HARVARD
John A. Paulson
School of Engineering
and Applied Sciences

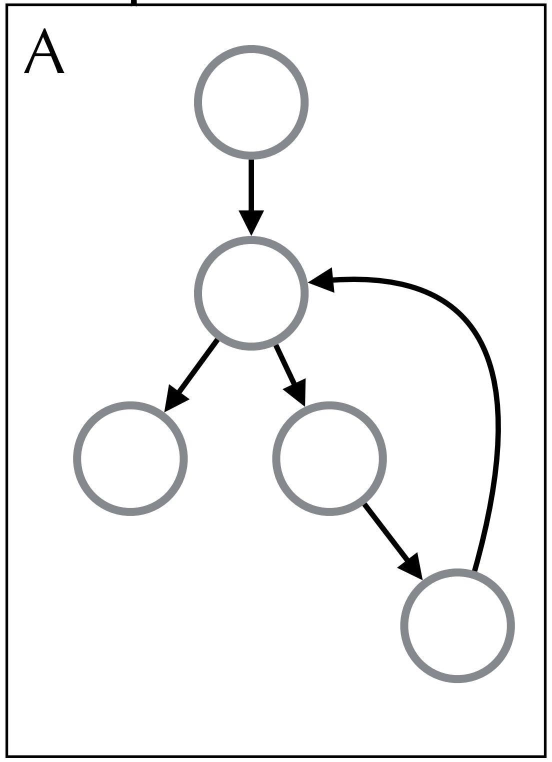
CS153: Compilers
Lecture 18: Loop Optimization I

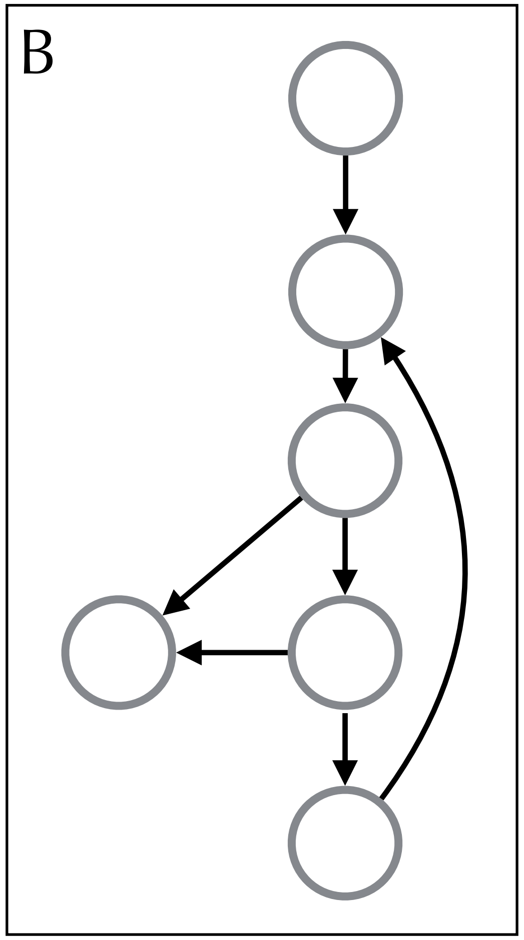
Stephen Chong

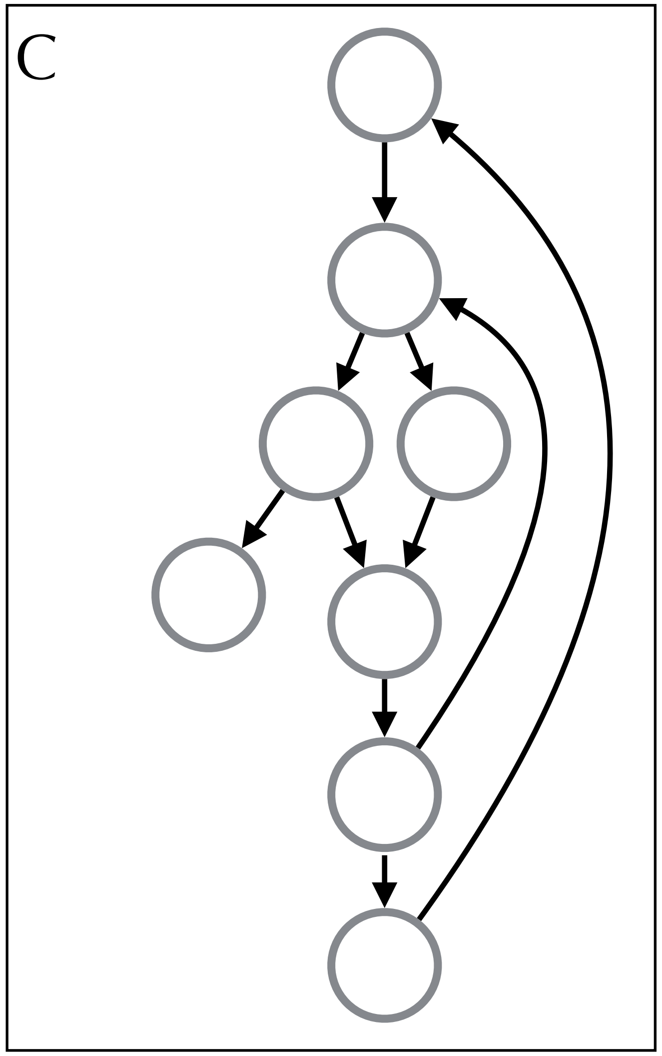
https://www.seas.harvard.edu/courses/cs153

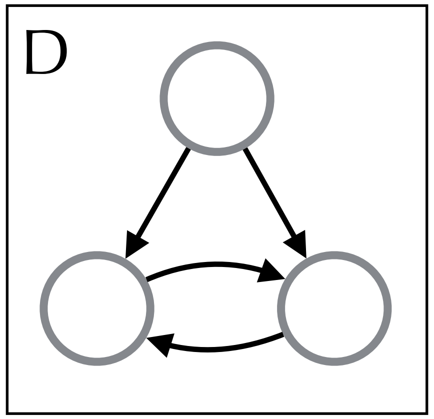
Pre-class Puzzle

- For each of these Control Flow Graphs (CFGs), what is a C program that corresponds to it?









