HARVARD
John A. Paulson
School of Engineering
and Applied Sciences

CS153: Compilers
Lecture 23:
Static Single Assignment Form

Stephen Chong

https://www.seas.harvard.edu/courses/cs153

Pre-class Puzzle

- Suppose we want to compute an analysis over CFGs. We have two possible algorithms.

  Algorithm A is simple but has worst-case $O(N^2)$
  where a CFG has N nodes and E edges

  Algorithm B is more complicated but has worst-case complexity $O(N + \log(E))$

  Which algorithm should we use? Why?

Announcements

- Project 6 due today
- Project 7 out
  - Due Thursday Nov 29 (9 days)
- Project 8 out
  - Due Saturday Dec 8 (18 days)
- Final exam: Wed December 12, 9am-12pm, Emerson 305
  - Covers everything except guest lectures
    - Lec 1-21, 23, 24, and all projects are fair game!
  - 30 multiple choice questions
  - Open book, open note, open laptop
  - No internet (except to look up notes, etc.),
    - No looking up answers, no communicating with anyone

Today

- Static Single Assignment form
  - What and why
  - SSA to CFG
  - CFG to SSA
    - Dominance frontiers
  - Optimization algorithms using SSA

Pure vs Imperative

- Consider CFG available expression analysis

Stmt	Gen	Kill
`x:=v`	{ v }	{e | x in e}

- If variables are immutable (i.e., are assigned exactly once) analysis simplifies!

Stmt	Gen	Kill
`x:=v`	{ v }	

  - Empty kill set!

Pure vs. Imperative

- 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 imperative form (i.e., CFGs) allowed us to have control-flow graphs, not just trees

Best of Both Worlds

- Static Single Assignment (SSA)
  - CFGs but with immutable variables
  - Plus a slight “hack” to make graphs work out
  - Now widely used (e.g., LLVM)
  - Intra-procedural representation only
    - An SSA representation for whole program is possible (i.e., each global variable and memory location has static single assignment), but difficult to compute

Idea Behind SSA

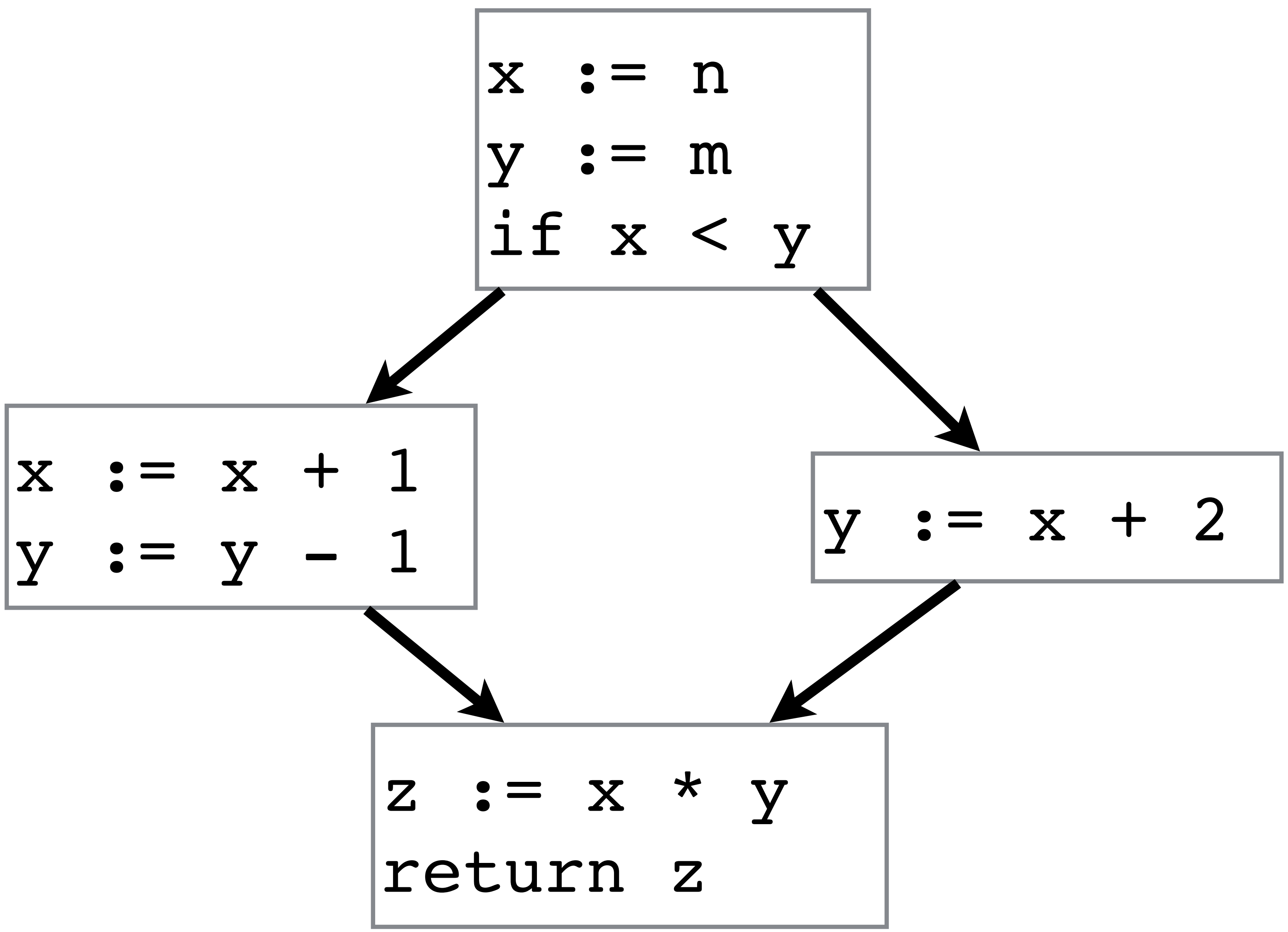
- Start with CFG code
- Give each definition a fresh name
- Propagate fresh name to subsequent uses

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

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

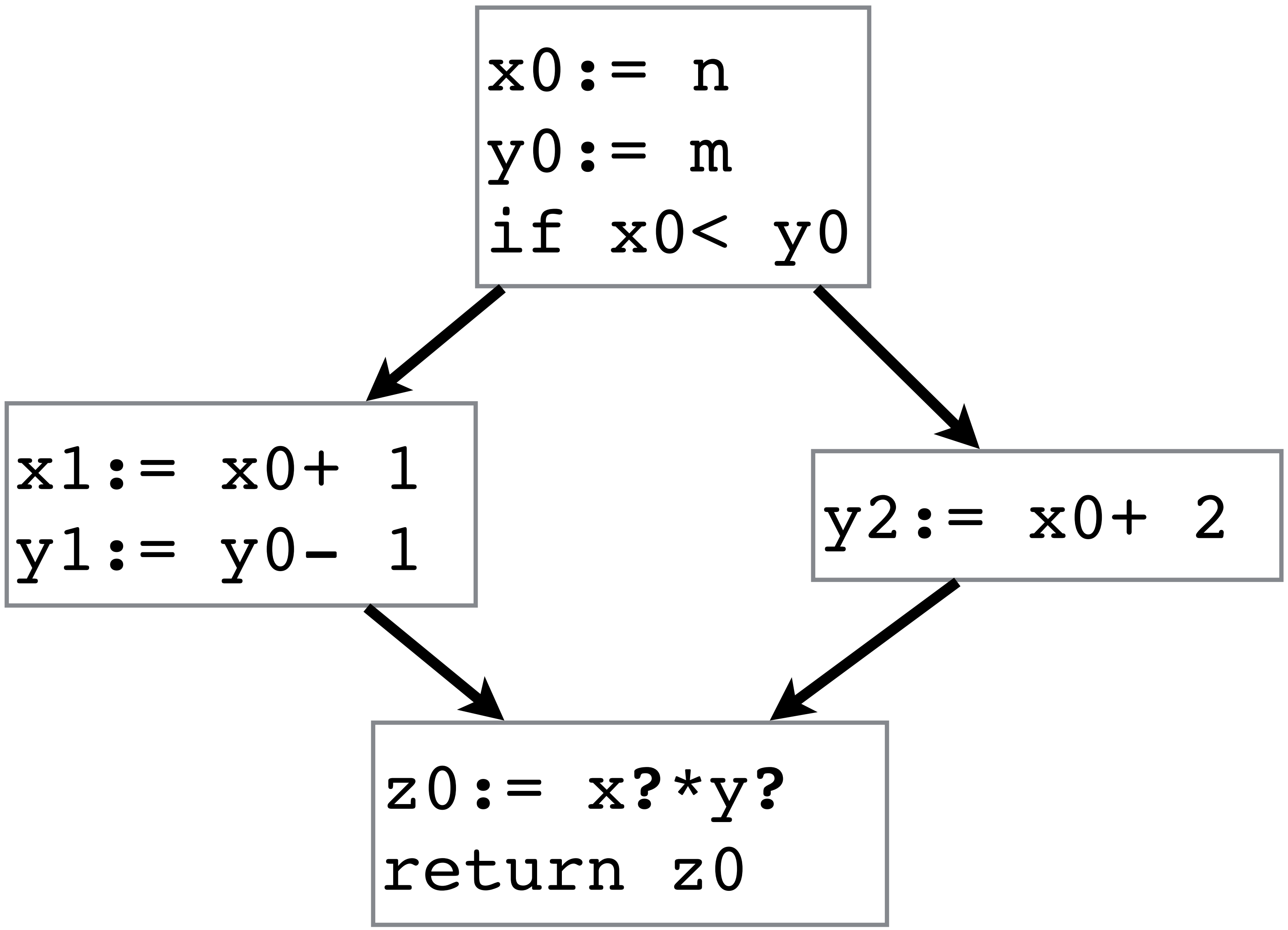
The Problem...

- What about control flow merges?



