Lecture Overview

Dominance

Immediate Dominators

Dominator Trees

Dominance Frontiers

SSA

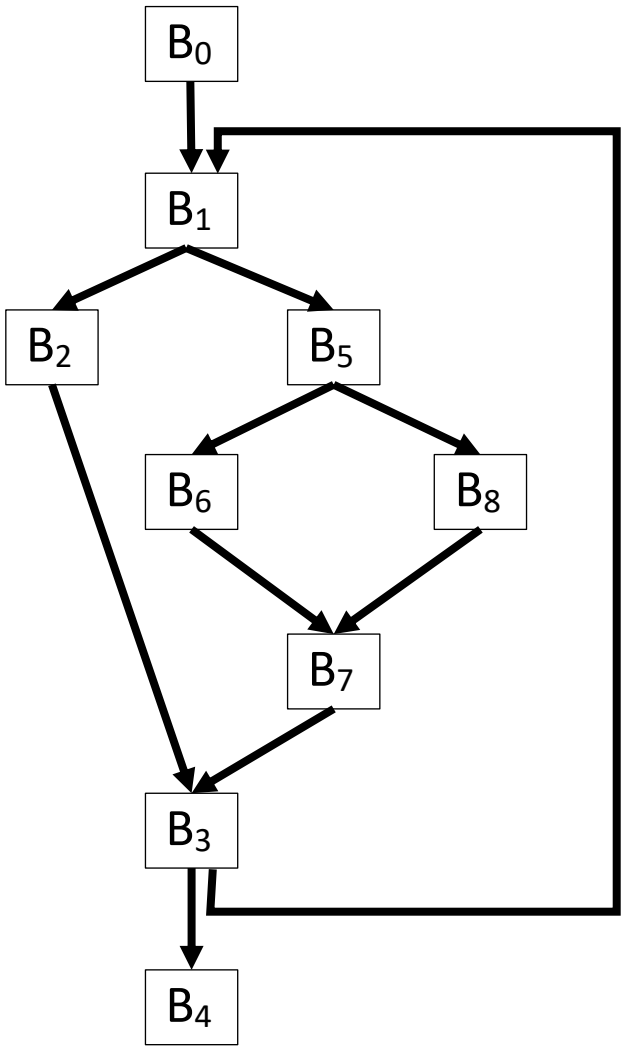
- **Maximal SSA**
- **Semipruned SSA**
  - **Placing ϕ-functions**
  - **Renaming**
- **Translation out of SSA**
- **Using SSA: Sparse Simple Constant Propagation**

[Chapter 9]

Dominance revisited

A key tool that compilers use to reason about the CFG is the notion of **dominators**.

A CFG node $\mathbf{B_i}$ dominates $\mathbf{B_j}$ iff $\mathbf{B_i}$ lies on every path from the entry node of the CFG to $\mathbf{B_j}$.



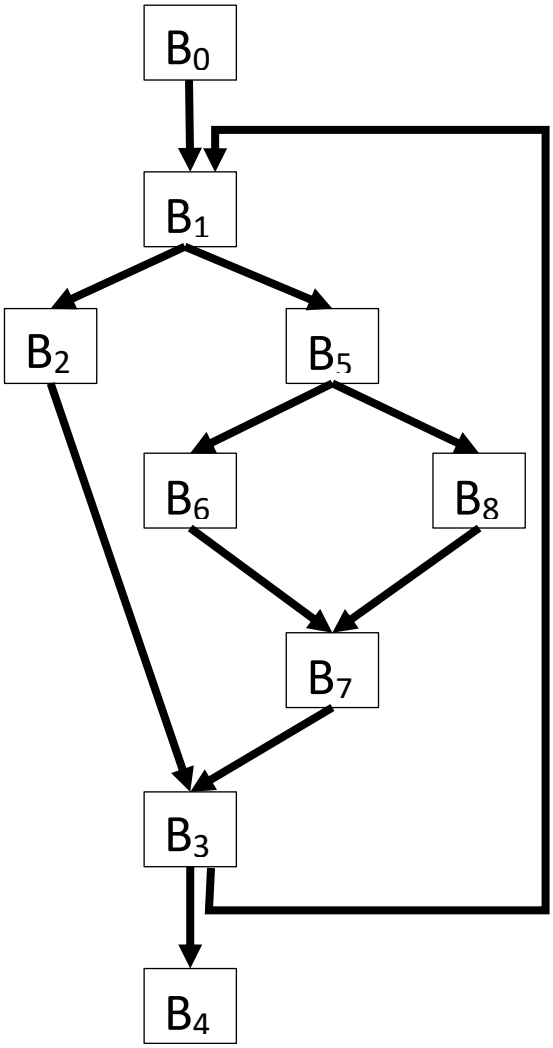
node	DOM(n)
B_0	{0}
B_1	{0,1}
B_2	{0,1,2}
B_3	{0,1,3}
B_4	{0,1,3,4}
B_5	{0,1,5}
B_6	{0,1,5,6}
B_7	{0,1,5,7}
B_8	{0,1,5,8}

Dominators play a key role in the construction of SSA form.

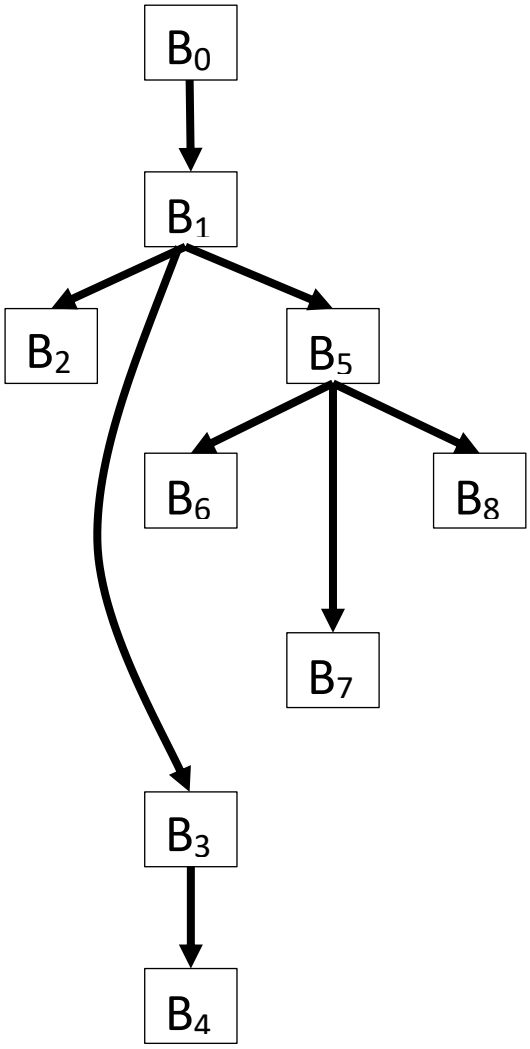
Dominator Trees

The nodes that strictly dominate *n* are **DOM**(*n*) – *n.* The closest strict dominator to *n* is called its **immediate dominator**, and denoted **IDOM**(*n*). The entry node has no immediate dominator.

The **dominator tree** compactly encodes **IDOM** and **DOM** information. It consists of edges (**IDOM**(*n*), *n*) for each node n.



Example CFG



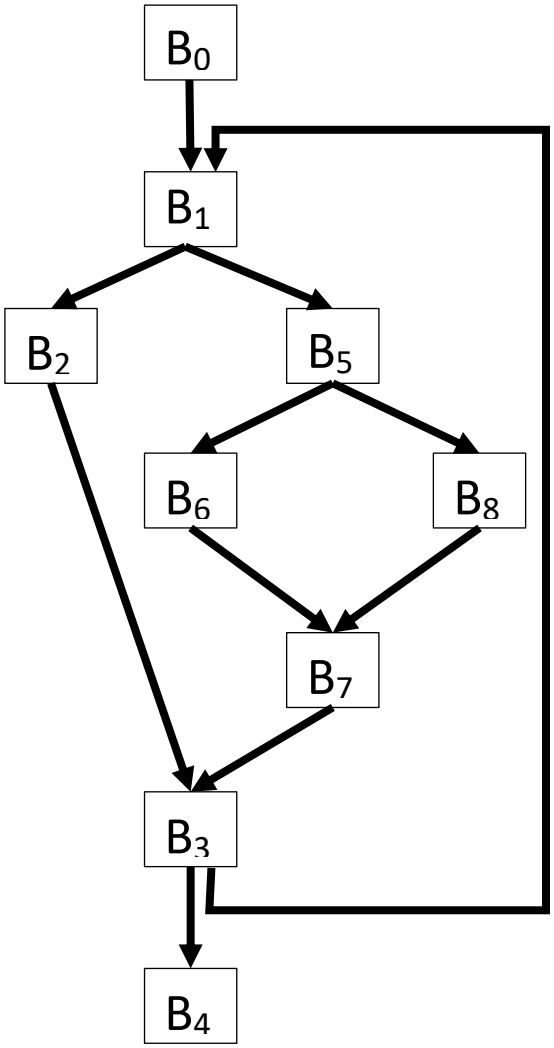
Dominator tree

Dominance Frontiers

The **dominance frontier DF**(*n*) of a node *n* is all nodes *m* where

1. *n* dominates a predecessor of *m*, and
2. *n* does not strictly dominate *m*.

