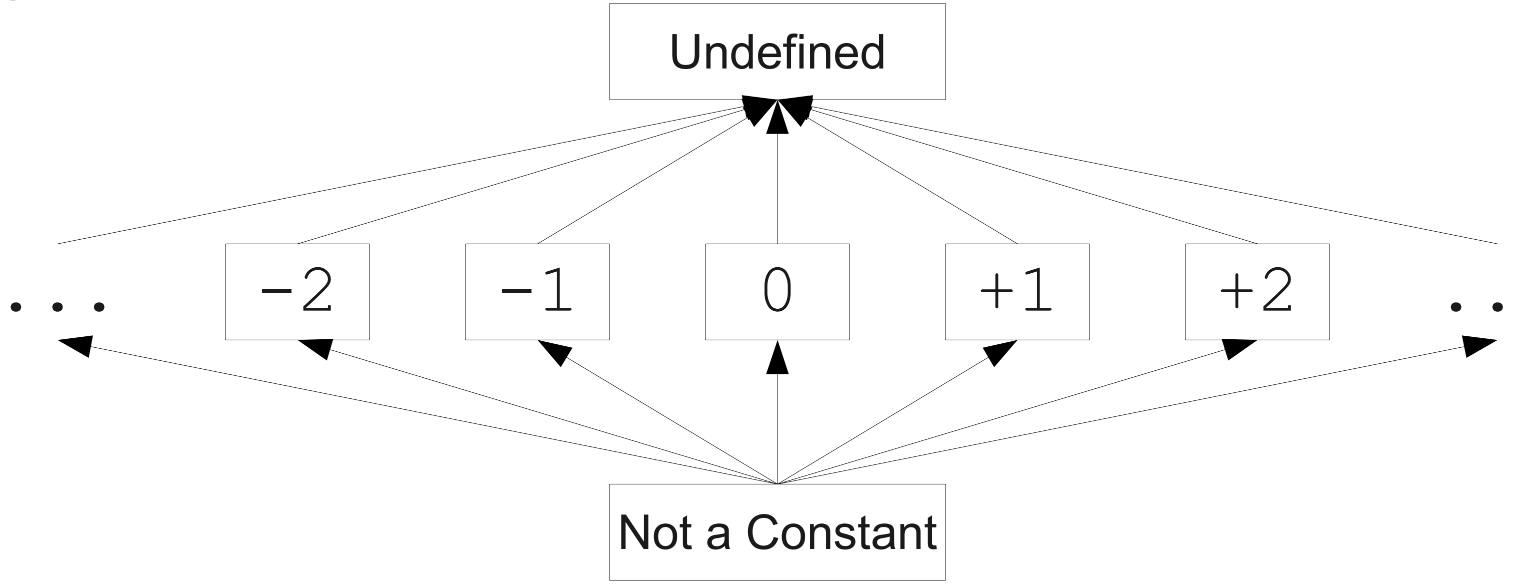
A Semilattice for Constant Propagation

- One possible semilattice for this analysis is shown here:



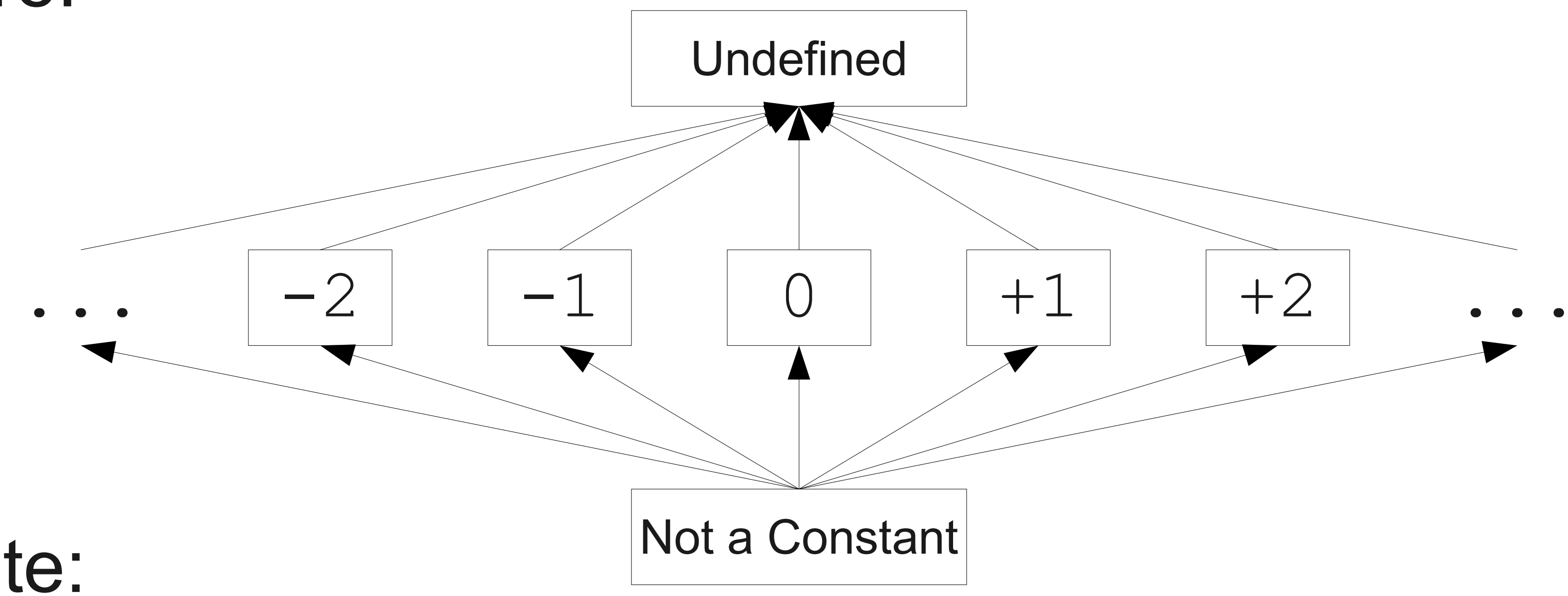
A Semilattice for Constant Propagation

- One possible semilattice for this analysis is shown here:



A Semilattice for Constant Propagation

- One possible semilattice for this analysis is shown here:

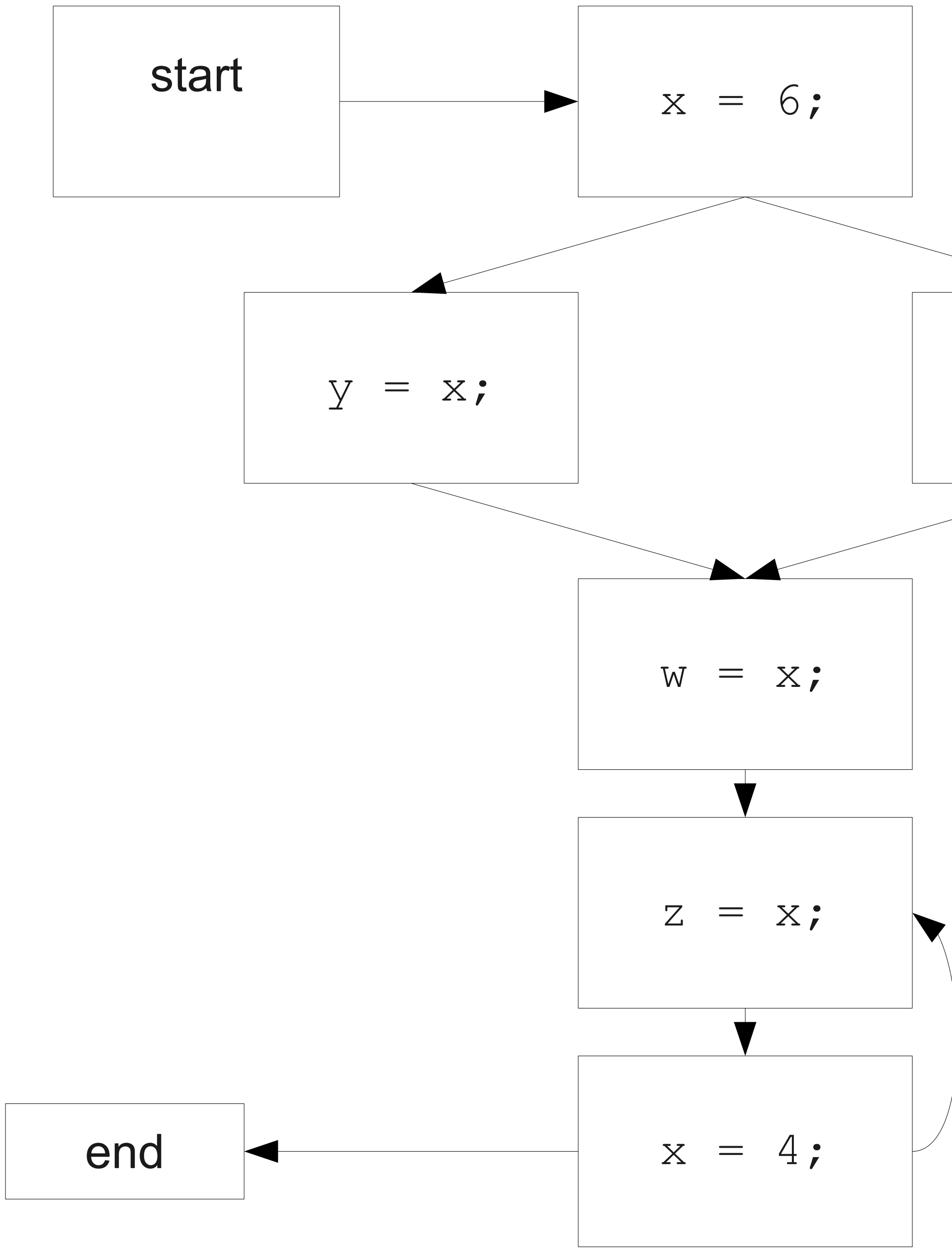


- Note:
  - The meet of any two different constants is **Not a Constant**.
  - The meet of **Undefined** and any value is that value.
  - The meet of **Not a Constant** and any value is **Not a Constant**.

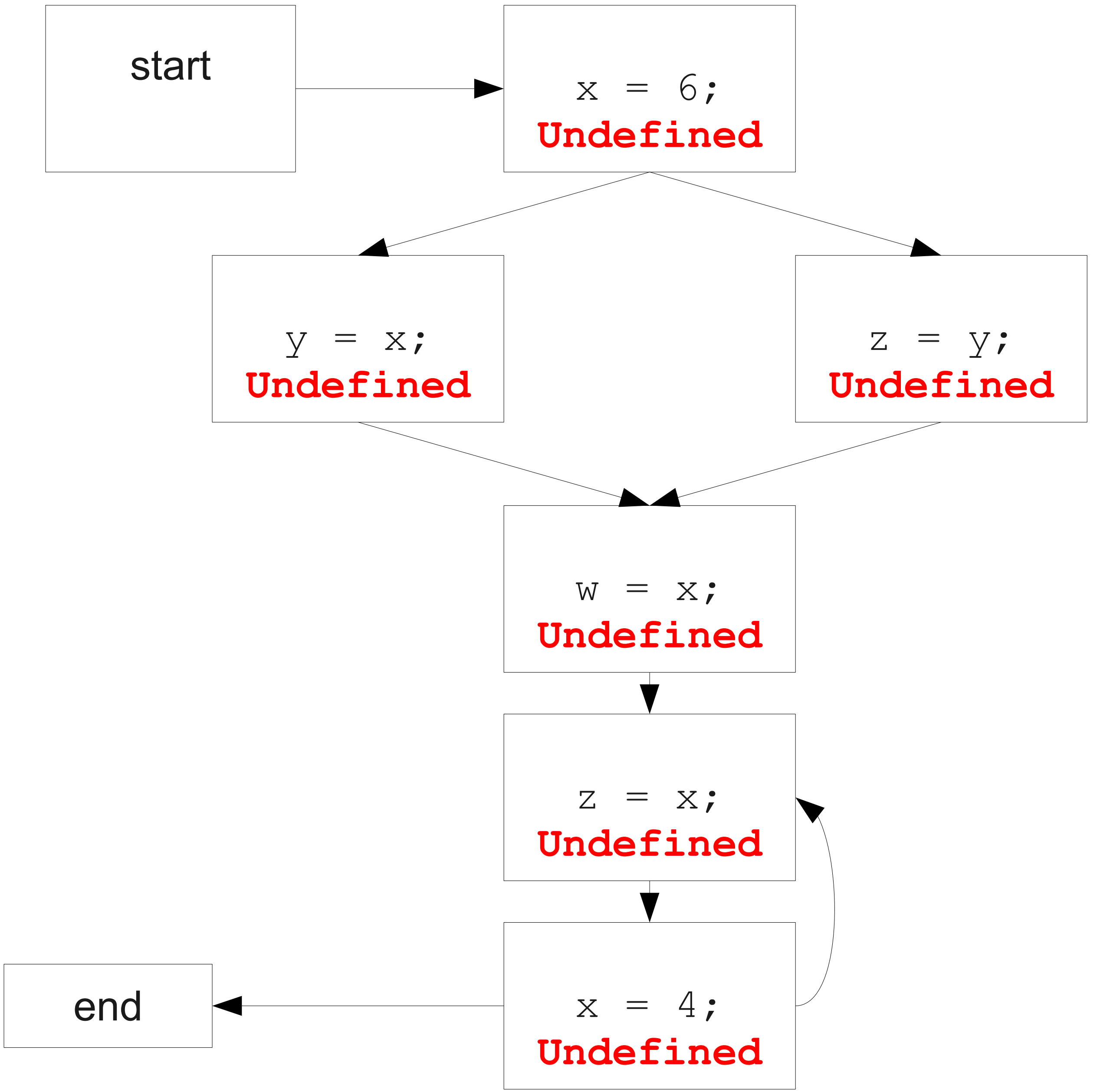
Dataflow for Constant Propagation

- Direction: **Forward**
- Semilattice: **Defined on previous slide**
- Transfer functions:
  - $f_{x=k}(V) = k$ *(assigning a constant)*
  - $f_{x=a+b}(V) =$ Not a Constant *(assigning a non-constant)*
  - $f_{y=a+b}(V) = V$ *(unrelated assignment)*
- Initial value: **x is Undefined**
  - (When might we use some other value?)