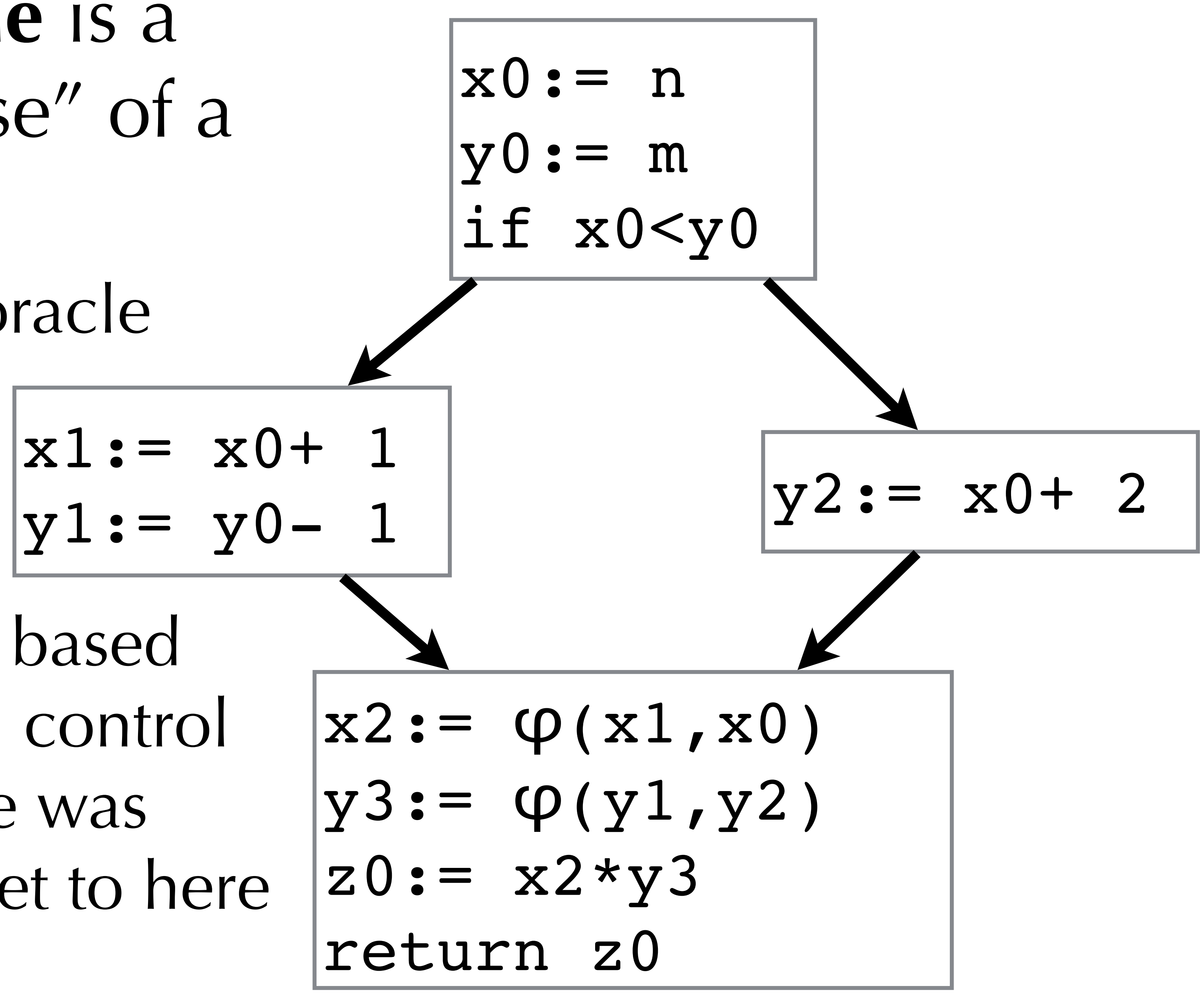
The Problem...

- What about control flow merges?



The Solution

- Insert "phony" expressions for the merge
- A **phi node** is a phony "use" of a variable
  - As if an oracle chooses to set **x2** to either **x0** or **x1** based on which control flow edge was used to get to here

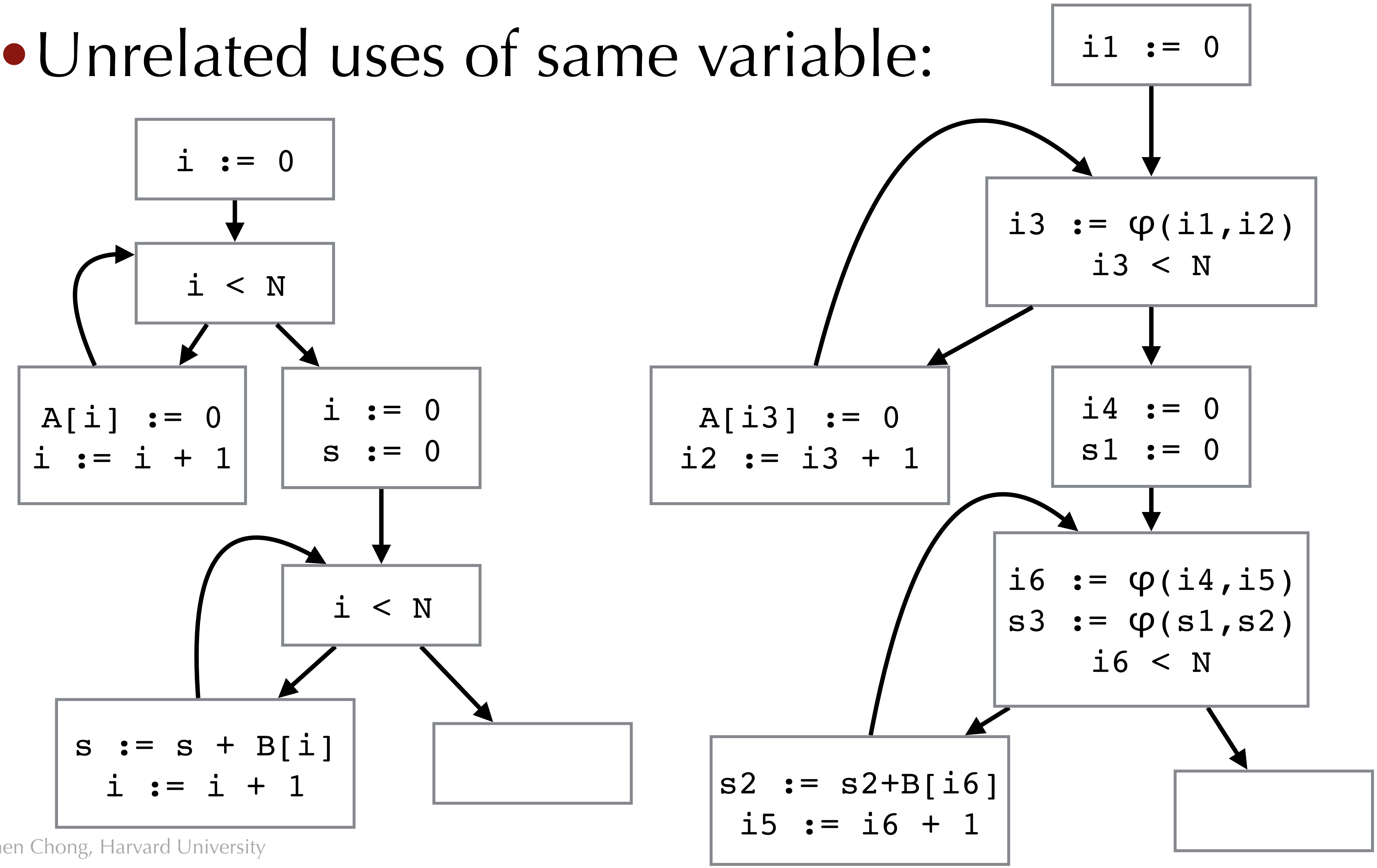


Wait, Remind Me Why Is This Useful

- Data-flow analysis and optimizations become simpler if each variable has 1 definition
- Compilers often build def-use chains
  - Connects definitions of variables with uses of them
  - Propagate dataflow facts directly from defs to uses, rather than through control flow graph
  - In SSA form, def-use chains are linear in size of original program; in non-SSA form may be quadratic
- Is relationship between SSA form and dominator structure of CFG
  - Simplifies algs such as interference graph construction
  - More info soon....
- Unrelated uses of same variable becomes different variables

Example

- Unrelated uses of same variable:

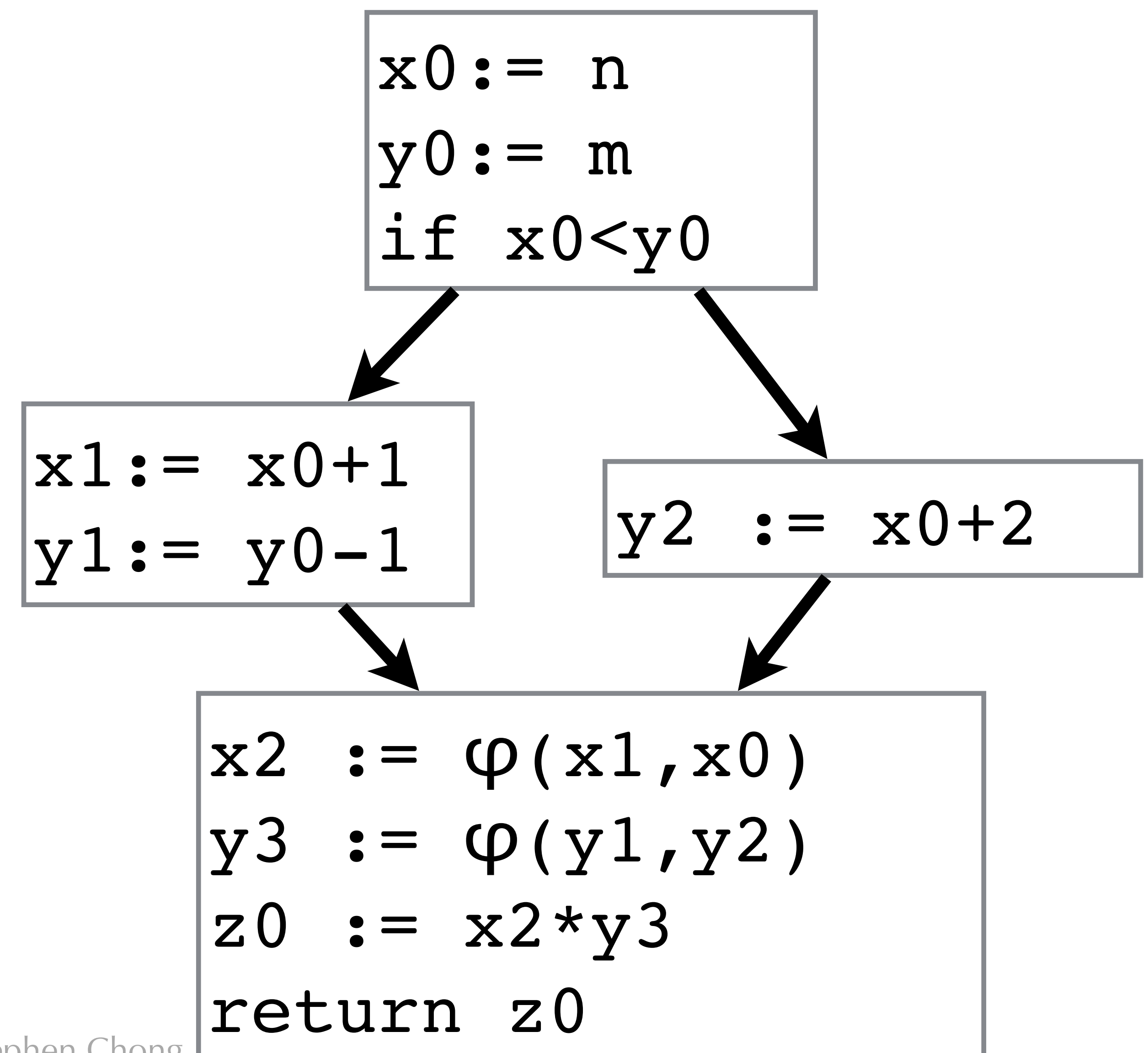


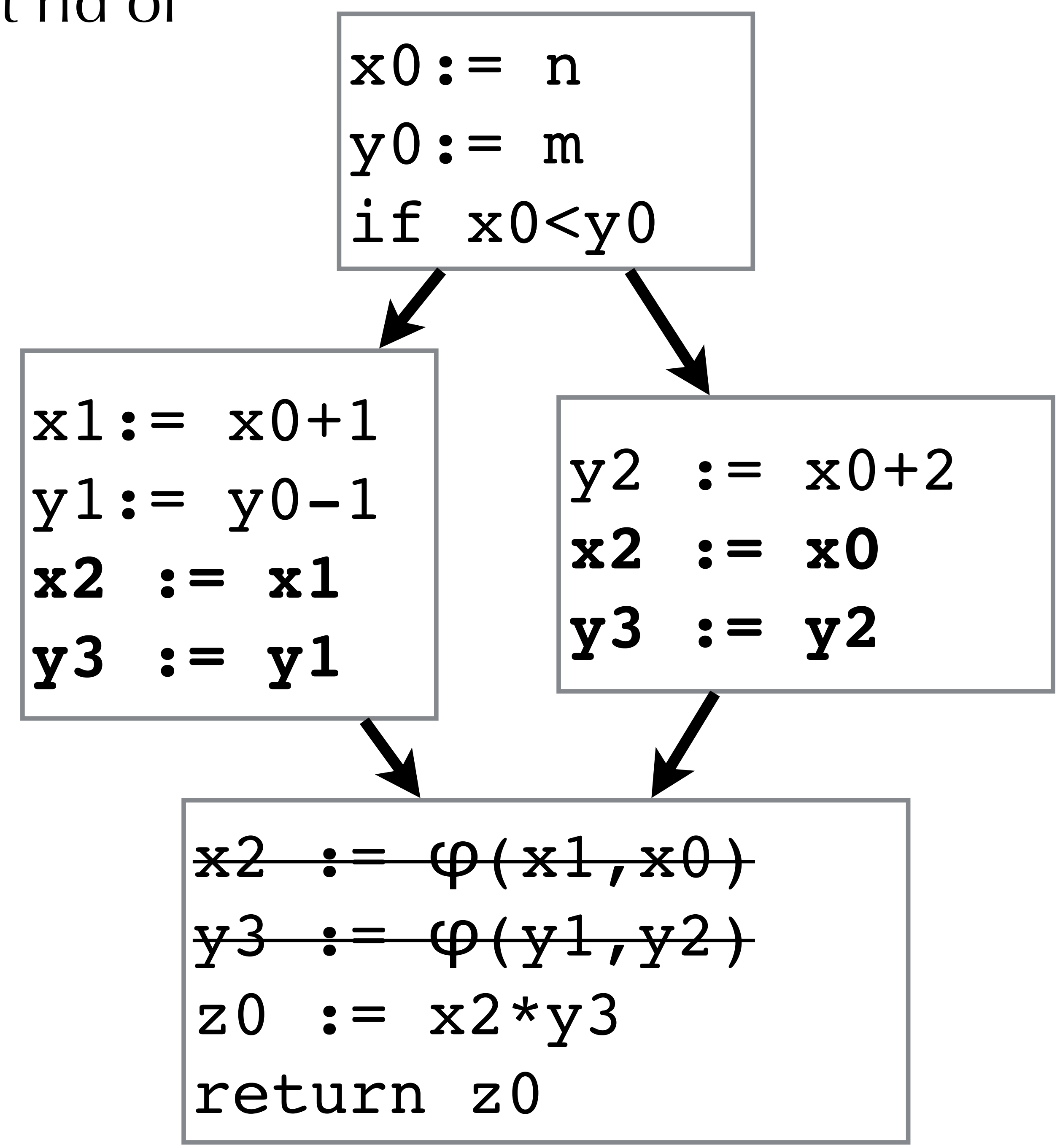
Remaining Issues

- How do we generate SSA from CFG representation?
  - In order to get benefits of SSA form
- How do we generate CFG (or MIPS) from SSA?
  - In order to take SSA form and continue with code generation

SSA Back to CFG

- Simply insert assignments corresponding to phi nodes on the edges
  - Coalescing register allocation will get rid of copies...

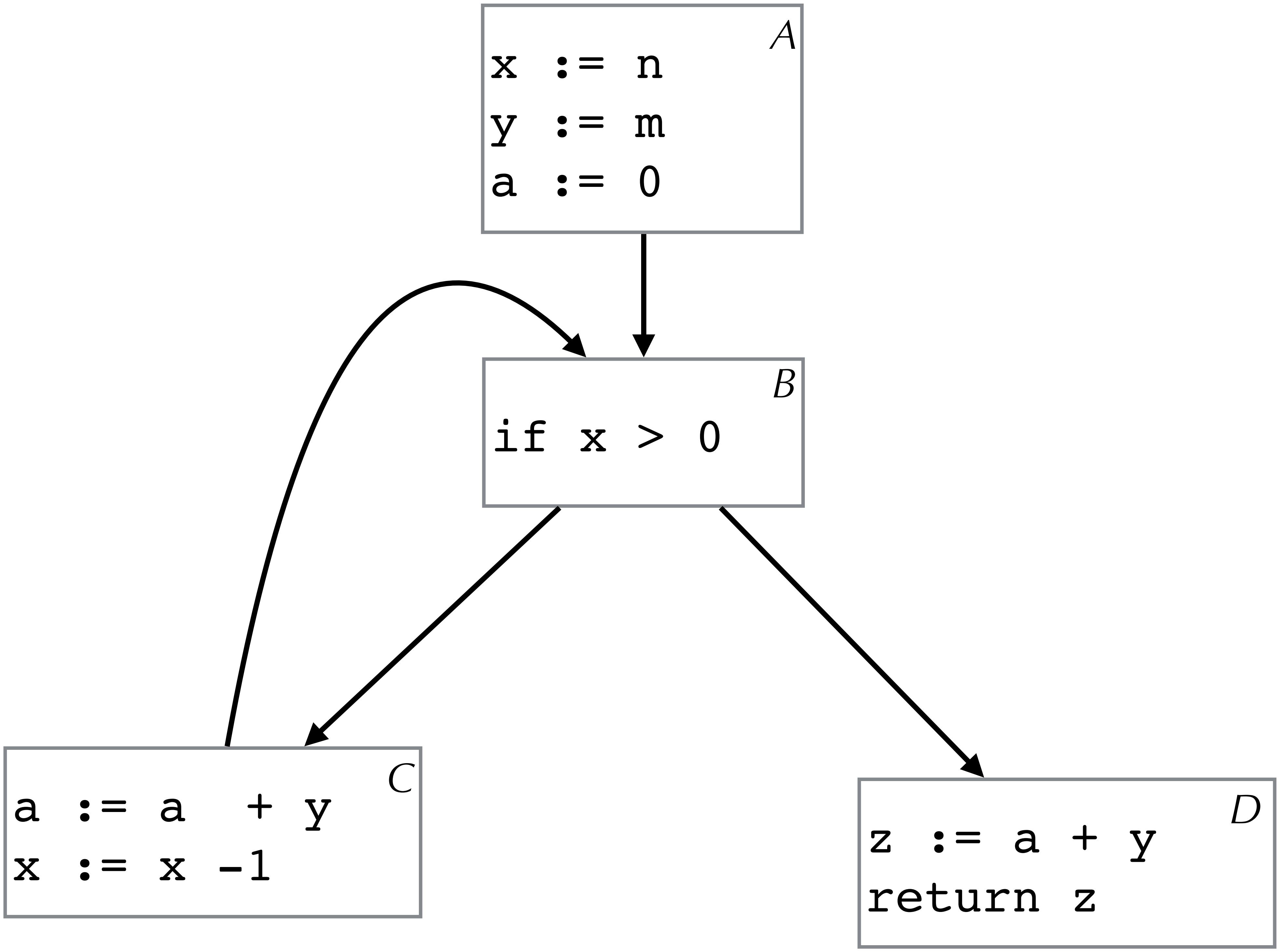




CFG to SSA, Naively

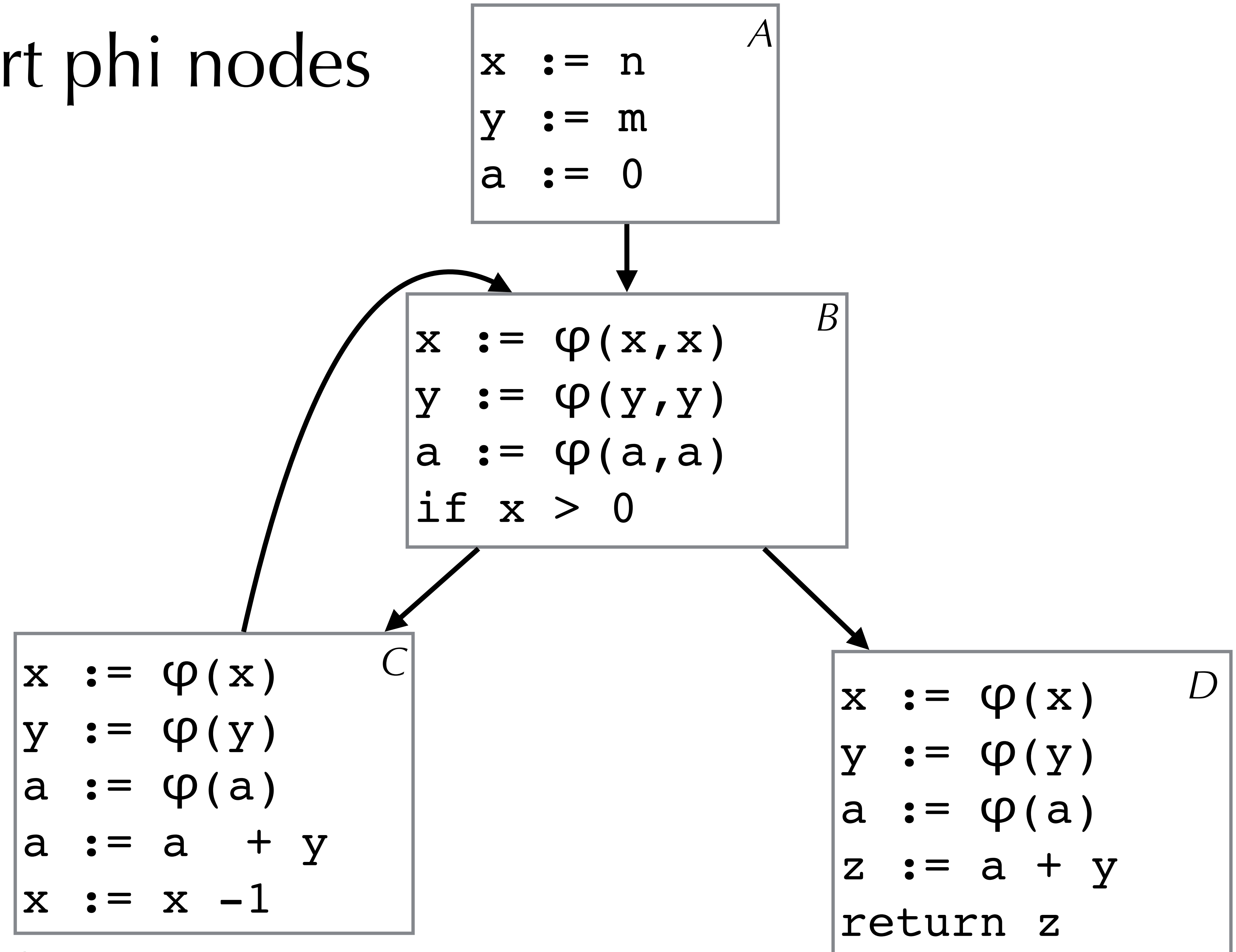
- Insert phi nodes in each basic block except the start node.
  - Could limit insertion to nodes with >1 predecessor, but for simplicity we will insert phi nodes everywhere.
- Calculate the dominator tree.
- Traverse the dominator tree in a breadth-first fashion:
  - give each definition of `x` a fresh index
  - propagate that index to all of the uses
    - each use of `x` that is not killed by a subsequent definition.
    - propagate the last definition of `x` to the successors' phi nodes.

