Lecture 3 - Outline

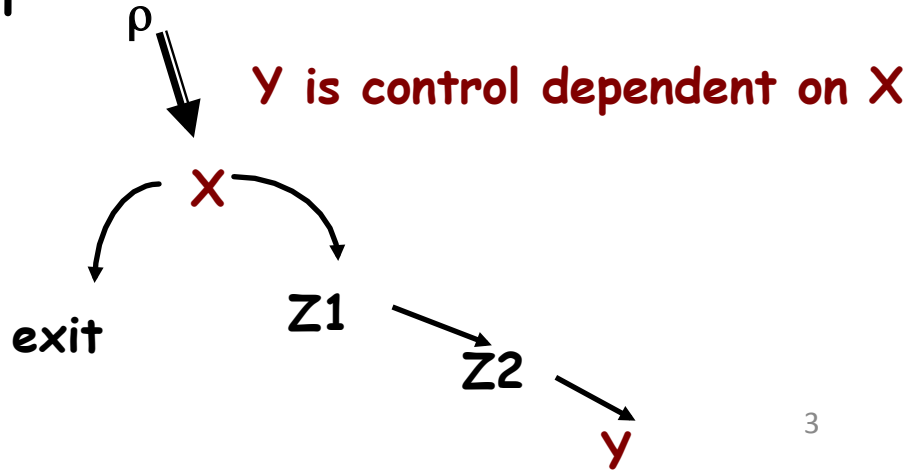
- Dependence analysis
  - Data dependence
  - Control dependence
- Program Dependence Graph (PDG) representation of code (intraprocedural)
- System Dependence Graph (SDG) representation of code (interprocedural)
- References:
  - Ferrante, et.al., "The Program Dependence Graph and Its Use in Optimization, TOPLAS, July 1987
  - Cytron, Ron; Ferrante, Jeanne; Rosen, Barry K.; Wegman, Mark N.; and Zadeck, F. Kenneth (1991). "Efficiently computing static single assignment form and the control dependence graph" (PDF). *ACM Transactions on Programming Languages and Systems* **13** (4): 451–490. doi:10.1145/115372.115320

Data Dependence

- Def-use relations define a data dependence
  - i.e., a flow from a write to a read of the value written
  - A "may" relation -- means the value at the read (use) may have been written by the write (def) OR may have been written by another write.
- The order of execution of writes must preserve data dependence relations

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)
- Y can only be control dependent on a node it does not postdominate



Intuition by Example

```
read(n);  k := 1; sum := 0;
product := 1;
while k<= n do
{     sum := sum + k;
      product := product * k;
      k := k + 1;}
write (sum);write (product);
```

Think about what decisions in the program control the execution of various statements.

The blue statements are always executed whenever this program is executed.

The red statement is executed at least once.

The orange statements are executed depending on the value of the loop test.

Control Dependence

$X, Y \in N(CFG)$

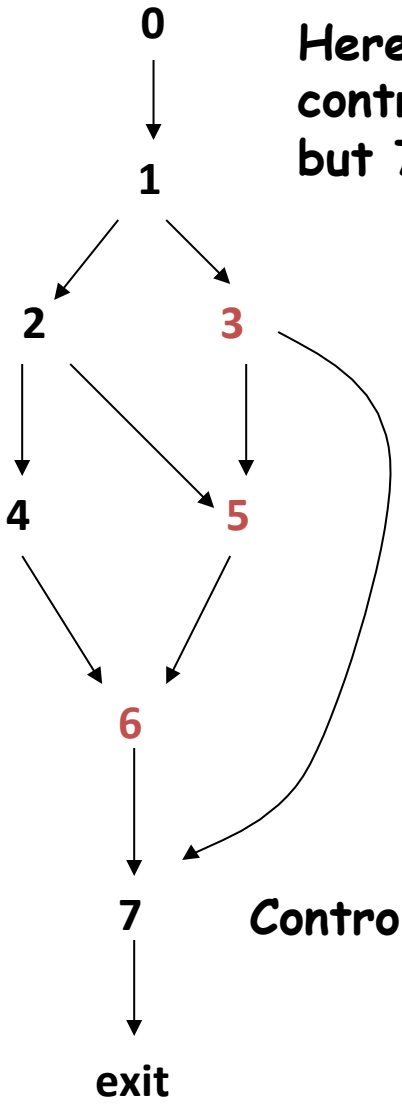
Y is control dependent on X iff

(i) ∃ path from X to Y $(X, Z_1, Z_2, \ldots, Z_k, Y)$ such that ∀ Z_i, $Z_i \neq X$, Z_i is postdominated by Y, and

