Optimization 2

15-411/15-611 Compiler Design

Seth Copen Goldstein

March 24, 2026

Today

- What is a loop?
- Control Flow Analysis
- Finding Loops
- Natural Loops
- Reducibility
- Classic Loop Optimizations
  - LICM
  - Induction Variable Elimination

Common loop optimizations

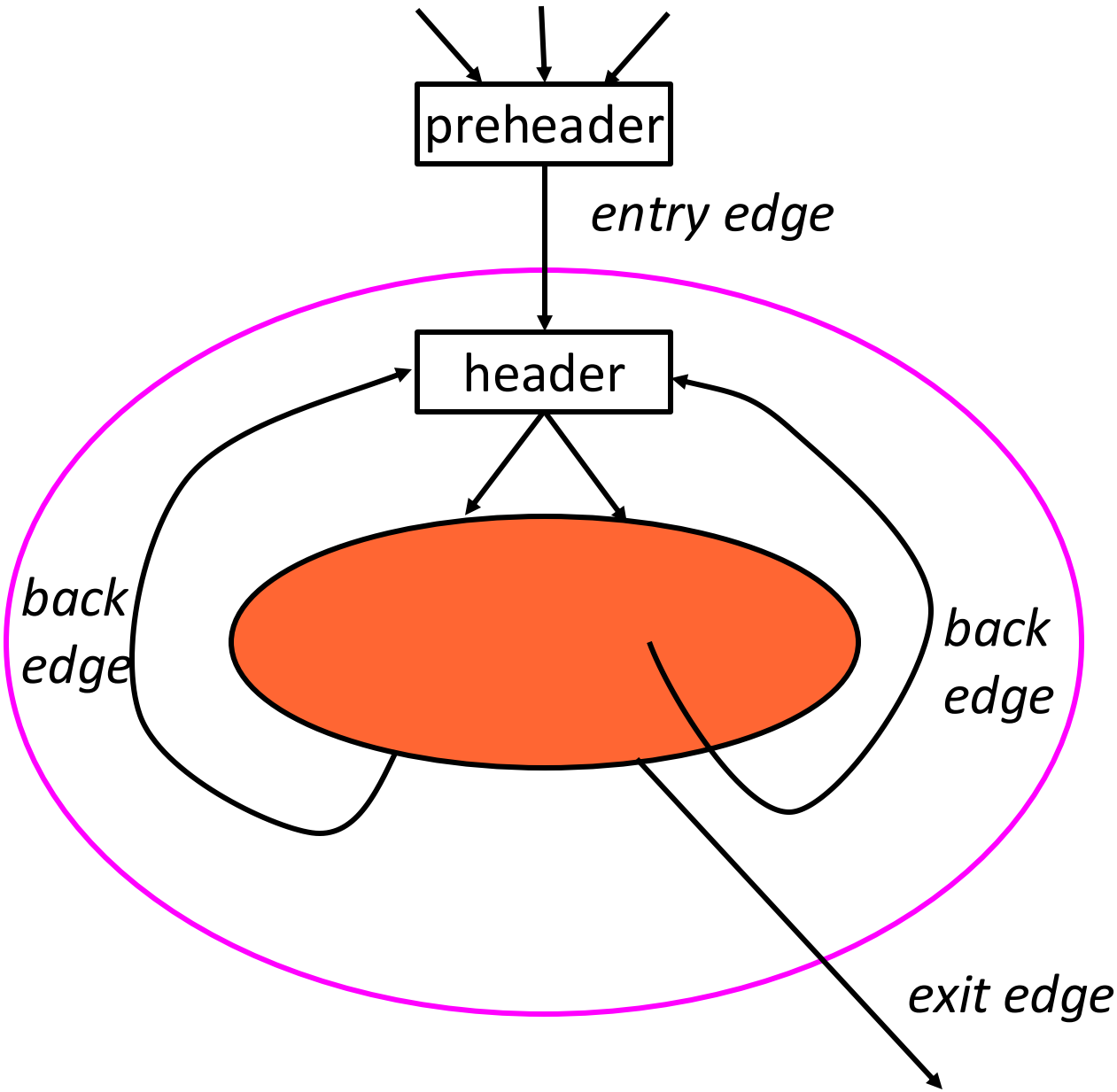
- Hoisting of loop-invariant computations
  - pre-compute before entering the loop
- Elimination of induction variables
  - change p=i*w+b to p=b,p+=w, when w,b invariant
- Loop unrolling
  - to improve scheduling of the loop body

Scalar opts, DF analysis, Control flow analysis

- Software pipelining
  - To improve scheduling of the loop body
- Loop permutation
  - to improve cache memory performance

Requires understanding data dependencies

A Loop



Loop Terminology

Loop: Strongly Connected Component of CFG

Entry Edge: tail not in loop, head in loop.

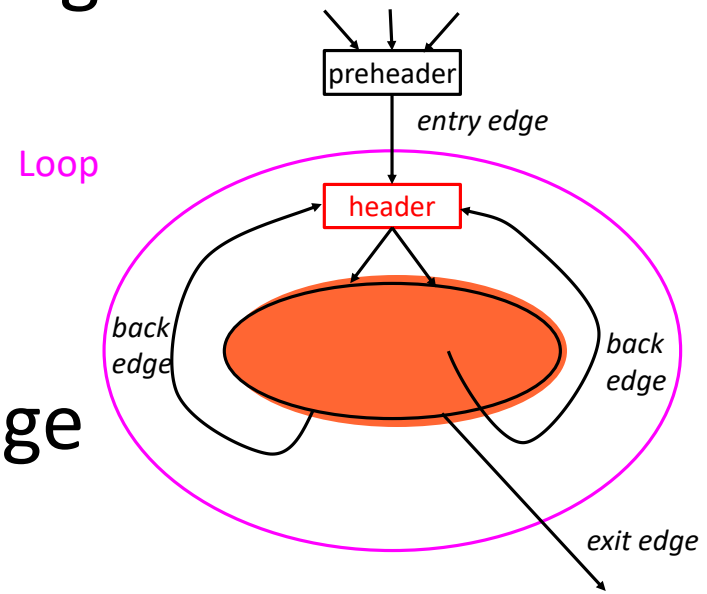
Exit edge: tail in loop, head not in loop

