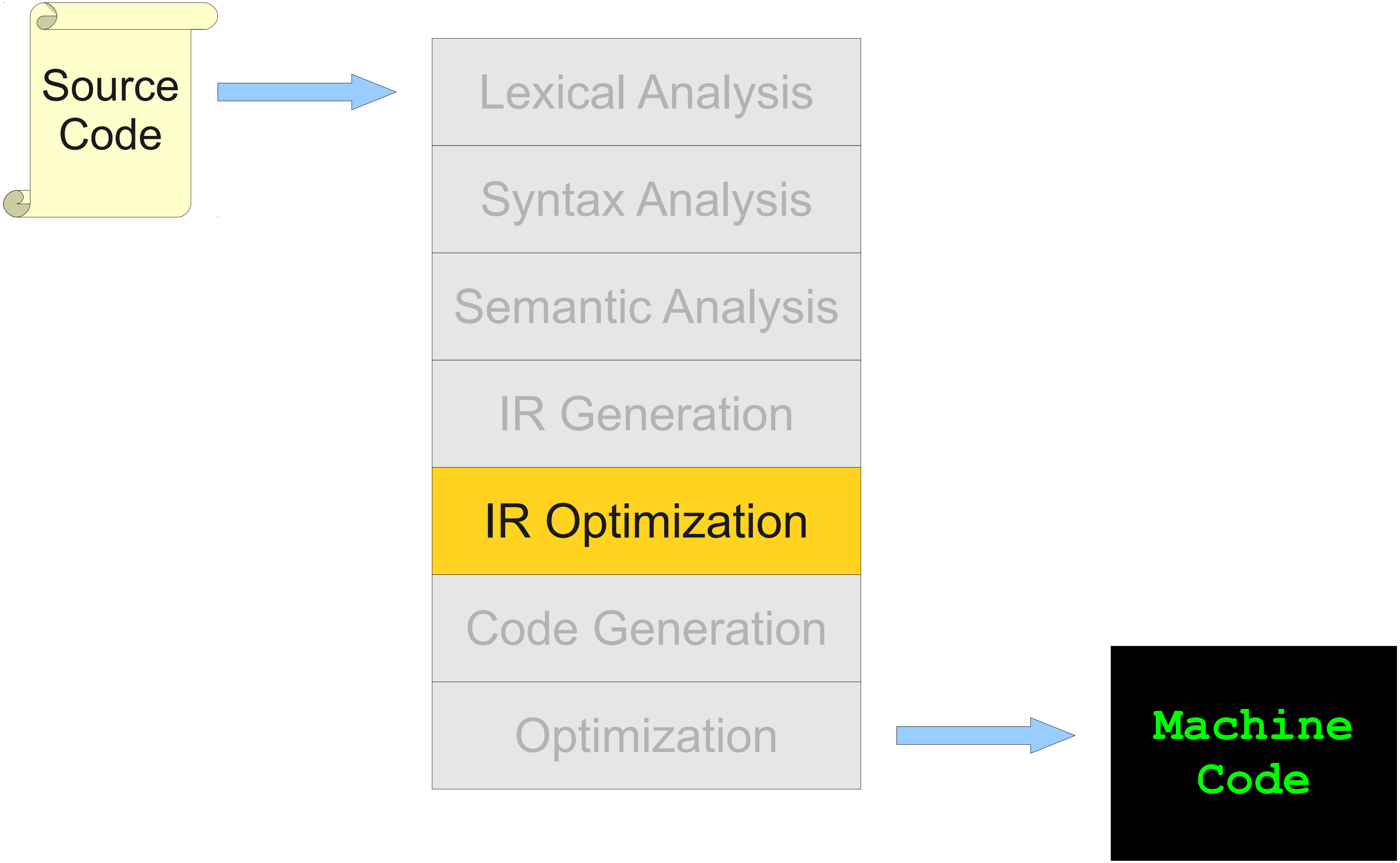
Global Optimization

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!

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

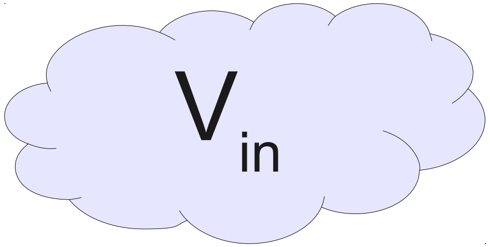
- A **local optimization** is an optimization of IR instructions within a single basic block.
- We saw five examples of this:
  - **Common subexpression elimination**.
  - **Copy propagation**.
  - **Dead code elimination**.
  - **Arithmetic simplification**.
  - **Constant folding**.

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.

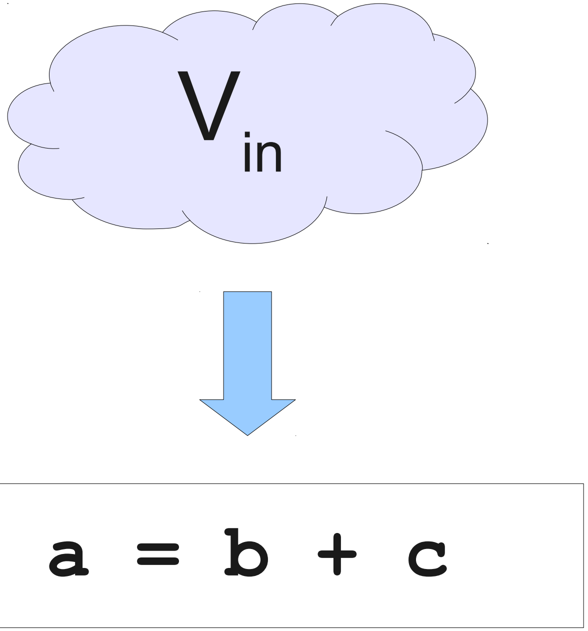
Another View of Local Analyses

Another View of Local Analyses



V_{in}

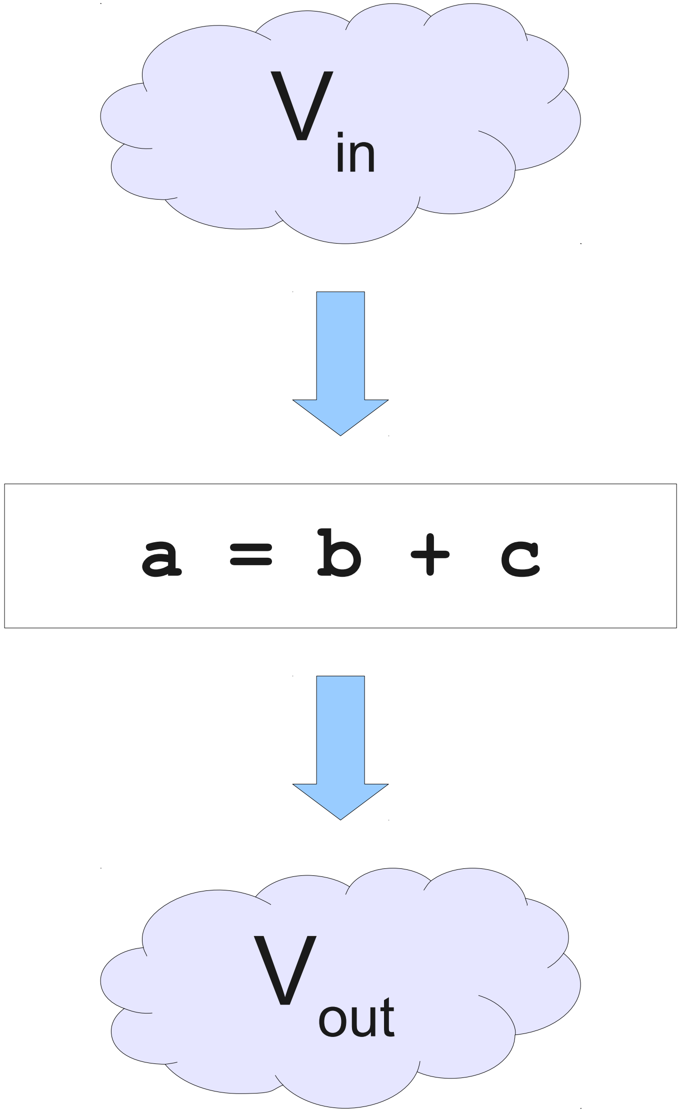
Another View of Local Analyses



V_{in}

`a = b + c`

Another View of Local Analyses

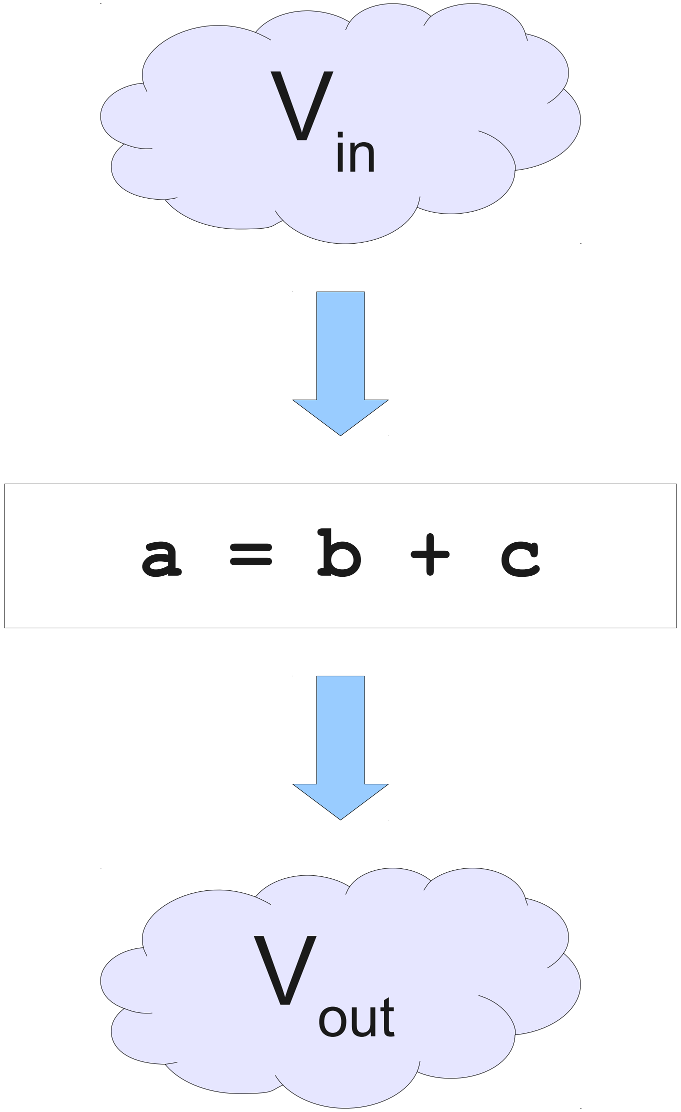


V_{in}

`a = b + c`

V_{out}

Another View of Local Analyses



V_{in}

`a = b + c`

V_{out}

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

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 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 value in **V** before the program starts.

Available Expressions

- **Direction**: Forward
- **Domain**: Sets of expressions assigned to variables.
- **Transfer functions**: Given a variable 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

- **Direction**: Backwards
- **Domain**: Sets of variables.
- **Transfer function**: Given a variable 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**.

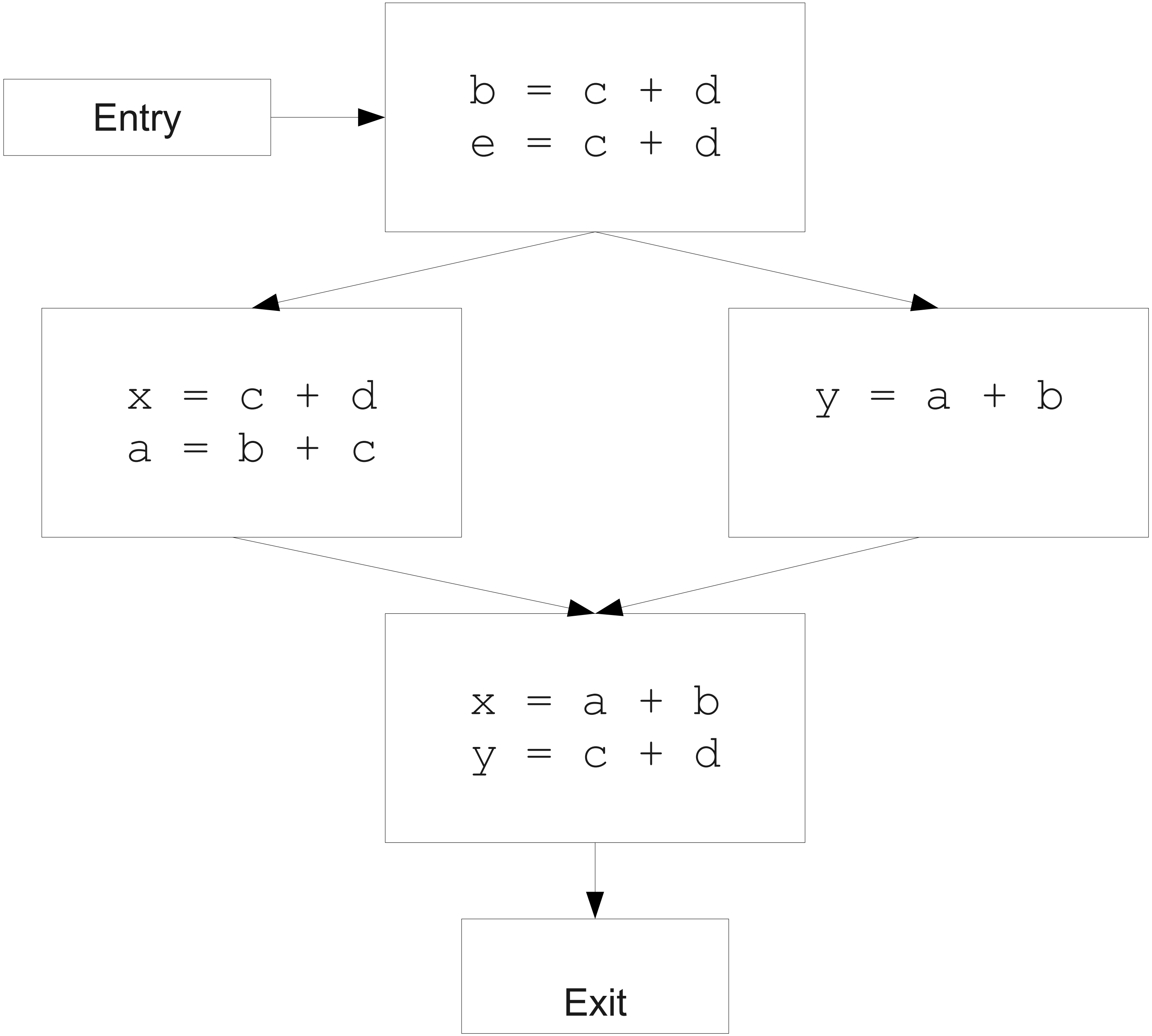
Global Dead Code Elimination

Global Dead Code 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



Entry

```
b = c + d
e = c + d
```

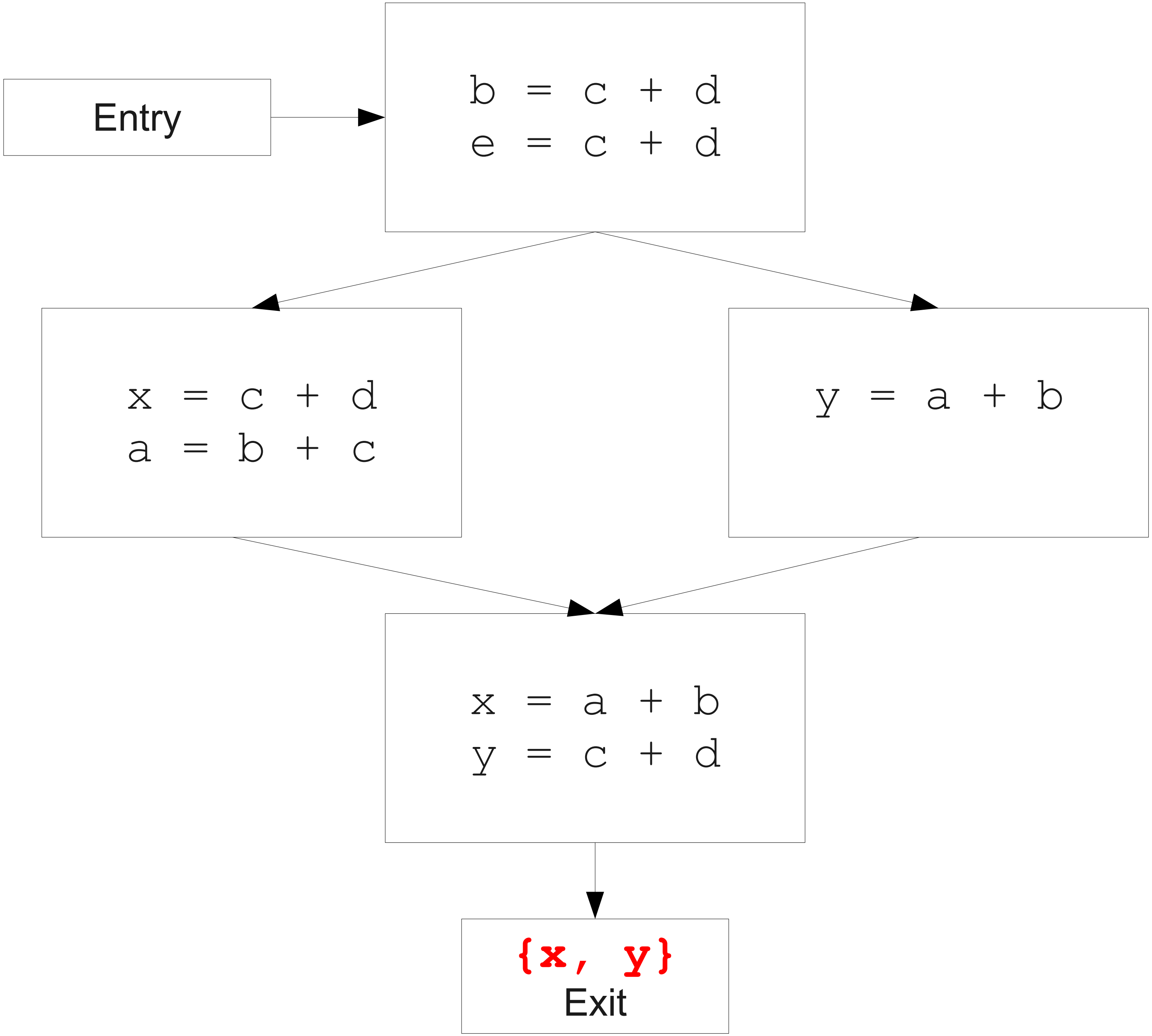
```
x = c + d
a = b + c
```

```
y = a + b
```

```
x = a + b
y = c + d
```

Exit

CFGs Without Loops



Entry

```
b = c + d
e = c + d
```

```
x = c + d
a = b + c
```

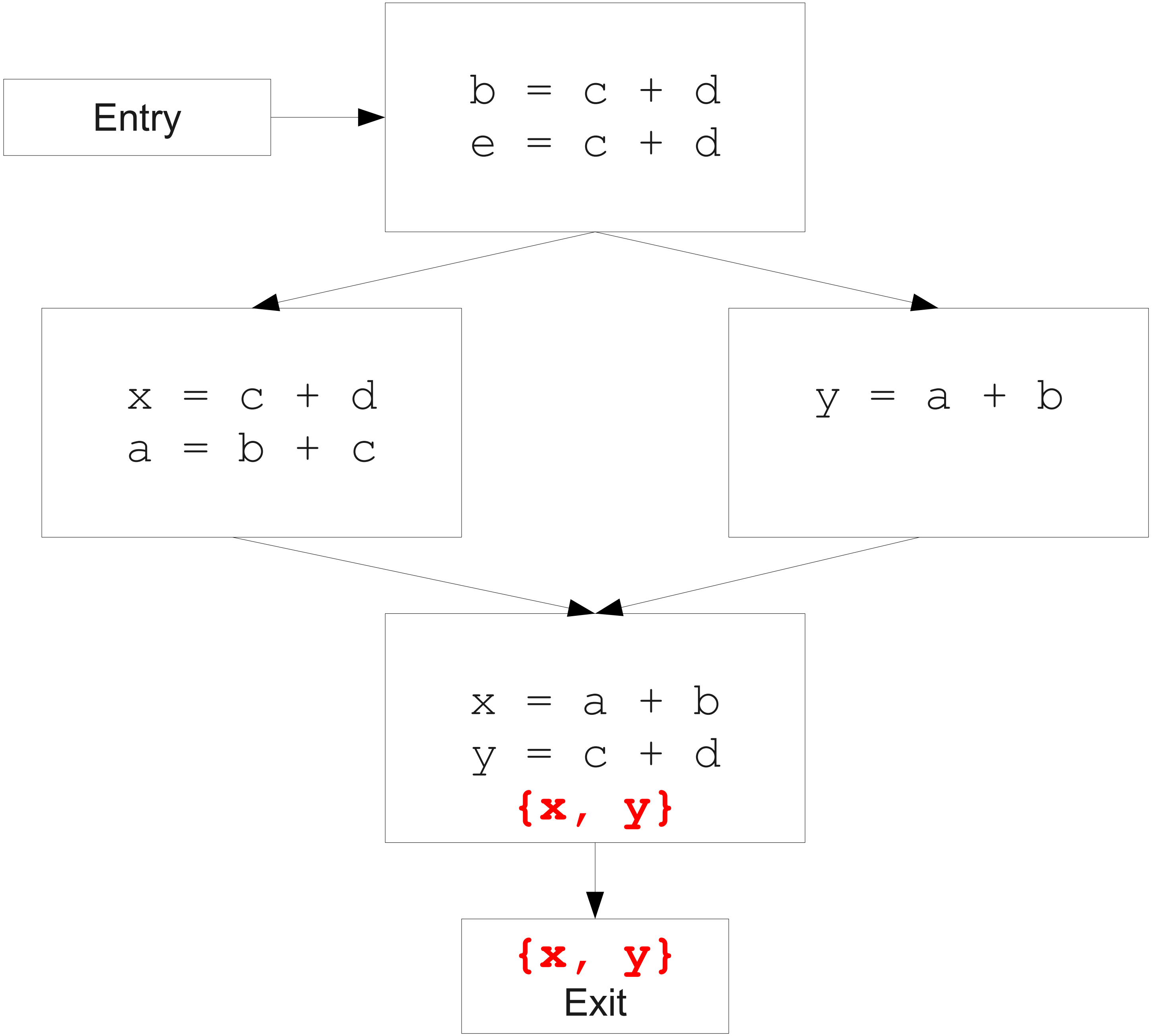
```
y = a + b
```

```
x = a + b
y = c + d
```

{x, y}

Exit

CFGs Without Loops



Entry

```
b = c + d
e = c + d
```

```
x = c + d
a = b + c
```

```
y = a + b
```

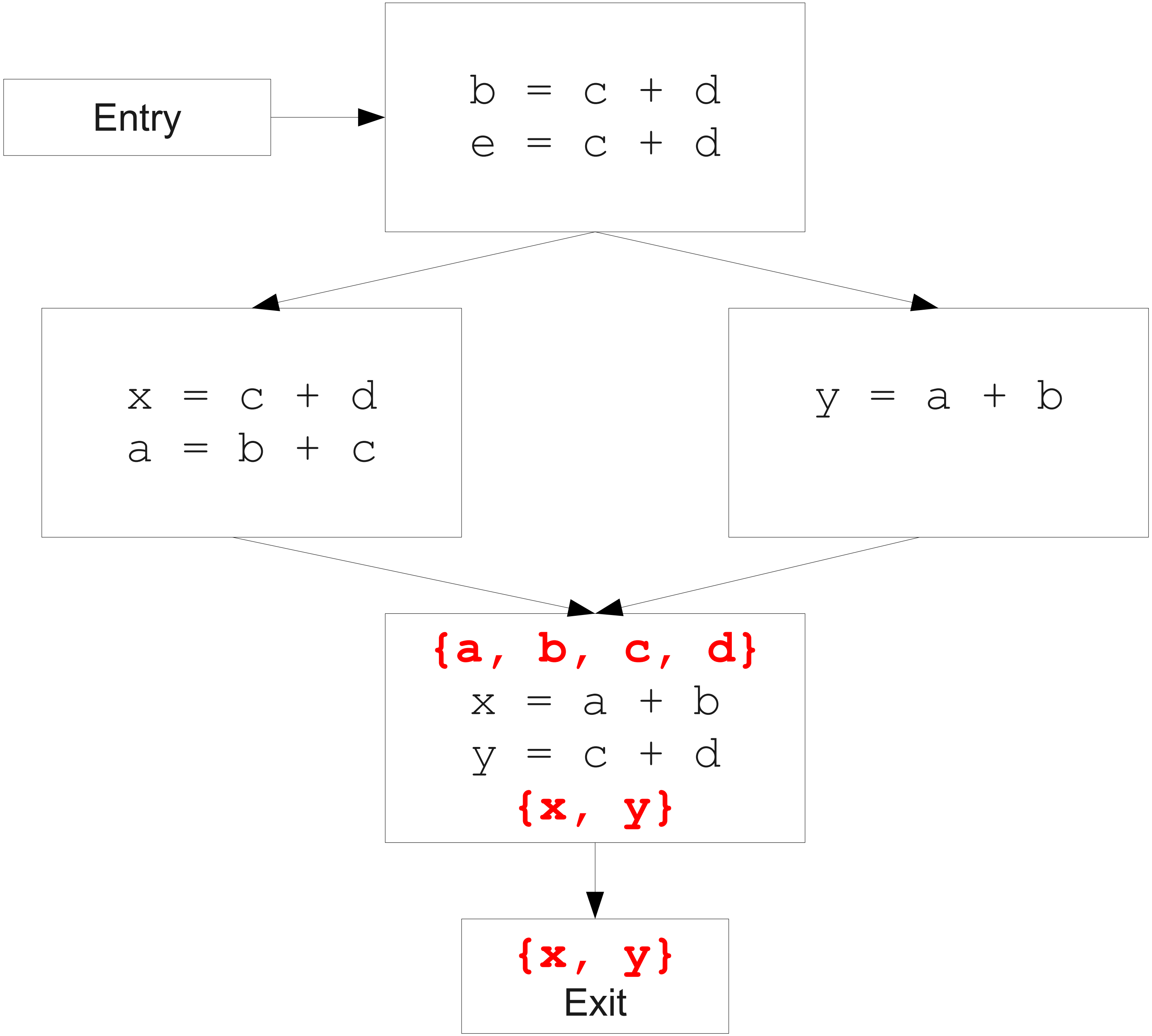
```
x = a + b
y = c + d
```

{x, y}

{x, y}

Exit

CFGs Without Loops



Entry

```
b = c + d
e = c + d
```

```
x = c + d
a = b + c
```

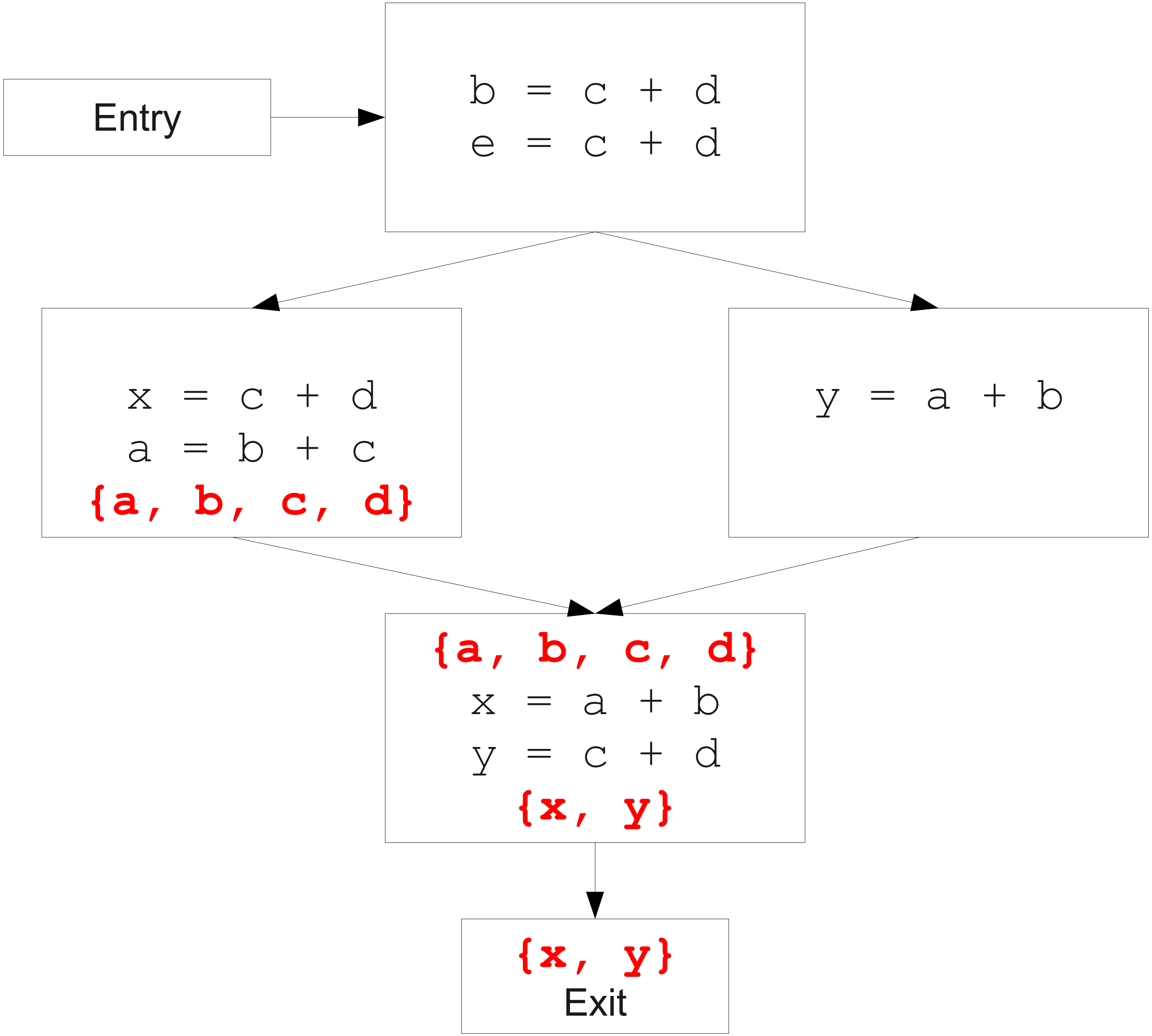
```
y = a + b
```

```
{a, b, c, d}
x = a + b
y = c + d
{x, y}
```

```
{x, y}
```

