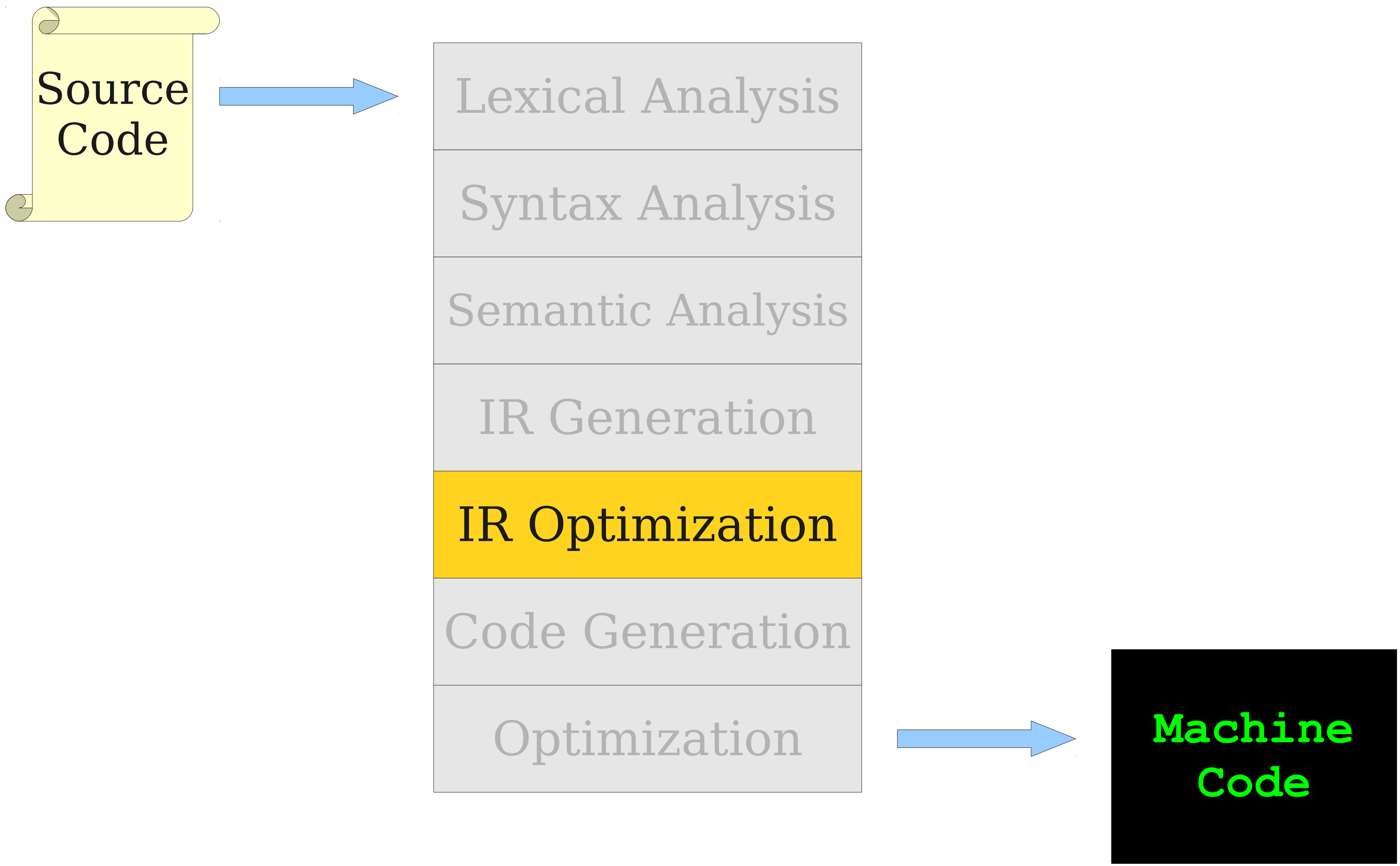
Global Optimization

Where We Are


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

Review of Local Optimization

Review from Last Time

- A **basic block** is a series of IR instructions where
  - there is one entry point into the basic block, and
  - there is one exit point out of the basic block.
- Intuitively, a block of IR instructions that all must execute as a unit.
- A **control-flow graph** (CFG) is a graph of the basic blocks of a function.
- Each edge in a CFG corresponds to a possible flow of control through the program.

Review from Last Time

- A **local optimization** is an optimization of IR instructions within a single basic block.
- We saw five examples of this:
  - **Common subexpression elimination**.
  - **Copy propagation**.
  - **Dead code elimination**.
  - **Arithmetic simplification**.
  - **Constant folding**.

Review from Last Time

- Last time, we defined two analyses used in our optimizations.
- **Available expressions**: Track what variables are assigned which expressions.
  - Compute by walking forward across the values in a basic block.
- **Live variables**: Track what variables will eventually be used.
  - Compute by walking backward across the values in a basic block.

Available Expressions

```
a = b;
c = b;
d = a + b;
e = a + b;
d = b;
f = a + b;
```

Available Expressions

```
{ }
a = b;

c = b;

d = a + b;

e = a + b;

d = b;

f = a + b;
```

Available Expressions

```
{ }
a = b;
{ a = b }
c = b;

d = a + b;

e = a + b;

d = b;

f = a + b;
```

Available Expressions

```
{ }
a = b;
{ a = b }
c = b;
{ a = b, c = b }
d = a + b;

e = a + b;

d = b;

f = a + b;
```

Available Expressions

```
{ }
a = b;
{ a = b }
c = b;
{ a = b, c = b }
d = a + b;
{ a = b, c = b, d = a + b }
e = a + b;

d = b;

f = a + b;
```

Available Expressions

```
{ }
a = b;
{ a = b }
c = b;
{ a = b, c = b }
d = a + b;
{ a = b, c = b, d = a + b }
e = a + b;
{ a = b, c = b, d = a + b, e = a + b }
d = b;

f = a + b;
```

Available Expressions

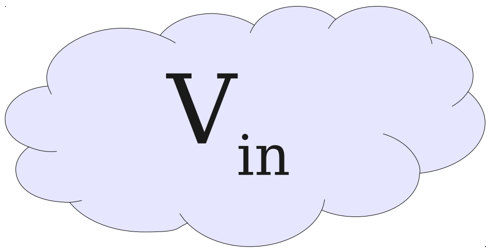
```
{ }
a = b;
{ a = b }
c = b;
{ a = b, c = b }
d = a + b;
{ a = b, c = b, d = a + b }
e = a + b;
{ a = b, c = b, d = a + b, e = a + b }
d = b;
{ a = b, c = b, d = b, e = a + b }
f = a + b;
```

Available Expressions

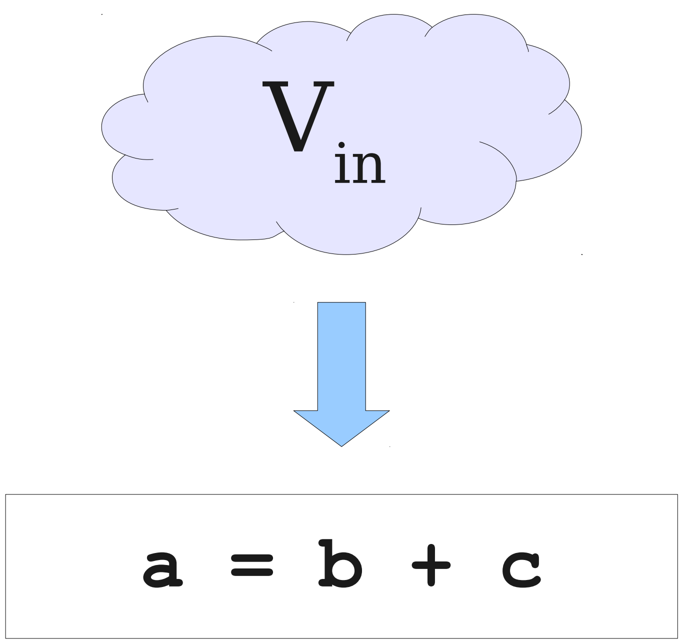
```
{ }
a = b;
{ a = b }
c = b;
{ a = b, c = b }
d = a + b;
{ a = b, c = b, d = a + b }
e = a + b;
{ a = b, c = b, d = a + b, e = a + b }
d = b;
{ a = b, c = b, d = b, e = a + b }
f = a + b;
{ a = b, c = b, d = b, e = a + b, f = a + b }
```

Another View of Local Analyses

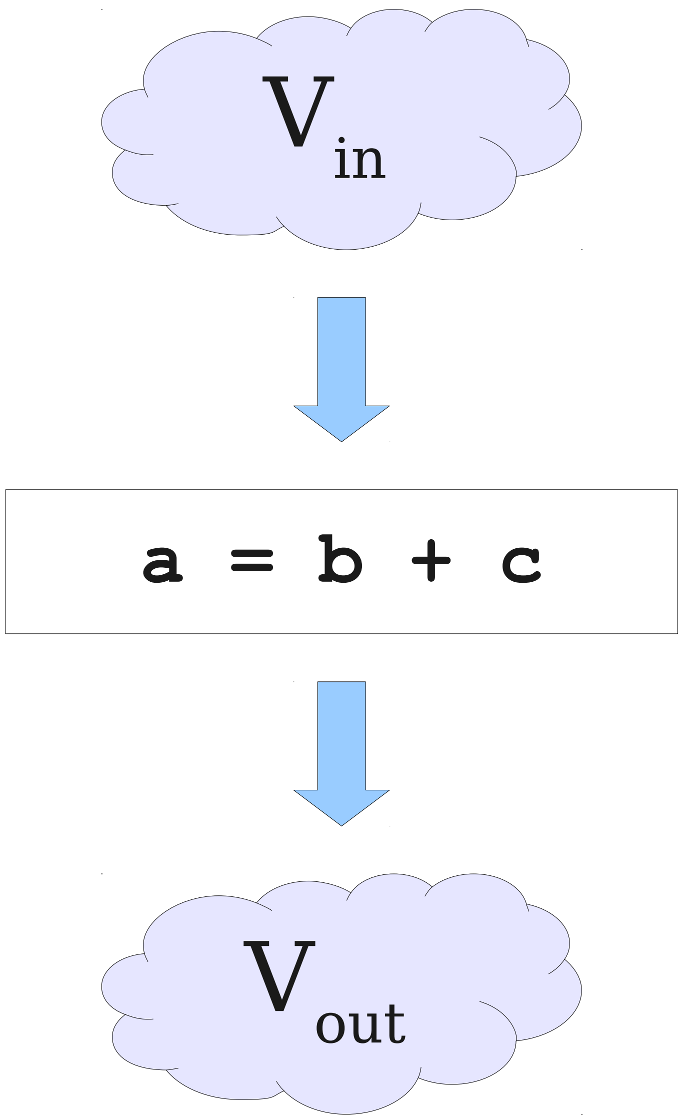
Another View of Local Analyses



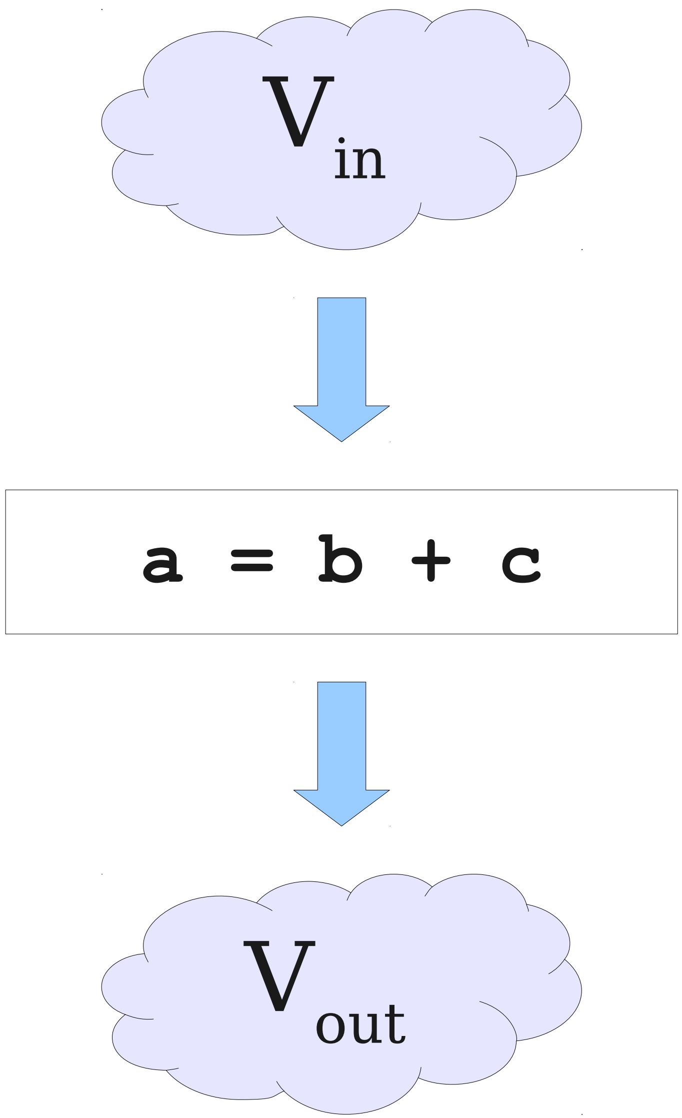
Another View of Local Analyses



Another View of Local Analyses



Another View of Local Analyses



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

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.

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 $f : \mathbf{V} \to \mathbf{V}$.
  - **I** is the initial information at the top (or bottom) of a basic block.

Available Expressions

```
a = b;
c = b;
d = a + b;
e = a + b;
d = b;
f = a + b;
```

Available Expressions

```
{ }
a = b;

c = b;

d = a + b;

e = a + b;

d = b;

f = a + b;
```

Available Expressions

```
{ }
a = b;
{ a = b }
c = b;

d = a + b;

e = a + b;

d = b;

f = a + b;
```

Available Expressions

```
{ }
a = b;
{ a = b }
c = b;
{ a = b, c = b }
d = a + b;

e = a + b;

d = b;

f = a + b;
```

Available Expressions

```
{ }
a = b;
{ a = b }
c = b;
{ a = b, c = b }
d = a + b;
{ a = b, c = b, d = a + b }
e = a + b;

d = b;

f = a + b;
```

Available Expressions

```
{ }
a = b;
{ a = b }
c = b;
{ a = b, c = b }
d = a + b;
{ a = b, c = b, d = a + b }
e = a + b;
{ a = b, c = b, d = a + b, e = a + b }
d = b;

f = a + b;
```

Available Expressions

```
{ }
a = b;
{ a = b }
c = b;
{ a = b, c = b }
d = a + b;
{ a = b, c = b, d = a + b }
e = a + b;
{ a = b, c = b, d = a + b, e = a + b }
d = b;
{ a = b, c = b, d = b, e = a + b }
f = a + b;
```

Available Expressions

{ }

`a = b;`

{ a = b }

`c = b;`

{ a = b, c = b }

`d = a + b;`

{ a = b, c = b, d = a + b }

`e = a + b;`

{ a = b, c = b, d = a + b, e = a + b }

`d = b;`

{ a = b, c = b, d = b, e = a + b }

`f = a + b;`

{ a = b, c = b, d = b, e = a + b, f = a + b }

Available Expressions

- **Direction**: Forward
- **Domain**: 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**.
- **Initial value**: Empty set of expressions.

Liveness Analysis

```
a = b;

c = a;

d = a + b;

e = d;

d = a;

f = e;
```

Liveness Analysis

```
a = b;

c = a;

d = a + b;

e = d;

d = a;

f = e;
```

{ b, d }

Liveness Analysis

```
a = b;

c = a;

d = a + b;

e = d;

d = a;
{ b, d, e }
f = e;
{ b, d }
```

Liveness Analysis

```
a = b;

c = a;

d = a + b;

e = d;
{ a, b, e }
d = a;
{ b, d, e }
f = e;
{ b, d }
```

Liveness Analysis

```
a = b;

c = a;

d = a + b;
{ a, b, d }
e = d;
{ a, b, e }
d = a;
{ b, d, e }
f = e;
{ b, d }
```

Liveness Analysis

```
a = b;

c = a;
```
`{ a, b }`
```
d = a + b;
```
`{ a, b, d }`
```
e = d;
```
`{ a, b, e }`
```
d = a;
```
`{ b, d, e }`
```
f = e;
```
`{ b, d }`

Liveness Analysis

```
a = b;
{ a, b }
c = a;
{ a, b }
d = a + b;
{ a, b, d }
e = d;
{ a, b, e }
d = a;
{ b, d, e }
f = e;
{ b, d }
```

Liveness Analysis

```
{ b }
a = b;
{ a, b }
c = a;
{ a, b }
d = a + b;
{ a, b, d }
e = d;
{ a, b, e }
d = a;
{ b, d, e }
f = e;
{ b, d }
```

Liveness Analysis

- **Direction**: Backwards
- **Domain**: Sets of variables.
- **Transfer function**: Given a set of variables *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: $f_{a = b + c}(V) = (V - \{\mathbf{a}\}) \cup \{\mathbf{b}, \mathbf{c}\}$
- **Initial value**: Depends on semantics of language.

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

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?)

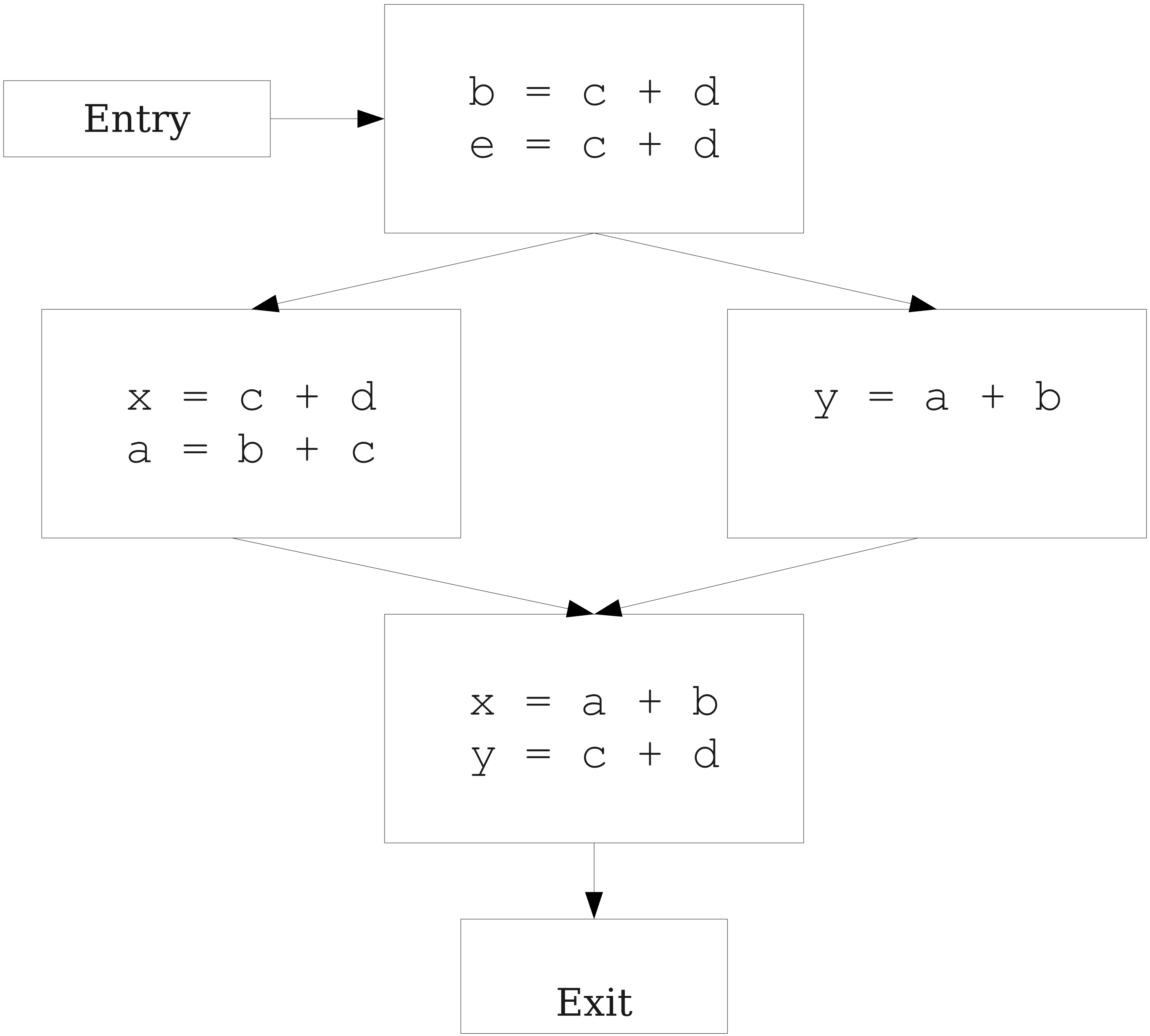
Local vs. Global Analysis

- Many of the optimizations from local analysis can still be applied globally.
  - We'll see how to do this later today.
- 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.
- We'll explore three analyses in detail:
  - **Global dead code elimination**.
  - **Global constant propagation**.
  - **Partial redundancy elimination**.

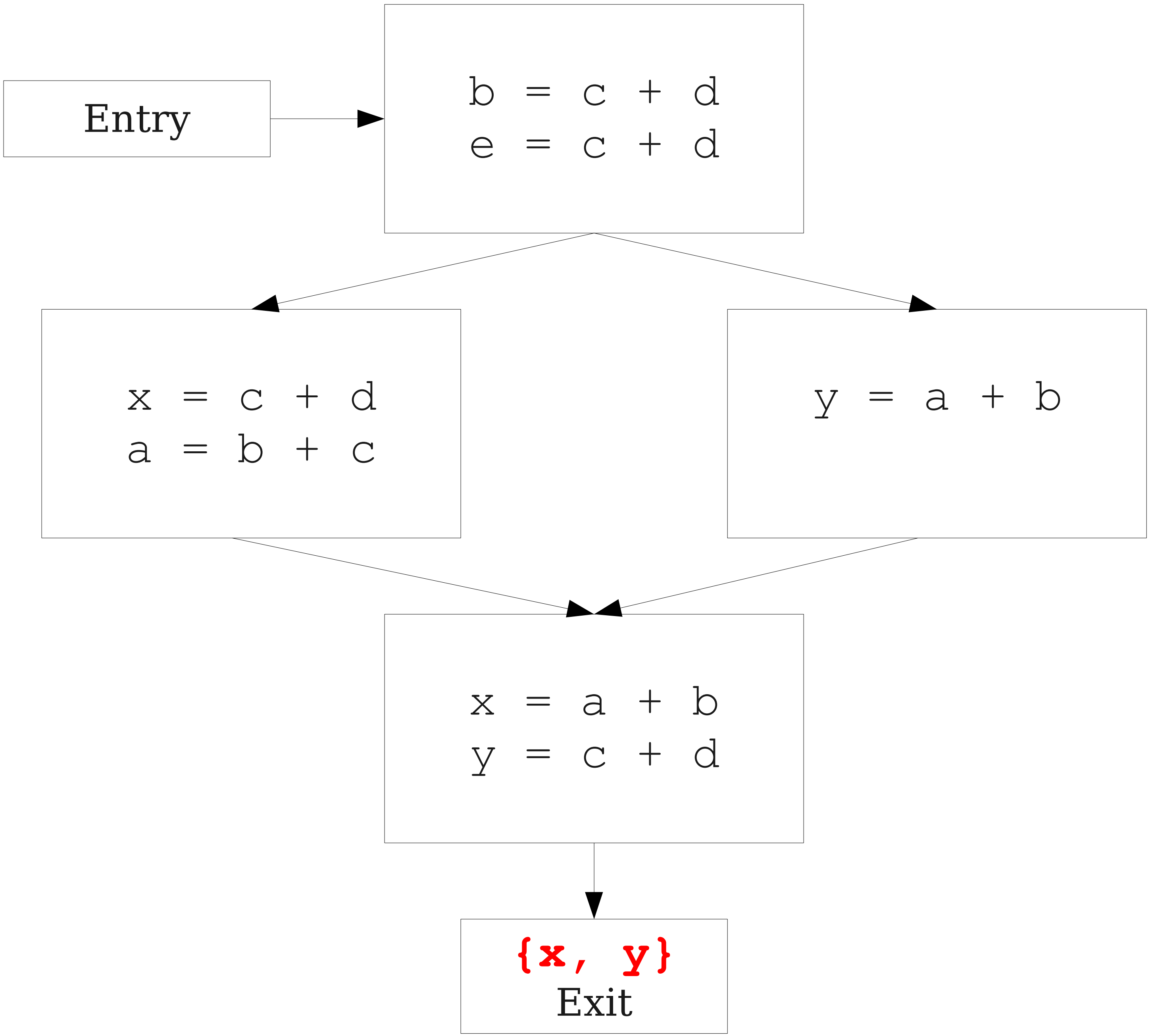
Global Dead Code Elimination

- 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?

