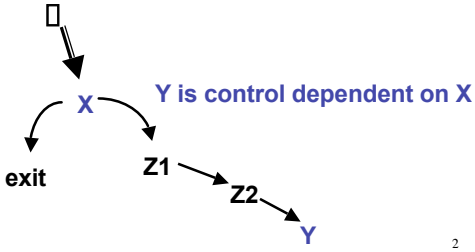
Static Single Assignment Form

- What is *control dependence?*
  - Dominators, postdominators
- SSA form - each use has one reaching defn
  - Dominance frontier

R. Cytron, J. Ferrante, B. Rosen, M. Wegman, K. Zadeck, "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph", ACM Toplas, vol 13, no 4, Oct 1991, pp 451-490.

Control Dependence

- Node Y is *control dependent* on node X means there is a logical test at X whose outcome determines if Y is executed.
- Y *postdominates* Z iff every execution path from Z to program exit includes Y (analogous to domination on the reverse control flow graph)



Control Dependence

X,Y ∈ N(CFG)

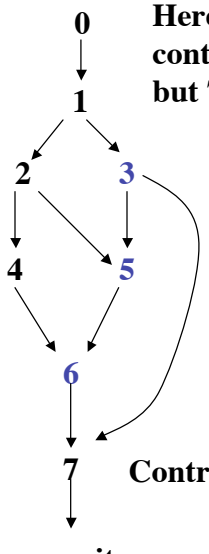
Y is control dependent on X iff

(i) ∃ path from X to Y (X, Z1, Z2,...,Zk, Y) such that ∀ Zi, Zi ≠ X, Zi is postdominated by Y, and

(ii) X is not postdominated by Y

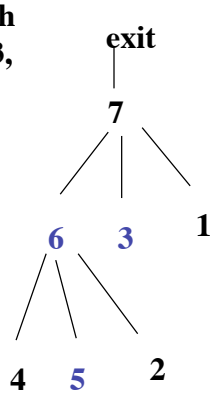
Idea: the predicate evaluated at X determines if Y executes, so once you know that X executes, you know if Y executes

Control Dependence - Example



Here nodes 5,6 are both control dependent on 3, but 7 is not.

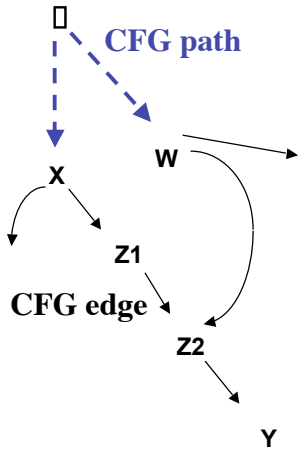
Control flowgraph



Postdominator Tree

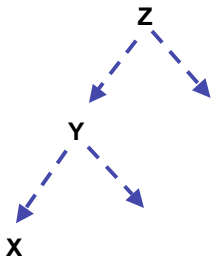
Properties

- **Relation is not unique**
  - **Y can be control dependent on more than one other CFG node**
  - **Y is control dependent on both W and X**



Properties

- **Relation is not transitive.**
  - **X is control dependent on Y, Y is control dependent on Z, but X is NOT control dependent on Z since X does not postdominateY.**



Control Dependence Algorithm

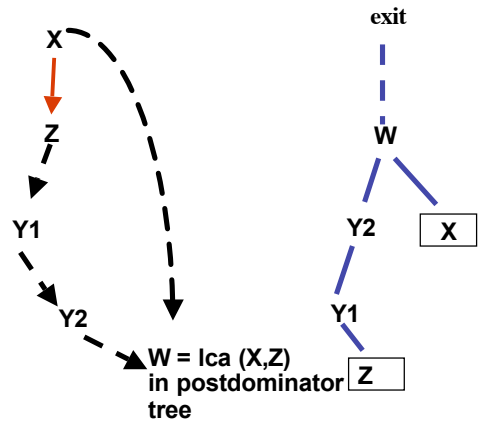
- **Intuition: look for CFG edges such that the target node does not postdominate the source node, then use the postdominator tree to find control dependences.**
- **Algorithm**
  1. **Find postdominators on CFG**
  2. **Form candidate edge set, S = { (X, Z) ∈ G | Z is not an ancestor of X in the postdominator tree}**
  3. **Find X and Z in postdominator tree (all ancestors of Z in tree postdominate Z)**

     Find all nodes that postdominate Z but not X, {Yi}.

     Z and {Yi} are all control dependent on X.

Illustration

Find X and Z in postdominator tree; (X,Z) is candidate edge; all ancestors of Z in tree postdominate Z. Find all nodes that postdominate Z but not X, {Yi}. Then Z and {Yi} are all control dependent on X.



Postdominators

- **Calculated on reverse CFG (same nodes, all edges reversed in direction) by fixed point iteration**

```
Pdom (exit) = {exit}   /* unique exit node */
for n ∈ N - {exit} do
    Pdom (n) = N    /* Max FP calculation */
while some Pdom(n) changes do
{   for n ∈ N - {exit} do
    Pdom(n) = {n} ∪ {∩ Pdom(j) }
                    j ∈ pred(n)
}
```

- **Forward dataflow problem on reversed CFG, meet semilattice**
- **Reflexive relation**

Validation

- **Claim: Given (X,Z) candidate edge in CFG, the least common ancestor(X,Z) in postdominator tree is X or parent(X).** (Ferrante, et.al., "The Program Dependence Graph and Its Use in Optimization, TOPLAS, July 1987)

  Proof: Let W= parent(X) in postdom tree. W ≠ Z because X not postdominated by Z. Assume W does not postdominate Z. Then ∃ path from Z to *exit* not containing W. But then adding (X,Z) to that path, creates a path from X to *exit* not containing W. CONTRADICTION.

