Static Single Assignment

15-411 Compiler Design

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 (ie, the same *value number) to expressions e1 and e2 if they are congruent*

If key “t2+1” is not in the table, then put it in with value “t1”

- next time we hit on “t2+1”, can replace it in the IR with “t1”

Example

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

i=v1

Hash(v1+1)->j

j=v2

k=v1

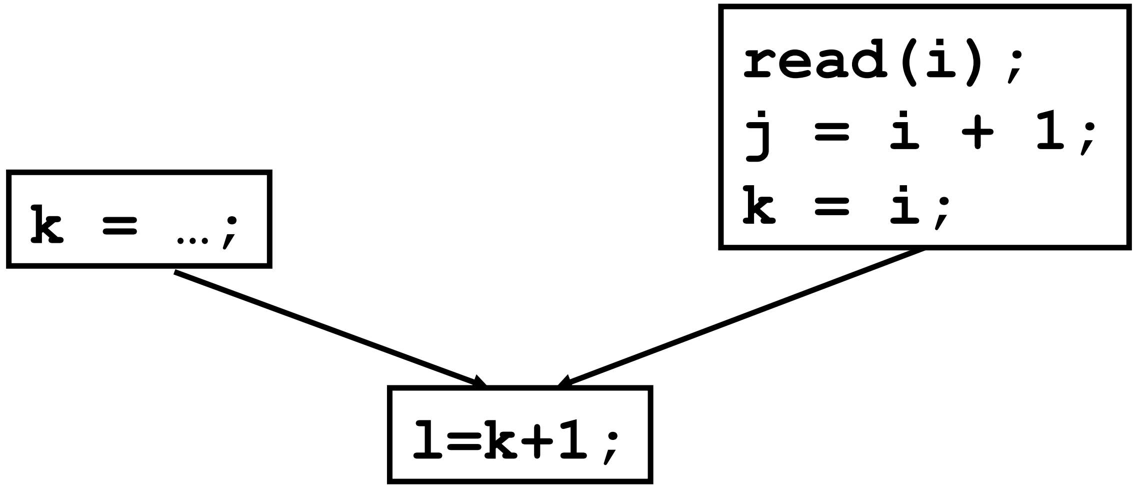
Hash(v1+1)->j

Therefore l=j

Global value numbering?

Local value numbering (within a basic block) is easy

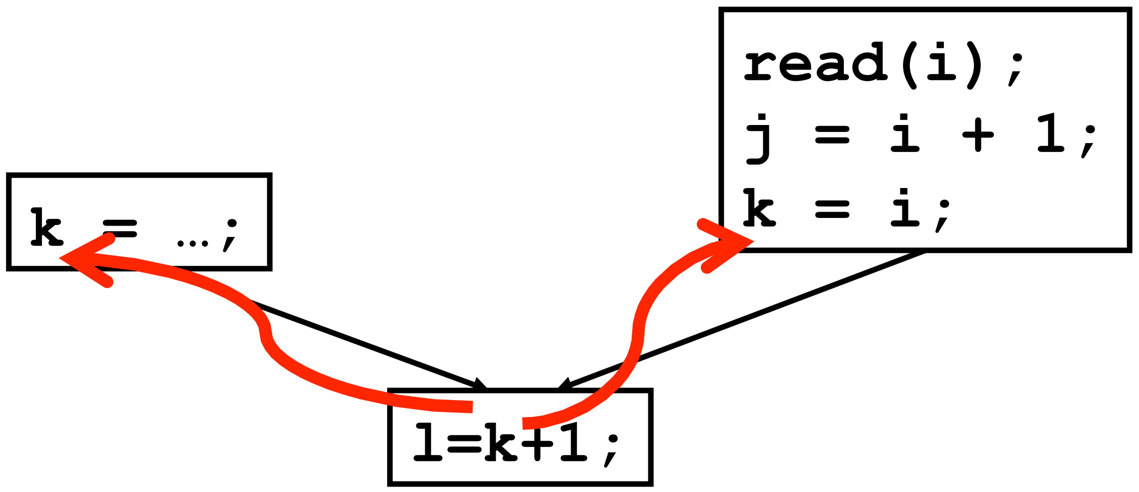
Global value numbering (within a procedure)?



Importance of use-def

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

Could do dataflow analysis for global value numbering

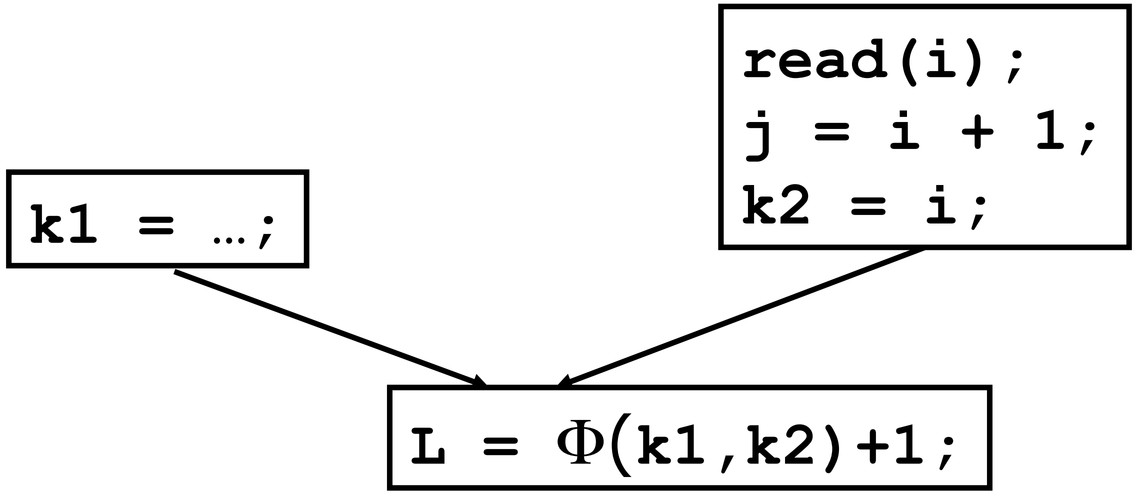


use-def analysis

Embed use-def into IR

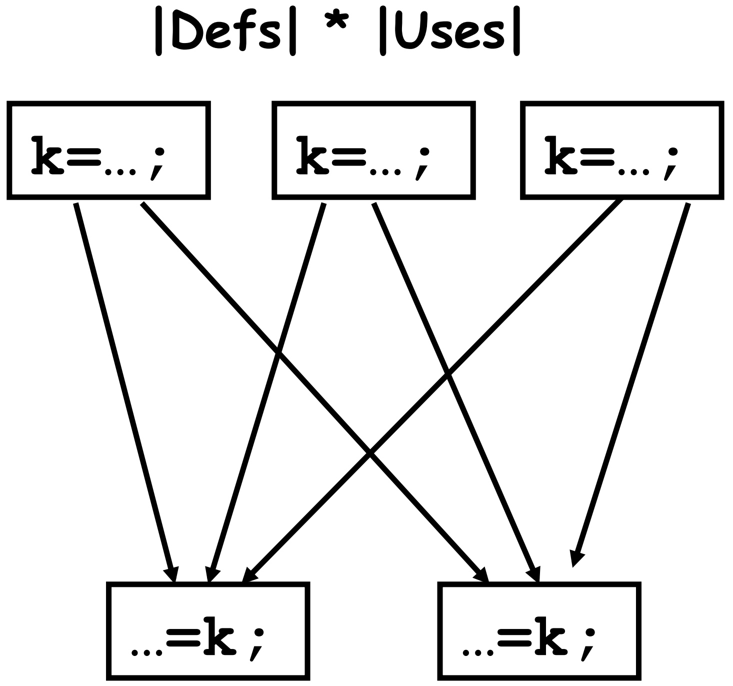
Use-def information is central to many optimizations

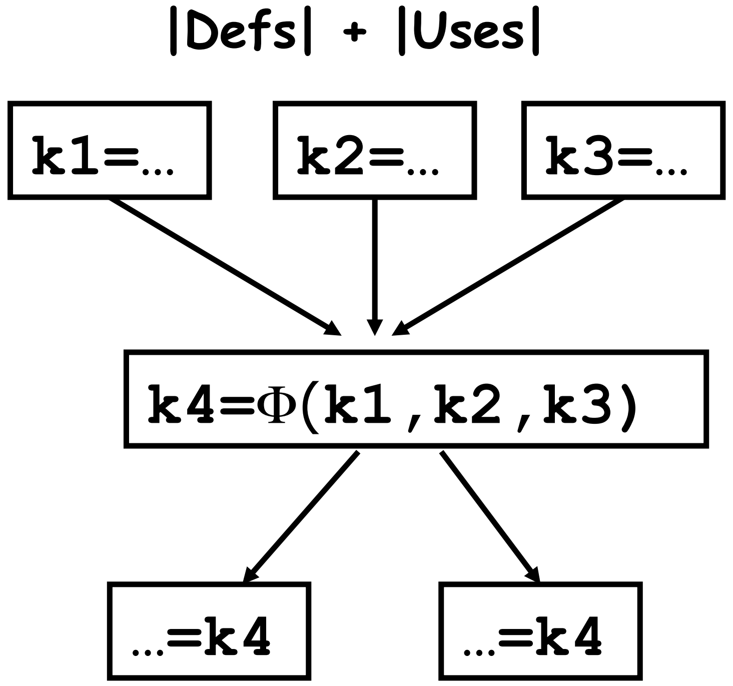
The point of static single assignment (SSA) is to represent Use-def information explicitly



SSA use-def

SSA reduces the representational complexity of use-defs





Local vs global

Many optimizations can be performed both locally and globally

- local: within a basic black
- 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

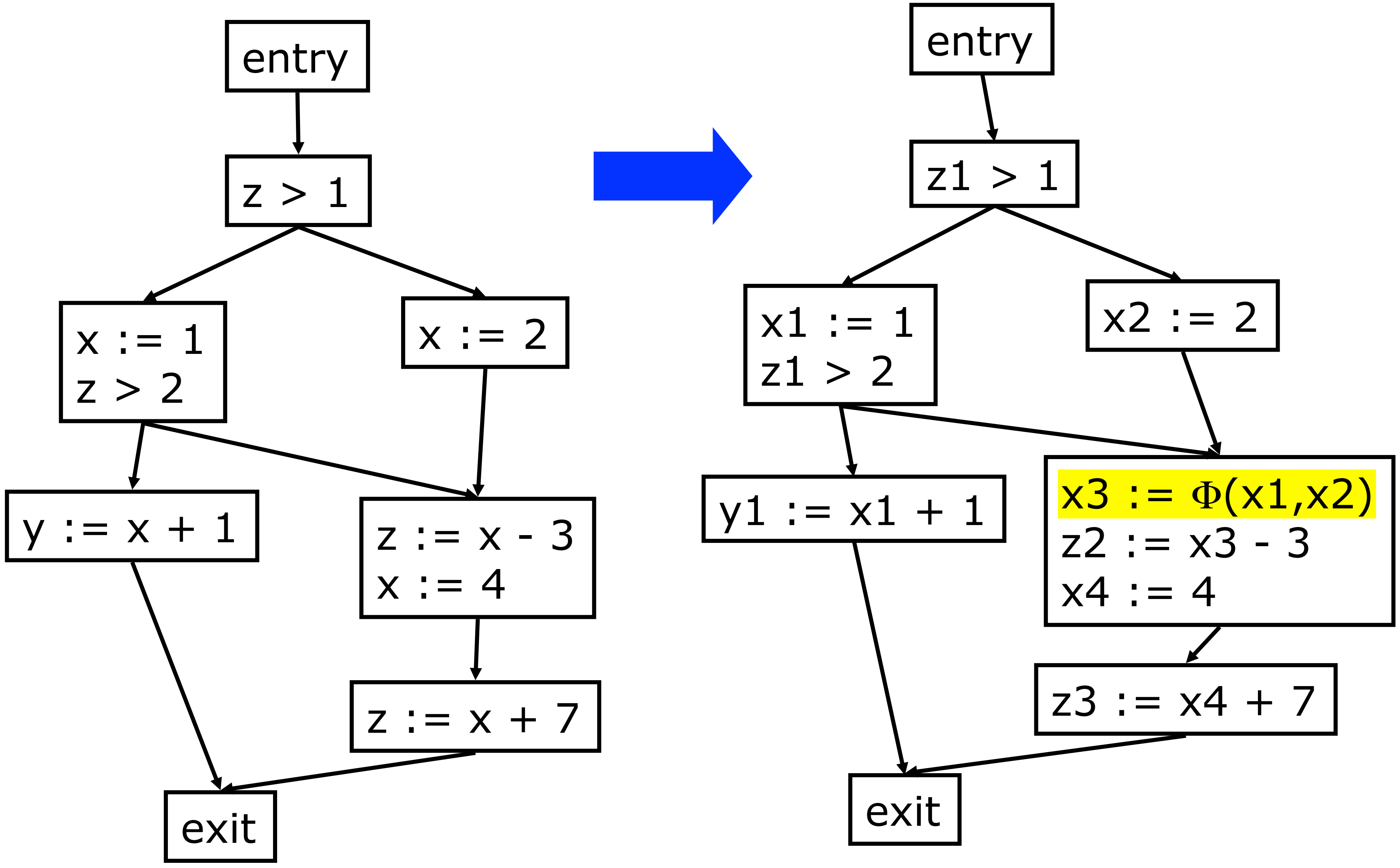
Why use SSA?

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

Several important optimizations trivial on SSA: redundancy eliminations like

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

Example


entry
z > 1
x := 1
z > 2
x := 2
y := x + 1
z := x - 3
x := 4
z := x + 7
exit
entry
z1 > 1
x1 := 1
z1 > 2
x2 := 2
y1 := x1 + 1
x3 := Φ(x1,x2)
z2 := x3 - 3
x4 := 4
z3 := x4 + 7
exit

Creating SSA form

Φ pseudofunctions select definition from actual control flow

To translate into SSA form:

- Insert trivial Φ functions
  - Φ(t,t,…,t), where the number of t’s is the number of incoming flow edges
- Globally analyze and rename definitions and uses of variables to establish SSA property

After optimizations, we discard Φ functions and replace by x3:=x2 on new blocks for back-edges. Then copy propagate & reg alloc

Minimal SSA form

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

Definitions:

- In a flowgraph, node *a* dominates node *b* ("*a dom b*") if every possible execution path from node *entry* to *b* includes *a*
- If *a* and *b* are different nodes, we say that *a strictly dominates b* ("*a sdom b*")
- If *a sdom b*, and there is no *c* such that *a sdom c* and *c sdom b*, we say that *a* is *the immediate dominator* of *b* ("*a idom b*" or, since *a* is unique: "*idom(b)=a*")

Dominance frontier

For a node *a*, the dominance frontier of *a*, DF[*a*], is the set of all nodes *b* such that *a* strictly dominates an immediate precedessor of *b* but not *b* itself. At DF we need some Φ functions

DF[*a*] is border between dom / undom

More formally:

- DF[*a*] = {*b* | (∃*c*∈Pred(*b*) such that *a dom c* but not *a sdom b*}

Computing DF[a]

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

- $DF[a] = \{b \mid (\exists c \in Pred(b)$ such that $a \text{ sdom } c$ but not $a \text{ sdom } b\}$

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

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

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:

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

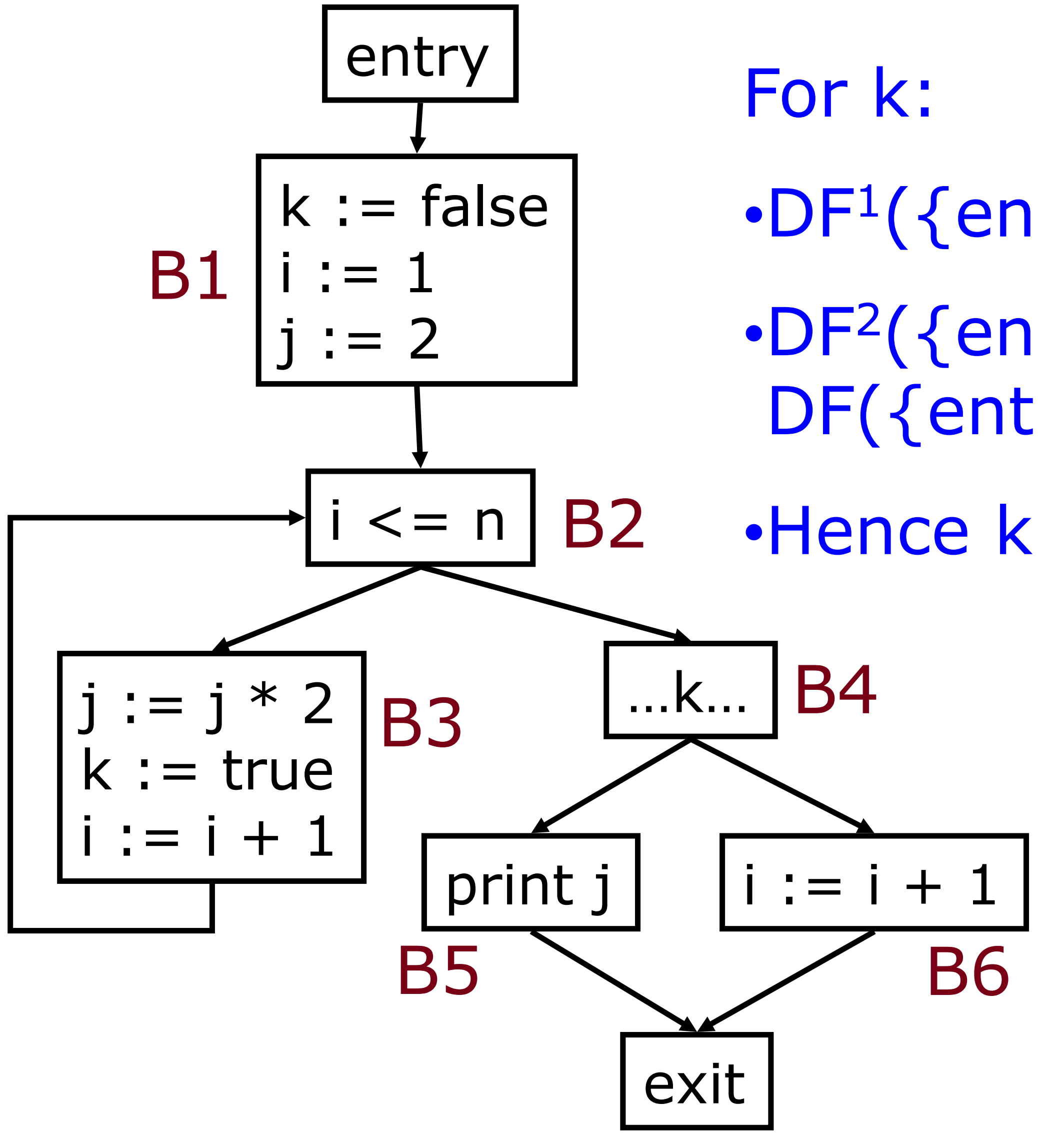
Iterated DF

Then, the iterated dominance frontier is defined as follows:

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

If S is the set of nodes that assign to variable k, then $DF^{+}[S \cup \{\mathit{entry}\}]$ is the set of nodes that need Φ functions for k

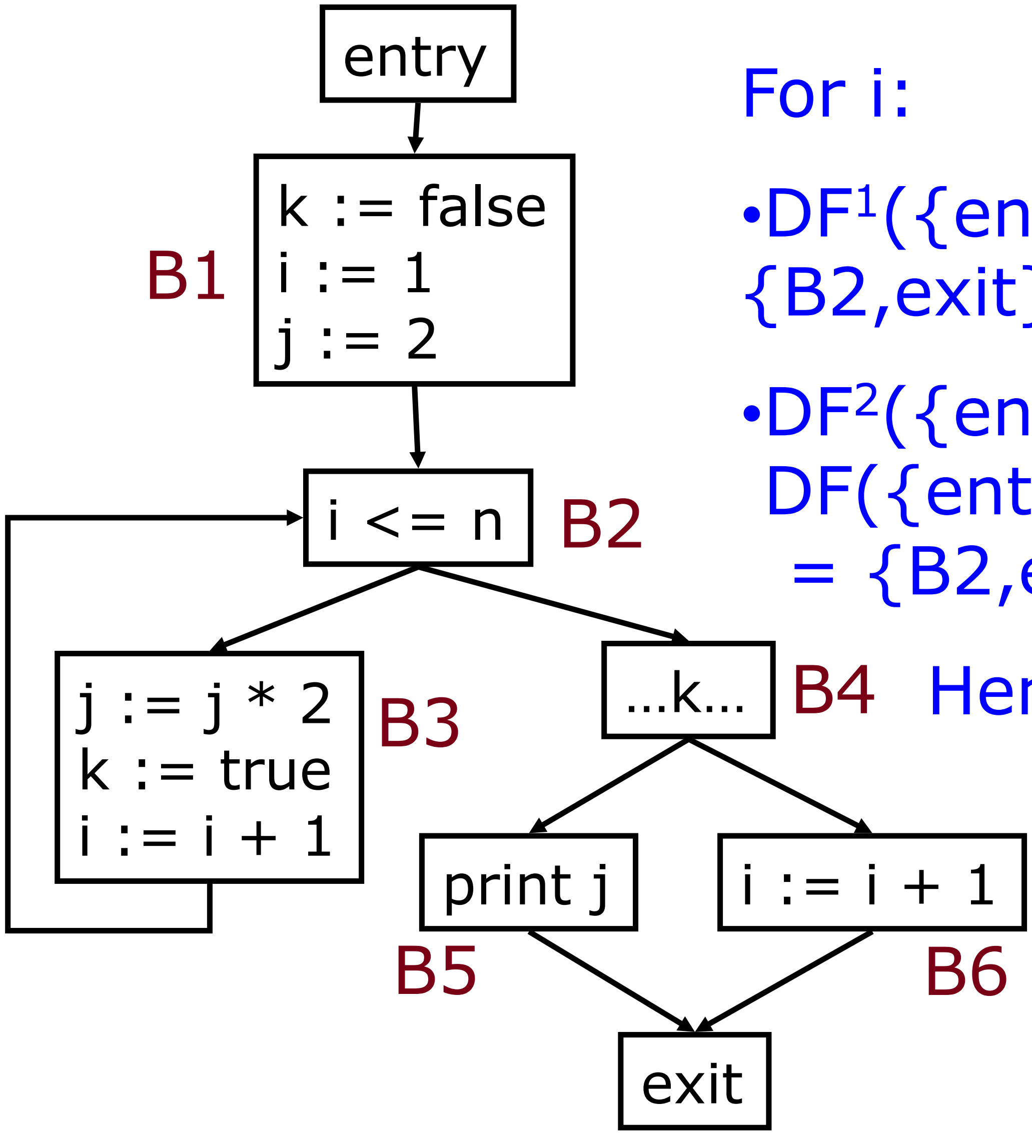
Example



For k:

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

Example

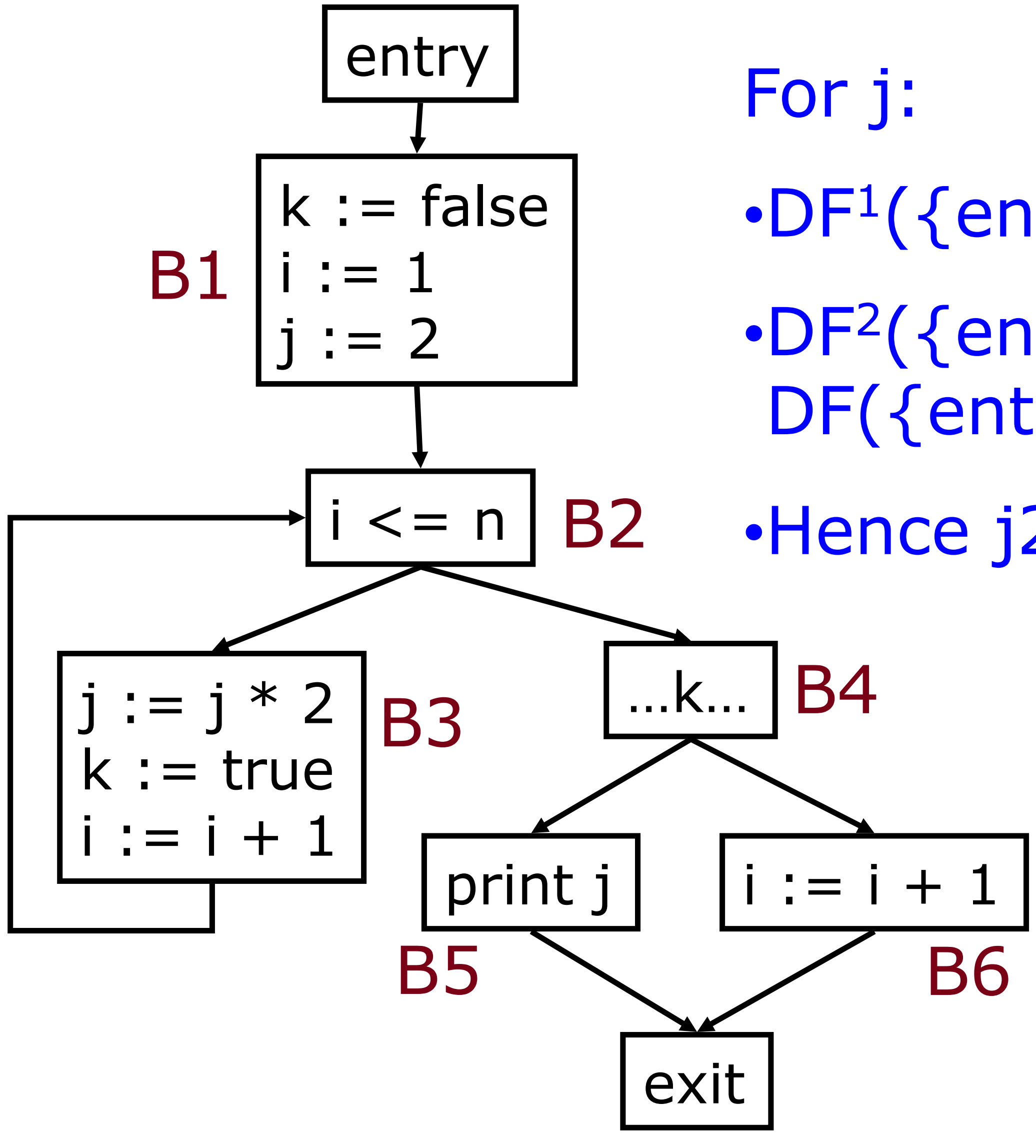


For i:

- $DF^1(\{entry,B1,B3,B6\}) = \{B2,exit\}$
- $DF^2(\{entry,B1,B3,B6\}) = DF(\{entry,B1,B2,B3,B6,exit\}) = \{B2,exit\}$

Hence ij:=Φ(…) at B2,exit

Example



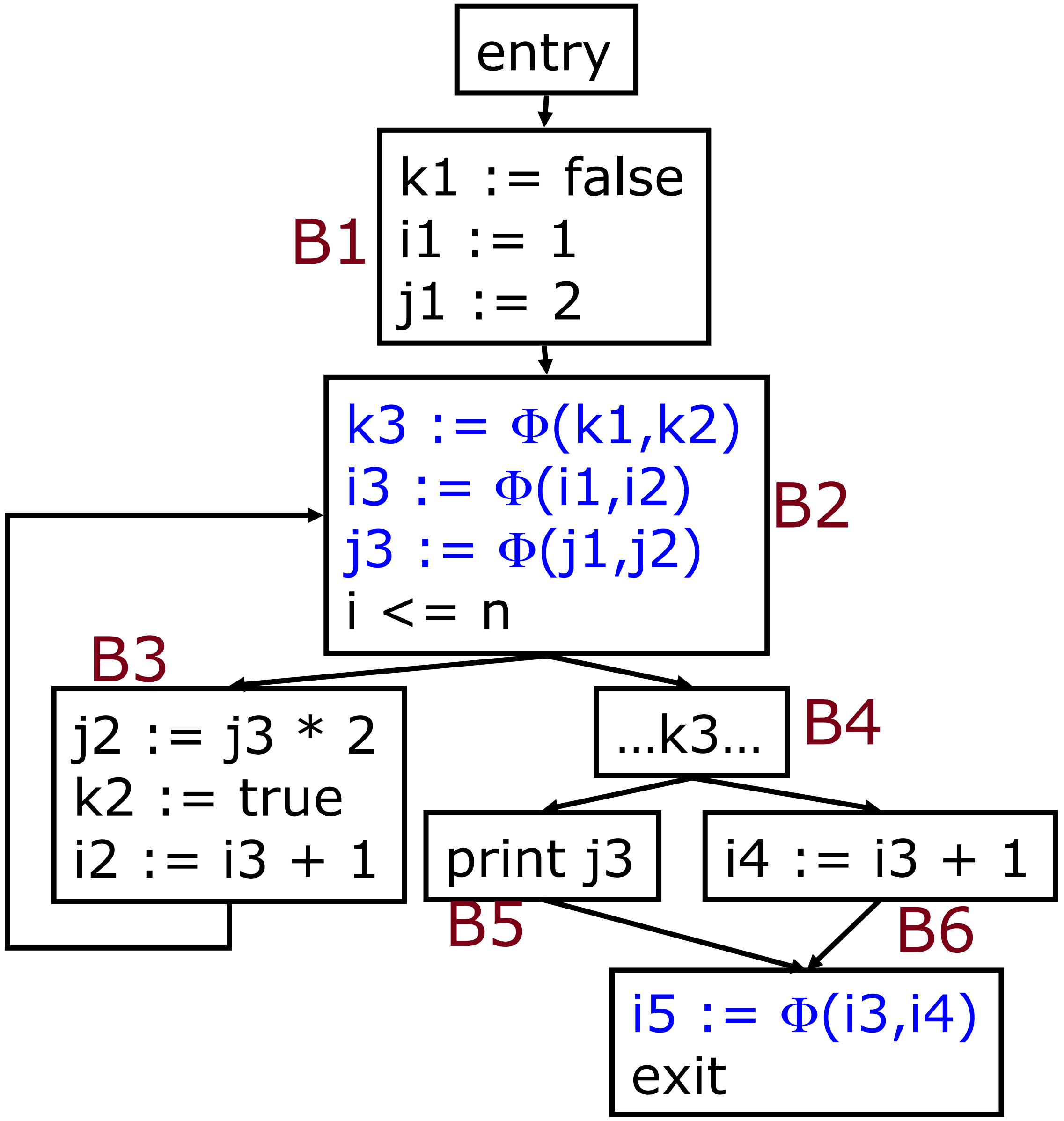
For j:

- $DF^1(\{entry,B1,B3\}) = \{B2\}$
- $DF^2(\{entry,B1,B3\}) = DF(\{entry,B1,B2,B3\}) = \{B2\}$
- 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 too complicated to implement.

SSA by AST value numbering

Walk AST to assign value numbers

- 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.
  $a2 := \Phi(a1,a2) \rightarrow a2 := a1$ $a1 := \Phi(a1) \rightarrow \varepsilon$

SSA by value number AST

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

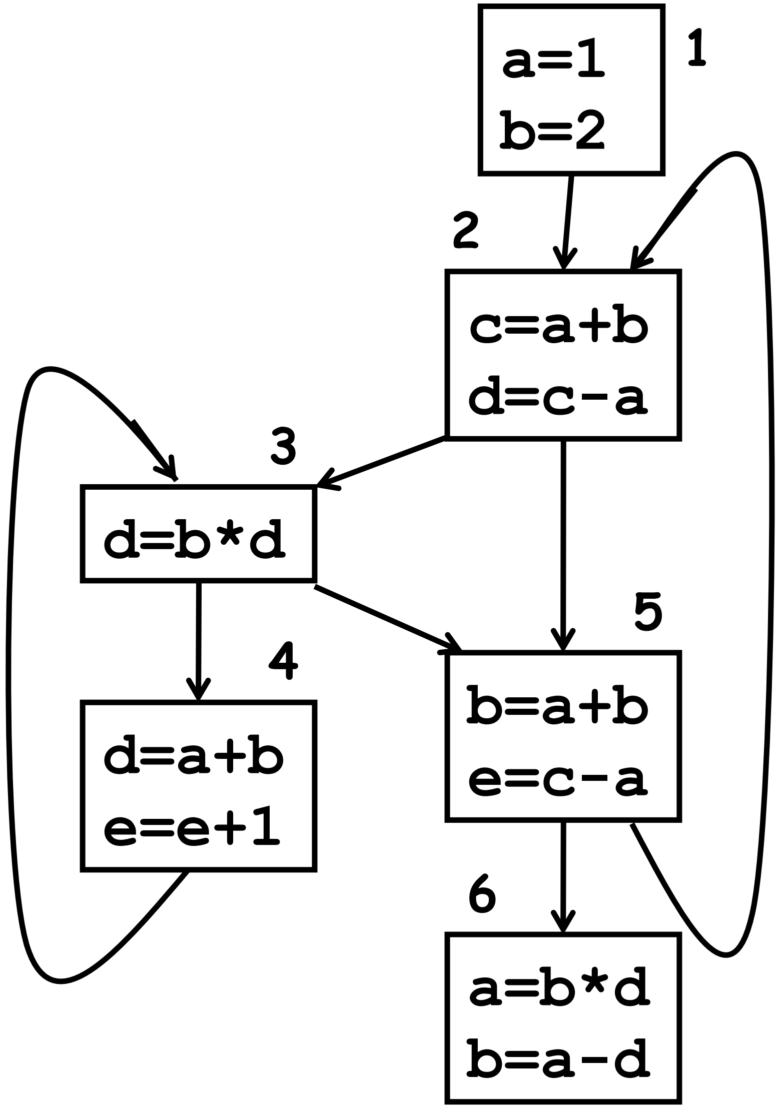
- Compute VN(y) and VN(z)
- Compute VN(@,y,z) for y@z
- New ➔ add VN(@,y,z)=VN(y)@VN(z)
- Put VN(@,y,z) into VN(x)
- This performs CSE already

VN(y) implementation for SSA

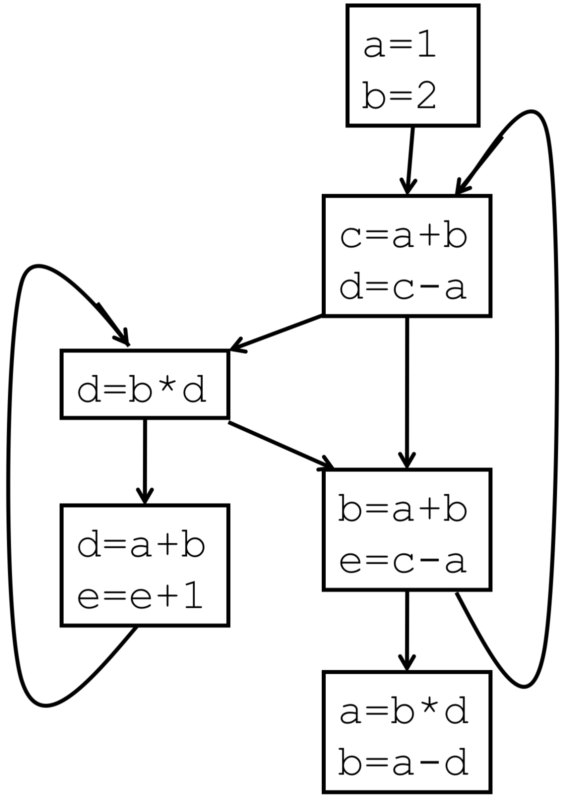
- Basic block has value w for y => use
- Exactly one predecessor => reuse VN(y) of predecessor
- More predecessors =>
  - get wi=VN(y) at each predecessor pi
  - Add VN(Φ,y,y)=Φ(w1,w2,…,wn)
    Nice: VN only adds Φ if still live
  - Put VN(Φ,y,y) for y

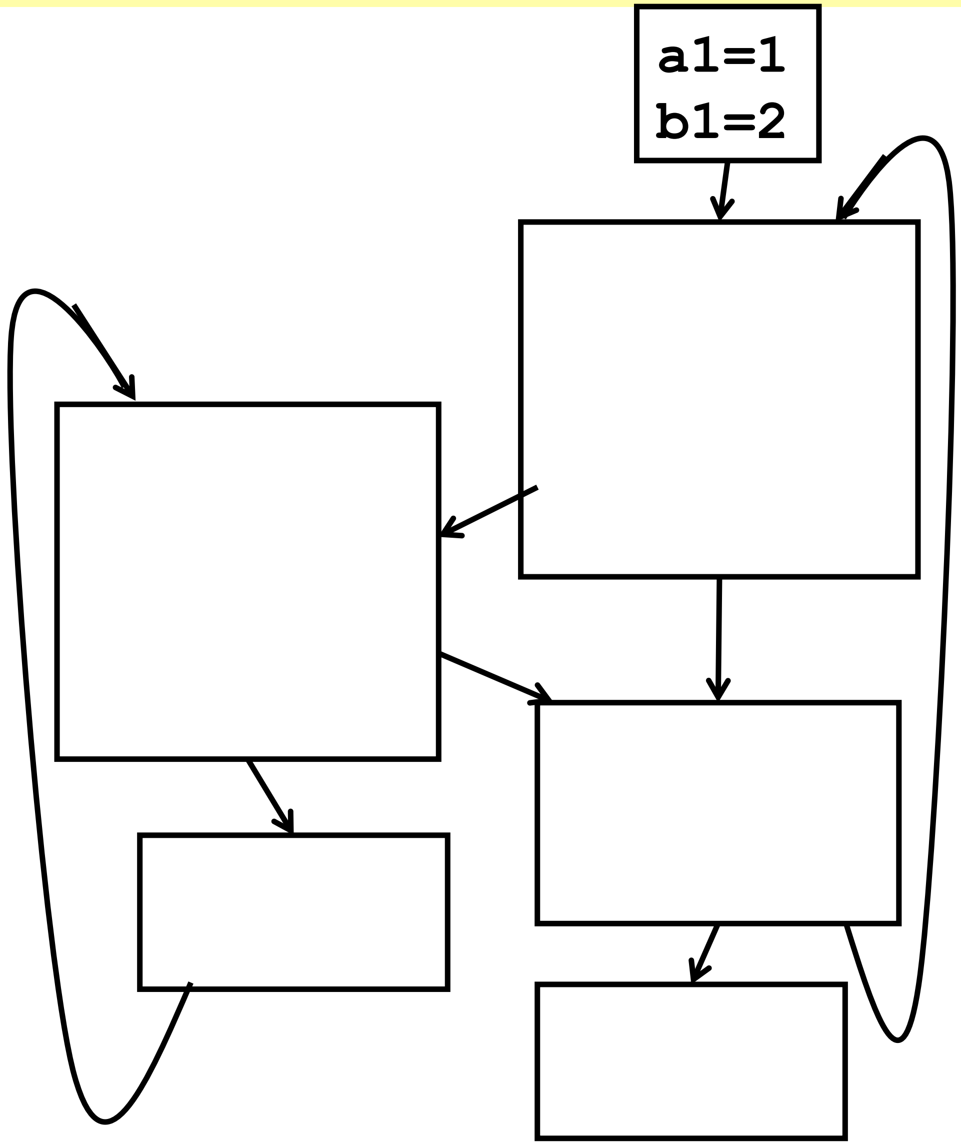
Basic block control flow graph

```
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
```



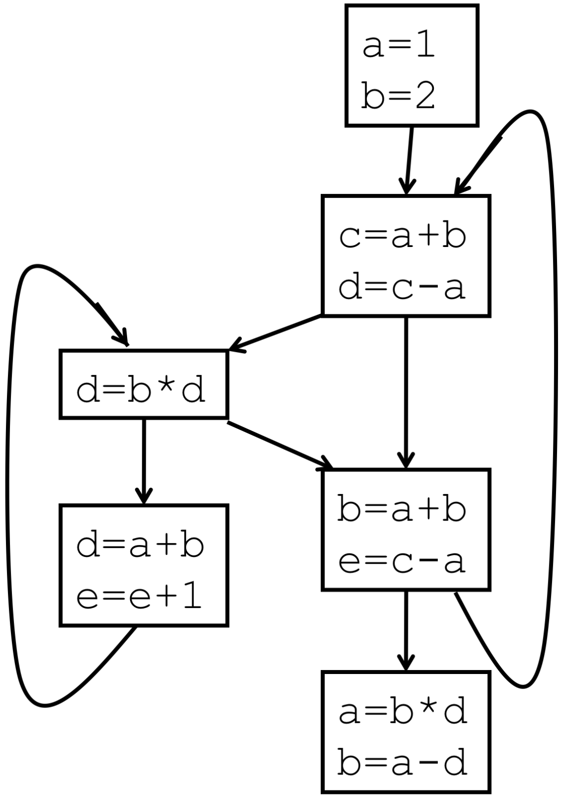
SSA AST-walk construction (1)

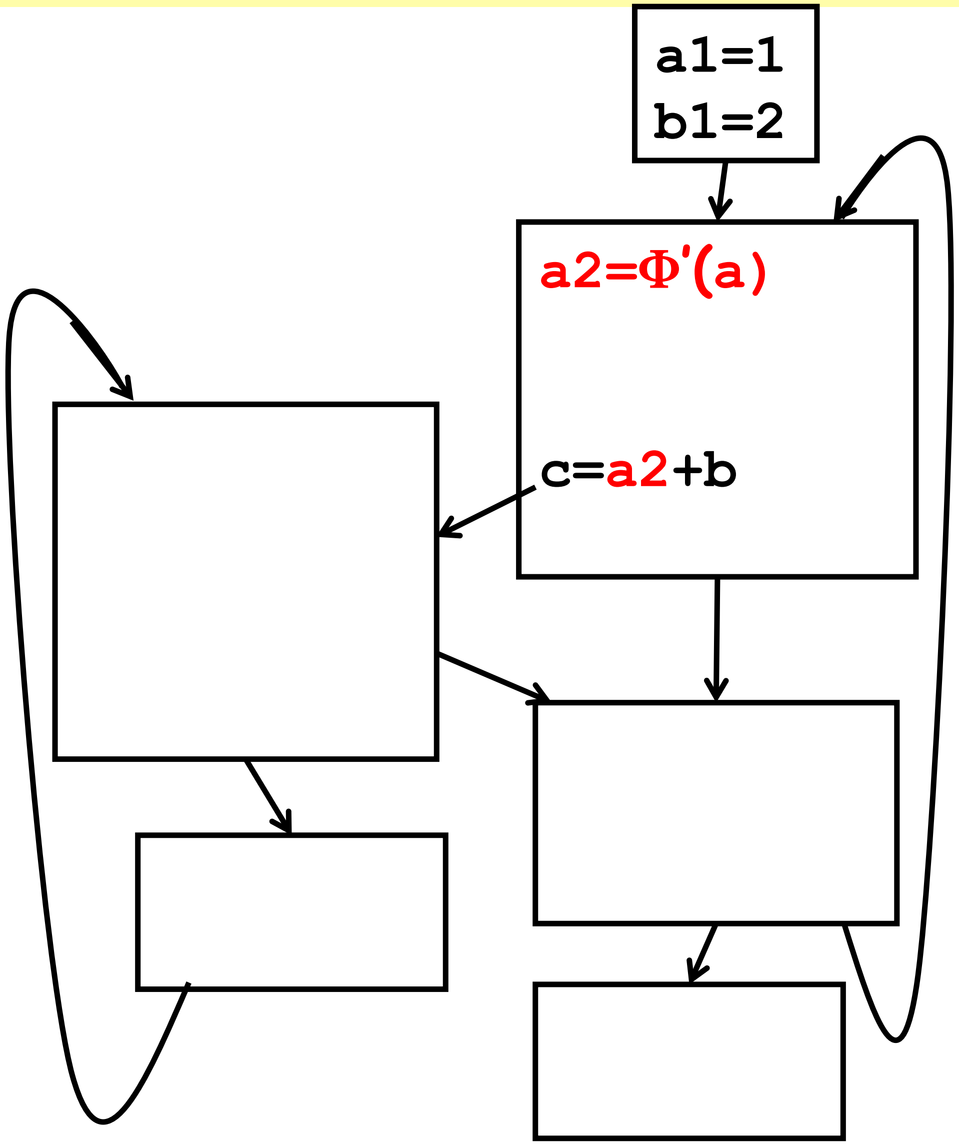




SSA AST-walk construction (2)

