RUTGERS

THE STATE UNIVERSITY OF NEW JERSEY

CS415 Compilers

Data Flow Analysis

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University

Roadmap

- Project 2: Due tomorrow
  Up to 3 days late, with 20% penalty each day
- Project 3: Will be posted this Thursday
- Homework 9: Will be posted this Thursday too

Data-flow Analysis

Definition

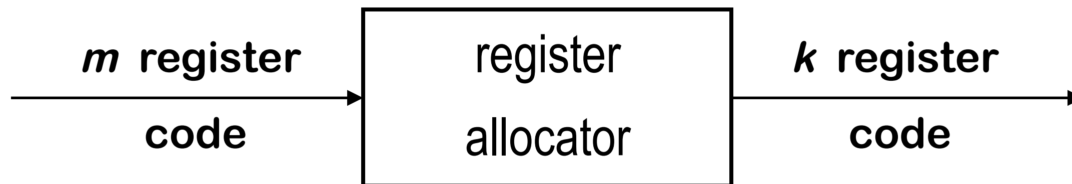
Data-flow analysis is a collection of techniques for *compile-time* reasoning about the *run-time* flow of values

- We use the results of DFA to prove safety & identify opportunities
- Almost always involves building a graph
  - → Control-flow graph, call graph, or derivatives thereof
  - → Sparse evaluation graphs to model flow of values *(efficiency)*
- Desired result is usually *meet over all paths* solution
  - → "What is true on every path from the entry?"
  - → "Can this happen on any path from the entry?"

Motivation : Global Register Allocation

At each point in the code

- Decide which values reside in registers
- Assign a particular register to each of those values



Optimal global register allocation is NP-complete

Global Register Allocation

We talked about local register allocation

- **Live ranges**
- **Top-down: rank live ranges and allocate based on priority**
- **Bottom-up: allocate on-the-fly and spill based on priority**

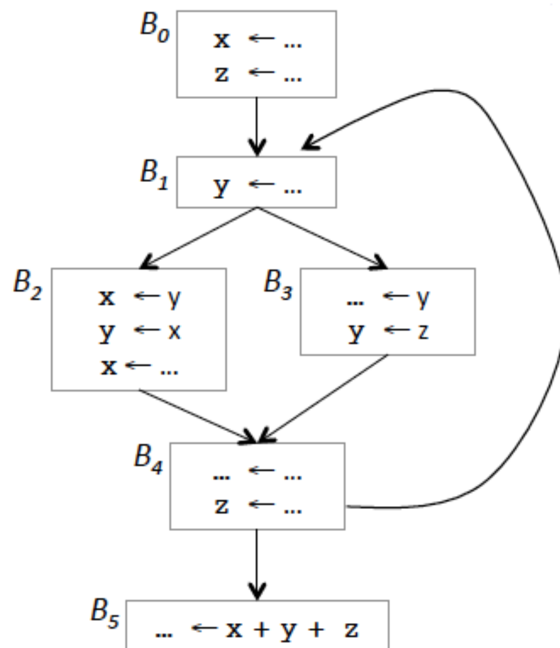
```
1  loadI   1028     ⇒ r1    // r1
2  load    r1       ⇒ r2    // r1 r2
3  mult    r1, r2   ⇒ r3    // r1 r2 r3
4  loadI   5        ⇒ r4    // r1 r2 r3 r4
5  sub     r4, r2   ⇒ r5    // r1    r3    r5
6  loadI   8        ⇒ r6    // r1    r3    r5 r6
7  mult    r5, r6   ⇒ r7    // r1    r3          r7
8  sub     r7, r3   ⇒ r8    // r1                   r8
9  store   r8       ⇒ r1    //
```

