LOOP OPTIMIZATIONS

PROGRAM ANALYSIS AND OPTIMIZATION – DCC888

Fernando Magno Quintão Pereira

fernando@dcc.ufmg.br

The material in these slides have been taken from Chapter 18 of "Modern Compiler Implementation in Java – Second Edition", by Andrew Appel and Jens Palsberg.

PPGCC
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO DCC-UFMG

The Importance of Loops

- A program spends most of its processing time in loops
  - There is even the famous rule 90/10, that says that 90% of the processing time is spent in 10% of the code.
- Thus, optimizing loops is essential for achieving high performance.
- Some optimizations transform the iteration space of the loop
  - We will not deal with them now
- We will be talking only about transformations that preserve the iteration space.
  - Examples: code hoisting, strength reduction, loop unrolling, etc.

What is the iteration space of a loop?

IDENTIFYING LOOPS

DCC 888

Identifying Loops

```
int main(int argc, char** argv) {
  int sum = 0;
  int i = 1;
  while (i < argc) {
    char* c = argv[i];
    while (*c != '\0') {
      c++;
      sum++;
    }
  }
  printf("sum = %d\n", sum);
}
```

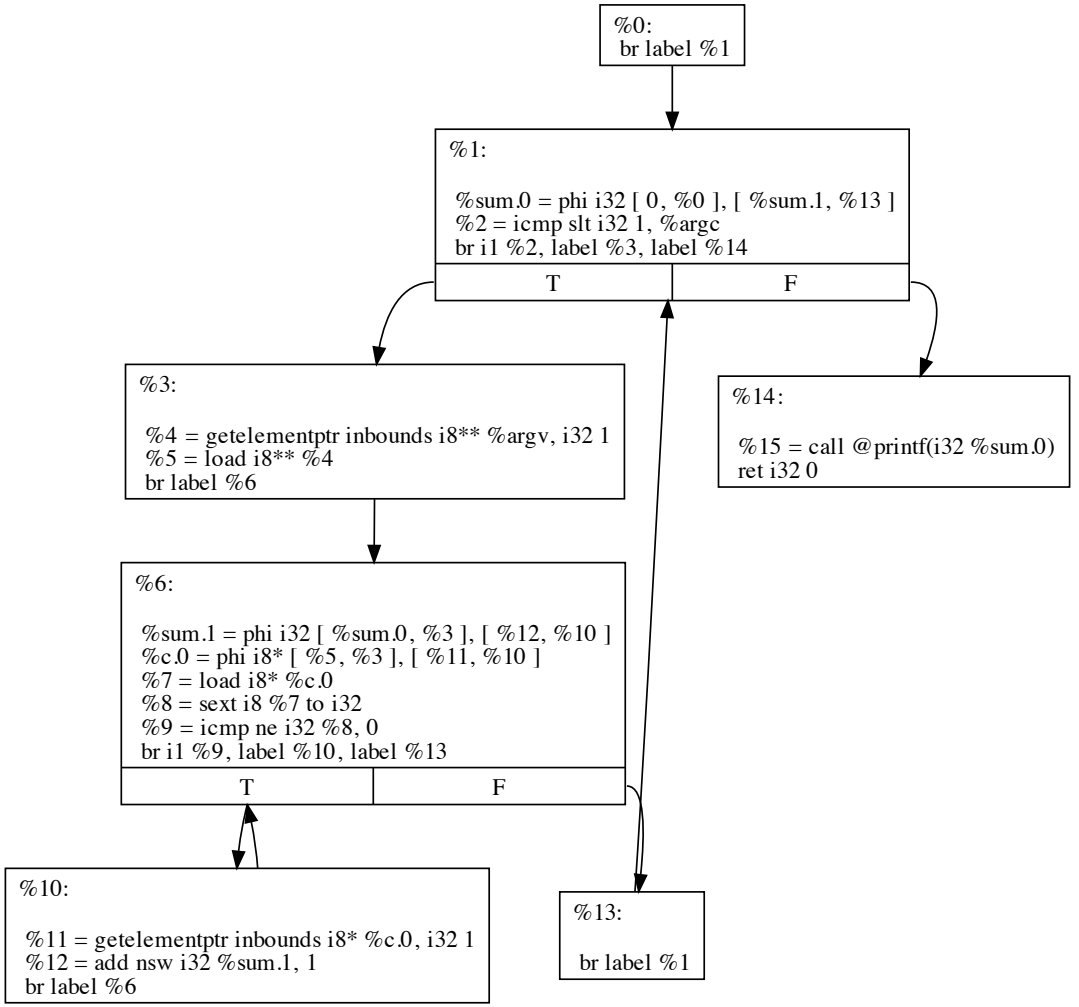
1) Consider the program below. How many loops does it have?

2) How could we identify these loops in the program's CFG?

Identifying Loops

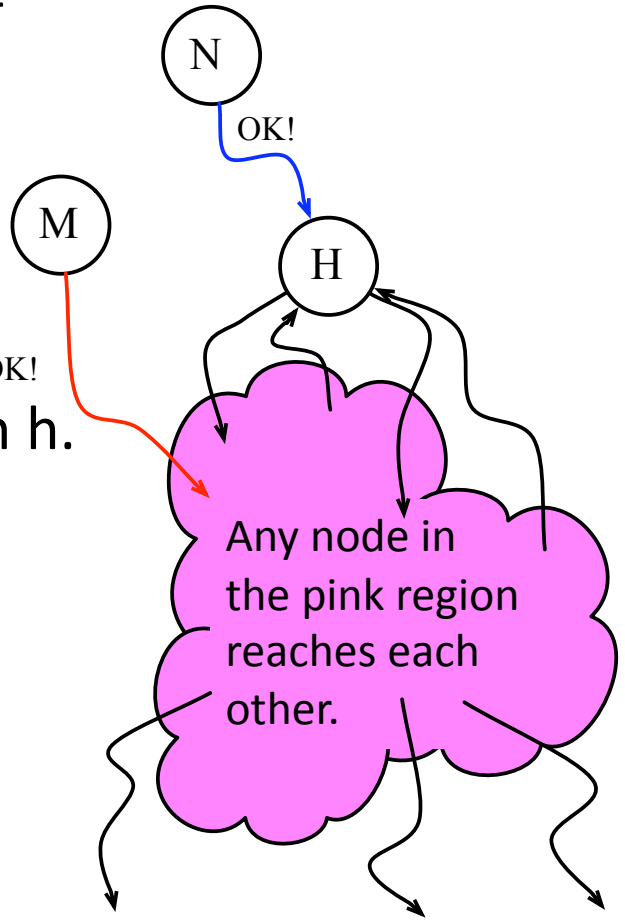
```
int main(int argc, char** argv) {
  int sum = 0;
  int i = 1;
  while (i < argc) {
    char* c = argv[i];
    while (*c != '\0') {
      c++;
      sum++;
    }
  }
  printf("sum = %d\n", su
}
```

And how can we identify loops in general?



Identifying Loops

- A loop in a control flow graph is a set of nodes S including a header node h with the following properties:
  1. From any node in S there is a path of directed edges leading to h.
  2. There is a path of directed edges from h to any node in S.
  3. There is no edge from any node outside S to any node in S other than h.

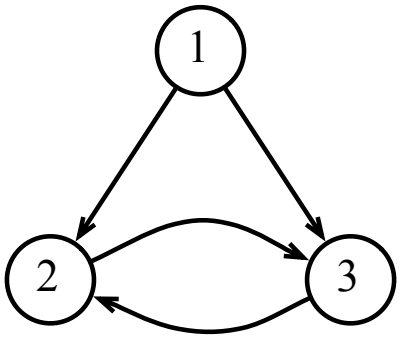


1) Why is (3) important to define loops?

2) How could we produce programs that break (3)?

Identifying Loops

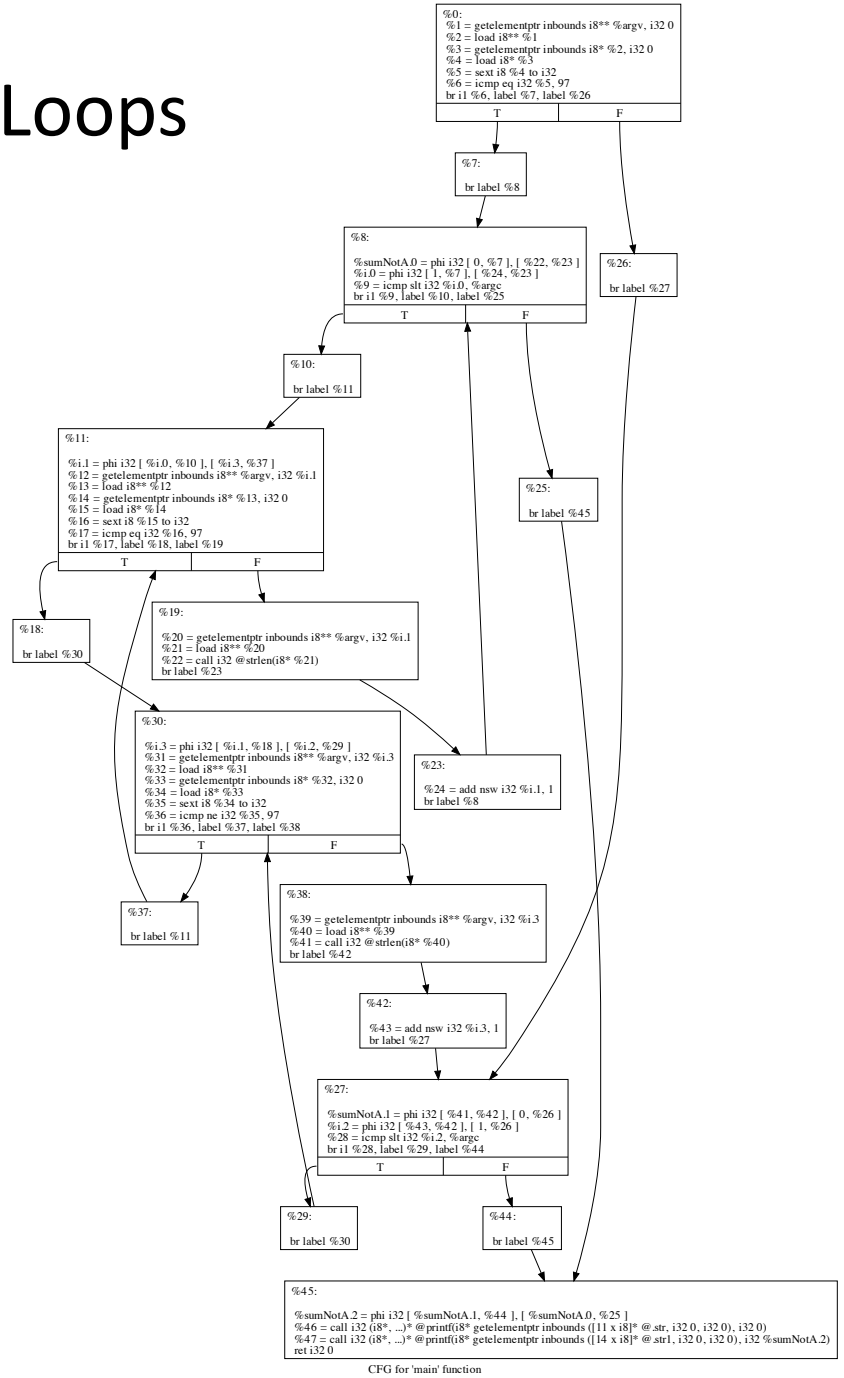
- As we have seen, a loop contains only one entry point.
  - So, what is **not** a loop? We are not interested in CFG cycles that contain two or more nodes that have predecessors outside the cycle.
  - These cycles have no interest to us, because most of the optimizations that we will describe in this class cannot be applied easily on them.
- The canonical example of a cycle that is not a loop is given on the right.
- If a cycle contains this pattern as a subgraph, then this cycle is not a loop.
- Any CFG that is free of this pattern is called a *reducible* control flow graph.



Identifying Loops

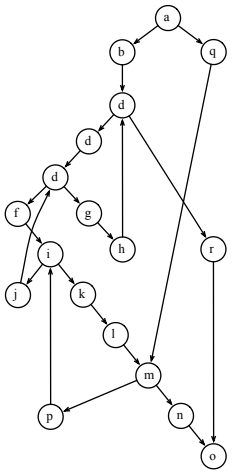
- A loop in a control flow graph is a set of nodes S including a header node h with the following properties:
  1. From any node in S there is a path of directed edges leading to h.
  2. There is a path of directed edges from h to any node in S.
  3. There is no edge from any node outside S to any node in S other than h.

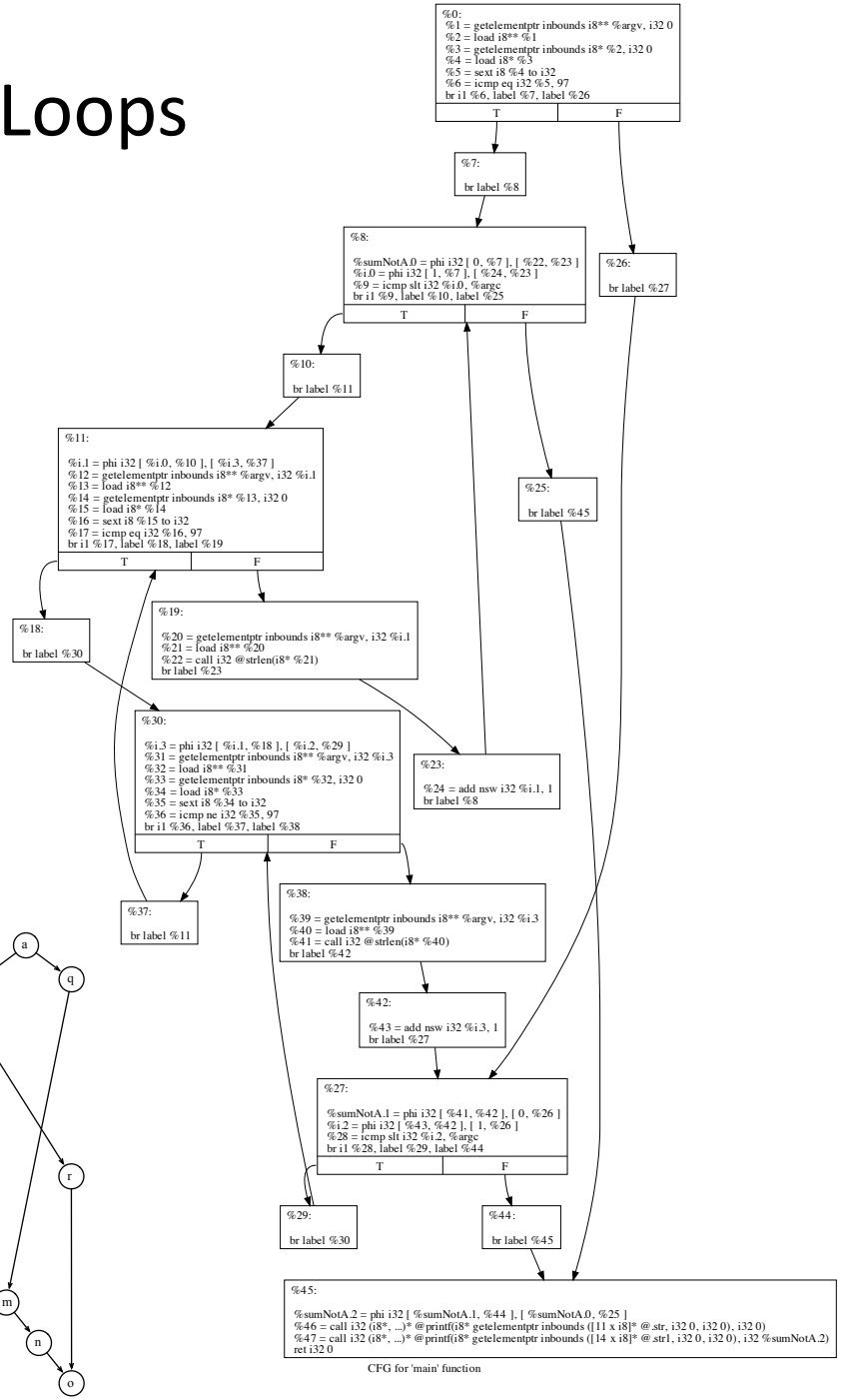
Is the CFG on the right reducible?



Identifying Loops

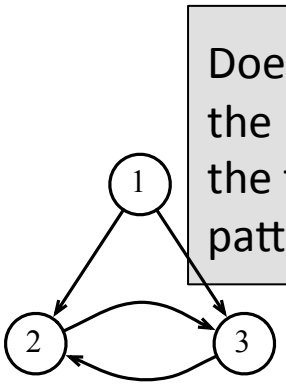
- We can collapse edges of our CFG, until we find the forbidden pattern.
- We collapse an edge (n_1, n_2) in the following way:
  - We delete the edge (n_1, n_2)
  - We create a new node n_{12}
  - We let all the predecessors of n_1 and all the predecessors of n_2 to be predecessors of n_{12}
  - We let all the successors of n_1 and all the successors of n_2 to be successors of n_{12}
  - We delete the nodes n_1 and n_2.



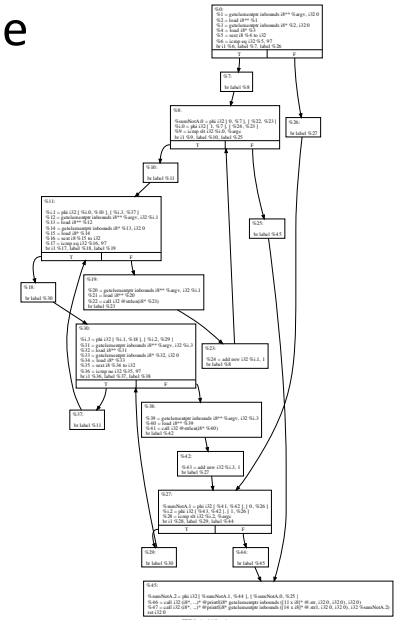


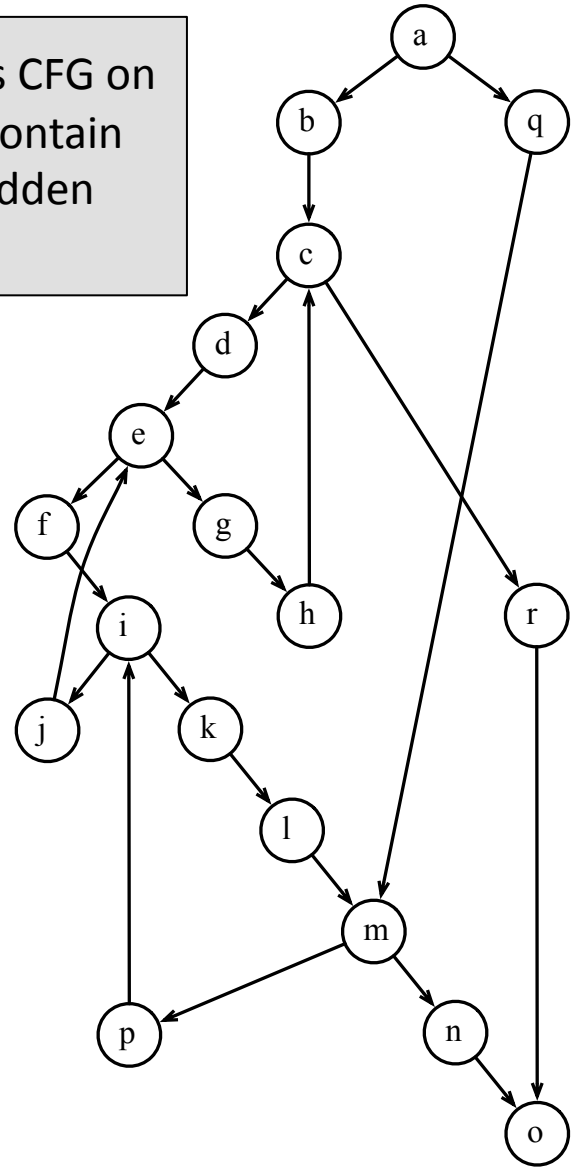
Identifying Loops

- We can collapse edges of our CFG, until we find the forbidden pattern.
- We collapse an edge (n_1, n_2) in the following way:
  - We delete the edge (n_1, n_2)
  - We create a new node n_{12}
  - We let all the predecessors of n_1 and all the predecessors of n_2 to be predecessors of n_{12}
  - We let all the successors of n_1 and all the successors of n_2 to be successors of n_{12}
  - We delete the nodes n_1 and n_2.



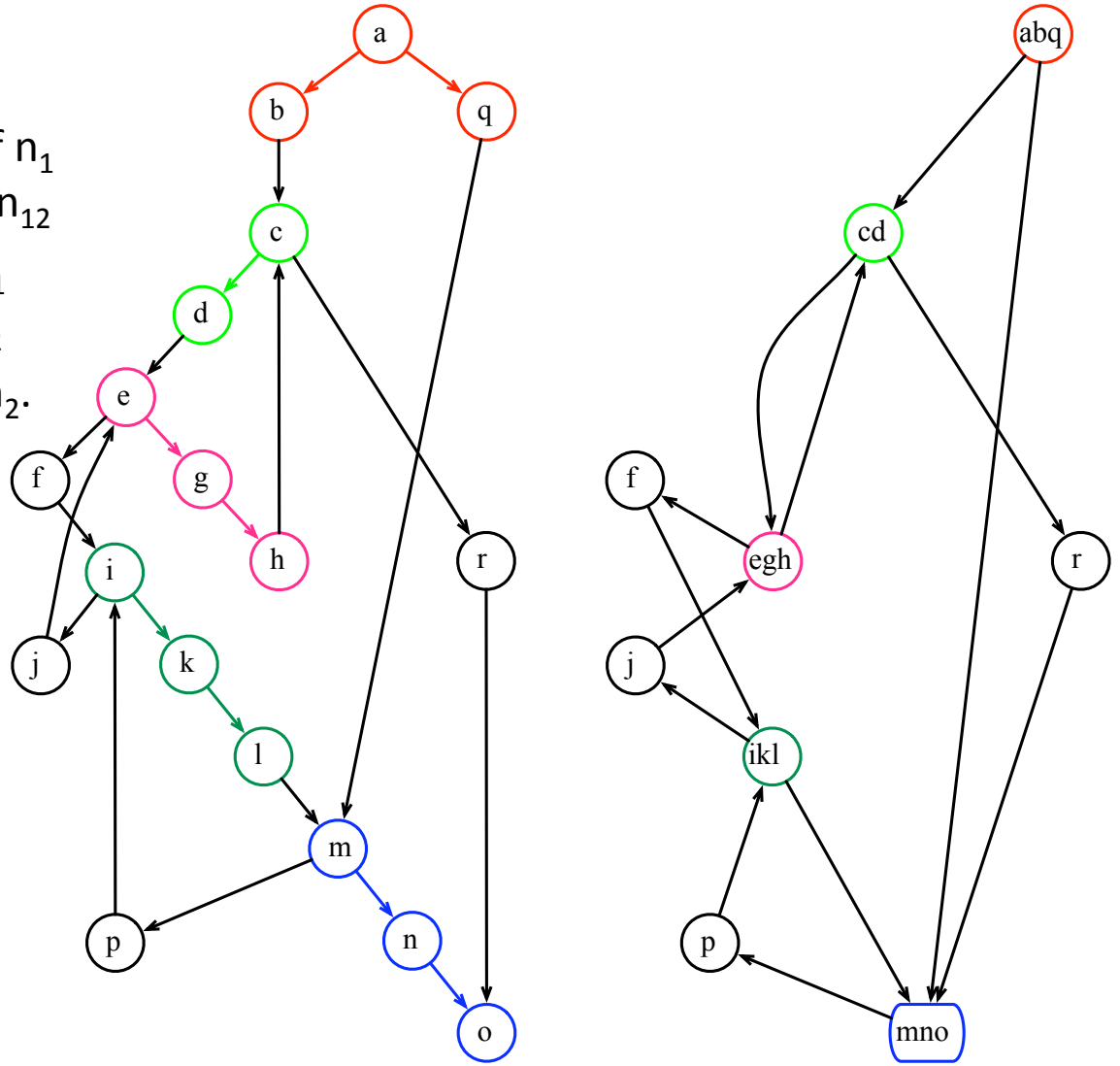
Does this CFG on the left contain the forbidden pattern?



