CMSC 631 – Program Analysis and Understanding

Static Single Assignment Form and Dominators

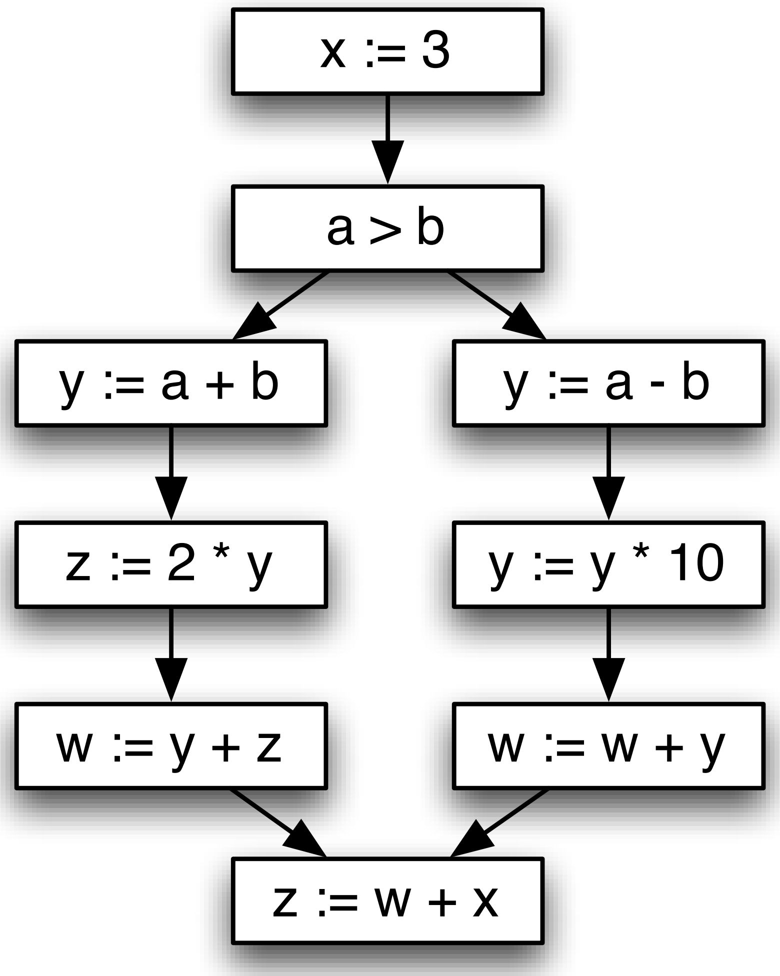
Motivation

- Data flow analysis needs to represent facts at every program point

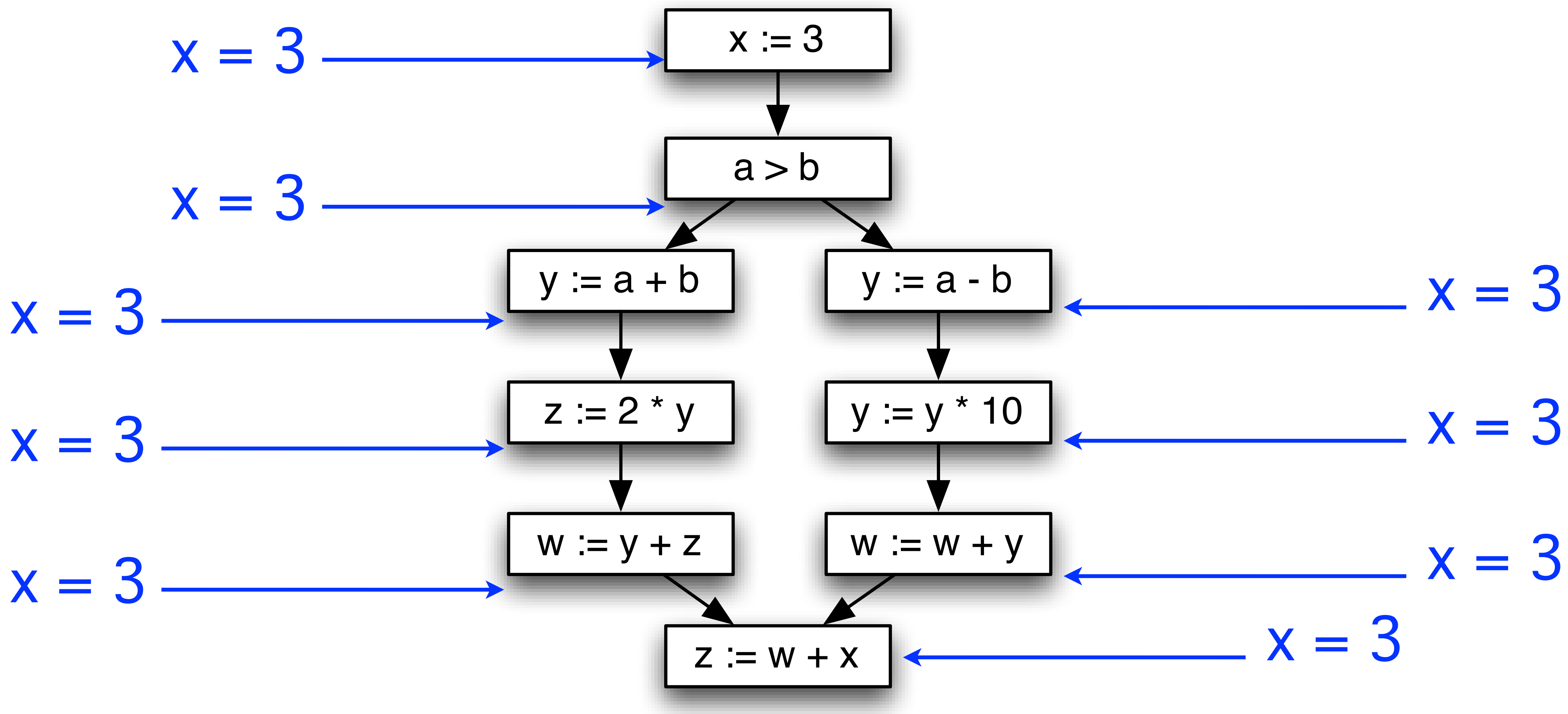
- What if
  - There are a lot of facts and
  - There are a lot of program points?
  - $\Rightarrow$ potentially takes a lot of space/time

