Recap

Last Time

– Algorithm to compute dominators

– Did you understand it?

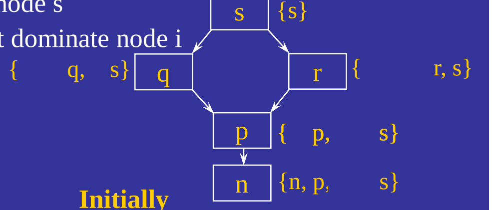
Exercise

– Can we start with the empty set and grow the set of dominators?

Computing Dominators: Example

Input: Set of nodes N and an entry node s

Output: Dom[i] = set of all nodes that dominate node i



Dom(s) = {s}

for each n $\in$ N – {s}

    Dom[n] = N

repeat

    change = false

    for each n $\in$ N – {s}

        D = {n} $\cup$ ($\cap_{p \in pred(n)}$ Dom[p])

        if D $\neq$ Dom[n]

            change = true

            Dom[n] = D

until !change

Initially

Dom[s] = {s}

Dom[q] = {n, p, q, r, s}. . .

Finally

Dom[q] = {q, s}

Dom[r] = {r, s}

Dom[p] = {p, s}

Dom[n] = {n, p, s}

Introduction to Data-flow Analysis

Last Time

– Control flow analysis

Today

– Introduce iterative data-flow analysis
  – Liveness analysis
  – Introduce other useful concepts

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 variables does this code have?

– How many registers do we need for these variables?

– Easy bound: 3

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

Data-flow Analysis

Idea

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

Example

– 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

Example

– Is b live on line 2?
– Is b live on line 4?

```
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
```

Motivation for Liveness Analysis

Register Allocation

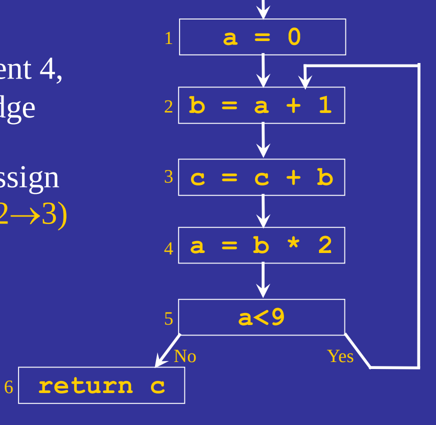
- A program contains an unbounded number of variables
- Must execute on a machine with a bounded number of registers
- Two variables can use the same register if they are never in use at the same time (*i.e*, never simultaneously live).

∴ 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)

Exercise: Liveness by Example

Live range of a

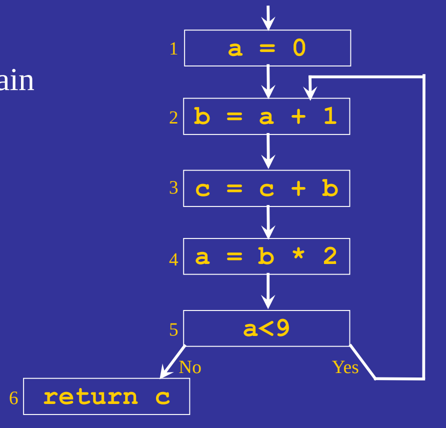
- a is live from (1→2) and again from (4→5→2)
- a is dead from (2→3→4)

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)



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

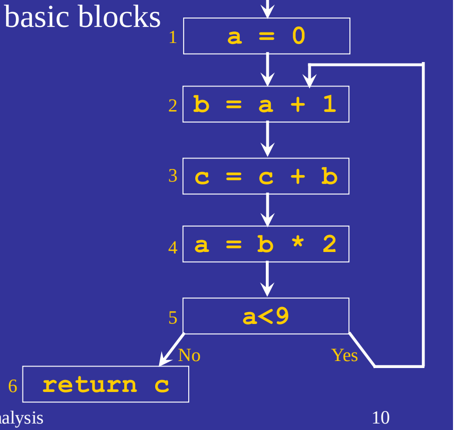
Control Flow Graphs (CFGs)

Simplification

- For now, we will use **CFGs** in which nodes represent program statements rather than basic blocks