Exit

CFGs Without Loops



Entry

b = c + d
e = c + d

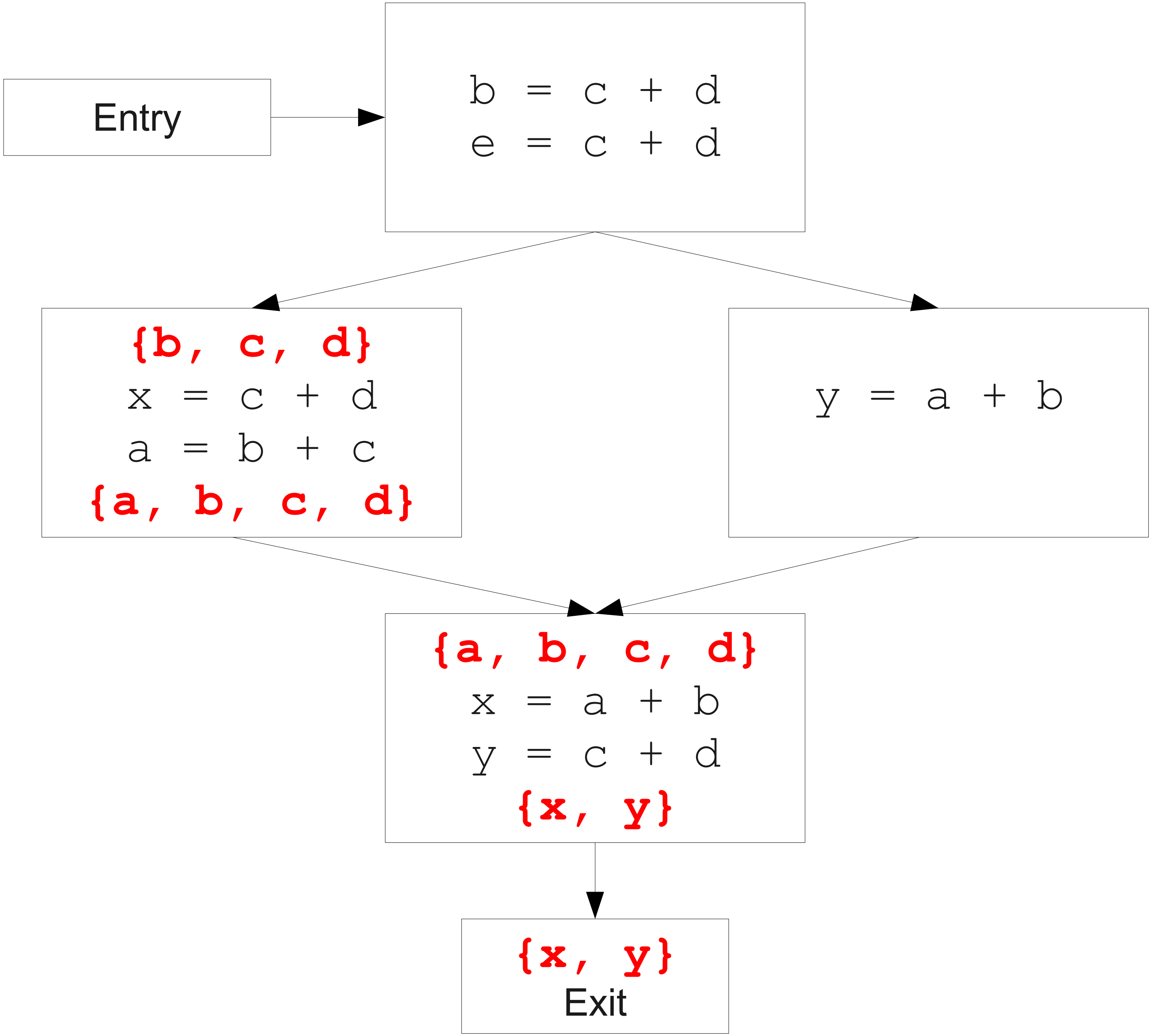
x = c + d
a = b + c
{a, b, c, d}

y = a + b

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

{x, y}
Exit

CFGs Without Loops



Entry

```
b = c + d
e = c + d
```

{b, c, d}

```
x = c + d
a = b + c
```

{a, b, c, d}

```
y = a + b
```

{a, b, c, d}

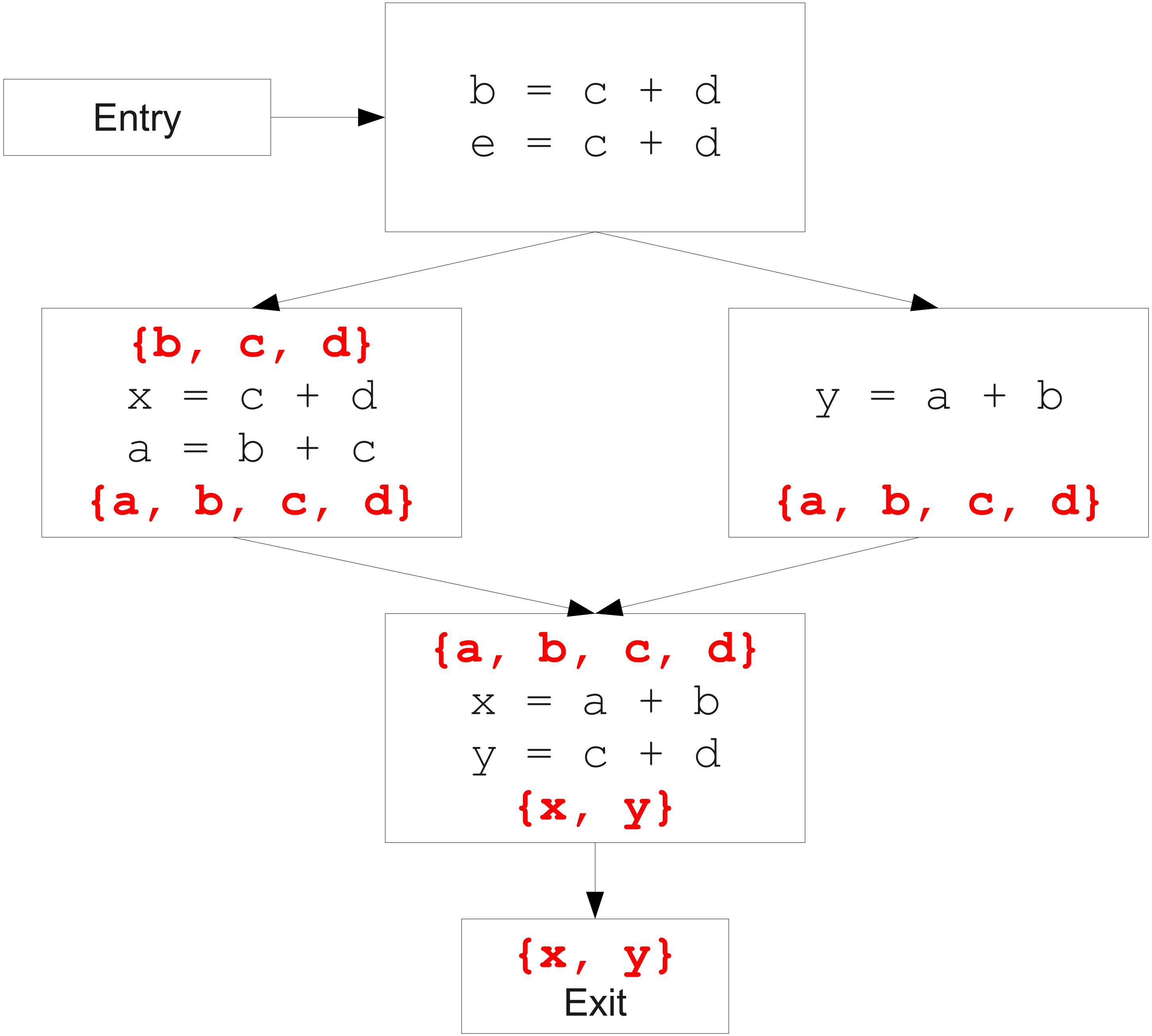
```
x = a + b
y = c + d
```

{x, y}

{x, y}

Exit

CFGs Without Loops



Entry

```
b = c + d
e = c + d
```

{b, c, d}

```
x = c + d
a = b + c
```

{a, b, c, d}

```
y = a + b
```

{a, b, c, d}

{a, b, c, d}

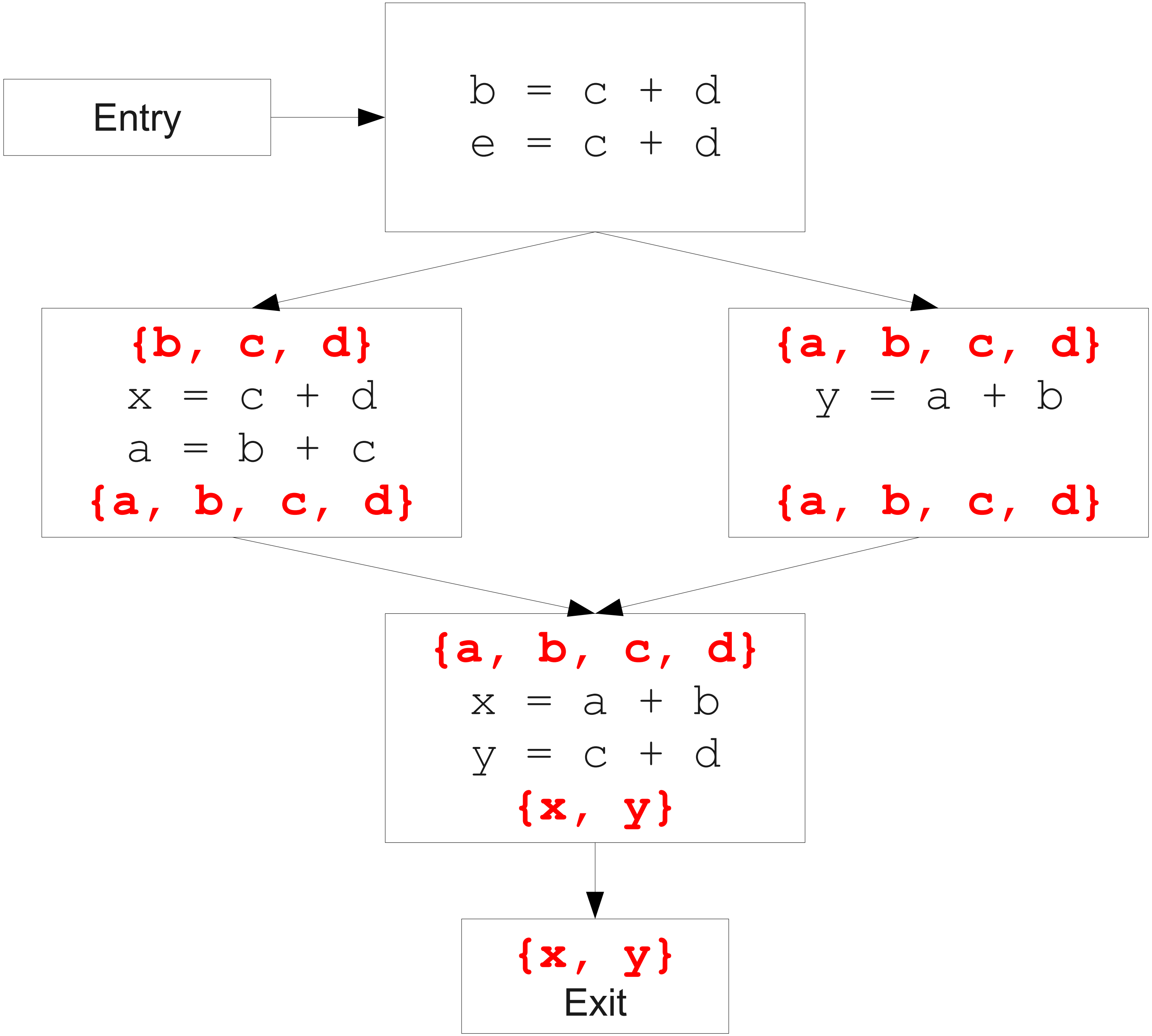
```
x = a + b
y = c + d
```

{x, y}

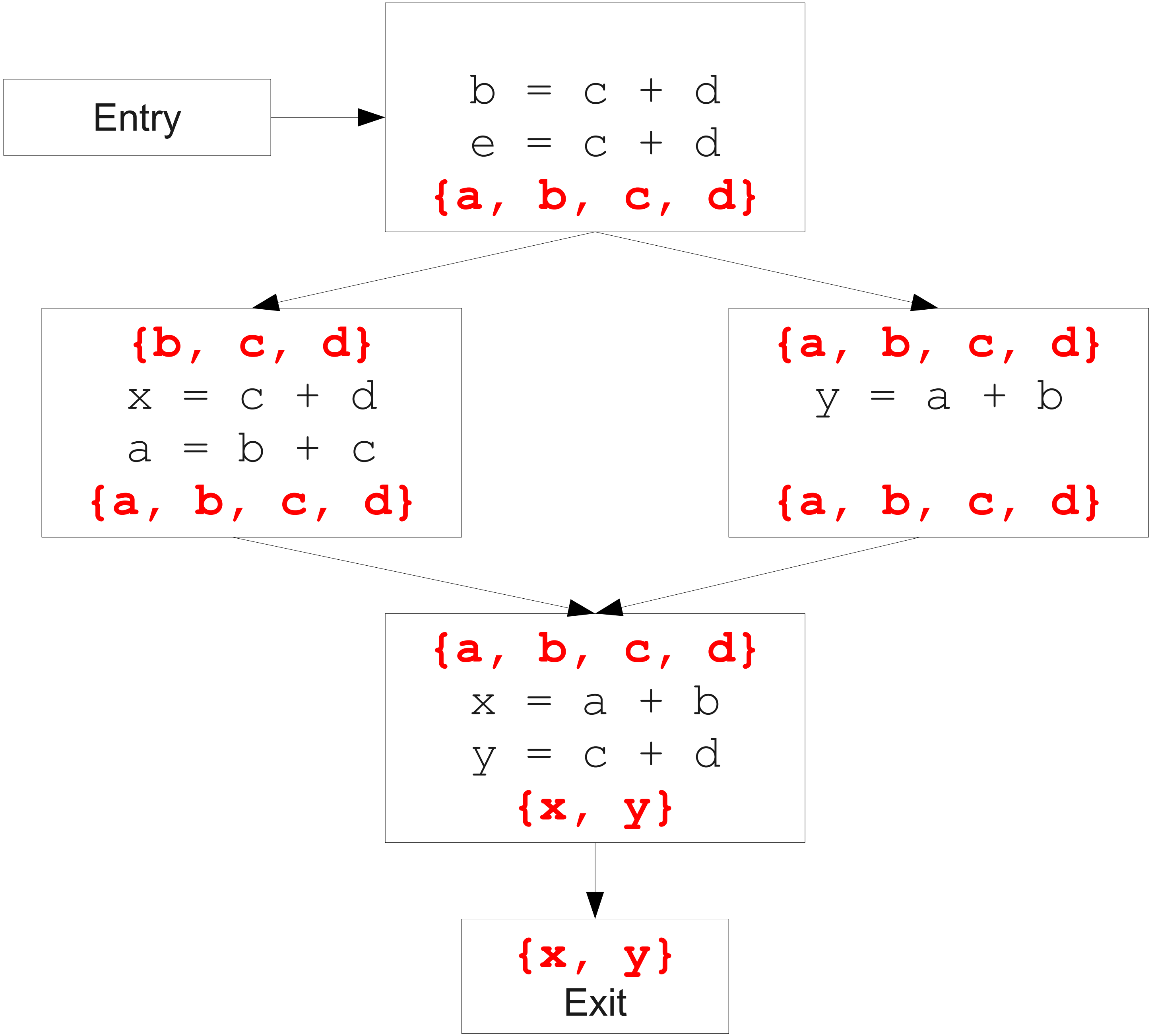
{x, y}

Exit

CFGs Without Loops



CFGs Without Loops



Entry

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

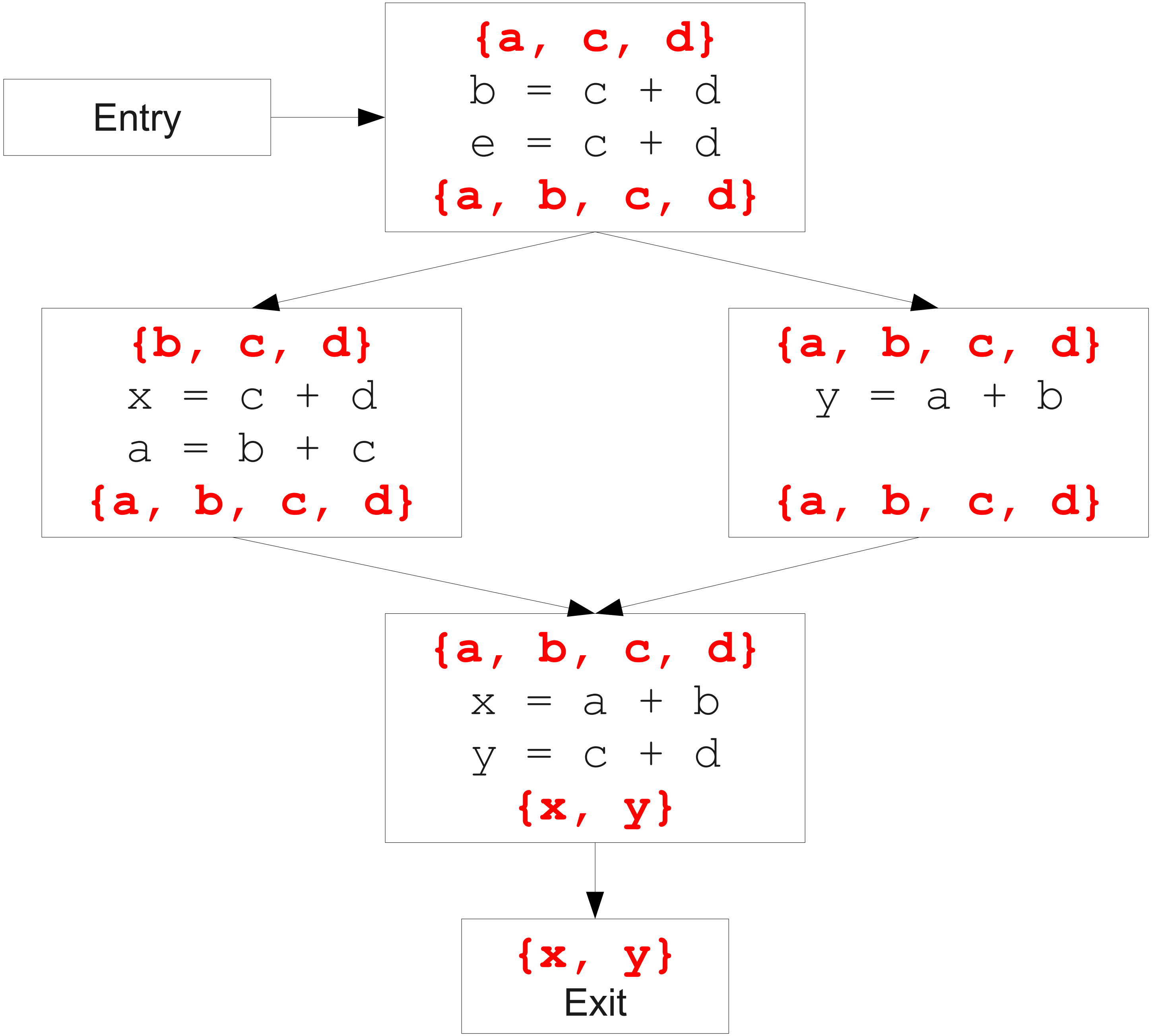
{b, c, d}
x = c + d
a = b + c
{a, b, c, d}