Announcements

- Project 5 out
  - Due Tuesday Nov 13 (12 days)
- Project 6 out
  - Due Tuesday Nov 20 (19 days)
- Project 7 out
  - Due Thursday Nov 29 (28 days)

Today

- More dataflow analyses
  - Available expressions
  - Reaching definitions
  - Liveness
- Loop optimization
  - Examples
  - Identifying loops
    - Dominators

Dataflow Analysis

- Last class we saw dataflow analysis for available expressions
- An expression `e` is **available** at program point *p* if on all paths from the entry to *p*, expression `e` is computed at least once, and there are no intervening assignment to the free variables of `e` **[NOTE: last lecture's definition corrected]**
- Defined available expression analysis using gen and kill sets; combined dataflow facts at merge points by intersection

Available Expressions Analysis

Stmt	Gen	Kill
`x:=v`	{ v }	{e | x in e}
`x:=`v_1 `op` v_2	{v_1 op v_2}	{e | x in e}
`x:=*(v+i)`	{}	{e | x in e}
`*(v+i):=x`	{}	{}
`jump L`	{}	{}
`return v`	{}	{}
`if v1 op v2 goto L1 else goto L2`	{}	{}
`x:=v(`v_1`,...`v_n`)`	{}	{e | x in e}

- $D_{in}[\texttt{L}] = \cap\{D_{out}[\texttt{L'}] \mid \texttt{L'} \text{ in } \mathit{pred}[\texttt{L}]\}$
- Transfer function for stmt *S*: $\lambda D. (D \cup Gen_S) - Kill_S$

Reaching Definitions

- A definition `x:=e` **reaches** a program point *p* if there is some path from the assignment to *p* that contains no other assignment to `x`
- Reaching definitions useful in several optimizations, including constant propagation
- Can also define reaching definitions analysis using gen and kill sets; combine dataflow facts at merge points by **union**

Reaching Definitions Analysis

- Assign a unique id to each definition
- Define *defs*(`x`) to be the set of all definitions of variable `x`

Stmt	Gen	Kill
d:`x:=v`	{ d }	*defs*(`x`)–{ d }
d:`x:=v`$_1$ `op v`$_2$	{ d }	*defs*(`x`)–{ d }
everything else	∅	∅

- $D_{in}[\mathtt{L}] = \cup\{D_{out}[\mathtt{L'}] \mid \mathtt{L'} \text{ in } pred[\mathtt{L}]\}$
- Transfer function for stmt S: $\lambda D.\ (D \cup Gen_S) - Kill_S$

Liveness

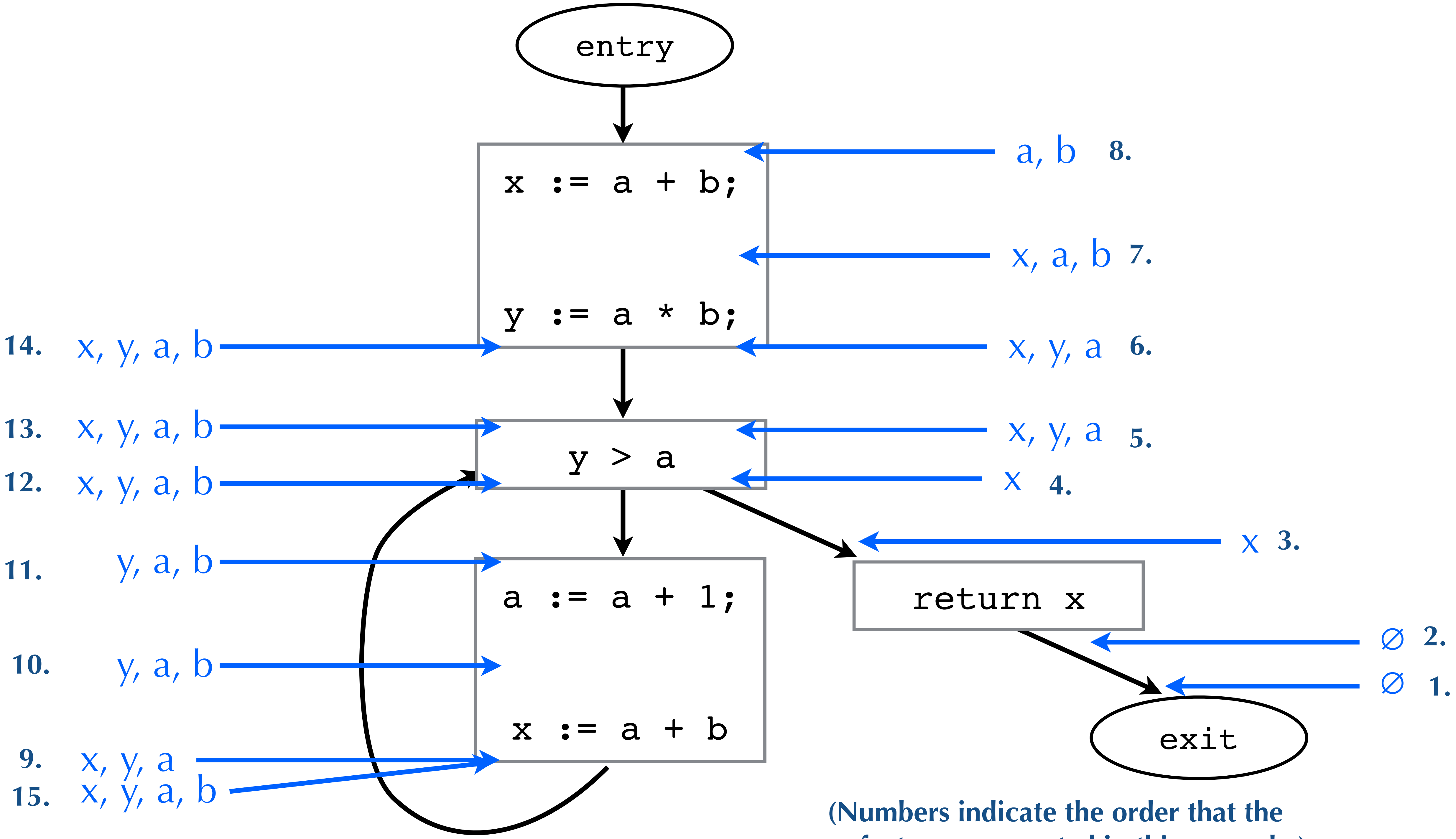
- Variable **`x`** is **live** at program point *p* is there is a path from *p* to a use of variable **`x`**
- Liveness useful in dead code elimination and register allocation
- Can also define using gen-kill sets
- However, we use a **backward dataflow analysis**
  - i.e., instead of flowing facts forwards over statement (computing D_{out} from D_{in}) we flow facts backwards over statements (compute D_{in} from D_{out})