Example

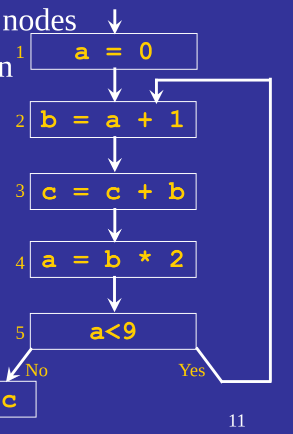
```
1           a := 0
2   L1:     b := a + 1
3           c := c + b
4           a := b * 2
5           if a < 9 goto L1
6           return c
```



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 predecessors of node n
  succ[n] is the set of 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}

Defs and Uses

Def (or definition) `a = 0`

- 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

Use `a < 9?`

- 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

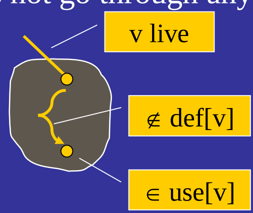
Uses and Defs (cont)

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

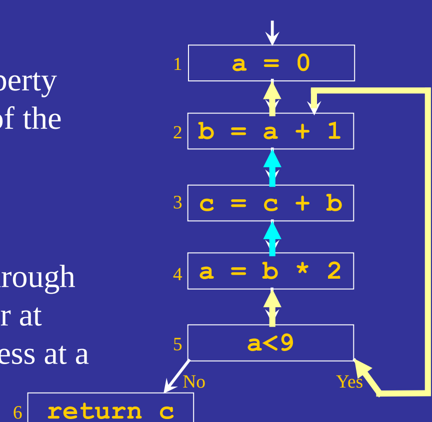
– Liveness of variables is a property that flows through the edges of the CFG

Direction of Flow

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

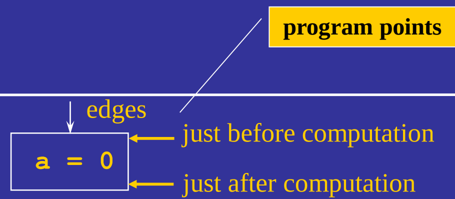
– Consider **a**

– Consider **b**



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

Liveness at Nodes

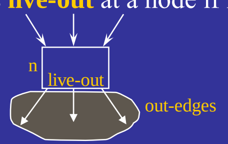


We have liveness on edges

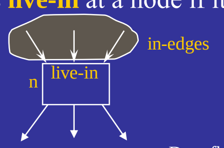
- How do we talk about liveness at nodes?

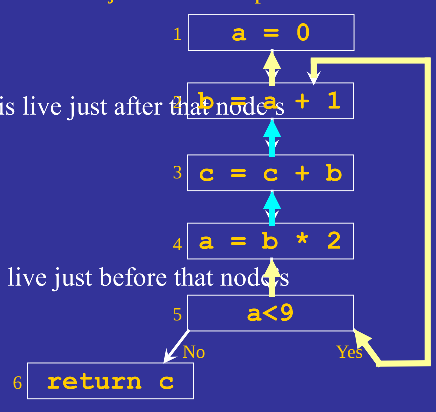
Two More Definitions

- A variable is **live-out** at a node if it is live just after that node's computation



- A variable is **live-in** at a node if it is live just before that node's computation

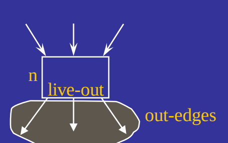




Liveness at Nodes (cont)

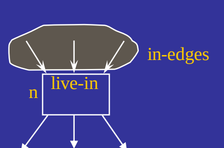
Live-out

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



Live-in

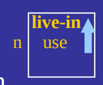
- How do we know if a variable is **live-in** at a node?