(ii) X is not postdominated by Y

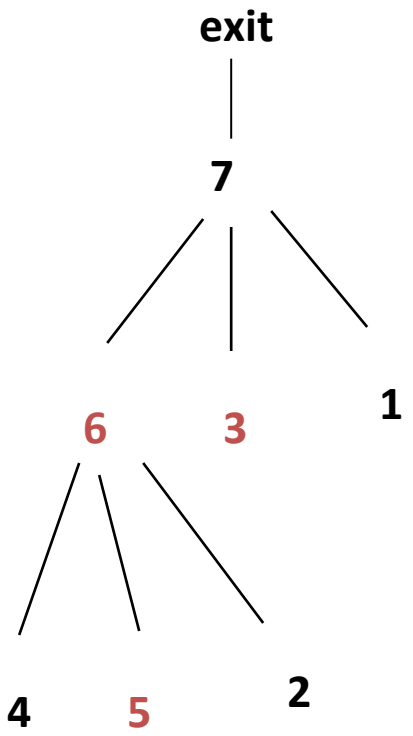
Idea: the predicate evaluated at X determines if Y executes, so once you know that X executes, you know if Y executes

Control Dependence - Example



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

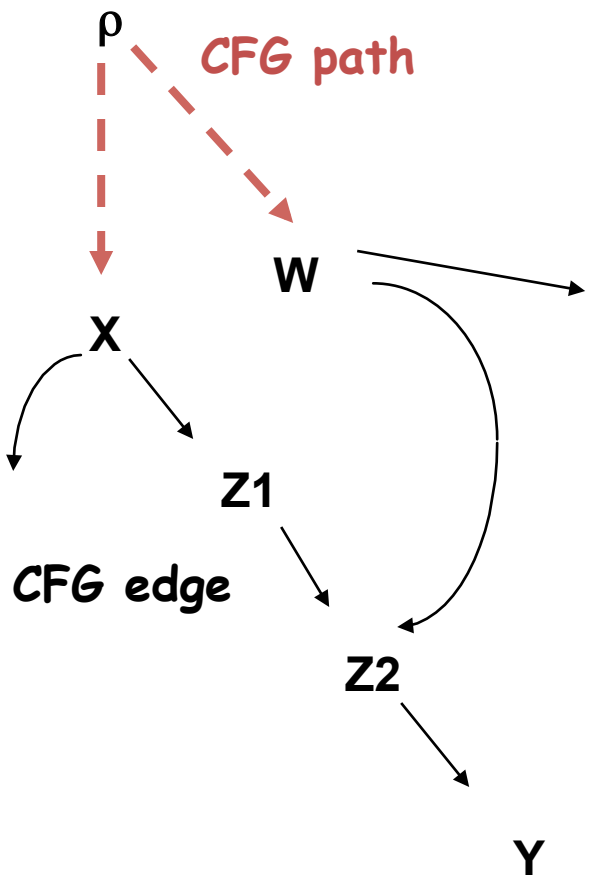
Control flowgraph



Postdominator Tree

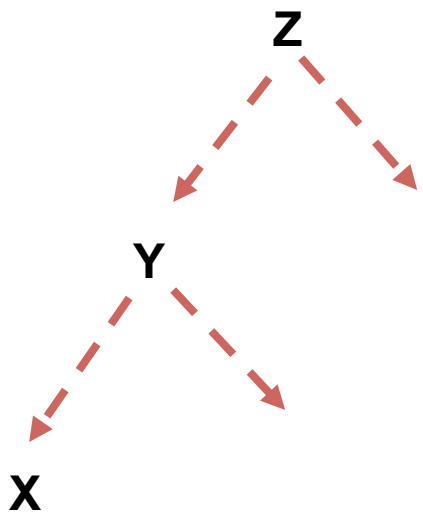
Properties

- 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 postdominate Y.



