

Introduction to Data-flow analysis

Last Time

- Implementing a Mark and Sweep GC

Today

- Control flow graphs
- Liveness analysis
- Register allocation

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

```
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
```

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

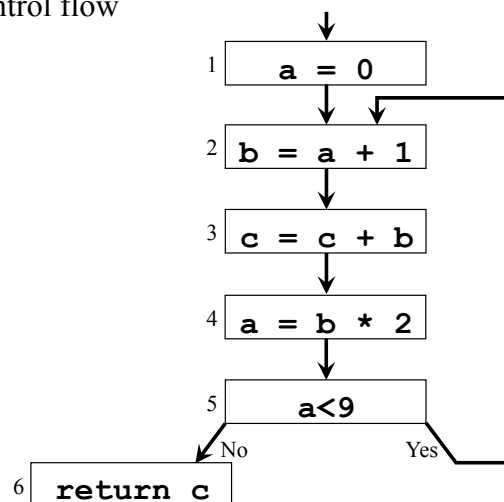
Control Flow Graphs (CFGs)

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
```



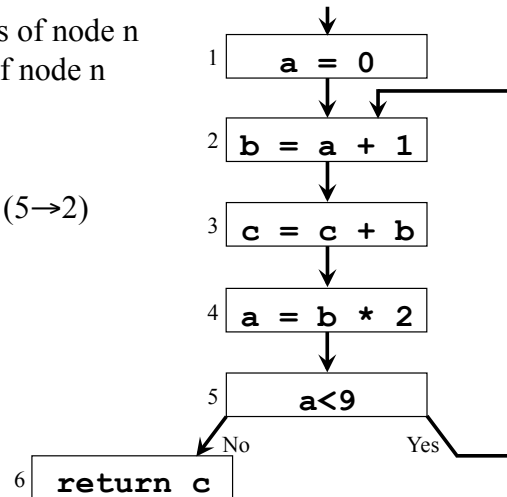
Terminology

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)
- $\text{succ}[5] = \{2,6\}$
- $\text{pred}[5] = \{4\}$
- $\text{pred}[2] = \{1,5\}$

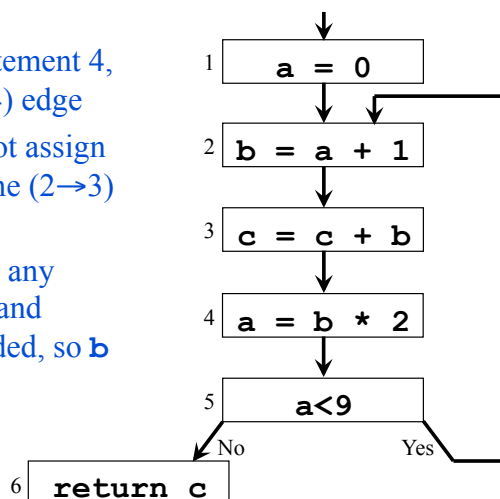


Liveness by Example

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

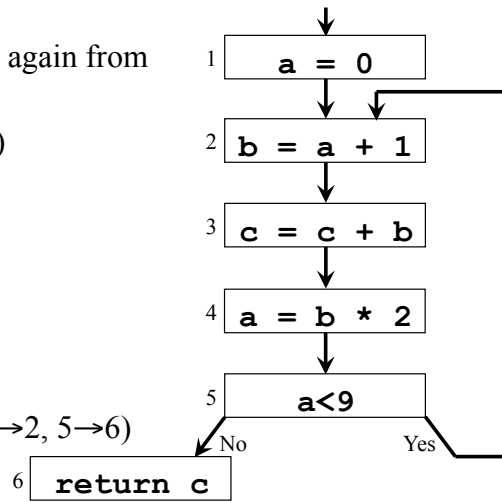
- a is live from (1→2) and again from (4→5→2)
- a is dead from (2→3→4)

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

Uses and Defs

Def (or definition)

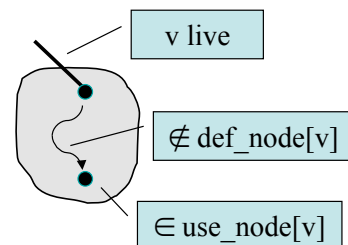
- An **assignment** of a value to a variable
- $\text{def_node}[v]$ = set of CFG nodes that define variable v
- $\text{def}[n]$ = set of variables that are defined at node n

a = 0

Use

- A **read** of a variable's value