- Most likely, we're keeping track of irrelevant facts

Example



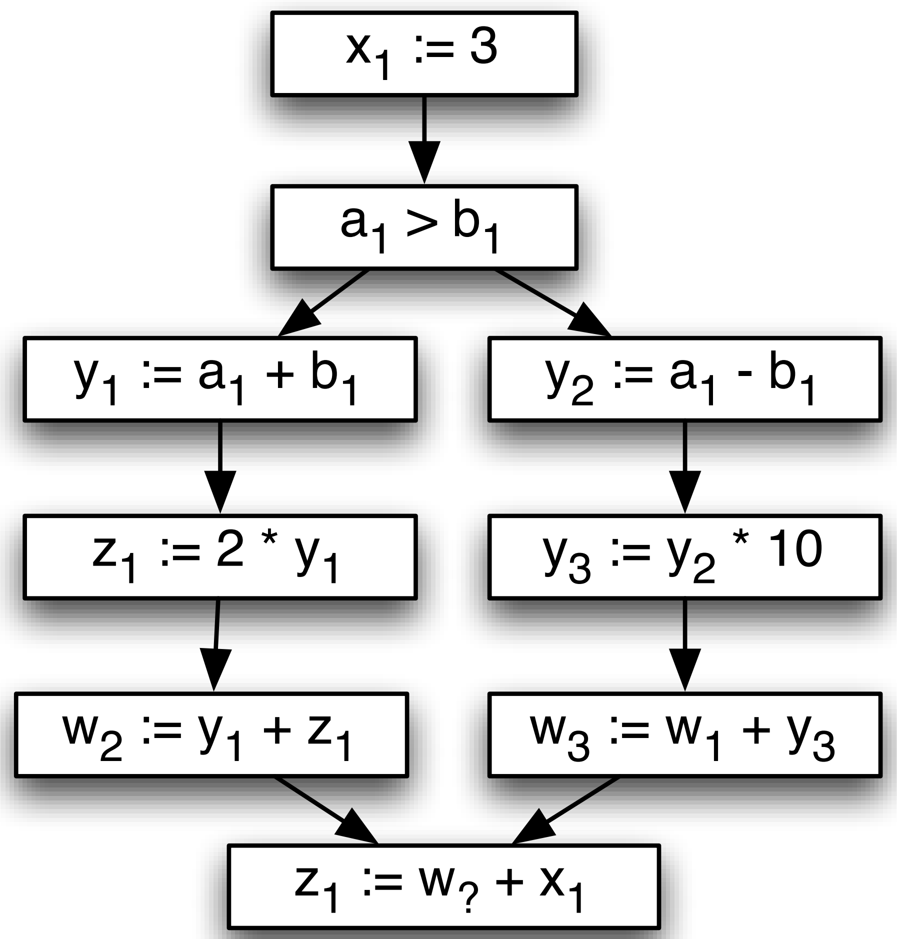
Example



Sparse Representation

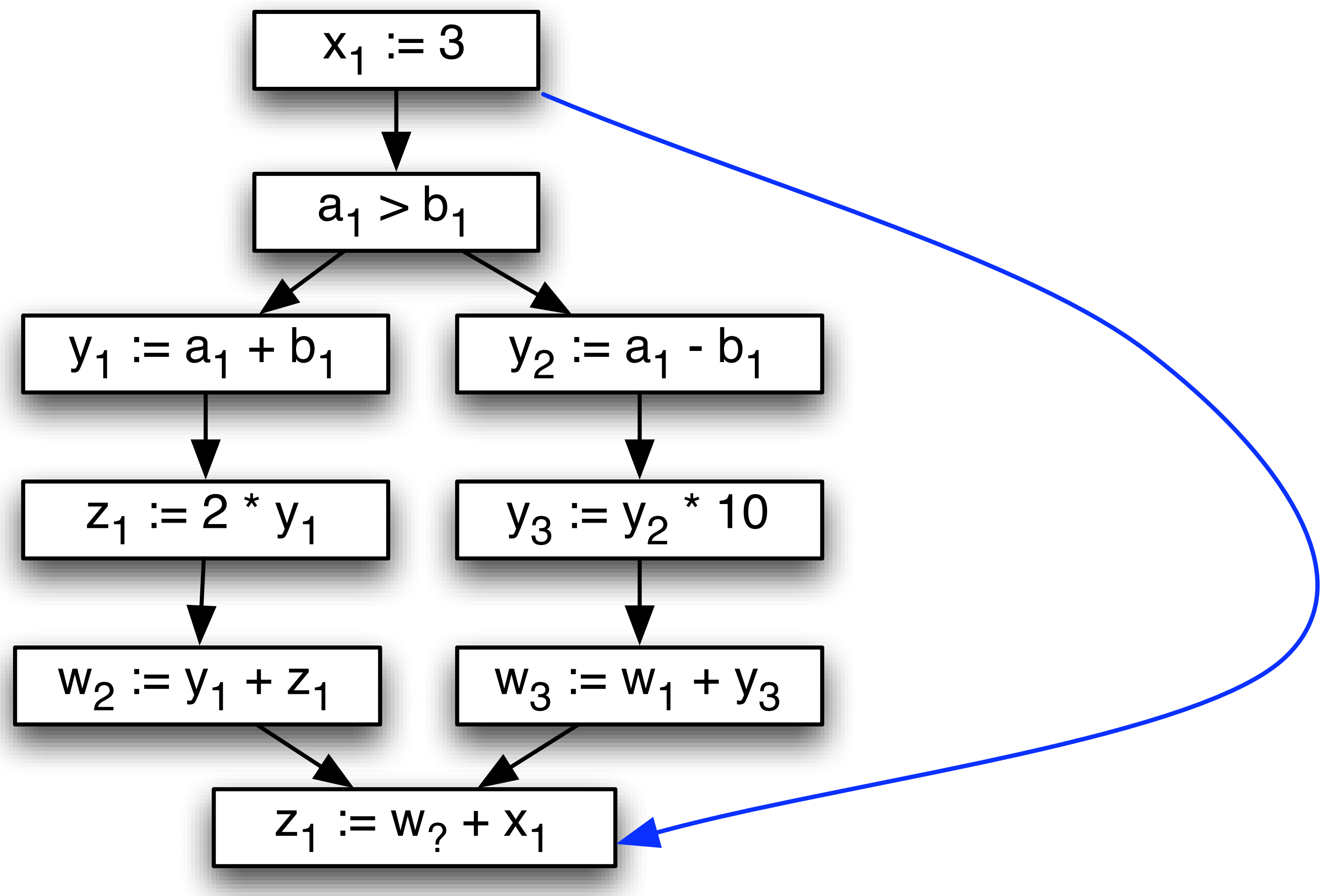
- Instead, we'd like to use a sparse representation
  - Only propagate facts about x where they're needed
- Enter *static single assignment* form
  - Each variable is defined (assigned to) exactly once
  - But may be used multiple times

Example: SSA



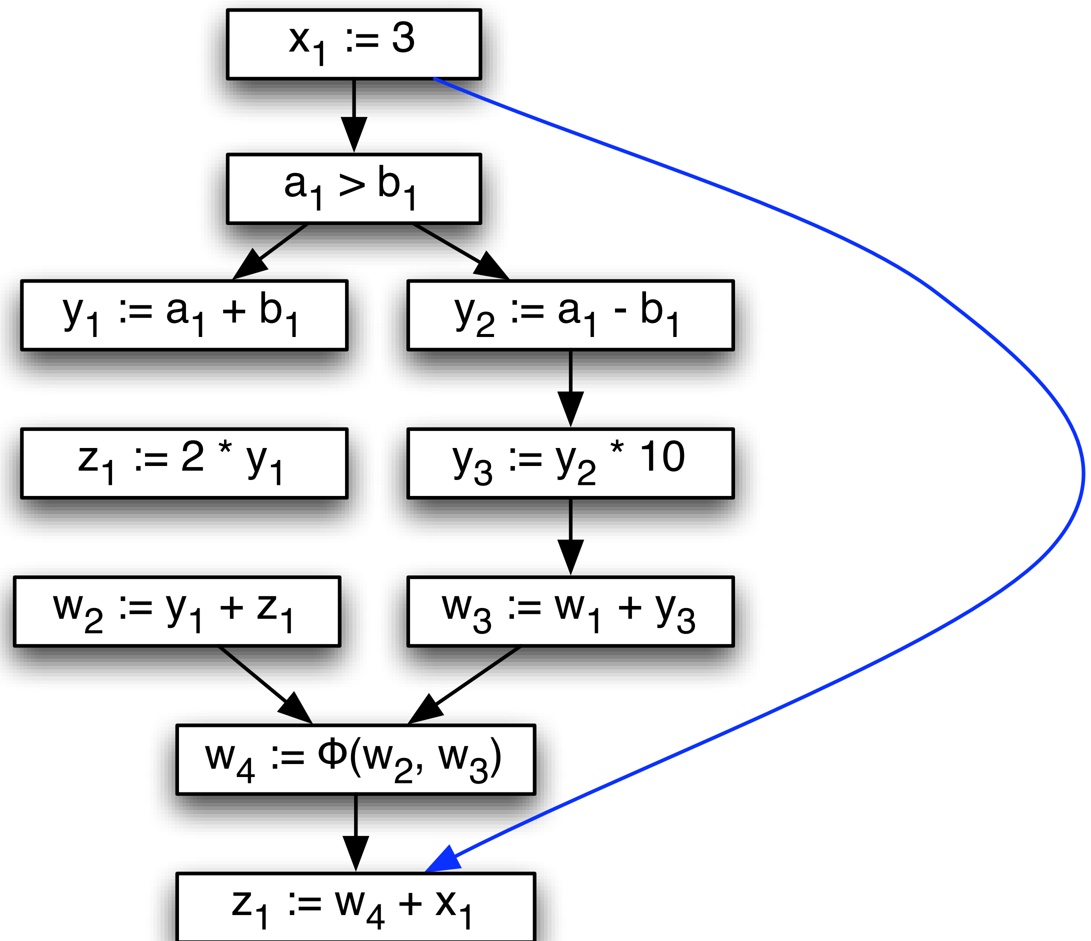
- Add *SSA edges* from definitions to uses
  - No intervening statements use/define variable
  - Safe to propagate only along SSA edges

Example: SSA



- Add *SSA edges* from definitions to uses
  - No intervening statements use/define variable
  - Safe to propagate only along SSA edges

What About Joins?



- Add Φ functions/nodes to model joins
  - Intuitively, takes meet of arguments
  - At code generation time, need to eliminate Φ nodes

Constant Propagation Revisited

- Initialize facts at each program point
  - C(n) := top
- Add all SSA edges to the worklist
- While the worklist isn't empty,
  - Remove an edge (x, y) from the worklist
  - C(y) := C(y) meet C(x)
  - Add SSA edges from y if C(y) changed

Def-Use Chains vs. SSA

Def-Use Chains vs. SSA

- Alternative: Don't do renaming; instead, compute simple def-use chains (reaching definitions)
  - Propagate facts along def-use chains

Def-Use Chains vs. SSA

- Alternative: Don't do renaming; instead, compute simple def-use chains (reaching definitions)
  - Propagate facts along def-use chains

- Drawback: Potentially quadratic size

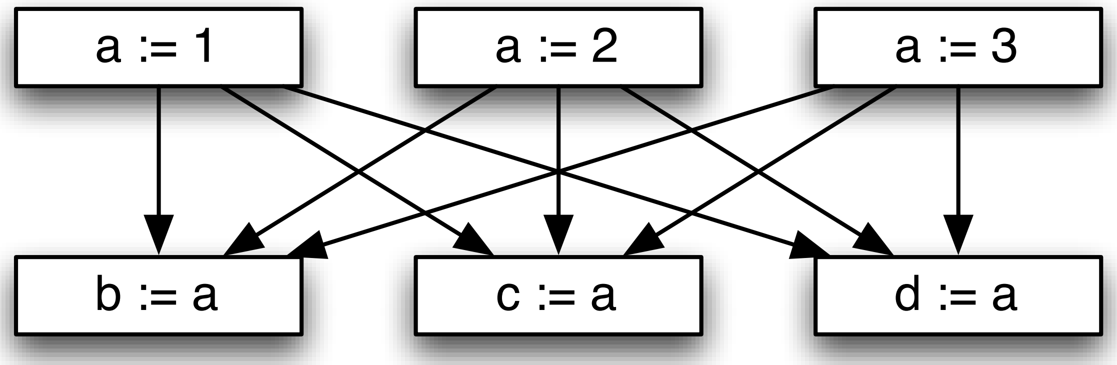
Def-Use Chains vs. SSA (cont'd)

```
case (...) of
  0:  a := 1;
  1:  a := 2;
  2:  a := 3;
end
case (...) of
  0:  b := a;
  1:  c := a;
  2:  d := a;
end
```