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

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

{x, y}
Exit

CFGs Without Loops



Entry

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

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

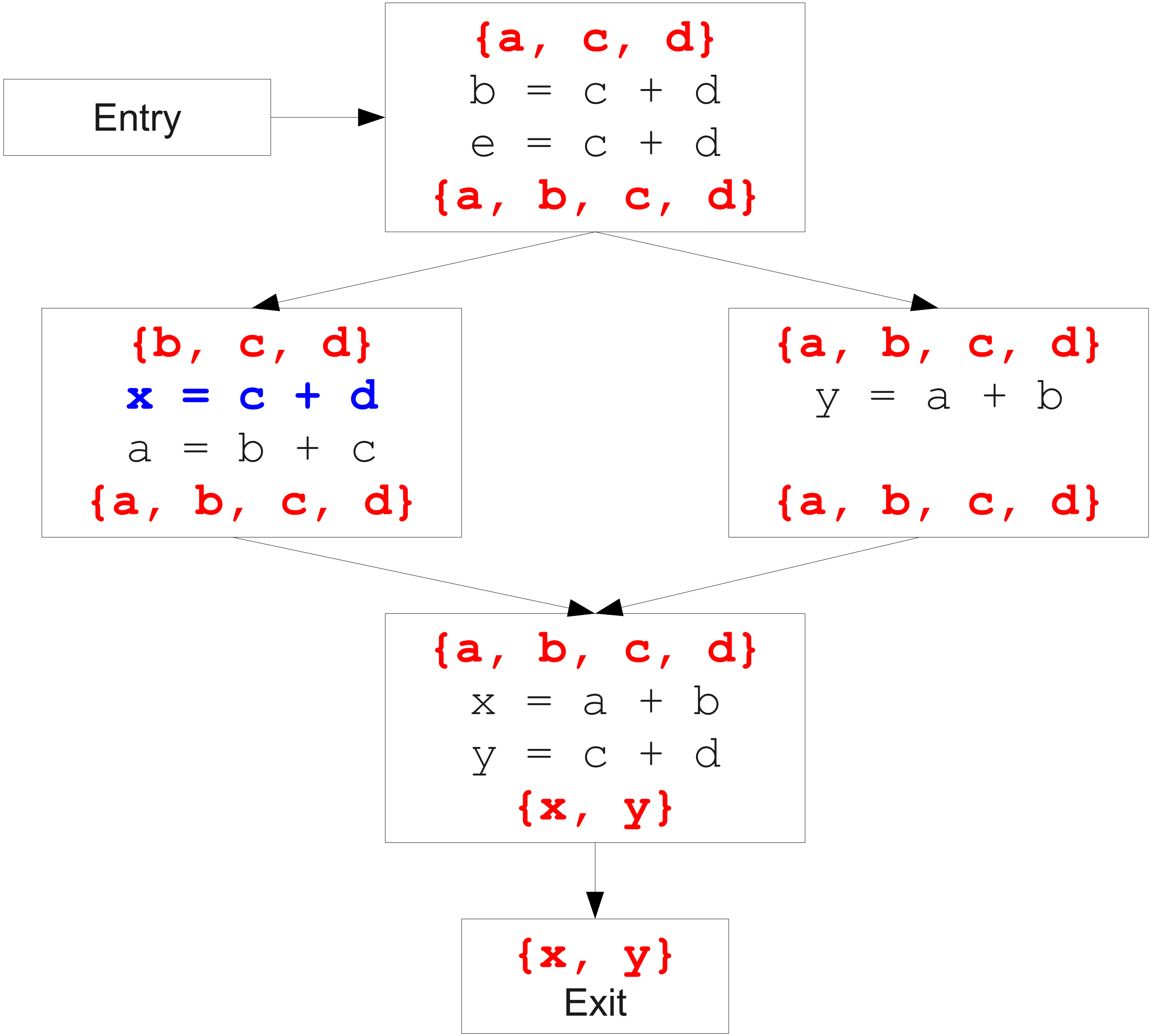
{a, b, c, d}
y = a + b

{a, b, c, d}

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

{x, y}
Exit

CFGs Without Loops



Entry

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

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

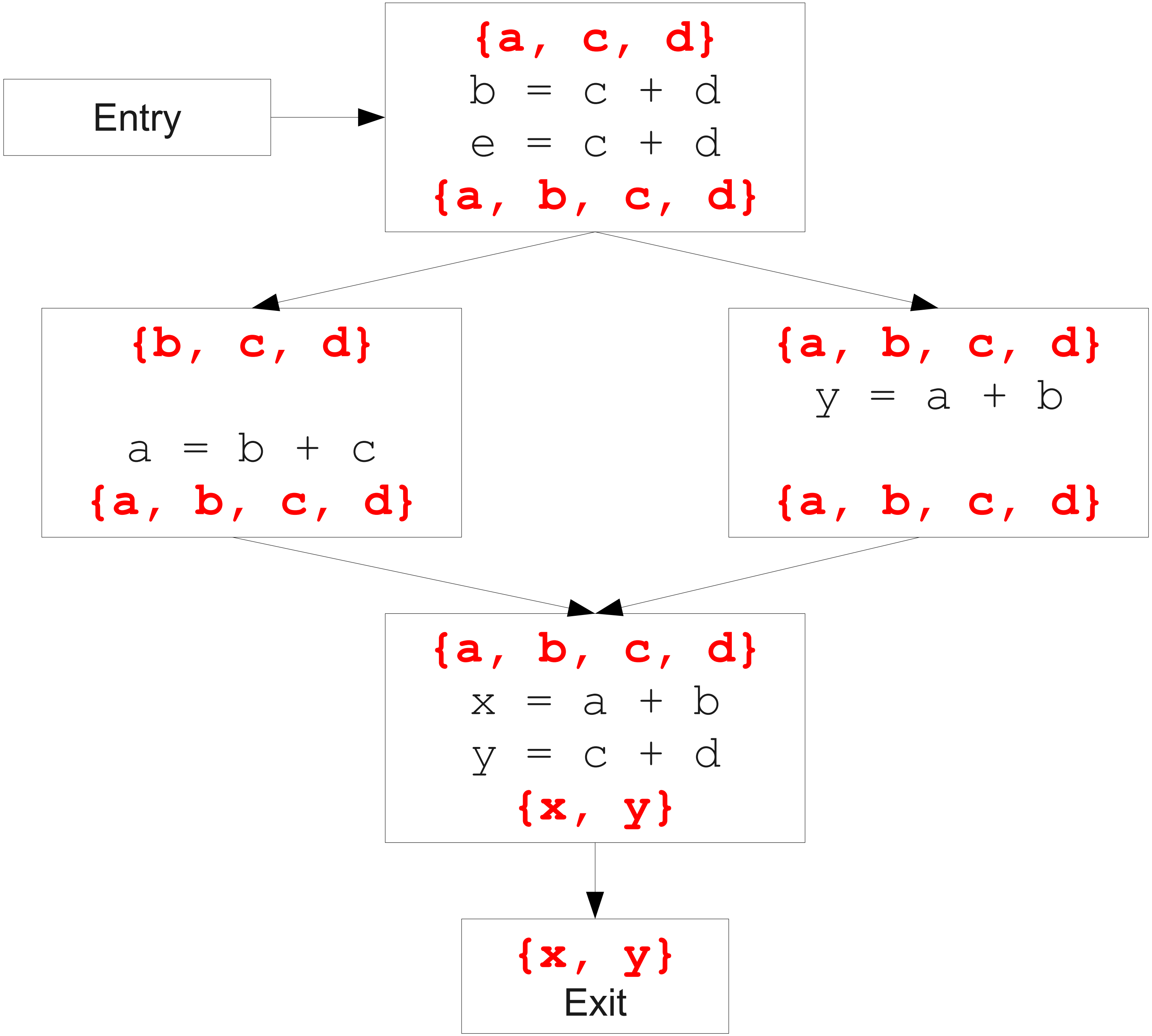
{a, b, c, d}
y = a + b

{a, b, c, d}

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

{x, y}
Exit

CFGs Without Loops



Entry

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

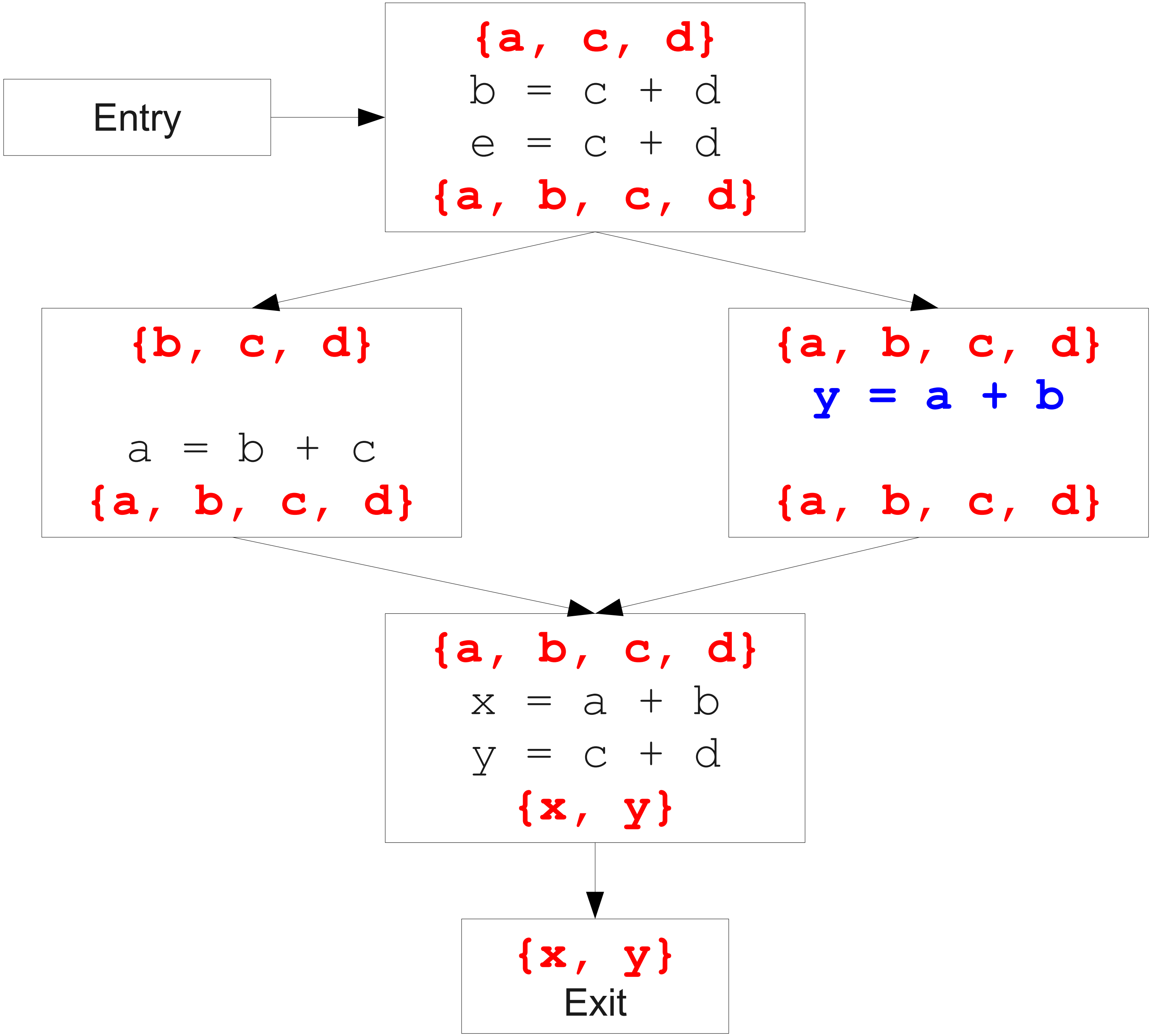
{b, c, d}
a = b + c
{a, b, c, d}

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

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

{x, y}
Exit

CFGs Without Loops



Entry

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

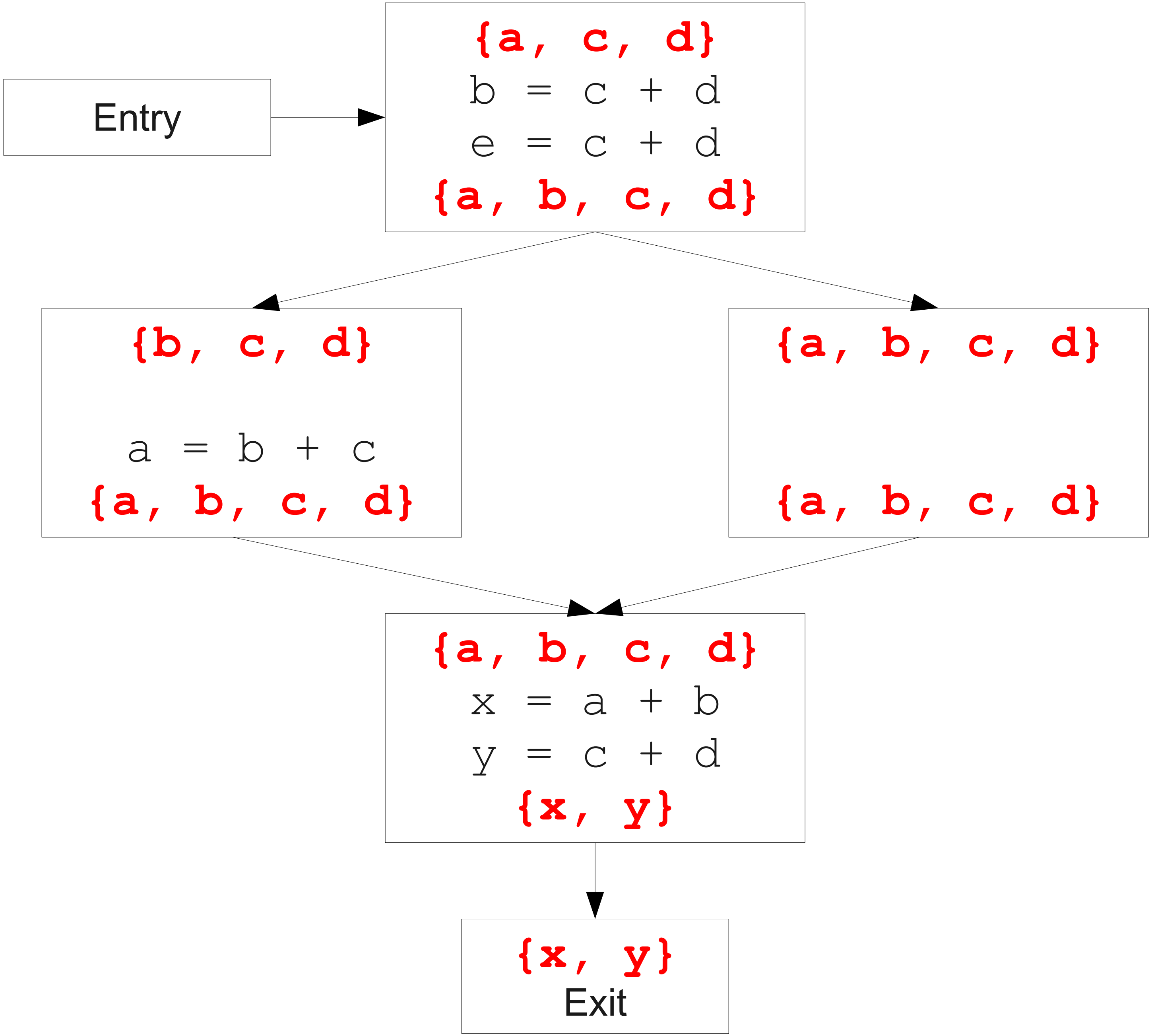
{b, c, d}
a = b + c
{a, b, c, d}

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

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

{x, y}
Exit

CFGs Without Loops



Entry

{a, c, d}

b = c + d

e = c + d

{a, b, c, d}

{b, c, d}

a = b + c

{a, b, c, d}

{a, b, c, d}

{a, b, c, d}

{a, b, c, d}

x = a + b

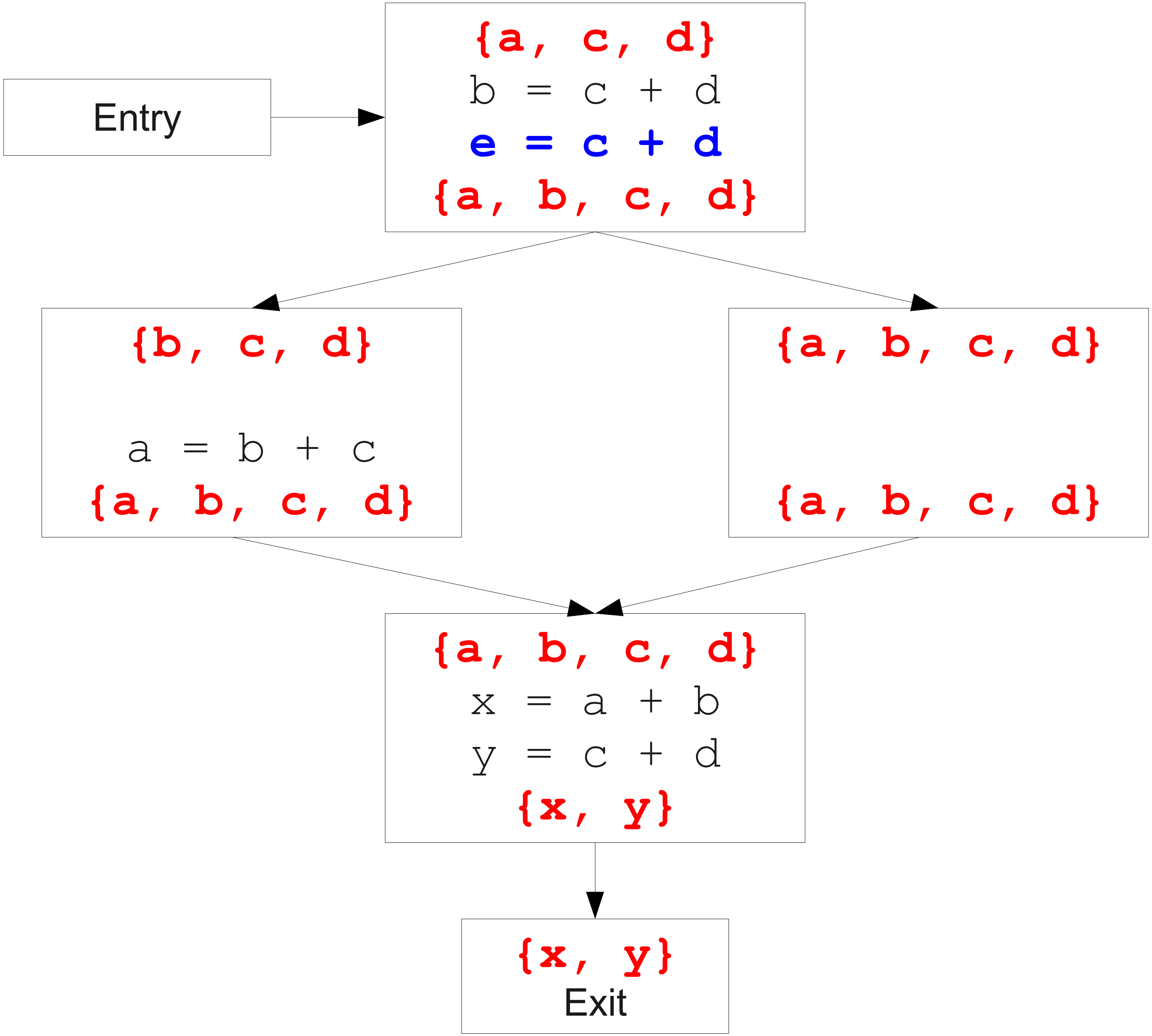
y = c + d

{x, y}

{x, y}

Exit

CFGs Without Loops



Entry

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

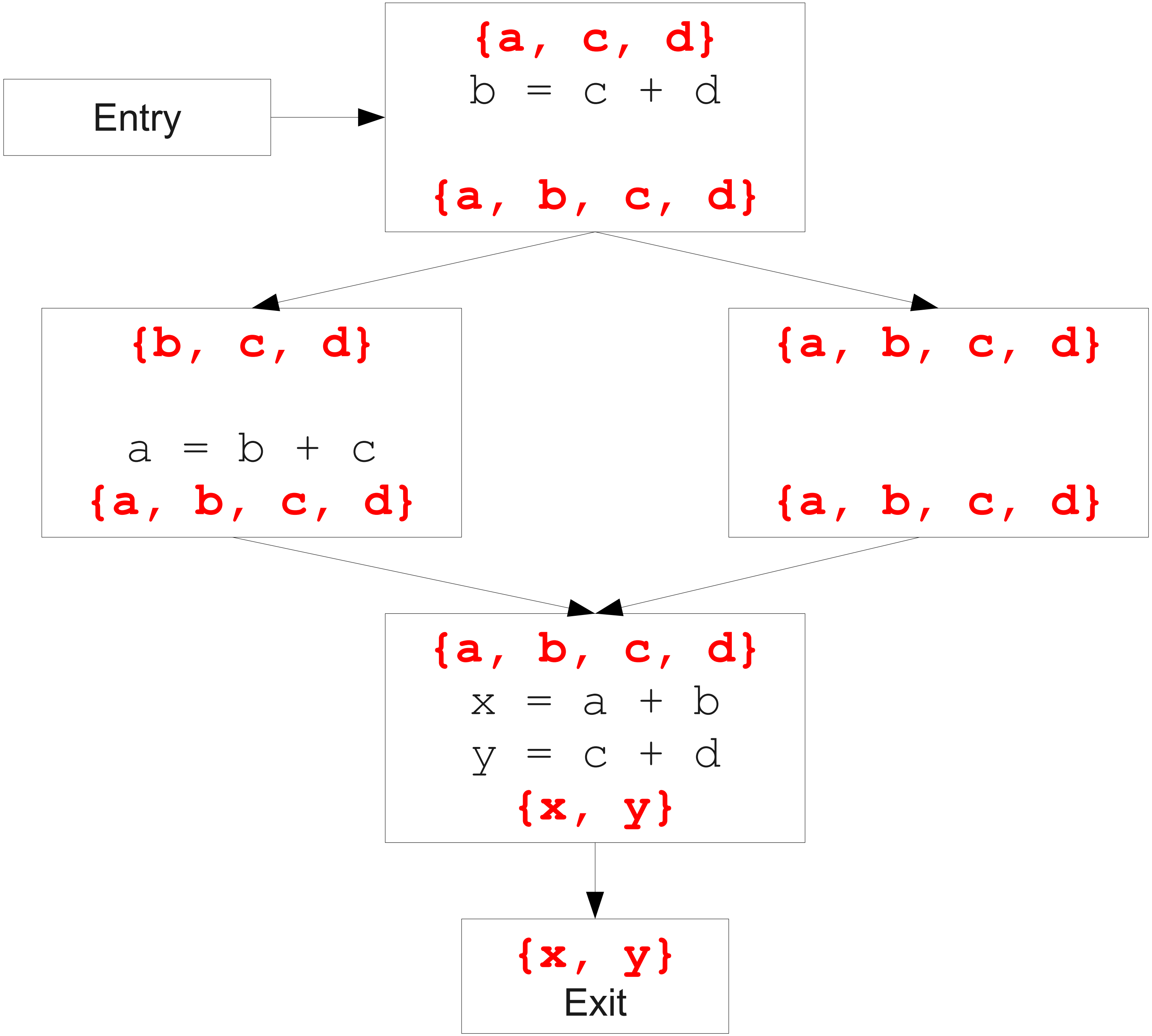
{b, c, d}
a = b + c
{a, b, c, d}

{a, b, c, d}
{a, b, c, d}

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

{x, y}
Exit

CFGs Without Loops



Entry

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

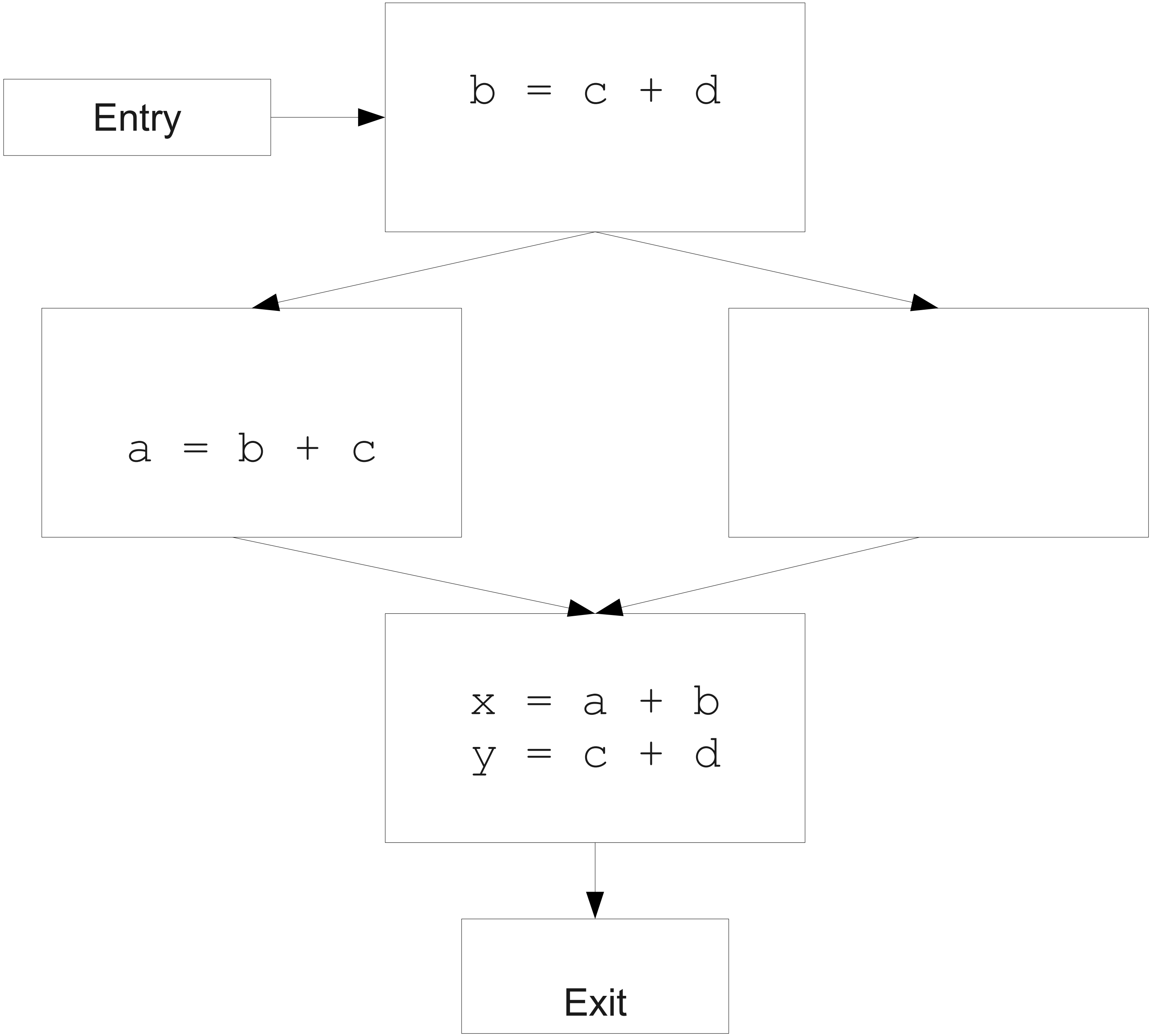
{b, c, d}
a = b + c
{a, b, c, d}

{a, b, c, d}
{a, b, c, d}

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

{x, y}
Exit

CFGs Without Loops



Entry

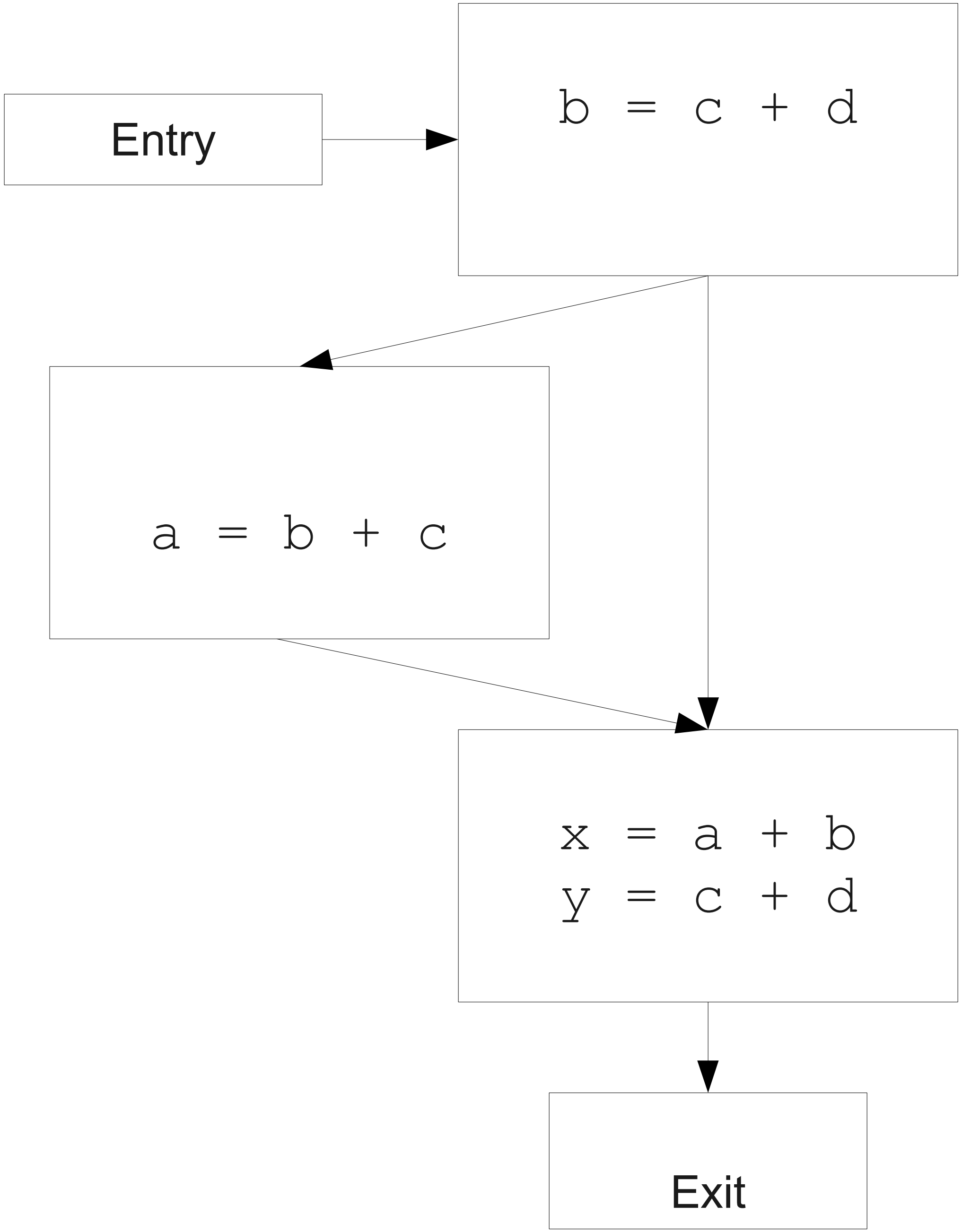
```
b = c + d
```

```
a = b + c
```

```
x = a + b
y = c + d
```

Exit

CFGs Without Loops



Entry

b = c + d

a = b + c

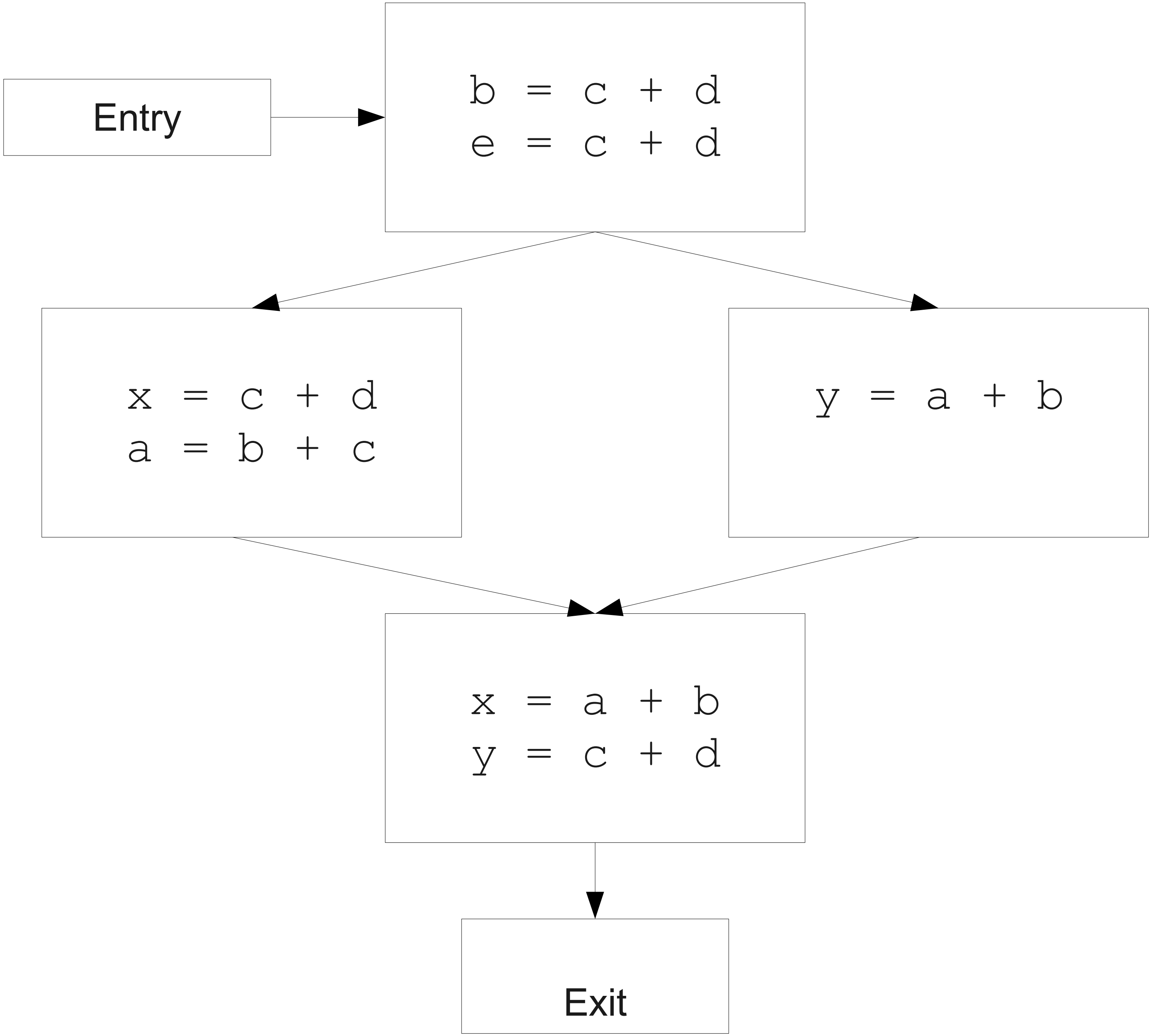
x = a + b
y = c + d

Exit

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



Entry

```
b = c + d
e = c + d
```

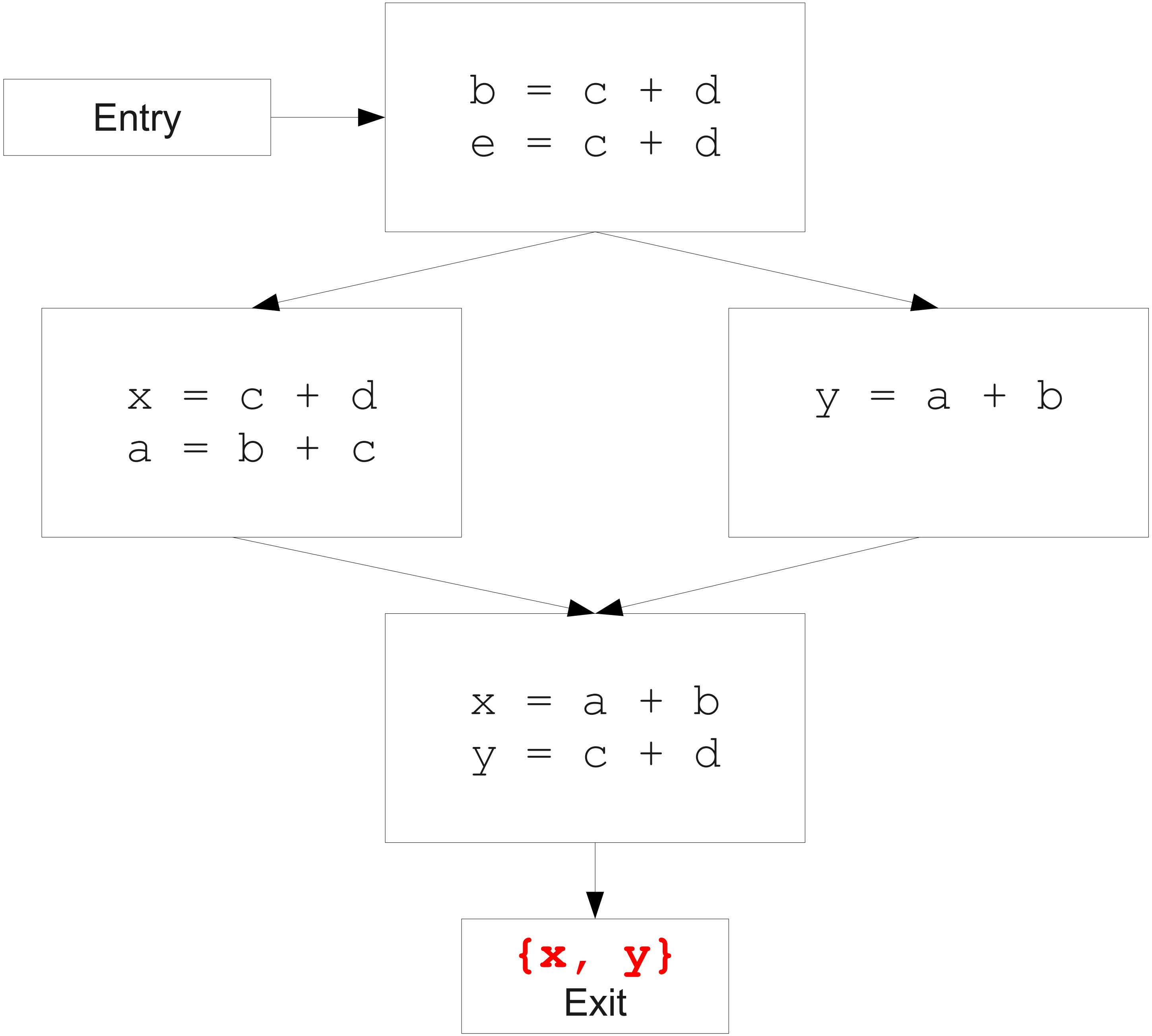
```
x = c + d
a = b + c
```

```
y = a + b
```

```
x = a + b
y = c + d
```

Exit

CFGs Without Loops



Entry

```
b = c + d
e = c + d
```

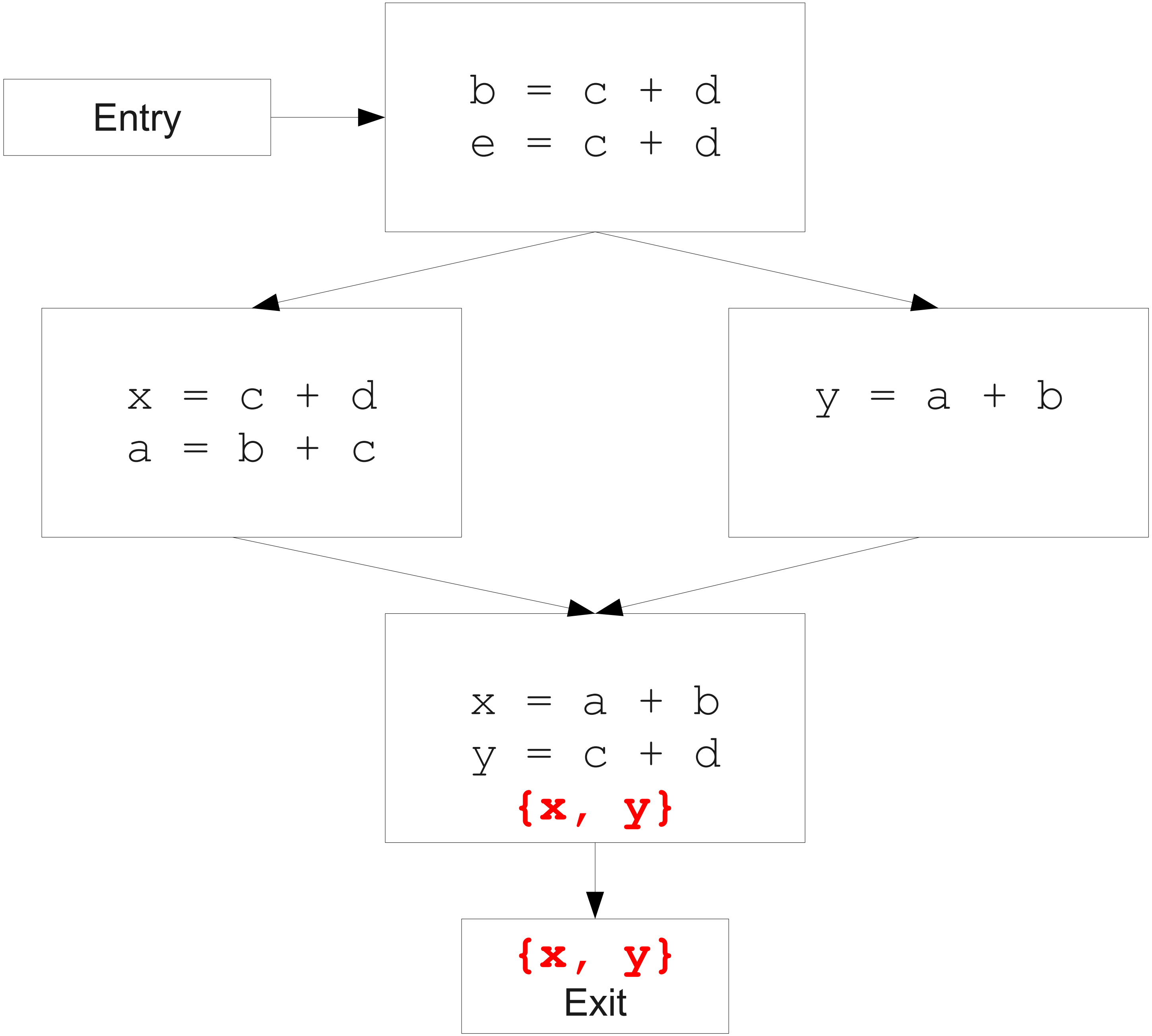
```
x = c + d
a = b + c
```

```
y = a + b
```

```
x = a + b
y = c + d
```

{x, y}
Exit

CFGs Without Loops



Entry

```
b = c + d
e = c + d
```

```
x = c + d
a = b + c
```

```
y = a + b
```

