Compilation 2024

Liveness analysis

Amin Timany
timany@cs.au.dk

Based on slides by Aslan Askarov and E. Ernst

Allocate registers for the following program

Assuming we only have %rax, %rcx, %rsi, %rdi

```
define i64 @fact(i64 %n){
    %acc = alloca i64
    store i64 1, i64* %acc
    %i = alloca i64
    store i64 1, i64* %i
    br label %loop
loop:
    %l = load i64, i64* %i
    %c = icmp sle i64 %l, %n
    br i1 %c, label %loop_body, label %after_loop
loop_body:
    %m = load i64, i64* %acc
    %k = load i64, i64* %i
    %r = mul i64 %m, %k
    store i64 %r, i64* %acc
    %k1 = add i64 %k, 1
    store i64 %k1, i64* %i
    br label %loop
after_loop:
    %res = load i64, i64* %acc
    ret i64 %res
}
```

llvm reg	x86-64 reg	explanation
%n		
%acc		
%i		
%l		
%c		
%m		
%k		
%r		
…		

Allocate registers for the following program

Assuming we only have %rax, %rcx, %rsi, %rdi

```
define i64 @fact(i64 %n){
    %acc = alloca i64
    store i64 1, i64* %acc
    %i = alloca i64
    store i64 1, i64* %i
    br label %loop
loop:
    %l = load i64, i64* %i
    %c = icmp sle i64 %l, %n
    br i1 %c, label %loop_body, label %after_loop
loop_body:
    %m = load i64, i64* %acc
    %k = load i64, i64* %i
    %r = mul i64 %m, %k
    store i64 %r, i64* %acc
    %k1 = add i64 %k, 1
    store i64 %k1, i64* %i
    br label %loop
after_loop:
    %res = load i64, i64* %acc
    ret i64 %res
}
```

llvm reg	x86-64 reg	explanation
%n	%rdi	1st arg
%acc		
%i		
%l		
%c		
%m		
%k		
%r		
…		

Allocate registers for the following program

Assuming we only have %rax, %rcx, %rsi, %rdi

```
define i64 @fact(i64 %n){
    %acc = alloca i64
    store i64 1, i64* %acc
    %i = alloca i64
    store i64 1, i64* %i
    br label %loop
loop:
    %l = load i64, i64* %i
    %c = icmp sle i64 %l, %n
    br i1 %c, label %loop_body, label %after_loop
loop_body:
    %m = load i64, i64* %acc
    %k = load i64, i64* %i
    %r = mul i64 %m, %k
    store i64 %r, i64* %acc
    %k1 = add i64 %k, 1
    store i64 %k1, i64* %i
    br label %loop
after_loop:
    %res = load i64, i64* %acc
    ret i64 %res
}
```

llvm reg	x86-64 reg	explanation
%n	%rdi	1st arg
%acc	—	stack address
%i		
%l		
%c		
%m		
%k		
%r		
…		

Allocate registers for the following program

Assuming we only have %rax, %rcx, %rsi, %rdi

```
define i64 @fact(i64 %n){
    %acc = alloca i64
    store i64 1, i64* %acc
    %i = alloca i64
    store i64 1, i64* %i
    br label %loop
loop:
    %l = load i64, i64* %i
    %c = icmp sle i64 %l, %n
    br i1 %c, label %loop_body, label %after_loop
loop_body:
    %m = load i64, i64* %acc
    %k = load i64, i64* %i
    %r = mul i64 %m, %k
    store i64 %r, i64* %acc
    %k1 = add i64 %k, 1
    store i64 %k1, i64* %i
    br label %loop
after_loop:
    %res = load i64, i64* %acc
    ret i64 %res
}
```

llvm reg	x86-64 reg	explanation
%n	%rdi	1st arg
%acc	—	stack address
%i	—	stack address
%l		
%c		
%m		
%k		
%r		
…		

Allocate registers for the following program

Assuming we only have %rax, %rcx, %rsi, %rdi

```
define i64 @fact(i64 %n){
    %acc = alloca i64
    store i64 1, i64* %acc
    %i = alloca i64
    store i64 1, i64* %i
    br label %loop
loop:
    %l = load i64, i64* %i
    %c = icmp sle i64 %l, %n
    br i1 %c, label %loop_body, label %after_loop
loop_body:
    %m = load i64, i64* %acc
    %k = load i64, i64* %i
    %r = mul i64 %m, %k
    store i64 %r, i64* %acc
    %k1 = add i64 %k, 1
    store i64 %k1, i64* %i
    br label %loop
after_loop:
    %res = load i64, i64* %acc
    ret i64 %res
}
```

llvm reg	x86-64 reg	explanation
%n	%rdi	1st arg
%acc	—	stack address
%i	—	stack address
%l	%rax	some free reg
%c		
%m		
%k		
%r		
…		

Allocate registers for the following program

Assuming we only have %rax, %rcx, %rsi, %rdi

```
define i64 @fact(i64 %n){
    %acc = alloca i64
    store i64 1, i64* %acc
    %i = alloca i64
    store i64 1, i64* %i
    br label %loop
loop:
    %l = load i64, i64* %i
    %c = icmp sle i64 %l, %n
    br i1 %c, label %loop_body, label %after_loop
loop_body:
    %m = load i64, i64* %acc
    %k = load i64, i64* %i
    %r = mul i64 %m, %k
    store i64 %r, i64* %acc
    %k1 = add i64 %k, 1
    store i64 %k1, i64* %i
    br label %loop
after_loop:
    %res = load i64, i64* %acc
    ret i64 %res
}
```

llvm reg	x86-64 reg	explanation
%n	%rdi	1st arg
%acc	—	stack address
%i	—	stack address
%l	%rax	some free reg
%c	—	in flags
%m		
%k		
%r		
…		

Allocate registers for the following program

Assuming we only have %rax, %rcx, %rsi, %rdi

```
define i64 @fact(i64 %n){
    %acc = alloca i64
    store i64 1, i64* %acc
    %i = alloca i64
    store i64 1, i64* %i
    br label %loop
loop:
    %l = load i64, i64* %i
    %c = icmp sle i64 %l, %n
    br i1 %c, label %loop_body, label %after_loop
loop_body:
    %m = load i64, i64* %acc
    %k = load i64, i64* %i
    %r = mul i64 %m, %k
    store i64 %r, i64* %acc
    %k1 = add i64 %k, 1
    store i64 %k1, i64* %i
    br label %loop
after_loop:
    %res = load i64, i64* %acc
    ret i64 %res
}
```

llvm reg	x86-64 reg	explanation
%n	%rdi	1st arg
%acc	—	stack address
%i	—	stack address
%l	%rax	some free reg
%c	—	in flags
%m	%rcx	some free reg
%k		
%r		
…		

Allocate registers for the following program

Assuming we only have %rax, %rcx, %rsi, %rdi

```
define i64 @fact(i64 %n){
    %acc = alloca i64
    store i64 1, i64* %acc
    %i = alloca i64
    store i64 1, i64* %i
    br label %loop
loop:
    %l = load i64, i64* %i
    %c = icmp sle i64 %l, %n
    br i1 %c, label %loop_body, label %after_loop
loop_body:
    %m = load i64, i64* %acc
    %k = load i64, i64* %i
    %r = mul i64 %m, %k
    store i64 %r, i64* %acc
    %k1 = add i64 %k, 1
    store i64 %k1, i64* %i
    br label %loop
after_loop:
    %res = load i64, i64* %acc
    ret i64 %res
}
```

llvm reg	x86-64 reg	explanation
%n	%rdi	1st arg
%acc	—	stack address
%i	—	stack address
%l	%rax	some free reg
%c	—	in flags
%m	%rcx	some free reg
%k	%rsi	some free reg
%r		
…		