Example

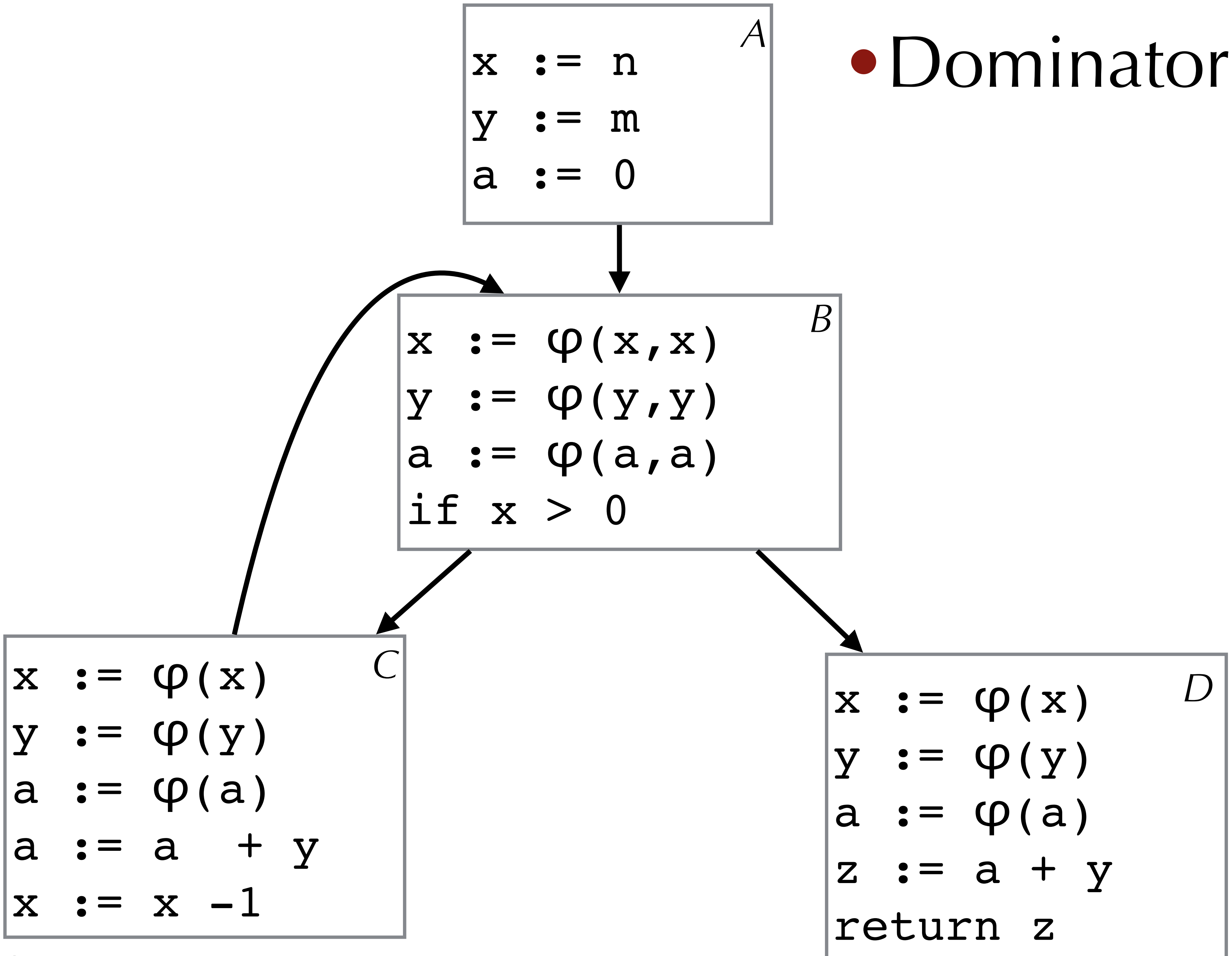


Example

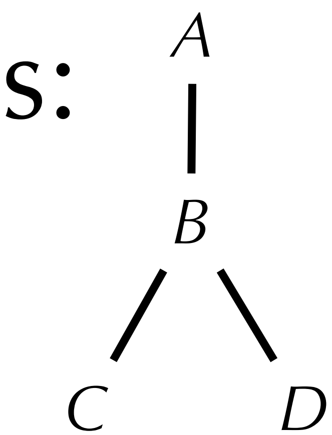
- Insert phi nodes



Example

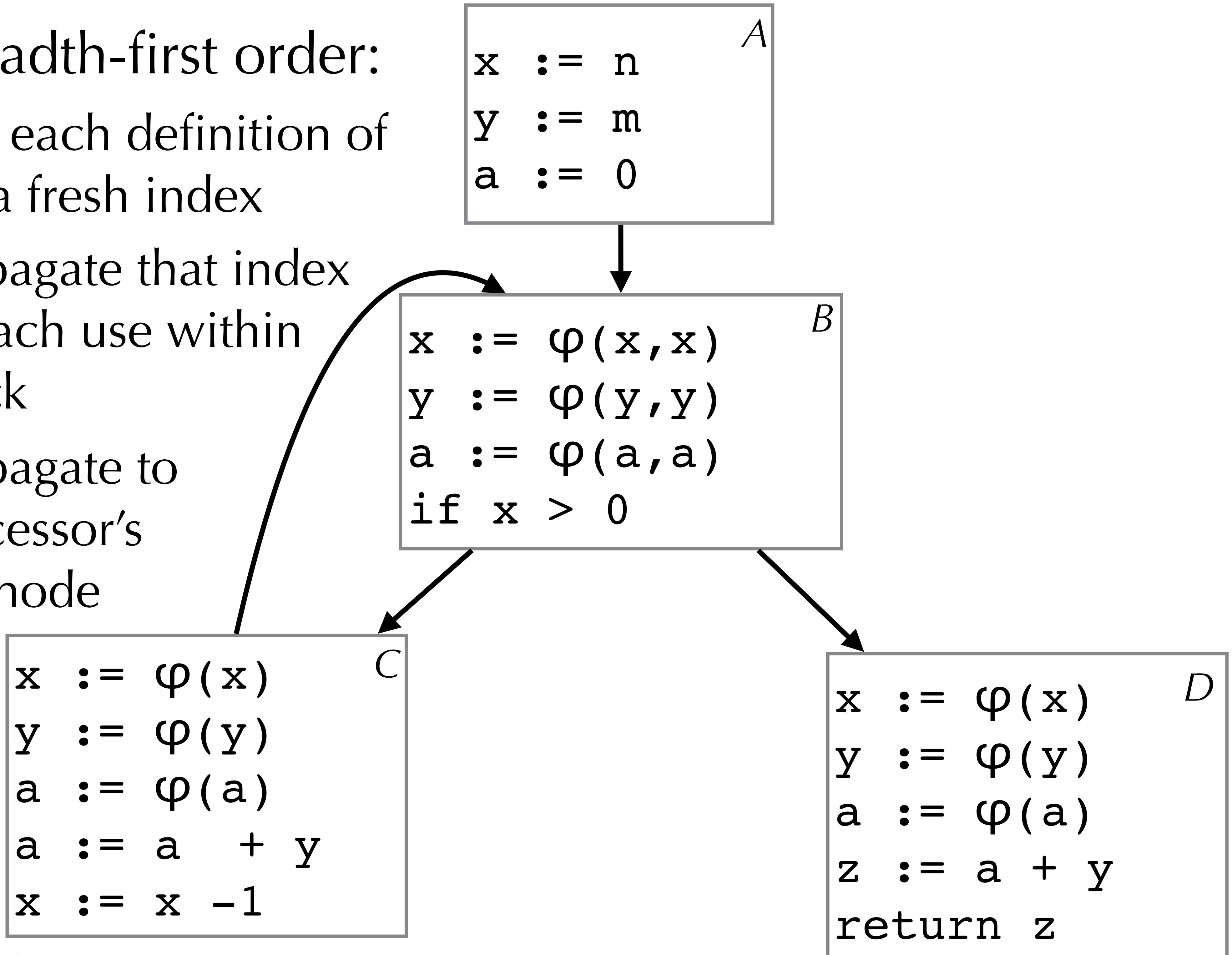


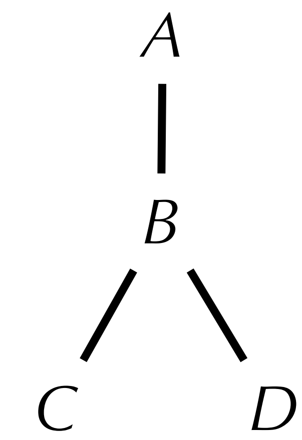
- Dominators:



Example

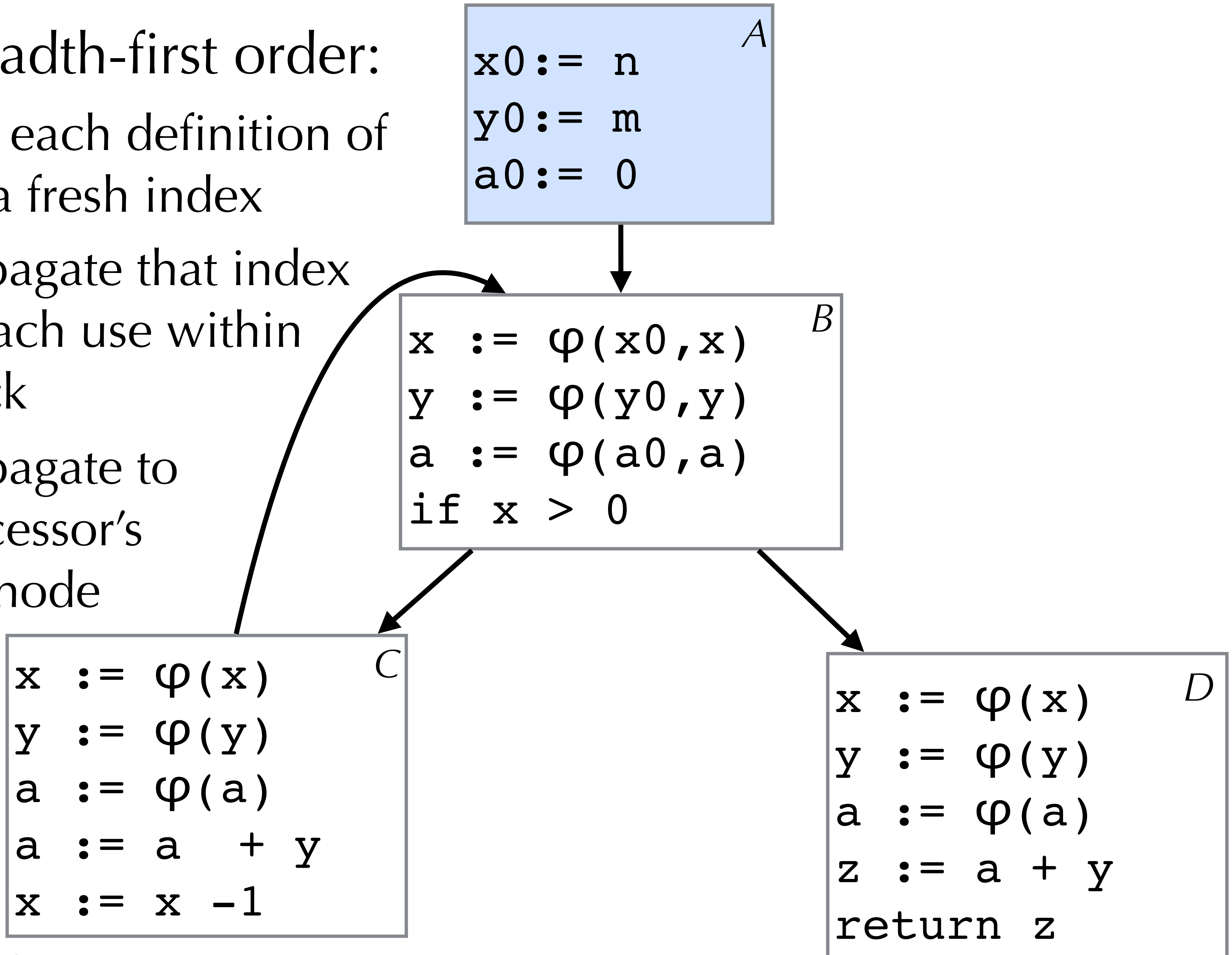
- In breadth-first order:
  - give each definition of var a fresh index
  - propagate that index to each use within block
  - propagate to successor's phi node