Control Dependence Algorithm

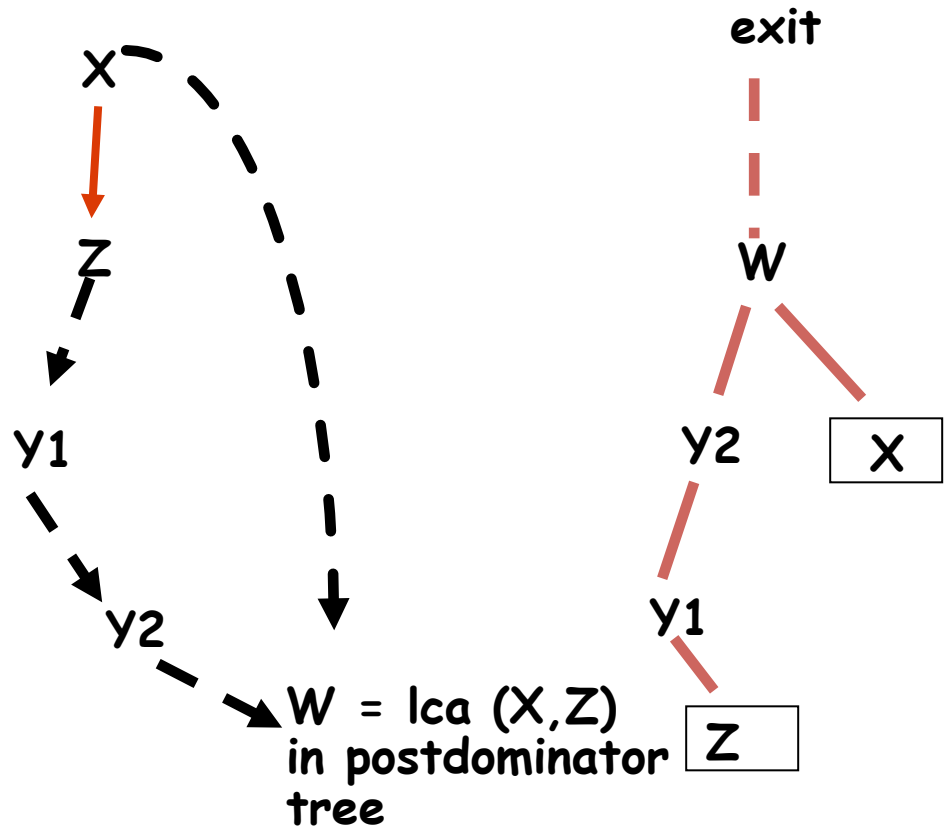
- Intuition: look for CFG edges such that the target node does not postdominate the source node, then use the postdominator tree to find control dependences.
- Algorithm
  1. Find postdominators on CFG
  2. Form candidate edge set, S = { (X, Z) $\in$ G | Z is not an ancestor of X in the postdominator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

     Z and {Y_i} 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
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.

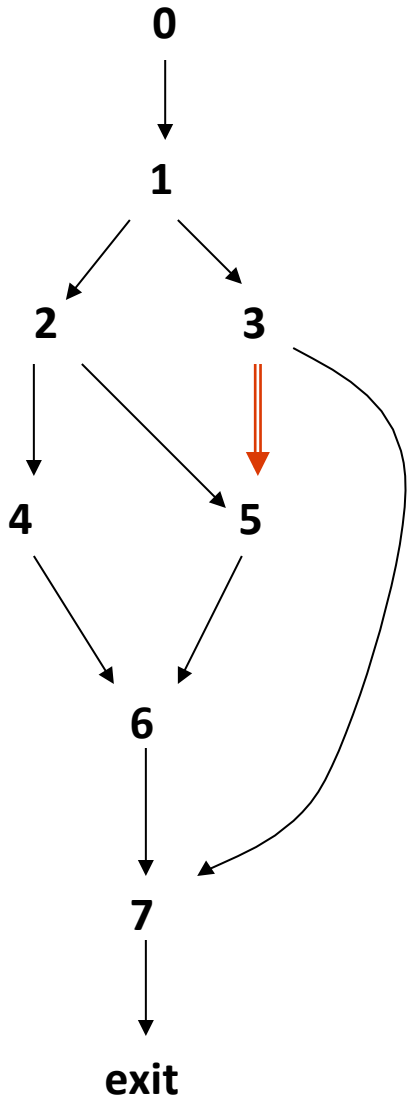
  Therefore, W is ancestor(Z) in postdom tree.

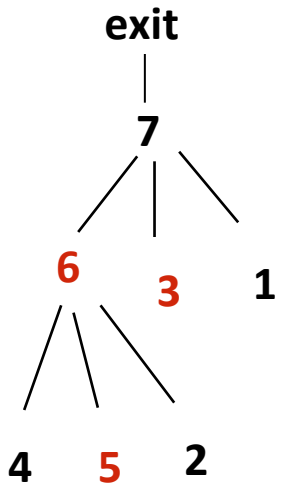
  Therefore, W or X is least common ancestor (X,Z) in postdom tree. qed.

