Class 2

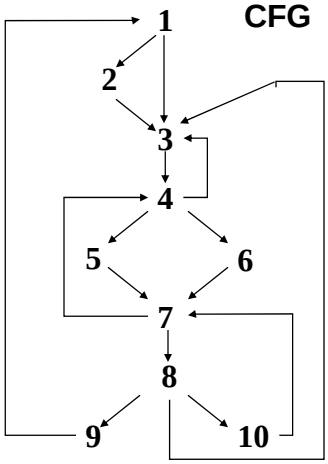
- Review; questions
- Basic Analyses (2)
- Assign (see Schedule for links)
  - Representation and Analysis of Software (Sections 1-5)
  - Additional reading: depth-first presentation, data-flow analysis, etc.
  - Problem Set 1: due 8/25/09

Review, Questions?

- T-Square, syllabus, etc.
- Problem Set 1

- Intermediate representations
- Control-flow analysis
- Search and ordering
- Dominance and postdominance

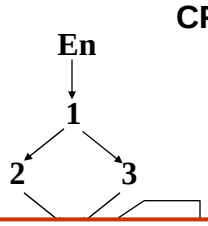
Search and Ordering (depth-first)



For Thursday:

Is there a depth-first presentation with depth greater than 3?

Dominators, Postdominators (dominator algorithm)



Intuition for algorithm

- N is set of nodes in CFG with En, Ex
- initialize *domin*(En) to {En}, *change* to false
- Initialize *domin*(n) to N for all n != En
- iterate over all n (except En) until no change in *domin* sets

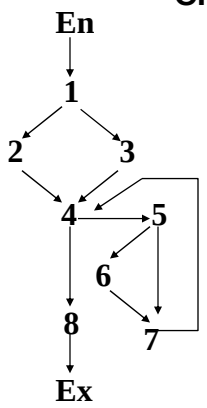
gn N to T
bute *domin*(n) by first taking the
section of T and *domin*(p), forall p, a
ecessor of n
add n to T (this is new *domin*(n))
= *domin*(n), a change has occurred
ssign T to *domin*(n)
hange is true

For Thursday:

Show iterations of the algorithm over the nodes in the CFG until the result converges?

Dominators, Postdominators (dominator algorithm)

CFG



Intuition for algorithm

- N is set of nodes in CFG with En, Ex
- initialize *domin*(En) to {En}, *change* to false
- Initialize *domin*(n) to N for all n != En
- iterate over all n (except En) until no change in *domin* sets
  - assign N to T
  - compute *domin*(n) by first taking the intersection of T and *domin*(p), forall p, a predecessor of n
  - then add n to T (this is new *domin*(n))
  - If T != *domin*(n), a change has occurred
    - assign T to *domin*(n)
    - *change* is true

Dominators, Postdominators

Node	domin	Iteration 1: domin
En	En	En
1	En,1,2,3,4,5,6,7,8,Ex	T={En,1,…,Ex}; T∩{En}→{En}; Add 1→{En,1}
2	En,1,2,3,4,5,6,7,8,Ex	T={En,1,…,Ex}; T∩{En,1}→{En,1}; Add 2→{En,1,2}
3	En,1,2,3,4,5,6,7,8,Ex	T={En,1,…,Ex}; T∩{En,1}→{En,1}; Add 3→{En,1,3}
4	En,1,2,3,4,5,6,7,8,Ex	T={En,1,…,Ex}; T∩{En,1,2}∩{En,1,3} ∩{En,1,…,Ex} →{En,1} Add 4→{En,1,4}
5	En,1,2,3,4,5,6,7,8,Ex	T={En,1,…,Ex}; T∩{En,1,4}→{En,1,4}; Add 5→{En,1,4,5}
6	En,1,2,3,4,5,6,7,8,Ex	T={En,1,…,Ex}; T∩{En,1,4,5}→{En,1,4,5} Add 6→{En,1,4,5,6}
7	En,1,2,3,4,5,6,7,8,Ex	T={En,1,…,Ex}; T∩{En,1,4,5,6}∩{En,1,4,5}→{En,1,4,5} Add 7→{En,1,4,5,7}
8	En,1,2,3,4,5,6,7,8,Ex	T={En,1,…,Ex}; T∩{En,1,4}→{En,1,4} Add 8→{En,1,4,8}
Ex	En,1,2,3,4,5,6,7,8,Ex	T={En,1,…,Ex}; T∩{En,1,4,8}→{En,1,4,8} Add Ex→{En,1,4,8,Ex}

