CS614: Advanced Compilers

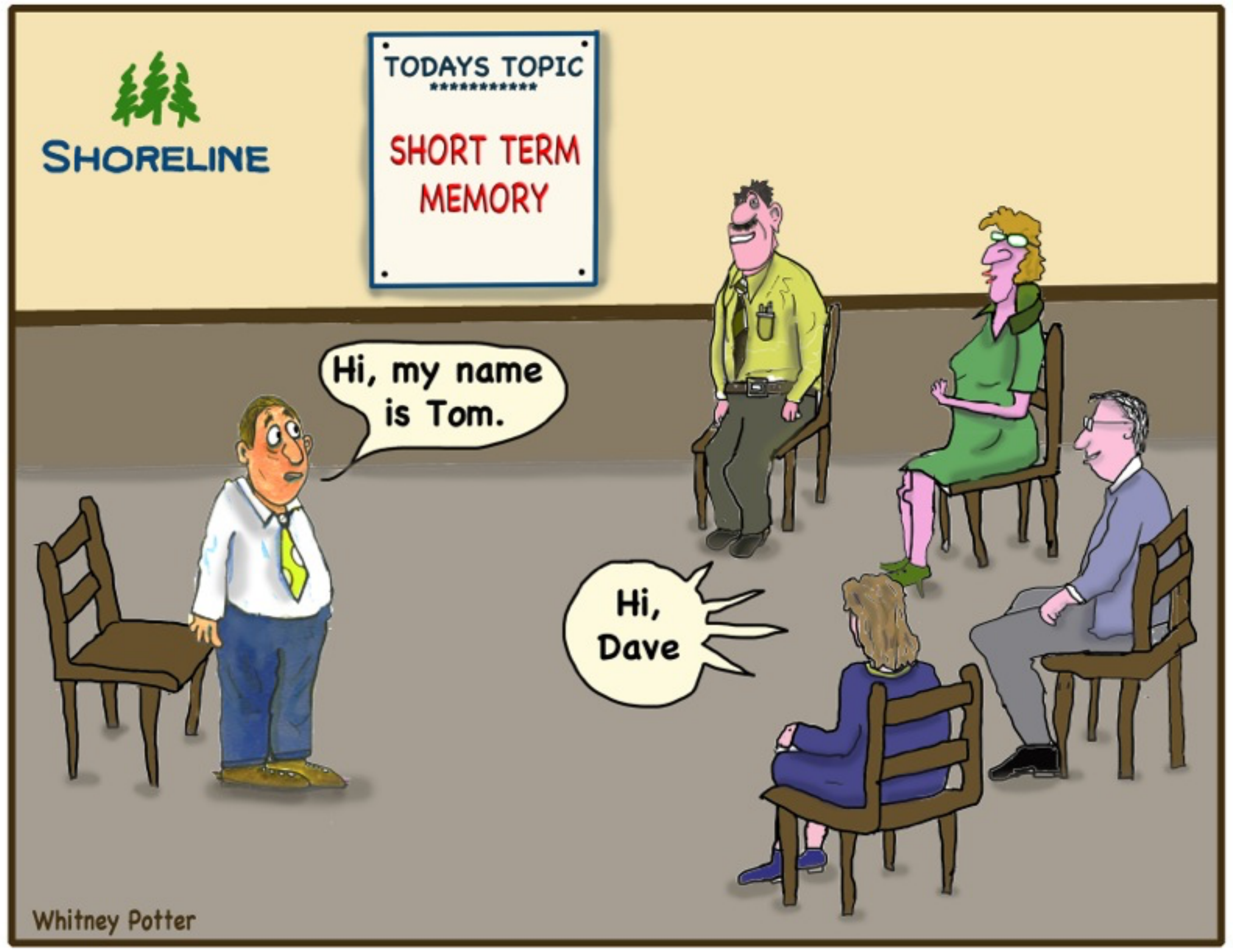
Control-Flow Graphs and IDFAs

Manas Thakur
CSE, IIT Bombay

Spring 2025

Things we learnt in the last class

- Examples of program optimizations
  - Machine independent
  - Machine dependent