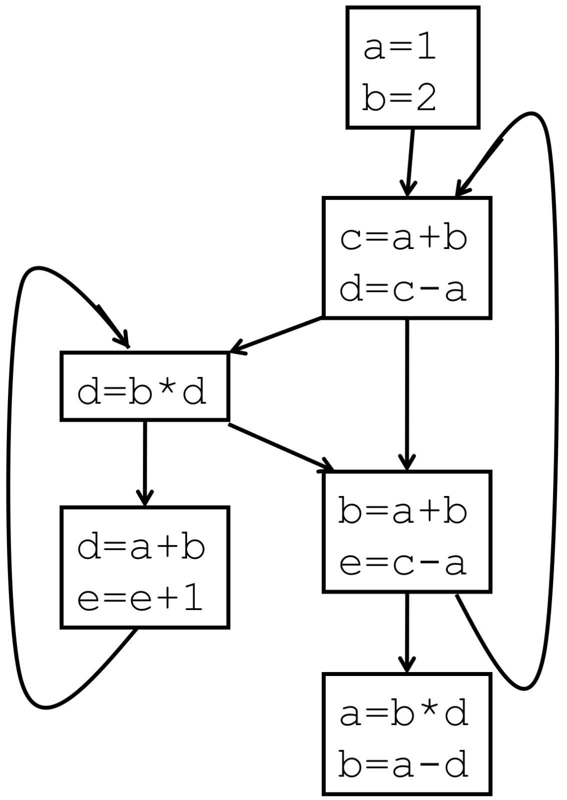
VN(a) => Φ'

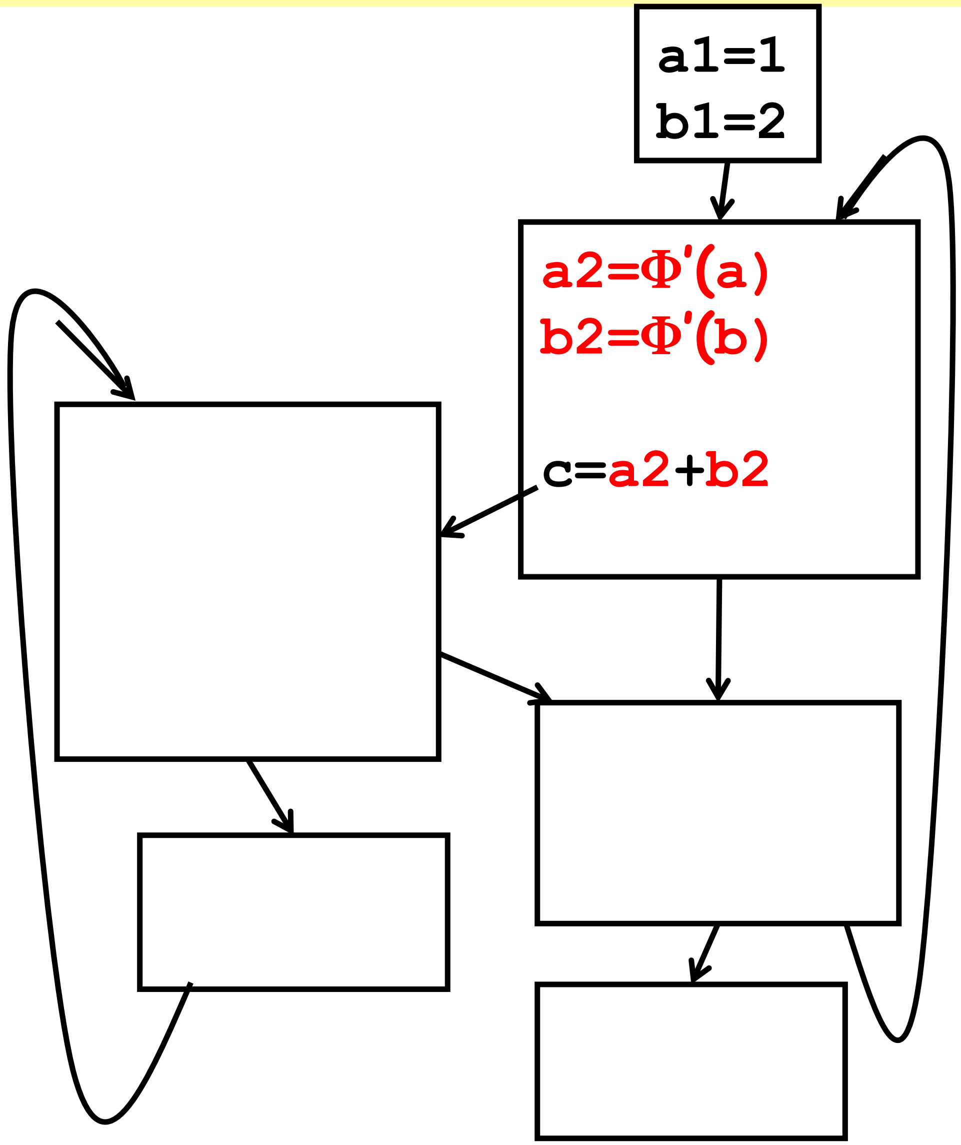




SSA AST-walk construction (2)

VN(a) => Φ'
VN(b) => Φ'





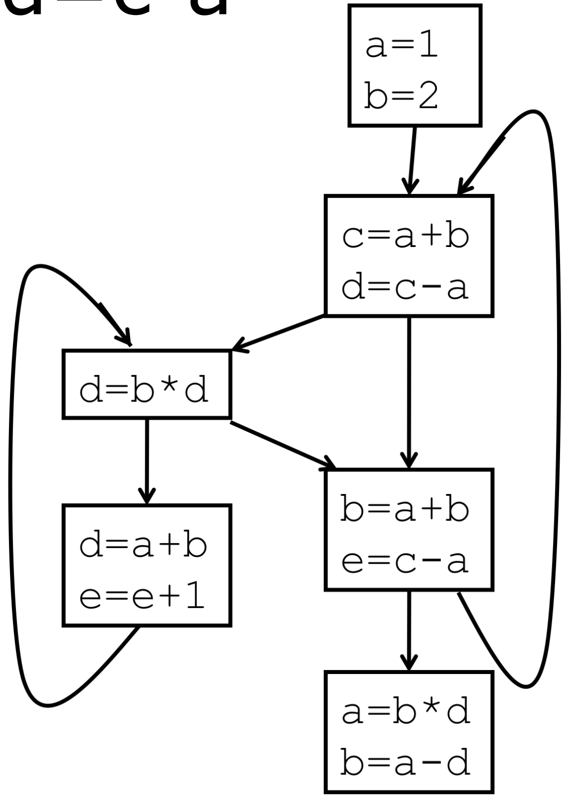
SSA AST-walk construction (2)

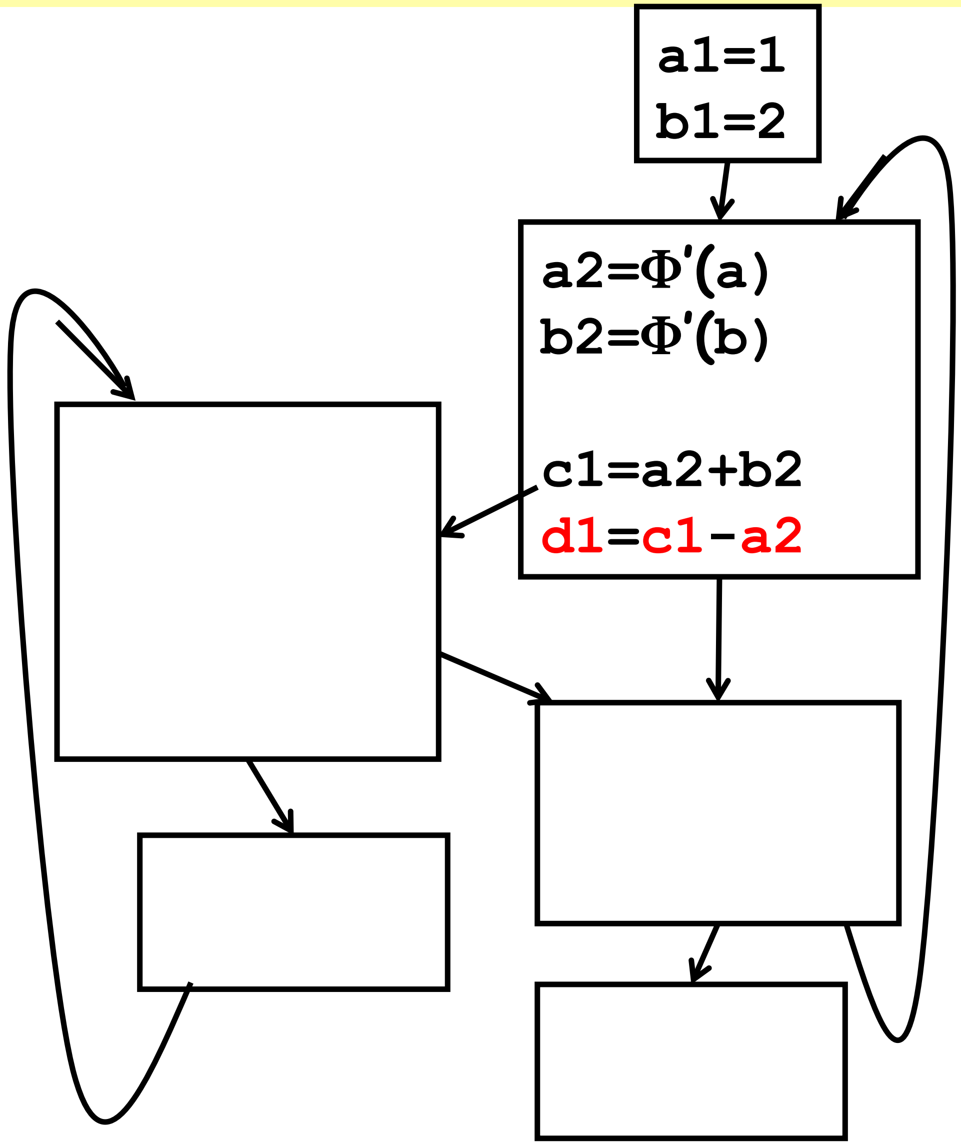
VN(a) => Φ'

VN(b) => Φ'

VN(c)

VN for d=c-a

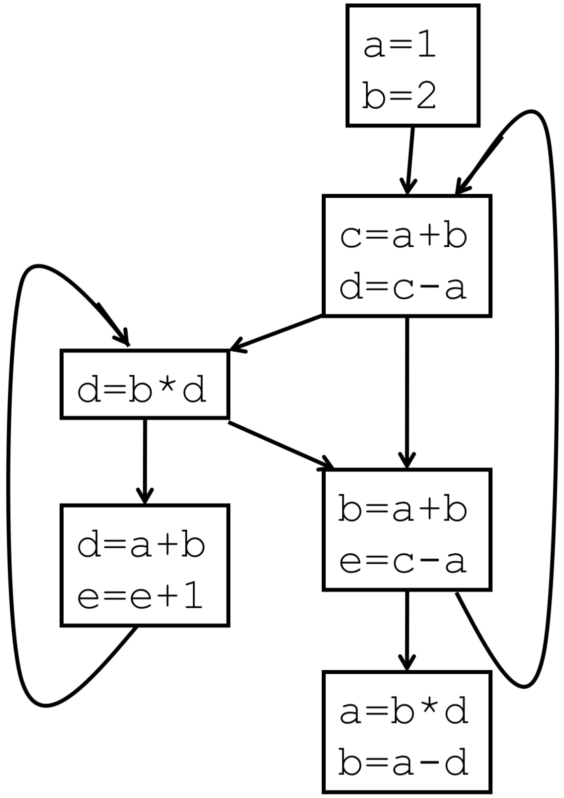


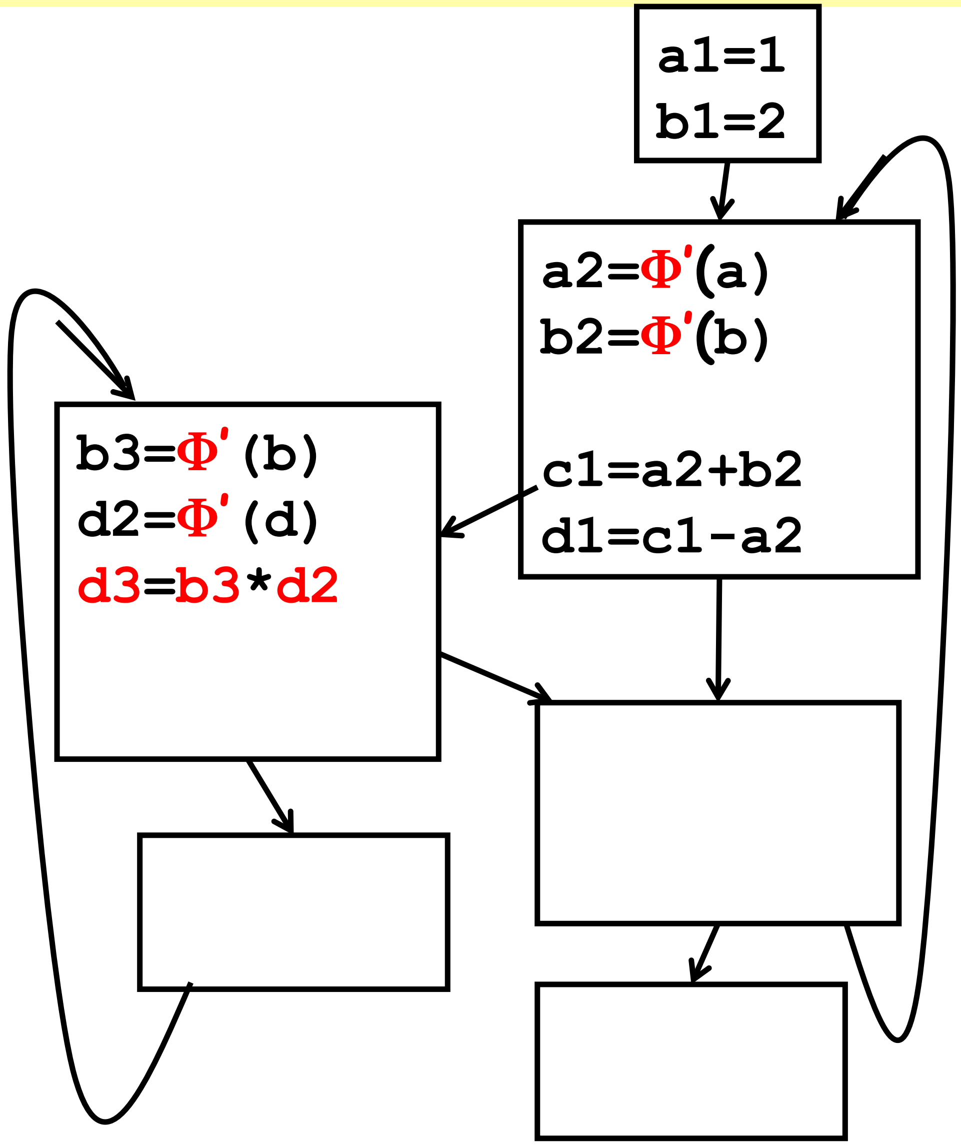


SSA AST-walk construction (3)

VN(b) => Φ′

VN(d) => Φ′



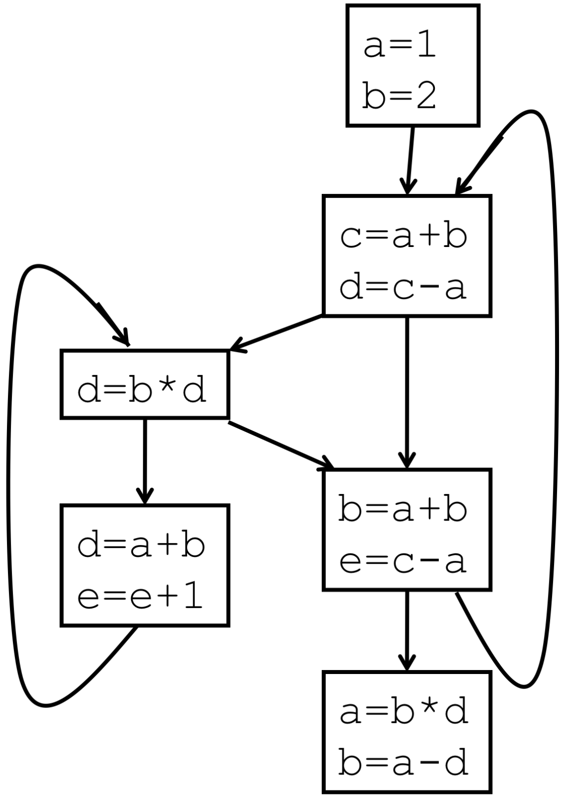


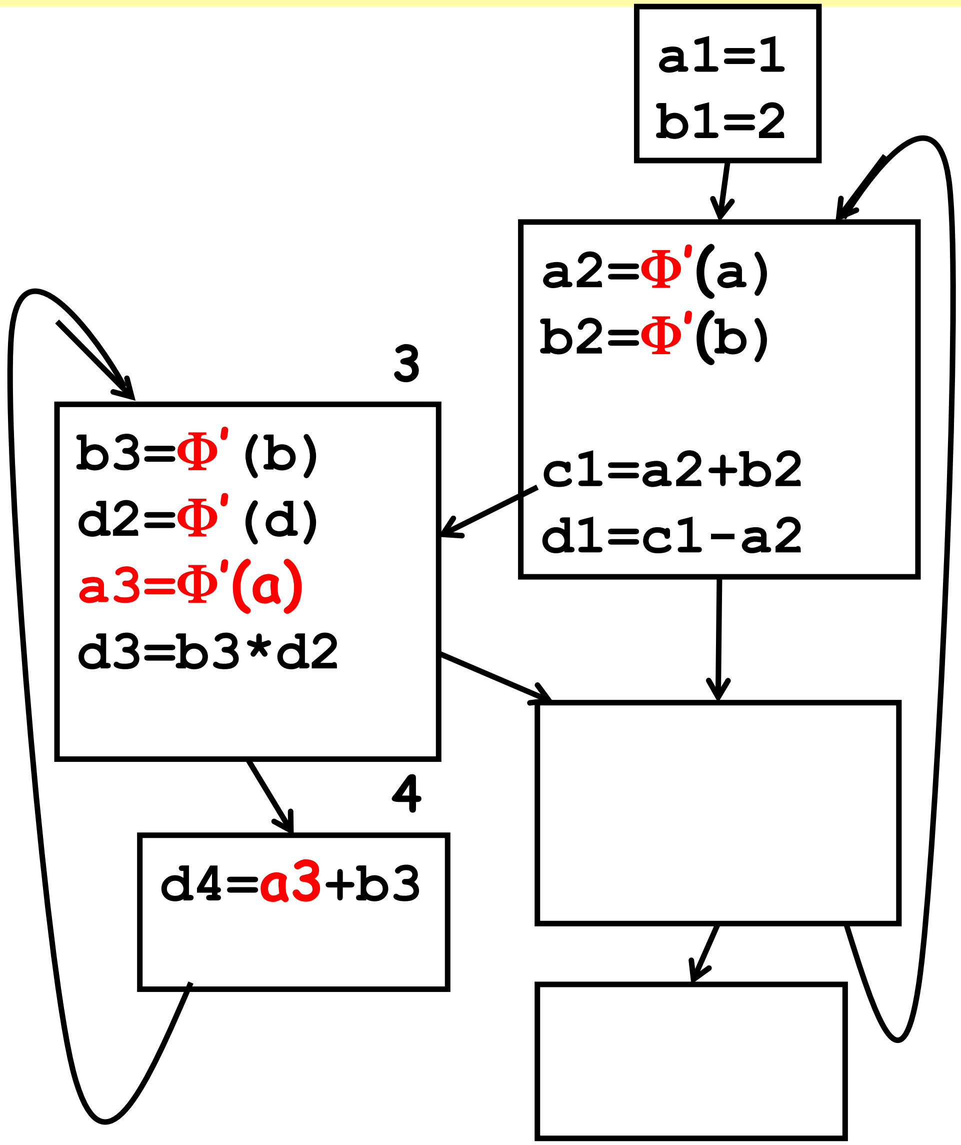
SSA AST-walk construction (4)