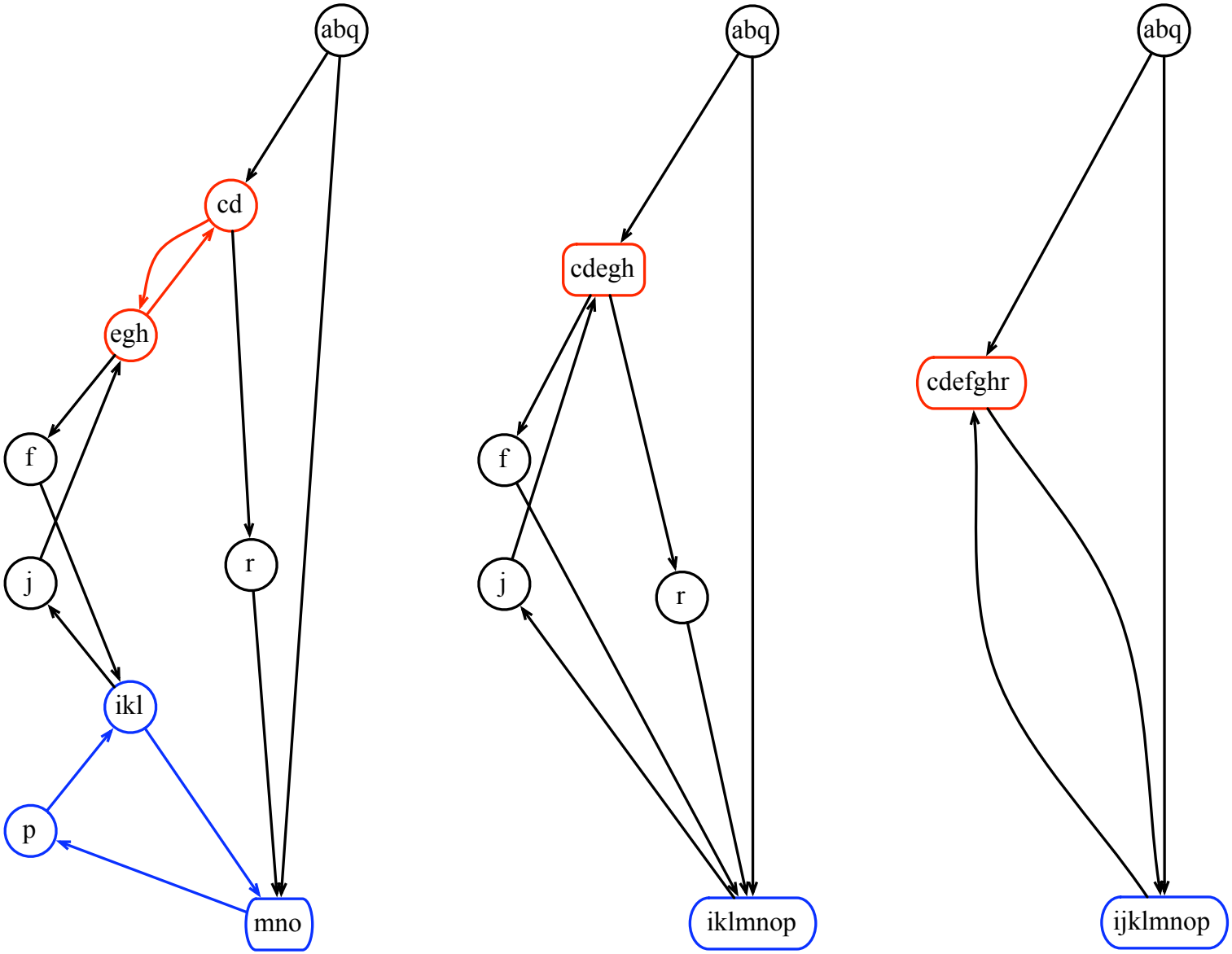


Identifying Loops

- We collapse an edge (n_1, n_2) in the following way:
  - We delete the edge (n_1, n_2)
  - We create a new node n_{12}
  - We let all the predecessors of n_1 and n_2 to be predecessors of n_{12}
  - We let all the successors of n_1 and n_2 to be successors of n_{12}
  - We delete the nodes n_1 and n_2.



Identifying Loops



Why Reducible Flow Graphs are Good

- Because the entry point of the loop – the header – is unique, we can use this block as the region where to place redundant code.
- Dataflow analyses tend to terminate faster in reducible flow graphs.
- Usually, syntactic loops, such as **for**, **while**, **repeat**, **continue** and **break** produce reducible flow graphs.
- Unreducible flow graphs are formed by goto statements.

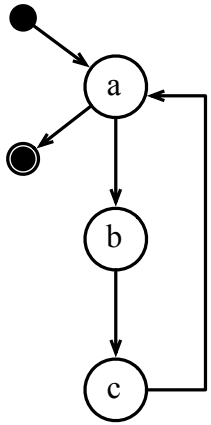
```
int main(int argc, char** argv) {
  int sumA = 0;
  int sumNotA = 0;
  int i = 1;
  while (i < argc) {
LNotA:
    if (argv[i][0] == 'a') {
      goto LA;
    } else {
      sumNotA = strlen(argv[i]);
    }
    i++;
  }
  goto End;
  while (i < argc) {
LA:
    if (argv[i][0] != 'a') {
      goto LNotA;
    } else {
      sumNotA = strlen(argv[i]);
    }
    i++;
  }
End:
  printf("sumA = %d\n", sumA);
  printf("sumNotA = %d\n", sumNotA);
}
```

Identifying Loops

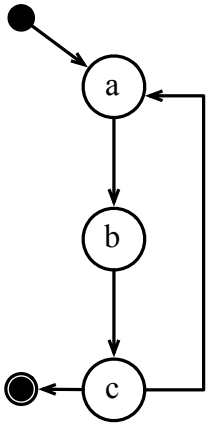
1) Is there any false loop here?

2) Which syntax could produce these loops?

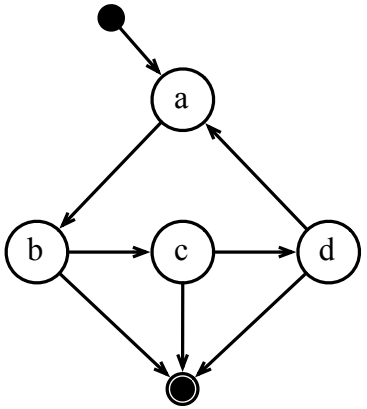
3) Which one is the header node of the actual loops?



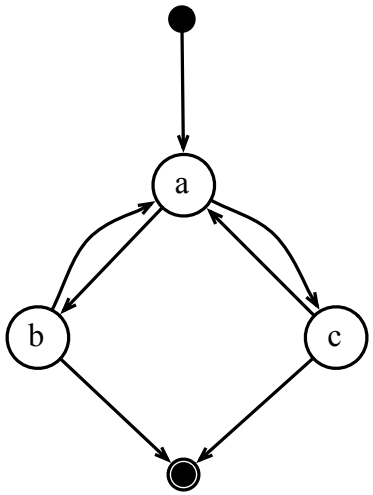
(a)



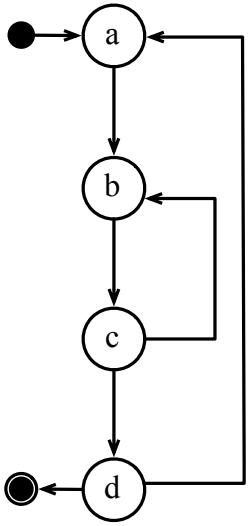
(b)



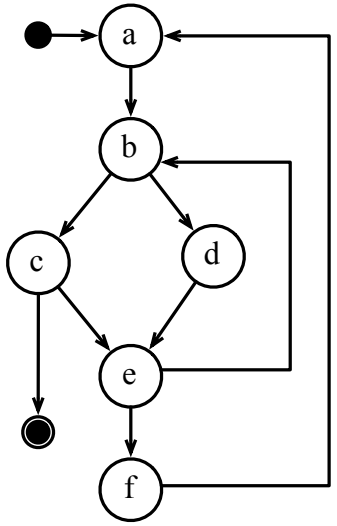
(c)



(d)



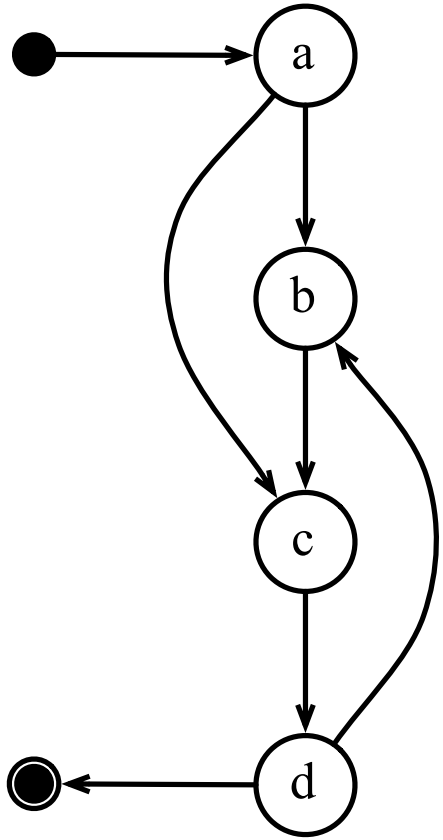
(e)



(f)

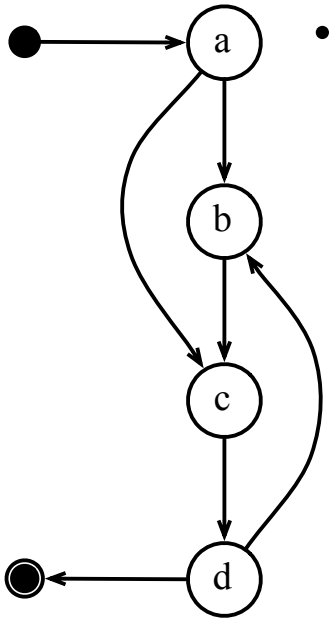
Identifying Loops

What about this CFG: is it reducible?

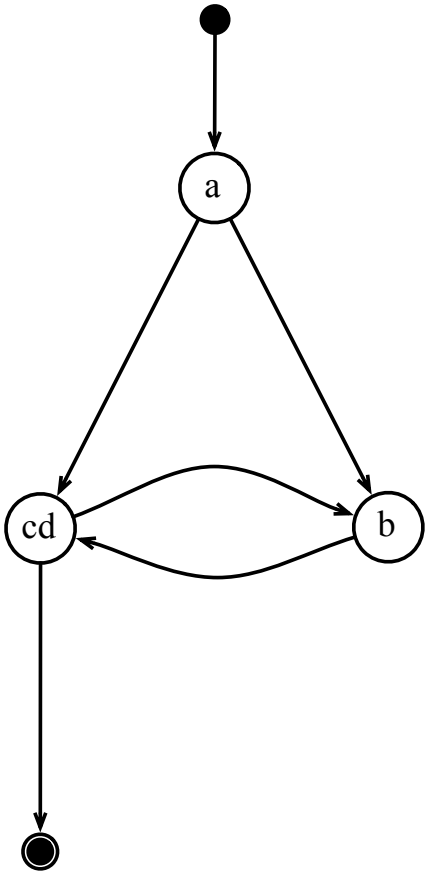


Identifying Loops

What about this CFG: is it reducible?



- We collapse an edge (n_1, n_2) in the following way:
  - We delete the edge (n_1, n_2)
  - We create a new node n_{12}
  - We let all the predecessors of n_1 and n_2 to be predecessors of n_{12}
  - We let all the successors of n_1 and n_2 to be successors of n_{12}
  - We delete the nodes n_1 and n_2.



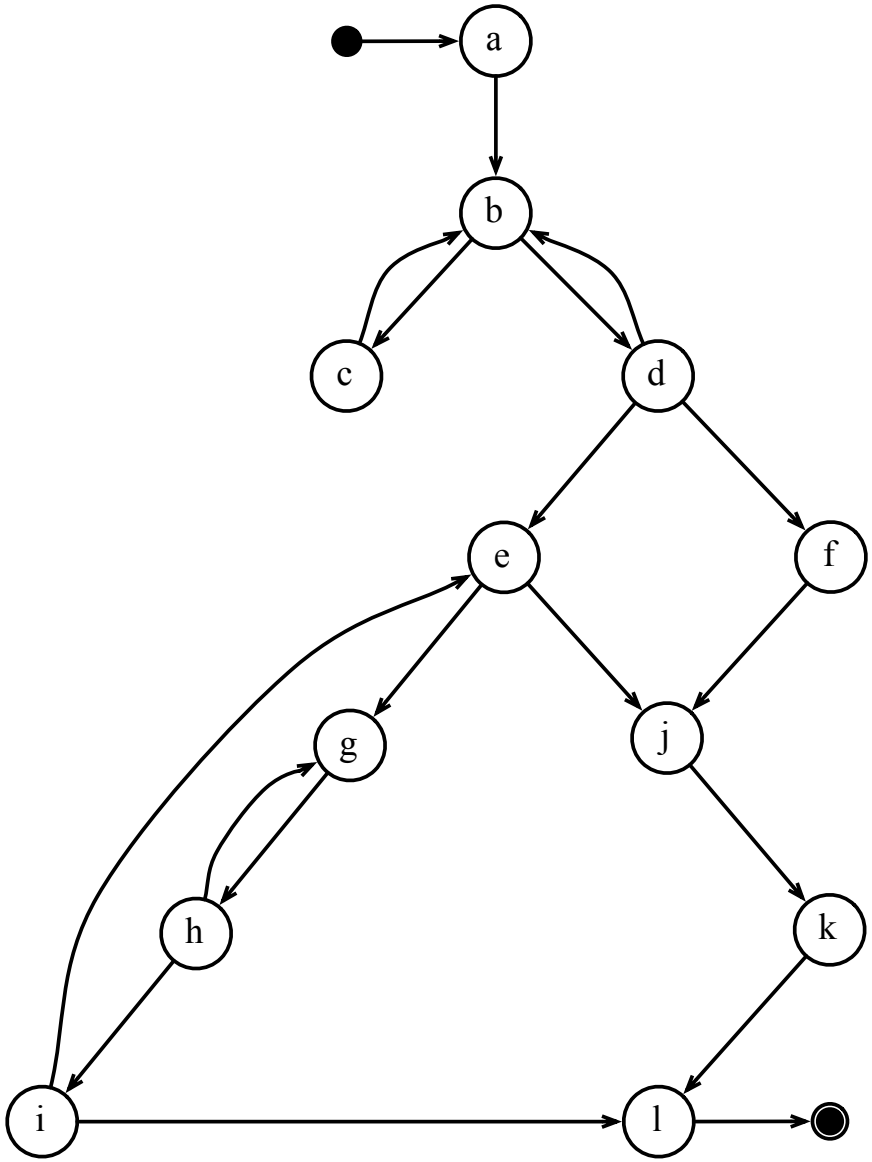
Dominators

- Dominators are a very important notion in compiler optimization.
- A node *d* dominates a node *n* if every path of directed edges from s_0 to *n* must go through *d*, where s_0 is the entry point of the control flow graph.
- We can find dominators via the equations below:

$$D[s_0] = \{s_0\} \qquad D[n] = \{n\} \cup (\bigcap_{p \in pred[n]} D[p]), \text{ for } n \neq s_0$$

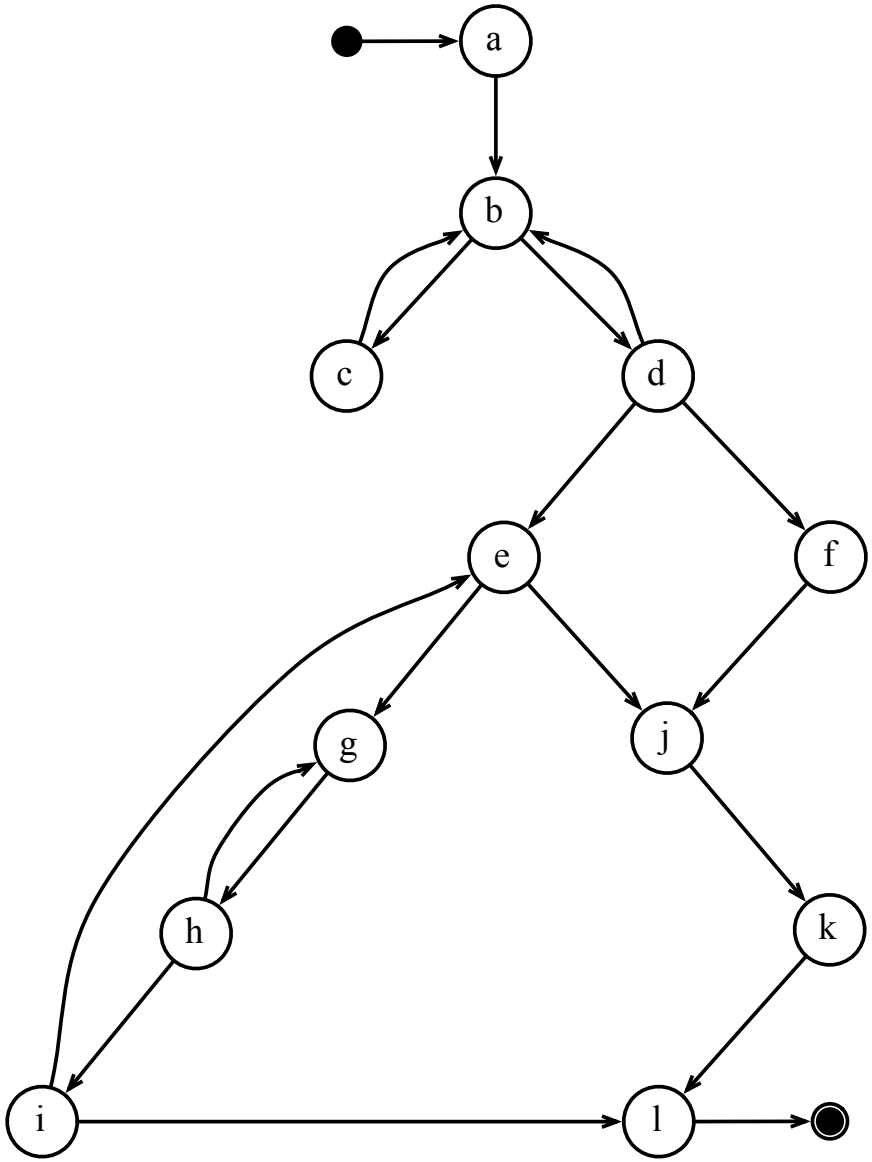
- We initialize every D[n] to all the nodes in the CFG.
- The assignments to D[n] make the set of dominators of n smaller each time due to the use of the intersection operator.

Dominators



1) Is this control flow graph reducible?
2) Can you build a table with the dominators of each node of this CFG?
3) Lets start with nodes "a" and "b". What are the dominators of these nodes?

Dominators



a: {a}

b: {a, b}

c:

d:

e:

f:

g:

h:

i:

j:

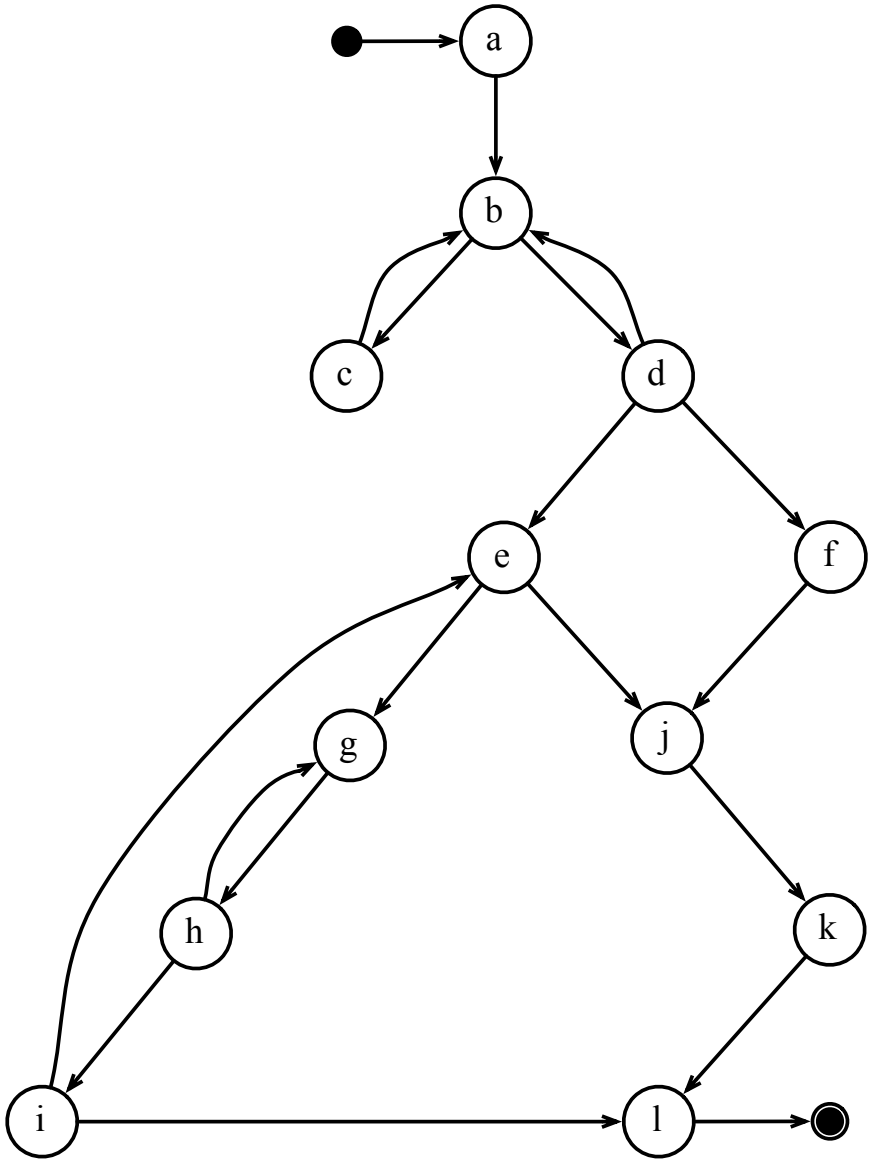
k:

l:

What about "c", "d", "e" and "f"? Remember the formula to compute **dominators**...

$$D[n] = \{n\} \cup (\cap_{p \in pred[n]} D[p])$$

Dominators



a: {a}

b: {a, b}

c: {a, b, c}

d: {a, b, d}

e: {a, b, d, e}

f: {a, b, d, f}

g:

h:

i:

j:

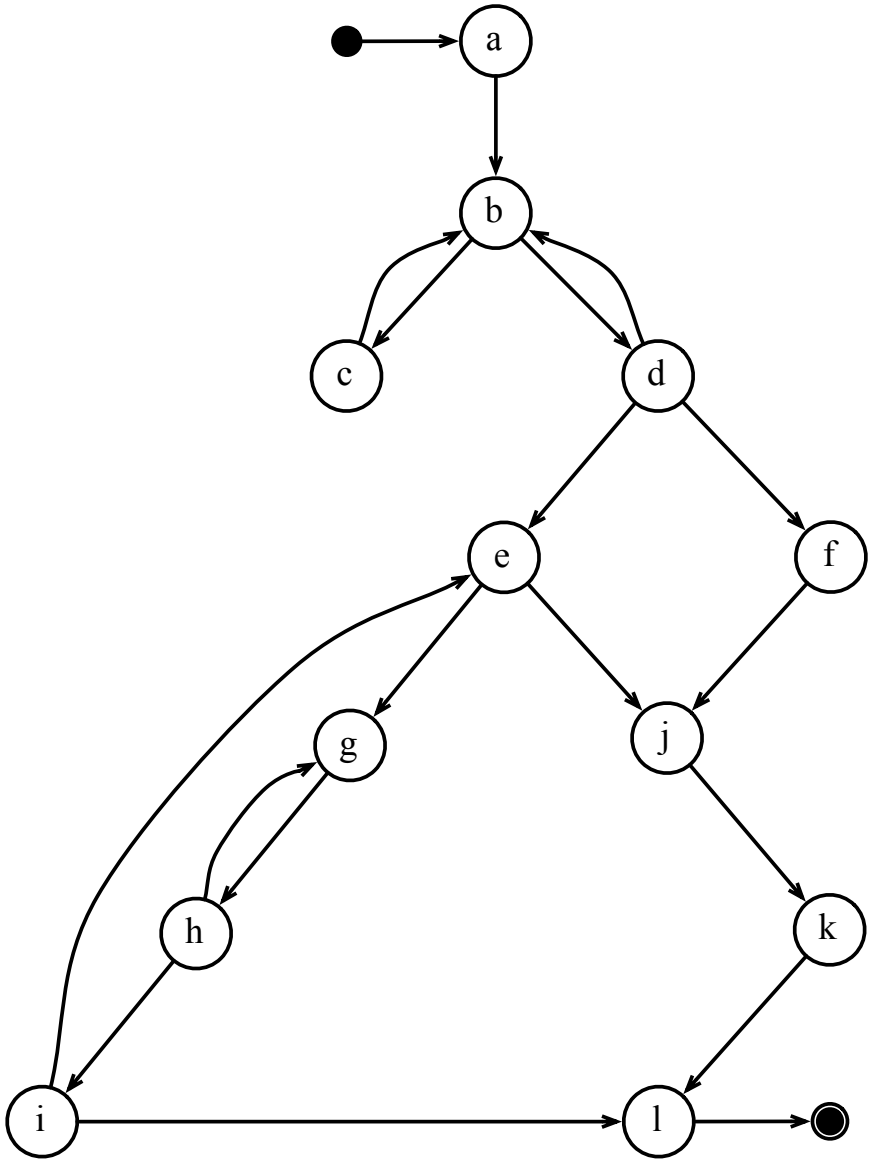
k:

l:

What are the dominators of "g" and "j"?

$$D[n] = \{n\} \cup (\cap_{p \in pred[n]} D[p])$$

Dominators



a: {a}

b: {a, b}

c: {a, b, c}

d: {a, b, d}

e: {a, b, d, e}

f: {a, b, d, f}

g: {a, b, d, e, g}

h:

i:

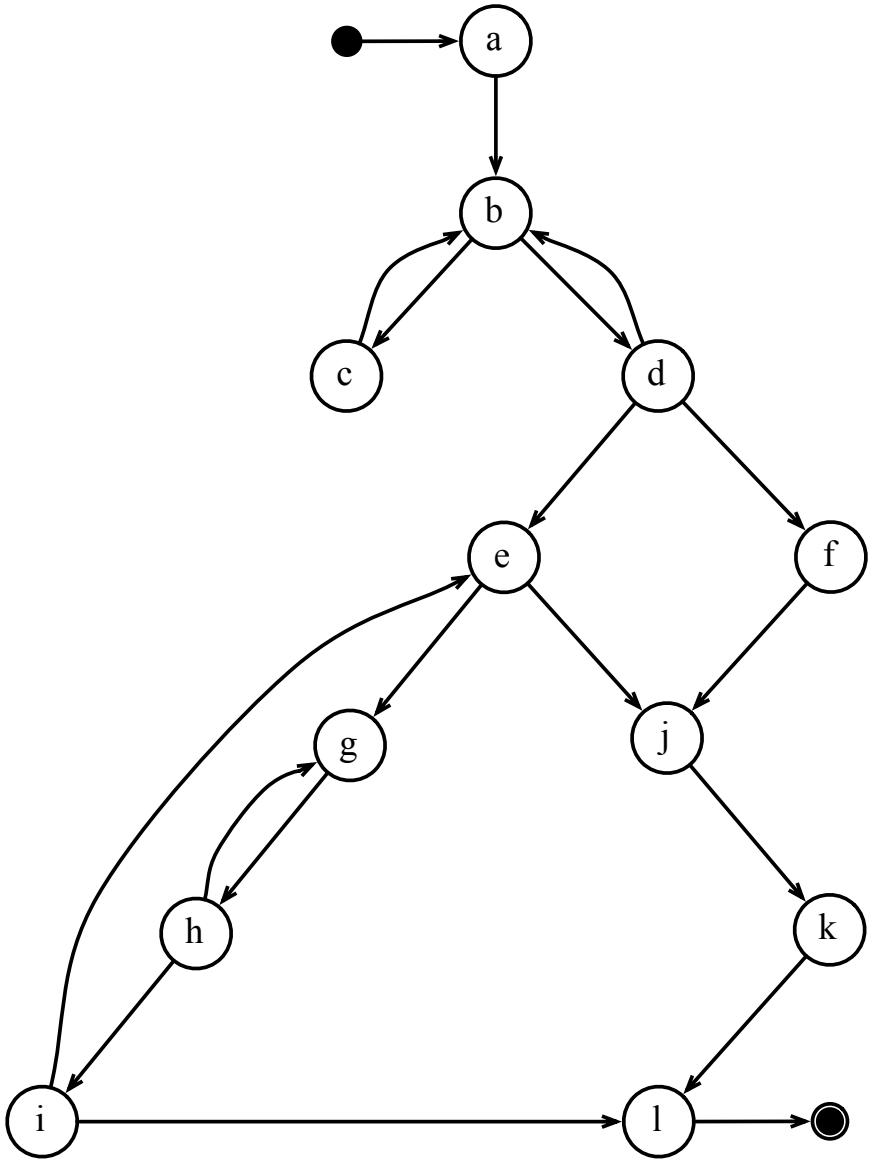
j: {a, b, d, j}

k:

l:

And what about the dominators of "h", "i" and "k"? Let's leave just "l" out now.

Dominators



a: {a}

b: {a, b}

c: {a, b, c}

d: {a, b, d}

e: {a, b, d, e}

f: {a, b, d, f}

g: {a, b, d, e, g}

h: {a, b, d, e, g, h}

i: {a, b, d, e, g, h, i}

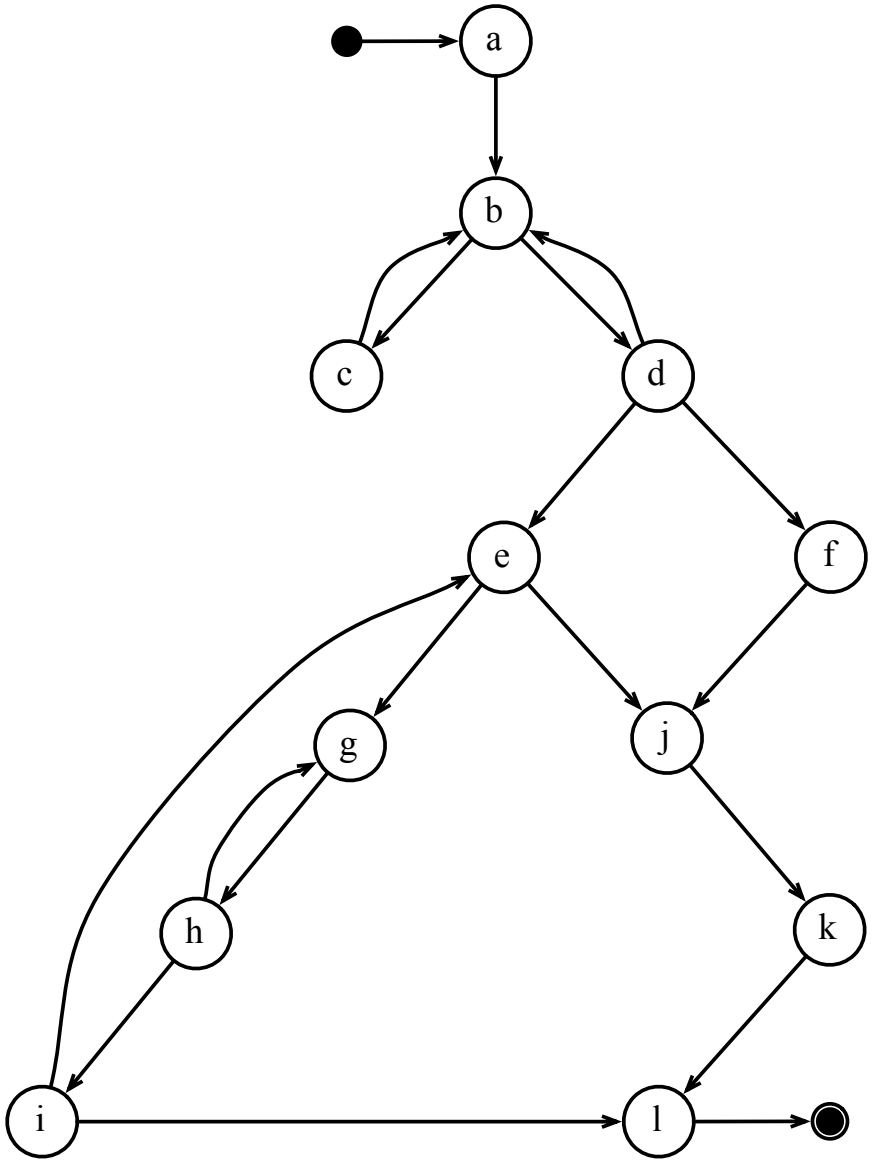
j: {a, b, d, j}

k: {a, b, d, j, k}

l:

Finally, what are the dominators of "l"?

Dominators



a: {a}

b: {a, b}

c: {a, b, c}

d: {a, b, d}

e: {a, b, d, e}

f: {a, b, d, f}

g: {a, b, d, e, g}

h: {a, b, d, e, g, h}

i: {a, b, d, e, g, h, i}

j: {a, b, d, j}

k: {a, b, d, j, k}

l: {a, b, d, l}

Immediate Dominators

- Every node n of a CFG, except its entry point, has one unique *immediate dominator*, which we shall denote by idom(n), such that:
  - idom(n) is not the same node as n
  - idom(n) dominates n,
  - idom(n) does not dominate any other dominator of n.

- We can prove that this statement is true via the following theorem:
  - In a connected graph, suppose *d* dominates *n*, and *e* dominates *n*. Then it must be the case that either *d* dominates *e*, or *e* dominates *d*.

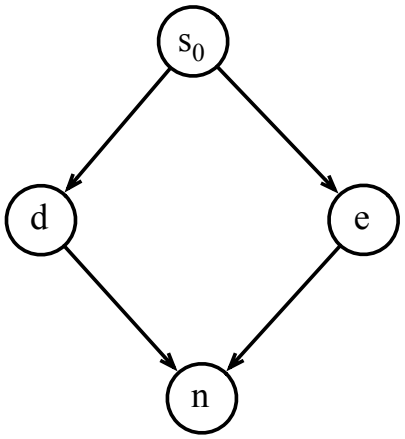
How can we prove this theorem?

Immediate Dominators

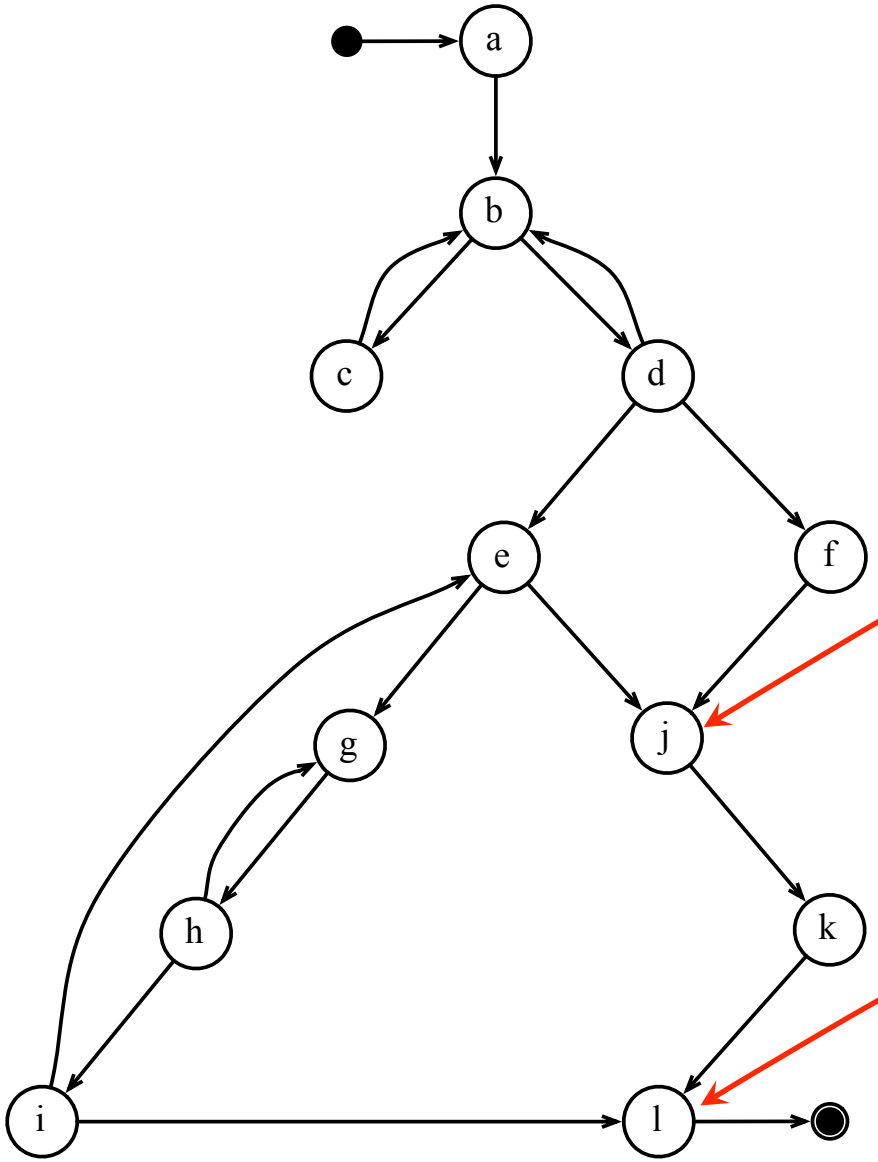
Theorem: In a connected graph, suppose *d* dominates *n*, and *e* dominates *n*. Then it must be the case that either *d* dominates *e*, or *e* dominates *d*.

The proof uses a simple contradiction:
If neither d nor e dominate each other, then there must be a path from s_0 to e that does not go through d. Therefore, any path from e to n must go through d. If that is not the case, then d would not dominate n.

We use an analogous argument to show that any path from d to n must go through e. Therefore, any path from d to n goes through e, and vice-versa. If d ≠ e, then this is an absurd, because a path that reaches n must leave either d or e last.



Immediate Dominators

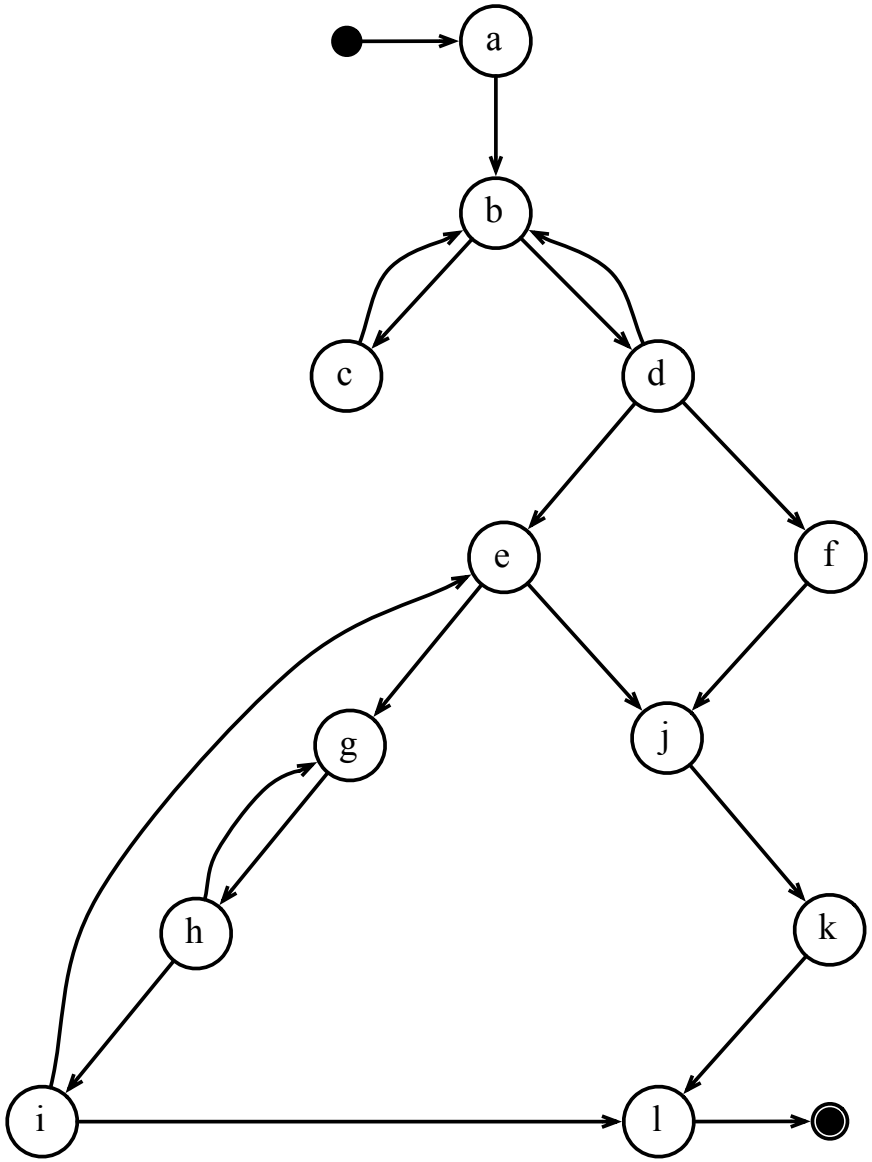


Which node is the immediate dominator of j?

Which node is the immediate dominator of l?

Can you point out the immediate dominator of each node in this CFG?

Immediate Dominators



idom(a): s_0

idom(b): a

idom(c): b

idom(d): b

idom(e): d

idom(f): d

idom(g): e

idom(h): g

idom(i): h

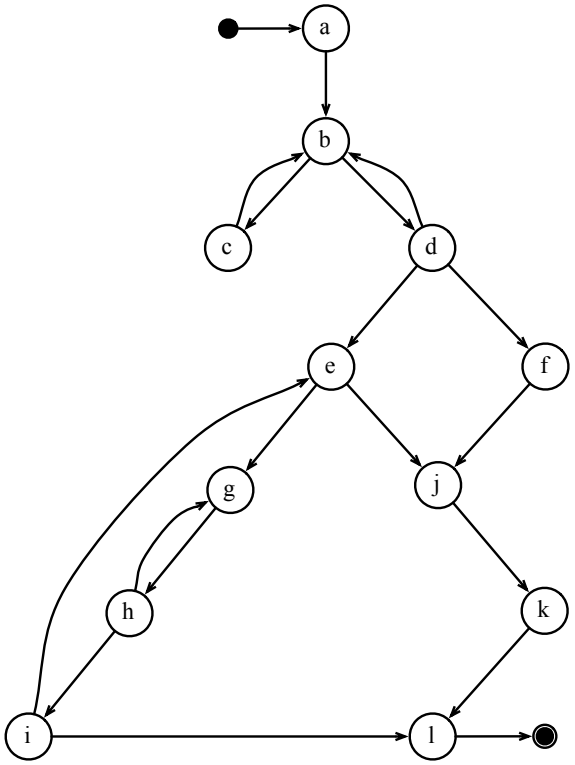
idom(j): d

idom(k): j

idom(l): d

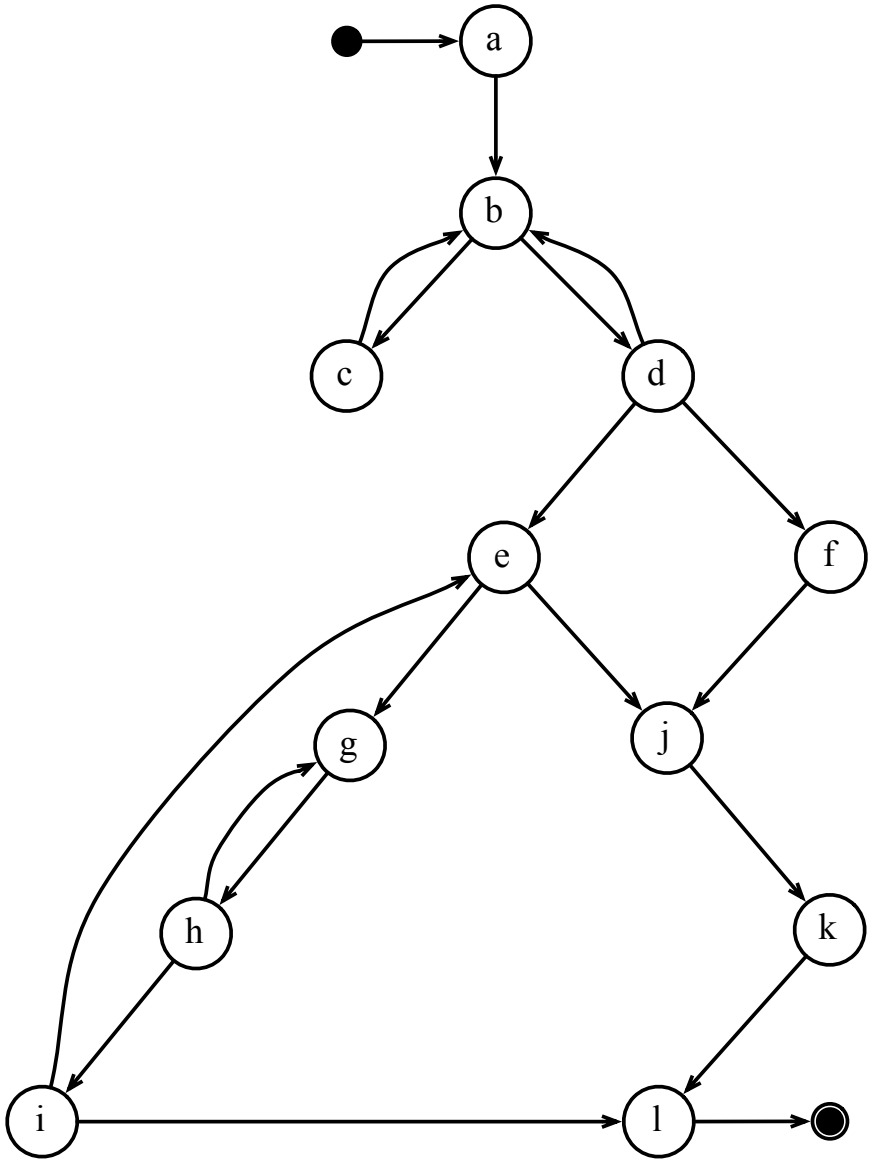
Dominator Trees

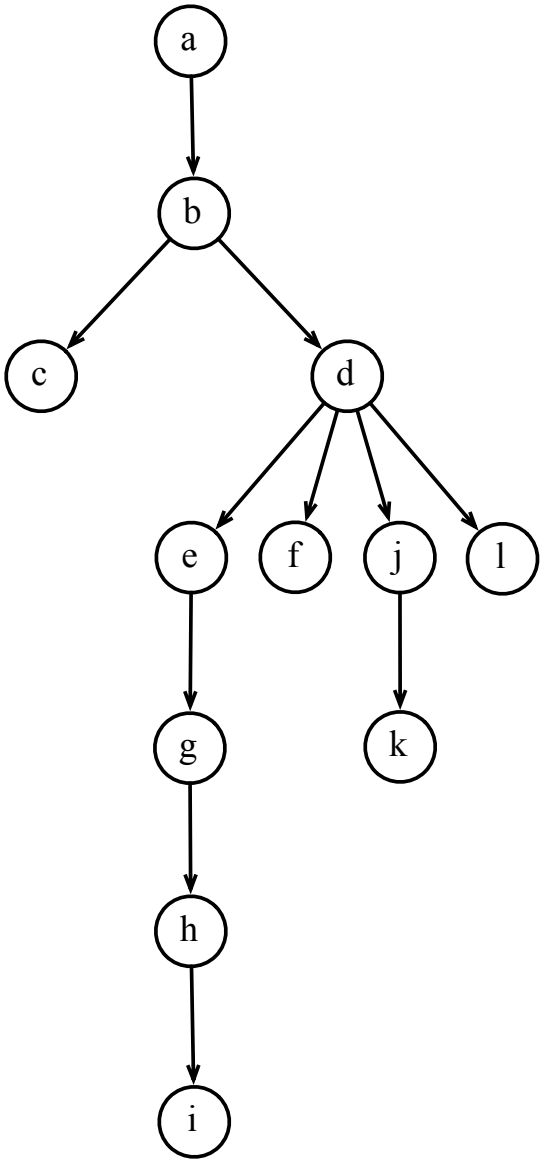
- The notion of immediate dominator defines a tree unambiguously: if d is the immediate dominator of n, then we add an edge (d, n) to this tree, which we call the *dominator tree* of the CFG.



