SSA and CPS

CS4410: Spring 2013

Monadic Form vs CFGs

Consider CFG available exp. analysis:

statement	gen's	kill's
$x{:=}v_1\ p\ v_2$	$x{:=}v_1\ p\ v_2$	{y:=e | x=y or x in e}

When variables are immutable, simplifies to:

statement	gen's	kill's
$x{:=}v_1\ p\ v_2$	$x{:=}v_1\ p\ v_2$	{}

(Assumes variables are unique.)

Monadic Form vs CFGs

Almost all data flow analyses simplify when variables are defined once.

- no kills in dataflow analysis
- can interpret as either functional or imperative

Our monadic form had this property, which made many of the optimizations simpler.

- e.g., just keep around a set of available definitions that we keep adding to.

On the other hand…

CFGs have their own advantages over monadic form.

- support control-flow *graphs* not just trees.

```
if b < c then
  let x1 = e1
      x2 = e2
      x3 = x1 + x2
  in x3
else
  let x1 = e4
      x2 = e5
      x3 = x1 + x2
  in x3
```

```
    if b >= c goto L1
    x1 := e1
    x2 := e2
    goto L2
L1: x1 := e4
    x2 := e5
L2: x3 := x1 + x2
```

Best of both worlds…

Static Single Assignment (SSA)
- CFGs but with functional variables
- A slight “hack” to make graphs work out
- Now widely used (e.g., LLVM).
- Intra-procedural representation only.

Continuation Passing Style (CPS)
- Inter-procedural representation.
- So slightly more general.
- Used by FP compilers (e.g., SML/NJ).

The idea behind SSA

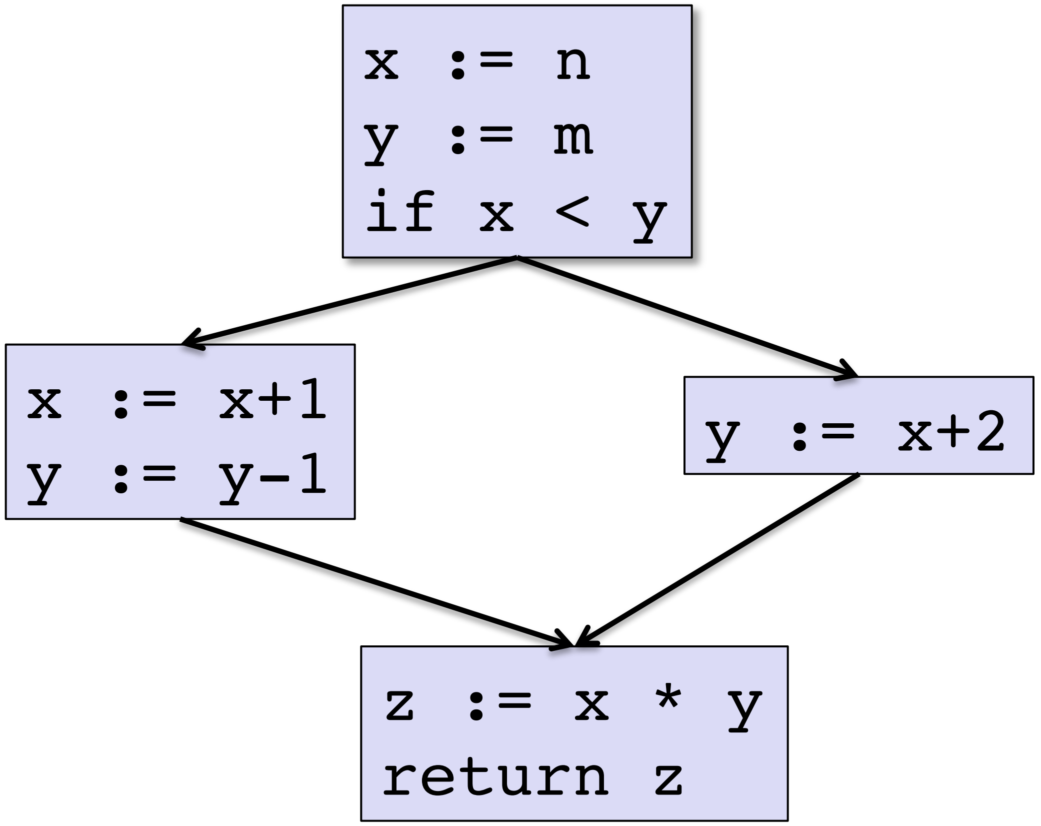
Start with CFG code and give each definition a fresh name, and propagate the fresh name to subsequent uses.

```
x := n
y := m
x := x + y
return x
```

```
x0 := n
y0 := m
x1 := x0 + y0
return x1
```

The problem…

What do we do with this?



```
x := n
y := m
if x < y
```

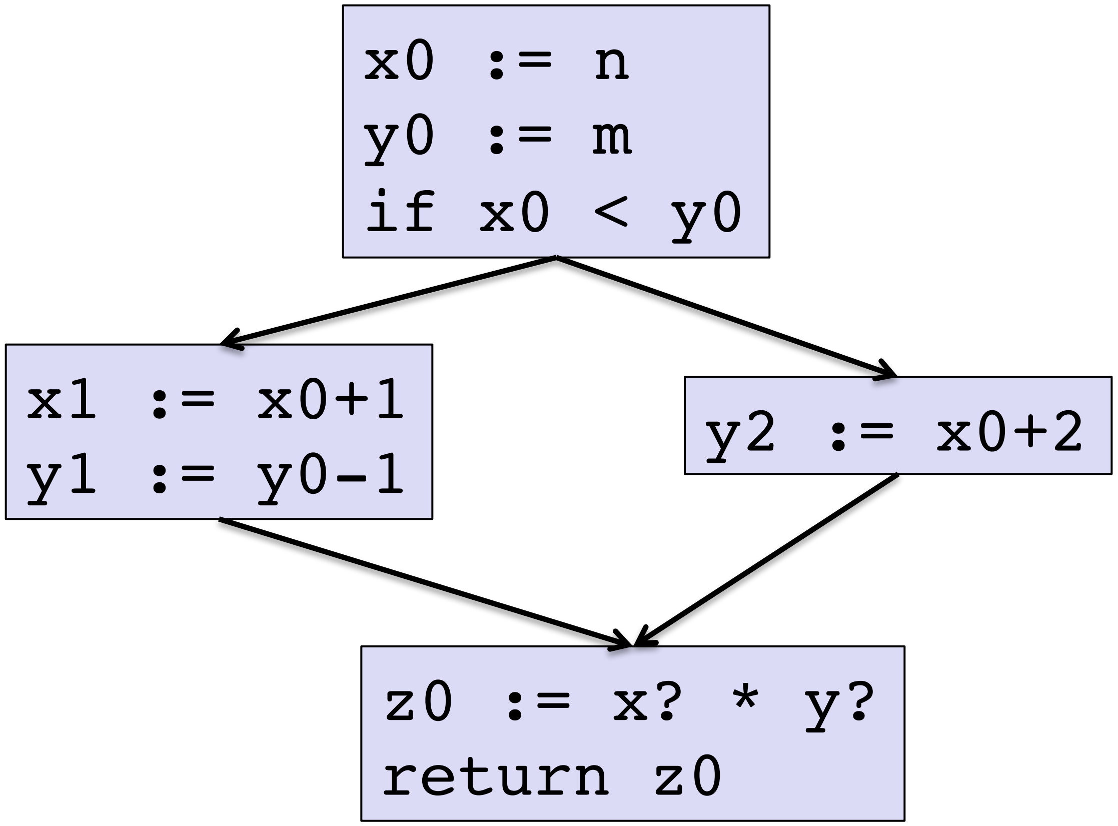
```
x := x+1
y := y-1
```

```
y := x+2
```

```
z := x * y
return z
```

The problem…

In particular, what happens at join points?

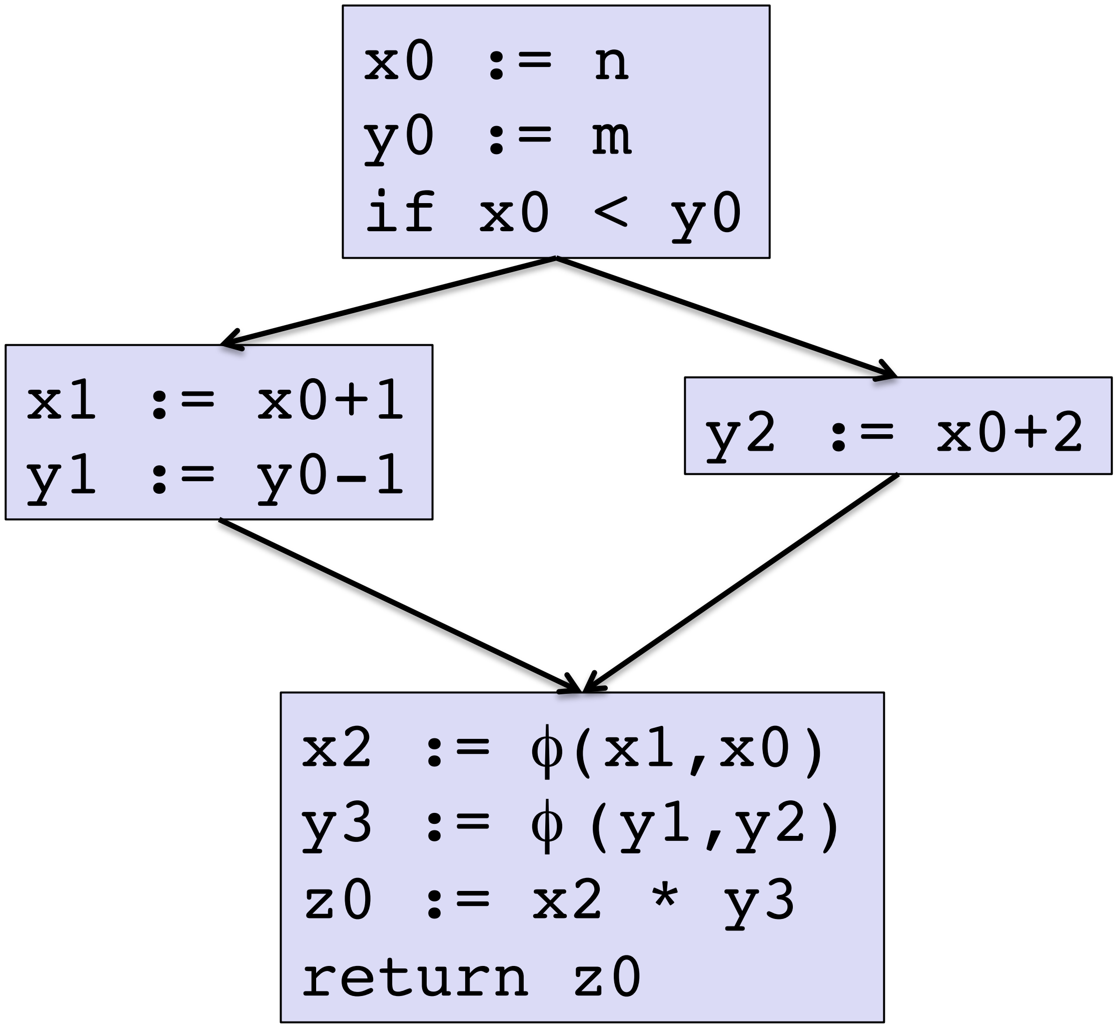

```
x0 := n
y0 := m
if x0 < y0
```

```
x1 := x0+1
y1 := y0-1
```

```
y2 := x0+2
```

```
z0 := x? * y?
return z0
```

The solution: “phony” nodes



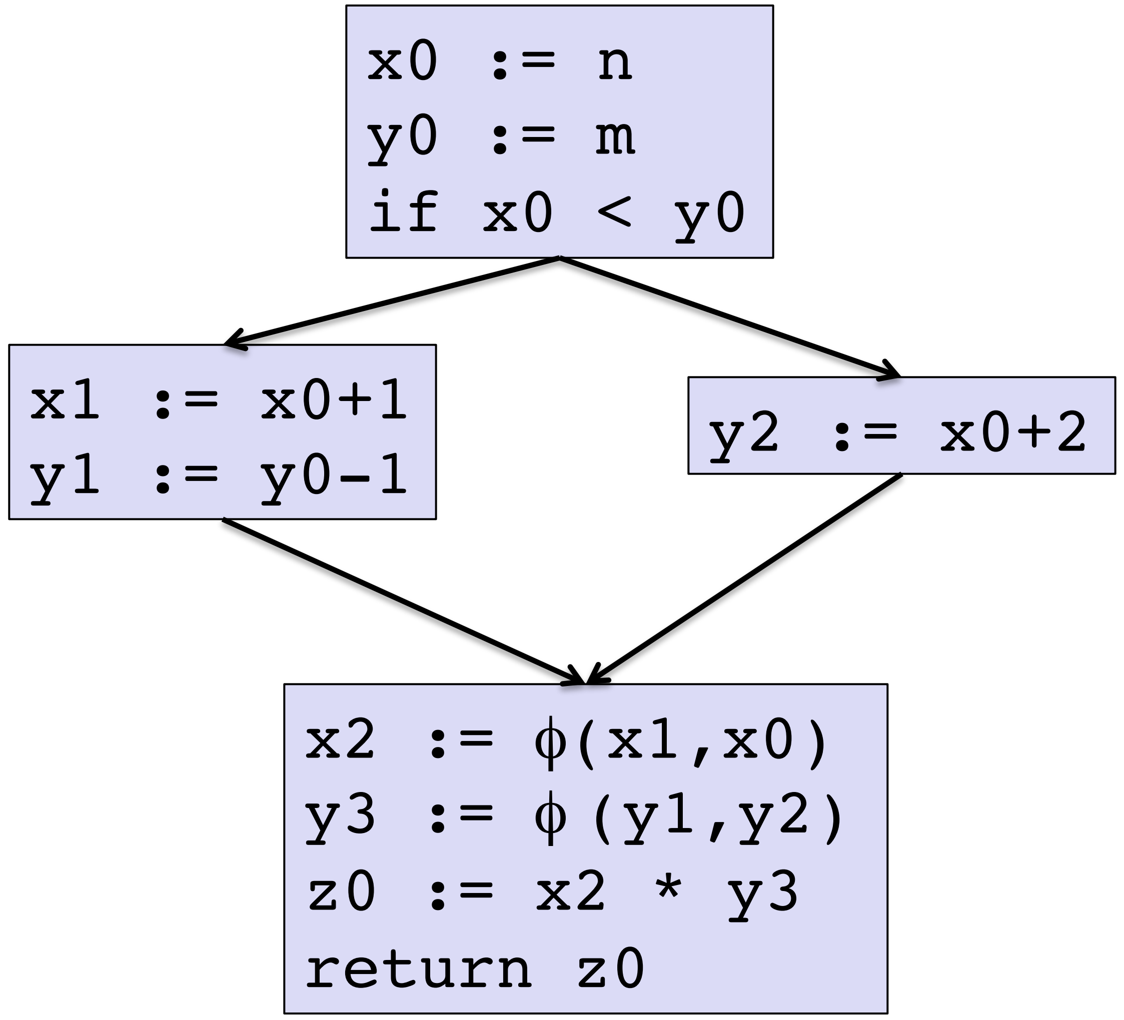
```
x0 := n
y0 := m
if x0 < y0
```

```
x1 := x0+1
y1 := y0-1
```

```
y2 := x0+2
```

```
x2 := ϕ(x1,x0)
y3 := ϕ(y1,y2)
z0 := x2 * y3
return z0
```