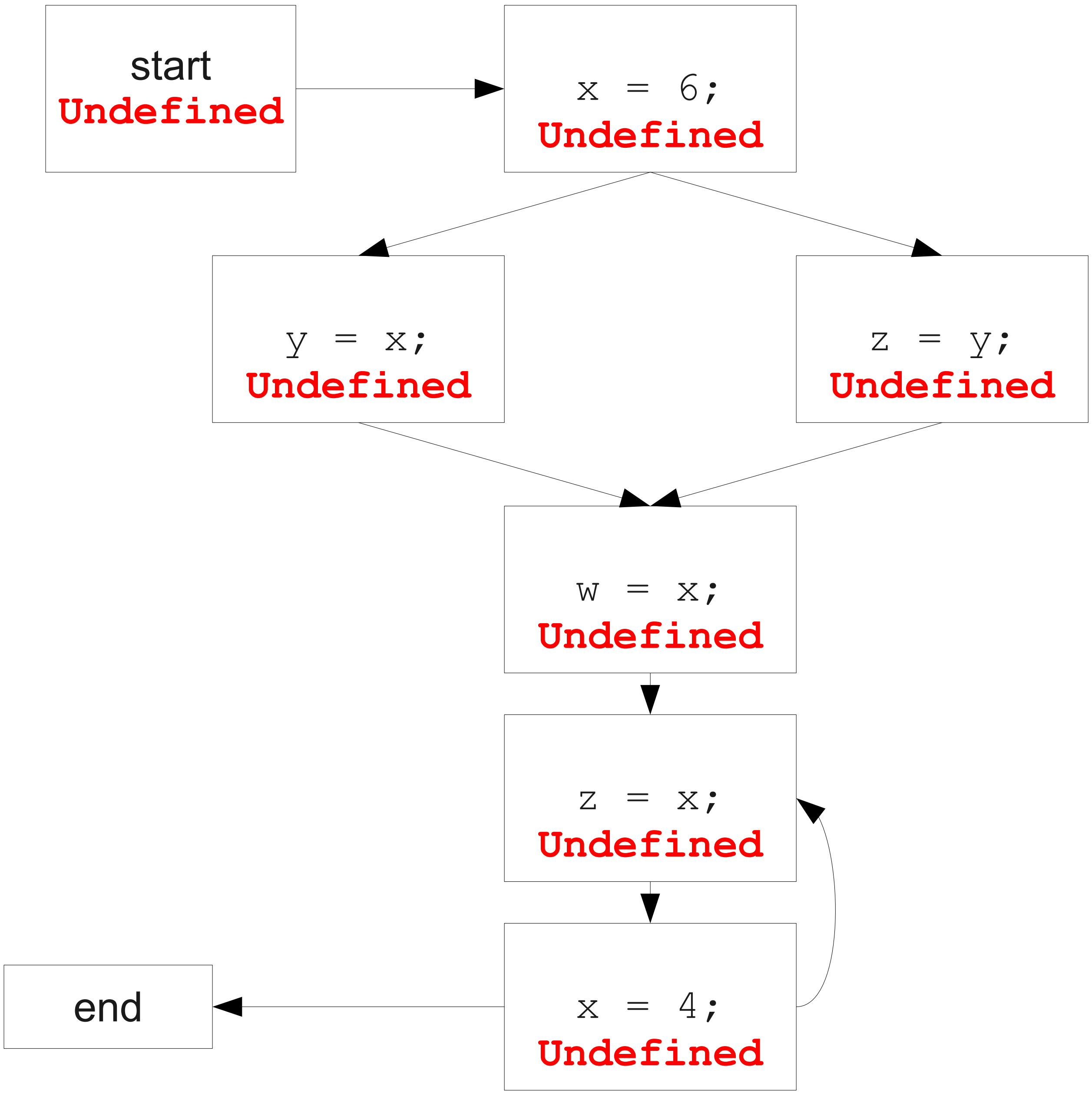
Global Constant Propagation



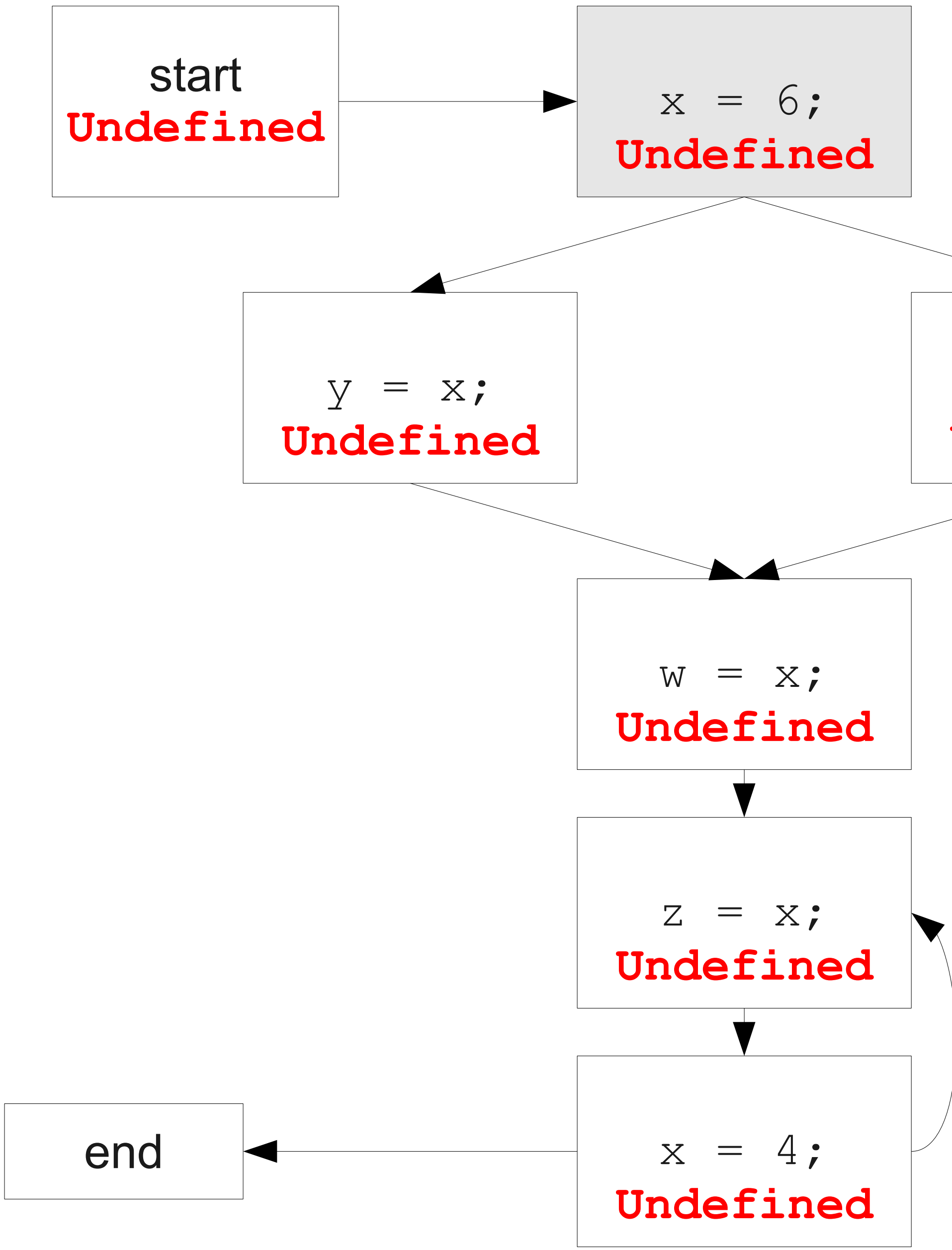
Global Constant Propagation



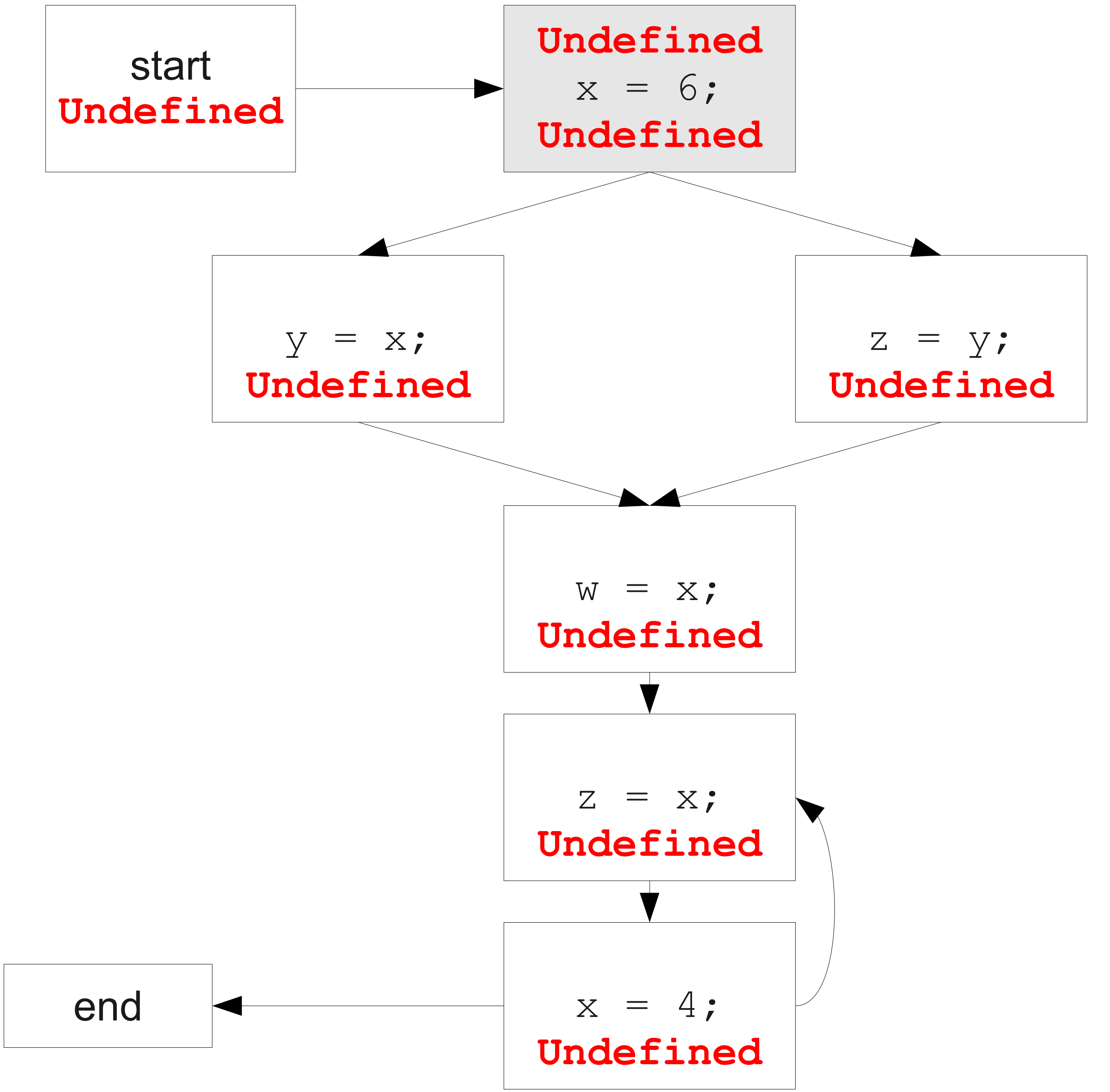
Global Constant Propagation



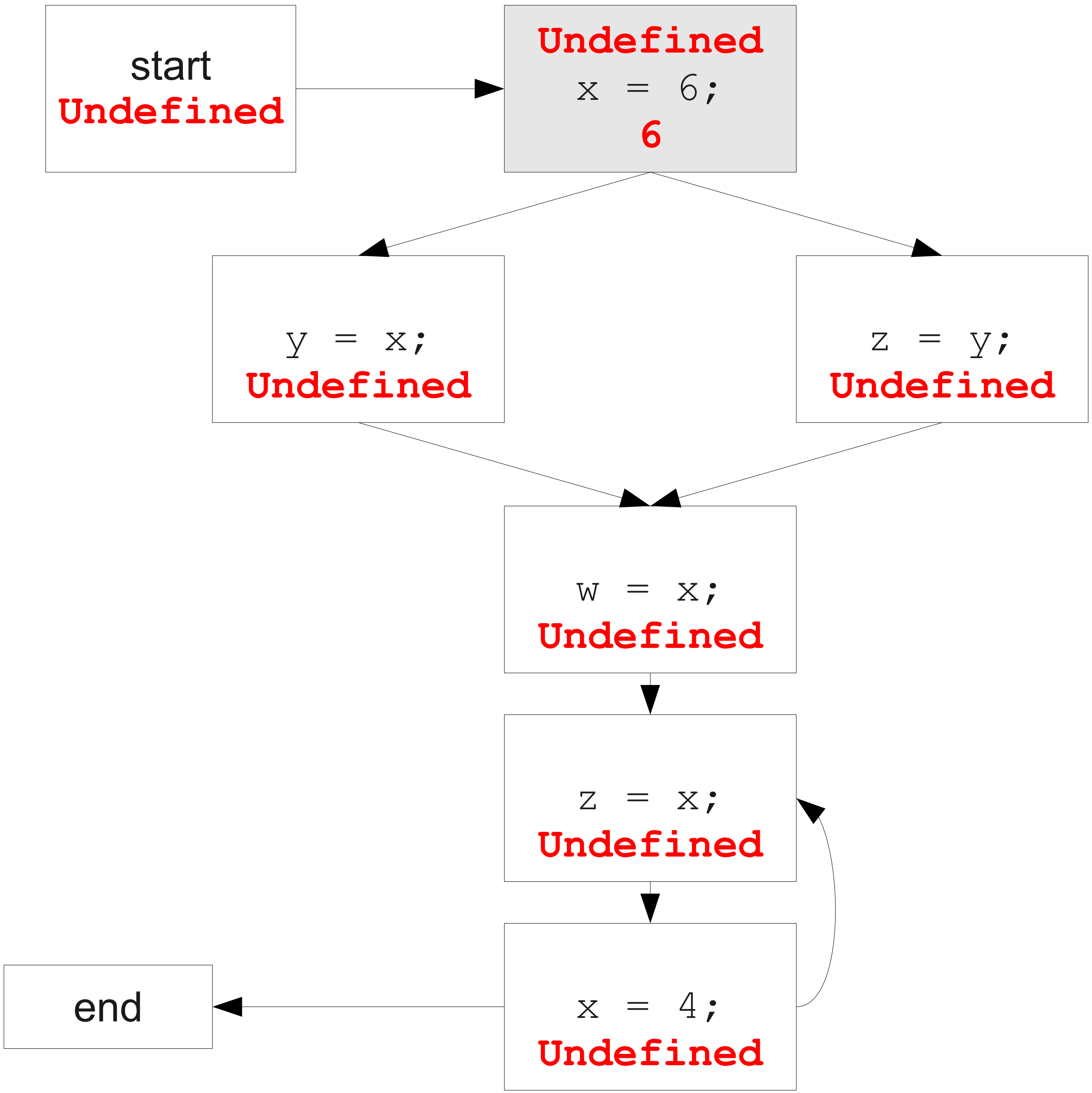
Global Constant Propagation



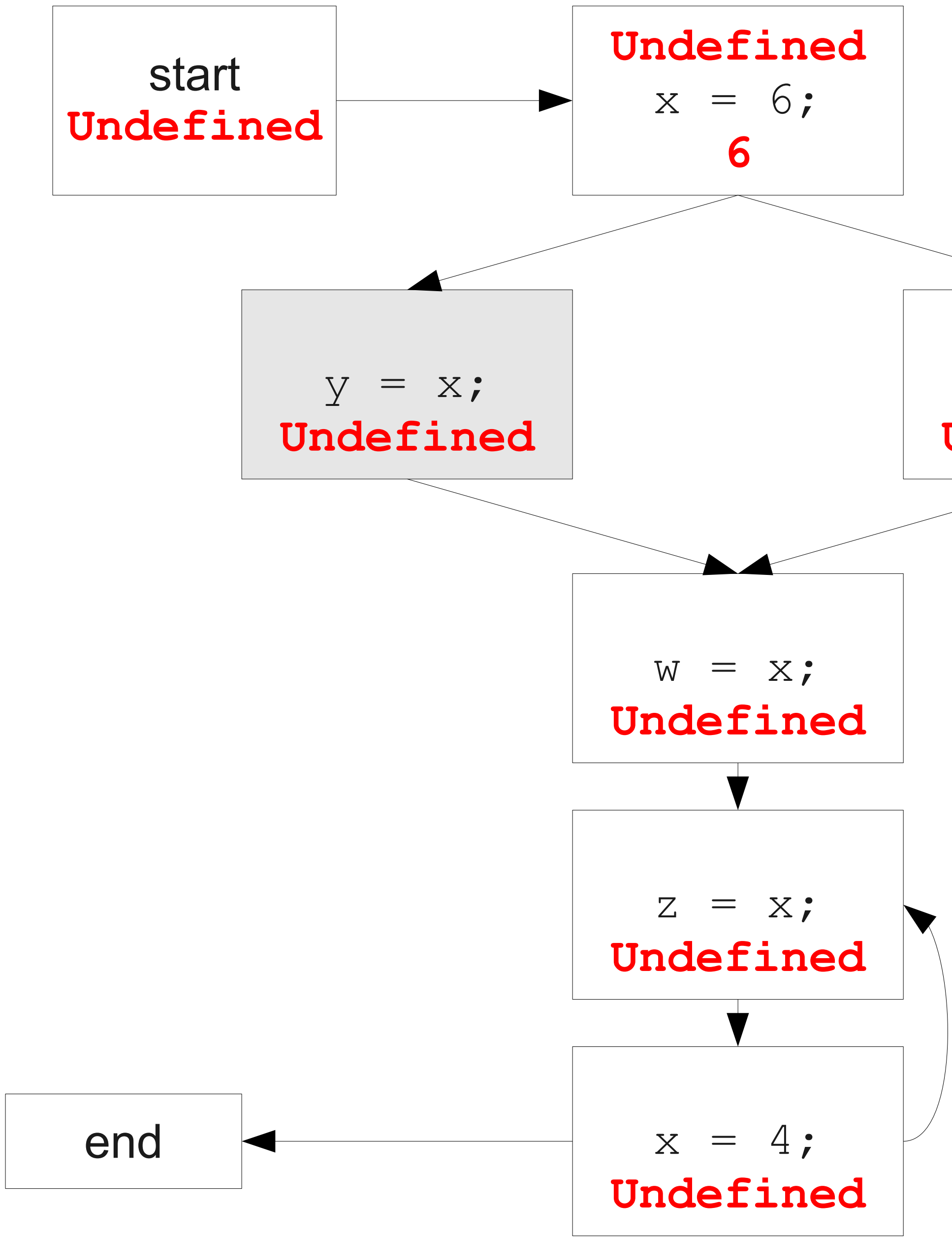
Global Constant Propagation



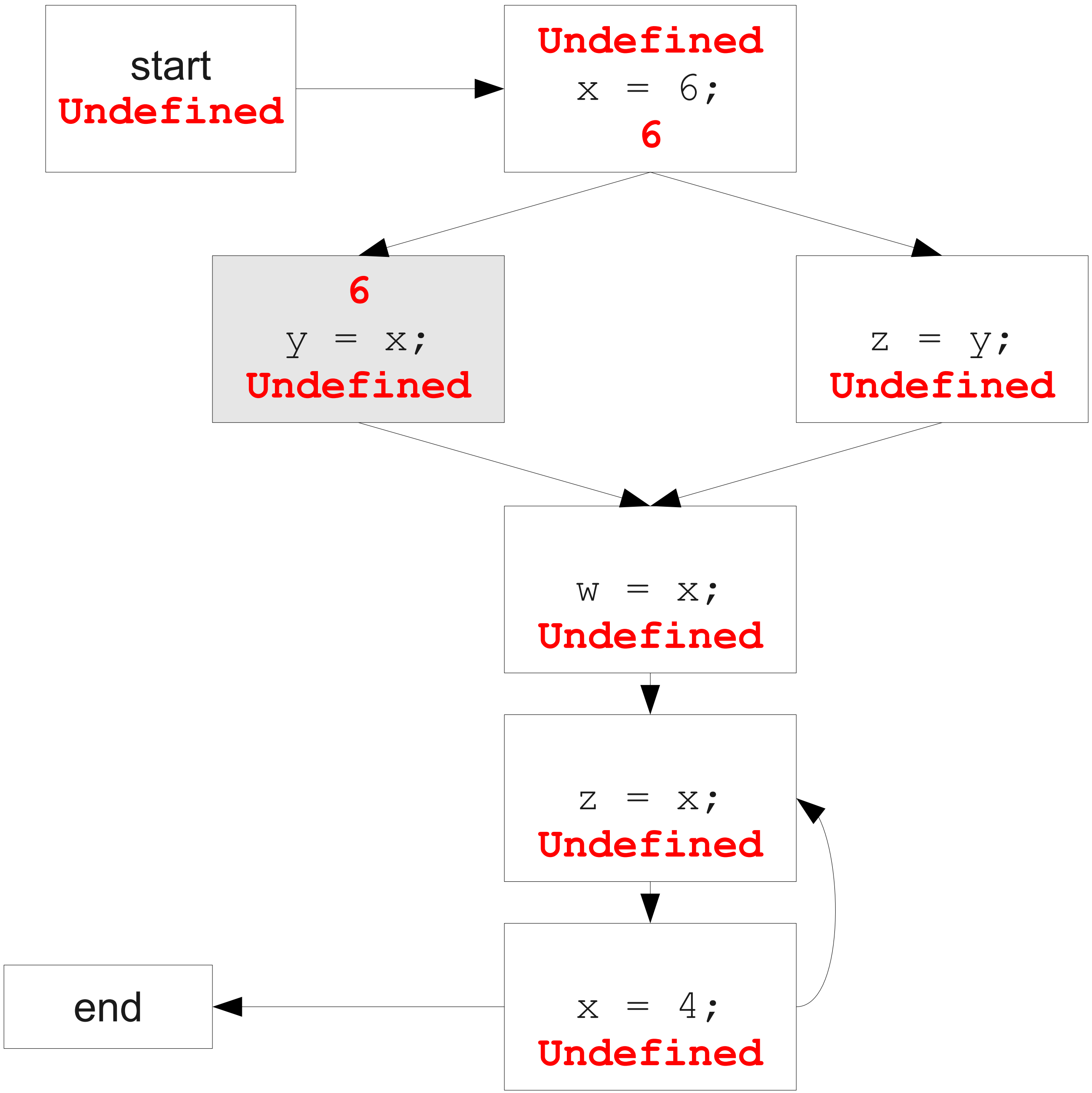
Global Constant Propagation



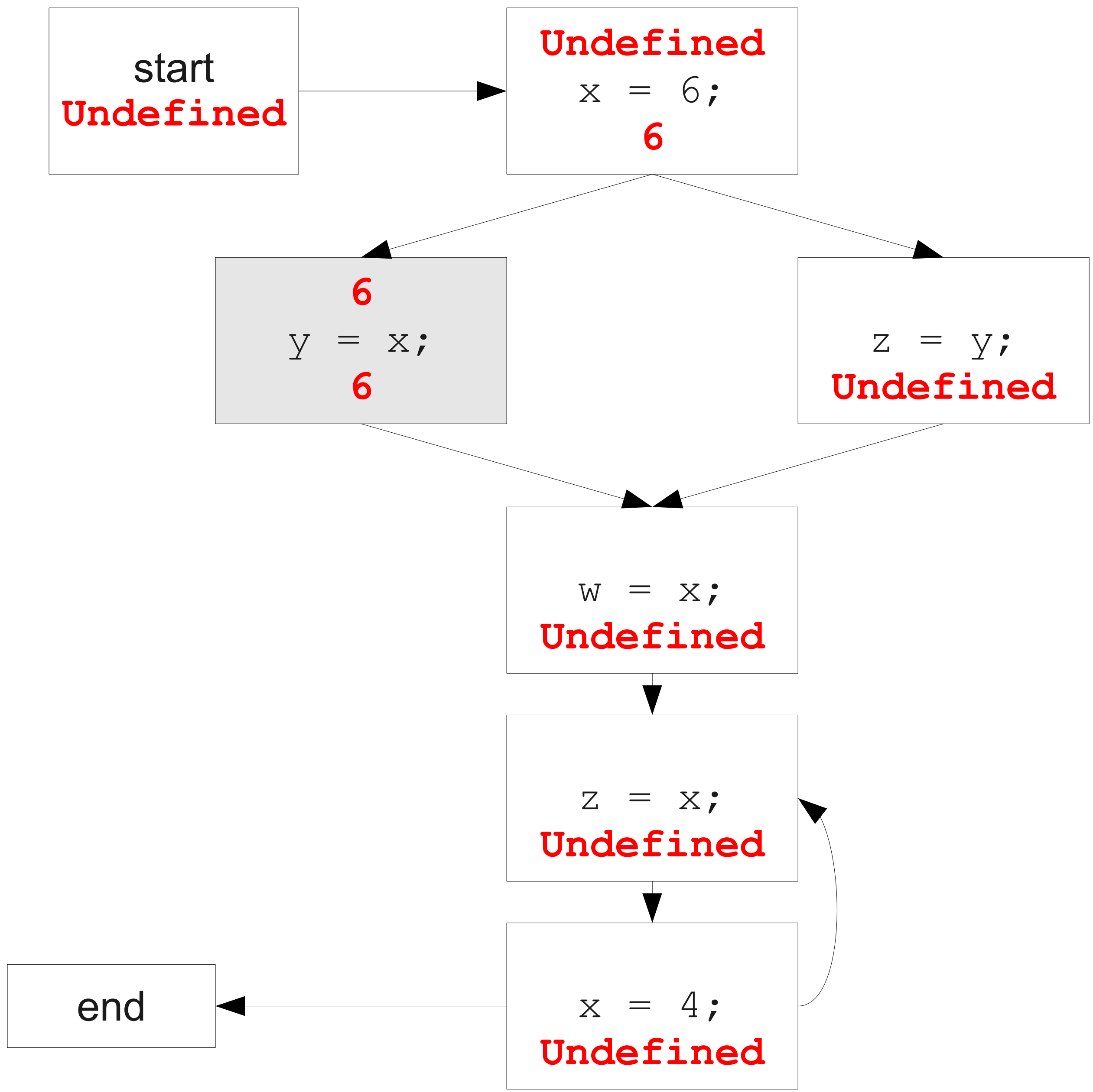
Global Constant Propagation



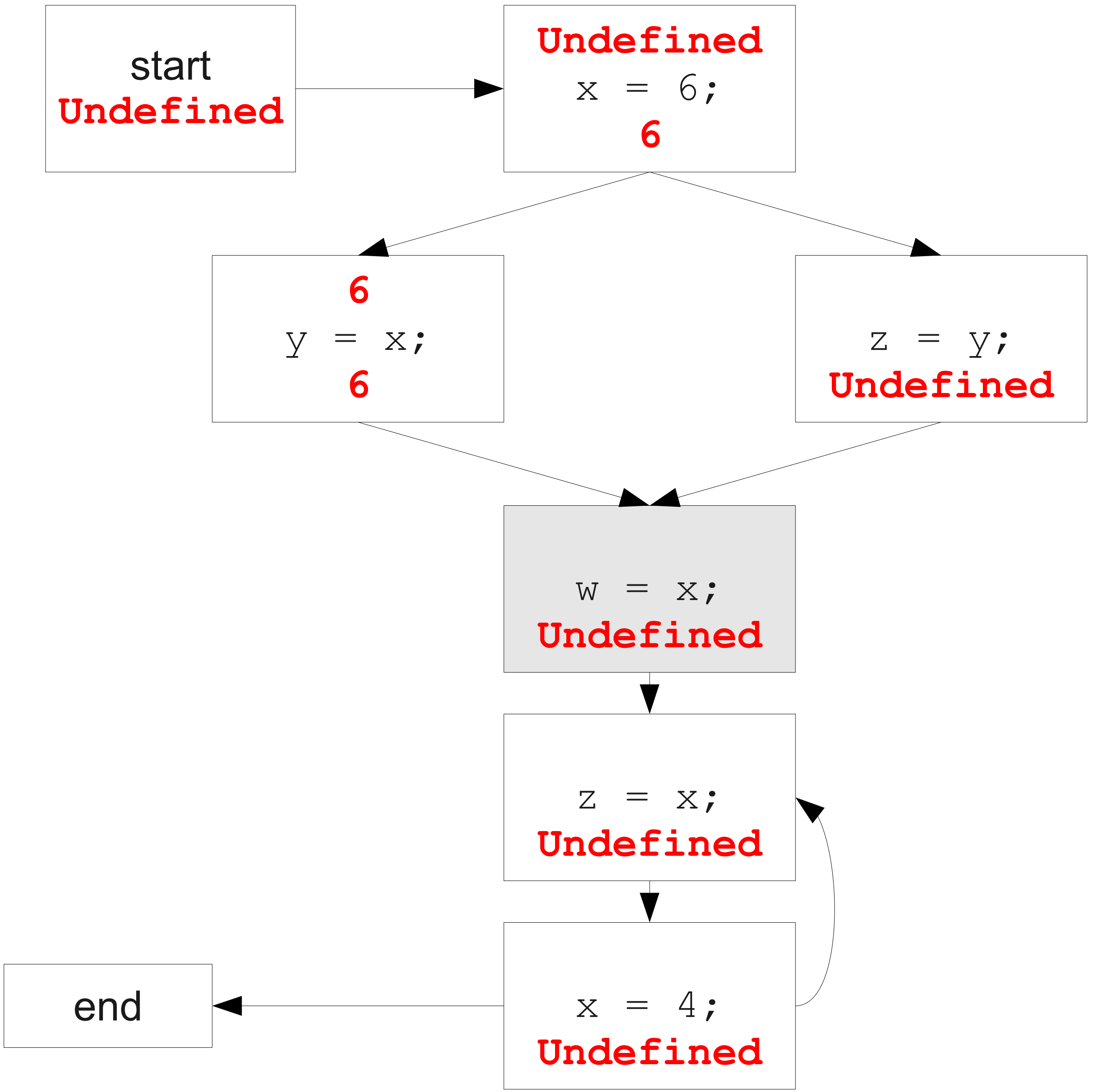
Global Constant Propagation



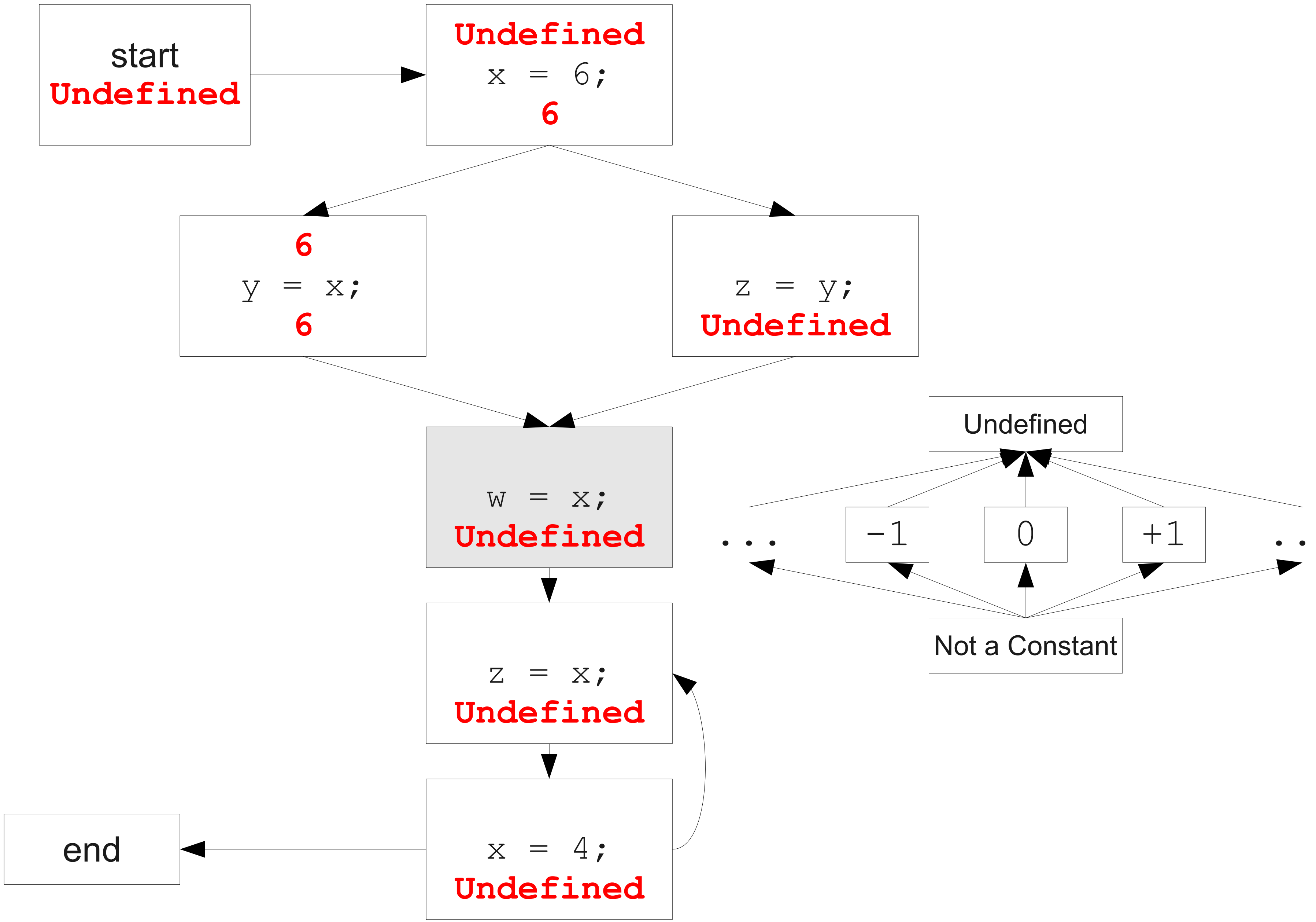
Global Constant Propagation



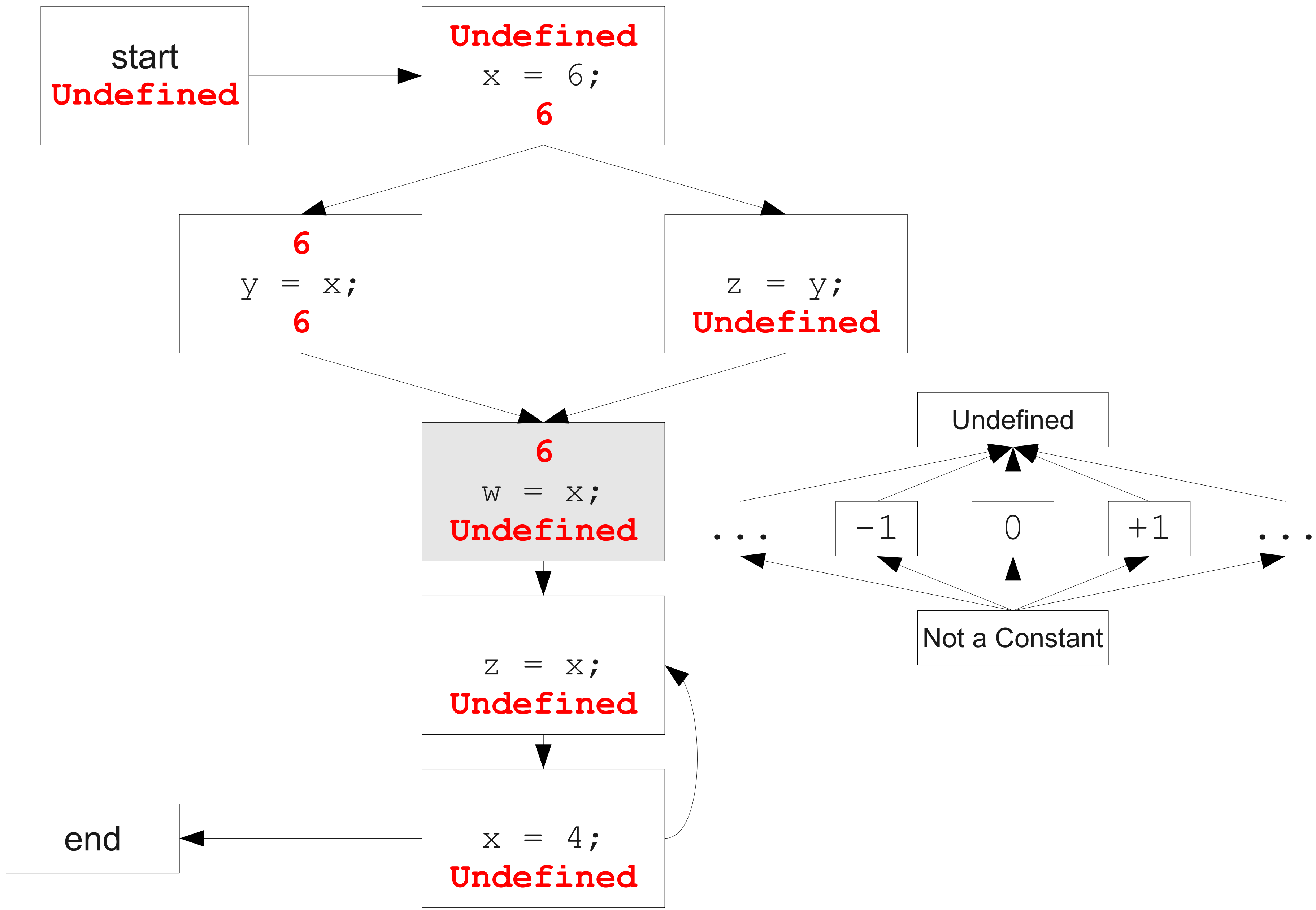
Global Constant Propagation



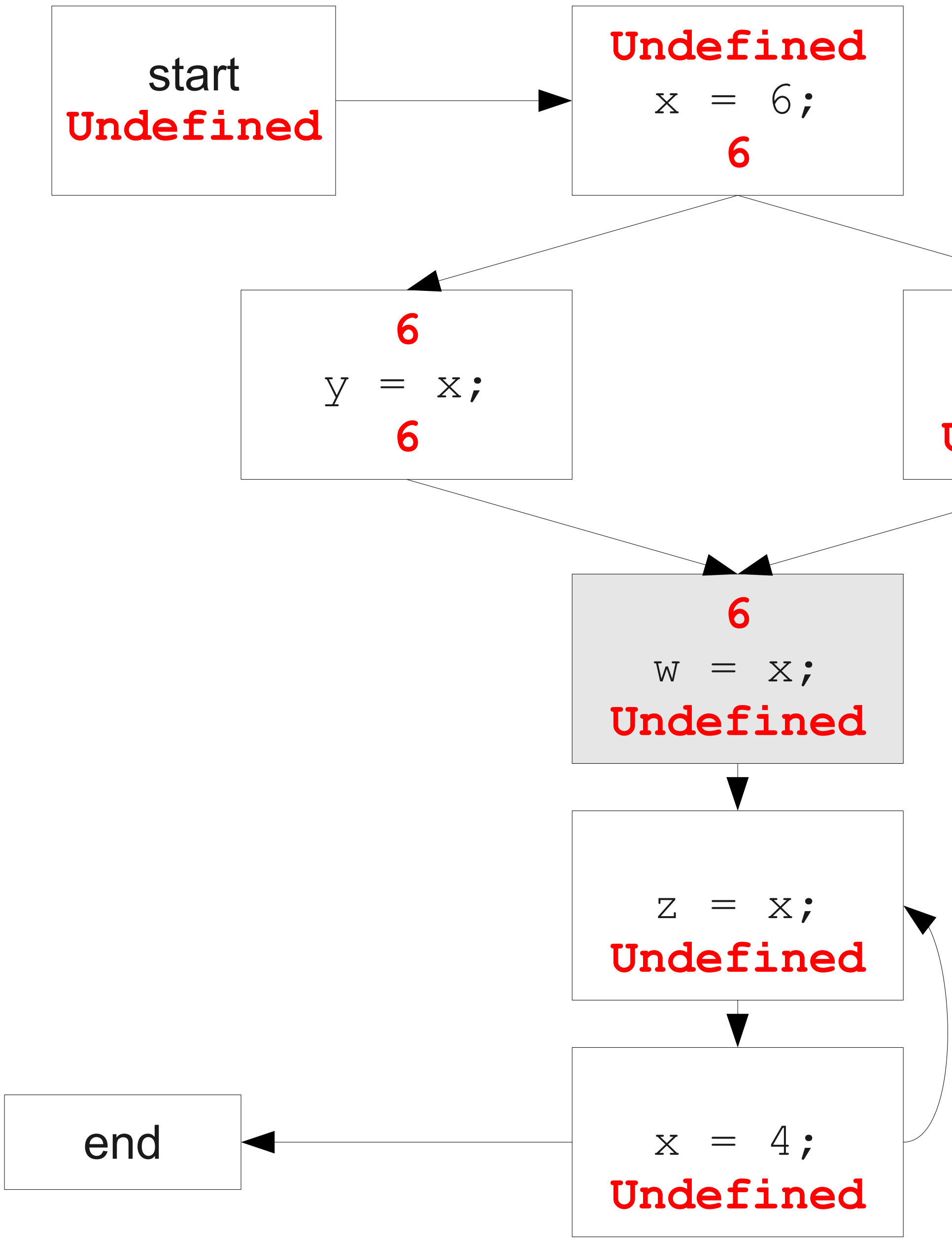
Global Constant Propagation



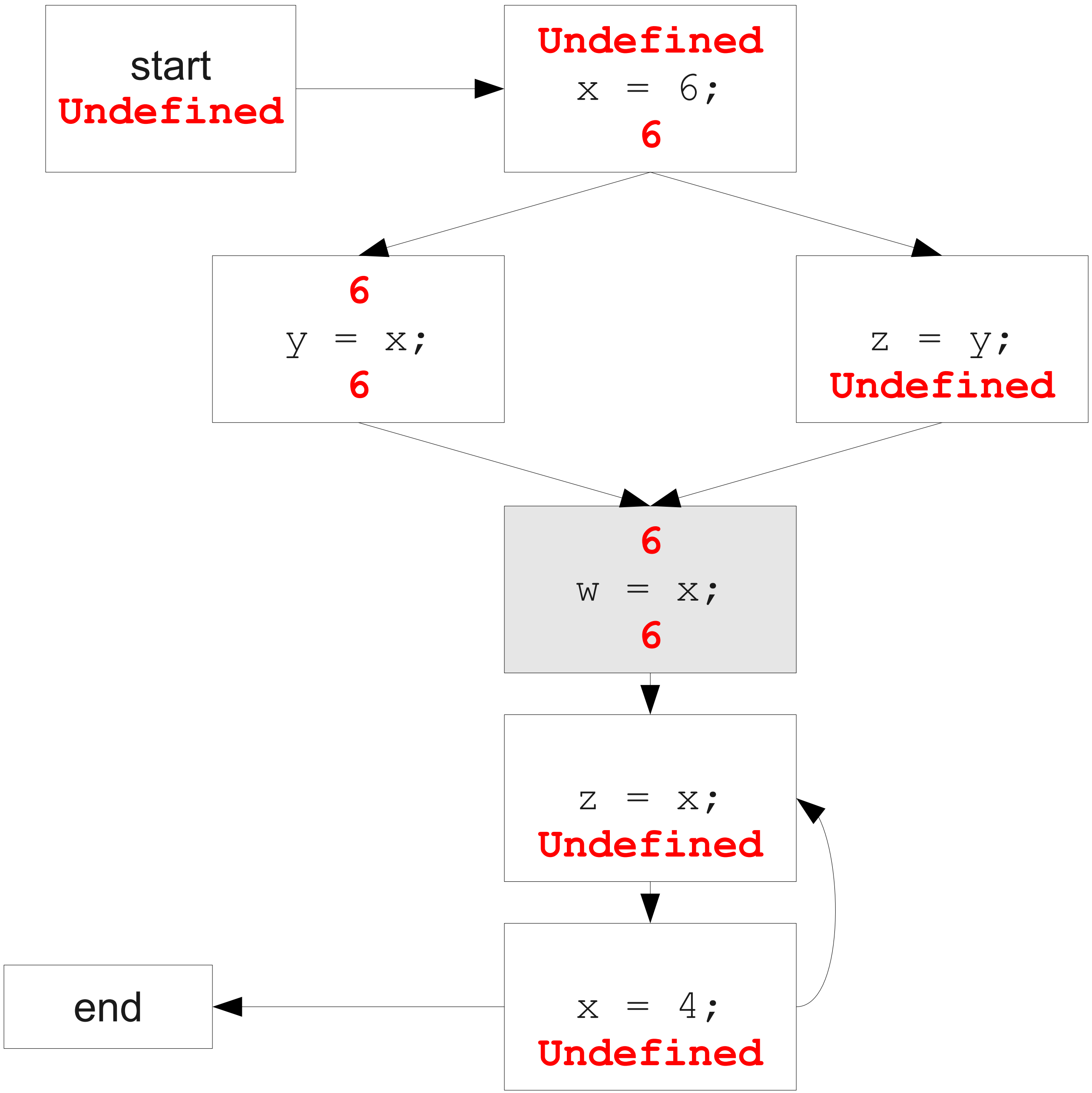
Global Constant Propagation



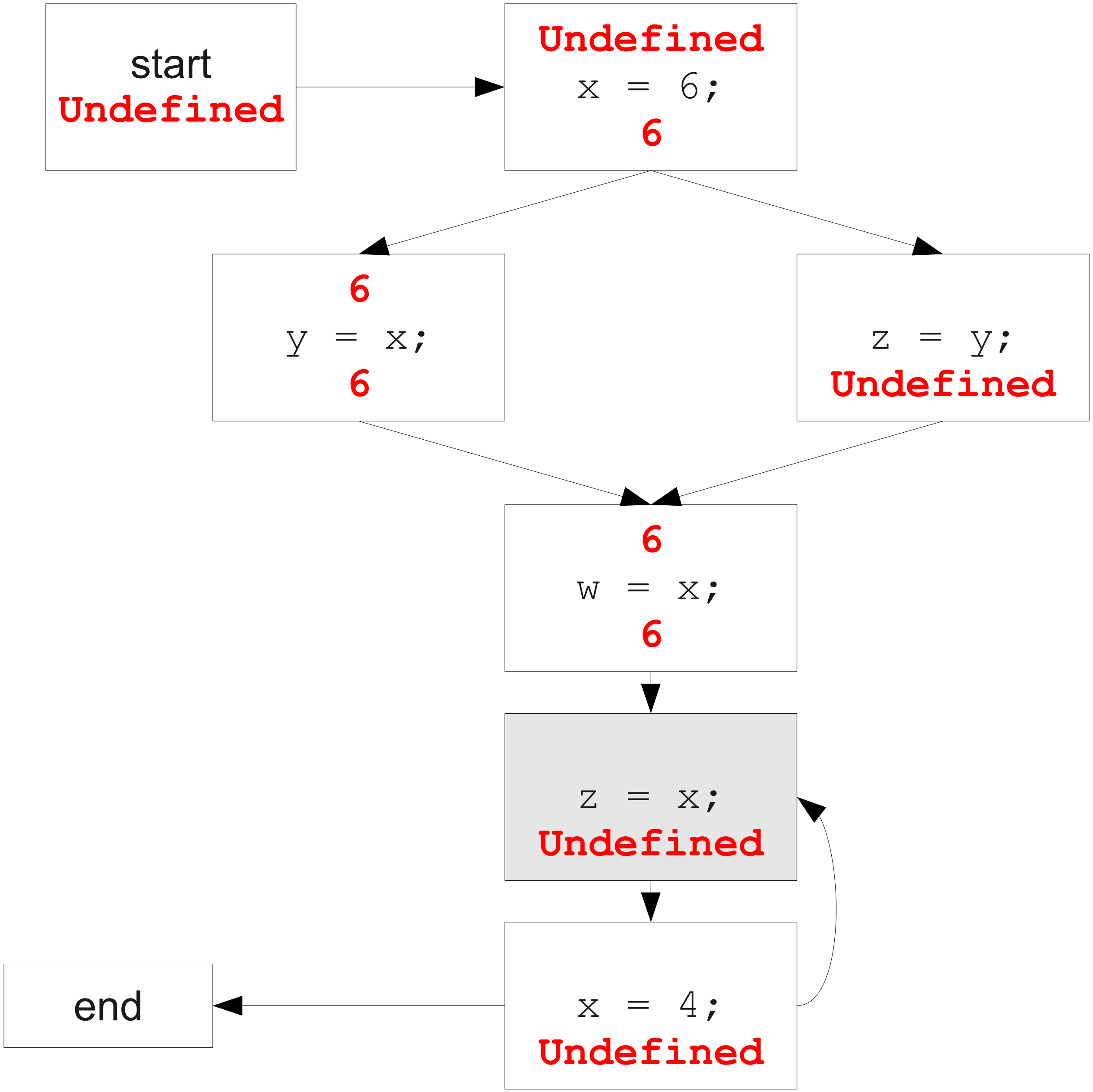
Global Constant Propagation



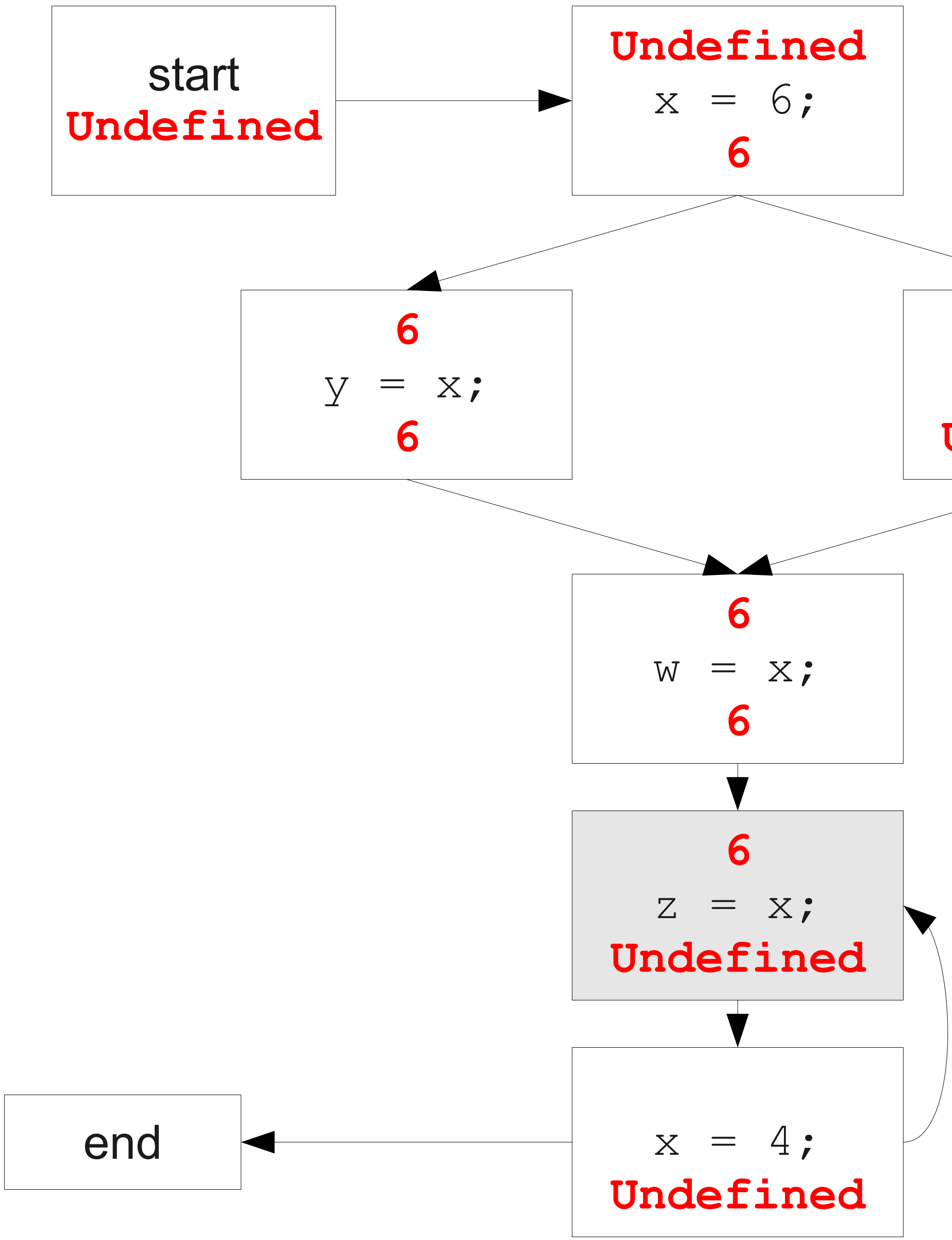
Global Constant Propagation



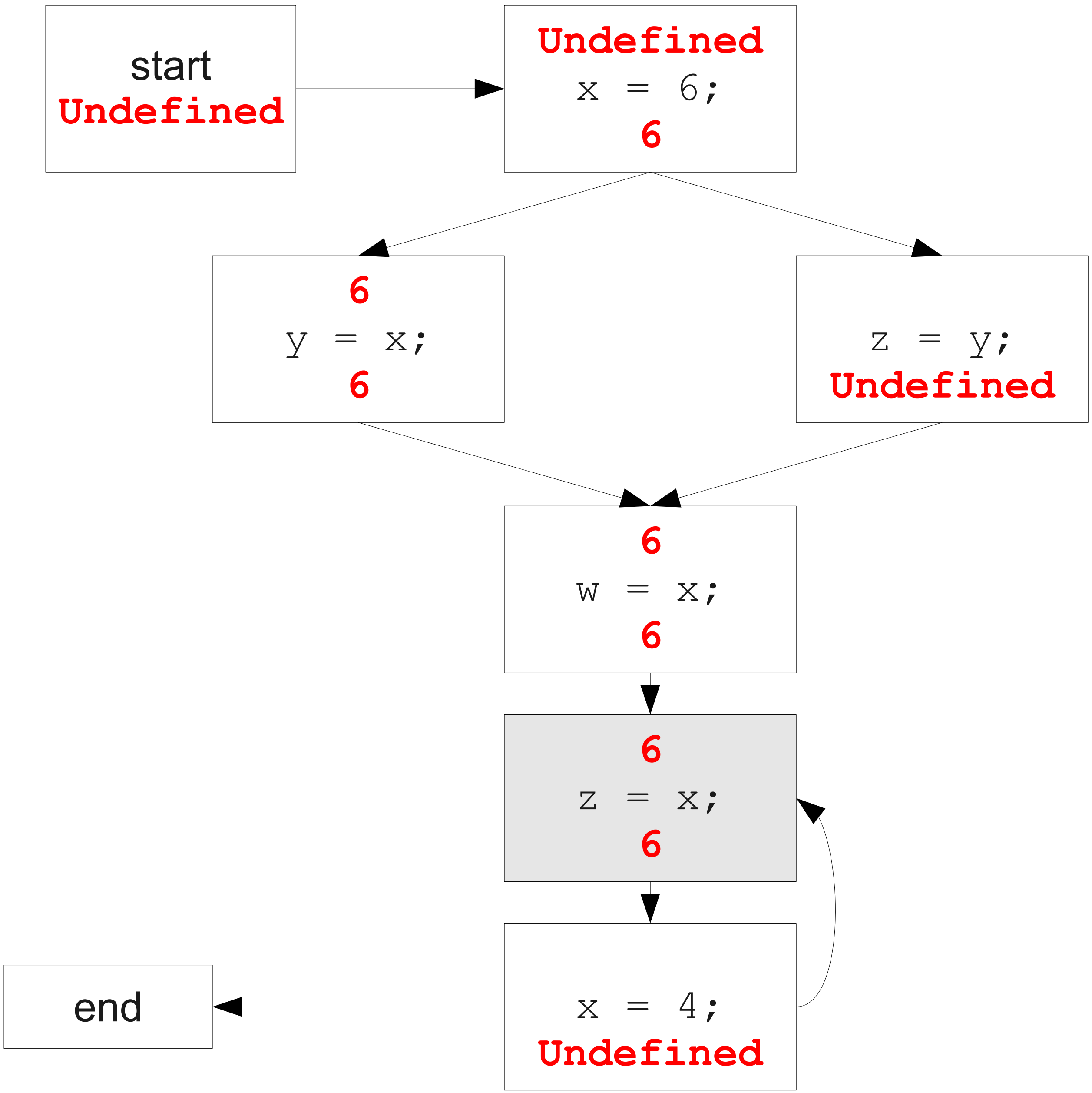
Global Constant Propagation



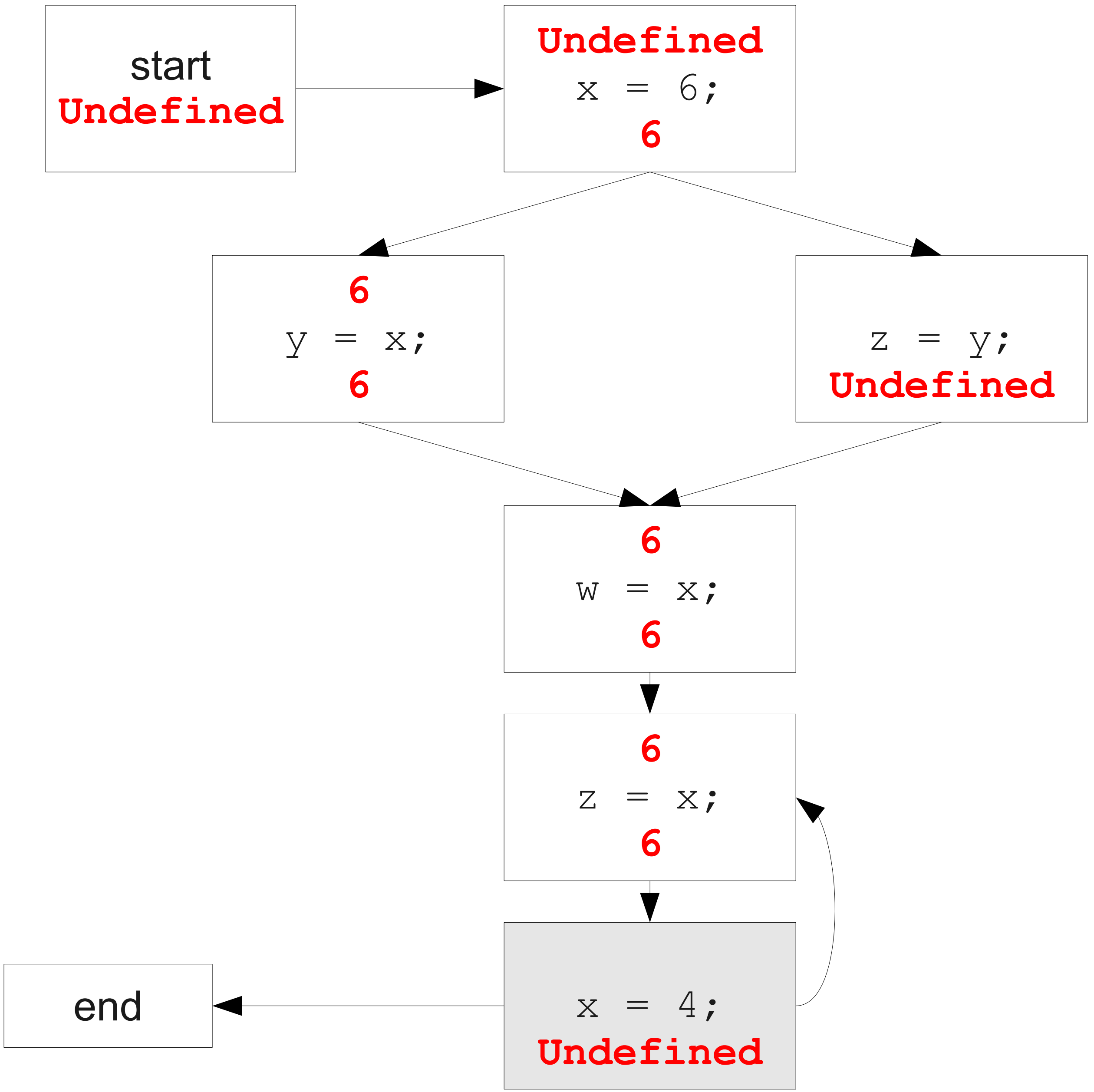
Global Constant Propagation



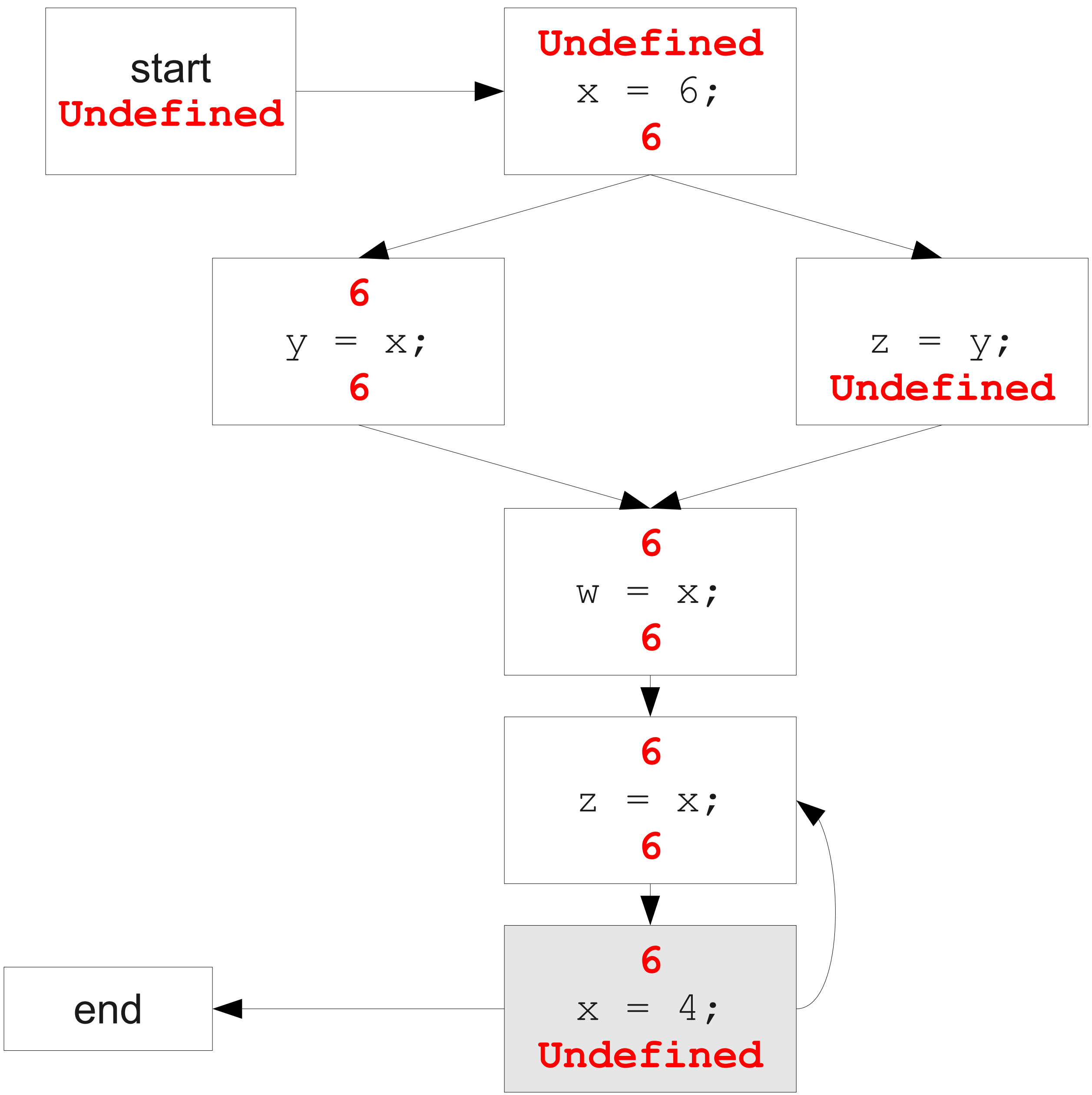
Global Constant Propagation



