PROGRAMMING LANGUAGES LABORATORY
Universidade Federal de Minas Gerais - Department of Computer Science

STATIC SINGLE ASSIGNMENT FORM

PROGRAM ANALYSIS AND OPTIMIZATION – DCC888

Fernando Magno Quintão Pereira
fernando@dcc.ufmg.br

The material in these slides have been taken from Chapter 19 – Static Single-Assignment Form – of "Modern Compiler Implementation in Java – Second Edition", by Andrew Appel and Jens Palsberg.

Control Flow Graphs Revisited

- We have seen how to produce and visualize the control flow graph of a program using LLVM:

```
$> clang -c -emit-llvm max.c -o max.bc

$> opt -view-cfg max.bc
```

- We have also seen how to use LLVM's opt to analyze and transform a program:

```
$> opt -mem2reg max.bc -o max.ssa.bc

$> opt -view-cfg max.ssa.bc
```

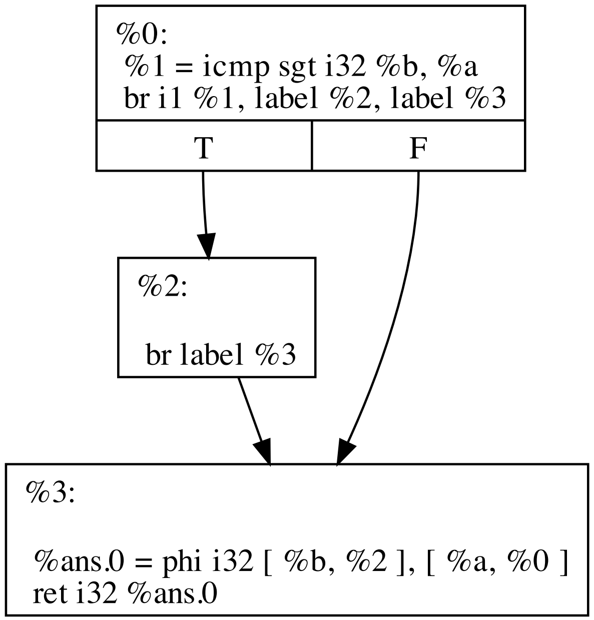
What do you think is the difference between max.bc and max.ssa.bc?

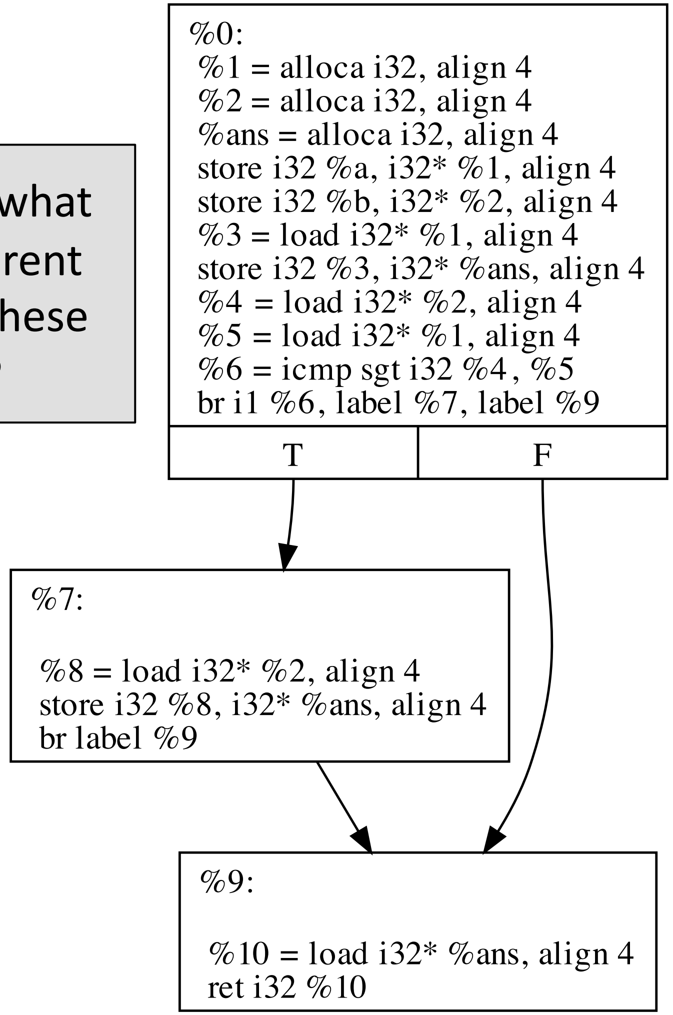
Control Flow Graphs Revisited

```c
int max(int a, int b) {
  int ans = a;
  if (b > a) {
    ans = b;
  }
  return ans;
}
```

`clang -c -emit-llvm max.c -o max.bc`

So, again, what is the different between these two CFGs?





`opt -mem2reg max.bc -o max.ssa.bc`

The Static Single Assignment Form

- The Static Single Assignment Form is an intermediate program representation that has the following property:

Can you explain this name: static single assignment?

Each variable in a SSA form program has only one definition site

- There have been many smart things in the science of compiler writing, but SSA form is certainly one of the smartest.
  - It simplifies many analyses and optimizations.
  - Today it is used in virtually any compiler of notice, e.g., gcc, LLVM, Jikes, Mozilla's IonMonkey, Ocelot, etc

SSA form is like the Highlander: “there can be only one”

The Importance of SSA Form

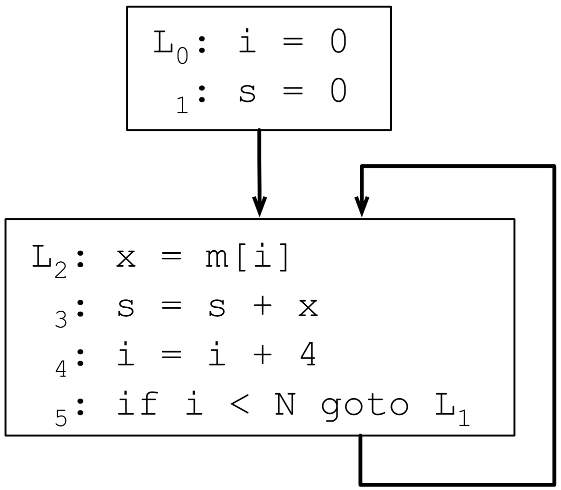
- The seminal paper that describes the SSA intermediate program representation has over 2100 citations♧.
- Almost every compiler text book talks about SSA form.
- Google Scholar returns over half a million results for the query "Static Single Assignment"

The SSA Seminar, in 2011, celebrated the 20th anniversary of the Static Single Assignment form (April 27-30, Autrans, France)

♧: Data collected on April of 2014, about the paper "Efficiently computing static single assignment form and the control dependence graph", published in 1991 by Cytron *et al*.

The Single Static Assignment Form

- This name comes out of the fact that each variable has only one definition site in the program.
- In other words, the entire program contains only one point where the variable is assigned a value.
- Were we talking about Single Dynamic Assignment, then we would be saying that during the <u>execution</u> of the program, the variable is assigned only once.



Variable i has two static assignment sites: at L_0 and at L_4; thus, this program is not in Static Single Assignment form. Variable s, also has two static definition sites. Variable x, on the other hand, has only one static definition site, at L_2. Nevertheless, x may be assigned many times dynamically, i.e., during the execution of the program.

TO AND FROM SSA FORM

DCC 888

Converting Straight-Line Code into SSA Form

- We call a program without branches a piece of "straight-line code".

```
double baskhara(double a, double b, double c) {
  double delta = b * b - 4 * a * c;
  double sqrDelta = sqrt(delta);
  double root = (b + sqrDelta) / 2 * a;
  return root;
}
```

1) Is the bitcode program in SSA form?

```
define double @baskhara(double %a, double %b, double %c) {
  %1 = fmul double %b, %b
  %2 = fmul double 4.000000e+00, %a
  %3 = fmul double %2, %c
  %4 = fsub double %1, %3
  %5 = call double @sqrt(double %4)
  %6 = fadd double %b, %5
  %7 = fdiv double %6, 2.000000e+00
  %8 = fmul double %7, %a
  ret double %8
}
```

2) How can we convert a straight-line program into SSA form?

```
$> clang -c -emit-llvm straight.c -o straight.bc
$> opt -mem2reg straight.bc -o straight.ssa.bc
$> llvm-dis straight.ssa.bc
```

Converting Straight-Line Code into SSA Form

- We call a program without branches a piece of "straight-line code".
- Converting a straight-line program, e.g., a basic block, into SSA is fairly straightforward.

```
L0: a = x + y
 1: b = a - 1
 2: a = y + b
 3: b = 4 * x
 4: a = a + b
```

Can you convert this program into SSA form?

for each variable a:
  Count[a] = 0
  Stack[a] = [0]
rename_basic_block(B) =
  for each instruction S in block B:
    for each use of a variable x in S:
      i = top(Stack[x])
      replace the use of x with x_i
    for each variable a that S defines
      count[a] = Count[a] + 1
      i = Count[a]
      push i onto Stack[a]
      replace definition of a with a_i

Converting Straight-Line Code into SSA Form

- We call a program without branches a piece of "straight-line code".
- Converting a straight-line program, e.g., a basic block, into SSA is fairly straightforward.

$L_0: a_1 = x_0 + y_0$
$_1: b_1 = a_1 - 1$
$_2: a_2 = y_0 + b_1$
$_3: b_2 = 4 * x_0$
$_4: a_3 = a_2 + b_2$

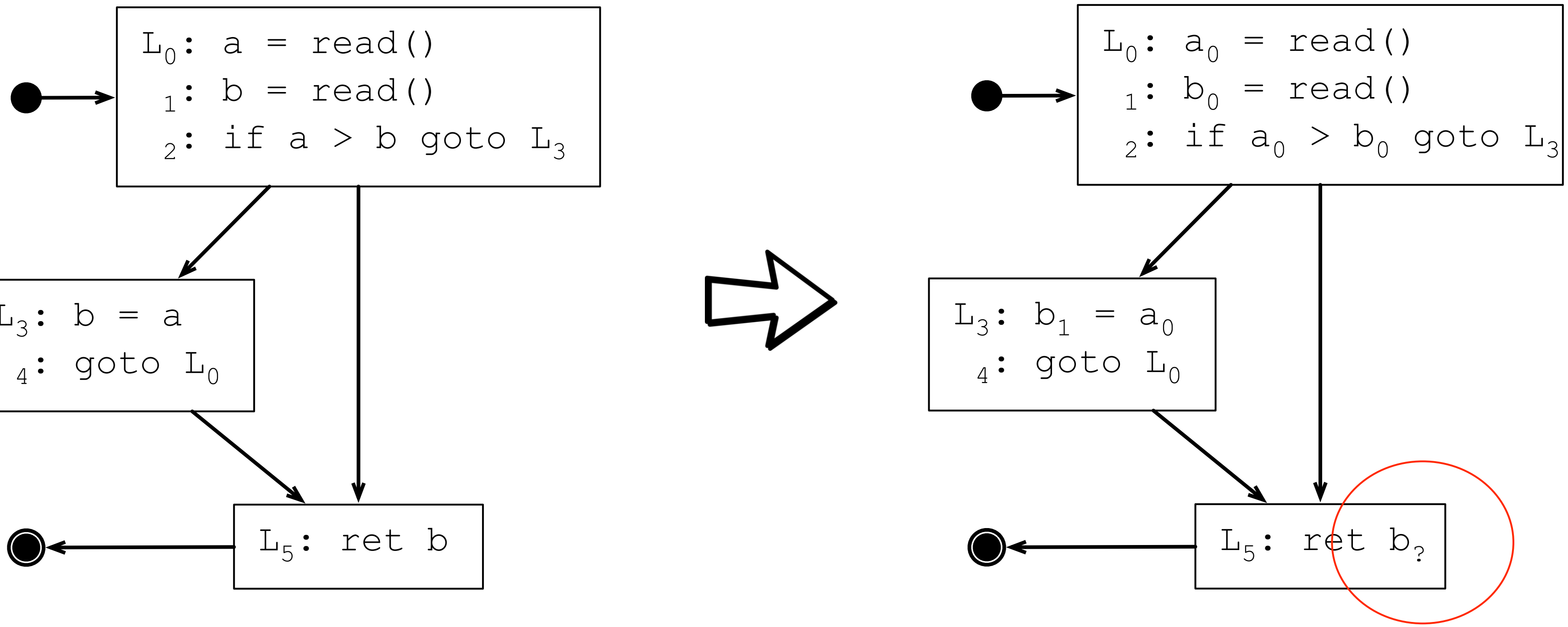
for each variable a:
  Count[a] = 0
  Stack[a] = [0]
rename_basic_block(B) =
  for each instruction S in block B:
    for each use of a variable x in S:
      i = top(Stack[x])
      replace the use of x with x_i
    for each variable a that S defines
      count[a] = Count[a] + 1
      i = Count[a]
      push i onto Stack[a]
      replace definition of a with a_i

Notice that we could do without the **stack**. How? But we will need it to generalize this method.

Phi-Functions

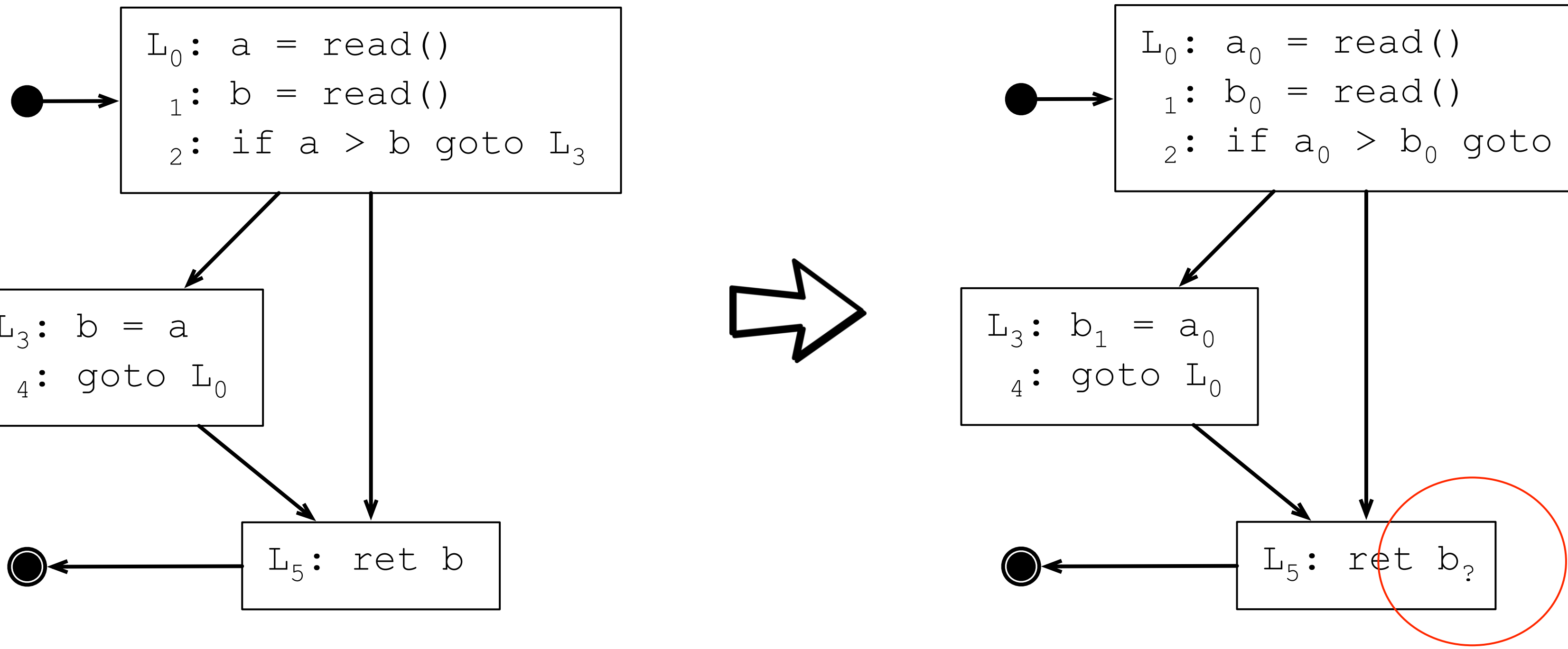
Having just one static assignment site for each variable brings some challenges, once we stop talking about straight-line programs, and start dealing with more complex flow graphs.

One important question is: once we convert this program to SSA form, which definition of b should we use at L_5?