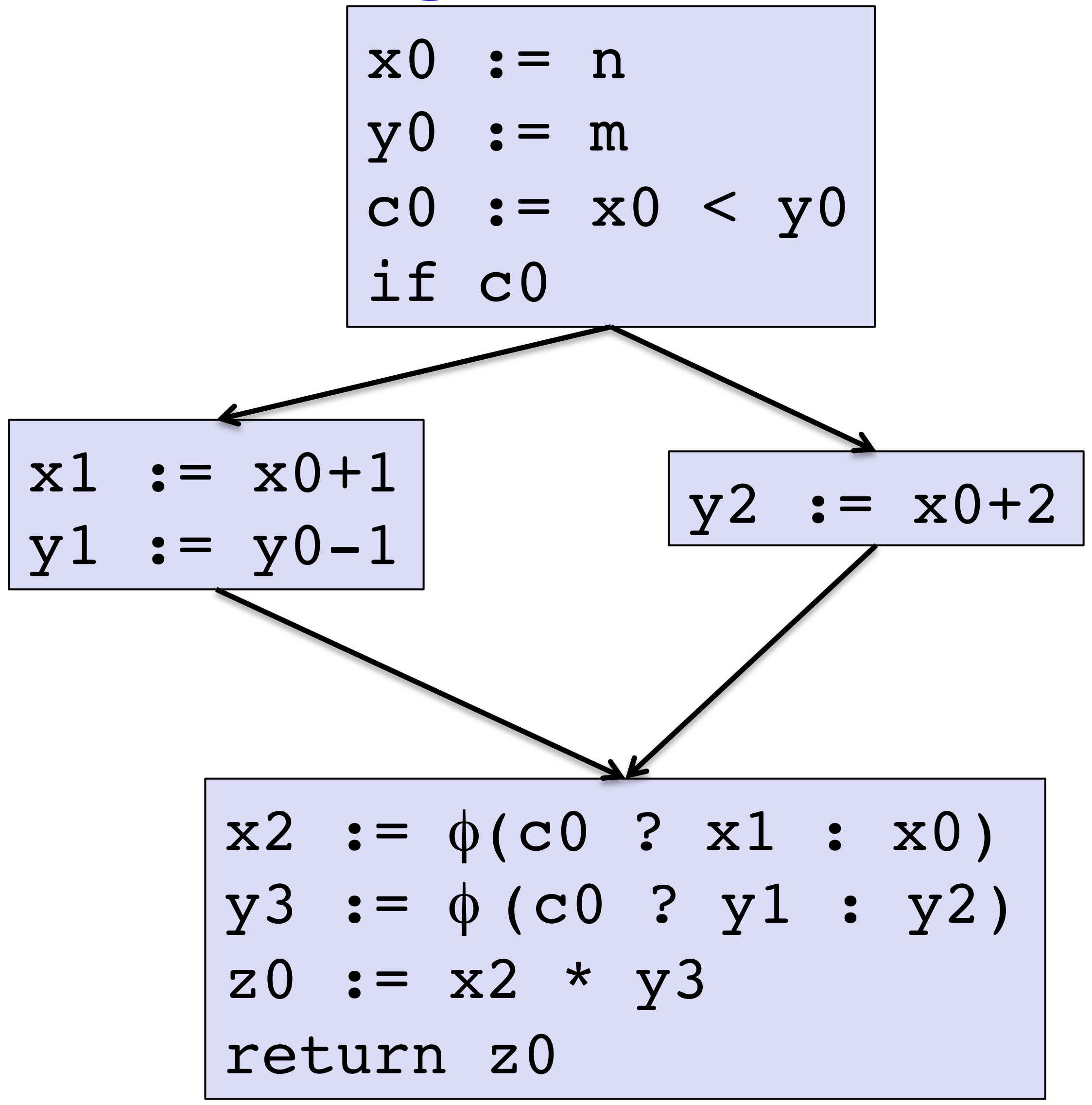
The solution: “phony” nodes



A phi-node is a phony “use” of a variable.

From an analysis standpoint, it’s as if an oracle chooses to set `x2` to either `x1` or `x0` based on how control got here.

Variant: gated SSA



```
x0  := n
y0  := m
c0  := x0 < y0
if c0
```

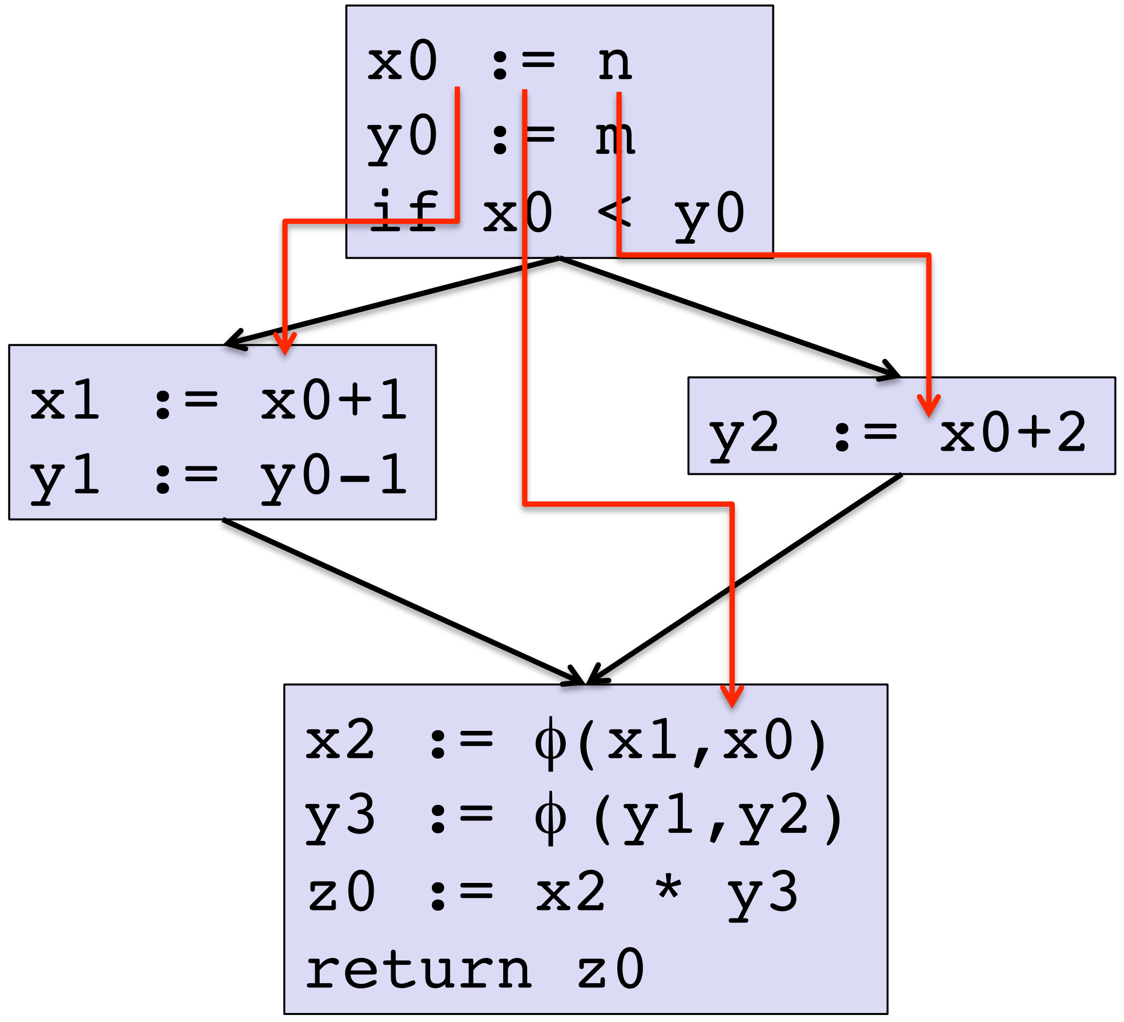
```
x1  := x0+1
y1  := y0-1
```

```
y2  := x0+2
```

```
x2  := ϕ(c0 ? x1 : x0)
y3  := ϕ (c0 ? y1 : y2)
z0  := x2 * y3
return z0
```

Use a functional "if" based on the tests that brought you here.

Back to normal SSA

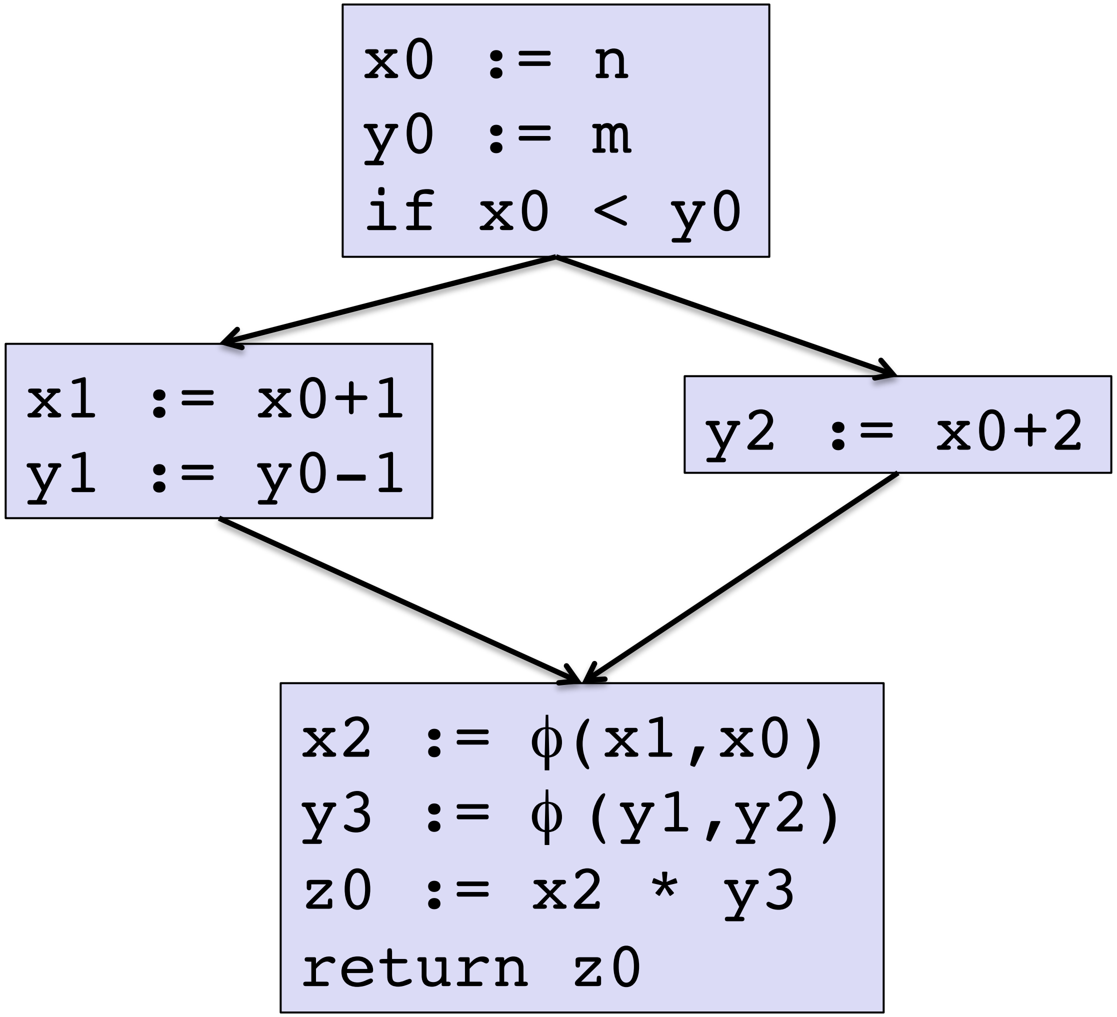


Compilers often build an overlay graph that connects definitions to uses ("def-use" chains.)

Then information really flows along these SSA def-use edges (e.g., constant propagation.)

Some practical benefits to SSA def-use over CFG…
(see Appel, Ex. 19.8)

Two Remaining Issues



```
x0 := n
y0 := m
if x0 < y0
```

```
x1 := x0+1
y1 := y0-1
```

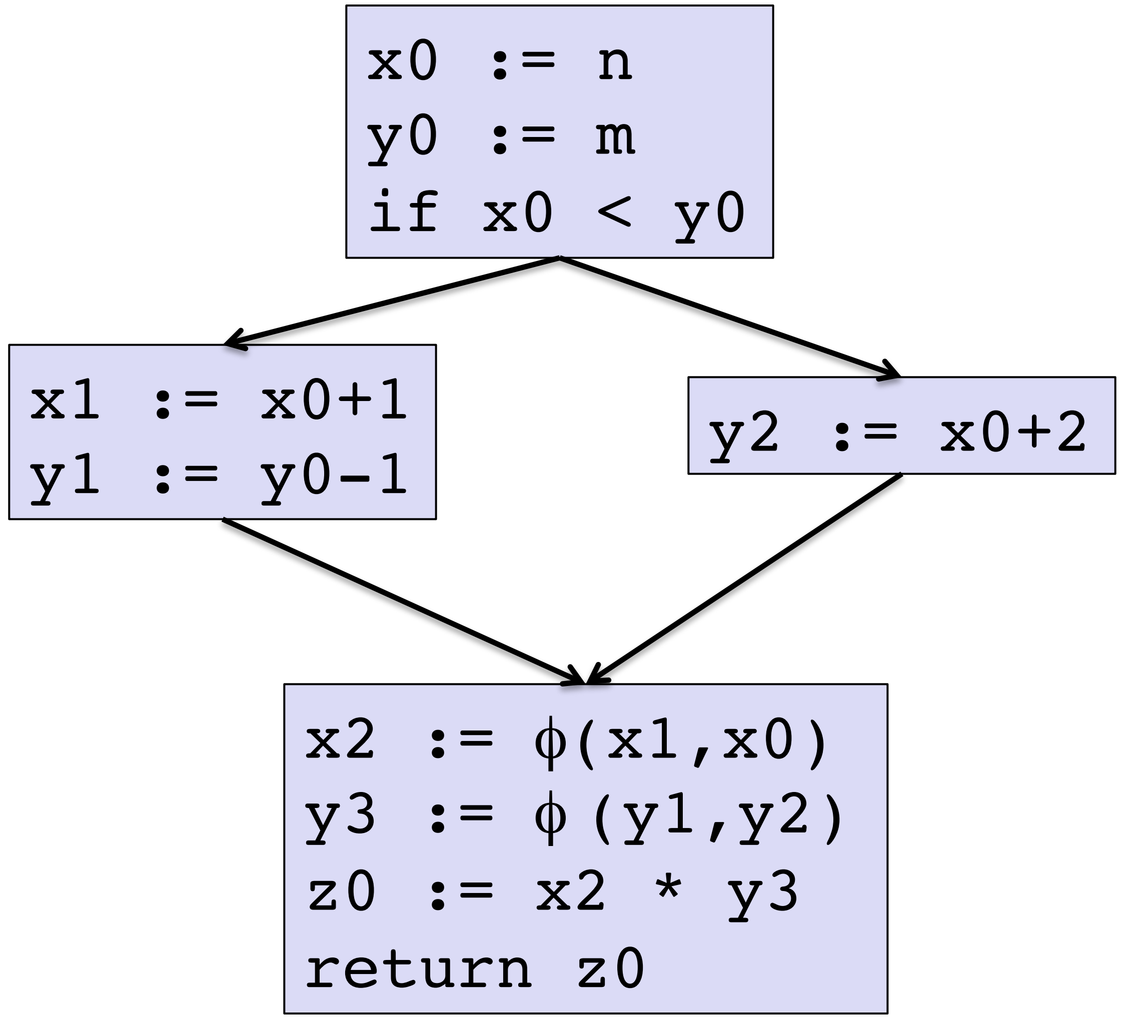
```
y2 := x0+2
```

```
x2 := φ(x1,x0)
y3 := φ (y1,y2)
z0 := x2 * y3
return z0
```

How do we generate SSA from the CFG representation?