  Therefore, W postdominates Z.

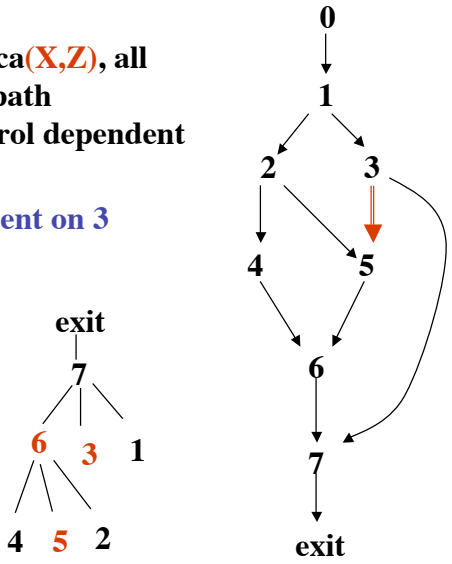
  Therefore, W is ancestor(Z) in postdom tree.

  Therefore, W or X is least common ancestor (X,Z) in postdom tree. qed.

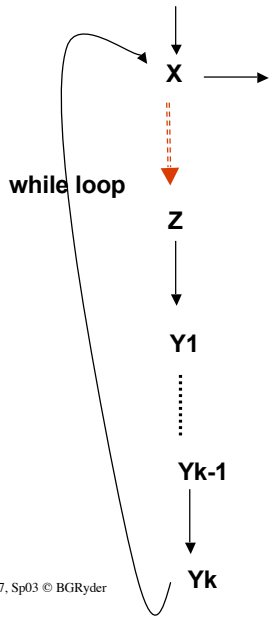
Case 1

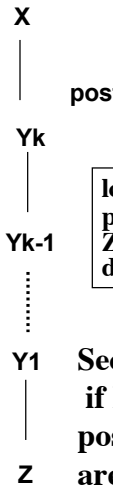
First case: if parent(X) = lca(X,Z), all nodes on postdom tree path (parent(X), Z] are control dependent on X.

5 and 6 control dependent on 3



Case 2: Loops



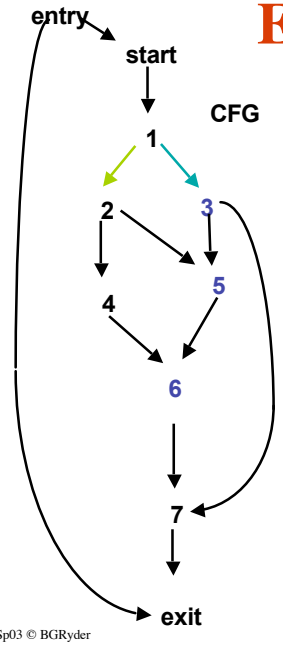


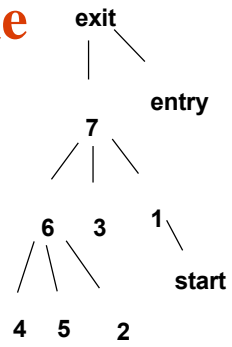
postdominator subtree

lca(X,Z) =X, and Z does not postdominate X.
Z,Y1,...,Yk are all control dependent on X.

Second case:
if X = lca(X,Z) all nodes on postdom tree path (X, Z] are control dependent on X.

Example

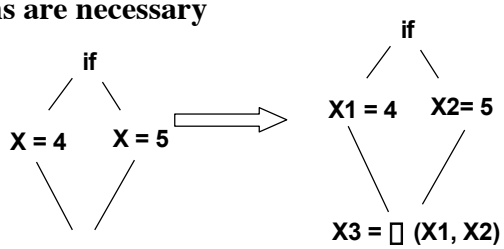




Find all edges (X,Z) st Z does not postdominate X.
(1,2) mark {2,6 } cd on 1.
(1,3) mark { 3 } cd on 1.
(2,4) mark { 4 } cd on 2.
(2,5) mark { 5 } cd on 3.
(3,5) mark { 5,6 } cd on 3.
(entry,start) mark {start,1,7} cd on entry.

Static Single Assignment

- **Idea:** each assignment of variables to be given a unique name; each use of a variable to be reached by only one definition of that variable
- Need to create □ functions at join nodes reached by more than one definition of same variable
  - **Dominance frontier** is a subset of these join nodes where □ functions are necessary



Dominance Frontier

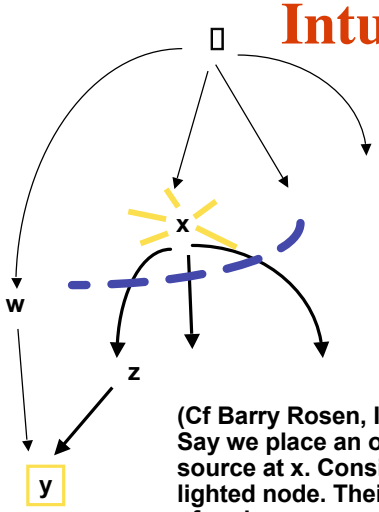
- Idom(X) is the immediate dominator of X
  - Idom(X) ≥ X means Idom(X) dominates X
  - Idom(X) >> X means Idom(X) strictly dominates X (i.e., Idom(X) ≠ X)
- *Dominance frontier of X (DF(X))* is the set of all CFG nodes Y, such that X dominates predecessor of Y in CFG, but X does not strictly dominate Y.

