Class 3

- Review; questions
- Basic Analyses (3)
- Assign (see Schedule for links)
  - Representation and Analysis of Software (Sections 1-5)
  - Additional readings:
    - Data-flow analysis
    - Control/program-dependence analysis
  - Problem Set 2: due 9/1/09

Review, Questions?

- Data-flow analysis

Data-flow Analysis

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

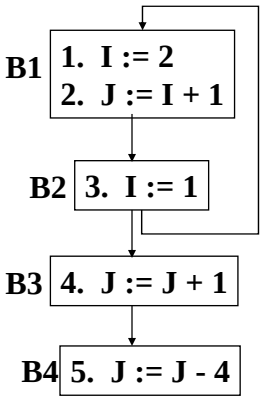
Introduction (overview)

- 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)

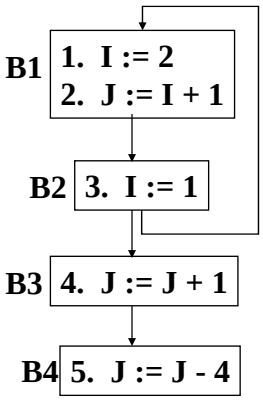


Compute the flow of data to points in the program—e.g.,

- 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)

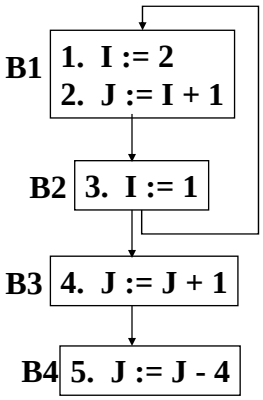


A definition of a variable or memory location is a point or statement where that variable gets a value—e.g., input statement, assignment statement.

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)

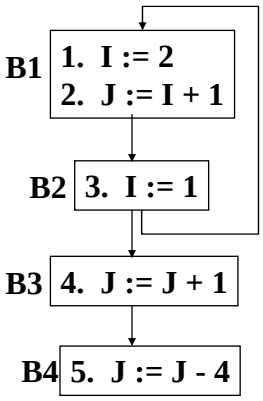
Such a path may exist in the graph but may not be executable (i.e., there may be no input to the program that will cause it to be executed); such a path is infeasible.

Data-flow Problems (reaching definitions)

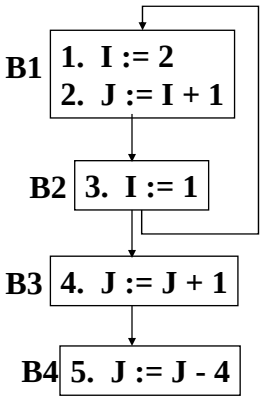


- Where are the definitions in the program?
  - Of variable I:
  - Of variable J:
- Which basic blocks (before block) do these definitions reach?
  - Def 1 reaches
  - Def 2 reaches
  - Def 3 reaches
  - Def 4 reaches
  - Def 5 reaches

Graph for examples

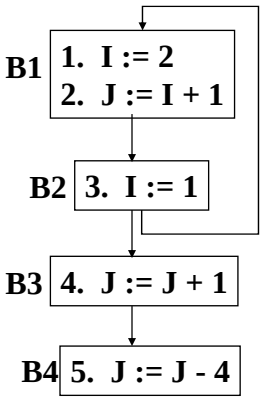


Data-flow Problems (reaching definitions)



- 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

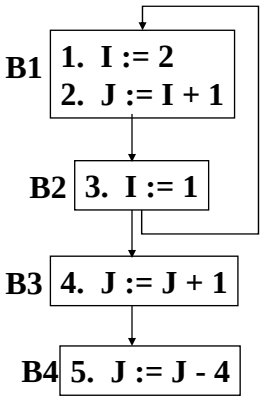
Iterative Data-flow Analysis (reaching definitions)



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]

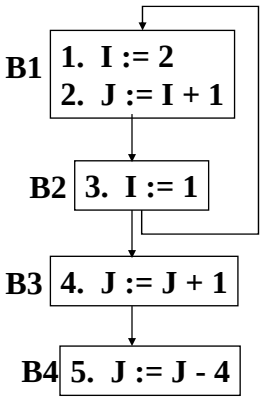
Iterative Data-flow Analysis (reaching definitions)



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])

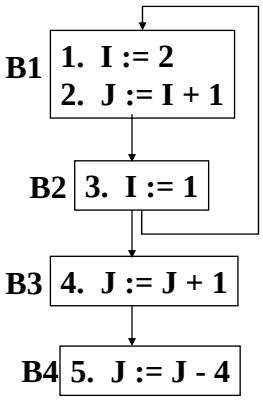
Iterative Data-flow Analysis (reaching definitions)



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)