NOTE: live ranges based on live-on-exit information

Global Register Allocation

Global register allocation requires

- Global analysis, global information
- Local spill code insertion

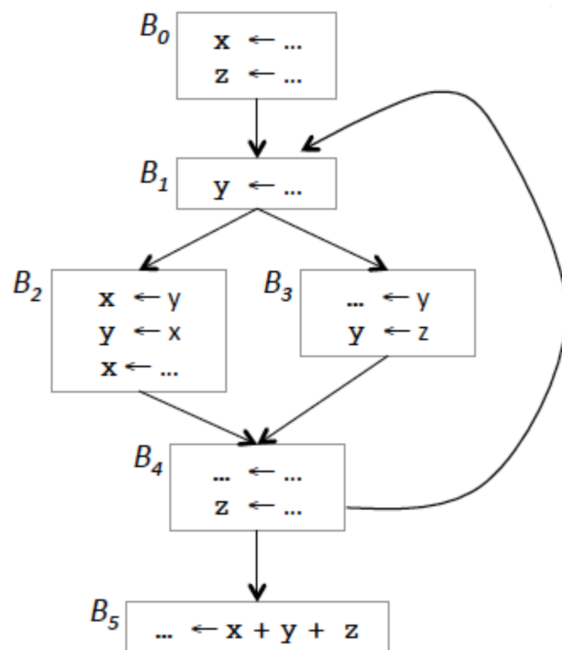


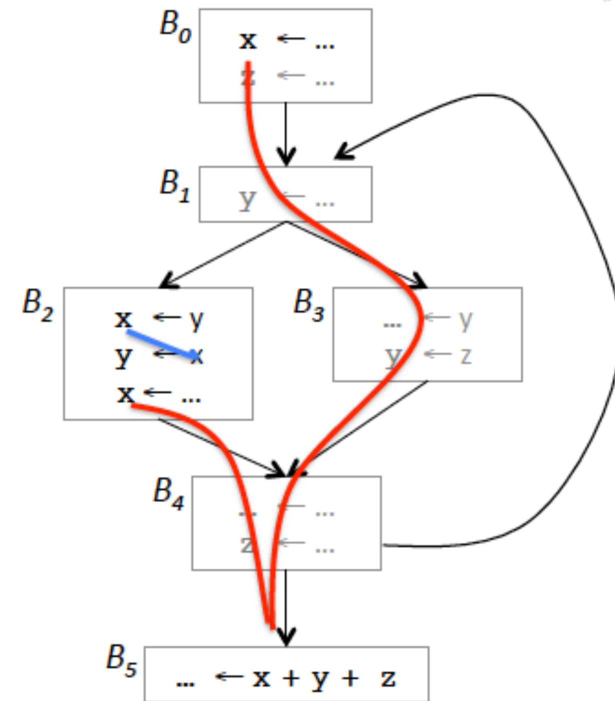
What is the live range of x, y and z?

Global Register Allocation

Global register allocation requires

- **Global analysis, global information**
- **Local spill code insertion**





RUTGERS

Global Register Allocation

Graph coloring based register allocation

- Interference rule:

> "interference": two variables interfere if there exists an operation where both are simultaneously live

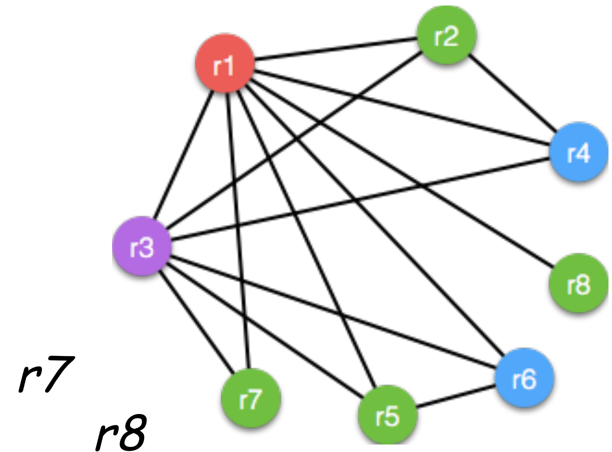
- Interference graph
  - → Nodes represent live ranges (variables)
  - → Edges represent interference between two live ranges

- Coloring problem
  - → Constraint: two incident nodes cannot have the same color
  - → A k-coloring of the interference graph can be mapped to an allocation of k registers

