Loops

or

Lather, Rinse, Repeat…

CS4410: Spring 2013

Program Loops

- Reading: Appel Ch. 18

- Loop = a computation repeatedly executed until a terminating condition is reached

- High-level loop constructs:
  - While loop: while (e) s;
  - For loop: for(i=0; i<u; i+=c) s;

Program Loops

- Why are loops important?
  - Most of the execution time is spent in loops
  - Typically: 90/10 rule, 10% code is a loop

- Therefore, loops are important targets of optimization

Loop Optimizations:

So we want techniques for improving them

- Low-level optimization:
  - Moving around code in a single loop
  - usually performed at 3-addr code stage or later
  - e.g., loop invariant removal, induction variable strength reduction & elimination, loop unrolling
- High-level optimization:
  - Restructuring loops, often affects multiple loops
  - e.g., loop fusion, loop interchange, loop tiling

Example: 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: 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: 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: induction variable

```
L0:  i := 0                /*  s=0;                   */
     s := 0                /*  for(i=0;i<100;i++)*/
     jump L2               /*    s += a[i];           */
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:  ...
```

Example: induction variable

```
L0:  i := 0                /*  s=0;                     */
     s := 0                /*  for(i=0;i<100;i++)*/
     jump L2               /*     s += a[i];            */
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:  ...
```

Note: t1 == i*4

at each point in loop

Example: 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: induction variable

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

Example: induction variable

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

Notice t1 no longer used!

Example: 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: 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:   ...
```

Example: 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:  ...
```