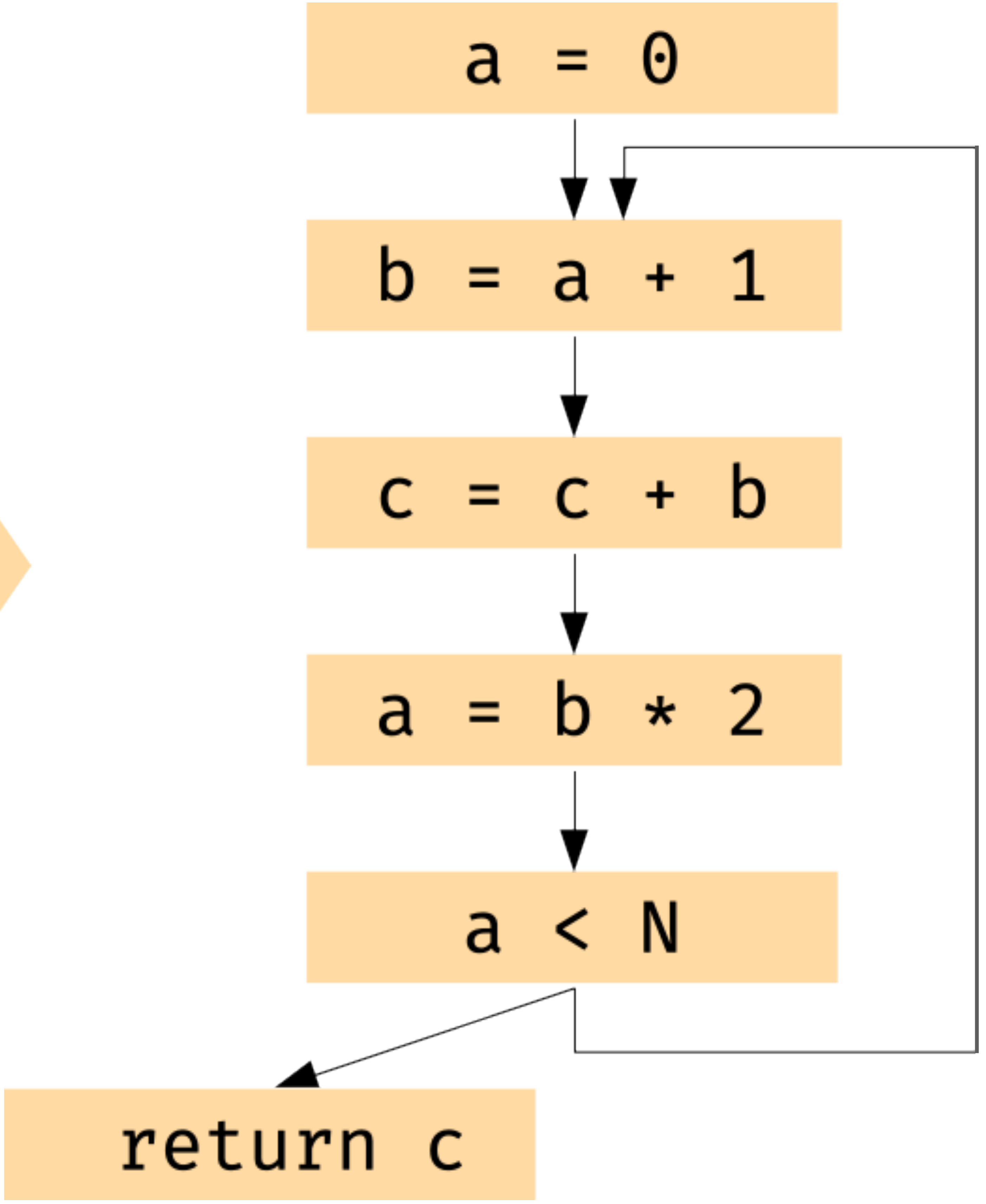
- Metrics to measure optimization
  - Time, memory, code size, energy

- Optimizations may interact with each other as well as themselves
- Optimizations are usually enabled by program analysis

Control-Flow Graphs

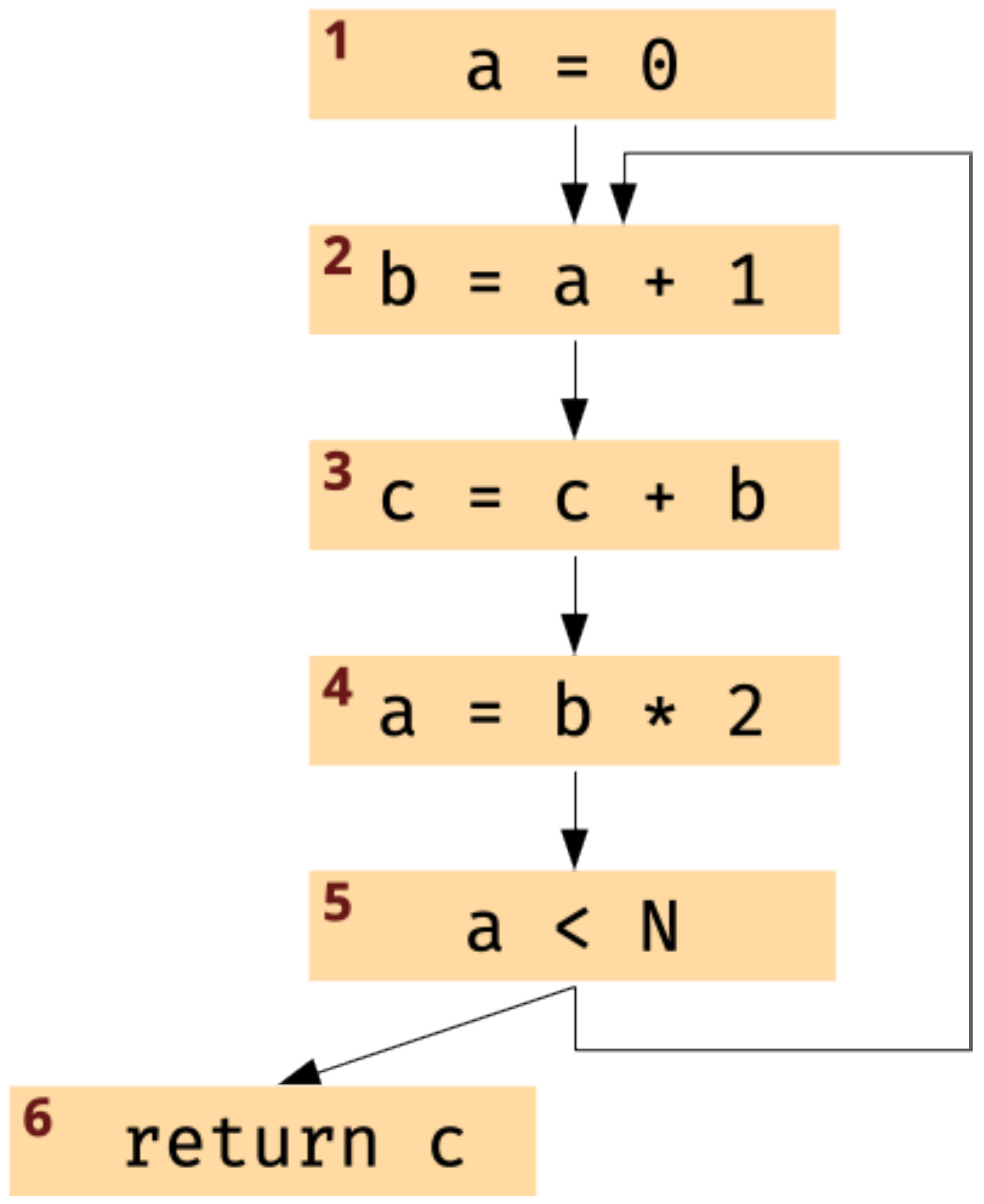
➤ Nodes represent instructions; edges represent flow of control

```
    a = 0
L1: b = a + 1
    c = c + b
    a = b * 2
    if a < N goto L1
    return c
```