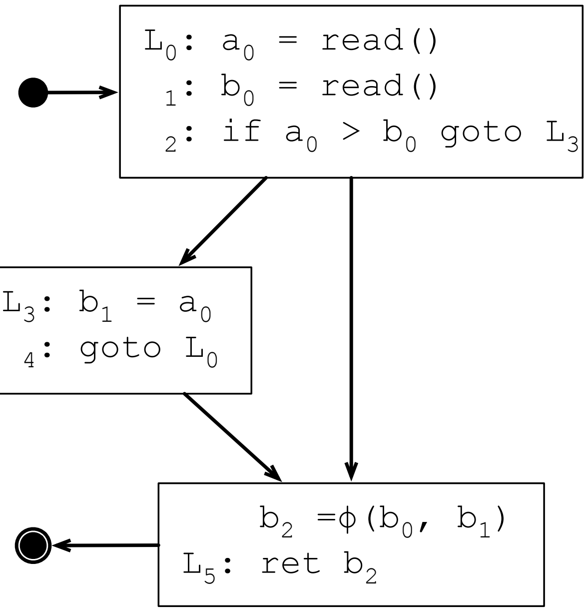
Phi-Functions

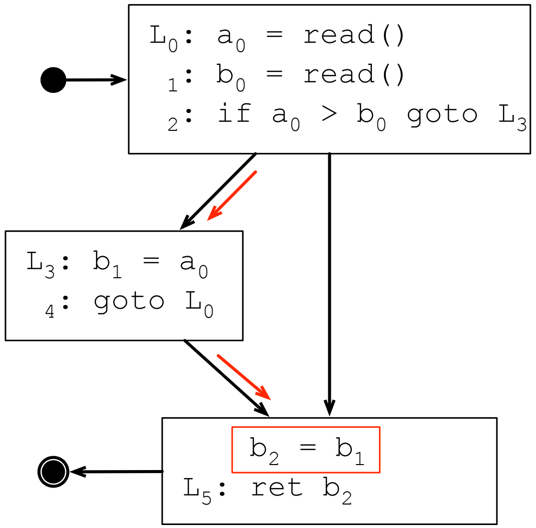
The answer to this question is: *it depends*! Indeed, the definition of b that we will use at L_5 will depend on which path execution flows. If the execution flow reaches L_5 coming from L_4, then we must use b_1. Otherwise, execution must reach L_5 coming from L_2, in which case we must use b_0

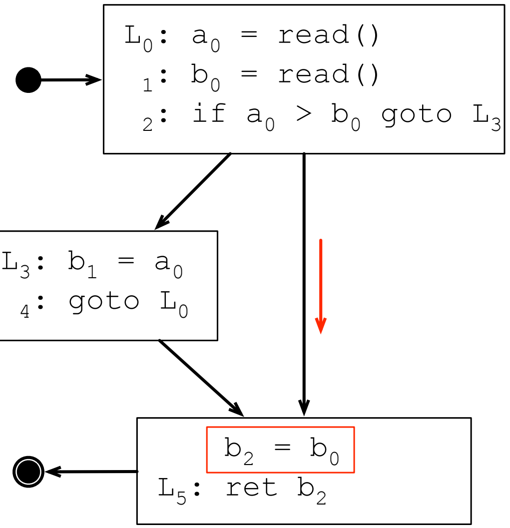


Phi-Functions

In order to represent this kind of behavior, we use a special notation: the phi-function. Phi-functions have the semantics of a multiplexer, copying the correct definition, depending on which path they are reached by the execution flow.



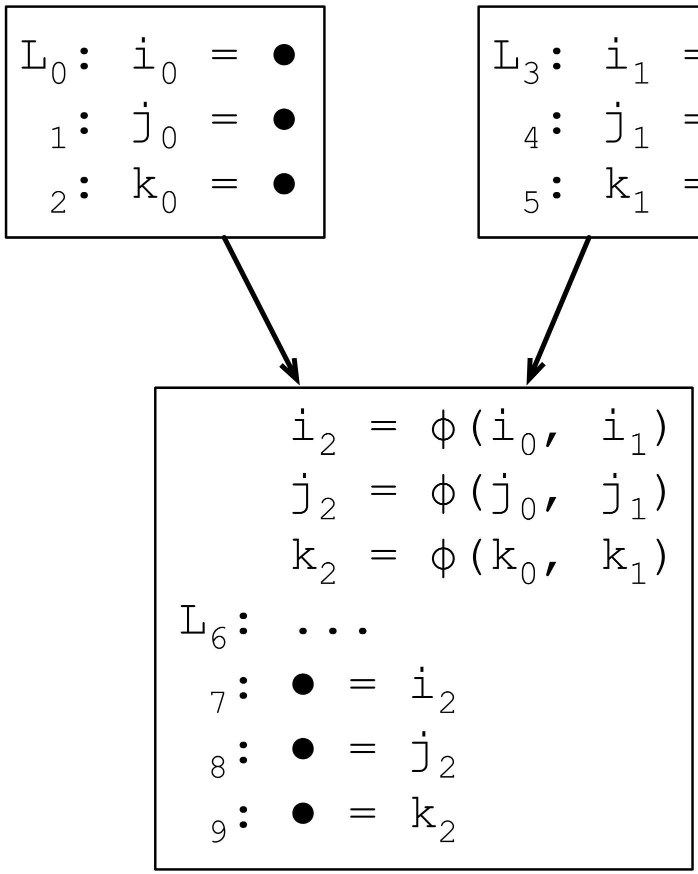




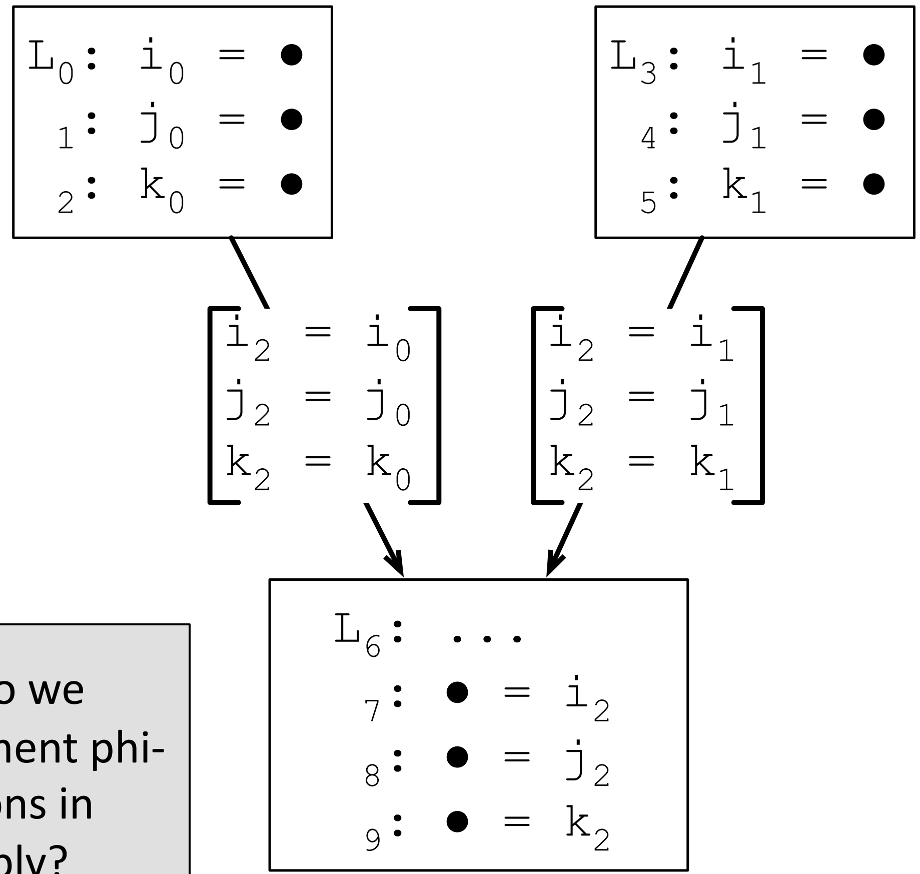
What happens once we have multiple phi-functions at the beginning of a block?

Phi-Functions

A set of N phi-functions with M arguments each at the beginning of a basic block represent M parallel copies. Each copy reads N inputs, and writes on N outputs.



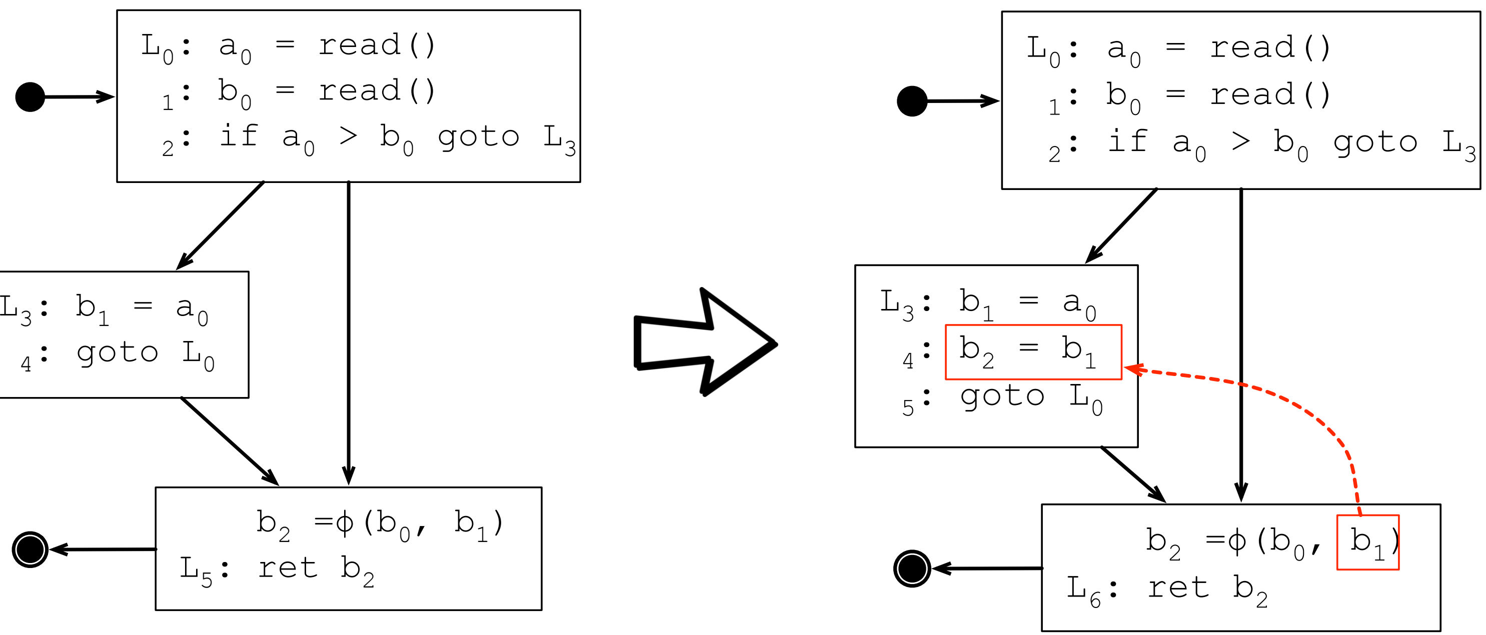
How do we implement phi-functions in assembly?



SSA Elimination

Compilers that use the SSA form usually contain a step, before the generation of actual assembly code, in which phi-functions are replaced by ordinary instructions. Normally these instructions are simple copies.

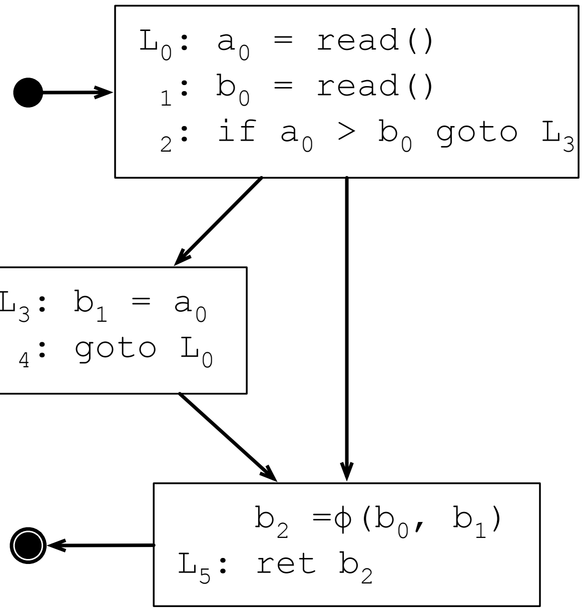
And where would we place the copy $b_2 = b_0$? Why is this an important question at all?

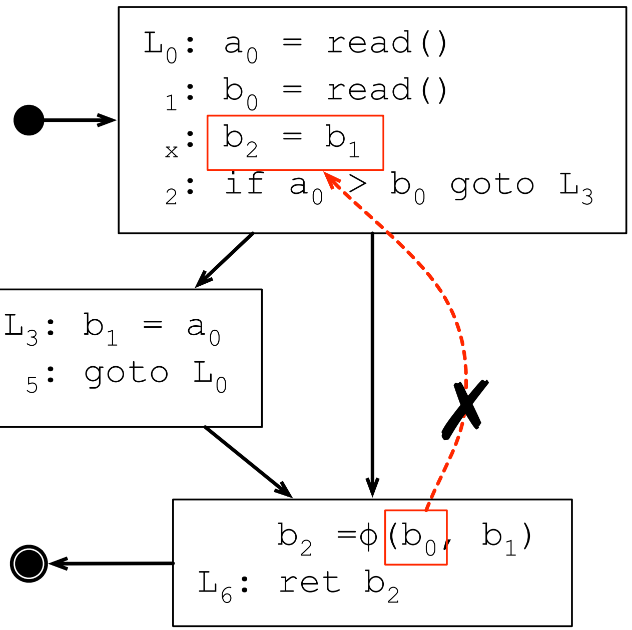


Critical Edges

The placement of the copy $b_2 = b_0$ is not simple, because the edge that links L_2 to L_5 is *critical*. A critical edge connects a block with multiple successors to a block with multiple predecessors.

If we were to put the copy between labels L1 and L2, then we would be creating a partial redundancy.





1) Where have we heard of critical edges before?

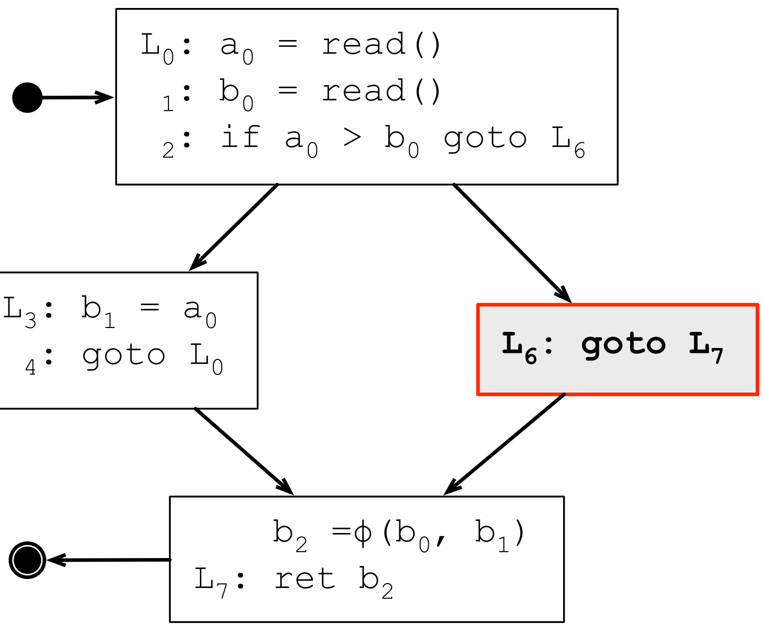
2) How can we solve this conundrum?

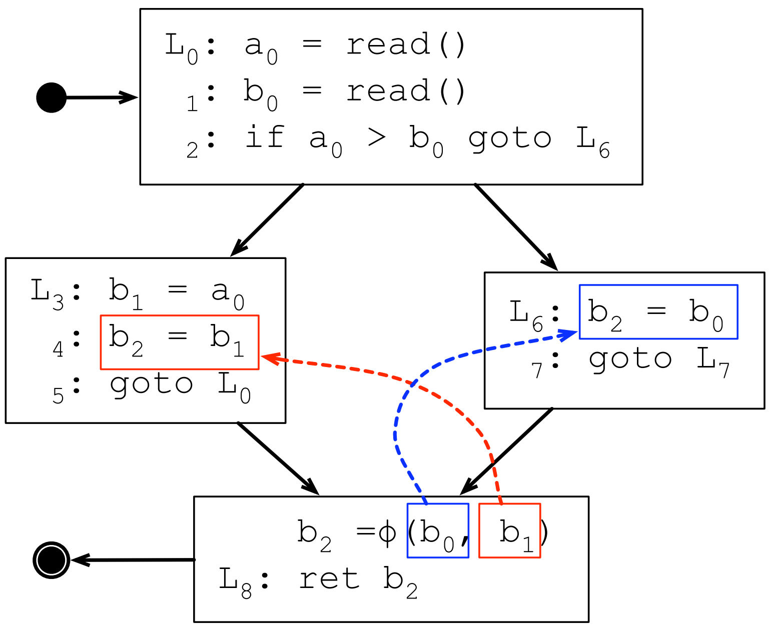
Anyway: where to insert phi-functions?

Edge Splitting

We can solve this problem by doing *critical edge splitting*. This CFG transformation consists in adding an empty basic block (empty, except by – perhaps – a goto statement) between each pair of blocks connected by a critical edge.

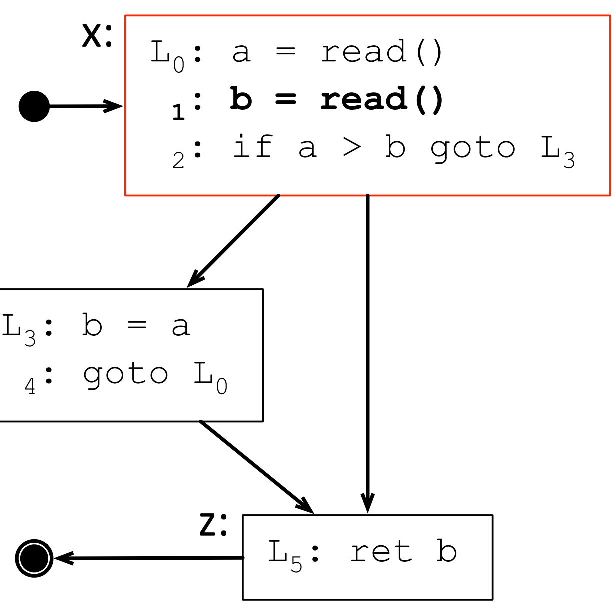
Notice that the CFGs that LLVM produces do not have critical edges.