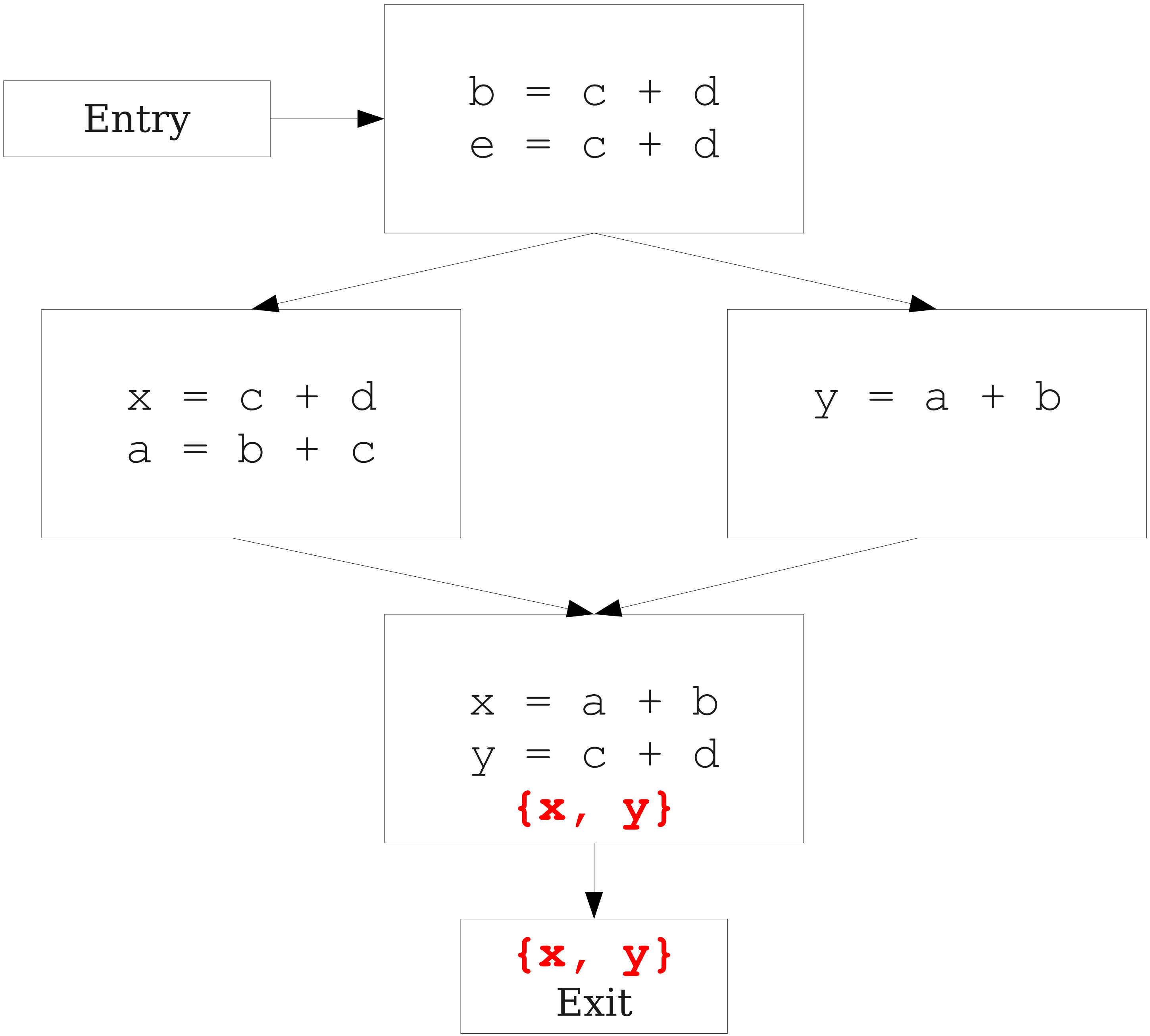
CFGs Without Loops



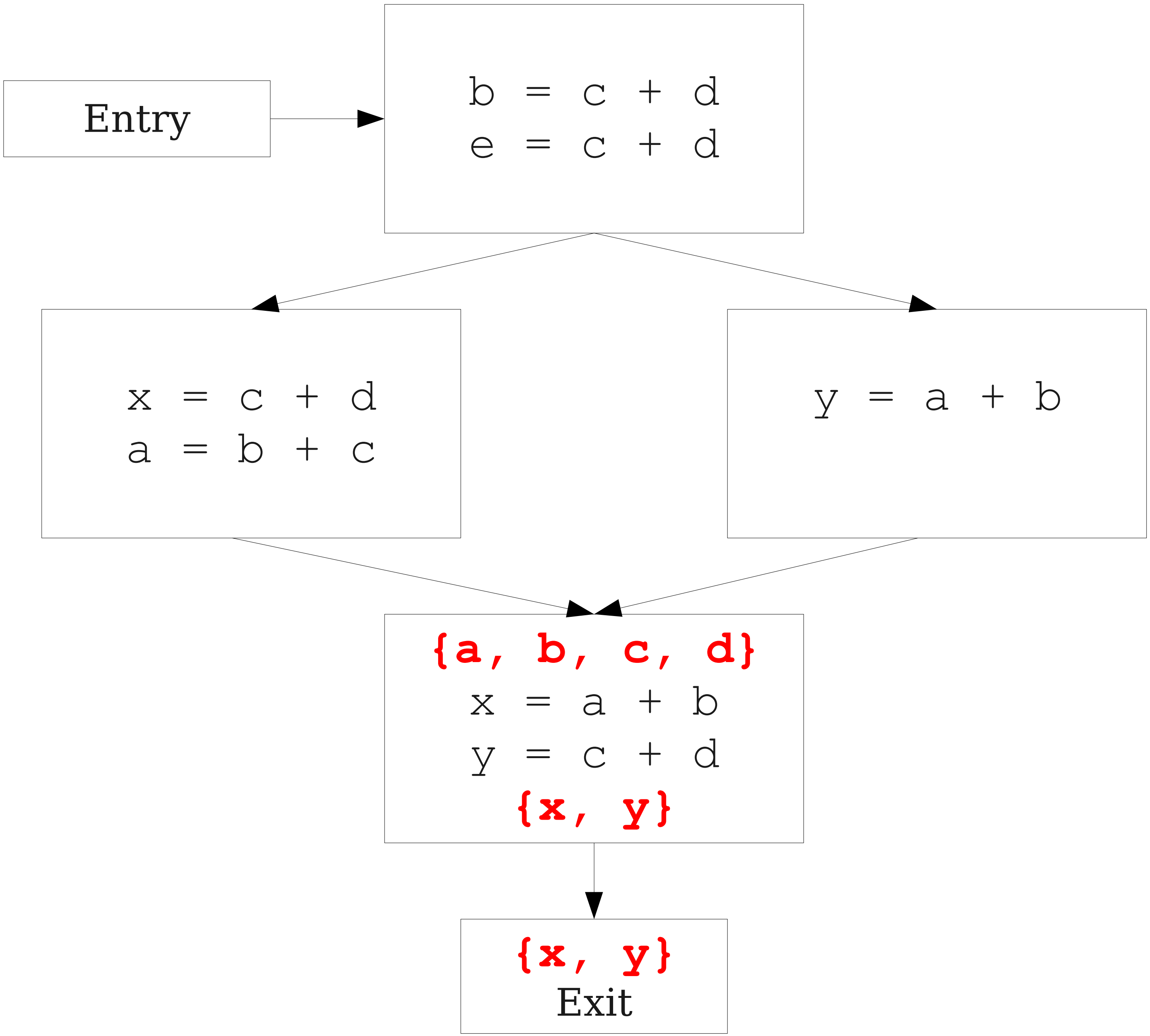
CFGs Without Loops



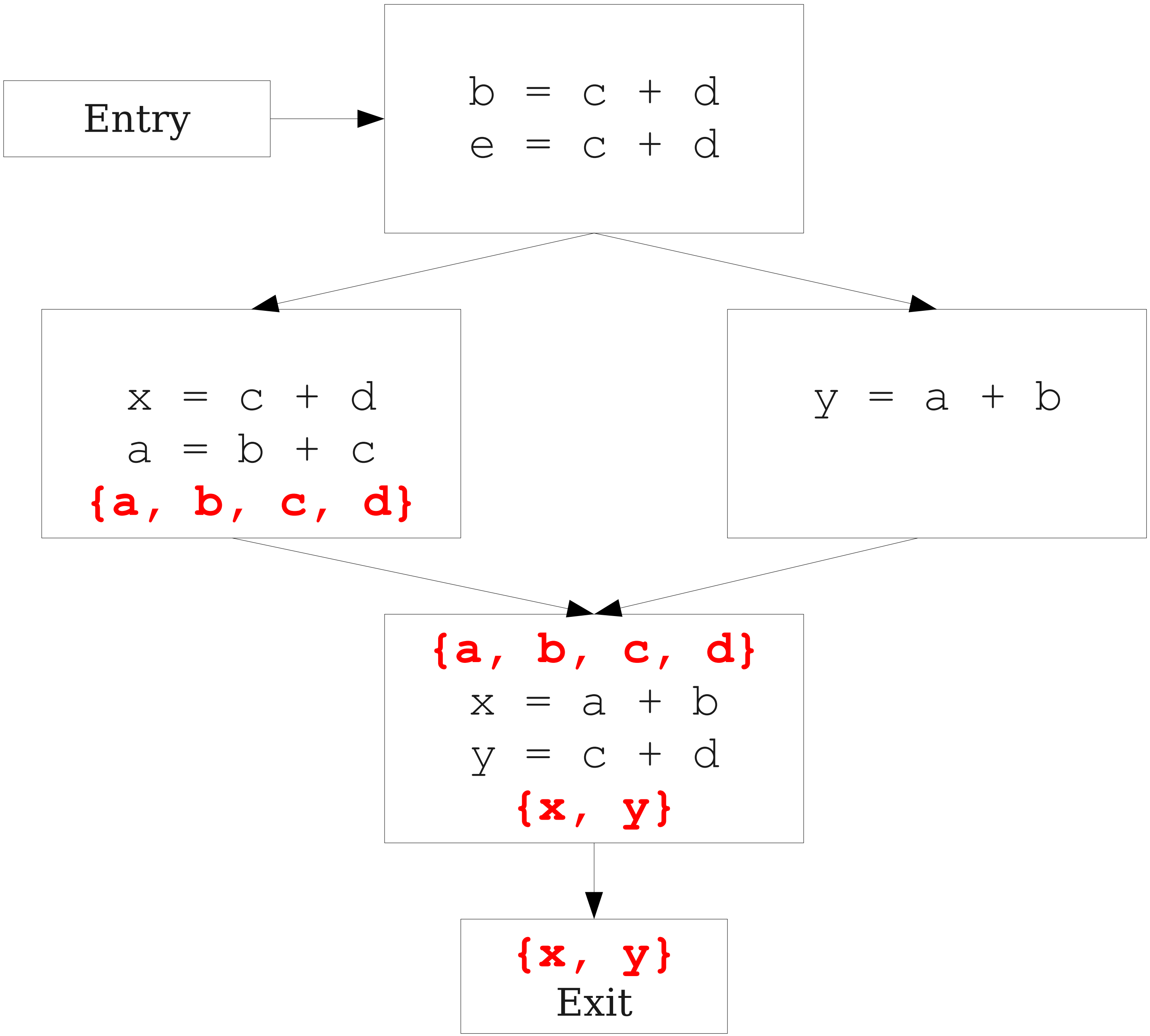
CFGs Without Loops



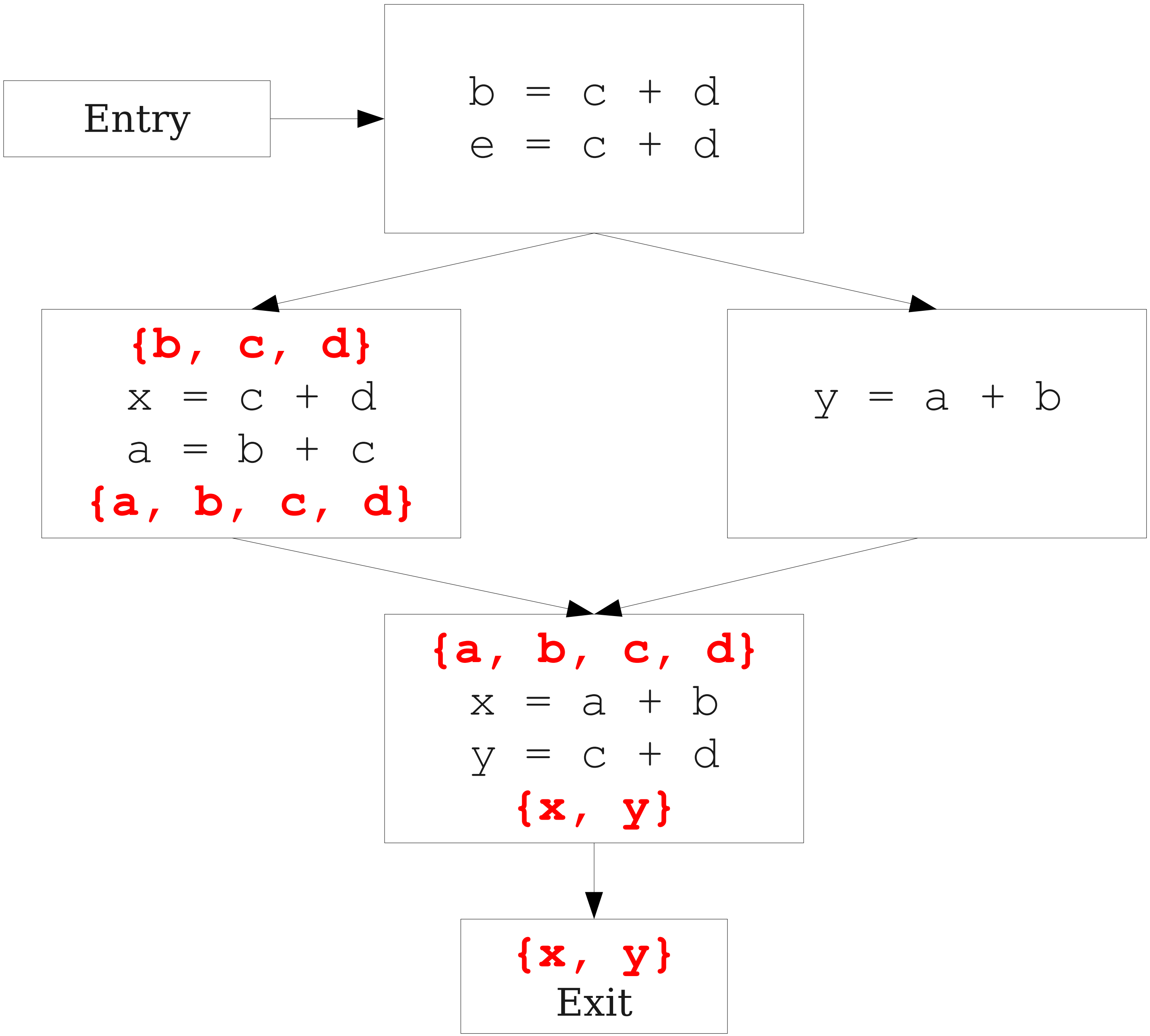
CFGs Without Loops



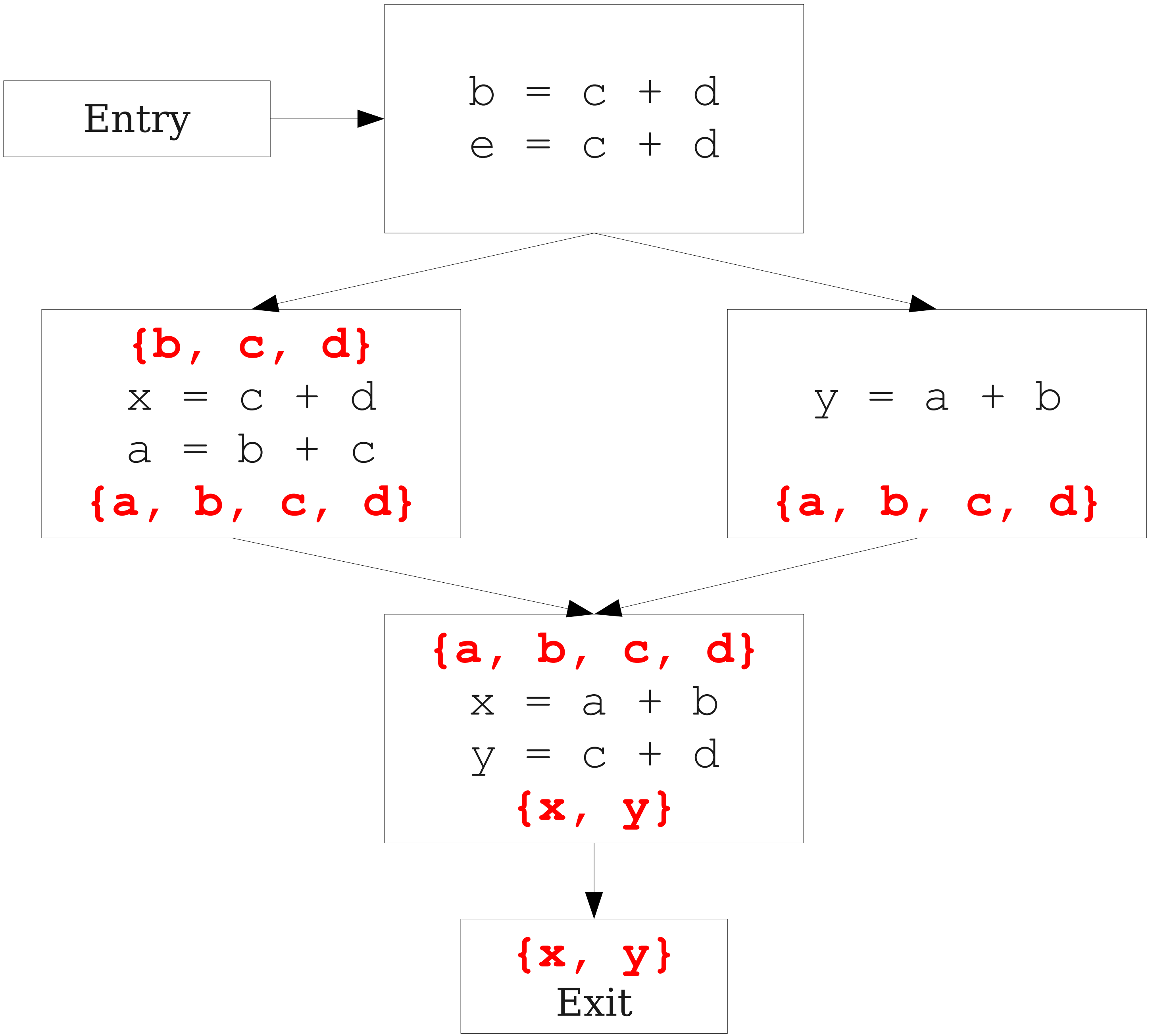
CFGs Without Loops



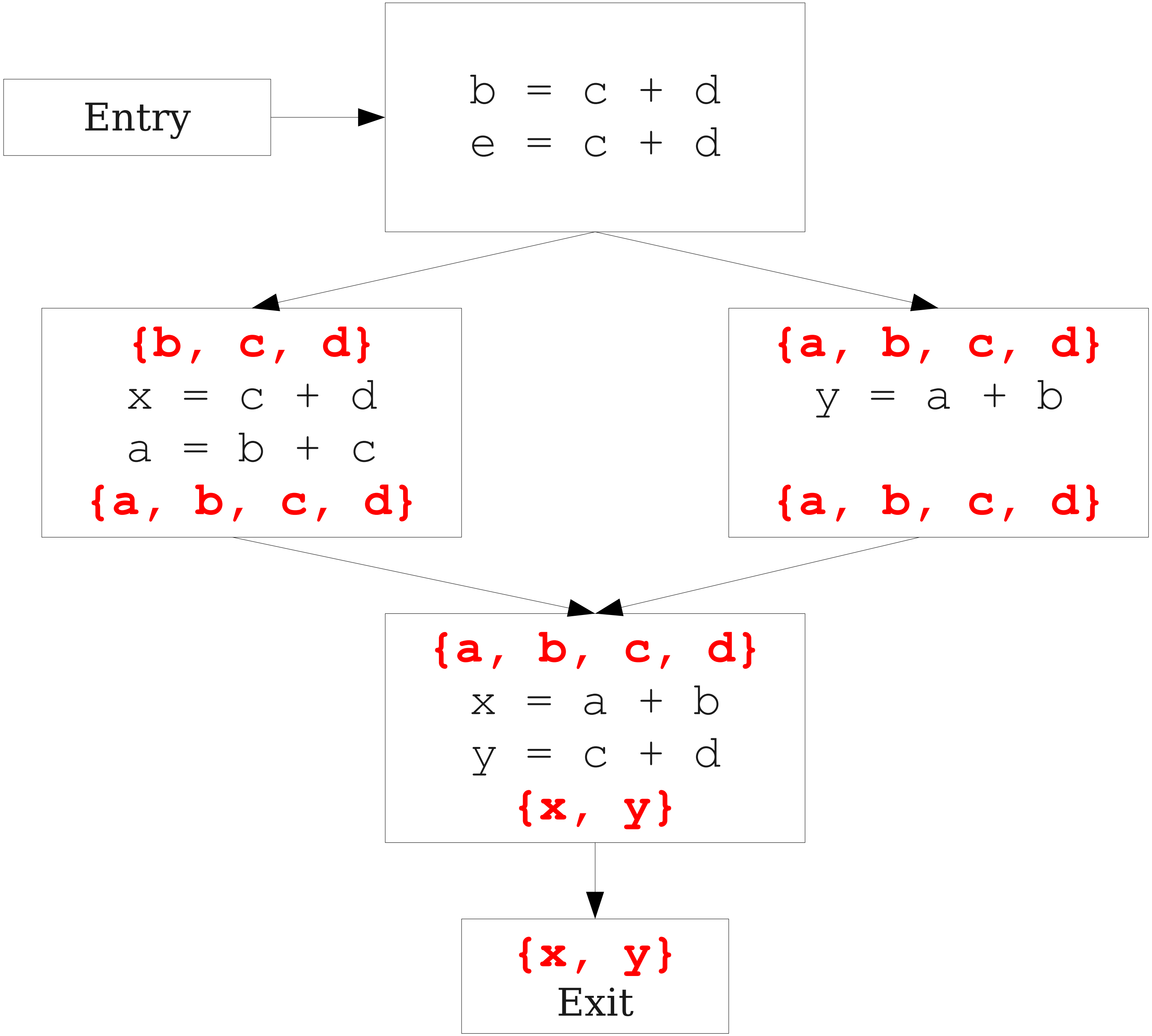
CFGs Without Loops



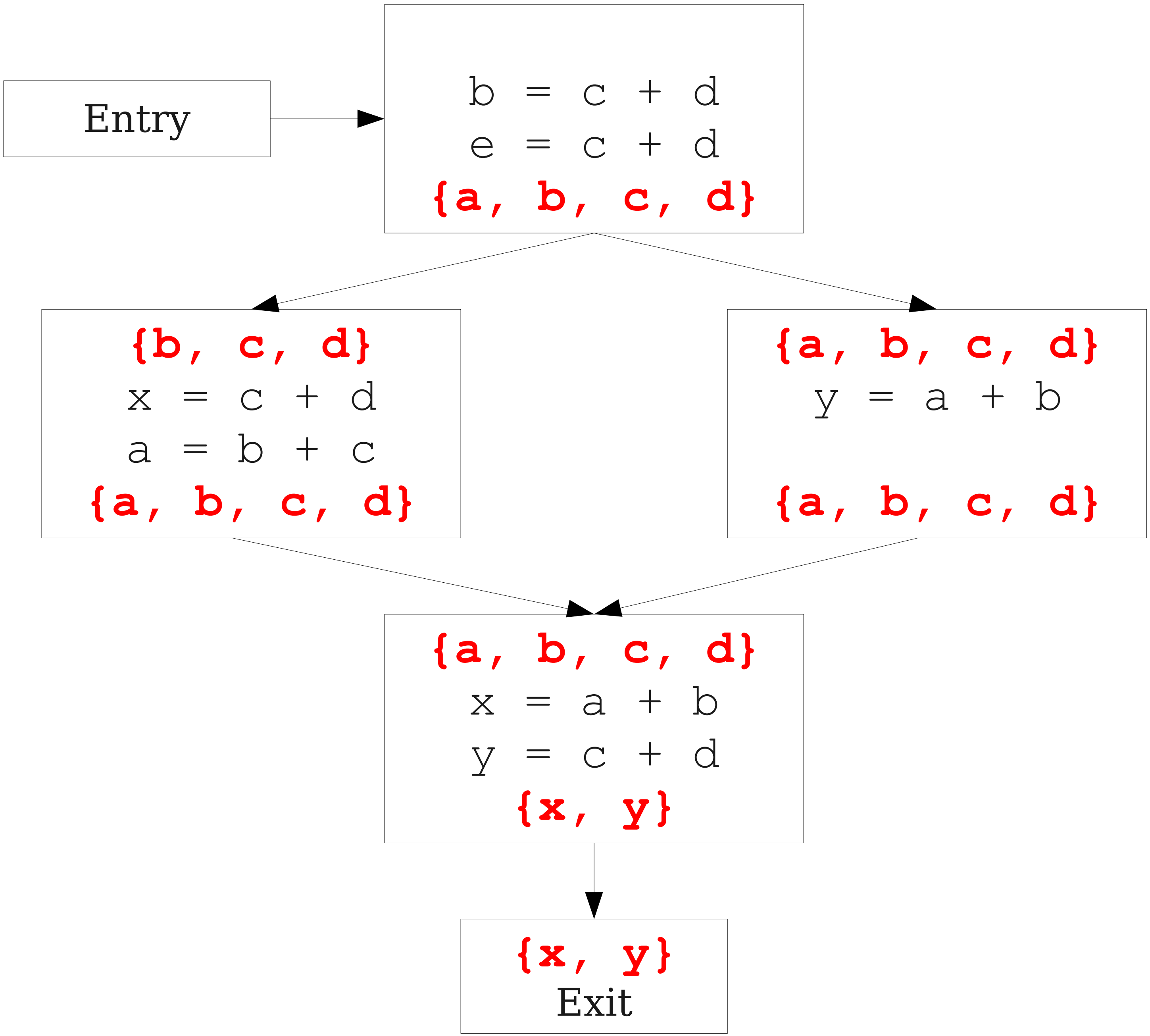
CFGs Without Loops



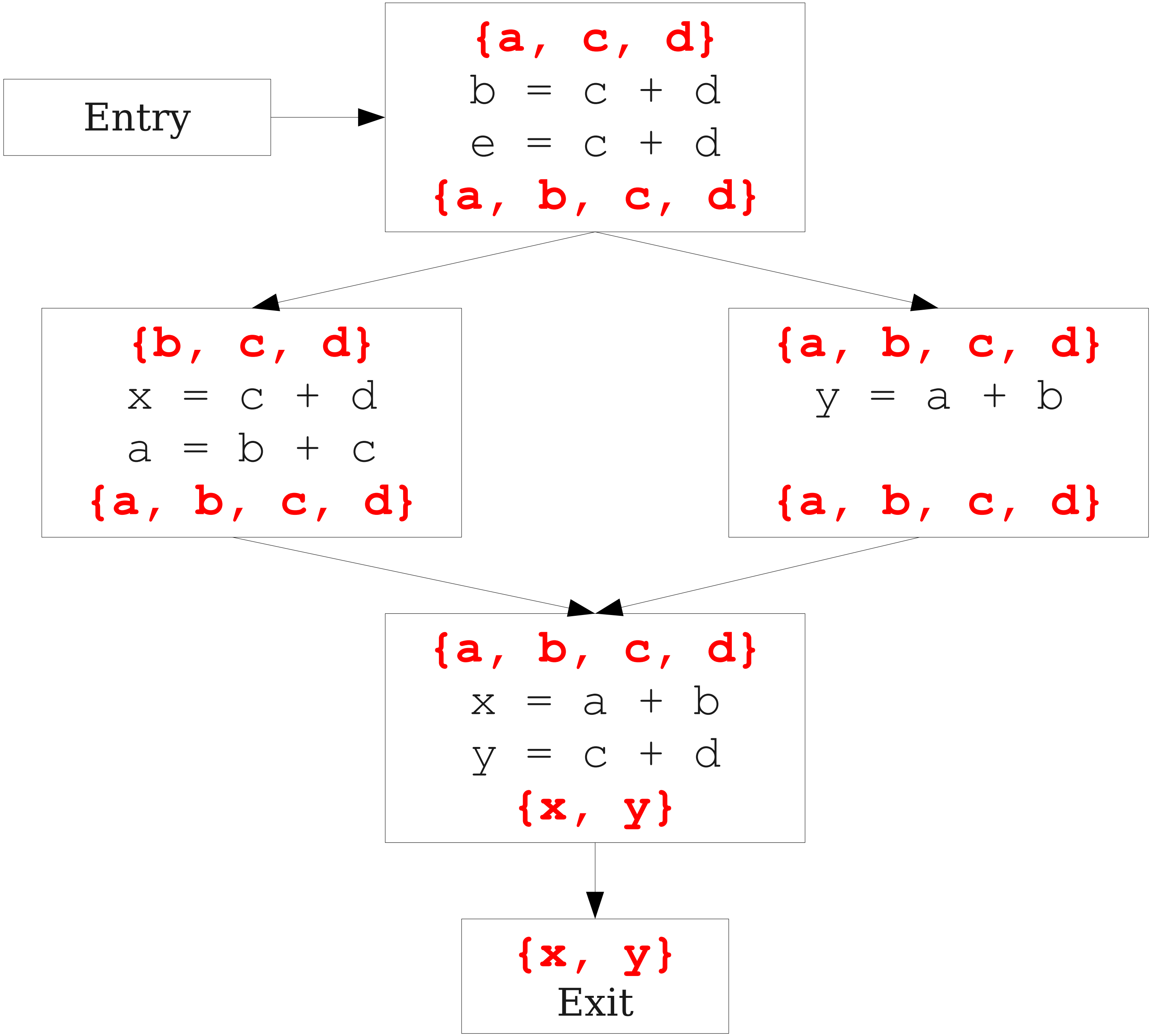
CFGs Without Loops



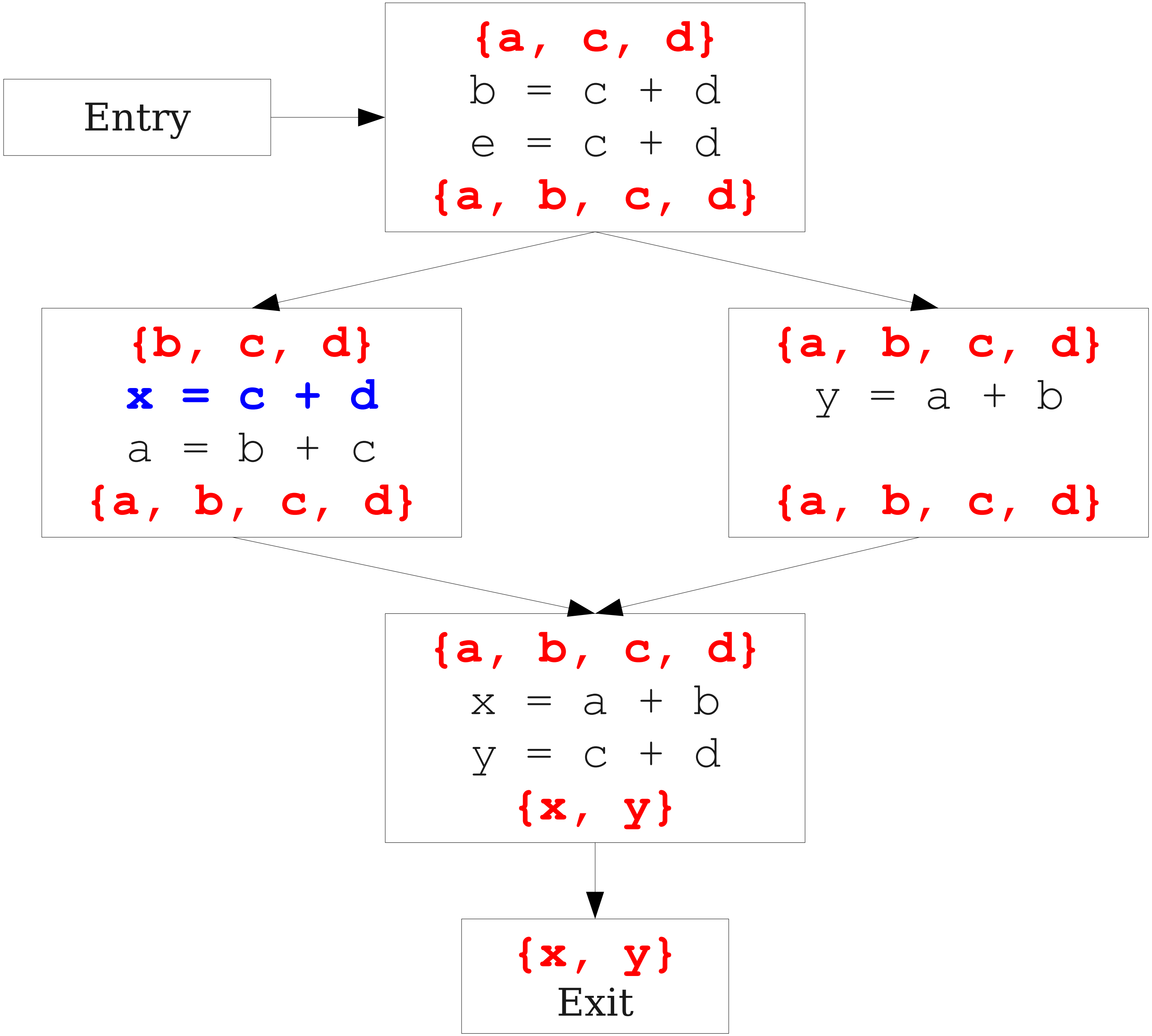
CFGs Without Loops



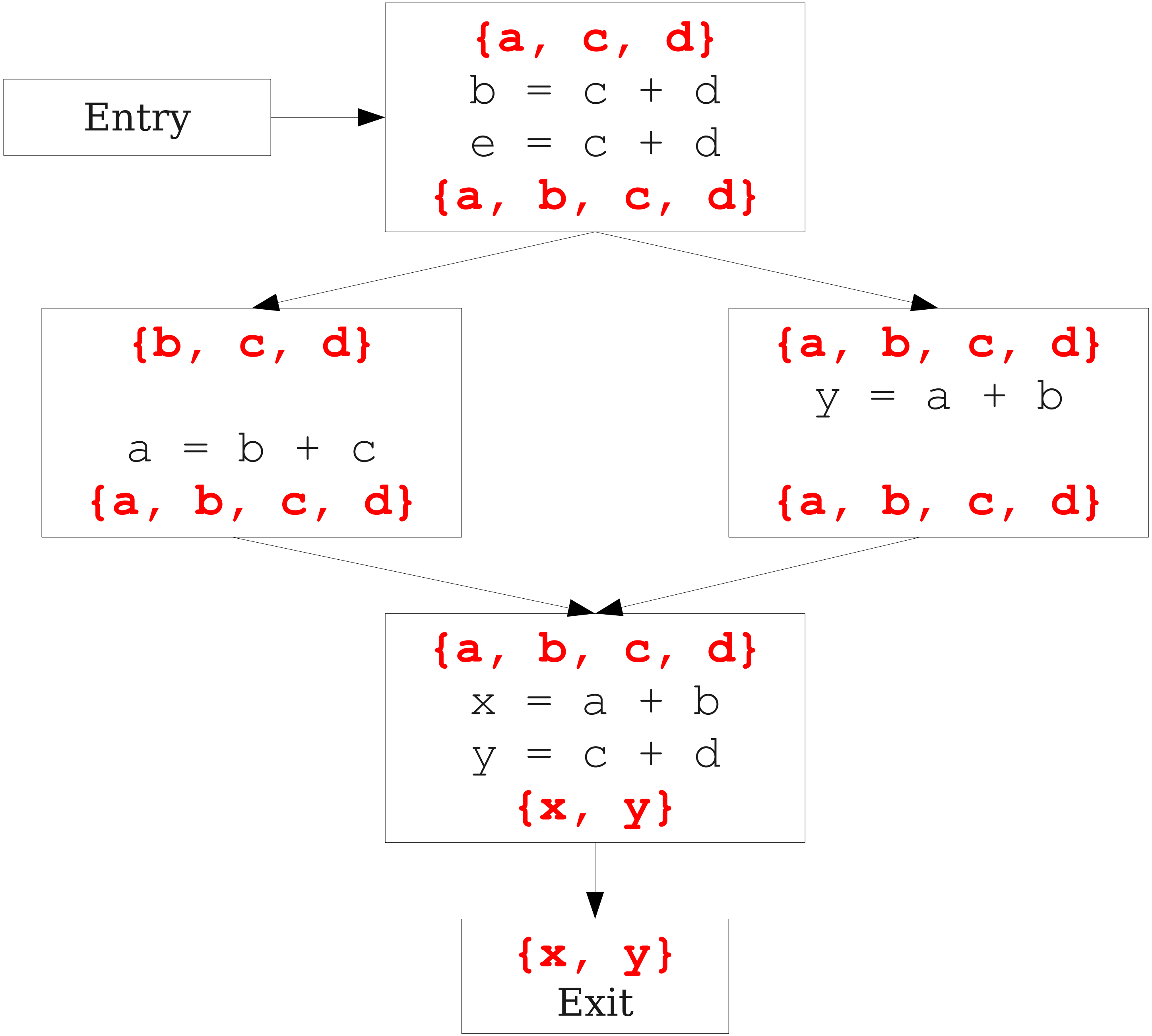
CFGs Without Loops



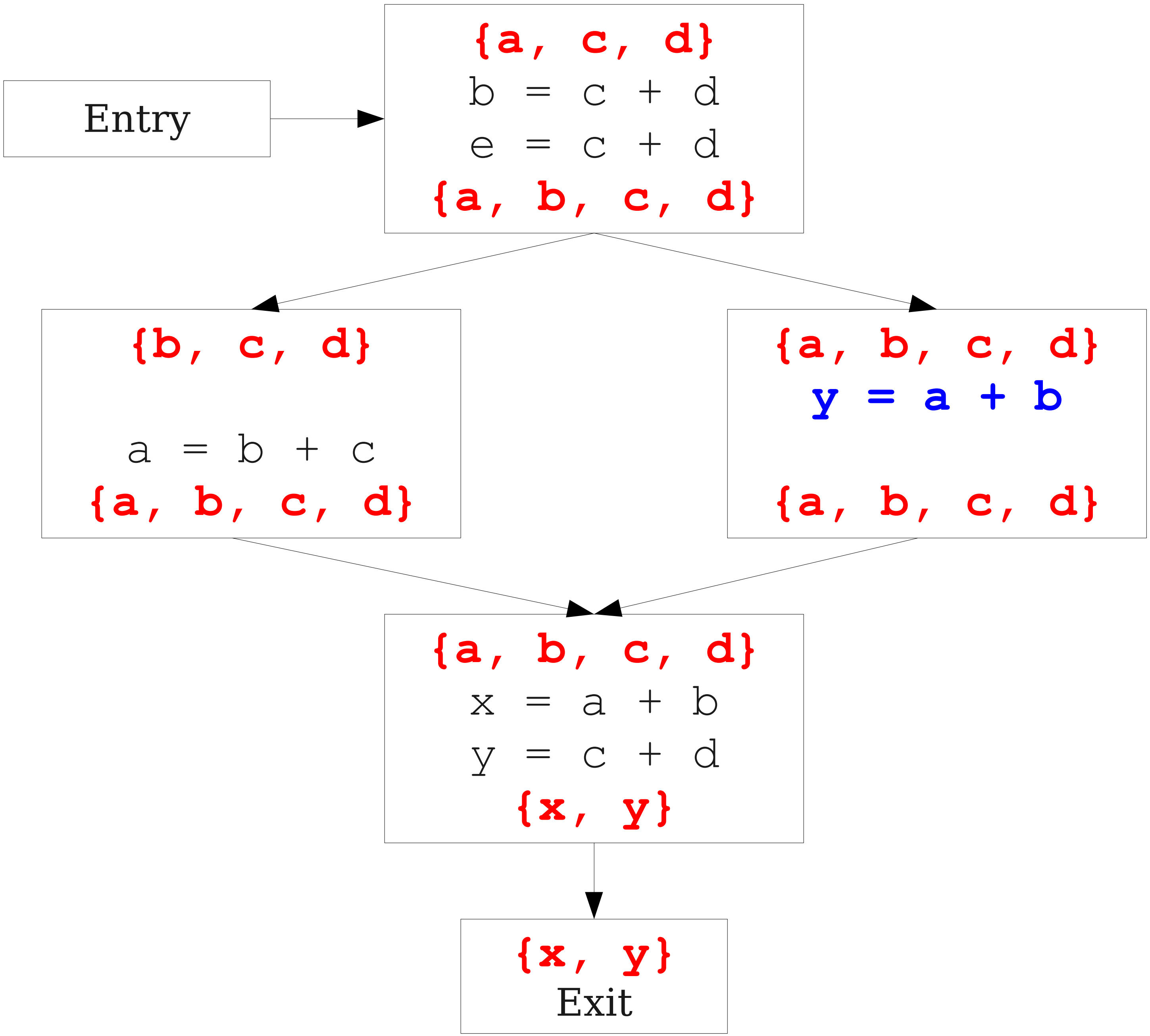
CFGs Without Loops



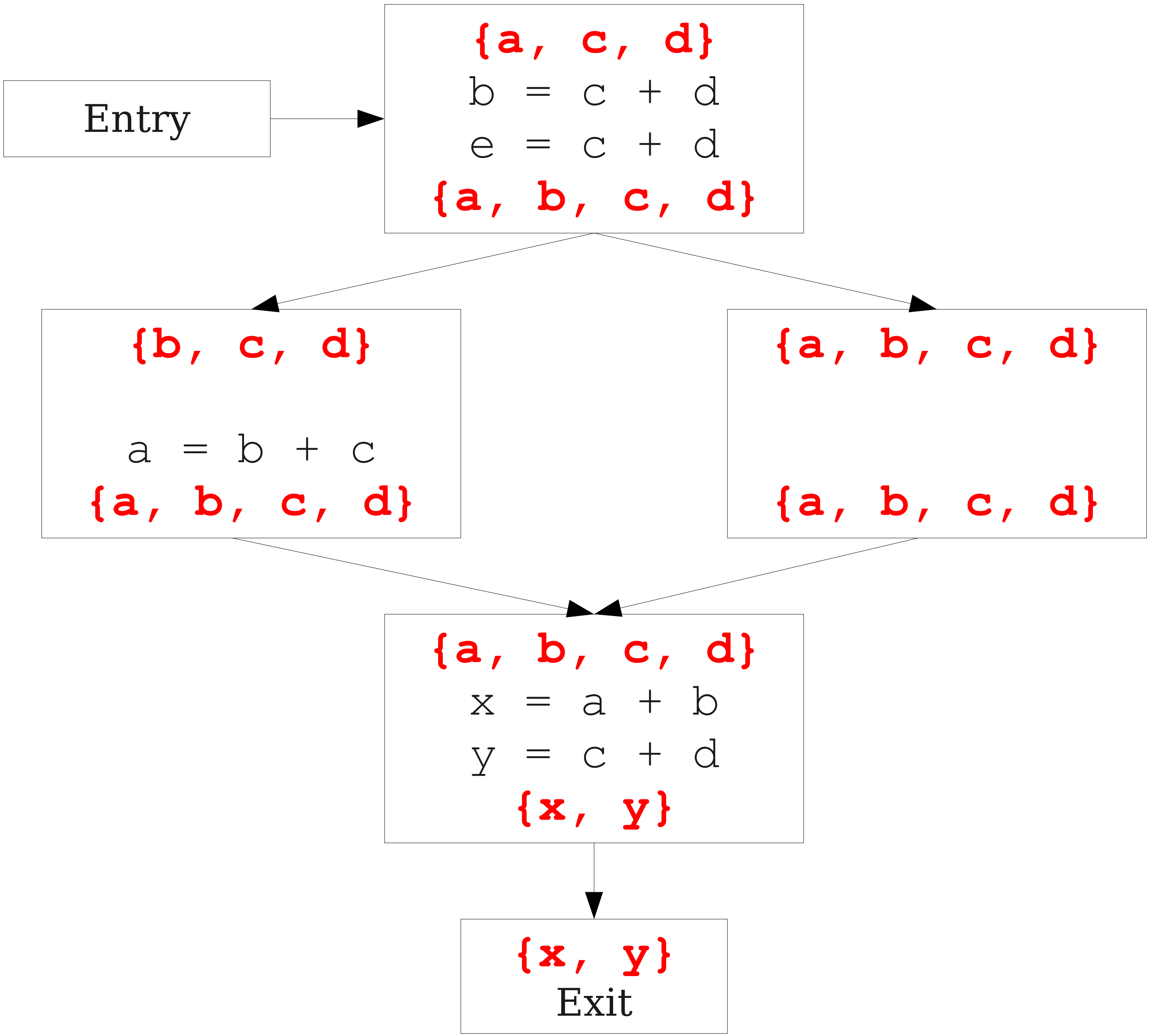
CFGs Without Loops



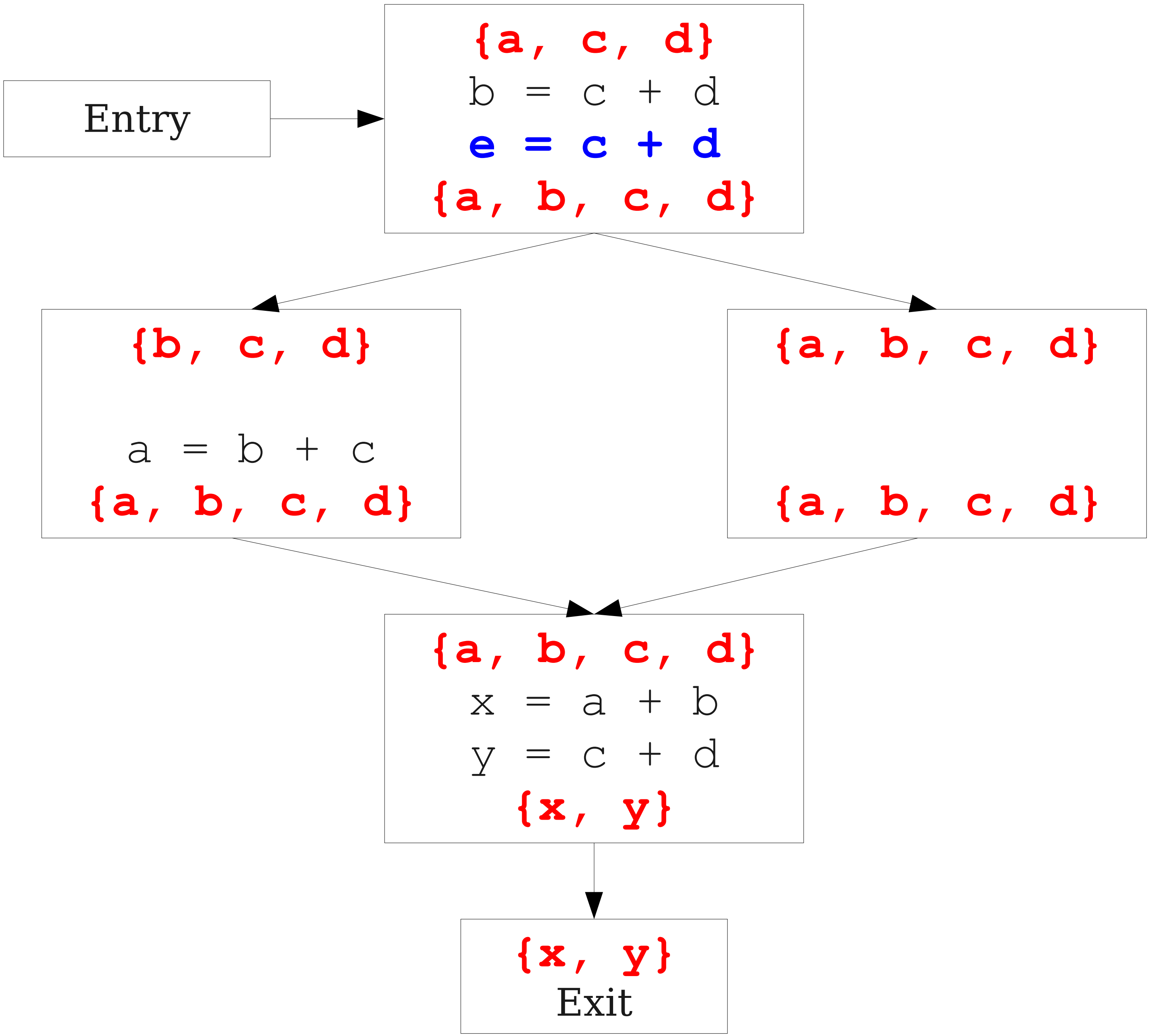
CFGs Without Loops



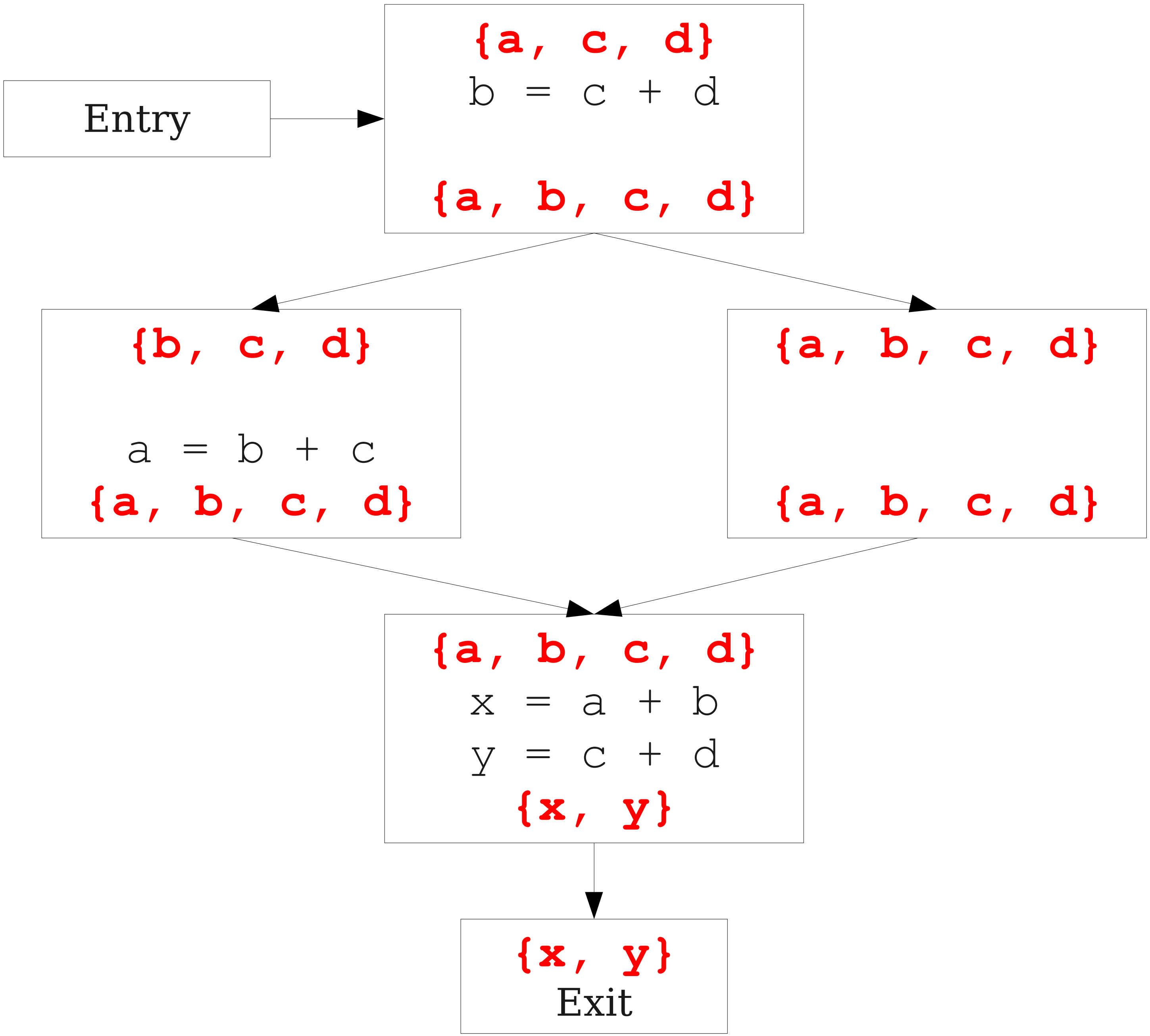
CFGs Without Loops



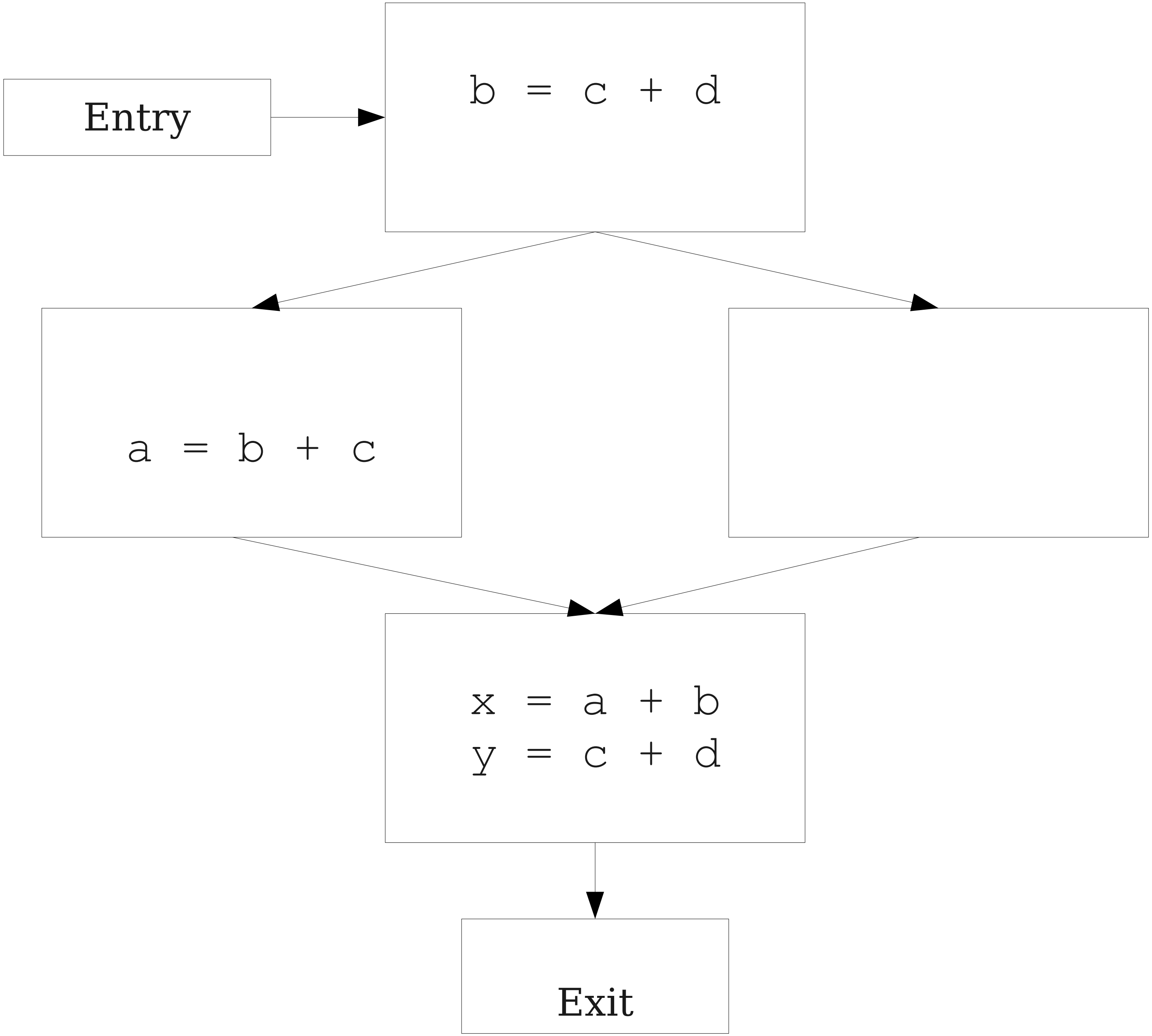
CFGs Without Loops



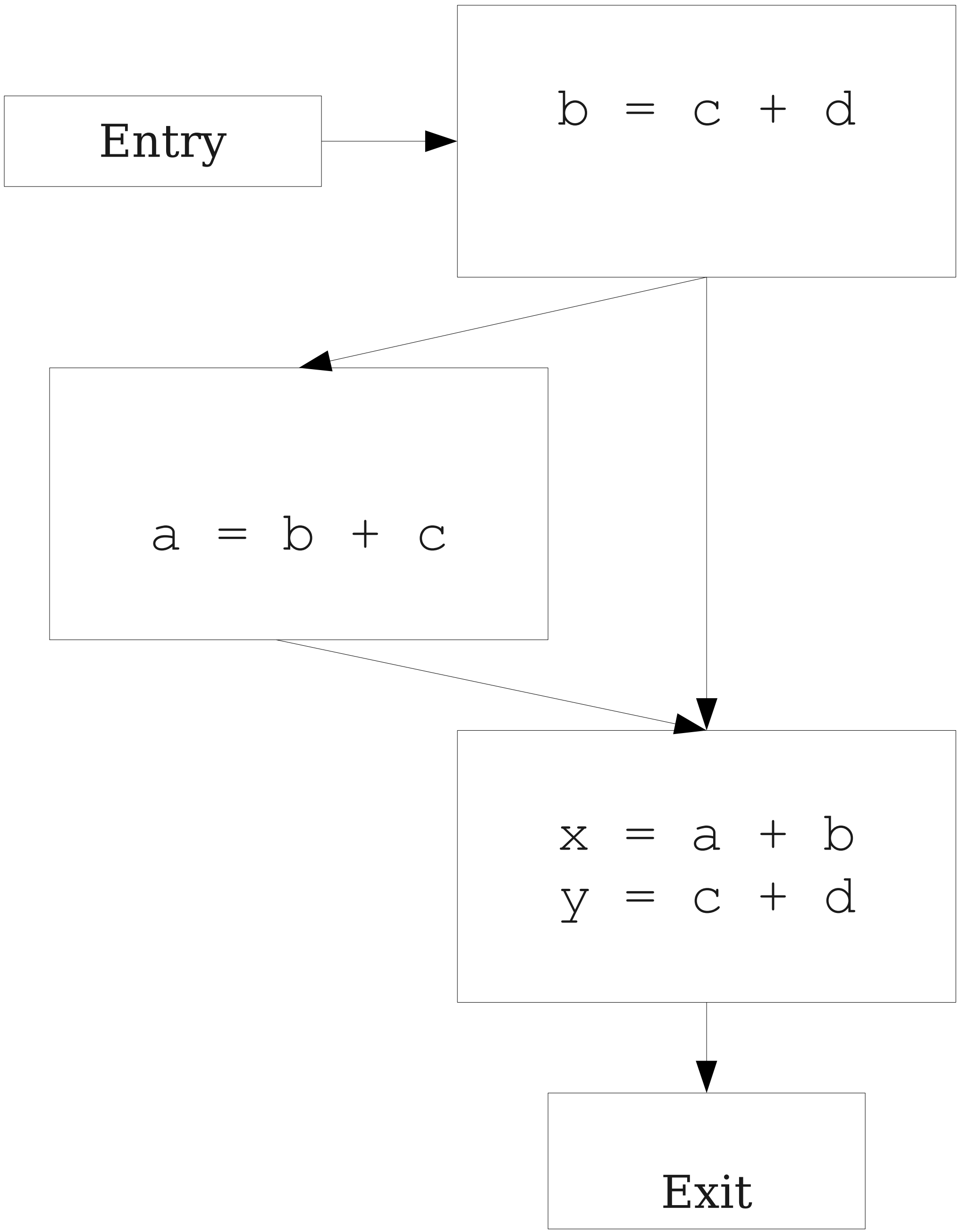
CFGs Without Loops



CFGs Without Loops



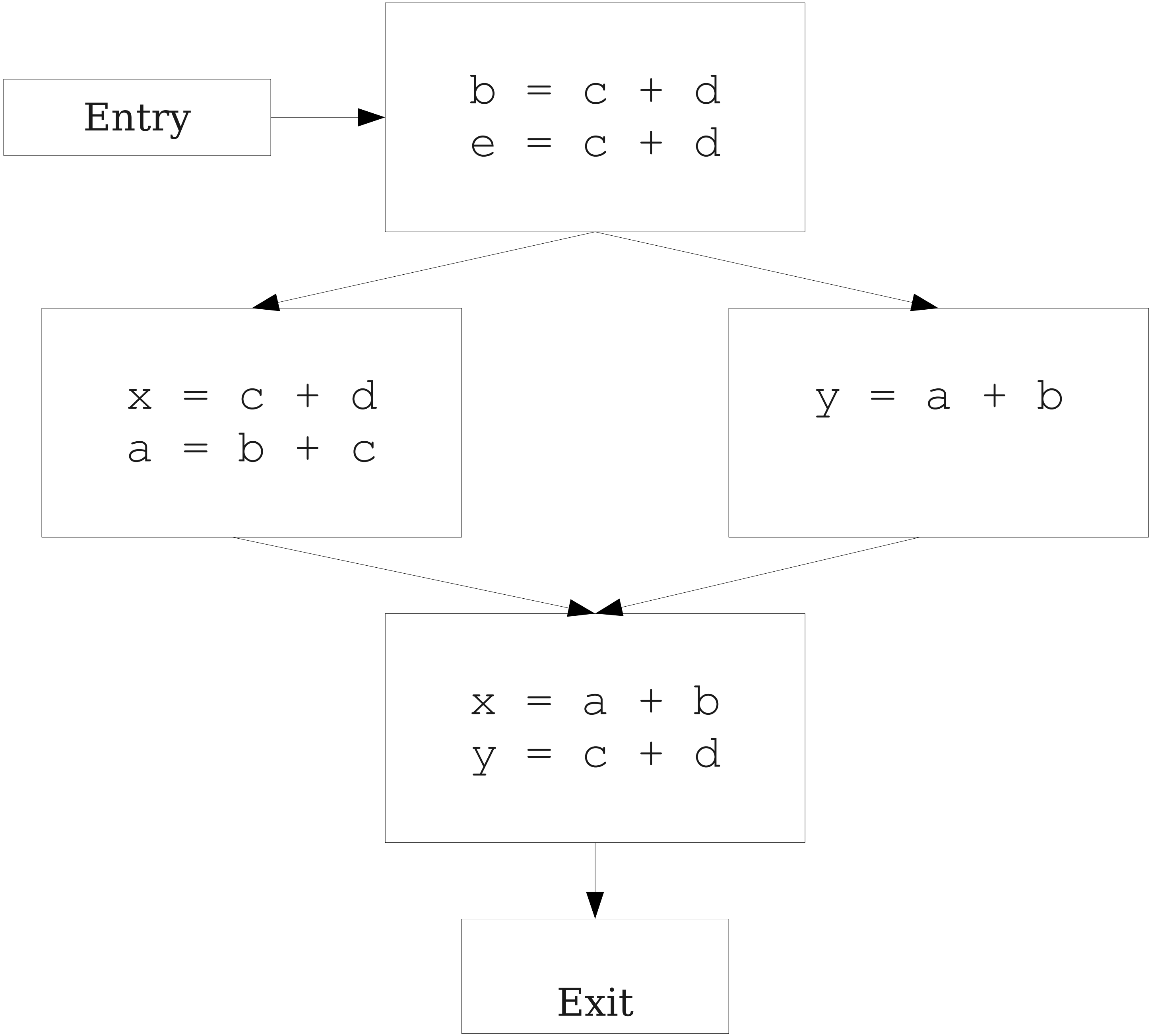
CFGs Without Loops



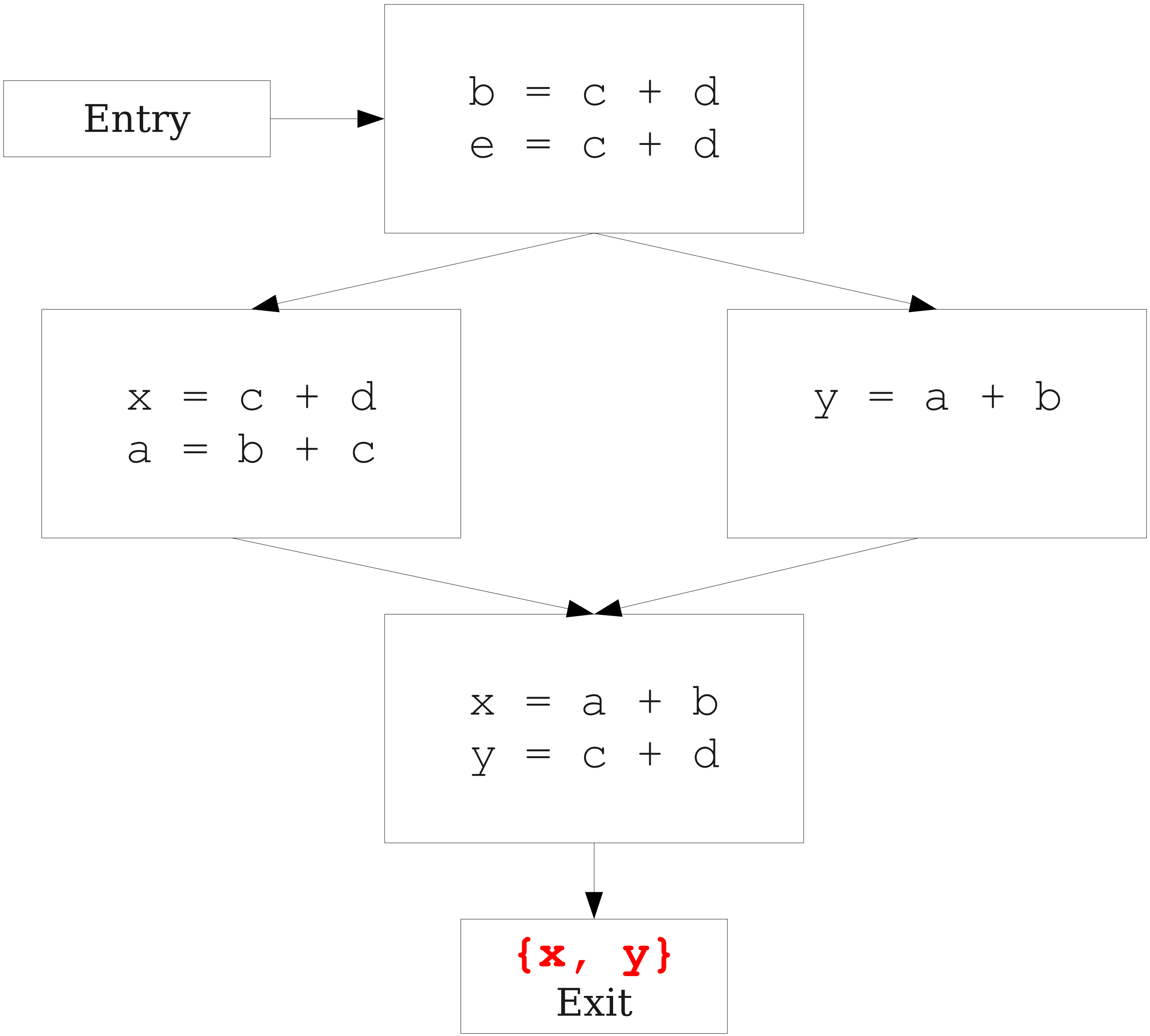
Major Changes, Part One

- 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.

