6.110 Computer Language Engineering

Re-lecture 4:
Program analysis and optimization

April 3rd, 2024

Terms and glossary ←
Local optimization
Global optimization
Implementation notes

Terms and glossary

- There are a lot of confusing/paradoxical terms in the literature.
- Optimization: Some "optimizations" actually make the program worse.
- Better call them "transformations"?

Terms and glossary

- Local optimization: Perform within each basic block only.
- Global optimization: Perform within a **function** only.
  - Not very "global."
  - We focus on dataflow optimizations in this class.
- Whole-program optimization
  - Interprocedural analysis/optimization

Terms and glossary

- Static analysis: Estimate, at compile time, what will happen at runtime
  - Need to take into account code paths that might not actually be executed.
- Dynamic analysis: Profiling, etc. at runtime
  - Counts toward the running time of the program
  - Must be *a priori* profitable to even bother checking
- Analyses enable transformations/optimizations!

Terms and glossary

- Biggest constraint: **safety**. Transformations must not change program behavior.

- What does it mean for program behavior to be the same?
  - Informally: Intuitive notion of observational equivalence
  - Formally: Need to define formal semantics and equivalence

Terms and glossary
Local optimization ←
Global optimization
Implementation notes

Local optimization

- Optimizations within each basic block only.
  - Straight-line code simplifies analysis.
  - No loops = perform analysis and transformation together.
  - Ad-hoc

- Stepping stones toward global optimization.
  - You need to understand straight-line code before you consider branches/loops.
  - Global optimization requires additional, more complicated dataflow analyses, but has the same idea.

Dead code elimination (DCE)

- Some definitions are useless.
  - **Definition (Def)**: assignment of an expression to a variable.
  - **Use**: a reference to a variable to use its value.
  - Literally *use*less!
  - Careful about global variables.

- Other optimizations (e.g. common subexpression elimination) introduce many temporaries.
  - Run DCE after those optimizations.

Dead code elimination (DCE)

- Basic idea
  - Run through the code backward
  - Maintain a set of variables that might be used after the current statement
  - Remove assignment to unused variables

Dead code elimination (DCE)

- Example: Assume only a is global.

```
a ← x+y     // {x,y,z}
t1 ← a      // {a,z}
b ← a+z     // {a,z}
t2 ← b      // {a,b}
c ← t1      // {a,b}
a ← b       // {b}
            // {a}
```

⇒

```
a ← x+y
b ← a+z
c ← a
a ← b
```

Dead code elimination (DCE)

- Extends naturally to global DCE.
  - Perform **liveness analysis** on control flow graph to figure out if a definition is used in any path. If not used, delete.
  - We'll see soon!

- There are also other sources of dead code besides unused variables.
  - Always-taken/skipped branches (recognized after constant propagation/folding)

Copy propagation (CP)

- If b←a and later statements use b, why not use a directly? Copy propagation automates this.

- Not useful on its own. Helps DCE.
  - Might be able to eliminate b←a

Copy propagation (CP)

- Idea:
  - Process the code in forward order.
  - Maintain `tmpToVar`: which variable to use instead of tmp
  - Maintain `varToTmps`: inverse of `tmpToVar`
  - Note: Might not need to differentiate temporaries and variables. The word "temporary" is just to make things easier to understand.

Copy propagation (CP)

- Algorithm:
  - When see `b`←`a`, set `tmpToVar[b]`←`a`.
  - When see a right hand side using `c`, replace with any element of `varToTmps[c]`.
  - Automatically maintain the inverse.

Copy propagation (CP)

- Example:

b ← a	b ← a
c ← b+1	c ← **a**+1
d ← b	d ← **a**
b ← d+c	b ← **a**+c
b ← d	b ← **a**

Copy propagation (CP)

- Edge case:

$a \leftarrow b$
$b \leftarrow c$
$d \leftarrow a$

$a \leftarrow b$
$b \leftarrow c$
$d \leftarrow b$???

Copy propagation (CP)

- The algorithm is not correct.
  - When see b←c, no `varToTmp[_]` should equal b.
  - Use `tmpToVars[b]` to figure out which entries to remove.
  - Alternatively, only do copy propagation for generated temps known to be immutable.

- This illustrates the pitfalls of ad-hoc algorithms.
  - Dataflow is much more disciplined.
  - Dataflow can be applied at statement level too!

Common subexpression elimination (CSE)

- Some expressions are used multiple times. We should be able to reuse the result from the first calculation.

a ← x+y

b ← **a+z**

b ← b+y

c ← **a+z**

Common subexpression elimination (CSE)

- Issue: Need to check if an expression is available

```
a ← x+y
b ← a+z
b ← b+y
c ← a+z  // wants to reuse line 2 but line 3 redefines b
```

Common subexpression elimination (CSE)

- Fix: Introduce new variables:

```
a ← x+y
b ← a+z
t ← b
b ← b+y
c ← t
```

Common subexpression elimination (CSE)

- Idea:
  - Uniquely identify each possible RHS expression (e.g. hash)
  - Keep track of **available expressions** (AE)
    - An expression becomes stale if one of its operands is re-defined
  - If the same expression appears and is available,
    - Introduce the temporary variable to store the expression
    - Use the temporary variable

Common subexpression elimination (CSE)

- CSE is usually implemented with a technique called **Local Value Numbering (LVN)**.

- CSE and LVN are *different*. Some literature treats them as the same and causes confusion.
  - Correct matchup: Local and global CSE using AEs.
  - Acceptable: Local and global "CSE" using LVN and GVN.
  - Many books/courses (including this):
    - Local "CSE" uses LVN
    - Global CSE uses AE.
    - Different techniques!

Local Value Numbering (LVN)

- **This is the one that's taught in lecture.**
  - Know this for the exam.

- Assign a number to represent a possible value. Propagate that number to show where that (same) value ends up.

- Also generate new temporaries to store the subexpressions.

Local Value Numbering (LVN)

- Example:

```
a  ←  x+y    // x→v1, y→v2, v1+v2→v3, a→v3
t1 ←  a      // v3→t1
b  ←  a+z    // z→v4, v3+v4→v5, b→v5
t2 ←  b      // v5→t2
b  ←  b+y    // v5+v2→v6, b→v6
t3 →  b      // v6→t3
c  ←  a+z t2
```

Local Value Numbering (LVN)

- LVN can give you the effect of other opts (partially!)
  - Constant propagation
  - Copy propagation
  - "there are cases where value numbering is more powerful than any of the [..] others and cases where each of them is more powerful than value numbering" (whale book)

$y \leftarrow a+b$

$x \leftarrow b$

$z \leftarrow a+x$

Common subexpression elimination (CSE)

- CSE (whether with AE or LVN) introduces a lot of temporaries and copies.
  - Run CP after CSE. (Might do a few rounds.)
  - Run DCE after CP.

Local optimization

- For the exam, you should be able to:
  - Manually perform each of these optimizations given a piece of code
  - Give example pieces of code that illustrate these optimizations

Terms and glossary
Local optimization
Global optimization ←
Implementation notes

Dataflow analysis

- Local optimization but it's not local.
- What's different?
  - Branches
  - Need to combine results from multiple predecessors
  - Need quantification ("there exists a path" or "for all paths")

Liveness

- Determine if a variable is "live" at a given point in CFG
  - Sometimes called "Live-Variable Analysis"

- "Live" means the current definition of that variable has a future use (up to the exit block) without a redefinition in between.

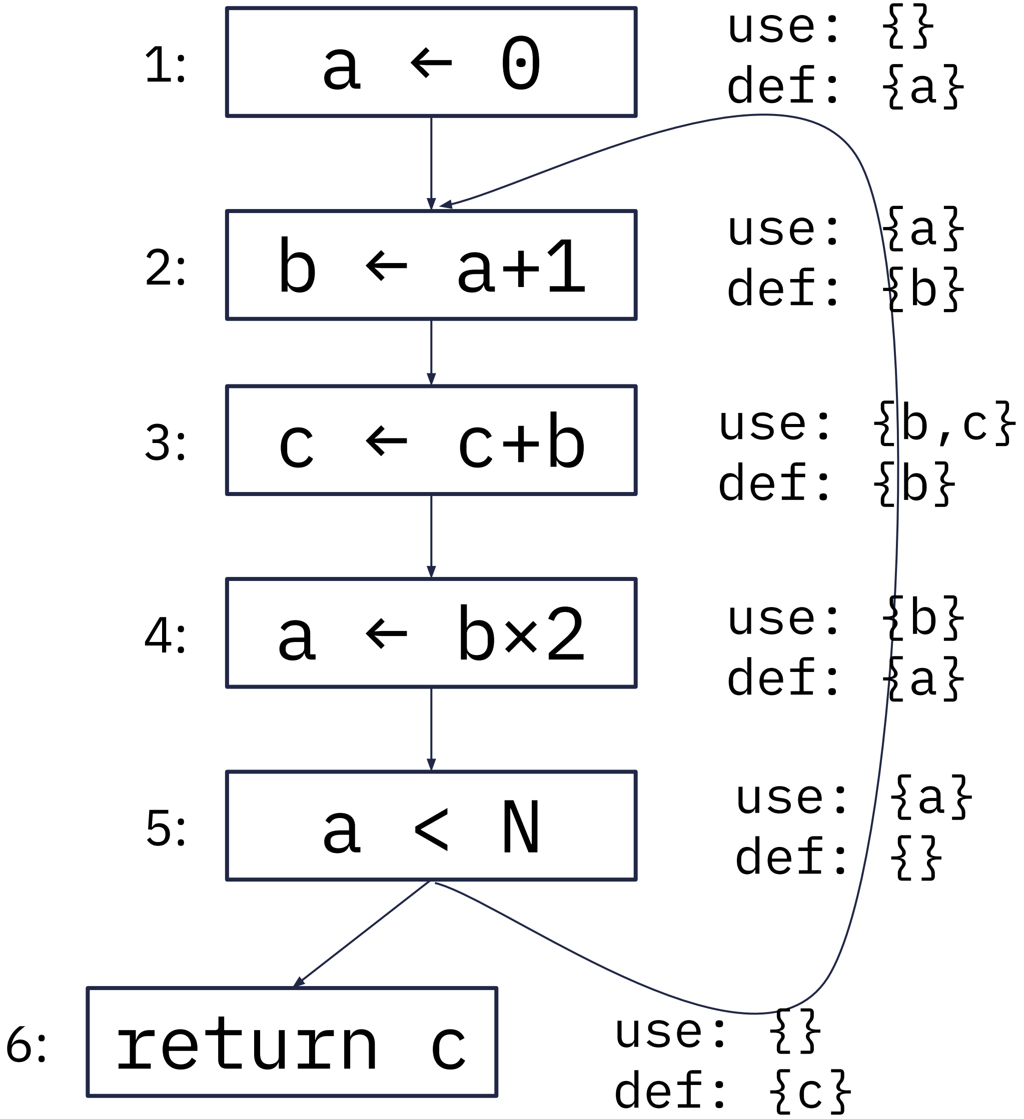
- Global variables are live at the end
- Local variables are dead at the end

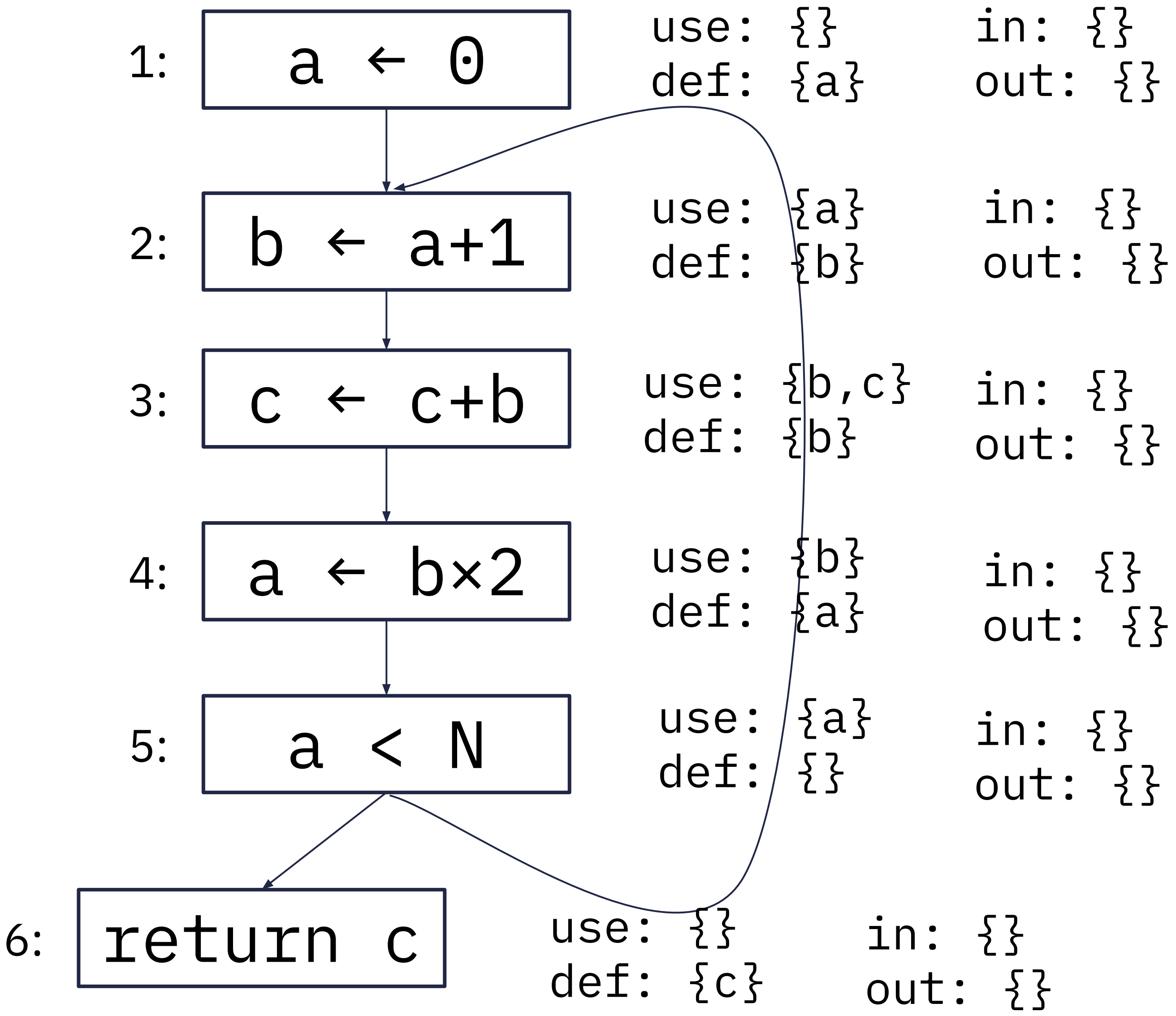
Liveness

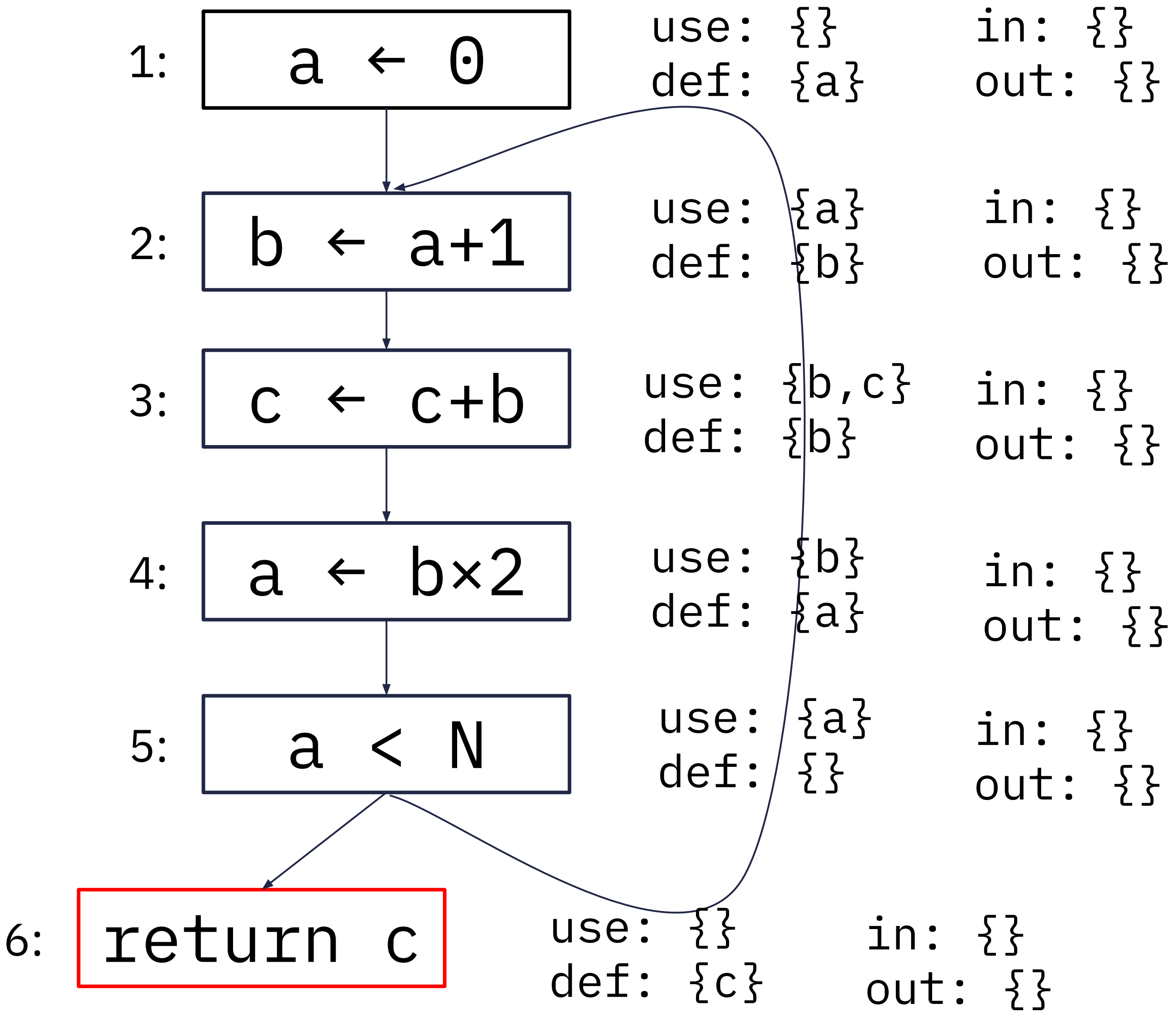
- Initialize those sets:
  - `use[`B_i`]` = variables used in B_i
  - `def[`B_i`]` = variables (re-)defined in B_i

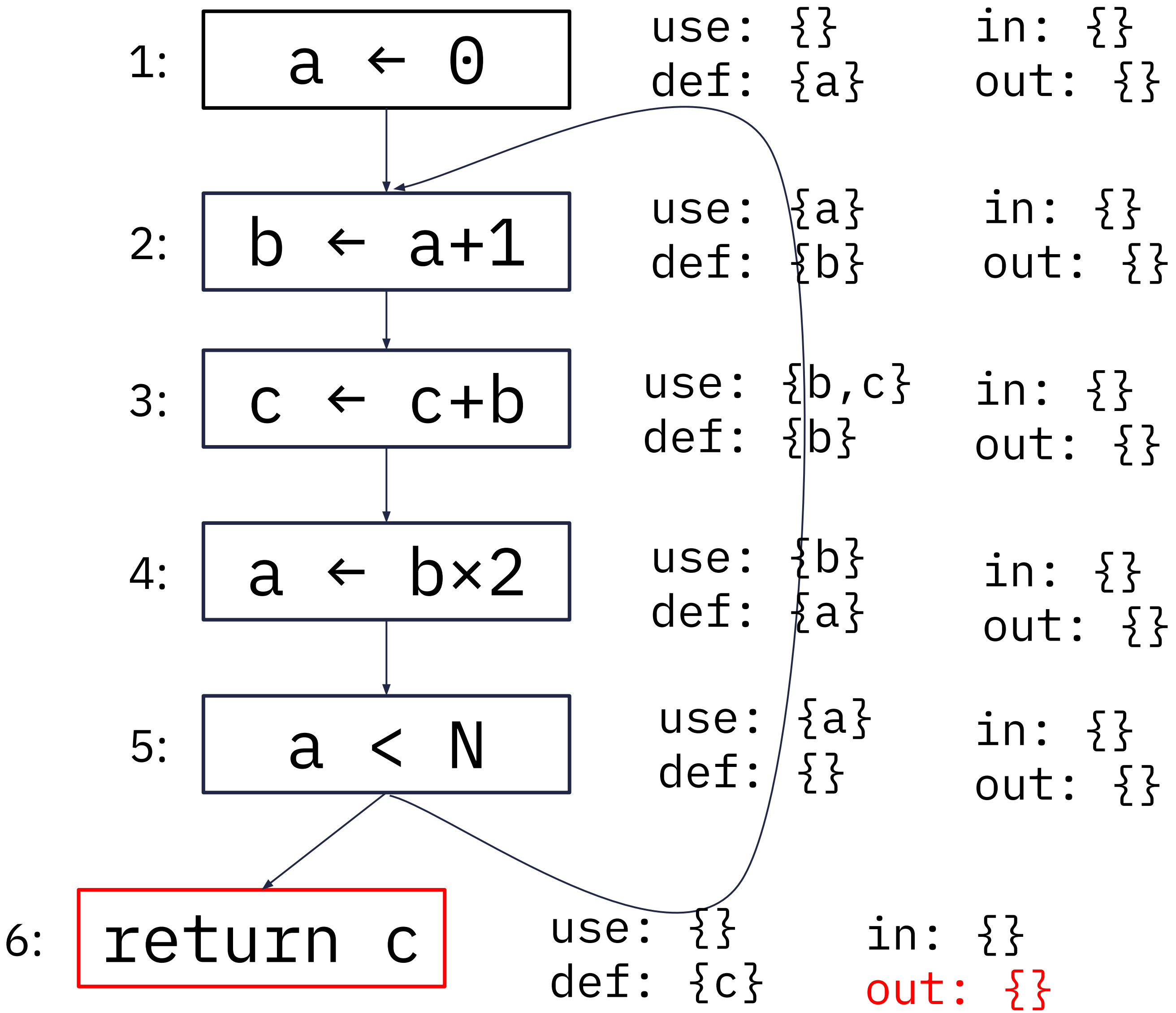
- Solve for
  - `in[`B_i`]` = live variables at the start of B_i
  - `out[`B_i`]` = live variables at the end of B_i