VN(a) at 4

⇒VN(a) at 3

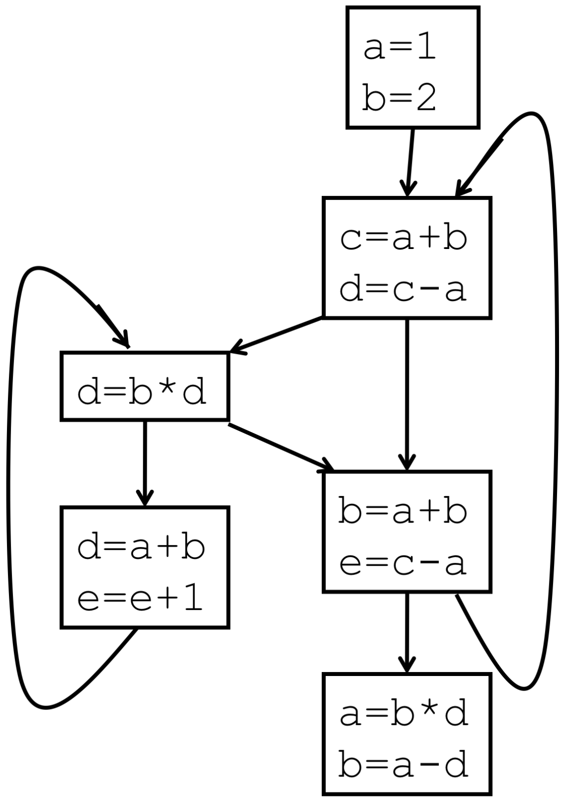
⇒ Φ′ at 3

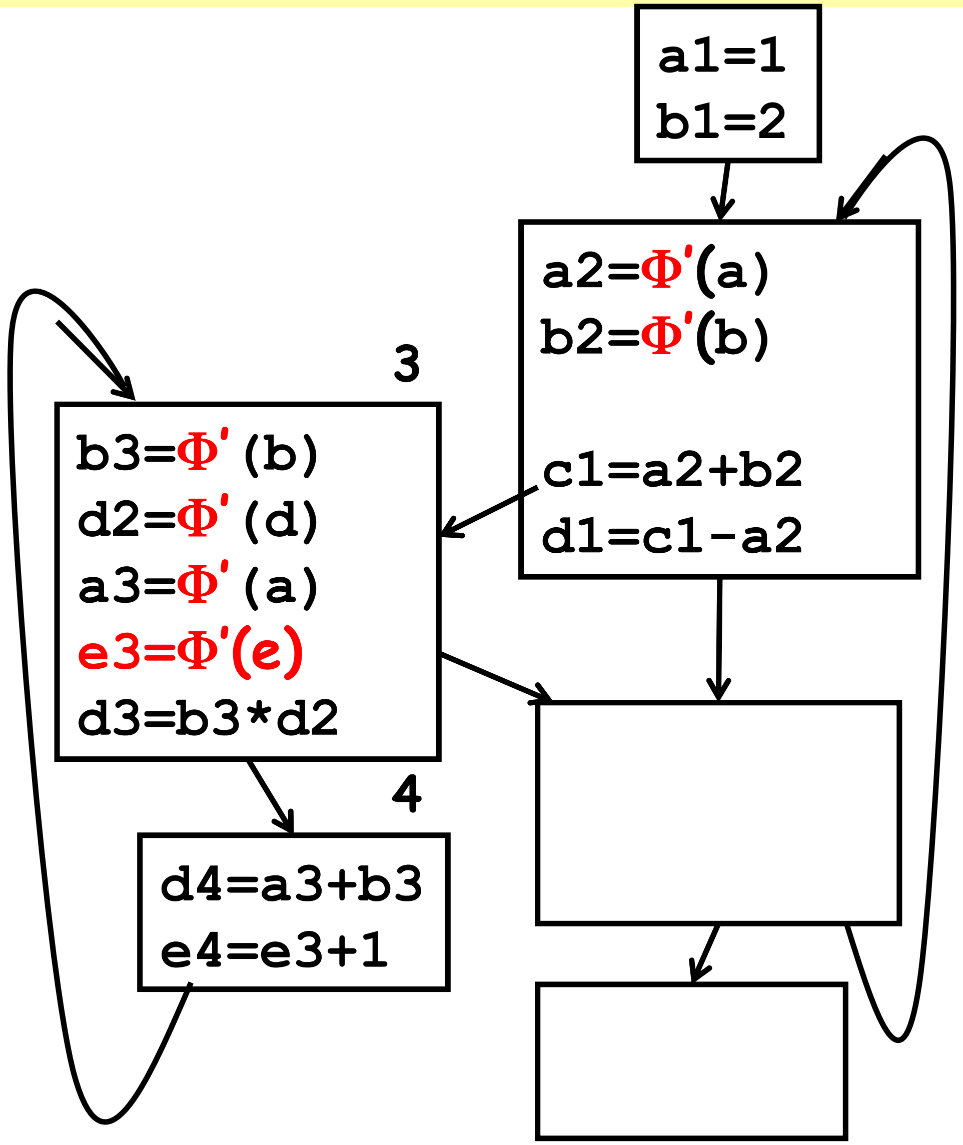




SSA AST-walk construction (4)

VN(e) at 4 asks
⇒VN(e) at 3
⇒ Φ' at 3



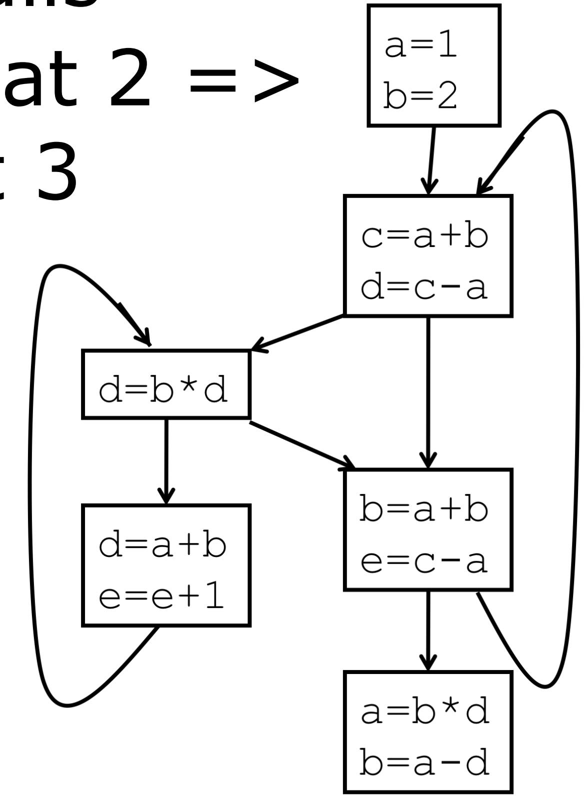


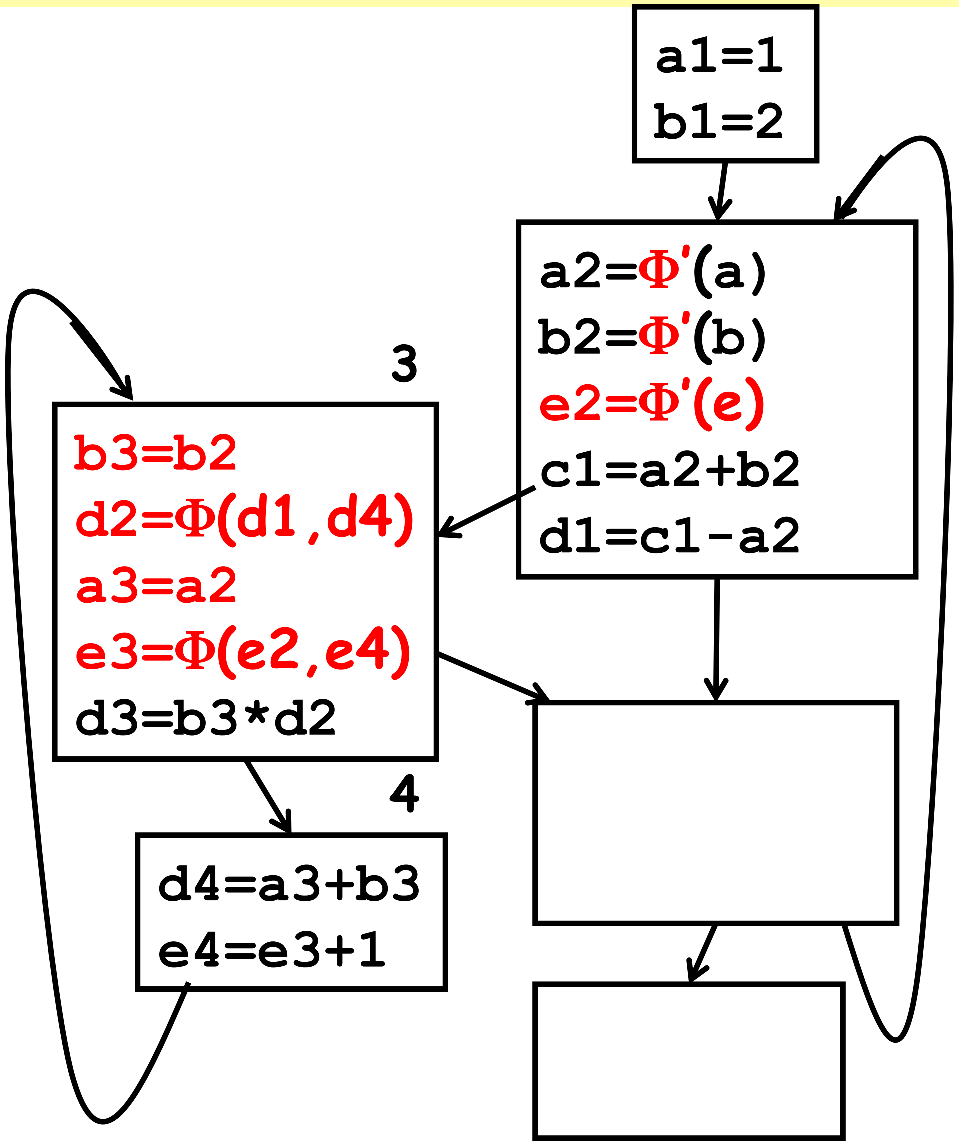
SSA AST-walk construction (4)

Φ' at 3 knows
all predecs.

$\Rightarrow \Phi' \Rightarrow \Phi$

This calls
VN(e) at 2 =>
Φ' at 3

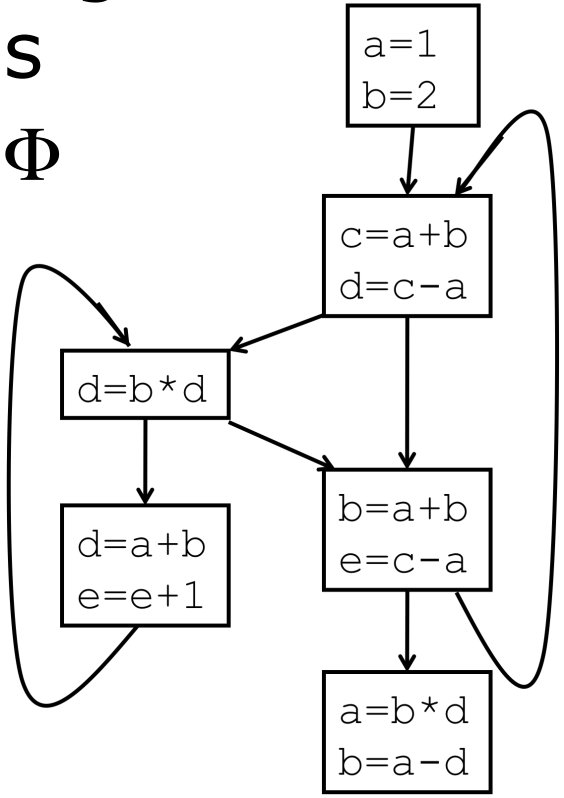


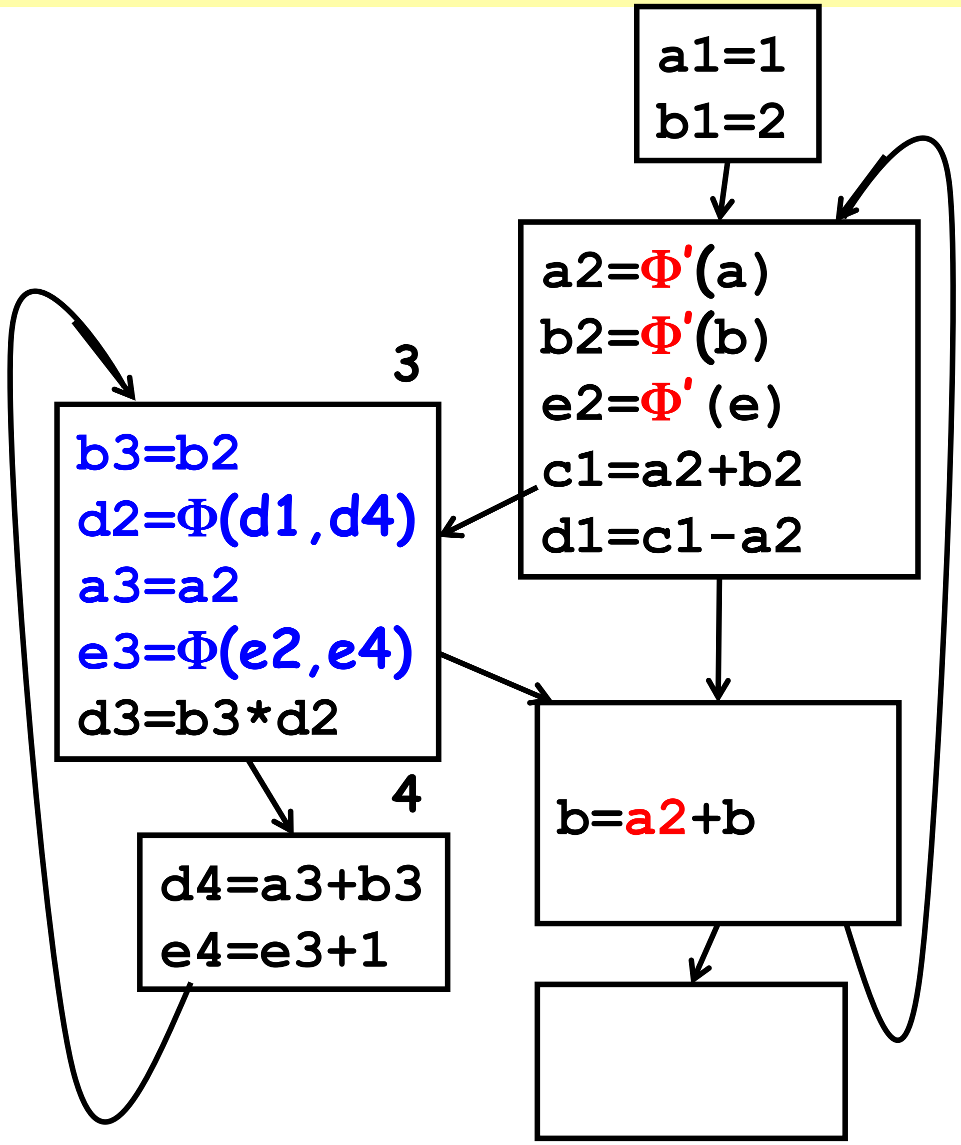


SSA AST-walk construction (5)

VN(a) at 5 has unique a2 when skipping copies

=> no Φ

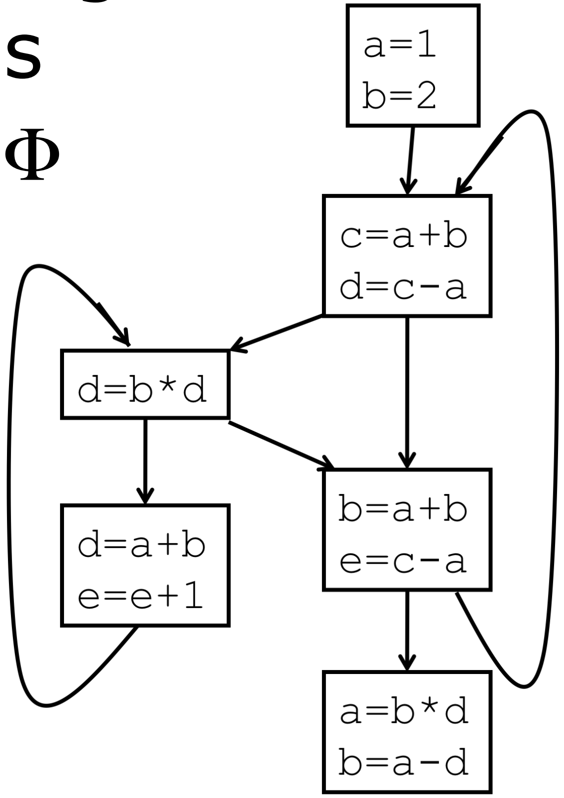


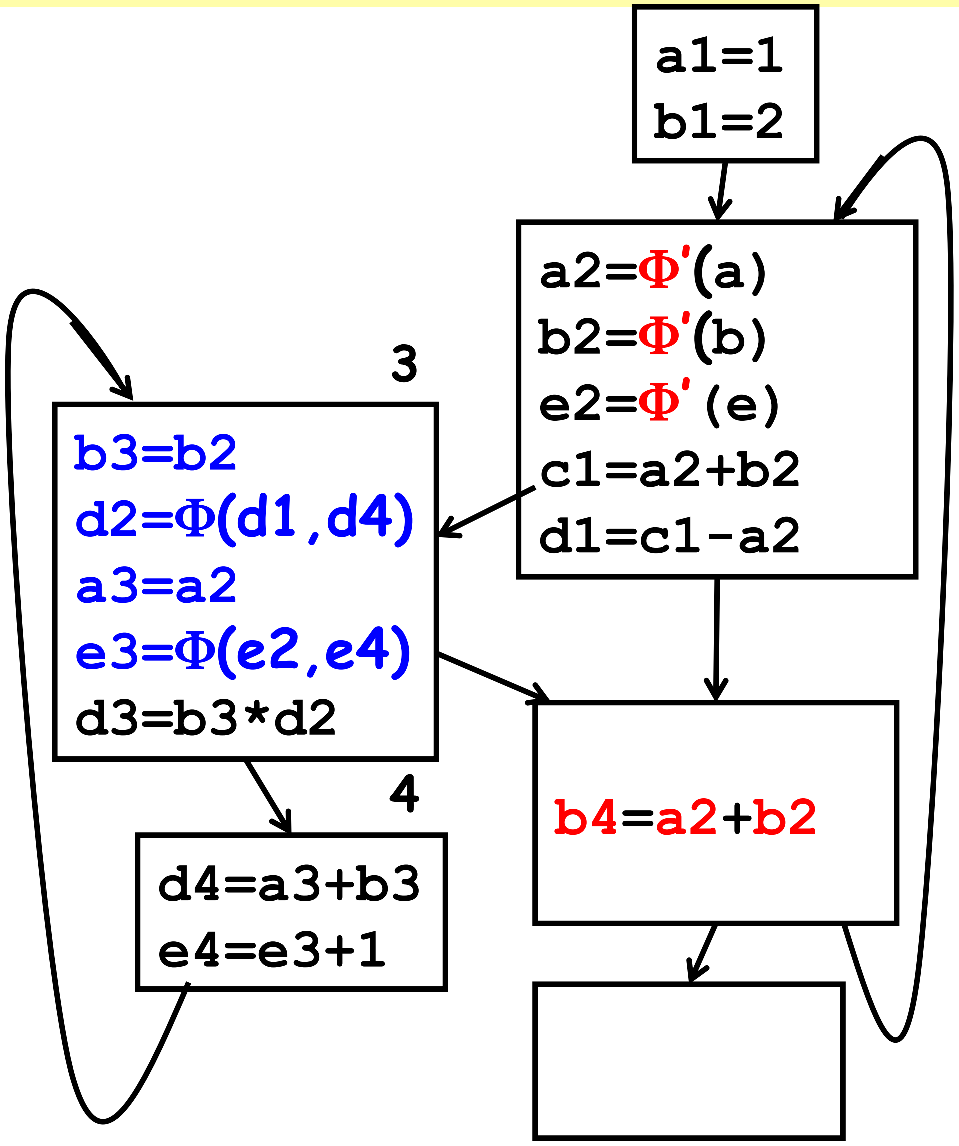


SSA AST-walk construction (5)

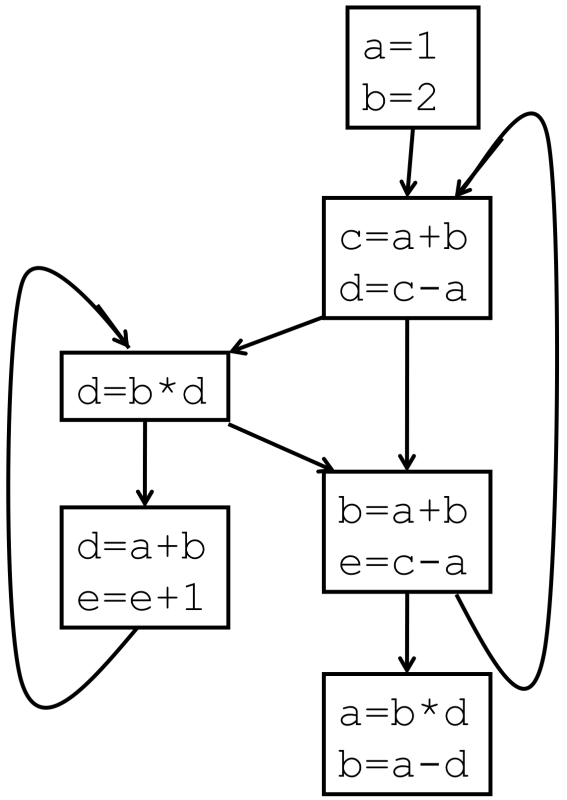
VN(b) at 5 has unique b2 when skipping copies

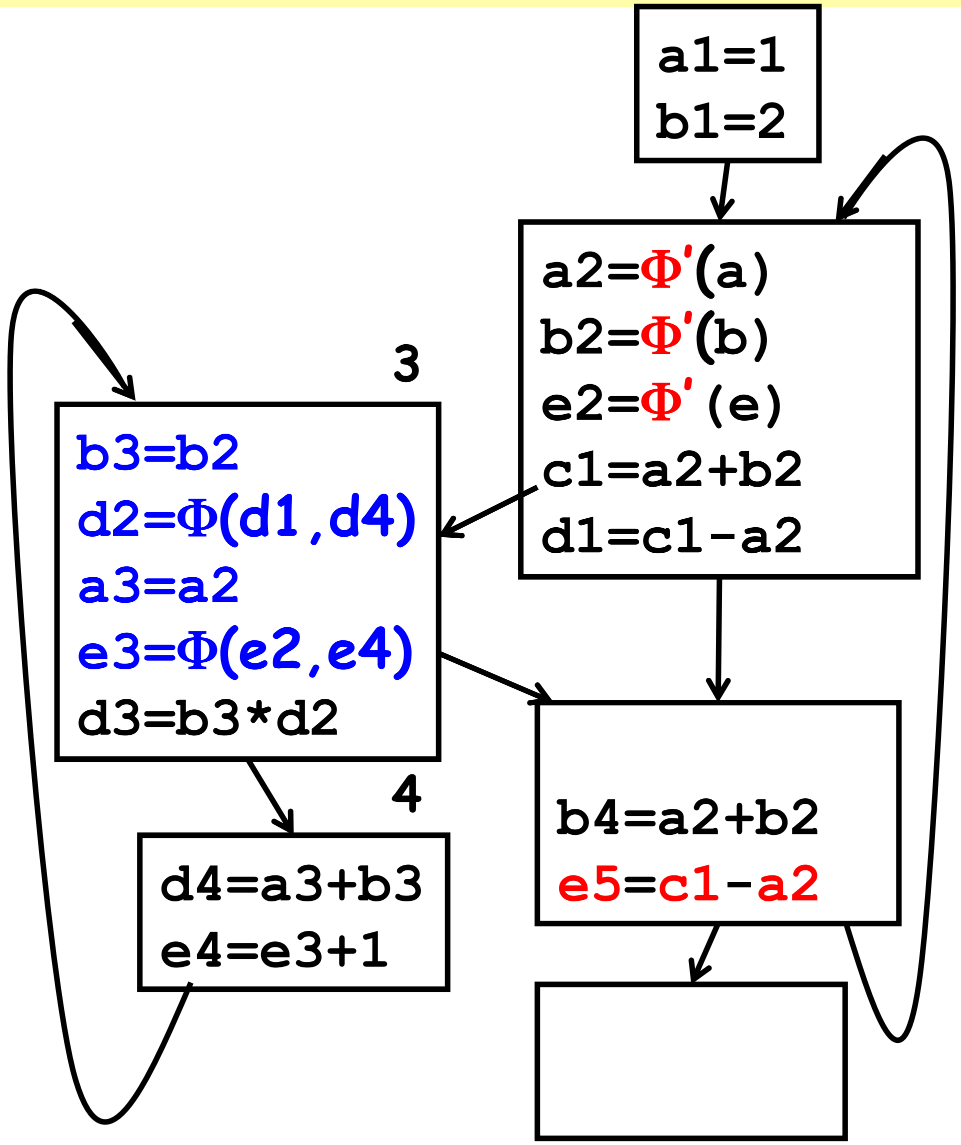
=> no Φ





SSA AST-walk construction (5)