Example: 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:  ...
```

Gotta find loops first:

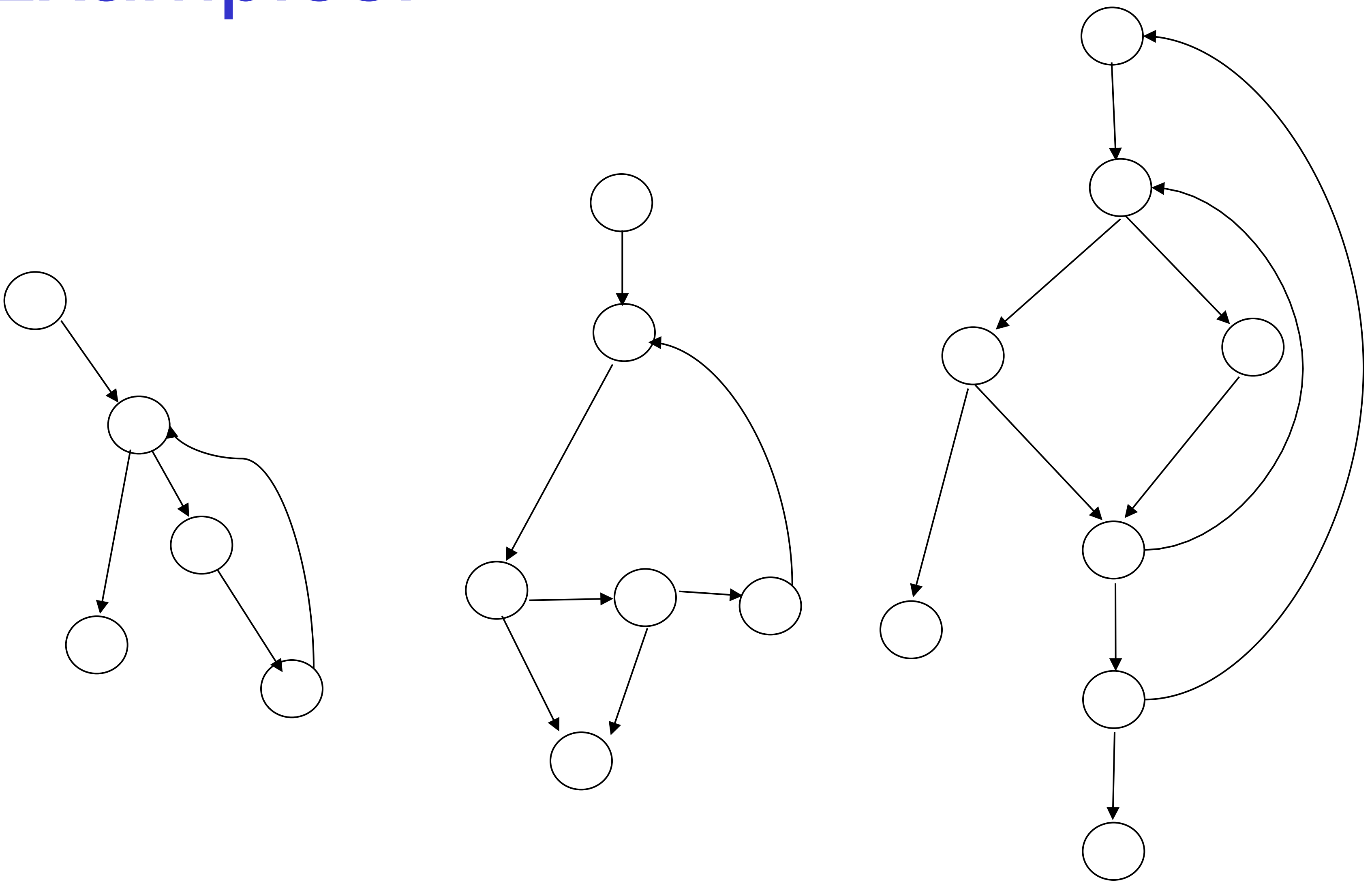
What is a loop?

- can't just "look" at graphs
- we're going to assume some additional structure

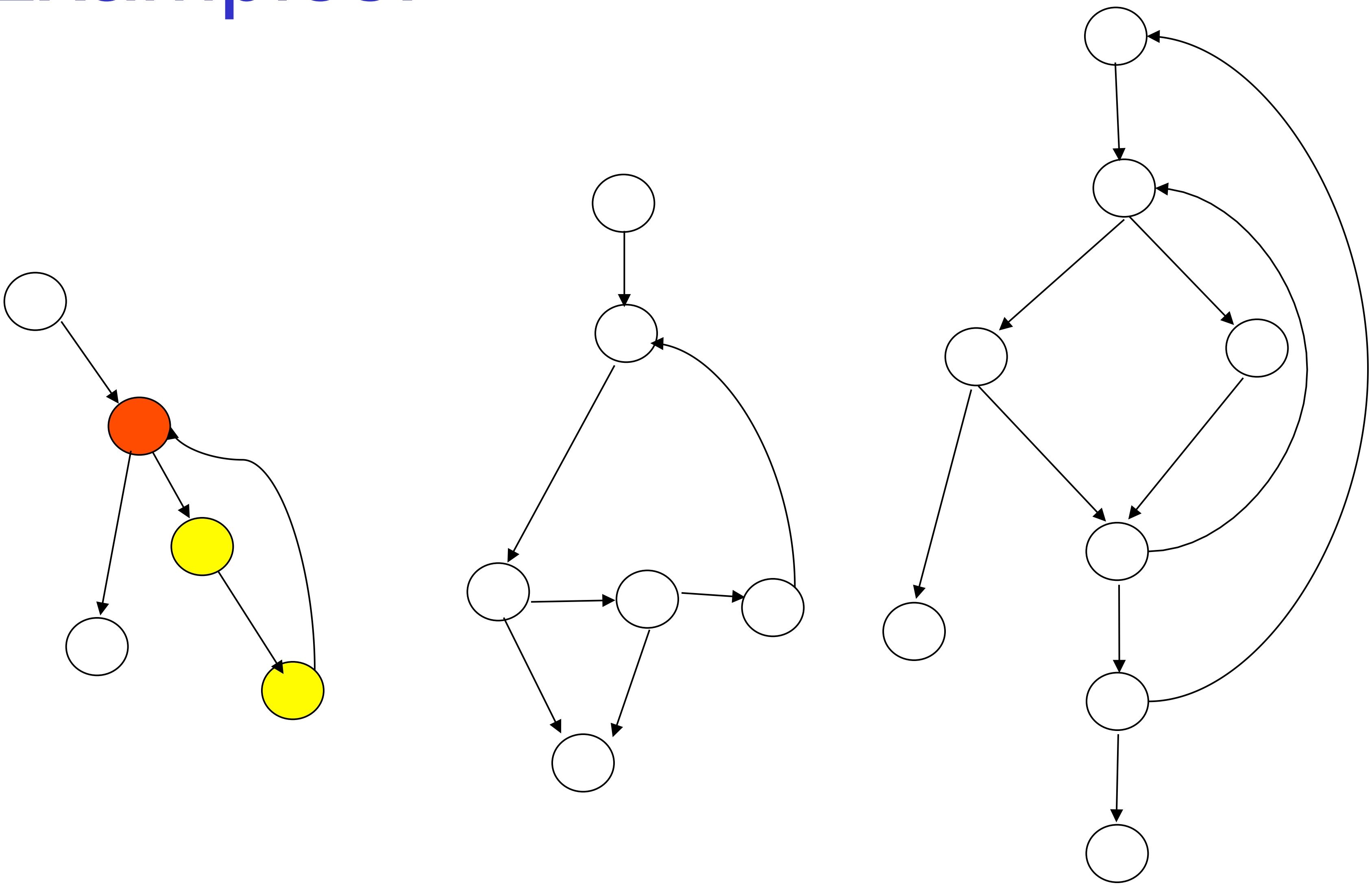
Defn: a *loop* is a subset S of nodes where:

- there is a distinguished *header* node h
- you can get from h to any node in S
- you can get from any node in S to h
- there's no edge from a node outside S to any other node than h.

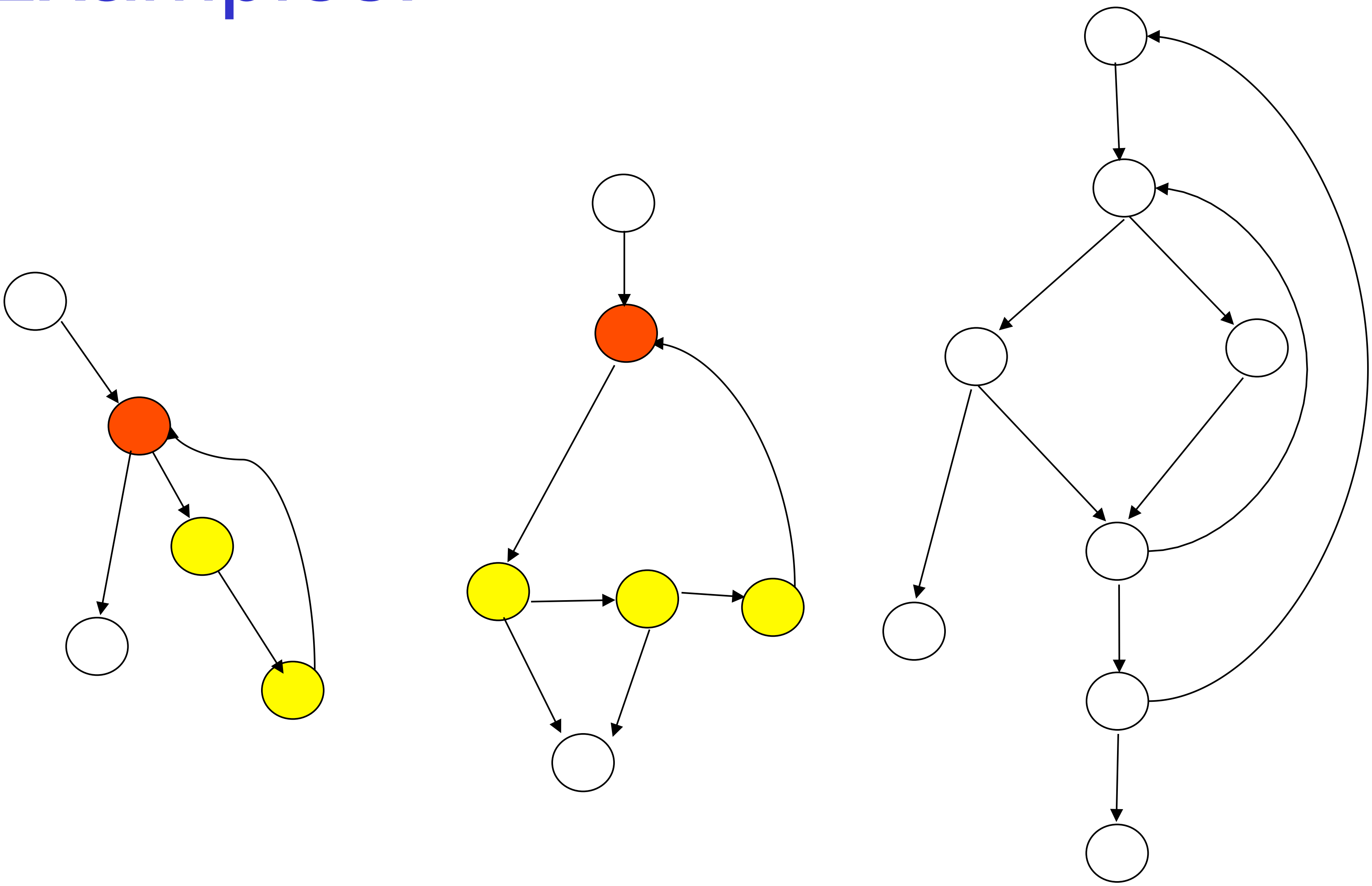
Examples:



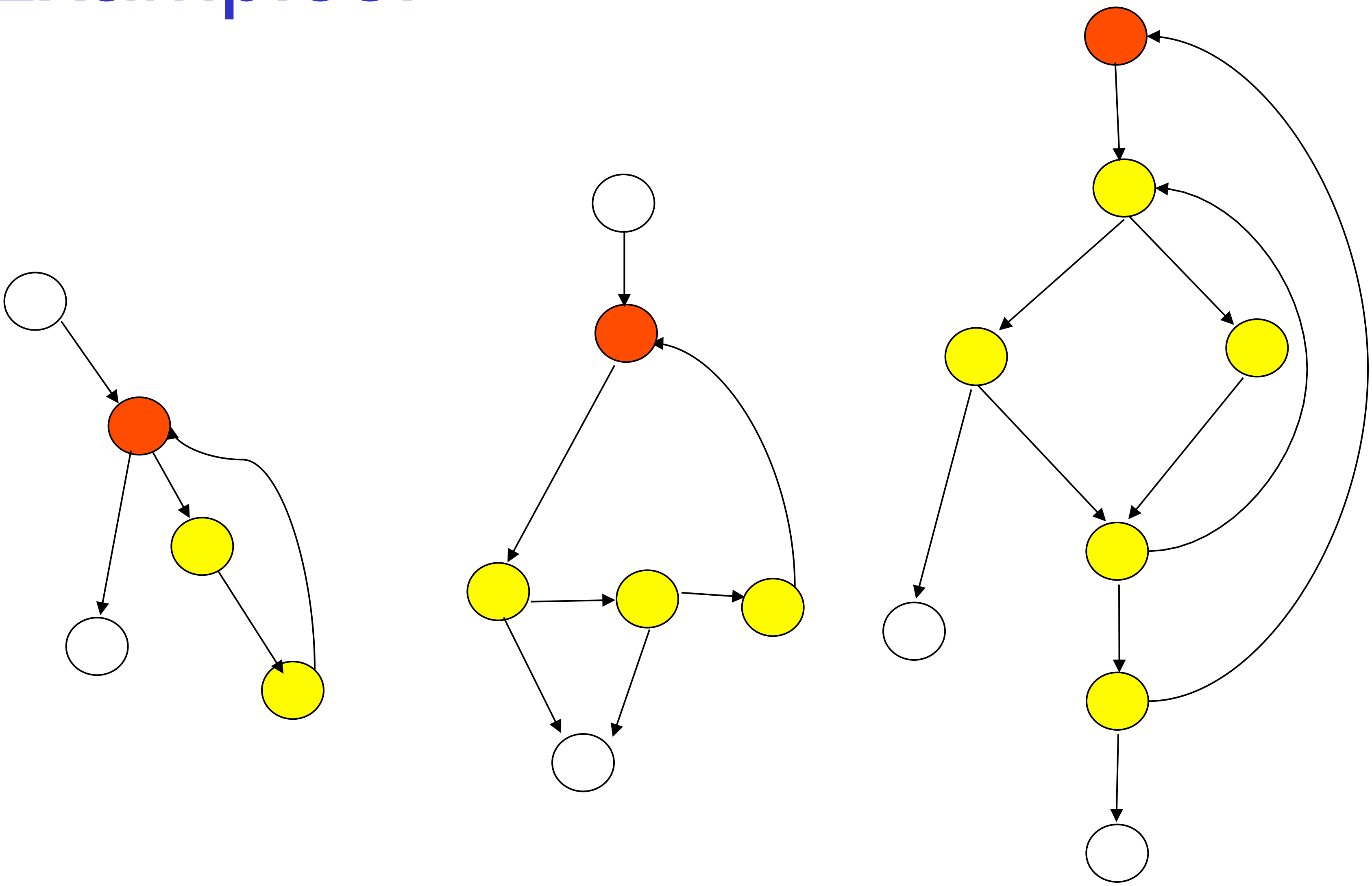
Examples:



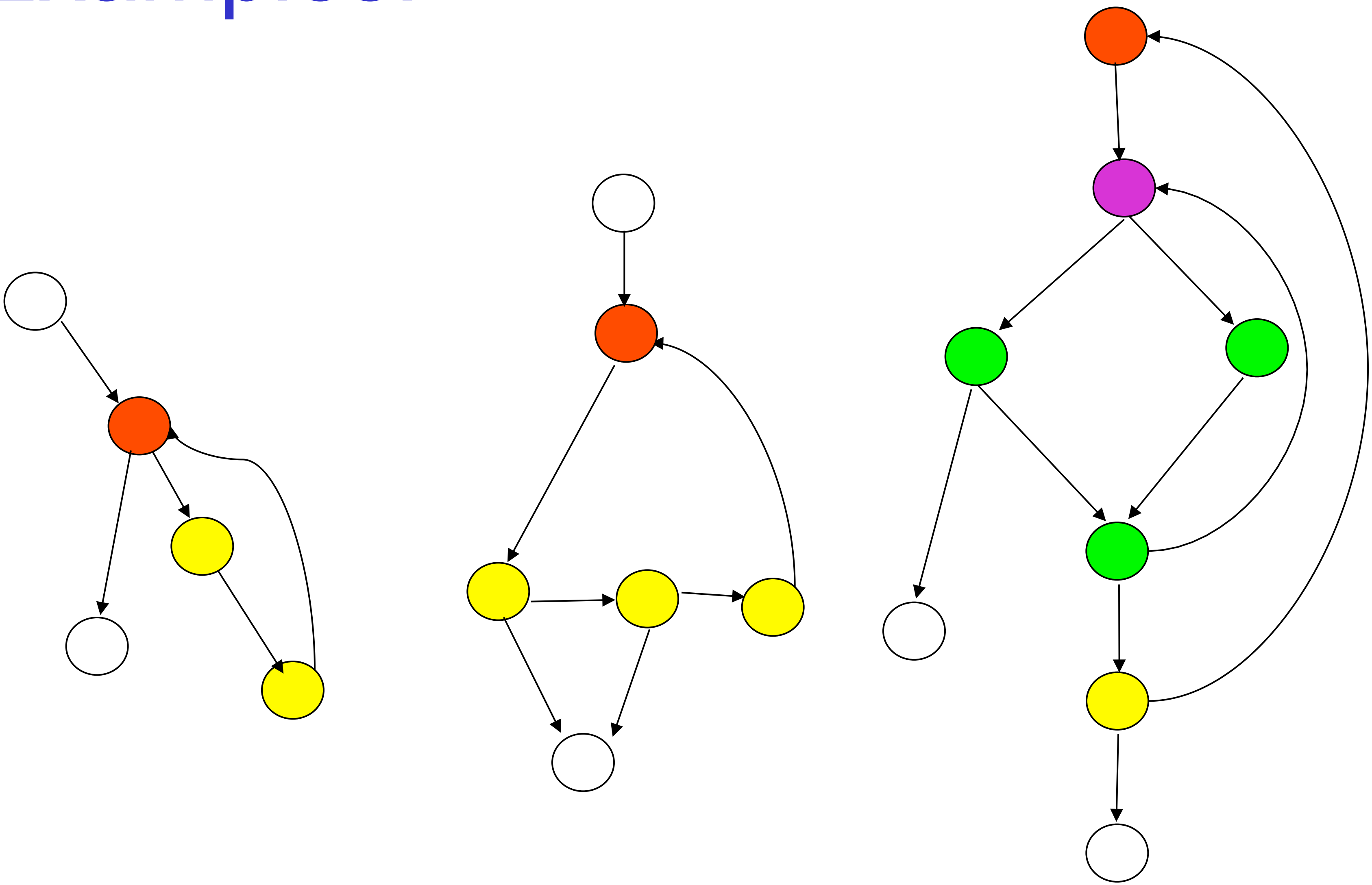
Examples:



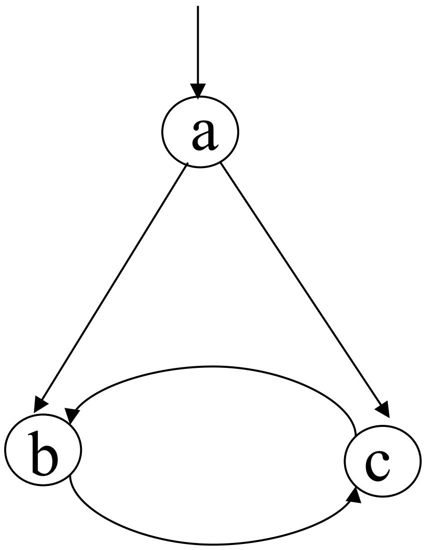
Examples:



Examples:

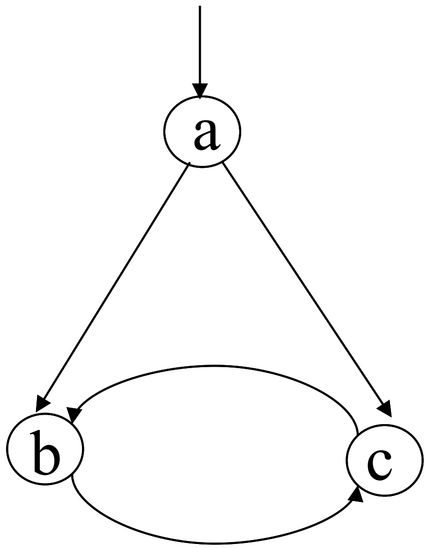


Consider:



Does it have a "loop"?

This graph is called *irreducible*



- a can't be header: no edge from c or b to it.
- b can't be header: c has outside edge from a.