What is the dominator tree of this CFG?

Dominator Trees





idom(a): s_0

idom(b): a

idom(c): b

idom(d): b

idom(e): d

idom(f): d

idom(g): e

idom(h): g

idom(i): h

idom(j): d

idom(k): j

idom(l): d

Nested Loops

- We generally want to optimize the most deeply nested loop first, before optimizing the enclosing loops.

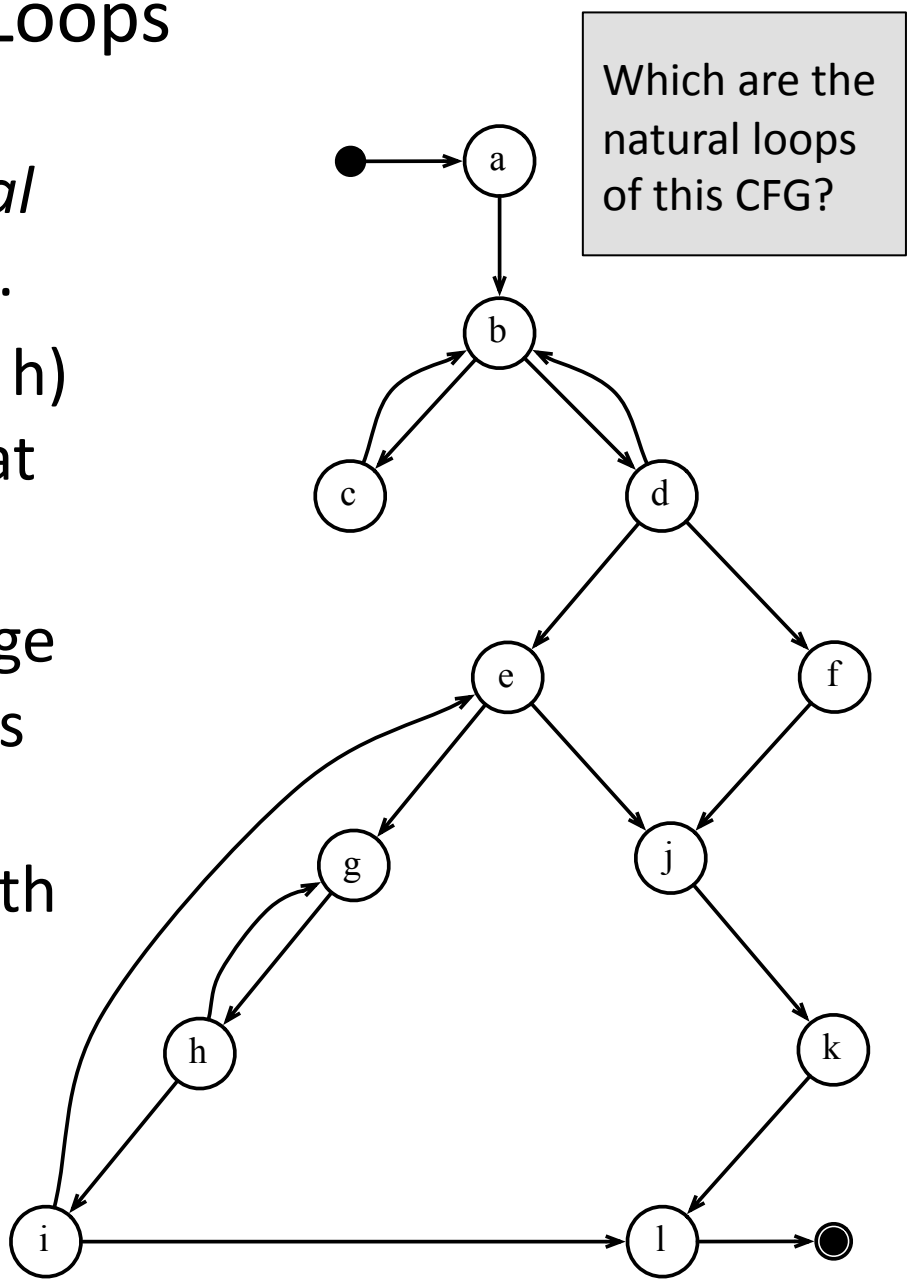
Given two loops from a reducible CFG, how can we tell if one of them is nested into the other?

```
int main(int argc, char** argv) {
  int sum = 0;
  int i = 1;
  while (i < argc) {
    char* c = argv[i];
    while (*c != '\0') {
      c++;
      sum++;
    }
  }
  printf("sum = %d\n", sum);
}
```

Nested loop

Natural Loops

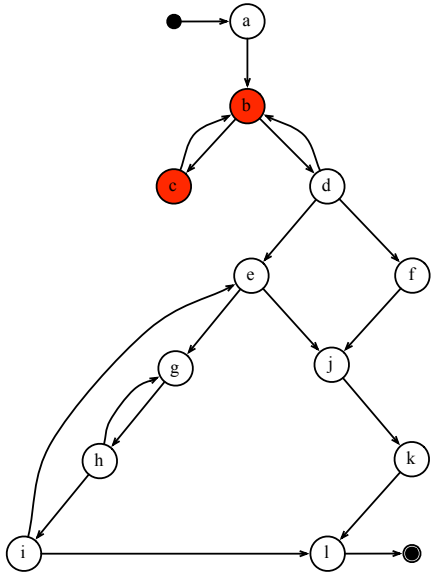
- We use the notion of a *natural loop*, to find the nested loops.
- A *back-edge* is a CFG edge (n, h) from a node n to a node h that dominates n.
- The natural loop of a back edge (n, h), where h dominates n, is the set of nodes x such that h dominates x and there is a path from x to n not containing h.
  - h is the header of this loop.

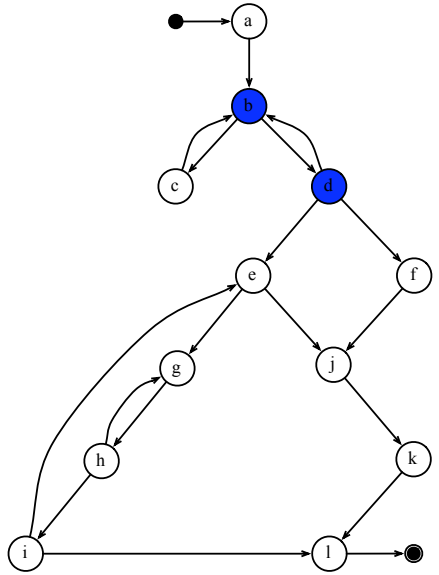


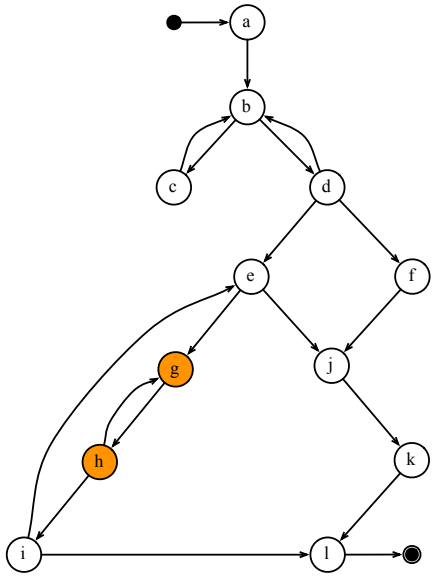
Natural Loops

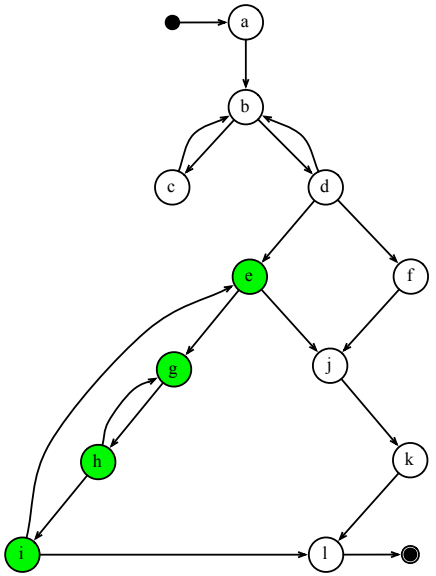
- The natural loop of a back edge (n, h), where h dominates n, is the set of nodes x such that h dominates x and there is a path from x to n not containing h.
  - h is the header of this loop.

So, how can we find the loop headers?





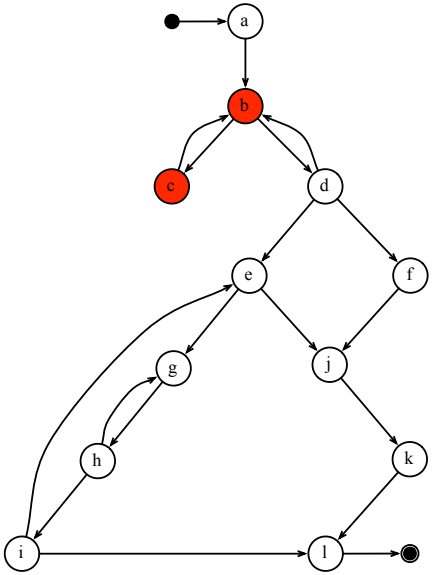


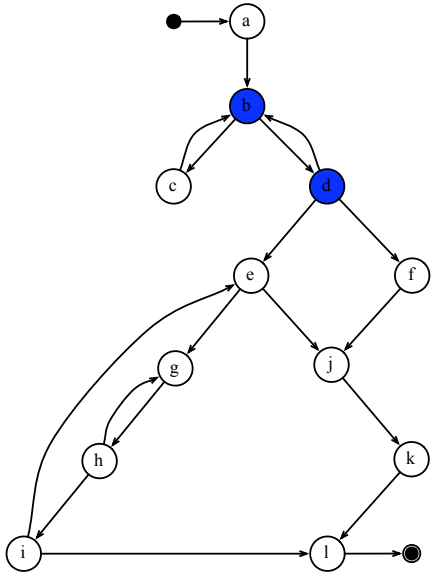


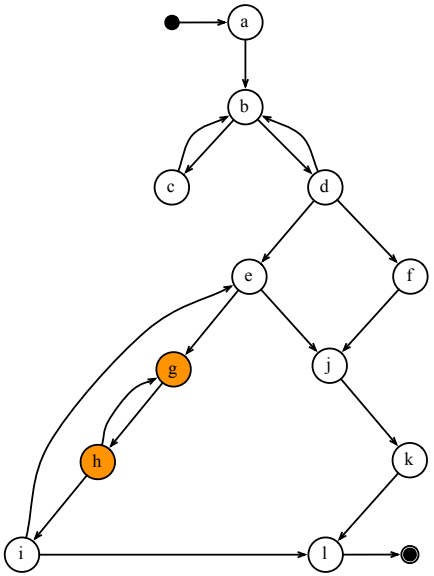
Natural Loops

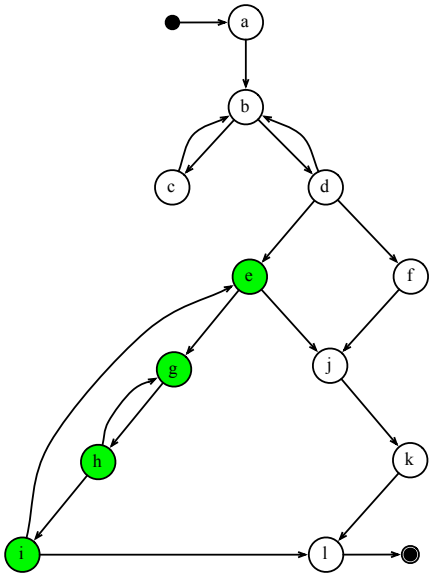
- The natural loop of a back edge (n, h), where h dominates n, is the set of nodes x such that h dominates x and there is a path from x to n not containing h.
  - h is the header of this loop.

A node h is a header if there exists a node n, such that h dominates it, and there is an edge (n, h) in the CFG.







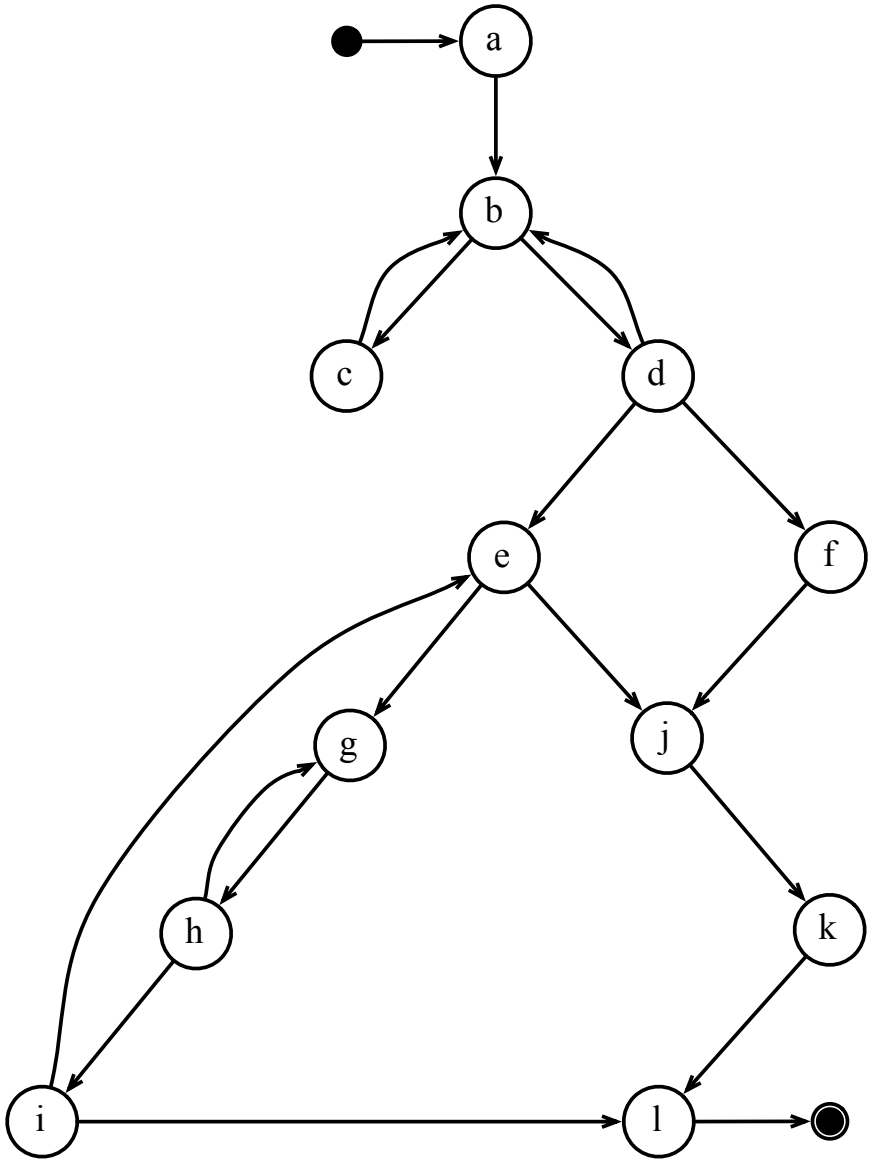


Finding the Loop Header

1) Given a strongly connected component of a reducible CFG, how can we identify the header node?

2) Which nodes are the loop headers in the control flow graph on the right?

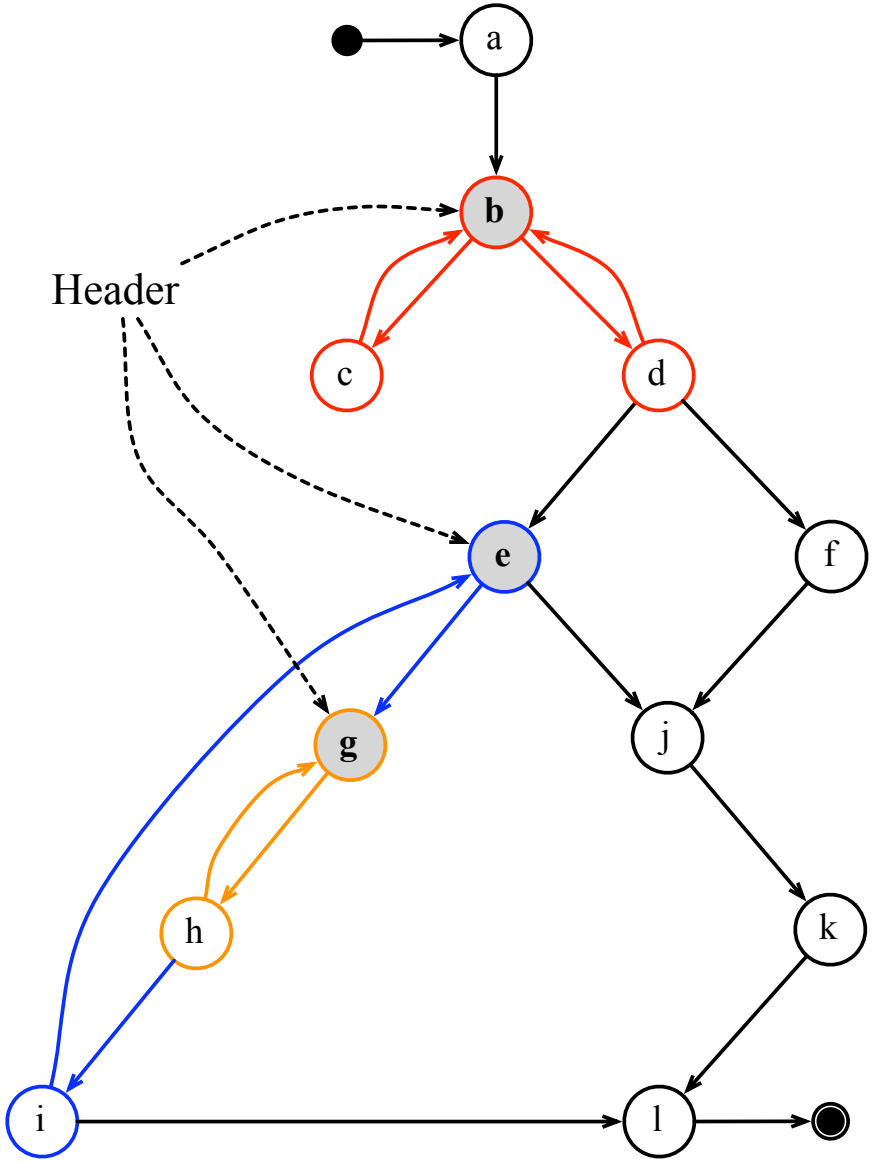
A node h is a header if there exists a node n, such that h dominates it, and there is an edge (n, h) in the CFG.