Loop Header: target of entry edge

Back Edge: target is header,
source is in loop

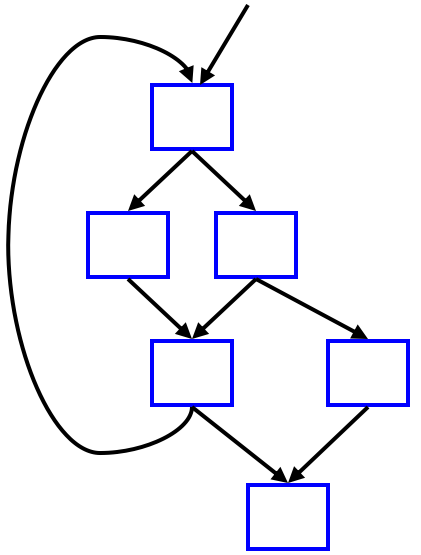
Preheader:
Source of the only entry edge

Natural Loop:
A Loop with only a single loop header



Finding Loops

- To optimize loops, we need to find them!
- Specifically:
  - loop-header node(s)
    - nodes in a loop that have immediate predecessors not in the loop
  - back edge(s)
    - control-flow edges to previously executed nodes
  - all nodes in the loop body



Control-flow analysis

- Many languages have goto and other complex control, so loops can be hard to find in general
- Determining the control structure of a program is called control-flow analysis
- Based on the notion of dominators

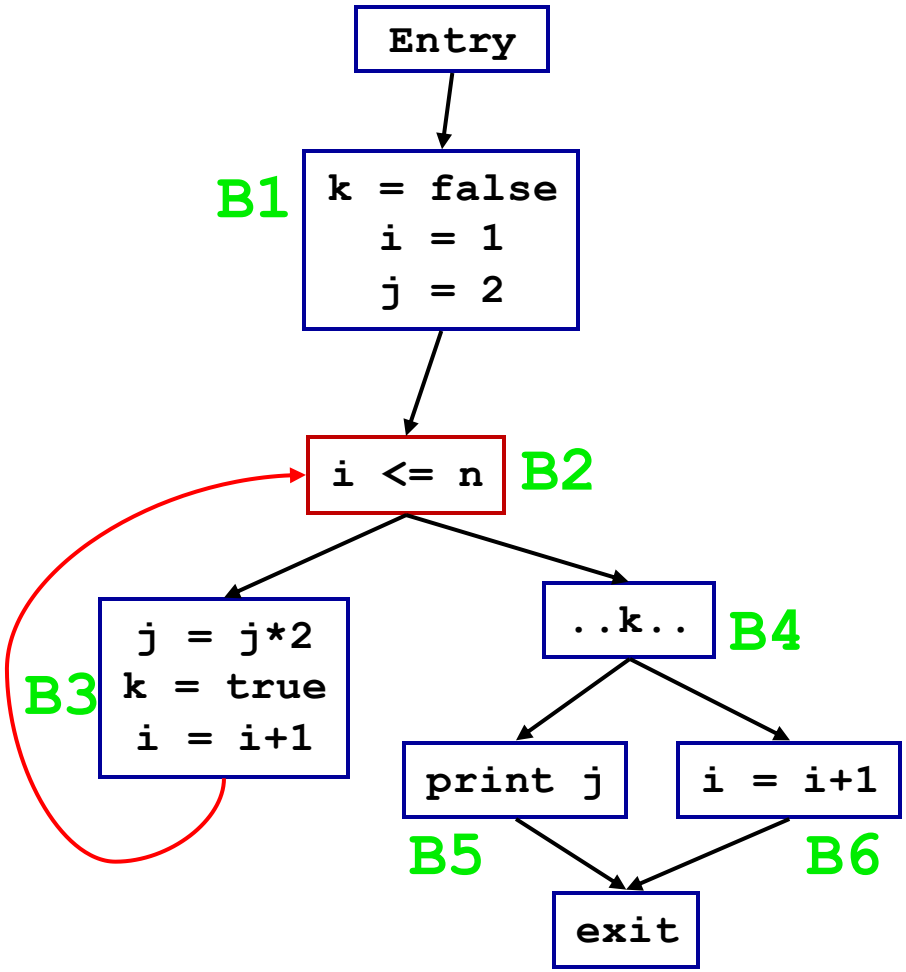
Recall: Dominators

- a dom b
  - node a dominates b if every possible execution path from entry to b includes a
- a sdom b
  - a strictly dominates b if a dom b and a != b
- a idom b
  - a immediately dominates b if a sdom b, AND there is no c such that a sdom c and c sdom b

Back edges and loop headers

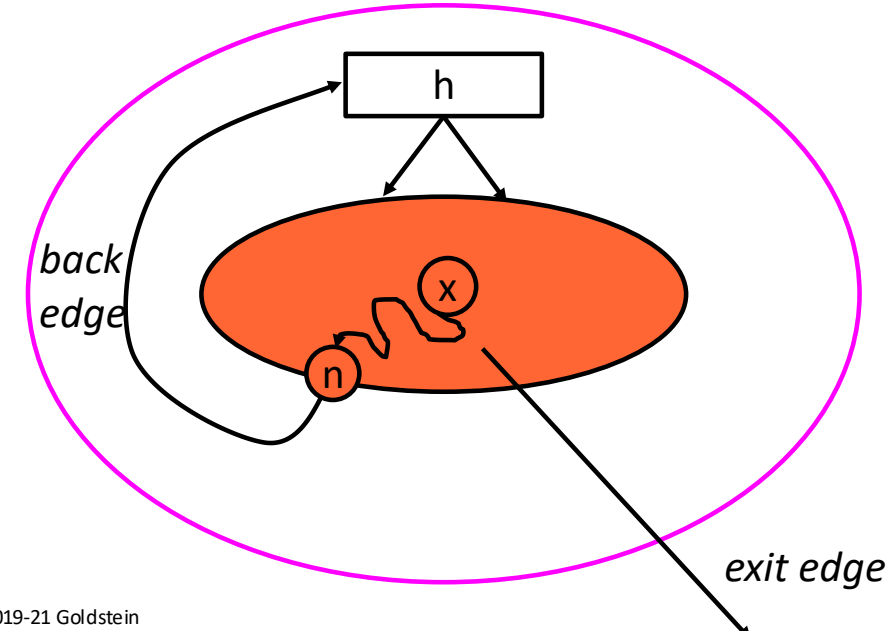
- A control-flow edge from node B3 to B2 is a back edge if
  B2 dom B3

