Global Optimization

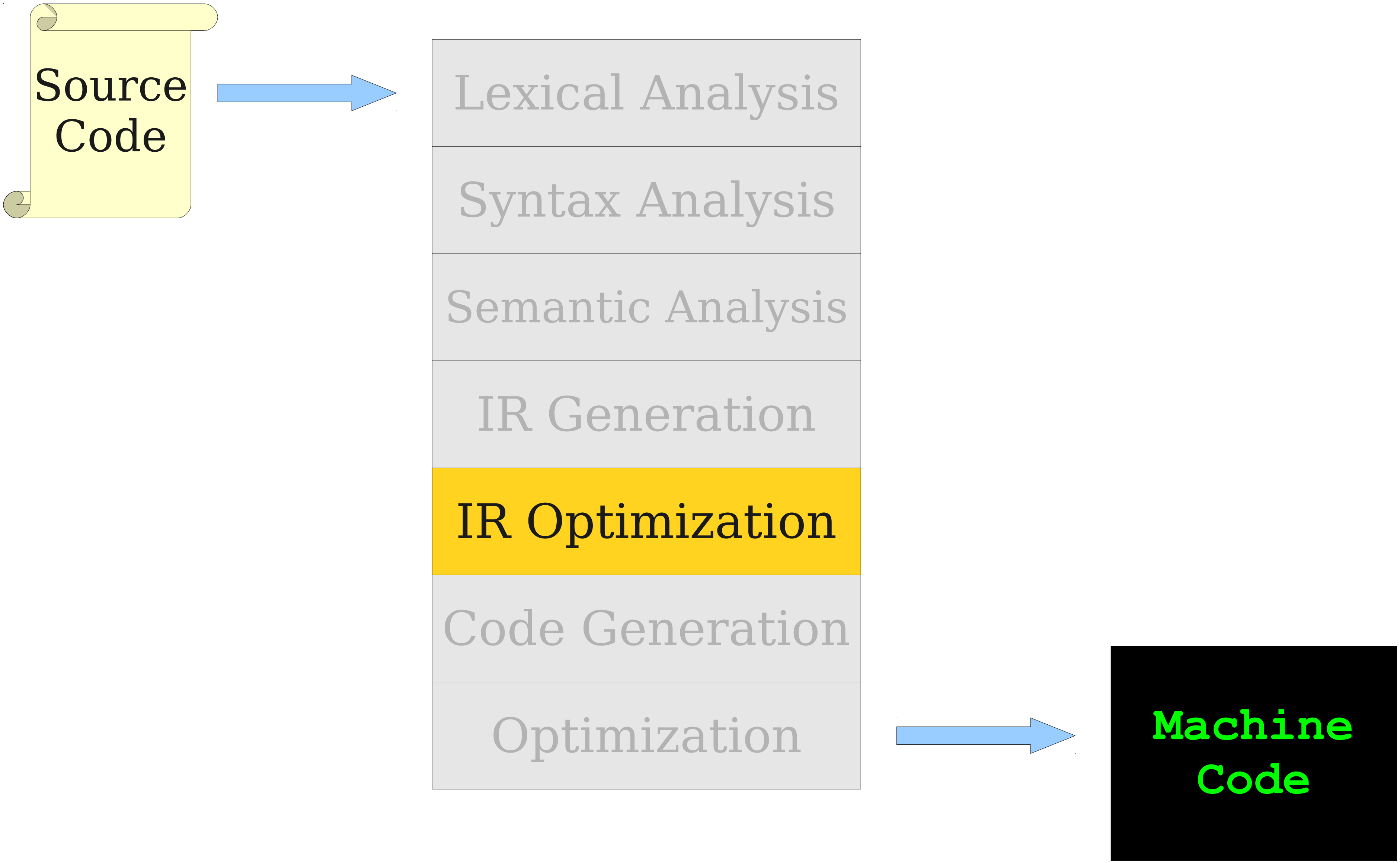
Part II

Announcements

- Programming Project 4 due Saturday, August 18 at 11:30AM.
    - OH today and tomorrow.
    - Ask questions via email!
    - Ask questions via Piazzza!
    - **No late submissions.**

Four Square!
5:30PM Thursday, Outside Gates

Where We Are


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

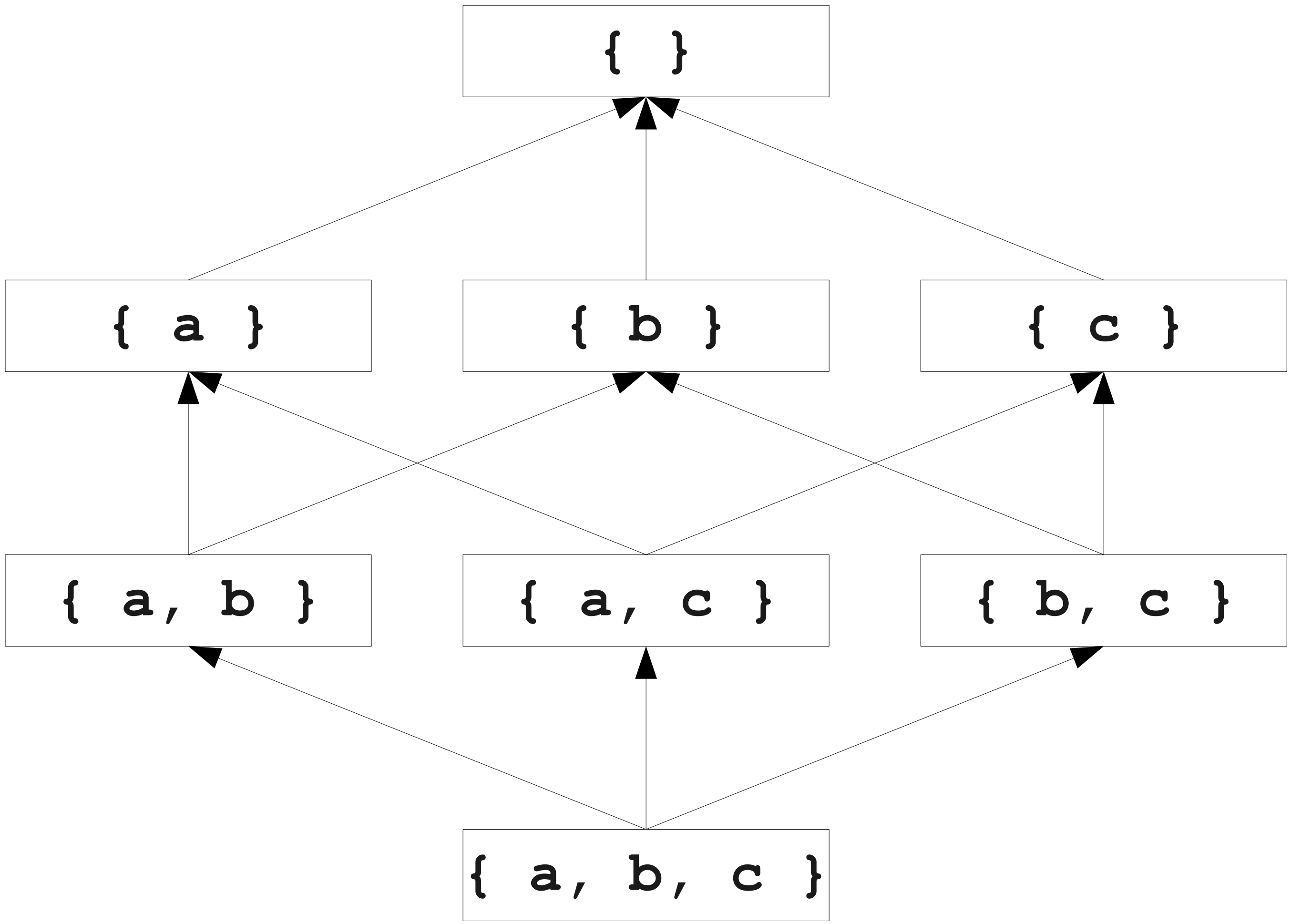
Review: Why Global Analysis is Hard

- 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.

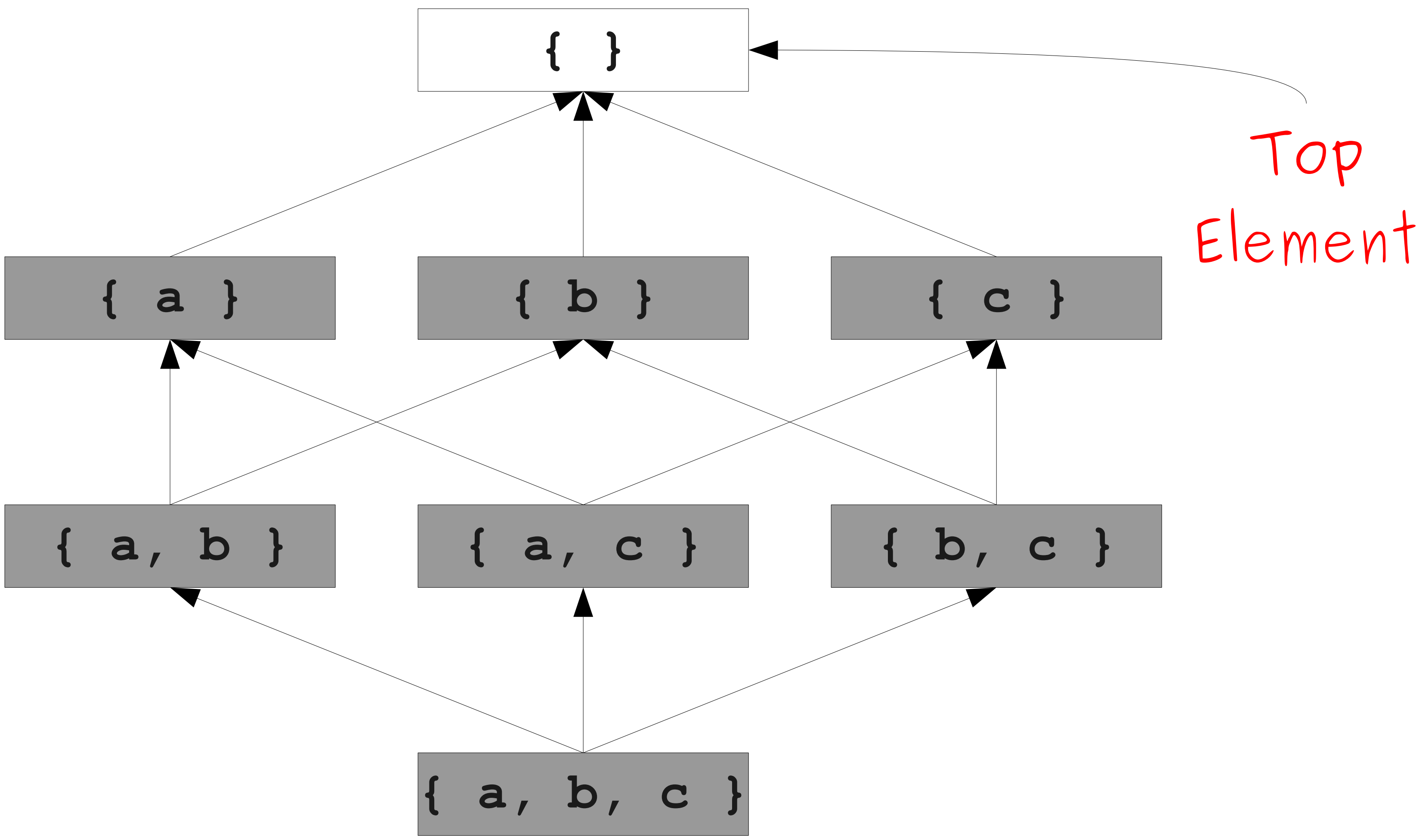
Review: Meet Semilattices

- 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.”

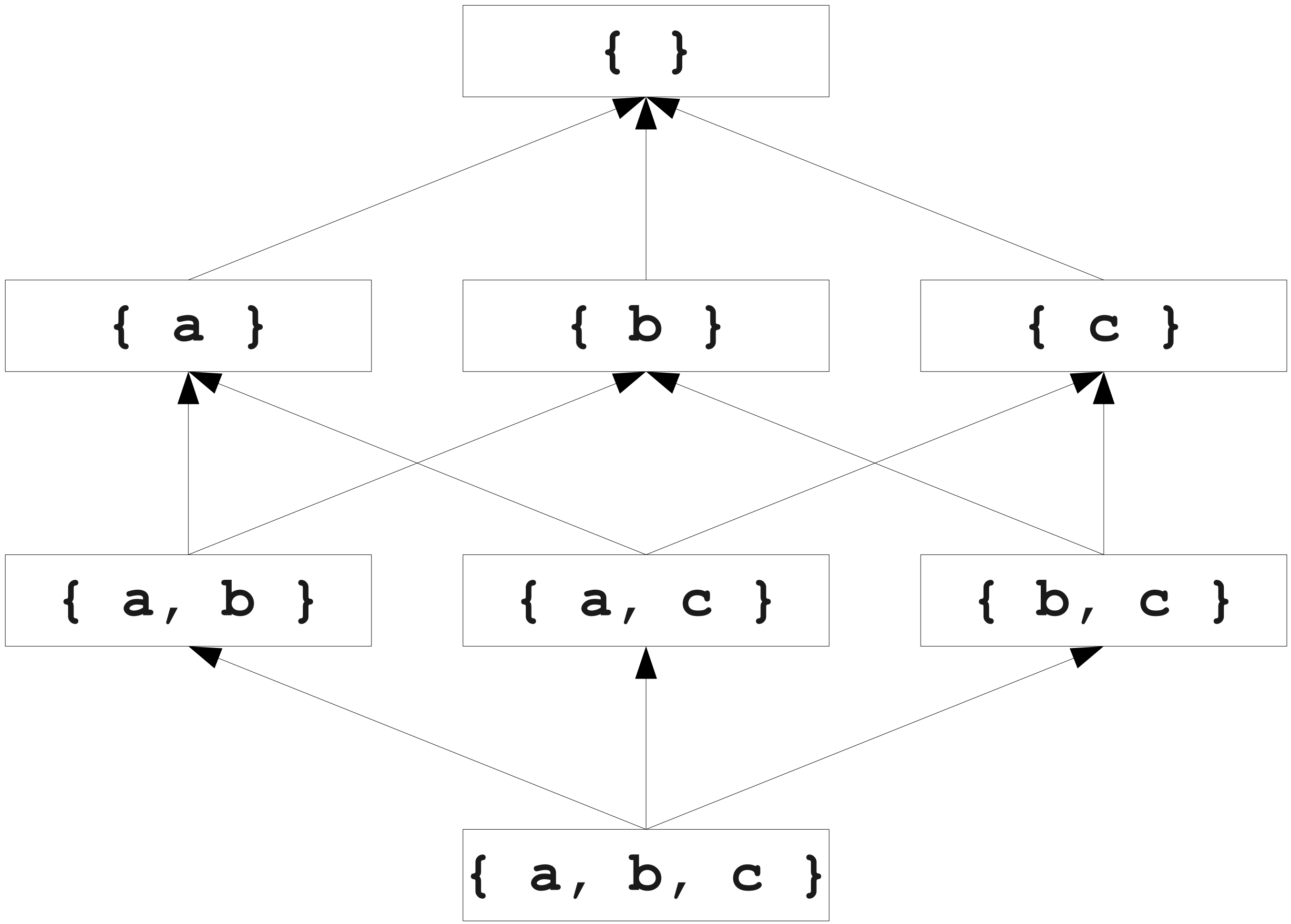
Meet Semilattices for Liveness



Meet Semilattices for Liveness



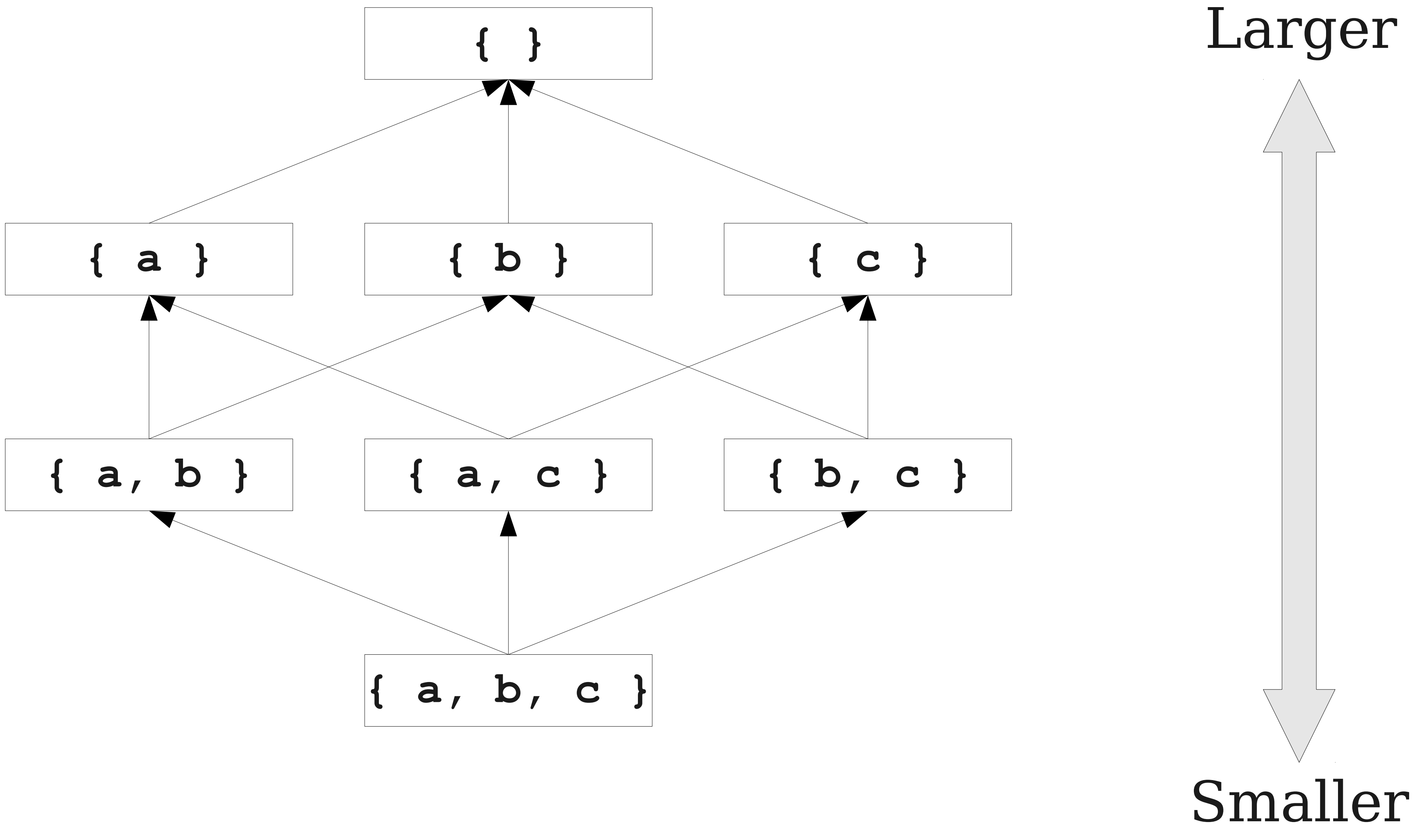
Meet Semilattices for Liveness



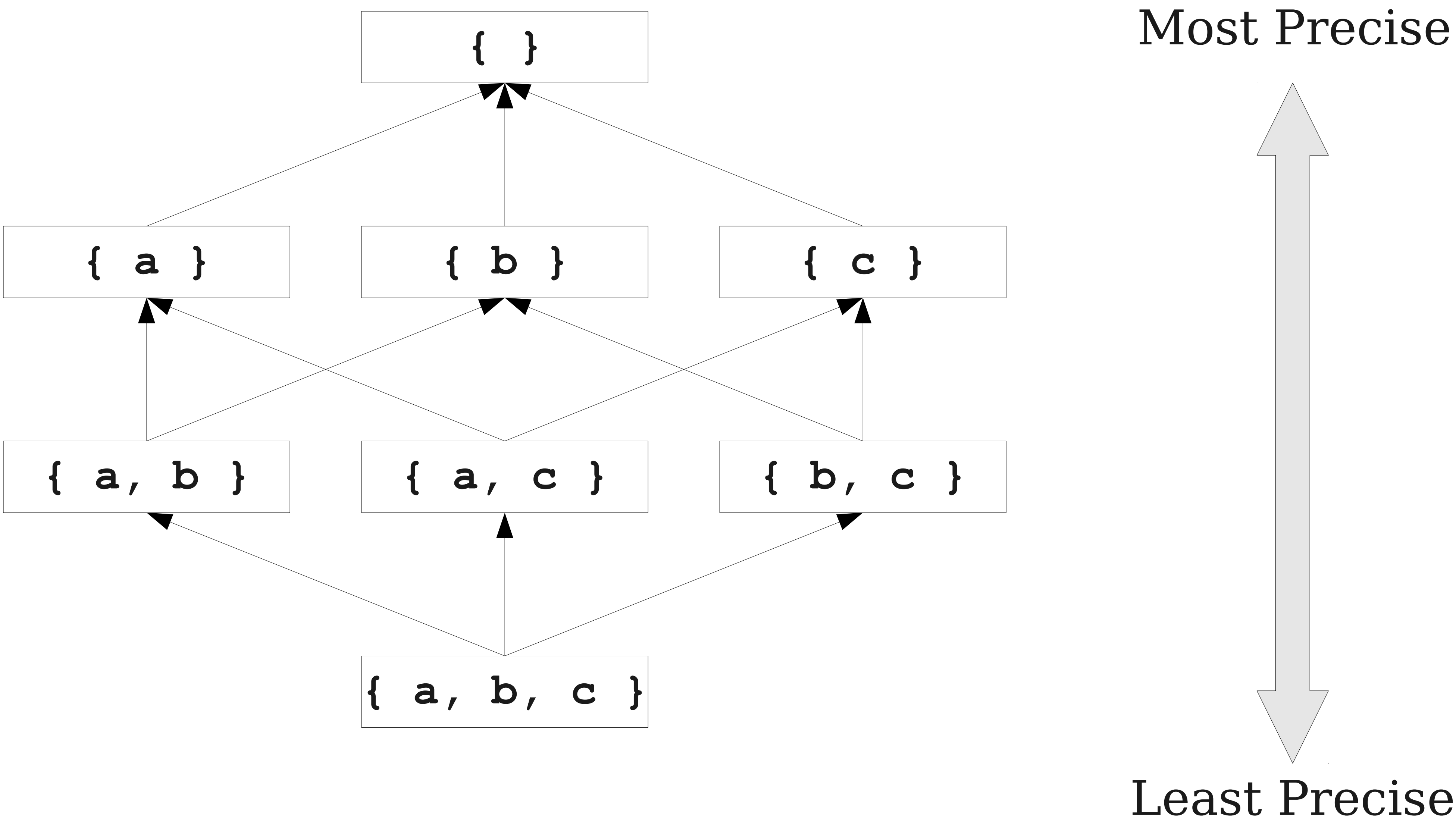
Review: Meet Semilattices

- A **meet semilattice** is a pair $(D, \wedge)$, where
  - D is a domain of elements.
  - $\wedge$ 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



Meet Semilattices and Orderings



Meet Semilattices and Orderings

- Every meet semilattice $(D, \wedge)$ induces an ordering relationship $\leq$ over its elements.
- Define $\mathbf{x \leq y \quad iff \quad x \wedge y = x}$
- Need to prove
  - Reflexivity: $x \leq x$
  - Antisymmetry: If $x \leq y$ and $y \leq x$, then $x = y$.
  - Transitivity: If $x \leq y$ and $y \leq z$, then $x \leq z$.

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
- What is the ordering over these elements?

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: $\emptyset \cup x = x$
- What is the ordering over these elements?

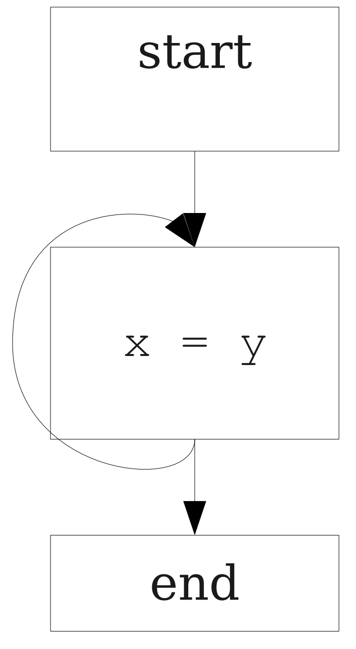
Proving Termination

- 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**.

