

Static Single Assignment

15-411 Compiler Design

Peter Lee, Frank Pfenning,
André Platzer

Redundancy elimination

Redundancy elimination optimizations attempt to remove redundant computations

Common redundancy elimination optimizations are

- value numbering
- conditional constant propagation
- common-subexpression elimination (CSE)
- partial-redundancy elimination

What they do

```
read(i);  
j = i + 1;  
k = i;  
l = k + 1;
```

*value
numbering
determines
that $j == l$*

```
i = 2;  
j = i * 2;  
k = i + 2;
```

*constant
propagation
determines
that $j == k$*

```
read(i);  
l = 2 * i;  
if (i > 0) goto L1;  
j = 2 * i;  
goto L2;  
L1: k = 2 * i;  
L2:
```

*CSE
determines
that 3rd “ $2*i$ ”
is redundant*

Value numbering

Basic idea:

- associate a symbolic value to each computation, in a way that any two computations with the same symbolic value always compute the same value
- Then we never need to recompute (locally within a basic block at least)

Congruence of expressions

Define a notion of *congruence* of expressions to see if they compute the same

- $x \oplus y$ is congruent to $a \otimes b$ if \oplus and \otimes are the same operator, and x is congruent to a and y is congruent to b
- we may also take commutativity into account

In SSA form variables x and a are congruent only if they are both live, and they are the same variable, or if they are provably the same value (by constant or copy propagation)

Local value numbering

Suppose we have

- $t1 = t2 + 1$

Look up the key “ $t2+1$ ” in a hash table

- Use a hash function that assigns the same hash value (i.e., the same *value number*) to expressions $e1$ and $e2$ if they are congruent

If key “ $t2+1$ ” is not in table, then put in “ $t2+1 \rightarrow t1$ ”

- next time we hit key “ $t2+1$ ”, replace it with value “ $t1$ ”

Example

```
read(i);  
j = i + 1;  
k = i;  
l = k + 1;
```

$i=v1$

$j=v2$

$k=v1$

$\text{Hash}(v1+1) \rightarrow j$

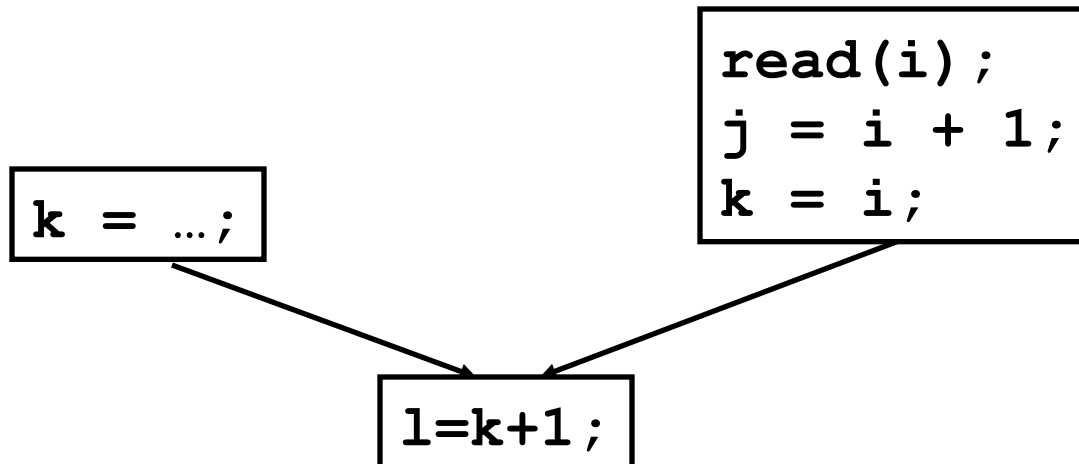
$\text{Hash}(v1+1) \rightarrow j$

Therefore $l=j$

Global value numbering?

Local value numbering (within a basic block) is easy

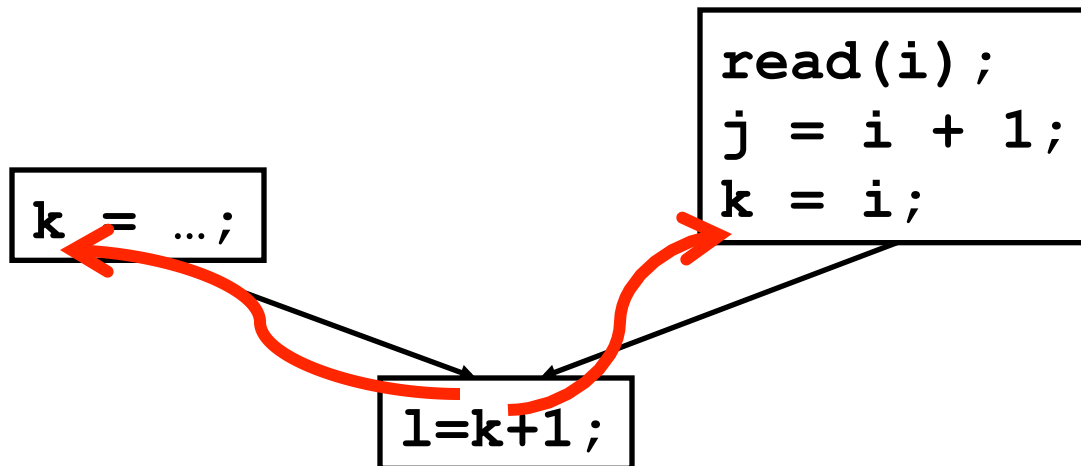
Global value numbering (within a procedure)?



Importance of use-def

In the global case, we must watch out for multiple assignments

Could do dataflow analysis for global value numbering

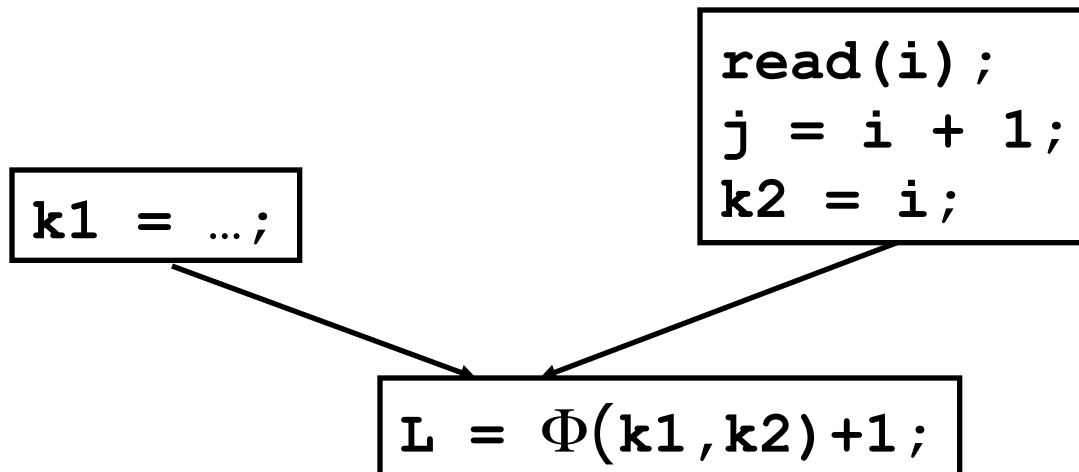


use-def analysis

Embed use-def into IR

Use-def information is central to many optimizations

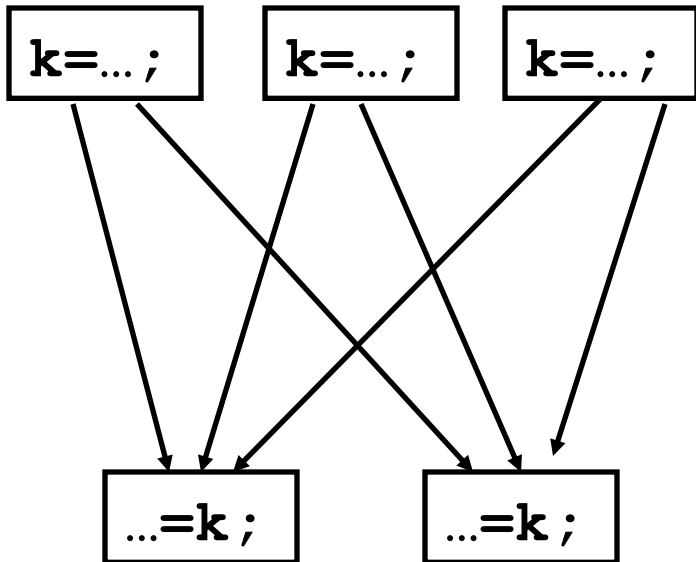
The point of static single assignment (SSA) is to represent
Use-def information explicitly



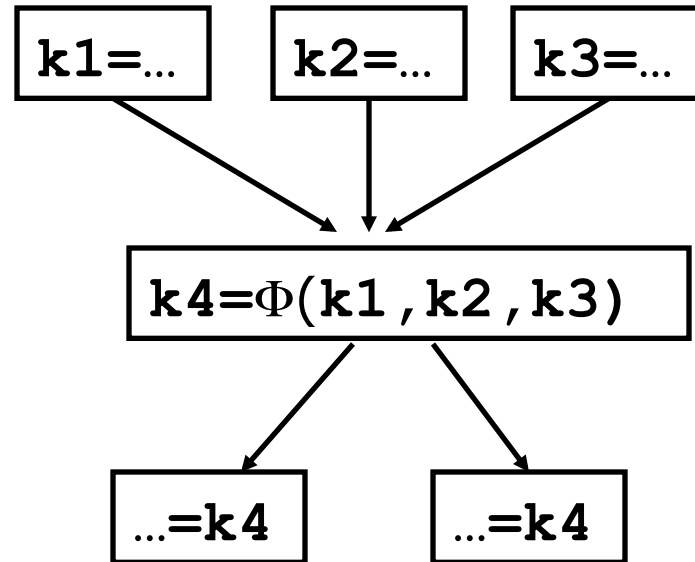
SSA use-def

SSA reduces the representational complexity of use-defs

$|Defs| * |Uses|$



$|Defs| + |Uses|$



Local vs global

Many optimizations can be performed both locally and globally

- local: within a basic block
- global: across basic blocks

Typically, only the global version needs dataflow analysis. But it often needs a lot of def-use analysis.

Global value numbering

Goal: global value numbering across basic blocks.

This is where converting IR to *static single assignment* (SSA) form helps.

SSA Form

SSA form

Static single-assignment (SSA) form arranges for every value computed by a program to have a unique definition

SSA is a way of structuring the intermediate representation so that every variable is (statically) assigned exactly once (hence it is a dynamic constant)

Equivalent to continuation-passing style IR

Developed at IBM: Cytron, Ferrante, Rosen, Wegman, Zadeck

Why use SSA?

SSA makes use-def chains explicit in the IR, which helps to simplify some analyses and optimizations

Several analyses/optimizations trivial on SSA: redundancy eliminations like

- Value numbering
- Conditional constant propagation
- Common subexpression elimination (SSA → local trafo)
- Some parts of partial redundancy elimination

SSA invariants

- Every def has a unique name
- Every use refers to a unique def
- Thus explicit def-use links

Create Maximal SSA (simple)

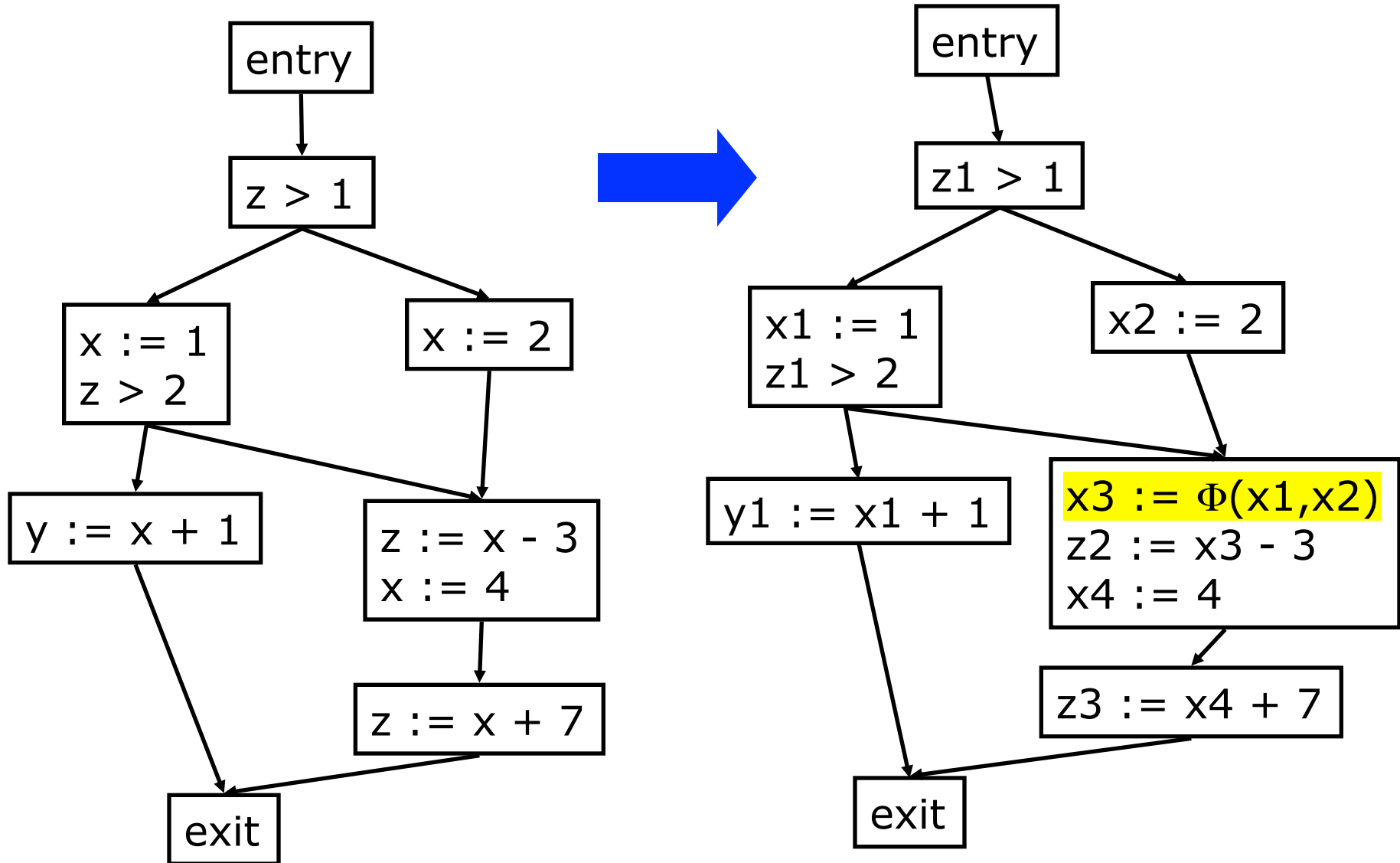
Φ pseudofunctions select definition from actual control flow

Simple SSA translation (with parallel Φ):

- Insert Φ functions at each block with merge-point
 - $\Phi(t, t, \dots, t)$, where the number of t 's is the number of incoming control flow edges
- Compute reach-def, which is unique. Rename def and uses of variables to establish SSA property

After optimizations: discard Φ functions and replace by parallel $x_3 := x_2$ on new blocks for back-edges. Then copy propagate & reg alloc

Example



Minimal SSA form

For inserting minimal Φ functions, we need to know the limit of where our definitions surely reach. An SSA form with the minimum number of Φ functions can be created by using *dominance frontiers*

- In a flowgraph, node *a* *dominates* node *b* (“ $a \geq b$ ”) if every possible execution path from node *entry* to *b* includes *a*
- Uses of var *x* dominated by *a* will use that def at *a*
- If *a* and *b* are also different nodes, we say that *a* *strictly dominates* *b* (“ $a > b$ ”)
- If $a > b \wedge \neg \exists c (a > c \wedge c > b)$, then *a* is *the immediate dominator* of *b* (“ $a = \text{idom}(b)$ ”)
→ defines *dominator tree*

Computing dominators (naïve)

$D[a]$ = a's dominators = set of nodes that dominate a

$$D[\text{entry}] = \{\text{entry}\}$$

$$D[a] = \{a\} \cup \bigcap_{b \in \text{Pred}(a)} D[b]$$

if all pred of a dominate c then $a \geq c$.

SSA \rightarrow $\text{def}(v) \geq \text{live}(v)$

Dominance frontier

For a node a , the dominance frontier of a , $DF[a]$, the first nodes that a just does not dominate anymore

Place Φ functions at $DF[a]$, because a 's defs have a competitor at $DF[a]$

- $DF[a] = \{b \mid \neg(a > b) \text{ but } \exists c \in \text{Pred}(b) \text{ with } a \geq c\}$

Computing DF[a]

A naïve approach to computing DF[a] for all nodes a would require quadratic time

- $DF[a] = \{b \mid \neg(a > b) \text{ but } \exists c \in \text{Pred}(b) \text{ with } a > c\}$

However, an approach that usually is linear time involves cutting into parts:

- $DF_{local}[a] = \{b \in \text{Succ}(a) \mid idom(b) \neq a\}$
- $DF_{up}[a,c] = \{b \in DF[c] \mid idom(c) = a \wedge idom(b) \neq a\}$

Then:

- $DF[a] = DF_{local}[a] \cup \bigcup_{c \in G \wedge idom(c) = a} DF_{up}[a,c]$

DF computation, cont' d

What we want, in the end, is the set of nodes that need Φ functions, for each variable, e.g., from the set S of nodes defining variable k

So we define $DF[S]$, for a set of flowgraph nodes S :

- $DF[S] = \bigcup_{a \in S} DF[a]$

Iterated DF

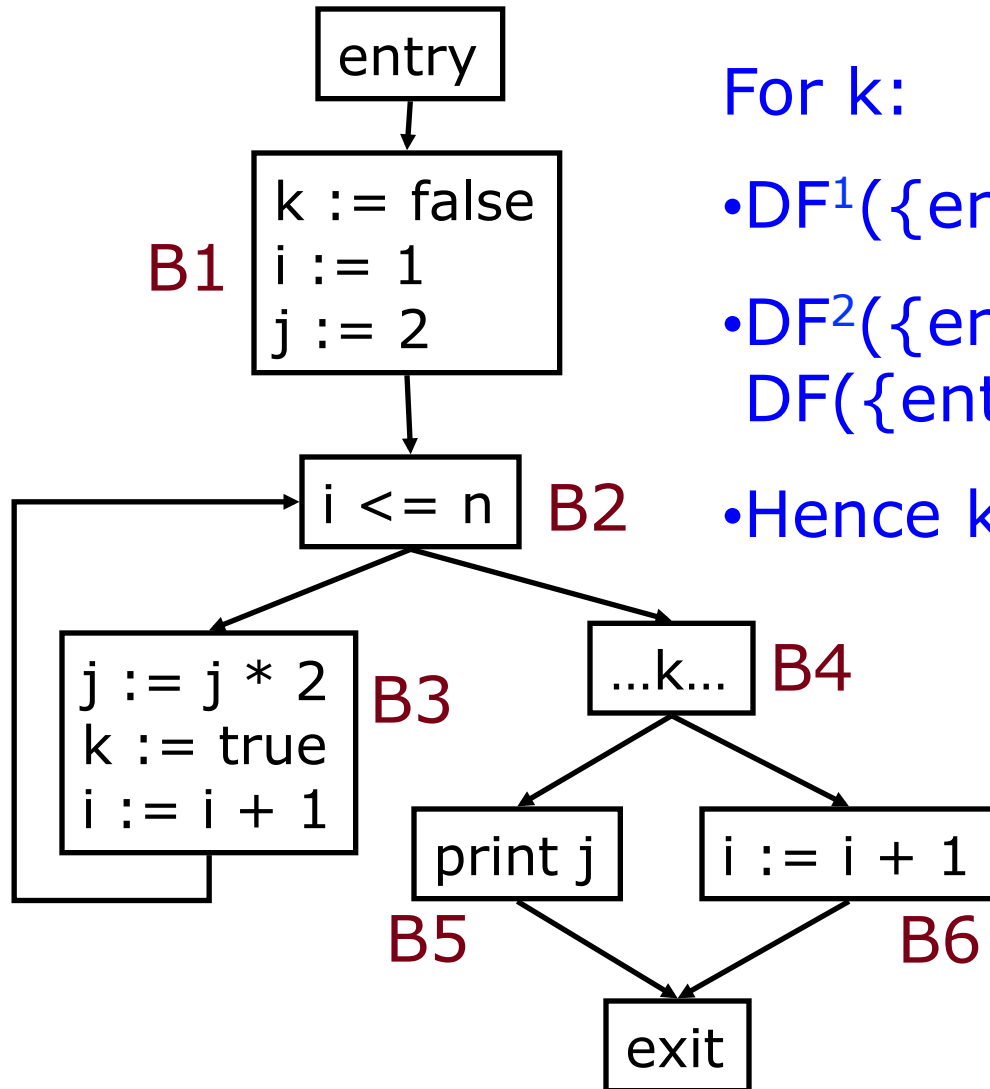
Φ are new defs, cause more Φ , thus iterate.