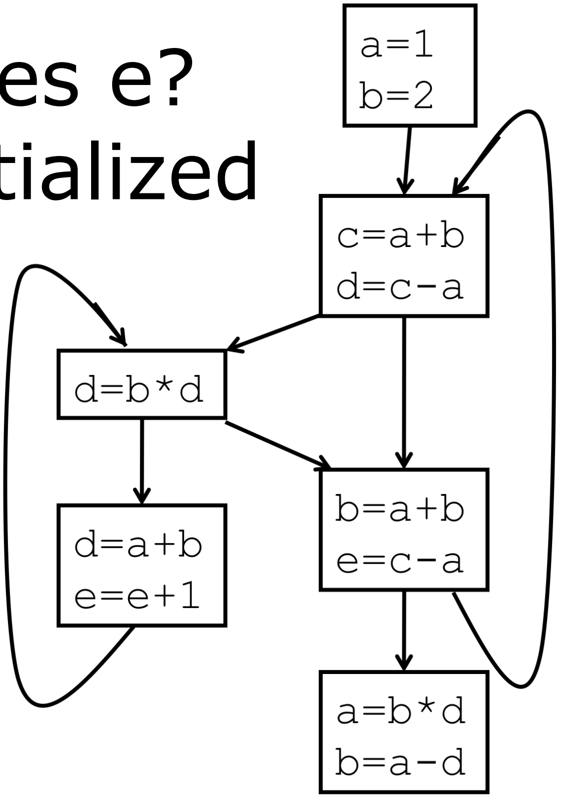


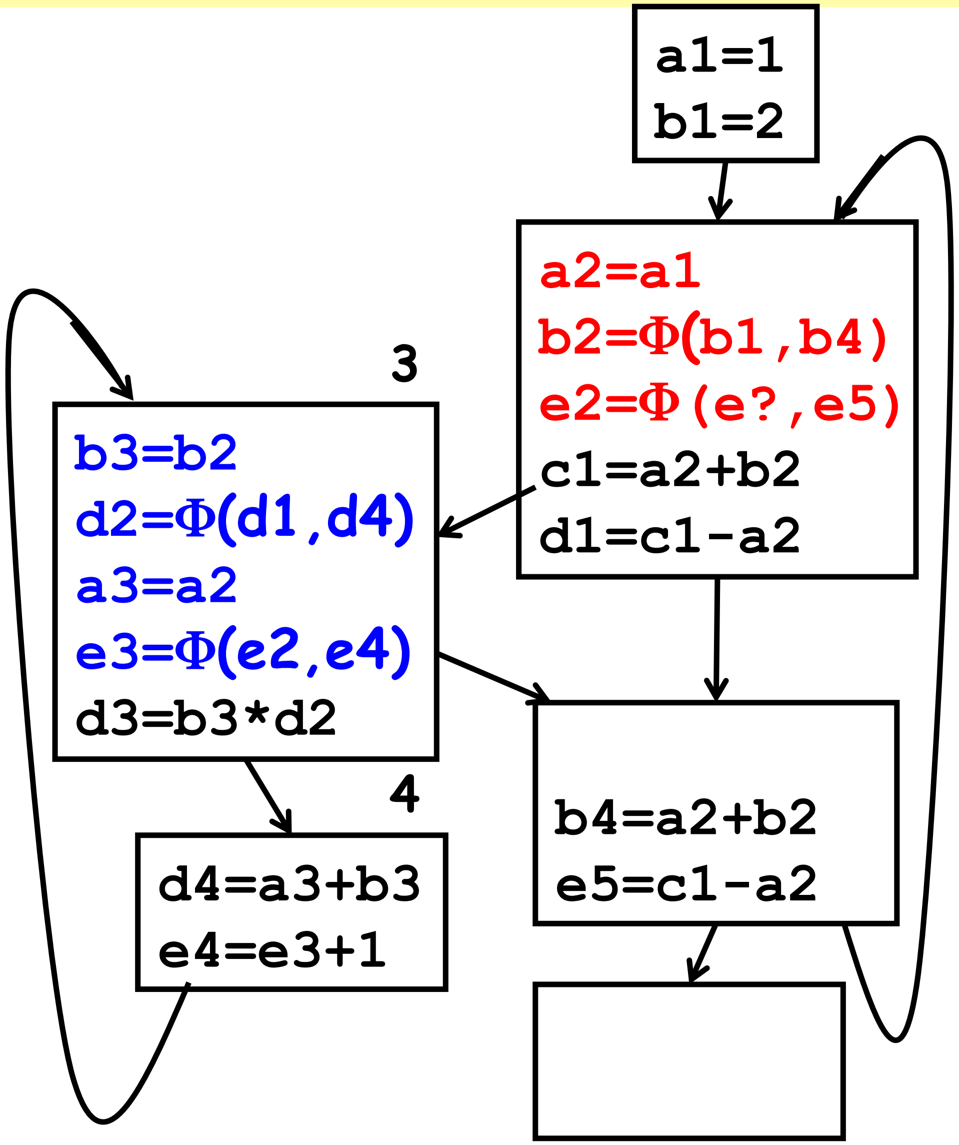
SSA AST-walk construction (5)

Φ′ at 2 knows all predecs.

⇒Φ′ => Φ

SSA sees e? uninitialized



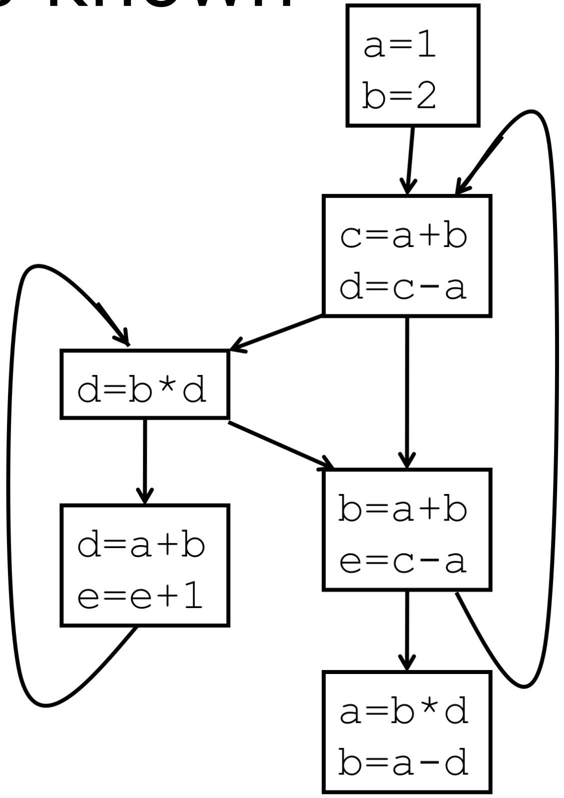


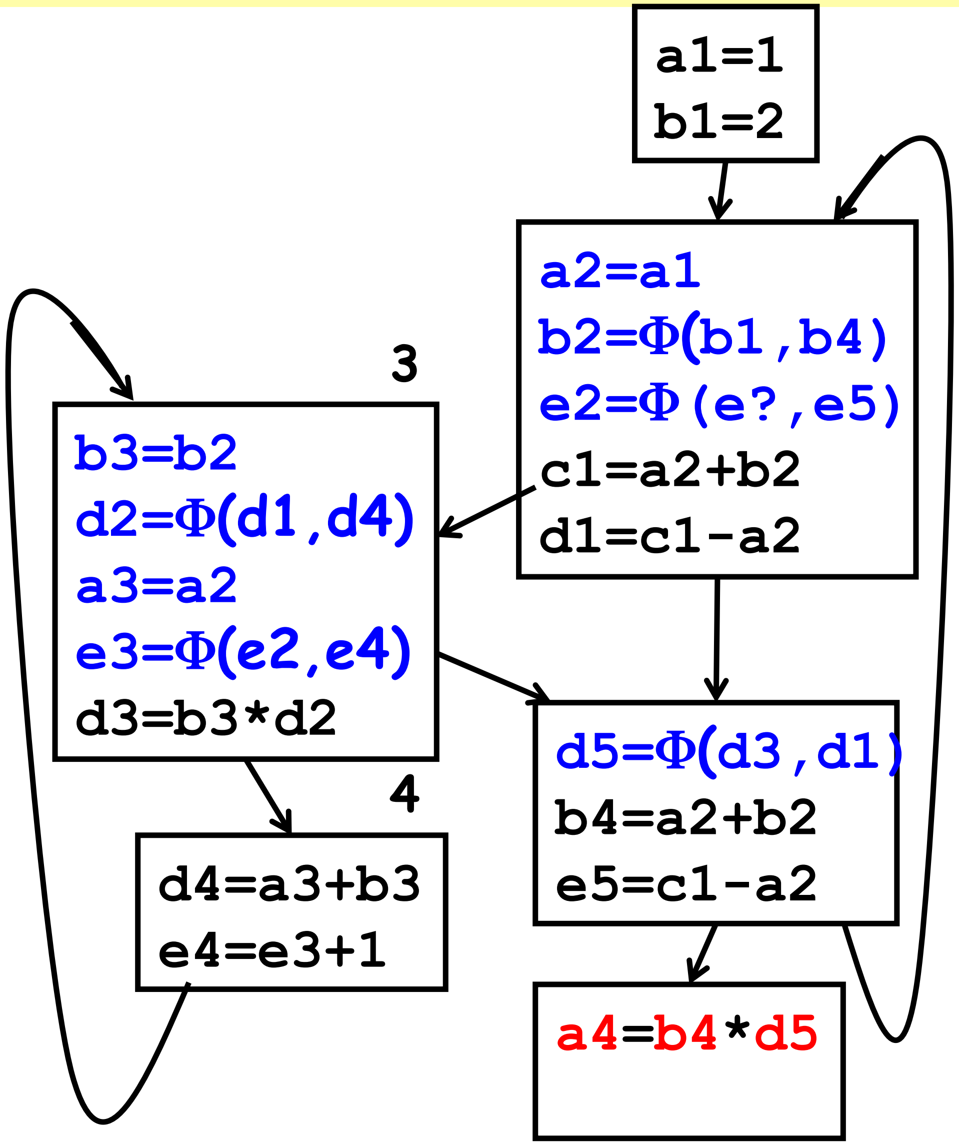
SSA AST-walk construction (6)

VN(d) at 6 ask

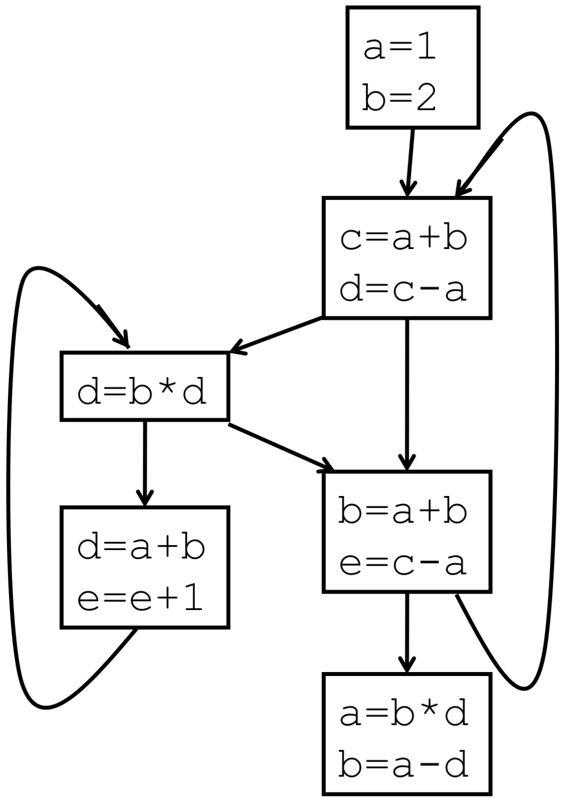
⇒VN(d) at 5

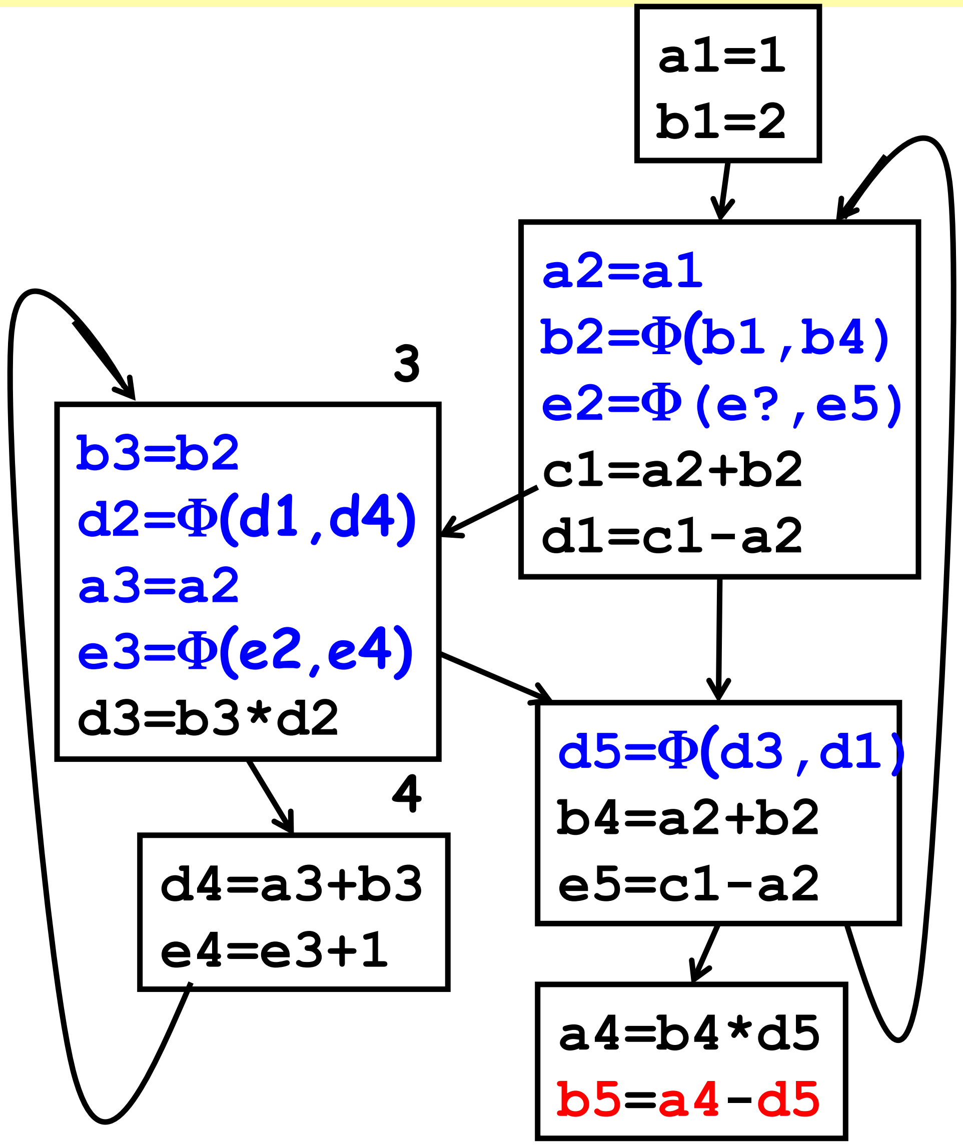
⇒ Φ at 5 as all preds known



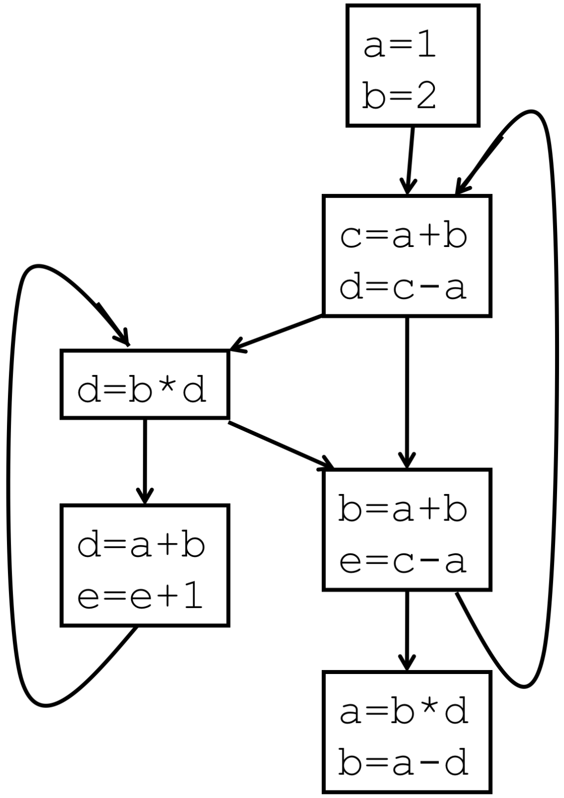


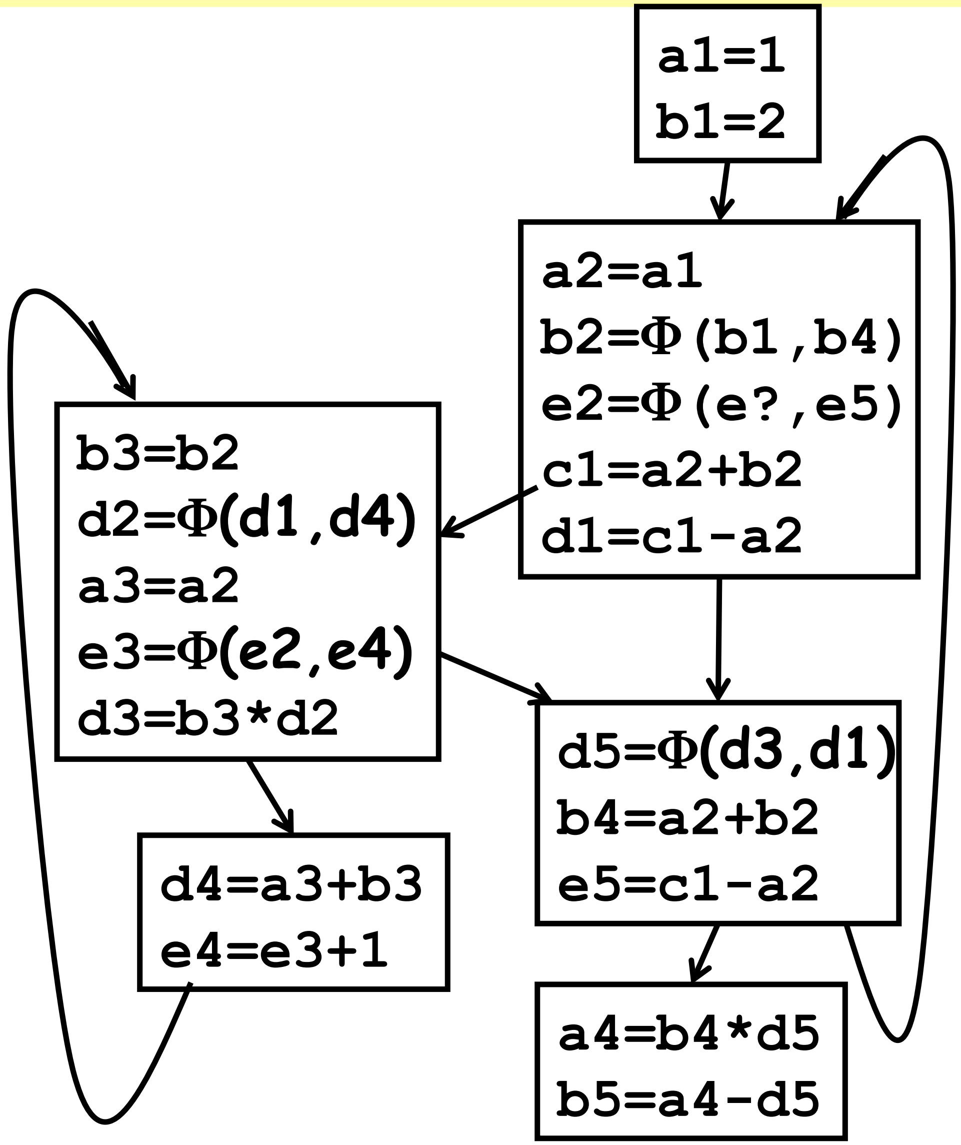
SSA AST-walk construction (6)