Liveness Analysis

Stmt	**Gen**	**Kill**
`x:=v`	$\{ \texttt{v} \mid \text{if } \texttt{v} \text{ is variable}\}$	$\{ \texttt{x} \}$
$\texttt{x:=v}_1\ \texttt{op}\ \texttt{v}_2$	$\{\texttt{v}_i \mid i \in 1,2,\ \texttt{v}_i \text{ is var}\}$	$\{ \texttt{x} \}$
`x:=*(v+i)`	$\{ \texttt{v} \mid \text{if } \texttt{v} \text{ is variable}\}$	$\{ \texttt{x} \}$
`*(v+i):=x`	$\{ \texttt{x} \} \cup \{ \texttt{v} \mid \text{if } \texttt{v} \text{ is variable}\}$	{}
`jump L`	{}	{}
`return v`	$\{ \texttt{v} \mid \text{if } \texttt{v} \text{ is variable}\}$	{}
`if v1 op v2 goto L1 else goto L2`	$\{\texttt{v}_i \mid i \in 1,2,\ \texttt{v}_i \text{ is var}\}$	{}
$\texttt{x:=v}_0\texttt{(v}_1\texttt{,...v}_n\texttt{)}$	$\{\texttt{v}_i \mid i \in 0..n,\ \texttt{v}_i \text{ is var}\}$	$\{ \texttt{x} \}$

- I.e., any use of a variable generates liveness, any definition kills liveness
- $D_{\text{out}}[\texttt{L}] = \cup\{D_{\text{in}}[\texttt{L}'] \mid \texttt{L}' \text{ in } \mathit{succ}[\texttt{L}]\ \}$
- Transfer function for stmt S: $\lambda \text{D}.\ (\text{D} \cup \text{Gen}_S) - \text{Kill}_S$

Liveness Example



a, b 8.

x, a, b 7.

x, y, a 6.

x, y, a 5.

x 4.

x 3.

∅ 2.

∅ 1.

14. x, y, a, b

13. x, y, a, b

12. x, y, a, b

11. y, a, b

10. y, a, b

9. x, y, a

15. x, y, a, b

(Numbers indicate the order that the facts are computed in this example.)

Very Busy Expressions

- An expression `v1 op v2` is **very busy** at program point *p* if on every path from *p*, expression `v1 op v2` is evaluated before the value of either `v1` or `v2` is changed
- Optimization
  - Can hoist very busy expression computation
- What kind of problem?
  - Forward or backward?
  - May or must?

Space of data flow analyses

	May	**Must**
Forward	Reaching definitions	Available expressions
Backward	Live variables	Very busy expressions

- Most dataflow analyses can be categorized in this way
  - i.e., forward or backward, may or must
  - A few don't fit, need bidrectional flow
- Many dataflow analyses can be expressed as gen/kill analyses

Loop Optimizations

- Vast majority of time spent in loops
- So we want techniques to improve loops!
  - Loop invariant removal
  - Induction variable elimination
  - Loop unrolling
  - Loop fusion
  - Loop fission
  - Loop peeling
  - Loop interchange
  - Loop tiling
  - Loop parallelization
  - Software pipelining

Example 1: Invariant Removal

```
L0:   t := 0


L1:   i := i + 1
      t := a + b
      *i := t
      if i<N goto L1 else L2


L2:   x := t
```

Example 1: Invariant Removal

```
L0:  t := 0
     t := a + b

L1:  i := i + 1

     *i := t
     if i<N goto L1 else L2


L2:  x := t
```

Example 2: Induction Variable

```
L0:   i := 0
      s := 0
      jump L2
L1:   t1 := i*4
      t2 := a+t1
      t3 := *t2
      s  := s + t3
      i  := i+1
L2:   if i < 100 goto L1 else goto L3
L3:   ...
```

```
s=0;
for (i=0; i < 100; i++)
  s += a[i];
```

Example 2: Induction Variable

```
L0:  i := 0
     s := 0
     jump L2
L1:  t1 := i*4
     t2 := a+t1
     t3 := *t2
     s  := s + t3
     i  := i+1
L2:  if i < 100 goto L1 else goto L3
L3:  ...
```

`t1` is always equal to `i*4` !

Example 2: Induction Variable

```
L0:  i  := 0
     s  := 0
     t1 := 0
     jump L2
L1:  t2 := a+t1
     t3 := *t2
     s  := s + t3
     i  := i+1
     t1 := t1+4
L2:  if i < 100 goto L1 else goto L3
L3:  ...
```

`t1` is always equal to `i*4` !

