CS293S SSA

Yufei Ding

Review of Last class

- Data flow analysis (DFA)
  - Live variables, Very busy expression, …
  - Some common concepts in DFA: domain, direction, may/must

Focus of This Class

- Static Single Assignment(SSA)
  - Maximal SSA
  - Minimal SSA
  - Semipruned SSA
  - Pruned SSA

Static Single Assignment (SSA)

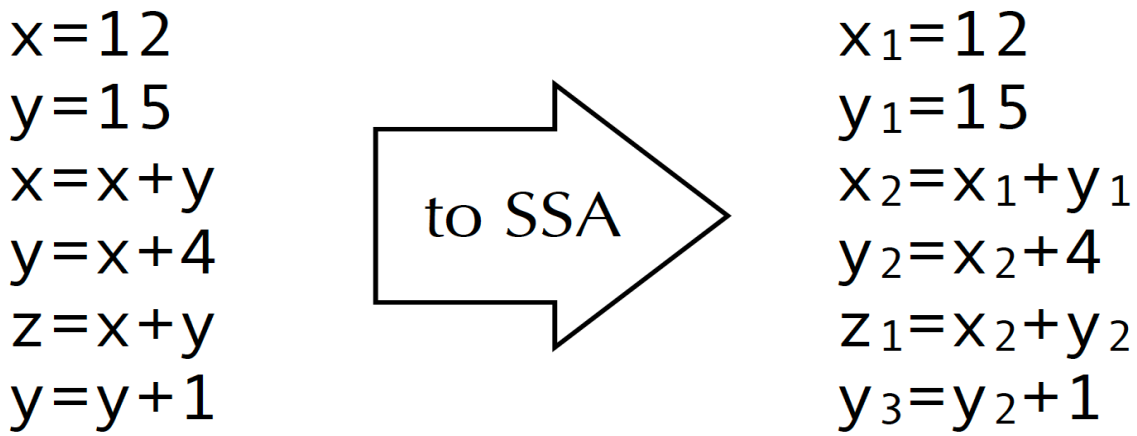
- SSA-form
  - Each name is defined exactly once
  - Each use refers to exactly one name

- Why static?
- That single definition can be executed many times when the program is run – if it is inside a loop – hence the qualifier static.

- Another key intermediate representation (IR).
  - serve as the basis for a large set of transformations.
  - simplify several optimizations and analysis, as we will see.

Straight-line code

- Transforming a piece of straight-line code – i.e. without branches – to SSA is trivial: each definition of a given name gives rise to a new version of that name, identified by a subscript:



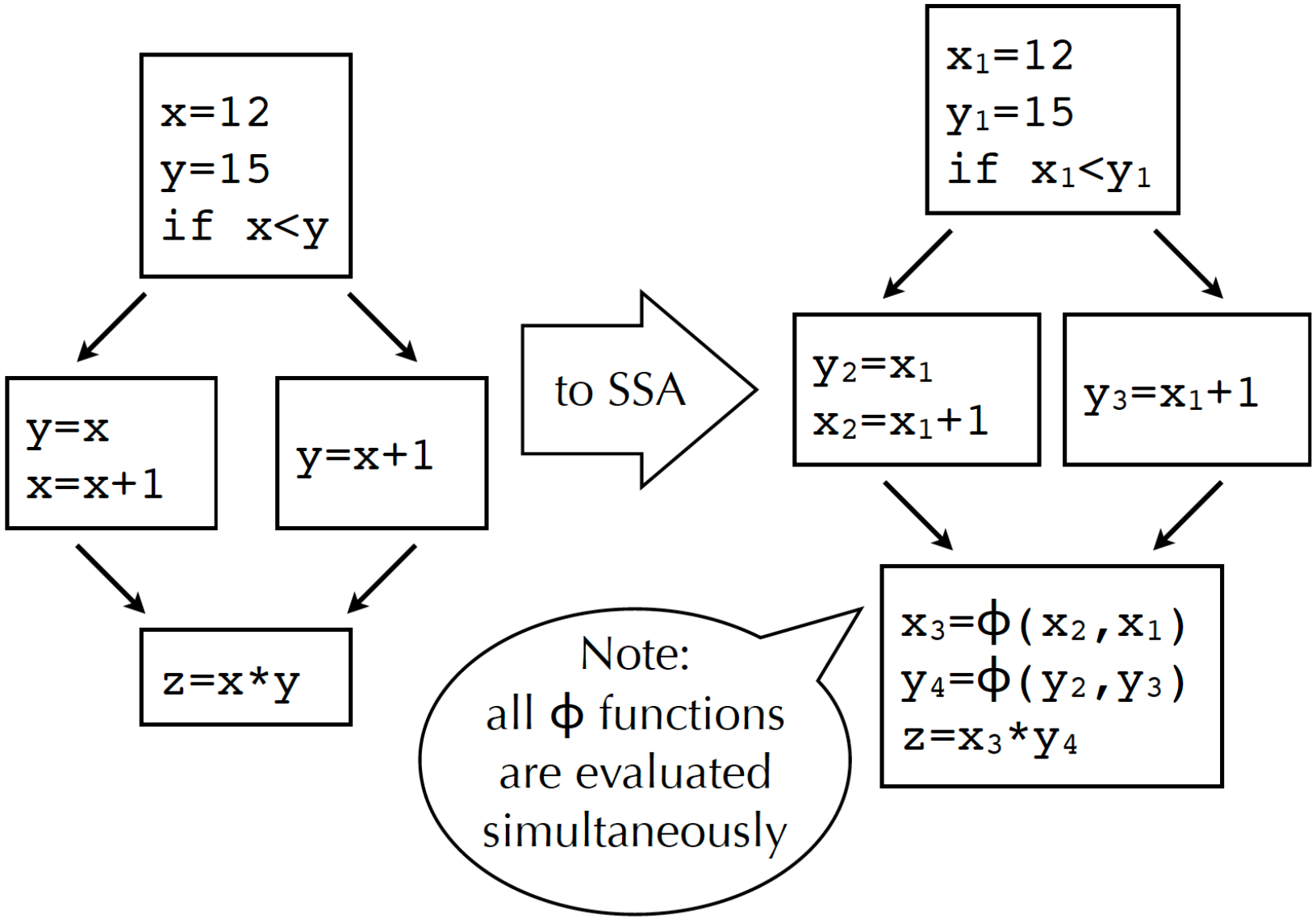
Ø-functions

- Join-points in the CFG – nodes with more than one predecessors – are more problematic, as each predecessor can bring its own version of a given name.

- To reconcile those different versions, a fictional Ø-function is introduced at the join point. That function takes as argument all the versions of the variable to reconcile, and automatically selects the right one depending on the flow of control.

Real machines do not implement a Ø-function directly in hardware.

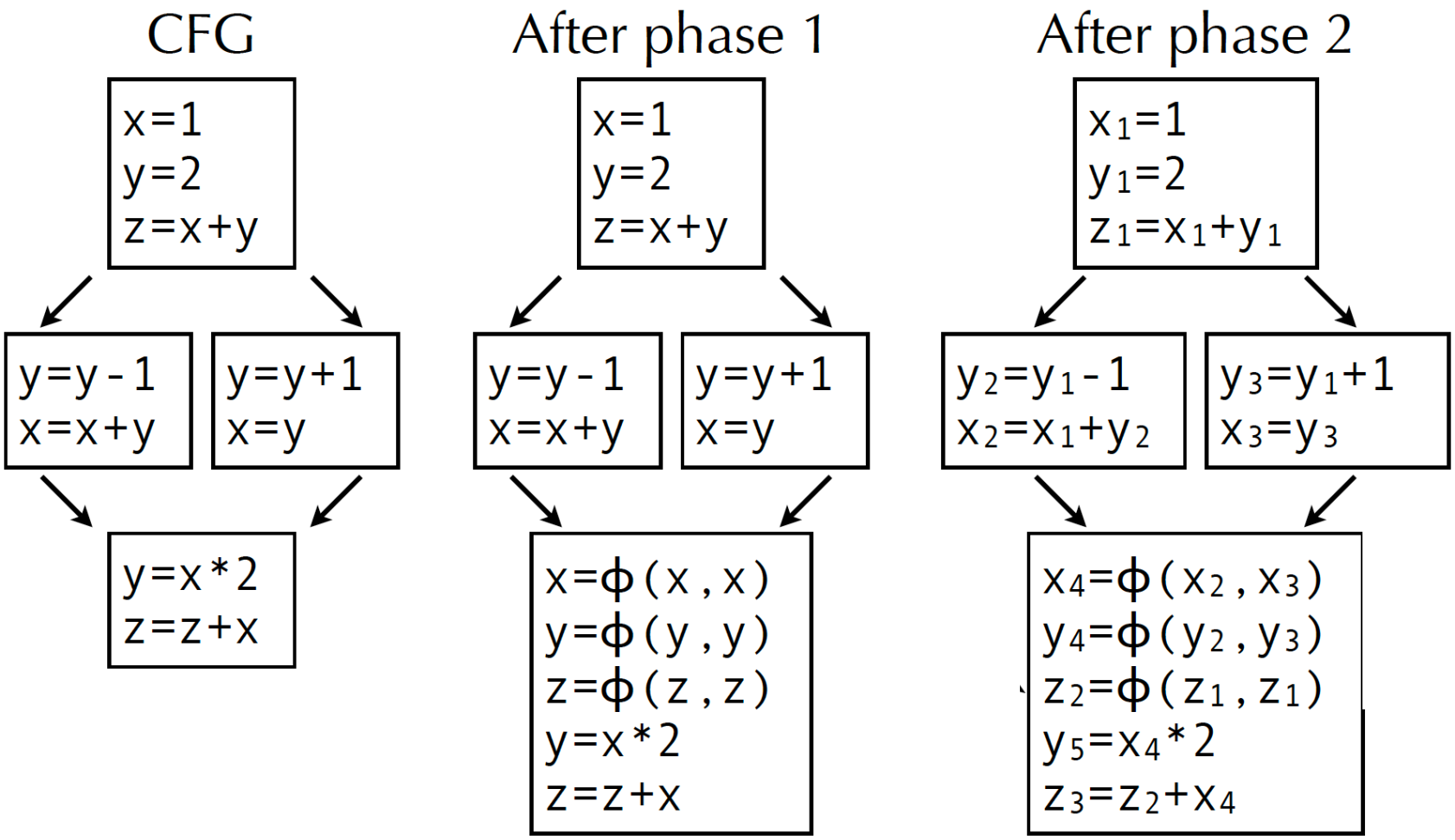
Ø-function Example



(Naïve) building of SSA form

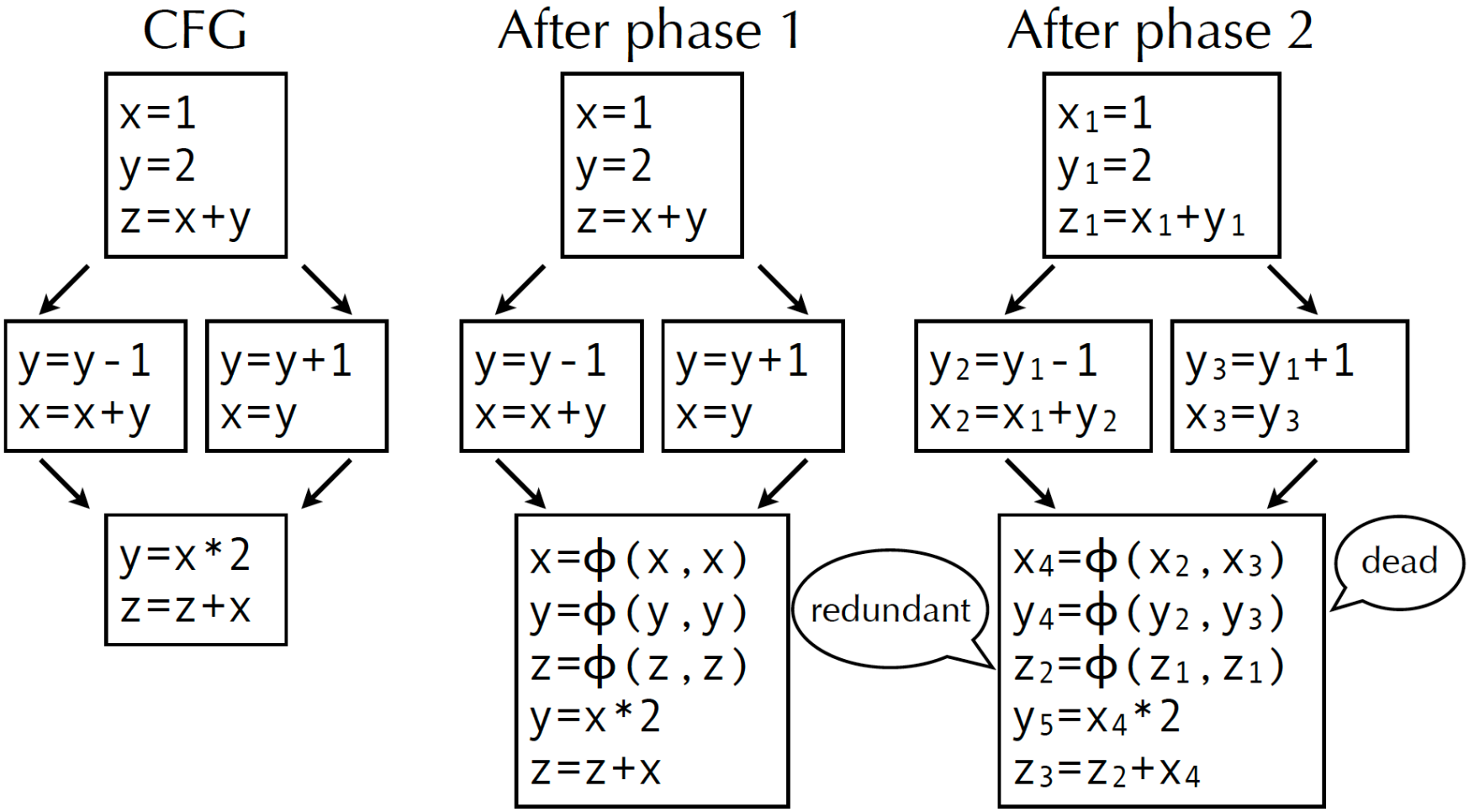
- for each variable x of the CFG, at each join point n, insert a Ø-function of the form x=Ø(x,…,x) with as many parameters as n has predecessors,
- Also known as maximal SSA.

(Naïve) building of SSA form



Are all #phi functions necessay?

(Naïve) building of SSA form



Any idea to make it a compilation optimization? It should be automatic and general applicable.

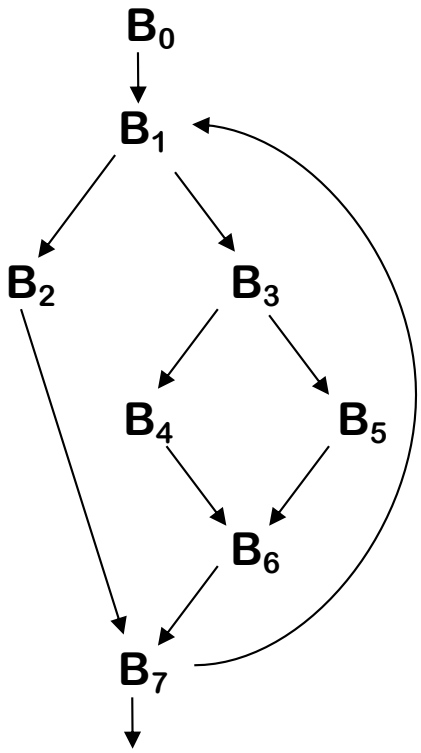
Smarter techniques

- The naïve technique just presented works, in the sense that the resulting program is in SSA form. It builds the maximal SSA form.