Def-Use Chains vs. SSA (cont'd)

```
case (...) of
  0:  a := 1;
  1:  a := 2;
  2:  a := 3;
end
case (...) of
  0:  b := a;
  1:  c := a;
  2:  d := a;
end
```

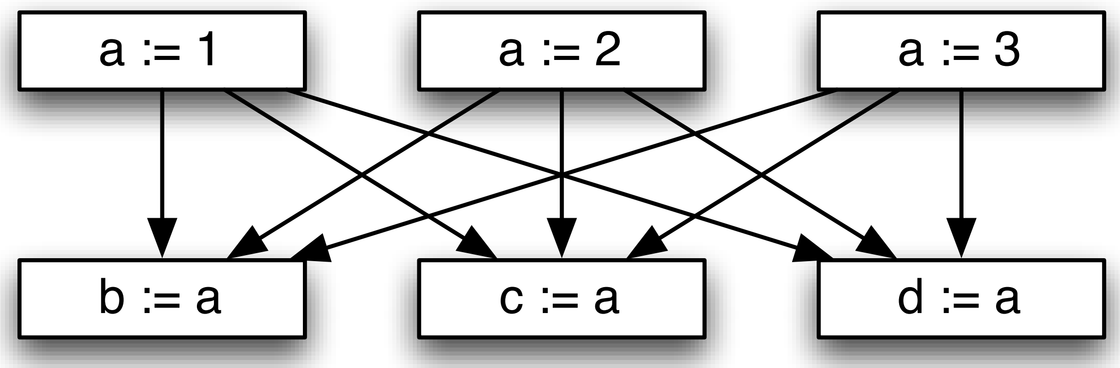
Def-Use Chains



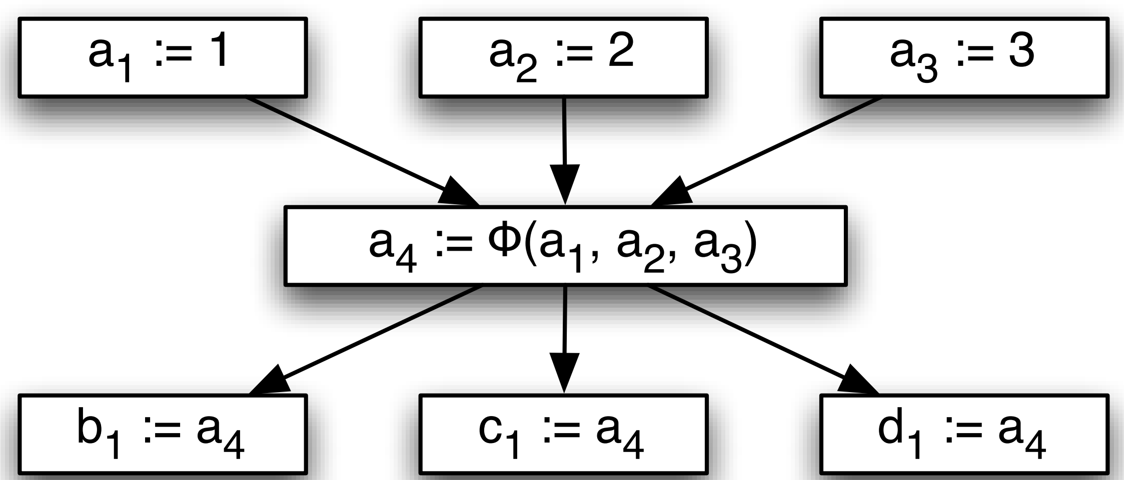
Def-Use Chains vs. SSA (cont'd)

```
case (...) of
  0:  a := 1;
  1:  a := 2;
  2:  a := 3;
end
case (...) of
  0:  b := a;
  1:  c := a;
  2:  d := a;
end
```

Def-Use Chains



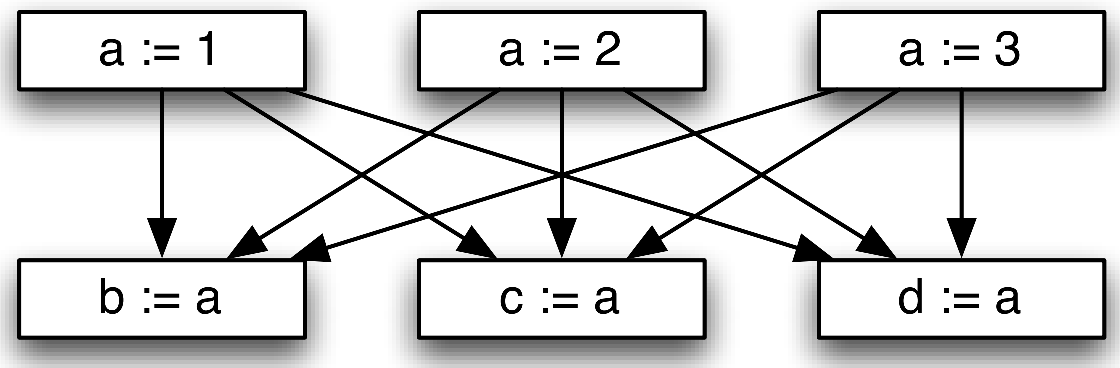
SSA Form



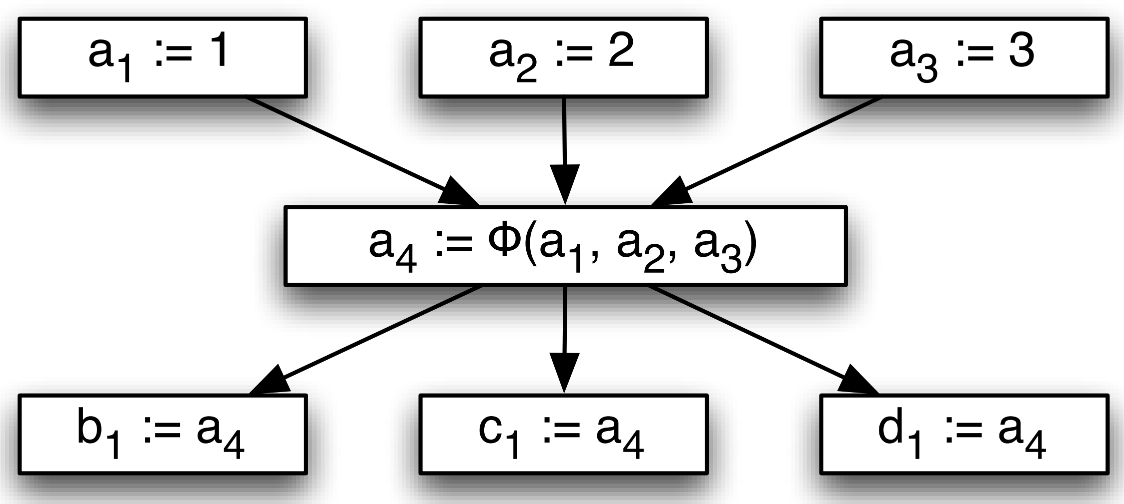
Def-Use Chains vs. SSA (cont'd)

```
case (...) of
  0:  a := 1;
  1:  a := 2;
  2:  a := 3;
end
case (...) of
  0:  b := a;
  1:  c := a;
  2:  d := a;
end
```

Def-Use Chains



SSA Form



Quadratic vs. (in practice) linear behavior

Computing SSA Form

Computing SSA Form

- Step 1: Compute the dominance frontier

Computing SSA Form

- Step 1: Compute the dominance frontier
- Step 2: Use dominance frontier to place Φ nodes

Computing SSA Form

- Step 1: Compute the dominance frontier
- Step 2: Use dominance frontier to place Φ nodes
  - Naive, impractical step 2: put a Φ function for every variable at the beginning of every block

Computing SSA Form

- Step 1: Compute the dominance frontier
- Step 2: Use dominance frontier to place Φ nodes
  - Naive, impractical step 2: put a Φ function for every variable at the beginning of every block
  - Better: If node X contains assignment to a, put Φ function for a in dominance frontier of X

Computing SSA Form

- Step 1: Compute the dominance frontier
- Step 2: Use dominance frontier to place Φ nodes
  - Naive, impractical step 2: put a Φ function for every variable at the beginning of every block
  - Better: If node X contains assignment to a, put Φ function for a in dominance frontier of X
    - Adding Φ fn may require introducing additional Φ fn

Computing SSA Form

- Step 1: Compute the dominance frontier
- Step 2: Use dominance frontier to place Φ nodes
  - Naive, impractical step 2: put a Φ function for every variable at the beginning of every block
  - Better: If node X contains assignment to a, put Φ function for a in dominance frontier of X
    - Adding Φ fn may require introducing additional Φ fn
- Step 3: Rename variables so only one definition per name