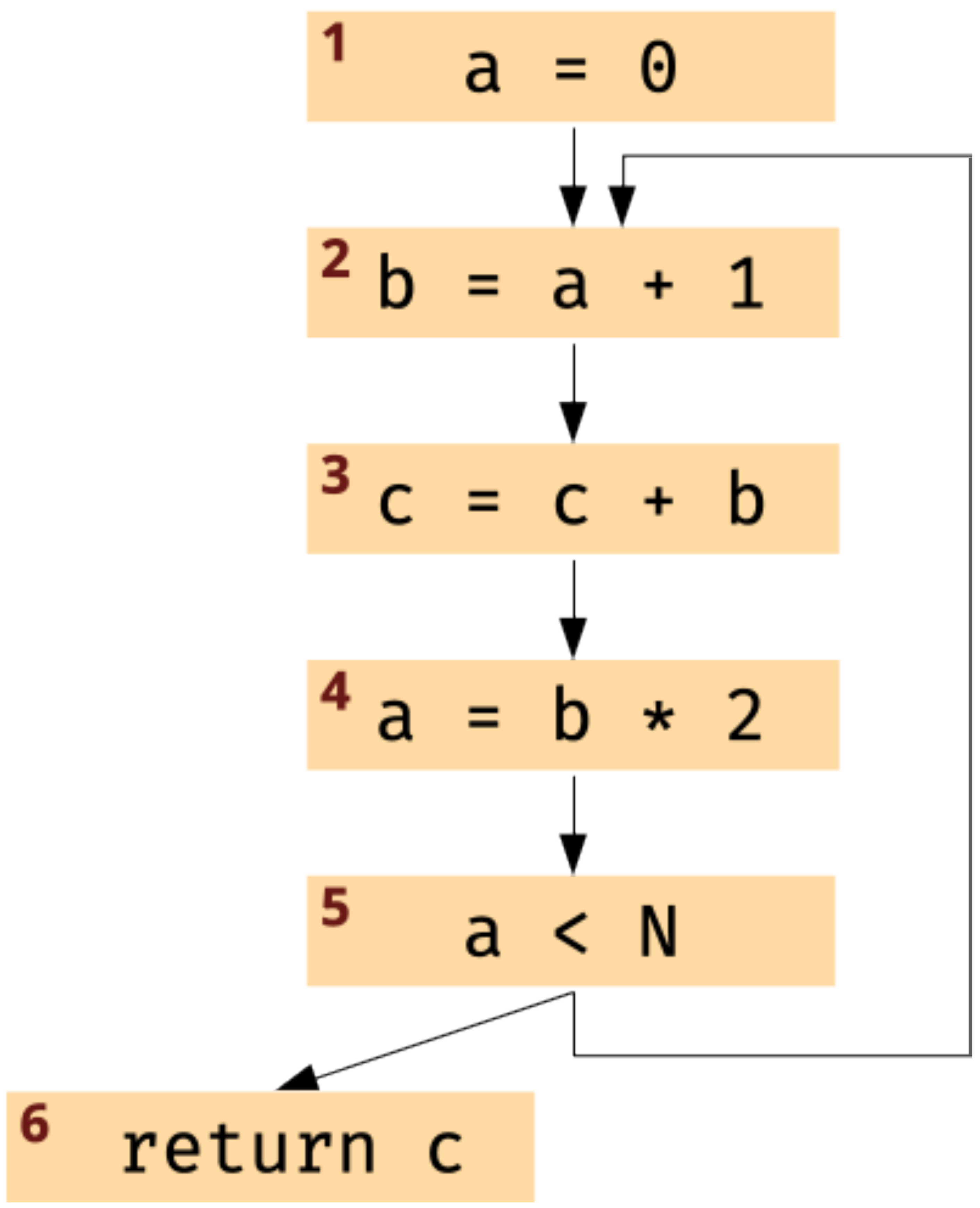
Some CFG Terminology

- `pred[n]` gives predecessors of `n`
  - `pred[1]? pred[4]? pred[2]?`
- `succ[n]` gives successors of `n`
  - `succ[2]? succ[5]?`
- `def(n)` gives variables defined by `n`
  - `def(3) = {c}`
- `use(n)` gives variables used by `n`
  - `use(3) = {b, c}`



Let's figure out *liveness* over CFGs

- A variable `v` is `live` on an edge if there is a directed path from that edge to a `use` of `v` that does not go through any `def` of `v`.
- A variable is `live-in` at a node if it is live on any of the in-edges of that node.
- A variable is `live-out` at a node if it is live on any of the out-edges of that node.