Dominators, Postdominators

Node	Iteration 1: domin	Iteration 2: domin
En	En	En
1	T={En,1,…,Ex}; T∩{En}→{En}; Add 1→{En,1}	{En,1}
2	T={En,1,…,Ex}; T∩{En,1}→{En,1}; Add 2→{En,1,2}	{En,1,2}
3	T={En,1,…,Ex}; T∩{En,1}→{En,1}; Add 3→{En,1,3}	{En,1,3}
4	T={En,1,…,Ex}; T∩{En,1,2}∩{En,1,3} ∩{En,1,…,Ex} →{En,1} Add 4→{En,1,4}	T={En,1,4}; T∩{En,1,2}∩{En,1,3} ∩{En,1,4,5,7} →{En,1} Add 4→{En,1,4}
5	T={En,1,…,Ex}; T∩{En,1,4}→{En,1,4}; Add 5→{En,1,4,5}	{En,1,4,5}
6	T={En,1,…,Ex}; T∩{En,1,4,5}→{En,1,4,5} Add 6→{En,1,4,5,6}	{En,1,4,5,6}
7	T={En,1,…,Ex}; T∩{En,1,4,5,6}∩{En,1,4,5}→{En,1,4,5} Add 7→{En,1,4,5,7}	{En,1,4,5,7}
8	T={En,1,…,Ex}; T∩{En,1,4}→{En,1,4} Add 8→{En,1,4,8}	{En,1,4,8}
Ex	T={En,1,…,Ex}; T∩{En,1,4,8}→{En,1,4,8} Add Ex→{En,1,4,8,Ex}	{En,1,4,8,Ex}

Dominators, Postdominators (dominator algorithm)

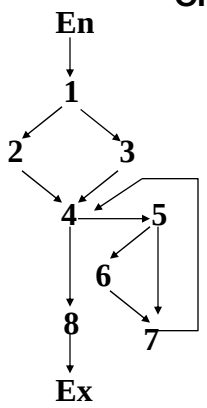
Node	domin	Iteration 1: domin
Ex	En,1,2,3,4,5,6,7,8,Ex	T={En,1,…,Ex}; T∩N→N Add Ex→N
8	En,1,2,3,4,5,6,7,8,Ex	T={En,1,…,Ex}; T∩N(for 4)→N; Add 8→N
7	En,1,2,3,4,5,6,7,8,Ex	T={En,1,…,Ex}; T∩N(for 5)∩N(for 6) →N; Add 7→N
6	En,1,2,3,4,5,6,7,8,Ex	T={En,1,…,Ex}; T∩N(for 5)→N Add 6→N
5	En,1,2,3,4,5,6,7,8,Ex	T={En,1,…,Ex}; T∩N(for 4) →N Add 5→N
4	En,1,2,3,4,5,6,7,8,Ex	T={En,1,…,Ex}; T∩N(for 2, 3, 7)→N Add 4→N
3	En,1,2,3,4,5,6,7,8,Ex	T={En,1,…,Ex}; T∩N(for 1)→N; Add 3→N
2	En,1,2,3,4,5,6,7,8,Ex	T={En,1,…,Ex}; T∩N(for 1)→N; Add 2→N
1	En,1,2,3,4,5,6,7,8,Ex	T={En,1,…,Ex}; T∩{En}→{En} Add 1→{En,1}
En	En	{En}

Dominators, Postdominators (dominator algorithm)

