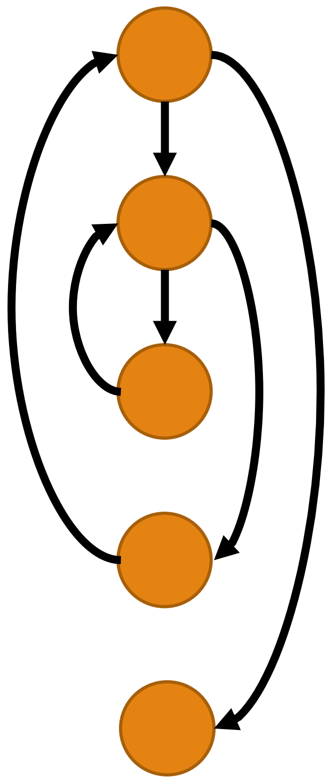
Loops

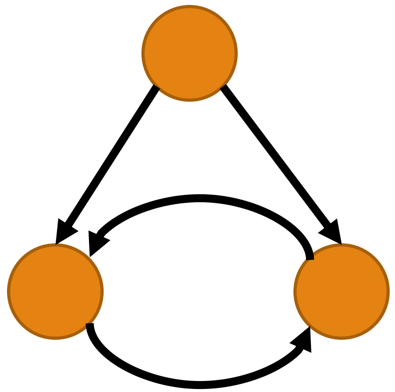
Loops!

```
while a.runs() loop {
  while b.runs() loop
    c.foo()
  pool;
  b.reset();
} pool
```



Not a Loop!

```
if a.isEven() then {
  Even:
    b.foo();
    goto Odd;
} else {
  Odd:
    b.bar();
    goto Even;
}
```



Optimizing Loops

Most program time is spent in loops.

- Otherwise, run time would be roughly proportional to program length.

Not-a-loop cycles are rare in practice, even with `goto`.

- Programmers tend not to think that way.
- How would you normally write the code on the previous slide?

Detecting Loops: Overview

"Natural Loops":

- Entry node ("***header***") that ***dominates*** all nodes in loop.
- ***Back edge*** from within loop body to header.

Independent of how loop is written syntactically.

- Same handling for `for`-loops, `while`-loops, etc.
- In practice, many uses of `goto` form natural loops.

Loop detection largely cribbed shamelessly from Jeffrey Ullman's slides at:
http://infolab.stanford.edu/~ullman/dragon/w06/lectures/dfa3.pdf

Dominators Revisited

X ***dominates*** Y (X $\geq$ Y)

- ***Every*** path to Y goes through X.
- Note: X $\geq$ X

X ***strictly dominates*** Y (X $>$ Y)

- X $\geq$ Y, but X $\neq$ Y.

Direction: Forward

Values: Sets of CFG nodes.

- v_{ENTRY} = {ENTRY}
- Initial value = *N*

Meet operator: $\cap$

Transfer function:

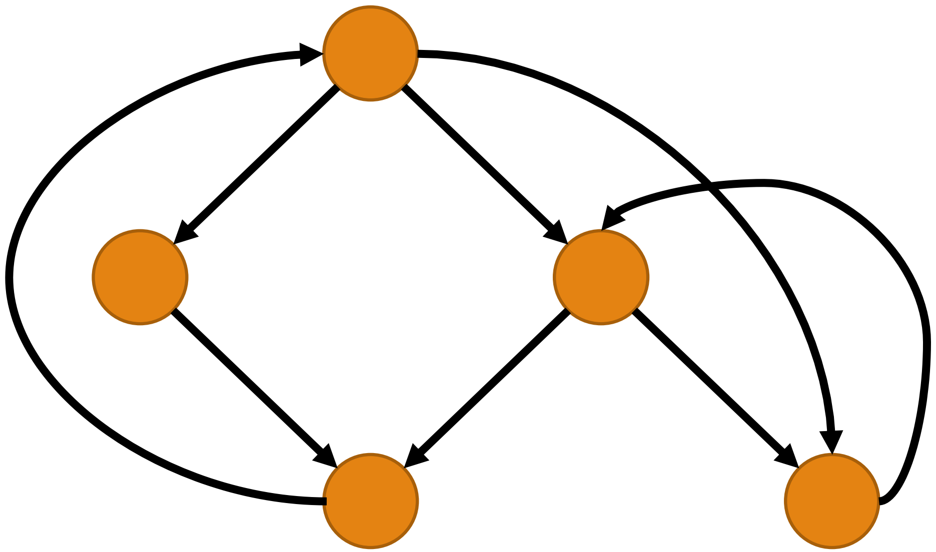
- $f_B(x) = x \cup \{B\}$

Kinds of Edges

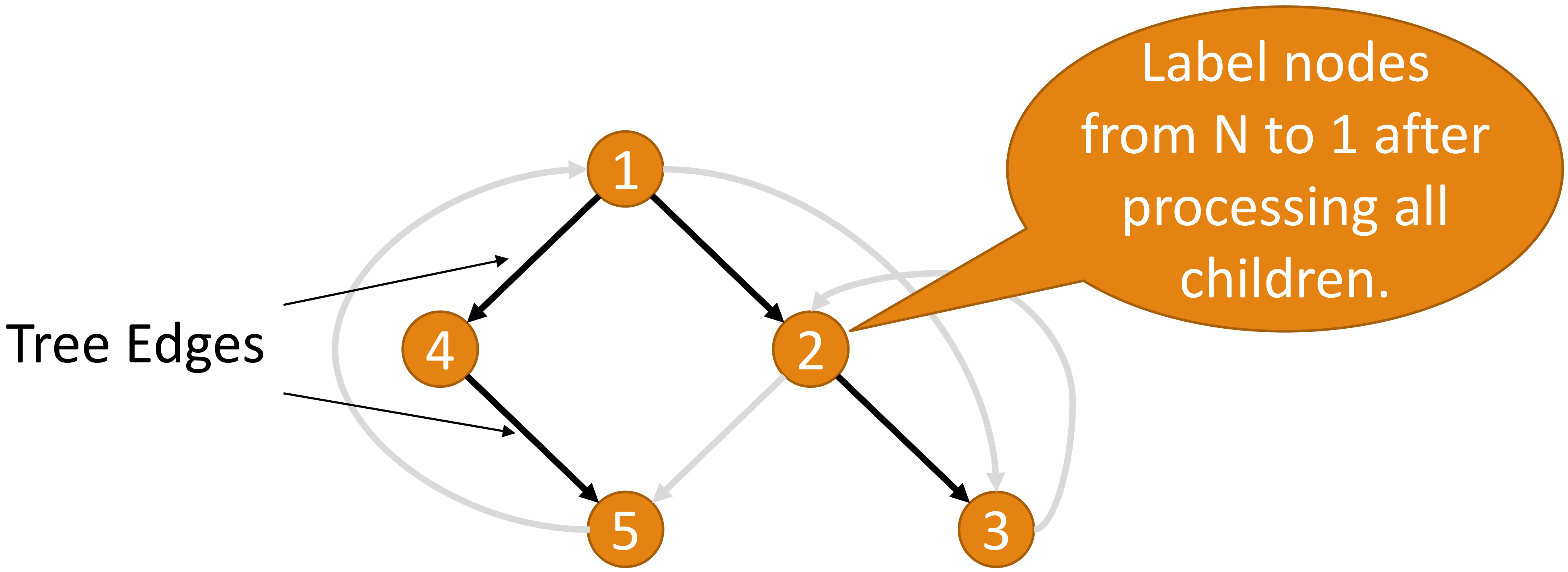
Defined relative to ***Depth-First Spanning Tree*** of CFG.

1. Tree edges.
2. ***Advancing edges***: Node to proper descendent (includes tree edges).
3. ***Retreating edges***: Node to ancestor (including self).
4. ***Cross edges***: No ancestor relationship between nodes.