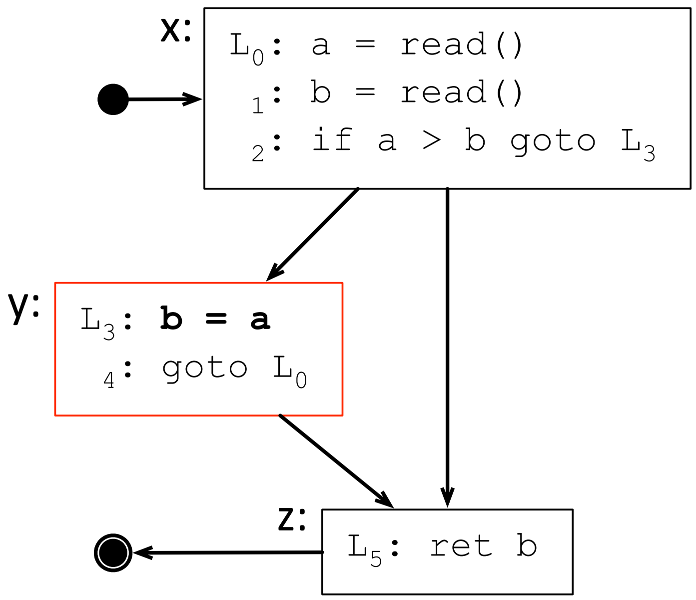
Criteria for Inserting Phi-Functions

- There should be a phi-function for variable ‘a’ at node ‘z’ of the flow graph exactly when all of the following are true:
  - There is a block x containing a definition of a
  - There is a block y (with y ≠ x) containing a definition of a
  - There is a nonempty path P_{xz} of edges from x to z
  - There is a nonempty path P_{yz} of edges from y to z
  - Paths P_{xz} and P_{yz} do not have any node in common other than z, and…
  - The node z does not appear within both P_{xz} and P_{yz} prior to the end, though it may appear in one or the other.



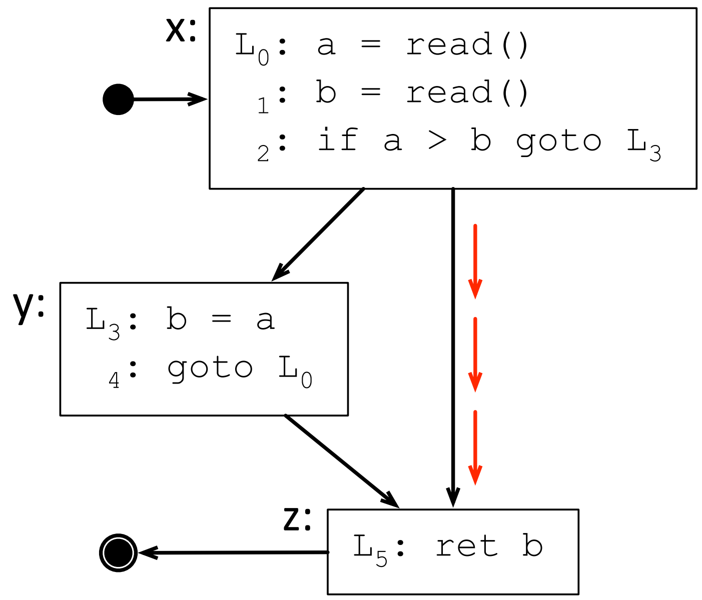
Criteria for Inserting Phi-Functions

- There should be a phi-function for variable a at node z of the flow graph exactly when all of the following are true:
  – There is a block x containing a definition of a
  – There is a block y (with y ≠ x) containing a definition of a
  – There is a nonempty path P_{xz} of edges from x to z
  – There is a nonempty path P_{yz} of edges from y to z
  – Paths P_{xz} and P_{yz} do not have any node in common other than z, and…
  – The node z does not appear within both P_{xz} and P_{yz} prior to the end, though it may appear in one or the other.



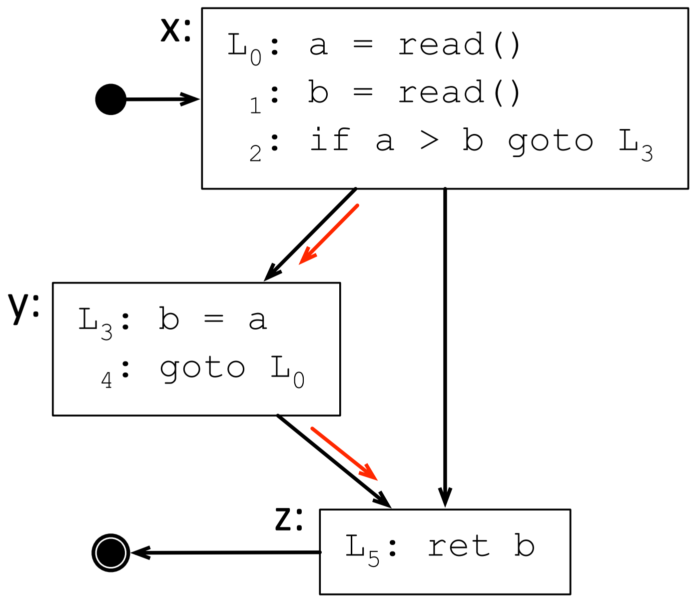
Criteria for Inserting Phi-Functions

- There should be a phi-function for variable a at node z of the flow graph exactly when all of the following are true:
  - There is a block x containing a definition of a
  - There is a block y (with y ≠ x) containing a definition of a
  - There is a nonempty path P_{xz} of edges from x to z
  - There is a nonempty path P_{yz} of edges from y to z
  - Paths P_{xz} and P_{yz} do not have any node in common other than z, and…
  - The node z does not appear within both P_{xz} and P_{yz} prior to the end, though it may appear in one or the other.



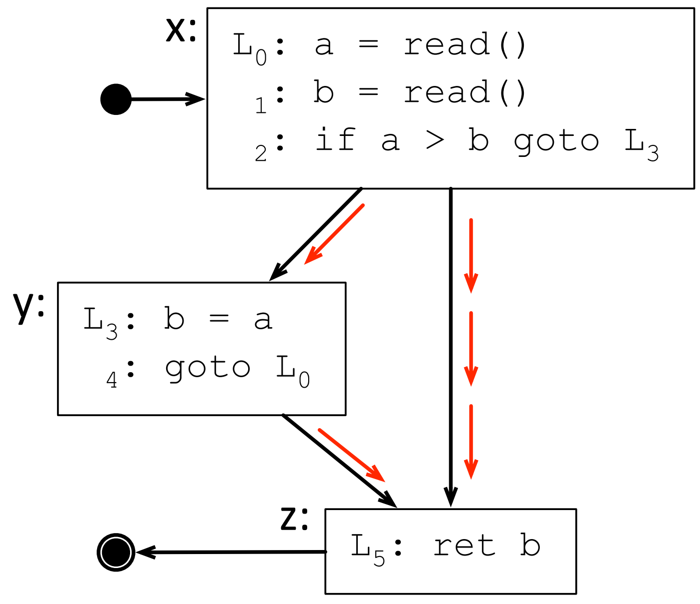
Criteria for Inserting Phi-Functions

- There should be a phi-function for variable a at node z of the flow graph exactly when all of the following are true:
  - There is a block x containing a definition of a
  - There is a block y (with y ≠ x) containing a definition of a
  - There is a nonempty path P_{xz} of edges from x to z
  - There is a nonempty path P_{yz} of edges from y to z
  - Paths P_{xz} and P_{yz} do not have any node in common other than z, and…
  - The node z does not appear within both P_{xz} and P_{yz} prior to the end, though it may appear in one or the other.



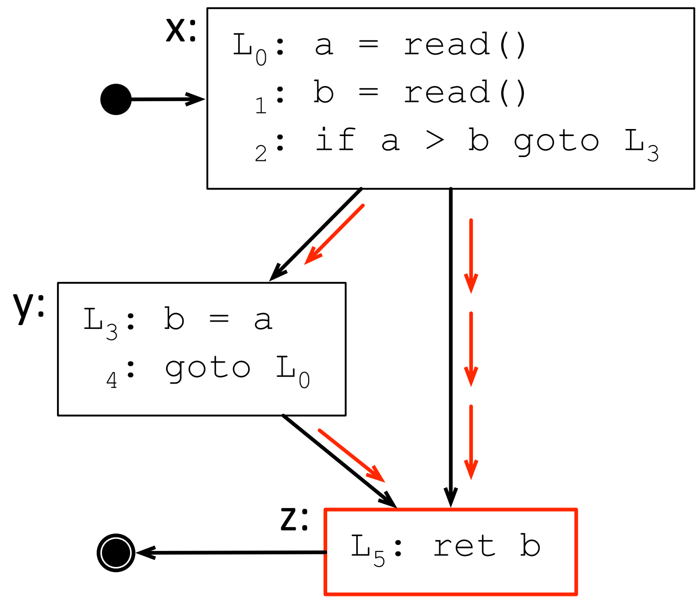
Criteria for Inserting Phi-Functions

- There should be a phi-function for variable a at node z of the flow graph exactly when all of the following are true:
  - There is a block x containing a definition of a
  - There is a block y (with y ≠ x) containing a definition of a
  - There is a nonempty path P_{xz} of edges from x to z
  - There is a nonempty path P_{yz} of edges from y to z
  - Paths P_{xz} and P_{yz} do not have any node in common other than z, and…
  - The node z does not appear within both P_{xz} and P_{yz} prior to the end, though it may appear in one or the other.



Criteria for Inserting Phi-Functions

- There should be a phi-function for variable a at node z of the flow graph exactly when all of the following are true:
  - There is a block x containing a definition of a
  - There is a block y (with y ≠ x) containing a definition of a
  - There is a nonempty path P_{xz} of edges from x to z
  - There is a nonempty path P_{yz} of edges from y to z
  - Paths P_{xz} and P_{yz} do not have any node in common other than z, and…
  - The node z does not appear within both P_{xz} and P_{yz} prior to the end, though it may appear in one or the other.



Iterative Creation of Phi-Functions

- When we insert a new phi-function in the program, we are creating a new definition of a variable.
- This new definition may raise the necessity of new phi-functions in the code.
- Thus, the path convergence criteria must be used iteratively, until we reach a fixed point:

What is the complexity of this algorithm?

while there are nodes *x*, *y*, and *z* satisfying the path-convergence criteria and *z* does not contain a phi-function for variable *a* **do**:

insert $a = \phi(a, a, \ldots, a)$ at node *z*, with as many parameters as *z* has predecessors.

Dominance Property of SSA Form

The previous algorithm is a bit too expensive. Let's see a faster one. But, to this end, we will need the goal of dominance frontier.

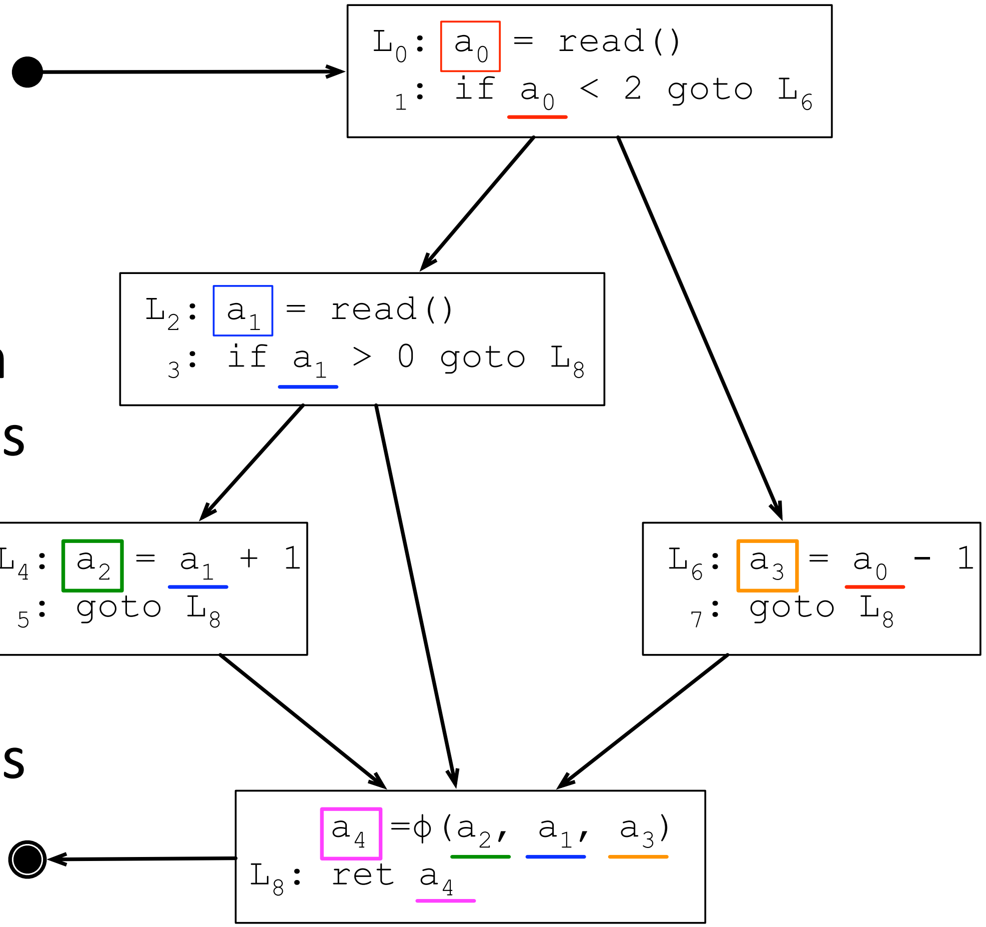
- A node *d* of a rooted, directed graph dominates another node *n* if every path from the root node to *n* goes through *d*.
- In Strict♤ SSA form programs, definitions of variables dominate their uses:
  – If x is the i-th argument of a phi-function in block n, then the definition of x dominates the i-th predecessor of n.
  – If x is used in a non-phi statement in block n, then the definition of x dominates node n.

Where have we heard of dominance before?

♤: A program is strict if every variable is initialized before it is used.

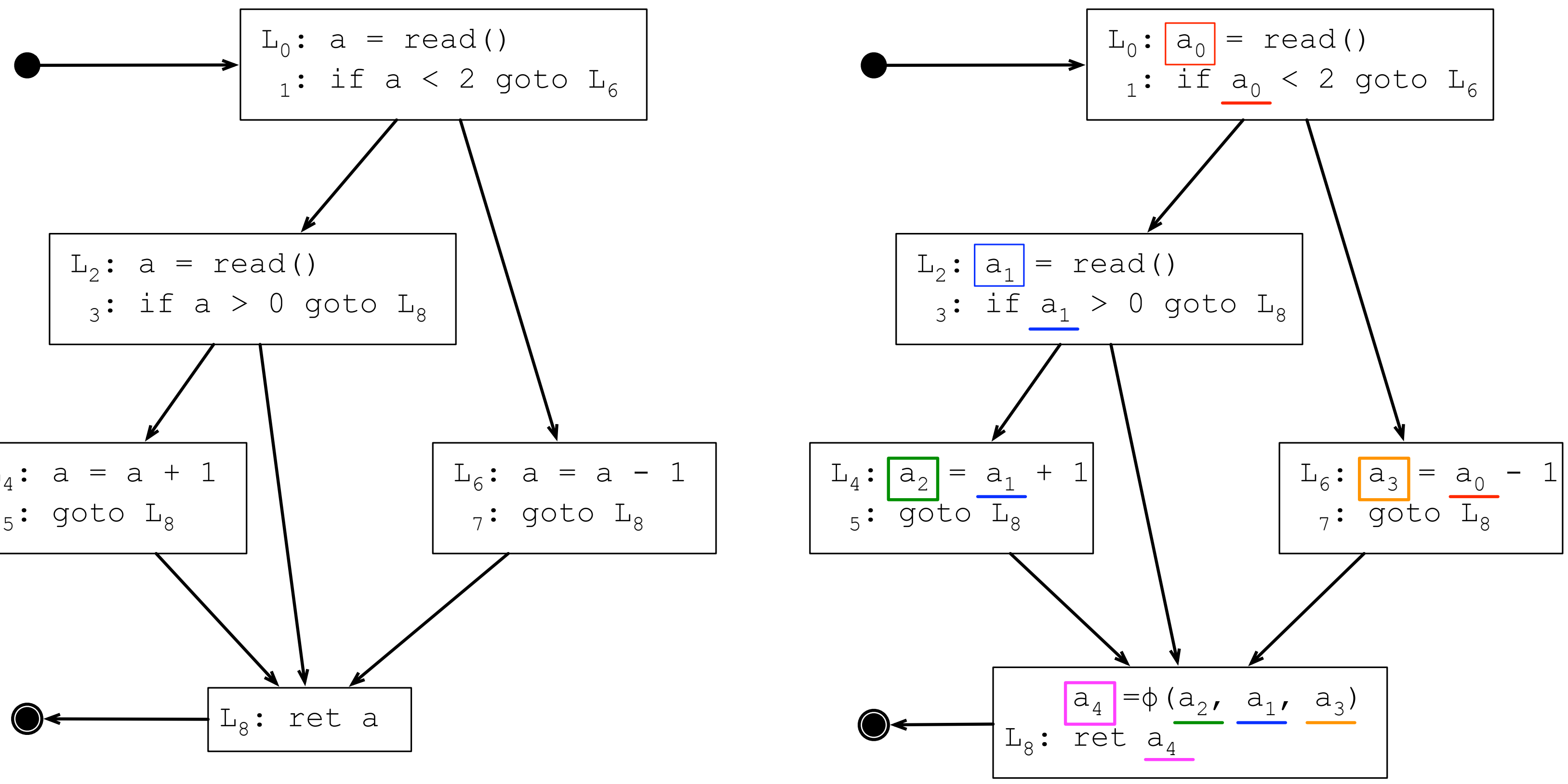
Dominance Property of SSA Form

- In Strict SSA form programs, definitions of variables dominate their uses:
  - If x is the i-th argument of a phi-function in block n, then the definition of x dominates the i-th predecessor of n.
  - If x is used in a non-phi statement in block n, then the definition of x dominates node n.