Node	Iteration 1: domin	Iteration 2: domin
Ex	T={En,1,…,Ex}; T∩N→N Add Ex→N	N
8	T={En,1,…,Ex}; T∩N(for 4)→N; Add 8→N	N
7	T={En,1,…,Ex}; T∩N(for 5)∩N(for 6) →N; Add 7→N	N
6	T={En,1,…,Ex}; T∩N(for 5)→N Add 6→N	N
5	T={En,1,…,Ex}; T∩N(for 4) →N Add 5→N	N
4	T={En,1,…,Ex}; T∩N(for 2, 3, 7)→N Add 4→N	N
3	T={En,1,…,Ex}; T∩N(for 1)→N; Add 3→N	T={En,1,…,Ex}; T∩{En,1}(for 1)→{En,1}; Add 3→{En,1,3}
2	T={En,1,…,Ex}; T∩N(for 1)→N; Add 2→N	T={En,1,…,Ex}; T∩{En,1}(for 1)→{En,1}; Add 2→{En,1,2}
1	T={En,1,…,Ex}; T∩{En}→{En} Add 1→{En,1}	T={En,1,…,Ex}; T∩{En}→{En} Add 1→{En,1}
En	{En}	{En}

Node Ordering for Efficiency

CFG

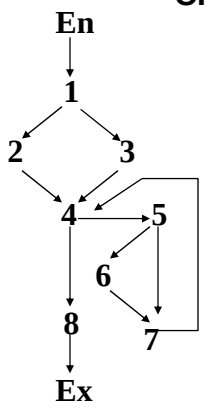


Visit in DF Order → 2 iterations

Visit in another order (e.g., reverse DF Order) → more, possibly many, iterations

Node Ordering for Efficiency

CFG



Visit in DF Order → 2 iterations

Visit in another order (e.g., reverse DF Order) → more, possibly many, iterations

Does requiring an ordering of nodes incur any additional overhead?

Loops and Reducibility

Finding Loops

- Loops are important—why?
- How do we identify loops?
- Not every cycle in a graph is a loop.
- There are different kinds of loops we need to consider
  - Irreducible loops
  - Reducible loops
- We identify “natural loops,” which account for most loops in real programs

Loops

We’ll consider what are known as natural loops

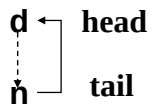
- Single entry node (header) that dominates all other nodes in the loop

Why is this important?

Loops

We'll consider what are known as natural loops

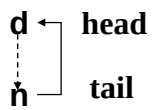
- Single entry node (header) that dominates all other nodes in the loop
- Nodes in loop form a strongly connected component (SCC): from every node there is at least one path back to the header
- There is a way to iterate: there is a back edge (n,d)
  whose target node d (called the head)
  dominates its source node n (called the tail)



Loops

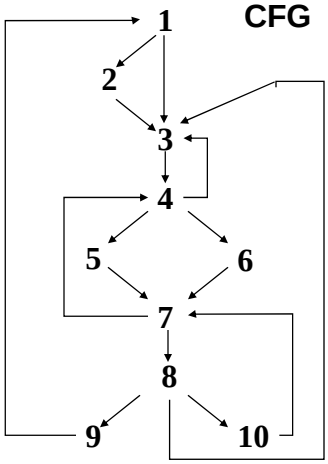
We'll consider what are known as natural loops

- Single entry node (header) that dominates all other nodes in the loop
- Nodes in loop form a strongly connected component (SCC): from every node there is at least one path back to the header
- There is a way to iterate: there is a back edge (n,d)
  whose target node d (called the head)
  dominates its source node n (called the tail)



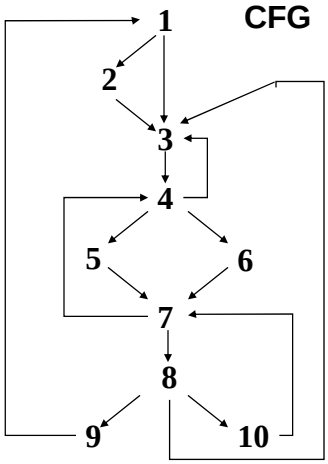
- If two back edges have the same target, then all nodes in the loop sets for these edges are in the same loop

Loops (example)



Which edges are back edges?