Iterated dominance frontier is defined as follows:

- $DF^+[S] = \lim_{i \rightarrow \infty} DF^i[S]$
- where
 - $DF^0[S] = S$
 - $DF^{i+1}[S] = DF[S \cup DF^i[S]]$

Theorem: Let $def(x)$ set of nodes assigning to x , and $live(x)$ nodes at which x is live. Then correct SSA has to place Φ at $DF^+[def(x)] \cap live(x)$

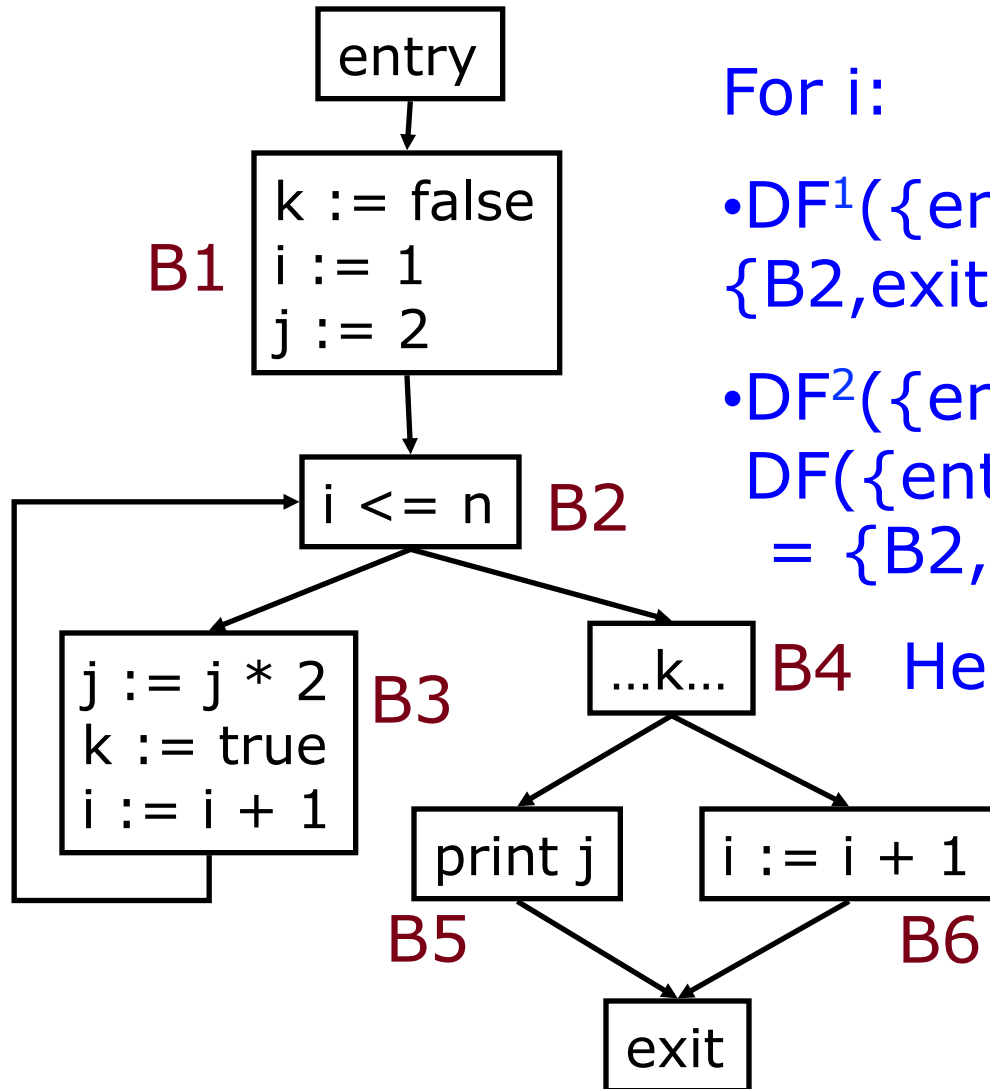
Example



For k :

- $DF^1(\{\text{entry}, B1, B3\}) = \{B2\}$
- $DF^2(\{\text{entry}, B1, B3\}) = DF(\{\text{entry}, B1, B2, B3\}) = \{B2\}$
- Hence $k2 := \Phi(k1, k3)$ at B2

Example



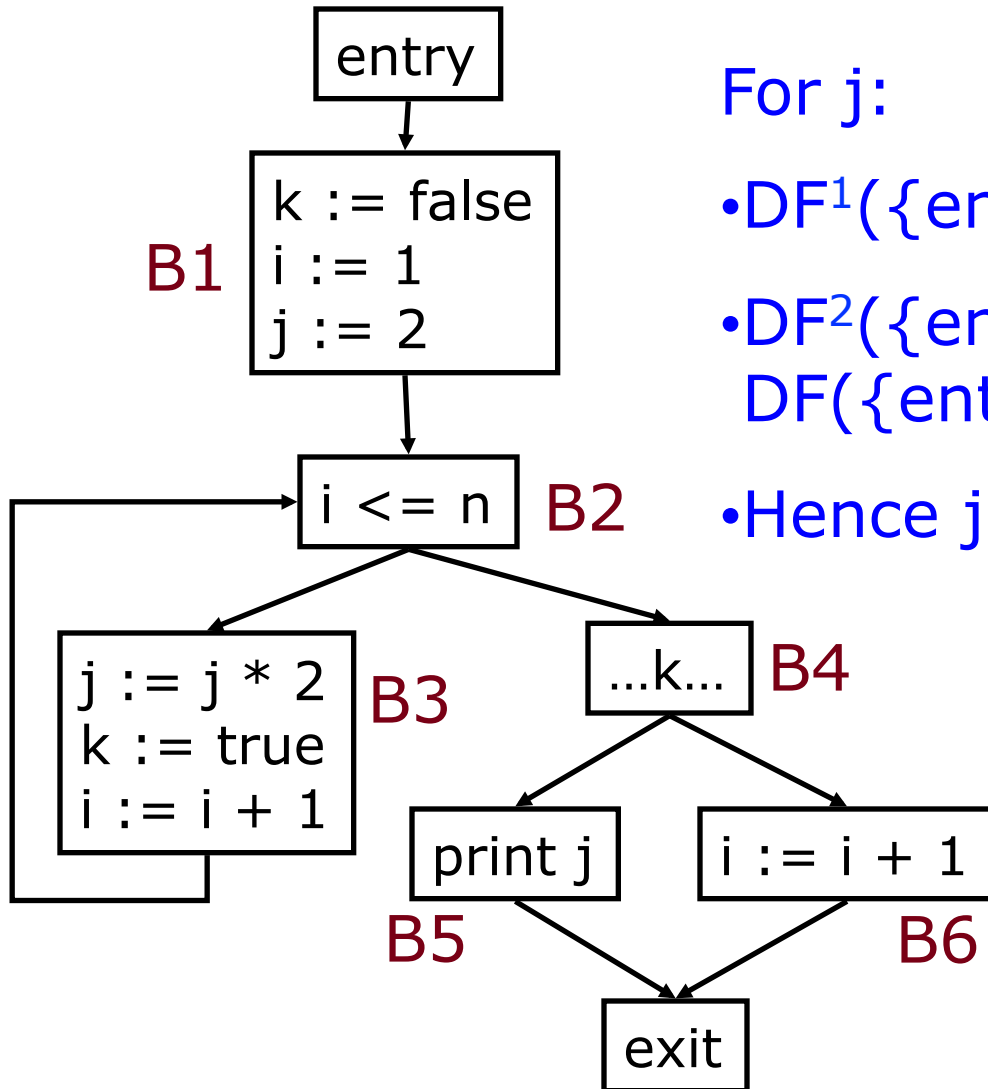
For i :

$$\bullet DF^1(\{\text{entry}, B1, B3, B6\}) = \{\text{B2}, \text{exit}\}$$

$$\bullet DF^2(\{\text{entry}, B1, B3, B6\}) = DF(\{\text{entry}, B1, B2, B3, B6, \text{exit}\}) = \{\text{B2}, \text{exit}\}$$

Hence $ij := \Phi(\dots)$ at B2, exit

Example



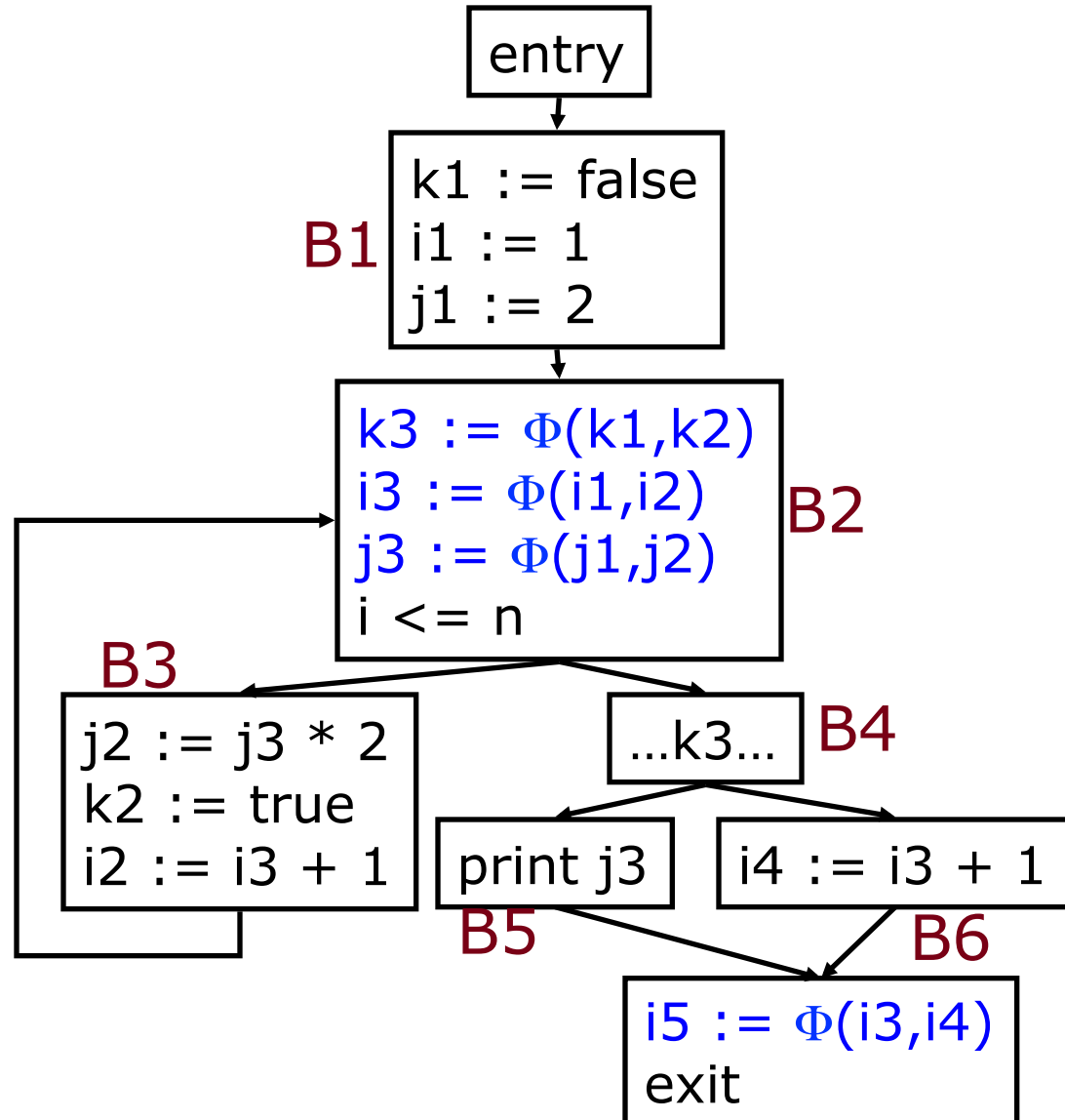
For j:

- $DF^1(\{\text{entry}, B1, B3\}) = \{B2\}$
- $DF^2(\{\text{entry}, B1, B3\}) = DF(\{\text{entry}, B1, B2, B3\}) = \{B2\}$
- Hence $j2 := \Phi(j1, j3)$ at B2

Example, cont' d

So, Φ nodes for i , j , and k are needed in $B2$, and i also needs one in exit

- exit Φ nodes are usually pruned



Other ways to get SSA

Although computing iterated dominance frontiers will result in the minimal SSA form, there are easier ways that work well for simple languages (good structure, no gotos)

Most translators always know when they are creating a join point in the control flow and can keep track of the immediate dominator

If so, it can also create the necessary Φ nodes during translation.

DF criterion more complicated to implement.

SSA by AST value numbering

Walk AST to assign value numbers [Click'95]

- If one predecessor, reuse number
- If more predecessors, add temp Φ'
function add other arguments later
- Loops: complete temp Φ' in 2 rounds
- Finally convert $\Phi' \Rightarrow \Phi$ or remove
superfluous Φ , e.g.
 $a2 := \Phi(a1, a2) \rightarrow a2 := a1 \quad a1 := \Phi(a1) \rightarrow \epsilon$

SSA by value number AST

For each $x=y@z$ (some operator @):

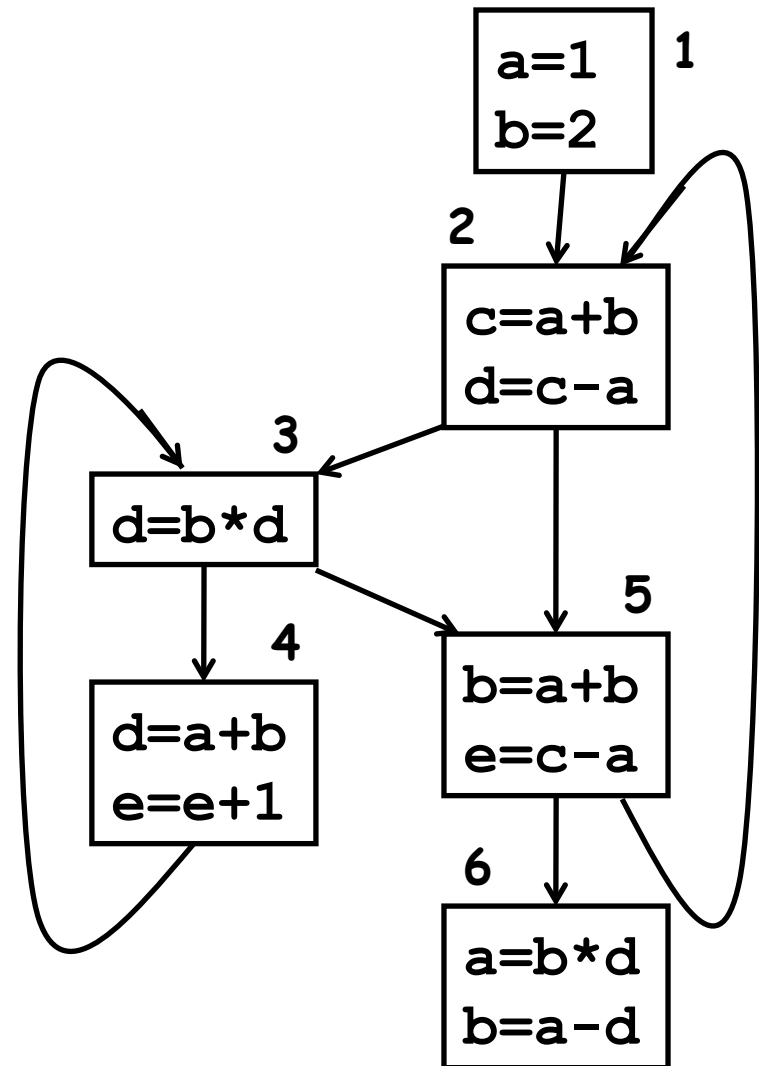
- Compute $VN(y)$ and $VN(z)$
- Compute $VN(@,y,z)$ for $y@z$
- New \rightarrow add $VN(@,y,z)=VN(y)@VN(z)$
- Put $VN(@,y,z)$ into $VN(x)$
- This performs CSE already

VN(y) implementation for SSA

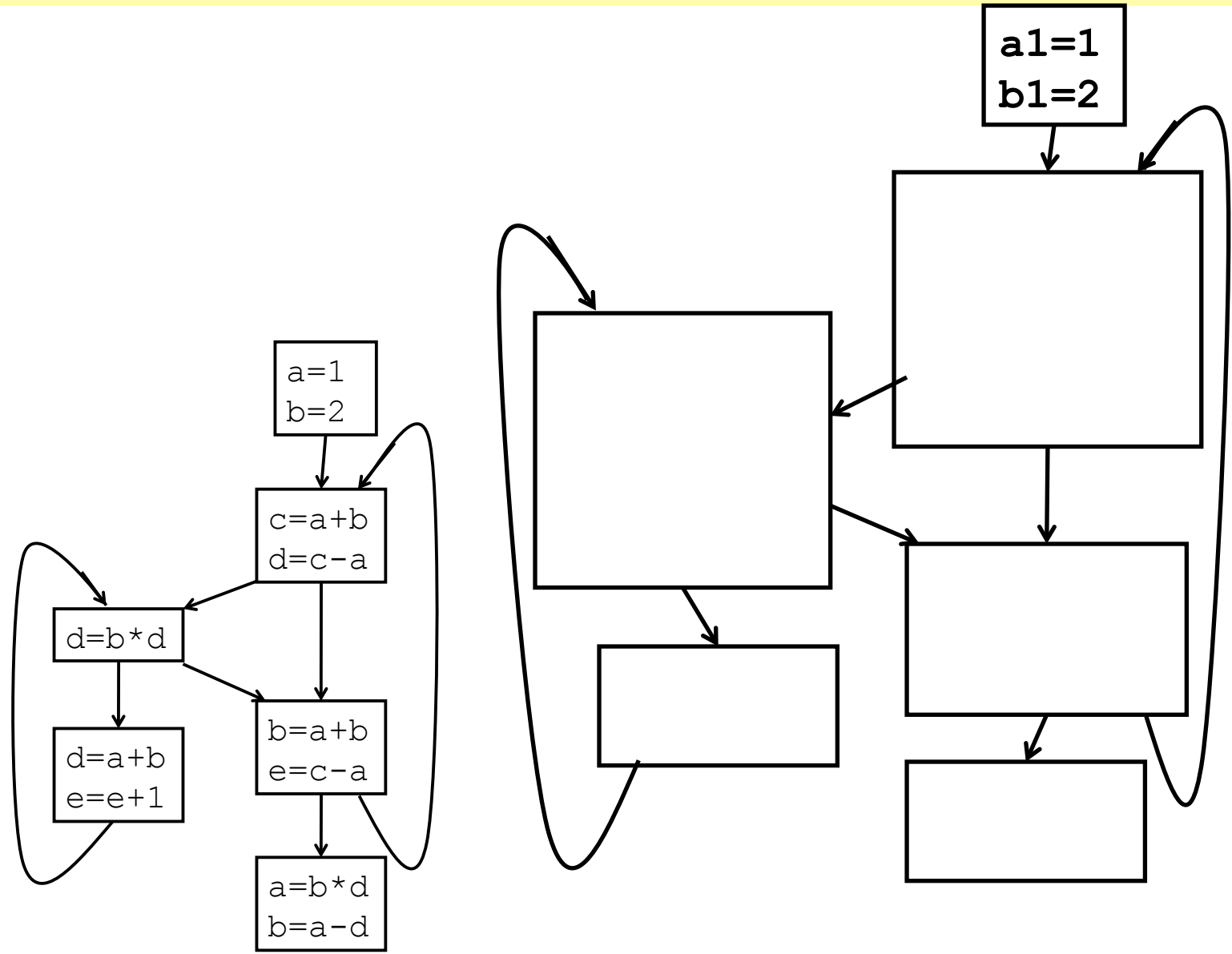
- Basic block has value w for $y \Rightarrow$ use
- Exactly one predecessor \Rightarrow reuse $VN(y)$ of predecessor
- More predecessors \Rightarrow
 - get $w_i = VN(y)$ at each predecessor p_i
 - Add $VN(\Phi, y, y) = \Phi(w_1, w_2, \dots, w_n)$
Nice: VN only adds Φ if still live
 - Put $VN(\Phi, y, y)$ into $VN(y)$ for y

Basic block control flow graph

```
a=1;
b=2;
while (true) {
  c=a+b;
  if ((d=c-a)!=0) {
    while((d=b*d)!=0) {
      d=a+b;
      e=e+1;
    }
    b=a+b;
    if ((e=c-a)!=0) break;
  }
  a=b*d;
  b=a-d;
}
```

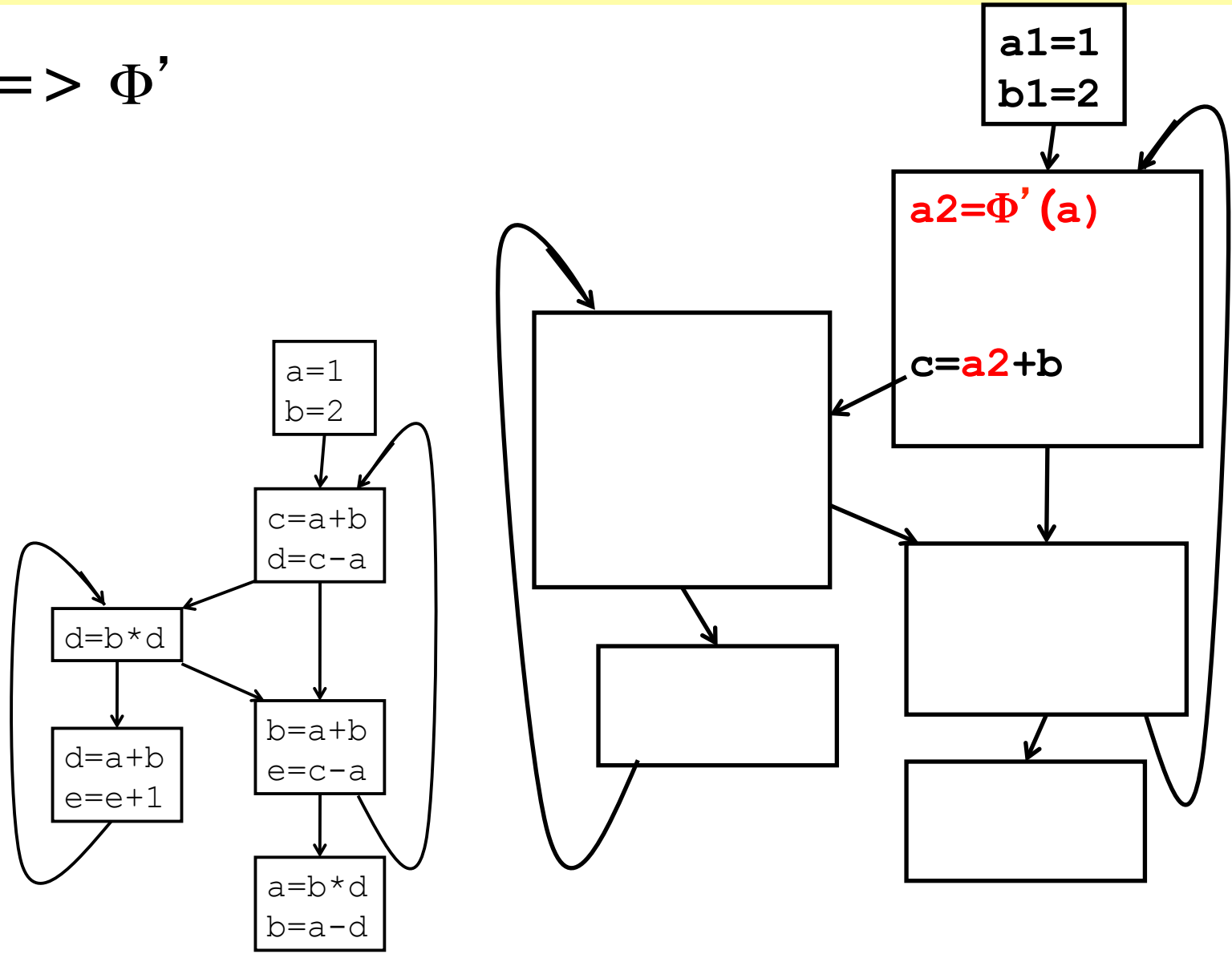


SSA AST-walk construction (1)



SSA AST-walk construction (2)

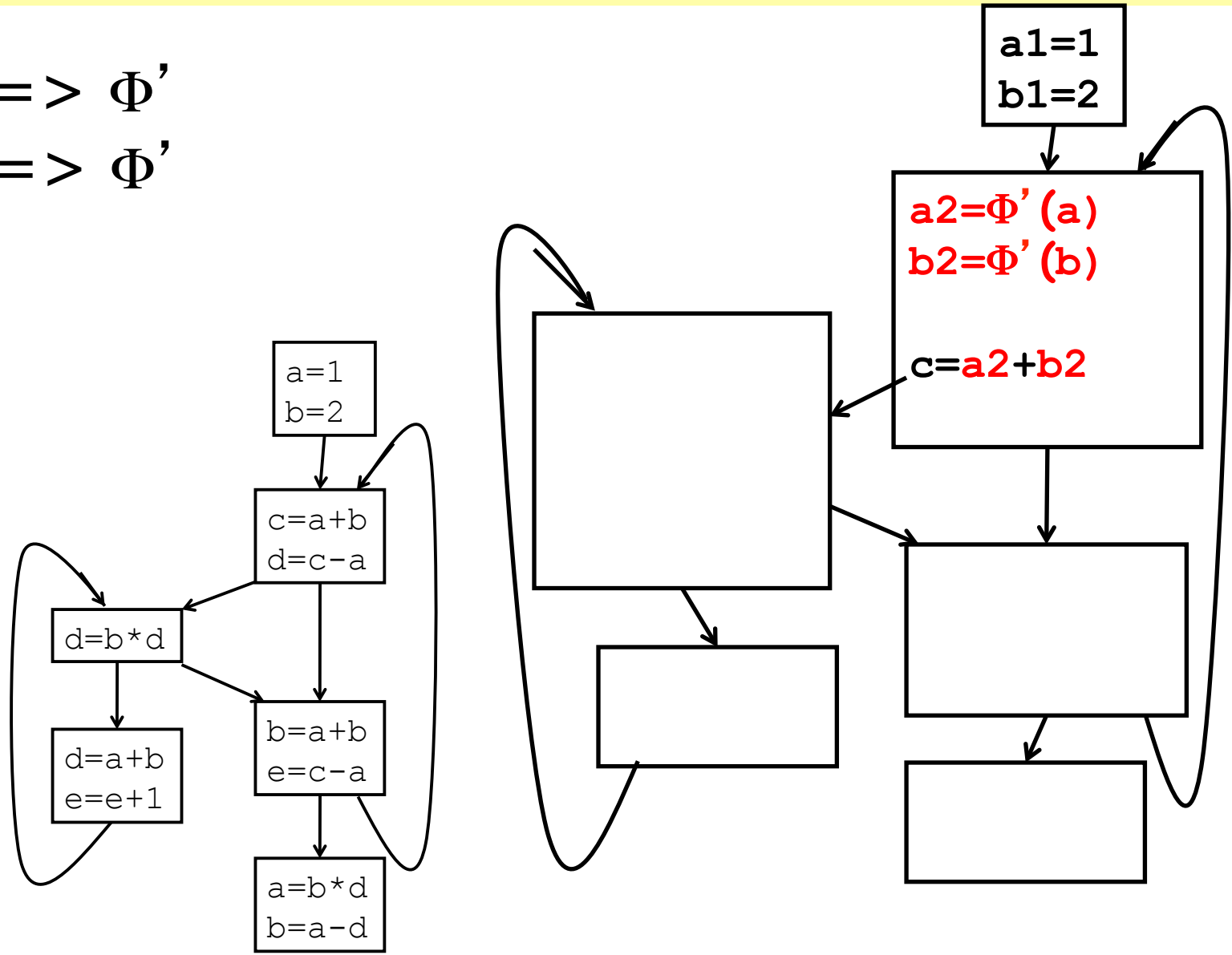
$VN(a) \Rightarrow \Phi'$



SSA AST-walk construction (2)