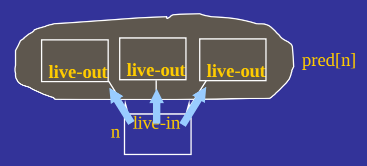
Computing Liveness

Rules for computing liveness

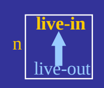
(1) Generate liveness:
If a variable is in use[n],
then it is live-in at node n



(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

(1) $in[n] = use[n] \cup (out[n] - def[n])$ (3)

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

Solving the Data-flow Equations

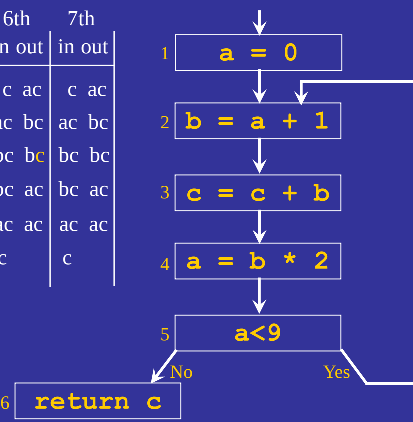
Algorithm

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

Exercise: Compute the First Iteration

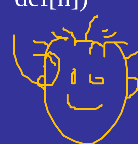
node #	use	def	1st in	1st out	2nd in	2nd out	3rd in	3rd out	4th in	4th out	5th in	5th out	6th in	6th out	7th in	7th out
1		a				a		a		ac	c	ac	c	ac	c	ac
2	a	b	a		a	bc	ac	bc	ac	bc	ac	bc	ac	bc	ac	bc
3	bc	c	bc		bc	b	bc	b	bc	b	bc	b	bc	bc	bc	bc
4	b	a	b		b	a	b	a	b	ac	bc	ac	bc	ac	bc	ac
5	a		a	a	a	ac	ac	ac	ac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



Data-flow Equations for Liveness

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

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



Example (cont)

Data-flow Equations for Liveness

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

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

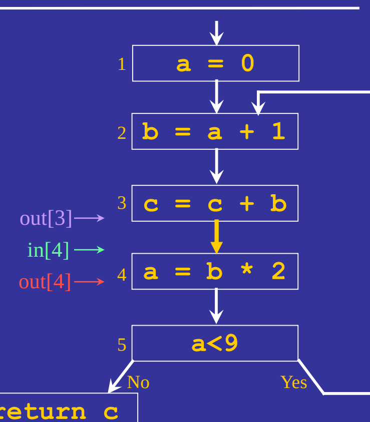
Improving Performance

Consider the **(3→4)** edge in the graph:

out[4] is used to compute in[4];

in[4] is used to compute out[3] . . .

So we should compute the sets in the order: out[4], in[4], out[3], in[3], . . .

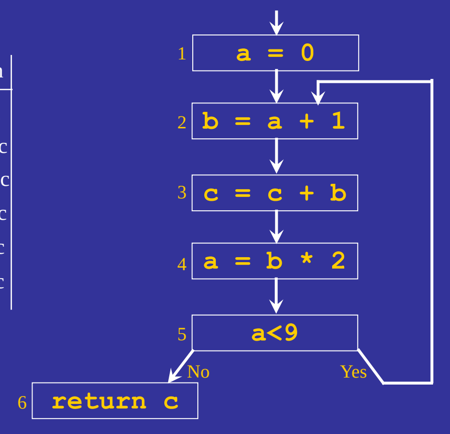


The order of computation should follow the direction of flow

Iterating Through the Flow Graph Backwards

node #	use	def	1st out	1st in	2nd out	2nd in	3rd out	3rd 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 much faster!



Solving the Data-flow Equations (reprise)

Algorithm

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

Time Complexity

Consider a program of size N

- Has N nodes in the flow graph and at most N variables
- Each live-in or live-out set has at most N elements
- Each set-union operation takes O(N) time
- The **for** loop body
  - constant # of set operations per node
  - O(N) nodes ⇒ $O(N^2)$ time for the loop
- Each iteration of the **repeat** loop can only make the set larger
- Each set can contain at most N variables ⇒ $2N^2$ iterations

Worst case: $O(N^4)$

Typical case: 2 to 3 iterations with good ordering & sparse sets ⇒ O(N) to $O(N^2)$

Concepts

Liveness

- Use in register allocation
- Generating liveness
- Flow and direction
- Data-flow equations and analysis
- Complexity
- Improving performance (basic blocks, single variable, bit sets)

Control flow graphs

- Predecessors and successors

Defs and uses

Next Time

Lecture

- Generalizing data-flow analysis

Assignment 2

- Now available
- Due February 13
- Please start early