- However, it introduces too many Ø-functions – some dead, some redundant – to be useful in practice.

- We will examine better techniques later, but to understand them we want to review the notion of dominance in a CFG.

Dominance example



Dominance

Node	Dominators
0	{ 0 }
1	{ **0**, 1 }
2	{ 0, **1**, 2 }
3	{ 0, **1**, 3 }
4	{ 0, 1, **3**, 4 }
5	{ 0, 1, **3**, 5 }
6	{ 0, 1, **3**, 6 }
7	{ 0, **1**, 7 }

(immediate dominator in bold)

For DVN in HW1, you compute the dominance tree manually, but everything should be automatic in a compiler optimization path? Hint: data flow analysis framework.

Dominance

- Recall that n dominates m iff n is on every path from n_0 to m
  - Every node dominates itself

$$\text{DOM}(n_0) = \{ n_0 \}$$

$$\text{DOM}(n) = \{ n \} \cup (\cap_{p \in \text{preds}(n)} \text{DOM}(p))$$

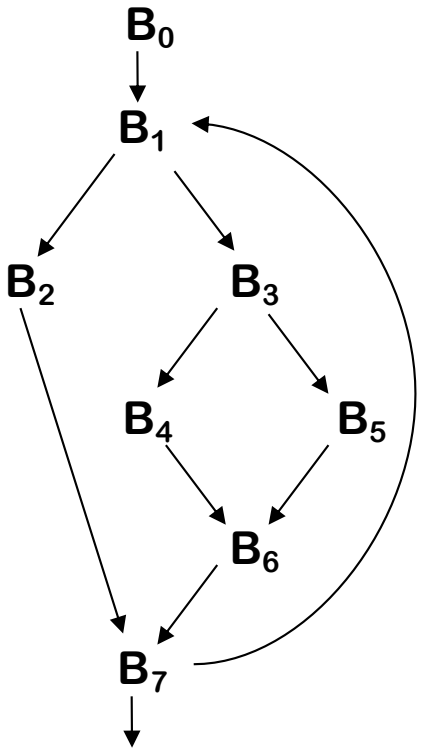
Initially, DOM(n) = all, $\forall\ n \neq n_0$

- These equations form a rapid data-flow framework
- n's **immediate dominator** is its closest dominator except itself, IDOM(n)†

†IDOM(n) ≠ n, unless n is n_0, by convention.

Example

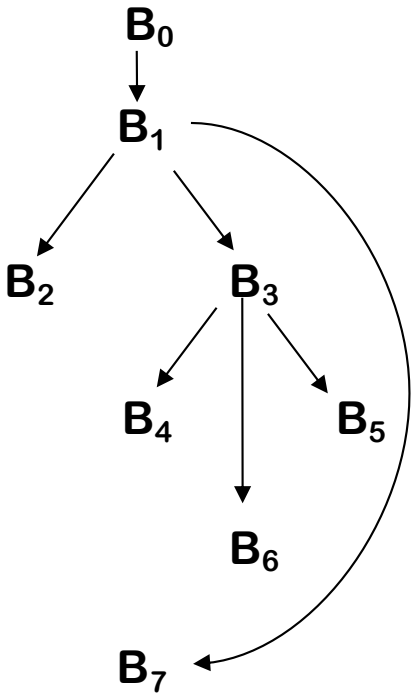
$$\text{DOM}(n) = \{ n \} \cup (\cap_{p \in preds(n)} \text{DOM}(p))$$



Contol Flow Graph

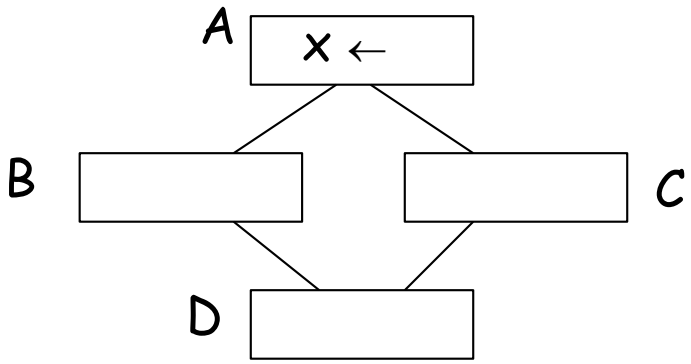
Progress of iterative solution for DOM

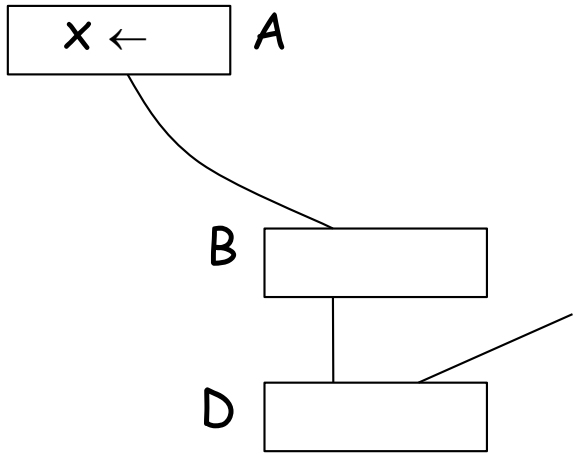
Iter-ation	DOM(*n*)							
	0	1	2	3	4	5	6	7
0	0	*N*	*N*	*N*	*N*	*N*	*N*	*N*
1	0	0,1	0,1,2	0,1,3	0,1,3,4	0,1,3,5	0,1,3,6	0,1,7
2	0	0,1	0,1,2	0,1,3	0,1,3,4	0,1,3,5	0,1,3,6	0,1,7



Dominance Tree

For a definition of x defined in a block n, it is enough to insert Ø-functions for that definition in the blocks that are right outside the dominated region of n.





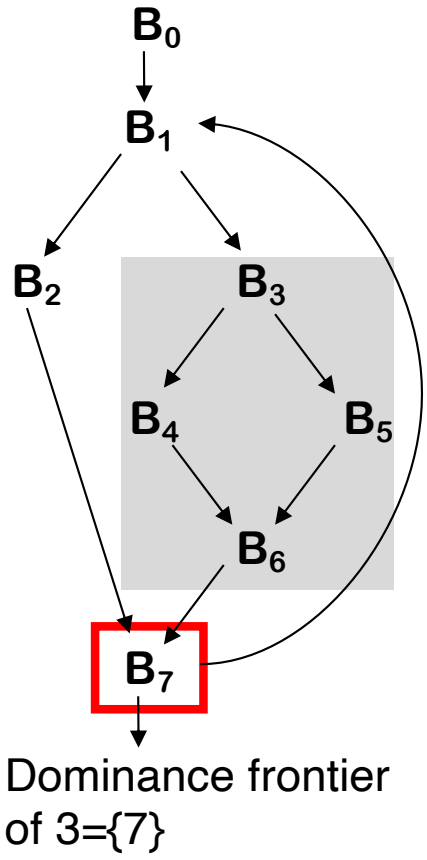
no insertion in D

$A \in$ **Dom(D)**

insert Ø-function in D

A $\notin$ Dom(D)
A $\in$ Dom(p(D))
i.e. D $\in$ DF(A)

Dominance Frontiers



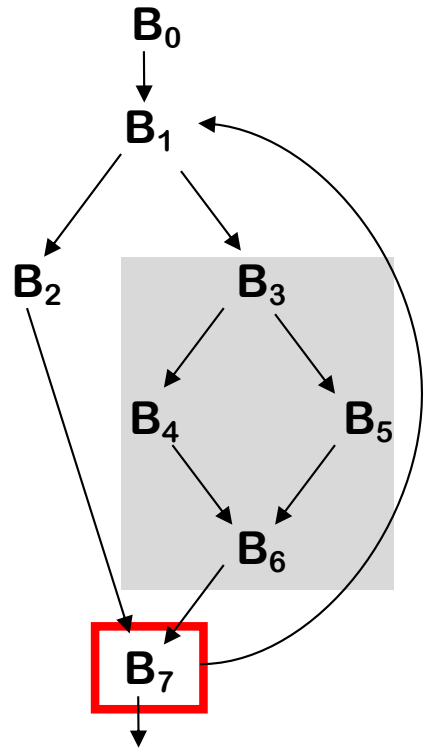
Dominance Frontiers

- **DF(n) is fringe just beyond the region n dominates**
- **m ∈DF(n) : iff n ∉(Dom(m) - {m}) but n ∈ DOM(p) for some p ∈ preds(m).**

i.e., n dominates p

i.e., n doesn't strictly dominate m

Dominance Frontiers



Dominance frontier of 3={7}

Dominance Frontiers

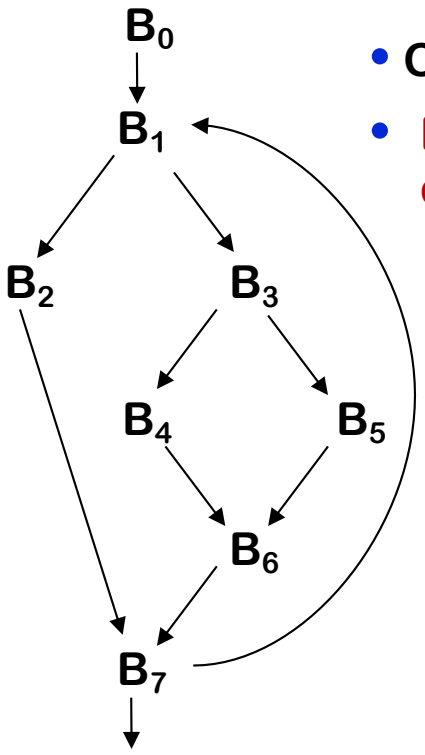
- **DF(n) is fringe just beyond the region n dominates**
- **m ∈DF(n) : iff n ∉(Dom(m) - {m}) but n ∈ DOM(p) for some p ∈ preds(m).**

i.e., n dominates p

i.e., n doesn't strictly dominate m

	0	1	2	3	4	5	6	7
DOM	0	0,1	0,1,2	0,1,3	0,1,3,4	0,1,3,5	0,1,3,6	0,1,7
DF	–	1	7	7	6	6	7	1

Computing Dominance Frontiers



CFG

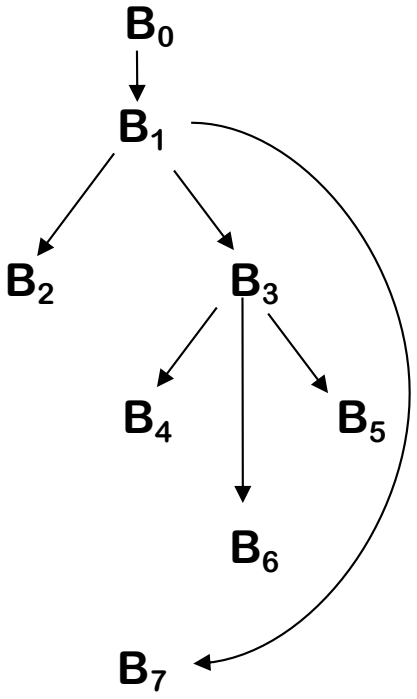
- Only join points are in DF(n) for some n
- Leads to a simple, "intuitive" algorithm for computing dominance frontiers

  For each join point x (i.e., |preds(x)| > 1)

  For each CFG predecessor of x

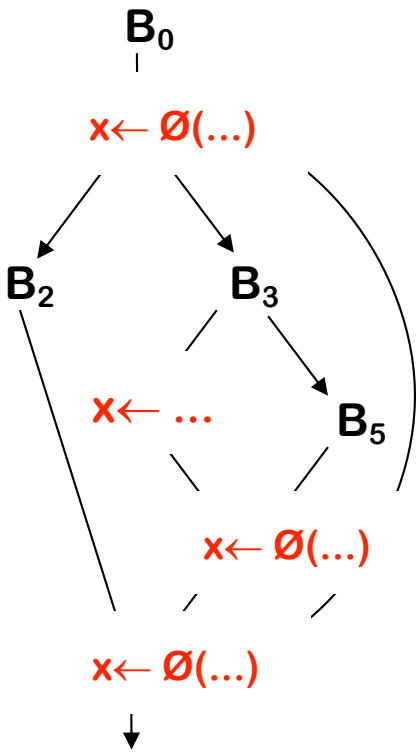
  Walk up to IDOM(x) in the dominator tree, adding x to DF(n) for each n in the walk except IDOM(x).

	0	1	2	3	4	5	6	7
DOM	0	0,1	0,1,2	0,1,3	0,1,3,4	0,1,3,5	0,1,3,6	0,1,7
DF	–	1	7	7	6	6	7	1



Dominance Tree

Example



- How to insert Ø-function for a specific definition?

	0	1	2	3	4	5	6	7
DOM	0	0,1	0,1,2	0,1,3	0,1,3,4	0,1,3,5	0,1,3,6	0,1,7
DF	–	1	7	7	6	6	7	1

- DF(4) is {6}, so ← in 4 forces Ø-function in 6
- ← in 6 forces Ø-function in DF(6) = {7}
- ← in 7 forces Ø-function in DF(7) = {1}
- ← in 1 forces Ø-function in DF(1) = Ø (halt)

Minimal SSA form

- Recall that the naïve technique to build SSA form presented earlier inserts Ø-functions for every variable at the beginning of every join point.

- Using dominance information, it is possible to do better, and compute minimal SSA form: for each definition of a variable x in a node n, insert a Ø-function for x in all nodes of DF(n).

- Notice that the inserted Ø-functions are definitions, and can therefore force the insertion of more Ø-functions.

Improving on minimal SSA

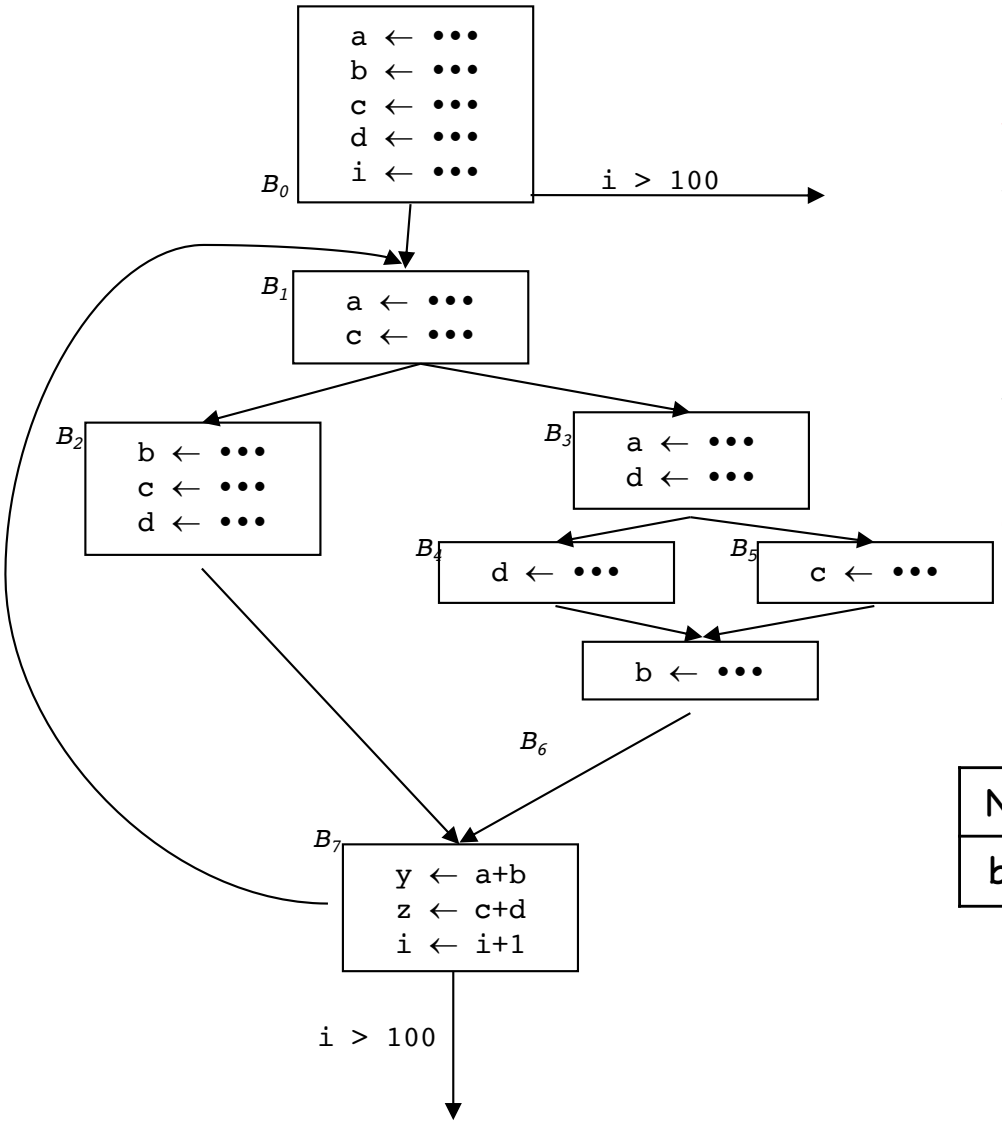
- Unfortunately, minimal SSA form is not necessarily optimal, and can contain dead Ø-functions. To solve that problem, improved techniques have been developed to build semi-pruned – which is still not optimal – and pruned SSA form.