Allocate registers for the following program

Assuming we only have %rax, %rcx, %rsi, %rdi

```
define i64 @fact(i64 %n){
    %acc = alloca i64
    store i64 1, i64* %acc
    %i = alloca i64
    store i64 1, i64* %i
    br label %loop
loop:
    %l = load i64, i64* %i
    %c = icmp sle i64 %l, %n
    br i1 %c, label %loop_body, label %after_loop
loop_body:
    %m = load i64, i64* %acc
    %k = load i64, i64* %i
    %r = mul i64 %m, %k
    store i64 %r, i64* %acc
    %k1 = add i64 %k, 1
    store i64 %k1, i64* %i
    br label %loop
after_loop:
    %res = load i64, i64* %acc
    ret i64 %res
}
```

llvm reg	x86-64 reg	explanation
%n	%rdi	1st arg
%acc	—	stack address
%i	—	stack address
%l	%rax	some free reg
%c	—	in flags
%m	%rcx	some free reg
%k	%rsi	some free reg
%r	?	
…		

Allocate registers for the following program

Assuming we only have %rax, %rcx, %rsi, %rdi

```
define i64 @fact(i64 %n){
    %acc = alloca i64
    store i64 1, i64* %acc
    %i = alloca i64
    store i64 1, i64* %i
    br label %loop
loop:
    %l = load i64, i64* %i
    %c = icmp sle i64 %l, %n
    br i1 %c, label %loop_body, label %after_loop
loop_body:
    %m = load i64, i64* %acc
    %k = load i64, i64* %i
    %r = mul i64 %m, %k
    store i64 %r, i64* %acc
    %k1 = add i64 %k, 1
    store i64 %k1, i64* %i
    br label %loop
after_loop:
    %res = load i64, i64* %acc
    ret i64 %res
}
```

llvm reg	x86-64 reg	explanation
%n	%rdi	1st arg
%acc	—	stack address
%i	—	stack address
%l	%rax	some free reg
%c	—	in flags
%m	%rcx	some free reg
%k	%rsi	some free reg
%r	?	
…		

We have to spill one of the x86-64 registers onto the stack!

Allocate registers for the following program

Assuming we only have %rax, %rcx, %rsi, %rdi

```
define i64 @fact(i64 %n){
    %acc = alloca i64
    store i64 1, i64* %acc
    %i = alloca i64
    store i64 1, i64* %i
    br label %loop
loop:
    %l = load i64, i64* %i
    %c = icmp sle i64 %l, %n
    br i1 %c, label %loop_body, label %after_loop
loop_body:
    %m = load i64, i64* %acc
    %k = load i64, i64* %i
    %r = mul i64 %m, %k
    store i64 %r, i64* %acc
    %k1 = add i64 %k, 1
    store i64 %k1, i64* %i
    br label %loop
after_loop:
    %res = load i64, i64* %acc
    ret i64 %res
}
```

llvm reg	x86-64 reg	explanation
%n	%rdi	1st arg
%acc	—	stack address
%i	—	stack address
%l	%rax	some free reg
%c	—	in flags
%m	%rcx	some free reg
%k	%rsi	some free reg
%r	?	
…		

We have to spill one of the x86-64 registers onto the stack!

Can we do better?

Allocate registers for the following program

Assuming we only have %rax, %rcx, %rsi, %rdi

```
define i64 @fact(i64 %n){
    %acc = alloca i64
    store i64 1, i64* %acc
    %i = alloca i64
    store i64 1, i64* %i
    br label %loop
loop:
    %l = load i64, i64* %i
    %c = icmp sle i64 %l, %n
    br i1 %c, label %loop_body, label %after_loop
loop_body:
    %m = load i64, i64* %acc
    %k = load i64, i64* %i
    %r = mul i64 %m, %k
    store i64 %r, i64* %acc
    %k1 = add i64 %k, 1
    store i64 %k1, i64* %i
    br label %loop
after_loop:
    %res = load i64, i64* %acc
    ret i64 %res
}
```

llvm reg	x86-64 reg	explanation
%n	%rdi	1st arg
%acc	—	stack address
%i	—	stack address
%l	%rax	some free reg
%c	—	in flags
%m	%rcx	some free reg
%k	%rsi	some free reg
%r	?	
…		

We have to spill one of the x86-64 registers onto the stack!

Can we do better?

Yes, place %r in %rax because %l is never used again!

Liveness analysis

- For good complexity management, we have used an unbounded number of “registers” (temporaries)
- To save resources, many of them can be merged
- How do we find out which can be safely merged?

Liveness

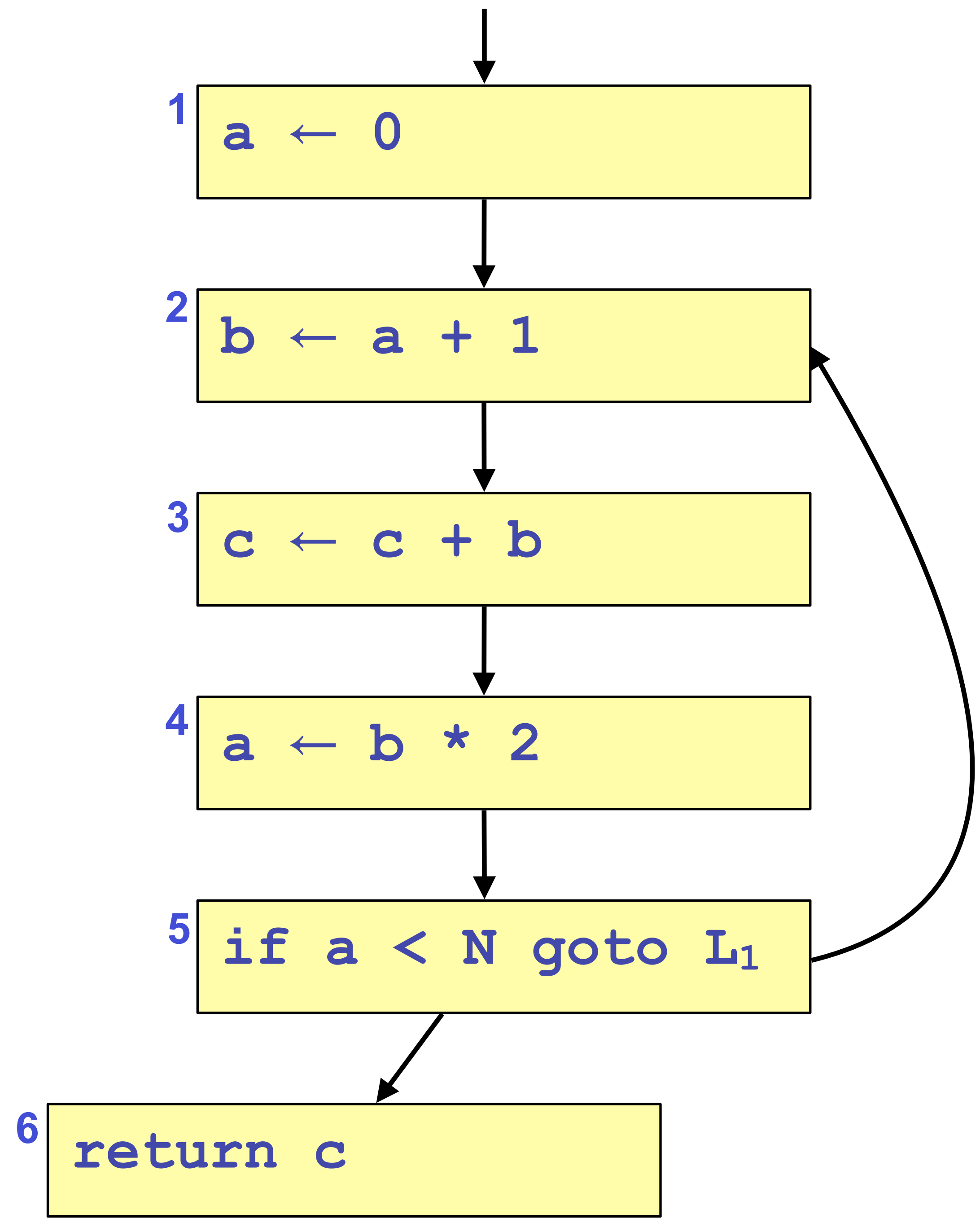
- A variable is live at a point in an execution if its value is needed in the future
- This is a dynamic definition
- Ex: t118 is live in line 2 ➜
- Static liveness: the value is definitely not needed vs. may be needed in the future

```
movl $4, t119
movl t119, t118
movl $0, t120
movl t120, %eax
imull t118
```