$VN(a) \Rightarrow \Phi'$

$VN(b) \Rightarrow \Phi'$



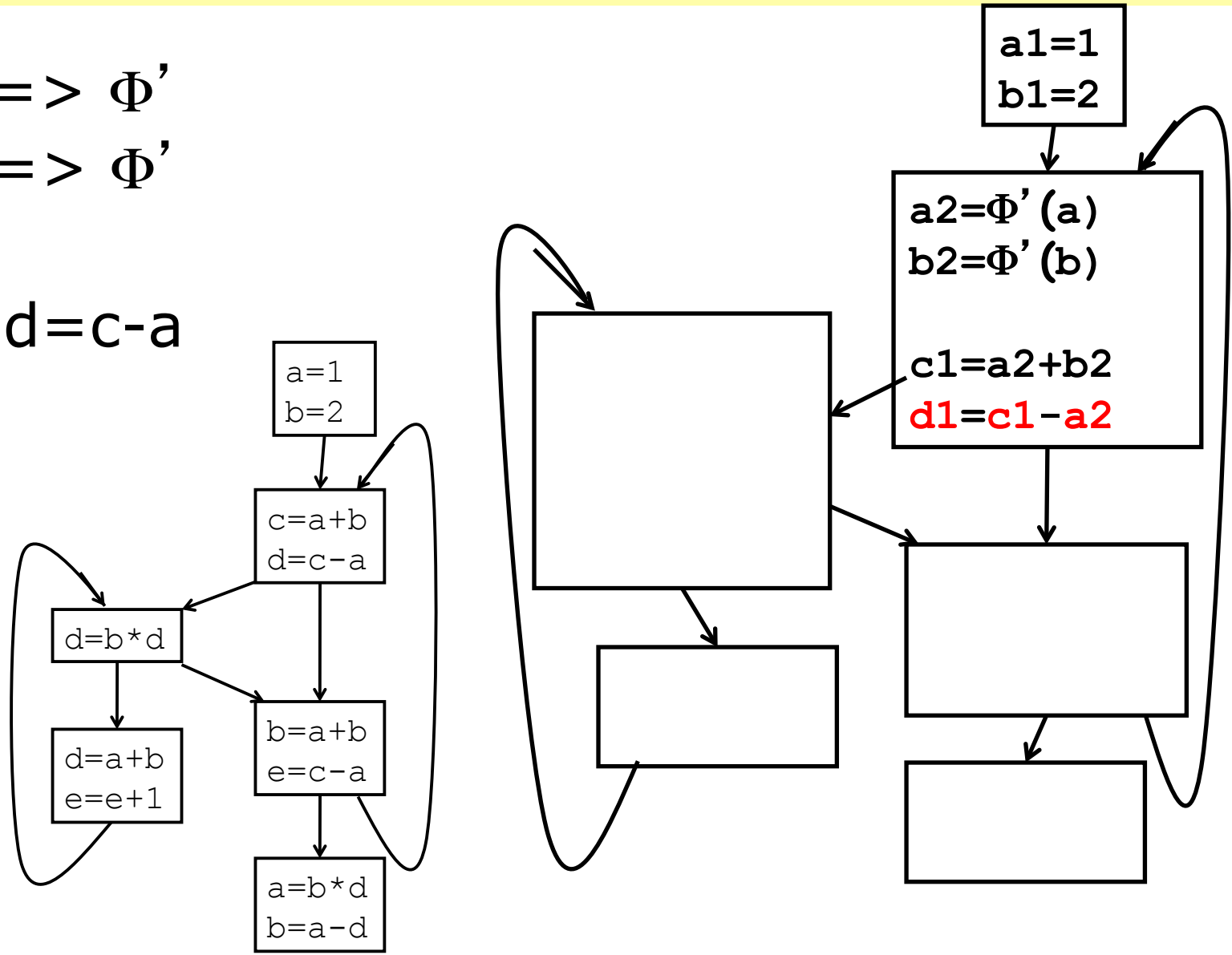
SSA AST-walk construction (2)

$VN(a) \Rightarrow \Phi'$

$VN(b) \Rightarrow \Phi'$

$VN(c)$

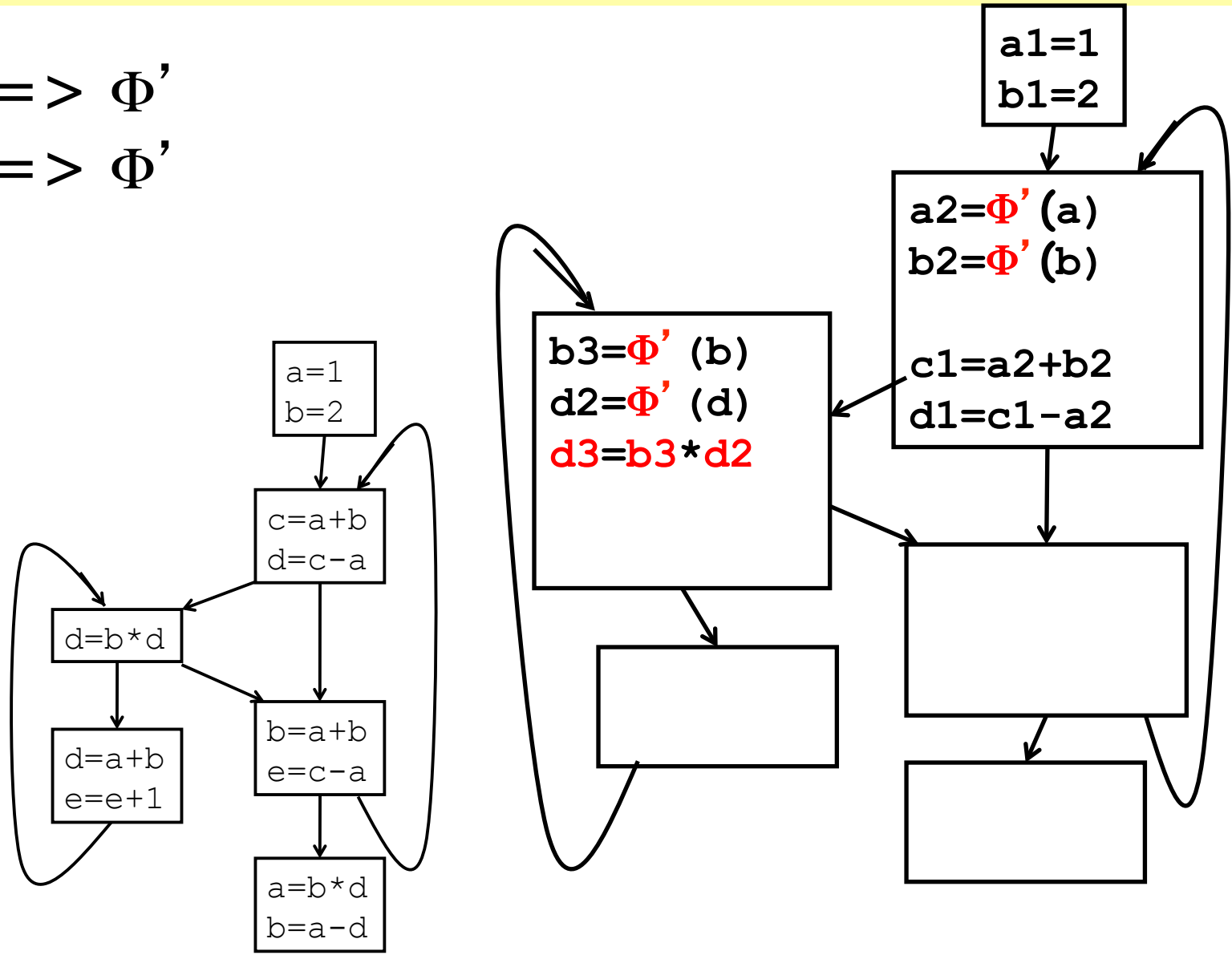
VN for $d=c-a$



SSA AST-walk construction (3)

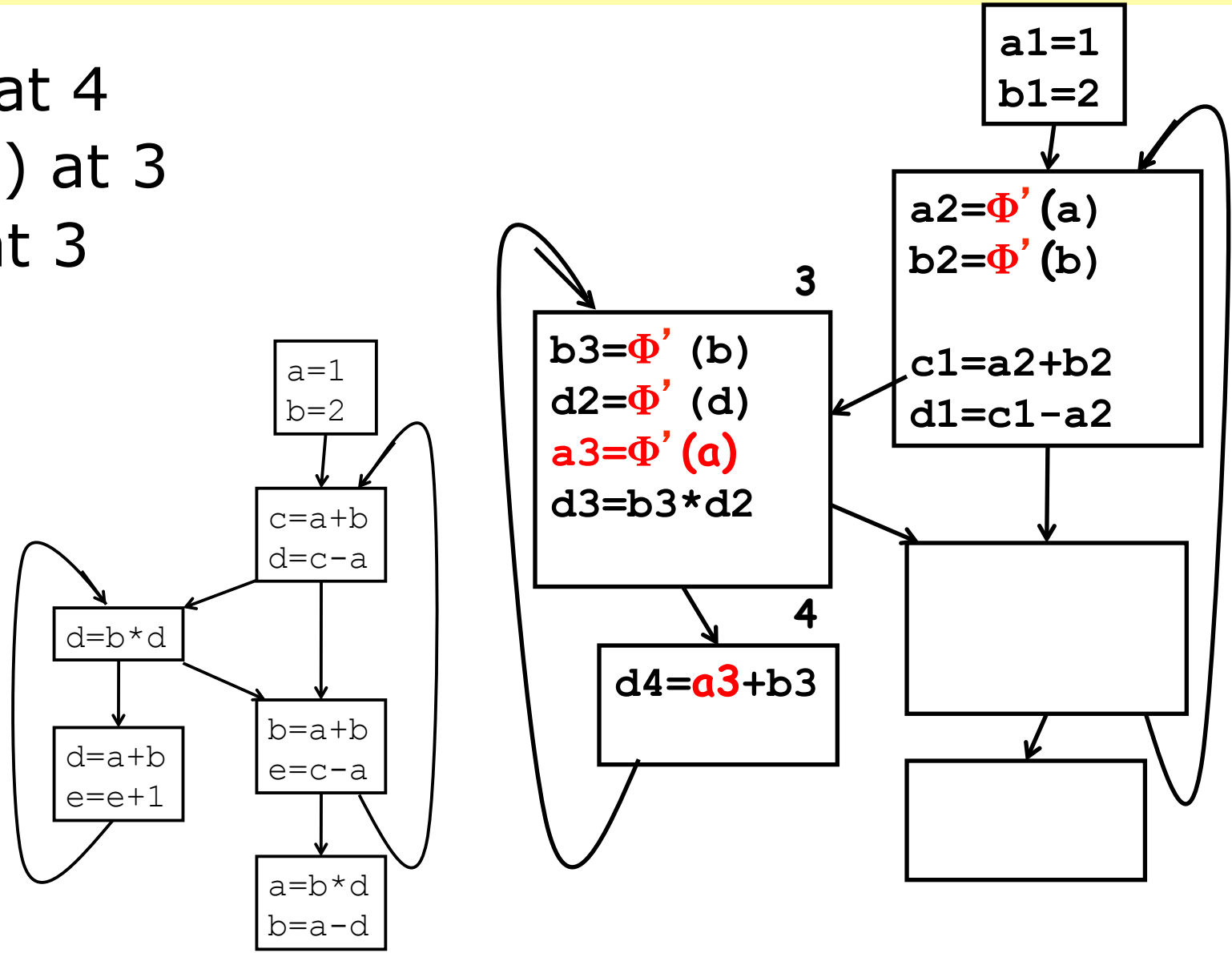
$VN(b) \Rightarrow \Phi'$

$VN(d) \Rightarrow \Phi'$



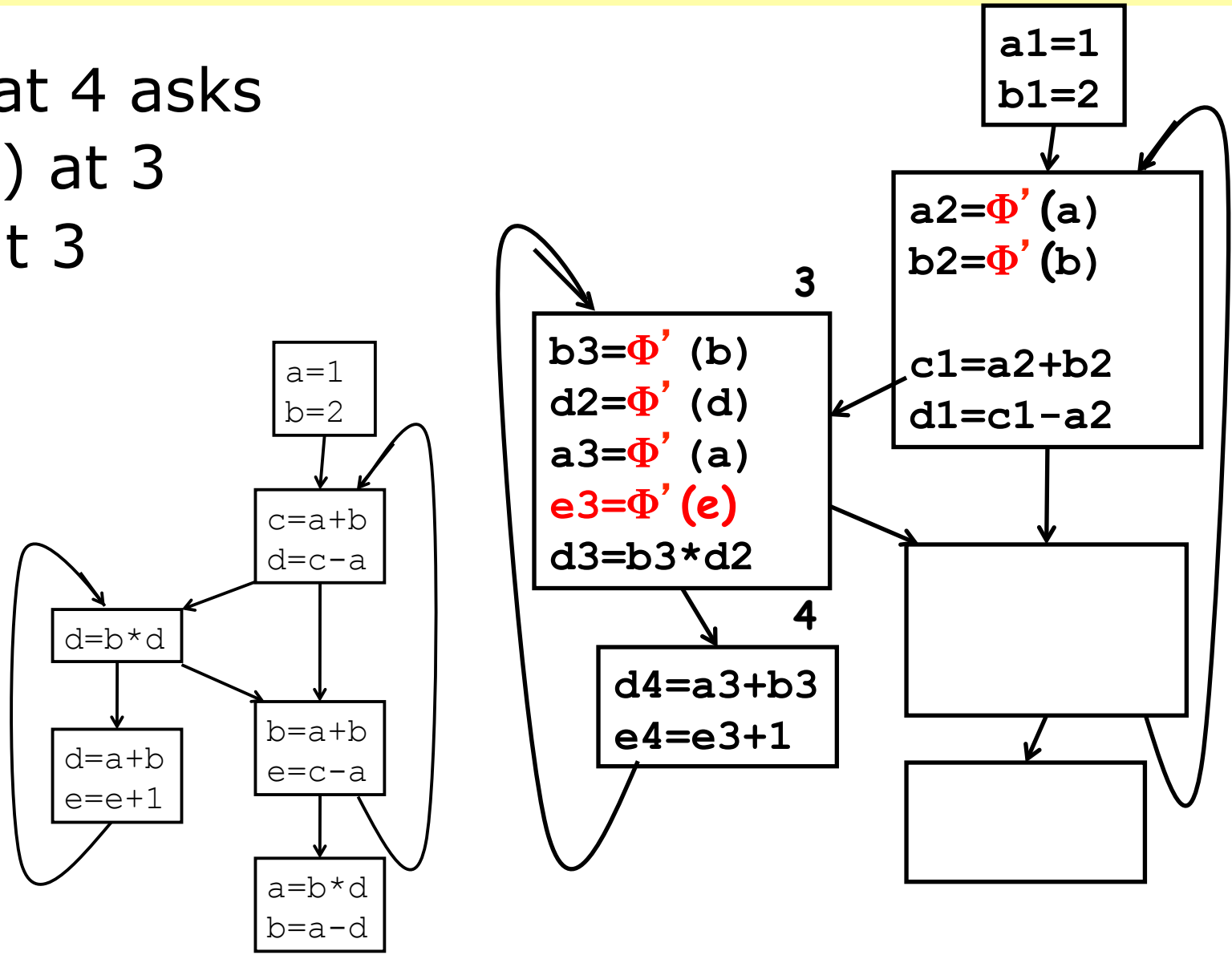
SSA AST-walk construction (4)

VN(a) at 4
⇒ VN(a) at 3
⇒ Φ' at 3



SSA AST-walk construction (4)

VN(e) at 4 asks
⇒ VN(e) at 3
⇒ Φ' at 3



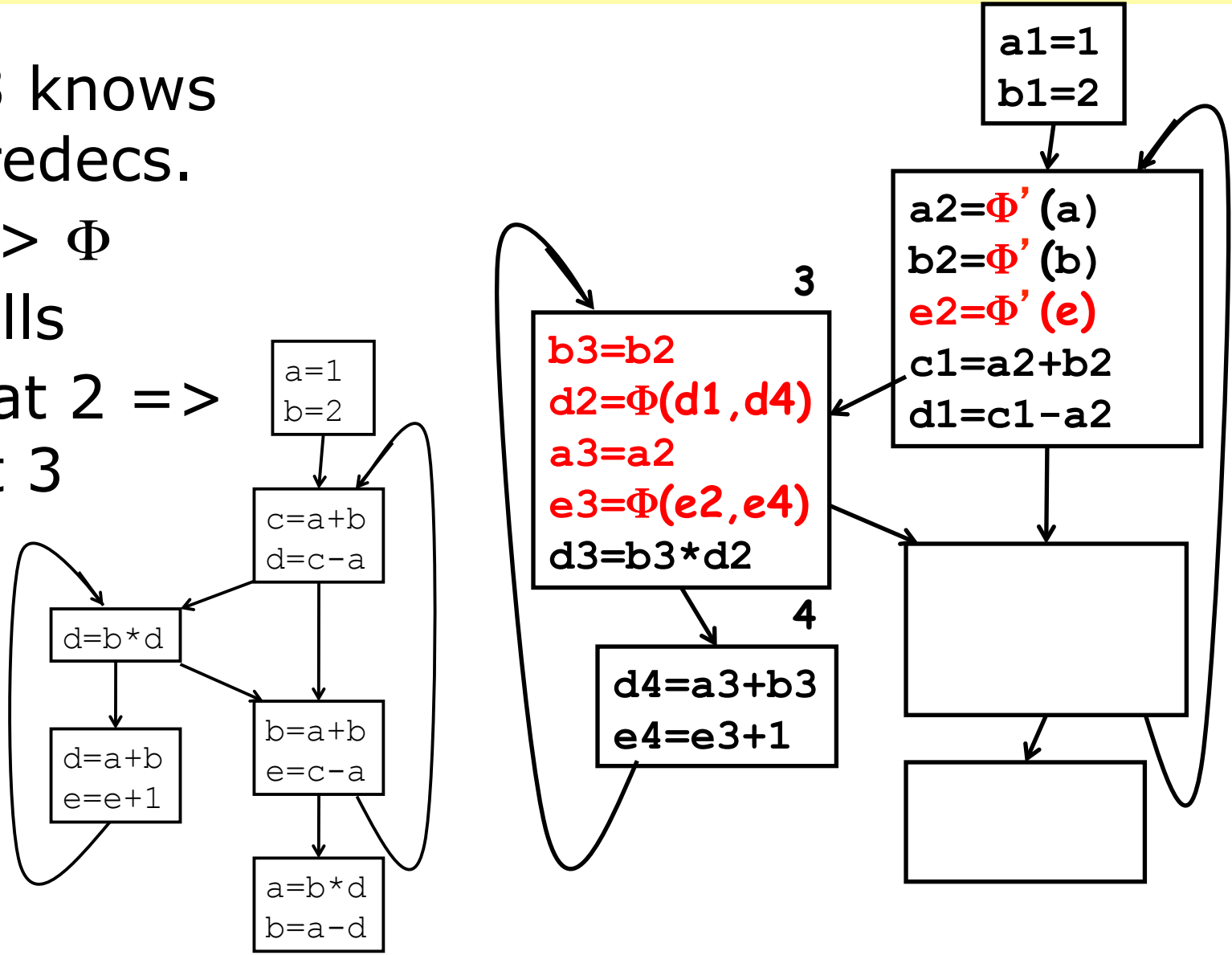
SSA AST-walk construction (4)

Φ' at 3 knows
all predecs.

$\Rightarrow \Phi' \Rightarrow \Phi$

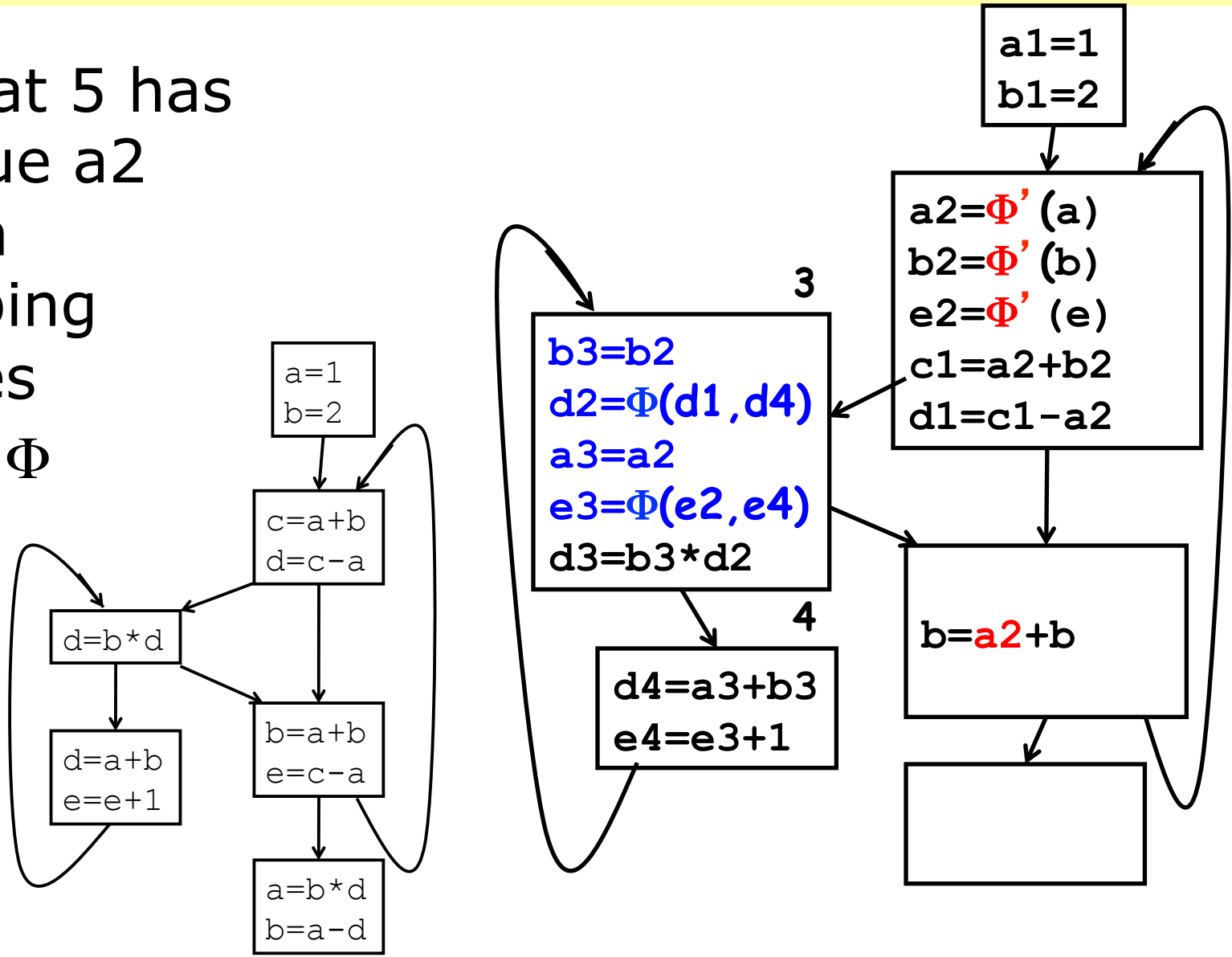
This calls

$VN(e)$ at 2 \Rightarrow
 Φ' at 3



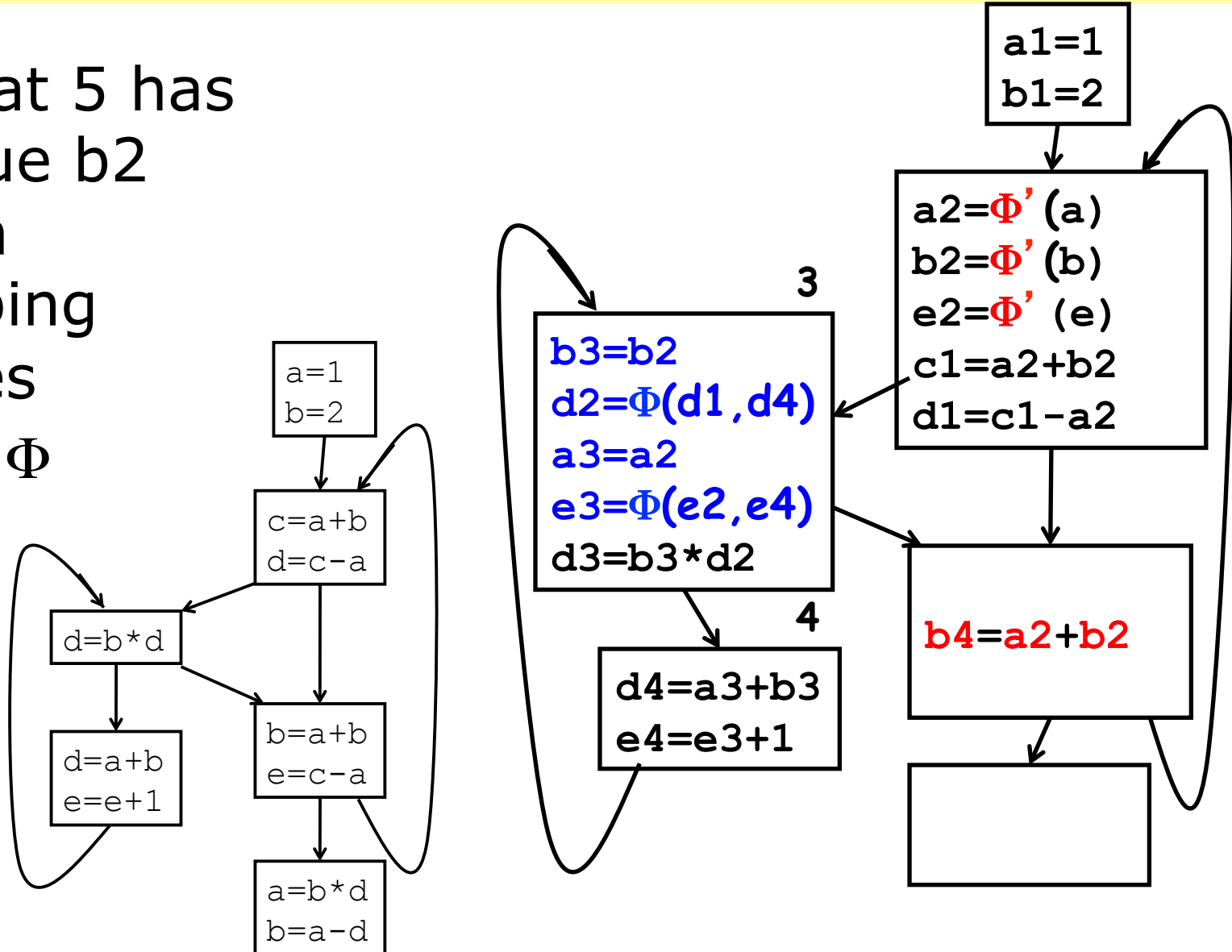
SSA AST-walk construction (5)