- **Check:**
  - `a: {1->2, 4->5->2}`
  - `b: {2->4}`

Computation of liveness

- Say *live-in* of `n` is `in[n]`, and *live-out* of `n` is `out[n]`.
- We can compute `in[n]` and `out[n]` for any `n` as follows:

```
in[n] = use[n] ∪ (out[n] – def[n])

   out[n] = ∀s∈succ[n] ∪ in[s]
```

Called transfer functions.

Liveness as an iterative dataflow analysis

```
for each n
    in[n] = {}; out[n] = {}
repeat
    for each n
        in'[n] = in[n]; out'[n] = out[n]
        in[n] = use[n] ∪ (out[n] – def[n])
        out[n] = ∀s∈succ[n] ∪ in[s]
until in'[n] == in[n] and out'[n] == out[n] ∀n
```

Initialize

IDFA

Save previous values

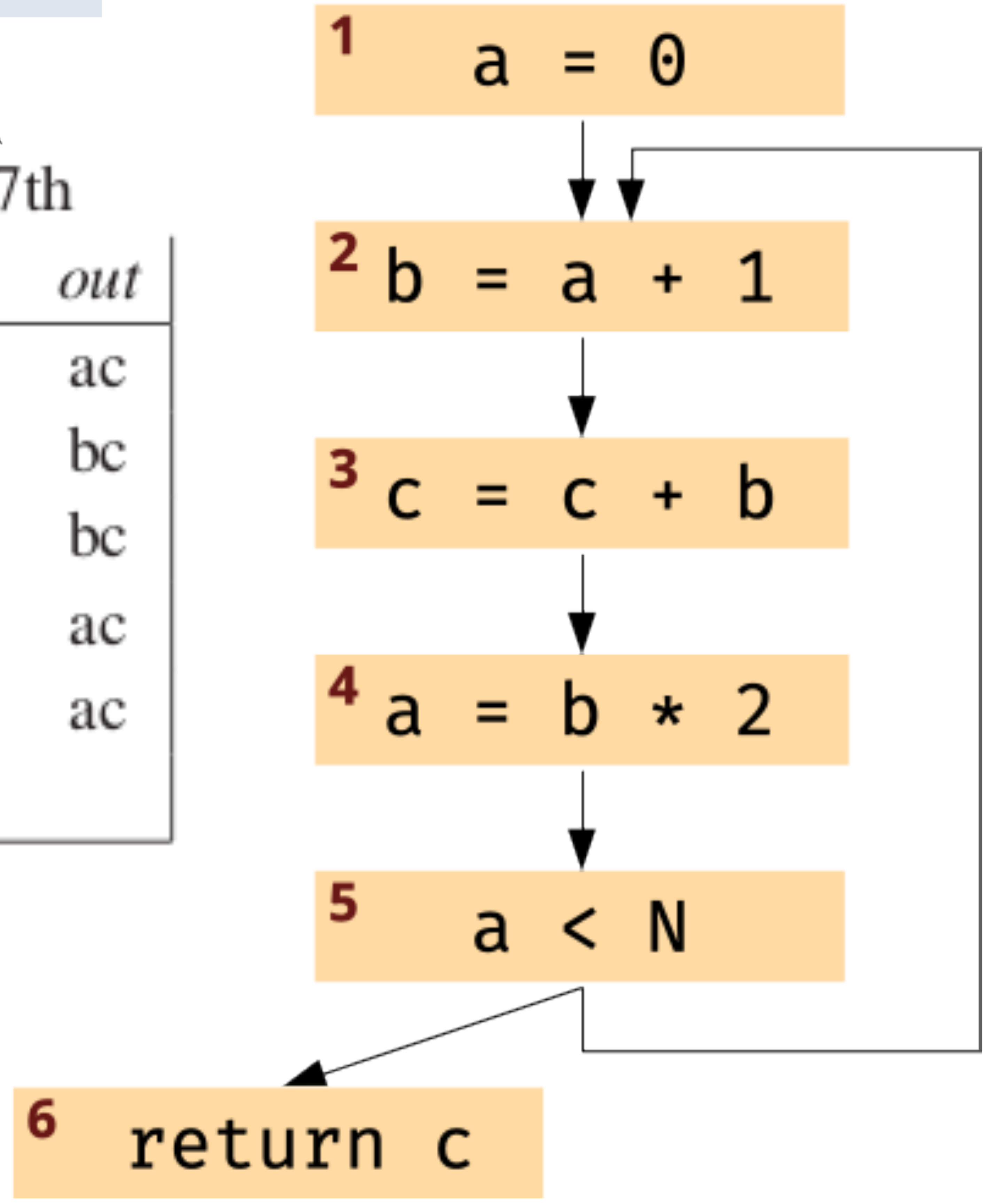
Compute new values

Repeat till fixed-point

Liveness analysis example

Fixed point

			1st		2nd		3rd		4th		5th		6th		7th	
	use	*def*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*
1		a				a		a		ac	c	ac	c	ac	c	ac
2	a	b	a		a	bc	ac	bc	ac	bc	ac	bc	ac	bc	ac	bc
3	bc	c	bc		bc	b	bc	b	bc	b	bc	b	bc	bc	bc	bc
4	b	a	b		b	a	b	a	b	ac	bc	ac	bc	ac	bc	ac
5	a		a	a	a	ac	ac	ac	ac	ac	ac	ac	ac	ac	ac	ac
6	c		c		c		c		c		c		c		c	