- $\text{use_node}[v]$ = set of CFG nodes that use variable v
- $\text{use}[n]$ = 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) \exists a directed path from that edge to a use of v (node in $\text{use_node}[v]$), and
 - (2) that path does not go through any def of v (no nodes in $\text{def_node}[v]$)

The Flow of Liveness

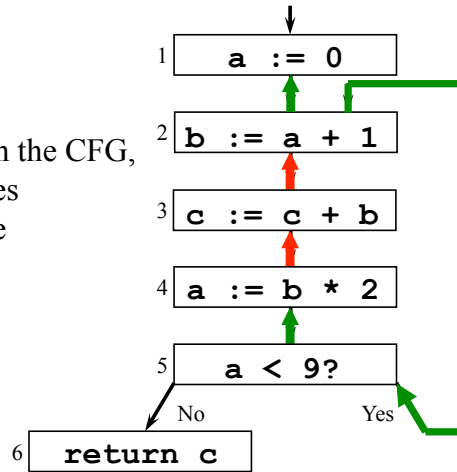
Data-flow

- 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**

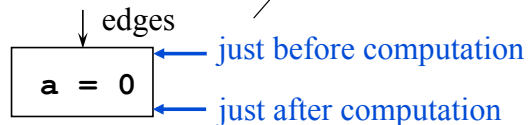


Liveness at Nodes

program points

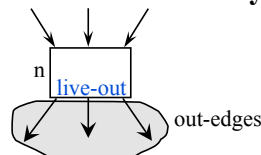
We have liveness on edges

- How do we talk about liveness at nodes?

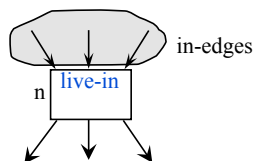


Two More Definitions

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

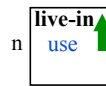


Computing Liveness

Rules for computing liveness

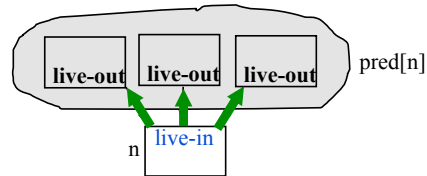
- (1) Generate liveness:

If a variable is in $use[n]$,
it is live-in at node n



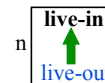
- (2) Push liveness across edges:

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

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

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

Solving the Data-flow Equations

Algorithm

