Dataflow Analysis

Overview

today's lecture

Control flow graphs

- intermediate representation

Overview
today's lecture

Control flow graphs

- intermediate representation

Non-local optimizations

- dead code elimination
- constant & copy propagation
- common subexpression elimination

Overview
today's lecture

Control flow graphs

- intermediate representation

Non-local optimizations

- dead code elimination
- constant & copy propagation
- common subexpression elimination

Data flow analyses

- liveness analysis
- reaching definitions
- available expressions

Overview

today's lecture

Control flow graphs

- intermediate representation

Non-local optimizations

- dead code elimination
- constant & copy propagation
- common subexpression elimination

Data flow analyses

- liveness analysis
- reaching definitions
- available expressions

Optimizations (again)

I

Optimizations

Optimizing Compilers
optimization

Fully optimizing compiler

- Opt(P) produces smallest program with same I/O behavior
- smallest program for non-terminating programs without I/O:
  Loop = (L : goto L)
- solves halting problem: Opt(Q) = Loop iff Q halts

Optimizing compiler

- produces program with same I/O behavior
- that is smaller or faster

Full employment theorem for compiler writers

- there is always a better optimizing compiler

Local vs Non-Local Optimizations

optimization

Local optimizations

- peephole optimization
- constant folding

```
iload 1
iload 1
mul
```

```
iload 1
dup
mul
```

```
(3 + 4) * x
```

```
7 * x
```

- pattern match + rewrite

Non-local optimizations

- constant propagation
- dead-code elimination

- require information from different parts of program

Program Optimizations
optimization

Register allocation

- keep non-overlapping temporaries in same register

Common-subexpression elimination

- if expression is computed more than once, eliminate one computation

Dead-code elimination

- delete computation whose result will never be used

Constant folding

- if operands of expression are constant, do computation at compile-time

And many more possible optimizations

Dead Code Elimination

example

```
a ← 0
b ← a + 1
c ← c + b
a ← 2 * b
return c
```

delete computation whose result will never be used

Dead Code Elimination
example

```
a ← 0
b ← a + 1
c ← c + b
a ← 2 * b
return c
```

⇨

```
a ← 0
b ← a + 1
c ← c + b

return c
```

delete computation whose result will never be used

Constant Propagation

example

```
a ← 0
b ← a + 1
c ← c + b
a ← 2 * b
return c
```

substitute reference to constant valued variable

Constant Propagation
example

```
a ← 0
b ← a + 1
c ← c + b
a ← 2 * b
return c
```

⇒

```
a ← 0
b ← 0 + 1
c ← c + b
a ← 2 * b
return c
```

substitute reference to constant valued variable

Constant Propagation + Folding

example

```
a ← 1
b ← a + 1
c ← c + b
a ← 2 * b
return c
```

Constant Propagation + Folding

example

```
a ← 1
b ← a + 1
c ← c + b
a ← 2 * b
return c
```

```
a ← 0
b ← 2
c ← c + 2
a ← 4
return c
```

Copy Propagation

example

```
a ← e
b ← a + 1
c ← c + b
a ← 2 * b
return c
```

Copy Propagation
example

```
a ← e
b ← a + 1
c ← c + b
a ← 2 * b
return c
```

⇒

```
a ← e
b ← e + 1
c ← c + b
a ← 2 * b
return c
```

Common Subexpression Elimination
example

```
c ← a + b
d ← 1
e ← a + b
```

if expression is computed more than once, eliminate one computation

Common Subexpression Elimination

example

```
c ← a + b
d ← 1
e ← a + b
```

```
x ← a + b
c ← x
d ← 1
e ← x
```

if expression is computed more than once, eliminate one computation

Intraprocedural Global Optimization
optimization

Terminology

- Intraprocedural: within a single procedure or function
- Global: spans all statements with procedure
- Interprocedural: operating on several procedures at once

Recipe

- Dataflow analysis: traverse flow graph, gather information
- Transformation: modify program using information from analysis

II

Control-flow Graphs

Intermediate Language
quadruples

Store

$a \leftarrow b \oplus c$

$a \leftarrow b$

Memory access

$a \leftarrow M[b]$

$M[a] \leftarrow b$

Functions

$f(a_1, \ldots, a_n)$

$b \leftarrow f(a_1, \ldots, a_n)$

Jumps

```
L:
goto L
if a ⊗ b
    goto L1
else
    goto L2
```

Control-Flow Graphs

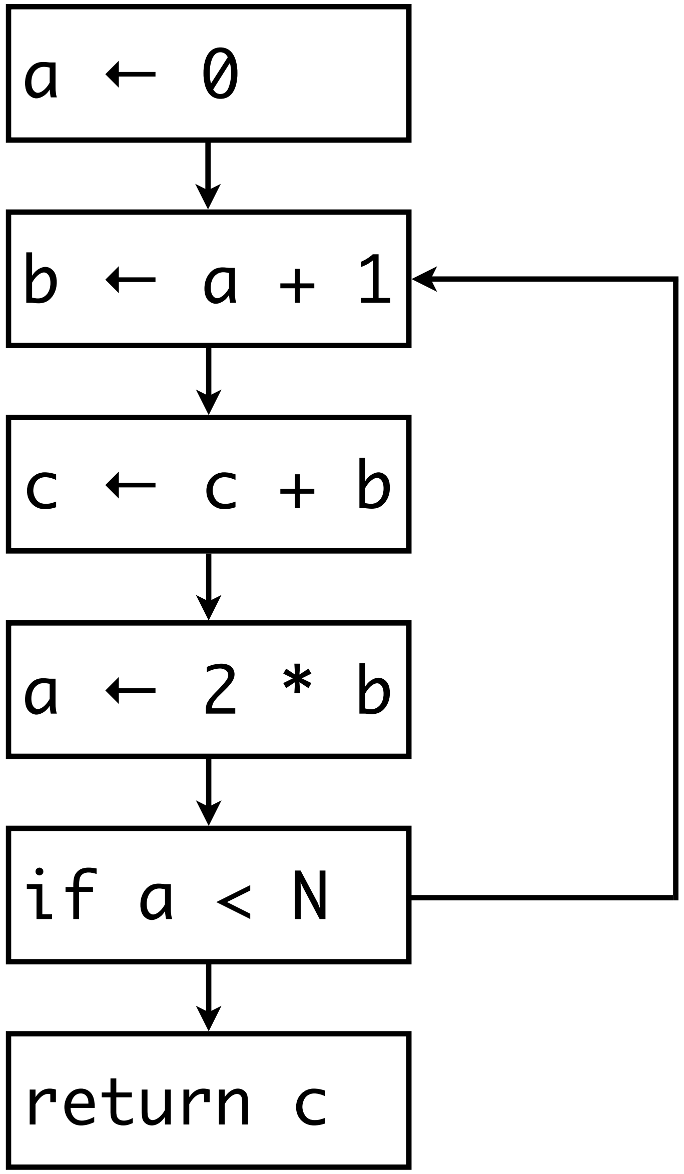
example

```
    a ← 0
L1: b ← a + 1
    c ← c + b
    a ← 2 * b
    if a < N
       goto L1
    else
       goto L2
L2: return c
```

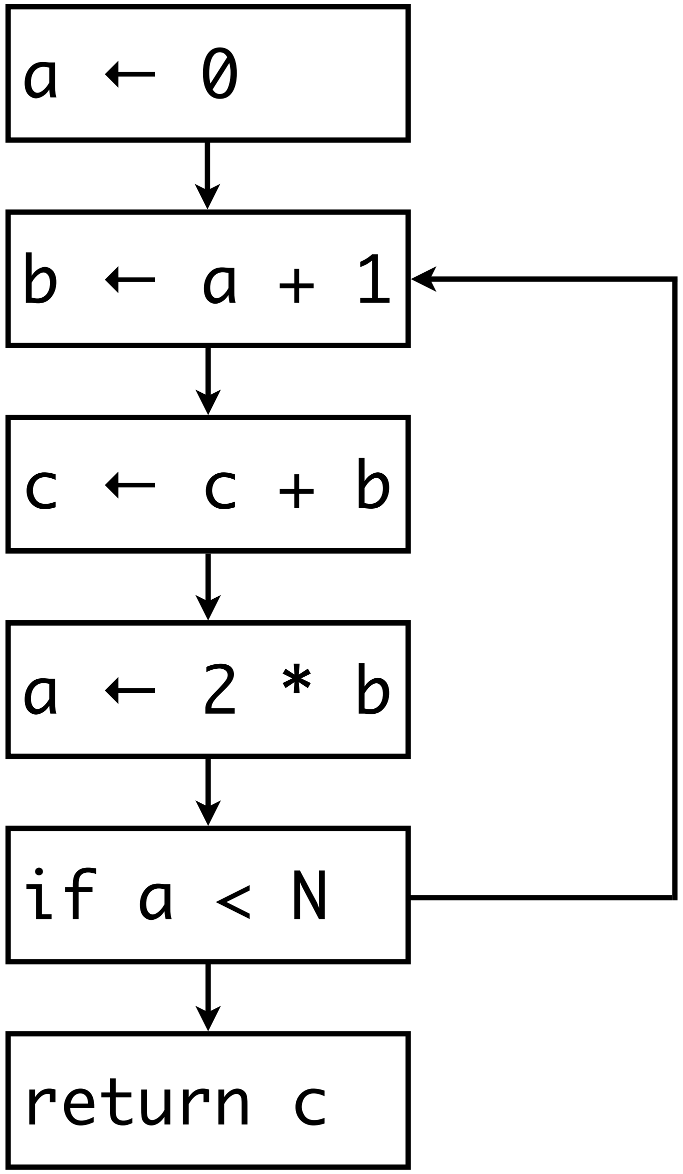
Control-Flow Graphs

example

```
    a ← 0
L1: b ← a + 1
    c ← c + b
    a ← 2 * b
    if a < N
       goto L1
    else
       goto L2
L2: return c
```