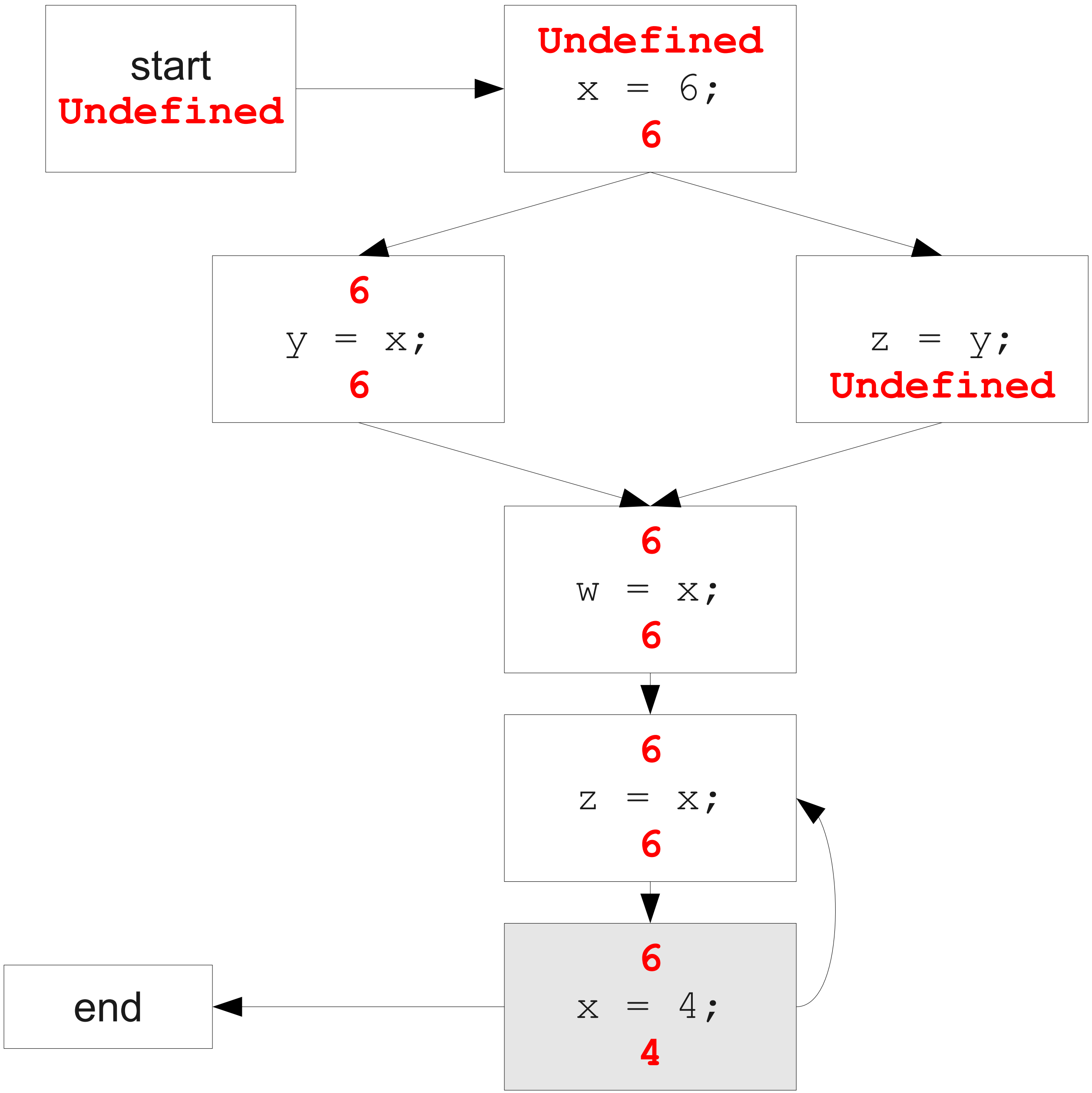
Global Constant Propagation



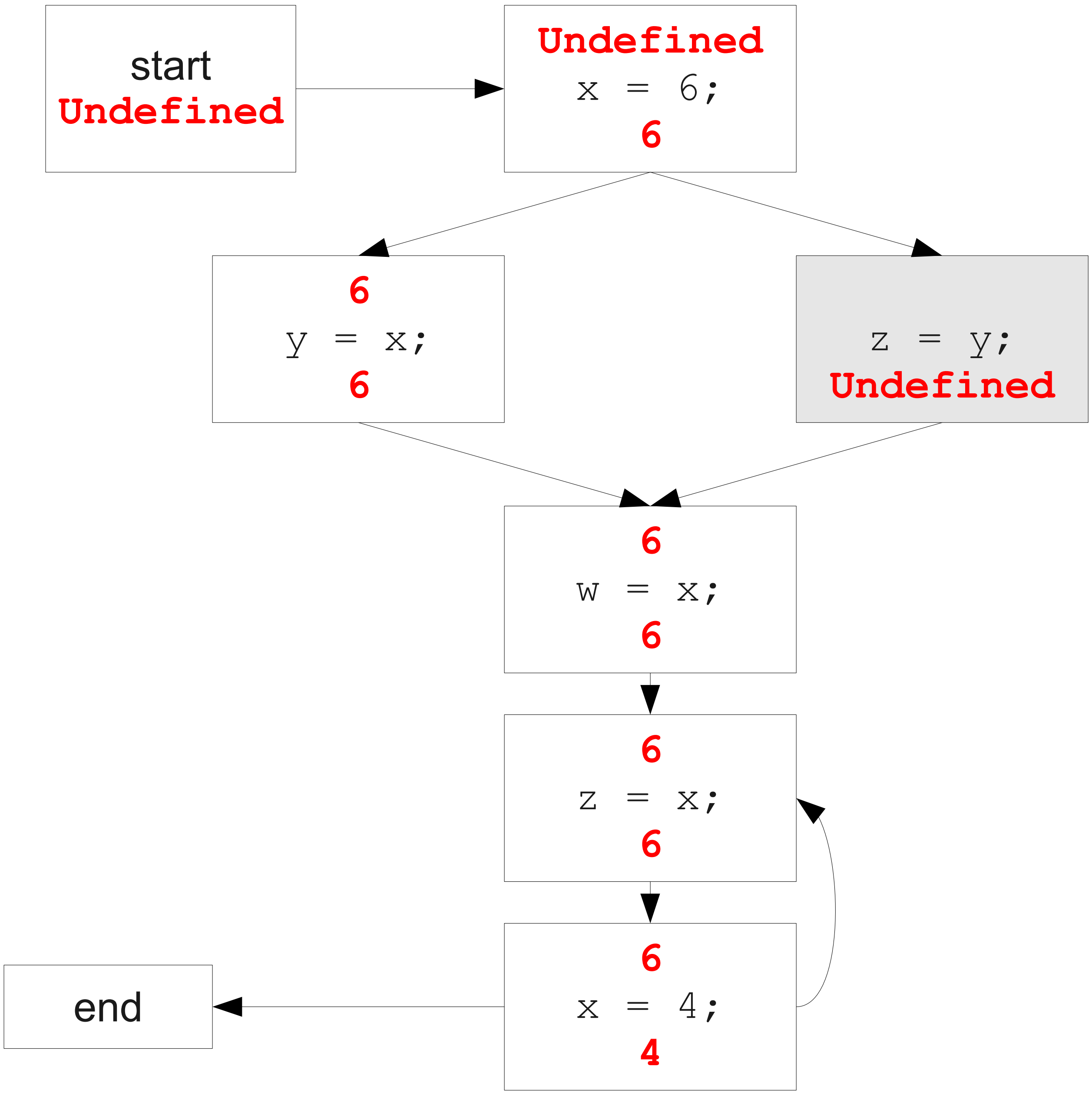
Global Constant Propagation



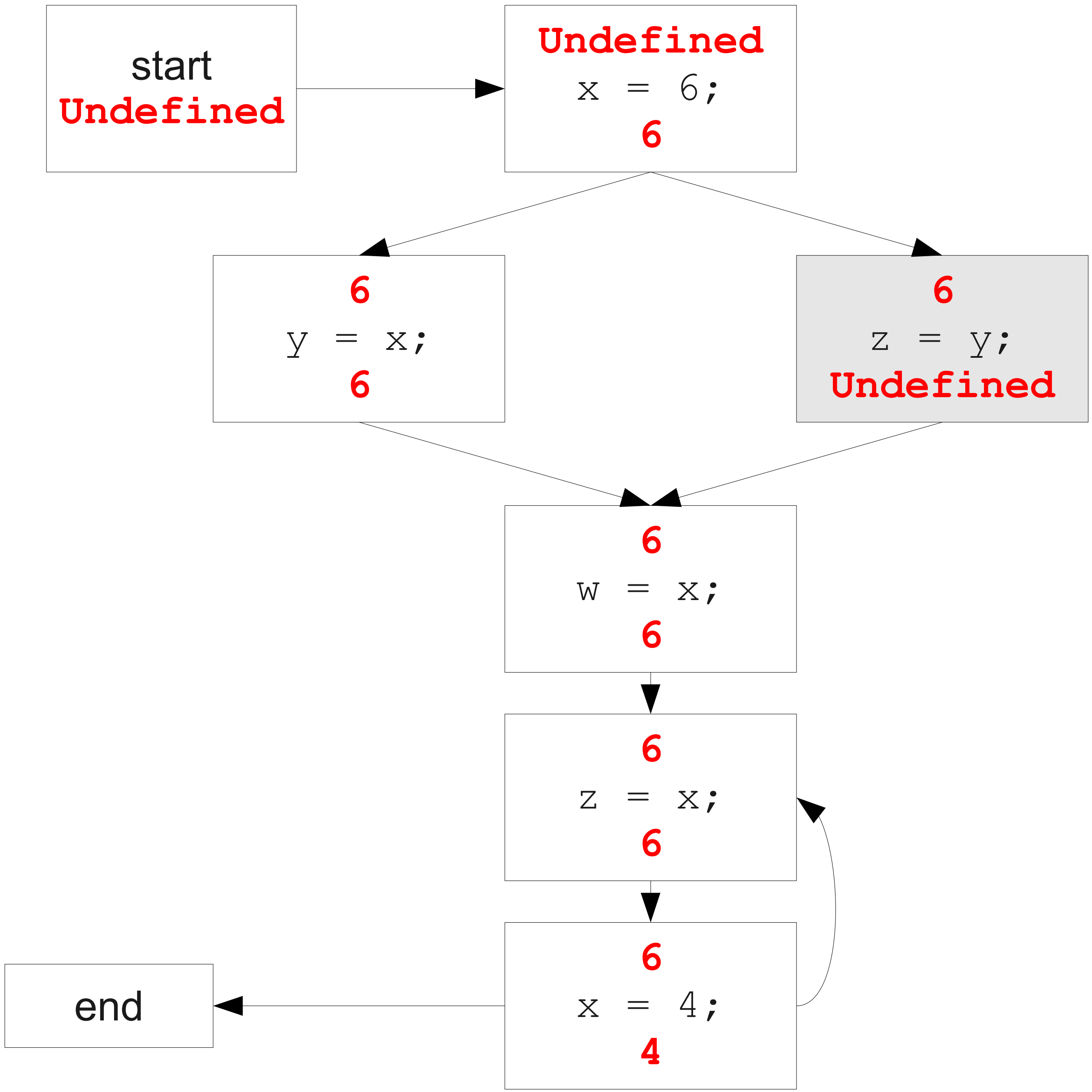
Global Constant Propagation



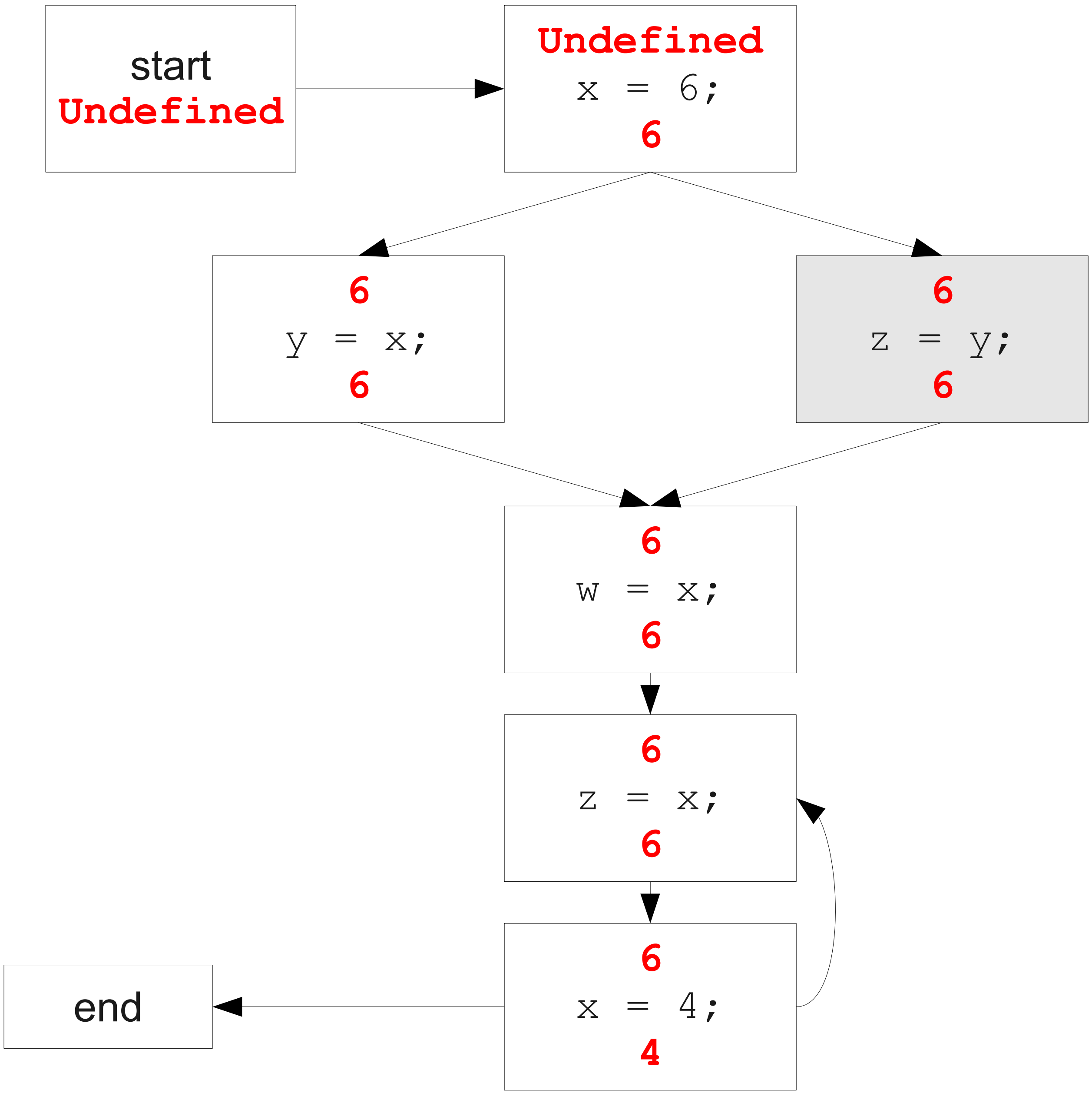
Global Constant Propagation



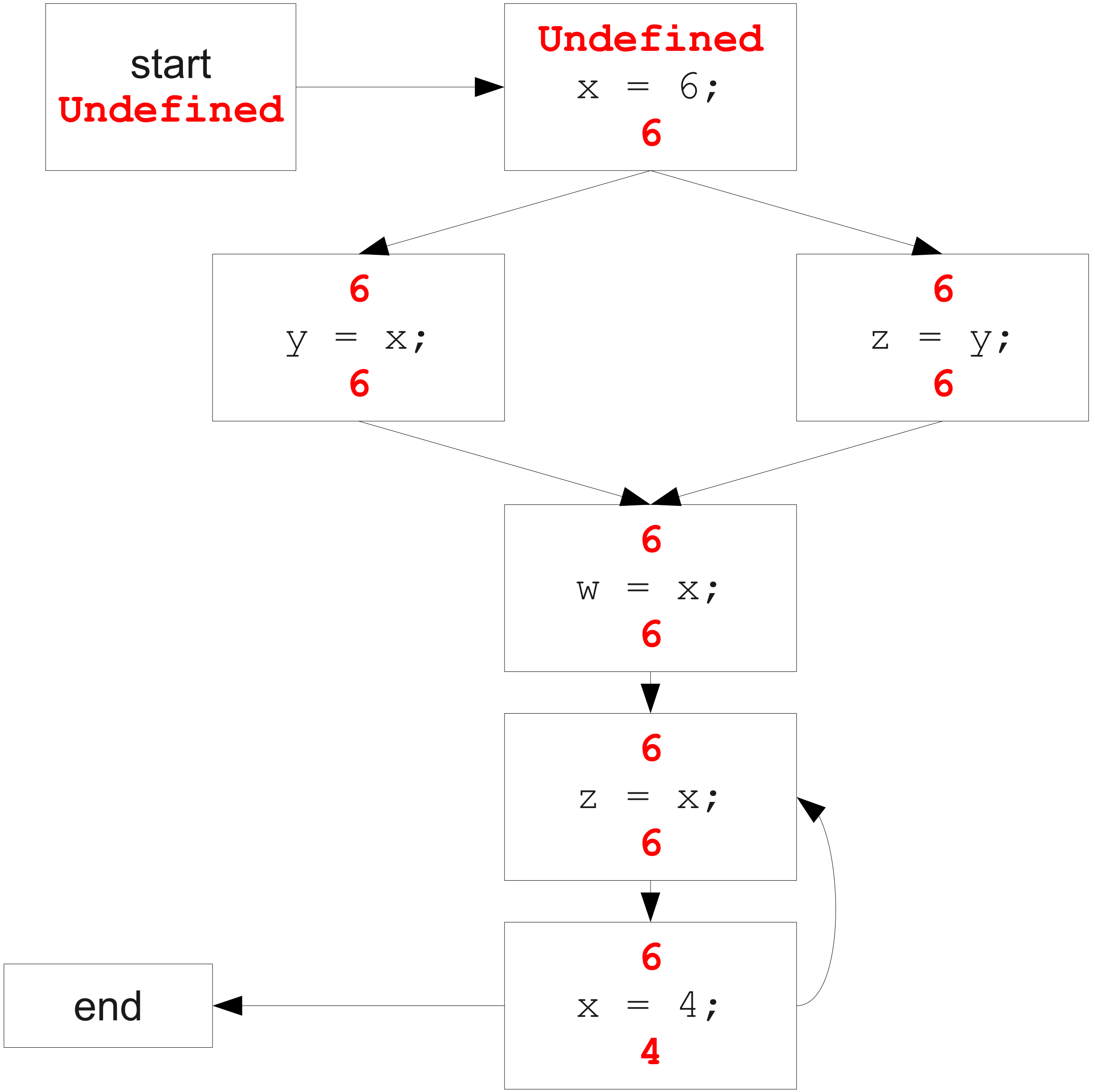
Global Constant Propagation



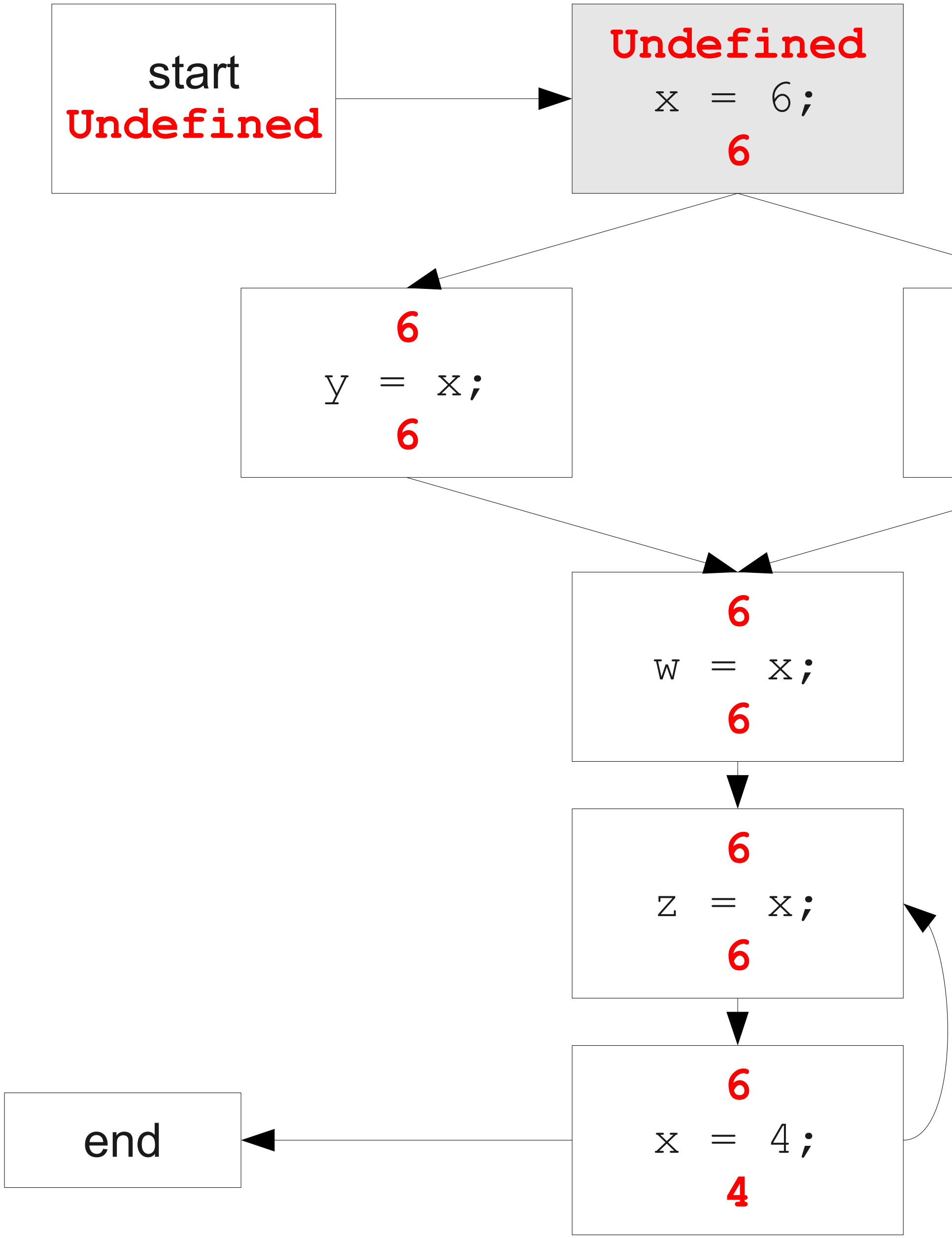
Global Constant Propagation



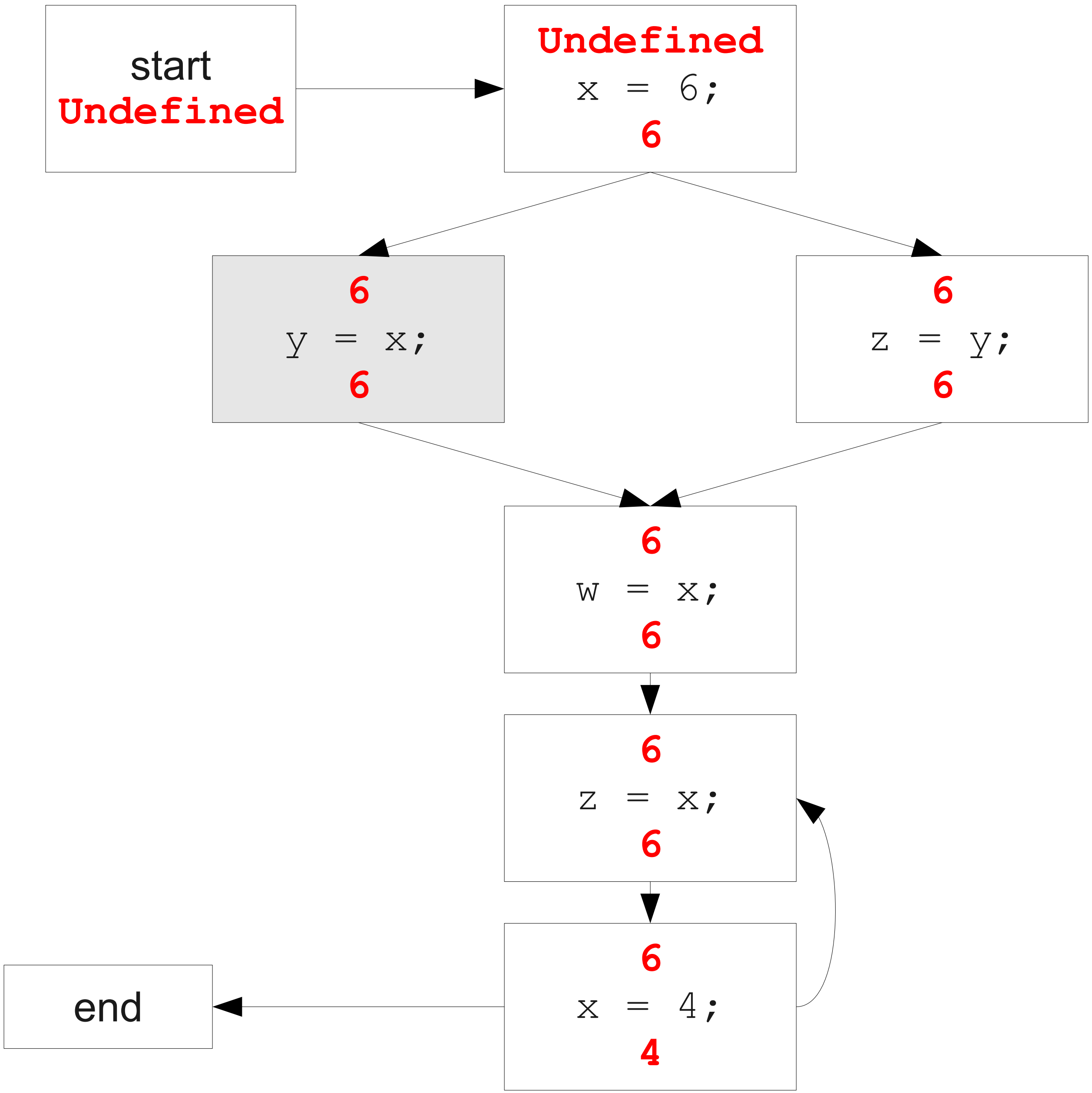
Global Constant Propagation



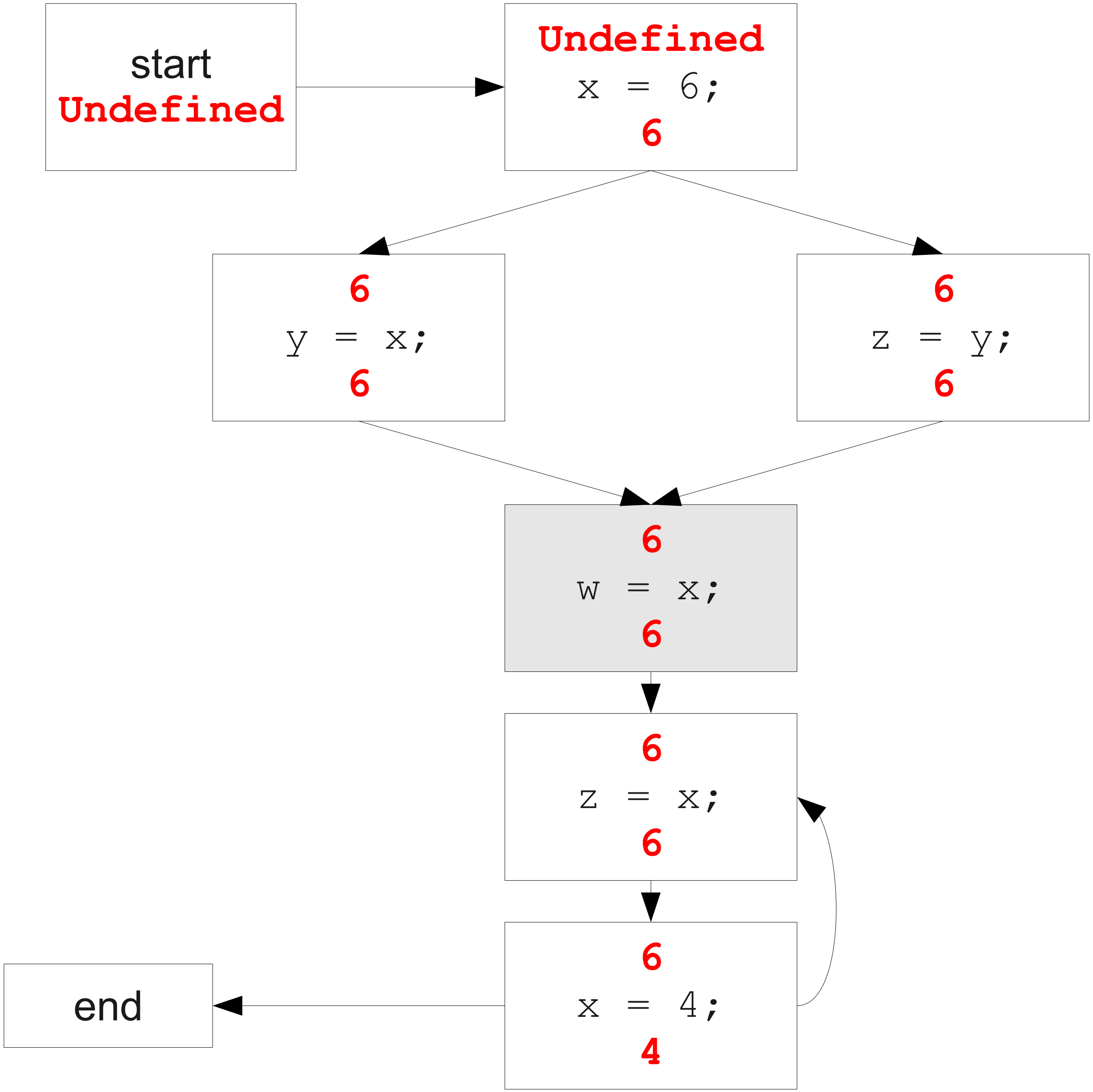
Global Constant Propagation



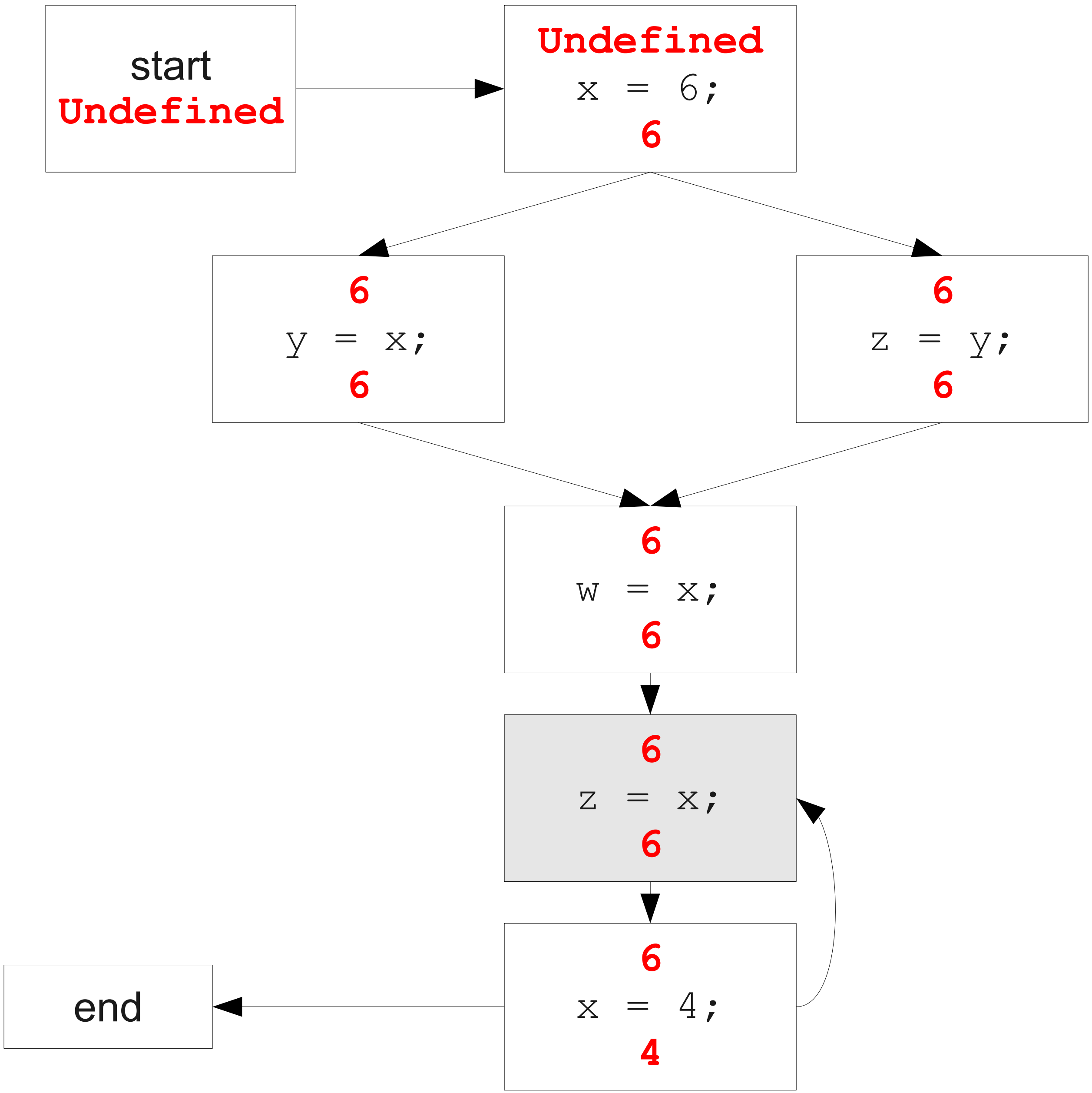
Global Constant Propagation



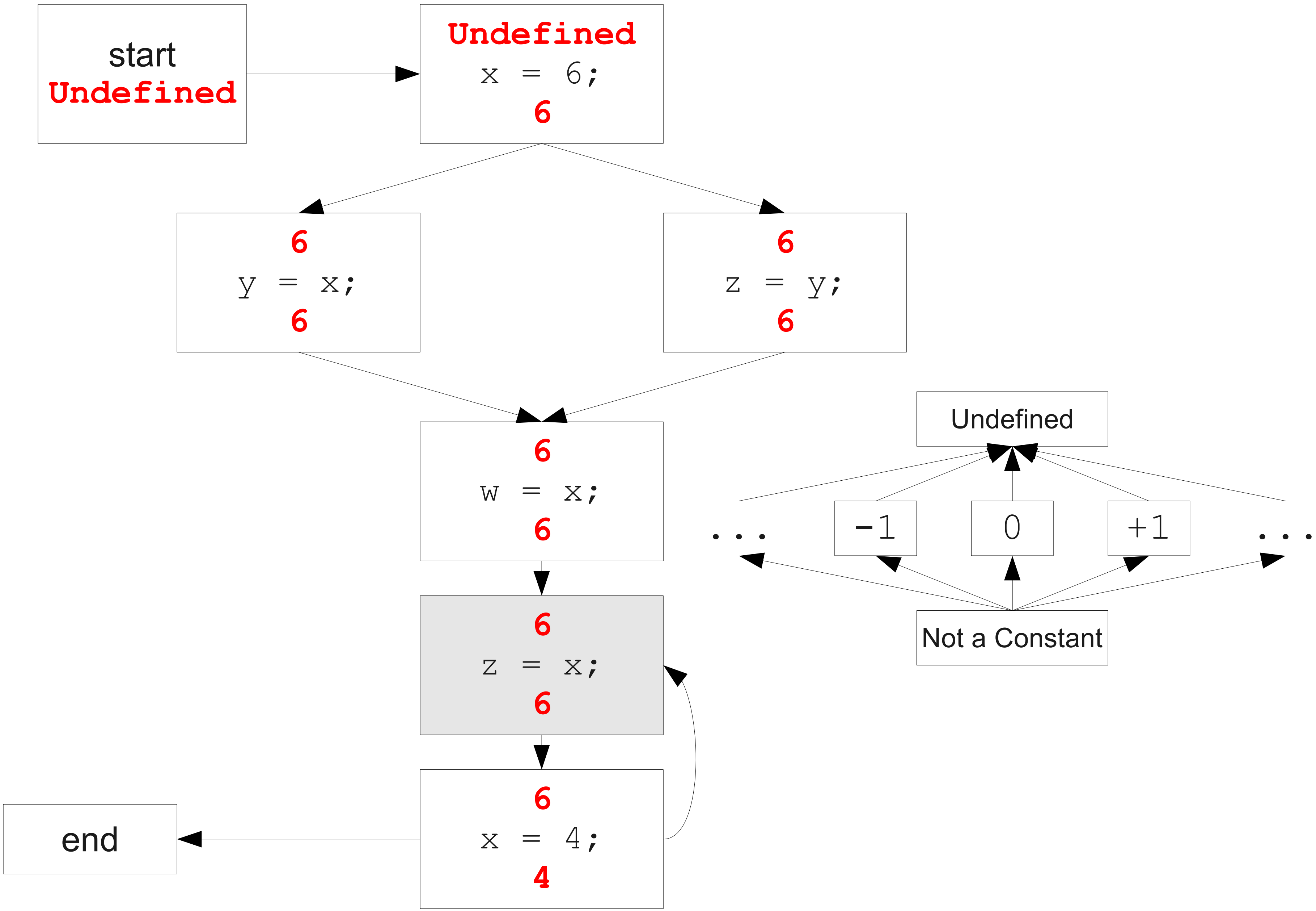
Global Constant Propagation



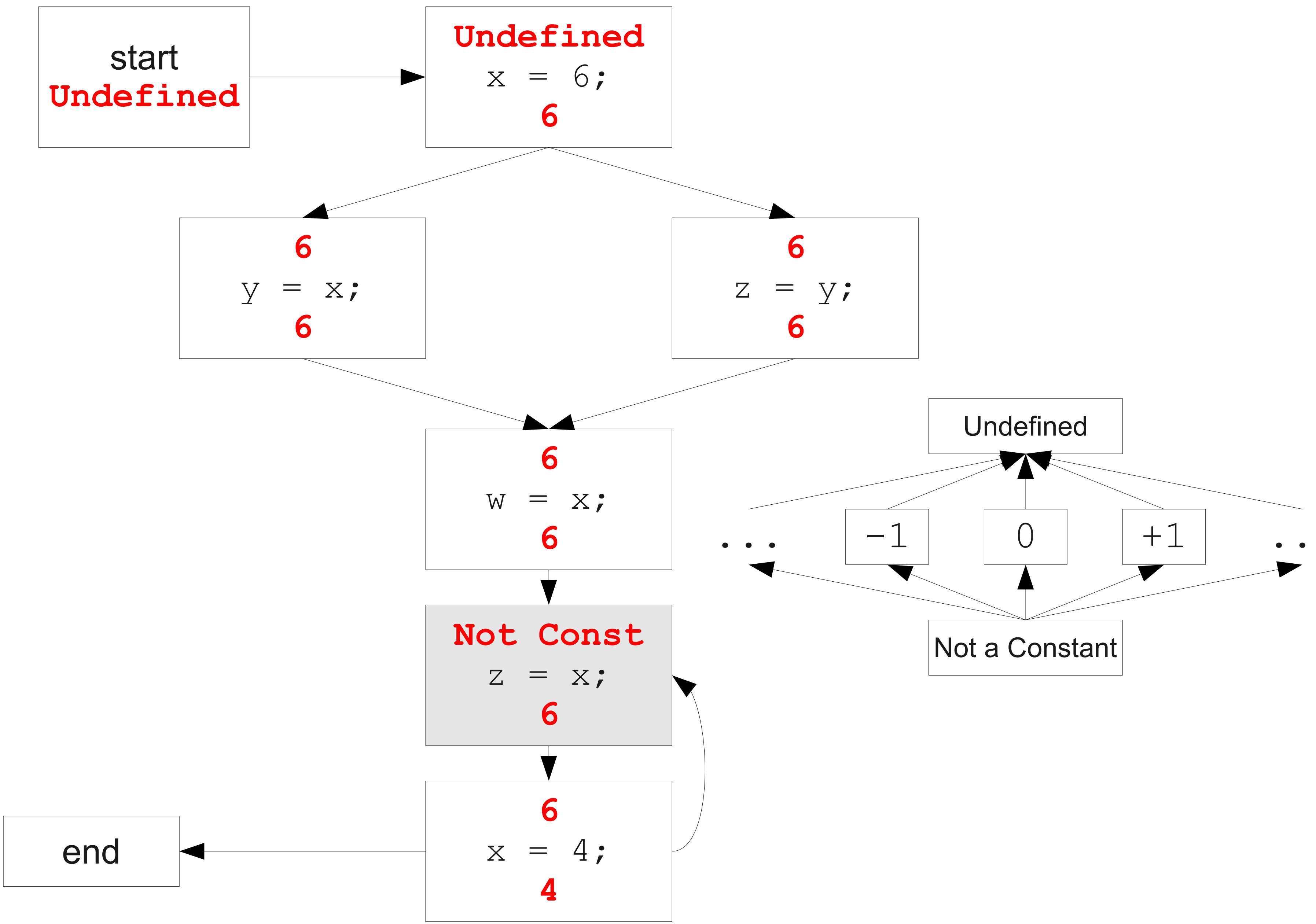
Global Constant Propagation



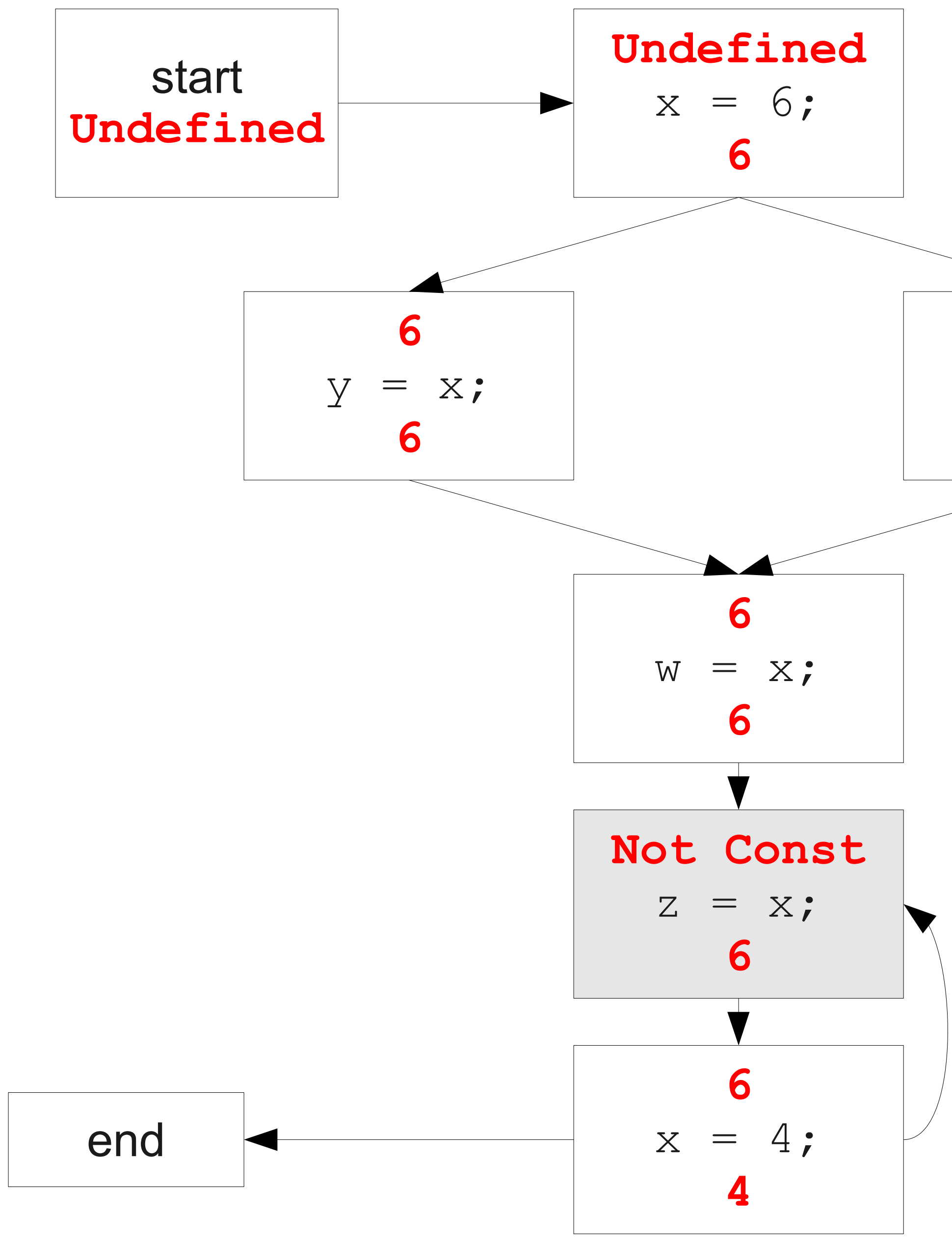
Global Constant Propagation



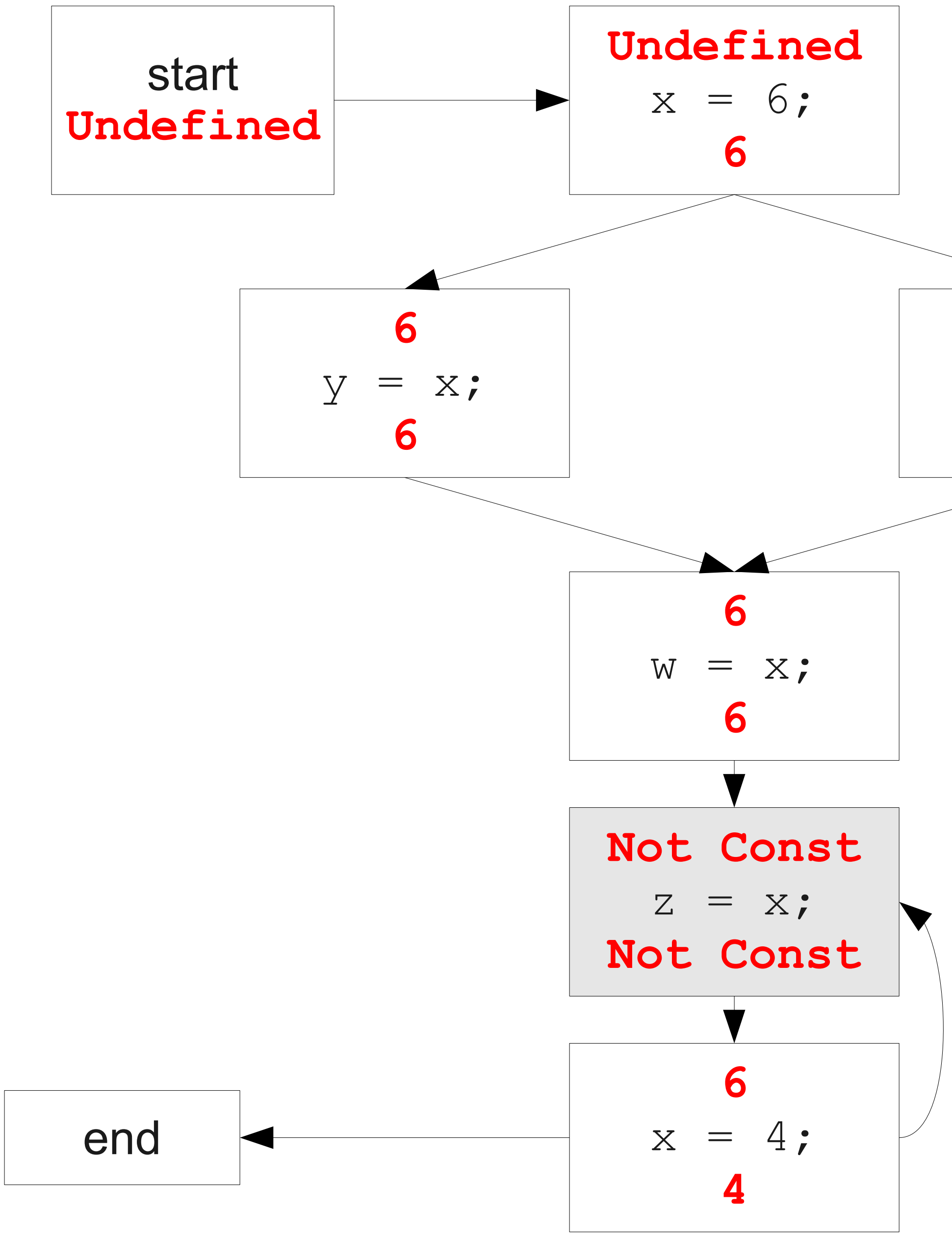
Global Constant Propagation



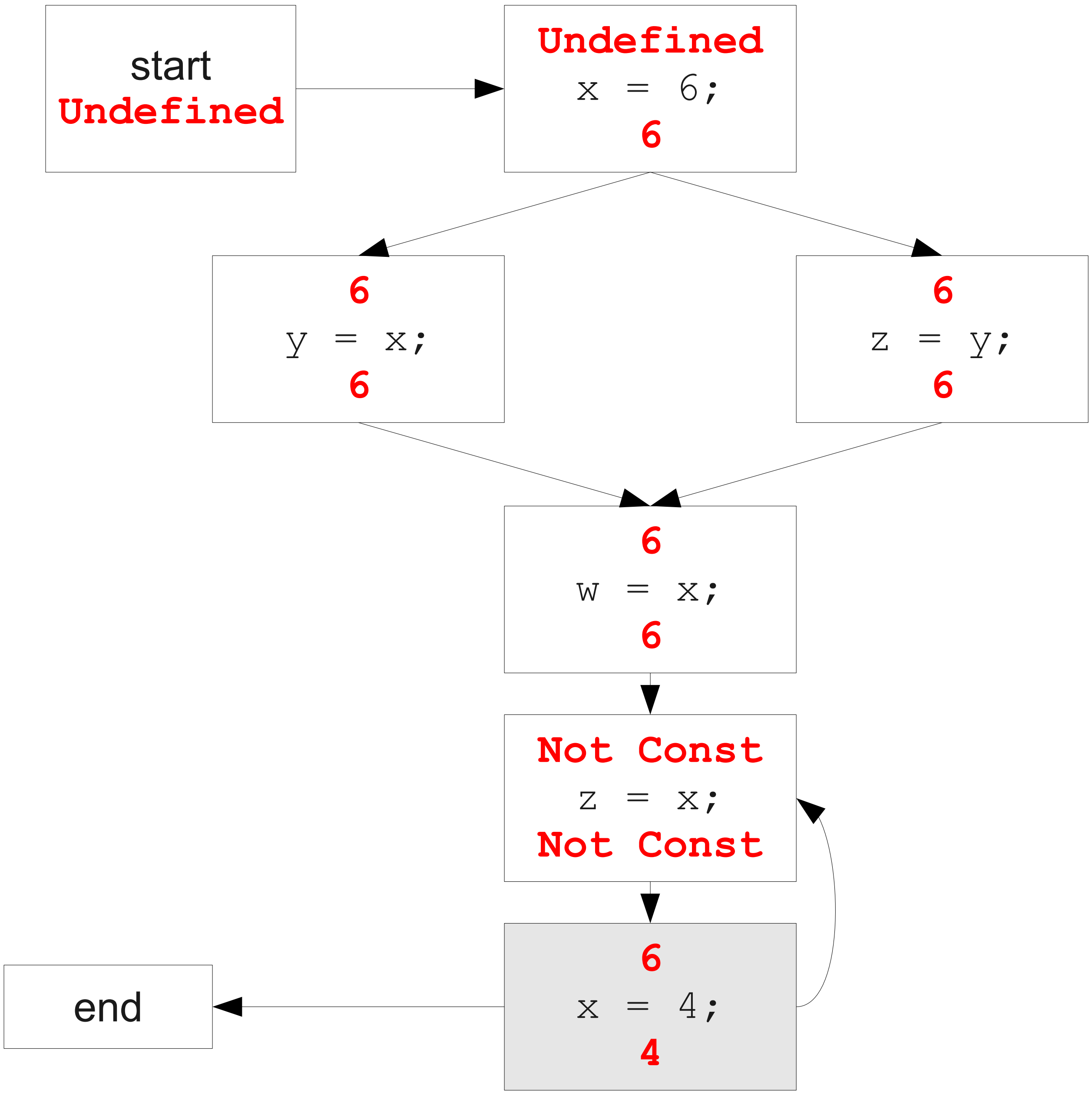
Global Constant Propagation



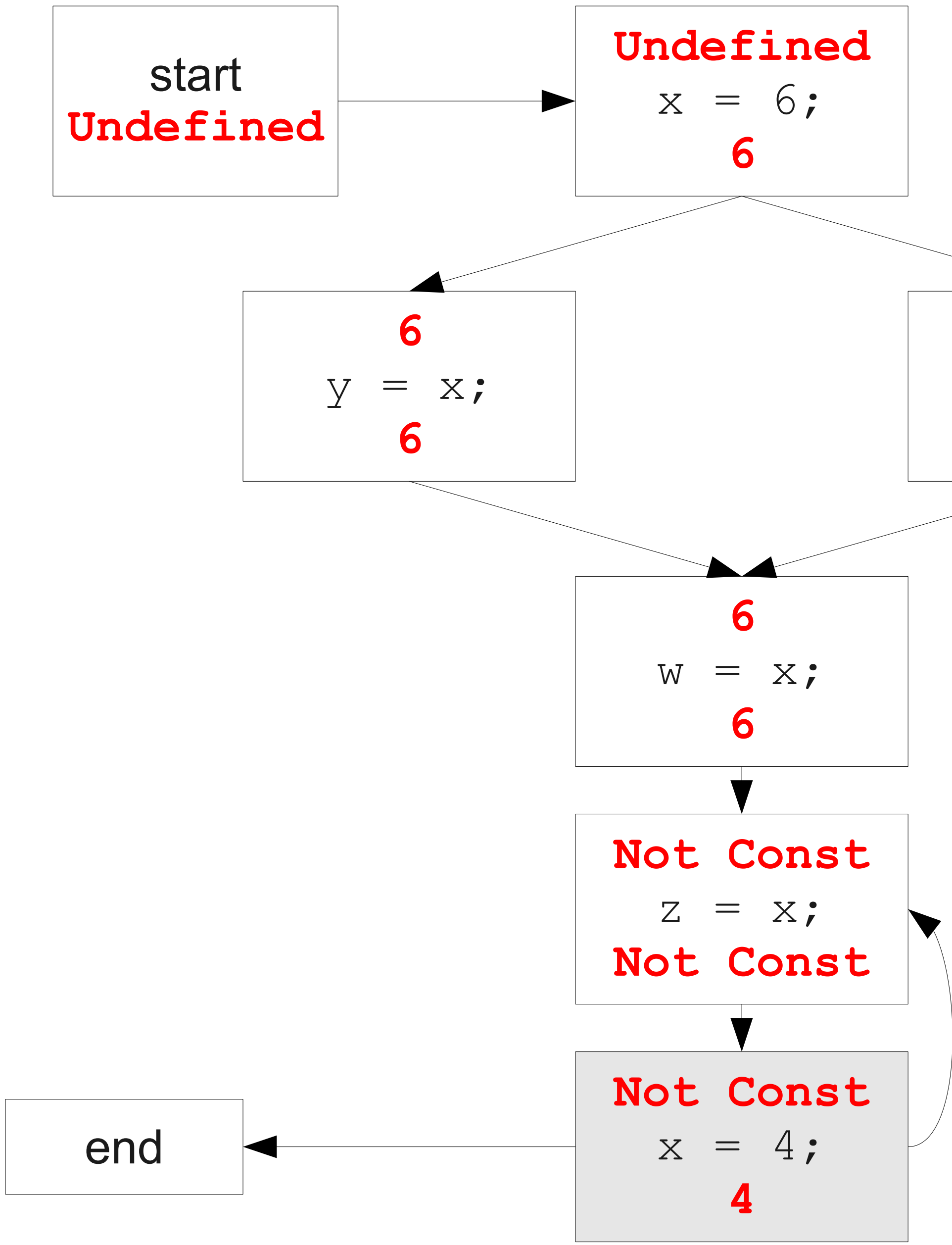
Global Constant Propagation



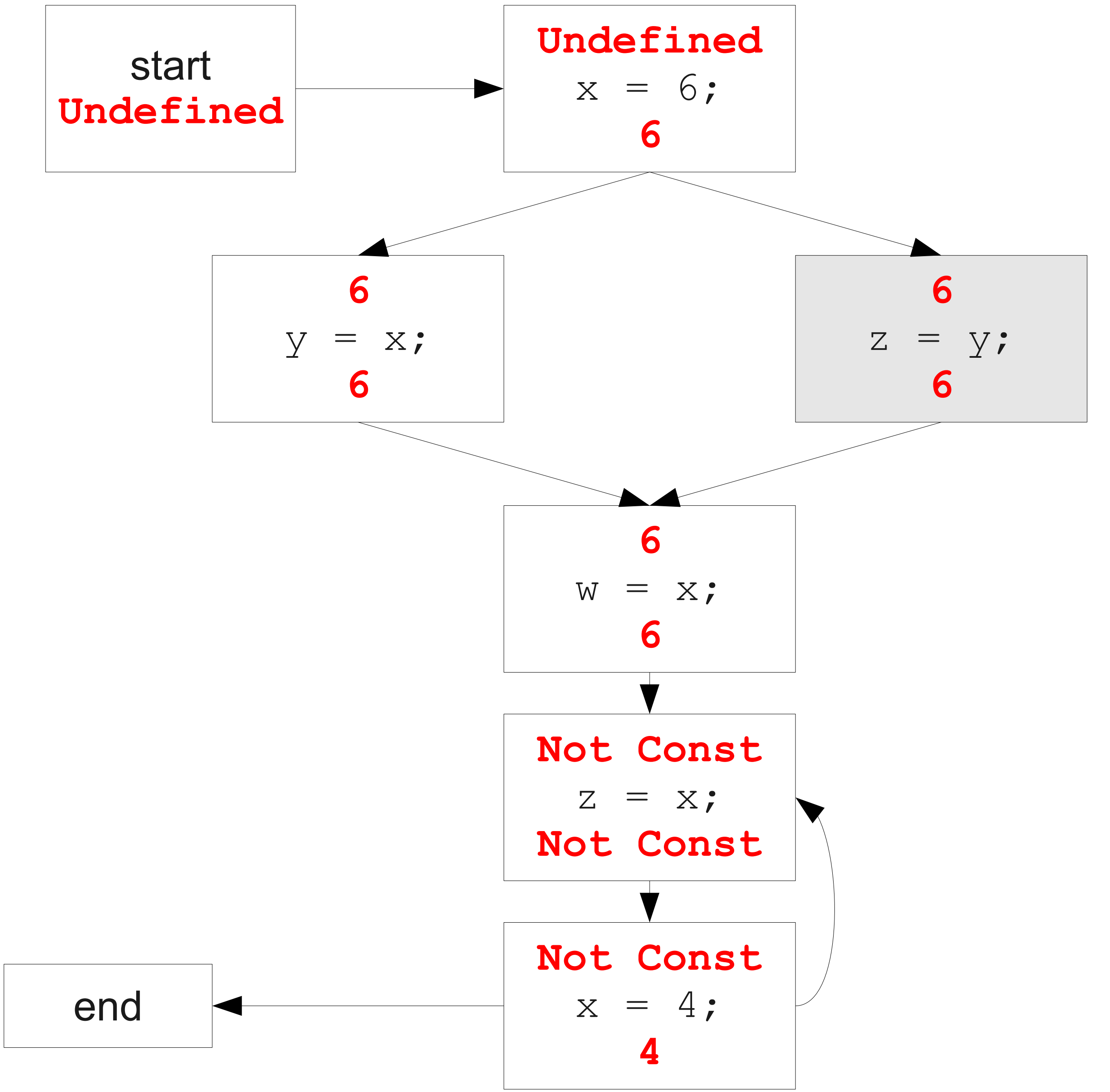
Global Constant Propagation



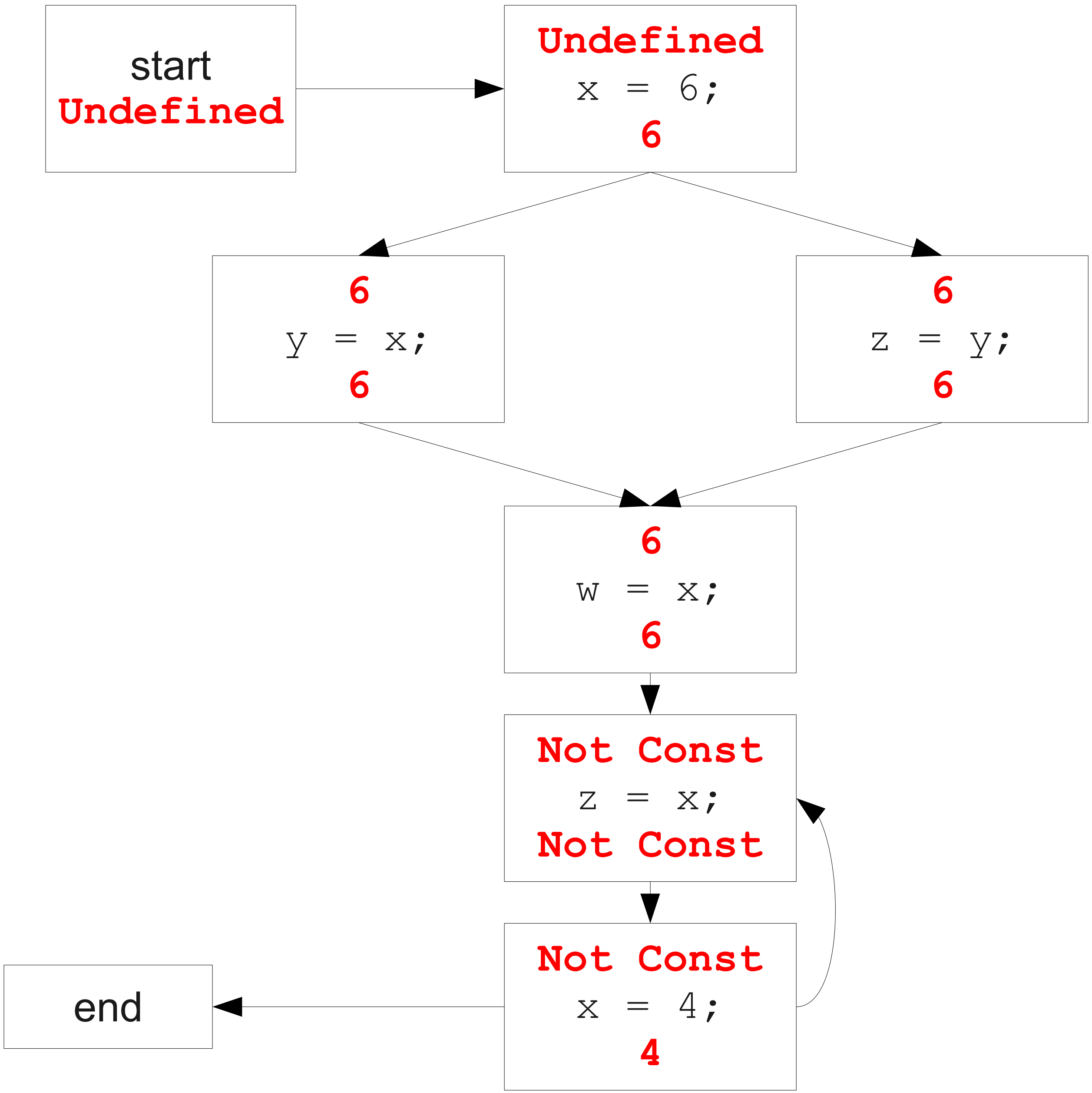