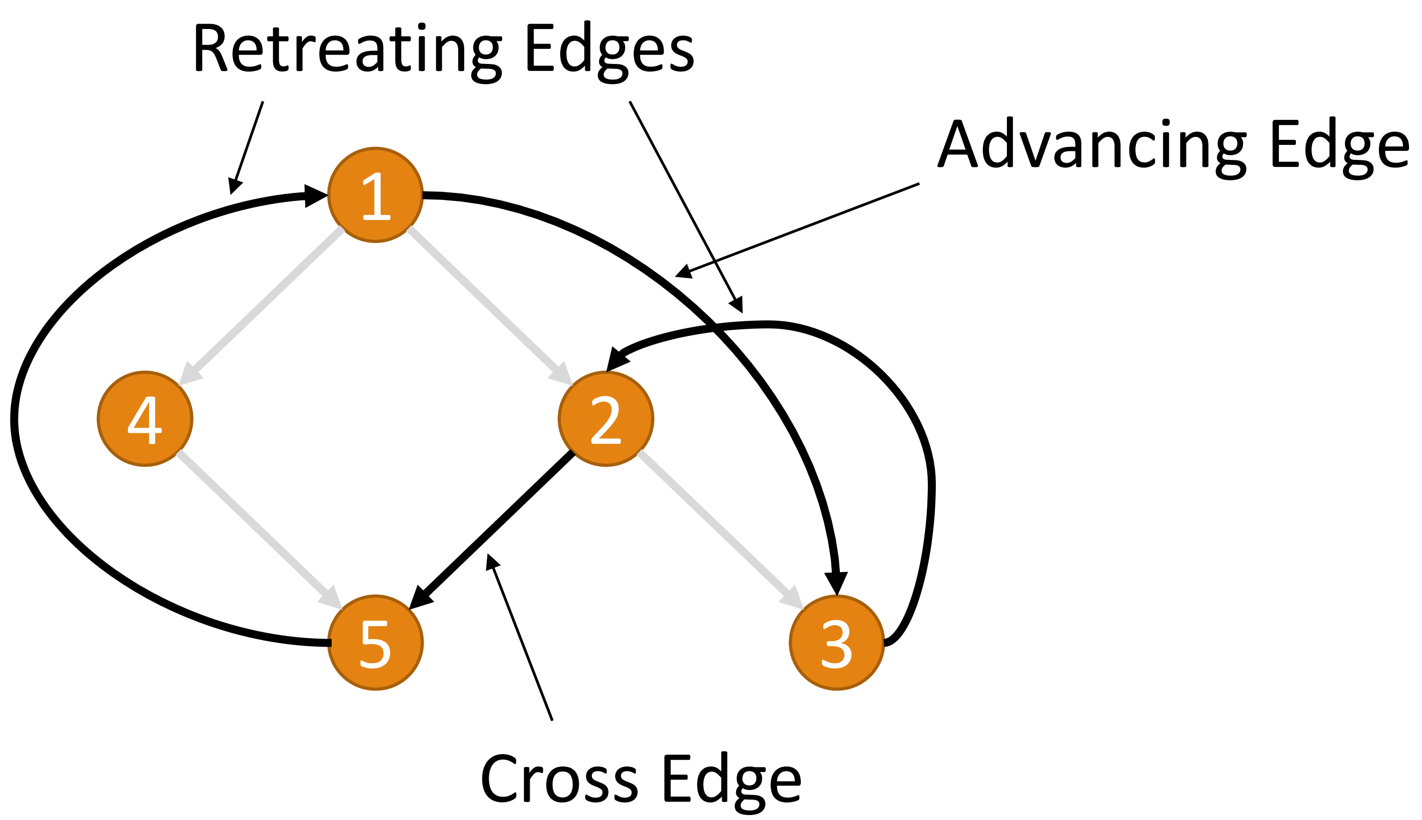
DFS Tree Edges Example



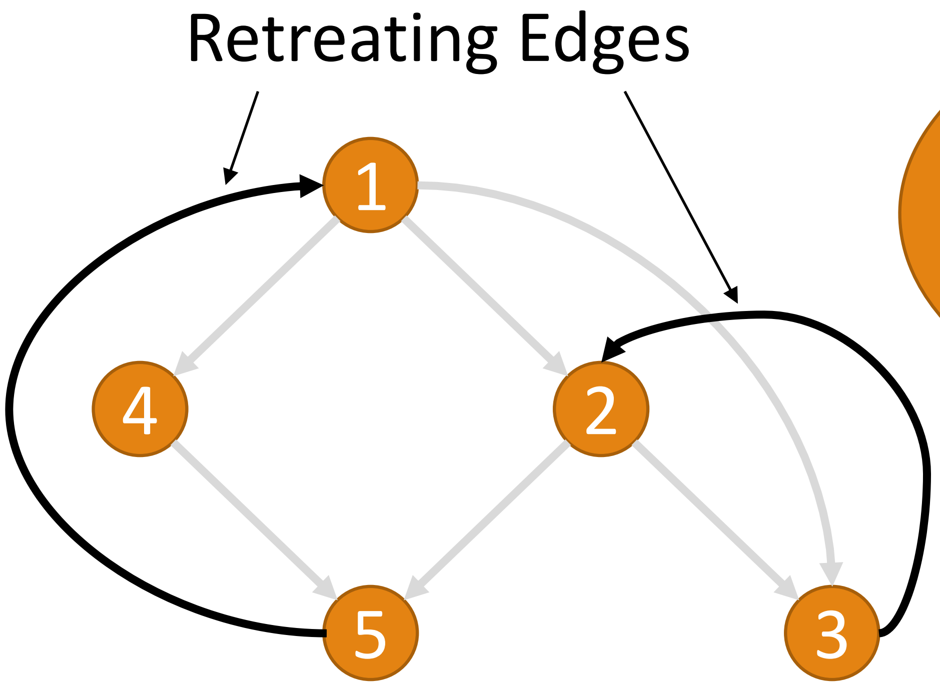
DFS Tree Edges Example



DFS Tree Edges Example



DFS Tree Edges Example



Retreating edges always go from high to low in DFS order.

Back Edges, Reducibility, and Depth

An edge is a ***back edge*** if its head dominates its tail.

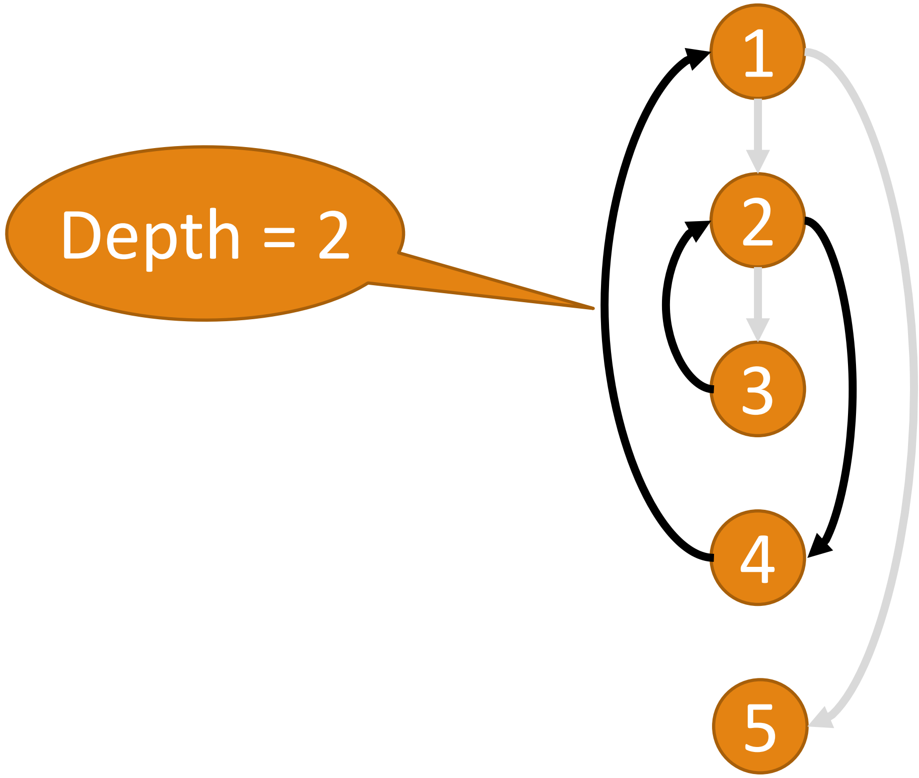
- Back edges are retreating edges.