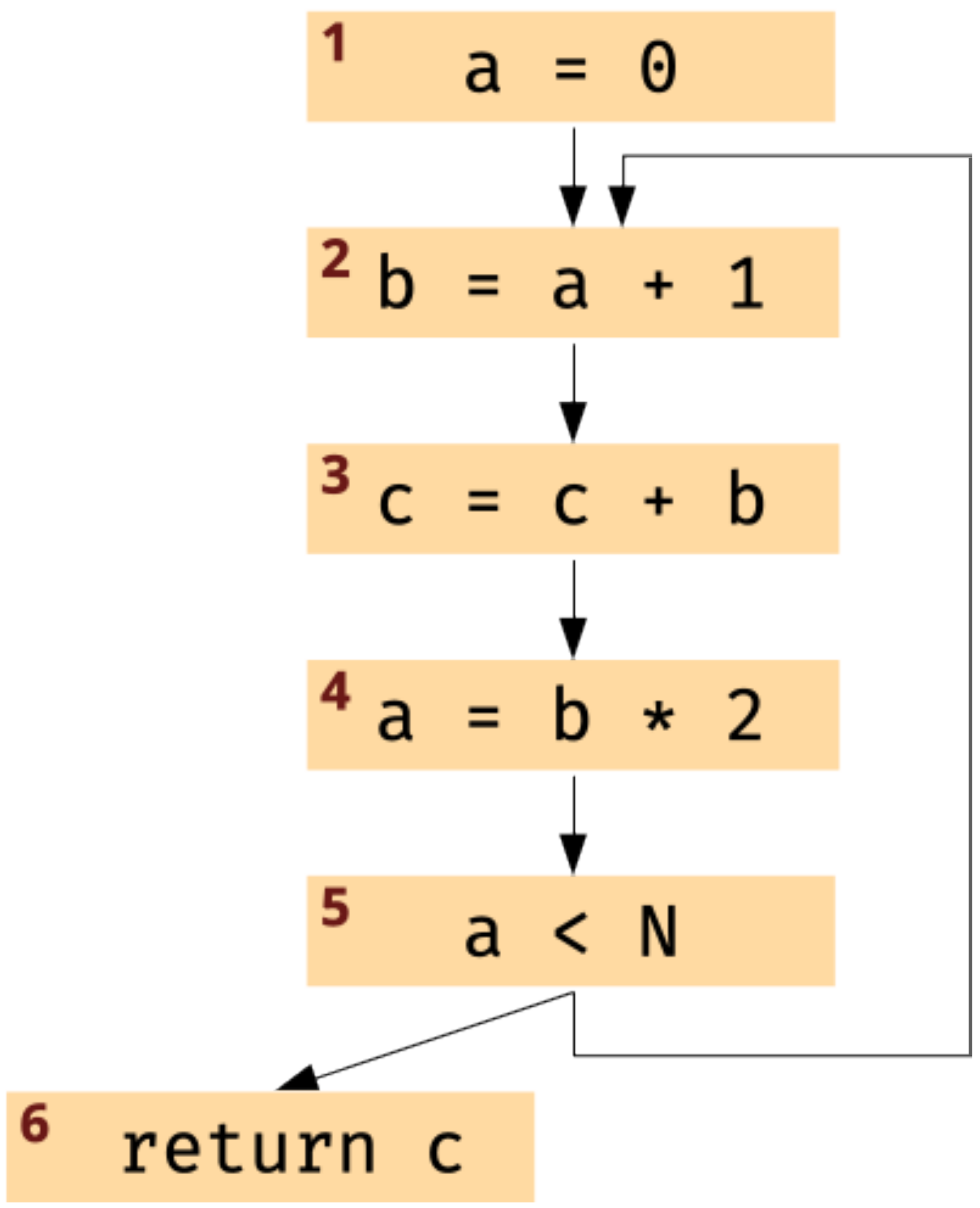



```
in[n] = use[n] ∪ (out[n] – def[n])

out[n] = ∀s∈succ[n] ∪ in[s]
```

In backward order

			1st		2nd		3rd	
	use	*def*	*out*	*in*	*out*	*in*	*out*	*in*
6	c			c		c		c
5	a		c	ac	ac	ac	ac	ac
4	b	a	ac	bc	ac	bc	ac	bc
3	bc	c	bc	bc	bc	bc	bc	bc
2	a	b	bc	ac	bc	ac	bc	ac
1		a	ac	c	ac	c	ac	c



- Fixed point in only 3 iterations!
- The order of processing statements matters for efficiency (not for correctness).

```
in[n] = use[n] ∪ (out[n] – def[n])

out[n] = ∀s∈succ[n] ∪ in[s]
```

Complexity of our liveness computation algorithm

```
for each n
    in[n] = {}; out[n] = {}
repeat
    for each n
        in'[n] = in[n]; out'[n] = out[n]
        in[n] = use[n] ∪ (out[n] – def[n])
        out[n] = ∀s∈succ[n] ∪ in[s]
until in'[n] == in[n] and out'[n] == out[n] ∀n
```

- For input program of size N
  - ≤N nodes in CFG
    - ⇒ N variables
    - ⇒ N elements per in/out
    - ⇒ O(N) time per set union
  - `for` loop performs constant number of set operations per node
    - ⇒ $O(N^2)$ time for `for` loop
  - Each iteration of `for` loop can only add to each set (monotonicity)
  - Sizes of all in and out sets sum to $2N^2$, thus bounding the number of iterations of the `repeat` loop
    - ⇒ worst-case complexity of $O(N^4)$
  - Much less in practice (usually O(N) or $O(N^2)$) if ordered properly.