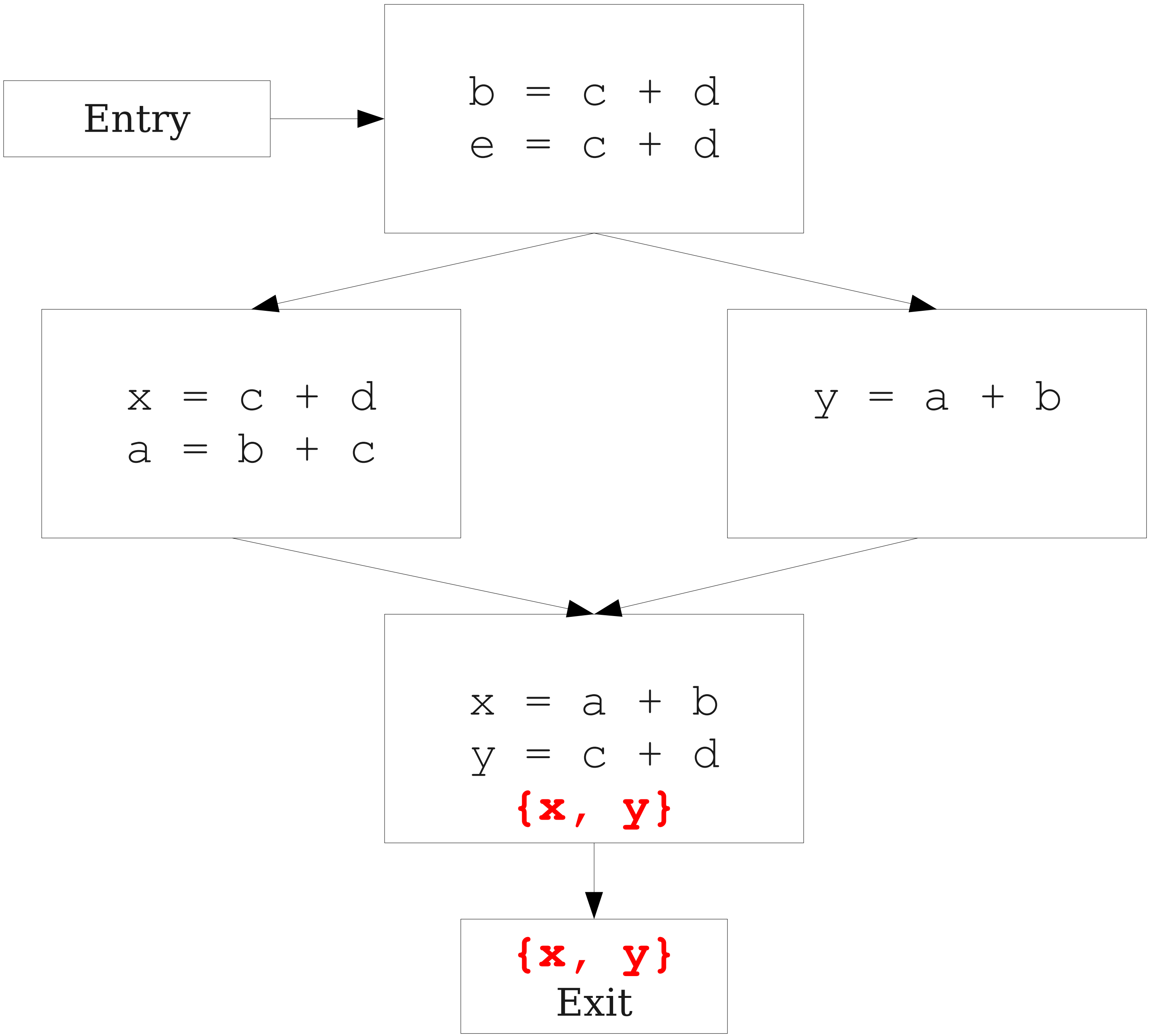
CFGs Without Loops



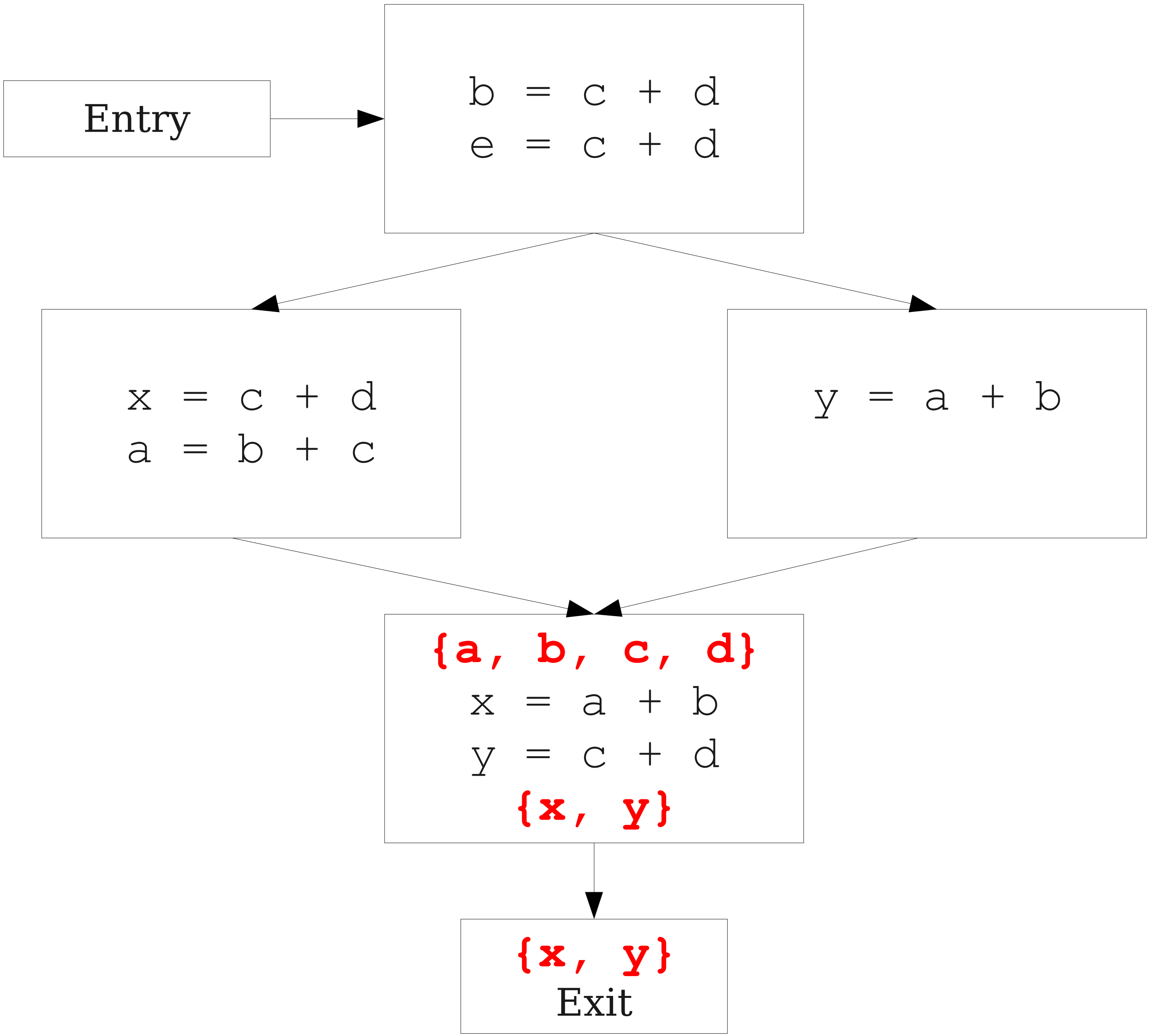
CFGs Without Loops



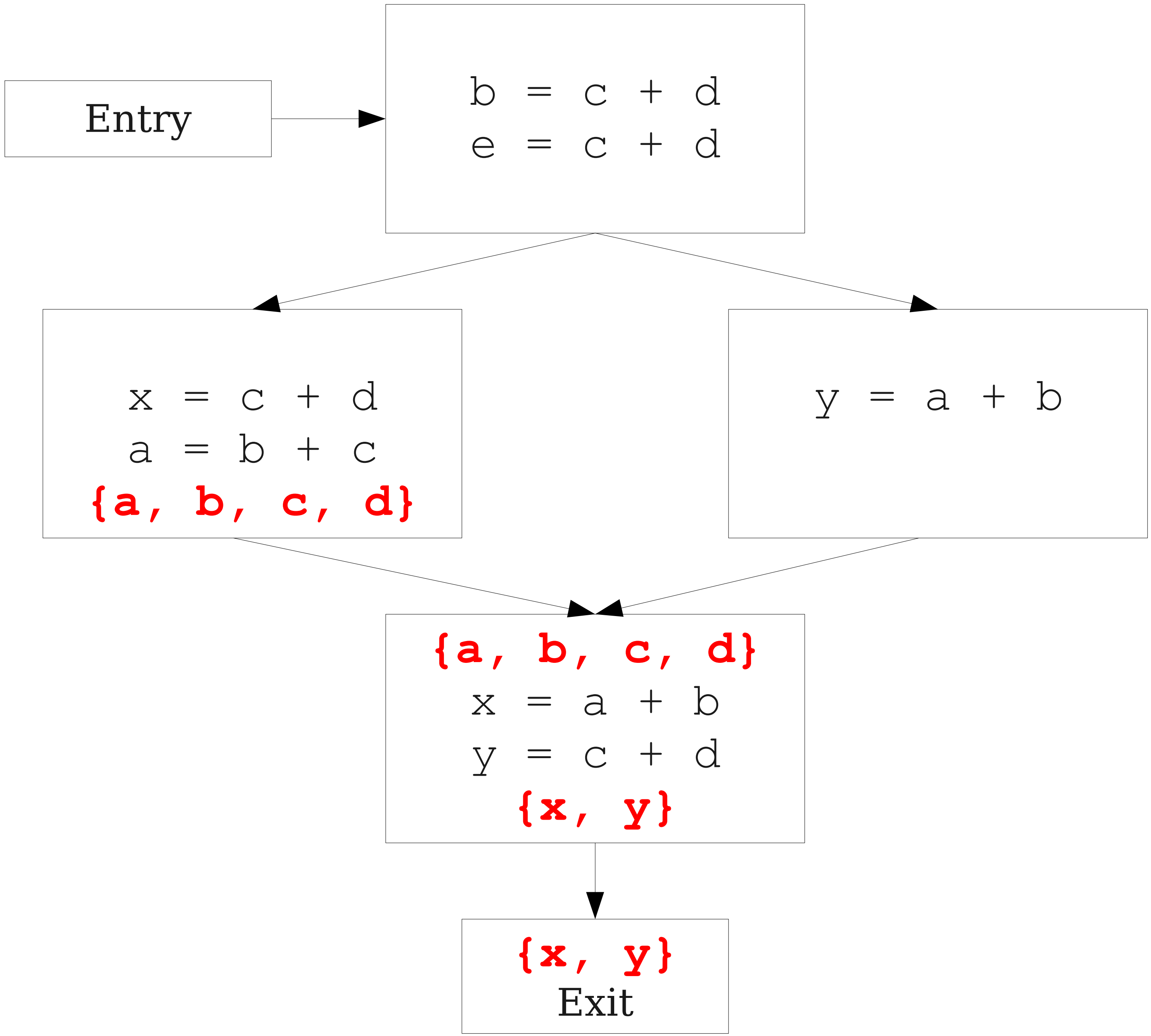
CFGs Without Loops



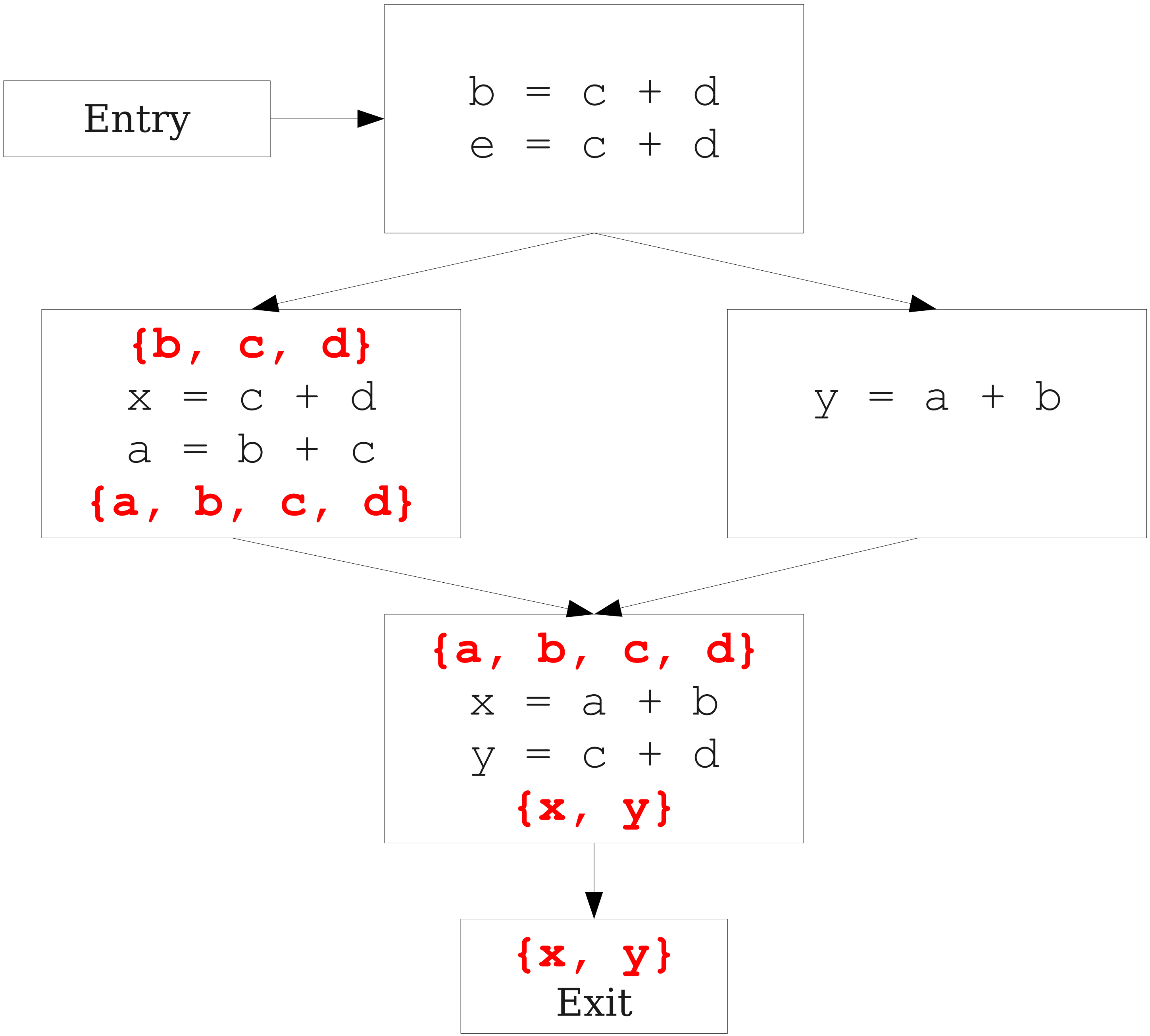
CFGs Without Loops



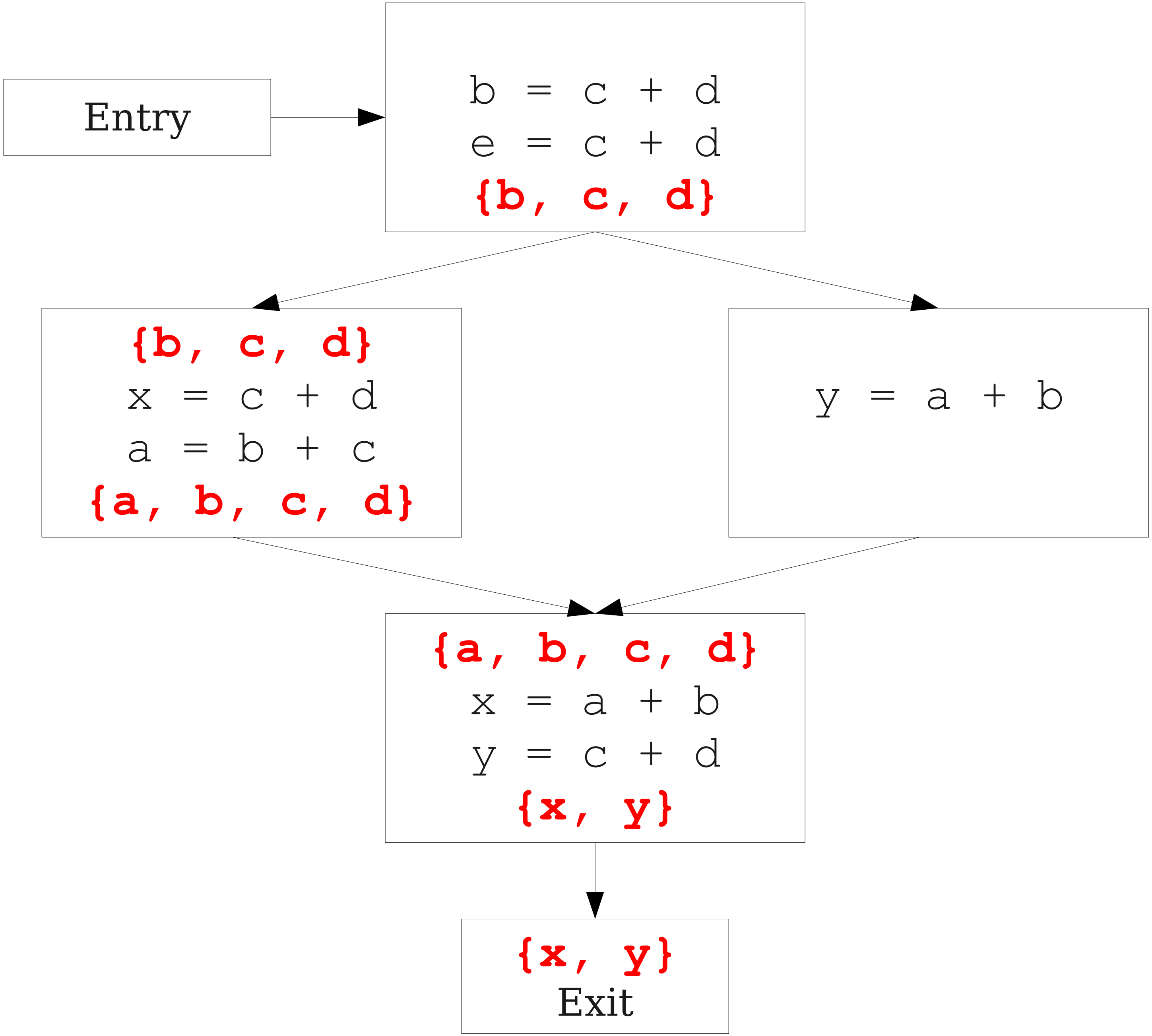
CFGs Without Loops



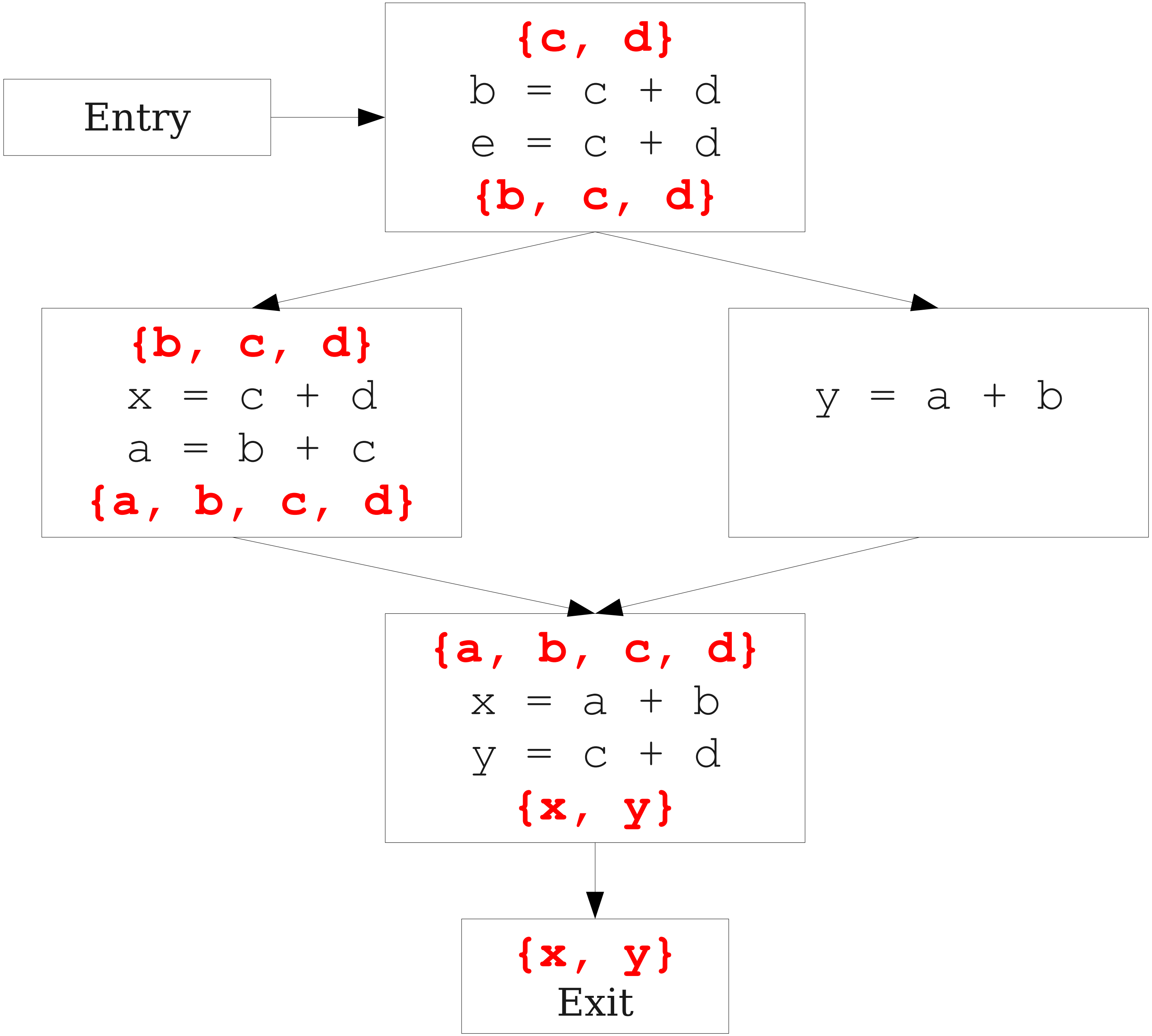
CFGs Without Loops



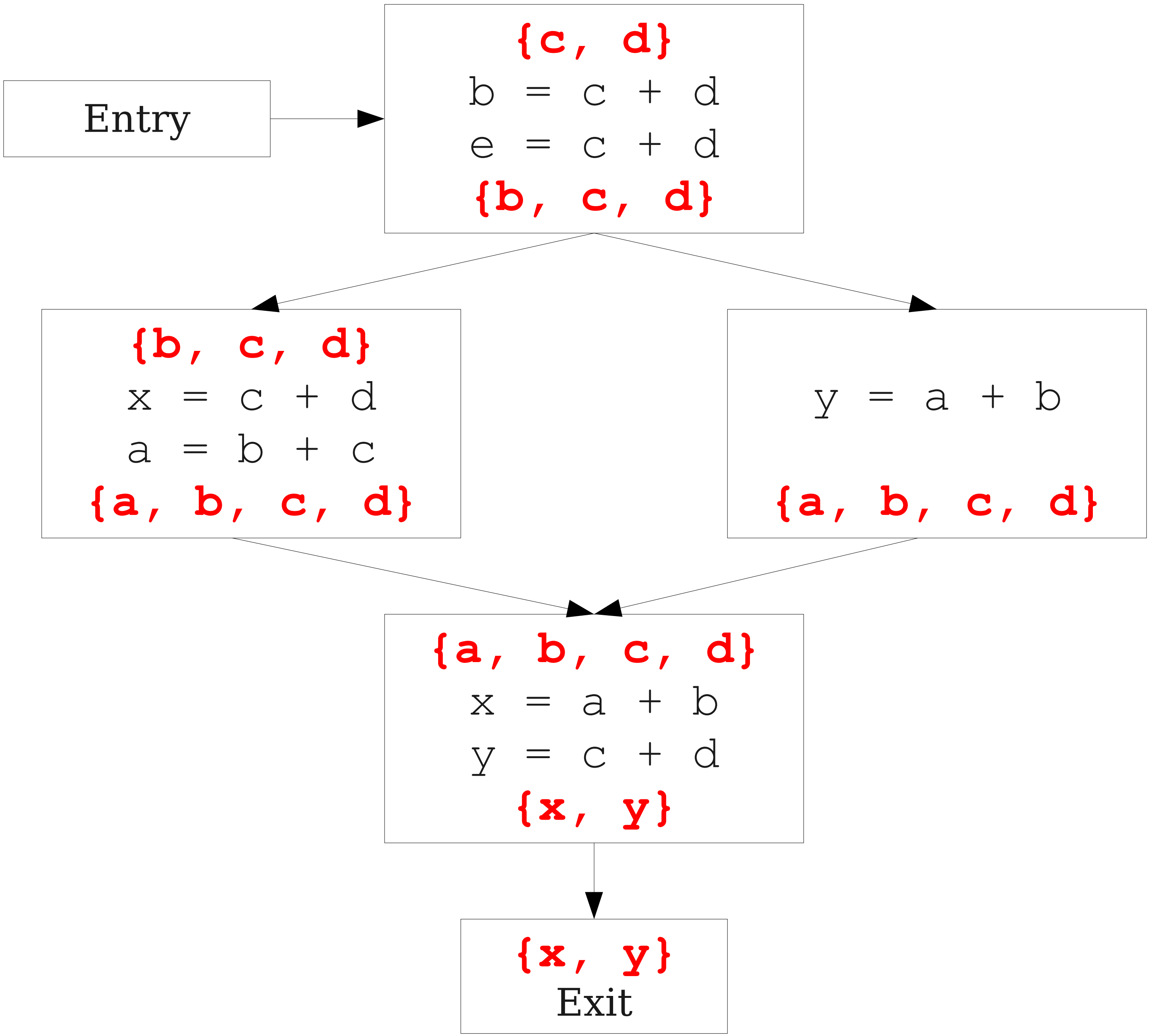
CFGs Without Loops



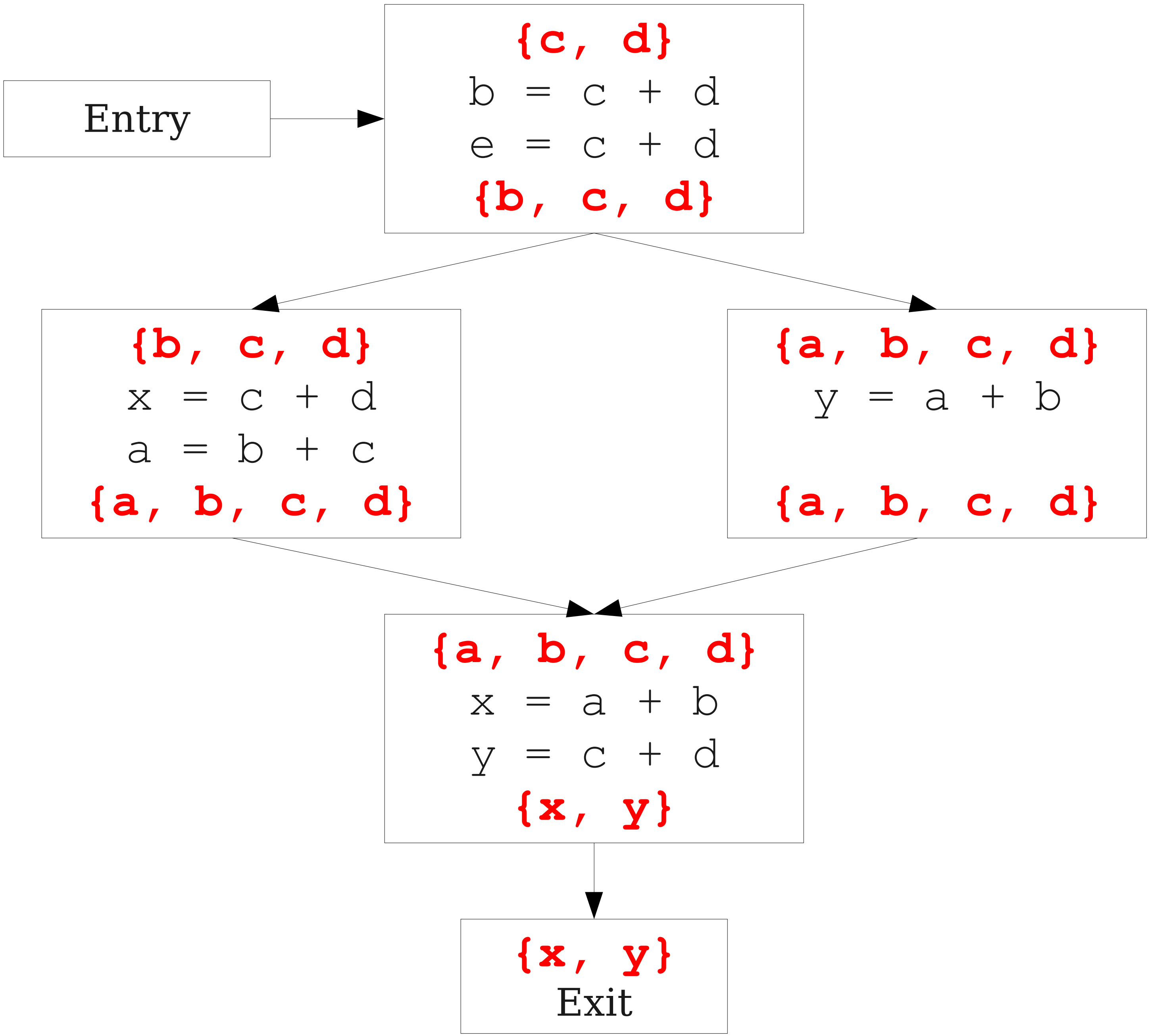
CFGs Without Loops



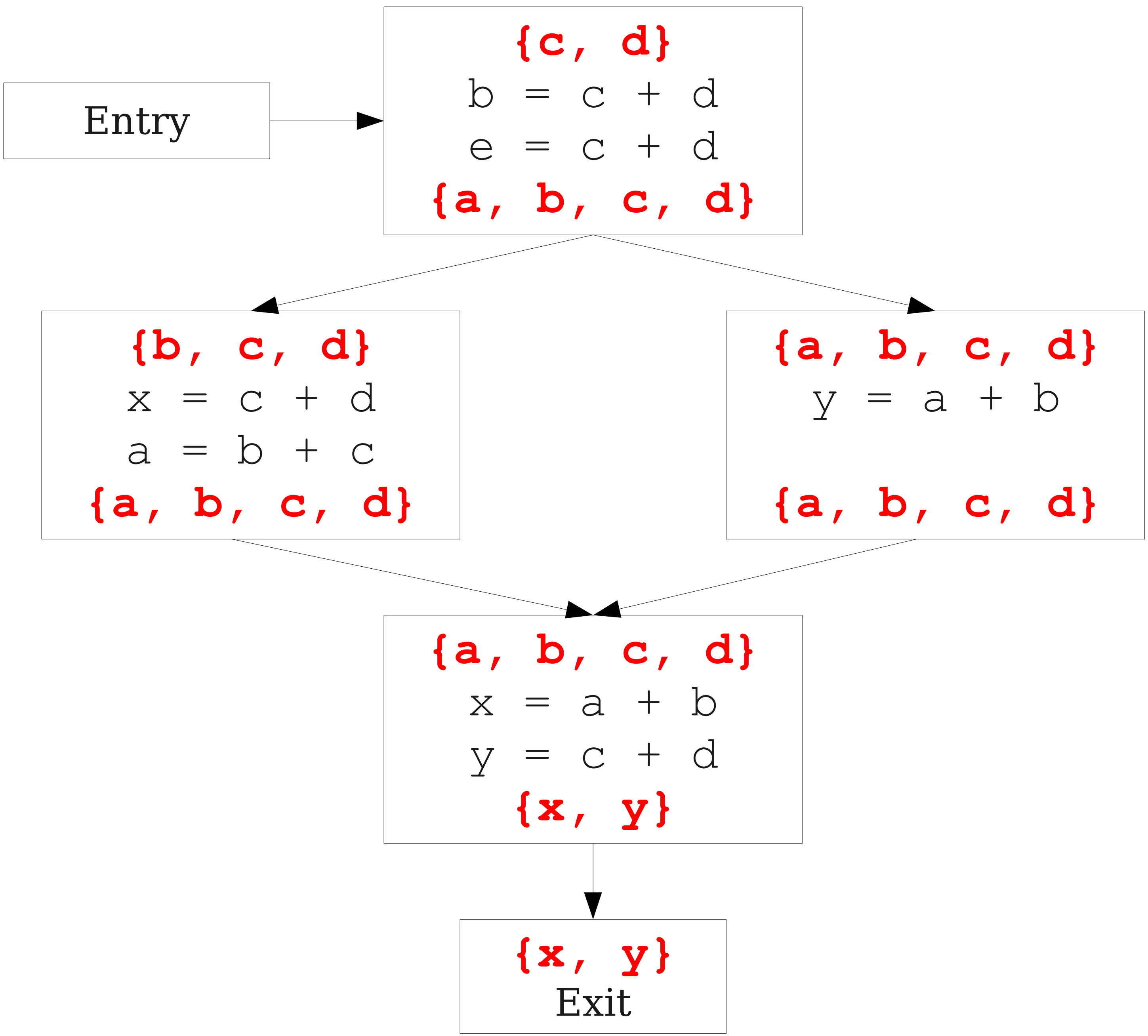
CFGs Without Loops



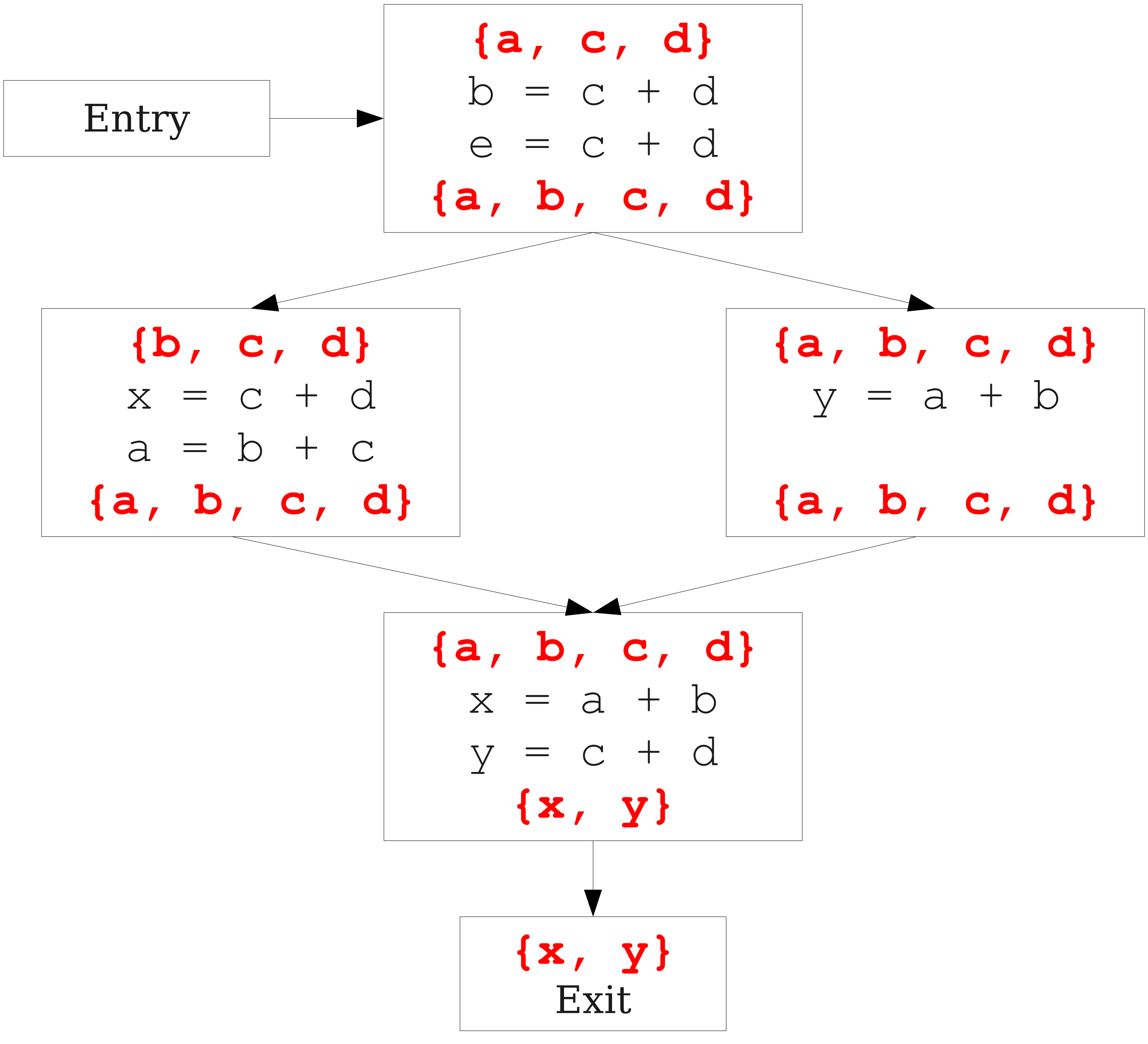
CFGs Without Loops



CFGs Without Loops



CFGs Without Loops



Major Changes, Part II

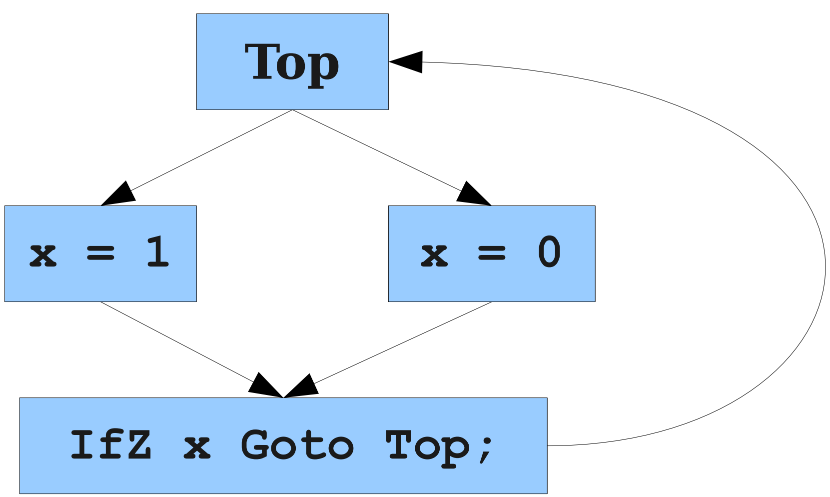
- In a local analysis, there is only one possible path through a basic block.
- In a global analysis, there may be **many** paths through a CFG.
- May need to recompute values multiple times as more information becomes available.
- Need to be careful when doing this not to loop infinitely!
  - (More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

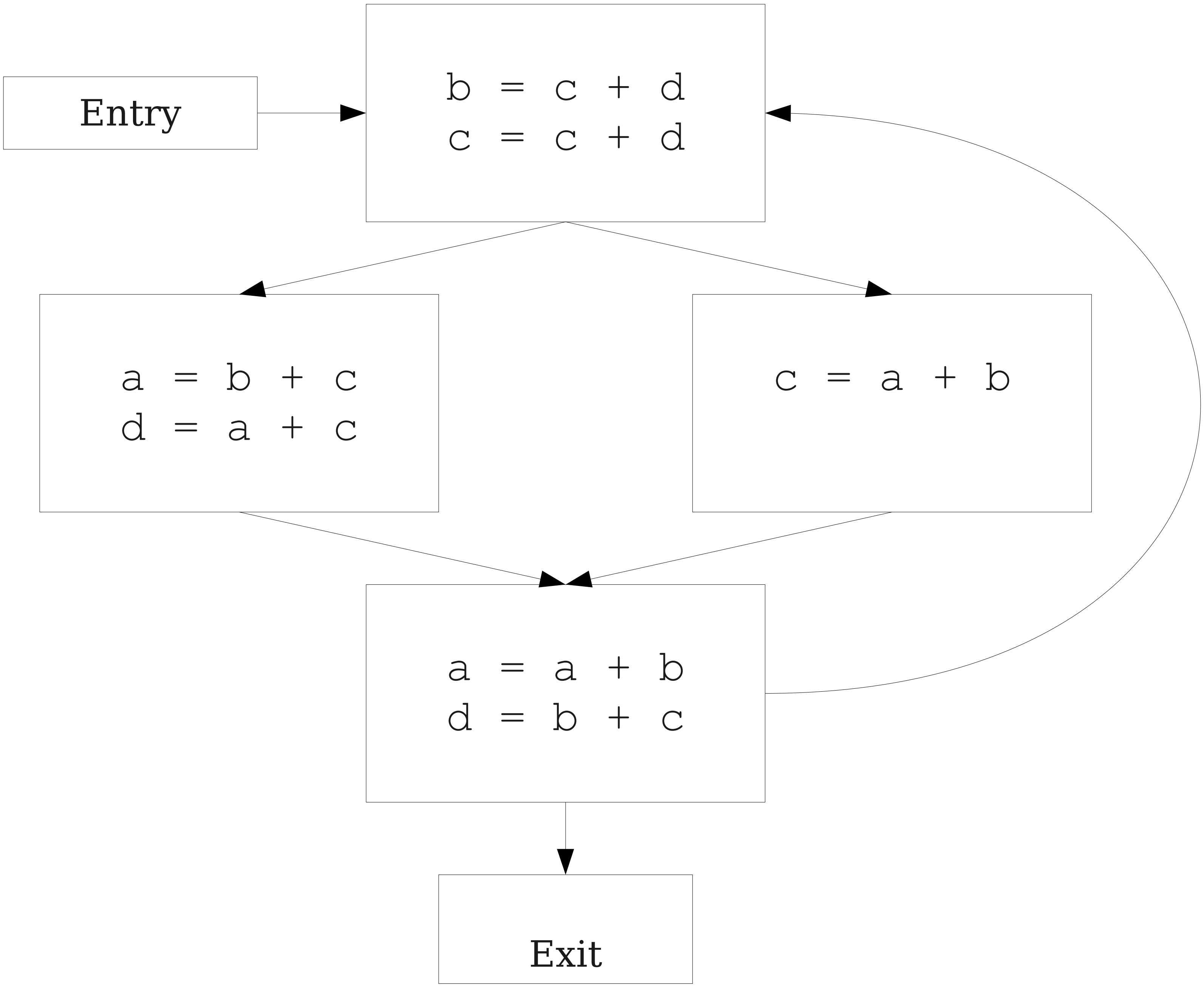


CFGs with Loops

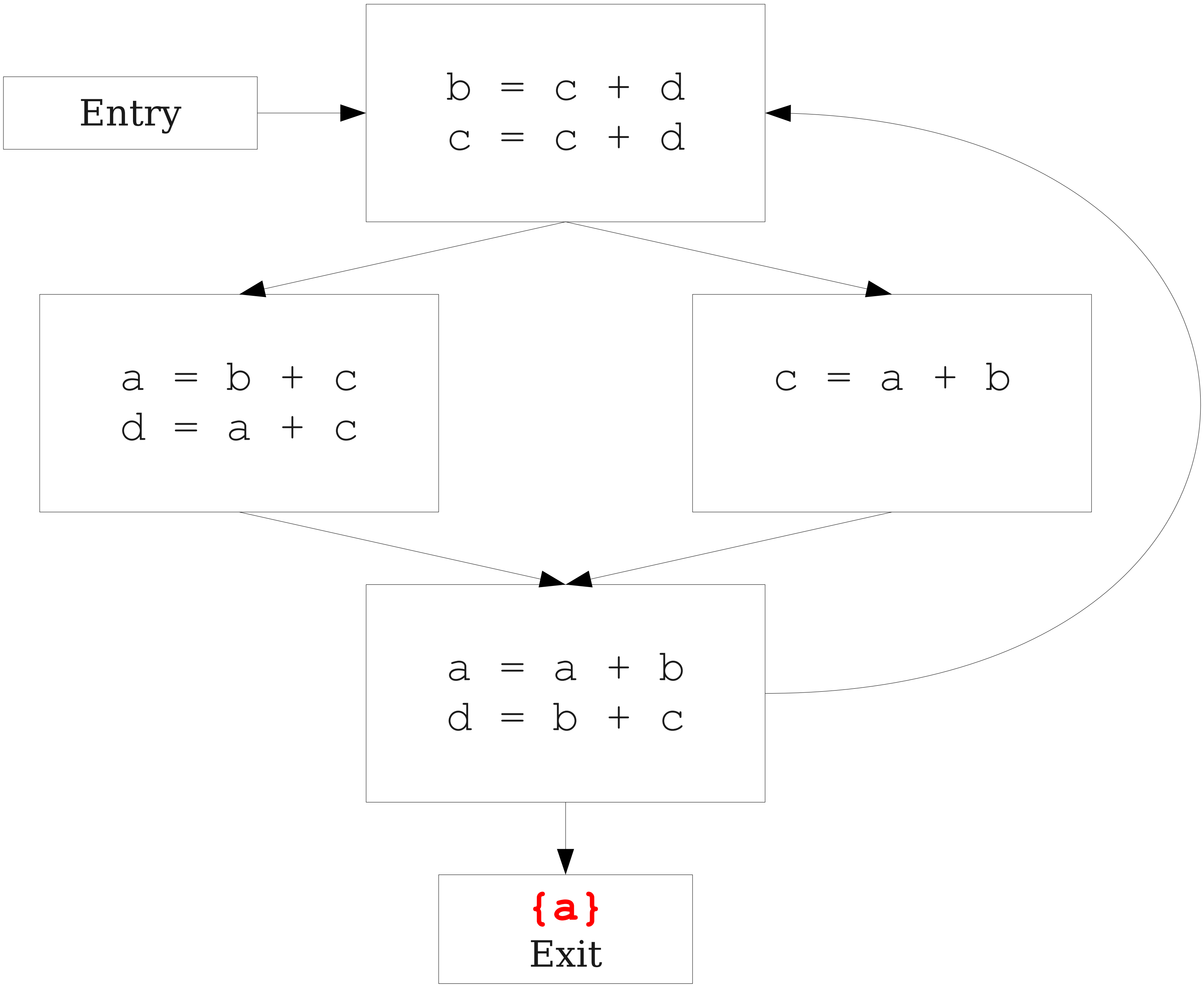
- Up to this point, we've considered loop-free CFGs, which have only finitely many possible paths.
- When we add loops into the picture, this is no longer true.
- Not all possible loops in a CFG can be realized in the actual program.
- **Sound approximation**: Assume that every possible path through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

CFGs With Loops



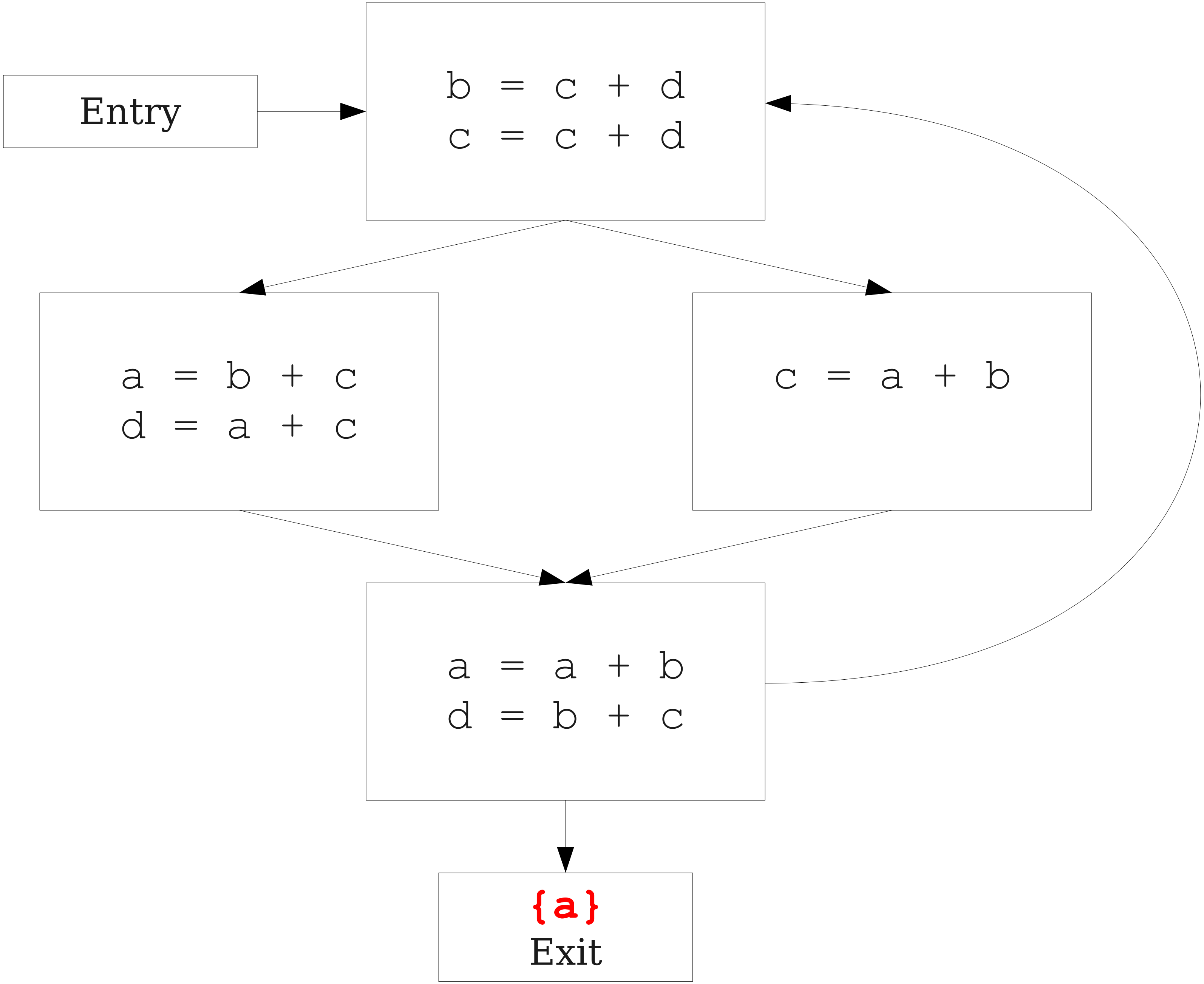
CFGs With Loops



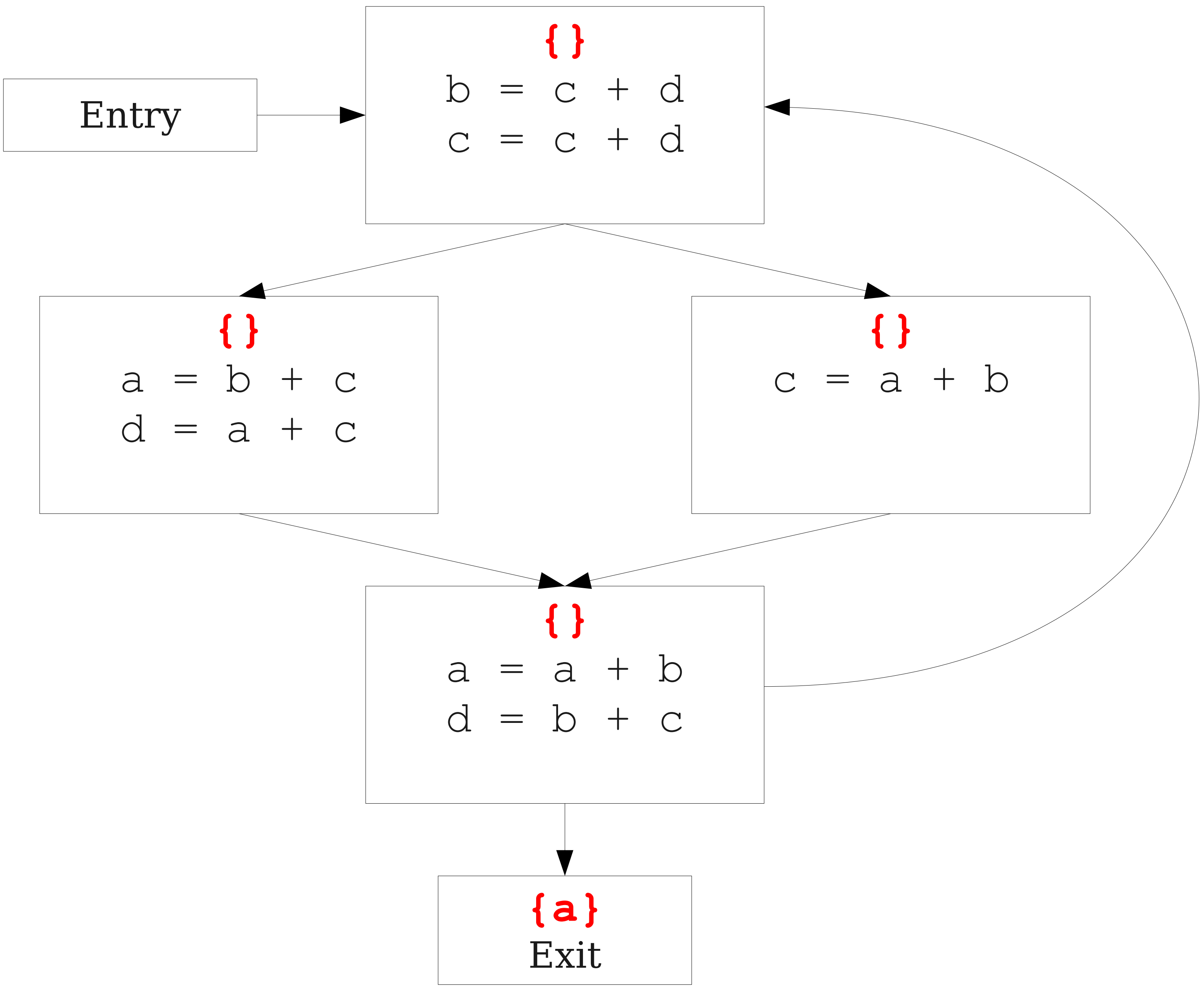
Major Changes, Part III

- 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.