A graph is ***reducible*** iff all retreating edges are back edges.

The ***depth*** of a CFG is the maximum number of retreating edges on any acyclic path.

- For reducible graphs, depth is fixed regardless of order of visiting children.

Depth Example



Loop Detection Algorithm

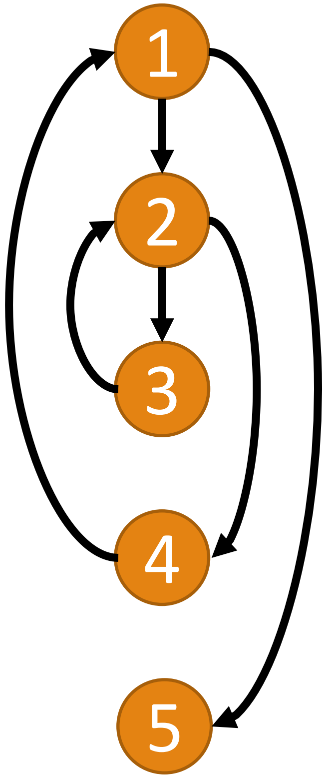
For each back edge $n \rightarrow d$:

$loop \leftarrow \{n, d\}$

Mark d as visited.

For each node n' in DFS of *reverse* edges from n:

$loop \leftarrow \{n'\} \cup loop$



Loop Detection Algorithm

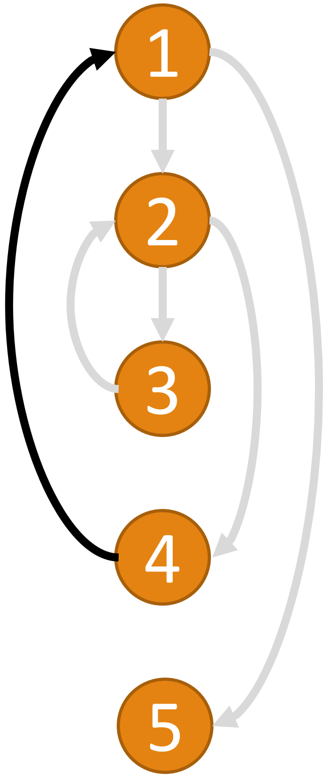
For each back edge $n \rightarrow d$:

$loop \leftarrow \{n, d\}$

Mark d as visited.

For each node n' in DFS of *reverse* edges from n:

$loop \leftarrow \{n'\} \cup loop$



Edge: $4 \rightarrow 1$

$loop = \{1, 4\}$

Loop Detection Algorithm

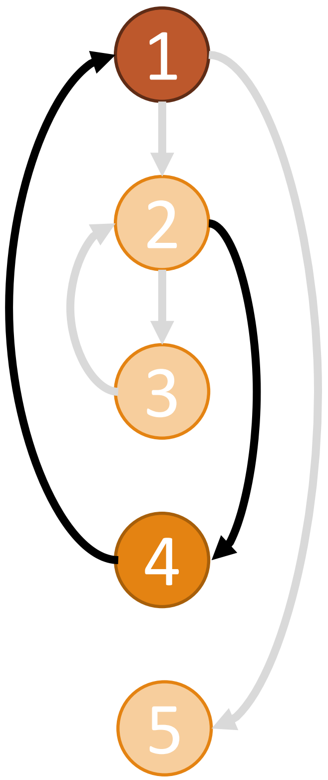
For each back edge $n \rightarrow d$:

$loop \leftarrow \{n, d\}$

Mark d as visited.

For each node n' in DFS of ***reverse*** edges from n:

$loop \leftarrow \{n'\} \cup loop$



Edge: $4 \rightarrow 1$

$loop = \{1, 4\}$

Loop Detection Algorithm

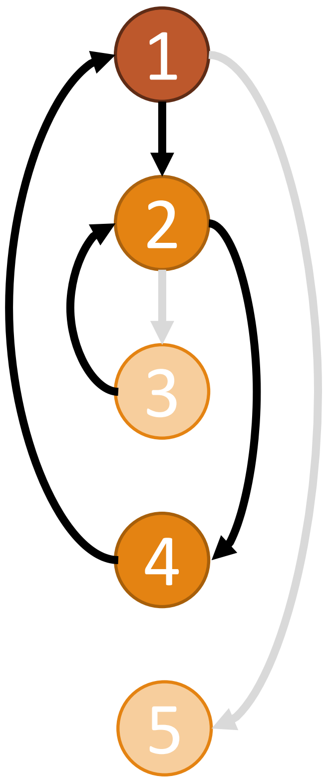
For each back edge $n \rightarrow d$:

$loop \leftarrow \{n, d\}$

Mark d as visited.