Semi-pruned SSA

- Observation: a variable that is only live in a single node can never have a live Ø-function.

- Therefore, the minimal technique can be further refined by first computing the set of global names – defined as the names that are live across more than one node – and producing Ø-functions for these names only.

Step 1: Global Variables



- Globals = $\bigcup_{all\ n\ in\ CFG}$ UEVar(n)

UEVar(n): Upper exposed variables in block n, i.e., variables used before it is redefined in block n. (We have learned this in liveness analysis.)

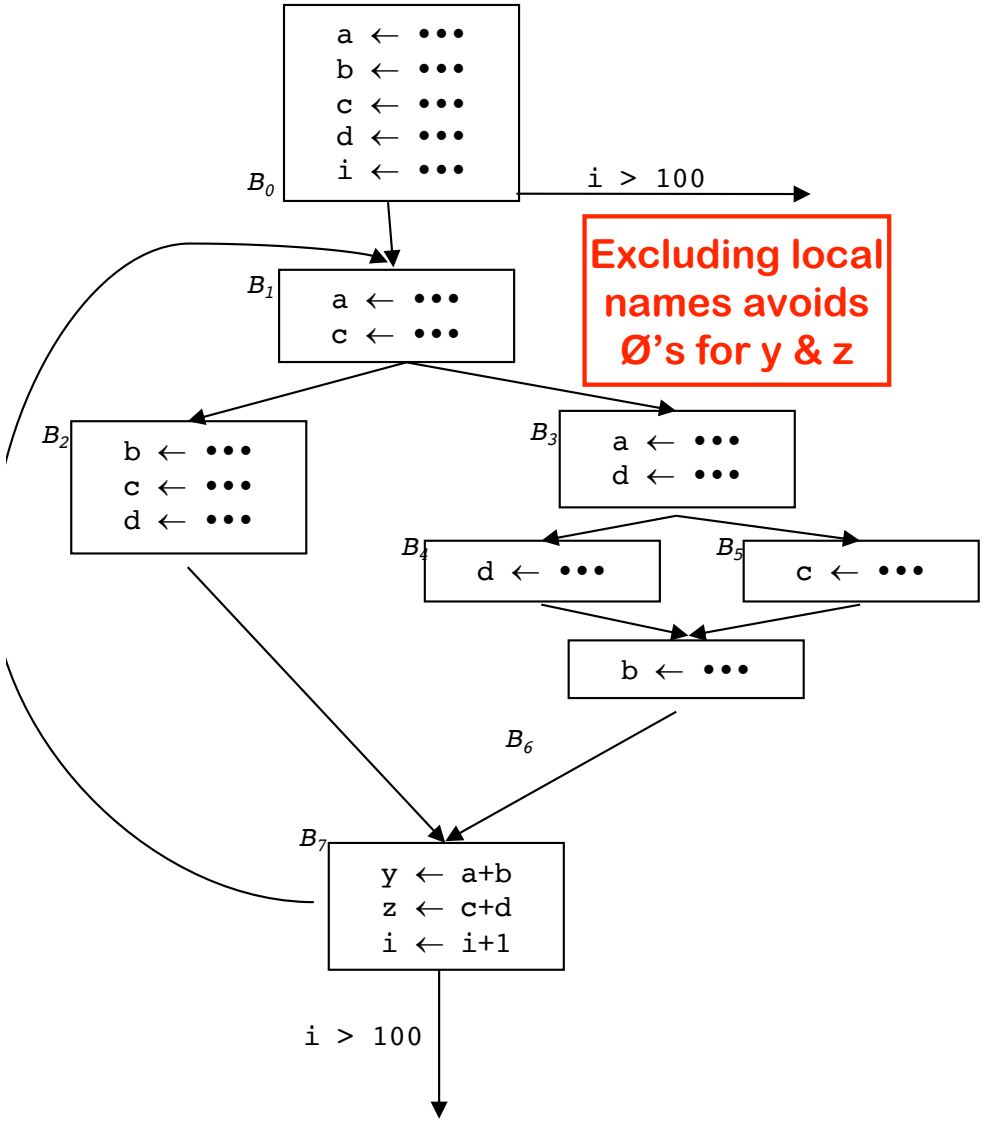
- Example

UEVAR (B7) = {a, b, c, d, i};
all others are empty set;
Globals = {a, b, c, d, i}, (y, z are local names)

- Get blocks where each of these Globals get defined

Names	a	b	c	d	i
blocks	0,1,3	0,2,6	0,1,2,5	0,2,3,4	0,7

Phase 2: inserting ϕ-functions



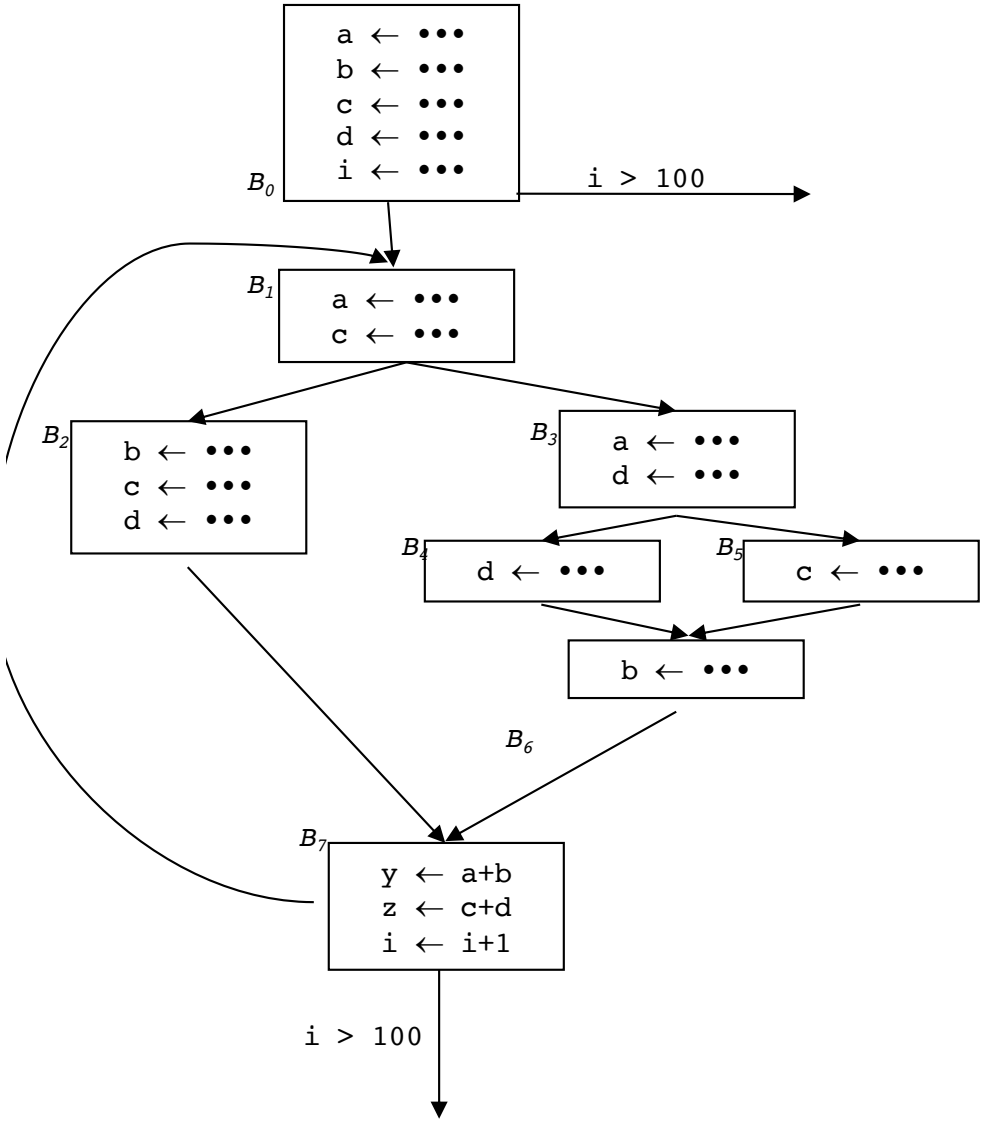
Excluding local names avoids Ø's for y & z

```
for each of the global name x
    work list = get all nodes in which x is defined
    for each node n in work list
        for each node m in DF(n)
            if (there is no ϕ-function for x in m)
                insert a ϕ-function for x to m
                work list = work list ∪ { m }
```

Names	a	b	c	d	i
blocks	0,1,3	0,2,6	0,1,2,5	0,2,3,4	0,7

Block	0	1	2	3	4	5	6	7
DF	-	1	7	7	6	6	7	1

Phase 2: inserting ϕ-functions

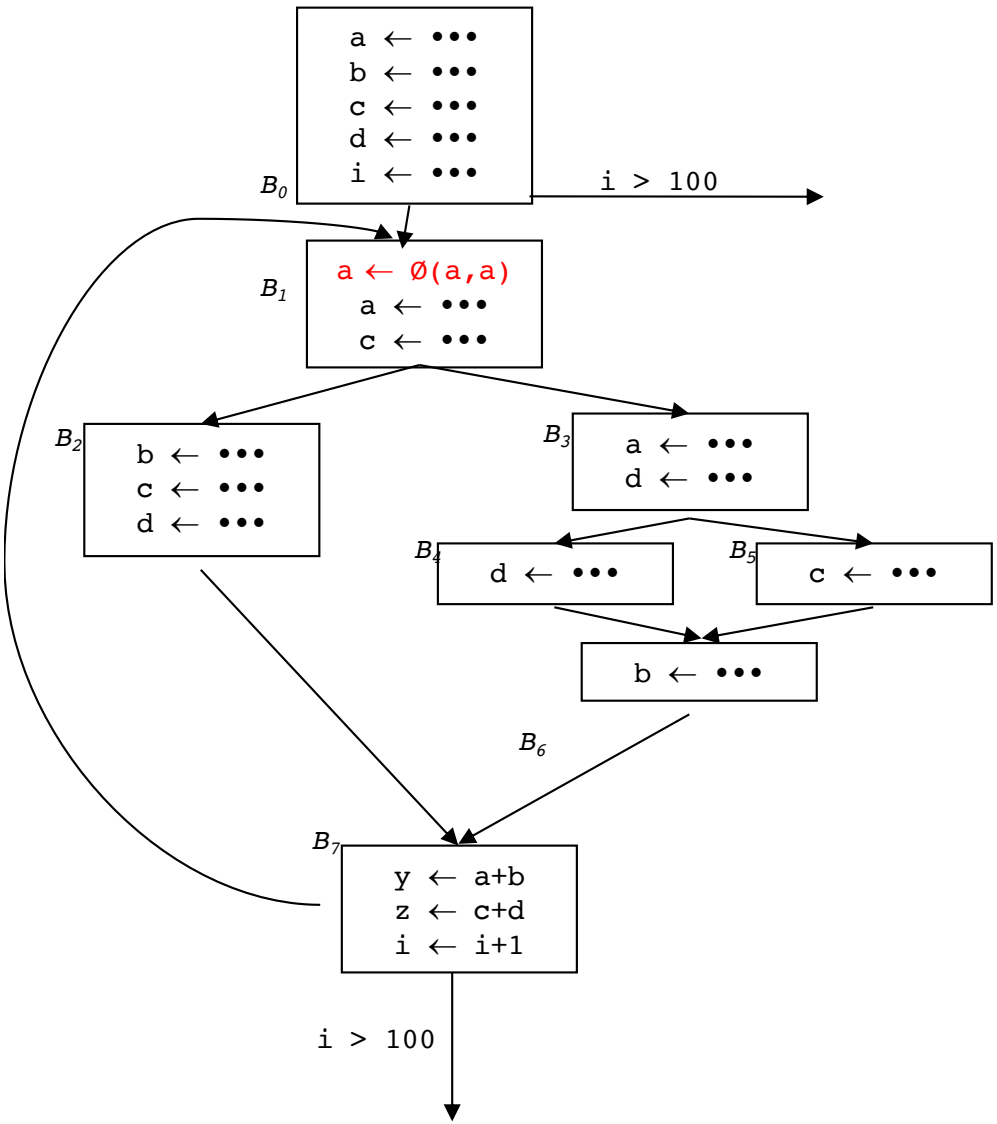



```
for each of the global name x
    work list = get all nodes in which x is defined
    for each node n in work list
        for each node m in DF(n)
            if (there is no ϕ-function for x in m)
                insert a ϕ-function for x to m
                work list = work list ∪ { m }
```