Case 1

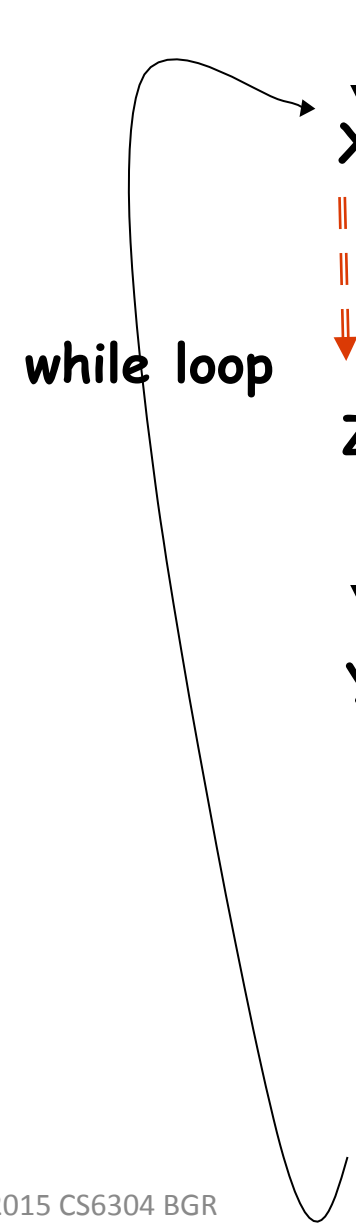
First case: if parent(X) = lca(X,Z), all nodes on postdom tree path (parent(X), Z] are control dependent on X.