These are the nodes "just outside" the dominance of *n*.



node	DF(n)
B_0	{}
B_1	{1}
B_2	{3}
B_3	{1}
B_4	{}
B_5	{3}
B_6	{7}
B_7	{3}
B_8	{7}

To compute dominance frontiers, we can do a dataflow analysis or use the following algorithm.

```
for all nodes i
    DF(i) = {}

for all nodes i
    if i has multiple predecessors then
        for each predecessor p of i
            runner = p
            while runner ≠ IDOM(i)
                DF(runner) = DF(runner) ∪ {i}
                runner = IDOM(runner)
```

SSA Revisited

It is desirable to use a single analysis to perform multiple transformations. Transforming code to good SSA encodes both data flow and control flow, and supports and simplifies many optimizations.

To transform code into SSA form, the compiler must insert the appropriate ϕ-functions for each variable into the code, and it must rename variables with subscripts to make these two properties hold:

1. Each definition in the procedure makes a new name.
2. Each use refers to a single definition.

Constructing Maximal SSA

1. *Inserting ϕ-functions.* At the start of each block with multiple predecessors, insert a ϕ-function, such as

   $$y = \phi(y, y)$$

   for every name *y* that the code either defines or uses in the current procedure. The ϕ-function should have one argument for each predecessor block in the CFG.

2. *Renaming*. After ϕ-function insertion, use a new subscript for each of the definitions of a variable, and compute Reaching Definitions. Because ϕ-functions are also definitions, they ensure that only one definition reaches any use. Now rename each use to have the subscript that reflects the definition that reaches it.

Maximal SSA potentially (and often) has unnecessary ϕ-functions, which waste space and computation time. We now look at a better SSA

construction, which makes **semipruned SSA,** which has fewer ϕ-functions.

Semipruned SSA: Placing ϕ-functions

The basic idea of this construction is simple: a definition of *x* in block **B** necessitates a ϕ-function for *x* at every node in the dominance frontier **DF(B)**. Since that ϕ-function is a new definition of *x*, it may, in turn, force the insertion of additional ϕ-functions.

One additional improvement that can be made is to only insert ϕ-functions for variables that are **live** (contain a value that can later be used) in more than one block. To apply this observation, one can compute the **global names,** which are simply those names that are live across multiple blocks, and ignore any non-global names when inserting ϕ-functions.

Global names are simply those names that have an upwards-exposed use in some block (i.e. are in the UEVar set). The algorithm to compute globals is:

```
Globals = {}
for each variable v
    Blocks(v) = {}

for each block b
    VarKill = {}
    for each operation x = y op z
        if y ∉ VarKill then
            Globals = Globals ∪ {y}
        if z ∉ VarKill then
            Globals = Globals ∪ {z}
        VarKill = VarKill ∪ {x}
        Blocks(x) = Blocks(x) ∪ {b}
```

While computing Globals, the algorithm also constructs, for each name, a list of all blocks that contain a definition of that name. These blocks

serve as a starting point for the ϕ-function insertion algorithm. That algorithm is shown below.

```
for each name x ∈ Globals
    WorkList = Blocks(x)
    for each block b in WorkList
        for each block d in DF(b)
            if d has no φ-function for x then
                insert a φ-function for x in d
                WorkList = WorkList ∪ {d}
```

B_0: i = 1
→ B_1

B_1: a = …
c = …
(a < c) → B_2, B_5

B_2: b = …
c = …
d = …
→ B_3

B_3: y = a + b
z = c + d
i = i + 1
(i ≤ 100) → B_1, B_4

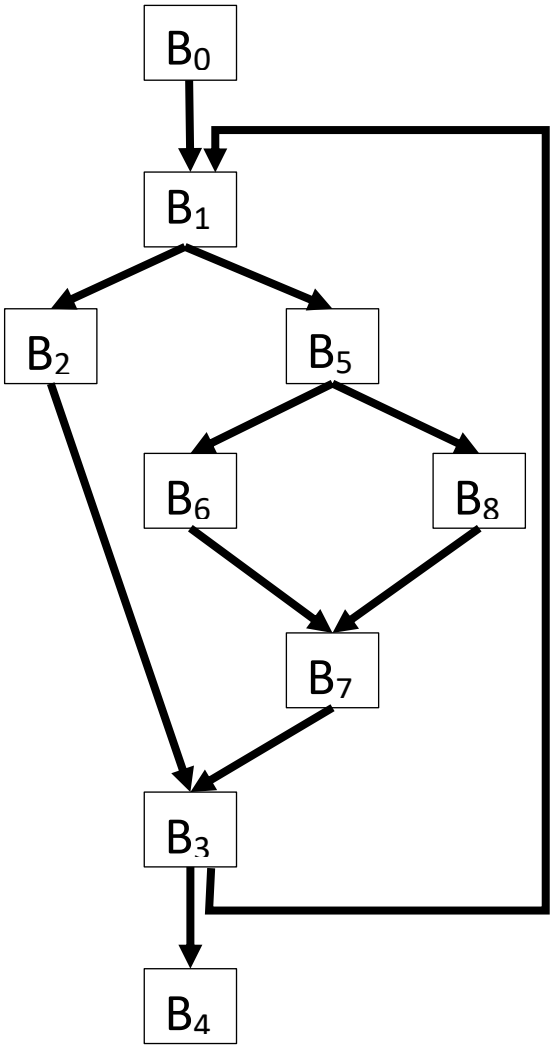
B_4: return

B_5: a = …
d = …
(a ≤ d) → B_6, B_8

B_6: d = …
→ B_7

B_7: b = …
→ B_3

B_8: c = …
→ B_7



	a	b	c	d	i	y	z
Blocks	1,5	2,7	1,2,8	2,5,6	0,3	3	3

B_0: i = 1
→ B_1

B_1: a = ϕ(a, a)
b = ϕ(b, b)
c = ϕ(c, c)
d = ϕ(d, d)
i = ϕ(i, i)
a = …
c = …
(a < c) → B_2, B_5

B_2: b = …
c = …
d = …
→ B_3

B_3: a = ϕ(a, a)
b = ϕ(b, b)
c = ϕ(c, c)
d = ϕ(d, d)
y = a + b
z = c + d
i = i + 1
(i ≤ 100) → B_1, B_4

B_4: return

B_5: a = …
d = …
(a ≤ d) → B_6, B_8

B_6: d = …
→ B_7

B_7: c = ϕ(c, c)
d = ϕ(d, d)
b = …
→ B_3

B_8: c = …
→ B_7

Semipruned SSA: Renaming

In the final SSA form, each global name becomes a base name, and individual definitions of that base name are distinguished by the addition of a numerical subscript.

