

# Static Single Assignment (SSA) Form

---

A sparse program representation for data-flow.

# Computing Static Single Assignment (SSA) Form

---

## Overview:

- What is SSA?
- Advantages of SSA over use-def chains
- “Flavors” of SSA
- Dominance frontiers
- Inserting  $\phi$ -nodes
- Renaming the temporaries
- Translating out of SSA form

R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, *Efficiently Computing Static Single Assignment Form and the Control Dependence Graph*, ACM TOPLAS

13(4):451–490, Oct 1991

# What is SSA?

---

- Each assignment to a temporary is given a unique name
- All of the uses reached by that assignment are renamed
- Easy for straight-line code

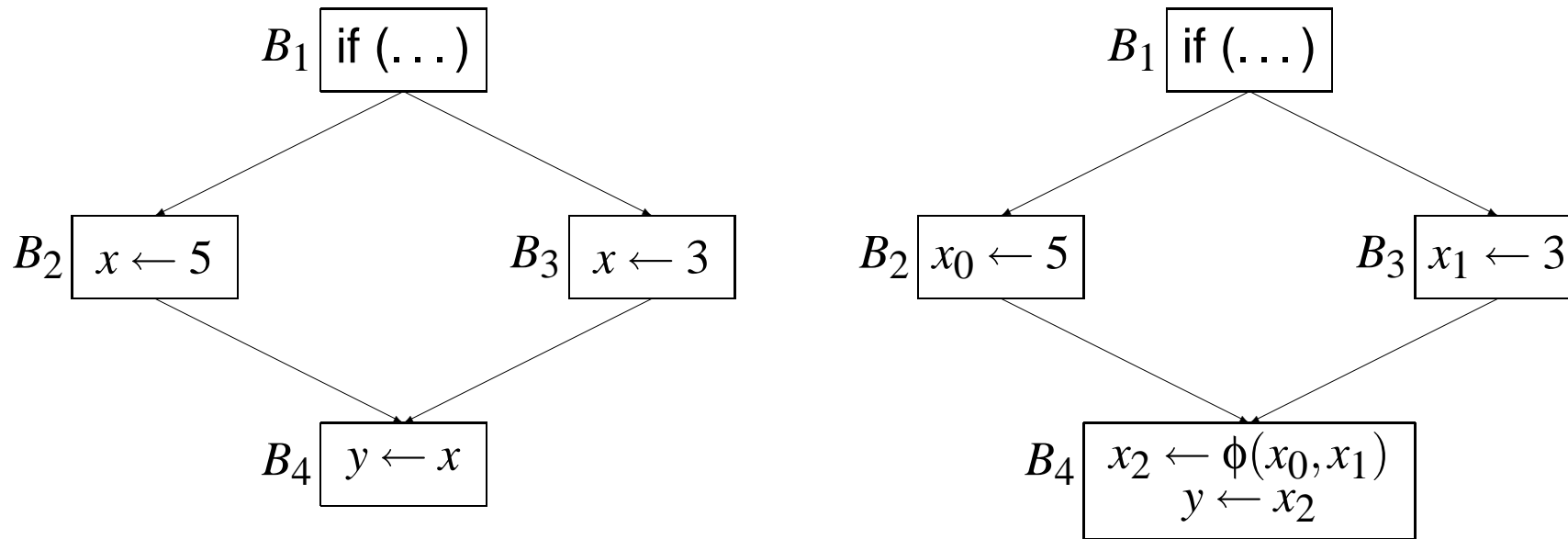
$$\begin{array}{l|l} v \leftarrow 4 & v_0 \leftarrow 4 \\ \leftarrow v + 5 & \leftarrow v_0 + 5 \\ v \leftarrow 6 & v_1 \leftarrow 6 \\ \leftarrow v + 7 & \leftarrow v_1 + 7 \end{array}$$

- What about control flow?