Finding the Loop Header

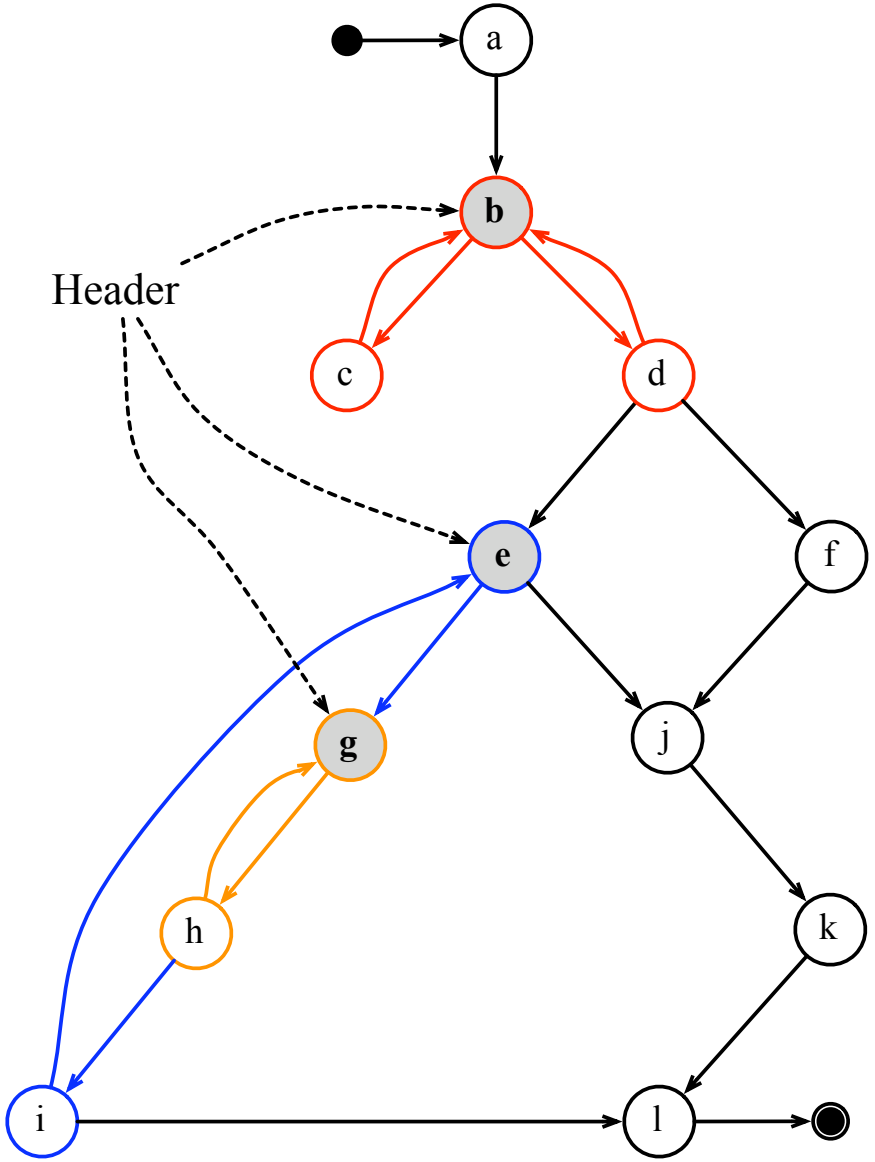
1) Can you explain why **this definition** of header is well defined only for loops of reducible flow graphs?

A node h is a header if there exists a node n, such that h dominates it, and there is an edge (n, h) in the CFG.



Finding the Loop Header

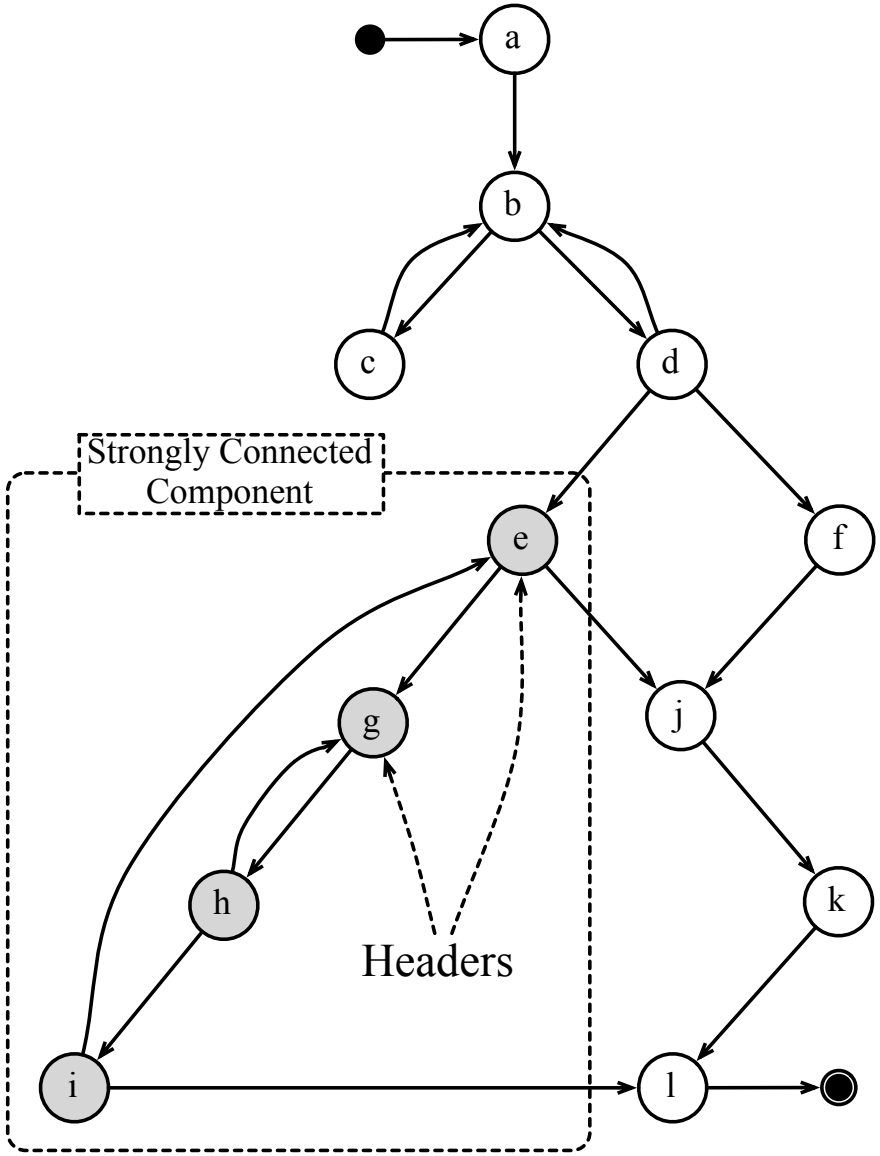
We need now a way to tell if a loop is nested within another loop. How can we do this?



Finding Nested Loops

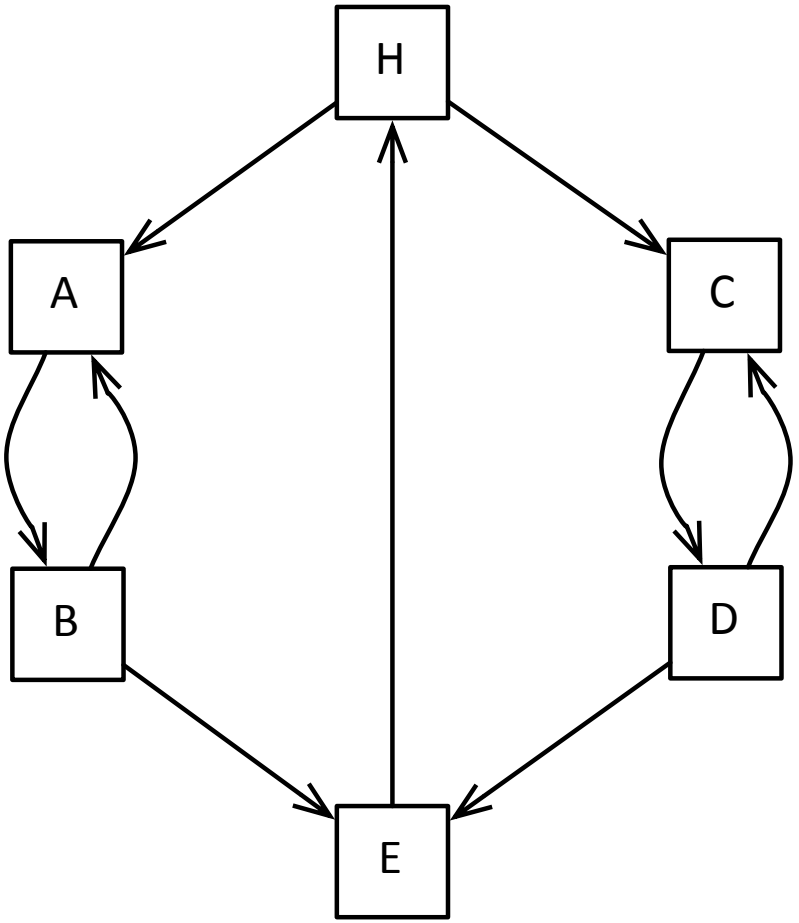
- If a strongly connected component contains two loop headers, h_1 and h_2, then the natural loops that sprout out of h_2 are nested within some of the loops that sprout out of h_1 if h_1 dominates h_2.

If a strongly connected component of a reducible CFG contains two loop headers, h_1 and h_2, then is it necessarily the case that one of these headers dominates the other. Why?



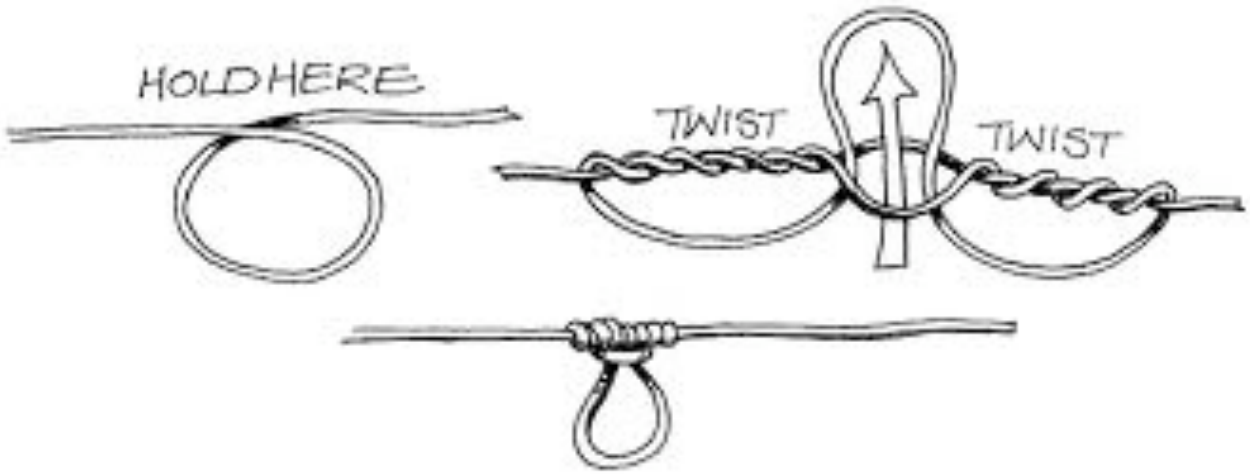
Finding Nested Loops

In this example, A and C are loop headers, but neither node dominates each other.



If a strongly connected component of a reducible CFG contains two loop headers, h_1 and h_2, then is it necessarily the case that one of these headers dominates the other. Why?

OPTMIZING LOOPS



DCC 888

Loop-Invariant Computation

- A computation is said to be loop-invariant if it always produces the same value at each iteration of the loop.
- A common optimization is to *hoist* invariant computations outside the loop.
- But, before we can optimize loop-invariant statements, we must be able to identify them.
- A statement t = a + b is invariant if *at least one* condition below is true about each operand:

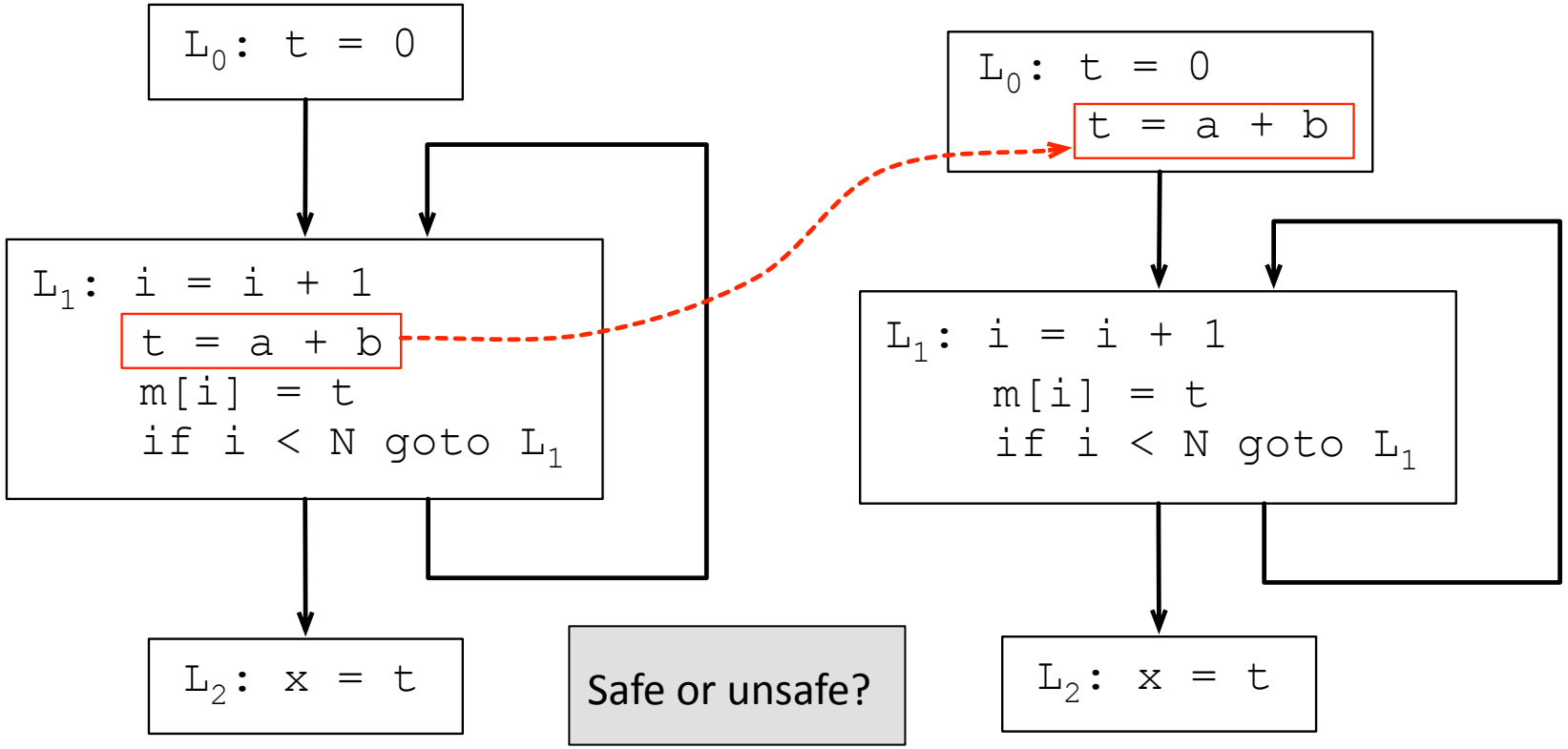
> Can you find a set of properties about the operands that determine that the computation is loop invariant? Hint: there are three conditions that can guarantee alone the invariance of that operand.

Loop-Invariant Computation

- A computation is said to be loop-invariant if it always produces the same value at each iteration of the loop.
- A common optimization is to *hoist* invariant computations outside the loop.
- But, before we can optimize loop-invariant statements, we must be able to identify them.
- A statement t = a + b is invariant if *at least one* condition below is true about each operand:
  - the operand is a constant
  - the operand is defined outside the loop
  - the operand is loop invariant, and no other definition of it reaches the statement

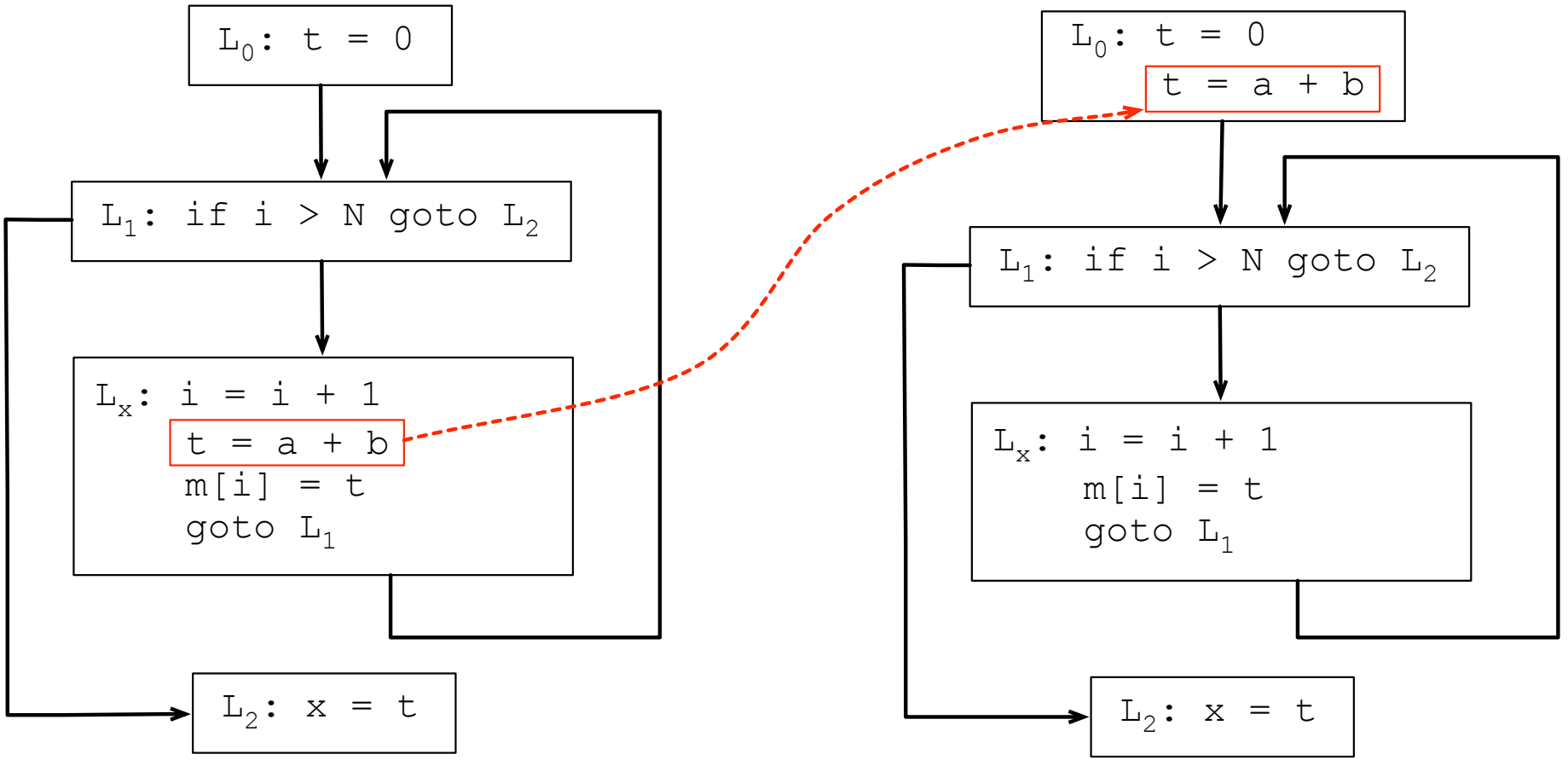
Loop-Invariant Code Hoisting

- The optimization that moves loop invariant computations to outside the loops is called *code hoisting*.
- Code hoisting is very effective, but it is not always safe:



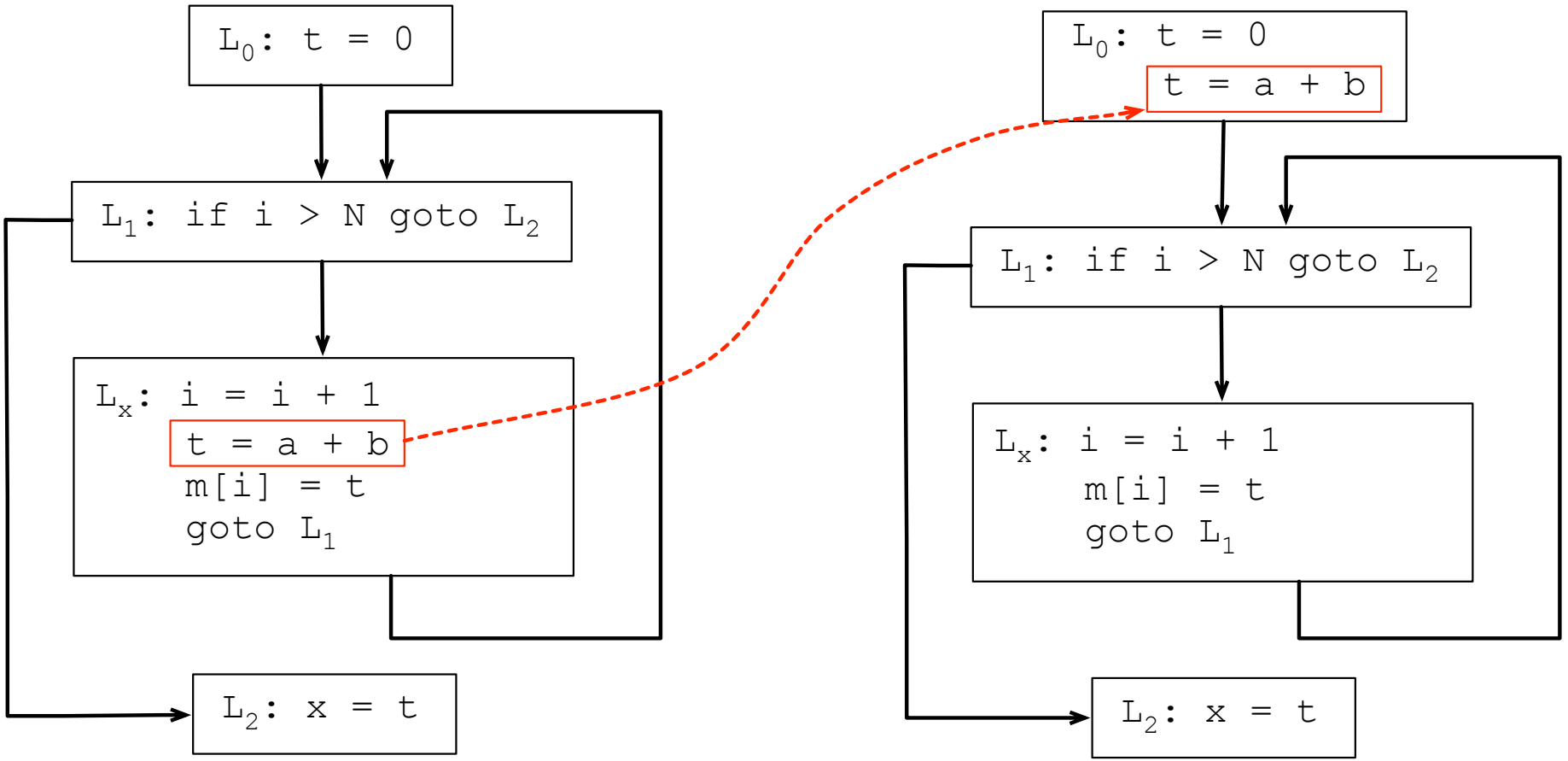
Loop-Invariant Code Hoisting

What about this case: is it safe or unsafe to move t = a + b to outside the loop?