Global Register Allocation

Graph coloring based register allocation

```
1  loadI   1028    ⇒ r1   // r1
2  load    r1      ⇒ r2   // r1 r2
3  mult    r1, r2  ⇒ r3   // r1 r2 r3
4  loadI   5       ⇒ r4   // r1 r2 r3 r4
5  sub     r4, r2  ⇒ r5   // r1    r3    r5
6  loadI   8       ⇒ r6   // r1    r3    r5 r6
7  mult    r5, r6  ⇒ r7   // r1    r3          r7
8  sub     r7, r3  ⇒ r8   // r1                   r8
9  store   r8      ⇒ r1   //
```



- **Coloring problem**
  - → **Constraint: two incident nodes cannot have the same color**
  - → **A k-coloring of the interference graph can be mapped to an allocation of k registers**

Global Register Allocation

- **A classical stack based coloring approach (Chaitin's algorithm)**

Observation:

Assuming a k-coloring problem, any vertex n that has fewer than k neighbors in the interference graph can always be colored!

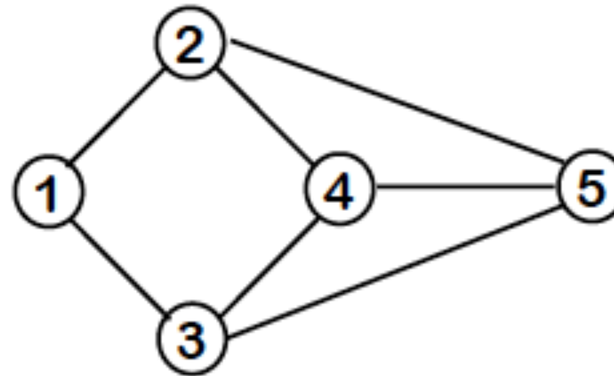
-- Pick any color not used by its neighbors — there must be one

- **Coloring problem**
  - → **Constraint: two incident nodes cannot have the same color**
  - → **A k-coloring of the interference graph can be mapped to an allocation of k registers**

Global Register Allocation

3 Registers

Stack

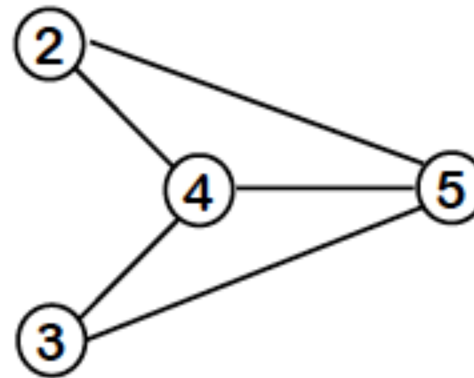


1 is the only node with degree < 3

Global Register Allocation

3 Registers

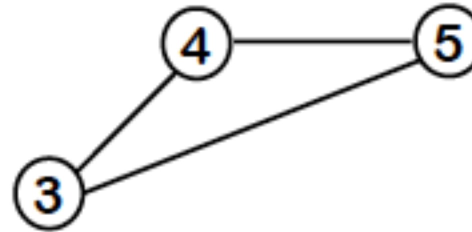
Stack



Now, 2 & 3 have degree < 3

Global Register Allocation

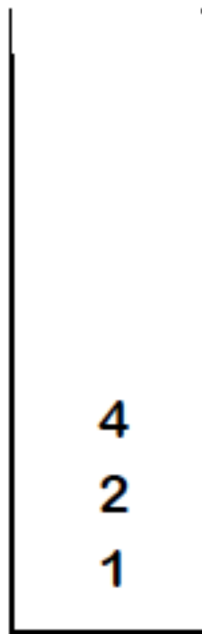
3 Registers



Now all nodes have degree < 3

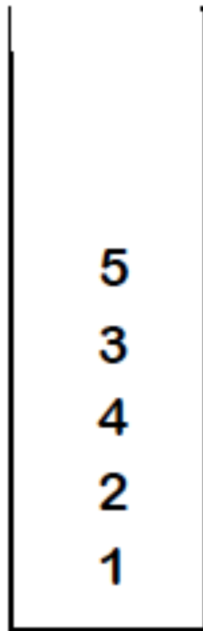
Global Register Allocation

3 Registers



Global Register Allocation

3 Registers



Stack

Colors:

1:

2:

3:

Global Register Allocation

3 Registers

Stack

Colors:

1:

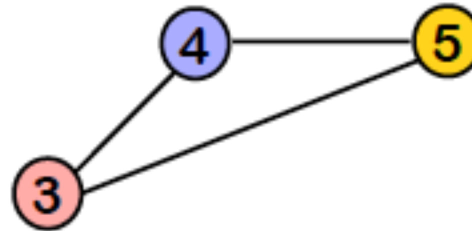
2:

3:

Global Register Allocation

3 Registers

Stack



Colors:

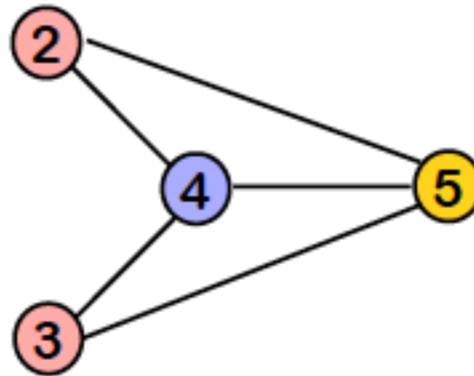
1:

2:

3:

Global Register Allocation

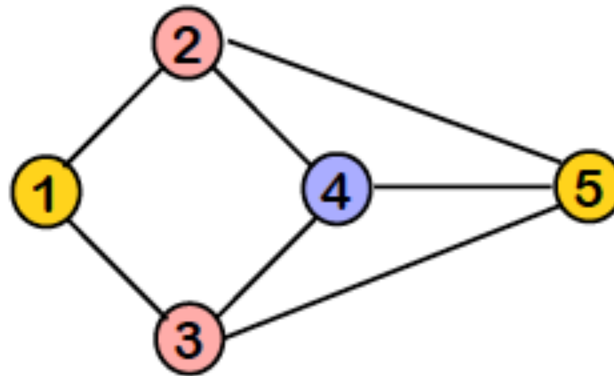
3 Registers



Colors:

Global Register Allocation

3 Registers



Colors:

1:

2:

3:

Global Register Allocation

- **A classical stack based coloring approach (Chaitin's algorithm)**

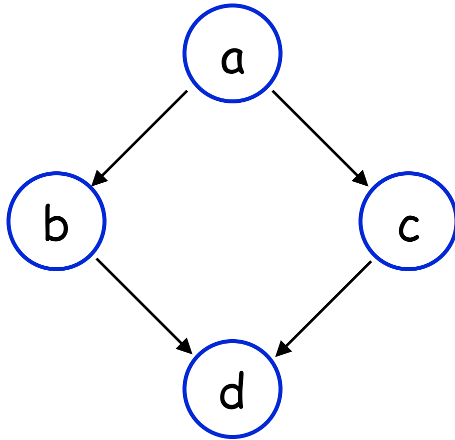
Observation:

Assuming a k-coloring problem, any vertex n that has fewer than k neighbors in the interference graph can always be colored!

-- Pick any color not used by its neighbors — there must be one

- **Coloring algorithm**
  - → Step 1: pick any vertex n such that deg(n) < k and put it on the stack
  - → Step 2: Remove that vertex and all its incident edges from the interference graph
  - → If there does not exist such vertex, pick one vertex m and spill, remove the vertex m and all its incident edges from the graph. Repeat until we can go back to step 1.