- Furthermore, in that case node B2 is a loop header



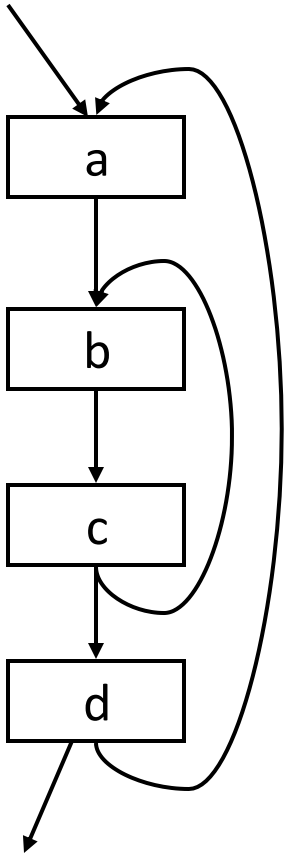
Natural loop

- Consider a back edge from node n to node h
- The natural loop of n→h is the set of nodes L such that for all x∈L:
  - h dom x and
  - there is a path from x to n not containing h



Examples

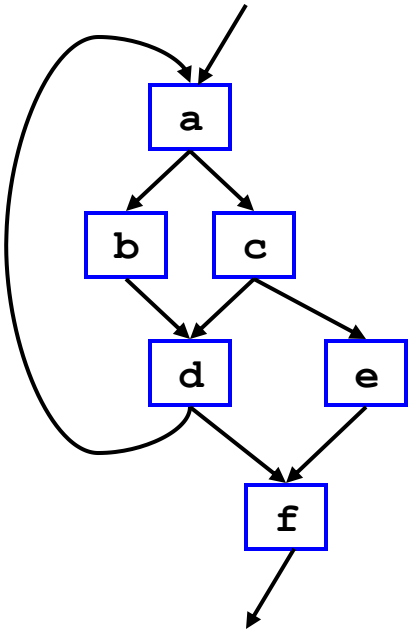
Simple example:



(often it's more complicated, since a FOR loop found in the source code might need an if/then guard)

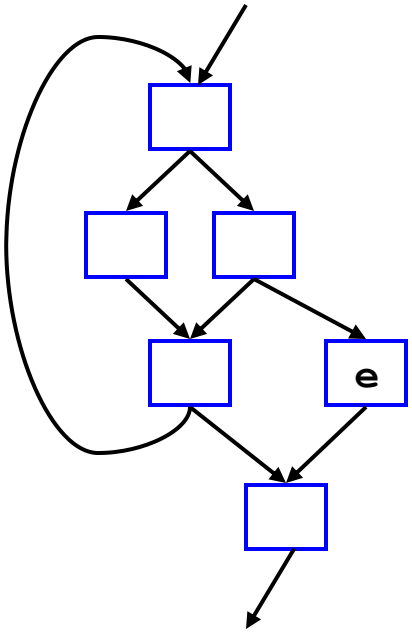
Examples

Try this:



Examples

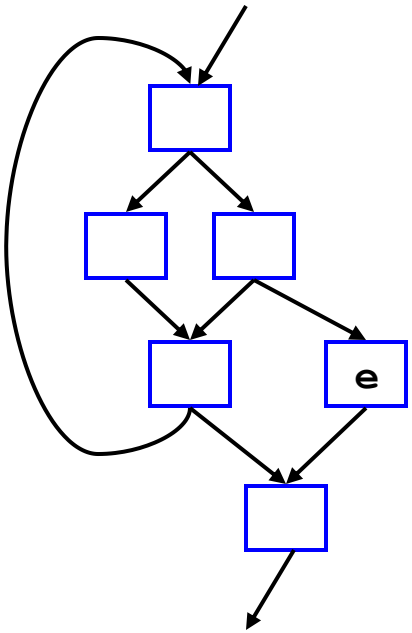
```
for (..) {
    if {
      …
  } else {
      …
   if (x) {
      e;
     break;
     )
    }
   }
```



Examples

```
for (..) {
    if {
      …
  } else {
      …
   if (x) {
       e;
     break;
      )
     }
    }
```

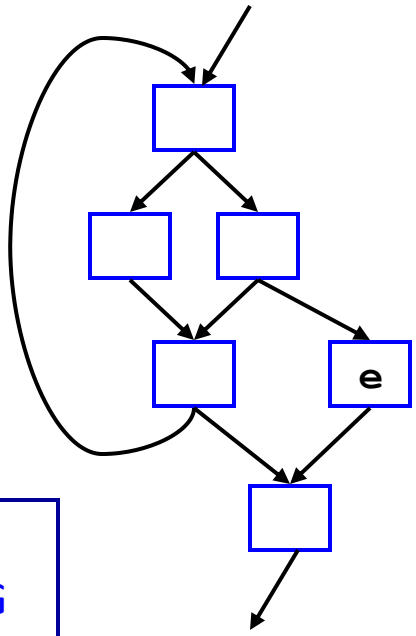
lexically, in loop, but not in natural loop



Examples

```
for (..) {
    if {
      …
  } else {
      …
   if (x) {
       e;
     break;
      )
     }
    }
```

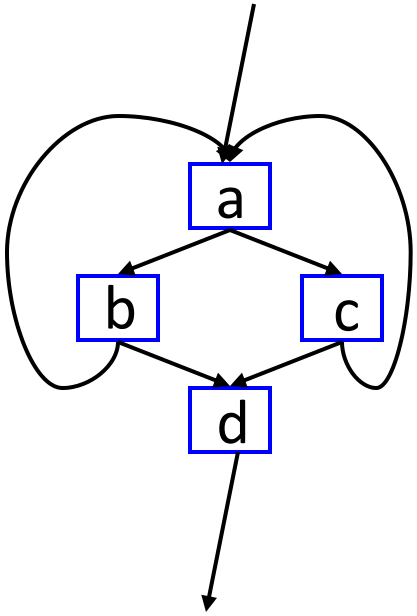
lexically, in loop, but not in natural loop



and another reason why CFG analysis is preferred over source/AST loops

Examples

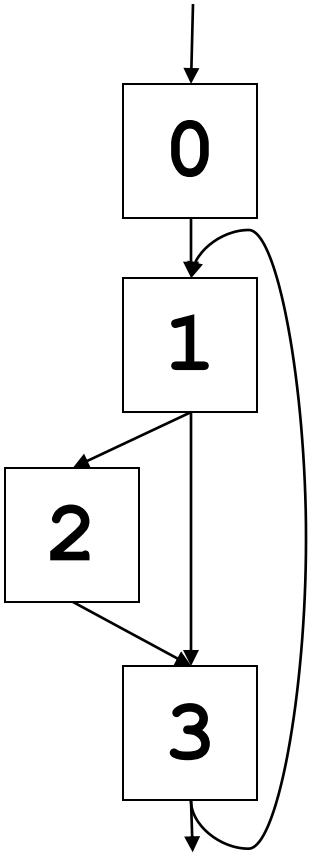
- Yes, it can happen in C



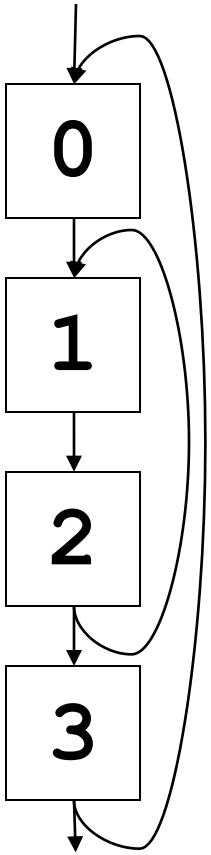
Natural Loops

One loop per header..