Example 2: Induction Variable

```
L0:  i := 0
     s := 0
     t1 := 0
     jump L2
L1:  t2 := a+t1
     t3 := *t2
     s  := s + t3
     i  := i+1
     t1 := t1+4
L2:  if i < 100 goto L1 else goto L3
L3:  ...
```

Example 2: Induction Variable

```
L0:   i  := 0
      s  := 0
      t1  := 0
      jump L2
L1:   t2  := a+t1
      t3  := *t2
      s   := s + t3
      i   := i+1
      t1  := t1+4
L2:   if i < 100 goto L1 else goto L3
L3:   ...
```

`t2` is always equal to `a+t1` == `a+i*4` !

Example 2: Induction Variable

```
L0:  i := 0
     s := 0
     t1 := 0
     t2 := a
     jump L2
L1:  t3 := *t2
     s  := s + t3
     i  := i+1
     t2 := t2+4
     t1 := t1+4
L2:  if i < 100 goto L1 else goto L3
L3:  ...
```

`t2` is always equal to `a+t1` == `a+i*4` !

Example 2: Induction Variable

```
L0:   i := 0
      s := 0
      t1 := 0
      t2 := a
      jump L2
L1:   t3 := *t2
      s  := s + t3
      i  := i+1
      t2 := t2+4
      t1 := t1+4
L2:   if i < 100 goto L1 else goto L3
L3:   ...
```

`t1` is no longer used!

Example 2: Induction Variable

```
L0:   i := 0
      s := 0
      t2 := a
      jump L2

L1:   t3 := *t2
      s  := s + t3
      i  := i+1
      t2 := t2+4

L2:   if i < 100 goto L1 else goto L3
L3:   ...
```

Example 2: Induction Variable

```
L0:   i := 0
      s := 0
      t2 := a
      jump L2

L1:   t3 := *t2
      s  := s + t3
      i  := i+1
      t2 := t2+4

L2:   if i < 100 goto L1 else goto L3
L3:   ...
```

`i` is now used just to count 100 iterations.
But `t2` = `4*i + a`
so `i < 100`
when
`t2 < a+400`

Example 2: Induction Variable

```
L0:   i := 0
      s := 0
      t2 := a
      t5 := t2 + 400
      jump L2
L1:   t3 := *t2
      s  := s + t3
      i  := i+1
      t2 := t2+4

L2:   if t2 < t5 goto L1 else goto L3
L3:   ...
```

`i` is now used just to count 100 iterations. But `t2` = `4*i + a` so `i < 100` when `t2 < a+400`

Example 2: Induction Variable

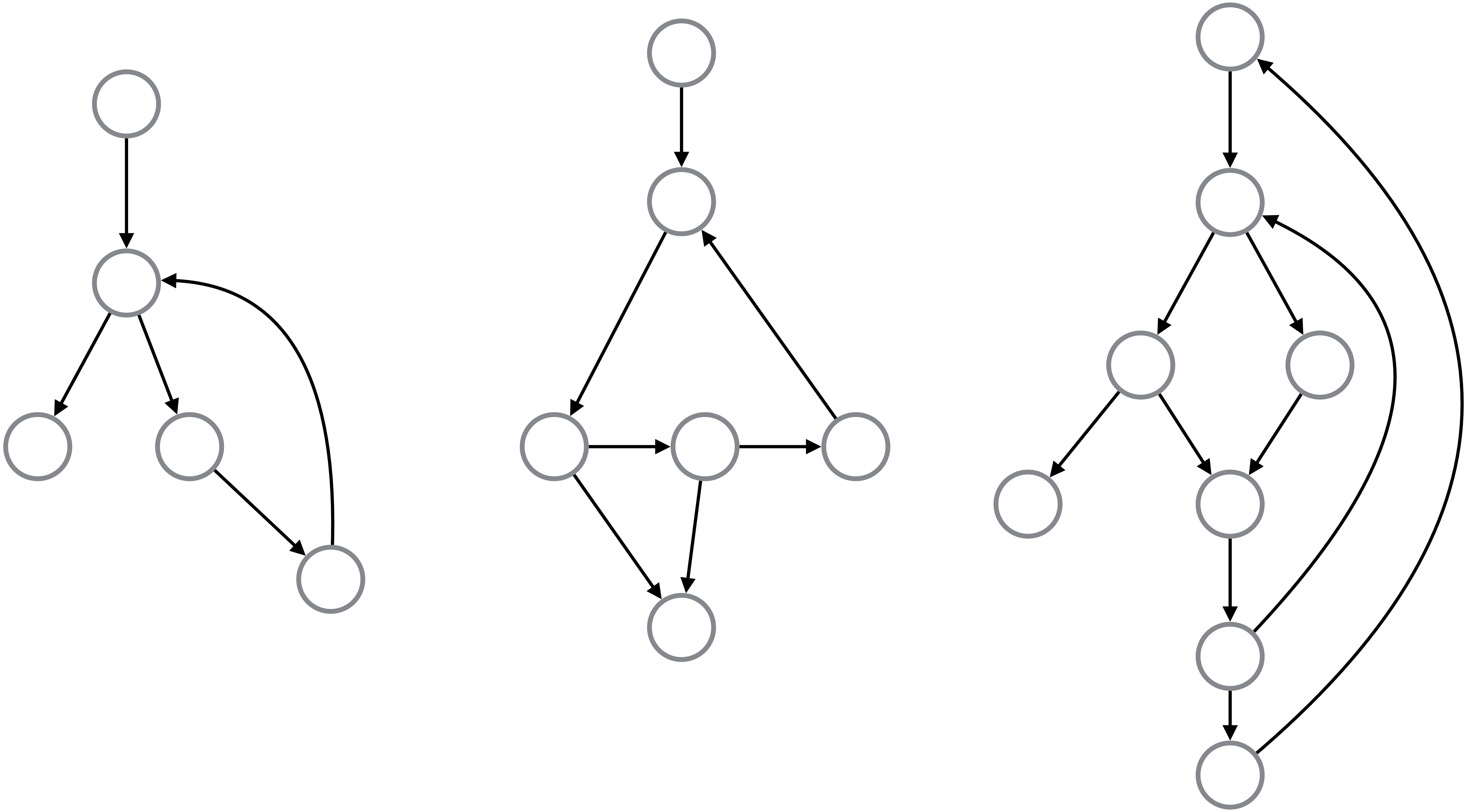
```
L0:   s := 0
      t2  := a
      t5  := t2 + 400
      jump L2

L1:   t3  := *t2
      s   := s + t3
      t2  := t2+4


L2:   if t2 < t5 goto L1 else goto L3
L3:   ...
```