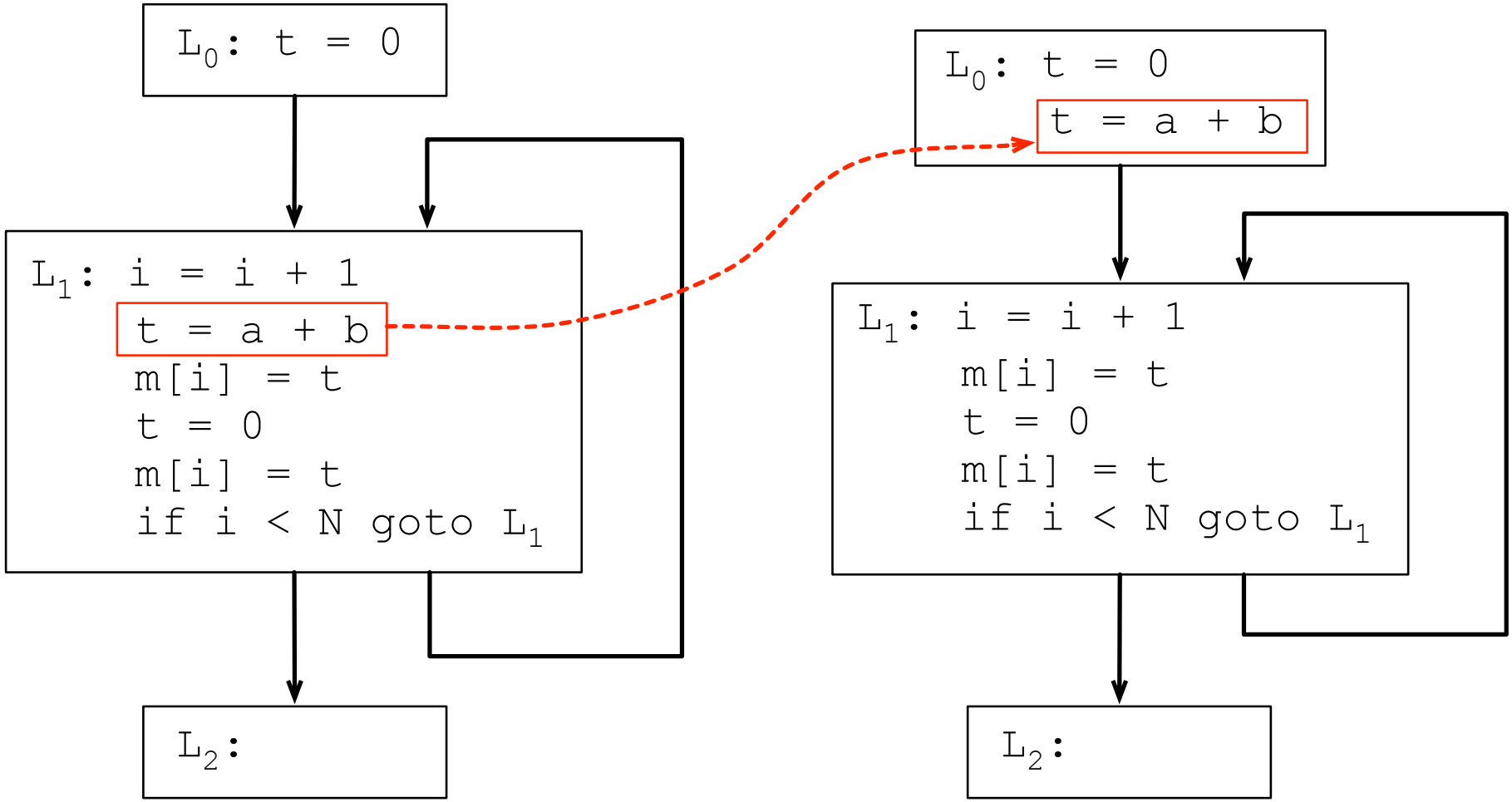
Loop-Invariant Code Hoisting

This case is unsafe, because the computation t = a + b will be executed, even if the loop iterates zero times, in the optimized version of the program.



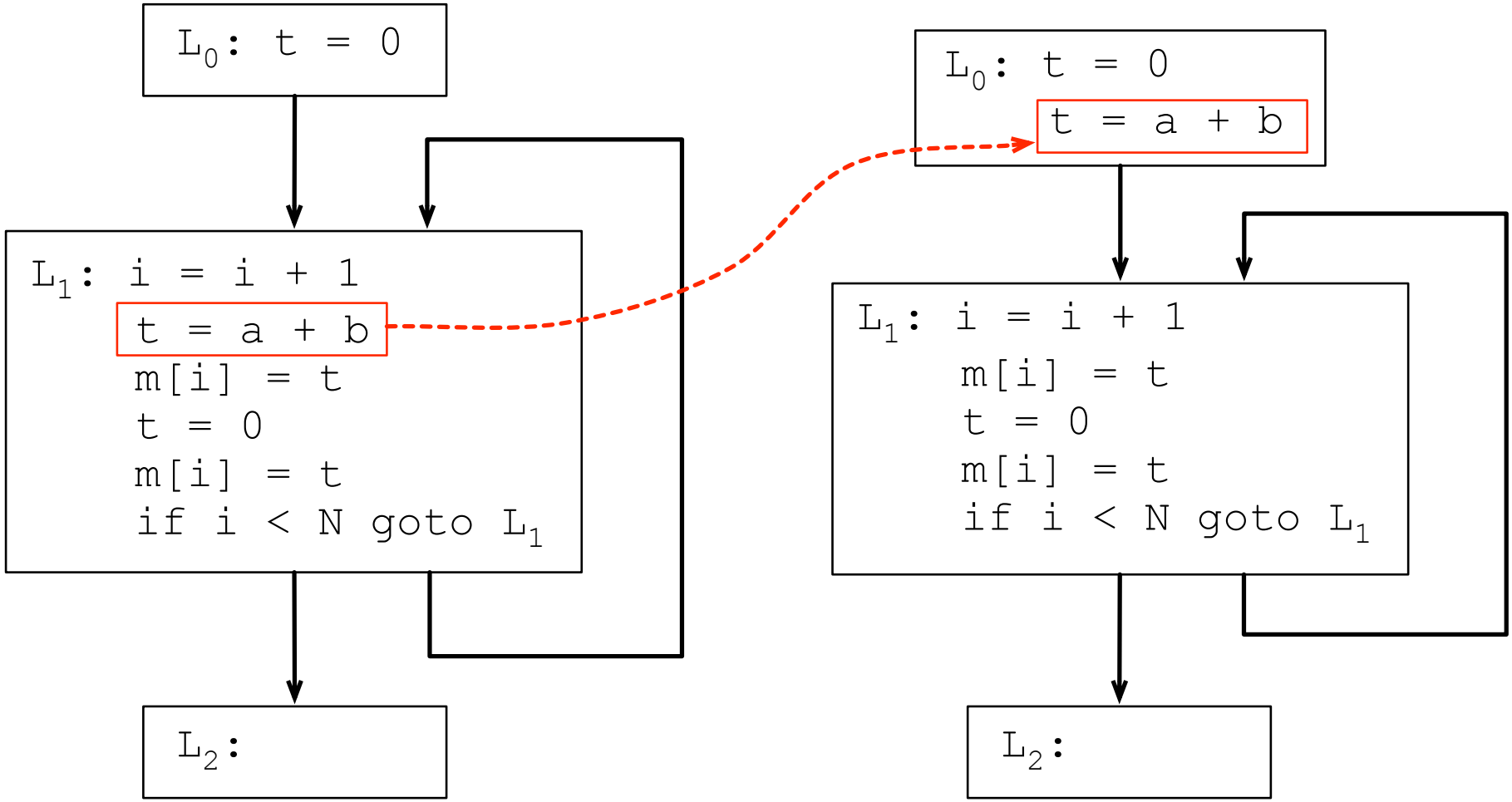
Loop-Invariant Code Hoisting

Again: safe or unsafe?



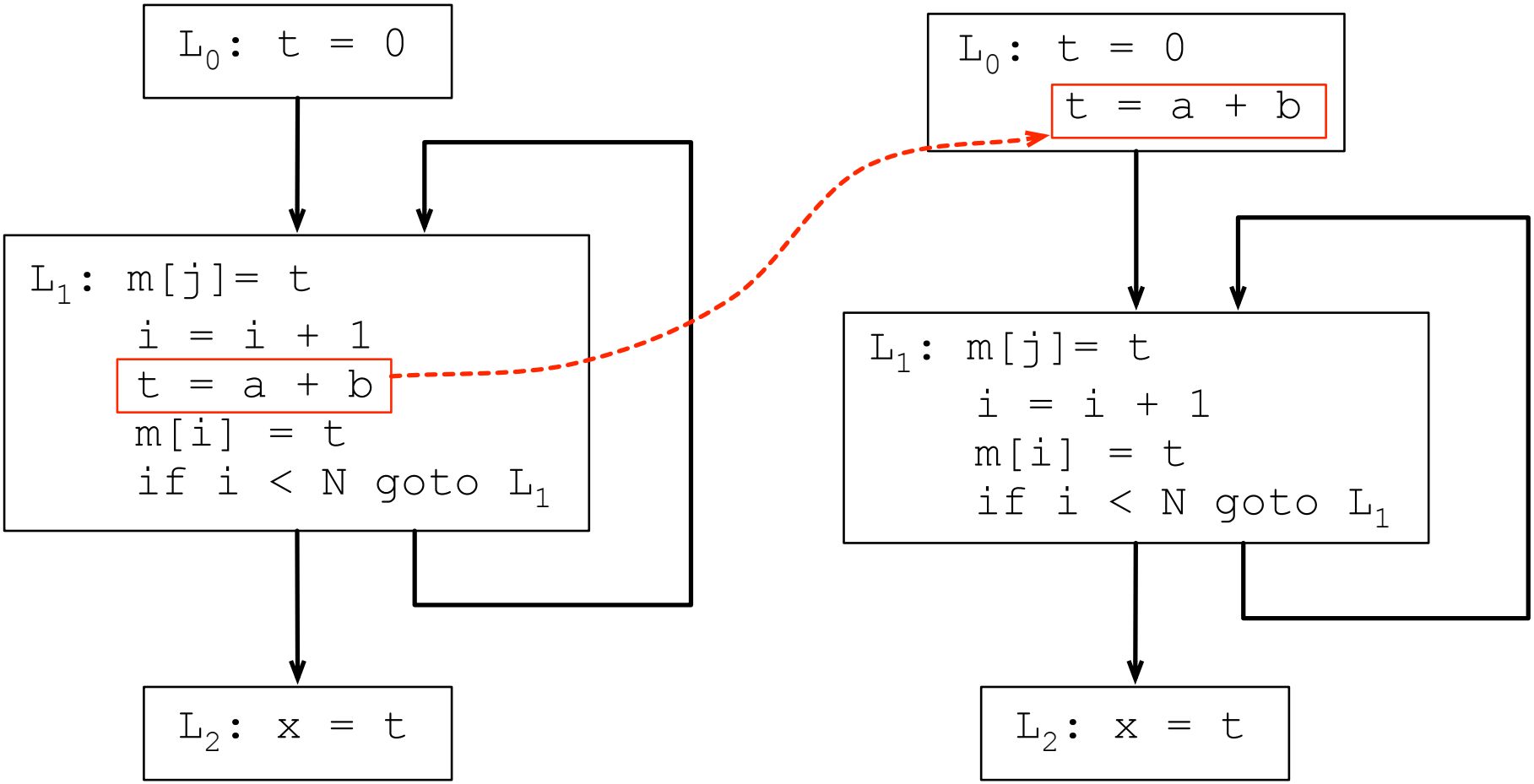
Loop-Invariant Code Hoisting

This optimization is also unsafe, because it is changing the value of t when the operation m[i] = t happens. It should be always a + b, but in the optimized code it may be 0 as well.



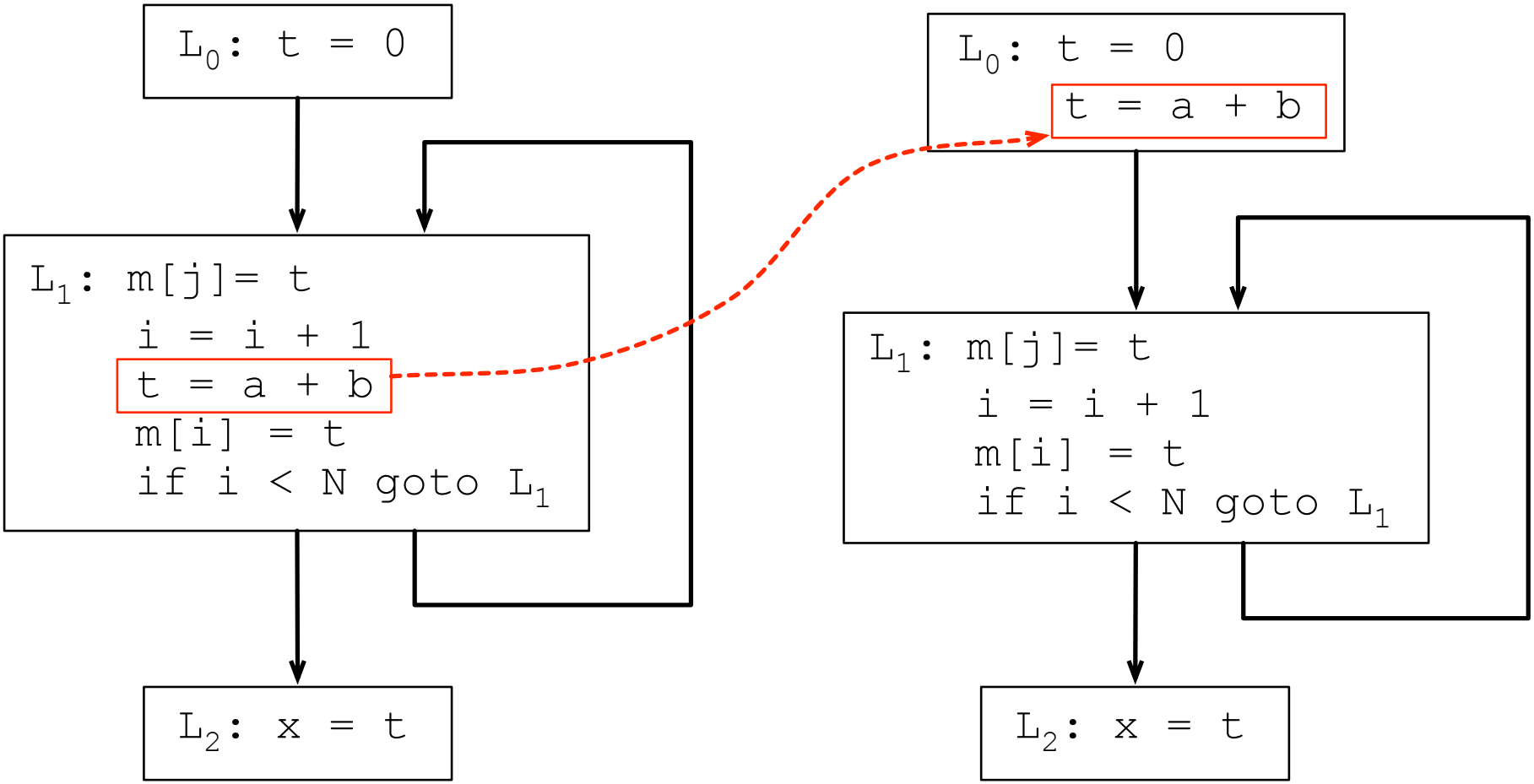
Loop-Invariant Code Hoisting

The last one:
safe or unsafe?



Loop-Invariant Code Hoisting

This optimization is unsafe, because it is changing the definition that reaches the statement m[j] = t in the first iteration of the loop. In this case, t should be initially zero.



PPGCC
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO DCC-UFMG

Loop-Invariant Code Hoisting

- So, when is it safe to move invariant computations outside the loop?

Loop-Invariant Code Hoisting

- So, when is it safe to move invariant computations outside the loop?
- We can move an statement t = a + b, located at a program point d, if these three conditions apply onto the code we are moving away:
  1. d dominates all loop exits at which t is live-out
  2. there is only one definition of t inside the loop
  3. t is not live-out of the loop header, e.g., the place where we are placing the statement t = a + b

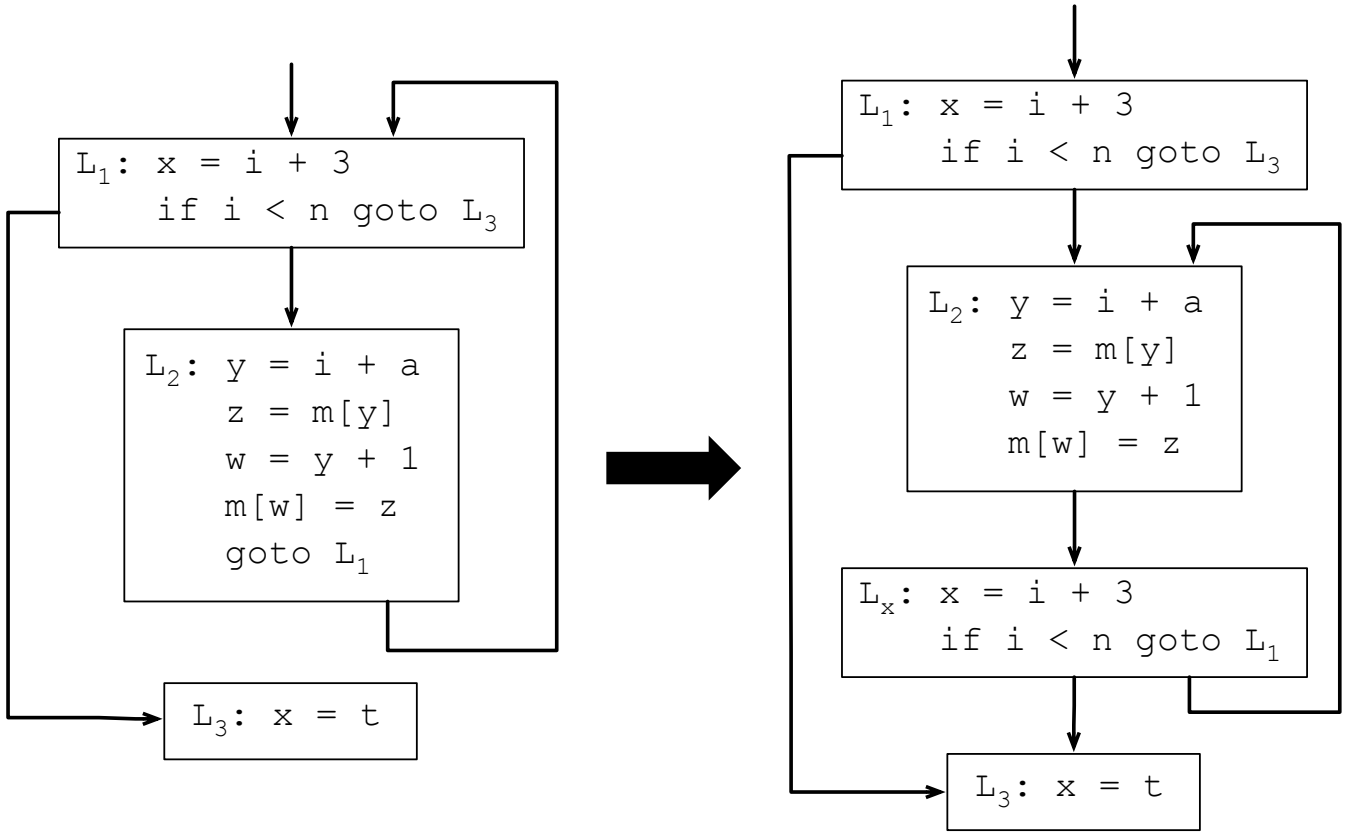
Condition (1) tends to prevent many hoistings. Is there a way to change the CFG, so that it becomes true?

Loop Inversion

- Loop inversion consists in transforming a while loop into a repeat-until loop.
- It provides a safe place where we can move invariant computation; hence, it ensures the validity of condition (1)

1) We have already mentioned loop inversion before. Do you remember the context?

2) Why is the second program more efficient than the first, in general?



PPGCC
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO DCC-UFMG

Induction Variables

- **Basic induction variable**: the variable i is a basic induction variable in a loop if the only definitions of i within that loop are of the form i = i + c or i = i – c, where c is loop invariant.
- **Derived induction variables**: the variable k is a derived induction variable in loop L if these conditions all apply:
  1. there is only one definition of k within L, of the form k = j * c or k = j + d, where j is an induction variable and c, d are loop-invariant.
  2. if j is a derived induction variable in the family of i, then:
     1. the only definition of j that reaches k is the one in the loop
     2. there is no definition of i on any path between the definition of j and the definition of k.

Induction Variables

1) What is each of these functions doing?

2) Which are the induction variables in these functions?

3) Which induction variables are basic and derived?

```
zeraCol(int* m, int N, int C, int W) {
  int i, j;
  for (i = 0; i < N; i++) {
    j = C + i * W;
    m[j] = 0;
  }
}

zeraDig(int* m, int N, int C) {
  int i, j;
  for (i = 0; i < N; i++) {
    j = i * C + i;
    m[j] = 0;
  }
}
```

Strength Reduction

- Multiplication is usually more expensive than addition.
  - This is true in many different processors.
  - Hence, an optimization consists in replacing multiplications by additions whenever possible.

How could we replace multiplications by additions in these functions below?

```
zeraDig(int* m, int N, int C) {
  int i, j;
  for (i = 0; i < N; i++) {
    j = i * C + i;
    m[j] = 0;
  }
}
```

```
zeraCol(int* m, int N, int C, int W) {
  int i, j;
  for (i = 0; i < N; i++) {
    j = C + i * W;
    m[j] = 0;
  }
}
```

Strength Reduction

```
zeraCol(int* m, int N, int C,
    int W) {
  int i, j;
  for (i = 0; i < N; i++) {
    j = C + i * W;
    m[j] = 0;
  }
}
```

```
zeraCol(int* m, int N, int C,
    int W) {
  int i, j = C;
  for (i = 0; i < N; i++) {
    j += W;
    m[j] = 0;
  }
}
```

```
zeraDig(int* m, int N, int C) {
  int i, j;
  for (i = 0; i < N; i++) {
    j = i * C + i;
    m[j] = 0;
  }
}
```

```
zeraDigSR(int* m, int N, int C) {
  int i, j;
  int j0 = 0;
  for (i = 0; i < N; i++, j0 += C) {
    j = j0 + i;
    m[j] = 0;
  }
}
```

Could you think on a systematic way to perform this optimization?