How do we generate CFG (or MIPS) from the SSA?

SSA back to CFG



```
x0 := n
y0 := m
if x0 < y0
```

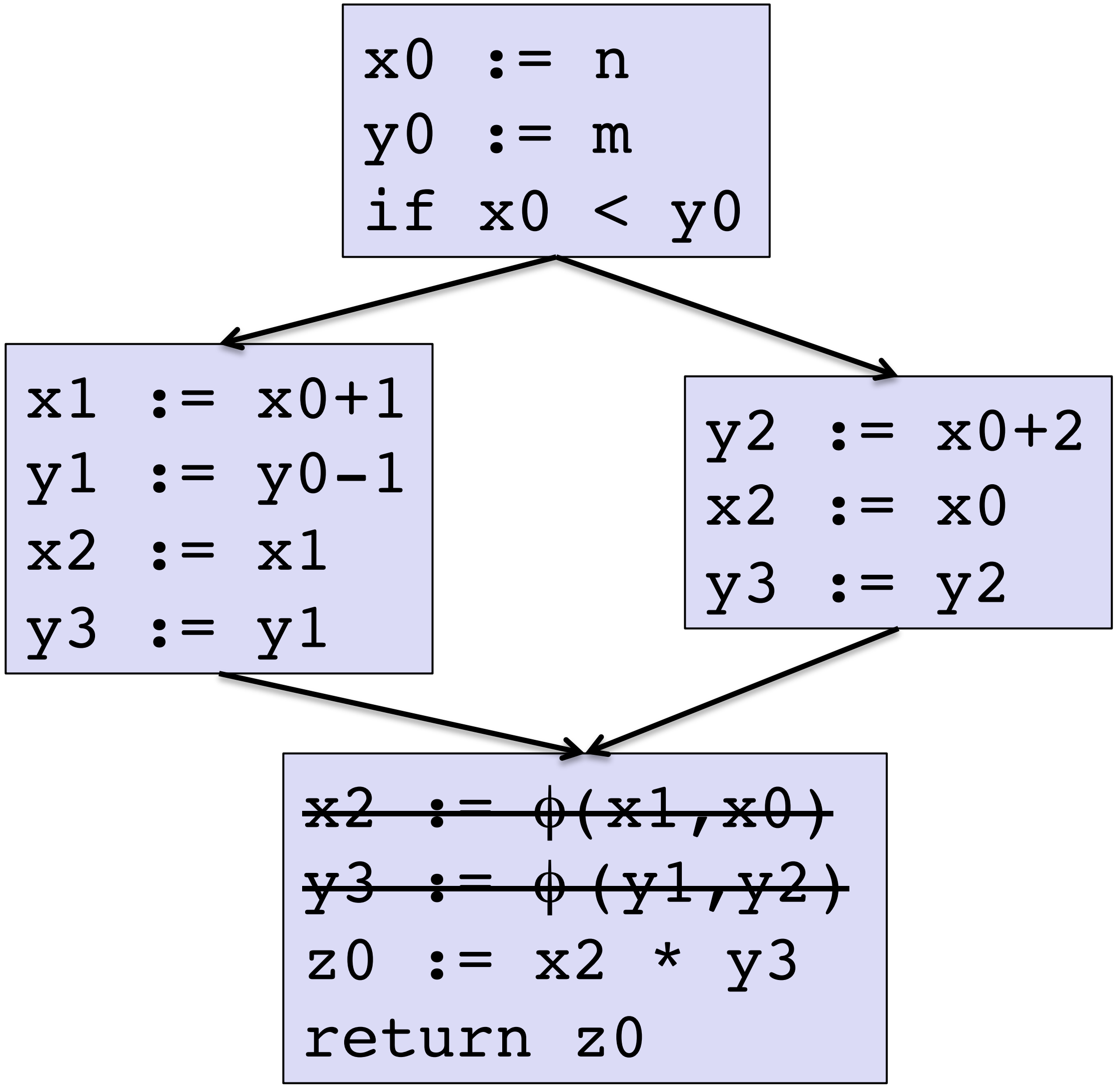
```
x1 := x0+1
y1 := y0-1
```

```
y2 := x0+2
```

```
x2 := ϕ(x1,x0)
y3 := ϕ (y1,y2)
z0 := x2 * y3
return z0
```

Just realize the assignments corresponding to the phi nodes on the edges.

SSA back to CFG



```
x0 := n
y0 := m
if x0 < y0
```

```
x1 := x0+1
y1 := y0-1
x2 := x1
y3 := y1
```

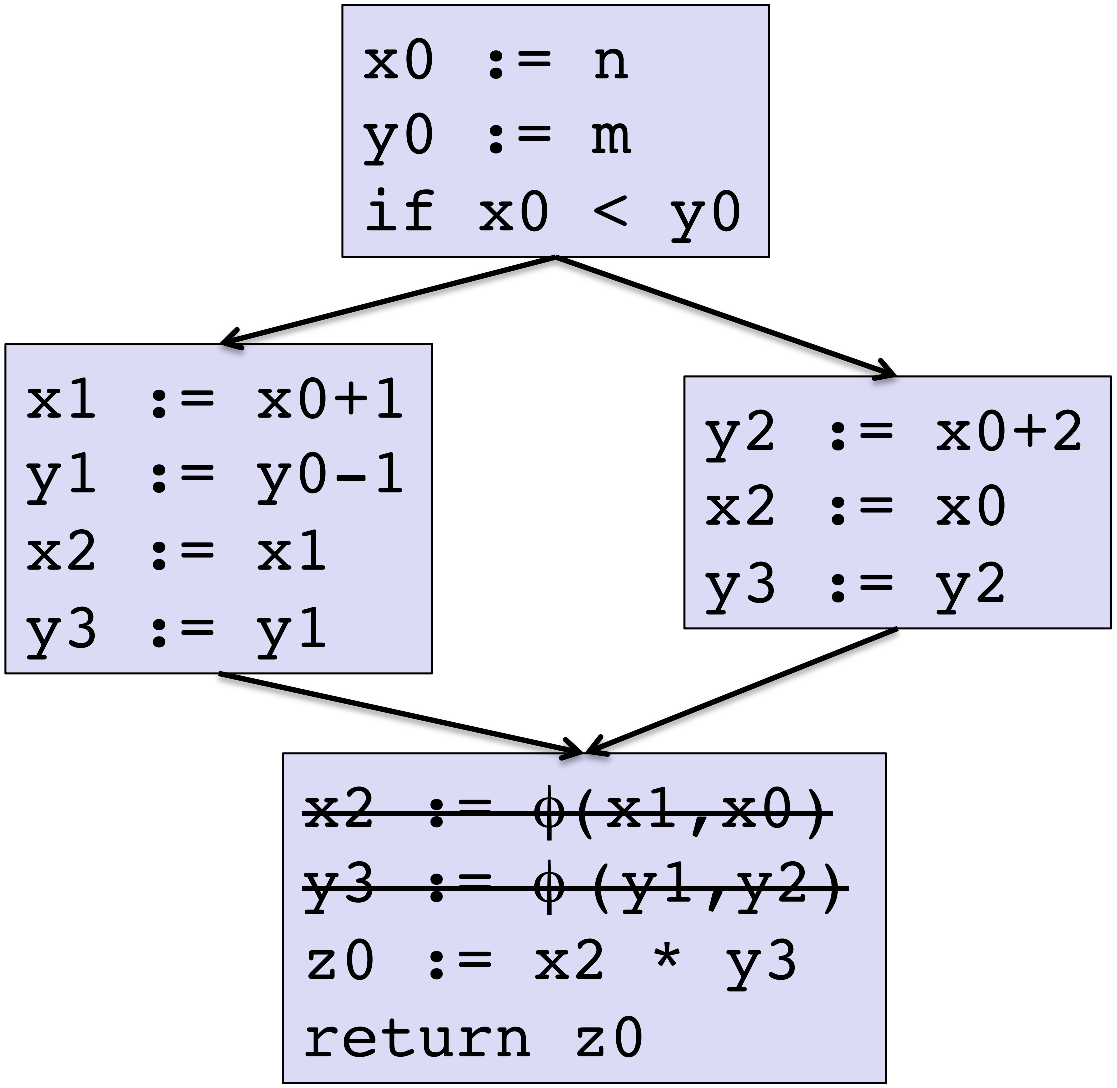
```
y2 := x0+2
x2 := x0
y3 := y2
```

~~x2 := ϕ(x1,x0)~~
~~y3 := ϕ (y1,y2)~~

```
z0 := x2 * y3
return z0
```

Just realize the assignments corresponding to the phi nodes on the edges.

SSA back to CFG



```
x0 := n
y0 := m
if x0 < y0
```

```
x1 := x0+1
y1 := y0-1
x2 := x1
y3 := y1
```

```
y2 := x0+2
x2 := x0
y3 := y2
```

~~x2 := ϕ(x1,x0)~~
~~y3 := ϕ (y1,y2)~~

```
z0 := x2 * y3
return z0
```

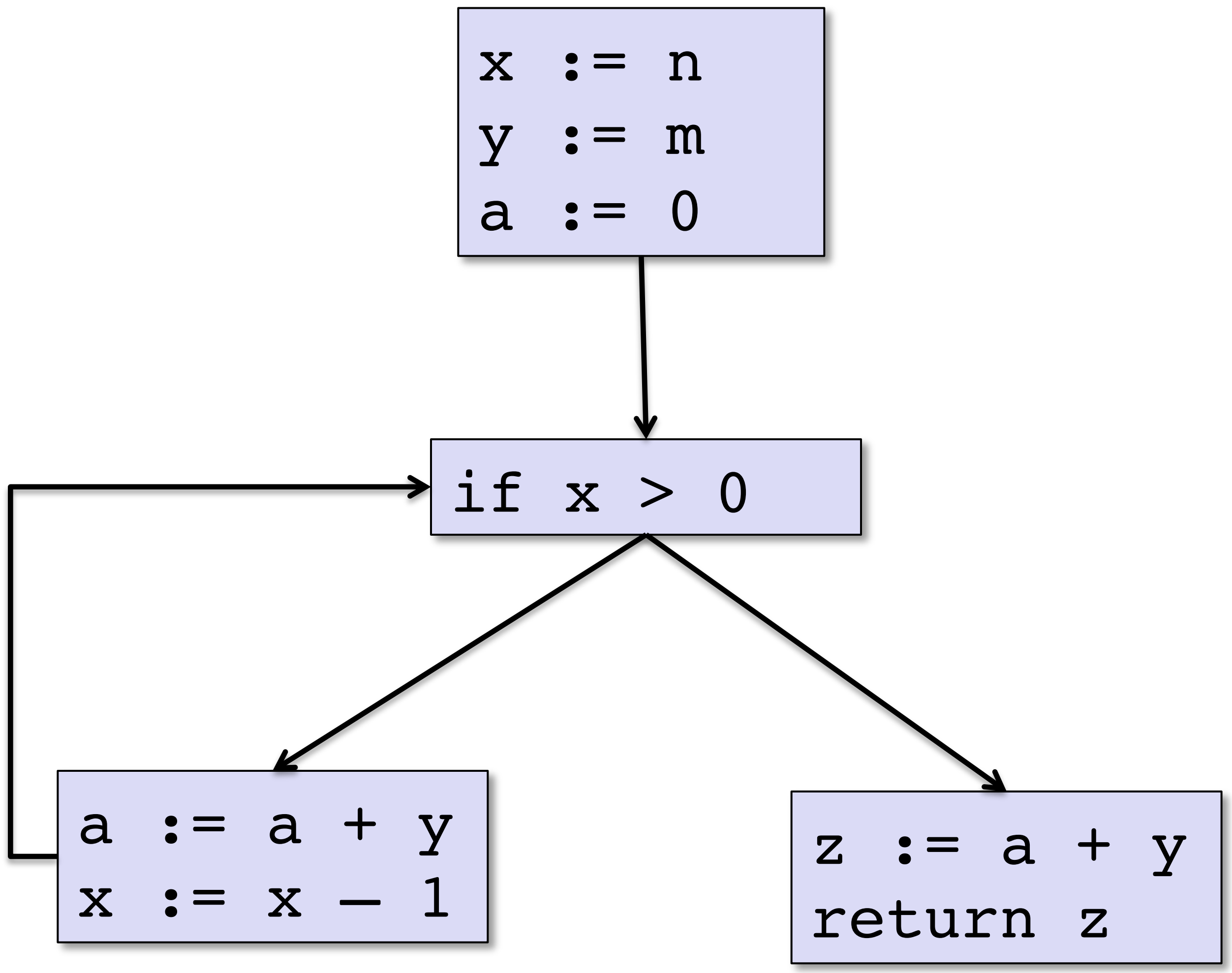
You can always rely upon either a copy propagation pass or a coalescing register allocator to get rid of all of these copies.

But this can blow up the size of the code considerably, so there are better algorithms that try to avoid this.