$\Rightarrow$   $\phi$ -nodes

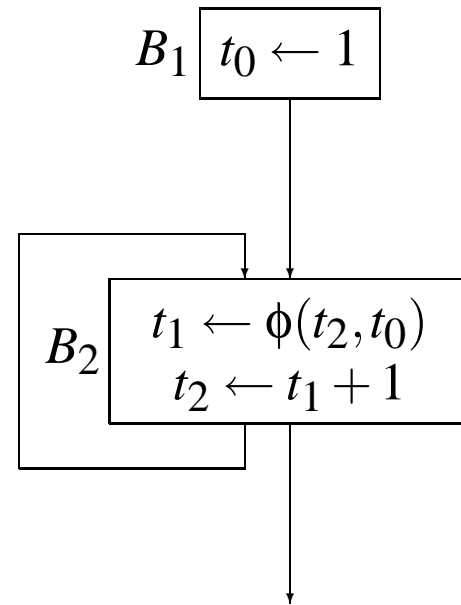
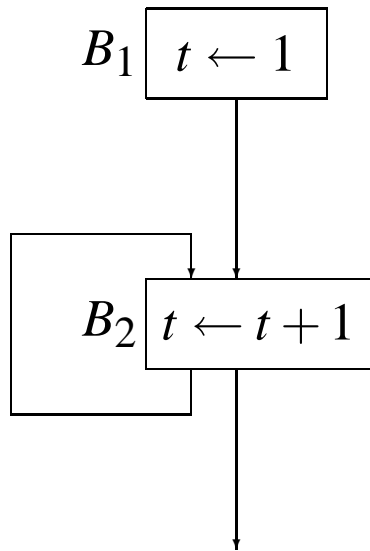
# What is SSA?

---



# What is SSA?

---



# Advantages of SSA over use-def chains

---

- More compact representation
- Easier to update?
- Each use has only one definition
- Definitions explicitly merge values  
May still reach multiple  $\phi$ -nodes

# “Flavors” of SSA

---

Where do we place  $\phi$ -nodes?

## Condition:

If two non-null paths  $x \rightarrow^+ z$  and  $y \rightarrow^+ z$  converge at node  $z$ , and nodes  $x$  and  $y$  contain assignments to  $t$  (in the original program), then a  $\phi$ -node for  $t$  must be inserted at  $z$  (in the new program)

## *minimal*

As few as possible subject to condition

## *semi-pruned*

by Preston Briggs

As few as possible subject to condition, and  $t$  must be live across some basic block

## *pruned*

As few as possible subject to condition, and no dead  $\phi$ -nodes

# Dominance Frontiers

---

From  $v$ 's point of view, these are the nodes at which other control paths that don't go through  $v$  make their earliest appearance.

The *dominance frontier* of  $v$  is the set of nodes  $DF(v)$  such that:

- $v$  dominates a predecessor of  $w \in DF(v)$ , but  $v$  does not strictly dominate  $w \in DF(v)$

$$DF(v) = \{w \mid (\exists u \in PRED(w)) [v \text{ DOM } u] \wedge v \overline{\text{DOM}} w\}$$

- $d$  dominates  $v$ ,  $d \text{ DOM } v$ , in a CFG *iff all* paths from *Entry* to  $v$  include  $d$
- $d$  *strictly* dominates  $v$ :

$$d \text{ DOM! } v \iff d \text{ DOM } v \text{ and } d \neq v$$

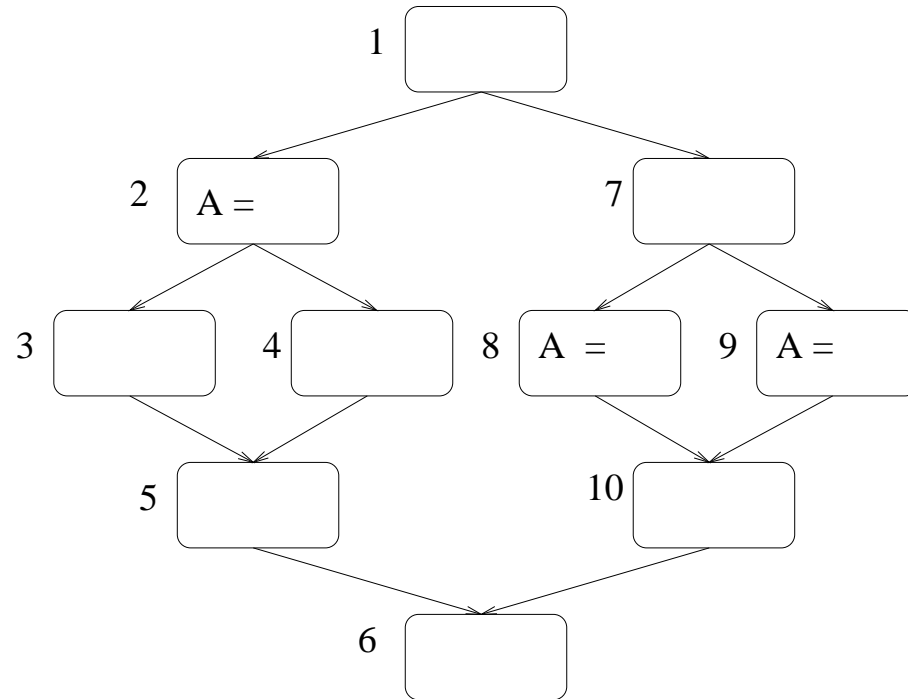
- The *immediate* dominator of  $v$ ,  $IDOM(v)$ , is the closest strict dominator of  $v$ :

$$d \text{ IDOM } v \iff d \text{ DOM! } v \wedge (\forall w \mid w \text{ DOM! } v) [w \text{ DOM } d]$$

