# Topic 15: Static Single Assignment

COS 320

Compiling Techniques

Princeton University
Spring 2016

Lennart Beringer

# Def-Use Chains, Use-Def Chains

Many optimizations need to find all use-sites of a definition, and/or all def-sites of a use:

- constant propagation needs the site of the unique reaching def
- copy propagation, common subexpression elimination,…

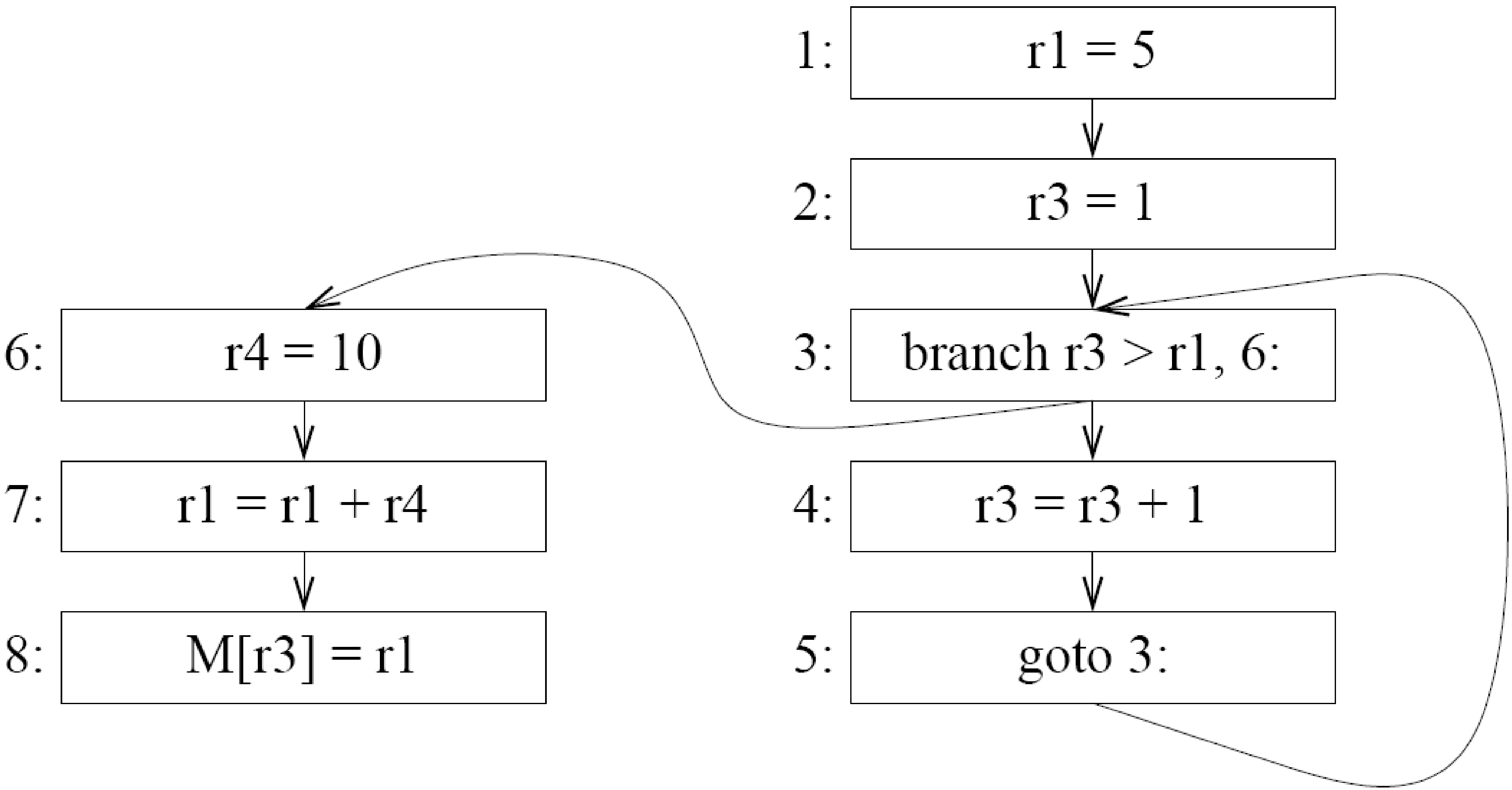
Data structures supporting these lookups:

- def-use chain: for each definition d of variable **r**, store the use sites of **r** that d reaches

- use-def chain: for each use site u of variable **r**, store the def-sites of **r** that reach u

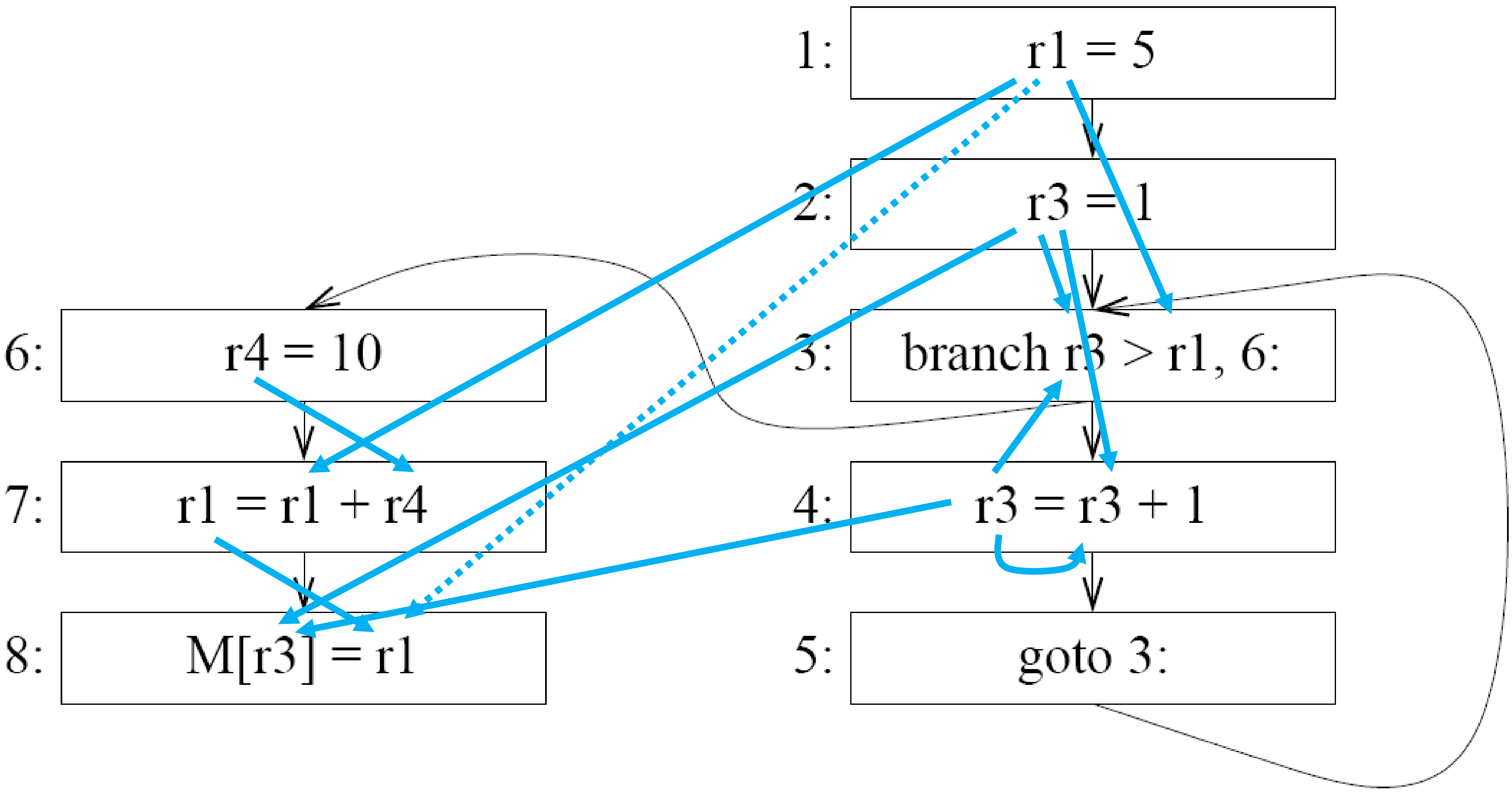
**N** definitions, **M** uses: 2***N*****M** relationships

# Use-Def Chains, Def-Use Chains




Add the def-use relationships…

# Use-Def Chains, Def-Use Chains

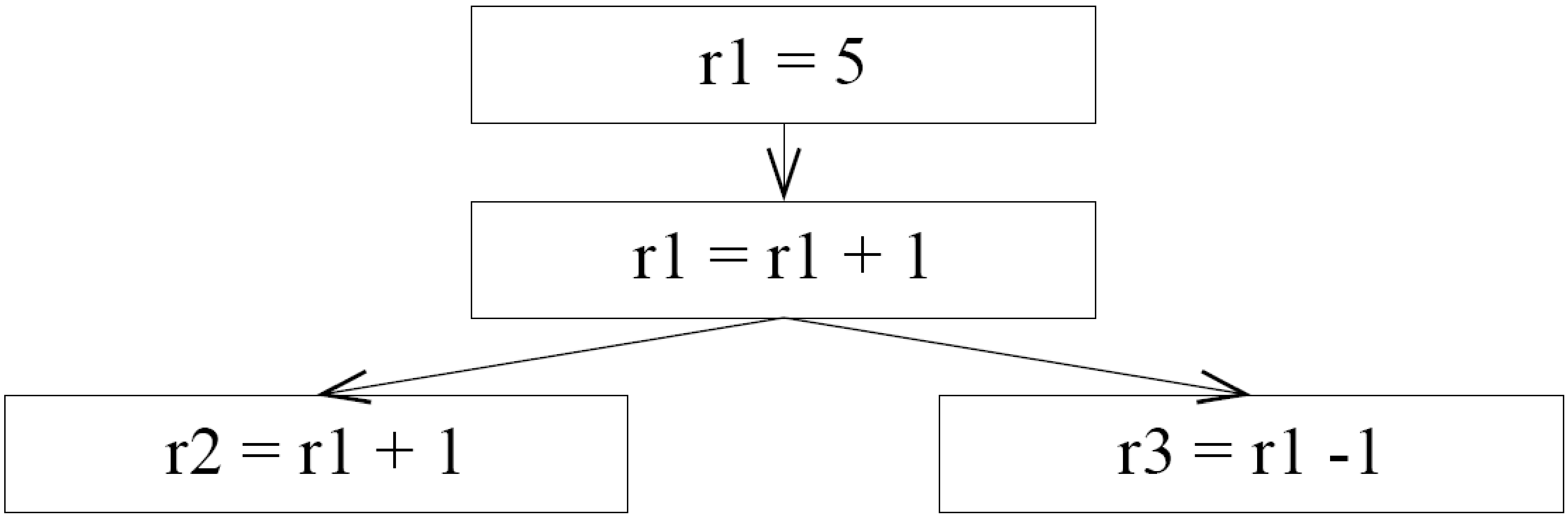



And these are just the def-use relationships…

# Static Single Assignment

**Static Single Assignment (SSA):**

- improvement on def-use chains
- each register has only one definition in program
- for each use $u$ of $r$, only one definition of $r$ reaches $u$

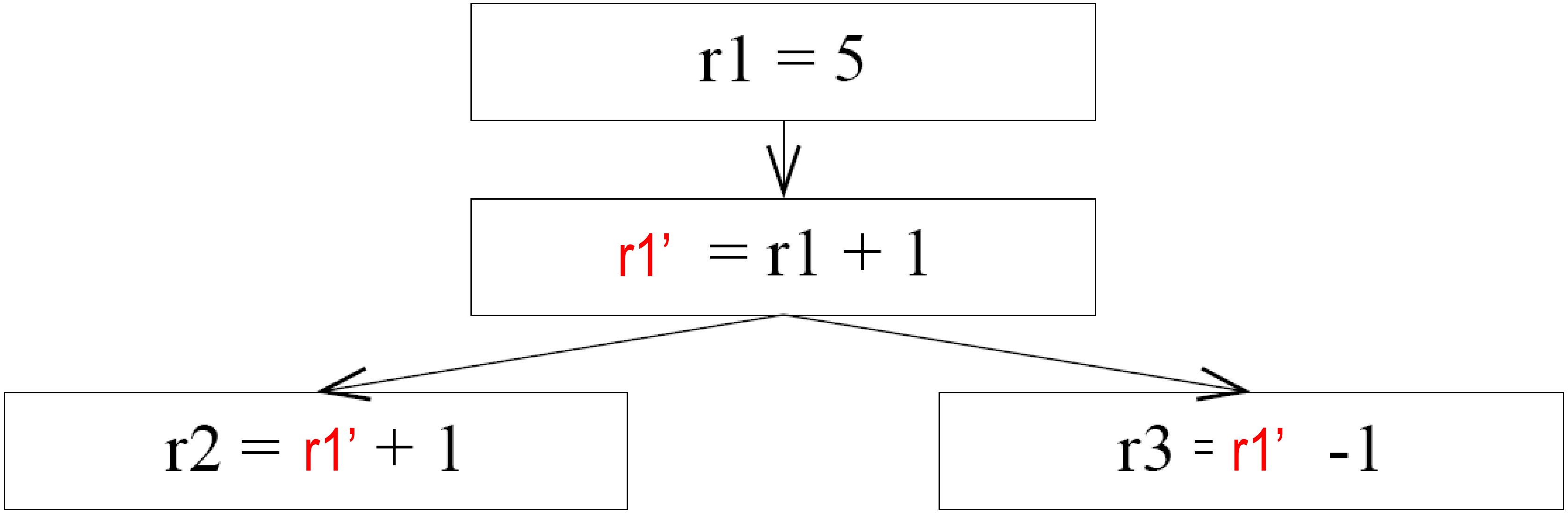



How can this be achieved?

# Static Single Assignment

**Static Single Assignment (SSA):**

- improvement on def-use chains
- each register has only one definition in program
- for each use $u$ of $r$, only one definition of $r$ reaches $u$




Rename variables consistently between defs and uses.

# Why SSA?

**Static Single Assignment Advantages:**

- Dataflow analysis and code optimization made simpler.
  - Variables have only one definition - no ambiguity.
  - Dominator information is encoded in the assignments.
- Less space required to represent def-use chains. For each variable, space is proportional to uses * defs. Distinguishing different defs makes use lists shorter and more precise: less overlap.
- Eliminates unnecessary relationships:

```
for i = 1 to N do A[i] = 0
for i = 1 to M do B[i] = 1
```

  - No reason why both loops should be forced to use same register to hold index register.
  - SSA renames second `i` to new register which may lead to better register allocation/optimization.

(Dynamic Single Assignment is also proposed in the literature.)

# Conversion to SSA Code

**Easy to convert basic blocks into SSA form:**

- Each definition modified to define brand-new register, instead of redefining old one.
- Each use of register modified to use most recently defined version.

```
r1 = r3 + r4

r2 = r1 - 1

r1 = r4 + r2

r2 = r5 * 4

r1 = r1 + r2
```

# Conversion to SSA Code

**Easy to convert basic blocks into SSA form:**

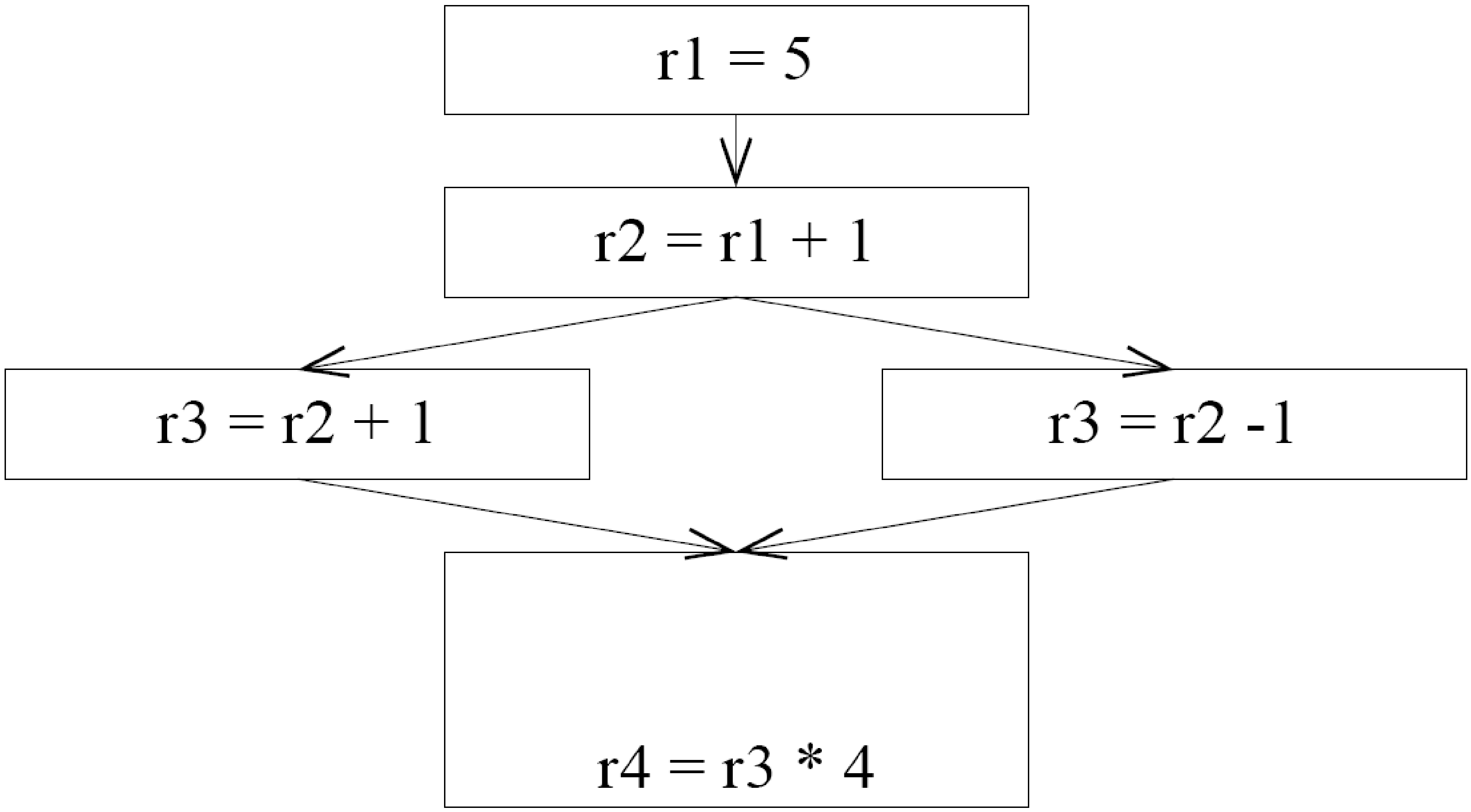
- Each definition modified to define brand-new register, instead of redefining old one.
- Each use of register modified to use most recently defined version.

```
r1 = r3 + r4
r2 = r1 - 1
r1 = r4 + r2
r2 = r5 * 4
r1 = r1 + r2
```

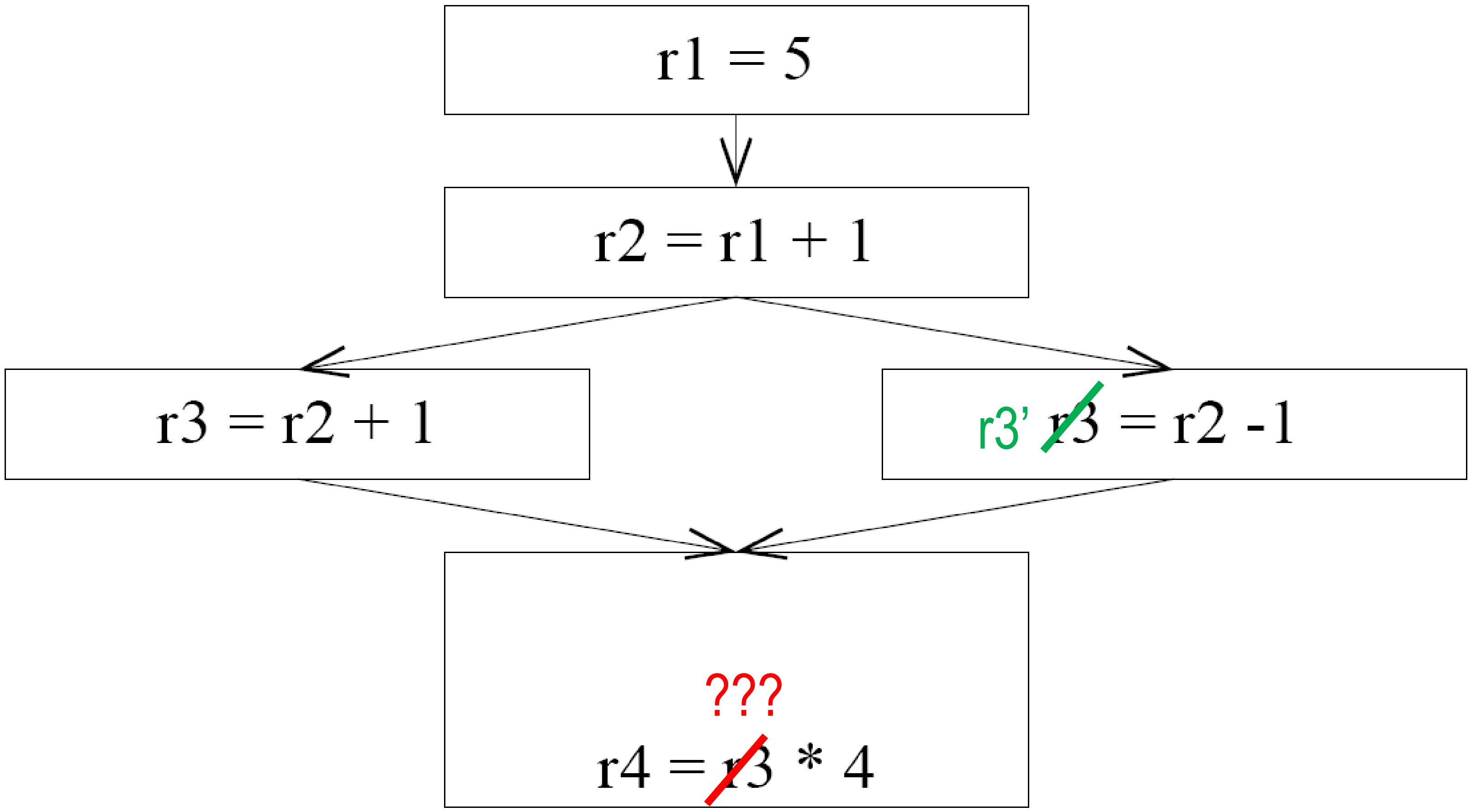
```
r1 = r3 + r4
r2 = r1 – 1
r1' = r4 + r2
r2' = r5 * 4
r1'' = r1' + r2'
```

**Control flow introduces problems.**

# Conversion to SSA Form

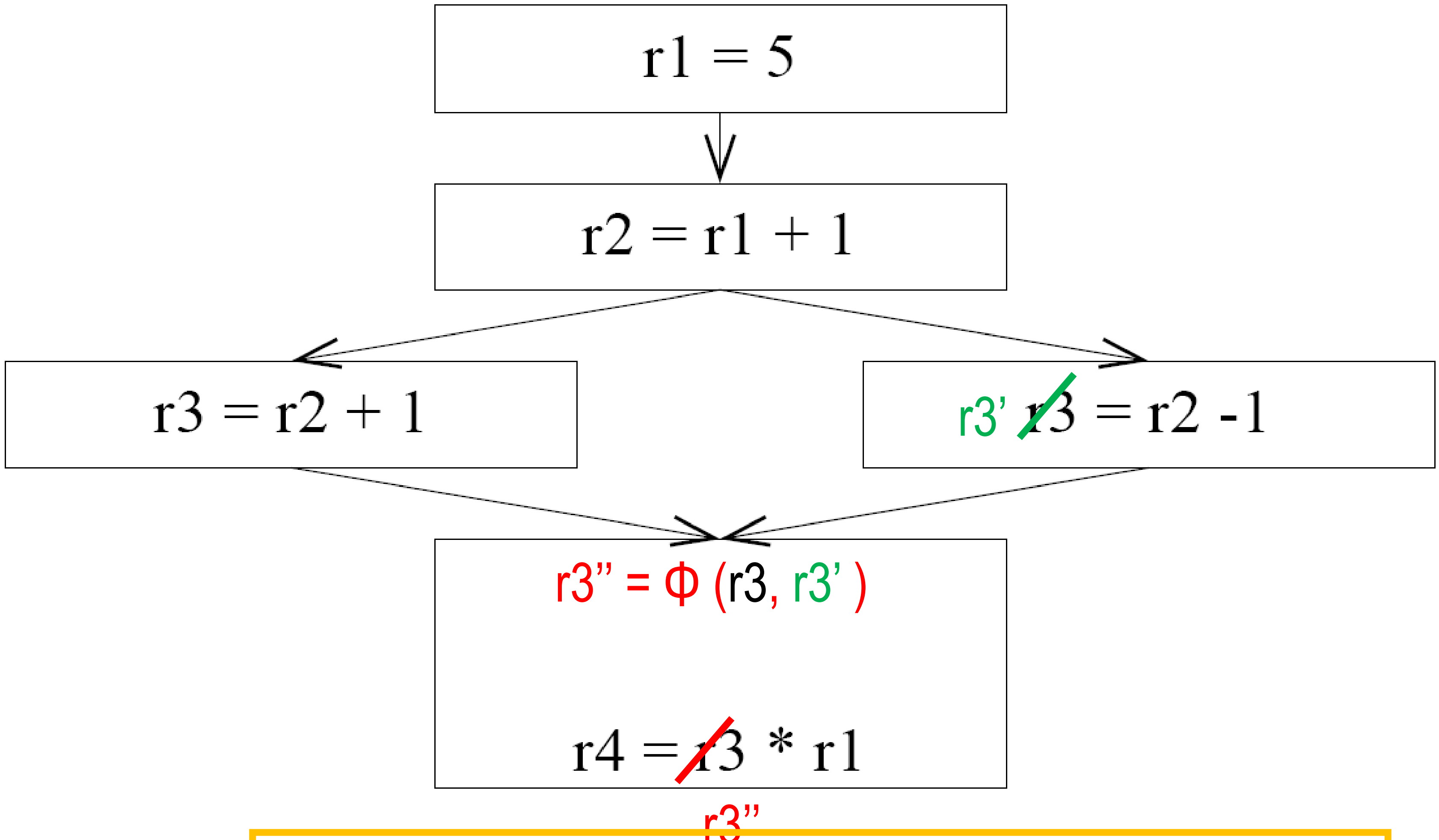


# Conversion to SSA Form




**Use $\phi$ functions.**

# Conversion to SSA Form




$r3'' = \phi(r3, r3')$: – $r3'' = r3$ if control enters from left
– $r3'' = r3'$ if control enters from right

# Conversion to SSA Form

- $\phi$-functions enable the use of r3 to be reached by exactly one definition of r3.
- Can implement $\phi$-functions as set of move operations on each incoming edge.

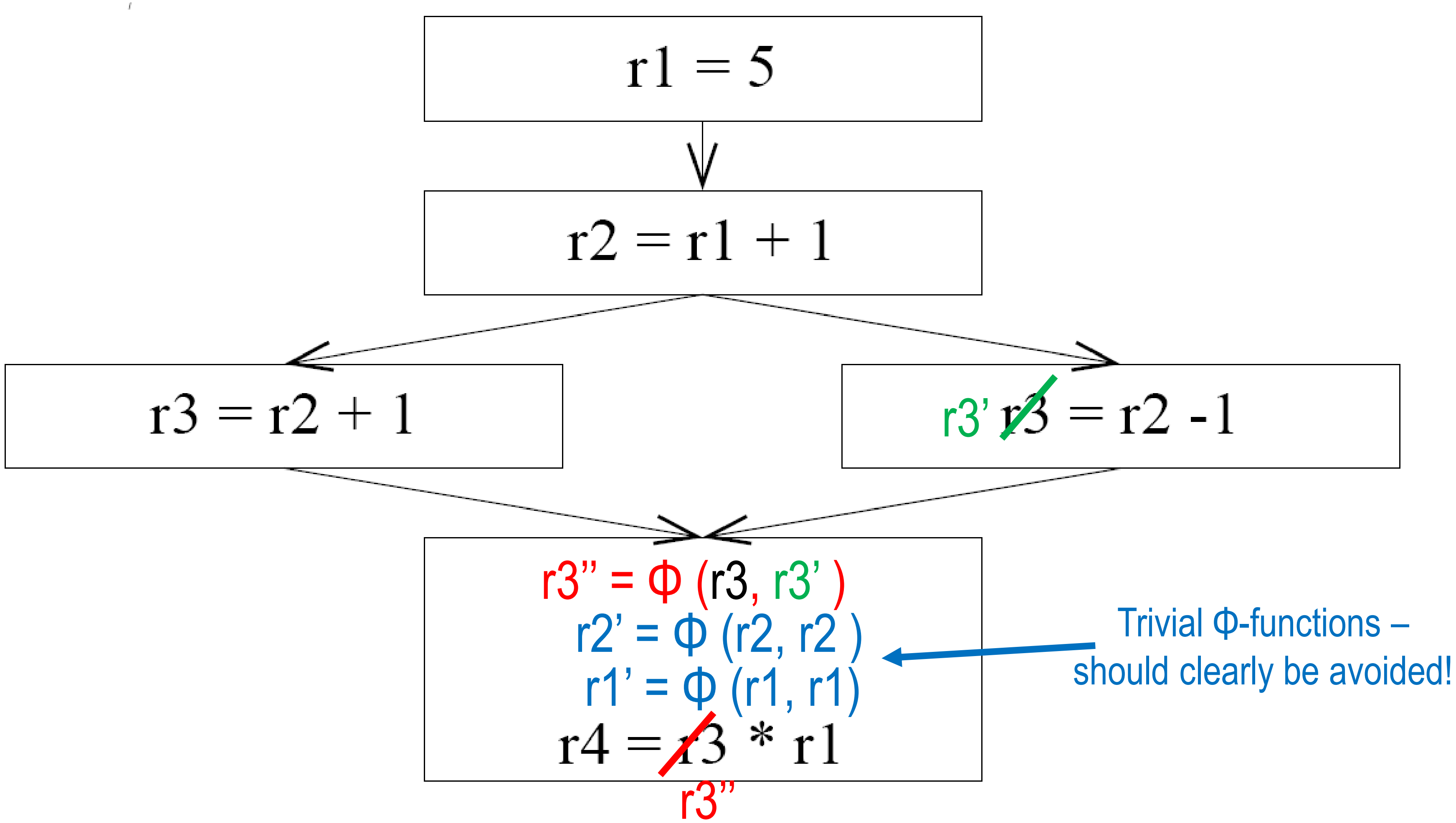
- for analysis & optimization: no implementation necessary: 
  Φ just used as notation
- left side of Φ-function constitutes a definition; variables in RHS are uses
- ordering of argument positions corresponds to (arbitrary) order of incoming control flow arcs, but left implicit (could name positions using the labels of predecessor basic blocks…)

- elimination of Φ-functions/translation out-of-SSA: 
  insert move instructions; often coalesced during register allocation
- typically, basic blocks have several Φ-functions – all near the top, with identical ordering of incomings arcs from control flow predecessors

# Conversion to SSA Form

Naïve insertion:

add a Φ-function for each register at each node with ≥2 predecessors




Can we do better?