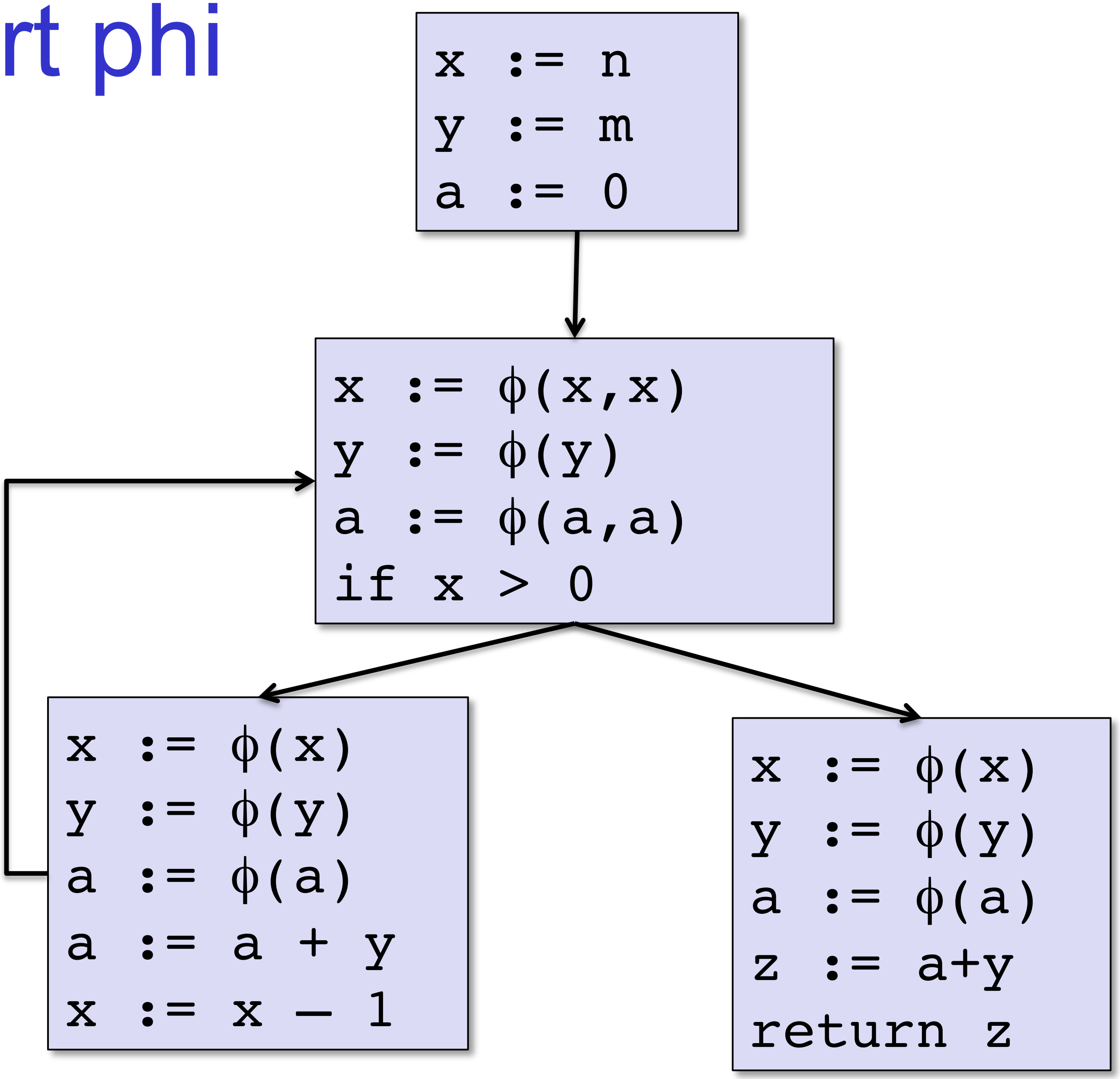
Naïve Conversion to SSA

- Insert phi nodes in each basic block except the start node.
- Calculate the dominator tree.
- Then, traversing the dominator tree in a breadth-first fashion:
  - give each definition of x a fresh index
  - propagate that index to all of the uses
    - each use of x that's not killed by a subsequent definition.
    - propagate the last definition of x to the successors' phi nodes.

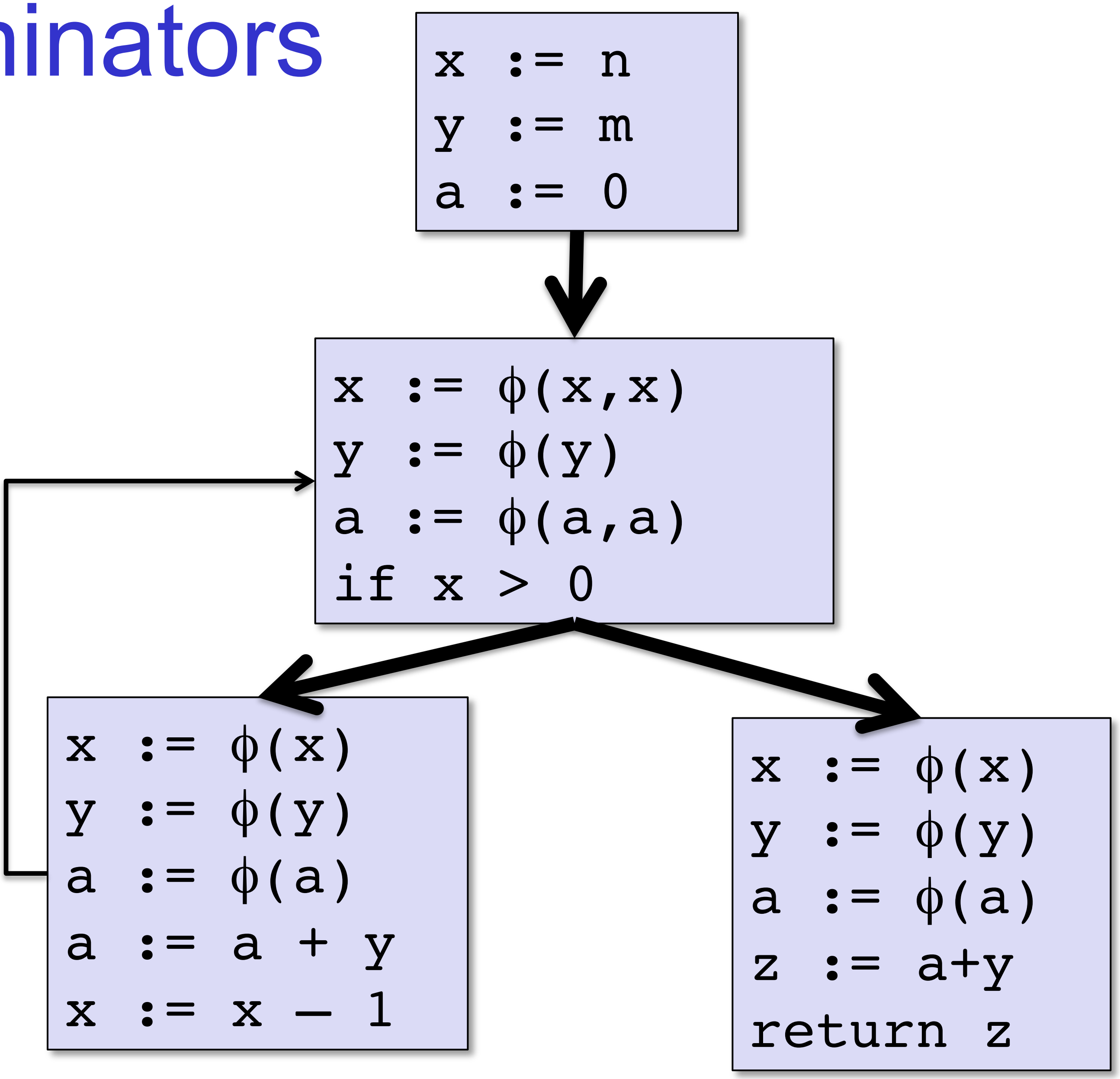
Example:



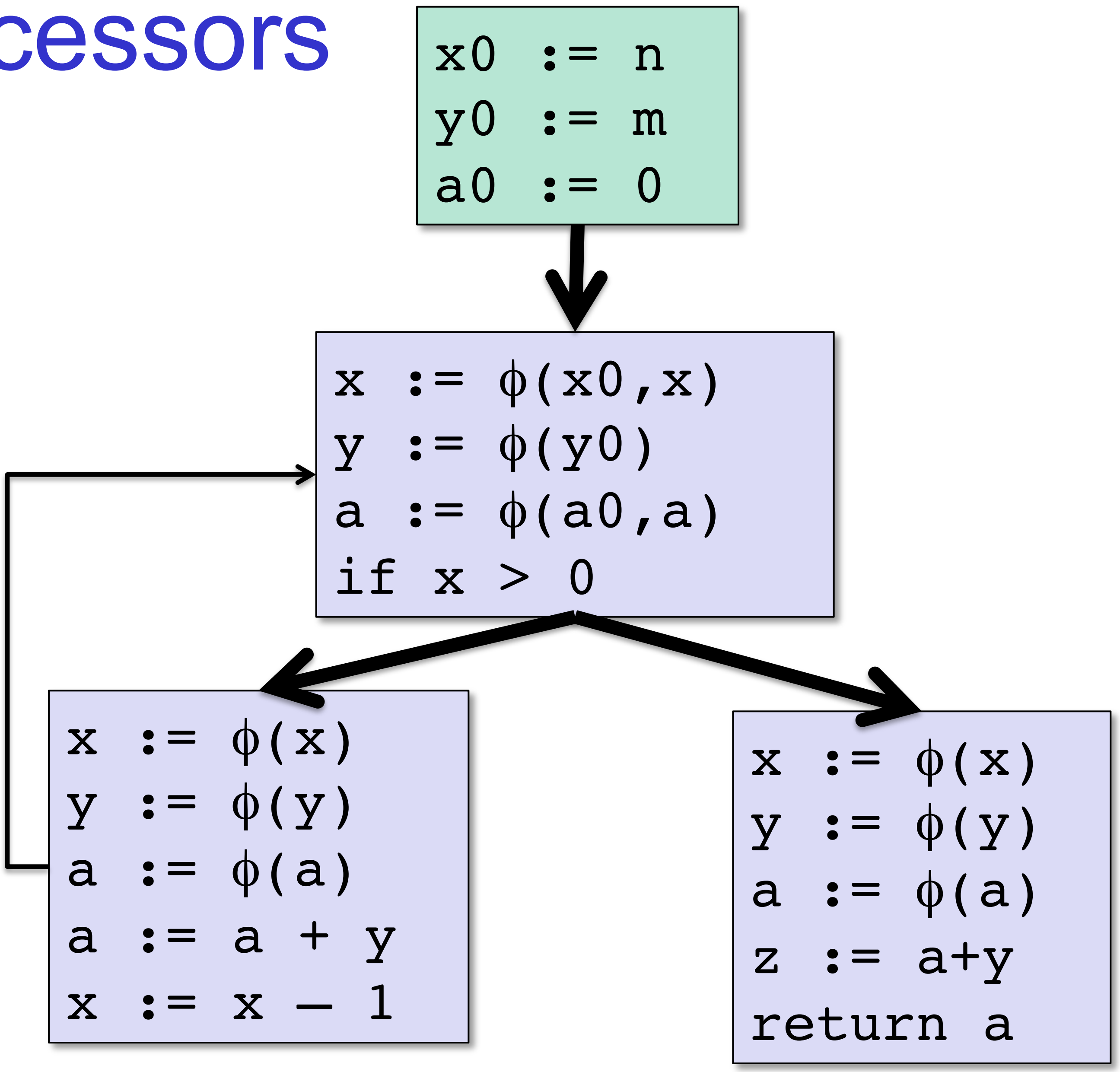
Insert phi


x := n
y := m
a := 0
x := ϕ(x,x)
y := ϕ(y)
a := ϕ(a,a)
if x > 0
x := ϕ(x)
y := ϕ(y)
a := ϕ(a)
a := a + y
x := x – 1
x := ϕ(x)
y := ϕ(y)
a := ϕ(a)
z := a+y
return z

Dominators


x := n
y := m
a := 0
x := ϕ(x,x)
y := ϕ(y)
a := ϕ(a,a)
if x > 0
x := ϕ(x)
y := ϕ(y)
a := ϕ(a)
a := a + y
x := x – 1
x := ϕ(x)
y := ϕ(y)
a := ϕ(a)
z := a+y
return z

Successors



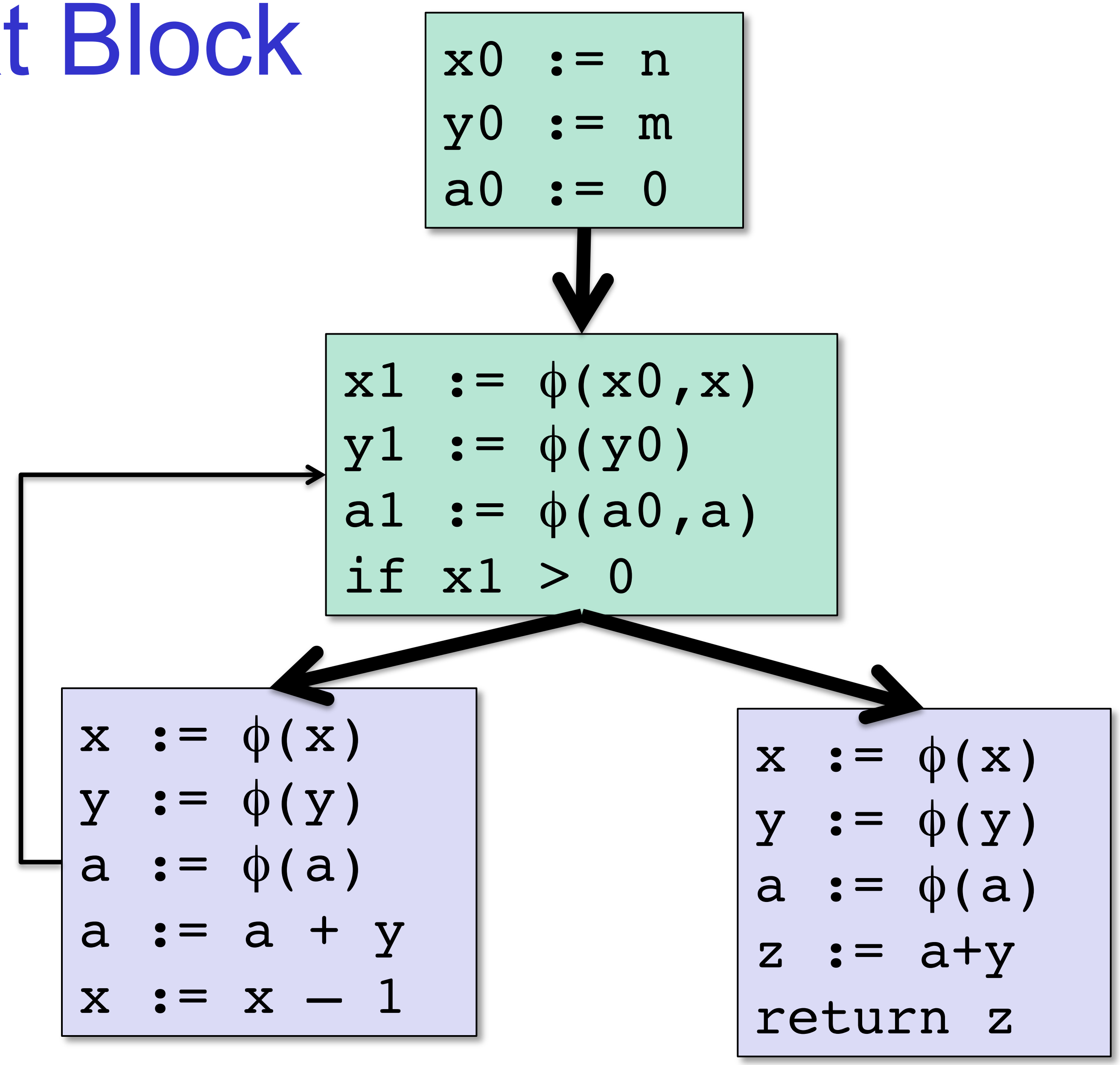
```
x0  :=  n
y0  :=  m
a0  :=  0
```

```
x  := ϕ(x0,x)
y  := ϕ(y0)
a  := ϕ(a0,a)
if x > 0
```

```
x  := ϕ(x)
y  := ϕ(y)
a  := ϕ(a)
a  := a + y
x  := x – 1
```

```
x  := ϕ(x)
y  := ϕ(y)
a  := ϕ(a)
z  := a+y
return a
```