VN(a) at 5 has
 unique a2
 when
 skipping
 copies
 => no Φ

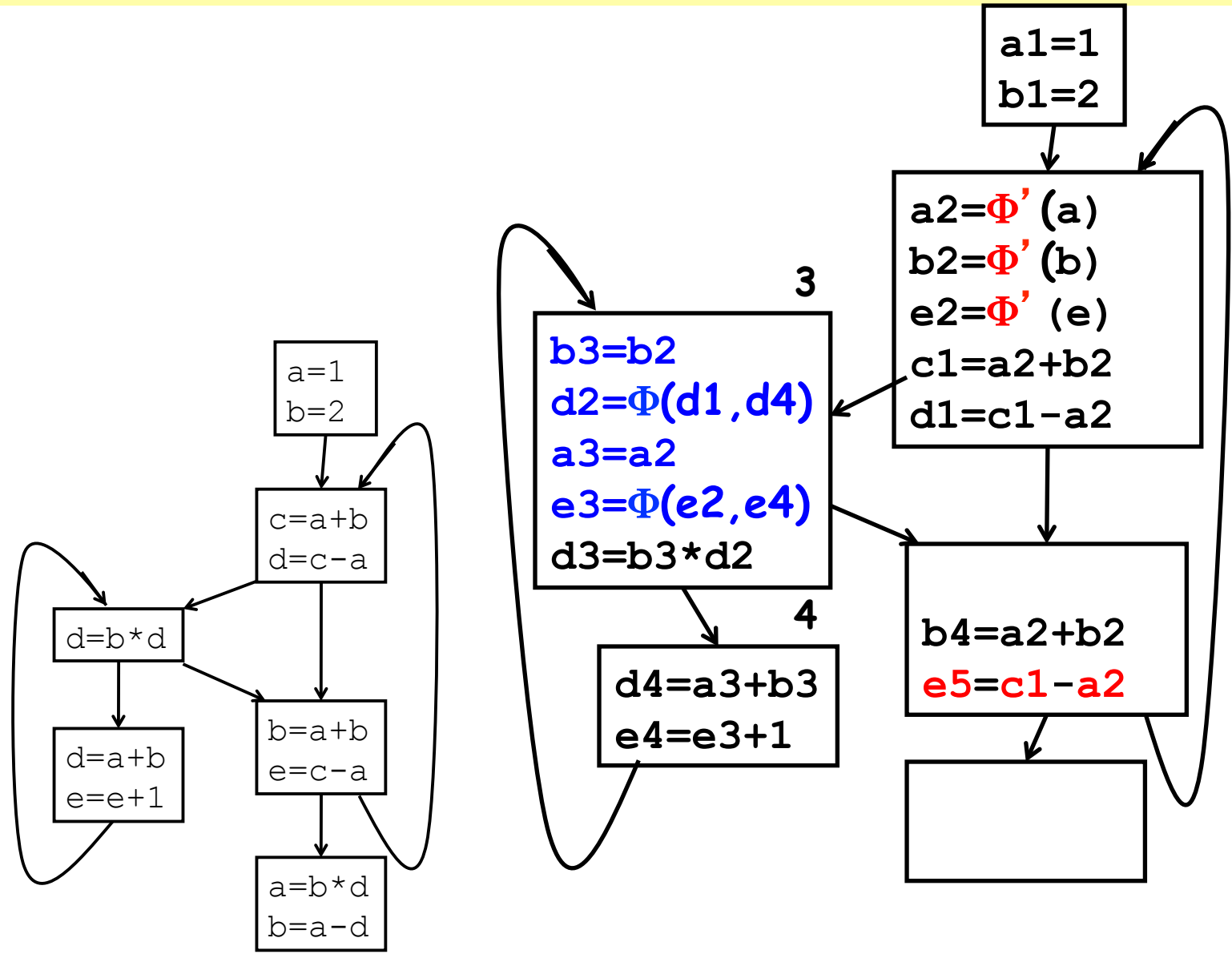


SSA AST-walk construction (5)

VN(b) at 5 has
 unique b2
 when
 skipping
 copies
 => no Φ



SSA AST-walk construction (5)

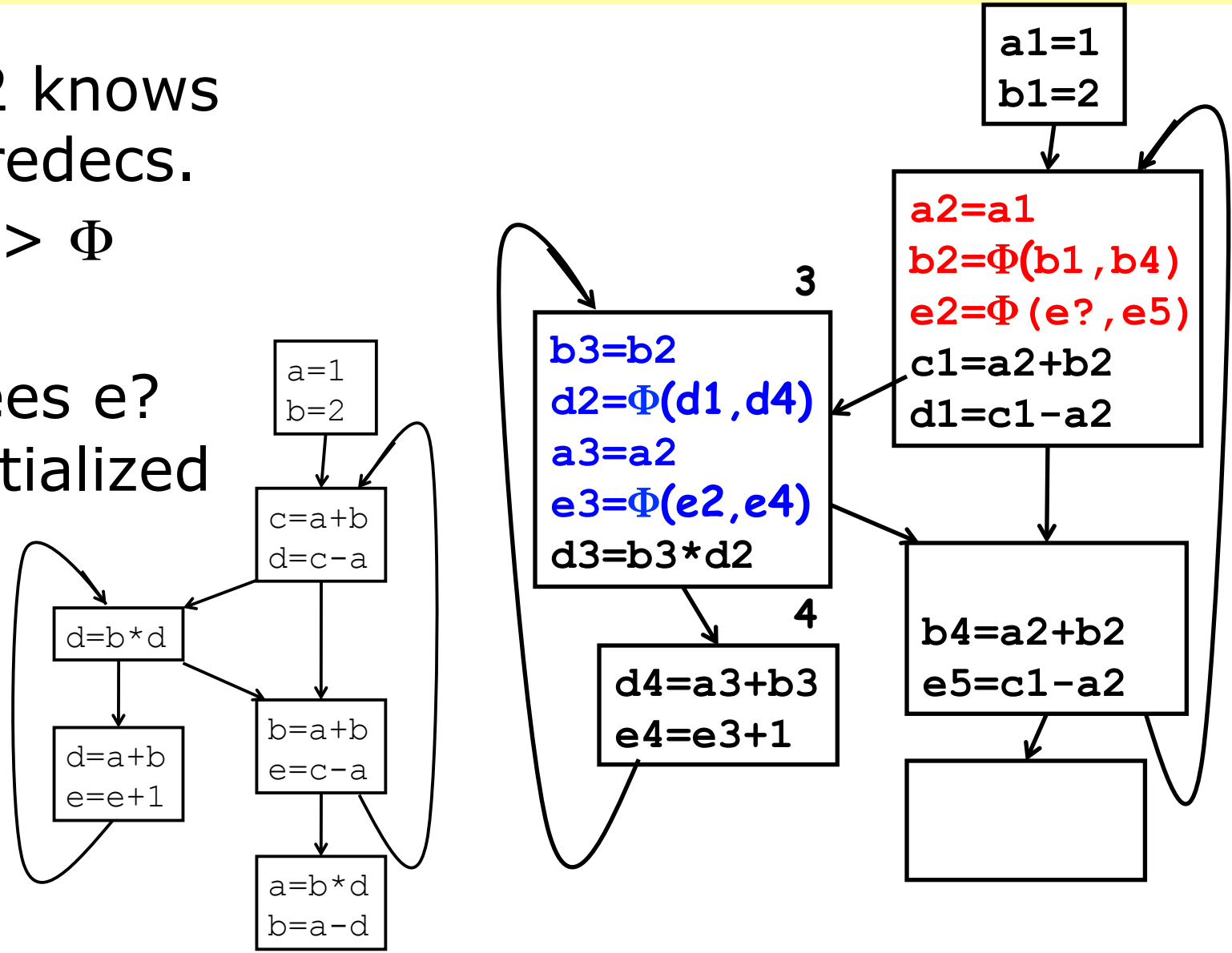


SSA AST-walk construction (5)

Φ' at 2 knows
all predecs.

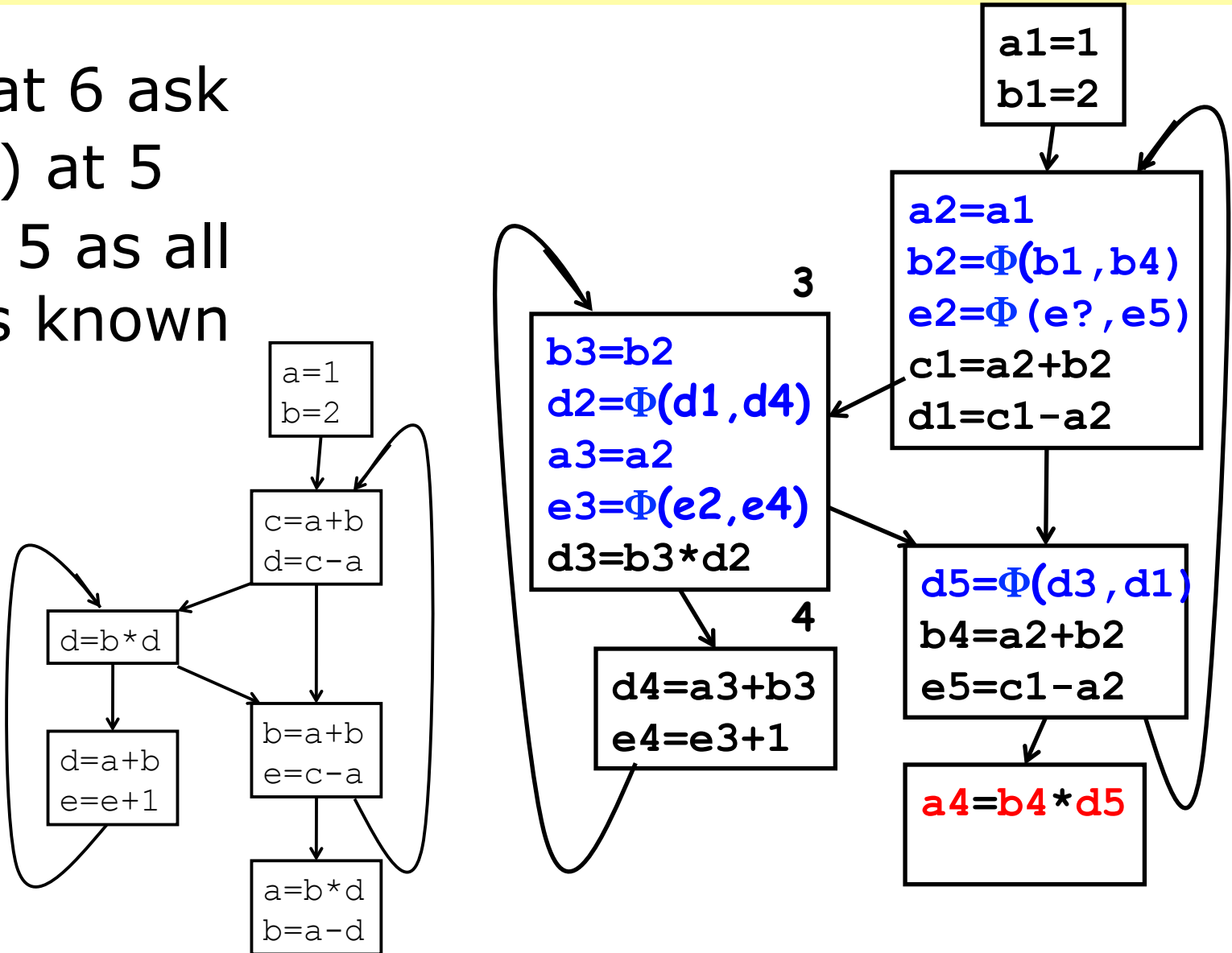
$\Rightarrow \Phi' \Rightarrow \Phi$

SSA sees $e?$
uninitialized

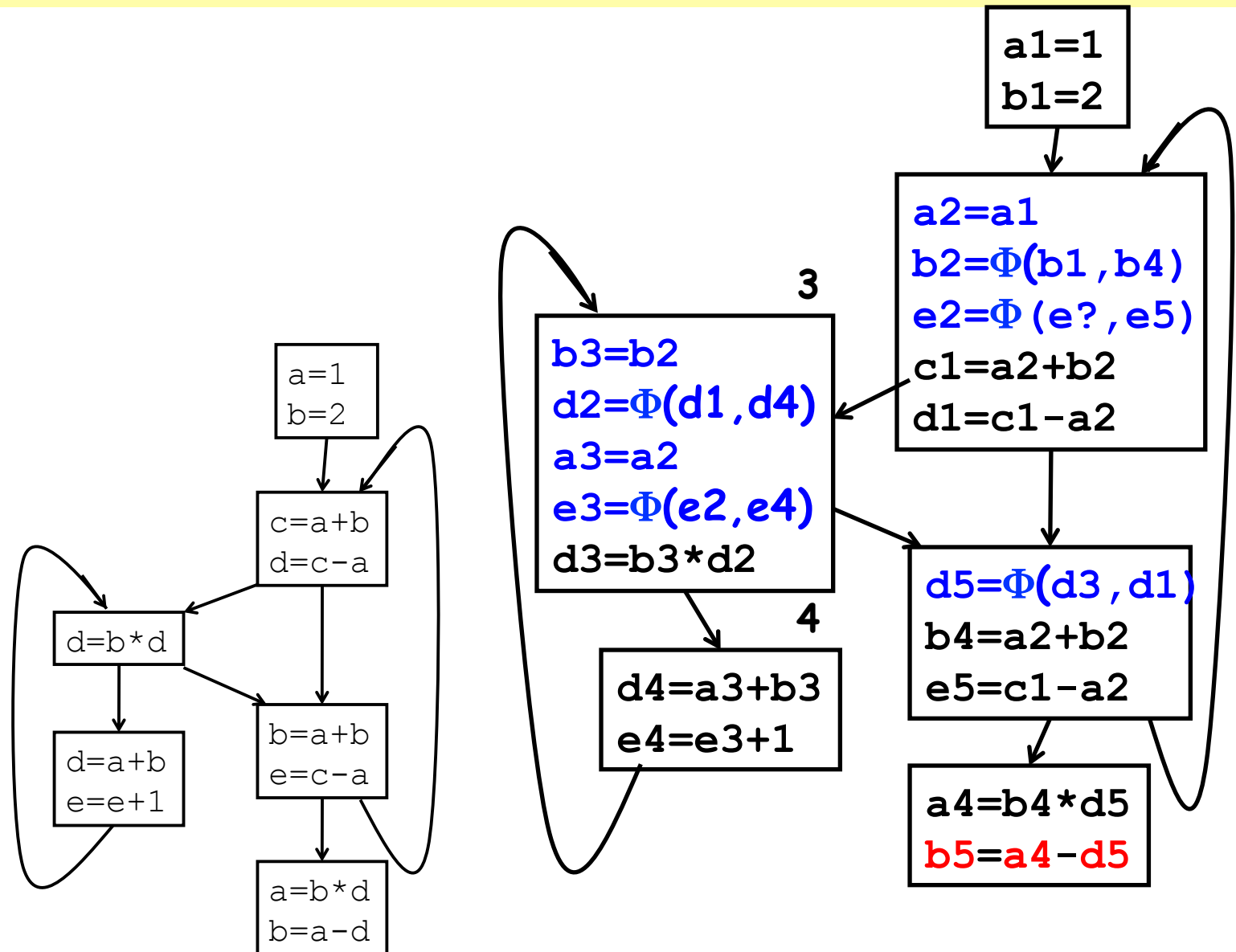


SSA AST-walk construction (6)

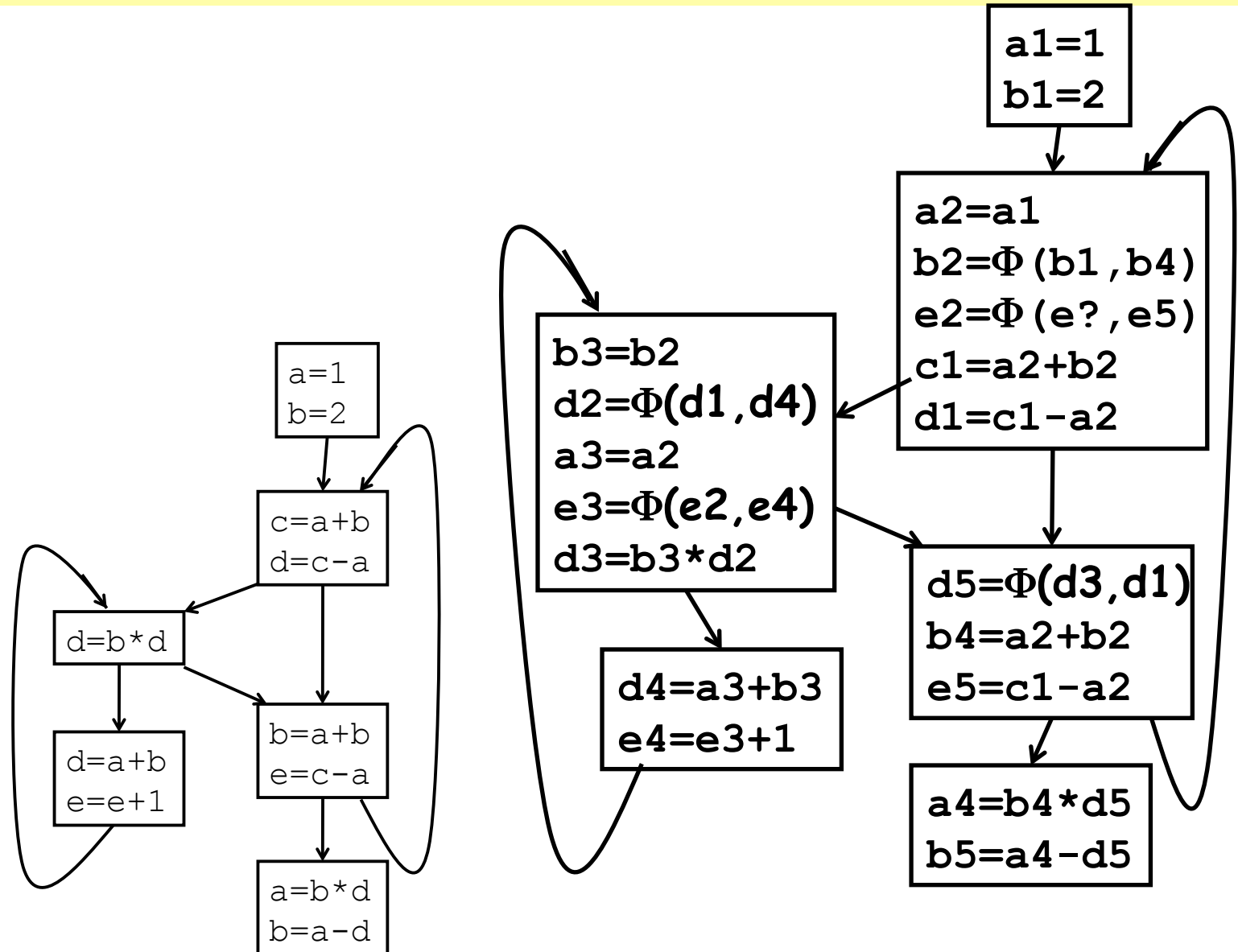
VN(d) at 6 ask
 \Rightarrow VN(d) at 5
 $\Rightarrow \Phi$ at 5 as all
 preds known



SSA AST-walk construction (6)