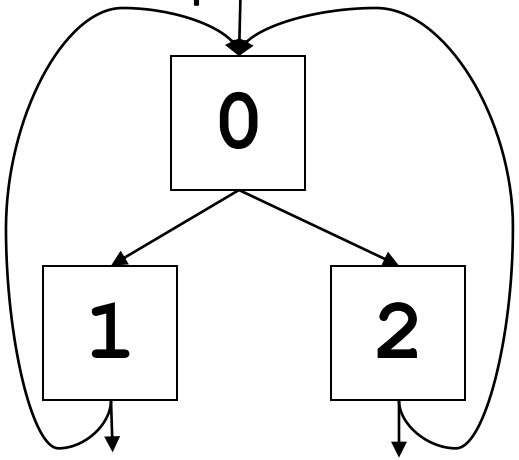
What are the natural loops?



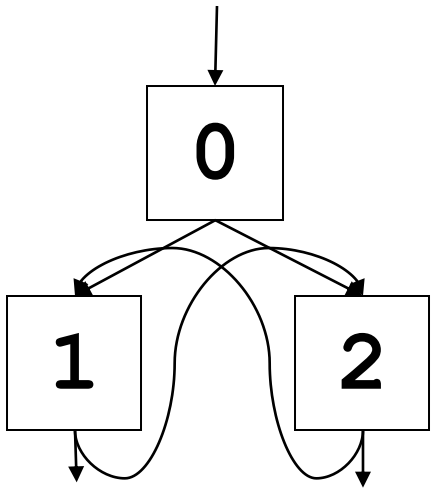
{1,2,3}



{1,2},{0,1,2,3}



{0,1,2}



{}

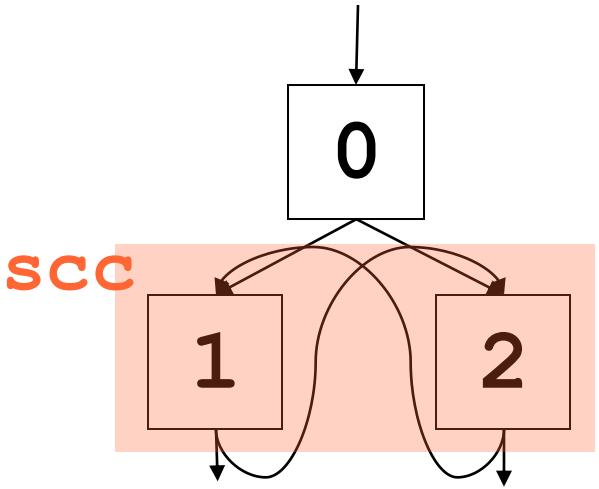
Nested Loops

- Unless two natural loops have the same header, they are either disjoint or nested within each other
- If A and B are loops (sets of blocks) with headers a and b such that $a \neq b$ and $b \in A$
  - $B \subset A$
  - loop B is *nested* within A
  - B is the *inner loop*
- Can compute the loop-nest tree

General Loops

- A more general looping structure is a *strongly connected component* of the control flow graph
  - subgraph <N_{scc},E_{scc}> such that

    every block in N_{scc} is reachable from every other node using only edges in E_{scc}



Not very useful definition of a loop

Reducible Flow Graphs

There is a special class of flow graphs, called reducible flow graphs, for which several code-optimizations are especially easy to perform.

In reducible flow graphs loops are unambiguously defined and dominators can be efficiently computed.

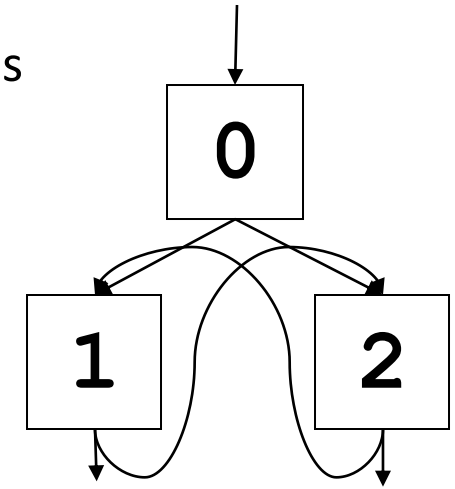
Reducible flow graphs

Definition: A flow graph G is reducible iff we can partition the edges into two disjoint groups, forward edges and back edges, with the following two properties.