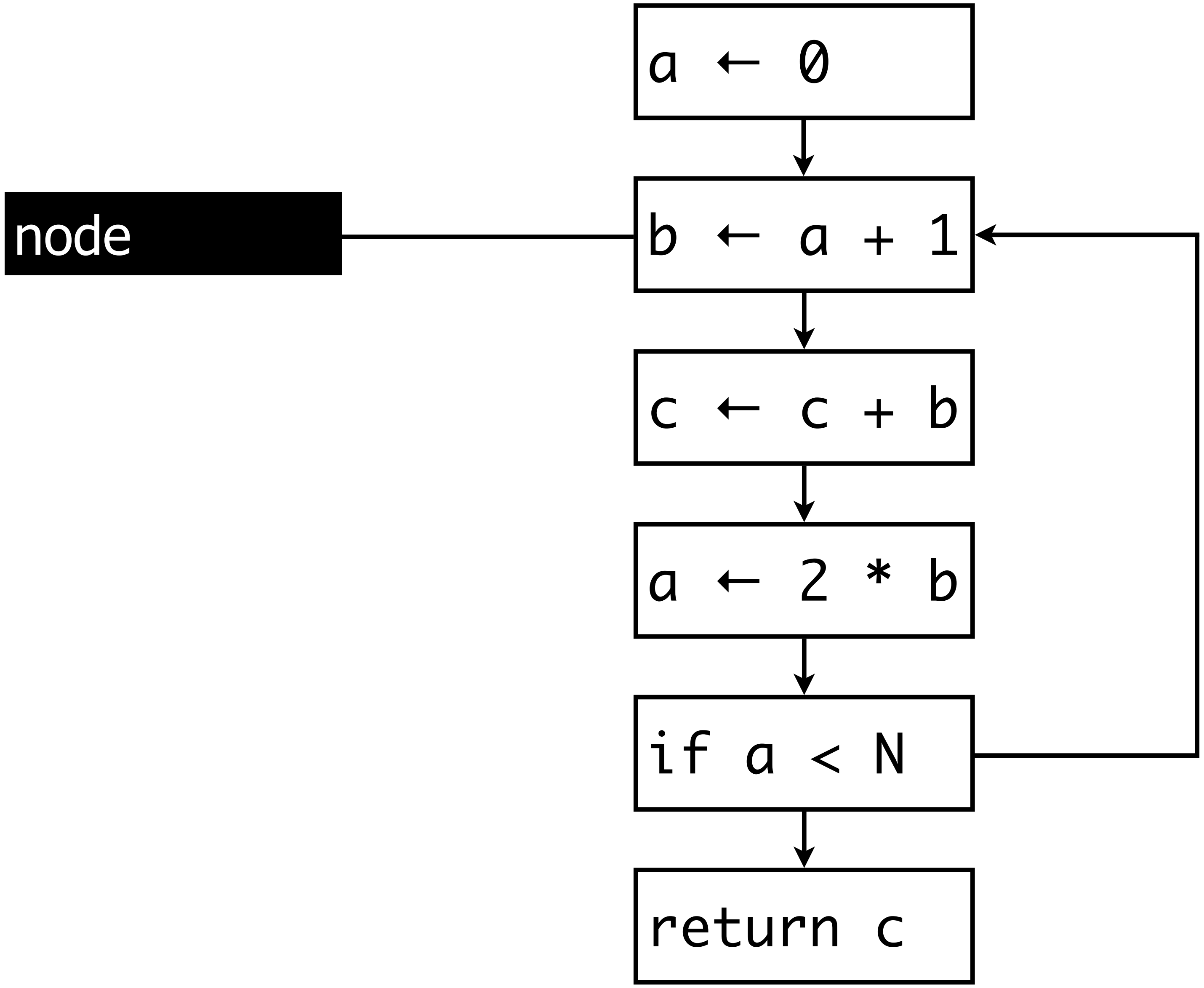


Control-Flow Graphs
terminology



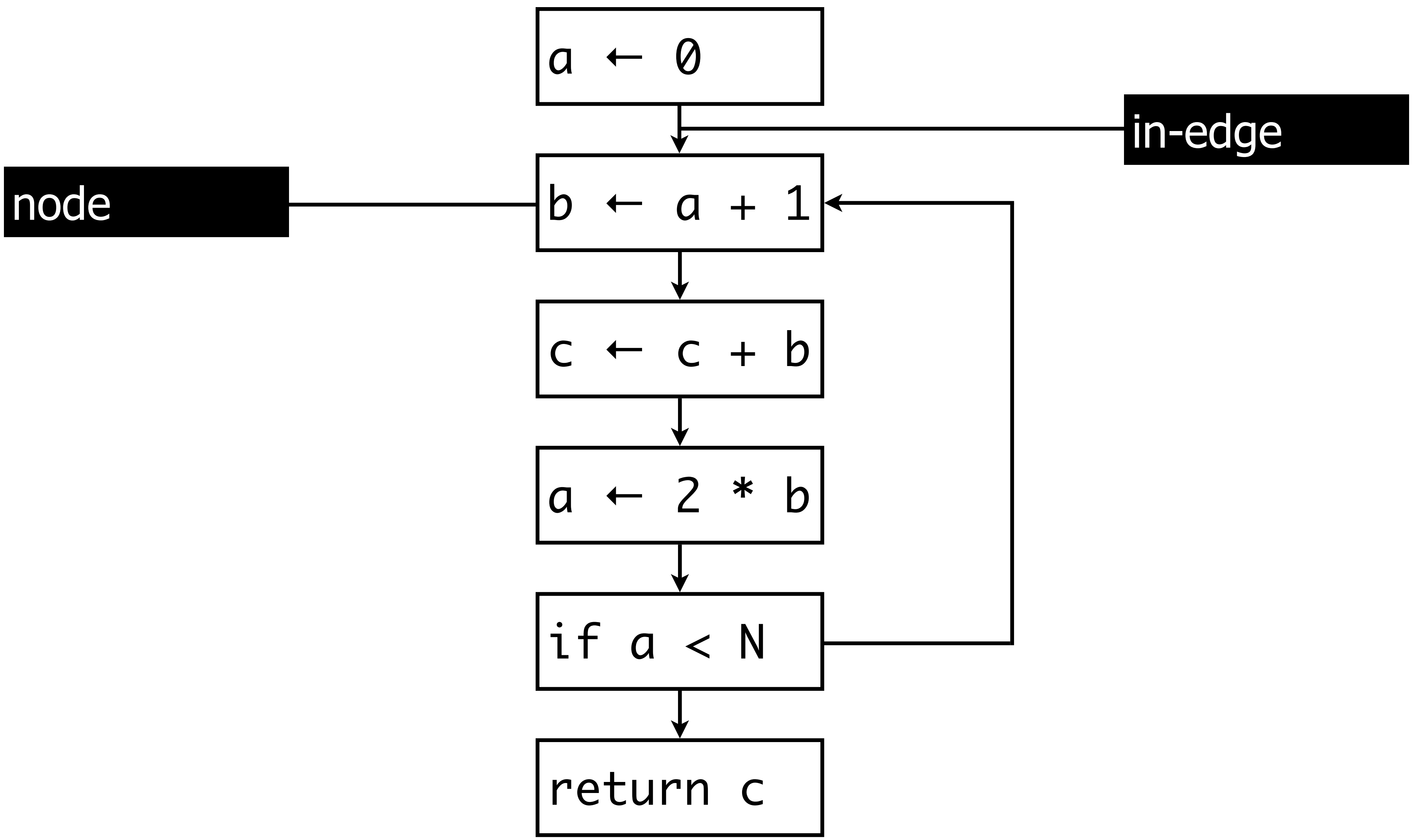
Control-Flow Graphs

terminology

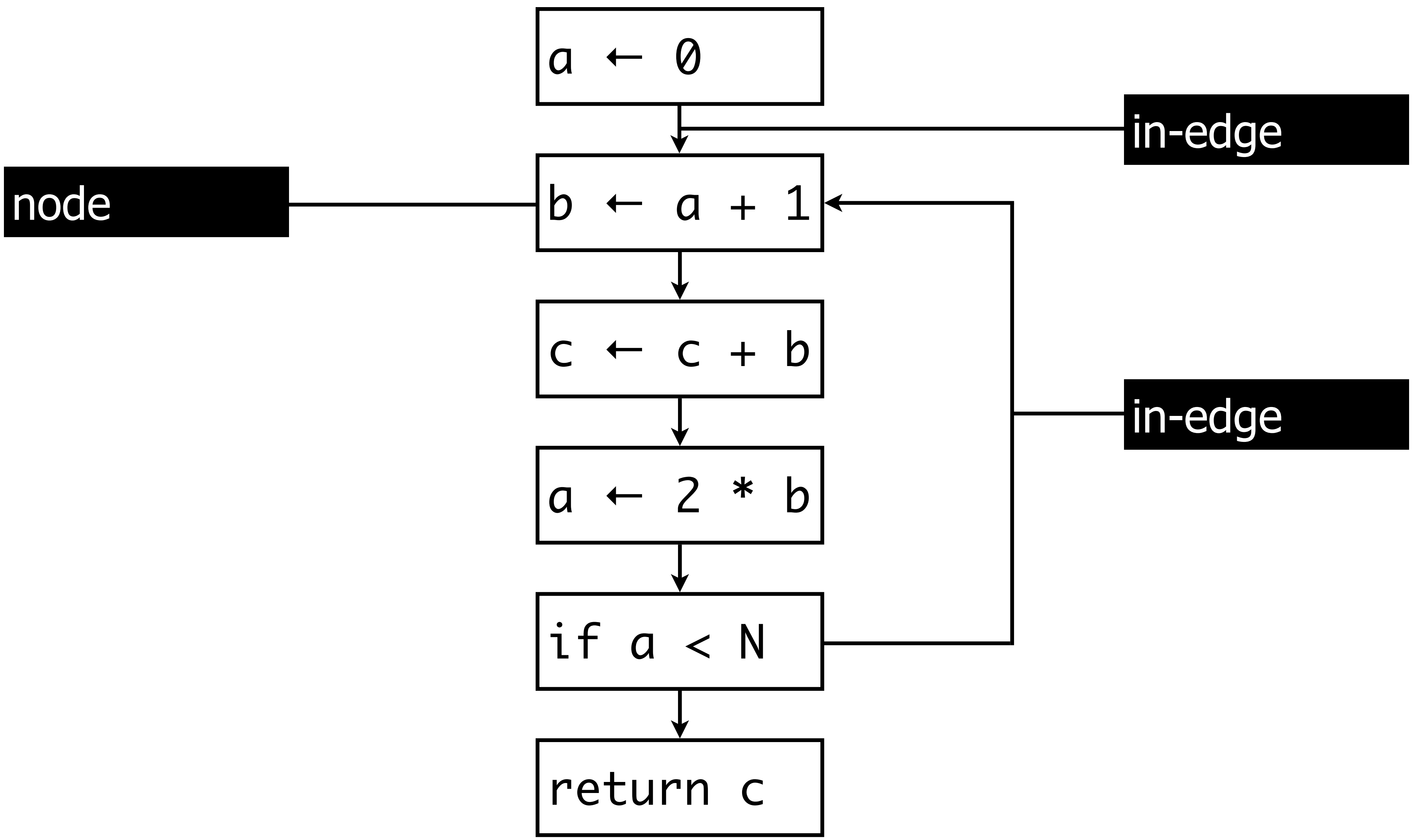


Control-Flow Graphs

terminology

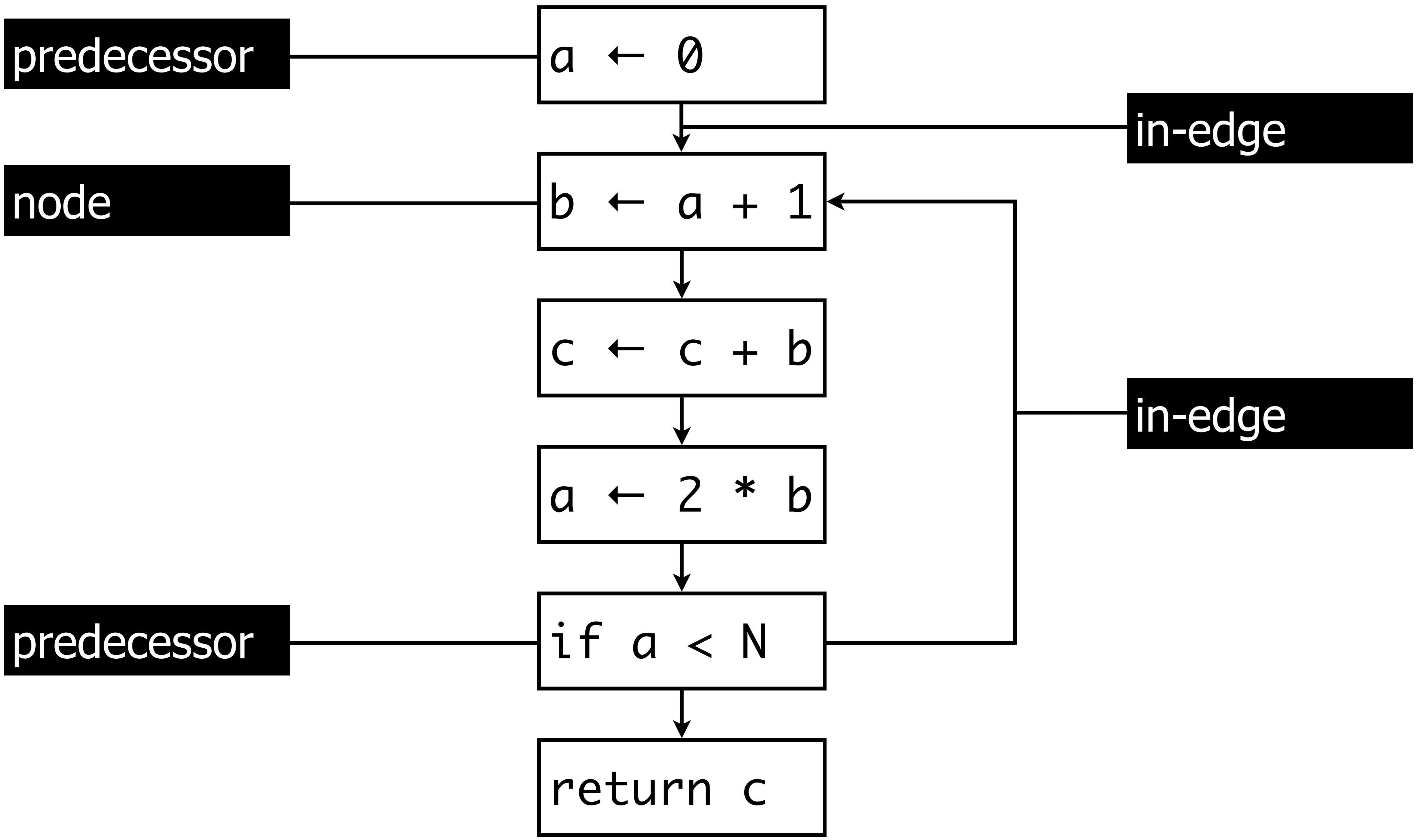


Control-Flow Graphs
terminology



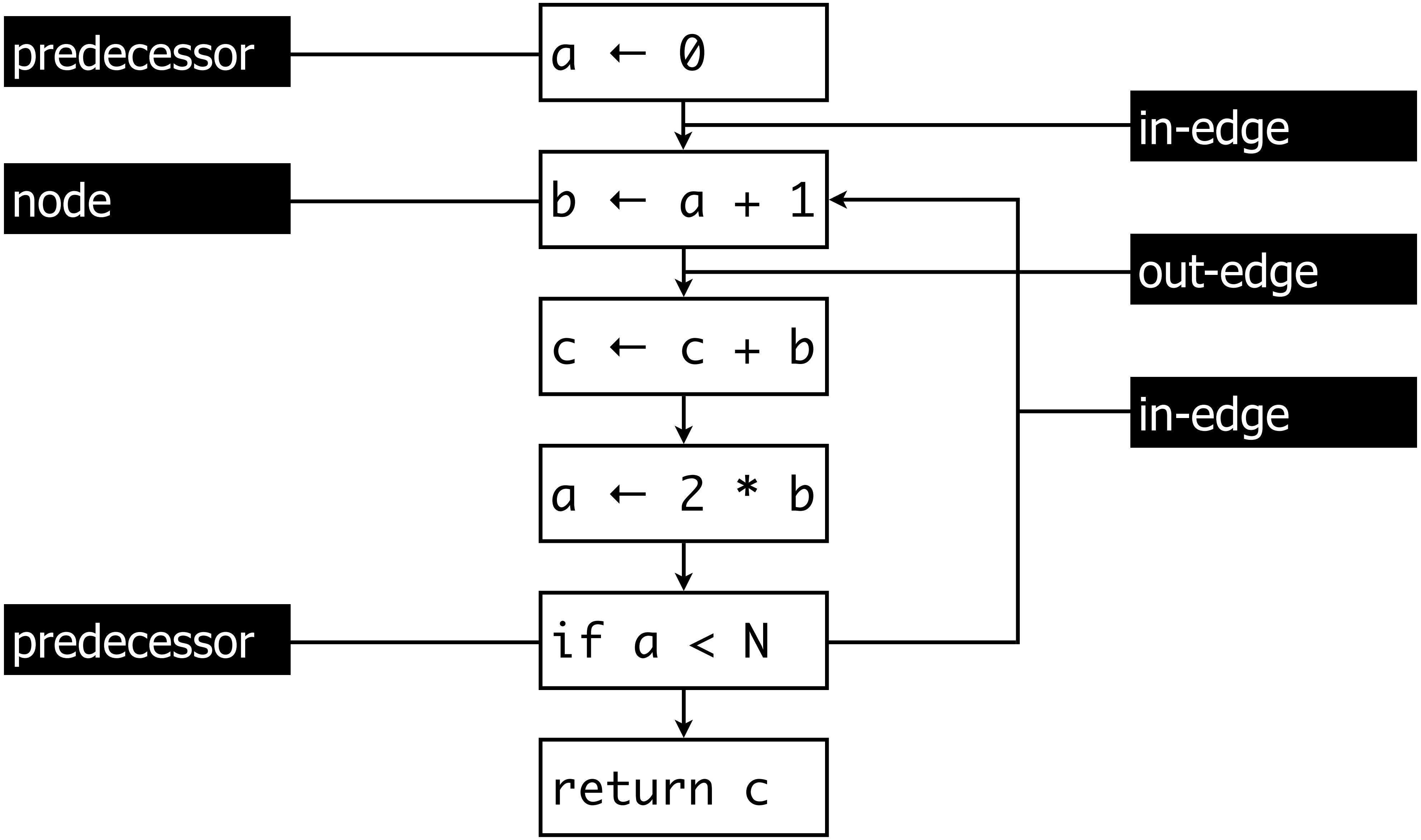
Control-Flow Graphs

terminology



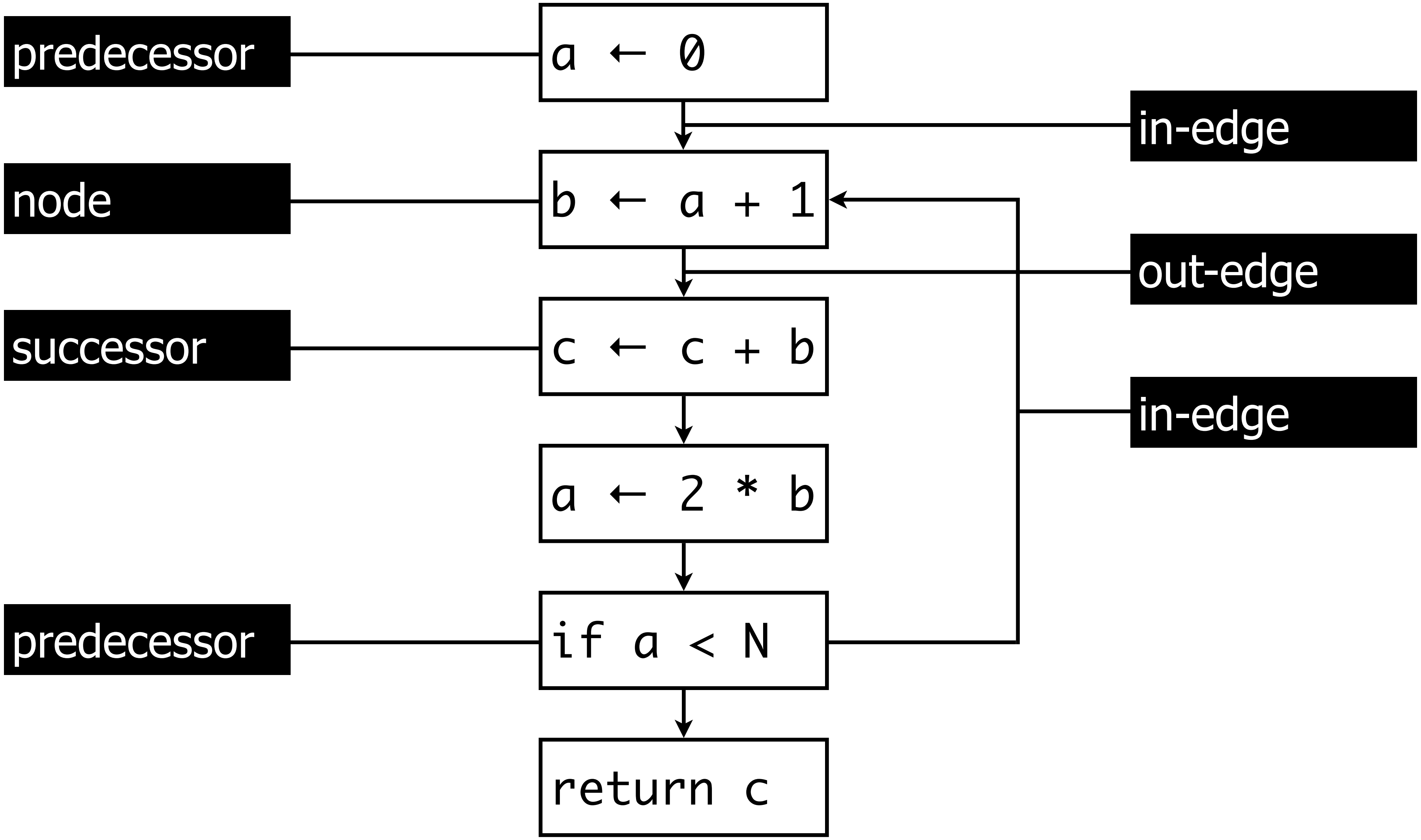
Control-Flow Graphs

terminology



Control-Flow Graphs

terminology



III

Liveness Analysis

Liveness Analysis
definition

Motivation: register allocation

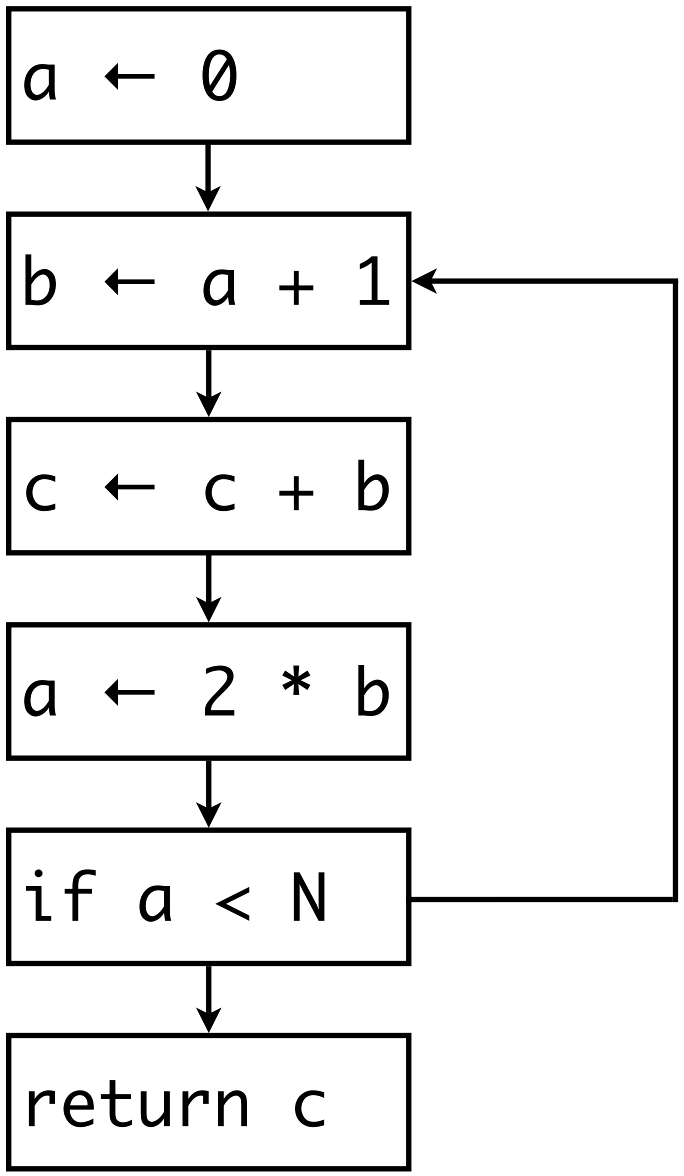
- intermediate code with unbounded number of temporary variables
- map to bounded number of registers
- if two variables are not 'live' at the same time, store in same register

Liveness

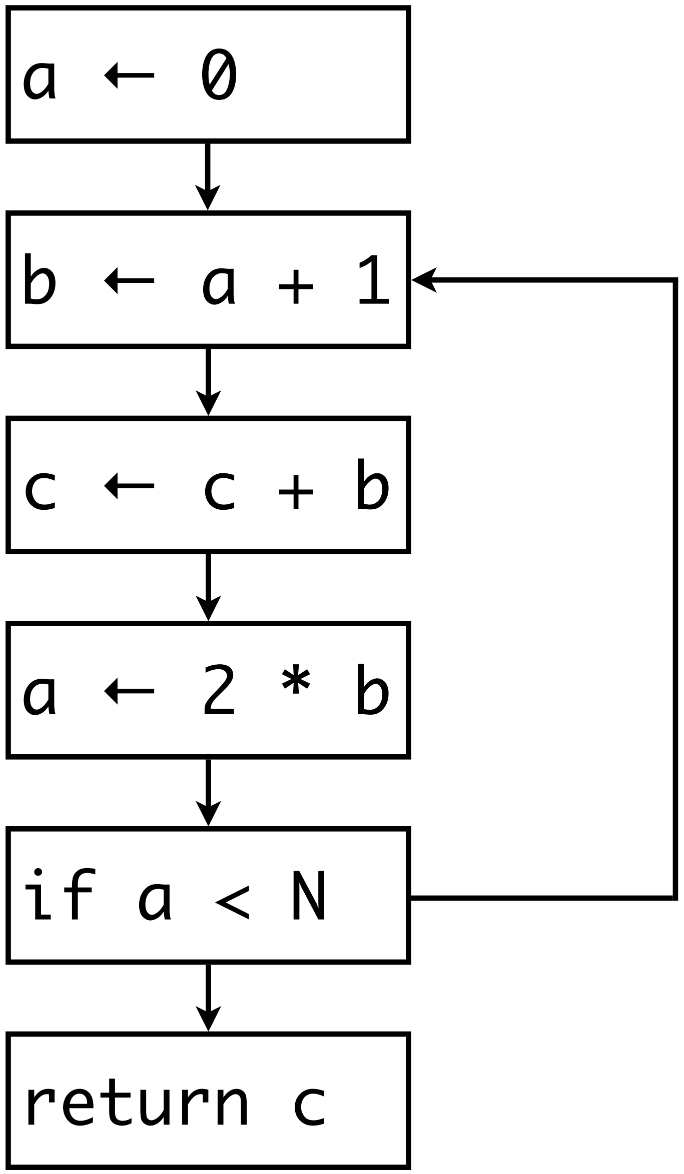
- a variable is **live** if it holds a value that may be needed in the future