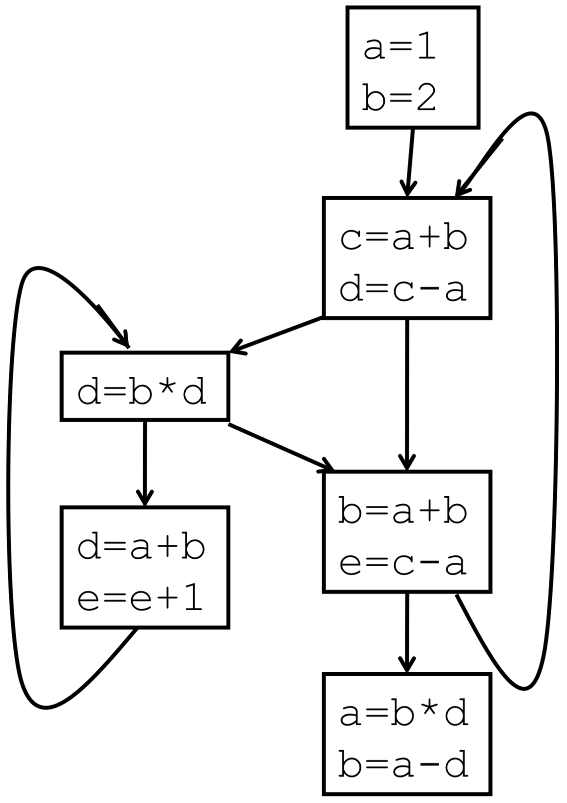


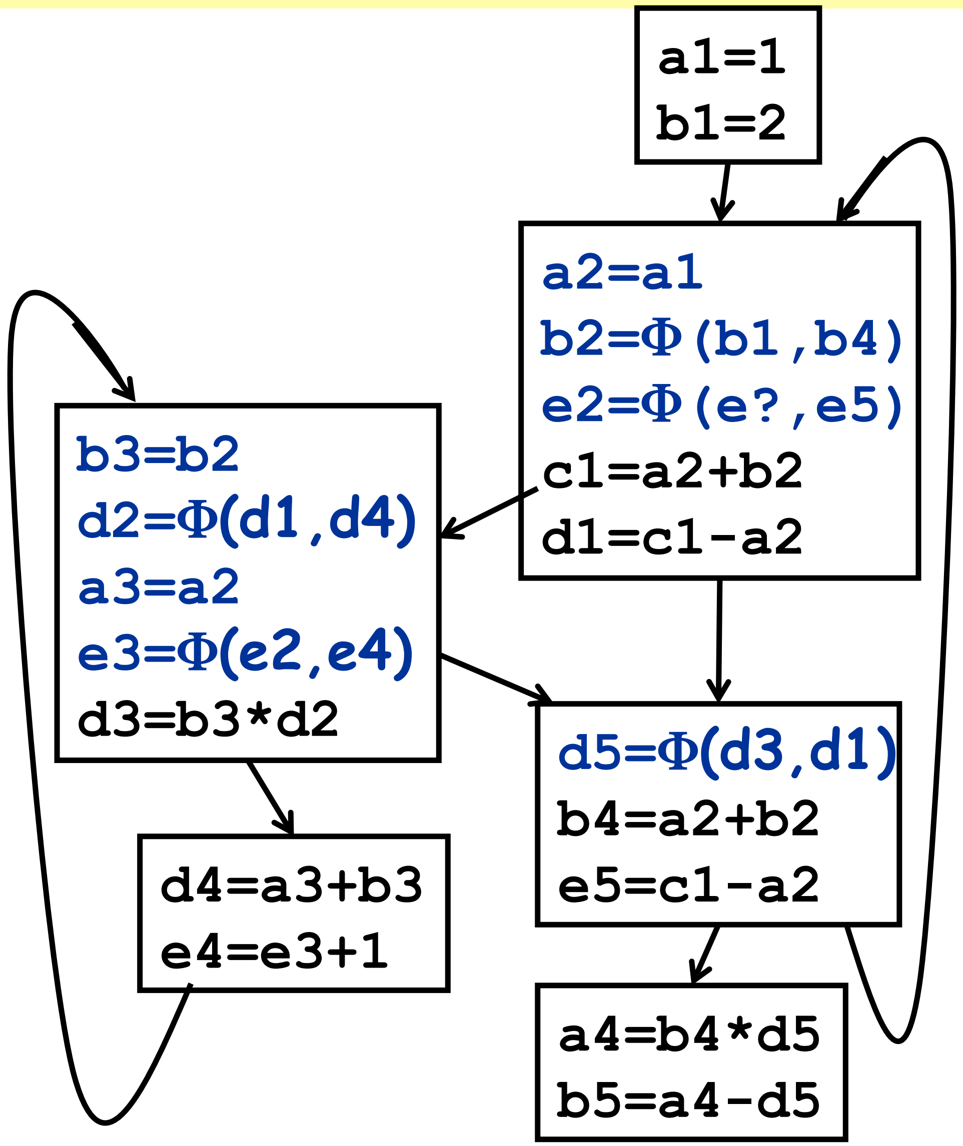
SSA AST-walk construction





SSA AST-walk constructed



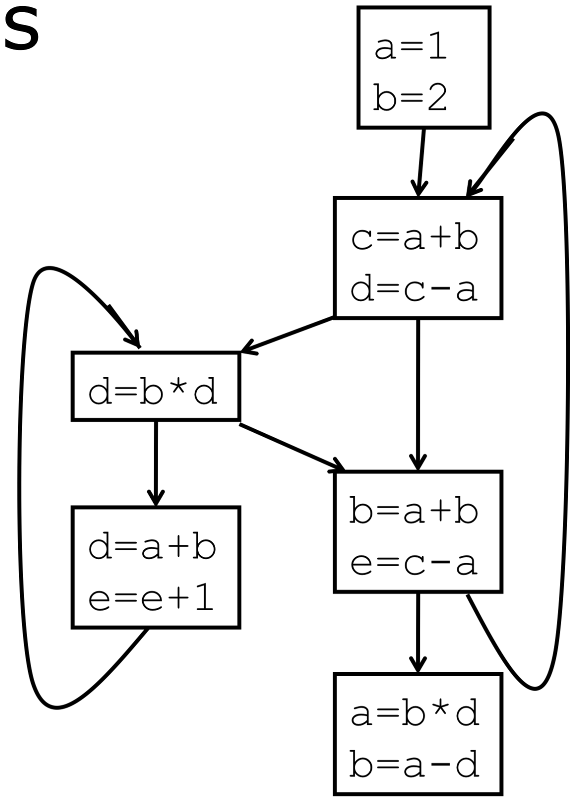


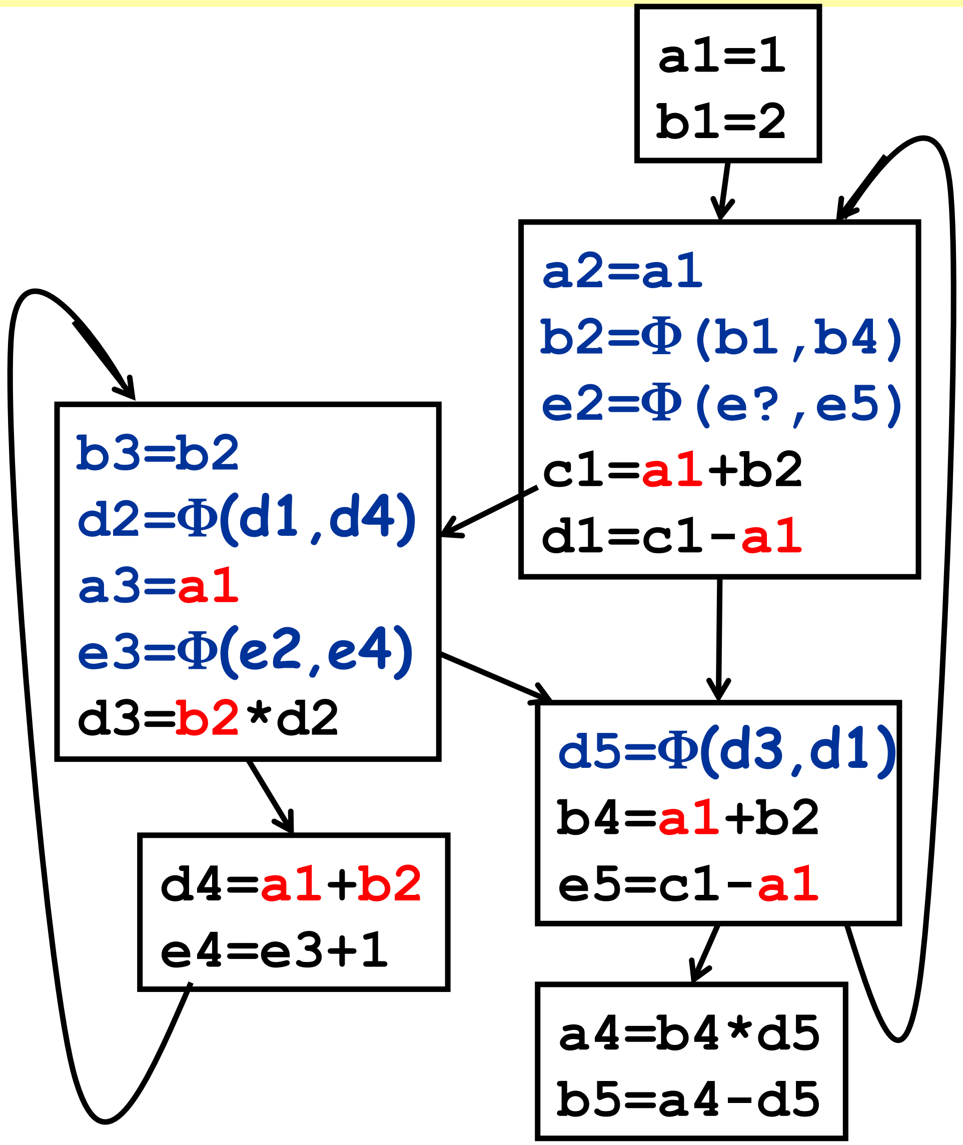
SSA Opt: copy propagate

substitute

a2=a1

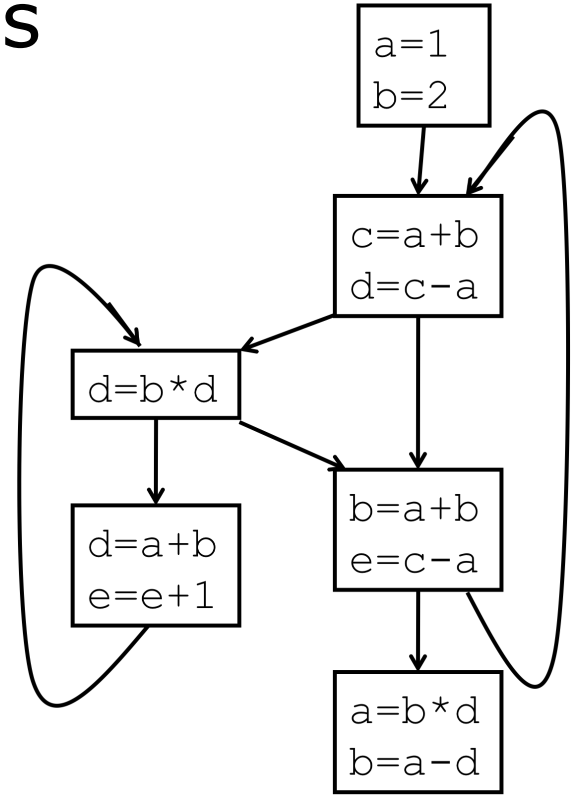
into all dominated nodes

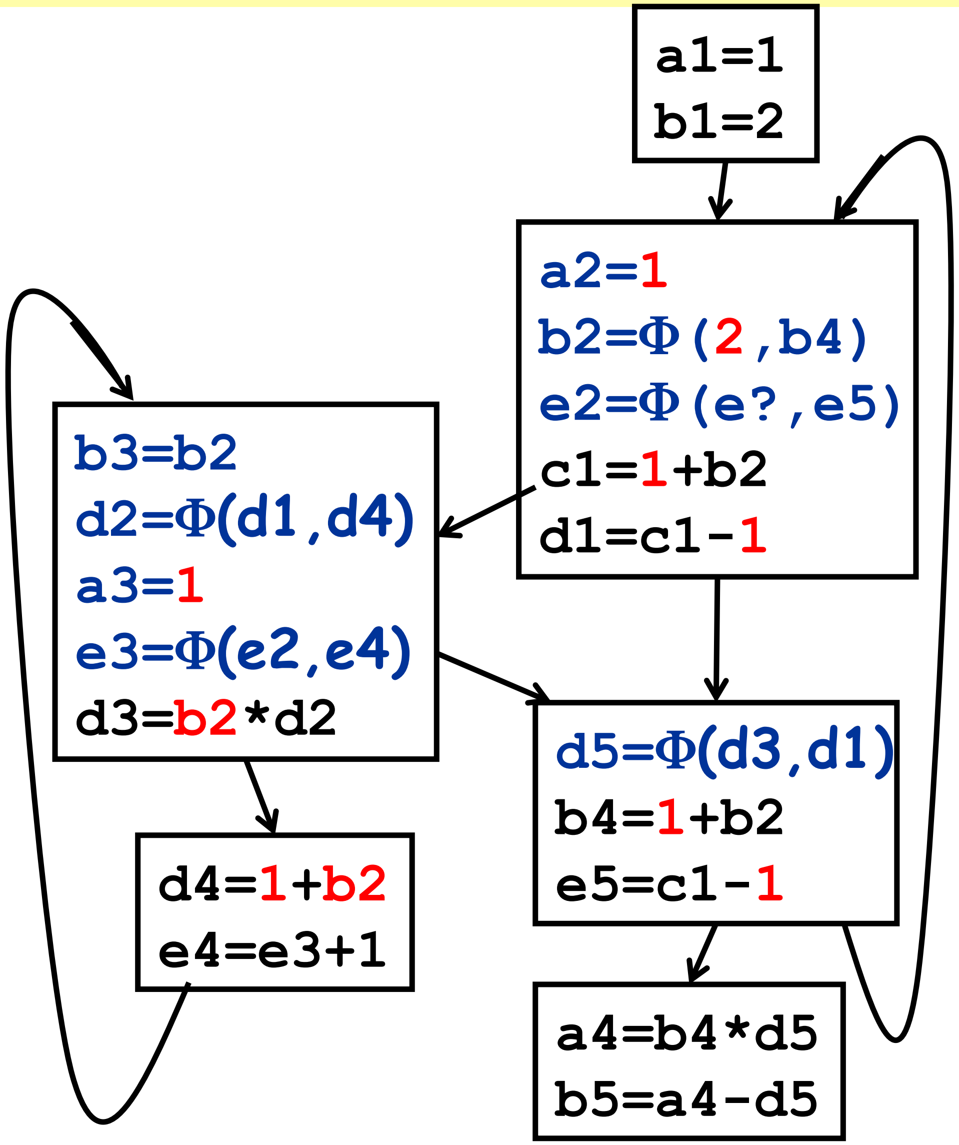




SSA Opt: constant propagate

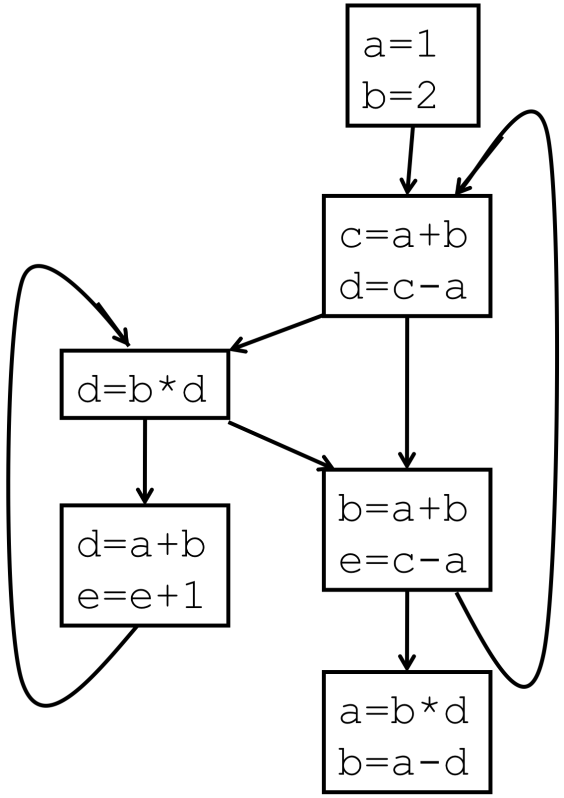
substitute
a1=1
into all
   dominated
   nodes