```

for each node  $n$  in CFG
     $in[n] = \emptyset$ ;  $out[n] = \emptyset$ 
repeat
    for each node  $n$  in CFG
         $in'[n] = in[n]$ 
         $out'[n] = out[n]$ 
         $in[n] = use[n] \cup (out[n] - def[n])$ 
         $out[n] = \bigcup_{s \in succ[n]} in[s]$ 
    until  $in'[n]=in[n]$  and  $out'[n]=out[n]$  for all  $n$ 

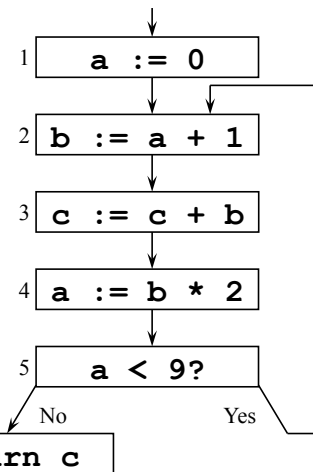
```

} initialize solutions
 } save current results
 } solve data-flow equations
 } test for convergence

This is **iterative data-flow analysis** (for 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 for Liveness

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

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

Liveness Analysis in the MiniJava compiler

Currently ...

- Parse into AST
- Allocate space on stack for locals and parameters and space in heap for member variables
- Use stack for expression evaluation
- Generate MIPS code from AST

To perform data-flow analysis ...

- Need intermediate representation like 3-address code
- Use temporaries for parameters, locals, and expression results
- Indicate uses and defs of temporaries in each 3-address code instruction
- Create a control-flow graph with each 3-address code instruction as a node

Register Allocation

Problem

- Assign an unbounded number of **symbolic** registers to a fixed number of **architectural** registers
- Simultaneously live data must be assigned to different architectural registers

Goal

- Minimize overhead of accessing data
 - Memory operations (loads & stores)
 - Register moves

Scope of Register Allocation

Expression

Local

Loop



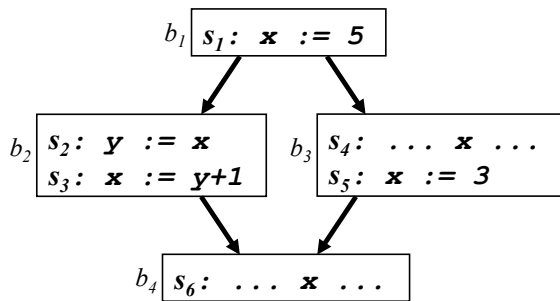
Global

Interprocedural

Granularity of Allocation

What is allocated to registers?

- Variables
- Live ranges/Webs (*i.e.*, du-chains with common uses)
- Values (*i.e.*, definitions; same as variables with SSA)



Variables: 2 (x & y)

Live Ranges/Webs: 3 ($s_1 \rightarrow s_2, s_4$;

$s_2 \rightarrow s_3$;

$s_3, s_5 \rightarrow s_6$)

Values: 4 ($s_1, s_2, s_3, s_5, \phi(s_3, s_5)$)

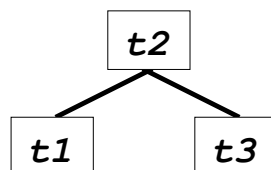
What are the tradeoffs?

Each allocation unit is given a symbolic register name (*e.g.*, t_1, t_2 , etc.)

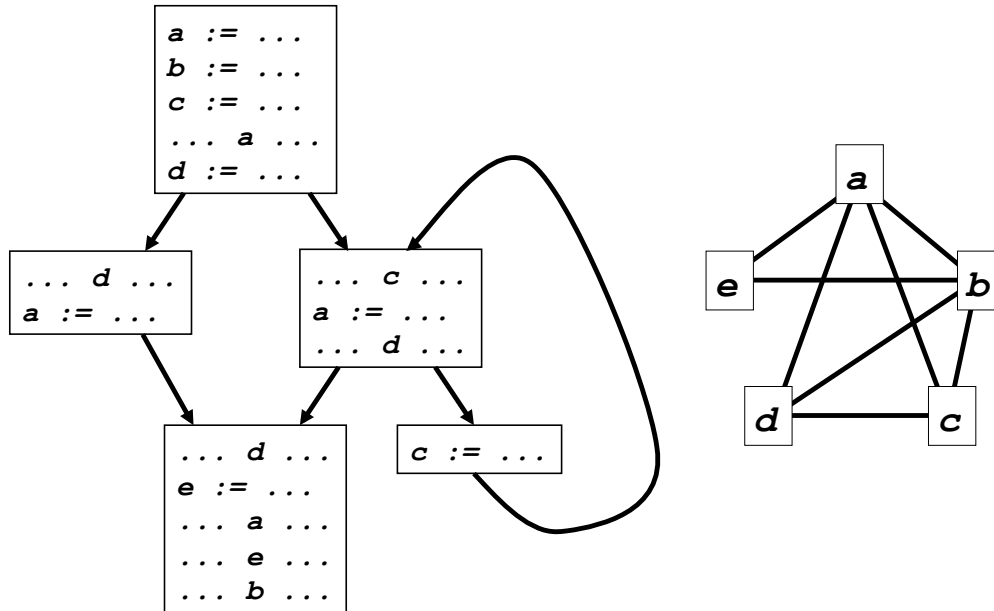
Global Register Allocation by Graph Coloring

Idea [Cocke 71], First allocator [Chaitin 81]

1. Construct **interference graph** $G=(N,E)$
 - Represents notion of “simultaneously live”
 - Nodes are units of allocation (*e.g.*, variables, live ranges, values)
 - \exists edge $(n_1, n_2) \in E$ if n_1 and n_2 are simultaneously live
 - Symmetric (not reflexive nor transitive)
2. Find **k -coloring** of G (for k registers)
 - Adjacent nodes can't have same color
3. **Allocate** the same register to all allocation units of the same color
 - Adjacent nodes must be allocated to distinct registers



Interference Graph Example (Variables)



Computing the Interference Graph

Use results of live variable analysis