Register Allocation

example



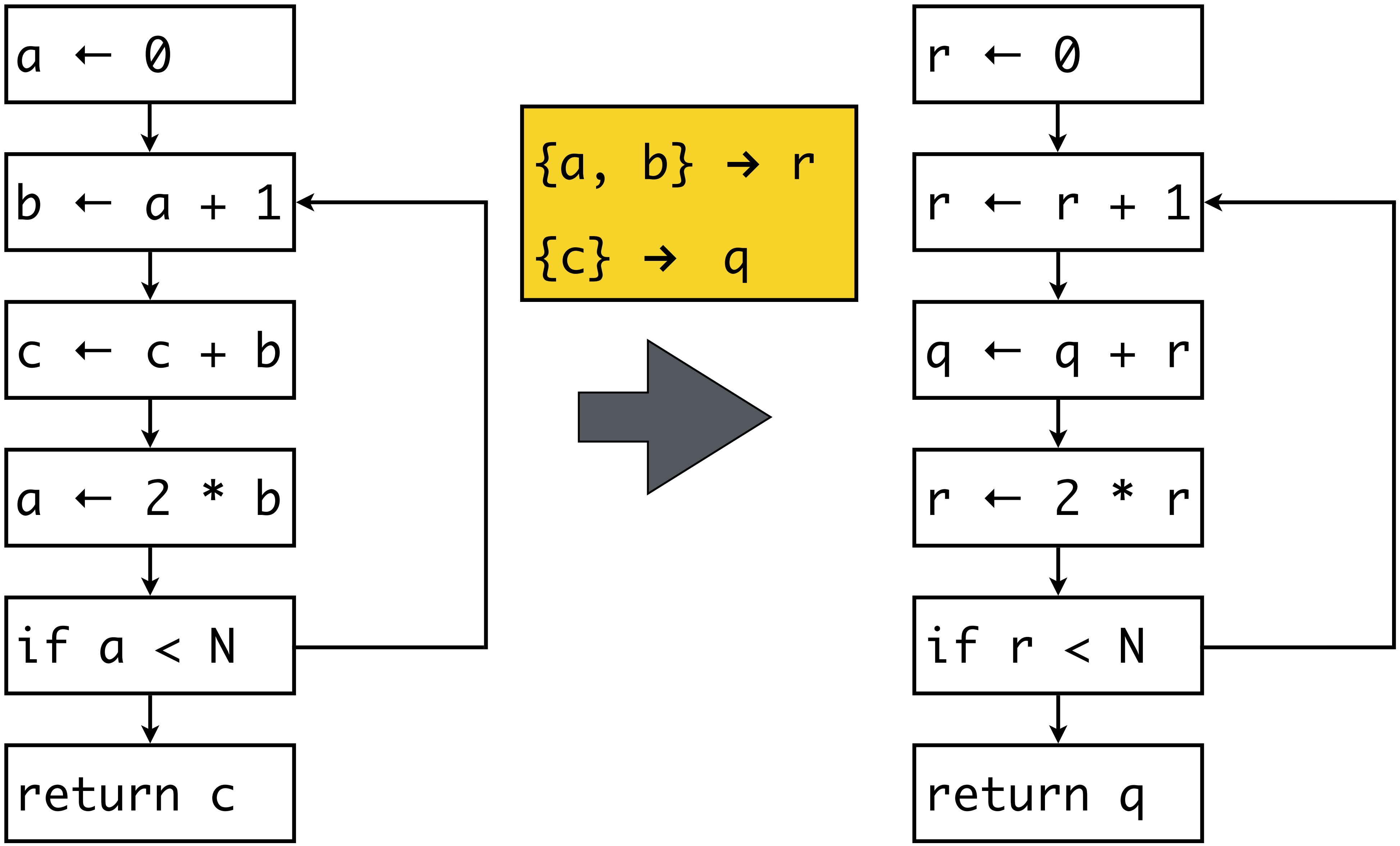
Register Allocation
example



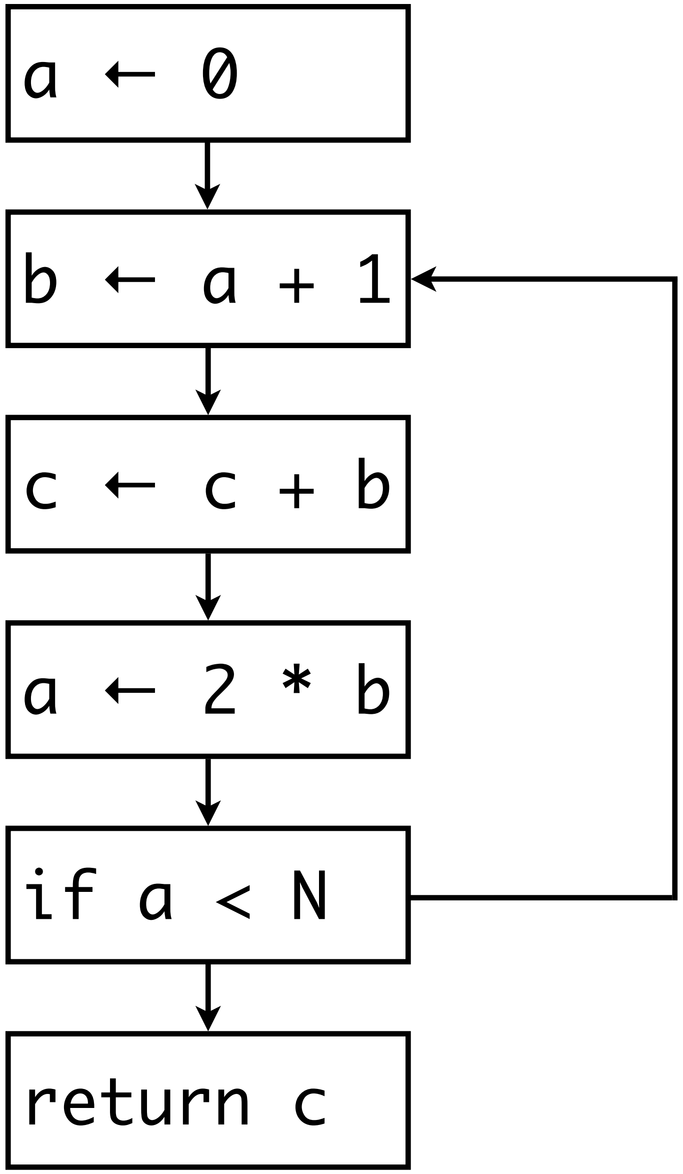
{a, b} → r
{c} → q

Register Allocation

example

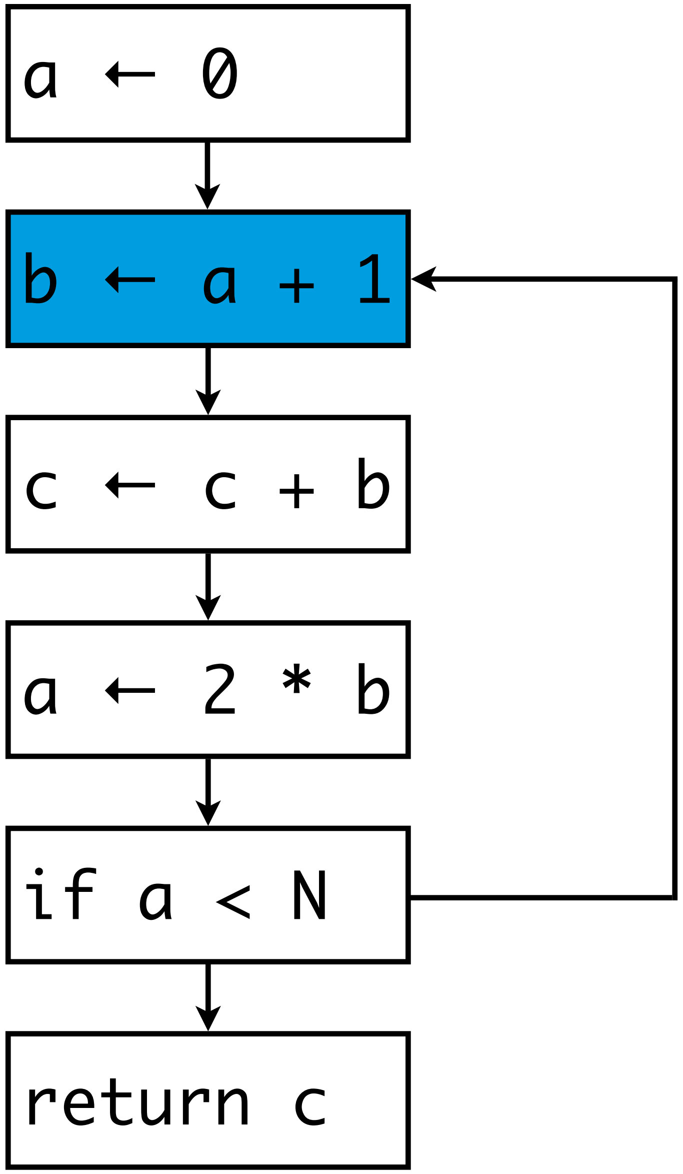


Liveness Analysis
example



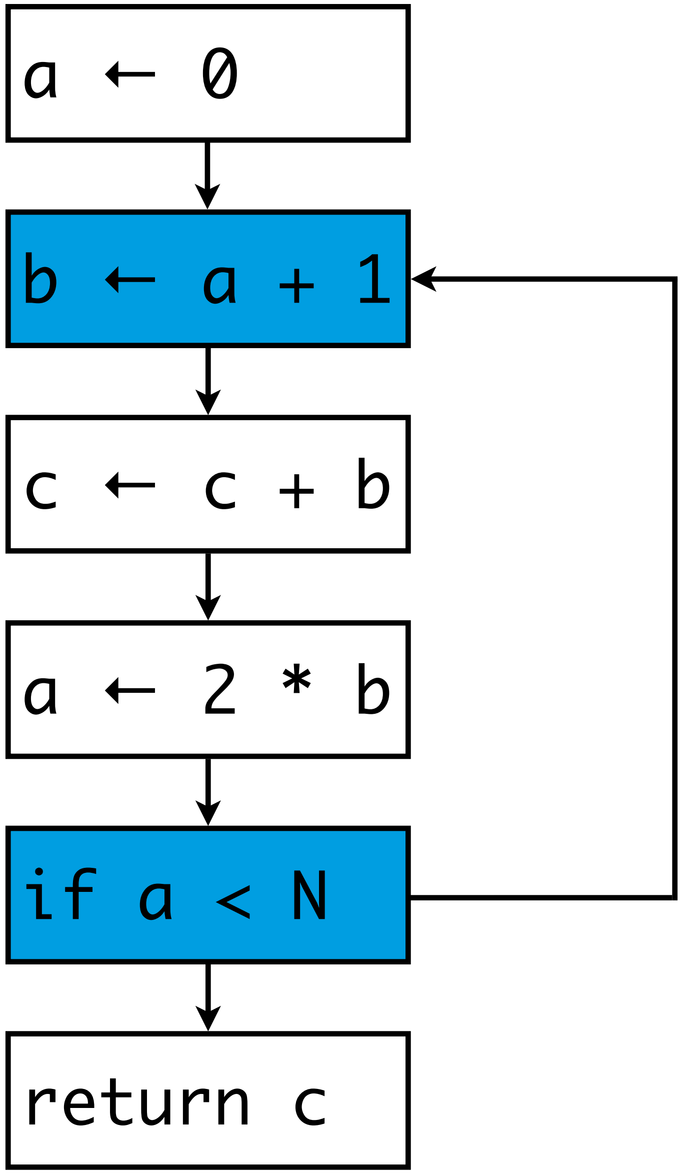
Liveness Analysis

example



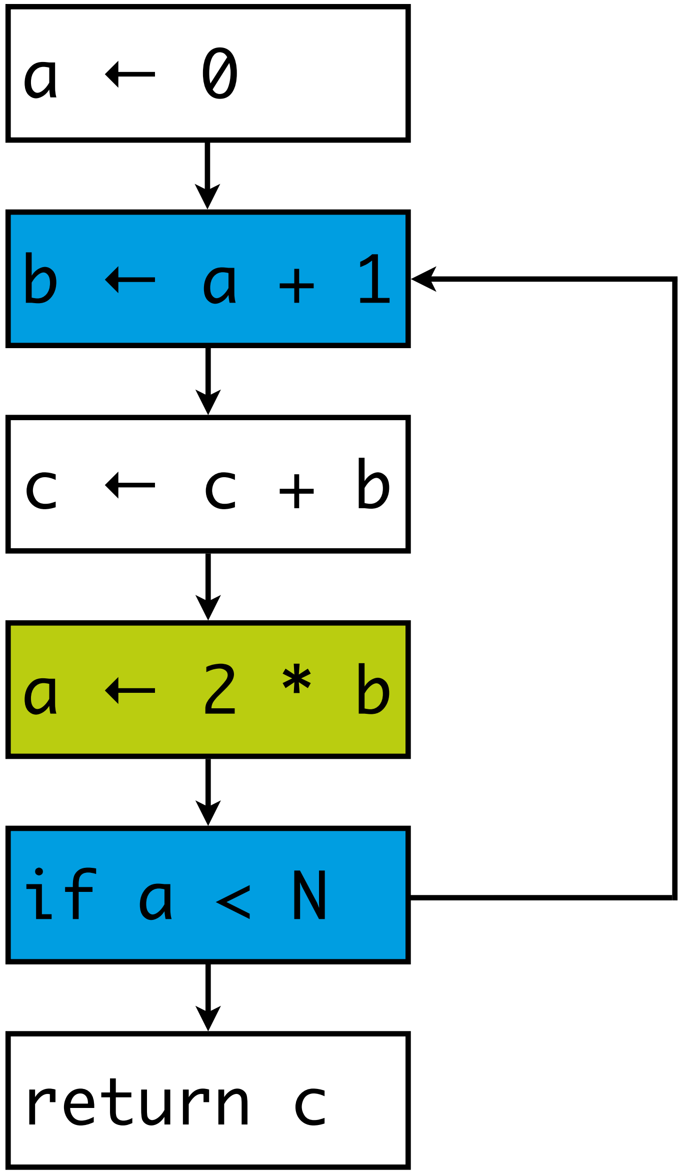
Liveness Analysis

example



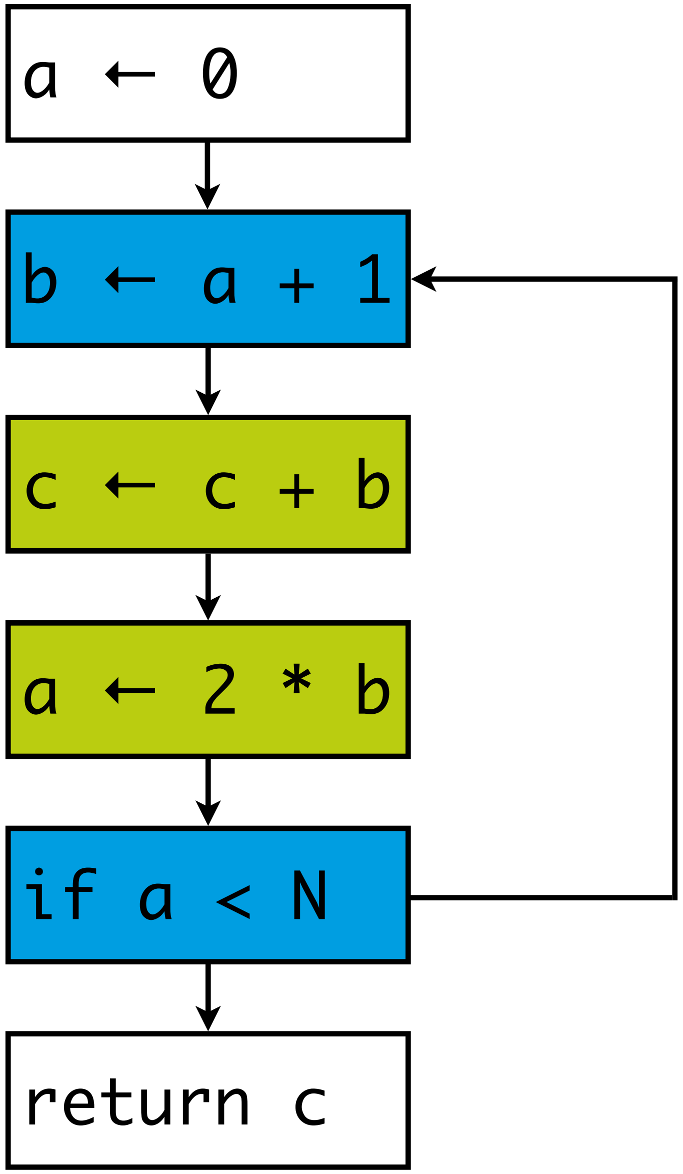
Liveness Analysis

example



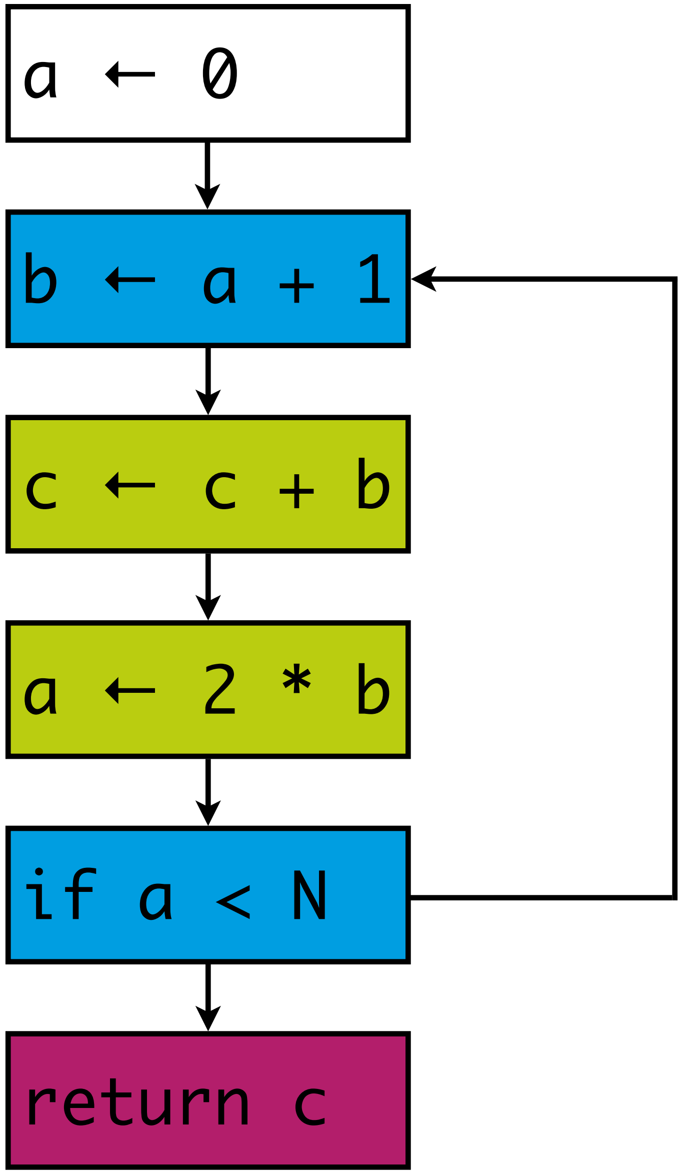
Liveness Analysis

example



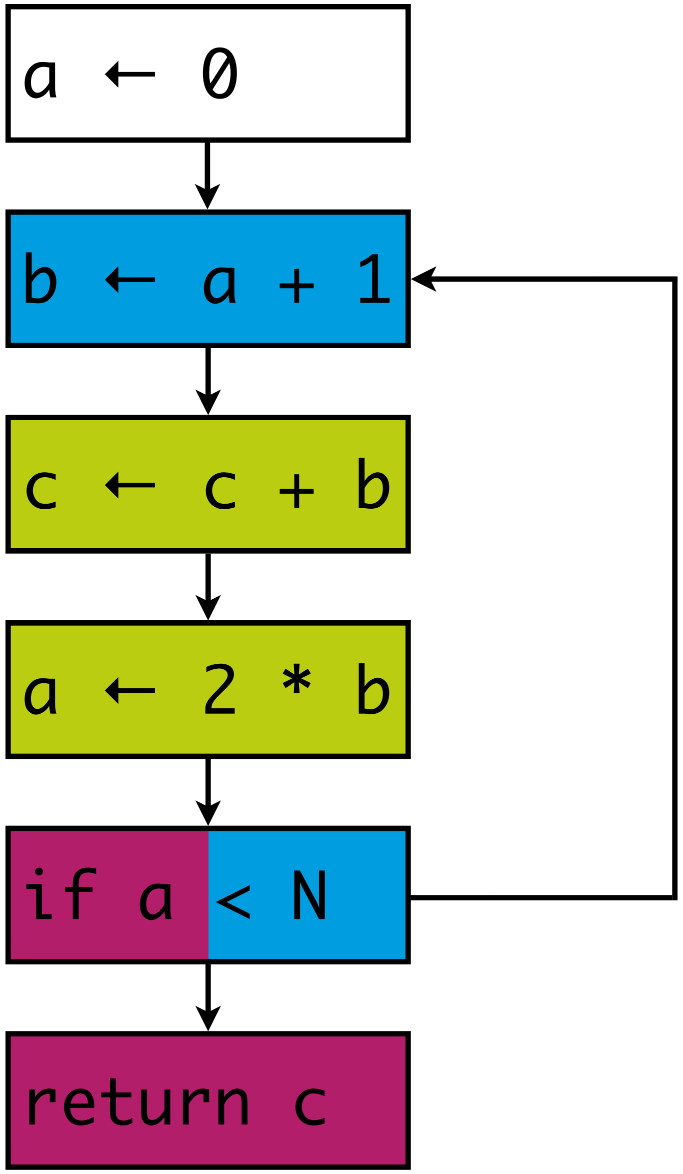
Liveness Analysis

example



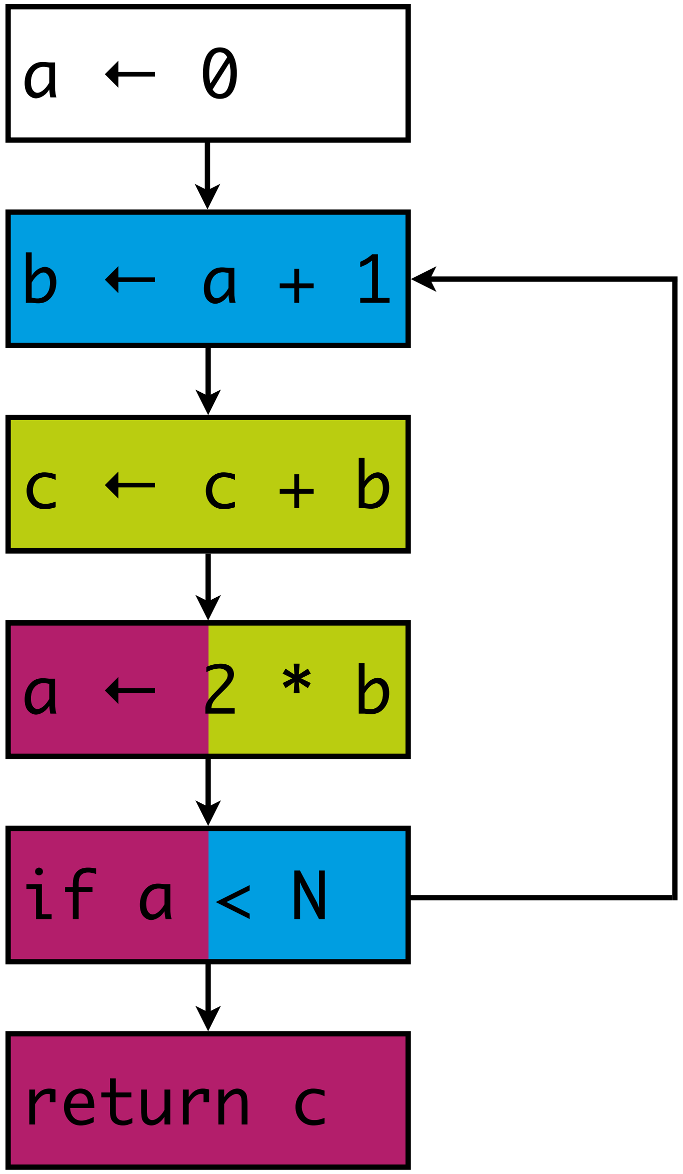
Liveness Analysis

example



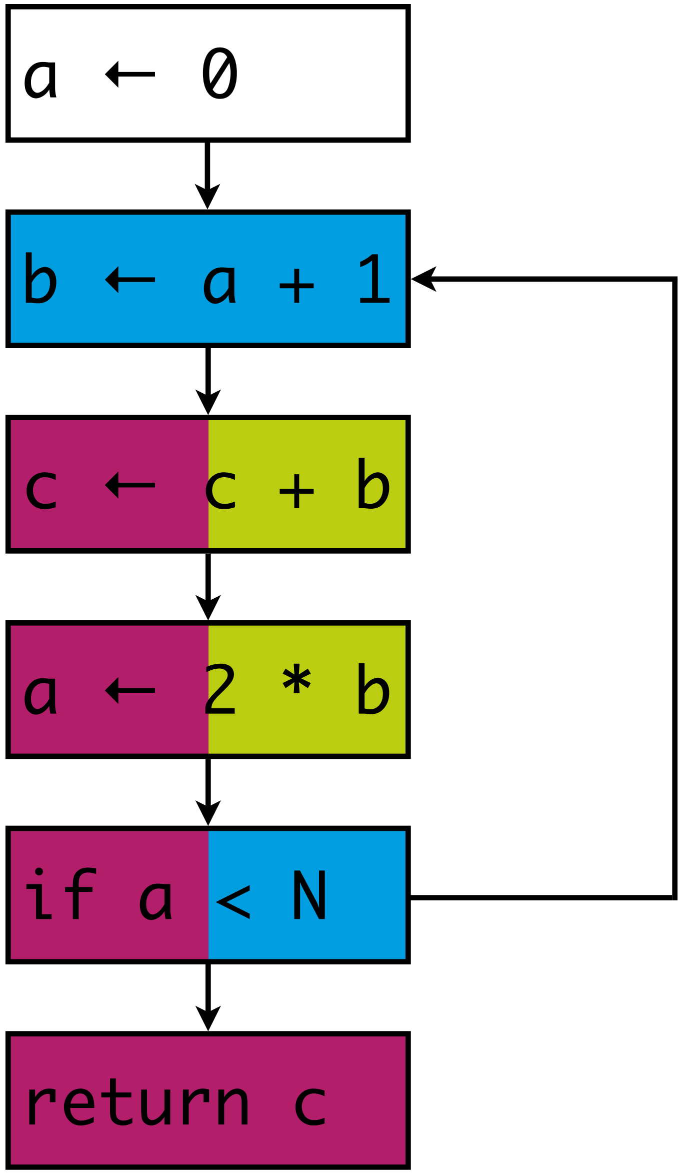
Liveness Analysis

example



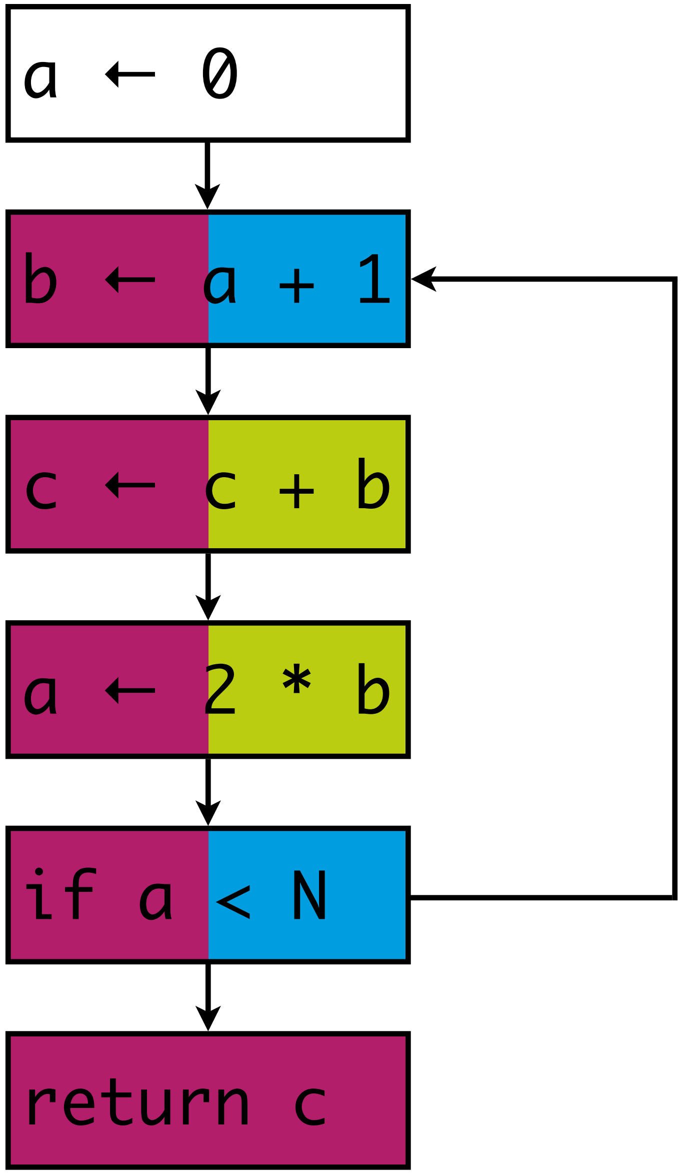
Liveness Analysis

example



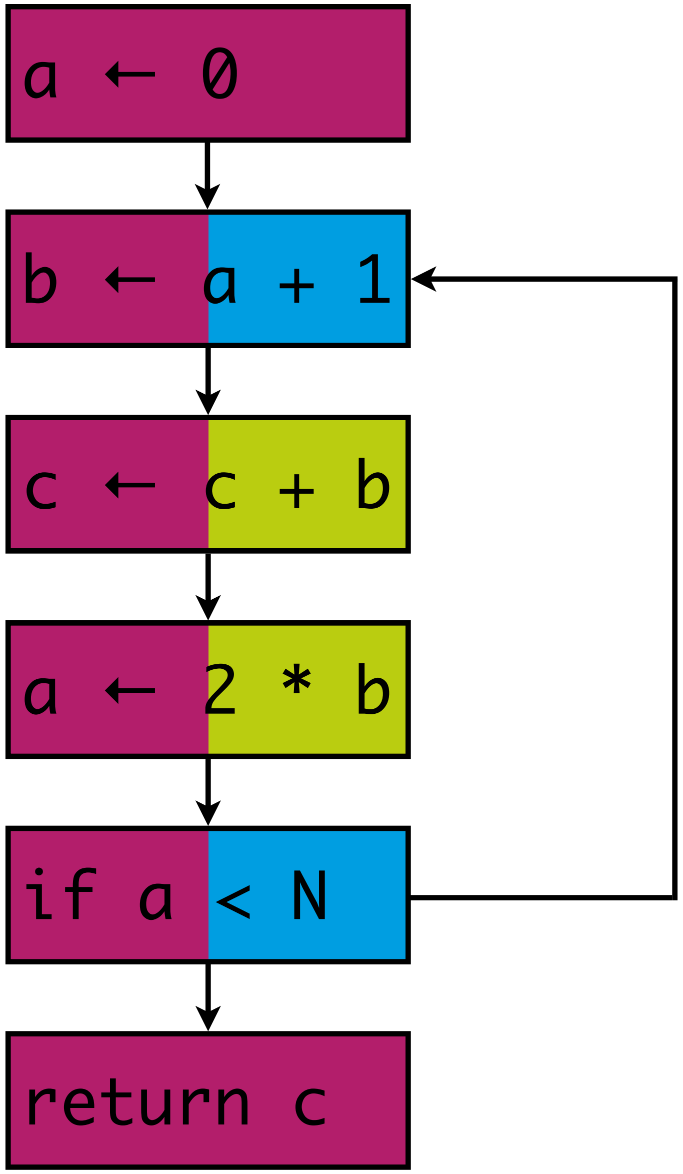
Liveness Analysis

example



Liveness Analysis

example



Liveness Analysis
definitions

Def and **use**

- an assignment to a variable **defines** a variable
- an occurrence of a variable in an expression **uses** that variable
- def[x] = {n | n defines x}, def[n] = {x | n defines x}
- use[x] = {n | n uses x}, use[n] = {x | n uses x}