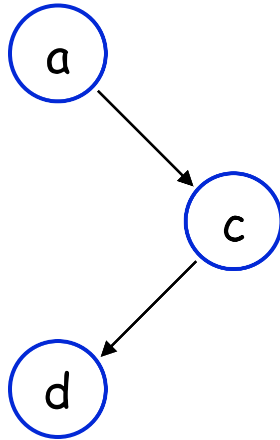
Optimistic Coloring Example



Chaitin's method

- No node has ° < 2
- Pick a node to spill, say b

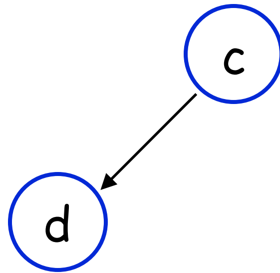
Optimistic Coloring Example



Chaitin's method

- No node has ° < 2
- Pick a node to spill, say b
- Remove a

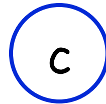
Optimistic Coloring Example



Chaitin's method

- No node has ° < 2
- Pick a node to spill, say b
- Removes a, then d

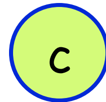
Optimistic Coloring Example



Chaitin's method

- **No node has ° < 2**
- **Pick a node to spill, say b**
- **Removes a, then d, then c**

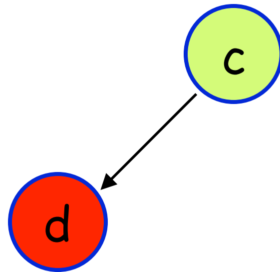
Optimistic Coloring Example



Chaitin's method

- No node has ° < 2
- Pick a node to spill, say b
- Removes a, then d, then c
- Reinsert & color c

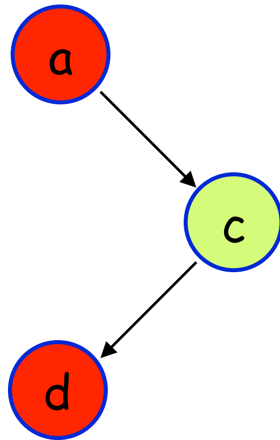
Optimistic Coloring Example



Chaitin's method

- No node has ° < 2
- Pick a node to spill, say b
- Removes a, then d, then c
- Reinsert & color c, d

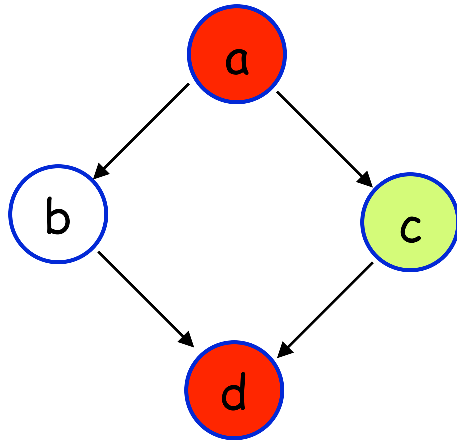
Optimistic Coloring Example



Chaitin's method

- No node has ° < 2
- Pick a node to spill, say b
- Removes a, then d, then c
- Reinsert & color c, d, and a

Optimistic Coloring Example

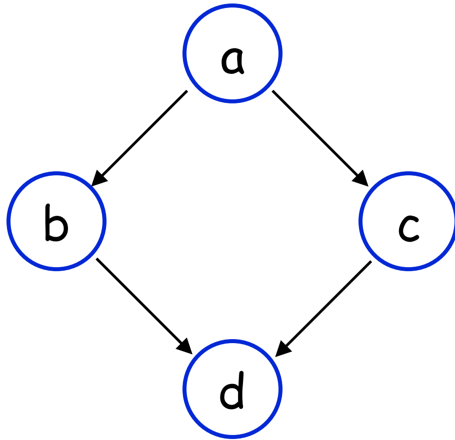


Chaitin's method

- No node has ° < 2
- Pick a node to spill, say b
- Removes a, then d, then c
- Reinsert & color c, d, and a

A color exists for b, but it has already been spilled !