  DF(X) = { Y | ∃ P ∈ pred(Y), X ≥ P and ¬(X >> Y)}

Dominance Frontier

- DF of a CFG node containing an assignment to variable v, contains those nodes which other assignments to v may reach
  - These are places in the CFG where φ functions will be necessary
- *Iterated dominance frontier* is the dominance frontier of all definitions for a variable, including the added φ functions

Intuition



(Cf Barry Rosen, IBM TJ Watson Research Center)
Say we place an opaque shade at node x with a light source at x. Consider all dark edges incident on a lighted node. Their targets form the dominance frontier of node x.

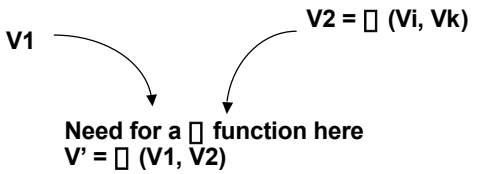
SSA Algorithm

- **Intuitive view:**
  - **Find dominance frontiers for all nodes in CFG**
  - **Locate where functions are needed for all variables v**
    - **Find all definitions of variable v**
    - **Find iterated dominance frontier for v**
  - **Rename variables, propagating definition names to their uses (v --> vi)**

SSA Form

- Two step translation
  - Put Φ functions at some control merge nodes
  - Rename variables in branch expressions or on either side of an assignment including Φ assignments

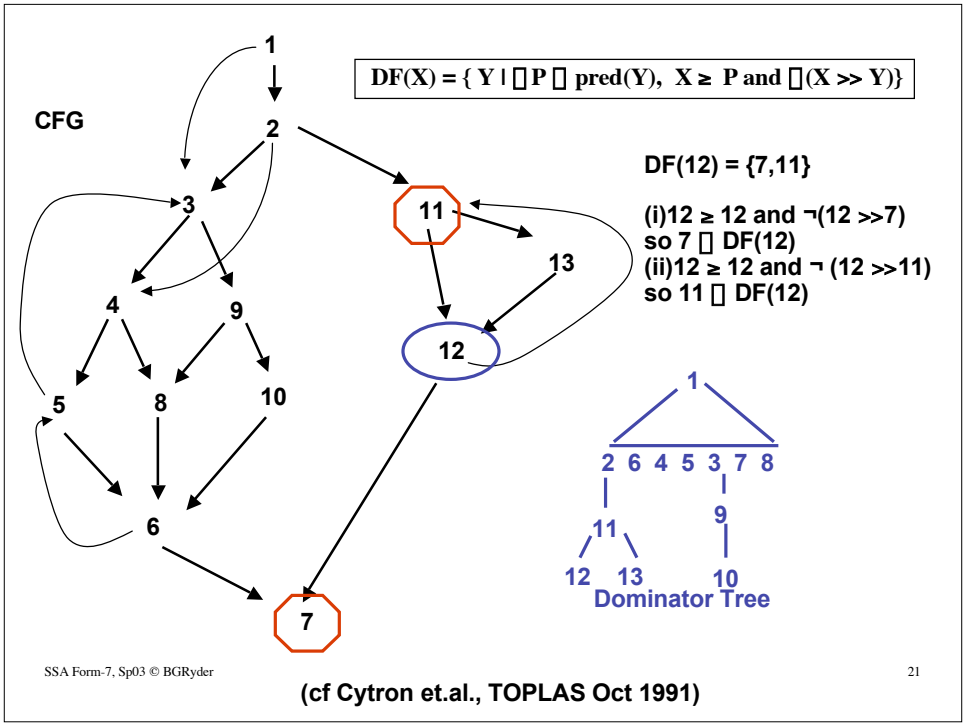
    (V --> Vi)