- Initially assume no variables are live (except globals at the end)


1:
a ← 0
use: {}
def: {a}
2:
b ← a+1
use: {a}
def: {b}
3:
c ← c+b
use: {b,c}
def: {b}
4:
a ← b×2
use: {b}
def: {a}
5:
a < N
use: {a}
def: {}
6:
return c
use: {}
def: {c}

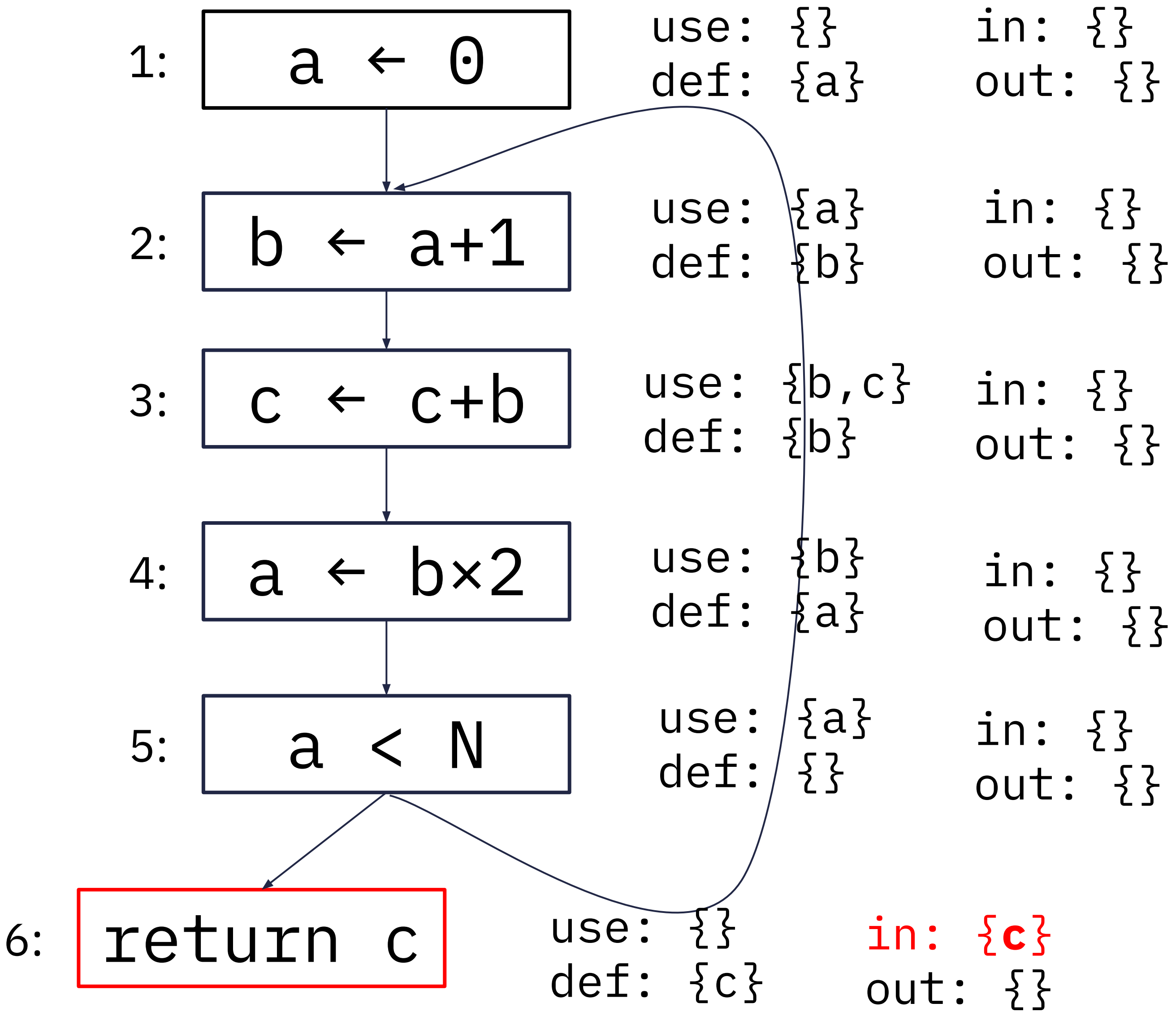

1:
a ← 0
use: {}
def: {a}
in: {}
out: {}
2:
b ← a+1
use: {a}
def: {b}
in: {}
out: {}
3:
c ← c+b
use: {b,c}
def: {b}
in: {}
out: {}
4:
a ← b×2
use: {b}
def: {a}
in: {}
out: {}
5:
a < N
use: {a}
def: {}
in: {}
out: {}
6:
return c
use: {}
def: {c}
in: {}
out: {}


1:
a ← 0
use: {}
def: {a}
in: {}
out: {}
2:
b ← a+1
use: {a}
def: {b}
in: {}
out: {}
3:
c ← c+b
use: {b,c}
def: {b}
in: {}
out: {}
4:
a ← b×2
use: {b}
def: {a}
in: {}
out: {}
5:
a < N
use: {a}
def: {}
in: {}
out: {}
6:
return c
use: {}
def: {c}
in: {}
out: {}



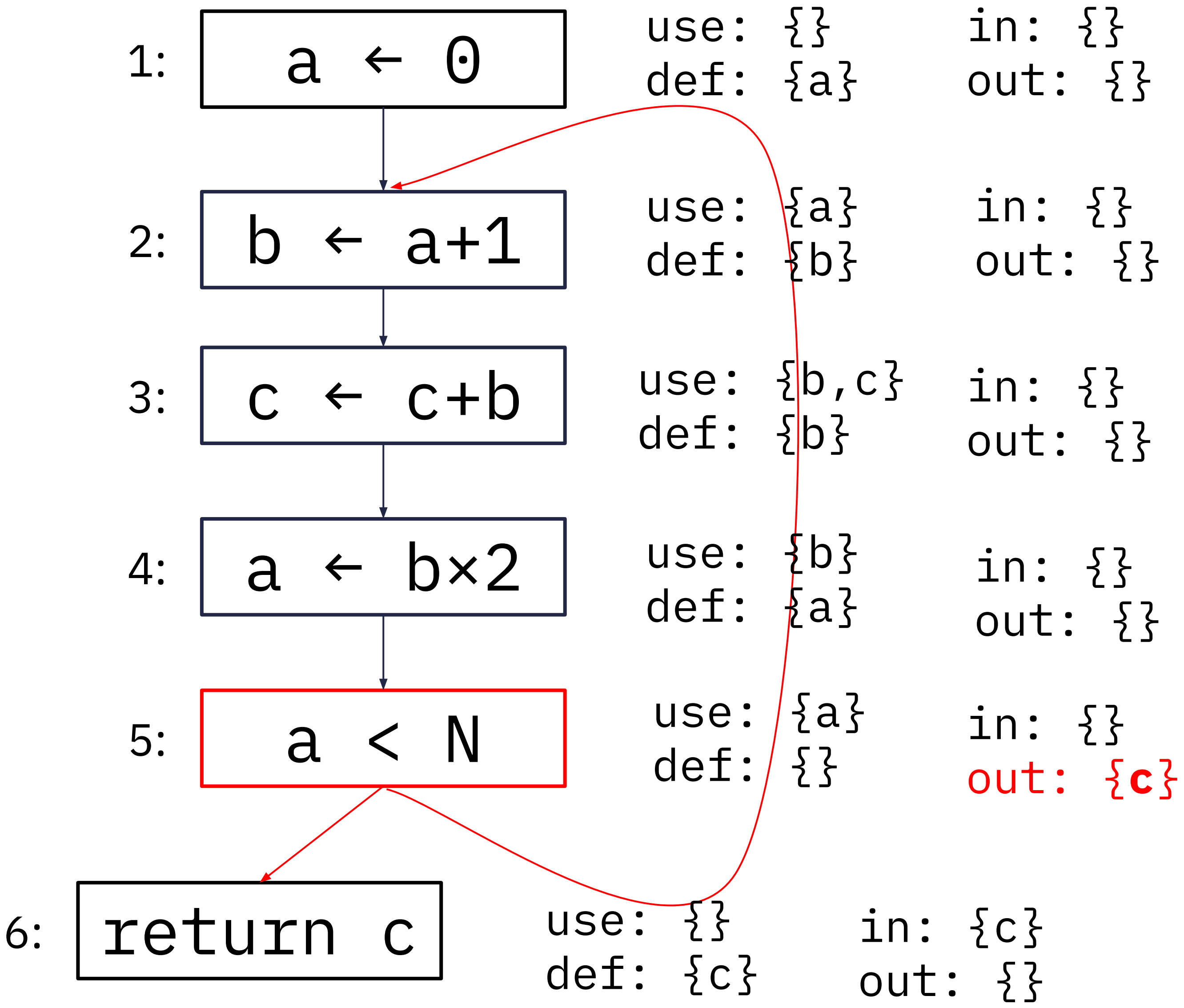
out[B_i] = ∪(in[s]; s successor of B_i)

$$\mathrm{in}[B_i] = \mathrm{use}[B_i] \cup (\mathrm{out}[B_i] - \mathrm{def}[B_i])$$



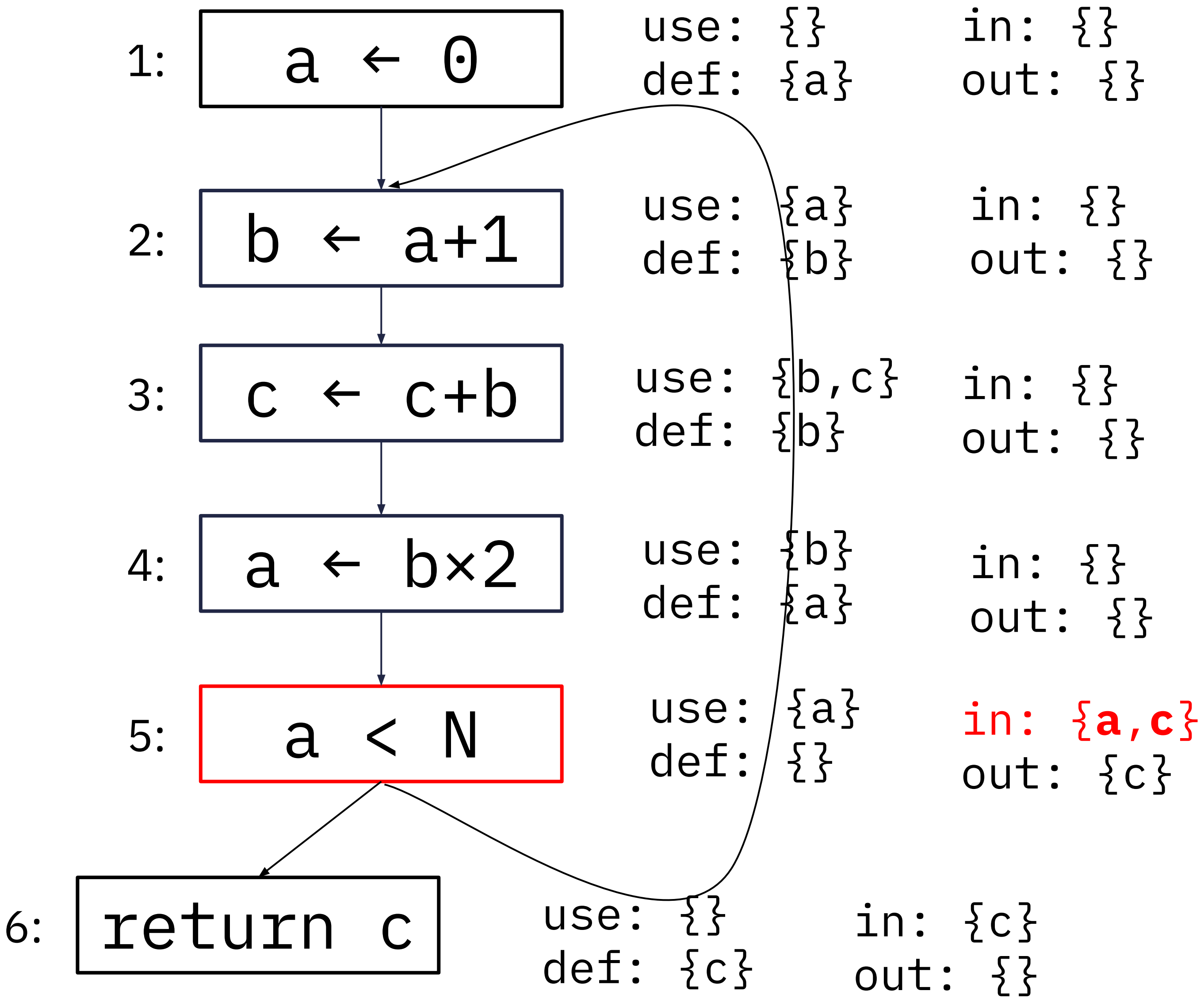
$$\mathrm{out}[B_i] = \cup(\mathrm{in}[s];\ s \text{ successor of } B_i)$$

$\text{in}[B_i] = \text{use}[B_i] \cup (\text{out}[B_i] - \text{def}[B_i])$



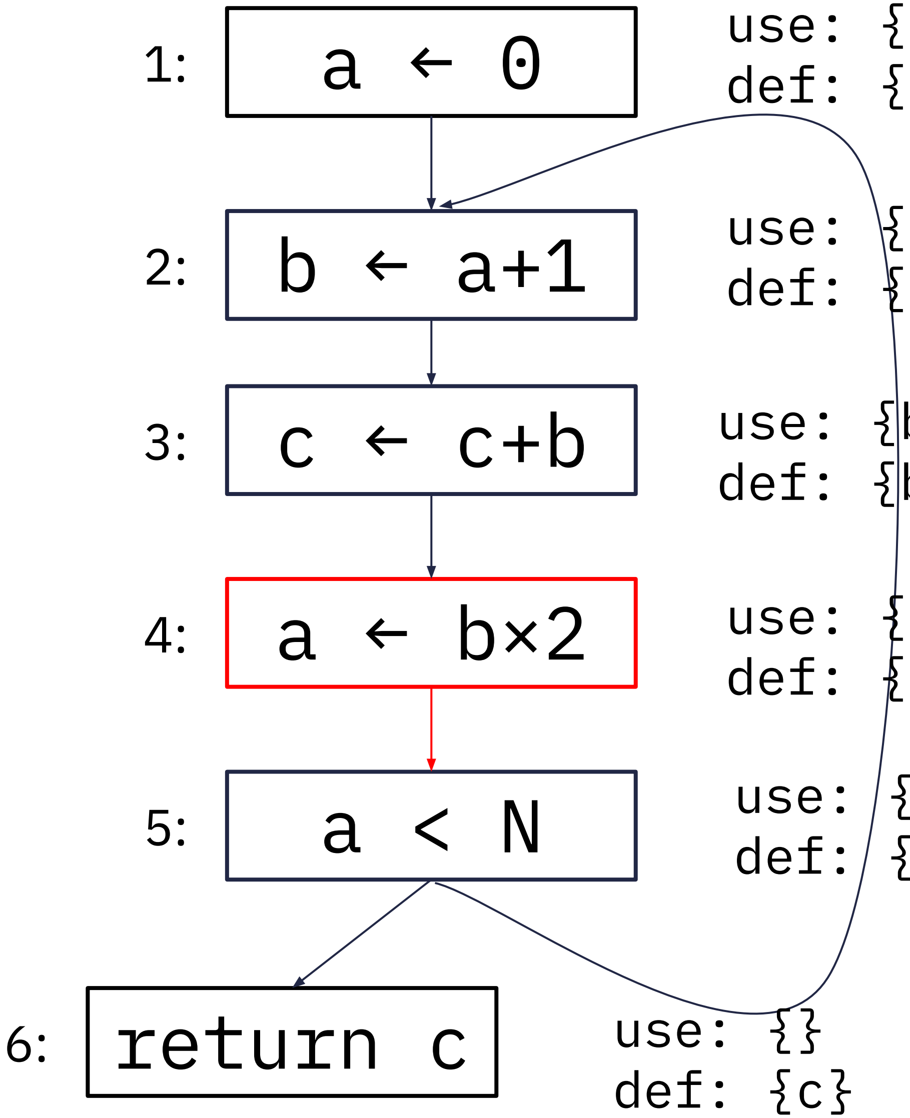
$\textbf{out}[B_i] = \cup(\textbf{in}[s];\ s \textbf{ successor of } B_i)$

$$\mathbf{in}[B_i] = \mathbf{use}[B_i] \cup (\mathbf{out}[B_i] - \mathbf{def}[B_i])$$



$$\mathrm{out}[B_i] = \cup(\mathrm{in}[s];\ s \text{ successor of } B_i)$$

$in[B_i] = use[B_i] \cup (out[B_i] - def[B_i])$



1: use: {} def: {a} — in: {} out: {}

2: use: {a} def: {b} — in: {} out: {}

3: use: {b,c} def: {b} — in: {} out: {}

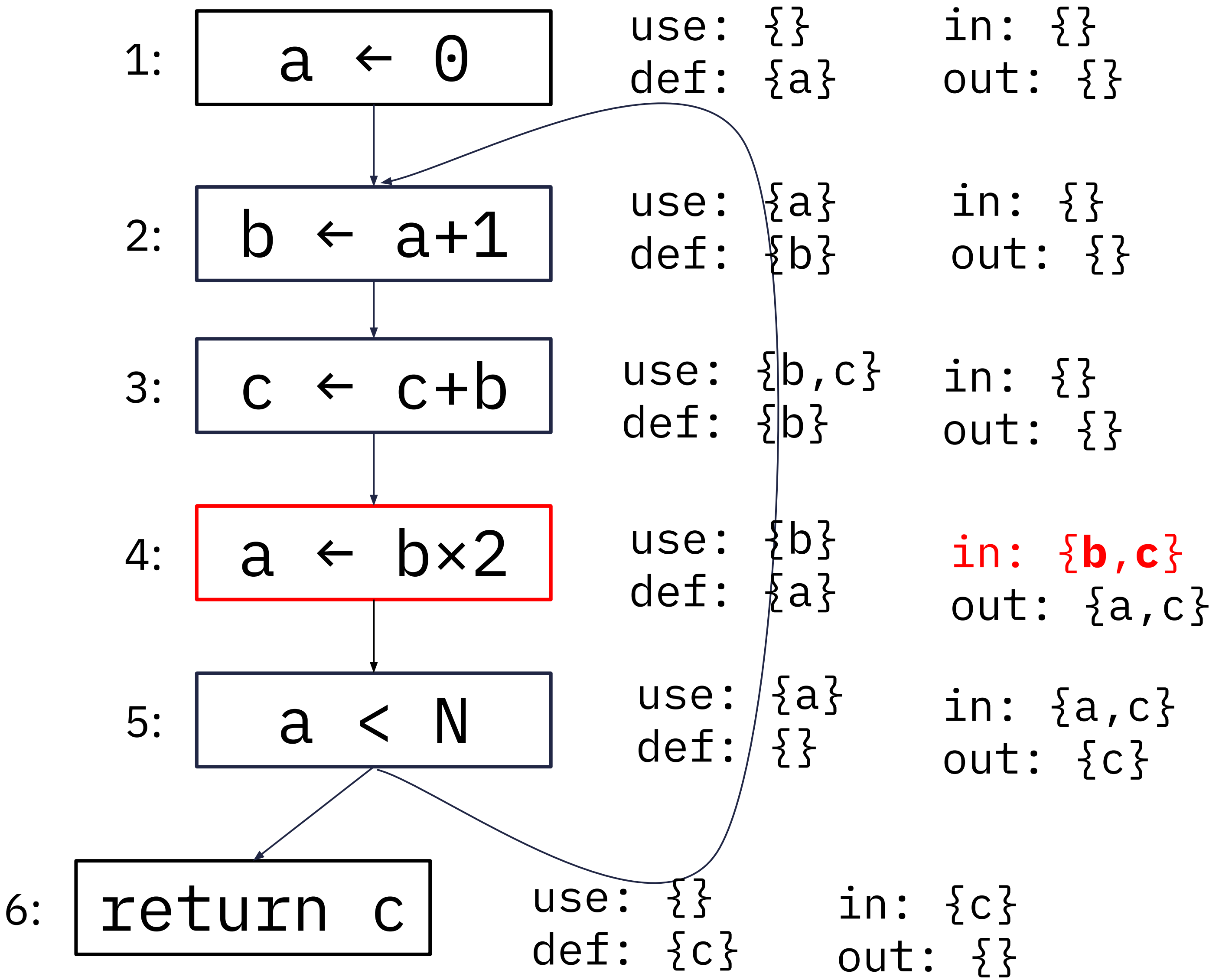
4: use: {b} def: {a} — in: {} out: **{a,c}**

