Compiler Design

Lecture 16: Liveness Analysis

Christophe Dubach
Winter 2022

Some material from Prof. Michelle Strout, CS553, Colorado State University.

Timestamp: 2022/03/08 14:36:00

Example of generated MIPS code (using virtual registers):

```
.data
x: .space 4
y: .space 4

.text
      la  v0, x
      lw  v1, (v0)
      add v2, v0, v1
      la  v3, y
      lw  v4, (v3)
      sub v5, v4, v2
      add v6, v2, v4
      sw  v5, (v0)
      sw  v6, (v3)
```

After "'proper" register allocation (possible output):

```
.data
x: .space 4
y: .space 4

.text
      la  $t0, x
      lw  $t1, ($t0)
      add $t2, $t0, $t1
      la  $t3, y
      lw  $t4, ($t3)
      sub $t5, $t4, $t2
      add $t6, $t2, $t4
      sw  $t5, ($t0)
      sw  $t6, ($t3)
```

What if less than 7 architectural registers available for allocation?

- Need to know which values is going to be used in the future.

Liveness

Definition

A variable (virtual register) is live at some point in the program if it has previously been defined by an instruction and will be used by an instruction in the future. It is dead otherwise.

Two variables can use the same architectural register if they are never used at the same time, *i.e.* never simulataneously live.

$\Rightarrow$ Register allocation use liveness information.

Example:

```
.data
x: .space 4
y: .space 4
.text                    | Live after instruction:
      la  v0, x          | v0
      lw  v1, (v0)       | v0 v1
      add v2, v1, v1     | v0    v2
      la  v3, y          | v0    v2 v3
      lw  v4, (v3)       | v0    v2 v3 v4
      sub v5, v4, v2     | v0    v2 v3 v4 v5
      add v6, v2, v4     | v0       v3    v5 v6
      sw  v5, (v0)       |          v3       v6
      sw  v6, (v3)       |
```

Question: how many architectural registers are needed?

Computing liveness is more complicated in the presence of control flow (*e.g.* loops, if-then-else).

Assembly pseudo-code: [1]

```
    a = 0
L1: b = a + 1
    c = c + b
    a = b*2
    if (a<9) goto L1
    return c
```

Question: what is the live range of b?

To answer this question we need to understand the dynamic flow of the program execution.

[1] We illustrate concepts at a slightly higher level than assembly from this point on.

Control-Flow Graph (CFG)

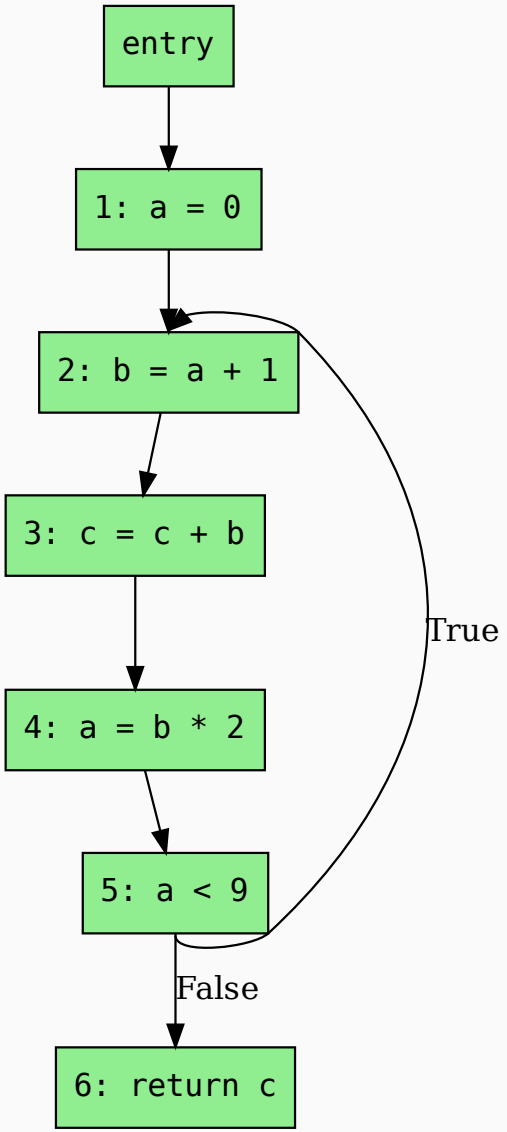
Concept invented in 1970 by:

Frances Allen (1932–2020), IBM, (1st woman to receive Turing Award in 2006!)

source: Rama, CC BY-SA 2.0 FR, wikimedia

```
    a = 0
L1: b = a + 1
    c = c + b
    a = b*2
    if (a<9) goto L1
    return c
```

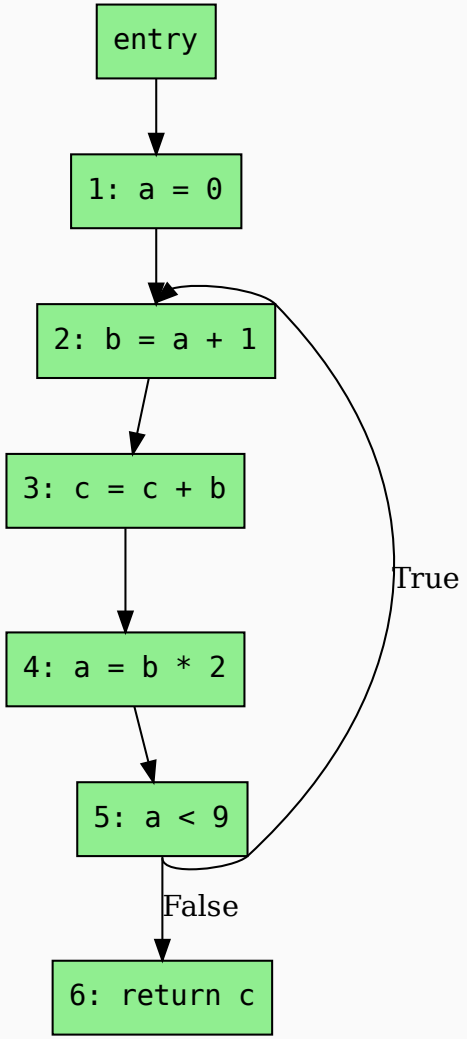
Directed graph:



What is the live range of b?

- **b** is used in statement 4, so **b** is live on the $3 \to 4$ edge
- since statement 3 does not define **b**, **b** is also live on the $2 \to 3$ edge
- statement 2 defines **b**, so any value of **b** on the $1 \to 2$ and $5 \to 2$ edges are not needed, so **b** is dead along these edges

b live range is $2 \to 3 \to 4$



Live range of **a**:

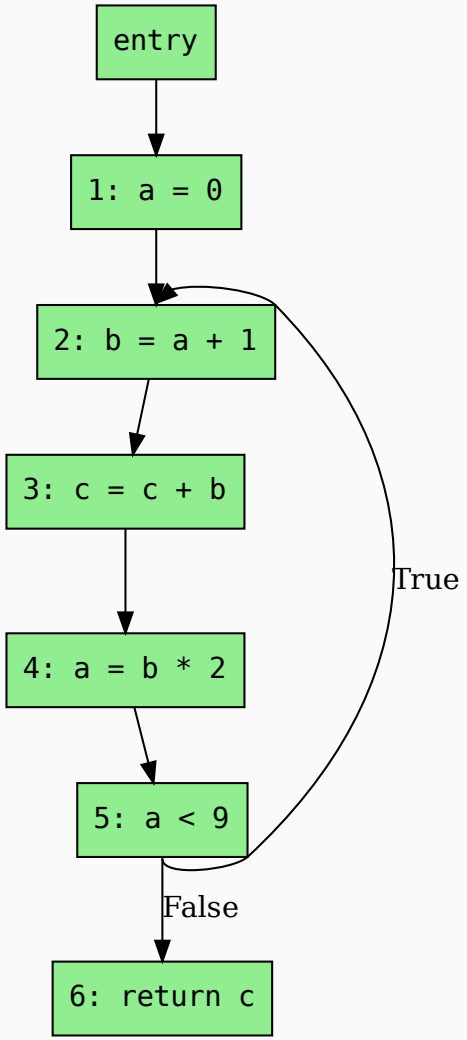
- $1 \to 2$ and $4 \to 5 \to 2$

Live range of **b**:

- $2 \to 3 \to 4$

Live range of **c**:

- *entry* $\to 1 \to 2 \to 3 \to 4 \to 5 \to 2$ and $5 \to 6$



Since **a** and **b** never simultaneously live, can share a register.