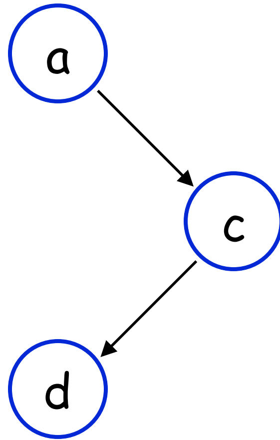
Optimistic Coloring Example



Briggs' method

- **No node has ° < 2**
- **Pick a node to spill, say b**

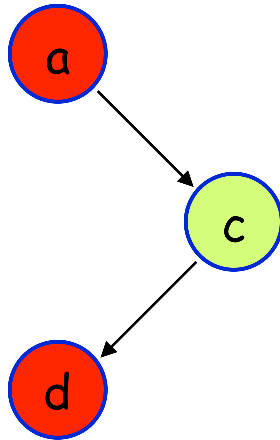
Optimistic Coloring Example



Briggs' method

- No node has ° < 2
- Pick a node to spill, say b
- Remove it & push it on the stack
- Now, proceed as with Chaitin
  > Removes a, d, c
  > Colors a, d, c

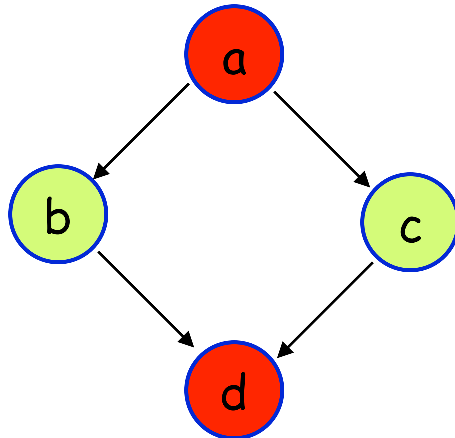
Optimistic Coloring Example



Briggs' method

- No node has ° < 2
- Pick a node to spill, say b
- Remove it & push it on the stack
- Now, proceed as with Chaitin
  - Removes a, d, c
  - Colors a, d, c
- Now, reinsert b

Optimistic Coloring Example



Briggs' method

- No node has ° < 2
- Pick a node to spill, say b
- Remove it & push it on the stack
- Now, proceed as with Chaitin
  - > Removes a, d, c
  - > Colors a, d, c
- Now, reinsert b ***& color it!***

Success !

This actually happens, in practice!

Back to data-flow Analysis

Definition

Data-flow analysis is a collection of techniques for *compile-time* reasoning about the *run-time* flow of values

- We use the results of DFA to prove safety & identify opportunities
  - → Not an end unto itself
- Almost always involves building a graph
  - → Control-flow graph, call graph, or derivatives thereof
  - → Sparse evaluation graphs to model flow of values *(efficiency)*
- Usually formulated as a set of *simultaneous equations*
  - → Sets attached to nodes and edges
  - → Often use sets with a lattice or semilattice structure
- Desired result is usually *meet over all paths* solution
  - → "What is true on every path from the entry?"
  - → "Can this happen on any path from the entry?"

Information Chains

A tuple that connects 2 data-flow events is a *chain*

event ≅ definition or use

- Chains express data-flow relationships directly
- Chains provide a graphical representation
- Chains jump across unrelated code, simplifying search