A variable is **live on an edge** if

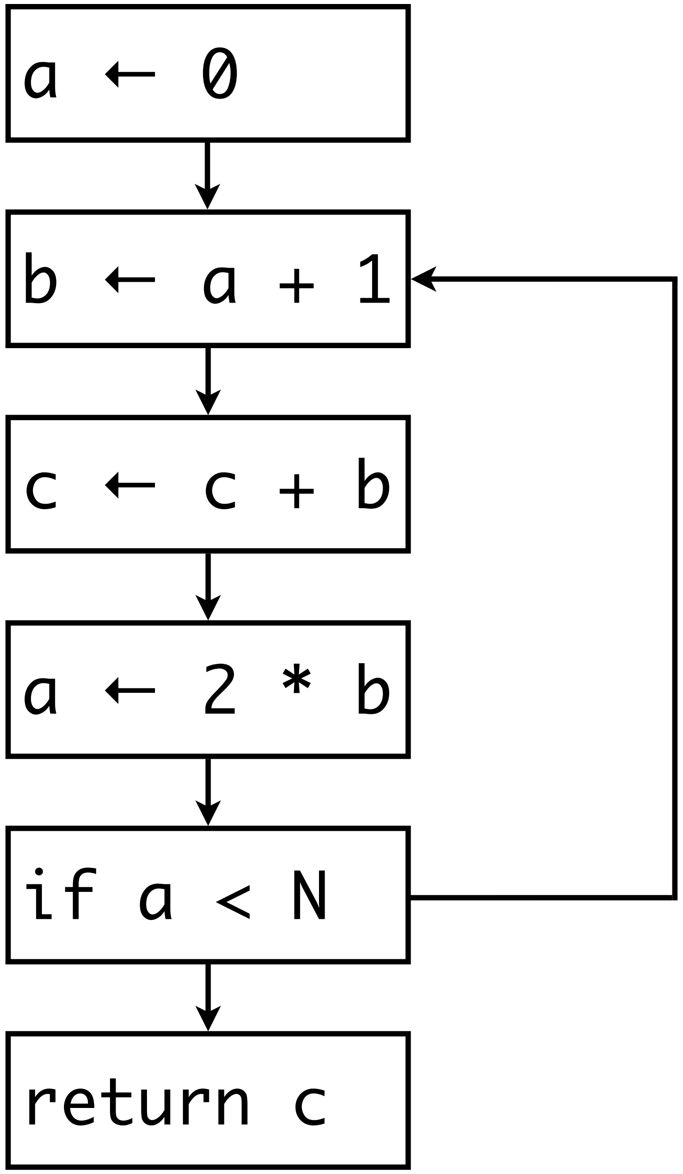
- there is a **directed path** from that edge **to** a **use** of the variable
- that does **not** go **through** any **def**

A variable is **live in/out at a node**

- **live-in**: it is live on any of the in-edges of that node
- **live-out**: it is live on any of the out-edges of the node