5: use: {a} def: {} — in: {a,c} out: {c}

6: use: {} def: {c} — in: {c} out: {}

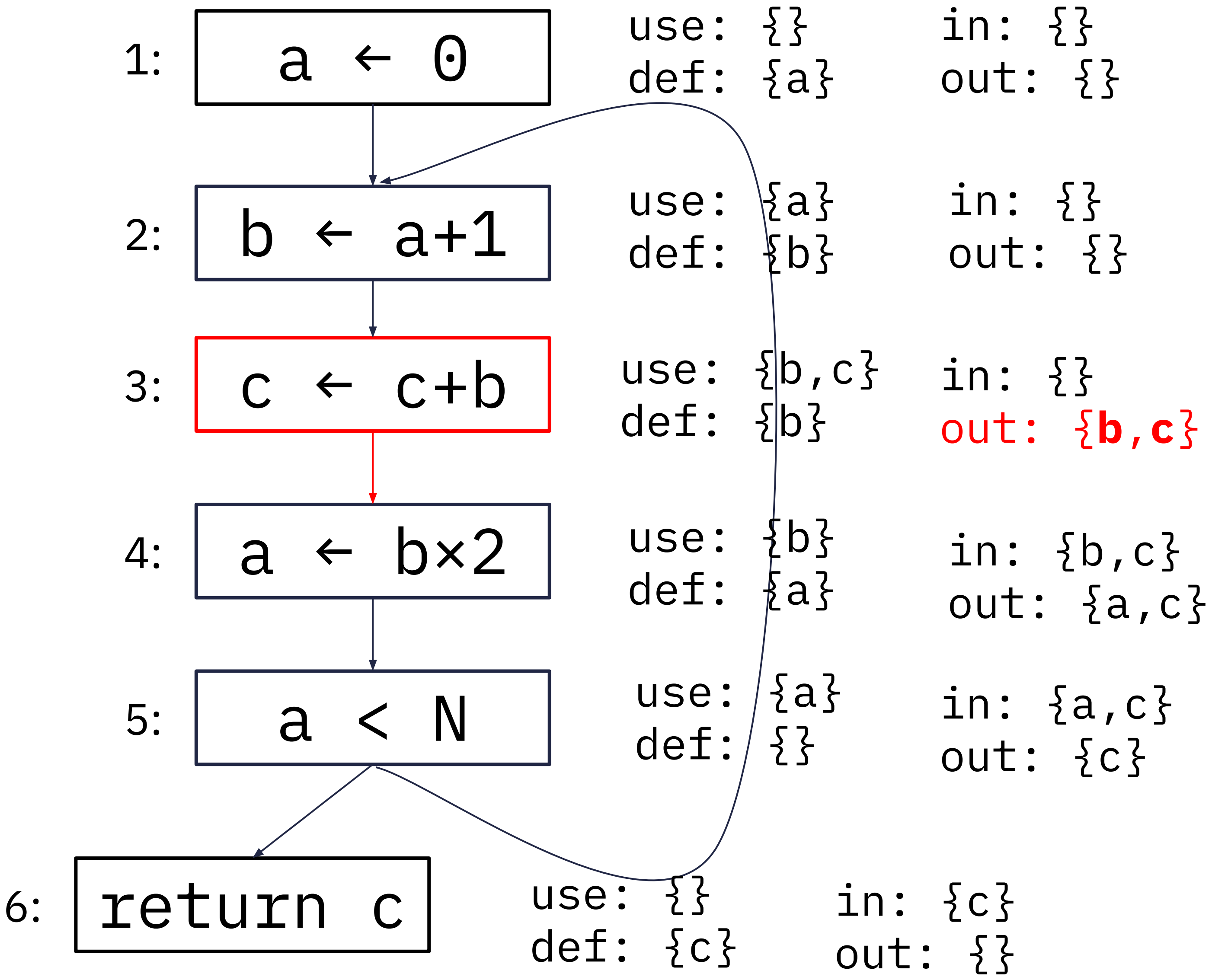
$\mathbf{out[B_i] = \cup(in[s];\ s\ successor\ of\ B_i)}$

$$in[B_i] = use[B_i] \cup (out[B_i] - def[B_i])$$



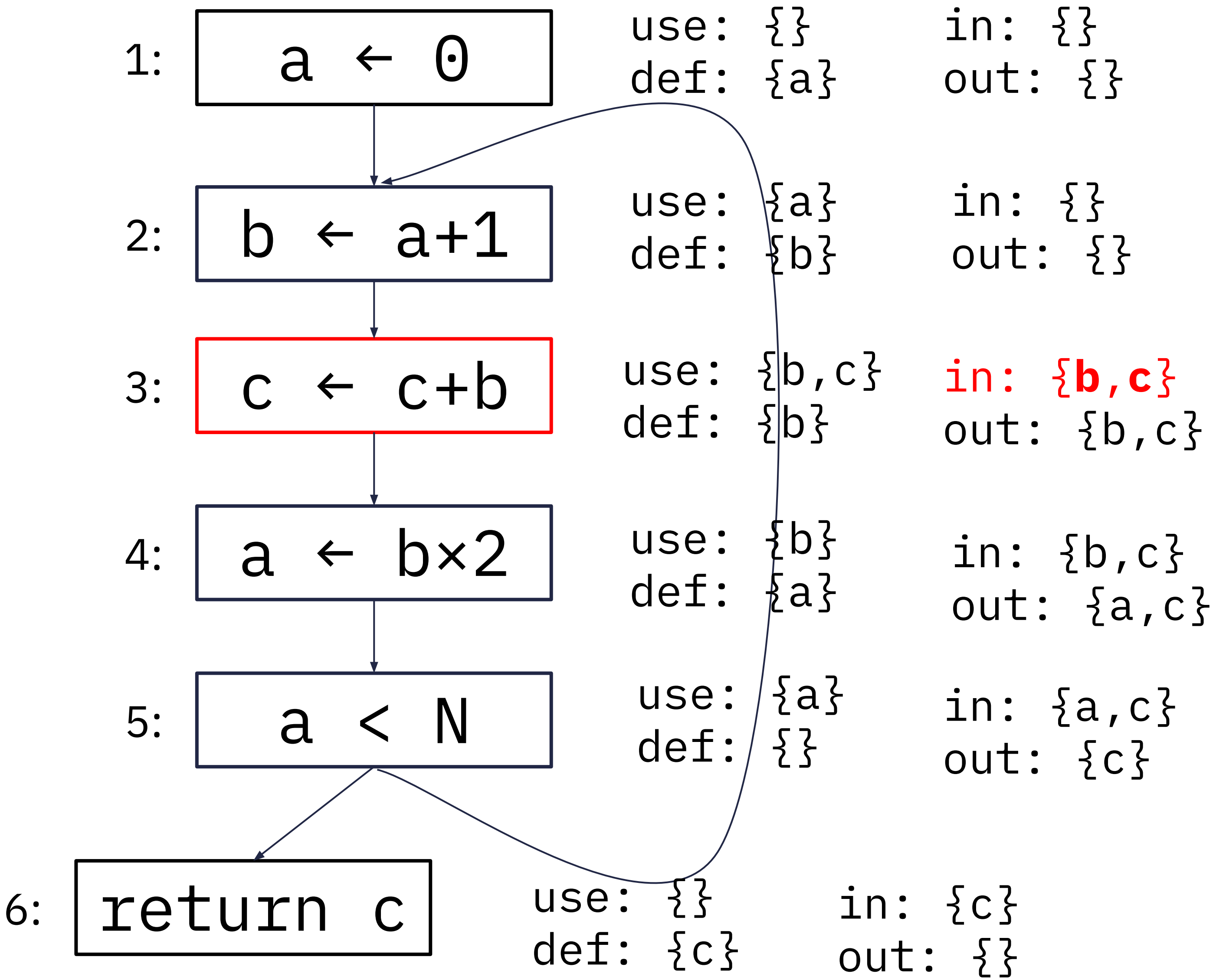
$$out[B_i] = \cup(in[s];\ s \text{ successor of } B_i)$$

$\mathtt{in}[B_i] = \mathtt{use}[B_i] \cup (\mathtt{out}[B_i] - \mathtt{def}[B_i])$



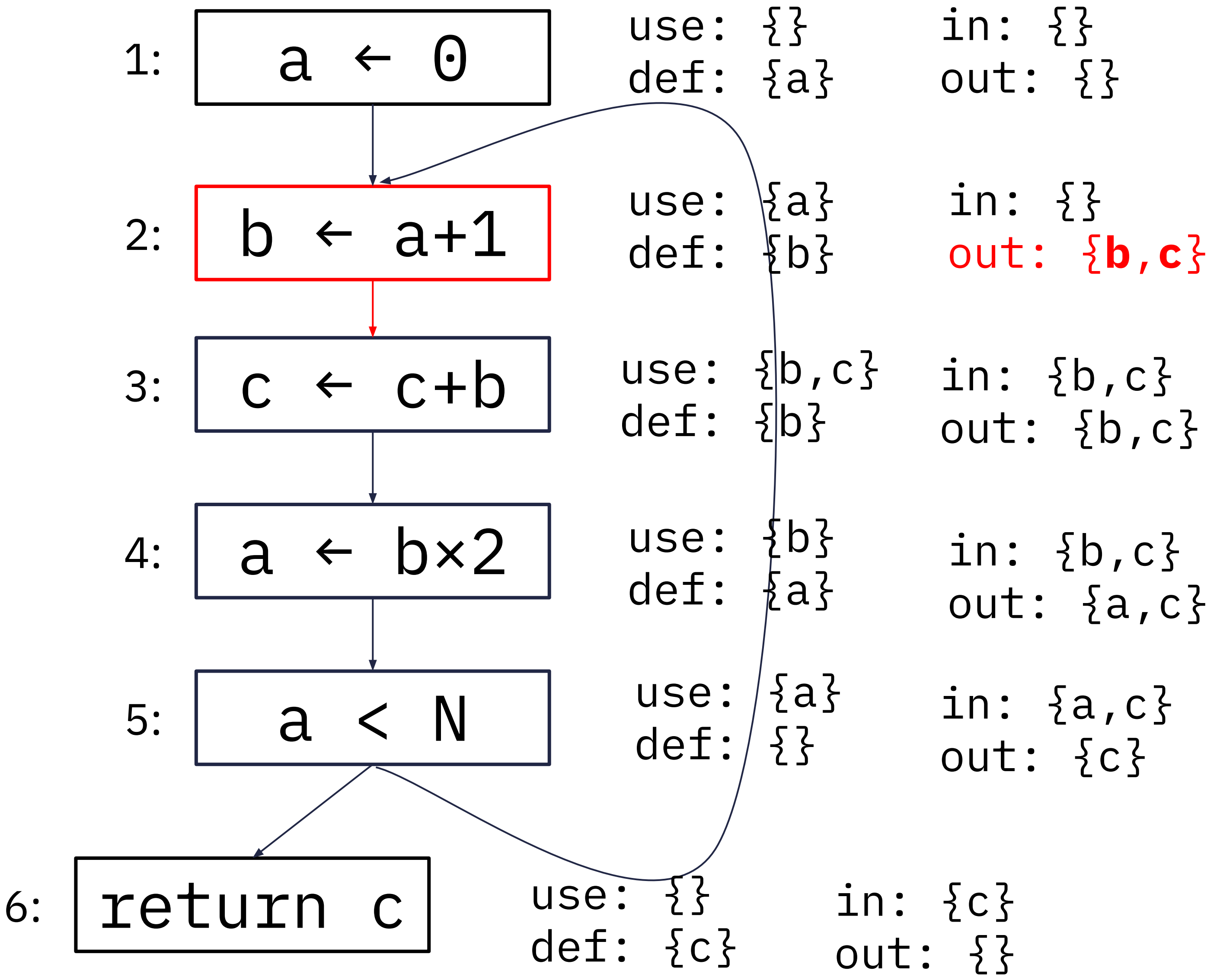
$\mathbf{out}[B_i] = \cup(\mathbf{in}[s];\ \mathbf{s\ successor\ of}\ B_i)$

$$in[B_i] = use[B_i] \cup (out[B_i] - def[B_i])$$



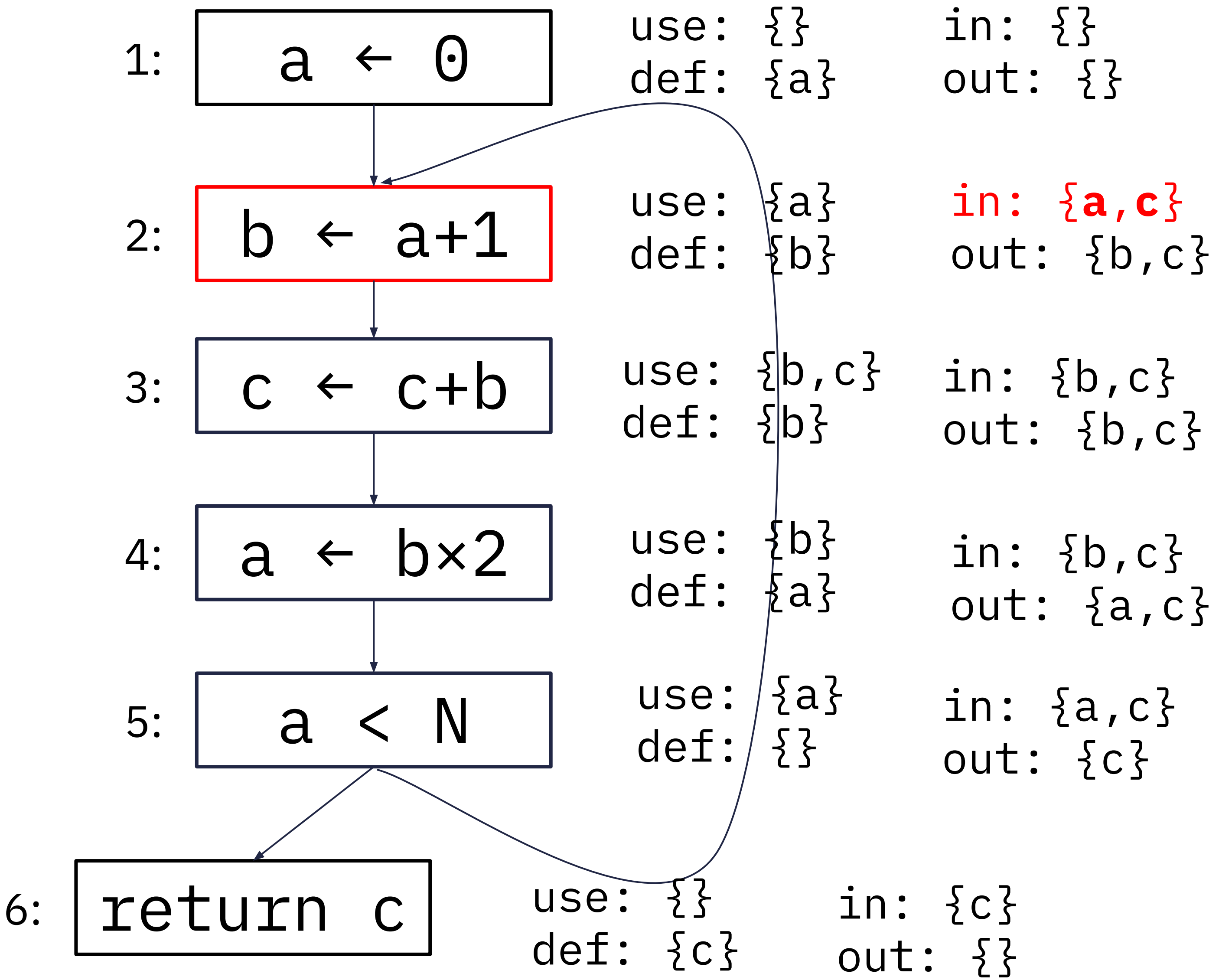
$$out[B_i] = \cup(in[s];\ s \text{ successor of } B_i)$$

$\mathtt{in}[B_i] = \mathtt{use}[B_i] \cup (\mathtt{out}[B_i] - \mathtt{def}[B_i])$



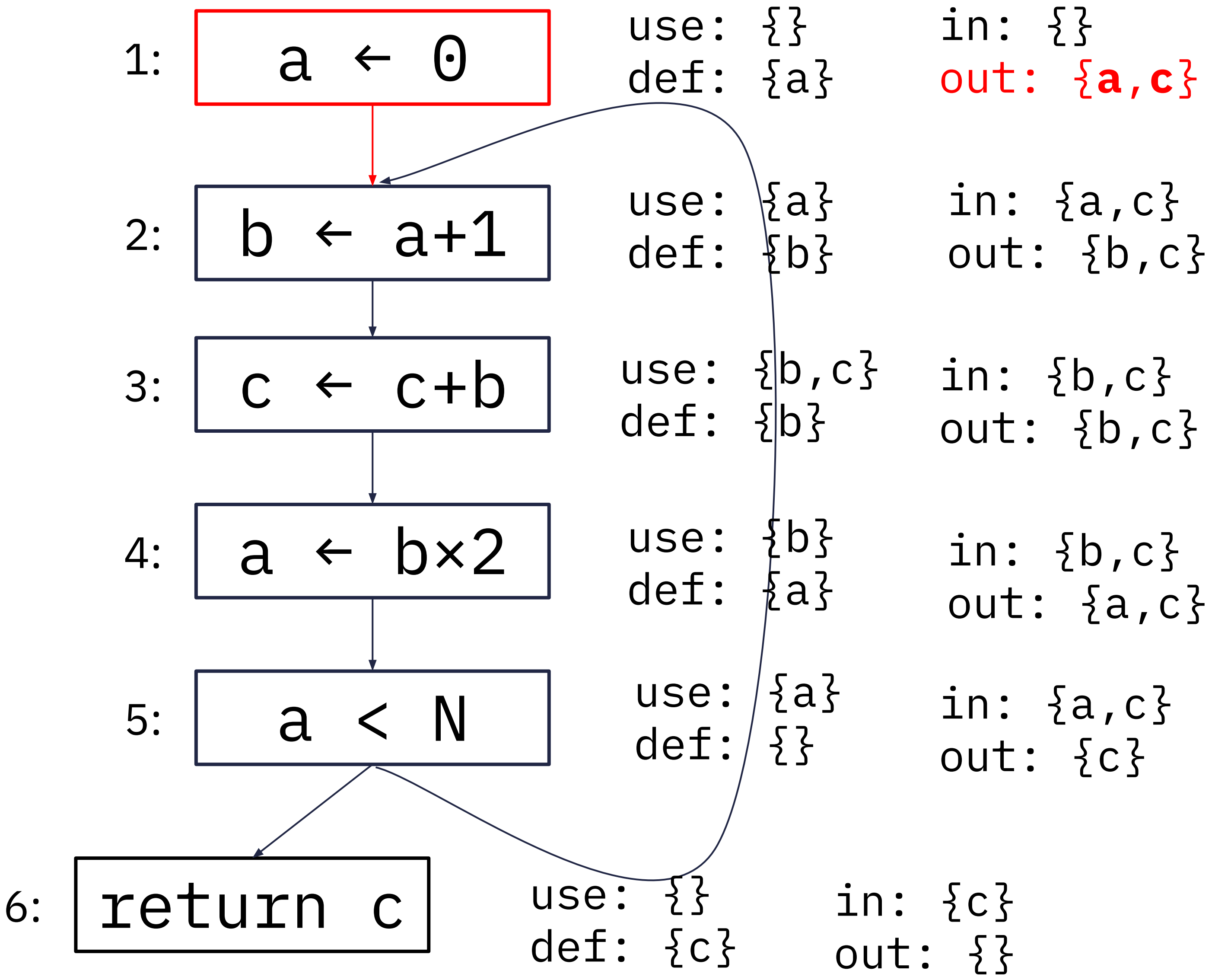
$\mathbf{out}[B_i] = \bigcup(\mathbf{in}[s];\ \mathbf{s\ successor\ of}\ B_i)$