We can build chains efficiently

Four interesting types of chain

Def-Use chains are the most common

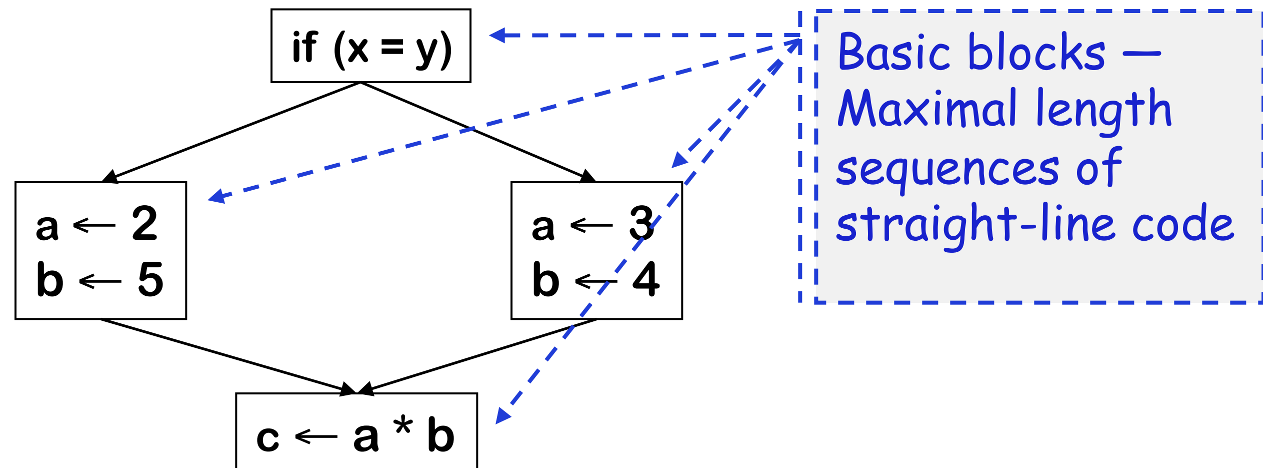
Source	Sink	Dependence Type
def	use	true, flow
use	def	anti
def	def	output
use	use	input

Review: Control-flow Graph (CFG)

Models the transfer of control in the procedure

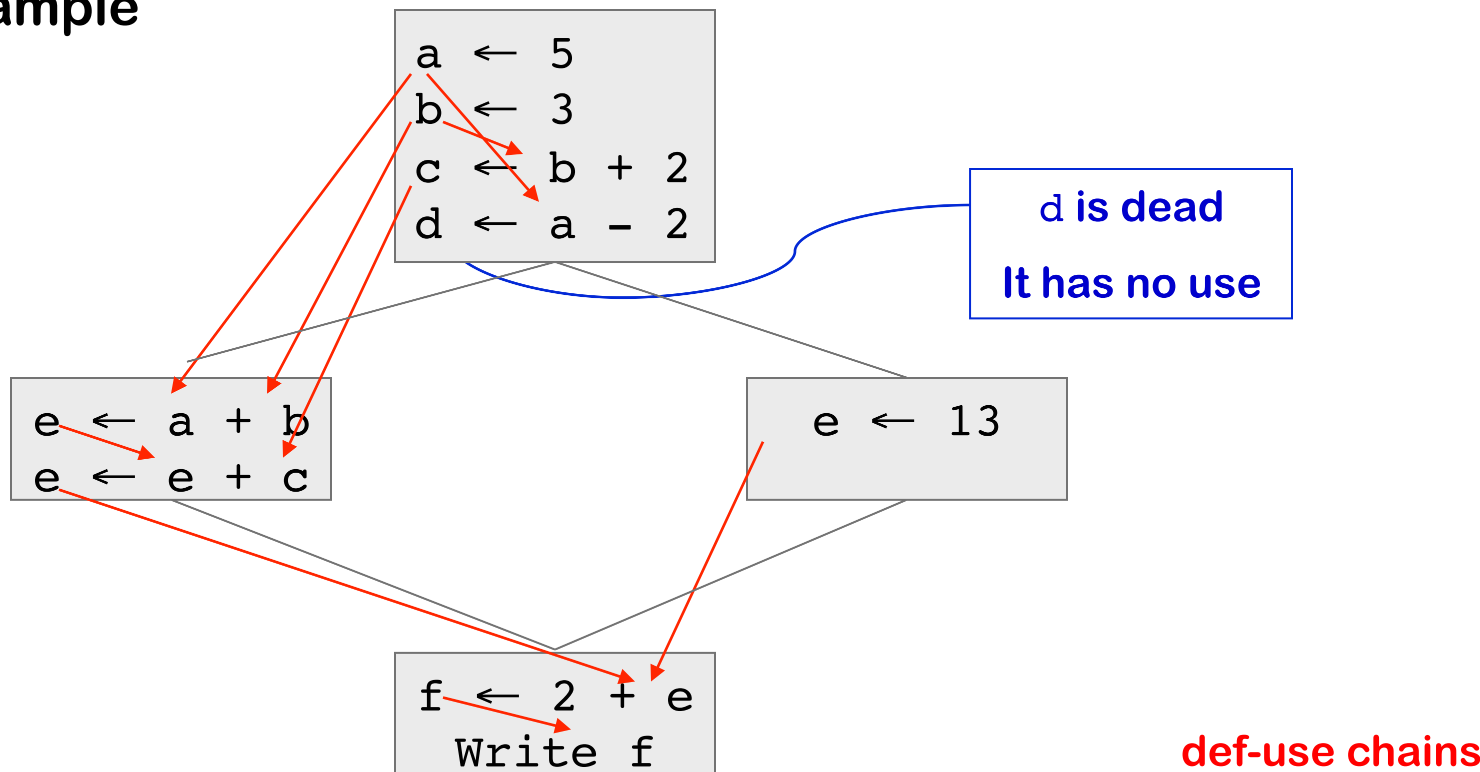
- Nodes in the graph are basic blocks
- Edges in the graph represent control flow