1. The forward edges form an acyclic graph in which every node can be reached from the initial node of G.

2. The back edges consist only of edges whose heads dominate their tails.

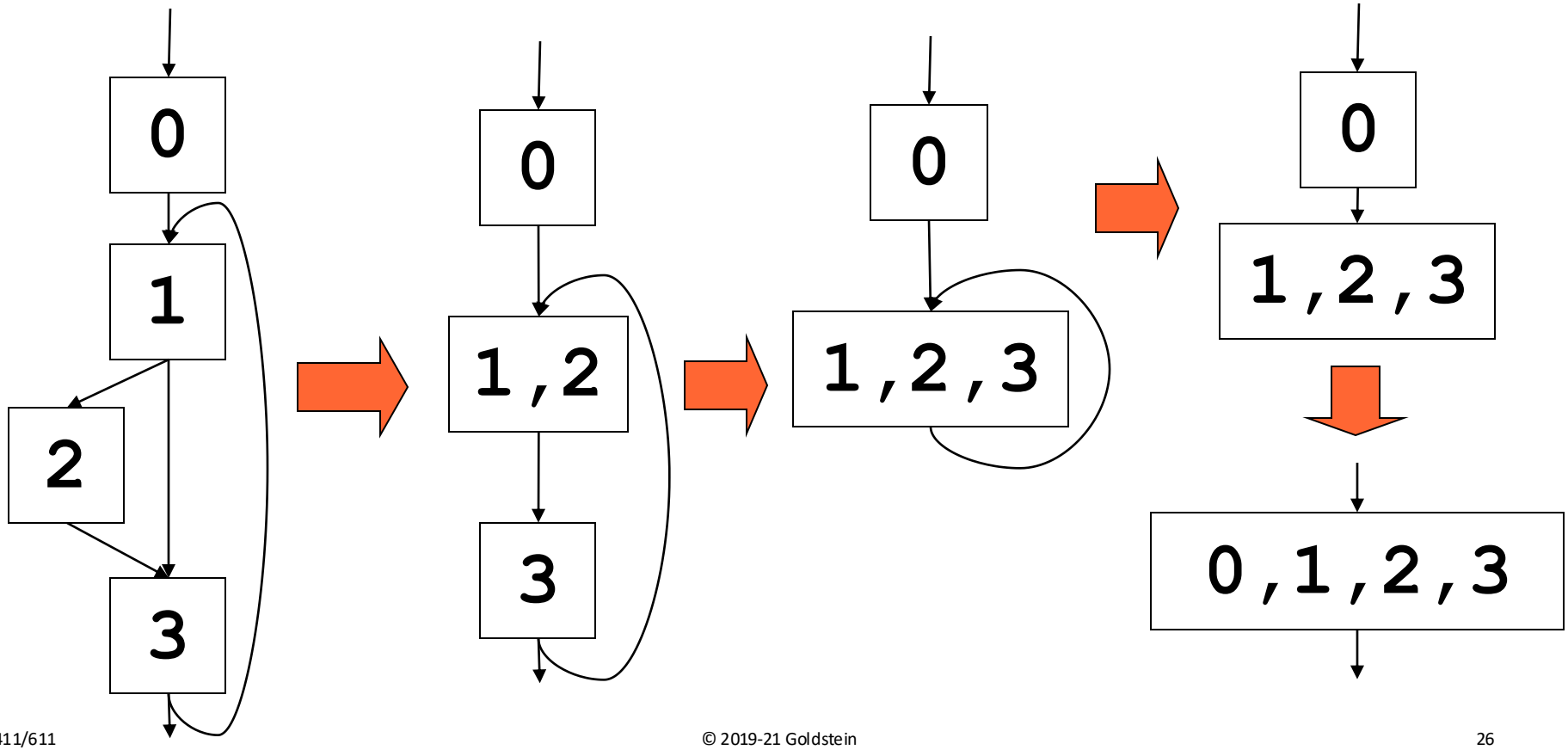
Why isn't this reducible?



This flow graph has no back edges. Thus, it would be reducible if the entire graph were acyclic, which is not the case.

Alternative definition

- Definition: A flow graph G is reducible if we can repeatedly collapse (reduce) together blocks (*x*,y) where x is the only predecessor of y (ignoring self loops) until we are left with a single node



Properties of Reducible Flow Graphs

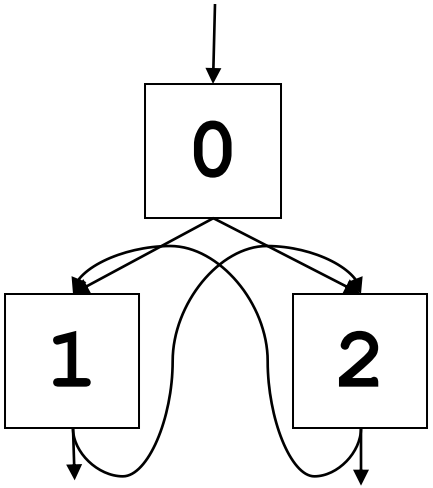
- In a reducible flow graph,
  all loops are natural loops
- Can use DFS to find loops
- Many analyses are more efficient
  - polynomial versus exponential

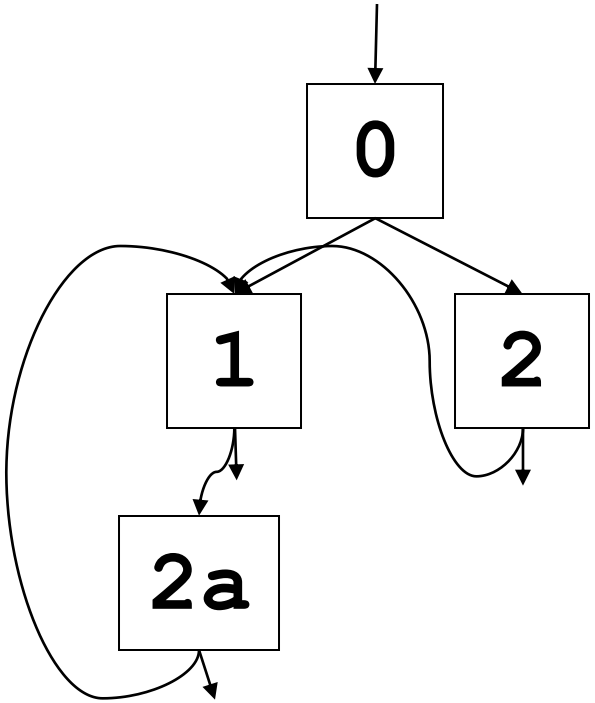
Good News

- Most flow graphs are reducible
- Languages prohibit irreducibility
  - goto free C
  - Java
  - C0 ☺
- Programmers usually don't use such constructs even if they're available
  - >90% of old Fortran code reducible

Dealing with Irreducibility

- Don't
- Can split nodes and duplicate code to get reducible graph
  - possible exponential blowup
- Other techniques…





Loop-invariant computations

- A definition

  t = x op y

  in a loop is (conservatively) loop-invariant if
  - x and y are constants, or
  - all reaching definitions of x and y are outside the loop, or
  - only one definition reaches x (or y), and that definition is loop-invariant
    - so keep marking iteratively

Loop-invariant computations

- If not in SSA Be careful

```
t = expr;
for () {
    s = t * 2;
    t = loop_invari
    x = t + 2;
    …
}
```

Of course, not an issue in SSA

```
    t1 = expr;
L1:
    brc L2;
    t2 = phi(t1, t3);
    s = t2 * 2;
    t3 = loop_invariant_expr;
    x1 = t3 * 2;
    …
    jmp L1;
L2:
```

- Even though t's two reaching expressions are each invariant, s is not invariant…