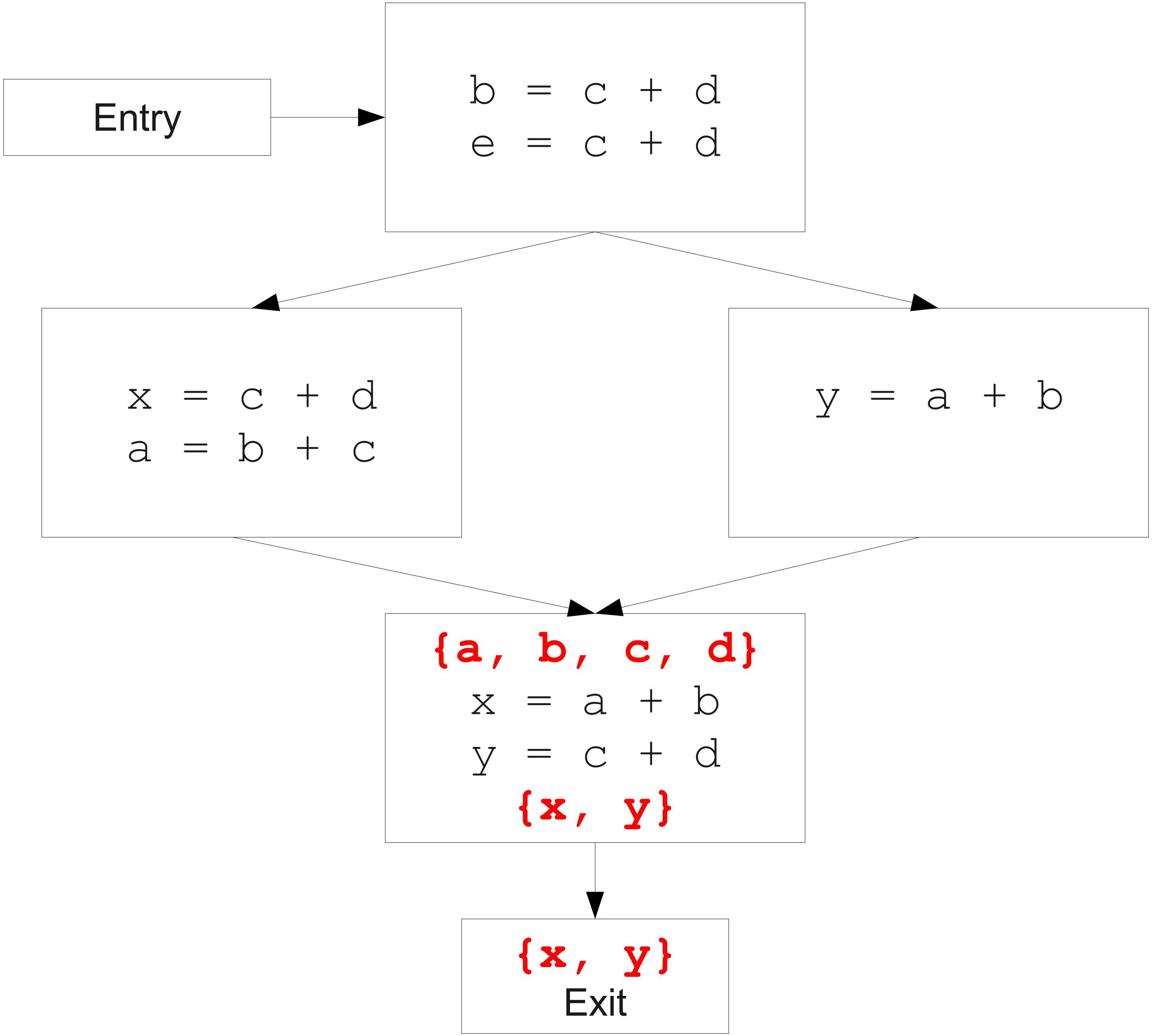
```
x = a + b
y = c + d
```

{x, y}

{x, y}

Exit

CFGs Without Loops



Entry

```
b = c + d
e = c + d
```

```
x = c + d
a = b + c
```

```
y = a + b
```

{a, b, c, d}

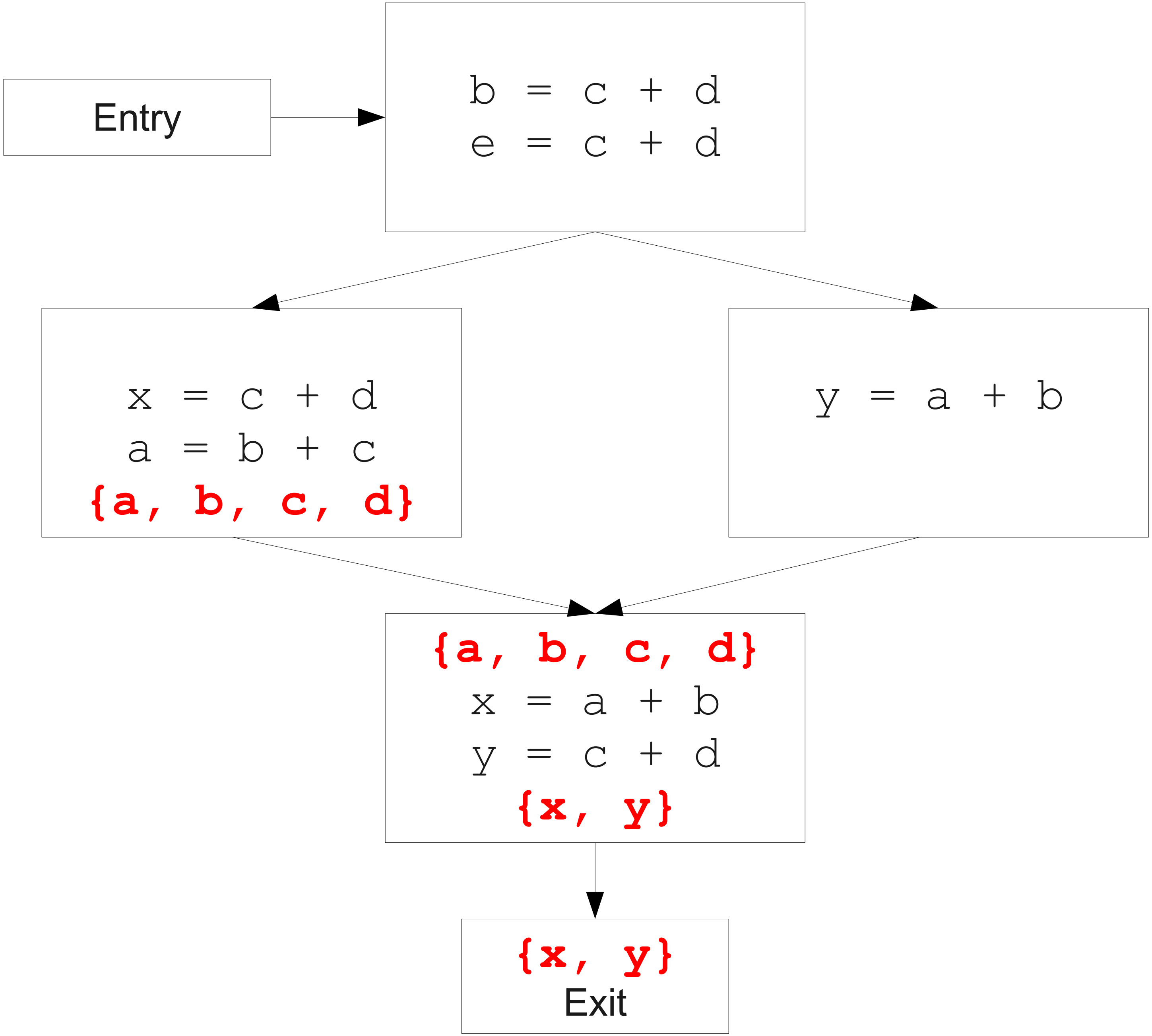
```
x = a + b
y = c + d
```

{x, y}

{x, y}

Exit

CFGs Without Loops



Entry

```
b = c + d
e = c + d
```

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

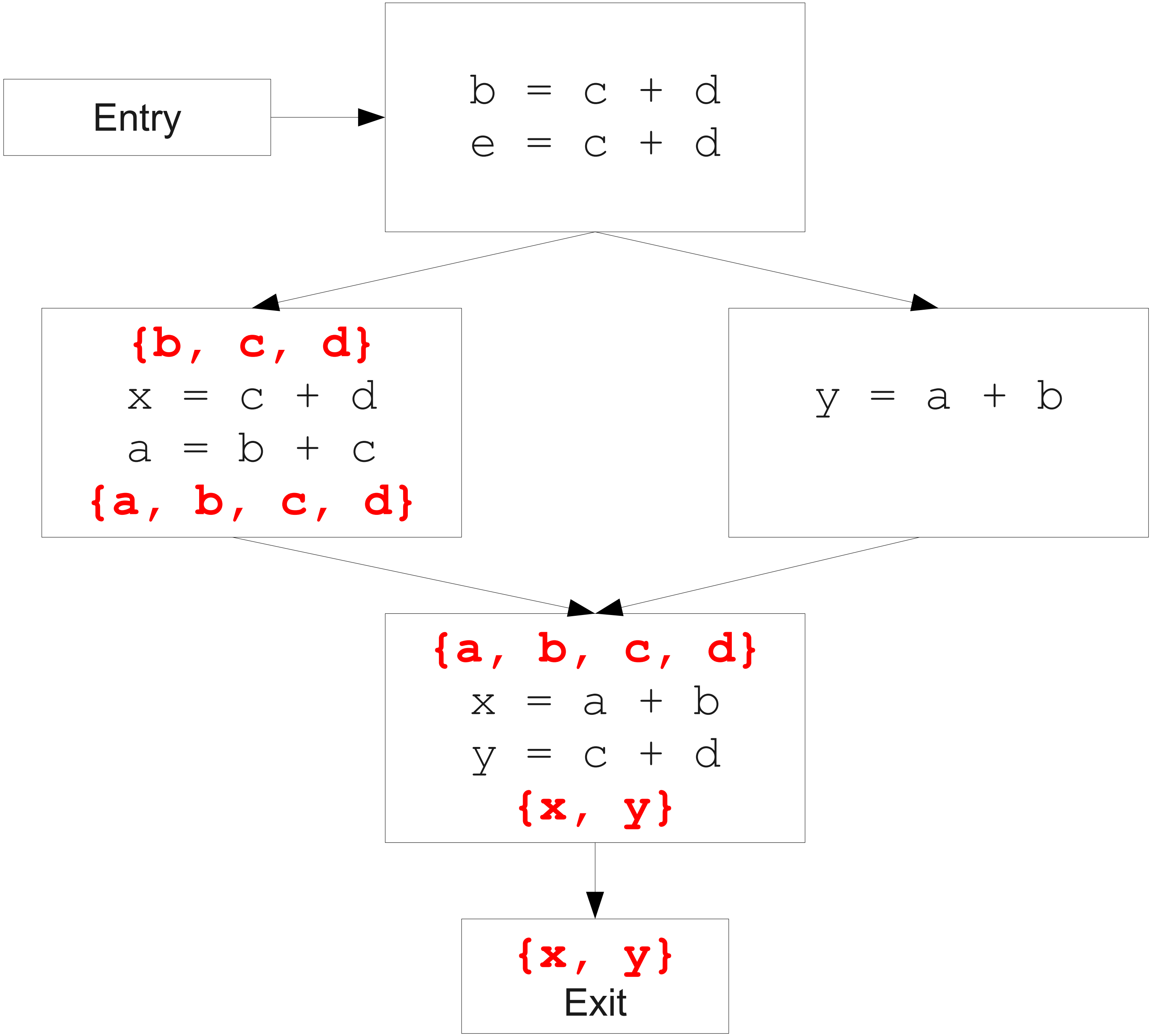
```
y = a + b
```

```
{a, b, c, d}
x = a + b
y = c + d
{x, y}
```

```
{x, y}
```

Exit

CFGs Without Loops



Entry

b = c + d
e = c + d

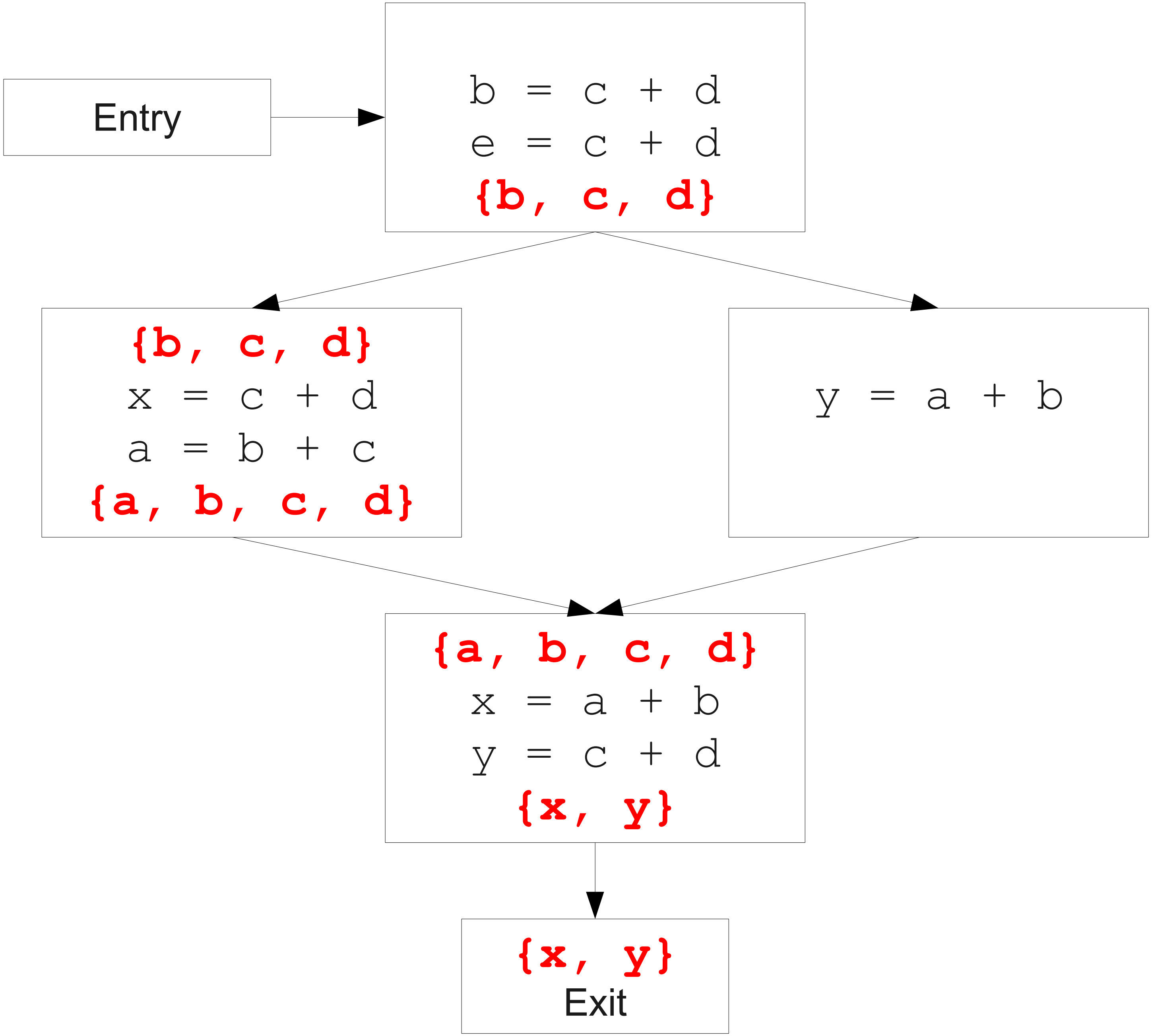
{b, c, d}
x = c + d
a = b + c
{a, b, c, d}

y = a + b

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

{x, y}
Exit

CFGs Without Loops



Entry

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

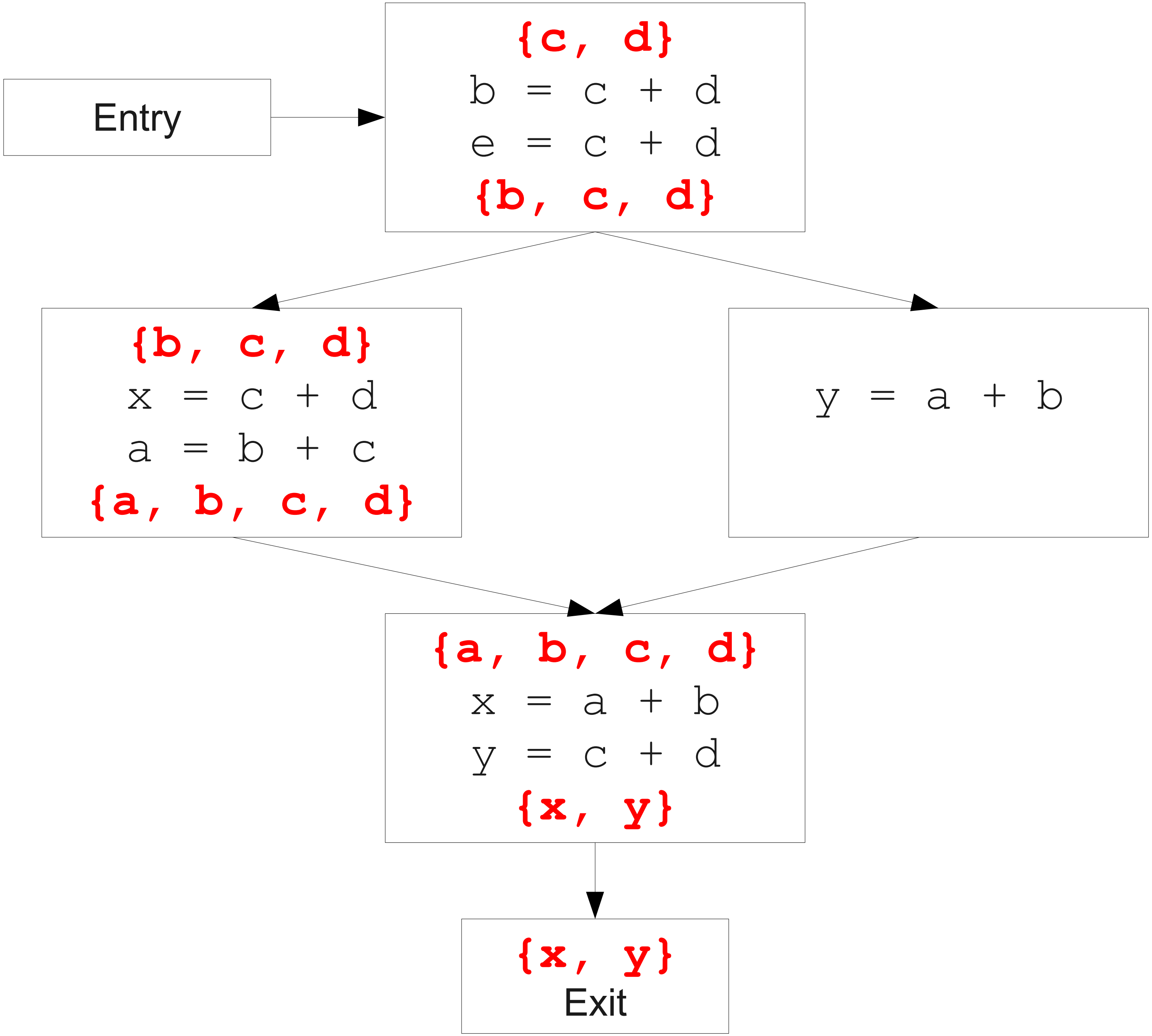
{b, c, d}
x = c + d
a = b + c
{a, b, c, d}

y = a + b

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

{x, y}
Exit

CFGs Without Loops



Entry

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

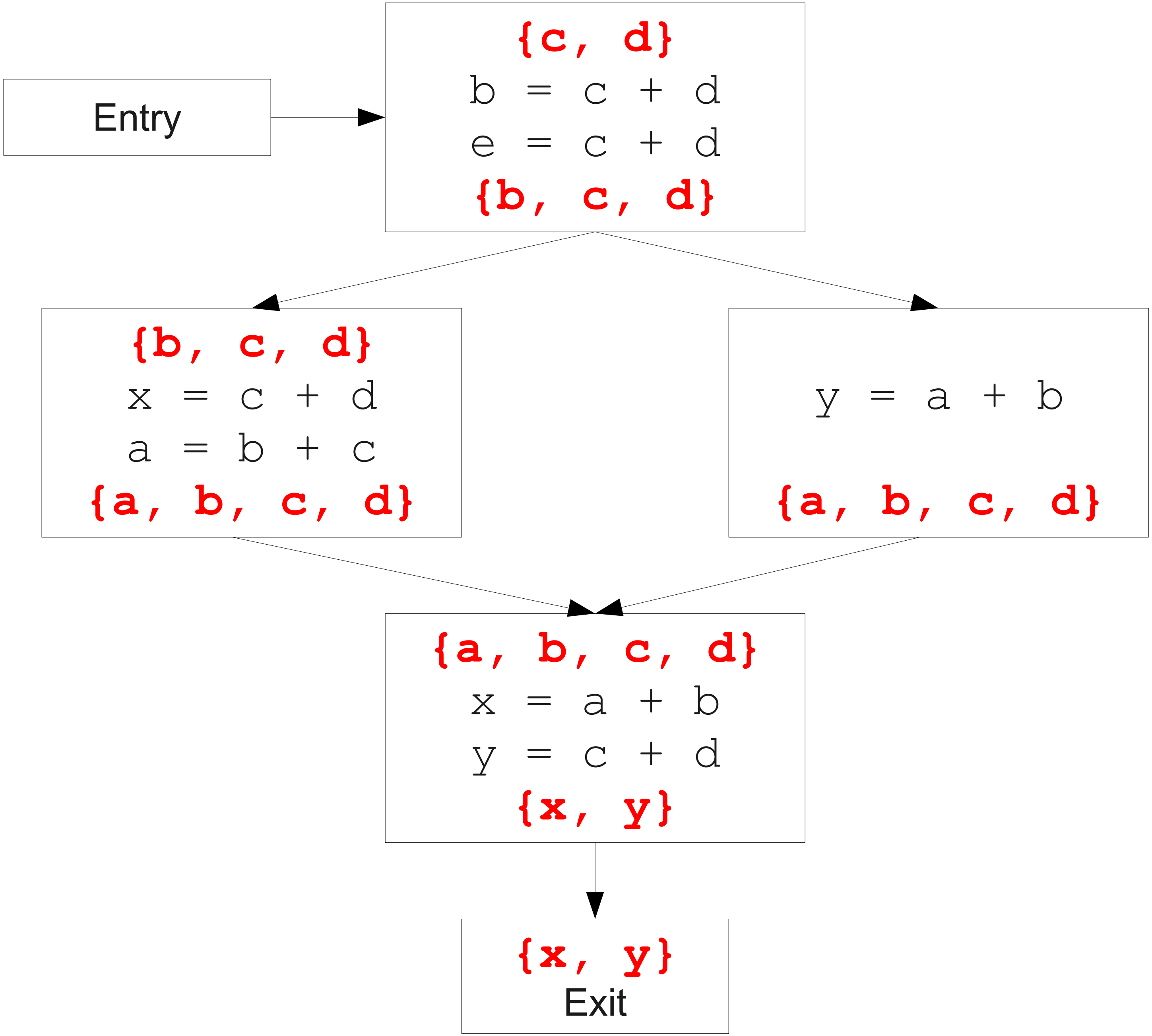
{b, c, d}
x = c + d
a = b + c
{a, b, c, d}

y = a + b

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

{x, y}
Exit

CFGs Without Loops



Entry

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

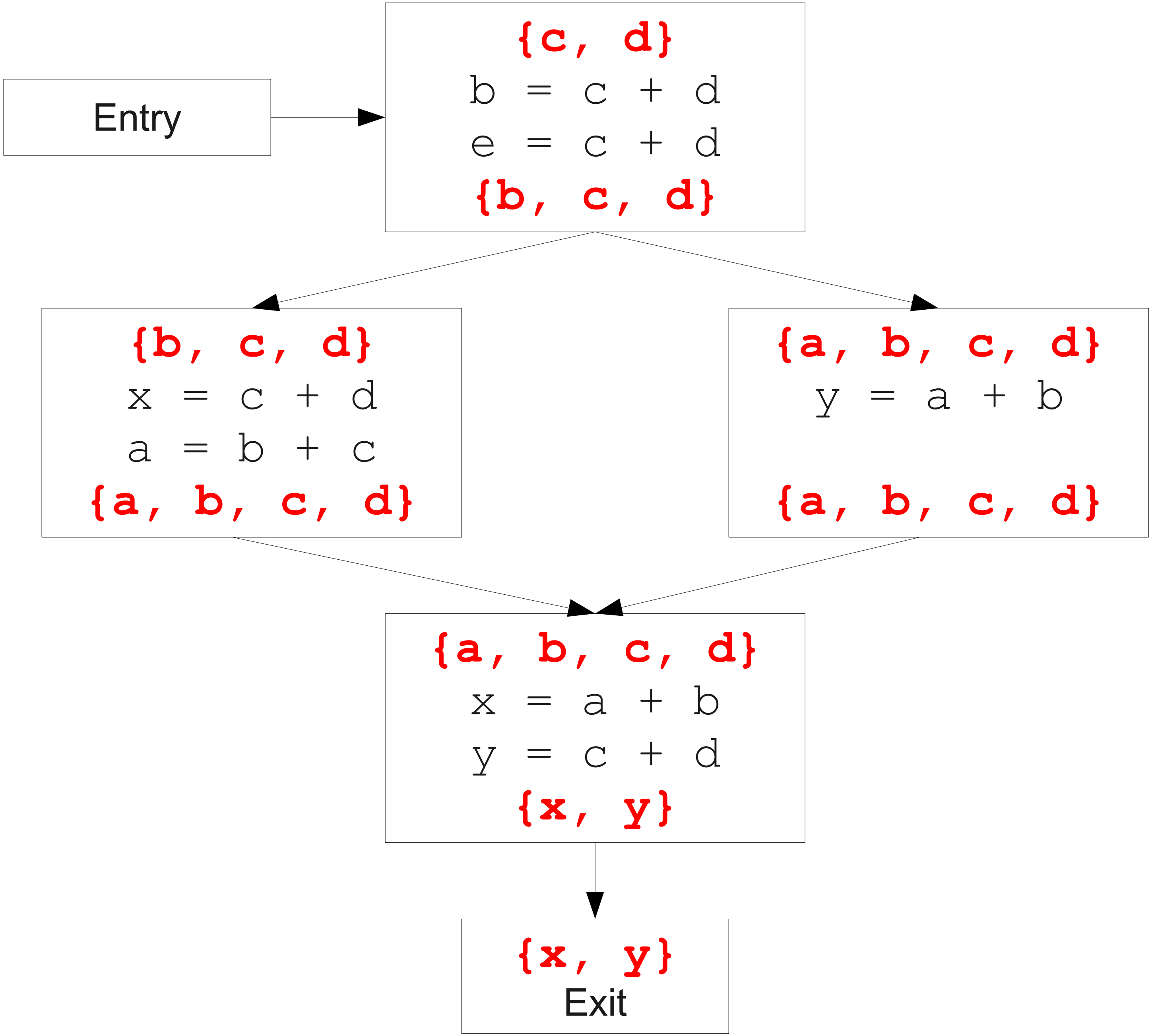
{b, c, d}
x = c + d
a = b + c
{a, b, c, d}

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

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

{x, y}
Exit

CFGs Without Loops



Entry

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

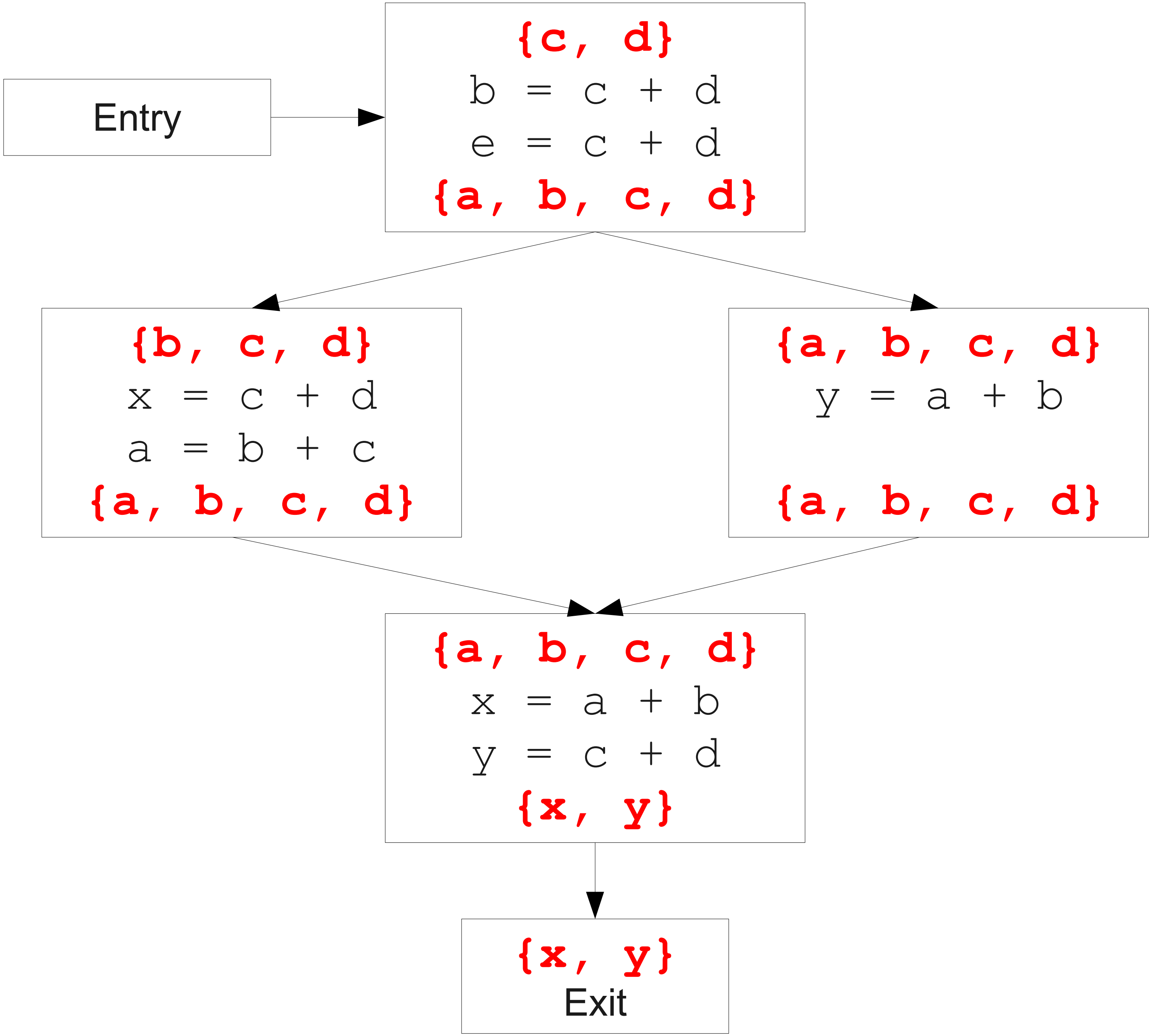
{b, c, d}
x = c + d
a = b + c
{a, b, c, d}

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

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

{x, y}
Exit

CFGs Without Loops



Entry

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

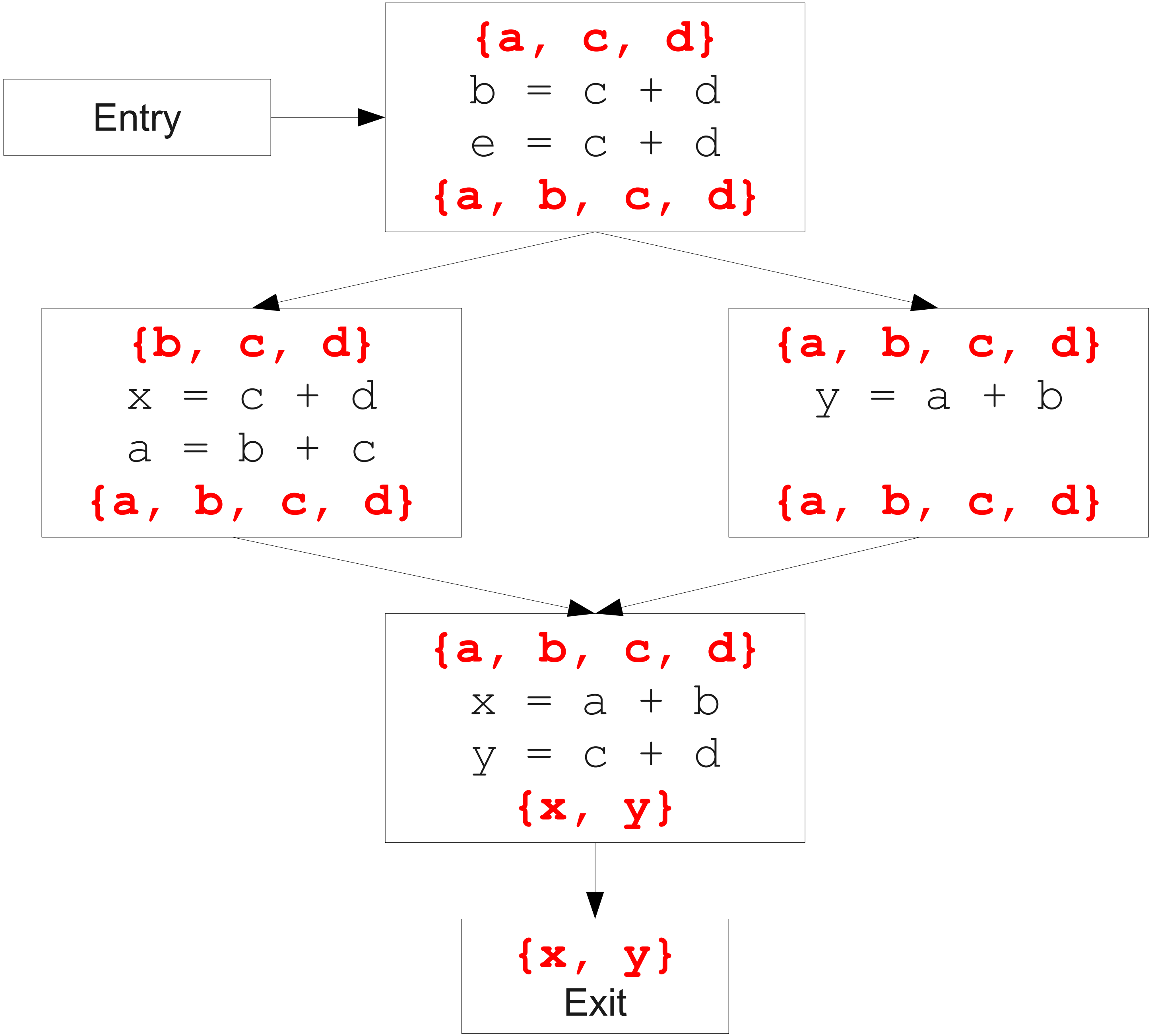
{b, c, d}
x = c + d
a = b + c
{a, b, c, d}