Next Block



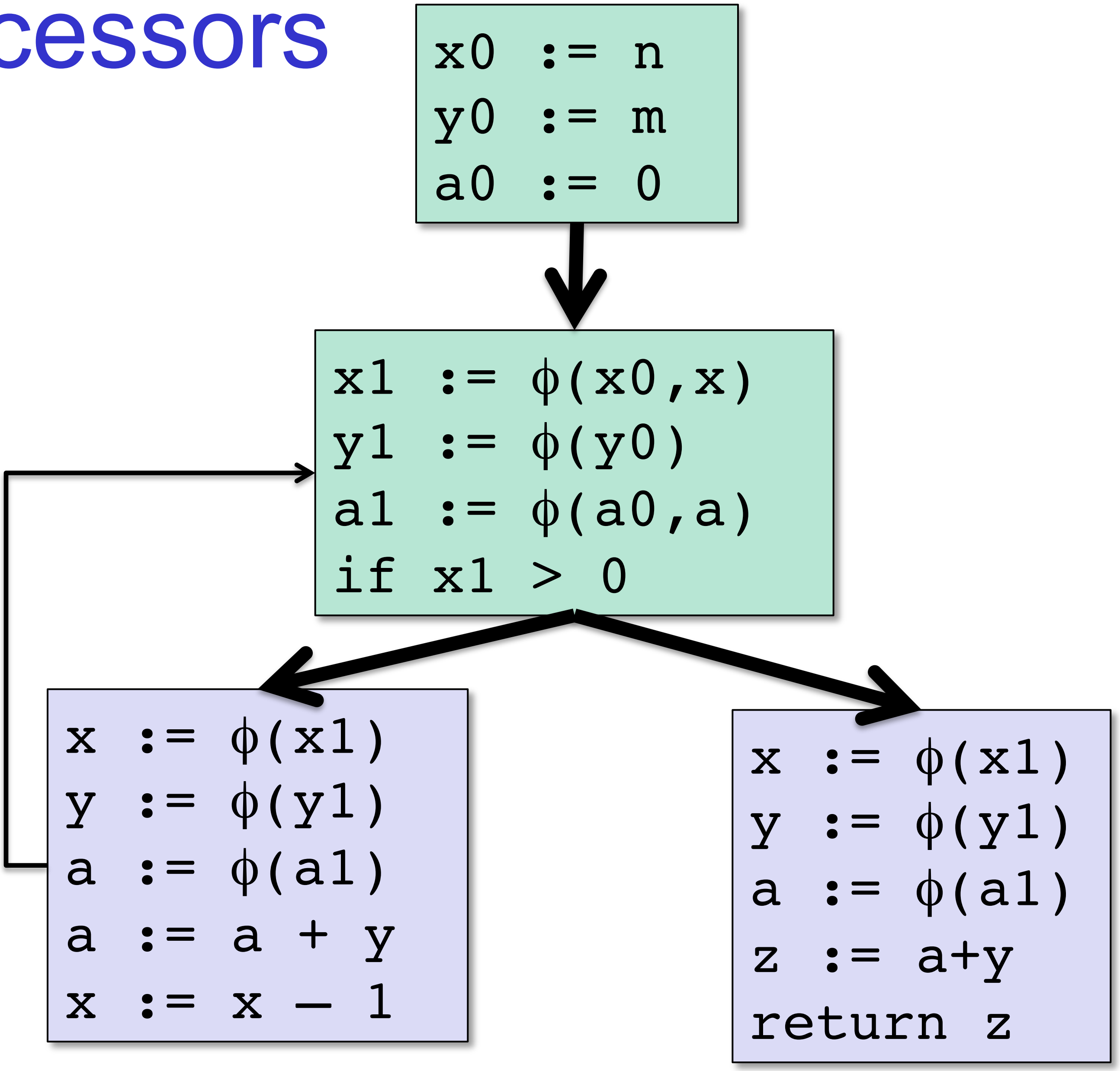
```
x0  := n
y0  := m
a0  := 0
```

```
x1  := ϕ(x0,x)
y1  := ϕ(y0)
a1  := ϕ(a0,a)
if x1 > 0
```

```
x  := ϕ(x)
y  := ϕ(y)
a  := ϕ(a)
a  := a + y
x  := x – 1
```

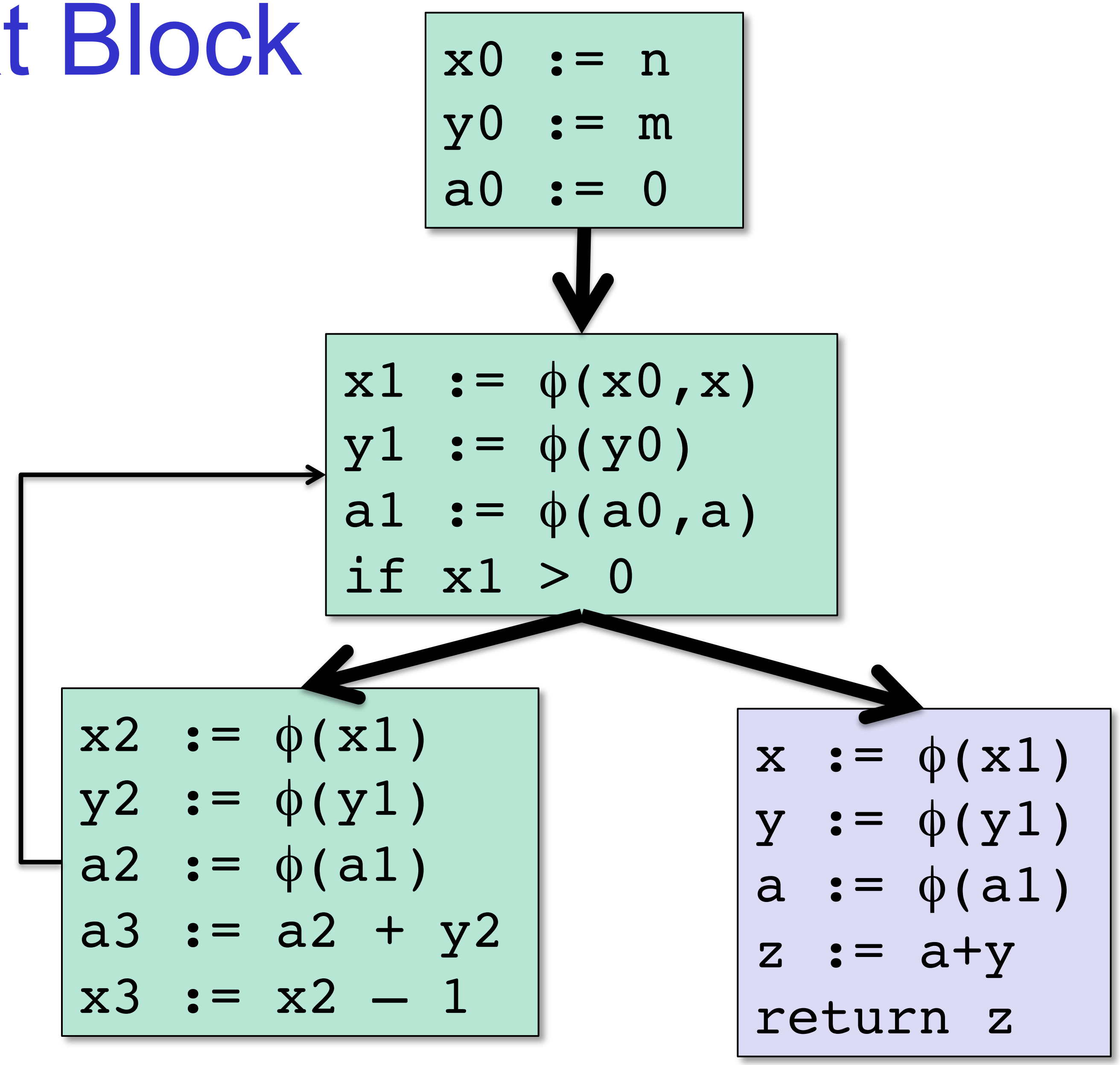
```
x  := ϕ(x)
y  := ϕ(y)
a  := ϕ(a)
z  := a+y
return z
```

Successors



x0 := n
y0 := m
a0 := 0
x1 := ϕ(x0,x)
y1 := ϕ(y0)
a1 := ϕ(a0,a)
if x1 > 0
x := ϕ(x1)
y := ϕ(y1)
a := ϕ(a1)
a := a + y
x := x – 1
x := ϕ(x1)
y := ϕ(y1)
a := ϕ(a1)
z := a+y
return z

Next Block

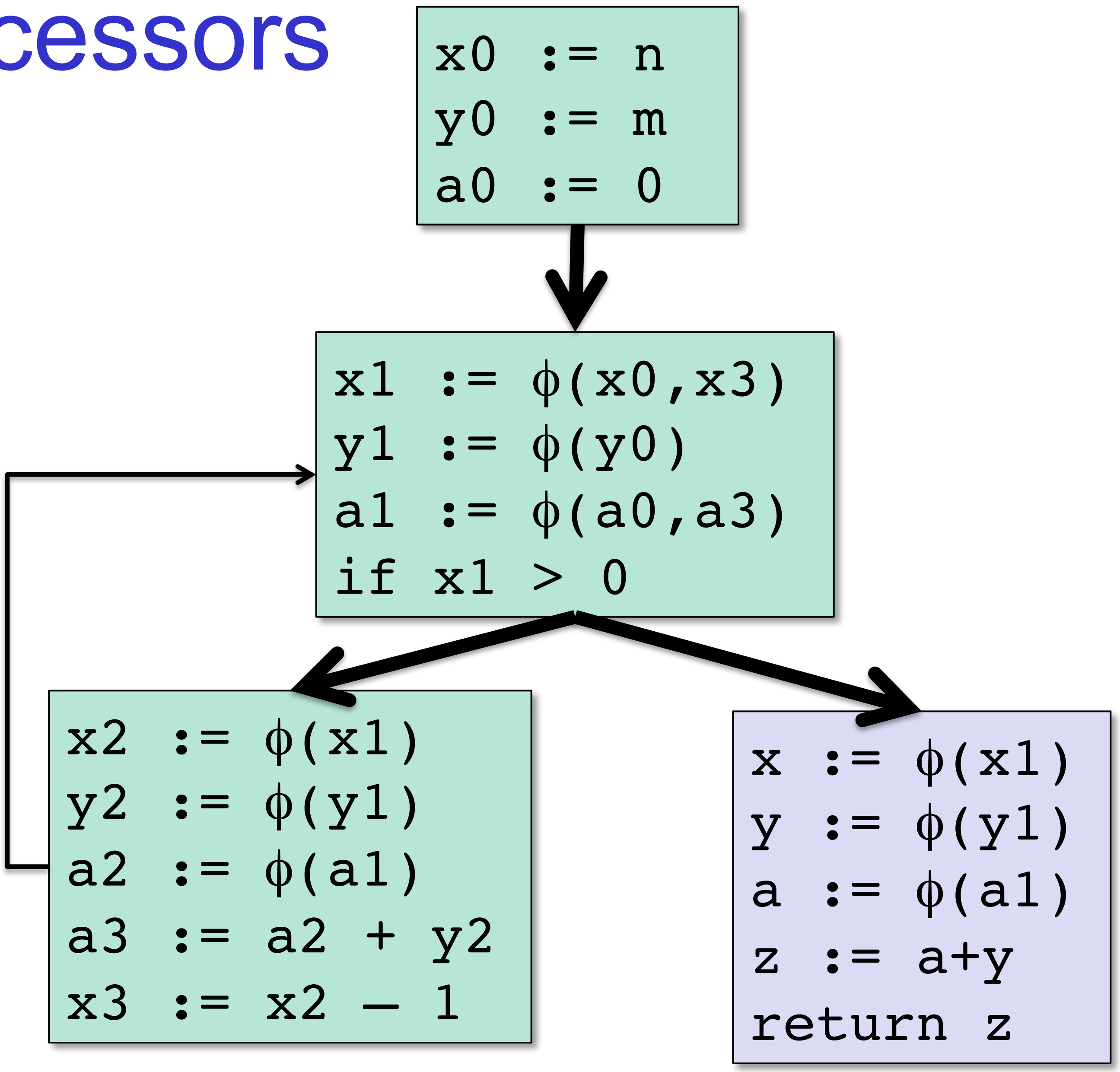

```
x0 := n
y0 := m
a0 := 0
```

```
x1 := ϕ(x0,x)
y1 := ϕ(y0)
a1 := ϕ(a0,a)
if x1 > 0
```

```
x2 := ϕ(x1)
y2 := ϕ(y1)
a2 := ϕ(a1)
a3 := a2 + y2
x3 := x2 – 1
```

```
x := ϕ(x1)
y := ϕ(y1)
a := ϕ(a1)
z := a+y
return z
```