Loops (example)



Which edges are back edges?

4 → 3
7 → 4
10 → 7
9 → 1
8 → 3

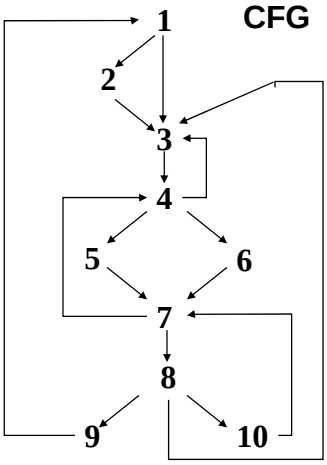
Loops

Construction of loops

1. Find dominators in CFG
2. Find back edges
3. Traverse back edge in reverse execution direction until the target of the back edge (i.e., head) is reached; all nodes encountered during this traversal form the loop.

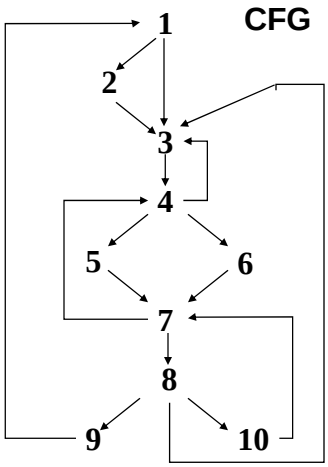
   Result is all nodes that can reach the source of the edge without going through the target

Loops (example)



Back Edge	Loop Induced
4 → 3	
7 → 4	
10 → 7	
8 → 3	
9 → 1	

Loops (example)



Back Edge	Loop Induced
4 → 3	{3,4,5,6,7,8,10}
7 → 4	{4,5,6,7,8,10}
10 → 7	{7,8,10}
8 → 3	{3,4,5,6,7,8,10}
9 → 1	{1,2,…,10}

Loops (algorithm)

Input: CFG and back edge *n* → *d*
Output: set of nodes in natural loop *n* → *d*
Method: start with *n*; consider nodes *m* != *d* that are in loop; each node in loop except *d* is pushed onto stack once so predecessors are examined

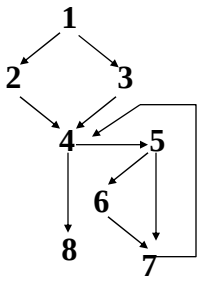
```
stack = empty
loop = {d}
insert (n}
while stack is not empty do
      pop m
      foreach predecessor p of m do
          insert (p)
```

```
procedure insert (m)
    if m is not in loop then
          loop = loop union {m}
          push m onto stack;
```

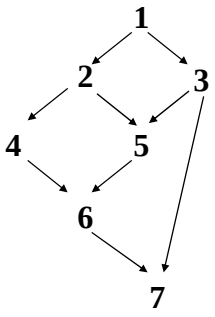
Reducibility

- First T1-T2 analysis: apply the following two transformations to the CFG:
  - T1: if n is a node with a self loop (i.e., an edge n→n), delete that edge
  - T2: if there is a node n, not the initial node, that has a unique predecessor, m, then m may consume n by deleting n and making all successors of n (including m, possibly) be successors of m
- Properties of T1-T2 transformations:
  - If T1-T2 transformations applied in any order until no more transformations are possible, a unique flow graph results
  - The CFG resulting from T1-T2 application is the *limit flow graph*