# Conversion to SSA Form

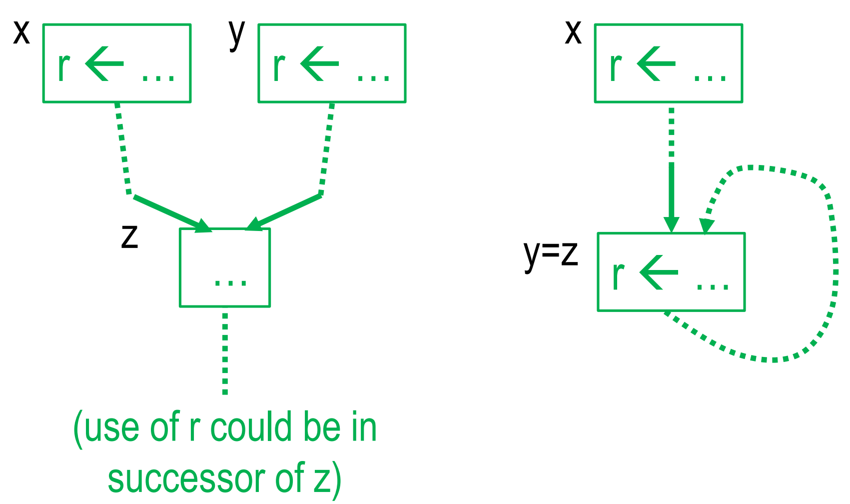
**Path-Convergence Criterion**: Insert a $\phi$-function for a register $r$ at node $z$ of the flow graph if ALL of the following are true:

1. There is a block $x$ containing a definition of $r$.
2. There is a block $y \neq x$ containing a definition of $r$.
3. There is a non-empty path $P_{xz}$ of edges from $x$ to $z$.
4. There is a non-empty path $P_{yz}$ of edges from $y$ to $z$.
5. Paths $P_{xz}$ and $P_{yz}$ do not have any node in common other than $z$.
6. The node $z$ does not appear within both $P_{xz}$ and $P_{yz}$ prior to the end, though it may appear in one or the other. (eg if y=z)

Assume CFG entry node contains implicit definition of each register:

- $r$ = actual parameter value
- $r$ = undefined

$\phi$-functions are counted as definitions.



# Conversion to SSA Form

Solve path-convergence iteratively:

WHILE (there are nodes $x$, $y$, $z$ satisfying conditions 1-6) &&
($z$ does not contain a $phi$-function for $r$) DO:
insert $r = \phi(r, r, ..., r)$ (one per predecessor) at node $z$.

- Costly to compute. (3 nested loops, for x, y, z)
- Since definitions dominate uses, use domination to simplify computation.

# Conversion to SSA Form

Solve path-convergence iteratively:

WHILE (there are nodes $x$, $y$, $z$ satisfying conditions 1-6) &&
($z$ does not contain a $phi$-function for $r$) DO:
insert $r = \phi(r, r, ..., r)$ (one per predecessor) at node $z$.

- Costly to compute. (3 nested loops, for x, y, z)
- Since definitions dominate uses, use domination to simplify computation.

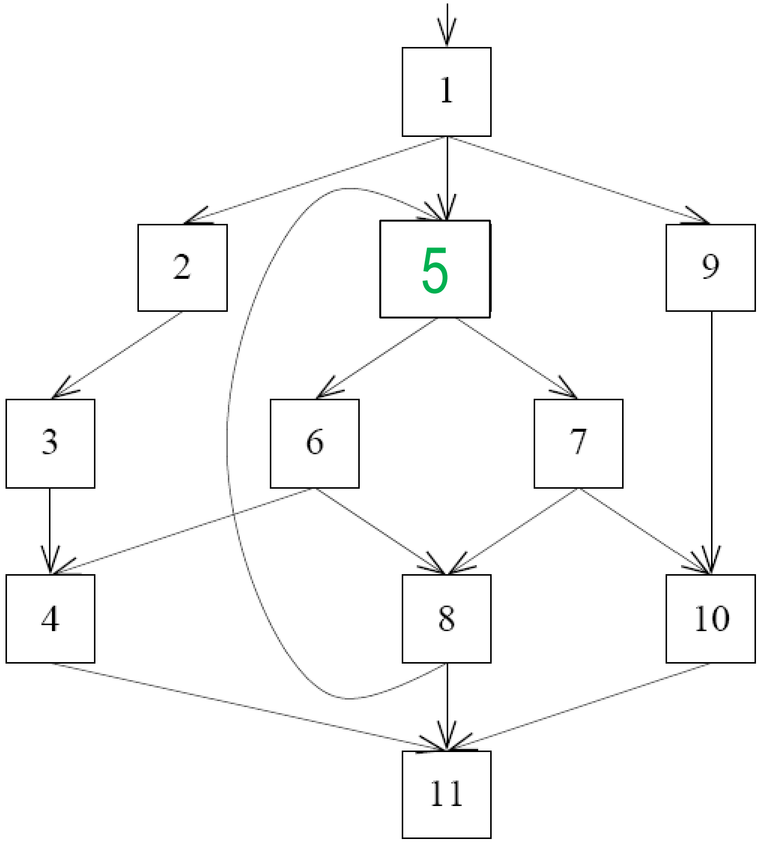
**Use *Dominance Frontier*...**

Remember **dominance**: node **x** dominates node **w** if every path from **entry** to **w** goes through **x**. (In particular, every node dominates itself)

# Dominance Frontier

**Definitions:**

- $x$ *strictly dominates* $w$ if $x$ dominates $w$ and $x \neq w$.
- *dominance frontier* of node $x$ is set of all nodes $w$ such that $x$ dominates a predecessor of $w$, but does not strictly dominate $w$.

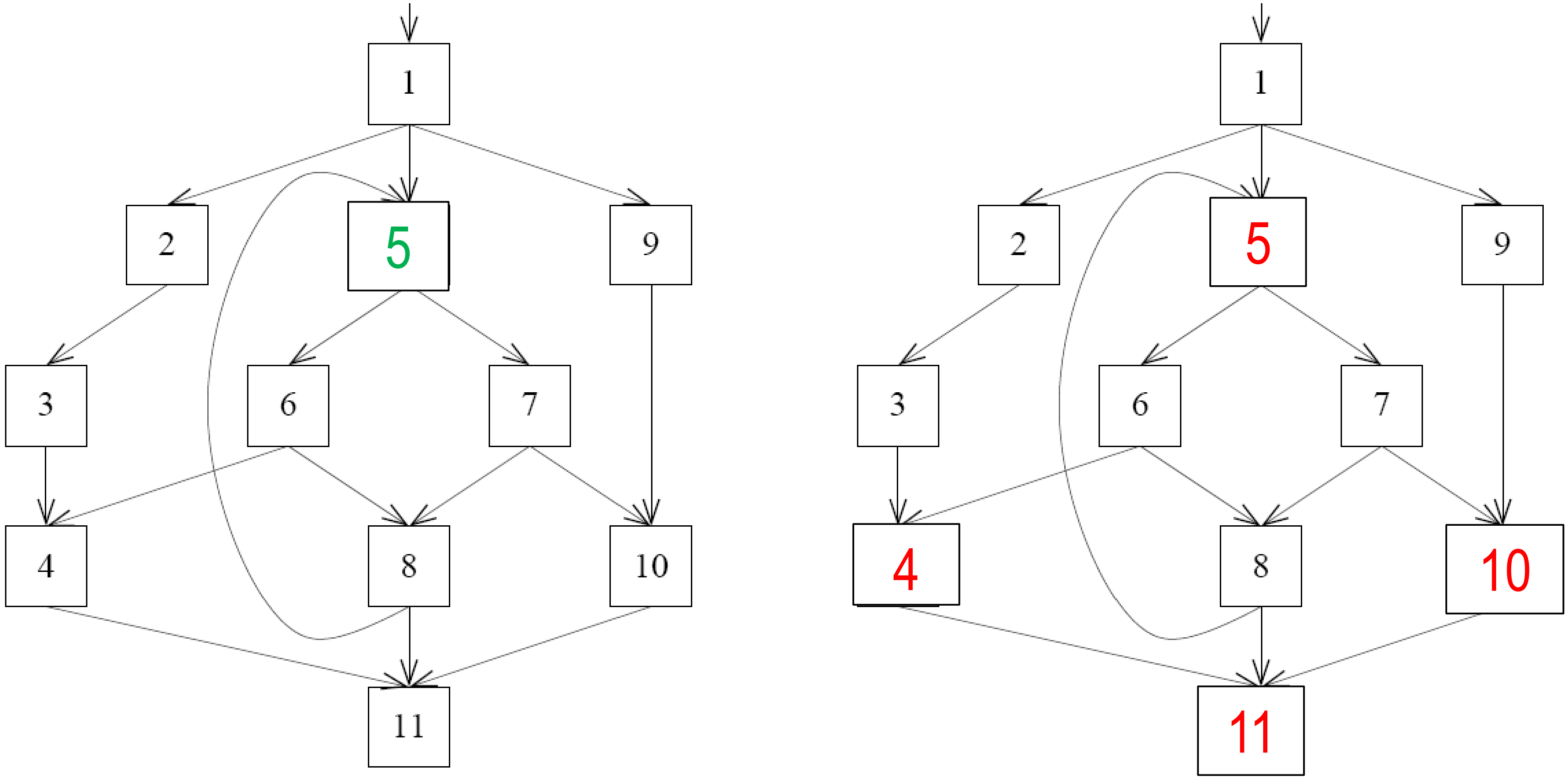



DF(5) = ?

# Dominance Frontier

**Definitions:**

- $x$ *strictly dominates* $w$ if $x$ dominates $w$ and $x \neq w$.
- *dominance frontier* of node x is set of all nodes w such that x dominates a predecessor of w, but does not strictly dominate w.




DF(5) = {4, 5, 10, 11}

# Dominance Frontier

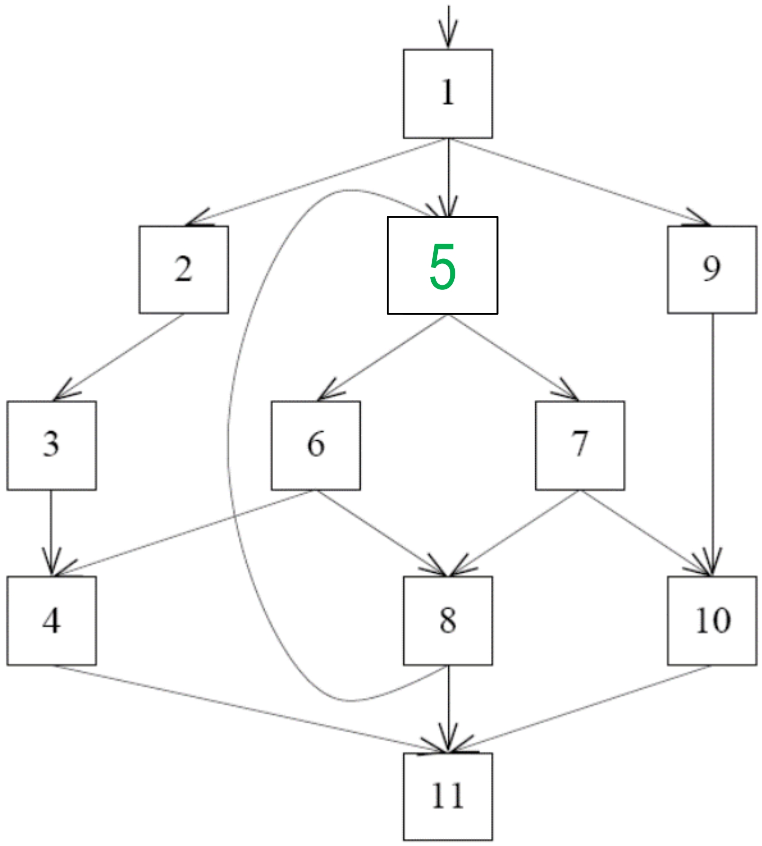
**Dominance Frontier Criterion:**

Whenever node **x** contains a **definition** of a register **r**, insert a Φ-function for **r** in all nodes **z** ϵ **DF**(**x**).

**Iterated Dominance Frontier Criterion:**

Apply dominance frontier condition repeatedly, to account for the fact that Φ-functions constitute definitions themselves.

Suppose **5** contains a definition of **r.**



# Dominance Frontier

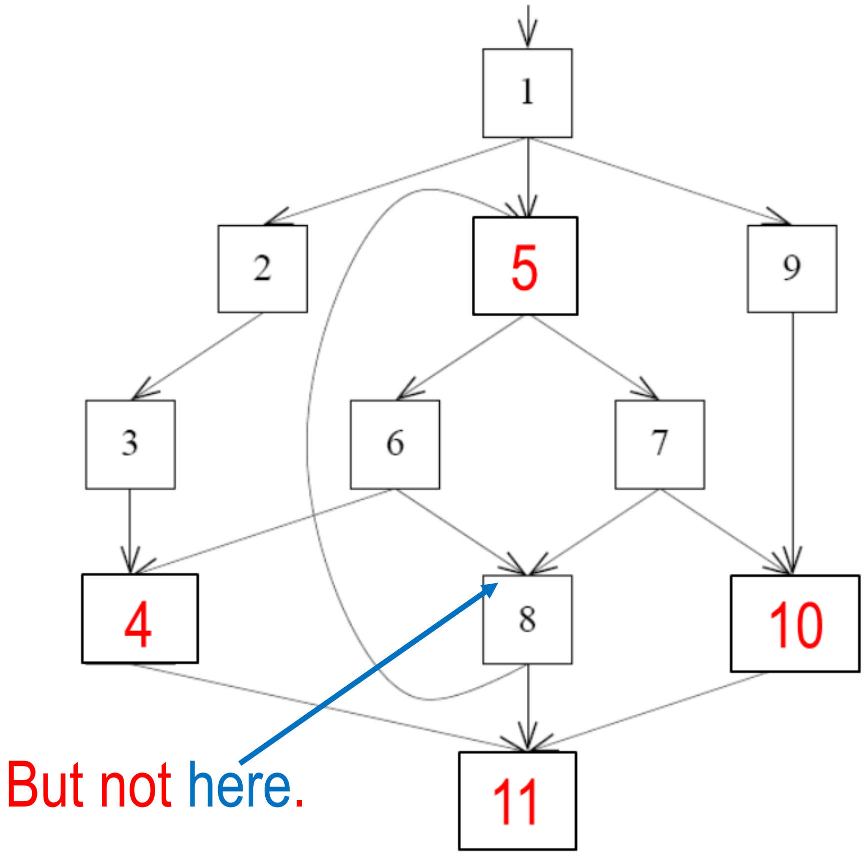
**Dominance Frontier Criterion:**

Whenever node **x** contains a **definition** of a register **r**, insert a Φ-function for **r** in all nodes **z** ϵ **DF**(**x**).

**Iterated Dominance Frontier Criterion:**

Apply dominance frontier condition repeatedly, to account for the fact that Φ-functions constitute definitions themselves.

Suppose **5** contains a definition of **r**.
Insert Φ-functions for **r** in red blocks.



# Dominance Frontier Computation

- Use dominator tree
- $DF[n]$: dominance frontier of $n$
- $DF_{local}[n]$: successors of $n$ in CFG that are not strictly dominated by $n$
- $DF_{up}[c]$: nodes in dominance frontier of $c$ that are not strictly dominated by $c$'s immediate dominator

See errata list of MCIML

Alternative formulation: $DF_{local}[n]$ = successors s of n with idom[s] <> n.

# Dominance Frontier Computation

- Use dominator tree
- $DF[n]$: dominance frontier of $n$
- $DF_{local}[n]$: successors of $n$ in CFG that are not strictly dominated by $n$
- $DF_{up}[c]$: nodes in dominance frontier of $c$ that are not strictly dominated by $c$'s immediate dominator

See errata list of MCIML

$$DF[n] = DF_{local}[n] \cup \left(\cup_{c \in children[n]} DF_{up}[c]\right)$$

- where $children[n]$ are the nodes whose idom is n.
- Work bottom up in dominator tree.

Leaf p satisfies DF[ p ] = $DF_{local}$[ p ] since children[p] = {}.

Alternative formulation: $DF_{local}$[n] = successors s of n with idom[s] <> n.

# Dominator Analysis (slide 22 from "Control Flow")

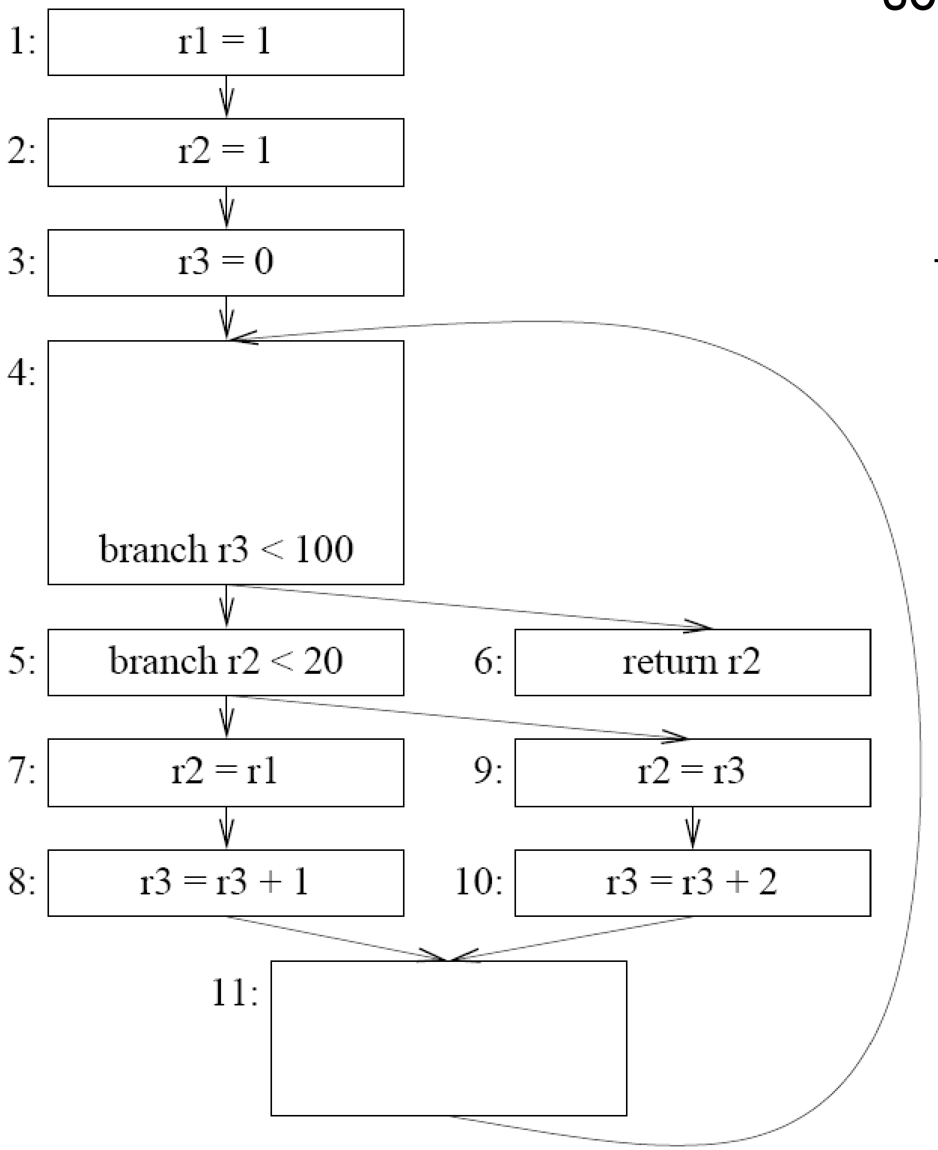
- If $d$ dominates each of the $p_i$, then $d$ dominates $n$.
- If $d$ dominates $n$, then $d$ dominates each of the $p_i$.
- $Dom[n]$ = set of nodes that dominate node $n$.
- $N$ = set of all nodes.
- Computation:

starting point: n dominated by all nodes

1. $Dom[s_0] = \{s_0\}$.
2. **for** $n \in N - \{s_0\}$ **do** $Dom[n] = N$
3. **while** (changes to any $Dom[n]$ occur) **do**
4.     **for** $n \in N - \{s_0\}$ **do**
5.         $Dom[n] = \{n\} \cup \left(\cap_{p \in pred[n]} Dom[p]\right)$.

nodes that dominate all predecessors of n

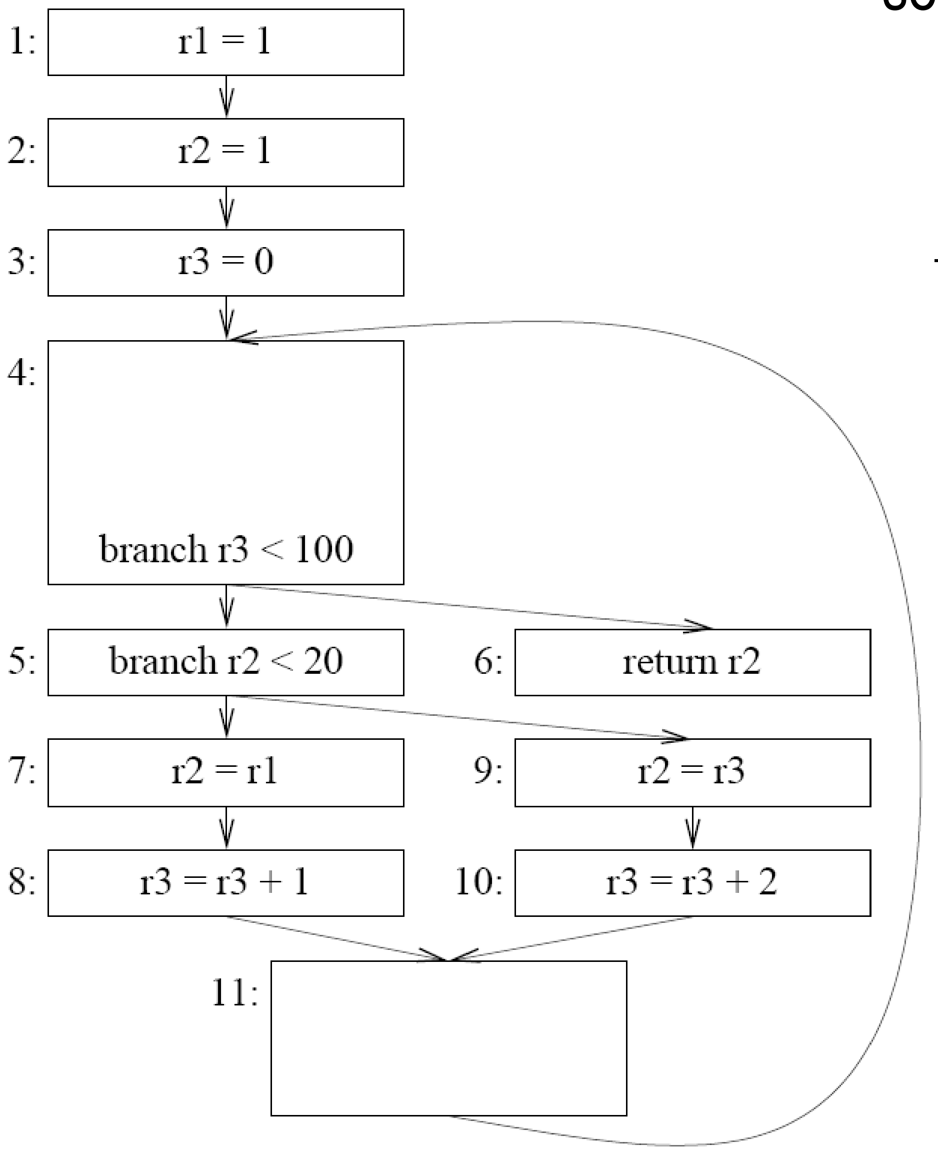
# SSA Example




set of nodes that dominate n

| Node | $DOM[n]$ | $IDOM[n]$ |
|---|---|---|
| 1 | | |
| 2 | | |
| 3 | | |
| 4 | | |
| 5 | | |
| 6 | | |
| 7 | | |
| 8 | | |
| 9 | | |
| 10 | | |
| 11 | | |

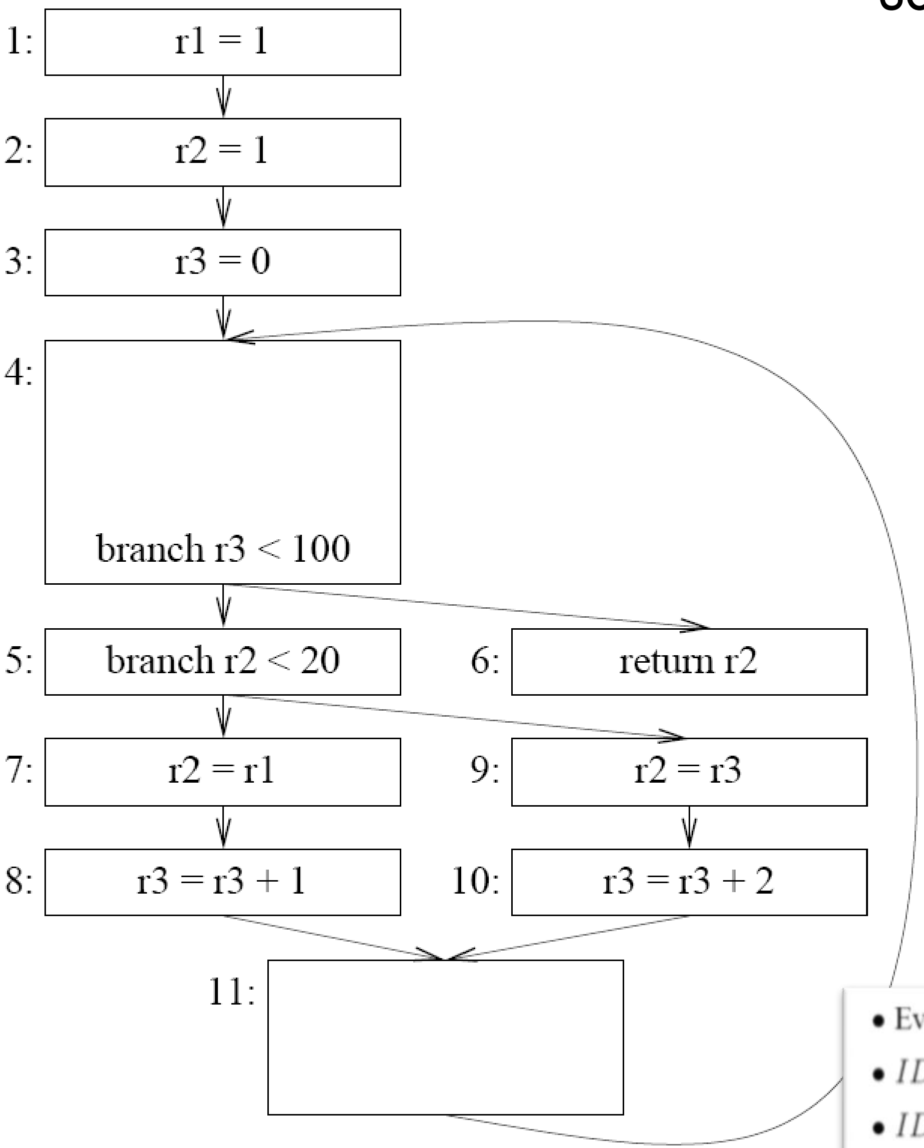
# SSA Example




set of nodes that dominate n

| Node | $DOM[n]$ | $IDOM[n]$ |
|---|---|---|
| 1 | 1 | |
| 2 | 1, 2 | |
| 3 | 1, 2, 3 | |
| 4 | 1, 2, 3, 4 | |
| 5 | 1, 2, 3, 4, 5 | |
| 6 | 1, 2, 3, 4, 6 | |
| 7 | 1, 2, 3, 4, 5, 7 | |
| 8 | 1, 2, 3, 4, 5, 7, 8 | |
| 9 | 1, 2, 3, 4, 5, 9 | |
| 10 | 1, 2, 3, 4, 5, 9, 10 | |
| 11 | 1, 2, 3, 4, 5, 11 | |

# SSA Example



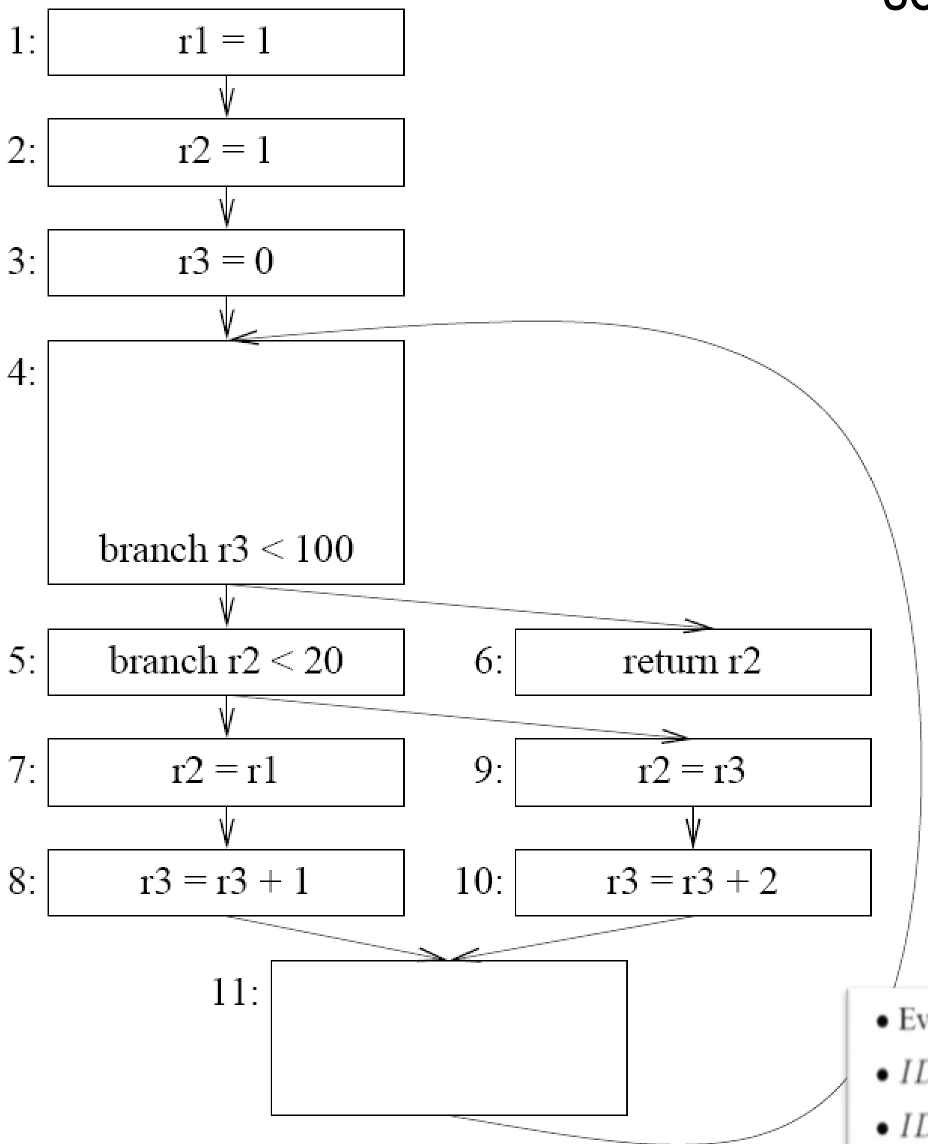

set of nodes that dominate n

| Node | $DOM[n]$ | $IDOM[n]$ |
|---|---|---|
| 1 | 1 | |
| 2 | 1, 2 | |
| 3 | 1, 2, 3 | |
| 4 | 1, 2, 3, 4 | |
| 5 | 1, 2, 3, 4, 5 | |
| 6 | 1, 2, 3, 4, 6 | |
| 7 | 1, 2, 3, 4, 5, 7 | |
| 8 | 1, 2, 3, 4, 5, 7, 8 | |
| 9 | 1, 2, 3, 4, 5, 9 | |
| 10 | 1, 2, 3, 4, 5, 9, 10 | |
| 11 | 1, 2, 3, 4, 5, 11 | |

- Every node $n$ ($n \neq s_0$) has exactly one immediate dominator $IDom[n]$.
- $IDom[n] \neq n$
- $IDom[n]$ dominates $n$
- $IDom[n]$ does not dominate any other dominator of $n$.