$\mathbf{in[B_i] = use[B_i] \cup (out[B_i] - def[B_i])}$



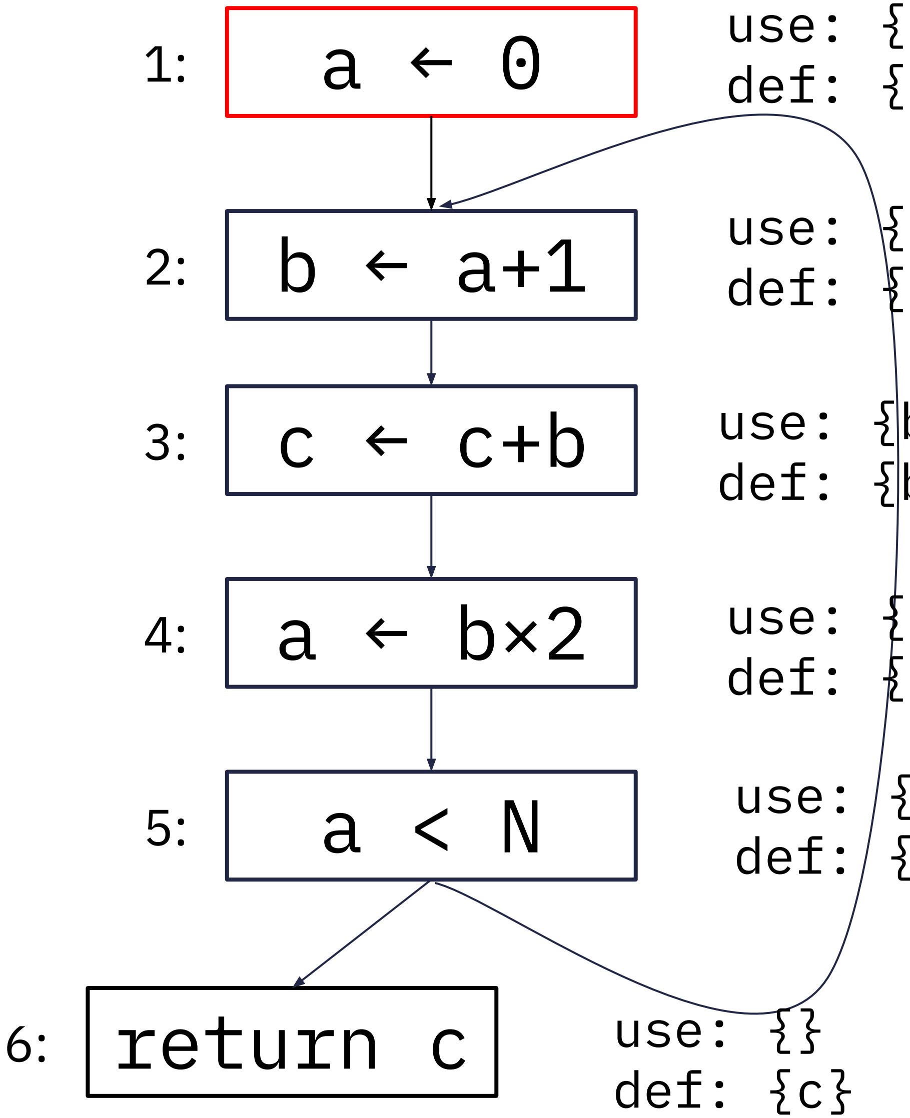
$out[B_i] = \cup(in[s];\ s \text{ successor of } B_i)$

$\texttt{in}[B_i] = \texttt{use}[B_i] \cup (\texttt{out}[B_i] - \texttt{def}[B_i])$



$\texttt{out}[B_i] = \cup(\texttt{in}[s];\ s\ \texttt{successor of}\ B_i)$

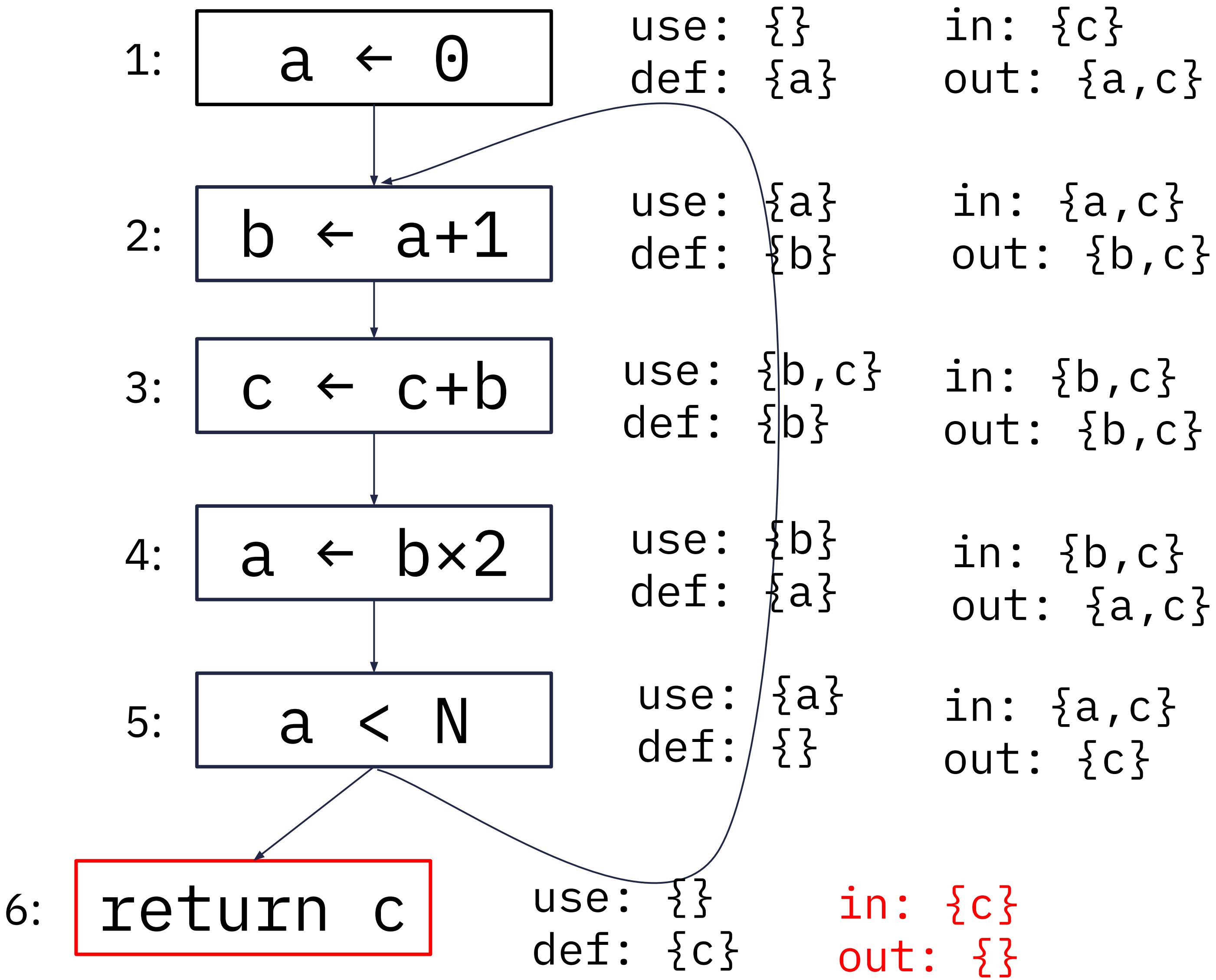
$in[B_i] = use[B_i] \cup (out[B_i] - def[B_i])$



Block	use	def	in	out
1	{}	{a}	{**c**}	{a,c}
2	{a}	{b}	{a,c}	{b,c}
3	{b,c}	{b}	{b,c}	{b,c}
4	{b}	{a}	{b,c}	{a,c}
5	{a}	{}	{a,c}	{c}
6	{}	{c}	{c}	{}

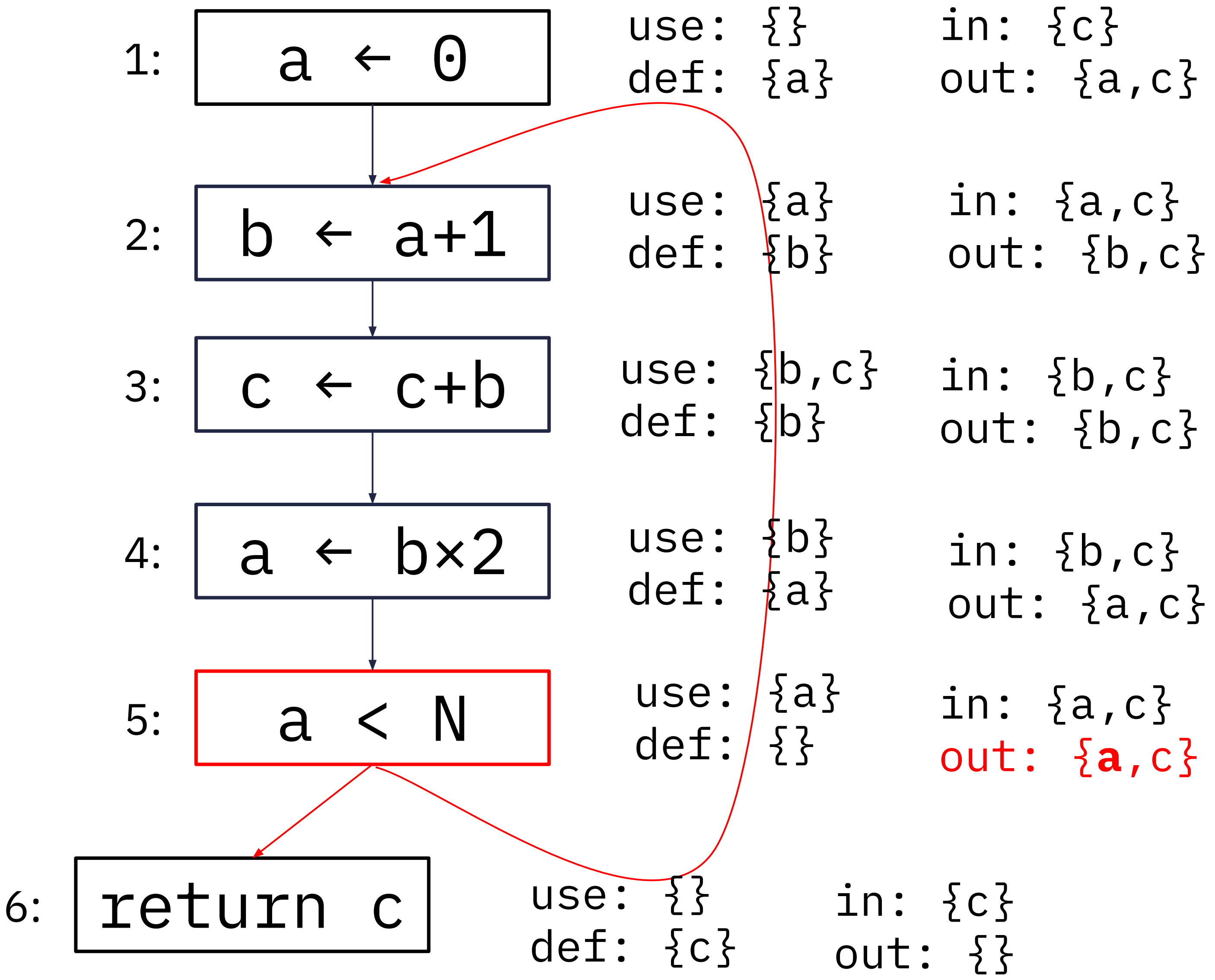
$\mathbf{out[B_i] = \cup(in[s];\ s\ successor\ of\ B_i)}$

$$\mathrm{in}[B_i] = \mathrm{use}[B_i] \cup (\mathrm{out}[B_i] - \mathrm{def}[B_i])$$



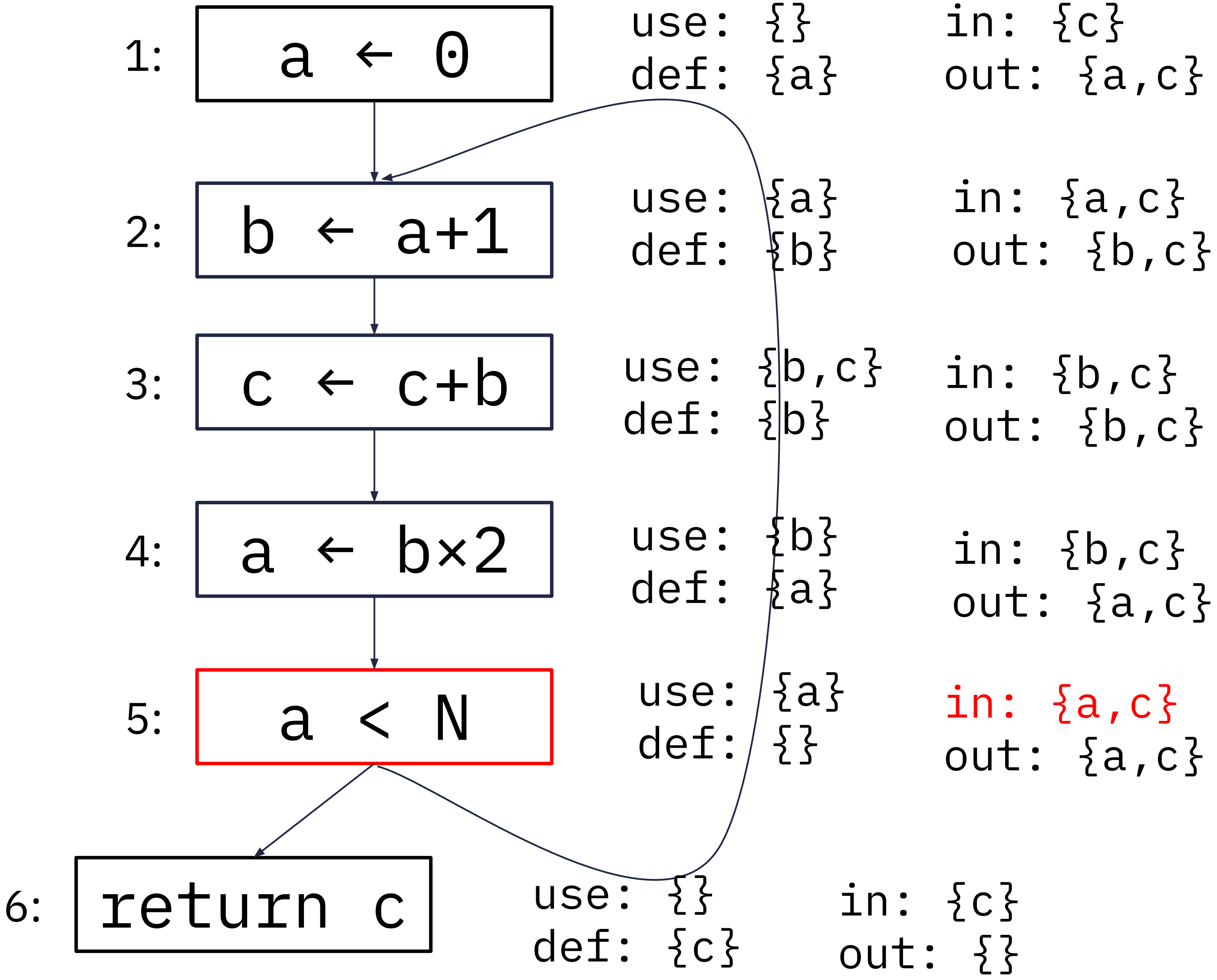
$$\mathrm{out}[B_i] = \cup(\mathrm{in}[s];\ s \text{ successor of } B_i)$$

$in[B_i] = use[B_i] \cup (out[B_i] - def[B_i])$



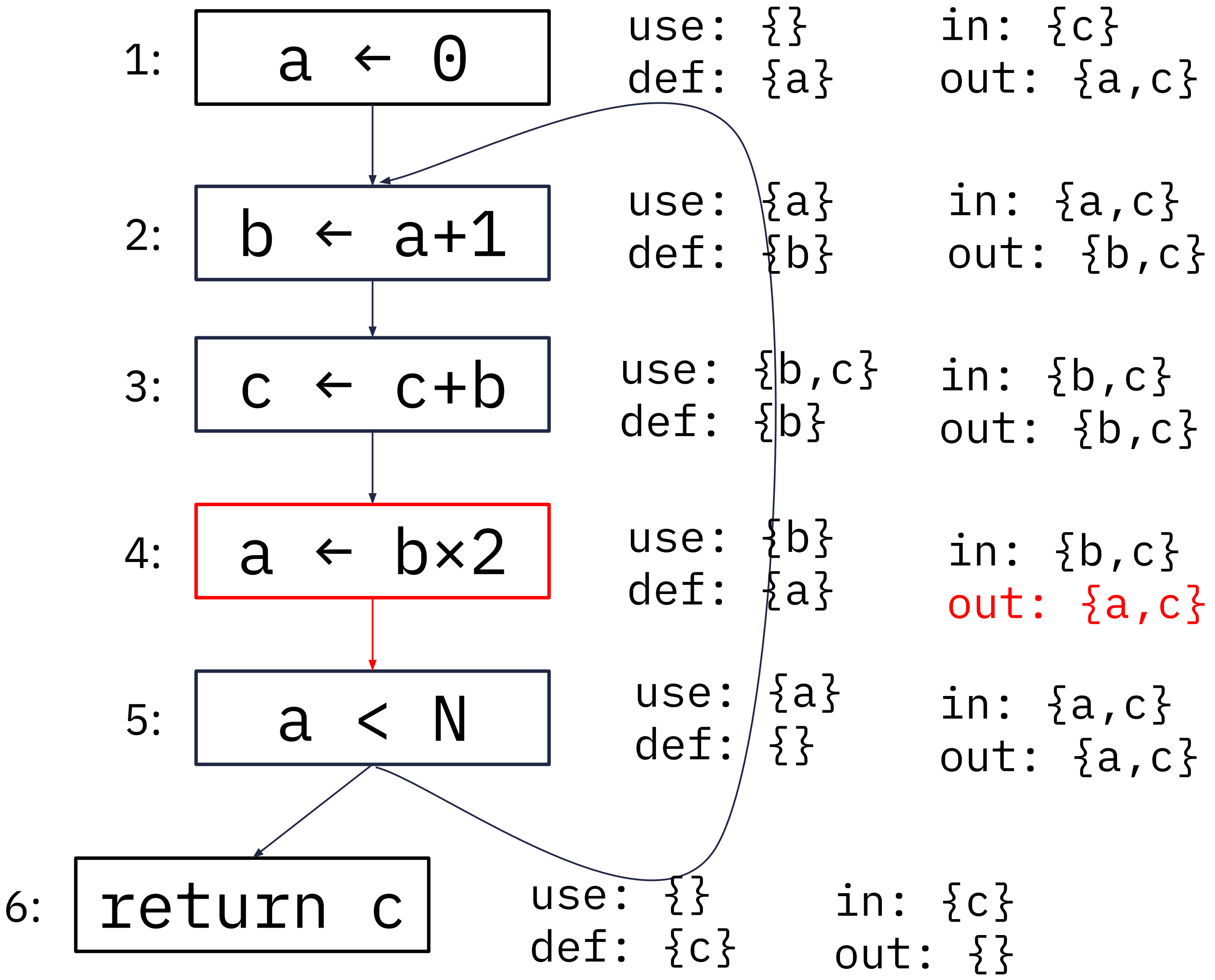
$out[B_i] = \cup(in[s];\ s \text{ successor of } B_i)$

$$\mathbf{in[B_i] = use[B_i] \cup (out[B_i] - def[B_i])}$$



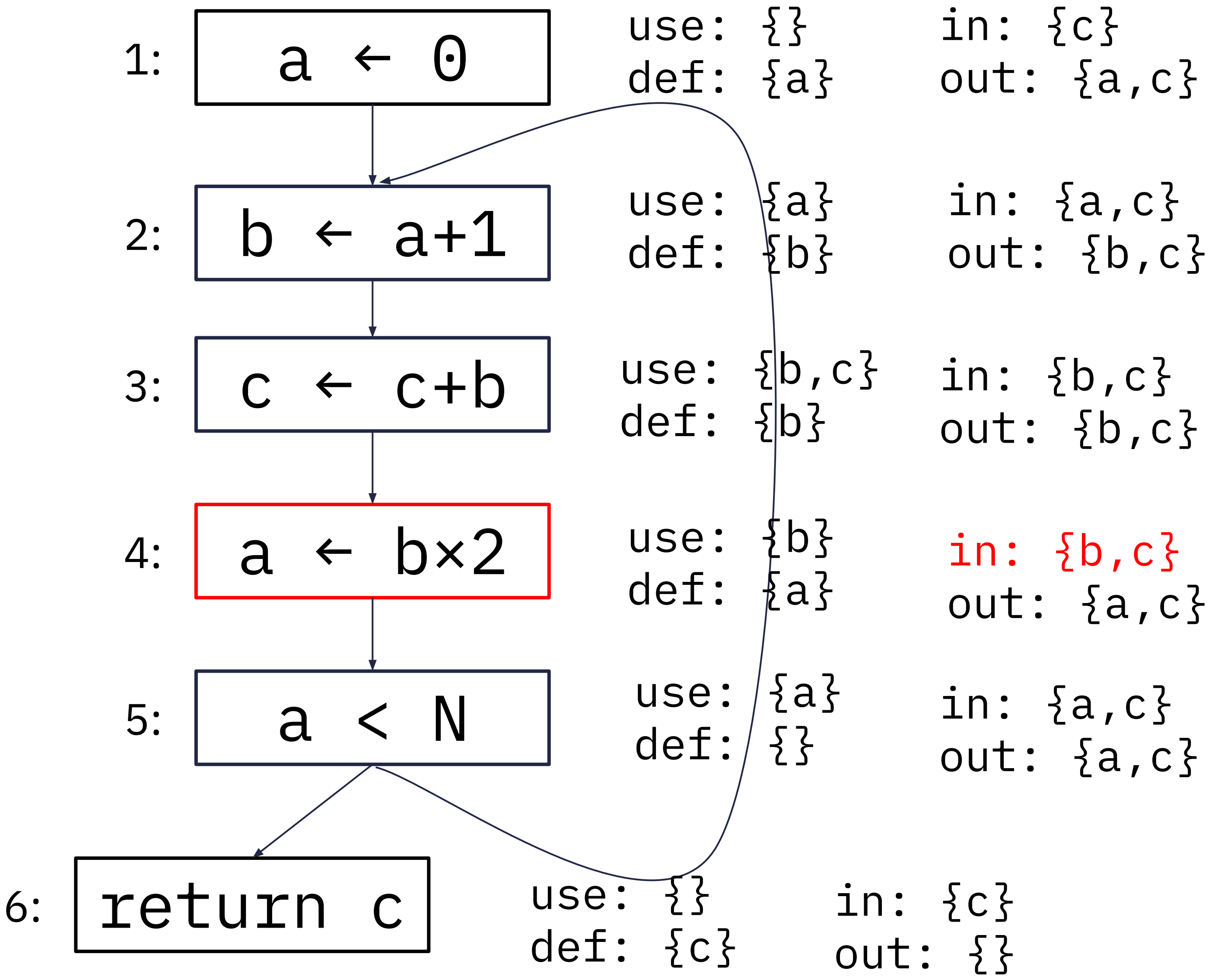
$$out[B_i] = \cup(in[s];\ s \text{ successor of } B_i)$$