Names	a	b	c	d	i
blocks	0,1,3	0,2,6	0,1,2,5	0,2,3,4	0,7

Block	0	1	2	3	4	5	6	7
DF	-	1	7	7	6	6	7	1

Phase 2: inserting ϕ-functions

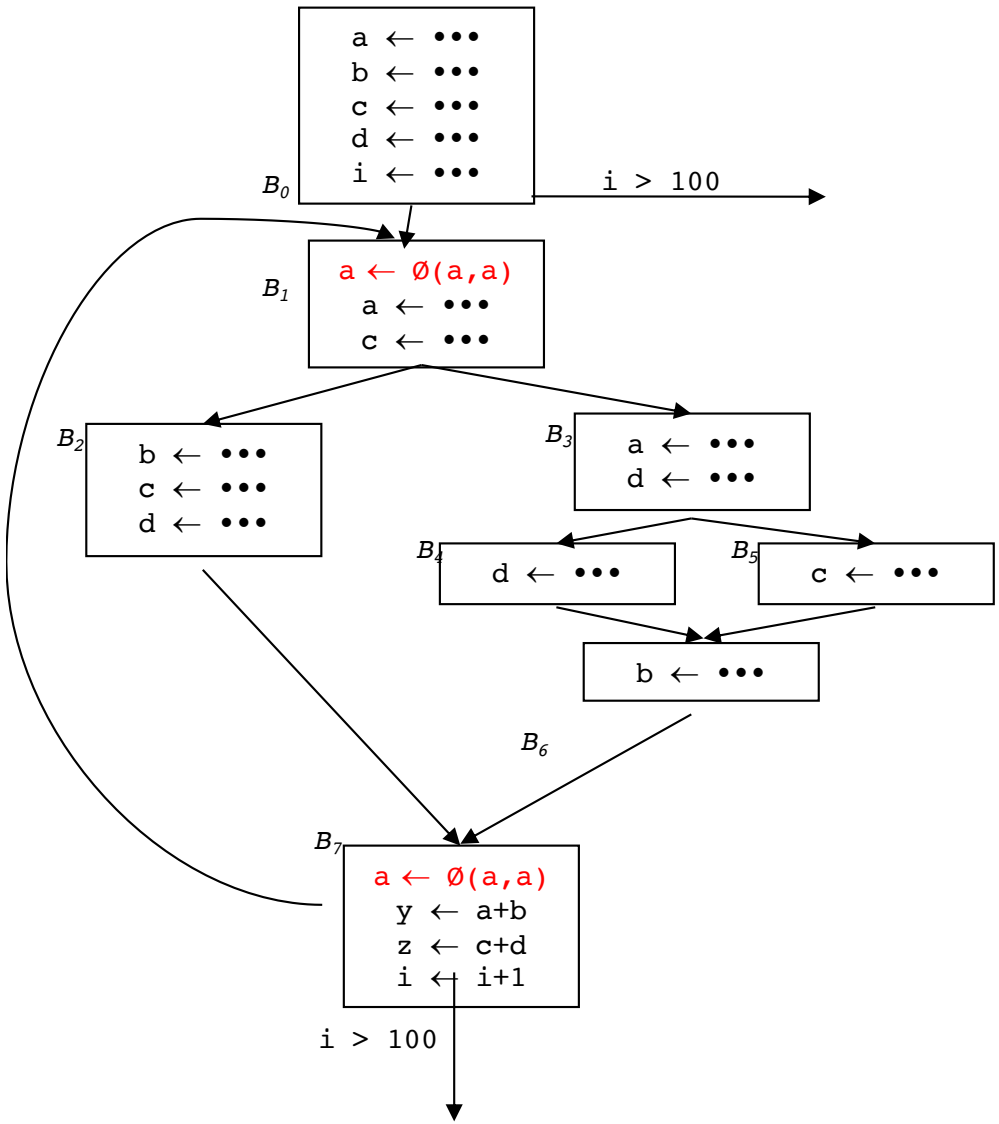


for each of the global name x
    work list = get all nodes in which x is defined
    for each node n in work list
        for each node m in DF(n)
            if (there is no ϕ-function for x in m)
                insert a ϕ-function for x to m
                work list = work list ∪ { m }

Names	a	b	c	d	i
blocks	0,1,3	0,2,6	0,1,2,5	0,2,3,4	0,7

Block	0	1	2	3	4	5	6	7
DF	-	1	7	7	6	6	7	1

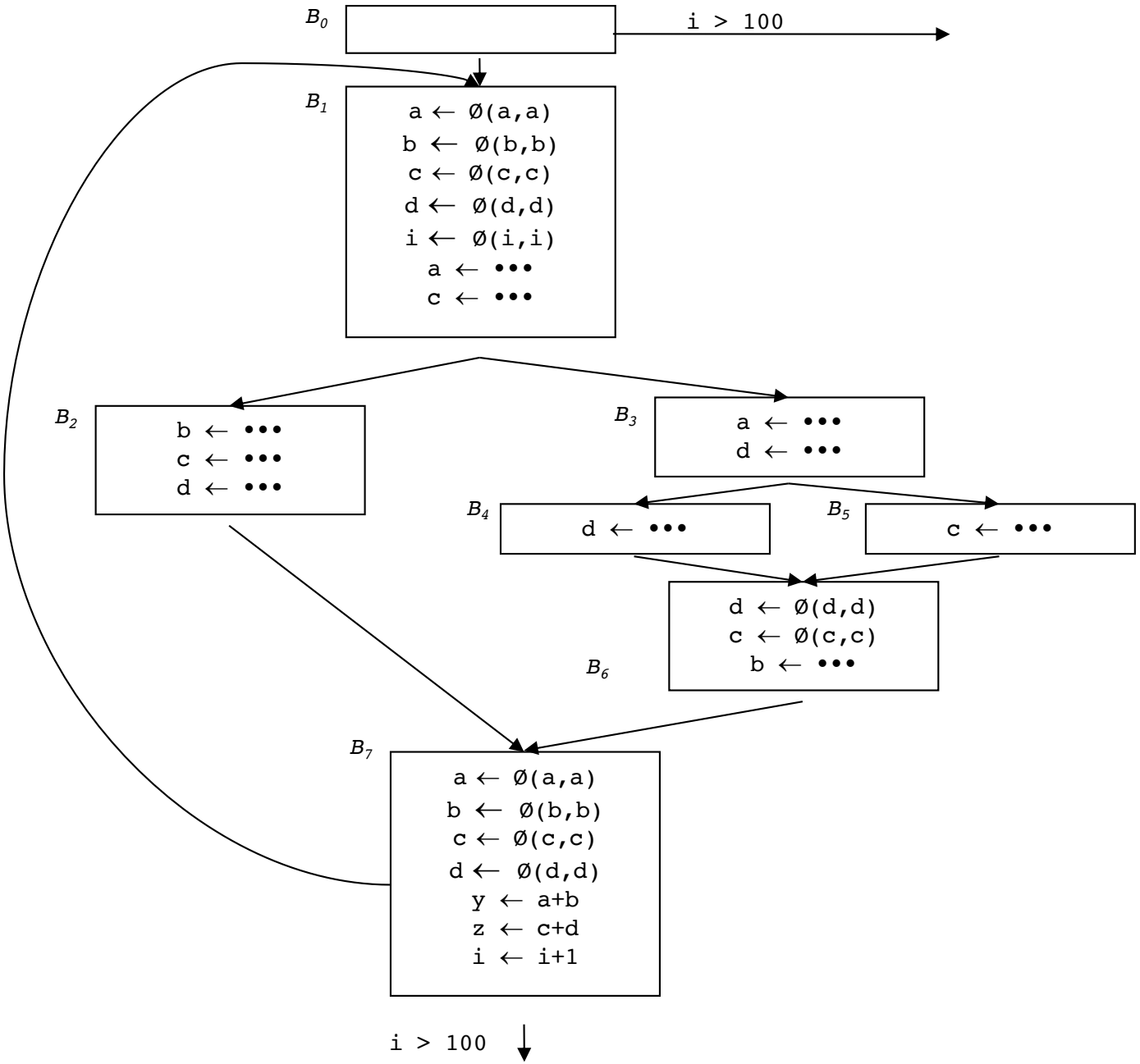
Phase 2: inserting ϕ-functions



for each of the global name x
    work list = get all nodes in which x is defined
    for each node n in work list
        for each node m in DF(n)
            if (there is no ϕ-function for x in m)
                insert a ϕ-function for x to m
                work list = work list ∪ { m }

Names	a	b	c	d	i
blocks	0,1,3	0,2,6	0,1,2,5	0,2,3,4	0,7

Block	0	1	2	3	4	5	6	7
DF	-	1	7	7	6	6	7	1


B0
i > 100
B1
a ← Ø(a,a)
b ← Ø(b,b)
c ← Ø(c,c)
d ← Ø(d,d)
i ← Ø(i,i)
a ← •••
c ← •••
B2
b ← •••
c ← •••
d ← •••
B3
a ← •••
d ← •••
B4
d ← •••
B5
c ← •••
d ← Ø(d,d)
c ← Ø(c,c)
b ← •••
B6
B7
a ← Ø(a,a)
b ← Ø(b,b)
c ← Ø(c,c)
d ← Ø(d,d)
y ← a+b
z ← c+d
i ← i+1
i > 100

Phase 3: renaming variables

Renaming is done by a pre-order traversal of the dominator tree, as follows:

for each node b in the dominator tree

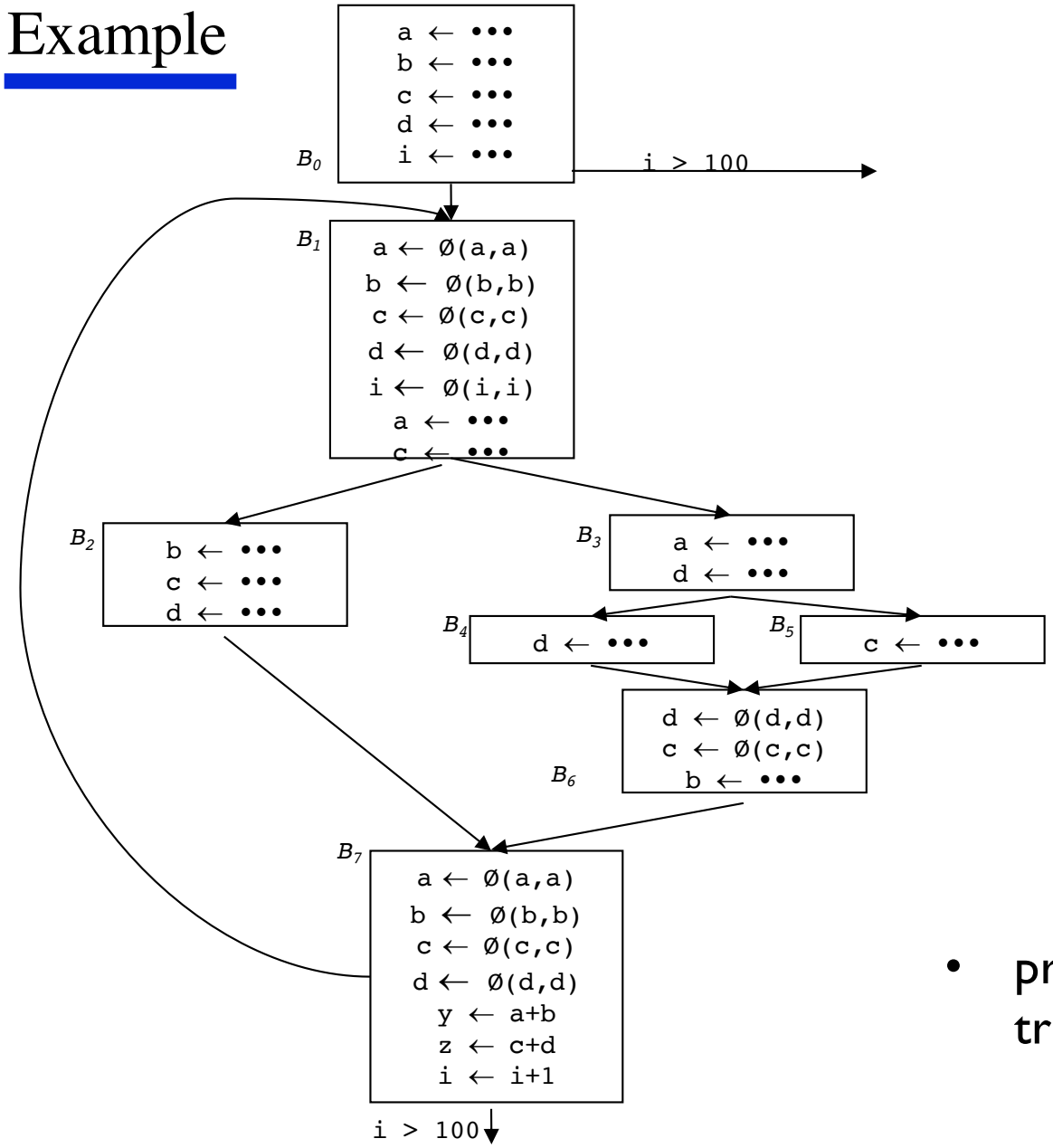
1. rename definitions and uses of variables in b
2. rename φ-functions parameters corresponding to b in all successors of n in the CFG.

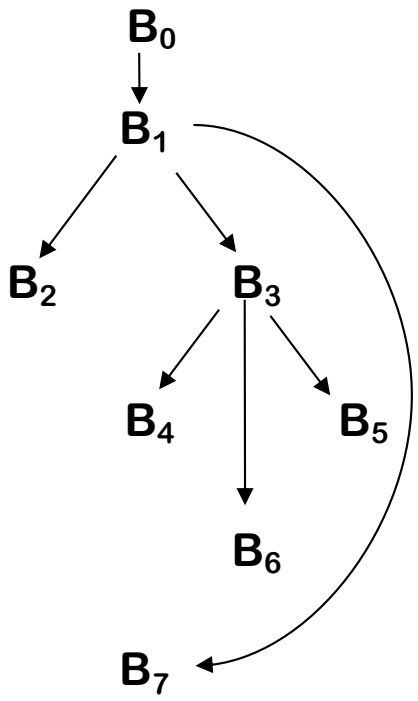
One possible Implementation via a set of stacks and counters.

1. Get the root node n_0 of the CFG
2. **Call Rename(n_0)**

```
Rename(b)
    for each Ø-function in b, x ¬ Ø(…)
        rename x as NewName(x)

    for each operation "x ¬ y op z" in b
        rewrite y as top(stack[y])
        rewrite z as top(stack[z])
        rewrite x as NewName(x)

    for each successor of b in the CFG
        rewrite appropriate Ø parameters

    for each successor s of b in dom. tree
        Rename(s)

    for each operation "x ¬ y op z" in b
        pop(stack[x])
```

