## Data-flow Analysis

**Idea**

- **Data-flow analysis** derives information about the **dynamic** behavior of a program by only examining the **static** code

**Example**

- How many registers do we need for the program on the right?
- Easy bound: the number of variables used (3)
- Better answer is found by considering the **dynamic** requirements of the program

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



## Liveness Analysis

**Definition**

- A variable is **live** at a particular point in the program if its value at that point will be used in the future (**dead**, otherwise).

  ∴ To compute liveness at a given point, we need to look into the future

**Motivation: Register Allocation**

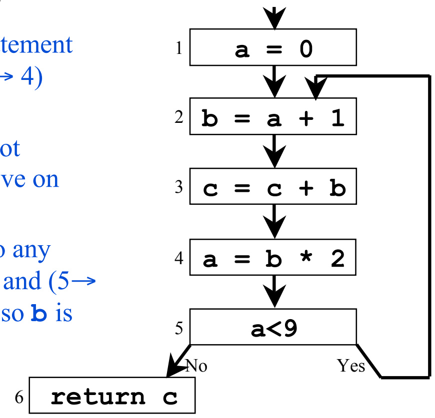
- A program contains an unbounded number of variables
- Must execute on a machine with a bounded number of registers
- Two variables can use the same register if they are never in use at the same time (*i.e,* never simultaneously live).

  ∴ Register allocation uses liveness information

## Liveness by Example

**What is the live range of b?**

- Variable **b** is read in statement 4, so **b** is live on the (3 → 4) edge
- Since statement 3 does not assign into **b**, **b** is also live on the (2→3) edge
- Statement 2 assigns **b**, so any value of **b** on the (1→2) and (5→2) edges are not needed, so **b** is dead along these edges




**b's** live range is (2→3→4)



## Liveness by Example (cont)

**Live range of a**

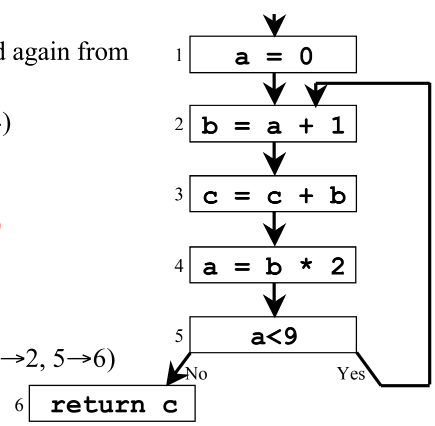
- **a** is live from (1→2) and again from (4→5→2)
- **a** is dead from (2→3→4)

**Live range of b**

- **b** is live from (2→3→4)

**Live range of c**

- **c** is live from (entry→1→2→3→4→5→2, 5→6)




Variables **a** and **b** are never simultaneously live, so they can share a register

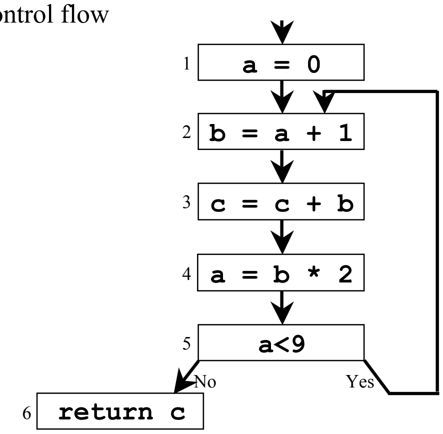
## Control Flow Graphs (CFGs)

**Definition**

– A **CFG** is a graph whose nodes represent program statements and whose directed edges represent control flow

**Example**

```
1       a := 0
2  L1:  b := a + 1
3       c := c + b
4       a := b * 2
5       if a < 9 goto L1
6       return c
```



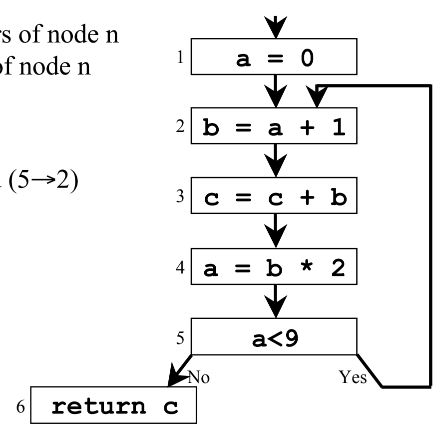



## Terminology

**Flow Graph Terms**

– A CFG node has **out-edges** that lead to **successor** nodes and **in-edges** that come from **predecessor** nodes

– **pred[n]** is the set of all predecessors of node n
**succ[n]** is the set of all successors of node n




**Examples**

– Out-edges of node 5: (5→6) and (5→2)

– succ[5] = {2,6}

– pred[5] = {4}

– pred[2] = {1,5}

## Uses and Defs

**Def (or definition)**

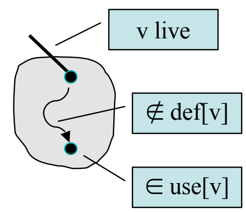
– An **assignment** of a value to a variable
– def[v] = set of CFG nodes that define variable v
– def[n] = set of variables that are defined at node n