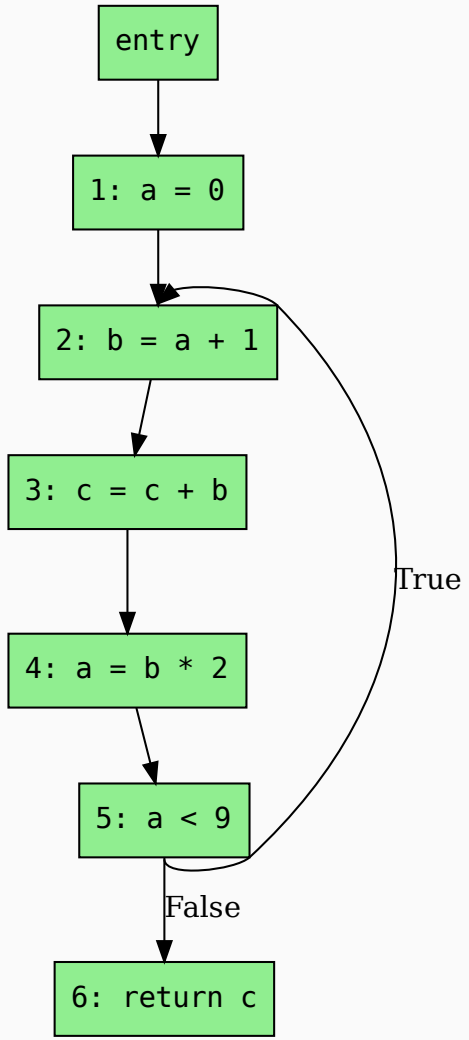
Terminology

Flow Graph

- a Control Flow Graph (CFG) has out-edges that leads to successor nodes and in-edges that come from predecessor nodes
- pred(n) = set of all predecessors of node n
  succ(n) = set of all successors of node n

Examples

- Out-edges of node 5: $5 \rightarrow 6$ and $5 \rightarrow 2$
- succ(5) = {2,6}
- pred(5) = {4}
- pred(2) = {1,5}



Uses and Defs

Def (definition)

- A write of a value to a variable
- def(v) = set of CFG nodes that define variable **v**
- def(n) = set of variables defined at node **n**

```
1: a = 0
```

Use

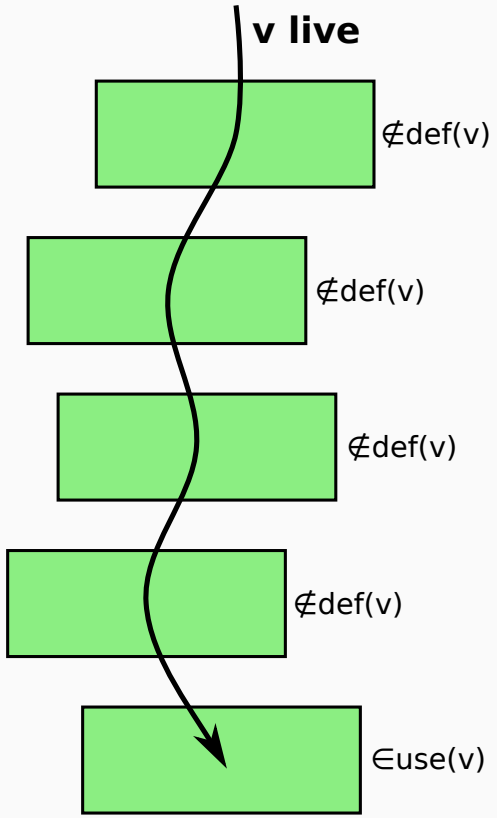
- A read of a variable's value
- use(v) = set of CFG nodes that use variable **v**
- use(n) = set of variables used at node **n**

```
5: a < 9
```

More precise definition of liveness

A variable v is live on a CFG edge if

- $\exists$ a directed path from that edge to a use of v (node $\in$ use(b)) and
- that path does not go through any def of v (nodes $\notin$ def(v)).