`i` is now used just to count 100 iterations. But `t2` = `4*i + a` so `i < 100` when `t2 < a+400`

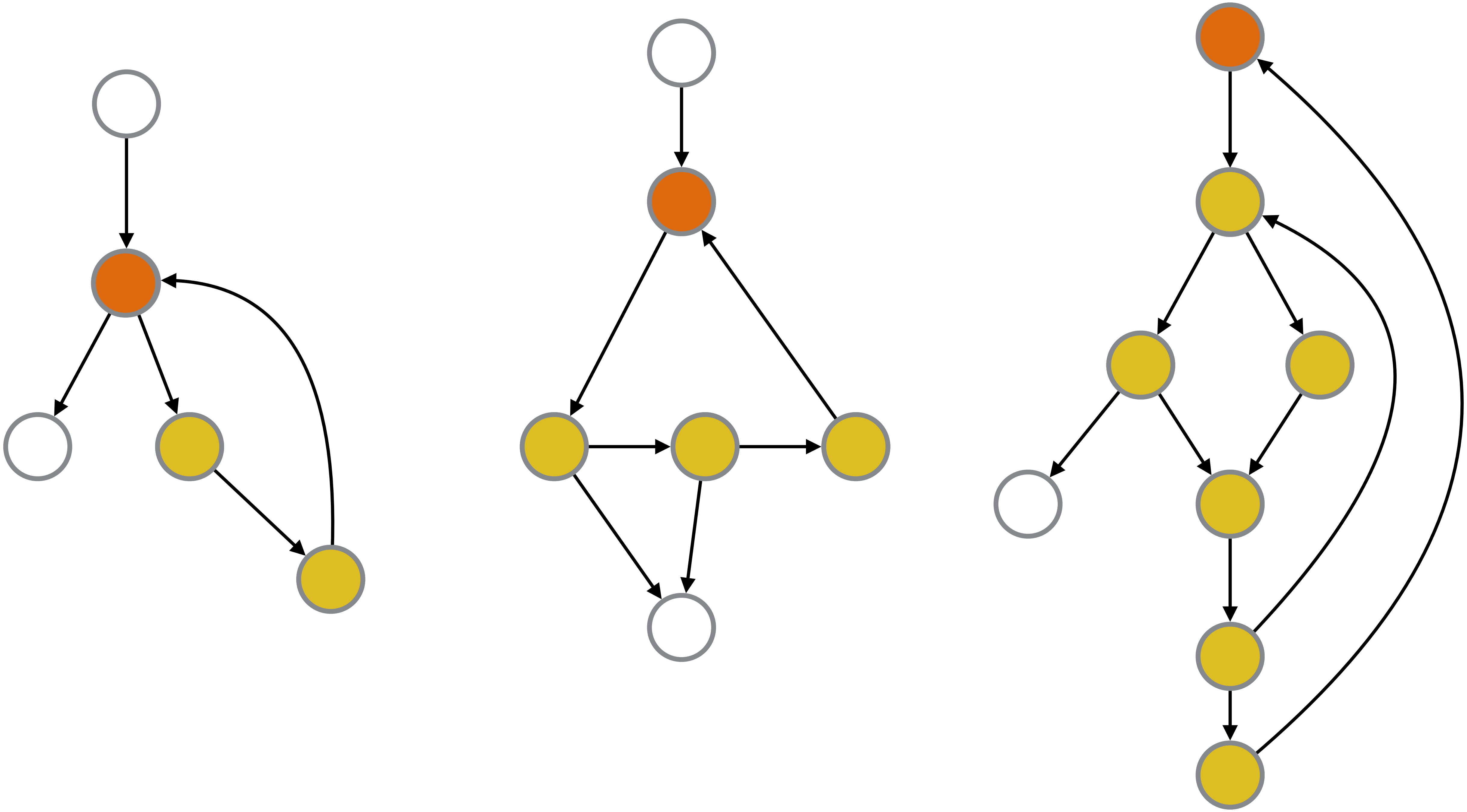
Loop Analysis

- How do we identify loops?
- What is a loop?
  - Can't just "look" at graphs
  - We're going to assume some additional structure
- **Definition:** a **loop** is a subset *S* of nodes where:
  - *S* is strongly connected:
    - For any two nodes in *S*, there is a path from one to the other using only nodes in *S*
  - There is a distinguished header node $h \in S$ such that there is no edge from a node outside *S* to $S \setminus \{h\}$