Dominators

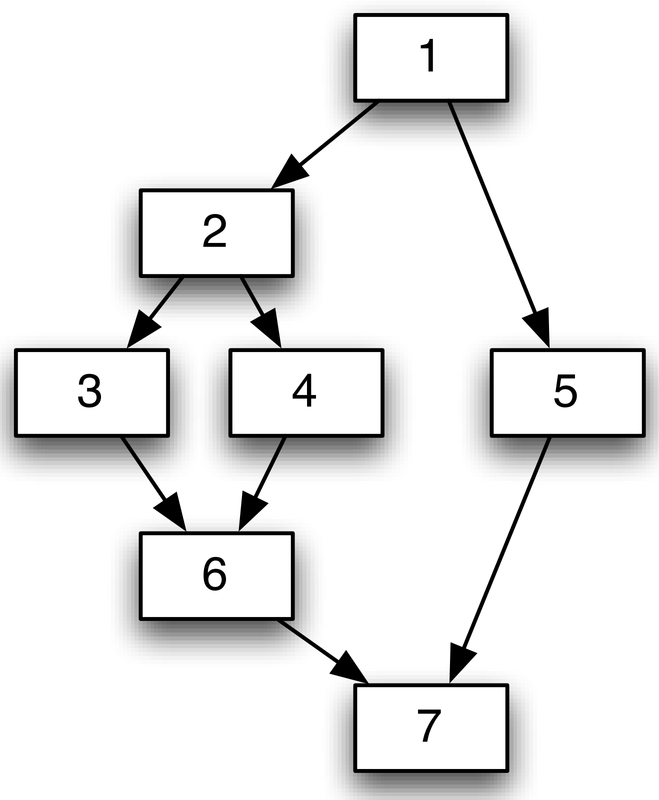
- Let X and Y be nodes in the CFG
  - Assume single entry point Entry

- X *dominates* Y (written $X \geq Y$) if
  - X appears on every path from Entry to Y
  - Note $\geq$ is reflexive

- X *strictly dominates* Y (written $X > Y$) if
  - X dominates Y but $X \neq Y$

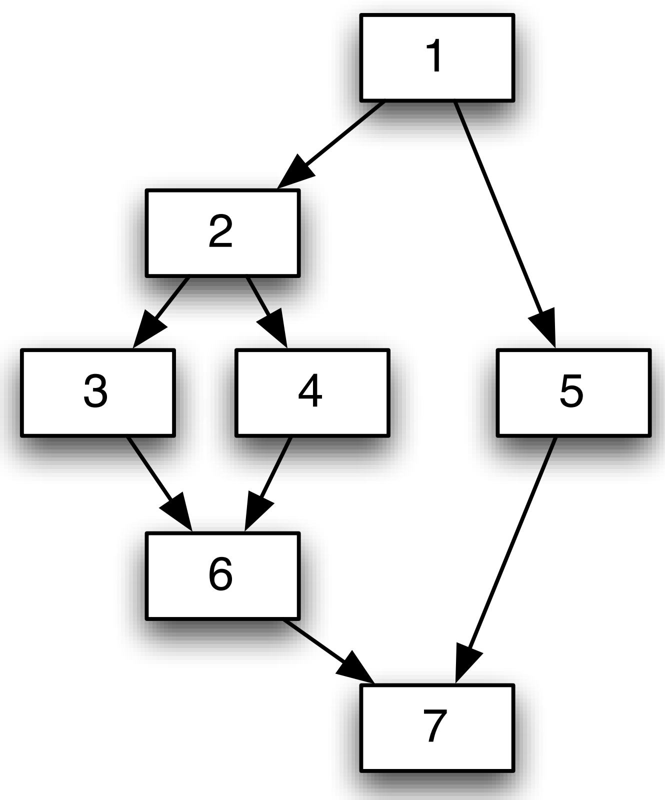
Dominator Tree

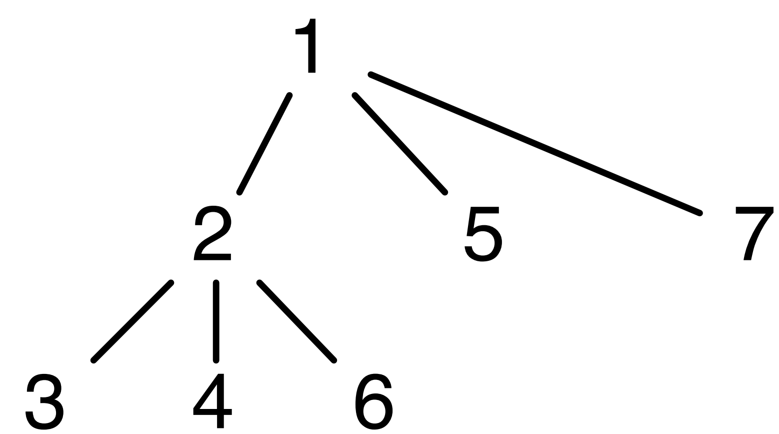
- The dominator relationship forms a tree
  - Edge from parent to child = parent dominates child
  - Note: edges are not same as CFG edges!



Dominator Tree

- The dominator relationship forms a tree
  - Edge from parent to child = parent dominates child
  - Note: edges are not same as CFG edges!





Computing Dominator Tree

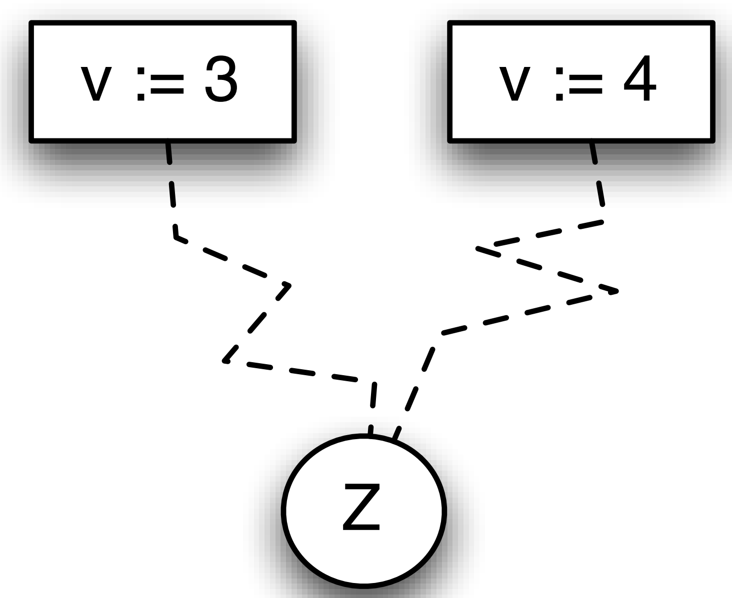
- Standard algorithm due to Lengauer and Tarjan
- Runs in time $O(E\alpha(E, N))$
  - E = # of edges, N = # of nodes
  - where $\alpha(\cdot)$ is the inverse Ackerman's function
  - Very slow growing; effectively constant in practice
- Algorithm quite difficult to understand
  - But lots of pseudo-code available

Why (Else) Are Dominators Useful?

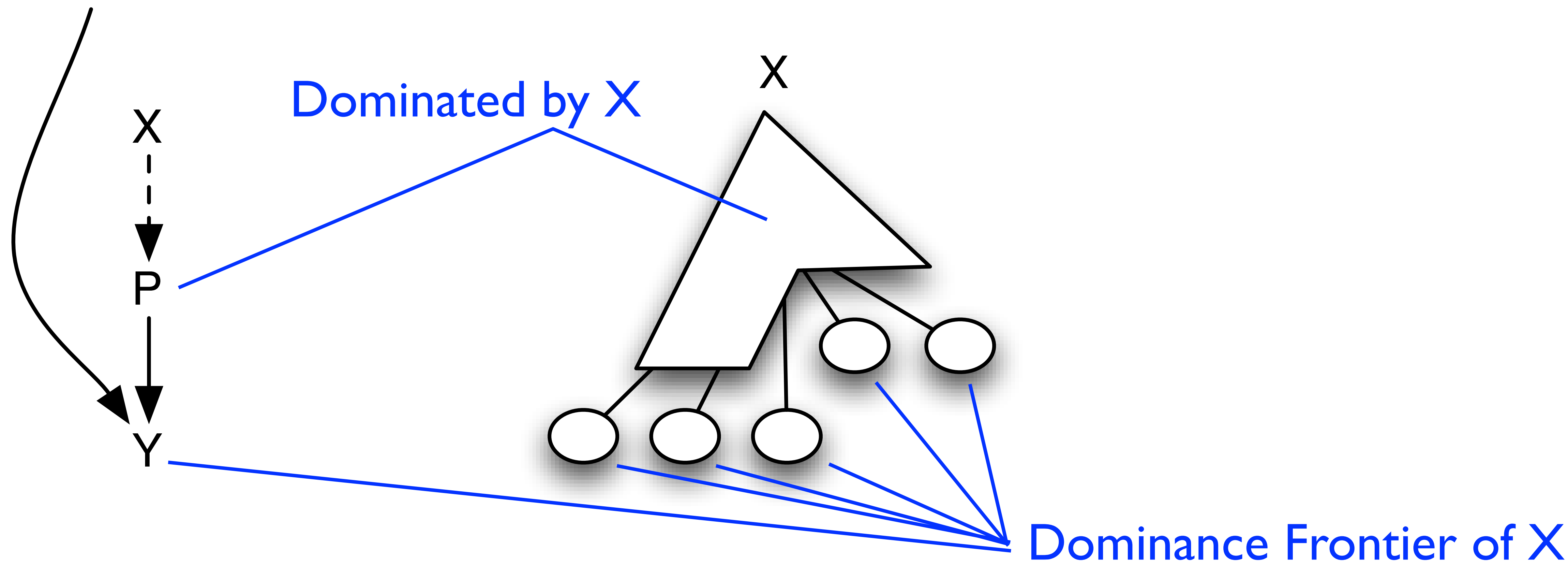
- Identify loops in CFG
  - All nodes X dominated by entry node H, where in CFG X can reach H and there is exactly one back edge to H among the X
    - A construct like "continue" must be represented as jumping to the end of the loop, to ensure one back edge
- Computing control dependences
  - Details given in the last few slides

Where do Φ Functions Go?