How does this observation helps us to build SSA form?

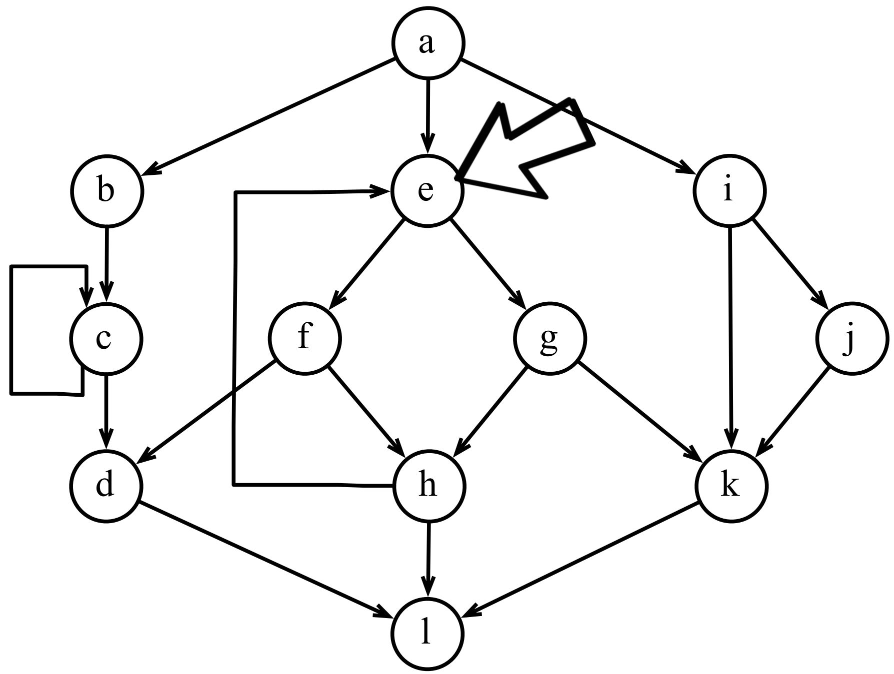
Dominance Property of SSA Form



For one, we can distribute phi-functions here and there, and then we only have to worry about one thing: we must ensure that every use of a variable v has the same name as the instance of v that dominates that use.

The Dominance Frontier

- There is an algorithm more efficient than the iterative application of the path-convergence criteria, which is almost linear time on the size of the program.
  - This algorithm relies on the notion of dominance frontier
- A node x strictly dominates w if x dominates w and x ≠w.
- The dominance frontier of a node x is the set of all nodes w such that x dominates a predecessor of w, but does not strictly dominate w.

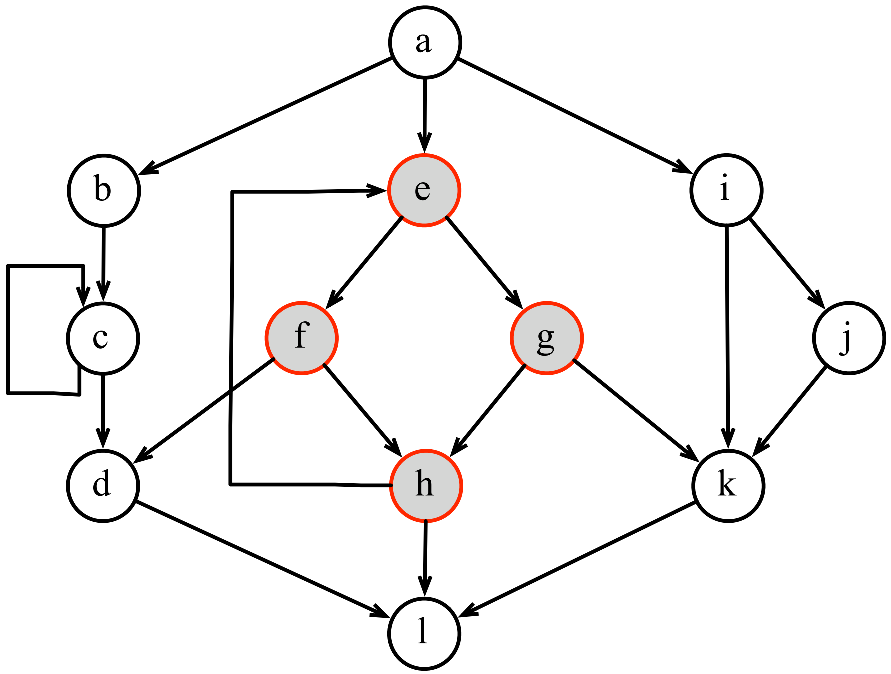


What are the nodes that e dominates?

The Dominance Frontier

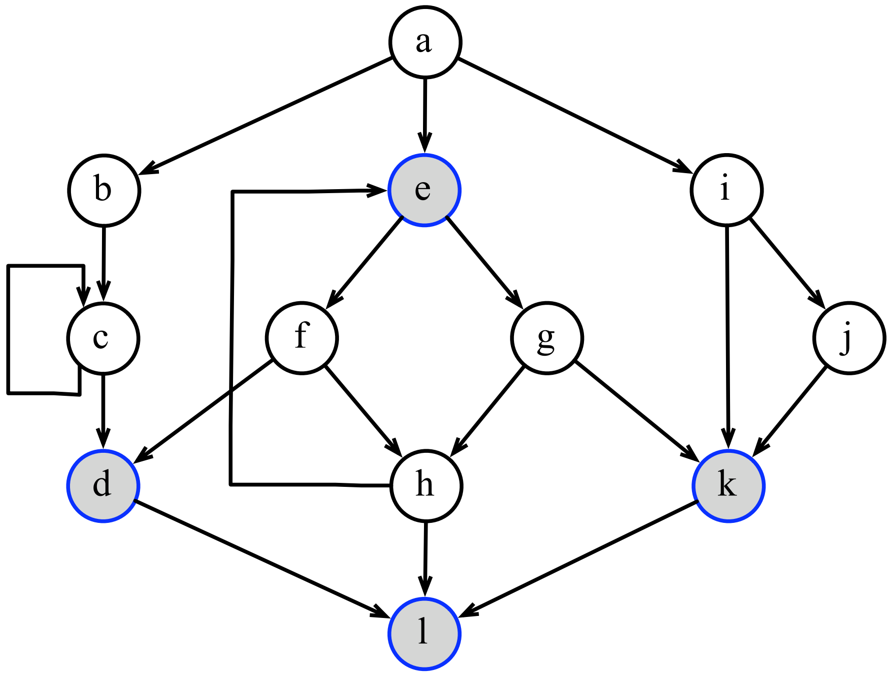
- There is an algorithm more efficient than the iterative application of the path-convergence criteria, which is almost linear time on the size of the program.
  - This algorithm relies on the notion of dominance frontier
- A node x strictly dominates w if x dominates w and x ≠w.
- The dominance frontier of a node x is the set of all nodes w such that x dominates a predecessor of w, but does not strictly dominate w.

What are the nodes in the dominance frontier of e?



The Dominance Frontier

- There is an algorithm more efficient than the iterative application of the path-convergence criteria, which is almost linear time on the size of the program.
  - This algorithm relies on the notion of dominance frontier
- A node x strictly dominates w if x dominates w and x ≠w.
- The dominance frontier of a node x is the set of all nodes w such that x dominates a predecessor of w, but does not strictly dominate w.



Why is e included in its dominance frontier?

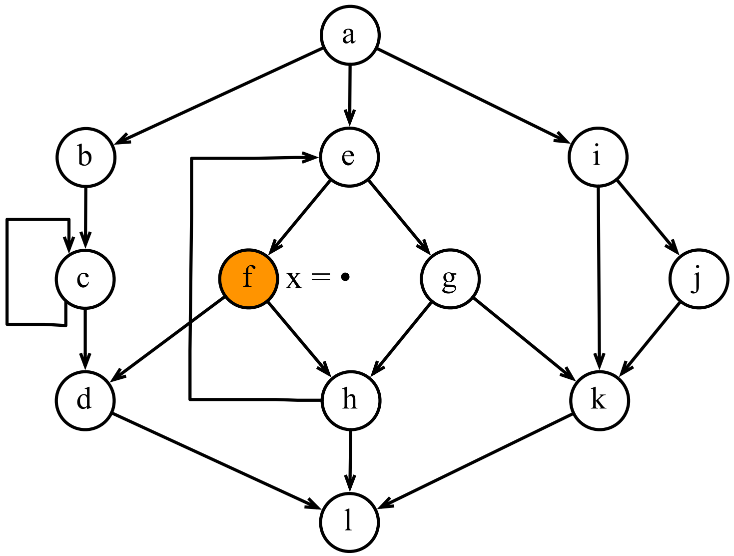
The Dominance Frontier Criterion

- **Dominance-Frontier Criterion**: Whenever node x contains a definition of some variable a, then any node z in the dominance frontier of x needs a phi-function for a.

- **Iterated dominance frontier**: since a phi-function itself is a kind of definition, we must iterate the dominance-frontier criterion until there are no nodes that need phi-functions.

Theorem: the iterated dominance frontier criterion and the iterated path-convergence criteria specify exactly the same set of nodes at which to put phi-functions.

The Dominance Frontier Criterion

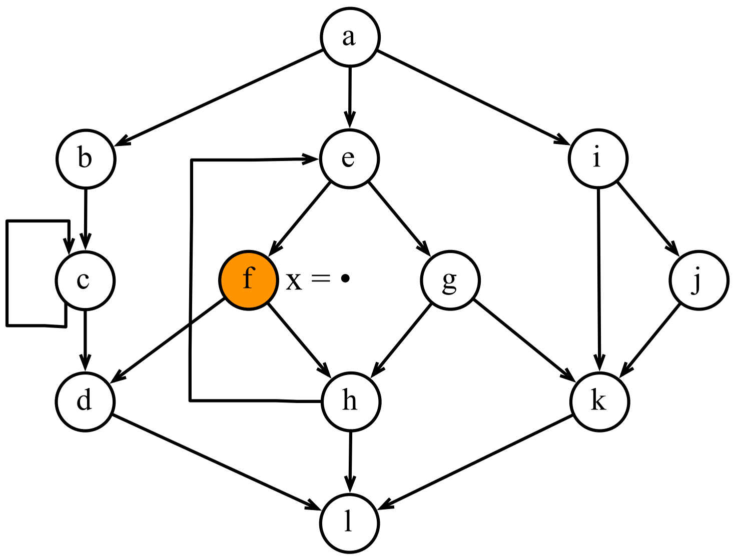


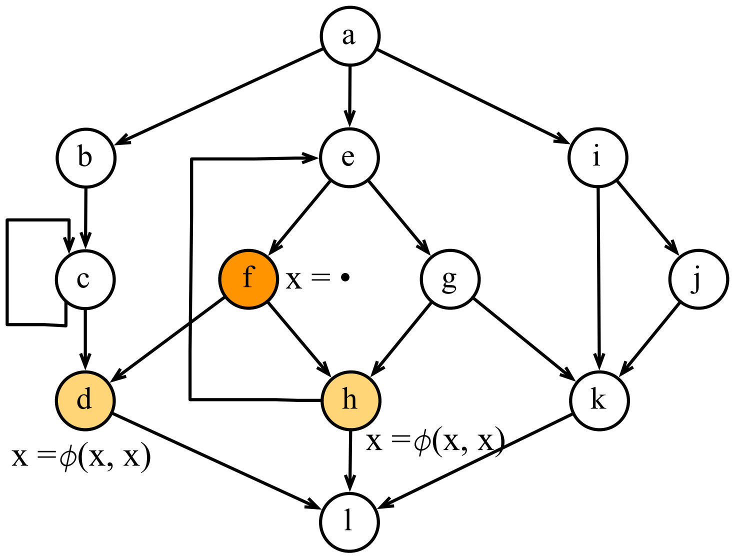
Where should we place phi-functions due to the definition of x at block f?

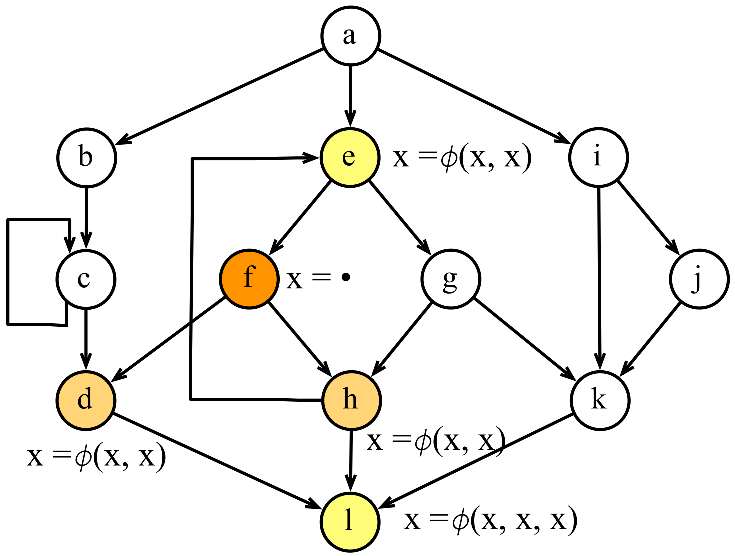
- **Dominance-Frontier Criterion**: Whenever node x contains a definition of some variable a, then any node z in the dominance frontier of x needs a phi-function for a.

- **Iterated dominance frontier**: since a phi-function itself is a kind of definition, we must iterate the dominance-frontier criterion until there are no nodes that need phi-functions.

The Dominance Frontier Criterion

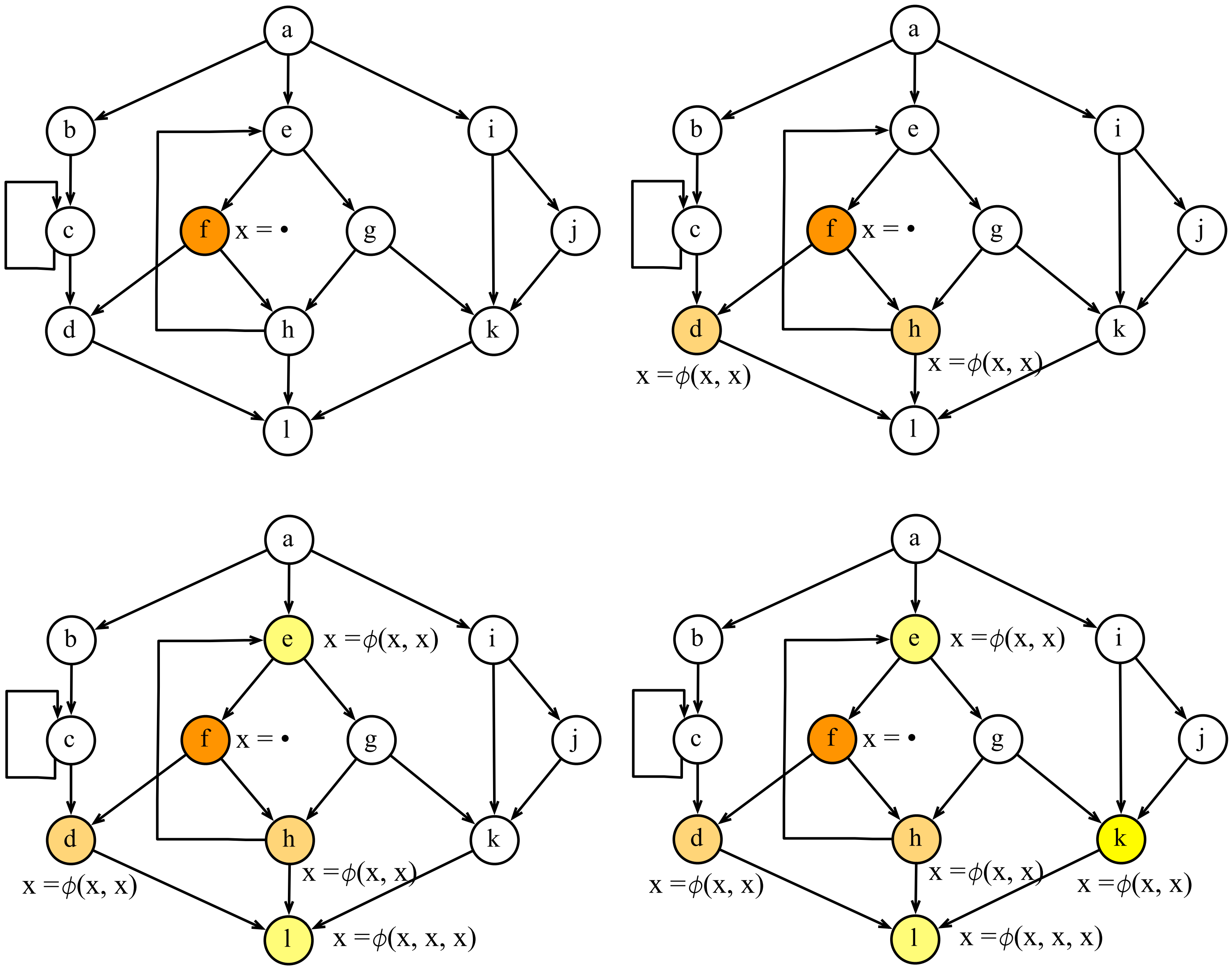






Is there any other place that should receive phi-functions?