- c can't be header: b has outside edge from a.

According to our definition, no loop.
But obviously, there's a cycle…

Reducible Flow Graphs

So why did we define loops this way?

- 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

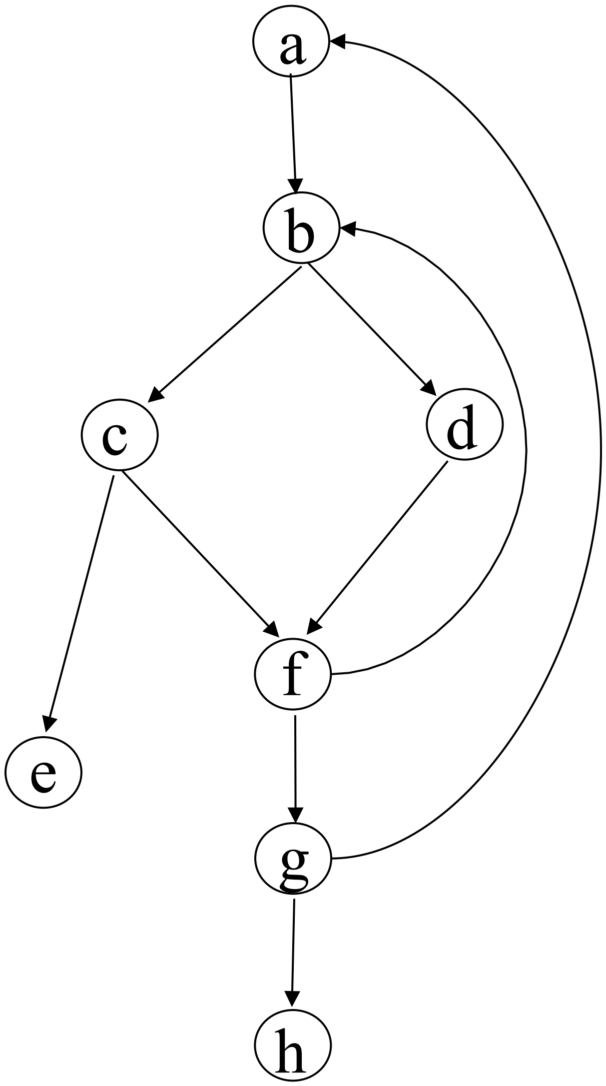
Defn: node d *dominates* node n if every path from the start node to n must go through d.

Defn: an edge from n to a dominator d is called a *back-edge*.

Defn: a *natural 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 that's how we find loops!

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?
f->b, g->a

loops?

Calculating Dominators:

D[n] : the set of nodes that dominate n.

D[n0] = {n0}

$D[n] = \{n\} \cup (D[p_1] \cap D[p_2] \cap \ldots \cap D[p_m])$

where $pred[n] = \{p_1, p_2, \ldots, p_n\}$

It's pretty easy to solve this equation.

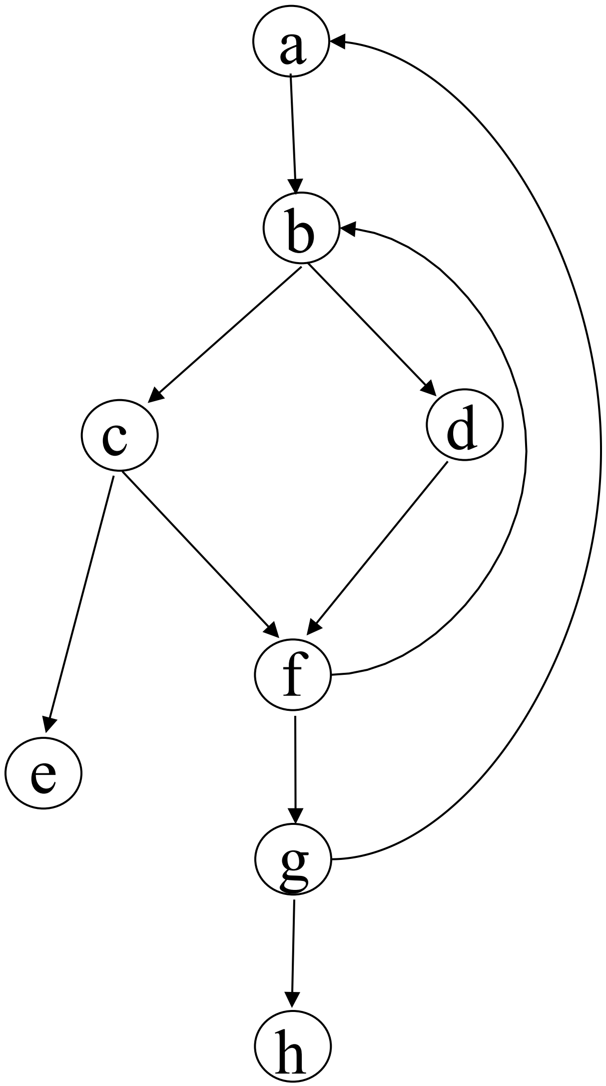
- start off assuming
  - D[n0] = {n0} (where n0 is start node, with no predecessors)