For each node n' in DFS of ***reverse*** edges from n:

$loop \leftarrow \{n'\} \cup loop$



Edge: $4 \rightarrow 1$

$loop = \{1, 2, 4\}$

Loop Detection Algorithm

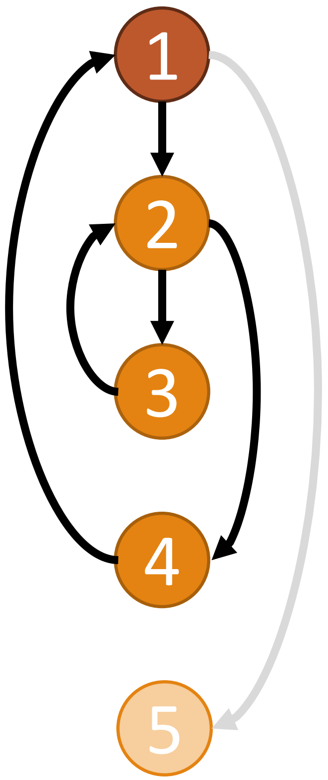
For each back edge $n \rightarrow d$:

$loop \leftarrow \{n, d\}$

Mark d as visited.

For each node n' in DFS of *reverse* edges from n:

$loop \leftarrow \{n'\} \cup loop$



Edge: $4 \rightarrow 1$

$loop = \{1, 2, 3, 4\}$

Loop Detection Algorithm

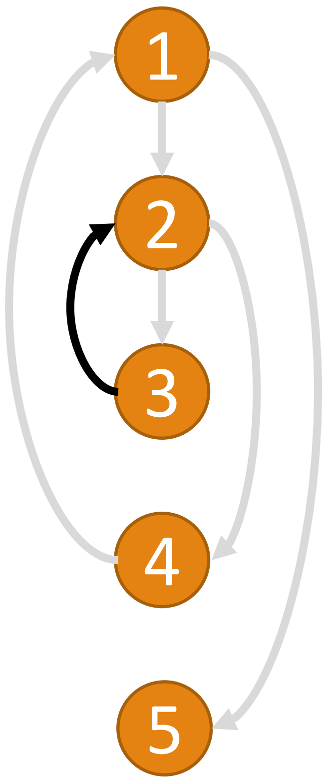
For each back edge $n \to d$:

$loop \leftarrow \{n, d\}$

Mark d as visited.

For each node n' in DFS of ***reverse*** edges from n:

$loop \leftarrow \{n'\} \cup loop$



Edge: $3 \to 2$

$loop = \{2, 3\}$

Loop Detection Algorithm

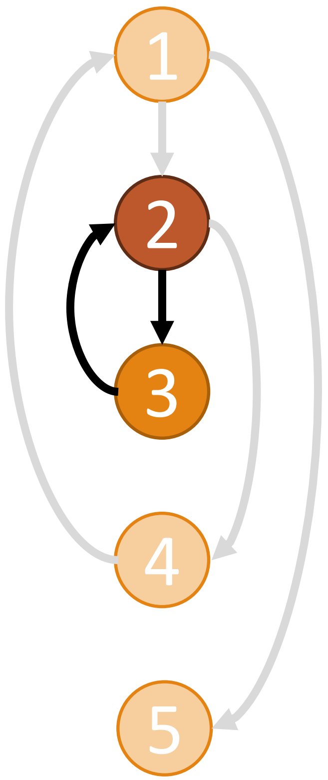
For each back edge $n \rightarrow d$:

$loop \leftarrow \{n, d\}$

Mark d as visited.

For each node n' in DFS of ***reverse*** edges from n:

$loop \leftarrow \{n'\} \cup loop$



Edge: $3 \rightarrow 2$

$loop = \{2, 3\}$

Loop Detection Algorithm

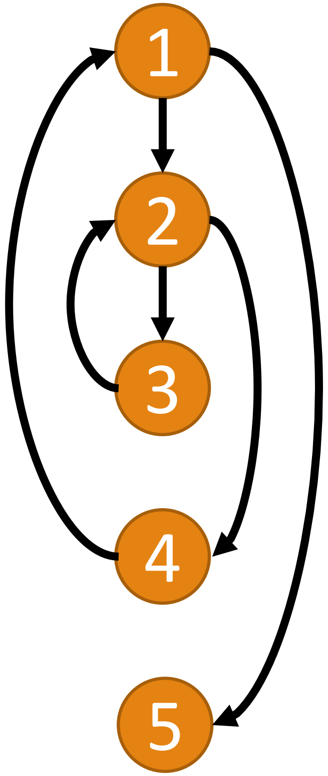
For each back edge $n \rightarrow d$:

$loop \leftarrow \{n, d\}$

Mark d as visited.

For each node n' in DFS of *reverse* edges from n:

$loop \leftarrow \{n'\} \cup loop$



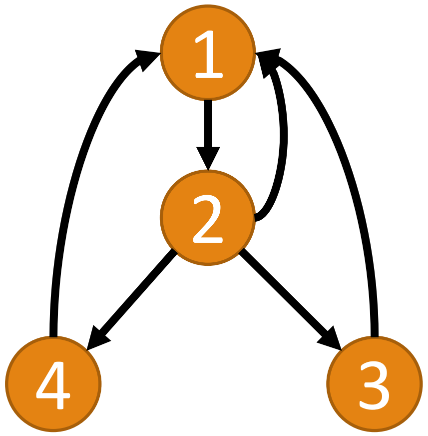
Found 2 loops:

- A: $\{1, 2, 3, 4\}$
- B: $\{2, 3\}$

Since $B \subset A$, we know A contains B.

B is ***innermost*** loop.