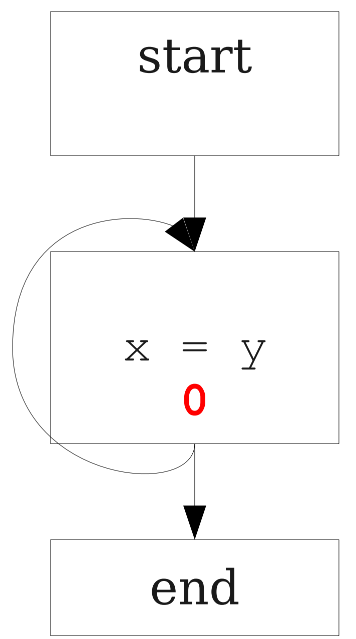
A Nonterminating Analysis

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

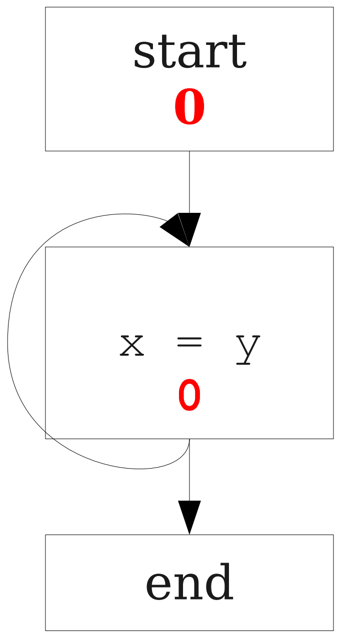
A Nonterminating Analysis



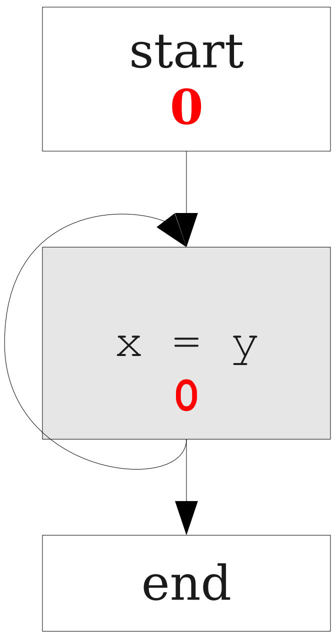
A Nonterminating Analysis



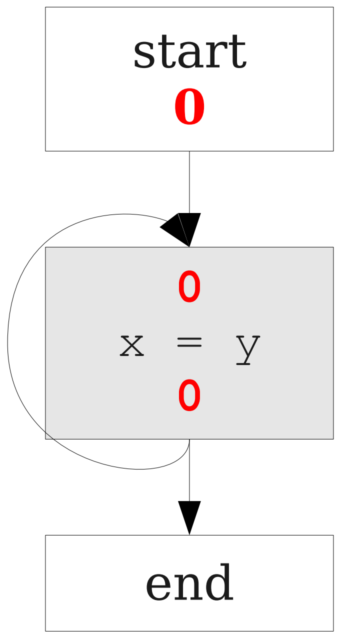
A Nonterminating Analysis



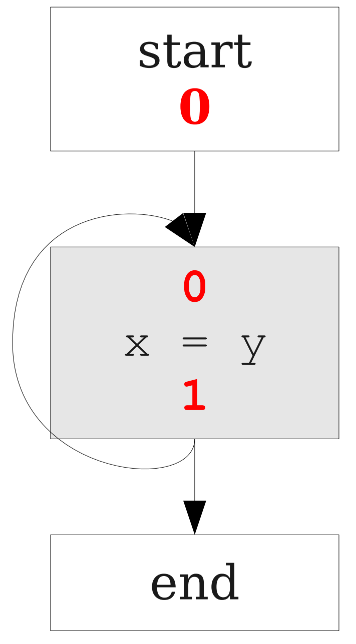
A Nonterminating Analysis



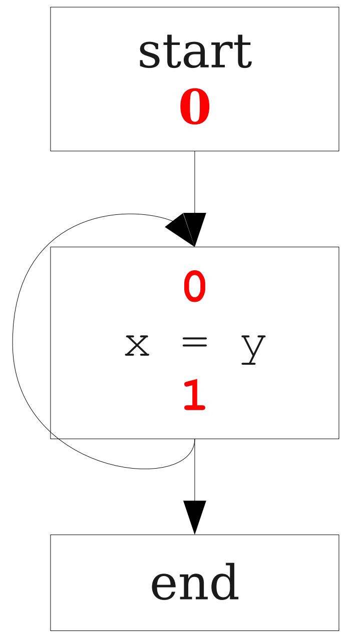
A Nonterminating Analysis



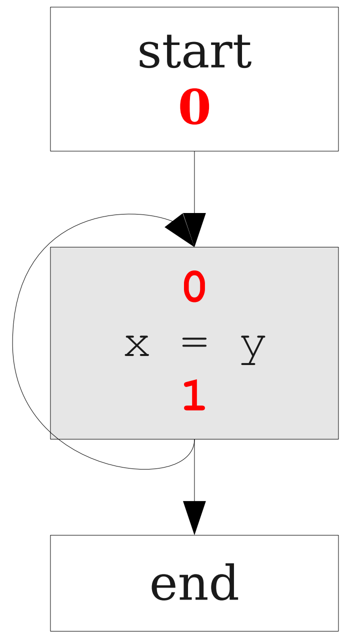
A Nonterminating Analysis



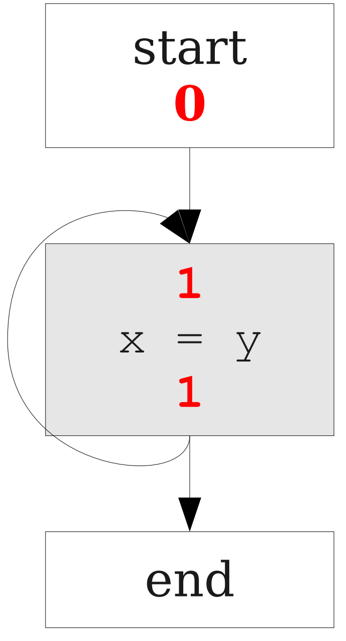
A Nonterminating Analysis



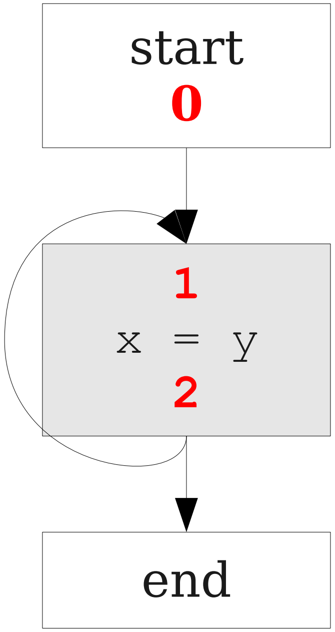
A Nonterminating Analysis



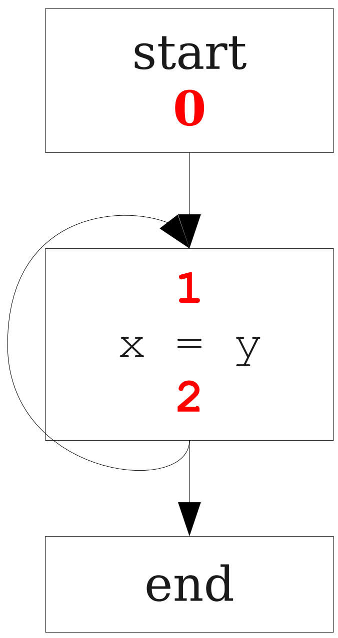
A Nonterminating Analysis



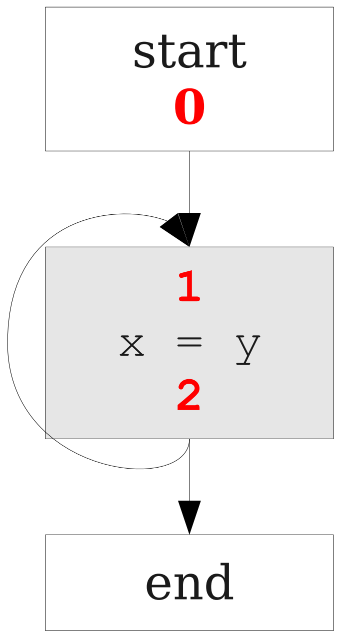
A Nonterminating Analysis



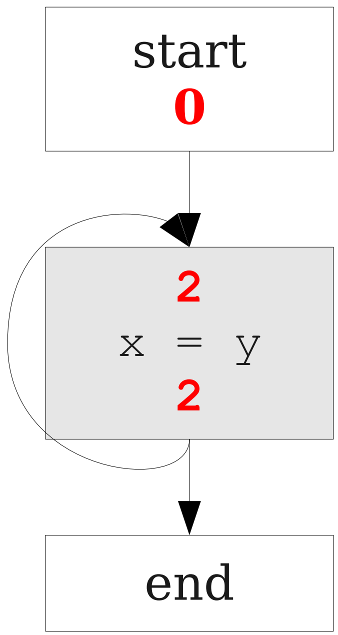
A Nonterminating Analysis



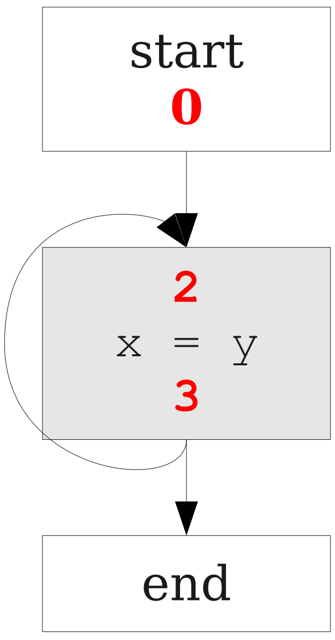
A Nonterminating Analysis



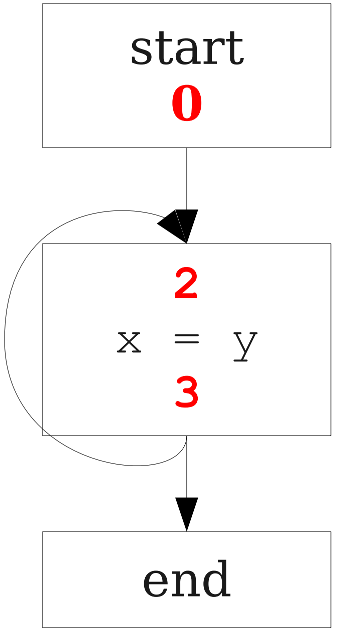
A Nonterminating Analysis



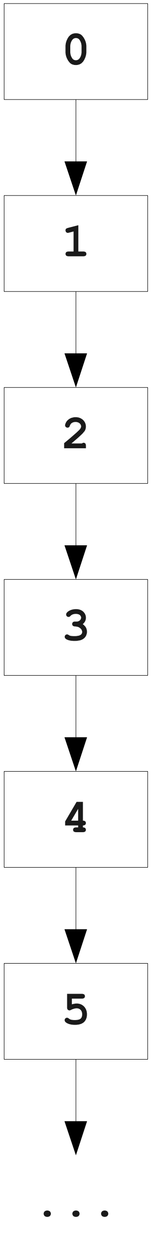
A Nonterminating Analysis



A Nonterminating Analysis



Why Doesn't This Terminate?

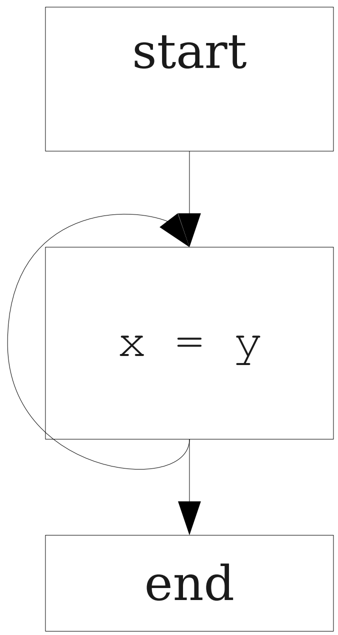


- **Values can decrease without bound.**
  - Note that "decrease" refers to the lattice ordering, not the ordering on the natural numbers.
- The **height** of a semilattice is the length of the longest decreasing sequence in that semilattice.
- The dataflow framework is not guaranteed to terminate for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

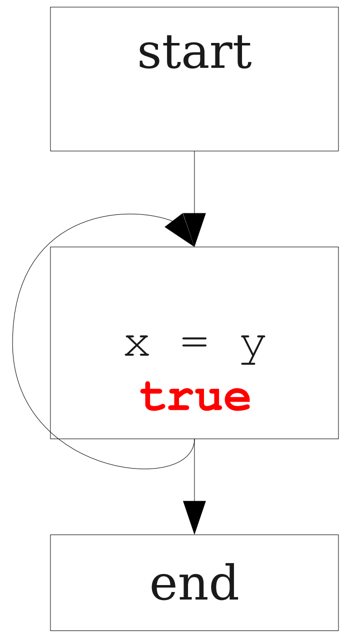
Another Nonterminating Analysis

- 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`

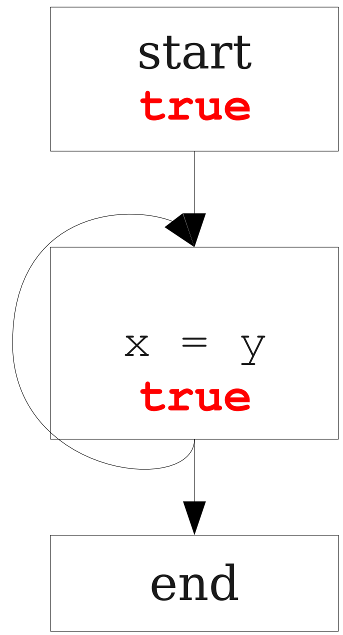
Another Nonterminating Analysis



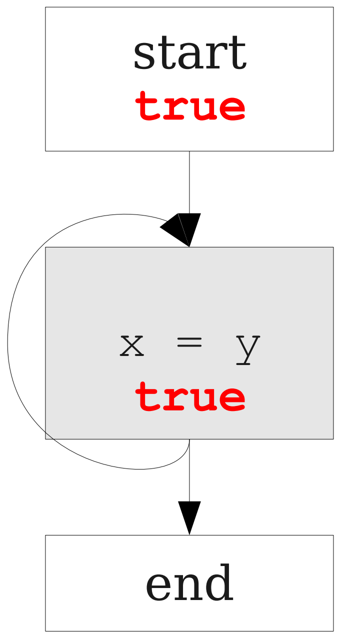
Another Nonterminating Analysis



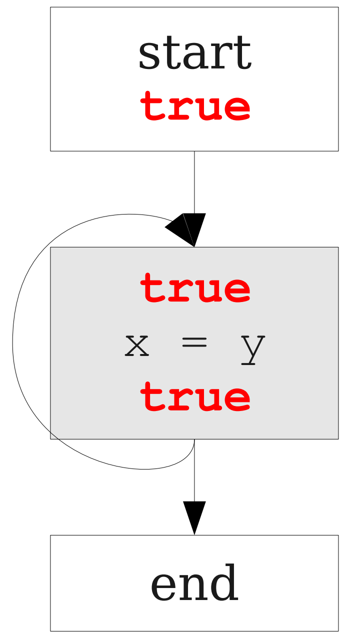
Another Nonterminating Analysis



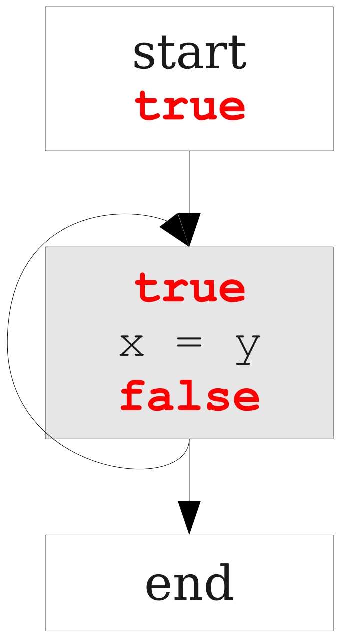
Another Nonterminating Analysis



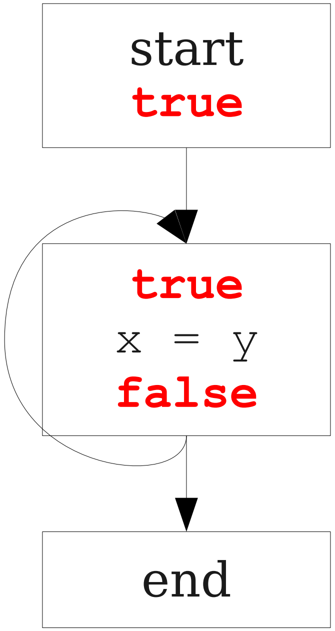
Another Nonterminating Analysis



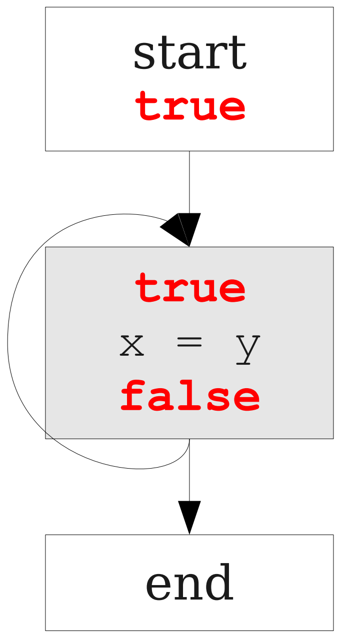
Another Nonterminating Analysis



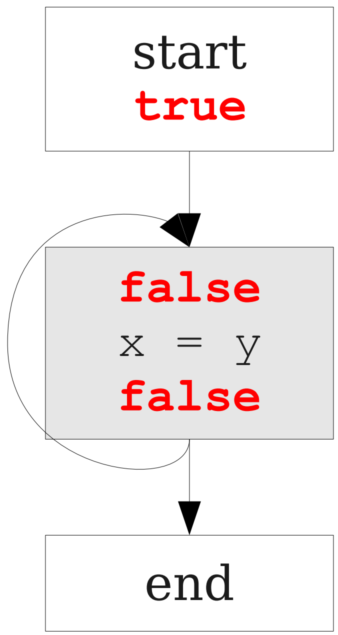
Another Nonterminating Analysis



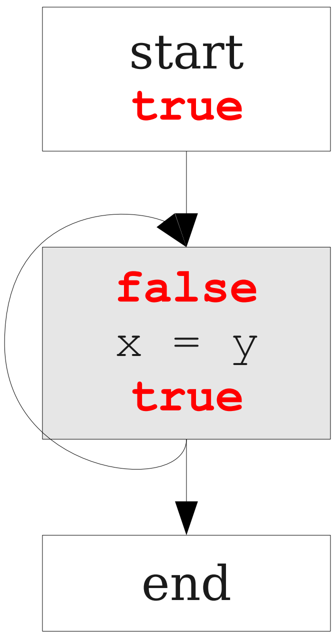
Another Nonterminating Analysis



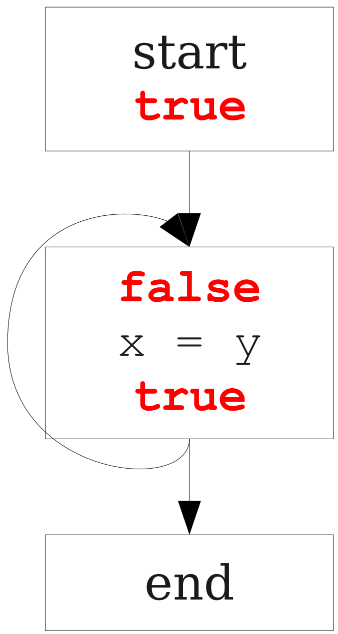
Another Nonterminating Analysis



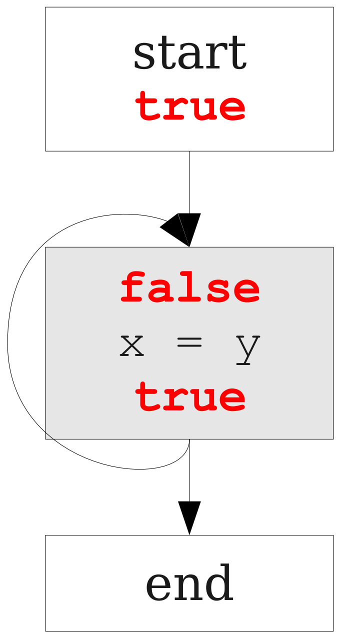
Another Nonterminating Analysis



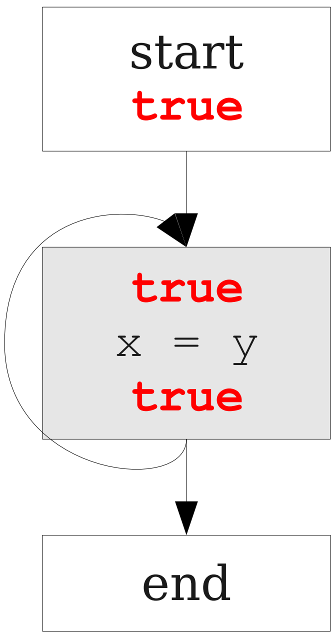
Another Nonterminating Analysis



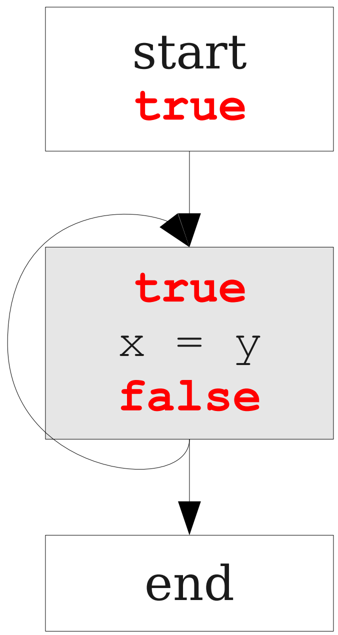
Another Nonterminating Analysis



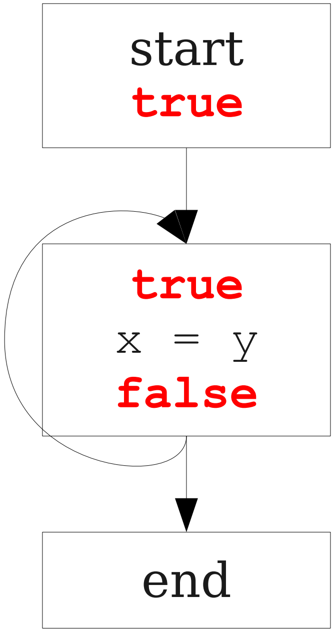
Another Nonterminating Analysis



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's wrong
with cycling 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** iff

$$\textbf{if } x \le y\textbf{, then } f(x) \le f(y)$$

- 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 $f(x) \le x$; we'll see an example.

Liveness and Monotonicity

- A transfer function is **monotone** iff

  if $x \leq y$, then $f(x) \leq f(y)$

- 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 $X \supseteq Y$.
- Is this monotone?

Constant Propagation is Monotone

- A transfer function is **monotone** iff

  if $x \leq y$, then $f(x) \leq f(y)$

- Recall our transfer functions are
  - $f_{x=k}(V) = k$
  - $f_{x=a+b}(V) =$ Not a Constant
  - $f_{y=a+b}(V) = V$
- Is this monotonic?

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)

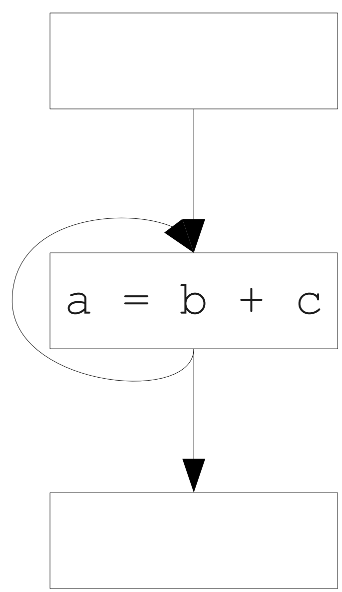
Partial-Redundancy Elimination

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.

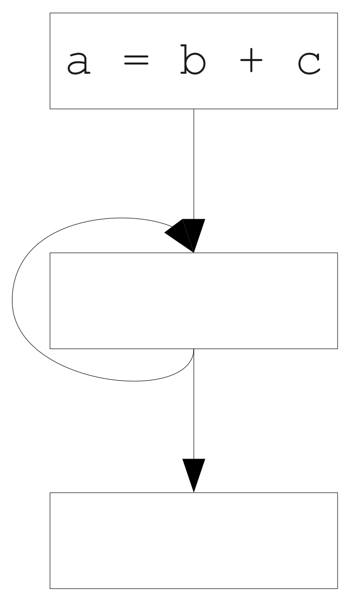
Code Size is Not Execution Time

- 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.