```
for each symbolic-register/temporary  $t_i$  do
  for each symbolic-register/temporary  $t_j$  ( $j < i$ ) do
    for each  $def \in \{\text{definitions of } t_i\}$  do
      if ( $t_j$  is live out at  $def$ ) then
         $E \leftarrow E \cup (t_i, t_j)$ 
```

Options

- treat all instructions the same
- treat MOVE instructions special
- which is better?

Allocating Registers Using the Interference Graph

K-coloring

- Color graph nodes using up to k colors
- Adjacent nodes must have different colors

Allocating to k registers \equiv finding a k -coloring of the interference graph

- Adjacent nodes must be allocated to distinct registers

But...

- Optimal graph coloring is NP-complete
 - Optimal register allocation is NP-complete, too (must approximate)
- What if we can't k -color a graph? (must **spill**)

Register Allocation: Spilling

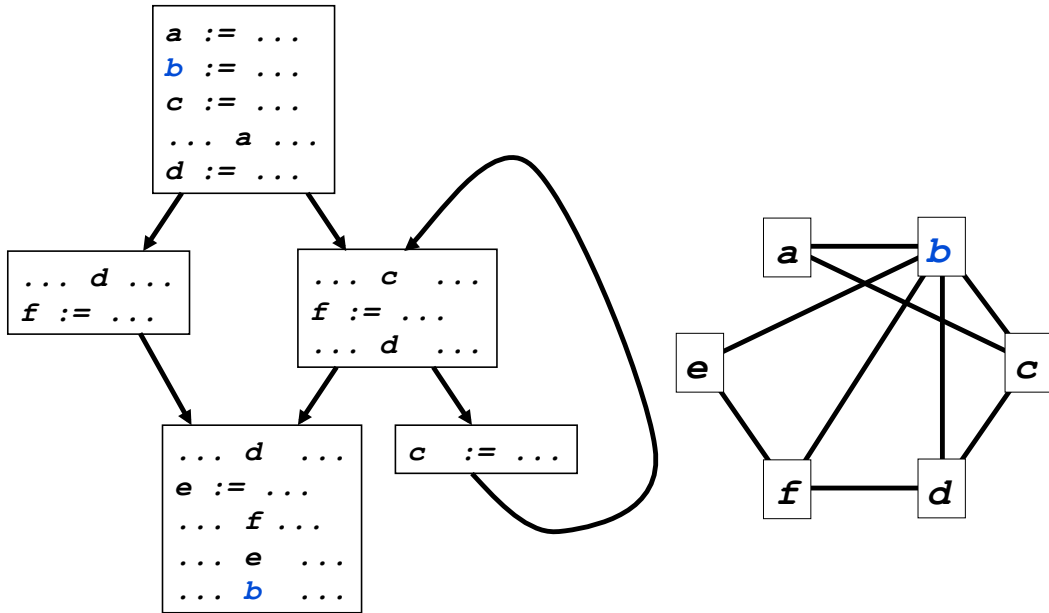
If we can't find a k -coloring of the interference graph

- Spill variables (nodes) until the graph is colorable

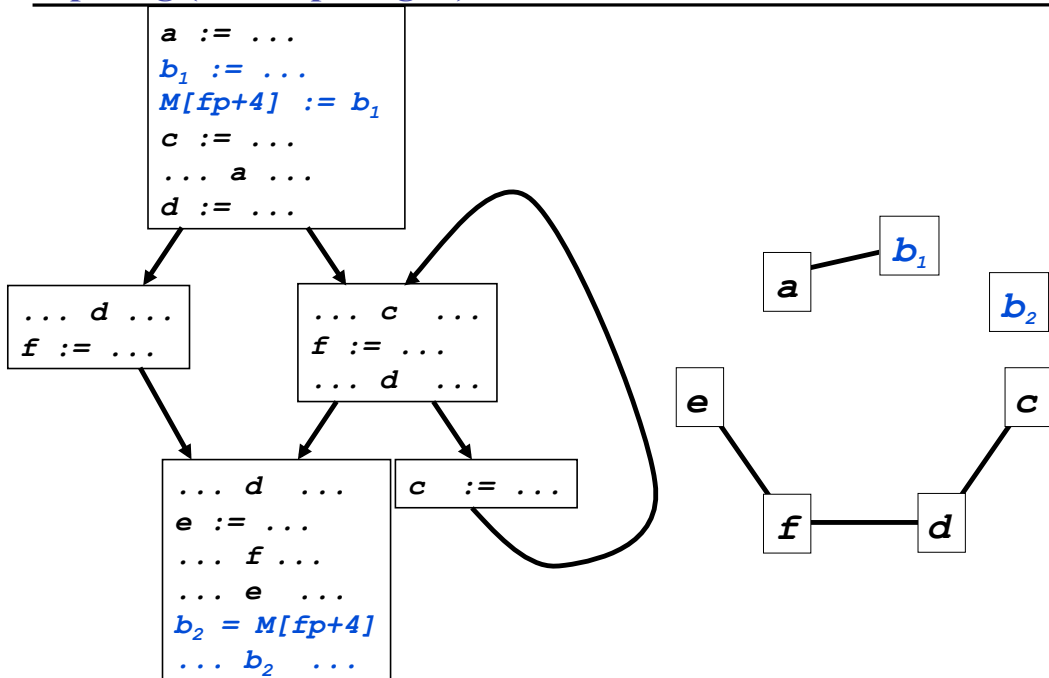
Choosing variables to spill

- Choose arbitrarily or
- Choose least frequently accessed variables
- Break ties by choosing nodes with the most conflicts in the interference graph
- Yes, these are heuristics!

Spilling (Original CFG and Interference Graph)



Spilling (After spilling b)



Simple Greedy Algorithm for Register Allocation