Least fixed points

- There is often more than one solution for a given dataflow problem.
  - Any solution to dataflow equations is a conservative approximation.
- Conservatively assuming a variable is live does not break the program:
  - Just means more registers may be needed.
- Assuming a variable is dead when really live <u>will</u> break things.

- Many possible solutions; but we want the smallest: the least fixed point.
- The iterative algorithm computes this least fixed point.

Recall our IDFA algorithm

```
for each n
    in[n] = {}; out[n] = {}
repeat
    for each n
        in'[n] = in[n]; out'[n] = out[n]
        in[n] = use[n] ∪ (out[n] − def[n])
        out[n] = ∀s∈succ[n] ∪ in[s]
until in'[n] == in[n] and out'[n] == out[n] ∀n
```

Initialize

IDFA

Save previous values

Compute new values

Repeat till fixed-point

- Do we need to process all the nodes in each iteration?

Worklist-based Implementation of IDFA

- Initialize a worklist of statements

- Forward analysis:
  - Prefer to start with the entry node
  - If `OUT(n)` changes, then add `succ(n)` to the worklist
- Backward analysis:
  - Prefer to start with the exit node
  - If `IN(n)` changes, then add `pred(n)` to the worklist

- In both the cases, iterate till fixed point.

Writing an IDFA (Cont.)

- Confluence (at control-flow merges):
  - union
  - intersection

- Requirement for termination:
  - finiteness of the set of possible dataflow values
  - unidirectional growth/shrinkage, called *monotonicity*
    - *for the dataflow values at each statement*

Liveness analysis revisited

- Direction:
  - Backward
- Confluence operation:
  - Union
- Flow functions:
  - `out[n] = ∀s∈succ[n] ∪ in[s]`
  - `in[n] = use[n] ∪ (out[n] – def[n])`

Common subexpressions revisited

➤ **Idea:** If a program computes the same value multiple times, reuse the value.

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

→

```
t = b + c;
a = t;
c = t;
d = b + c;
```

➤ Subexpressions can be reused until operands are redefined.

➤ Say given a node `n`, the expressions computed at `n` are denoted as `gen(n)` and the ones killed (operands redefined) at `n` are denoted as `kill(n)`.

Computing common subexpressions as an IDFA

- Direction:
  - Forward
- Confluence operation:
  - Intersection
- Flow functions:
  - `in[n] = ∀p∈pred[n] ∩ out[p]`
  - `out[n] = gen[n] ∪ (in[n] – kill[n])`

Are we efficient enough?

- When can IDFAs take a lot of time?
- Which operations could be **expensive**?
  - Confluence
  - Comparison (equality check)

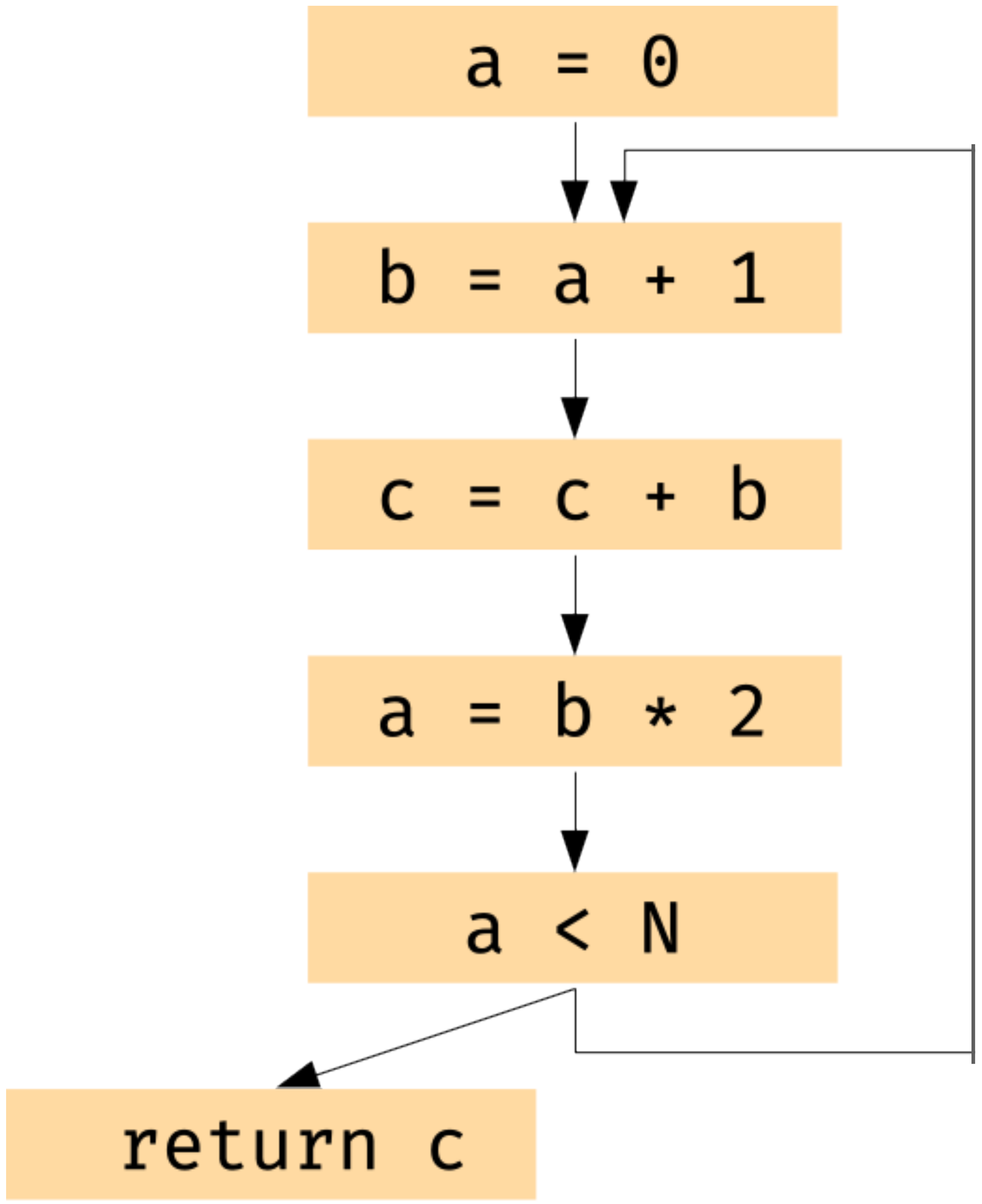
```
for each n
   in[n] = {}; out[n] = {}              Initialize
repeat
   for each n
      in'[n] = in[n]; out'[n] = out[n]          Save previous values
      in[n] = use[n] ∪ (out[n] – def[n])        Compute
      out[n] = ∀s∈succ[n] ∪ in[s]               new values
until in'[n] == in[n] and out'[n] == out[n] ∀n
                                    Repeat till fixed-point
```