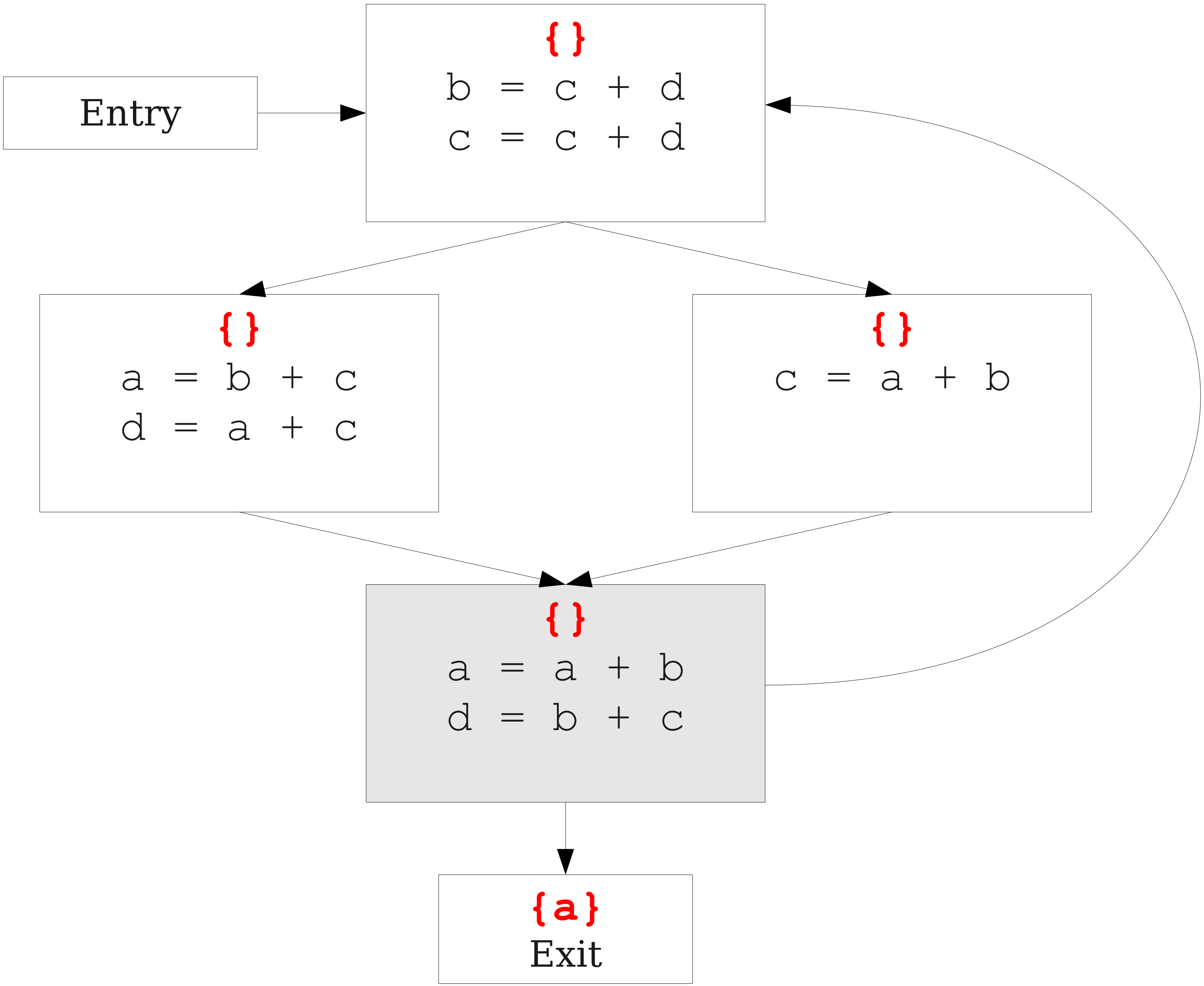
CFGs With Loops



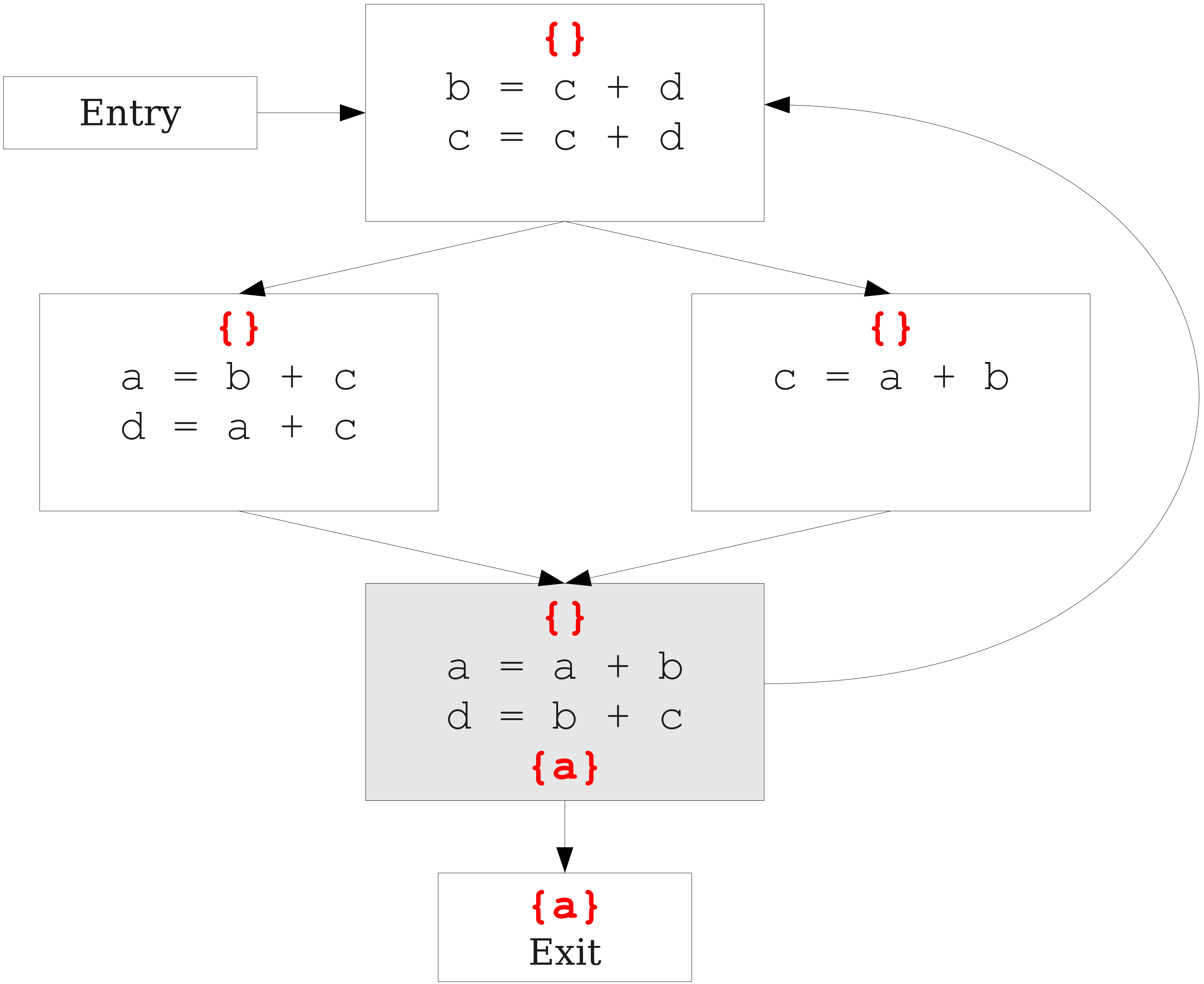
CFGs With Loops



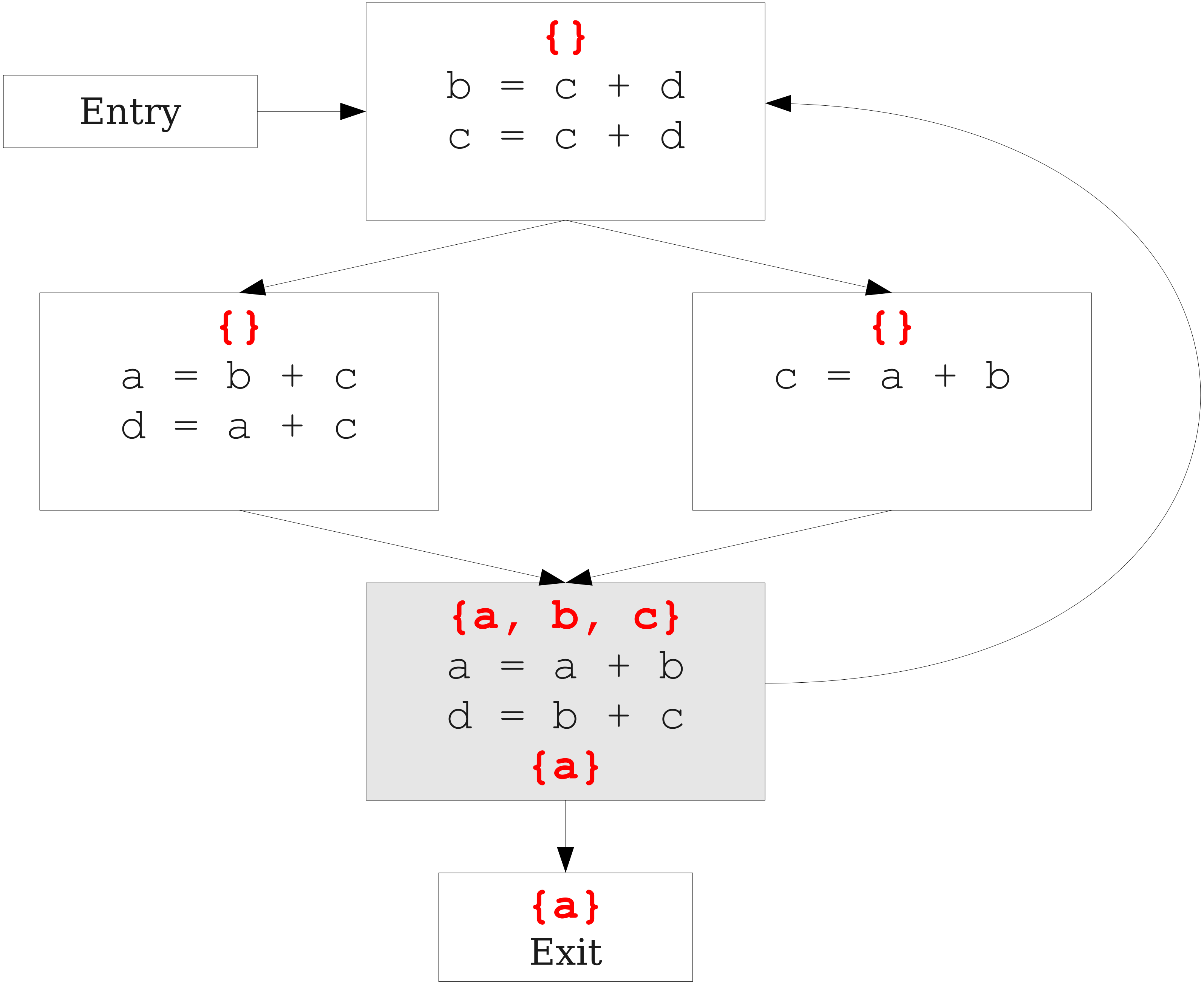
CFGs With Loops



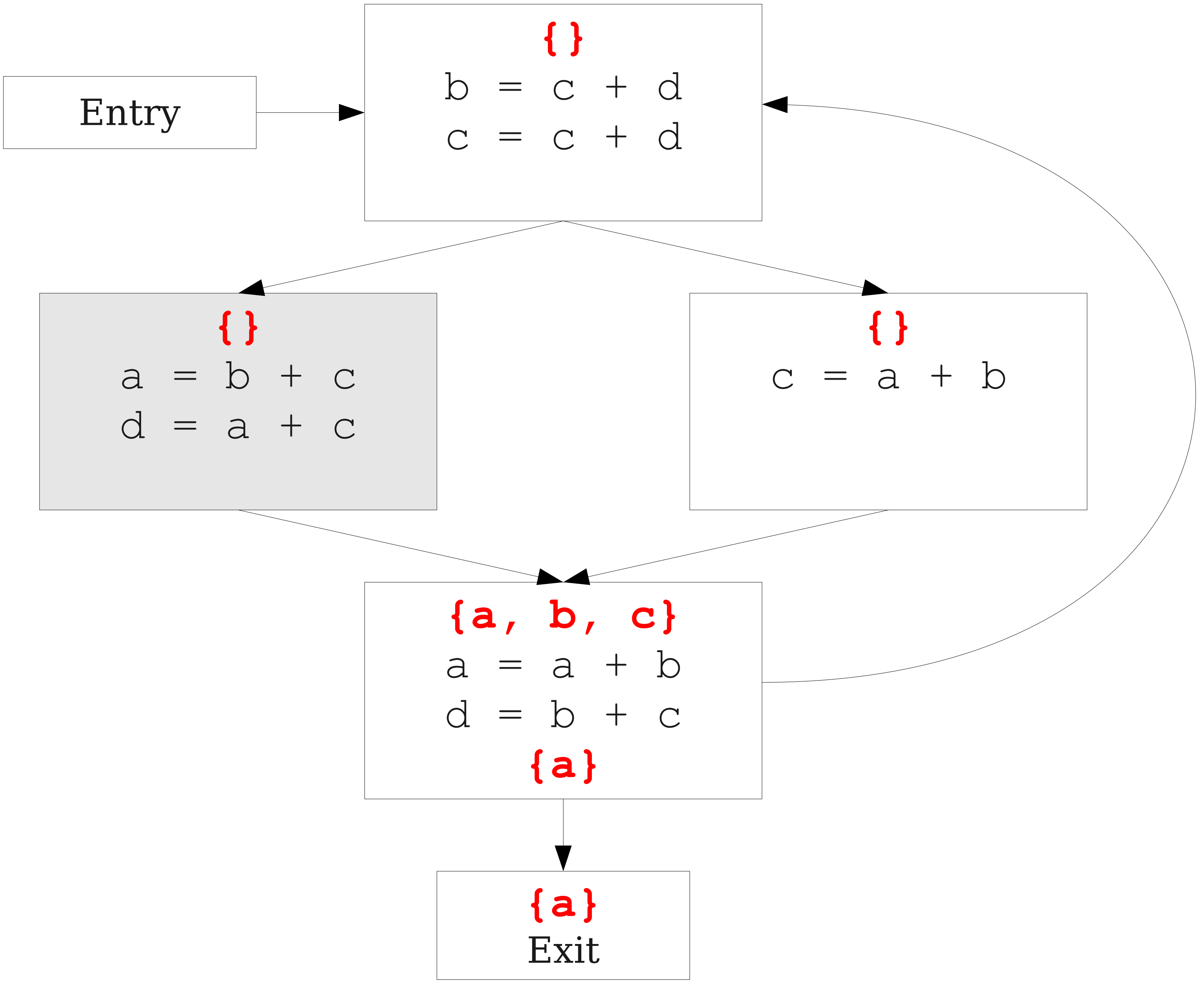
CFGs With Loops



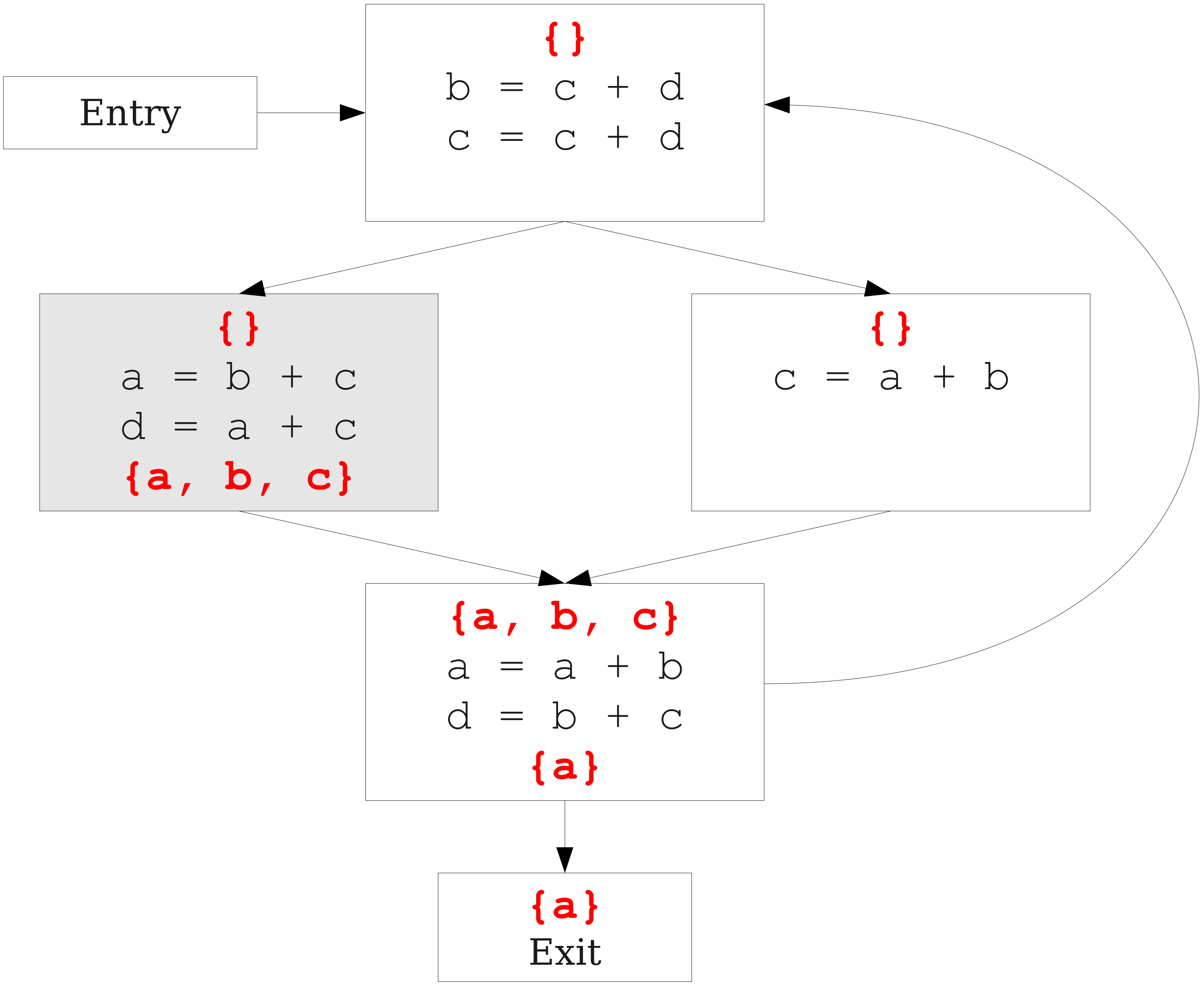
CFGs With Loops



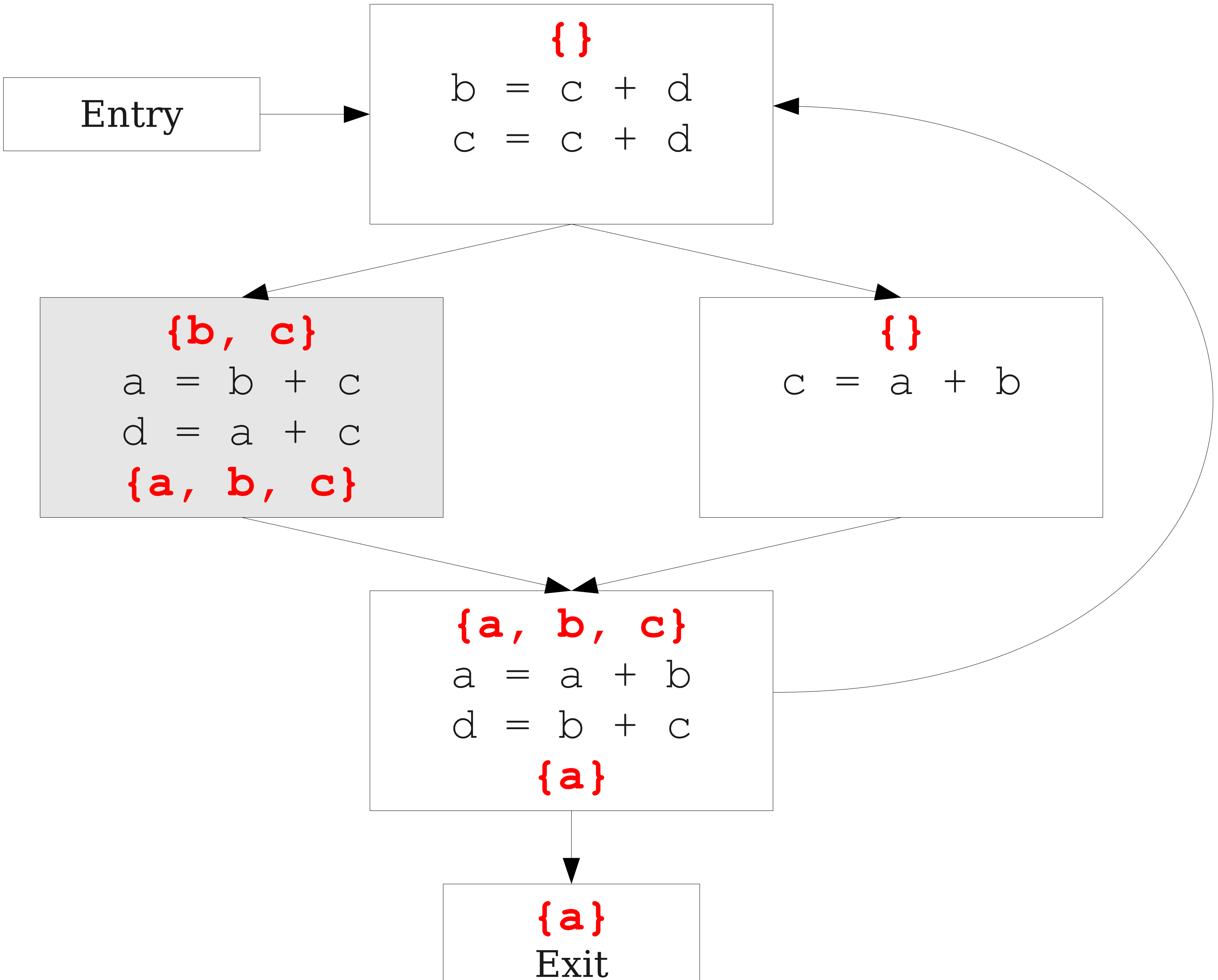
CFGs With Loops



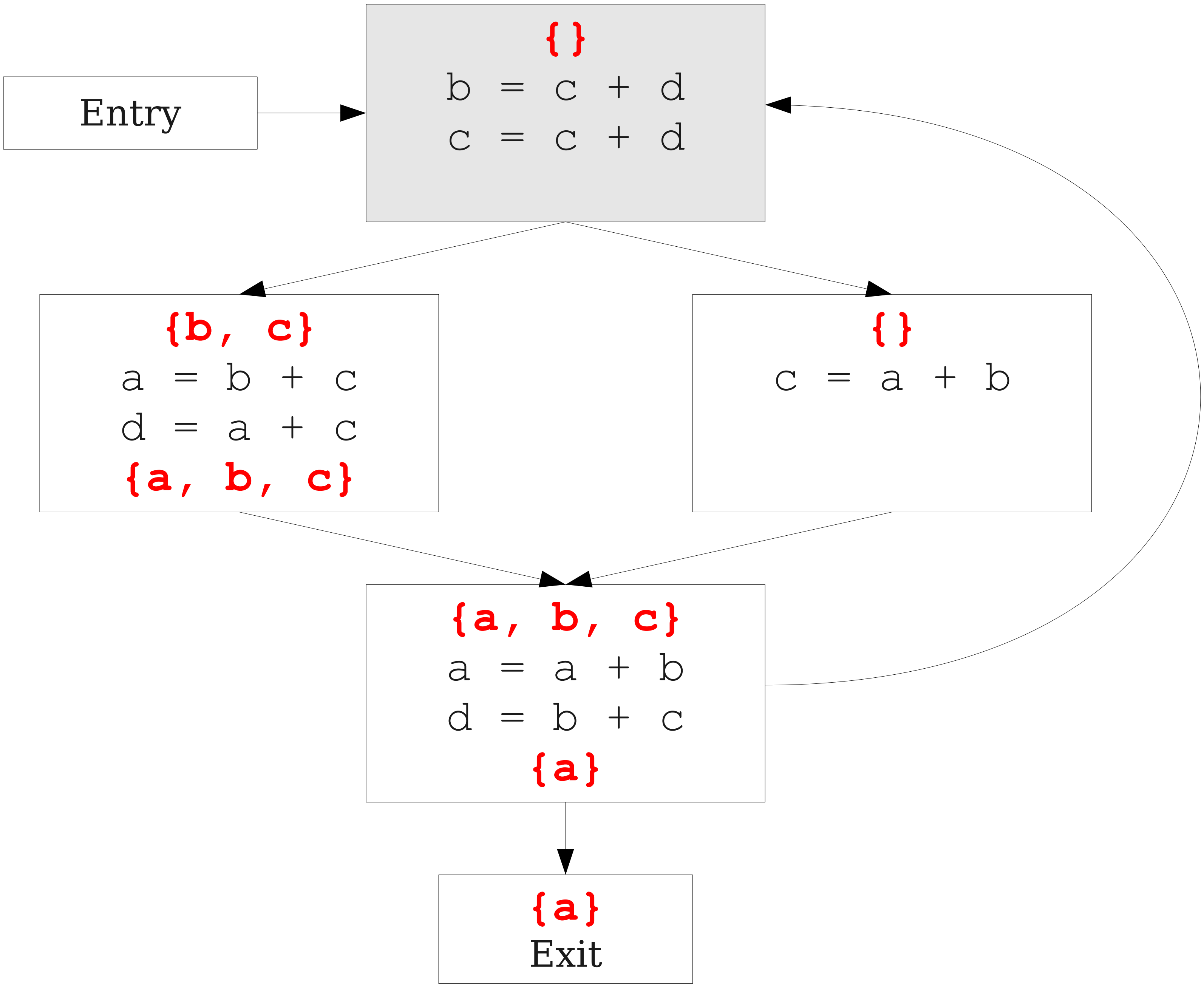
CFGs With Loops



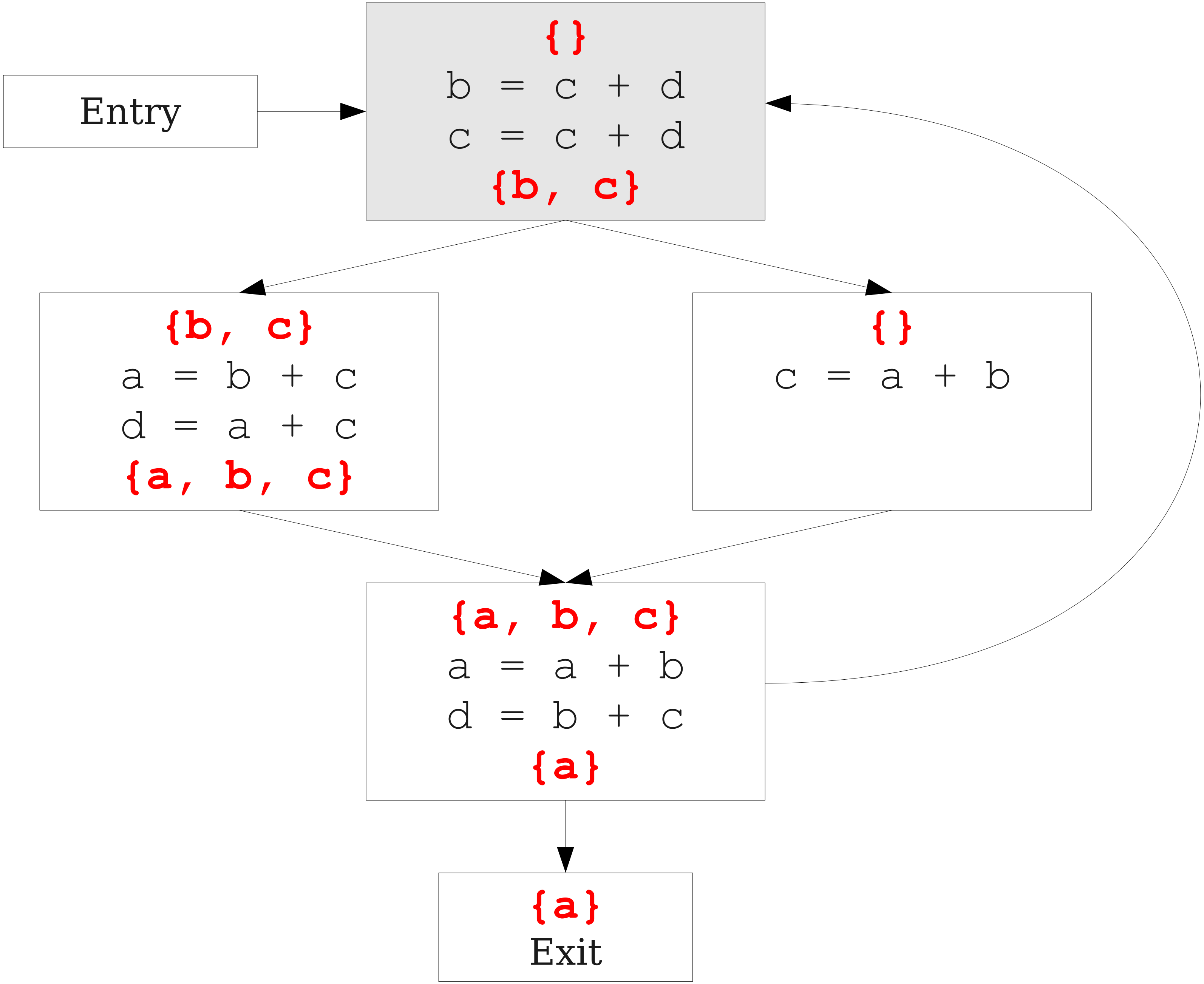
CFGs With Loops



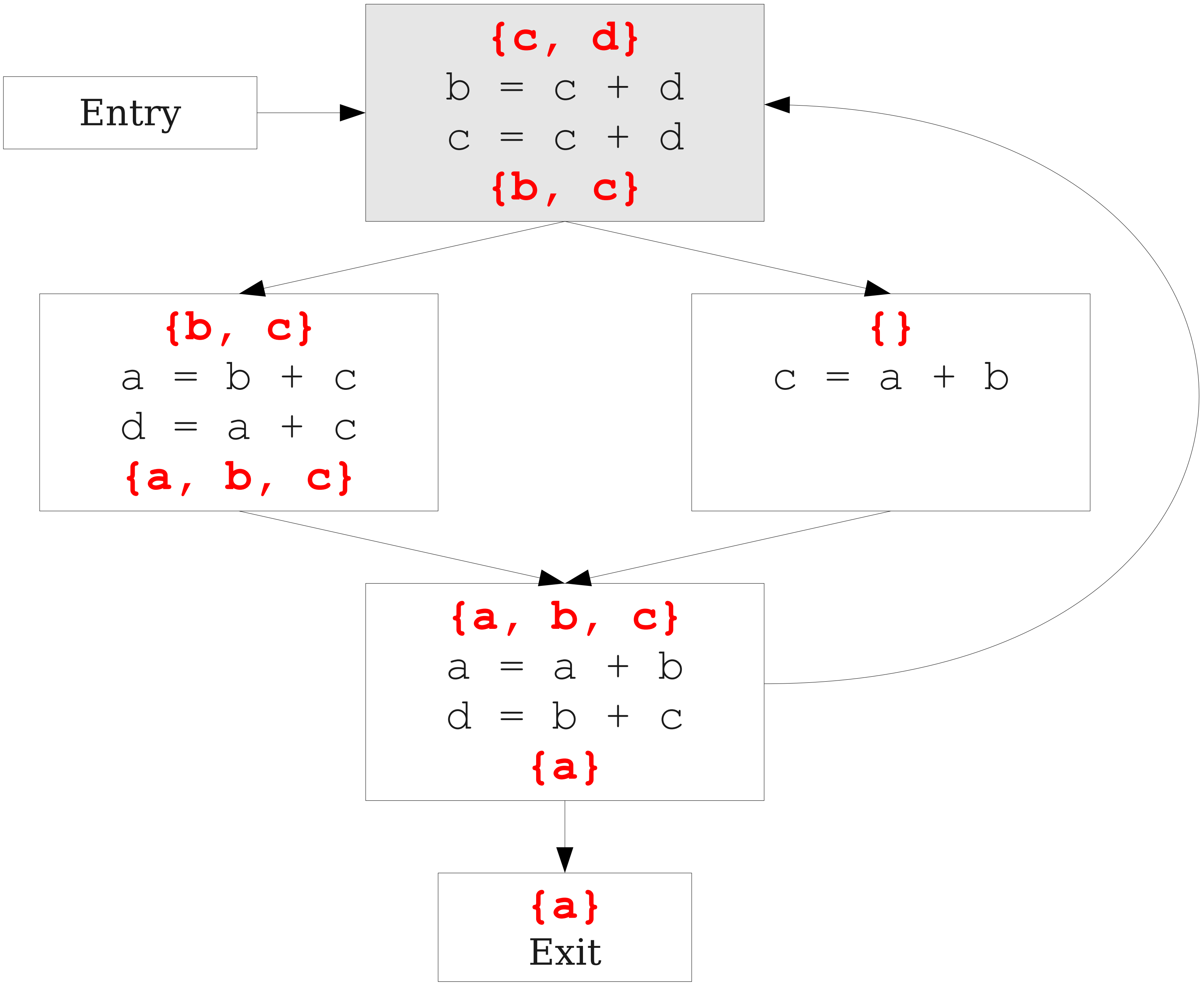
CFGs With Loops



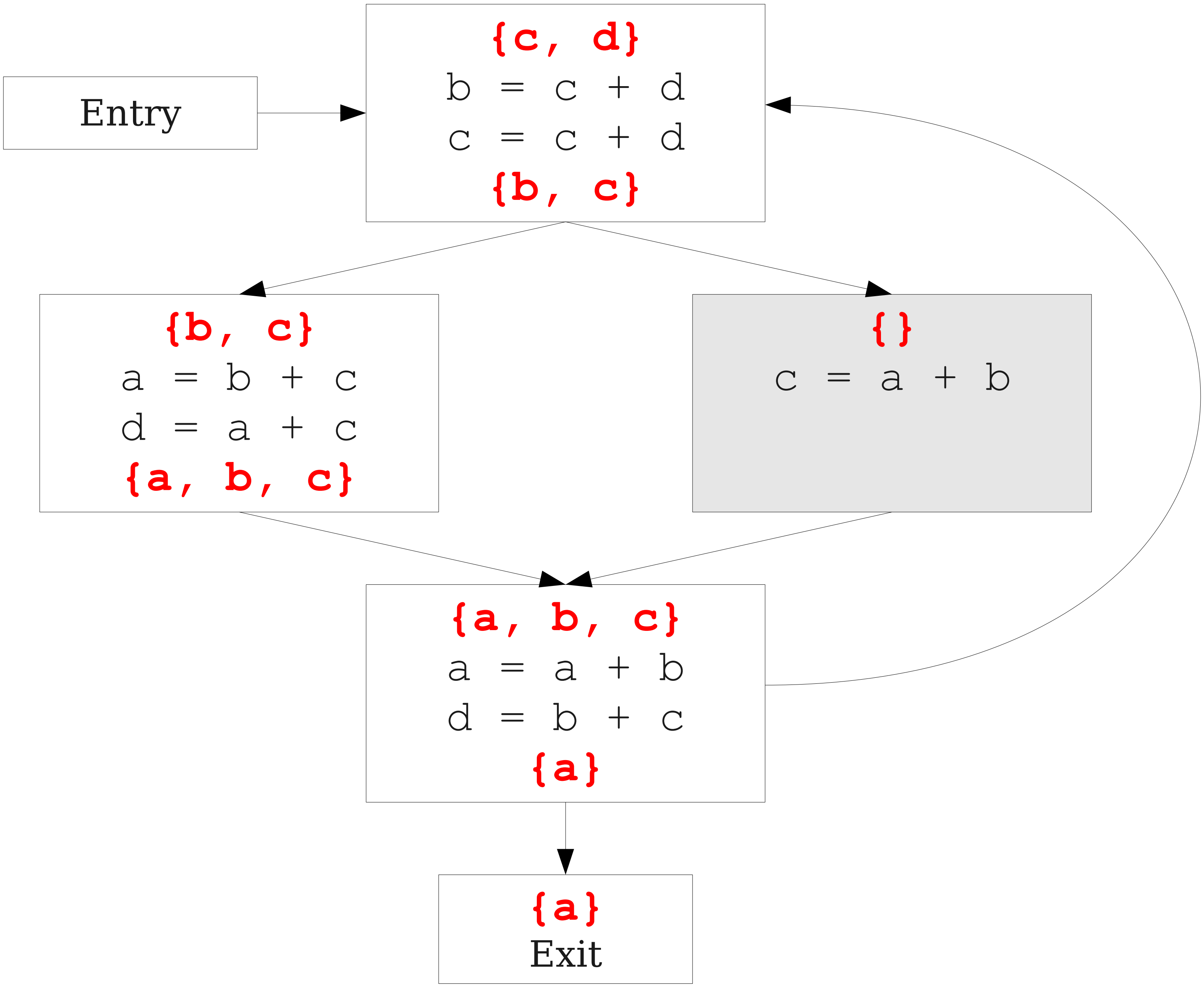
CFGs With Loops



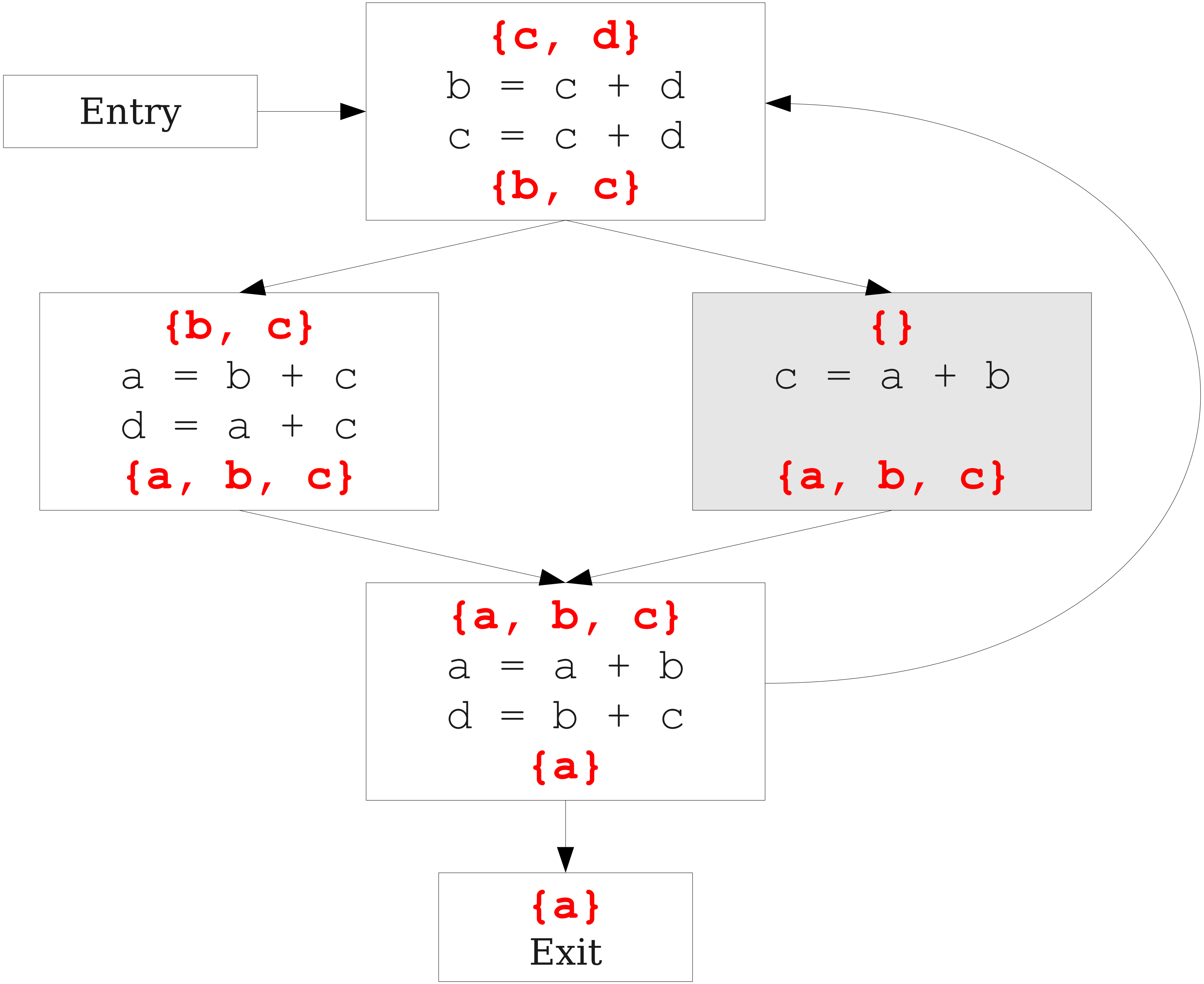
CFGs With Loops



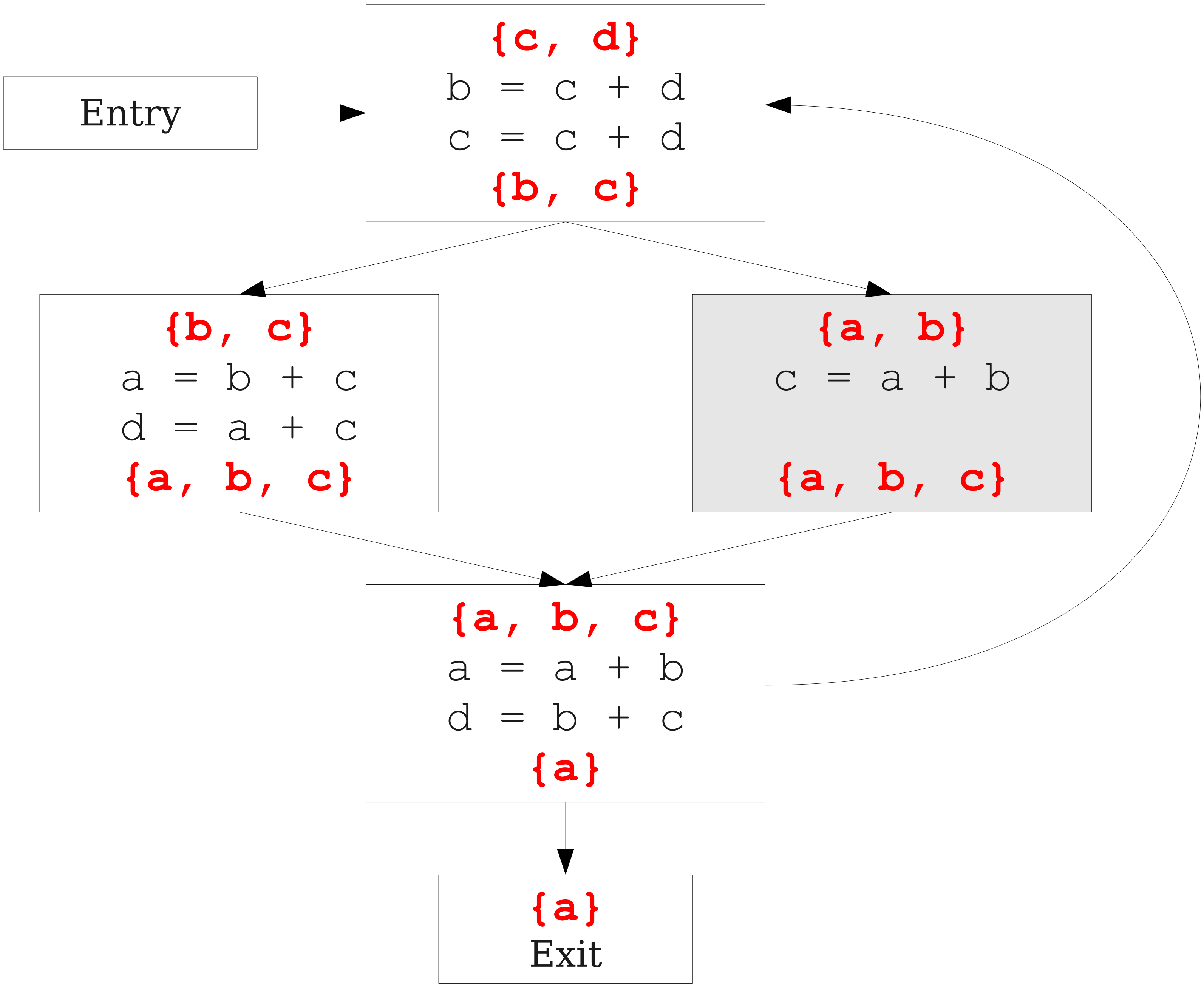
CFGs With Loops



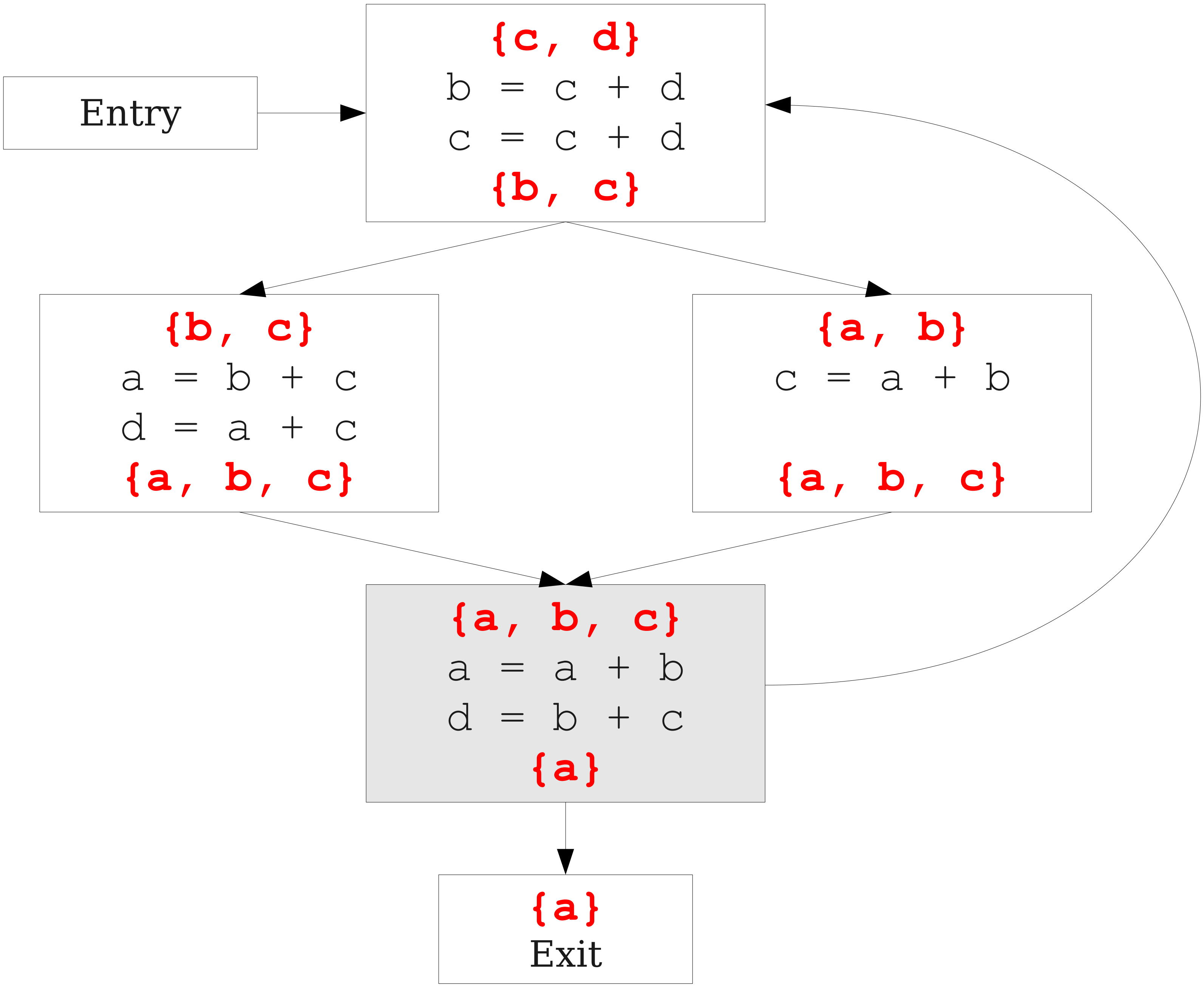
CFGs With Loops



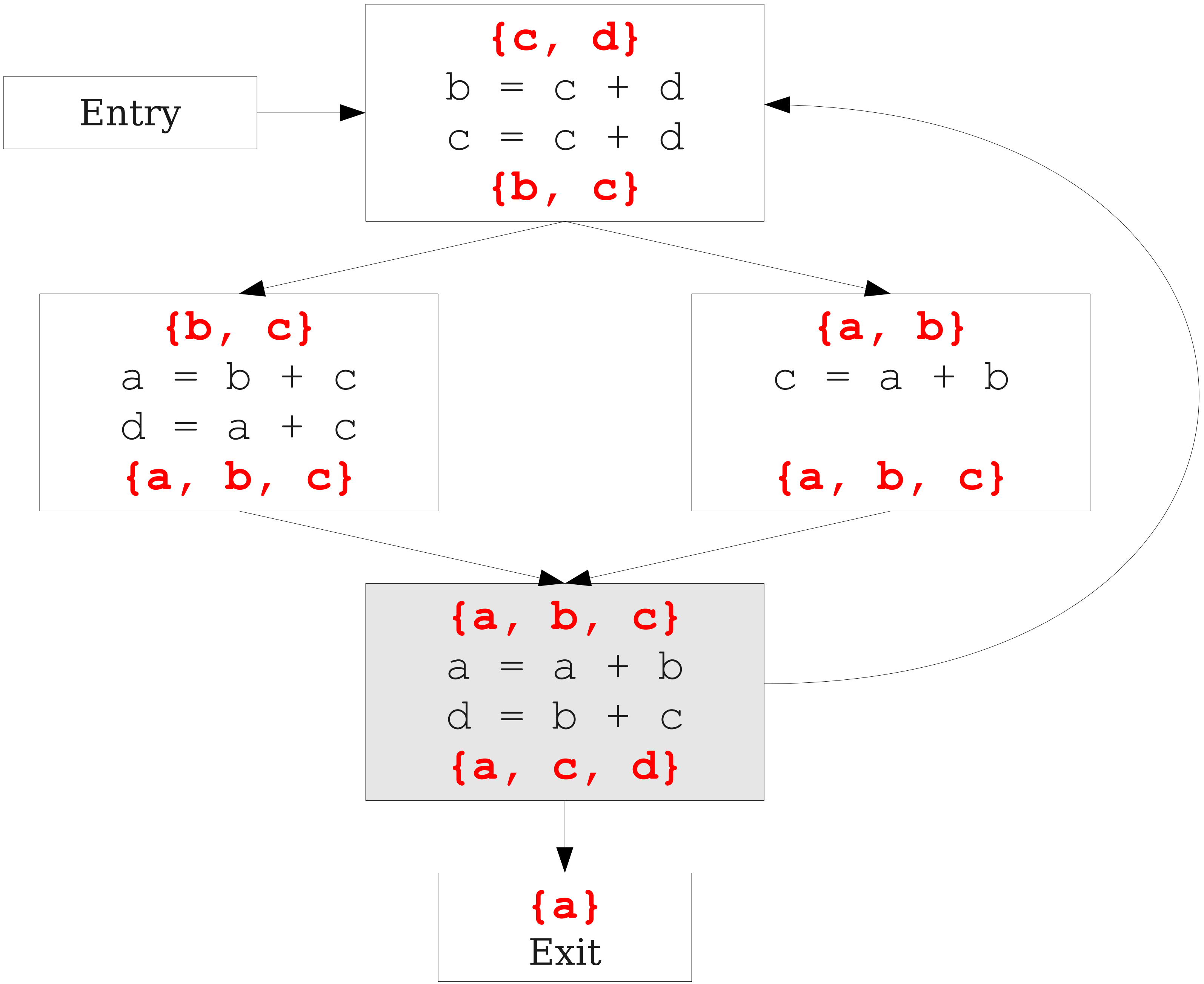
CFGs With Loops



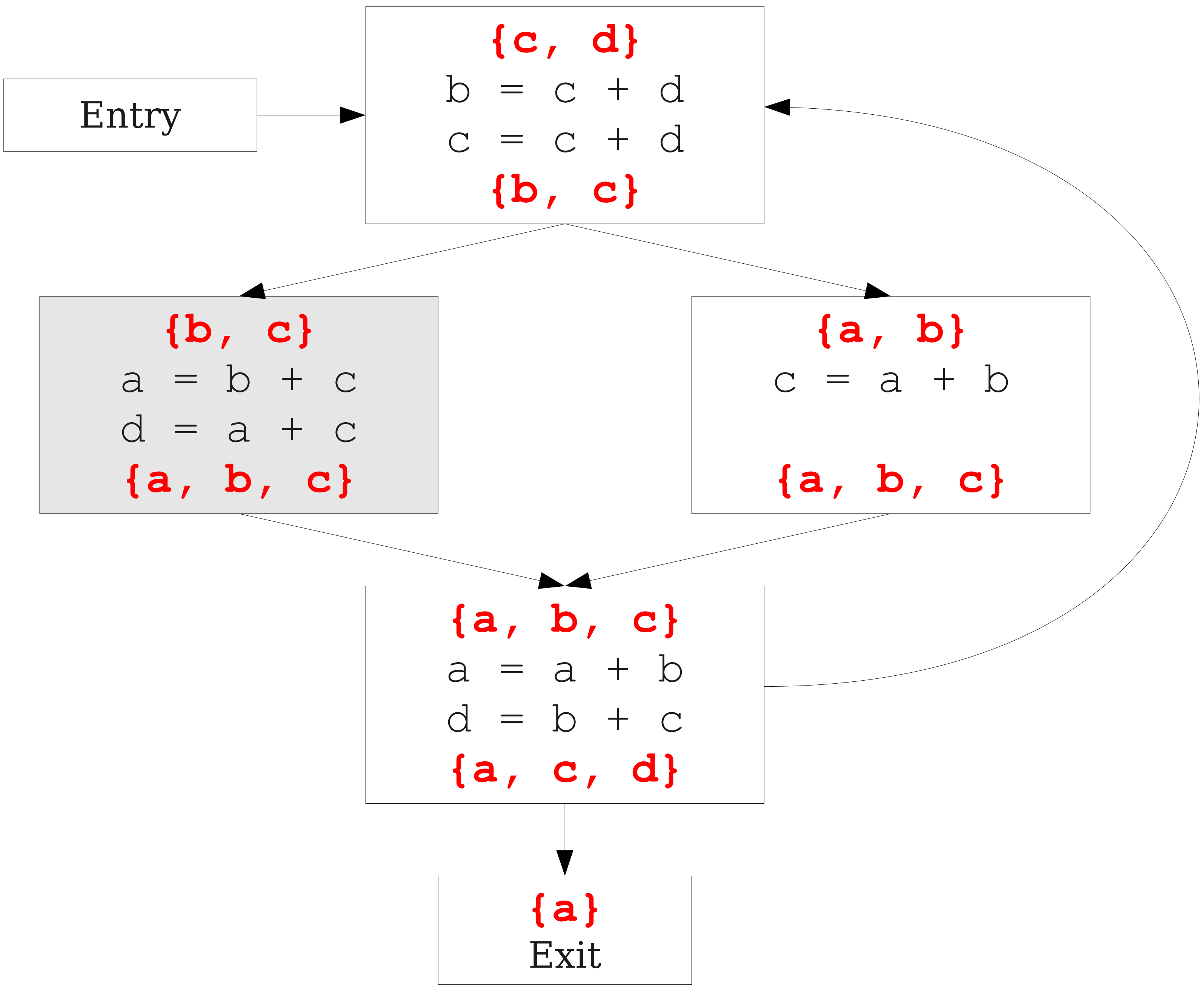
CFGs With Loops



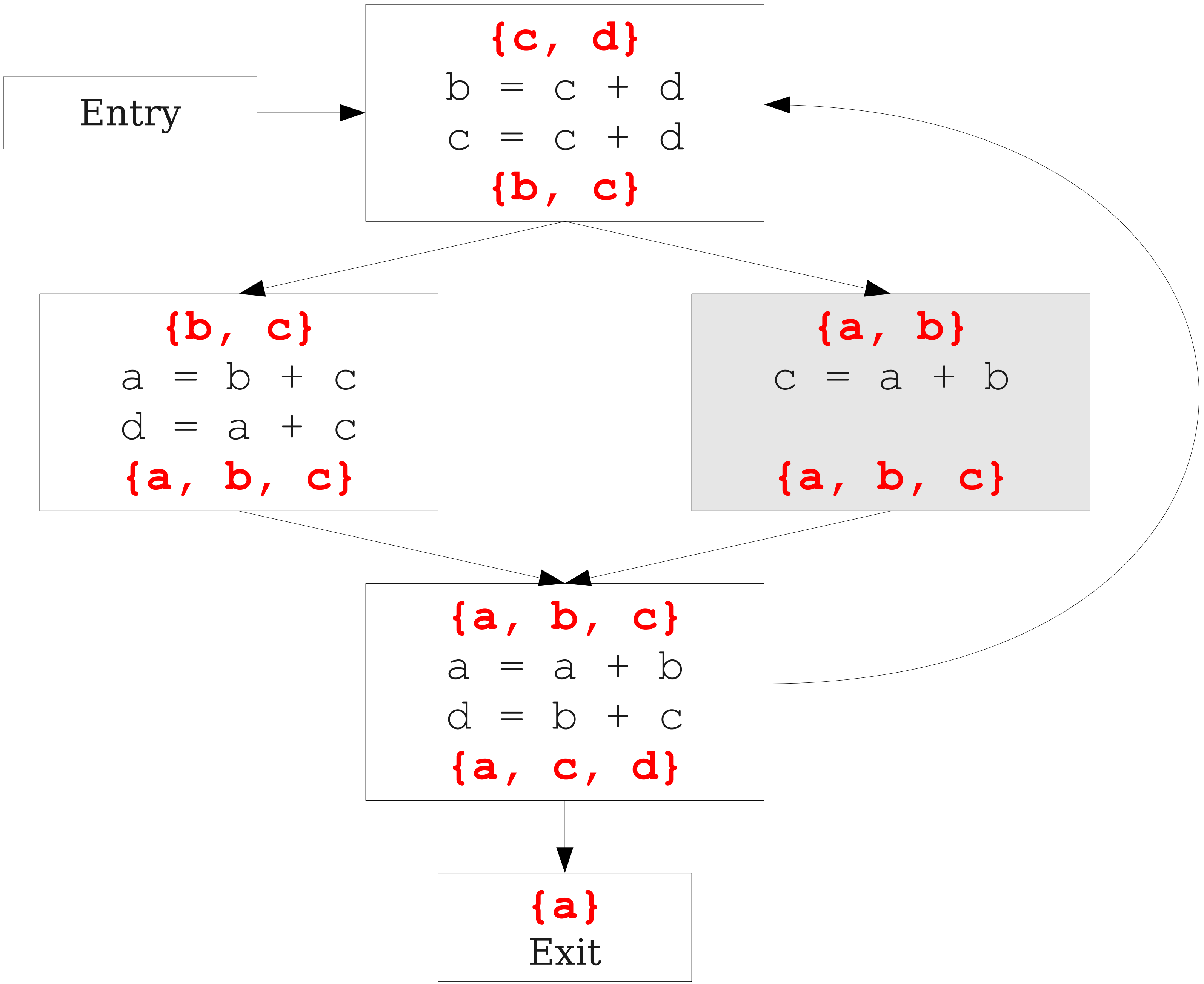
CFGs With Loops



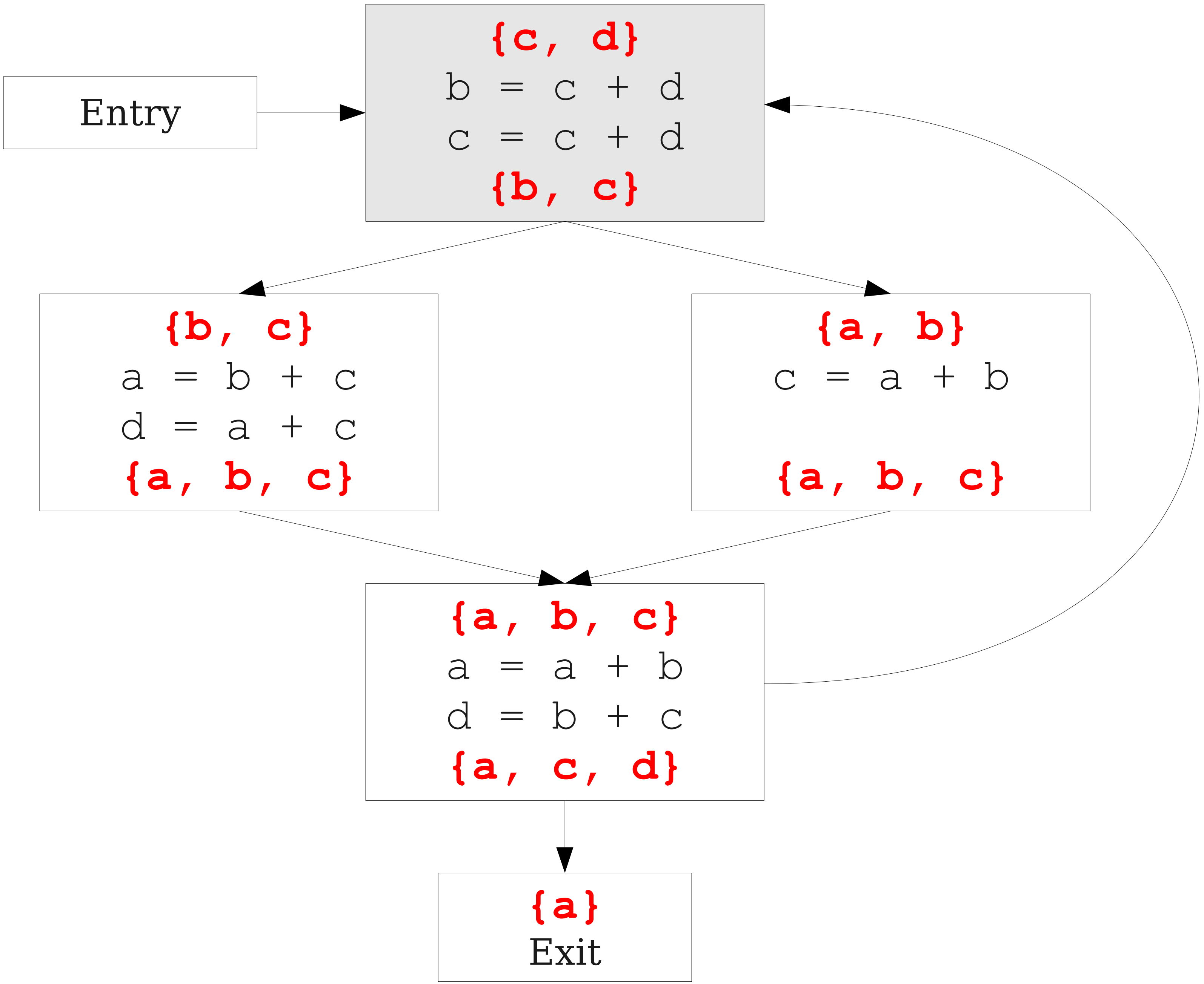
CFGs With Loops



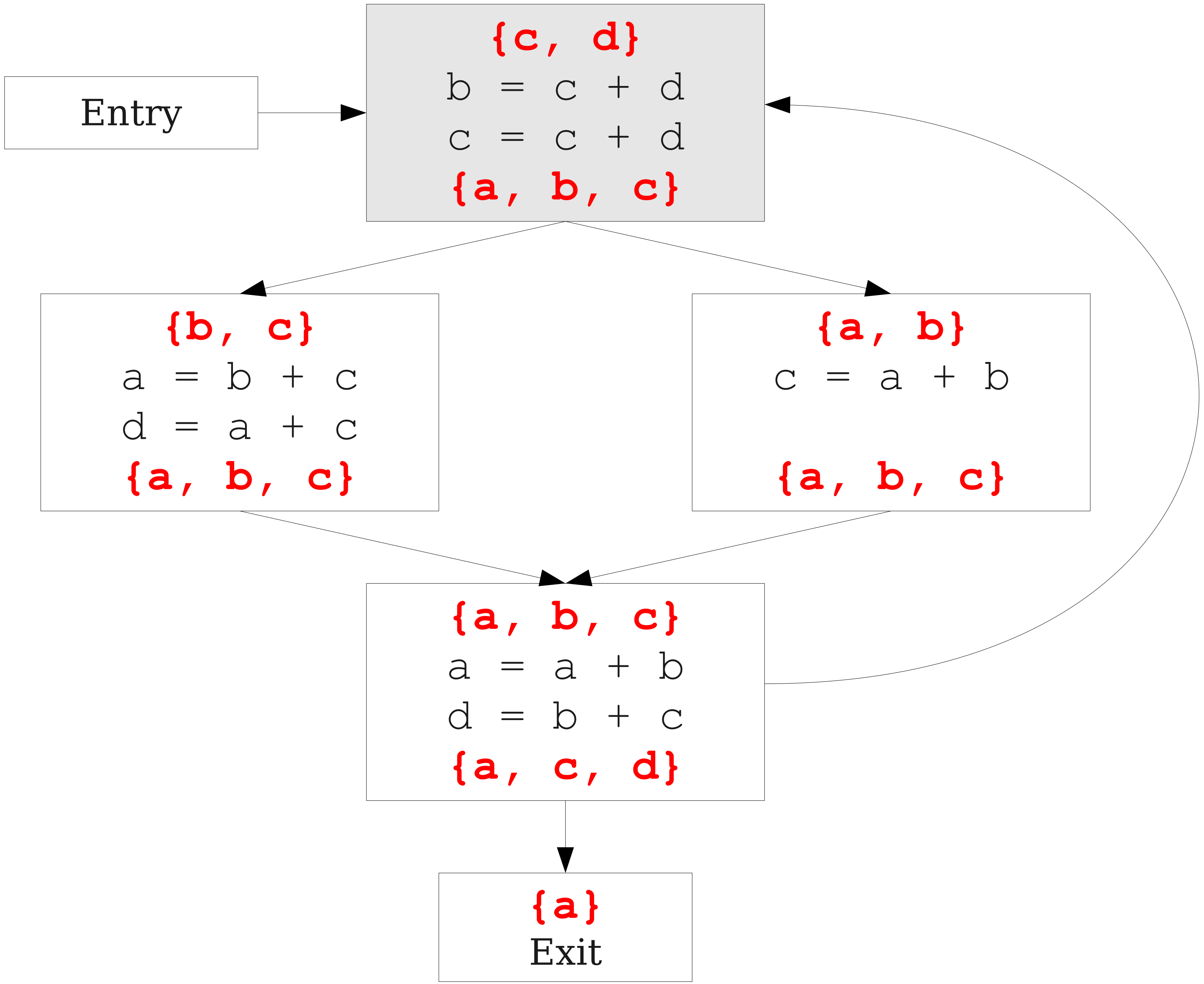
CFGs With Loops



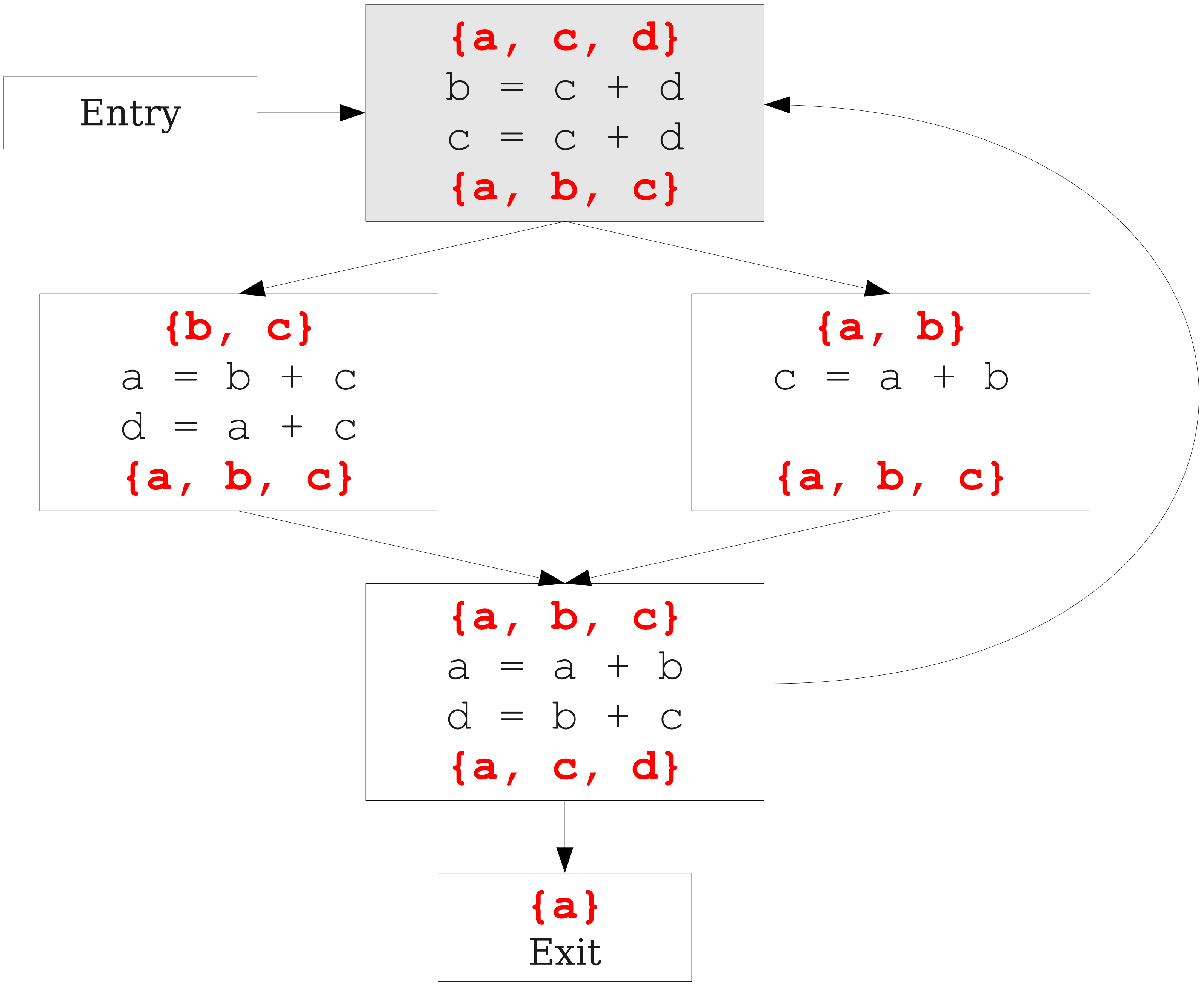
CFGs With Loops



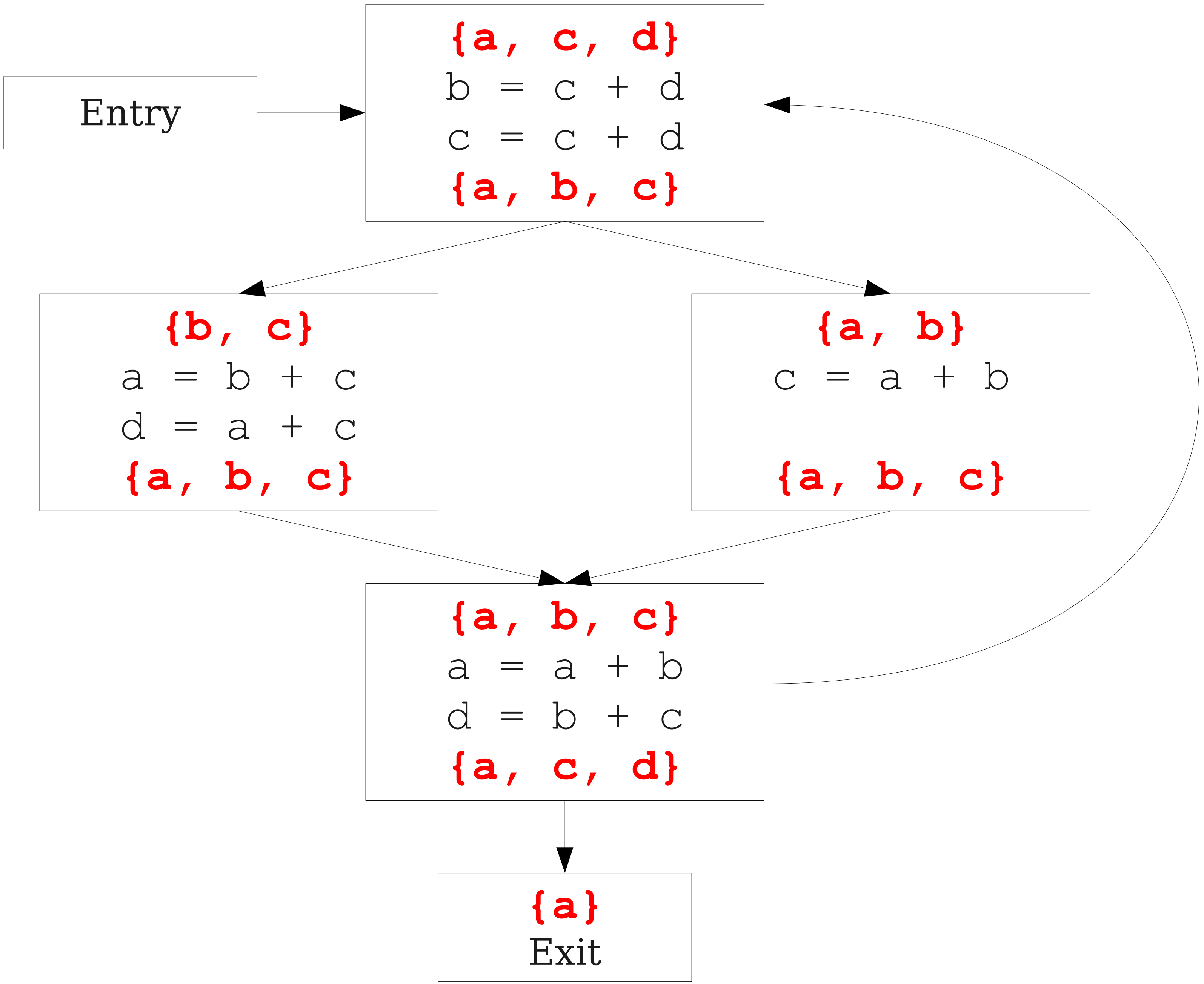
CFGs With Loops



CFGs With Loops



CFGs With Loops



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!**

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.

Theory to the Rescue

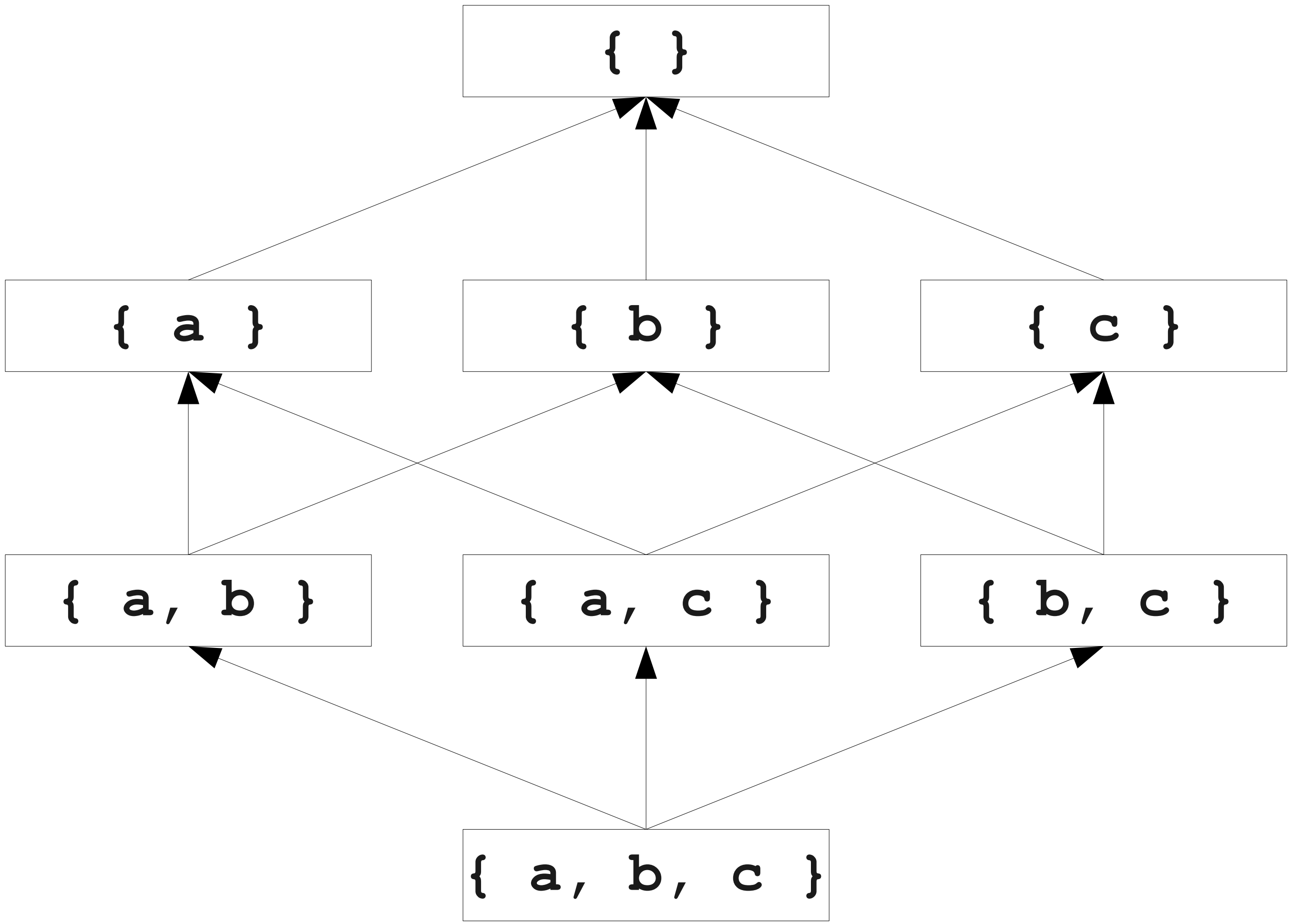
- Building up all of the machinery to design this analysis was tricky.
- The key ideas, however, are mostly independent of the analysis:
  - We need to be able to compute functions describing the behavior of each statement.
  - We need to be able to merge several subcomputations together.
  - We need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

Meet Semilattices

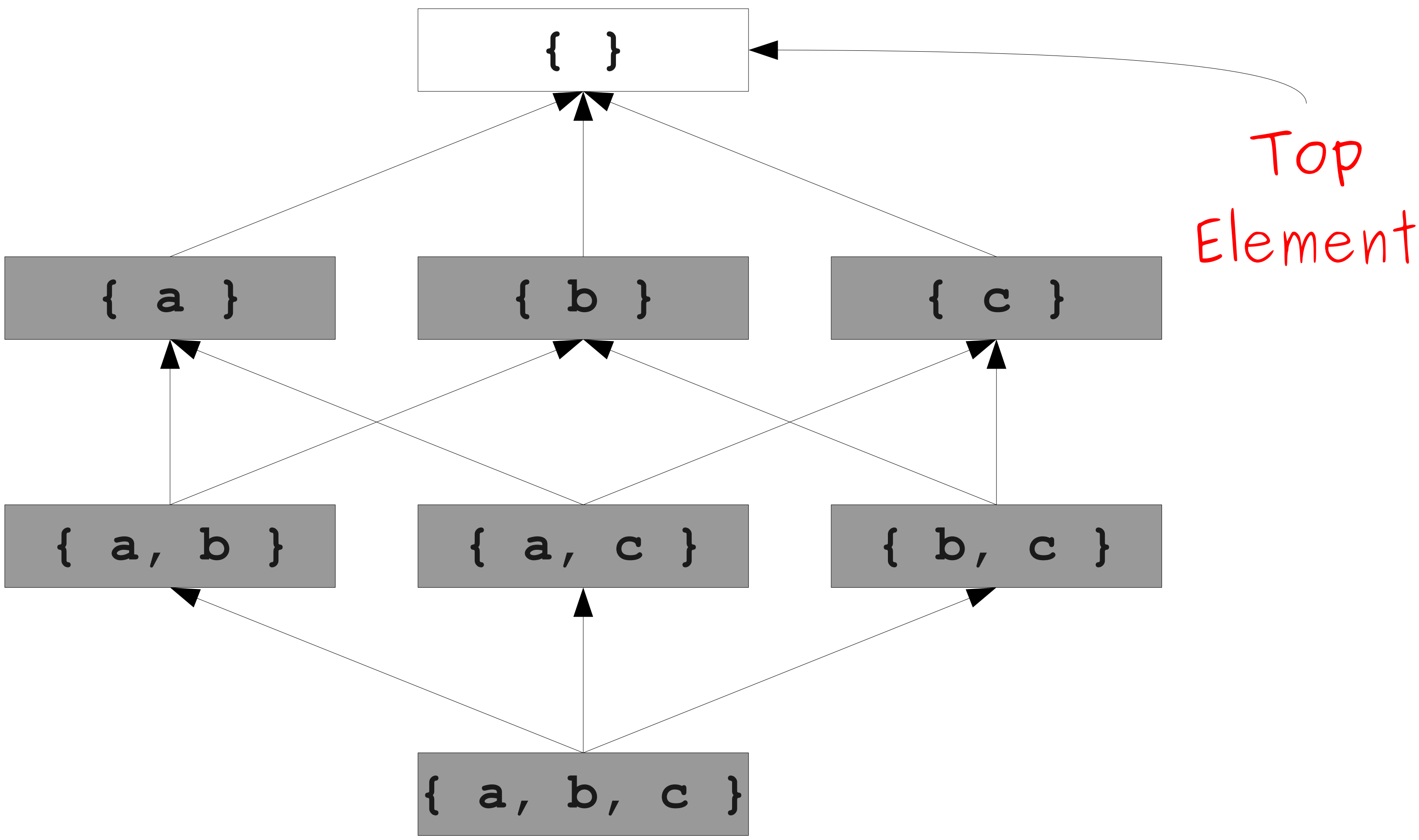
- A **meet semilattice** is a ordering defined on a set of elements.
- Any two elements have some **meet** that is the largest element smaller than both elements.
- There is a unique **top element**, which is larger than all other elements.
- Intuitively:
  - The meet of two elements represents combining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