The Dominance Frontier Criterion



Computing the Dominance Frontier

We compute the dominance frontier of the nodes of a graph by iterating the following equations:

$$DF[n] = DF_{local}[n] \cup \{ DF_{up}[c] \mid c \in children[n] \}$$

Where:

- $DF_{local}[n]$: the successors of n that are not strictly dominated by n
- $DF_{up}[c]$: nodes in the dominance frontier of c that are not strictly dominated by n.
- children[n]: the set of children of node n in the dominator tree

1) It should be clear why we need $DF_{local}[n]$, right?

2) But, why do we have **this** second part of the equation?

Computing the Dominance Frontier

We compute the dominance frontier of the nodes of a graph by iterating the following equations:

$DF[n] = DF_{local}[n] \cup \{ DF_{up}[c] \mid c \in children[n] \}$

Where:

- $DF_{local}[n]$: the successors of n that are not strictly dominated by n
- $DF_{up}[c]$: nodes in the dominance frontier of c that are not strictly dominated by n.
- children[c]: the set of children of node c in the dominator tree

The algorithm below computes the dominance frontier of every node in the CFG. It must be called from the root node:

computeDF[n]:

S = {}

for each node y in succ[n]

    if idom(y) ≠ n

        S = S ∪ {y}

for each child c of n in the dom-tree

    computeDF[c]

    for each w ∈ DF[c]

        if n does not dom w, or n = w

            S = S ∪ {w}

DF[n] = S

Inserting Phi-Functions

place-phi-functions:

for each node n:

  for each variable a $\in A_{orig}[n]$:

    defsites[a] = defsites[a] $\cup$ [n]

for each variable a:

  W = defsites[a]

  while W $\neq$ empty list

    remove some node n from W

    for each y in DF[n]:

      if a $\notin A_{phi}[y]$

        insert-phi(y, a)

        $A_{phi}[y] = A_{phi}[y] \cup \{a\}$

        if a $\notin A_{orig}[y]$

          $W = W \cup \{y\}$

insert-phi(y, a):

insert the statement a = ϕ(a, a, …, a) at the top of block y, where the phi-function has as many arguments as y has predecessors

Where:

- $A_{orig}[n]$: the set of variables defined at node n
- $A_{phi}[a]$: the set of nodes that must have phi-functions for variable a

Notice that W can grow, due to **this** union. How do we know that this algorithm terminates?

Renaming Variables

- We already have a procedure that renames variables in straight-line segments of code
- We must now extend this procedure to handle general control flow graphs.

How should we extend this algorithm to handle general CFGs?

for each variable a:

  Count[a] = 0

  Stack[a] = [0]

rename-basic-block(*B*):

  for each instruction *S* in block *B*:

    for each use of a variable *x* in *S*:

      i = top(Stack[x])

      replace the use of x with x_i

    for each variable a that *S* defines

      count[a] = Count[a] + 1

      i = Count[a]

      push i onto Stack[a]

      replace definition of a with a_i

Renaming Variables

Does this algorithm ensure that the definition of a variable dominates all its uses?

Child is the successor of n in the dominator tree. Why we cannot use the successors of n in the CFG?

rename(n):

rename-basic-block(n)

for each successor Y of n, **where** n is the j-th predecessor of Y:

  for each phi-function f in Y, **where** the operand of f is 'a'

    i = top(Stack[a])

    replace j-th operand with a_i

for each child X of n:

  rename(X)

for each instruction S ∈ n:

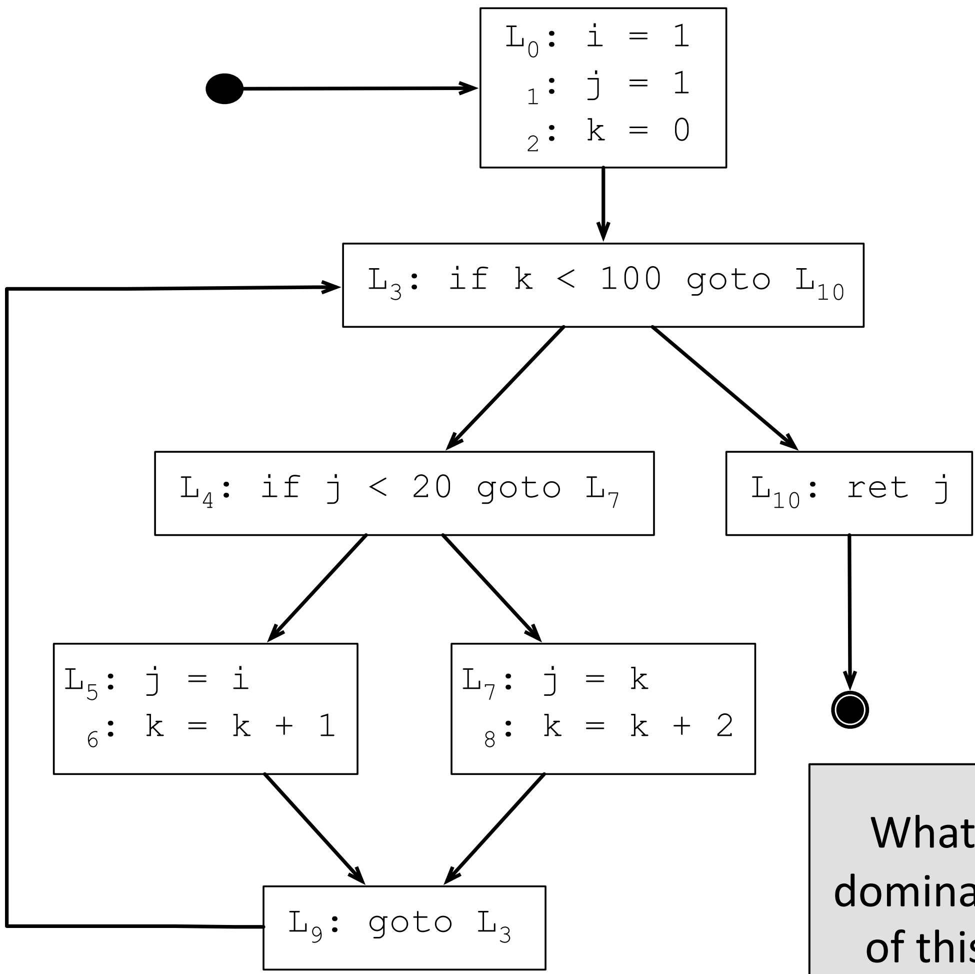
  for each variable that S defines:

    pop Stack[a]

Putting it All Together

- Lets convert the following program to SSA form:

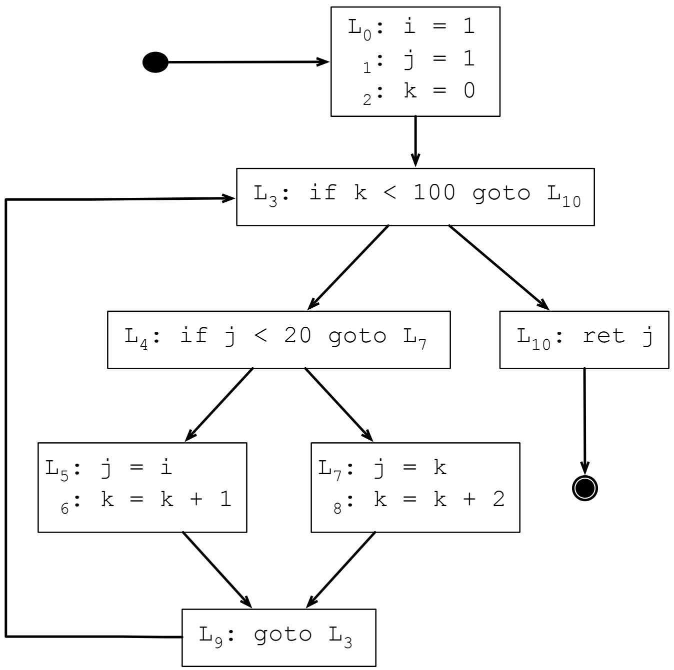
```
i = 1
j = 1
k = 0
while k < 100
    if j < 20
        j = i
        k = k + 1
    else
        j = k
        k = k + 2
return j
```



What is the dominator tree of this CFG?

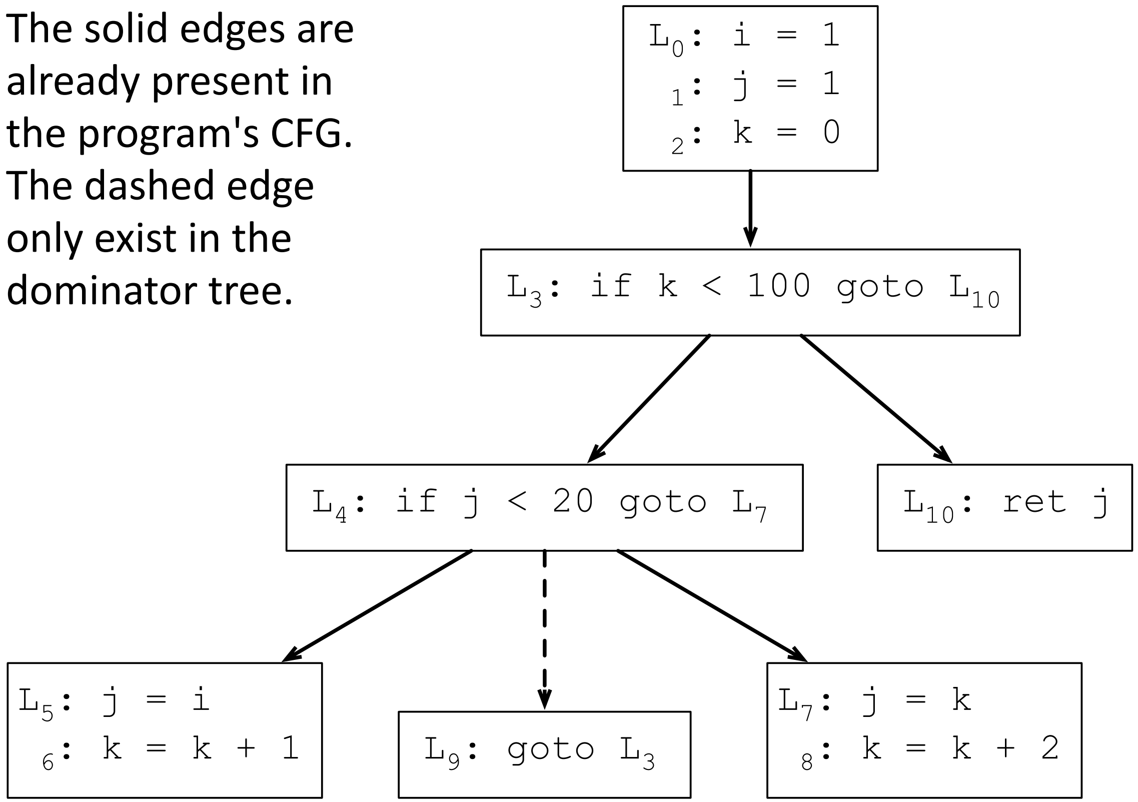
Putting it All Together

Can you compute the dominance frontier of each node?

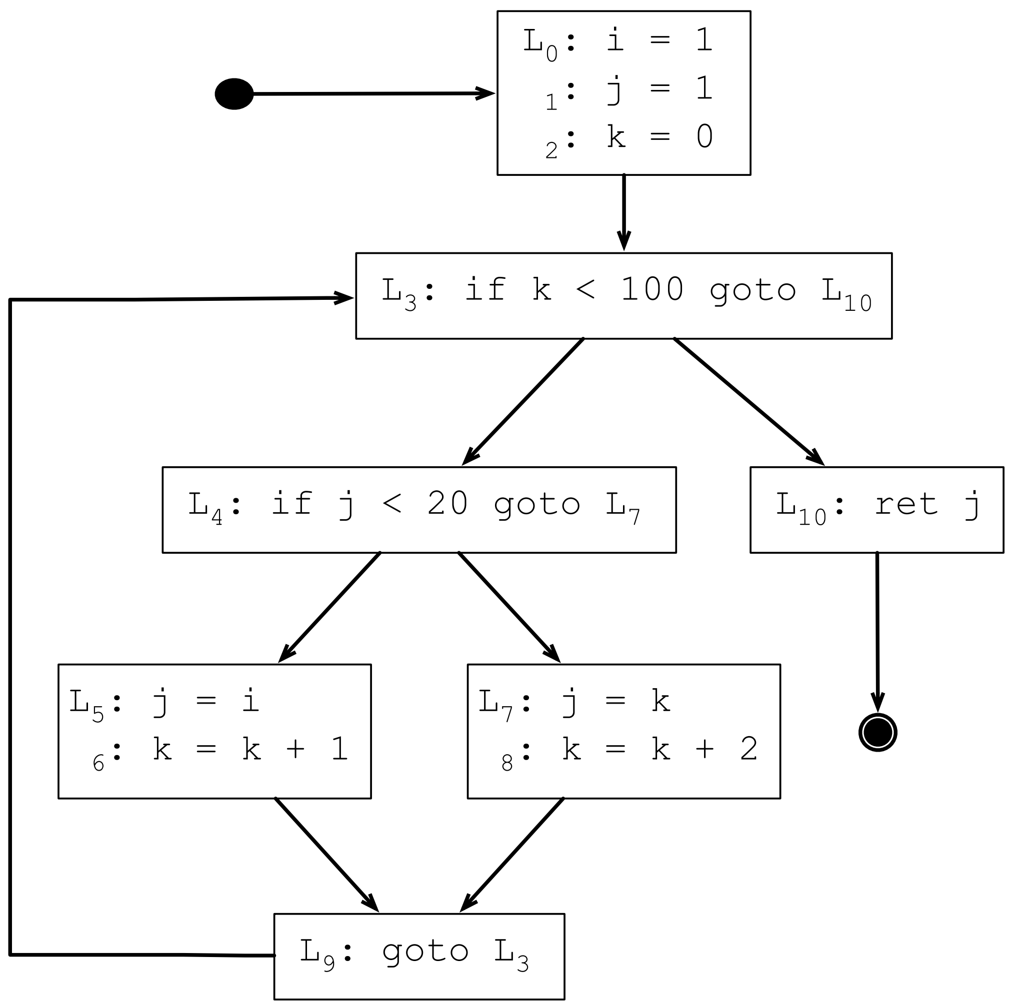


The Dominator Tree:

The solid edges are already present in the program's CFG. The dashed edge only exist in the dominator tree.



Computing the Dominance Frontier



The dominance frontier of each node is listed below:

L_0: {}

L_3: {L_3}

L_4: {L_3}

L_5: {L_9}

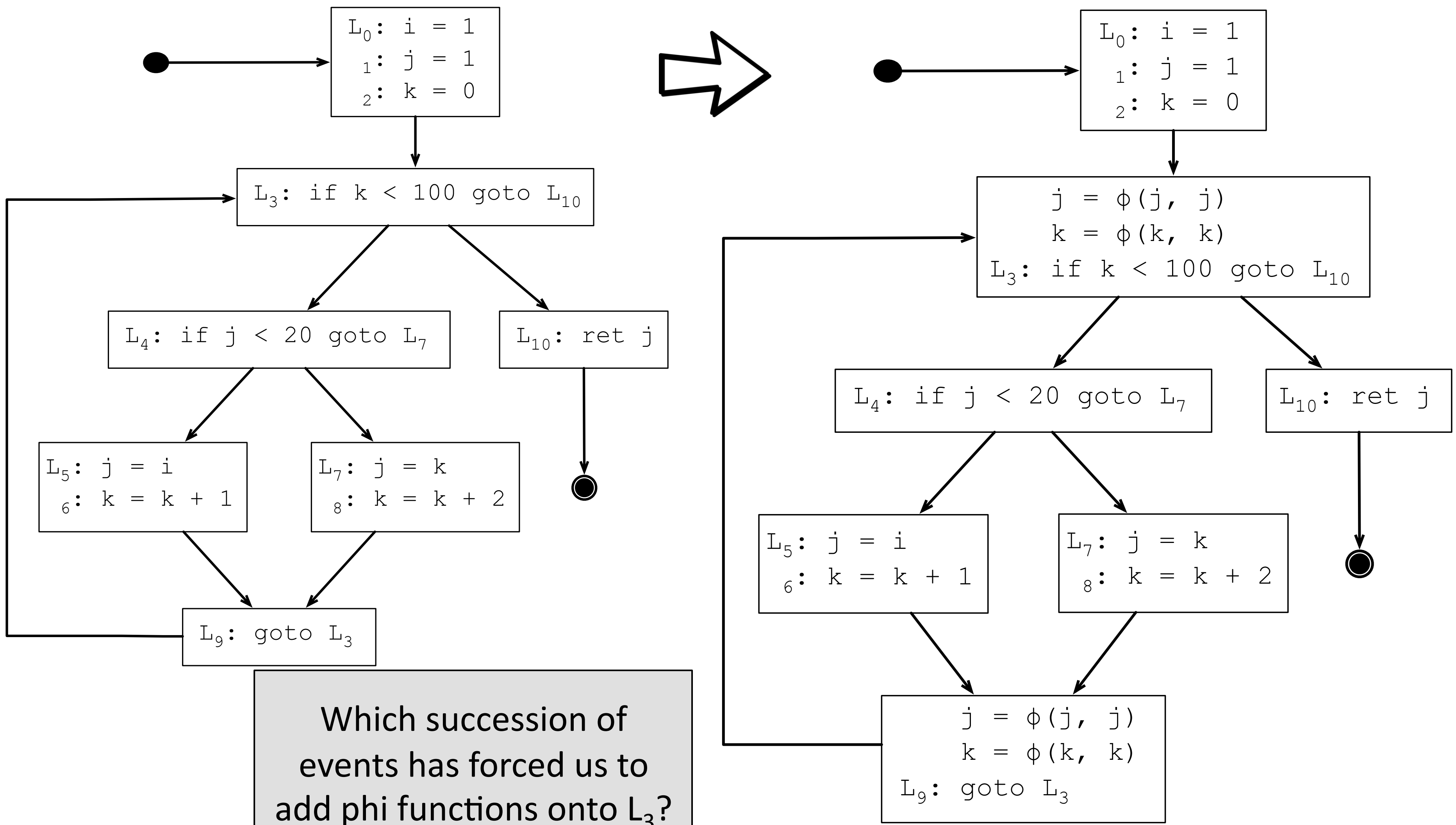
L_7: {L_9}

L_9: {L_3}

L_{10}: {}

Can you insert phi-functions in the CFG on the left, given these dominance frontiers?