Overlapping Loops



Loops:

- A: {1, 2}
- B: {1, 2, 3}
- C: {1, 2, 4}

Merge B and C:

- BC: {1, 2, 3, 4}

BC contains A.

Loop Unrolling

Loop Example

```
    li r0 <- 0
    syscall IO.in_int
    li r2 <- 0
    li r3 <- 1
L1: ble r1 r0 L2
    add r2 <- r2 r0
    add r0 <- r0 r3
    jmp L1
L2: mov r1 <- r2
    syscall IO.out_int
```

```
> ./cool --profile test.cl-asm
8191
33542145
PROFILE:              instructions = 32774
PROFILE:           pushes and pops =     0
PROFILE:                cache hits =     0
PROFILE:              cache misses =    15
PROFILE:        branch predictions = 16382
PROFILE:     branch mispredictions =     1
PROFILE:           multiplications =     0
PROFILE:                 divisions =     0
PROFILE:              system calls =     4
CYCLES: 38294
```

Loop Example

```
    li r3 <- 1
L1: ble r1 r0 L2
    add r2 <- r2 r0
    add r0 <- r0 r3

    jmp L1
L2: mov r1 <- r2
    syscall IO.out_int
```

```
>
```

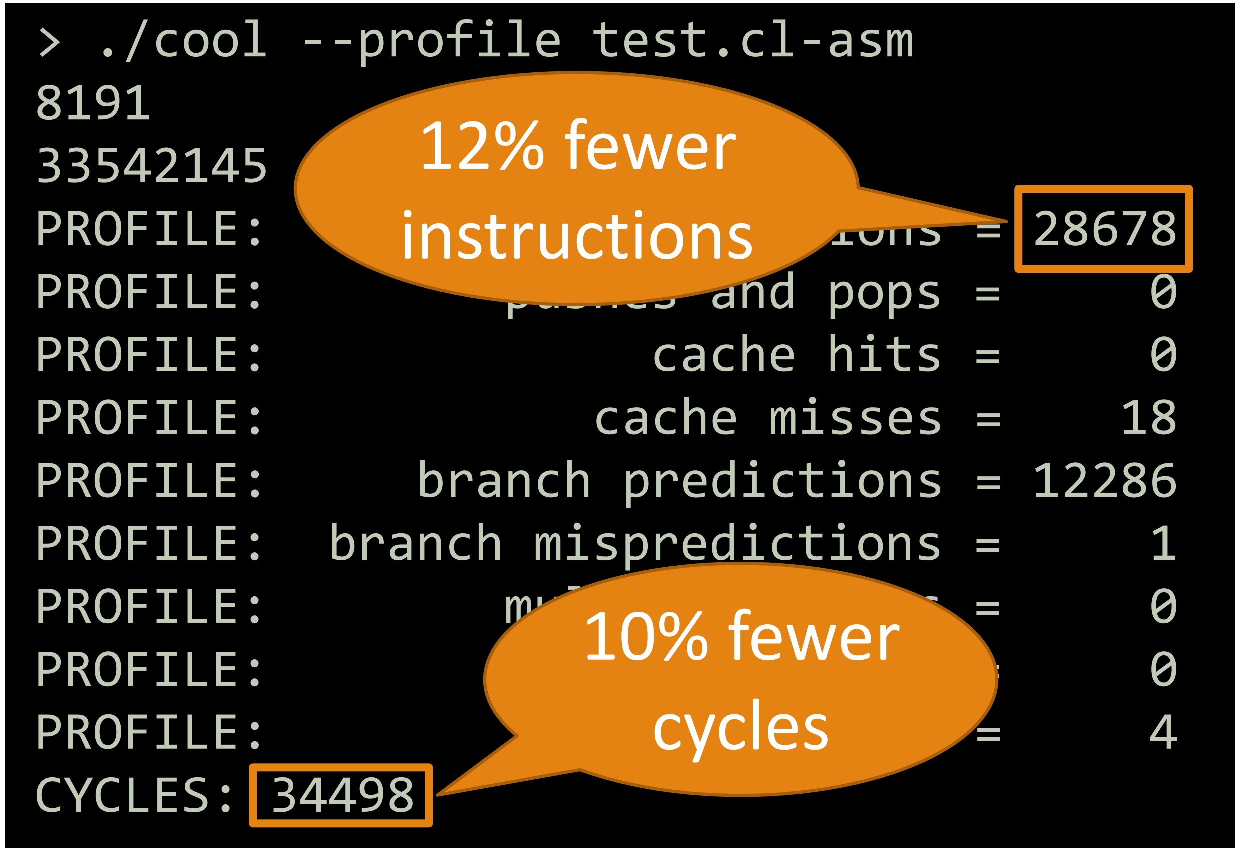
Loop Example

```
    li r3 <- 1
L1: ble r1 r0 L2
    add r2 <- r2 r0
    add r0 <- r0 r3
    ble r1 r0 L2
    add r2 <- r2 r0
    add r0 <- r0 r3
    jmp L1
L2: mov r1 <- r2
    syscall IO.out_int
```

```
> ./cool --profile test.cl-asm
8191
33542145
PROFILE:           instructions = 28678
PROFILE:         pushes and pops =     0
PROFILE:              cache hits =     0
PROFILE:            cache misses =    18
PROFILE:      branch predictions = 12286
PROFILE:   branch mispredictions =     1
PROFILE:         multiplications =     0
PROFILE:               divisions =     0
PROFILE:            system calls =     4
CYCLES: 34498
```