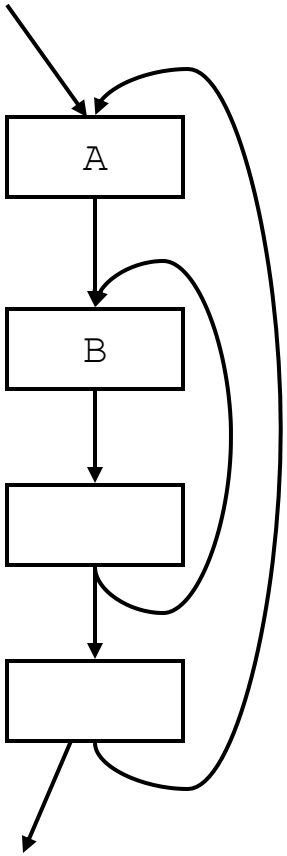
Hoisting

- In order to “hoist” a loop-invariant computation out of a loop, we need a place to put it

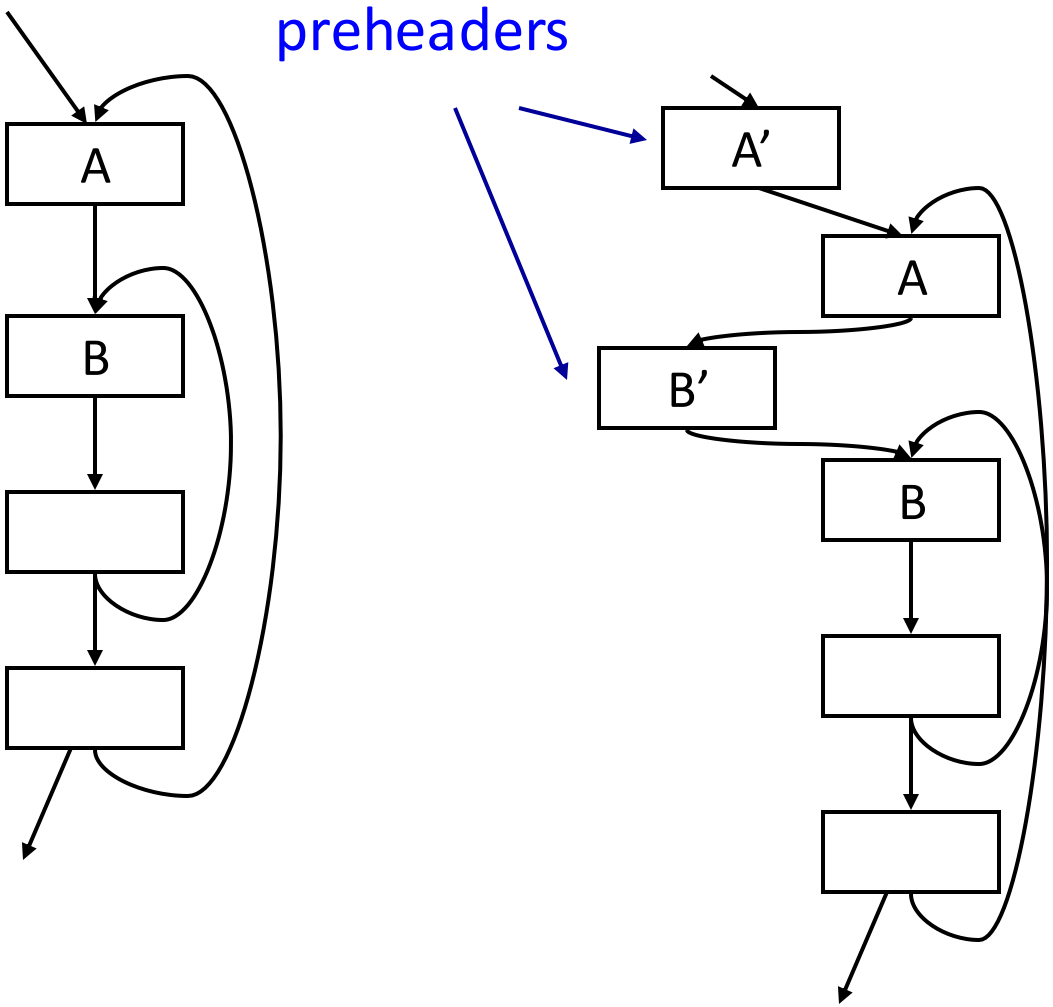
- We could copy it to all immediate predecessors (except along the back-edge) of the loop header...

- ...But we can avoid code duplication by ensuring there is a pre-header

Hoisting Uses Pre-Headers



Hoisting Uses Pre-Headers



Hoisting conditions

- For a loop-invariant definition

  d: t = x op y

- we can hoist d into the loop's pre-header only if
  1. d's block dominates all loop exits at which t is live-out, and
  2. d is the only definition of t in the loop, and
  3. t is not live-out of the pre-header

We need to be careful...

- All hoisting conditions must be satisfied!

```
L0:
 t = 0
L1:
 i = i + 1
 t = a * b
 M[i] = t
 if i<N goto L1
L2:
 x = t
```

OK

```
L0:
 t = 0
L1:
 if i>=N goto L2
 i = i + 1
 t = a * b
 M[i] = t
 goto L1
L2:
 x = t
```

violates 1,3

```
L0:
 t = 0
L1:
 i = i + 1
 t = a * b
 M[i] = t
 t = 0
 M[j] = t
 if i<N goto L1
L2:
```

violates 2

We need to be careful...

- All hoisting conditions must be satisfied!

```
L0:
 t = 0
L1:
 i = i + 1
 t = a * b
 M[i] = t
 if i<N goto L1
L2:
 x = t
```

OK

```
L0:
 t = 0
L1:
 if i>=N goto L2
 i = i + 1
 t = a * b
 M[i] = t
 goto L1
L2:
 x = t
```

violates 1,3

```
L0:
 t = 0
L1:
 i = i + 1
 t = a * b
 M[i] = t
 t = 0
 M[j] = t
 if i<N goto L1
L2:
```

violates 2

LICM subsumed by PRE

- Don’t have to implement Loop invariant code motion if you implement PRE, since PRE subsumes it anyway!
- (But, PRE is difficult)
- (And, weird in SSA)

Loop optimizations: Induction-variable Strength reduction

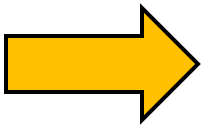
The basic idea of IVE

- Suppose we have a loop variable
  - i initially 0; each iteration i = i + 1
- and a variable that linearly depends on it:

  x = i * c1 + c2
- In such cases, we can try to
  - initialize x = i_o * c1 + c2 (execute once)
  - increment x by c1 each iteration

Simple Example of IVE

```
for i = 0 to n
    a[i] = 0
```

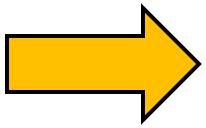


```
        i <- 0
H:
        if i >= n goto exit
        j <- i * 4
        k <- j + a
        M[k] <- 0
        i <- i + 1
        goto H
```

Clearly, j & k do not need to be computed anew each time since they are related to i and i changes linearly.