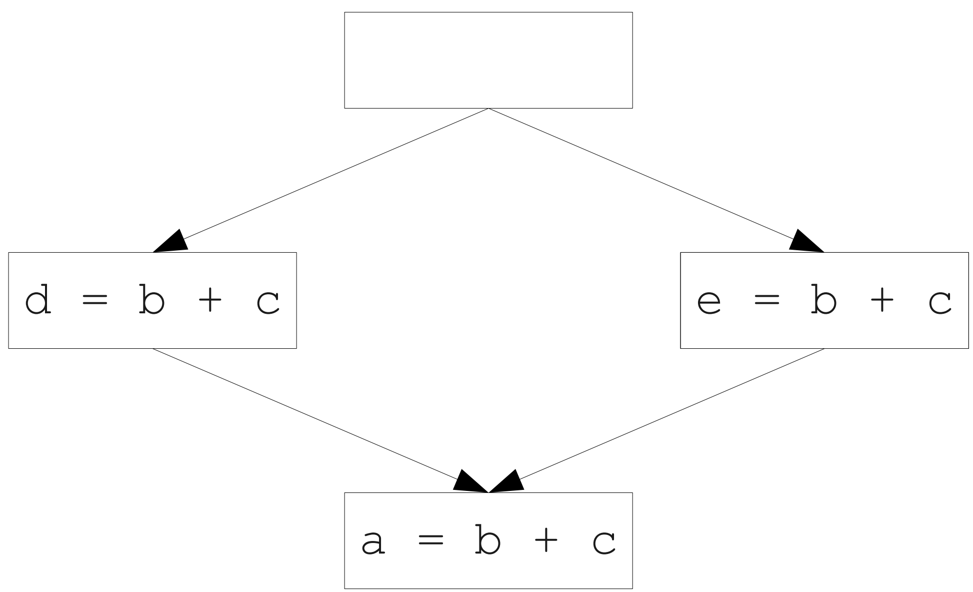


Code Size is Not Execution Time

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

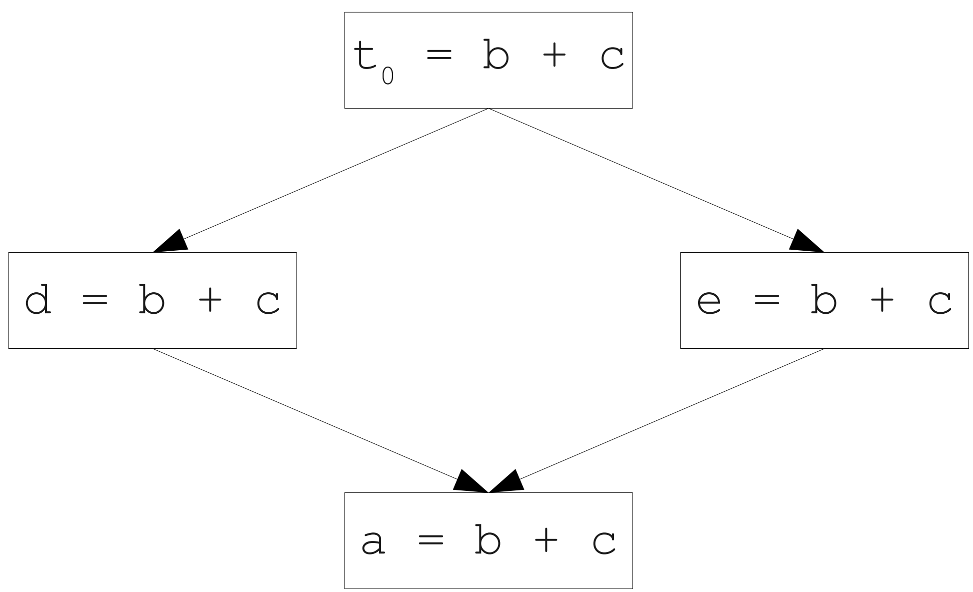
Code Size is Not Execution Time

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



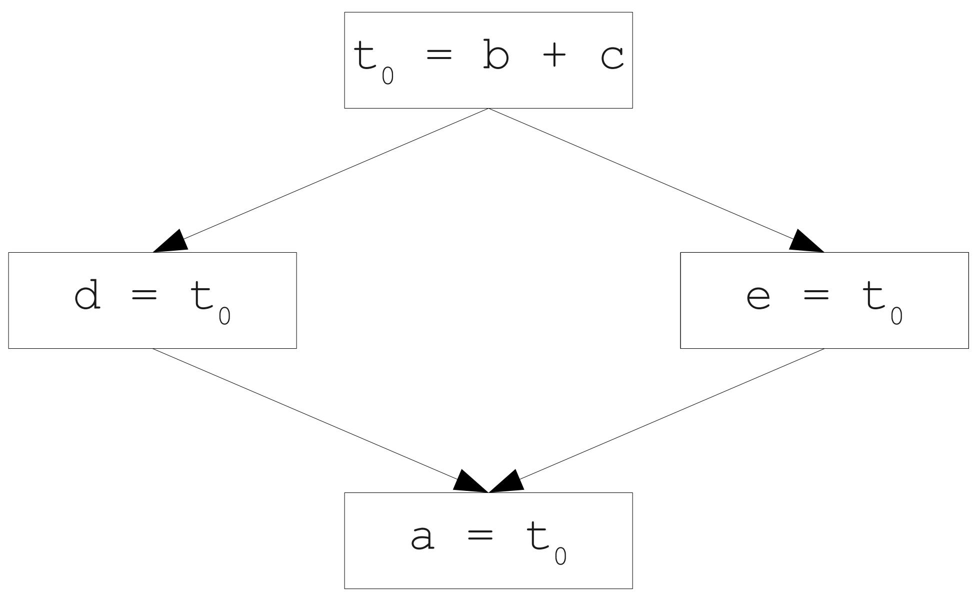
Code Size is Not Execution Time

- 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:

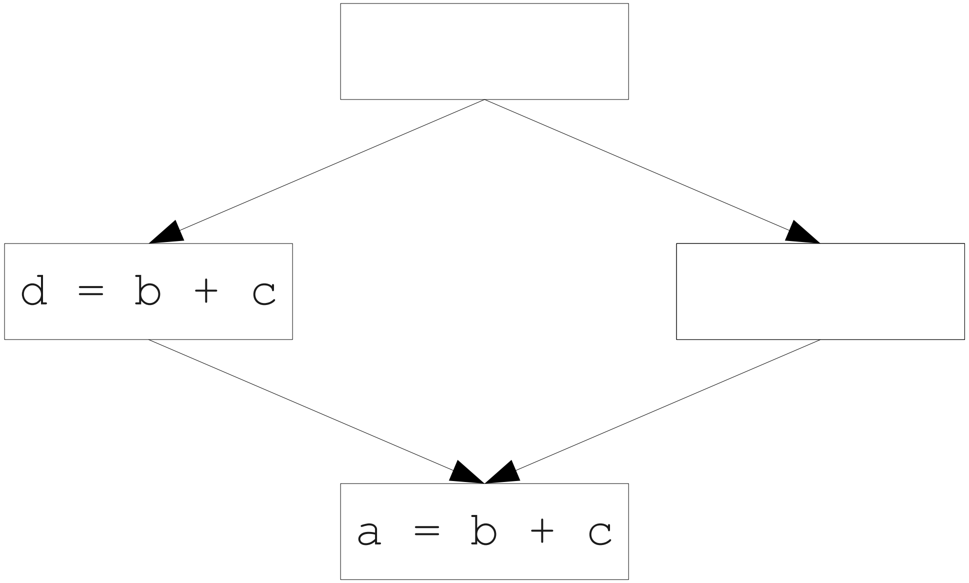


Eliminating Redundancy

- 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.

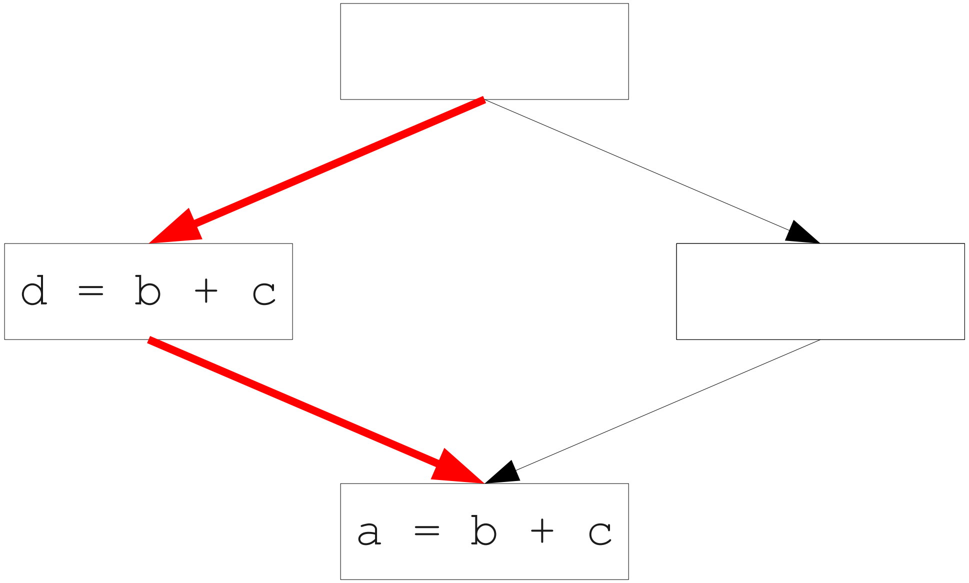
Partial Redundancy

- 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.



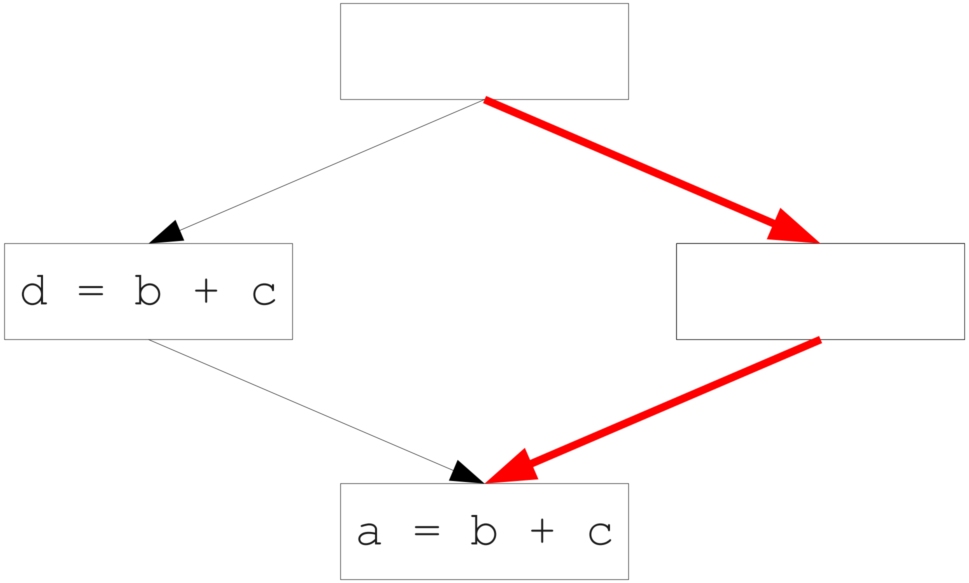
Partial Redundancy

- 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

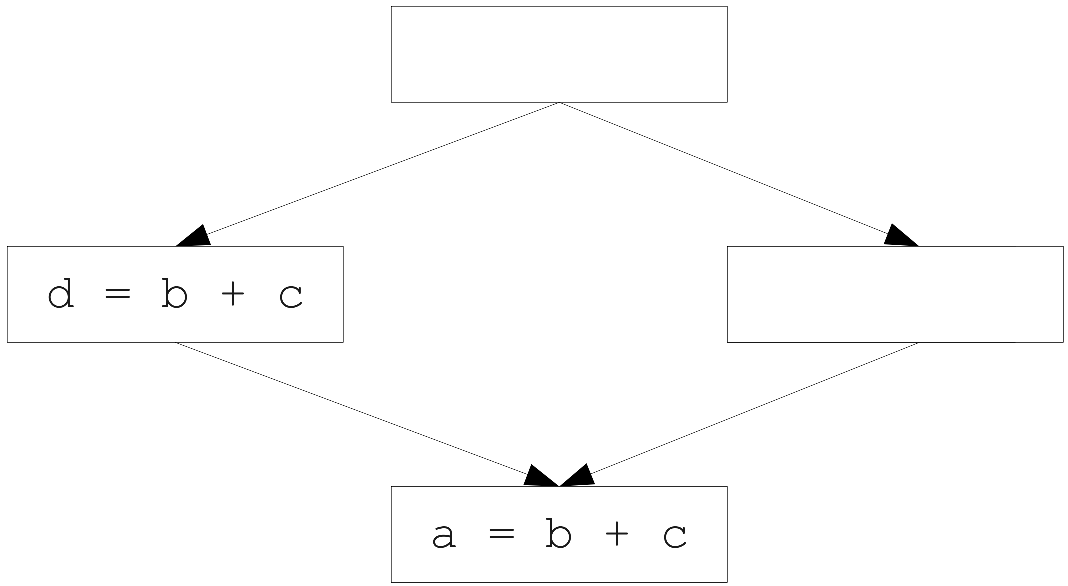
- One of the trickiest cases of redundancy to eliminate is **partial redundancy**.
- A computation is **partially redundant** if its value is known on only some of the paths that reach it.



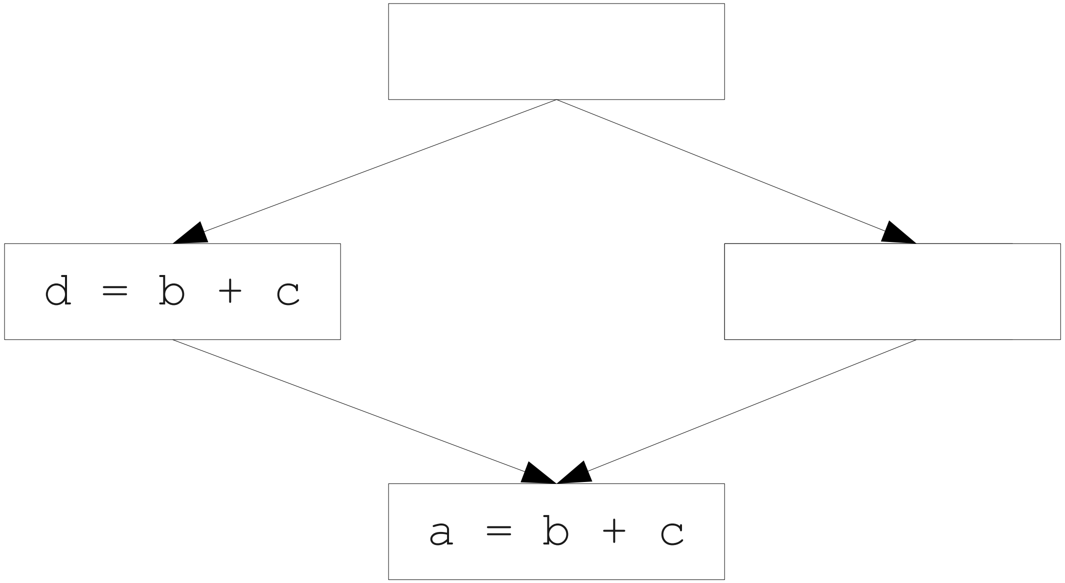
Eliminating Partial Redundancy

- 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

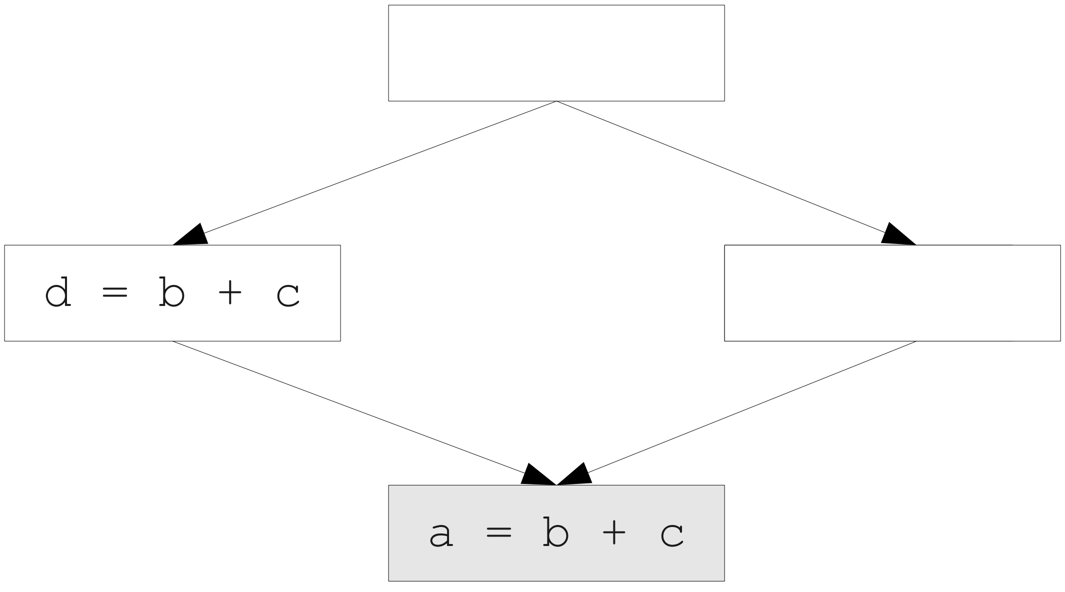


The Key Observation



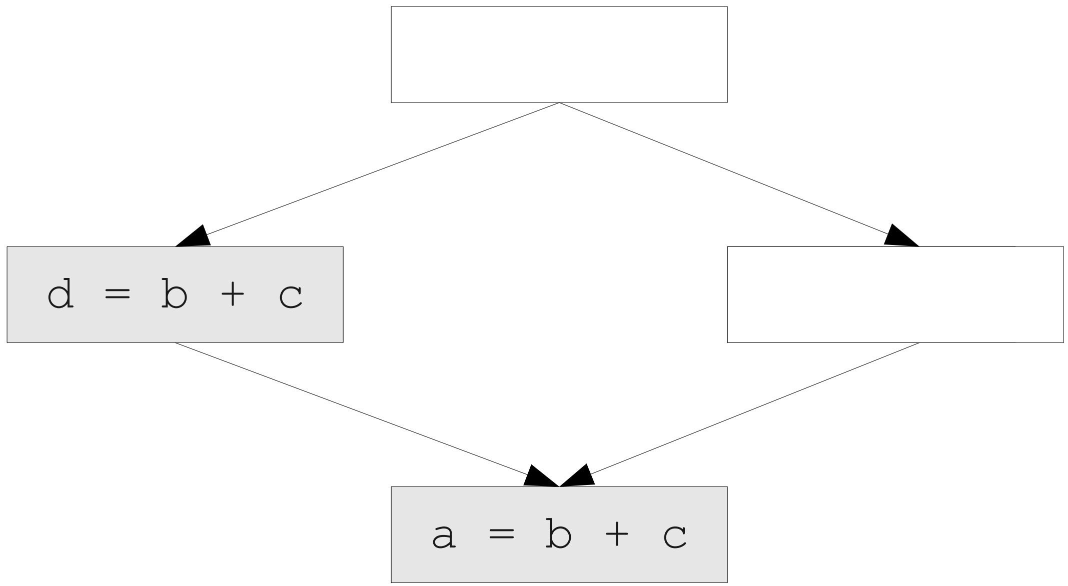
Where in the program is it <u>guaranteed</u> that we will eventually need the value of **b + c**?

The Key Observation



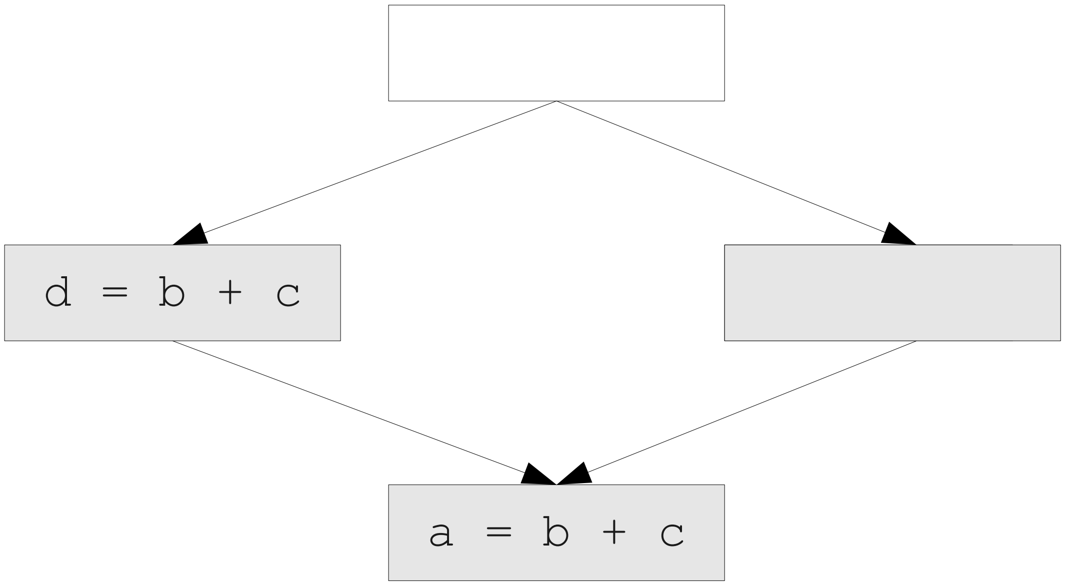
Where in the program is it <u>guaranteed</u> that we will eventually need the value of **b + c**?

The Key Observation



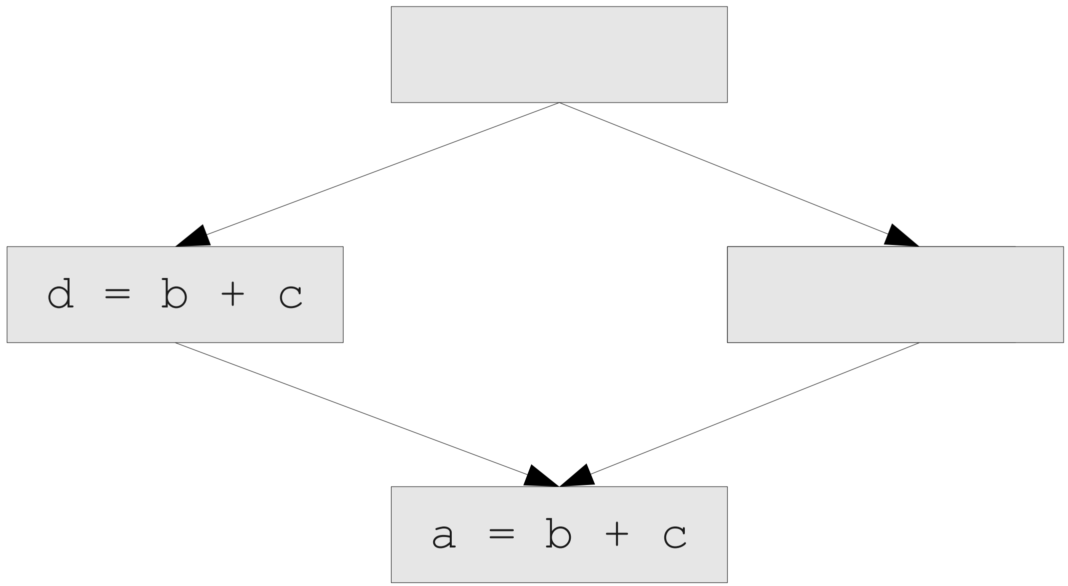
Where in the program is it <u>guaranteed</u> that we will eventually need the value of **b + c**?

The Key Observation



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

The Key Observation

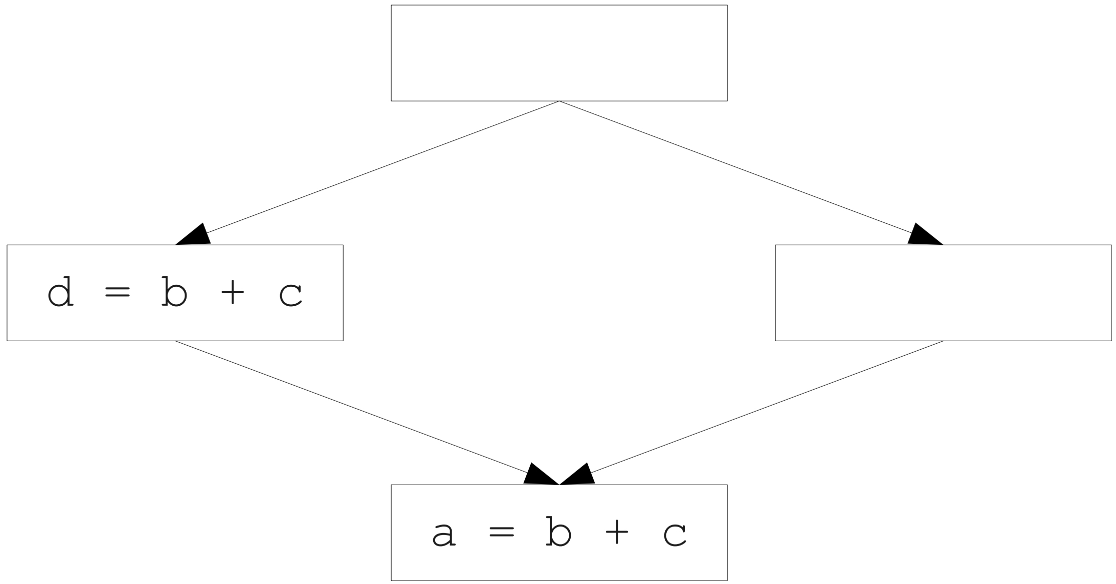


Where in the program is it <u>guaranteed</u> that we will eventually need the value of **b + c**?