Graphs for examples



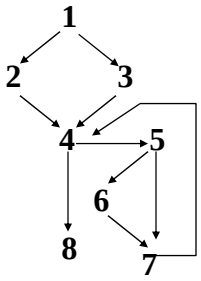
Graph 1



Graph 2

Reducibility

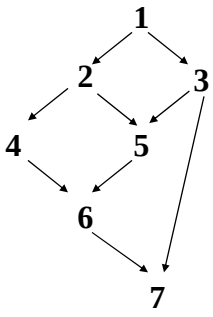
Apply T1-T2 transformations to this CFG



Graph 1

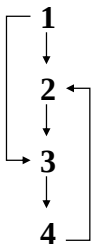
Reducibility

Apply T1-T2 transformations to this CFG

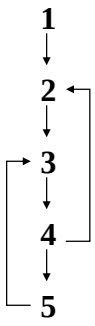


Graph 2

Graphs for examples



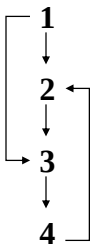
Graph 3



Graph 4

Reducibility

Apply T1-T2 transformations to this CFG



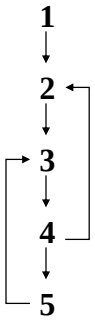
Graph 3

Reducibility

- A flow graph is reducible iff
  - its edges can be partitioned into two groups
    - Forward edges forming an acyclic graph in which every node can be reached from the initial node and
    - Back edges in which the head dominates the tail (i.e., every retreating edge is a back edge)
  - T1-T2 transformations applied to the graph result in a single node

Reducibility (example)

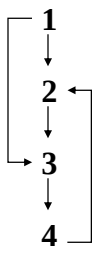
Is this graph reducible?



Graph 4

Reducibility (example)

Is this graph reducible?



Graph 3

Data-flow Analysis

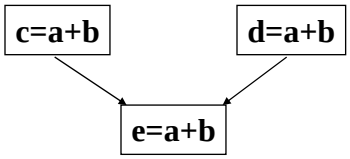
Data-flow Analysis

1. Introduction (motivation, overview)
2. Data-flow problems (reaching definitions, etc.)
3. Iterative data-flow analysis
4. Other types of data-flow analysis: worklist, interval
5. DU-chains, UD-Chains, Webs
6. Data-dependence graph

Introduction (uses of data-flow)

Compiler Optimization

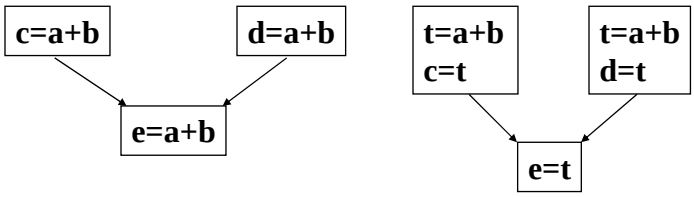
- *common subexpression elimination*



Introduction (uses of data-flow)

Compiler Optimization

- *common subexpression elimination*



- need to know available expressions: which expressions have been computed at the point before this statement

Introduction (uses of data-flow)

Compiler Optimization

- *constant propagation*
  - suppose every assignment to **c** that reaches this statement assigns 5

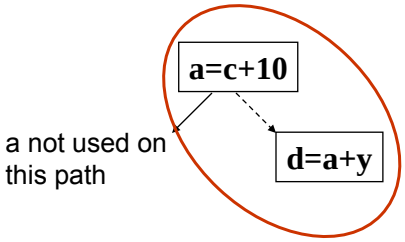
a=c+10

  - then **a** can be replaced by 15
  - need to know reaching definitions: which definitions of variable c reach this statement

Introduction (uses of data-flow)

Software Engineering Tasks

- *data-flow testing*
  - suppose that a statement assigns a value but the use of that value is never executed under test

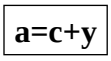


  - need definition-use pairs (du-pairs): associations between definitions and uses of the same variable or memory location

Introduction (uses of data-flow)

Software Engineering Tasks

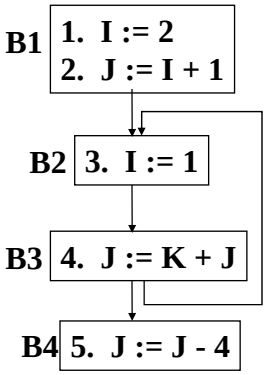
- *Debugging*
  - suppose that **a** has the incorrect value in the statement