Computing Liveness

Flow of Liveness

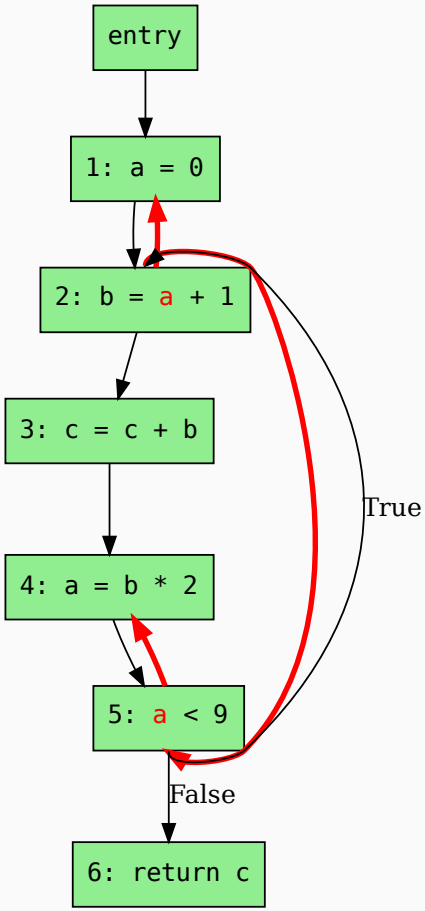
Data-flow

- Liveness of variables is a property that flows through the edges of the CFG

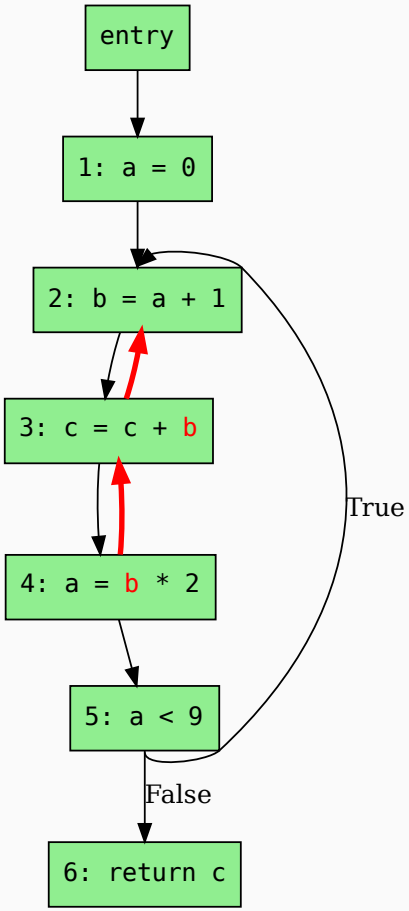
Direction of flow

- Liveness flows backward in the CFG:
  behaviour of future nodes determines liveness at a given node

Example: flow of liveness for a



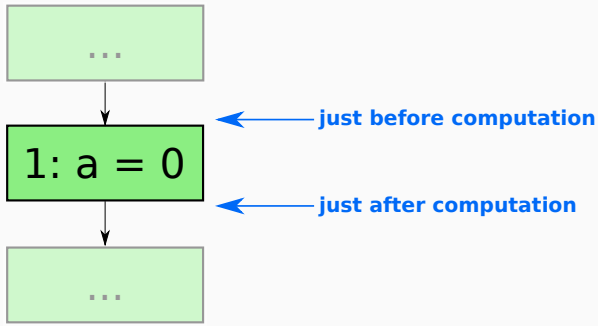
Example: flow of liveness for b



Liveness at Nodes

We have liveness on edges

- before and after each node



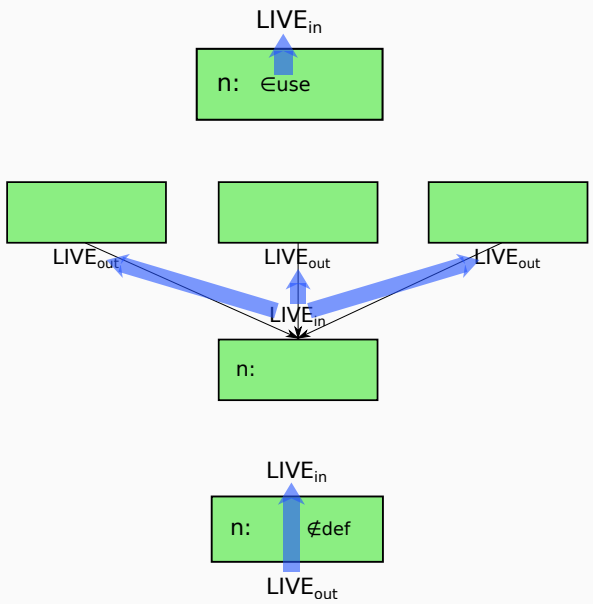
Two more definitions:

- A variable is live-out at a node if it is live on any of that node's out-edges
- A variable is live-in at a node if it is live on any of that node's in-edges

Computing Liveness

Rules for computing liveness

1. Generate liveness:
   $v \in use(n) \Rightarrow v \in \text{LIVE}_{in}(n)$
2. Push liveness across edges:
   $v \in \text{LIVE}_{in}(n) \Rightarrow \forall_{p \in pred(n)} v \in \text{LIVE}_{out}(p)$
3. Push liveness across nodes:
   $v \in \text{LIVE}_{out}(n) \wedge v \notin def(n) \Rightarrow v \in \text{LIVE}_{in}(n)$



Data-flow equations

$$\text{LIVE}_{in}(n) = \boxed{use(n)}_1 \cup \boxed{(\text{LIVE}_{out}(n) - def(n))}_3$$

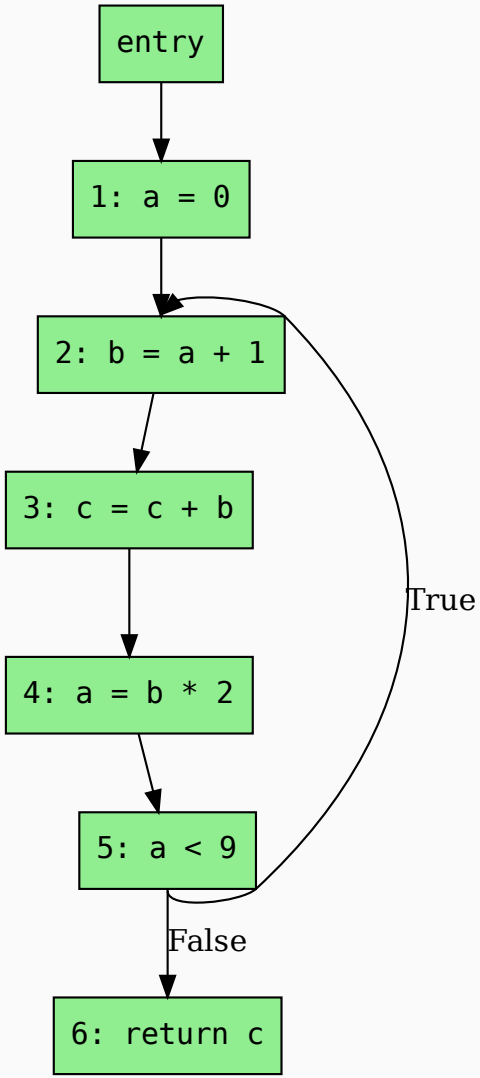
$$\text{LIVE}_{out}(n) = \boxed{\bigcup_{\forall s \in succ(n)} \text{LIVE}_{in}(s)}_2$$

Solving the Data-flow equations

1: **for all** node n $\in$ CFG **do**
2: $\quad \text{LIVE}_{in}(n) = \varnothing$
3: $\quad \text{LIVE}_{out}(n) = \varnothing$
4: **end for**
5: **repeat**
6: $\quad$ **for all** node n $\in$ CFG **do**
7: $\quad\quad \text{LIVE}'_{in}(n) = \text{LIVE}_{in}(n)$
8: $\quad\quad \text{LIVE}'_{out}(n) = \text{LIVE}_{out}(n)$
9: $\quad\quad \text{LIVE}_{in}(n) = use(n) \cup (\text{LIVE}_{out}(n) - def(n))$
10: $\quad\quad \text{LIVE}_{out}(n) = \bigcup_{\forall s \in succ(n)} \text{LIVE}_{in}(s)$
11: $\quad$ **end for**
12: **until** $\text{LIVE}'_{in}(n) = \text{LIVE}_{in}(n) \wedge \text{LIVE}'_{out}(n) = \text{LIVE}_{out}(n) \forall n$

This is a fix-point algorithm for iterative liveness analysis.