Hence: last dominator of n on any path from s0 to n is IDom[n]

# SSA Example



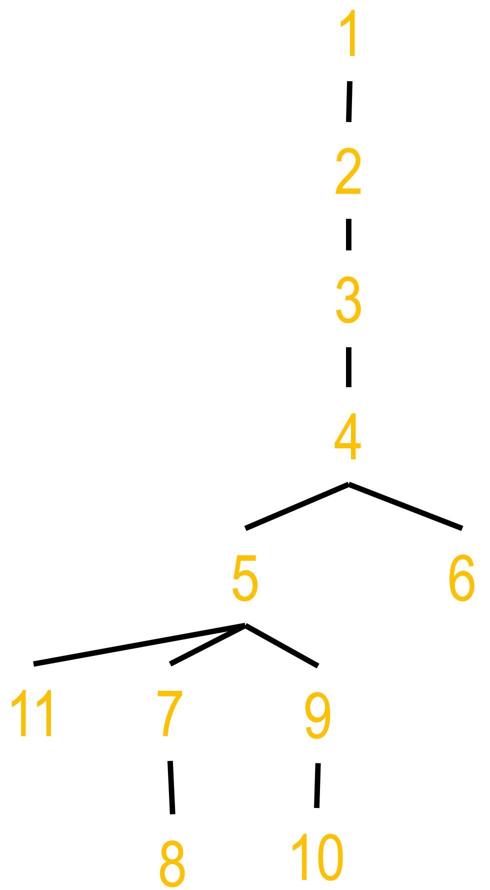

set of nodes that dominate n

| Node | $DOM[n]$ | $IDOM[n]$ |
|---|---|---|
| 1 | 1 | -- |
| 2 | 1, 2 | 1 |
| 3 | 1, 2, 3 | 2 |
| 4 | 1, 2, 3, 4 | 3 |
| 5 | 1, 2, 3, 4, 5 | 4 |
| 6 | 1, 2, 3, 4, 6 | 4 |
| 7 | 1, 2, 3, 4, 5, 7 | 5 |
| 8 | 1, 2, 3, 4, 5, 7, 8 | 7 |
| 9 | 1, 2, 3, 4, 5, 9 | 5 |
| 10 | 1, 2, 3, 4, 5, 9, 10 | 9 |
| 11 | 1, 2, 3, 4, 5, 11 | 5 |

- Every node $n$ ($n \neq s_0$) has exactly one immediate dominator $IDom[n]$.
- $IDom[n] \neq n$
- $IDom[n]$ dominates $n$
- $IDom[n]$ does not dominate any other dominator of $n$.

**Hence: last dominator of n on any path from s0 to n is IDom[n]**

# SSA Example




**Dominator Tree**

set of nodes that dominate n

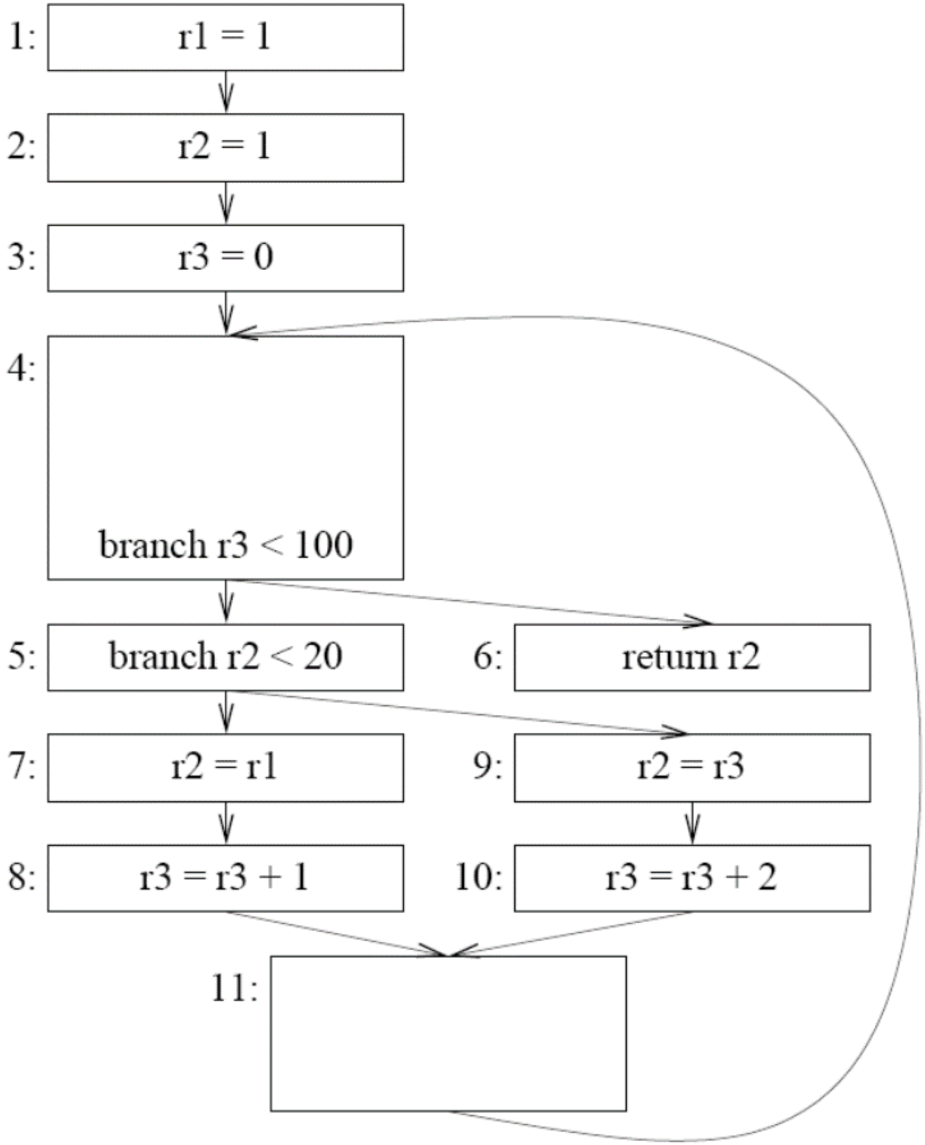
| Node | $DOM[n]$ | $IDOM[n]$ |
|---|---|---|
| 1 | 1 | -- |
| 2 | 1, 2 | 1 |
| 3 | 1, 2, 3 | 2 |
| 4 | 1, 2, 3, 4 | 3 |
| 5 | 1, 2, 3, 4, 5 | 4 |
| 6 | 1, 2, 3, 4, 6 | 4 |
| 7 | 1, 2, 3, 4, 5, 7 | 5 |
| 8 | 1, 2, 3, 4, 5, 7, 8 | 7 |
| 9 | 1, 2, 3, 4, 5, 9 | 5 |
| 10 | 1, 2, 3, 4, 5, 9, 10 | 9 |
| 11 | 1, 2, 3, 4, 5, 11 | 5 |

- Every node $n$ ($n \neq s_0$) has exactly one immediate dominator $IDom[n]$.
- $IDom[n] \neq n$
- $IDom[n]$ dominates $n$
- $IDom[n]$ does not dominate any other dominator of $n$.

**Hence: last dominator of n on any path from s0 to n is IDom[n]**

# SSA Example

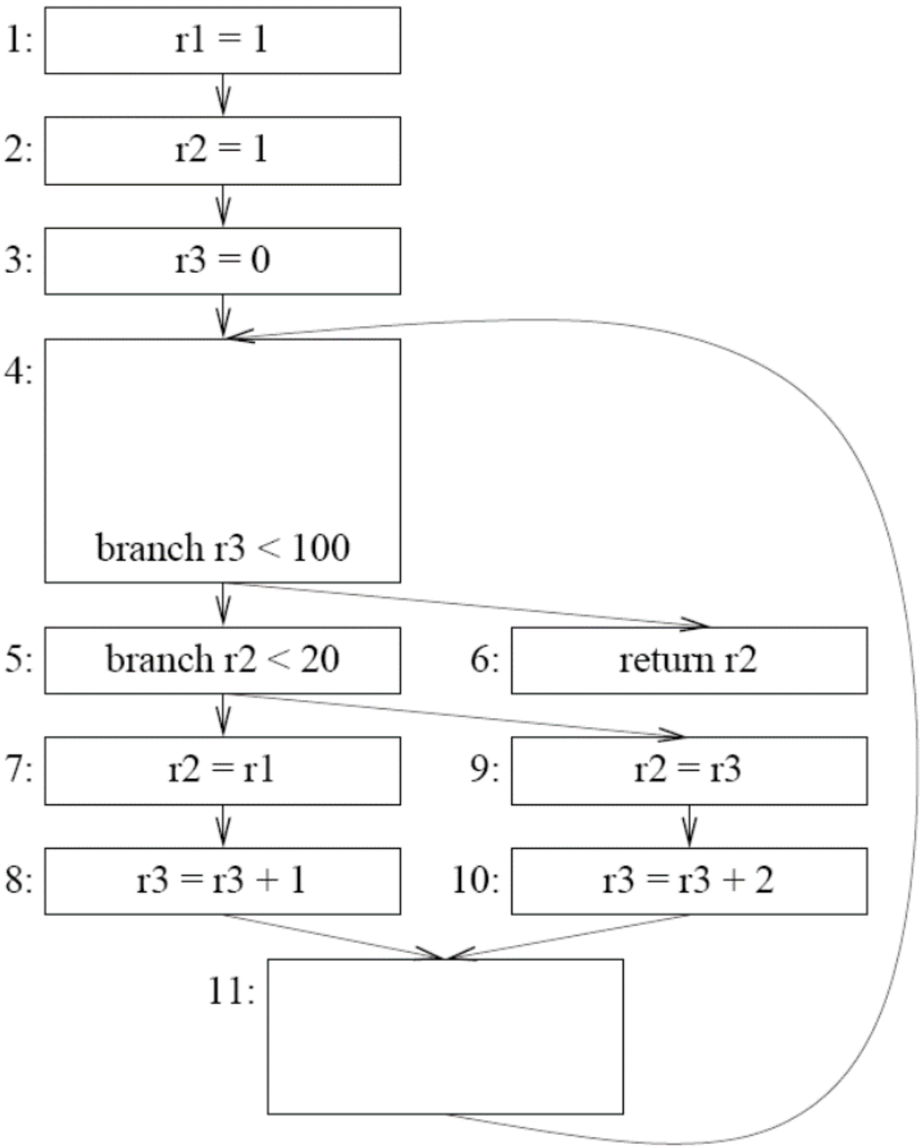
$DF_{local}[n]$ = successors s of n with idom[s] <> n.




| Node | $IDOM[n]$ | | $DF_{local}[n]$ |
|---|---|---|---|
| 1 | -- | | |
| 2 | 1 | | |
| 3 | 2 | | |
| 4 | 3 | | |
| 5 | 4 | | |
| 6 | 4 | | |
| 7 | 5 | | |
| 8 | 7 | | |
| 9 | 5 | | |
| 10 | 9 | | |
| 11 | 5 | | |

# SSA Example

$DF_{local}[n]$ = successors s of n with idom[s] <> n.



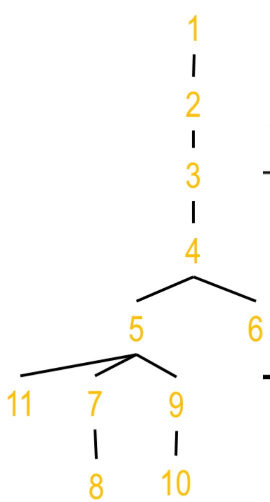

| Node | $IDOM[n]$ | | $DF_{local}[n]$ |
|---|---|---|---|
| 1 | -- | | -- |
| 2 | 1 | | -- |
| 3 | 2 | | -- |
| 4 | 3 | | -- |
| 5 | 4 | | -- |
| 6 | 4 | | -- |
| 7 | 5 | | -- |
| 8 | 7 | | 11 |
| 9 | 5 | | -- |
| 10 | 9 | | 11 |
| 11 | 5 | | 4 |

# SSA Example

- $DF_{local}[n]$: successors of $n$ in CFG that are not strictly dominated by $n$
- $DF_{up}[c]$: nodes in dominance frontier of $c$ that are not strictly dominated by $c$'s immediate dominator

$$DF[n] = DF_{local}[n] \cup \left(\cup_{c \in children[n]} DF_{up}[c]\right)$$

- where $children[n]$ are the nodes whose idom is n.
- Work bottom up in dominator tree.
  Leaf p satisfies DF[ p ] = $DF_{local}$[ p ].




Dominator Tree

| Node | $IDOM[n]$ | $DF_{local}[n]$ |
|---|---|---|
| 1 | -- | -- |
| 2 | 1 | -- |
| 3 | 2 | -- |
| 4 | 3 | -- |
| 5 | 4 | -- |
| 6 | 4 | -- |
| 7 | 5 | -- |
| 8 | 7 | 11 |
| 9 | 5 | -- |
| 10 | 9 | 11 |
| 11 | 5 | 4 |

| n | $U_{c(n)}$ $DF_{up}[c]$ | DF[n] | $DF_{up}[n]$ |
|---|---|---|---|
| 1 | | | |
| 2 | | | |
| 3 | | | |
| 4 | | | |
| 5 | | | |
| 6 | {} | | |
| 7 | | | |
| 8 | {} | | |
| 9 | | | |
| 10 | {} | | |
| 11 | {} | | |

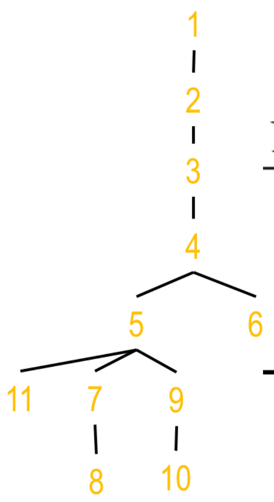
# SSA Example

- $DF_{local}[n]$: successors of $n$ in CFG that are not strictly dominated by $n$
- $DF_{up}[c]$: nodes in dominance frontier of $c$ that are not strictly dominated by $c$'s immediate dominator

$$DF[n] = DF_{local}[n] \cup \left(\cup_{c \in children[n]} DF_{up}[c]\right)$$

- where $children[n]$ are the nodes whose idom is n.
- Work bottom up in dominator tree.

Leaf p satisfies DF[ p ] = $DF_{local}$[ p ].




Dominator Tree

| Node | $IDOM[n]$ | $DF_{local}[n]$ |
|---|---|---|
| 1 | -- | -- |
| 2 | 1 | -- |
| 3 | 2 | -- |
| 4 | 3 | -- |
| 5 | 4 | -- |
| 6 | 4 | -- |
| 7 | 5 | -- |
| 8 | 7 | 11 |
| 9 | 5 | -- |
| 10 | 9 | 11 |
| 11 | 5 | 4 |

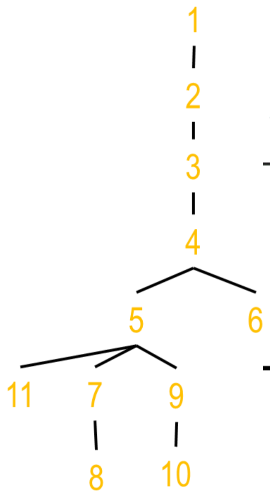
| n | $U_{c(n)} DF_{up}[c]$ | DF[n] | $DF_{up}[n]$ |
|---|---|---|---|
| 1 | | | |
| 2 | | | |
| 3 | | | |
| 4 | | | |
| 5 | | | |
| 6 | {} | -- | |
| 7 | | | |
| 8 | {} | 11 | |
| 9 | | | |
| 10 | {} | 11 | |
| 11 | {} | 4 | |

# SSA Example

- $DF_{local}[n]$: successors of $n$ in CFG that are not strictly dominated by $n$
- $DF_{up}[c]$: nodes in dominance frontier of $c$ that are not strictly dominated by $c$'s immediate dominator

$$DF[n] = DF_{local}[n] \cup \left(\cup_{c \in children[n]} DF_{up}[c]\right)$$

- where $children[n]$ are the nodes whose idom is n.
- Work bottom up in dominator tree.
  Leaf p satisfies DF[ p ] = $DF_{local}$[ p ].



Dominator Tree

| Node | $IDOM[n]$ | | $DF_{local}[n]$ |
|---|---|---|---|
| 1 | -- | | -- |
| 2 | 1 | | -- |
| 3 | 2 | | -- |
| 4 | 3 | | -- |
| 5 | 4 | | -- |
| 6 | 4 | | -- |
| 7 | 5 | | -- |
| 8 | 7 | | 11 |
| 9 | 5 | | -- |
| 10 | 9 | | 11 |
| 11 | 5 | | 4 |

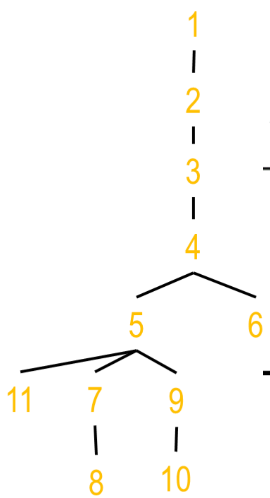
| n | $U_{c(n)}$ $DF_{up}[c]$ | DF[n] | $DF_{up}[n]$ |
|---|---|---|---|
| 1 | | | |
| 2 | | | |
| 3 | | | |
| 4 | | | |
| 5 | | | |
| 6 | {} | -- | -- |
| 7 | | | |
| 8 | {} | 11 | 11 |
| 9 | | | |
| 10 | {} | 11 | 11 |
| 11 | {} | 4 | 4 |

# SSA Example

- $DF_{local}[n]$: successors of $n$ in CFG that are not strictly dominated by $n$
- $DF_{up}[c]$: nodes in dominance frontier of $c$ that are not strictly dominated by $c$'s immediate dominator

$$DF[n] = DF_{local}[n] \cup \left(\cup_{c \in children[n]} DF_{up}[c]\right)$$

- where $children[n]$ are the nodes whose idom is n.
- Work bottom up in dominator tree.
  Leaf p satisfies DF[ p ] = $DF_{local}$[ p ]




Dominator Tree

| Node | $IDOM[n]$ | $DF_{local}[n]$ |
|---|---|---|
| 1 | -- | -- |
| 2 | 1 | -- |
| 3 | 2 | -- |
| 4 | 3 | -- |
| 5 | 4 | -- |
| 6 | 4 | -- |
| 7 | 5 | -- |
| 8 | 7 | 11 |
| 9 | 5 | -- |
| 10 | 9 | 11 |
| 11 | 5 | 4 |

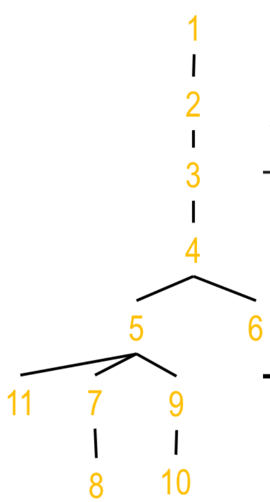
| n | $U_{c(n)}$ $DF_{up}[c]$ | DF[n] | $DF_{up}[n]$ |
|---|---|---|---|
| 1 | | | |
| 2 | | | |
| 3 | | | |
| 4 | | | |
| 5 | | | |
| 6 | {} | -- | -- |
| 7 | 11 | | |
| 8 | {} | 11 | 11 |
| 9 | 11 | | |
| 10 | {} | 11 | 11 |
| 11 | {} | 4 | 4 |

# SSA Example

- $DF_{local}[n]$: successors of $n$ in CFG that are not strictly dominated by $n$
- $DF_{up}[c]$: nodes in dominance frontier of $c$ that are not strictly dominated by $c$'s immediate dominator

$$DF[n] = DF_{local}[n] \cup (\cup_{c \in children[n]} DF_{up}[c])$$

- where $children[n]$ are the nodes whose idom is n.
- Work bottom up in dominator tree.
  Leaf p satisfies DF[ p ] = $DF_{local}$[ p ]



Dominator Tree

| Node | $IDOM[n]$ | $DF_{local}[n]$ |
|---|---|---|
| 1 | -- | -- |
| 2 | 1 | -- |
| 3 | 2 | -- |
| 4 | 3 | -- |
| 5 | 4 | -- |
| 6 | 4 | -- |
| 7 | 5 | -- |
| 8 | 7 | 11 |
| 9 | 5 | -- |
| 10 | 9 | 11 |
| 11 | 5 | 4 |

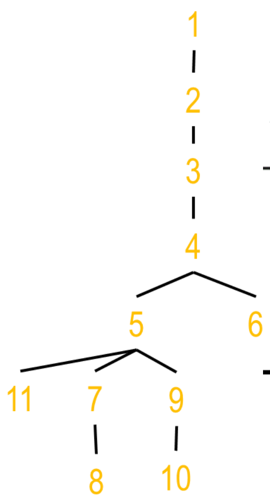
| n | $U_{c(n)}$ $DF_{up}[c]$ | DF[n] | $DF_{up}[n]$ |
|---|---|---|---|
| 1 | | | |
| 2 | | | |
| 3 | | | |
| 4 | | | |
| 5 | | | |
| 6 | {} | -- | -- |
| 7 | 11 | 11 | |
| 8 | {} | 11 | 11 |
| 9 | 11 | 11 | |
| 10 | {} | 11 | 11 |
| 11 | {} | 4 | 4 |

# SSA Example

- $DF_{local}[n]$: successors of $n$ in CFG that are not strictly dominated by $n$
- $DF_{up}[c]$: nodes in dominance frontier of $c$ that are not strictly dominated by $c$'s immediate dominator

$$DF[n] = DF_{local}[n] \cup (\cup_{c \in children[n]} DF_{up}[c])$$

- where $children[n]$ are the nodes whose idom is n.
- Work bottom up in dominator tree.
  Leaf p satisfies DF[ p ] = $DF_{local}$[ p ].



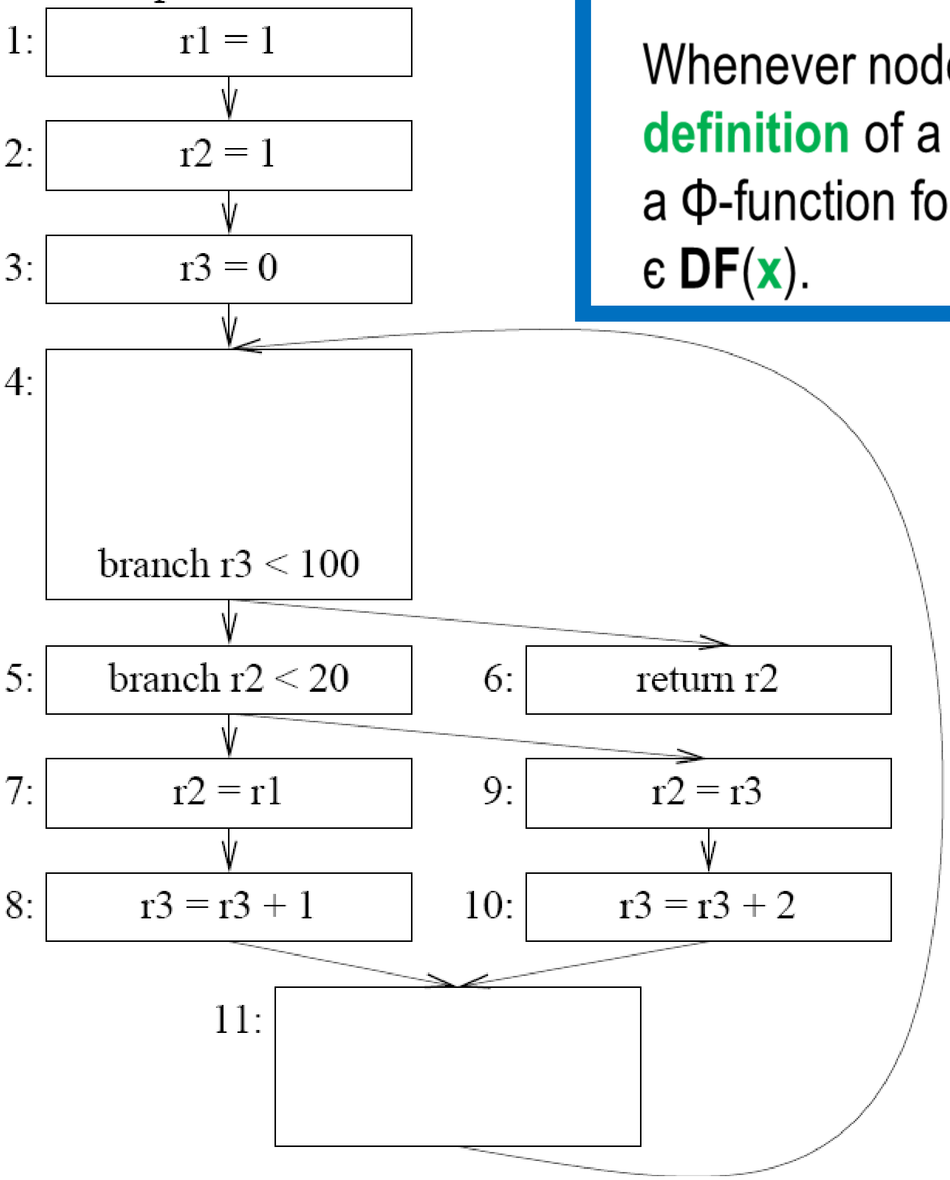

Dominator Tree

| Node | $IDOM[n]$ | | $DF_{local}[n]$ |
|---|---|---|---|
| 1 | -- | | -- |
| 2 | 1 | | -- |
| 3 | 2 | | -- |
| 4 | 3 | | -- |
| 5 | 4 | | -- |
| 6 | 4 | | -- |
| 7 | 5 | | -- |
| 8 | 7 | | 11 |
| 9 | 5 | | -- |
| 10 | 9 | | 11 |
| 11 | 5 | | 4 |

| n | $U_{c(n)}$ $DF_{up}[c]$ | DF[n] | $DF_{up}[n]$ |
|---|---|---|---|
| 1 | | -- | |
| 2 | : | -- | : |
| 3 | : | -- | : |
| 4 | | -- | |
| 5 | | 4 | |
| 6 | {} | -- | -- |
| 7 | 11 | 11 | ... |
| 8 | {} | 11 | 11 |
| 9 | 11 | 11 | ... |
| 10 | {} | 11 | 11 |
| 11 | {} | 4 | 4 |

# SSA Example

**Insert *phi*-functions:**



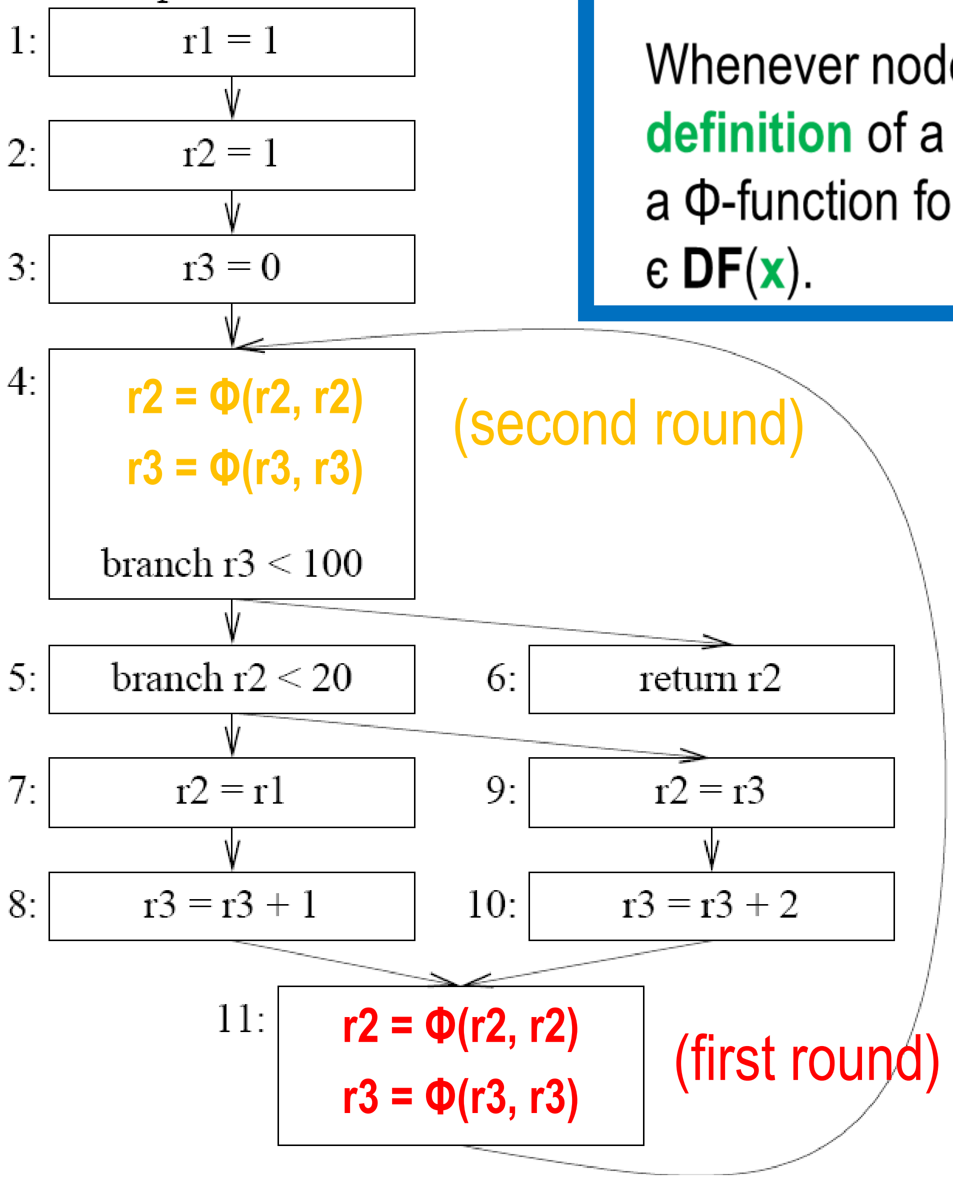

**Dominance Frontier Criterion:**

Whenever node **x** contains a **definition** of a register **r**, insert a Φ-function for **r** in all nodes **z** ϵ **DF**(**x**).