Example

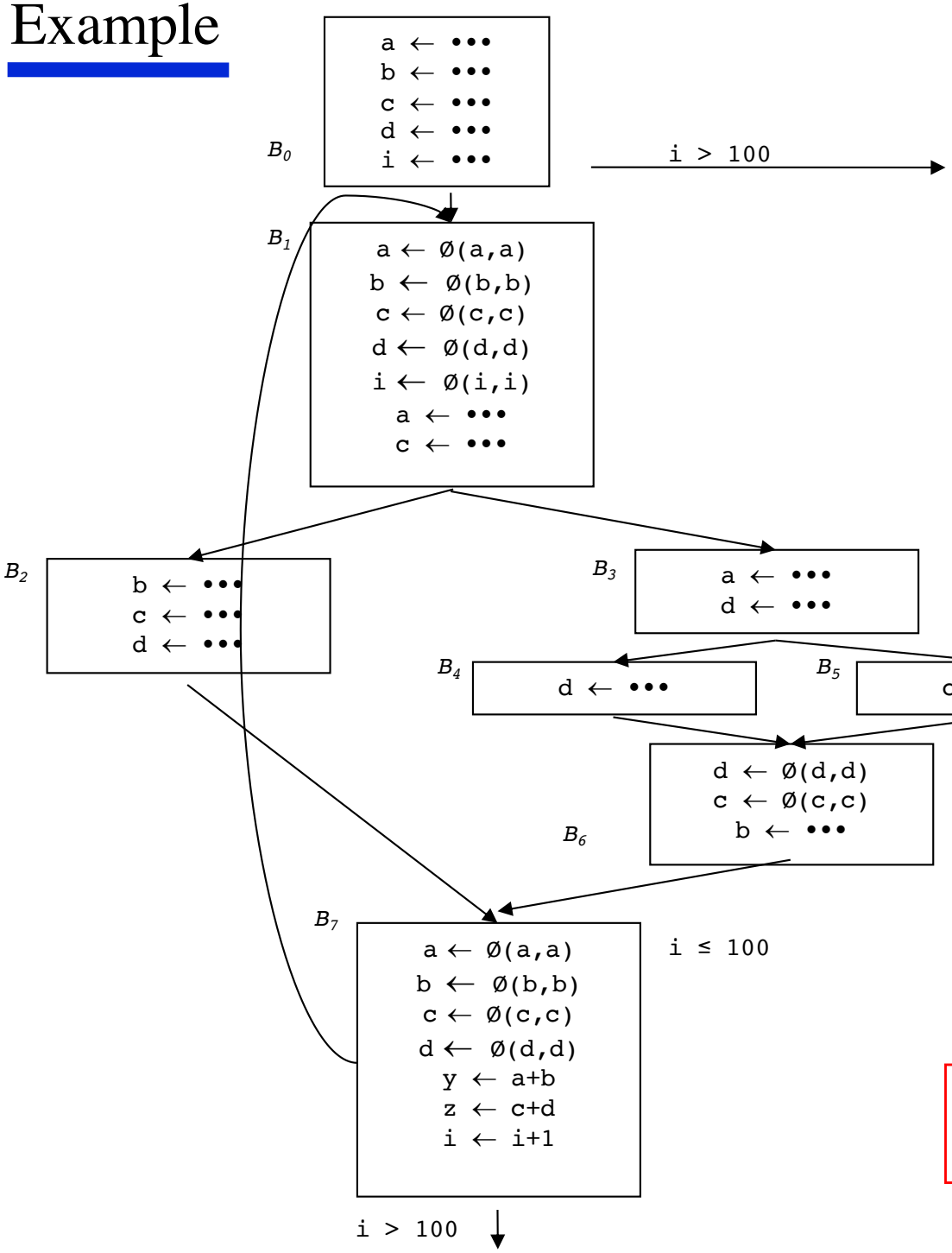




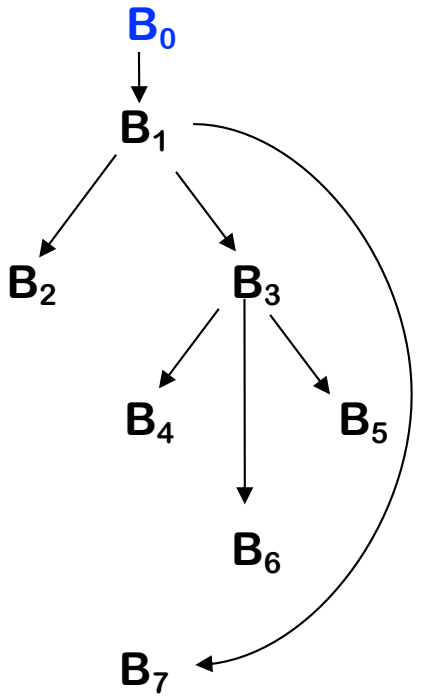
Dominance Tree

- pre-order traversal of the dominator tree: {0,1,2,3,4,5,6,7}

Example



Dominance Tree



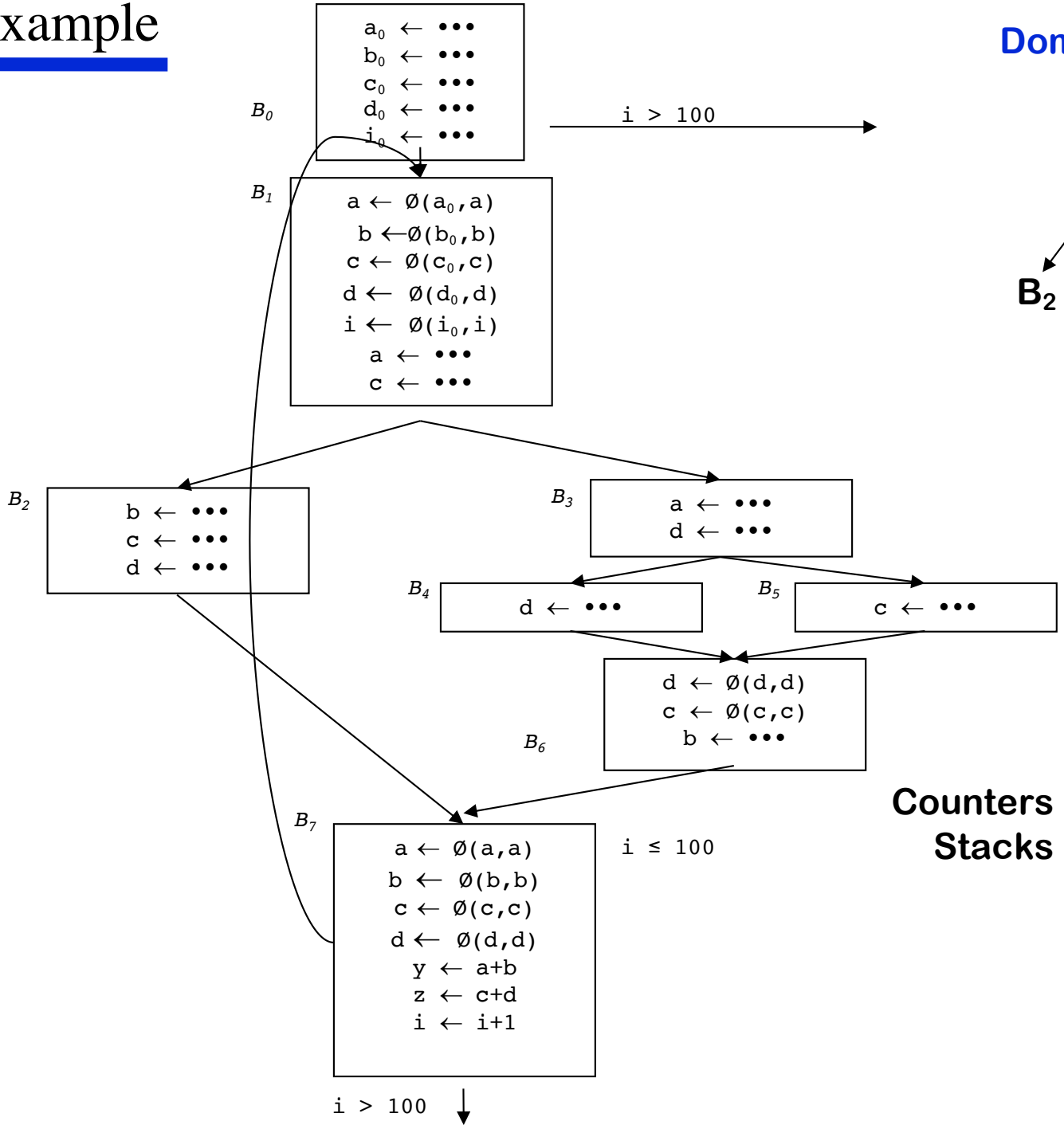
Before processing B_0

	a	b	c	d	i
Counters	0	0	0	0	0

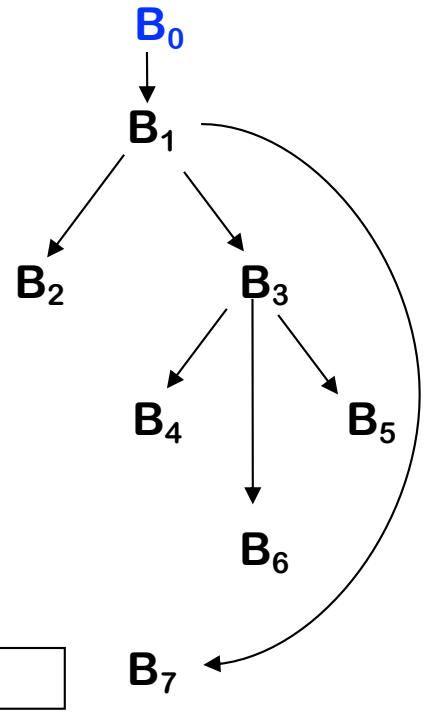
Stacks

- stacks for uses of variables
- counters for new definitions

Example



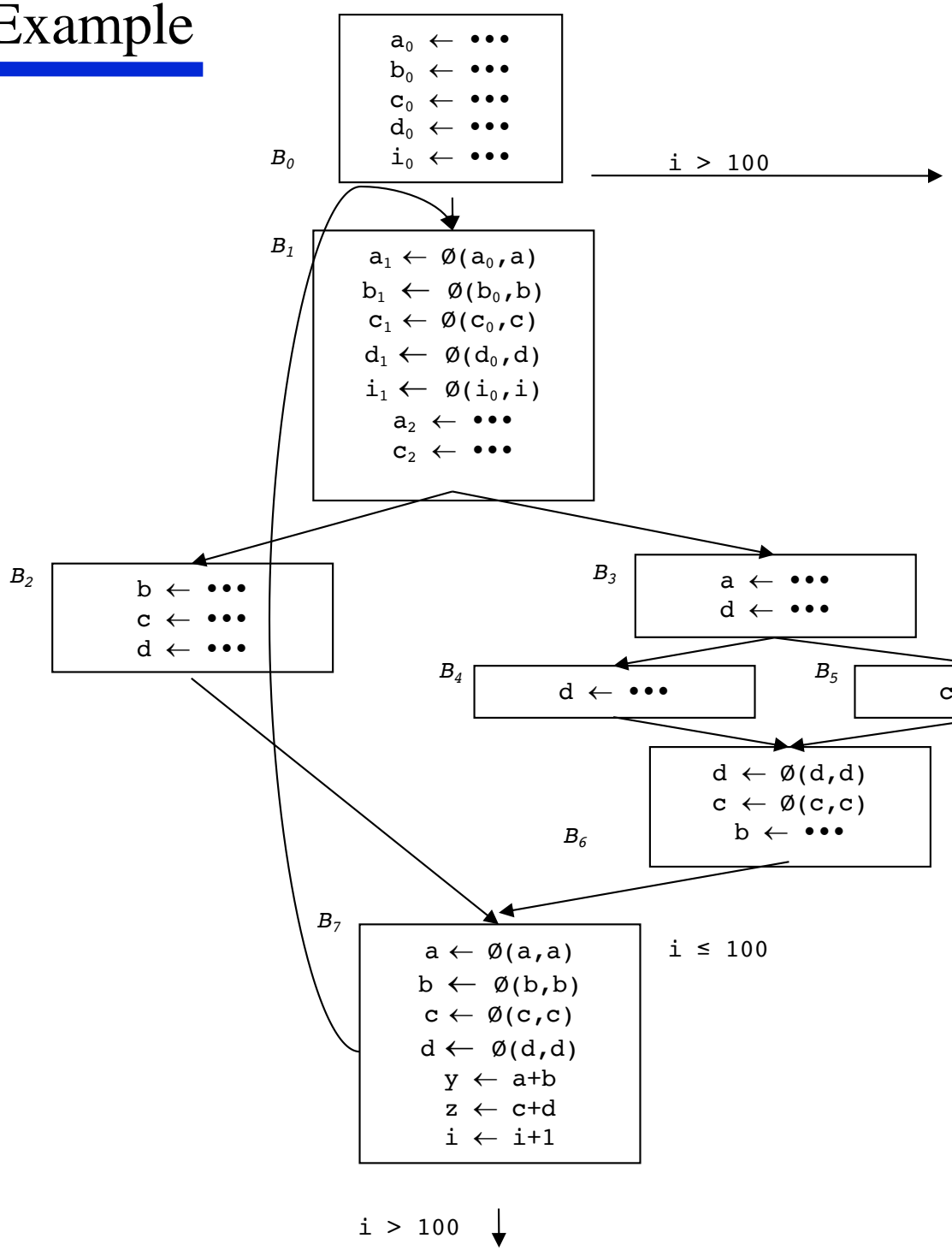
Dominance Tree



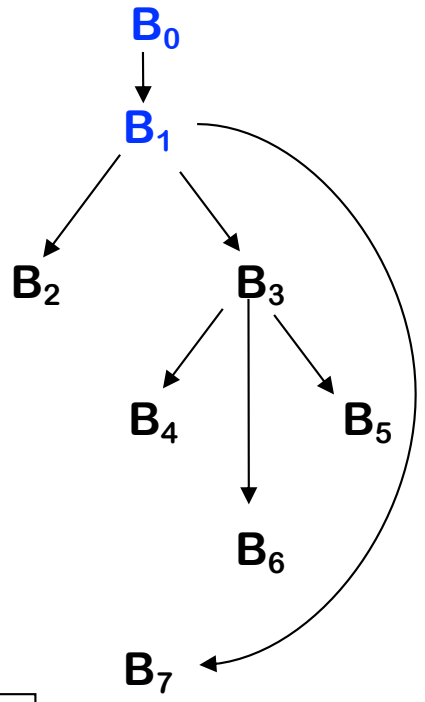
End of B_0

	a	b	c	d	i
Counters	1	1	1	1	1
Stacks	a_0	b_0	c_0	d_0	i_0

Example



Dominance Tree