Simple Example of IVE

```
        i <- 0
H:
        if i >= n goto exit
        j <- i * 4
        k <- j + a
        M[k] <- 0
        i <- i + 1
        goto H
```

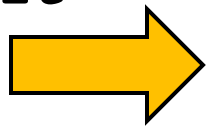


```
        i <- 0
        j' <- 0
        k' <- a
H:
        if i >= n goto exit
        j <- j'
        k <- k'
        M[k] <- 0
        i <- i + 1
        j' <- j' + 4
        k' <- k' + 4
        goto H
```

But, then we don't even need j (or j')

Simple Example of IVE

```
        i <- 0
        j' <- 0
        k' <- a
H:
        if i >= n goto exit
        j <- j'
        k <- k'
        M[k] <- 0
        i <- i + 1
        j' <- j' + 4
        k' <- k' + 4
        goto H
```



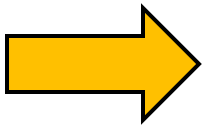
```
        i <- 0
        k' <- a
H:
        if i >= n goto exit
        k <- k'
        M[k] <- 0
        i <- i + 1
        k' <- k' + 4
        goto H
```

Do we need i?

Simple Example of IVE

Rewrite comparison

```
        i <- 0
        k' <- a
H:
        if i >= n goto exit
        k <- k'
        M[k] <- 0
        i <- i + 1
        k' <- k' + 4
        goto H
```



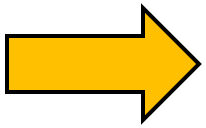
```
        i <- 0
        k' <- a
H:
        if k' >= a+(n*4) goto exi
        k <- k'
        M[k] <- 0
        k' <- k' + 4
        goto H
```

But, a+(n*4) is loop invariant

Simple Example of IVE

Invariant code motion on a+(n*4)

```
        i <- 0
        k' <- a
H:
        if k' >= a+(n*4)goto exit
        k <- k'
        M[k] <- 0
        k' <- k' + 4
        goto H
```



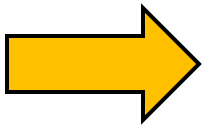
```
        k' <- a
        n' <- a + (n * 4)
H:
        if k' >= n' goto exit
        k <- k'
        M[k] <- 0
        k' <- k' + 4
        goto H
```

now, we do copy propagation and eliminate k

Simple Example of IVE

Copy propagation

```
        k' <- a
        n' <- a + (n * 4)
H:
        if k' >= n' goto exit
        k <- k'
        M[k] <- 0
        k' <- k' + 4
        goto H
```



```
        k' <- a
        n' <- a + (n * 4)
H:
        if k' >= n' goto exit
        M[k'] <- 0
        k' <- k' + 4
        goto H
```

Voila!

Simple Example of IVE

Compare original and result of IVE

```
    i <- 0
H:
    if i >= n goto exit
    j <- i * 4
    k <- j + a
    M[k] <- 0
    i <- i + 1
    goto H
```

```
    k' <- a
    n' <- a + (n * 4)
H:
    if k' >= n' goto exit
    M[k'] <- 0
    k' <- k' + 4
    goto H
```

Voila!

What we did

- identified induction variables (i,j,k)
- strength reduction (changed * into +)
- dead-code elimination (j <- j')
- useless-variable elimination (j' <- j' + 4)
  (This can also be done with ADCE)
- loop invariant identification & code-motion
- almost useless-variable elimination (i)
- copy propagation

Is it faster?

- On some hardware, adds are much faster than multiplies
- Fewer instructions (better $ behavior)
- Furthermore, one fewer value is computed,
  - thus potentially saving a register
  - and decreasing the possibility of spilling

Loop preparation

- Before attempting IVE, it is best to first perform :
  - constant propagation & constant folding
  - copy propagation
  - loop-invariant hoisting

How to do it, step 1

- First, find the basic IVs
  - scan loop body for defs of the form

    x = x + c or x = x – c

    where c is loop-invariant
  - record these basic IVs as

    x = (x, 0, c)
  - this represents the IV: x = x * c

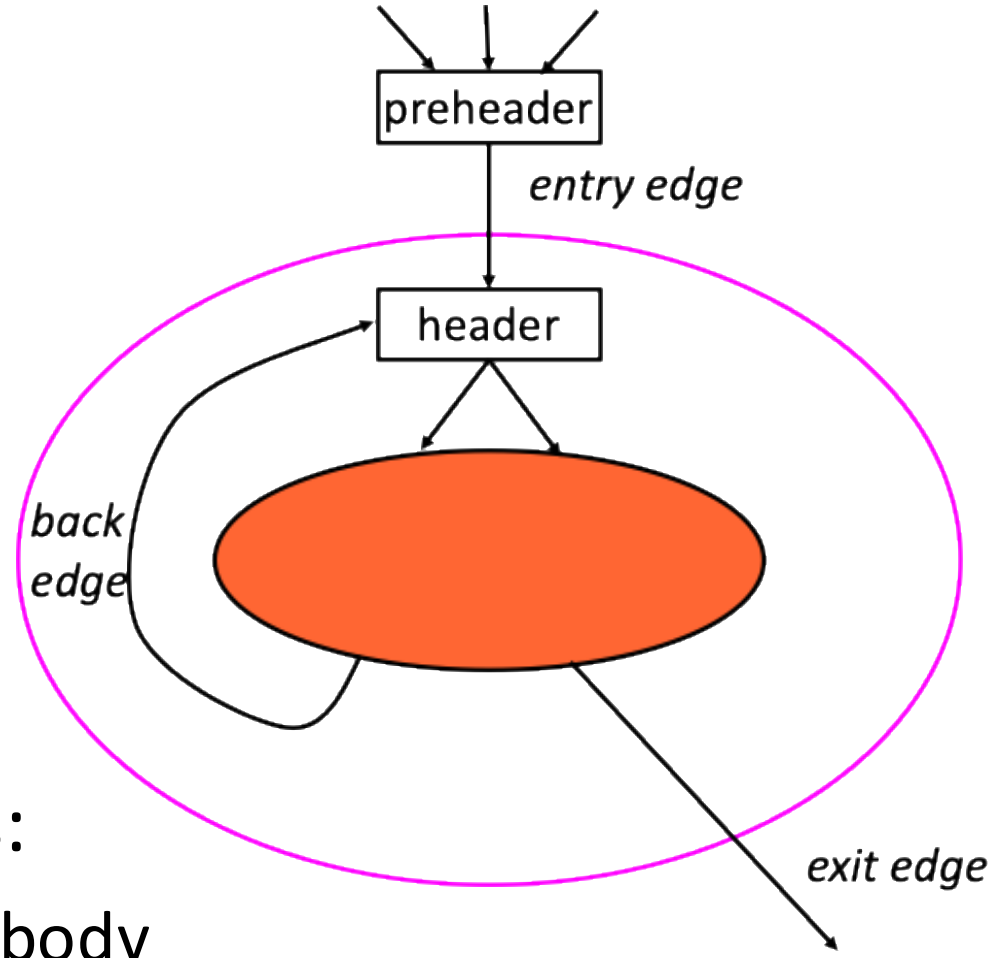
Take a step back and consider problem in SSA