Global Constant Propagation



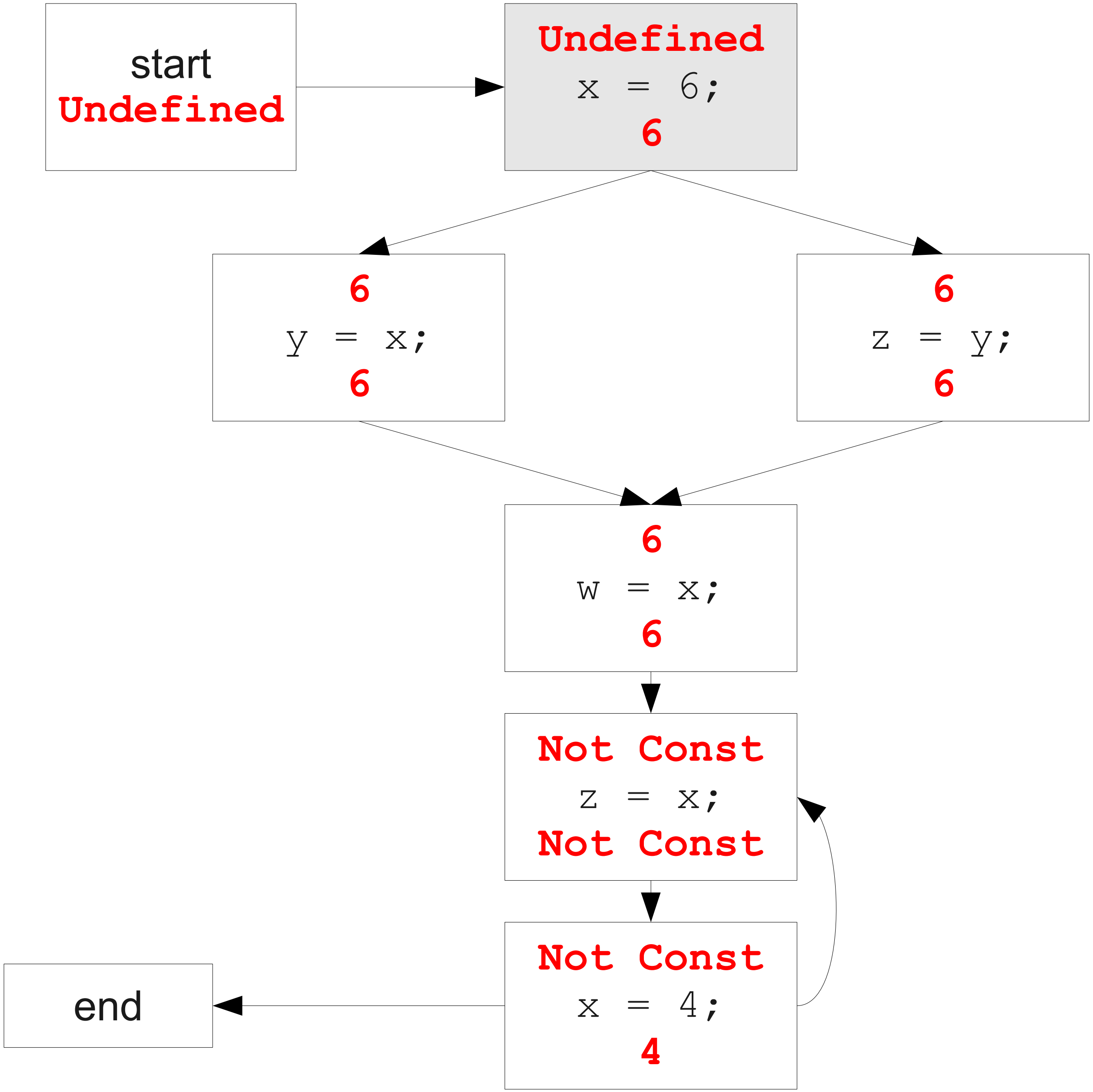
Global Constant Propagation



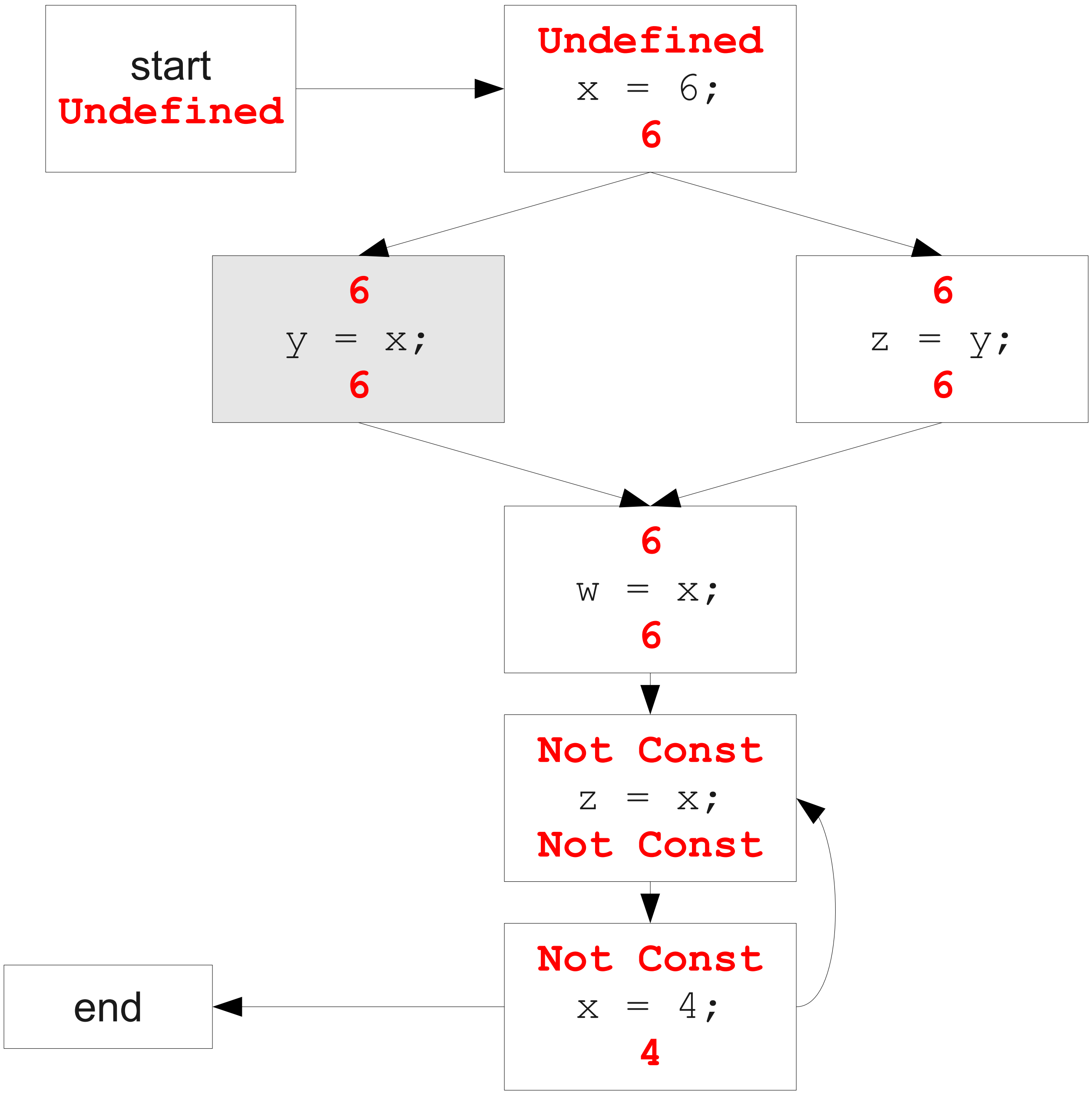
Global Constant Propagation



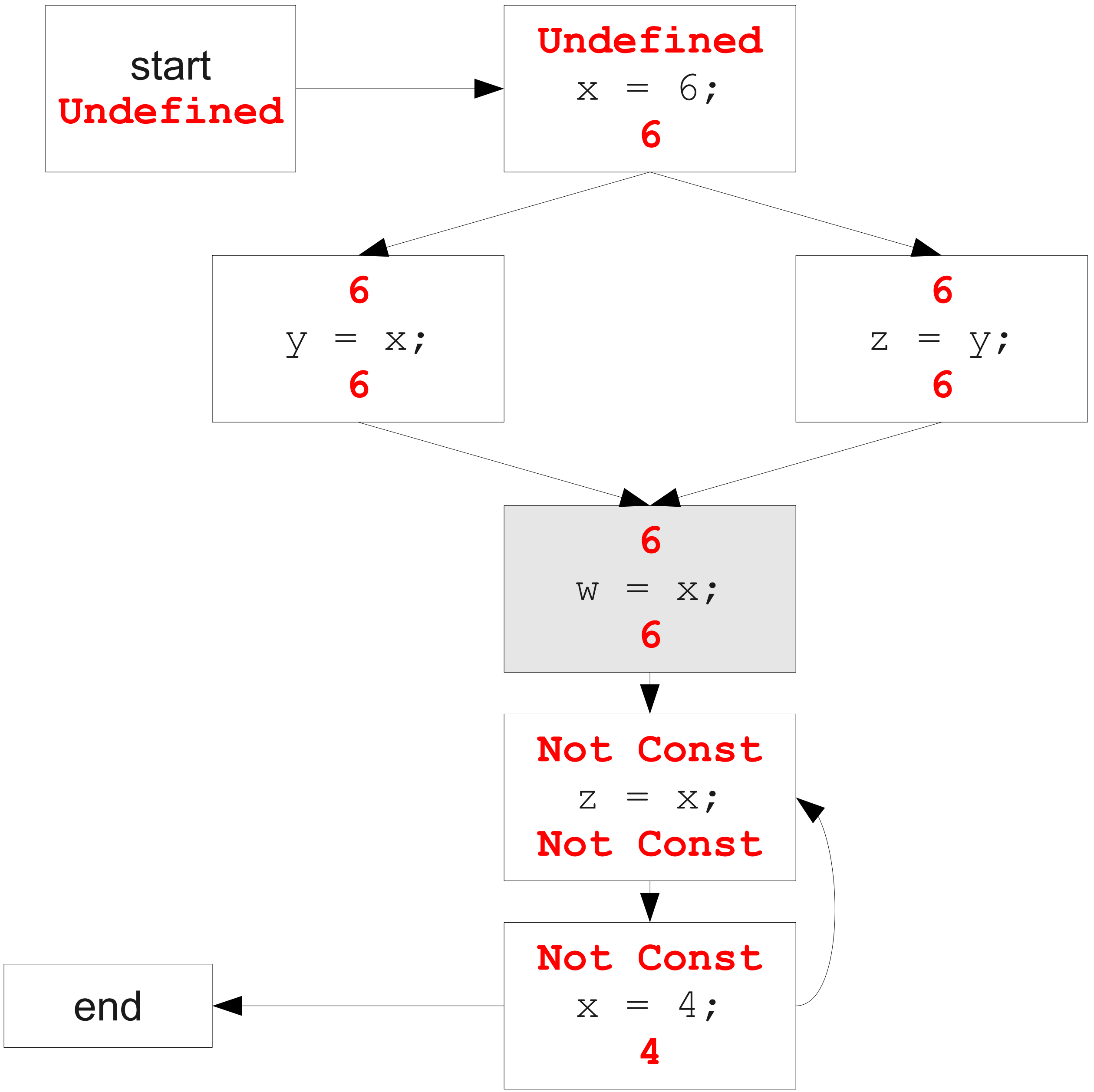
Global Constant Propagation



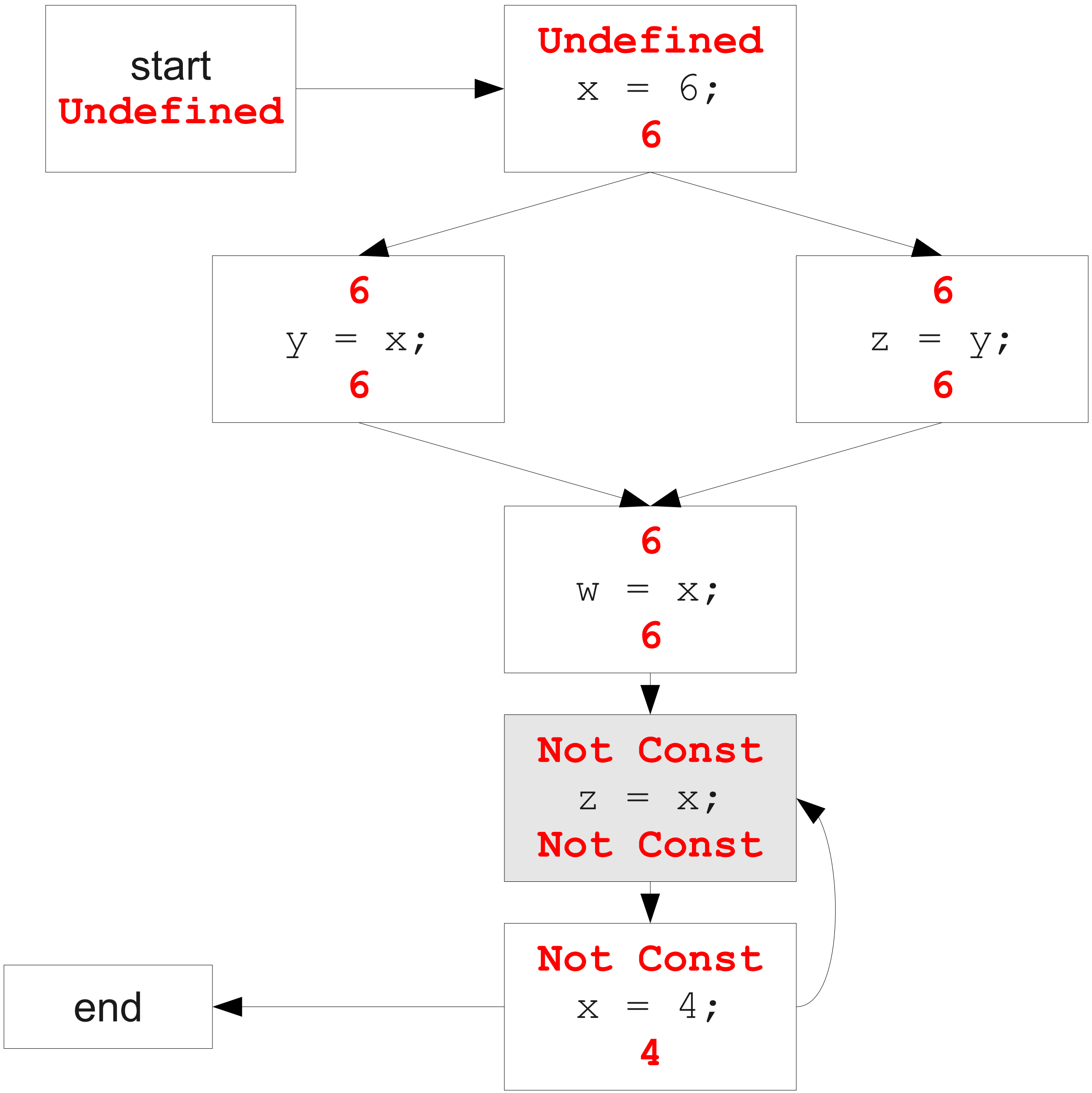
Global Constant Propagation



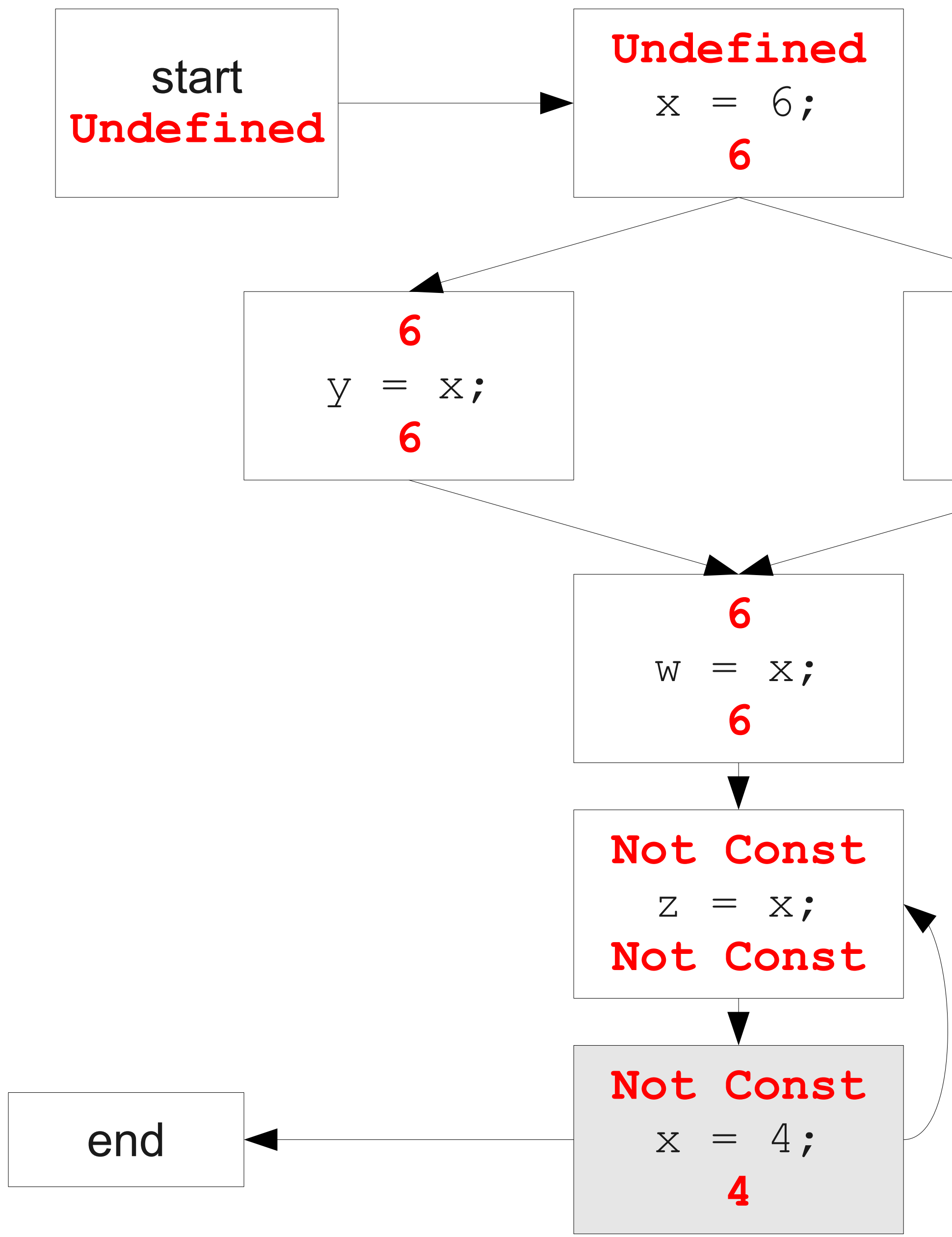
Global Constant Propagation



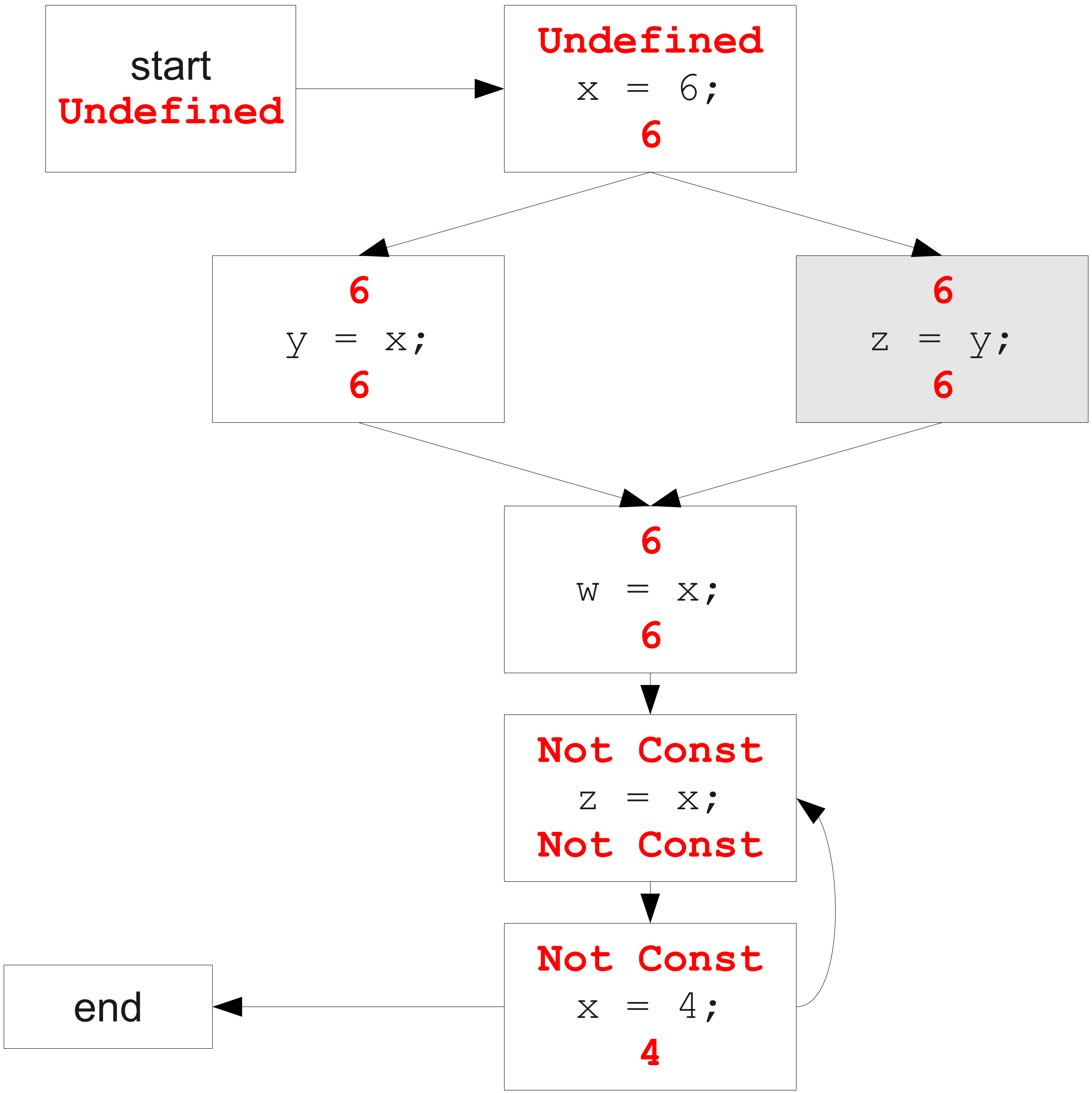
Global Constant Propagation



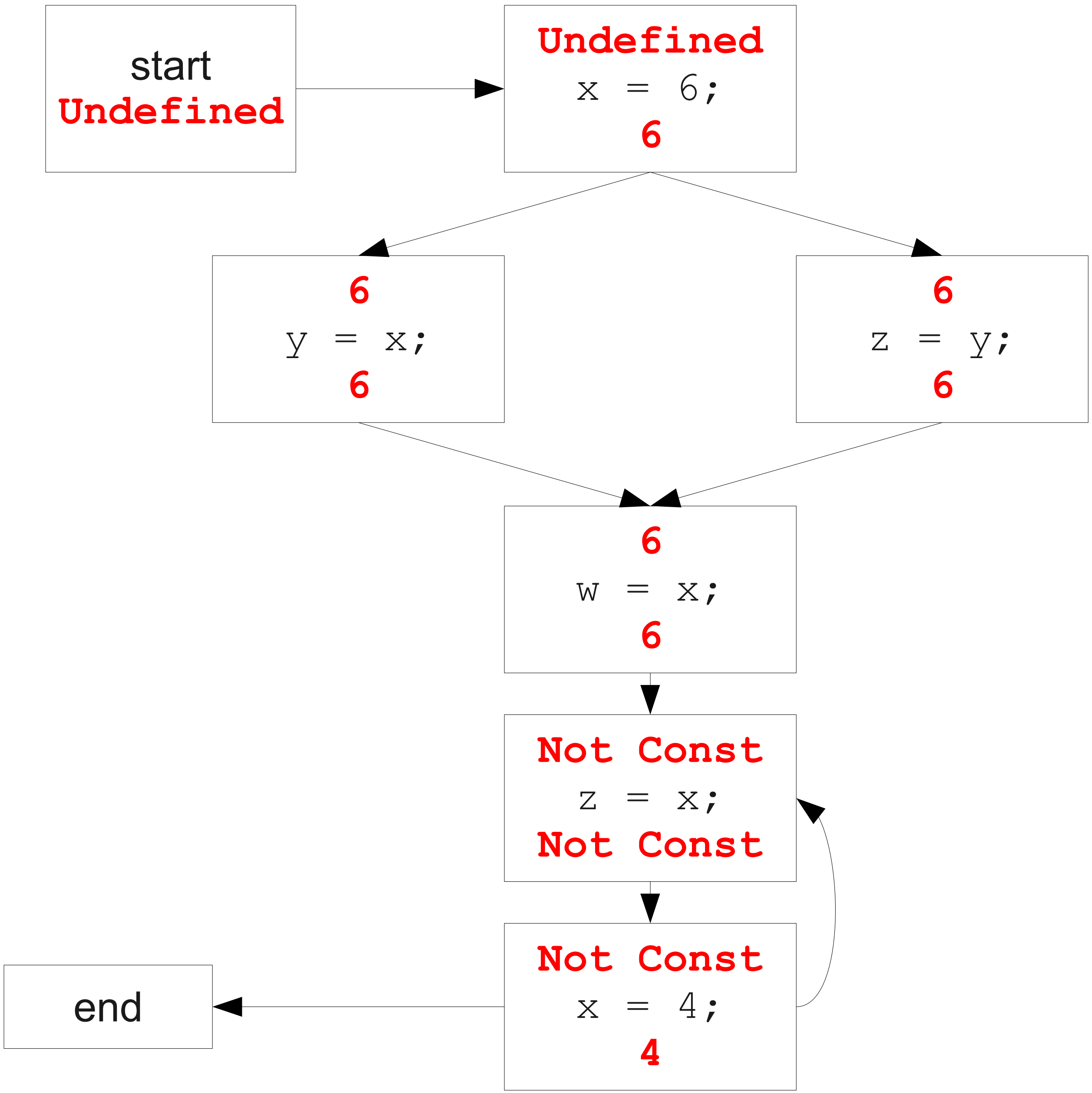
Global Constant Propagation



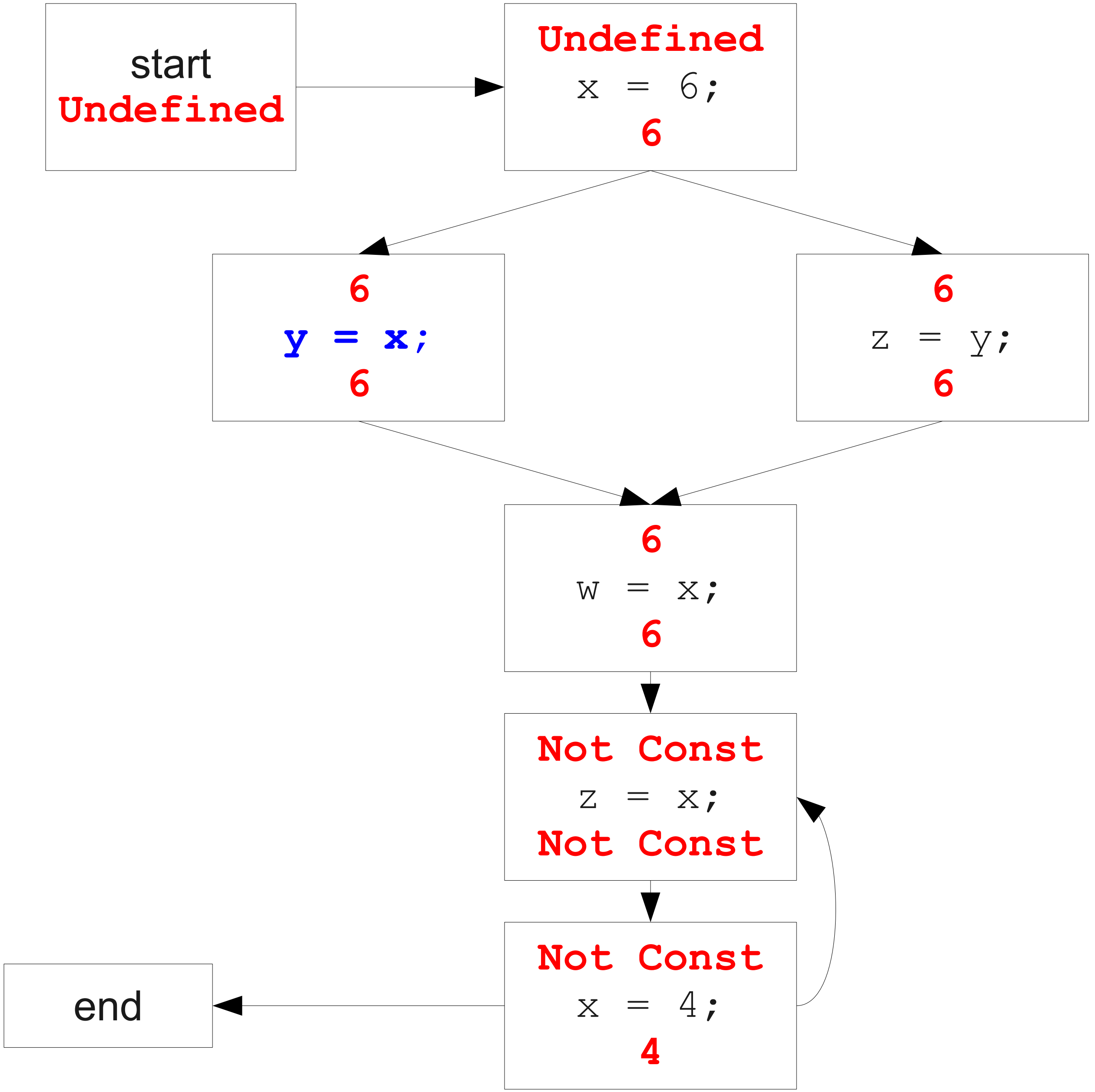
Global Constant Propagation



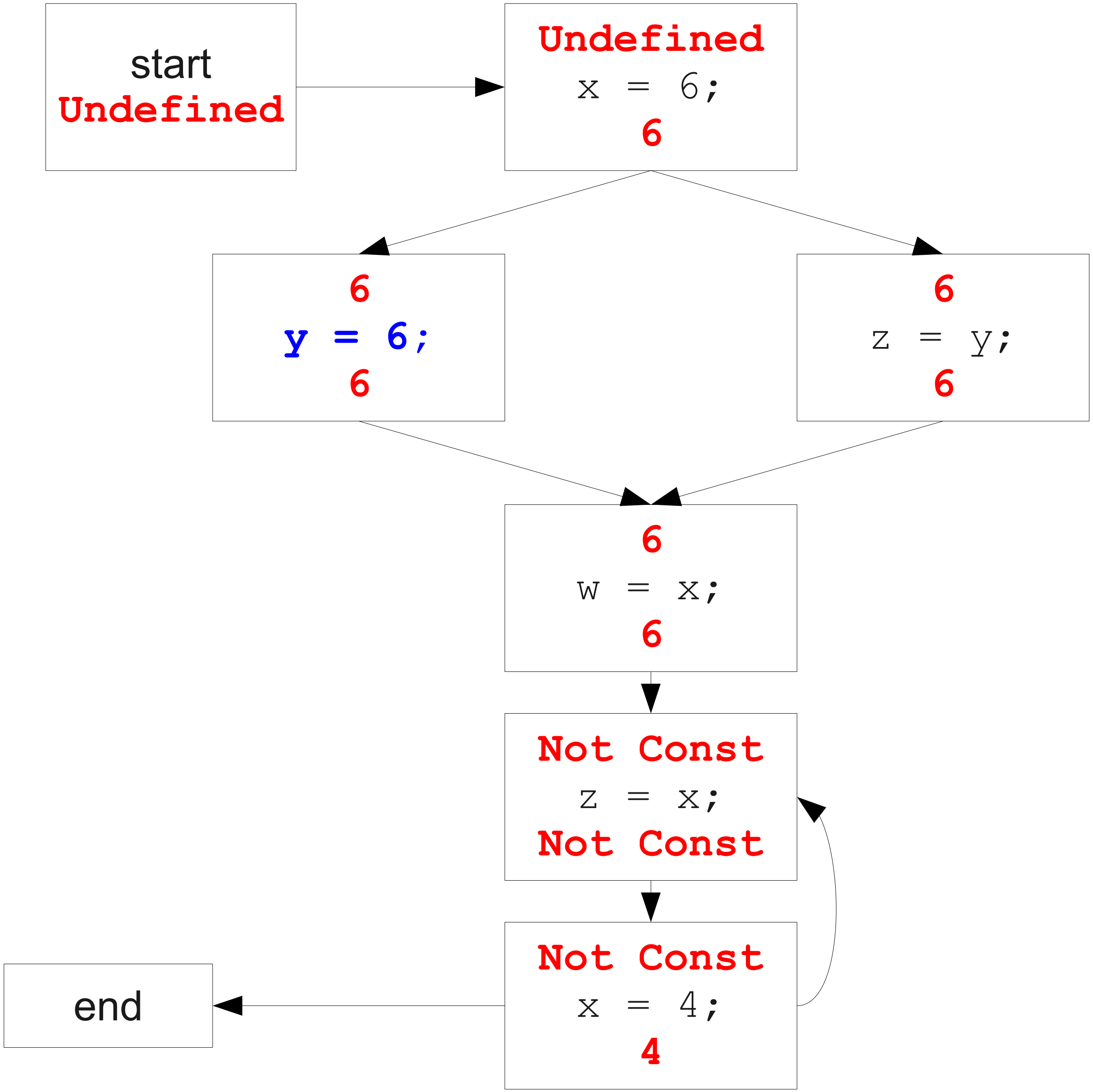
Global Constant Propagation



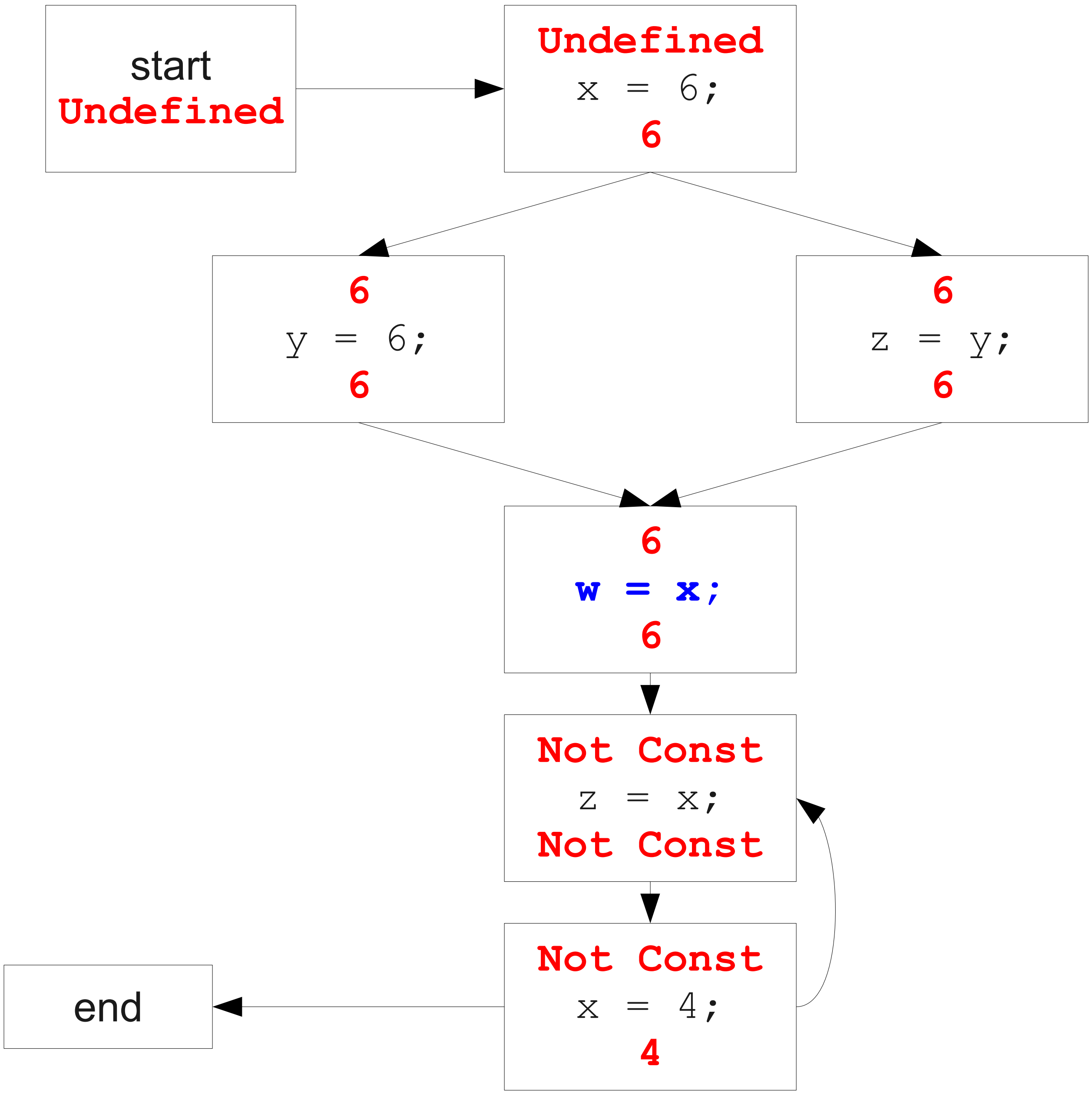
Global Constant Propagation



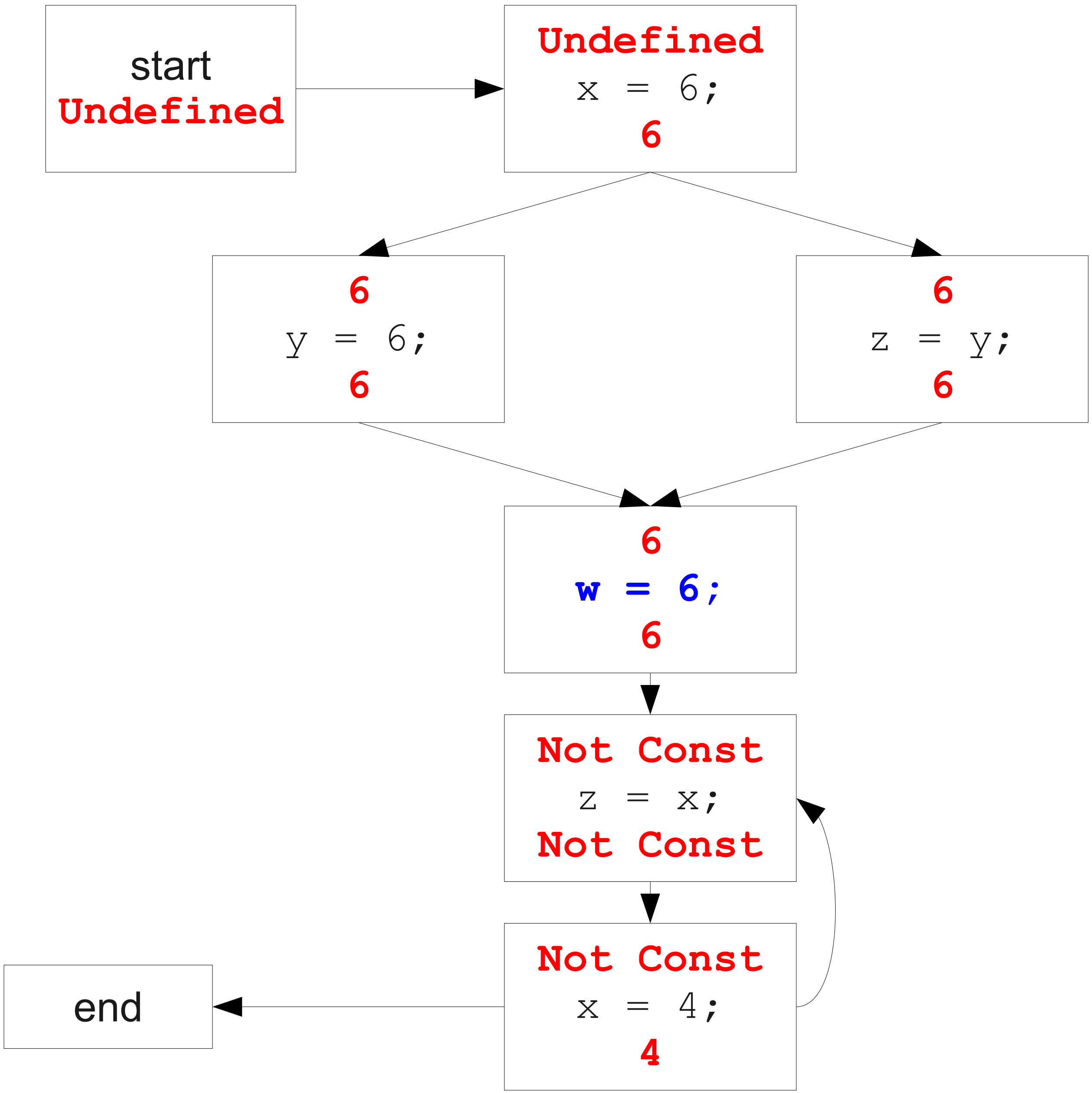
Global Constant Propagation



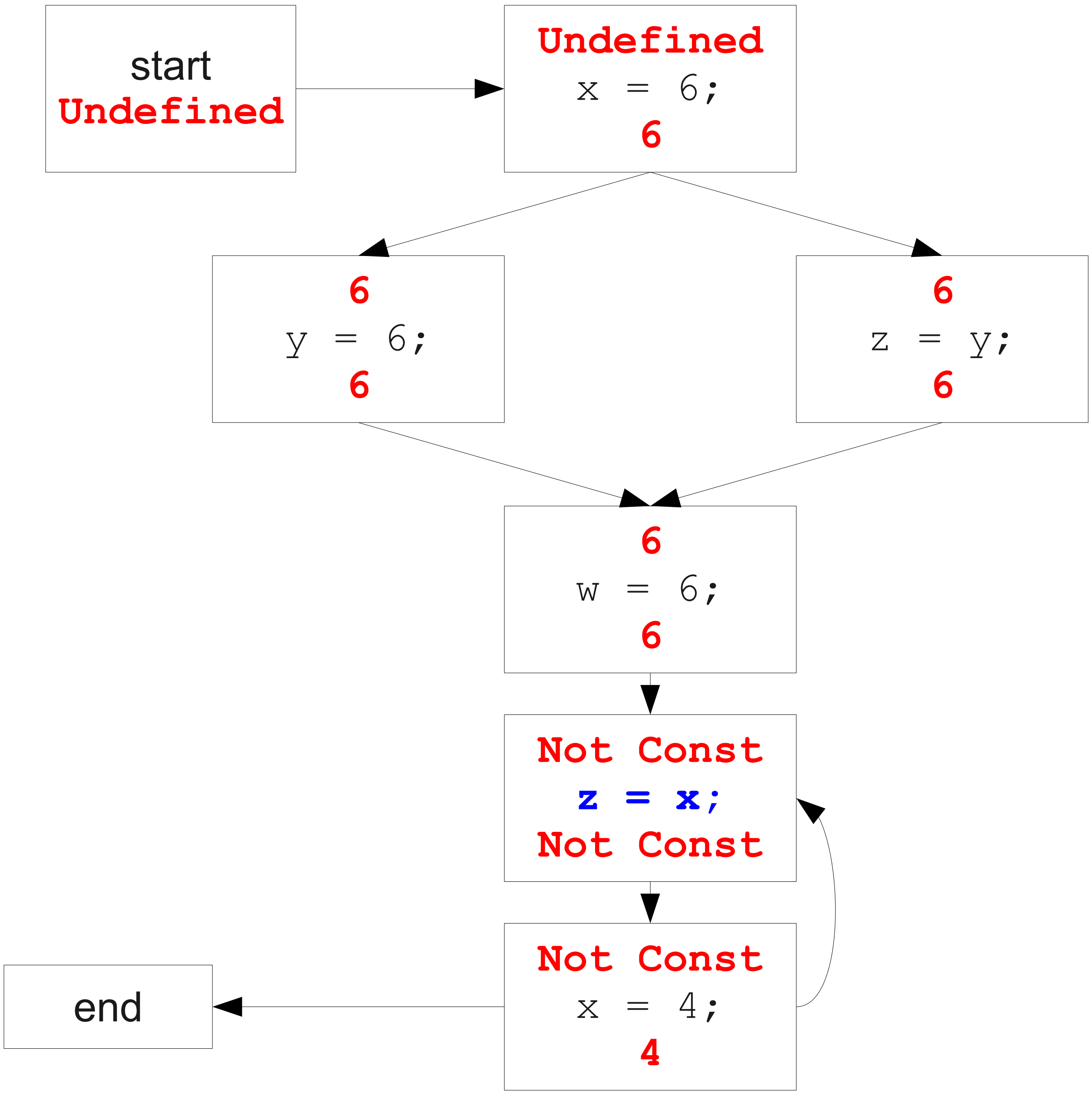
Global Constant Propagation



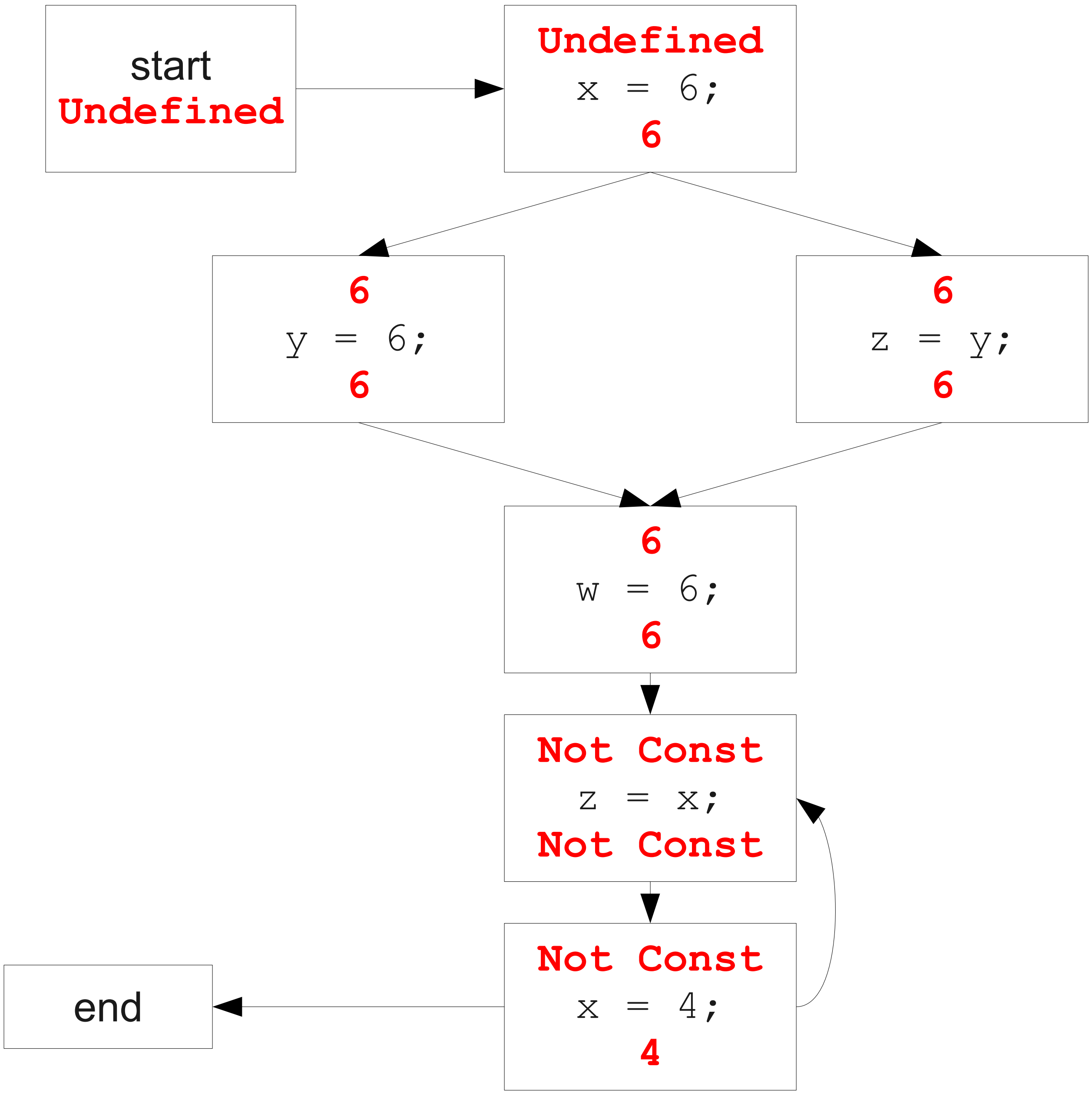
Global Constant Propagation



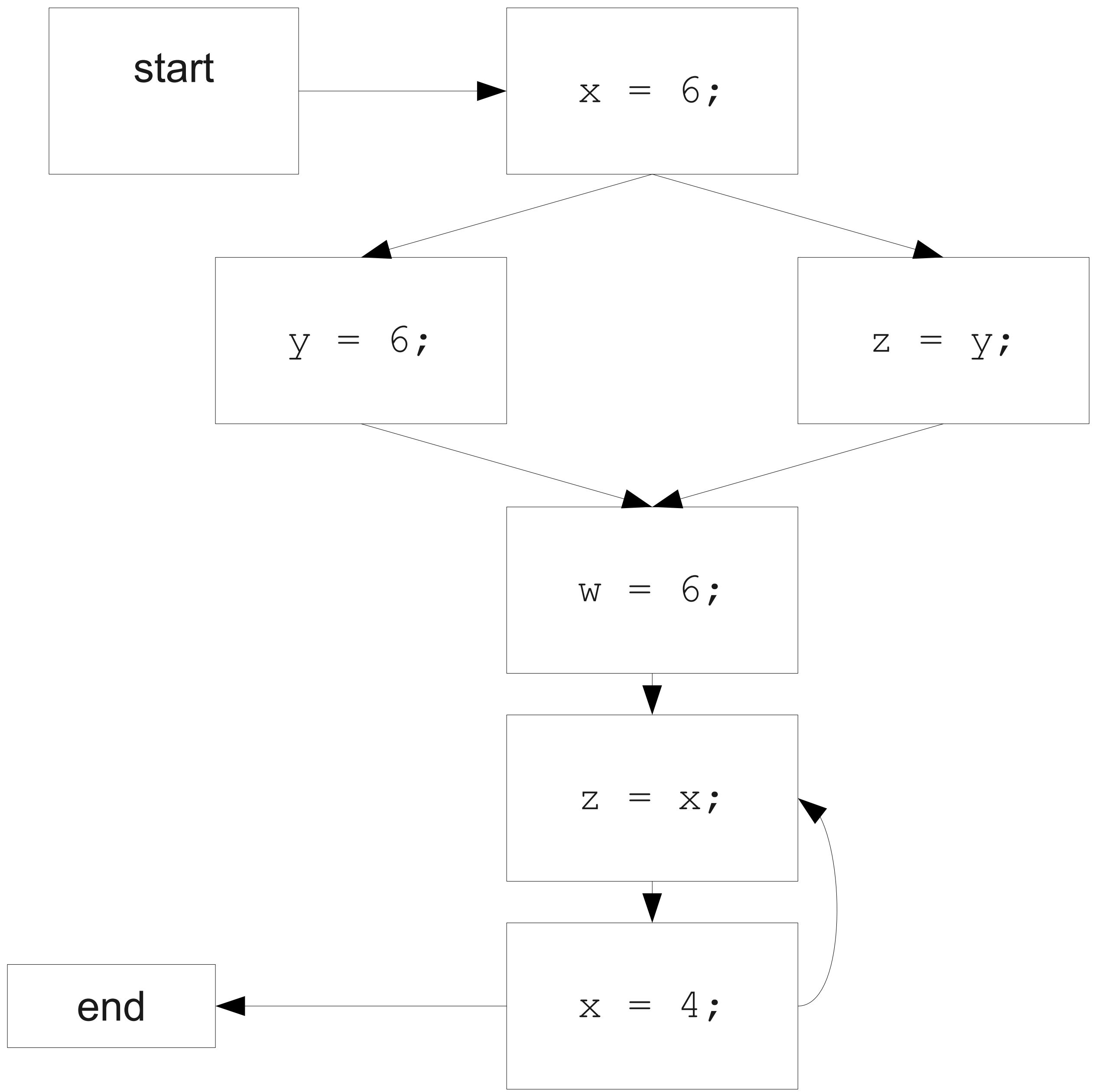
Global Constant Propagation



Global Constant Propagation



Global Constant Propagation



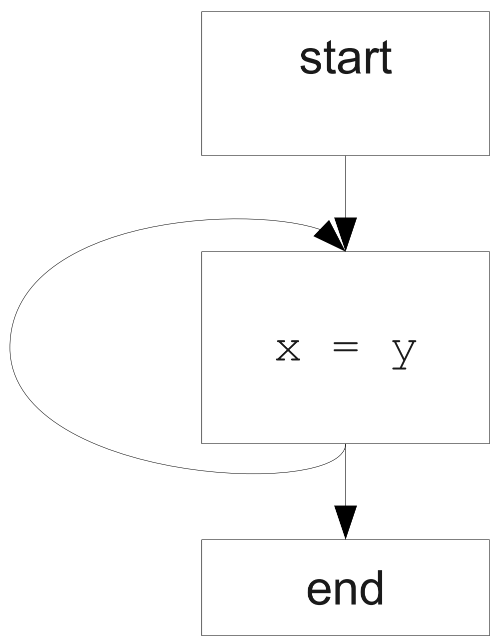
Proving Termination

- 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**.