  - need data dependence information: statements that can affect the incorrect value at this point

Introduction (uses of data-flow)

Software Engineering Tasks



- *Uninitialized variables*

  A variable is uninitialized if there is a path from entry on which the variable is not defined

Introduction (overview)

Data-flow analysis provides information for these and other tasks by computing the flow of different types of data to points in the program

- For structured programs, data-flow analysis can be performed on an AST; in general, intraprocedural (global) data-flow analysis performed on the CFG
- Exact solutions to most problems are undecidable—e.g.,
  - May depend on input
  - May depend on outcome of a conditional statement
  - May depend on termination of loop

  Thus, we compute approximations to the exact solution

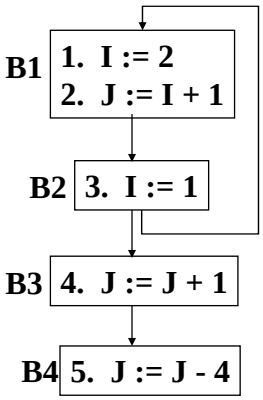
Introduction (overview)

- Approximate analysis can overestimate the solution:
  - Solution contains actual information plus some spurious information but does not omit any actual information
  - This type of information is safe or conservative
- Approximate analysis can underestimate the solution:
  - Solution may not contains all information in the actual solution
  - This type of information in unsafe

Introduction (overview)

- Approximate analysis can overestimate the solution:
  - Solution contains actual information plus some spurious information but does not omit any actual information
  - This type of information is safe or conservative
- Approximate analysis can underestimate the solution:
  - Solution may not contains all information in the actual solution
  - This type of information in unsafe
- For optimization, need safe, conservative analysis
- For software engineering tasks, may be able to use unsafe analysis information
- Biggest challenge for data-flow analysis: provide safe but precise (i.e., minimize the spurious information) information in an efficient way

Introduction (overview)

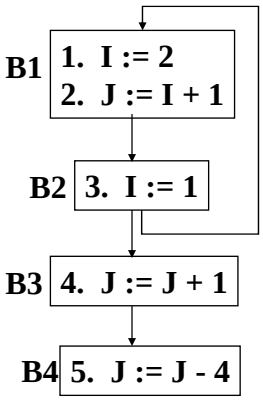


Compute the flow of data to points in the program—e.g.,

- Where does the assignment to I in statement 1 reach?
- Where does the expression computed in statement 2 reach?
- Which uses of variable J are reachable from the end of B1?
- Is the value of variable I live after statement 3?

Interesting points before and after basic blocks or statements

Data-flow Problems (reaching definitions)

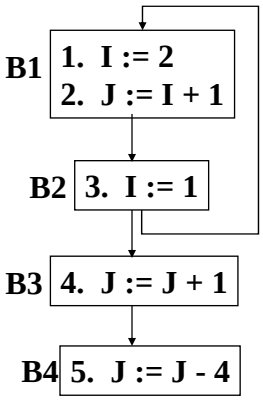


A definition of a variable or memory location is a point or statement where that variable gets a value—e.g., input statement, assignment statement.

A definition of A reaches a point p if there exists a control-flow path in the CFG from the definition to p with no other definitions of A on the path (called a definition-clear path)

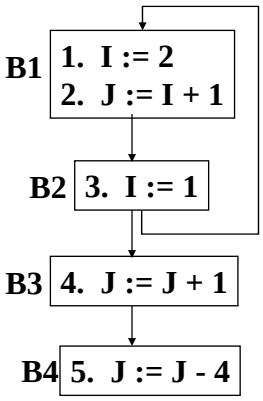
Such a path may exist in the graph but may not be executable (i.e., there may be no input to the program that will cause it to be executed); such a path is infeasible.