IDFA

- Compilers may have to perform several IDFAs.

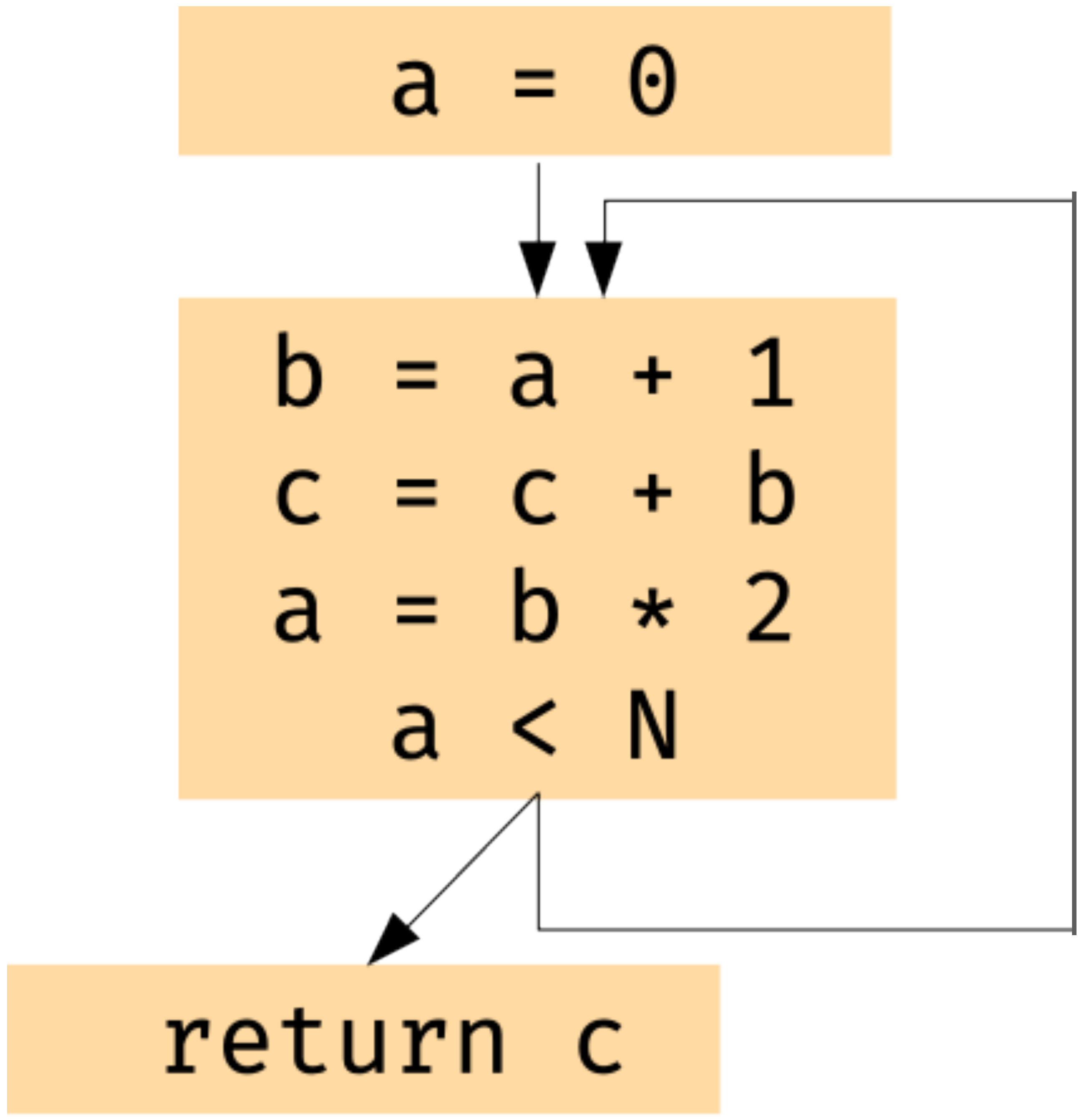
- How can we make an IDFA more efficient (perhaps with some loss of precision)?