The algorithm uses a preorder walk over the procedure's dominator tree. In each block, it first renames the values defined by the ϕ-functions at the head of the block, then it visits each operation in the block, in order. After the operations in the block have been rewritten, the algorithm rewrites the appropriate ϕ-function parameters in each CFG successor of the block. Then it recurses on dominator tree children. When it returns from those recursive calls, it restores the set of SSA names to the state that existed before the block was entered.

```
for each global name i
    counter[i] = 0
    stack[i] = {}
Rename(B0)

Rename(B)
    for each ϕ-function x = ϕ(…) in B,
        rewrite x as NewName(x)
    for each operation x = y op z   in B,
        rewrite y with subscript top(stack[y])
        rewrite z with subscript top(stack[z])
        rewrite x as NewName(x)
    for each successor of B in the CFG
        fill in ϕ-function parameters
    for each successor S of B in the dom. tree
        Rename(S)
    for each operation x = y op z in B, and each
        ϕ-function x = ϕ(…)
        pop(stack[x])

NewName(x)
    i = counter[x]
    counter[x]++
    push i onto stack[x]
    return "xi"
```

B_0: $i_0 = 1$
$\rightarrow B_1$

B_1: $a_1 = \phi(a_0, a_3)$
$b_1 = \phi(b_0, b_3)$
$c_1 = \phi(c_0, c_4)$
$d_1 = \phi(d_0, d_3)$
$i_1 = \phi(i_0, i_2)$
$a_2 = \ldots$
$c_2 = \ldots$
$(a_2 < c_2) \rightarrow B_2, B_5$

B_2: $b_2 = \ldots$
$c_3 = \ldots$
$d_2 = \ldots$
$\rightarrow B_3$

B_3: $a_3 = \phi(a_2, a_4)$
$b_3 = \phi(b_2, b_4)$
$c_4 = \phi(c_3, c_5)$
$d_3 = \phi(d_2, d_6)$
$y = a_3 + b_3$
$z = c_4 + d_3$
$i_2 = i_1 + 1$
$(i_2 \leq 100) \rightarrow B_1, B_4$

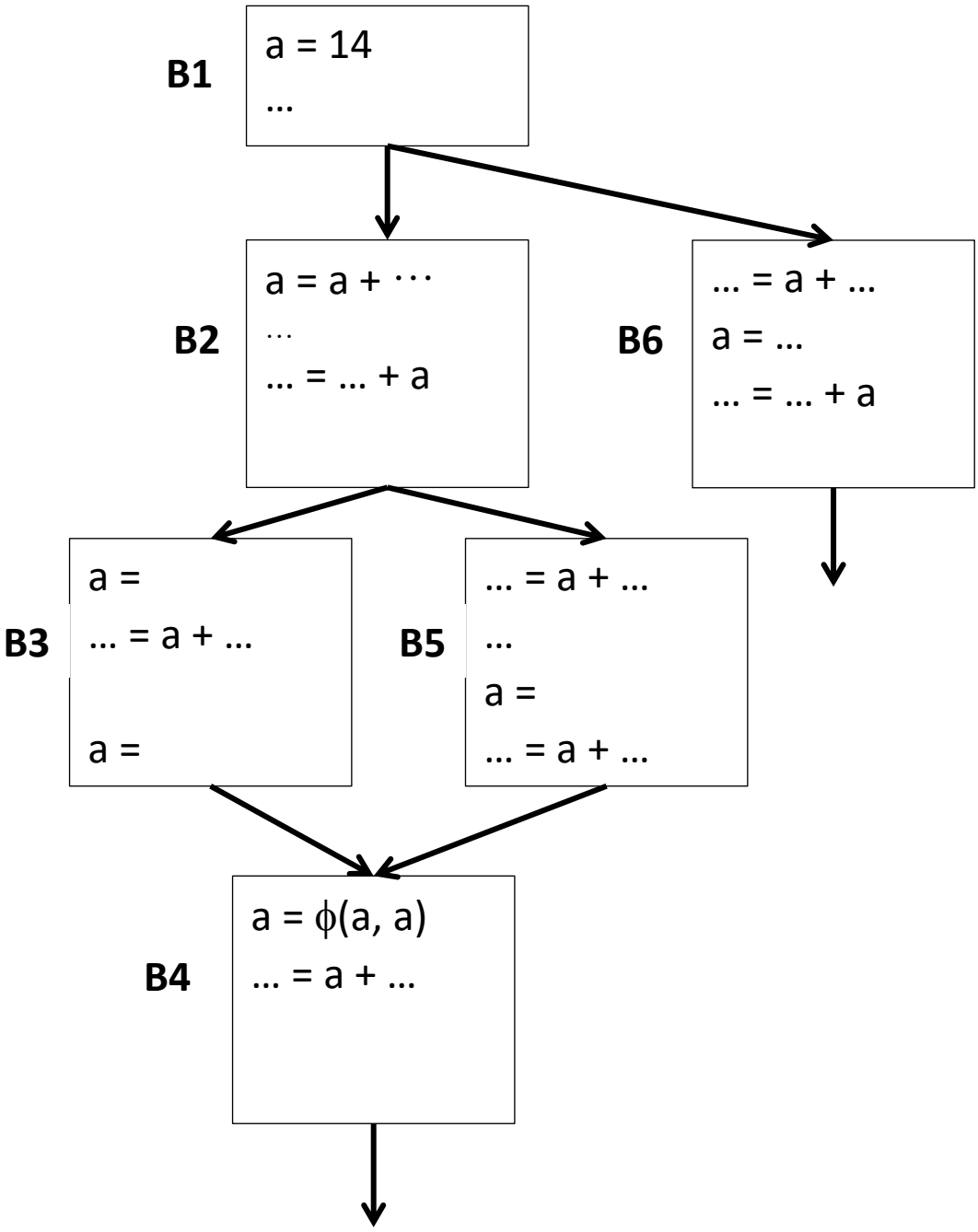
B_4: return

B_5: $a_4 = \ldots$
$d_4 = \ldots$
$(a_4 \leq d_4) \rightarrow B_6, B_8$

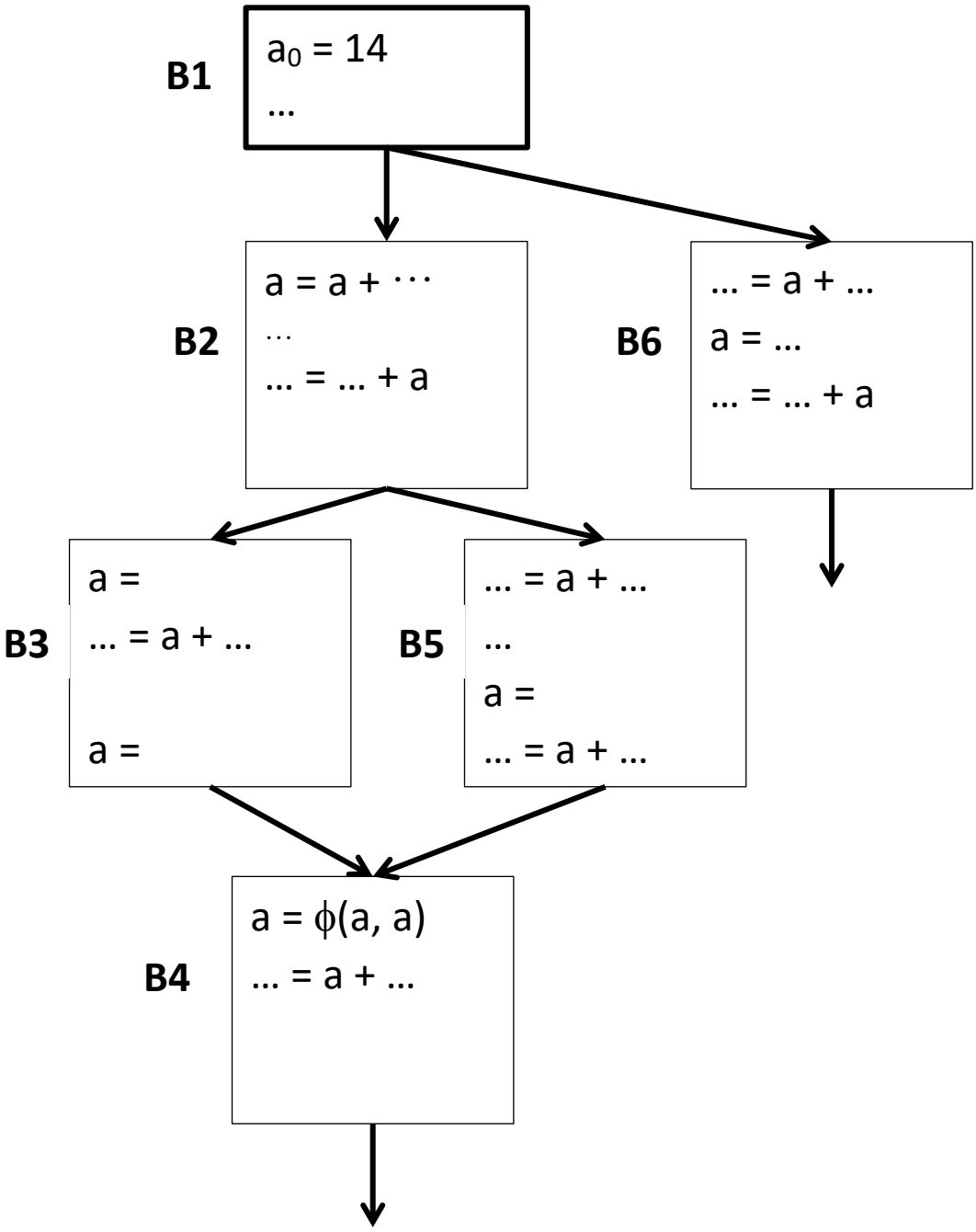
B_6: $d_5 = \ldots$
$\rightarrow B_7$

B_7: $c_5 = \phi(c_2, c_6)$
$d_6 = \phi(d_5, d_4)$
$b_4 = \ldots$
$\rightarrow B_3$