Example



Information Chains

Example



def-use chains

Notation

Assume that, ∀ operation *i* and each variable *v*,

- *DEFS(v,i)* is the set of operations that may have defined *v* most recently before *i*, along some path in the CFG
- *USES(v,i)* is the set of operations that may use the value of *v* computed at *i*, along some path in the CFG

$$x \in DEFS(A,y) \Leftrightarrow y \in USES(A,x)$$

To construct DEF-USE chains, we solve *reaching definitions* (*YADFP*)

- A definition *d* of some variable *v* <u>*reaches*</u> an operation *i* if and only if *i* reads *v* and there is a <u>*v-clear*</u> path from *d* to *i*
  - → *v-clear* ⇒ no definition of *v* on the path
- Prior definition of *v* in same block ⇒ $|DEFS(v,i)| = 1$
- No prior definition ⇒ $|DEFS(v,i)| \geq 1$ ← The chains are non-local in this case

Reaching Definitions

Domain is |definitions|, same as number of operations

The equations

$$REACHES(n) = \emptyset, \forall\ n \in N$$

$$REACHES(n) = \cup_{p \in preds(n)} (DEDEF(p) \cup (REACHES(p) \cap \overline{DEFKILL(p)}))$$

- *REACHES(n)* is the set of definitions that reach block *n*
- *DEDEF(N)* is the set of definitions in *n* that reach the end of *n*
- *DEFKILL(n)* is the set of defs obscured by a new def in *n*

Computing *REACHES(n)*

- Use any data-flow method

Building DEFS sets

The Plan

1. Find basic blocks & build the CFG
2. ∀ block *b*, compute *REACHES(b)* (to the fixed point)

3. ∀ block *b*, ∀ operation *i*, ∀ referenced name *v*,

   Set *DEFS(v,i)* according to the earlier rule

   A.) If there is a prior definition, *d*, of *v* in *b*

   DEFS(v,i) ← *d*

   B.) Otherwise

   DEFS(v,i) ← {*d* | *d* defines *v* & *d* ∈ *REACHES(b)*}

To build USES

- Invert *DEFS*, or
- Solve *reachable uses*, and use the analogous construction

Use use-def chain in Static Single Assignment Form (SSA)

- The main idea: each name defined exactly once in program
- Introduce ϕ-functions to make it work

Original

```
x ← ...
y ← ...
while (x < k)
    x ← x + 1
    y ← y + x
```

SSA-form

```
          x0 ← ...
          y0 ← ...
          if (x0 > k) goto next
loop:       x1 ← φ(x0,x2)
            y1 ← φ(y0,y2)
            x2 ← x1 + 1
            y2 ← y1 + x2
            if (x2 < k) goto loop
next:      ...
```

Strengths of SSA-form

- Sharper analysis
- ϕ-functions give hints about placement
- (sometimes) faster algorithms

Next class

More Dataflow Analysis and Inter-procedure Opt

Read EaC: Chapter 9.1 – 9.3, ALSU: Chapter 12