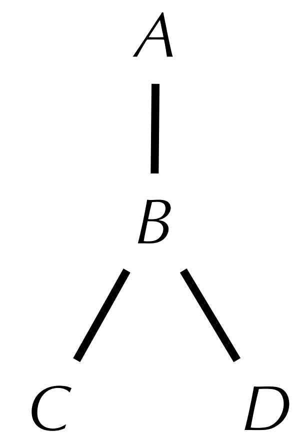




Example

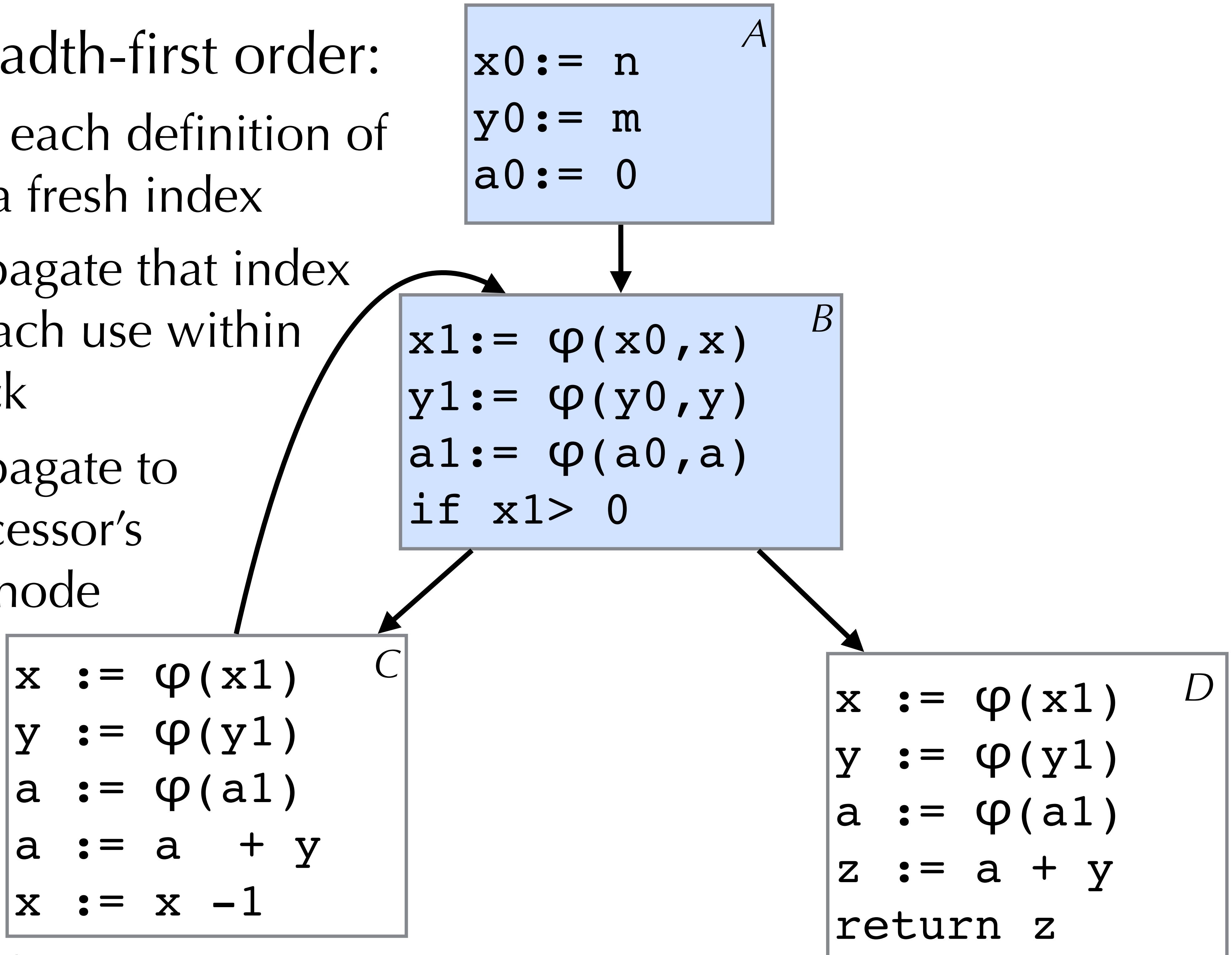
- In breadth-first order:
  - give each definition of var a fresh index
  - propagate that index to each use within block
  - propagate to successor's phi node

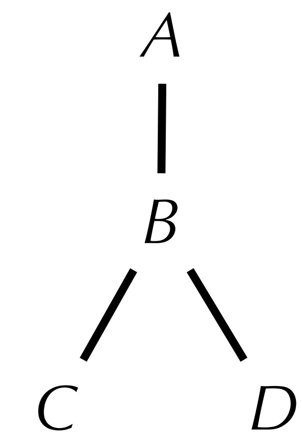




Example

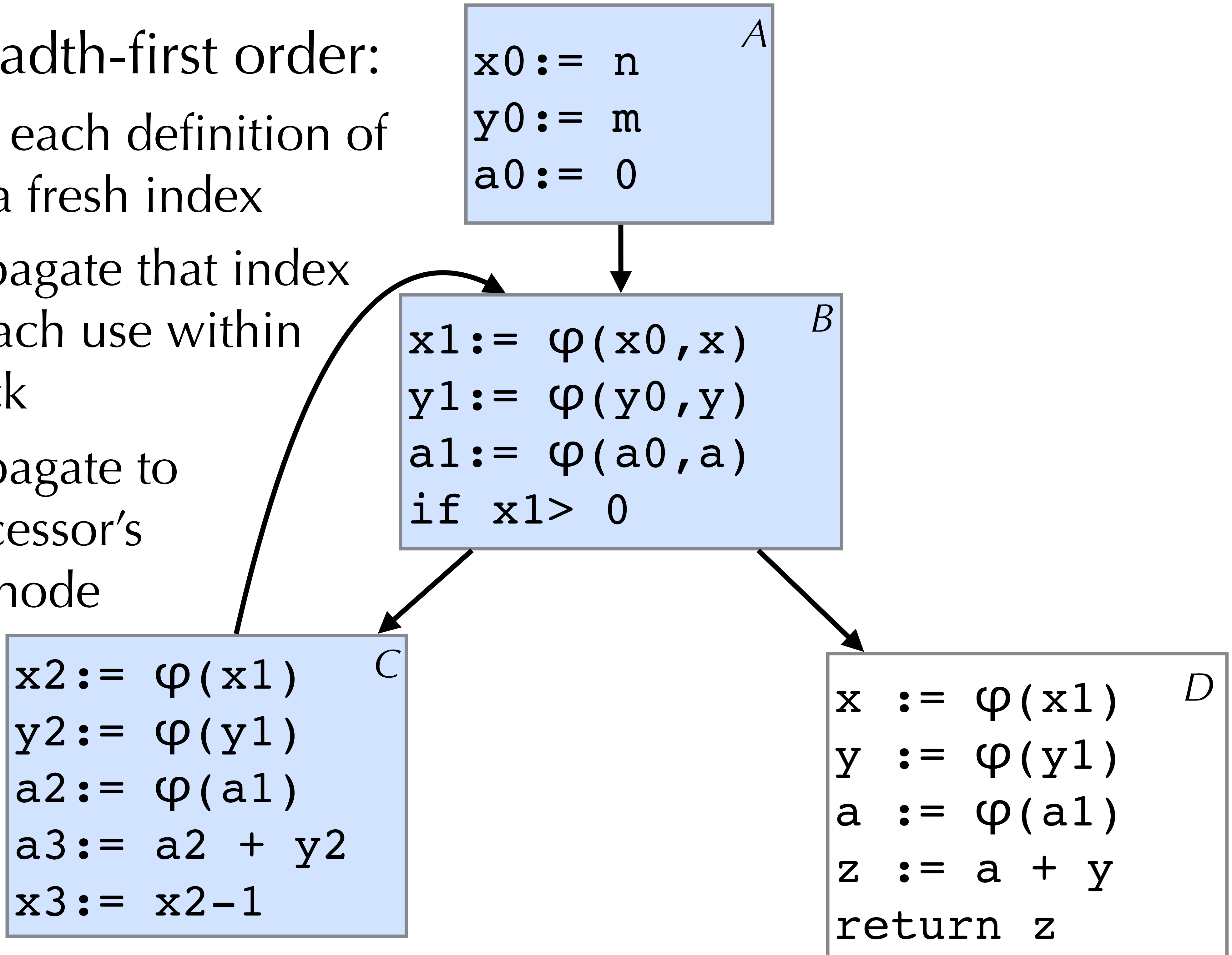
- In breadth-first order:
  - give each definition of var a fresh index
  - propagate that index to each use within block
  - propagate to successor's phi node