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

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

{x, y}
Exit

CFGs Without Loops



Entry

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

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

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

{a, b, c, d}

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

{x, y}
Exit

Major Changes, Part II

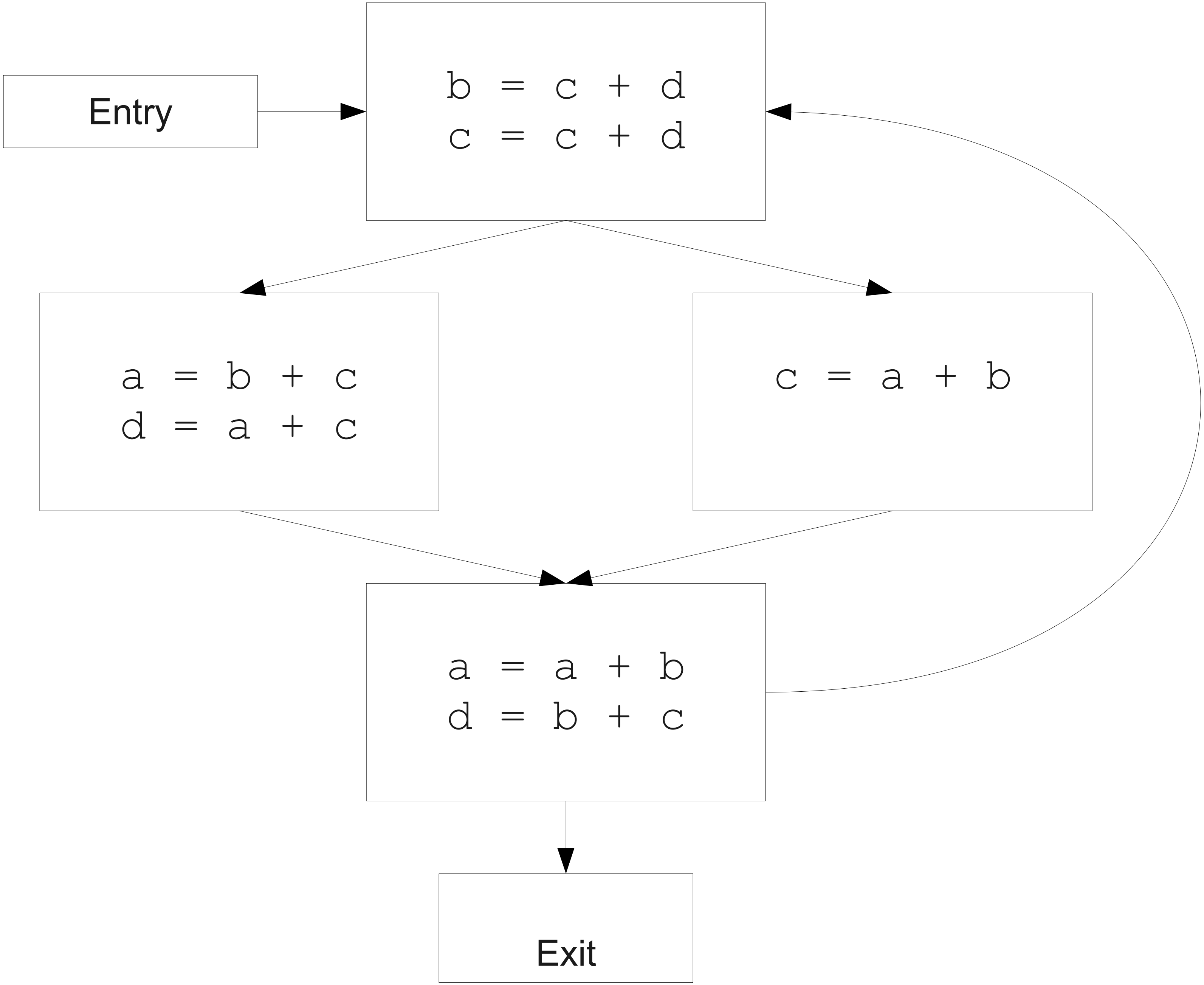
- In a local analysis, there is only one possible path through a basic block.
- In a global analysis, there may be **many** paths through a CFG.
- May need to recompute values multiple times as more information becomes available.
- Need to be careful when doing this not to loop infinitely!
  - (More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

CFGs with Loops

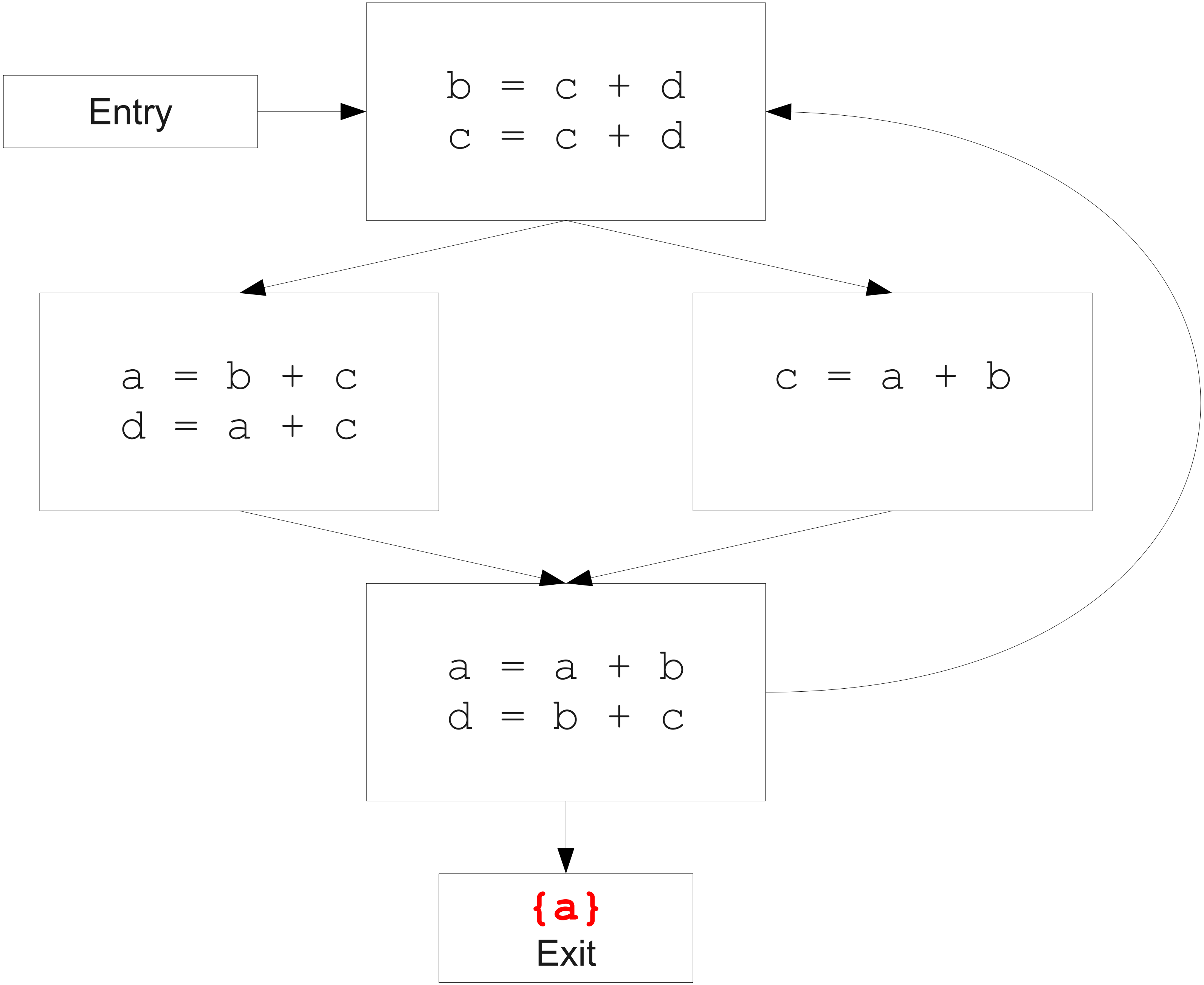
- Up to this point, we've considered loop-free CFGs, which have only finitely many possible paths.
- When we add loops into the picture, this is no longer true.
- Not all possible loops in a CFG can be realized in the actual program.
- **Sound approximation**: Assume that every possible path through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



Entry

```
b = c + d
c = c + d
```

```
a = b + c
d = a + c
```

```
c = a + b
```

```
a = a + b
d = b + c
```

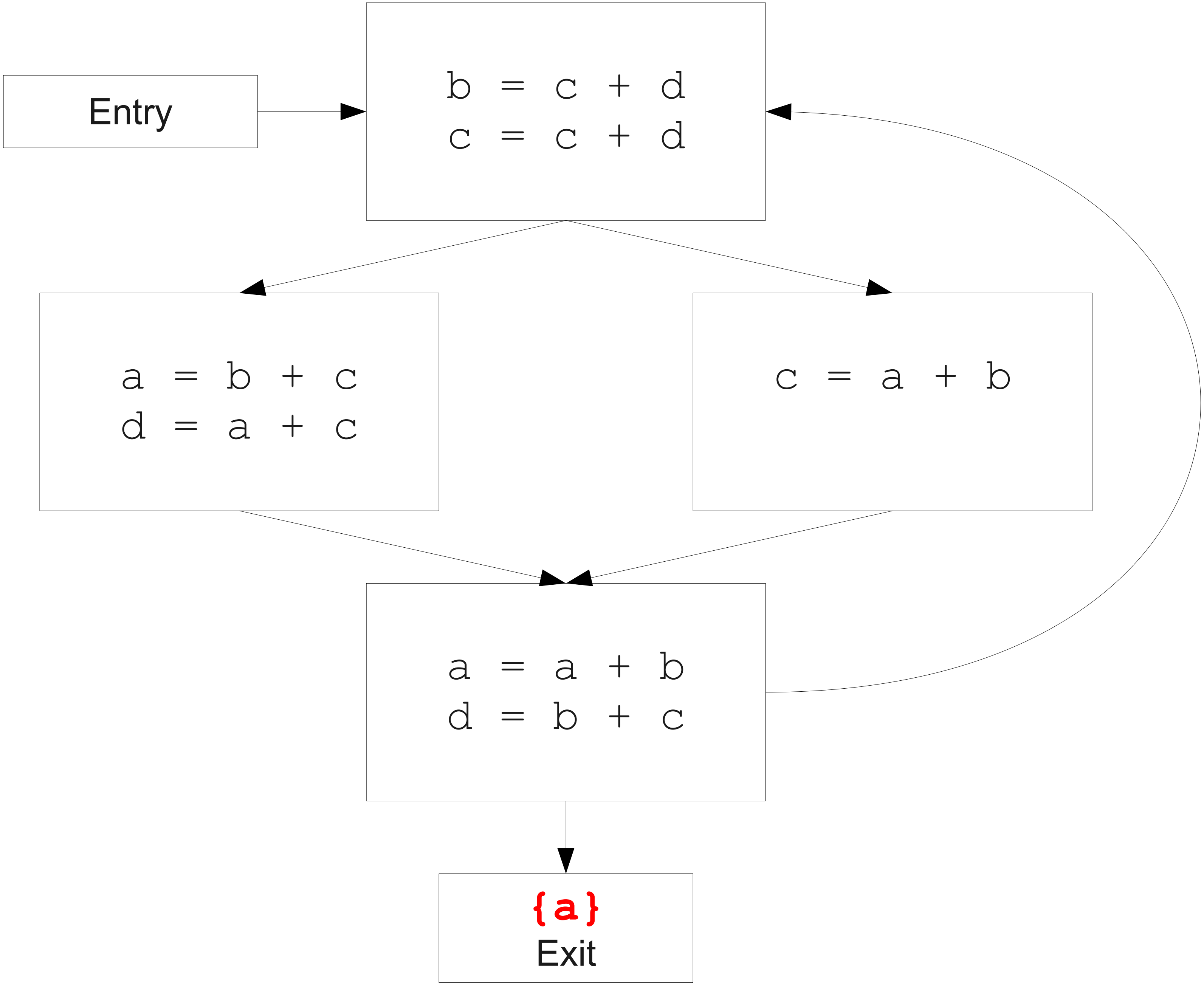
{a}

Exit

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



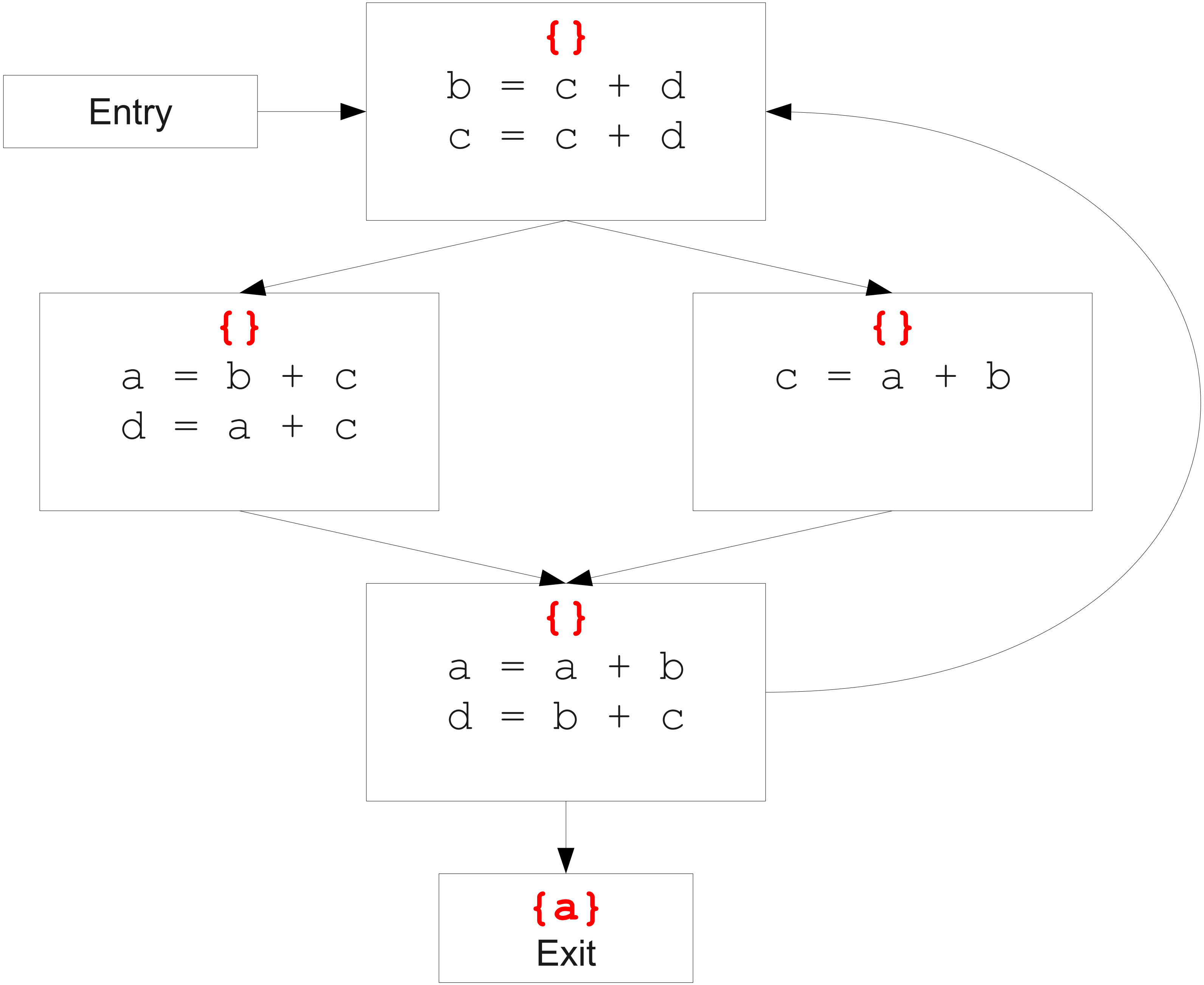
```
b = c + d
c = c + d
```

```
a = b + c
d = a + c
```

```
c = a + b
```

```
a = a + b
d = b + c
```