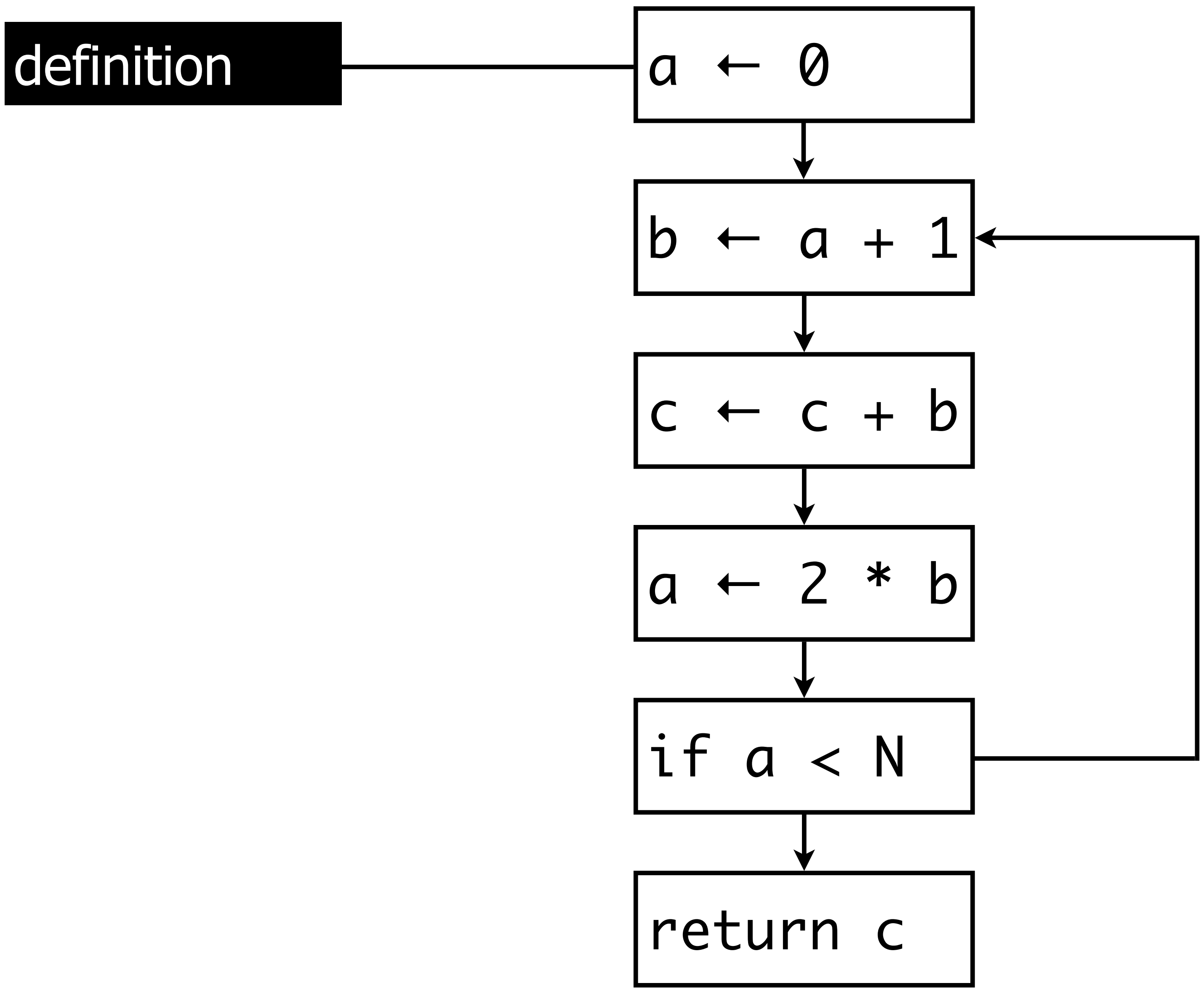
Liveness Analysis

terminology



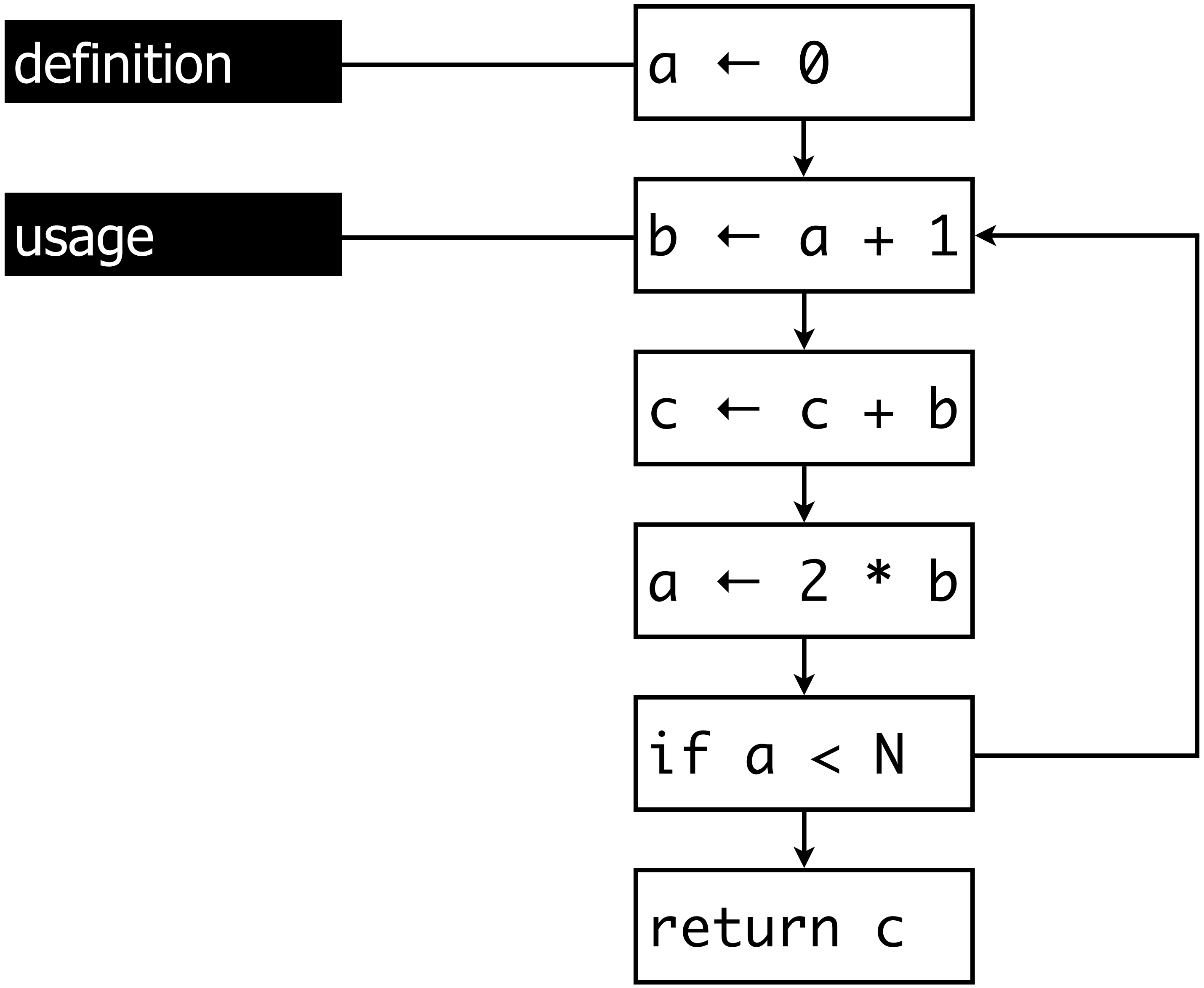
Liveness Analysis

terminology



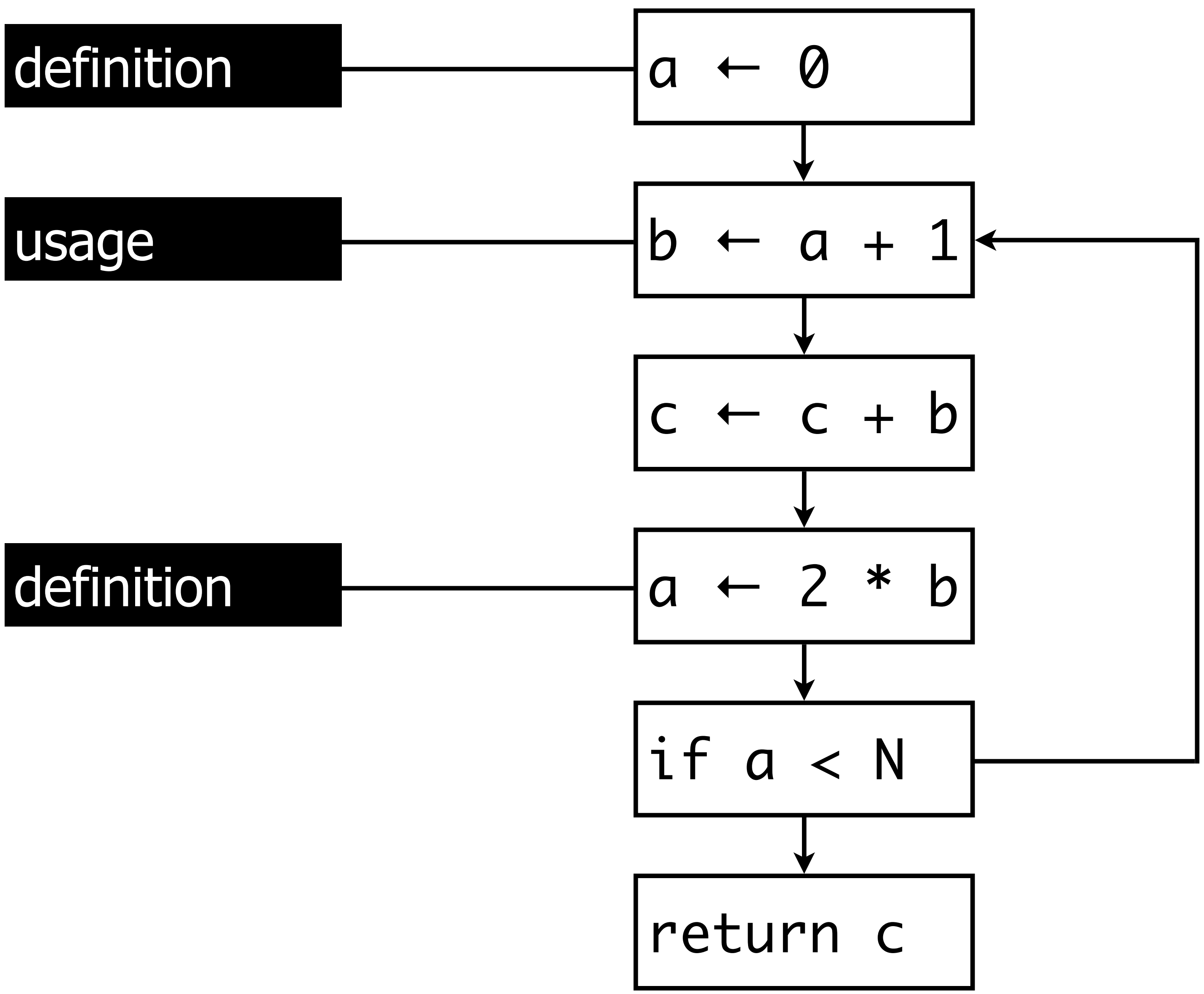
Liveness Analysis

terminology



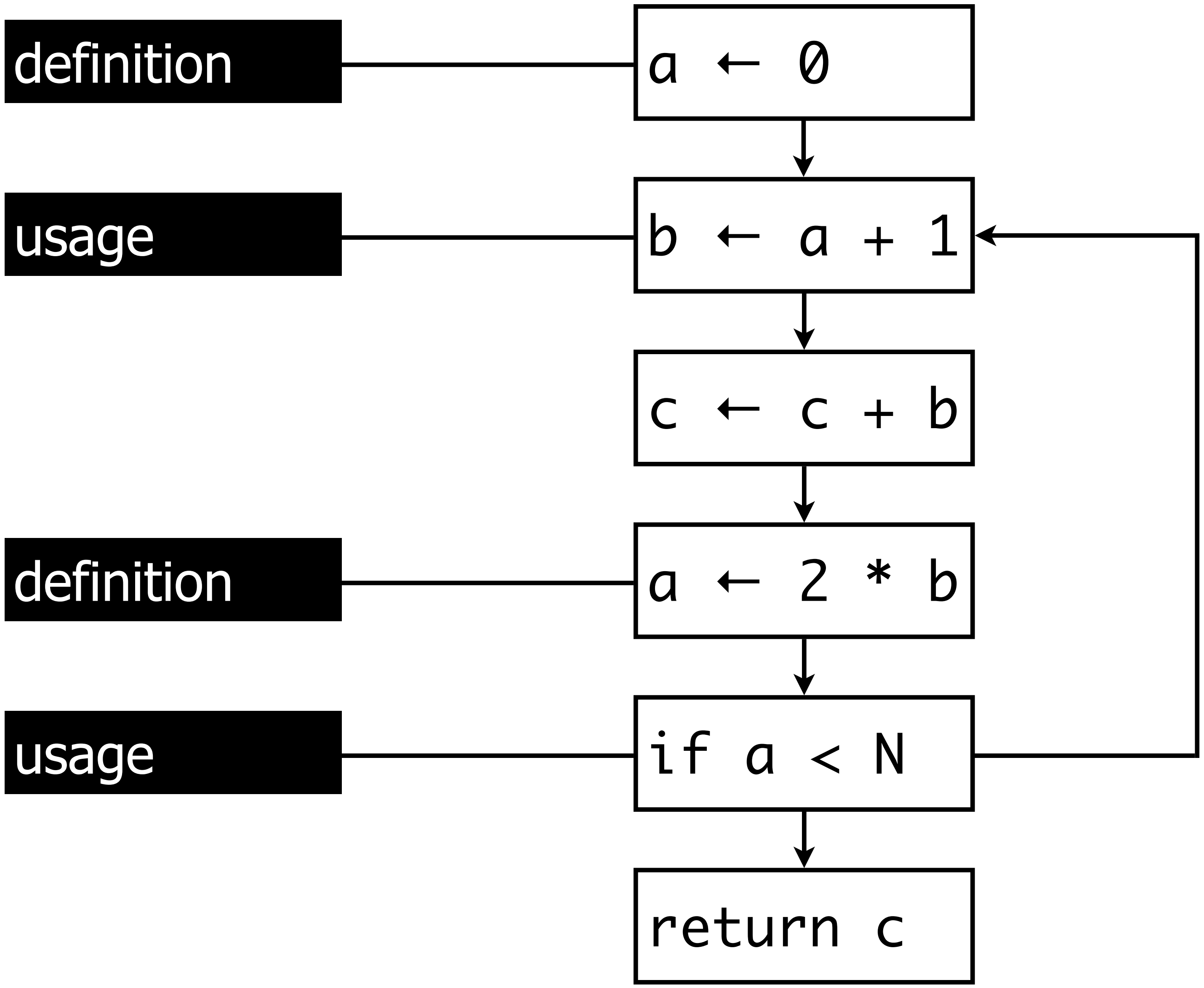
Liveness Analysis

terminology



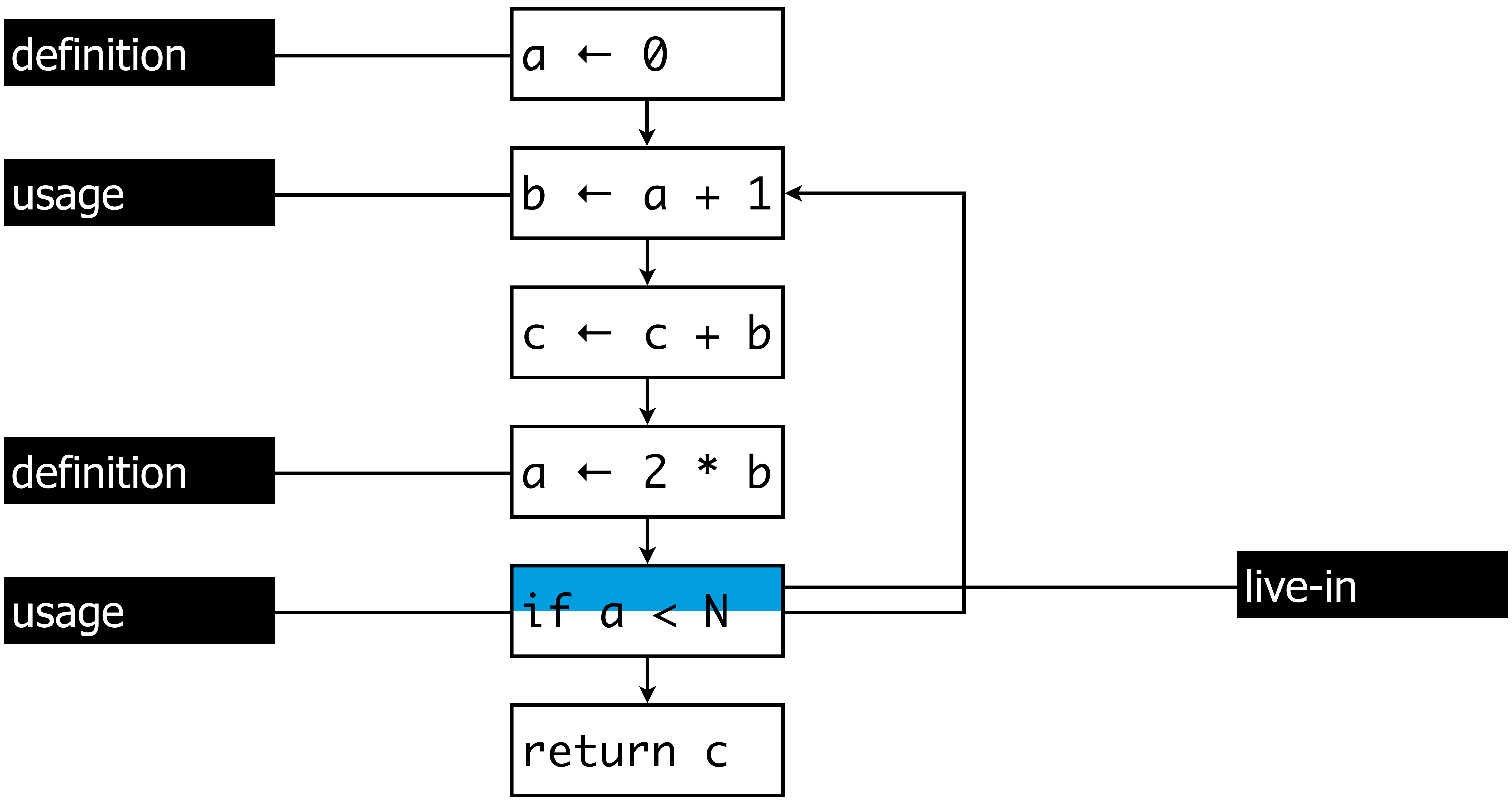
Liveness Analysis

terminology



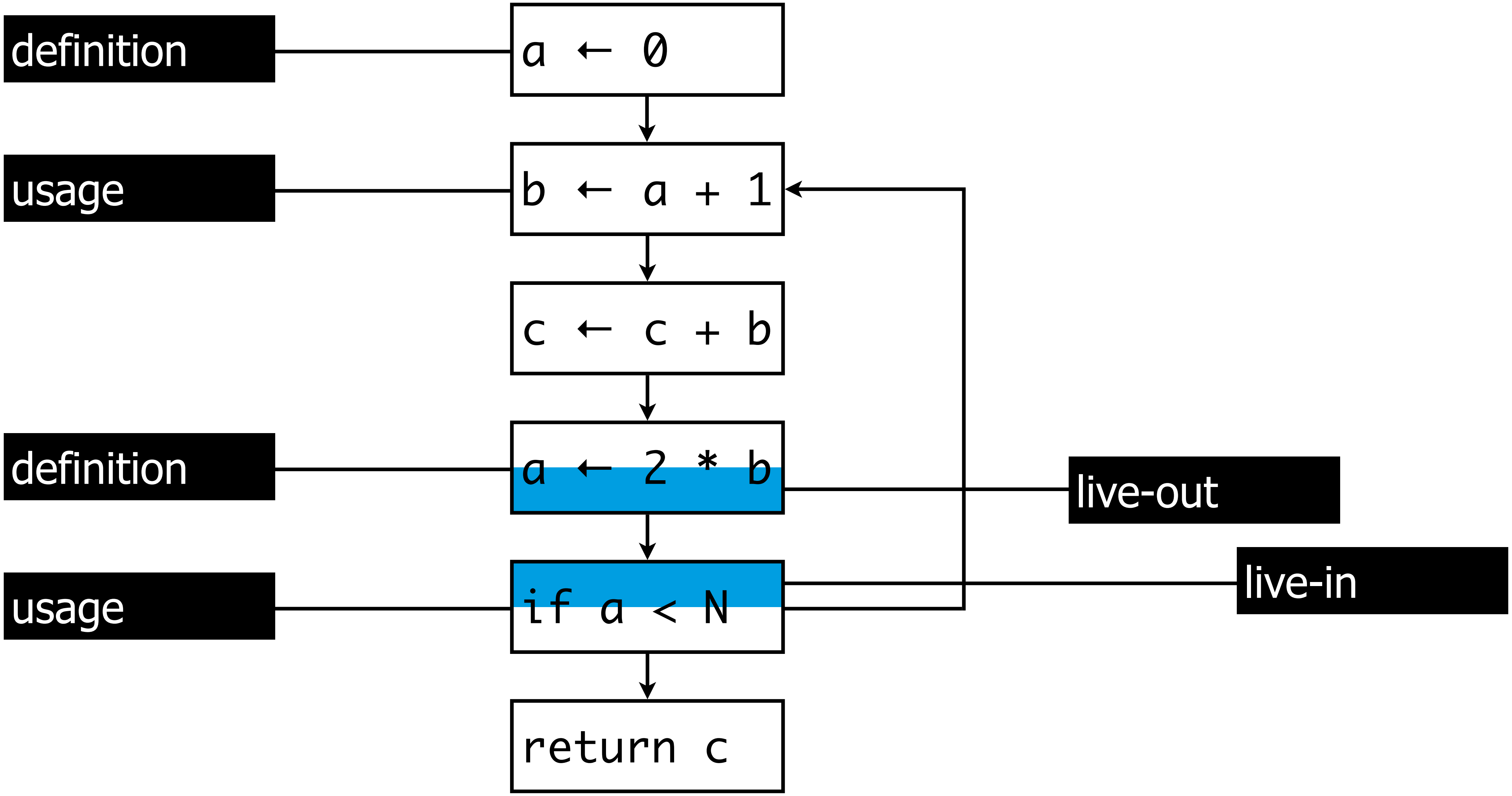
Liveness Analysis

terminology



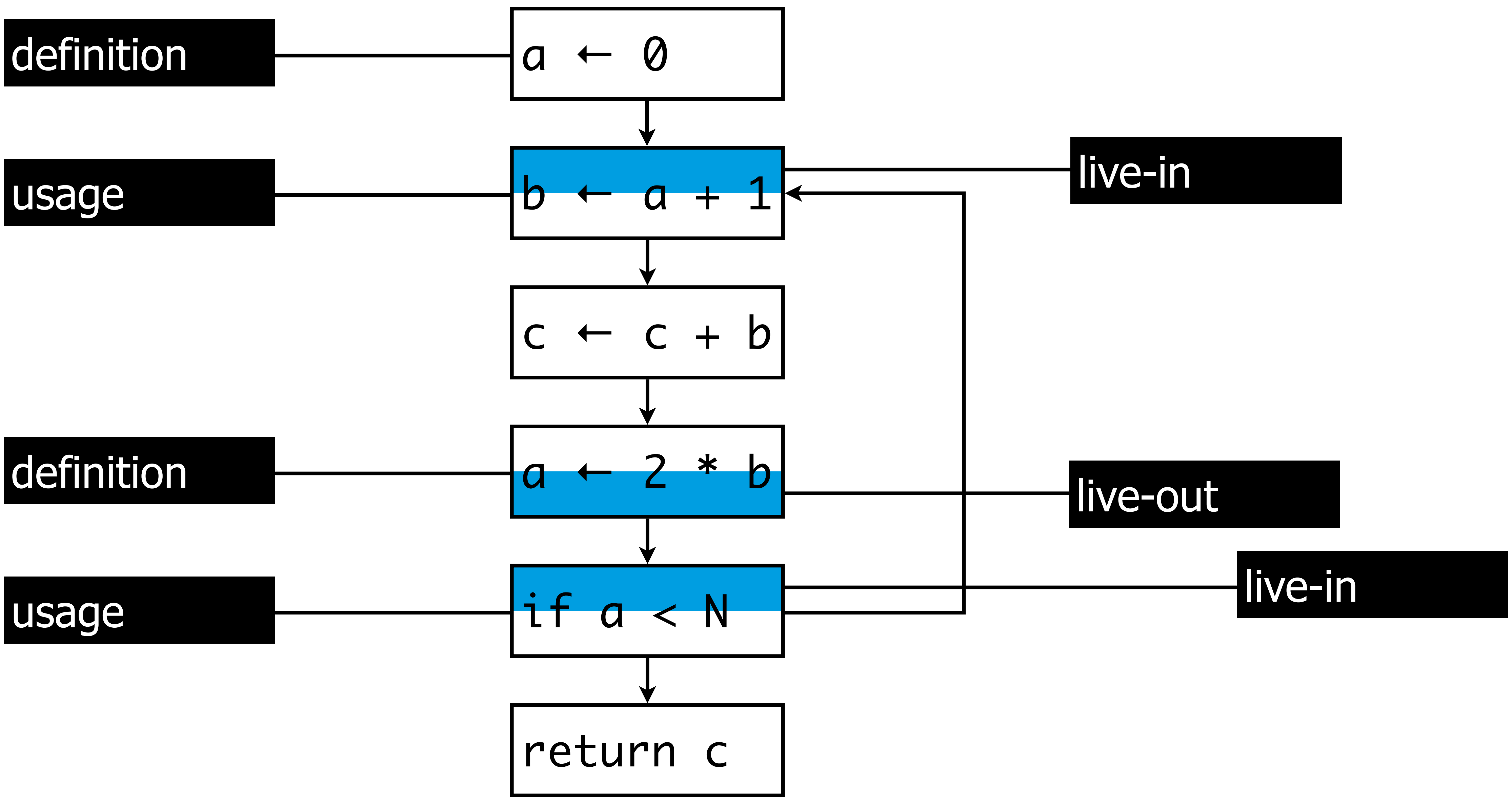
Liveness Analysis

terminology



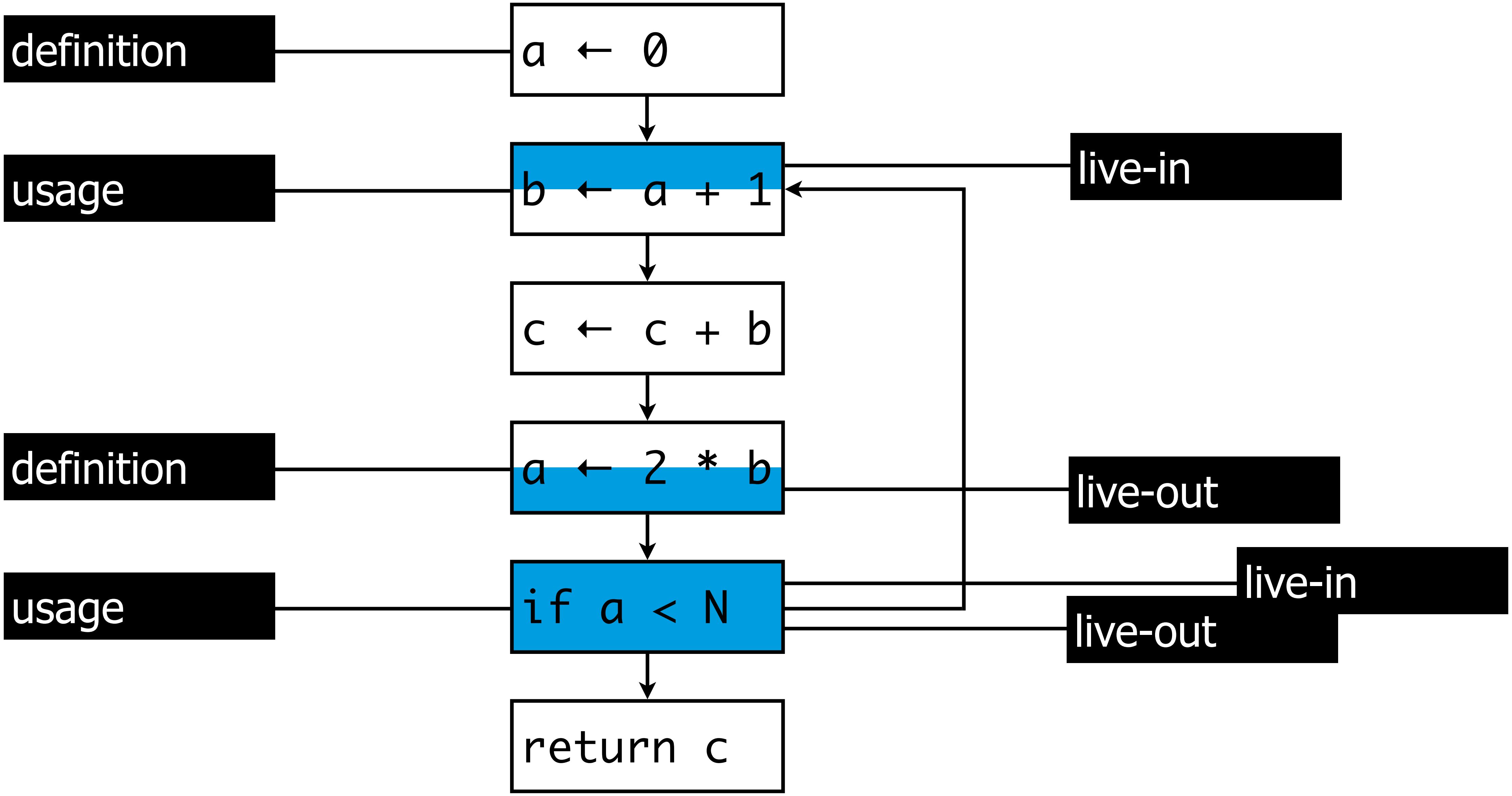
Liveness Analysis

terminology



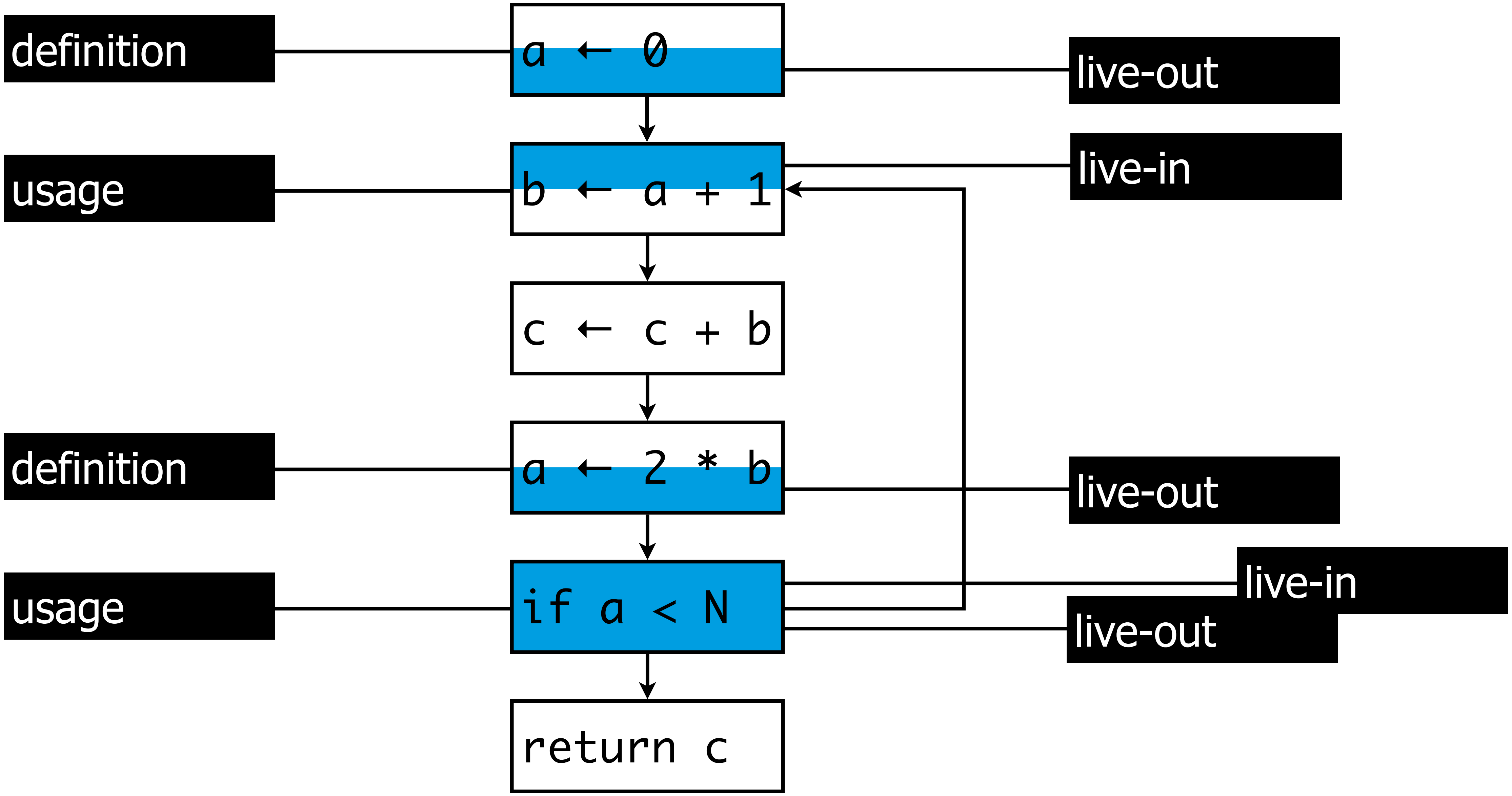
Liveness Analysis

terminology



Liveness Analysis

terminology



Liveness Analysis

formalization

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

$$out[n] = \bigcup_{s \in succ[n]} in[s]$$

1. If a variable is used at node n, then it is live-in at n

2. If a variable is live-out but not defined at node n, it is live-in at n

3. If a variable is live-in at node n, then it is live-out at its predecessors

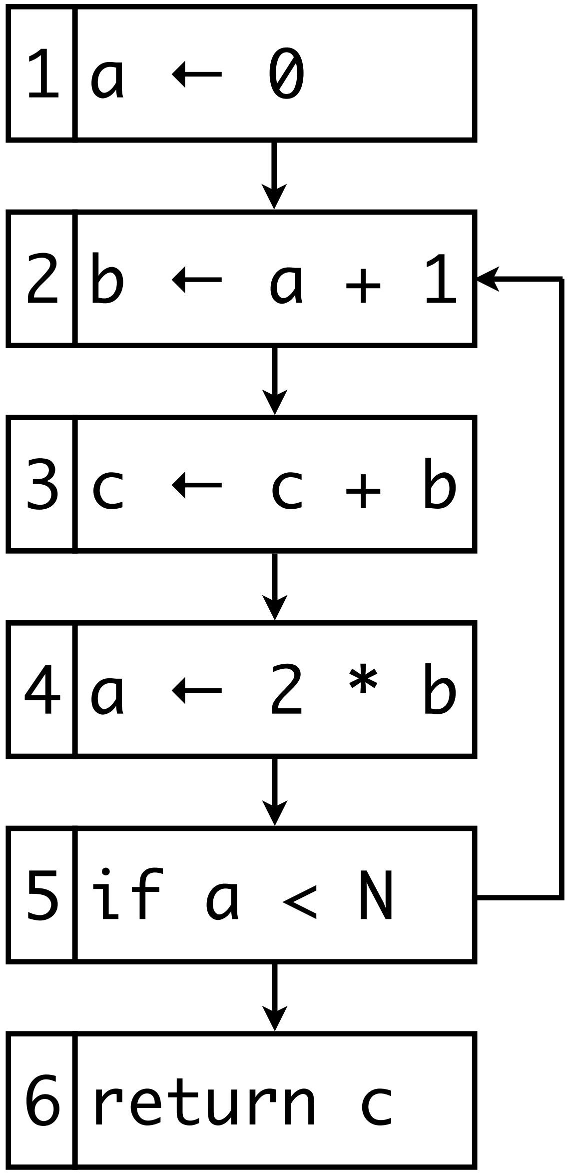
Liveness Analysis

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])
      for each s in succ[n]
         out[n] = out[n] ∪ in[s]
until
   for all n: in[n] = in'[n] and out[n] = out'[n]
```

Liveness Analysis

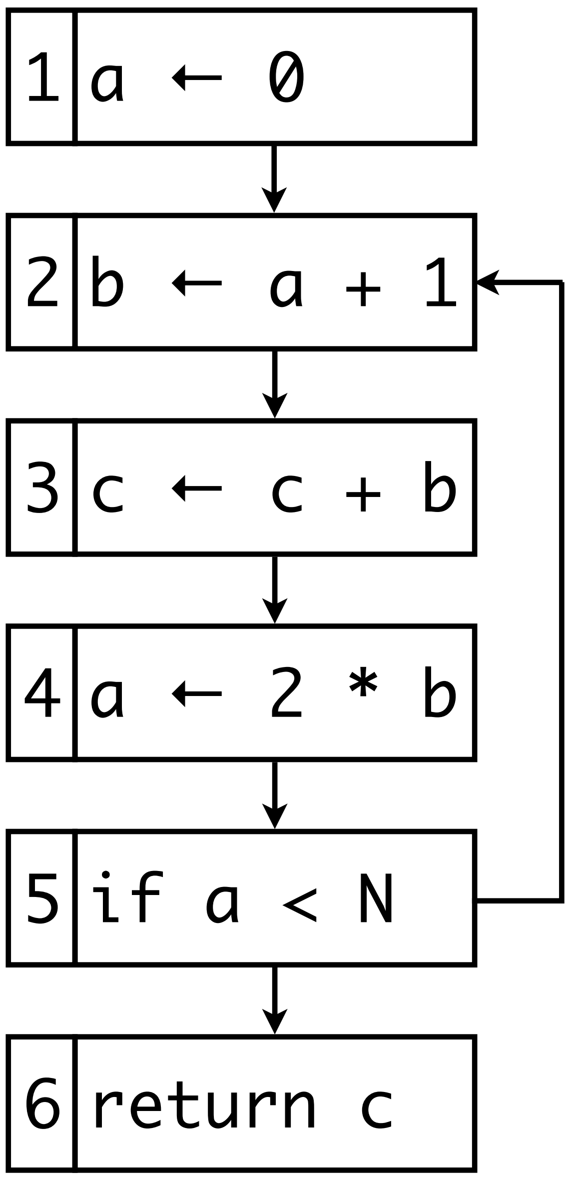
example



	u
1	
2	a
3	bc
4	b
5	a
6	c

Liveness Analysis

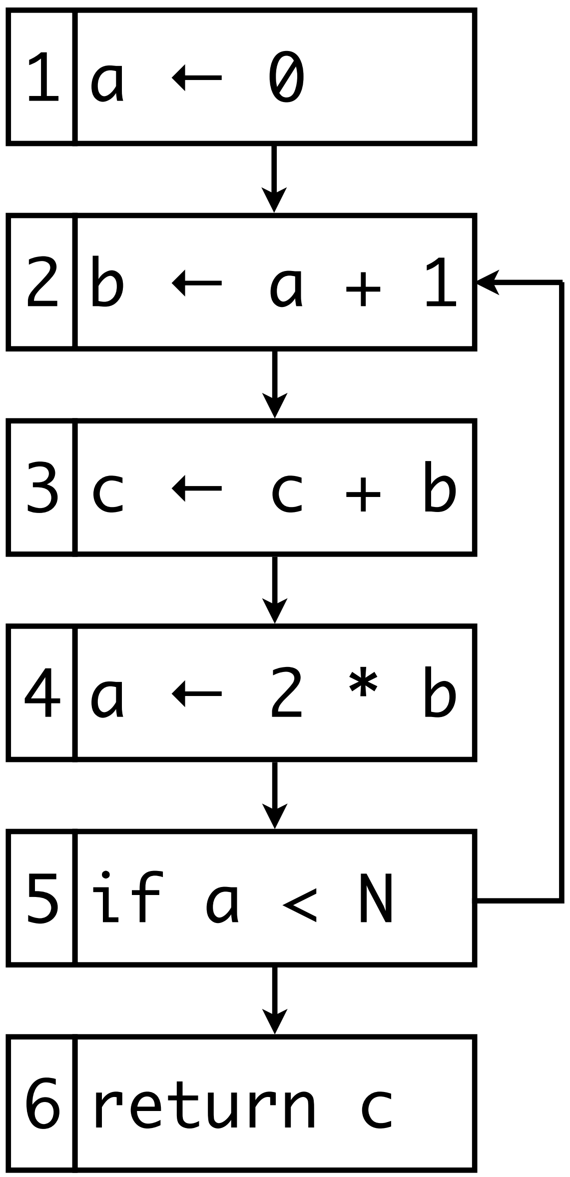
example



	u	d														
1		a														
2	a	b														
3	bc	c														
4	b	a														
5	a															
6	c															

Liveness Analysis

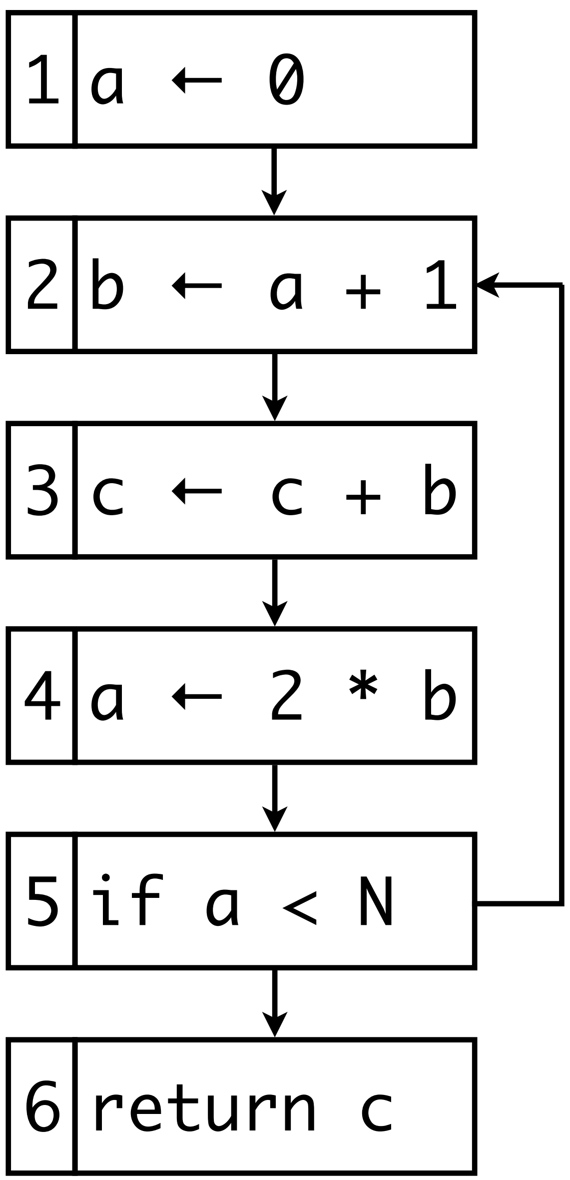
example



			1	
	u	d	i	o
1		a		
2	a	b	a	
3	bc	c	bc	
4	b	a	b	
5	a		a	a
6	c		c	

Liveness Analysis

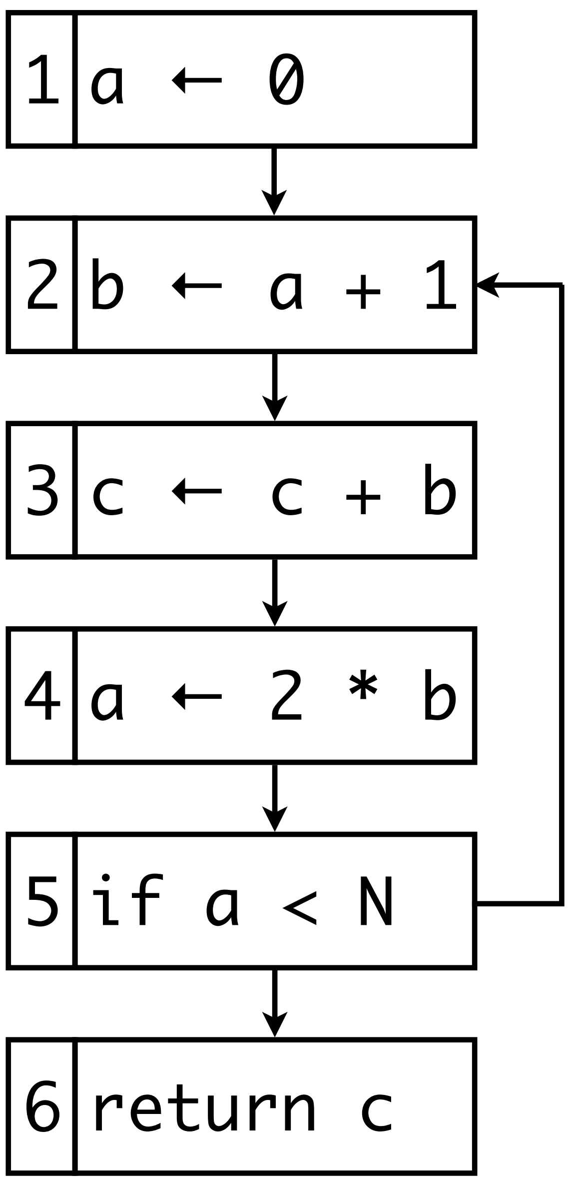
example



			1		2	
	u	d	i	o	i	o
1		a				a
2	a	b	a		a	bc
3	bc	c	bc		bc	b
4	b	a	b		b	a
5	a		a	a	a	ac
6	c		c		c	

Liveness Analysis

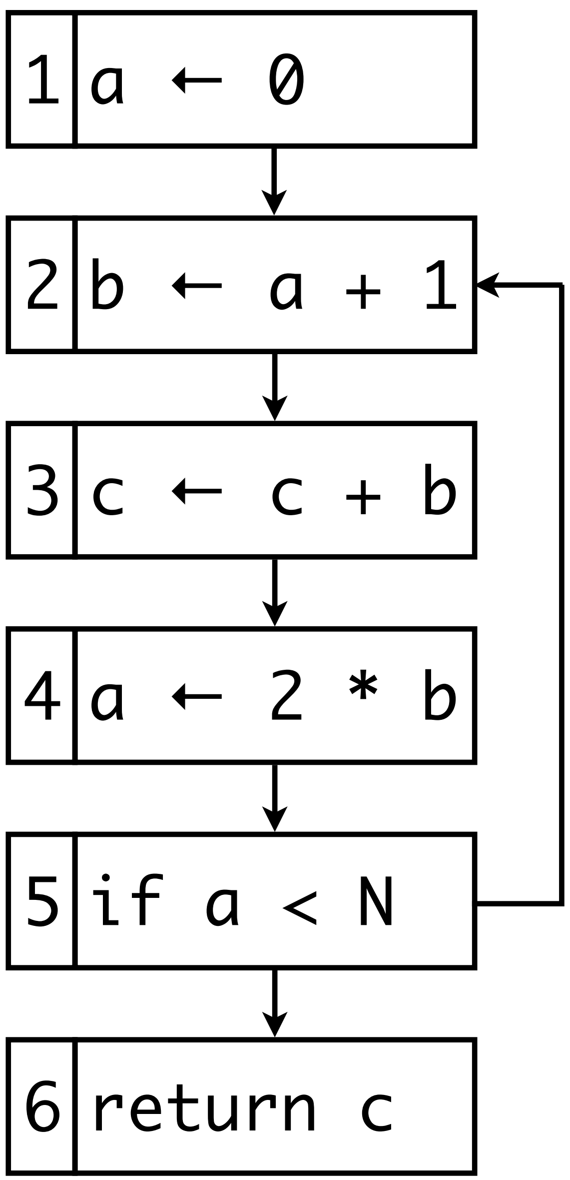
example



			1		2		3	
	u	d	i	o	i	o	i	o
1		a				a		a
2	a	b	a		a	bc	ac	bc
3	bc	c	bc		bc	b	bc	b
4	b	a	b		b	a	b	a
5	a		a	a	a	ac	ac	ac
6	c		c		c		c	

Liveness Analysis

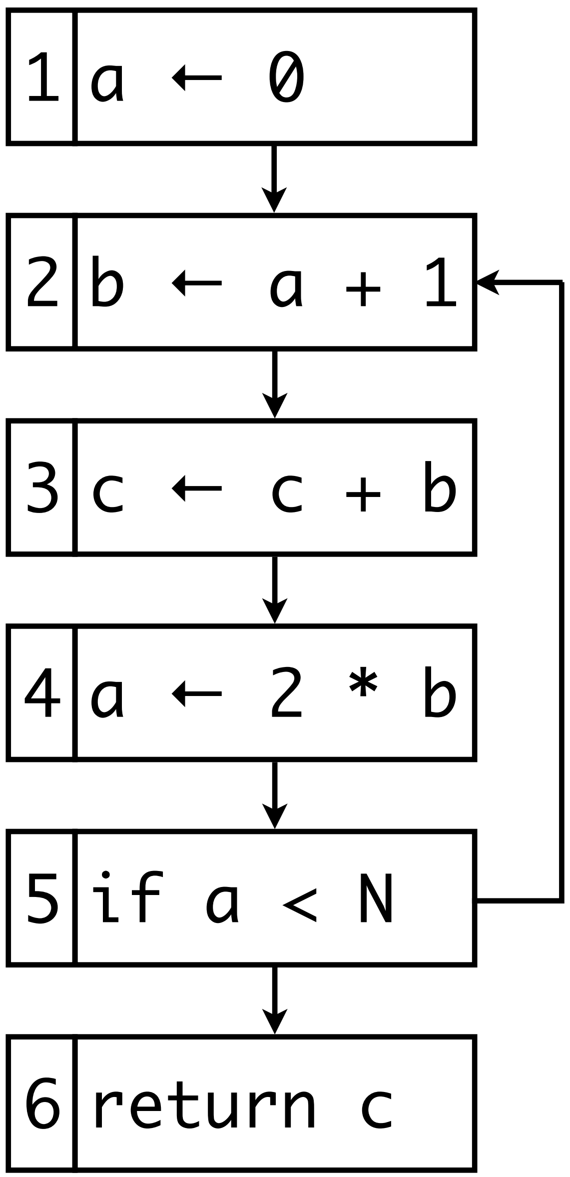
example



			1		2		3		4							
	u	d	i	o	i	o	i	o	i	o						
1		a				a		a		ac						
2	a	b	a		a	bc	ac	bc	ac	bc						
3	bc	c	bc		bc	b	bc	b	bc	b						
4	b	a	b		b	a	b	a	b	ac						
5	a		a	a	a	ac	ac	ac	ac	ac						
6	c		c		c		c		c							

Liveness Analysis

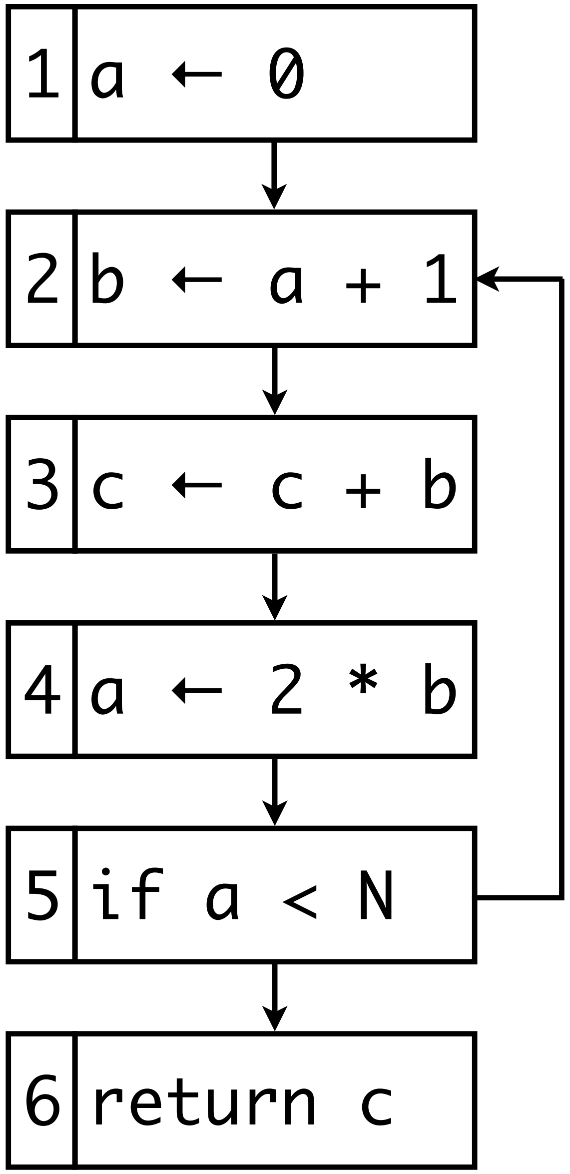
example



			1		2		3		4		5					
	u	d	i	o	i	o	i	o	i	o	i	o				
1		a				a		a		ac	c	ac				
2	a	b	a		a	bc	ac	bc	ac	bc	ac	bc				
3	bc	c	bc		bc	b	bc	b	bc	b	bc	b				
4	b	a	b		b	a	b	a	b	ac	bc	ac				