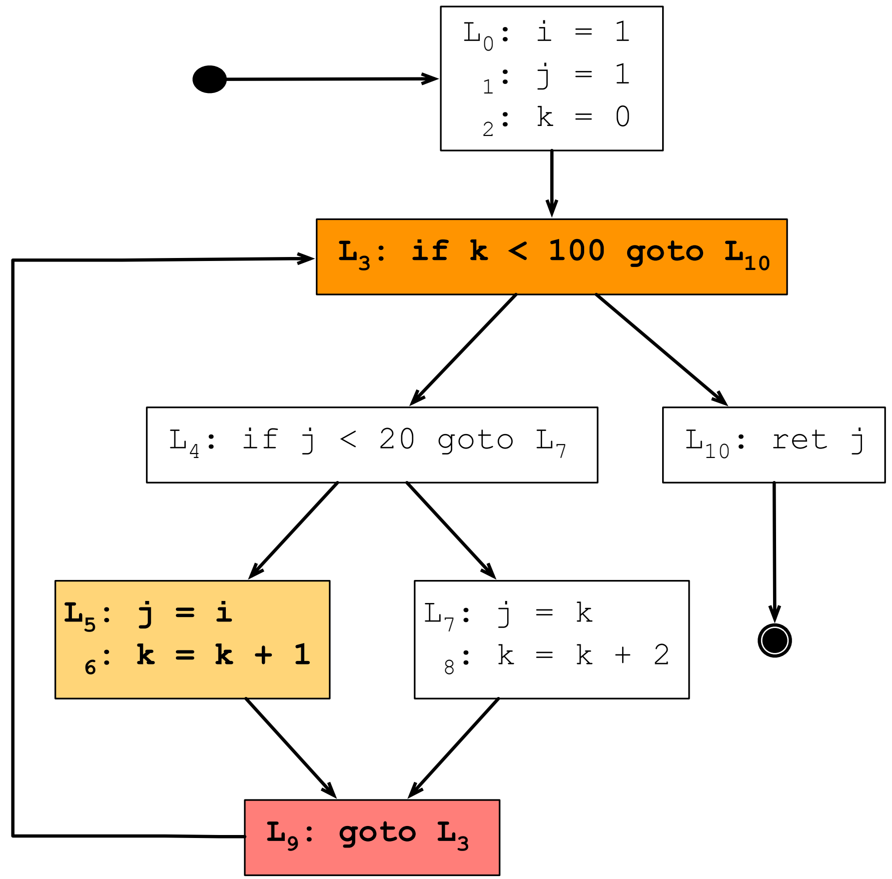
Inserting Phi-Functions



Which succession of events has forced us to add phi functions onto L_3?

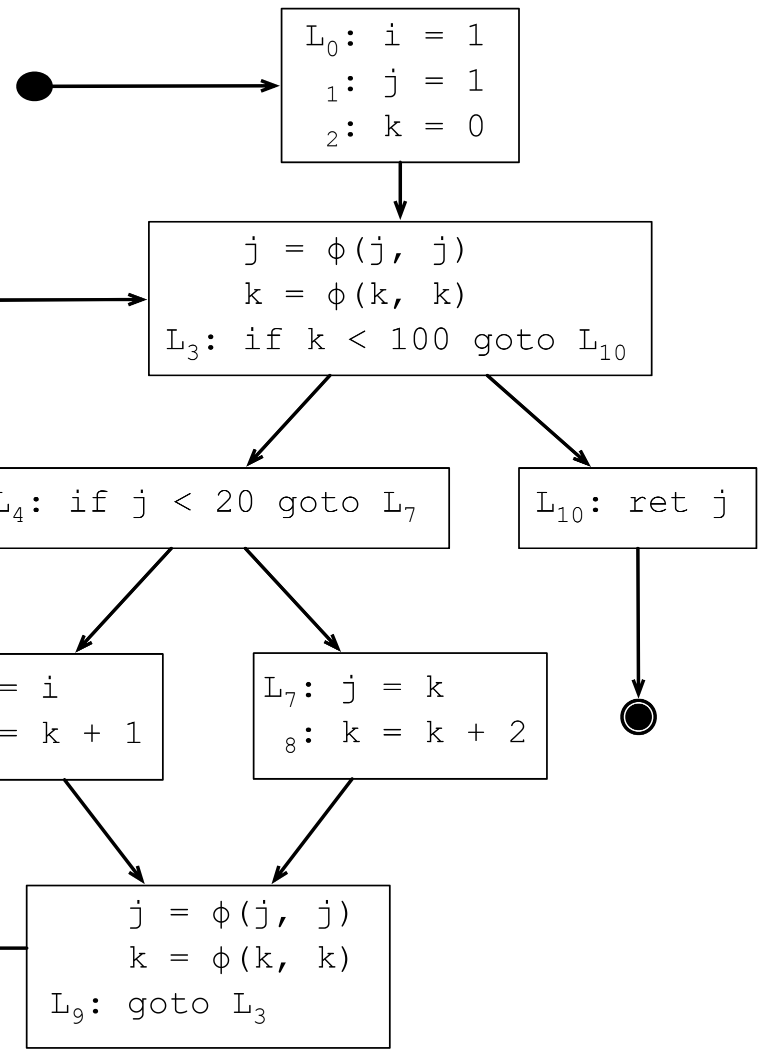
Iterated Dominance Frontier

- Node L_5 does not dominate L_9, although L_9 is a successor of L_5. Therefore, L_9 is in the dominance frontier of L_5. L_9 should have a phi-function for every variable defined inside L_5.
- We repeat the process for L_9, after all, we are considering the iterated dominance frontier.
- L_3 is in the dominance frontier of L_9, and should also have a phi-function for every variable defined in L_5. Notice that these variables are now redefined at L_9, due to the phi-functions.

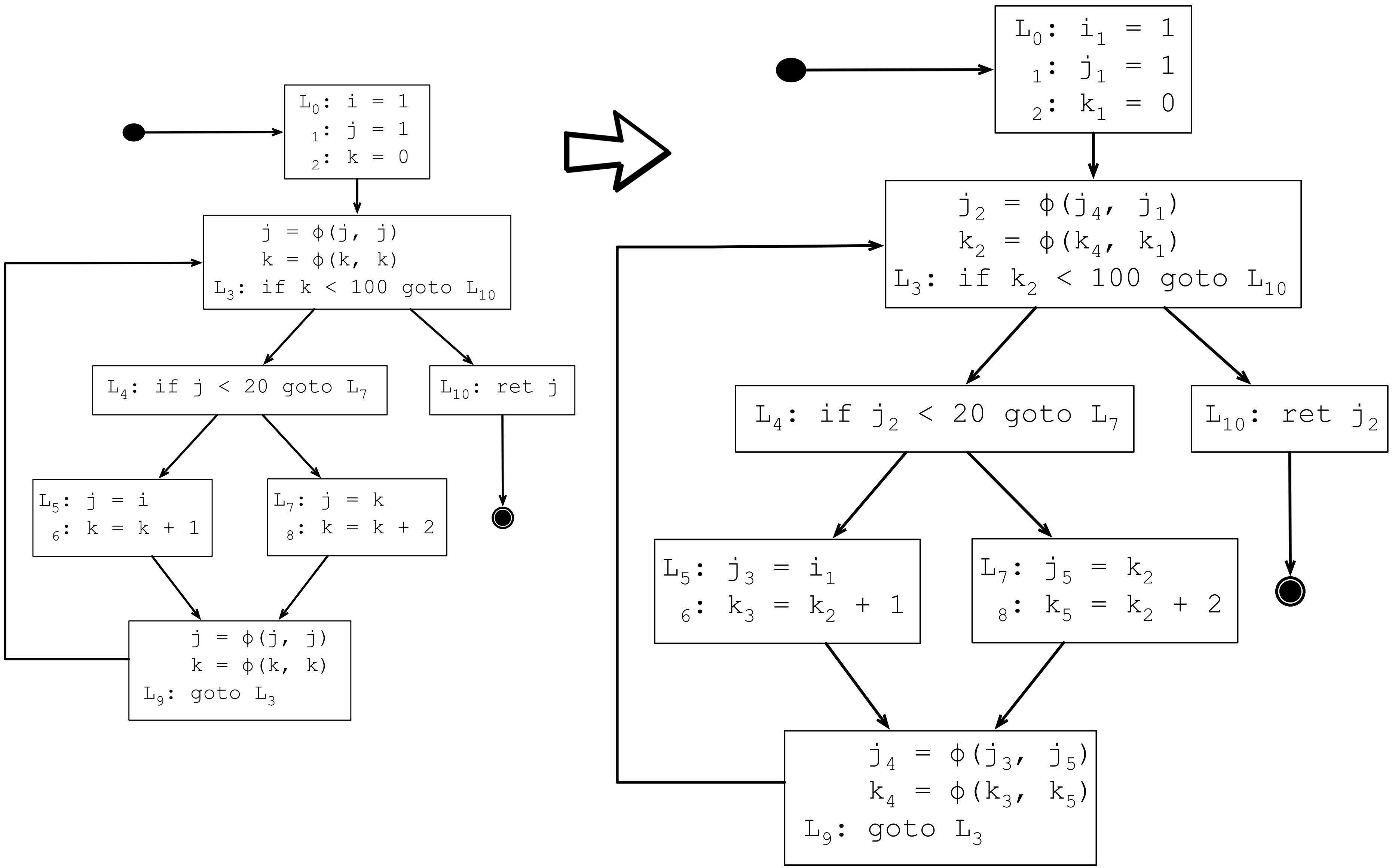


The Arity of Phi-Functions

1) Could we have a phi-function in a node that has only one predecessor?
2) Could we have a phi-function with more than two arguments?
3) Can you rename the variables in this program?

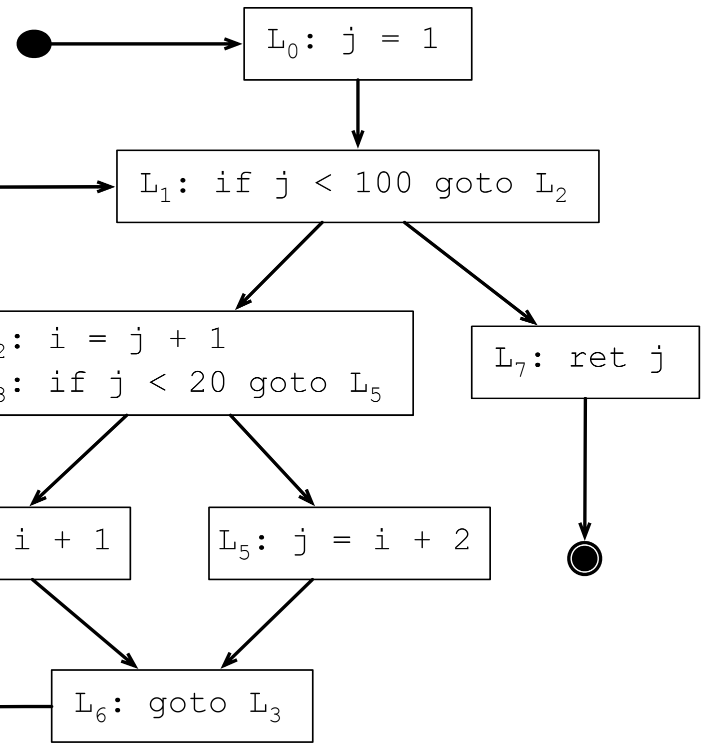


After Variable Renaming



Pruning SSA Form

- The algorithm that we just described computes what is called *Minimal SSA Form*.
- This name may be a bit misleading: it is minimal according to the definition of SSA, but it may create dead variables.



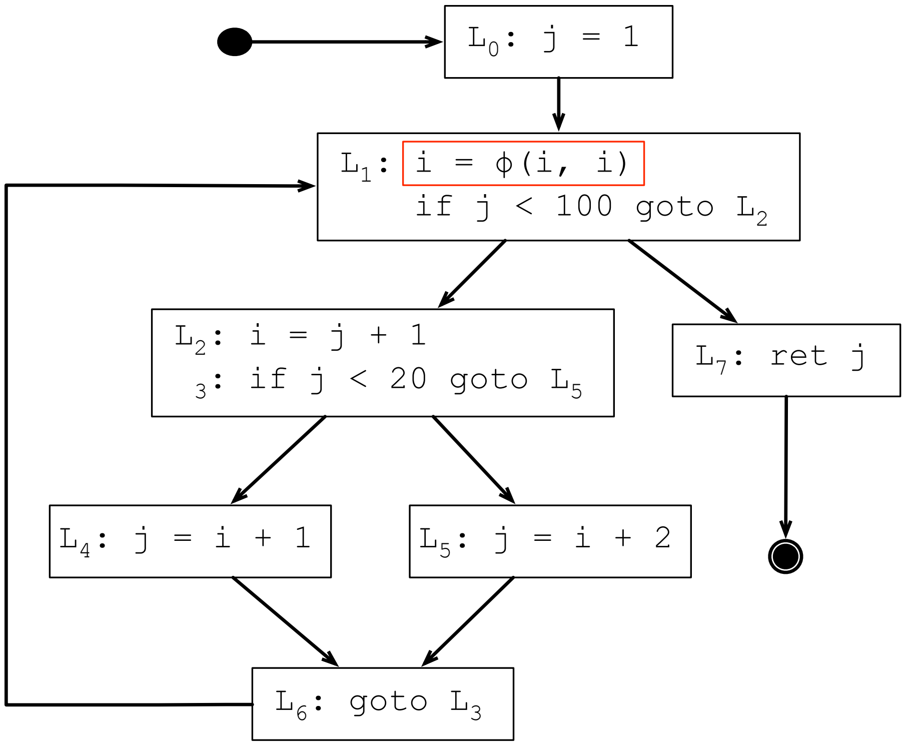
Where are we going to have phi-functions for variable i in this program?

Pruning SSA Form

- The algorithm that we just described computes what is called *Minimal SSA Form*.

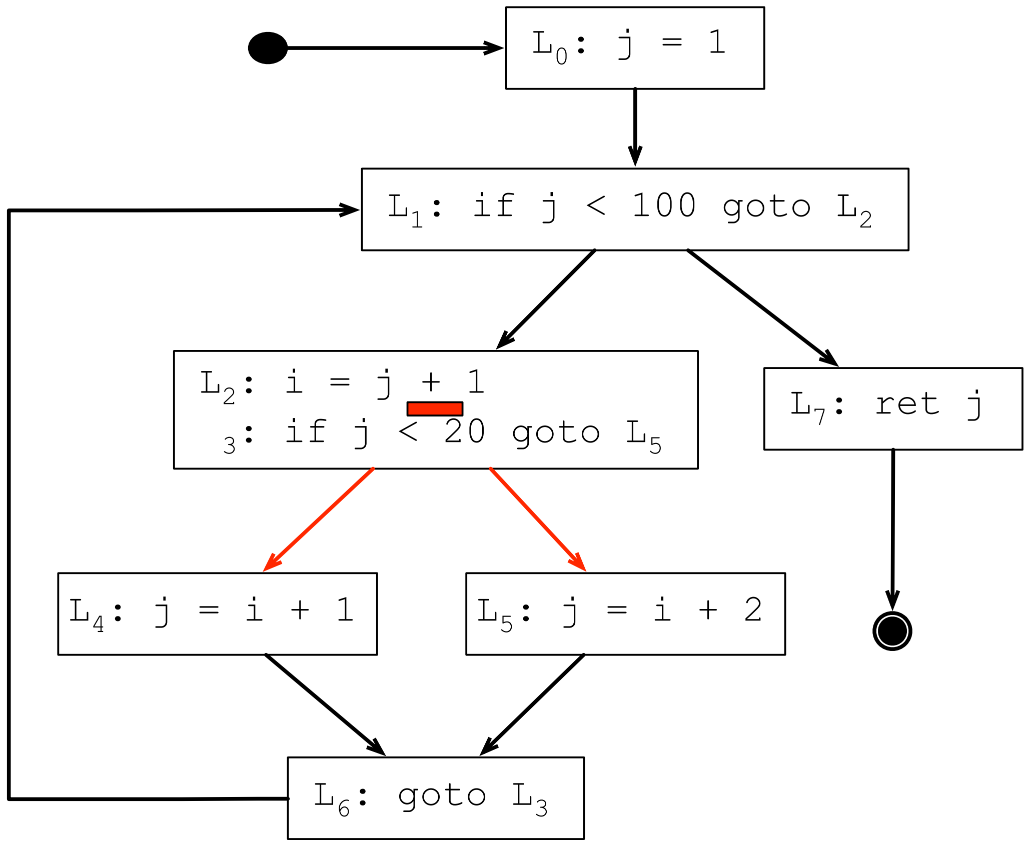
We have a phi-function for i at L_1, because this block is in the dominance frontier of L_2, a block where i is defined. This phi function exists even though it is not useful at all.

How could we eliminate useless phi-functions like the one in this example?



Pruned SSA Form

- We can add a liveness check to the algorithm that inserts phi-functions, in such a way that we only add a phi-function i = φ(i, …, i), at a program point p if i is alive at p.