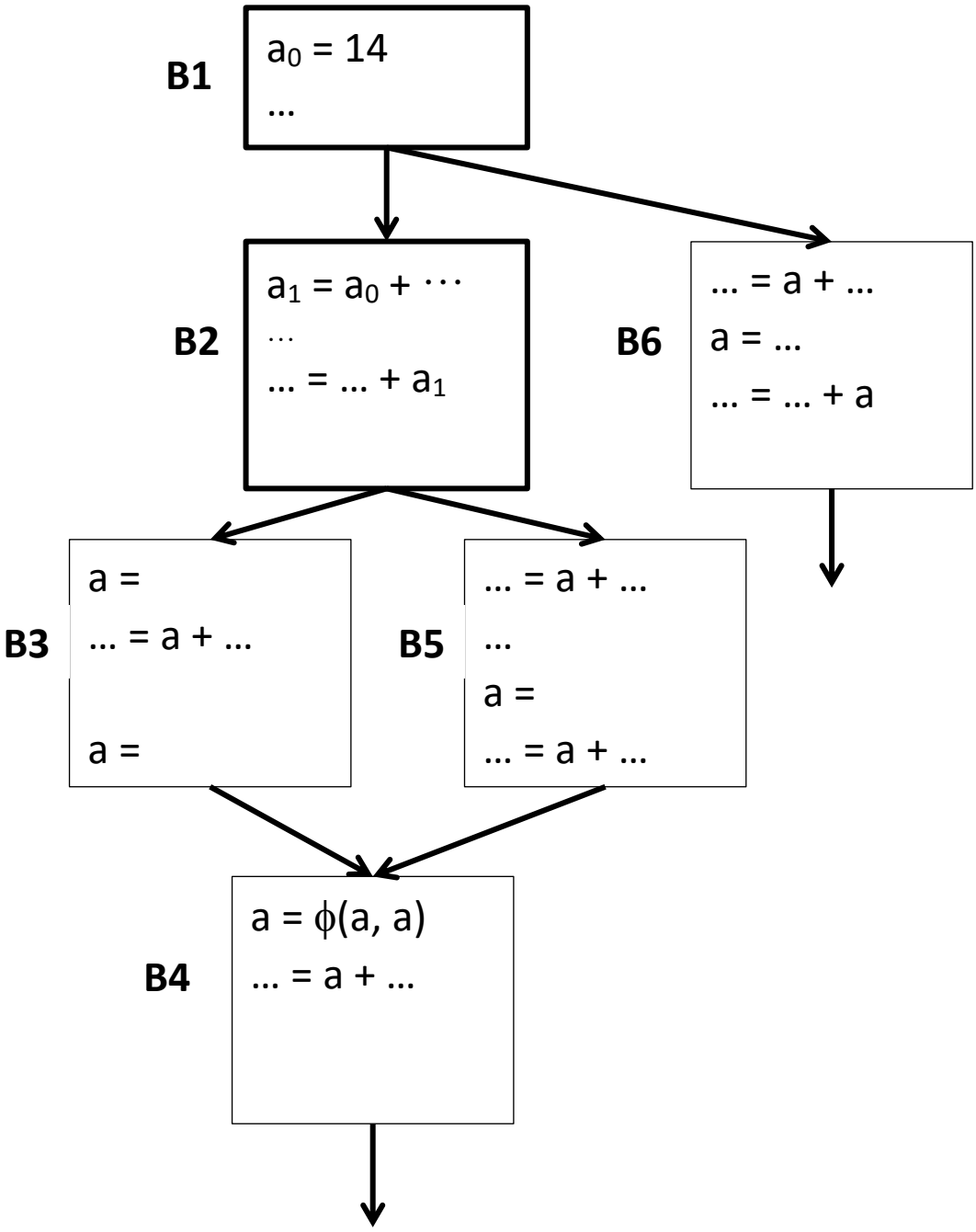
B_8: $c_6 = \ldots$
$\rightarrow B_7$



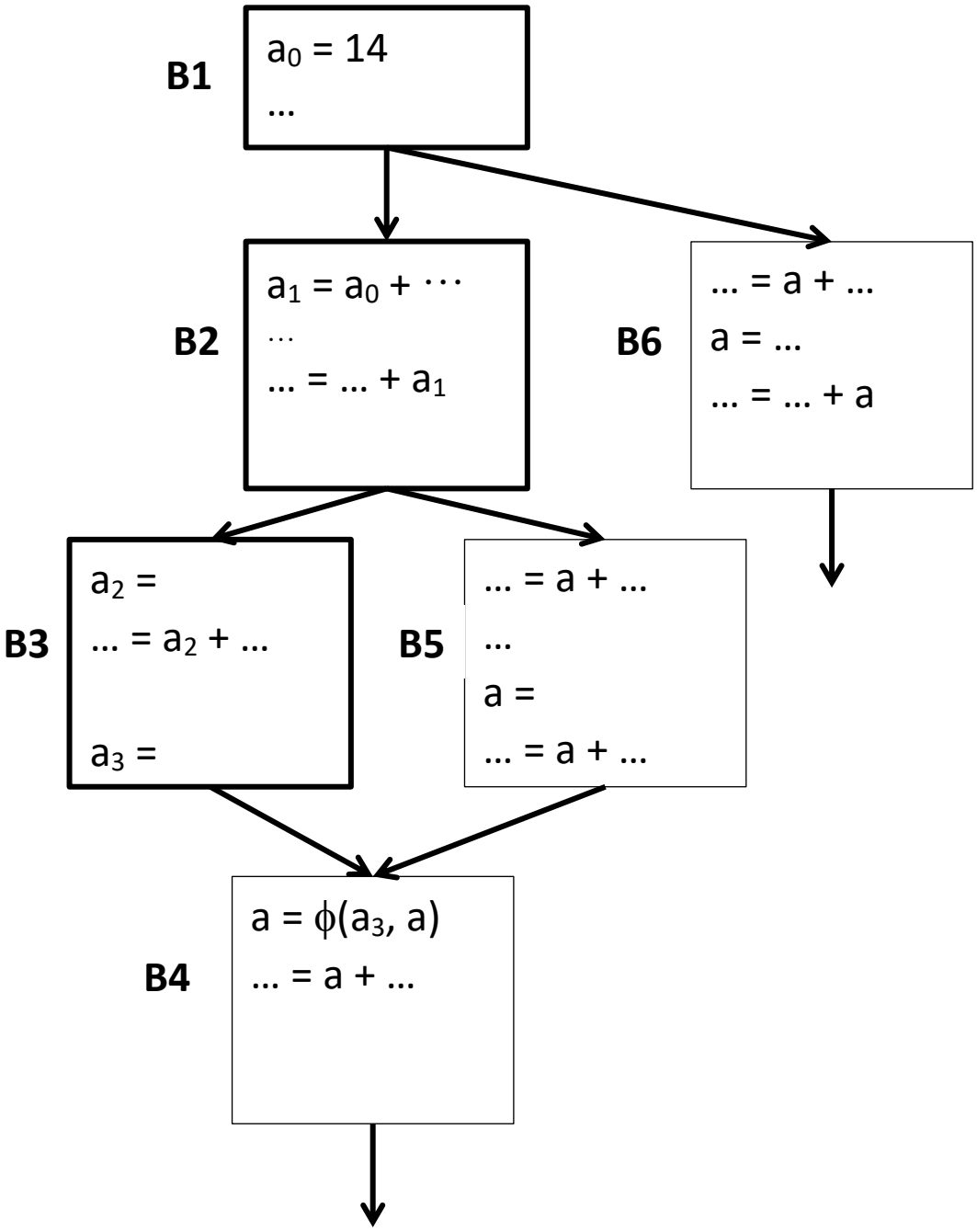
stack	counter
	0



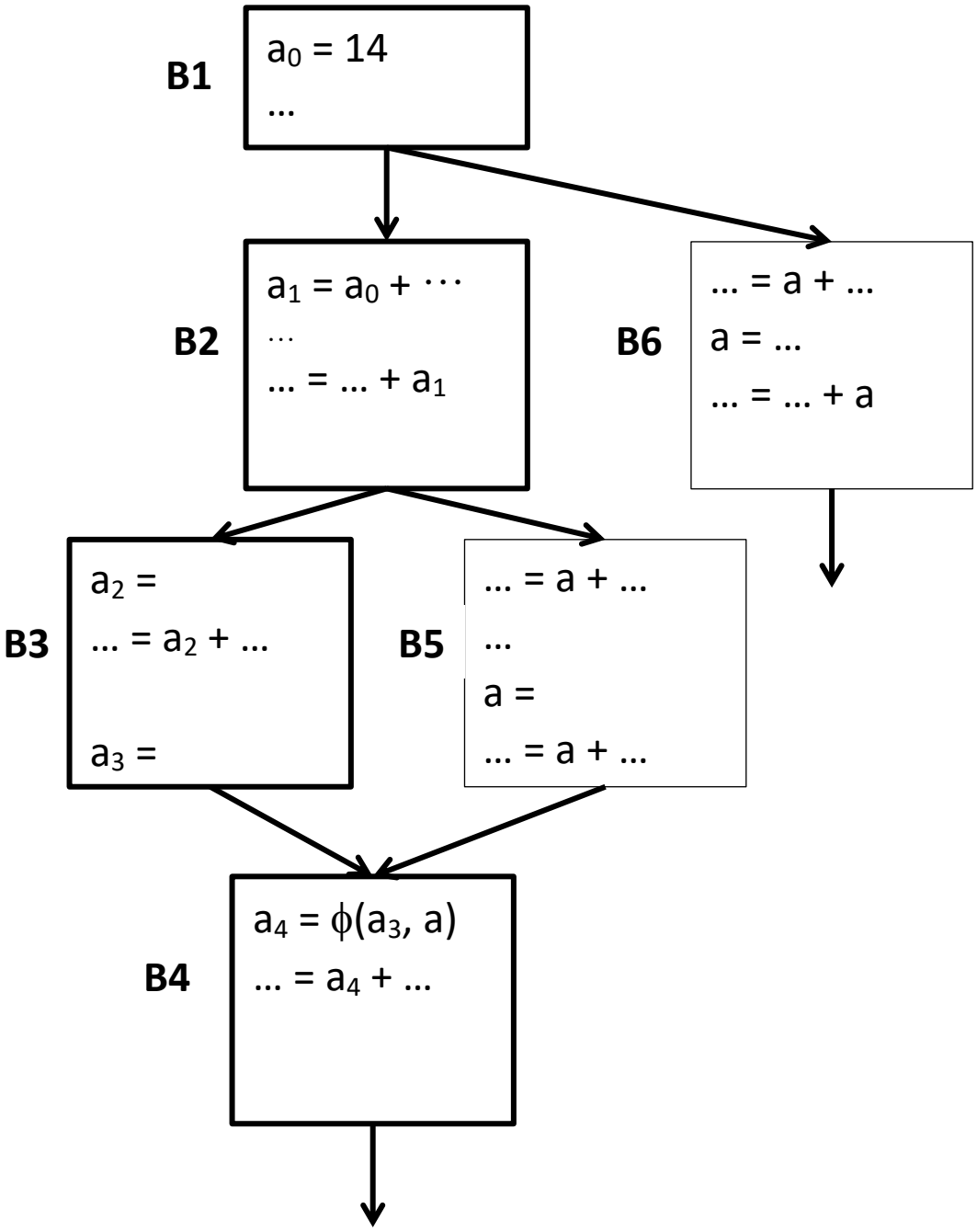
stack	counter
0	1



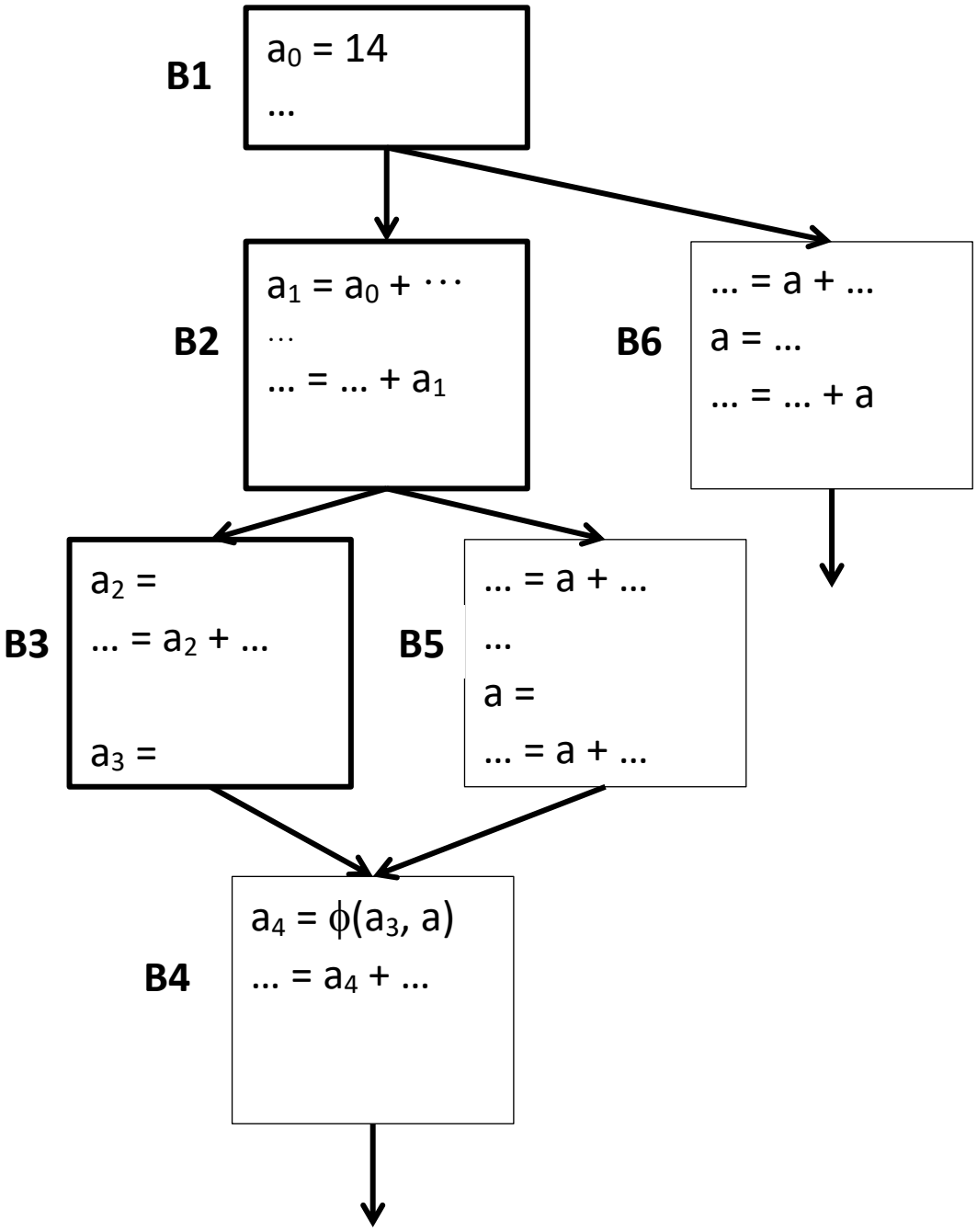
stack	counter
0	2
1	



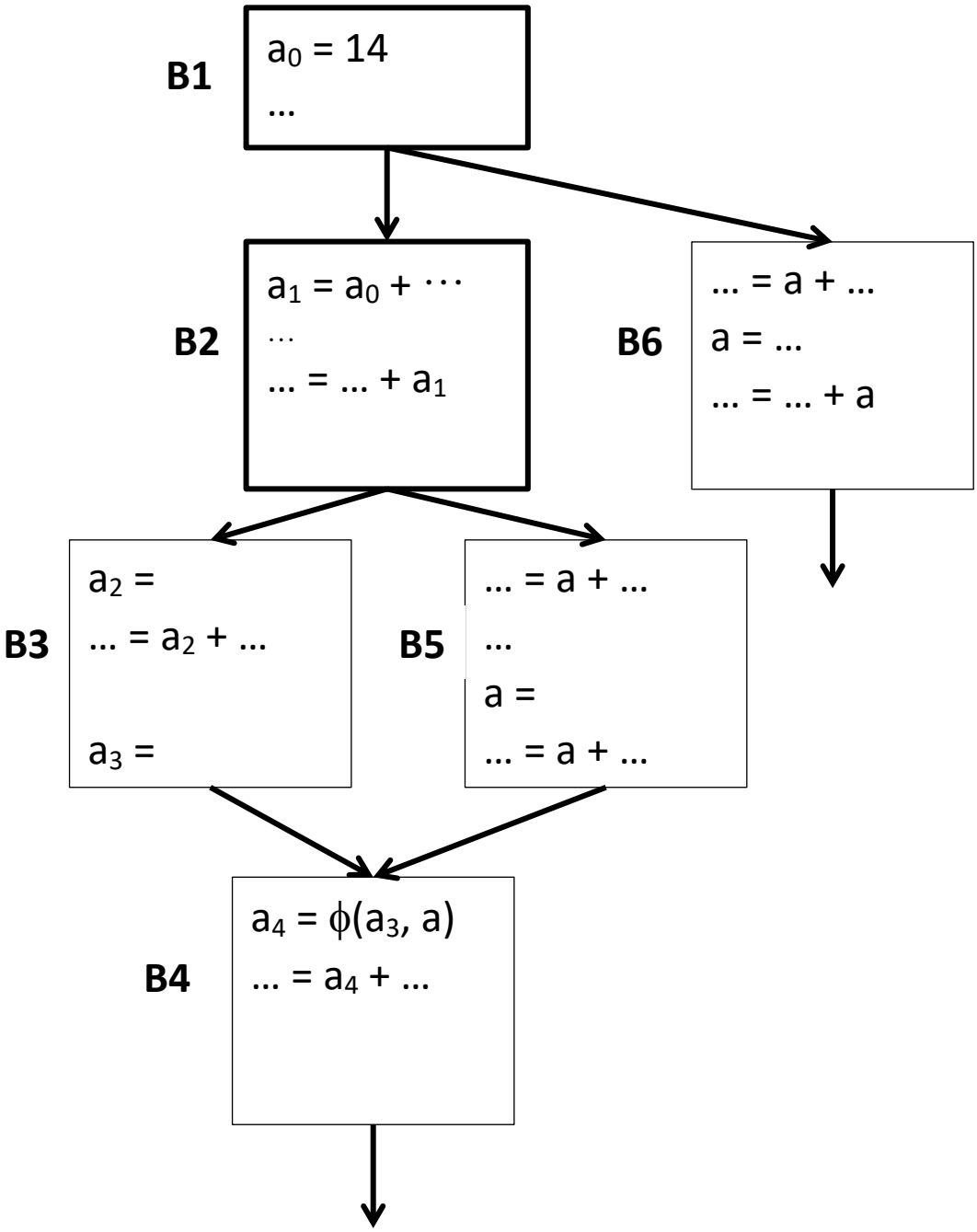
stack	counter
0	4
1	
2	
3	



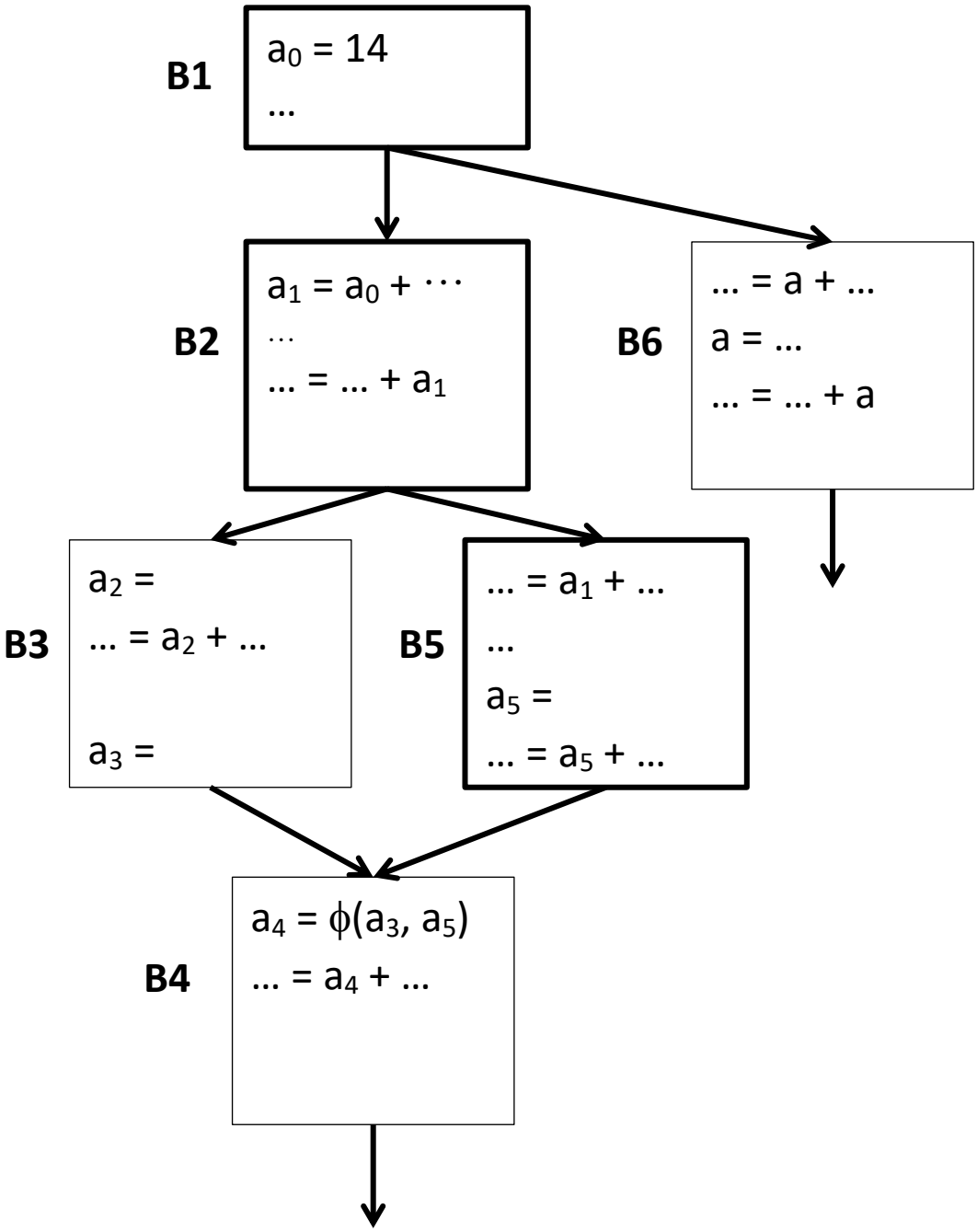
stack	counter
0	5
1	
2	
3	
4	



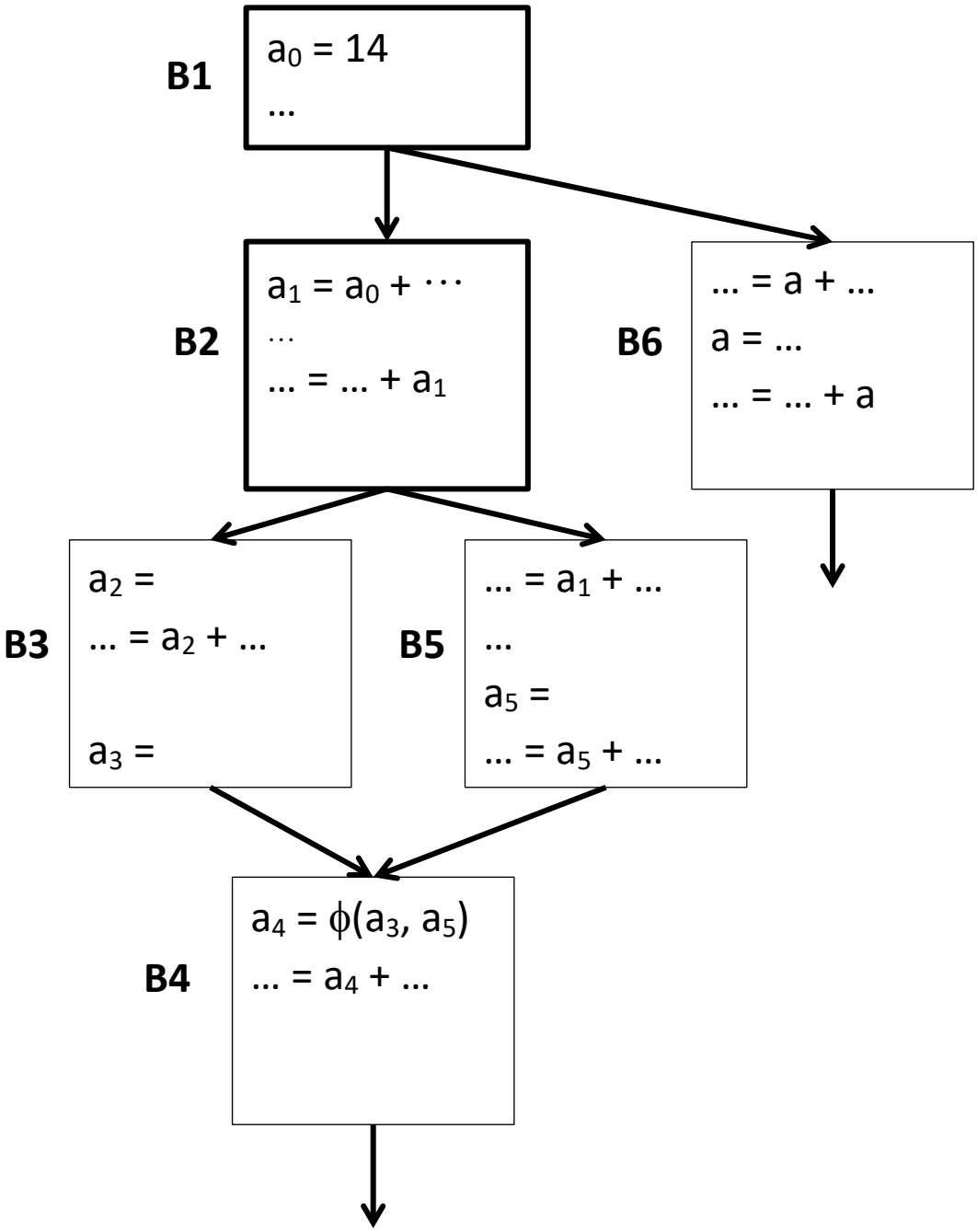
stack	counter
0	5
1	
2	
3	



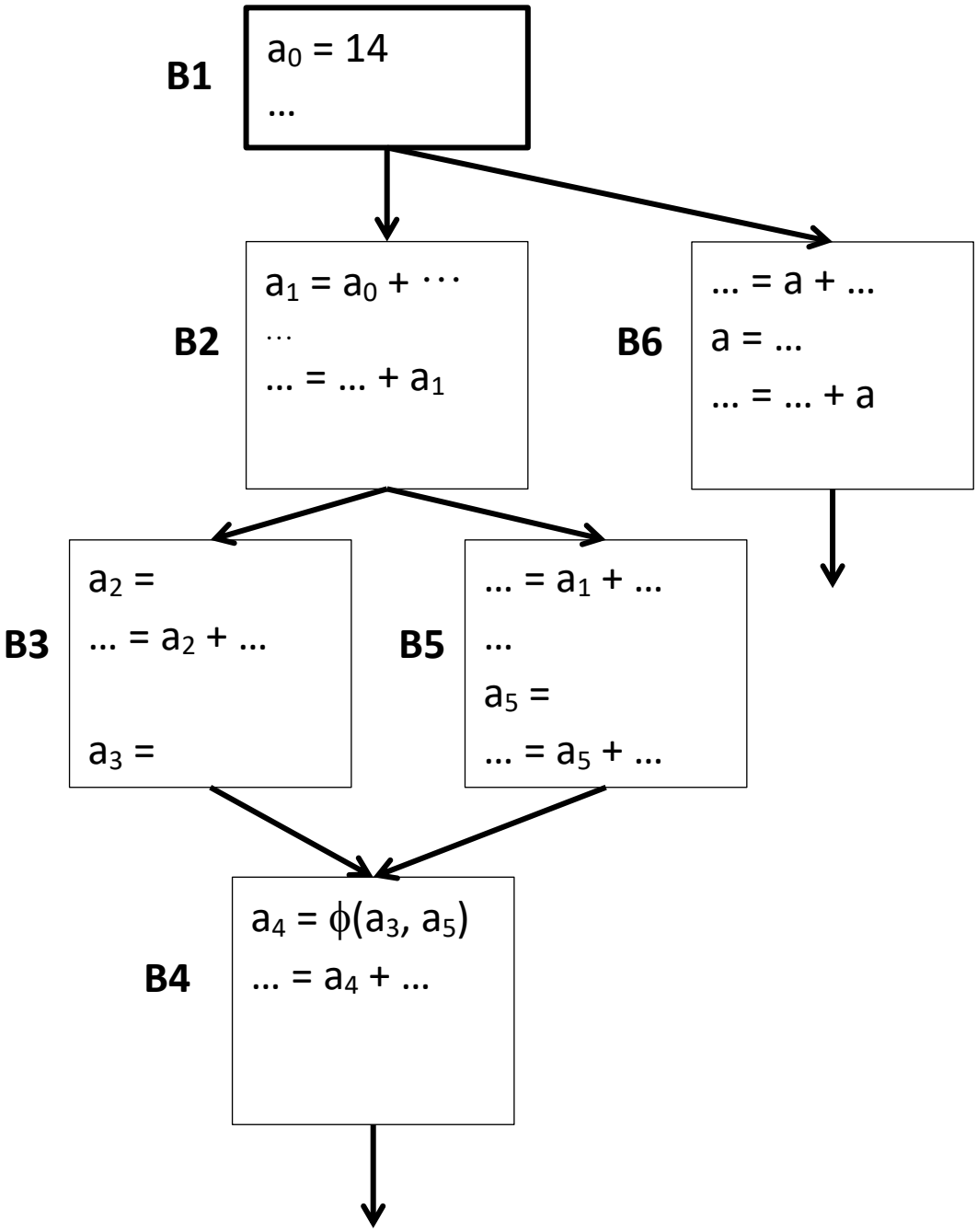
stack	counter
0	5
1	



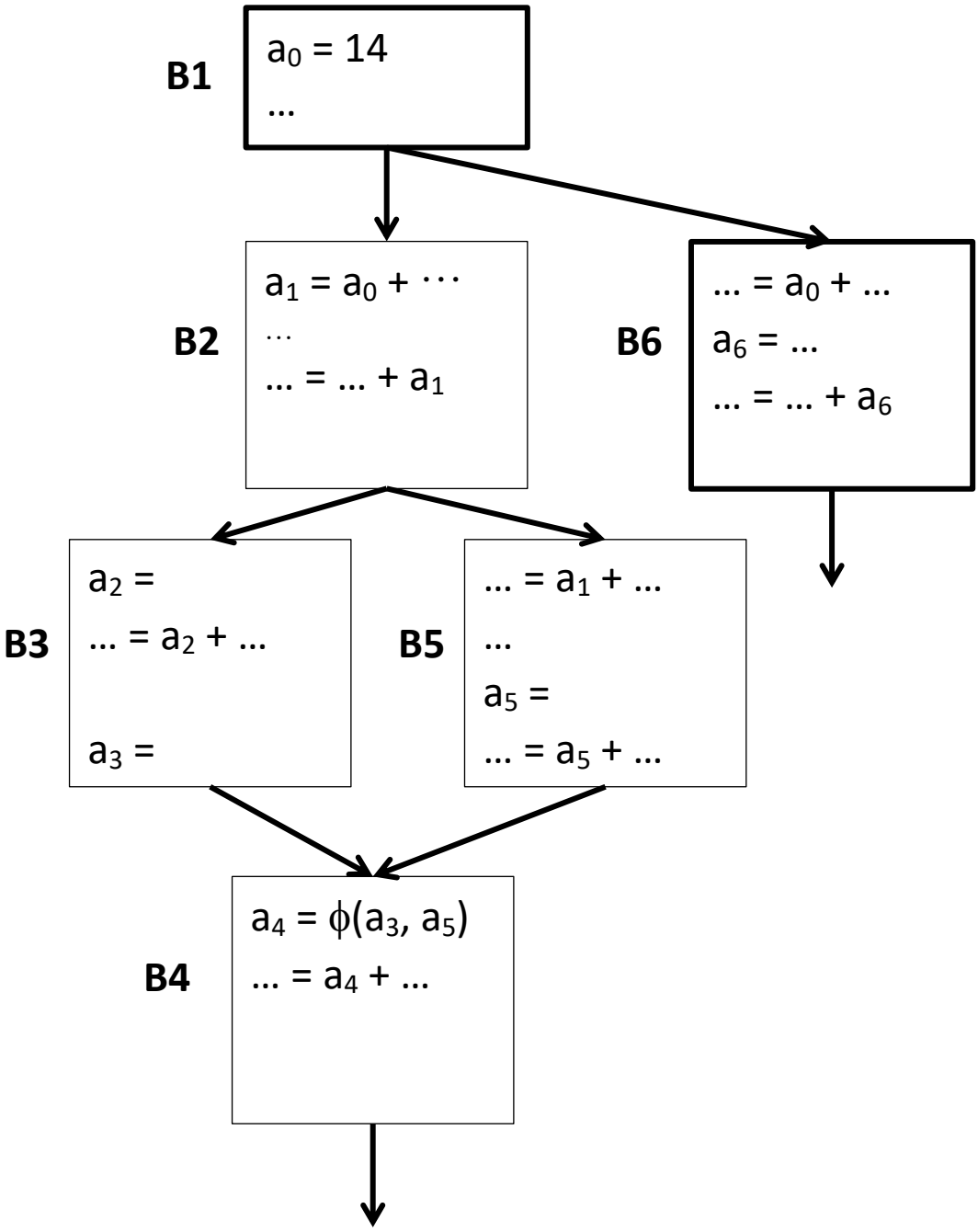
stack	counter
0	6
1	
5	



stack	counter
0	6
1	



stack	counter
0	6



stack	counter
0	7
6	

Translation out of SSA form

Because modern processors don't implement ϕ-functions, we need to translate SSA form back into executable code. It is tempting to believe that this can be done by removing the subscripts and the ϕ-functions, but this can produce incorrect code if the SSA form has been manipulated by optimizations.