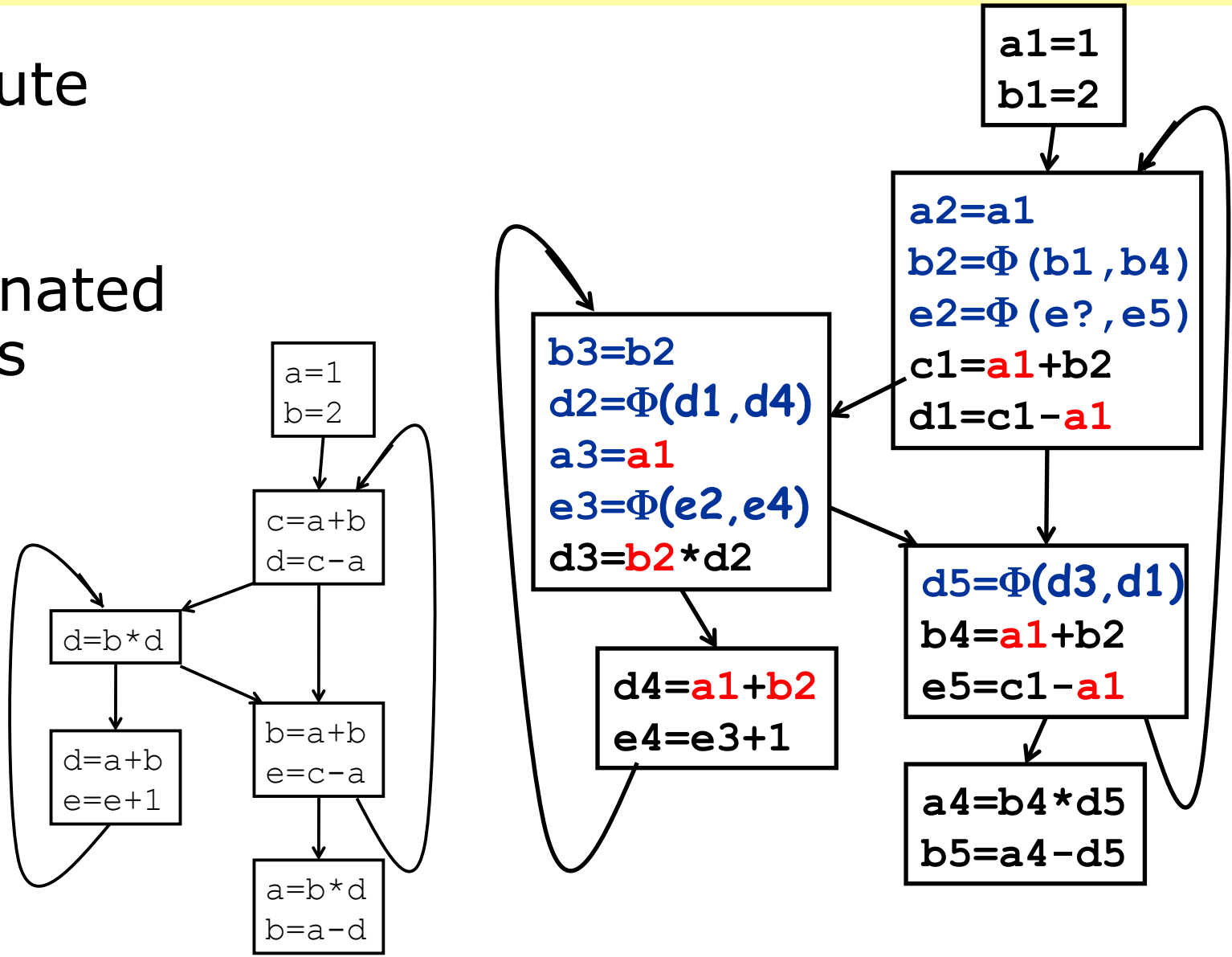


SSA AST-walk construction



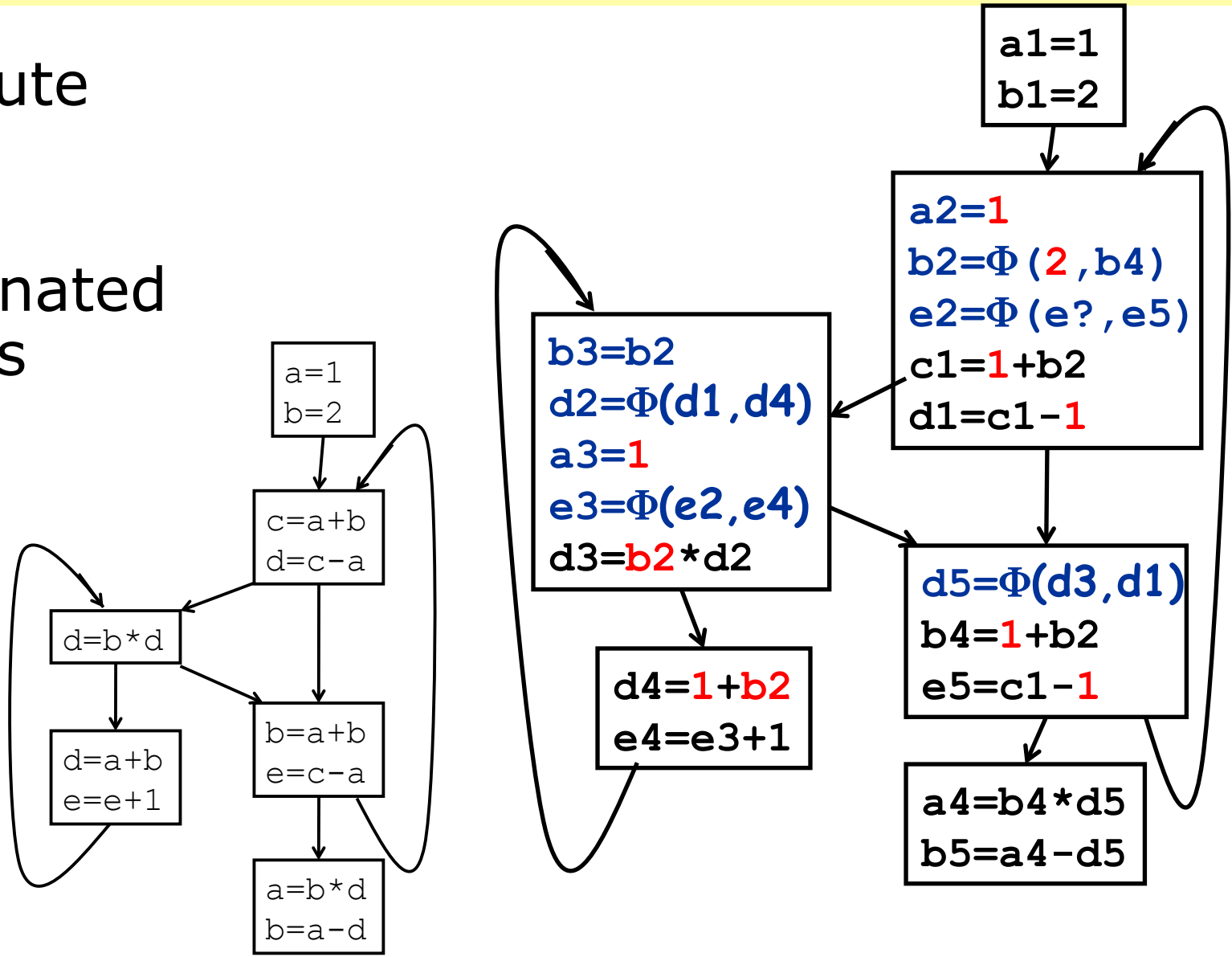
SSA Opt: copy propagate

substitute
 $a_2 = a_1$
into all
dominated
nodes



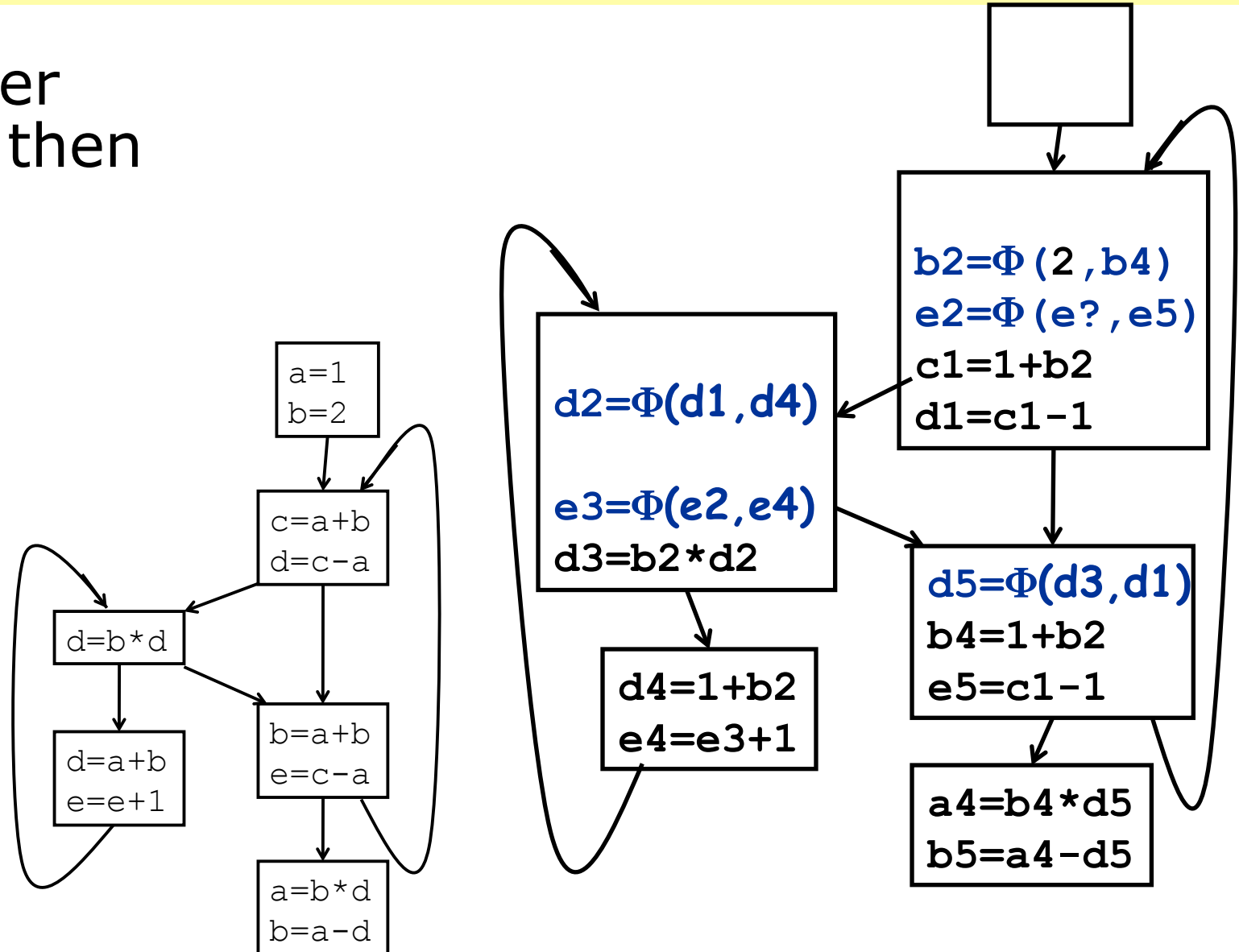
SSA Opt: constant propagate

substitute
 $a_1=1$
into all
dominated
nodes



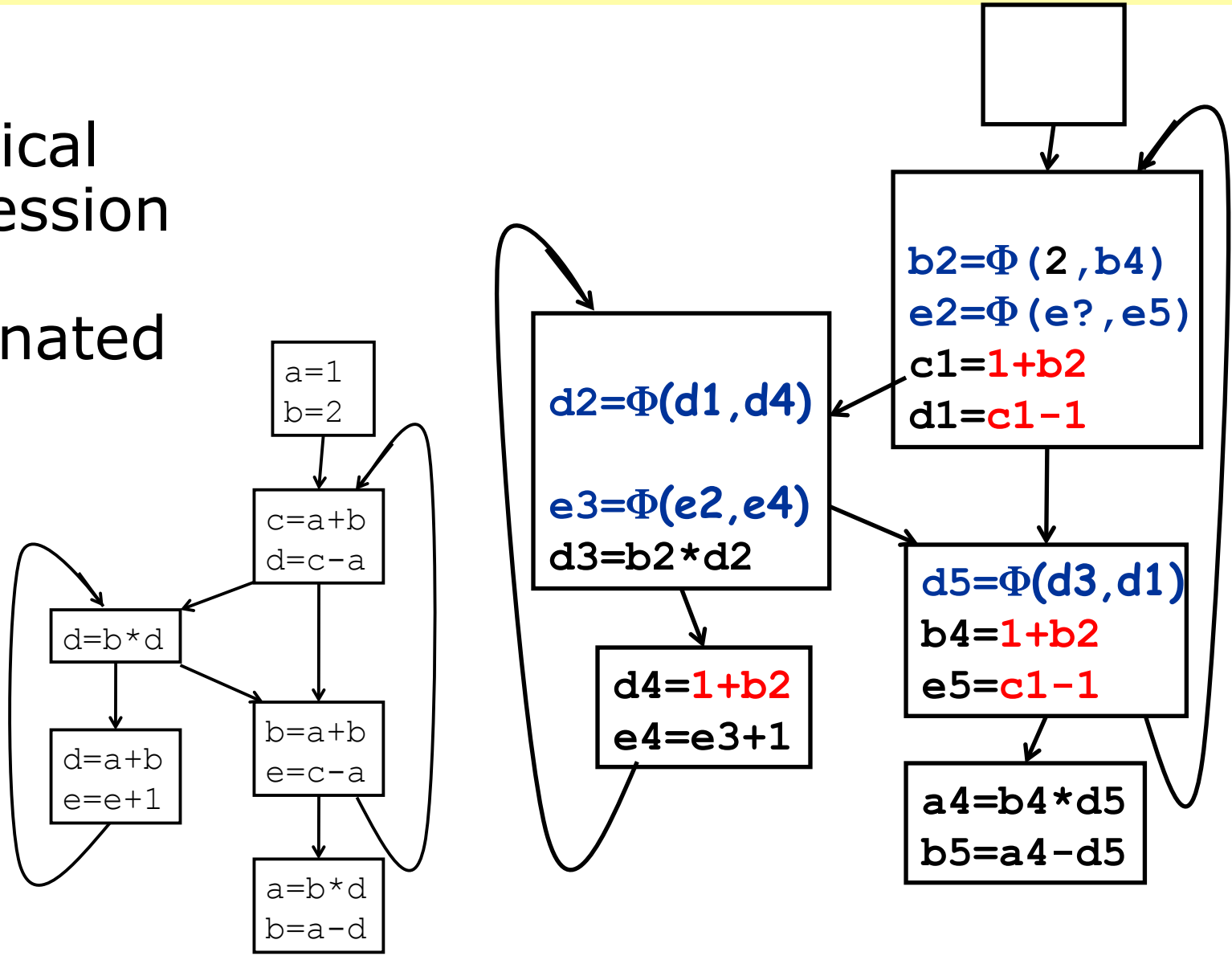
SSA Opt: eliminate dead code

a1 never
read then
dead



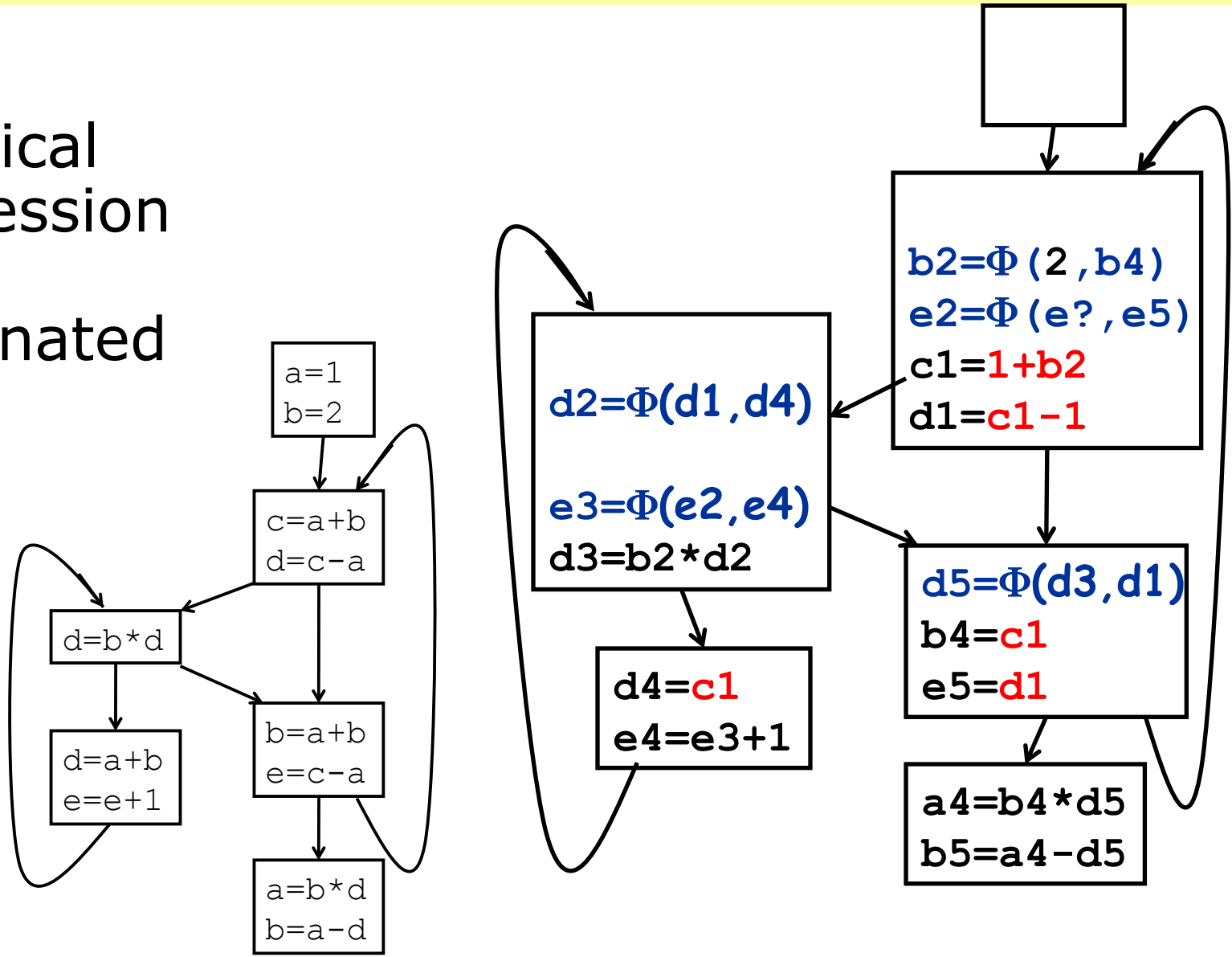
SSA Opt: CSE

reuse
identical
expression
if
dominated

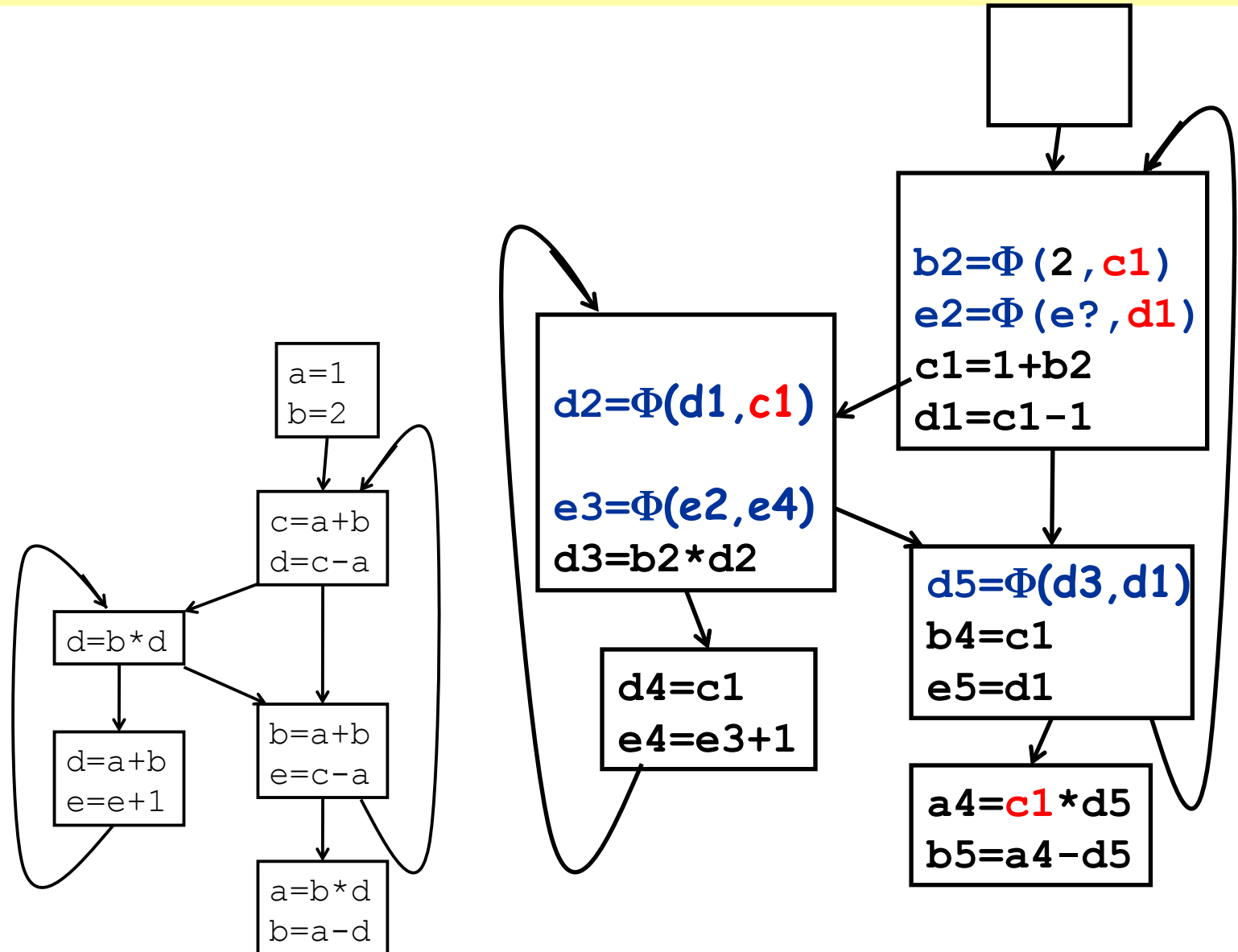


SSA Opt: CSE

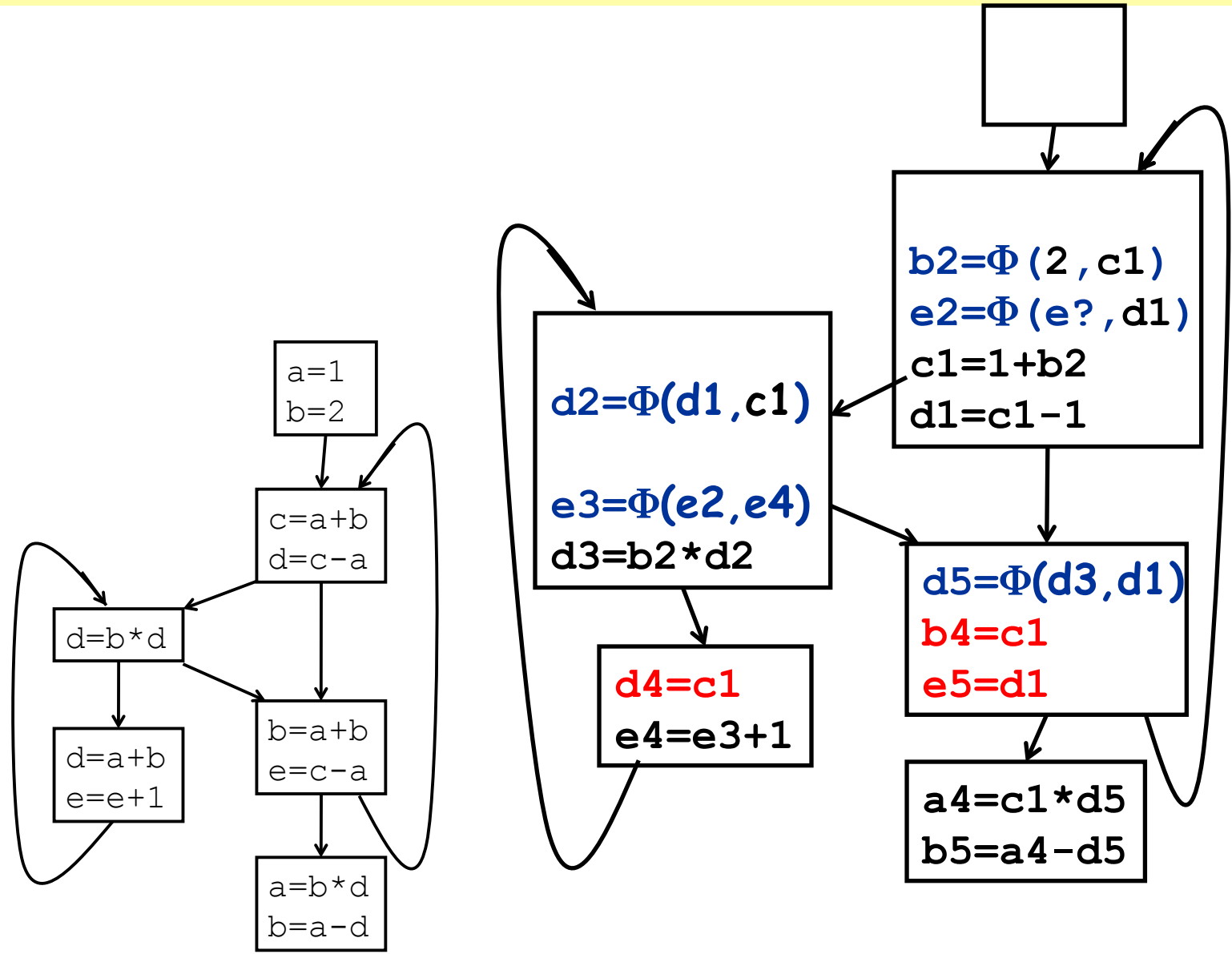
reuse
identical
expression
if
dominated



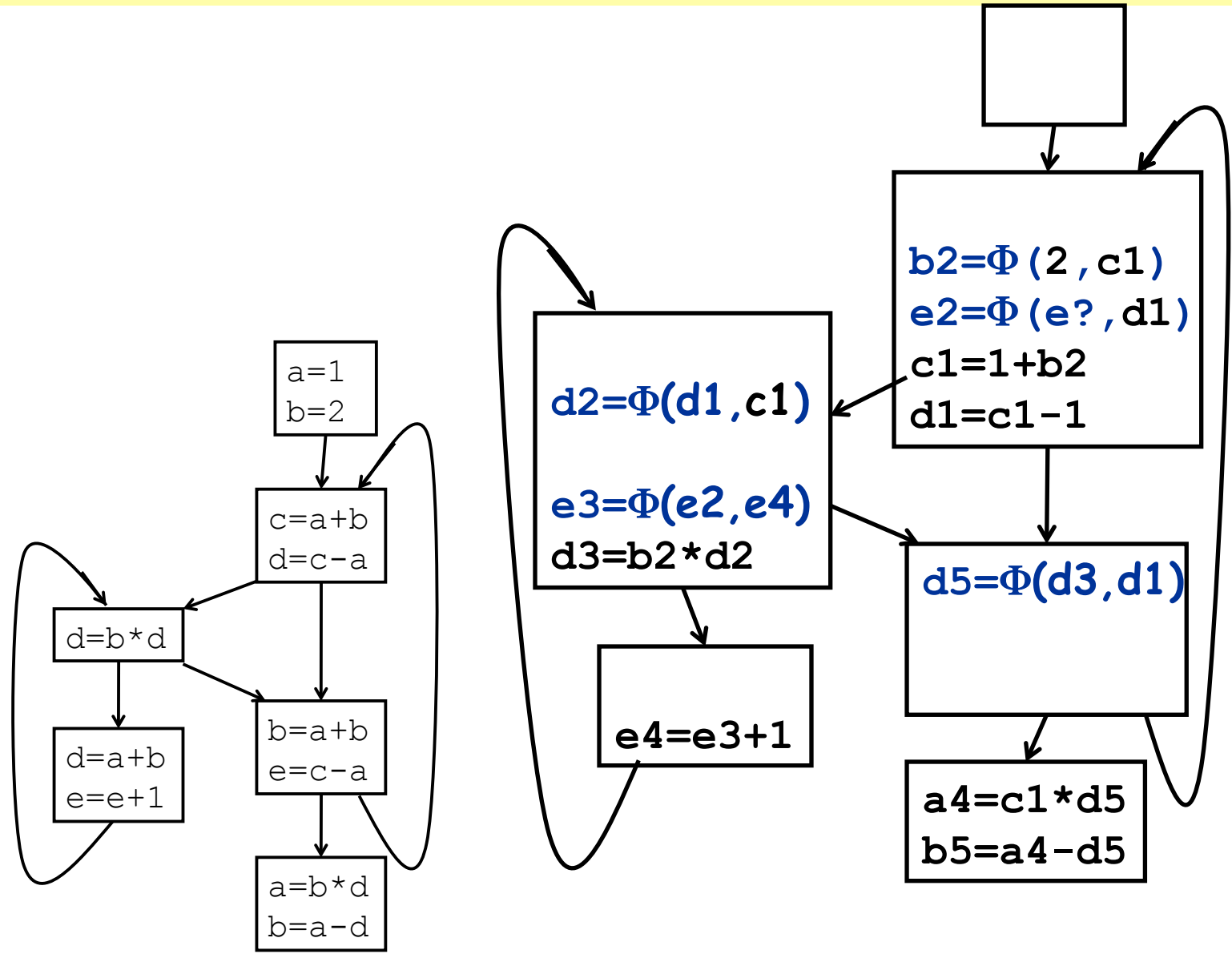
SSA Opt: copy propagate



SSA Opt: eliminate dead code



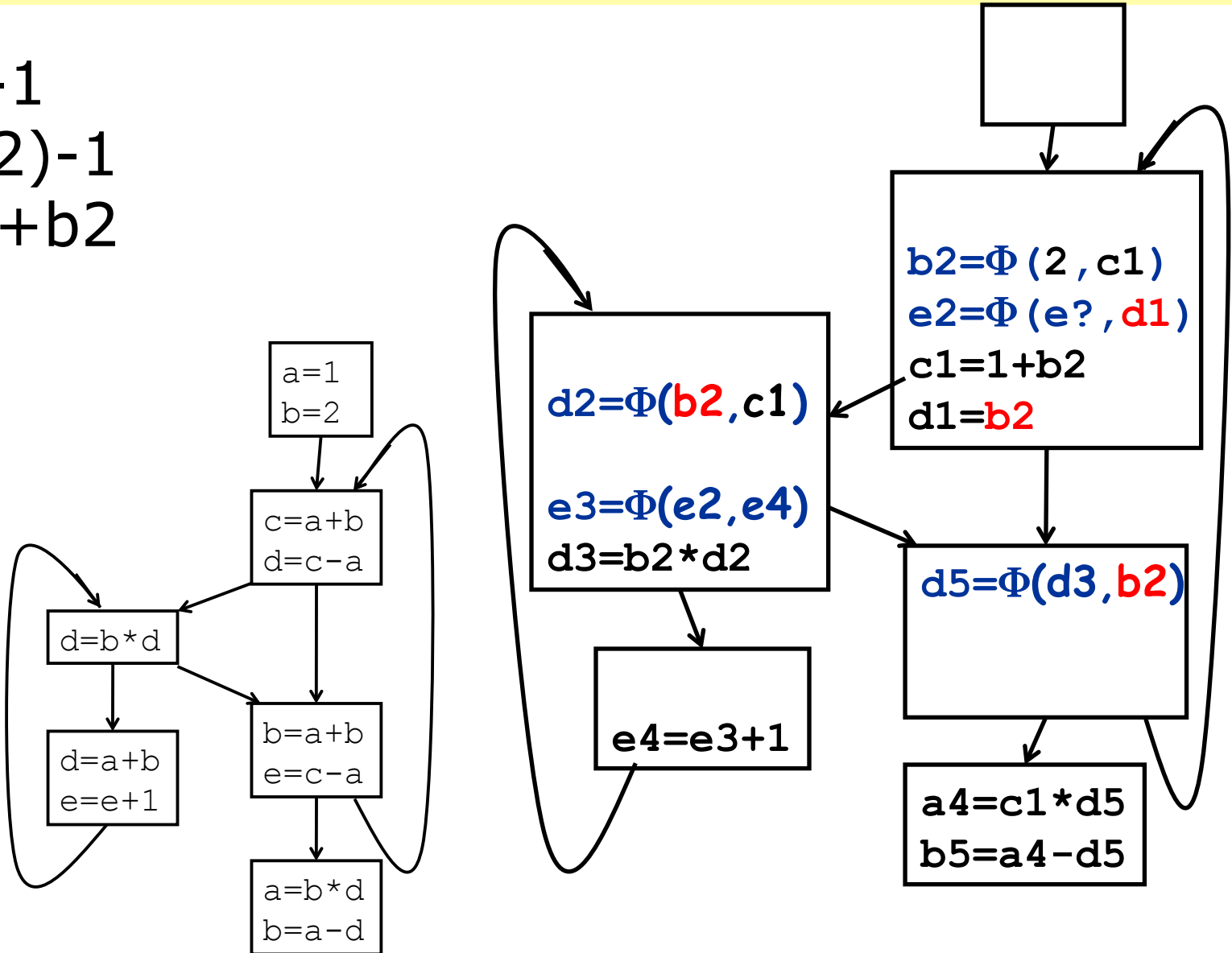
SSA Opt: eliminate dead code



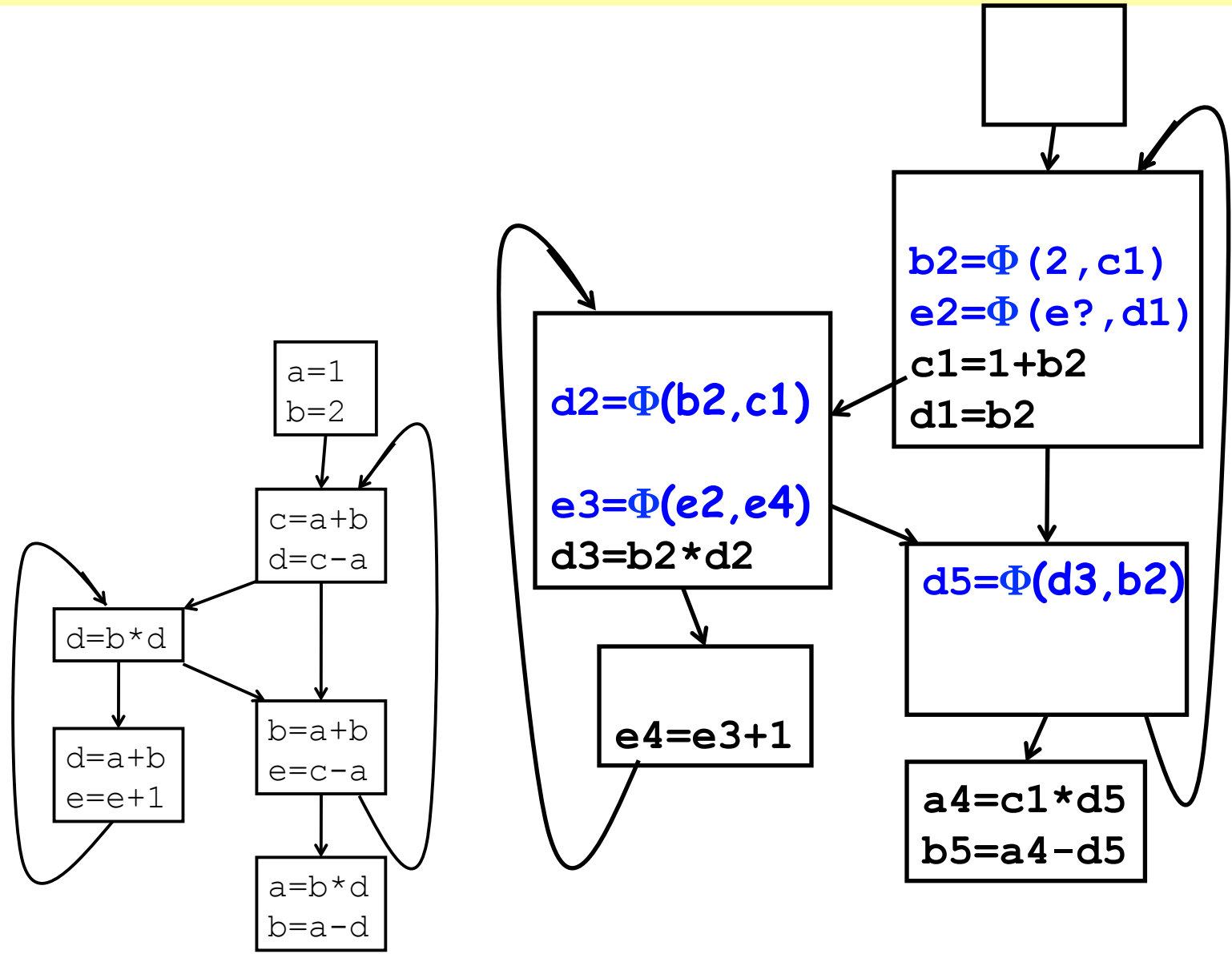
SSA Opt: arithmetic

$d1 = c1 - 1$
 $= (1 + b2) - 1$
 $= (1 - 1) + b2$
 $= 0 + b2$
 $= b2$
 good?

flags?
 floats?



SSA Optimized

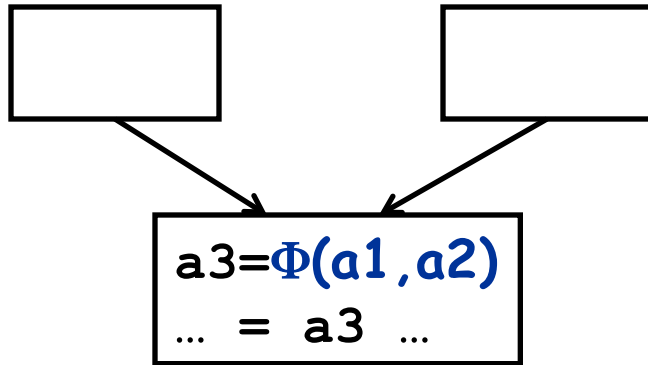


DeSSA: get rid of Φ wrong way

DeSSA can't just drop numbers

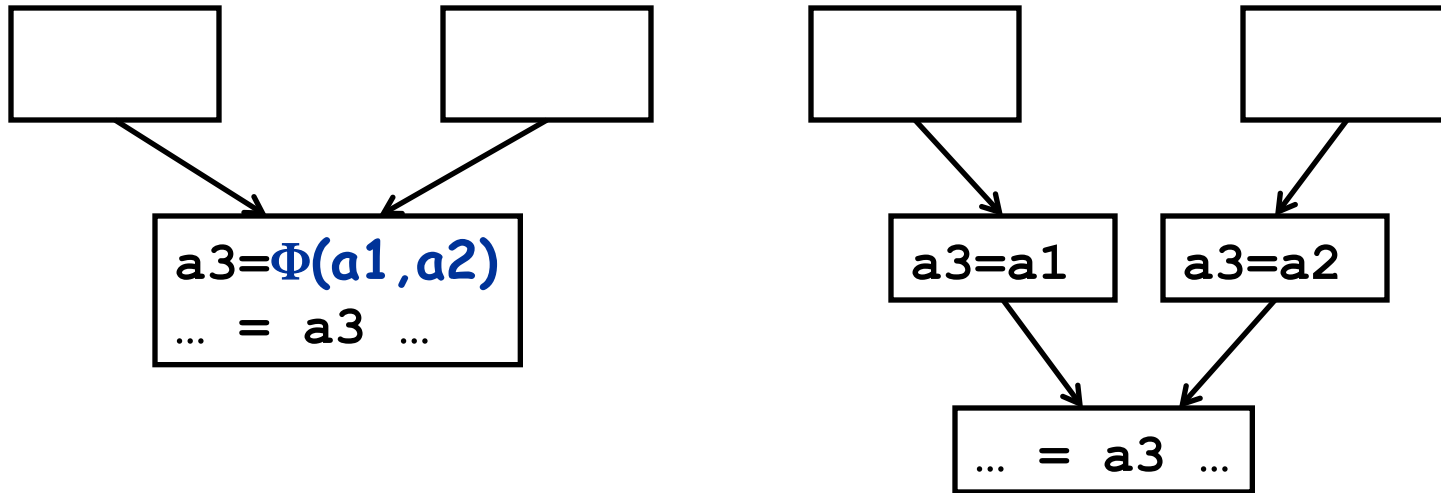
org	SSA	LVN	drop