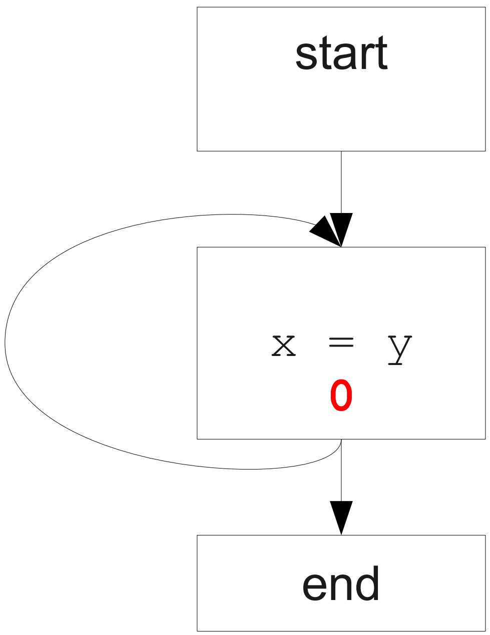
A Nonterminating Analysis

- The following analysis will loop infinitely on any CFG containing a loop:
- Direction: **Forward**
- Domain: **The natural numbers 0, 1, 2, …**
- Meet operator: **max**
- Transfer function: **f(n) = n + 1**
- Initial value: **0**

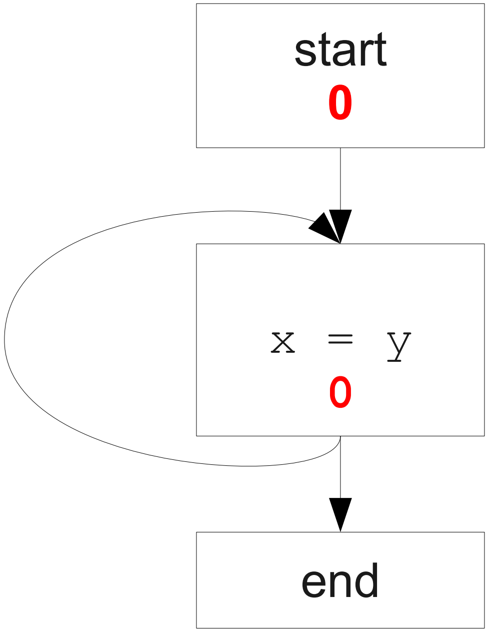
A Nonterminating Analysis



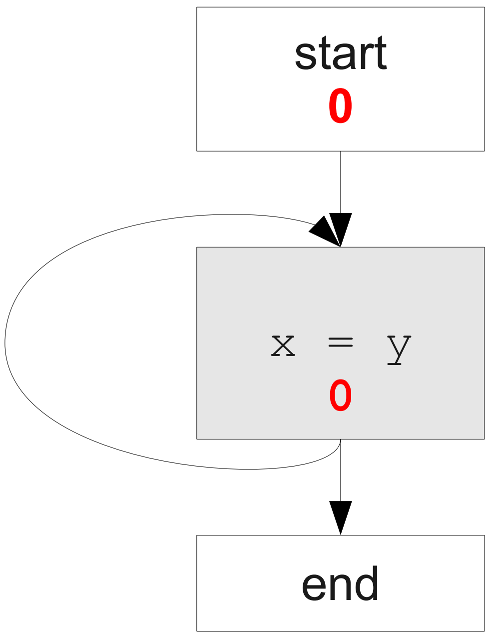
A Nonterminating Analysis



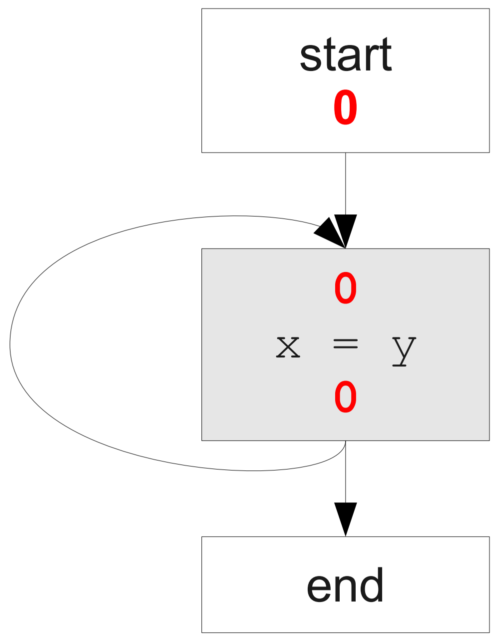
A Nonterminating Analysis



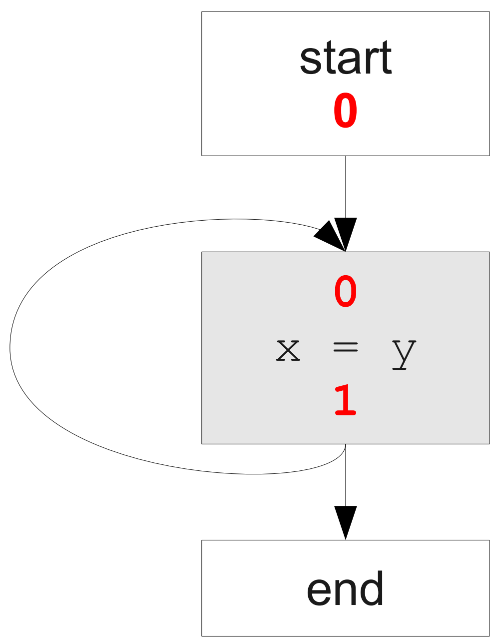
A Nonterminating Analysis



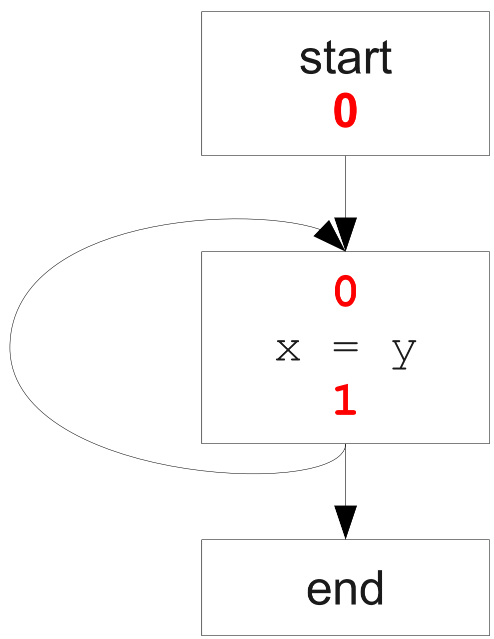
A Nonterminating Analysis



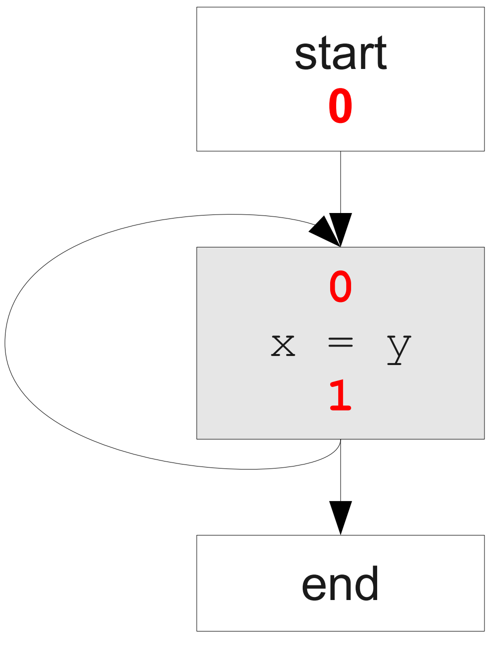
A Nonterminating Analysis



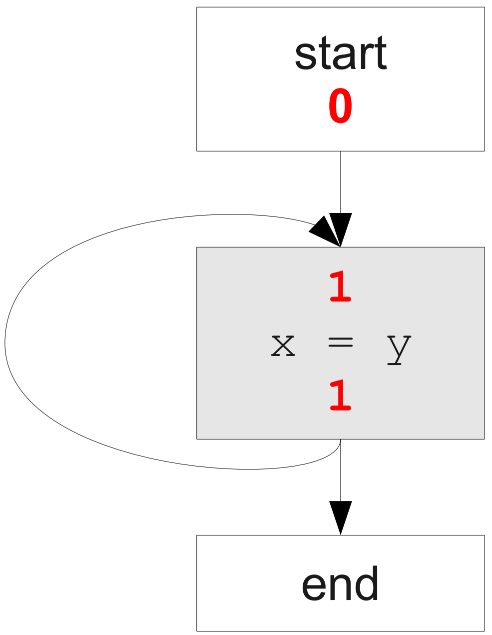
A Nonterminating Analysis



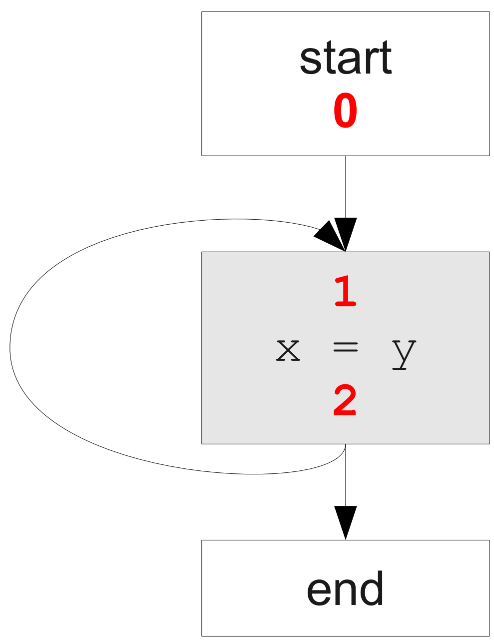
A Nonterminating Analysis



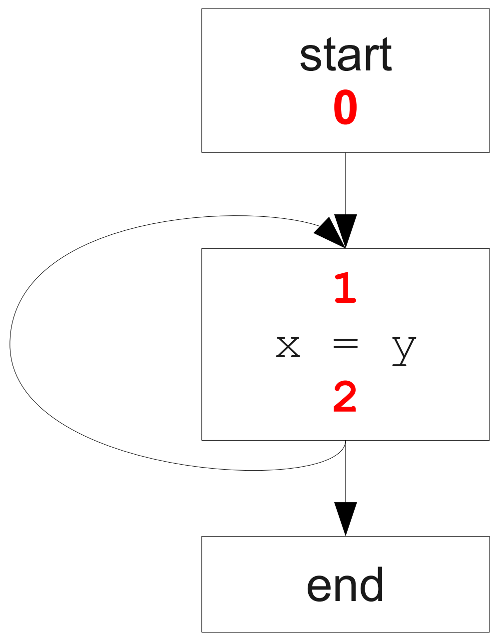
A Nonterminating Analysis



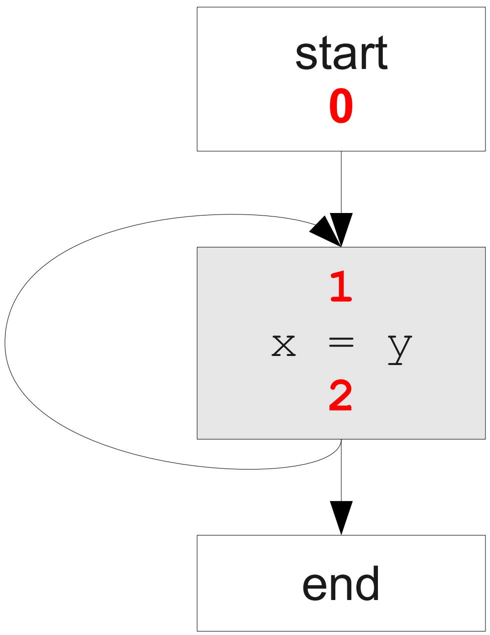
A Nonterminating Analysis



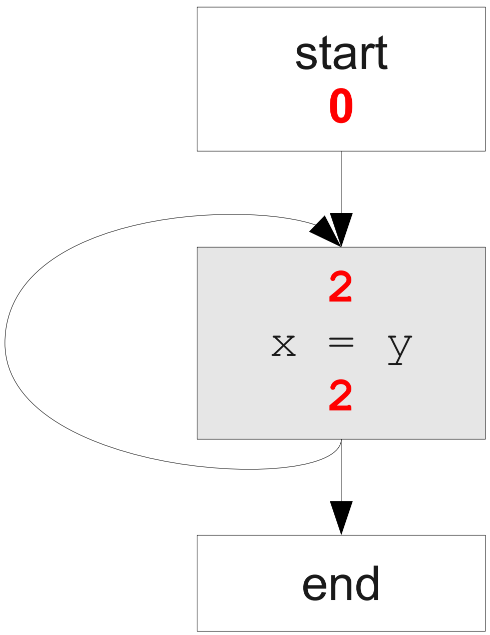
A Nonterminating Analysis



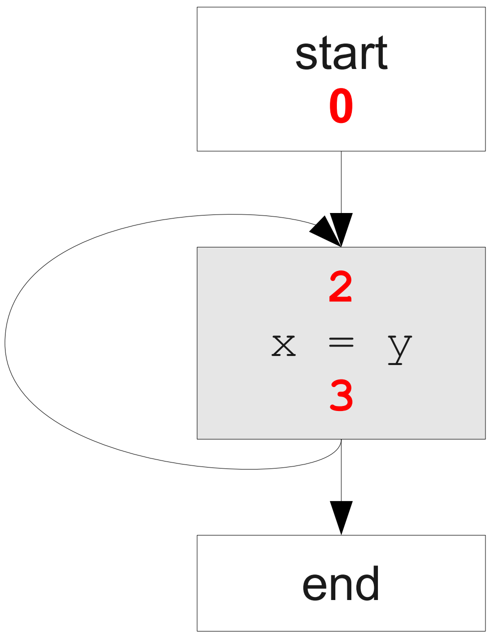
A Nonterminating Analysis



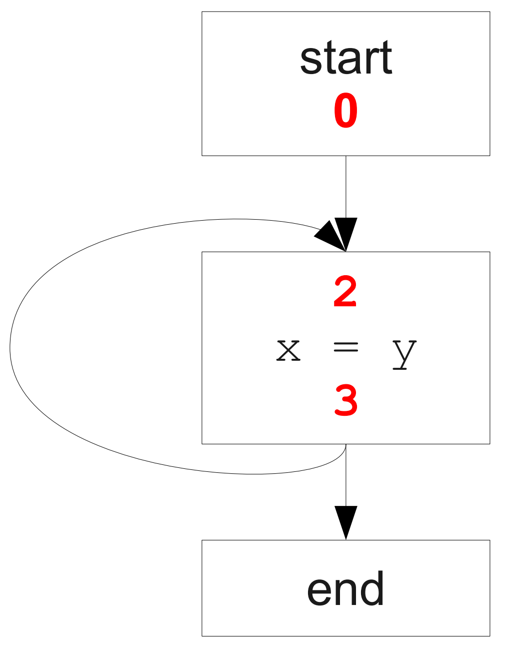
A Nonterminating Analysis



A Nonterminating Analysis

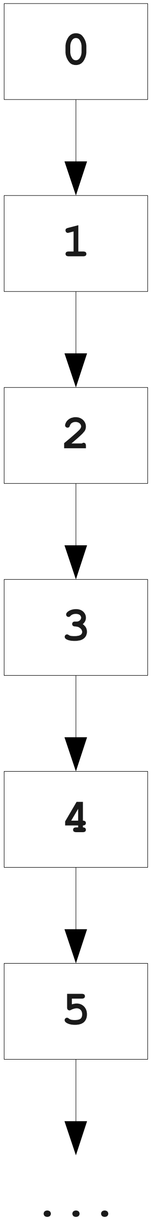


A Nonterminating Analysis



Why Doesn't This Terminate?

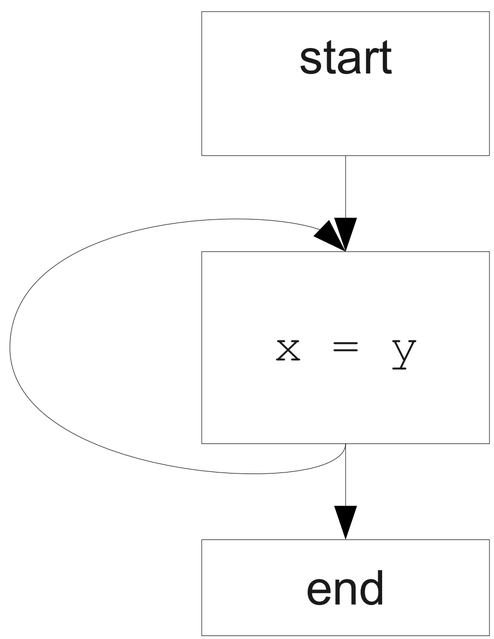
- **Values can decrease without bound.**
  - Note that “decrease” refers to the lattice ordering, not the ordering on the natural numbers.
- The **height** of a semilattice is the length of the longest decreasing sequence in that semilattice.
- The dataflow framework is not guaranteed to terminate for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



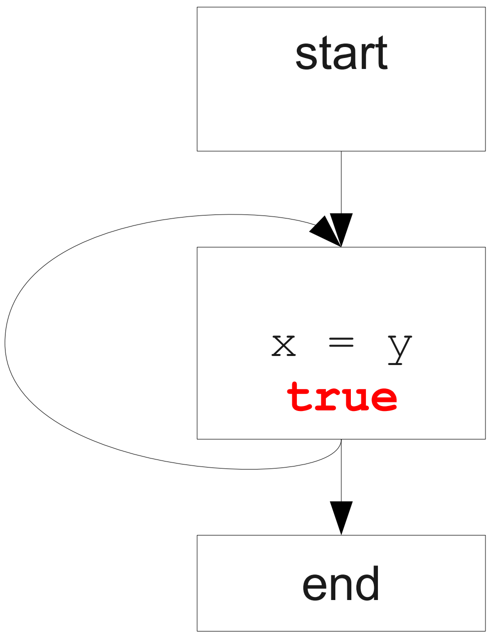
Another Nonterminating Analysis

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

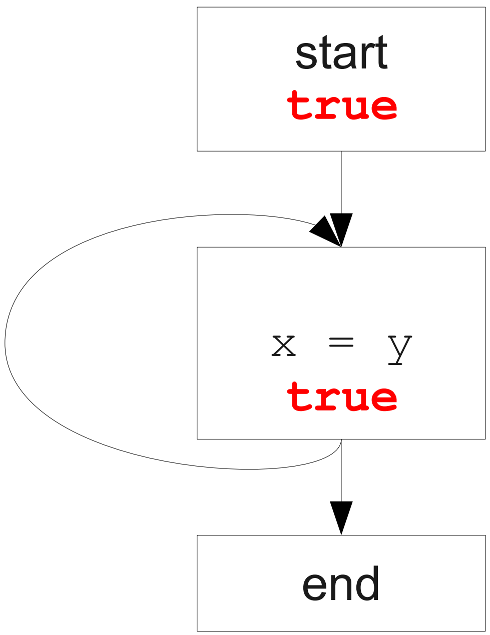
Another Nonterminating Analysis



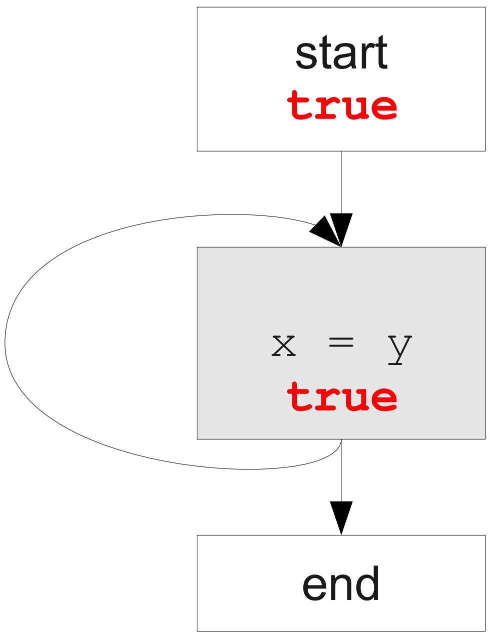
Another Nonterminating Analysis



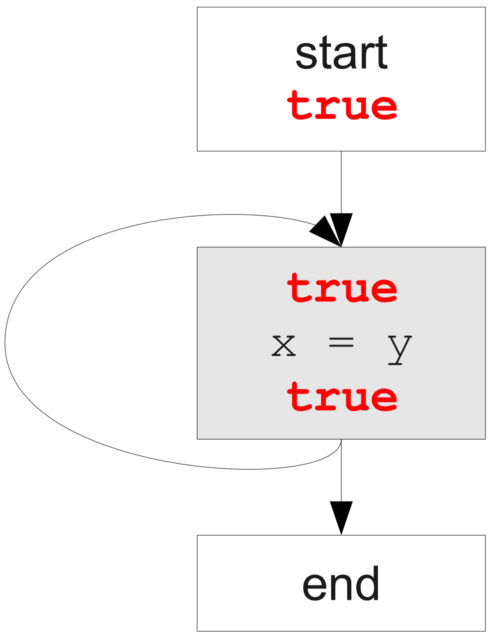
Another Nonterminating Analysis



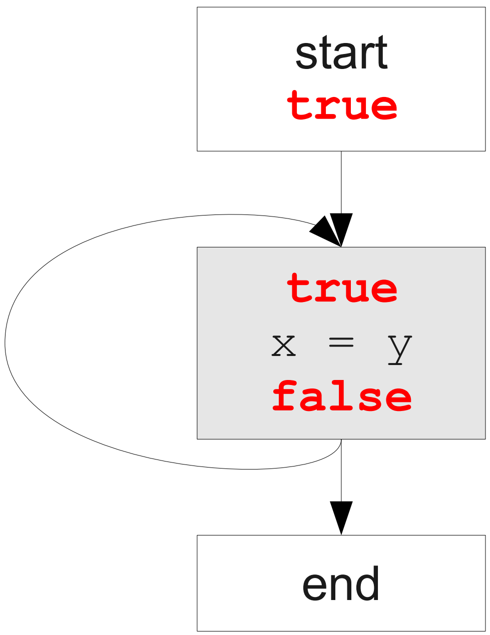
Another Nonterminating Analysis



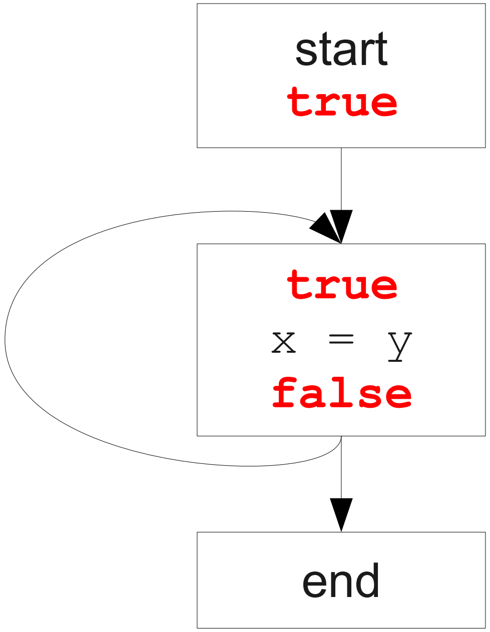
Another Nonterminating Analysis



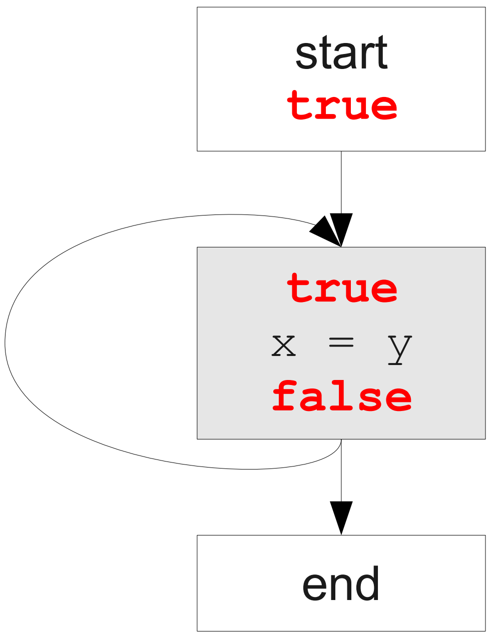
Another Nonterminating Analysis



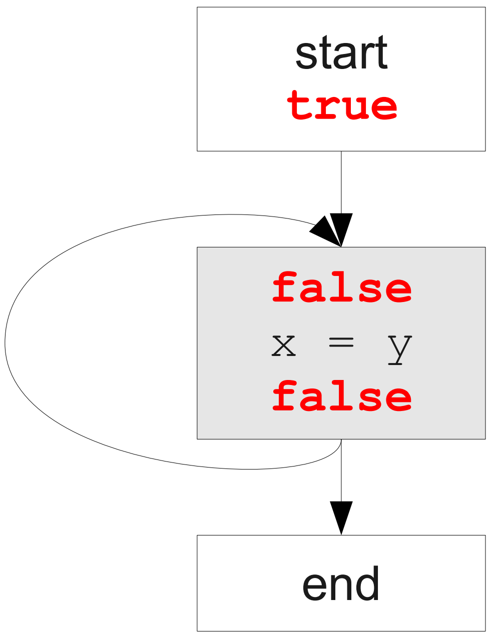
Another Nonterminating Analysis



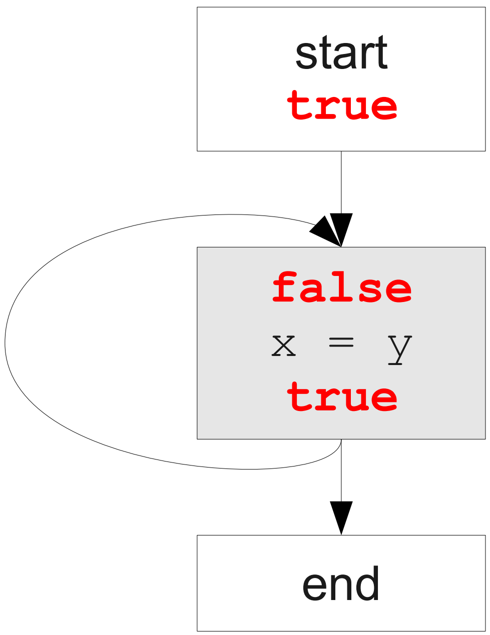
Another Nonterminating Analysis



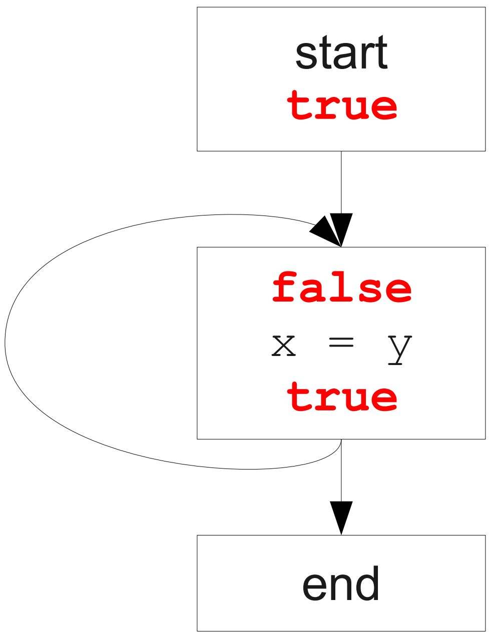
Another Nonterminating Analysis



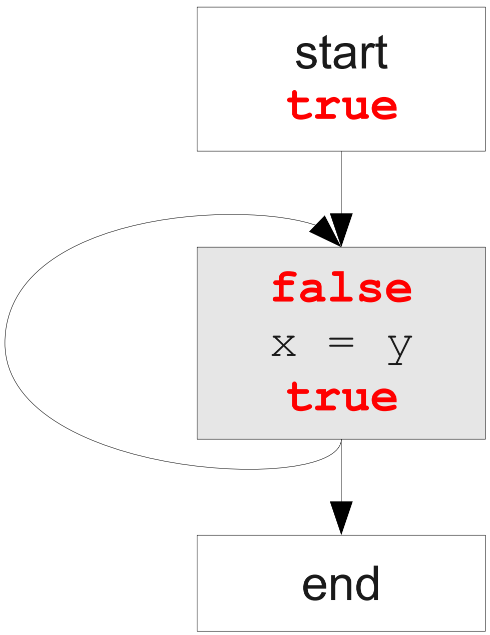
Another Nonterminating Analysis



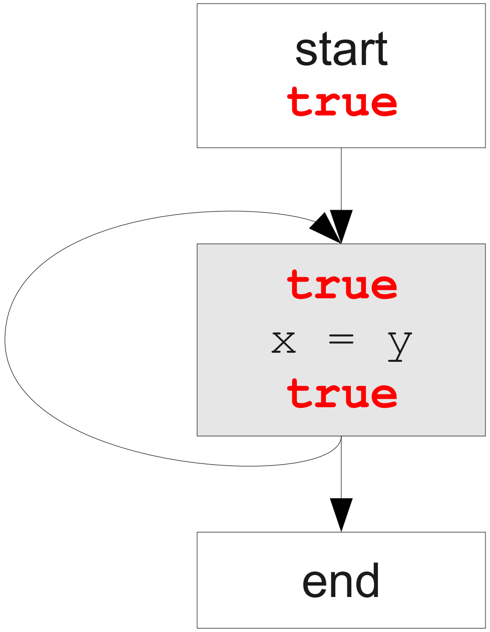
Another Nonterminating Analysis



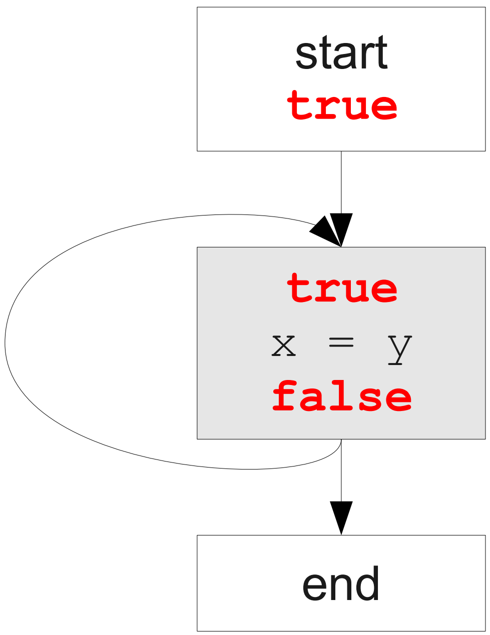
Another Nonterminating Analysis



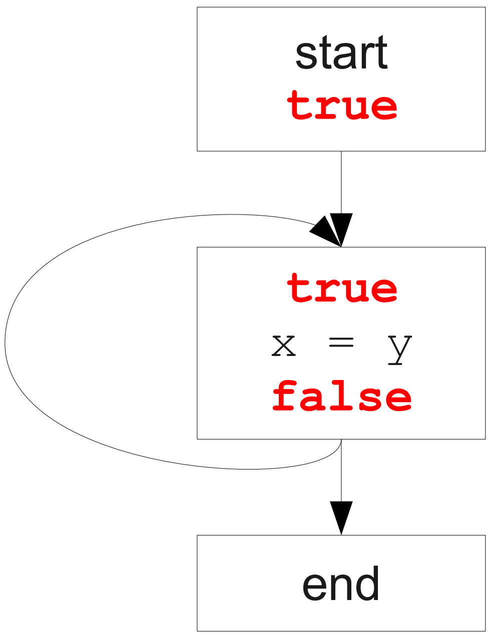
Another Nonterminating Analysis



Another Nonterminating Analysis



Another Nonterminating Analysis



What Went Wrong (This Time)?

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

What Went Wrong (This Time)?

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

What Went Wrong (This Time)?

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

What Went Wrong (This Time)?

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

Monotone Transfer Functions

- A transfer function is **monotone** if for all x and y

  if $x \leq y$, $f(x) \leq f(y)$
- 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 $f(x) \leq x$; we'll see an example.

Constant Propagation is Monotone

- A transfer function is **monotone** if for all x and y

  if $x \leq y$, $f(x) \leq f(y)$
- Recall our transfer functions are
  - $f_{x = k}(V) = k$
  - $f_{x = a + b}(V) =$ Not a Constant
  - $f_{y = a + b}(V) = V$
- The first two rules are monotone:
  - $f(x) = f(y)$ for all x and y, so $f(x) \leq f(y)$ for all x and y.
- The last rule is monotone:
  - if $x \leq y$, $f(x) = x \leq y = f(y)$

Liveness is Monotone

- A transfer function is **monotone** if for all x and y

  if $x \leq y$, $f(x) \leq f(y)$
- Recall our transfer function for **a** = **b** + **c** is
  - $f_{a = b + c}(V) = (V - \mathbf{a}) \cup \{\mathbf{b}, \mathbf{c}\}$
- Recall that our meet semilattice has set union as a transfer function and induces an ordering relationship $X \leq Y$ iff $X \supseteq Y$.
- Suppose $X \supseteq Y$.
- Then $(X - \mathbf{a}) \supseteq (Y - \mathbf{a})$.
- Then $(X - \mathbf{a}) \cup \{\mathbf{b}, \mathbf{c}\} \supseteq (Y - \mathbf{a}) \cup \{\mathbf{b}, \mathbf{c}\}$.
- So $f(X) \supseteq f(Y)$.

The Grand Result

- **Theorem:** A dataflow analysis with a finite-height semilattice and family of monotone transfer functions always terminates.
- Proof sketch:
  - Run the data-flow iteration once to get some initial values.
  - From this point forward:
    - The meet operator can only bring values down.
    - The transfer function can never raise values back up above where they were in the past (monotonicity)
    - Values cannot decrease indefinitely (finite height)

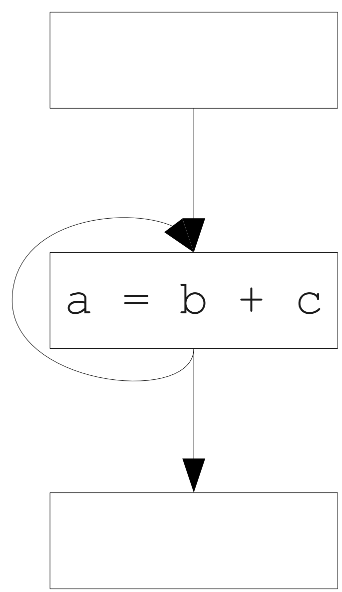
Partial-Redundancy Elimination

Code Size is Not Execution Time

- All of the analyses we've seen so far have worked by simplifying or eliminating IR code.
- However, much of optimization results from **moving** code from one basic block to another.

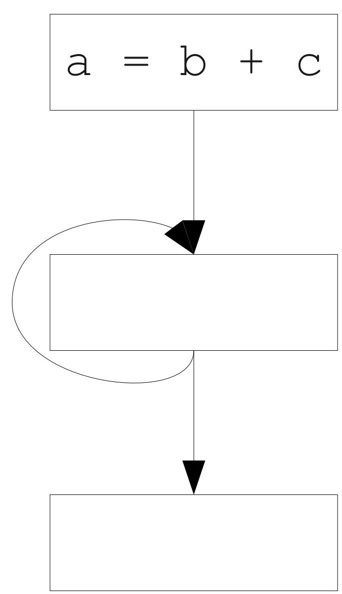
Code Size is Not Execution Time

- All of the analyses we've seen so far have worked by simplifying or eliminating IR code.
- However, much of optimization results from **moving** code from one basic block to another.



Code Size is Not Execution Time

- All of the analyses we've seen so far have worked by simplifying or eliminating IR code.
- However, much of optimization results from **moving** code from one basic block to another.

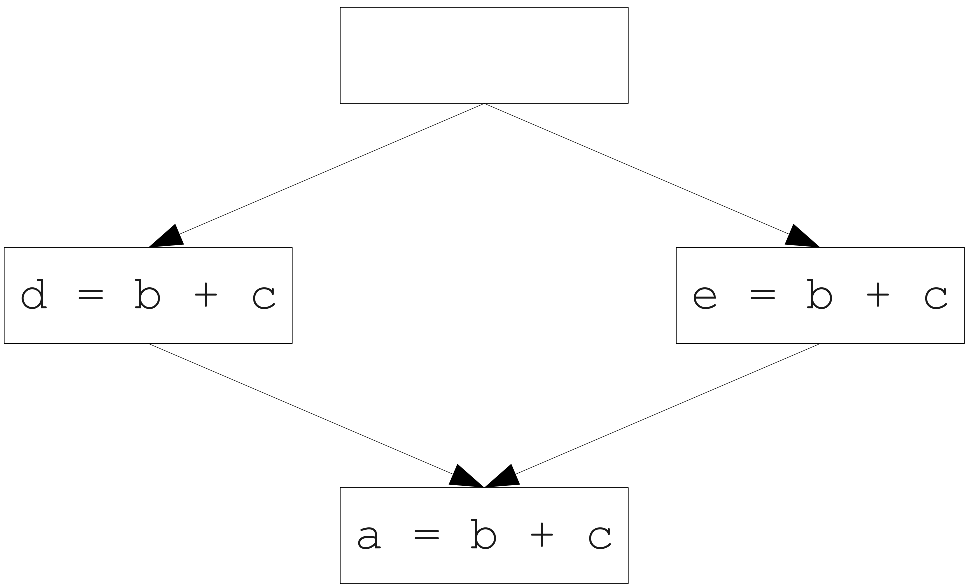


Code Size is Not Execution Time

- In some cases, it is possible to decrease execution time by **inserting** new code into the program.
- One possible example:

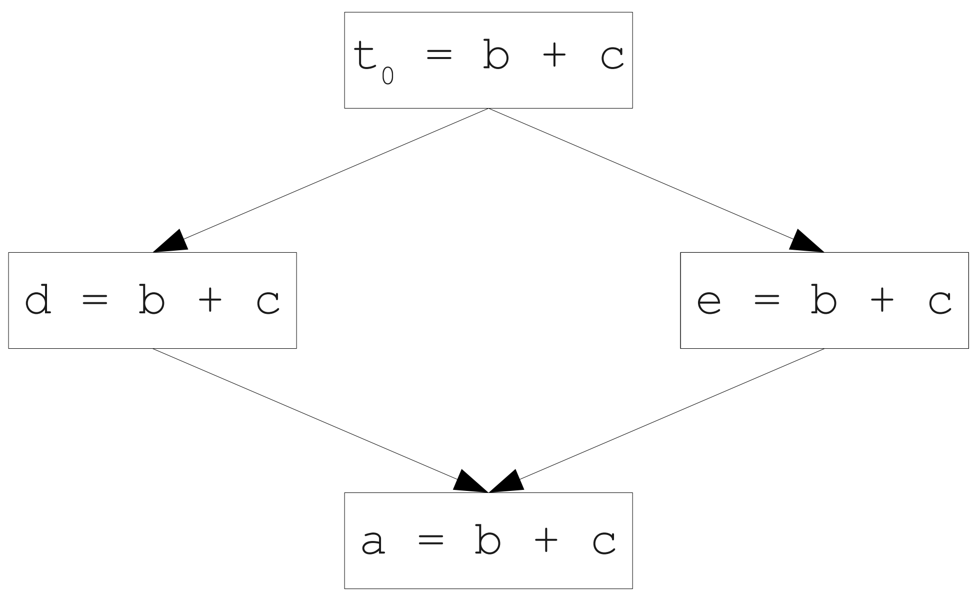
Code Size is Not Execution Time

- In some cases, it is possible to decrease execution time by **inserting** new code into the program.
- One possible example:



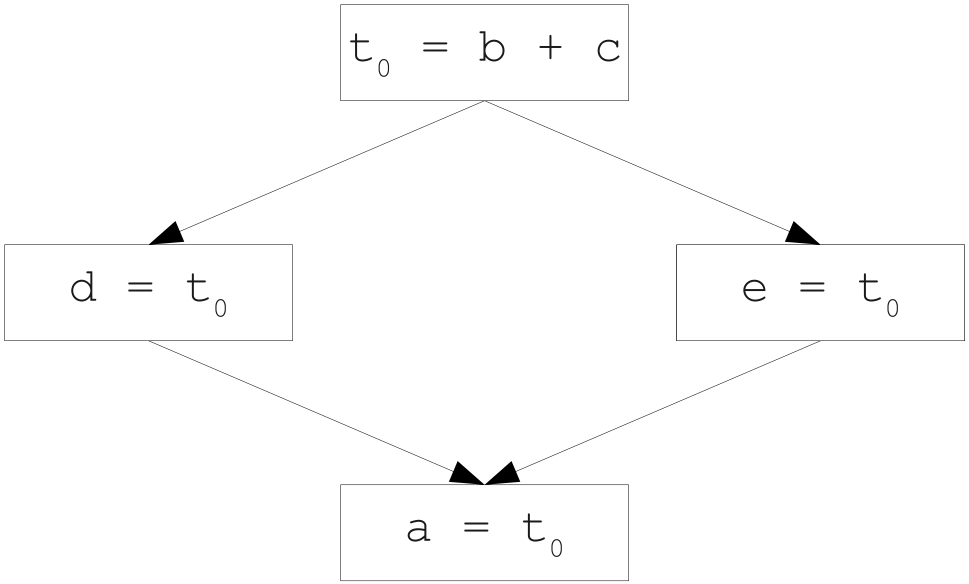
Code Size is Not Execution Time

- In some cases, it is possible to decrease execution time by **inserting** new code into the program.
- One possible example:



Code Size is Not Execution Time

- In some cases, it is possible to decrease execution time by **inserting** new code into the program.
- One possible example:

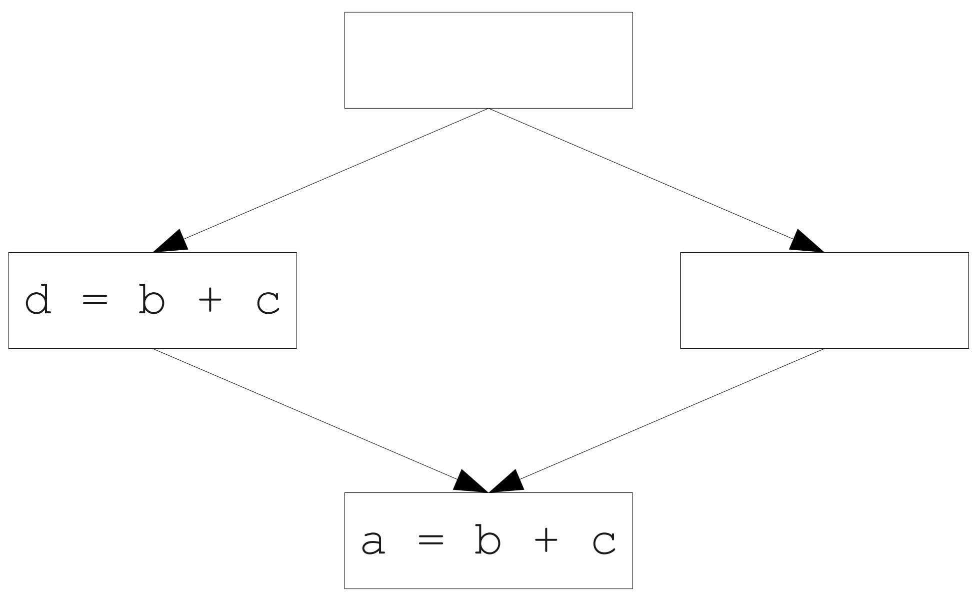


Eliminating Redundancy

- A computation in a program is said to be **redundant** if it computes a value that is already known.
- Common subexpressions are one example of redundancy.
- Loop-invariant code is another example.
- Virtually all optimizing compilers have some logic to try to eliminate redundancy.