  - D[n] = all nodes (where n is not the start node)
- iteratively update D[n] based on predecessors until you reach a fixed point.

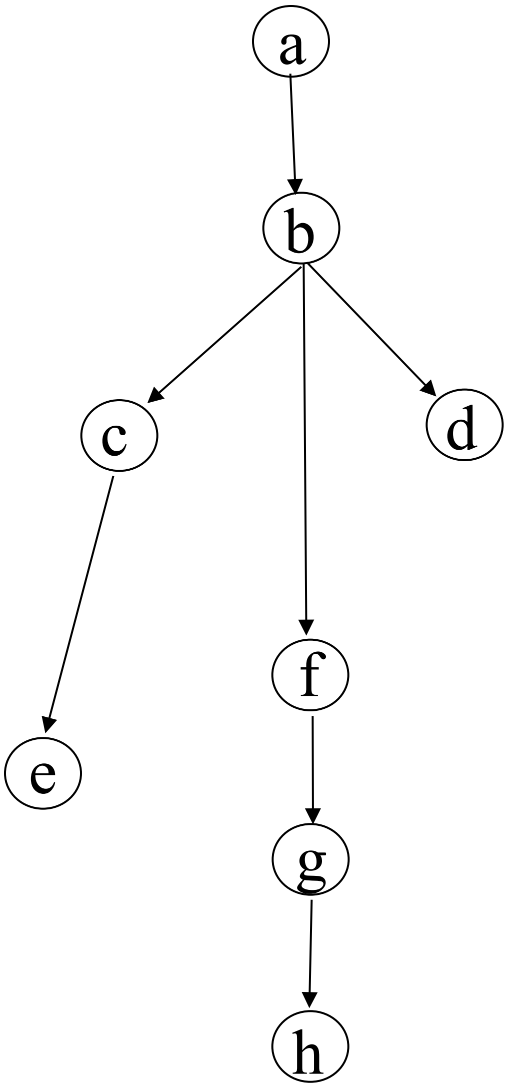
Representing Dominators

- We don't actually need to keep around the set of all dominators for each node.
- Instead, we construct a *dominator tree*.
  - if both d and e dominate n, then either d dominates e or vice versa.
  - that tells us there is a "closest" or *immediate dominator*.

Example:

Immediate Dominator Tree





Nested Loops

- If loops A & B have headers a & b s.t. a != b and a dominates b, and all of the nodes in B are a subset of nodes in A, then we say B is *nested* within A.

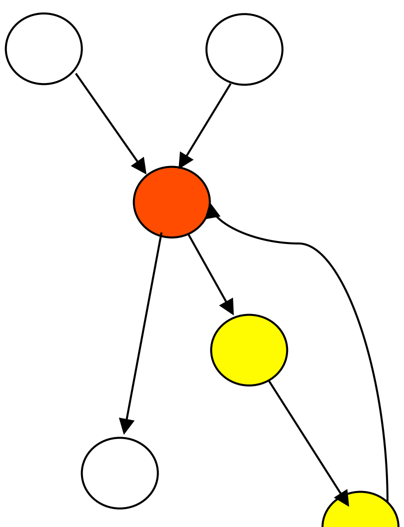
- We usually concentrate our attention on nested loops first (since we spend more time in them.)

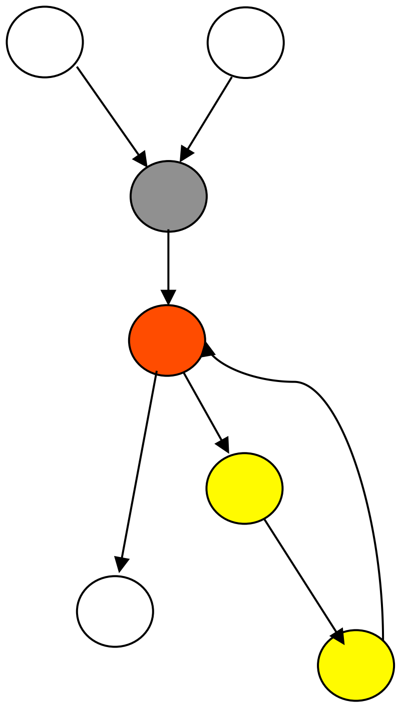
Disjoint and Nested Loops

- Property: for any two natural loops in a flow graph, one of the following is true:
  1. They are disjoint
  2. They are nested
  3. They have the same header
- Eliminate alternative 3: if two loops have the same header and none is nested in the other, combine all nodes into a single loop.

Loop Preheader

- Several optimizations add code before header
- Insert a new basic block (called preheader) in the CFG to hold this code





Loop Optimizations

- Now we know the loops

- Next: optimize these loops
  - Loop invariant code motion
  - Strength reduction of induction variables
  - Induction variable elimination

Loop Invariant Computation

A definition x:=… *reaches* a control-flow point if there is a path from the assignment to that point that contains no other assignment to x.

An assignment $x := v_1 \oplus v_2$ is *invariant* for a loop if for both operands v_1 & v_2 either

- they are constant, or
- all of their definitions that reach the assignment are outside the loop, or
- only one definition reaches the assignment and it is loop invariant.

Example:

```
L0:   t := 0

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

L2:   x := t
```

Calculating Reaching Defn's:

Assign a unique id to each definition.

Define defs(x) to be the set of all definitions of the temp x.

	Gen	Kill
$d : x := v_1 \oplus v_2$	{d}	defs(x) - {d}
$d : x := v$	{d}	defs(x) - {d}
<everything else>	{ }	{ }

$\text{DefIn}[n] = \text{DefOut}[p_1] \cap \ldots \cap \text{DefOut}[p_n]$
where $\text{Pred}[n] = \{p_1, \ldots, p_n\}$

$\text{DefOut}[n] = \text{Gen}[n] \cup (\text{DefIn}[n] - \text{Kill}[n])$

Hoisting / Code Motion

We would like to *hoist* invariant computations out of the loop.

But this is trickier than it sounds:

- We have already dealt with problem of where to place the hoisted statements by introducing preheader nodes
- Even then, we can run into trouble…

Valid Hoisting:

```
L0:   t := 0

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

L2:   x := t
```

Valid Hoisting:

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

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

L2:   x := t
```

Invalid Hoisting:

```
L0:   t := 0

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

L2:   x := t
```

t's definition is loop invariant but hoisting it conflicts with this use of the old t.

Conditions for Safe Hoisting:

An invariant assignment d:x:= $v_1 \oplus v_2$ is safe to hoist if:

- d dominates all loop exits at which x is *live-out,* and
- there is only one definition of x in the loop, and
- x is not live-out at the entry point for the loop (the pre-header.)

Induction Variables

- An induction variable is a variable in a loop, whose value is a function of the loop iteration number: $v = f(i)$
- In compilers, this is a linear function:

  $$f(i) = c*i + d$$

- Observation: linear combinations of linear functions are linear functions
  - Consequence: linear combinations of induction variables are induction variables

Families of Induction Variables

- *Basic induction variable*: a variable whose only definition in the loop body is of the form i = i + c (where c is loop invariant)

- *Derived induction variables*: Each basic induction variable i defines a family of induction variables Fam(i)
  - i in Fam(i)
  - k in Fam(i) if there is only one defn of k in the loop body, and it has the form k = j*c or k = j+c, where
    - j in Fam(i)
    - c is loop invariant
    - The only defn of j that reaches defn of k is in the loop
    - There is no defn of I between the defns of j and k

Induction Variables

```
      s := 0
      i := 0
L1:   if i >= n goto L2
      j := i*4
      k := j+a
      x := *k
      s := s+x
      i := i+1
L2:   ...
```

We can express **j** & **k** as linear functions of **i**:

```
j = 4*i + 0
k = 4*i + a
```

where the coefficients are either constants or loop-invariant.

Induction Variables

```
      s  := 0
      i  := 0
L1:   if i >= n goto L2
      j  := i*4
      k  := j+a
      x  := *k
      s  := s+x
      i  := i+1
L2:   ...
```

So let's represent them as triples of the form
(t, e_0, e_1):

```
j = (i, 0, 4)
k = (i, a, 4)
i = (i, 1, 1)
```

Induction Variables

```
      s := 0
      i := 0
L1:   if i >= n goto L2
      j := i*4
      k := j+a
      x := *k
      s := s+x
      i := i+1
L2:   ...
```

Note that **i** only changes by the *same* amount each iteration of the loop.