CFGs With Loops



Entry

{ }
b = c + d
c = c + d

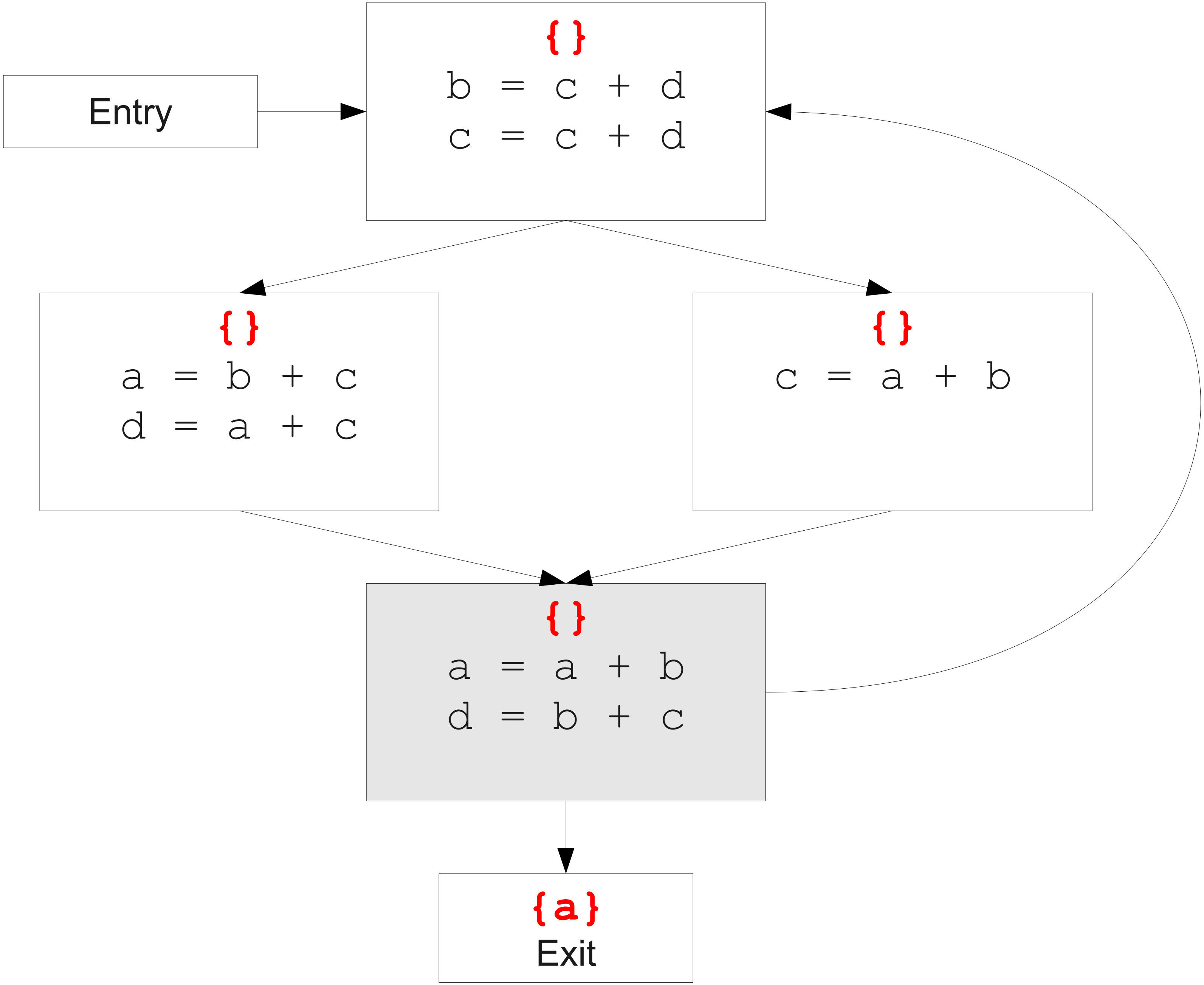
{ }
a = b + c
d = a + c

{ }
c = a + b

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

{a}
Exit

CFGs With Loops



Entry

{ }
b = c + d
c = c + d

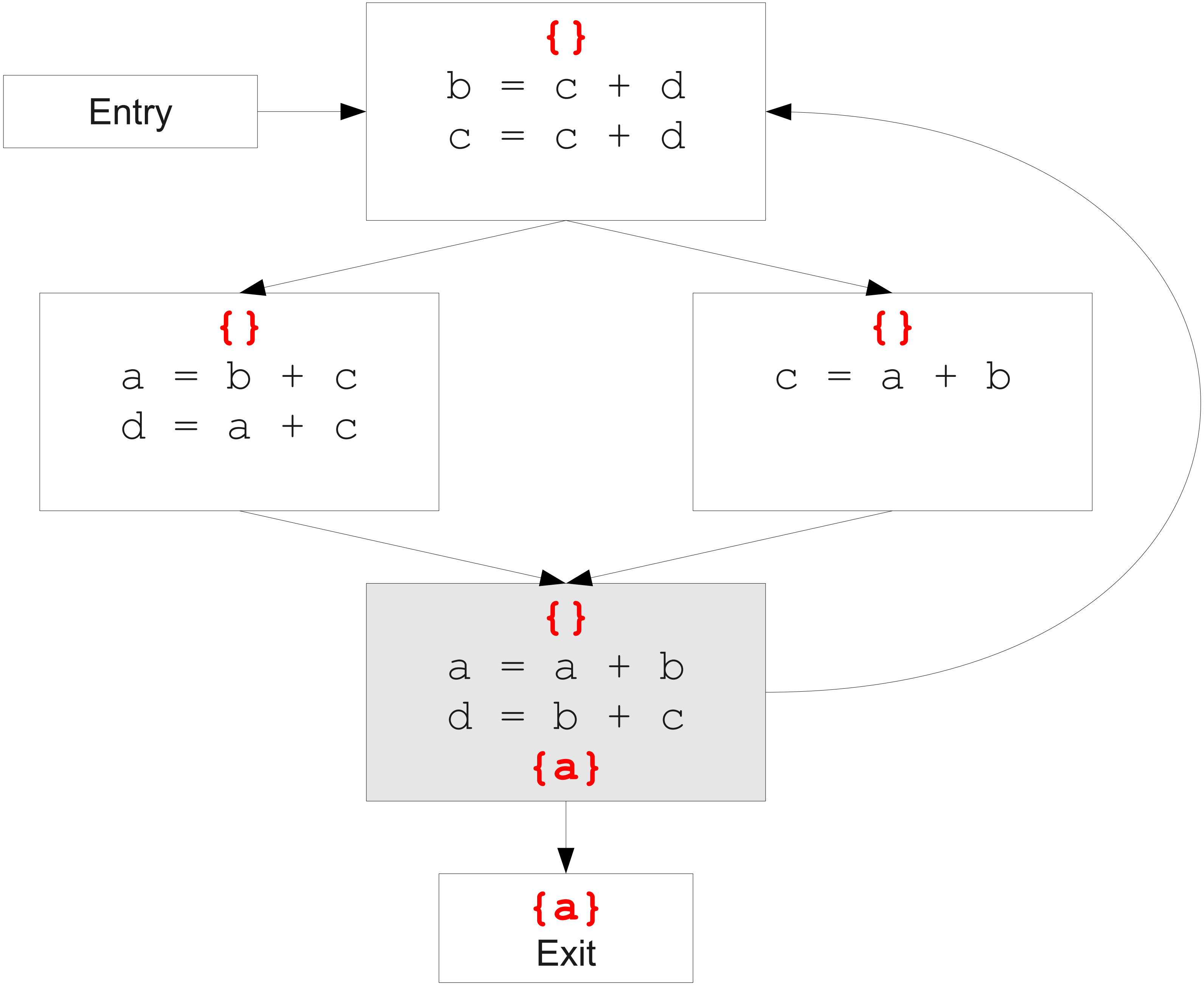
{ }
a = b + c
d = a + c

{ }
c = a + b

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

{a}
Exit

CFGs With Loops



Entry

{ }
b = c + d
c = c + d

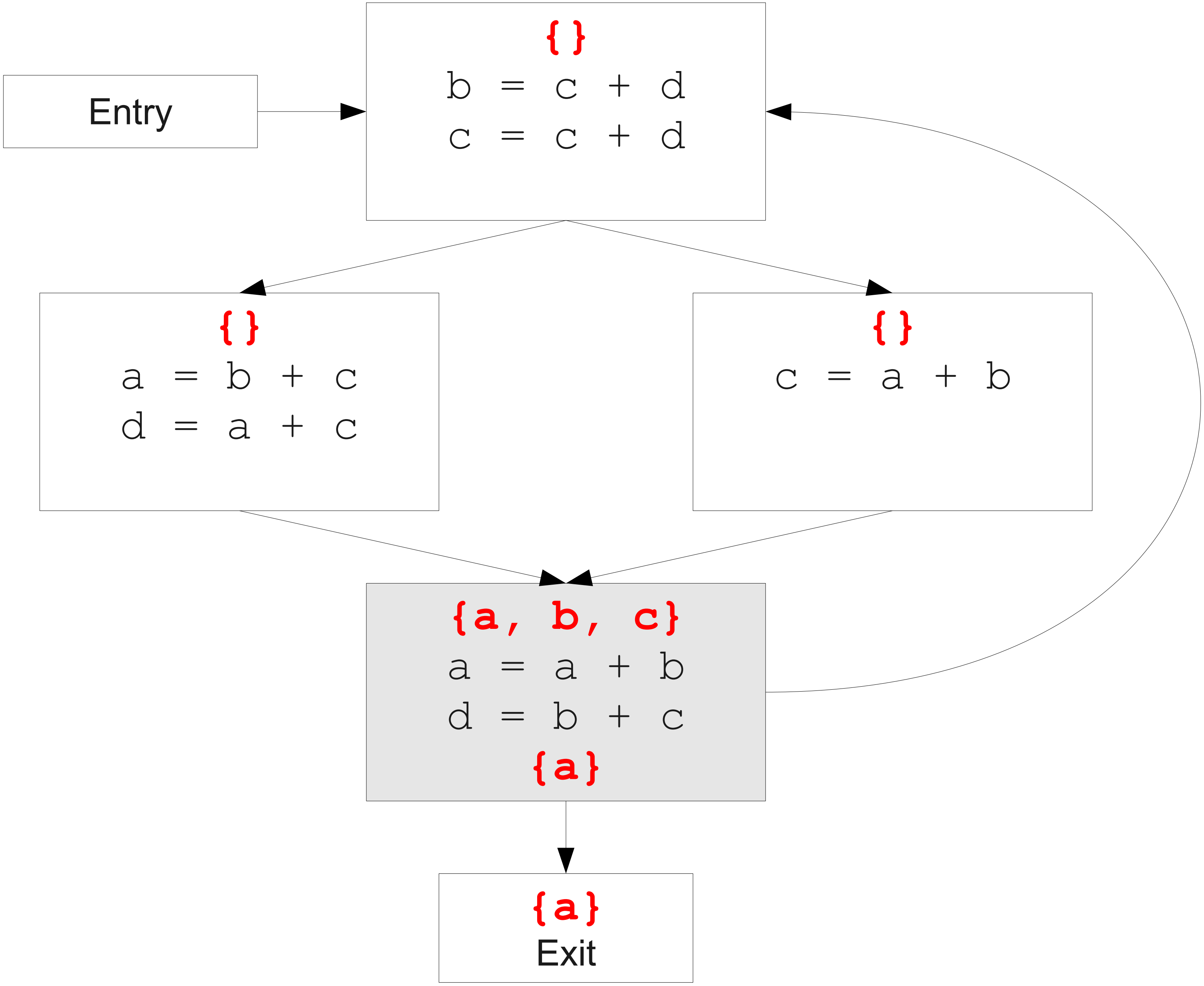
{ }
a = b + c
d = a + c

{ }
c = a + b

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

{a}
Exit

CFGs With Loops



Entry

```
{ }
b = c + d
c = c + d
```

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

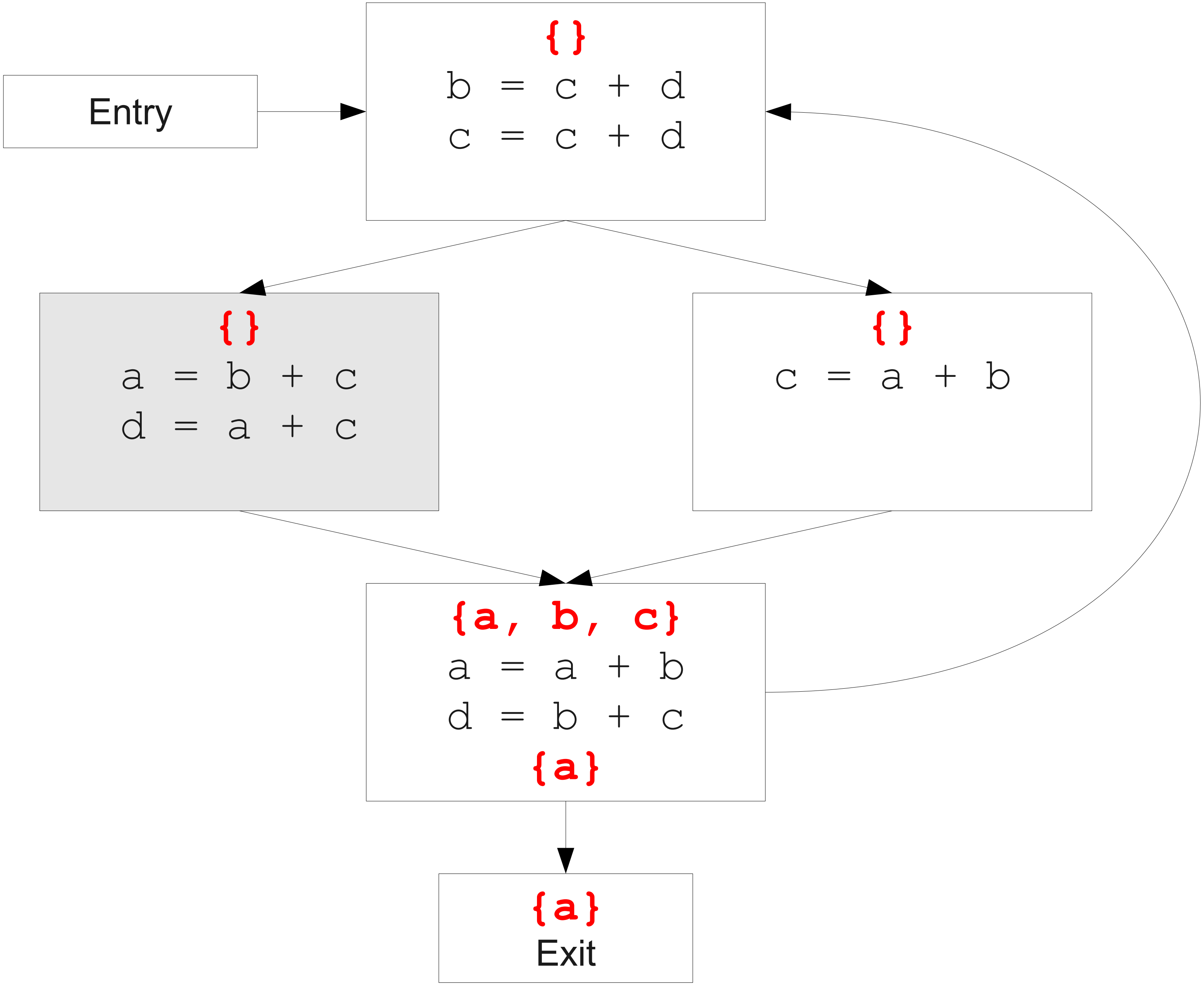
```
{ }
c = a + b
```

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

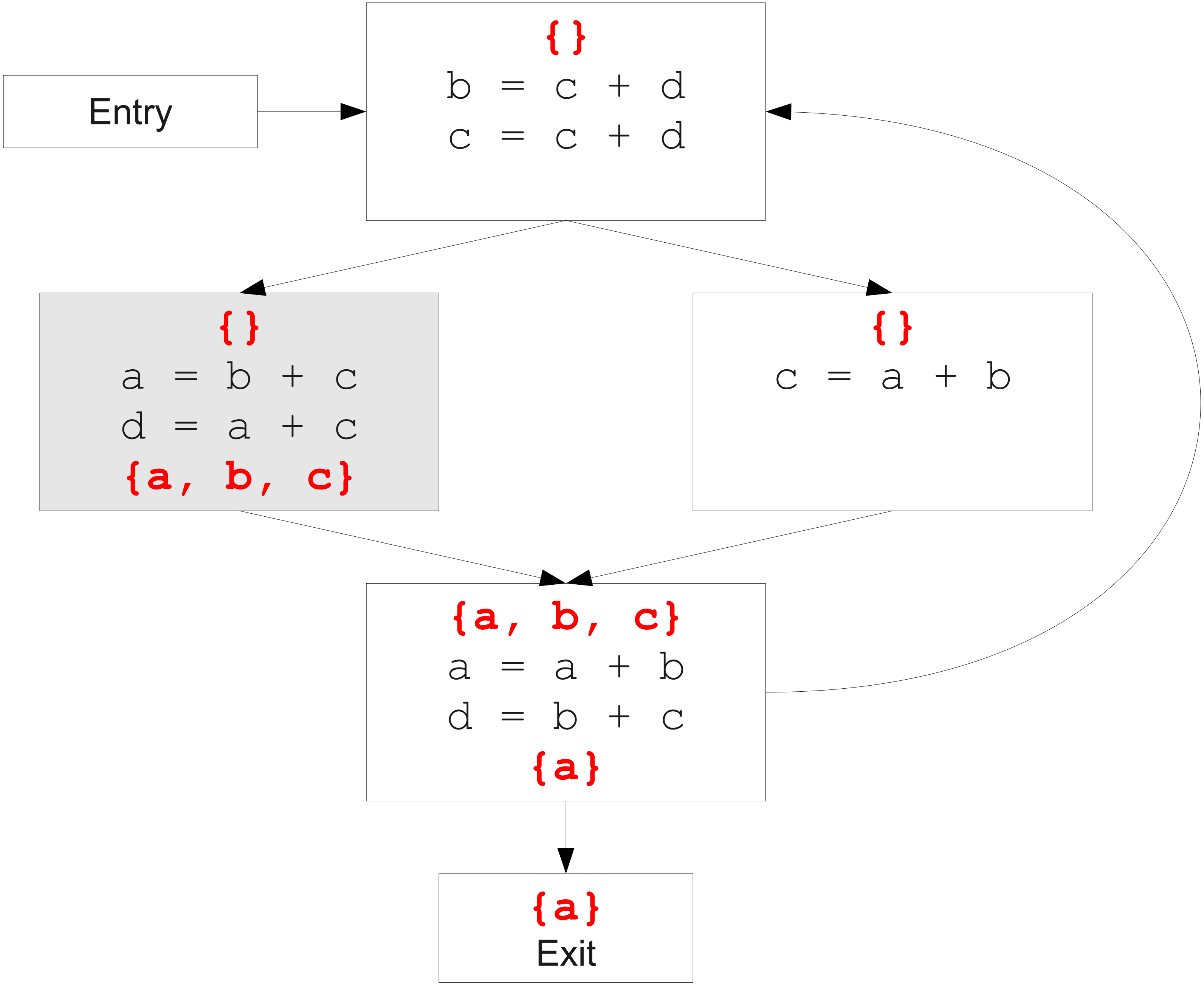
```
{a}
```

Exit

CFGs With Loops



CFGs With Loops



Entry

```
{ }
b = c + d
c = c + d
```

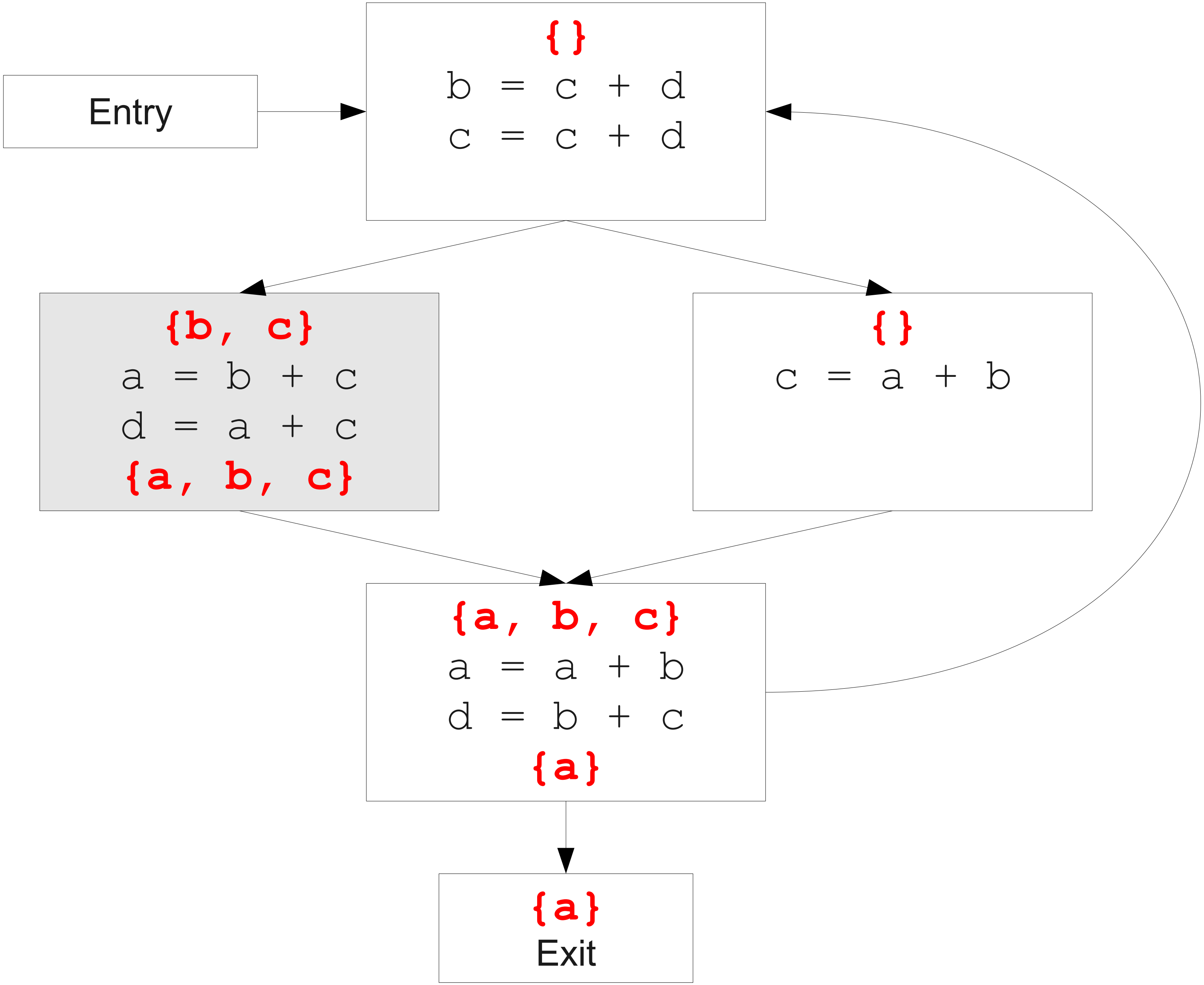
```
{ }
a = b + c
d = a + c
{a, b, c}
```

```
{ }
c = a + b
```