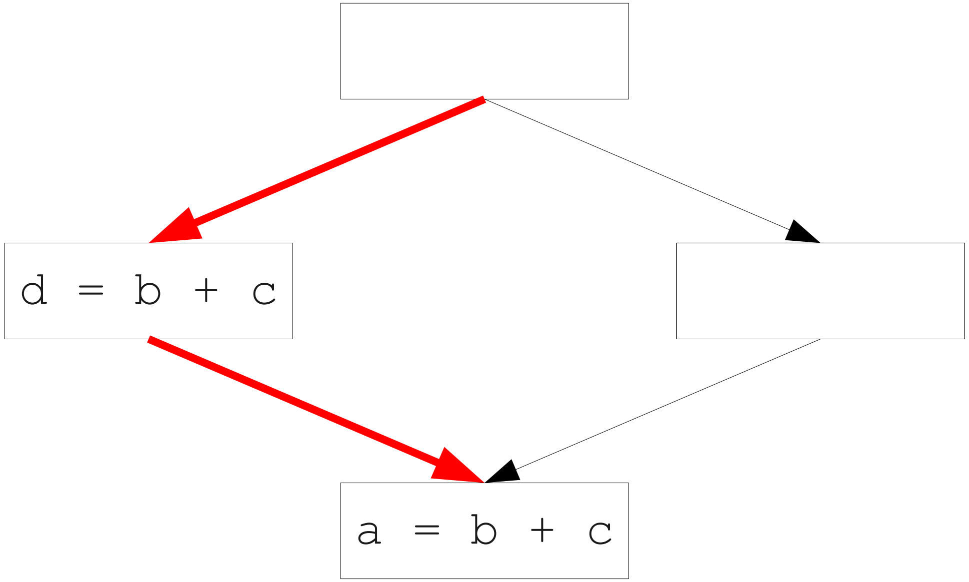
Partial Redundancy

- One of the trickiest cases of redundancy to eliminate is **partial redundancy**.
- A computation is **partially redundant** if its value is known on only some of the paths that reach it.



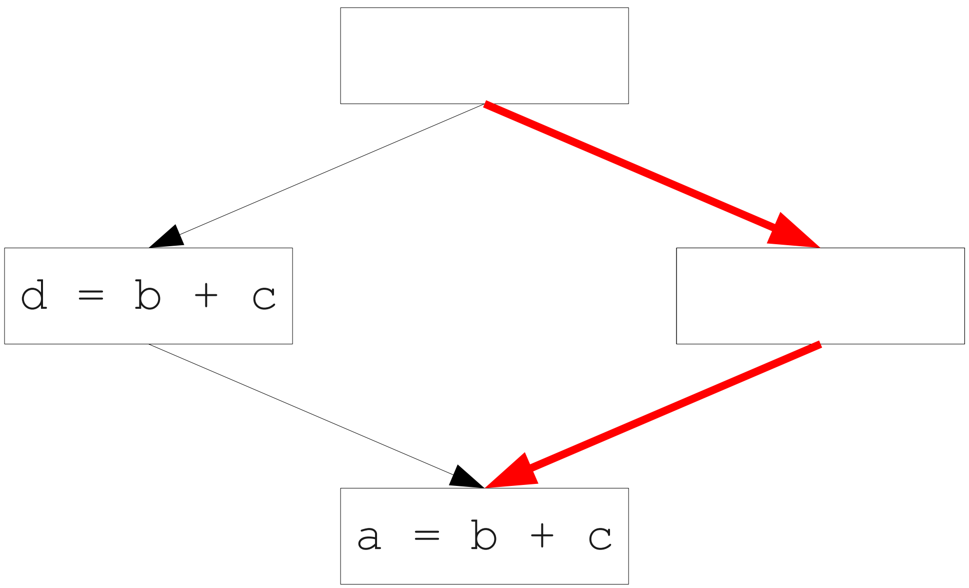
Partial Redundancy

- One of the trickiest cases of redundancy to eliminate is **partial redundancy**.
- A computation is **partially redundant** if its value is known on only some of the paths that reach it.



Partial Redundancy

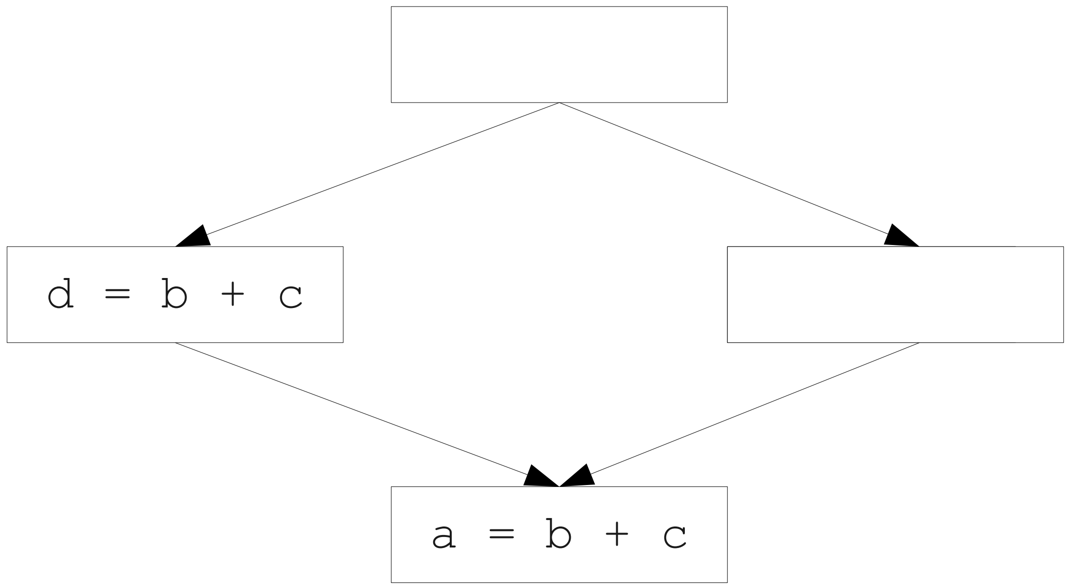
- One of the trickiest cases of redundancy to eliminate is **partial redundancy**.
- A computation is **partially redundant** if its value is known on only some of the paths that reach it.



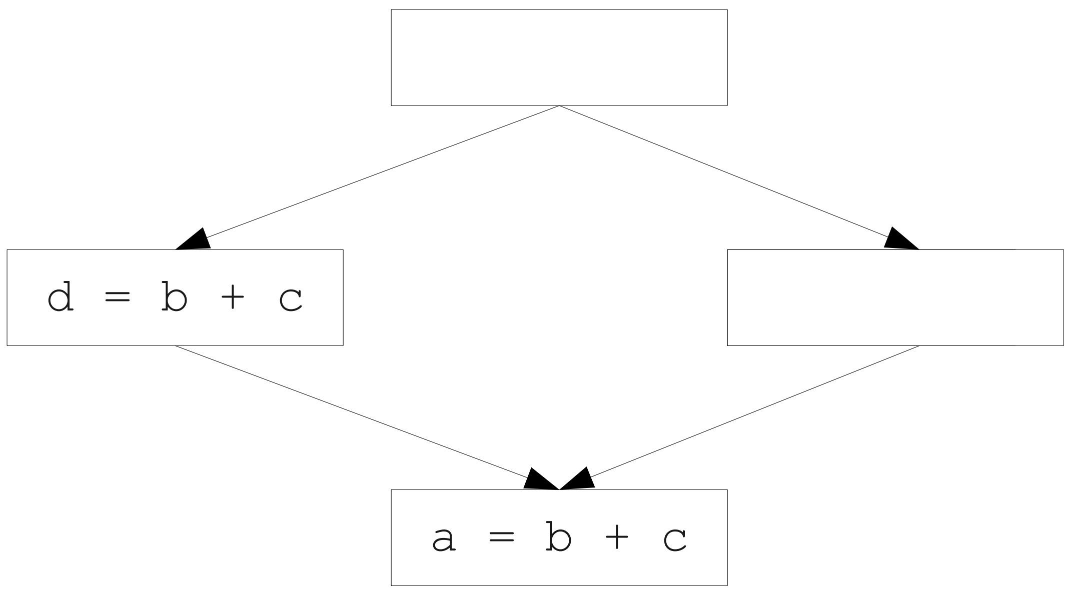
Eliminating Partial Redundancy

- Goal: Eliminate partial redundancy without making any execution of the program do more work than before.
- Optimized code should **always** be at least as good as the original.

The Key Observation

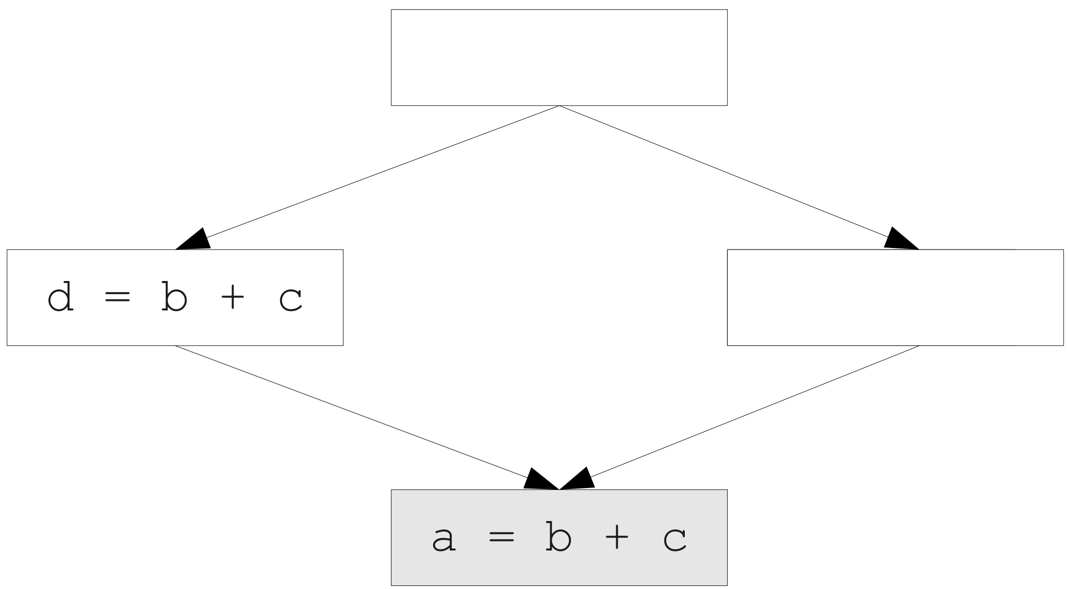


The Key Observation



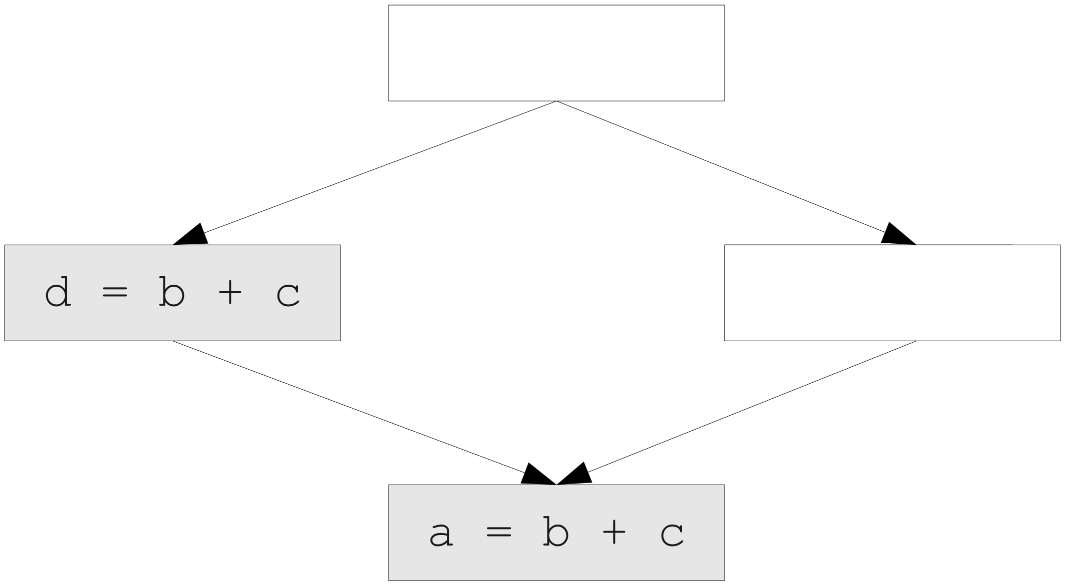
Where in the program is it <u>guaranteed</u> that we will need the value of **b + c**?

The Key Observation



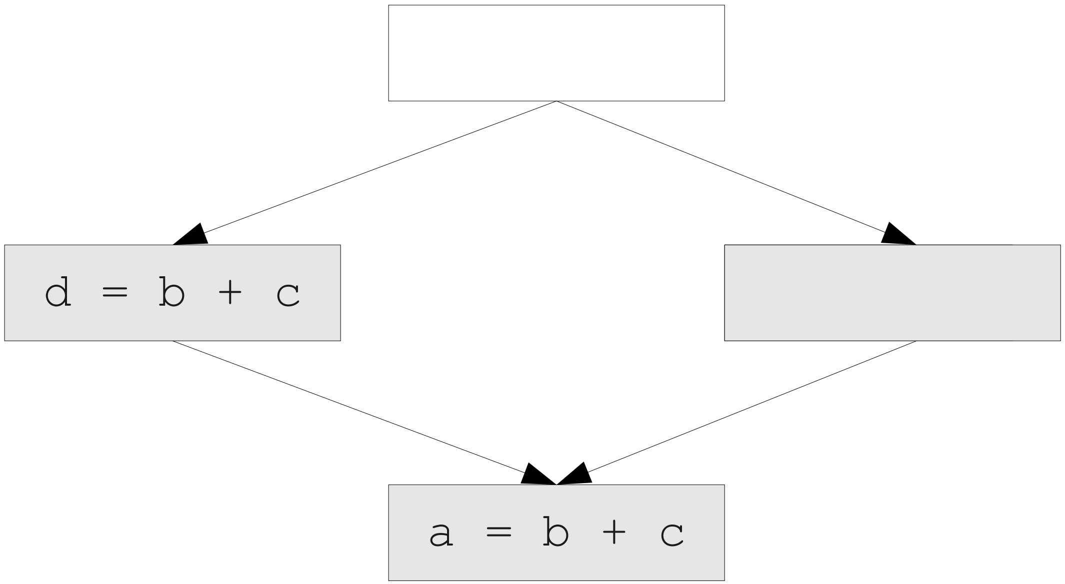
Where in the program is it <u>guaranteed</u> that we will need the value of **b + c**?

The Key Observation



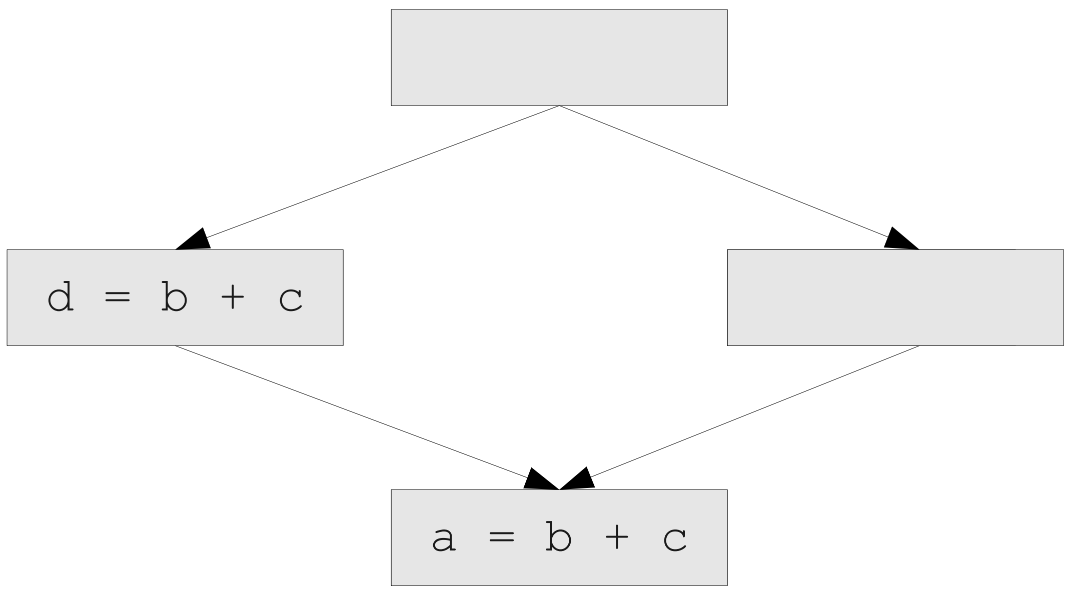
Where in the program is it guaranteed that we will need the value of **b + c**?

The Key Observation



Where in the program is it guaranteed that we will need the value of **b + c**?

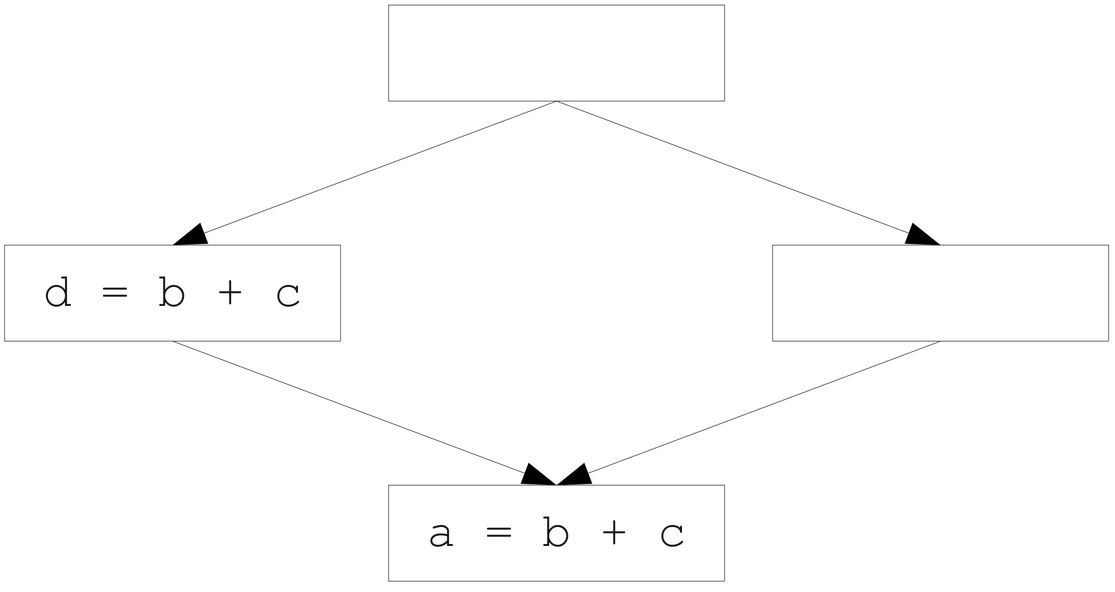
The Key Observation



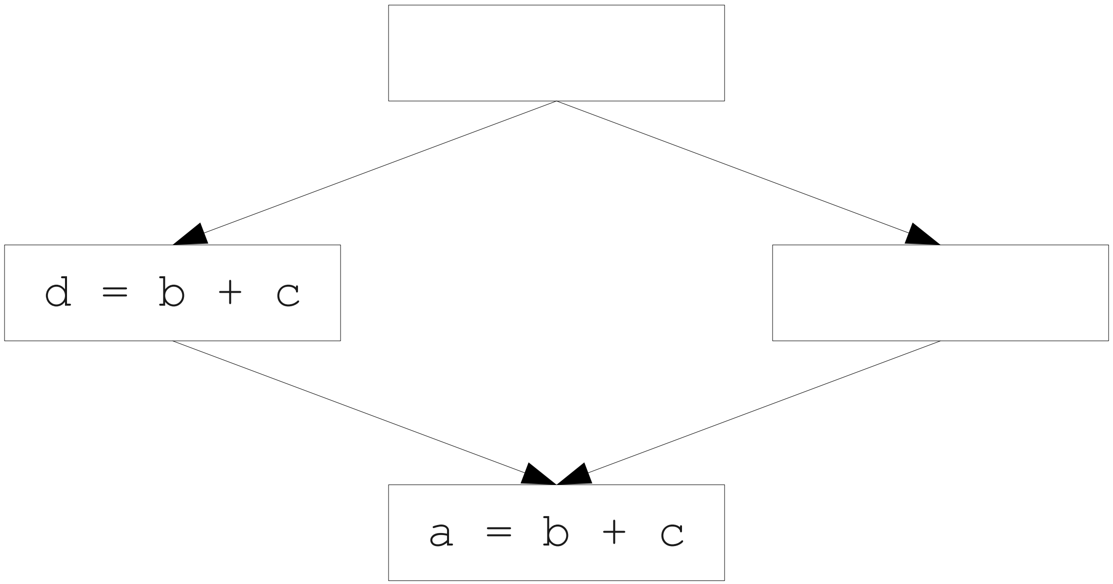
Where in the program is it guaranteed that we will need the value of **b + c**?

Although not all paths through the program might directly need an expression, they may **anticipate** the expression.

The Second Key Observation

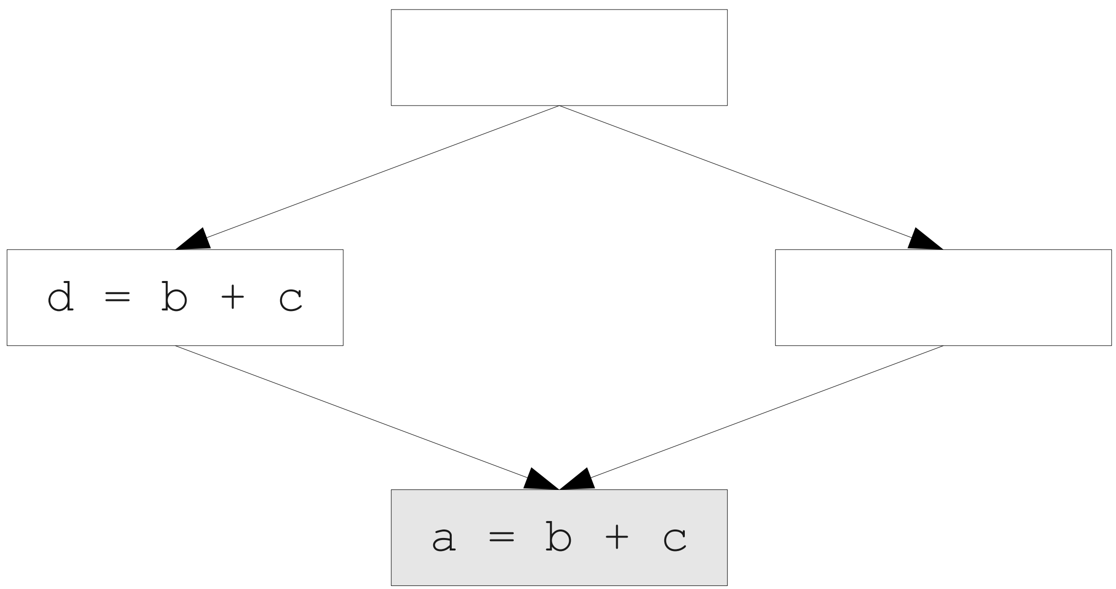


The Second Key Observation



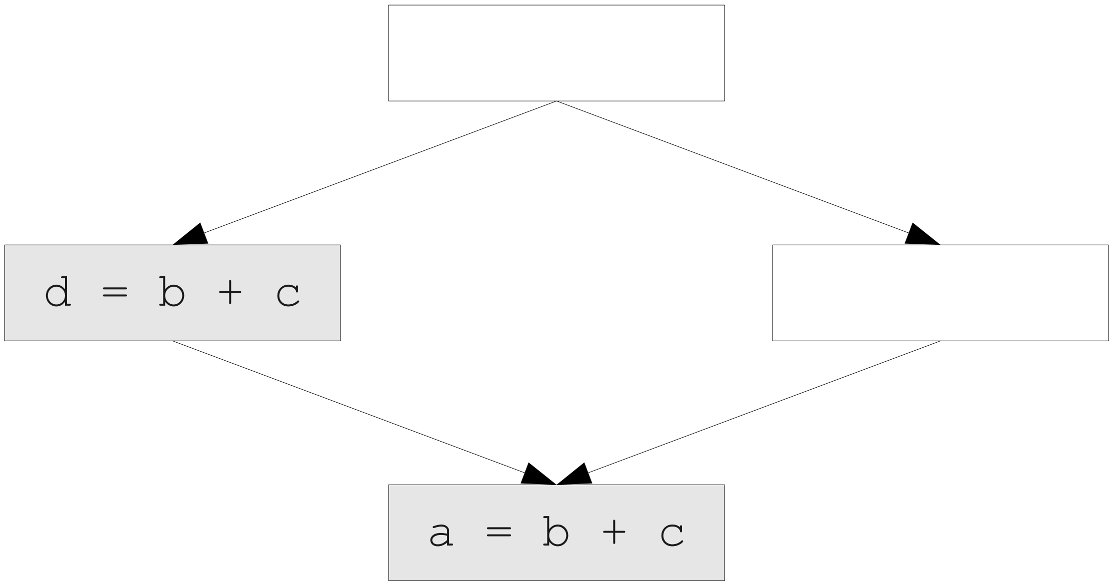
Where in the program is the value of **b + c** already computed?

The Second Key Observation



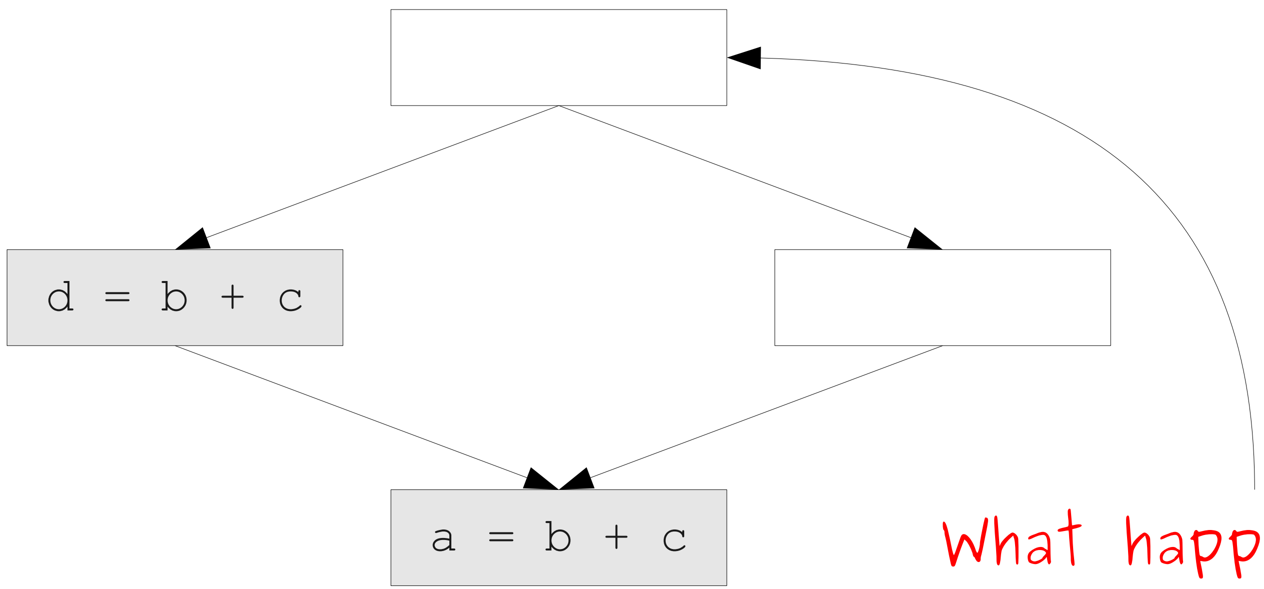
Where in the program
is the value of **b + c**
already computed?

The Second Key Observation



Where in the program
is the value of **b + c**
already computed?

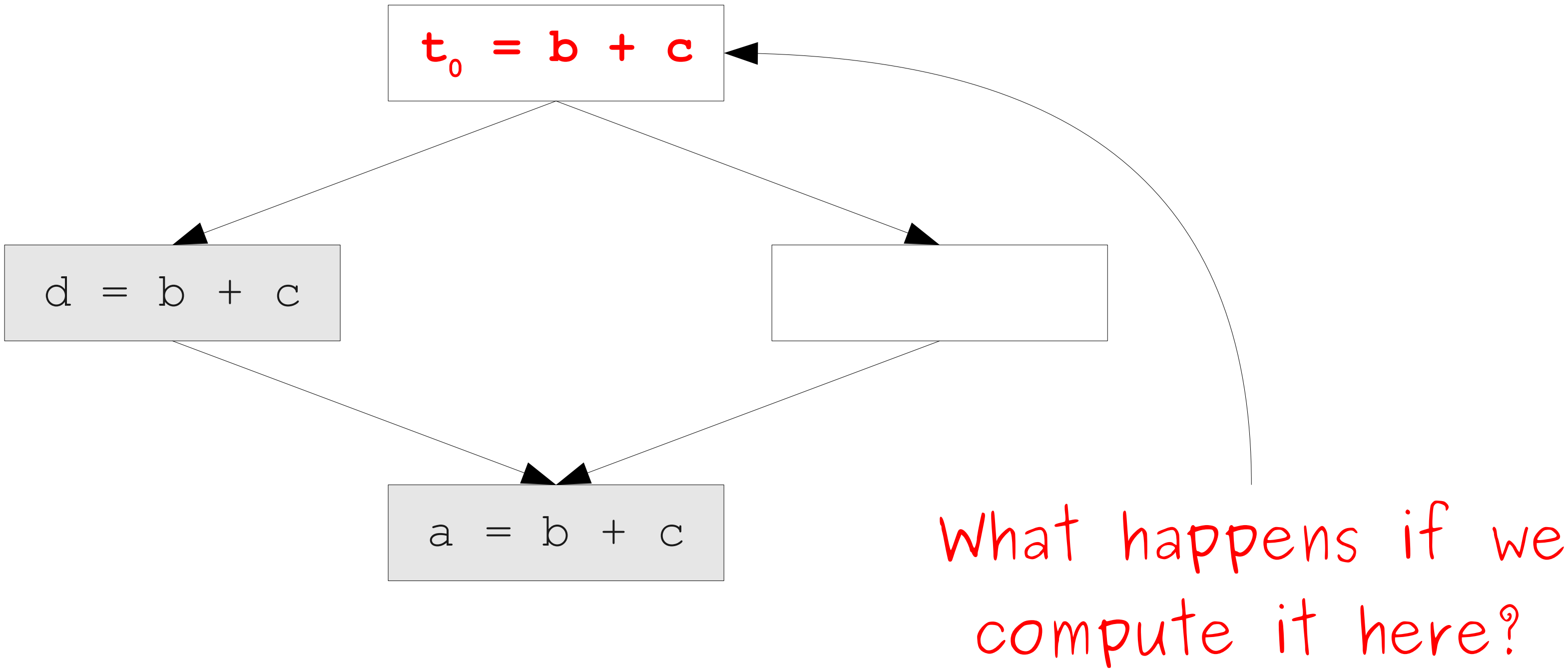
The Second Key Observation



What happens if we compute it here?

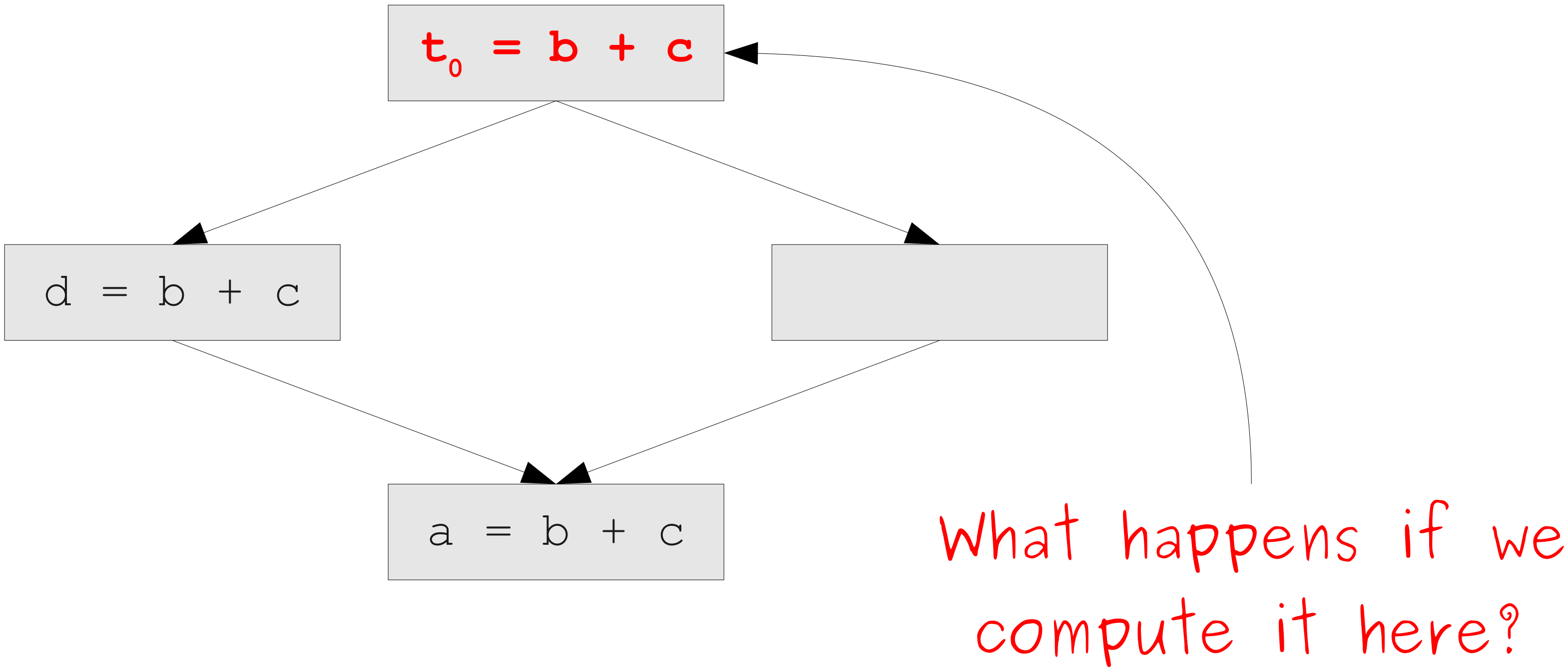
Where in the program is the value of **b + c** already computed?

The Second Key Observation



Where in the program is the value of **b + c** already computed?

The Second Key Observation



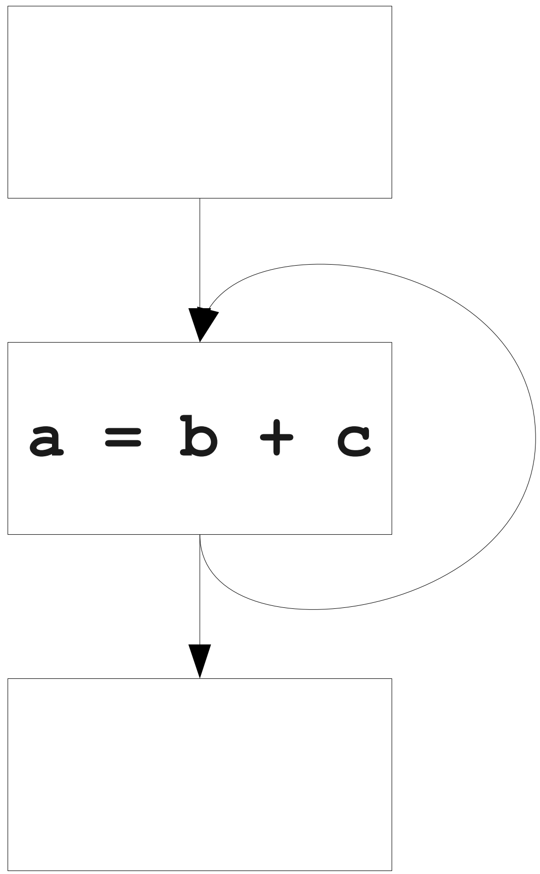
Where in the program is the value of **b + c** already computed?

Partial-Redundancy Elimination

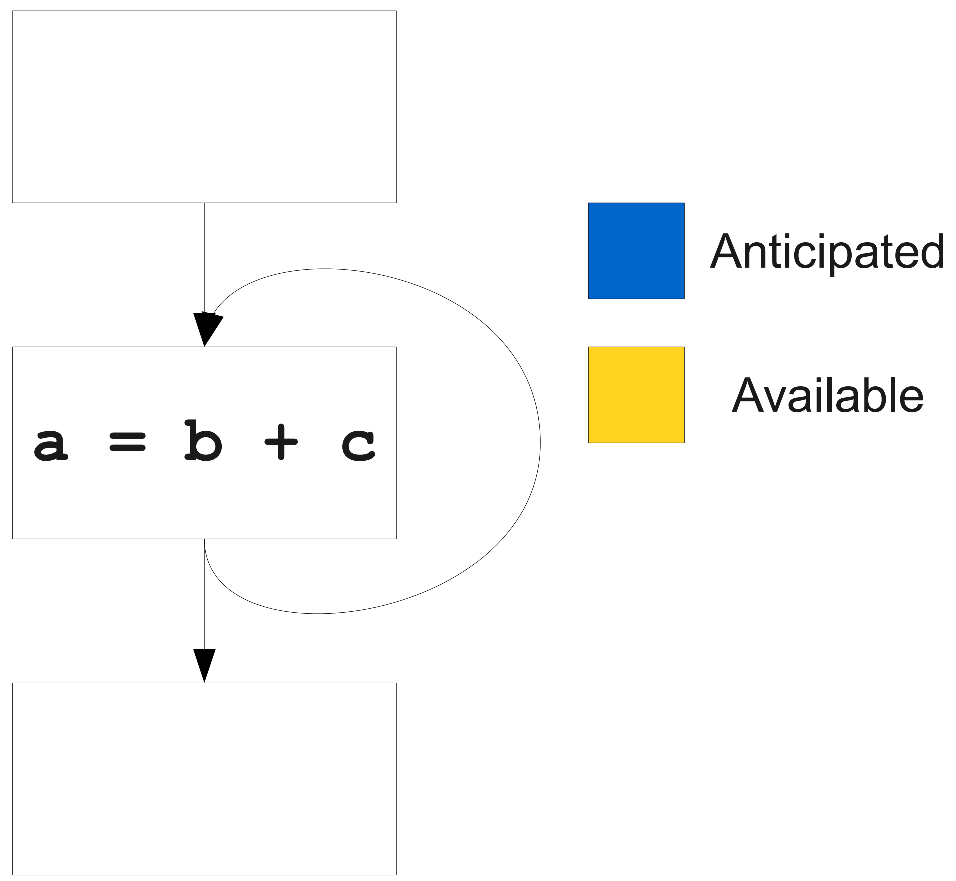
- **Idea:** Make the expression available everywhere that it's anticipated.
- Run an analysis to locate where the expression is anticipated.
- Run a second analysis to locate where the expression is available.
- Place the expression at the earliest locations where the expression is **anticipated** but not **available**.