Successors



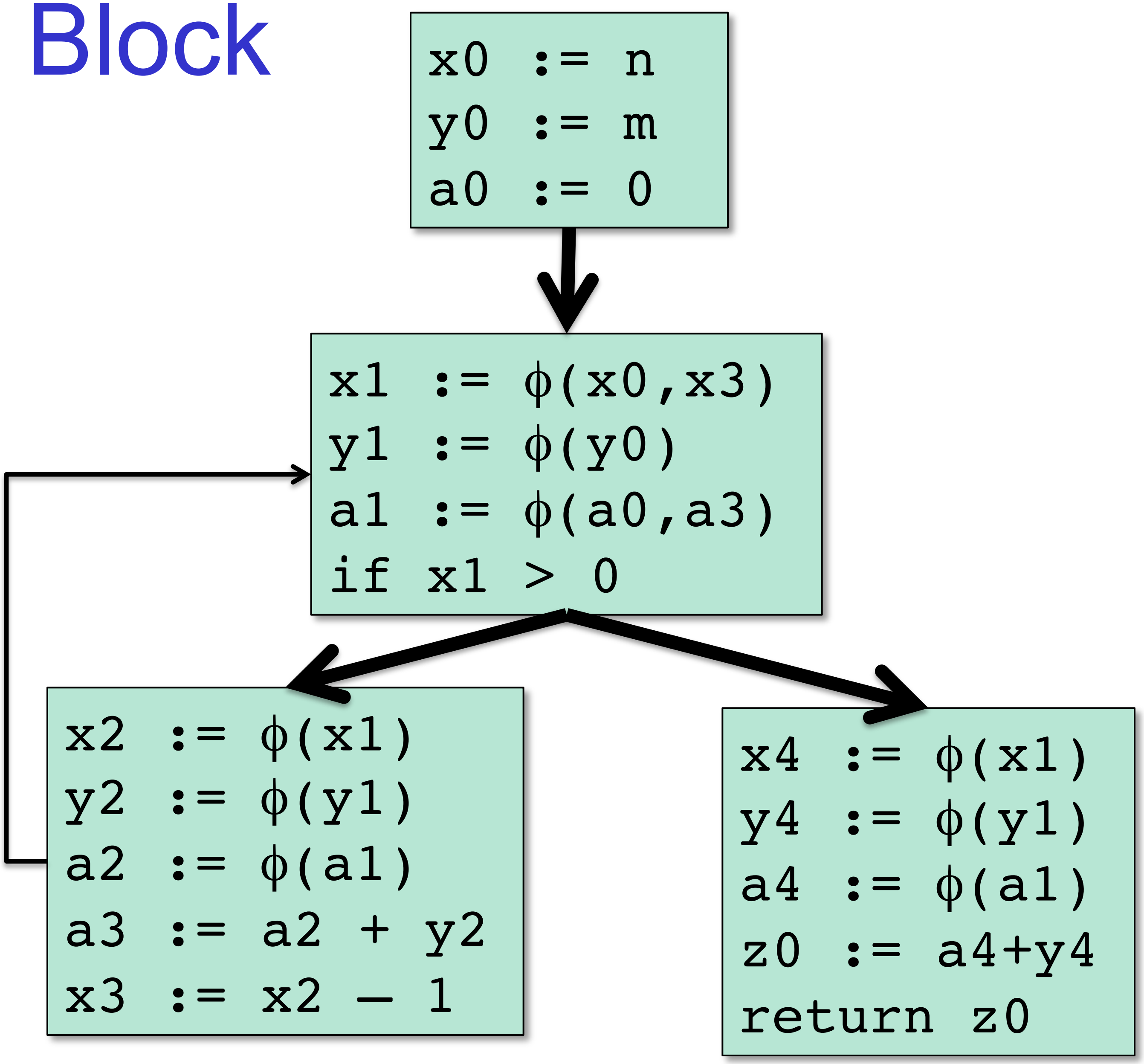
```
x0  := n
y0  := m
a0  := 0
```

```
x1  := ϕ(x0,x3)
y1  := ϕ(y0)
a1  := ϕ(a0,a3)
if x1 > 0
```

```
x2  := ϕ(x1)
y2  := ϕ(y1)
a2  := ϕ(a1)
a3  := a2 + y2
x3  := x2 – 1
```

```
x  := ϕ(x1)
y  := ϕ(y1)
a  := ϕ(a1)
z  := a+y
return z
```

Last Block



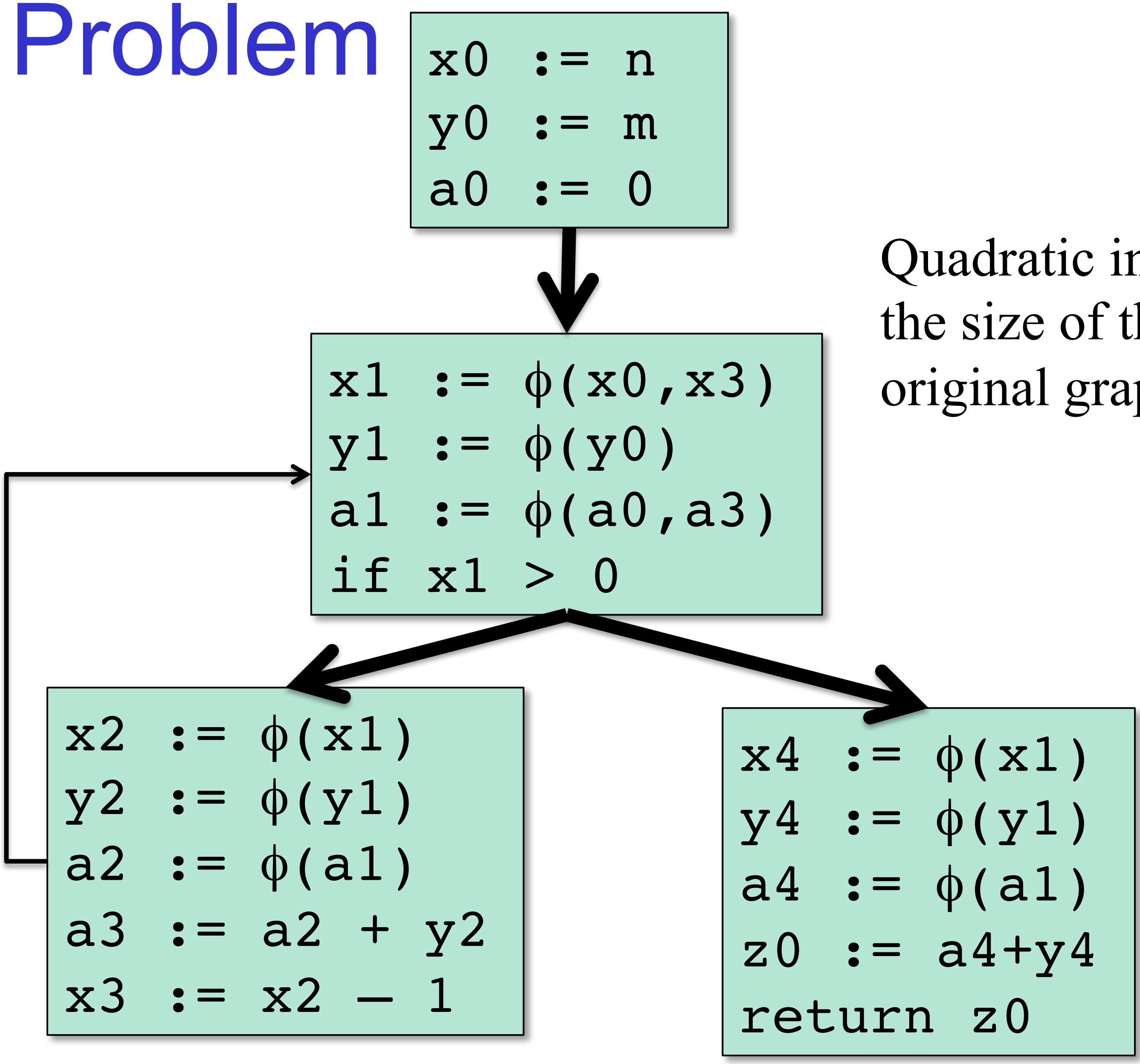
```
x0  := n
y0  := m
a0  := 0
```

```
x1  := ϕ(x0,x3)
y1  := ϕ(y0)
a1  := ϕ(a0,a3)
if x1 > 0
```

```
x2  := ϕ(x1)
y2  := ϕ(y1)
a2  := ϕ(a1)
a3  := a2 + y2
x3  := x2 – 1
```

```
x4  := ϕ(x1)
y4  := ϕ(y1)
a4  := ϕ(a1)
z0  := a4+y4
return z0
```

Key Problem



```
x0  := n
y0  := m
a0  := 0
```

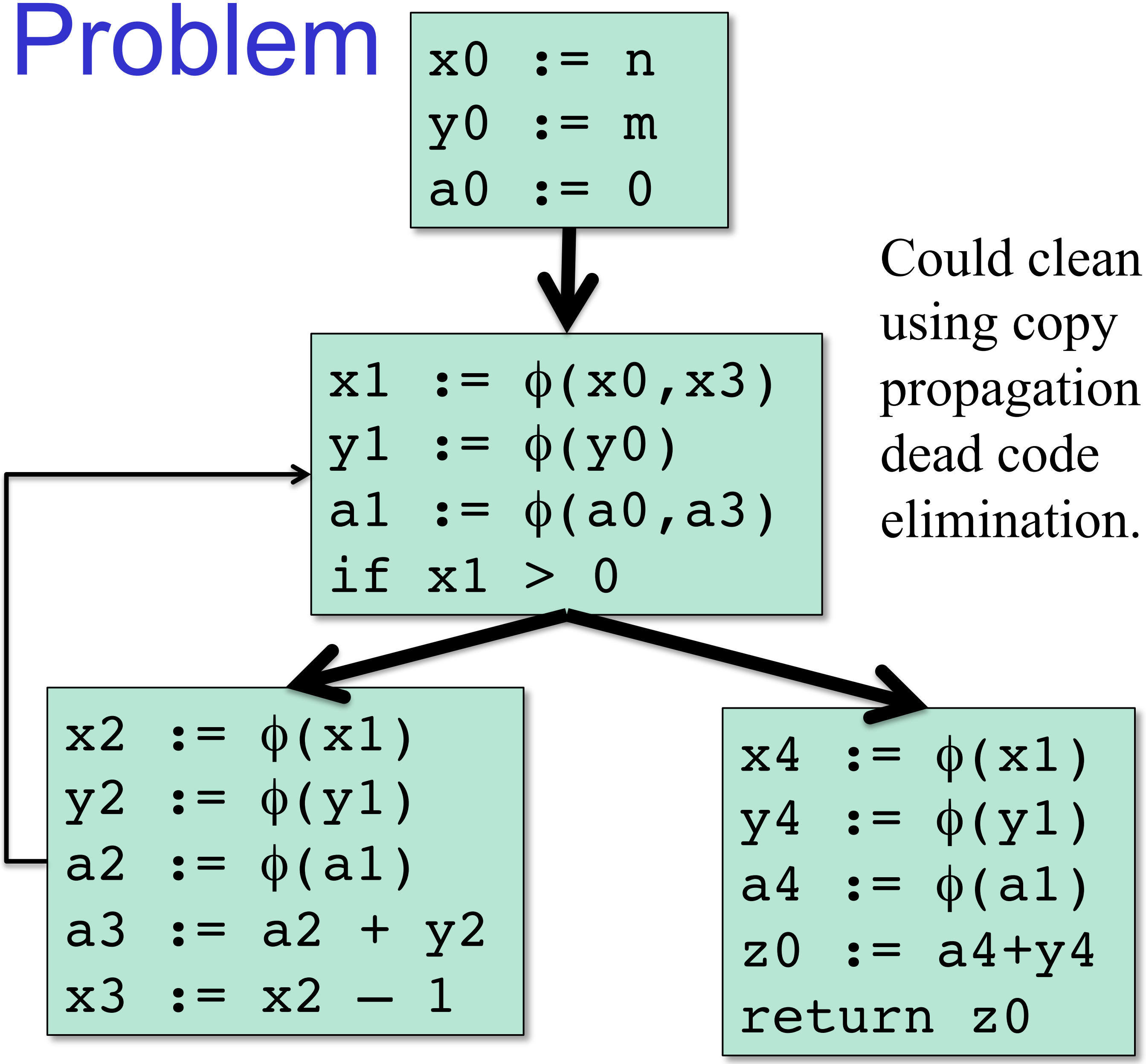
```
x1  := ϕ(x0,x3)
y1  := ϕ(y0)
a1  := ϕ(a0,a3)
if x1 > 0
```

```
x2  := ϕ(x1)
y2  := ϕ(y1)
a2  := ϕ(a1)
a3  := a2 + y2
x3  := x2 – 1
```

```
x4  := ϕ(x1)
y4  := ϕ(y1)
a4  := ϕ(a1)
z0  := a4+y4
return z0
```

Quadratic in the size of the original graph!

Key Problem



```
x0 := n
y0 := m
a0 := 0
```

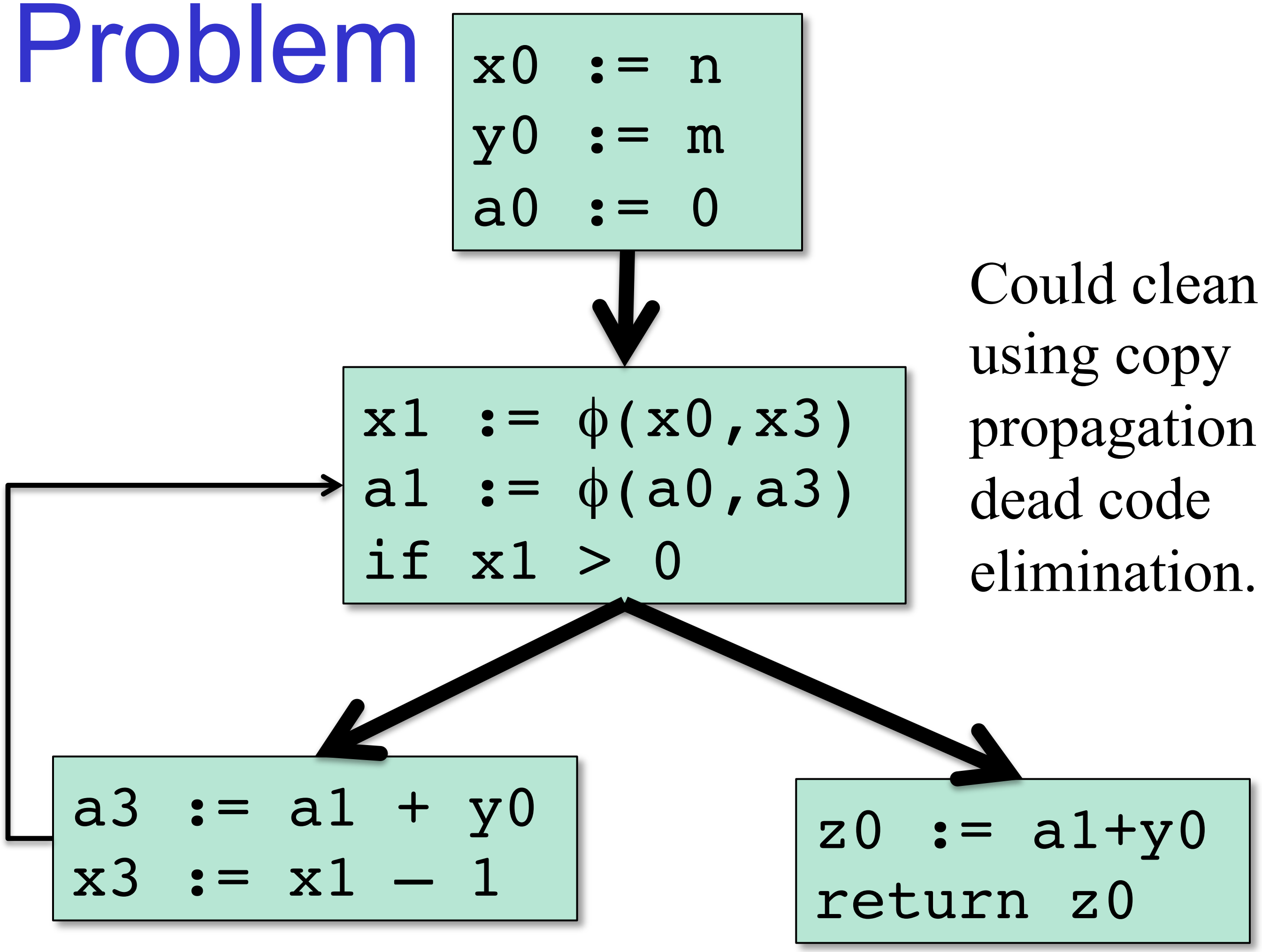
```
x1 := ϕ(x0,x3)
y1 := ϕ(y0)
a1 := ϕ(a0,a3)
if x1 > 0
```

```
x2 := ϕ(x1)
y2 := ϕ(y1)
a2 := ϕ(a1)
a3 := a2 + y2
x3 := x2 – 1
```

```
x4 := ϕ(x1)
y4 := ϕ(y1)
a4 := ϕ(a1)
z0 := a4+y4
return z0
```

Could clean up using copy propagation and dead code elimination.