$\text{in}[B_i] = \text{use}[B_i] \cup (\text{out}[B_i] - \text{def}[B_i])$



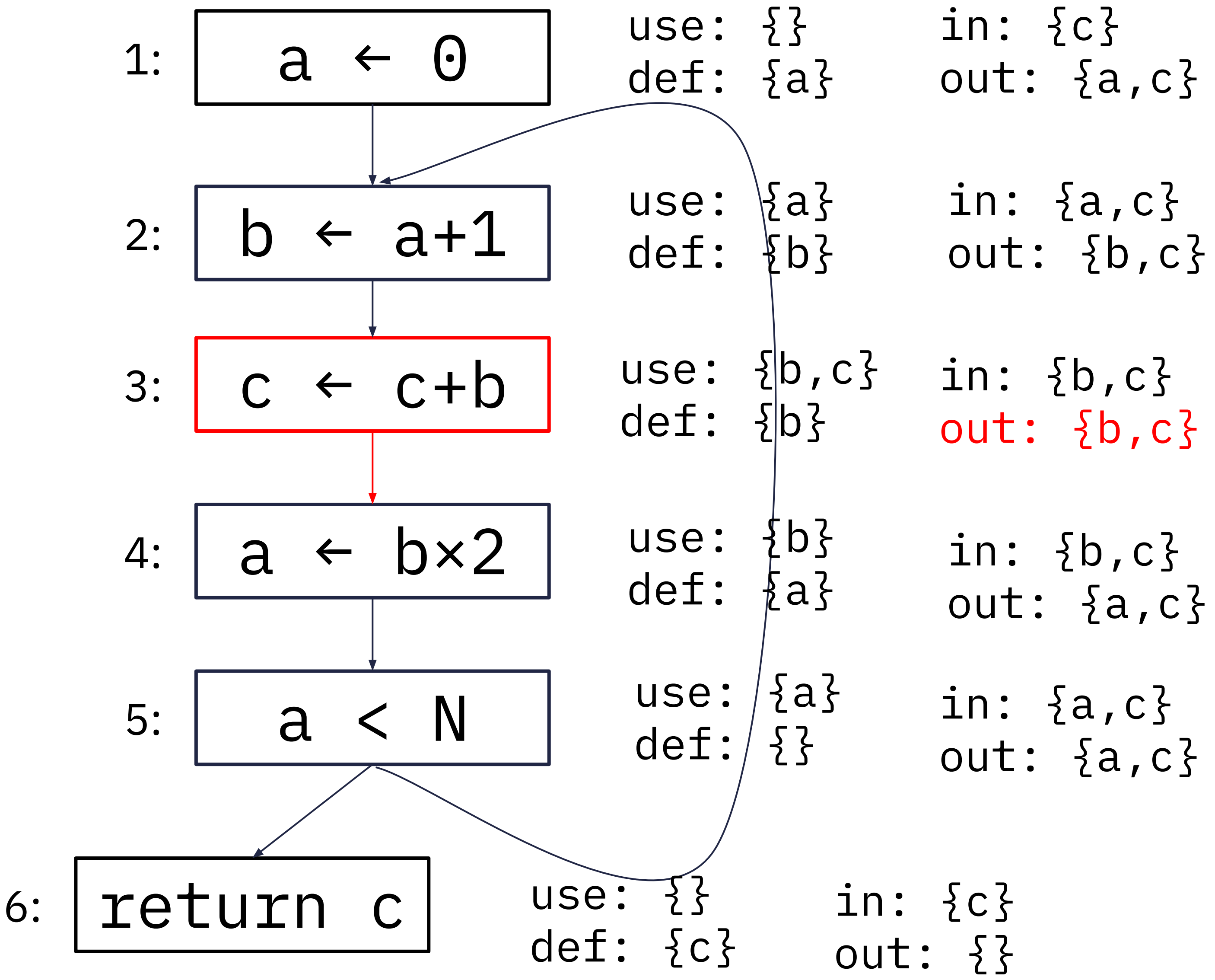
$\mathbf{out}[B_i] = \bigcup(\mathbf{in}[s];\ s \textbf{ successor of } B_i)$

$$\mathrm{in}[B_i] = \mathrm{use}[B_i] \cup (\mathrm{out}[B_i] - \mathrm{def}[B_i])$$



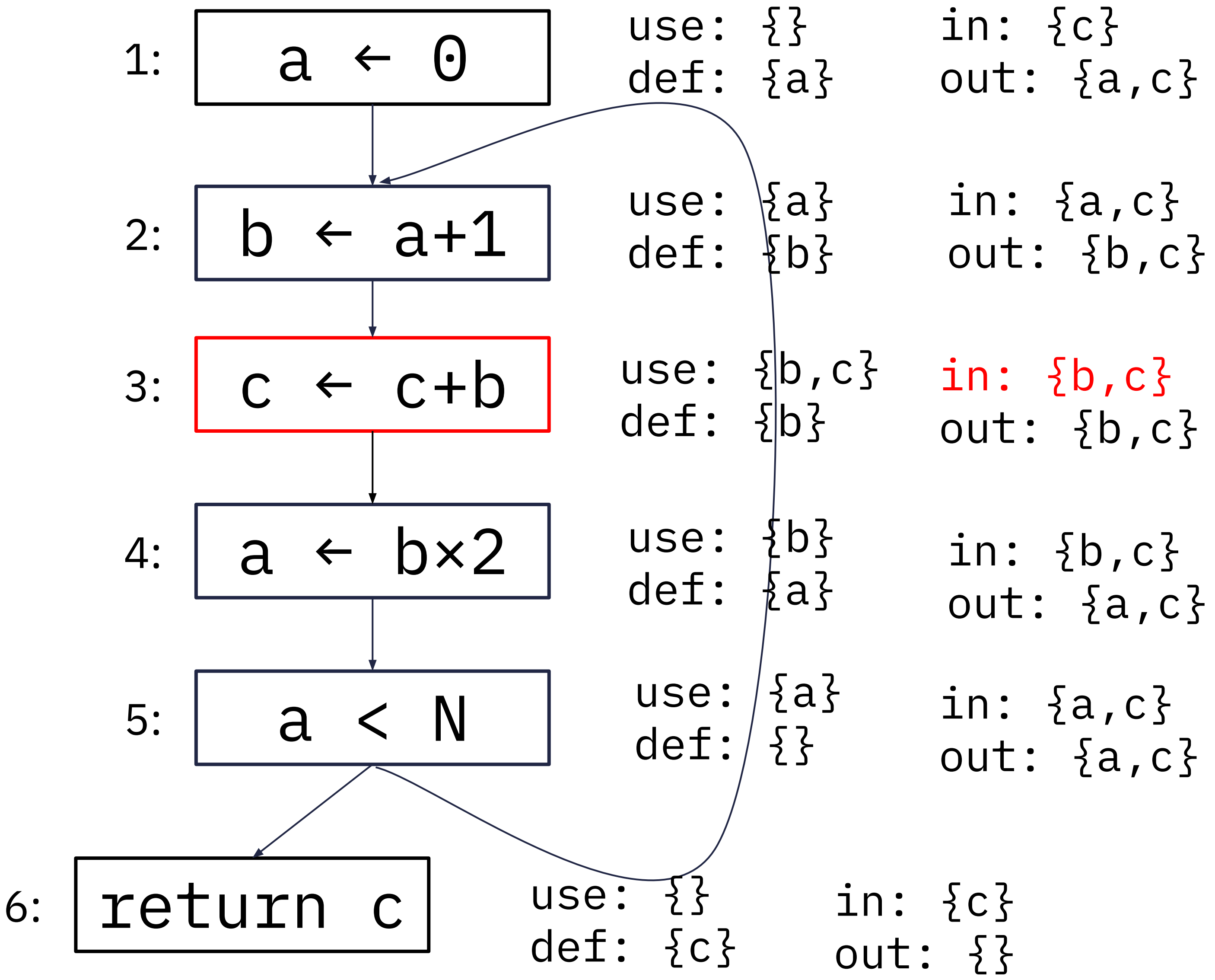
$$\mathrm{out}[B_i] = \bigcup(\mathrm{in}[s];\ s \text{ successor of } B_i)$$

$in[B_i] = use[B_i] \cup (out[B_i] - def[B_i])$



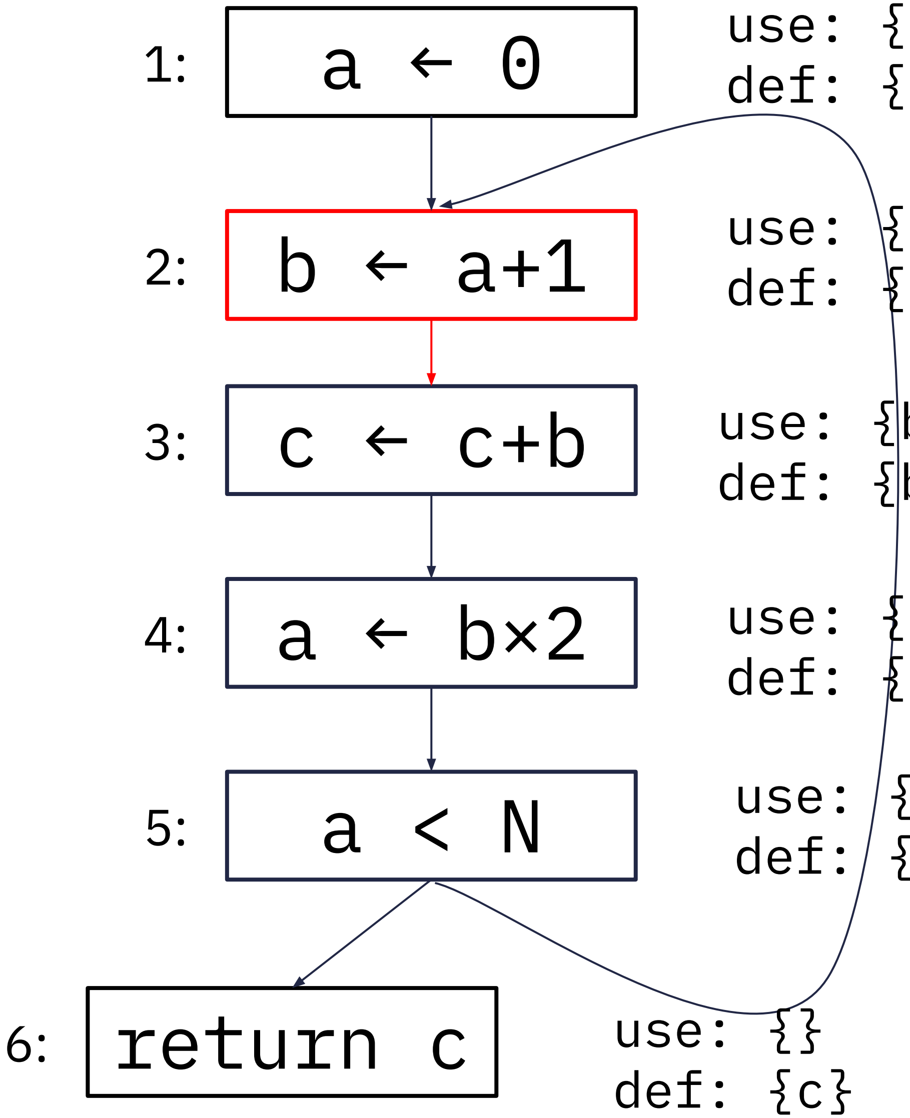
$\mathbf{out[B_i] = \cup(in[s];\ s\ successor\ of\ B_i)}$

$$in[B_i] = use[B_i] \cup (out[B_i] - def[B_i])$$



$$out[B_i] = \cup(in[s];\ s \text{ successor of } B_i)$$

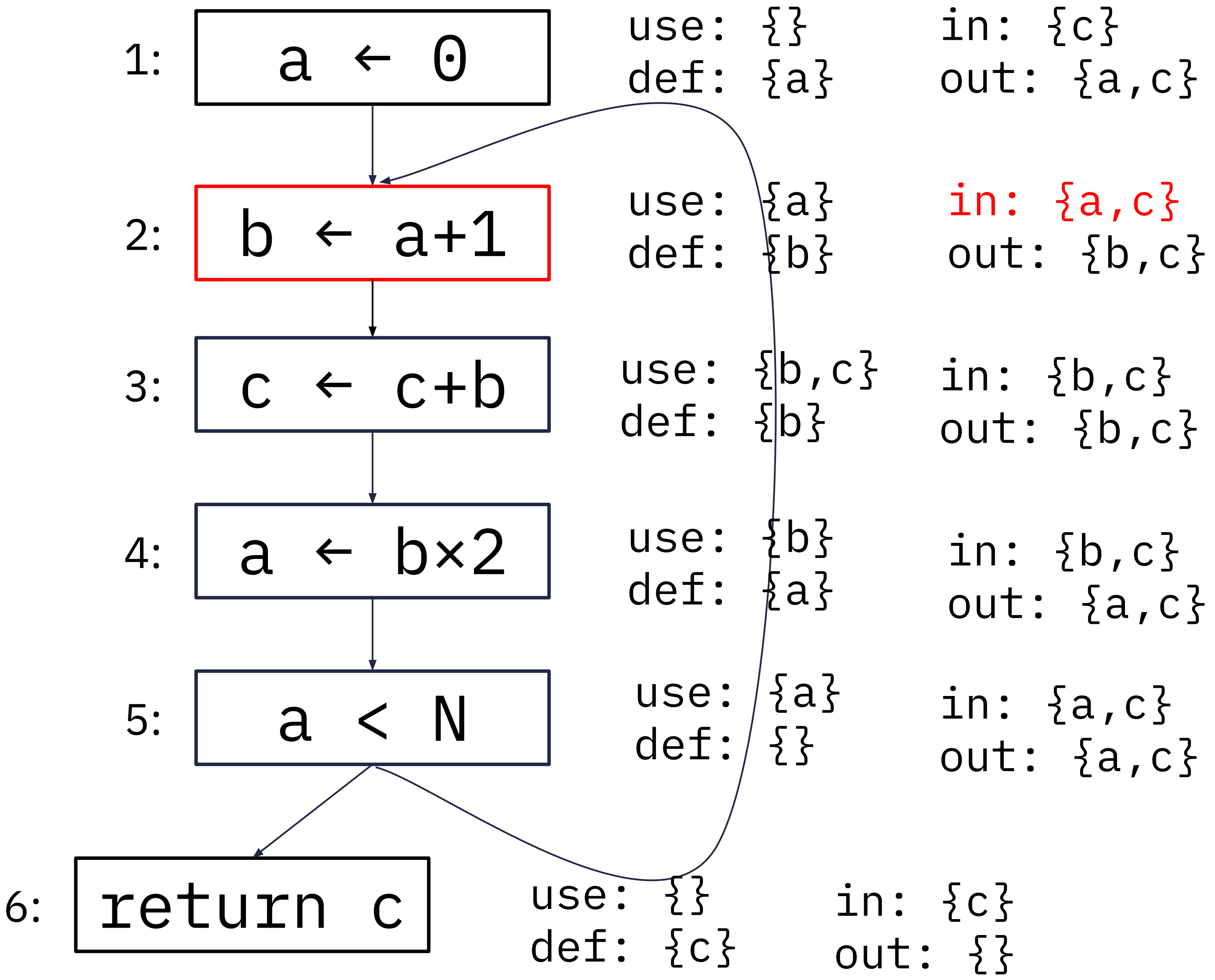
$in[B_i] = use[B_i] \cup (out[B_i] - def[B_i])$



Block	use	def	in	out
1	{}	{a}	{c}	{a,c}
2	{a}	{b}	{a,c}	{b,c}
3	{b,c}	{b}	{b,c}	{b,c}
4	{b}	{a}	{b,c}	{a,c}
5	{a}	{}	{a,c}	{a,c}
6	{}	{c}	{c}	{}

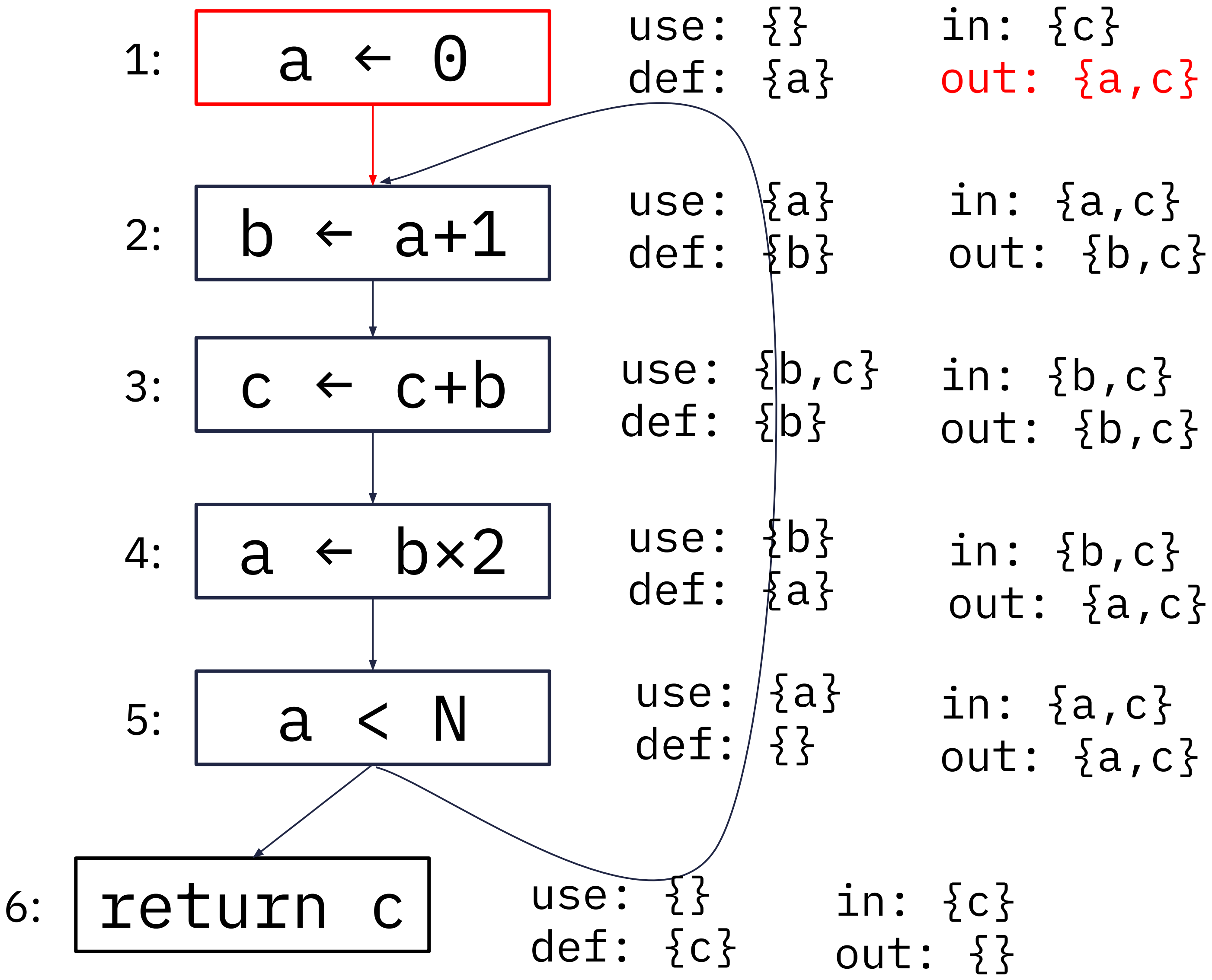
$out[B_i] = \bigcup(in[s];\ s \text{ successor of } B_i)$