5	a		a	a	a	ac	ac	ac	ac	ac	ac	ac				
6	c		c		c		c		c		c					

Liveness Analysis

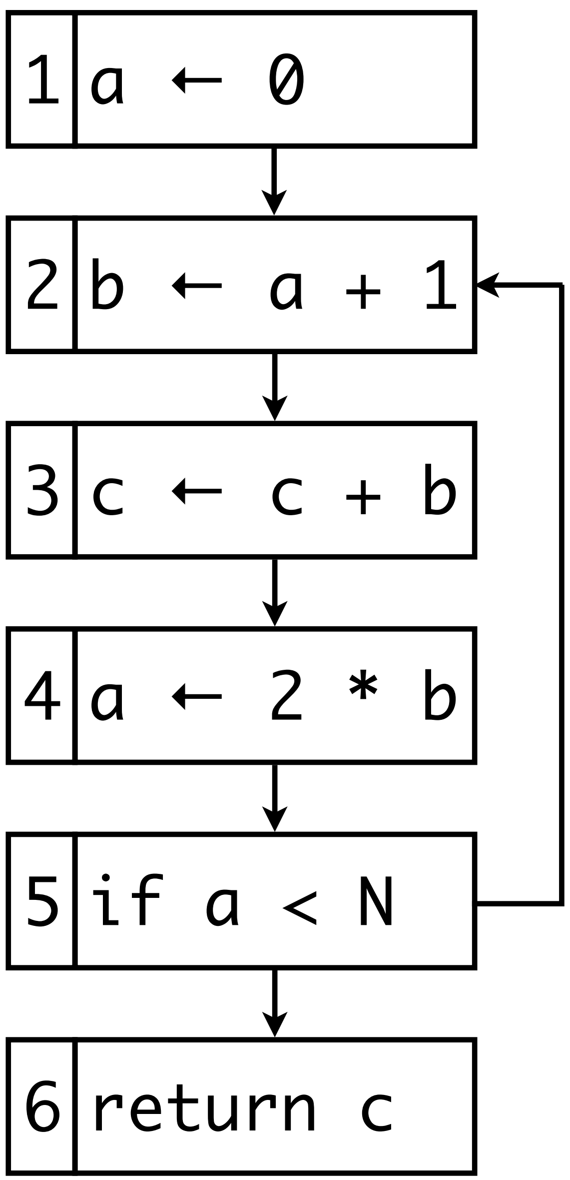
example



			1		2		3		4		5		6	
	u	d	i	o	i	o	i	o	i	o	i	o	i	o
1		a				a		a		ac	c	ac	c	ac
2	a	b	a		a	bc	ac	bc	ac	bc	ac	bc	ac	bc
3	bc	c	bc		bc	b	bc	b	bc	b	bc	b	bc	bc
4	b	a	b		b	a	b	a	b	ac	bc	ac	bc	ac
5	a		a	a	a	ac	ac	ac	ac	ac	ac	ac	ac	ac
6	c		c		c		c		c		c		c	

Liveness Analysis

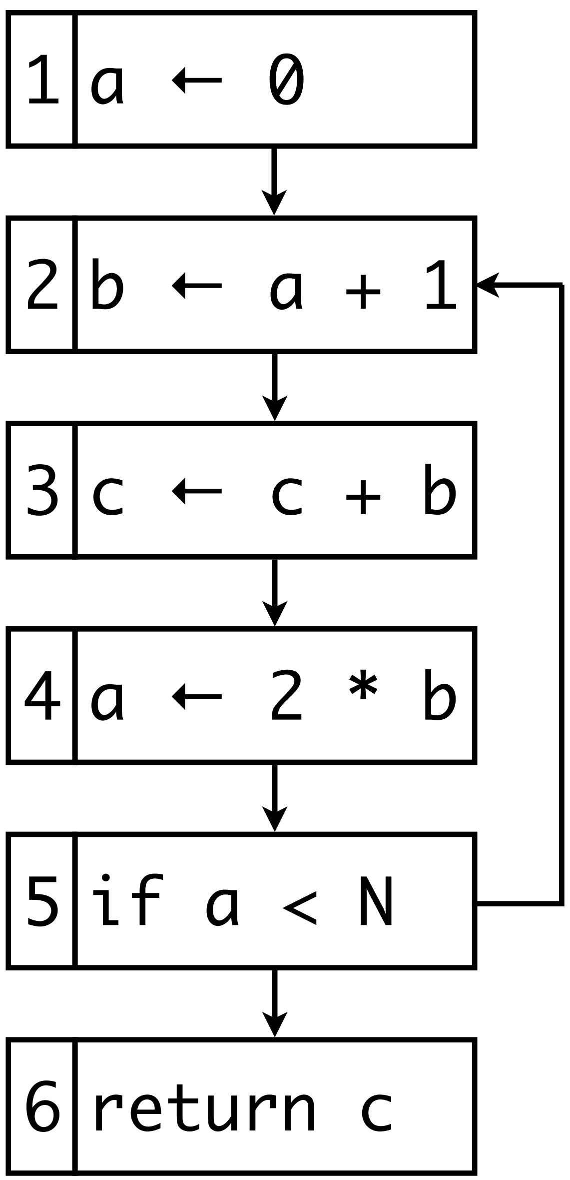
example



			1		2		3		4		5		6		7	
	u	d	i	o	i	o	i	o	i	o	i	o	i	o	i	o
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	

Liveness Analysis

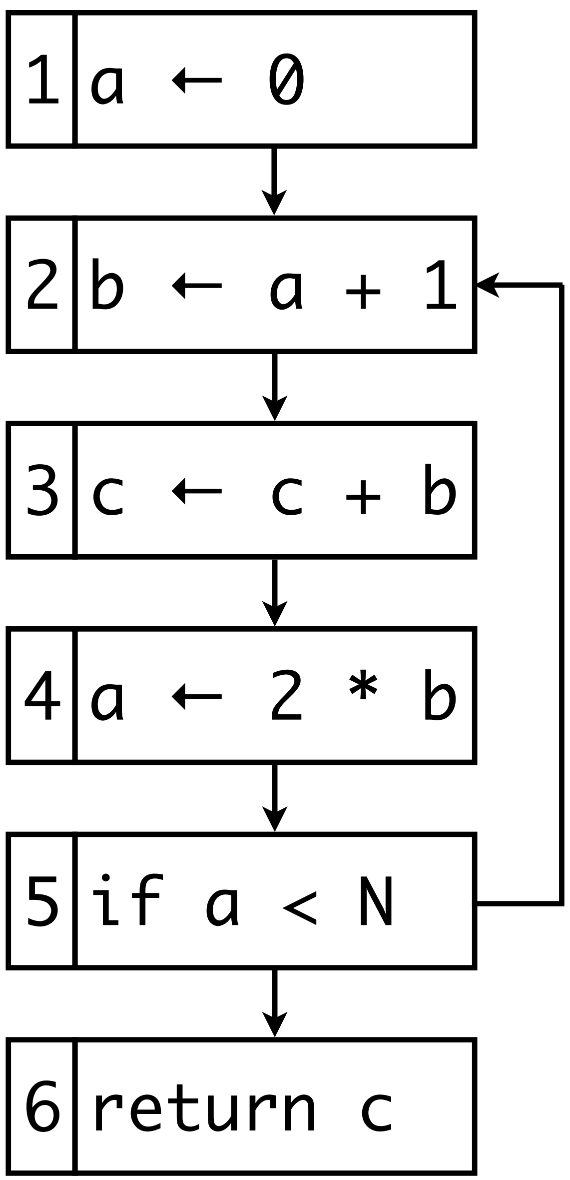
optimization



	u							
6	c							
5	a							
4	b							
3	bc							
2	a							
1								

Liveness Analysis

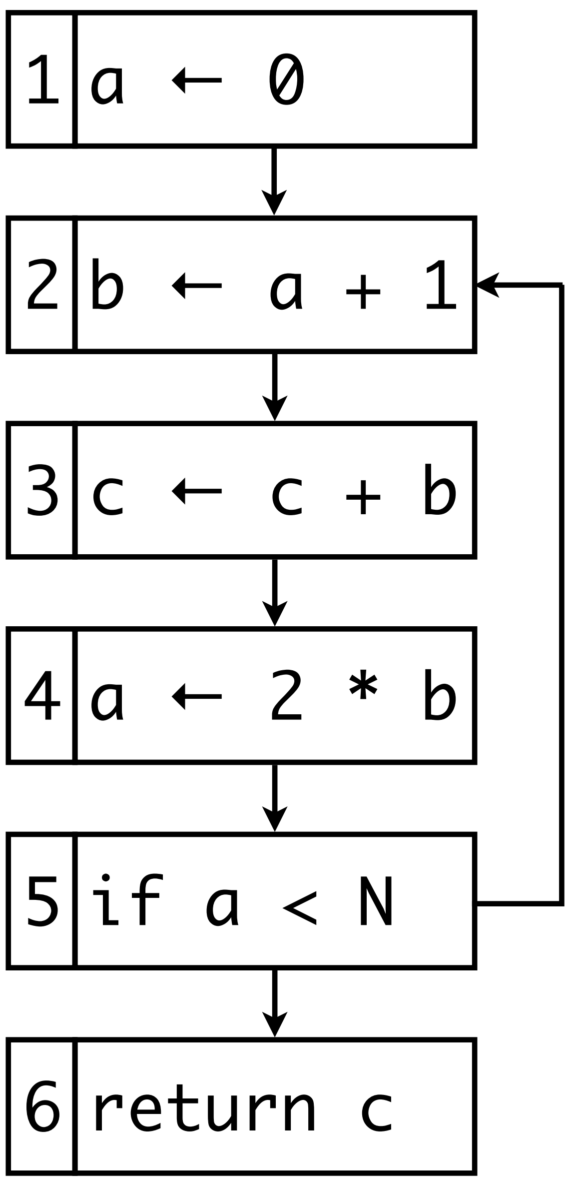
optimization



	u	d						
6	c							
5	a							
4	b	a						
3	bc	c						
2	a	b						
1		a						

Liveness Analysis

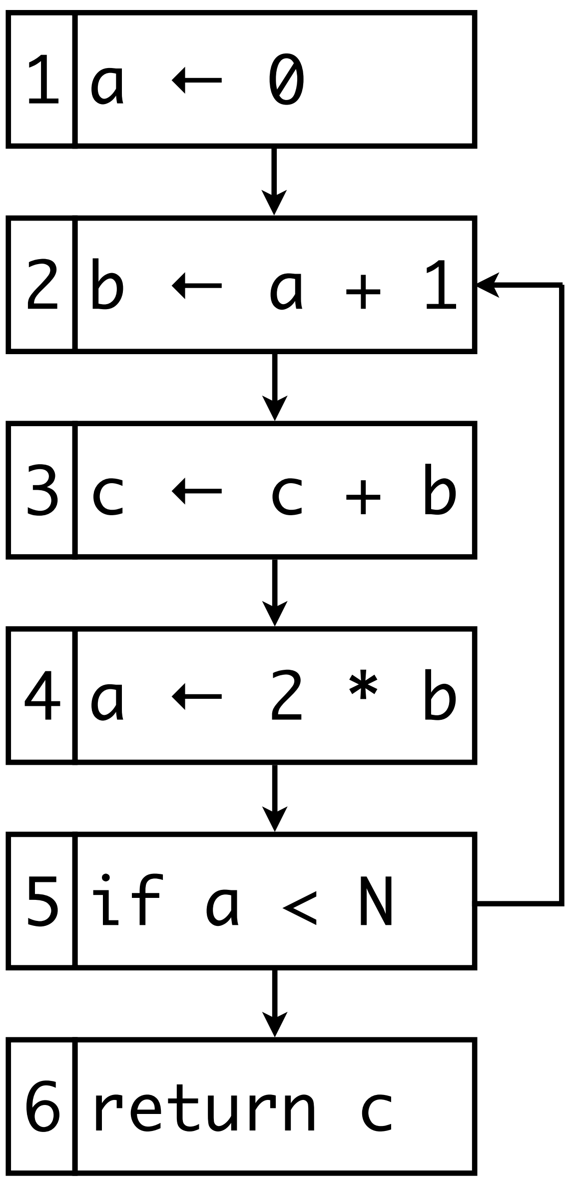
optimization



	1							
	u	d	o	i				
6	c			c				
5	a		c	ac				
4	b	a	ac	bc				
3	bc	c	bc	bc				
2	a	b	bc	ac				
1		a	ac	a				

Liveness Analysis

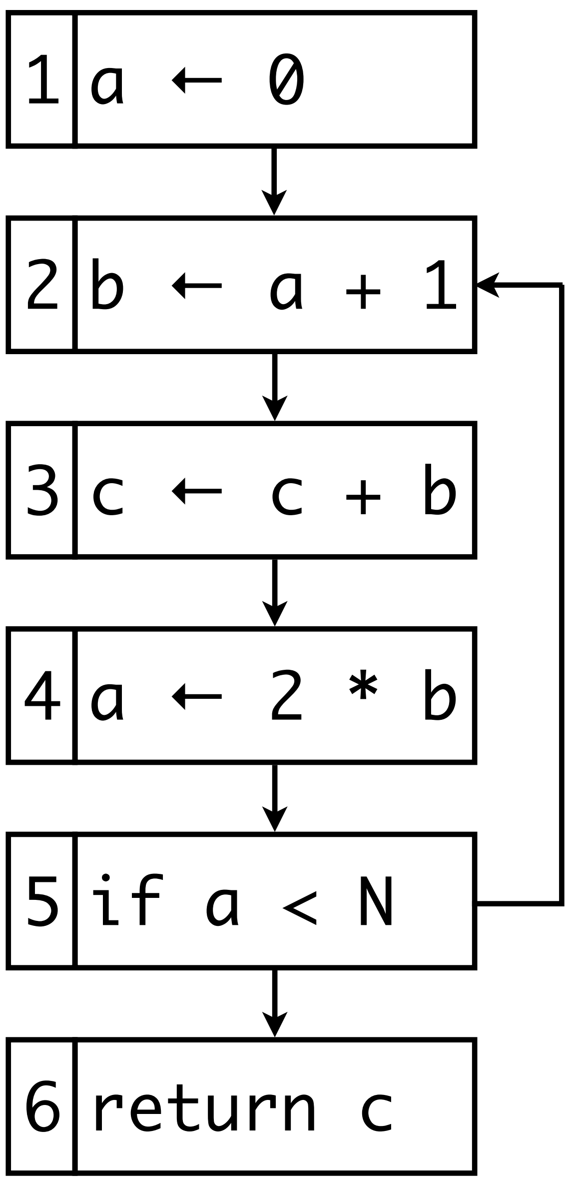
optimization



			1		2	
	u	d	o	i	o	i
6	c			c		c
5	a		c	ac	ac	ac
4	b	a	ac	bc	ac	bc
3	bc	c	bc	bc	bc	bc
2	a	b	bc	ac	bc	ac
1		a	ac	a	ac	c

Liveness Analysis

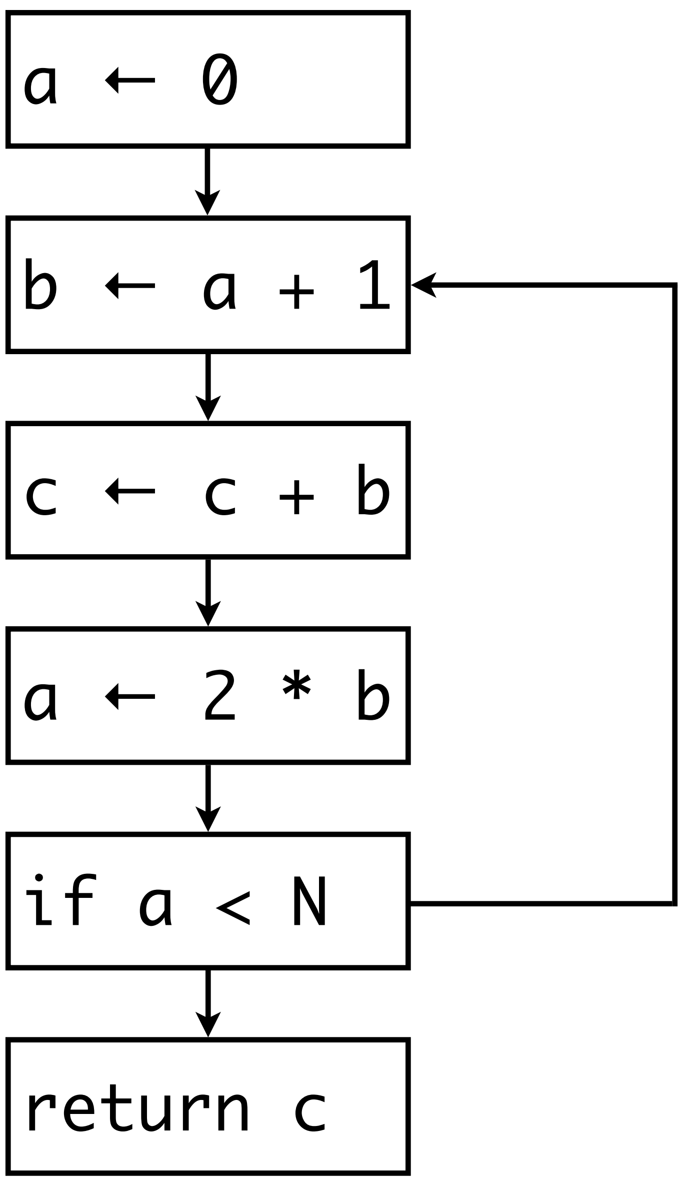
optimization



			1		2		3	
	u	d	o	i	o	i	o	i
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	a	ac	c	ac	c

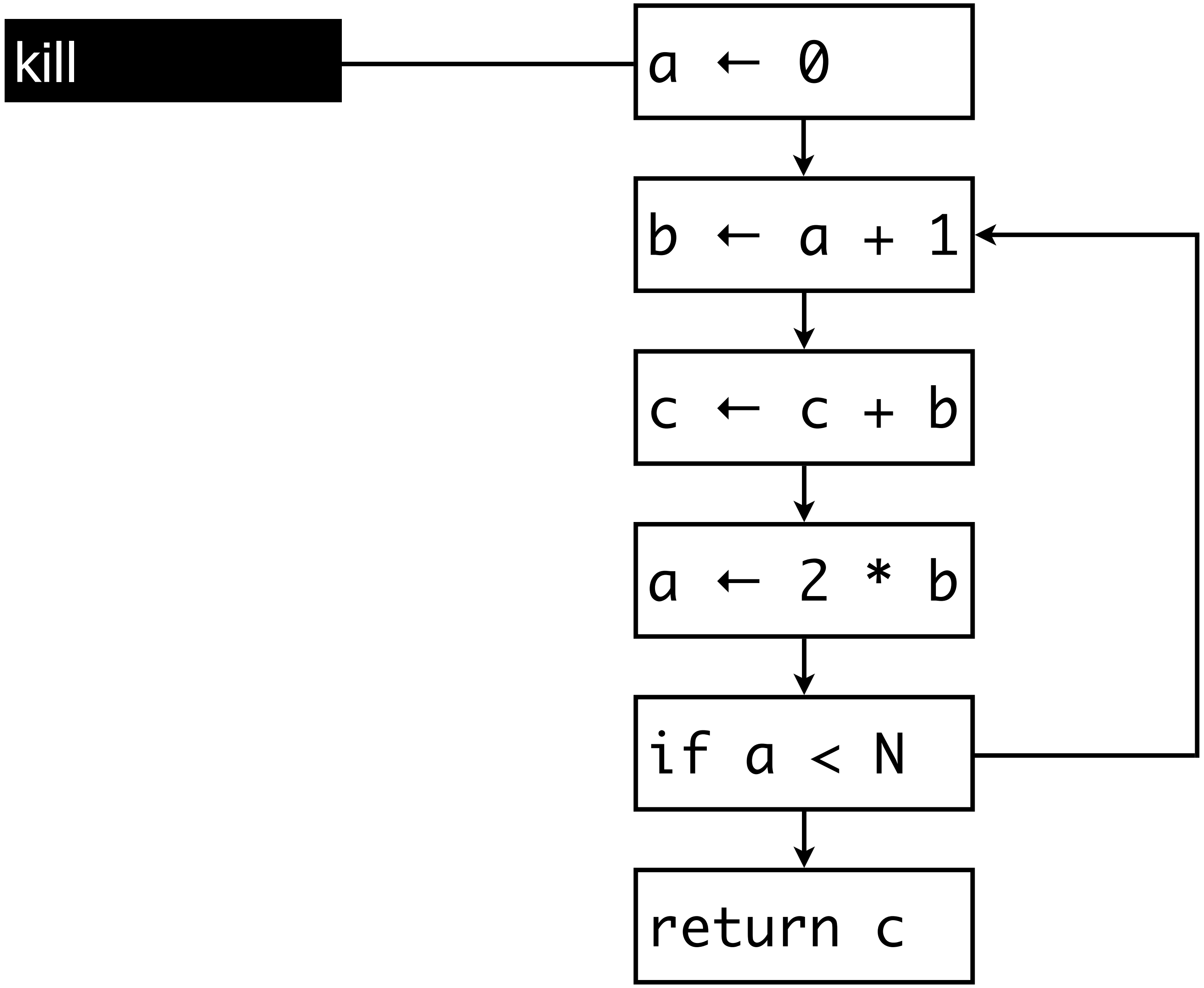
Liveness Analysis

generalisation



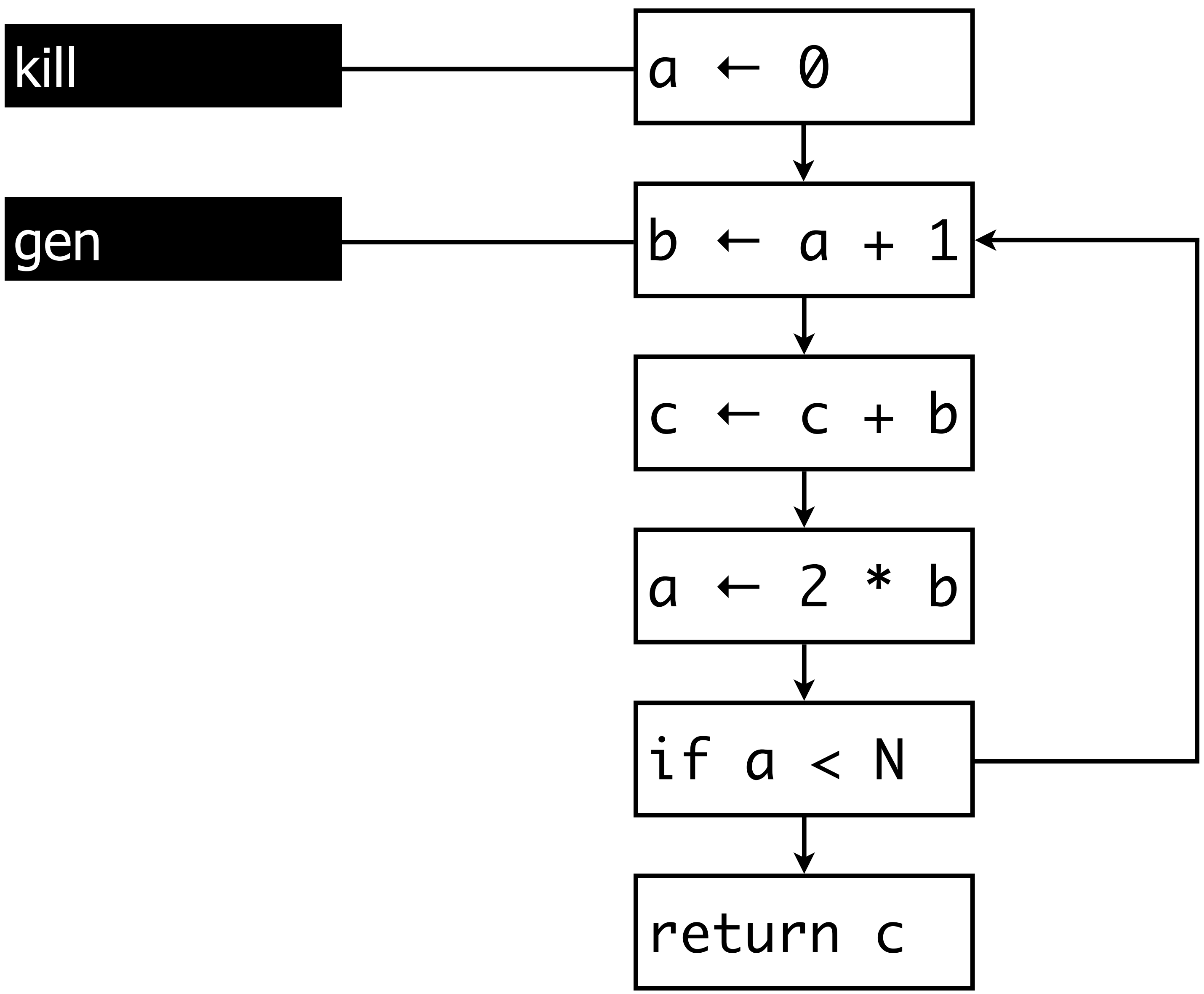
Liveness Analysis

generalisation



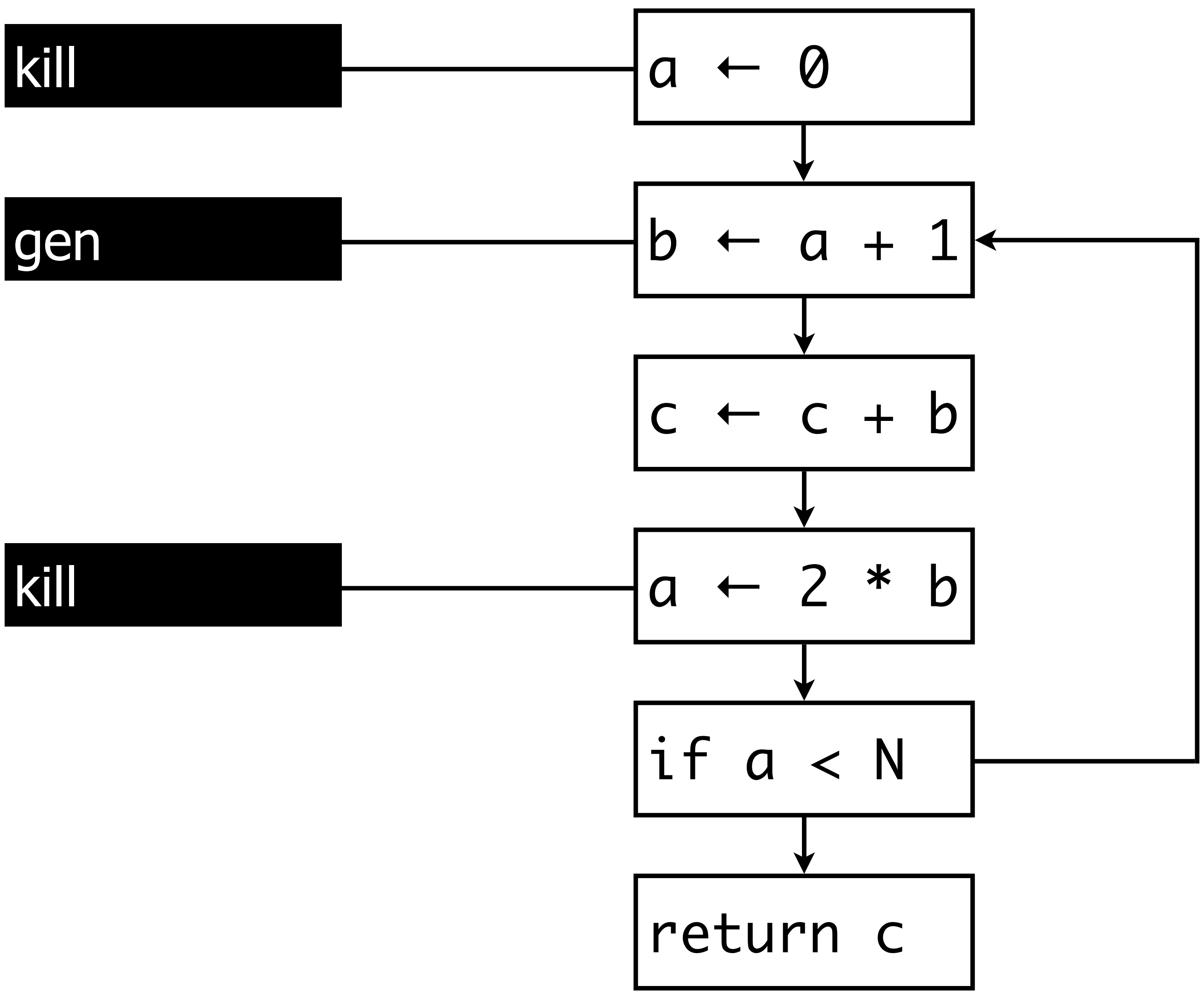
Liveness Analysis

generalisation



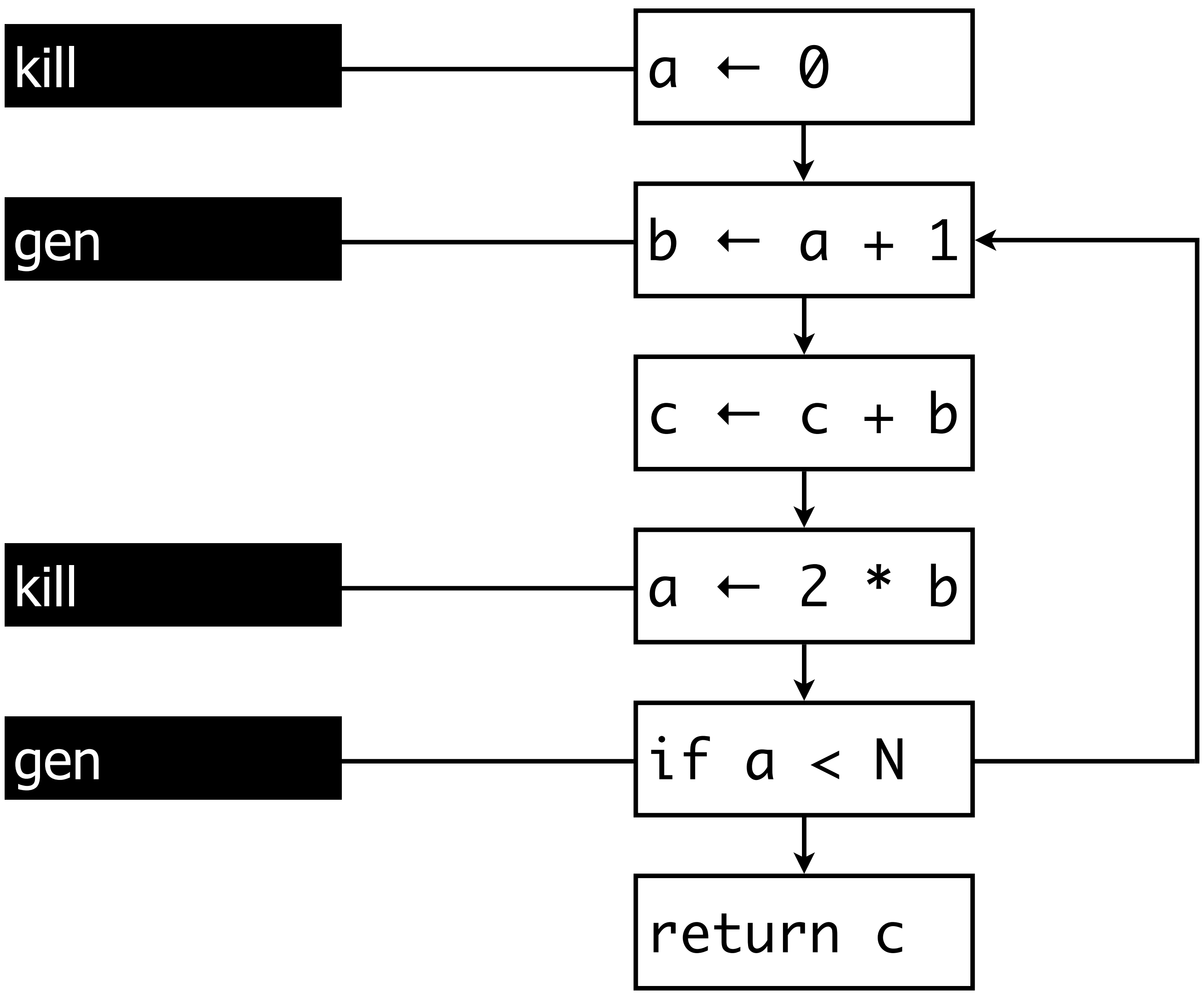
Liveness Analysis

generalisation



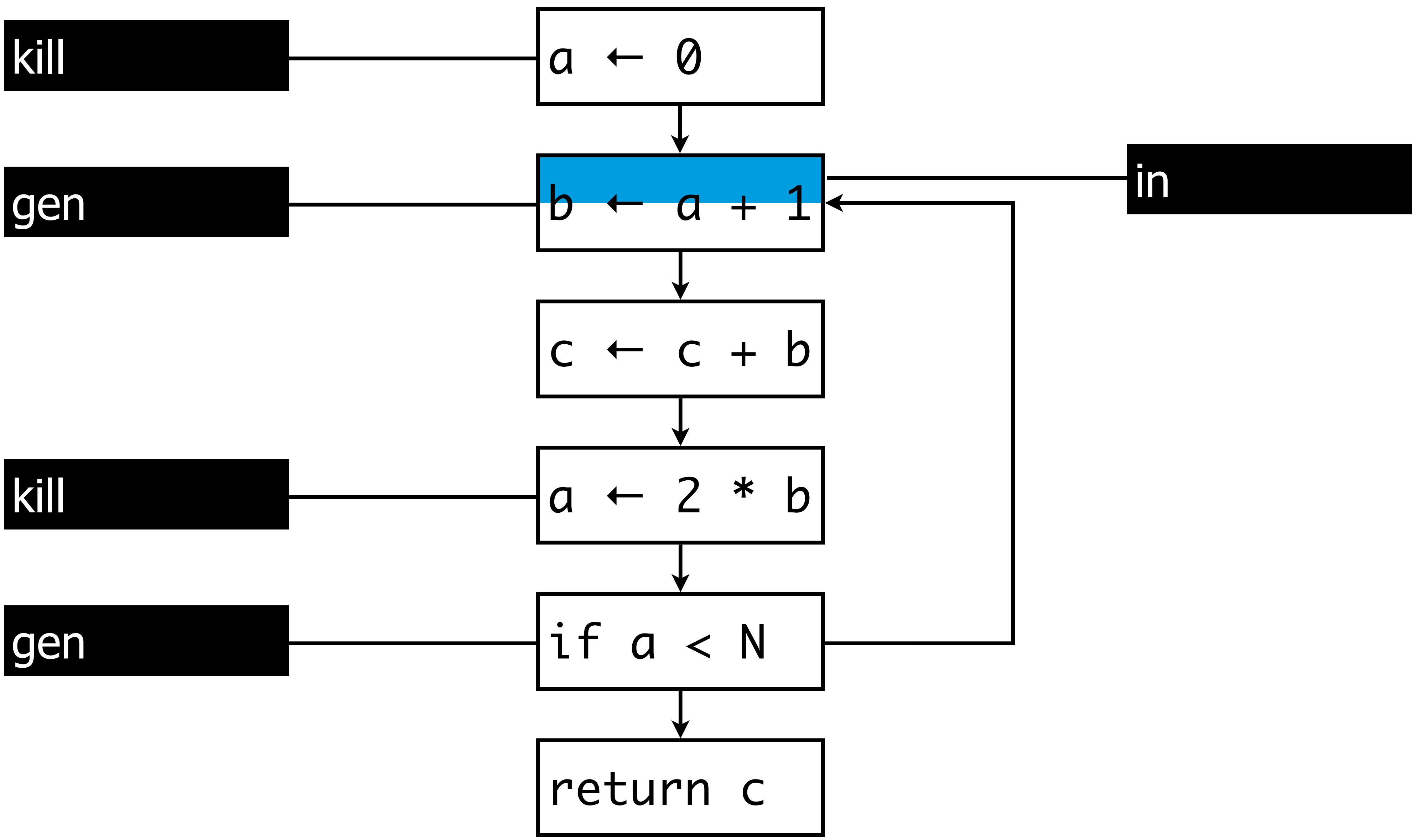
Liveness Analysis

generalisation



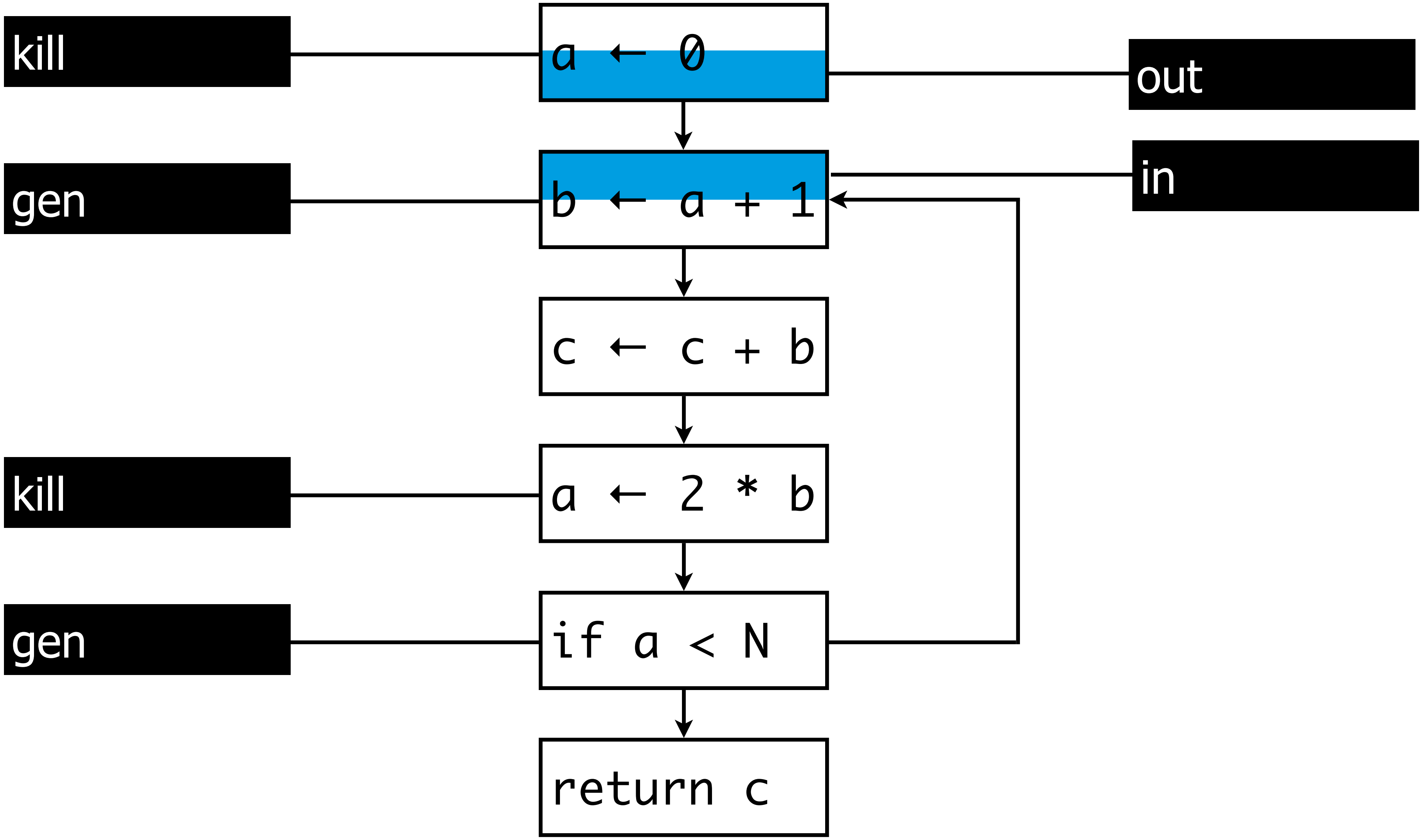
Liveness Analysis

generalisation



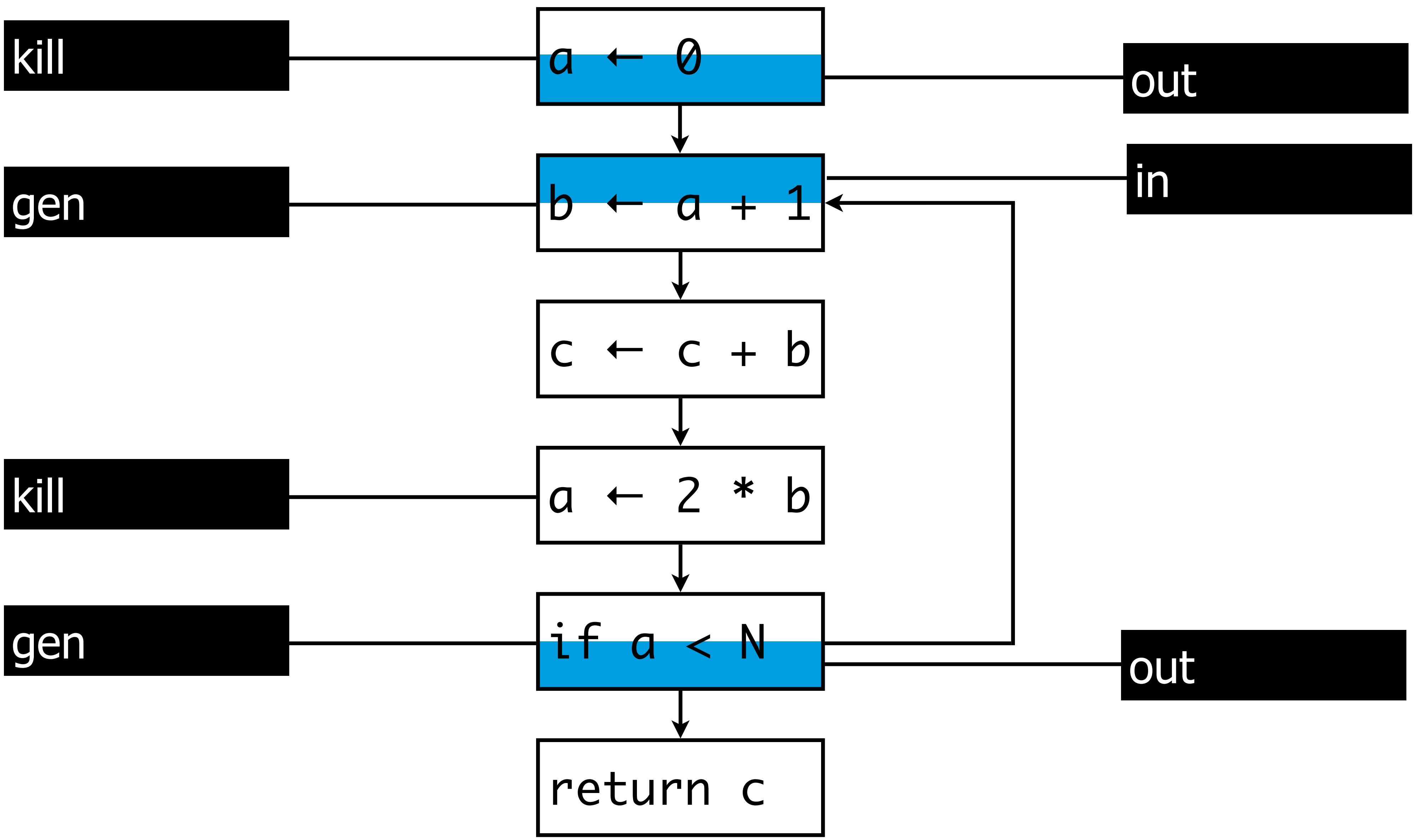
Liveness Analysis

generalisation



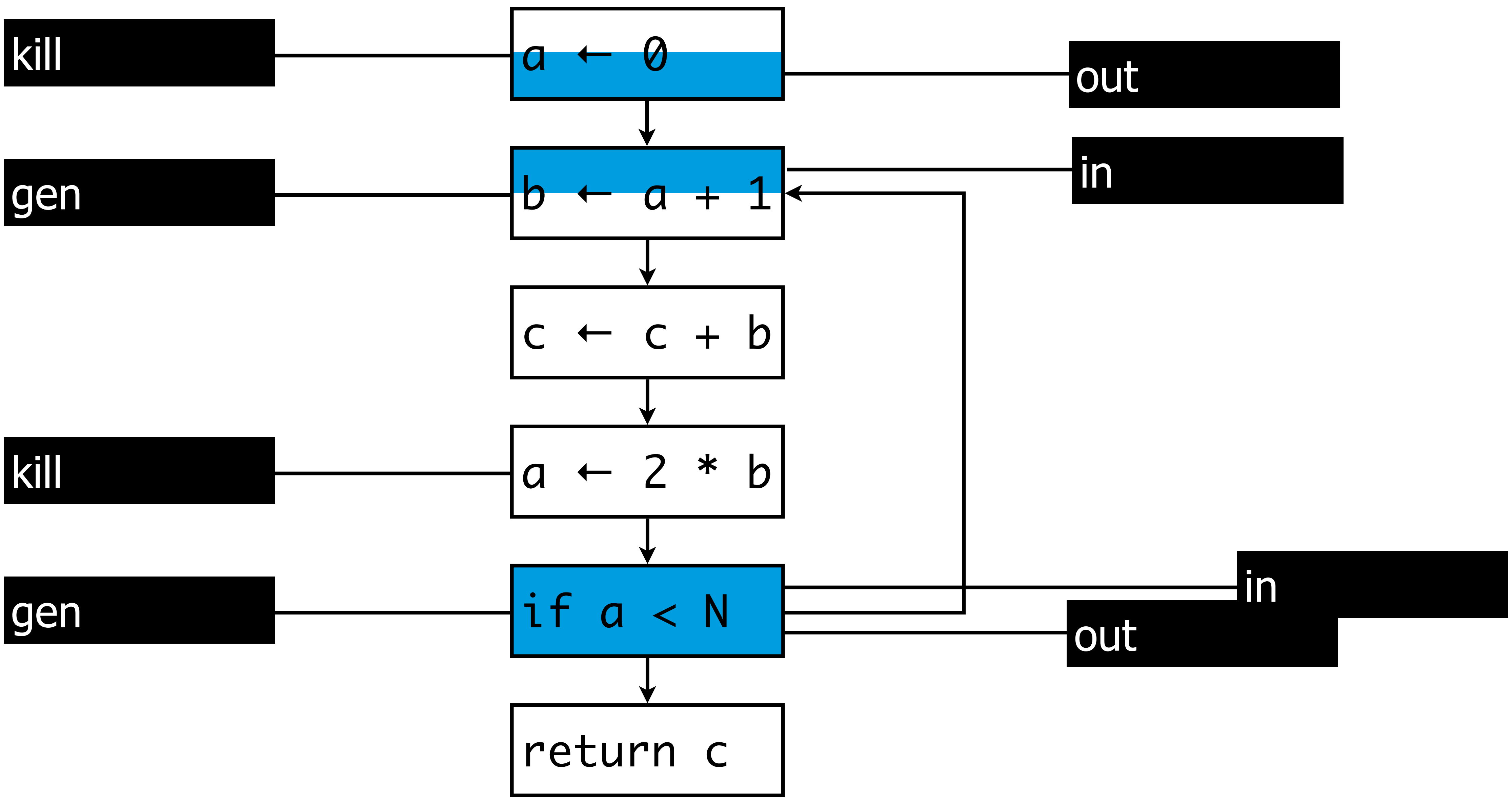
Liveness Analysis

generalisation



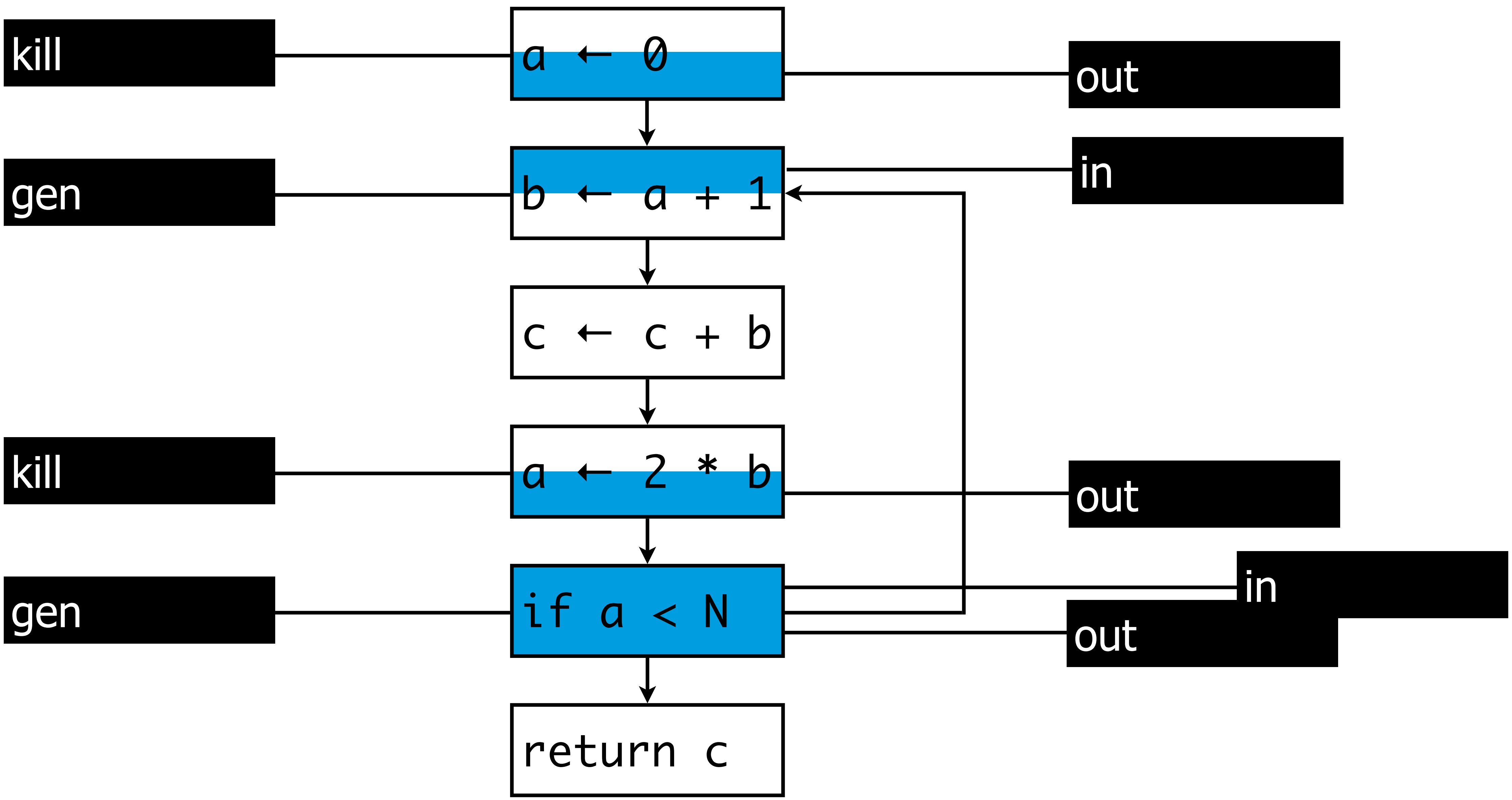
Liveness Analysis

generalisation



Liveness Analysis

generalisation



Liveness Analysis

gen & kill

	gen	
$a \leftarrow b \oplus c$	$\{b,c\}$	
$a \leftarrow b$	$\{b\}$	
$a \leftarrow M[b]$	$\{b\}$	
$M[a] \leftarrow b$	$\{a,b\}$	
$f(a_1, \ldots, a_n)$	$\{a_1,\ldots,a_n\}$	
$a \leftarrow f(a_1, \ldots, a_n)$	$\{a_1,\ldots,a_n\}$	
goto L		
if $a \otimes b$	$\{a,b\}$	

Liveness Analysis

gen & kill

	gen	kill
$a \leftarrow b \oplus c$	$\{b,c\}$	$\{a\}$
$a \leftarrow b$	$\{b\}$	$\{a\}$
$a \leftarrow M[b]$	$\{b\}$	$\{a\}$
$M[a] \leftarrow b$	$\{a,b\}$	