a=x+y	a1=x1+y1	a1=x1+y1	a=x+y
b=x+y	b1=x1+y1	b1=a1	b=a
a=55	a2=55	a2=55	a=55
c=x+y	c1=x1+y1	c1=a1	c=a

DeSSA: get rid of Φ easy way



How to get rid of Φ the easy way?

DeSSA: get rid of Φ easy way?



Just case split by incoming edge

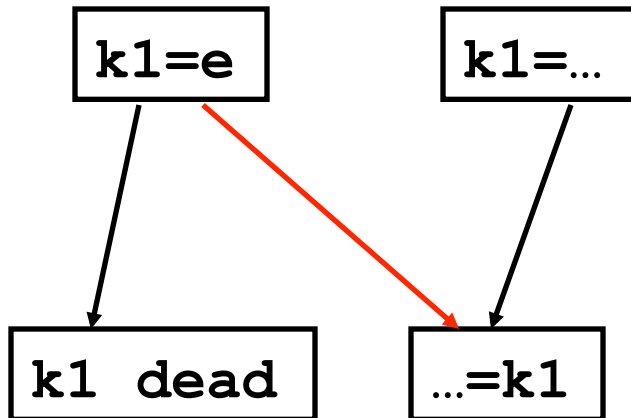
Then do copy propagation again

Register allocation merges var range

BUT: need parallel assignments!

Fancy: critical edges

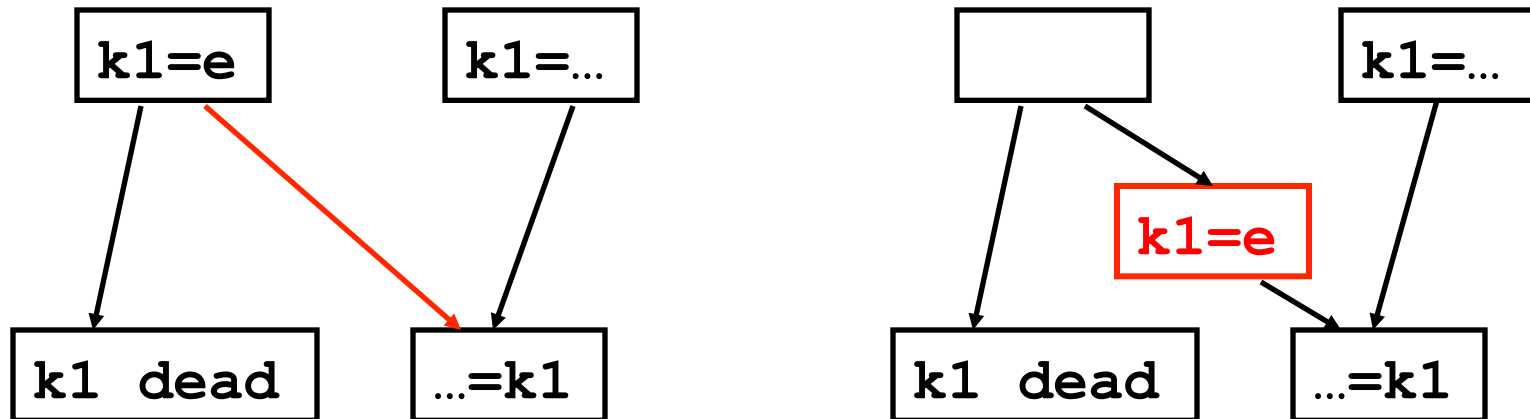
Critical edge in CFG where source has multiple successors and target multiple predecessors.



Critical edge makes optimal placement of $k1=e$ assignment impossible
 $k1=e$ assignment is unnecessary for left succ but incorrect for right

Fancy: critical edges

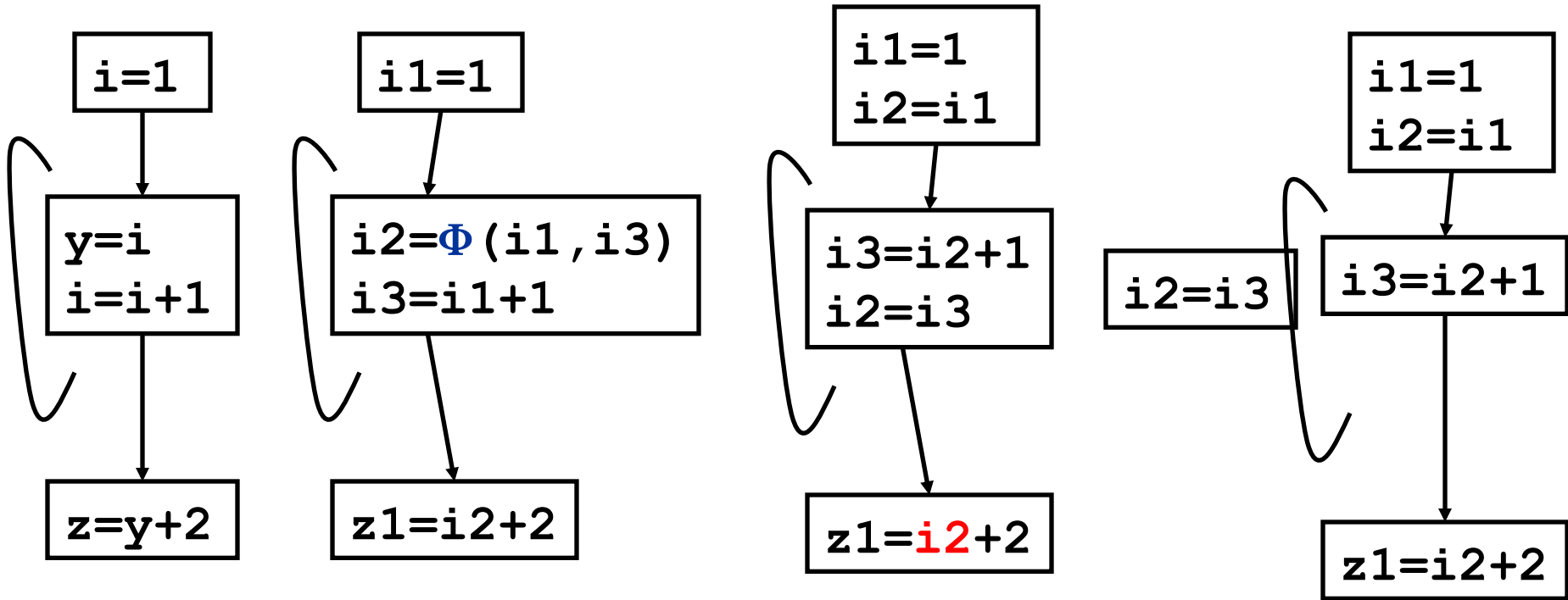
Critical edge in CFG where source has multiple successors and target multiple predecessors.



Critical edge makes optimal placement of $k1=e$ assignment impossible
 $k1=e$ assignment is unnecessary for left succ but incorrect for right

Solution: add block on critical edge (optimize: don't add elsewhere)

DeSSA: lost copies & critical split



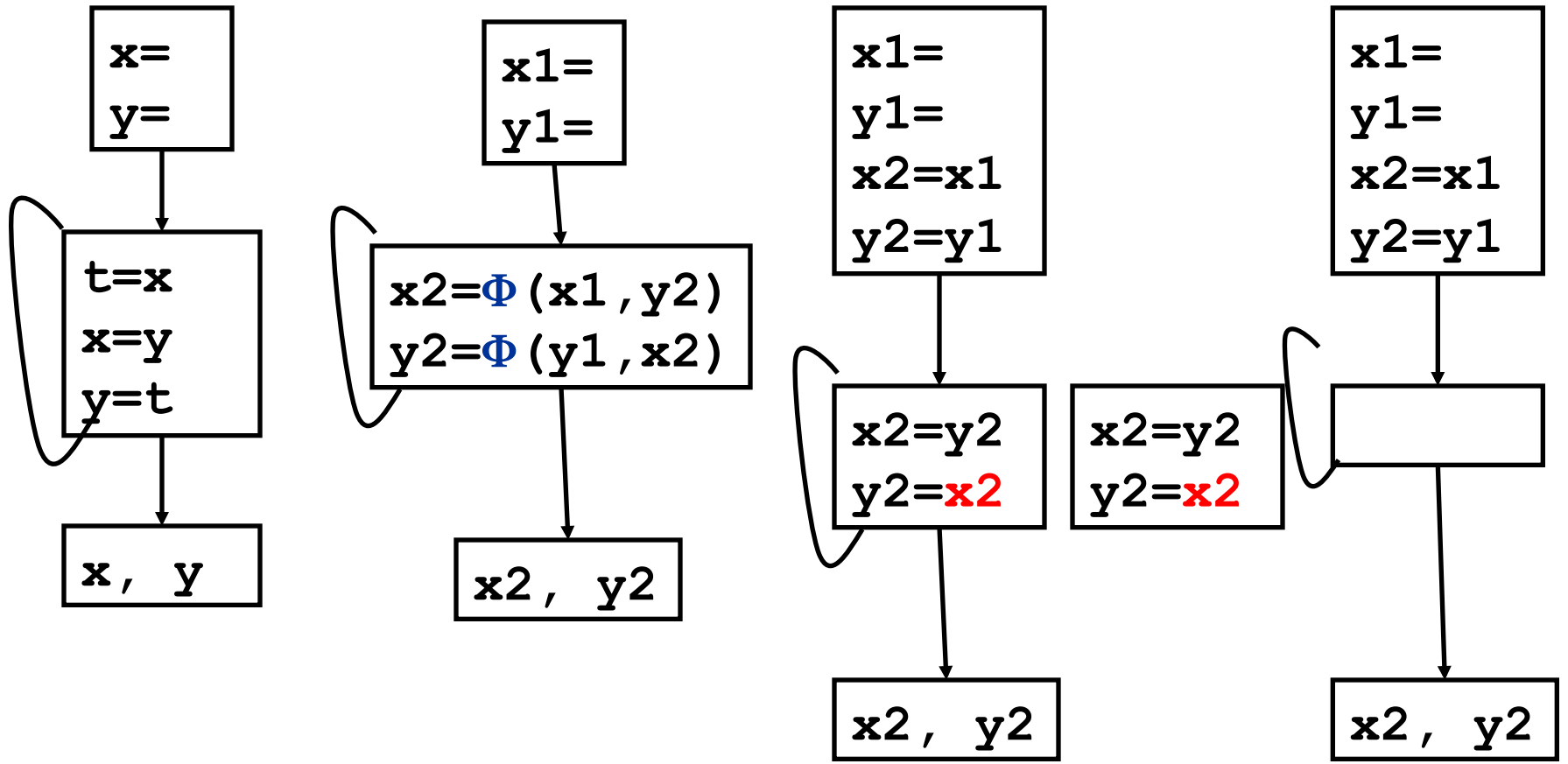
SSA©

Lost copy

split critical!