In our example, i is only alive at the spots painted in red. Thus, there is no need to insert a phi-function at L_1, given that i is not alive there.

SPARSE ANALYSES

DCC 888

Sparse Analyses

- The Static Single Assignment form "sparsifies" many dataflow analyses.
- A sparse analysis associates information with the variable itself, instead of associating information with pairs formed by variables and program points.

These analyses require an essential property to work correctly. Which property?

Sparse Analyses

- The Static Single Assignment form "sparsifies" many dataflow analyses.
- A sparse analysis associates information with the variable itself, instead of associating information with pairs formed by variables and program points.
- These analyses only work correctly if the information associated with a variable is invariant along the entire live range of that variable. Examples of information include:
  - The variable is a constant
  - The variable is used somewhere
  - etc

Dead Code Elimination

- Dead code elimination is a code optimization that removes from the program instructions whose definitions have no uses.
- This optimization has a fairly simple implementation for SSA form programs:

while there is some variable v with no uses and the statement that defines v has no other side effects, delete the statement that defines v from the program.

What is the asymptotic complexity of this algorithm?

Dead Code Elimination

- We associate a counter with each variable.
- We traverse the program, and increment this counter each time the variable is used.
- Then we proceed to iterative mode:

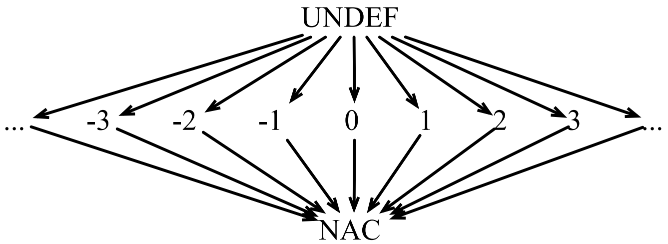
```
while there exists v, such that counter[v] = 0
  remove the instruction that defined v, e.g., "v = E"
  for each variable x used in E
    decrement counter[x]
```

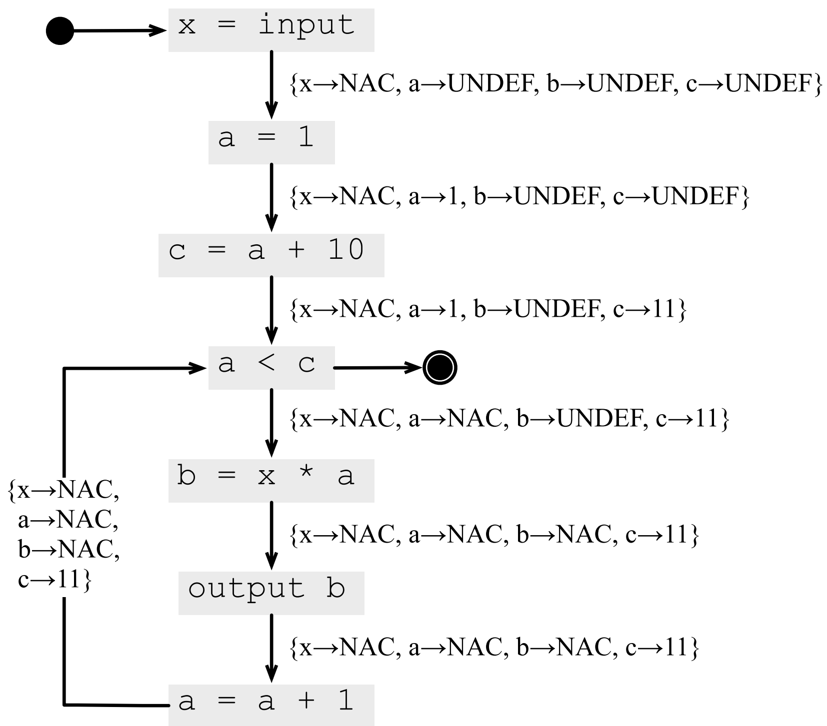
Can you think about data-structures that help you to implement this algorithm efficiently?

Sparse Constant Propagation

- We have seen constant propagation before.
- In our first try at this optimization, we would associate each variable with an element in the constant propagation lattice at each program point.
- The SSA form lets us simplify and improve this algorithm substantially.

How does the SSA form improve CP?

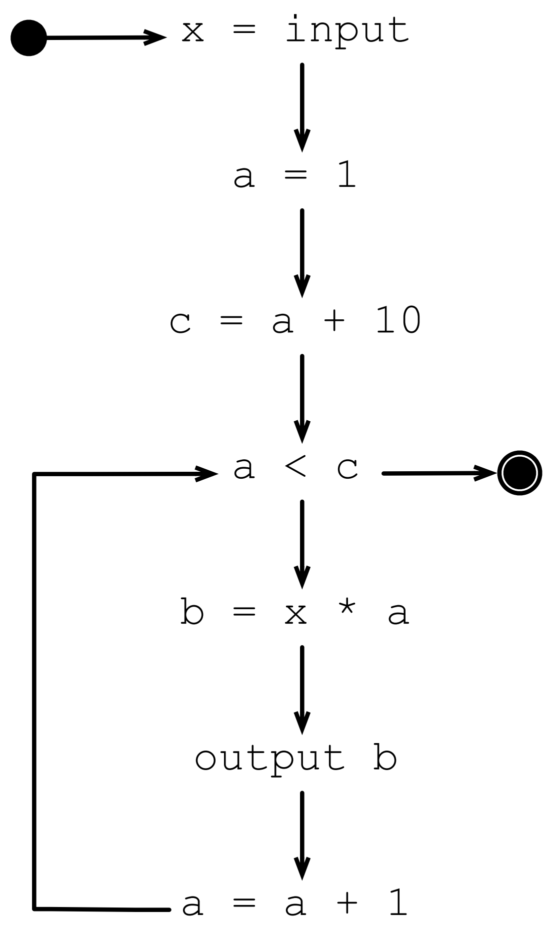




Sparse Constant Propagation

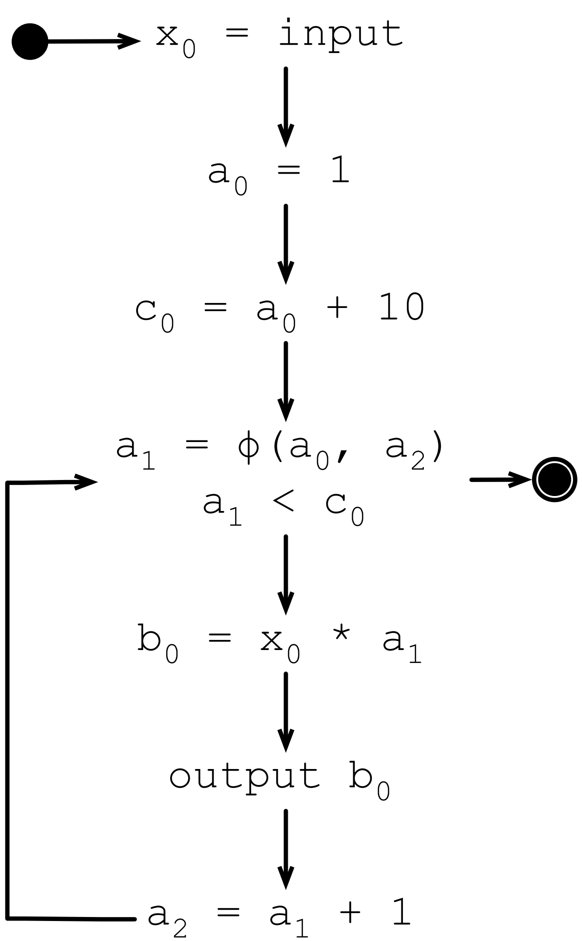
- The only event that determines if a variable is constant or not is its assignment.
- In SSA form programs, the assignment site is unique for each variable.
- And the information associated with a variable – that the variable is constant or not – does not change along the live range of that variable.

How does this example look like in SSA form?



Sparse Constant Propagation

- The only event that determines if a variable is constant or not is its assignment.
- In SSA form programs, the assignment site is unique for each variable.
- And the information associated with a variable – that the variable is constant or not – does not change along the live range of that variable.



Can you come up with a constraint system to solve constant propagation?

Sparse Constant Propagation

We associate each program variable v with an abstract state [v]. This abstract state is an element in the lattice of constant propagation.

The rules on the right define an abstract interpretation for each relevant instruction in the target program.

We keep interpreting these instructions abstractly, until the abstract state of each program variable stops chancing.

$v = c$ $$[\![v]\!] = c$$

$v = v'$ $$\frac{[\![v']\!] = a}{[\![v]\!] = a}$$

$v = v_0 + v_1$ $$\frac{[\![v_0]\!] = c_0 \qquad [\![v_1]\!] = c_1}{[\![v]\!] = c_0 + c_1}$$

$$\frac{[\![v_i]\!] = NAC, i \in \{0, 1\}}{[\![v]\!] = NAC}$$

$v = \phi(v_0, \ldots, v_n)$ $$\frac{[\![v_i]\!] = a_i, 0 \leq i \leq n}{[\![v]\!] = c_0 \ \wedge \ \ldots \ \wedge \ c_n}$$

Sparse Constant Propagation

1) Do you remember the meaning of the meet operator $\wedge$?

2) How can we avoid testing if [v] is UNDEF in the addition?

3) What is the **time** complexity of solving this problem?

4) What is the **space** complexity of solving this problem?

$v = c$ $\quad$ $[\![v]\!] = c$

$v = \texttt{input}$ $\quad$ $[\![v]\!] = NAC$

$v = v'$ $\quad$ $\dfrac{[\![v']\!] = a}{[\![v]\!] = a}$

$v = v' + c$ $\quad$ $\dfrac{[\![v']\!] = c'}{[\![v]\!] = c' + c}$

$v = v' + c$ $\quad$ $\dfrac{[\![v']\!] = NAC}{[\![v]\!] = NAC}$

$v = v_0 + v_1$ $\quad$ $\dfrac{[\![v_0]\!] = c_0 \qquad [\![v_1]\!] = c_1}{2 \;\; [\![v]\!] = c_0 + c_1}$

$\dfrac{[\![v_i]\!] = NAC, i \in \{0, 1\}}{[\![v]\!] = NAC}$

$v = \phi(v_0, \ldots, v_n)$ $\quad$ $\dfrac{[\![v_i]\!] = a_i, 0 \leq i \leq n}{[\![v]\!] = a_0 \ \wedge \ \ldots \ \wedge \ a_n}$

Solution Guided by the Dominator Tree

- The Static Single Assignment gives us a very efficient way to solve these constraints: we can interpret them in the order defined by the dominator tree.
  - If we follow this ordering, then we are guaranteed that upon finding statements other than phi-functions, the parameters of these statements will have been assigned an abstract state.

```
x0 = input
a0 = 1
c0 = a0 + 10
a1 = ϕ(a0, a2)
a1 < c0
b0 = x0 * a1
output b0
a2 = a1 + 1
```

$[x_0] = \text{NAC}$

What is going to be the abstract state of a_0?

Why and when is this statement true?

Solution Guided by the Dominator Tree

- We evaluate each instruction according to their abstract interpretation.

$$v = c \qquad\qquad [\![v]\!] = c$$

$$v = \texttt{input} \qquad\qquad [\![v]\!] = NAC$$

```
x0 = input
a0 = 1
c0 = a0 + 10
a1 = ϕ(a0, a2)
a1 < c0
b0 = x0 * a1
output b0
a2 = a1 + 1
```

$[x_0] = \text{NAC}$

$[a_0] = 1$

What is going to be the abstract state of c_0?

Solution Guided by the Dominator Tree

- Therefore, our abstract interpretation system must have as many entries as there are relevant statements in the programming language used to write the program that we are analyzing.

$$v = v' + c \qquad \frac{[\![v']\!] = c'}{[\![v]\!] = c' + c}$$

$$v = v' + c \qquad \frac{[\![v']\!] = NAC}{[\![v]\!] = NAC}$$

```
x0 = input          [x0] = NAC
   ↓
a0 = 1              [a0] = 1
   ↓
c0 = a0 + 10        [c0] = 11
   ↓
a1 = ϕ(a0, a2)
a1 < c0
   ↓
b0 = x0 * a1
   ↓
output b0
   ↓
a2 = a1 + 1
```

What is going to be the abstract state of a_1 and b_0?

Solution Guided by the Dominator Tree

- Phi-Functions are just a bit trickier: once we find them, we may not have seen all their parameters, even if we go through the dominance three.
  - This is only true for phi-functions.

Why?

- But, if we have not seen the argument before, than its value is UNDEF, and UNDEF $\wedge$ a = a for any abstract state a.

```
x0 = input
a0 = 1
c0 = a0 + 10
a1 = ϕ(a0, a2)
a1 < c0
b0 = x0 * a1
output b0
a2 = a1 + 1
```

$[x_0] = \text{NAC}$

$[a_0] = 1$

$[c_0] = 11$

$[a_1] = 1$

$[b_0] = \text{NAC}$

What is going to be the abstract state of a_2?

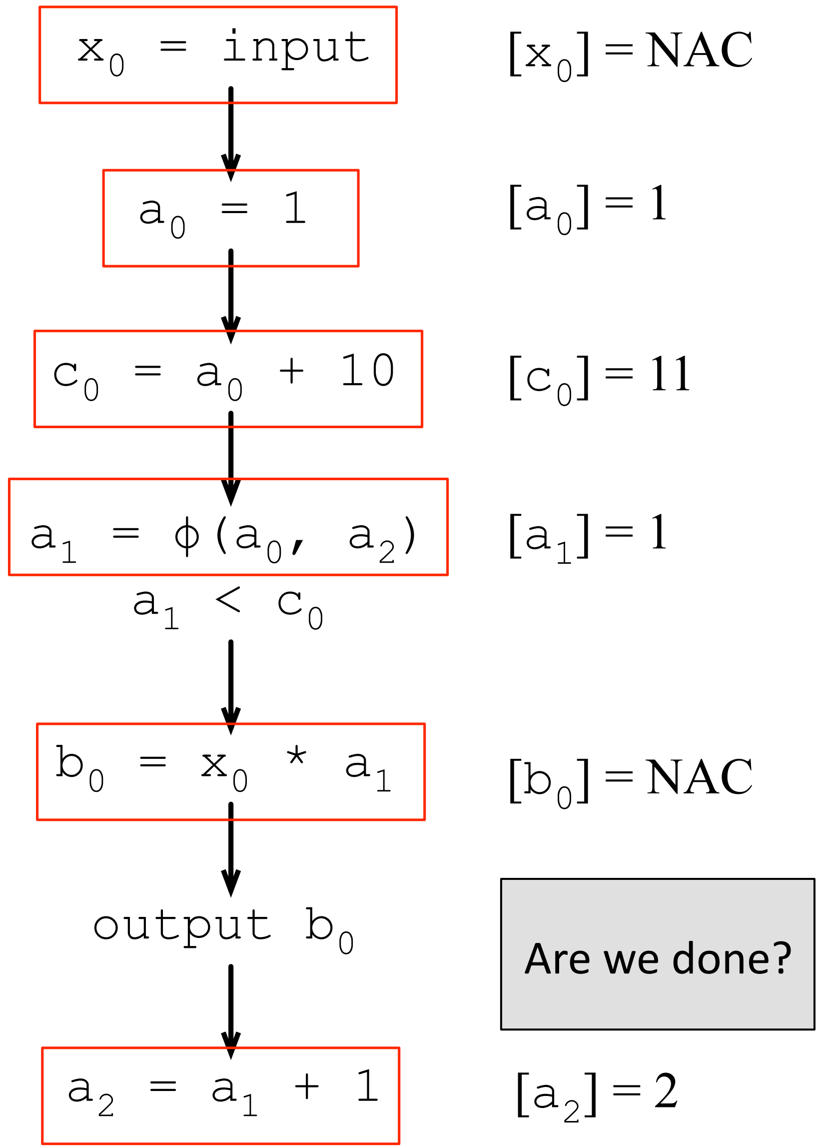
Solution Guided by the Dominator Tree

- Notice that even though we may have many different instructions in our intermediate language, the abstract semantics of many of them may be the same.
- As an example, multiplication, addition, subtraction, and most of the binary operations have the same abstract semantics.

$$v = v_0 + v_1$$

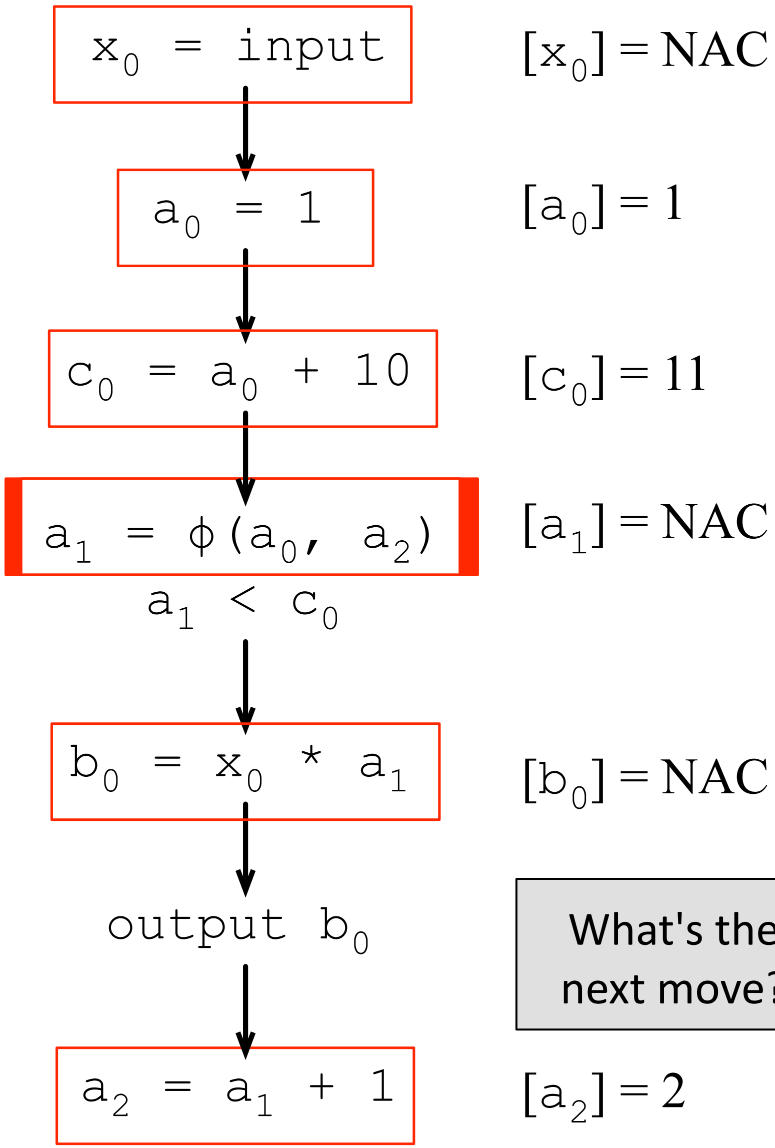
$$\frac{[\![v_0]\!] = c_0 \qquad [\![v_1]\!] = c_1}{2 \;\; [\![v]\!] = c_0 + c_1}$$

$$\frac{[\![v_i]\!] = NAC, i \in \{0, 1\}}{[\![v]\!] = NAC}$$



Solution Guided by the Dominator Tree

- We may have to iterate the propagation of information.
- Changes will happen initially at phi-functions, because it is possible that not all their arguments had been initialized when we first found them.