Meet Semilattices for Liveness

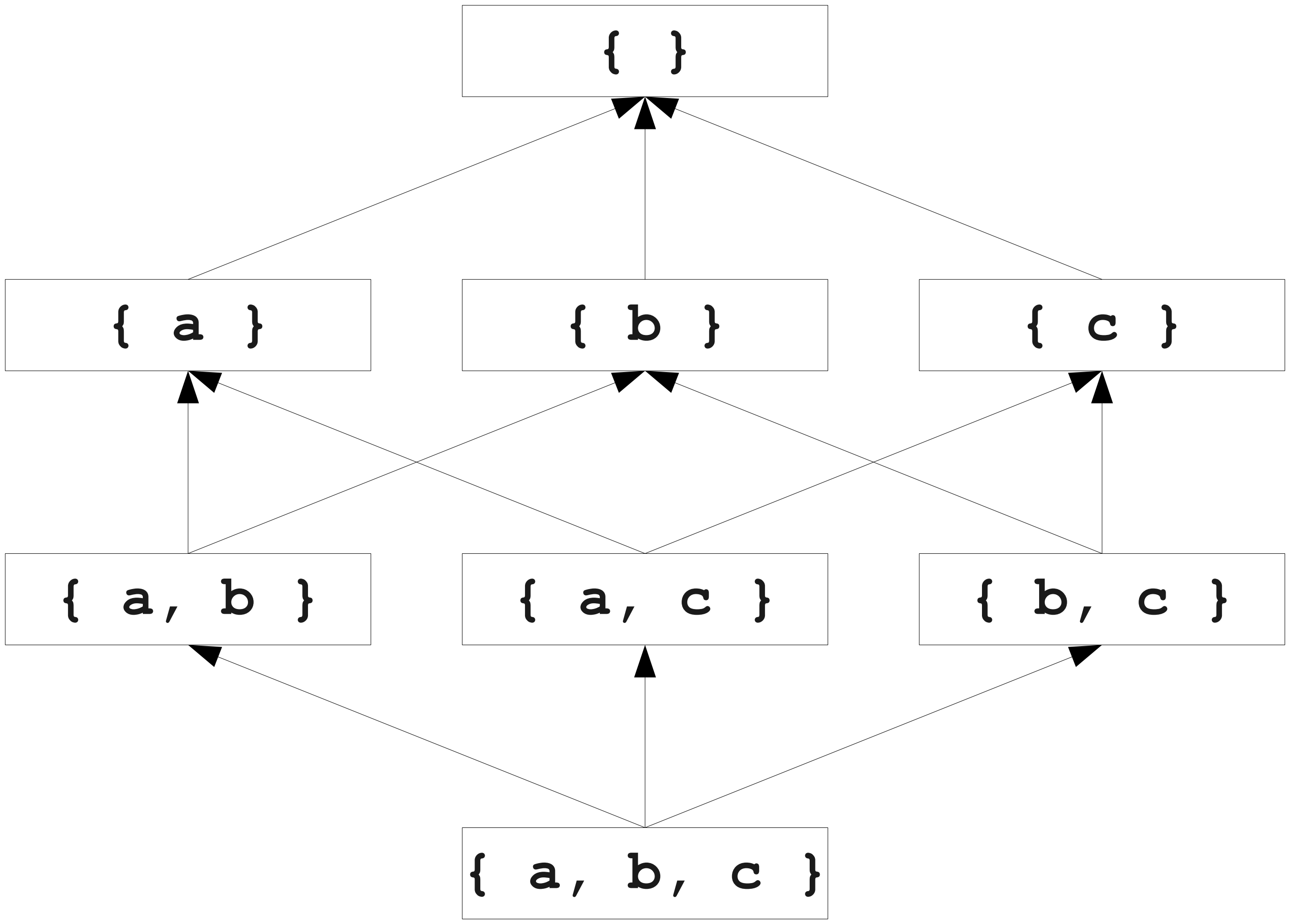
Meet Semilattices for Liveness



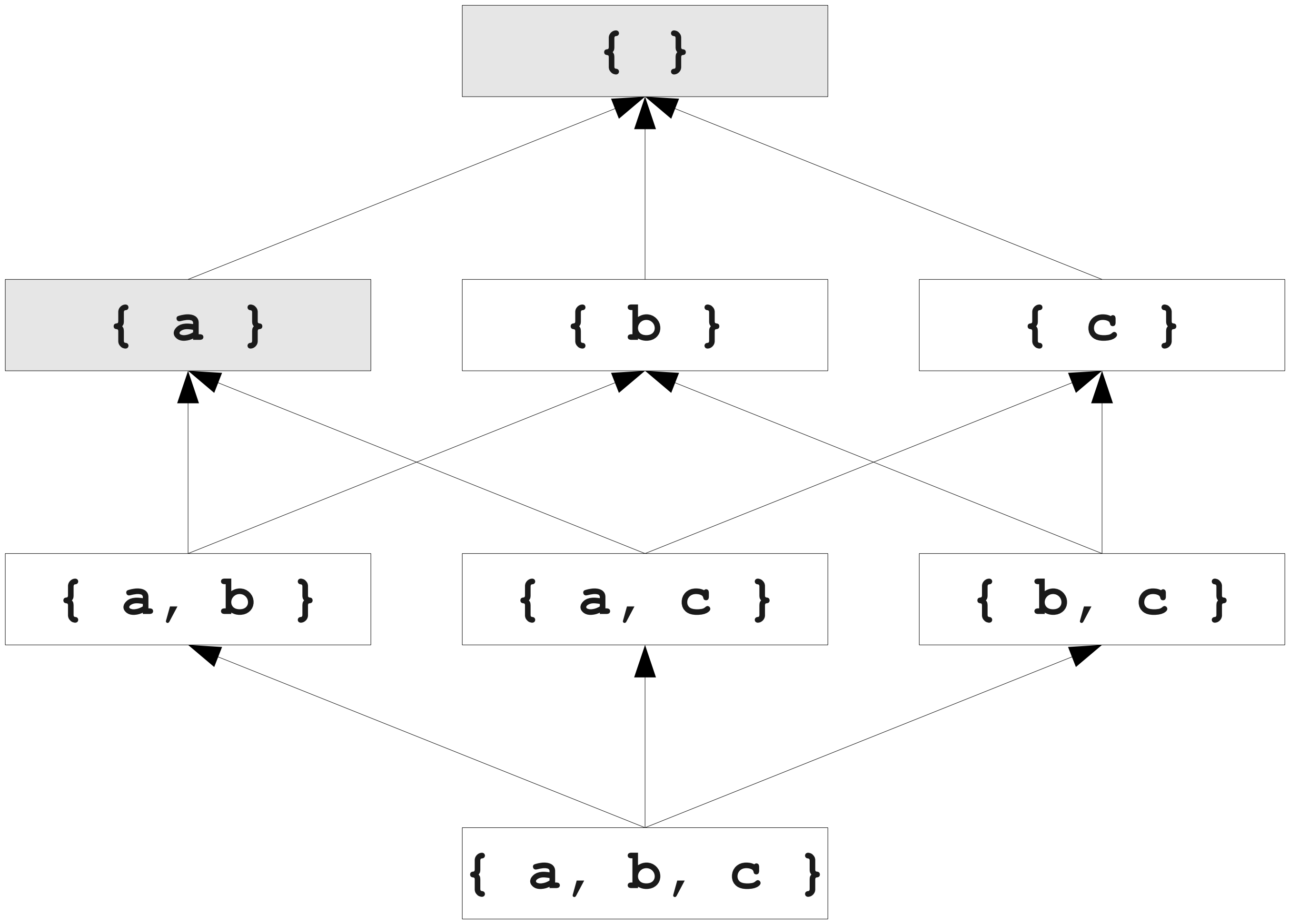
Meet Semilattices for Liveness



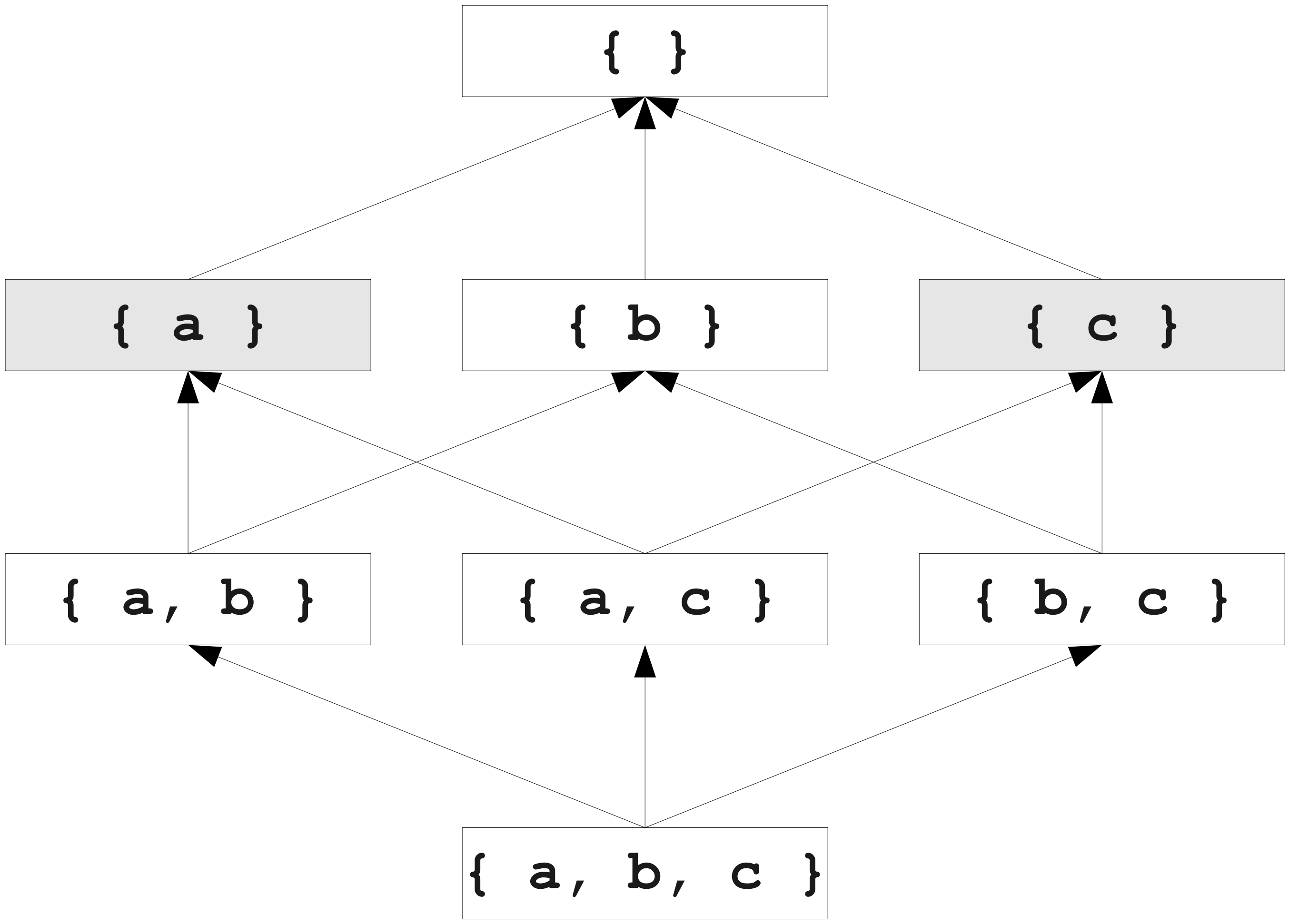
Meet Semilattices for Liveness



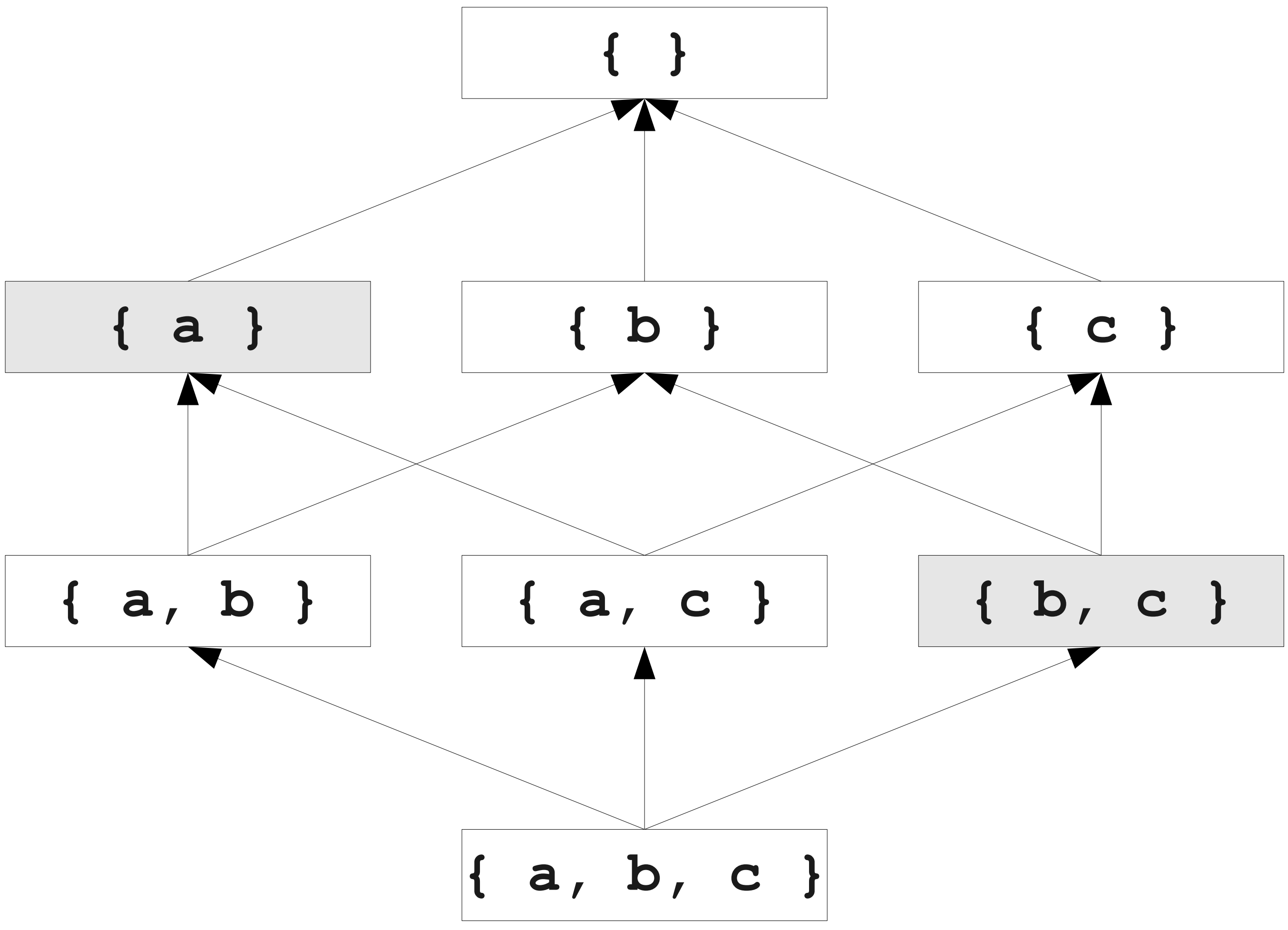
Meet Semilattices for Liveness



Meet Semilattices for Liveness



Meet Semilattices for Liveness



Formal Definitions

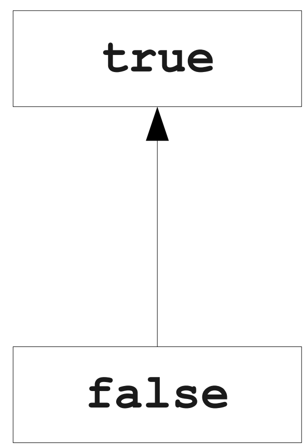
- A **meet semilattice** is a pair $(D, \wedge)$, where
  - D is a domain of elements.
  - $\wedge$ is a **meet operator** that is
    - **idempotent:** $x \wedge x = x$
    - **commutative:** $x \wedge y = y \wedge x$
    - **associative:** $(x \wedge y) \wedge z = x \wedge (y \wedge z)$
- If $x \wedge y = z$, 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.

An Example Semilattice

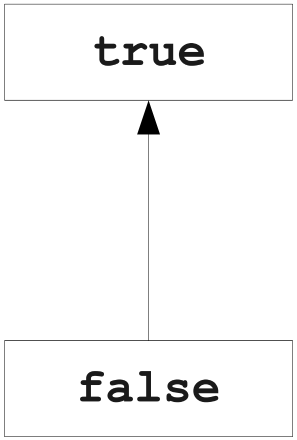
- The set of natural numbers and the **max** function.
- Idempotent
  - **max**{a, a} = a
- Commutative
  - **max**{a, b} = **max**{b, a}
- Associative
  - **max**{a, **max**{b, c}} = **max**{**max**{a, b}, c}
- Top element is 0:
  - **max**{0, a} = a

Is this a Meet Semilattice?

Is this a Meet Semilattice?



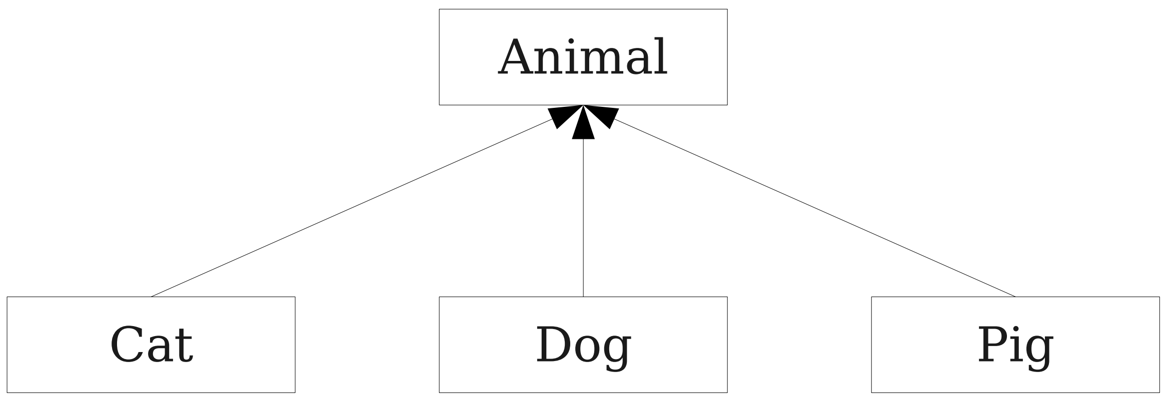
Is this a Meet Semilattice?



What is the meet operator here?

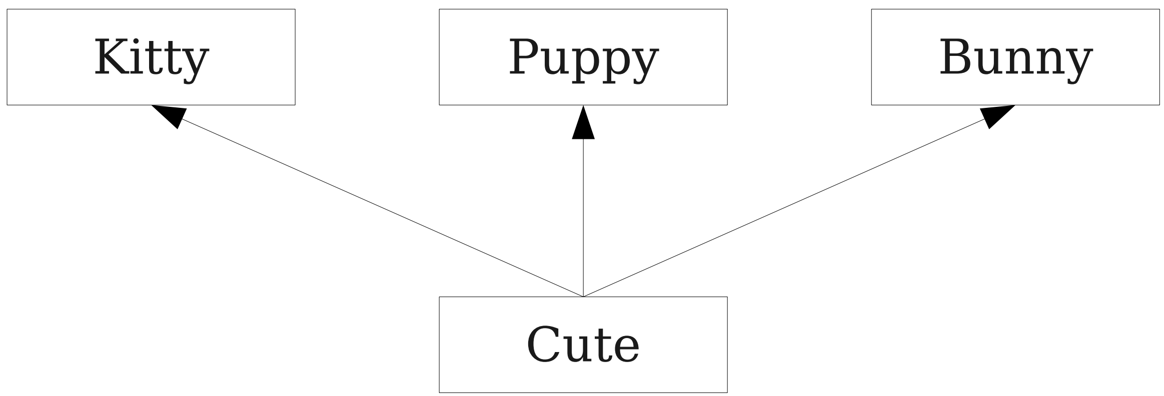
Is this a Meet Semilattice?

Is this a Meet Semilattice?

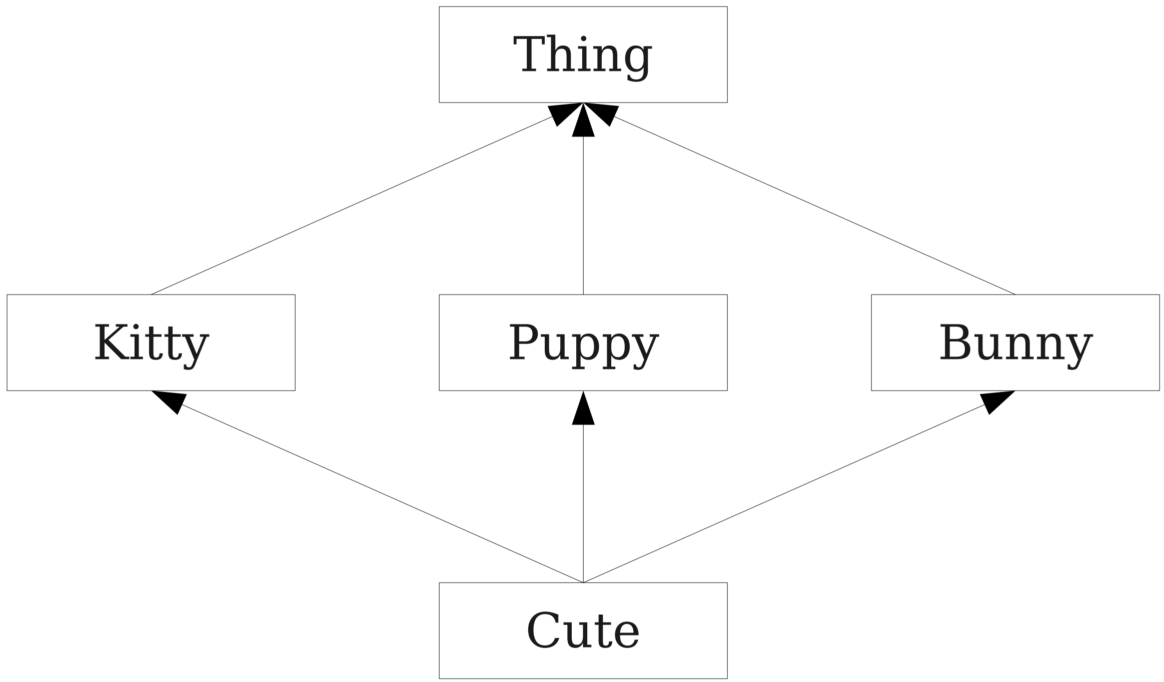


Is this a Meet Semilattice?

Is this a Meet Semilattice?



Is this a Meet Semilattice?



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: $\emptyset \cup x = x$

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.
- 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 $f : \mathbf{V} \rightarrow \mathbf{V}$
  - **I** is an initial value.
- The only difference from local analysis is the introduction of the meet operator.

Running Global Analyses

- Assume that (**D**, **V**, **∧**, **F**, **I**) 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**] = 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.

For Comparison

- Set IN[**s**] = $\top$ for all statement **s**.
- Set IN[**exit**] = I.
- Repeat until no changes occur:
  - For each statement **s**:
    - Set OUT[**s**] = IN[$\boldsymbol{x}_1$]$\wedge\ldots\wedge$IN[$\boldsymbol{x}_n$] where $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$ are successors of **s**.
    - Set IN[**s**] = f_s (OUT[**s**])

- Set IN[**s**] = { } for each statement **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[$\boldsymbol{x}$] for each successor $\boldsymbol{x}$ of **s**.
    - Set IN[**s**] to (OUT[**s**] – **a**) ∪ {**b**, **c**}.

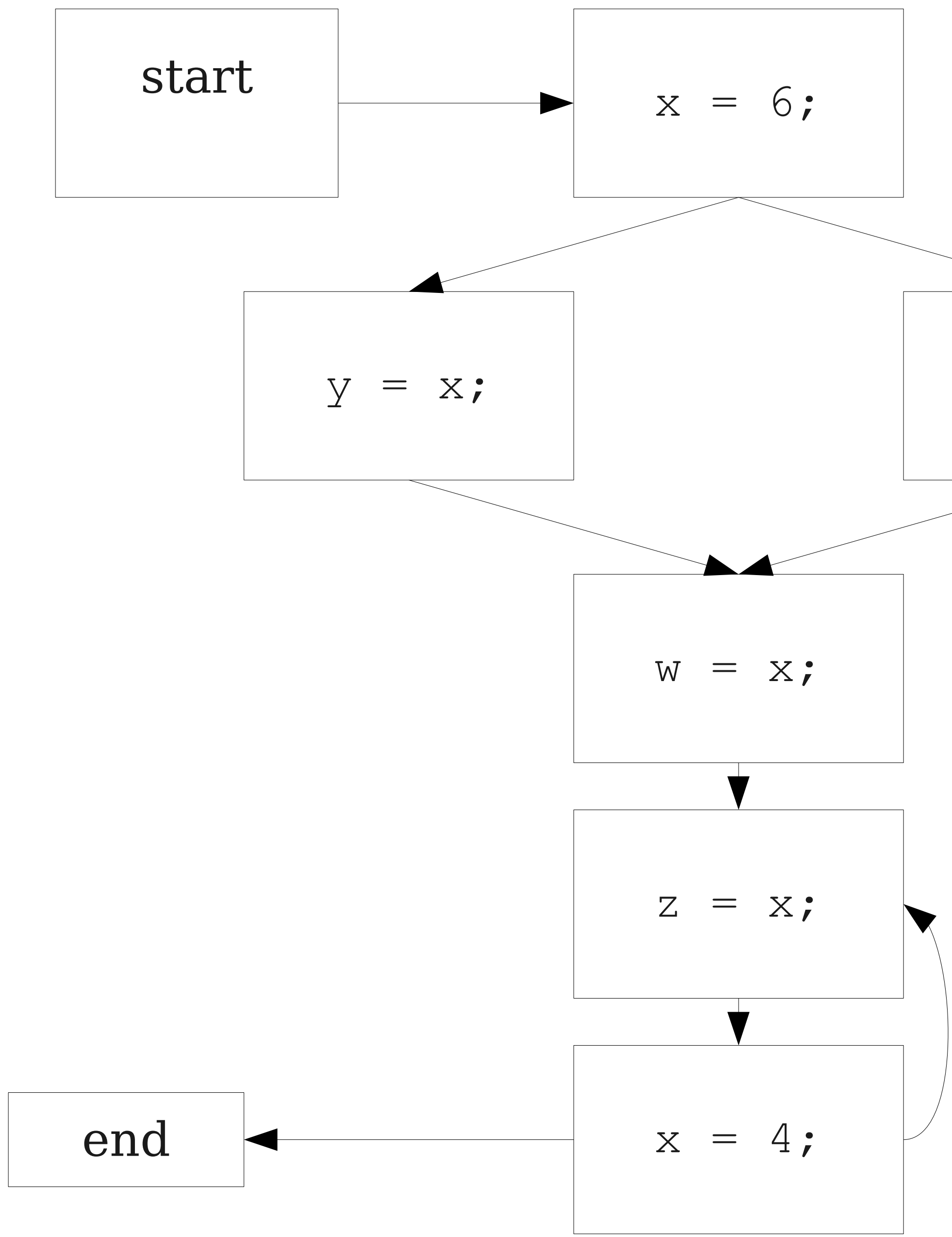
The Dataflow Framework

- 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.

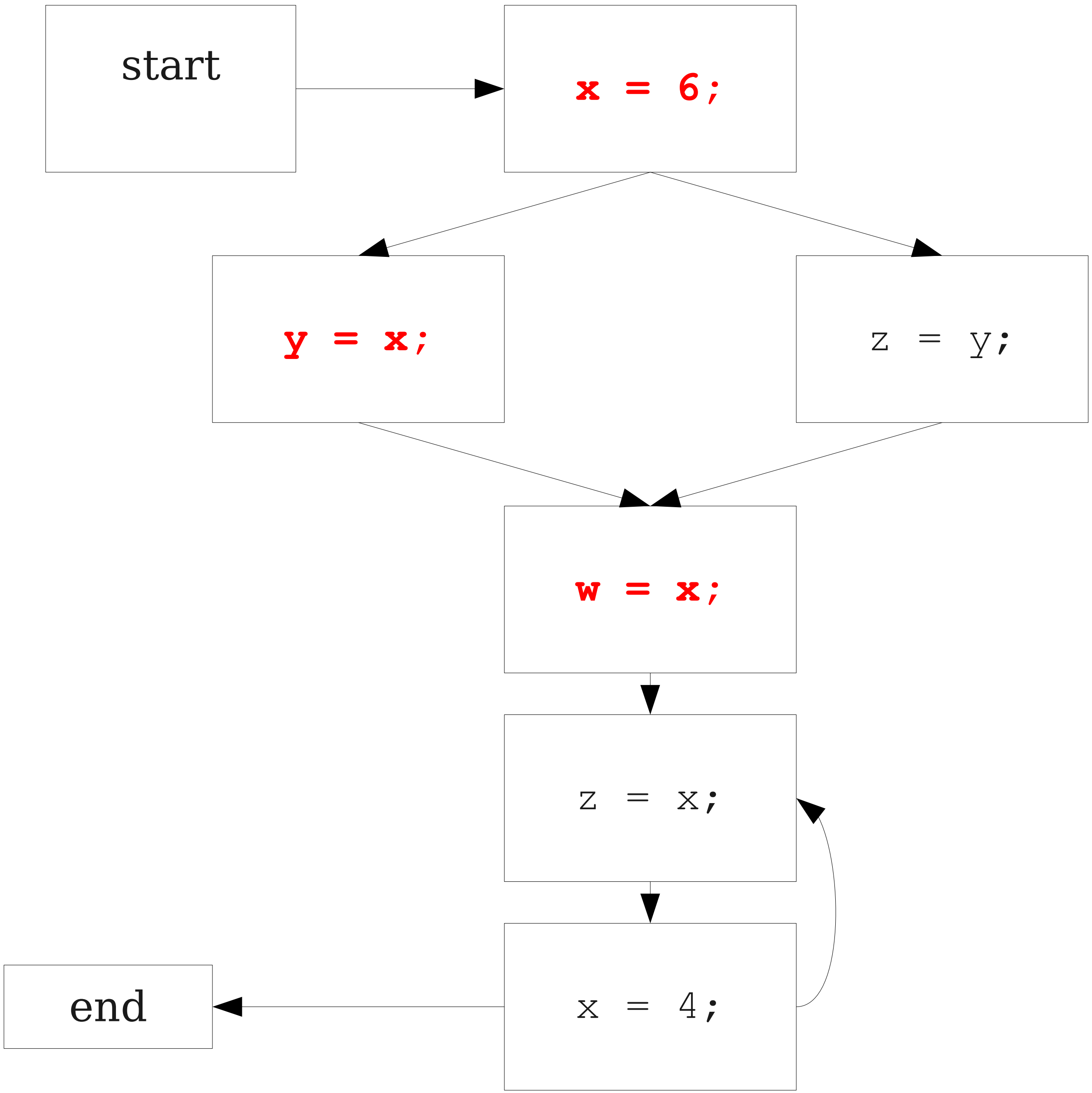
Global Constant Propagation

- **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.

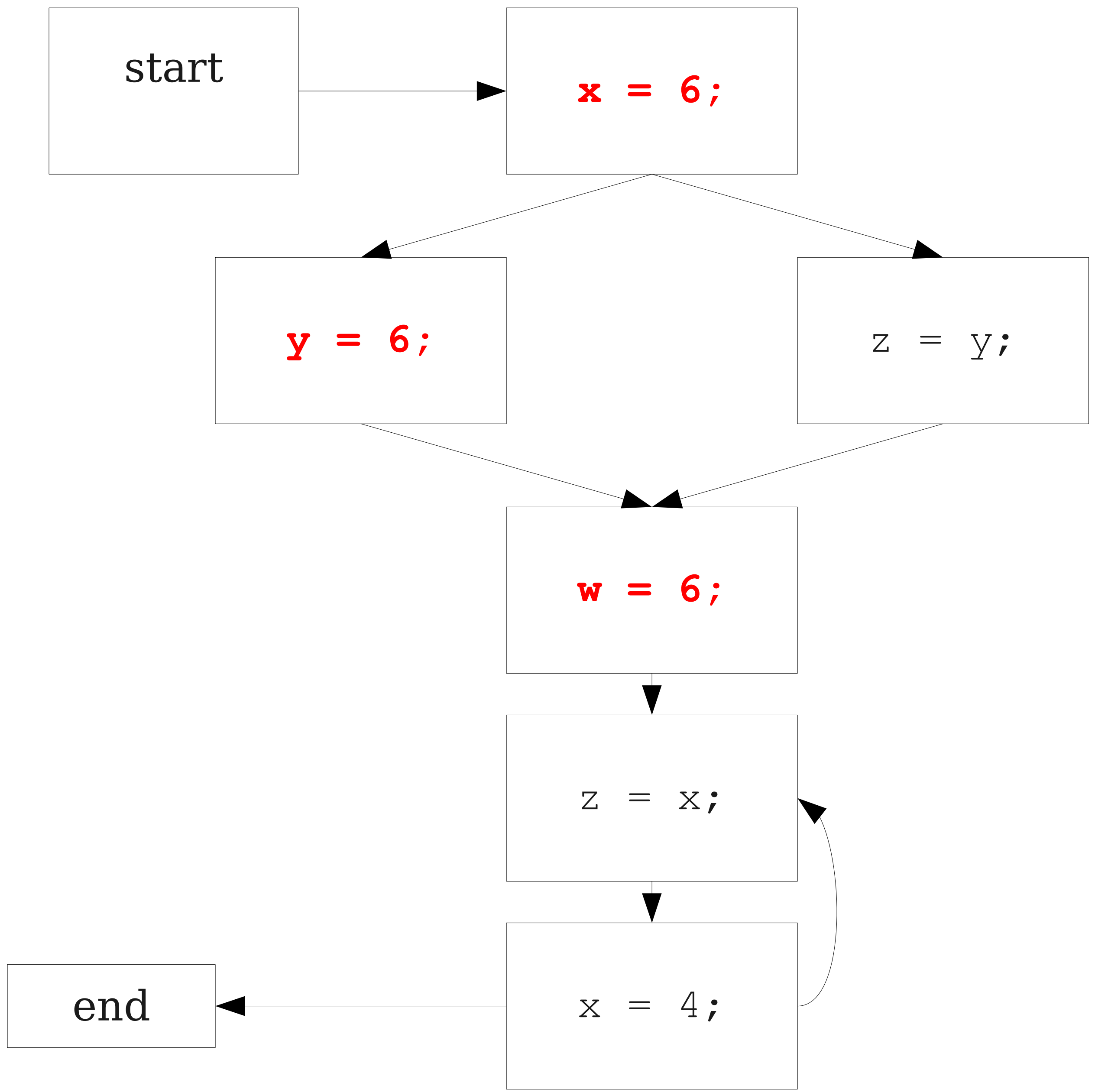
Global Constant Propagation



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

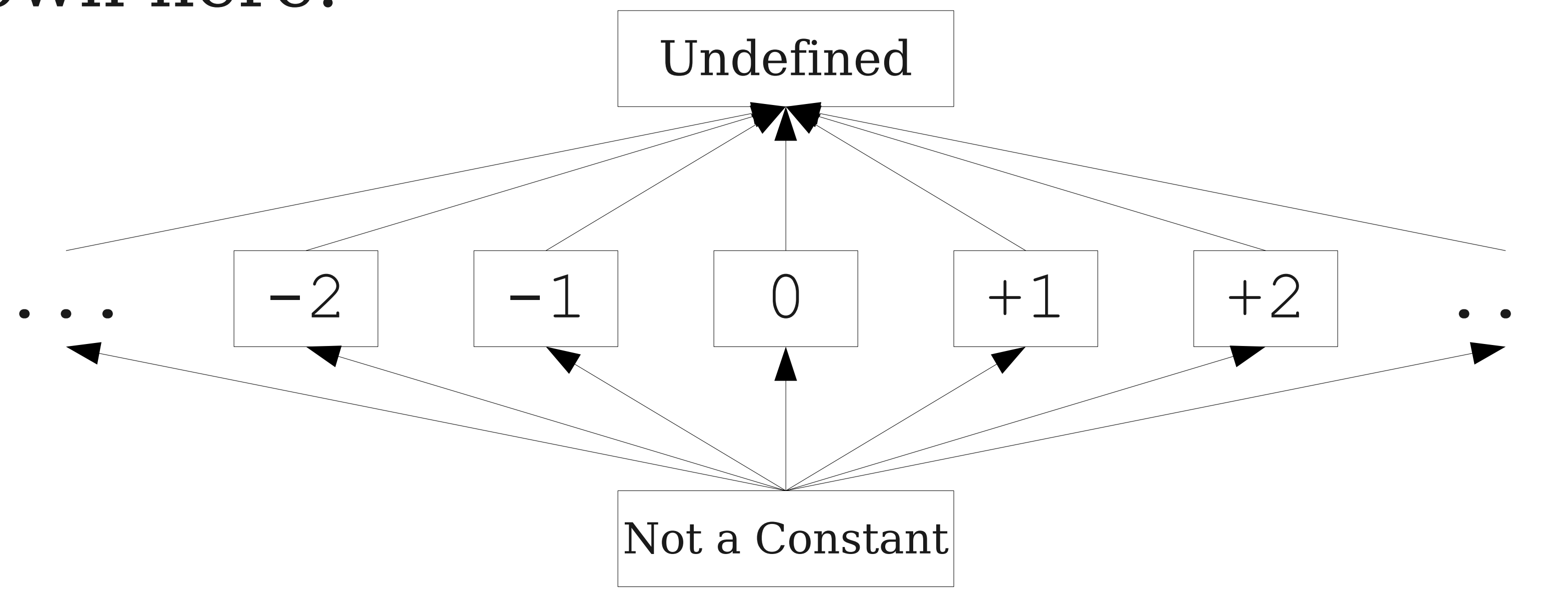
- For now, consider just some single variable **x**.
- At each point in the program, we know one of three things about the value of **x**:
  - **x** is definitely not a constant, since it's been assigned two values or assigned a value that we know isn't a constant.
  - **x** is definitely a constant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

- 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.)
- The meet 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 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.)

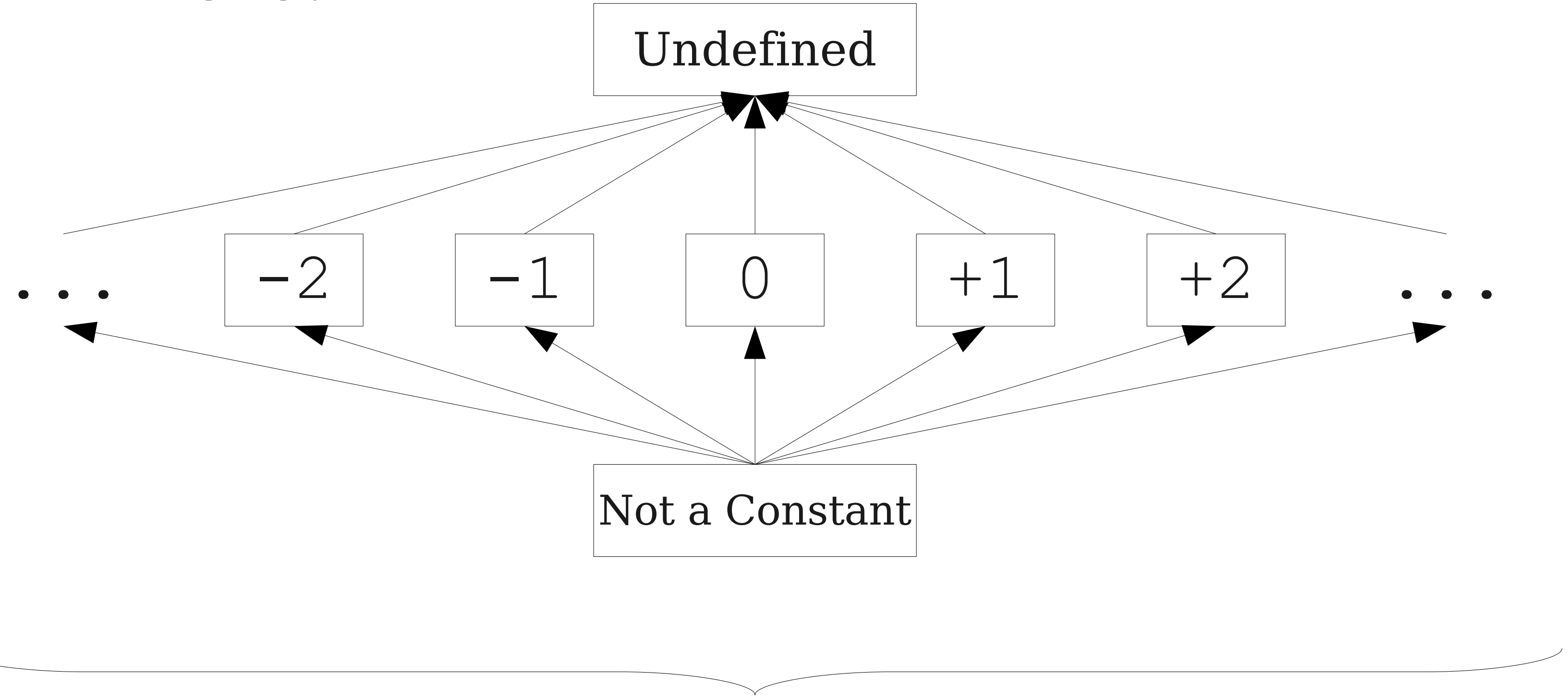
A Semilattice for Constant Propagation

- One possible semilattice for this analysis is shown here:



A Semilattice for Constant Propagation

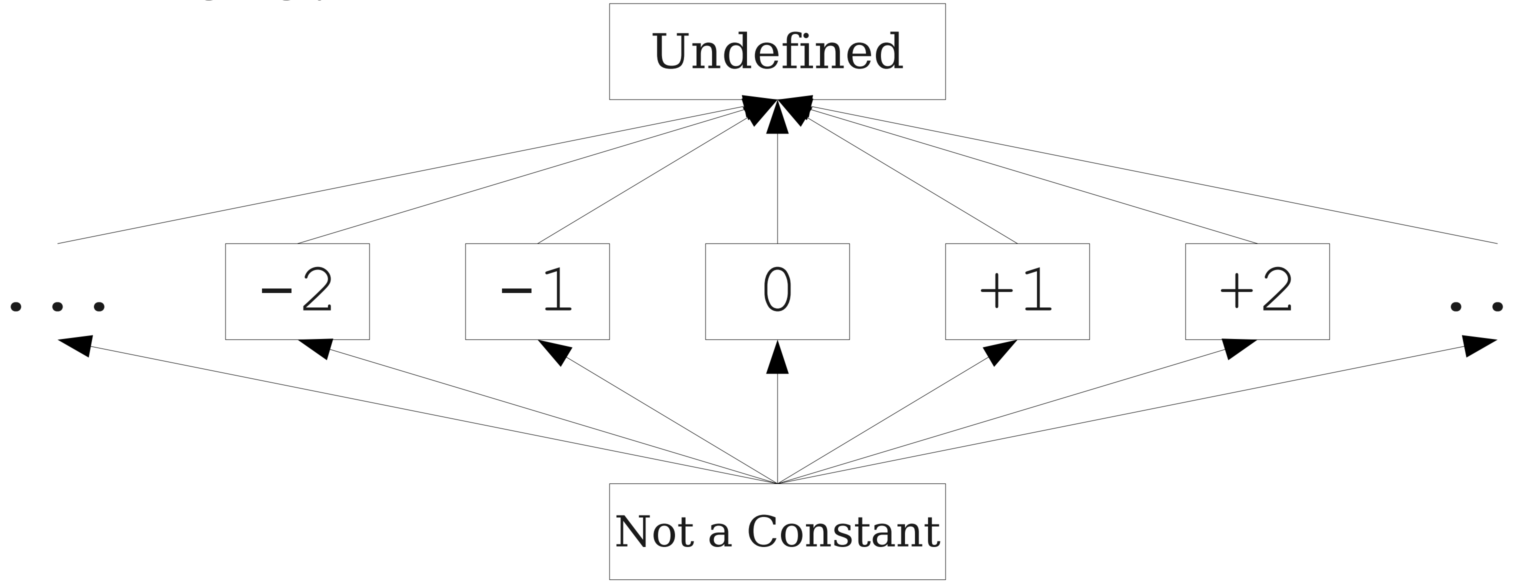
- One possible semilattice for this analysis is shown here:



This lattice is infinitely wide!

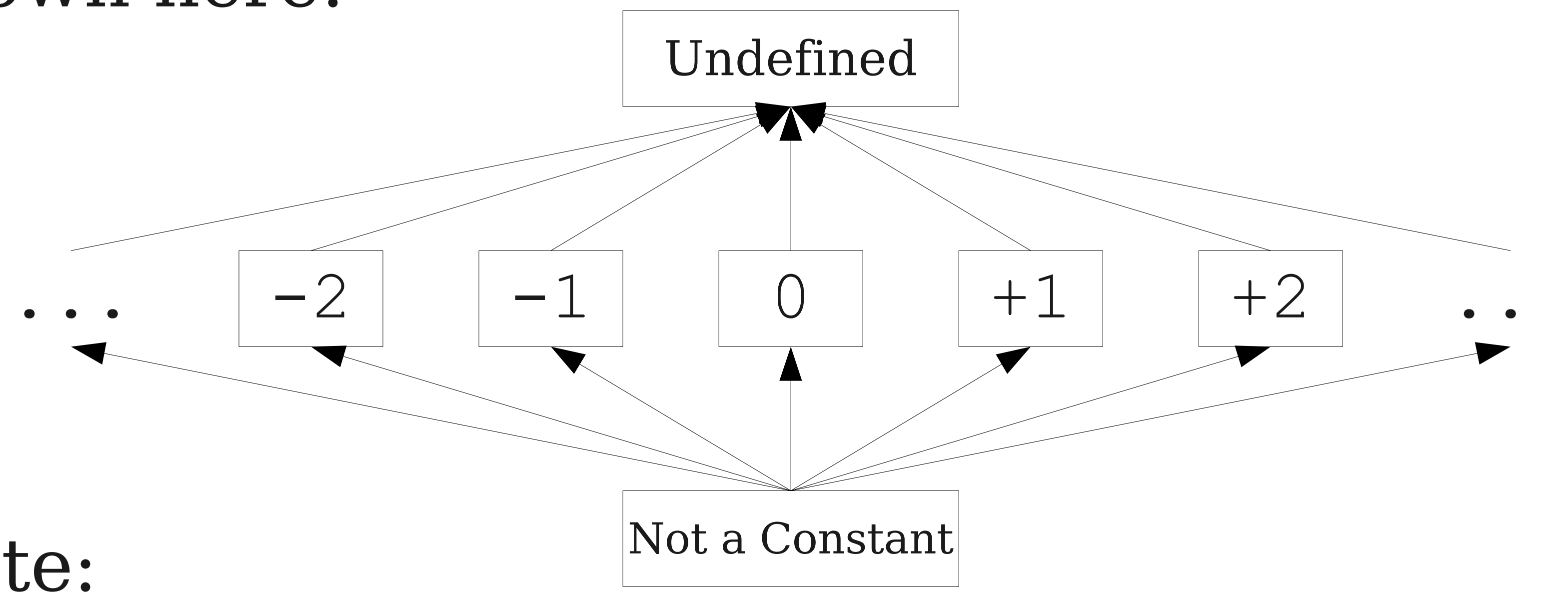
A Semilattice for Constant Propagation

- One possible semilattice for this analysis is shown here:



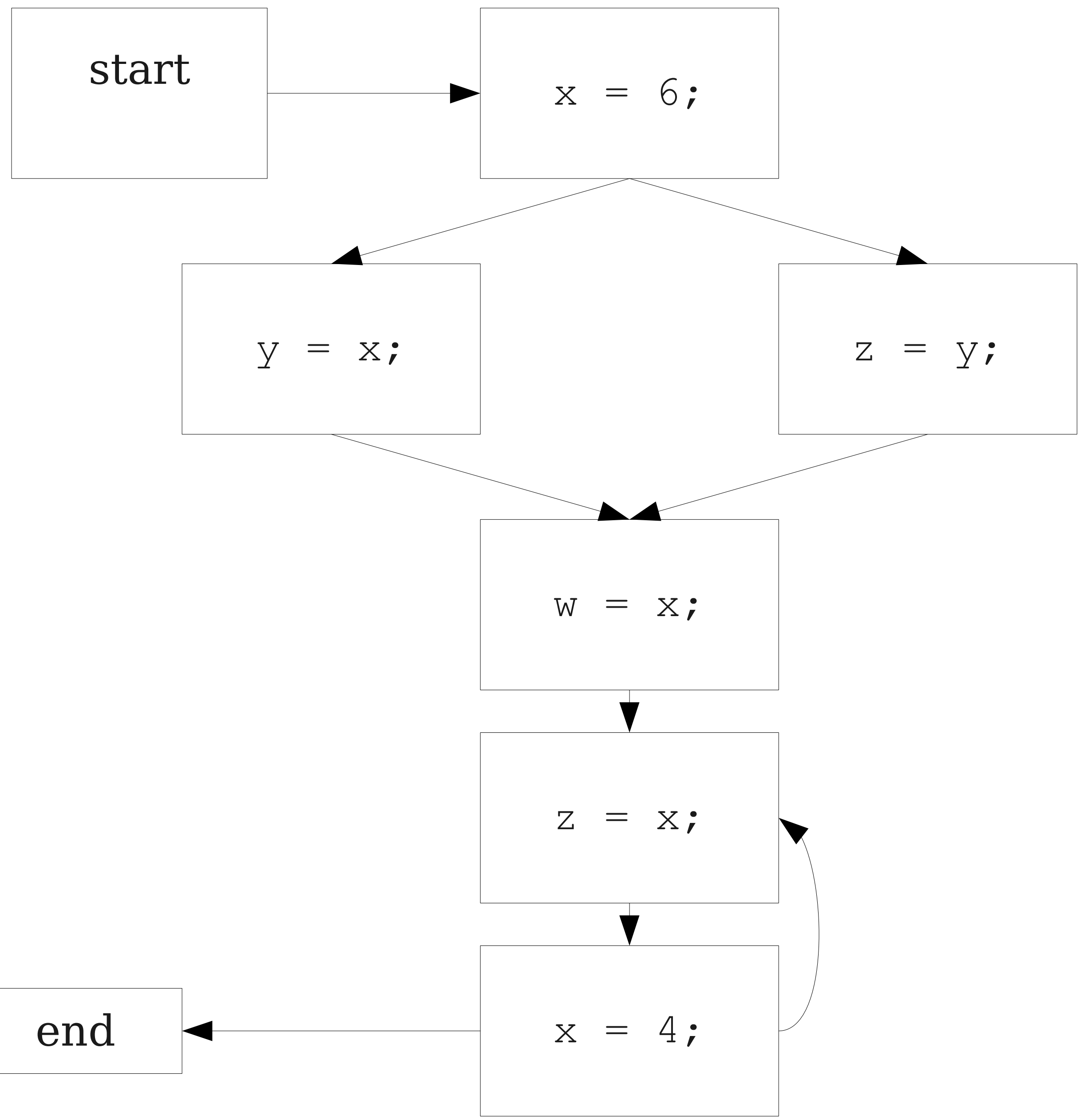
A Semilattice for Constant Propagation

- One possible semilattice for this analysis is shown here:

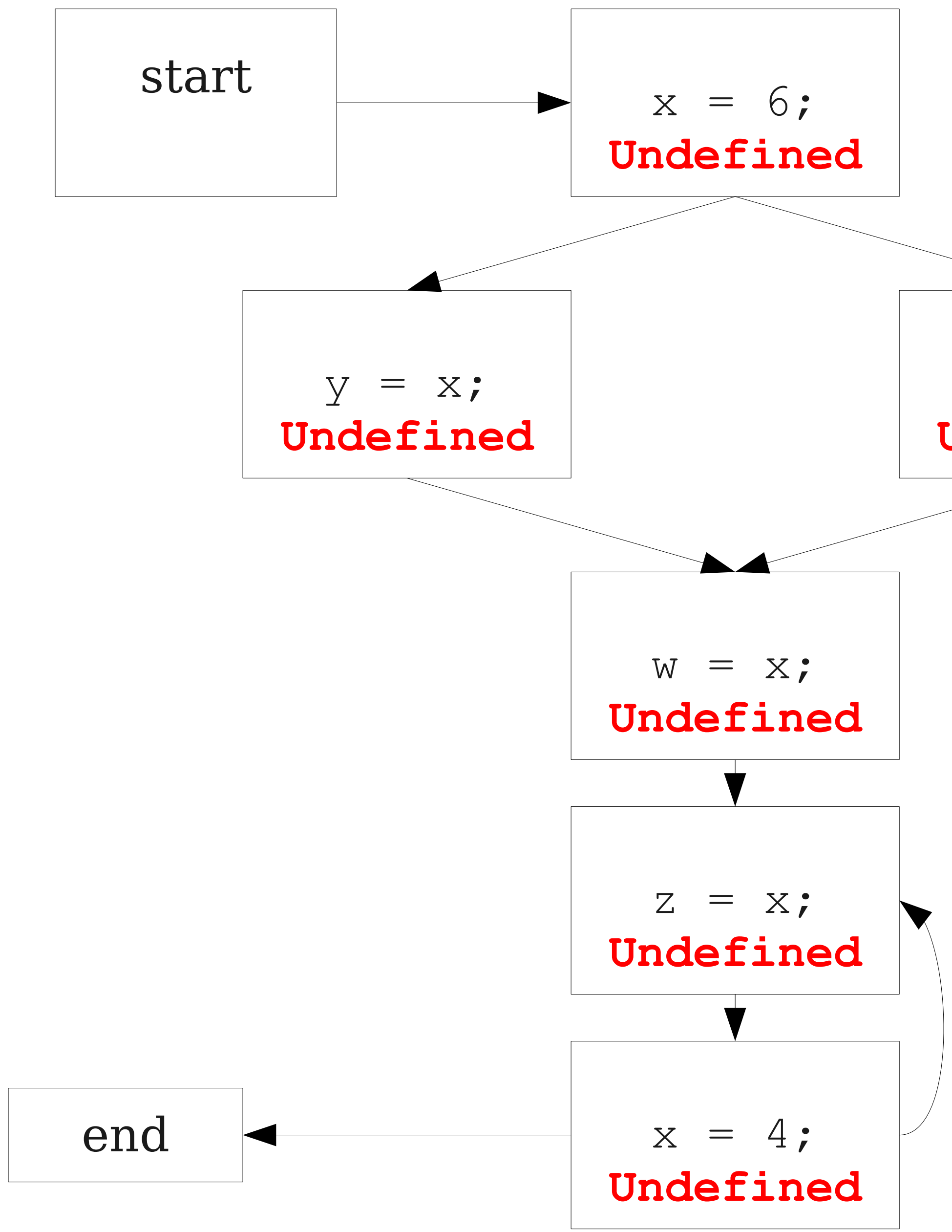


- Note:
  - The meet of any two different constants is **Not a Constant**.
  - The meet of **Undefined** and any value is that value.
  - The meet of **Not a Constant** and any value is **Not a Constant**.