| n | DF[n] |
|---|---|
| 1 | {} |
| 2 | {} |
| 3 | {} |
| 4 | {} |
| 5 | 4 |
| 6 | {} |
| 7 | 11 |
| 8 | 11 |
| 9 | 11 |
| 10 | 11 |
| 11 | 4 |

# SSA Example

**Insert *phi*-functions:**




**Dominance Frontier Criterion:**

Whenever node **x** contains a **definition** of a register **r**, insert a Φ-function for **r** in all nodes **z** є **DF**(**x**).

| n | DF[n] |
|---|---|
| 1 | {} |
| 2 | {} |
| 3 | {} |
| 4 | {} |
| 5 | 4 |
| 6 | {} |
| 7 | 11 |
| 8 | 11 |
| 9 | 11 |
| 10 | 11 |
| 11 | 4 |

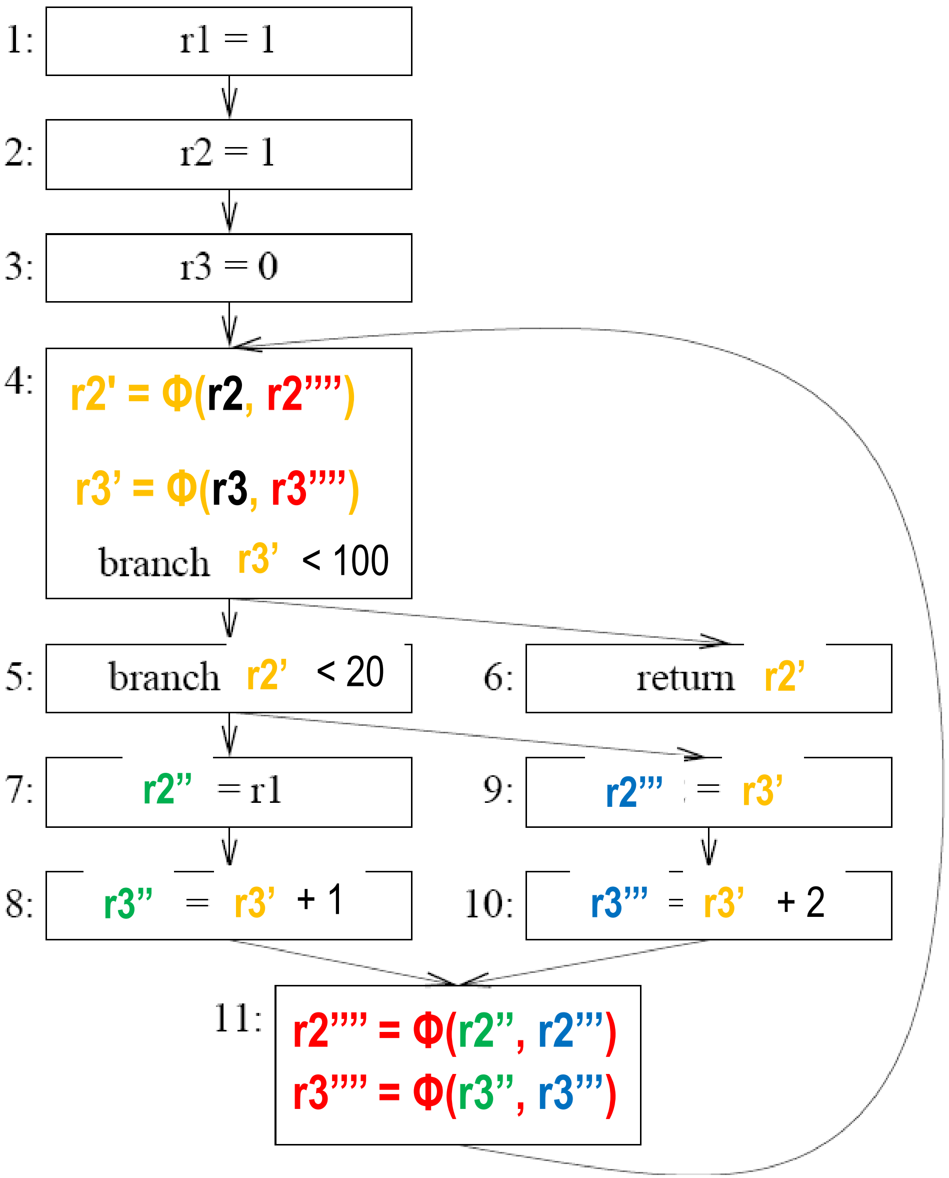
# SSA Example

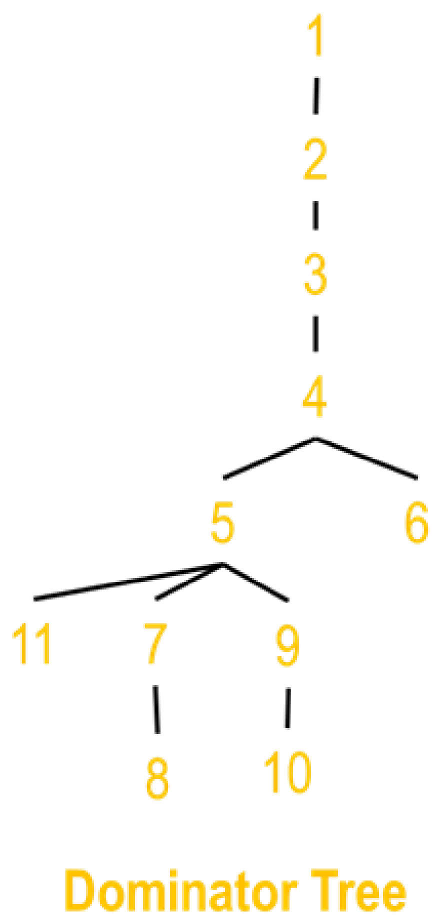
**Rename Variables:**

1. traverse dominator tree, renaming different definitions of $r$ to $r_1, r_2, r_3 \ldots$
2. rename each regular use of $r$ to most recent definition of $r$
3. rename $\phi$-function arguments with each incoming edge's unique definition

# SSA Example

**Rename Variables:**



# Alternative construction methods for SSA

Lengauer-Tarjan: efficient computation of dominance tree

- near linear time
- uses depth-first spanning tree
- see MCIML, Section 19.2

John Aycock, Nigel Horspool: *Simple Generation of Static Single Assignment Form.9nd Conference on Compiler Construction (CC 2000), pages 110—124, LNCS 1781, Springer 2000*

- *Starts from "crude" placement of Φ-functions: in every block, for every variable*
- *then iteratively eliminates unnecessary Φ-functions*
- *For reducible CFG*

M. Braun, et al.: *Simple and Efficient Construction of Static Single Assignment Form.22nd Conference on Compiler Construction (CC 2013), pages 102—122, LNCS 7791, Springer 2013*

- *avoids computation of dominance or iterated DF*
- *works directly on AST (avoids CFG)*

# Static Single Assignment

**Static Single Assignment Advantages:**

- Less space required to represent def-use chains. For each variable, space is proportional to uses * defs.
- Eliminates unnecessary relationships:

```
for i = 1 to N do A[i] = 0
for i = 1 to M do B[i] = 1
```

  - No reason why both loops should be forced to use same register to hold index register.
  - SSA renames second `i` to new register which may lead to better register allocation.
- SSA form make certain optimizations quick and easy $\rightarrow$ dominance property.
  - Variables have only one definition - no ambiguity.
  - Dominator information is encoded in the assignments.

# SSA Dominance Property

Dominance property of SSA form: definitions dominate uses

- If $x$ is $i^{\text{th}}$ argument of $\phi$-function in node $n$, then definition of $x$ dominates $i^{\text{th}}$ predecessor of $n$.
- If $x$ is used in non-$\phi$ statement in node $n$, then definition of $x$ dominates $n$.

# SSA Dead Code Elimination

Given $d$: `t = x op y`

- `t` is live at end of node $d$ if there exists path from end of $d$ to use of `t` that does not go through definition of `t`.
- if program not in SSA form, need to perform liveness analysis to determine if `t` live at end of $d$.
- if program is in SSA form:

# SSA Dead Code Elimination

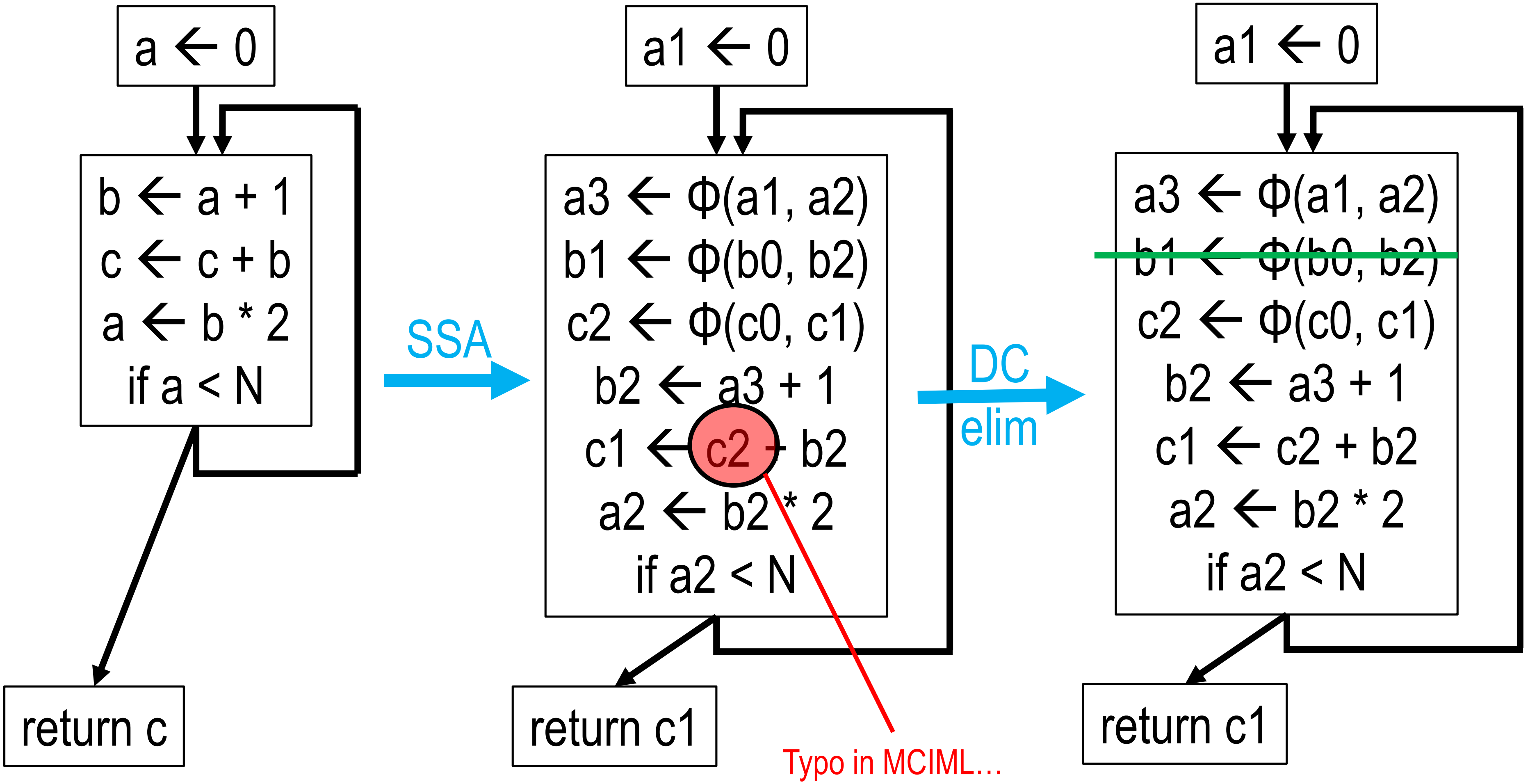
Given $d$: `t = x op y`

- `t` is live at end of node $d$ if there exists path from end of $d$ to use of `t` that does not go through definition of `t`.
- if program not in SSA form, need to perform liveness analysis to determine if `t` live at end of $d$.
- if program is in SSA form:
  - cannot be another definition of `t`
  - if there exists use of `t`, then path from end of $d$ to use exists, since definitions dominate uses.
    - every use has a unique definition
    - `t` is live at end of node $d$ if `t` is used at least once

# SSA Dead Code Elimination

Algorithm:

WHILE (for each temporary t with no uses &&
statement defining t has no other side-effects) DO
delete statement definition t



# SSA Simple Constant Propagation

Given $d$: `t = c`, c is constant Given $u$: `x = t op b`

- if program not in SSA form:
  - need to perform reaching definition analysis
  - use of `t` in $u$ may be replaced by `c` if $d$ reaches $u$ and no other definition of `t` reaches $u$
- if program is in SSA form:

# SSA Simple Constant Propagation

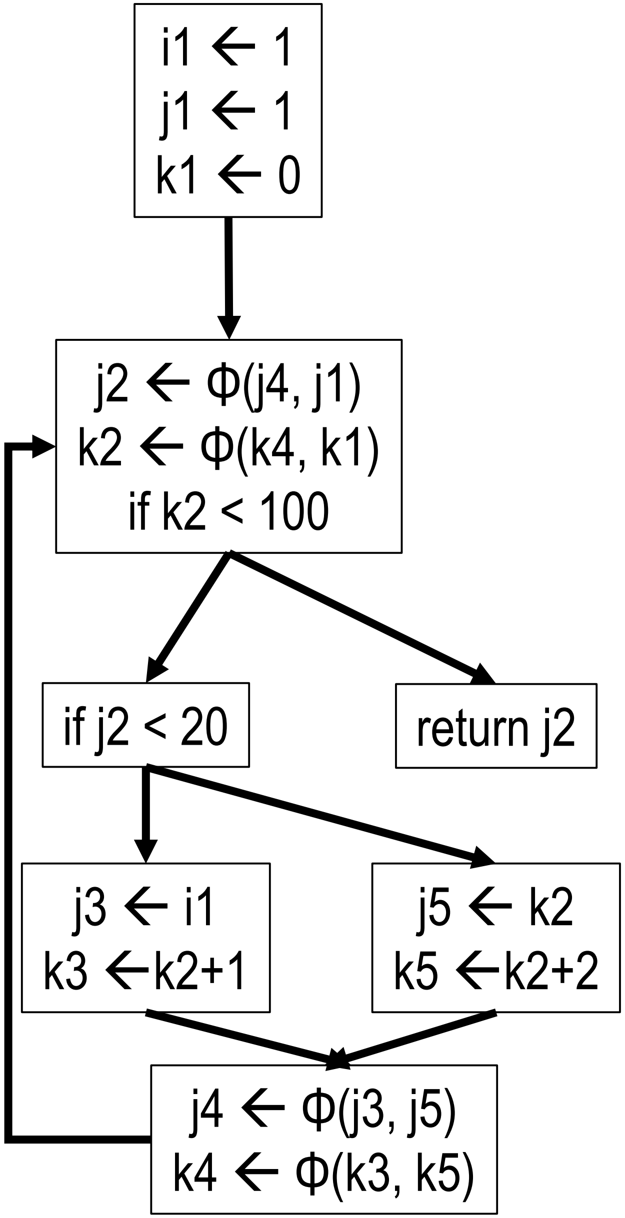
Given $d$: `t = c`, c is constant Given $u$: `x = t op b`

- if program not in SSA form:
  - need to perform reaching definition analysis
  - use of `t` in $u$ may be replaced by `c` if $d$ reaches $u$ and no other definition of `t` reaches $u$
- if program is in SSA form:
  - $d$ reaches $u$, since definitions dominate uses, and no other definition of `t` exists on path from $d$ to $u$
  - $d$ is only definition of `t` that reaches $u$, since it is the only definition of `t`.
    * any use of `t` can be replaced by `c`
    * any $\phi$-function of form `v =` $\phi(c_1, c_2, \ldots, c_n)$, where $c_i = c$, can be replaced by `v = c`

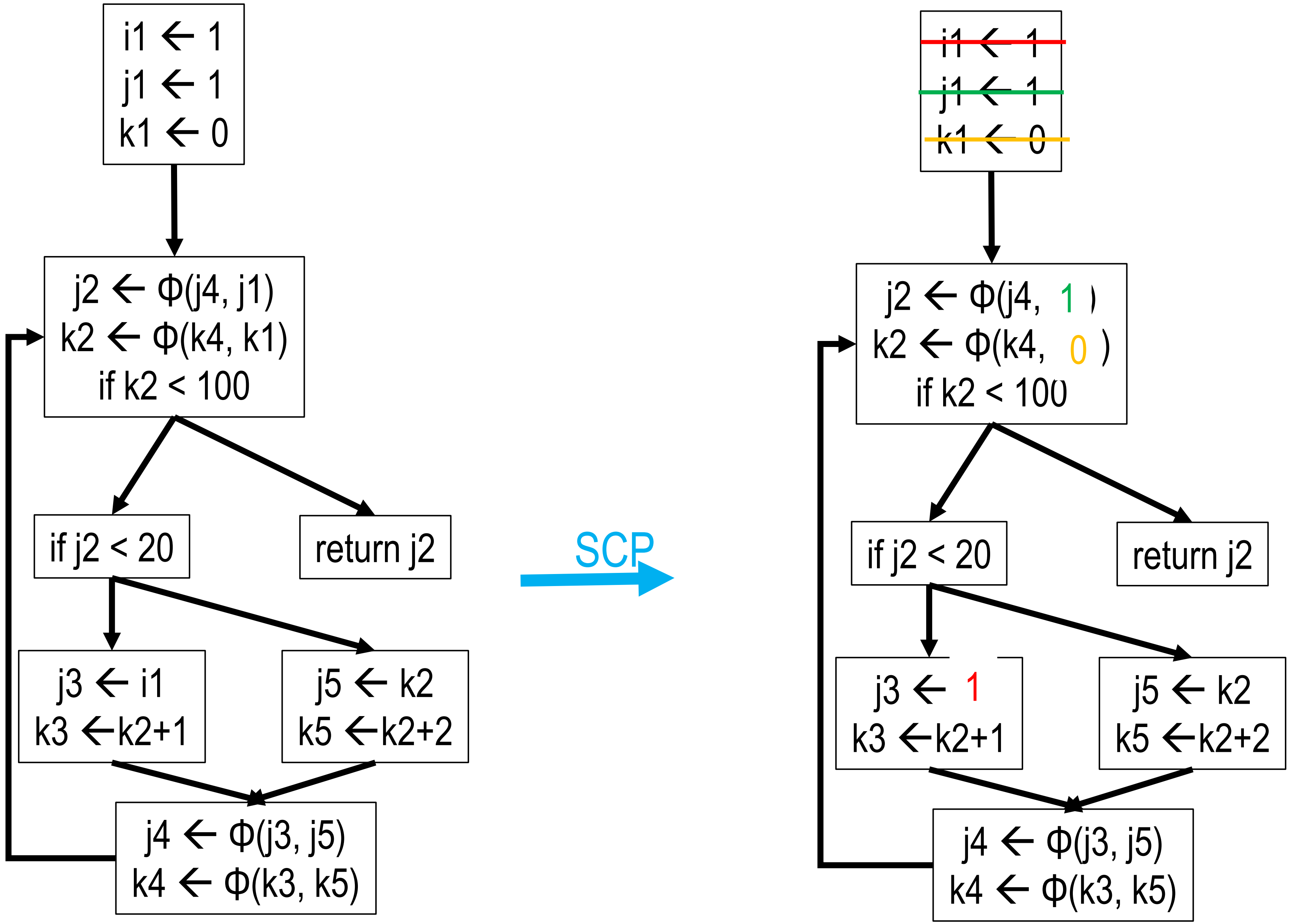
eliminate branches whose outcome is constant

Similarly: copy propagation, constant folding, constant condition, elimination of unreachable code

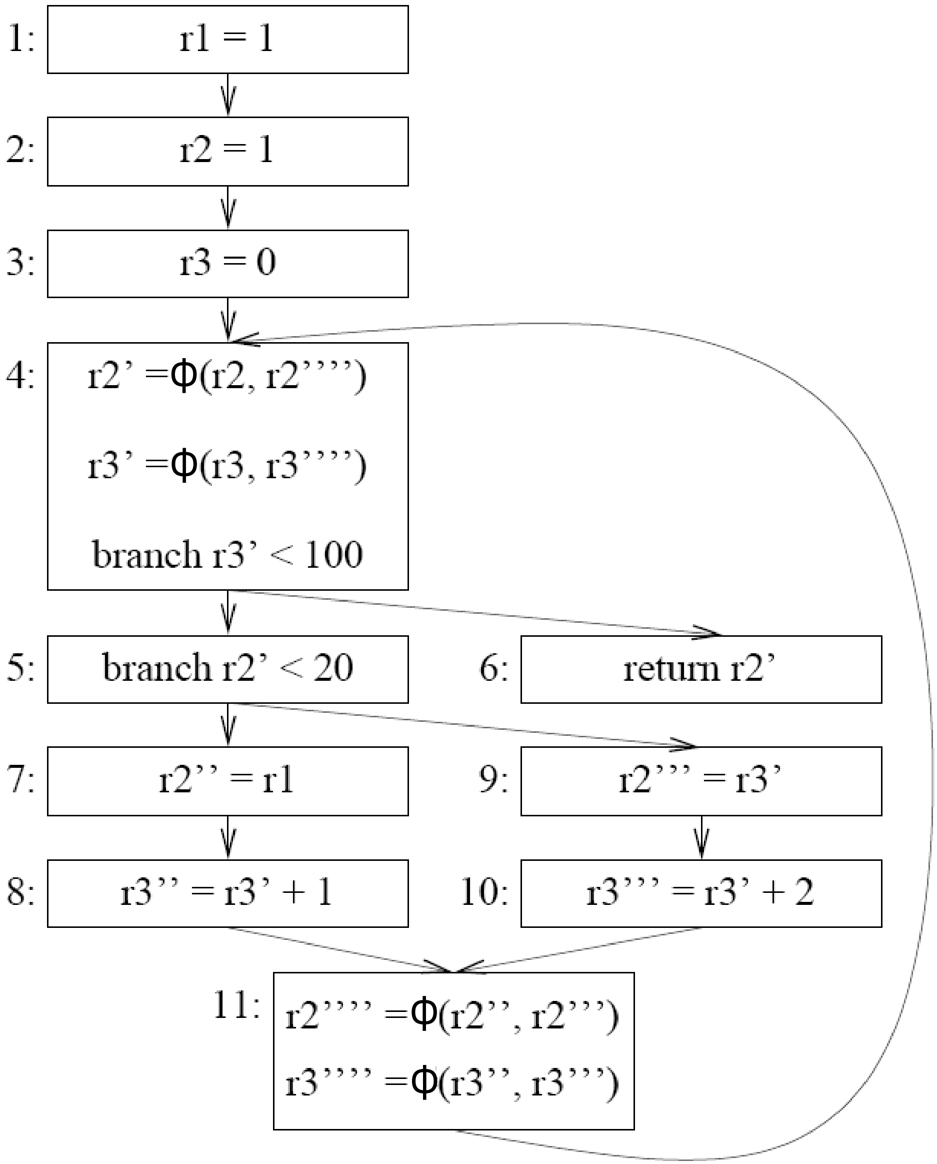
# SSA Simple Constant Propagation



# SSA Simple Constant Propagation



# SSA Conditional Constant Propagation




- `r2` always has value of 1
- nodes 9, 10 never executed
- "simple" constant propagation algorithms assumes (through reaching definitions analysis) nodes 9, 10 may be executed.
- cannot optimize use of `r2` in node 5 since definitions 7 and 9 both reach 5.

# SSA Conditional Constant Propagation

Much smarter than “simple” constant propagation:

- Does not assume a node can execute until evidence exists that it can be.
- Does not assume register is non-constant unless evidence exists that it is.

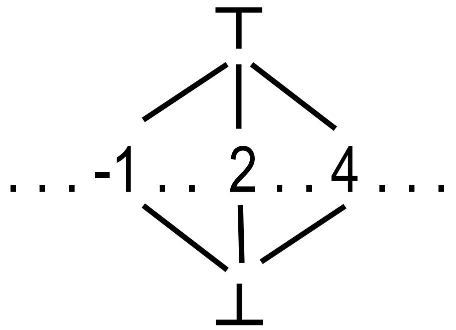
# SSA Conditional Constant Propagation

Much smarter than "simple" constant propagation:

- Does not assume a node can execute until evidence exists that it can be.
- Does not assume register is non-constant unless evidence exists that it is.

Track run-time value of each register `r` using *lattice* of values:

- $V[r] = \perp$ (bottom): compiler has seen no evidence that any assignment to `r` is ever executed.
- $V[r] = 4$: compiler has seen evidence that an assignment `r = 4` is executed, but has seen no evidence that `r` is ever assigned to another value.
- $V[r] = \top$ (top): compiler has seen evidence that `r` will have, at various times, two different values, or some value that is not predictable at compile-time.



# SSA Conditional Constant Propagation

Much smarter than "simple" constant propagation:

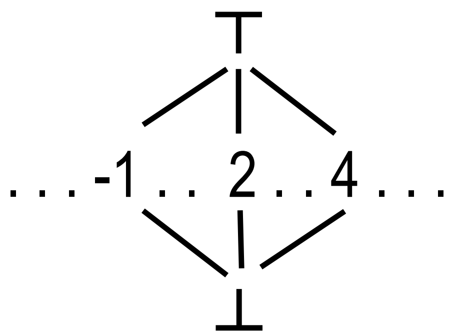
- Does not assume a node can execute until evidence exists that it can be.
- Does not assume register is non-constant unless evidence exists that it is.

Track run-time value of each register `r` using *lattice* of values:

- $V[r] = \bot$ (bottom): compiler has seen no evidence that any assignment to `r` is ever executed.
- $V[r] = 4$: compiler has seen evidence that an assignment `r = 4` is executed, but has seen no evidence that `r` is ever assigned to another value.
- $V[r] = \top$ (top): compiler has seen evidence that `r` will have, at various times, two different values, or some value that is not predictable at compile-time.

Also:

- all registers start at bottom of lattice
- new information can only move registers up in lattice



# SSA Conditional Constant Propagation

Track executability of each node in $N$:

- $E[N] =$ false: compiler has seen no evidence that node $N$ can ever be executed.
- $E[N] =$ true: compiler has seen evidence that node $N$ can be executed.

Initially:

- $V[r] = \perp$, for all registers $r$
- $E[s_0] =$ true, $s_0$ is CFG start node
- $E[N] =$ false, for all CFG nodes $N \neq s_0$

# SSA Conditional Constant Propagation

Algorithm: apply following conditions until no more changes occur to $E$ or $V$ values:

1. Given: register `r` with no definition (formal parameter, uninitialized).
   Action: $V[r] = \top$
2. Given: executable node $B$ with only one successor $C$
   Action: $E[C] = \text{true}$
3. Given: executable assignment `r = x op y`, $V[x] = c_1$ and $V[y] = c_2$
   Action: $V[r] = c_1 \text{op} c_2$ In particular, use this rule for **r = c.**
4. Given: executable assignment `r = x op y`, $V[x] = \top$ or $V[y] = \top$
   Action: $V[r] = \top$
5. Given: executable assignment `r` $=$ $\phi(x_1, x_2, ..., x_n)$, $V[x_i] = c_1$, $V[x_j] = c_2$, and predecessors $i$ and $j$ are executable
   Action: $V[r] = \top$
6. Given: executable assignment **r = M [...]** or **r = f (...)**:
   Action: V[ r ] = ⊤

# SSA Conditional Constant Propagation

7. Given: executable assignment **r = Φ (x$_1$, …, x$_n$)** where V [ **x$_i$** ] = ⊤
   for some i such that the i$^{th}$ predecessor is executable:
   Action: V[ **r** ] = ⊤

8. Given: executable assignment **r = Φ (x$_1$, …, x$_n$)** where
   -- V [ **x$_i$** ] = $c_i$ for some i where the i$^{th}$ predecessor is executable, and
   -- for each j≠i, either the j$^{th}$ predecessor is not executable or V[ **x$_j$** ] є { ⊥ , $c_i$ }:
   Action: V[ **r** ] = $c_i$

9. Given: executable branch **br x bop y, L1 (else L2)** where V [ x ] = ⊤ or V [ y ] = ⊤
   Action: E[ **L1** ] = true and E[ **L2** ] = true

10. Given: executable branch **br x bop y, L1 (else L2)** where V [ x ] = $c_1$ and V [ y ] = $c_2$
    Action: E[ **L1** ] = true or E[ **L2** ] = true depending on $c_1$ **bop** $c_2$

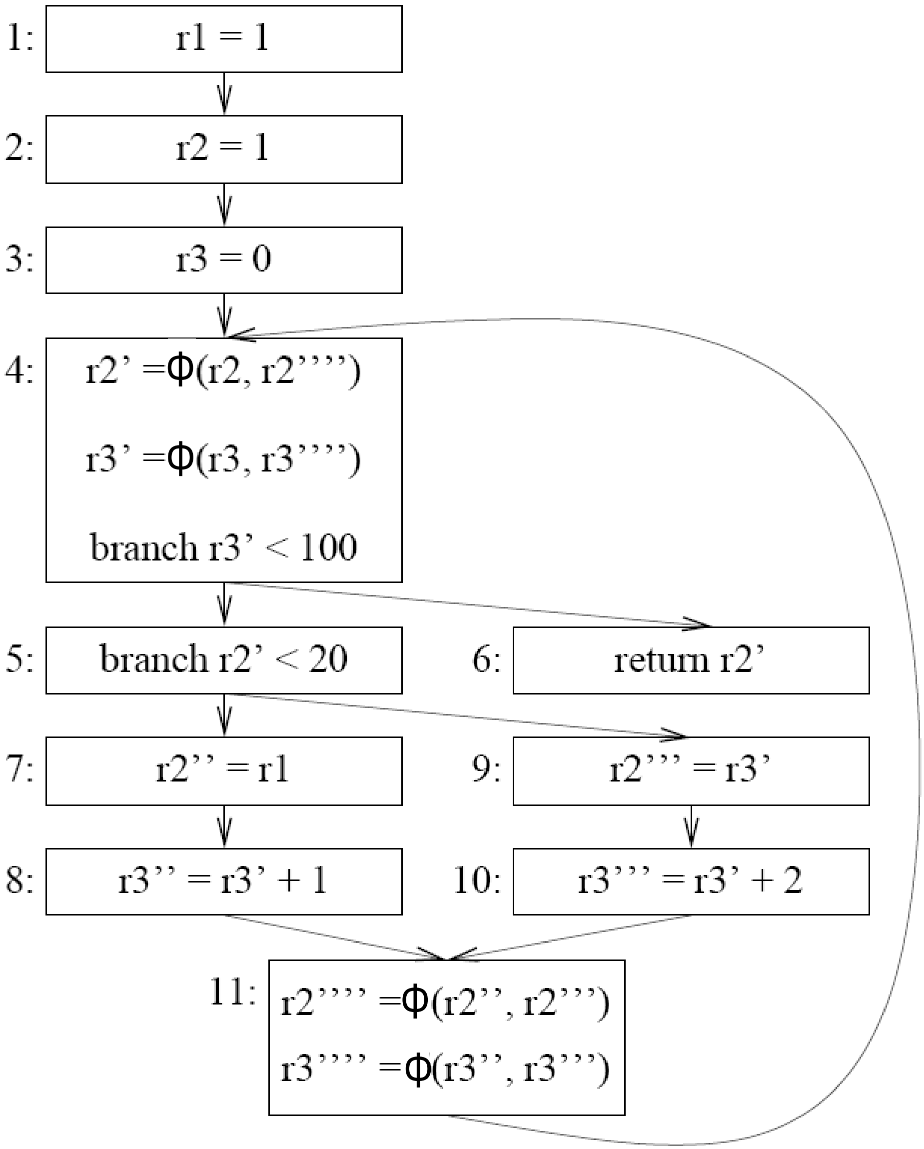
Iterate until no update possible.