$f(a_1, \ldots, a_n)$	$\{a_1,\ldots,a_n\}$	
$a \leftarrow f(a_1, \ldots, a_n)$	$\{a_1,\ldots,a_n\}$	$\{a\}$
goto L		
if $a \otimes b$	$\{a,b\}$	

Liveness Analysis
generalisation

$$in[n] = gen[n] \cup (out[n] - kill[n])$$

$$out[n] = \bigcup_{s \in succ[n]} in[s]$$

IV

More Analyses

Reaching Definitions
definition

Unambiguous definition of a

- a statement of the form (d : a ← b ⊕ c) or (d : a ← M[x])

Definition d reaches statement u if

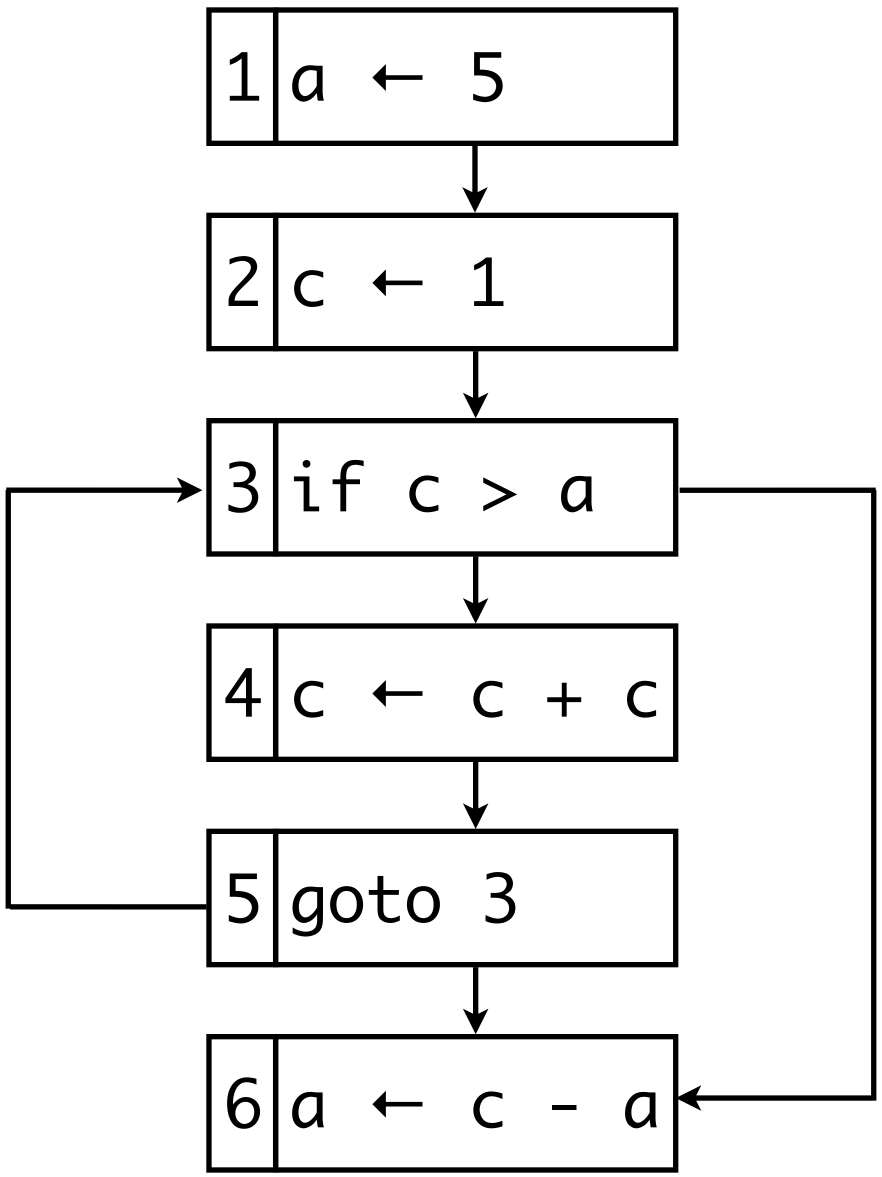
- there is some path in control-flow graph from d to u
- that does not contain an unambiguous definition of a

Used in

- constant propagation

Reaching Definitions

example



Reaching Definitions

gen & kill

	gen	
d: a ← b ⊕ c	{d}	
d: a ← b	{d}	
d: a ← M[b]	{d}	
M[a] ← b		
f(a_1, …, a_n)		
d: a ← f(a_1, …, a_n)	{d}	
goto L		
if a ⊗ b		

defs(a) : all definitions of a

Reaching Definitions

gen & kill

	gen	kill
d: $a \leftarrow b \oplus c$	{d}	defs(a)-{d}
d: $a \leftarrow b$	{d}	defs(a)-{d}
d: $a \leftarrow M[b]$	{d}	defs(a)-{d}
$M[a] \leftarrow b$		
$f(a_1, \ldots, a_n)$		
d: $a \leftarrow f(a_1, \ldots, a_n)$	{d}	defs(a)-{d}
goto L		
if $a \otimes b$		

defs(a) : all definitions of a

Reaching Definitions

formalisation

$$in[n] = \bigcup_{p \in pred[n]} out[p]$$

$$out[n] = gen[n] \cup \left(in[n] - kill[n]\right)$$

Available Expressions
definition

An expression ($b \oplus c$) is available at node n if

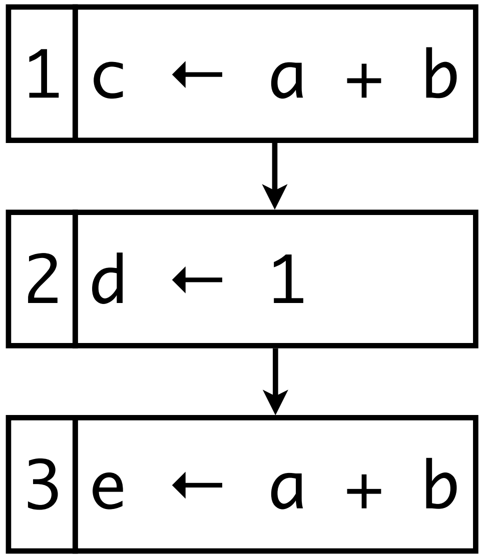
- on every path from the entry node to node n
- ($b \oplus c$) is computed at least once, and
- there are no definitions of b or c since most recent occurrence of ($b \oplus c$) on that path

Used in

- common-subexpression elimination

Available Expressions

example



Available Expressions

gen & kill

	gen	
d: $a \leftarrow b \oplus c$	{b⊕c}-kill	
d: $a \leftarrow b$		
d: $a \leftarrow M[b]$	{M[b]}-kill	
$M[a] \leftarrow b$		
$f(a_1, \ldots, a_n)$		
d: $a \leftarrow f(a_1, \ldots, a_n)$		
goto L		
if $a \otimes b$		

Available Expressions

gen & kill

	gen	kill
d: a ← b ⊕ c	{b⊕c}-kill	exps(a)
d: a ← b		
d: a ← M[b]	{M[b]}-kill	exps(a)
M[a] ← b		exps(M[_])
$f(a_1, \ldots, a_n)$		exps(M[_])
d: a ← $f(a_1, \ldots, a_n)$		exps(M[_]) ∪ exps(a)
goto L		
if a ⊗ b		

Available Expressions

formalisation

$$in[n] = \bigcap_{p \in pred[n]} out[p]$$

$$out[n] = gen[n] \cup (in[n] - kill[n])$$

Reaching Expressions
definition

An expression ($s : a \leftarrow b \oplus c$) reaches node n if

- there is a path from s to n
- that does not go through assignment to b or c
- or through any computation of ($b \oplus c$)

Used in

- common-subexpression elimination

V

Optimizations

Dead Code Elimination

example