- We need a Φ function at node Z if
  - Two non-null CFG paths that both define v
  - Such that both paths start at two distinct nodes and end at Z

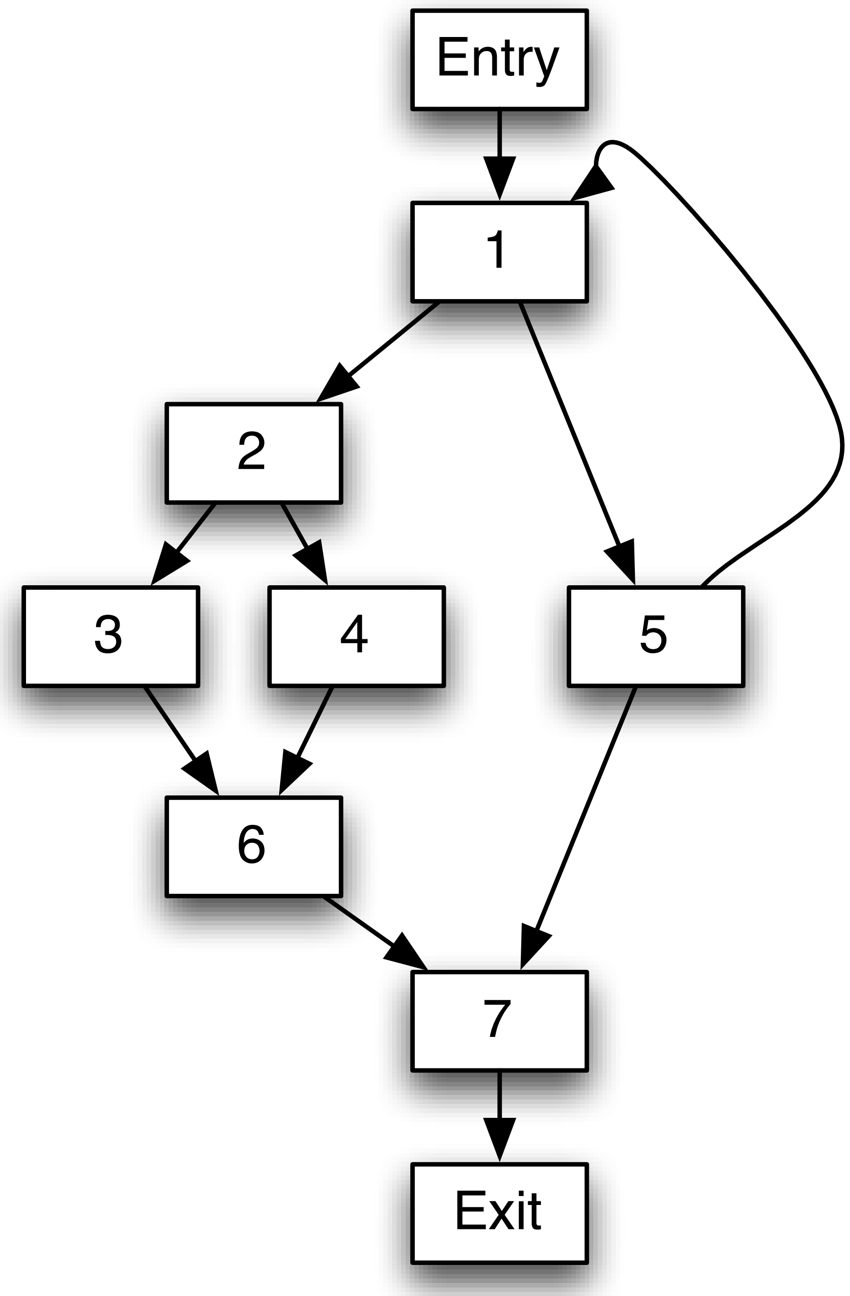


Dominance Frontiers: Illustration



- Y is in the dominance frontier of X iff
  - X dominates a predecessor of Y
  - X does not strictly dominate Y