```

for each  $n \in N$  do           { select  $n$  in decreasing order of weight }
  if  $n$  can be colored then
    do it                         { reserve a register for  $n$  }
  else
    Remove  $n$  (and its edges) from graph { allocate  $n$  to stack (spill) }
  
```

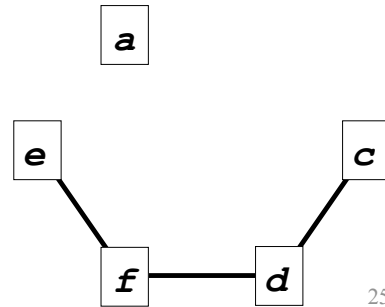
```

a := ...
r1 := ...
M[fp+4] := r1
c := ...
... a ...
d := ...
  
```

(After spilling b)

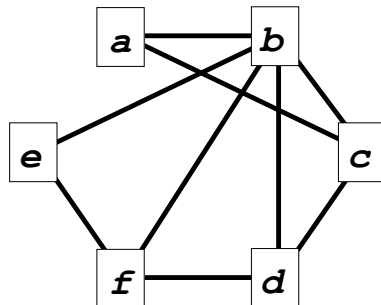
```

... d ...
e := ...
... f ...
... e ...
r2 = M[fp+4]
... r2 ...
  
```



Example

Attempt to 3-color this graph (, ,)



Weighted order:

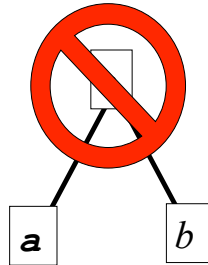
```

a
b
c
d
f
e
  
```

What if you use a different order?

Example

Attempt to 2-color this graph ( , )



Weighted order:

a
b
c

Concepts

Liveness

- Used in register allocation
- Generating liveness
- Flow and direction
- Data-flow equations and analysis

Control flow graphs

Register allocation

- scope of allocation
- granularity: what is being allocated to a register
- order that allocation units are visited in matters in all heuristic algorithms

Global approach: greedy coloring

Next Time

Reading

- Ch. 8.4, 9.2-9.25, intro to data-flow analysis
- Ch 8.8 and Briggs paper, register allocation

Lecture

- Improvements to graph coloring register allocators
- Register allocation across procedure calls

Suggested Exercises

- See schedule on website