To accomplish this, we can replace each ϕ-function with a set of copy operations—one along each incoming edge. (The edges are **split** to insert a new basic block along each of them.)

This may inflate the code with a lot of copy operations, but typically some form of copy-propagation or copy-folding optimization is run afterwards to bring the code down in size.

Using SSA Form: Sparse Simple Constant Propagation

In SSCP, the compiler annotates each SSA name with a value. The set of possible values forms a **semilattice**. A semilattice consists of a set **L** of values and a meet operator $\wedge$. The meet operator is idempotent, commutative, and associative. It imposes an order on the elements of **L** as follows:

$a \geq b$ if and only if $a \wedge b = b$, and
$a > b$ if and only if $a \geq b$ and $a \neq b$

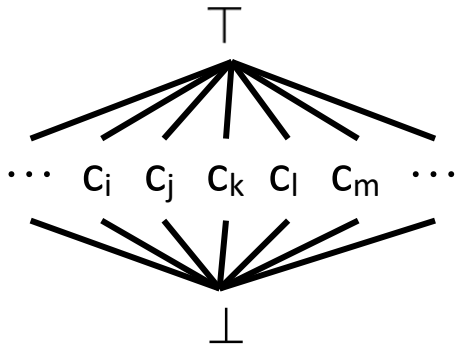
A semilattice has a bottom element, $\bot$, with the properties that

$\forall a \in L,\ a \wedge \bot = \bot$, and
$\forall a \in L,\ a \geq \bot$.

Some semilattices also have a top element, $\top$, with the properties that