Liveness Example

- Consider a program and a graph showing control

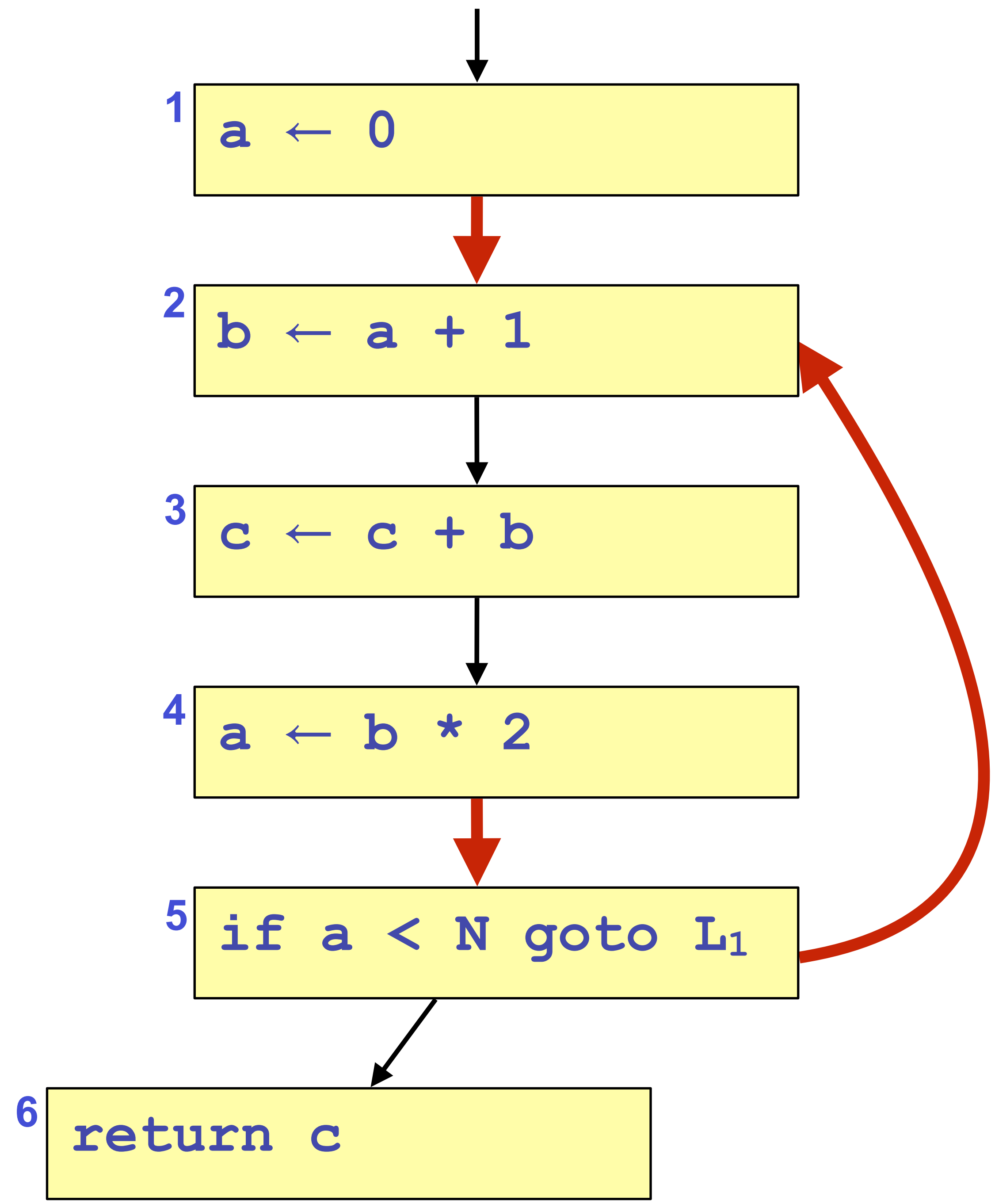
```
      a ← 0
L1:   b ← a + 1
      c ← c + b
      a ← b * 2
      if a < N goto L1
      return c
```



Liveness Example

- Liveness of variable a

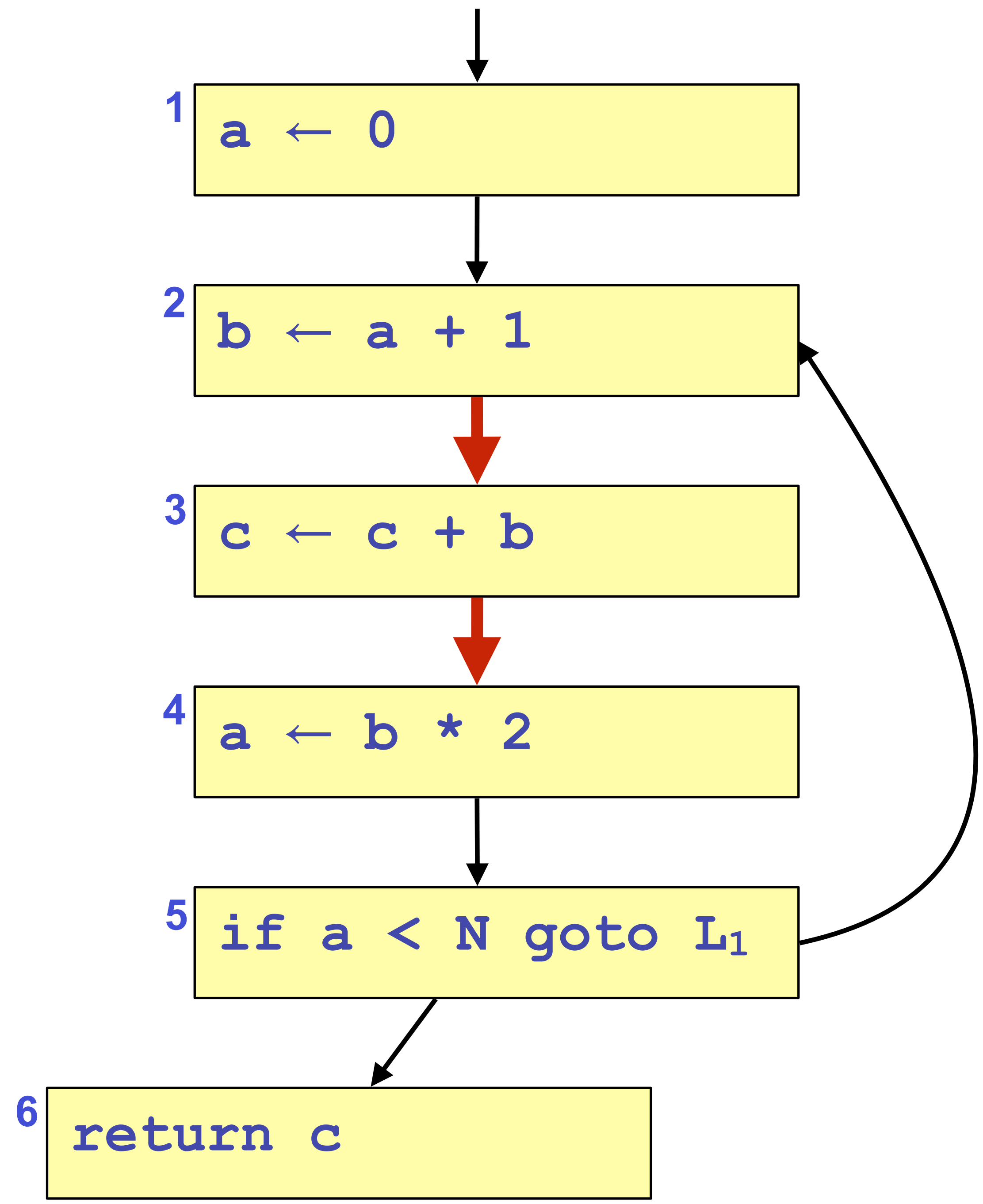
```
     a ← 0
L1:  b ← a + 1
     c ← c + b
     a ← b * 2
     if a < N goto L1
     return c
```