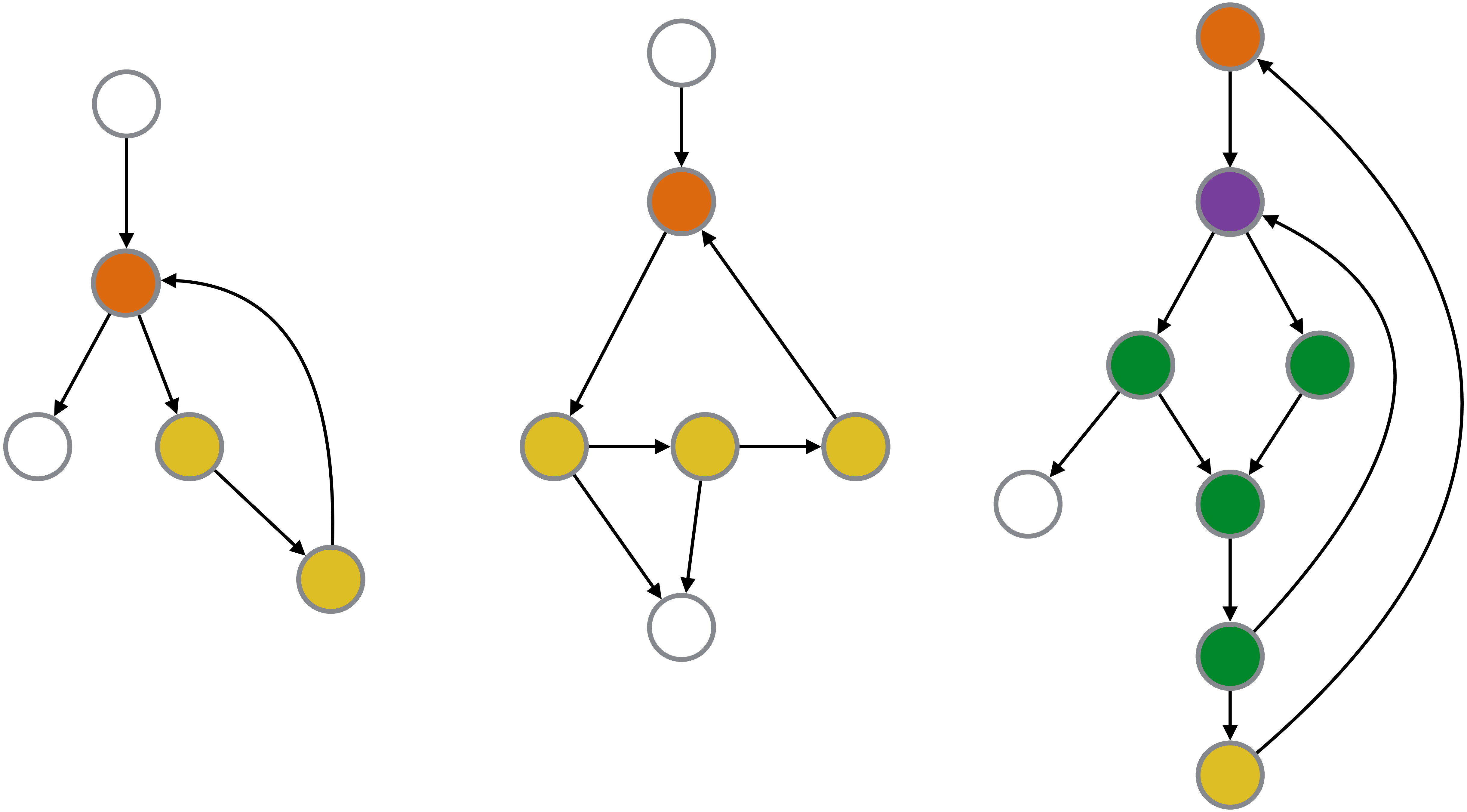
Examples



Examples

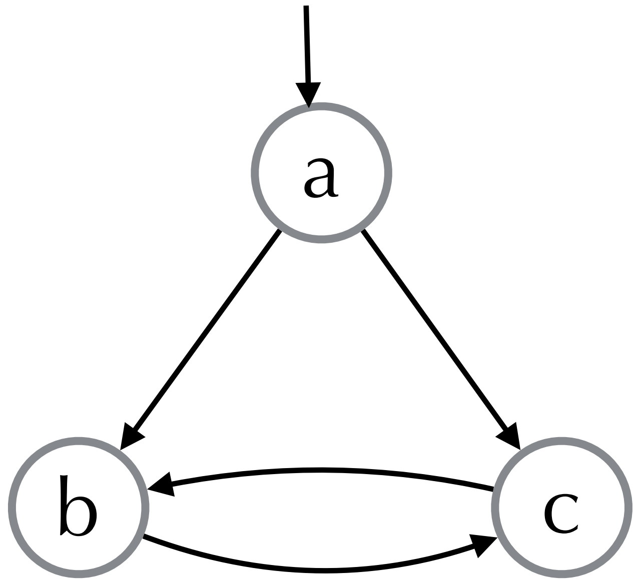


Examples



Non-example

- Consider the following:



- a can’t be header
  - No path from b to a or c to a
- b can’t be header
  - Has outside edge from a
- c can’t be header
  - Has outside edge from a
- So no loop...
- But clearly a cycle!

Reducible Flow Graphs

- So why did we define loops this way?
- Loop header gives us a "handle" for the loop
  - e.g., a good spot for hoisting invariant statements
- Structured control-flow only produces **reducible graphs**
  - a graph where all cycles are loops according to our definition.
  - Java: only reducible graphs
  - C/C++: goto can produce irreducible graph
- Many analyses & loop optimizations depend upon having reducible graphs