$$in[B_i] = use[B_i] \cup (out[B_i] - def[B_i])$$



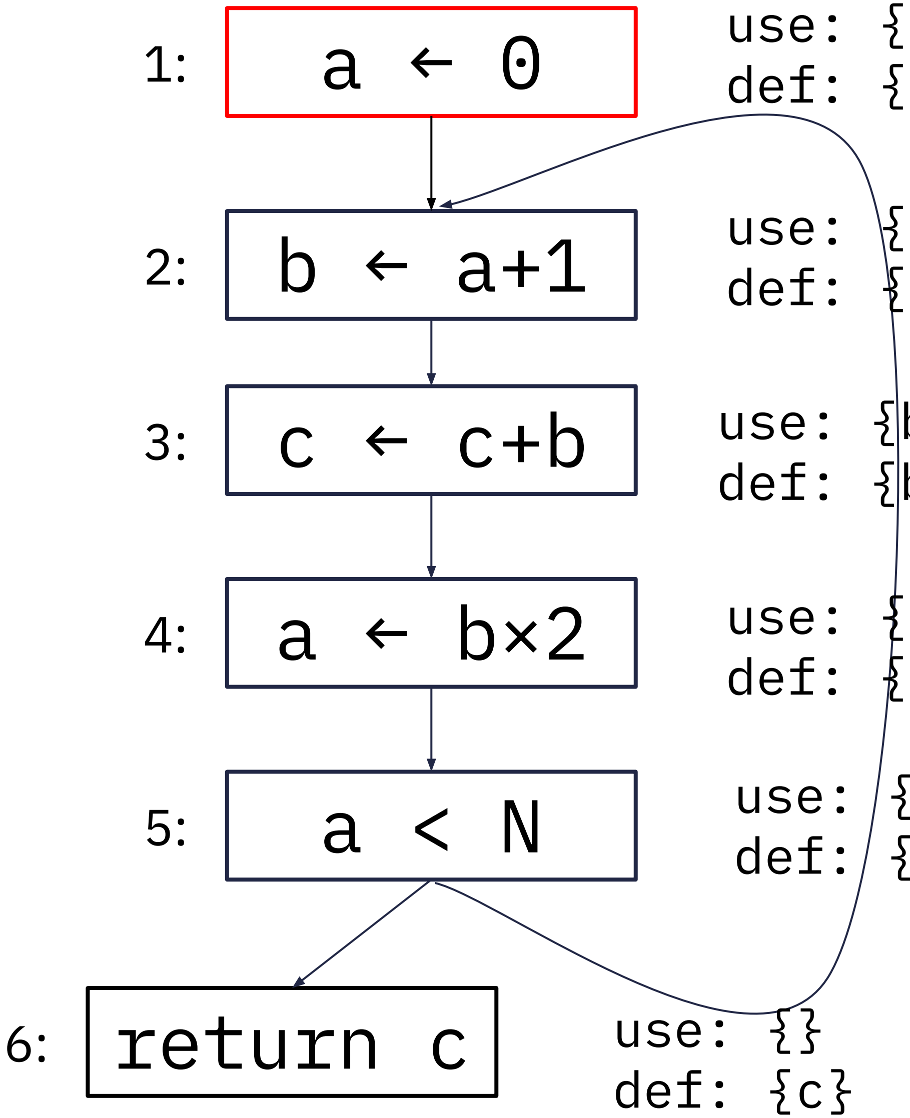
$$out[B_i] = \cup(in[s];\ s \text{ successor of } B_i)$$

$$in[B_i] = use[B_i] \cup (out[B_i] - def[B_i])$$



$$\mathbf{out[B_i] = \cup(in[s];\ s\ successor\ of\ B_i)}$$

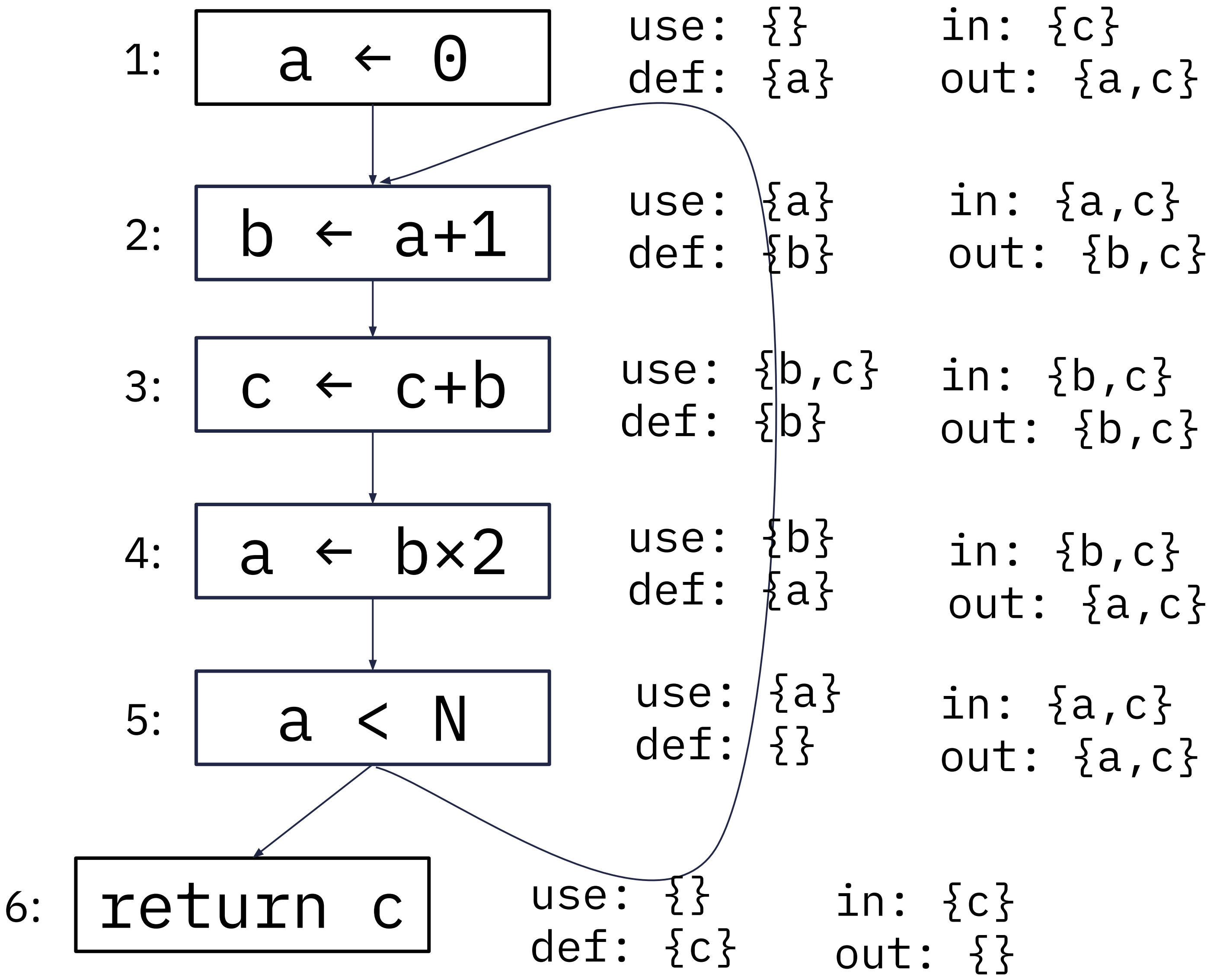
$$in[B_i] = use[B_i] \cup (out[B_i] - def[B_i])$$



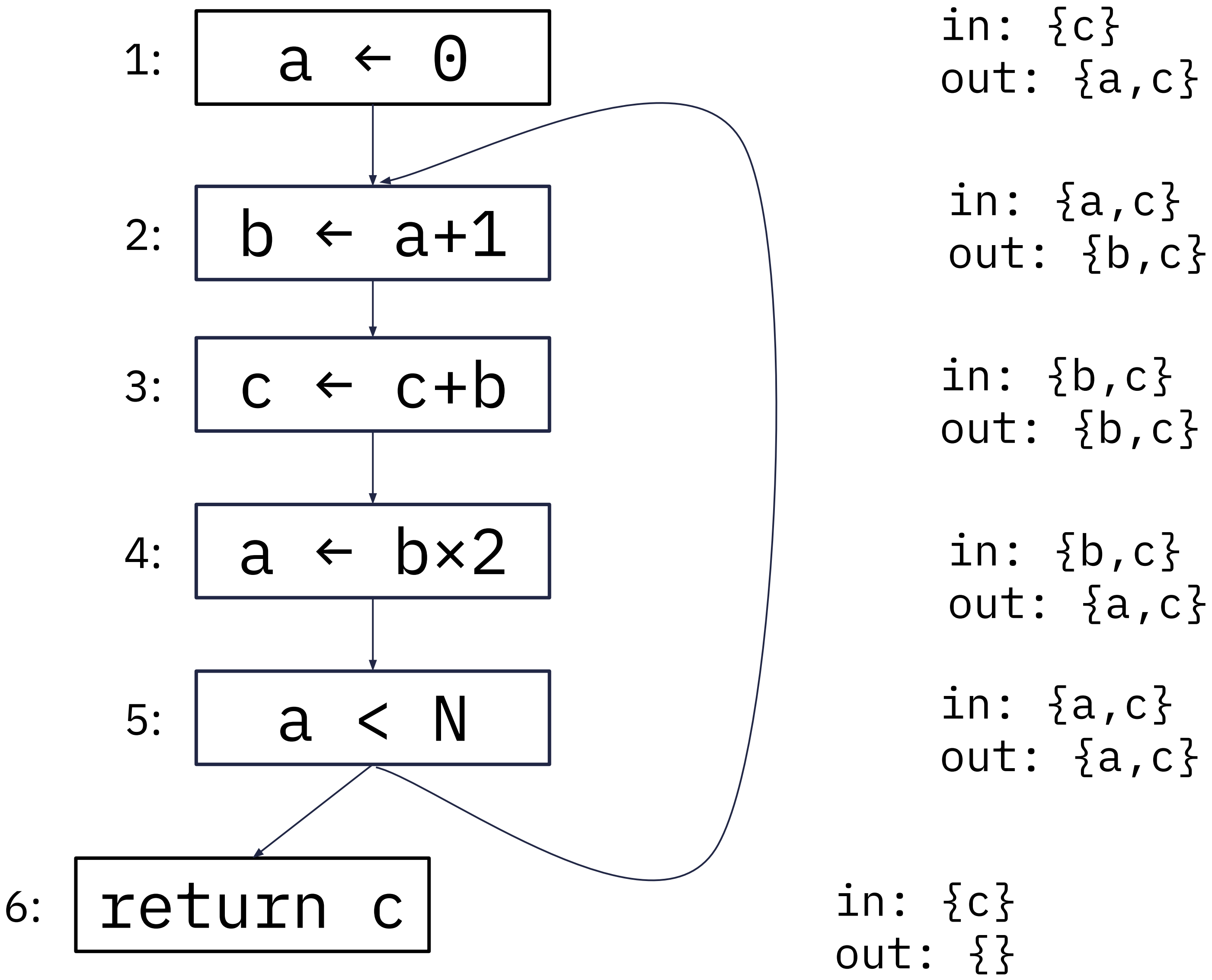
Block	use	def	in	out
1	{}	{a}	{c}	{a,c}
2	{a}	{b}	{a,c}	{b,c}
3	{b,c}	{b}	{b,c}	{b,c}
4	{b}	{a}	{b,c}	{a,c}
5	{a}	{}	{a,c}	{a,c}
6	{}	{c}	{c}	{}

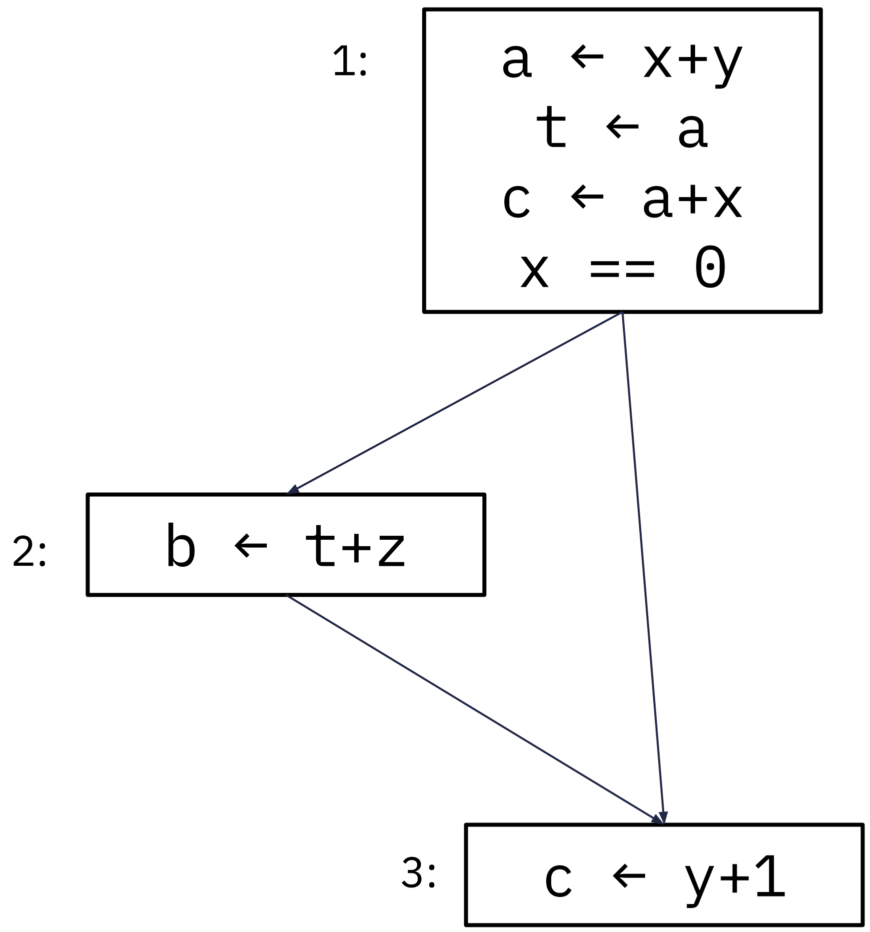
$$out[B_i] = \cup(in[s];\ s \text{ successor of } B_i)$$

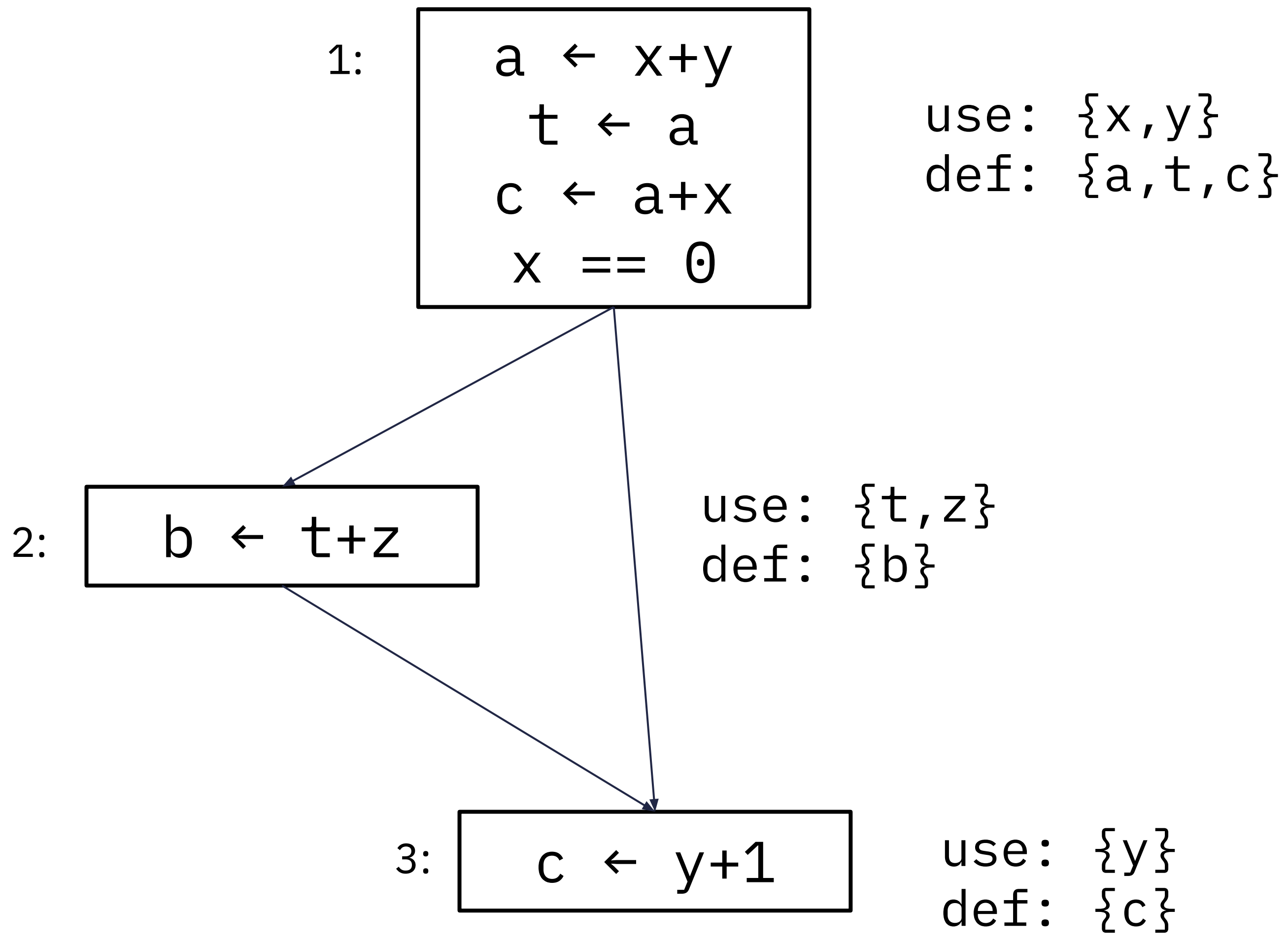
$in[B_i] = use[B_i] \cup (out[B_i] - def[B_i])$

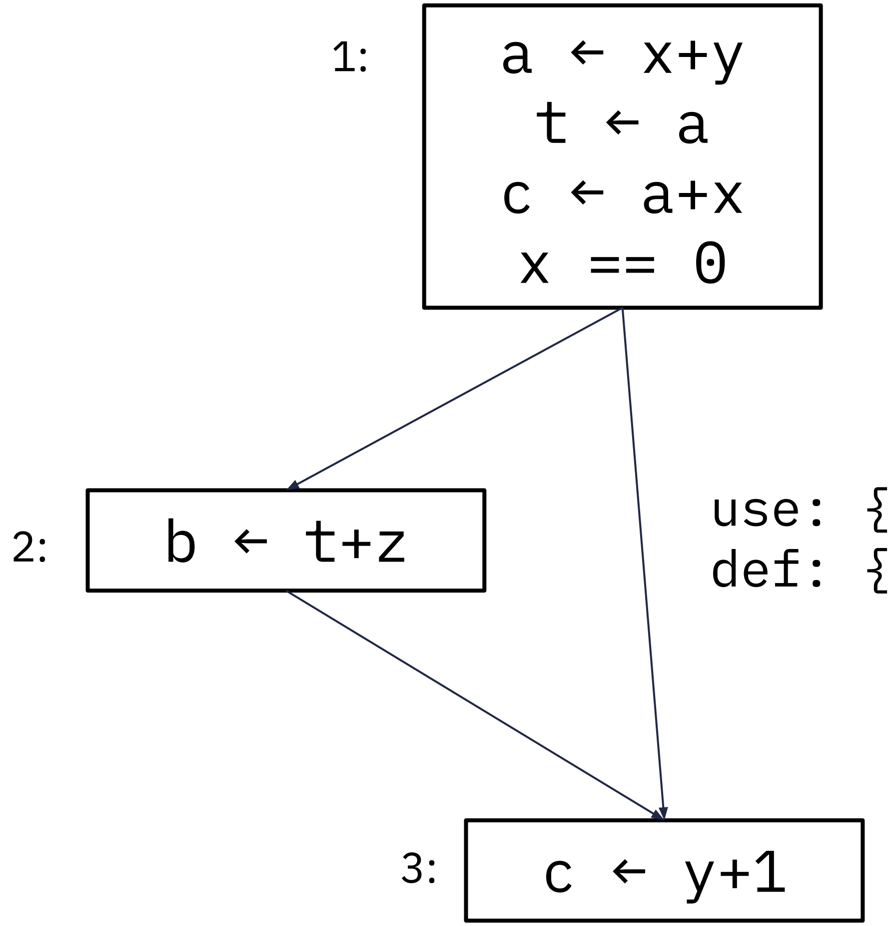


$out[B_i] = \cup(in[s];\ s \text{ successor of } B_i)$


1:
a ← 0
2:
b ← a+1
3:
c ← c+b
4:
a ← b×2
5:
a < N
6:
return c
in: {c}
out: {a,c}
in: {a,c}
out: {b,c}
in: {b,c}
out: {b,c}
in: {b,c}
out: {a,c}
in: {a,c}
out: {a,c}
in: {c}
out: {}