Loop Example

```
    li r3 <- 1
L1: ble r1 r0 L2
    add r2 <- r2 r0
    add r0 <- r0 r3
    ble r1 r0 L2
    add r2 <- r2 r0
    add r0 <- r0 r3
    jmp L1
L2: mov r1 <- r2
    syscall IO.out_int
```



Loop Example

```
    li r3 <- 1
L1: ble r1 r0 L2
    add r2 <- r2 r
    add r0 <- r0 r
    ble r1 r0 L2
    add r2 <- r2 r0
    add r0 <- r0 r3
    jmp L1
L2: mov r1 <- r2
    syscall IO.out_int
```

Can we remove this?

```
> ./cool --profile test.cl-asm
 91
 2145
 ILE:            instructions = 28678
 ILE:          pushes and pops =     0
 OFILE:             cache hits =     0
PROFILE:          cache misses =    18
PROFILE:    branch predictions = 12286
PROFILE: branch mispredictions =     1
PROFILE:       multiplications =     0
PROFILE:             divisions =     0
PROFILE:          system calls =     4
CYCLES: 34498
```

Loop Example

```
    li r3 <- 1
L1: ble r1 r0 L2
    add r2 <- r2 r0
    add r0 <- r0 r3

    add r2 <- r2 r0
    add r0 <- r0 r3
    jmp L1
L2: mov r1 <- r2
    syscall IO.out_int
```

```
> ./cool --profile test.cl-asm
8191
33550336
PROFILE:             instructions = 24586
PROFILE:          pushes and pops =     0
PROFILE:               cache hits =     0
PROFILE:             cache misses =    17
PROFILE:       branch predictions =  8192
PROFILE:    branch mispredictions =     1
PROFILE:          multiplications =     0
PROFILE:                divisions =     0
PROFILE:             system calls =     4
CYCLES: 30306
```

Loop Example

```
    li r3 <- 1
L1: ble r1 r0 L2
    add r2 <- r2 r0
    add r0 <- r0 r3

    add r2 <- r2 r0
    add r0 <- r0 r3
    jmp L1
L2: mov r1 <- r2
    syscall IO.out_int
```

```
> ./cool --profile test.cl-asm
8191
33550336
PROFILE:           instructions = 24586
PROFILE:        pushes and pops =     0
PROFILE:             cache hits =     0
PROFILE:           cache misses =    17
PROFILE:     branch predictions =  8192
PROFILE:  branch mispredictions =     1
PROFILE:        mu              =     0
PROFILE:                              0
PROFILE:                        =     4
CYCLES: 30306
```

21% fewer cycles

Loop Example

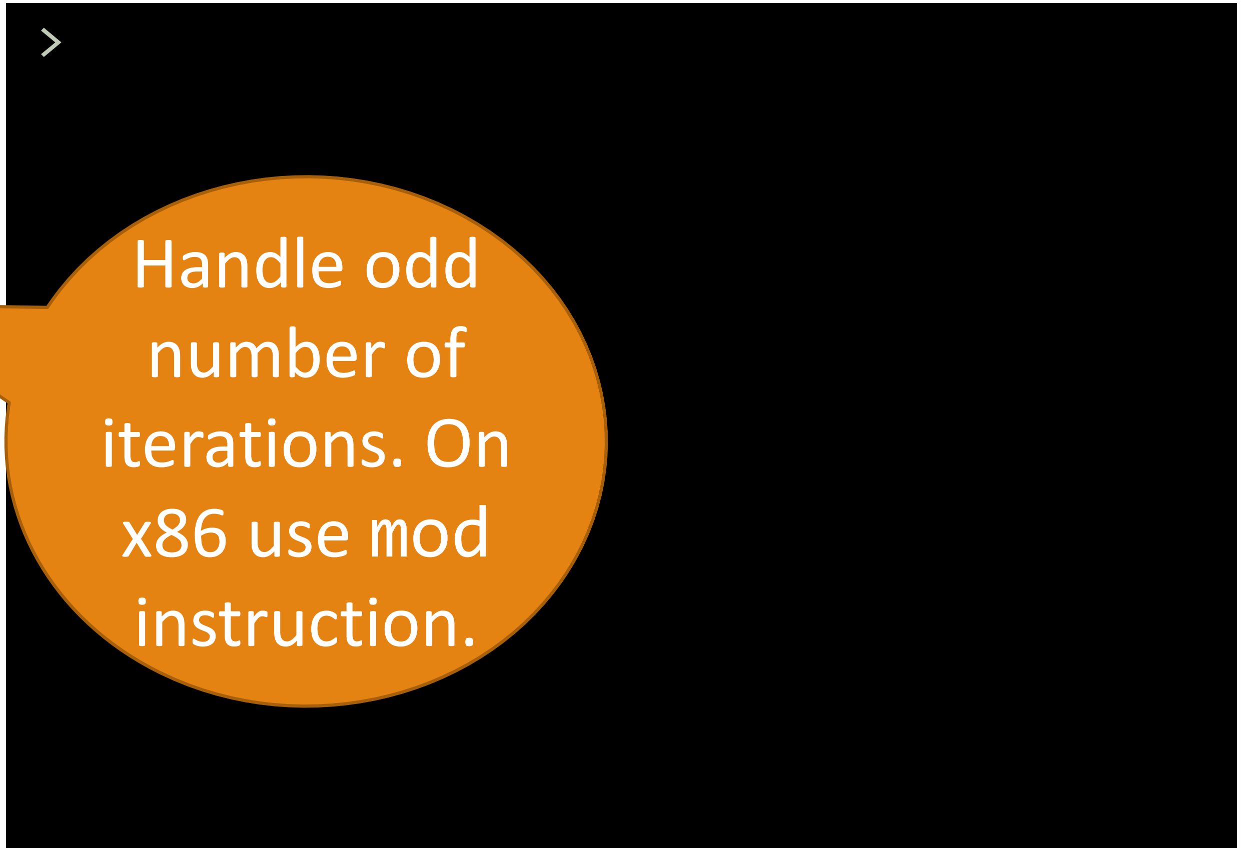
```
    li r3 <- 1
L1: ble r1 r0 L2
    add r2 <- r2 r0
    add r0 <- r0 r3

    add r2 <- r2 r0
    add r0 <- r0 r3
    jmp L1
L2: mov r1 <- r2
    syscall IO.out_int
```

```
> ./cool --profile test.cl-asm
8191
33550336
PROFILE:                      ons =  24586
PROFILE:                      ps =      0
PROFILE:                   e hits =     0
PROFILE:             cache misses =    17
PROFILE:       branch predictions =  8192
PROFILE:    branch mispredictions =     1
PROFILE:          multiplications =     0
PROFILE:                divisions =     0
PROFILE:             system calls =     4
CYCLES: 30306
```