Data-flow Problems (reaching definitions)

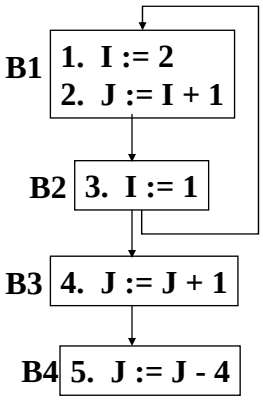


- Where are the definitions in the program?
  - Of variable I:
  - Of variable J:
- Which basic blocks (before block) do these definitions reach?
  - Def 1 reaches
  - Def 2 reaches
  - Def 3 reaches
  - Def 4 reaches
  - Def 5 reaches

Graph for examples

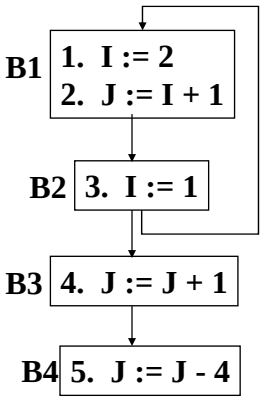


Data-flow Problems (reaching definitions)



- Where are the definitions in the program?
  - Of variable I: 1, 3
  - Of variable J: 2, 4, 5
- Which basic blocks (before block) do these definitions reach?
  - Def 1 reaches B2
  - Def 2 reaches B1, B2, B3
  - Def 3 reaches B1, B3, B4
  - Def 4 reaches B4
  - Def 5 reaches exit

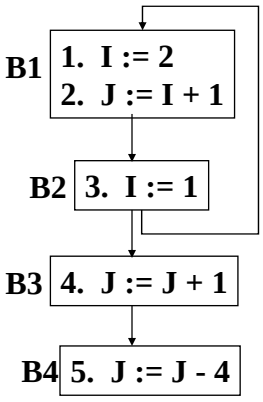
Iterative Data-flow Analysis (reaching definitions)



Method:

1. Compute two kinds of local information (i.e., within a basic block)
   - GEN[B] is the set of definitions that are created (generated) within B
   - KILL[B] is the set of definitions that, if they reach the point before B (i.e., the beginning of B) won't reach the end of B or
     PRSV[B] is the set of definitions that are preserved (not killed) by B
2. Compute two other sets by propagation
   - IN[B] is the set of definitions that reach the beginning of B; also RCHin[B]
   - OUT[B] is the set of definitions that reach the end of B; also RCHout[B]

Iterative Data-flow Analysis (reaching definitions)



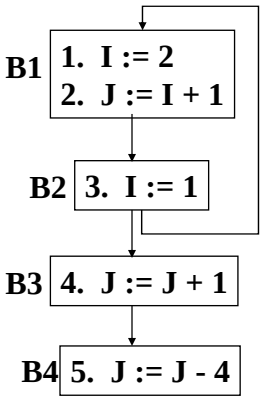
Method (cont'd):

3. Propagation method:
   - Initialize the IN[B], OUT[B] sets for all B
   - Iterate over all B until there are no changes to the IN[B], OUT[B] sets
   - On each iteration, visit all B, and compute IN[B], OUT[B] as
   - IN[B] = U OUT[P], P is a predecessor of B
   - OUT[B] = GEN[B] U (IN[B] ∩ PRSV[B]) or
   - OUT[B] = GEN[B] U (IN[B] – Kill[B])

Iterative Data-flow Analysis (reaching

	Init GEN	Init KILL	Init IN	Init OUT	Iter1 IN	Iter1 OUT	Iter2 IN	Iter2 OUT
1								
2								
3								
4								

Iterative Data-flow Analysis (reaching definitions)



Data-flow for example (set approach)

	Init GEN	Init KILL	Init IN	Init OUT	Iter1 IN	Iter1 OUT	Iter2 IN	Iter2 OUT
1	1,2	3,4,5	--	1,2	3	1,2	2,3	1,2
2	3	1	--	3	1,2	2,3	3	2,3
3	4	2,5	--	4	2,3	3,4	2,3	3,4
4	5	2,4	--	5	3,4	3,5	5	3,5

All entries are sets; sets in red indicate changes from last iteration thus, requiring another iteration of the algorithm