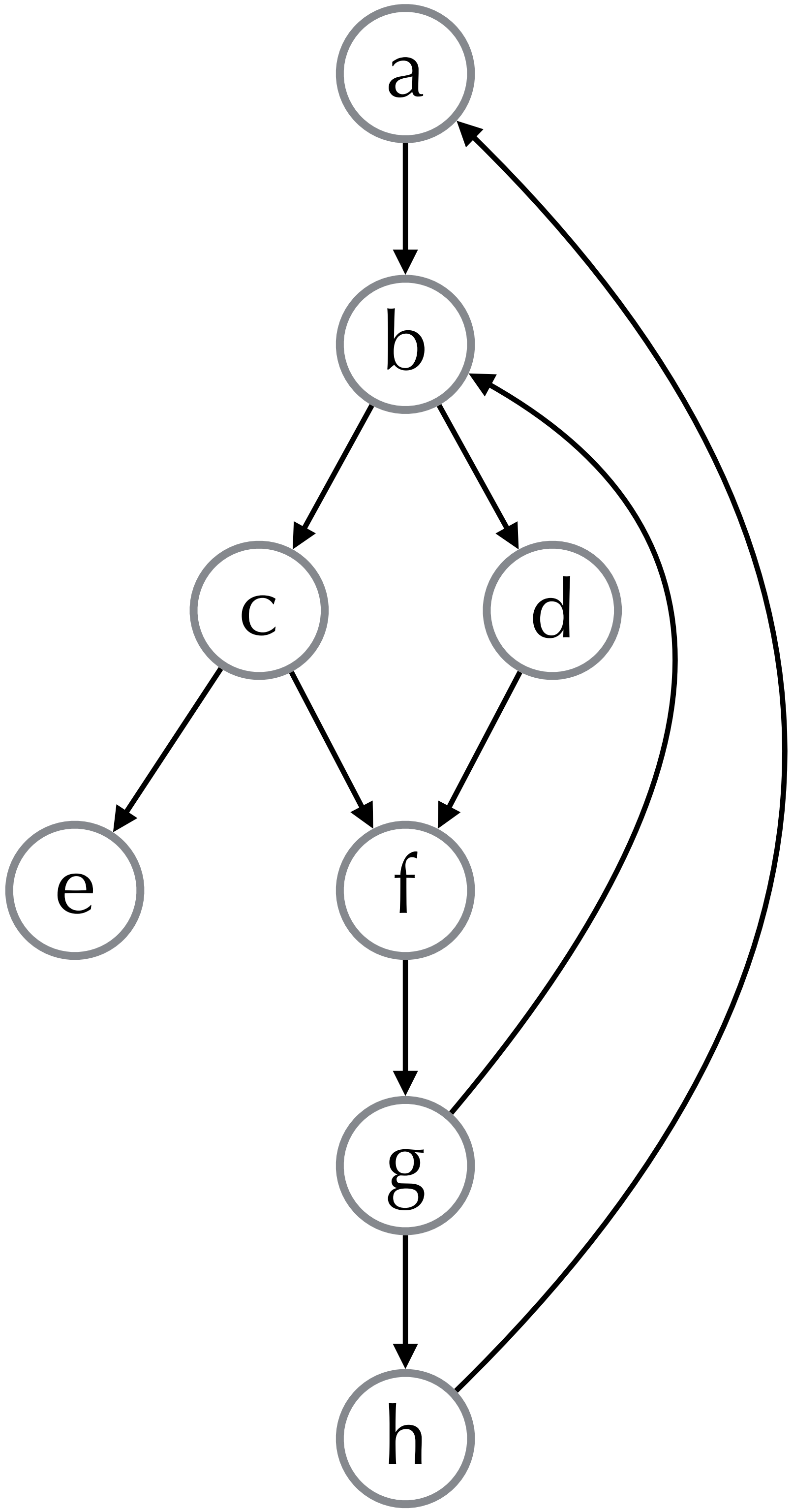
Finding Loops

- **Definition:** node *d* **dominates** node *n* if every path from the start node to *n* must go through *d*
- **Definition:** an edge from *n* to a dominator *d* is called a **back-edge**
- **Definition:** a **loop** of a back edge *n*→*d* is the set of nodes *x* such that *d* dominates *x* and there is a path from *x* to *n* not including *d*

- So to find loops, we figure out dominators, and identify back edges

Example

- a dominates a,b,c,d,e,f,g,h
- b dominates b,c,d,e,f,g,h
- c dominates c,e
- d dominates d
- e dominates e
- f dominates f,g,h
- g dominates g,h
- h dominates h
- back-edges?
  - g→b
  - h→a
- loops?



Calculating Dominators

- $D[n]$: the set of nodes that dominate n
- $D[n] = \{n\} \cup (D[p_1] \cap D[p_2] \cap \ldots \cap D[p_m])$

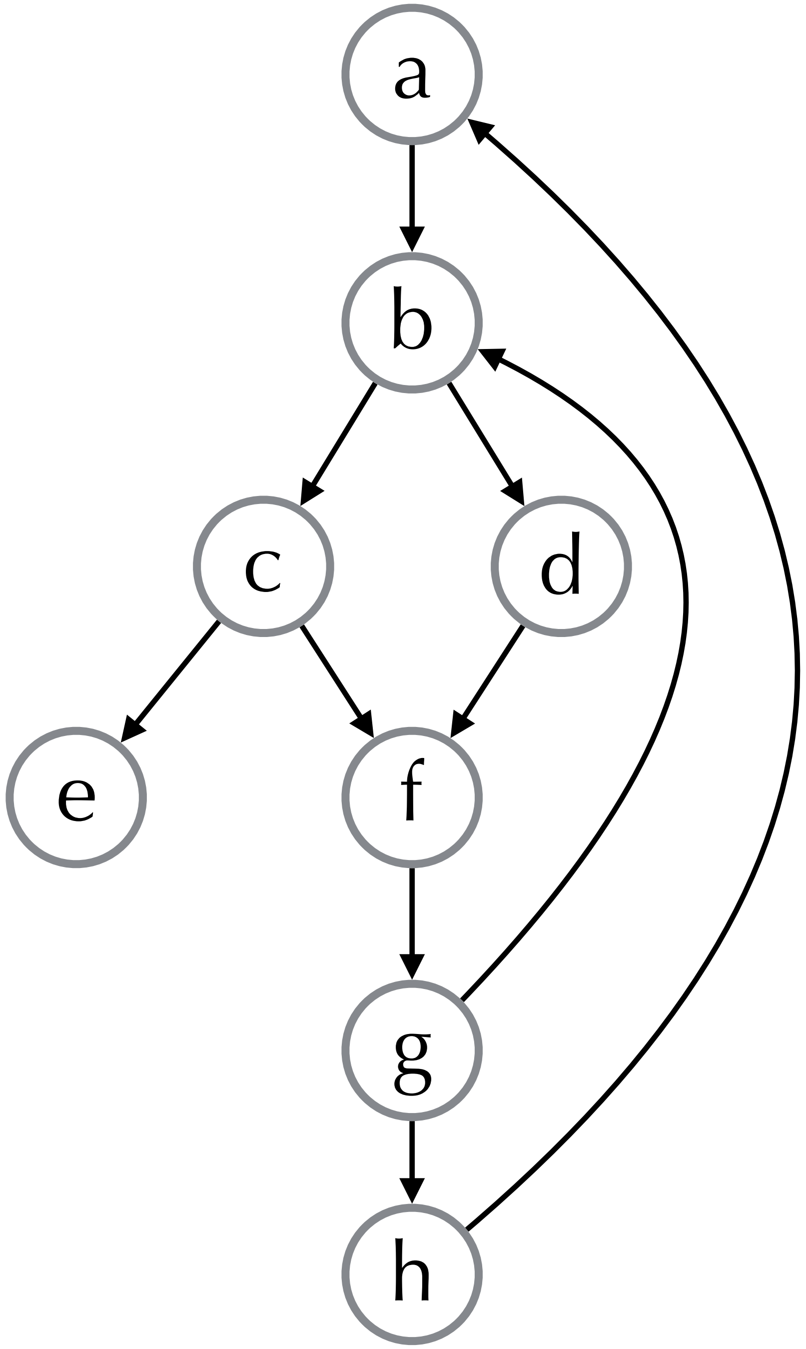
  where $pred[n] = \{p_1,p_2,\ldots,p_m\}$
- It's pretty easy to solve this equation:
  - start off assuming $D[n]$ is all nodes.
    - except for the start node (which is dominated only by itself)
  - iteratively update $D[n]$ based on predecessors until you reach a fixed point