Example



DF(1) =

DF(2) =

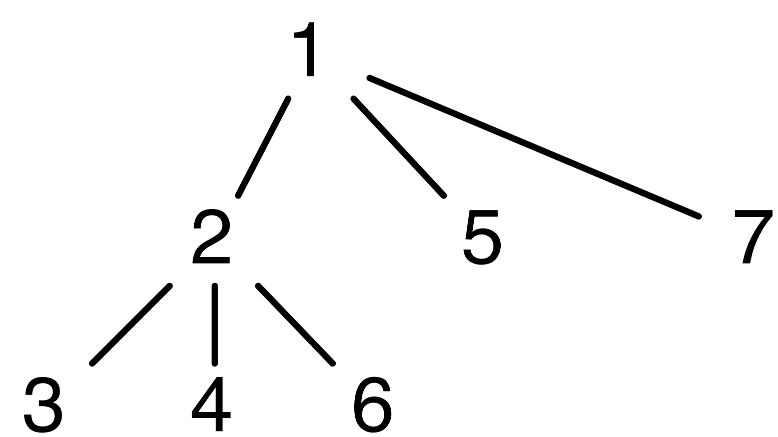
DF(3) =

DF(4) =

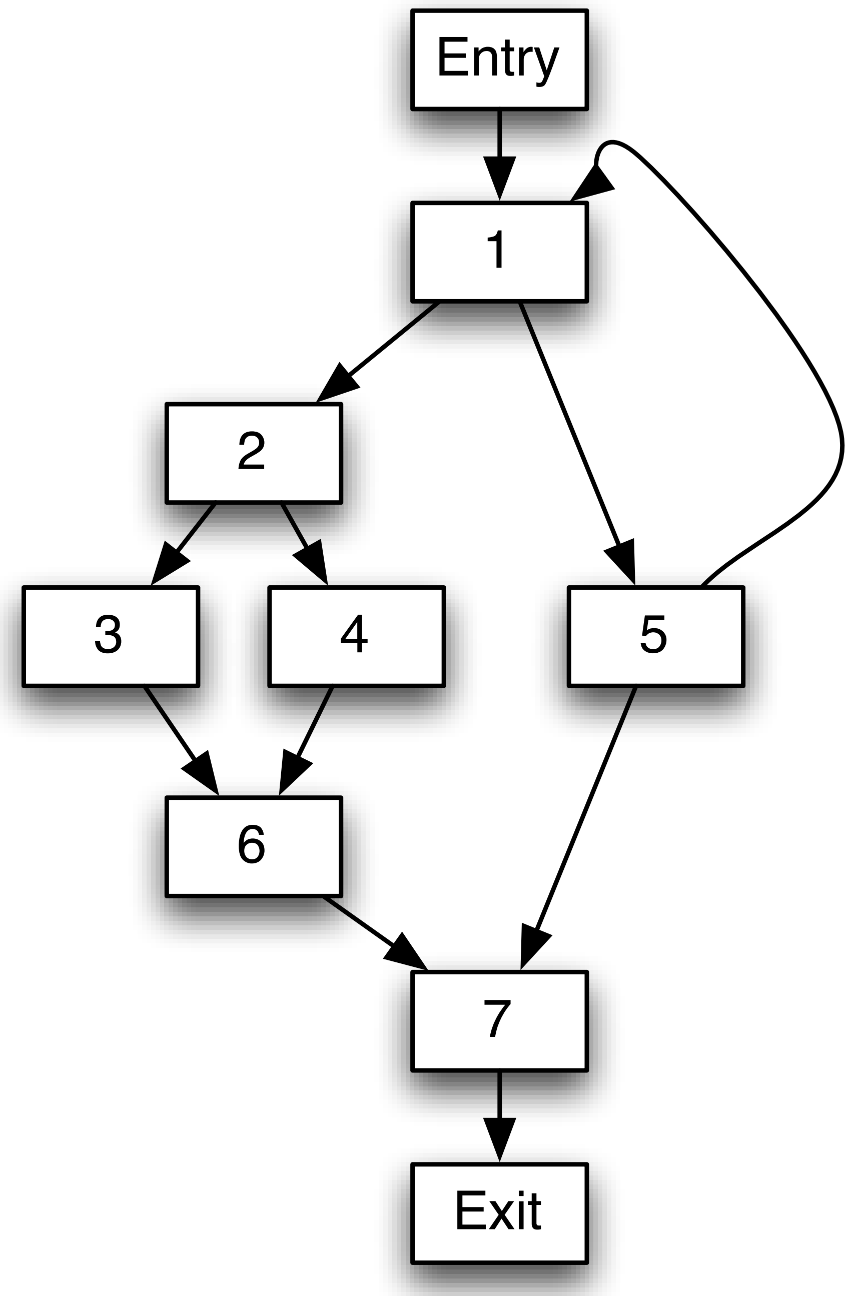
DF(5) =

DF(6) =

DF(7) =



Example



$DF(1) = \{1\}$

$DF(2) = \{7\}$

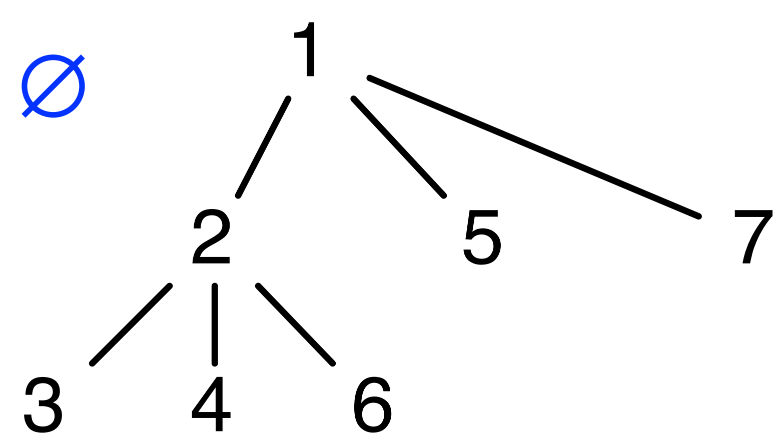
$DF(3) = \{6\}$

$DF(4) = \{6\}$

$DF(5) = \{1, 7\}$

$DF(6) = \{7\}$

$DF(7) = \emptyset$

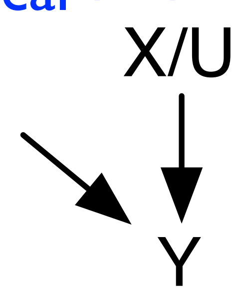


Computing Dominance Frontiers

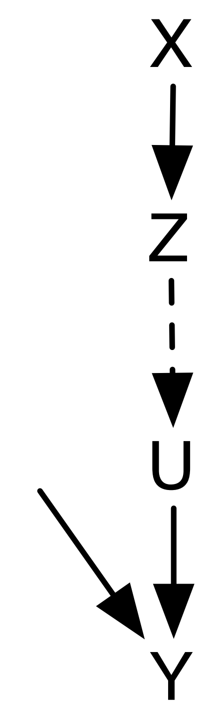
- $DF(X) = DF_{local}(X) \cup DF_{up}(S_X)$ where
  - $DF_{local}(X) = \{Y \in succ(X) \mid X \not\gg Y\}$
    - Any successor of X not (strictly) dominated by X is in $DF(X)$
  - $S_X = \{ Z \mid idom(Z) = X \}$
    - $idom(Z)$ is the parent of Z in the dominator tree
  - $DF_{up}(S_X) = \{Y \in DF(Z) \mid Z \in S_X \text{ and } X \not\gg Y \}$
    - Nodes from $DF(Z)$ that are not strictly dominated by X are also in $DF(X)$

Why Is This Sufficient?

- Suppose $Y \in DF(X)$
  - Then there is a $U \in pred(Y)$ such that $X \geq U$, $X \not> Y$
  - If $U=X$, then $Y \in DF_{local}(X) = \{Y \in succ(X) \mid X \not> Y\}$



  - Otherwise $U \neq X$
    - Then there is a node Z such that $idom(Z)=X$ and $Z \geq U$
      - Possibly $Z=U$
    - Since $X \not> Y$, $Z \not> Y$, hence $Y \in DF(Z)$
  - Therefore $Y \in DF_{up}(\{Z\}) = \{Y \in DF(Z) \mid X \not> Y\}$



Algorithm

- Let $sdom(X) = \{Y \mid X>Y\}$
- In a postorder traversal on dominator tree
  - $DF(X) = succ(X) - sdom(X)$
    - I.e., $DF(X) = DF_{local}(X)$
  - For each Z such that $idom(Z) = X$ do
    - $DF(X) = DF(X) \cup (DF(Z) - sdom(X))$
    - I.e., $DF(X) = DF(X) \cup DF_{up}(Z)$

Equivalent Algorithm

- In a postorder traversal on dominator tree
  - $DF(X) = succ(X)$
  - For each Z such that $idom(Z) = X$ do
    - $DF(X) = DF(X) \cup DF(Z)$
  - $DF(X) = DF(X) - sdom(X)$
- There's another equivalent algorithm that runs in $O(E+|DF|)$

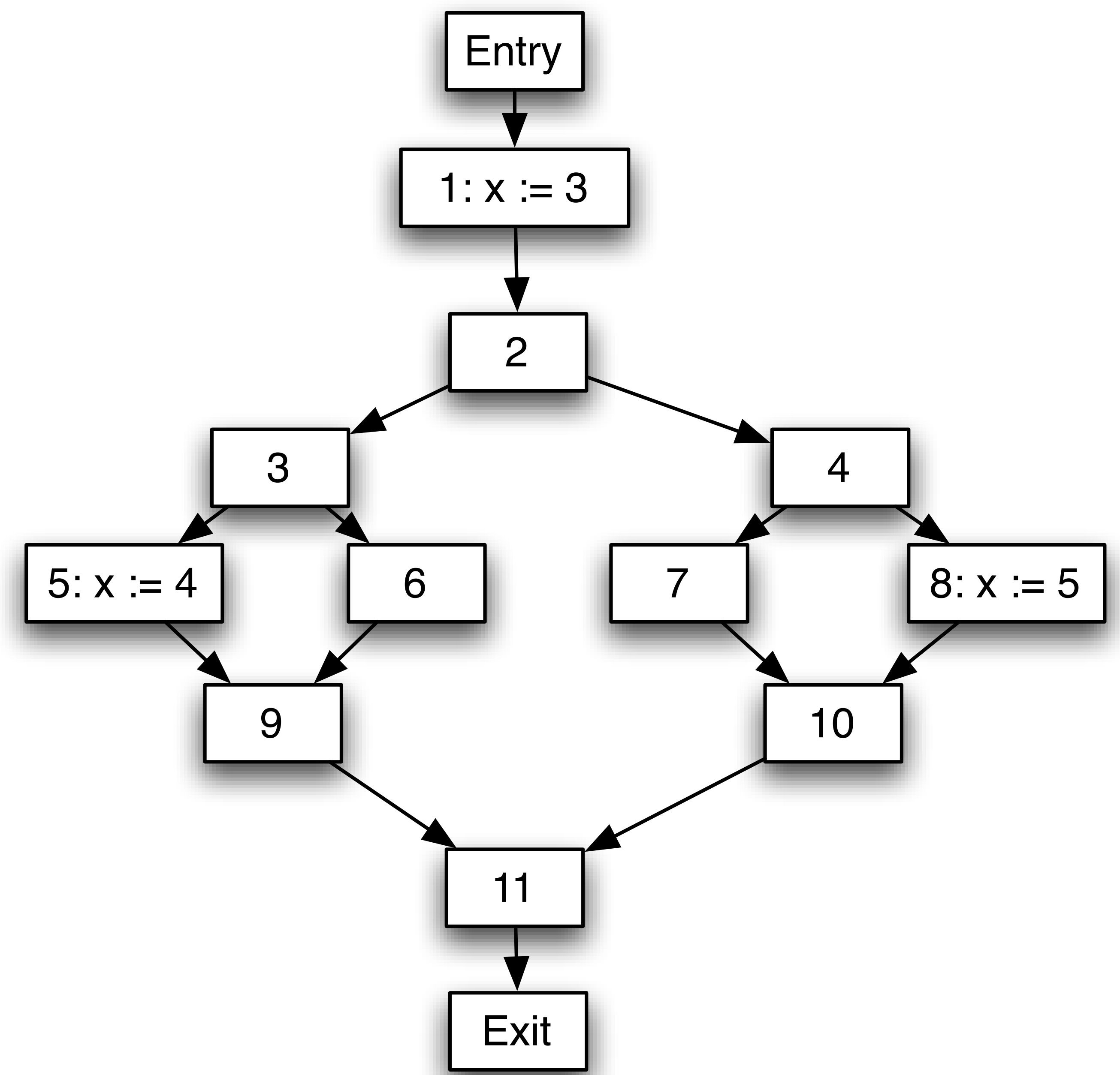
Computing SSA Form

- Step 1: Compute the dominance frontier
- Step 2: Use dominance frontier to place Φ nodes
- Step 3: Rename variables so only one definition per name