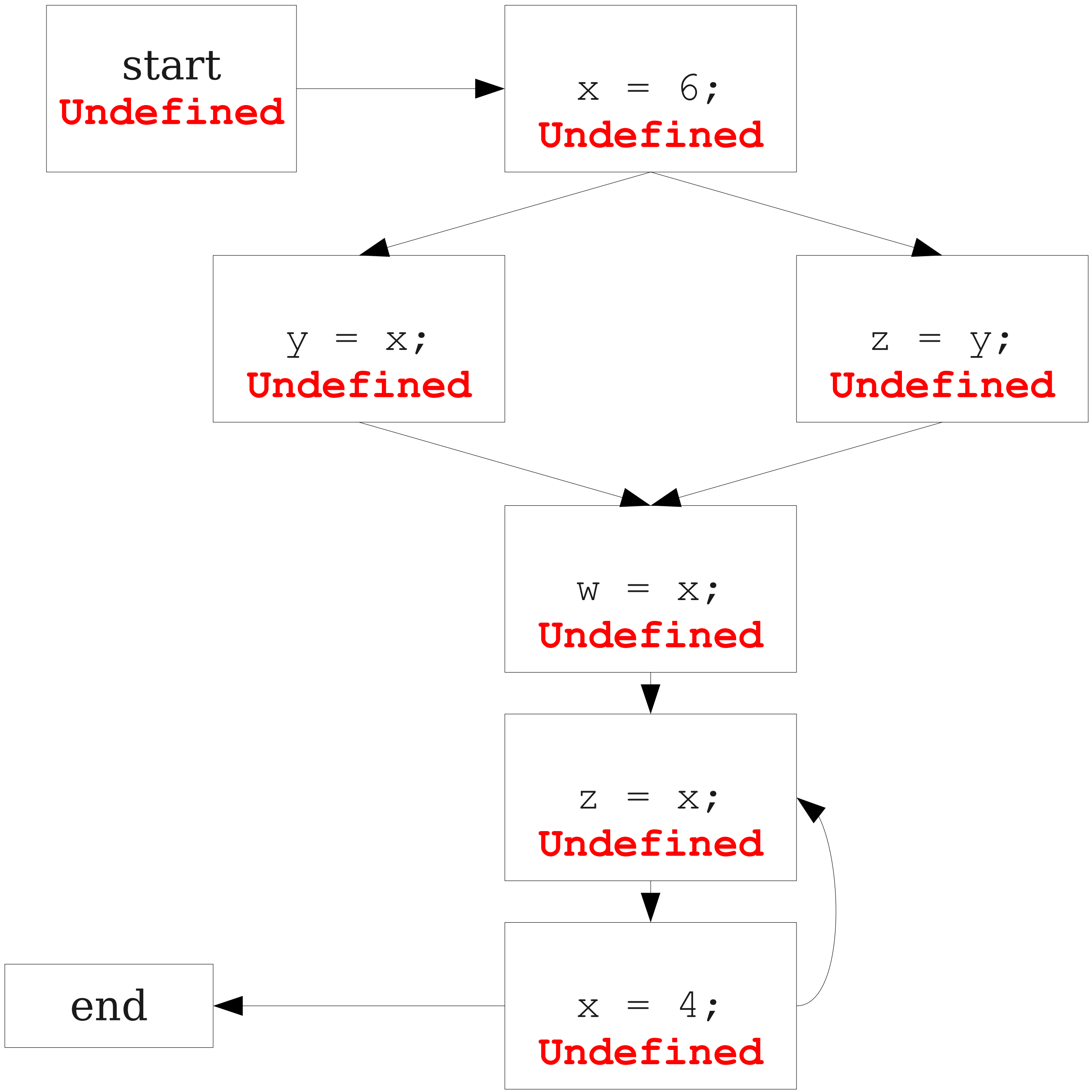
Global Constant Propagation



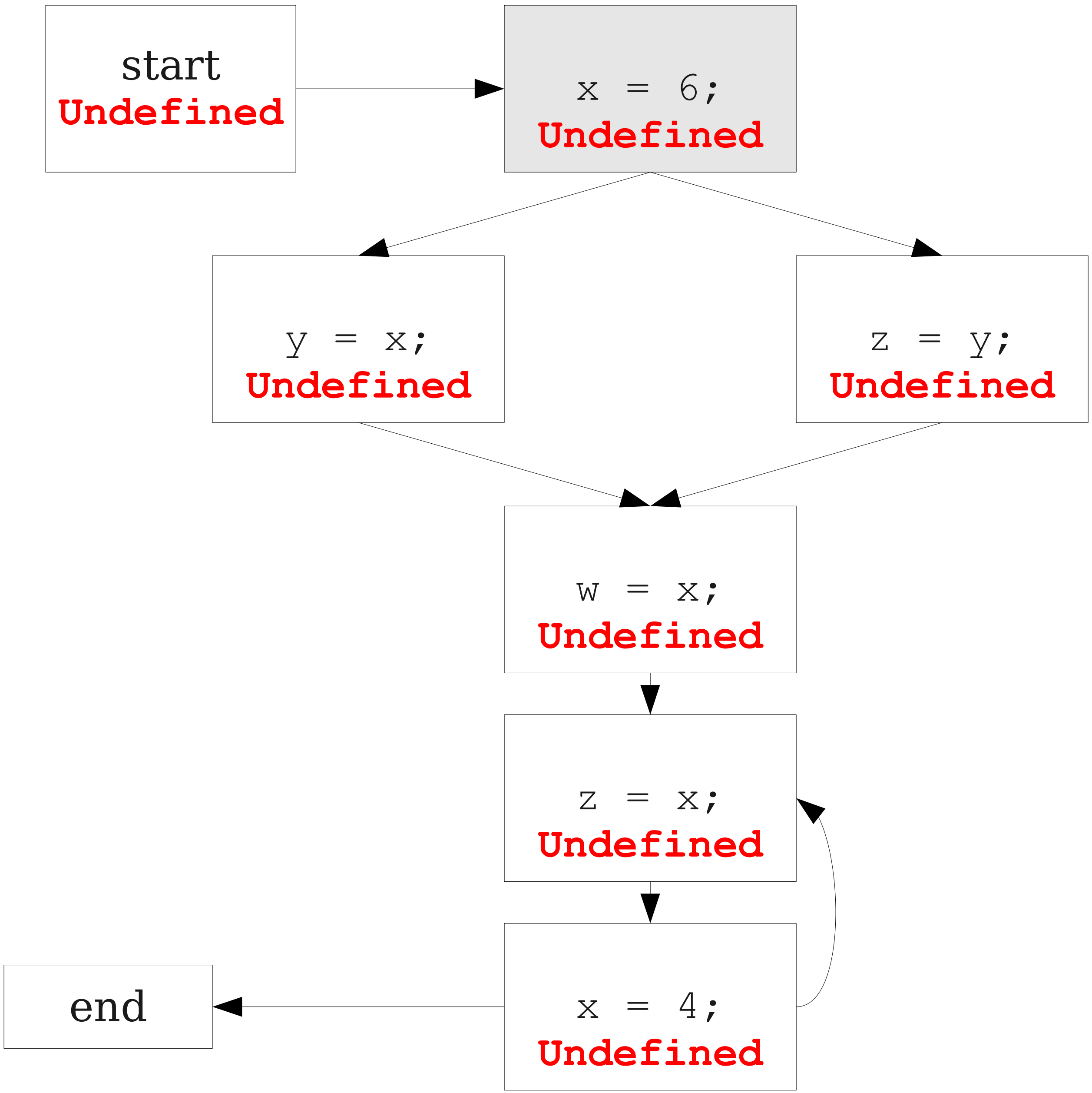
Global Constant Propagation



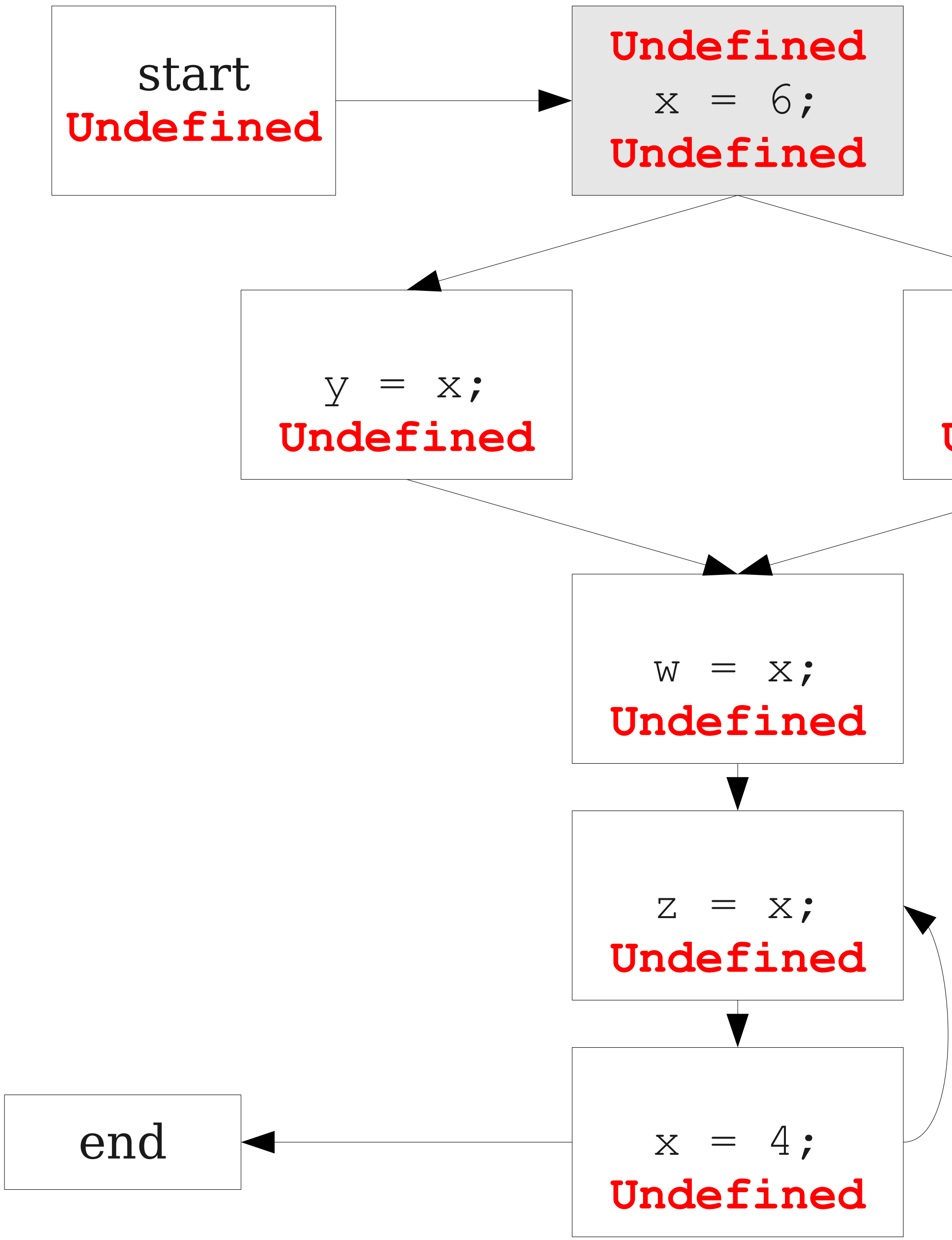
Global Constant Propagation



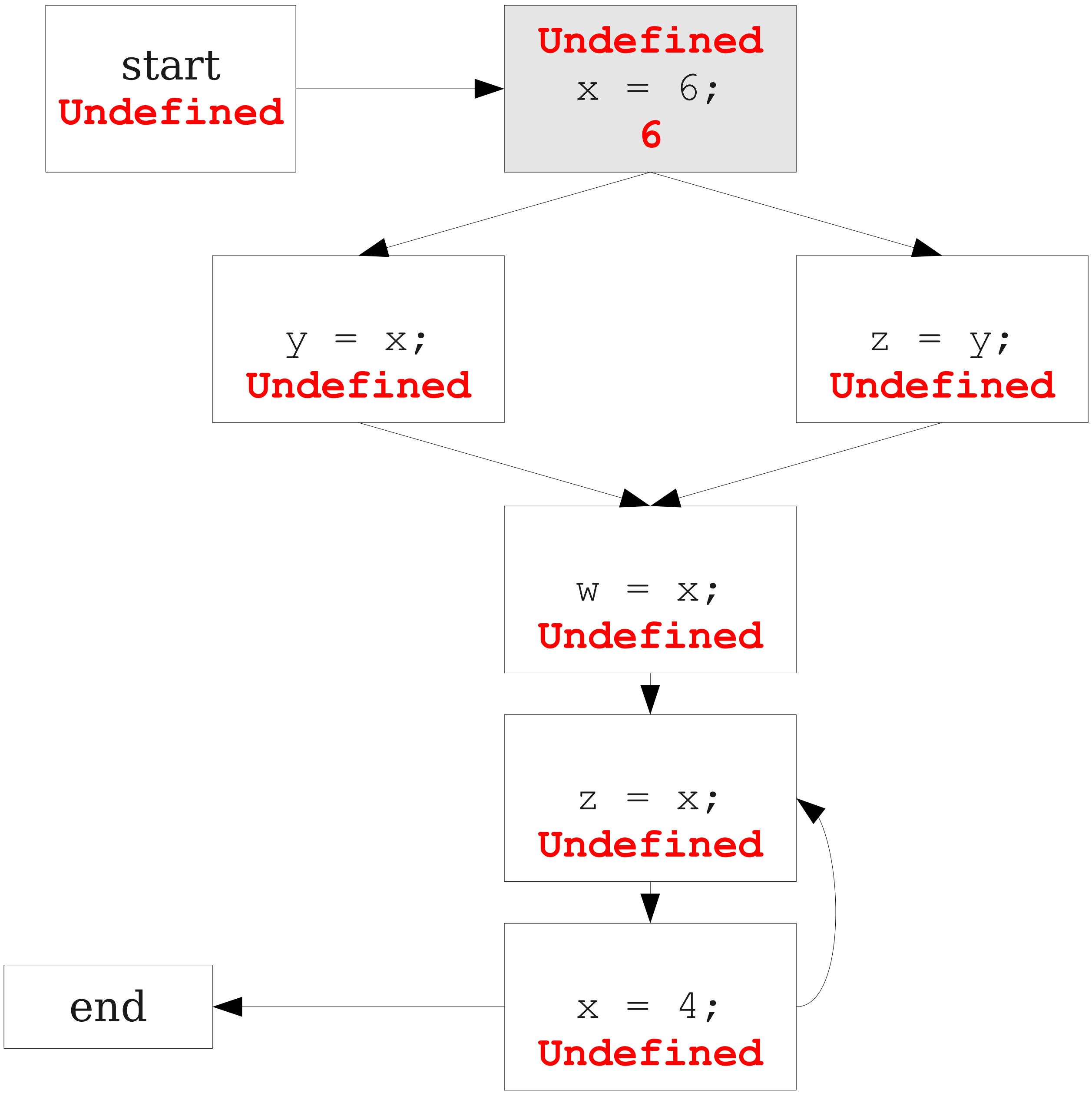
Global Constant Propagation



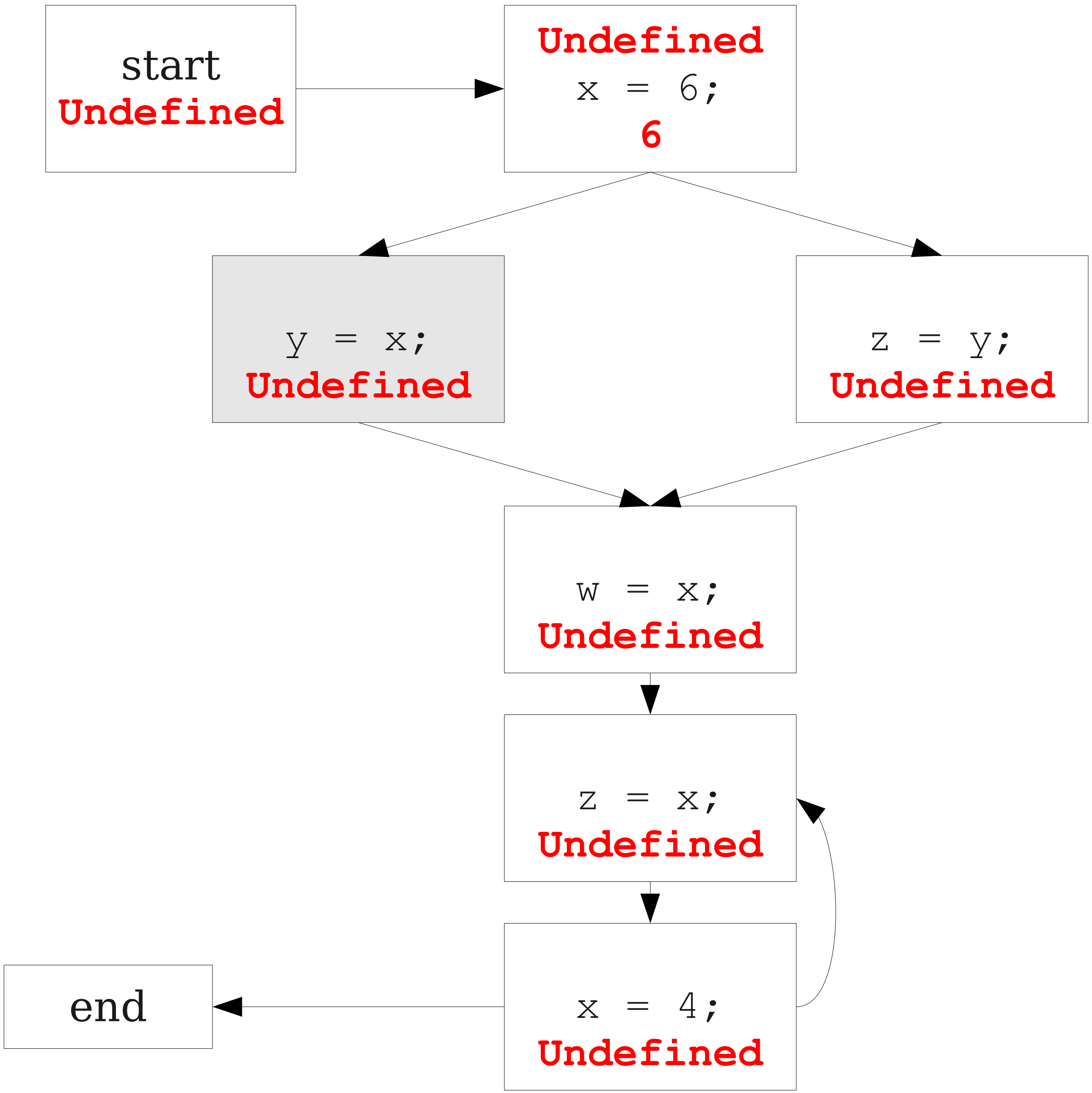
Global Constant Propagation



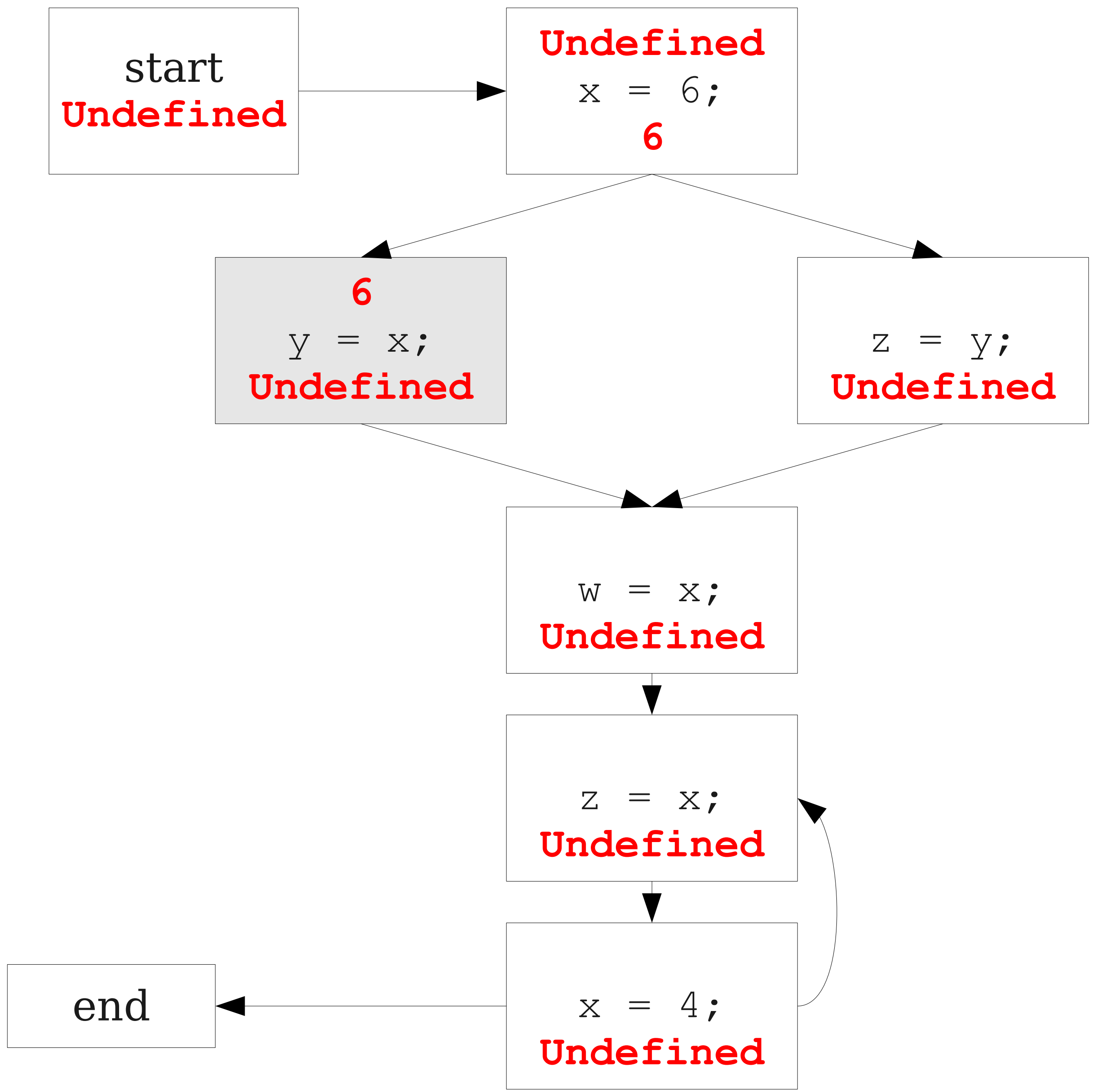
Global Constant Propagation



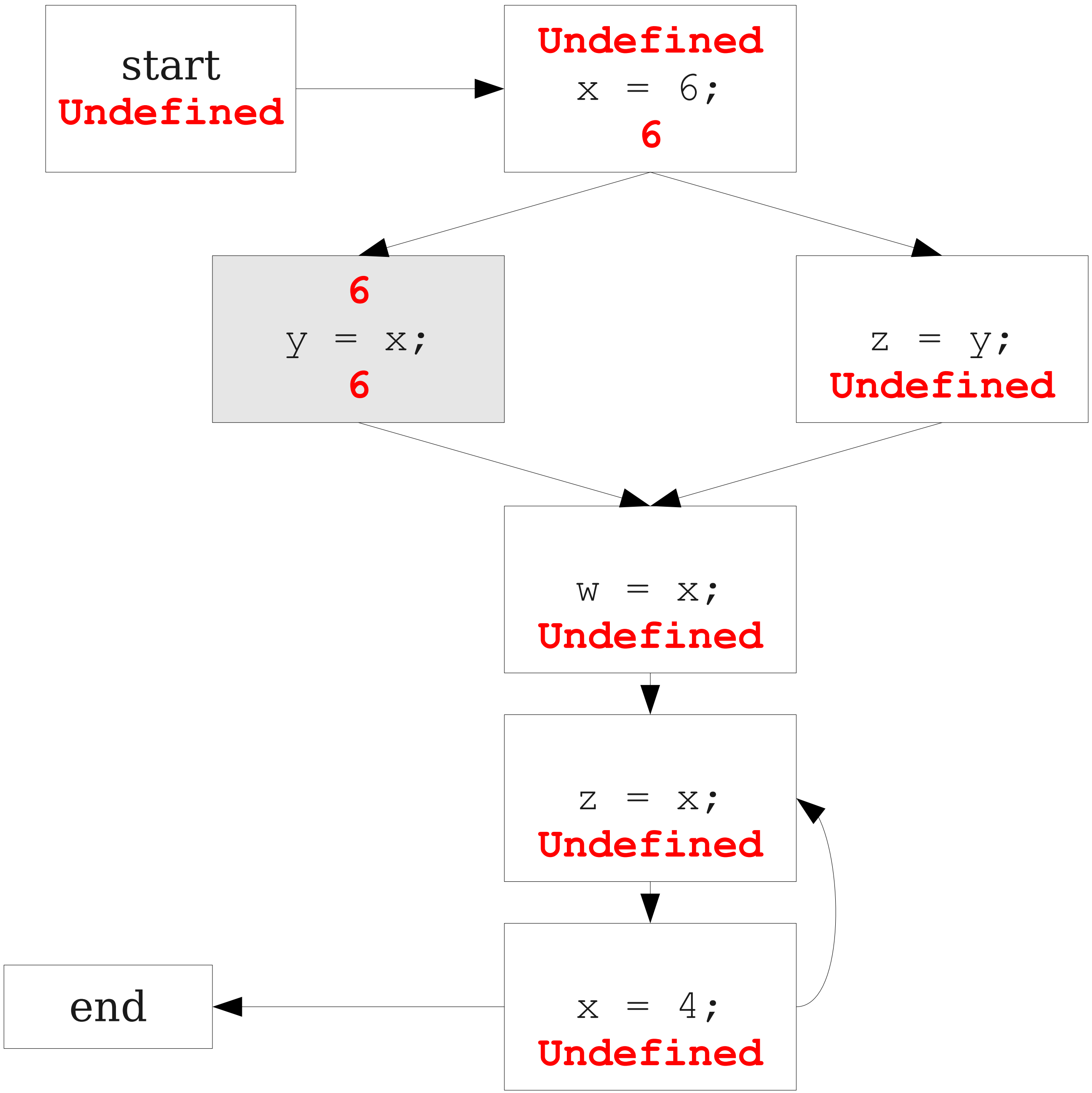
Global Constant Propagation



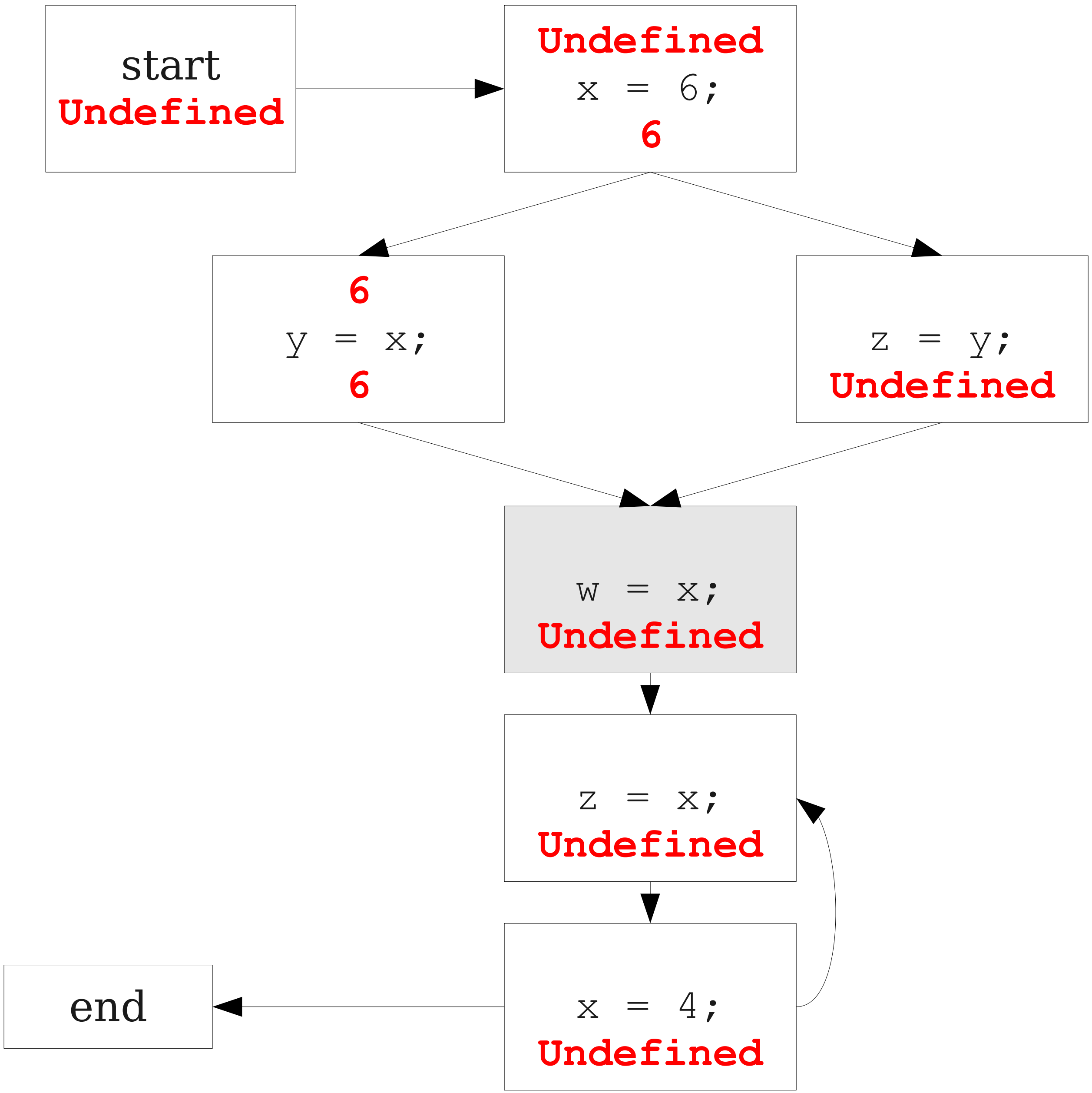
Global Constant Propagation



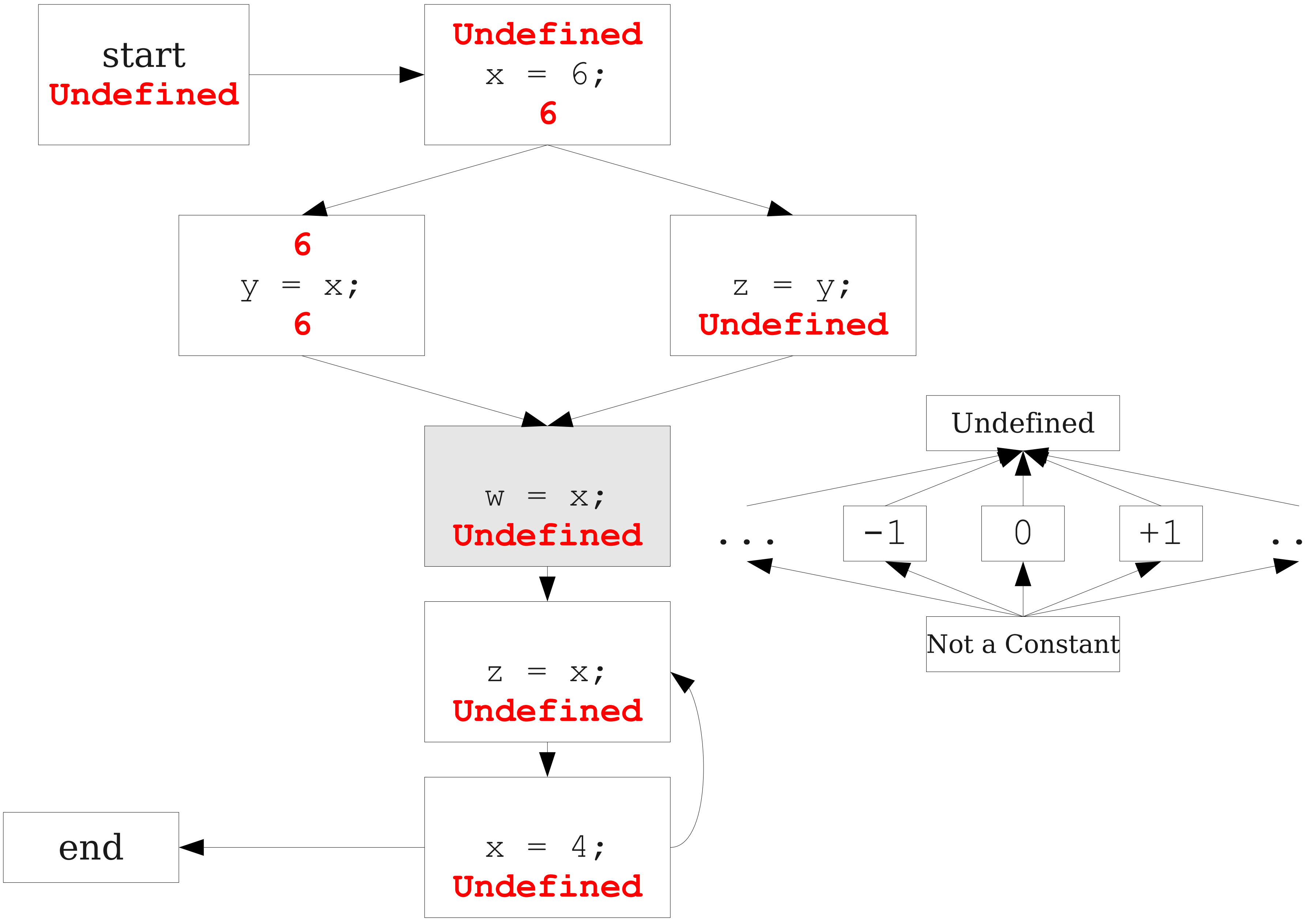
Global Constant Propagation



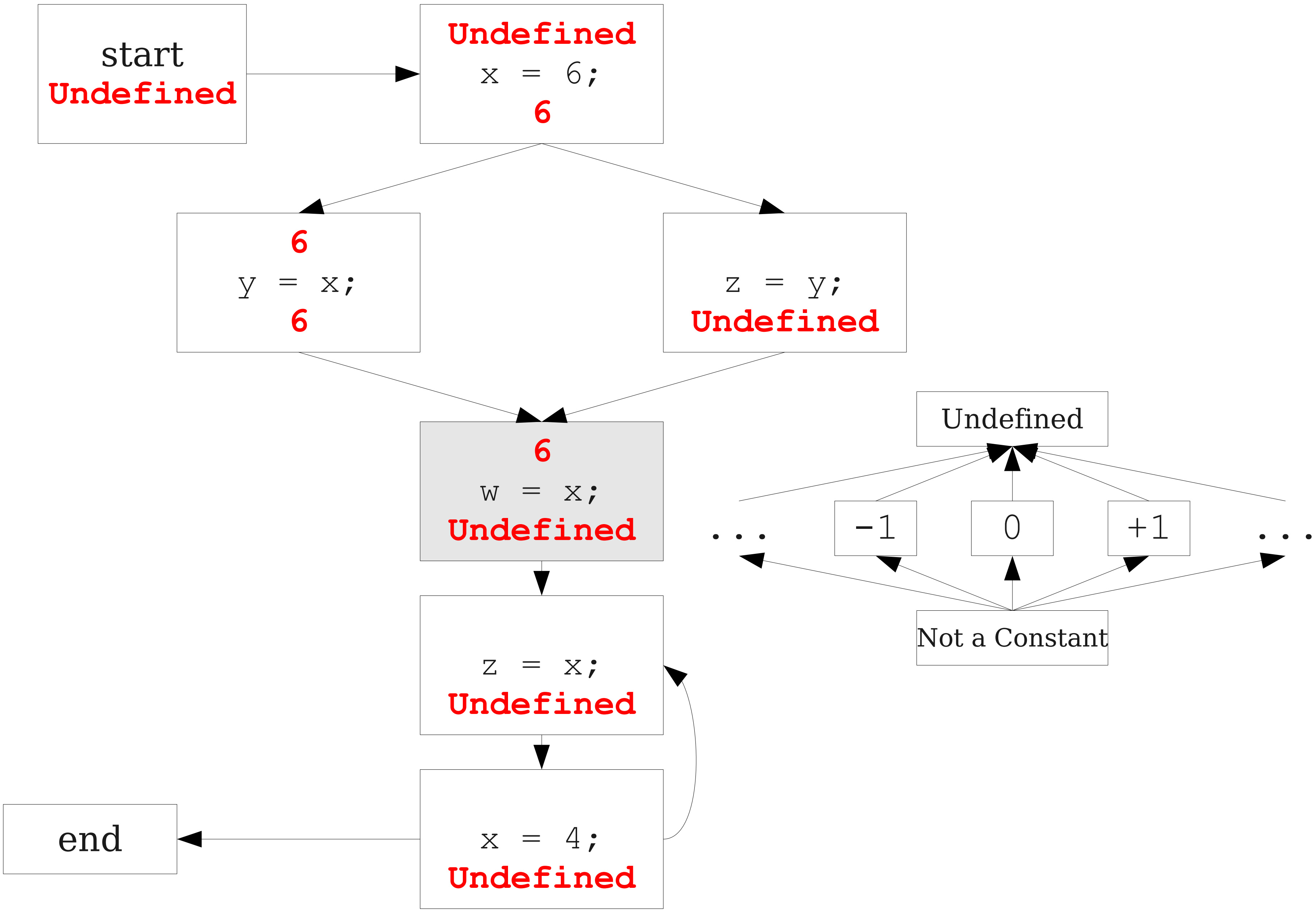
Global Constant Propagation



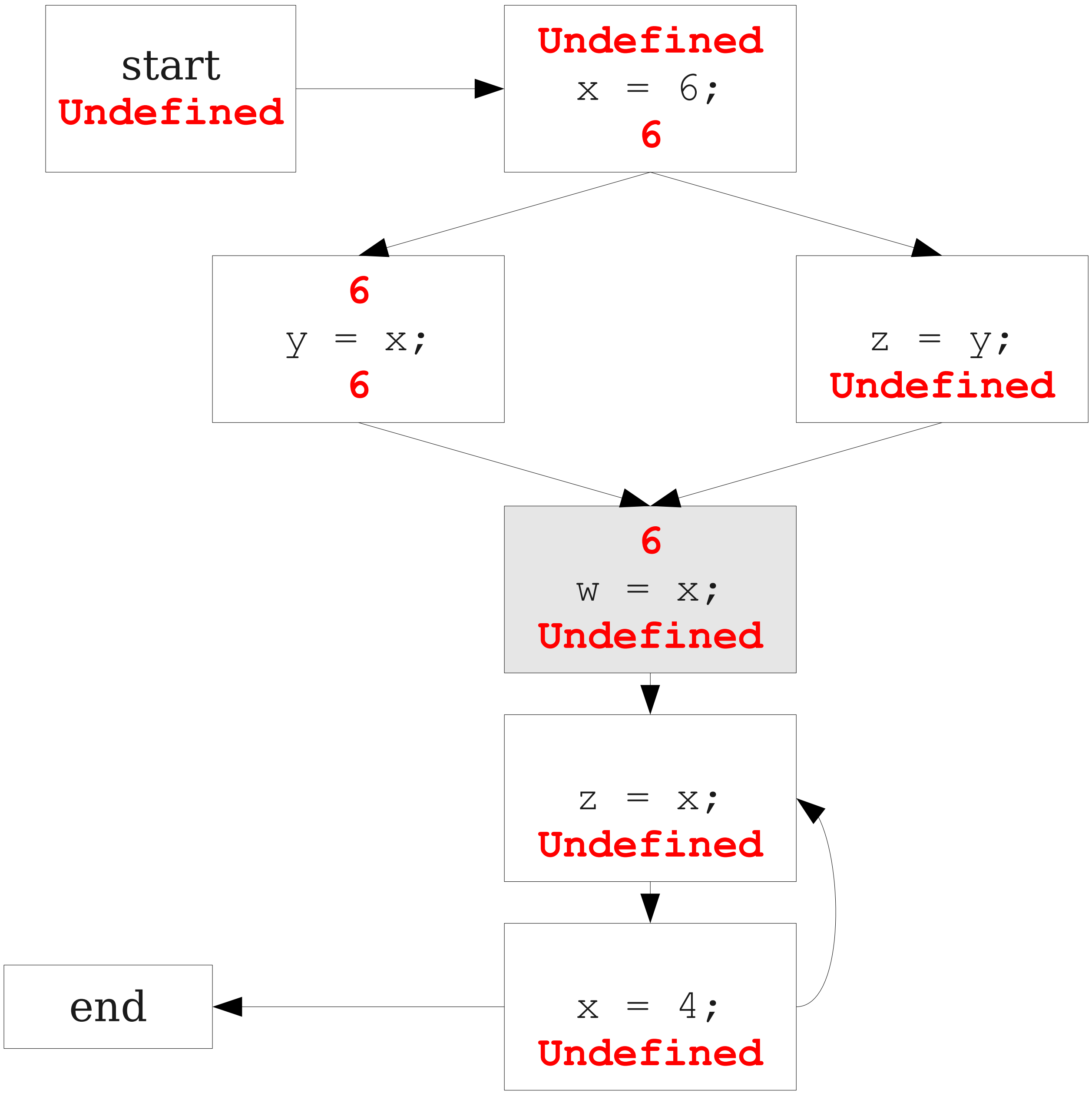
Global Constant Propagation



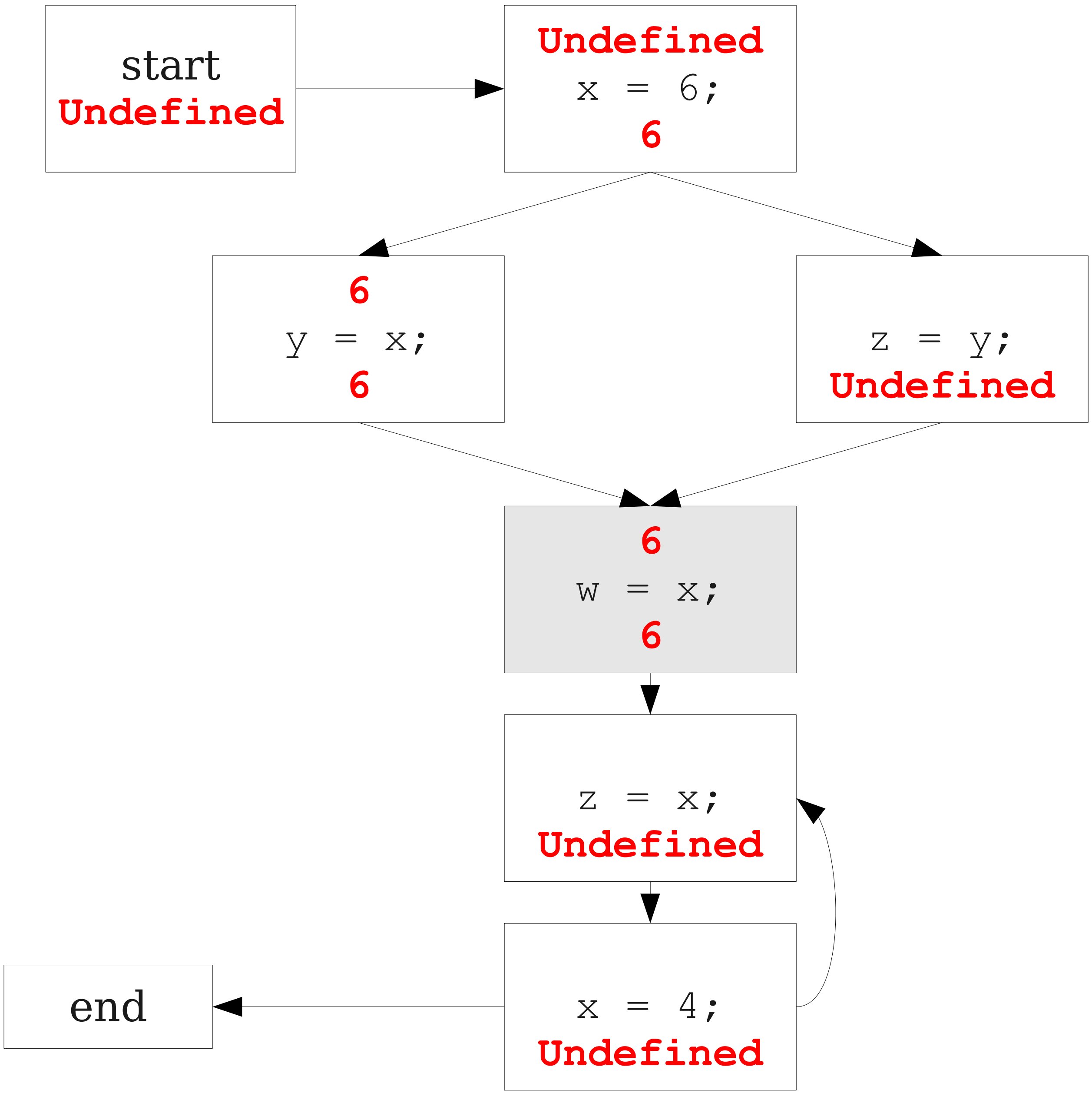
Global Constant Propagation



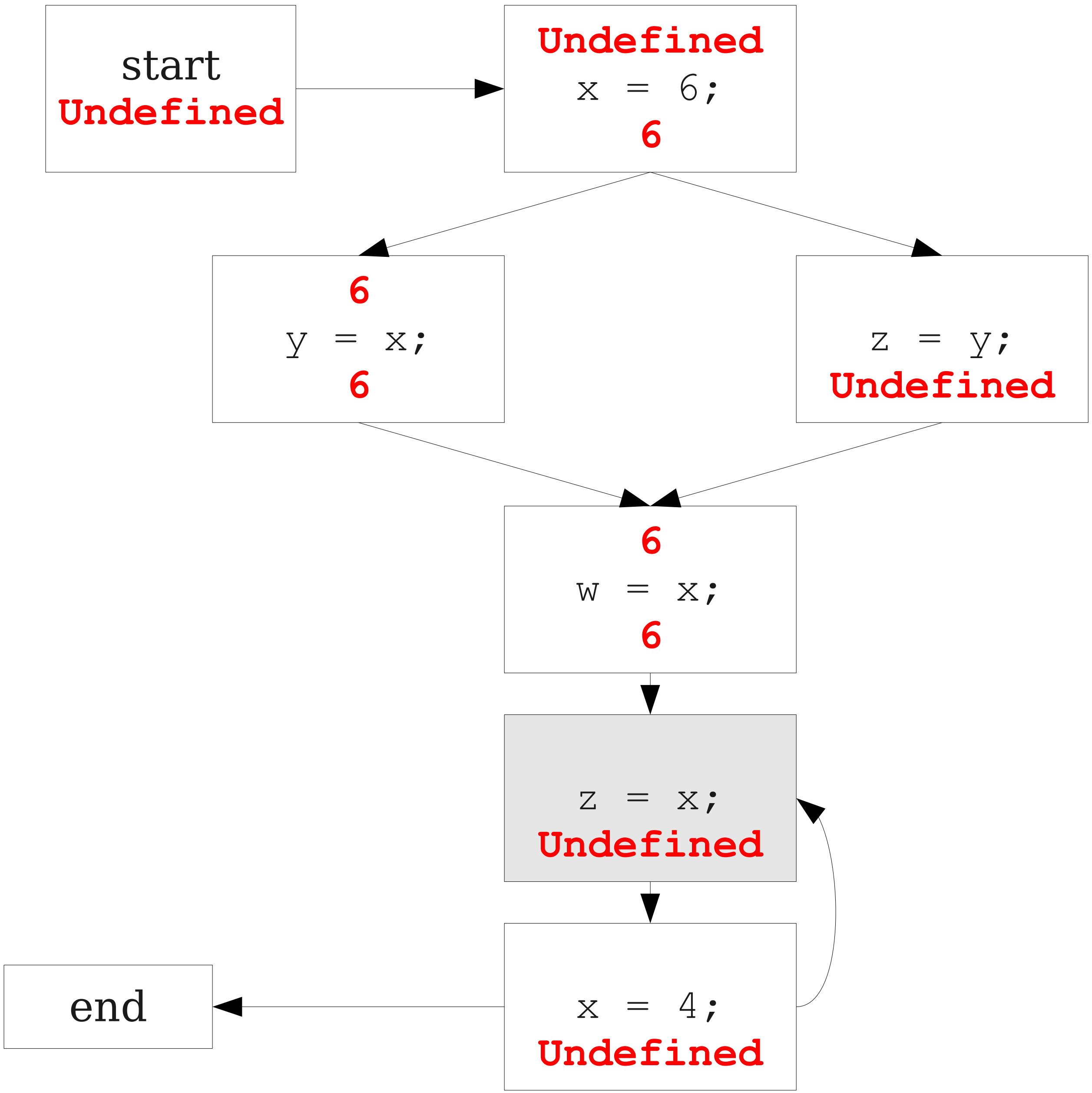
Global Constant Propagation



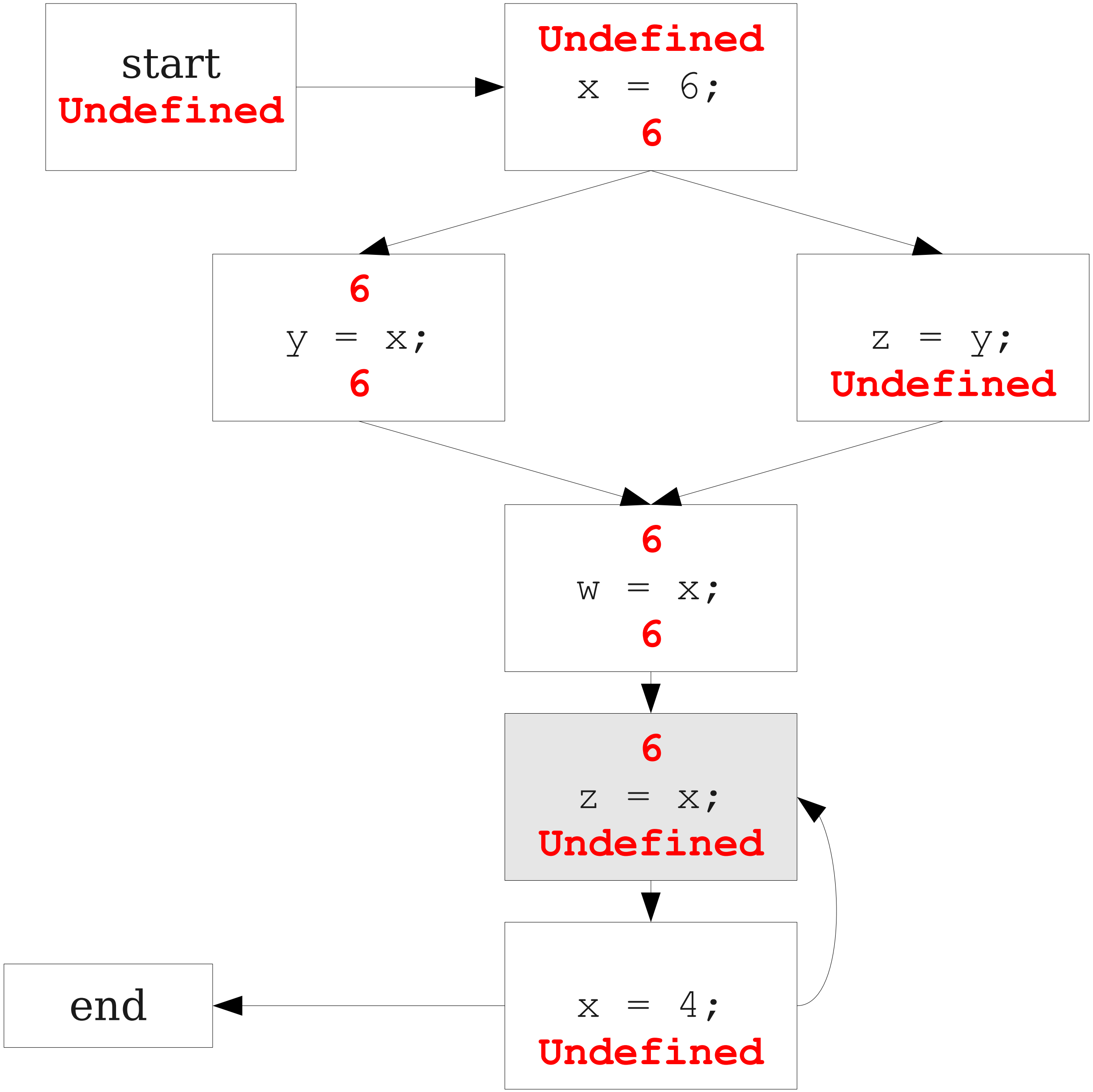
Global Constant Propagation



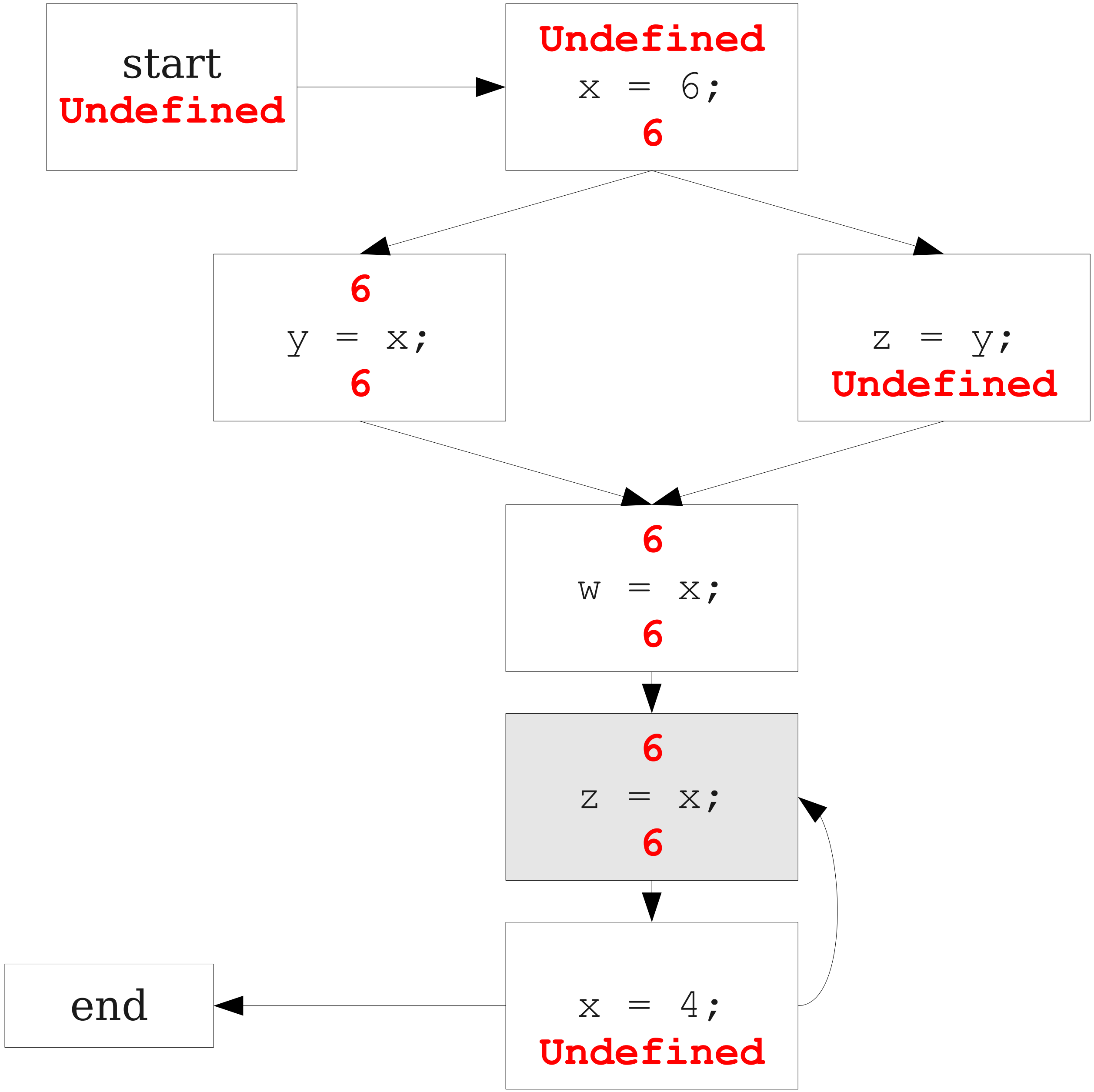
Global Constant Propagation



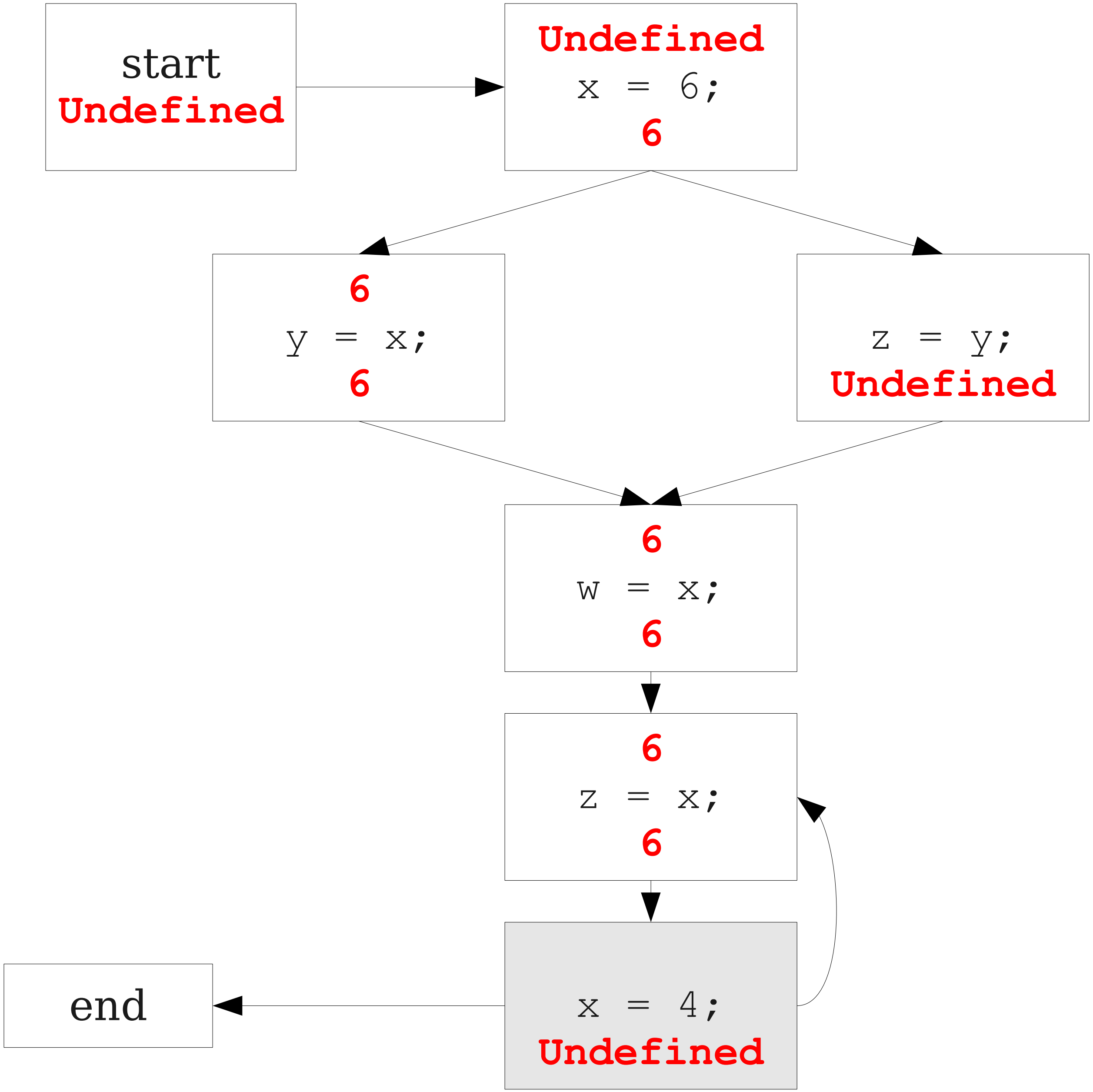
Global Constant Propagation



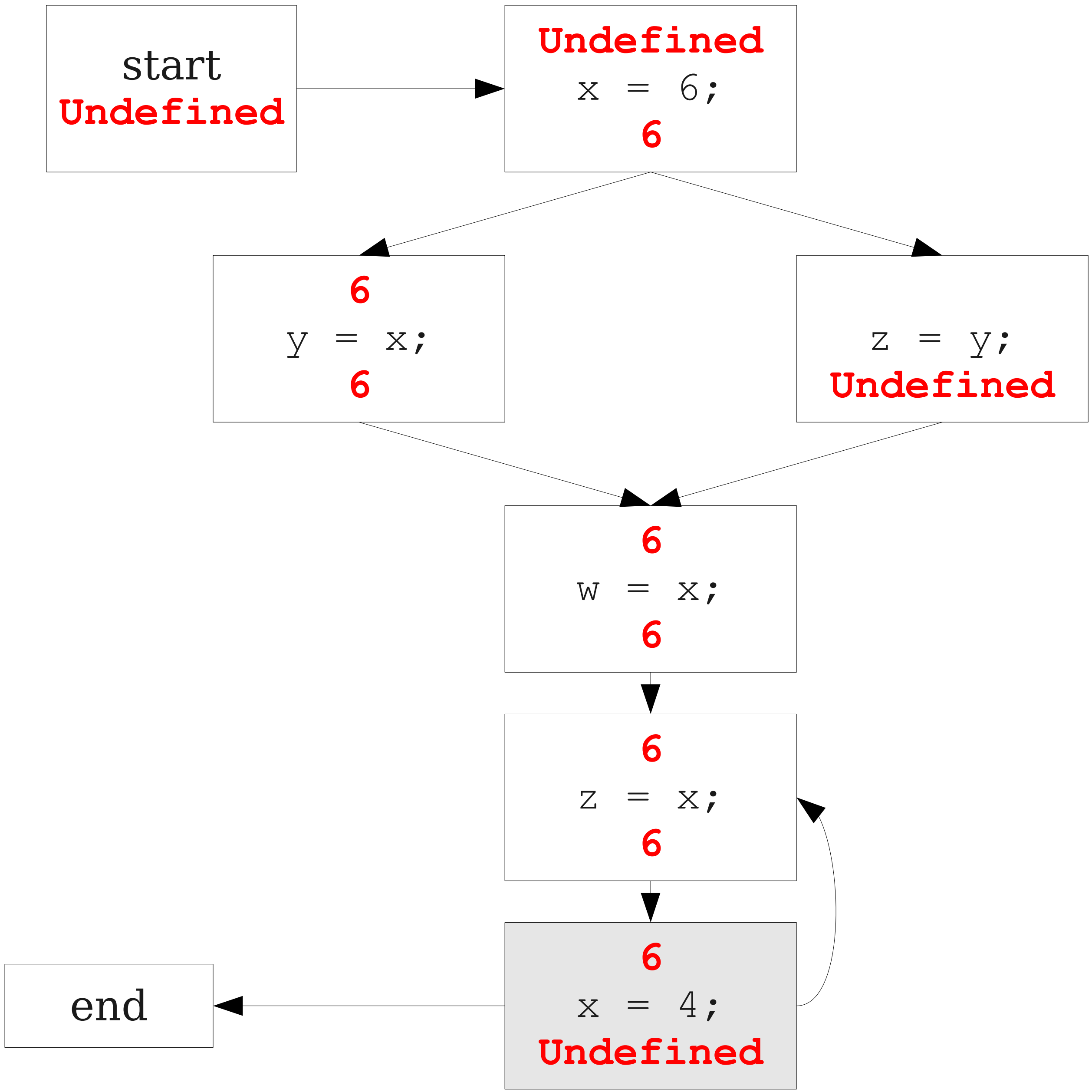