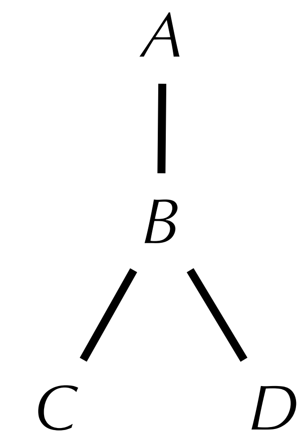




Example

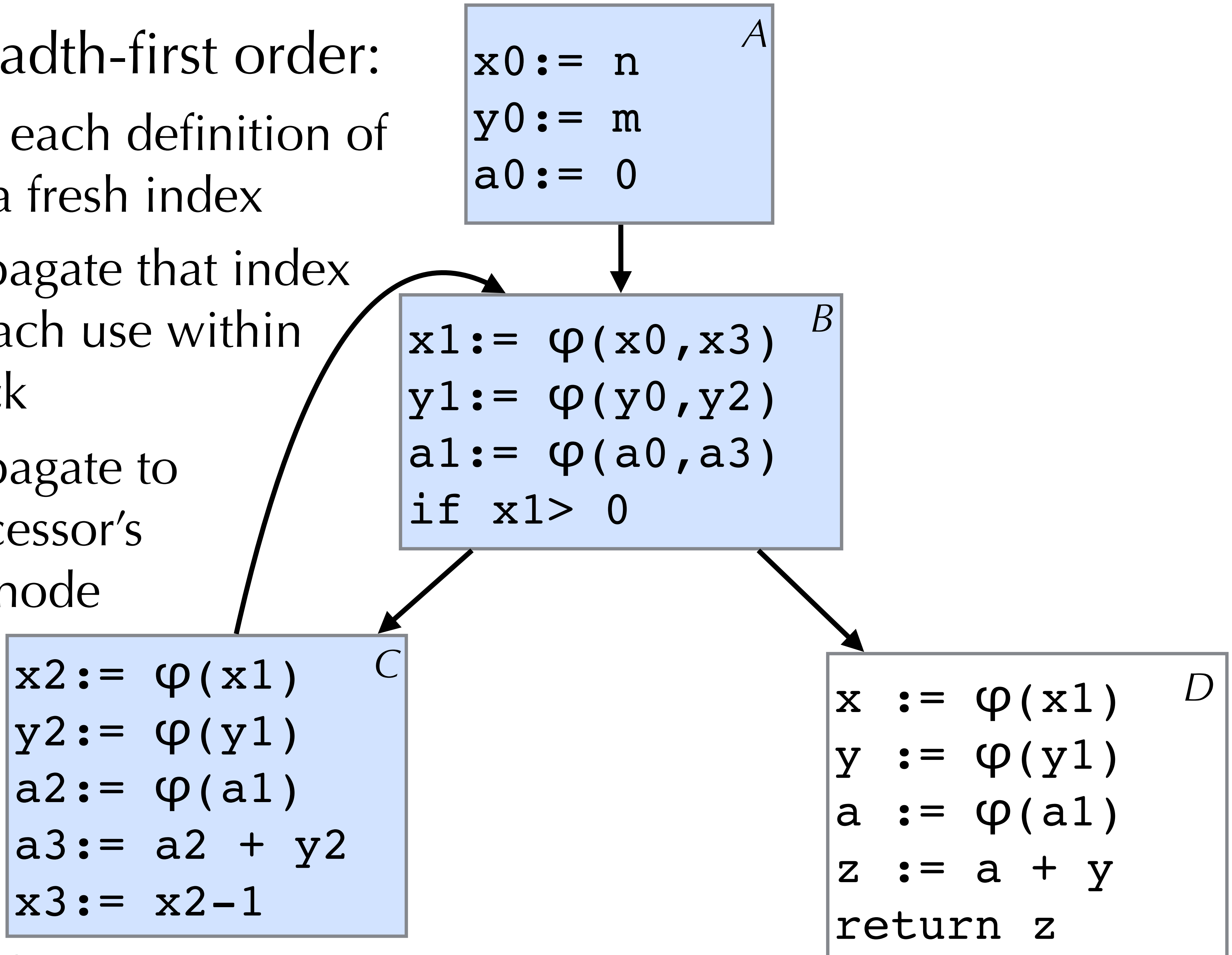
- In breadth-first order:
  - give each definition of var a fresh index
  - propagate that index to each use within block
  - propagate to successor's phi node

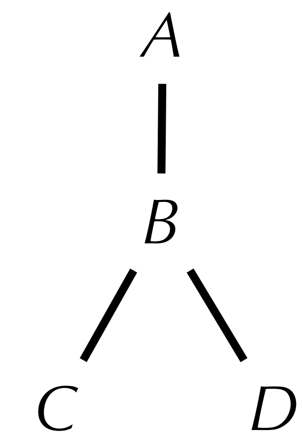




Example

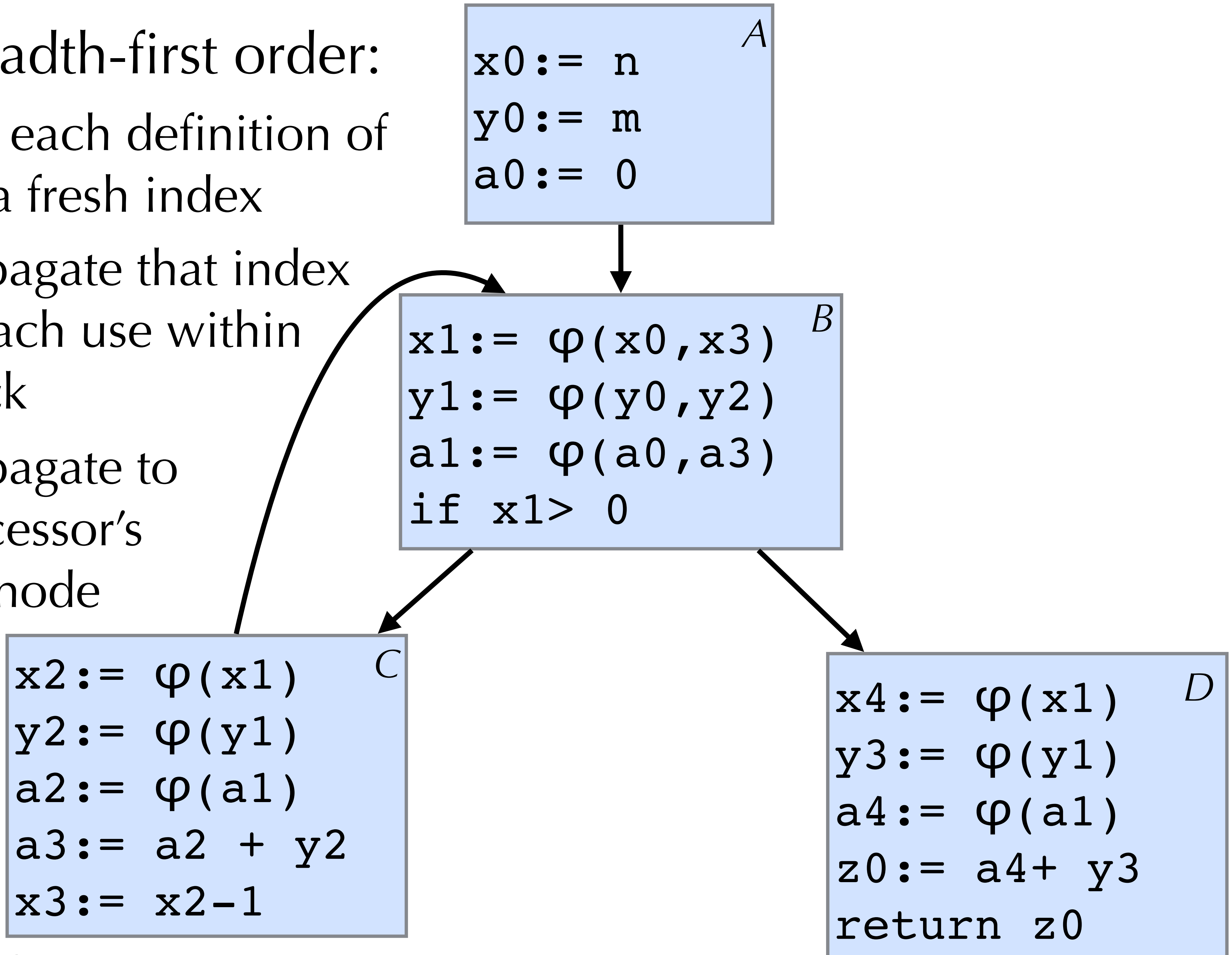
- In breadth-first order:
  - give each definition of var a fresh index
  - propagate that index to each use within block
  - propagate to successor's phi node

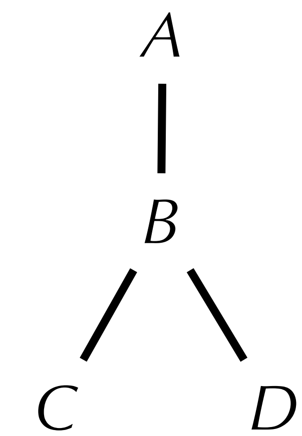




Example

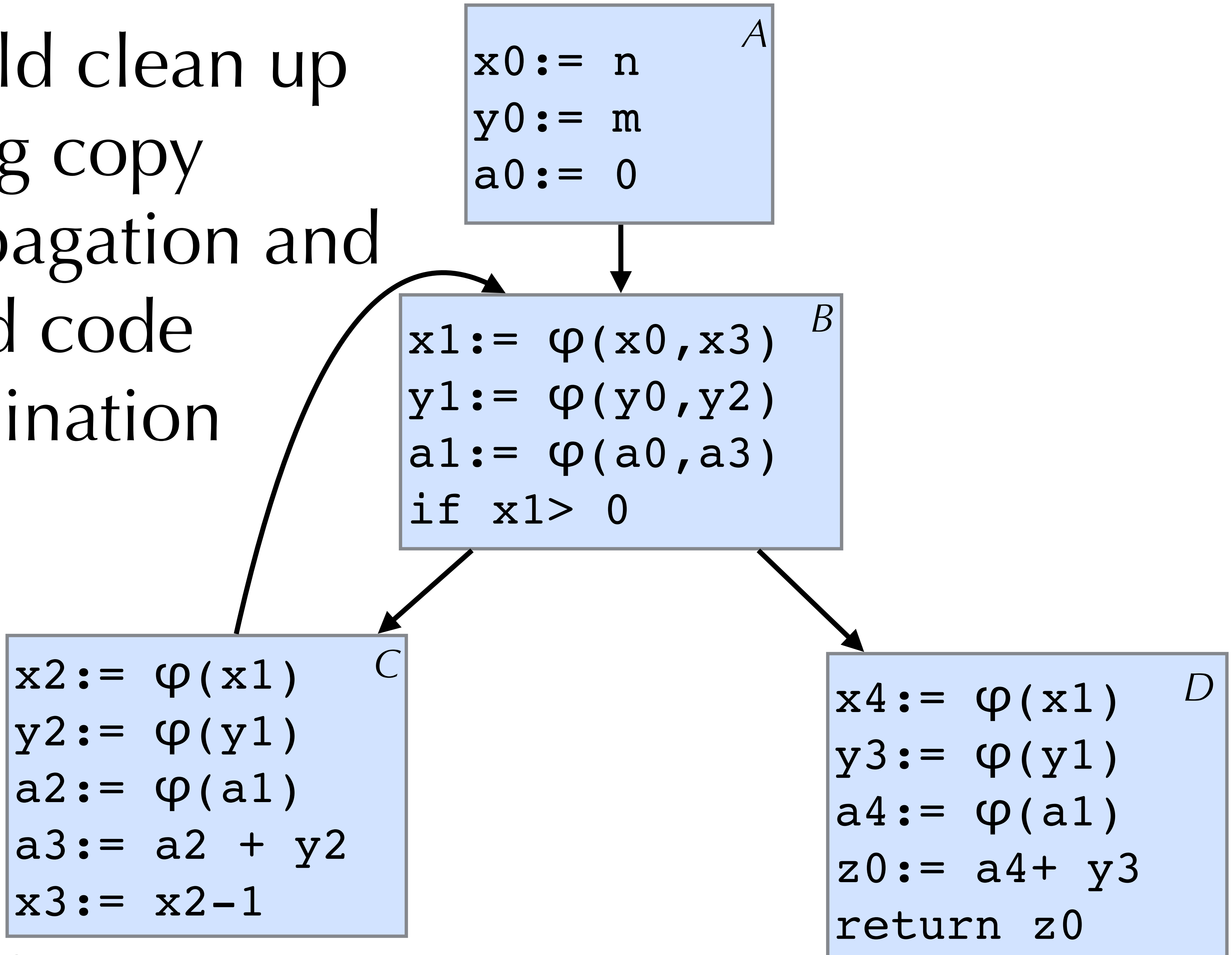
- In breadth-first order:
  - give each definition of var a fresh index
  - propagate that index to each use within block
  - propagate to successor's phi node