Liveness Example

- Liveness of variable b

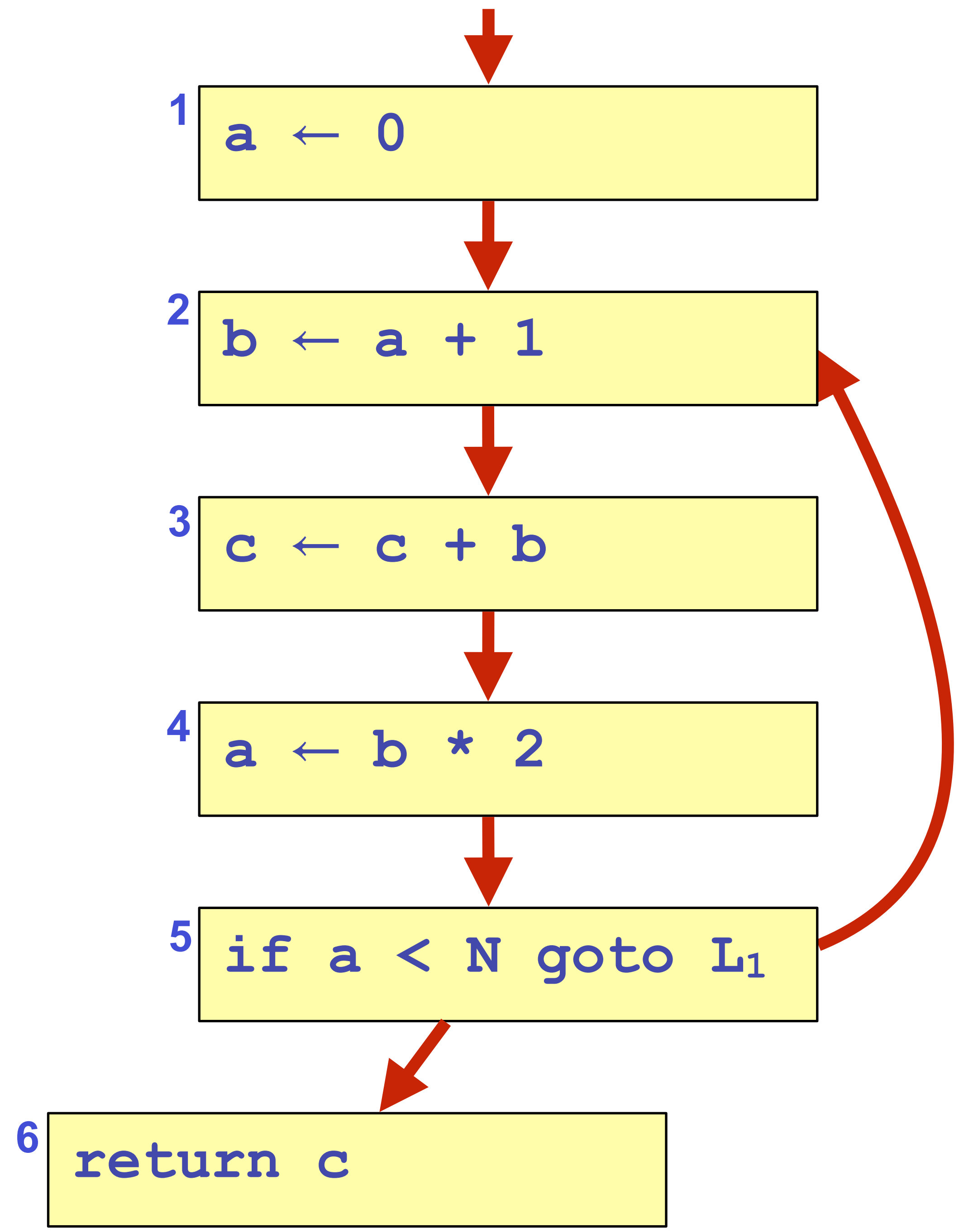
```
      a ← 0
L1:   b ← a + 1
      c ← c + b
      a ← b * 2
      if a < N goto L1
      return c
```