Global Constant Propagation



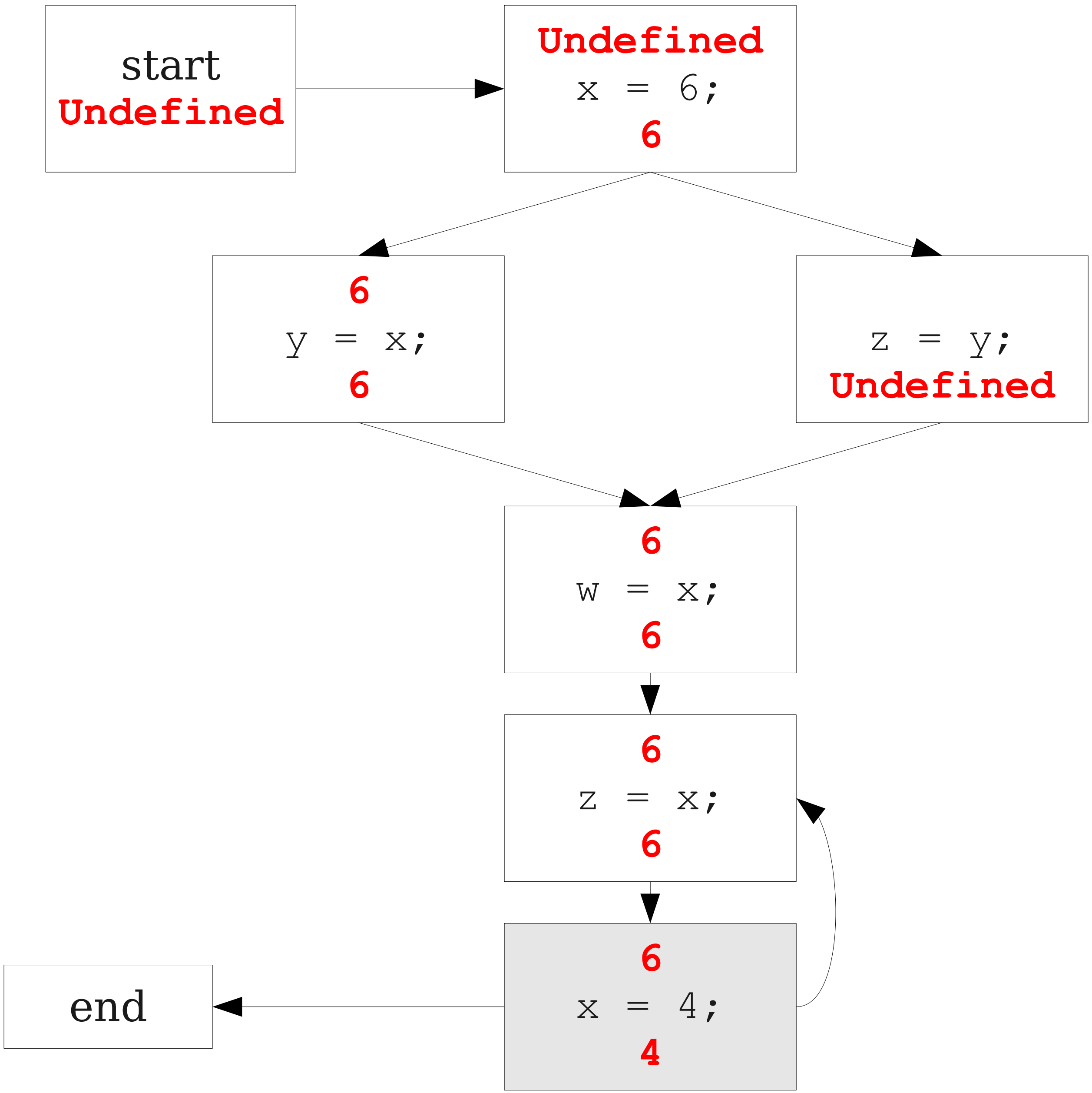
Global Constant Propagation



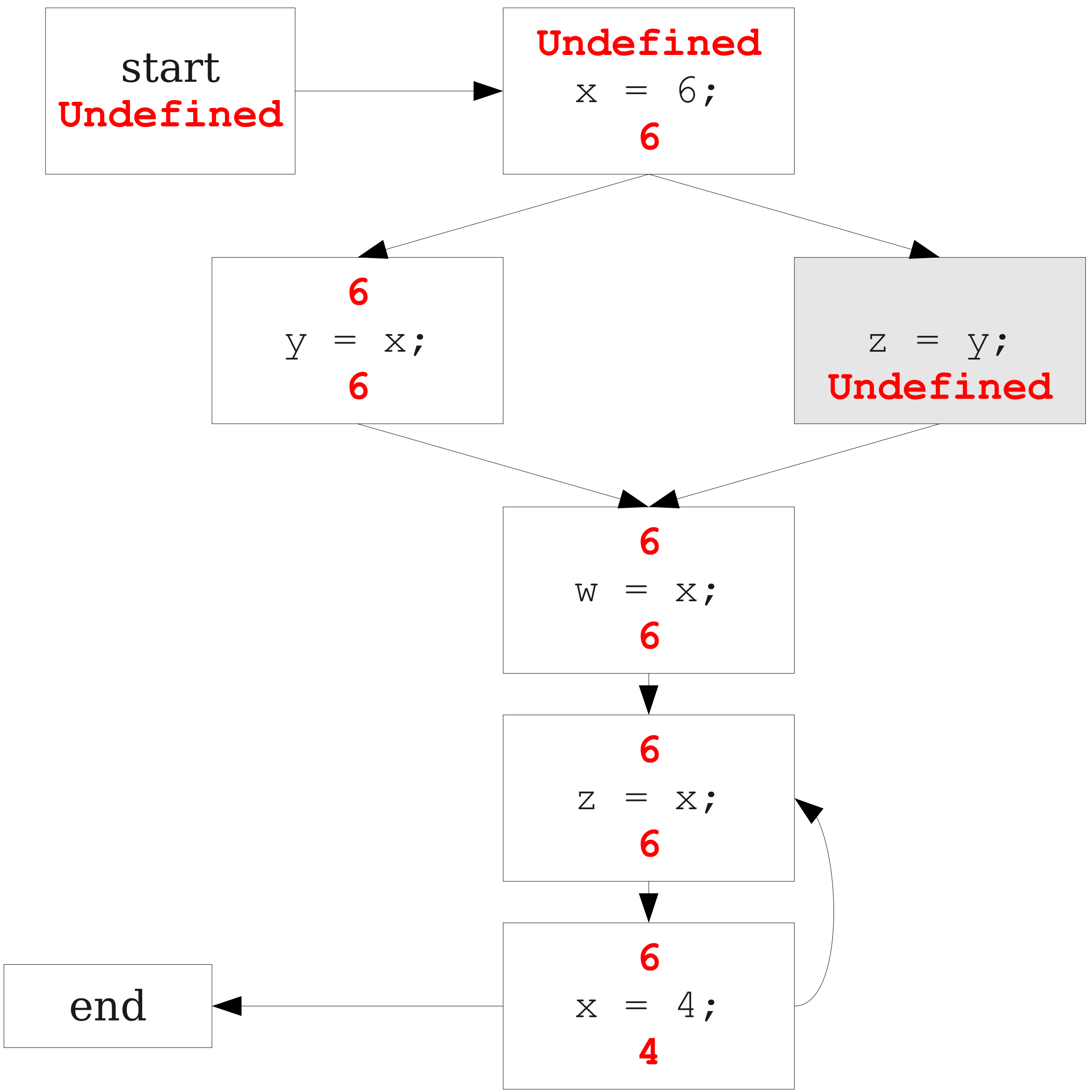
Global Constant Propagation



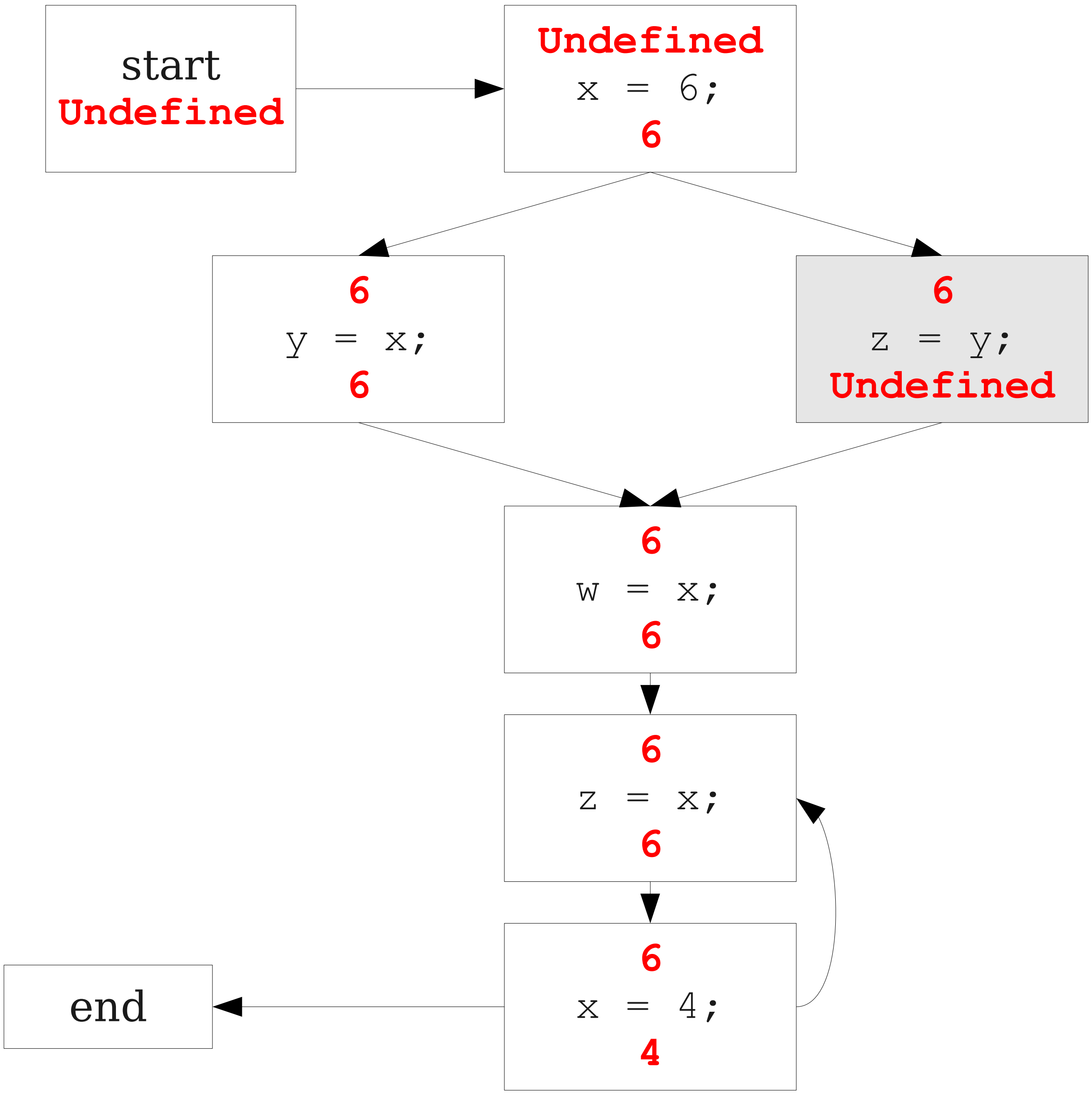
Global Constant Propagation



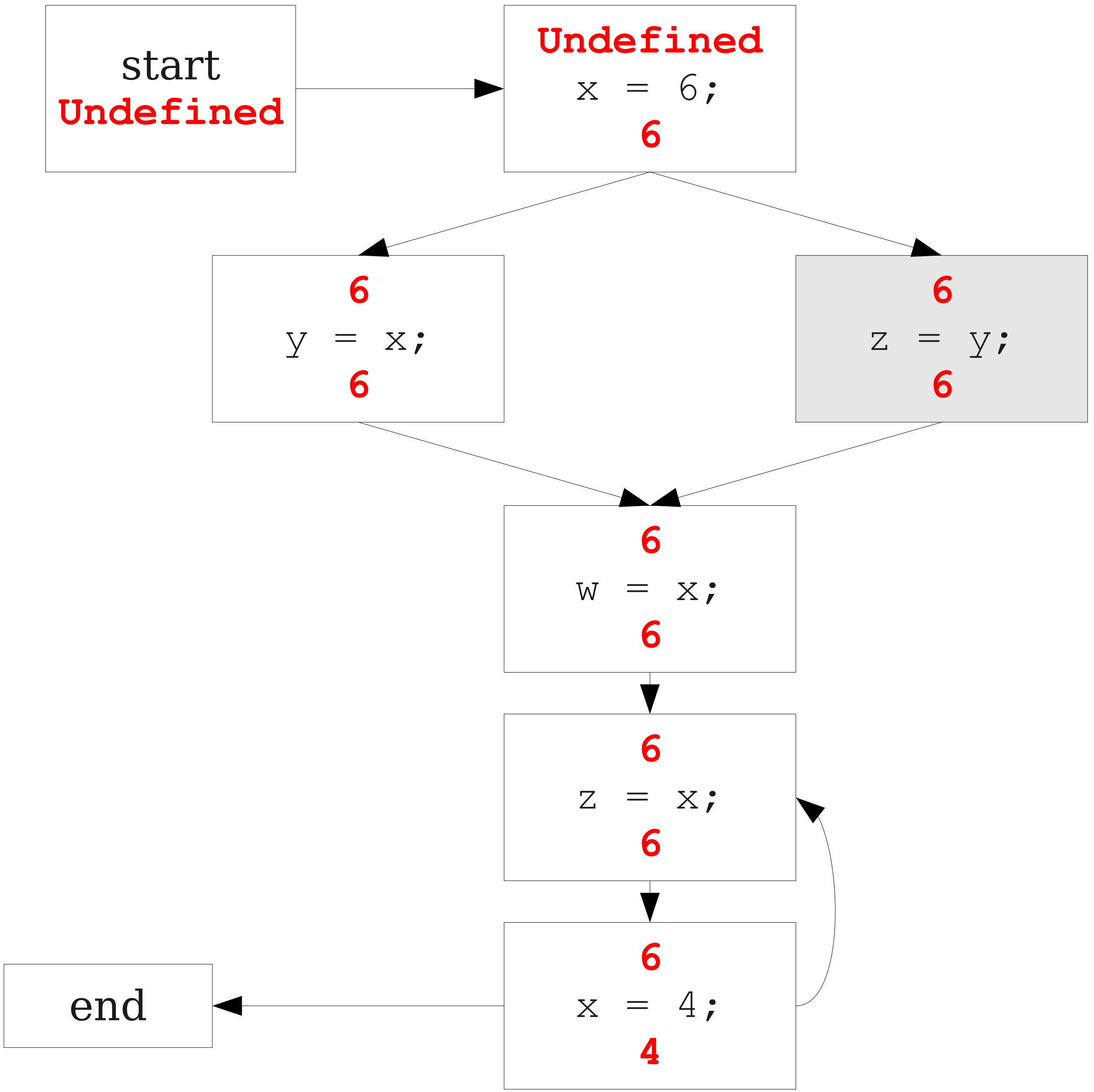
Global Constant Propagation



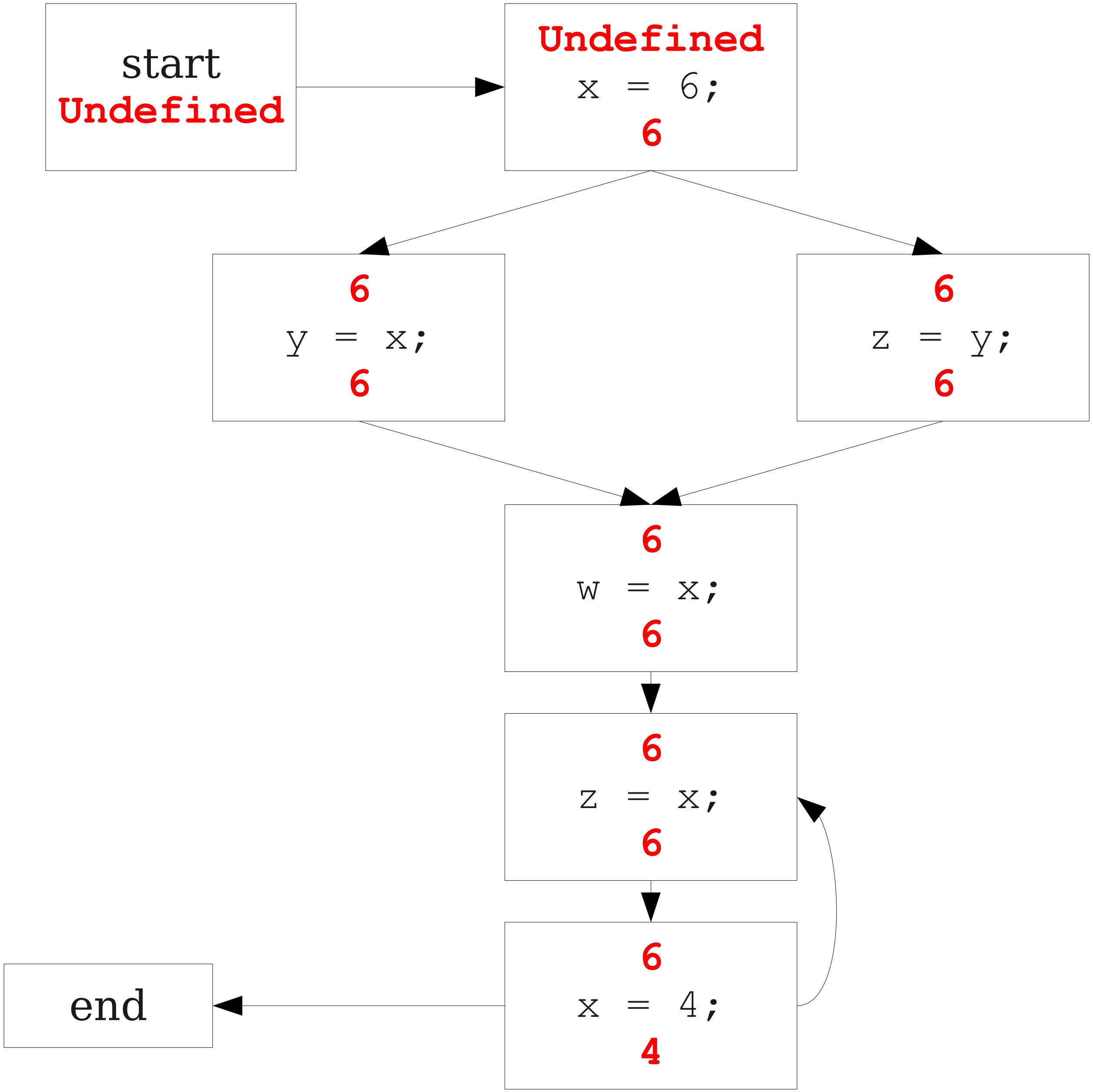
Global Constant Propagation



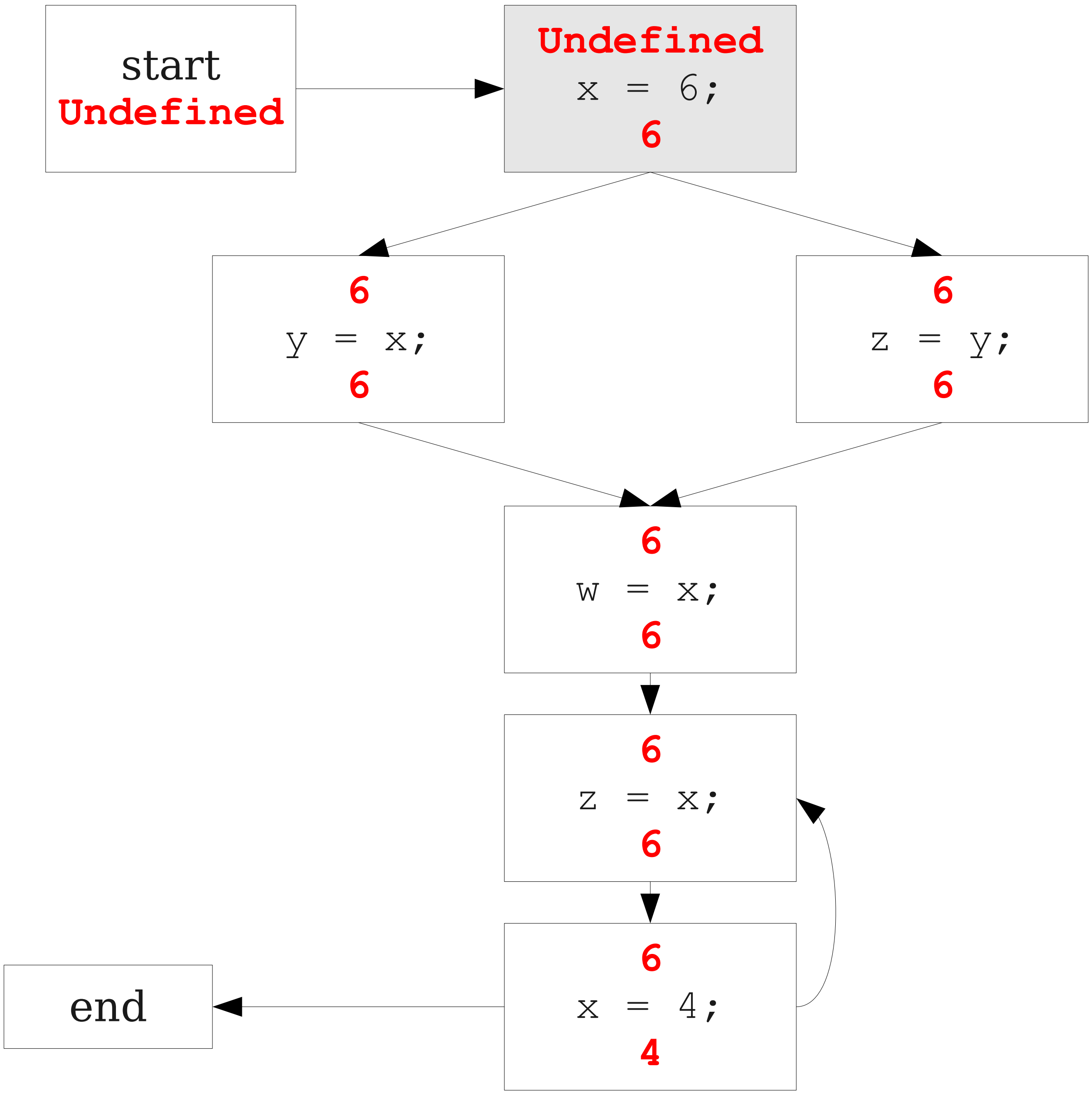
Global Constant Propagation



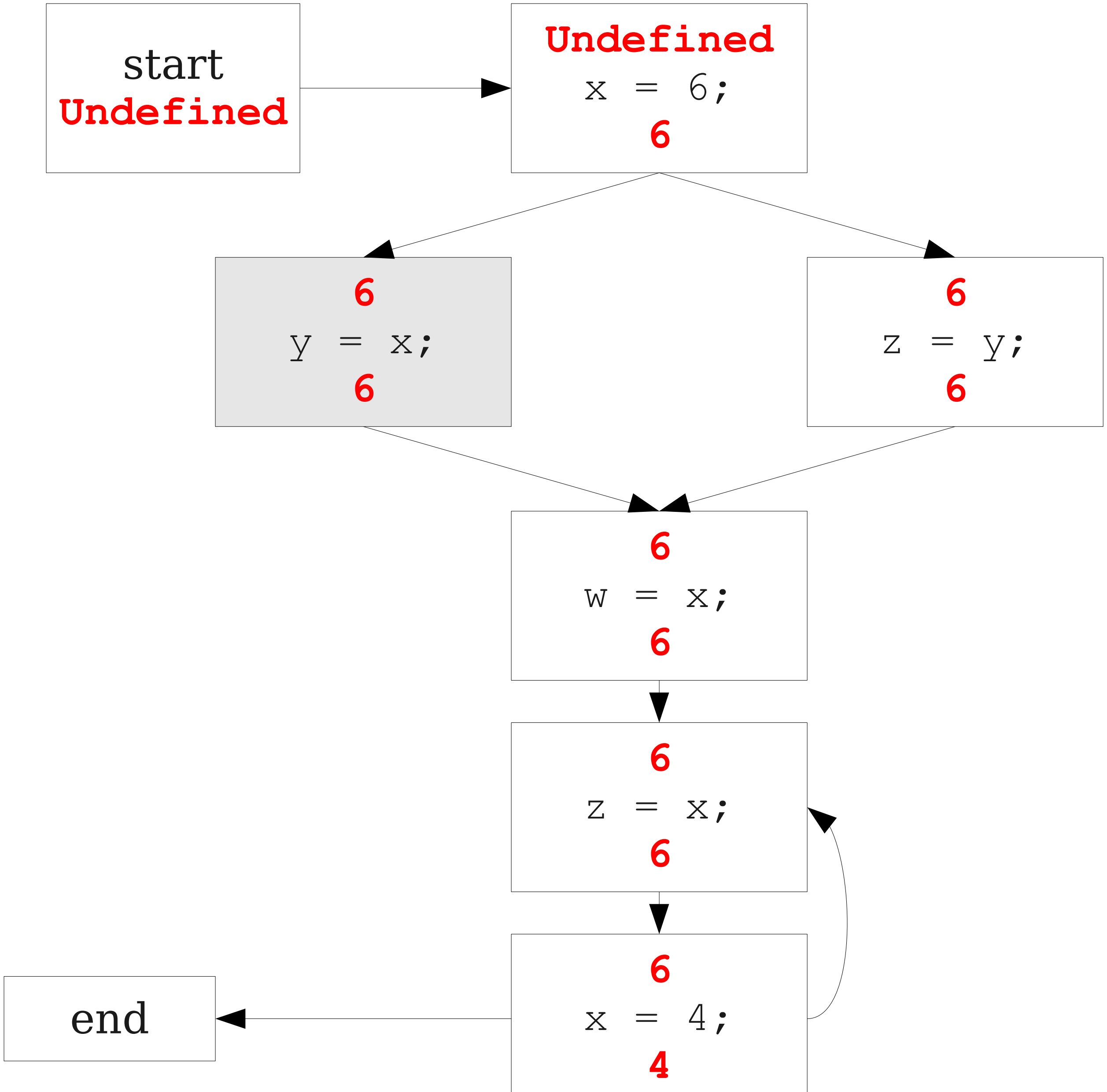
Global Constant Propagation



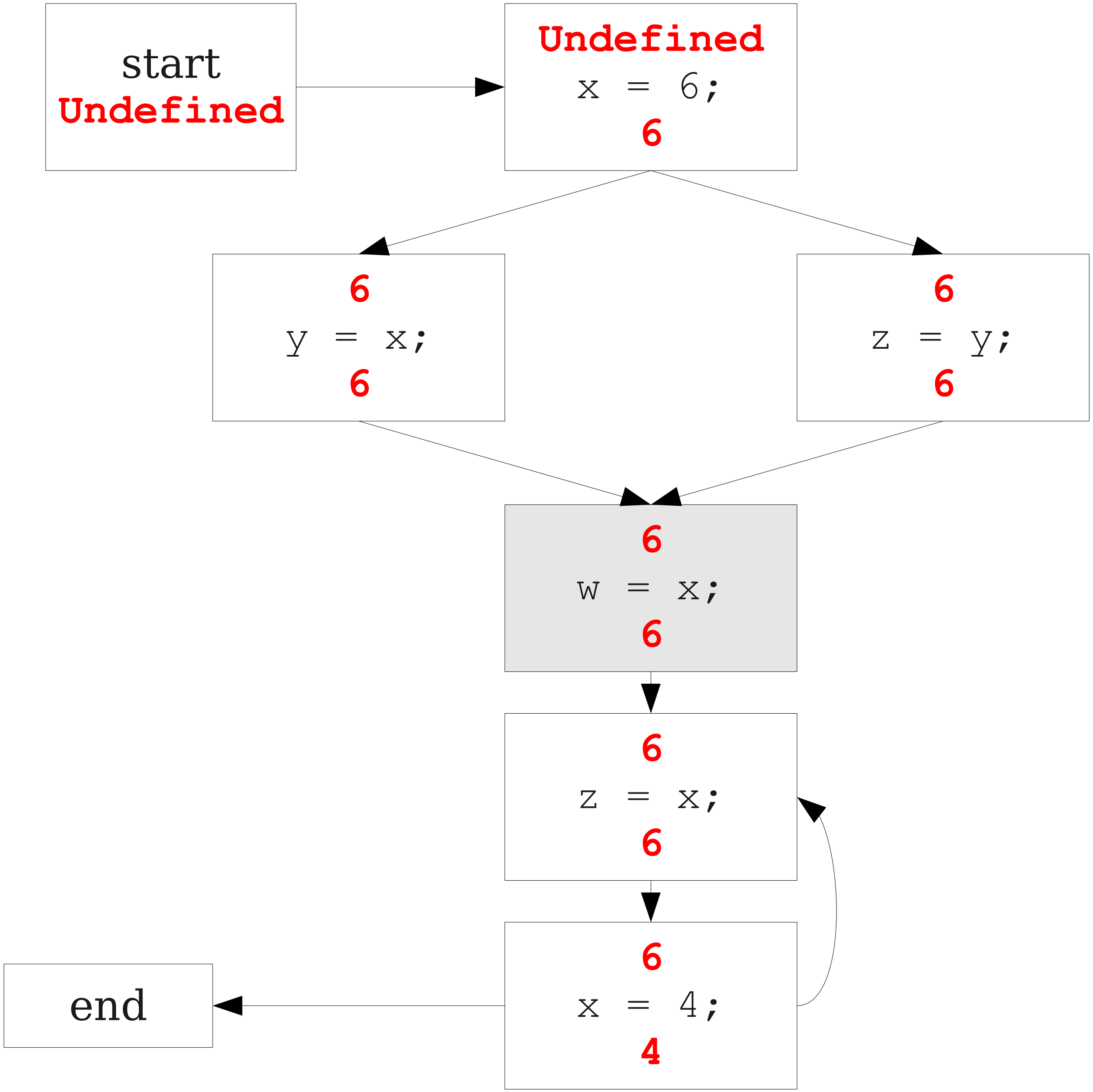
Global Constant Propagation



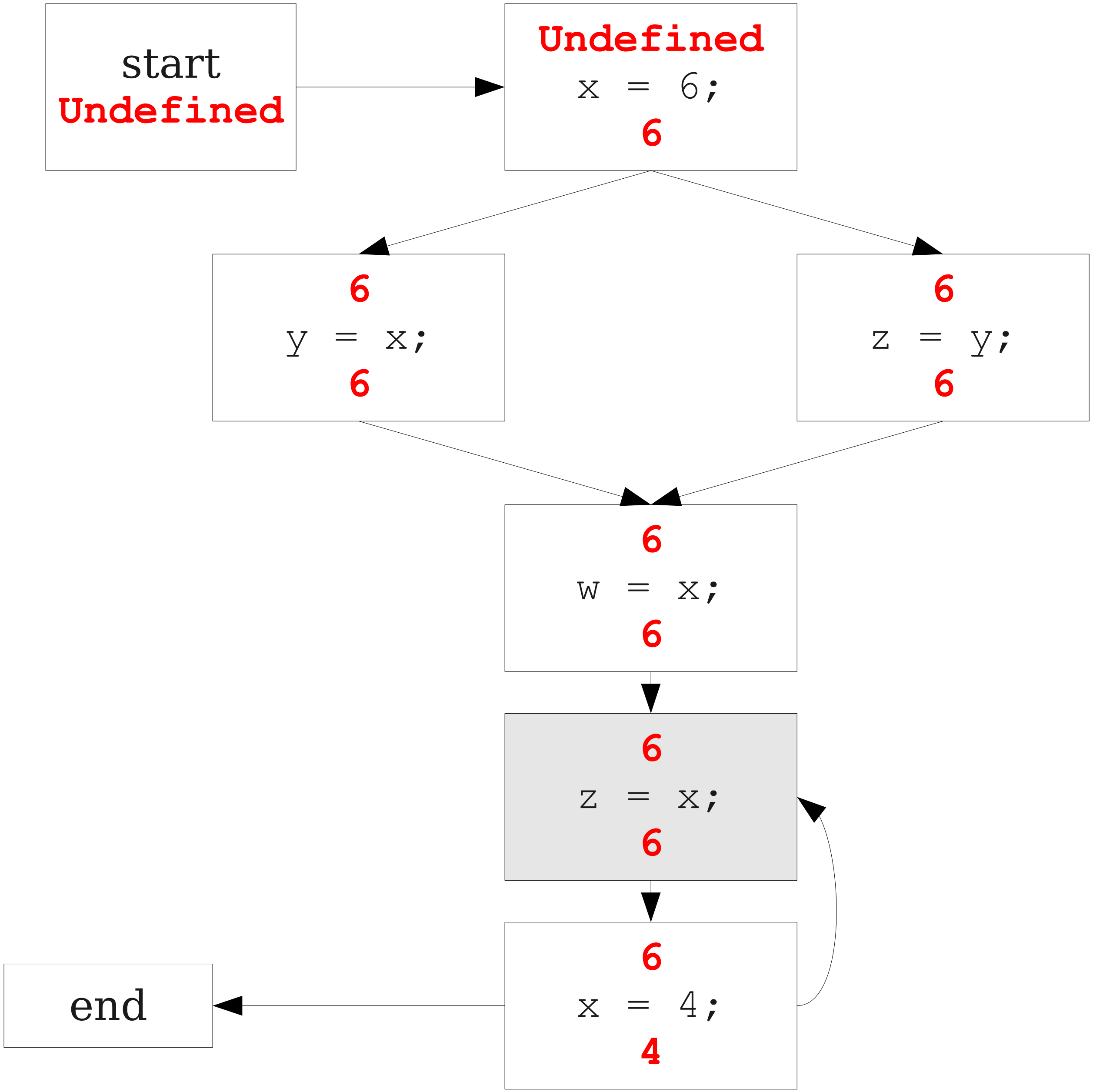
Global Constant Propagation



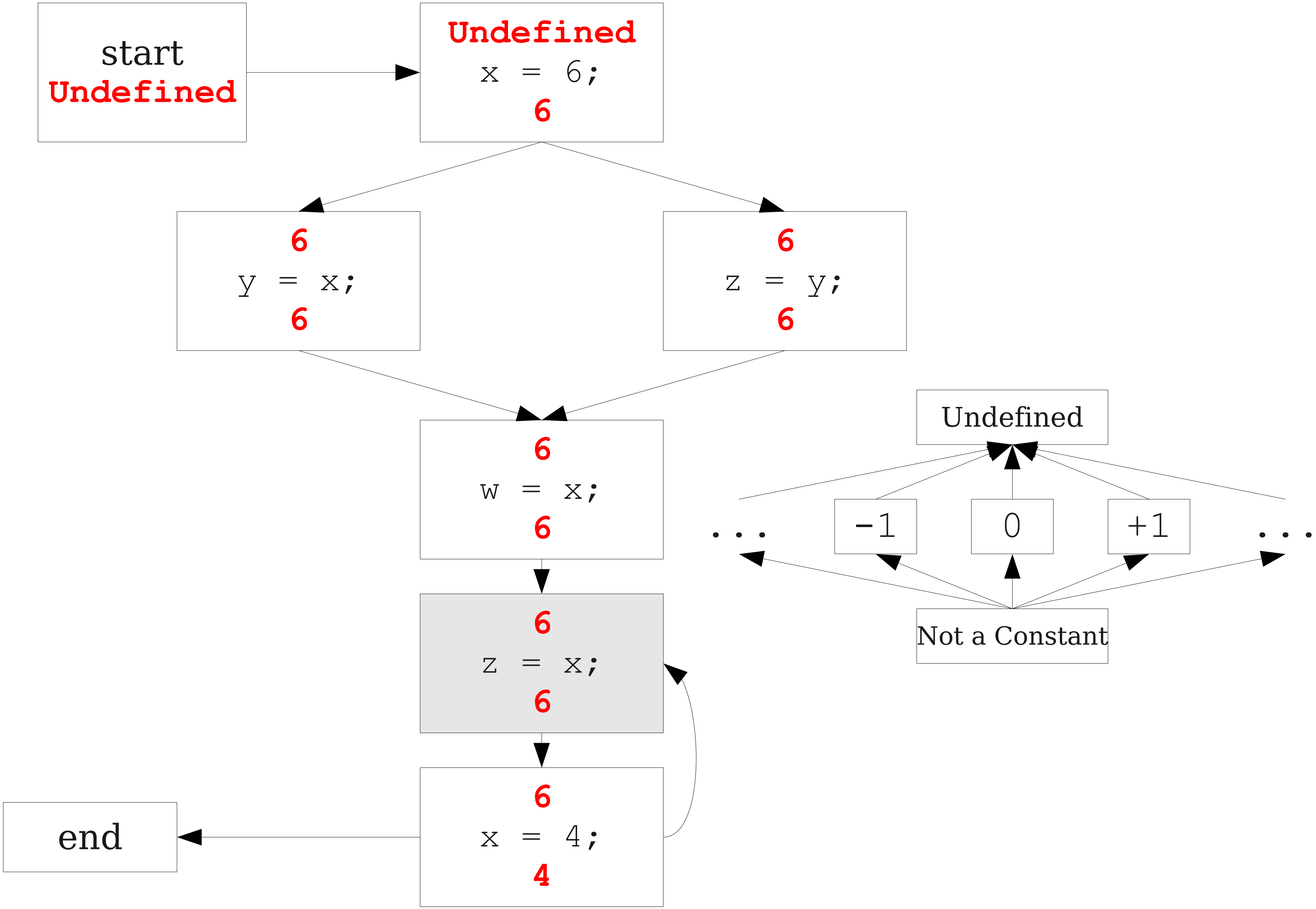
Global Constant Propagation



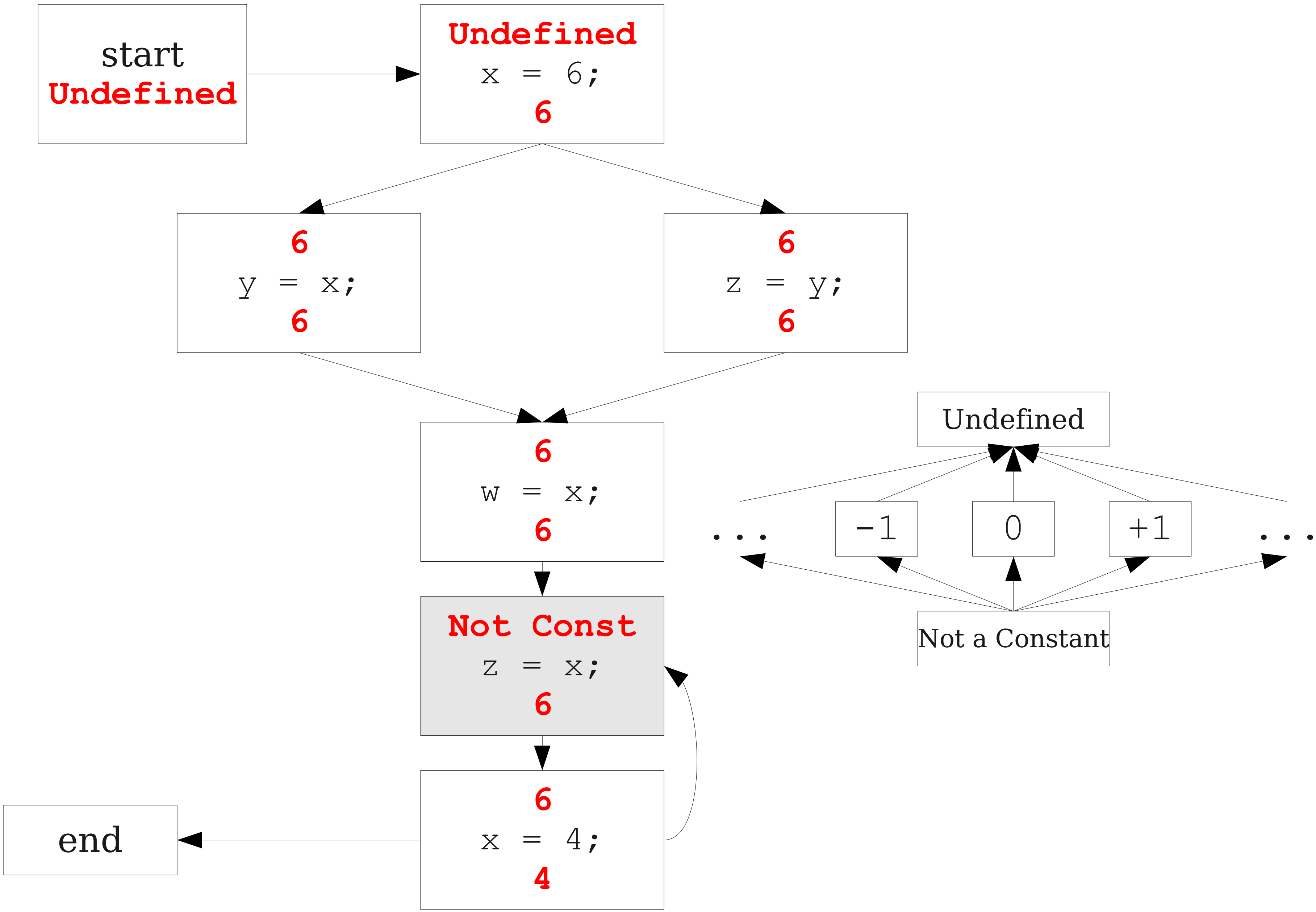
Global Constant Propagation



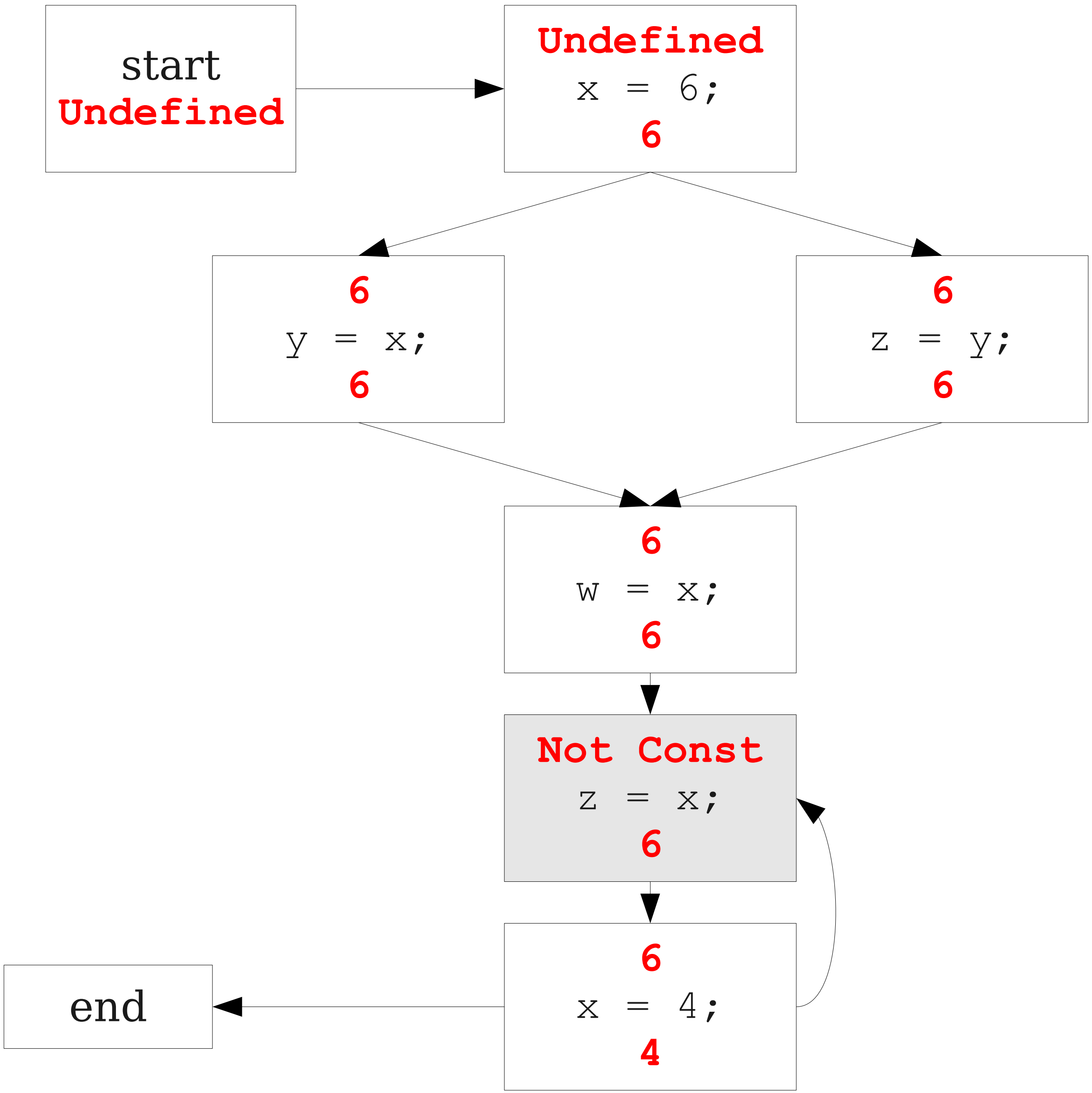
Global Constant Propagation



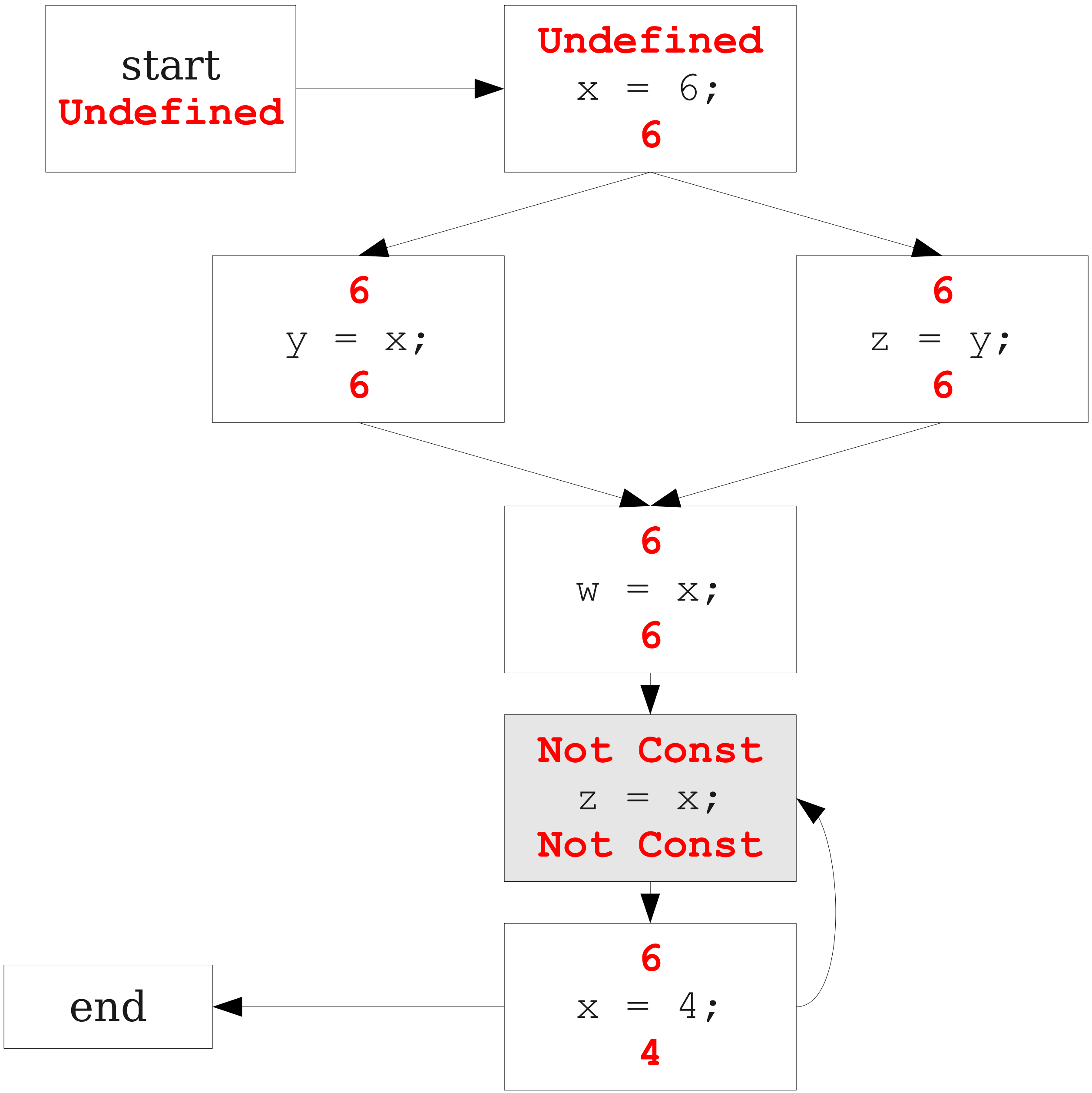
Global Constant Propagation



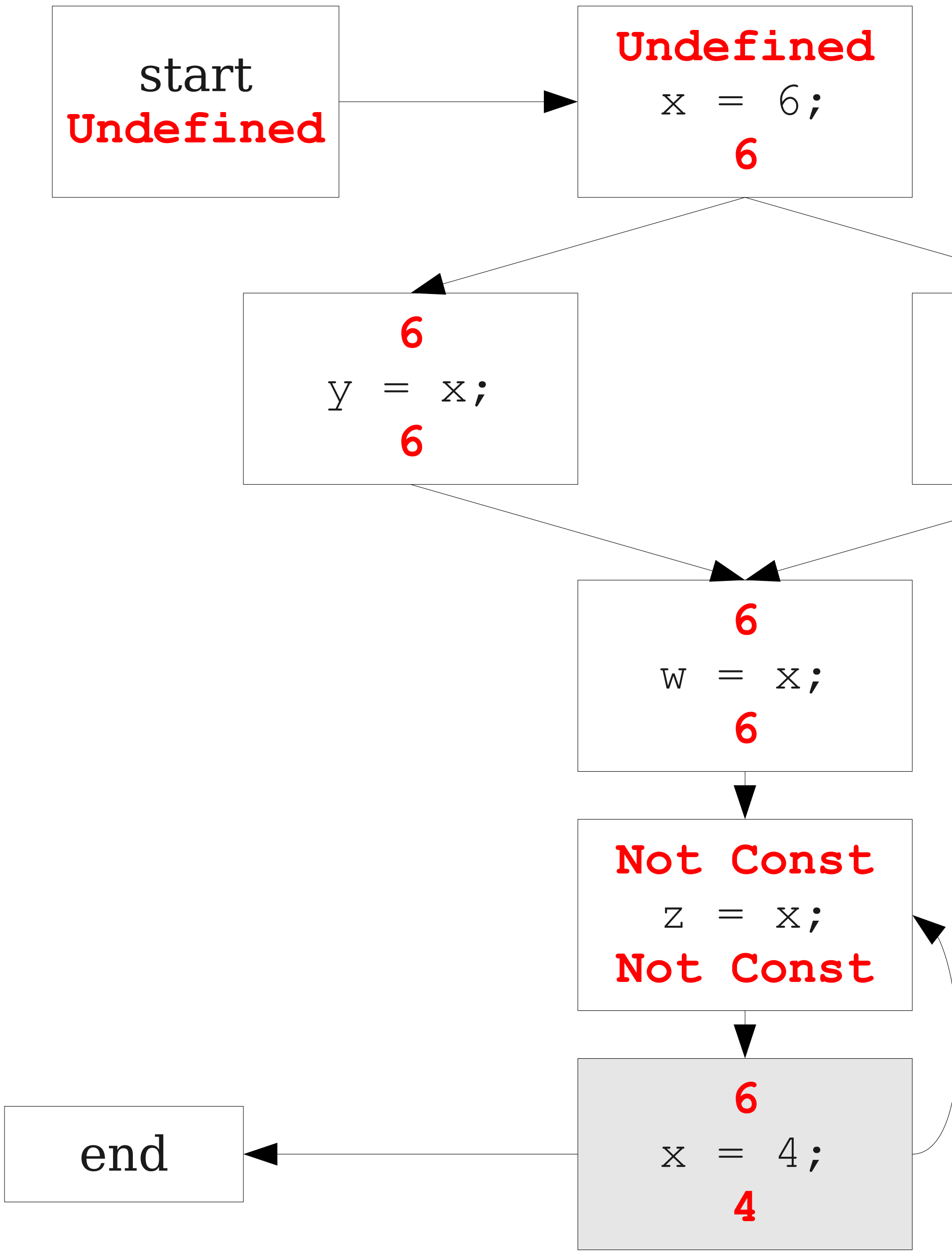
Global Constant Propagation



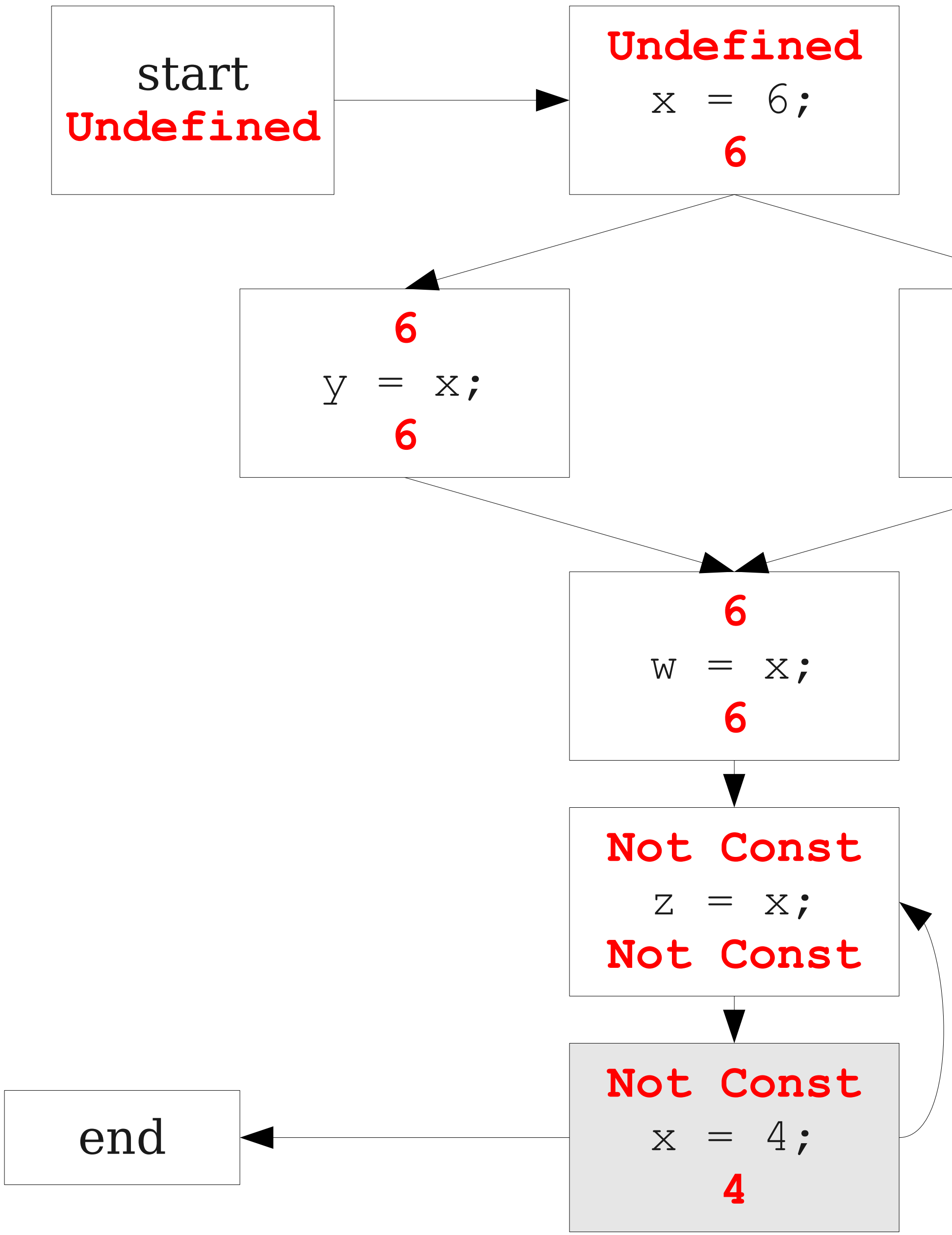
Global Constant Propagation



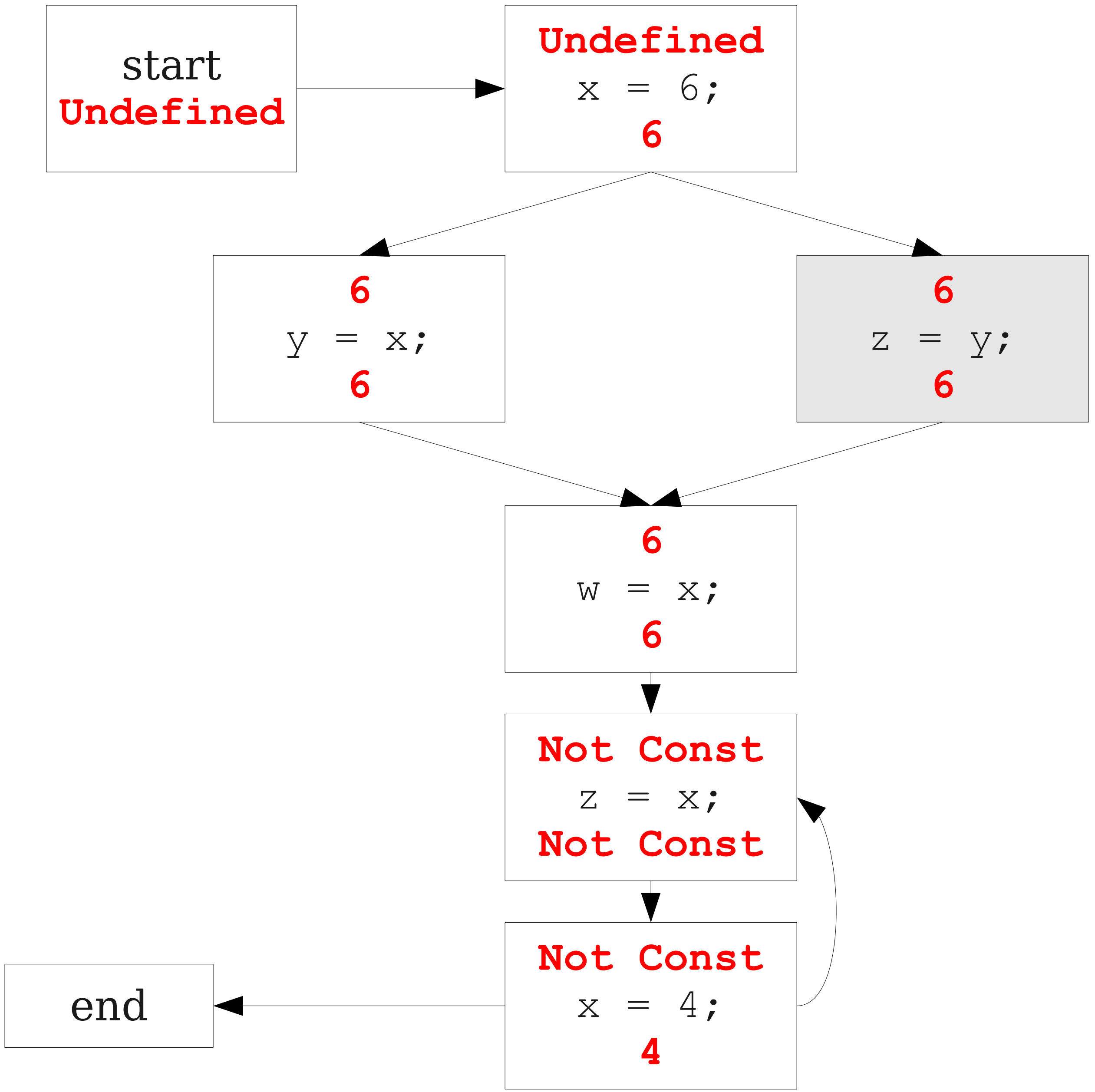
Global Constant Propagation



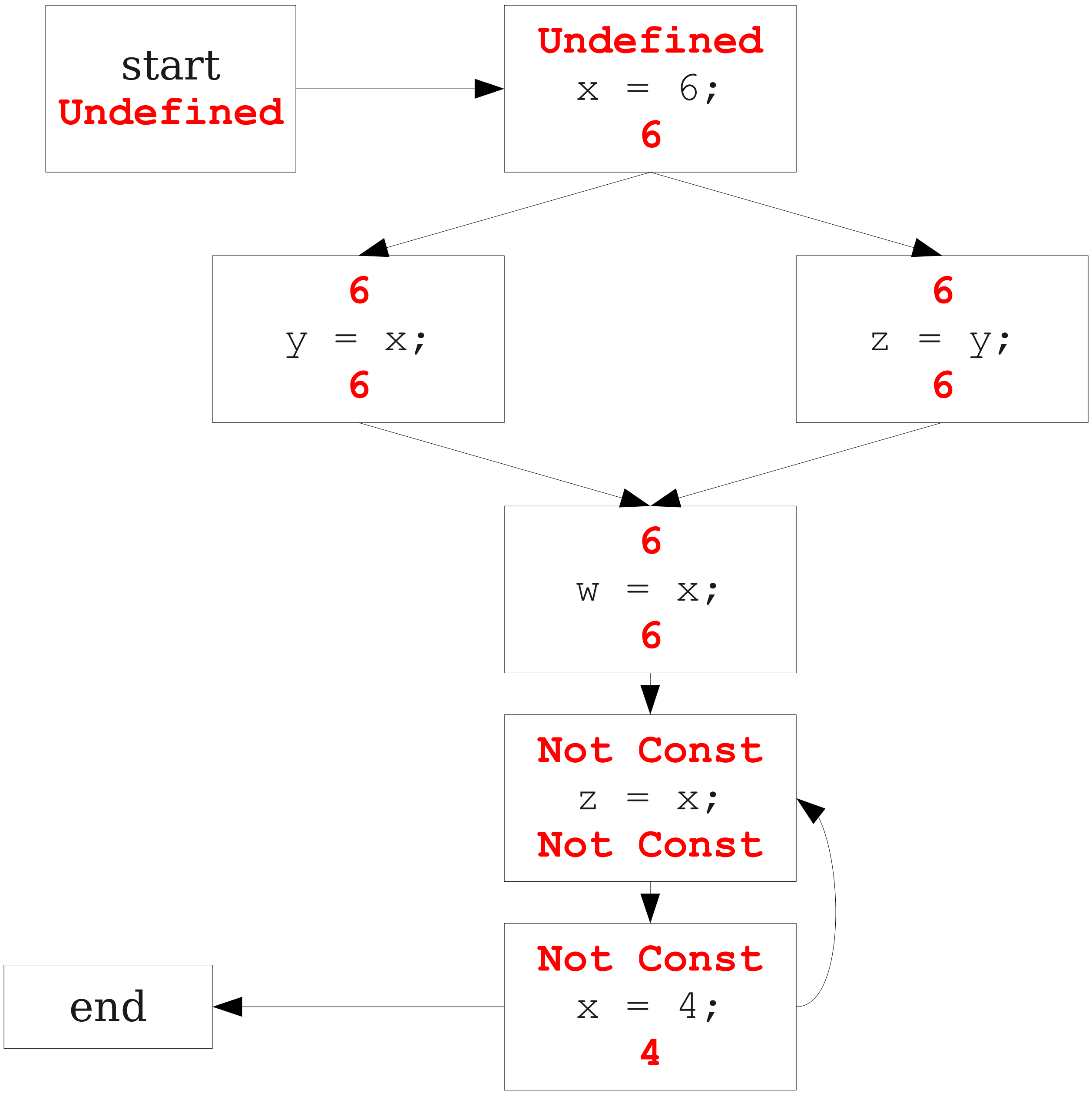