$\forall a \in L,\ a \wedge \top = a$, and
$\forall a \in L,\ \top \geq a$.

In constant propagation, the structure of the semilattice used to model program values plays a critical role in the algorithm's runtime complexity. The semilattice for a single SSA name is:



For any two constants c_i and c_j, the meet $c_i \wedge c_j = \bot$. In SSCP, the algorithm initializes the value associated with each name to $\top$, which indicates that the algorithm has no knowledge of the SSA name's value. If the algorithm discovers that the name X has constant value c_i, it models that knowledge by assigning Value(X) the semilattice element c_i. If it discovers that X has a changing value, it models that fact with the value $\bot$.

Here's the initialization phase of SSCP:

```
WorkList = {}

for each SSA name n
    if n is defined by a ϕ-function
        Value(n) = ⊤
    else if n's definition gives a constant c_i
        Value(n) = c_i
    else if n's value cannot be known
        Value(n) = ⊥
    else if n's value is not known,
        Value(n) = ⊤

    if Value(n) != ⊤
        add n to WorkList
```

And here's the main (propagation) phase:

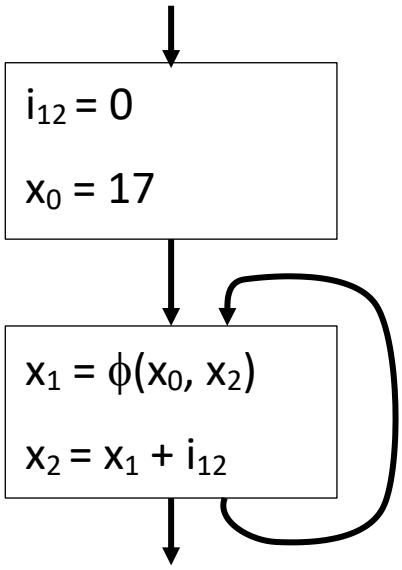
```
while (WorkList != {})
    remove some n from WorkList

    for each operation op that uses n
        let m be the SSA name defined by op
        if Value(m) != ⊥
            t = Value(m)
            Value(m) = result of interpreting op
                over lattice values
            if Value(m) != t
                add m to WorkList
```

To interpret an op over lattice values, arguments $\top$ and c_i give $\top$, arguments c_i and c_j give c_k, and either argument being $\bot$ gives $\bot$. This holds unless (1) the operation is a ϕ-function, in which case the result is the meet of all of the operands, or (2) there is a special case based on the operator, such as $\bot * 0 = 0$.

Optimism

SSCP is regarded as an **optimistic** algorithm. This is because it uses $\top$ rather than $\bot$ to initialize variables whose state is unknown. This initialization allows constant values to propagate into and around loops. Using $\bot$ instead leads to a **pessimistic** algorithm, which in this case doesn't identify as many constants.



Pessimistic time	x_0	x_1	x_2
0	17	$\bot$	$\bot$
1	17	$\bot$	$\bot$

Optimistic time	x_0	x_1	x_2
0	17	$\top$	$\top$
1	17	17	17

SSA plays a crucial role in SSCP. Without SSA, one would have to use a traditional data-flow algorithm, with something like

ConstantsIn(n) =

$$\left(\bigcap_{m \in \mathrm{preds}(n)} \boldsymbol{ConstantsOut}\,(m) \right)$$

where ConstantsOut(m) is some function of the instructions in block m, and of ConstantsIn(m). This leads to a much more complicated and costly algorithm than SSCP.

The propagation step of SSCP is sparse, and is best implemented by having def-use chains, so one can step from the definition of a variable directly to its uses. It is a simple matter to make def-use chains from SSA form.

So SSA leads to an efficient, understandable sparse algorithm for global constant propagation.