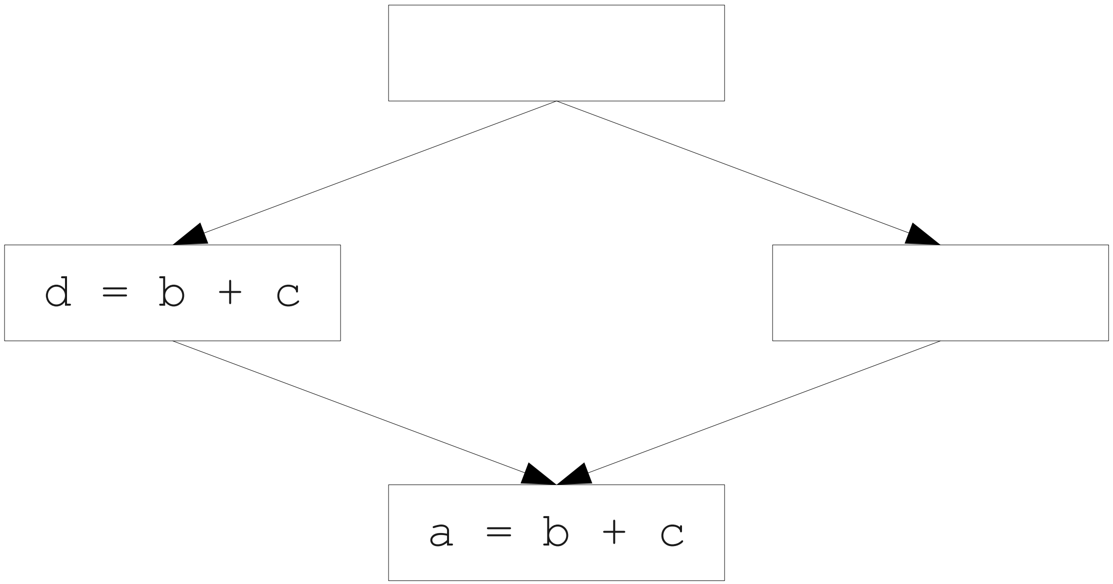
An expression is called **anticipated** at a program point if the expression is guaranteed to be used after that point.

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

The Second Key Observation

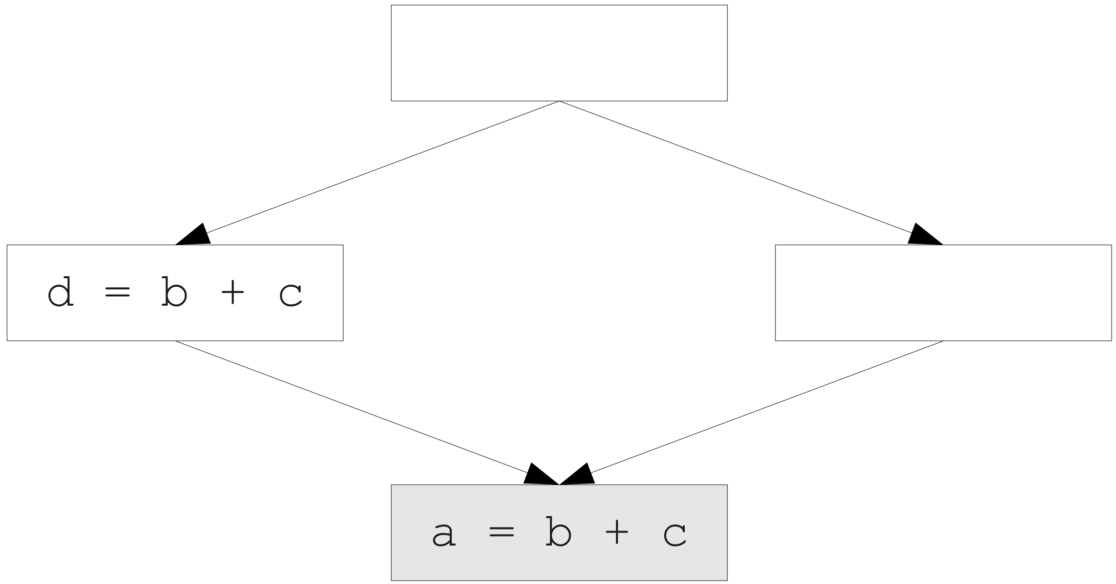


The Second Key Observation



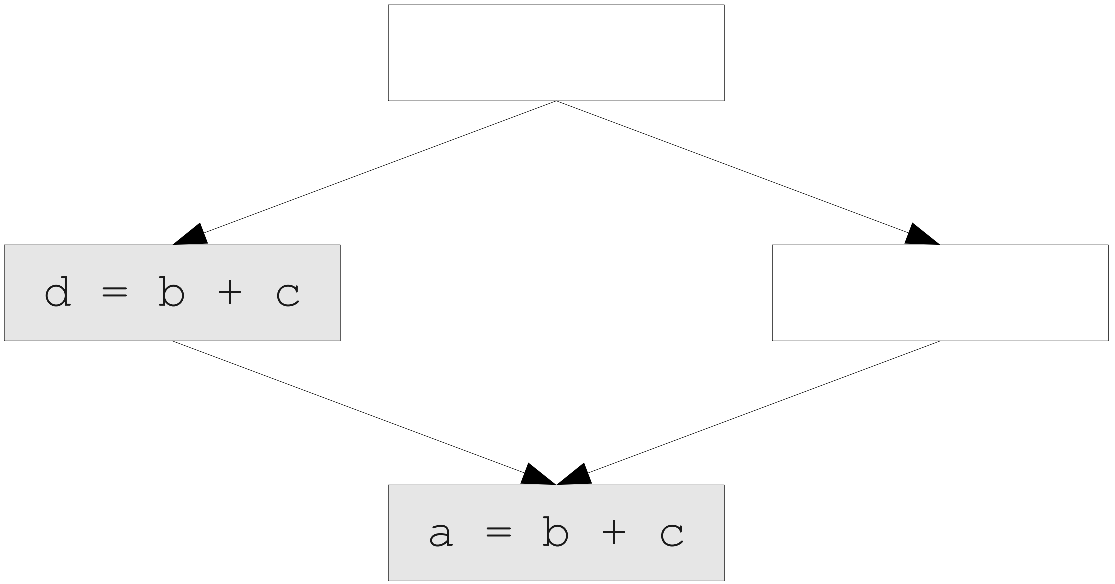
Where in the program
is the value of **`b + c`**
already computed?

The Second Key Observation



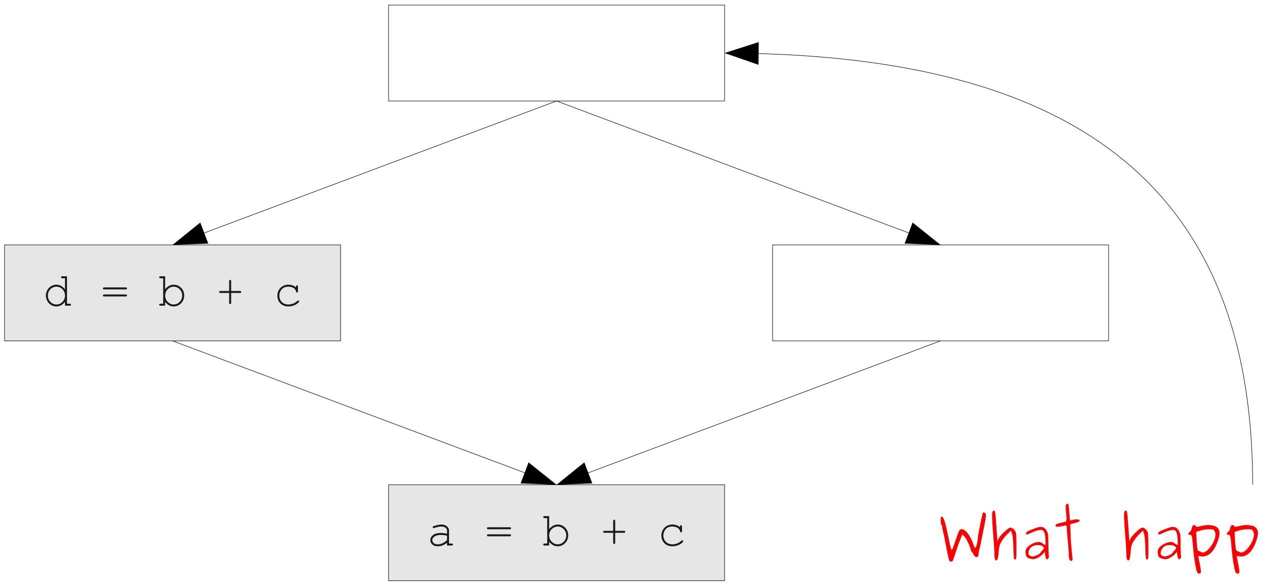
Where in the program
is the value of **b + c**
already computed?

The Second Key Observation



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

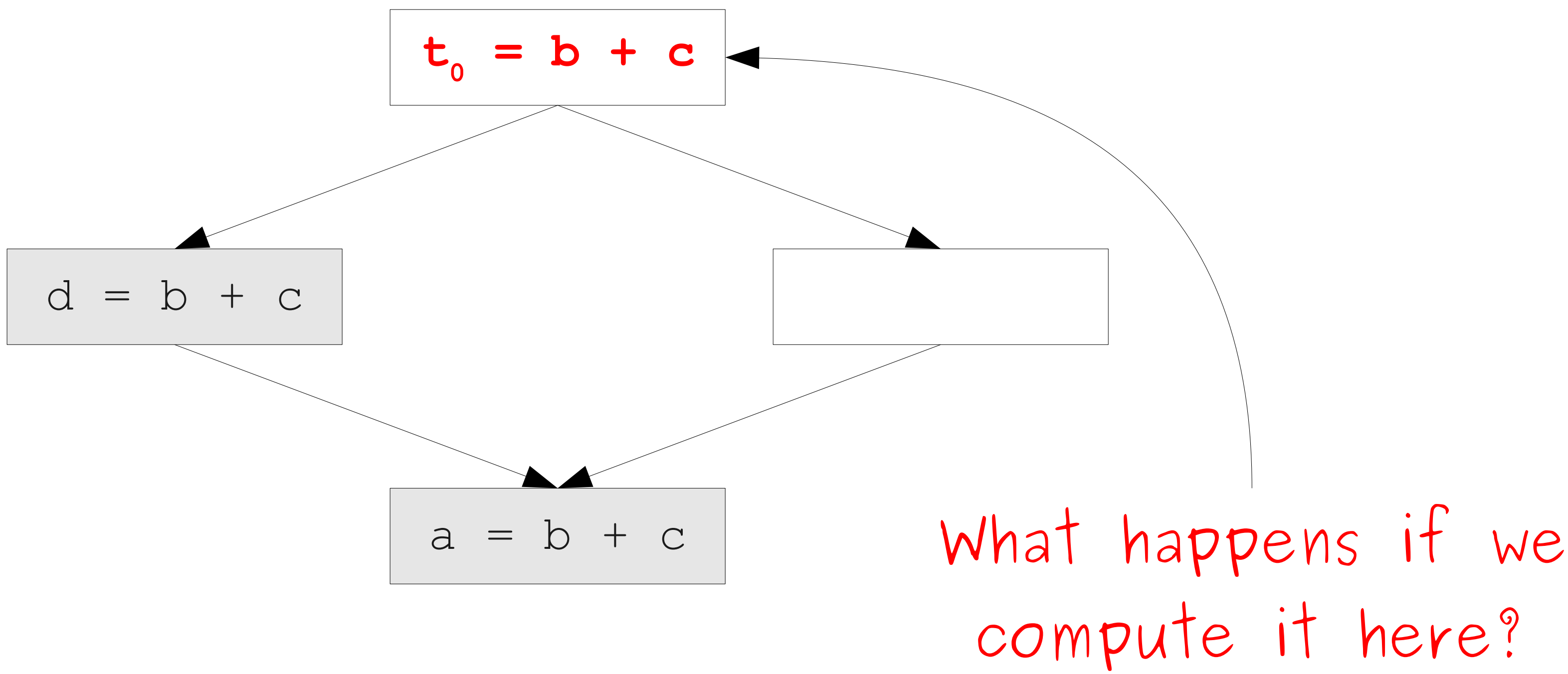
The Second Key Observation



What happens if we compute it here?

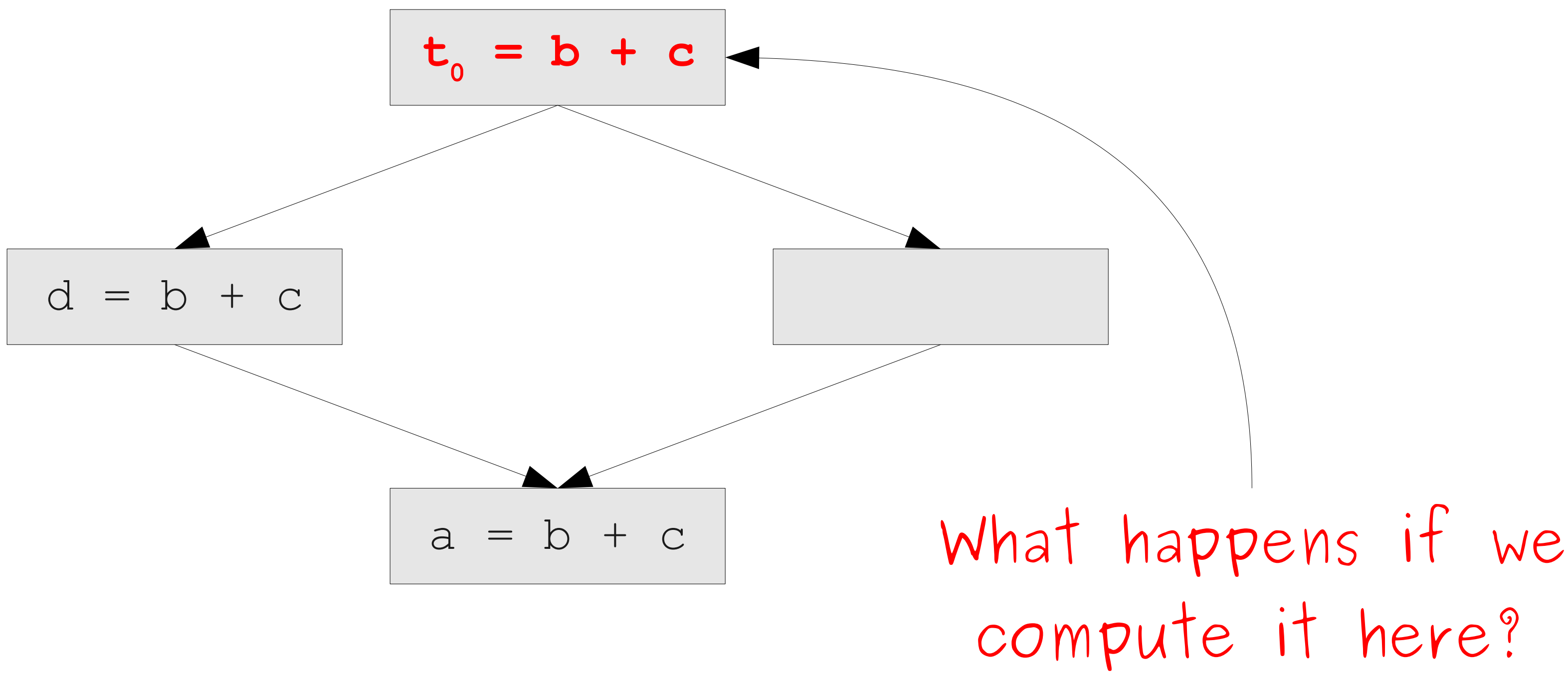
Where in the program is the value of **b + c** already computed?

The Second Key Observation



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

The Second Key Observation



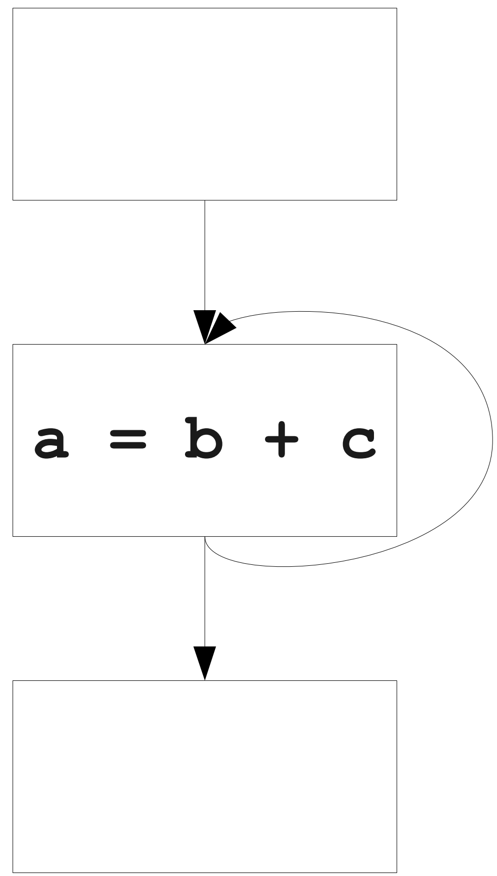
Where in the program is the value of **b + c** already computed?

Partial-Redundancy Elimination

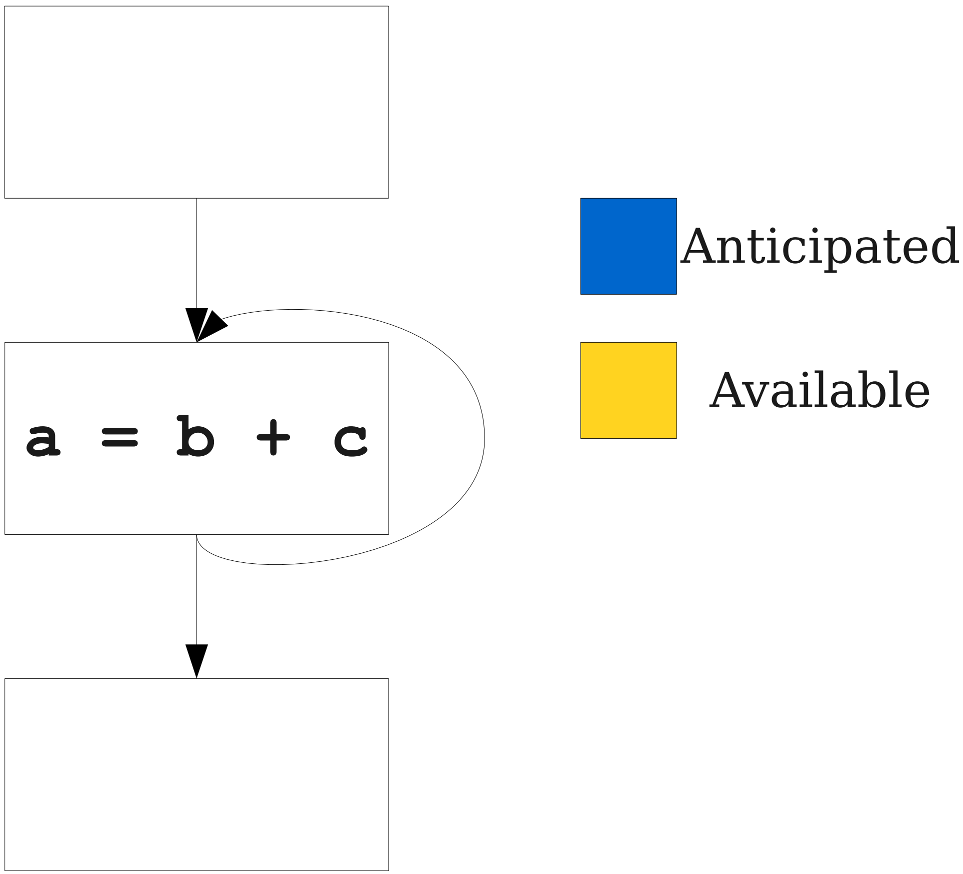
- **Idea:** Make the expression *available* everywhere that it's *anticipated*.
- Run an analysis to locate where the expression is anticipated.
- Run a second analysis to locate where the expression is available.
- Place the expression at the earliest locations where the expression is **anticipated**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