```
a ← 0
b ← a + 1
c ← c + b
a ← 2 * b
return c
```

```
a ← 0
b ← a + 1
c ← c + b

return c
```

```
consider:
  (s : a ← b ⊕ c) or (s : a ← M[x])
  if a is not live-out at s
transform:
  remove s
```

Common Subexpression Elimination

example

```
c ← a + b
d ← 1
e ← a + b
```

```
x ← a + b
c ← x
d ← 1
e ← x
```

```
consider:
  (n : a ← b ⊕ c) reaches (s : d ← b ⊕ c)
  path from n to s does not compute b ⊕ c or define b or c
  e is a fresh variable
transform:
  n  : a ← b ⊕ c
  n’ : e ← a
  …
  s : d ← e
```

Constant Propagation

example

```
a ← 0
b ← a + 1
c ← c + b
a ← 2 * b
return c
```

```
a ← 0
b ← 0 + 1
c ← c + b
a ← 2 * b
return c
```

```
consider: (d : a ← c) & (n : e ← a ⊕ b)
if c is constant
   & (d reaches n)
   & (no other def of a reaches n)
transform:
  (n : e ← a ⊕ b) => (n : e ← c ⊕ b)
```

Copy Propagation

example

```
a ← e
b ← a + 1
c ← c + b
a ← 2 * b
return c
```

```
a ← e
b ← e + 1
c ← c + b
a ← 2 * b
return c
```

```
consider: (d : a ← z) & (n : e ← a ⊕ b)
if z is a variable
   & (d reaches n)
   & (no other def of a reaches n)
   & (no def of z on path from d to n)
transform:
  (n : e ← a ⊕ b) => (n : e ← z ⊕ b)
```