$IDOM(v)$  is  $v$ 's parent in the *dominator tree*

# Dominance Frontier: Example

---



DF(8) =

DF(9) =

DF(2) =

DF({8,9}) =

DF(10) =

DF({2,8,9,10}) =

# Iterated Dominance Frontier

---

Extend the dominance frontier mapping from nodes to sets of nodes:

$$DF(S) = \bigcup_{n \in S} DF(n)$$

The *iterated* dominance frontier  $DF^+(S)$  is the limit of the sequence:

$$\begin{aligned} DF_1(S) &= DF(S) \\ DF_{i+1}(S) &= DF(S \cup DF_i(S)) \end{aligned}$$

## Theorem:

The set of nodes that need  $\phi$ -nodes for any temporary  $t$  is the iterated dominance frontier  $DF^+(S)$ , where  $S$  is the set of nodes that define  $t$

# Iterated Dominance Frontier Algorithm: $DF^+(S)$

---

**Input:** Set of blocks  $S$

**Output:**  $DF^+(S)$

$workList \leftarrow \{\}$

$DF^+(S) \leftarrow \{\}$

**foreach**  $n \in S$  **do**

$DF^+(S) \leftarrow DF^+(S) \cup \{n\}$

$workList \leftarrow workList \cup \{n\}$

**end**

**while**  $workList \neq \{\}$  **do**

take  $n$  from  $workList$

**foreach**  $c \in DF(n)$  **do**

**if**  $c \notin DF^+(S)$  **then**

$DF^+(S) \leftarrow DF^+(S) \cup \{c\}$

$workList \leftarrow workList \cup \{c\}$

**end**

**end**

**end**

## Inserting $\phi$ -nodes (minimal SSA)

---

```
foreach  $t \in \text{Temporaries}$  do  
   $S \leftarrow \{n \mid t \in \text{Def}(n)\} \cup \text{Entry}$   
  Compute  $\text{DF}^+(S)$   
  foreach  $n \in \text{DF}^+(S)$  do  
    Insert a  $\phi$ -node for  $t$  at  $n$   
  end  
end
```

## Inserting $\phi$ -nodes for globals (semi-pruned SSA)

---

Compute *local* liveness: globals are those live across block boundaries (*ie*, used before definition in *any* basic block)

**foreach**  $t \in \text{Temporaries}$  **do**

**if**  $t \in \text{Globals}$  **then**

$S \leftarrow \{n \mid t \in \text{Def}(n)\} \cup \text{Entry}$

        Compute  $\text{DF}^+(S)$

**foreach**  $n \in \text{DF}^+(S)$  **do**

            Insert a  $\phi$ -node for  $t$  at  $n$

**end**

**end**

**end**

## Inserting fewest $\phi$ -nodes (pruned SSA)

---

Compute *global* liveness: nodes where each temporary is live-in

**foreach**  $t \in \text{Temporaries}$  **do**

**if**  $t \in \text{Globals}$  **then**

$S \leftarrow \{n \mid t \in \text{Defs}(n)\} \cup \text{Entry}$

        Compute  $\text{DF}^+(S)$

**foreach**  $n \in \text{DF}^+(S)$  **do**

**if**  $t$  live-in at  $n$  **then**

                Insert a  $\phi$ -node for  $t$  at  $n$

**end**

**end**

**end**

**end**

# Renaming the temporaries

---

After  $\phi$ -node insertion, uses of  $t$  are either:

**original:** dominated by the definition that computes  $t$ .

If not, then  $\exists$  path to use avoiding definition, which means separate paths from definitions converge between definition and use, thus inserting another definition.

*ie*, each use dominated by an evaluation of  $t$  or a  $\phi$ -node for  $t$

$\phi$ : has a corresponding predecessor  $p$ , dominated by the definition of  $t$  (as before)

Thus, walk dominator tree, replacing each definition and its dominated uses with a new temporary.

Use a stack to hold current name (subscript) for each set of dominated nodes.

Propagate names from each block to corresponding  $\phi$ -node operands of its successors.