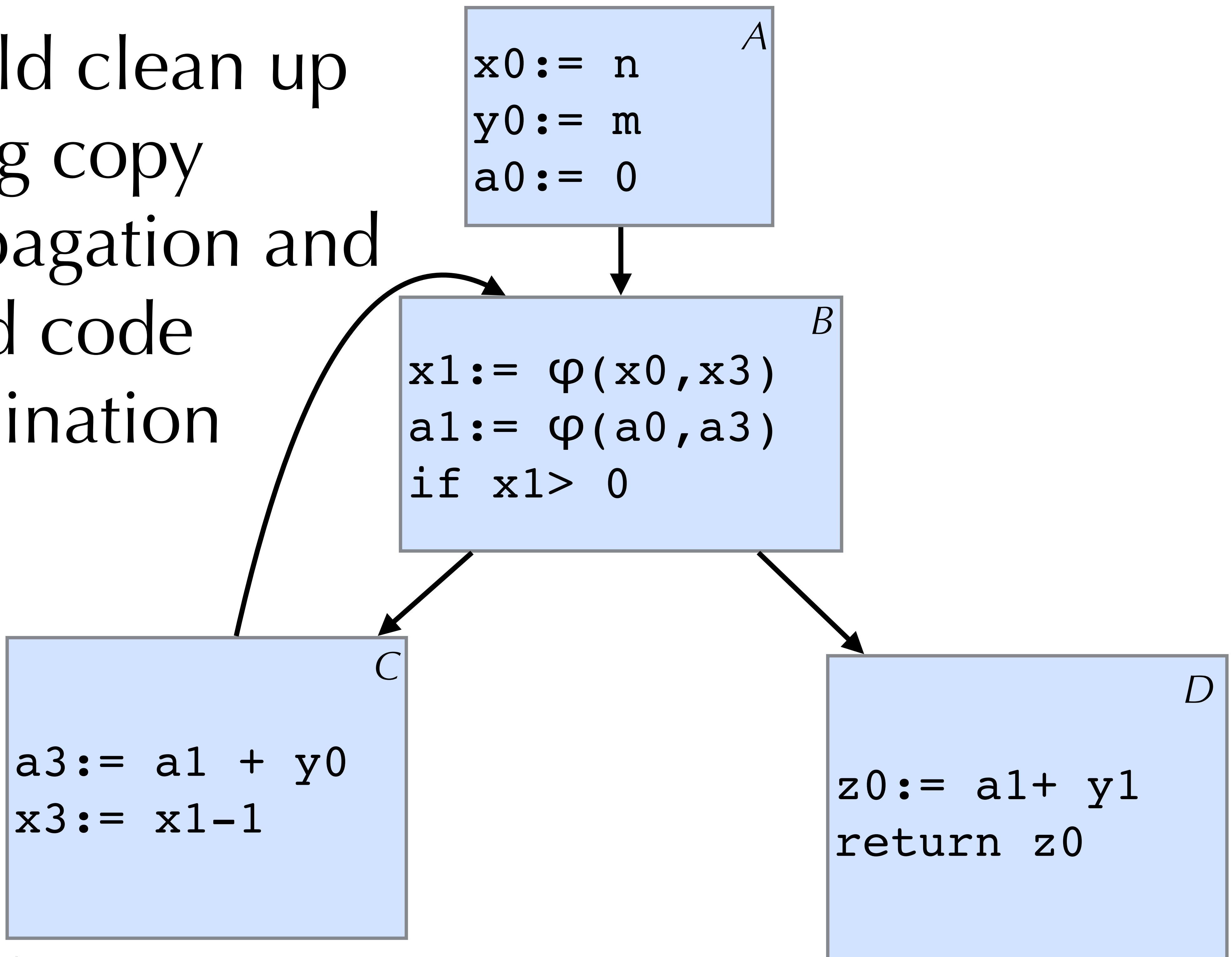
Example

- Could clean up using copy propagation and dead code elimination



Example

- Could clean up using copy propagation and dead code elimination



Smarter Algorithm for CFG to SSA

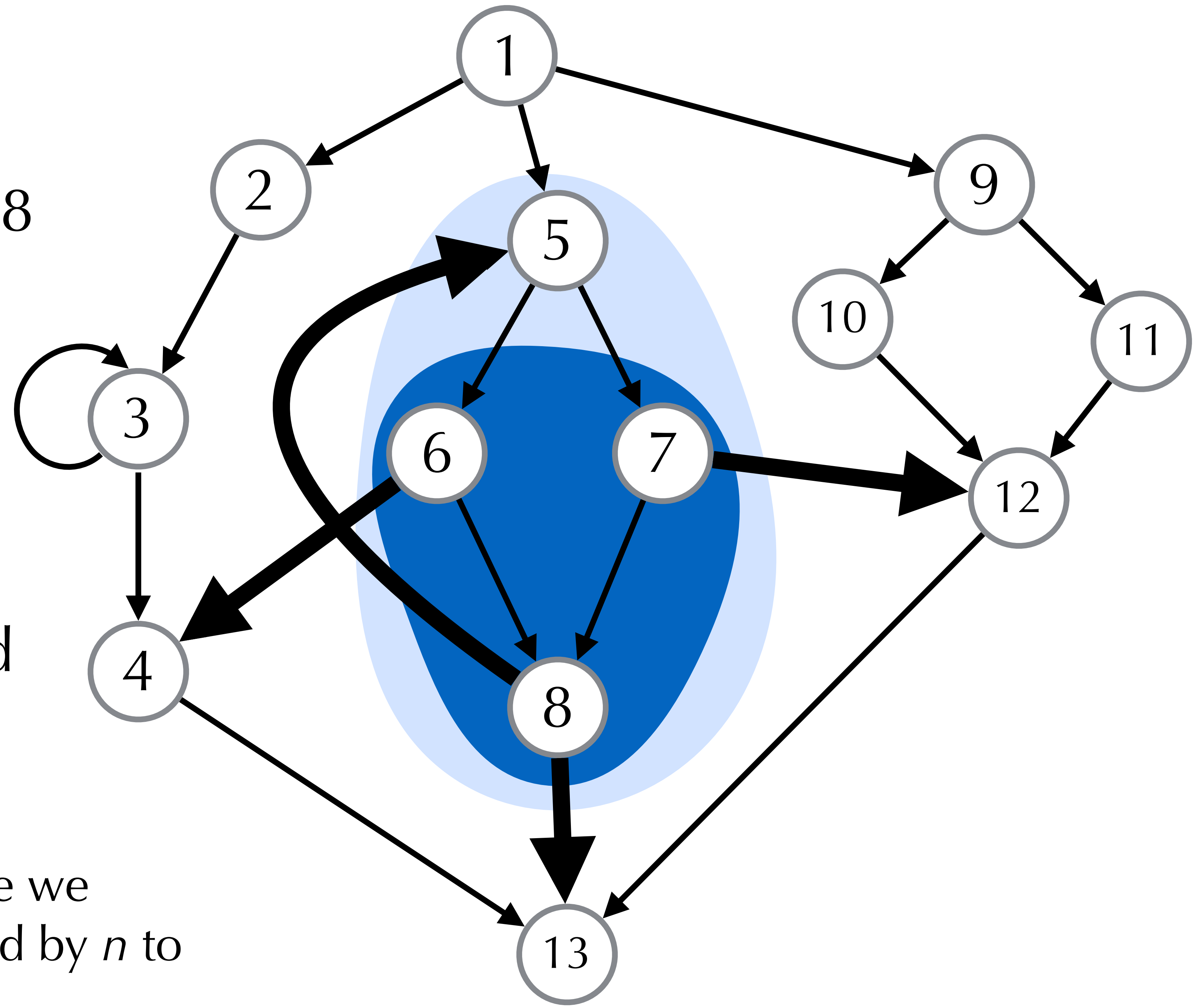
- Compute the **dominance frontier**
- Use dominance frontier to place phi nodes
  - Whenever block *n* defines `x`, put a phi node for `x` in every block in the dominance frontier of *n*
- Do renaming pass using dominator tree

Dominance Frontier

- Definition: ***d* dominates *n*** if every path from the start node to *n* must go through *d*
- Definition: if *d* dominates *n* and *d*≠*n*, we say ***d* strictly dominates *n***
- Definition: the **dominance frontier** of *n* is the set of all nodes *w* such that
  - 1. *n* dominates a predecessor of *w*
  - 2. *n* does not strictly dominate *w*

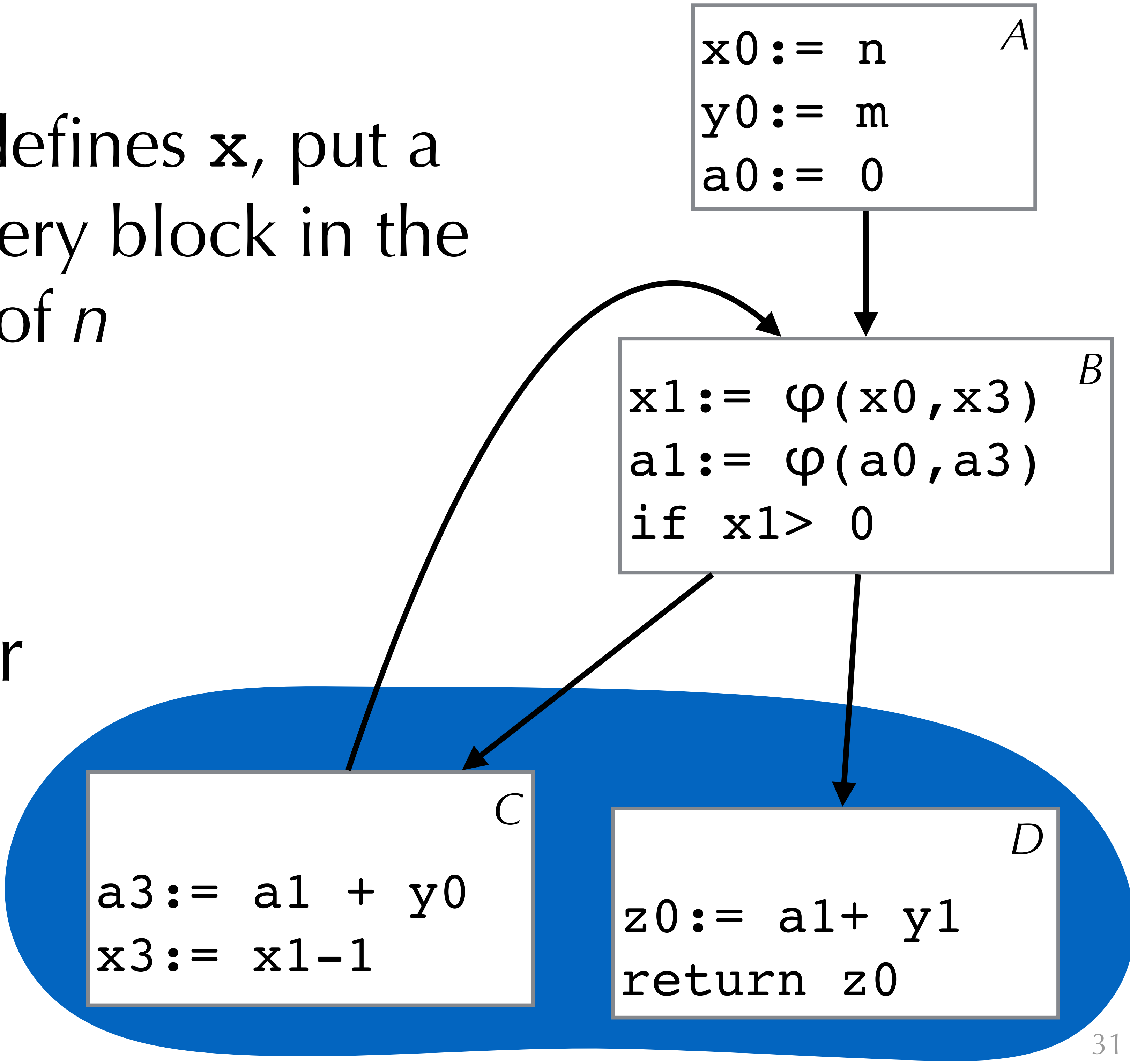
Example

- Node 5
  - dominates 5,6,7,8
  - strictly dominates 6,7,8
- Dominance frontier of 5 is 4,5,12,13
  - Targets of edges from nodes dominated to nodes not strictly dominated
- Dominance frontier of *n*: where we transition from being dominated by *n* to being not strictly dominated