Key Problem



```
x0 := n
y0 := m
a0 := 0
```

```
x1 := ϕ(x0,x3)
a1 := ϕ(a0,a3)
if x1 > 0
```

```
a3 := a1 + y0
x3 := x1 – 1
```

```
z0 := a1+y0
return z0
```

Could clean up using copy propagation and dead code elimination.

Smarter Algorithm

- Compute the *dominance frontier.*
- Use dominance frontier to place the phi nodes.
  - If a block B defines x then put a phi node in every block in the dominance frontier of B.
- Do renaming pass using dominator tree.

This isn't optimal but in practice, produces code that's linear in the size of the input and is efficient to compute.

Dominance Frontiers

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

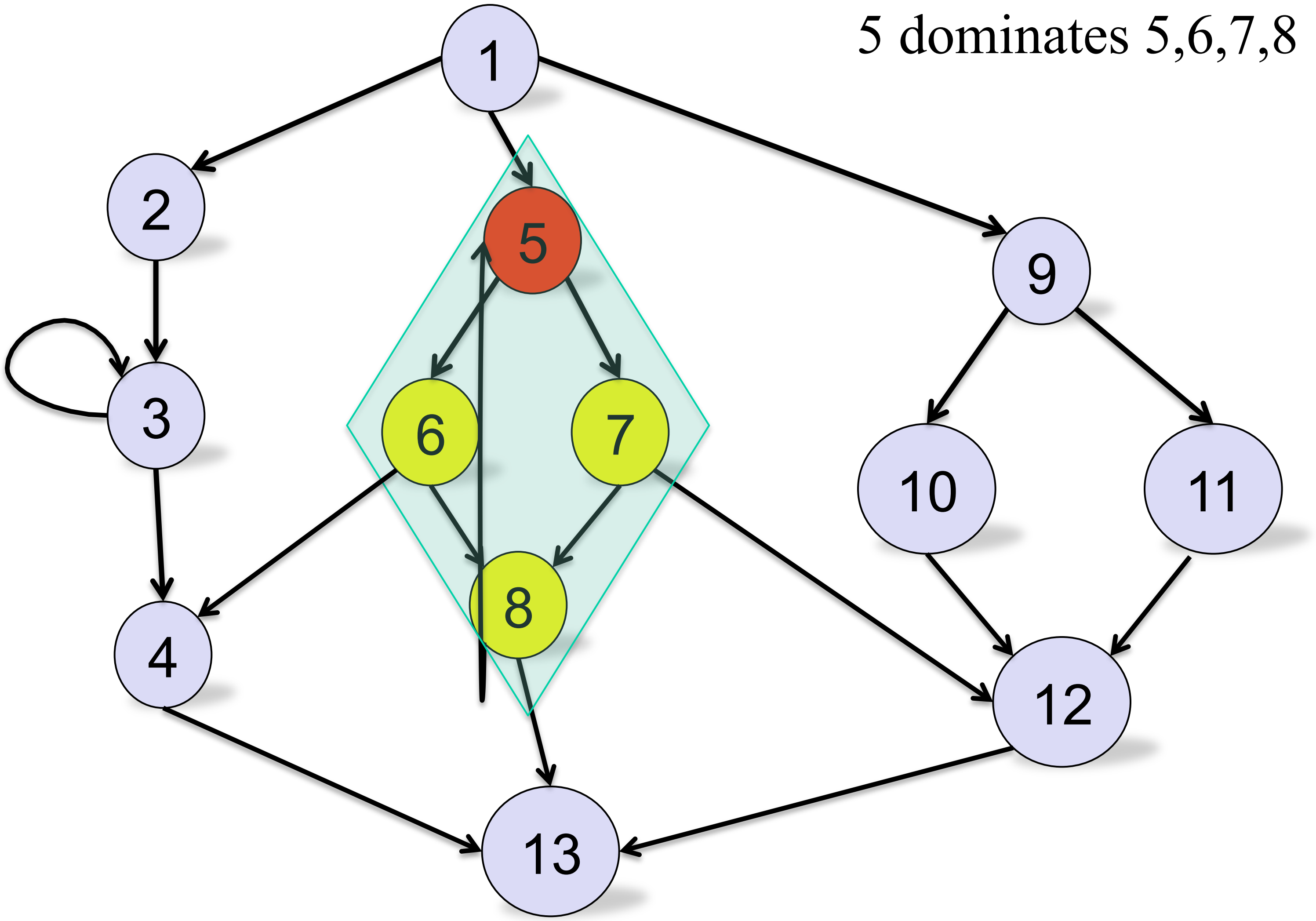
Defn: If d dominates n and d ≠ n, we say d *strictly dominates* n.

Defn: the *dominance frontier* of x is the set of all nodes w such that:

1. x dominates a predecessor of w
2. x does not strictly dominate w.

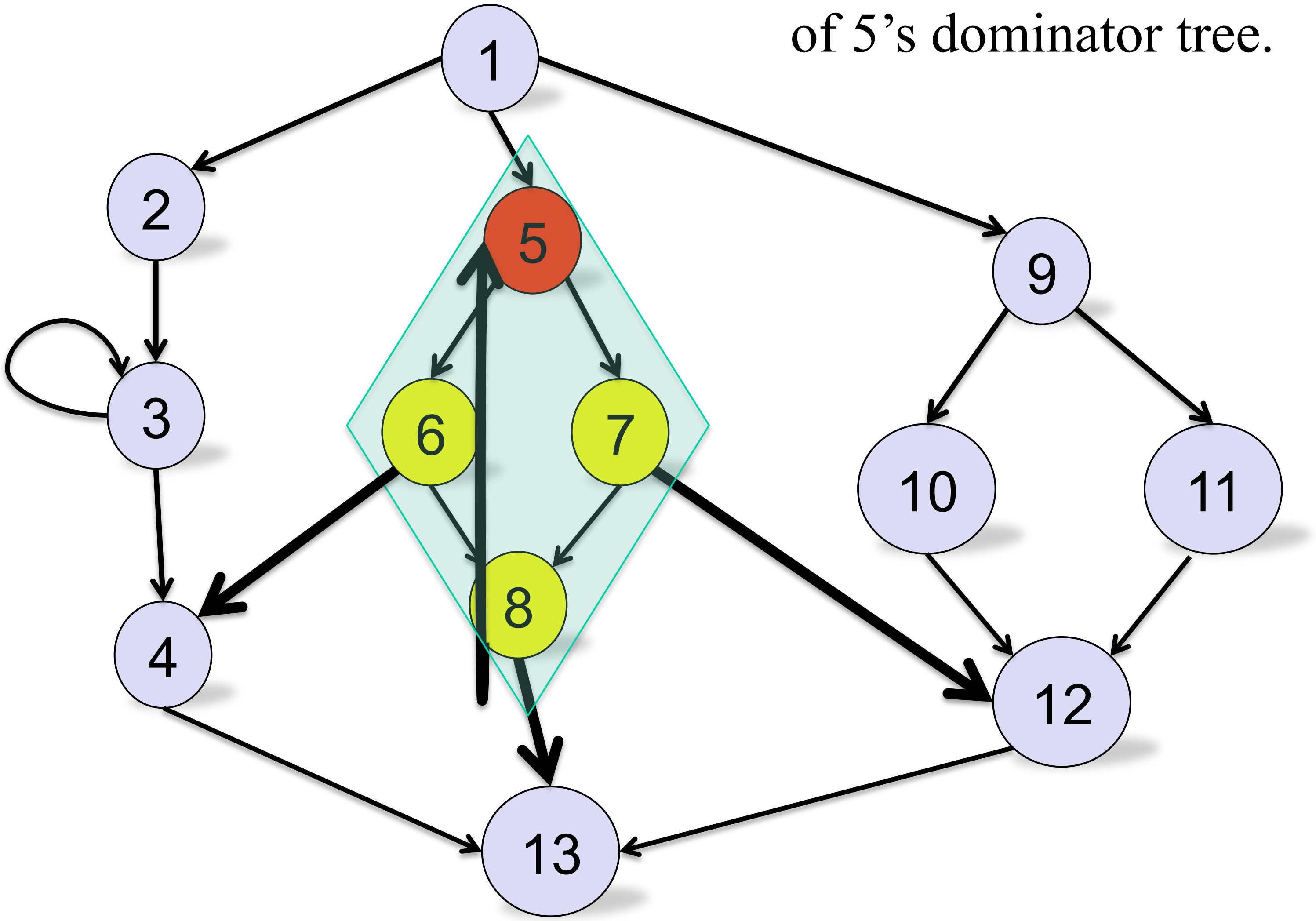
Example (Fig 19.5)

5 dominates 5,6,7,8



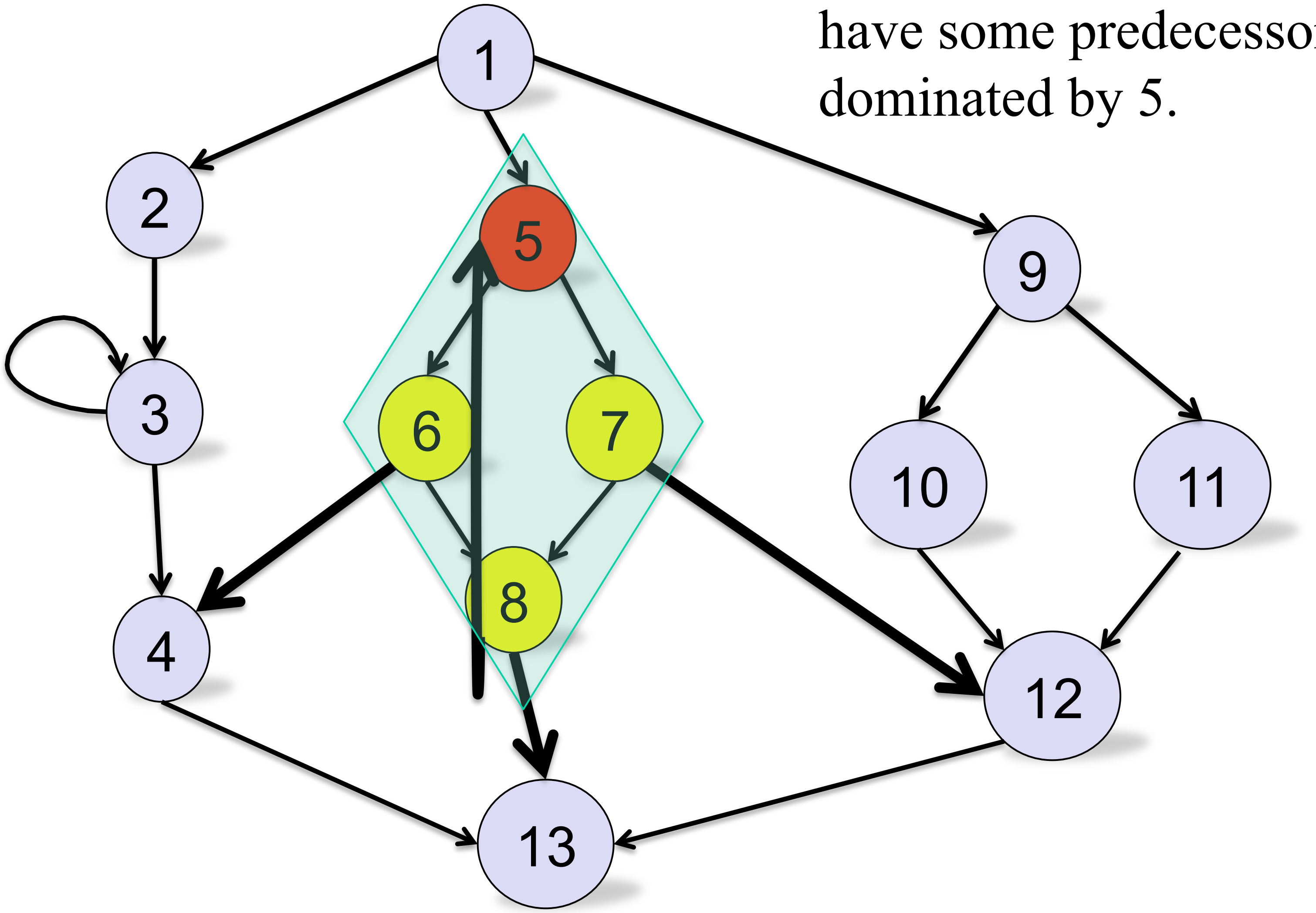
Example (Fig 19.5)

These are edges that cross from the frontier of 5's dominator tree.