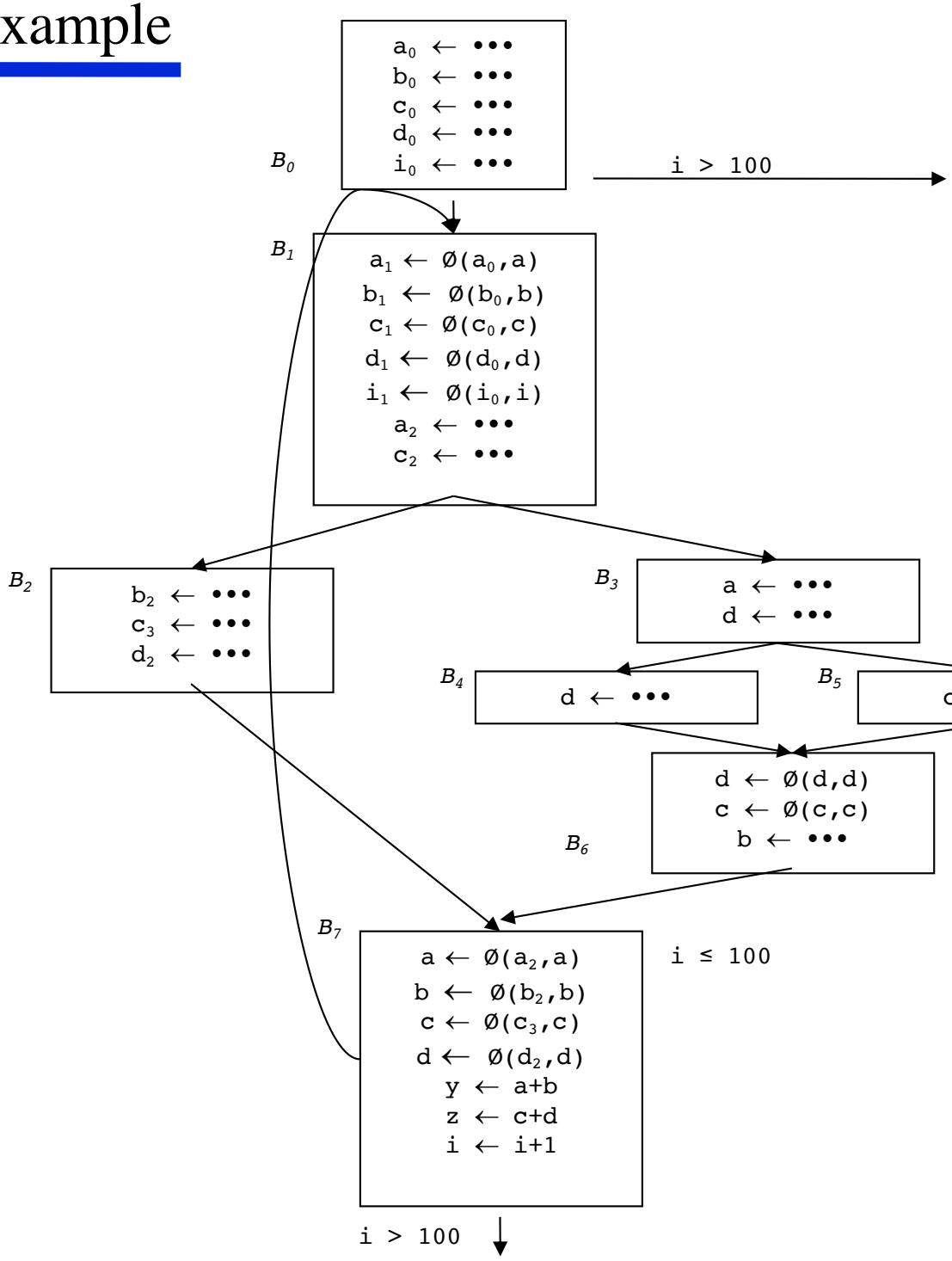


End of B_1

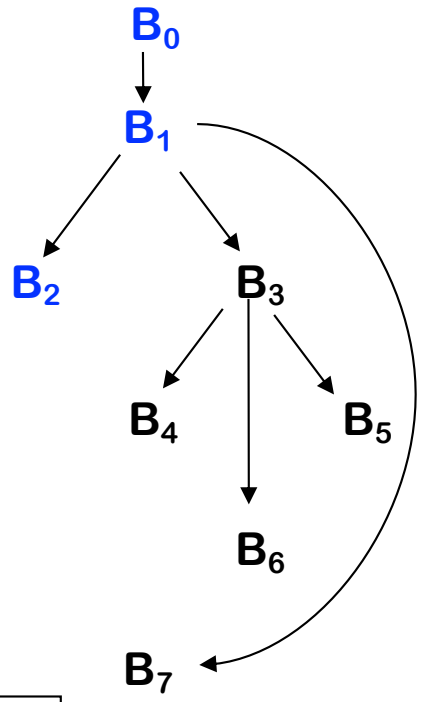
Counters / Stacks

a	b	c	d	i
3	2	3	2	2
a_0	b_0	c_0	d_0	i_0
a_1	b_1	c_1	d_1	i_1
a_2		c_2		

Example



Dominance Tree

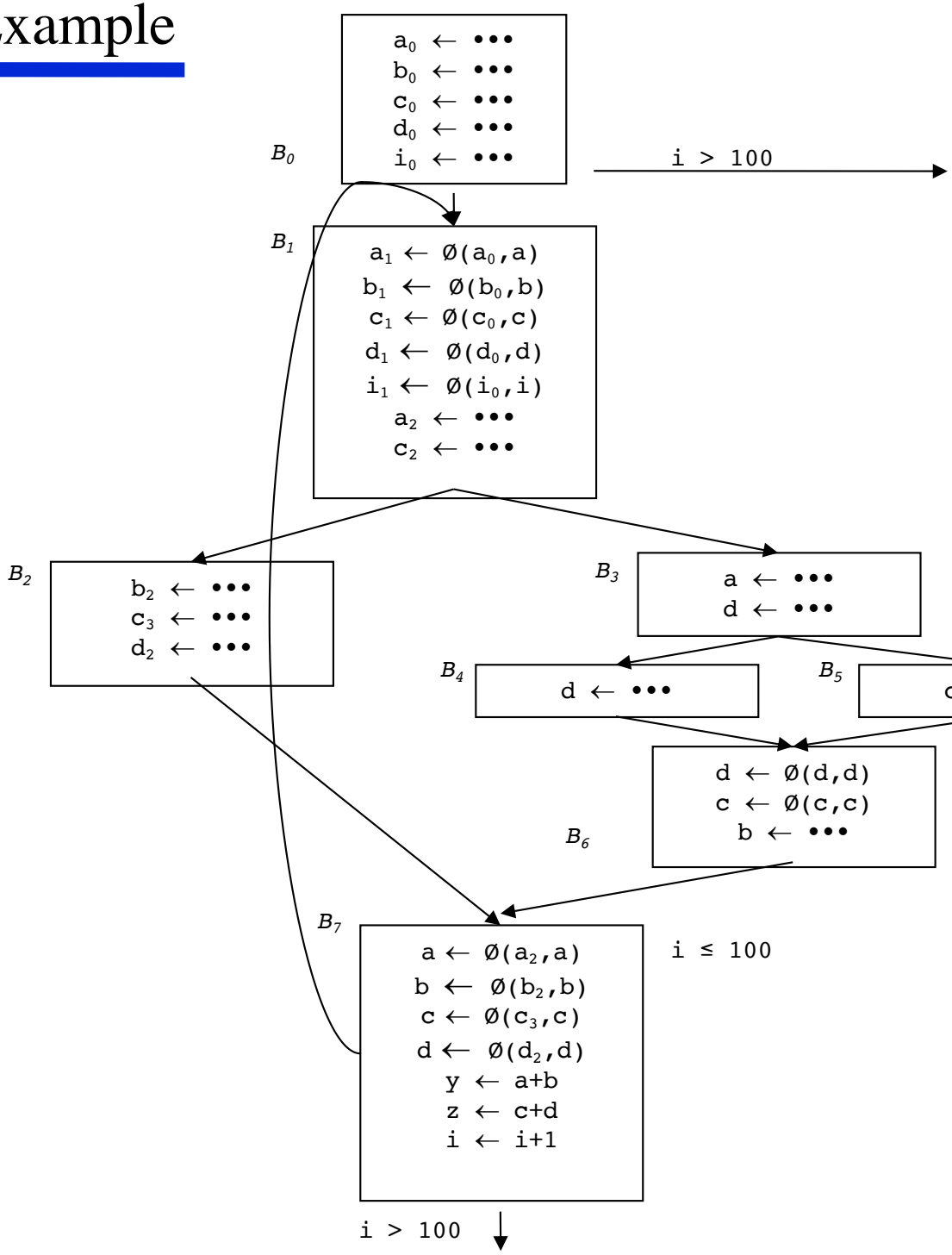


End of B_2

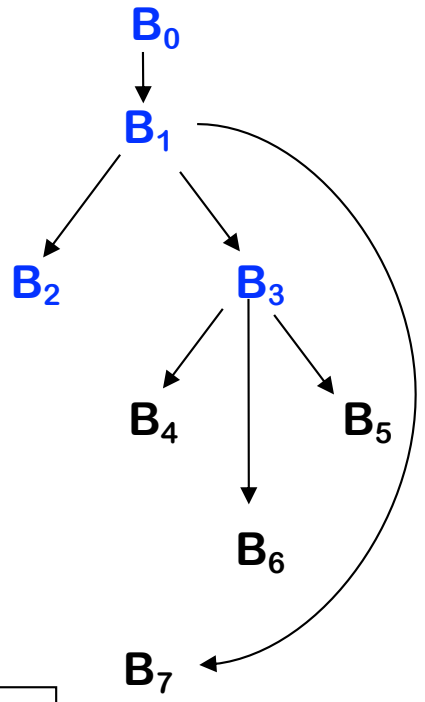
Counters
Stacks

a	b	c	d	i
3	3	4	3	2
a_0	b_0	c_0	d_0	i_0
a_1	b_1	c_1	d_1	i_1
a_2	b_2	c_2	d_2	
		c_3		

Example



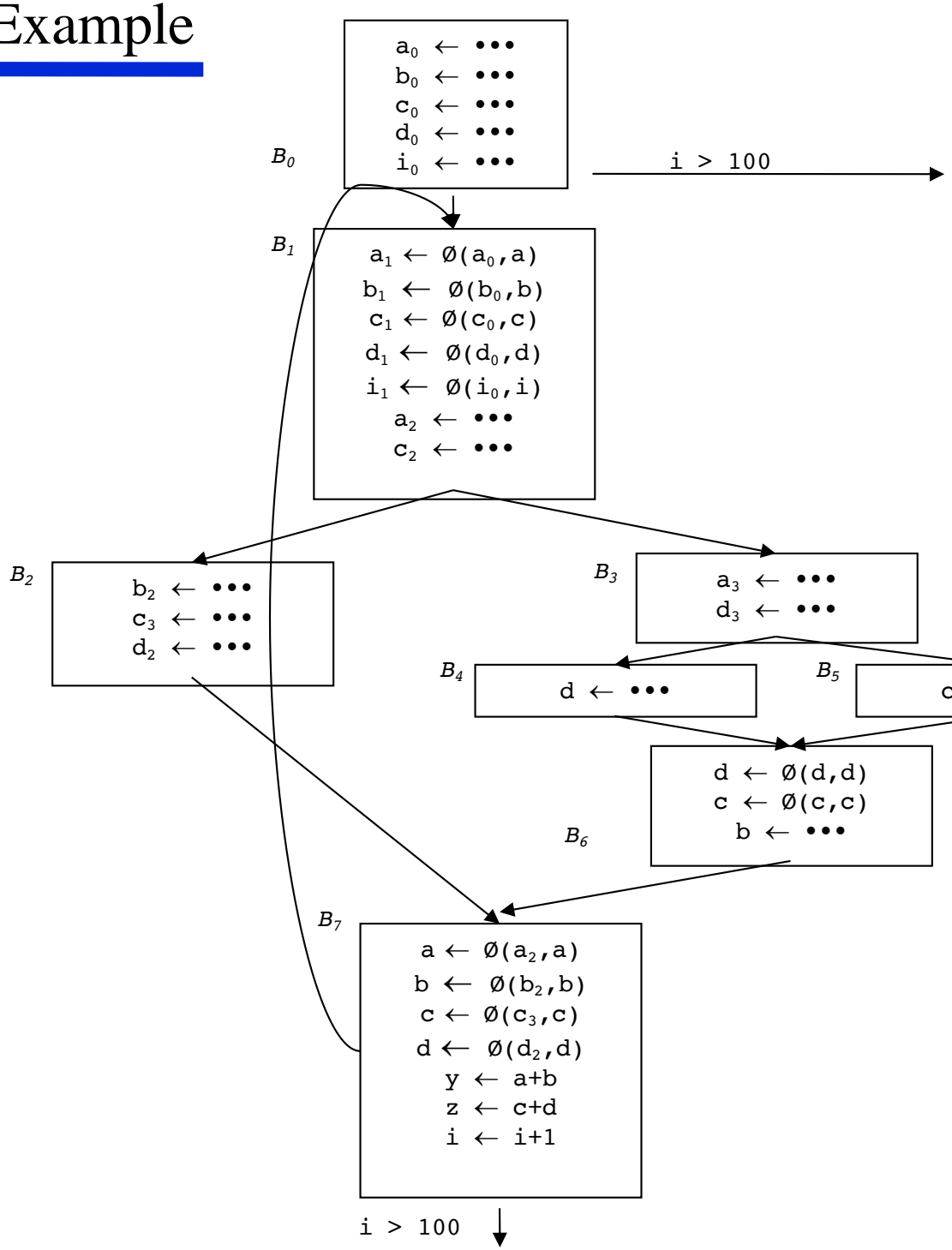
Dominance Tree



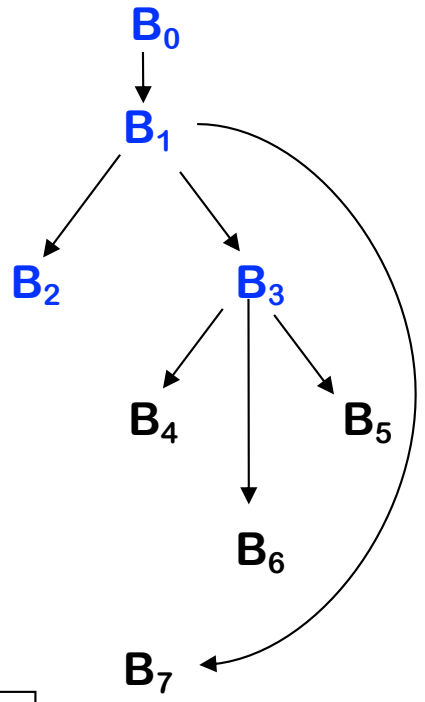
Before starting B_3

	a	b	c	d	i
Counters	3	3	4	3	2
Stacks	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2		c_2		