Eliminating Redundancy

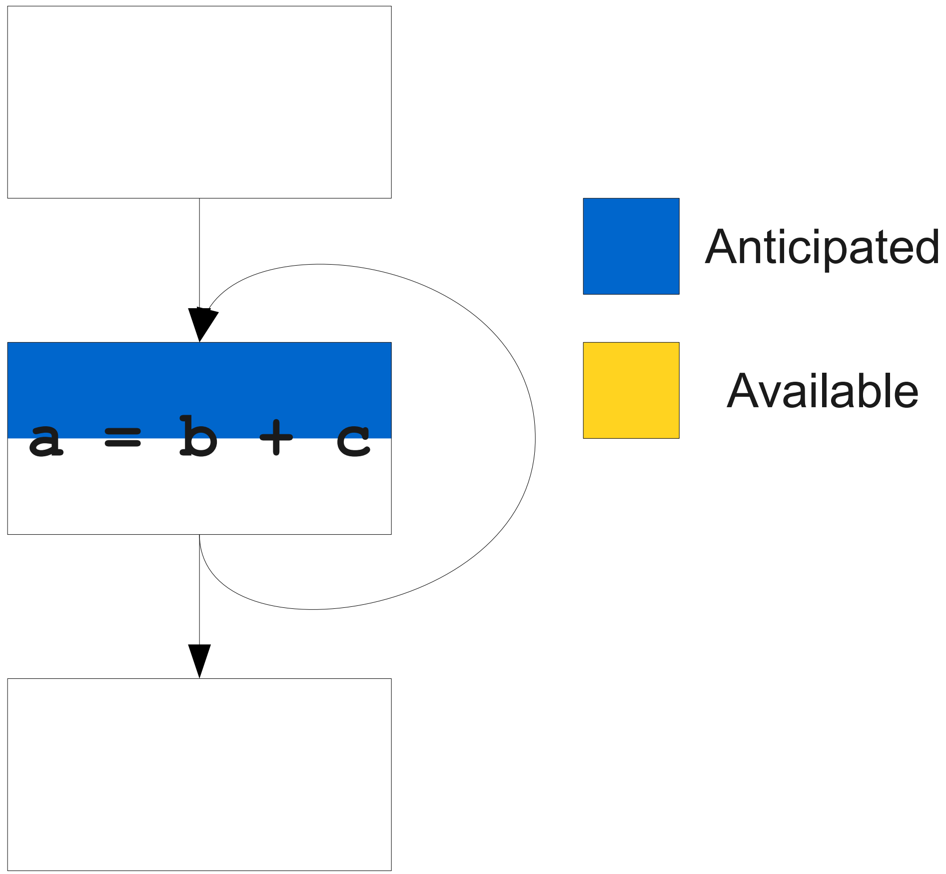
Eliminating Redundancy



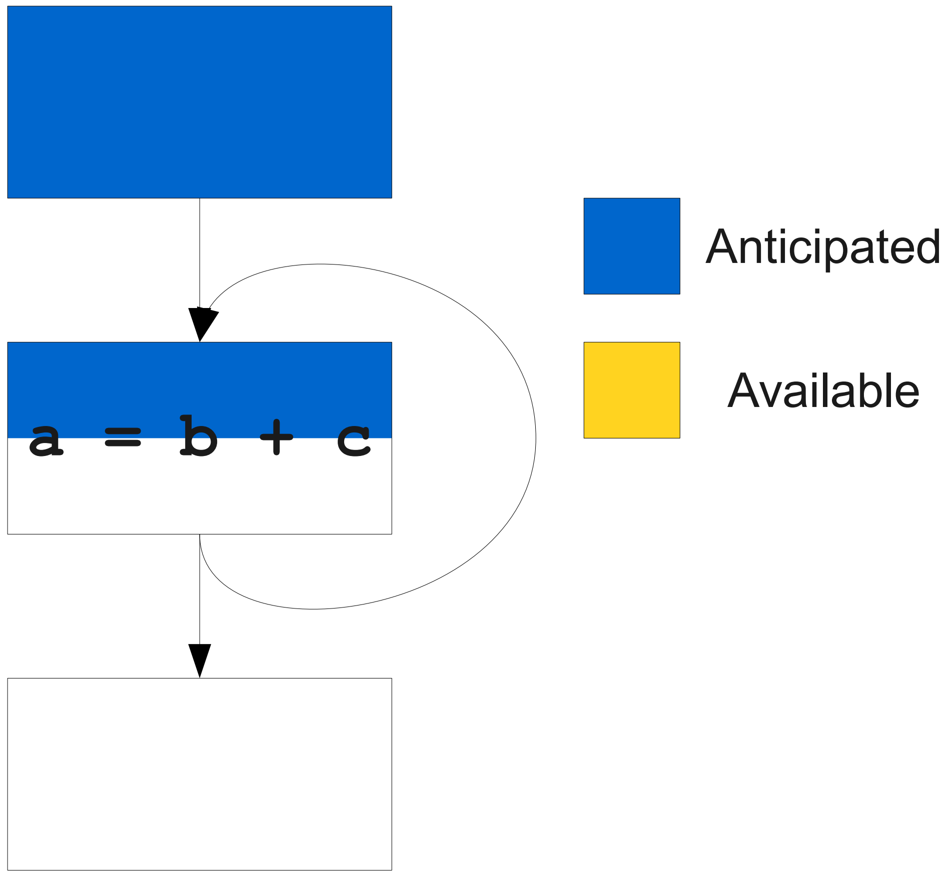
Eliminating Redundancy



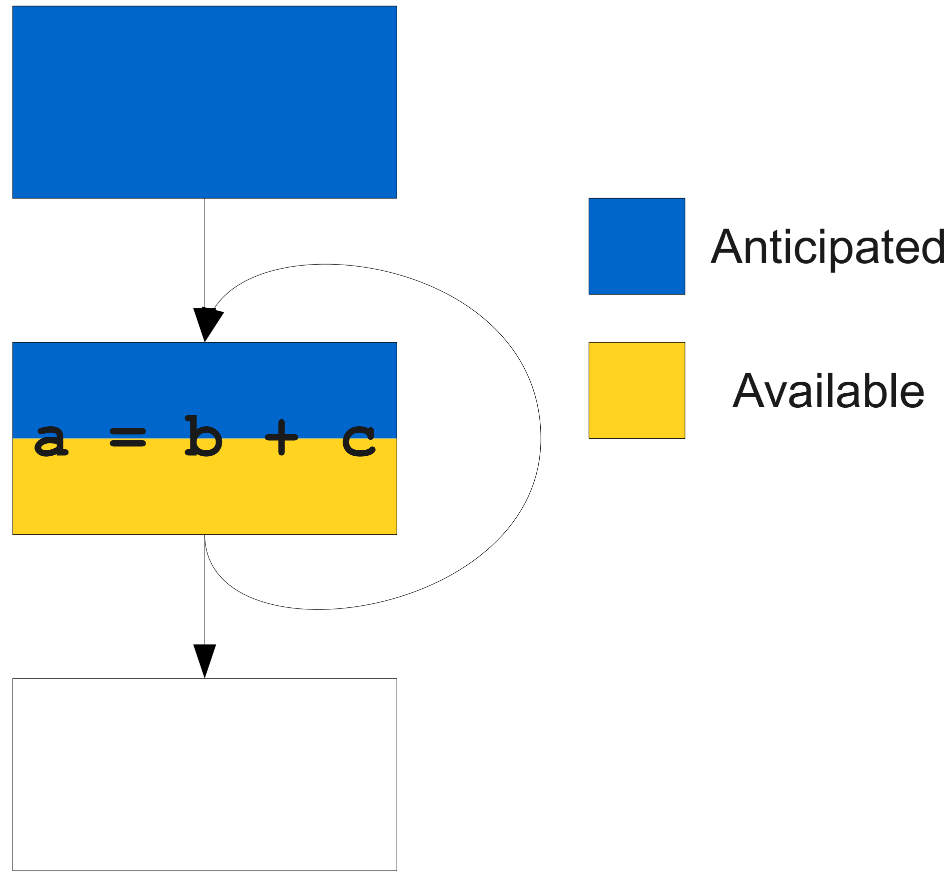
Eliminating Redundancy



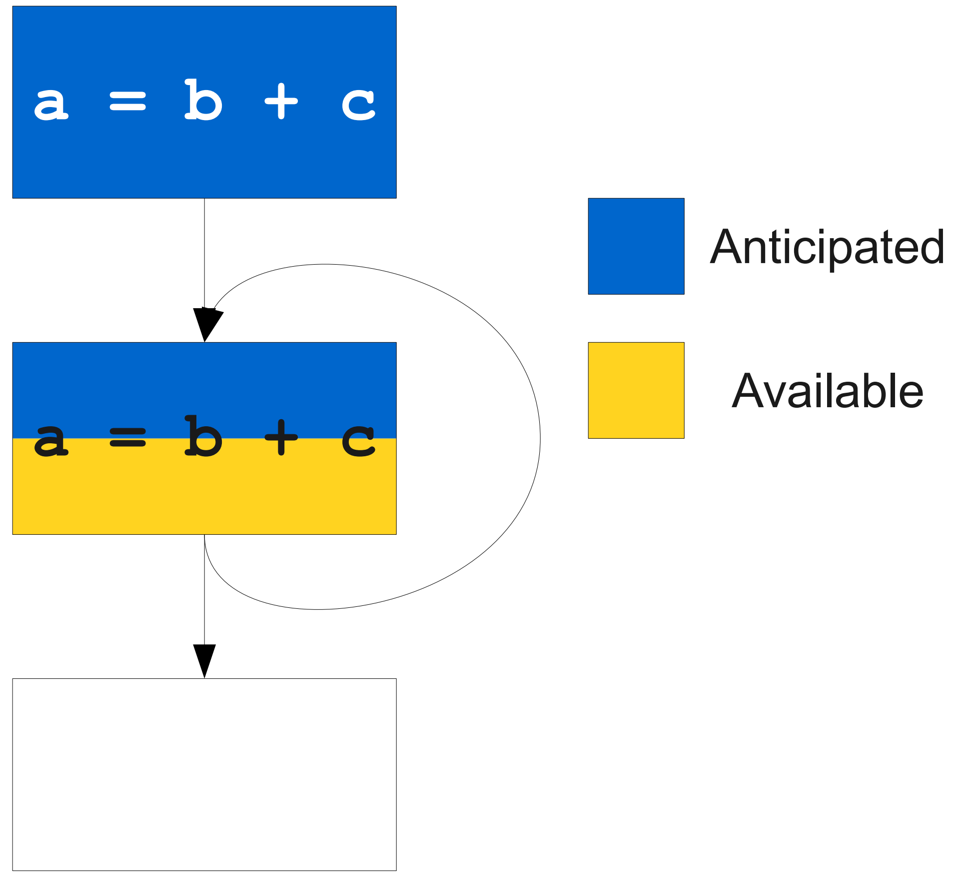
Eliminating Redundancy



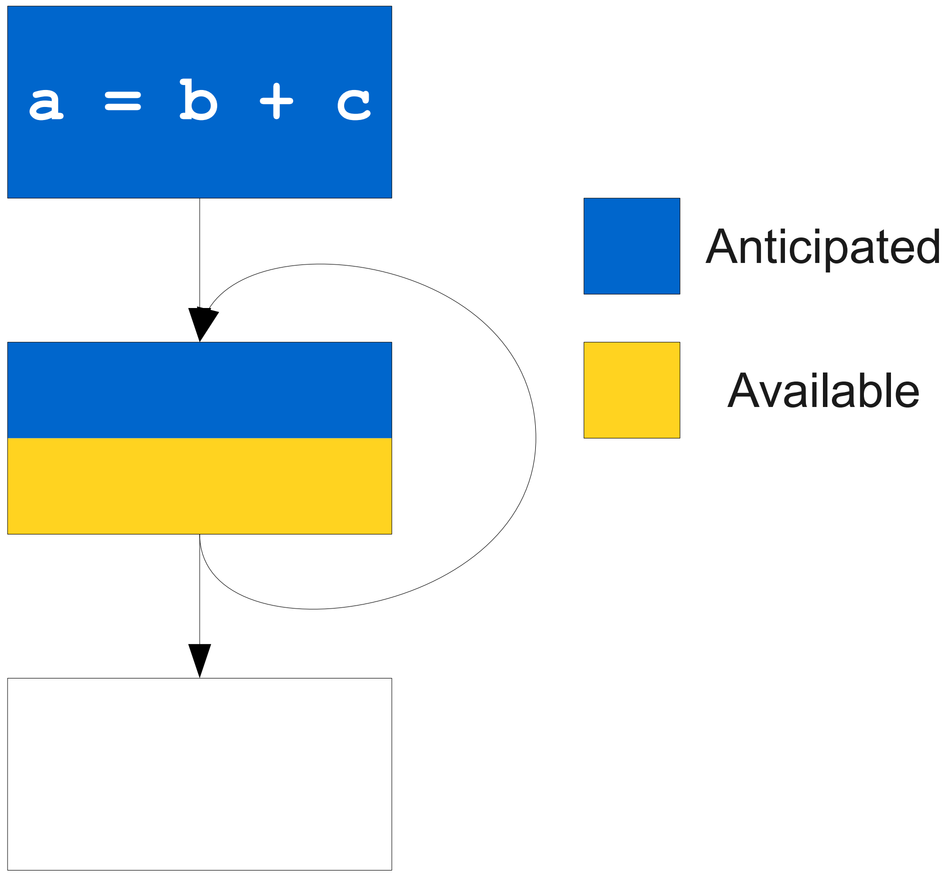
Eliminating Redundancy



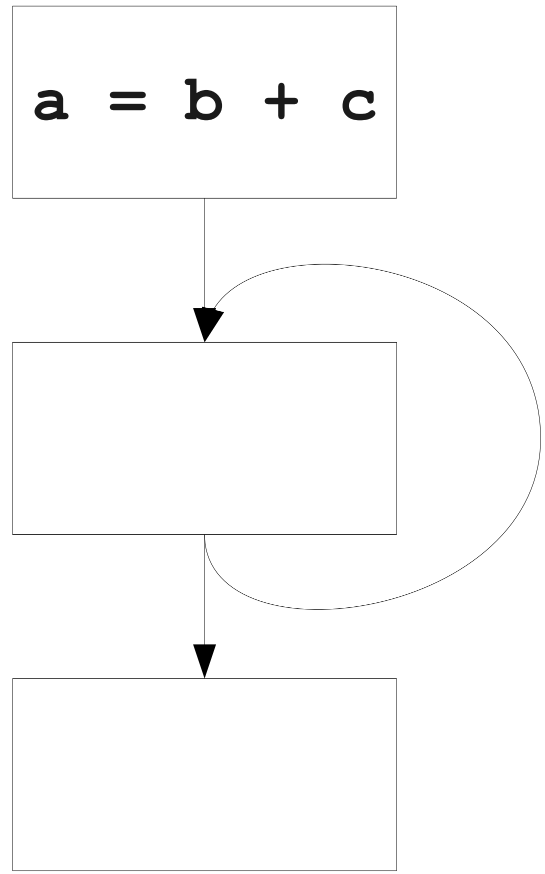
Eliminating Redundancy



Eliminating Redundancy

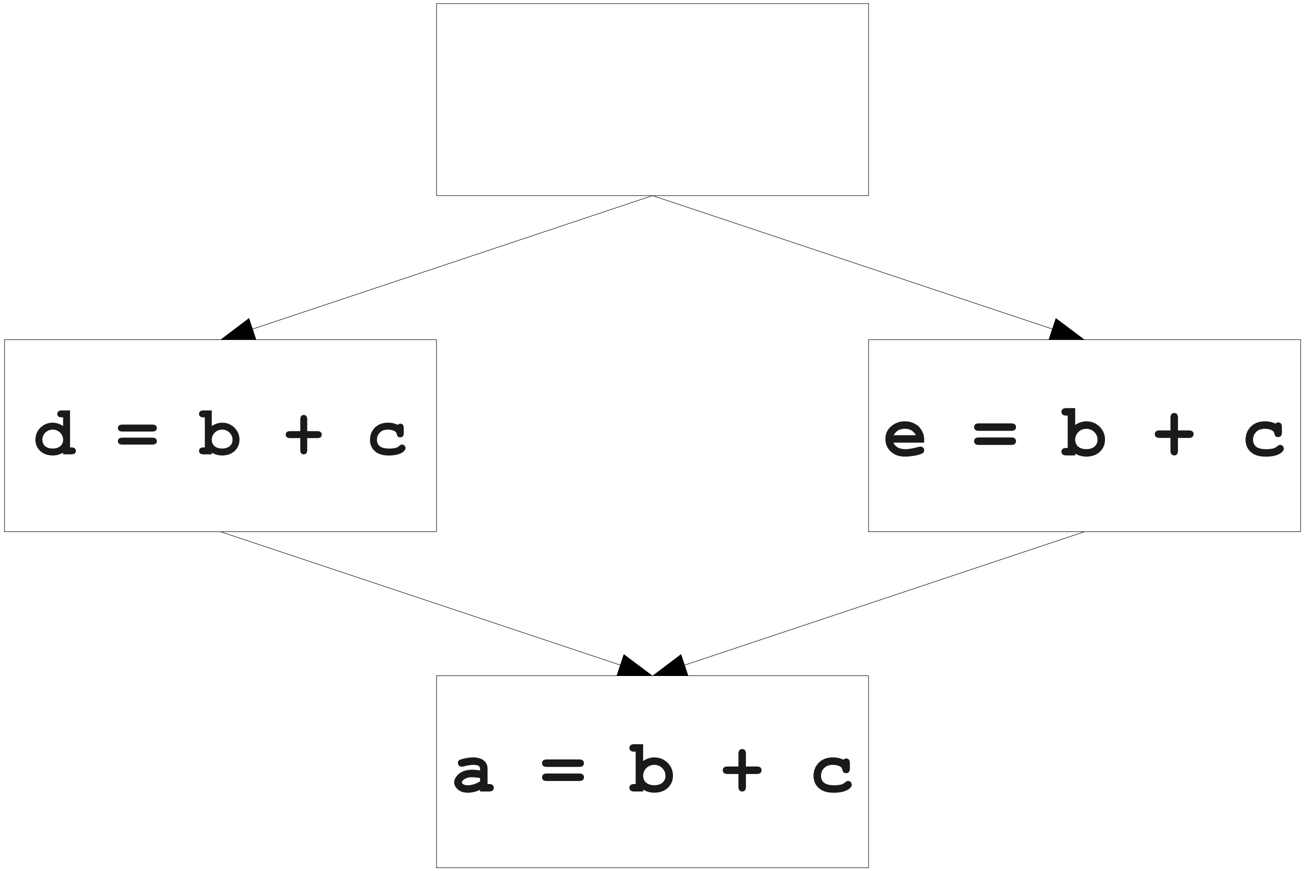


Eliminating Redundancy

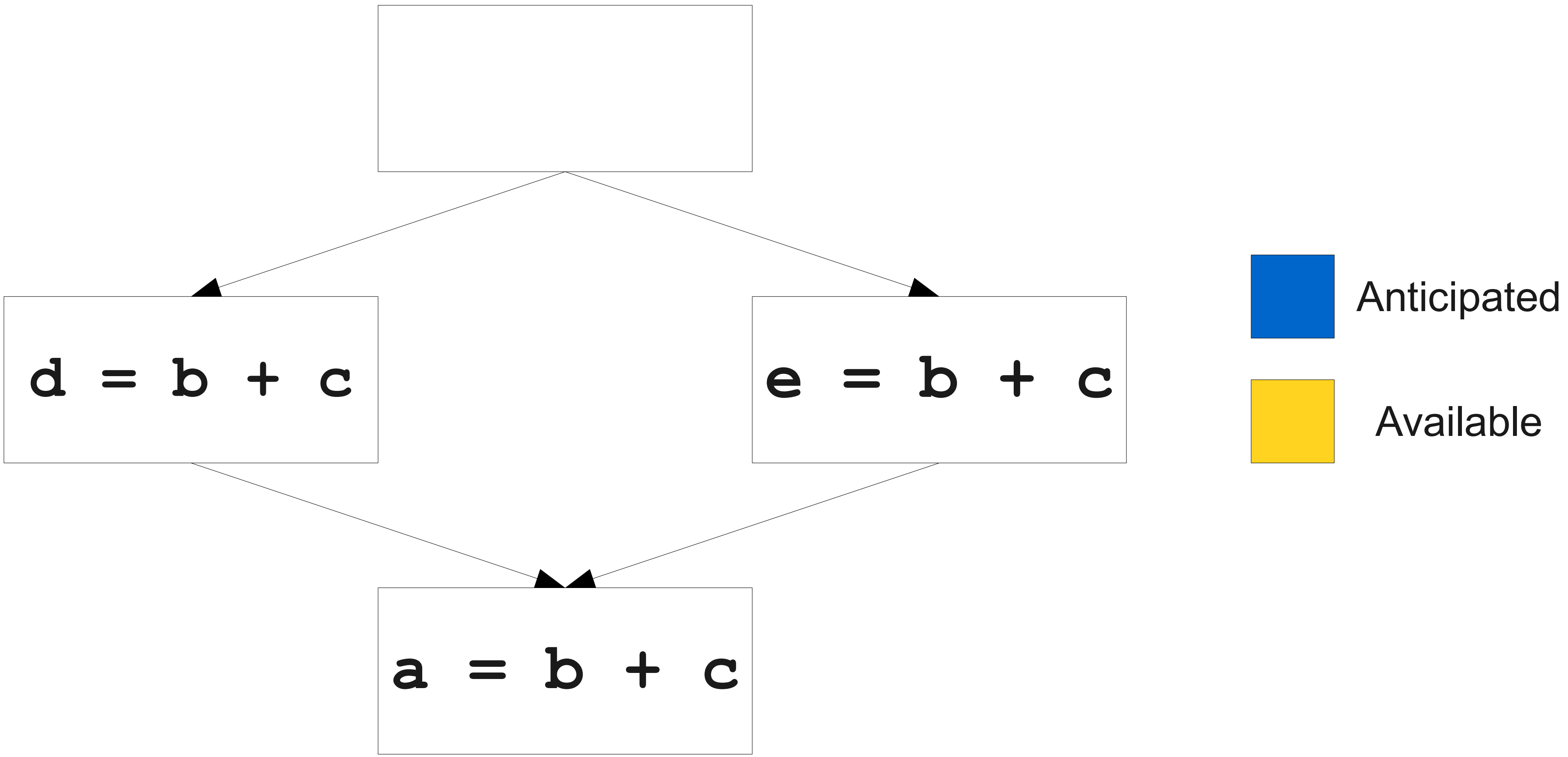


Eliminating Redundancy II

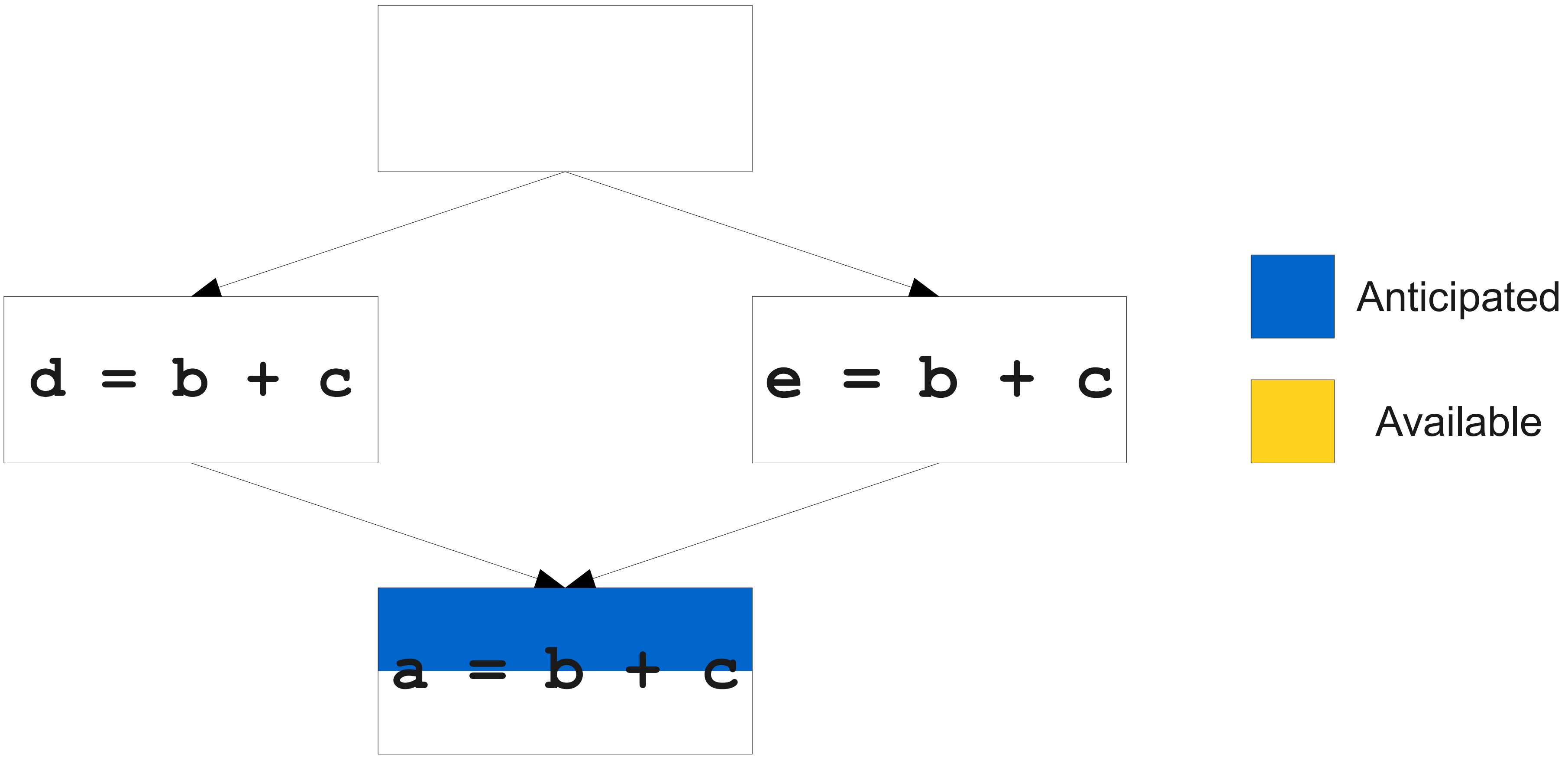
Eliminating Redundancy II



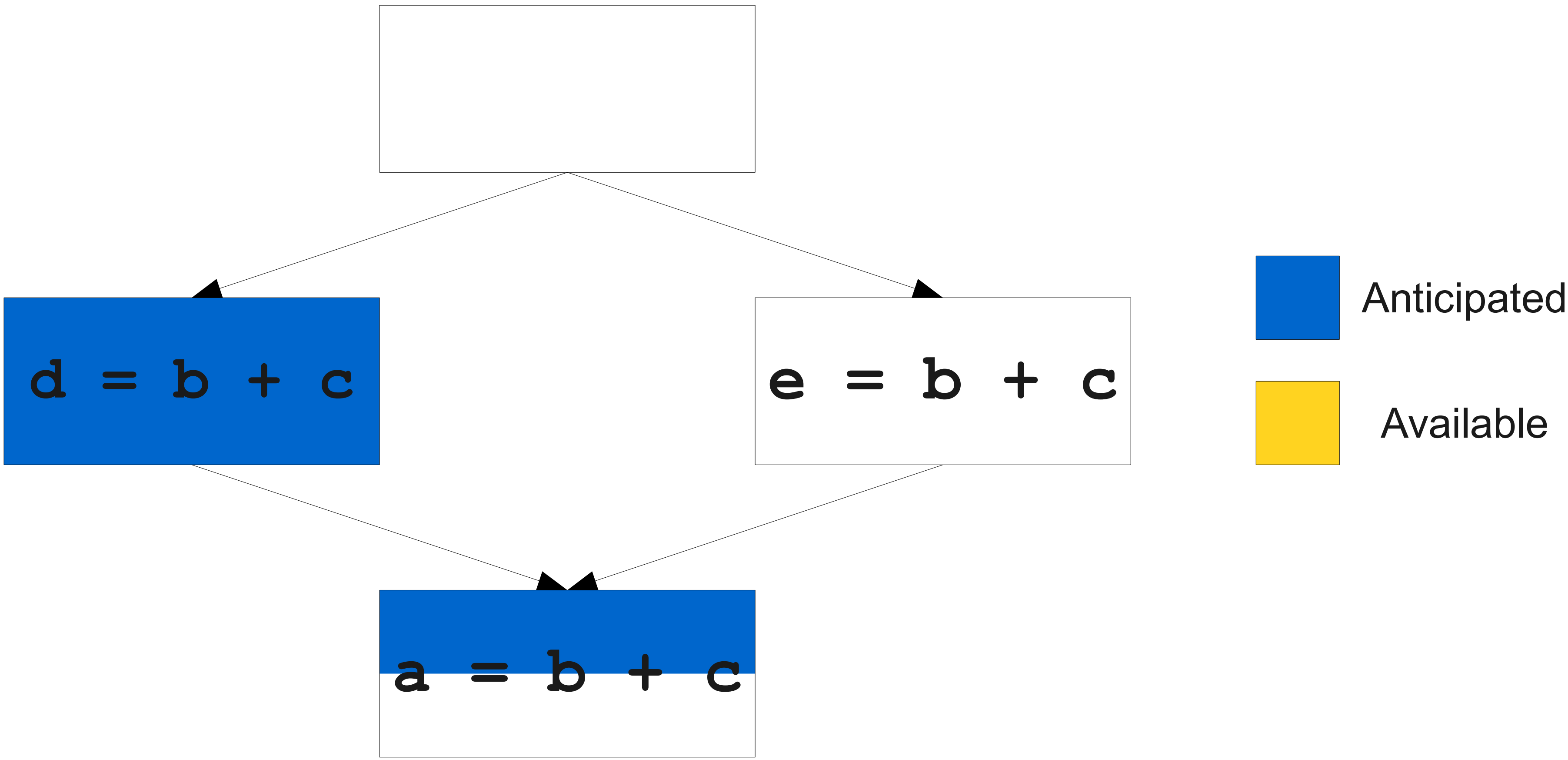
Eliminating Redundancy II



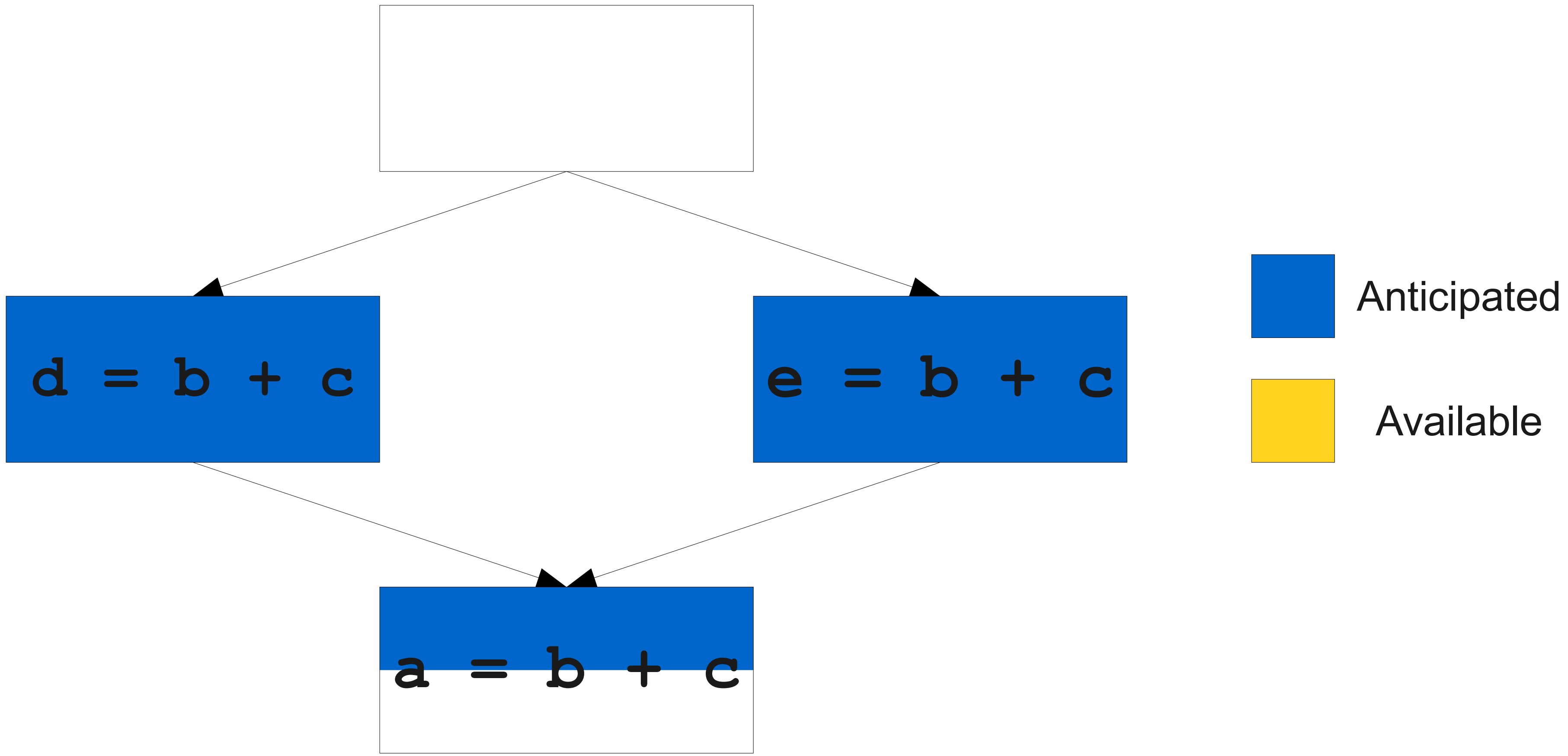
Eliminating Redundancy II



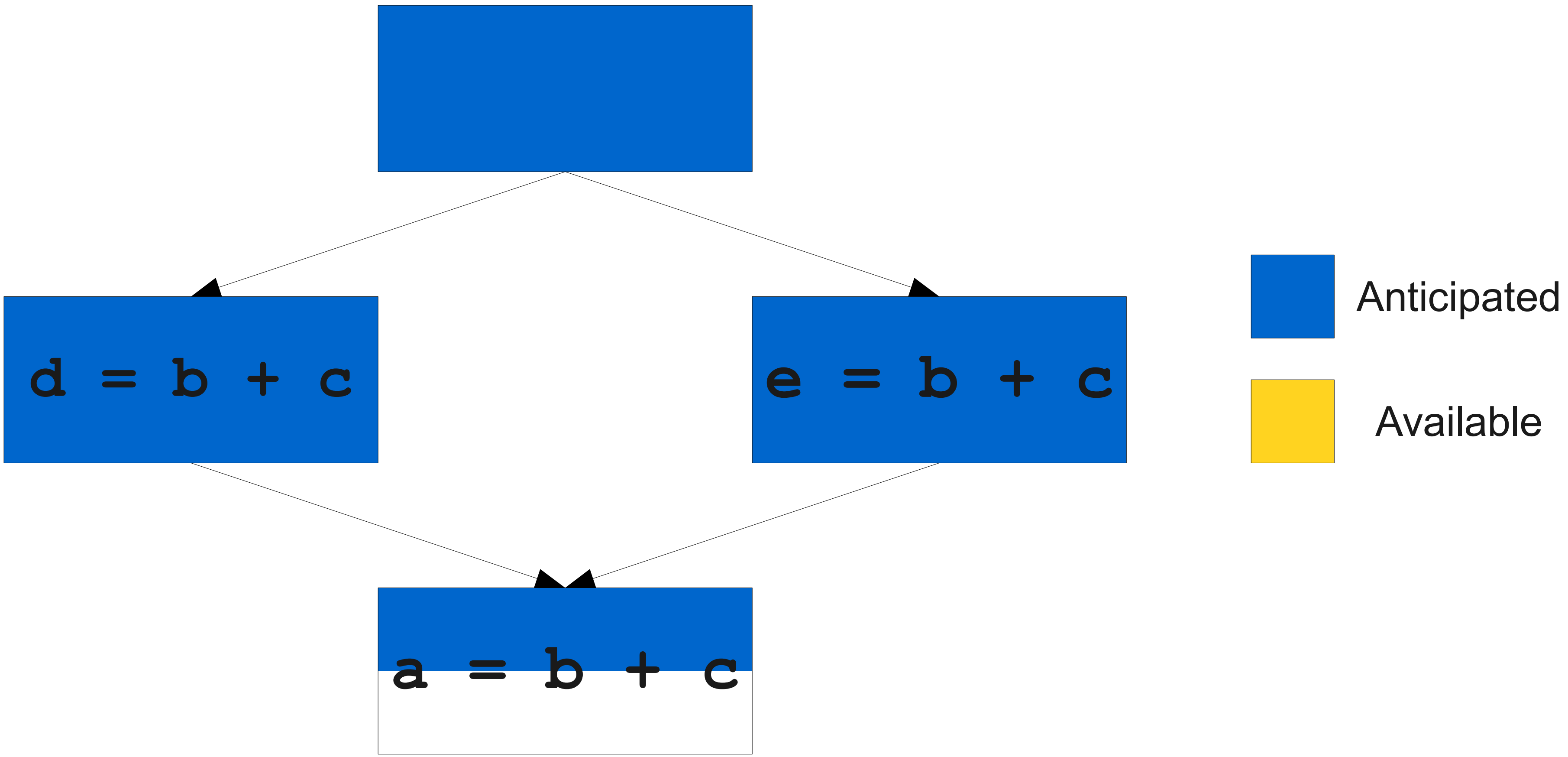
Eliminating Redundancy II



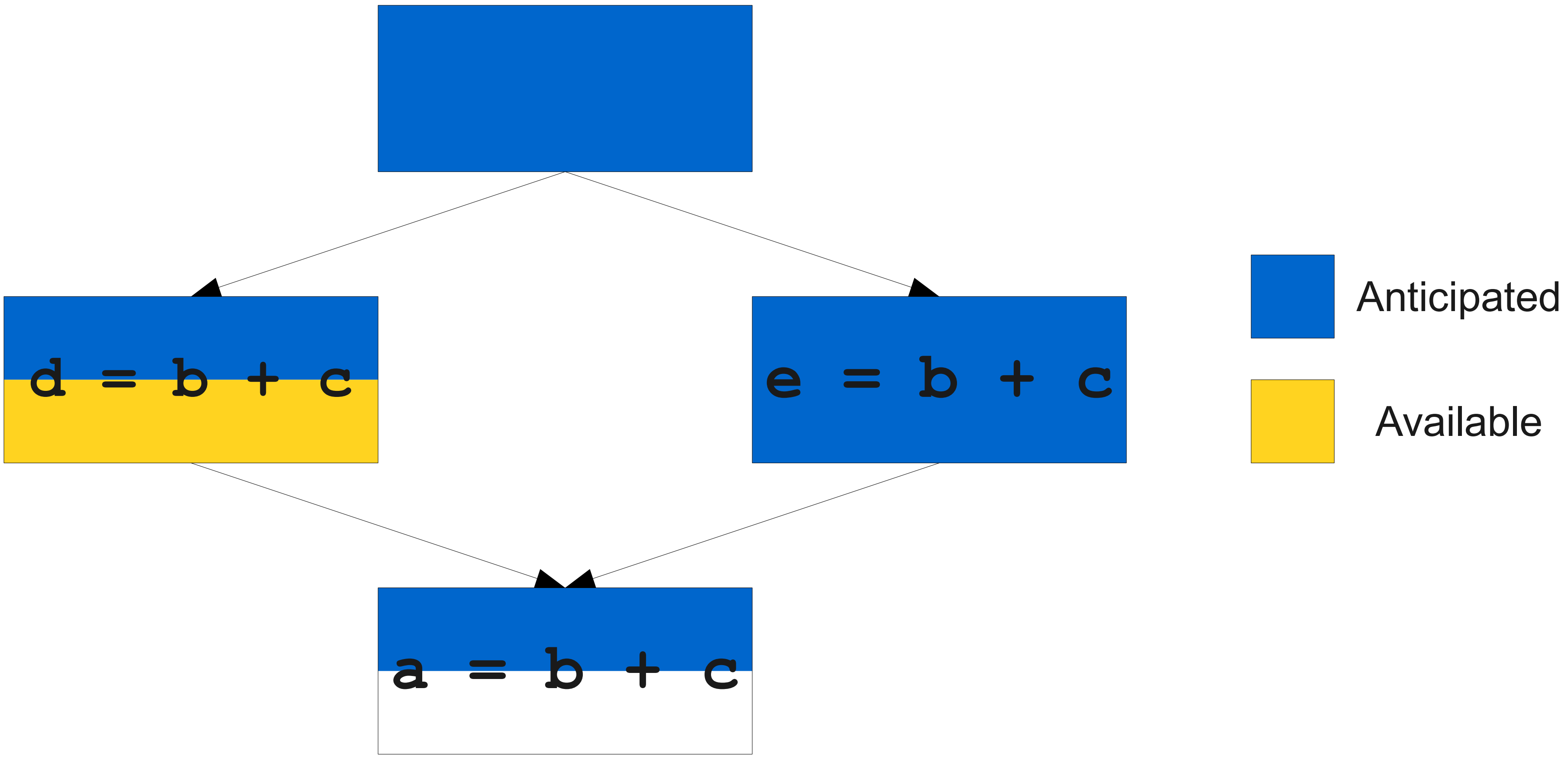
Eliminating Redundancy II



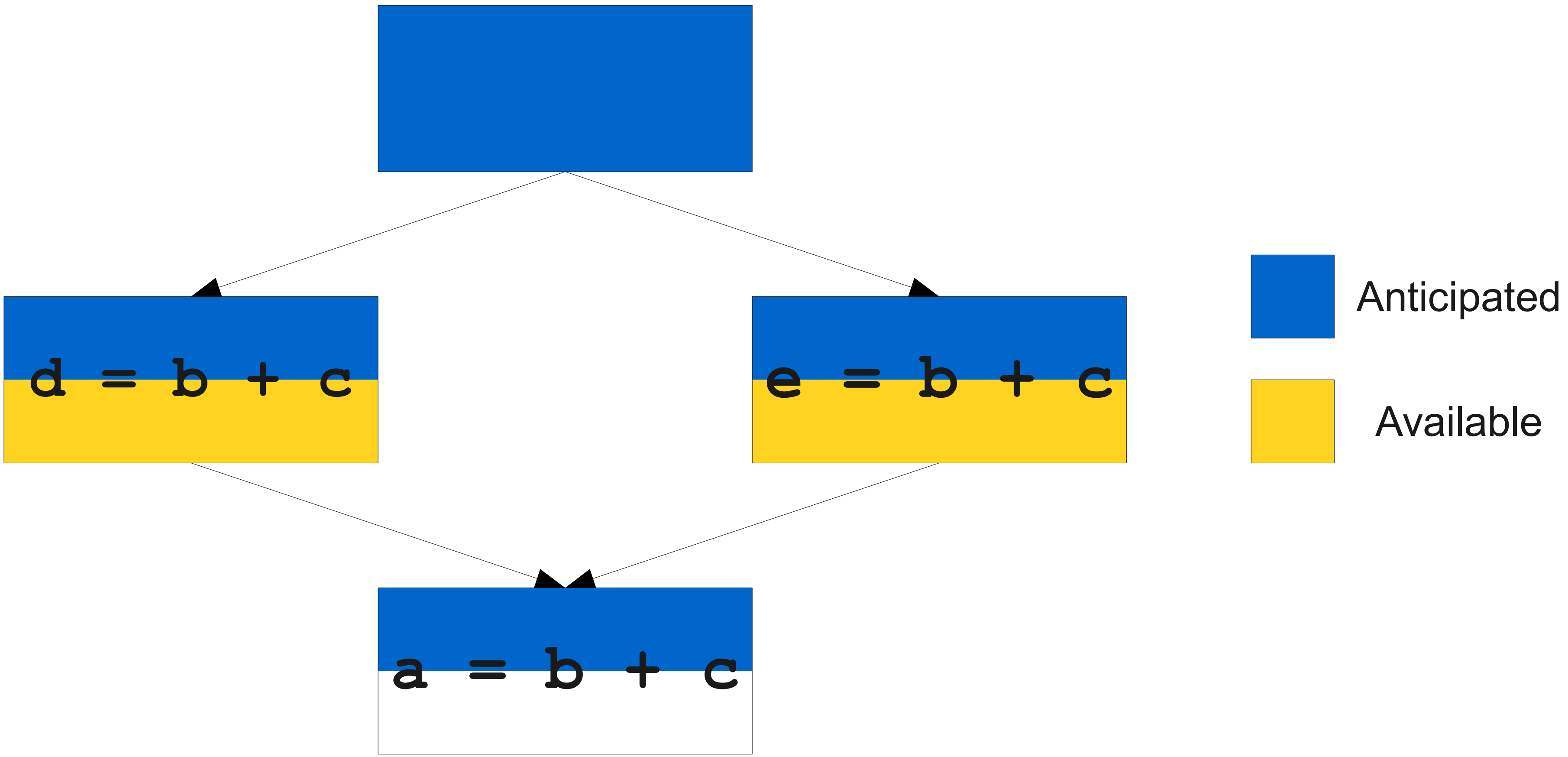
Eliminating Redundancy II



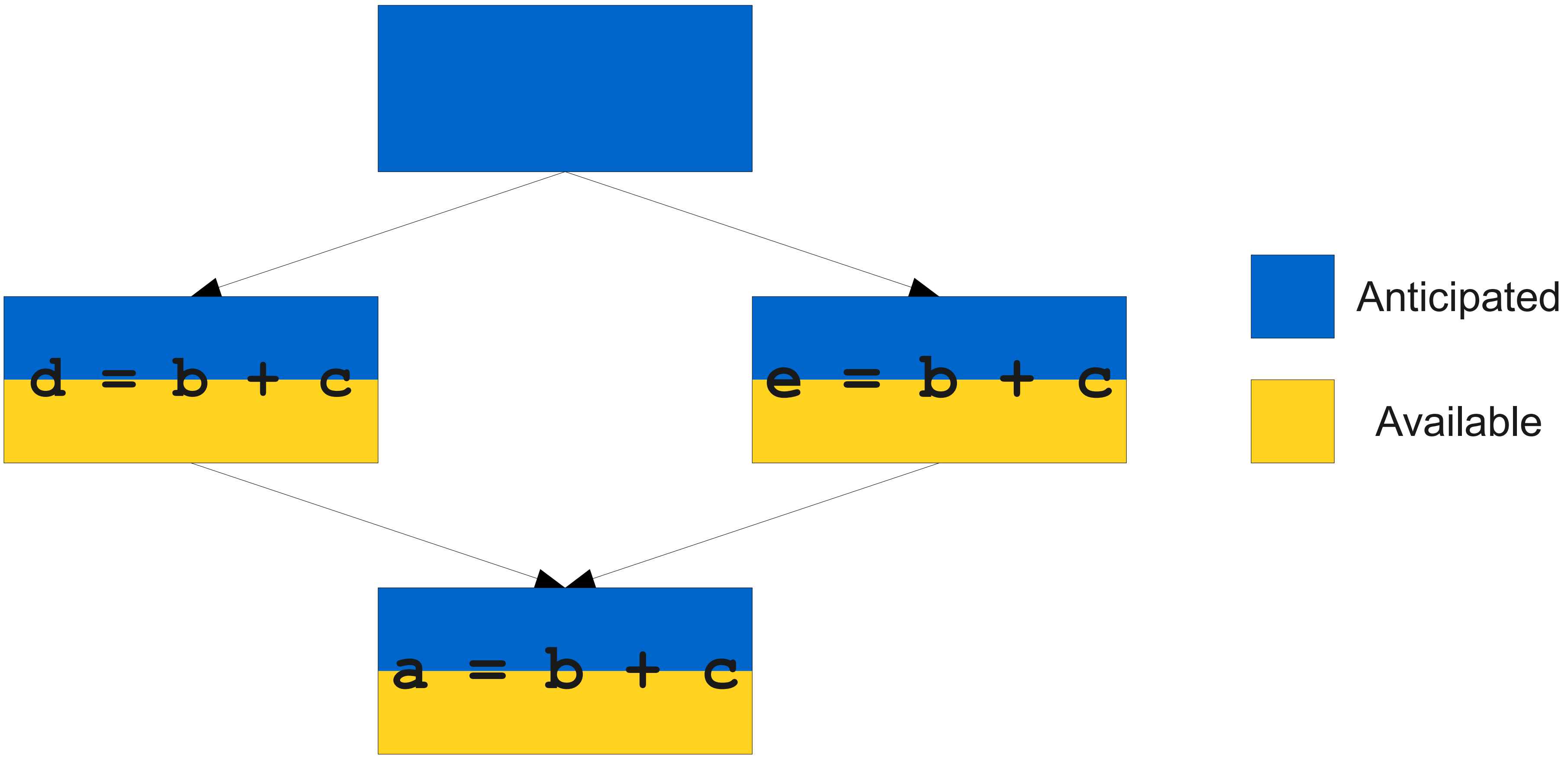
Eliminating Redundancy II



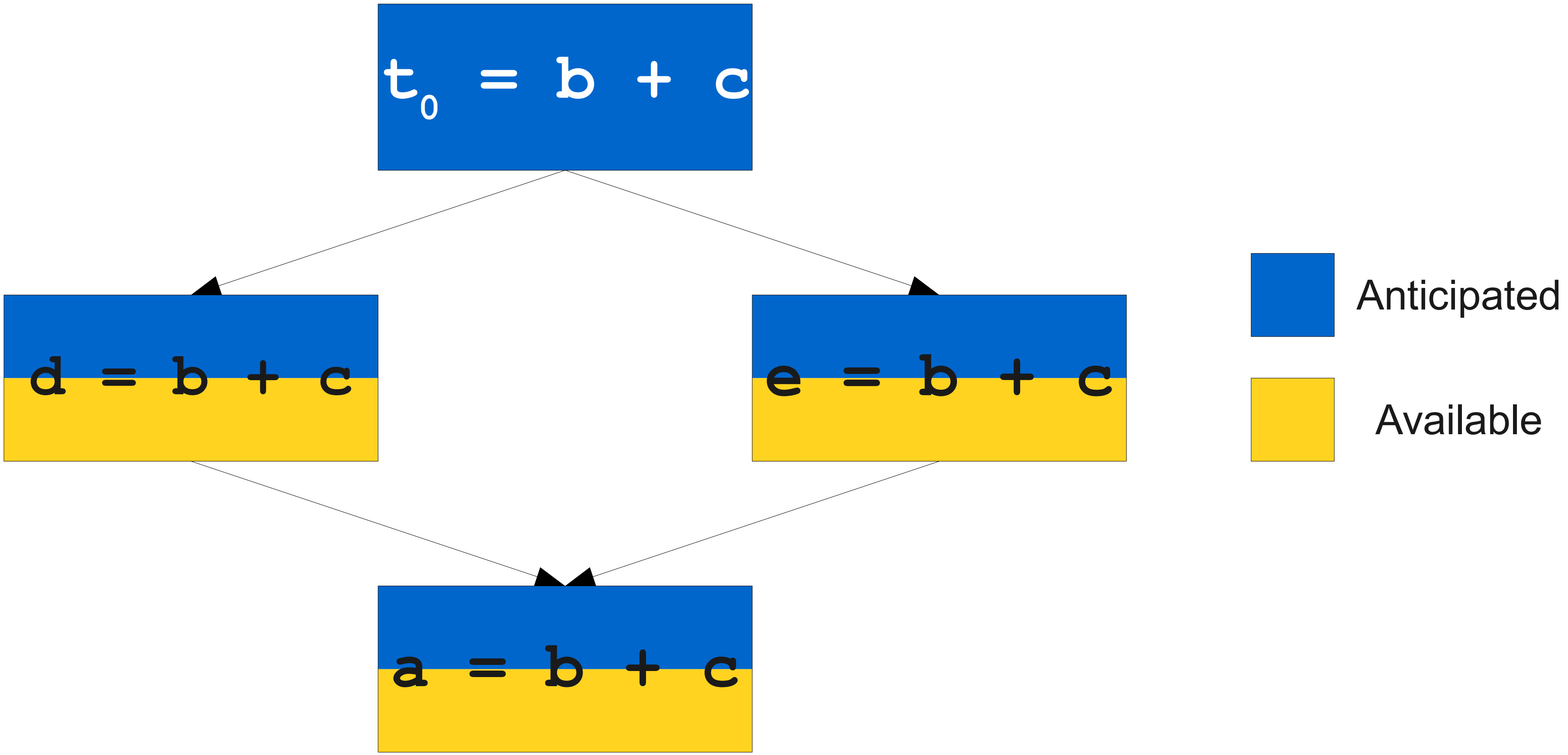
Eliminating Redundancy II



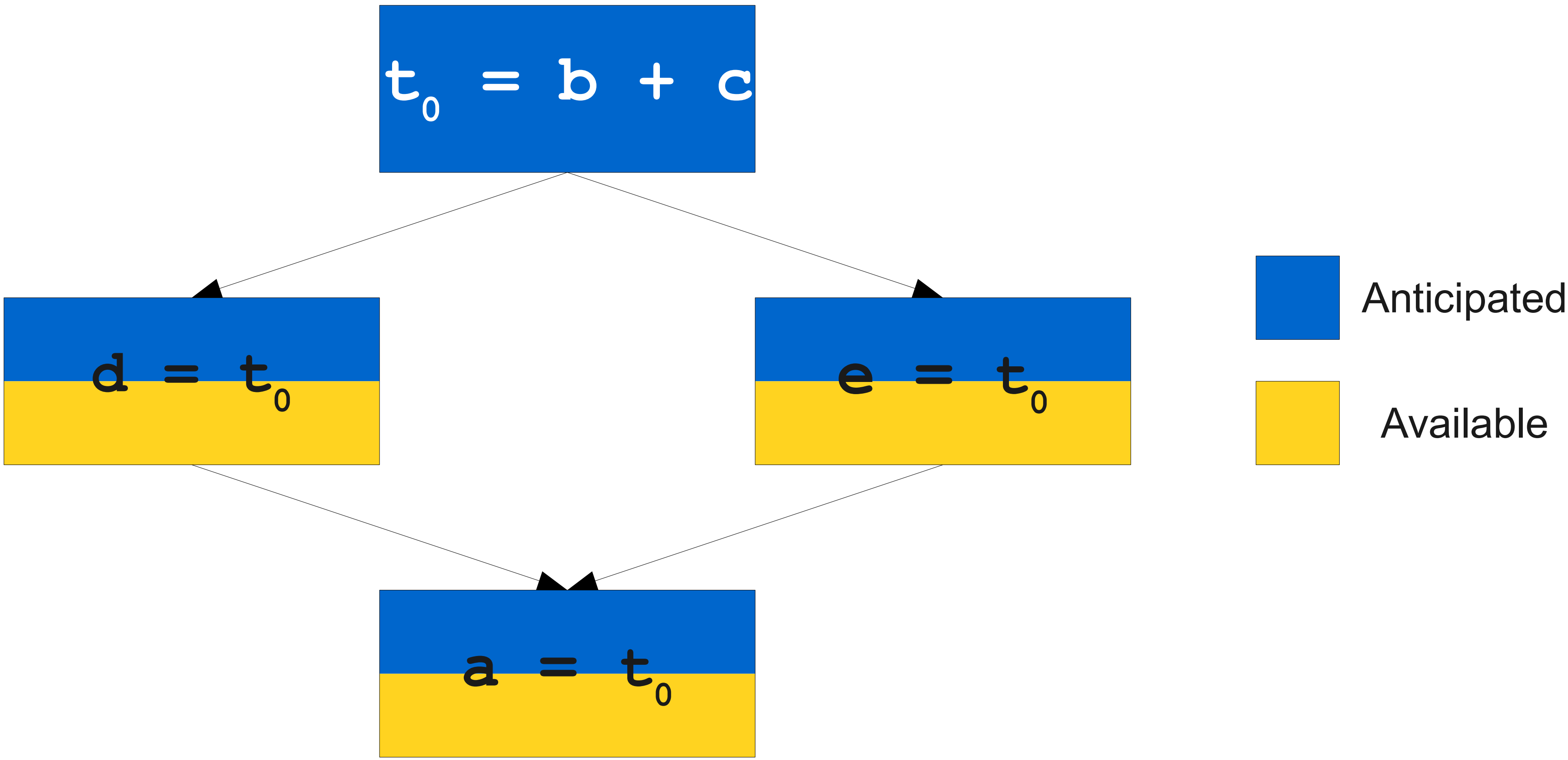
Eliminating Redundancy II



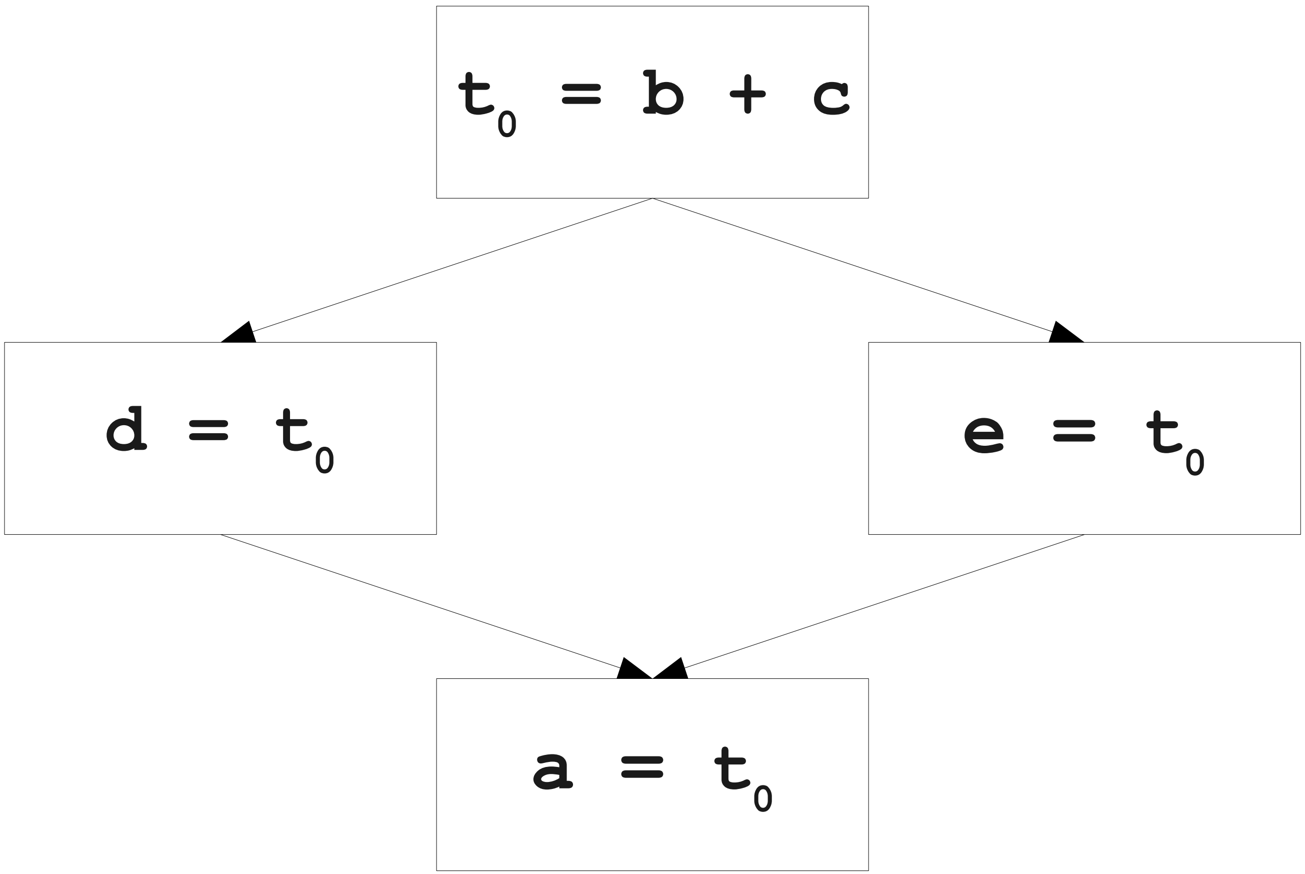
Eliminating Redundancy II