# SSA Conditional Constant Propagation

Given $V$, $E$ values, program can be optimized as follows:

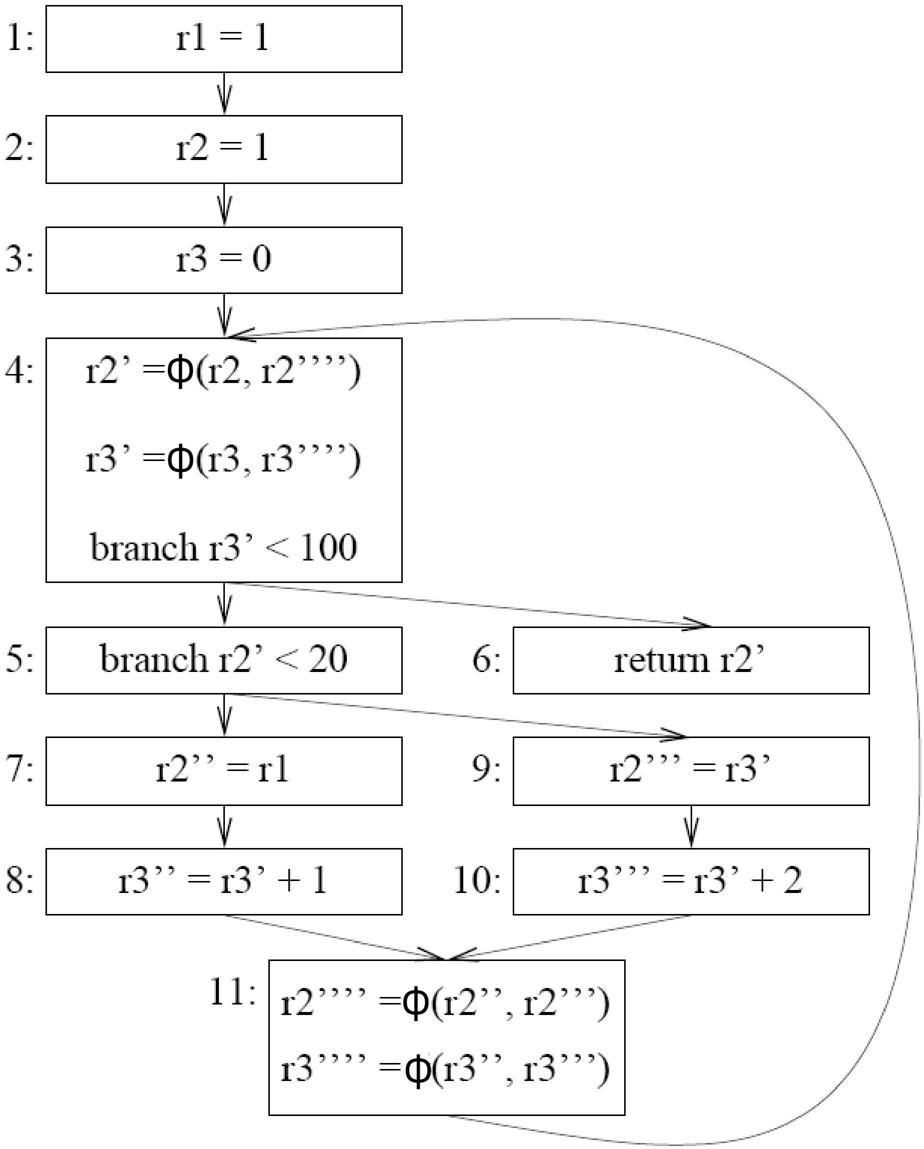
- if $E[B] =$ false, delete node $B$ form CFG.
- if $V[r] = c$, replace each use of `r` by $c$, delete assignment to $r$.

# SSA Conditional Constant Propagation: example



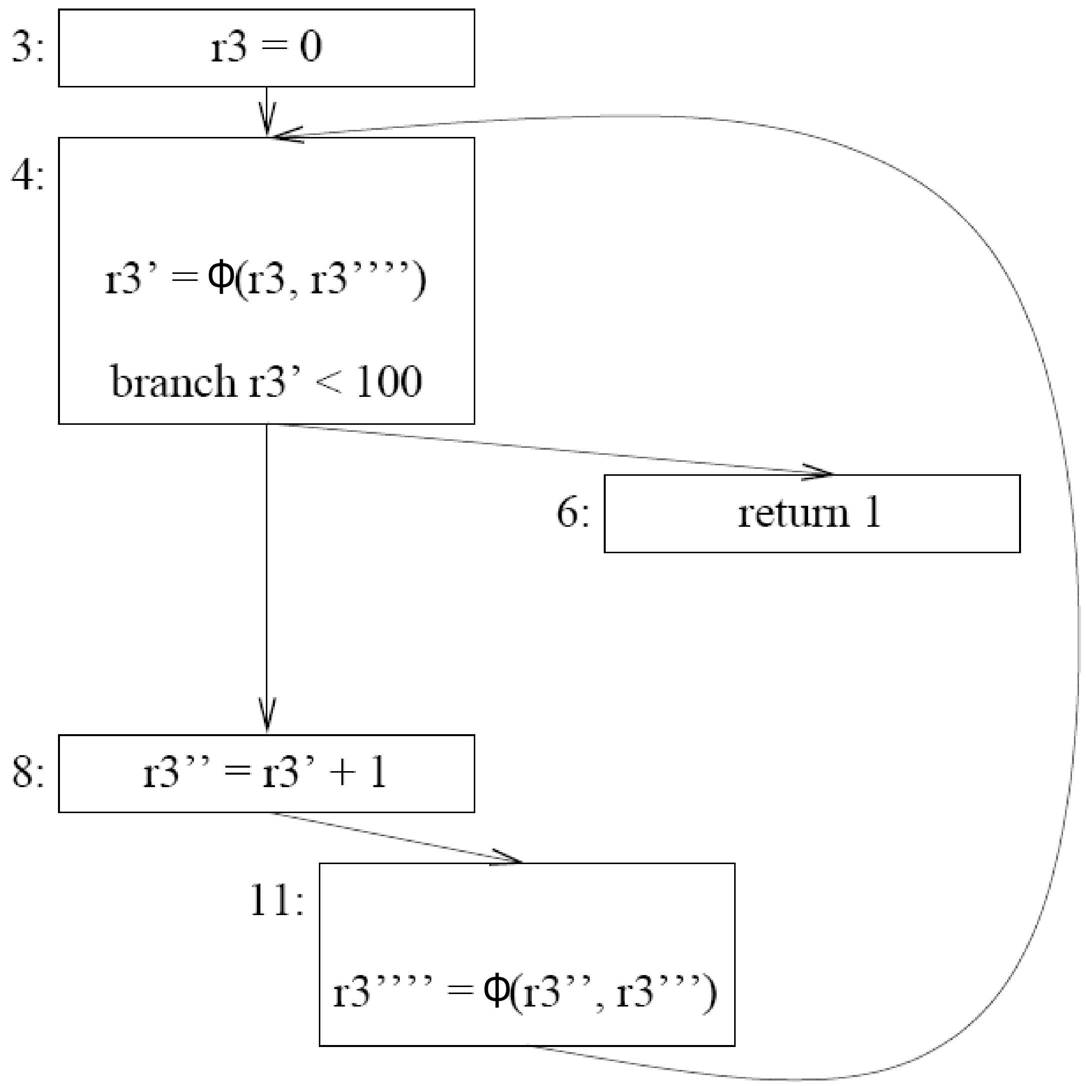

| $N$ | $E[N]$ | $r$ | $V[r]$ |
|---|---|---|---|
| 1 | t | r1 | $\bot$ |
| 2 | f | r2 | $\bot$ |
| 3 | f | r2' | $\bot$ |
| 4 | f | r2'' | $\bot$ |
| 5 | f | r2''' | $\bot$ |
| 6 | f | r2'''' | $\bot$ |
| 7 | f | r3 | $\bot$ |
| 8 | f | r3' | $\bot$ |
| 9 | f | r3'' | $\bot$ |
| 10 | f | r3''' | $\bot$ |
| 11 | f | r3'''' | $\bot$ |

# SSA Conditional Constant Propagation: example



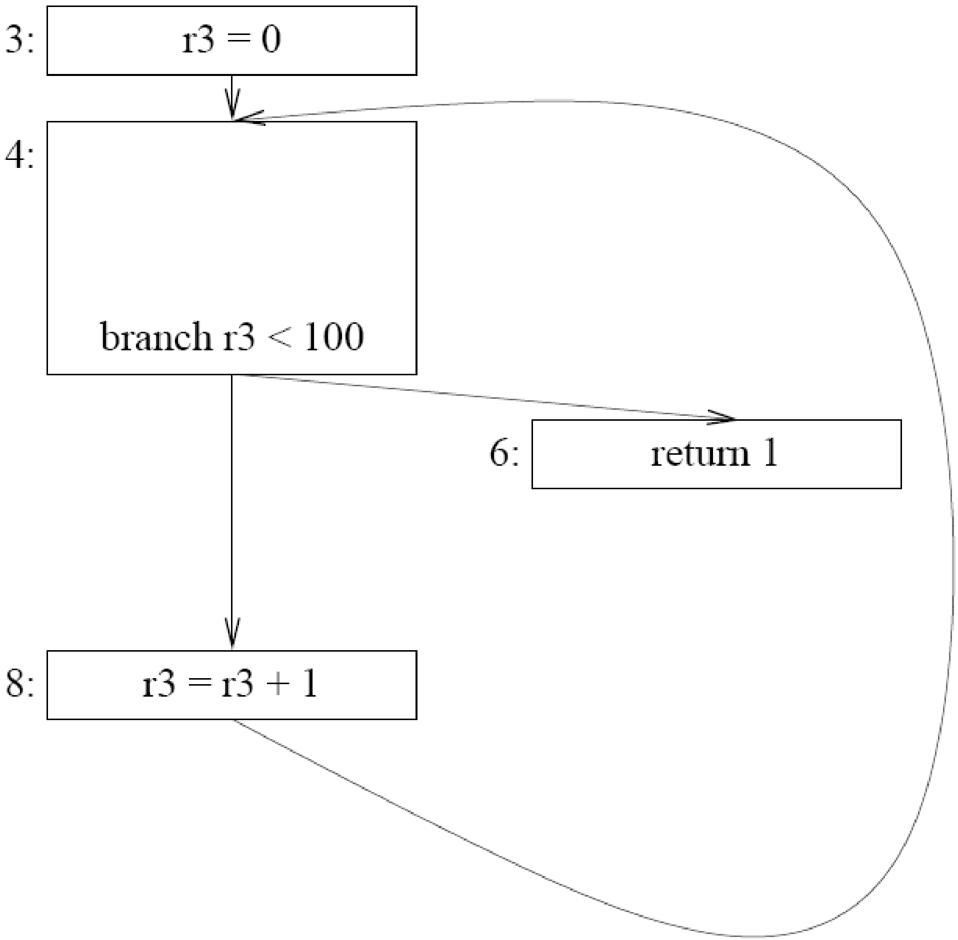

| $N$ | $E[N]$ | $r$ | $V[r]$ |
|---|---|---|---|
| 1 | | r1 | |
| 2 | | r2 | |
| 3 | | r2’ | |
| 4 | | r2’’ | |
| 5 | | r2’’’ | |
| 6 | | r2’’’’ | |
| 7 | | r3 | |
| 8 | | r3’ | |
| 9 | | r3’’ | |
| 10 | | r3’’’ | |
| 11 | | r3’’’’ | |

# SSA Conditional Constant Propagation: example



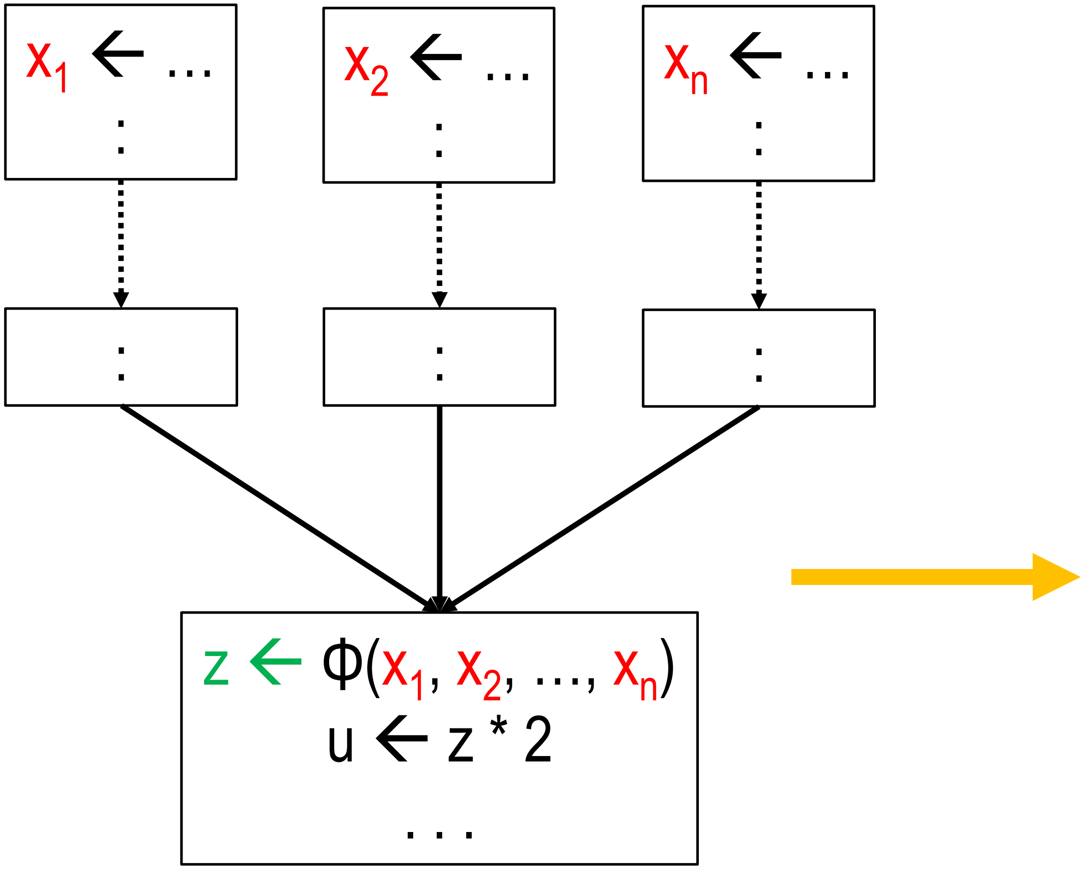

Next: eliminate Φ-functions: easy in this case - map all versions of r3 to r3

# SSA Conditional Constant Propagation: example



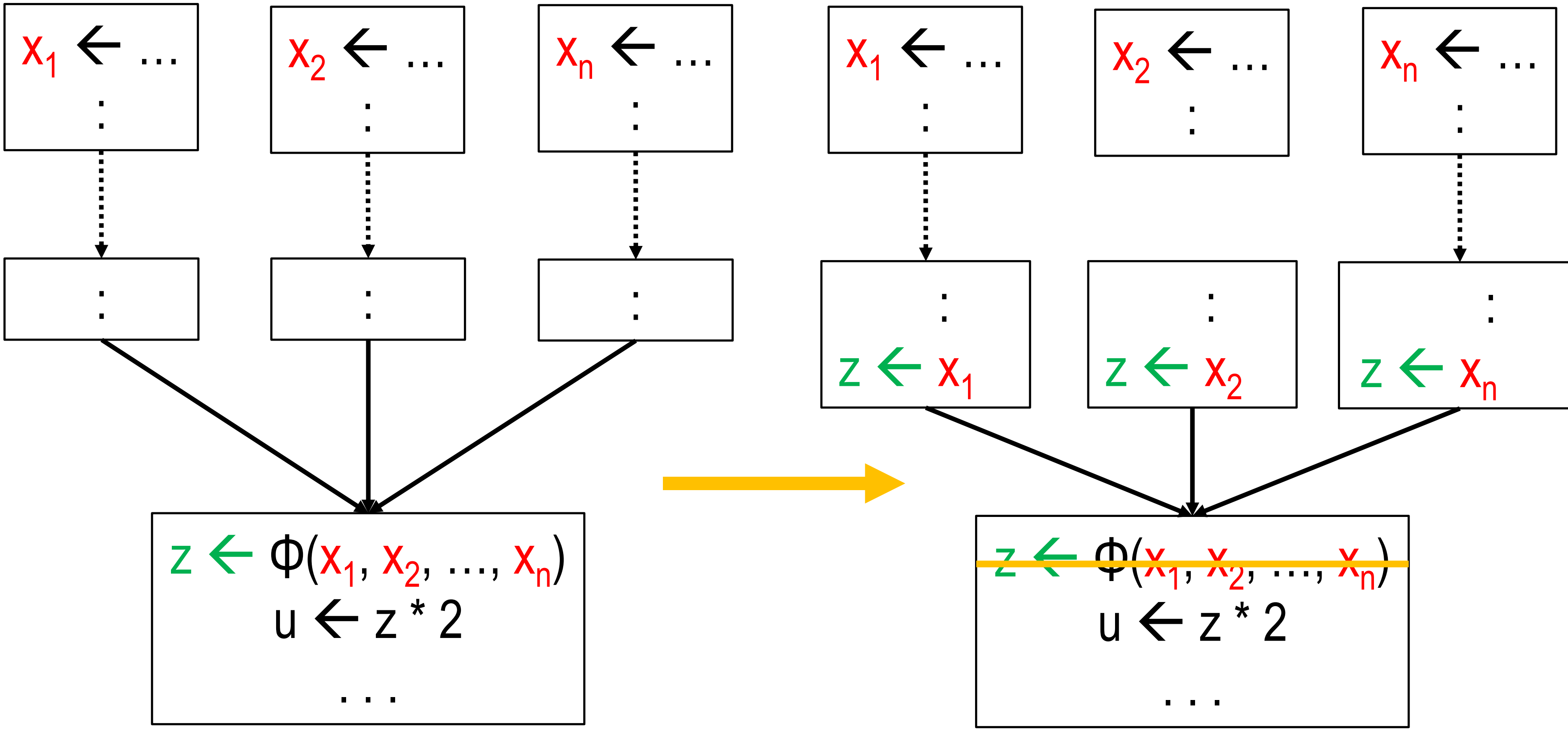
# Translating out of SSA: elimination of Φ-functions

Intuitive interpretation of Φ-functions suggests insertion of move instructions at the end of immediate control flow predecessors



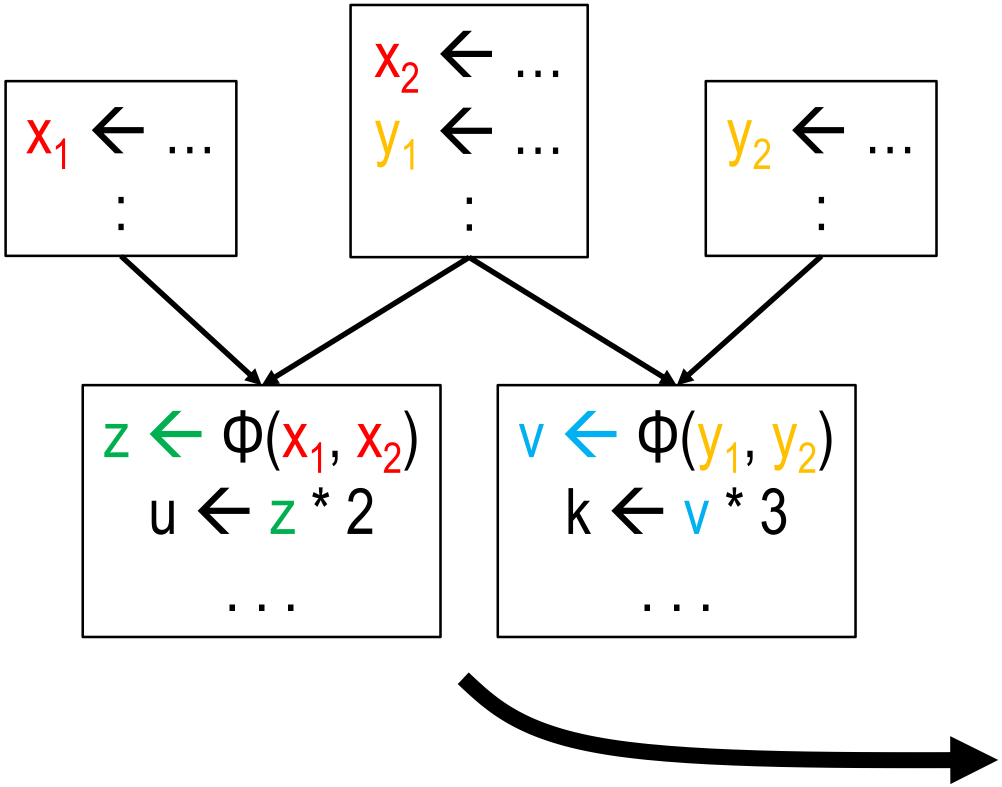
# Translating out of SSA: elimination of Φ-functions

Intuitive interpretation of Φ-functions suggests insertion of move instructions at the end of immediate control flow predecessors

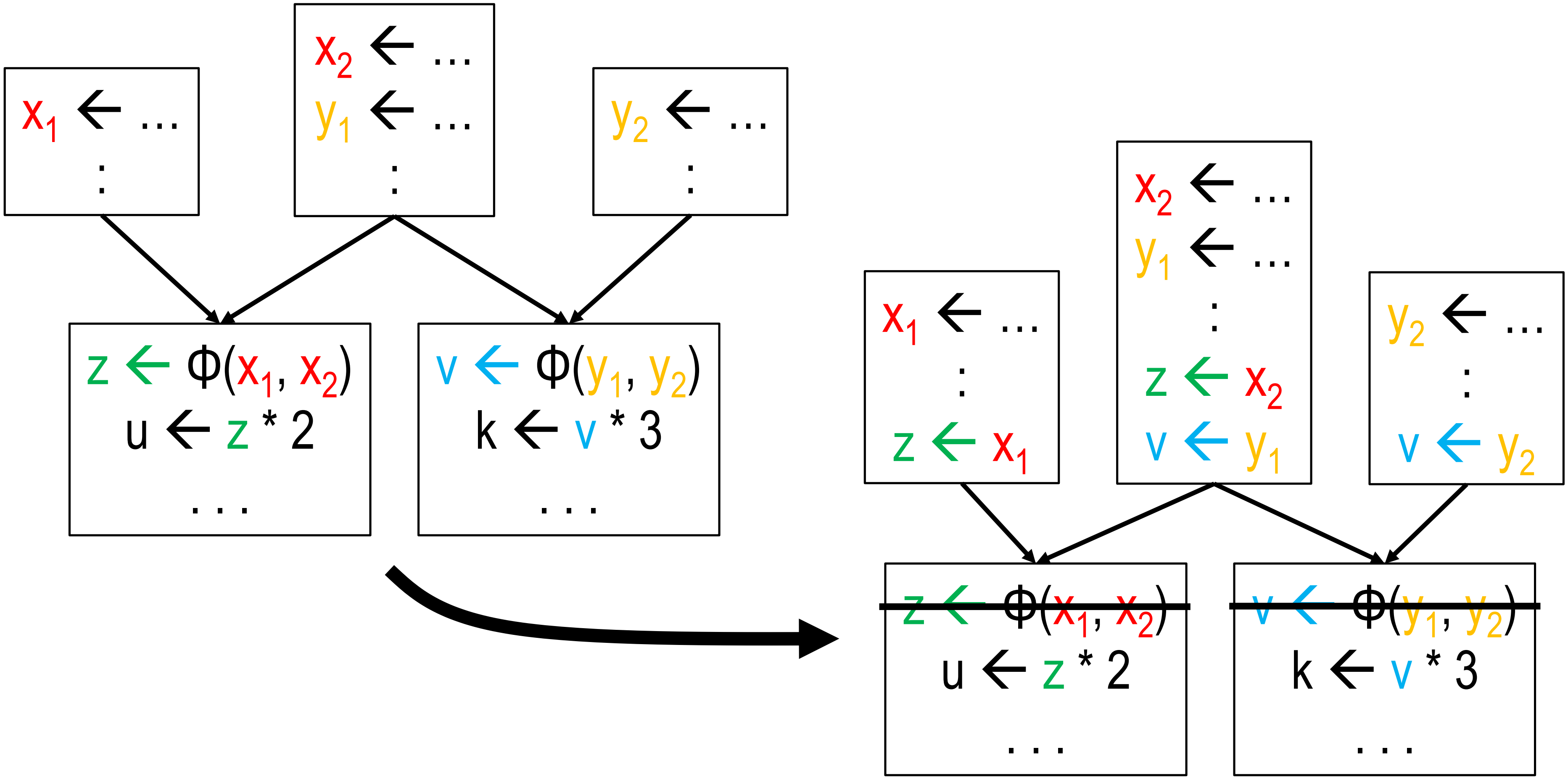



Then rely on register allocator to coalesce / eliminate moves when possible.