Strength Reduction

The Basic Optimization:

- Let i be a basic induction variable initialized with 'a', and augmented by 'b' at each iteration, e.g., (`for int i = a; i < …, i += b`)
- Let j be a derived induction variable of the form j = i * c
  - we create a new variable j'.
    - initialize j' outside the loop, with j' = a * c
  - after each assignment i = i + b, we create an assignment j' = j' + c * b
    - we should compute c * b outside the loop
  - replace the unique assignment to j by j = j'

How does this transformation work for the program below?

```
for (i = a; i < ...; i += b) {
   j = i * c;    ... = j;
}
```

Strength Reduction

```
for (i = a; i < ...; i += b) {
    j = i * c;
    ... = j;
}
```

```
j' = a * c;
for (i = a; i < ...; i += b) {
    j = i * c;
    ... = j;
}
```

- we create a new variable j'.
  - initialize j' outside the loop, with j' = a * c
- after each assignment i = i + b, we create an assignment j' = j' + c * b
  - we should compute c * b outside the loop
- replace the unique assignment to j by j = j'

```
j' = a * c;
t = c * b;
for (i = a; i < ...; i += b) {
    j = i * c;
    j' = j' + t
    ... = j;
}
```

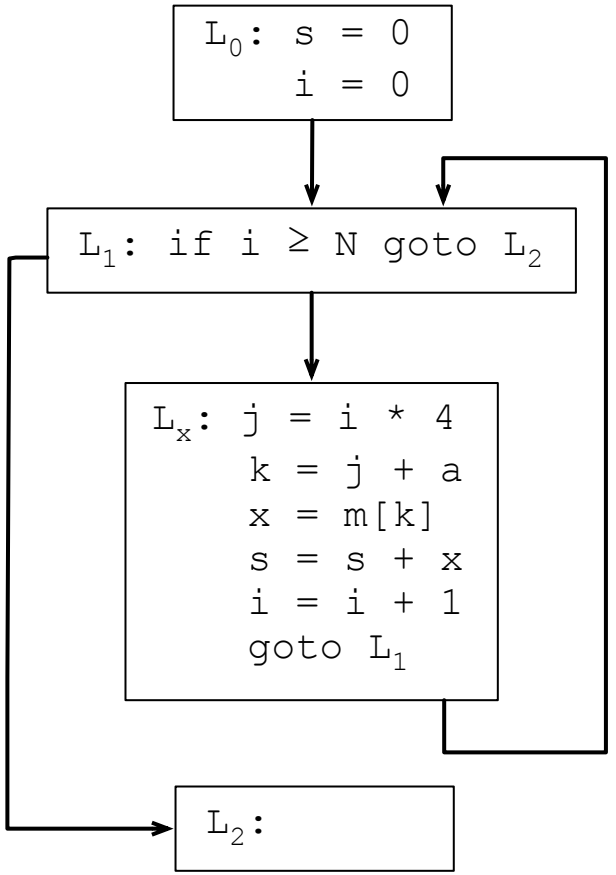
```
j' = a * c;
t = c * b;
for (i = a; i < ...; i += b) {
    j' = j' + t
    ... = j';
}
```

Strength Reduction

The Basic Optimization:

- Let i be a basic induction variable initialized with 'a', and augmented by 'b' at each iteration, e.g., (`for int i = a; i < …, i += b`)
- Let j be a derived induction variable of the form j = i * c
  - we create a new variable j'.
    - initialize j' outside the loop, with j' = a * c
  - after each assignment i = i + b, we create an assignment j' = j' + c * b
    - we should compute c * b outside the loop
  - replace the unique assignment to j by j = j'

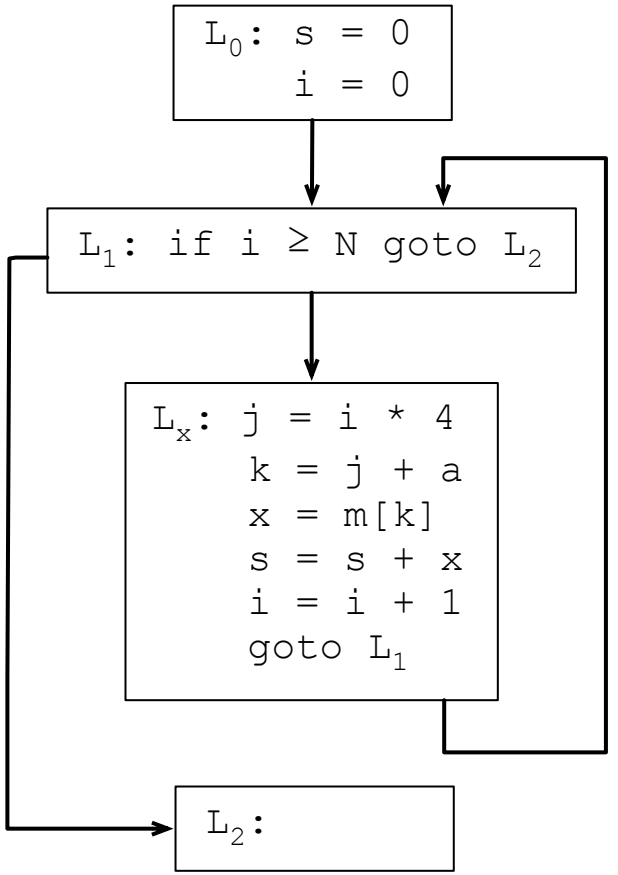
Which are the basic and derivided induction variables in this code below?



Strength Reduction

- i is a basic induction variable, and at the n-th iteration of the loop, we have that $i_n = 0 + n$
- j is a derived induction variable, and at the n-th iteration of the loop we have that $j_n = 0 + i_n * 4 = 0 + n * 4$
- k is a derived induction variable, and at the n-th iteration of the loop we have that $k_n = a + i_n = a + n * 4$

3) Can you apply strength reduction on this example program?

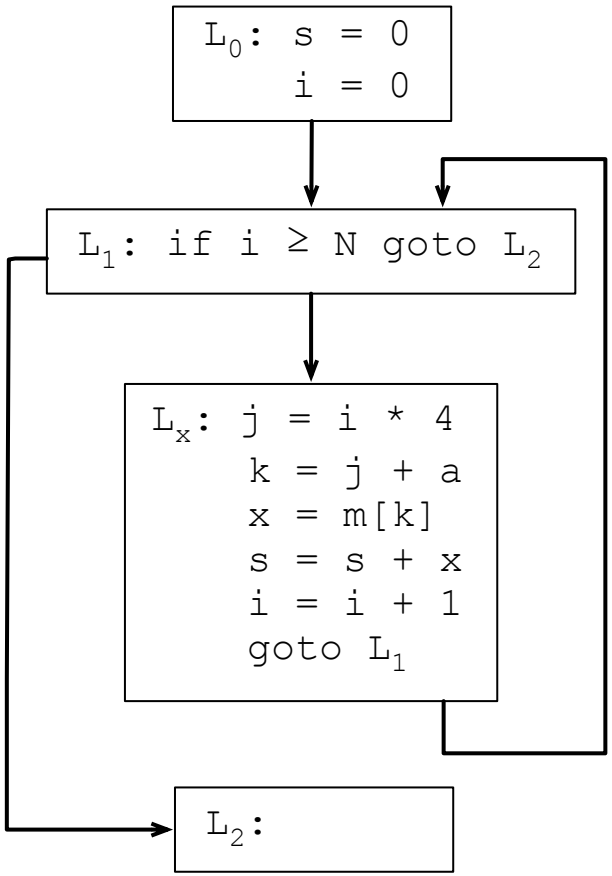


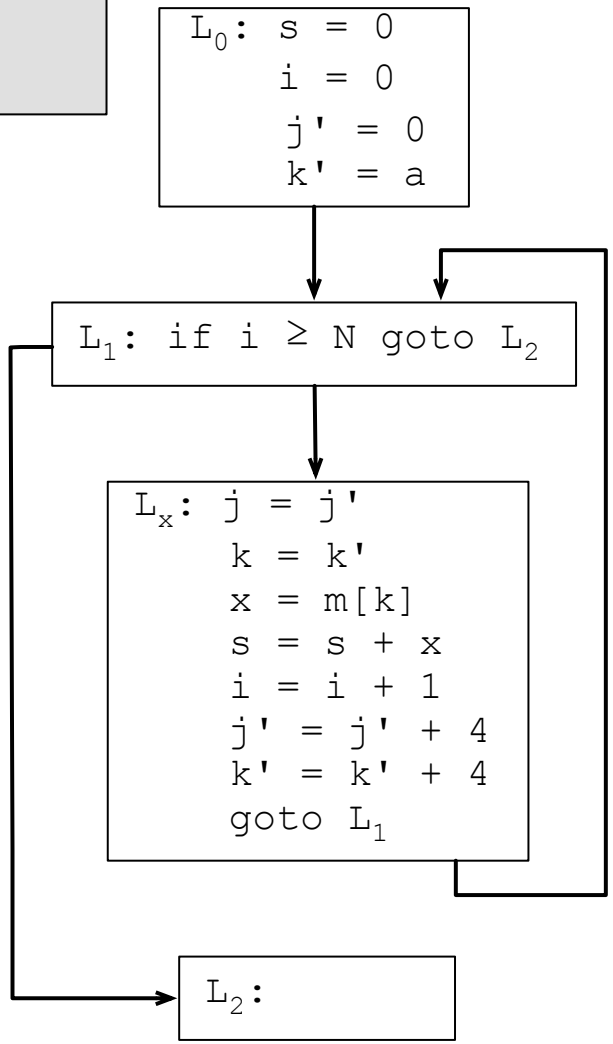
1) How should j' and k' be initialized?

2) How should these variables be incremented?

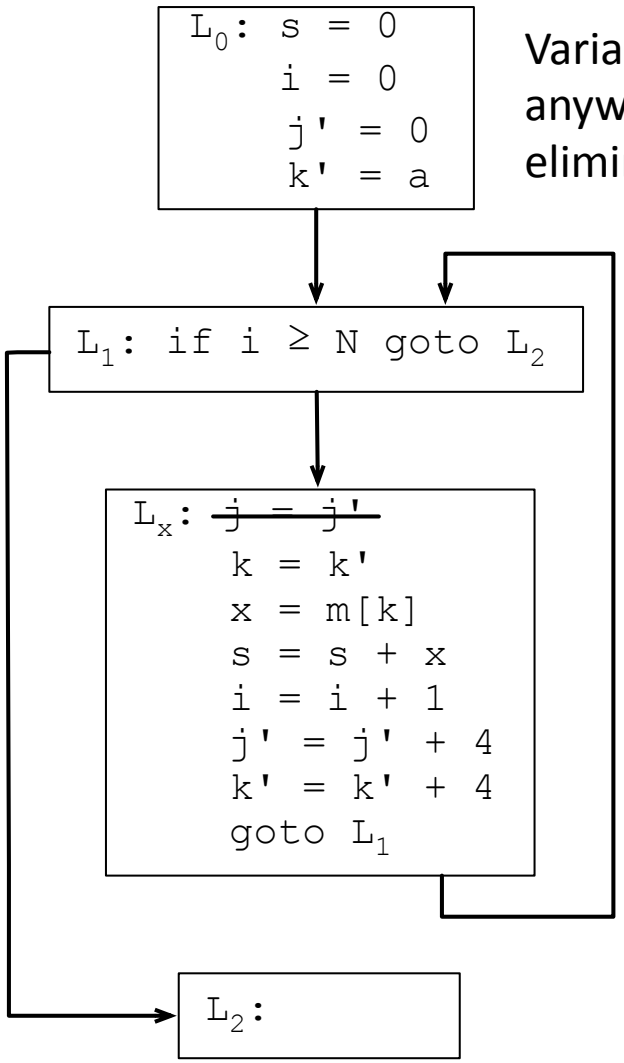
Strength Reduction

The program on the right should go through some obvious optimizations. Which ones?

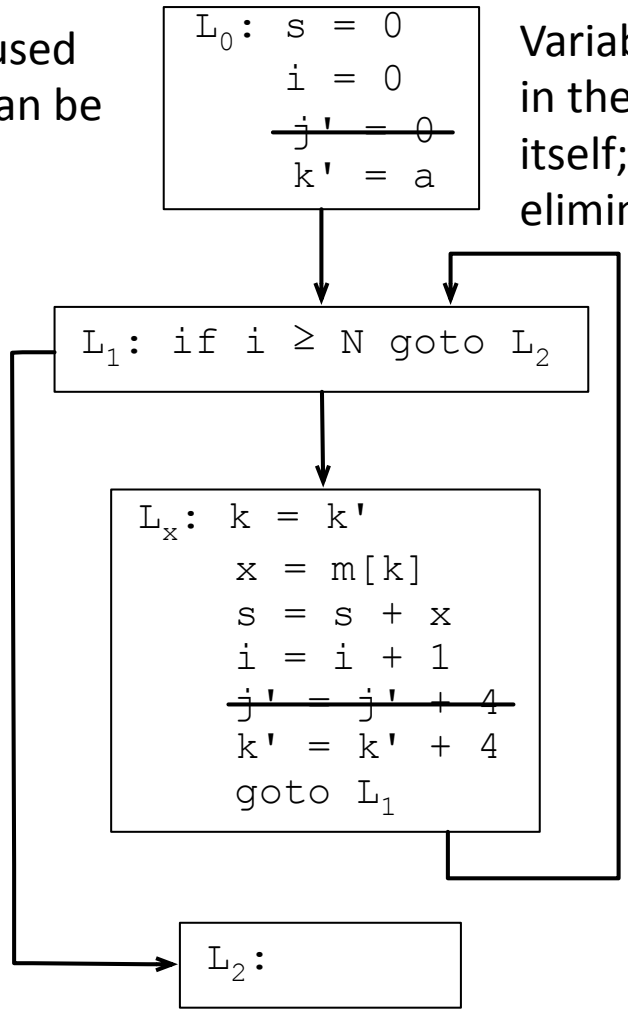




Dead Code Elimination



Variable j is not used anywhere, and can be eliminated.

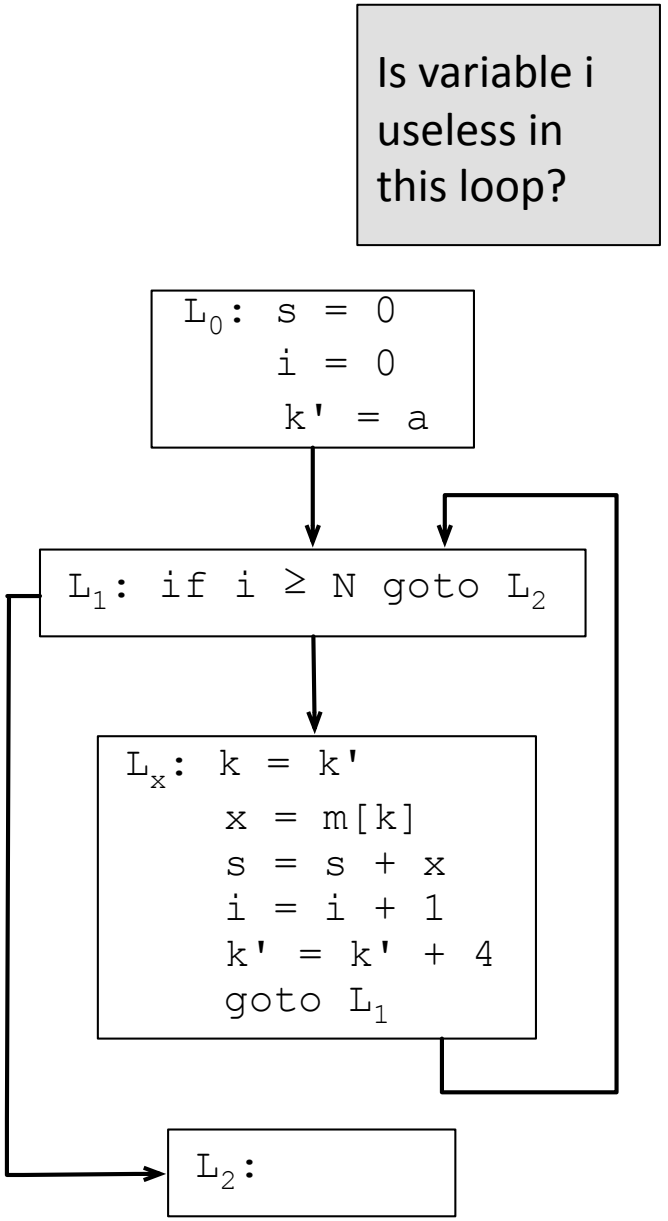


Variable j' is only used in the definition of itself; hence, it can be eliminated as well.

Look at the original program again. Variable j was useful there, but it is useless now. Why?

"Almost" Useless Variable

- A variable i is almost useless if:
  - it is used only in these two situations:
    - comparisons against loop-invariant values
    - in the definition of itself
  - there is some other induction variable related to i that is not useless.
- An almost-useless variable may be made useless by modifying the comparison to use the related induction variable.



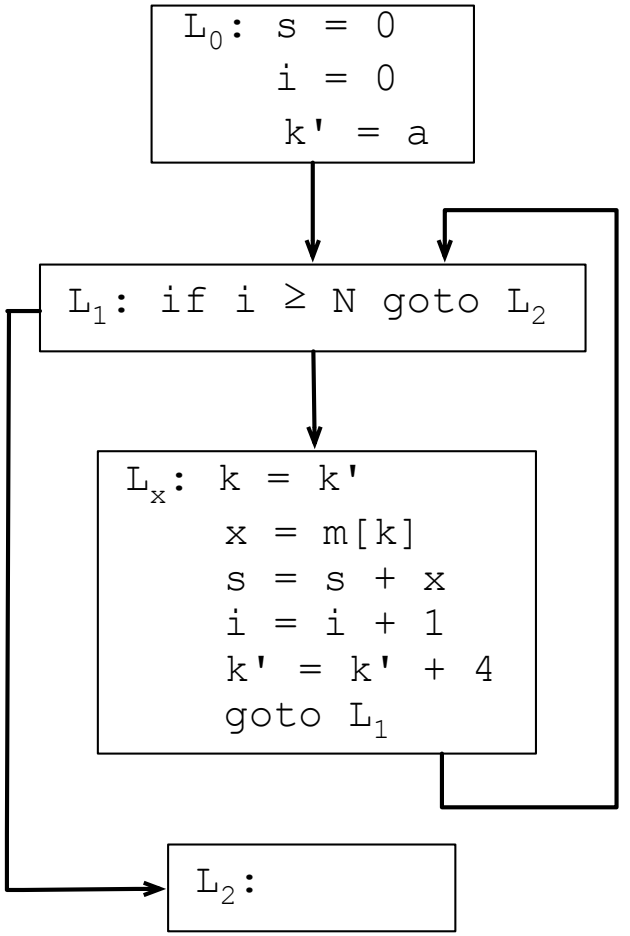
"Almost" Useless Variable

1) How can we remove the useless variable i?

2) We want to change the comparison i ≥ N to use a variable different than i

We know that k' = 4 * i + a

How could we rewrite this comparison?



"Almost" Useless Variable

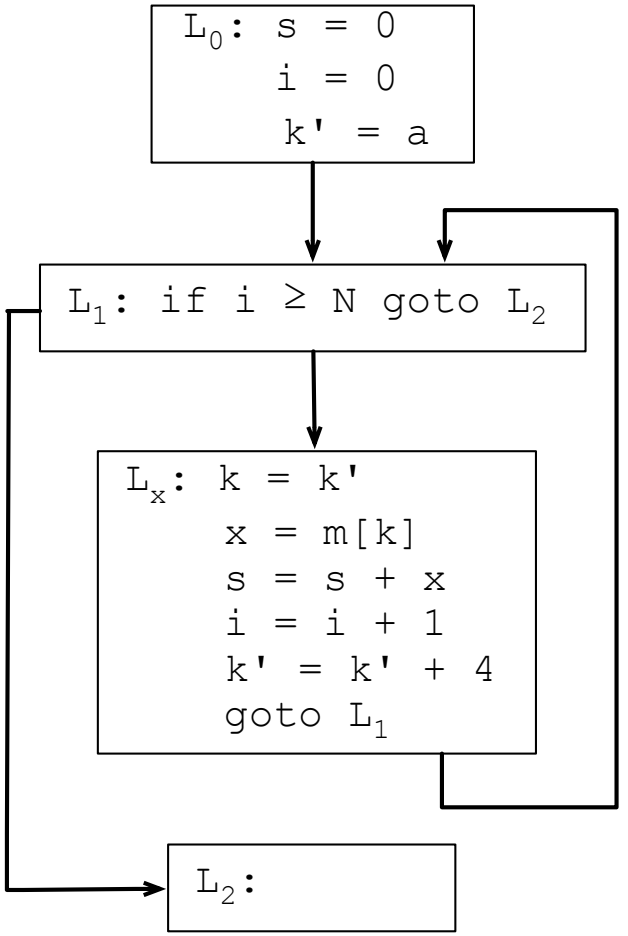
$k' = 4 * i + a \Rightarrow i = k'/4 - a/4$

$i = k'/4 - a/4 \ \wedge \ i \geq N \ \Rightarrow \ k'/4 - a/4 \geq N$

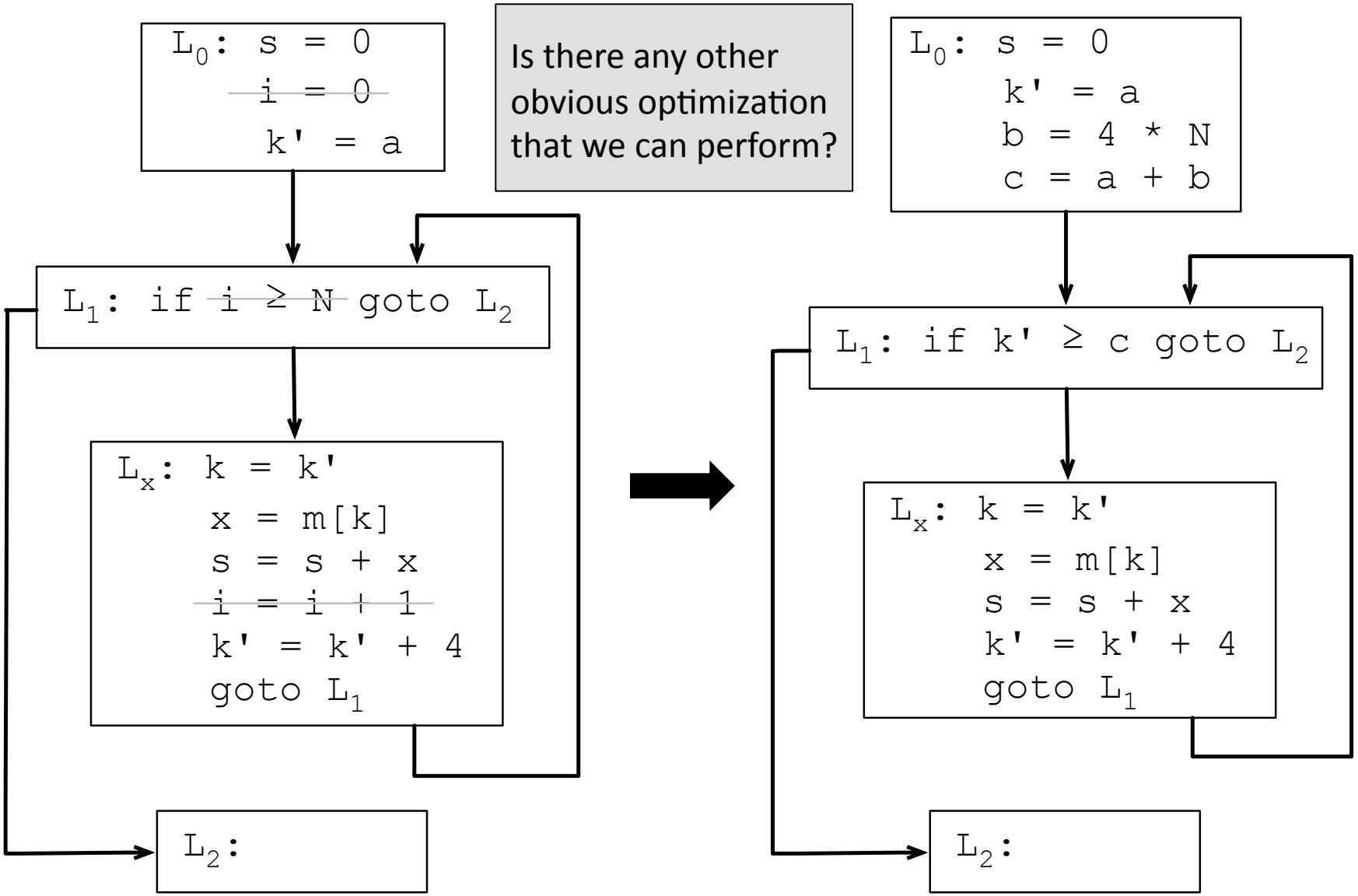
$k'/4 - a/4 \geq N \ \Rightarrow \ k' \geq 4 * N + a$

We can move 4 * N + a to outside the loop:

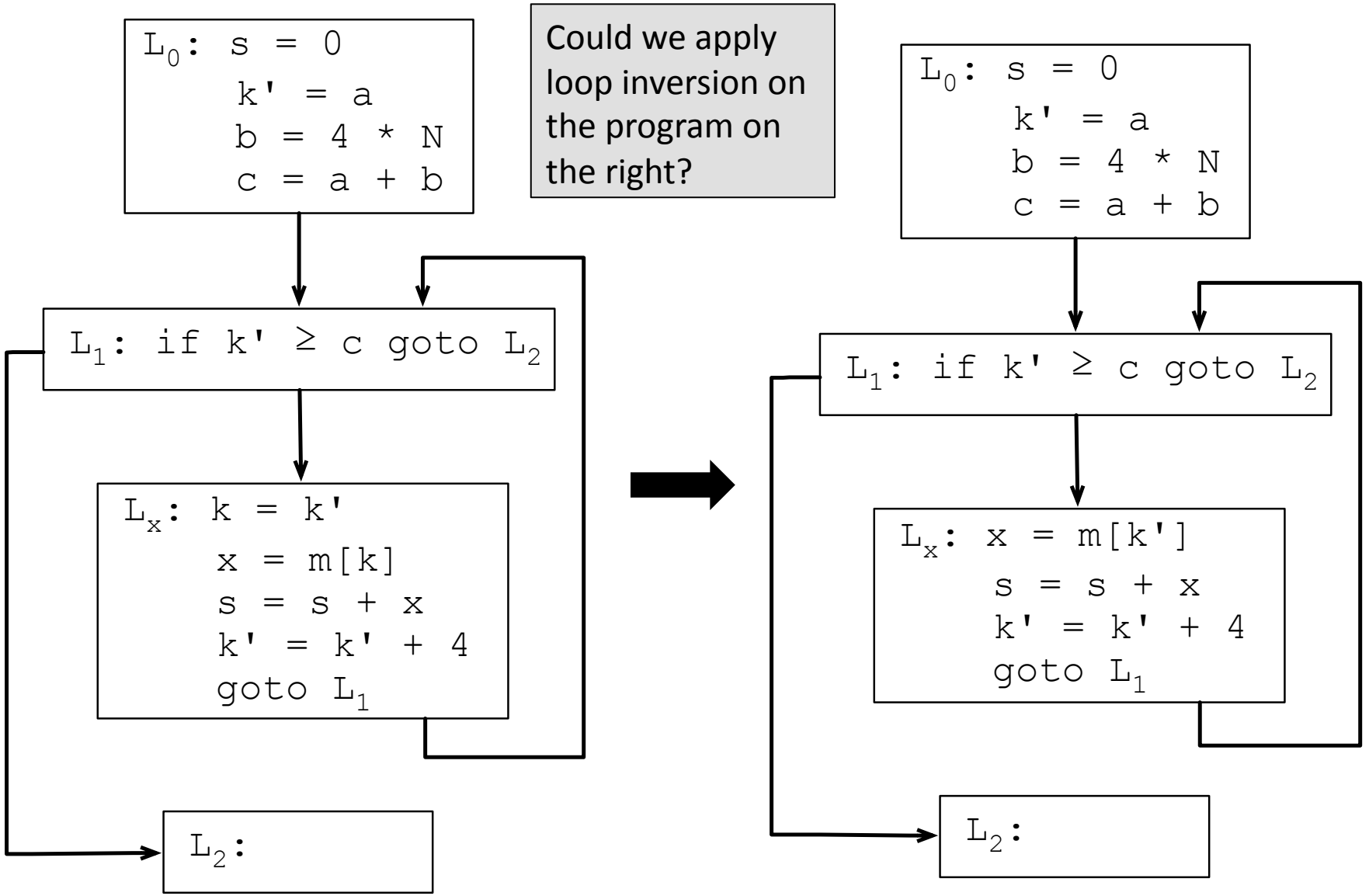
$b = 4 * N; c = a + b; k' \geq c$



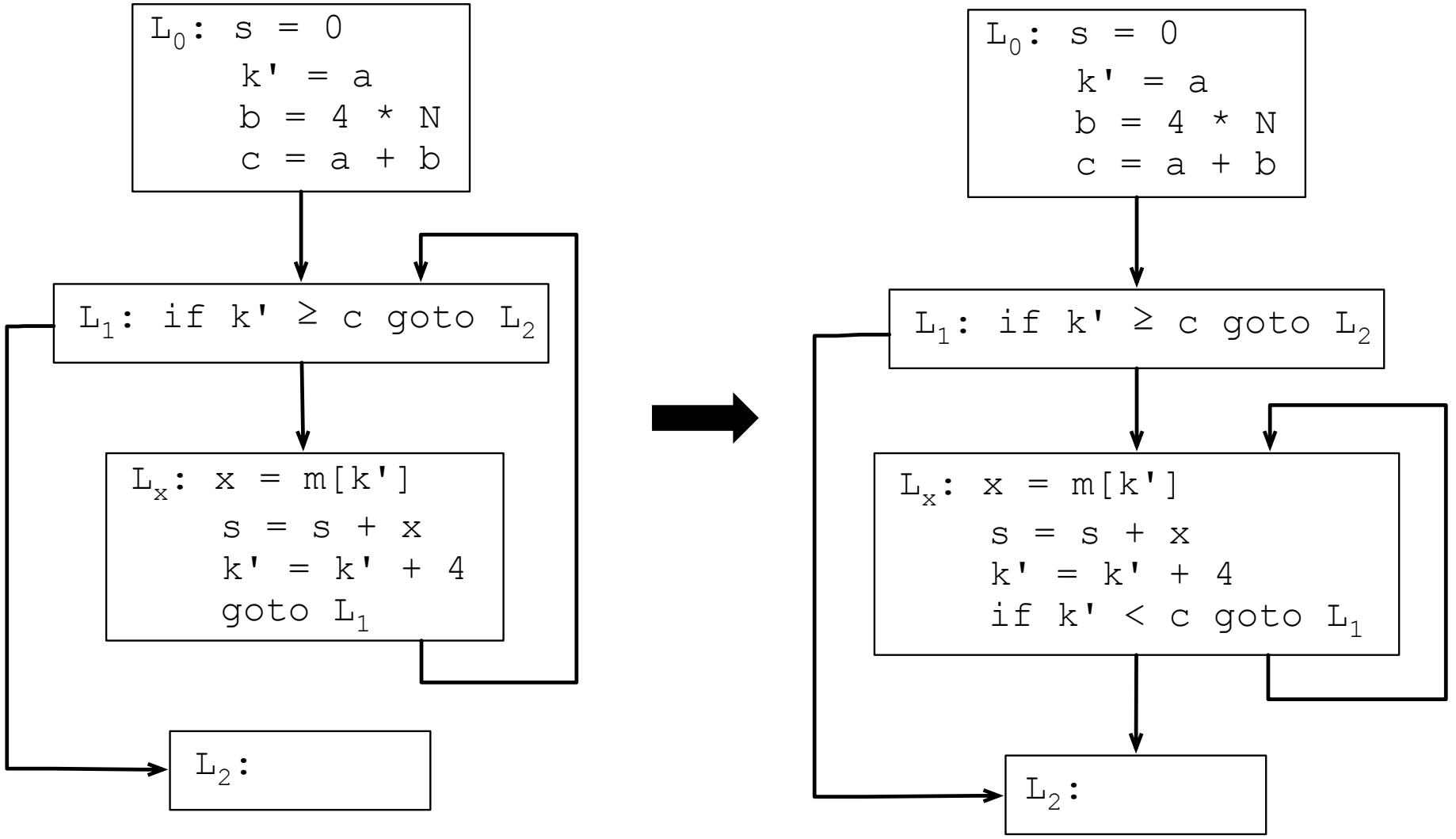
"Almost" Useless Variable



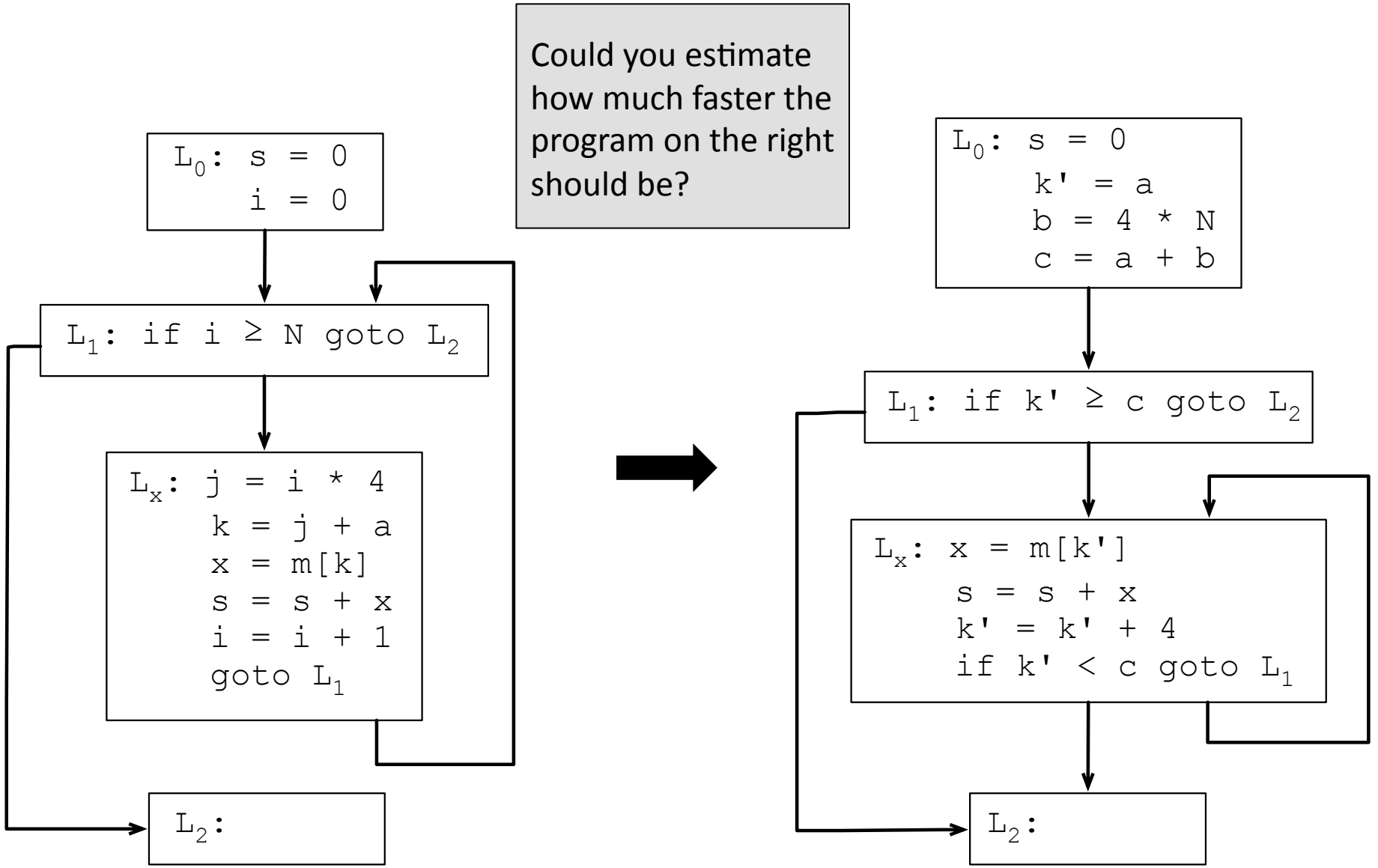
Copy Propagation



Loop Inversion (Again)



Comparing Original and Optimized Loop

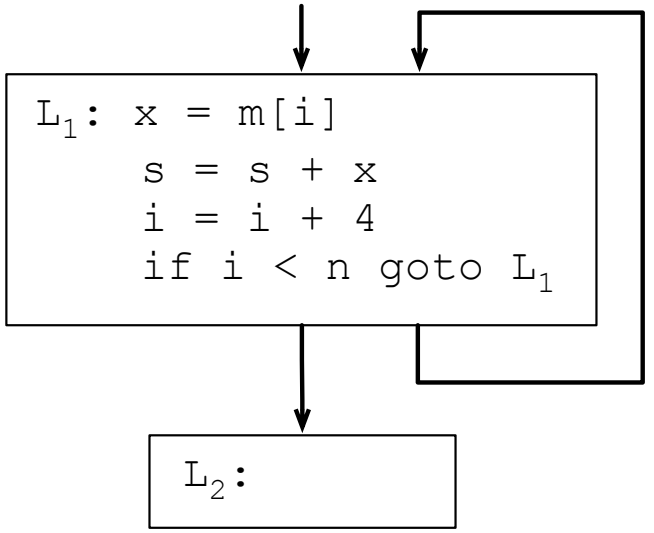


Loop Unrolling

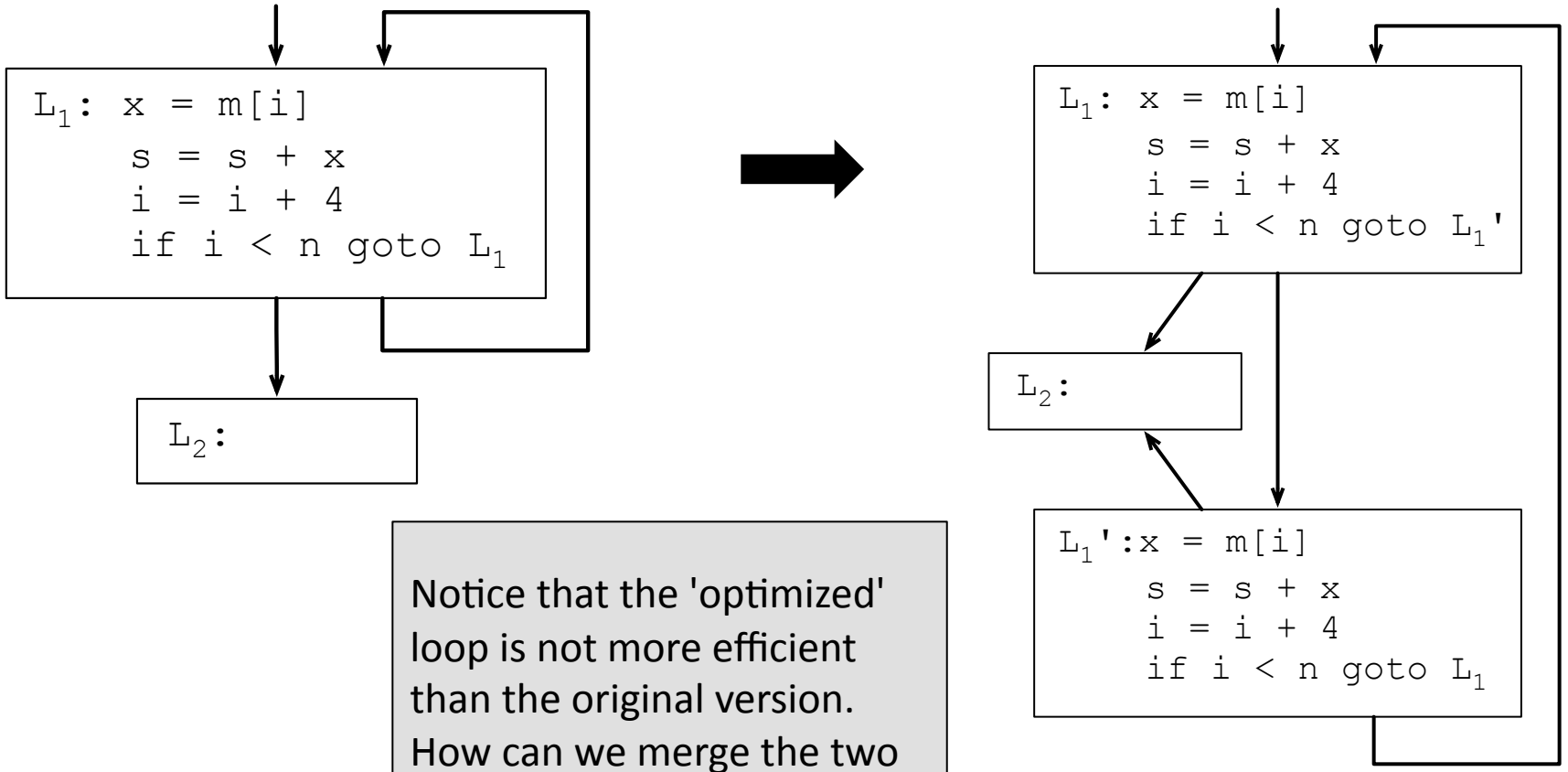
- Loop unrolling is an optimization that consists in transforming the loop, so that we execute more of its commands in less iterations.
- we can unroll – once – a loop L with header node h and back edges (s_i, h) as follows:
  1. Copy the nodes to make a loop L' with header h' and back edges (s_i', h')
  2. Change all the back edges in L from (s_i, h) to (s_i, h')
  3. Change all the back edges in L' from (s_i', h') to (s_i', h)

- Replace the induction variables of L' by those of L, incremented by the factor present into a single iteration of the loop.

How could we apply steps 1-3 onto this loop?

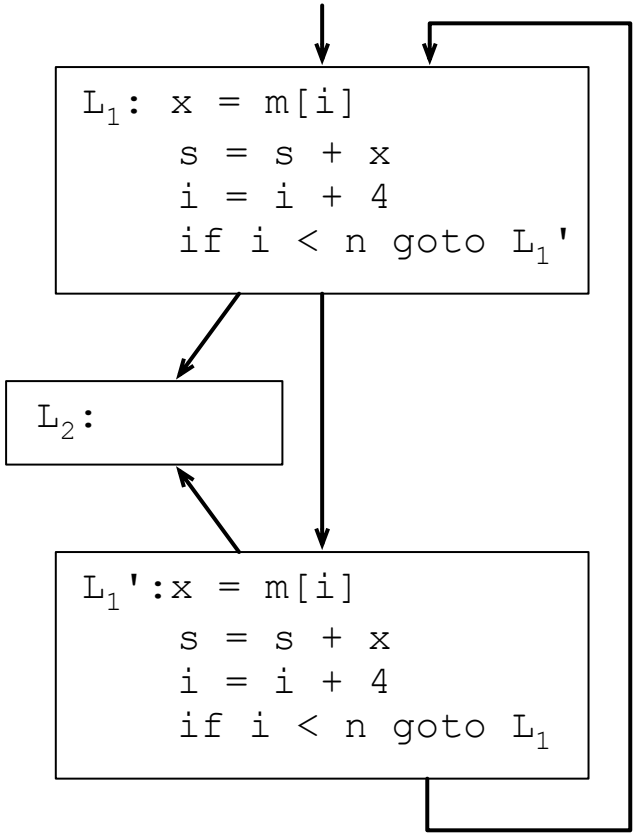


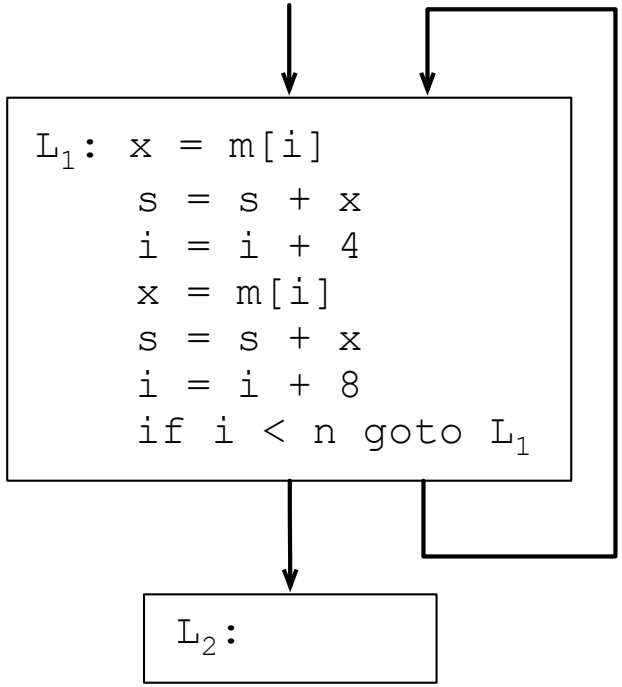
Loop Unrolling



Notice that the 'optimized' loop is not more efficient than the original version. How can we merge the two blocks of the loop?

Loop Unrolling

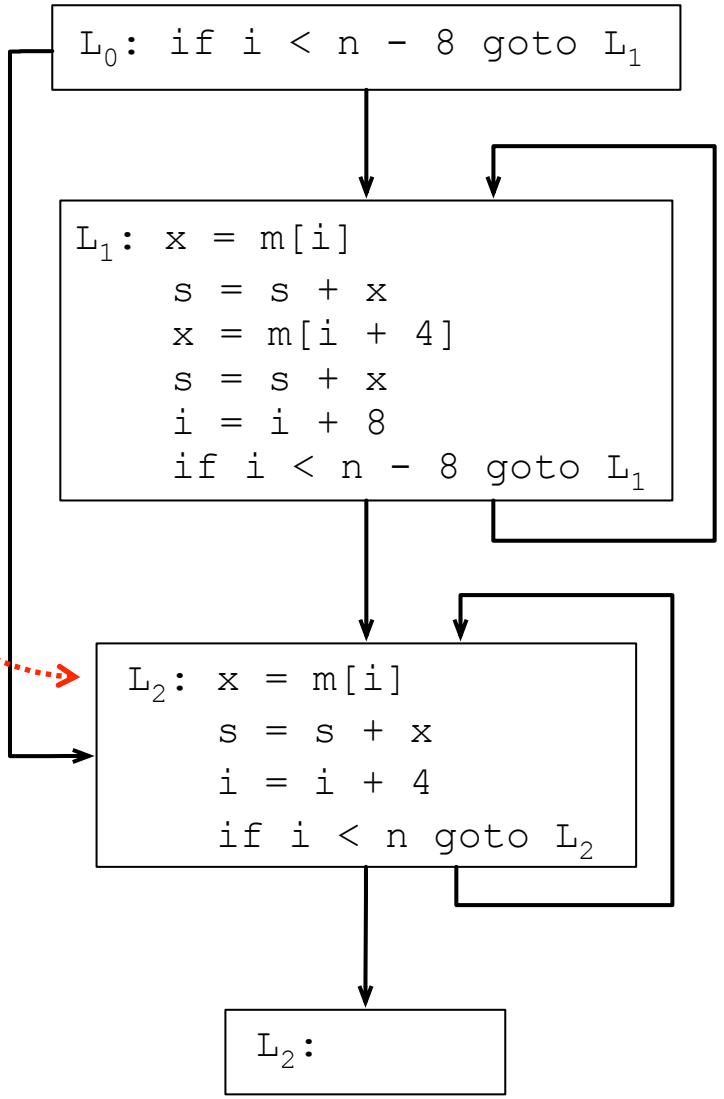




Serious problem: what if the optimized loop executes an odd number of iterations?

Loop Unrolling + Epilogue

- If a loop is unrolled a factor of N times, then we need to insert after it an epilogue, which will execute (T mod N) iterations, where T is the total number of times an actual execution of the loop iterates.



A Bit of History

- Compiler writers have been focusing on loops since the very beginning of their science.
- Lowry and Medlock described the induction variable optimization. They seem to be the first to talk about dominators as well.
- The notion of reducible flow graphs was introduced by F. Allen, which, by the way, got the Turing Award!

- Lowry, E. S. and Medlock, C. W. "Object Code Optimization". CACM 12(1), 13-22 (1969)
- Allen, F. E. "Control Flow Analysis". SIGPLAN Notices 23(7) 308-317 (1970)