Should be 33542145!

Loop Example

```
     li r3 <- 1
     li r4 <- 2
     div r5 <- r1 r4
     mul r6 <- r5 r4
     beq r6 r1 L1
     add r2 <- r2 r0
     add r0 <- r0 r3
L1:  ble r1 r0 L2
     add r2 <- r2 r0
     add r0 <- r0 r3
```



Handle odd number of iterations. On x86 use mod instruction.

Loop Example

```
    li r3 <- 1
    li r4 <- 2
    div r5 <- r1 r4
    mul r6 <- r5 r4
    beq r6 r1 L1
    add r2 <- r2 r0
    add r0 <- r0 r3
L1: ble r1 r0 L2
    add r2 <- r2 r0
    add r0 <- r0 r3
```

```
> ./cool test.cl-asm --profile
8191
33542145
PROFILE:              uctions =  24586
PROFILE:             nd pops =      0
PROFILE:            ache hits =      0
PROFILE:         cache misses =     23
PROFILE:   branch predictions =   8191
PROFILE: branch mispredictions =     1
PROFILE:       m                =      1
PROFILE:                              1
PROFILE:                        =      4
CYCLES: 30956
```

Right answer!

19% fewer cycles

Bonus Material

Induction Variables

Knowing loop bounds would help remove loop instructions.

Many loop indices are ***affine expressions*** of program variables.

- E.g., $c_0 + c_1 v_1 + c_2 v_2 \ldots$

Induction variables: affine expressions of number of iterations.

- I.e., $c_0 + c_1 i$

Symbolic analysis can learn induction variables.

Affine Expression Example

```
for (int m = 10; m < 20; m++) {
    x = m * 3;
    a = foo(x);
    y = a + 10;
}
```

```
m = ?
x = ?
a = ?
y = ?
```

Affine Expression Example

```
for (int m = 10; m < 20; m++) {
    x = m * 3;
    a = foo(x);
    y = a + 10;
}
```

m = $i + 10$
x = $3i + 30$
a = ?
y = a + 10

Data-Flow Analysis for Affine Expressions

Values: ⊤ (unknown), affine expression, or ⊥ (not affine).

- Let $f(m)$ be a function to look up variables in the current data-flow value m.

Meet operator:

- $$(f_1 \wedge f_2)(m)(v) = \begin{cases} f_1(m)(v), & f_1(m)(v) = f_2(m)(v) \\ \bot, & \text{otherwise} \end{cases}$$

Data-Flow Analysis for Affine Expressions

Transfer functions:

- For assignment statements to x
  - when $(c_1 = 0 \text{ or } y = \perp)$ and $(c_2 = 0 \text{ or } z = \perp)$:
  - $f_s(m)(x) = \begin{cases} m(v), & v \neq x \\ c_0 + c_1 m(y) + c_2 m(z), & x \leftarrow c_0 + c_1 y + c_2 z \\ \perp, & \text{otherwise} \end{cases}$
- Otherwise, $f_s = I$

Composition: $f_2 \circ f_1$ = substitute values from f_1 into f_2.

Handling Iteration

Let f^i denote composing f with itself i times.

- ***Basic induction variables***:
  If $f(m)(x) = m(x) + c$, $f^i(m)(x) = m(x) + ci$
- ***Symbolic constants***:
  If $f(m)(x) = m(x)$, $f^i(m)(x) = m(x)$

Handling Iteration

Let f^i denote composing f with itself i times.

- ***Induction variables*** (if x_1 ... are basic induction variables or symbolic constants):

  If $f(m)(x) = c_0 + c_1 m(x_1) + \cdots$,
  $f^i(m)(x) = c_0 + c_1 f^i(m)(x_1) \ldots$

- Otherwise, $f^i(m)(x) = \bot$.

Symbolic Analysis for Affine Expressions

Start with innermost loops and work outward.