We say that **i** is a *linear induction variable*.

So it's easy to express the change in **j** & **k**.

Induction Variables

```
      s := 0
      i := 0
L1:   if i >= n goto L2
      j := i*4
      k := j+a
      x := *k
      s := s+x
      i := i+1
L2:   ...
```

If **i** changes by c, then since:

j = 4*i + 0
k = 4*i + a

we know that **j** & **k** change by 4*c.

Finding Induction Variables

Scan loop body to find all basic induction variables

do

Scan loop to find all variables k with one assignment of form k = j*b, where j is an induction variable <i,c,d>, and make k an induction variable with triple <i,c*b,d>

Scan loop to find all variables k with one assignment of form k = j+/-b where j is an induction variable with triple <i,c,d>, and make k and induction variable with triple <i,c,d+/-b)

until no more induction variables found

Strength Reduction

For each derived induction variable j of the form (i, e_0, e_1) make a fresh temp j'.

At the loop pre-header, initialize j' to e_0.

After each i:=i+c, define j':=j'+(e_1*c).

- note that e_1*c can be computed in the loop header (i.e., it's loop invariant.)

Replace the unique assignment of j in the loop with j := j'.

Example

```
      s := 0
      i := 0
      j' := 0
      k' := a
L1:   if i >= n goto L2
      j := i*4
      k := j+a
      x := *k
      s := s+x
      i := i+1
L2:   ...
```

Example

```
      s  := 0
      i  := 0
      j' := 0
      k' := a
L1:   if i >= n goto L2
      j  := i*4
      k  := j+a
      x  := *k
      s  := s+x
      i  := i+1
      j' := j'+4
      k' := k'+4
L2:   ...
```

Example

```
      s  := 0
      i  := 0
      j'  := 0
      k'  := a
L1:   if i >= n goto L2
      j  := j'
      k  := k'
      x  := *k
      s  := s+x
      i  := i+1
      j'  := j'+4
      k'  := k'+4
L2:   ...
```

Copy-propagation or coalescing will eliminate the distinction between **j**/**j**' and **k**/**k**'.

Useless Variables

```
     s  := 0
     i  := 0
     j'  := 0
     k'  := a
L1:  if i >= n goto L2
     x  := *k'
     s  := s+x
     i  := i+1
     j'  := j'+4
     k'  := k'+4
L2:  ...
```

A variable is *useless* for L if it is not live out at all exits from L and its only use is in a definition of itself.

For example, `j'` is useless.

We can delete useless variables from loops.

Useless Variables

```
      s := 0
      i := 0
      j' := 0
      k' := a
L1:   if i >= n goto L2
      x := *k'
      s := s+x
      i := i+1
      k' := k'+4
L2:   ...
```

DCE will pick up the dead initialization in the pre-header…

Almost Useless Variables

```
      s := 0
      i := 0
      k' := a
L1:   if i >= n goto L2
      x := *k'
      s := s+x
      i := i+1
      k' := k'+4
L2:   ...
```

The variable **`i`** is almost useless. It would be if it weren't used in the comparison…

See Appel for how to determine when/how it's safe to rewrite this test in terms of other induction variables in the family of **`i`**.

High-Level Loop Optimizations

- Require restructuring loops or sets of loops
  - Combining two loops (loop fusion)
  - Switching the order of a nested loop (loop interchange)
  - Completely changing the traversal order (loop tiling)

- These sorts of high level optimizations usually take place at the AST level (where loop structure is obvious)

Cache Behavior

Most loop transformations target cache behavior

- Attempt to increase *spatial* or *temporal* locality
- Locality can be exploited when there is *reuse* of data (for temporal locality) or recent access of nearby data (for spatial locality)

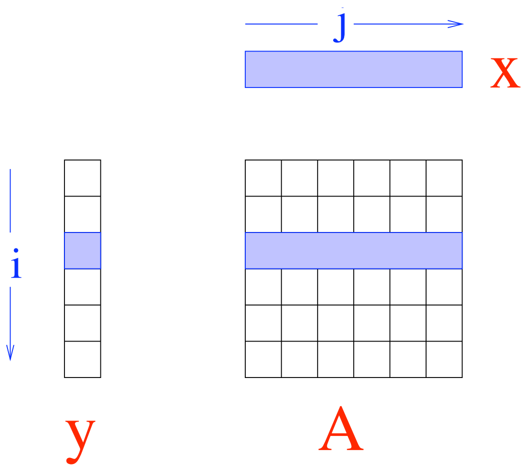
Loops are a good opportunity for this: many loops iterate through matrices or arrays

- Consider matrix-vector multiply example

Cache Behavior

Loops are a good opportunity for this: many loops iterate through matrices or arrays

- Consider matrix-vector multiply example
  - Multiple traversals or vector: opportunity for spatial and temporal locality
  - Regular access to array: opportunity for spatial locality



$$y = Ax$$

```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    y[i] += A[i][j] * x[j]
```

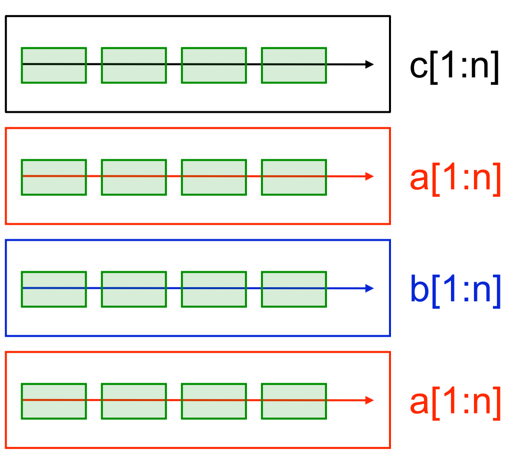
Loop Fusion

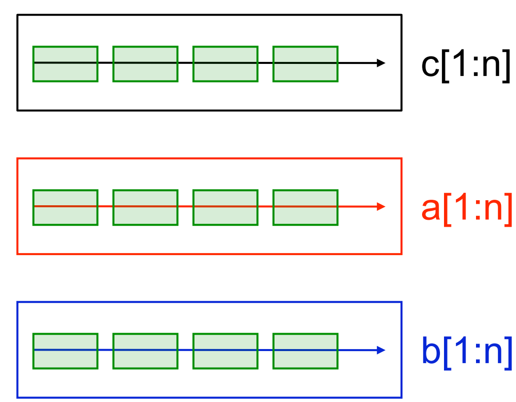
Combine two loops together into a single loop