# Translating out of SSA -- issue I

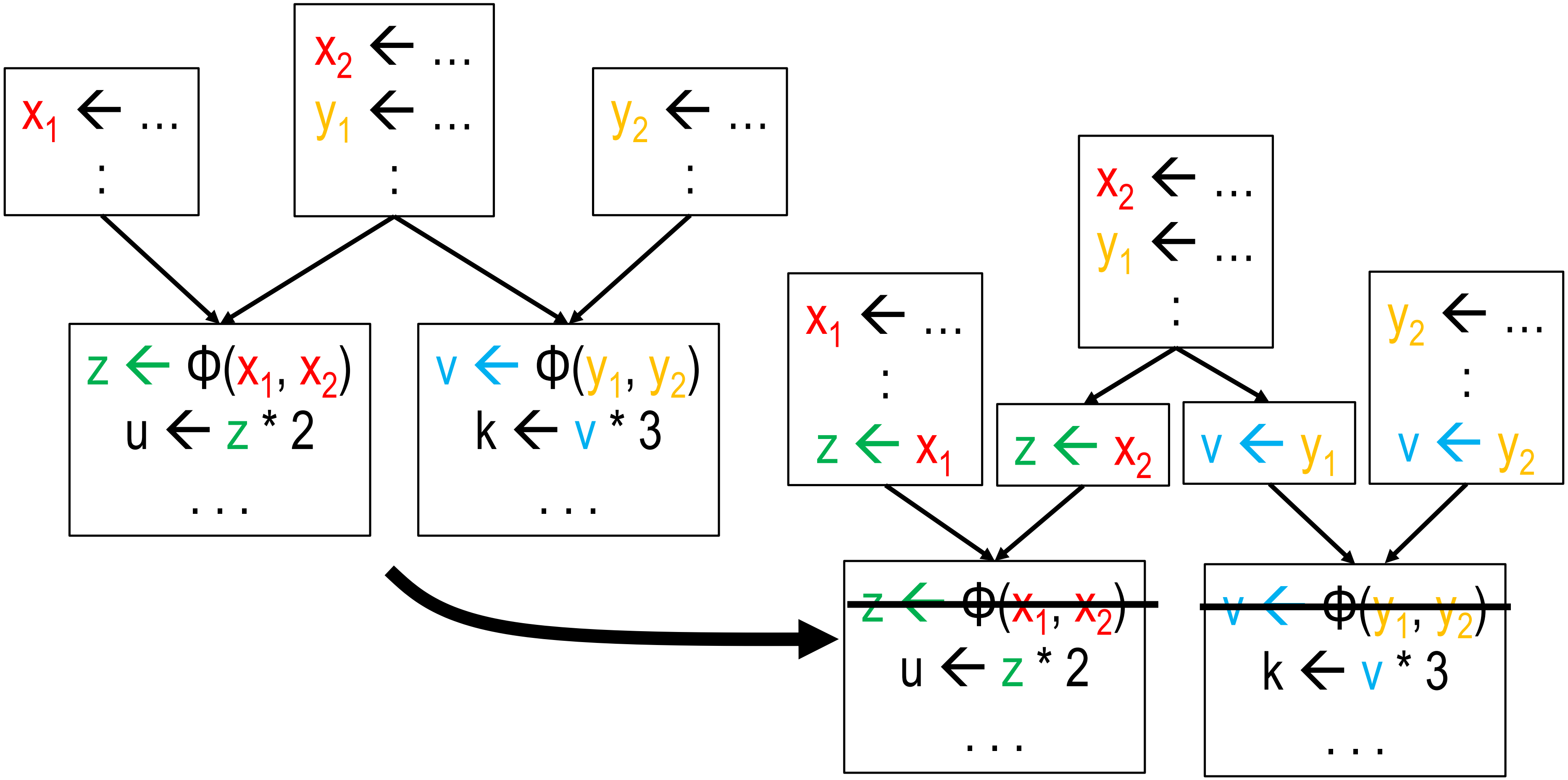


# Translating out of SSA -- issue I



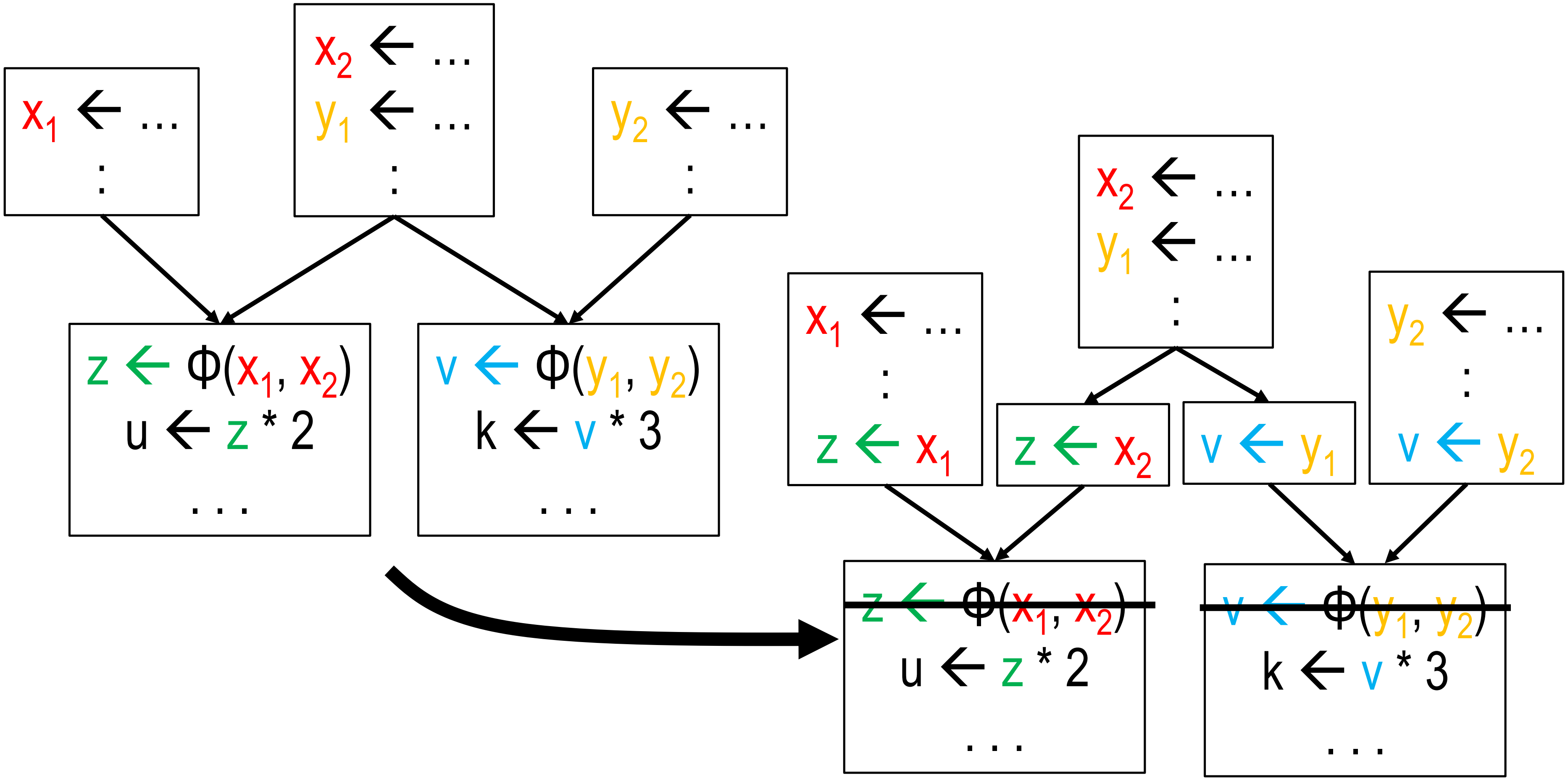

Move instructions pile up in blocks with multiple successors – they're not dead.

# Translating out of SSA -- issue I




**Solution**: place move instructions "in the CFG edge", in a new basic block, whenever predecessor block has several successors.

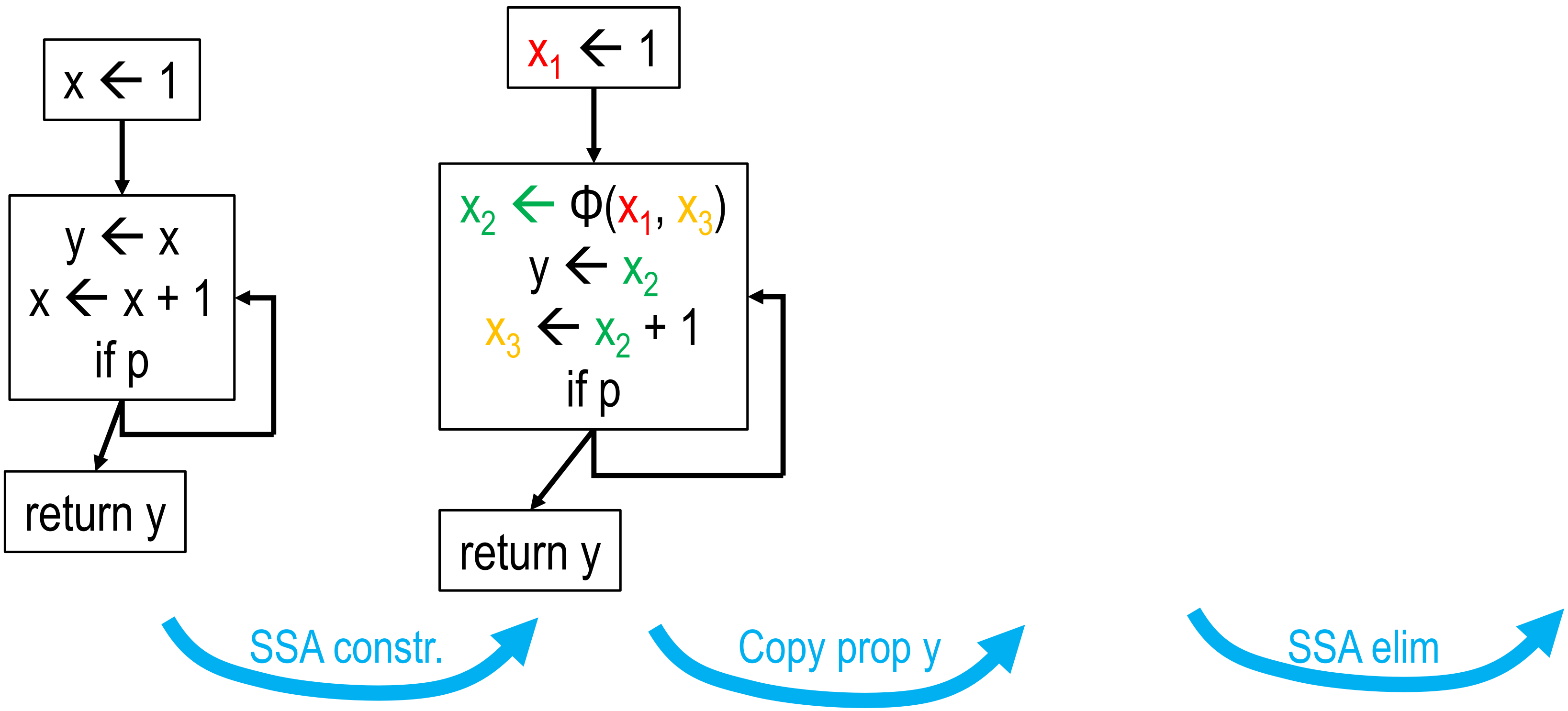
# Translating out of SSA -- issue I



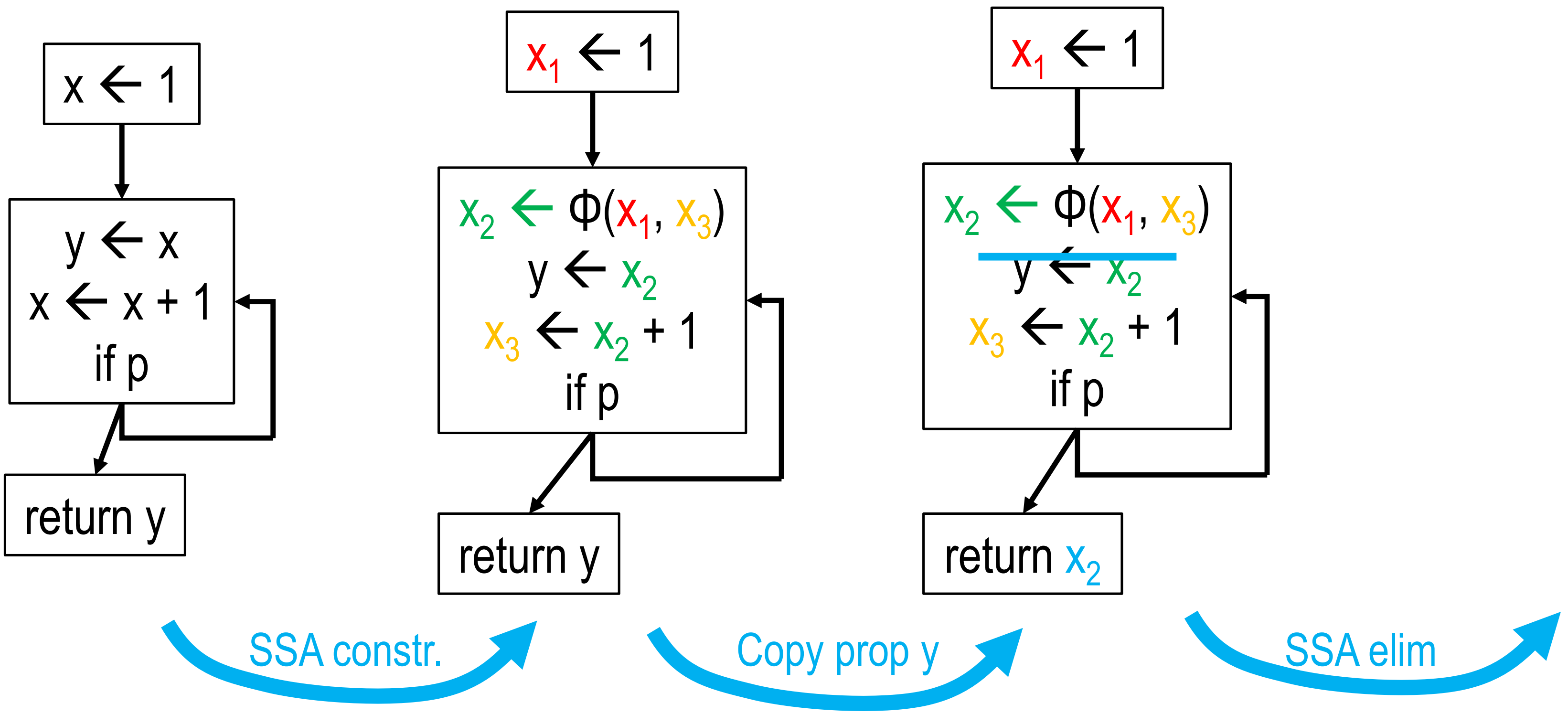

"**Edge-split SSA form**": each CFG edge is either its source block's only out-edge or its sink block's only in-edge.

Easy to achieve during SSA construction: add empty blocks.

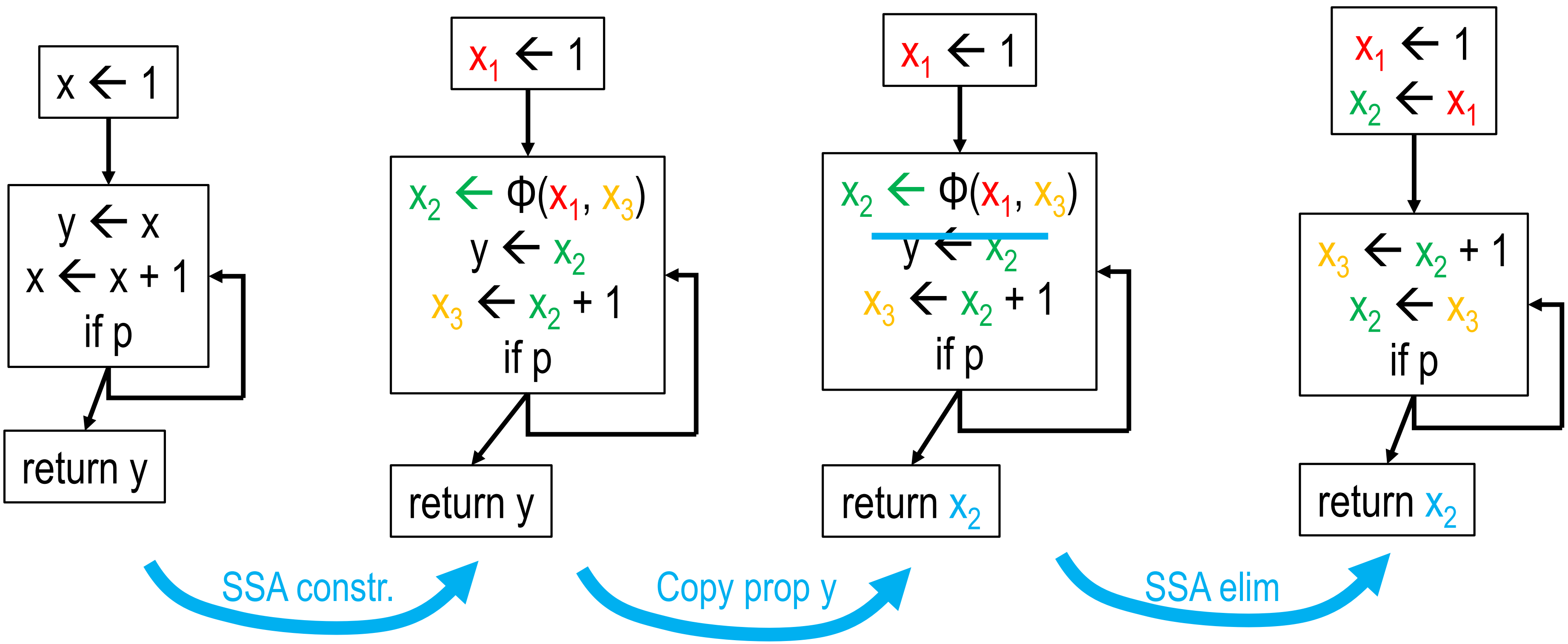
# More motivation for edge splitting: “lost copy” problem



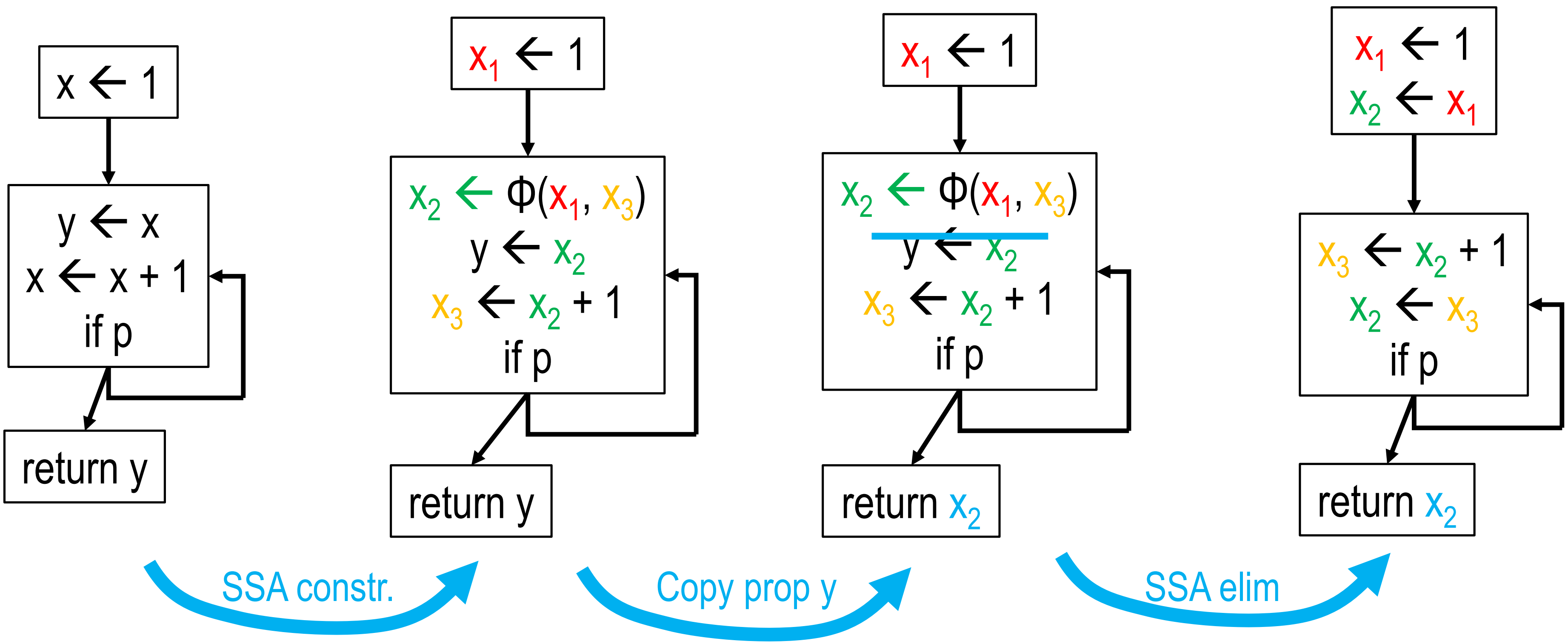
# More motivation for edge splitting: "lost copy" problem



# More motivation for edge splitting: "lost copy" problem

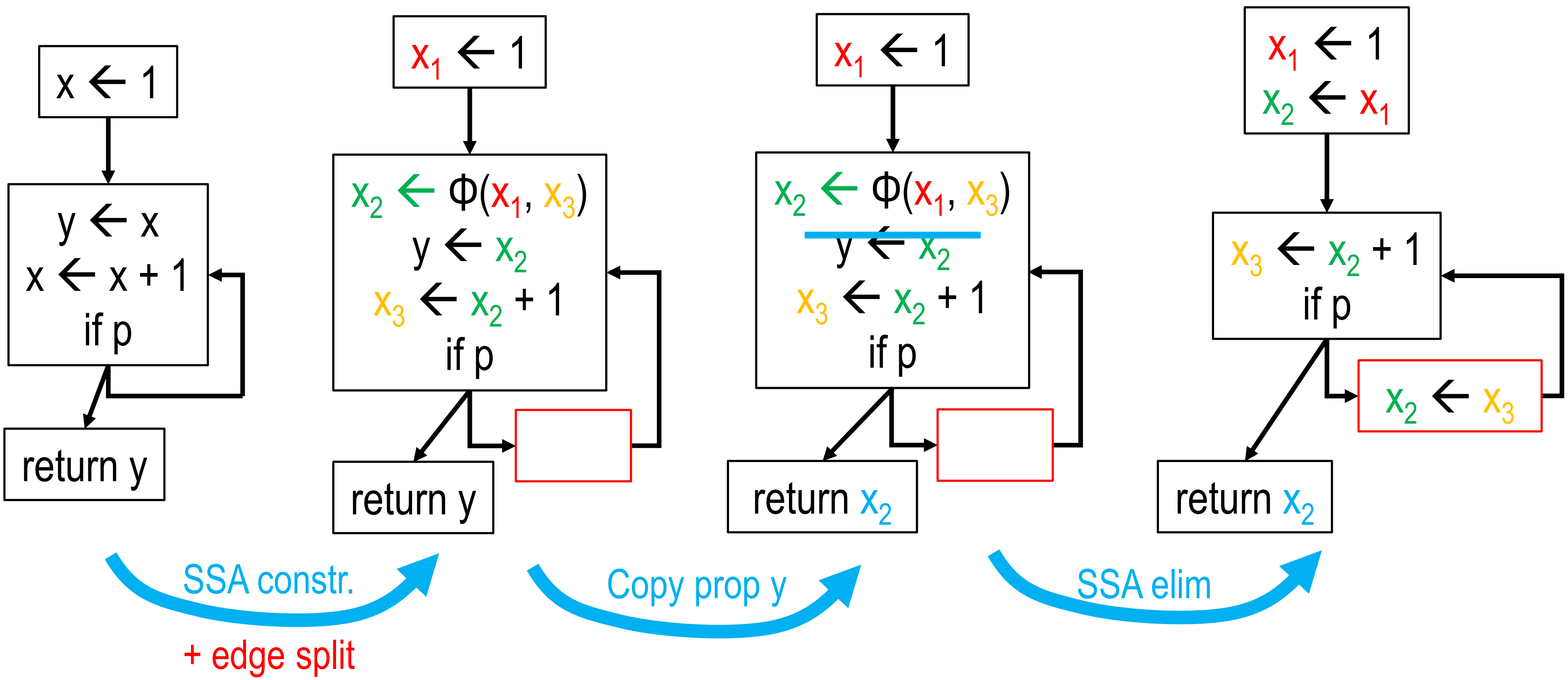


# More motivation for edge splitting: “lost copy” problem




Incorrect result: copy propagation + Φ-elimination incompatible.

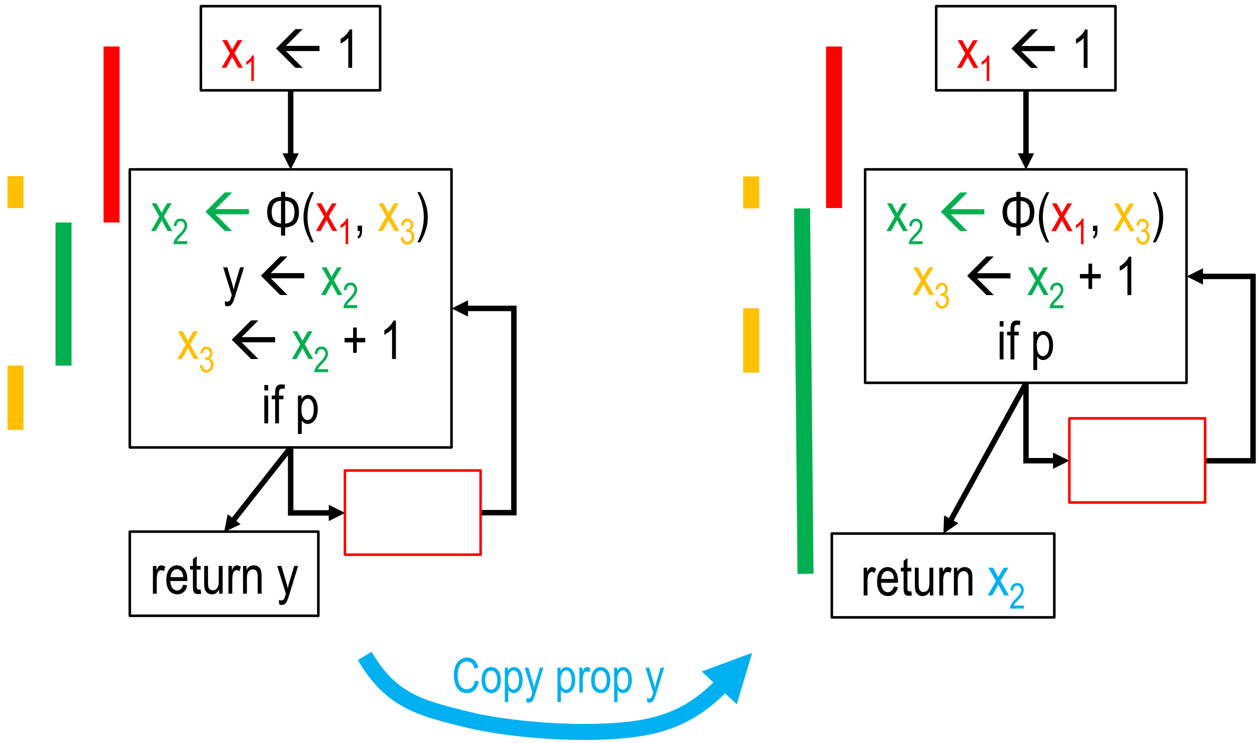
# More motivation for edge splitting: "lost copy" problem




Edge split makes copy propagation + Φ-elimination compatible.

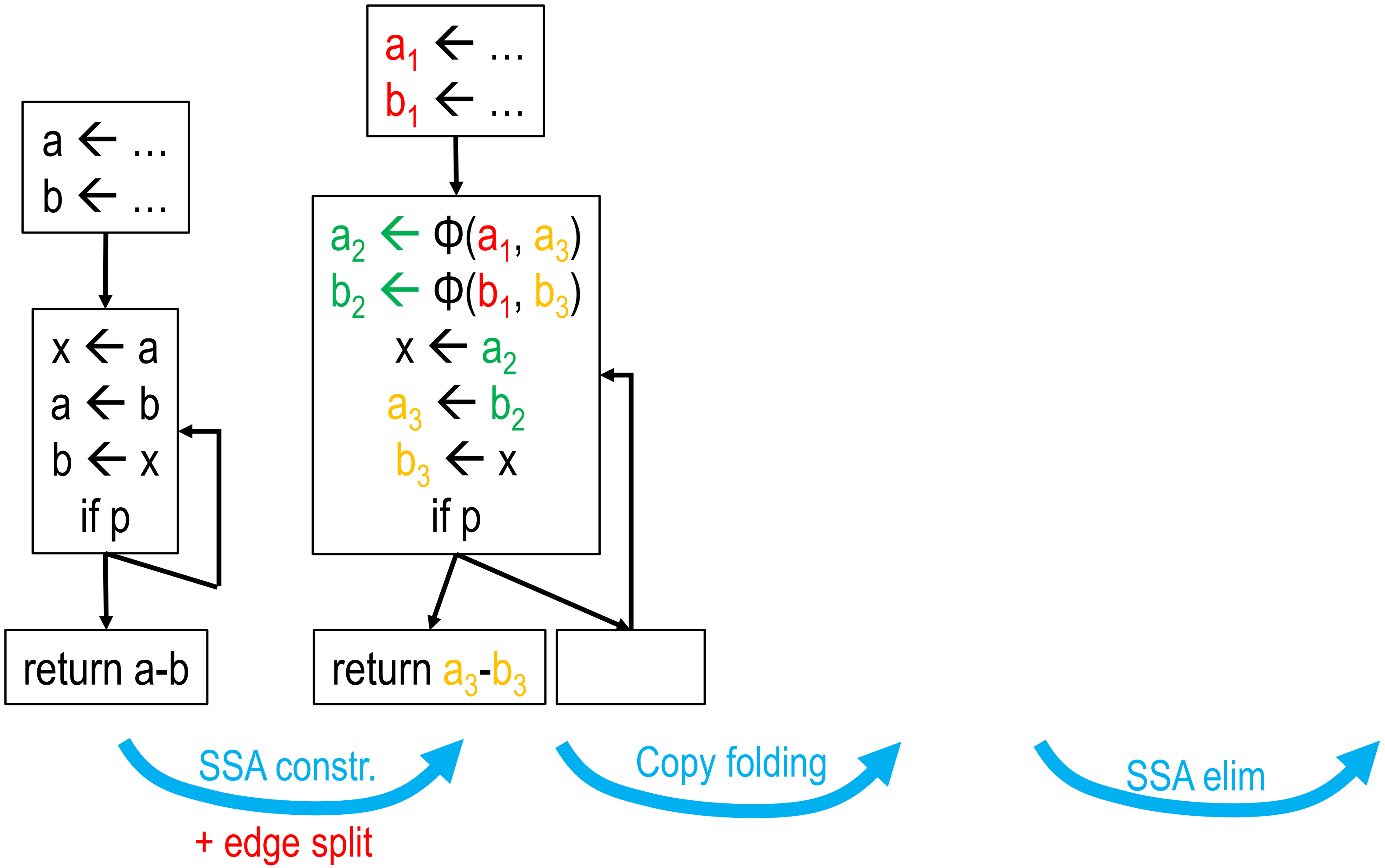
# More motivation for edge splitting: "lost copy" problem

Root cause: copy propagation (and other transformations) potentially alter liveness ranges, so that the ranges of different SSA-versions $\mathbf{x_i}$ of a source-program variable **x** are not any longer distinct.

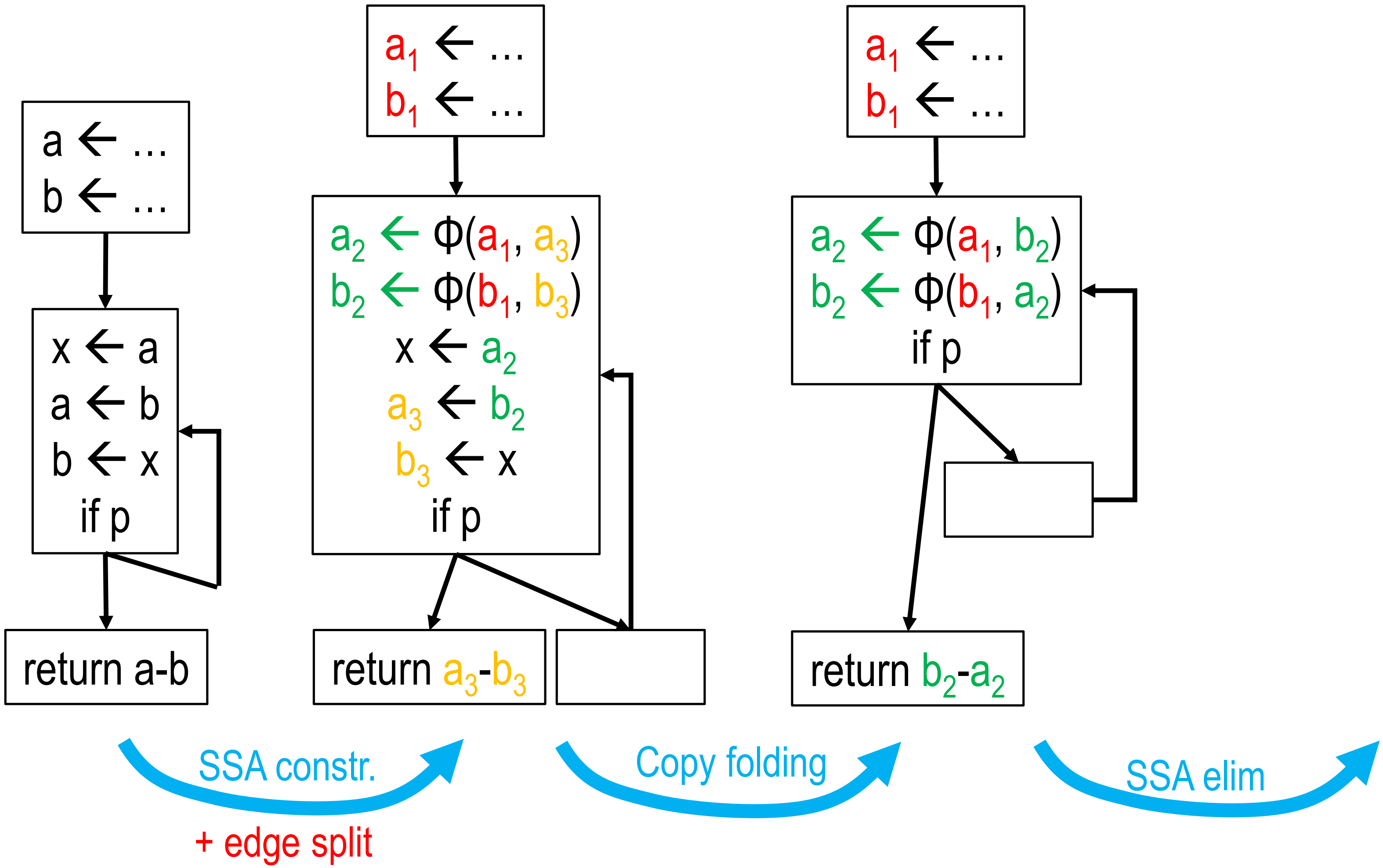



After SSA construction, different "versions" $\mathbf{x_i}$ of a source-program variable **x** are "first-class citizens", unrelated to each other or to **x**.