Basic Blocks



Each instruction as a node

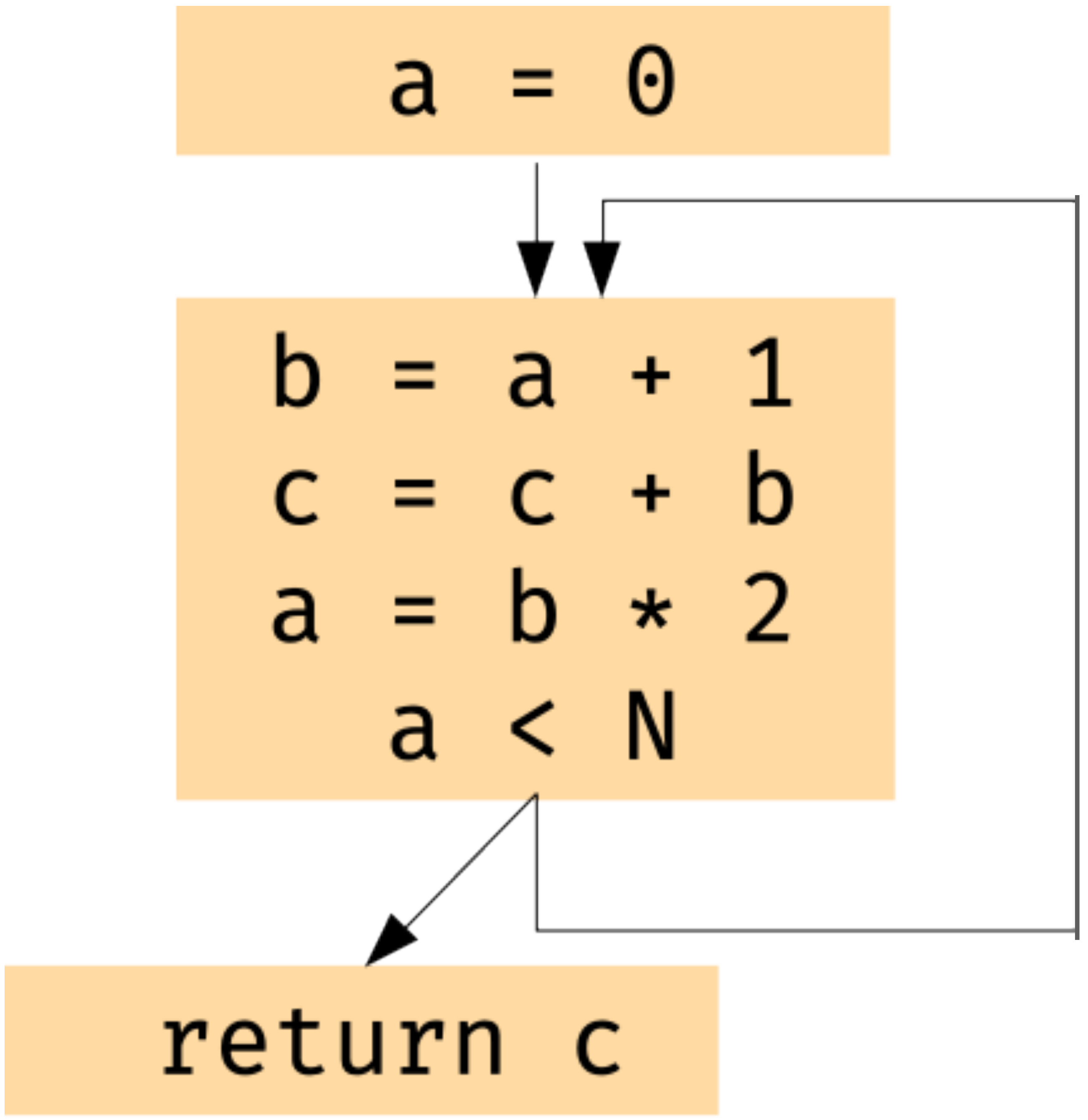
```
    a = 0
L1: b = a + 1
    c = c + b
    a = b * 2
    if a < N goto L1
    return c
```



Using basic blocks

Basic Blocks (Cont.)

- **Idea:**
  - Once execution enters a basic block, all statements are executed in sequence.
  - Single-entry, single-exit region
- **Details:**
  - Starts with a label
  - Ends with one or more branches
  - Edges may be labeled with predicates
    - True/false
    - Exceptions
- **Key:** Improve efficiency, with reasonable precision.



Using basic blocks

Have you got a compiler's eyes yet?

```
S1: y = 1;
S2: y = 2;
S3: x = y;
```

➤ What's the advantage if the above program is rewritten as follows?

```
S1: y1 = 1;
S2: y2 = 2;
S3: x = y2;
```

<u>Next Class</u>

Static Single Assignment (SSA) Form

➤ Def-use becomes explicit; analysis can become faster.