- Why is this useful? Is it always legal?

```
for (i=1;i<=n;i++)
   c[i] = a[i];
for (i=1;i<=n;i++)
   b[i] = a[i];
```

```
for (i=1;i<=n;i++)
  { c[i] = a[i];
    b[i] = a[i]; }
```



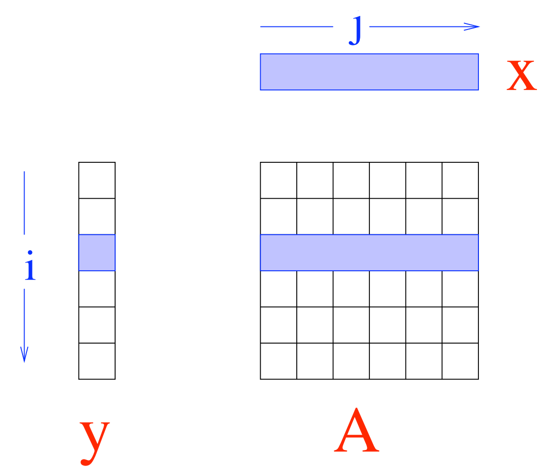


Loop Interchange

Change the order of a nested loop

- This is not always legal: it changes the order in which elements are accessed

Consider matrix-matrix multiply when A is stored in column-major order (i.e., each column is stored in contiguous memory)



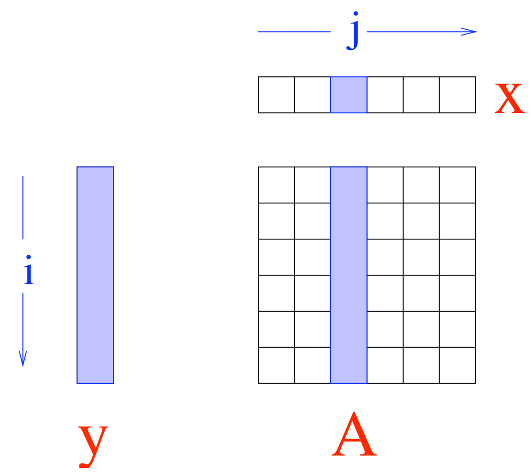

```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    y[i] += A[i][j] * x[j]
```

Loop Interchange

Change the order of a nested loop

- This is not always legal: it changes the order in which elements are accessed

Consider matrix-matrix multiply when A is stored in column-major order (i.e., each column is stored in contiguous memory)




```
for (j = 0; j < N; j++)
  for (i = 0; i < N; i++)
    y[i] += A[i][j] * x[j]
```

Loop Tiling

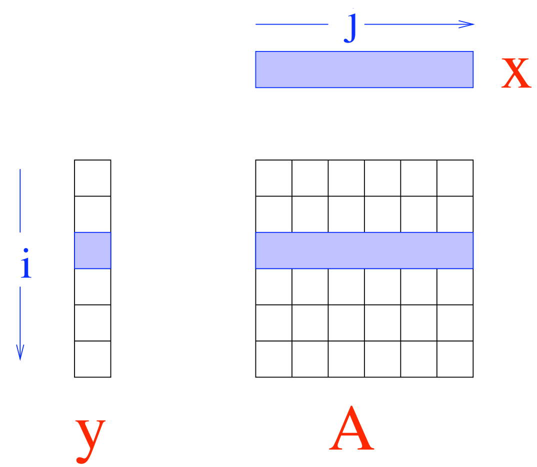
```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    y[i] += A[i][j] * x[j]
```

Also called “loop blocking”

Goal: break up loop into smaller pieces to get spatial & temporal locality

- One of the more complex loop transformations
- Create new inner loops so data accessed in inner loops fit in cache
- Also changes iteration order so may not be legal

```
for (ii = 0; ii < N; ii += B)
  for (jj = 0; jj < N; jj += B)
    for (i = ii; i < ii+B; i++)
      for (j = jj; j < jj+B; j++)
        y[i] += A[i][j] * x[j]
```



Loop Tiling

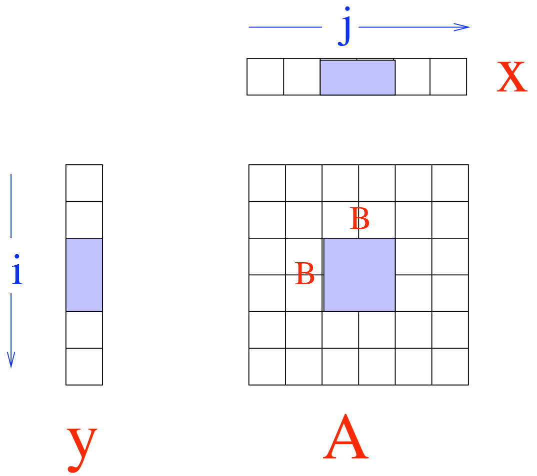
```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    y[i] += A[i][j] * x[j]
```

Also called "loop blocking"

Goal: break up loop into smaller pieces to get spatial & temporal locality

- One of the more complex loop transformations
- Create new inner loops so data accessed in inner loops fit in cache
- Also changes iteration order so may not be legal

```
for (ii = 0; ii < N; ii += B)
  for (jj = 0; jj < N; jj += B)
    for (i = ii; i < ii+B; i++)
      for (j = jj; j < jj+B; j++)
        y[i] += A[i][j] * x[j]
```



Loop Optimizations

- Loop transformations can have dramatic effects on performance
- Transforming loops correctly and automatically is very difficult!
- Researchers have developed many techniques to determine legality of loop transformations and automatically transform loops.