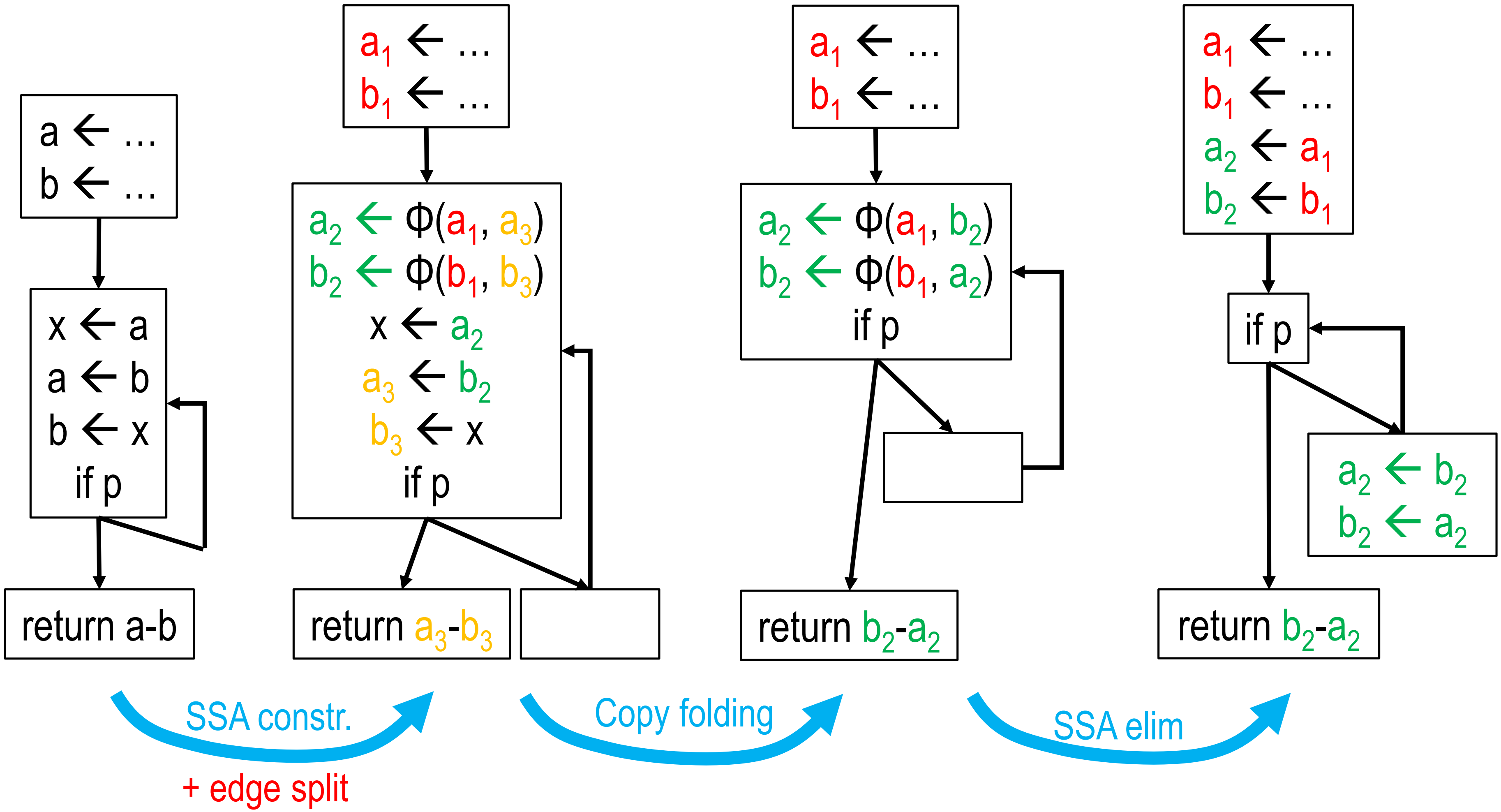
# Translating out of SSA -- issue II: “swap problem”



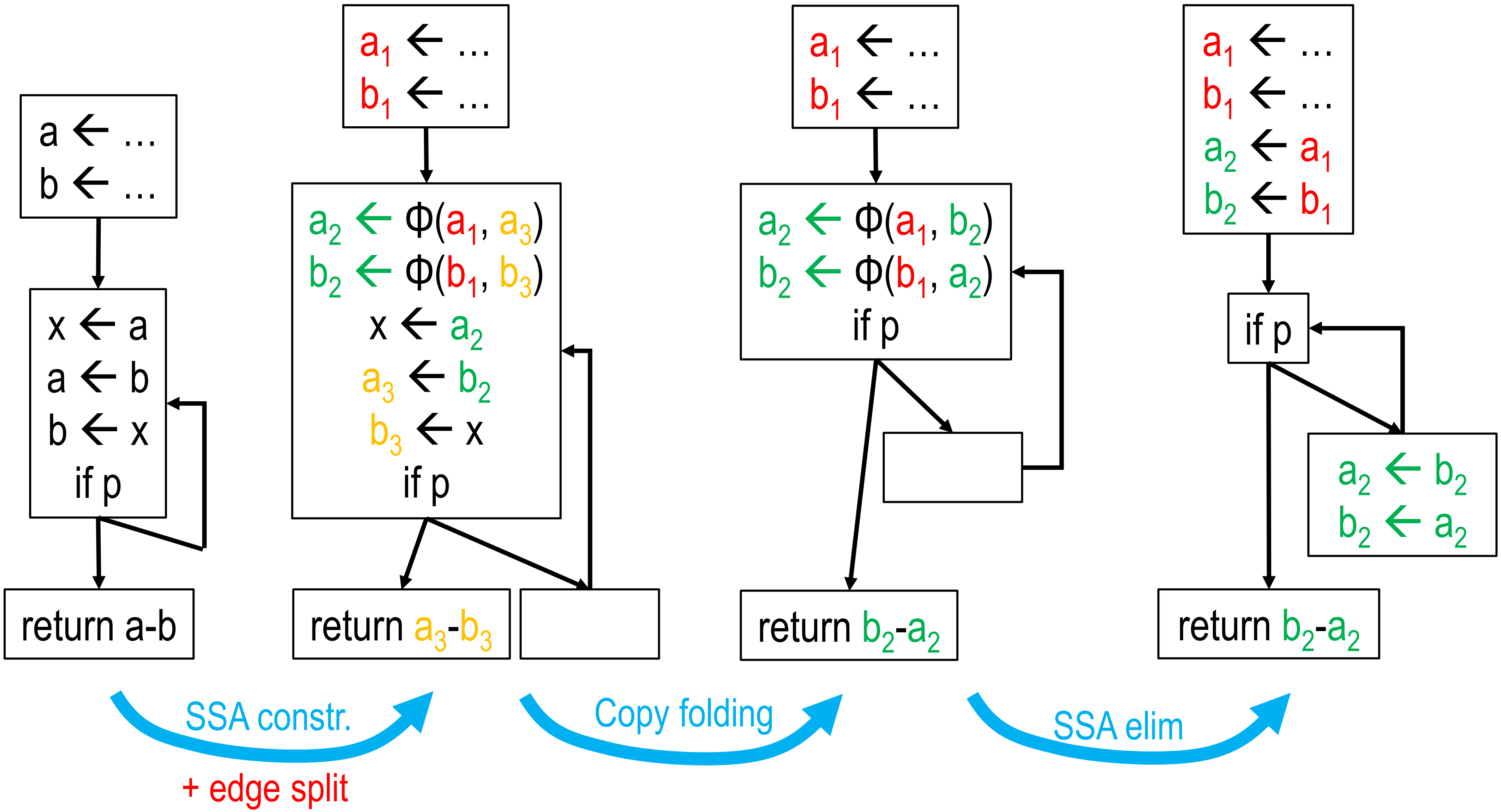
# Translating out of SSA -- issue II: “swap problem”



# Translating out of SSA -- issue II: “swap problem”



# Translating out of SSA -- issue II: “swap problem”



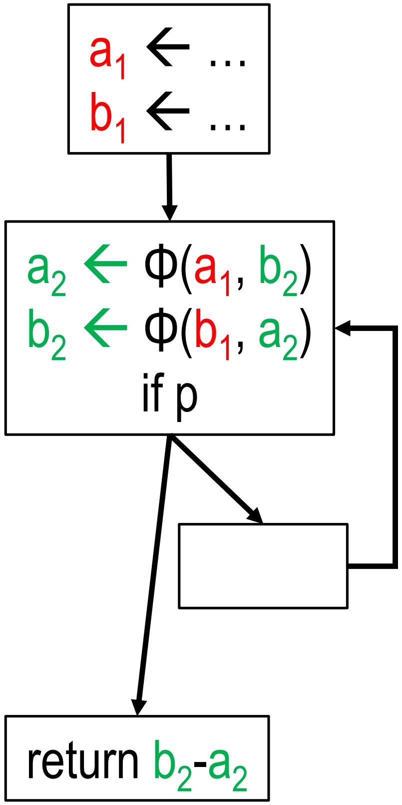

Incorrect result: copy folding + Φ-elimination incompatible.

**p true**: correct result

**p false**: a and b are identified in first loop iteration, so $b_2$=$a_2$ holds upon loop exit, so return value is 0.

# Translating out of SSA -- issue II: “swap problem”

Root cause: the moves should “execute in parallel”, ie **first** read their RHS, then assign to the LHS variables in **parallel**!




Φ-functions in a basic block should be considered a single Φ-block, of concurrent assignment, so that the relative order of Φ-functions is irrelevant:

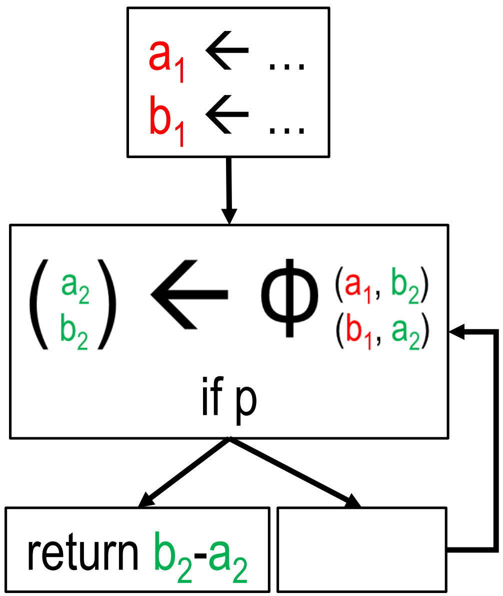
$$\begin{pmatrix} a_2 \\ b_2 \end{pmatrix} \leftarrow \Phi \begin{matrix} (a_1, b_2) \\ (b_1, a_2) \end{matrix}$$

# Translating out of SSA -- issue II: “swap problem”

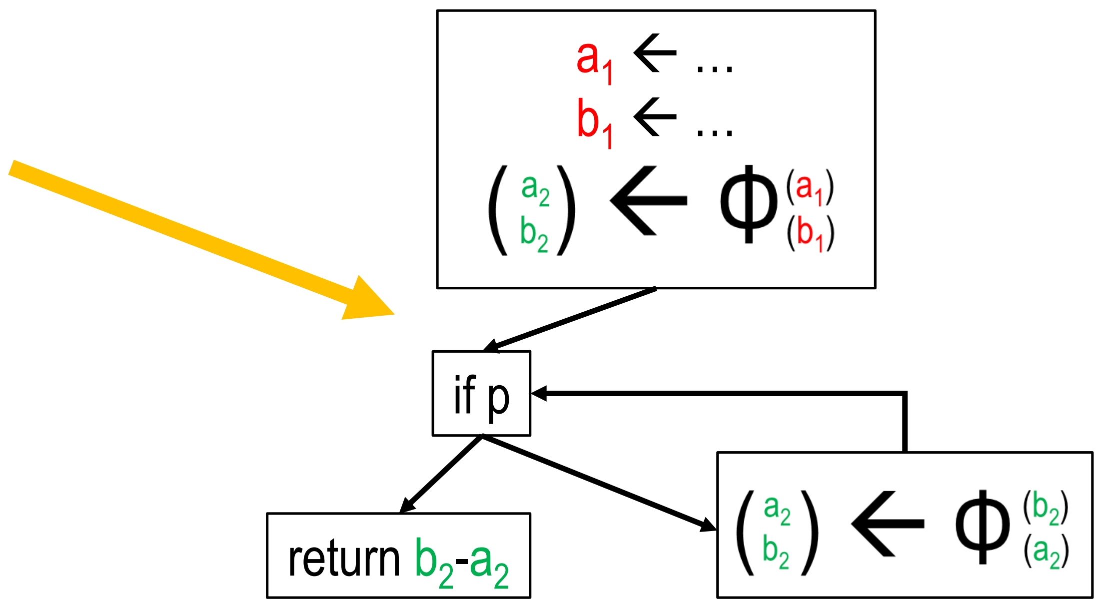
The Φ-functions in a basic block should be considered concurrent – as a single Φ-block:

$$\begin{pmatrix} a_2 \\ b_2 \end{pmatrix} \leftarrow \Phi \begin{matrix} (a_1, b_2) \\ (b_1, a_2) \end{matrix}$$

And replacement of Φ by moves should respect this interpretation.

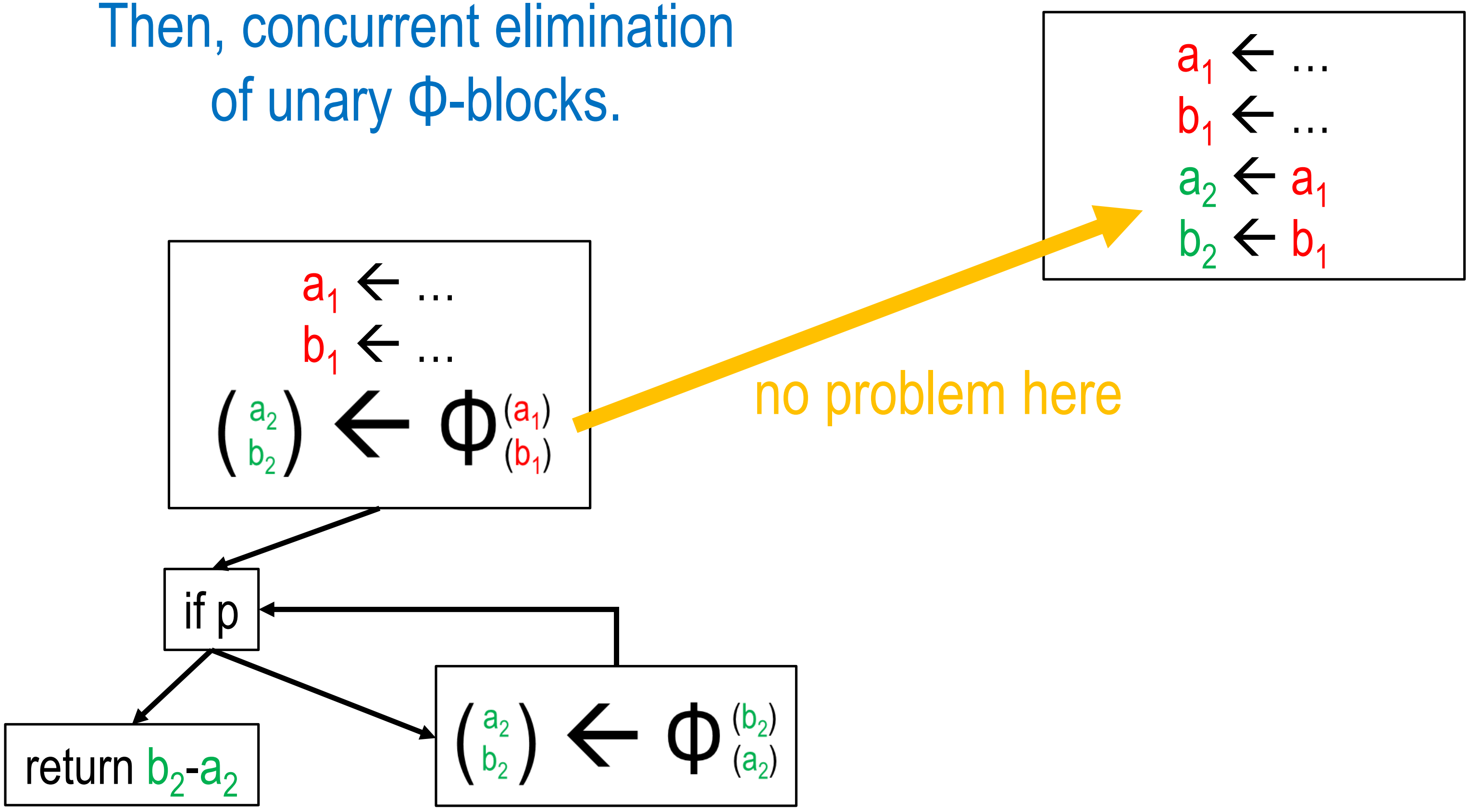



Conceptual intermediate step: unary Φ-blocks at the end of the CFG predecessors / in the incoming CFG edges.



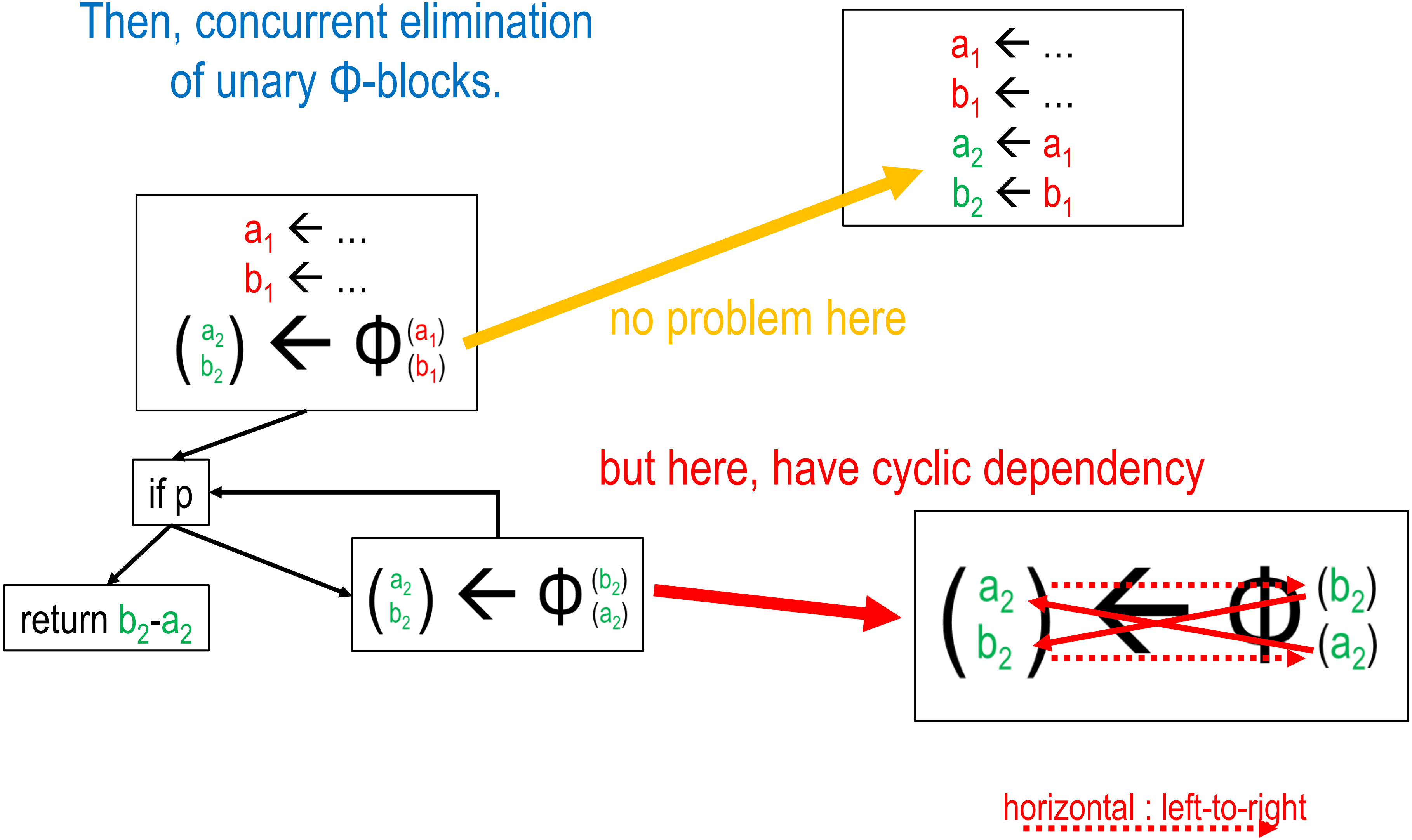
# Translating out of SSA -- issue II: “swap problem”

Then, concurrent elimination of unary Φ-blocks.



# Translating out of SSA -- issue II: "swap problem"

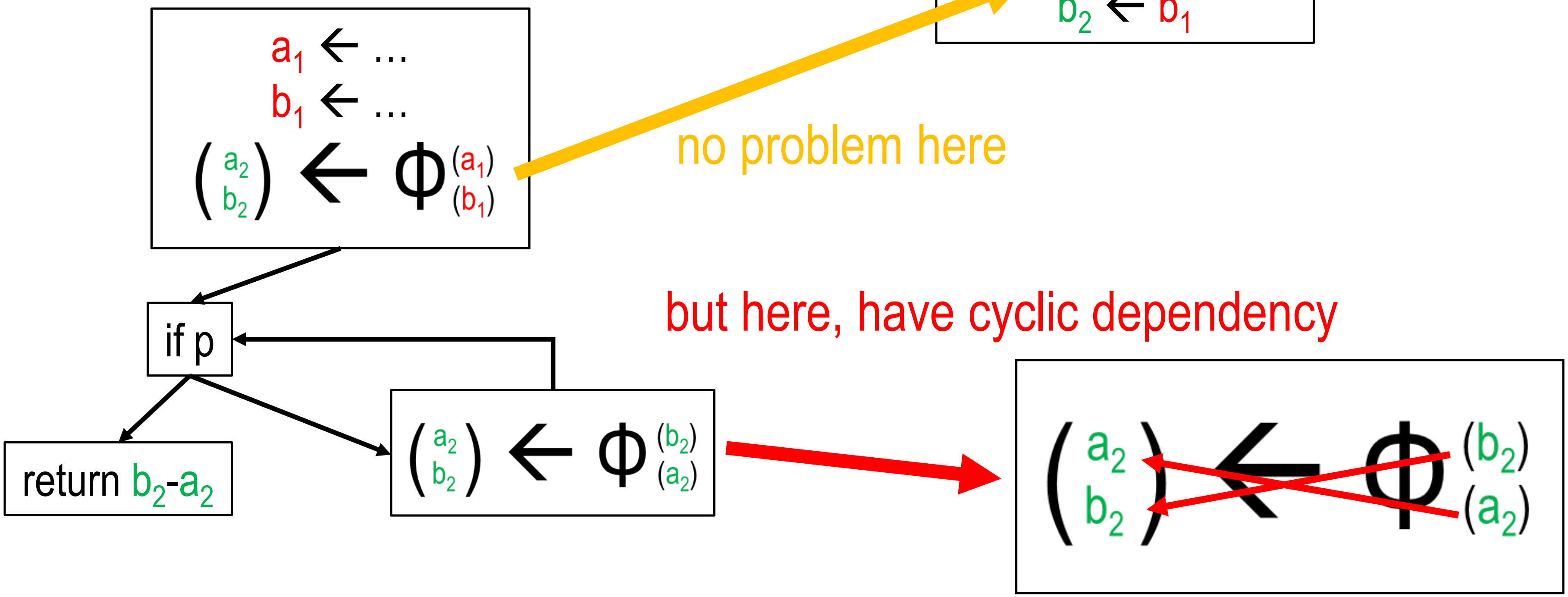
Then, concurrent elimination of unary Φ-blocks.



# Translating out of SSA -- issue II: “swap problem”

Then, concurrent elimination of unary Φ-blocks.

$$a_1 \leftarrow \ldots$$
$$b_1 \leftarrow \ldots$$
$$a_2 \leftarrow a_1$$
$$b_2 \leftarrow b_1$$

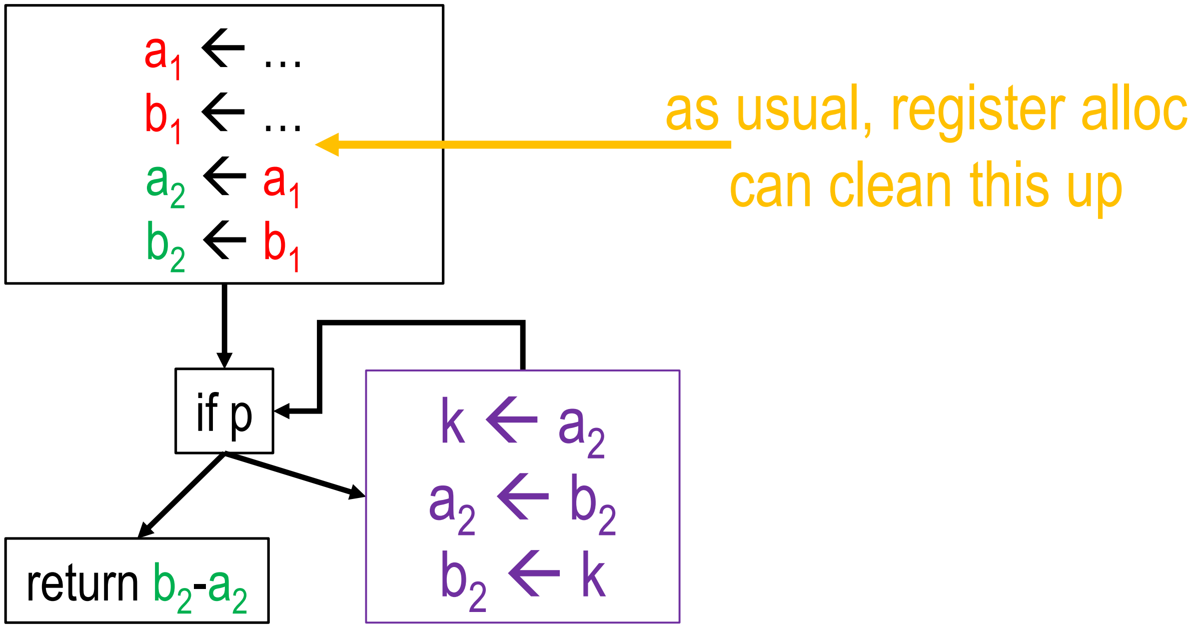



Breaking dependence cycle into sequence of move instructions requires an additional variable.

$$k \leftarrow a_2$$
$$a_2 \leftarrow b_2$$
$$b_2 \leftarrow k$$

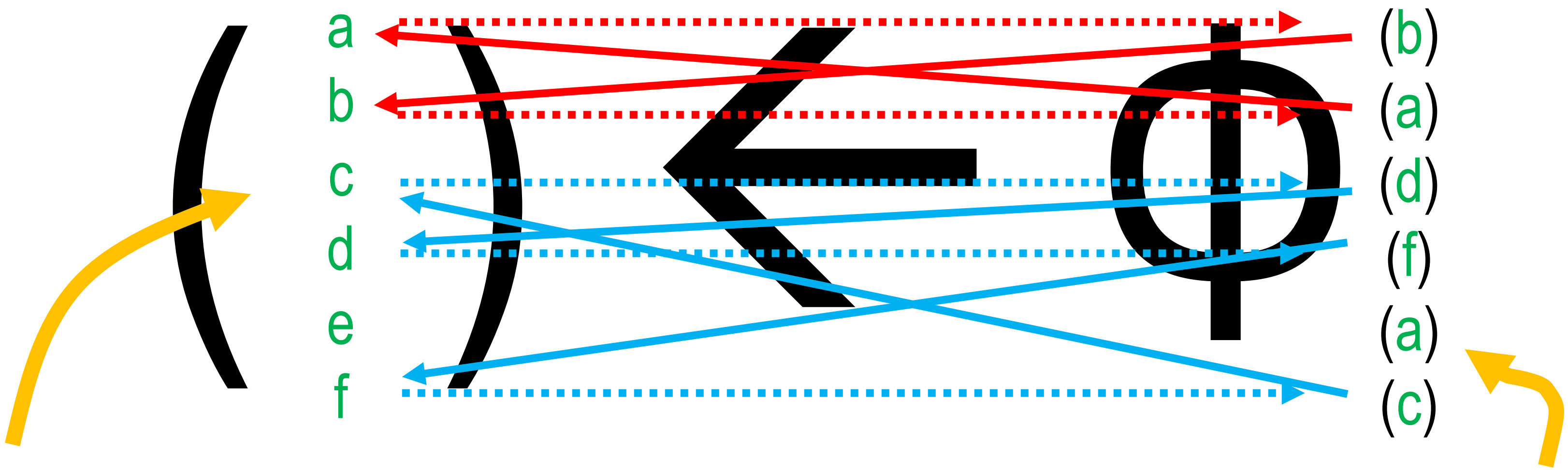
# Translating out of SSA -- issue II: “swap problem”

Resulting code has correct behavior, for p=true and p=false.



# Translating out of SSA -- issue II: “swap problem”

In general, the variables in a (unary) Φ-block can form multiple (non-overlapping) cycles, of different length.