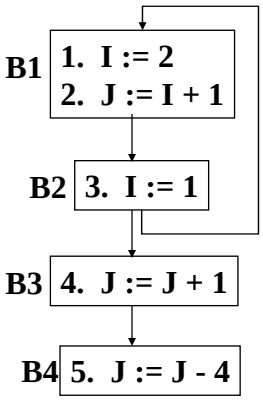


Data-flow for example (set approach)

	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	1,2	2,3
3	4	2,5	--	4	2,3	3,4	2,3	3,4
4	5	2,4	--	5	3,4	3,5	3,4	3,5

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

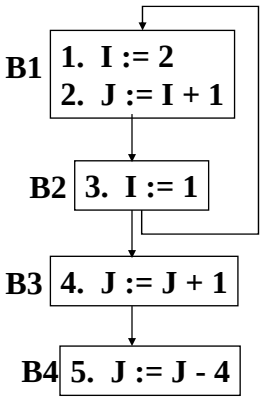
Iterative Data-flow Analysis (reaching definitions)



Data-flow for example (bit-vector approach)

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

Iterative Data-flow Analysis (reaching definitions)



Data-flow for example (bit-vector approach)

	Init GEN	Init KILL	Init IN	Init OUT	Iter1 IN	Iter1 OUT
1	11000	00111	00000	11000	00100	11000
2	00100	10000	00000	00100	11000	01100
3	00010	01001	00000	00010	01100	00110
4	00001	01010	00000	00001	00110	00101

Iterative Data-flow Analysis (reaching definitions)

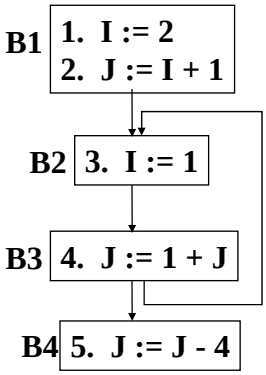
```
algorithm ReachingDefinitions
Input:  CFG w/GEN[B], KILL[B] for all B
Output:  IN[B], OUT[B] for all B
Method:  Described on slides 17, 18
begin ReachingDefinitions
      IN[B]=empty; OUT[B]=GEN[B], for all B;  change = true
      while change do begin
            Change = false
            foreach B do begin
                  In[B] = union OUT[P], P is a predecessor of B
                  Oldout = OUT[B]
                  OUT[B] = GEN[B] union (IN[B] – Kill[B])
                  if OUT[B] != Oldout then change = true
            endfor
      endwhile
end Reaching Definitions
```


Iterative Data-flow Analysis (reaching definitions)

Questions about algorithm:

1. Is the algorithm guaranteed to converge? Why or why not?
2. What is the worst-case time complexity of the algorithm?
3. What is the worst-case space complexity of the algorithm?
4. How does depth-first ordering improve the worst-case time complexity?

Iterative Data-flow Analysis (reachable uses)

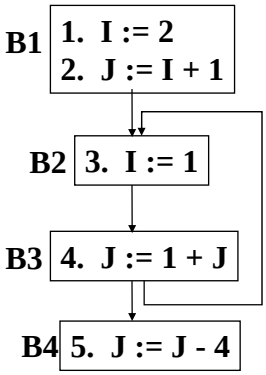


A use of a variable or memory location is a point or statement where that variable is referenced by not changed --- e.g., used in a computation, used in a conditional, output

A use of A is reachable from a point p if there exists a control-flow path in the CFG from the p to the use with no definitions of A on the path

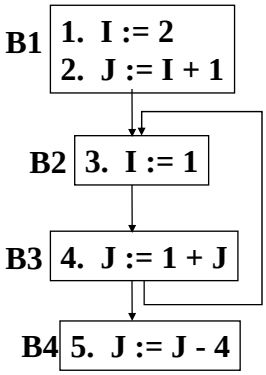
Reachable uses also called upwards exposed uses

Iterative Data-flow Analysis (reachable uses)



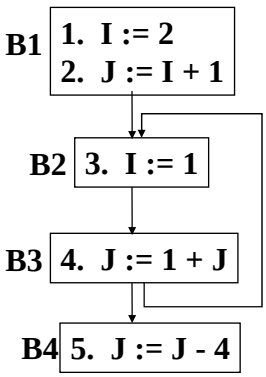
- Where are the uses in the program?
  - Of variable I:
  - Of variable J:
- From which basic blocks (end of block) are these uses reachable?