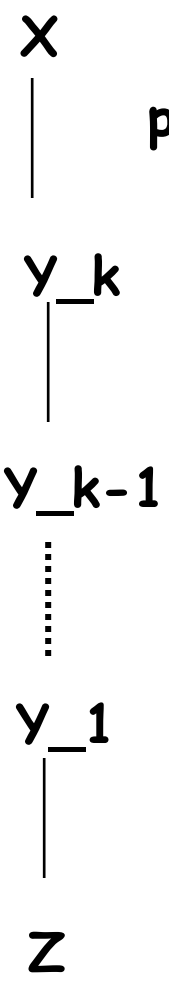
5 and 6 control dependent on 3





Case 2: Loops



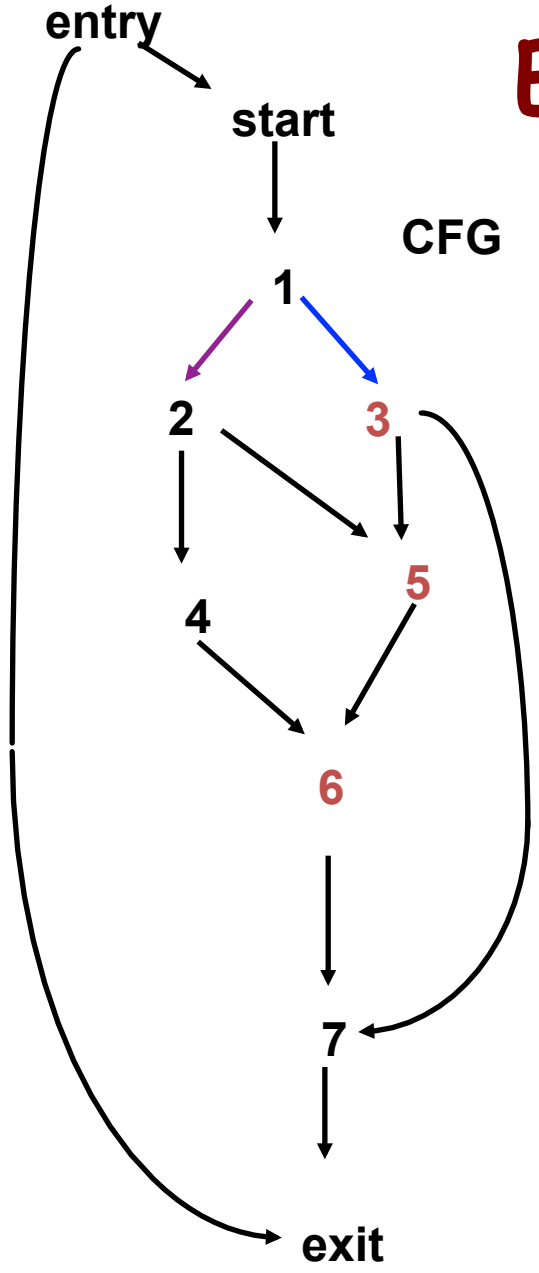


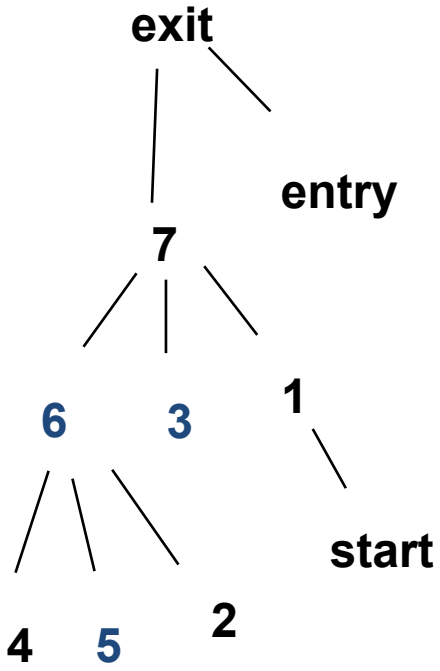
postdominator subtree

lca(X,Z) =X, and Z does not postdominate X.
Z,Y_1,...,Y_k 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 2.

(3,5) mark { 5,6 } cd on 3.

(entry,start) mark {start,1,7} cd on entry.

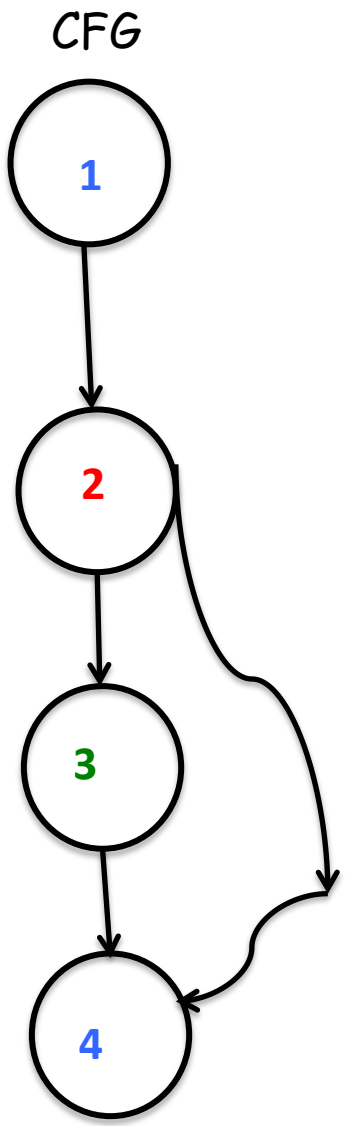
Example at Board

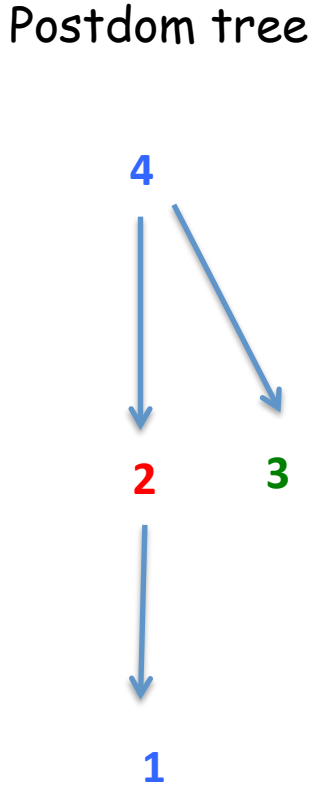
Program Dependence Graph

- A data structure that removes unnecessary sequential flow of control from a program
  - Nodes are computations (e.g., statements)
  - Edges connect computations along immediate def-use dependences and along immediate control dependences
  - Historically was used for automatic parallelization, but also uncovered relevant relations to slicing
  - Allows easier tracing of how values flow through a program – related to security information flow problems