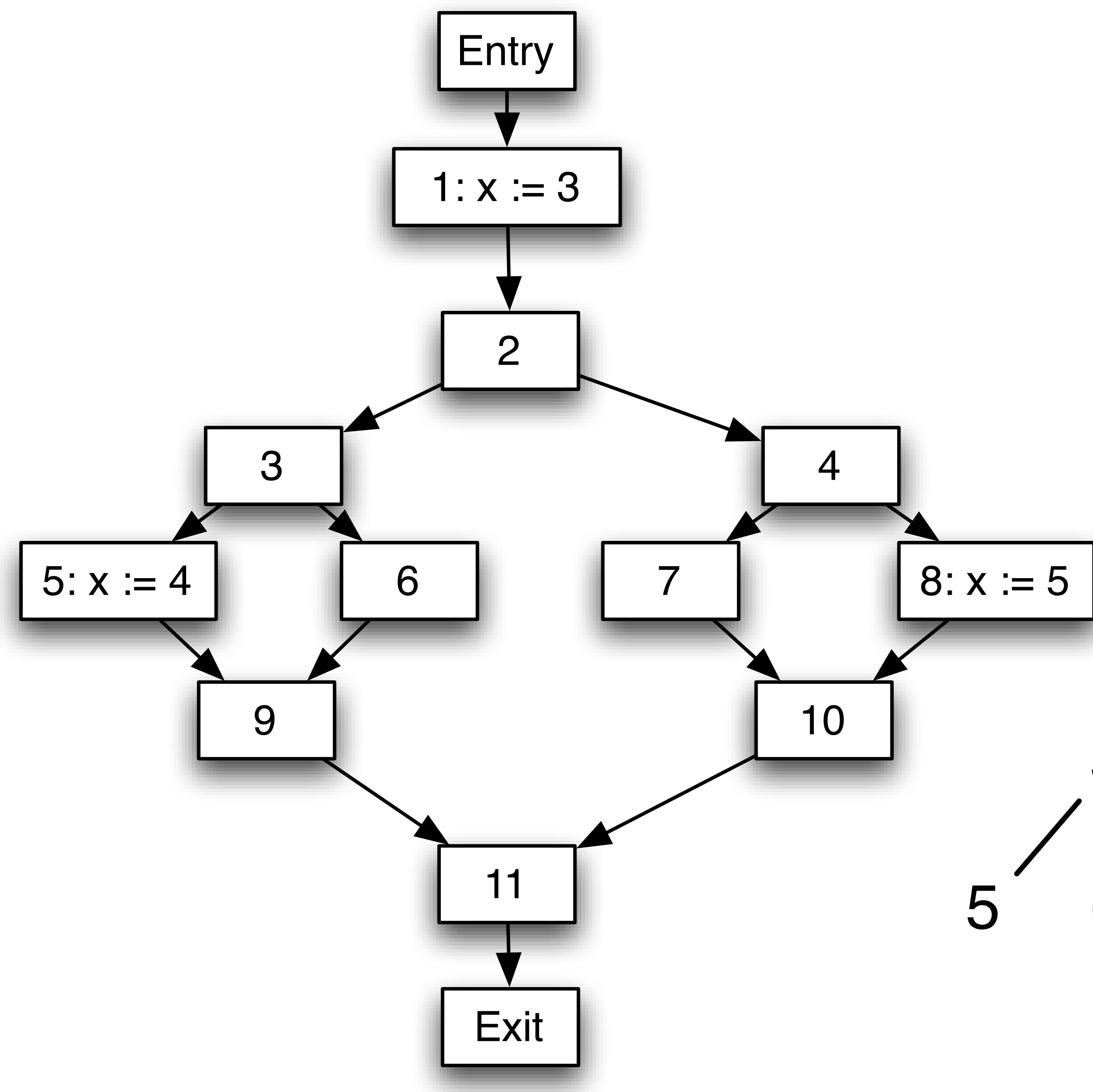
Step 2: Placing Φ Functions for v

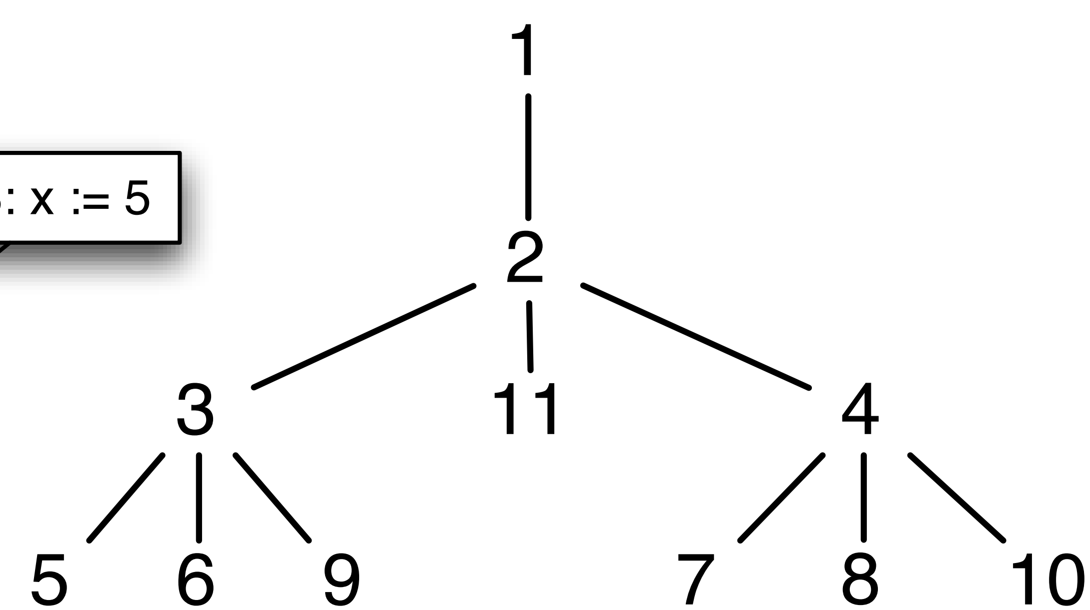
- Let S be the set of nodes that define v
- Need to place Φ function in every node in DF(S)
  - Recall, those are all the places where the definition of v in S and some other definition of v may meet
- But a Φ function adds another definition of v!
  - v := Φ(v, ..., v)
- So, iterate
  - $DF_1 = DF(S)$
  - $DF_{i+1} = DF(S \cup DF_i)$

Example

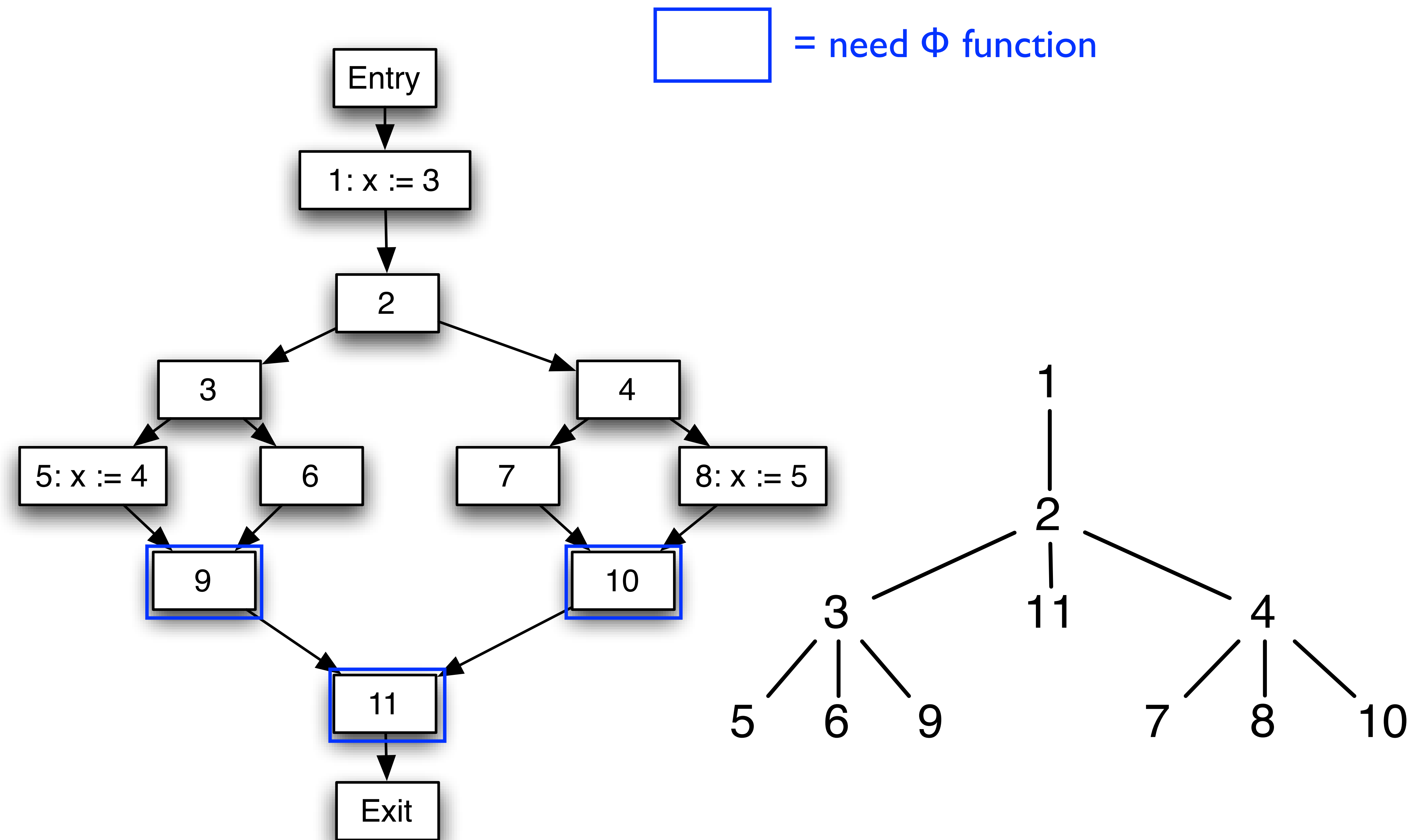


Example





Example

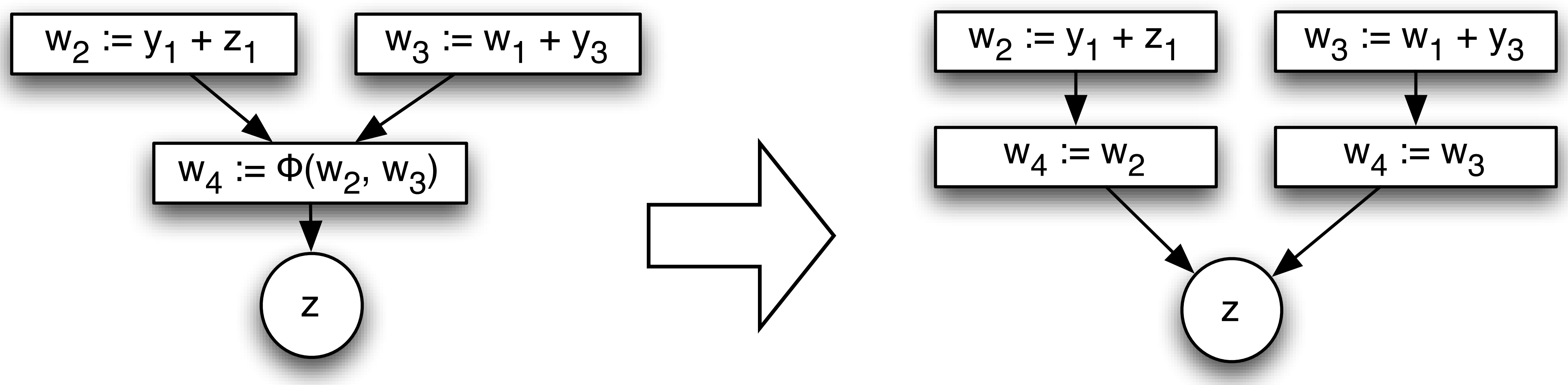

= need Φ function
Entry
1: x := 3
2
3
4
5: x := 4
6
7
8: x := 5
9
10
11
Exit
1
2
3
11
4
5
6
9
7
8
10