1:
a ← x+y
t ← a
c ← a+x
x == 0
2:
b ← t+z
3:
c ← y+1

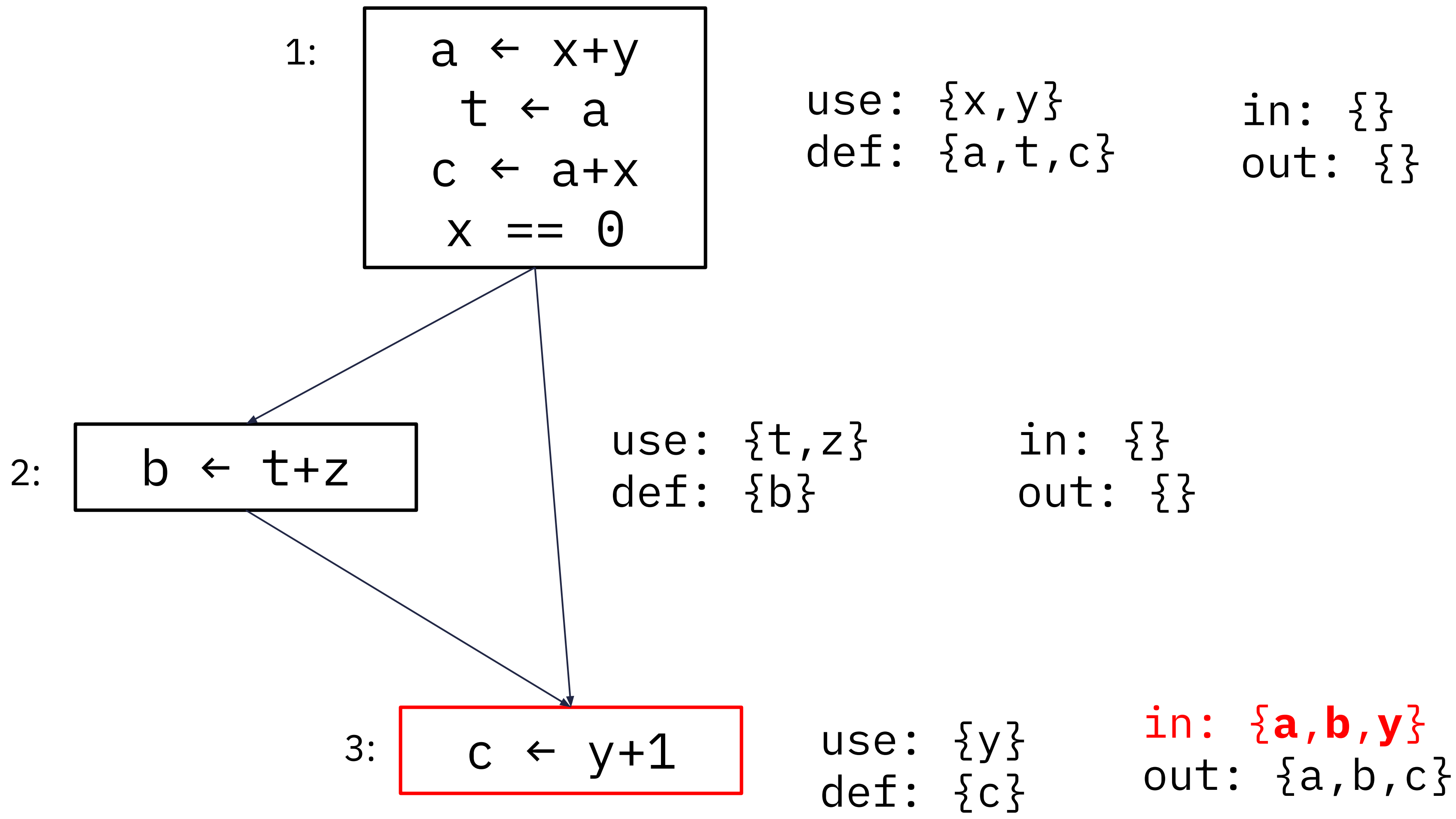

1:
a ← x+y
t ← a
c ← a+x
x == 0
use: {x,y}
def: {a,t,c}
2:
b ← t+z
use: {t,z}
def: {b}
3:
c ← y+1
use: {y}
def: {c}



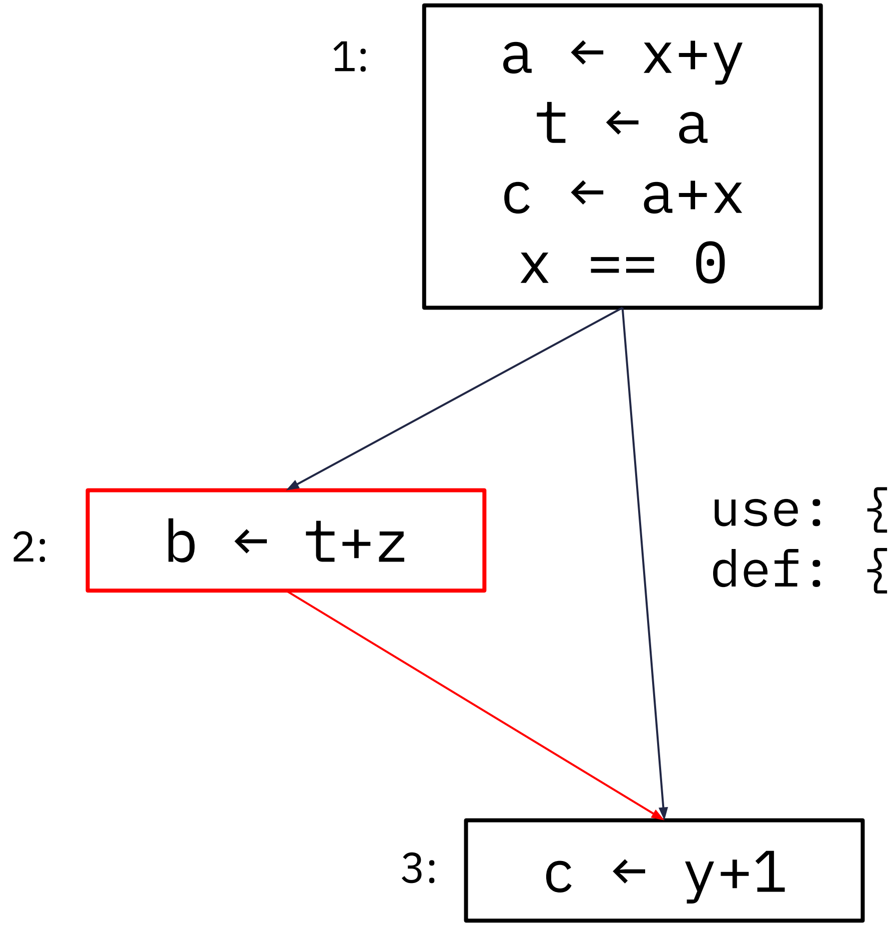
Block 1:
use: {x,y}
def: {a,t,c}
in: {}
out: {}

Block 2:
use: {t,z}
def: {b}
in: {}
out: {}

Block 3:
use: {y}
def: {c}
in: {}
out: {**a**,**b**,**c**}



1:
a ← x+y
t ← a
c ← a+x
x == 0
use: {x,y}
def: {a,t,c}
in: {}
out: {}
2:
b ← t+z
use: {t,z}
def: {b}
in: {}
out: {}
3:
c ← y+1
use: {y}
def: {c}
in: {**a**,**b**,**y**}
out: {a,b,c}



Block 1:

use: {x,y}
def: {a,t,c}

in: {}
out: {}

Block 2:

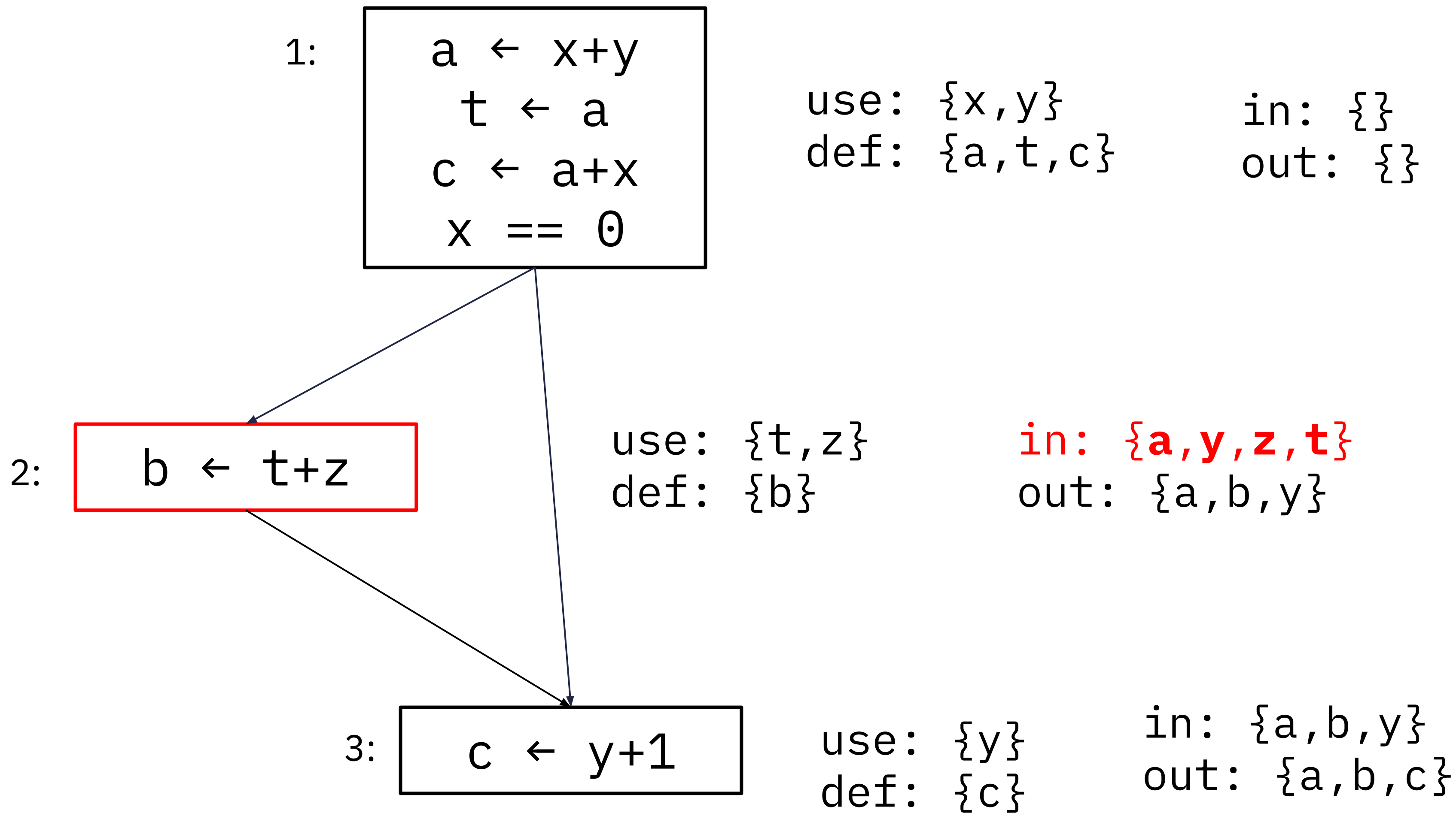
use: {t,z}
def: {b}

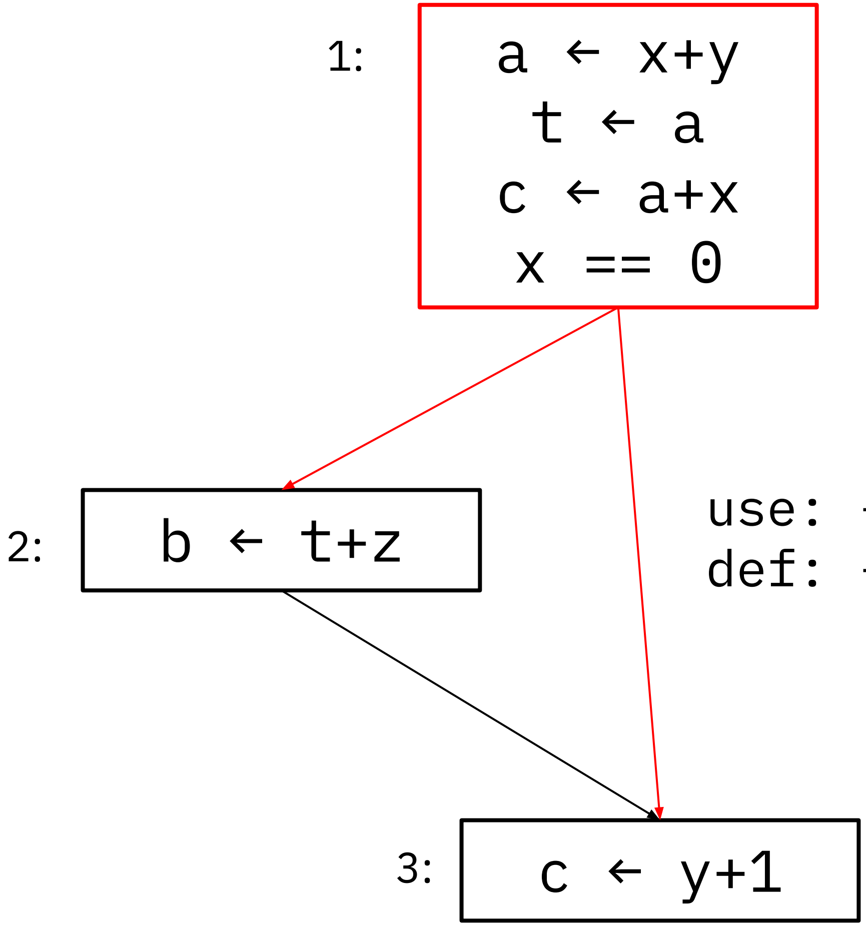
in: {}
out: {a,b,y}

Block 3:

use: {y}
def: {c}

in: {a,b,y}
out: {a,b,c}


1:
a ← x+y
t ← a
c ← a+x
x == 0
use: {x,y}
def: {a,t,c}
in: {}
out: {}
2:
b ← t+z
use: {t,z}
def: {b}
in: {a,y,z,t}
out: {a,b,y}
3:
c ← y+1
use: {y}
def: {c}
in: {a,b,y}
out: {a,b,c}



use: {x,y}
def: {a,t,c}

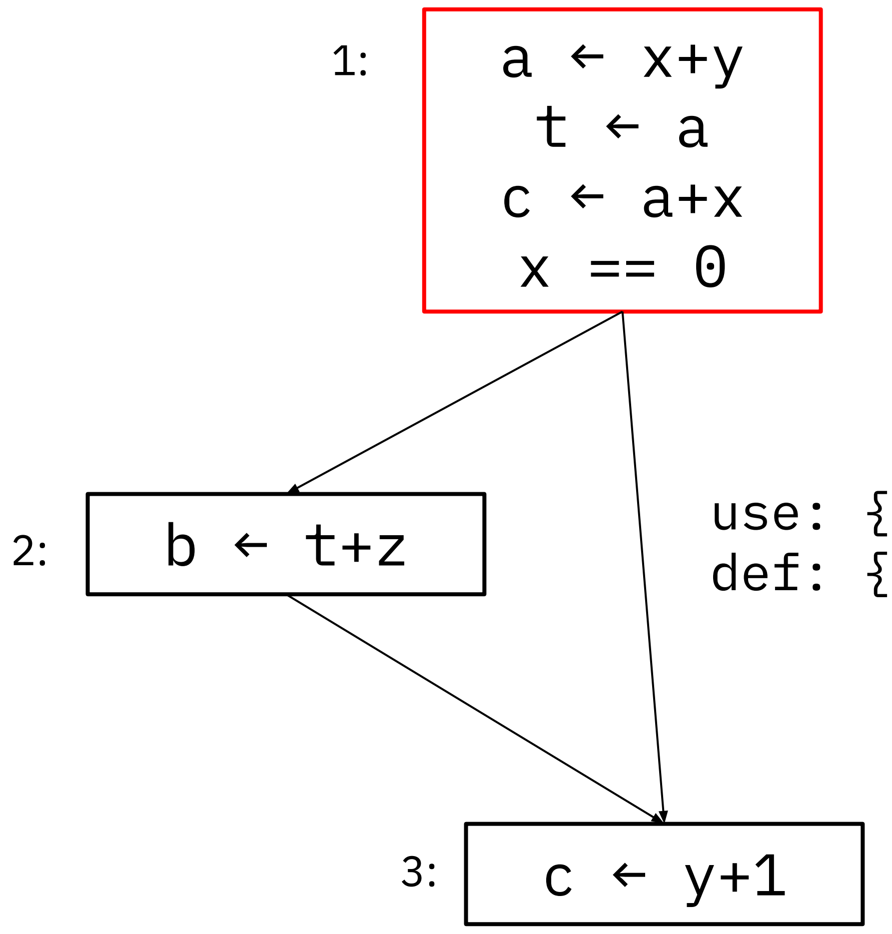
in: {}
out: **{a,b,y,z,t}**

use: {t,z}
def: {b}

in: {a,y,z,t}
out: {a,b,y}

use: {y}
def: {c}

in: {a,b,y}
out: {a,b,c}



use: {x,y}
def: {a,t,c}

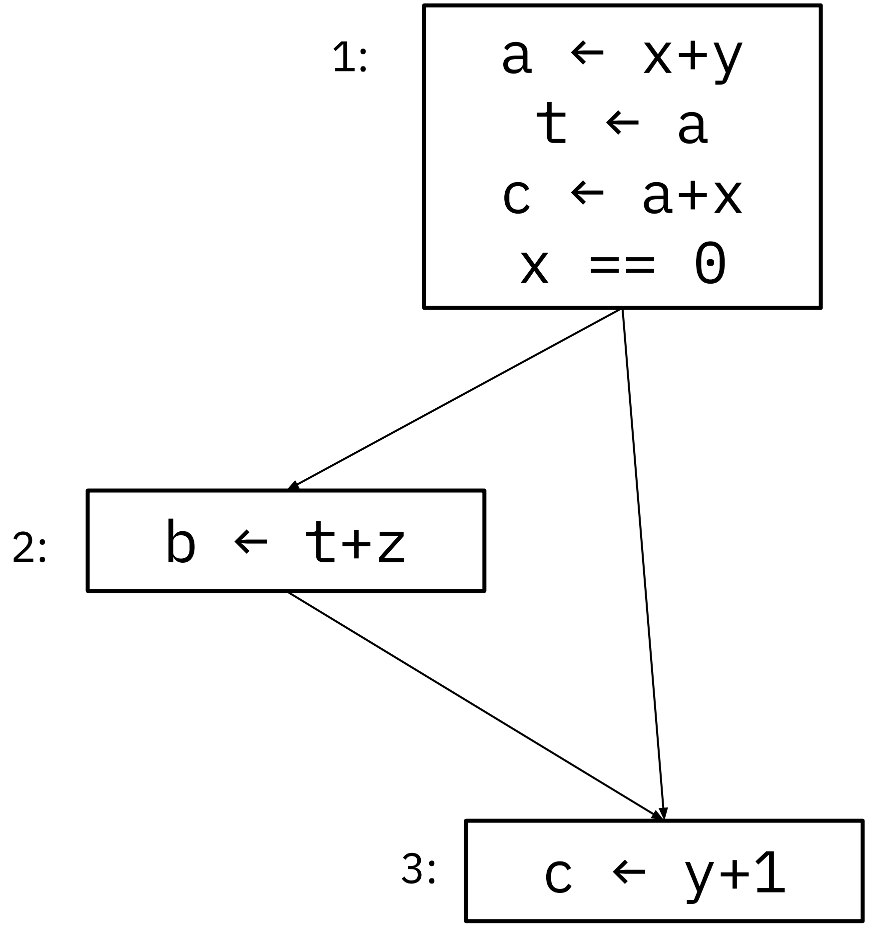
in: {**b**,**x**,**y**,**z**}
out: {a,b,y,z,t}

use: {t,z}
def: {b}

in: {a,y,z,t}
out: {a,b,y}

use: {y}
def: {c}

in: {a,b,y}
out: {a,b,c}



in: {b,x,y,z}
out: {a,b,y,z,t}

in: {a,y,z,t}
out: {a,b,y}

in: {a,b,y}
out: {a,b,c}

Liveness

- Dataflow equations:
  - $\texttt{in}[B_i] = \texttt{use}[B_i] \cup (\texttt{out}[B_i] - \texttt{def}[B_i])$
  - $\texttt{out}[B_i] = \texttt{in}[B_{s1}] \cup \texttt{in}[B_{s2}] \cup \ldots \cup \texttt{in}[B_{sk}]$
  - $\texttt{out}[B_{exit}]$ = set of globals
- Satisfies Gen/Kill pattern:
  - `gen = use`, `kill = def`

Dead code elimination

- If y is dead after the line y←…, delete this line.
- But consider this example (assume x is local):

```
                        // live-in:
x₀ ← a₀                 // {x₀,x₁,x₂}
x₁ ← x₀ + a₁            // {x₁,x₂}
x₂ ← x₁ + a₂            // {x₂}
x₃ ← x₂ + a₃            // {}
```

Needs three iterations of liveness+DCE to optimize!

Dead code elimination

- $\texttt{in}[\texttt{B}_\texttt{i}] = \texttt{use}[\texttt{B}_\texttt{i}] \cup (\texttt{out}[\texttt{B}_\texttt{i}] - \texttt{def}[\texttt{B}_\texttt{i}])$
  still leaves out some information.