Example

- Recall alg:
  - Whenever block *n* defines **x**, put a phi node for **x** in every block in the dominance frontier of *n*
- Block *B* strictly dominates *C*,*D*
- Dominance frontier of *B* is *B*



Notes

- Adding a phi node for variable **`x`** is a new definition of **`x`**
  - Need to iterate until we satisfy the dominance frontier criterion:
    - Whenever block *n* defines **`x`**, put a phi node for **`x`** in every block in the dominance frontier of *n*
- 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
- Computing dominator tree using iterative dataflow algorithm
  - With careful engineering, worst case complexity is quadratic, but in practice linear
  - See "A Simple, Fast Dominance Algorithm" by Cooper, Harvey, and Kennedy, *Software Practice & Experience 4*, 2001
    - Faster than an $O(N+\log(E))$ algorithm for CFGs with <30,000 nodes

Optimization Algorithms Using SSA

- We promised some optimization algorithms were simpler in SSA! Let's look at some...
- Assume that our compiler data structures include:
  - Statement
  - Variable: has definition site (statement) and list of use sites
  - Block: has list of statements, ordered list of predecessors, successor(s)

Dead-Code Elimination

- Recall: Variable **`x`** is **live** at program point *p* is there is a path from *p* to a use of variable **`x`**
- A variable is live at its definition site if and only if its list of uses is non empty
  - Thanks SSA! Definition site dominates all uses, so there is a path from definition site to use site
- Iterative alg for removing dead code:
  - While there is a variable **`x`** with no uses and the statement that defines **`x`** has no other side effects:
    - Delete the statement that defines **`x`**

Work-list Algorithm for DCE

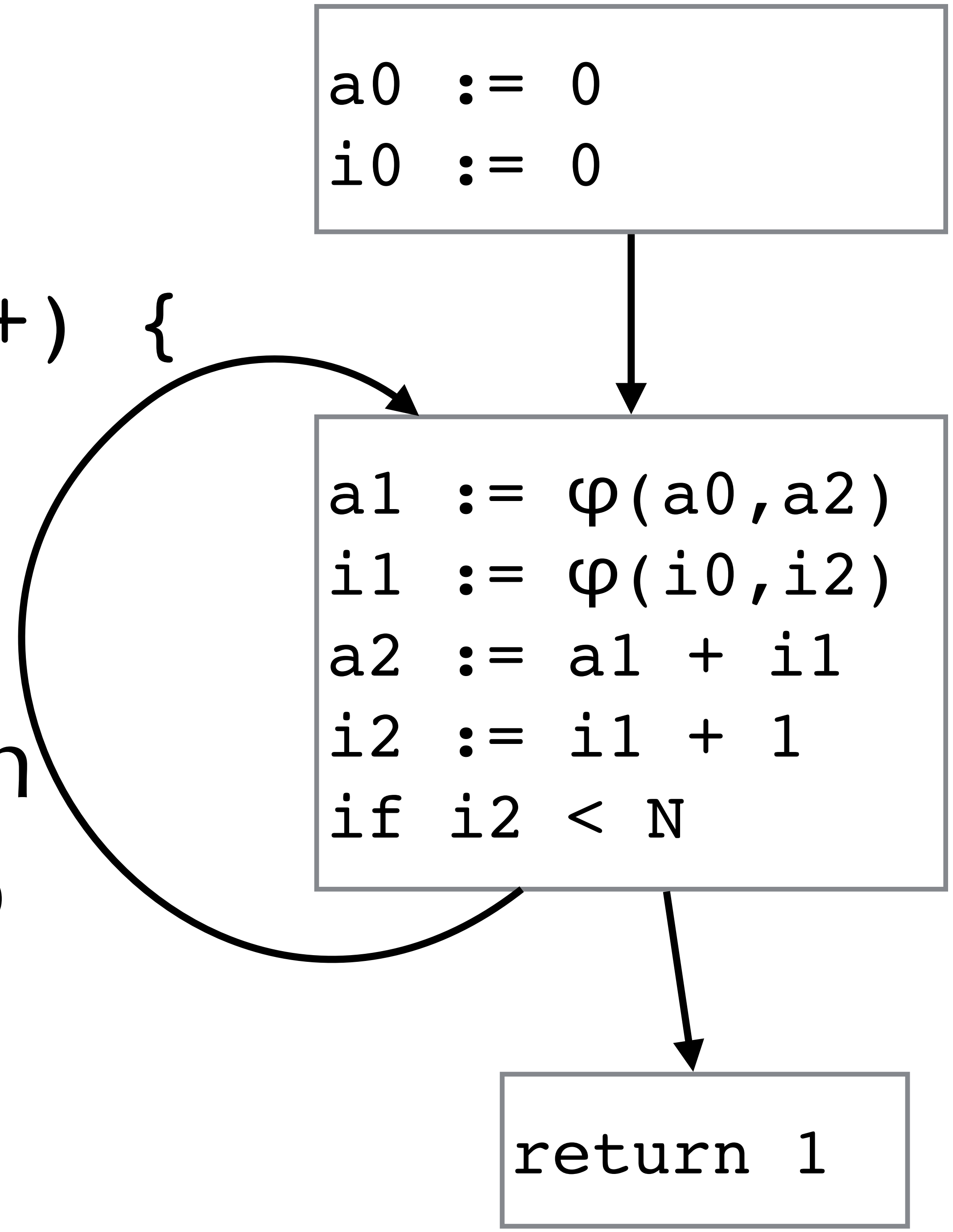
$W \leftarrow$ all variables in SSA program
while W is not empty:
    remove some v from W
    if v's list of uses is empty:
        let S be v's statement of definition
        if S has no side effects other than assignment to v:
            delete S from program
            for each x_i used by S:
                delete S from list of uses of x_i
                $W \leftarrow W \cup \{ x_i \}$

More Agressive DCE

- Consider program

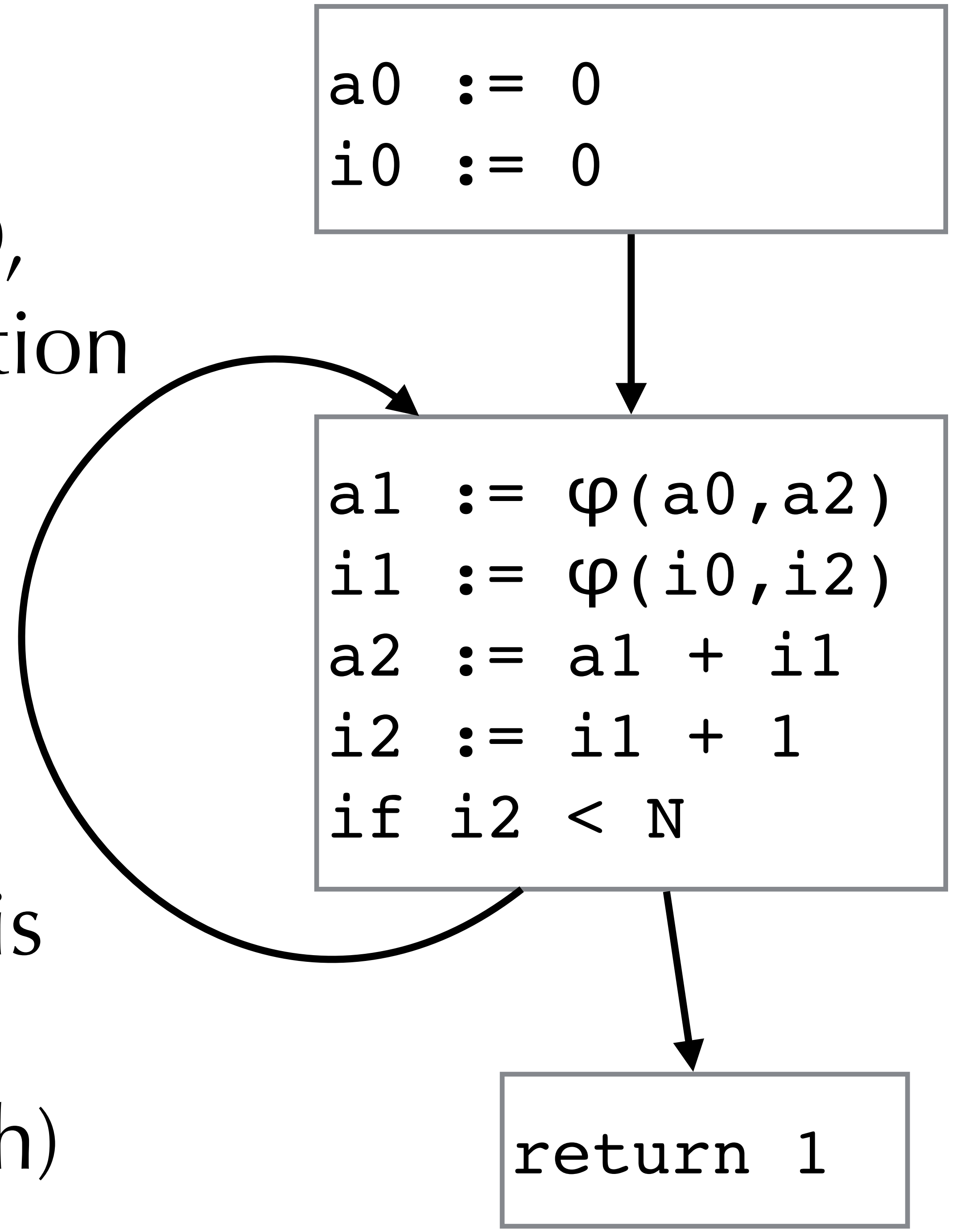
```
a := 0;
for (int i = 0; i < N; i++) {
   a := a+i;
}
return 1
```

- Variables are live at definition site, but doesn't contribute to result of program!



More Agressive DCE

- Mark live any statement that:
  - 1. stores into mem, performs I/O, returns from function, calls function that may have side effects
  - 2. defines variable that is used in a live statement
  - 3. is a conditional branch that affects whether a live statement is executed (i.e., live statement is **control dependent** on the branch)
- Remove all unmarked statements



More Agressive DCE

- Mark live any statement that:
  - 1. stores into mem, performs I/O, returns from function, calls function that may have side effects
  - 2. defines variable that is used in a live statement
  - 3. is a conditional branch that affects whether a live statement is executed (i.e., live statement is **control dependent** on the branch)
- Remove all unmarked statements

```
return 1
```

Simple Constant Propagation

- Any statement `x:=c` for constant `c`: can replace uses of `x` with `c`
- Any phi node `x:=φ(c,...,c)` can be replaced with `x:=c`
- Easy to detect and implement with SSA form!

$W \leftarrow$ all statements in SSA program
while W is not empty:
    remove some S from W
    if S is of form `x:=φ(c,...,c)`:
        replace S with `x:=c`
    if S is of form `x:=c`:
        delete S from program
        for each statement T that uses `x`
            substitute `c` for `x` in T
            $W \leftarrow W \cup \{ T \}$