Example

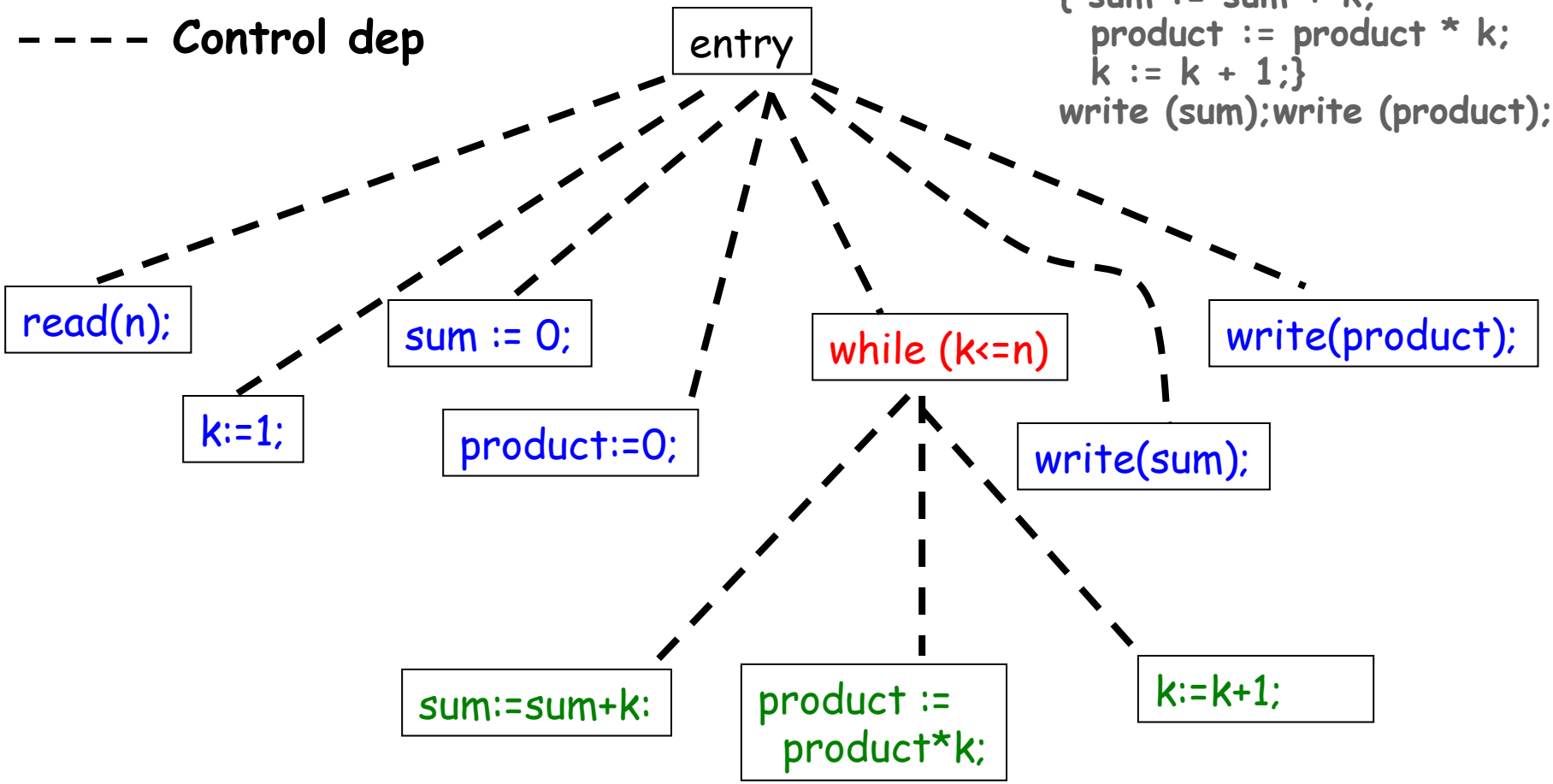
```
read(n);  k := 1; sum := 0;
product := 1;
while k<= n do
{     sum := sum + k;
      product := product * k;
      k := k + 1;}
write (sum);write (product);
```