Liveness Example

- Liveness of variable c

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

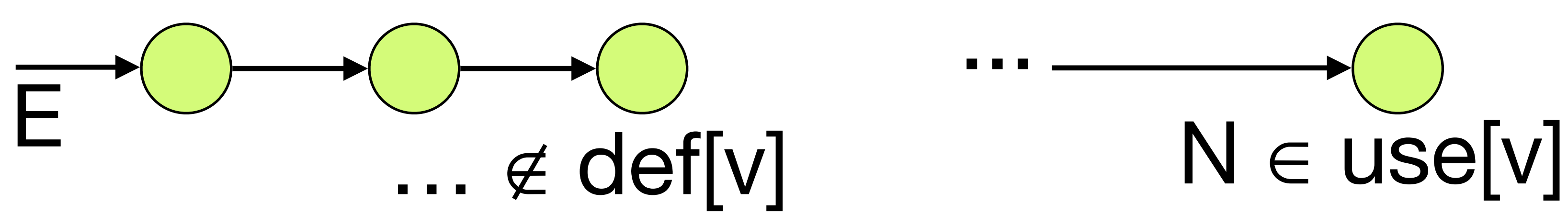


Concepts – Terminology

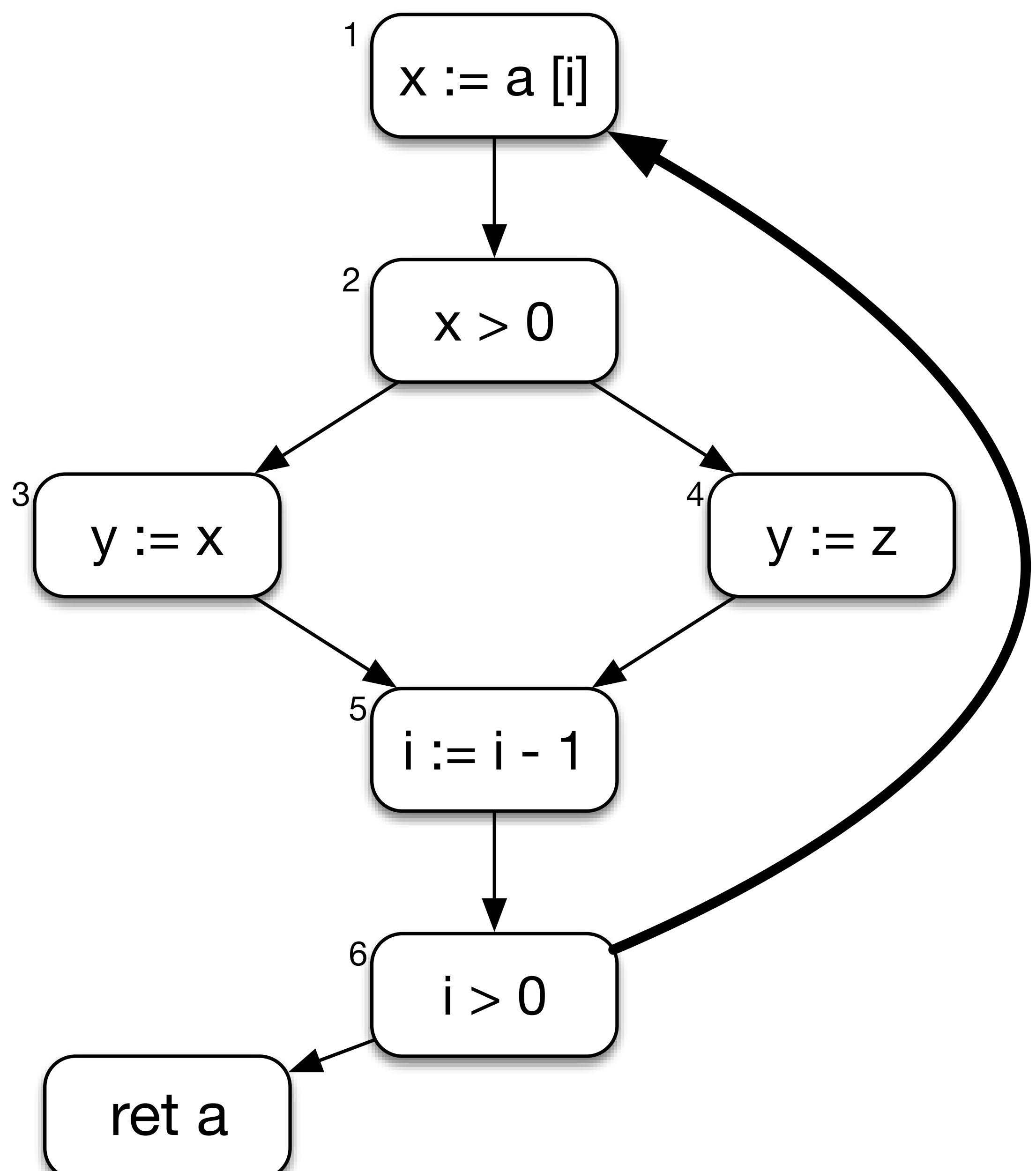
- Wish to reason about data-flow: tracing values
- Concepts:
  - Graph: *node*, *edge*, *predecessor*, *successor*, *in-edge*, *out-edge*
  - Variable: node *defines* var, node *uses* var, var *live on* edge, var *live-in* at node, var *live-out* at node
- Data: use[n],use[v],def[n],def[v],in[n],out[n],succ,pred
- Liveness, formally: variable v is live on edge E if there exists a directed path starting with E and ending in node N $\in$ use[v] not going through any N' $\in$ def[v]

Concepts – Terminology

- Wish to reason about data-flow: tracing values
- Concepts:
  - Graph: *node*, *edge*, *predecessor*, *successor*, *in-edge*, *out-edge*
  - Variable: node *defines* var, node *uses* var, var *live on* edge, var *live-in* at node, var *live-out* at node
- Data: use[n],use[v],def[n],def[v],in[n],out[n],succ,pred
- Liveness, formally: variable v is live on edge E if there exists a directed path starting with E and ending in node N $\in$ use[v] not going through any N' $\in$ def[v]



Question



What variables are *live* on edge 6 - 1?

a) {a , i, x }

b) {a , i, y}

c) {a , i, z}

d) {a , i}

e) ALL

Data-Flow Related Properties

- Local properties computed directly:
  - succ, pred, use[n], def[n]
- Properties derived by simple summary:
  - use[v], def[v]
- Properties depending on graph structure:
  - in[n], out[n]

Data-Flow Related Properties

- Local relations satisfied by solution:

$$v \in use[n] \Rightarrow v \in in[n]$$
$$v \in in[n] \Rightarrow \forall m \in pred[n]:\ v \in out[m]$$
$$v \in out[n] \setminus def[n] \Rightarrow v \in in[n]$$

- Global equations satisfied by solution:

$$in[n] = use[n] \cup (out[n] \setminus def[n])$$
$$out[n] = \bigcup_{s \in succ[n]} in[s]$$

Computing Data-Flow Properties

- Simple algorithm, using *fixed point iteration*

```
foreach n { in[n] := {}; out[n] := {} }
repeat
        foreach 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 (∀n: in'[n]=in[n] ∧ out'[n] = out[n])
```

Compare!

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

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

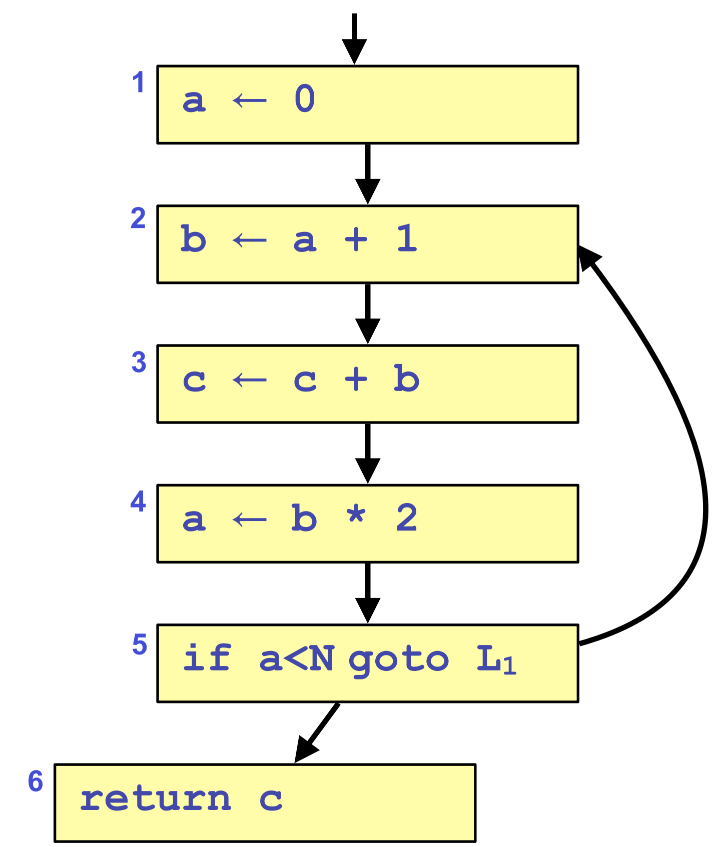
Example Liveness Computation

- Recall:

$$\text{in}[n] = \text{use}[n] \cup (\text{out}[n] \setminus \text{def}[n])$$

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

node	*use def*								
1	_ a								
2	a b								
3	bc c								
4	b a								
5	a _								
6	c _								