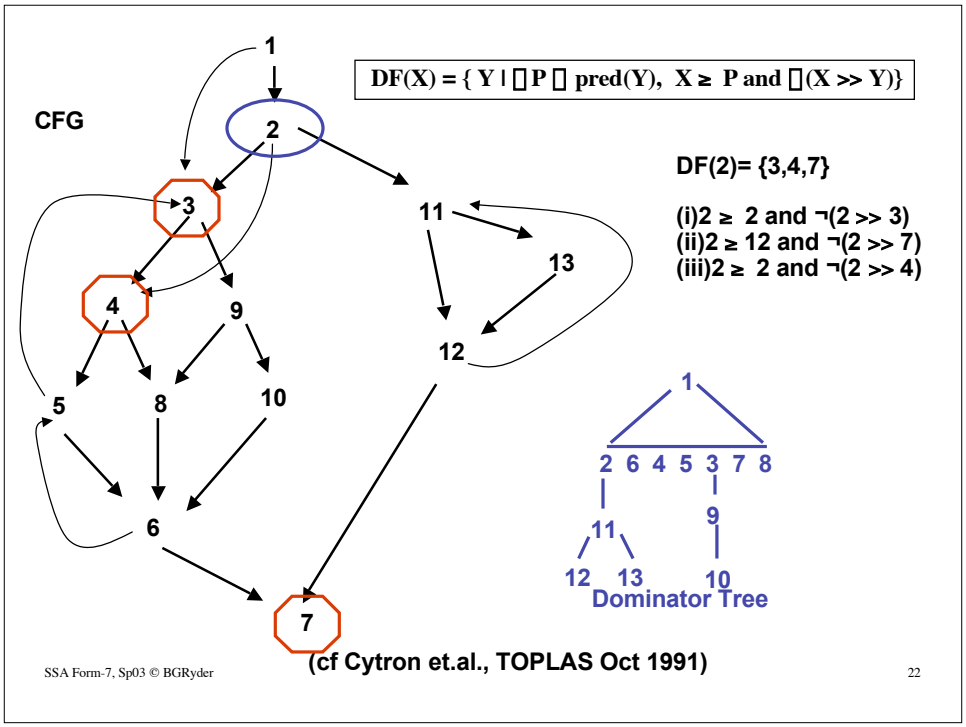
- *Minimal* SSA form
  - Each use of a variable is target of only one assignment statement in the program text
  - Number of Φ functions inserted is as small as possible

SSA

- *Pruned* SSA form
  - Only creates Φ functions at program join points p for a variable that has live uses at p or after p.
- Special cases: arrays, structures, aliasing
- Main work in SSA algorithm: figure out where to put the Φ functions?



(cf Cytron et.al., TOPLAS Oct 1991)

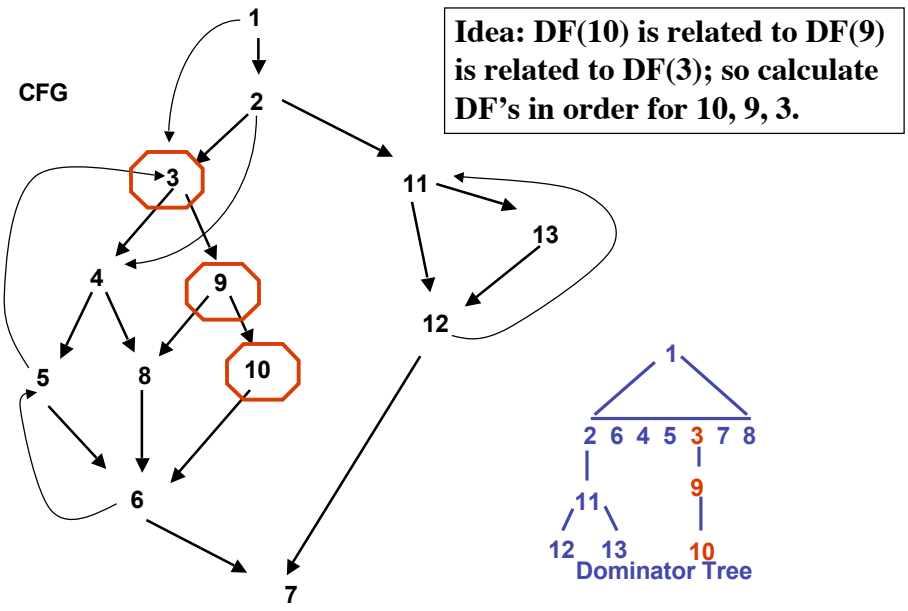


(cf Cytron et.al., TOPLAS Oct 1991)

Initial Algorithm for DF

Give control flowgraph G and its dominator tree.
1. DomBy(X) = {Z | X is ancestor of Z in dominator tree}
2. foreach (X ∈ N) do
3. foreach Y ∈ DomBy(X) do
{ foreach Z ∈ succ(Y) do
if ¬ (Z ∈ (DomBy(X) - {X})) then
DF(X) <-- DF(X) ∪ {Z}
}

- **This is inefficient because we are reusing the dominance relation for every node in steps 1 and 3. we could accomplish the same work just knowing the immediate dominance relation, if we processed the nodes in a 'good' order in step 2.**



Improved DF Calculation

- Define $DF(X) = DF_{local}(X) \cup DF_{up}(Z)$ where
  Z is child(X) in dominator tree
  $DF_{local}(X) = \{Y \in succ(X) \mid \neg(X >> Y)\}$
  $DF_{up}(Z) = \{ Y \in DF(Z) \mid \neg(idom(Z) >> Y)\}$
- Then can recursively calculate DF for all nodes in a bottom up traversal of the dominator tree.
- Easier, equivalent conditions to use
  $DF_{local}(X) = \{Y \in succ(X) \mid idom(Y) \neq X \}$
  $DF_{up}(Z) = \{ Y \in DF(Z) \mid idom(Y) \neq parent(Z)\}$