PDG of Example

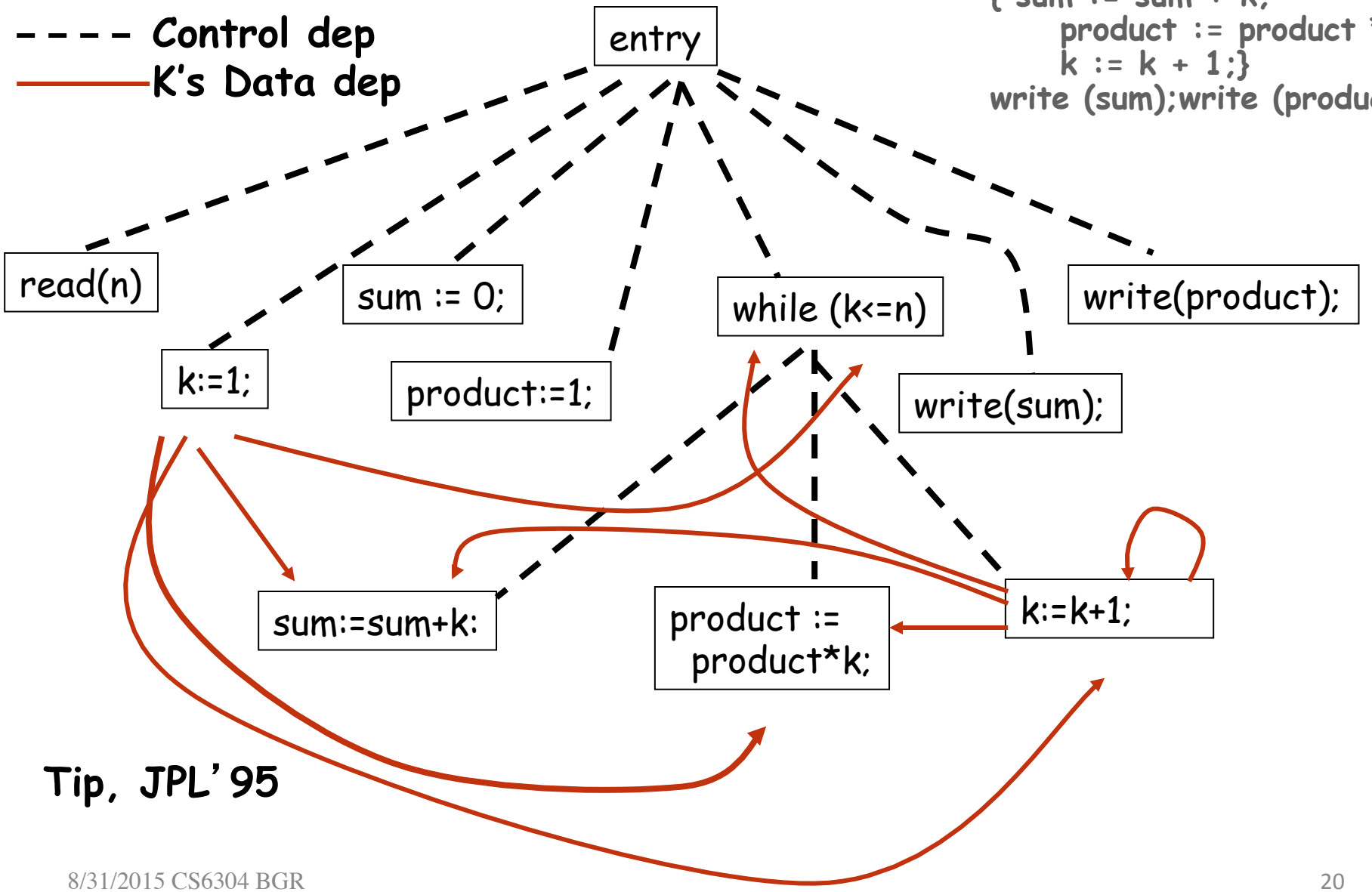
```
read(n);  k := 1; sum := 0;
product := 1;
while k<= n do
{ sum := sum + k;
  product := product * k;
  k := k + 1;}
write (sum);write (product);
```



F. Tip, "A Survey of Program Slicing Techniques", Journal Of Programming Languages,1995

PDG of Example

```
read(n);  k := 1; sum := 0;
product := 1;
while k<= n do
{ sum := sum + k;
     product := product * k;
     k := k + 1;}
write (sum);write (product);
```



Tip, JPL'95

Horwitz-Reps-Binkley Slicing

- Introduced new interprocedural program representation -- System Dependence Graph (SDG) from the PDGs of each procedure
  - Compute interprocedural summary info, adding summary edges to SDG between input and output params
  - In 2 passes, extract interprocedural slices from an SDG
- Modeled parameter passing by call by value-result
- Key idea: how to walk the graph so as to avoid infeasible interprocedural paths

Horwitz, Reps, Binkley, "Interprocedural Slicing Using Dependence Graphs", TOPLAS, Jan 1990, vol 12, no 1