Finding Induction Variables

- Important:
  - Induction variable elimination
  - Strength reduction
  - Address recurrence elimination
  - Useless variable elimination
  - Loop unrolling
  - Bounds checking
- Finding in SSA
  - Basic IV: updated by a loop-invariant step
  - Derived IV: Affine function of a basic IV

Use Shape of Natural Loop

- Single preheader
- Single loop header
- Single backedge

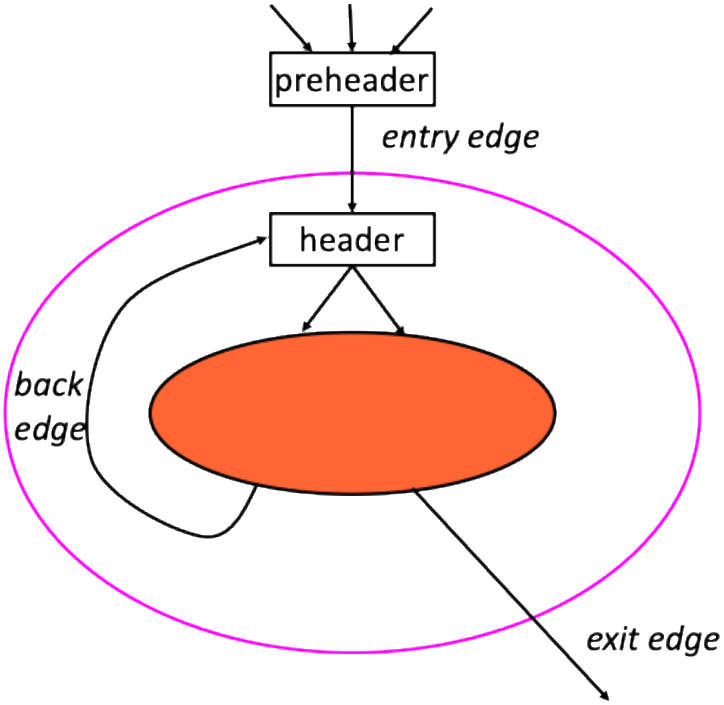


- Canonical Loop has:
  - Header dominates body
  - Φs in header are loop-carried dependencies

Simple IV Example

```
preheader:
   i0 ← 0
   goto header

header:
   i1 ← Φ(i0,i2)
   …
   i2 ← ii + 1
   if (?)goto header else exit
```



Basic Induction Variable

- **i** is basic IV if
  - **i** result of ϕ in header
  - **i = Φ(init , back)**
  - **init** comes from outside loop
  - **back** comes from backedge
  - **back** computed from **i** by ± loop-invariant

- Represent as (i, 1, 0)

```
preheader:
  i0 ← 0
  goto header

header:
  i1 ← Φ(i0,i2)
  …
  i2 ← ii + 1
  if (?)goto header else exit
```

Derived Induction Variable

- All IV represented as (family, offset, multiple)
- `j` is derived IV if
  - `j` an affine function of i, j=a*i + b
  - represented as (i, b, a)
  - j is in the family of i

Representing IVs

- Characterize all induction variables by:

  (base-variable, offset, multiple)

  – where the offset and multiple are loop-invariant

- IOW, after an induction variable is defined it equals:

  offset + multiple * base-variable

How to do it, step 2

- Scan for derived IVs of the form

  k = i * c1 + c2

  – where i is a basic IV,
  and
  c1 and c2 are loop invariant
- We say k is in the family of i
- Record as k = (i, c2, c1)

How to do it, step 3

- Iterate, looking for derived IVs of the form

  k = j * c1 + c2

  - where IV j =(i, a, b), and
  - c1 and c2 are loop invariant
- Record as k = (i, a*c1, b*c1+c2)

Simple Example of IVE

```
        i0 <- 0
H:
        i1 = Φ(i0, i2)
        if i1 >= n goto exit
        j <- i1 * 4
        k <- j + a
        M[k] <- 0
        i2 <- i1 + 1
        goto H
```

i: (i_1, 0, 1) i.e., i = 0 + 1 * i

j: (i_1, 0, 4) i.e., j = 0 + 4 * i

k: (i_1, a, 4) i.e., k = a + 4 * i

So, j & k are in family of i

IV Optimizations

- Once we have identified all IVs and recorded their tuples, we perform 3 optimizations:
  - strength reduction
  - useless-variable elimination
  - Comparison rewriting

Strength Reduction

- For an induction variable k = (i, c1, c2)
  - initialize k_0 = i * c2 + c1 in the preheader
  - Add phi in header, k_1=Φ(k0,k2)
  - replace k with k_1
  - Before back-edge add

    $k_2 = k_1 + c2$

Notes

- Are the c1, c2 constant, or just invariant?
  - if constant, then you can keep folding them: they're always a constant even for derived IVs
  - otherwise, they can be expressions of loop-invariant variables

Is it faster? (2)

- On some hardware, adds are much faster than multiplies
- But…not always a win!
  - Constant multiplies might otherwise be reduced to shifts/adds that result in even better code than IVE
  - Scaling of addresses (i*4) might come for free on your processor's address modes
- So maybe: only convert **`i*c1+c2`** when c1 is loop invariant but not a constant

Next Up

- Loop Unrolling
- Dependence analysis

Common loop optimizations

- Hoisting of loop-invariant computations
  - pre-compute before entering the loop
- Elimination of induction variables
  - change p=i*w+b to p=b,p+=w, when w,b invariant
- Loop unrolling
  - to to improve scheduling of the loop body
- Software pipelining
  - To improve scheduling of the loop body
- Loop permutation
  - to improve cache memory performance

Requires understanding data dependencies