Global Constant Propagation



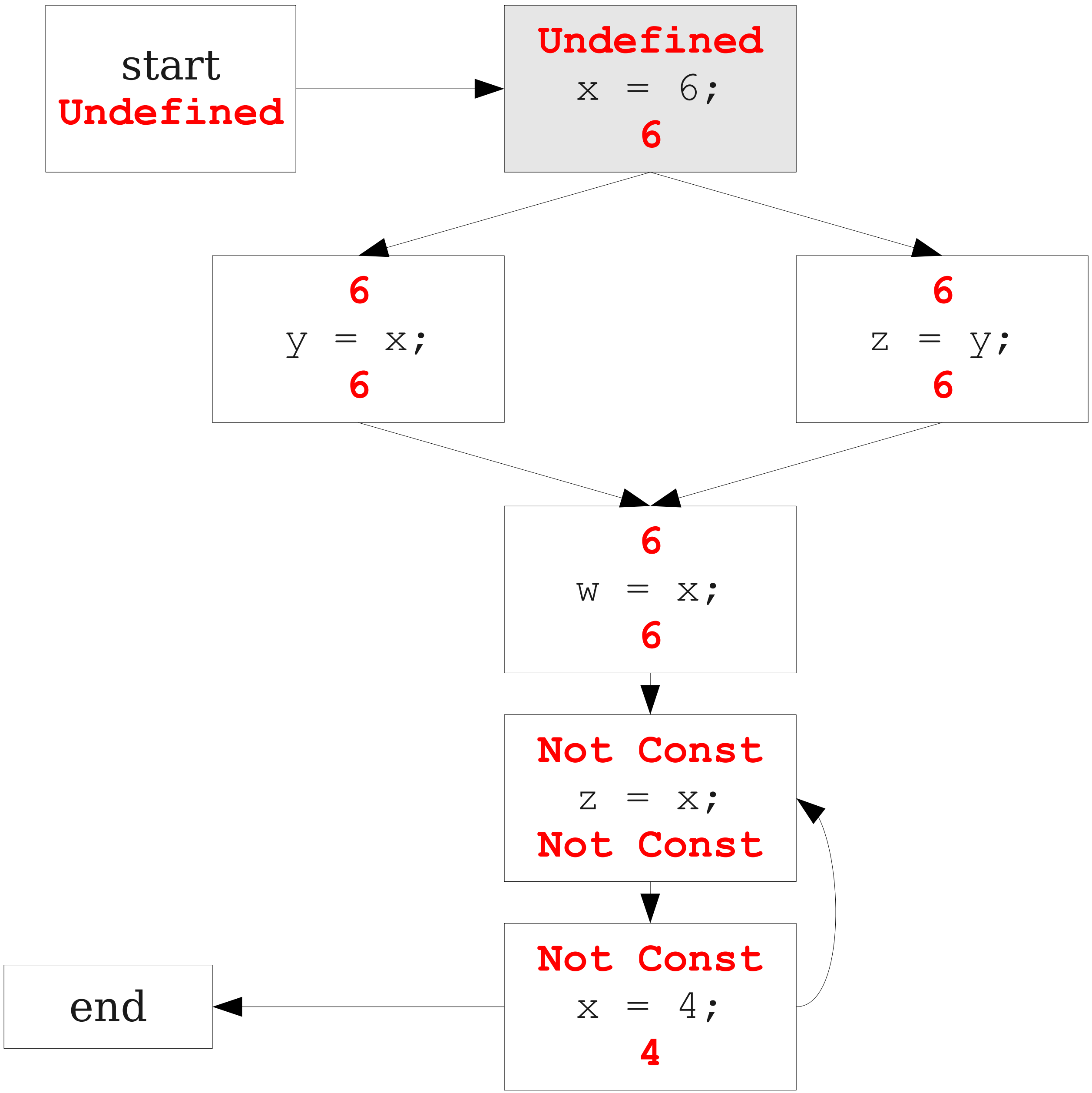
Global Constant Propagation



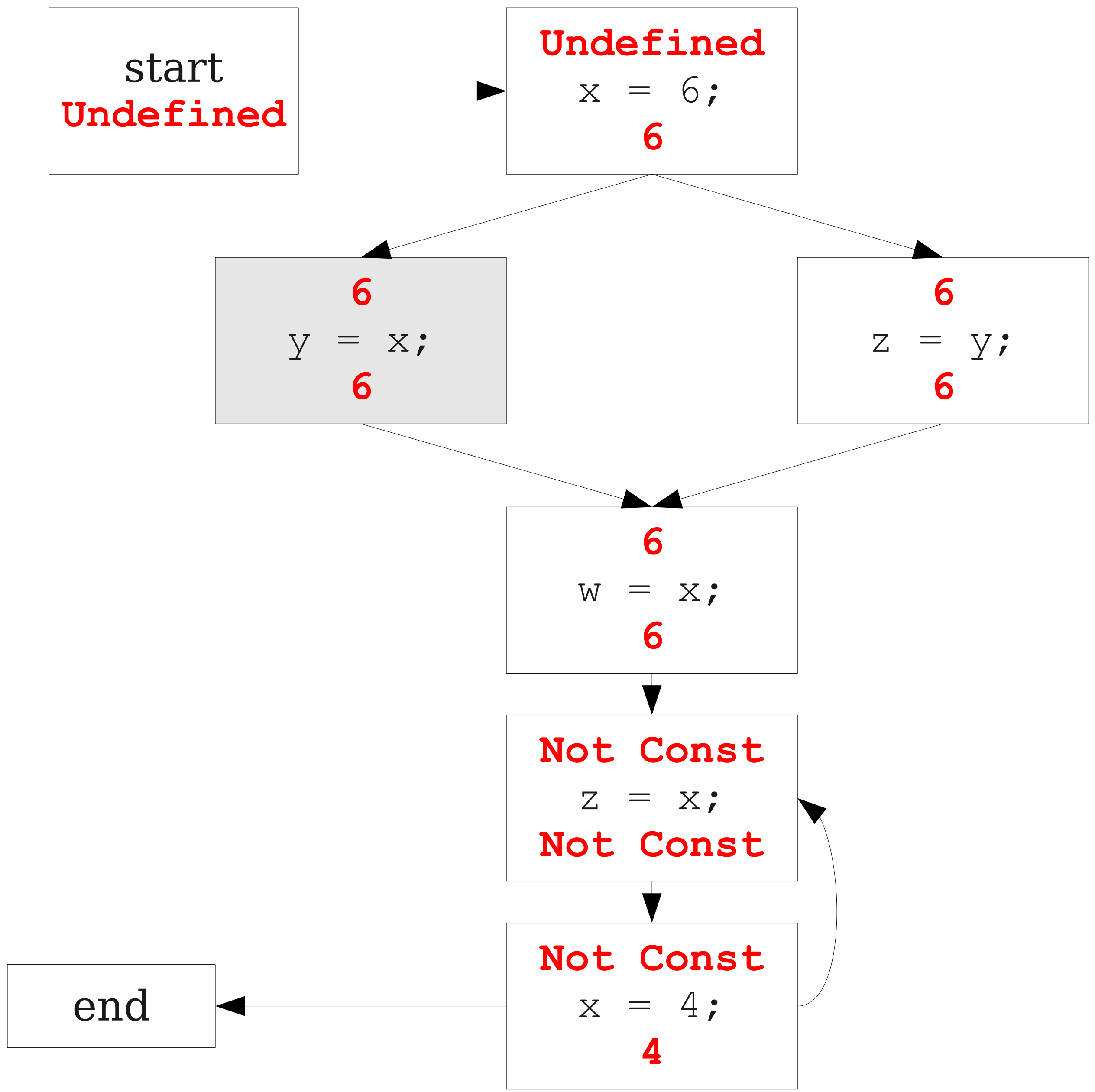
Global Constant Propagation



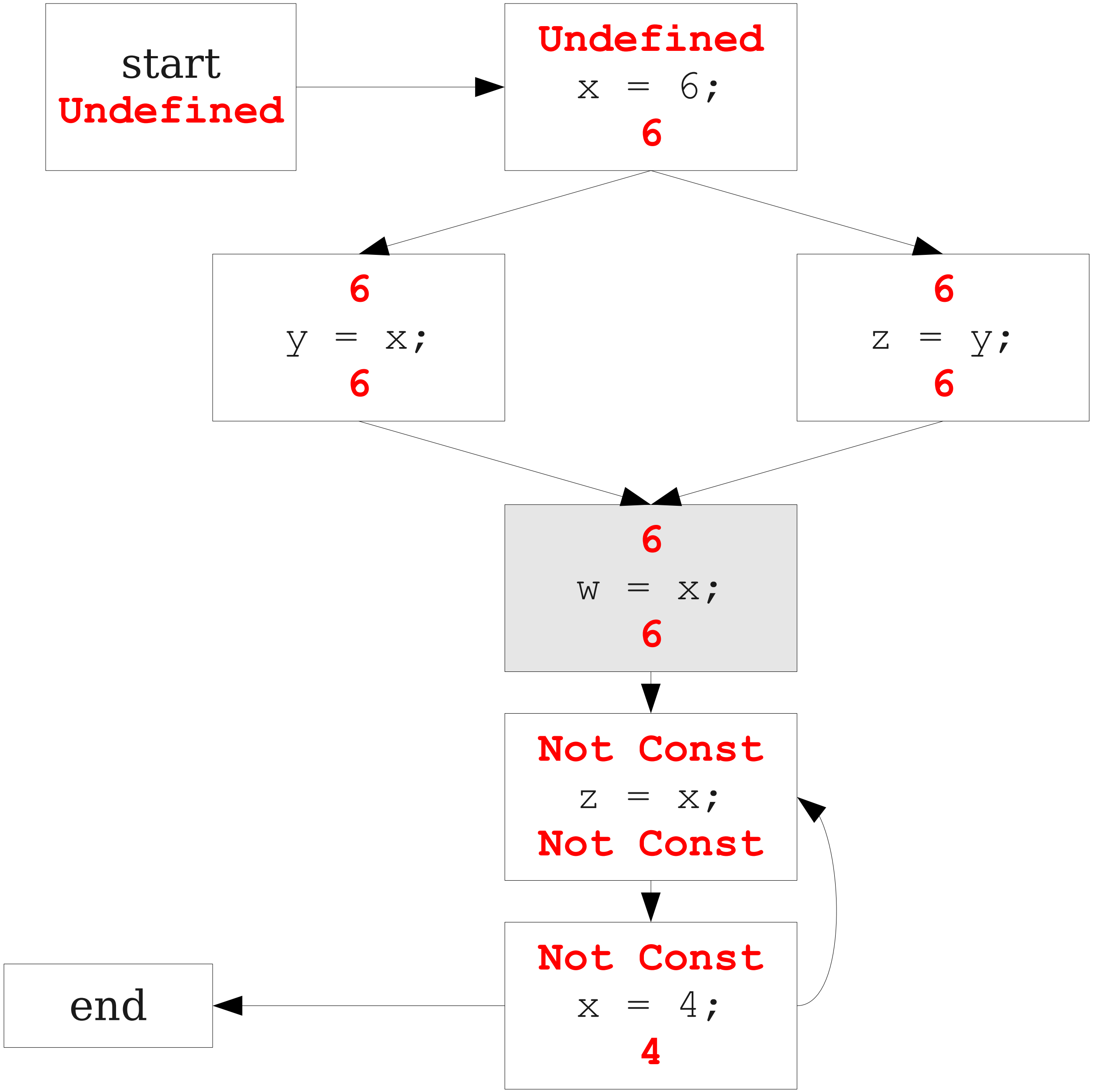
Global Constant Propagation



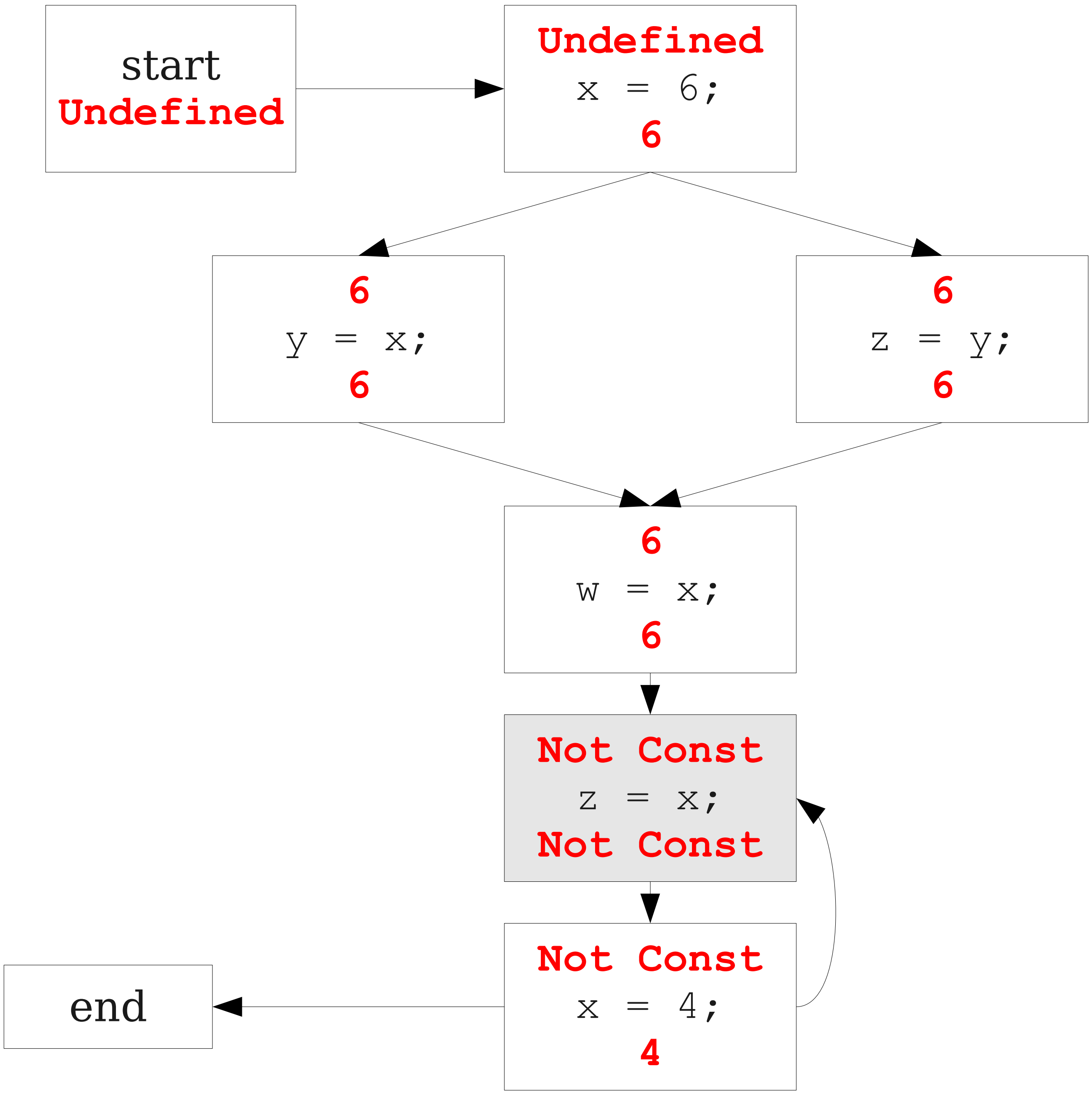
Global Constant Propagation



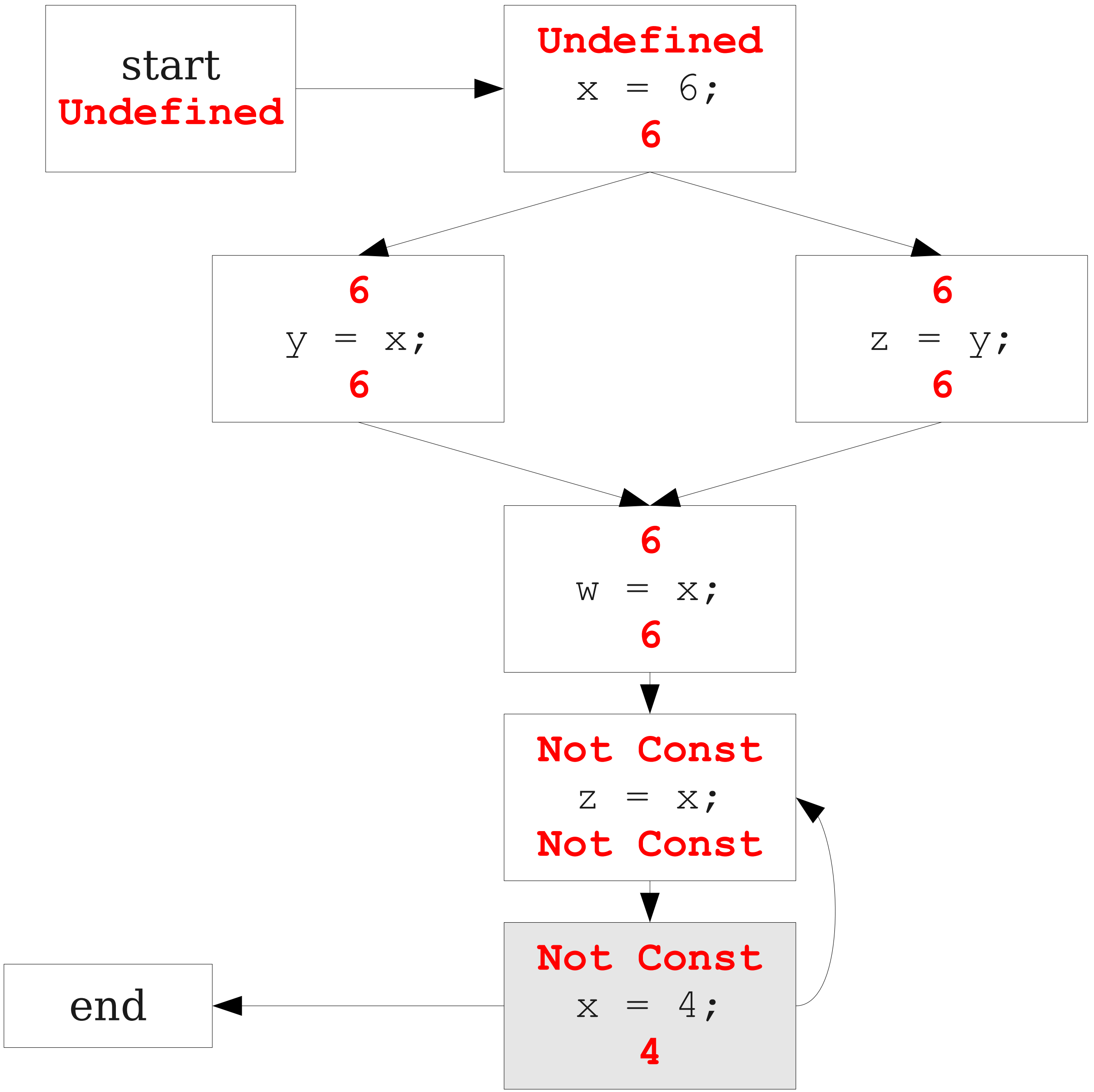
Global Constant Propagation



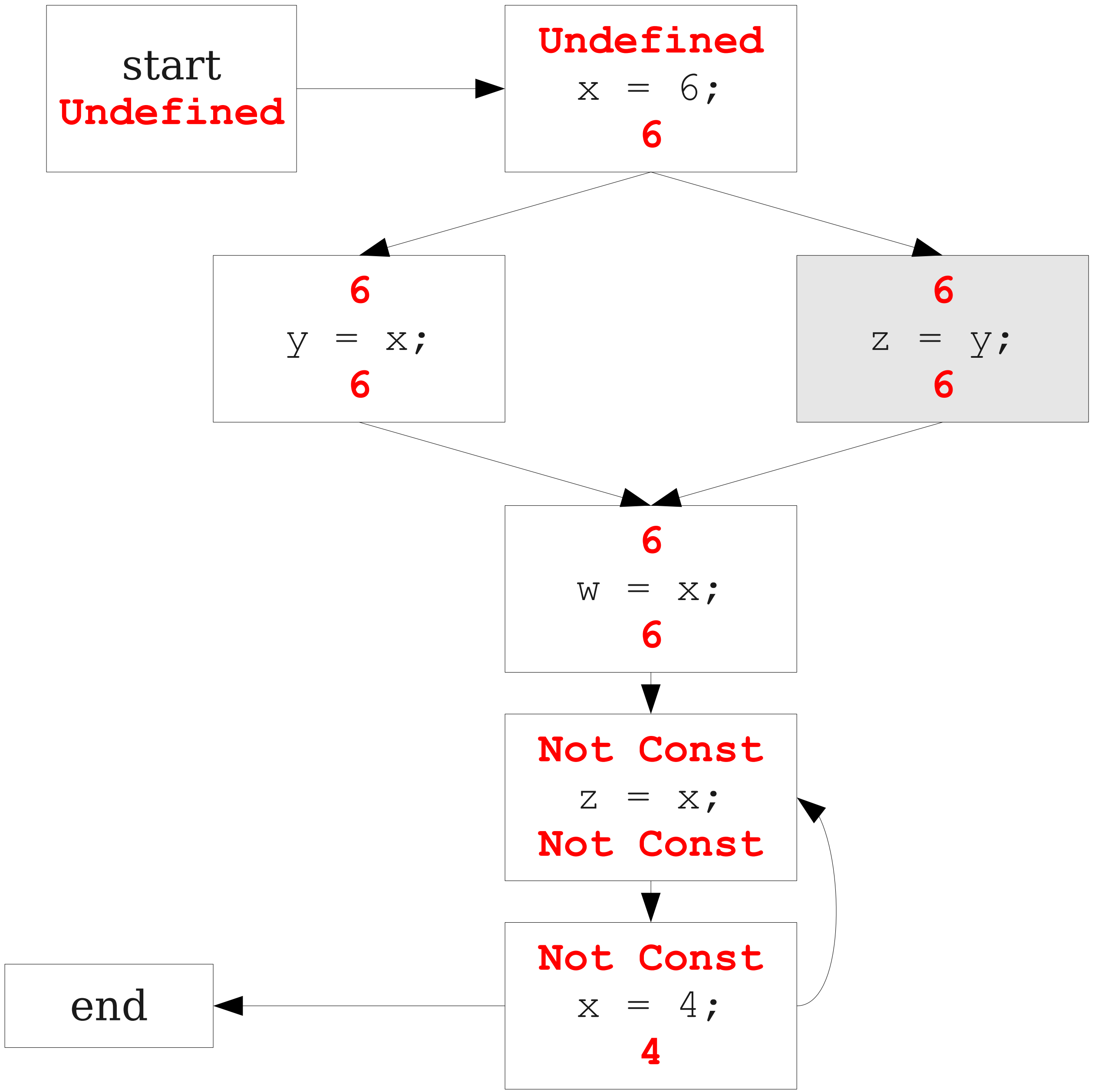
Global Constant Propagation



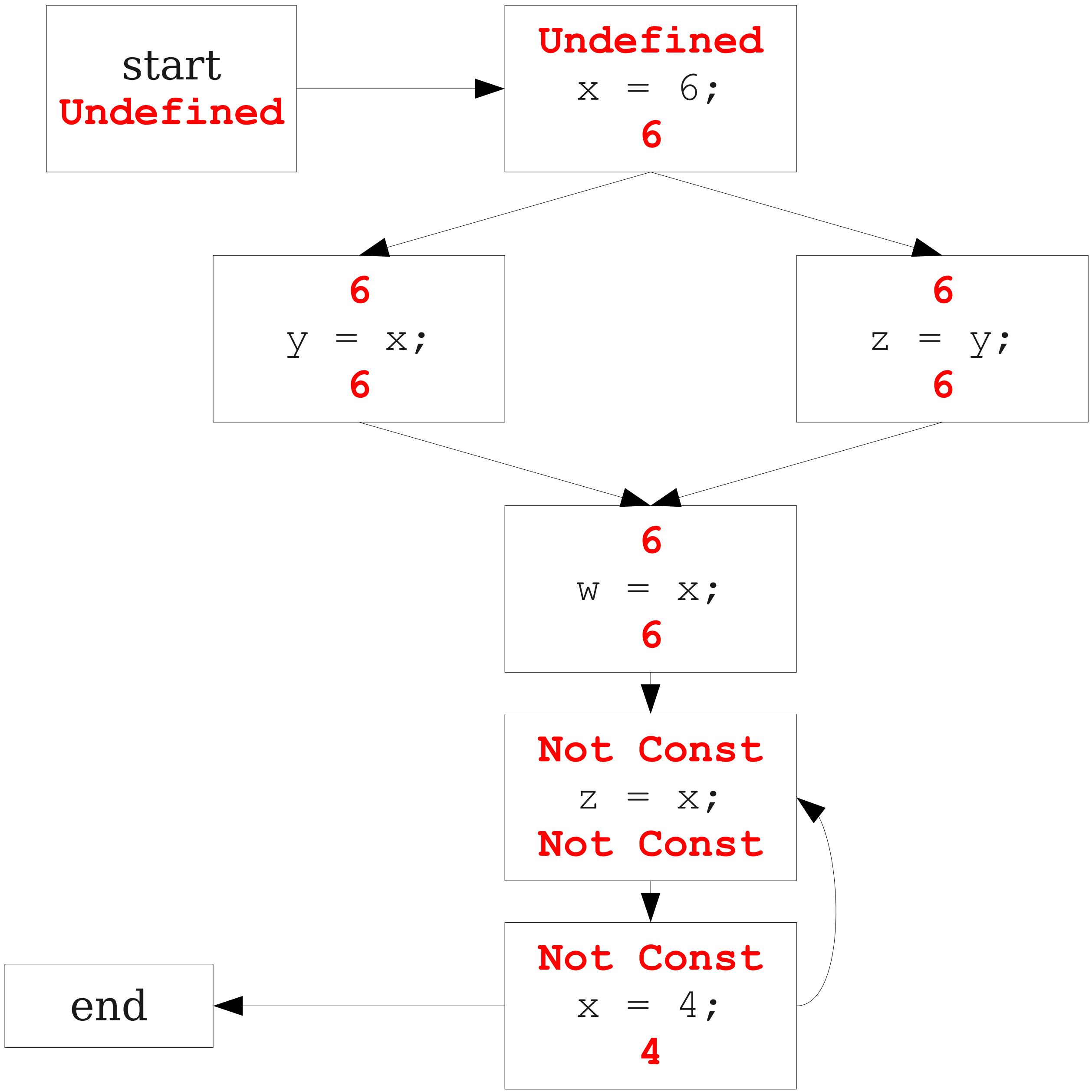
Global Constant Propagation



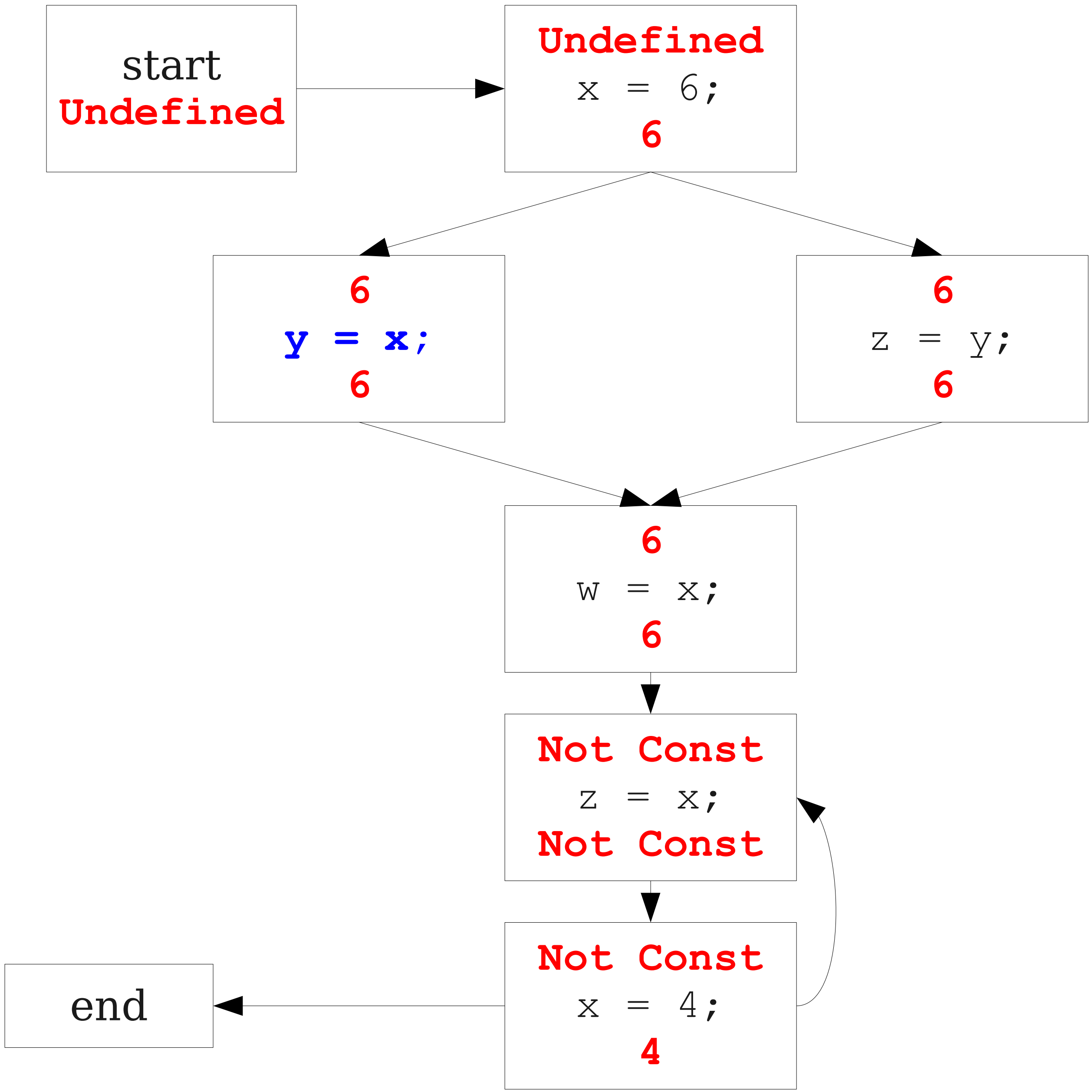
Global Constant Propagation



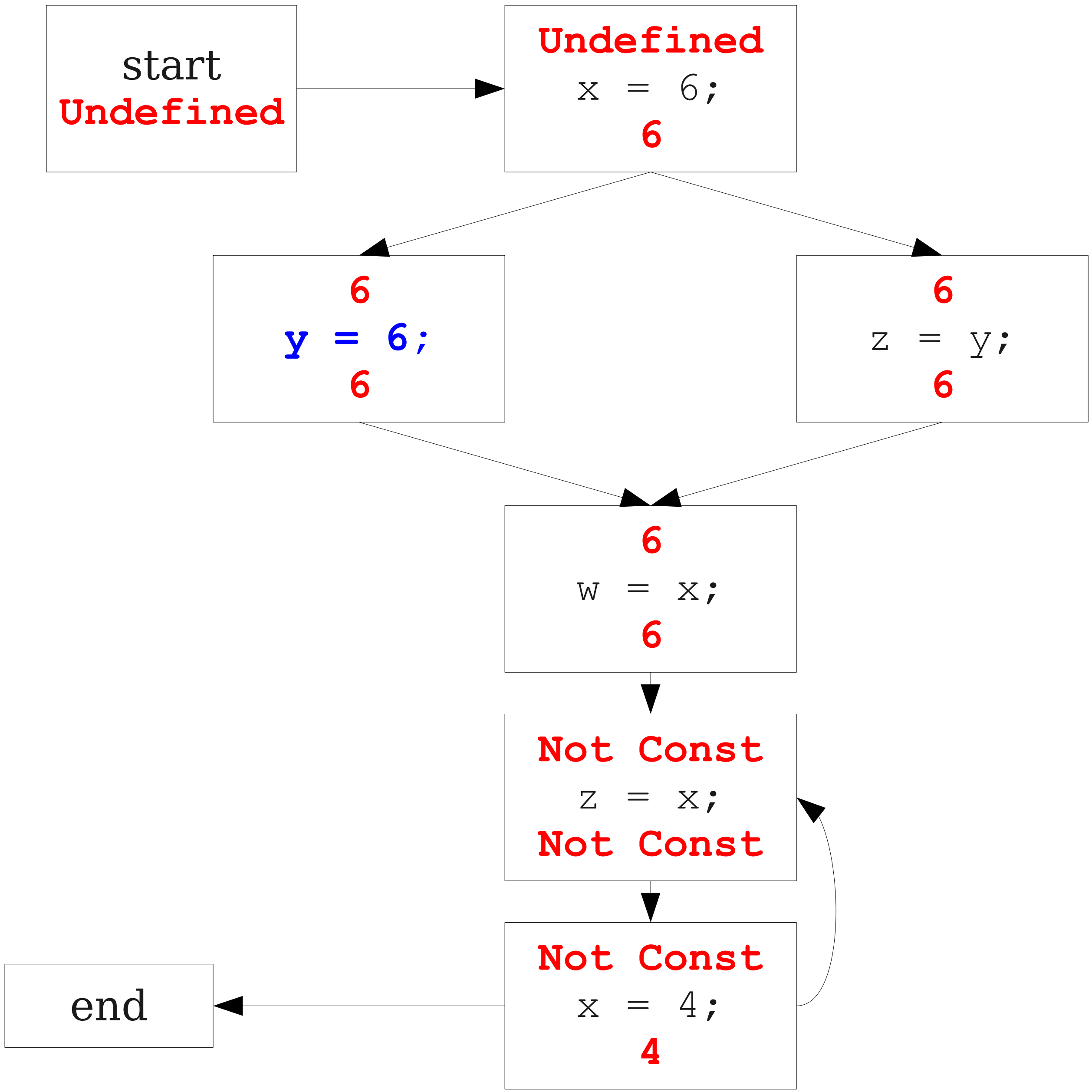
Global Constant Propagation



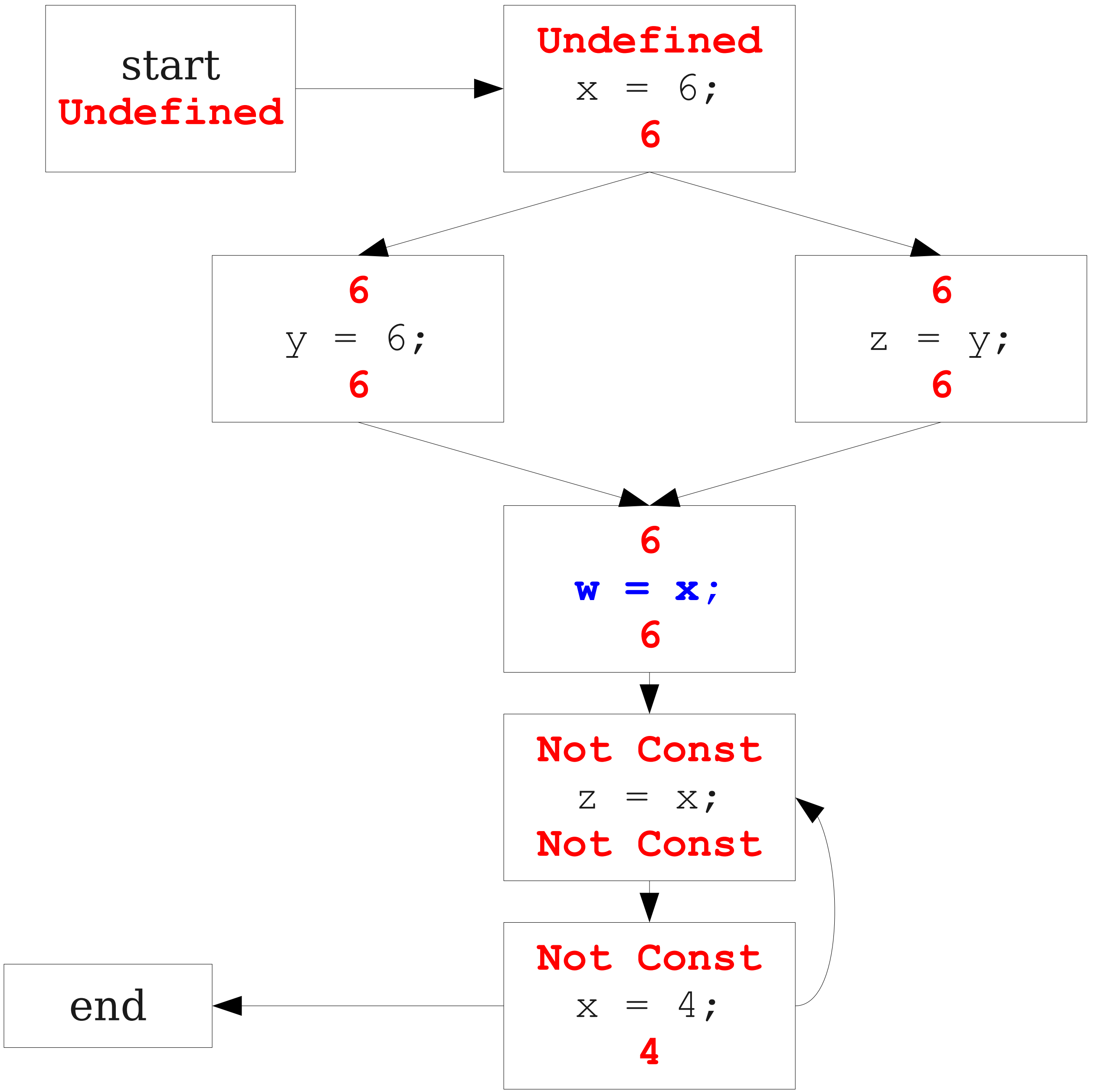
Global Constant Propagation



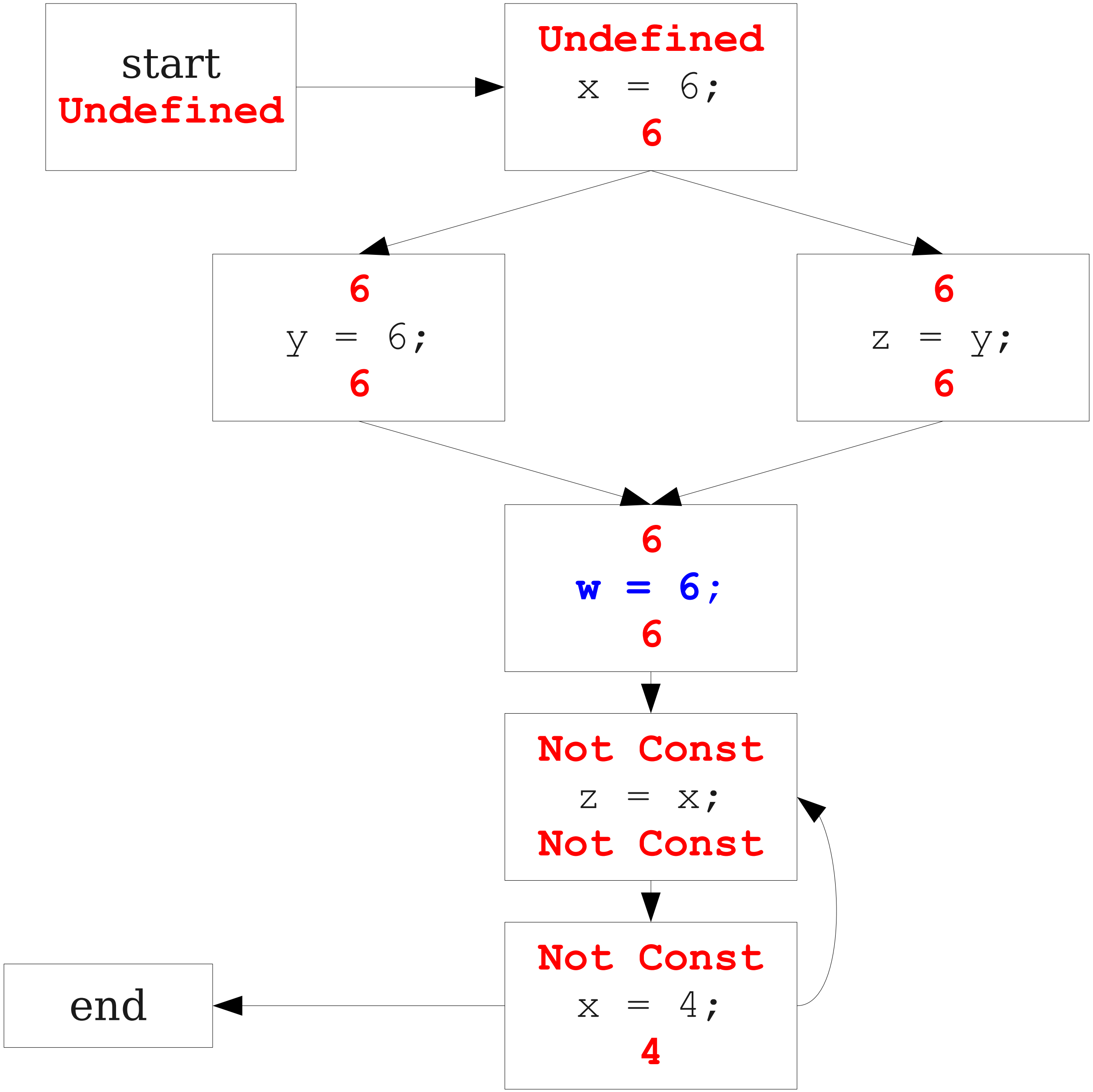
Global Constant Propagation



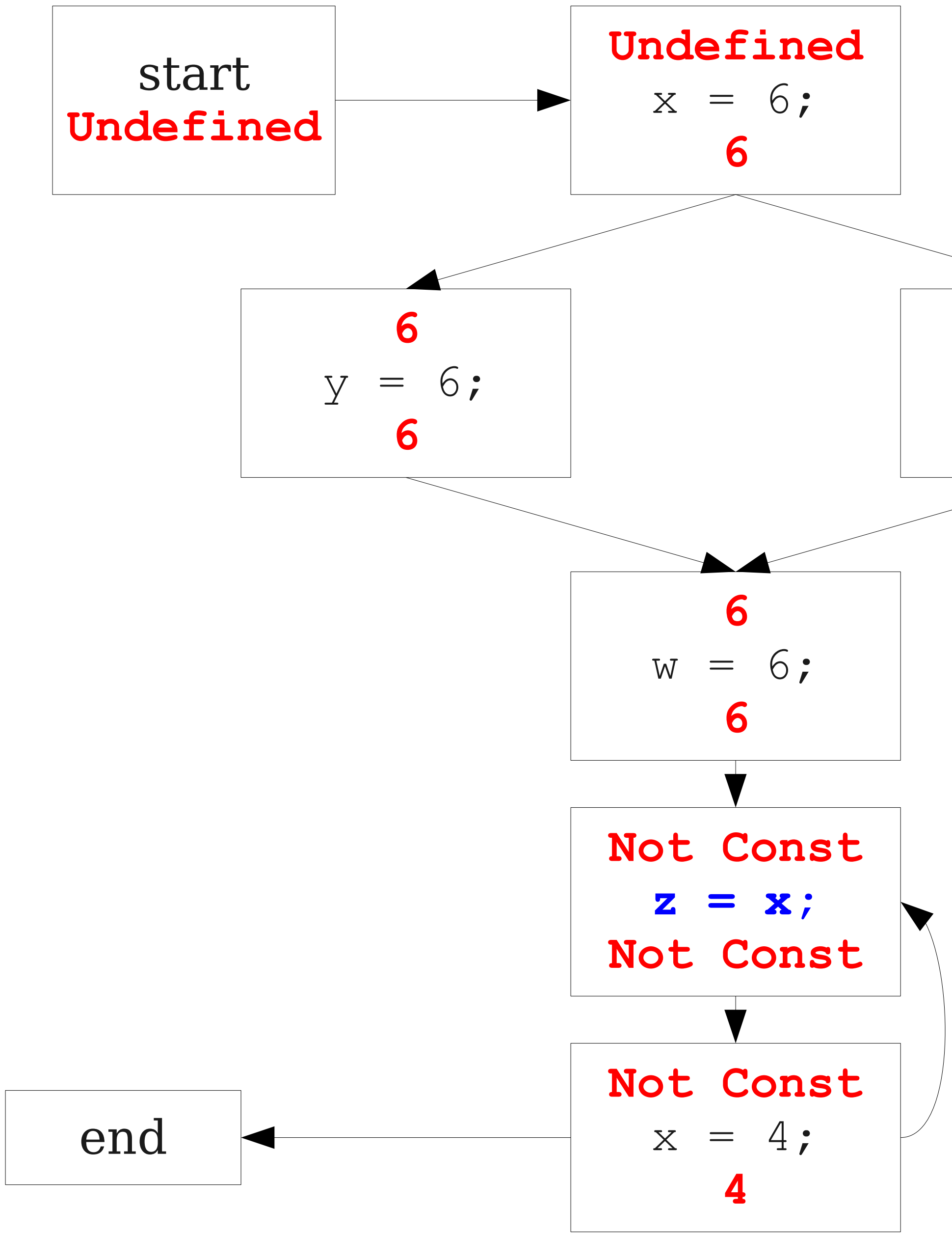
Global Constant Propagation



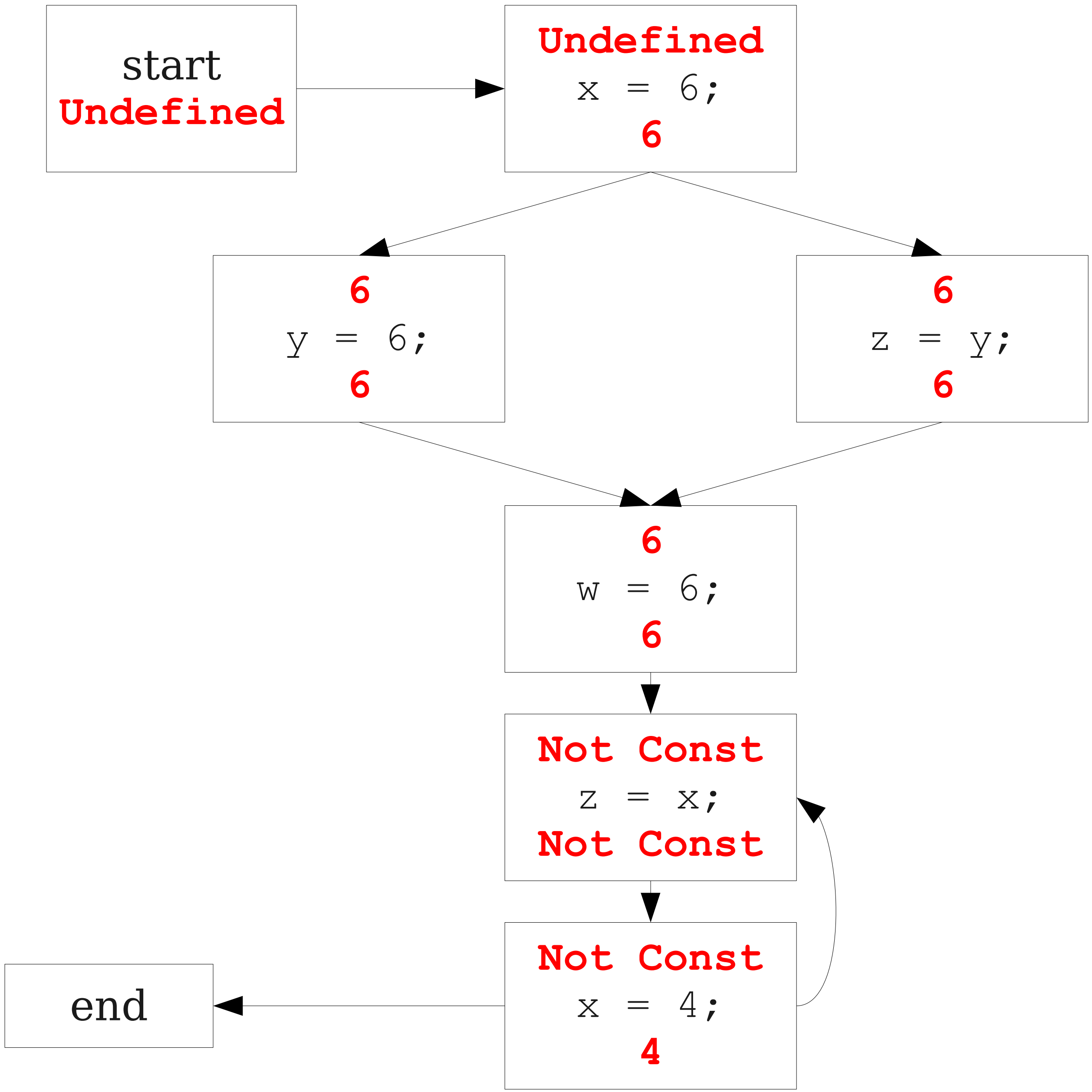
Global Constant Propagation



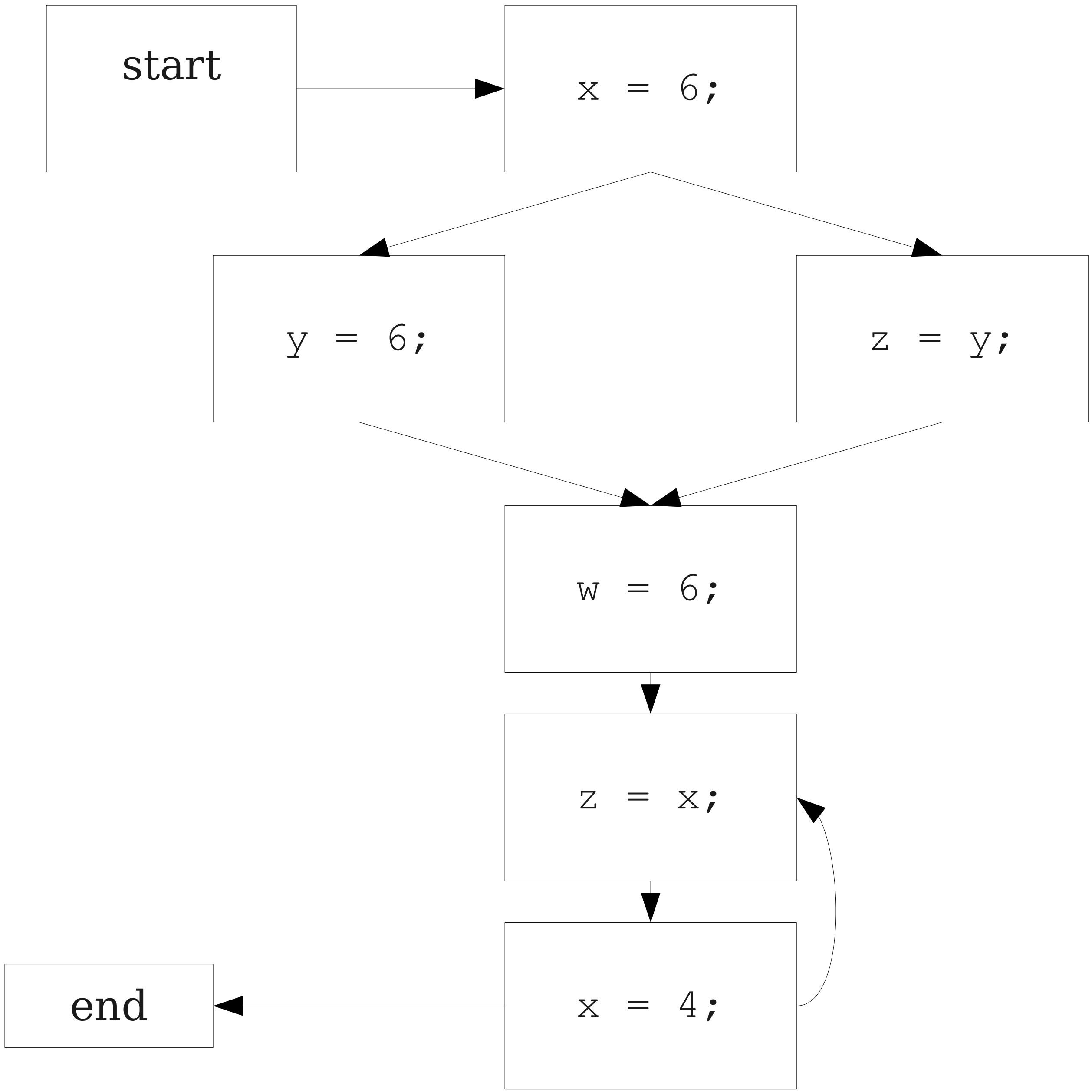
Global Constant Propagation



Global Constant Propagation



Global Constant Propagation



Dataflow for Constant Propagation

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