SDG

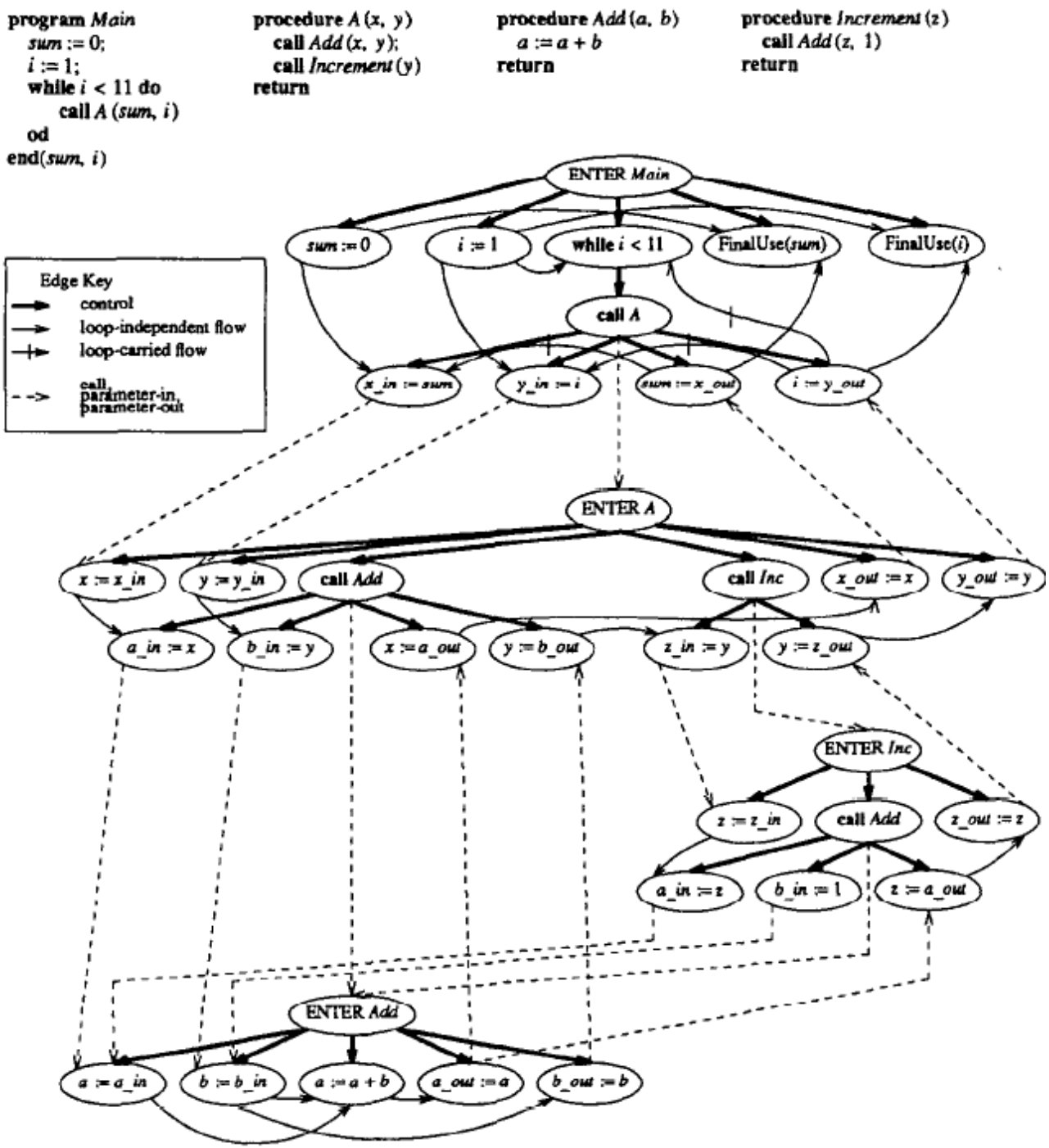
- Formed from PDGs for each procedure and *main*
  - *Intraprocedural*
    - Added actual-in, actual-out vertices for parameters control dependent on the call-site vertex
    - Added formal-in, formal-out vertices control dependent on procedure entry vertex
  - *Interprocedural*
    - Entry vertex of callee is control dependent on call-site vertex
    - Param-in edge between actual-in and formal-in vertices
    - Param-out edge between actual-out and formal-out vertices
    - Summary edges representing transitive interprocedural data dependences

Following static execution paths using the SDG

- Assume start at vertex x
- Find all vertices from which x can be reached *without* descending into procedure calls
- Find all remaining vertices by descending into all previously encountered procedure calls, but *not ascending* up into callers.

Horwitz,
Reps,
Binkley,
TOPLAS
Jan 1990

Step 1:
SDG w.
control dep &
data-dep edges



Horwitz, Reps, Binkley, TOPLAS Jan 1990

Step 2: SDG w. summary edges between params