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

```
{a}
Exit
```

CFGs With Loops



Entry

{ }

b = c + d

c = c + d

{b, c}

a = b + c

d = a + c

{a, b, c}

{ }

c = a + b

{a, b, c}

a = a + b

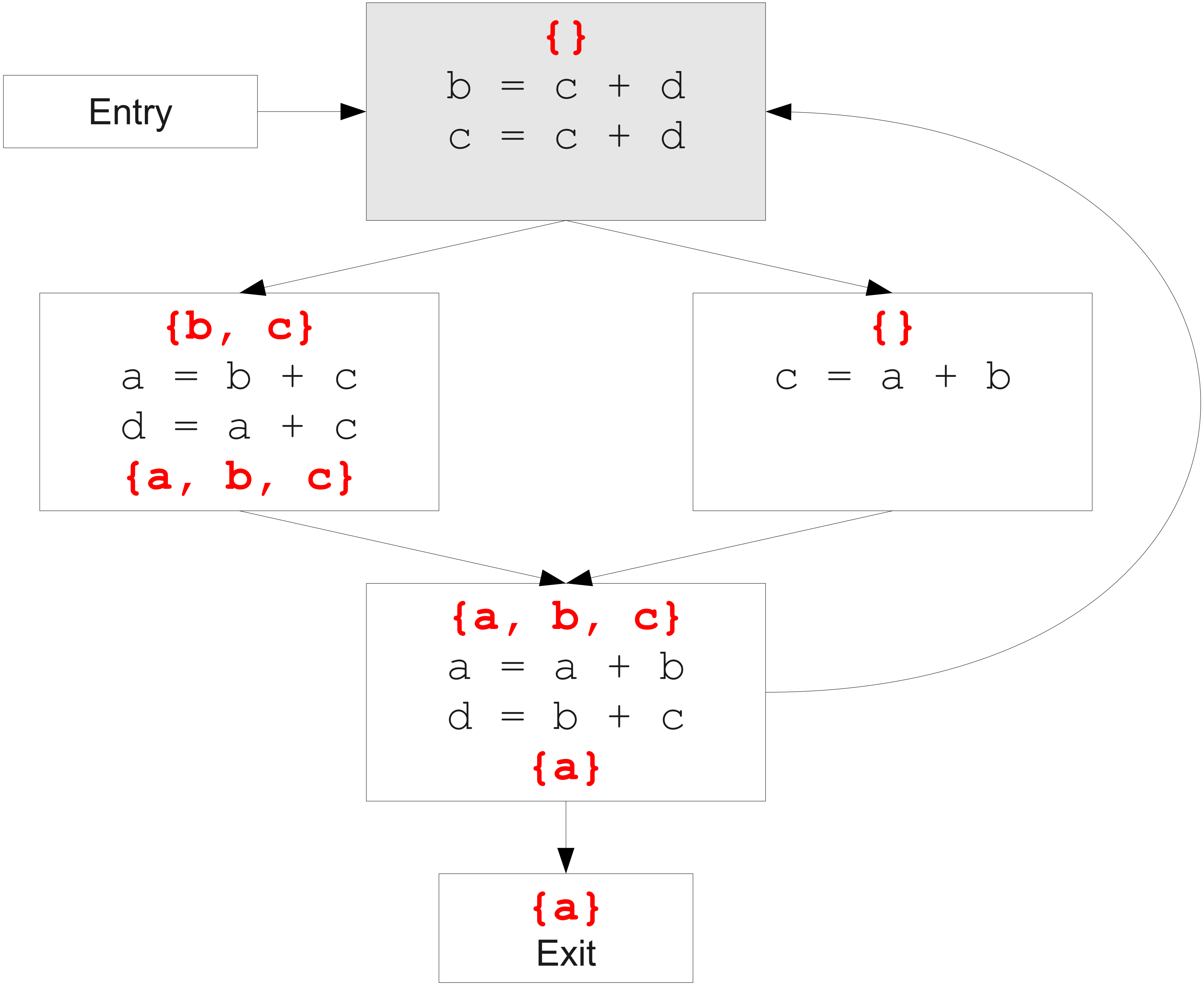
d = b + c

{a}

{a}

Exit

CFGs With Loops



Entry

{ }
b = c + d
c = c + d

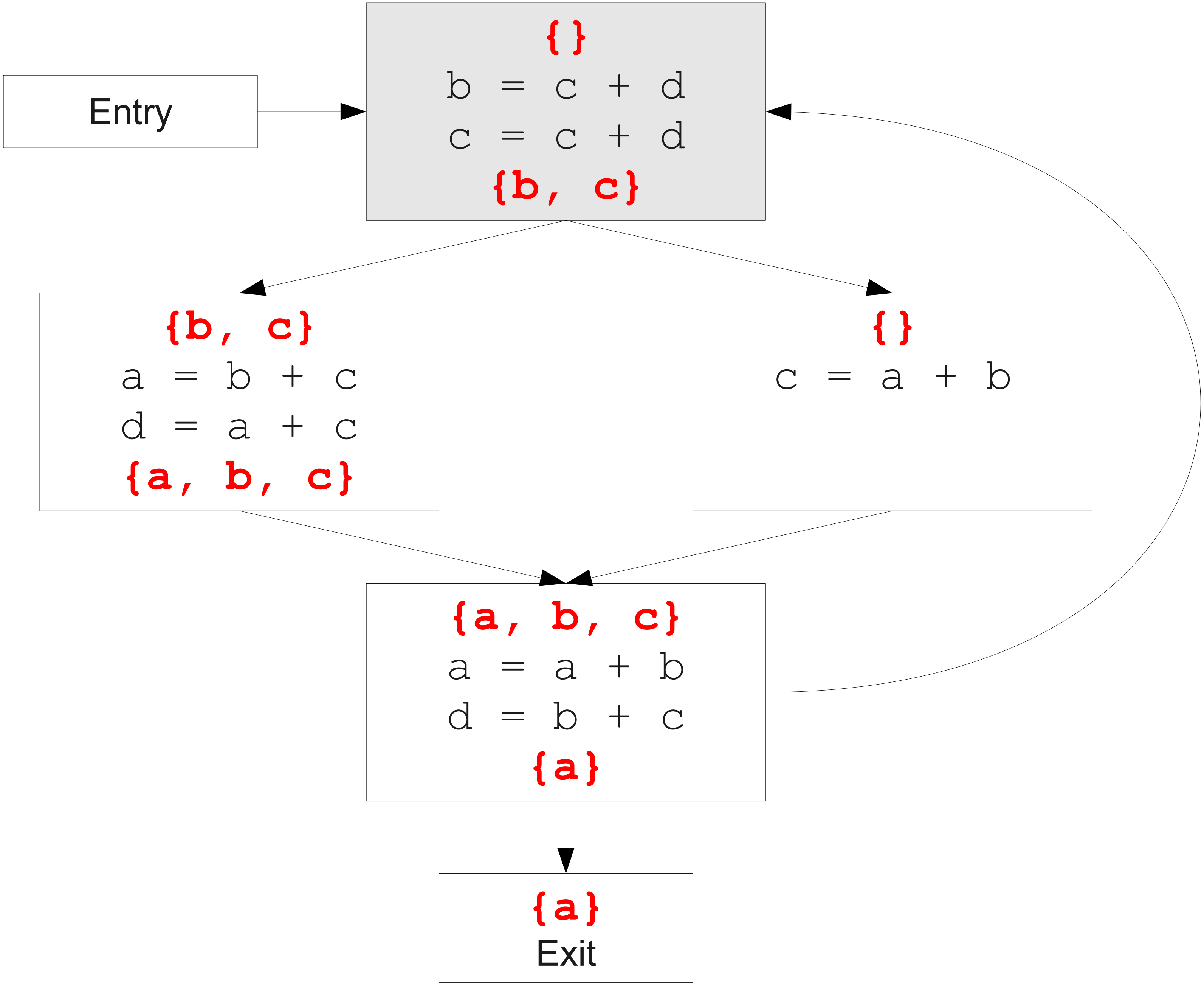
{b, c}
a = b + c
d = a + c
{a, b, c}

{ }
c = a + b

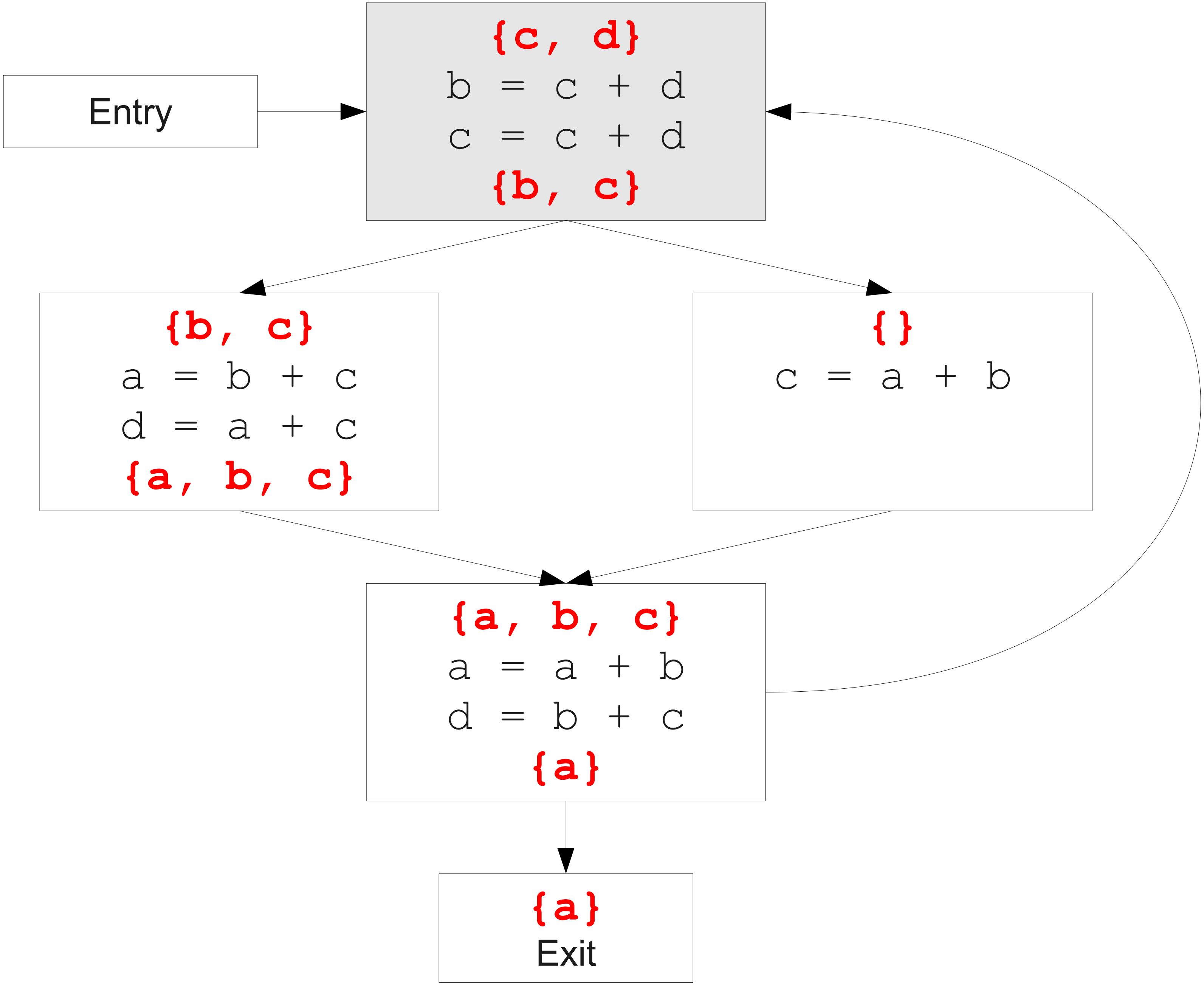
{a, b, c}
a = a + b
d = b + c
{a}

{a}
Exit

CFGs With Loops



CFGs With Loops



Entry

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

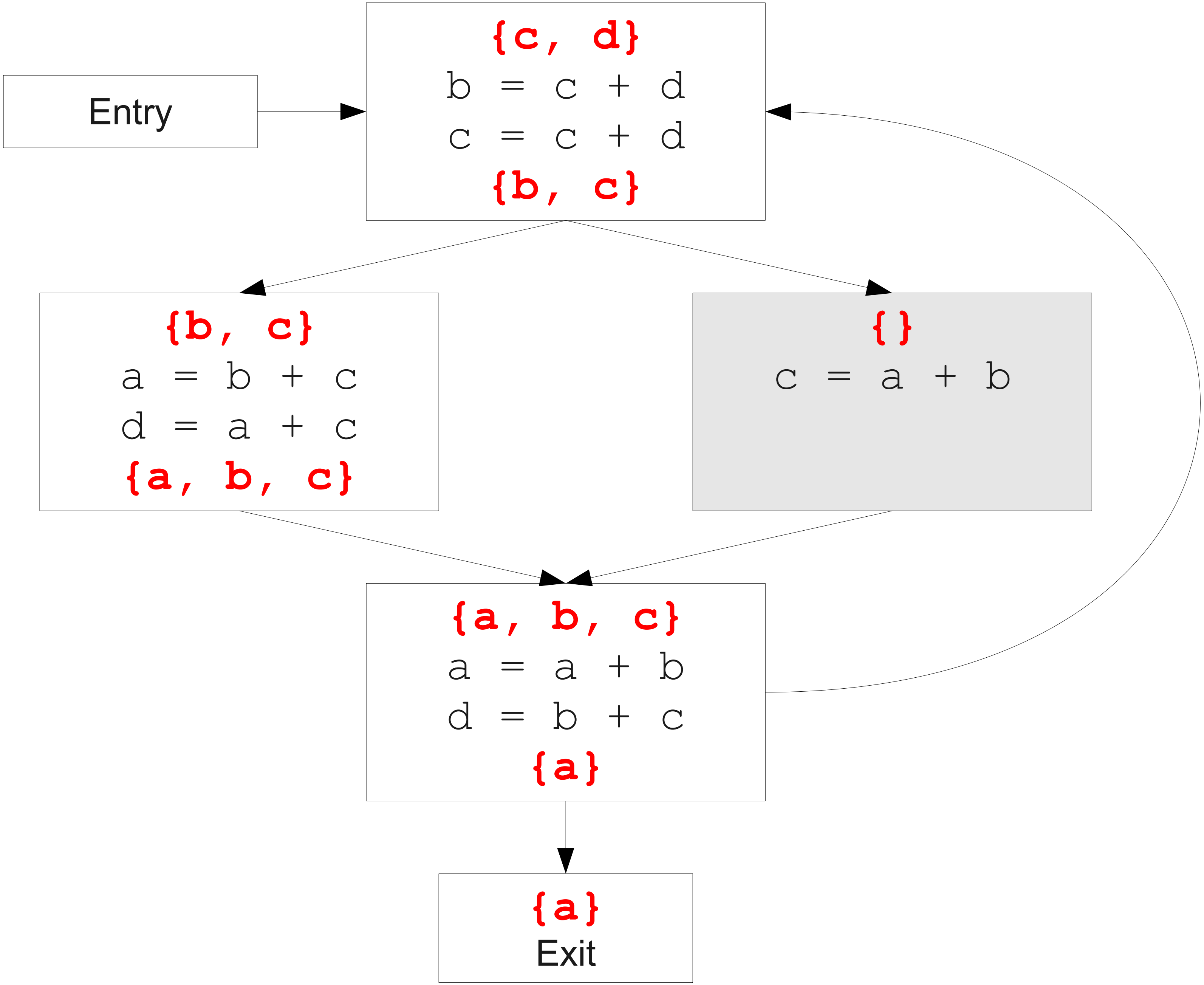
{b, c}
a = b + c
d = a + c
{a, b, c}

{ }
c = a + b

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

{a}
Exit

CFGs With Loops



Entry

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

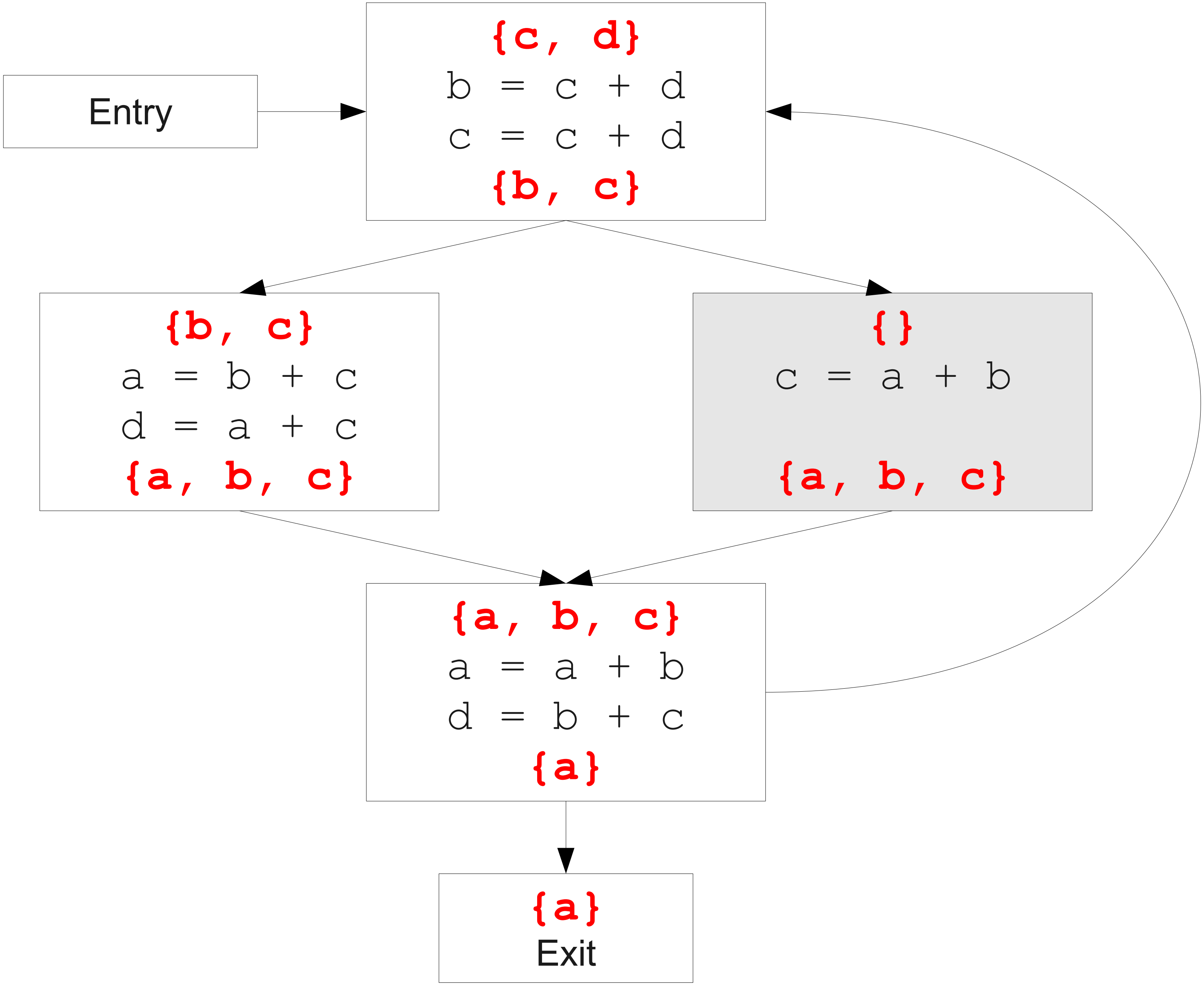
{b, c}
a = b + c
d = a + c
{a, b, c}

{}
c = a + b

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

{a}
Exit

CFGs With Loops



Entry

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

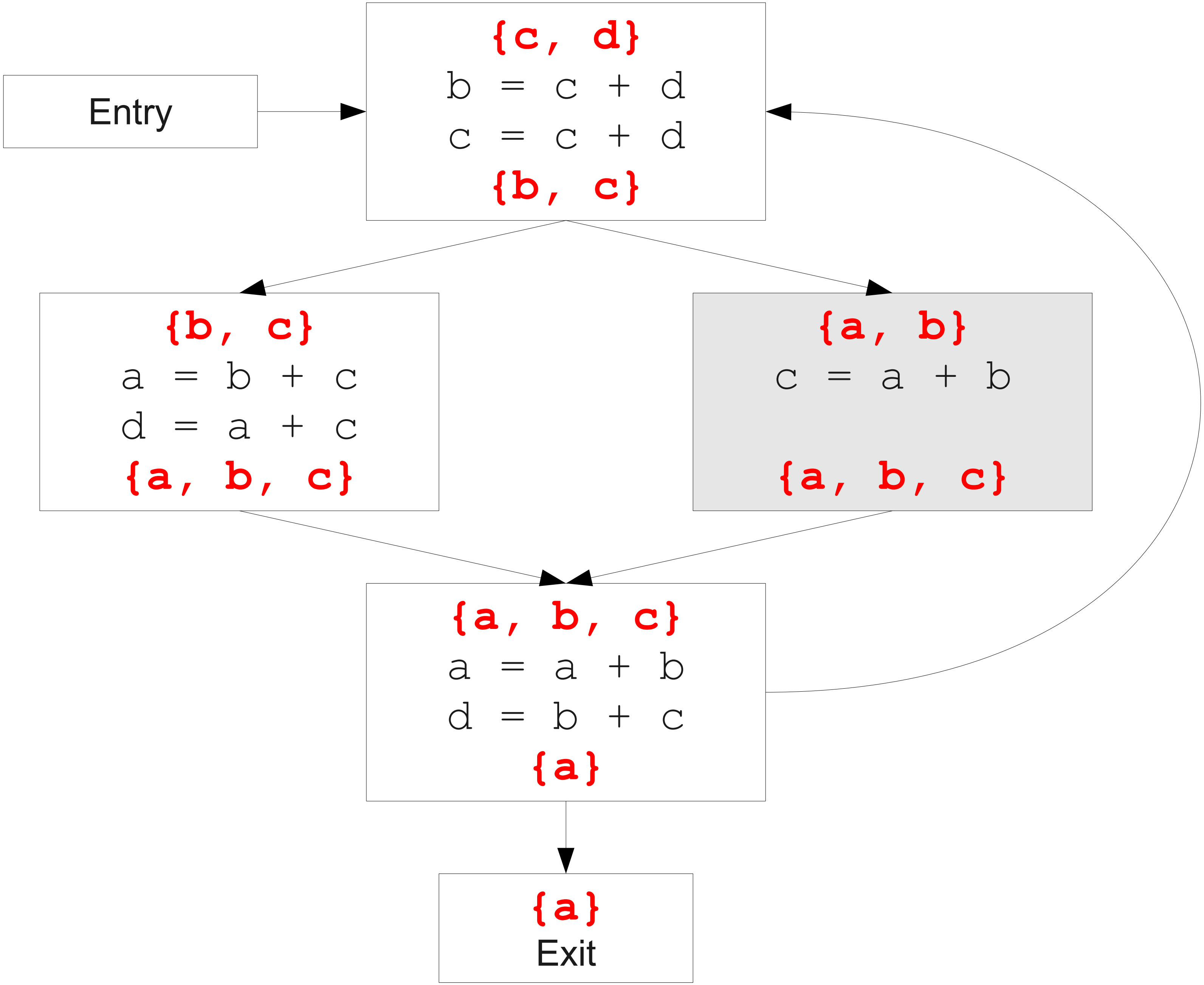
{b, c}
a = b + c
d = a + c
{a, b, c}

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

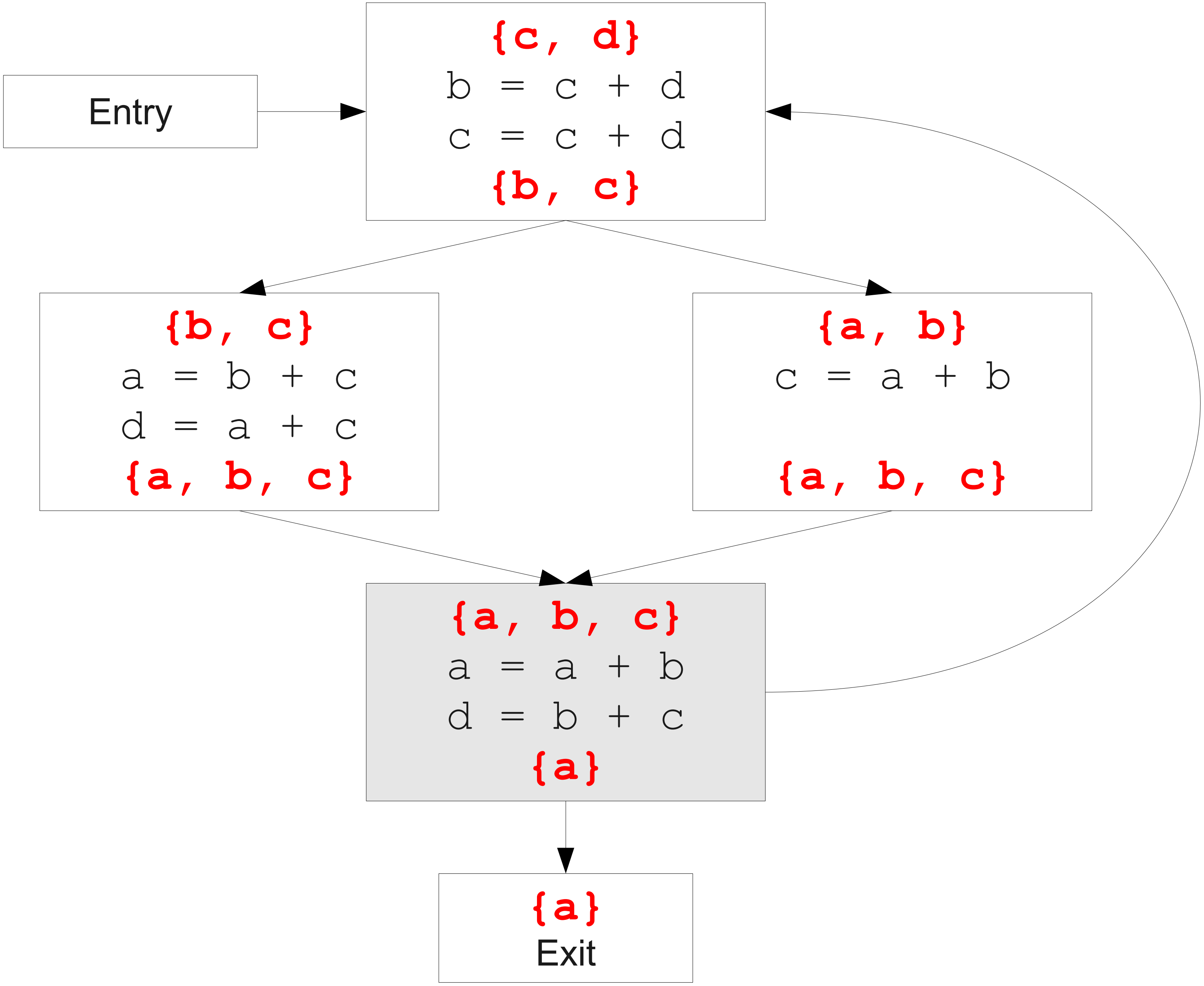
{a, b, c}
a = a + b
d = b + c
{a}

{a}
Exit

CFGs With Loops



CFGs With Loops



Entry

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

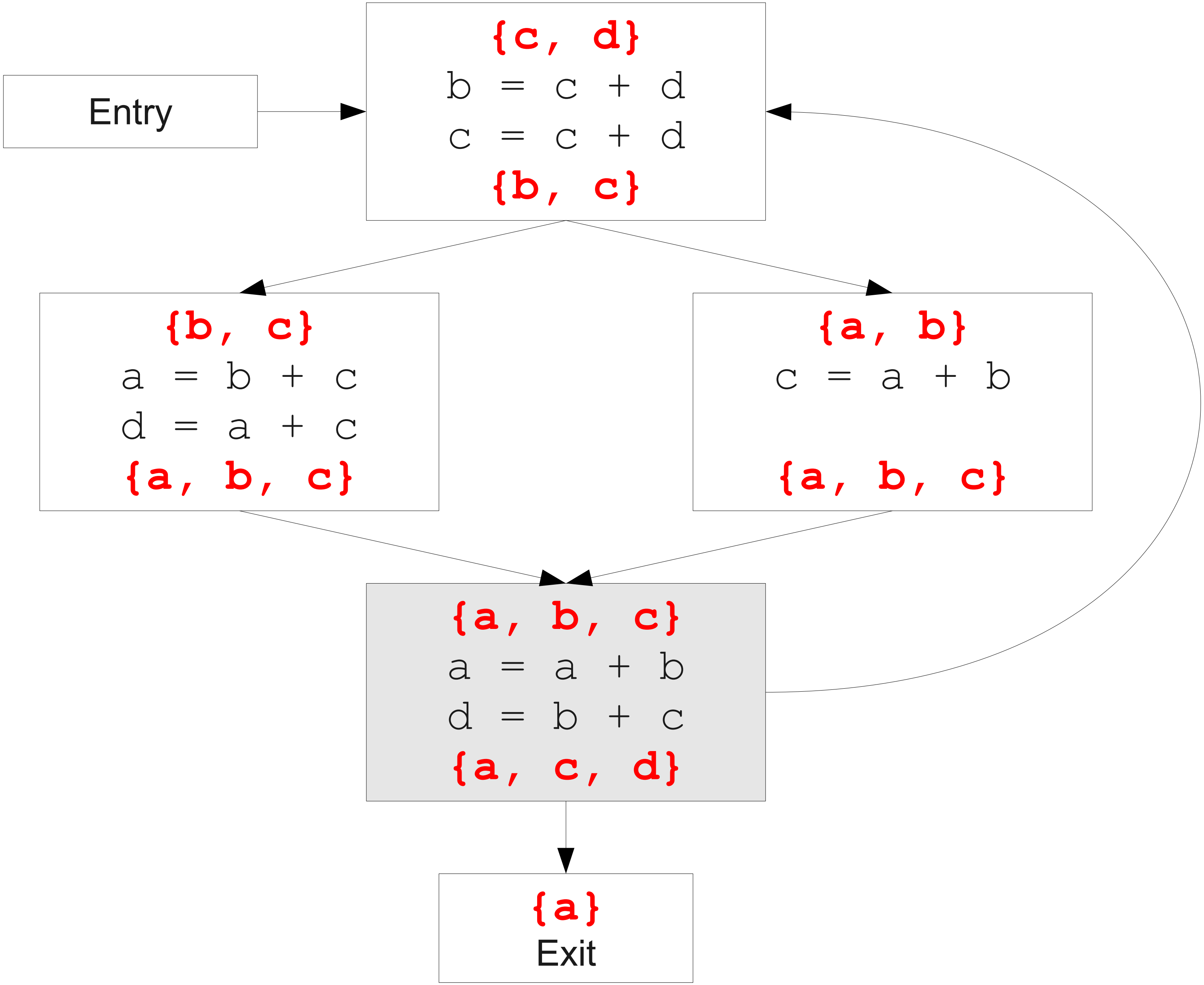
{b, c}
a = b + c
d = a + c
{a, b, c}

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

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

{a}
Exit

CFGs With Loops



Entry

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

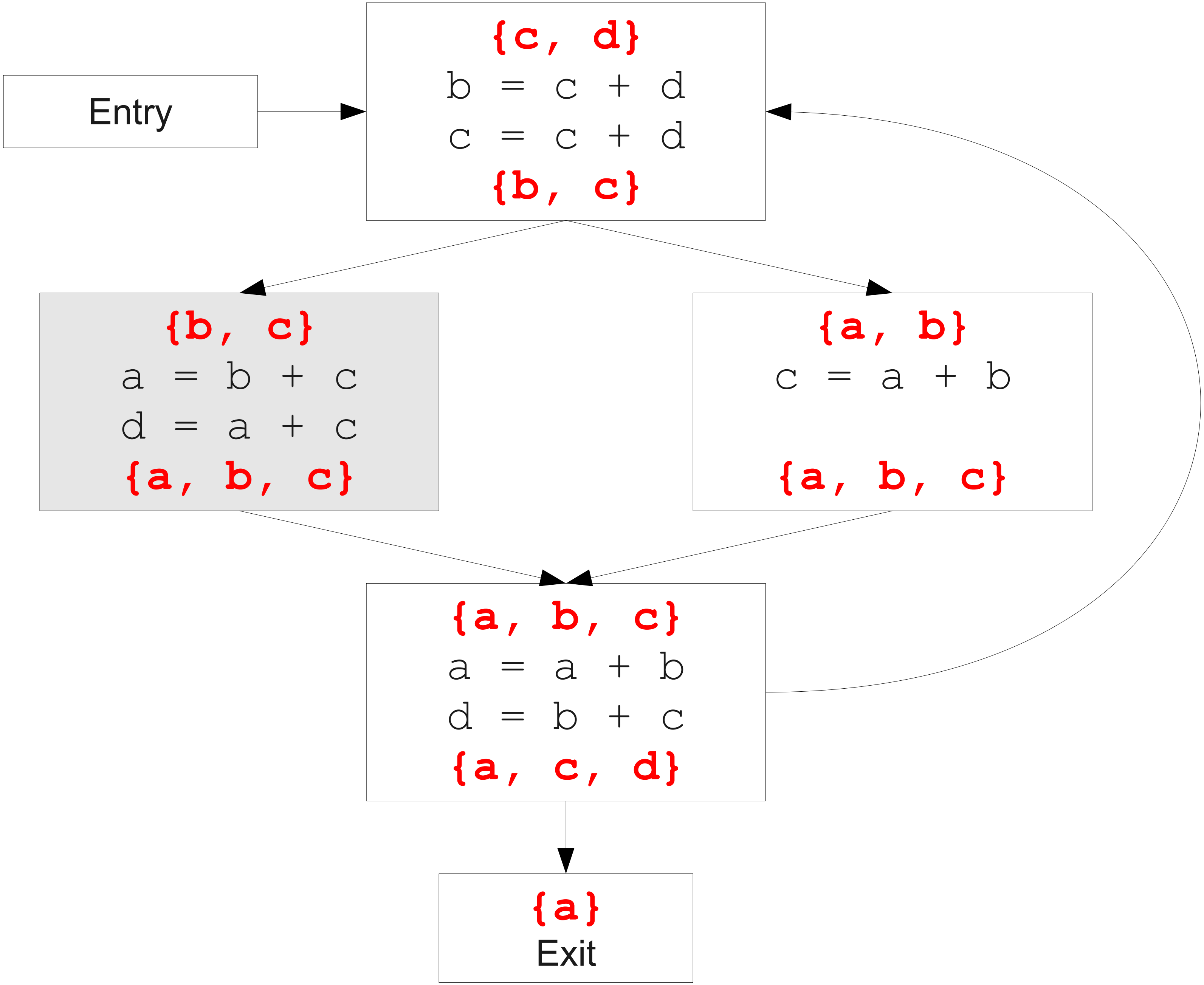
{b, c}
a = b + c
d = a + c
{a, b, c}

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

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

{a}
Exit

CFGs With Loops



Entry

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

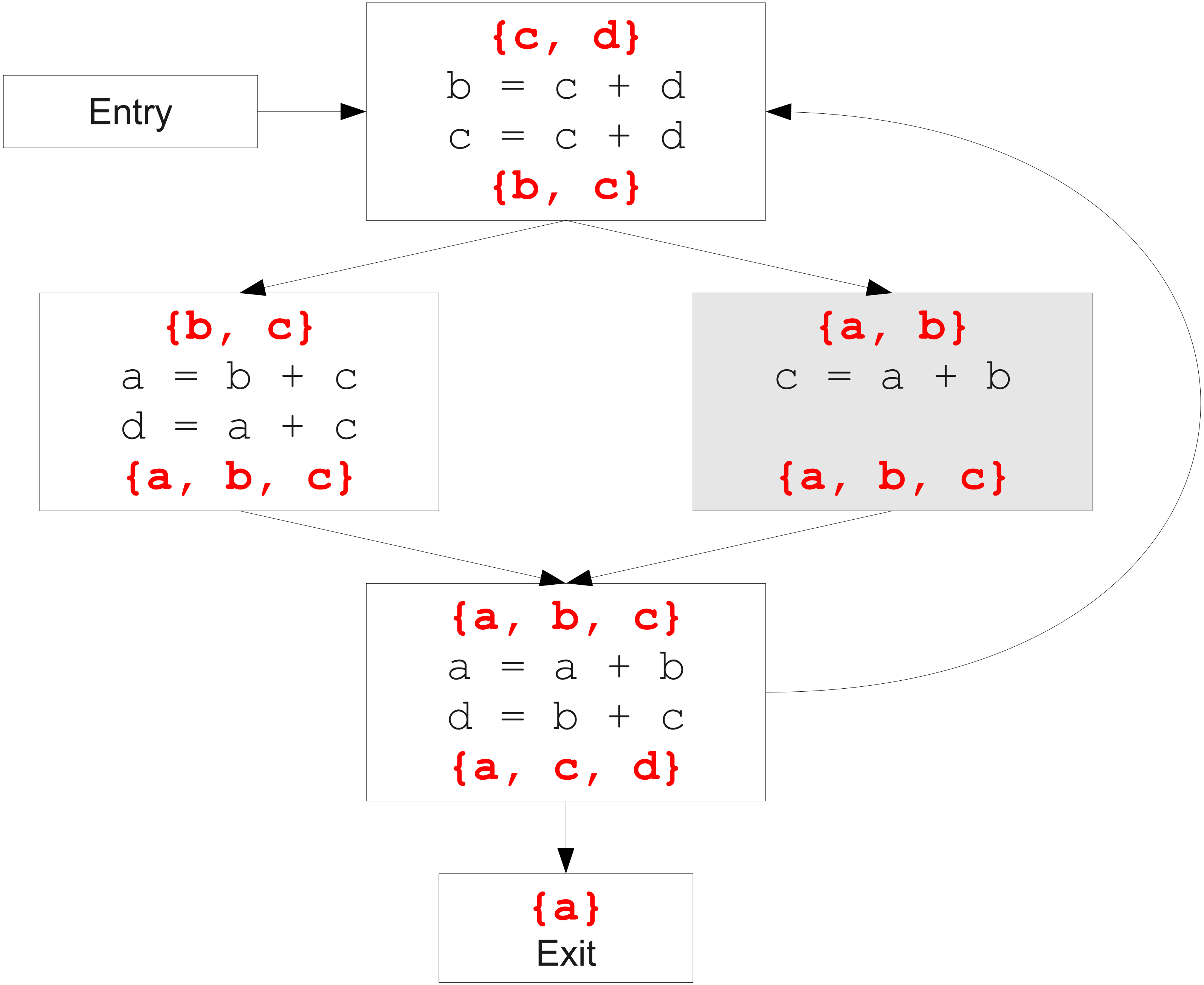
{b, c}
a = b + c
d = a + c
{a, b, c}

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

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

{a}
Exit

CFGs With Loops



Entry

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

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

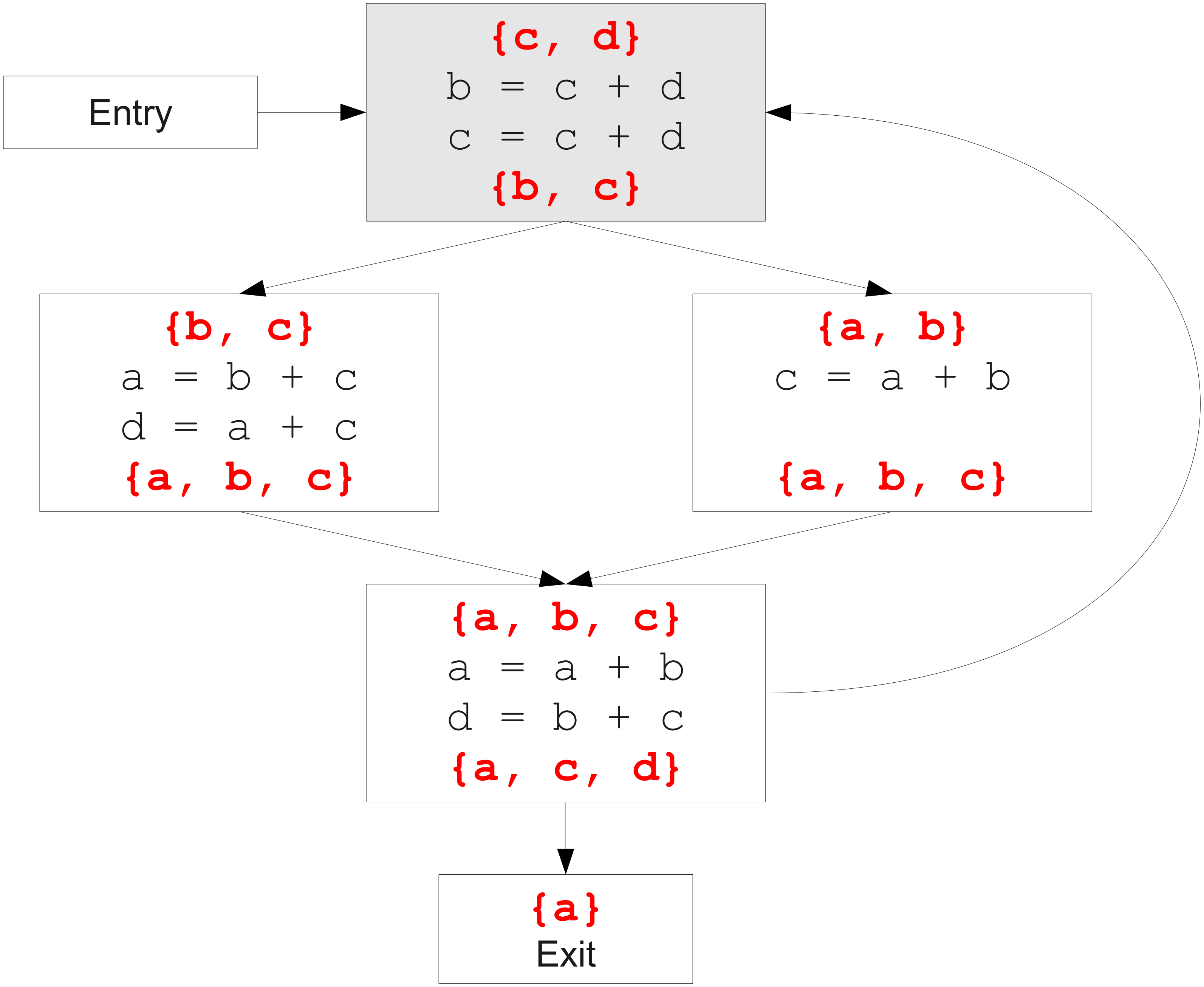
{a, b}
c = a + b

{a, b, c}

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

{a}
Exit

CFGs With Loops



Entry

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

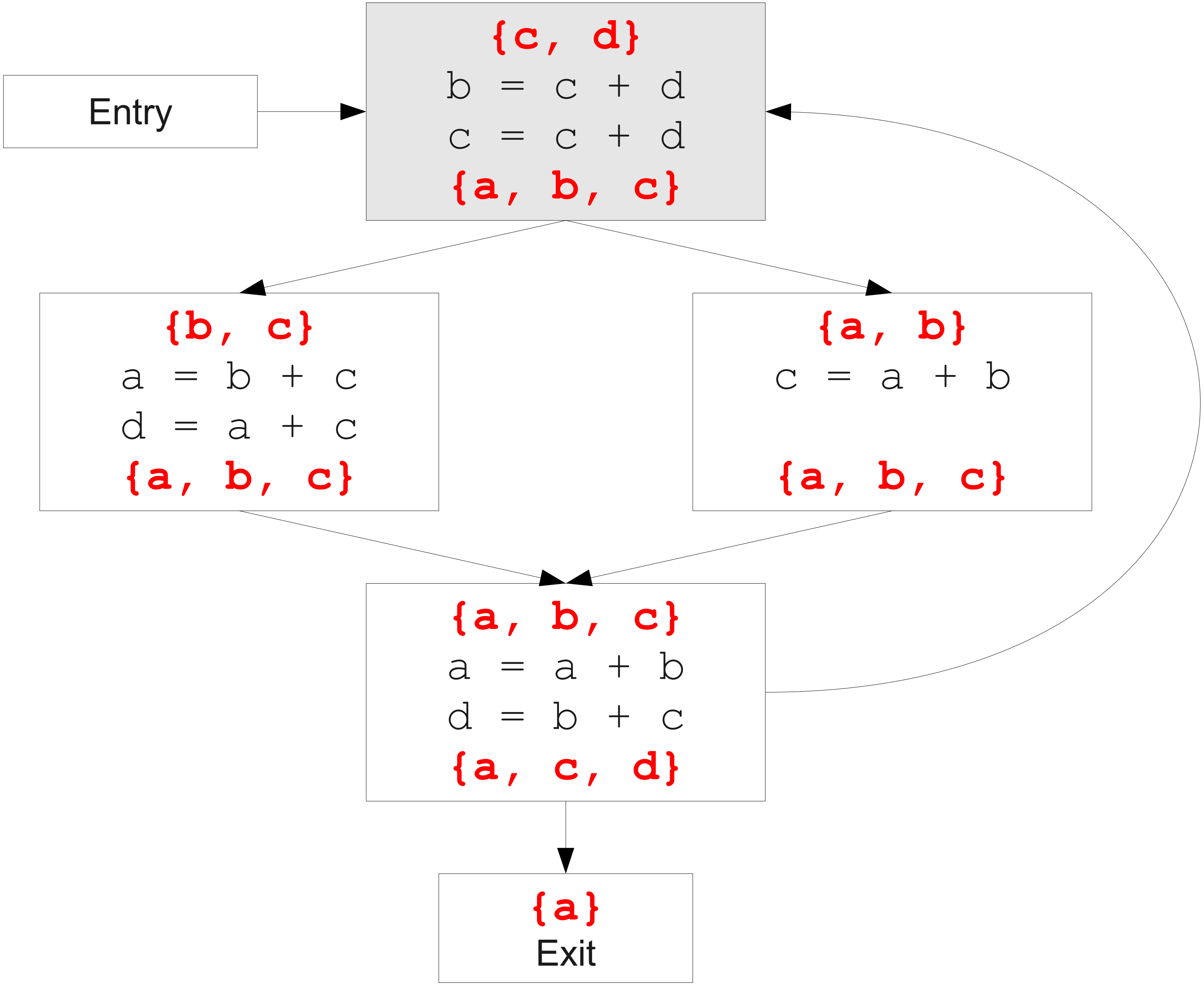
{b, c}
a = b + c
d = a + c
{a, b, c}

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

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

{a}
Exit

CFGs With Loops



Entry

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

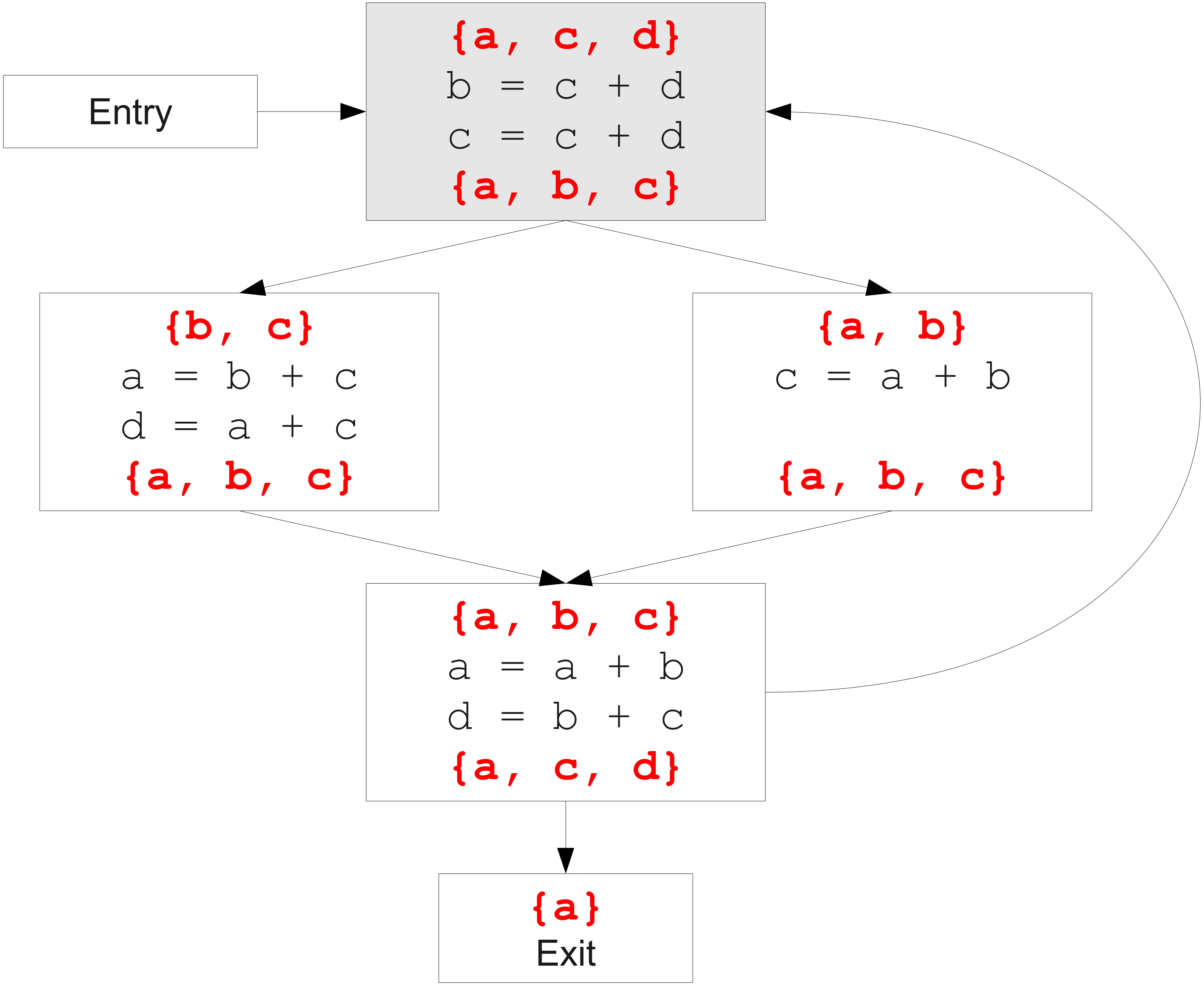
{b, c}
a = b + c
d = a + c
{a, b, c}

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

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

{a}
Exit

CFGs With Loops



Entry

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

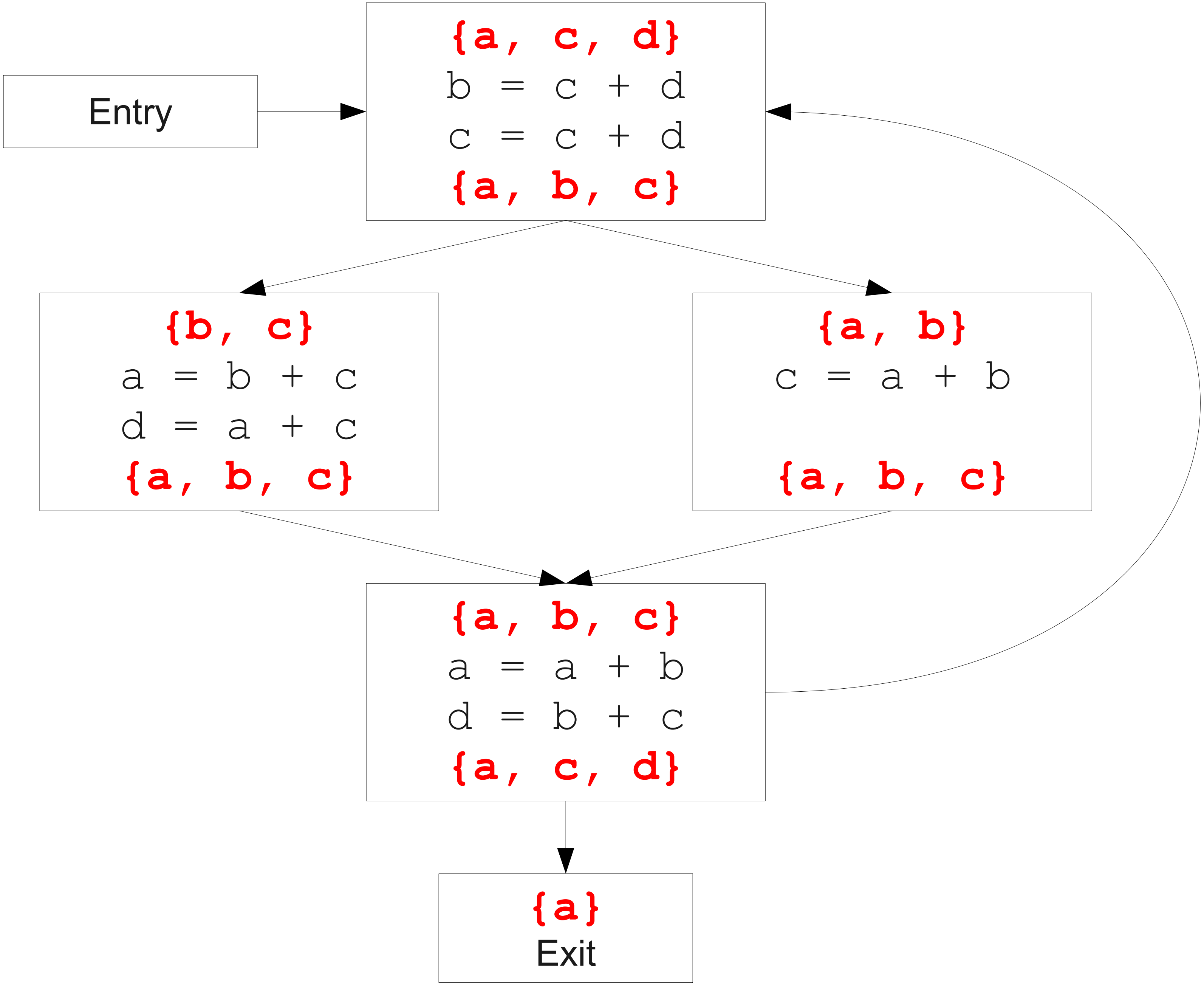
{b, c}
a = b + c
d = a + c
{a, b, c}

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

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

{a}
Exit

CFGs With Loops



Entry

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

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

{a, b}
c = a + b

{a, b, c}

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

{a}
Exit

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

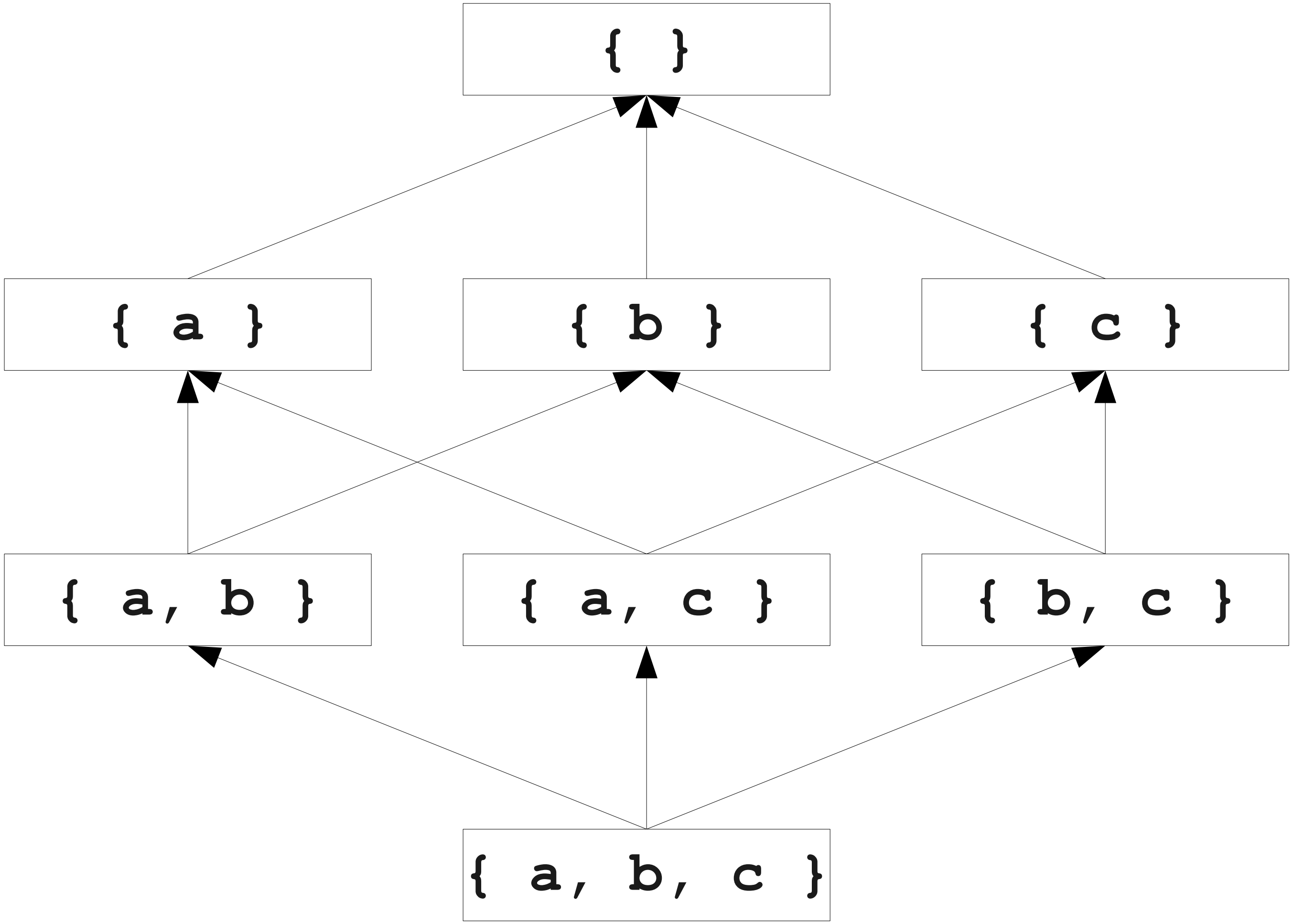
- Building up all of the machinery to design this analysis was tricky.
- The key ideas, however, are mostly independent of the analysis:
  - We need to be able to compute functions describing the behavior of each statement.
  - We need to be able to merge several subcomputations together.
  - We need 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 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