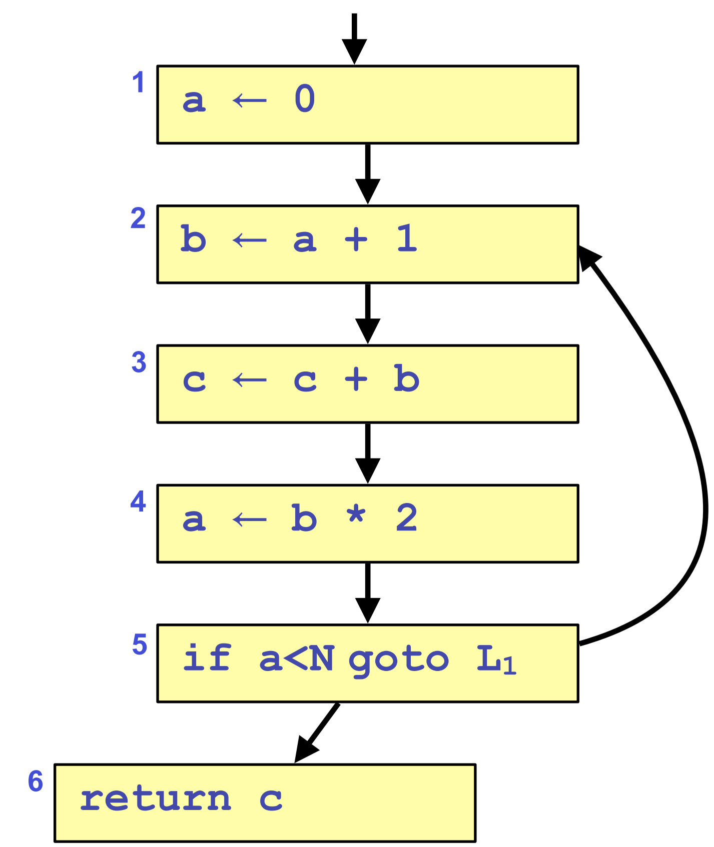
Example Liveness Computation

- Recall:

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

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

node	*use def*	*in out*							
1	_ a	_ _							
2	a b	_ _							
3	bc c	_ _							
4	b a	_ _							
5	a _	_ _							
6	c _	_ _							



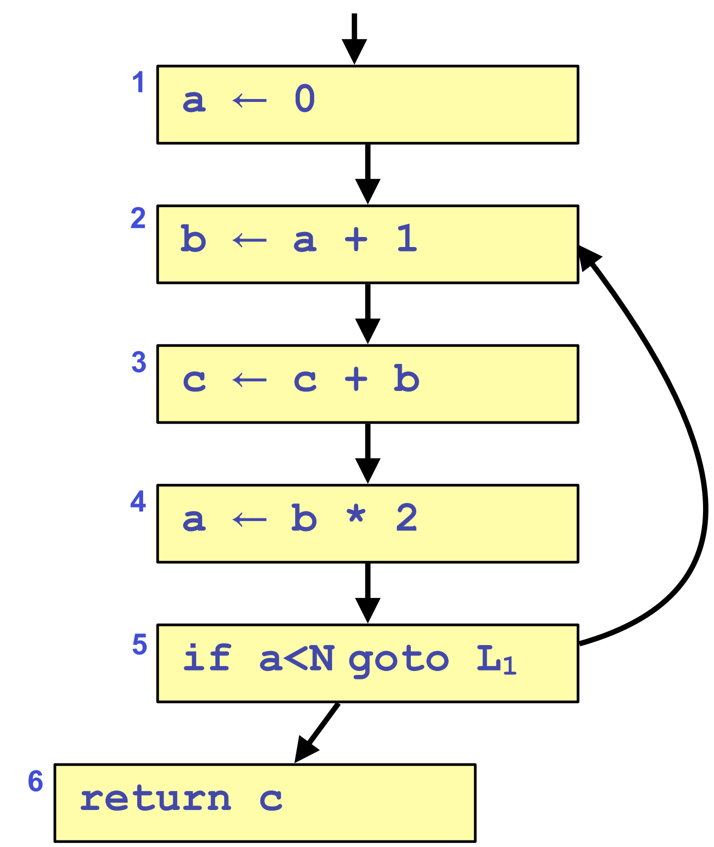
Example Liveness Computation

- Recall:

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

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

node	*use def*	*in out*	*in out*						
1	_ a	_ _	_ _						
2	a b	_ _	a _						
3	bc c	_ _	bc _						
4	b a	_ _	b _						
5	a _	_ _	a a						
6	c _	_ _	c _						



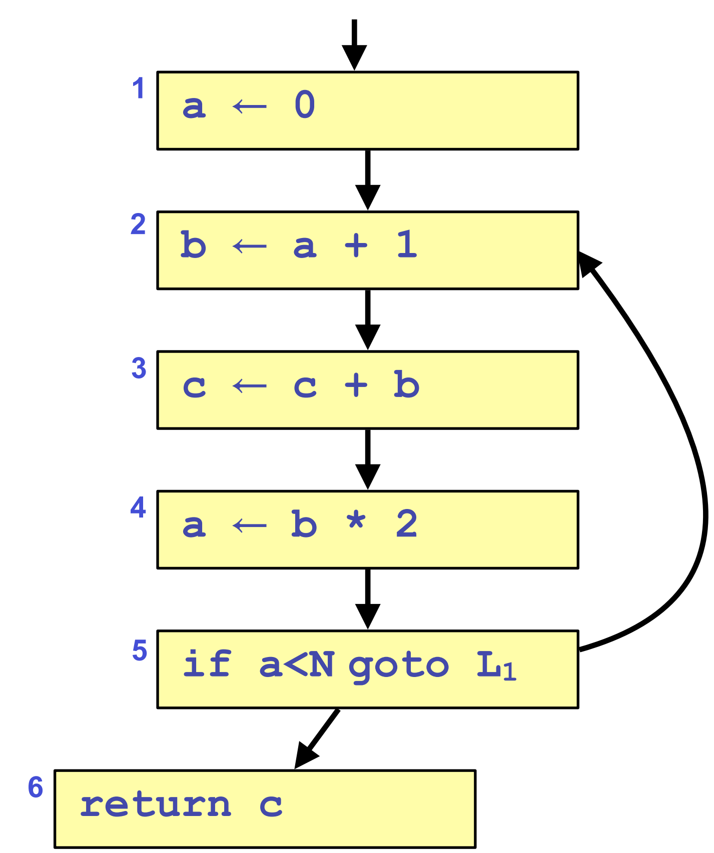
Example Liveness Computation

- Recall:

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

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

node	*use def*	*in out*	*in out*	*in out*					
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 _					