`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 that are used at node n

`a < 9?`




**More precise definition of liveness**

– A variable v is live on a CFG edge if
(1) ∃ a directed path from that edge to a use of v (node in use[v]), and
(2) that path does not go through any def of v (no nodes in def[v])



## The Flow of Liveness

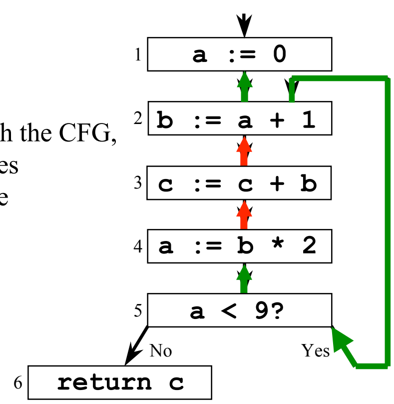
**Data-flow**

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

**Direction of Flow**

– Liveness flows **backwards** through the CFG, because the behavior at future nodes determines liveness at a given node

– Consider a
– Consider b
– Later, we'll see other properties that flow **forward**



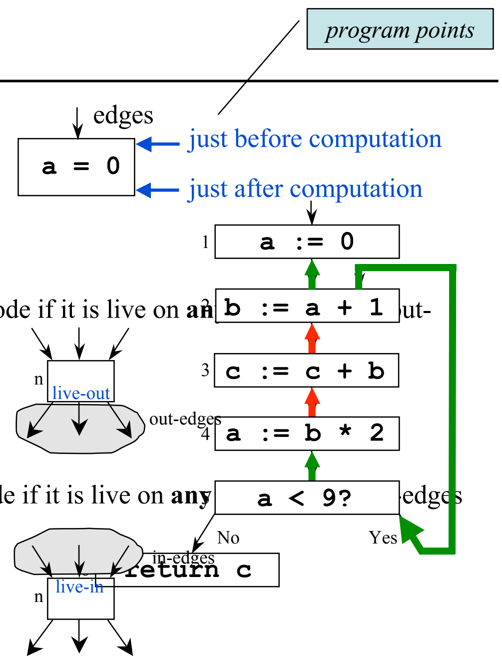
## Liveness at Nodes

**We have liveness on edges**

- How do we talk about liveness at nodes?

**Two More Definitions**

- A variable is **live-out** at a node if it is live on **any** out-edges
- A variable is **live-in** at a node if it is live on **any** in-edges

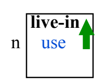




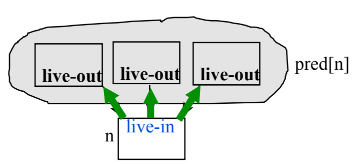

## Computing Liveness

**Rules for computing liveness**

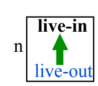
**(1)** Generate liveness:
If a variable is in use[n],
it is live-in at node n




**(2)** Push liveness across edges:
If a variable is live-in at a node n
then it is live-out at all nodes in pred[n]




**(3)** Push liveness across nodes:
If a variable is live-out at node n and not in def[n]
then the variable is also live-in at n




**Data-flow equations**

$$\text{(1)}\ \boxed{\mathbf{in[n] = use[n]}} \cup \boxed{\mathbf{(out[n] - def[n])}}\ \text{(3)}$$

$$\mathbf{out[n]} = \boxed{\bigcup_{s \in succ[n]} \mathbf{in[s]}}\ \text{(2)}$$

## Solving the Data-flow Equations

**Algorithm**

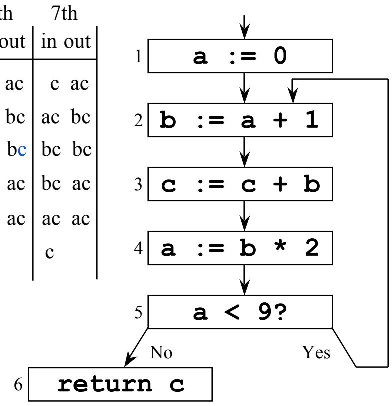
**for each** node n in CFG
    in[n] = ∅;  out[n] = ∅ — initialize solutions
**repeat**
    **for each** node n in CFG
        in'[n] = in[n]
        out'[n] = out[n] — save current results
        in[n] = use[n] ∪ (out[n] – def[n])
        out[n] = $\bigcup_{s \in succ[n]}$ in[s] — solve data-flow equations
**until** in'[n]=in[n] and out'[n]=out[n] for all n — test for convergence

This is **iterative data-flow analysis** (for liveness analysis)



## Example

| node # | use | def | 1st in | 1st out | 2nd in | 2nd out | 3rd in | 3rd out | 4th in | 4th out | 5th in | 5th out | 6th in | 6th out | 7th in | 7th 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 for Liveness**

in[n] = use[n] ∪ (out[n] – def[n])

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