Step 3: Renaming Variables

- Top-down (DFS) traversal of dominator tree
  - At definition of v, push new # for v onto the stack
  - When leaving node with definition of v, pop stack
  - Intuitively: Works because there's a Φ function, hence a new definition of v, just beyond region dominated by definition
- Can be done in $O(E+|DF|)$ time
  - Linear in size of CFG with Φ functions

Eliminating Φ Functions

- Basic idea: Φ represents facts that value of join may come from different paths
  - So just set along each possible path



Eliminating Φ Functions in Practice

- Copies performed at Φ fns may not be useful
  - Joined value may not be used later in the program
    - (So why leave it in?)
- Use dead code elimination to kill useless Φs
- Subsequent register allocation will map the (now very large) number of variables onto the actual set of machine register

Efficiency in Practice

- Claimed:
  - SSA grows linearly with size of program

Table I. Summary Statistics of Our Experiment

Package name	Statements in all procedures	Statements per procedure: Min	Median	Max	Description
EISPACK	7,034	22	89	327	Dense matrix eigenvectors and values
FLO52	2,054	9	54	351	Flow past an airfoil
SPICE	14,093	8	43	753	Circuit simulation
Totals	23,181	8	55	753	221 FORTRAN procedures

Cytron, Ferrante, Rosen, Wegman, and Zadeck, Efficiently Computing Static Single Assignment Form and the Control Dependence Graph, TOPLAS 13(4), Oct 1991.

Efficiency in Practice (cont'd)

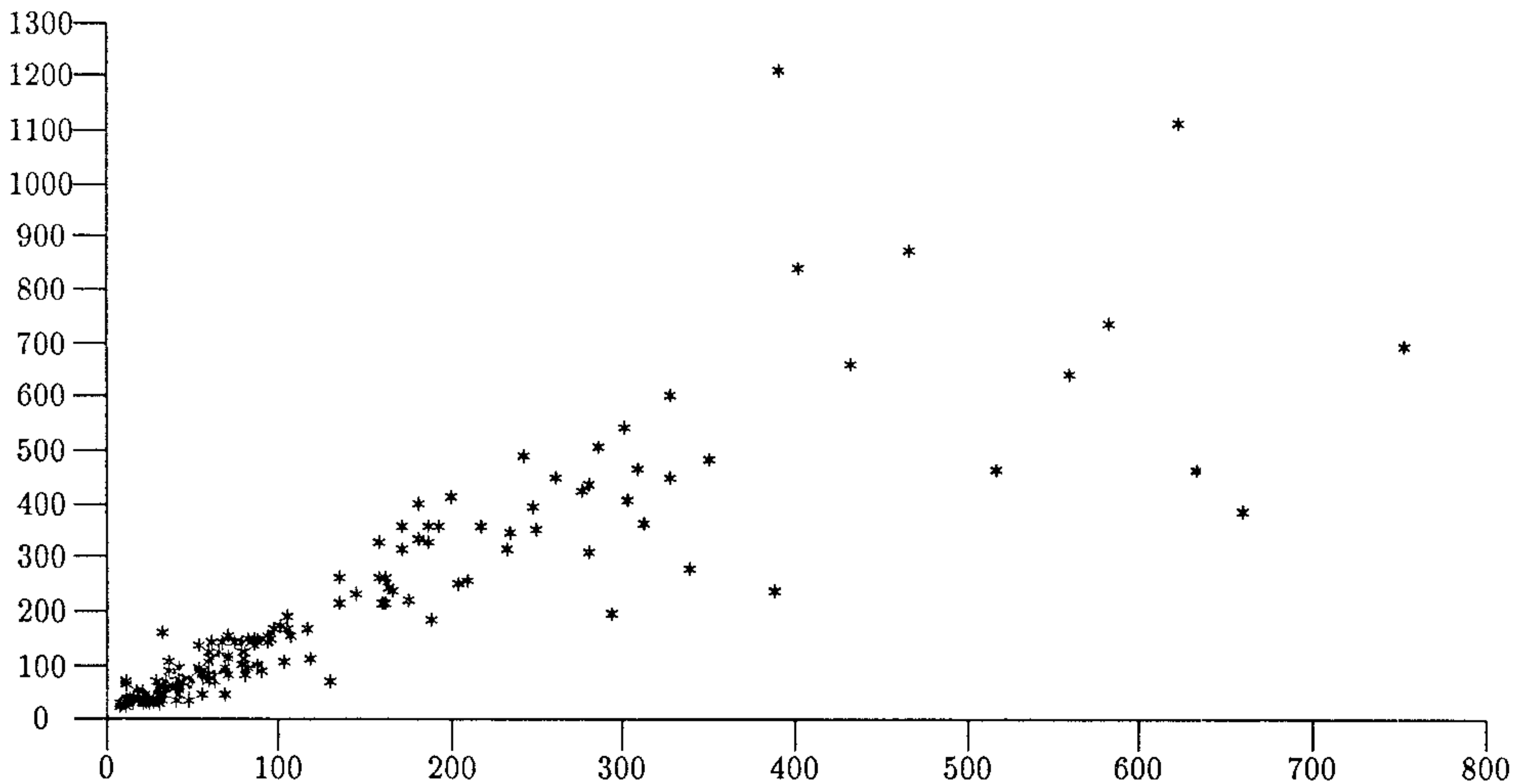


Fig. 21. Number of ϕ-functions versus number of program statements.

- Convincing?

Arrays

- Need to handle array accesses
  - A[i] := A[j] + B[k]
- Problem: How do we know whether A[i], A[j], and B[k] are all distinct?
  - Could have A=B, e.g., foo(int A[], int B[]){} ... foo(a,a)
  - Could have i=j
- History: significant research on determining array dependencies, for parallelizing compilers

Arrays (cont'd)

- One possibility: treat arrays as single variables
  - Then don't need to worry about updates to them

```
* := A(i);          * := A(i);
A(j) := V;          A := Update(A, j, V);
* := A(k) + 2;      T := A(k);
                    * := T + 2;
```

- Update(A, j, V) makes a copy of A
  - Then try to collapse unnecessary copies

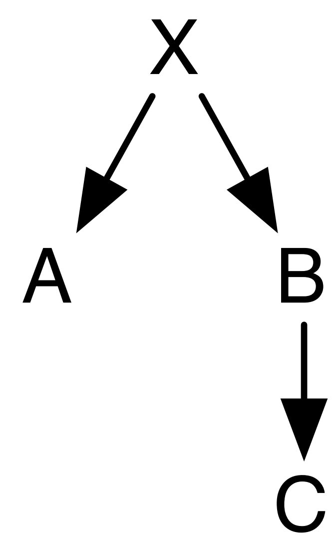
- Structures are arrays with constant indexes

Pointers

- For each statement S, let
  - MustMod(S) = variables always modified by S
  - MayMod(S) = variables sometimes modified by S
    - So if $v \notin$ MayMod(S), then S must not modify v
  - MayUse(S) = variables sometimes used by S
- Then assume that statement S
  - writes to MayMod(S)
  - reads MayUse(S) ∪ (MayMod(S) - MustMod(S))
- Convincing? We'll talk more about pointers later

Dominators for Control Dependence

- As mentioned earlier, dominators can be used to compute control dependences
- Y is *control dependent* on X if whether Y is executed depends on a test at X



- A, B, and C are control dependent on X

Postdominators and Control

- Y *postdominates* X if every path from X to Exit contains Y
  - I.e., if X is executed, then Y is always executed
- Then, Y is control dependent on X if
  - There is a path $X \rightarrow Z_1 \rightarrow \cdots \rightarrow Z_n \rightarrow Y$ such that Y postdominates all Z_i and
  - Y does not postdominate X
  - I.e., there is some path from X on which Y is always executed, and there is some path on which Y is not executed

Dominance Frontiers, Take 2

- Postdominators are just dominators on the CFG with the edges reversed
- To see what Y is control dependent on, we want to find the Xs such that in the reverse CFG
  - There is a path $X \leftarrow Z_1 \leftarrow \cdots \leftarrow Z_n \leftarrow Y$ where
    - for all $i, Y \geq Z_i$ and
    - $Y \not\geq X$
- I.e., we want to find $DF(Y)$ in the reverse CFG!