Improved DF Algorithm

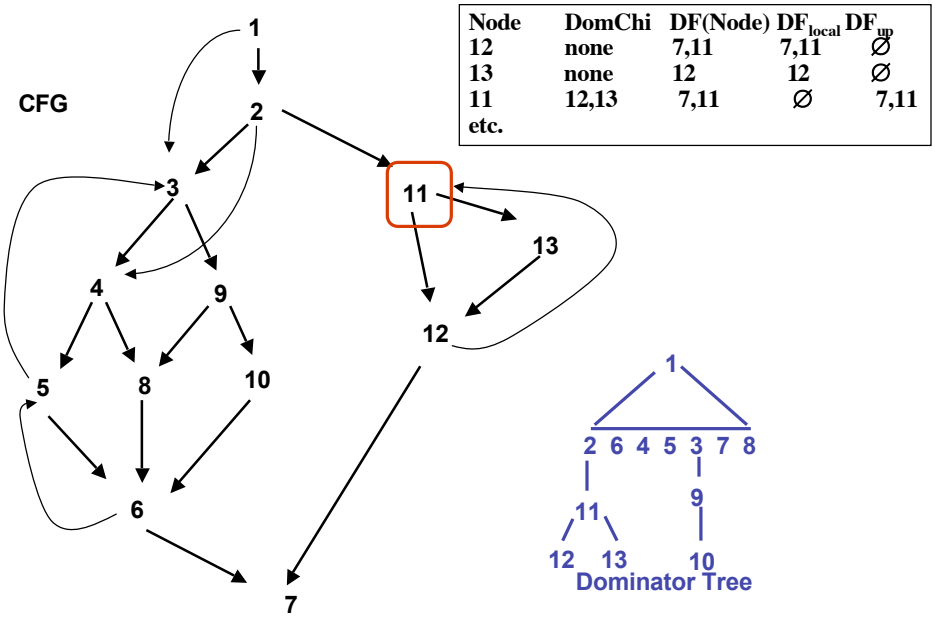
Traverse dominator tree in bottom up order; at node X do:

DF(X) <-- ∅

foreach Y ∈ succ(X) do
  if (idom(Y) ≠ X) then DF(X) <-- DF(X) ∪ {Y}

foreach Z ∈ Children (X) do
  foreach Y ∈ DF(Z) do
    if (idom(Y) ≠ X) then DF(X) <-- DF(X) ∪ {Y}

Note: no storage needed for DF_{local} or DF_{up}



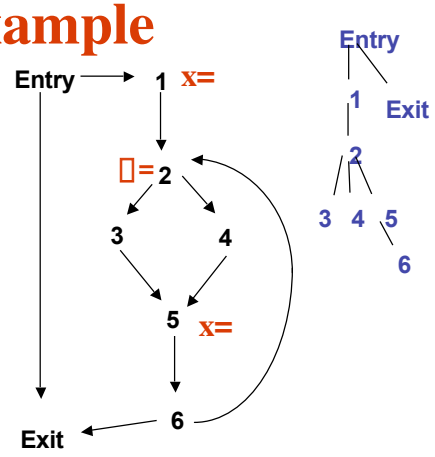
Node	DomChi	DF(Node)	DF_{local}	DF_{up}
12	none	7,11	7,11	∅
13	none	12	12	∅
11	12,13	7,11	∅	7,11
etc.				

Where to place the φ's?

- Want dominance frontiers of all nodes containing φ functions or source level definitions for a particular variable v
- *Iterated Dominance Frontier*
  - Let S = all nodes containing a definition of variable v in program; then *DF(S)* will be set of nodes where φ functions for variable v should be place initially.
  - $DF_1 = DF(S)$
  - $DF_{i+1} = DF(S \cup DF_i)$
- We need to calculate the iterated dominance frontier with respect to every variable in the program to convert it into SSA form

Example

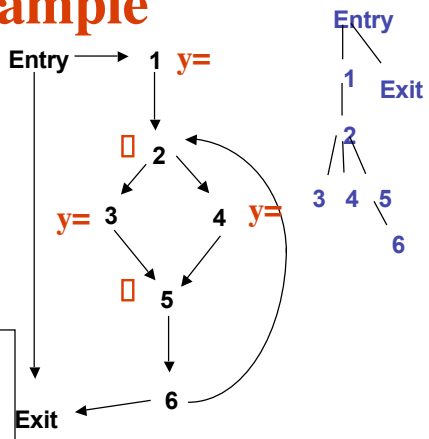
```
x = 1;
y = 1;  /*1*/
Repeat /*2*/
   if( )then y = x /*3*/
   else y = x + 1; /*4*/
   print (x,y); /*5*/
   x = 2 * x;
until (  );  /*6*/
```



Find defns of x.
Calculate DF(1), DF(5):
DF(1) = {Exit}; DF(5) = { 2,Exit }
Calculate the iterated DF for x: $DF_1 = DF(1,5) = \{ 2 \}$
$DF_2 = DF(1,2,5) = \{ 2 \}$
so $DF_2 = DF_1$

Example

```
x = 1;
y = 1;  /*1*/
Repeat /*2*/
   if( )then y = x /*3*/
   else y = x + 1; /*4*/
   print (x,y); /*5*/
   x = 2 * x;
until (  );  /*6*/
```



Find defns of y.
Calculate DF(1), DF(3), DF(4):
DF(1) = {Exit}; DF(3) ={5}; DF(4) = {5}
Calculate the iterated DF for y:
$DF_1 = DF(1,3,4) = \{ 5 \}$
$DF_2 = DF(1,3,4,5) = \{ 5,2 \}$
$DF_3 = DF_2$