Eliminating Redundancy

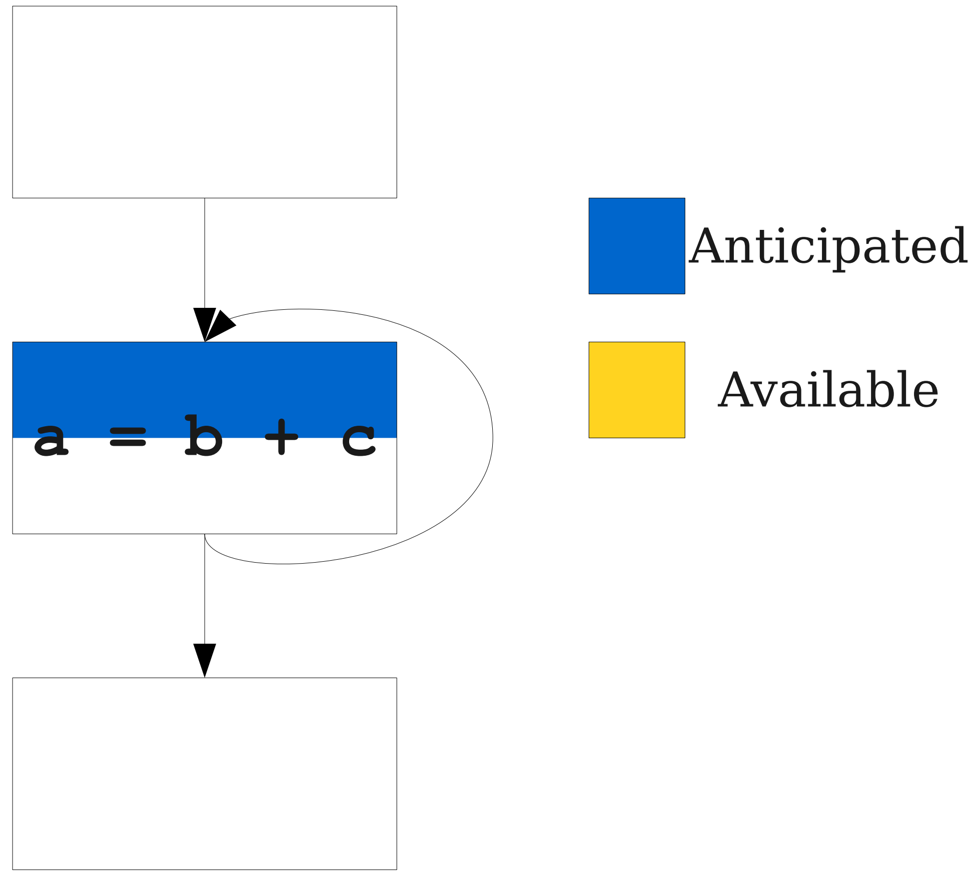
Eliminating Redundancy



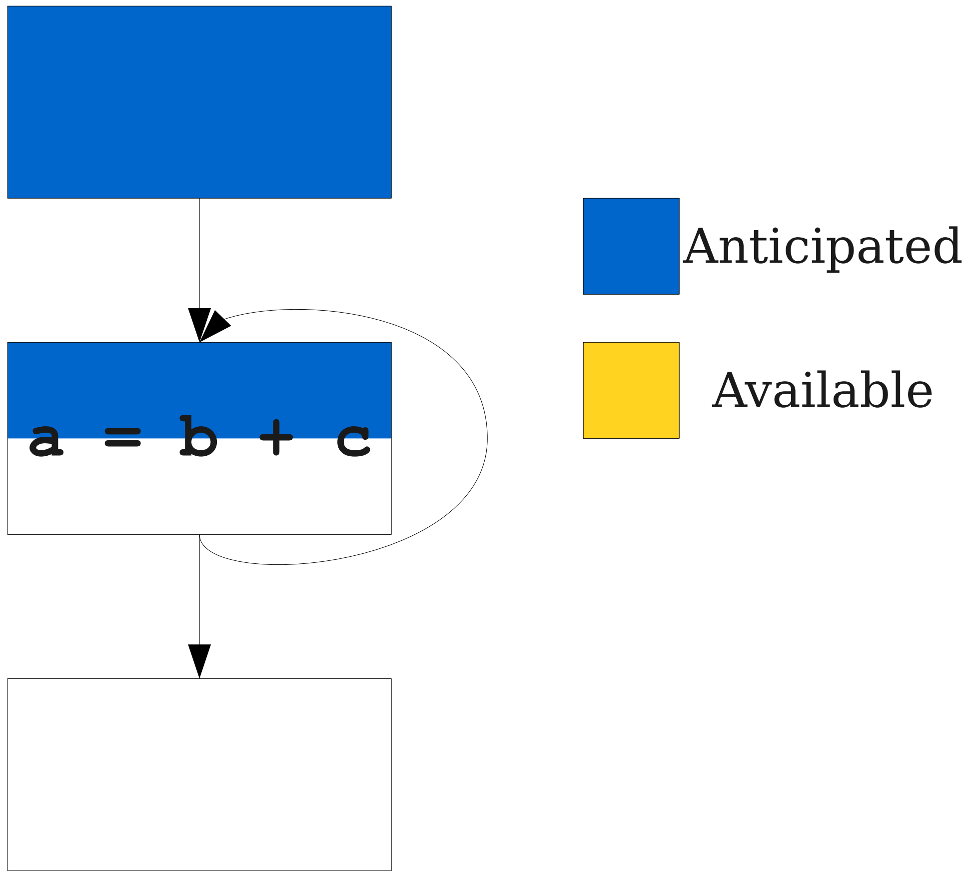
Eliminating Redundancy



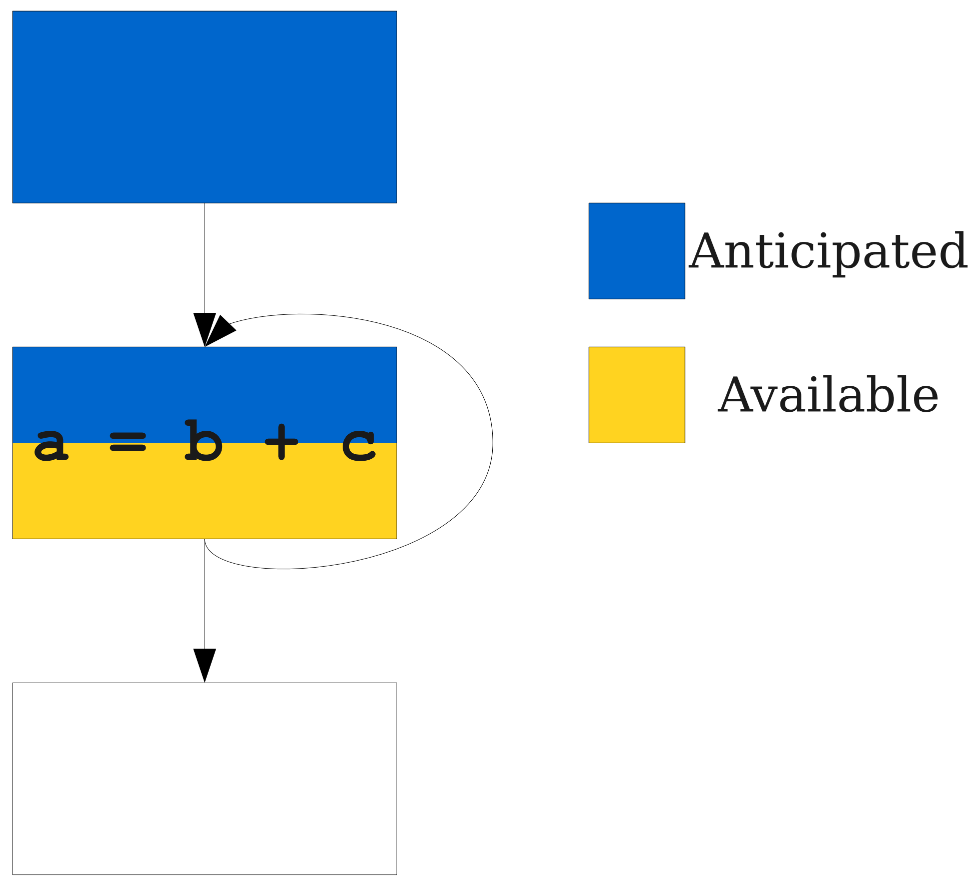
Eliminating Redundancy



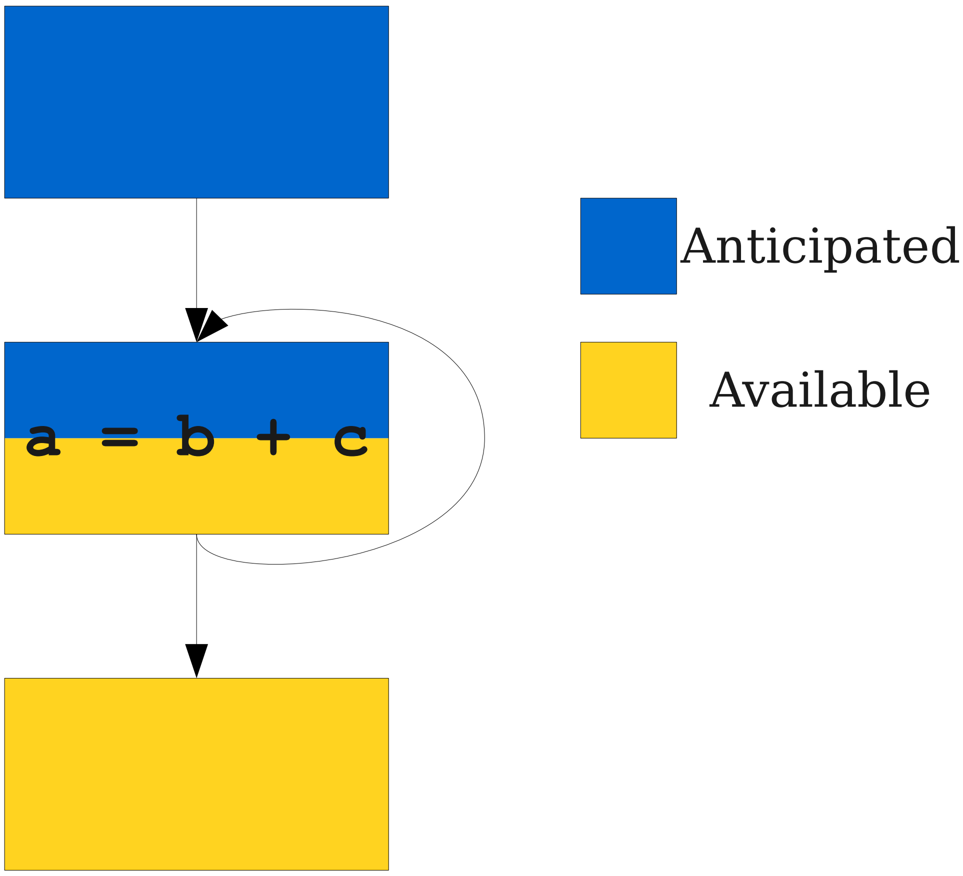
Eliminating Redundancy



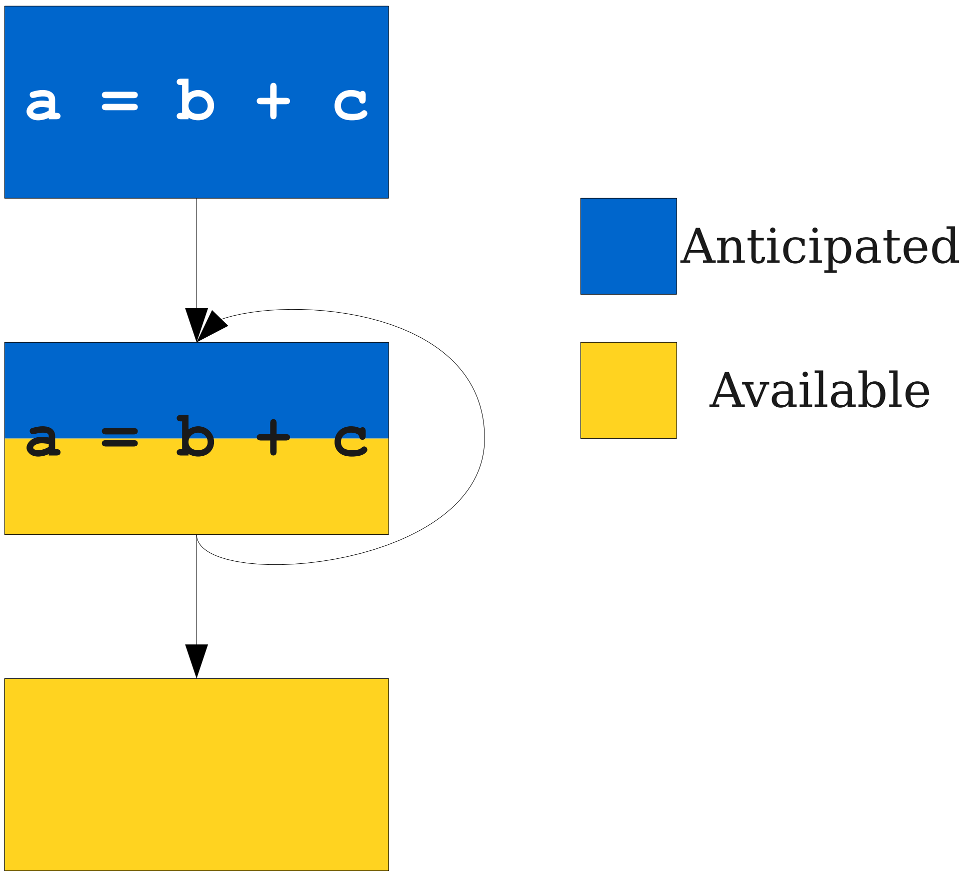
Eliminating Redundancy



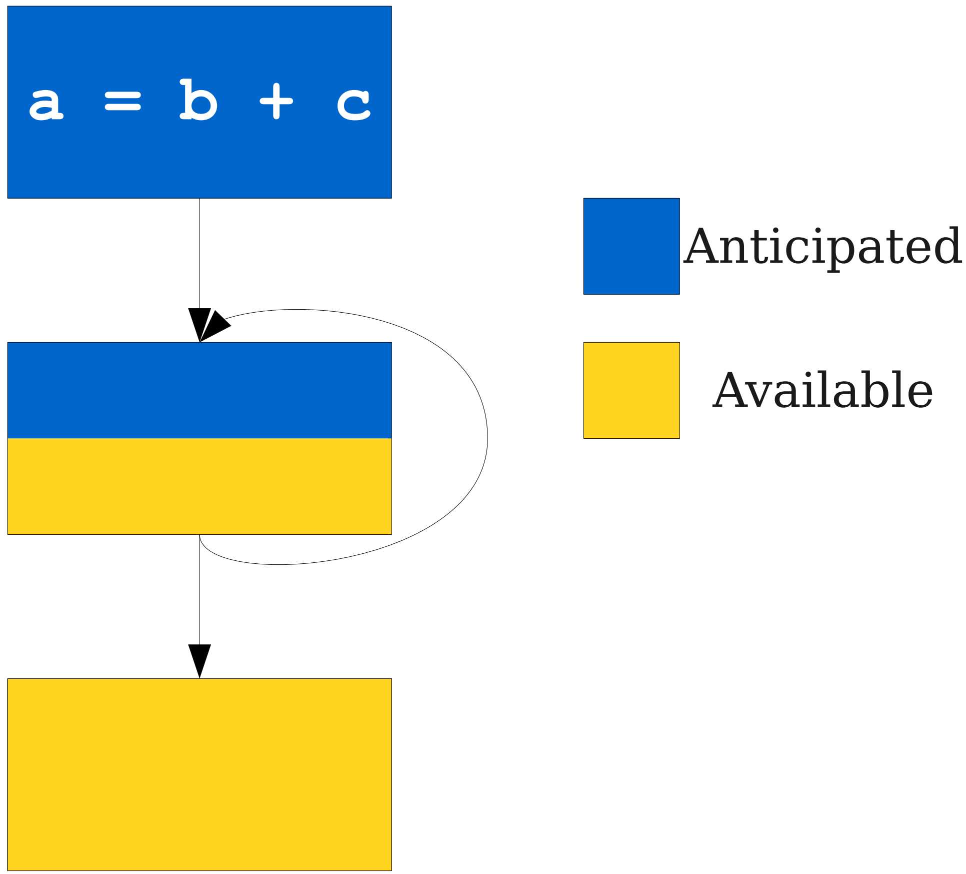
Eliminating Redundancy



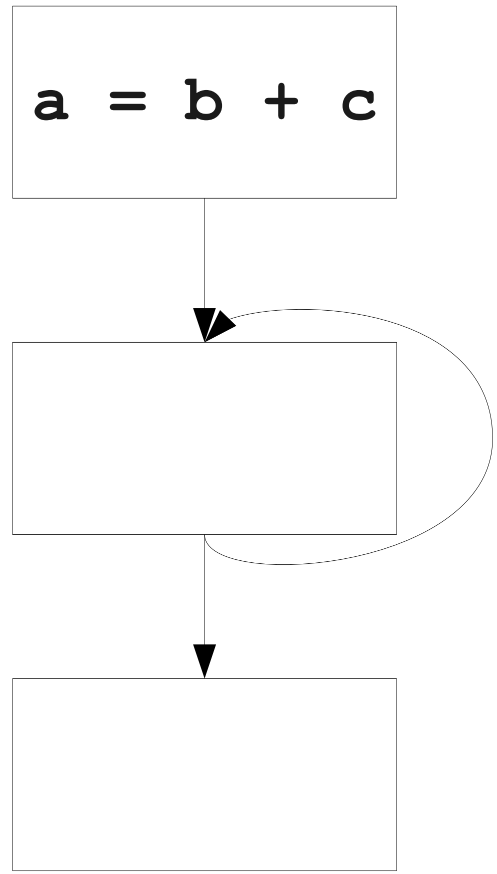
Eliminating Redundancy



Eliminating Redundancy

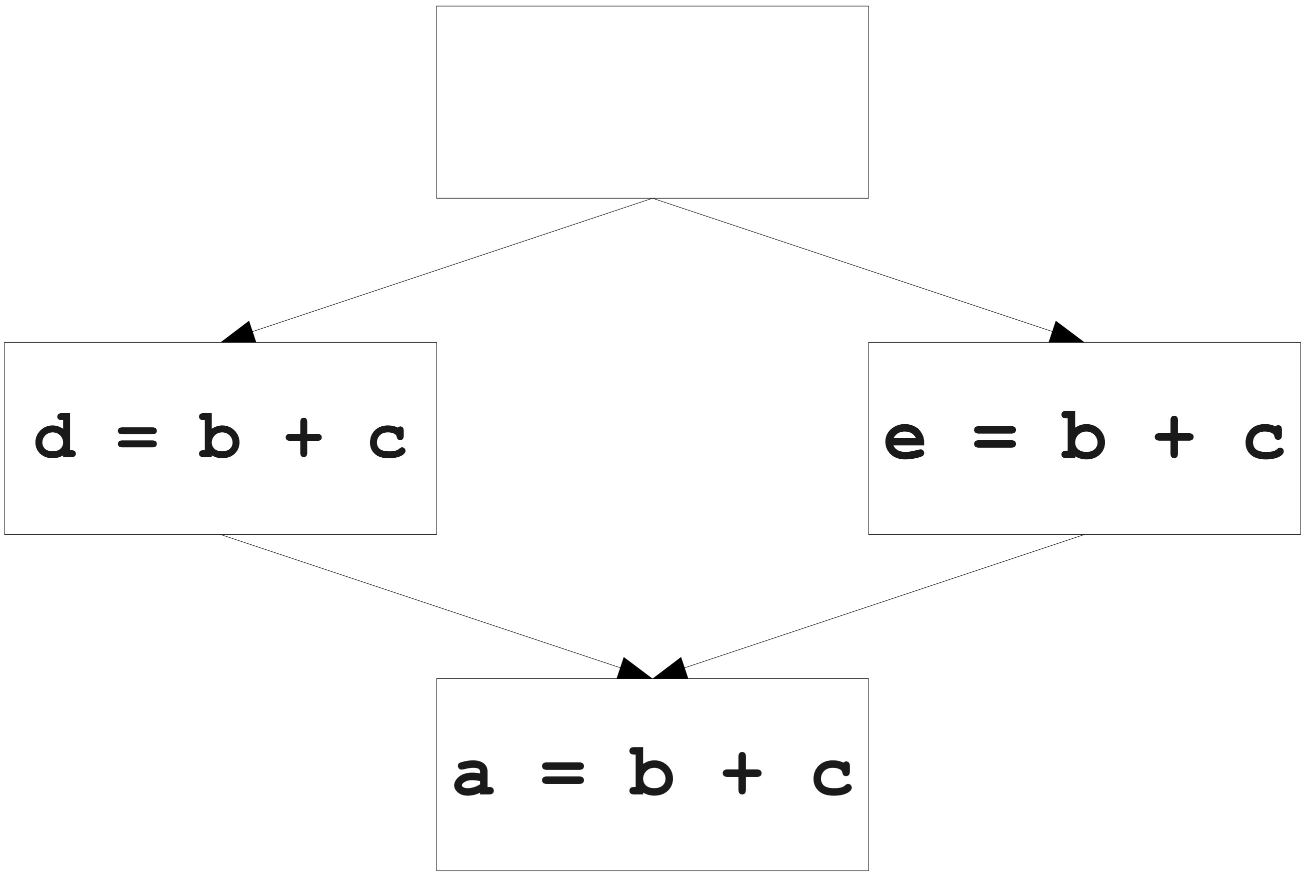


Eliminating Redundancy

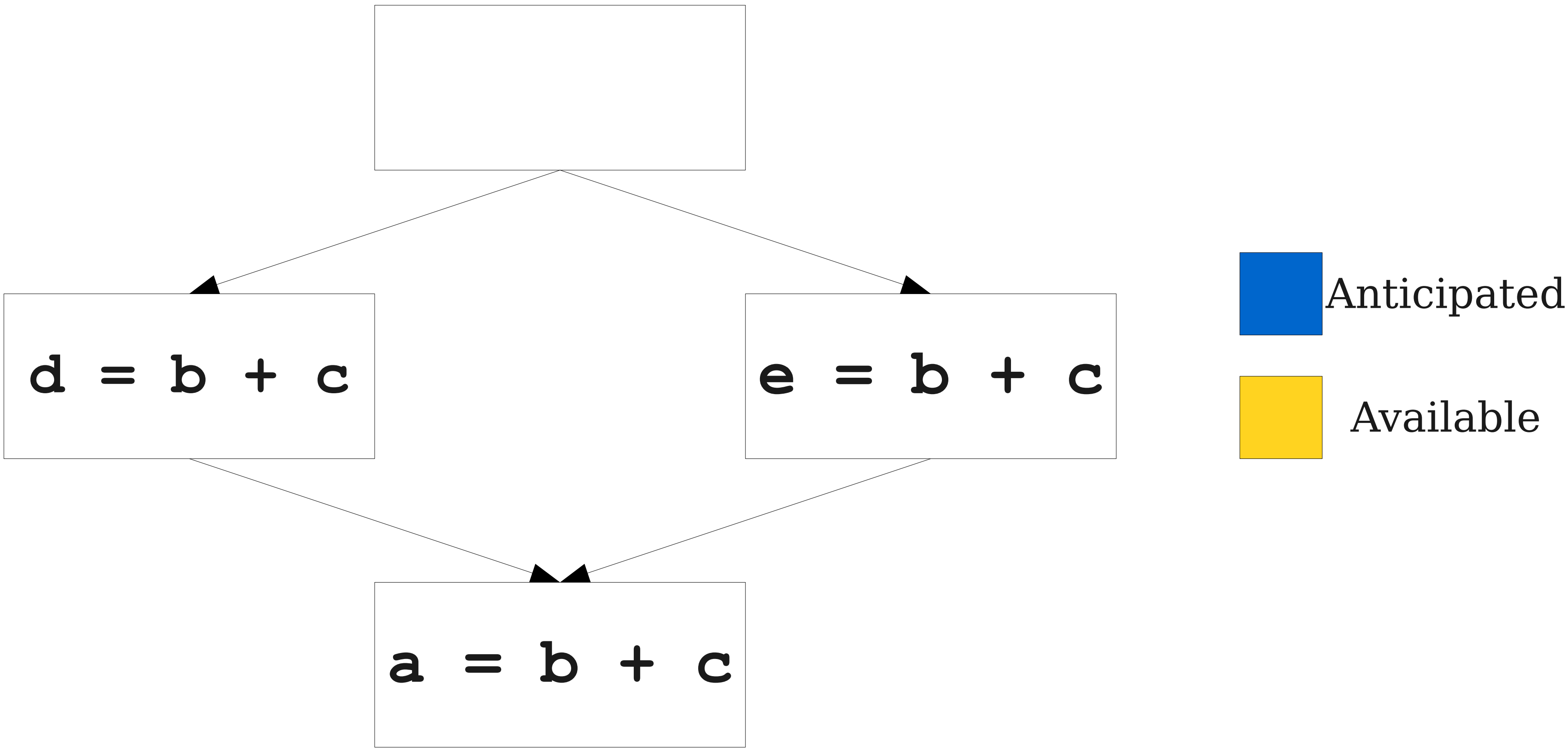


Eliminating Redundancy II

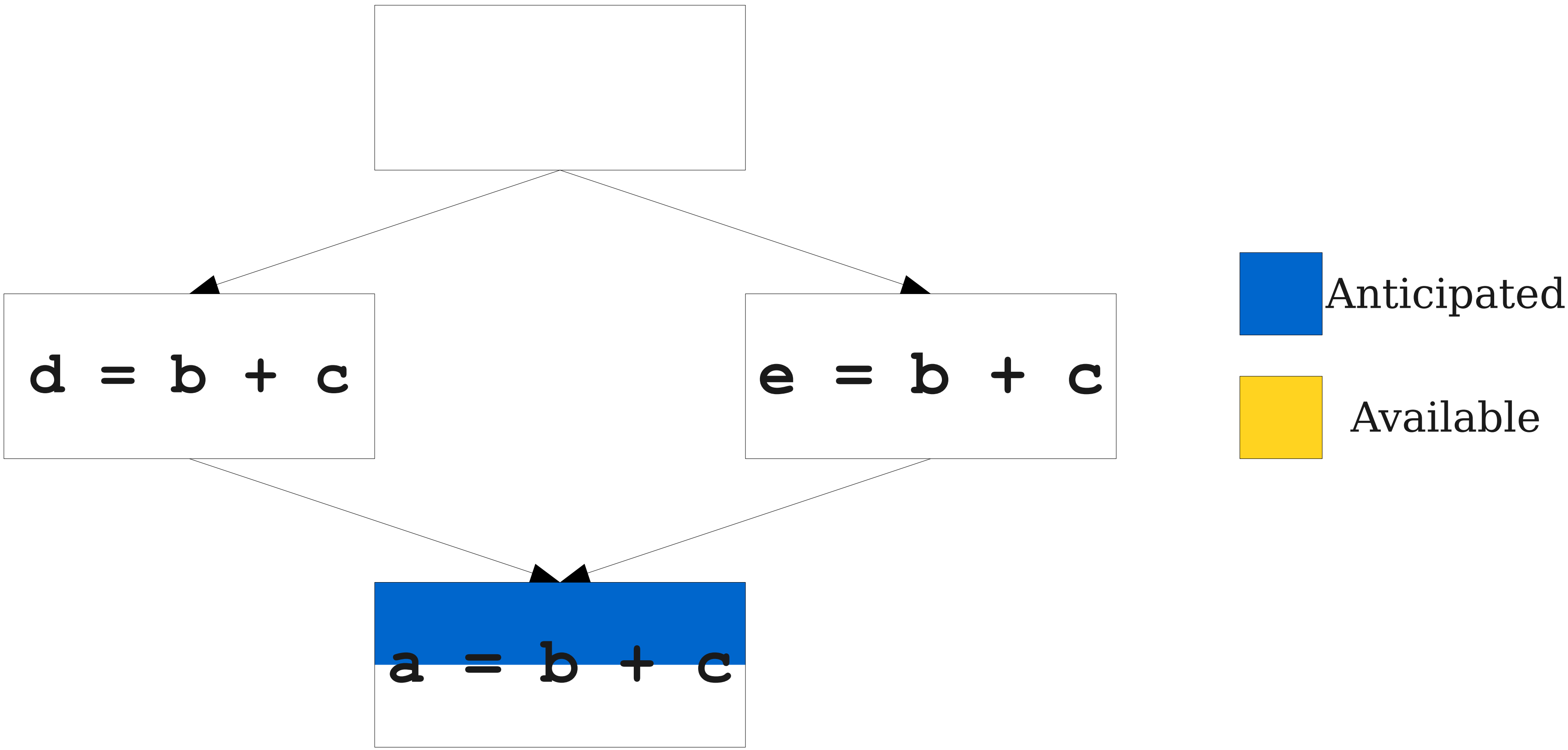
Eliminating Redundancy II



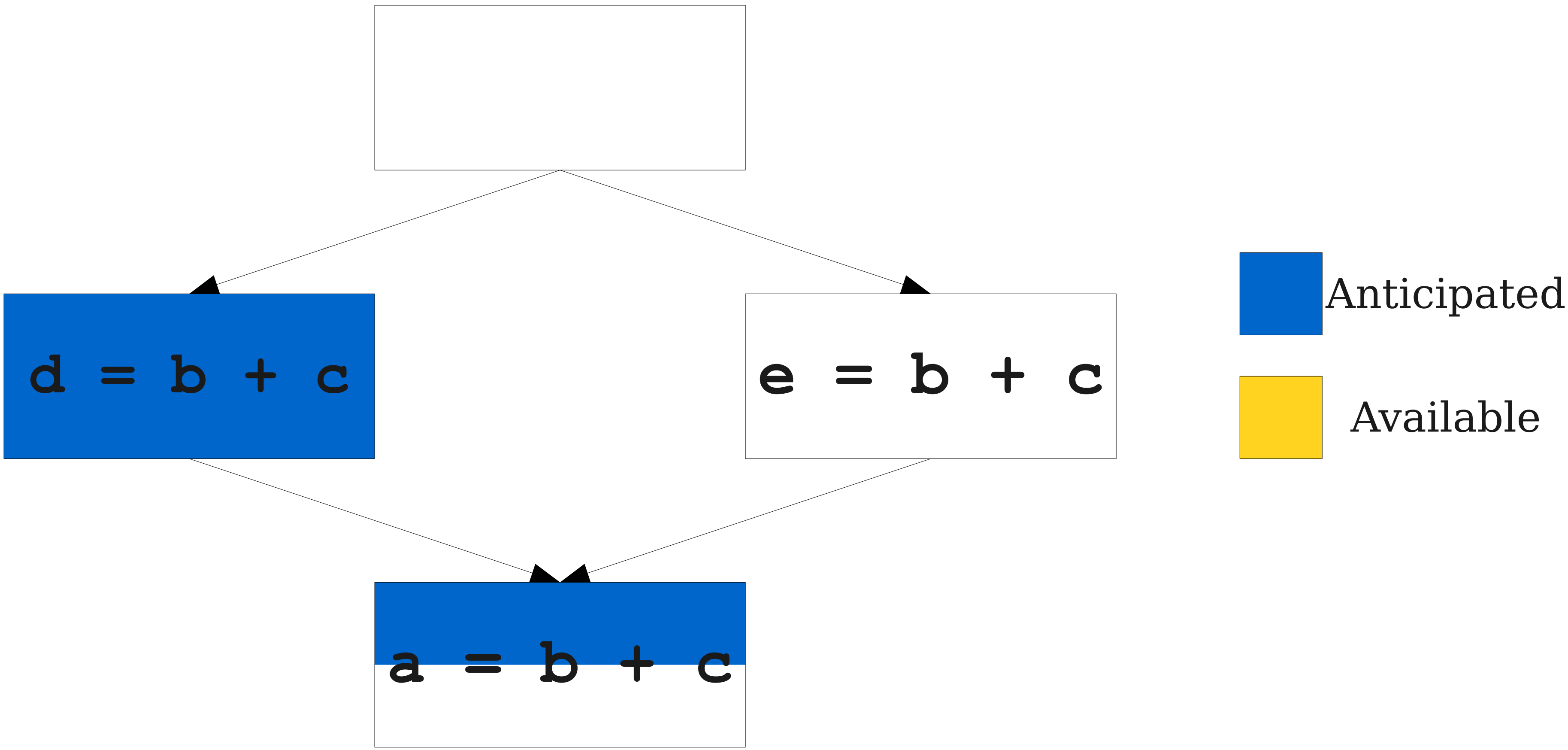
Eliminating Redundancy II



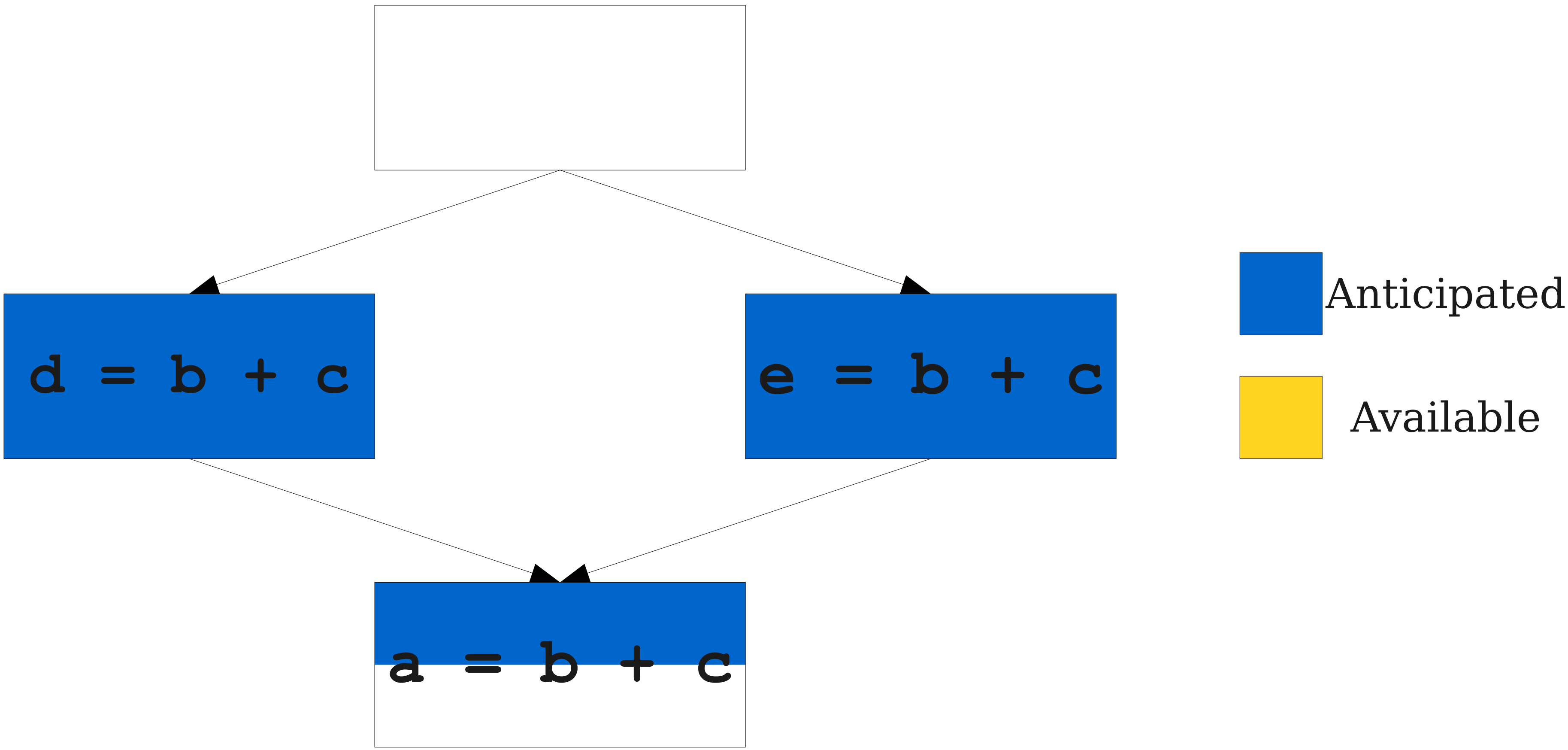
Eliminating Redundancy II



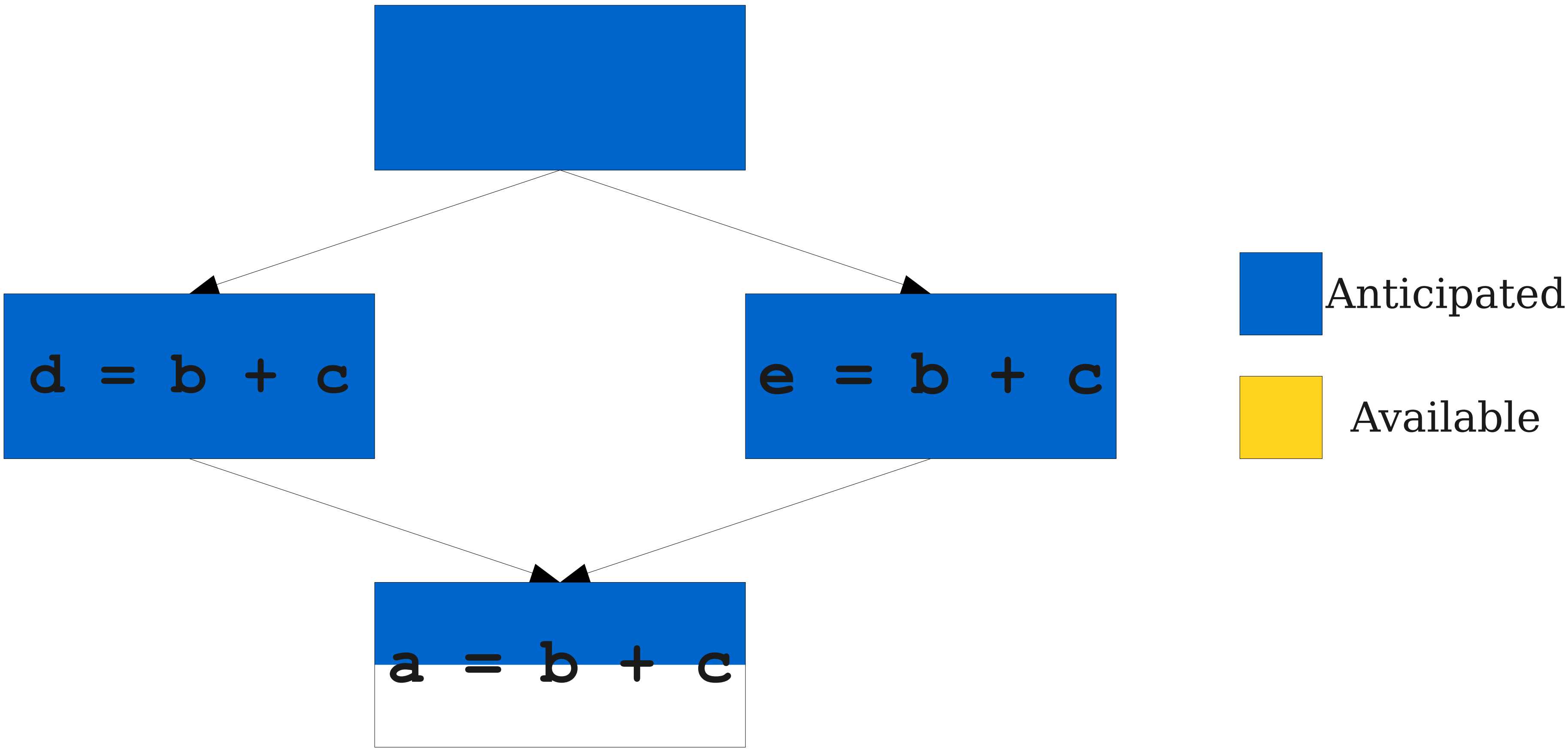