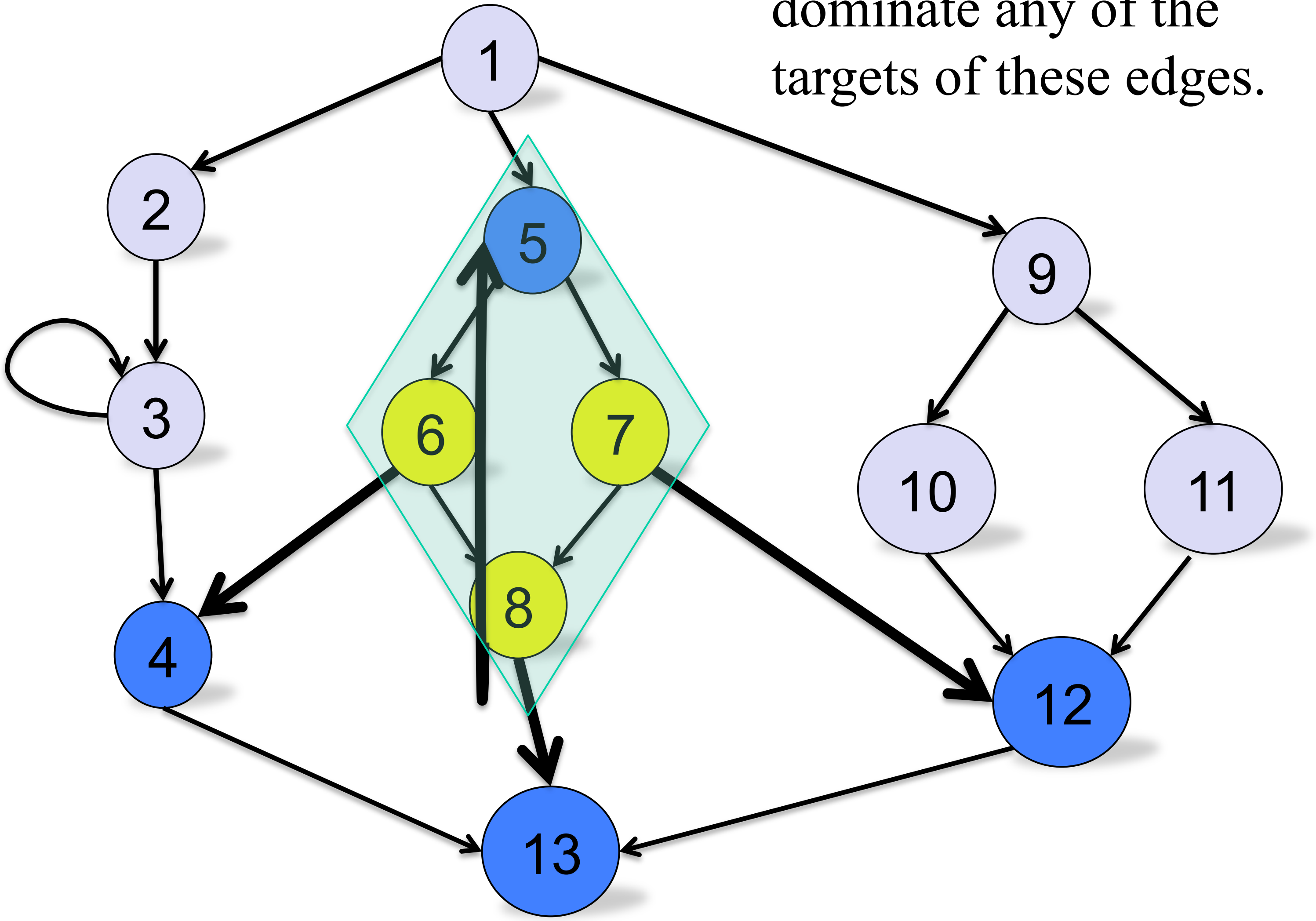
Example (Fig 19.5)

This identifies nodes that satisfy the first condition: nodes that have some predecessor dominated by 5.



Example (Fig 19.5)

5 does not *strictly* dominate any of the targets of these edges.



Computing Dominance Frontiers

local[n]: successors of n not strictly dominated by n.

up[n]: nodes in dominance frontier of n that are not strictly dominated by n’s immediate dominator.

DF[n] = local[n] U { up[c] | c in children[n] }

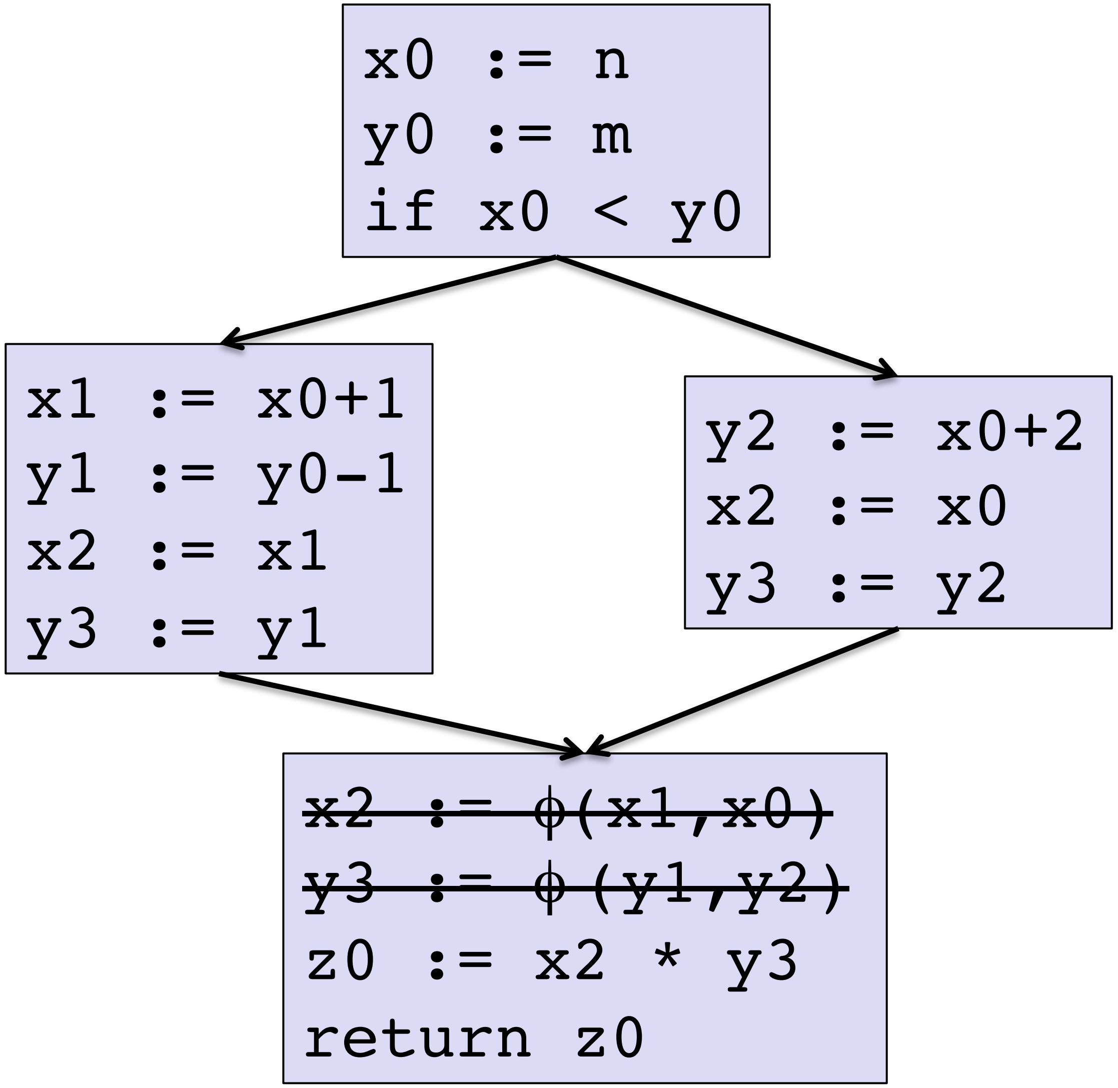
Algorithm

```
computeDF[n] =
  S := {}
  for each y in succ[n]     (* compute local[n] *)
    if immediate_dominator(y) ≠ n
        S := S U {y}
  for each child c of n in dominator tree
    computeDF[c]
    for each w in DF[c]  (* compute up[c] *)
      if n does not dominate w or n = w
        S := S U {w}
  DF[n] := S
```

A few notes

- Algorithm does work proportional to number of edges in control flow graph + size of the dominance frontiers.
  - pathological cases can lead to quadratic behavior.
  - in practice, linear
- All depends upon computing dominator tree.
  - iterative dataflow algorithm is cubic in worst case.
  - but Lengauer & Tarjan give an essentially linear time algorithm.

Recall



```
x0 := n
y0 := m
if x0 < y0
```

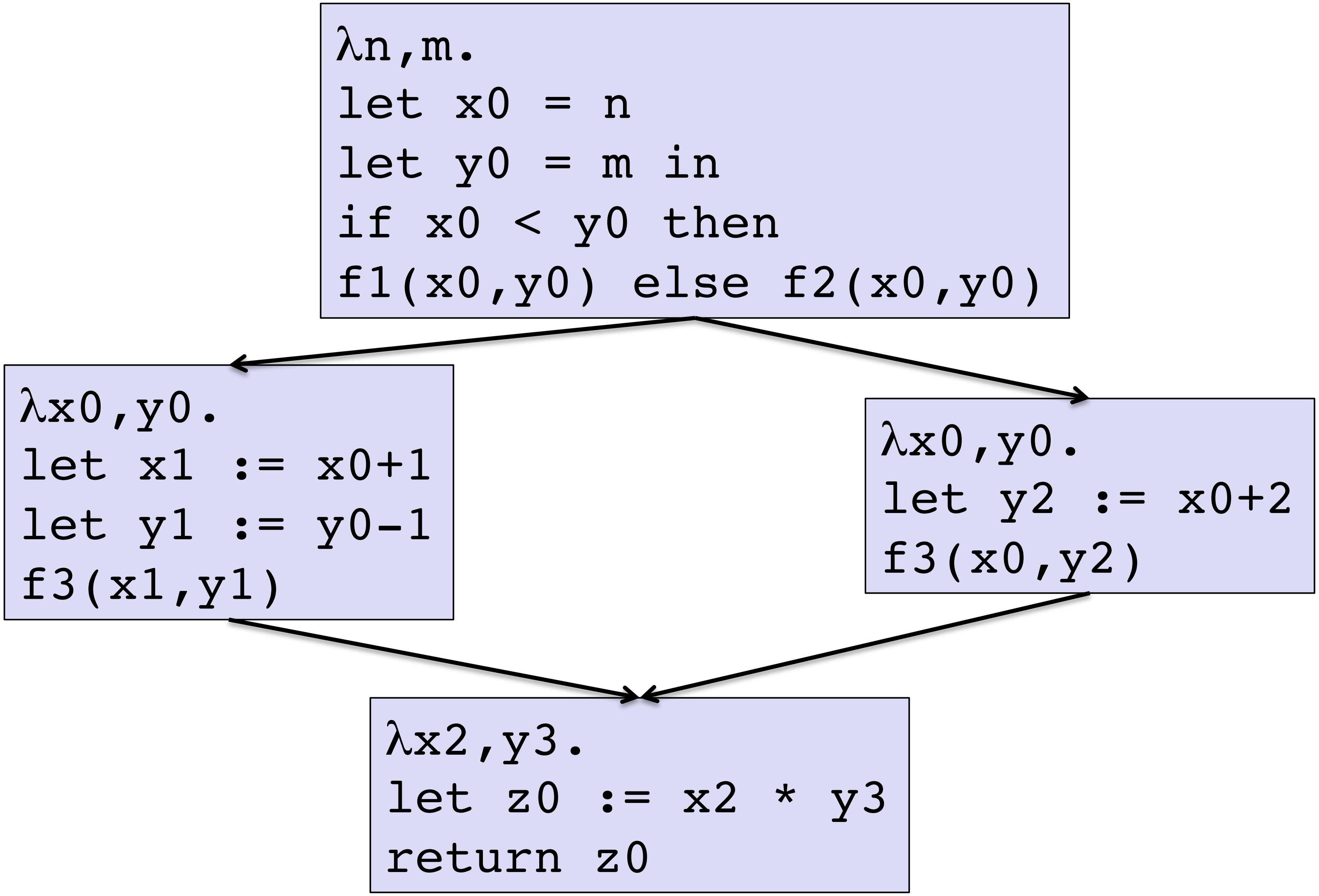
```
x1 := x0+1
y1 := y0-1
x2 := x1
y3 := y1
```

```
y2 := x0+2
x2 := x0
y3 := y2
```

```
~~x2 := ϕ(x1,x0)~~
~~y3 := ϕ (y1,y2)~~
z0 := x2 * y3
return z0
```

Just realize the assignments corresponding to the phi nodes on the edges.

CPS



```
λn,m.
let x0 = n
let y0 = m in
if x0 < y0 then
f1(x0,y0) else f2(x0,y0)
```

```
λx0,y0.
let x1 := x0+1
let y1 := y0-1
f3(x1,y1)
```

```
λx0,y0.
let y2 := x0+2
f3(x0,y2)
```

```
λx2,y3.
let z0 := x2 * y3
return z0
```

CPS less compact than SSA

- Can always encode SSA.
- But requires us to match up a block's output variables to its successor's input variables: f(v1,…x…,vn) → λ x1,…x…,xn.
- It's possible to avoid some of this threading, but not as much as in SSA.
  - Worst case is again quadratic in the size of the code.
  - CPS: tree-based scope for variables
  - SSA: graph-based scope for variables

CPS more powerful than SSA

- On the other hand, CPS supports *dynamic* control flow graphs.
  - e.g., “goto x” where x is a variable, not a static label name.
- That makes it possible to encode strictly more than SSA can.
  - return addresses (no need for special return instruction – just goto return address.)
  - exceptions, function calls, loops, etc. all turn into just “goto f(x1,…,xn)”.

Core CPS language

```
op ::= x | true | false | i | …

v := op | λx₁,…,xₙ.exp
    | prim(op₁,…,opₙ)

exp ::=
   op(op₁,...,opₙ)
 | if cond(op₁,op₂) exp₁ else exp₂
 | let x = v in exp
 | letrec x₁ = v₁,…, xₙ = vₙ in exp
```

CPS

```
let f3 = λx2,y3.
          let z0 := x2 * y3 in
          return z0
```

```
let f1 = λx0,y0.
          let x1 := x0+1 in
          let y1 := y0-1 in
          f3(x1,y1)
```

```
let f2 = λx0,y0.
          let y2 := x0+2 in
          f3(x0,y2)
```

```
let x0 = n in
let y0 = m in
if x0 < y0 then
f1(x0,y0) else f2(x0,y0)
```

Dataflow SSA/CFG vs CPS

- To solve dataflow equations, for CFG or SSA, we iterate over the control flow graph.

- But for CPS, we don't know what the graph is (in general) at compile time.
  - our "successors" and "predecessors" depend upon which values flow into the variables wt jump to.
  - To figure out which functions we might jump to, we need to do dataflow analysis…
  - Oops!

Simultaneous Solution: CFA

- In general, we must simultaneously solve for (an approximation of) the dynamic control flow graph, and the set of values that a variable might take on.
- This is called control-flow analysis (CFA).
- The good news: if you solve this, then you don't need lots of special cases in your dataflow analyses (e.g., for function calls/returns, dynamic method resolution, exceptions, threads, etc.)
- The bad news: must use very coarse approximations to scale.

Final Exam

- Final: April 19th, 10:30-12:30, Snell Library 045
- The final exam will be cumulative
- You may be tested on anything covered in readings or lectures, except for Typed Assembly Language.
- Assigned reading from Appel:
  - Ch 1, 2, 3 (intro, lexing, parsing)
  - Ch 13 (garbage collection)
  - Ch 14 (OO languages)
  - Ch 8 (basic blocks)
  - Ch 17 (dataflow analysis)
  - Ch 11 (register allocation)
  - Ch 18 (loop optimizations)
  - Ch 19 (SSA), skip 19.2