Or preserve a value that's still live in a temp

DeSSA: swap problem



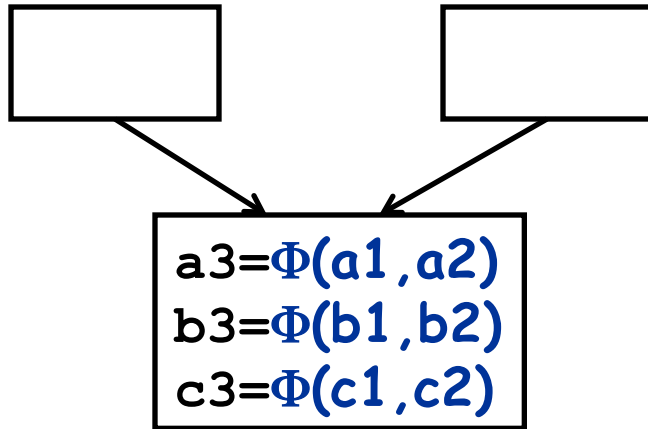
SSA©

overwritten

overwritten

use temps instead!

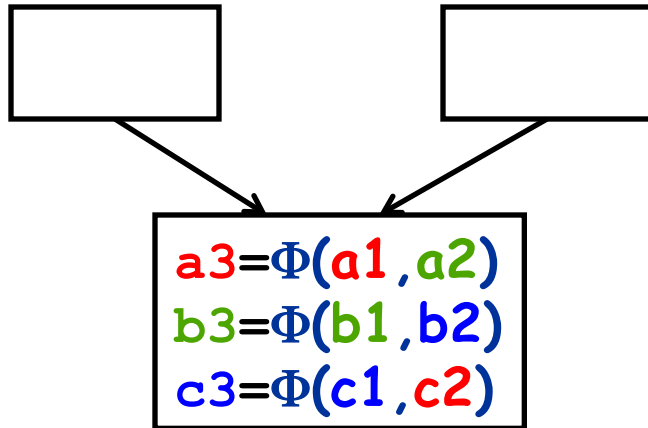
DeSSA: get rid of Φ elegant way



all Φ by **parallel** copy of argument i

$$a3, b3, c3 = a_i, b_i, c_i$$

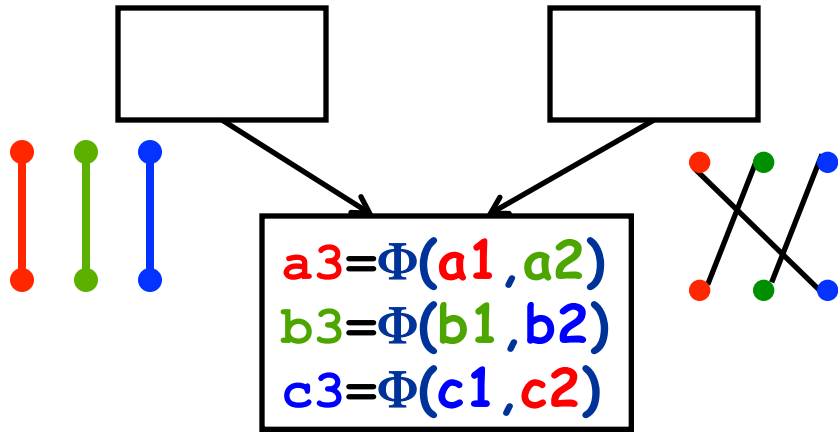
DeSSA: get rid of Φ fancy way



each edge i is permutation on regs

- implementable?

DeSSA: get rid of Φ fancy way



each edge i is permutation on registers

- implementable with 1 temp register
(\rightarrow impacts interference graph)
- implement by series of triple-xor swaps

3XOR swap

$$x = x \oplus y; \quad y = x \oplus y; \quad x = x \oplus y$$

$$x = X$$

$$y = Y$$

$$x = X \oplus Y$$

$$y = Y$$

$$y = X \oplus Y$$

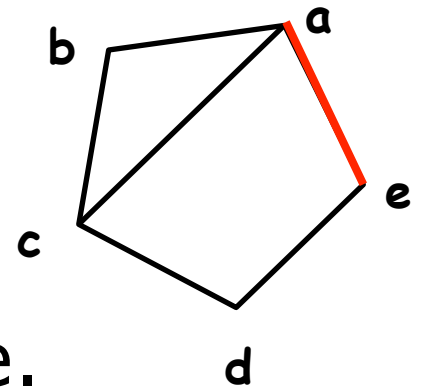
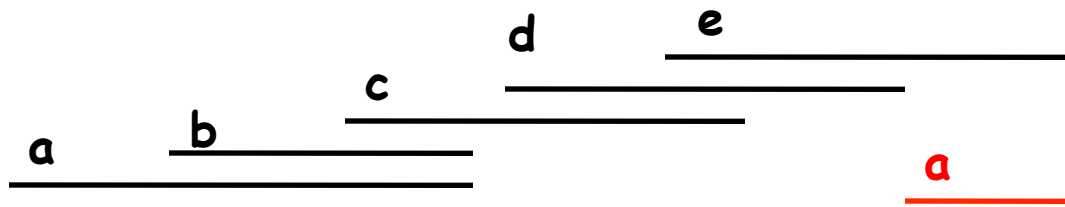
$$y = (X \oplus Y) \oplus Y = X \quad (a, c, i)$$

$$y = (X \oplus Y) \oplus X = Y \\ (a, c, i)$$

$$y = X$$

Register allocation on SSA in P

Register allocation is fast (polynomial) for programs with chordal interference graphs, e.g., SSA.



No edge (a,e) can lead to a cycle. That would require a to be live again. This violates SSA single assignment. Parallel Φ less interference than temps

Interference graphs from SSA

Interference graph \leq dominance tree

x, y interfere since both live at c

→ $x \geq c$ and $y \geq c$

→ Dominance tree $x \geq y$ or $y \geq x$

→ x live at y or y live at x (dom lives)

→ vars that interfere with x dominate x

SSA construction: easy way

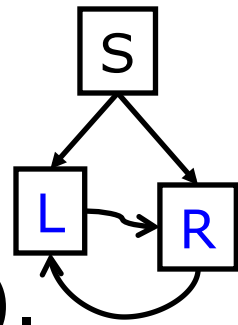
Translate into full SSA, i.e., Φ at every merge of control flow by syntax-directed translation (see IR-trees).
Hint: use block-id i as index i .

Repeatedly optimize

$x_i := \Phi(x_i, \dots, x_i) \rightarrow \text{skip}$

$x_i := \Phi(x_i, x_j, x_i, \dots, x_j) \rightarrow x_i := x_j$

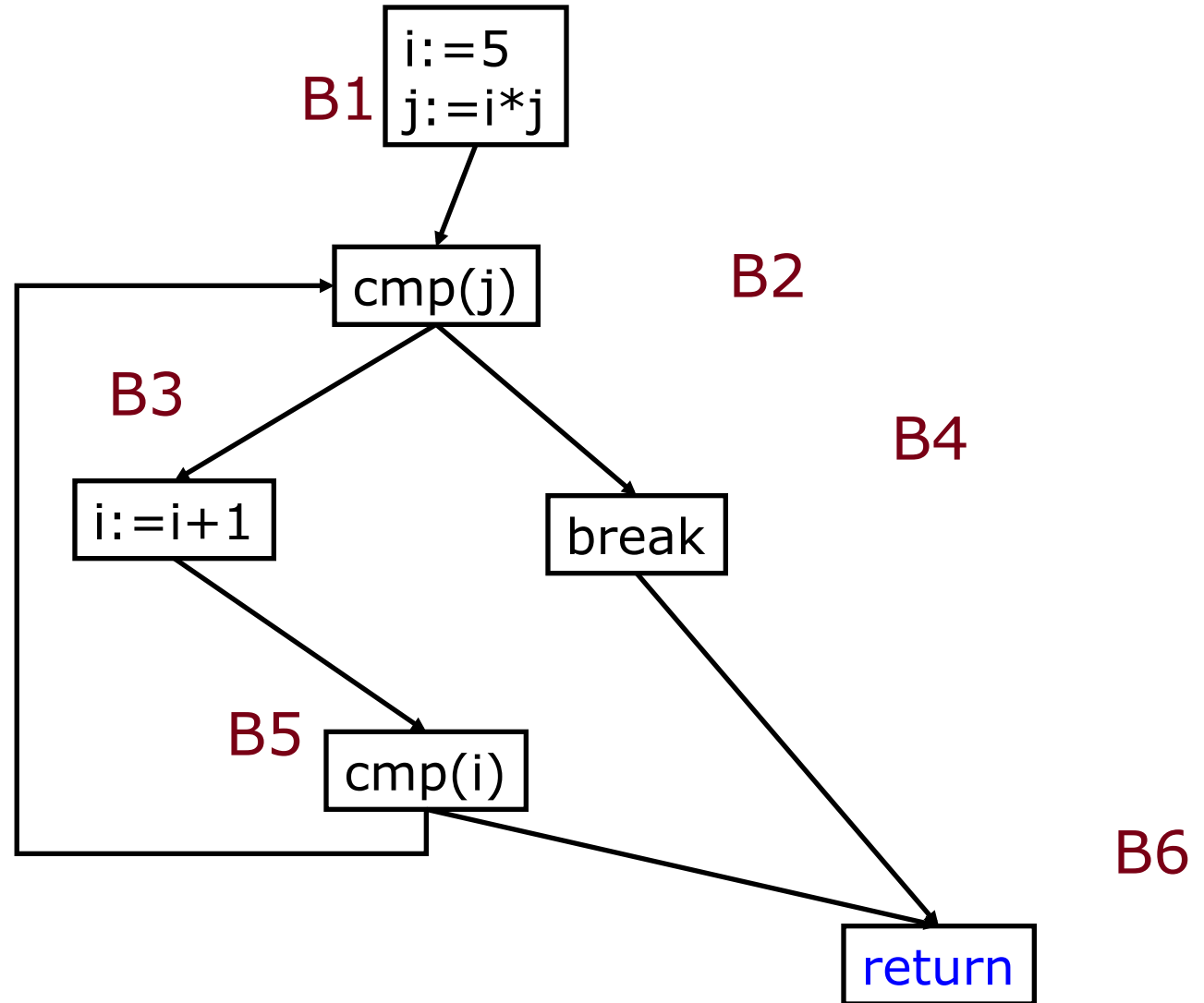
Theorem: Correct SSA. Minimal if reducible (loops no multiple entries).



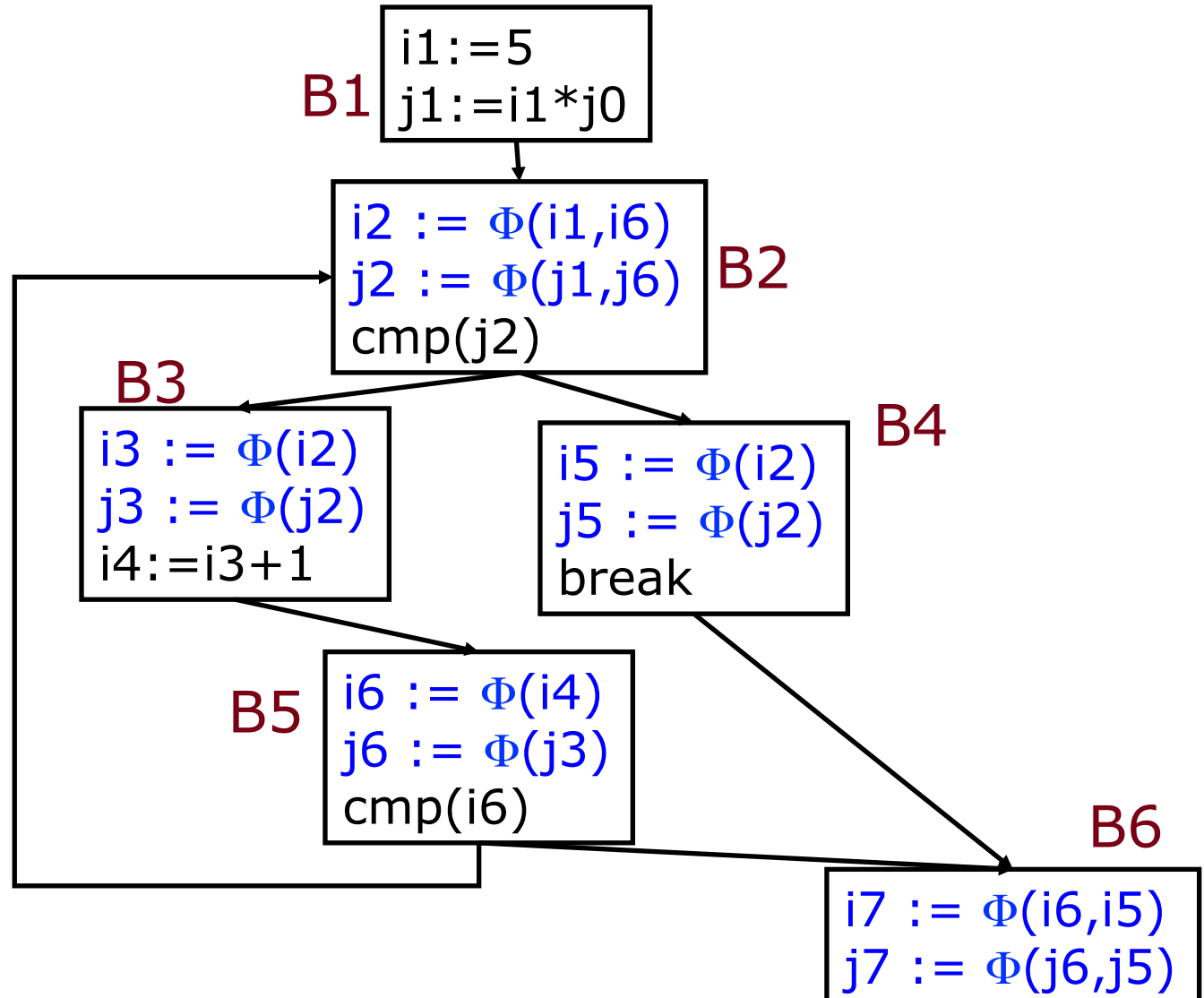
SSA construction: easy way

```
i=5
j=i*j
do {
  use j;
  if (j > 5)
    i = i + 1;
  else
    break;
} while (i < 50);
```