Eliminating Redundancy II



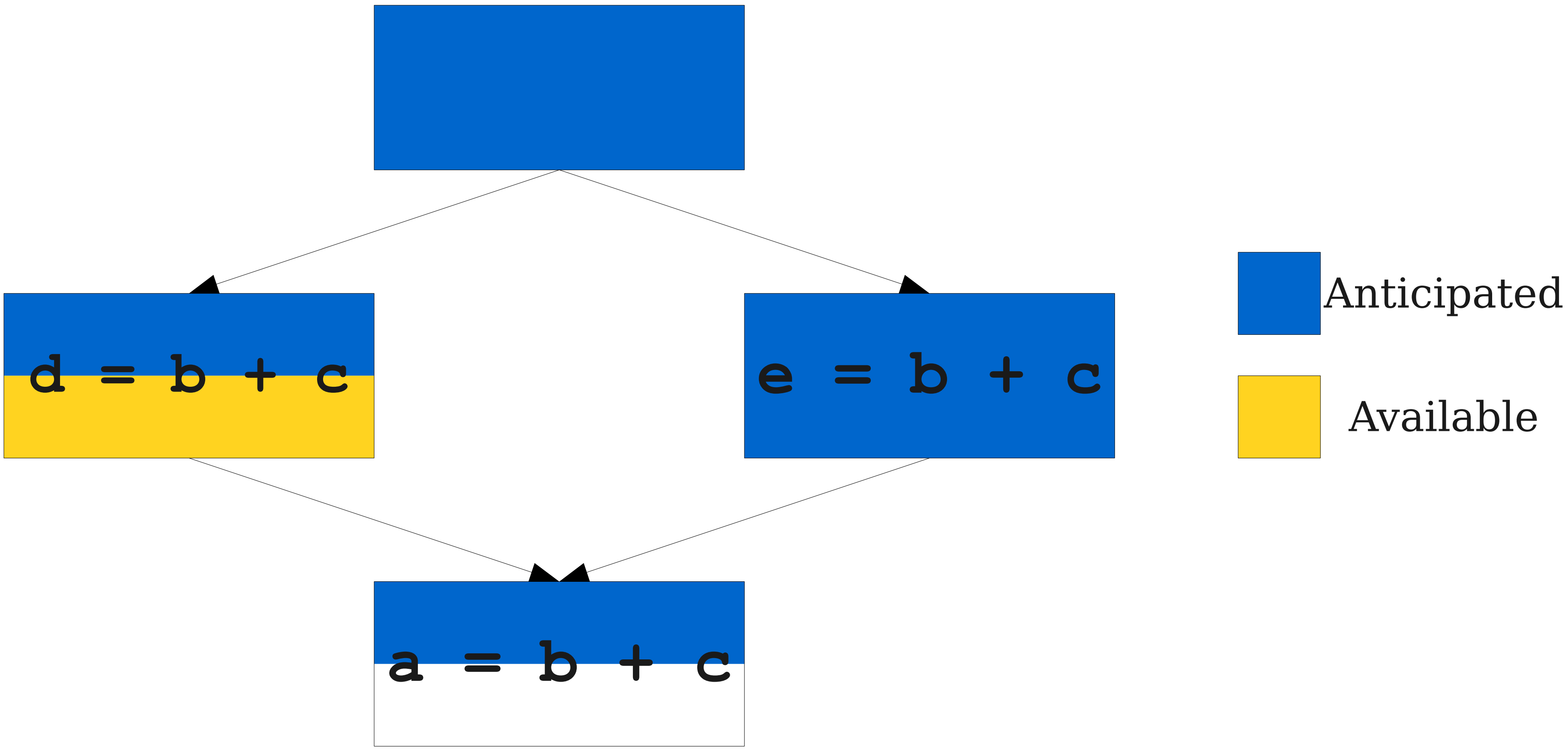
Eliminating Redundancy II



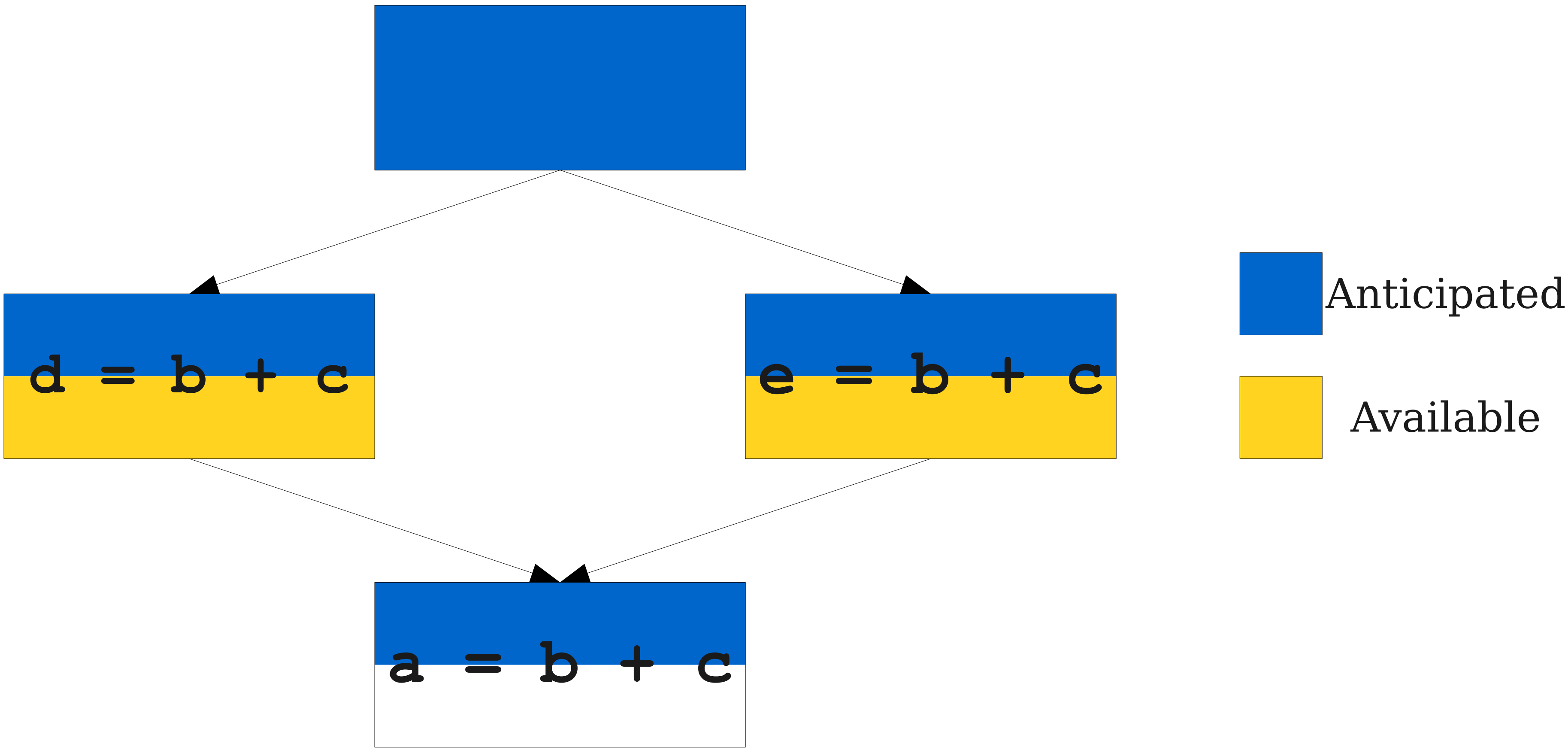
Eliminating Redundancy II



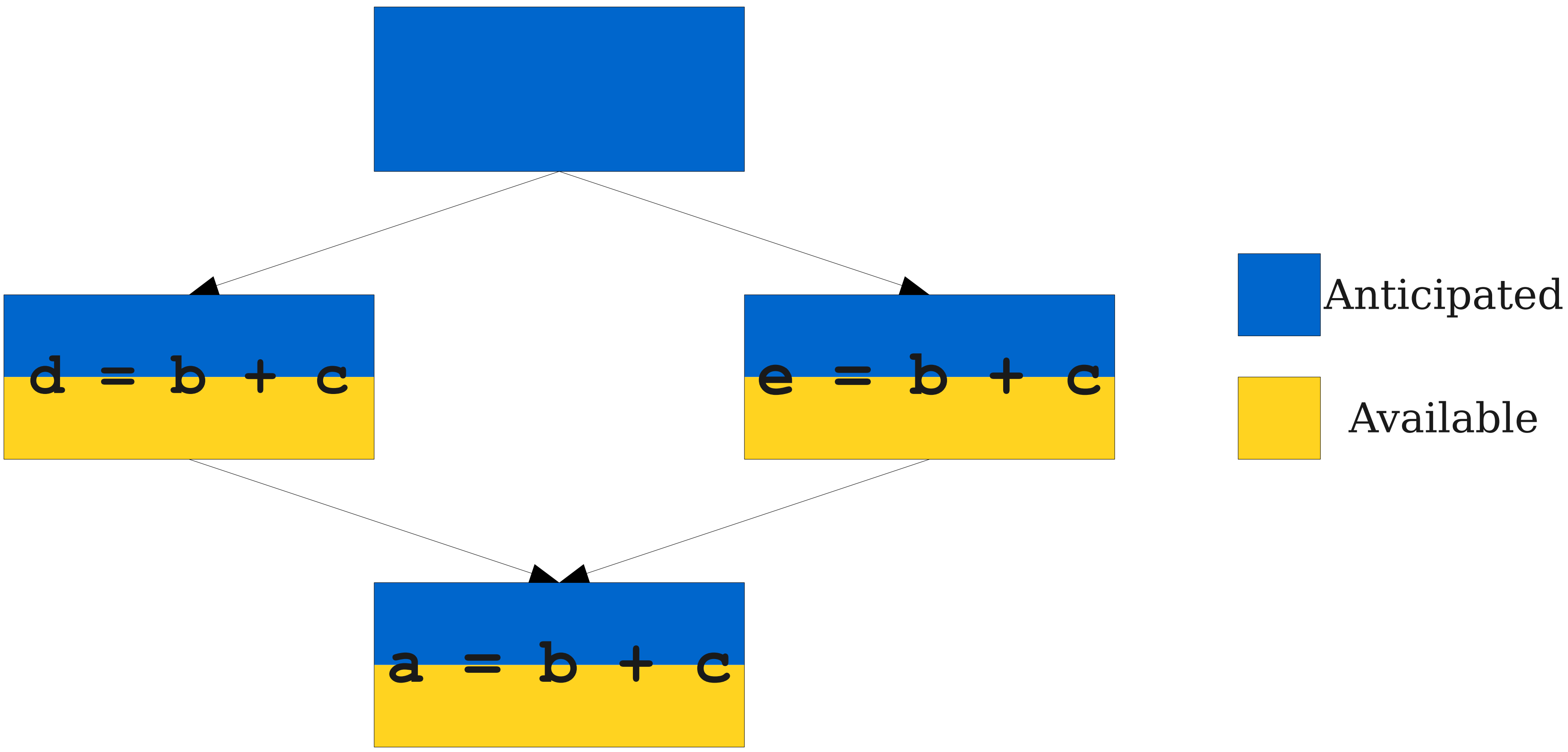
Eliminating Redundancy II



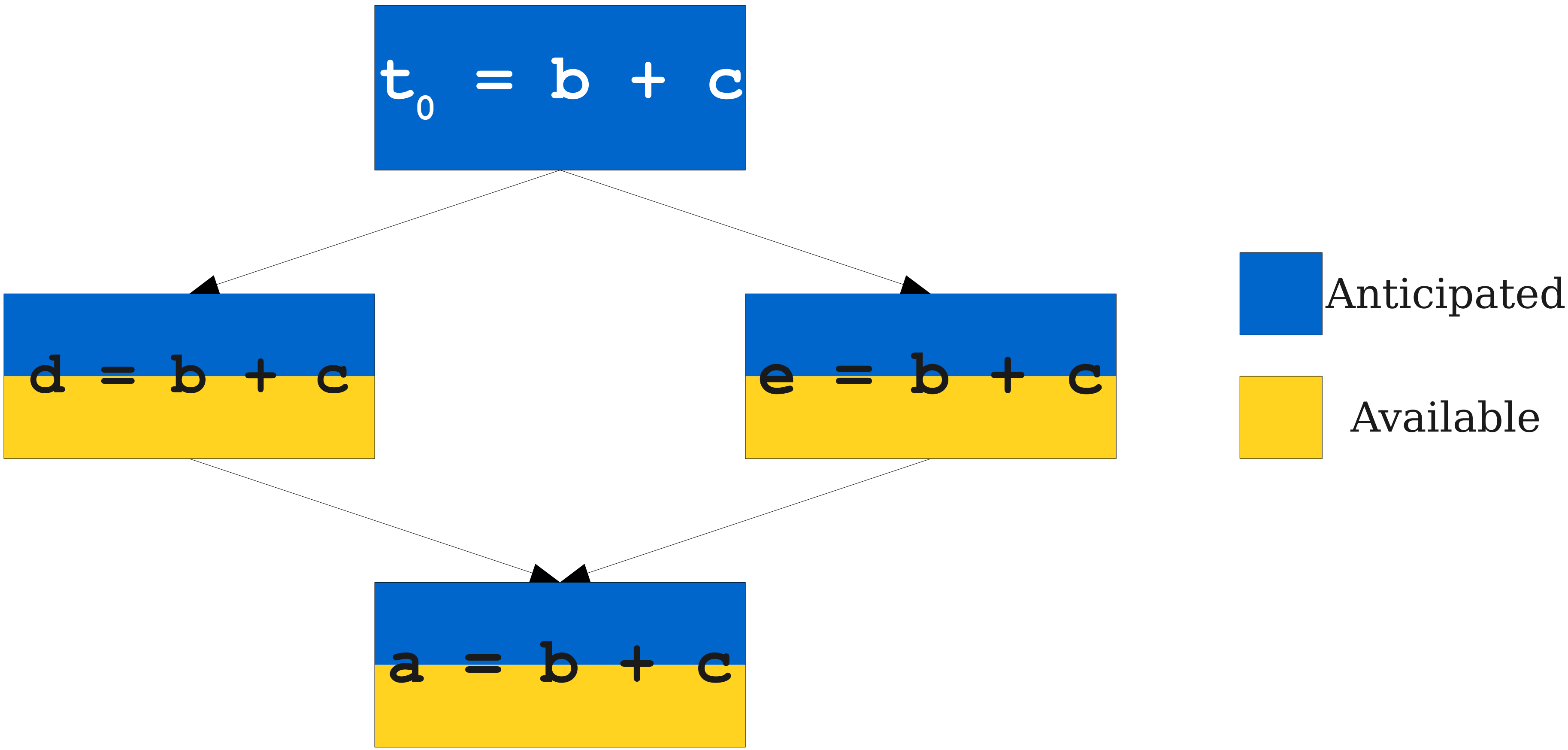
Eliminating Redundancy II



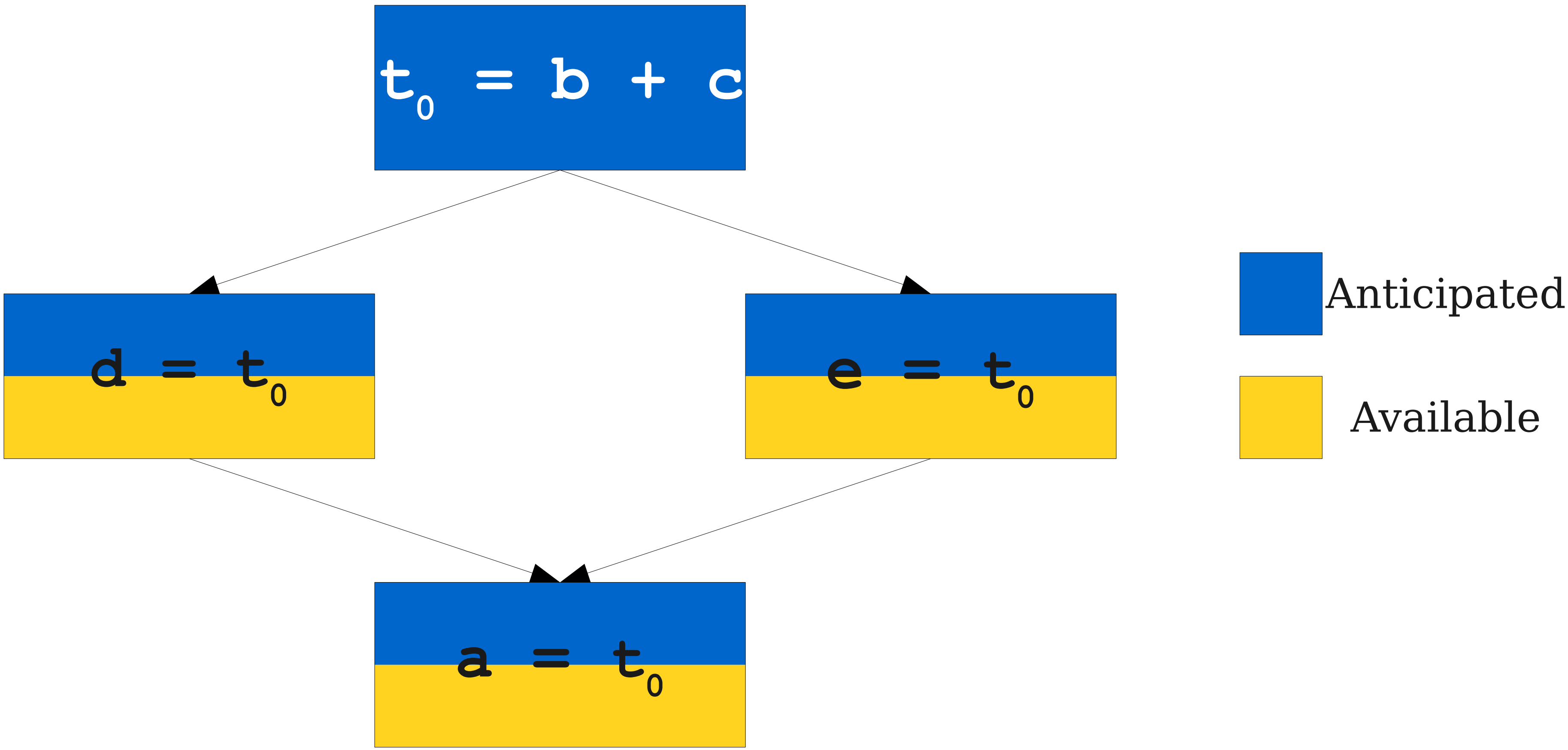
Eliminating Redundancy II



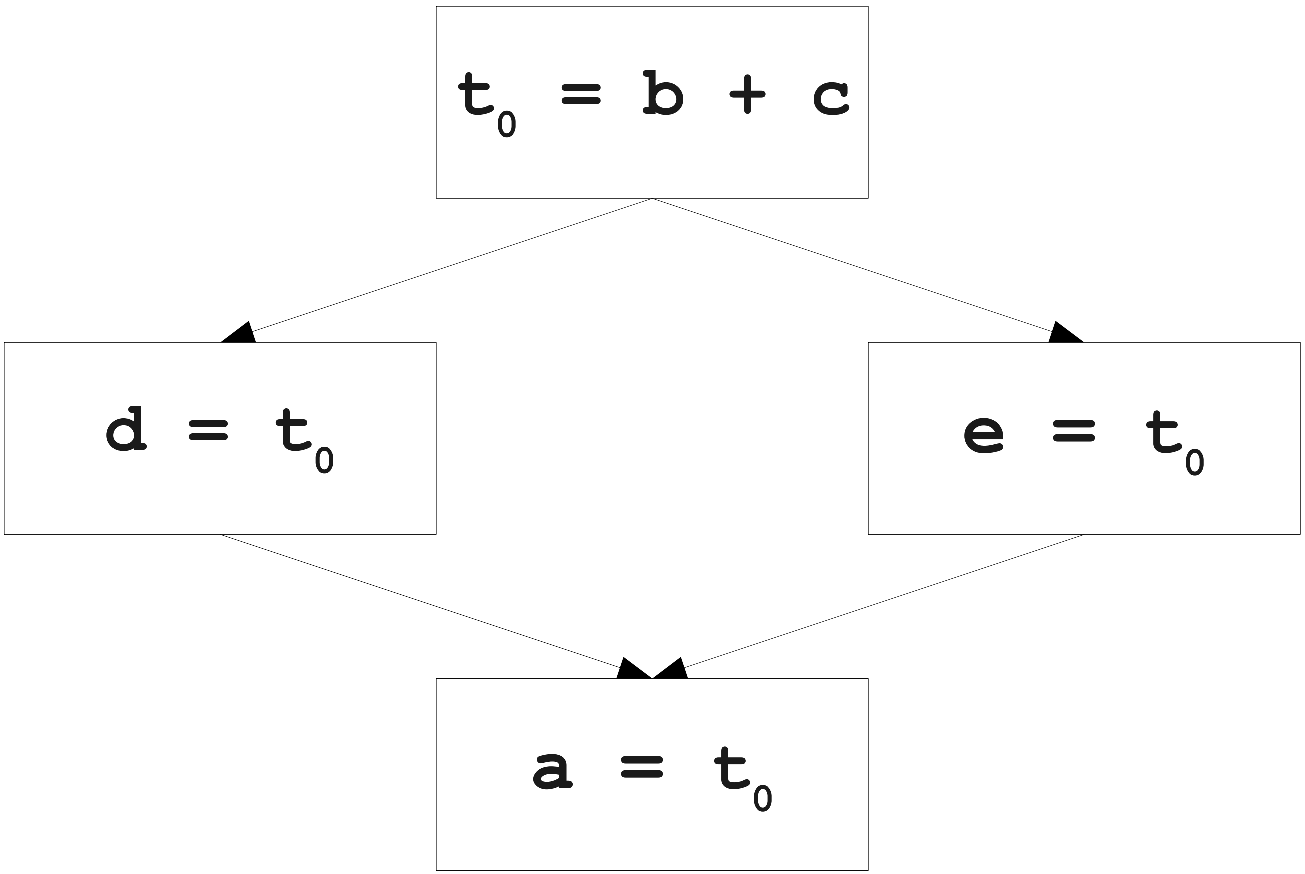
Eliminating Redundancy II



Eliminating Redundancy II

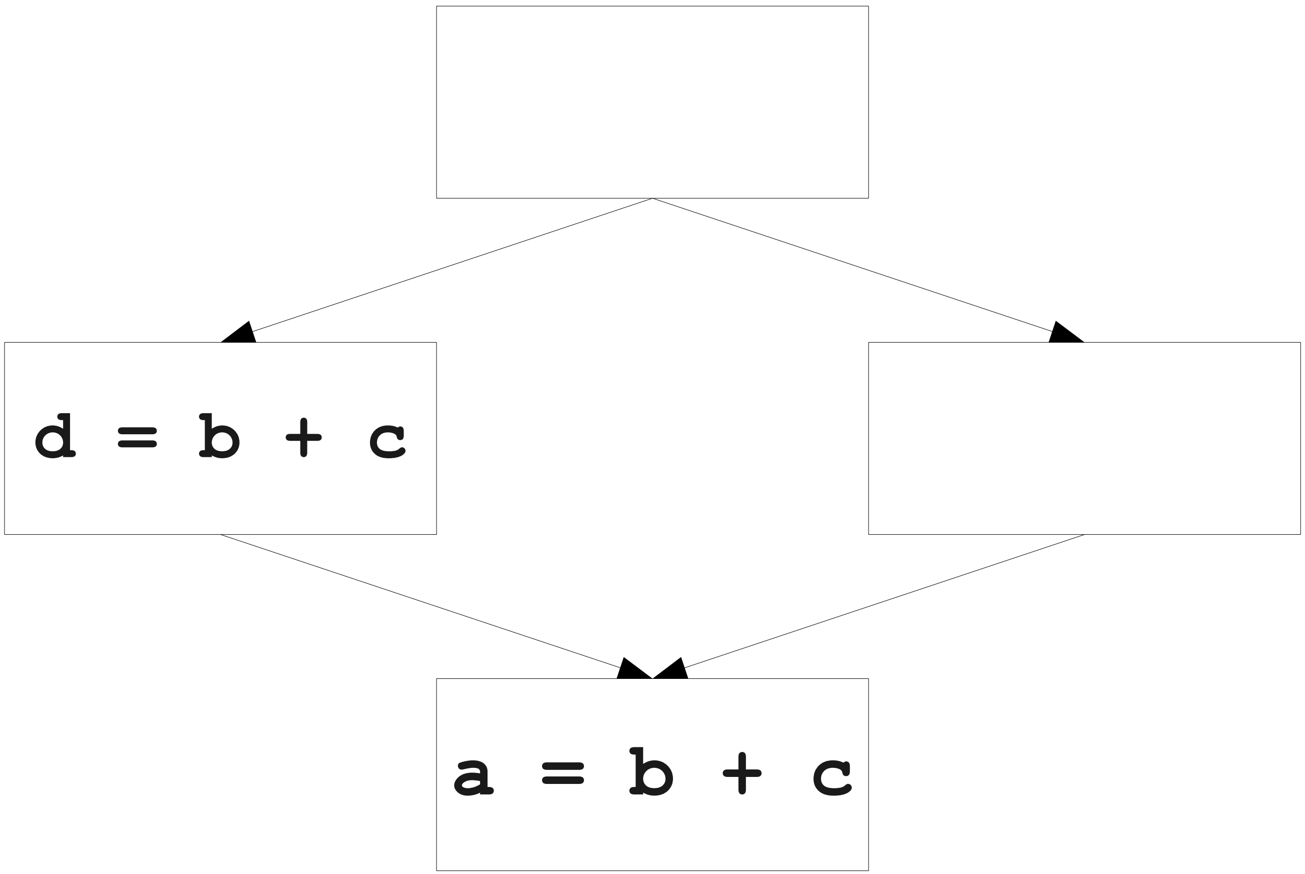


Eliminating Redundancy II

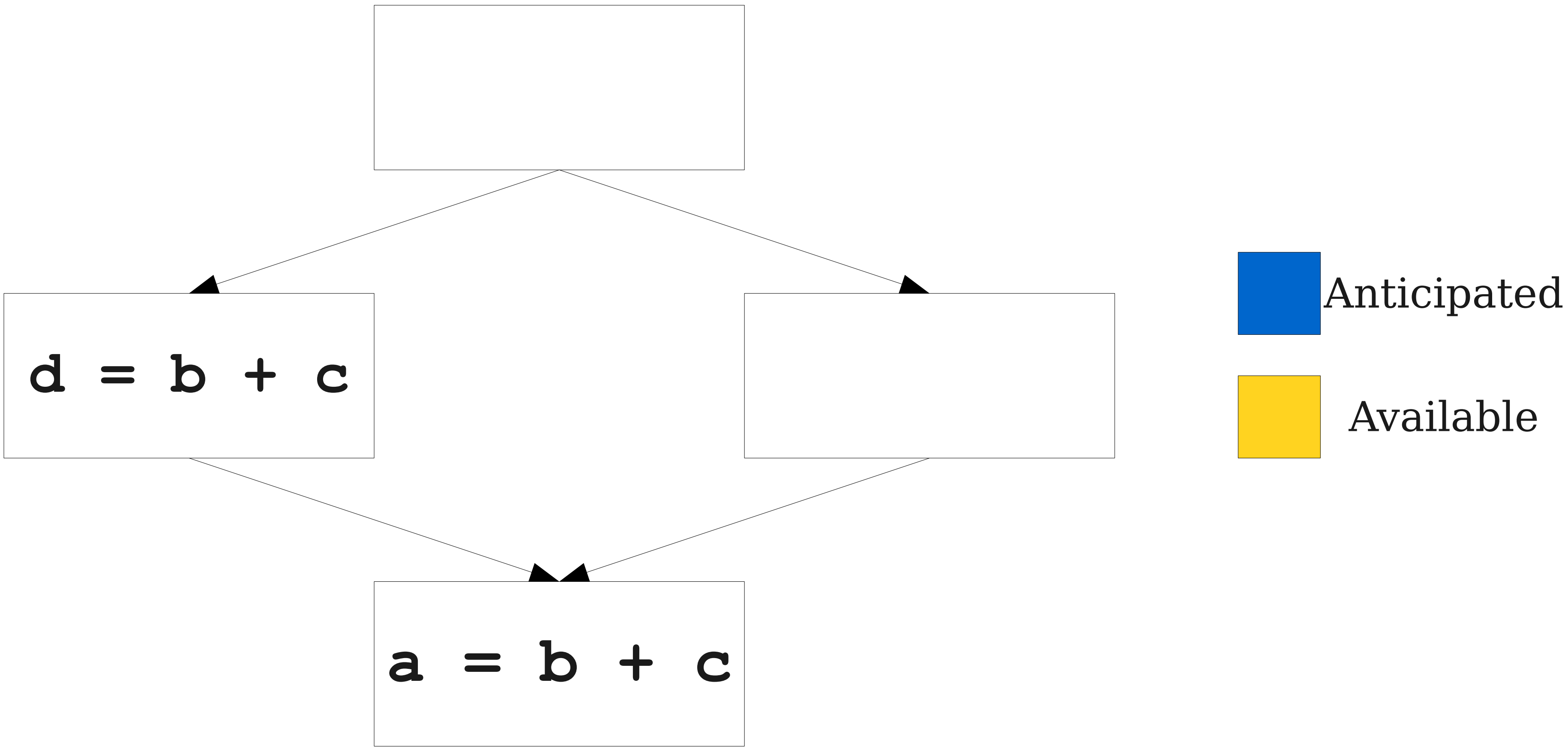


Eliminating Redundancy III

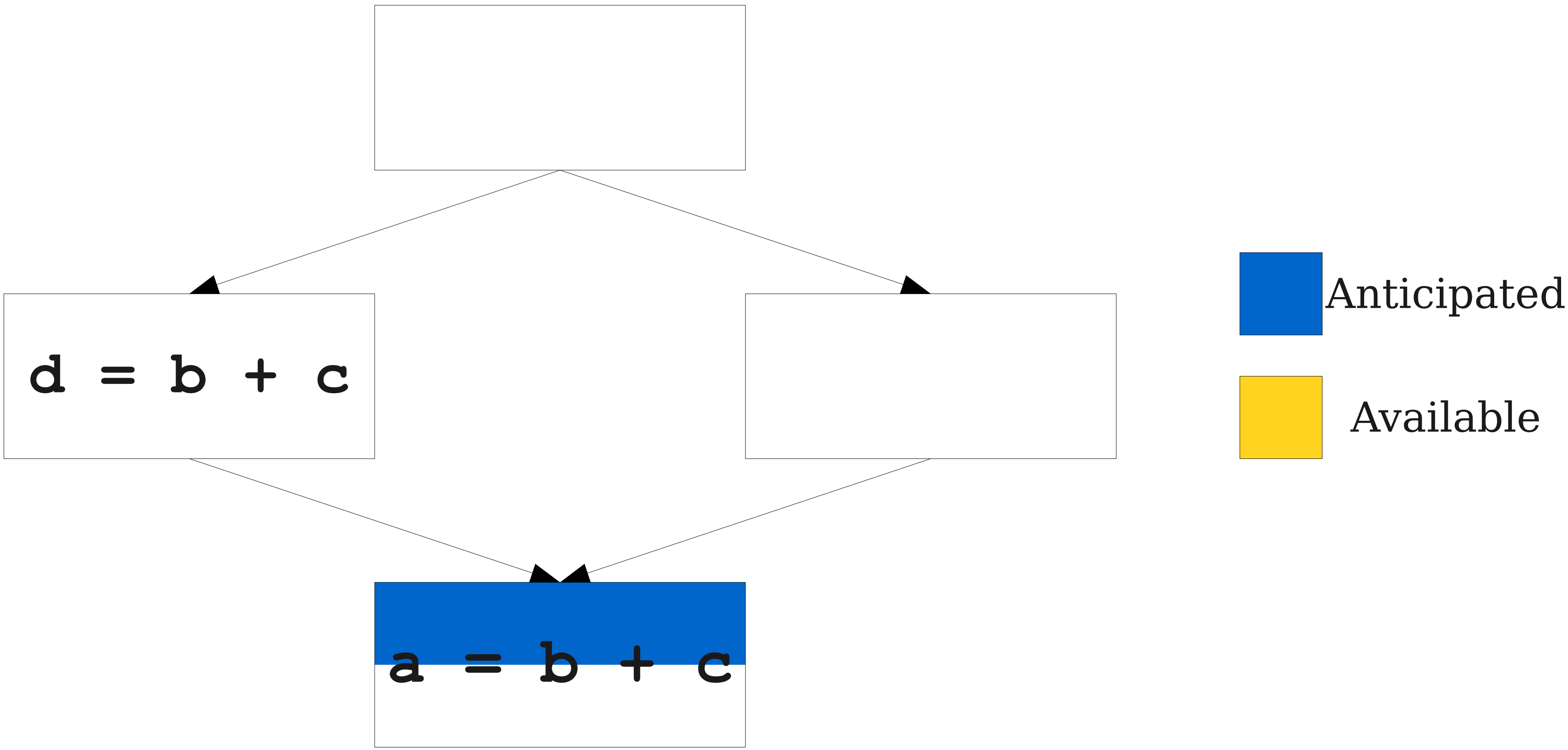
Eliminating Redundancy III



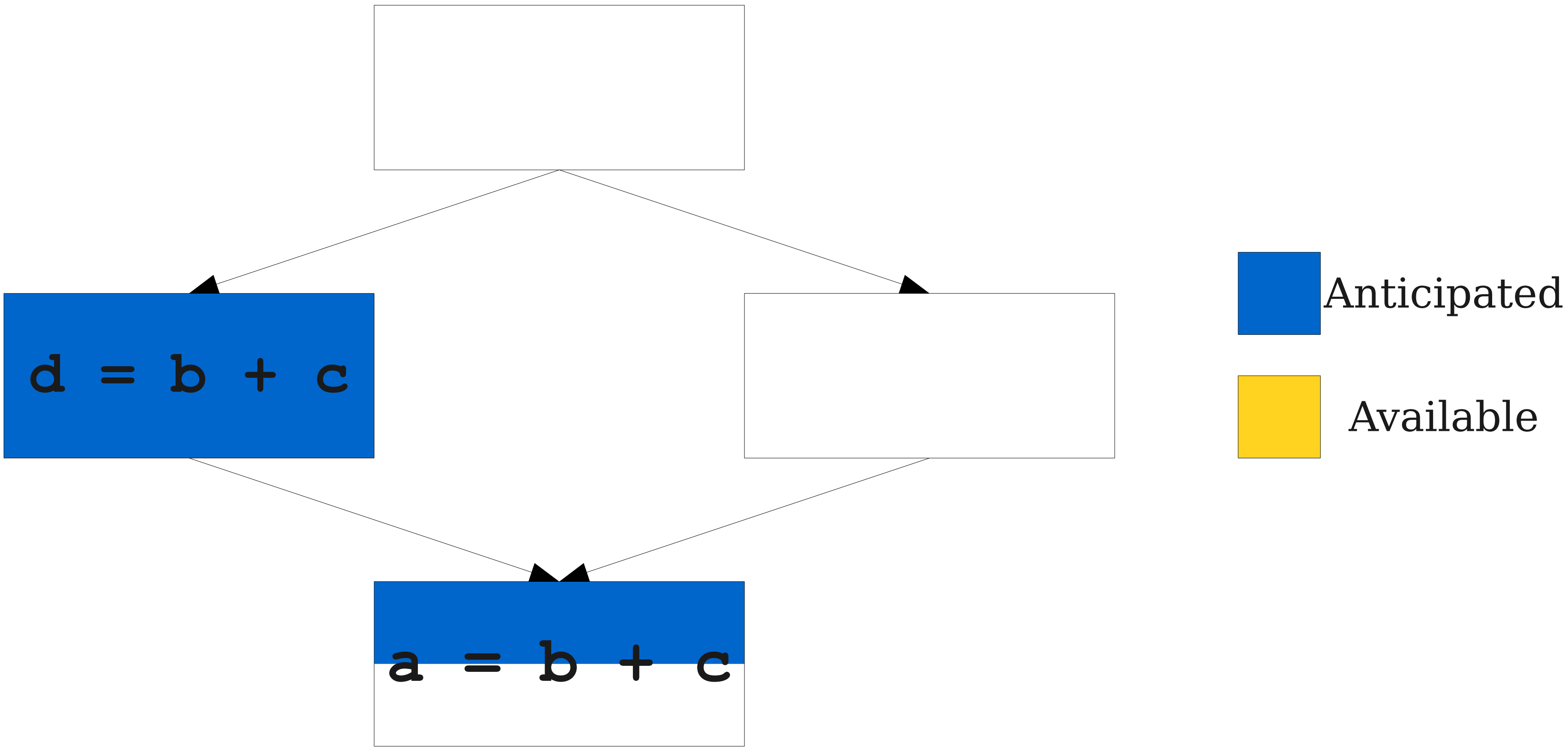
Eliminating Redundancy III



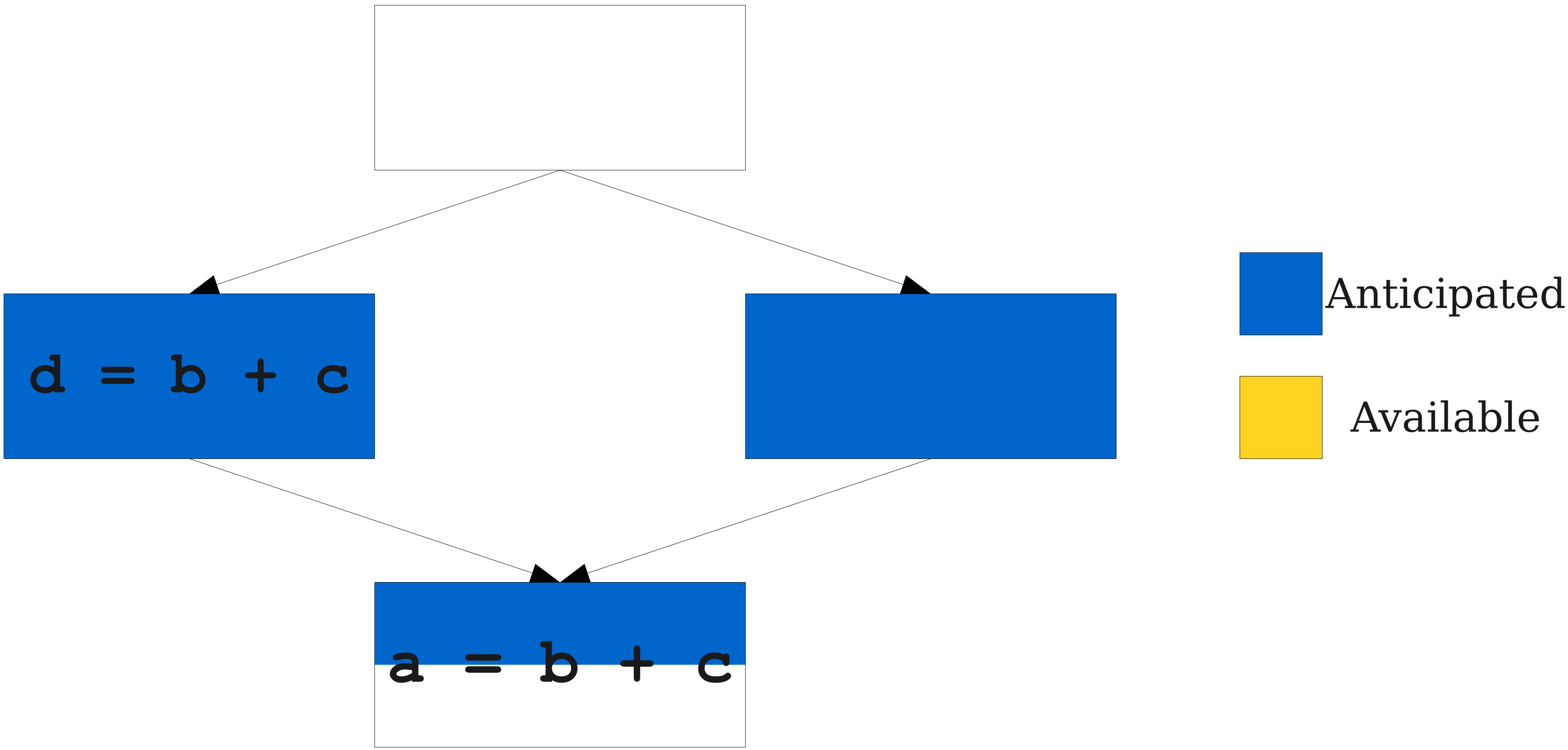
Eliminating Redundancy III



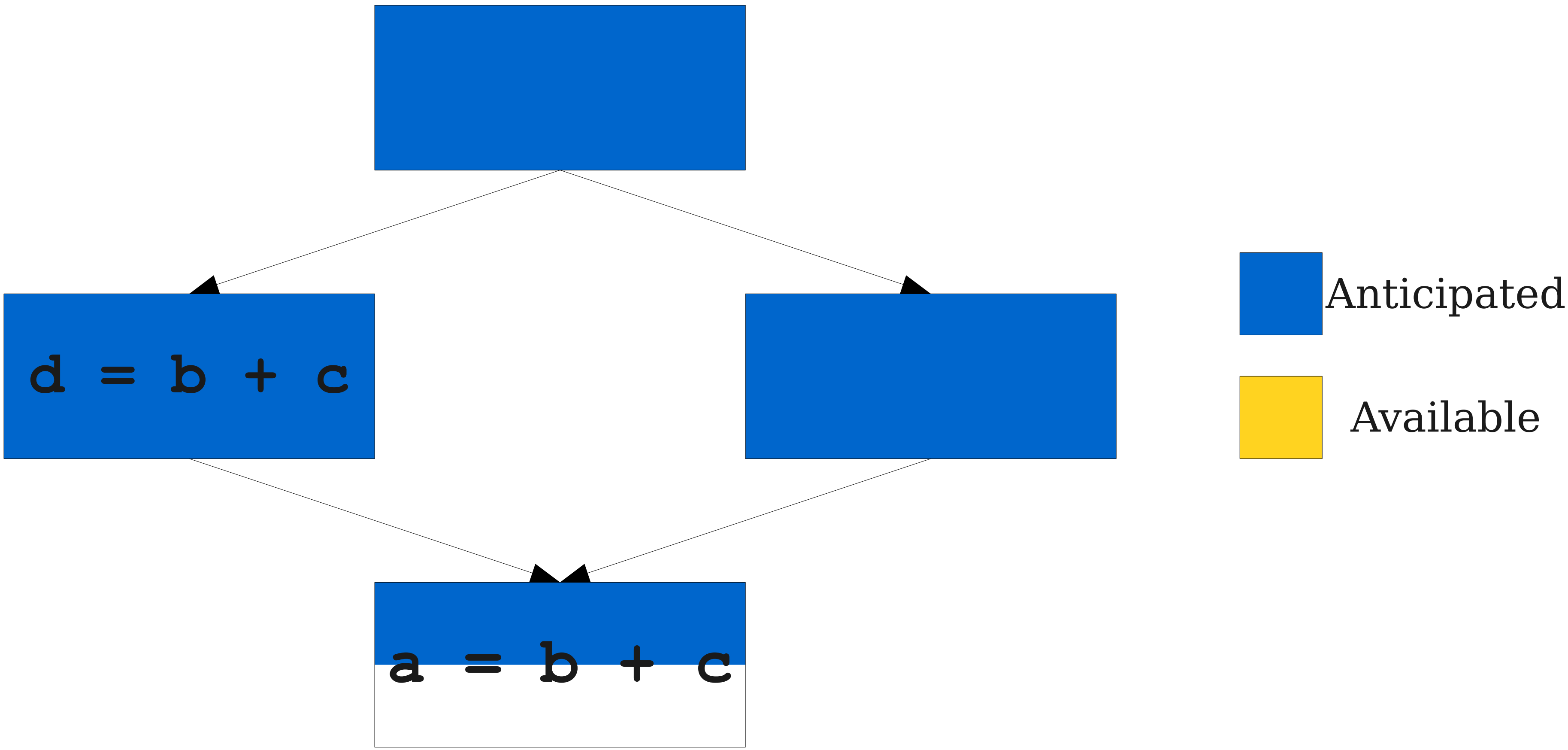
Eliminating Redundancy III