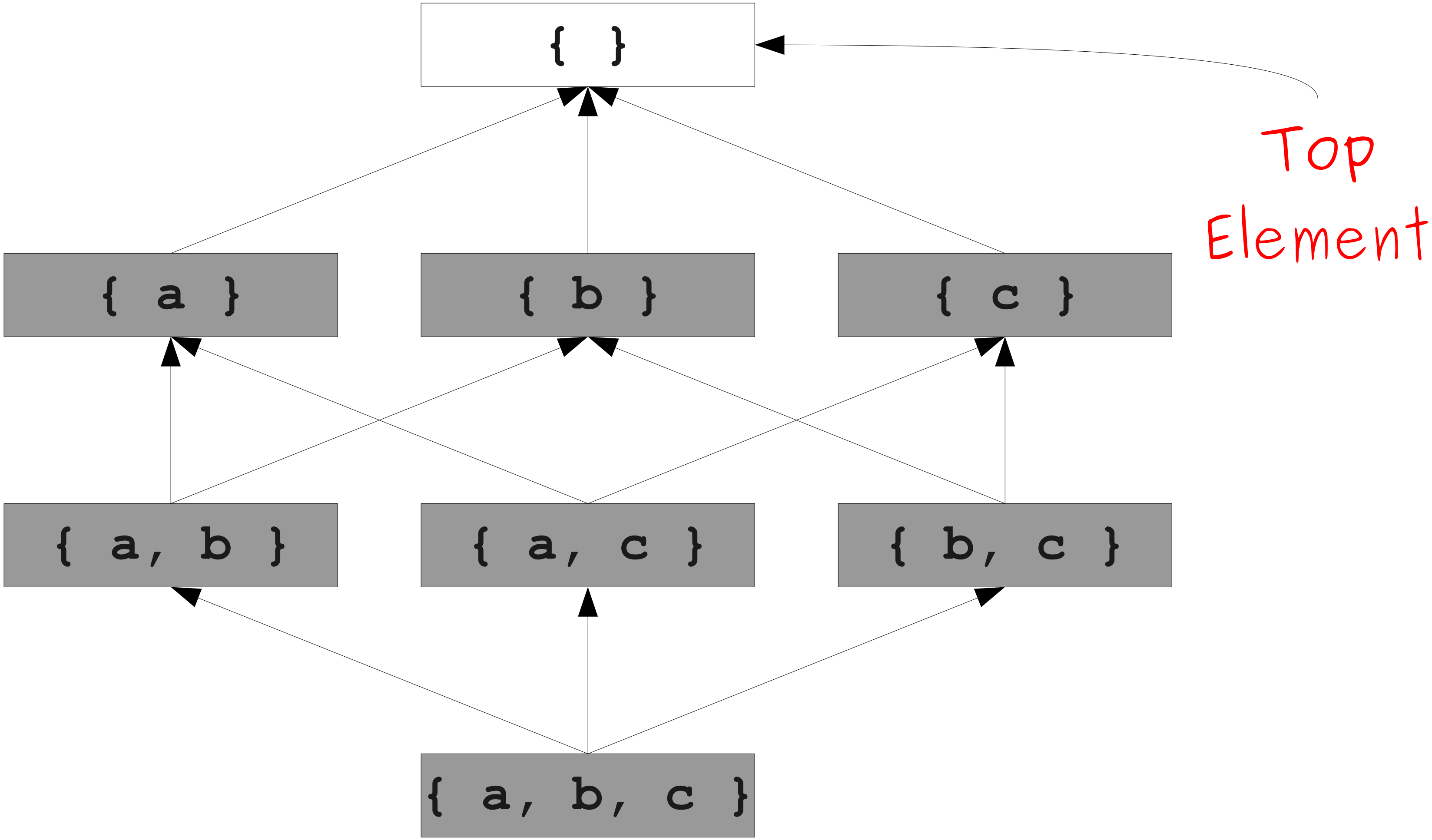
`{ }`

`{ a }` `{ b }` `{ c }`

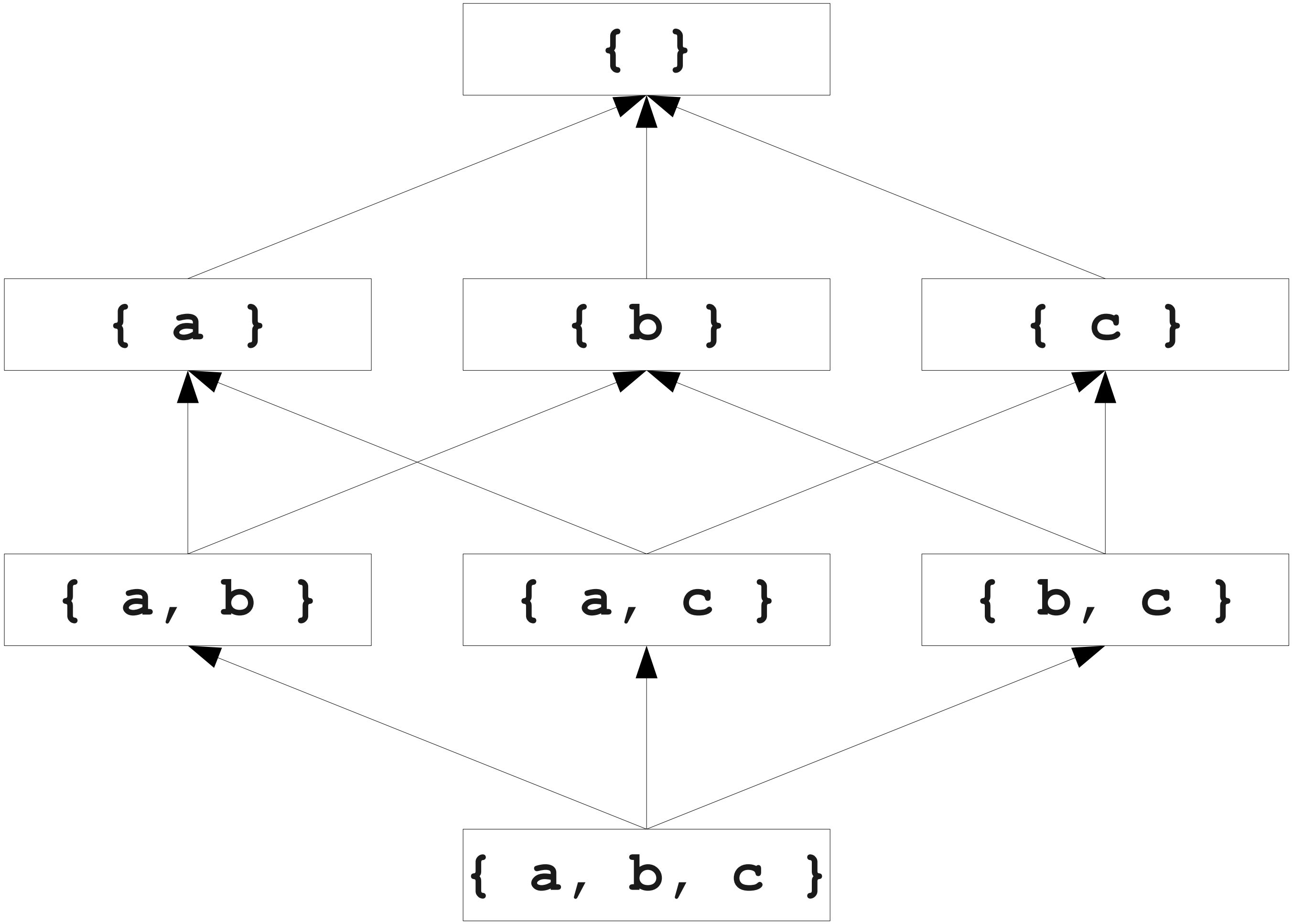
`{ a, b }` `{ a, c }` `{ b, c }`

`{ a, b, c }`

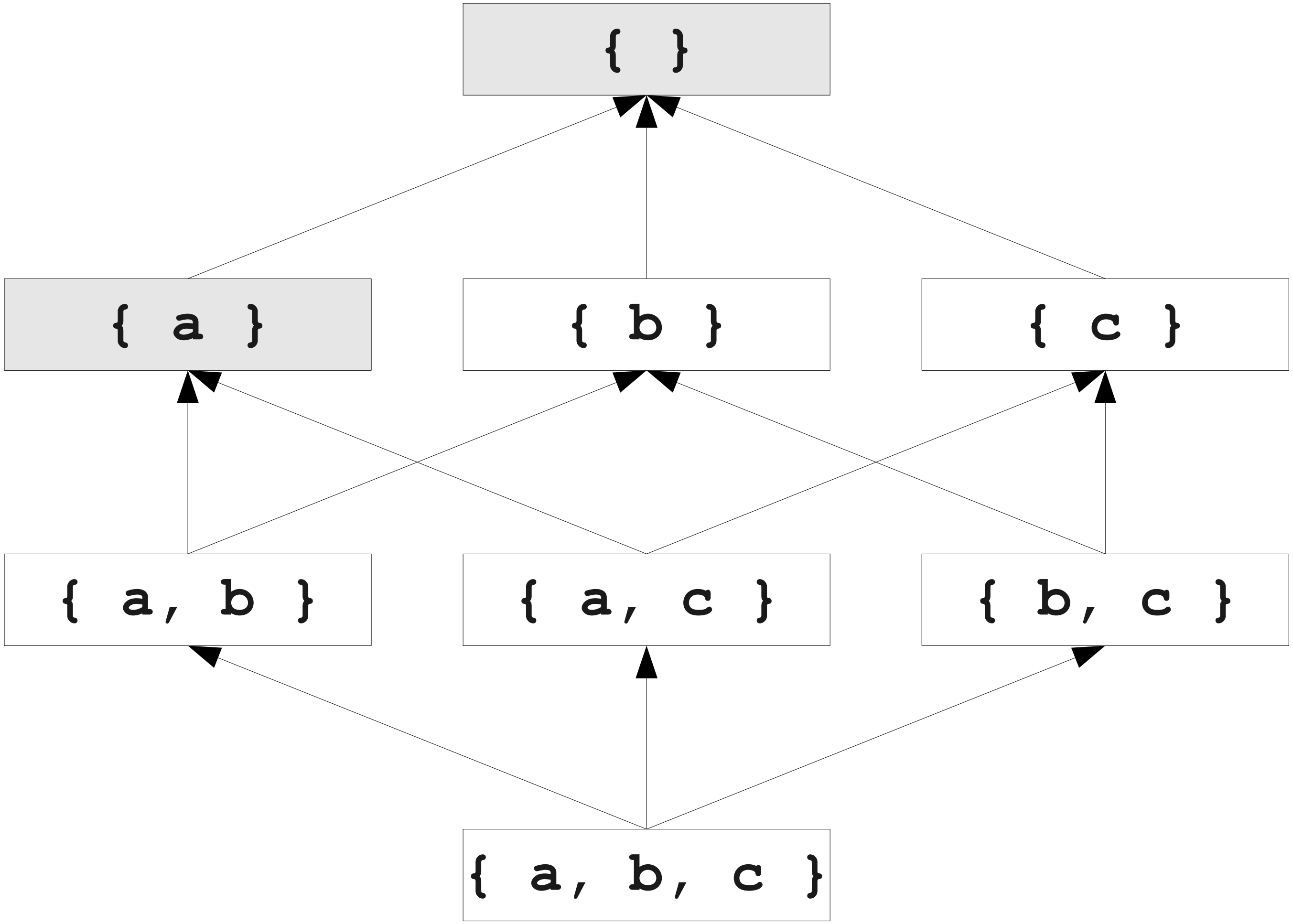
Meet Semilattices for Liveness



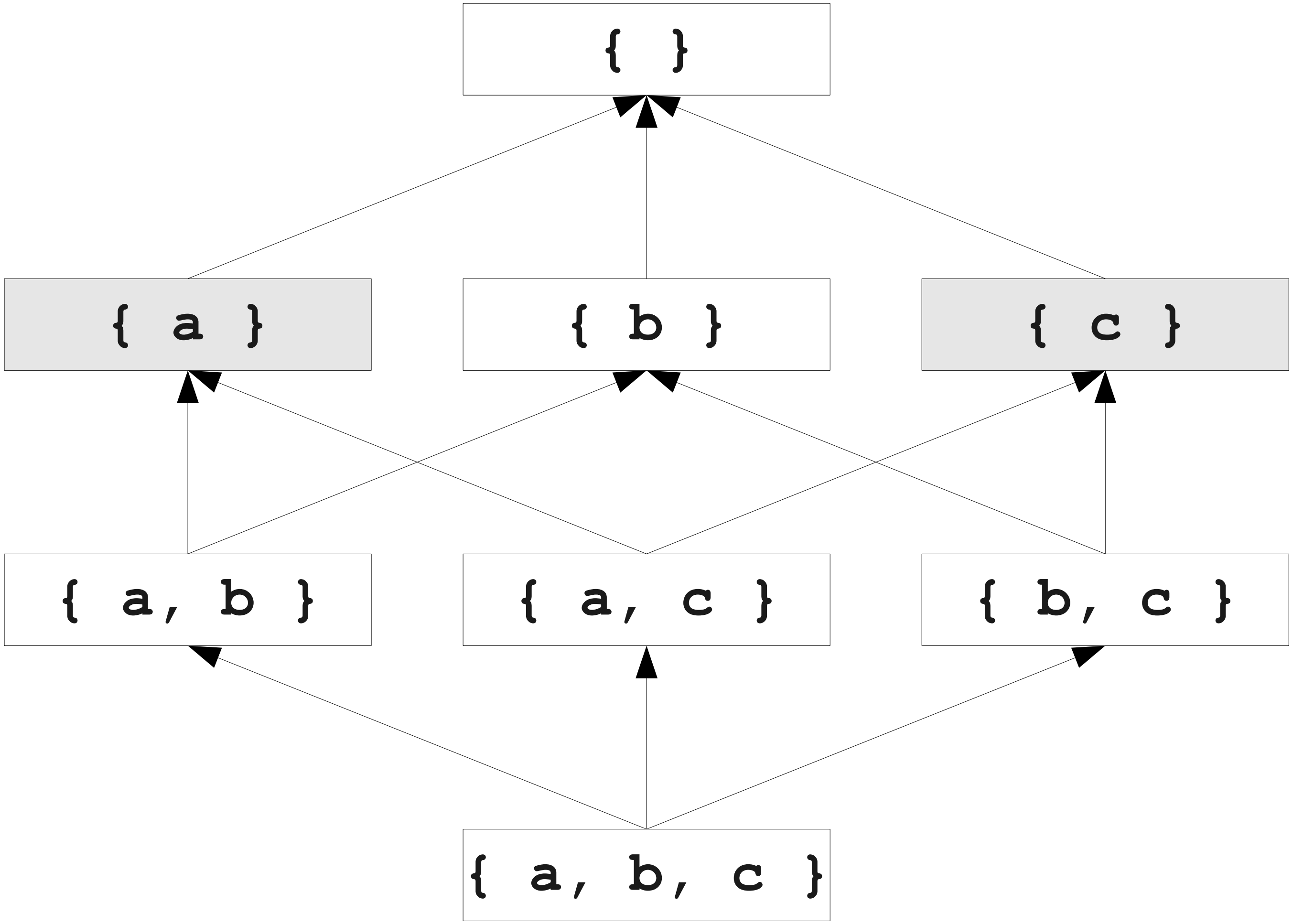
Meet Semilattices for Liveness



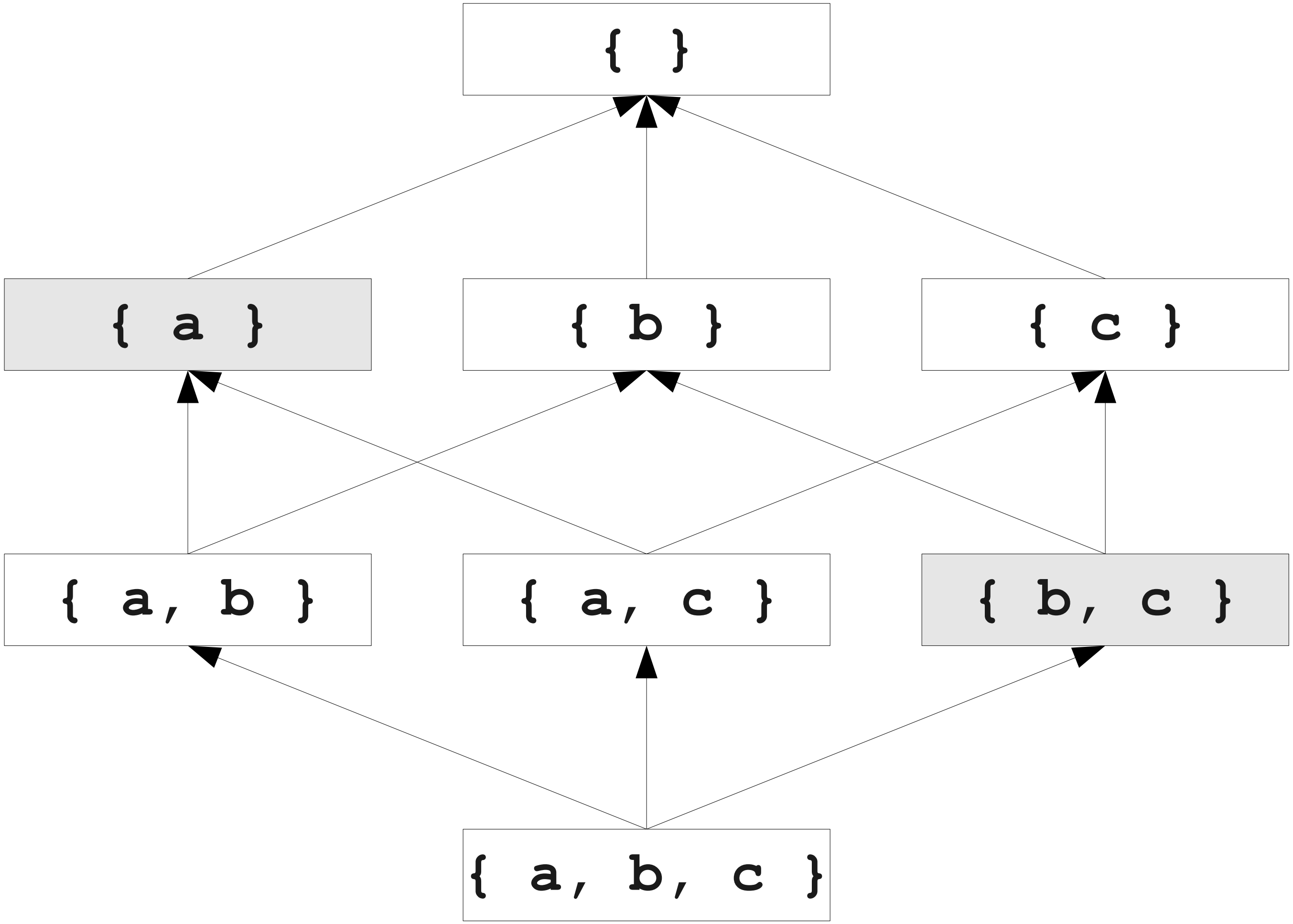
Meet Semilattices for Liveness



Meet Semilattices for Liveness



Meet Semilattices for Liveness

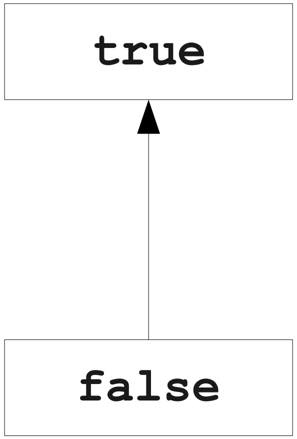


Formal Definitions

- 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.

Is this a Meet Semilattice?

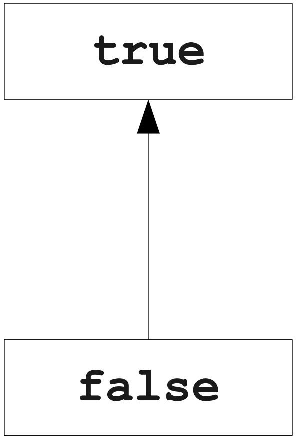
Is this a Meet Semilattice?



`true`

`false`

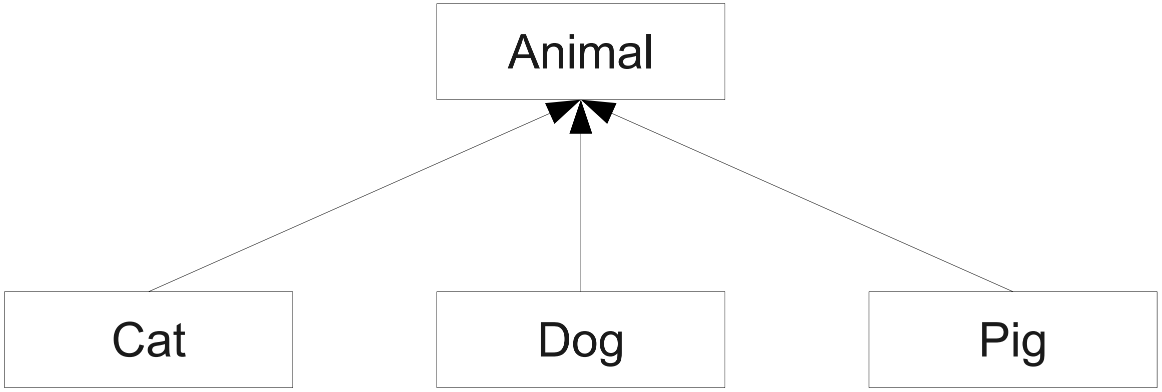
Is this a Meet Semilattice?



What is the meet operator here?

Is this a Meet Semilattice?

Is this a Meet Semilattice?



Animal

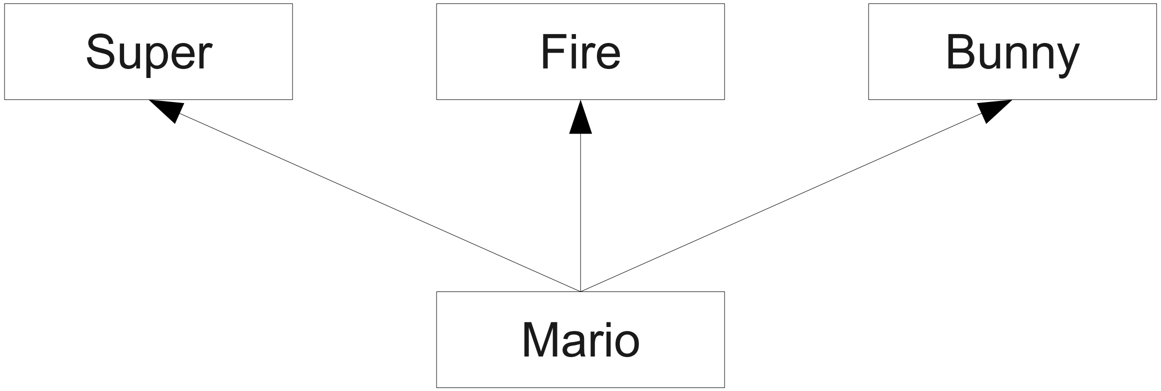
Cat

Dog

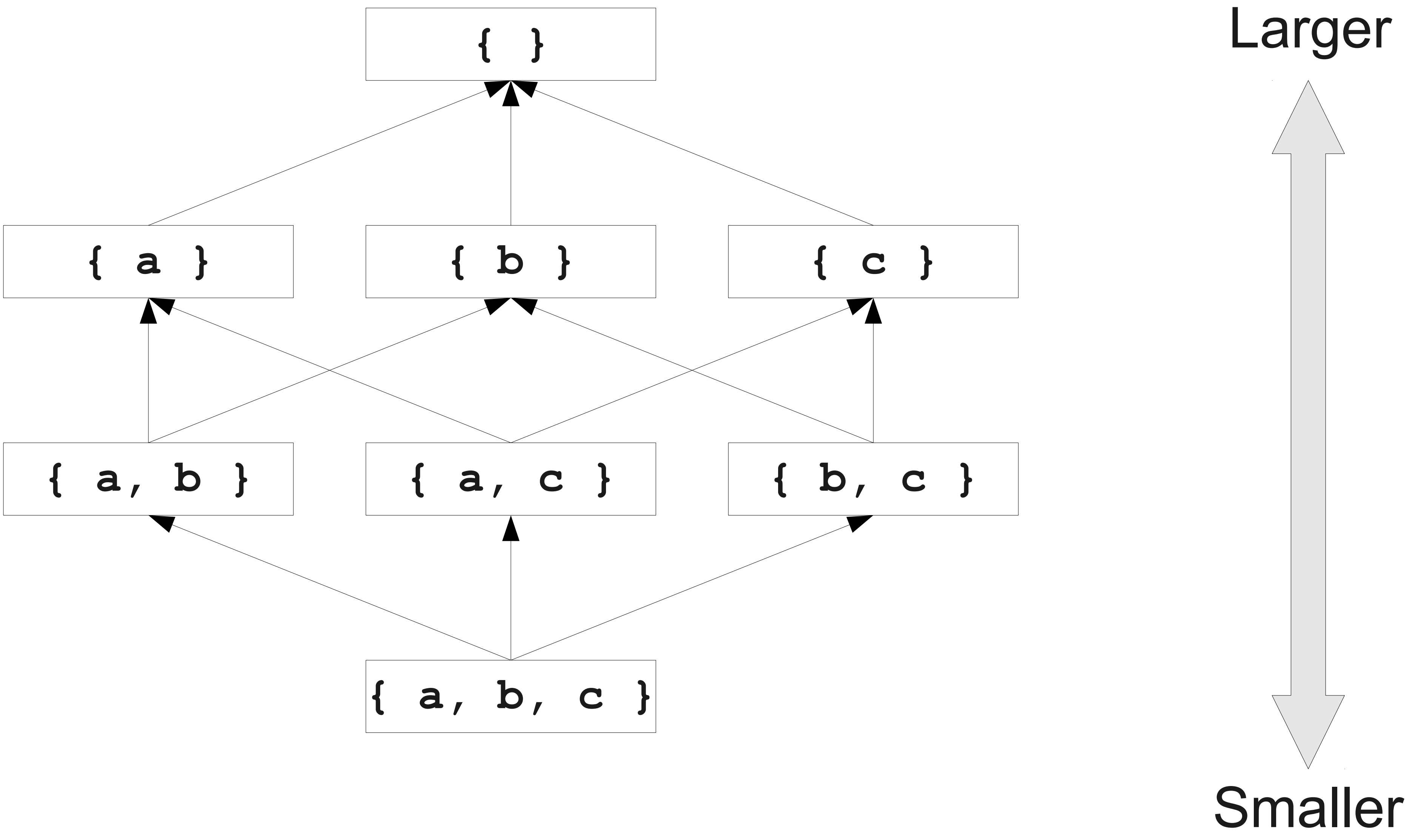
Pig

Is this a Meet Semilattice?

Is this a Meet Semilattice?



Meet Semilattices and Orderings



`{ }`

`{ a }` `{ b }` `{ c }`

`{ a, b }` `{ a, c }` `{ b, c }`

`{ a, b, c }`

Larger

Smaller

Meet Semilattices and Orderings

- Every meet semilattice (D, ∧) induces an ordering relationship ≤ over its elements.
- Define **x ≤ y iff x ∧ y = x**
- x ≤ x because x ∧ x = x
- x ≤ y and y ≤ x implies y = x because
  - x ≤ y means that x ∧ y = x.
  - y ≤ x means that y ∧ x = y.
  - By commutativity, x = x ∧ y = y ∧ x = y
- x ≤ y and y ≤ z means that x ≤ z because
  - x ≤ y means that x ∧ y = x.
  - y ≤ z means that y ∧ z = y
  - so x ∧ z = (x ∧ y) ∧ z = x ∧ (y ∧ z) = x ∧ 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
- What is the ordering relationship over this lattice?

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 relationship over this lattice?
  - x ≤ y iff x ∧ y = x iff **max**{x, y} = x iff x is a 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: $\{ \} \cup x = x$
- What is the ordering relationship over this lattice?

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: $\{ \} \cup x = x$
- What is the ordering relationship over this lattice?
  - $x \leq y$ iff $x \land y = x$ 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.
- 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.

Next Time

- **Using Semilattices**
  - The dataflow framework.
  - Global constant propagation.
  - Termination and correctness.
- **Code motion optimizations**
  - Loop-invariant code motion.
  - Partial redundancy elimination.