# Renaming the temporaries

---

**foreach**  $t \in \text{Temporaries}$  **do**  $\text{count}[t] \leftarrow 0$ ;  $\text{stack}[t] \leftarrow \text{empty}$ ;  $\text{stack}[t].\text{push}(0)$   
*Rename(Entry)*

**proc** *Rename*( $n$ )  $\equiv$

**foreach** *statement*  $I \in n$  **do**

**if**  $s \neq \phi$  **then** **foreach**  $t \in \text{Uses}(I)$  **do**

$i \leftarrow \text{stack}[t].\text{top}$

replace use of  $t$  with  $t_i$  in  $I$

**foreach**  $t \in \text{Defs}(I)$  **do**

$i \leftarrow ++\text{count}[t]$ ;  $\text{stack}[t].\text{push}(i)$

replace def of  $t$  with  $t_i$  in  $I$

**foreach**  $s \in \text{SUCC}(n)$  **do**

given  $n$  is the  $j$ th predecessor of  $s$

**foreach**  $\phi \in s$  **do**

given  $t$  is the  $j$ th operand of  $\phi$

$i \leftarrow \text{stack}[t].\text{top}$

replace  $j$ th operand of  $\phi$  with  $t_i$

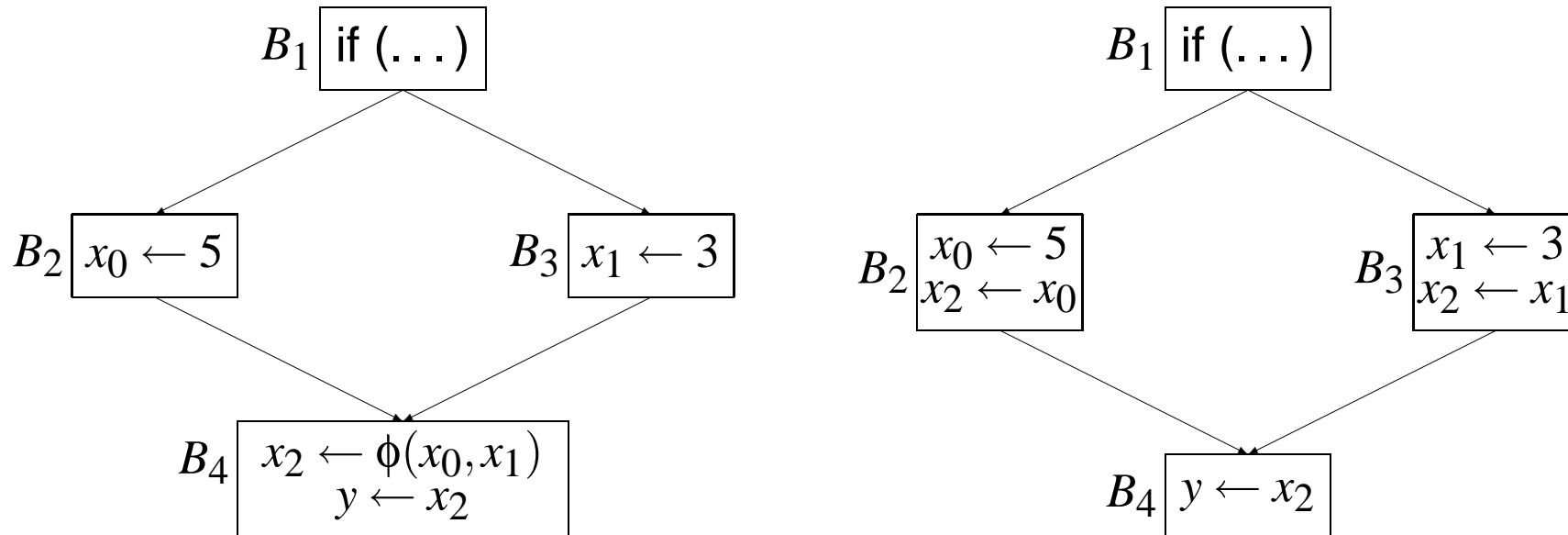
**foreach**  $c \in \text{Children}(n)$  **do** *Rename*( $c$ )

**foreach** *statement*  $I \in n, t \in \text{Defs}(I)$  **do**  $\text{stack}[t].\text{pop}()$

# Translating Out of SSA Form

---

Replace  $\phi$ -nodes with copy statements in predecessors



# Next Time

---

## Static Single Assignment

- Induction variables (standard vs. SSA)
- Loop Invariant Code Motion with SSA