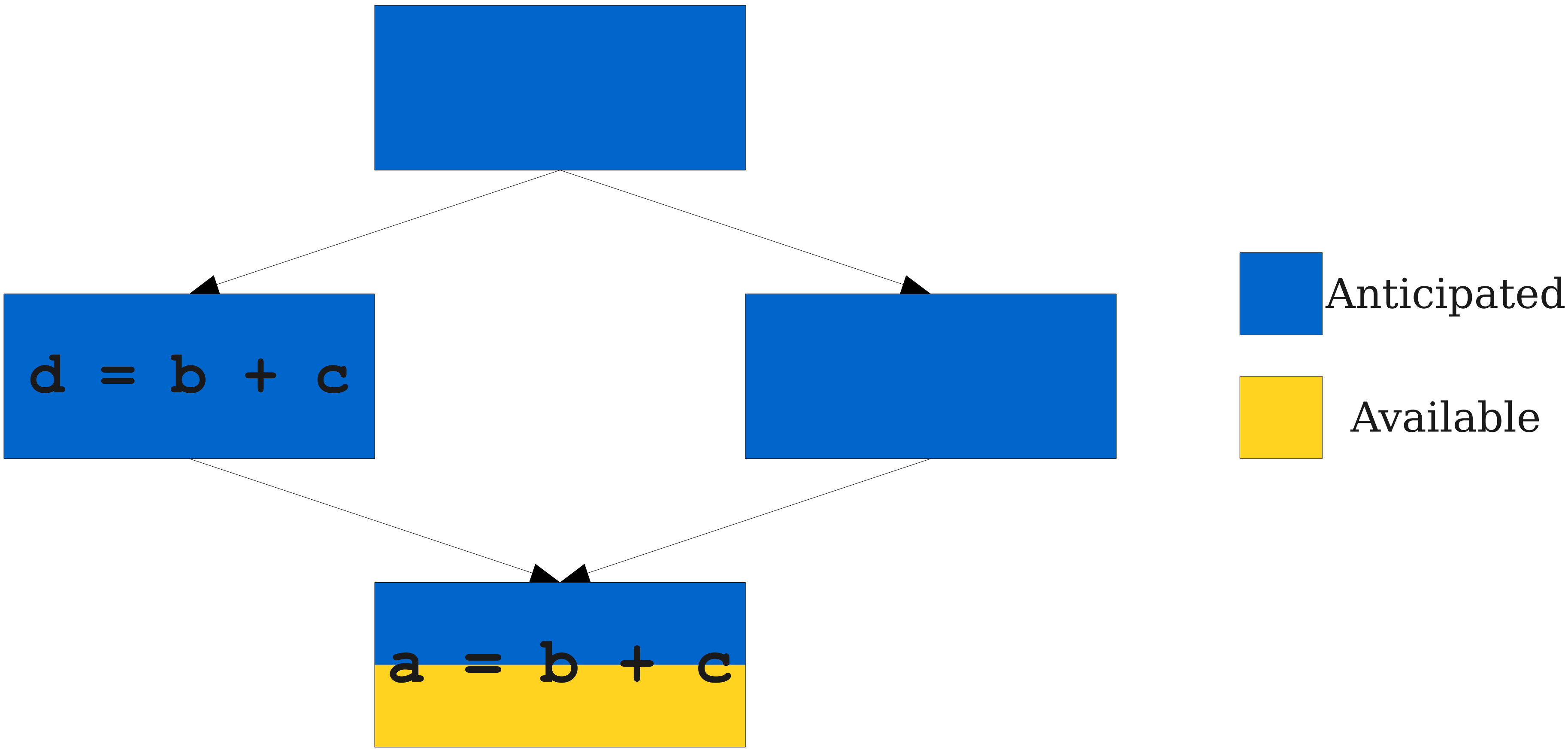
Eliminating Redundancy III



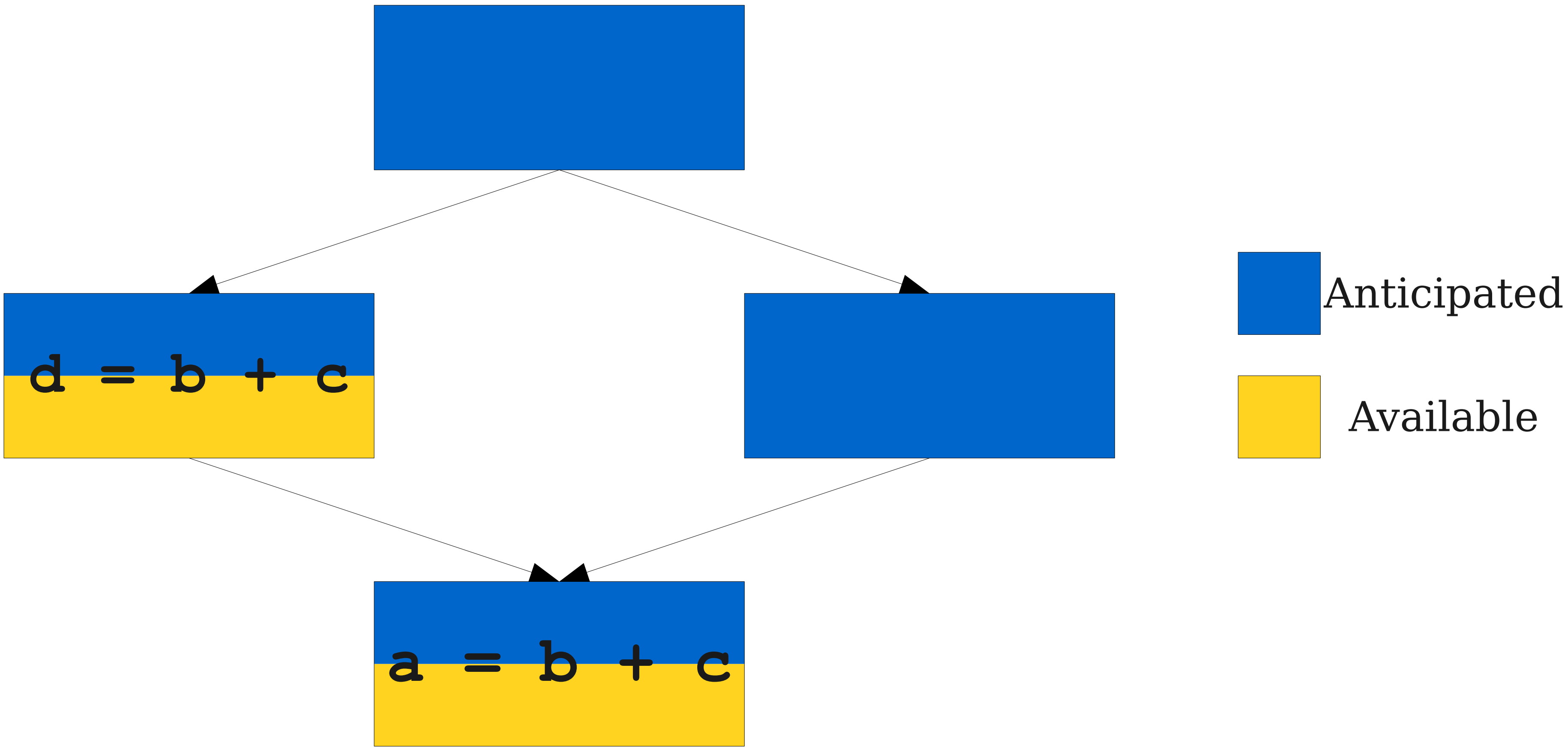
Eliminating Redundancy III



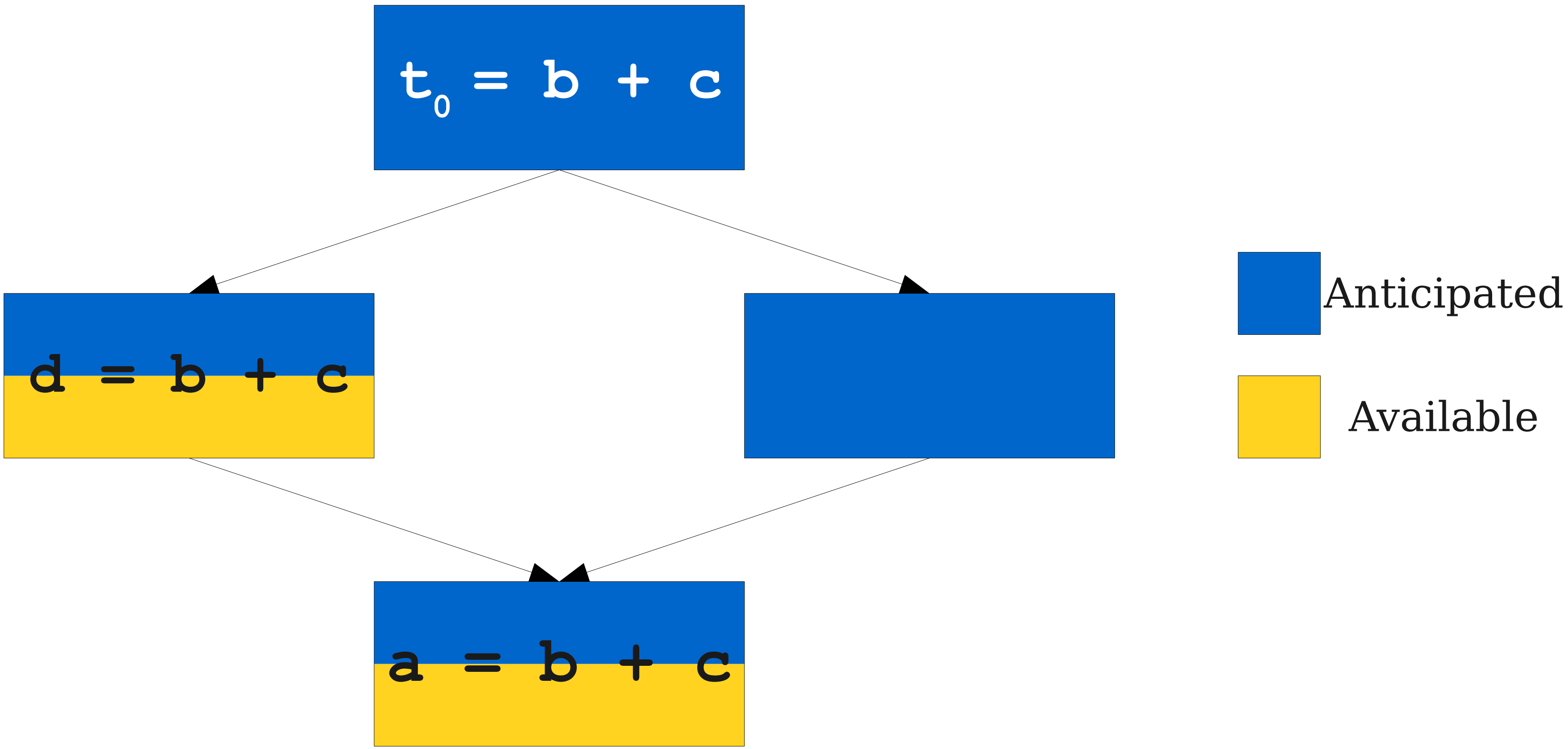
Eliminating Redundancy III



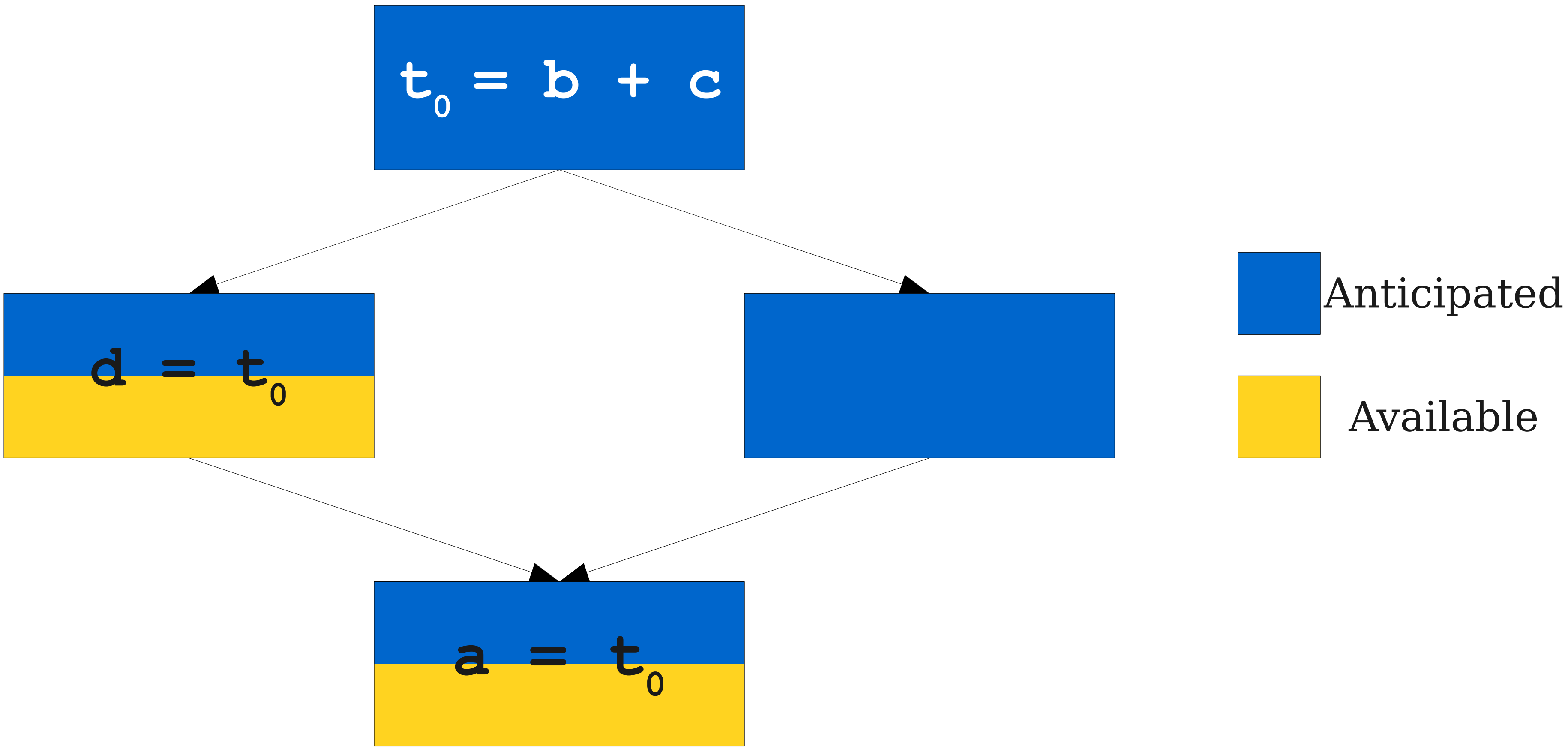
Eliminating Redundancy III



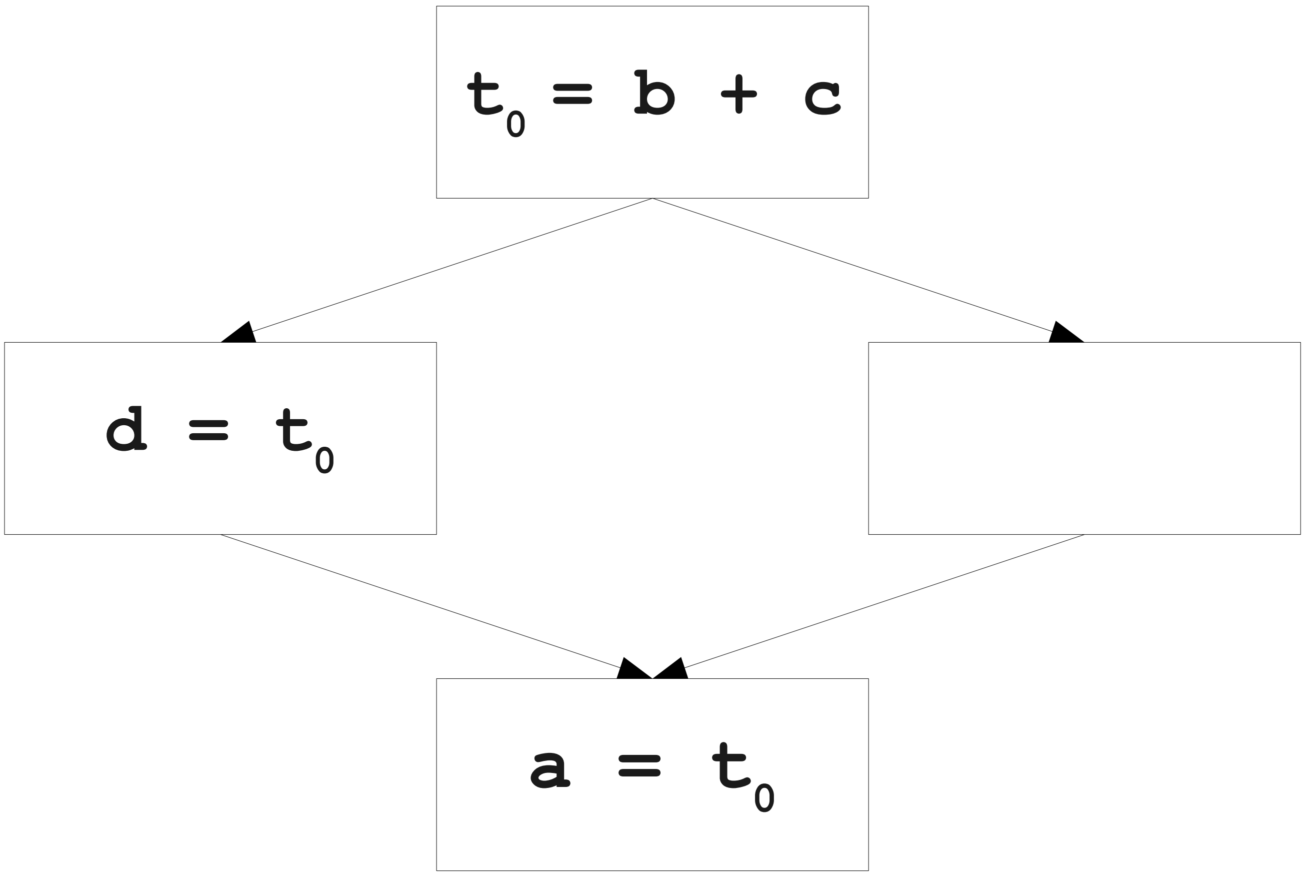
Eliminating Redundancy III



Eliminating Redundancy III



Eliminating Redundancy III



Partial Redundancy Elimination

- Powerful optimization; handles a huge number of disparate cases.
- Subsumes common subexpression elimination, loop invariant code motion, full redundancy elimination, and copy propagation.
- Almost all compilers do this.

In Practice

- Partial-redundancy elimination is typically implemented using *four* dataflow analyses.
  - Pass one: Determine where anticipated.
  - Pass two: Determine where available.
  - Pass three: Find best placement.
  - Pass four: Cleanup unnecessary temporaries.
- A bit more complex than what we covered:
  - Have to add basic blocks at some points.
  - For very complex CFGs, might miss some redundancy.
- See Dragon Book, Ch. 9.5 for more details.

Summary

- The **dataflow framework** gives a unified framework for defining global analyses.
- All of the following analyses can be formulated in the dataflow framework:
  - Global dead code elimination.
  - Global constant propagation.
  - Partial redundancy elimination.
- **Meet semilattices** give a way of describing how to initialize the analysis and combine intermediate results.
- **Monotone transfer functions**, combined with **finite-height lattices**, are necessary to guarantee termination.

Next Time

- **Register Allocation**
  - The memory hierarchy.
  - Naive register allocation.
  - Linear scan allocation.
  - Graph-coloring allocation.