Example



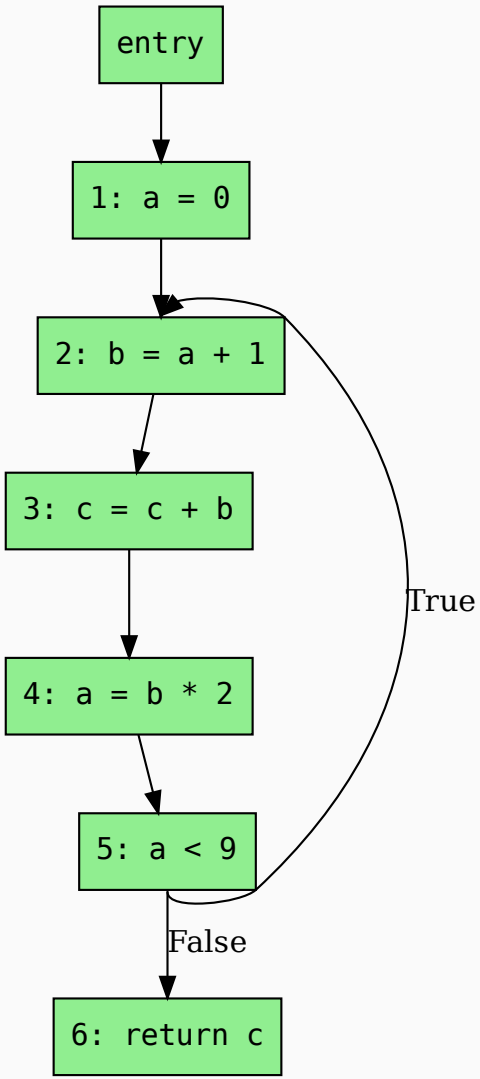
node	use	def	1st		2nd		3rd		4th		5th		6th		7th	
			in	out	in	out	in	out	in	out	in	out	in	out	in	out
1		a				a		a		ac	c	ac	c	ac	c	ac
2	a	b	a		a	bc	ac	bc	ac	bc	ac	bc	ac	bc	ac	bc
3	bc	c	bc		bc	b	bc	b	bc	b	bc	b	bc	bc	bc	bc
4	b	a	b		b	a	b	a	b	ac	bc	ac	bc	ac	bc	ac
5	a		a	a	a	ac	ac	ac	ac	ac	ac	ac	ac	ac	ac	ac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

Data-flow equations

$$\text{LIVE}_{in}(n) = use(n) \cup (\text{LIVE}_{out}(n) - def(n))$$

$$\text{LIVE}_{out}(n) = \bigcup_{\forall s \in succ(n)} \text{LIVE}_{in}(s)$$

There is something inefficient about this process.



For instance, consider the $3 \to 4$ edge in the graph:

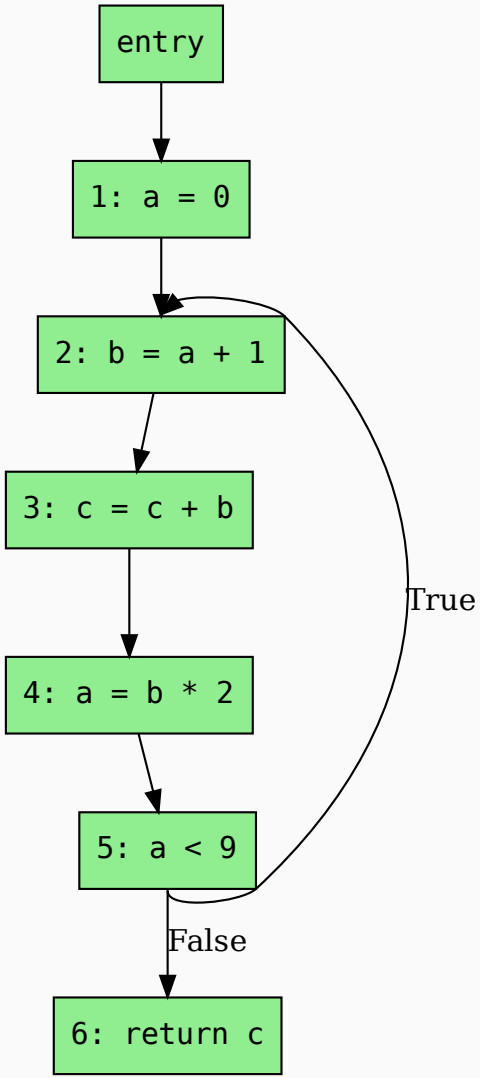
- $\text{LIVE}_{out}(4)$ is used to compute $\text{LIVE}_{in}(4)$
- $\text{LIVE}_{in}(4)$ is used to compute $\text{LIVE}_{out}(3)$

The algorithm would converge faster if we process the nodes backwards.