Variable Renaming

new code:

```
X1 = 1;
Y1 = 1;
repeat
   X2 = Φ(X1, X3);
   Y2 = Φ(Y1, Y5);
    if ( ) then Y3 = X2
          else Y4 = X2 + 1;
   Y5 = Φ(Y3, Y4);
   print (X2, Y5);
   X3 = 2*X2;
until (  );
```

old code:

```
x = 1;
y = 1;  /*1*/
Repeat /*2*/
   if( )then y = x /*3*/
   else y = x + 1; /*4*/
   print (x,y); /*5*/
   x = 2 * x;
until (  );  /*6*/
```

- Renaming done through top down traversal of the dominator tree; process user definitions and Φ functions in same traversal;
- For each variable Z need stack for current index i for Zi and count of total # defs for Z seen so far

More on control dependence

- Dominance frontiers in reverse CFG yield the control dependence relation
  - Y is control dependent on X in CFG *iff* X ∈ DF(Y) on reverse CFG
    - For structured programs (i.e., *whiles, ifs*, straight-line code), DF(n) contains ≤ 2 nodes; therefore, every node is control dependent on at most 2 other nodes in structured programs
  - If there is a non-null path from X to Y such that Y postdominates every node after X on that path in the CFG, then Y will dominate those nodes, but not X in the reverse CFG
  - If Y does not strictly postdominate X in the CFG, then Y will not strictly dominate X in the reverse CFG

Experimental Observations

- Size of dominance frontier varies linearly with size of program
- Size of control dependence graph is linear in size of program
- Found no relation between program size and
  - total #assignments in final program/ total # assignments in original program
  - total # defns and uses in final program/total # defns and uses in original program
- Claimed number of □ functions varies linearly with program size (Cytron et.al., Fig 21)

Complexity

- Each assignment statement is mapped to a tuple of identifiers <u,v,z..> = <...>, same length tuple of expressions
- Finding DFs for CFG takes $O(E + \sum_n |DF(n)|)$ *0*
- Let $A_{total}(n) = A_{original}(n) + A_{\square}(n)$

Then work of □ function placement (i.e., finding iterated dominance frontiers) is $O(\sum_n (A_{total}(n) * |DF(n)|))$ *1*

where |DF(n)| is observably small in practice, so this is effectively $O(A_{total}(n) * averDF)$, for averDF being weighted average of |DF(X)| sizes

- Renaming takes $O(M_{tot})$ *2* where M_{tot} is total number of variable occurrances in resulting program
- So worst case complexity is the sum of these three terms.

Optimizing the DF algorithm

"Computing □ nodes in Linear Time Using DJ graphs", V. Shreedhar and G. Gao, JI Programming Languages 1995

- **Idea: if Y is an ancestor of X in the dominator tree, then if DF(X) has been computed, you can use it to compute DF(Y).**
  - **Yields a linear time algorithm for □ placement**
- **Use new graphical representation, *DJ graph*, related to dominator tree**
  - **Nodes are same as CFG**
  - **Edges are either dominator tree edges or join edges**
    - **CFG edge (X,Y) is a *join edge*, if X does not strictly dominate Y (Y called *join node*)**

DJ Graph

"Computing □ nodes in Linear Time Using DJ graphs", V. Shreedhar and G. Gao, JI Programming Languages 1995

- **DJ graph built from dominator tree and CFG**
  - **D-edges: from dominator tree (entire tree included)**
  - **J-edges: all (X,Y) from CFG st □(X>>Y)**
    - **join edges, Y is join node**
    - **Find by marking all edges (X,Y) where X□=idom(Y)**
  - **Essentially, DJ graph is dominator tree augmented with join edges**

DJ Graph

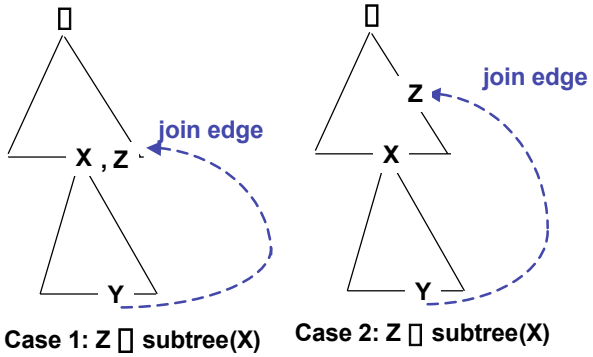
- DJ graph is linear in size of CFG (Thm 3.1) because $|E_{DJ}| < |N| + |E|$
- Assign *level number* to each dominator tree node, equal to its depth in dominator tree from root
  - Use these to order the DF computations
- if X → DJ graph, then X.level ≥ Y.level,

→ Y → DF(X) and → Y → IDF(X) (Thm 3.2)