SSA construction: easy way



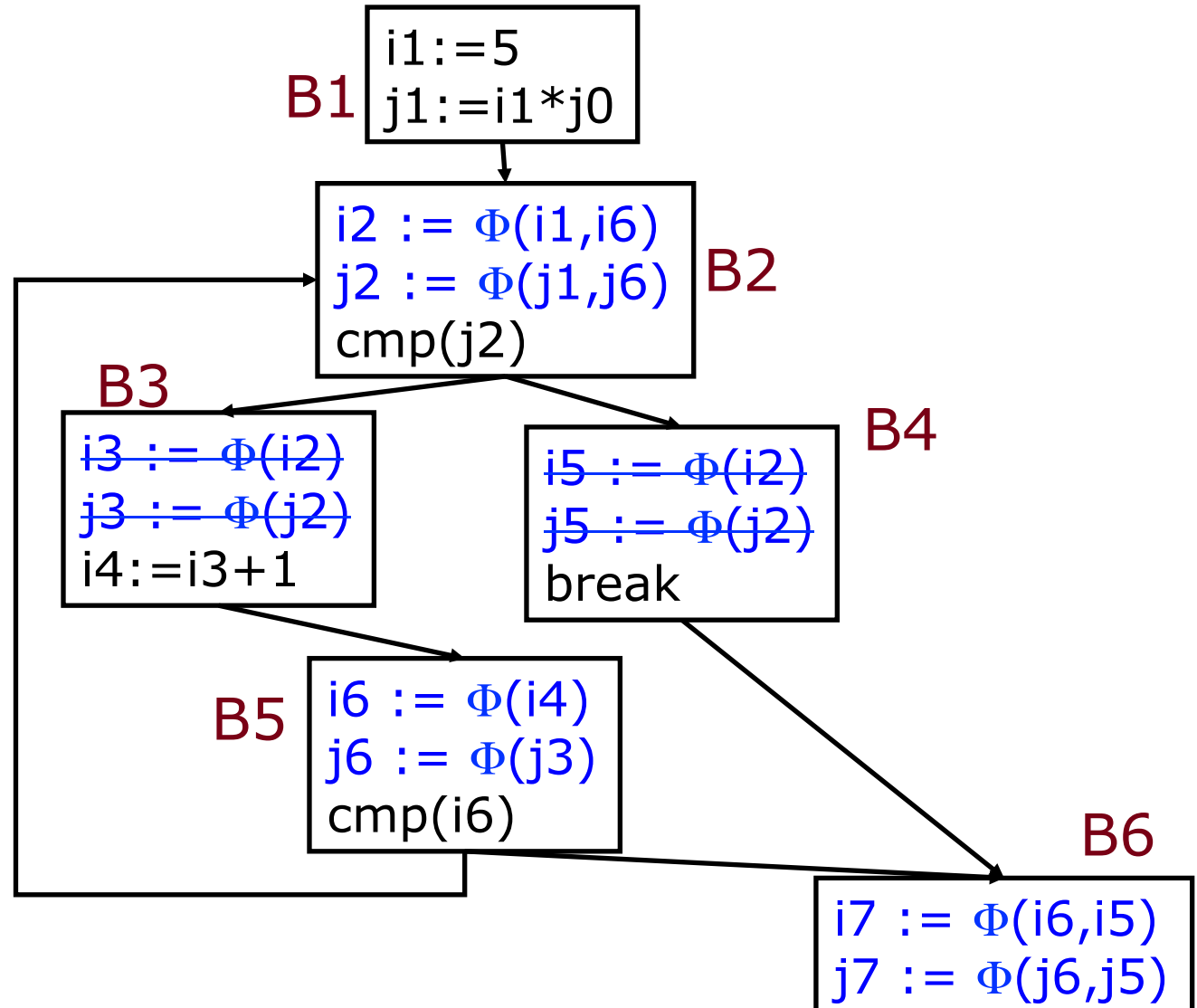
SSA construction: easy full Φ



SSA construction: easy full Φ

$i_3 = i_2$
 $i_5 = i_2$

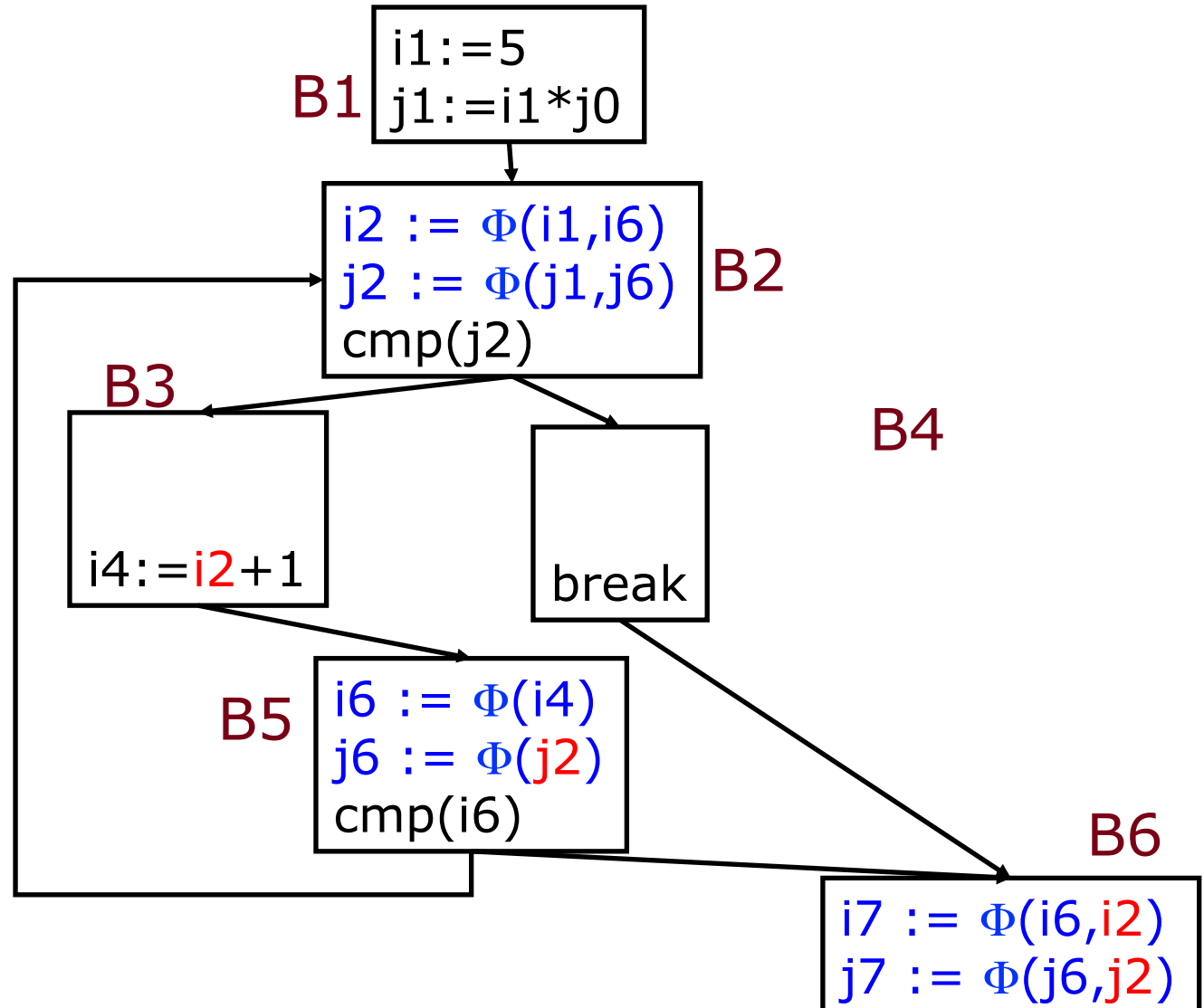
$j_3 = j_2$
 $j_5 = j_2$



SSA construction: $x_i := \Phi(x_i, x_i)$

$i_3 = i_2$
 $i_5 = i_2$

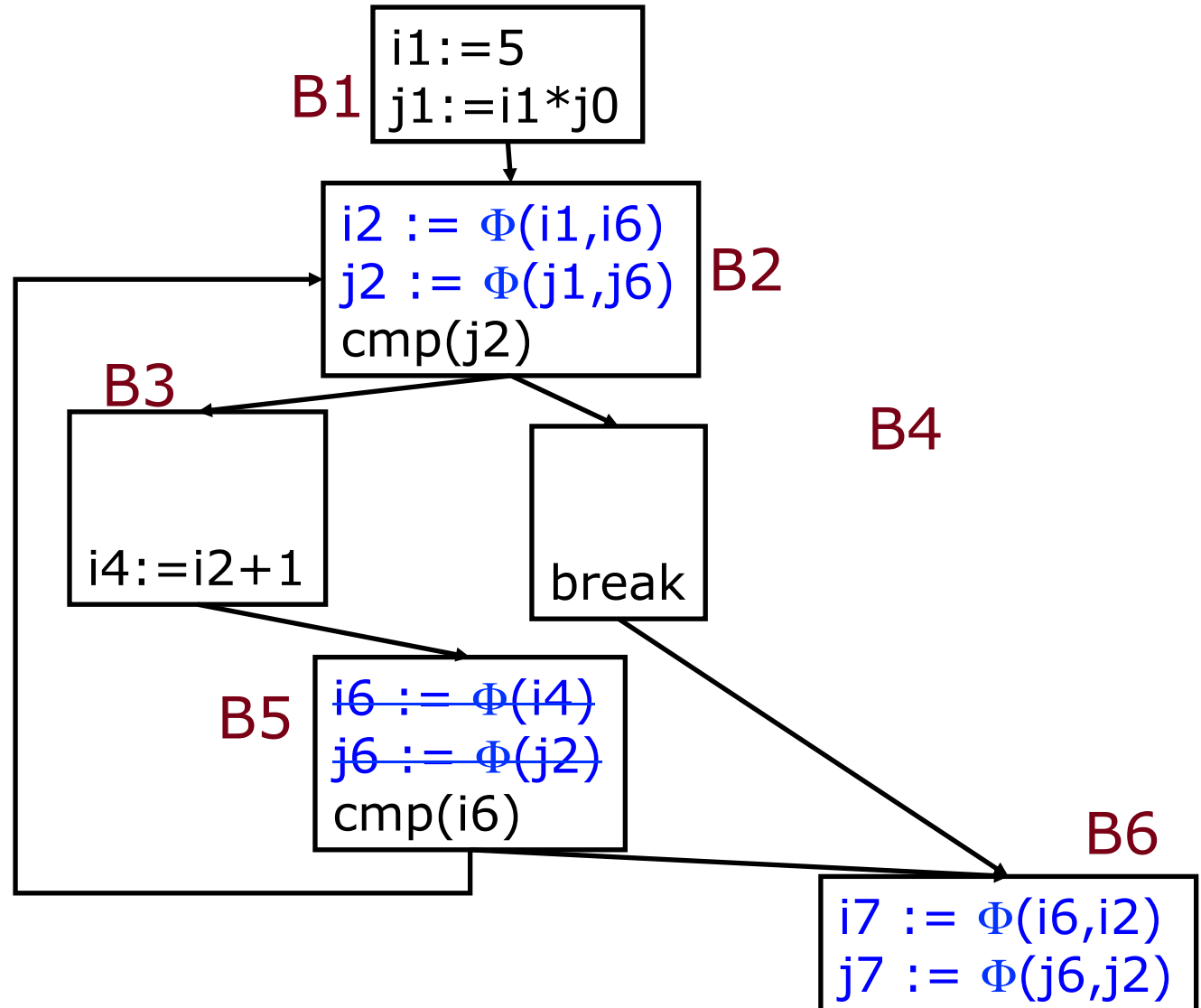
$j_3 = j_2$
 $j_5 = j_2$



SSA construction: $x_i := \Phi(x_i, x_i)$

$i_3 = i_2$
 $i_5 = i_2$
 $i_6 = i_4$

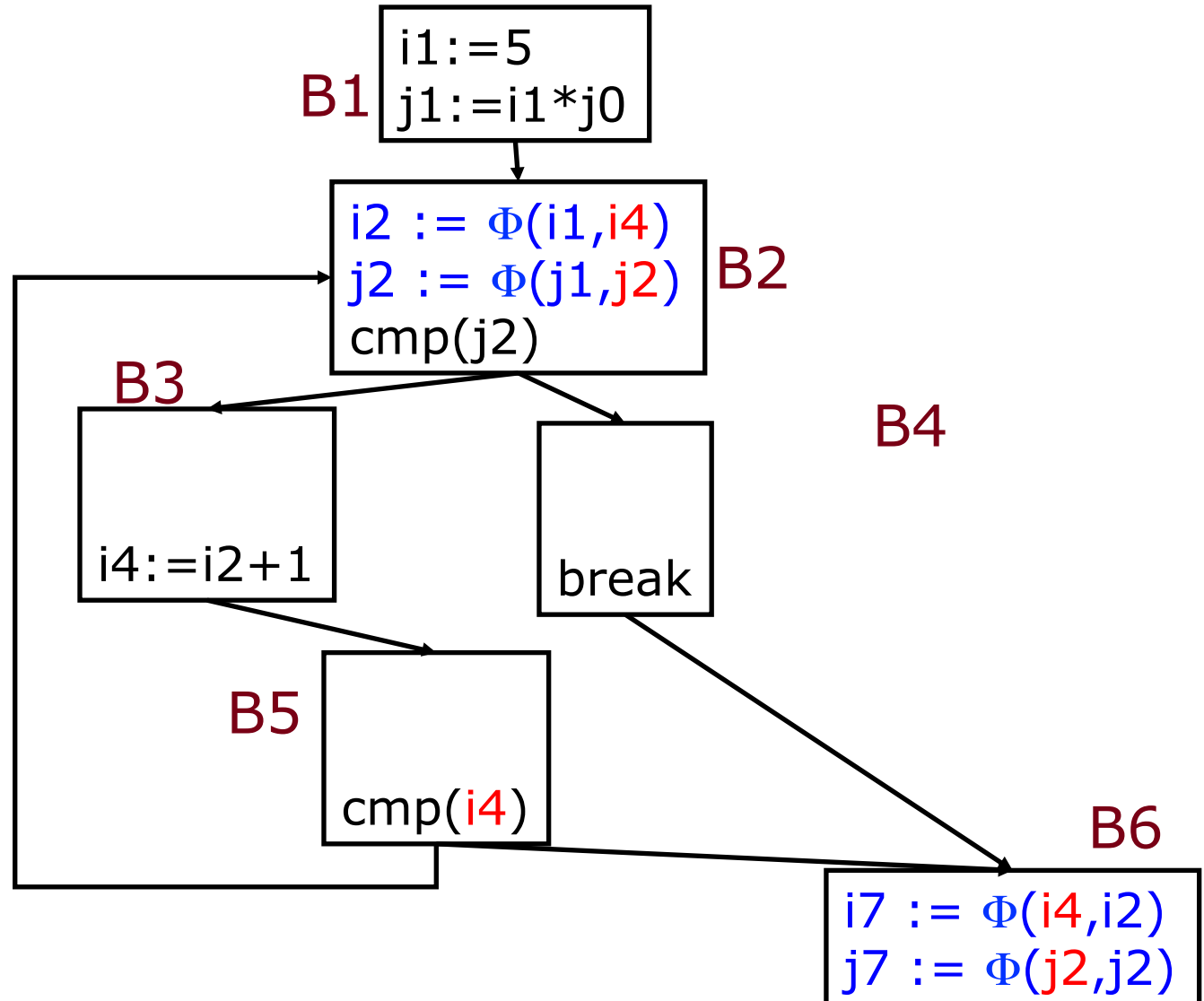
$j_3 = j_2$
 $j_5 = j_2$
 $j_6 = j_2$



SSA construction: $x_i := \Phi(x_i, x_i)$

$i_3 = i_2$
 $i_5 = i_2$
 $i_6 = i_4$

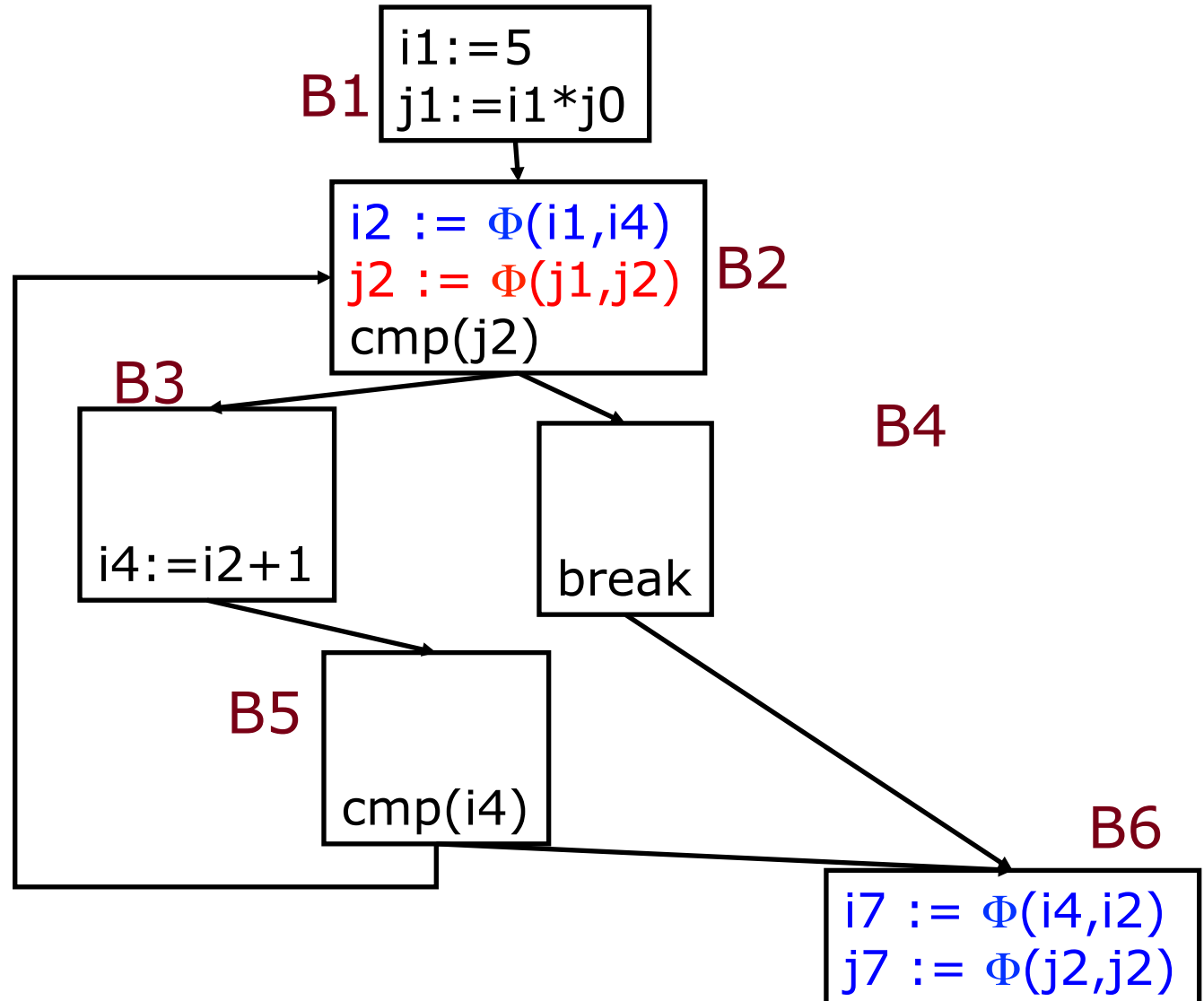
$j_3 = j_2$
 $j_5 = j_2$
 $j_6 = j_2$



SSA construction: $x_i := \Phi(x_i, x_j)$

$i_3 = i_2$
 $i_5 = i_2$
 $i_6 = i_4$

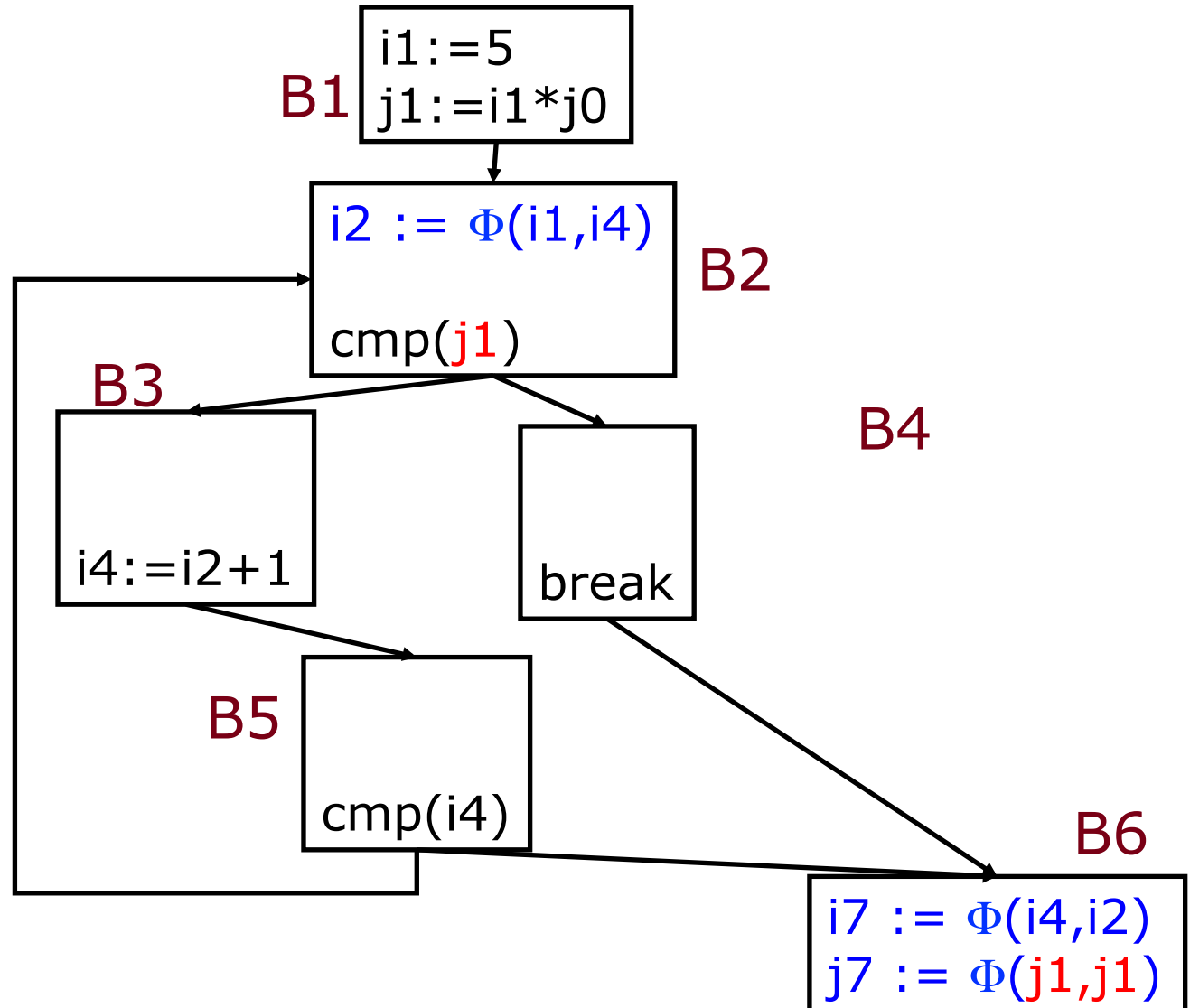
$j_3 = j_2$
 $j_5 = j_2$
 $j_6 = j_2$
 $j_2 = j_1$



SSA construction: $x_i := \Phi(x_i, x_j)$

$i_3 = i_2$
 $i_5 = i_2$
 $i_6 = i_4$

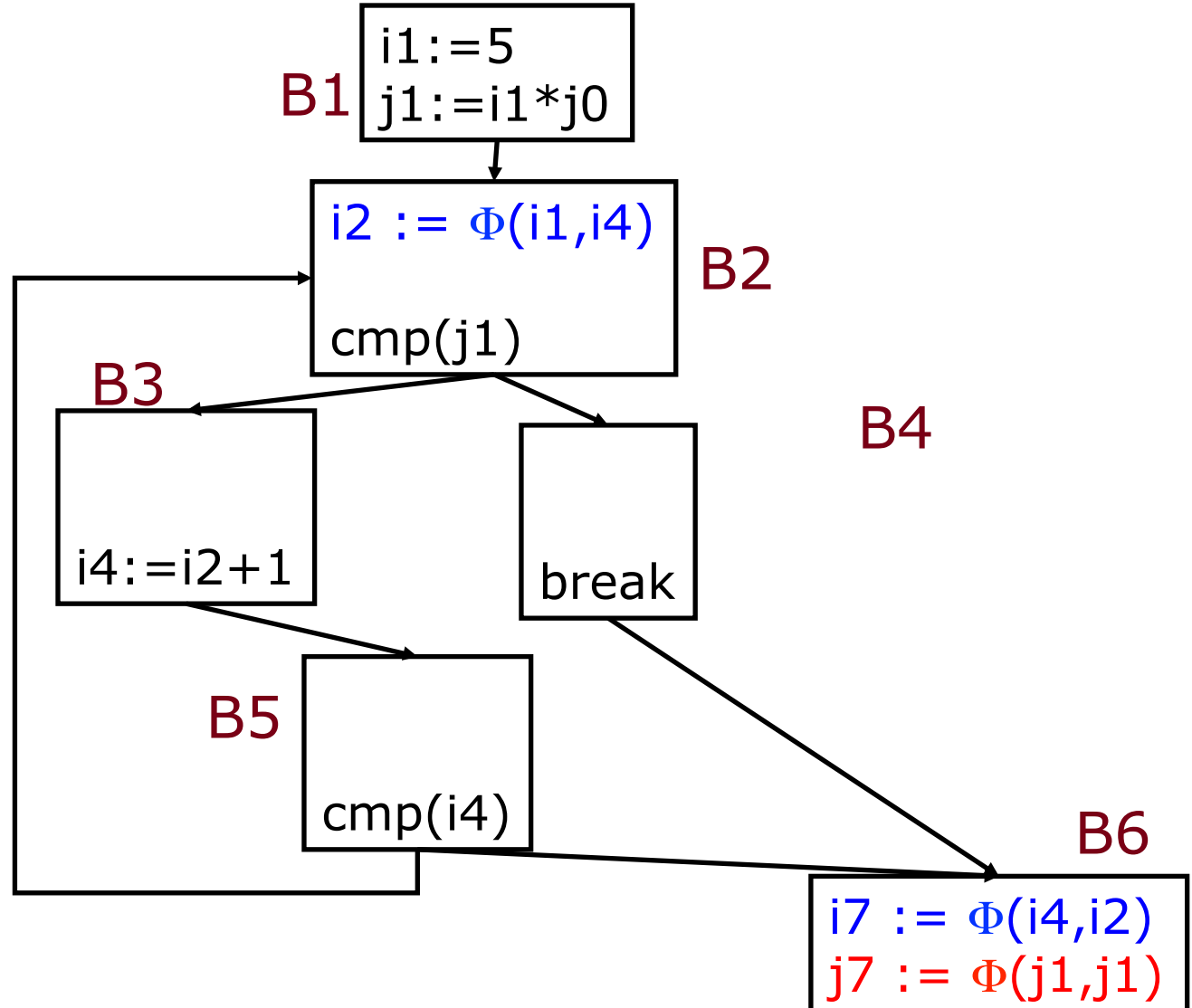
$j_3 = j_2$
 $j_5 = j_2$
 $j_6 = j_2$
 $j_2 = j_1$



SSA construction: $x_i := \Phi(x_i, x_j)$

$i_3 = i_2$
 $i_5 = i_2$
 $i_6 = i_4$

$j_3 = j_2$
 $j_5 = j_2$
 $j_6 = j_2$
 $j_2 = j_1$

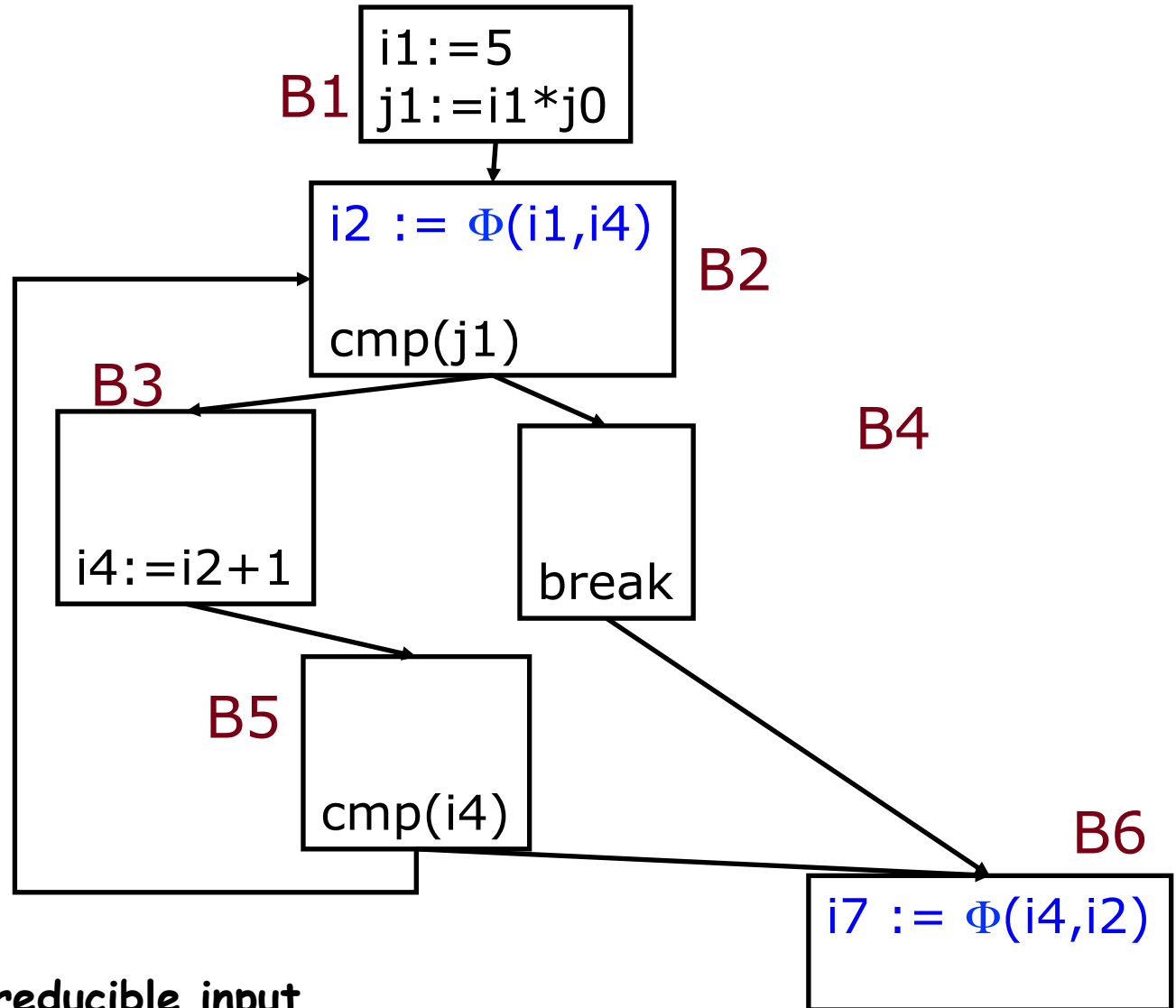


SSA construction: $x_i := \Phi(x_i, x_j)$

$i_3 = i_2$
 $i_5 = i_2$
 $i_6 = i_4$

$j_3 = j_2$
 $j_5 = j_2$
 $j_6 = j_2$
 $j_2 = j_1$
 $j_7 = j_1$

union find for
representatives of
equivalence classes



Minimal SSA since reducible input

Project advice

The bottom line for your project:

- You don't need to generate SSA form for your project
- However, if you decide to do this, then it is advisable to simplify matters by generating SSA directly during AST translation, not working with DF

Summary

- SSA has had a huge impact on compiler design
- Most modern production compilers use SSA (including gcc, suif, llvm, hotspot, ...)
- Compiler frameworks (i.e., toolkits for creating compilers) all use SSA

Quiz (1)

1. Compare following constructions. What are the benefits? What's the explanation of one in terms of the others? What's the relation of Φ' and $\Phi(t,t,t)$?
 1. Max SSA
 2. dominance frontier SSA
 3. AST walk SSA

Quiz (2)

1. Which optimizations can SSA perform that local value numbering cannot?
2. Which disadvantages does SSA have? Can SSA make your compiler worse?
3. What problems in register allocation does SSA solve? Which does it cause?

Quiz (3)

1. Why do dominance frontiers have to be iterated?
2. Why do dominance frontiers work on a set of blocks?
3. Is 1 temp enough to implement Φ ?

Some References

John Aycock and Nigel R. Horspool.
[Simple Generation of Static Single-Assignment Form](#). CC,
LNCS 1781, Springer, 2000.

Mark N. Wegman and Kenneth F. Zadeck,
[Constant propagation with conditional branches](#), ACM
TOPLAS, 13(2), 1991.

Bowen Alpern, Mark N. Wegman, and Kenneth F. Zadeck.
[Detecting Equality of Variables in Programs](#), POPL 1988.

Oliver Rüthing, Jens Knoop, and Bernhard Steffen.
[Detecting Equalities of Variables: Combining Efficiency
with Precision](#), SAS, LNCS 1694, Springer, 1999.

Keith D. Cooper, Timothy J. Harvey, and Ken Kennedy.
[A Simple, Fast Dominance Algorithm](#). SPE, 4, 2001.