Eliminating Redundancy II

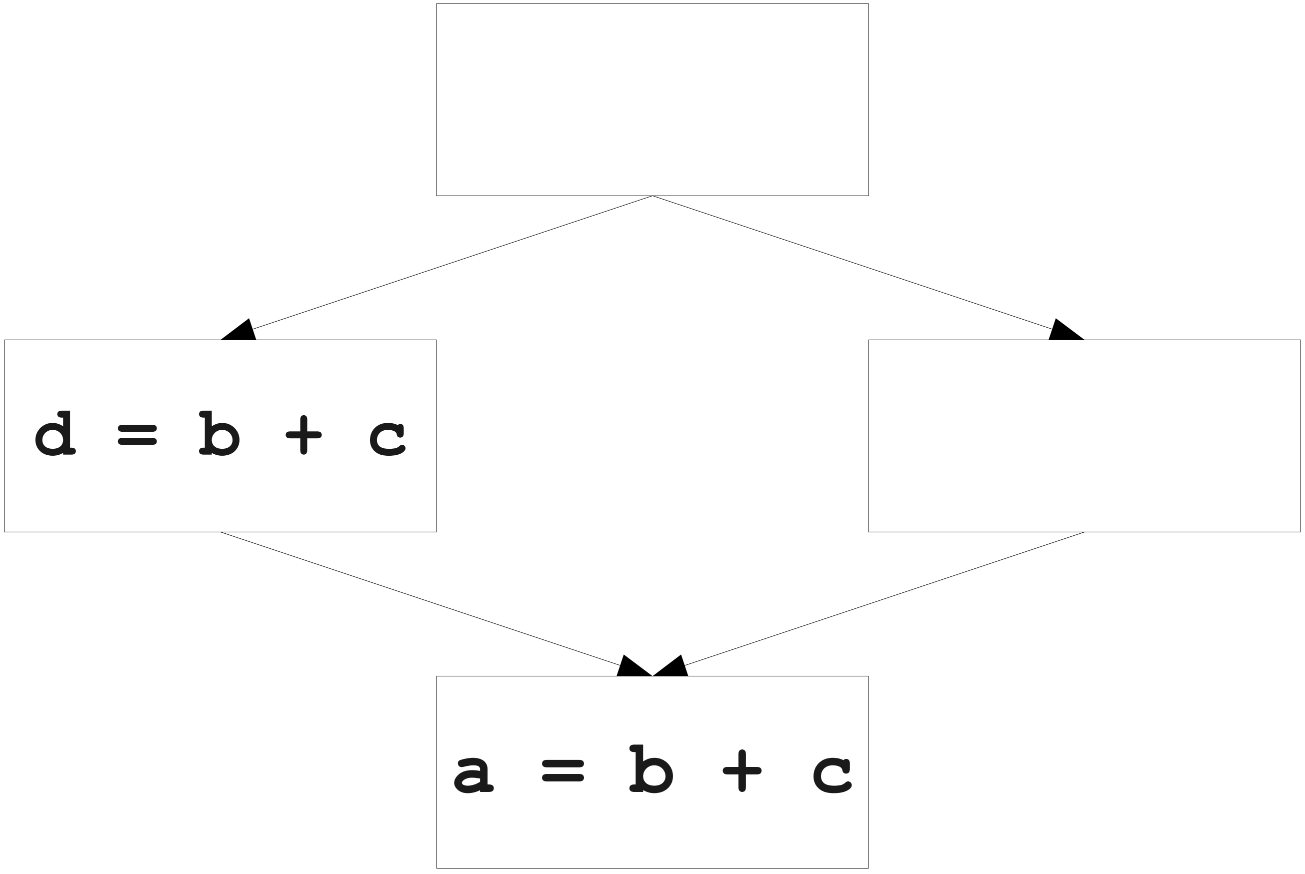


Eliminating Redundancy II

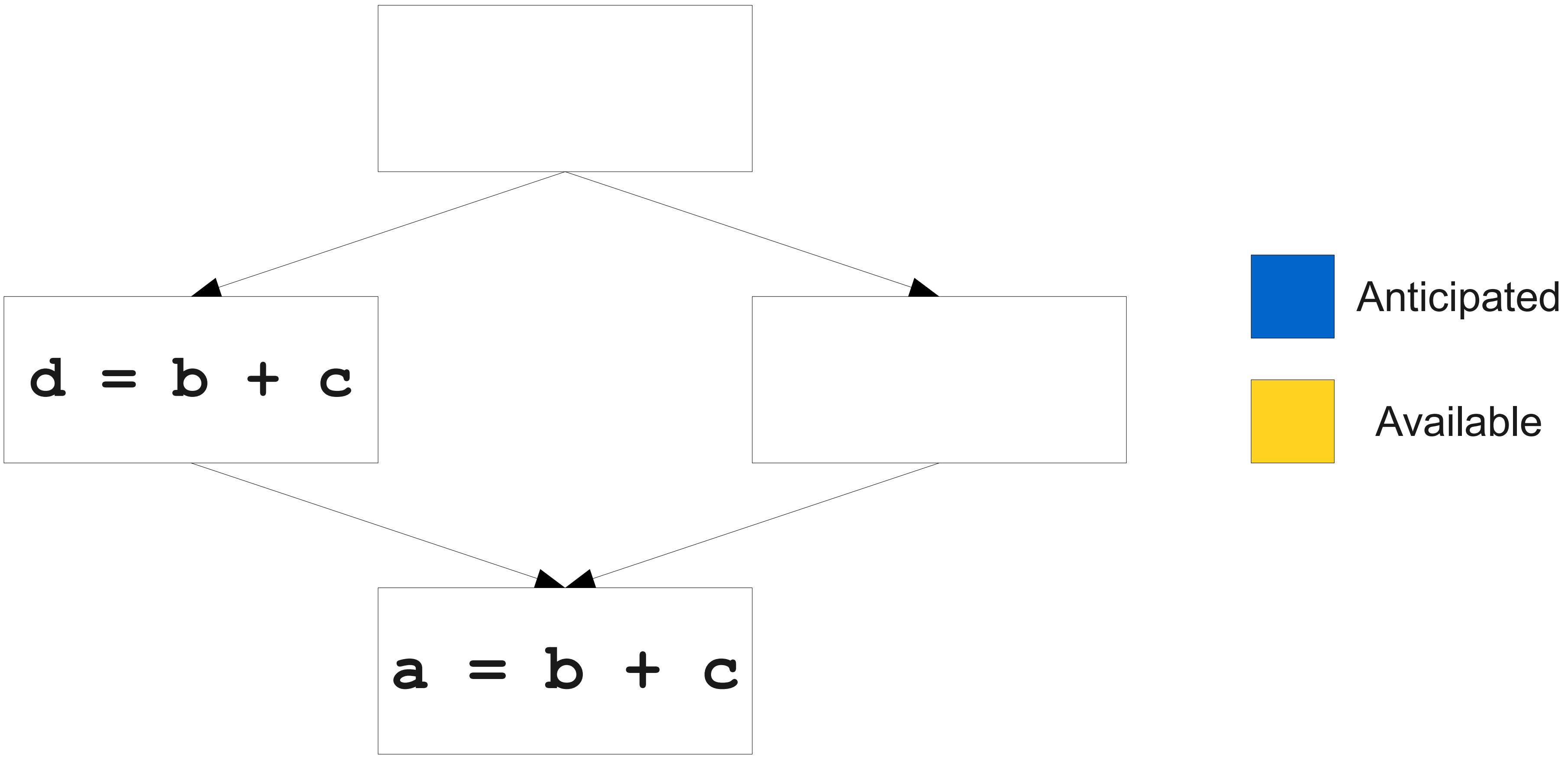


Eliminating Redundancy III

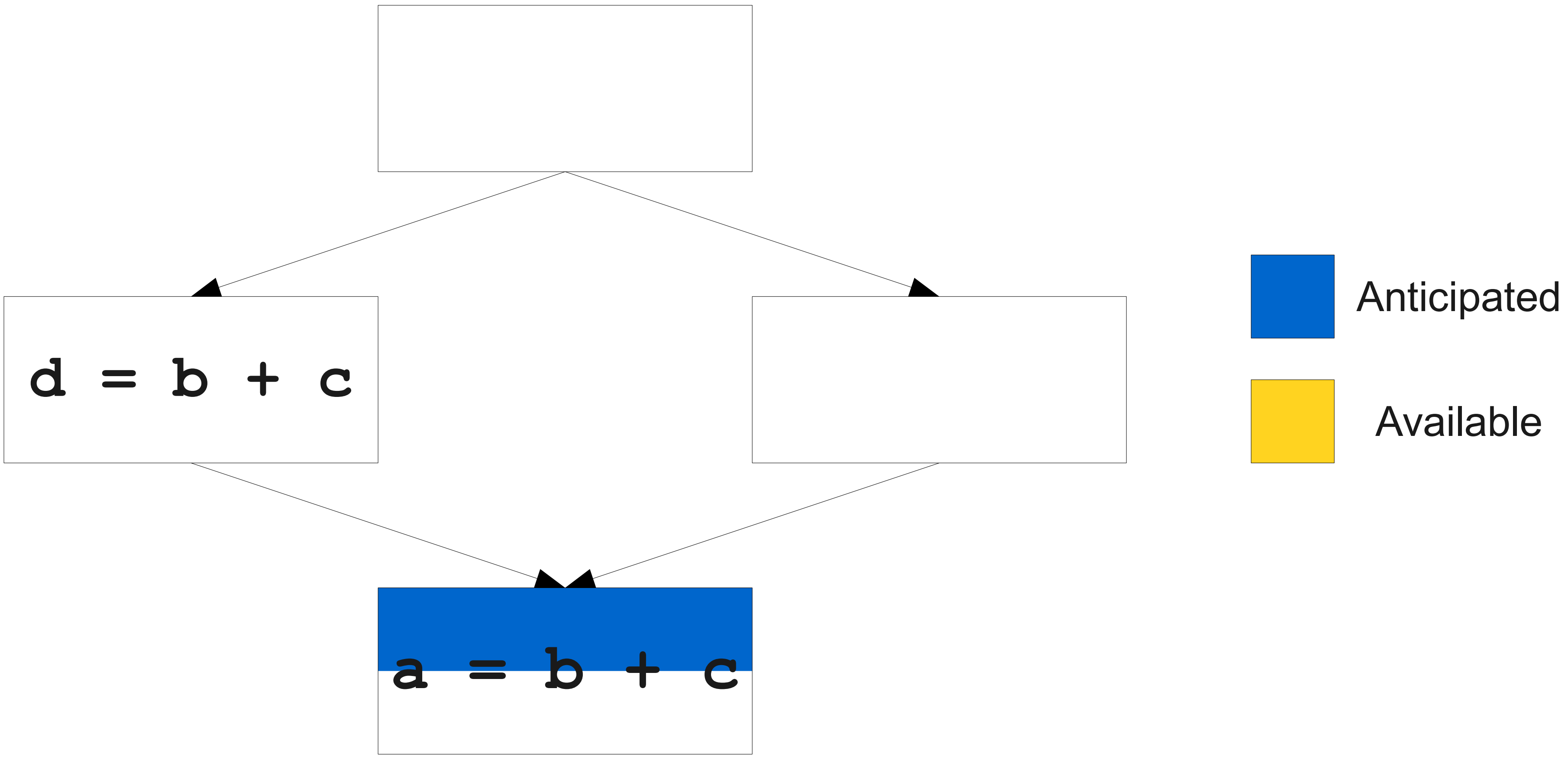
Eliminating Redundancy III



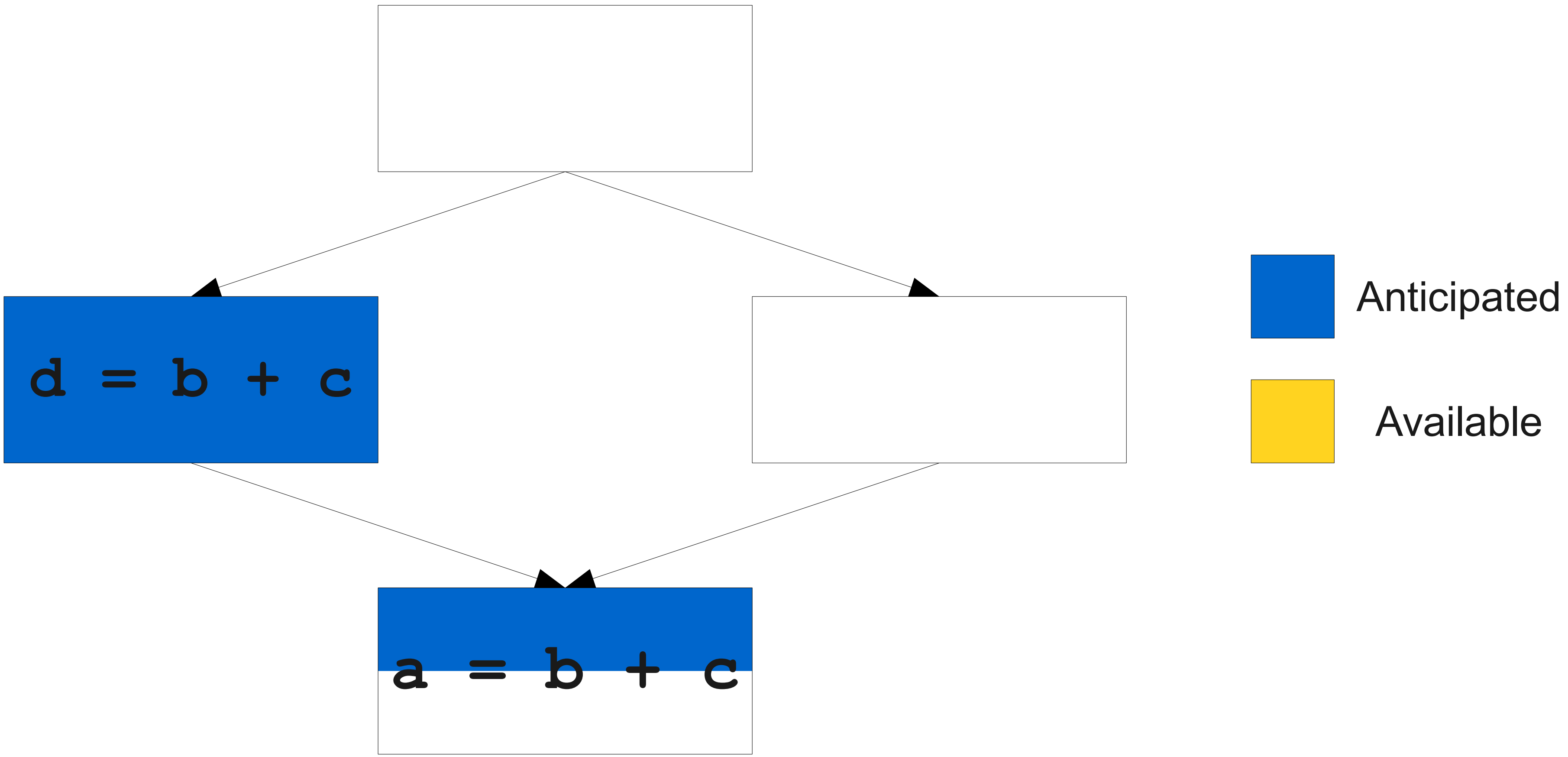
Eliminating Redundancy III



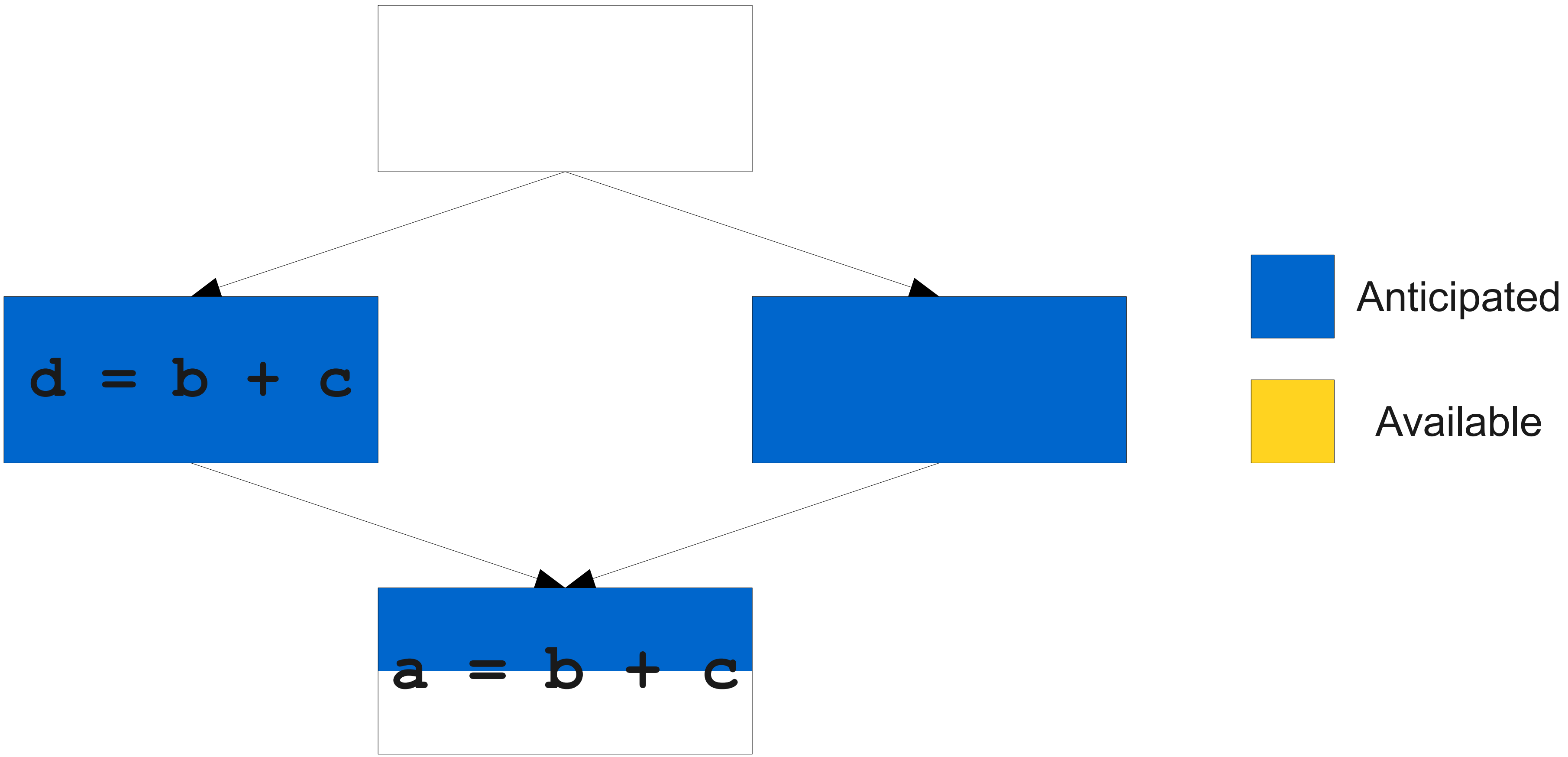
Eliminating Redundancy III



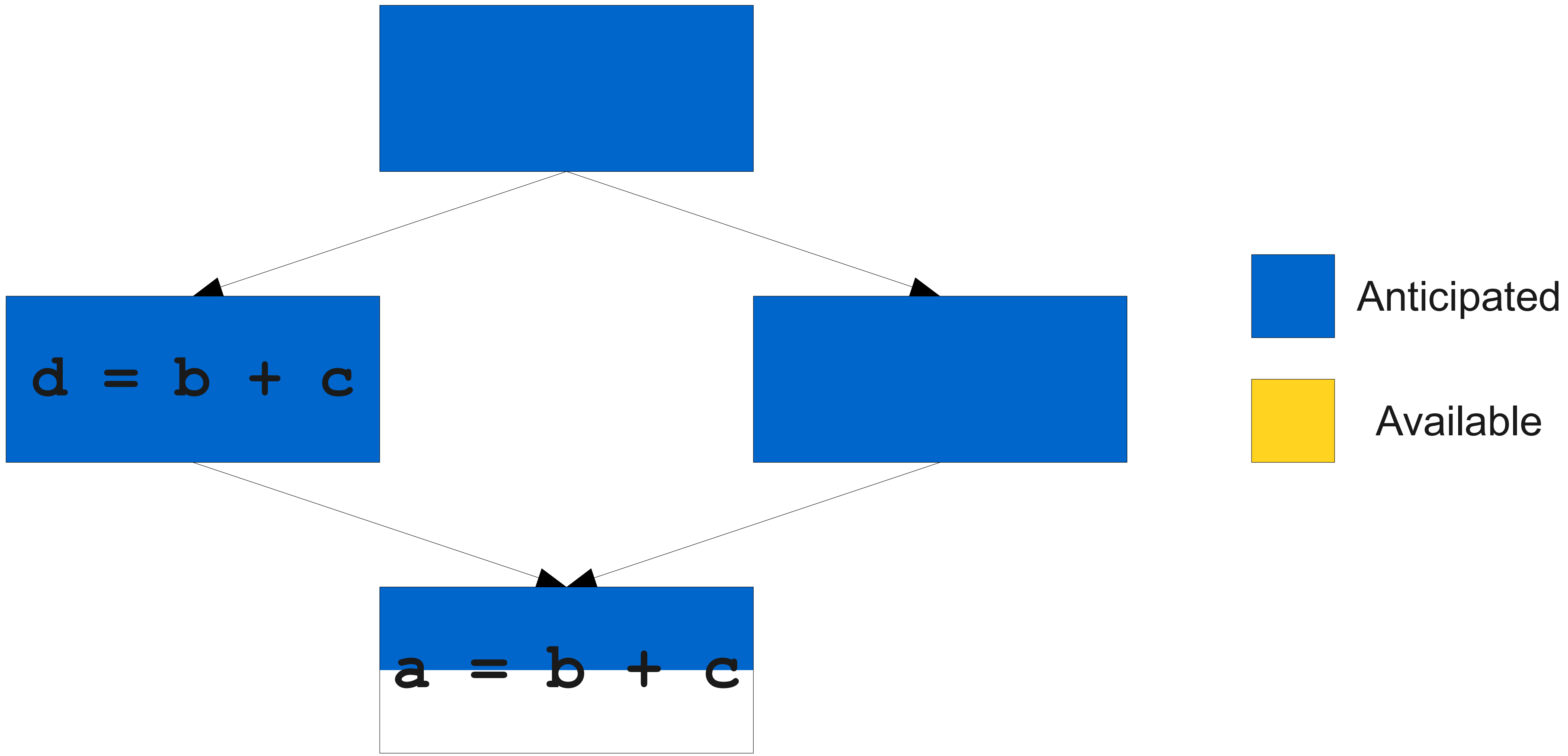
Eliminating Redundancy III



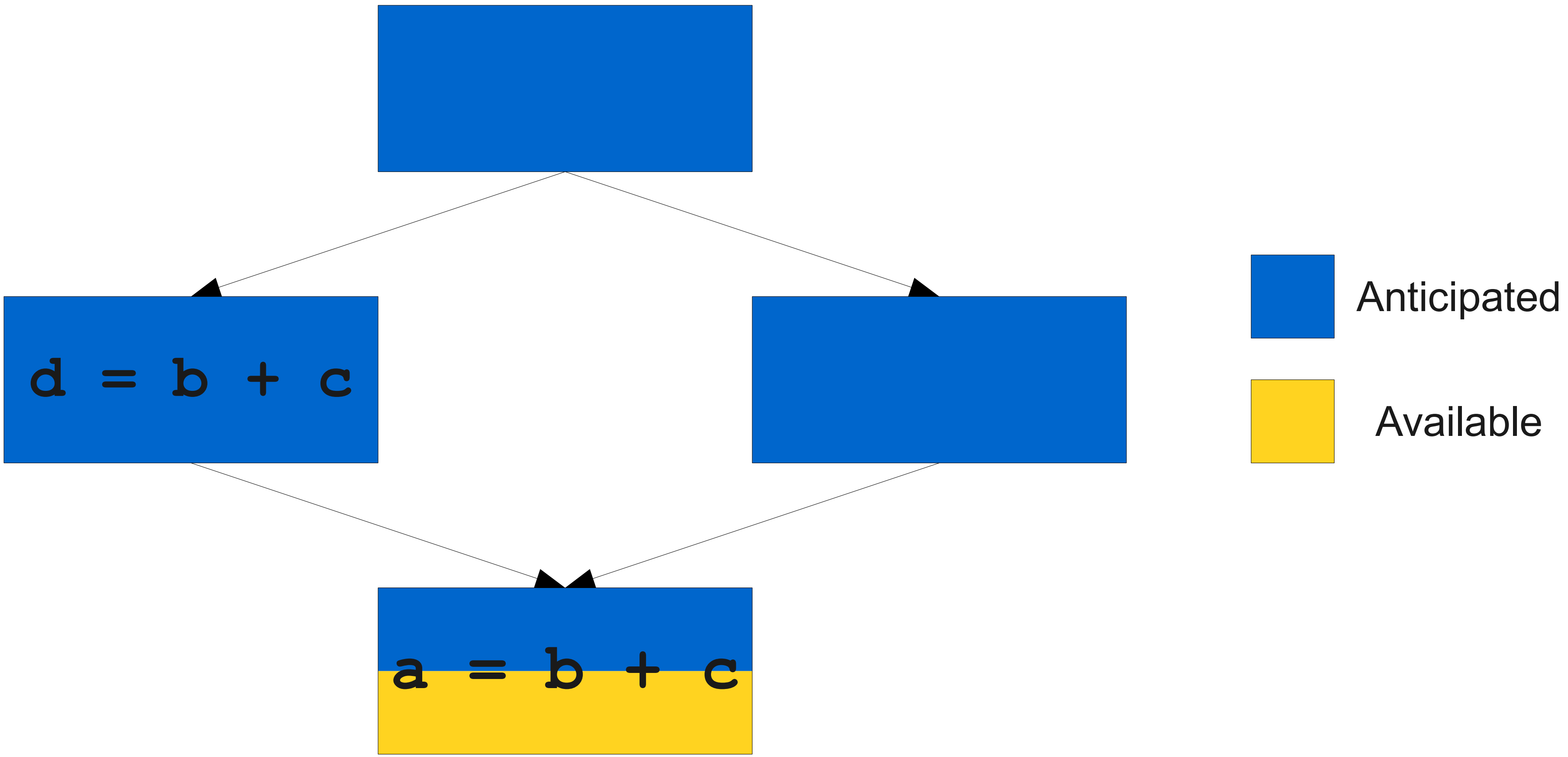
Eliminating Redundancy III



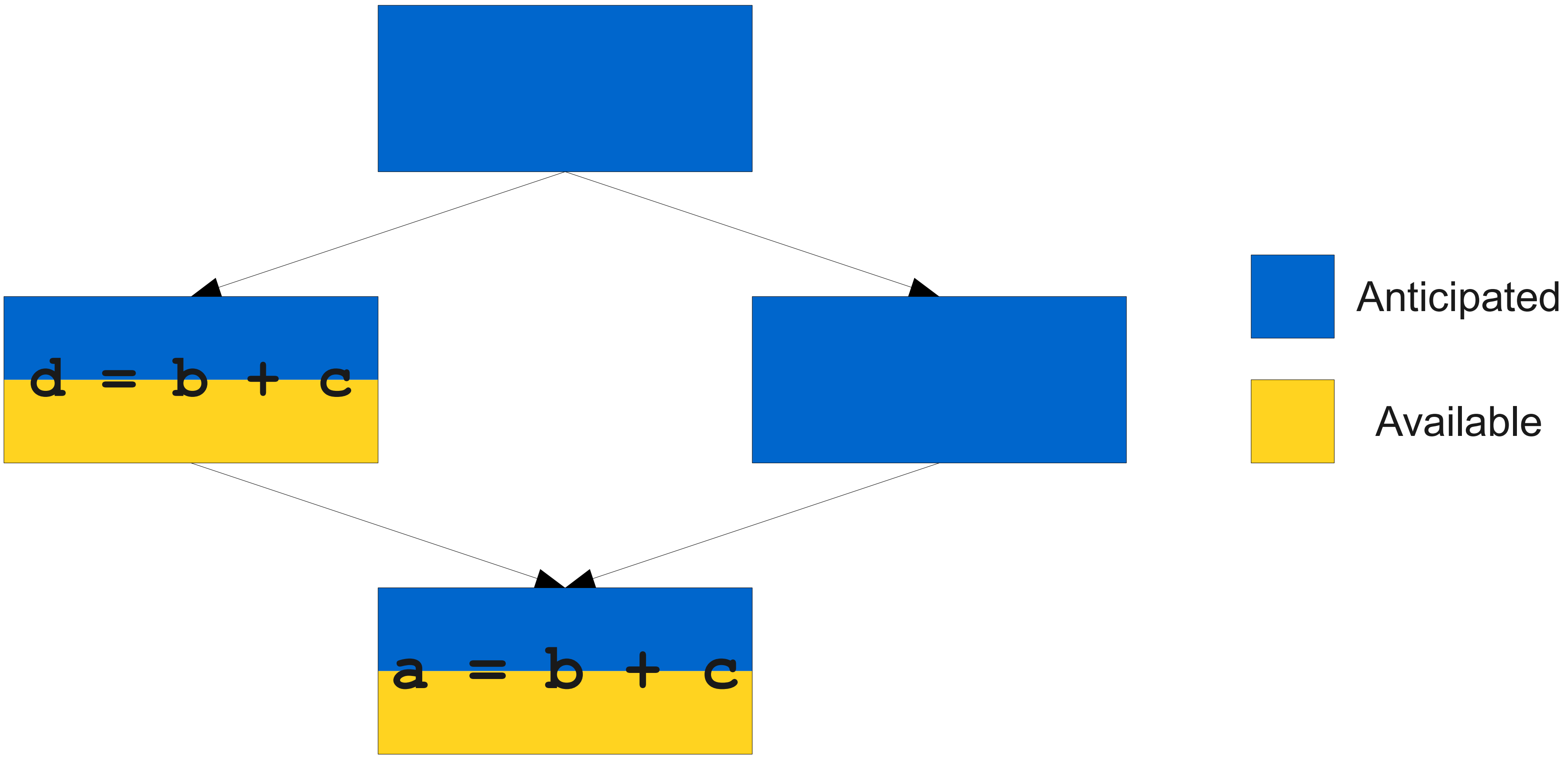
Eliminating Redundancy III



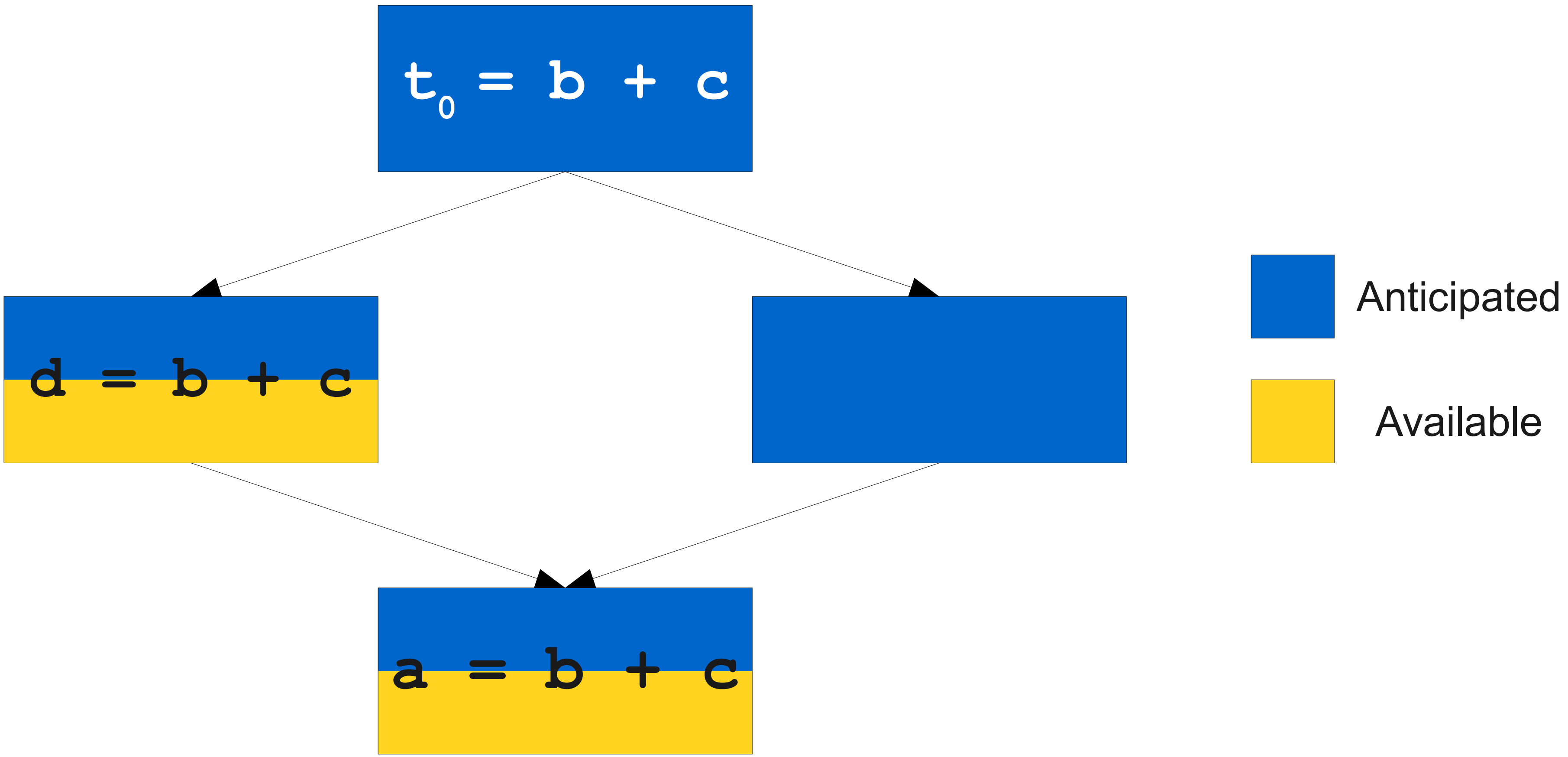
Eliminating Redundancy III



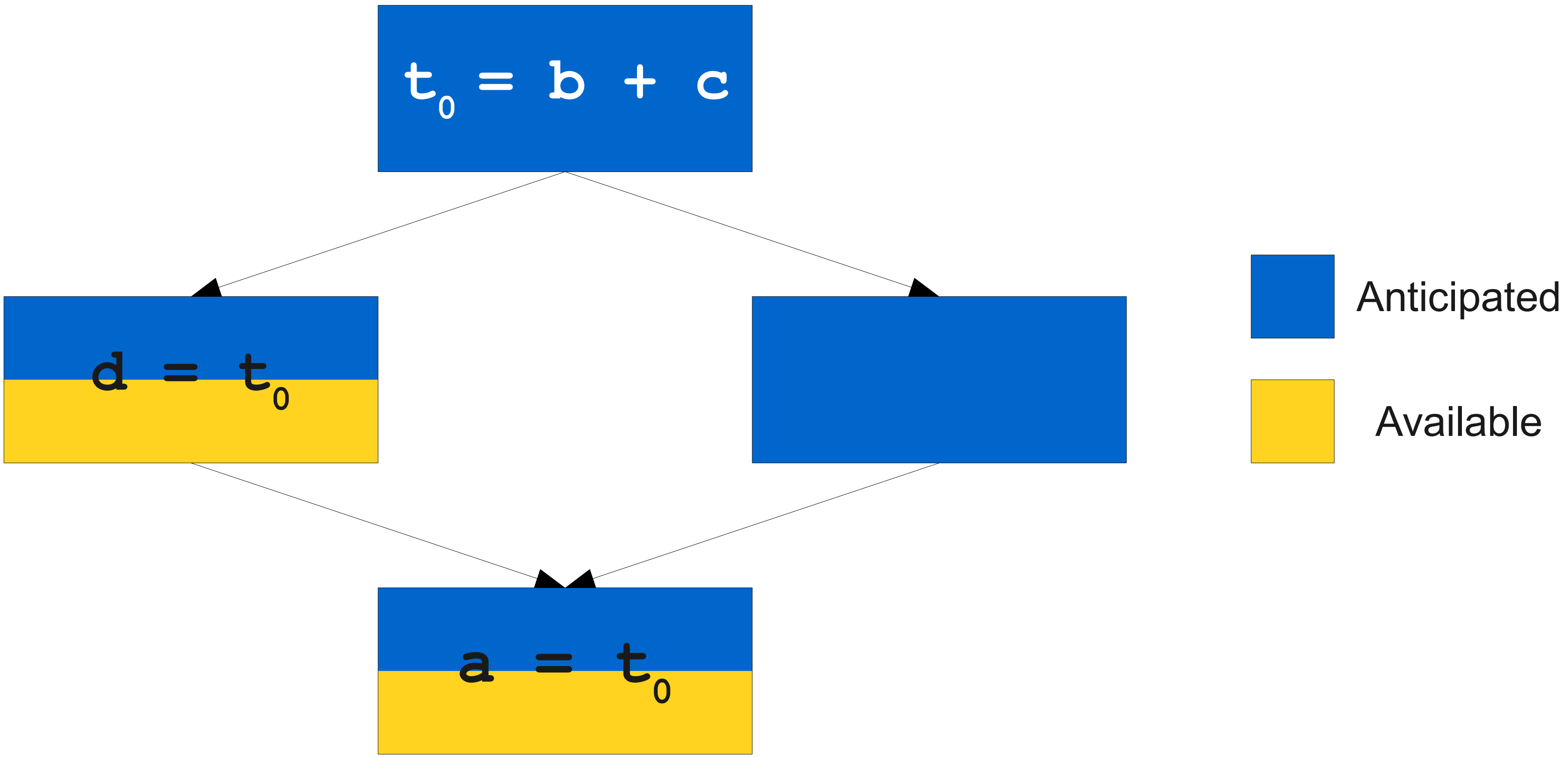
Eliminating Redundancy III



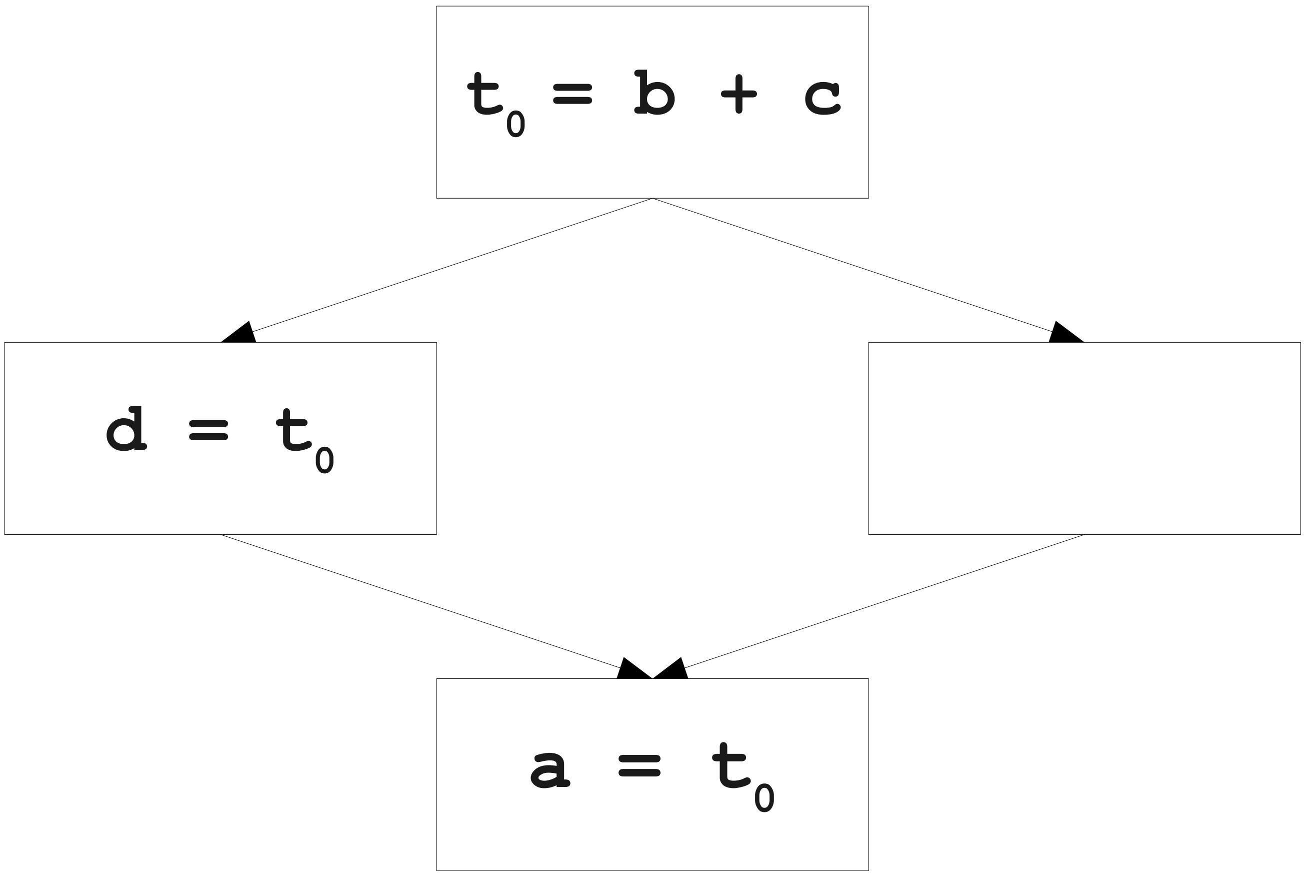
Eliminating Redundancy III



Eliminating Redundancy III



Eliminating Redundancy III



In Practice

- Partial-redundancy elimination is typically implemented using **four** dataflow analyses.
- A bit more complex than what we covered:
  - How to avoid keeping expressions around too long?
  - How to avoid introducing unnecessary temporaries?
- See Dragon Book, Ch. 9.5 for more precise details.

Next Time

- **Code Generation**
  - The machine at a glance.
  - Register allocation.