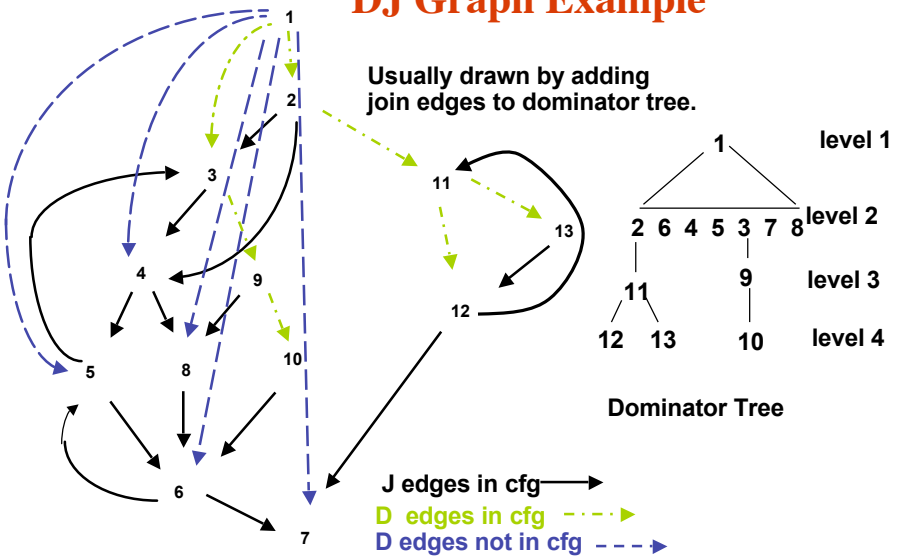
- **Idea:** calculate DF(X) in bottom up order on dom tree; Use D-edges to order the DF calculation; use J-edges to identify the → nodes

DJ Graph

- A node Z → DF(X) *iff* → Y → subtree of dom tree rooted at X with (Y,Z) as a J-edge and Z.level ≤ X.level (lemma 3.1)



DJ Graph Example



Naive DF Calculation

DF(X) = ∅
∀Y ∈ Subtree(X)
{ If ((Y,Z) is a J edge)
if (Z.level ≤ X.level) DF(X) = DF(X) ∪ {Z}
}

- If use algorithm to compute DF's for all nodes and then apply definition of iterated DF (IDF), have a quadratic method
- But we can alter this approach to compute IDF in linear time ∝ CFG nodes
- Key observations
  - Order of DF computation wrt dominator tree is crucial
  - Can limit J edges to check to compute DF

Algorithm Defns

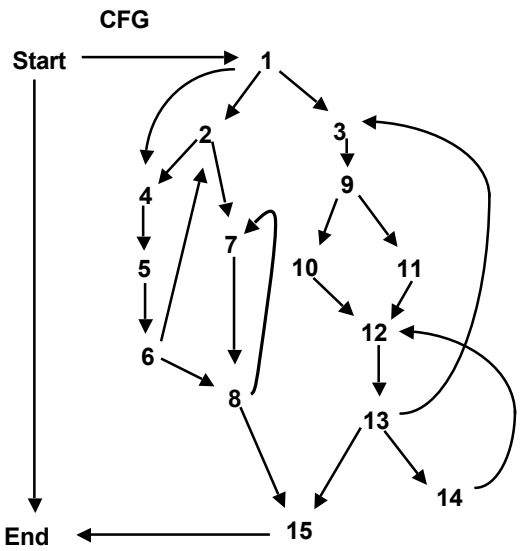
- **Each node can be marked**
  - **(visited, not visited), (in $N_{\square}$, not in $N_{\square}$), (in $N_{\square}$, not in $N_{\square}$), with its level number**
- **Each CFG edge is a D or J edge**
- ***Ordered buckets* - a restricted priority queue, implemented as an array of lists**
  - **Node W with level = i is saved to be processed in OrderedBuckets.list[i]**

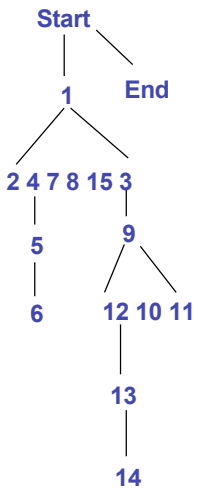
Algorithm Overview

Input: set of initial variable definition nodes($N_{\square}$); Output: IDF($N_{\square}$)= $N_{\square}$

```
/* Initialize all data structures; Insert N□ into OrderedBuckets */
while (OrderedBuckets ≠∅ ) do
    CurrentRoot = GetNode(); //removes node with max level from
    OrderedBuckets
    Mark CurrentRoot as visited;//CurrentRoot is global variable
    Visit(CurrentRoot);
endwhile;
procedure visit(X)
{   □ Y □ succ(X) in DJ graph do
    {    if ((X,Y) is J edge) then
              { if (Y.level □ CurrentRoot.level) then
    // Add Y to IDF(CurrentRoot) if not already there and mark it added
    (in N□); add Y to OrderedBuckets unless Y there already}
         else // (X,Y) is D edge; if Y is not yet visited, then mark it visited
    and execute visit(Y);
}
```

Example

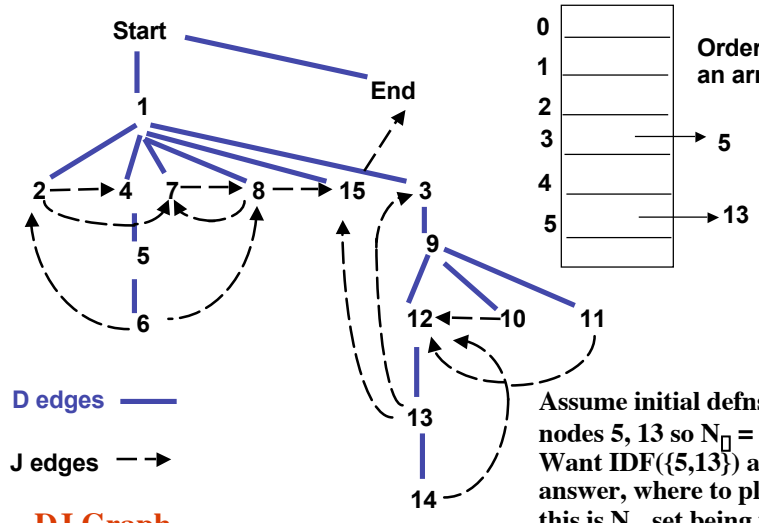




Dom Tree

"Computing □ nodes in Linear Time Using DJ graphs", V. Shreedhar and G. Gao, JI Programming Languages 1995, p196