Backward Liveness Analysis

1: **for all** node n $\in$ CFG **do**
2: $\quad \text{LIVE}_{in}(n) = \varnothing$
3: $\quad \text{LIVE}_{out}(n) = \varnothing$
4: **end for**
5: **repeat**
6: $\quad$ **for all** node n $\in$ CFG **in "reverse DFS visited order"** do
7: $\quad\quad \text{LIVE}'_{in}(n) = \text{LIVE}_{in}(n)$
8: $\quad\quad \text{LIVE}'_{out}(n) = \text{LIVE}_{out}(n)$
9: $\quad\quad \text{LIVE}_{out}(n) = \bigcup_{\forall s \in succ(n)} \text{LIVE}_{in}(s)$
10: $\quad\quad \text{LIVE}_{in}(n) = use(n) \cup (\text{LIVE}_{out}(n) - def(n))$
11: $\quad$ **end for**
12: **until** $\text{LIVE}'_{in}(n) = \text{LIVE}_{in}(n) \wedge \text{LIVE}'_{out}(n) = \text{LIVE}_{out}(n) \forall n$

Example with Backward Liveness Analysis



node	use	def	1st		2nd		3rd	
			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

Converges in only 3 iterations!

Data-flow equations

$$\text{LIVE}_{out}(n) = \bigcup_{\forall s \in succ(n)} \text{LIVE}_{in}(s)$$

$$\text{LIVE}_{in}(n) = use(n) \cup (\text{LIVE}_{out}(n) - def(n))$$

More performance considerations

Basic Block

A straight sequence of assembly instruction which (usually) finishes with a branch/jump instruction.

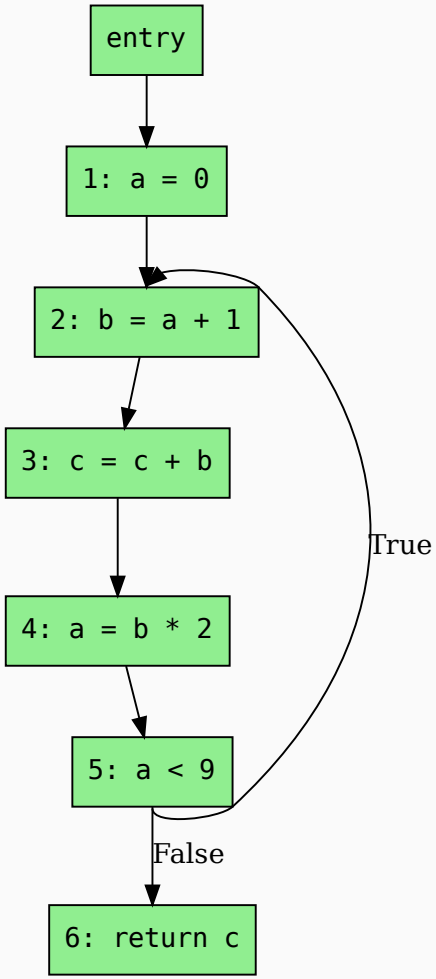
Key property: Either *all* the instructions in the sequence execute or none execute.

Can significantly decrease the size that a CFG occupies in memory by grouping nodes that have a single predecessor and a single successor into basic blocks.

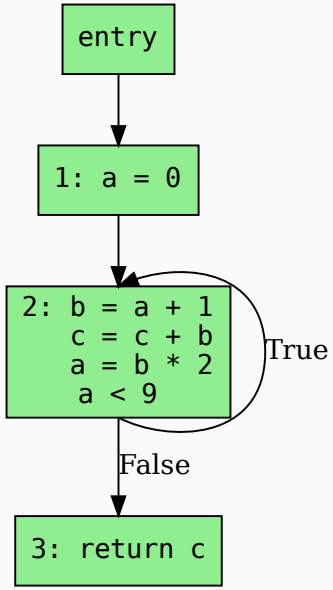
The instructions in a basic block can be simply represented as a list (rather than a graph).

Example

No basic blocks:



With basic blocks:



Next lecture

- Proper register allocation