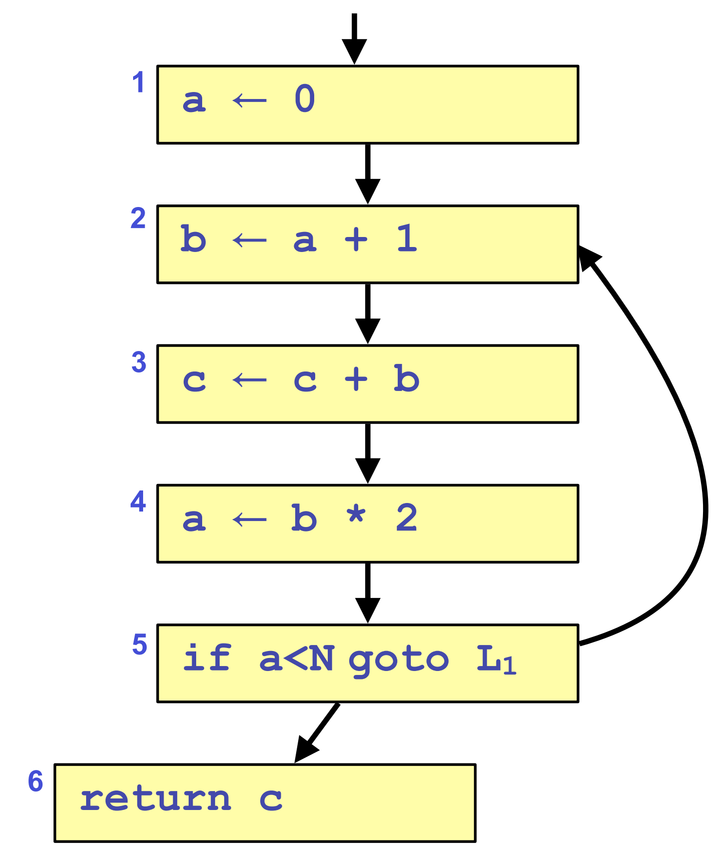
Example Liveness Computation

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

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

- Recall:

node	*use def*	*in out*	*in out*	*in out*	*in out*				
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 _				



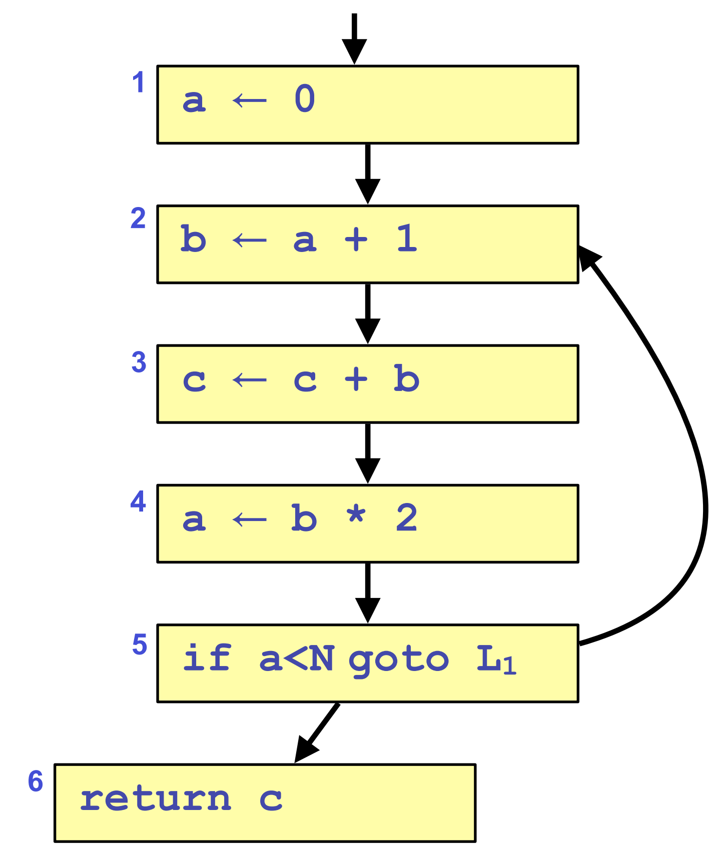
Example Liveness Computation

- Recall:

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

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

node	*use def*	*in out*	*in out*	*in out*	*in out*	*in out*			
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 _			



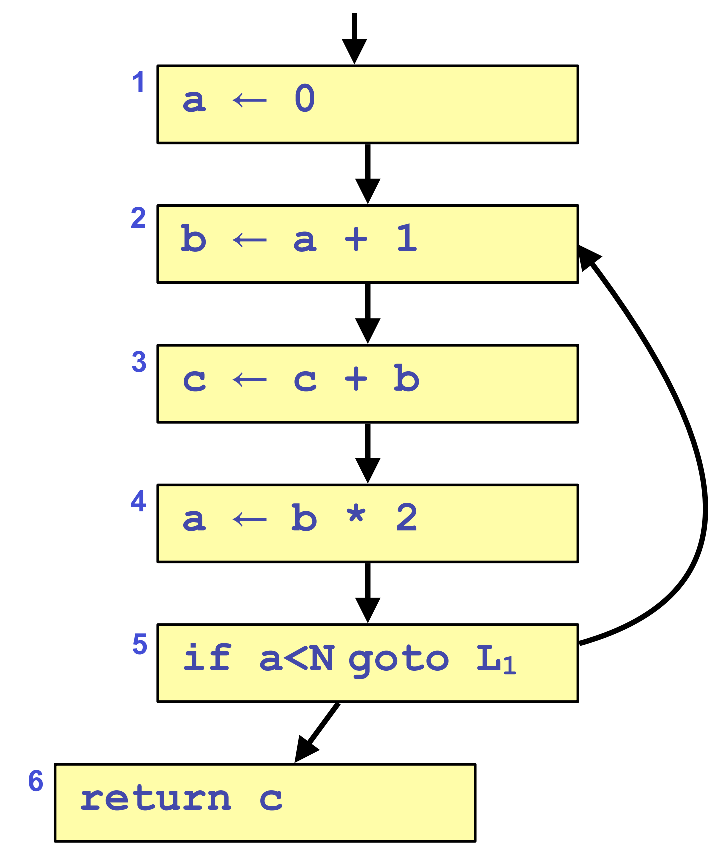
Example Liveness Computation

- Recall:

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

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

node	*use def*	*in out*	*in out*	*in out*	*in out*	*in out*	*in out*		
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 _		



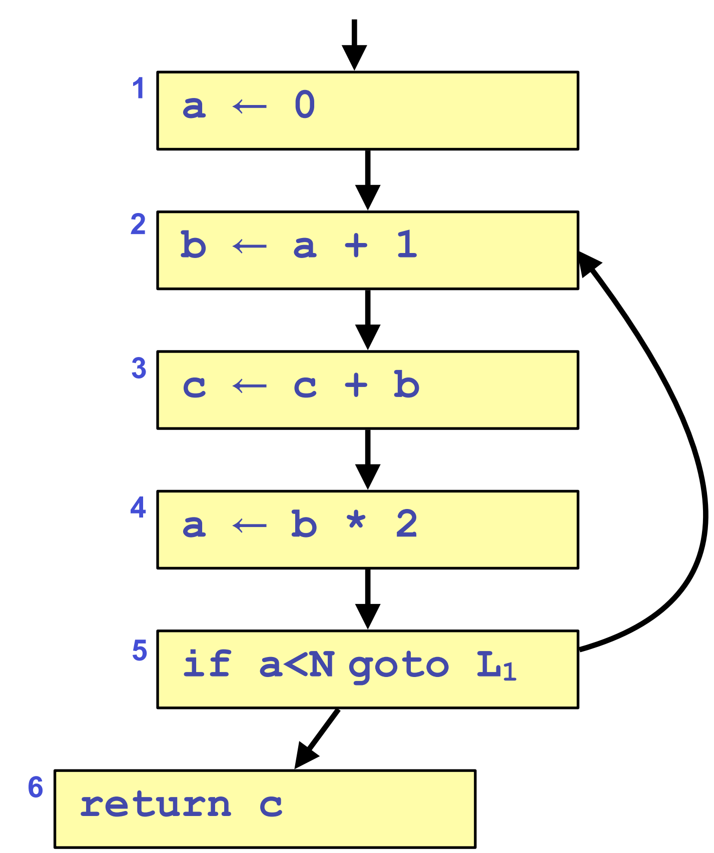
Example Liveness Computation

- Recall:

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

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

node	*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	
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 _	



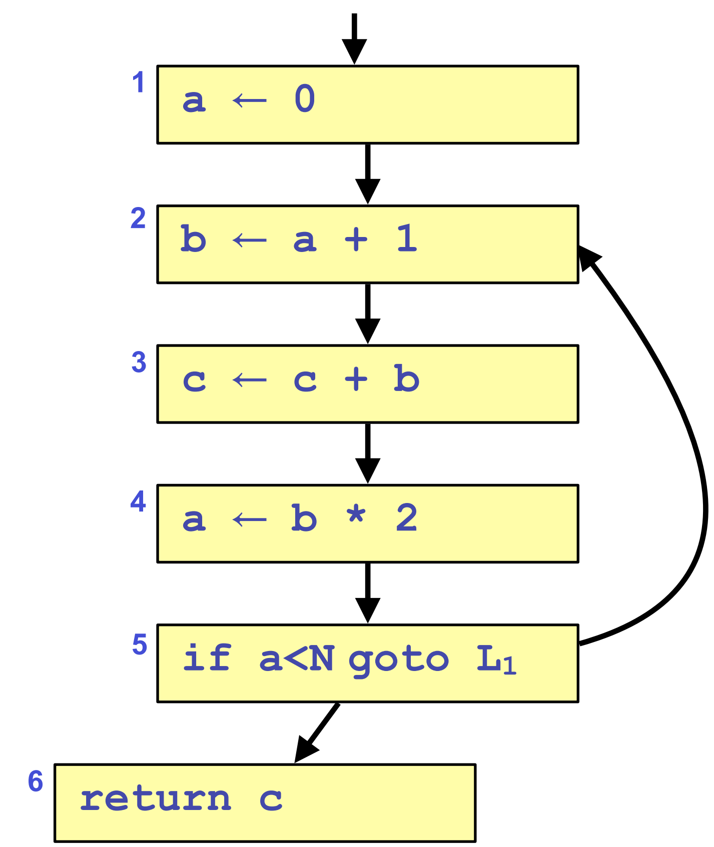
Example Liveness Computation

- Recall:

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

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

node	*use def*	*in out*	*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 _



Extras on Liveness Computations

- Use dependency guided ordering!
- Use basic blocks rather than instructions: Larger use[n], def[n], simpler graph — nice trade-off!
- Work on single variable at a time: Good with many temps having short live ranges
- Set representation:
  - bit arrays: union = bit-or, good for dense data
  - sorted lists: union = merge, good for sparse data
  - … union linear in any case, as well as set-difference …

Time Complexity

- With program size *N* “in some sense”
  - number of vars: $O(N)$
  - number of nodes: $O(N)$

```
foreach n { in[n] := {}; out[n] := {} }
repeat
        foreach 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 (∀n: in'[n]=in[n] ∧ out'[n] = out[n])
```

$O(N)$

$O(N^4)$

$O(N^2)$

$O(N)$

In practice, with good ordering: $O(N)$ … $O(N^2)$

Interference Graphs

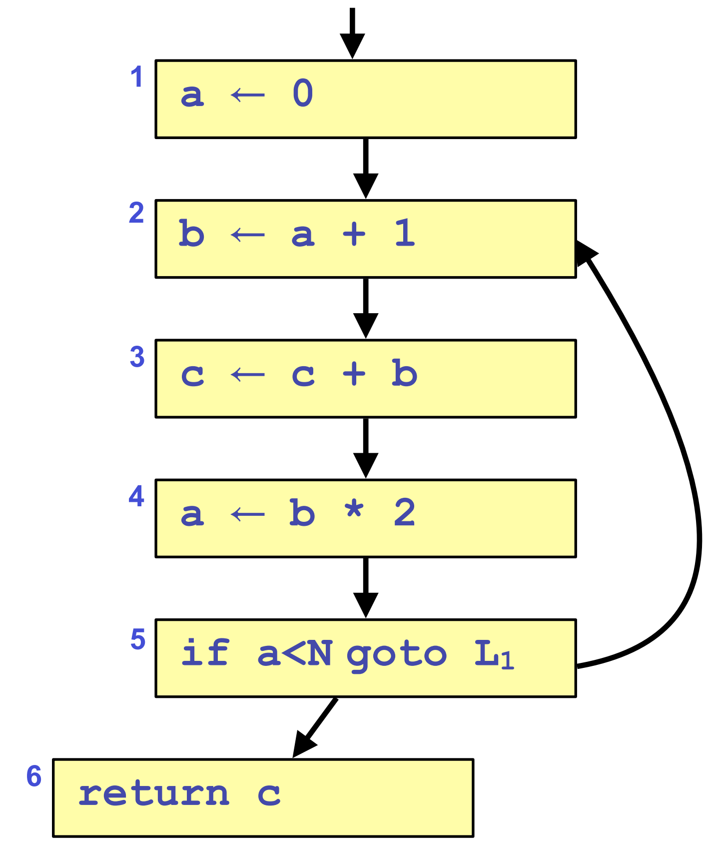
- Liveness is used for several purposes, notably register allocation
- Point: can merge two temps without “conflict”

- Def: interference exists among two vars *v,w* iff they have overlapping liveness paths

- Main case: both are live on same edge
- Tricky exceptions:
  - MOVE instructions
  - Instructions can only produce value in specific register

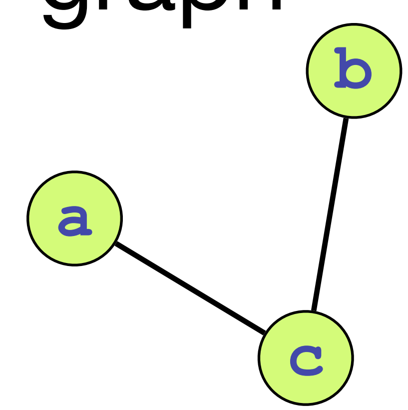
Interference Graph Example

- Recall example graph:



- Live ranges:
  - a: { 1→2, 4 →5, 5→2 }
  - b: { 2→3, 3→4 }
  - c: everywhere

- Interference represented as matrix or graph



MOVE and Interference

- MOVE instructions are special: We know that the values of two registers are equal
- … so they *can!* be merged even though they have overlapping live paths

- Algorithm core:
  - at non-MOVE n defining a with out[n] = $b_1 \ldots b_j$, add interference edges $(a,b_1)..(a,b_j)$
  - at MOVE n defining a using c with out[n]= $b_1 \ldots b_j$, add interference edges $(a,b_1)..(a,b_j)$, omitting (a,b_k) if $b_k=c$

Summary

- Temporaries allocated liberally — now clean up
- Liveness enables merging conflict-free temps
- Main concepts/data: use, def, in, out
- Local relations/global equations
- Algorithm similar to global equations
- Note ordering of nodes!
- Complexity bad (N^4), in practice near-linear
- Interference graphs — note exception for MOVE