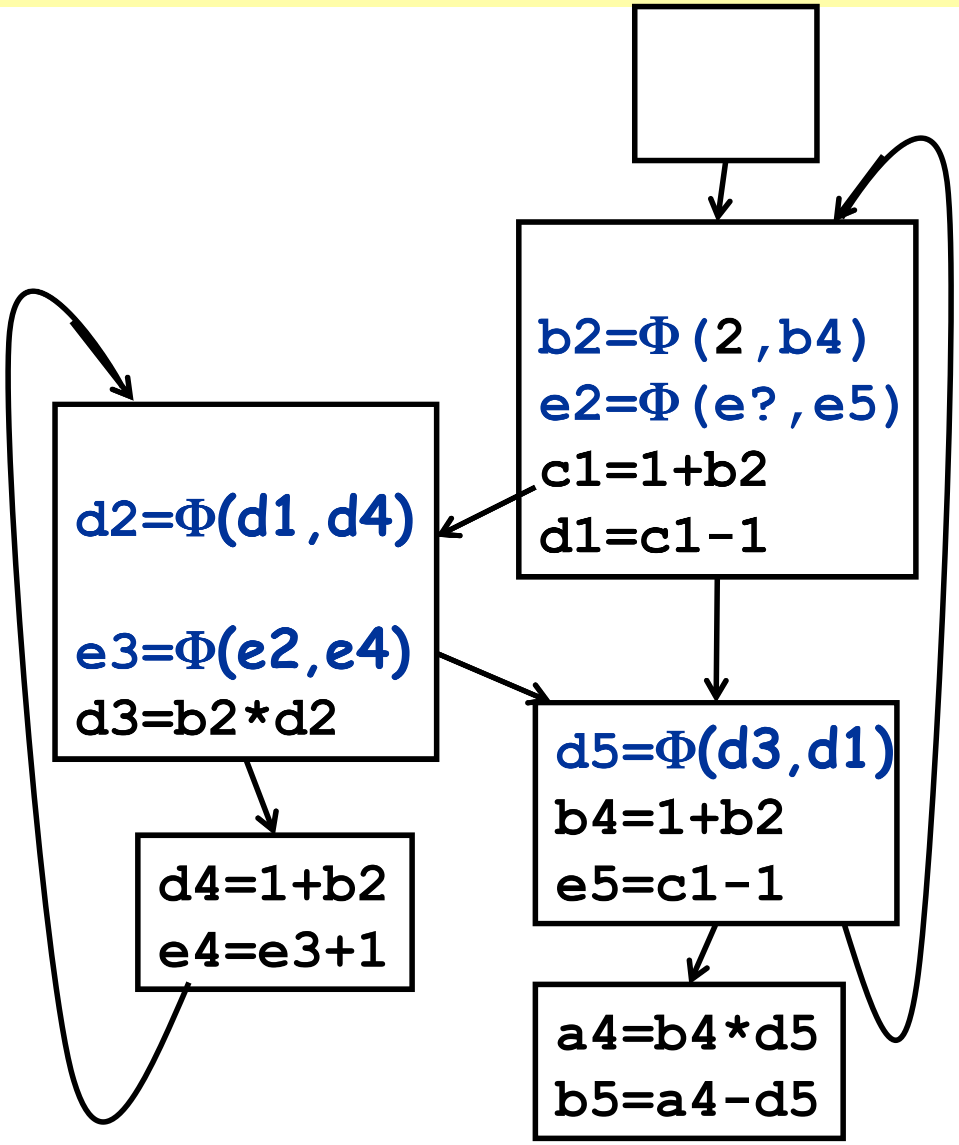




SSA Opt: eliminate dead code

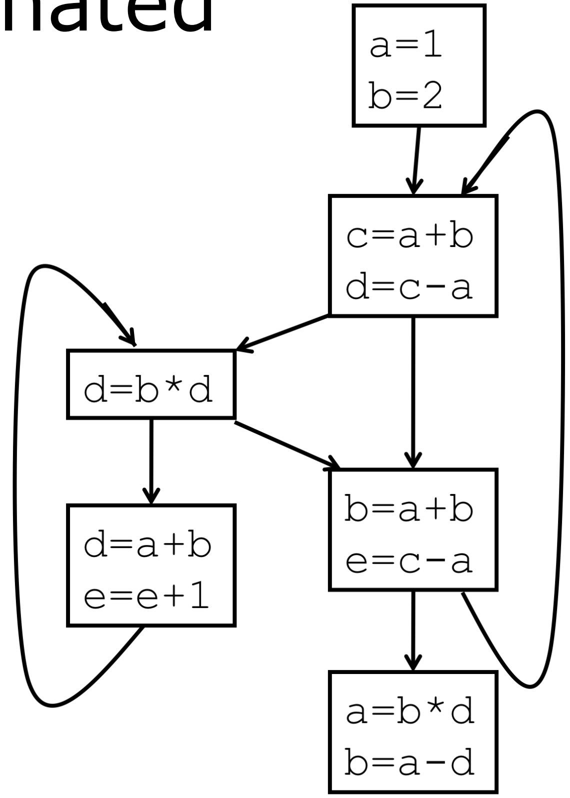
a1 never read then dead

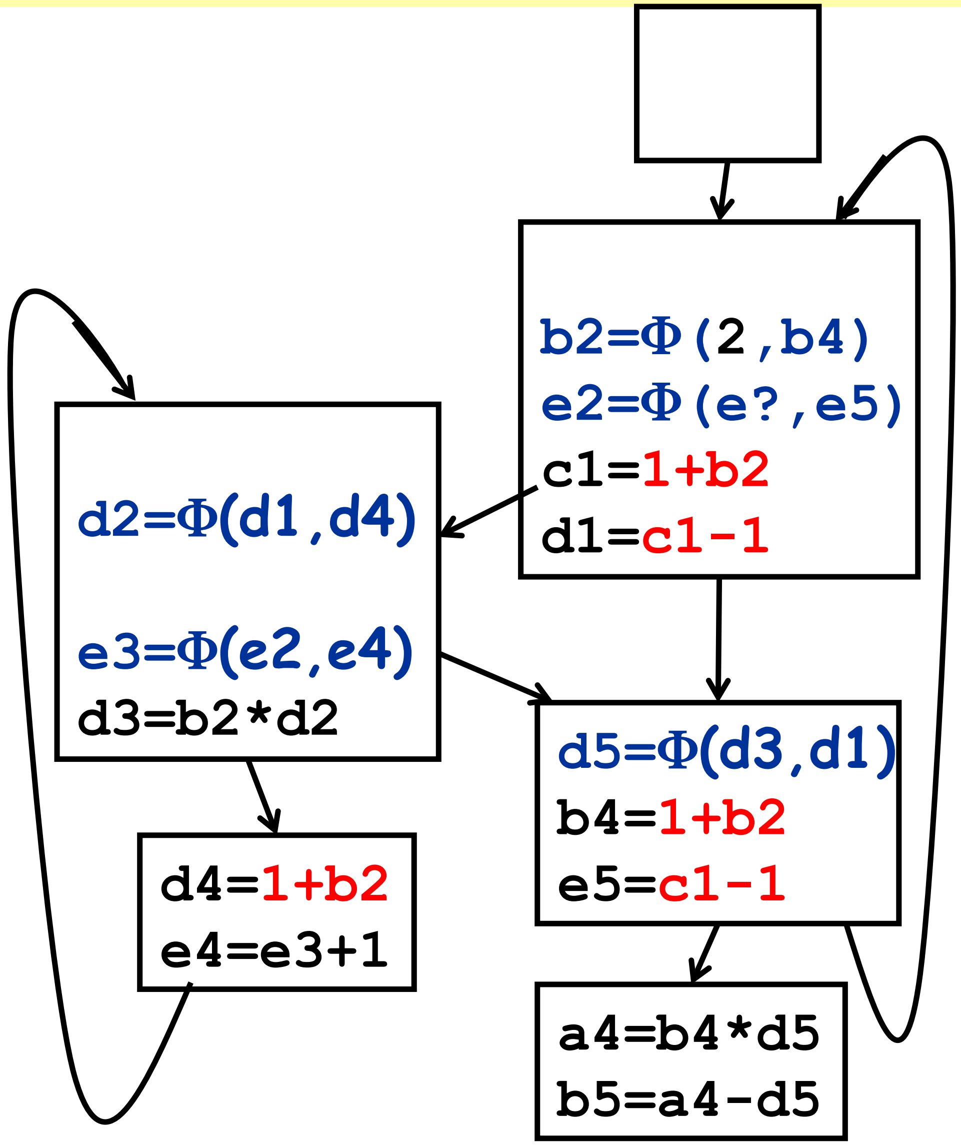




SSA Opt: CSE

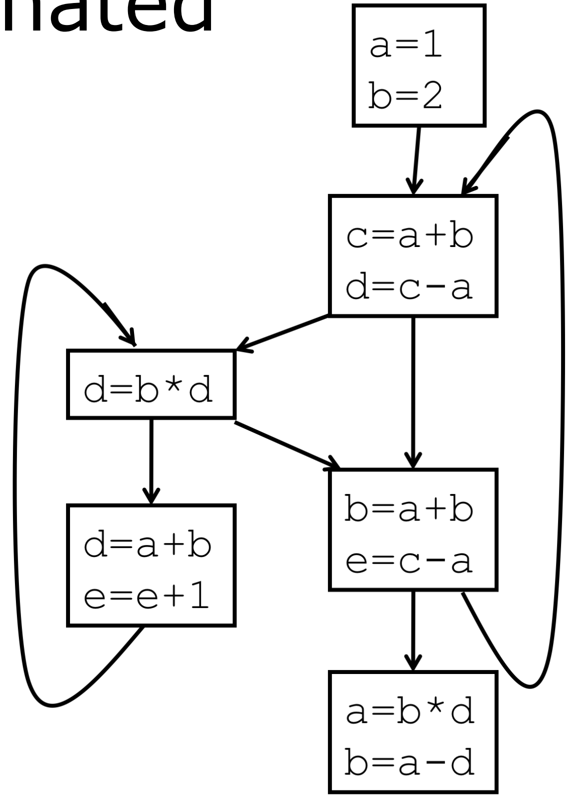
reuse
identical
expression
if
dominated

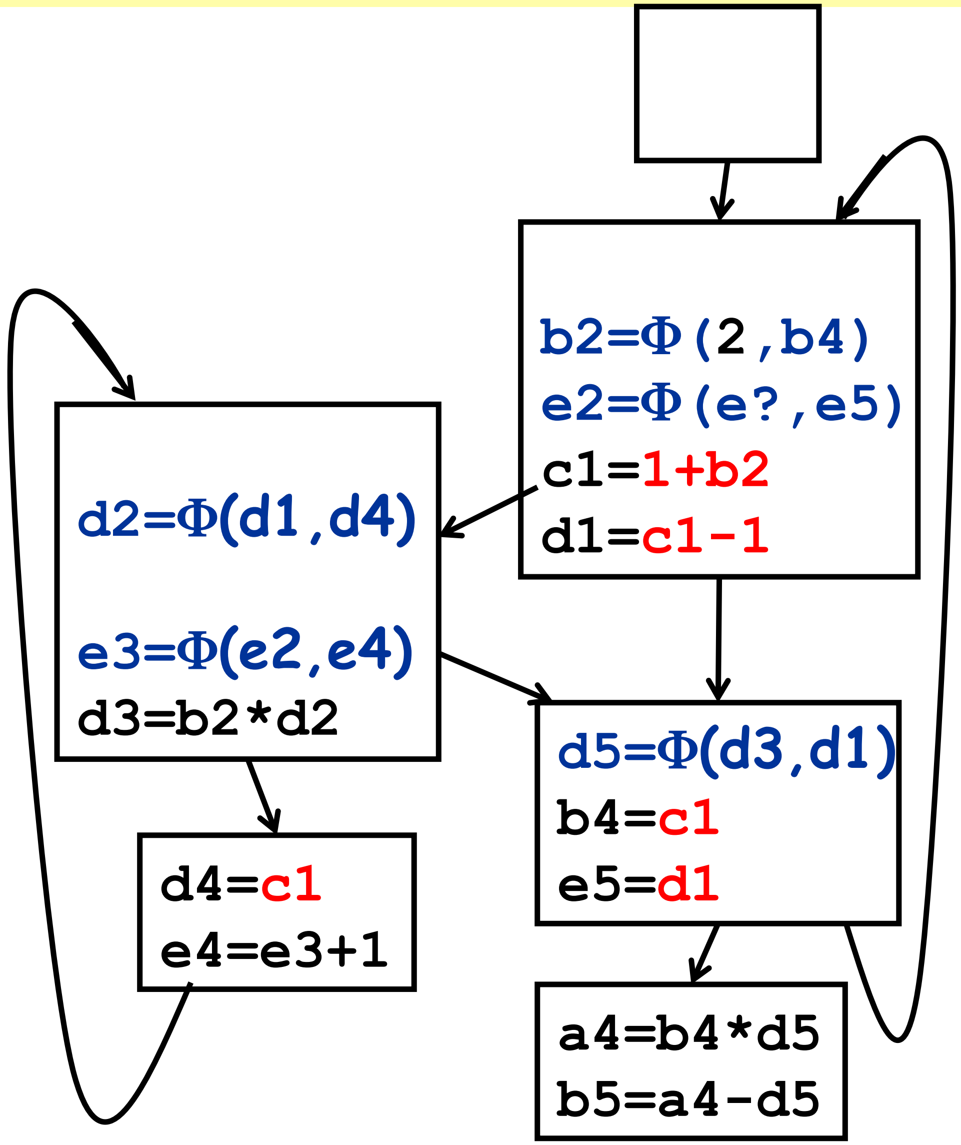




SSA Opt: CSE

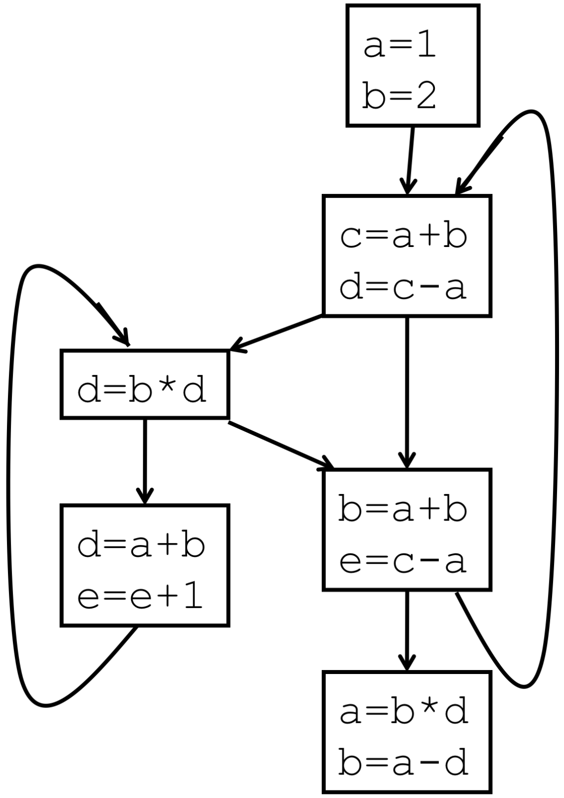
reuse
 identical
 expression
 if
 dominated

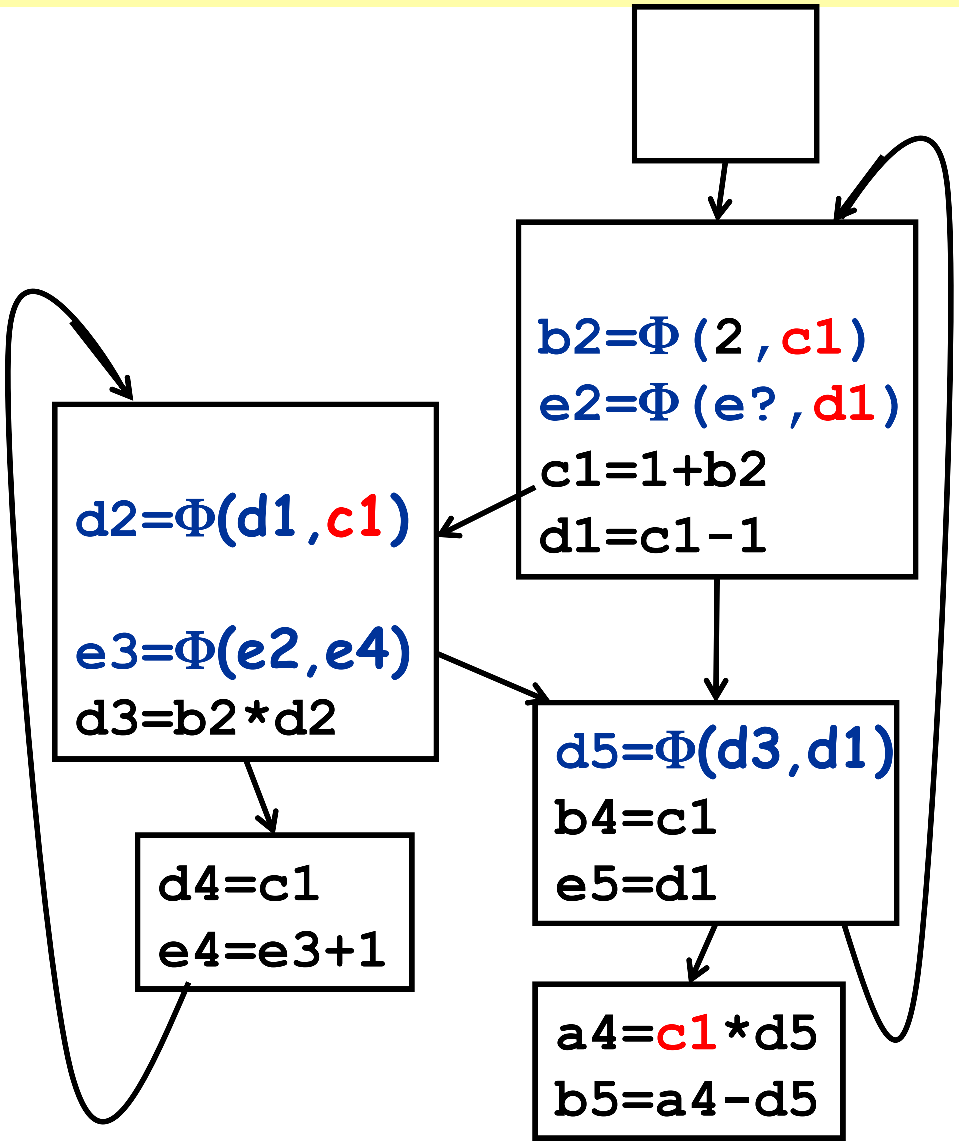




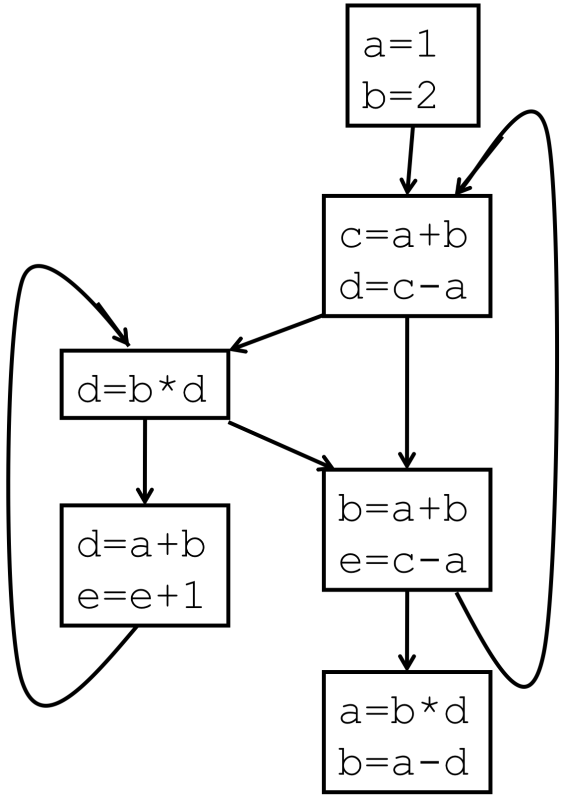
SSA Opt: copy propagate

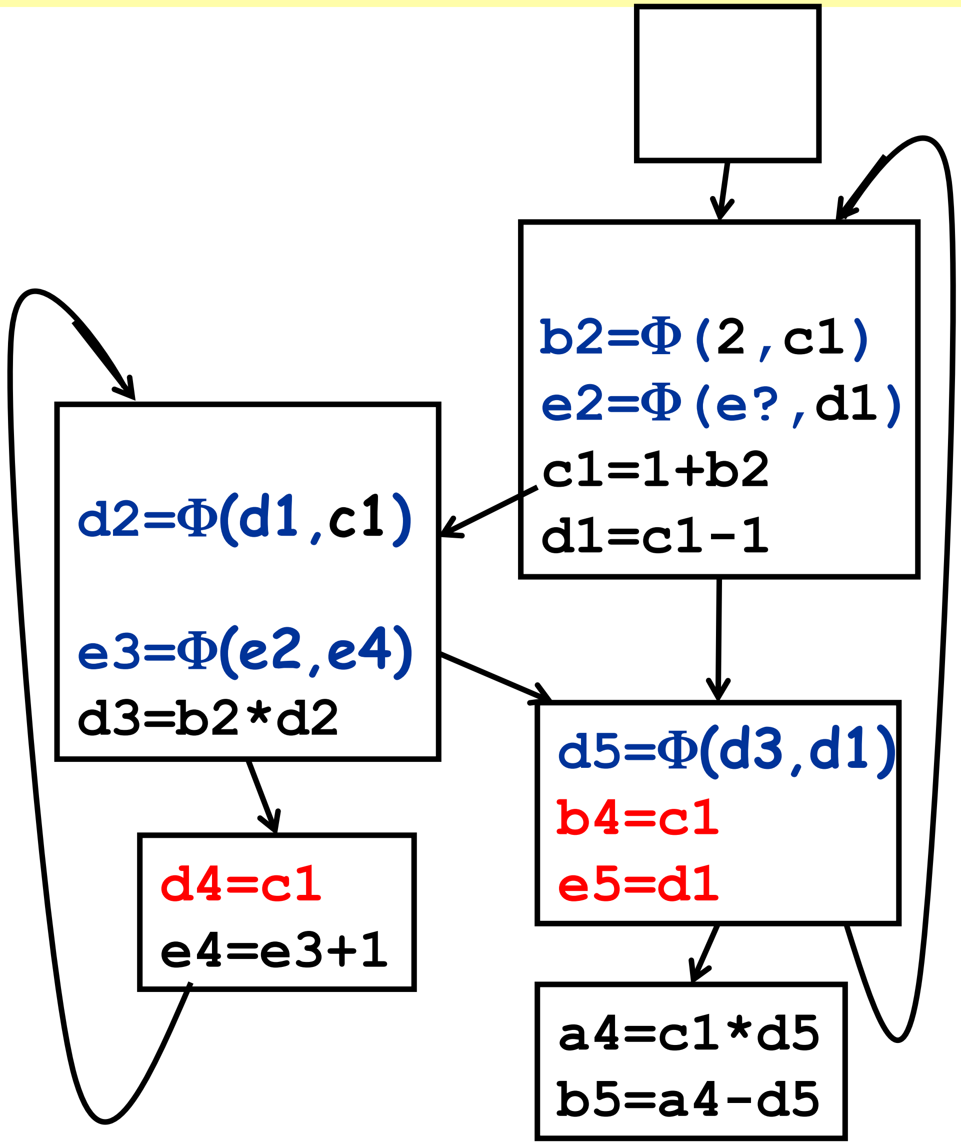
c1
=(1+b2)-1
=(1-1)+b2
=0+b2
=b2
good?