Example



Dominance Tree

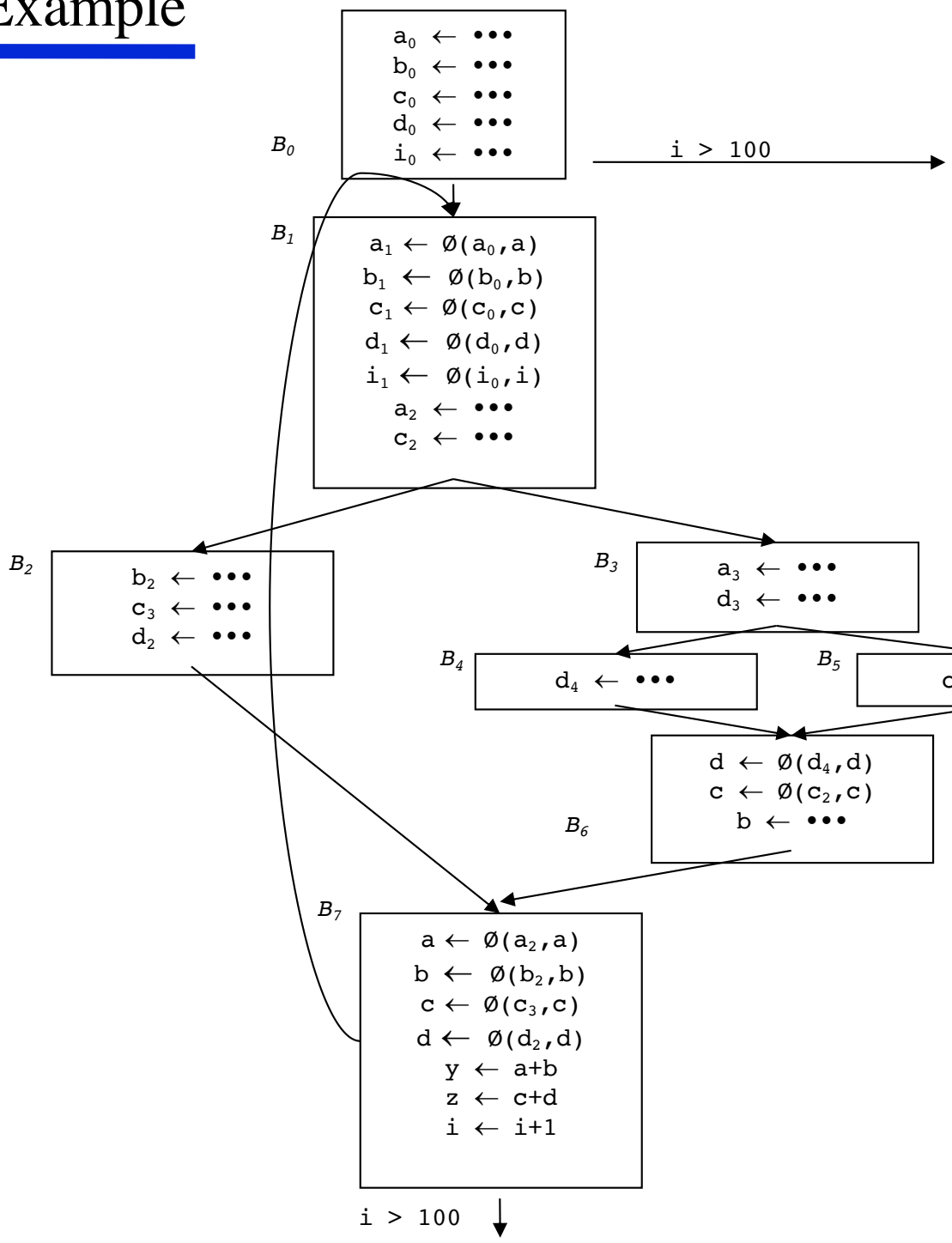


End of B_3

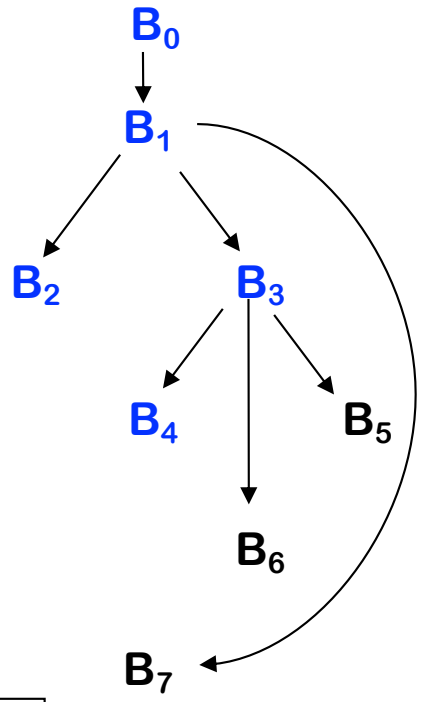
Counters / Stacks

a	b	c	d	i
4	3	4	4	2
a_0	b_0	c_0	d_0	i_0
a_1	b_1	c_1	d_1	i_1
a_2		c_2	d_3	
a_3				

Example



Dominance Tree

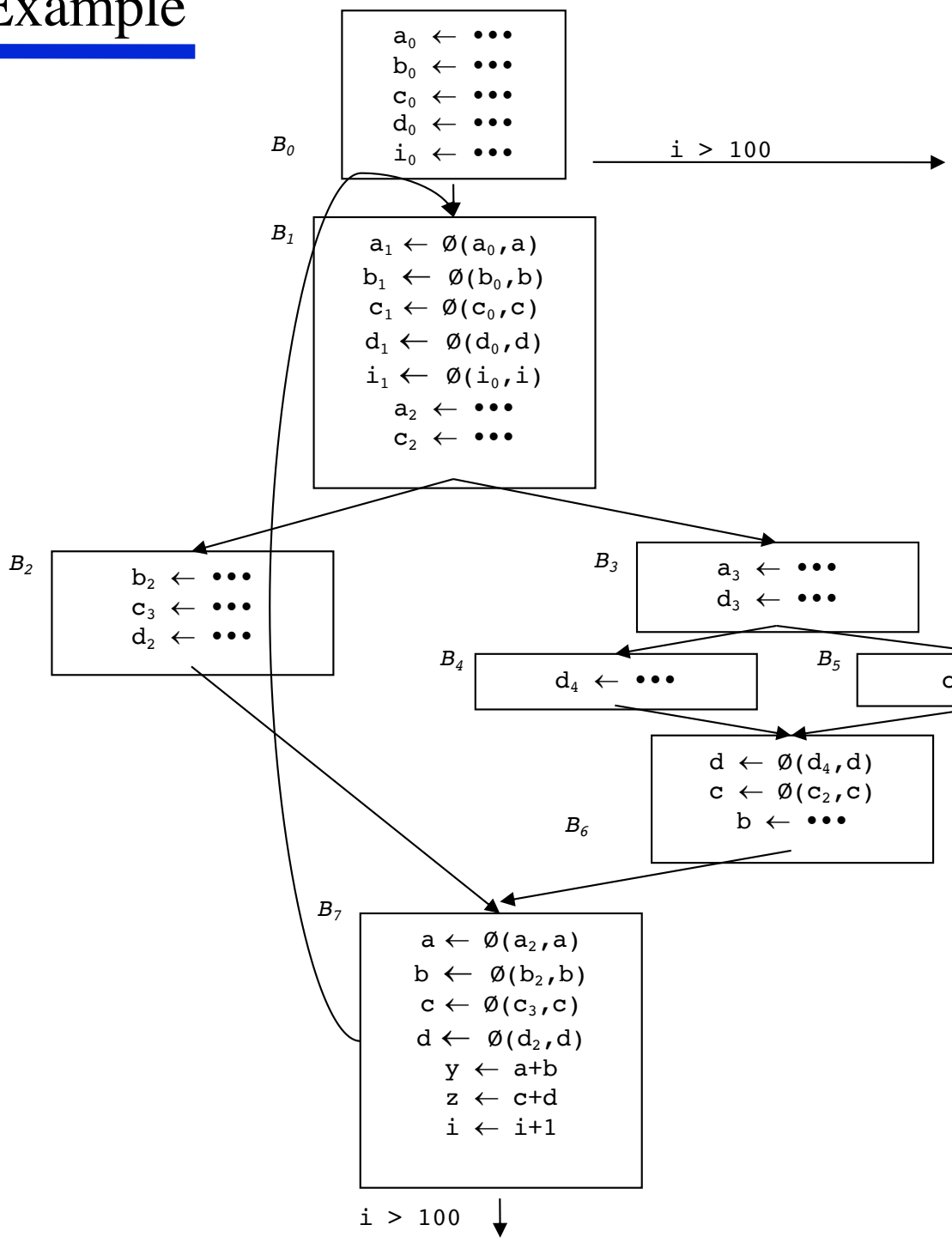


End of B_4

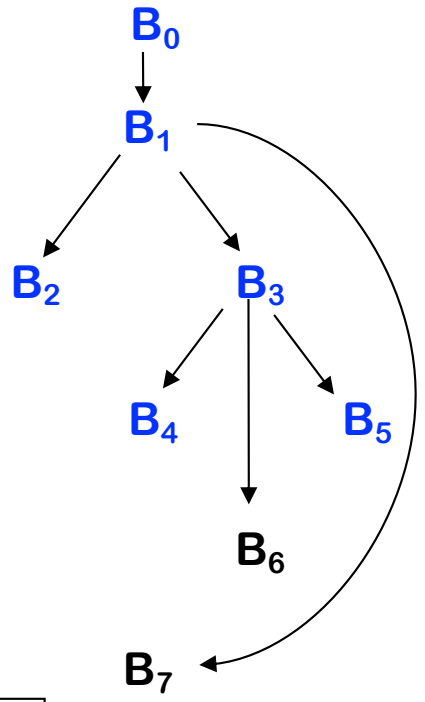
Counters
Stacks

a	b	c	d	i
4	3	4	5	2
a_0	b_0	c_0	d_0	i_0
a_1	b_1	c_1	d_1	i_1
a_2		c_2	d_3	
a_3			d_4	

Example



Dominance Tree

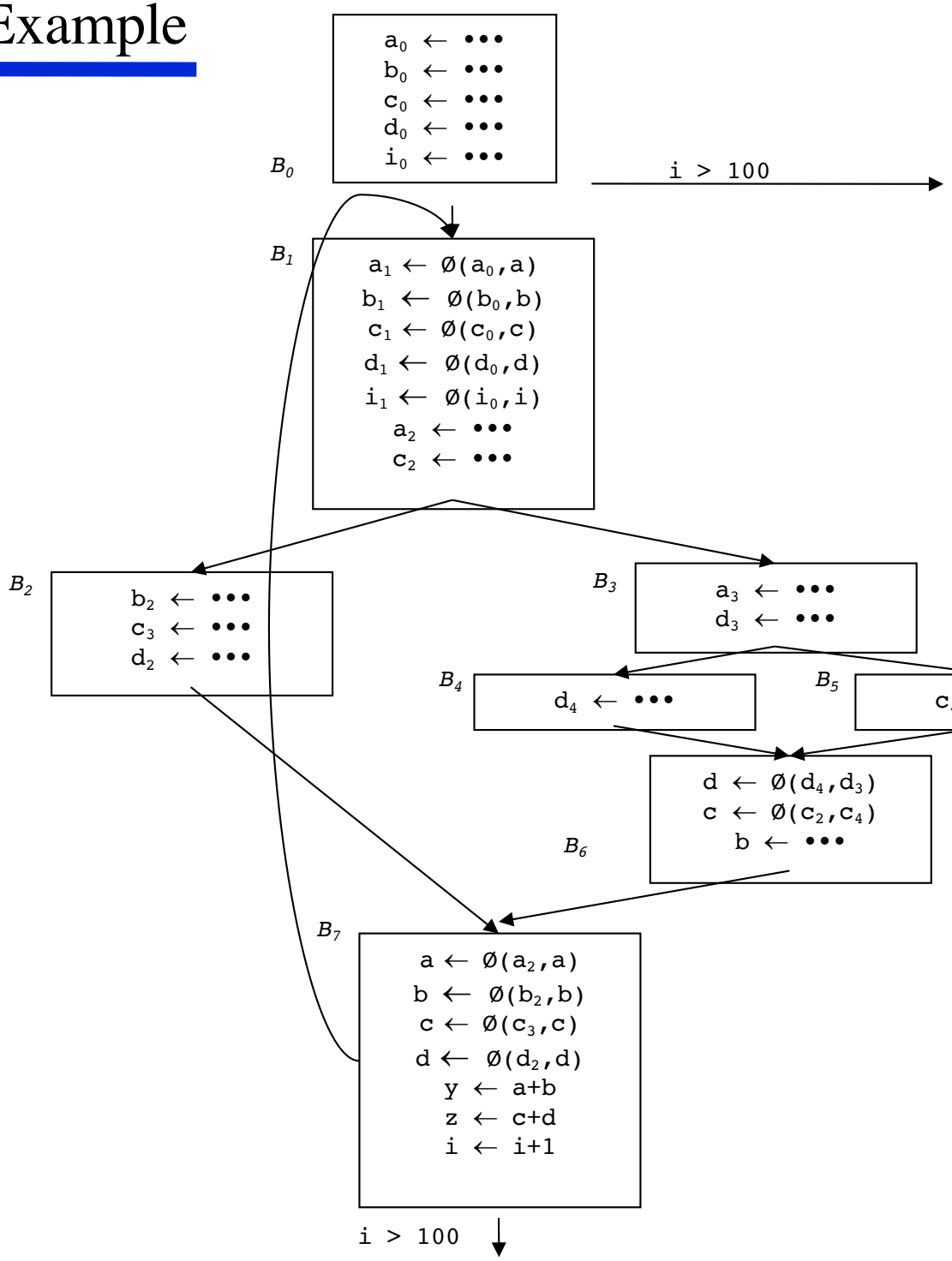


Before processing B_5

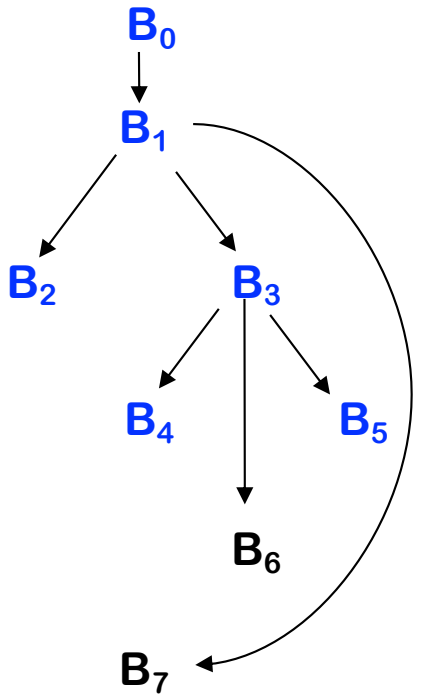
Counters
Stacks

a	b	c	d	i
4	3	4	5	2
a_0	b_0	c_0	d_0	i_0
a_1	b_1	c_1	d_1	i_1
a_2		c_2	d_3	
a_3				

Example



Dominance Tree

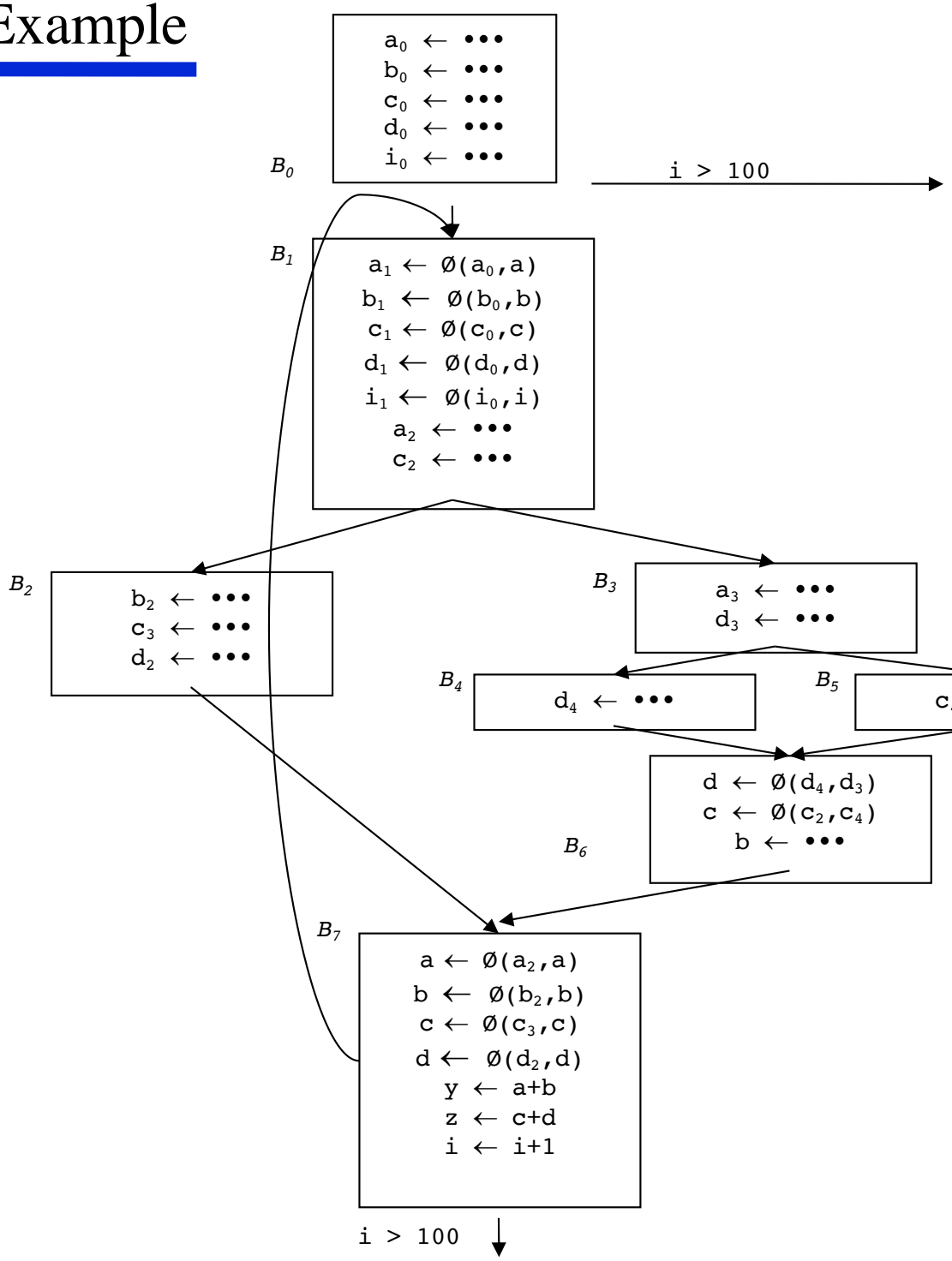


End of B_5

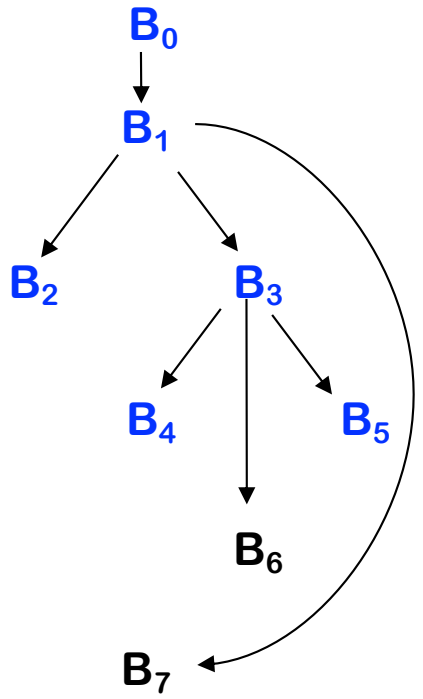
Counters
Stacks

a	b	c	d	i
4	3	5	5	2
a_0	b_0	c_0	d_0	i_0
a_1	b_1	c_1	d_1	i_1
a_2		c_2	d_3	
a_3		c_4		