Representing Dominators

- Don't actually need to keep set of all dominators for each node
- Instead, construct a **dominator tree**
  - Insight: if both *d* and *e* dominate *n*, then either *d* dominates *e* or vice versa
  - So that means that node *n* has a "closest" or **immediate dominator**

Example

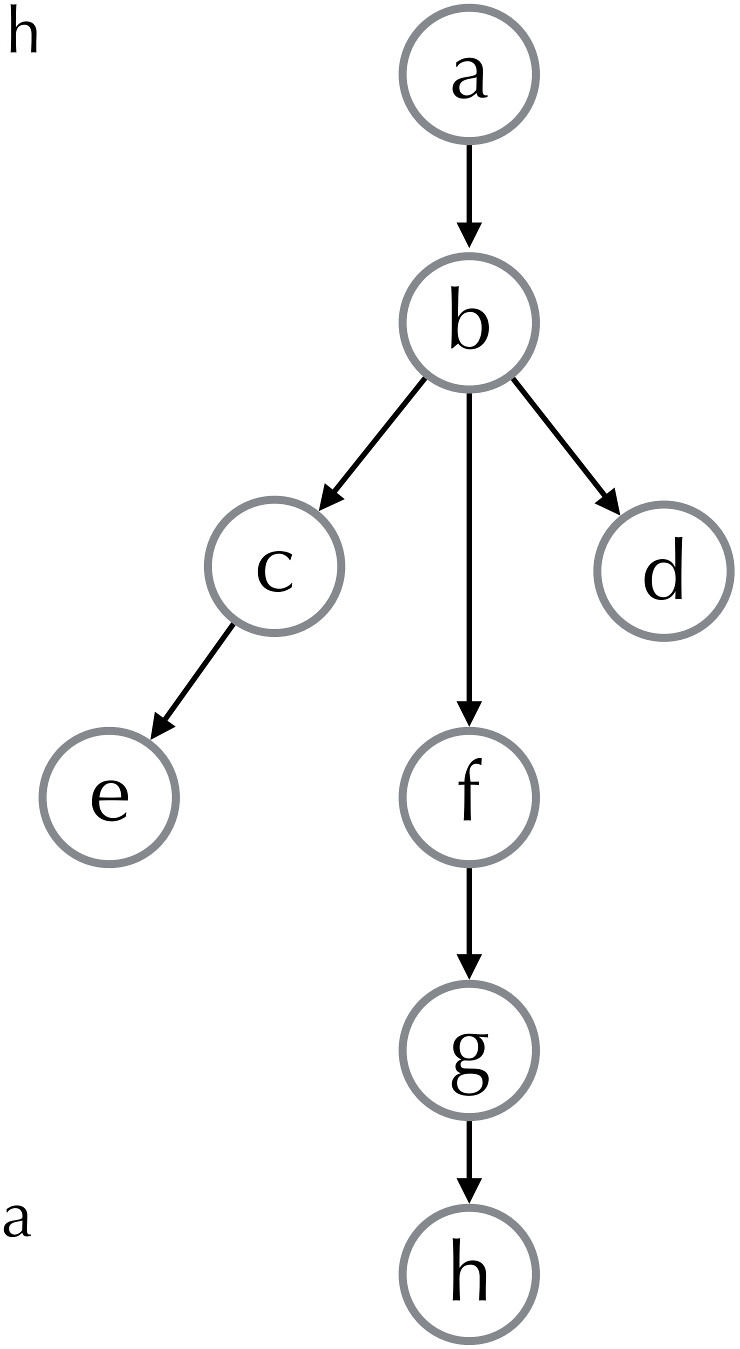
CFG



a dominates a,b,c,d,e,f,g,h
b dominates b,c,d,e,f,g,h
c dominates c,e
d dominates d
e dominates e
f dominates f,g,h
g dominates g,h
h dominates h

a dominated by a
b dominated by b,a
c dominated by c,b,a
d dominated by d,b,a
e dominated by e,c,b,a
f dominated by f,b,a
g dominated by g,f,b,a
h dominated by h,g,f,b,a

Immediate Dominator Tree



Nested Loops

- If loops A and B have distinct headers and all nodes in B are in A (i.e., B⊆A), then we say B is **nested** within A
- An **inner loop** is a nested loop that doesn't contain any other loops
- We usually concentrate our attention on nested loops first (since we spend most time in them)