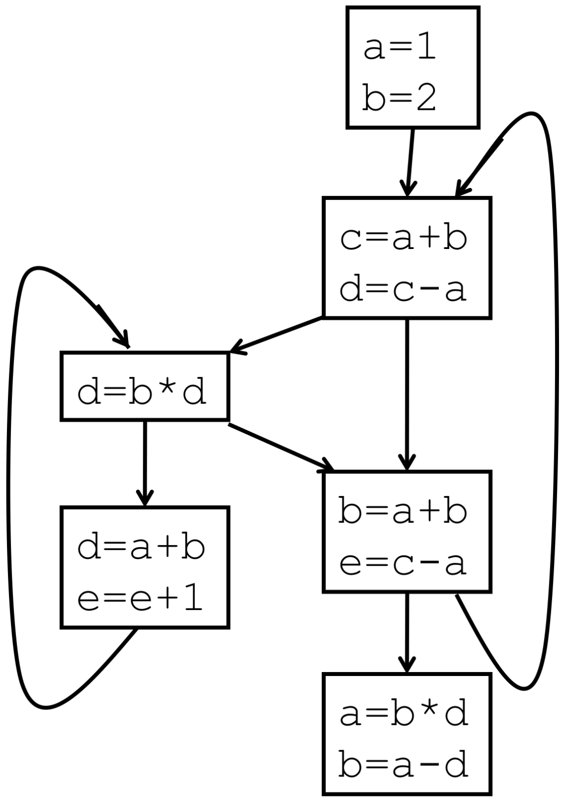


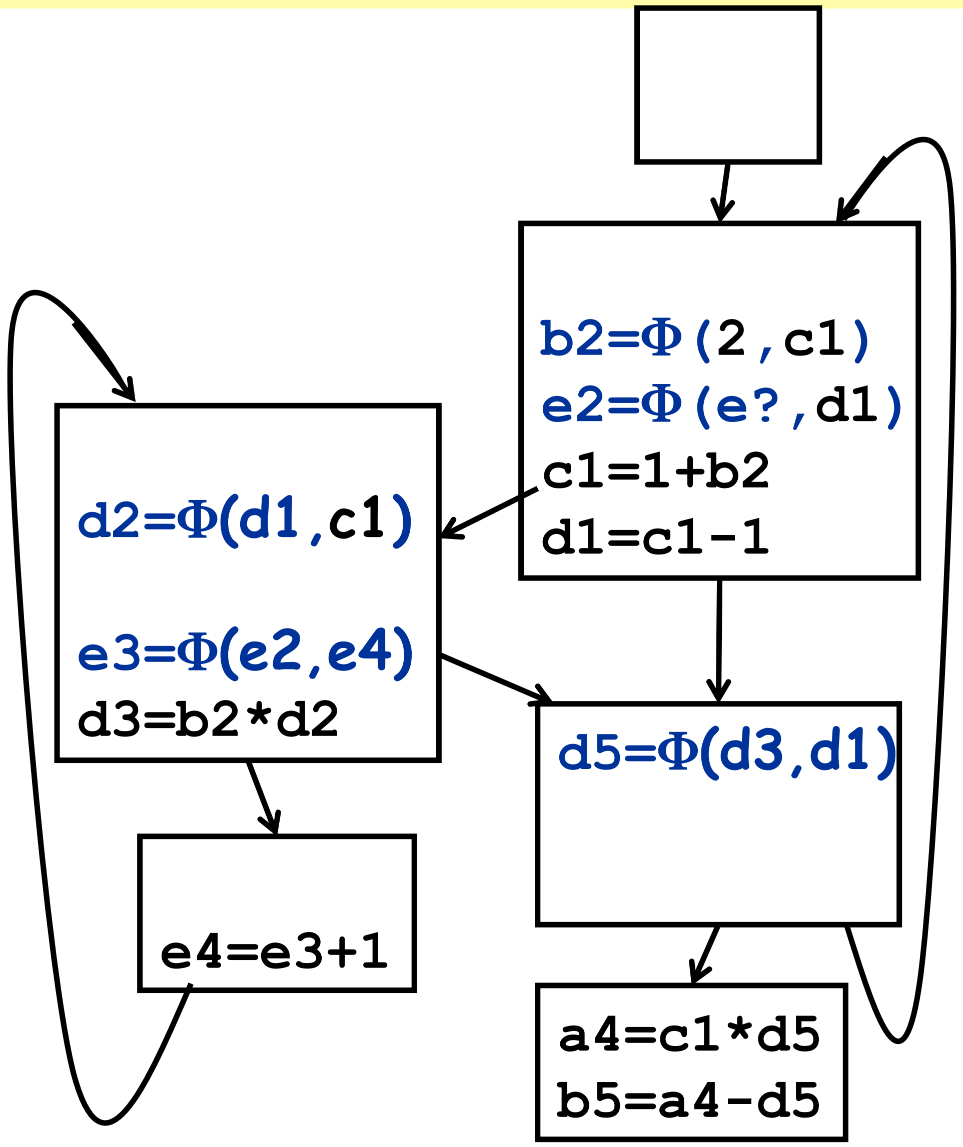
SSA Opt: eliminate dead code





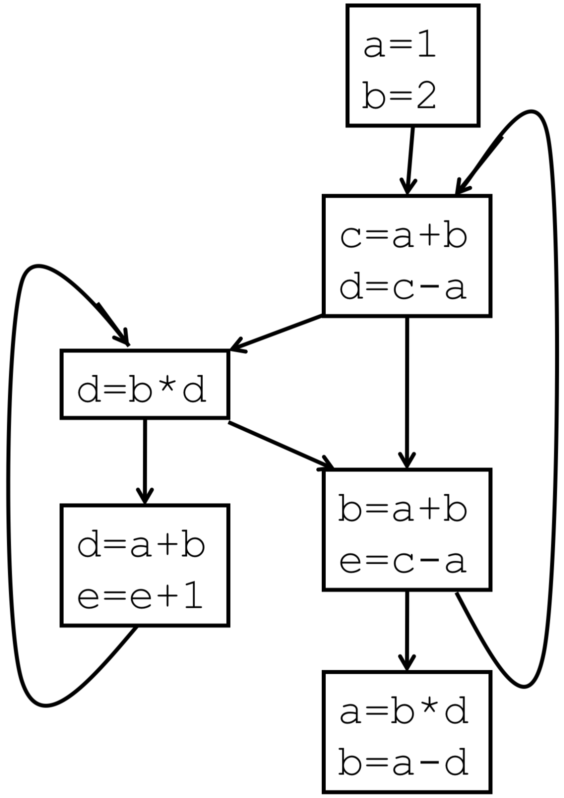
SSA Opt: eliminate dead code

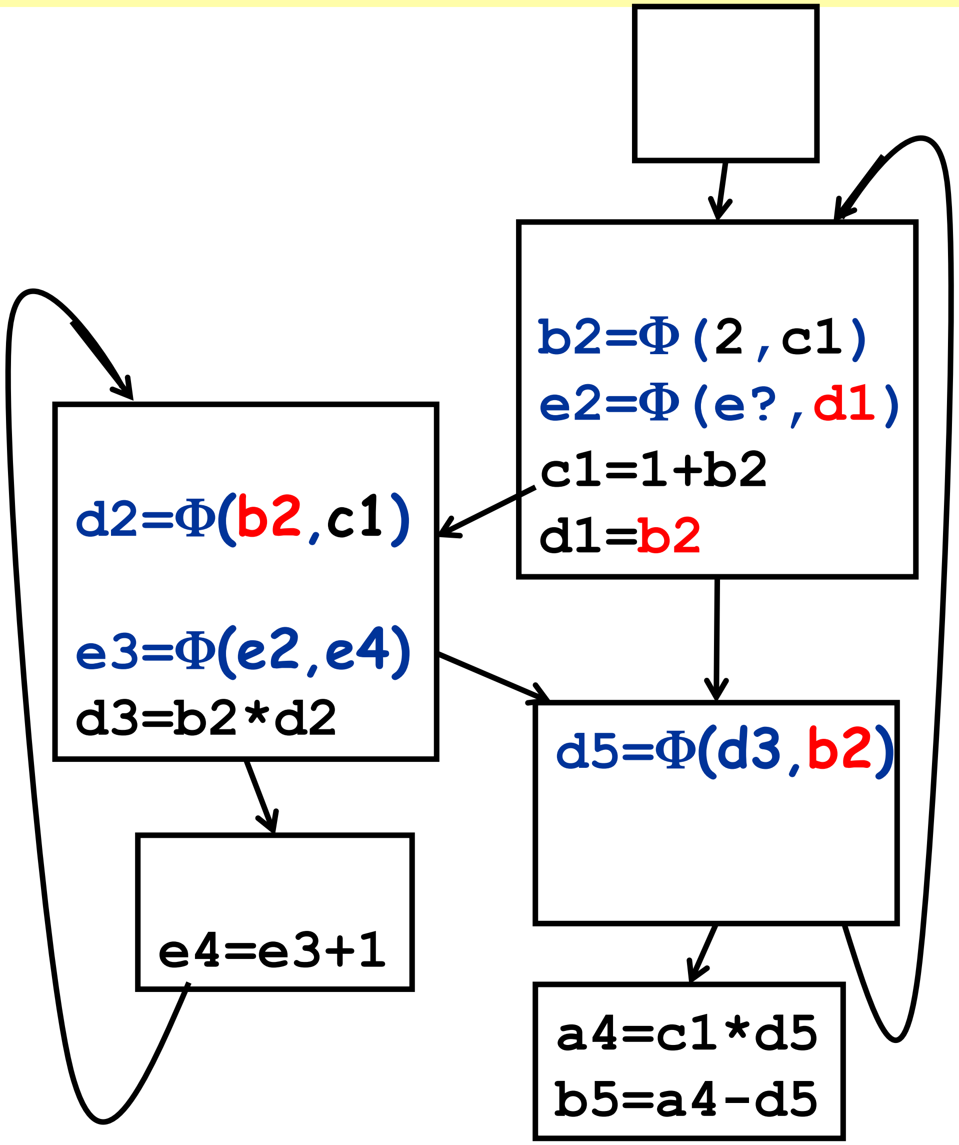




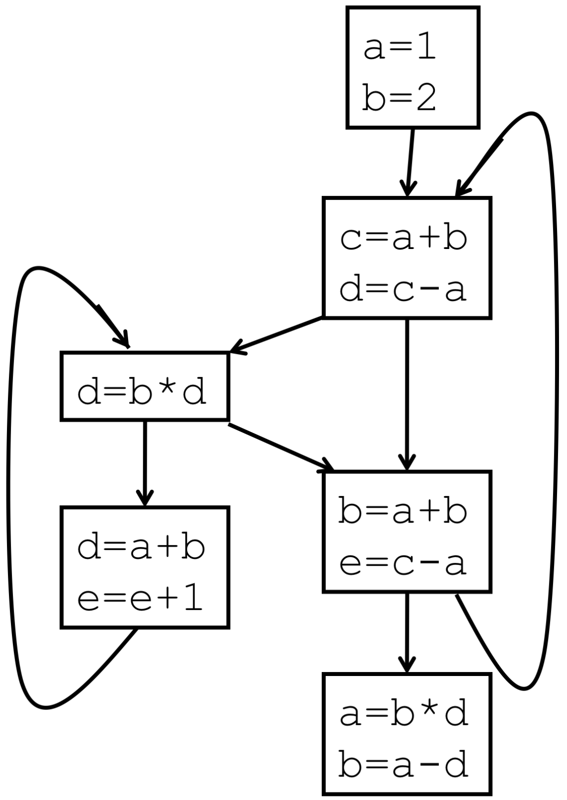
SSA Opt: arithmetic

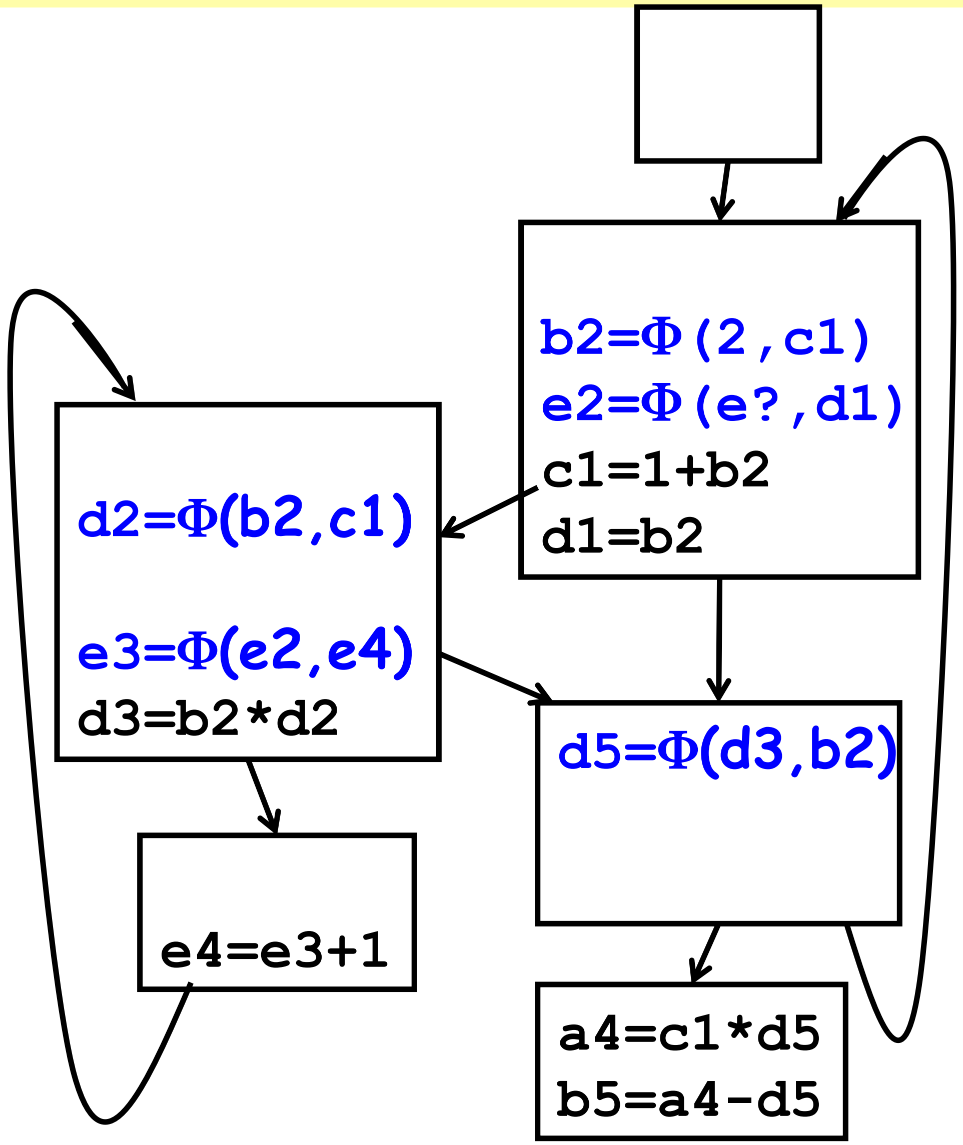
c1
=(1+b2)-1
=(1-1)+b2
=0+b2
=b2
good?



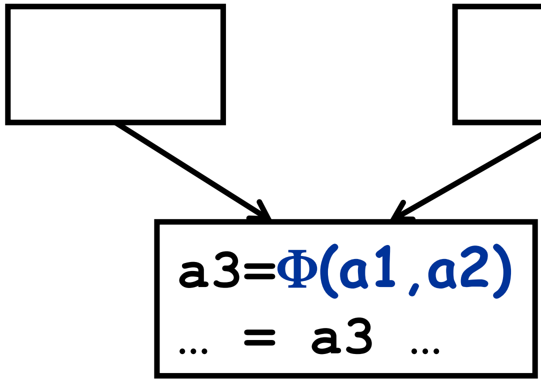


SSA Optimized



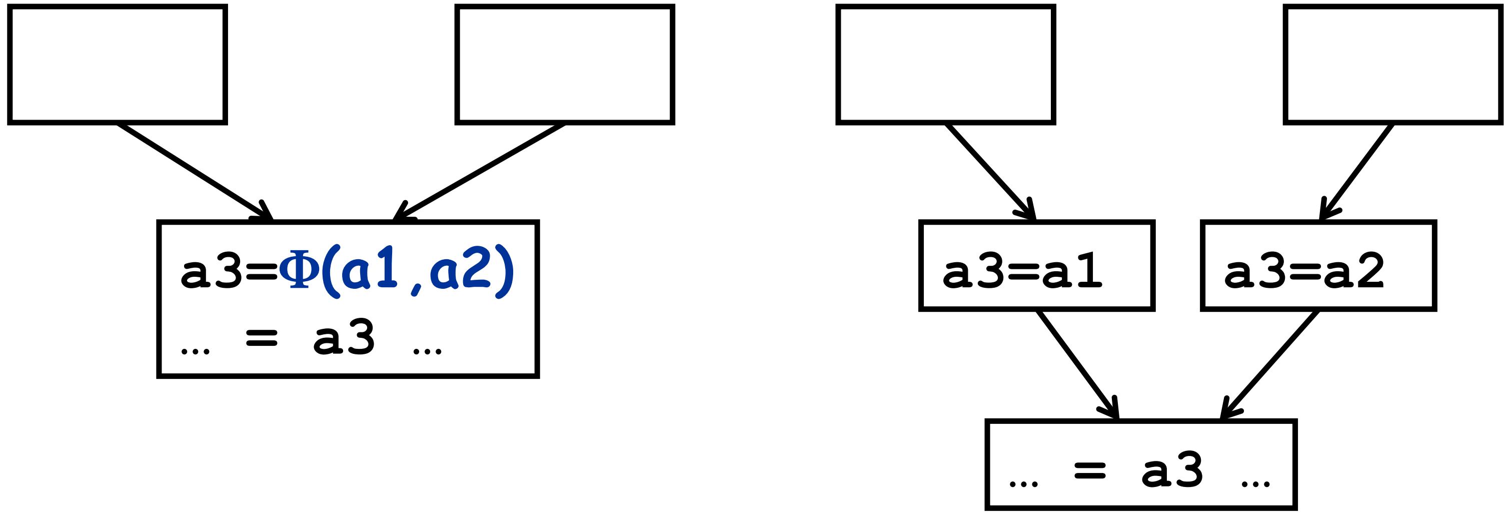


DeSSA: get rid of Φ easy way



How to get rid of Φ the easy way?

DeSSA: get rid of Φ easy way



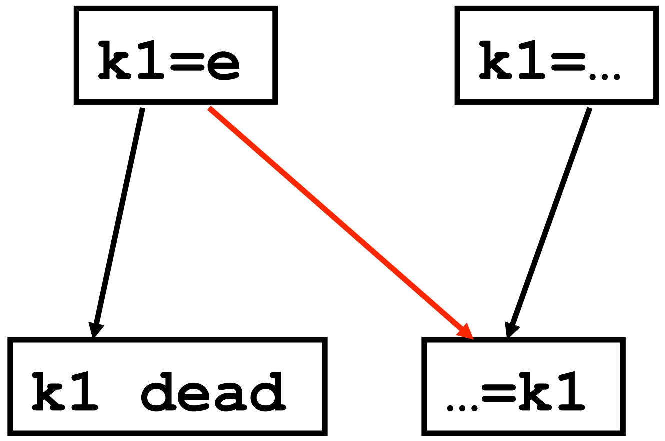
Just case split

Then do copy propagation again!

Register allocation merges var range

Fancy: critical edges

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

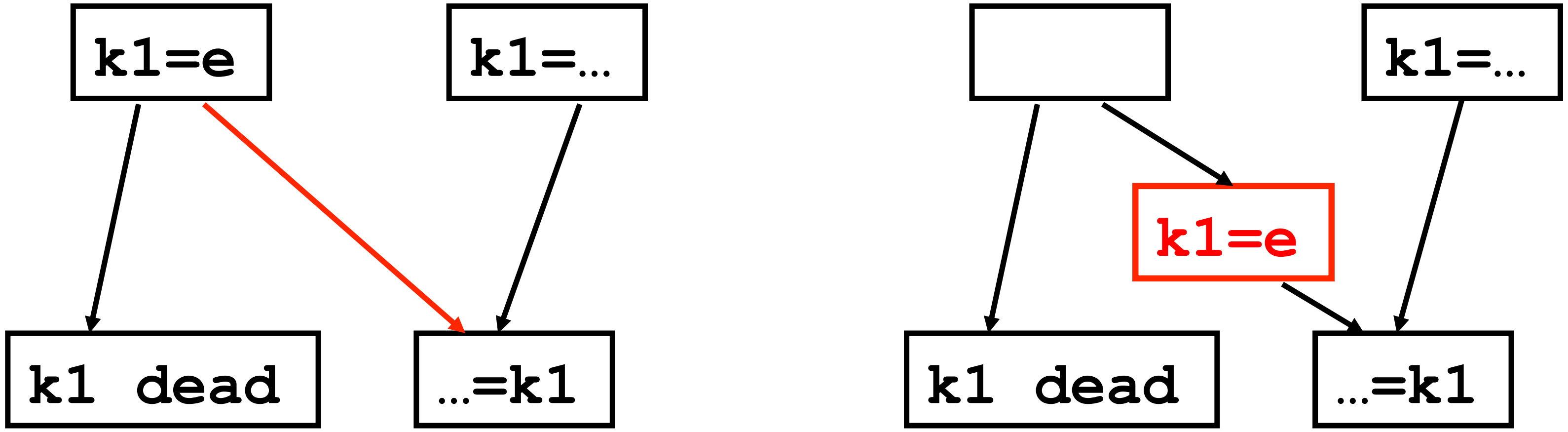


Critical edge makes optimal placement of k1=e assignment impossible

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.

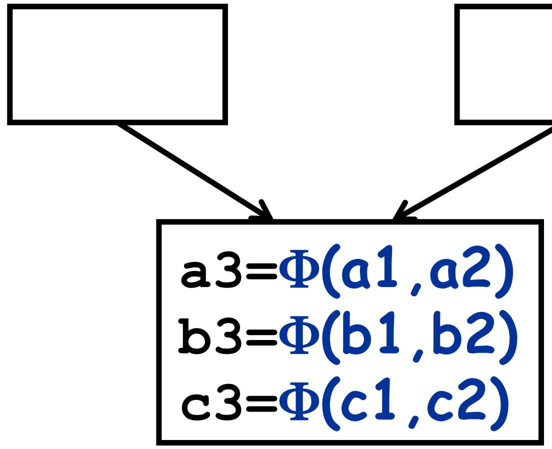


Critical edge makes optimal placement of k1=e assignment impossible

k1=e assignment is unnecessary for left succ but incorrect for right

Solution: add block on critical edge

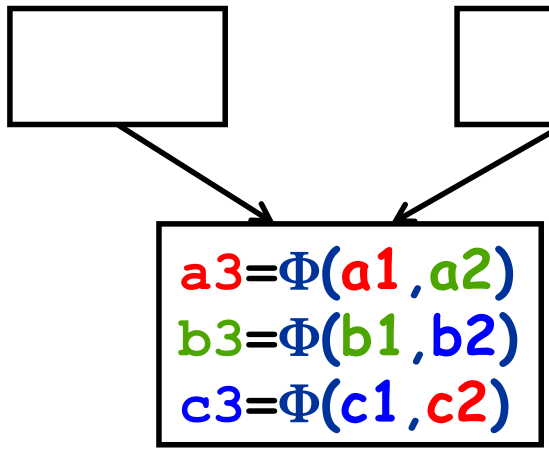
DeSSA: get rid of Φ elegant way



all Φ by parallel copy of argument i

a3,b3,c3 = ai,bi,ci

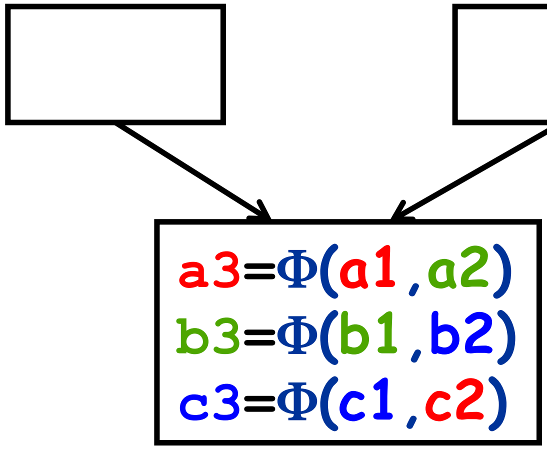
DeSSA: get rid of Φ fancy way



each edge i is permutation on regs

- implementable?

DeSSA: get rid of Φ fancy way



each edge i is permutation on regs

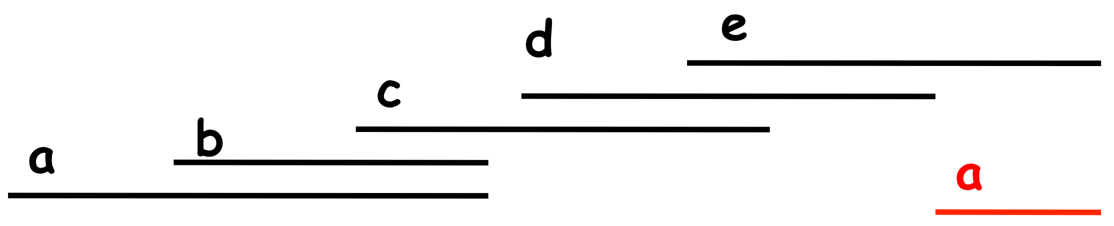
- implementable with a temp register
- impl by series of triple-xor swaps
- x=x^y;   y=x^y;   x=x^y

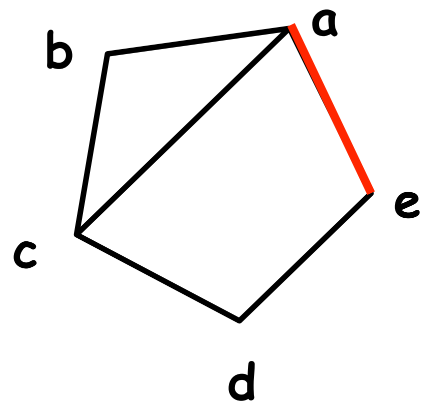
XOR swap

x=X	y=Y
x=X^Y	y=Y
y=X^Y	y=(X^Y)^Y=X (a&c)
y=(X^Y)^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. For that would need a live again. This violates SSA single assignment.

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