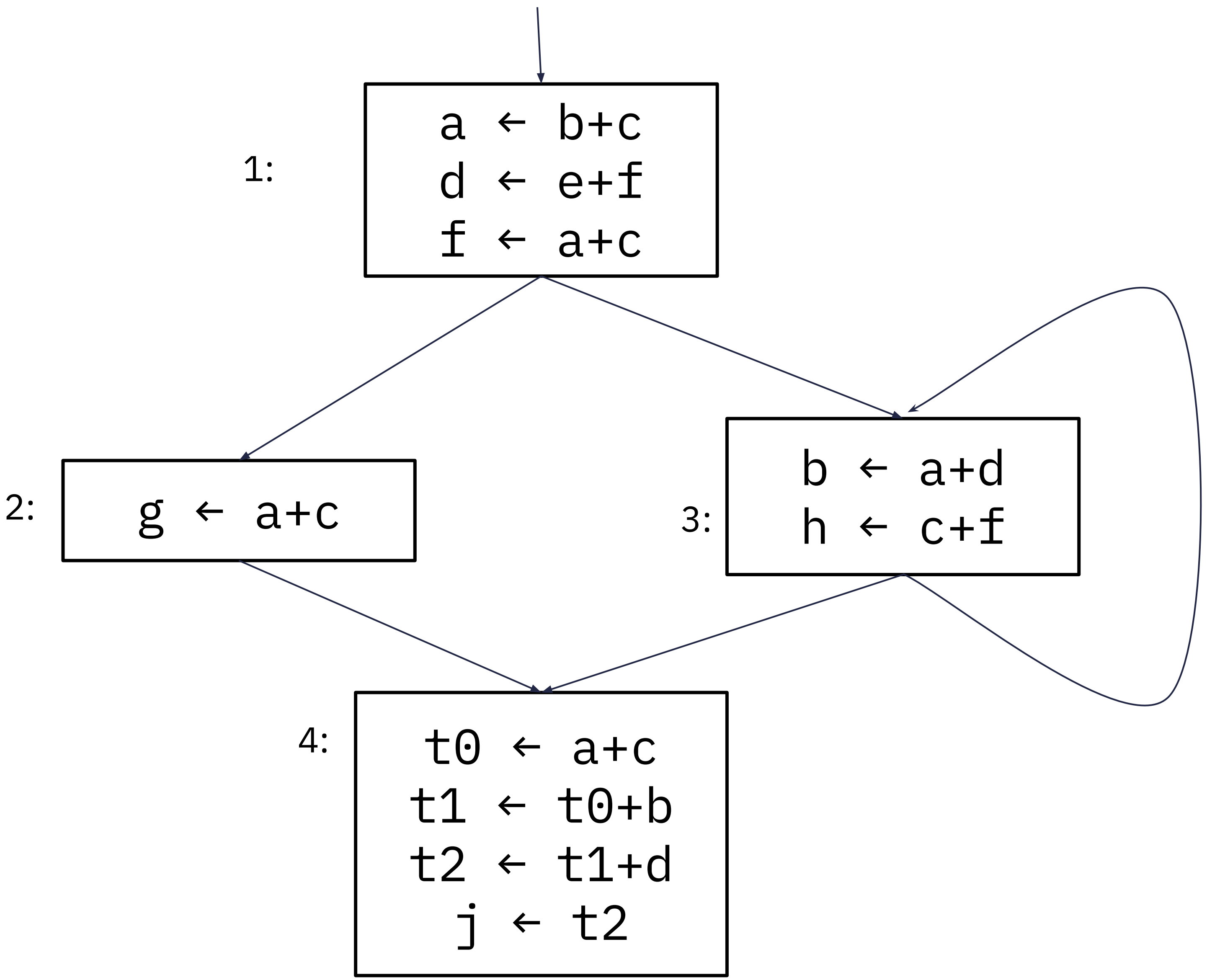
- A better, cascading liveness analysis
  - If y is not live after `y ← ...`, then we can remove the line *and* consider the variables in the RHS as not used.
  - $\texttt{in}[\texttt{B}_\texttt{i}] = \texttt{f}(\texttt{out}[\texttt{B}_\texttt{i}], \texttt{B}_\texttt{i})$
  - Doesn't fit nicely into the gen/kill pattern.
    - Alternatively, think of `use` as a function of $\texttt{B}_\texttt{i}$ *and* $\texttt{out}[\texttt{B}_\texttt{i}]$

Available expressions

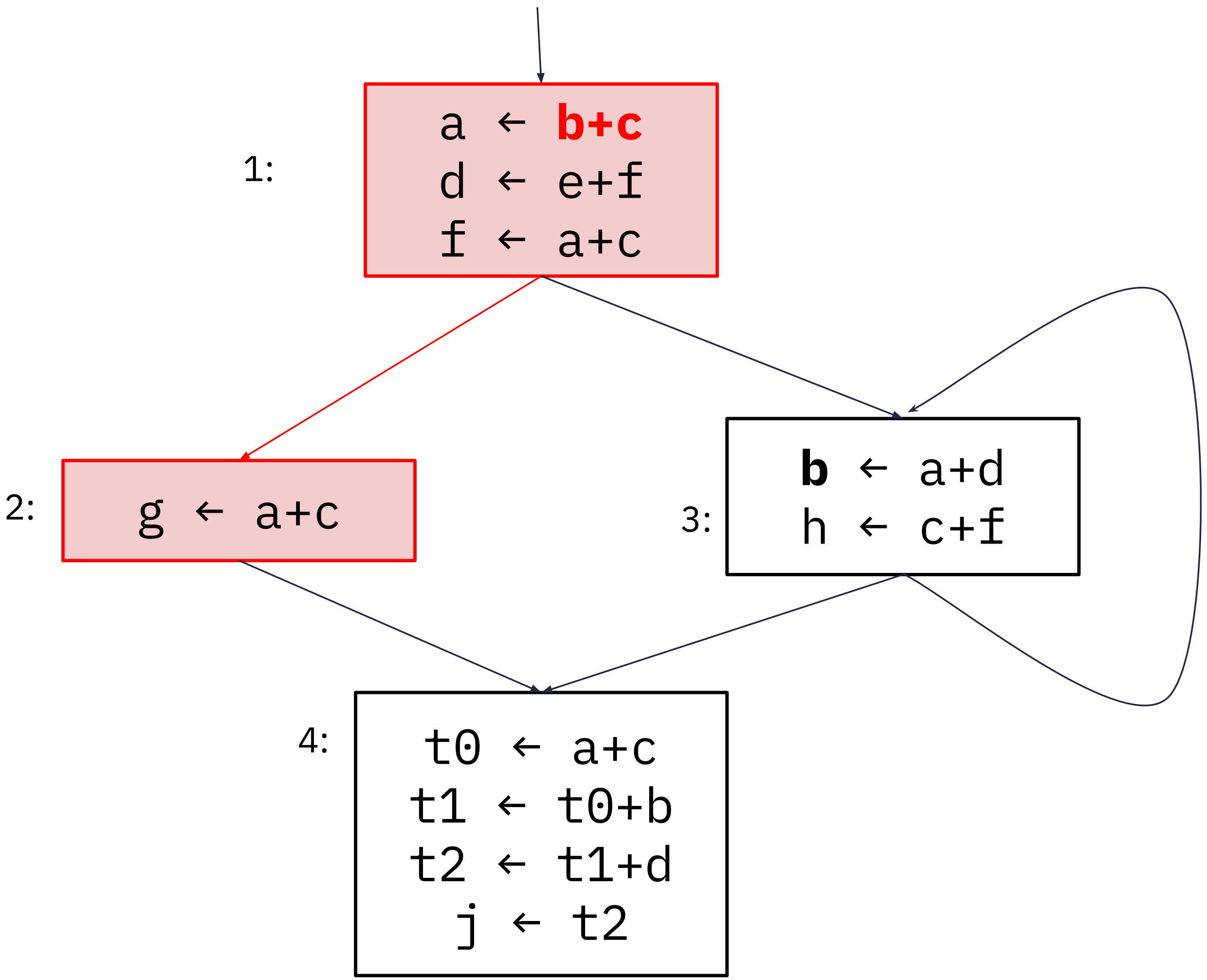
- An expression is **available** at point p if
  - **all** paths from initial node to p evaluate this expression
  - the evaluation doesn't become stale before reaching p (i.e. no operands are re-defined on the path)

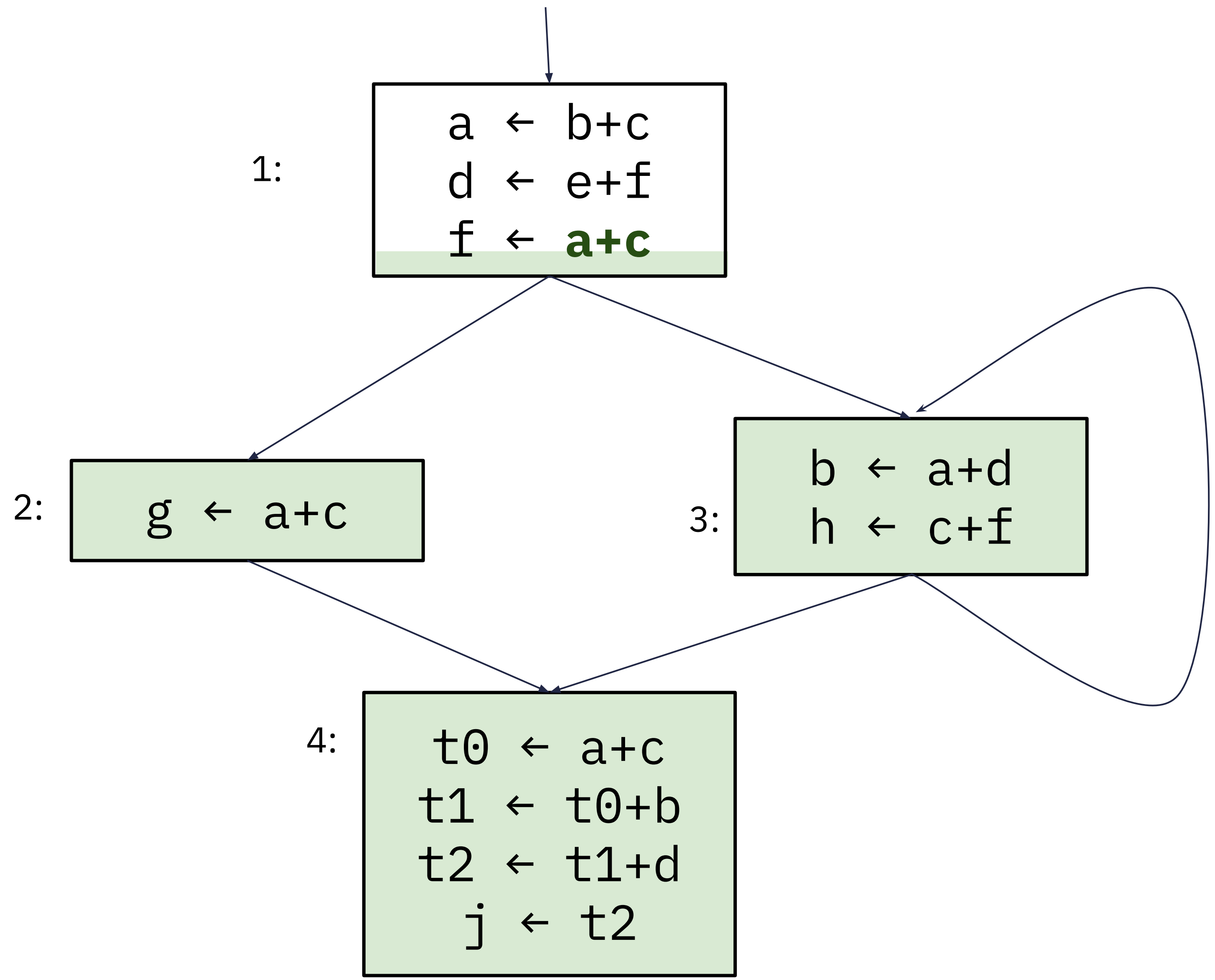
- One way to identify an expression is by hashing it

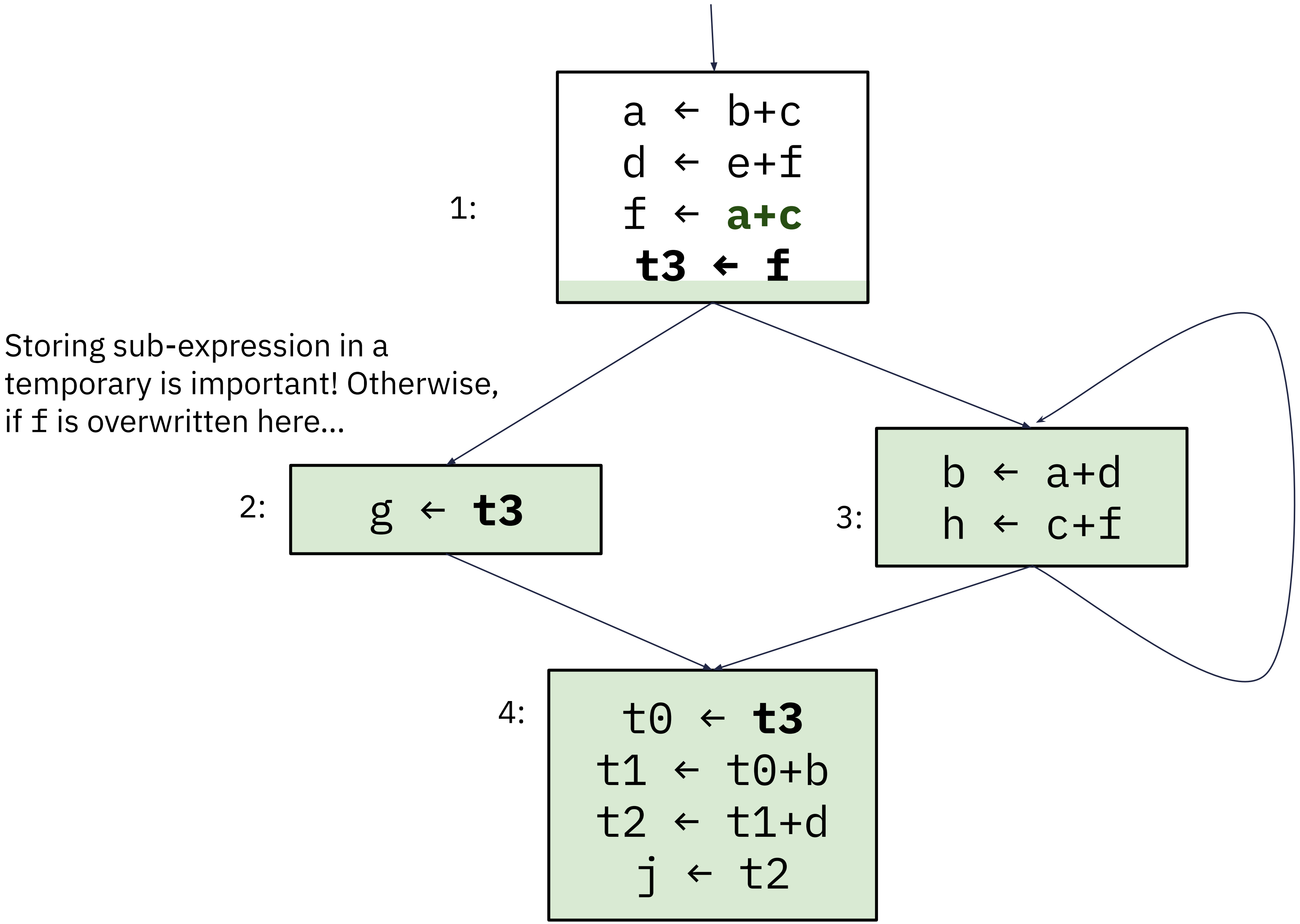
- Used to implement global CSE transform



1:
a ← b+c
d ← e+f
f ← a+c
2:
g ← a+c
3:
b ← a+d
h ← c+f
4:
t0 ← a+c
t1 ← t0+b
t2 ← t1+d
j ← t2


1:
a ← b+c
d ← e+f
f ← a+c
2:
g ← a+c
3:
b ← a+d
h ← c+f
4:
t0 ← a+c
t1 ← t0+b
t2 ← t1+d
j ← t2


1:
a ← b+c
d ← e+f
f ← a+c
2:
g ← a+c
3:
b ← a+d
h ← c+f
4:
t0 ← a+c
t1 ← t0+b
t2 ← t1+d
j ← t2



1:
a ← b+c
d ← e+f
f ← a+c
t3 ← f
Storing sub-expression in a temporary is important! Otherwise, if f is overwritten here...
2:
g ← t3
3:
b ← a+d
h ← c+f
4:
t0 ← t3
t1 ← t0+b
t2 ← t1+d
j ← t2

Available expressions

- universe = set of ≤2-operand expressions
- `in`[B_i] = expressions available at the start of B_i
- `out`[B_i] = expressions available at the end of B_i
- `gen`[B_i] = expressions computed in B_i
- `kill`[B_i] = expressions made stale by B_i

Available expressions

- $in[B_i] = out[B_{p1}] \cap out[B_{p2}] \cap \ldots\ out[B_{pk}]$
- $out[B_i] = (in[B_i] - kill[B_i]) \cup gen[B_i]$
- $in[B_{entry}] = 0$
- Initially assume $out[B_i]$ = universe ??

Least fixed point

- Iterative dataflow analysis works because of lattice theory. There is always a unique least fixed point (in these cases).

- Intuitively, our modifications are **monotone**.
  - Start with empty set then keep union-ing.
  - Start with full set then keep intersect-ing.
  - Must eventually reach universe or empty set in the worst case (no useful result for optimization).

Optimism/Pessimism

- Depends on what the analysis is used for

- Analysis is "pessimistic" or "conservative" if stopping the analysis early means we simply get worse result (but is still safe)

- Analysis is "optimistic" or "aggressive" if stopping early means the result could lead to wrong optimization

Reaching Definitions

- $\texttt{in}[B_i] = \texttt{out}[B_{p1}] \cup \texttt{out}[B_{p2}] \cup \ldots \texttt{out}[B_{pk}]$
- $\texttt{out}[B_i] = (\texttt{in}[B_i] - \texttt{kill}[B_i]) \cup \texttt{gen}[B_i]$
- $\texttt{in}[B_{entry}] = 0$
- Initially assume $\texttt{out}[B_i] = 0$

Reaching Definitions

- Uniquely identify each assignment statement ("def")
- A definition reaches a use if there **exists** a path from the definition to the use where the definition isn't killed on the path
- Used for constant propagation.
  - If a variable has exactly one reaching definition and the definition is a constant value.
- Can this be used for copy propagation?

Other dataflow analyses

- Dominance (read Cooper et al.)
- Anticipable expressions
- Upward-exposed uses

- Or combine analysis results:
  - Use-def/Def-use chains

Other global optimizations

- Inlining
  - How to decide whether to inline or not inline?
- Global code placement
  - Place procedures that are used together closer
- Global register allocation
  - We will study this!

Global Optimization

- For the exam, you should be able to:
  - Perform reaching definitions, available expressions, and liveness analysis given a CFG (statements or basic blocks)
  - Write dataflow equations for these optimizations
  - Perform global optimizations.
  - Explain advantages and limitations of each optimization

- You will learn more about theory of dataflow analysis in later lectures
  - Lattice theory
  - How to design arbitrary analyses

Terms and glossary
Local optimization
Global optimization
Implementation notes ←

Implementation notes

- Use array of nodes, not pointer-and-objects
  - Key: Need to be able to remove/add statements
  - Especially relevant if you don't use basic blocks
  - You will need adjacency list and reverse adj. list

- 1 node = 1 statement is *somewhat* easier
  - More time/memory-consuming but who cares
  - No need to propagate information inside a basic block
  - One tricky thing: Need to be able to add/remove nodes/merge points/join points.

Implementation notes

- Make a parameterized dataflow framework
  - Parameterized on direction, meet operator (union or intersection), initial values, transfer function

	Reaching Definitions	Live Variables	Available Expressions
Domain	Sets of definitions	Sets of variables	Sets of expressions
Direction	Forwards	Backwards	Forwards
Transfer function	$gen_B \cup (x - kill_B)$	$use_B \cup (x - def_B)$	$e\_gen_B \cup (x - e\_kill_B)$
Boundary	$\text{OUT}[\text{ENTRY}] = \emptyset$	$\text{IN}[\text{EXIT}] = \emptyset$	$\text{OUT}[\text{ENTRY}] = \emptyset$
Meet ($\wedge$)	$\cup$	$\cup$	$\cap$
Equations	$\text{OUT}[B] = f_B(\text{IN}[B])$ $\text{IN}[B] = \bigwedge_{P,pred(B)} \text{OUT}[P]$	$\text{IN}[B] = f_B(\text{OUT}[B])$ $\text{OUT}[B] = \bigwedge_{S,succ(B)} \text{IN}[S]$	$\text{OUT}[B] = f_B(\text{IN}[B])$ $\text{IN}[B] = \bigwedge_{P,pred(B)} \text{OUT}[P]$
Initialize	$\text{OUT}[B] = \emptyset$	$\text{IN}[B] = \emptyset$	$\text{OUT}[B] = U$

Figure 9.21: Summary of three data-flow problems

Implementation notes

- Represent these things differently
  - Global variables
  - Array variables
  - Local variables
    - Distinguish vars vs. temps? Debatable.

Implementation notes

- Do not underestimate phase 4
    - You will need some analysis results for register allocation
    - Be ready to refactor as needed

- Bonus: Learn SSA
    - It is *the* hot new thing in compiler backend development