New (implicit) sanity condition of SSA: LHS variables should be distinct!

Variables may occur repeatedly in RHS – but only participate in one cycle.

The cycles can be broken in succession, so the single additional variable/register **k** can be reused!

The moves not involved in a cycle (like e ← a) are emitted first.

# Translating out of SSA -- discussion

Some care is needed to avoid lost copies and the swap problem, but basic principle – manifest the intuitive meaning of Φ-functions by locally inserting copy instructions "in the incoming edges" – works fine.

Alternative: globally identify groups of variables that can be unified

- first guess - the original variables: works fine, until aggressive optimizations yield overlapping liveness ranges etc.

- Φ-congruence classes (Sreedhar et al., *Translating out of static single assignment form*. 6th Static Analysis Symposium, LNCS 1694, Springer, 1999)

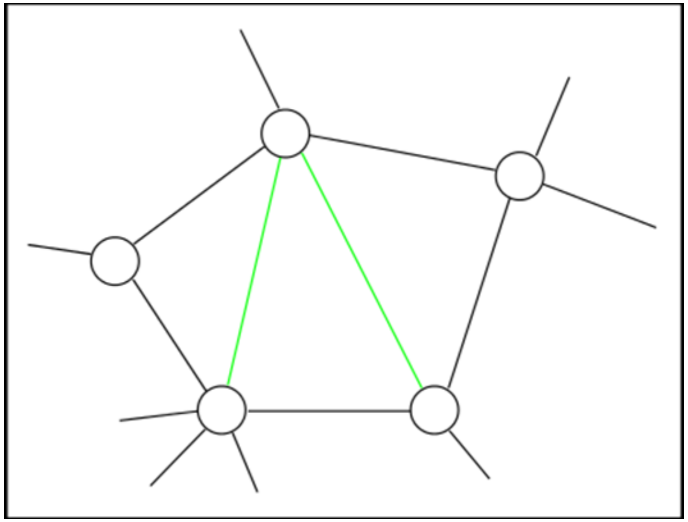
Insertion of moves, effect on liveness ranges, etc suggest exploration of interaction between SSA and register allocation

# SSA and register allocation

S. Hack et al., *Register allocation for programs in SSA form. 15th Conference on Compiler Construction (CC'06), LNCS 3923, Springer, 2006*

Interference graphs of SSA programs are **chordal** graphs.

Any cycle of > 3 vertices has a *chord*, i.e. an edge that is not part of the cycle but connects two of its vertices.



Key properties of chordal graphs:

1. their chromatic number is equal to the size of the largest clique
2. they can be optimally colored in **quadratic** time (w.r.t. number of nodes)

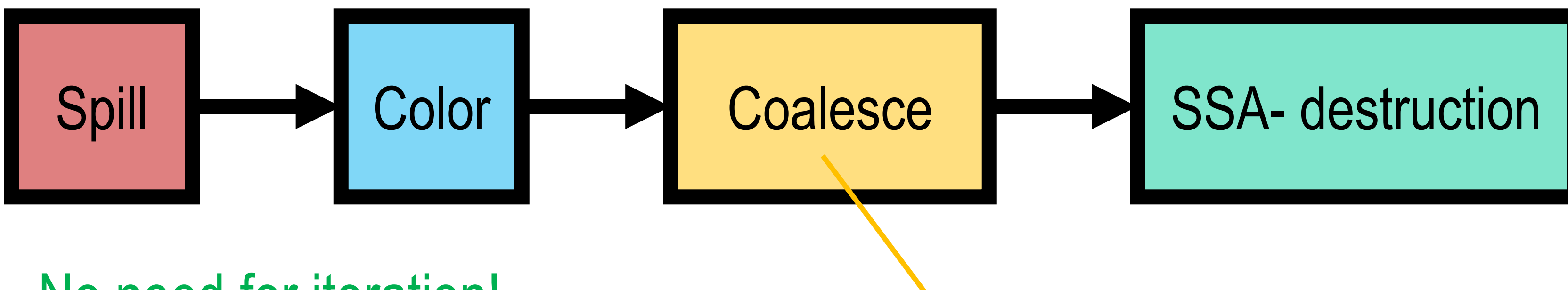
# SSA and register allocation

S. Hack et al., *Register allocation for programs in SSA form. 15th Conference on Compiler Construction (CC'06), LNCS 3923, Springer, 2006*

**Also**: the largest clique in the interference graph of an SSA program P is locally manifest in P: there is at least one instruction $i_P$ where all members of the clique are live.

Can hence traverse program and obtain required number of colors – and know which variables to spill/coalesce in case we don't have this many registers.

Resulting approach to register allocation:




No need for iteration!

Don't merge nodes in G, but share reg for variables in a Φ-node.

In ordinary programs, iteration was needed since spilling/coalescing was not guaranteed to reduce the number of colors needed. For SSA, this is guaranteed, if we spill/coalesce variables live at $i_P$.

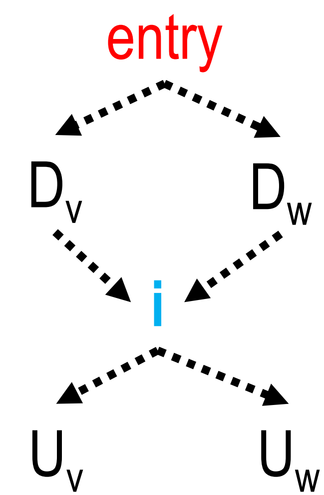
# SSA and register allocation: Hack et al.'s result

Remember: interference graph of an SSA program P

- interference graph: G=(V, E) where
  nodes V: program variables
  edges E: (v, w) є E if there is a program point at which v and w are both live
- SSA: each use of a variable v is dominated by the (unique) definition $D_v$ of v

Lemma 1: if v and w interfere, either $D_v$ dominates $D_w$, or $D_w$ dominates $D_v$.

Idea: Let i be the instruction at which v and w both live.
Thus, there are paths i $\dashrightarrow$ $U_v$ and i $\dashrightarrow$ $U_w$
to some uses of v and w. As $U_v$ is dominated by $D_v$, there
is a path $D_v$ $\dashrightarrow$ i. Similarly, there is a path
from $D_w$ to i. Hence, entry $\dashrightarrow$ $D_v$ $\dashrightarrow$ i $\dashrightarrow$ $U_w$
must contain $D_w$, and entry $\dashrightarrow$ $D_w$ $\dashrightarrow$ i $\dashrightarrow$ $U_v$
must contain $D_v$. From this obtain claim…



# SSA and register allocation: Hack et al.'s result

Lemma 1: if v and w interfere, either $D_v$ dominates $D_w$, or $D_w$ dominates $D_v$.

Lemma 2: if v and w interfere and $D_v$ dominates $D_w$, then v is live at $D_v$.

# SSA and register allocation: Hack et al.'s result

Lemma 1: if v and w interfere, either $D_v$ dominates $D_w$, or $D_w$ dominates $D_v$.

Lemma 2: if v and w interfere and $D_v$ dominates $D_w$, then v is live at $D_v$.

**Theorem 1**: Let C = $\{c_1, \ldots c_n\}$ be a clique in G, ie $(c_i,c_j) \in E$ forall $i \neq j$. Then, there is a label in P where $c_1, \ldots, c_n$ are all live.

Proof :

- by Lemma 1, the nodes $c_1, \ldots c_n$ are totally ordered by the dominance relationship: $c_{\sigma(1)}, \ldots, c_{\sigma(n)}$ for some permutation σ of {1, ..n}
- as dominance is transitive, all $c_{\sigma(i)}$ dominate $c_{\sigma(n)}$
- by Lemma 2, all $c_{\sigma(i)}$ are hence all live at $c_{\sigma(n)}$.

# SSA and register allocation: Hack et al.'s result

- we color nodes by stack-based simplify-select (cf Kempe).
- suppose we can simplify nodes in a **perfect elimination order:** when a node is removed, its remaining neighbors form a clique
- then, when we reinsert the node, we again have a clique
- the size of the latter clique is bound by $\omega(G)$, the size of G' largest clique

# SSA and register allocation: Hack et al.'s result

- we color nodes by stack-based simplify-select (cf Kempe).
- suppose we can simplify nodes in a **perfect elimination order:** when a node is removed, its remaining neighbors form a clique
- then, when we reinsert the node, we again have a clique
- the size of the latter clique is bound by ω(G), the size of G' largest clique

**Theorem 2**: G admits simplification by a PEO.

(admitting simplification by PEO is equivalent to being chordal)

# SSA and register allocation: Hack et al.'s result

- we color nodes by stack-based simplify-select (cf Kempe).
- suppose we can simplify nodes in a **perfect elimination order:** when a node is removed, its remaining neighbors form a clique
- then, when we reinsert the node, we again have a clique
- the size of the latter clique is bound by ω(G), the size of G' largest clique

**Theorem 2**: G admits simplification by a PEO.

(admitting simplification by PEO is equivalent to being chordal)

**Theorem 3**: Chordal graphs are perfect:
max colors needed = size of the largest clique

Thus, we can color G (using a PEO) using ω(G) many colors, and P contains an instruction where ω(G) variables are live (and no instruction with more).

Thus: can traverse P, search for largest **local** live-set, and obtain #registers.

# SSA and functional programming

SSA:

- each variable has a unique site of **def**inition; different uses of the same source-program variable name are disambiguated
- the **def**-site **dominates** all **use**s
- in straight-line code, each variable is assigned to only **once**

# SSA and functional programming

SSA:

- each variable has a unique site of **def**inition; different uses of the same source-program variable name are disambiguated
- the **def**-site **dominates** all **use**s
- in straight-line code, each variable is assigned to only **once**

Functional code:

- each name has a unique site of **binding**: **let** x = $e_1$ in $e_2$; different uses of the same name are kept apart by the language definition, or can be explicitly disambiguated by α-renaming
- the **binding**-site determines a **scope** that contains all **uses**
- in straight-line code, the value to which a name is bound is **never changes**

# SSA and functional programming

SSA:

- each variable has a unique site of **def**inition; different uses of the same source-program variable name are disambiguated
- the **def**-site **dominates** all **use**s
- in straight-line code, each variable is assigned to only **once**

Functional code:

- each name has a unique site of **binding**: **let** x = $e_1$ in $e_2$; different uses of the same name are kept apart by the language definition, or can be explicitly disambiguated by α-renaming
- the **binding**-site determines a **scope** that contains all **uses**
- in straight-line code, the value to which a name is bound **never changes** – and in a recursive function, we're in different stack frames (but see details on stack frames in later lecture).

Andrew W. Appel: *SSA is Functional Programming*. ACM SIGPLAN Notices, April 1998 .

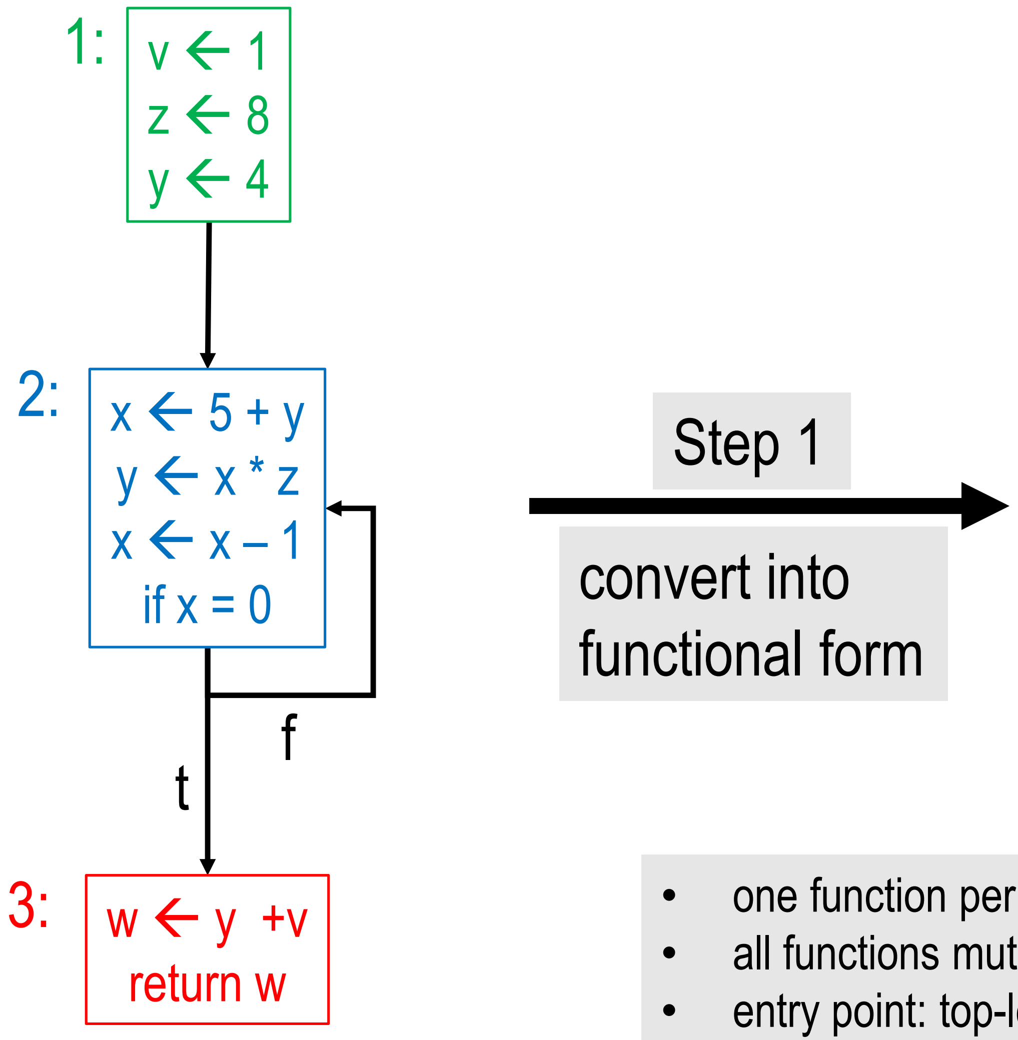
# SSA and functional programming - correspondences

| Functional concept | Imperative/SSA concept |
|---|---|
| variable binding in let | assignment (point of definition) |
| $\alpha$-renaming | variable renaming |
| unique association of binding occurrences to uses | unique association of defs to uses |
| formal parameter of continuation/local function | $\phi$-function (point of definition) |
| lexical scope of bound variable | dominance region |

| Functional concept | Imperative/SSA concept |
|---|---|
| subterm relationship | control flow successor relationship |
| arity of function $f_i$ | number of $\phi$-functions at beginning of $b_i$ |
| distinctness of formal parameters of $f_i$ | distinctness of LHS-variables in the $\phi$-block of $b_i$ |
| number of call sites of function $f_i$ | arity of $\phi$-functions in block $b_i$ |
| parameter lifting/dropping | addition/removal of $\phi$-function |
| block floating/sinking | reordering according to dominator tree structure |
| potential nesting structure | dominator tree |
| nesting level | maximal level index in dominator tree |

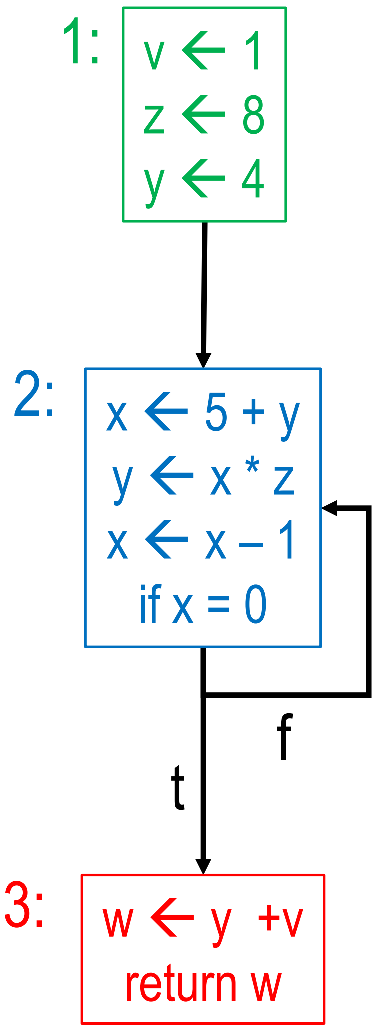
- construction of SSA can be recast as transformation of a corresponding functional program; destruction, too
- latent structural properties of SSA often explicit in FP view
- correctness arguments for SSA analyses & transformations transfer to/from functional view

# SSA construction in functional style




- one function per basic block
- all functions mutually (tail-)recursive
- entry point: top-level initial function call
- function bodies: let-bindings for basic instructions (ANF)
- liveness analysis yields formal parameter and argument lists

# SSA construction in functional style



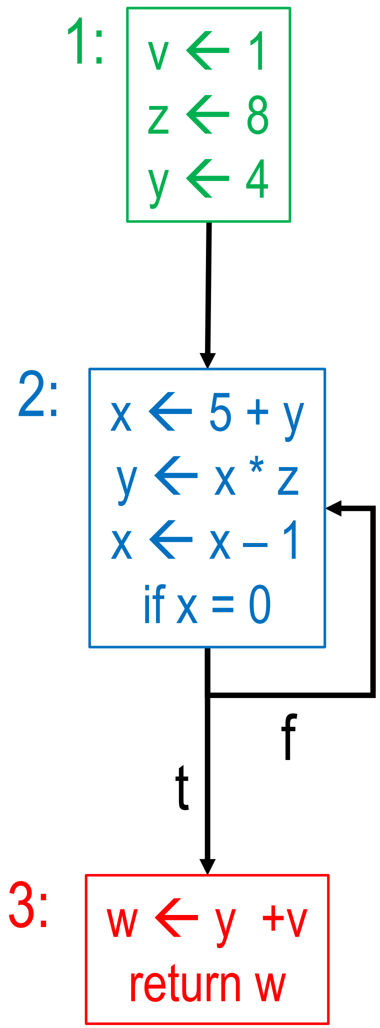

Step 1

convert into functional form

```
let fun f_1() = let val v = 1
                    val z = 8
                    val y = 4
                in f_2(v, z, y) end
and f_2(v ,z, y) = let val x = 5 + y
                       val y = x * z
                       val x = x – 1
                   in if x=0 then f_3(y, v)
                      else f_2(v, z, y) end
and f_3(y, v) = let val w = y + v
                in w end
in f_1() end;
```

- one function per basic block
- all functions mutually (tail-)recursive
- entry point: top-level initial function call
- function bodies: let-bindings for basic instructions (ANF)
- liveness analysis yields formal parameter and argument lists

# SSA construction in functional style




- *all functions closed*
- *variables not globally unique, but uses have unique defs (scope)*

Step 1

convert into functional form

```
let fun f1() = let val v = 1
                   val z = 8
                   val y = 4
               in f2(v, z, y) end
and f2(v ,z, y) = let val x = 5 + y
                      val y = x * z
                      val x = x – 1
                  in if x=0 then f3(y, v)
                     else f2(v, z, y) end
and f3(y, v) = let val w = y + v
               in w end
in f1() end;
```

- one function per basic block
- all functions mutually (tail-)recursive
- entry point: top-level initial function call
- function bodies: let-bindings for basic instructions (ANF)
- liveness analysis yields formal parameter and argument lists

# SSA construction in functional style

let fun $f_1$() = let val v = 1<br>
                  val z = 8<br>
                  val y = 4<br>
              in $f_2$(v, z, y) end<br>
and $f_2$(v, z, y) = let val x = 5 + y<br>
                  val y = x * z<br>
                  val x = x – 1<br>
              in if x=0 then $f_3$(y, v)<br>
                  else $f_2$(v, z, y) end<br>
and $f_3$(y, v) = let val w = y + v<br>
              in w end<br>
in $f_1$() end;

optional

make names unique

- as functions are closed, can rename each function definition individually

# SSA construction in functional style

let fun $f_1$() = let val v = 1
val z = 8
val y = 4
in $f_2$(v, z, y) end
and $f_2$(v, z, y) = let val x = 5 + y
val y = x * z
val x = x – 1
in if x=0 then $f_3$(y, v)
else $f_2$(v, z, y) end
and $f_3$(y, v) = let val w = y + v
in w end
in $f_1$() end;

optional

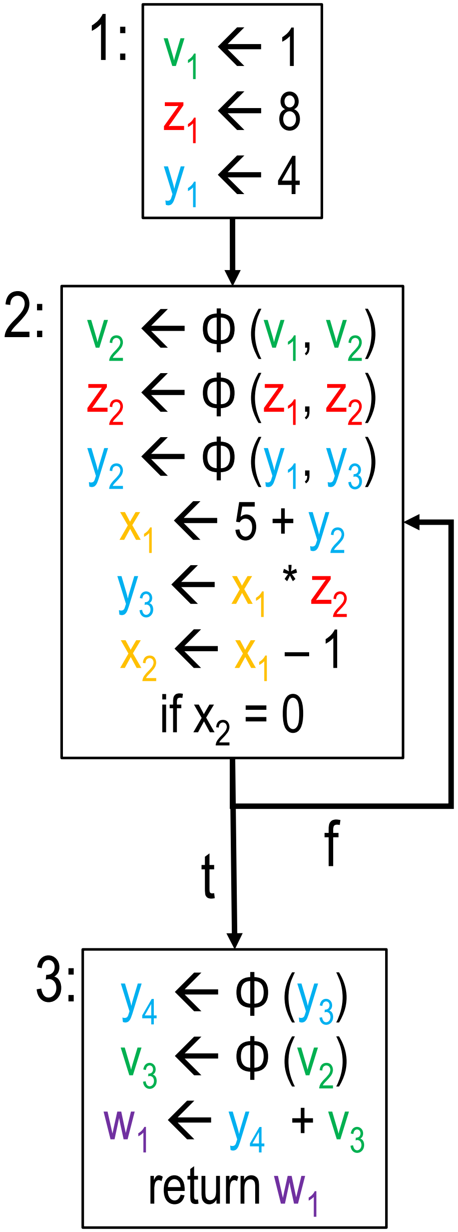
→

make names unique

let fun $f_1$() = let val $v_1$ = 1
val $z_1$ = 8
val $y_1$ = 4
in $f_2$($v_1$, $z_1$, $y_1$) end
and $f_2$($v_2$ ,$z_2$, $y_2$) = let val $x_1$ = 5 + $y_2$
val $y_3$ = $x_1$ * $z_2$
val $x_2$ = $x_1$ – 1
in if $x_2$=0 then $f_3$($y_3$, $v_2$)
else $f_2$($v_2$, $z_2$, $y_3$) end
and $f_3$($y_4$, $v_3$) = let val $w_1$ = $y_4$ + $v_3$
in $w_1$ end
in $f_1$() end;

- as functions are closed, can rename each function definition individually

# SSA construction in functional style




interpret back in imperative form

let fun $f_1()$ = let val $v_1 = 1$
val $z_1 = 8$
val $y_1 = 4$
in $f_2(v_1, z_1, y_1)$ end
and $f_2(v_2, z_2, y_2)$ = let val $x_1 = 5 + y_2$
val $y_3 = x_1 * z_2$
val $x_2 = x_1 - 1$
in if $x_2$=0 then $f_3(y_3, v_2)$
else $f_2(v_2, z_2, y_3)$ end
and $f_3(y_4, v_3)$ = let val $w_1 = y_4 + v_3$
in $w_1$ end
in $f_1()$ end;

- each formal parameter of a function definition is the LHS of a Φ-function. Arguments are the function arguments at calls
- arity of functions, distinctness of LHS variables etc all ok
- resulting code “pruned SSA”
- which functional prog avoids the unnecessary Φ-functions?

“unnecessary”: all call sites provide identical arguments

# Removing unnecessary arguments: λ-dropping

- transformation of functional programs to eliminate formal parameters
- can be performed before or after names are made unique - former option more instructive
- (inverse operation: λ-lifting)
- 2 phases: block sinking and parameter dropping

remove parameters

modify nesting structure of function definitions

# Removing unnecessary arguments: block sinking

Observation: **if**

- all calls to **g** are in body of **f** (or **g**), and
- **g** is closed (all free variables of body are parameters)

**then** the definition of **g** can be moved inside the definition of **f**

```
let fun …
and f(…) = let … in g(…) end
and g(…) = let …in
             if … then g(…) else h (…) end
and h (…) = …(*no call to g*)
in … end;
```

Placing **g** near the end of **f**'s body is advantageous for next step…

```
let fun …
and f(…) = let … in
             let fun g(…) = let …in
                 if … then g(…) else h (…) end
             in g(…) end
and h (…) = …(*no call to g*)
in … end;
```

Note: **g** is allowed to

- make recursive calls
- make calls to "host function" **f**
- make calls to other functions, like **h**

# Block sinking: example

```
let fun f1() = let val v = 1
                   val z = 8
                   val y = 4
               in f2(v, z, y) end
and f2(v, z, y) = let val x = 5 + y
                      val y = x * z
                      val x = x – 1
                  in if x=0 then f3(y, v)
                     else f2(v, z, y) end
and f3(y, v) = let val w = y + v
               in w end
in f1() end;
```

move $f_3$ into $f_2$

move $f_2$ into $f_1$

```
let fun f1() = let val v = 1
                   val z = 8
                   val y = 4
               in let fun f2(v, z, y) =
                      let val x = 5 + y
                          val y = x * z
                          val x = x – 1
                      in if x=0
                      then let fun f3(y, v) =
                                   let val w = y + v
                                   in w end
                           in f3(y, v) end
                      else f2(v, z, y) end
                  in f2(v, z, y) end
in f1() end;
```

(in fact, insert $f_3$ “in the edge” ie only in the then-branch – cf edge split form)

# Block sinking: example

```
let fun f1() = let val v = 1
                   val z = 8
                   val y = 4
               in f2(v, z, y) end
and f2(v, z, y) = let val x = 5 + y
                      val y = x * z
                      val x = x – 1
                  in if x=0 then f3(y, v)
                     else f2(v, z, y) end
and f3(y, v) = let val w = y + v
               in w end
in f1() end;
```

move $f_3$ into $f_2$

move $f_2$ into $f_1$

```
let fun f1() = let val v = 1
                   val z = 8
                   val y = 4
               in let fun f2(v, z, y) =
                      let val x = 5 + y
                          val y = x * z
                          val x = x – 1
                      in if x=0
                      then let fun f3(y, v) =
                                   let val w = y + v
                                   in w end
                           in f3(y, v) end
                      else f2(v, z, y) end
                  in f2(v, z, y) end
in f1() end;
```

Block sinking makes dominance structure explicit: $f_2 = idom(f_3)$, and $f_1 = idom(f_2)$

# Parameter dropping I

```
let fun f1() = let val v = 1
                   val z = 8
                   val y = 4
               in let fun f2(v, z, y) =
                      let val x = 5 + y
                          val y = x * z
                          val x = x – 1
                      in if x=0
                         then let fun f3(y, v) =
                                  let val w = y + v
                                  in w end
                              in f3(y, v) end
                         else f2(v, z, y) end
                  in f2(v, z, y) end
in f1() end;
```

Parameters y and v of $f_3$:
tightest scope for y (ie the def of) surrounding the call to $f_3$ is also the tightest scope surrounding the function definition $f_3$.
Can hence remove parameter y – and similarly parameter v.

```
let fun f1() = ... in if x=0
                      then let fun f3() =
                               let val w = y + v
                               in w end
                           in f3() end
                      else …
```

# Parameter dropping II

```
let fun f1() = let val v = 1
                   val z = 8
                   val y = 4
               in let fun f2(v, z, y) =
                      let val x = 5 + y
                          val y = x * z
                          val x = x – 1
                      in if x=0
                      then let fun f3() = …
                           in f3() end
                      else f2(v, z, y) end
                  in f2(v, z, y) end
in f1() end;
```

Similarly, the external call to $f_2$ from within the body of $f_1$ would allow to remove all three parameters from $f_2$.

# Parameter dropping III

```
let fun f1() = let val v = 1
                   val z = 8
                   val y = 4
               in let fun f2(v, z, y) =
                      let val x = 5 + y
                          val y = x * z
                          val x = x – 1
                      in if x=0
                         then let fun f3() = …
                              in f3() end
                         else f2(v, z, y) end
                  in f2(v, z, y) end
in f1() end;
```

Similarly, the external call to $f_2$ from within the body of $f_1$ would allow to remove all three parameters from $f_2$.

Recursive call of $f_2$:

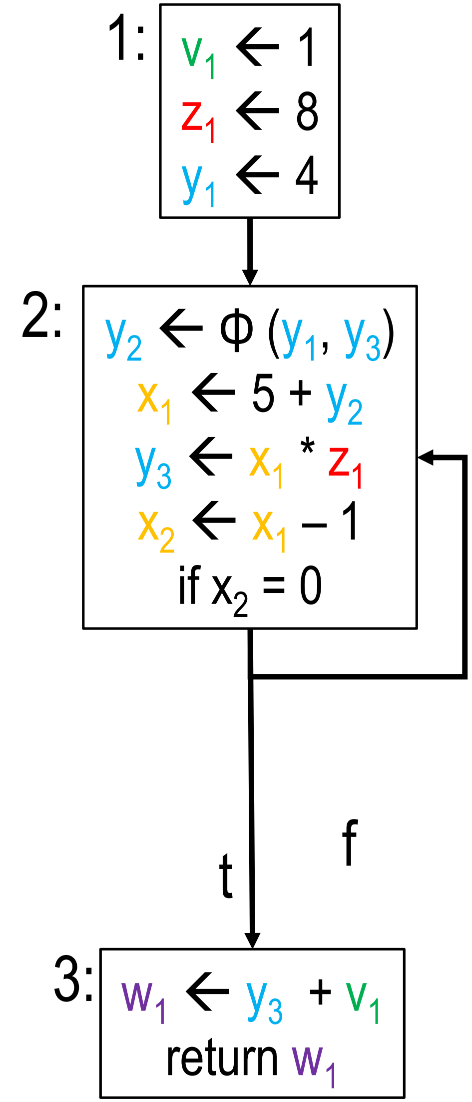
- **admits** the removal of parameters **v** and **z**, since the defs associated with the uses at the call site are the defs in the formal parameter list
- does not **admit** the removal of parameters **y**, since the def associated with the use of **y** at the call site is **not** the def in the formal parameter list

# Parameter dropping IV

```
let fun f1() = let val v = 1
                   val z = 8
                   val y = 4
               in let fun f2(y) =
                      let val x = 5 + y
                          val y = x * z
                          val x = x – 1
                      in if x=0
                      then let fun f3() =
                                   let val w = y + v
                                   in w end
                           in f3() end
                      else f2(y) end
                  in f2(y) end
in f1() end;
```

make names distinct

read as SSA program




Superfluous Φ-functions avoided.

# SSA and functional programming - summary

SSA discipline shares many properties with tail-recursive, first-order fragment of functional languages

- transfer of analysis/optimization algorithms
- suitable intermediate format for compiling functional and imperative languages

- function calls not in tail position: calls to imperative functions/methods/procedures
- alternative functional representation of control flow: continuations