Example

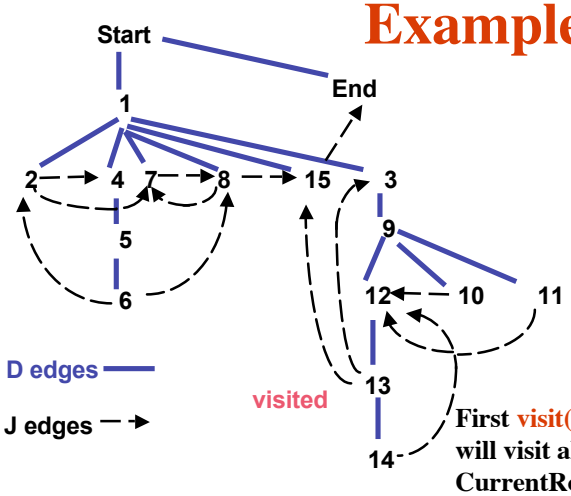


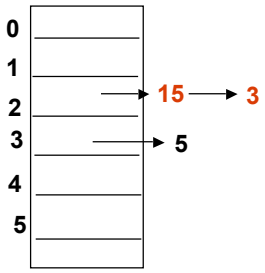
DJ Graph

OrderedBuckets, an array of lists

Assume initial defns at nodes 5, 13 so $N_{□}$ = {13,5}. Want IDF({5,13}) as final answer, where to place □ nodes; this is $N_{□}$ set being formed. Initialize OB with $N_{□}$ level(13)=5, level(5)=3.

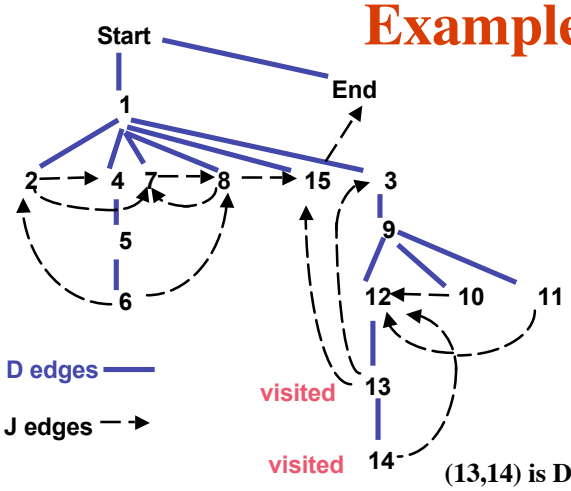
Example

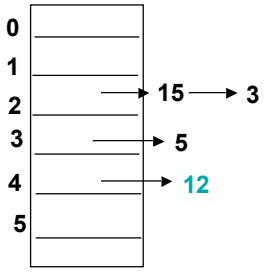




First visit(13), removing it from OB. will visit all succ(13) in DJ graph. CurrentRoot = 13. (13,15), (13,3) are J edges. test levels and find must add 15, 3 to $N_{\square}$ and OrderedBuckets;

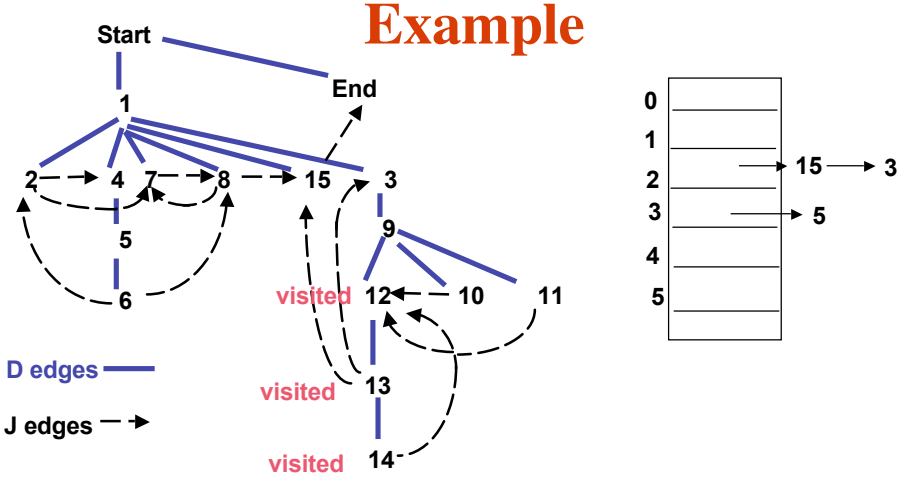
Example





(13,14) is D edge so visit(14).
(14,12) is J edge so add 12 to $N_{\square}$ and OrderedBuckets;
then visit(13) terminates.

Example



Now do visit(12); since (12,13) is D edge and 13 marked visited, DF(12) = empty and it adds nothing to $N_{\square}$. . Next do visit(5), etc.