What's the next move?

Solution Guided by the Dominator Tree

- The propagation of abstract states can be implemented very efficiently: we only need to propagate information from a variable v to the variables u defined by instructions that use v.
- In this example, we only need to propagate the new abstract state of a_1 to a_2, as this variable is defined in an instruction that uses a_1.

x_0 = input — $[x_0] = \text{NAC}$

↓

a_0 = 1 — $[a_0] = 1$

↓

c_0 = a_0 + 10 — $[c_0] = 11$

↓

$a_1 = \phi(a_0, a_2)$ — $[a_1] = \text{NAC}$

$a_1 < c_0$

↓

$b_0 = x_0 * a_1$ — $[b_0] = \text{NAC}$

↓

output b_0

↓

$a_2 = a_1 + 1$ — $[a_2] = \text{NAC}$

Again: what is the complexity of this algorithm?

And again: are we done?

```
bool ConstantPropagation::runOnFunction(Function &F) {
  // Initialize the worklist to all of the instructions ready to process...
  std::set<Instruction*> WorkList;
  for(inst_iterator i = inst_begin(F), e = inst_end(F); i != e; ++i) {
      WorkList.insert(&*i);
  }
  bool Changed = false;

  while (!WorkList.empty()) {
    Instruction *I = *WorkList.begin();
    WorkList.erase(WorkList.begin());   // Get an element from the worklist
    if (!I->use_empty())                // Don't muck with dead instructions
      if (Constant *C = ConstantFoldInstruction(I)) {
        // Add all of the users of this instruction to the worklist, they
        //  might be constant propagatable now...
        for (Value::use_iterator UI = I->use_begin(), UE = I->use_end();
             UI != UE; ++UI)
          WorkList.insert(cast<Instruction>(*UI));
        // Replace all of the uses of a variable with uses of the constant.
        I->replaceAllUsesWith(C);
        // Remove the dead instruction.
        WorkList.erase(I);
        I->eraseFromParent();
        // We made a change to the function...
        Changed = true;
        ++NumInstKilled;
      }
  }
  return Changed;
}
```

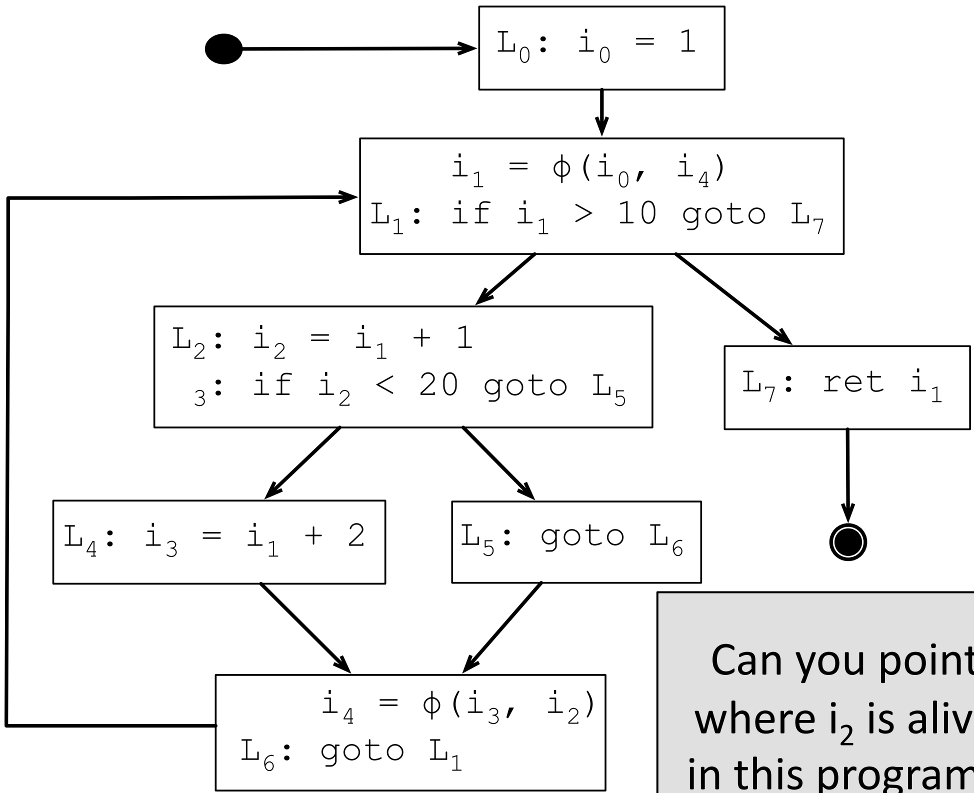
1) Where is the abstract interpretation implemented?

2) Can you see a "graph" in this implementation of the constant propagation algorithm?

♧:This code has been taken from `llvm/lib/Transforms/Scalar/ConstantProp.cpp`

Liveness Analysis in SSA Form Programs

- The problem of determining the program points along which a variable is alive has a simple solution for SSA form programs.



Can you point where i_2 is alive in this program?

For each statement S in the program:
    IN[S] = OUT[S] = {}

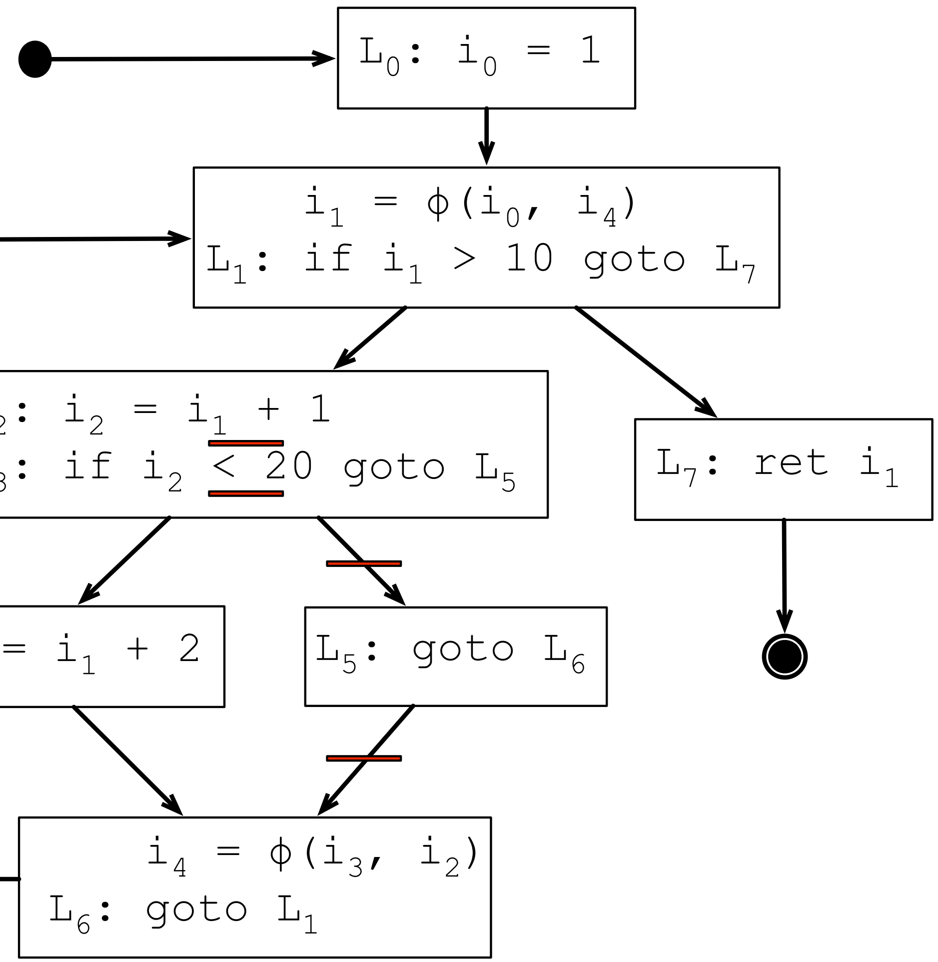
For each variable v in the program:
    For each statement S that uses v:
        live(S, v)

live(S, v):
    IN[S] = IN[S] ∪ {v}
    For each P in pred(S):
      if P does not define v
          OUT[P] = OUT[P] ∪ {v}
          live(P, v)

Liveness Analysis in SSA Form Programs

The points where i2 is alive have been marked with red rectangles.

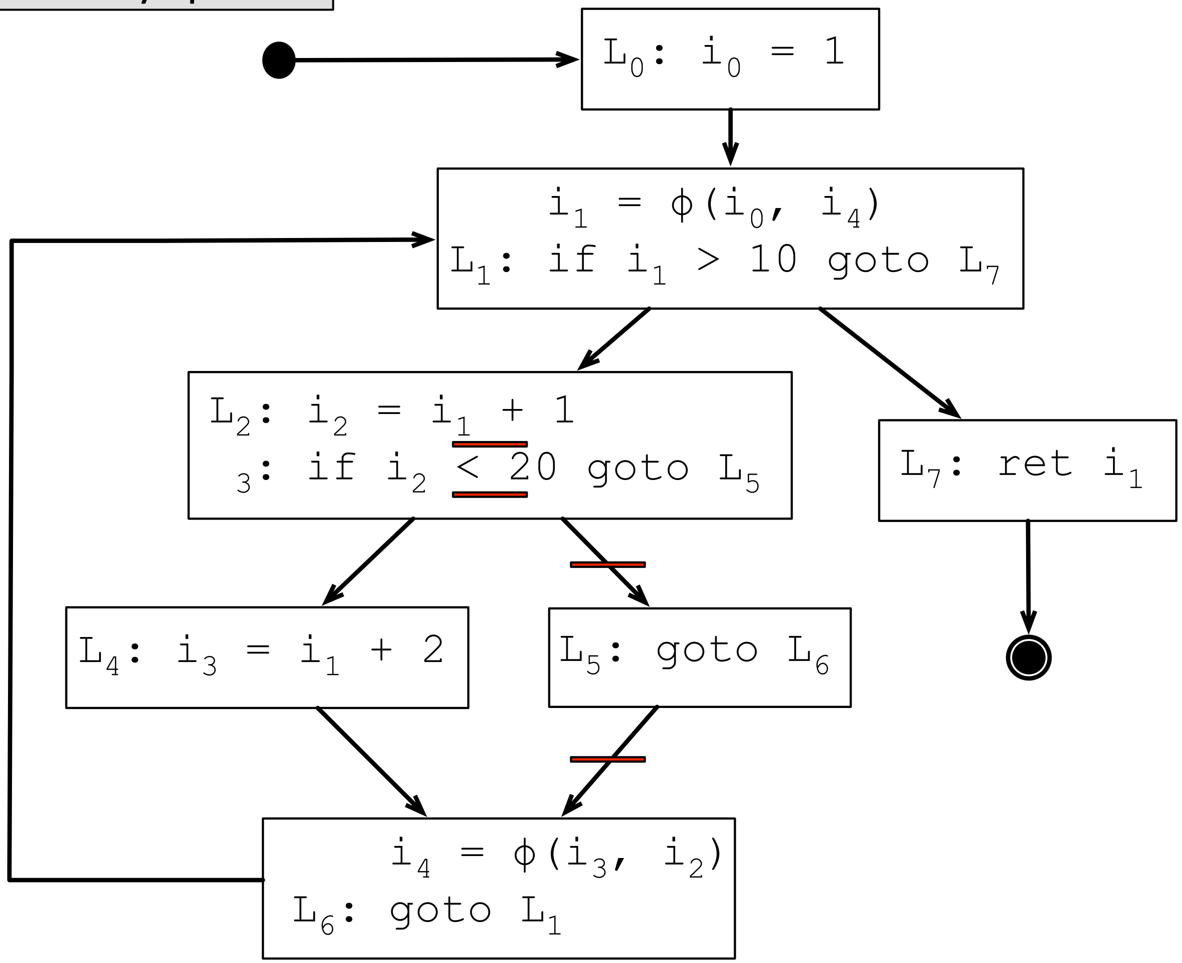
Tricky question: is i_2 alive anywhere at block L_6?



Liveness Analysis in SSA Form Programs

The answer for the tricky question is **NO**. Uses of variables in phi-functions are considered in a different way. The variable is effectively used in the OUT set of the predecessor block where its definition comes from. In other words, i_2 is alive at OUT[L_5], but is not alive at IN[L_6].

Could i_2 and i_3 be allocated into the same memory space?



Liveness Analysis in SSA Form Programs

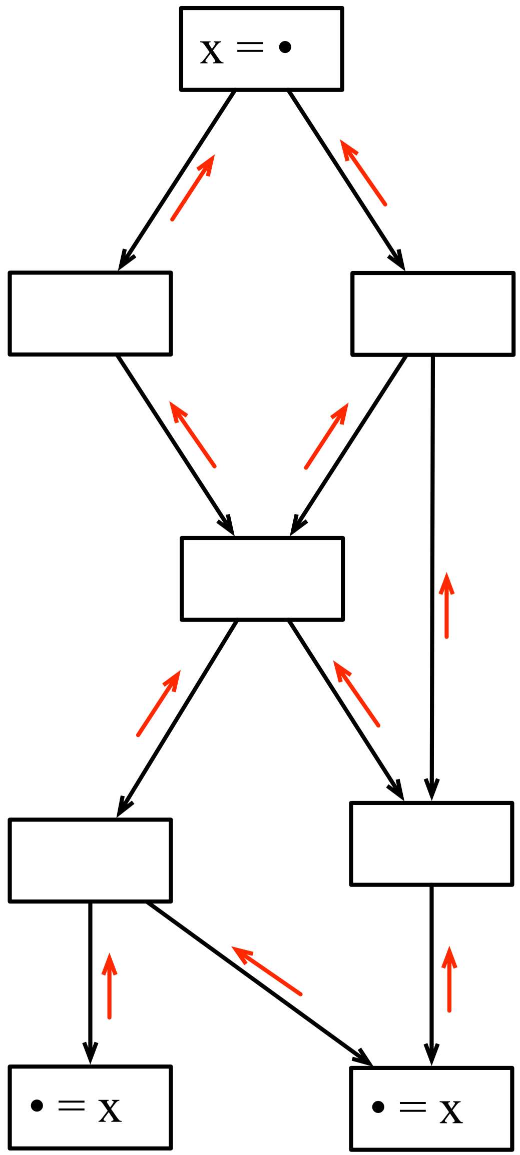
Why can we solve liveness analysis for SSA form programs without having to iterate through a fixed point algorithm?

```
For each statement S in the program:
  IN[S] = OUT[S] = {}

For each variable v in the program:
  For each statement S that uses v:
    live(S, v)

live(S, v):
  IN[S] = IN[S] ∪ {v}
  For each P in pred(S):
   if P does not define v
     OUT[P] = OUT[P] ∪ {v}
     live(P, v)
```

Liveness Analysis in SSA Form Programs



Our algorithm works due to the key property of SSA form programs: every use of a variable v is dominated by the definition of v. Thus, we can traverse the CFG of the program, starting from the uses of a variable, until we stop at its definition. We are certain to stop, because of the key property. Otherwise, the variable is used without being defined. In this case, we will reach the root node of the CFG, and we assume that the variable is alive at the input of the program.

A Bit of History

- The Static Single Assignment form was introduced by Ron Cytron, in 1989
- Compilers usually find dominators via Lengauer/Tarjan's algorithm.
- There are many flavors of SSA form. One of the most common is the pruned SSA form, due to Briggs *et al*.

- Cytron, R. and Ferrante, J. and Rosen, B. and Wegman, M. and Zadeck, F. "An Efficient Method of Computing Static Single Assignment Form", POPL, (1989) pp 25-35
- Lengauer, T. and Tarjan, R. "A Fast Algorithm for Finding Dominators in a Flowgraph", TOPLAS, 1:1 (1979) pp 121-141
- Briggs, P. and Cooper, K. and Harvey, J. and Simpson, L. "Practical Improvements to the Construction and Destruction of Static Single Assignment Form", SP&E (28:8), (1998) pp 859-881