Example



Dominance Tree



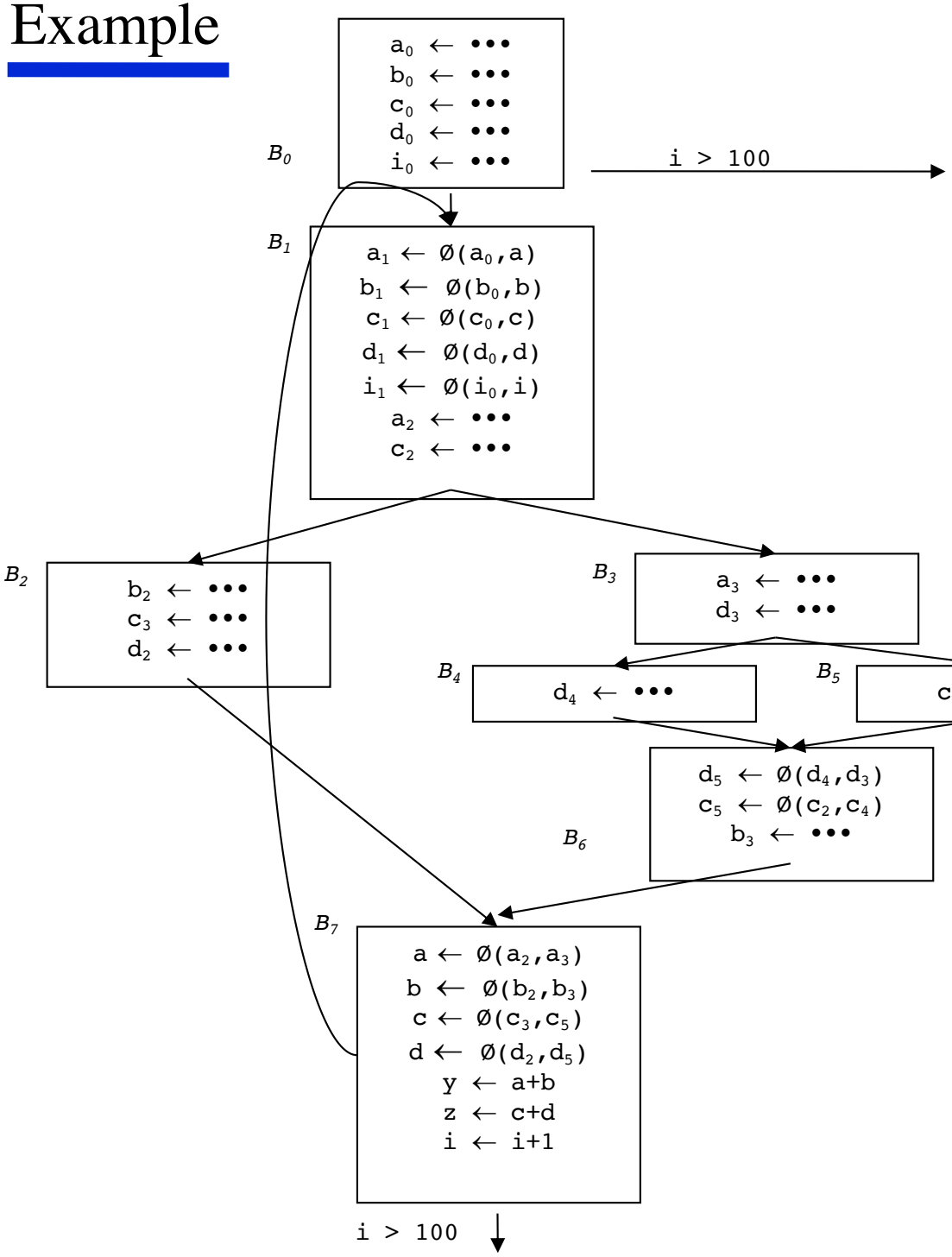
Before B_6

Counters

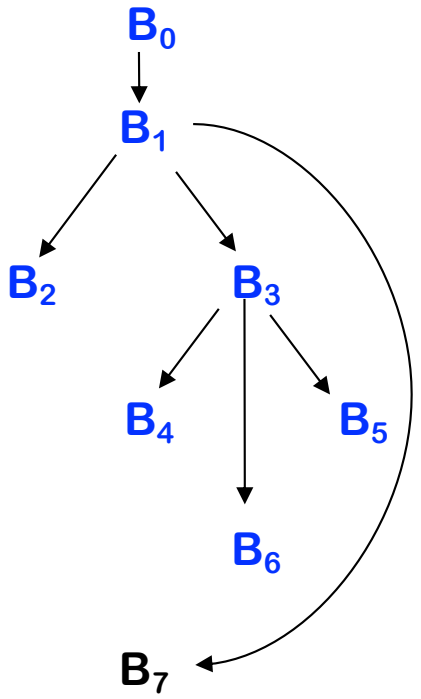
Stacks

a	b	c	d	i
4	3	5	5	2
a_0	b_0	c_0	d_0	i_0
a_1	b_1	c_1	d_1	i_1
a_2		c_2	d_3	
a_3				

Example



Dominance Tree

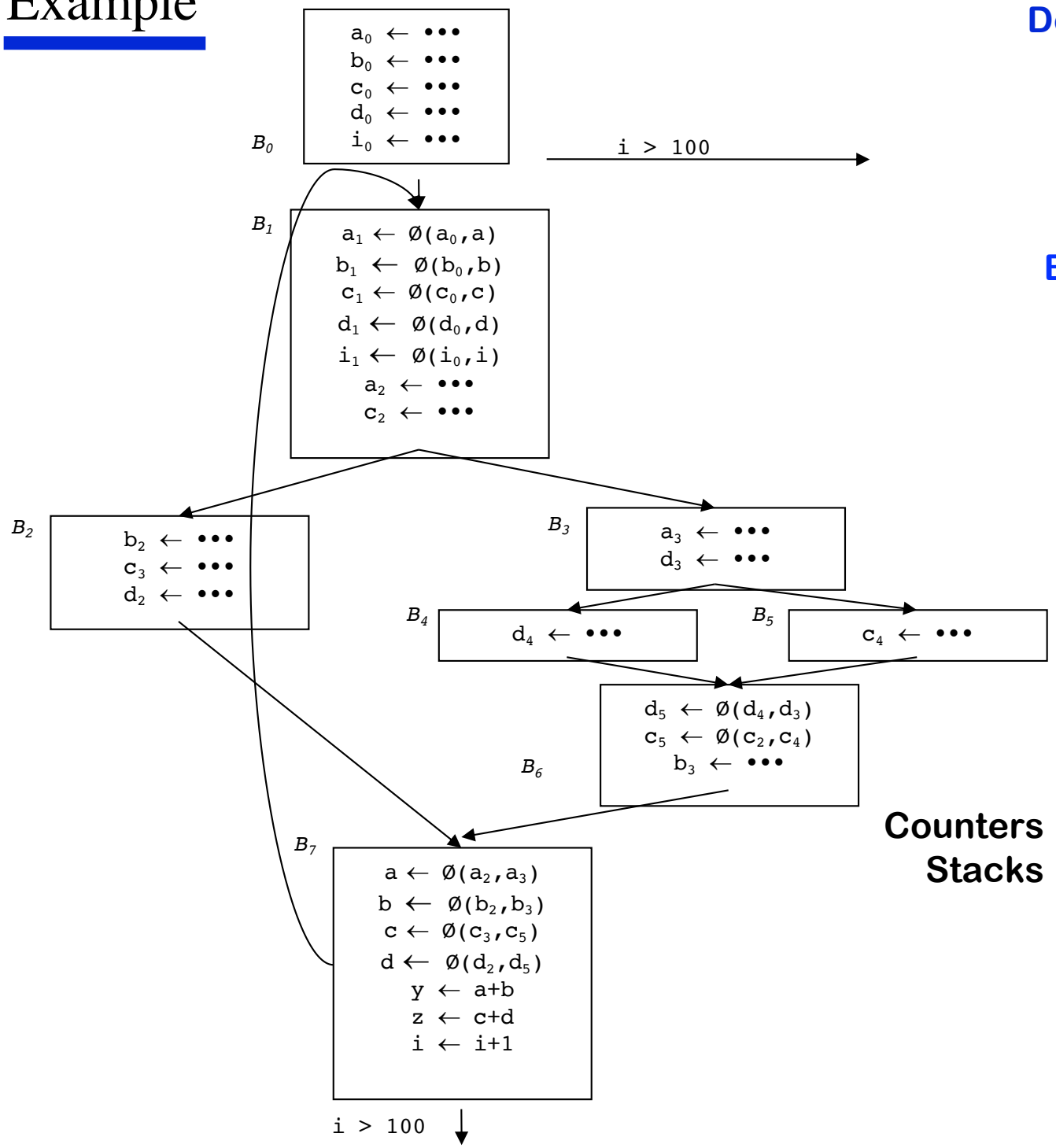


End of B_6

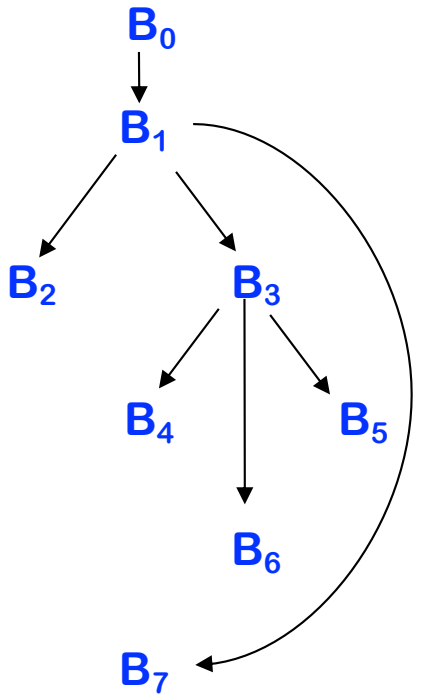
Counters
Stacks

a	b	c	d	i
4	4	6	6	2
a_0	b_0	c_0	d_0	i_0
a_1	b_1	c_1	d_1	i_1
a_2	b_3	c_2	d_3	
a_3		c_5	d_5	

Example



Dominance Tree

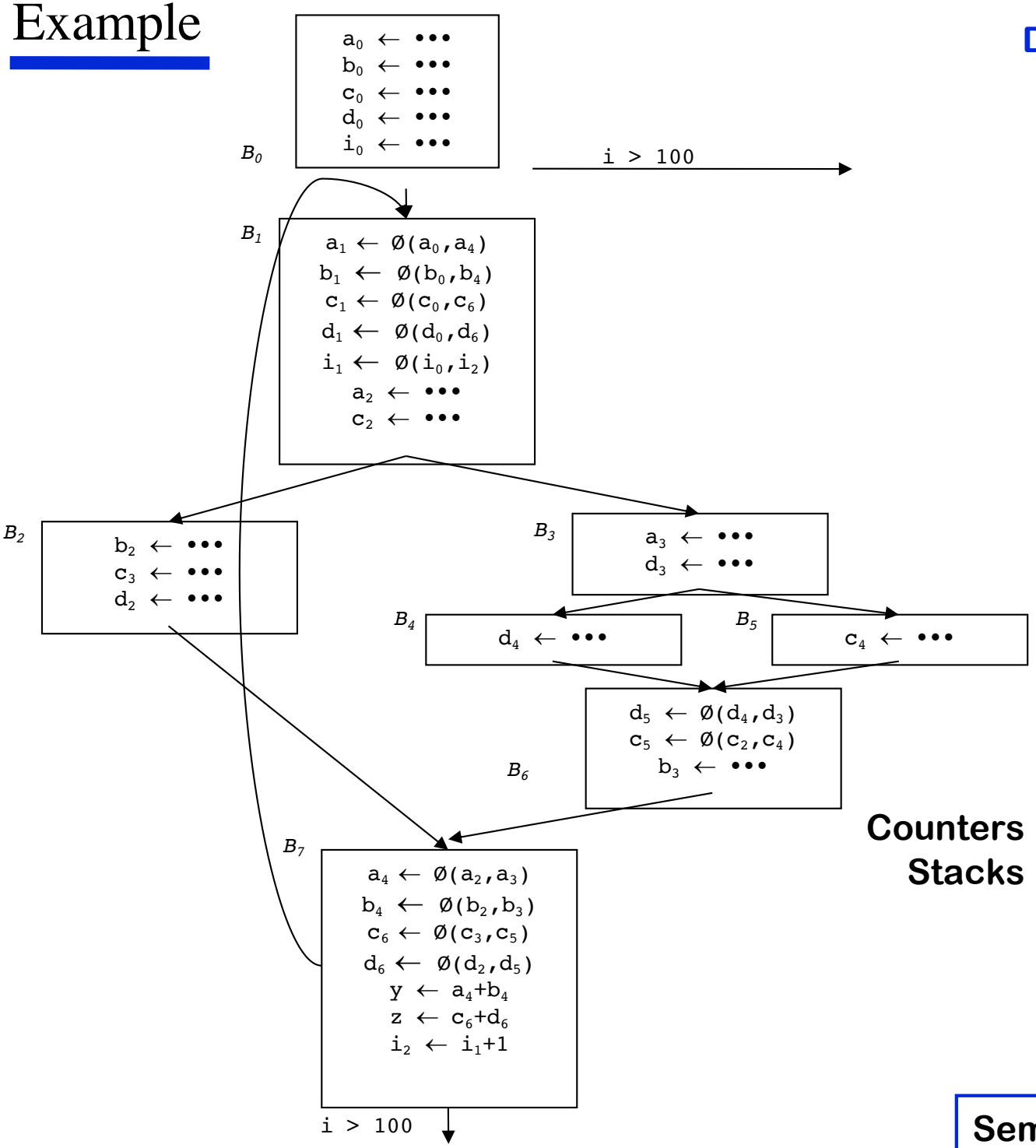


Before B_7

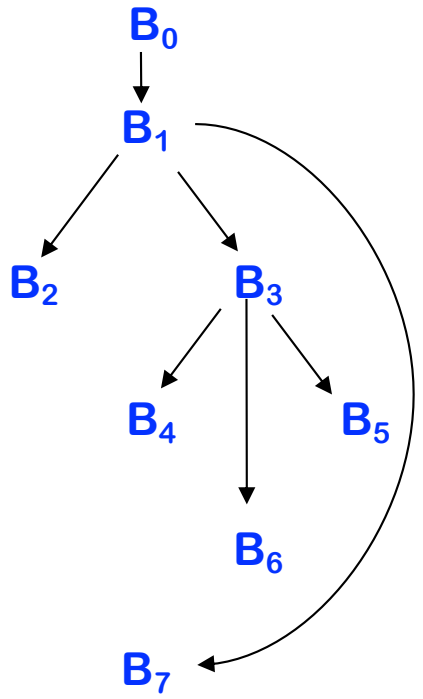
Counters / Stacks

a	b	c	d	i
4	4	6	6	2
a_0	b_0	c_0	d_0	i_0
a_1	b_1	c_1	d_1	i_1
a_2		c_2		

Example



Dominance Tree



End of B_7

a	b	c	d	i
5	5	7	7	3
a_0	b_0	c_0	d_0	i_0
a_1	b_1	c_1	d_1	i_1
a_2	b_4	c_2	d_6	i_2
a_4		c_6		

Counters
Stacks

Semi-pruned SSA, done!

Semi-pruned SSA V.S. Pruned SSA

- Semi-pruned SSA: discard names used in only one block
  - Significant reduction in total number of Ø-functions
  - Needs only local Live (appearance) information (cheap to compute)

- Pruned SSA: only insert Ø-functions where their value is live
  - Inserts even fewer Ø-functions, but costs more to do