Iterative Data-flow Analysis (reachable uses)



- Where are the uses in the program?
  - Of variable I: 2.1
  - Of variable J: 4.2, 5.1
- From which basic blocks (end of block) are these uses reachable?
  - Use 4.2 is reachable from B1, B2, B3
  - Use 5.1 is reachable from B3

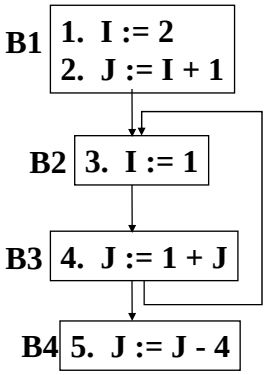
Iterative Data-flow Analysis (reachable uses)



Method:

1. Compute two kinds of local information (i.e., within a basic block)
   - GEN[B] is the set of uses that are created (generated) within B and can be reached from the beginning of B (called upwards exposed uses); sometimes called USE[B]
   - KILL[B] is the set of uses that, if they can be reached from the end of B, they cannot be reached from the beginning of B; sometimes called DEF[B]
2. Compute two other sets by propagation
   - IN[B] is the set of uses that can be reached from the end of B
   - OUT[B] is the set of uses that can be reached from the beginning of B

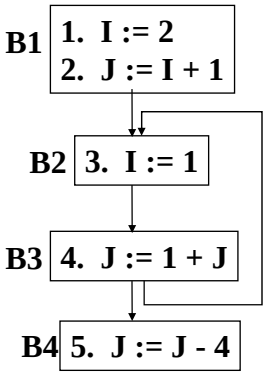
Iterative Data-flow Analysis (reachable uses)



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

Iterative Data-flow Analysis (reachable uses)



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] = union OUT[S], S is a successor of B

     OUT[B] = GEN[B] union (IN[B] – Kill[B])

Iterative Data-flow Analysis (reachable uses)

Questions about algorithm:

1. Is the algorithm guaranteed to converge? Why or why not?
2. What is the worst-case time complexity of the algorithm?
3. What is the worst-case space complexity of the algorithm?
4. How does depth-first ordering improve the w-c time complexity?

Iterative Data-flow Analysis (reachable uses)

Similarities between RD and RU

Differences between RD and RU

Iterative Data-flow Analysis (reachable uses)

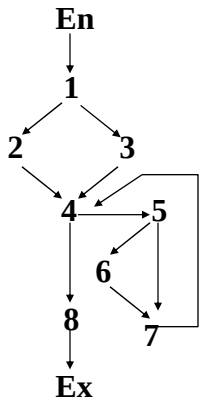
Similarities between RD and RU

- Local information (GEN and KILL) computed for each B
- IN and OUT sets defined: IN at point where data flows into B from outside B; OUT at point where data flow out of B
- Flow into block computed as union of predecessors in flow
- Iteration until no more changes

Differences between RD and RU

- RD flow is forward; RU flow is backward
- RD best ordering is depth-first (topological); RU best ordering is reverse depth-first (reverse topological)

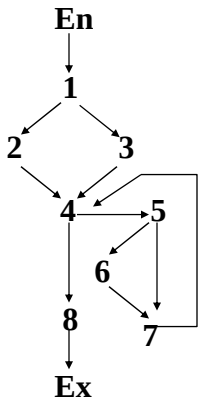
Iterative Data-flow Analysis (dominators)



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

Iterative Data-flow Analysis (dominators)

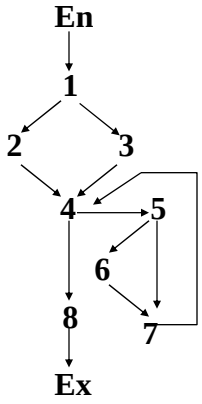


Dom as iterative data-flow:

1. Compute two kinds of local information (i.e., within a basic block)
   - GEN[B]
   - KILL[B]
2. Compute two other sets by propagation
   - IN[B]
   - OUT[B]
   - Initialize
   - Iterate over all B

   On each iteration, visit all B, and compute

Iterative Data-flow Analysis (dominators)



Dom as iterative data-flow:

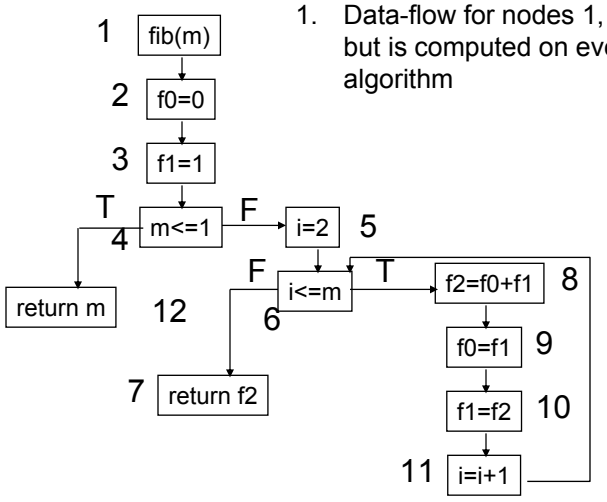
1. Compute two kinds of local information (i.e., within a basic block)
   - GEN[B] is the node itself
   - KILL[B] is empty
2. Compute two other sets by propagation
   - IN[B] is the set of dominators of nodes that are predecessors of B
   - OUT[B] is the set of dominators of B
   - Initialize the IN[B], OUT[B] sets for all B
   - Iterate over all B until there are no changes in IN[B] or OUT[B]
   - On each iteration, visit all B, and compute IN[B] as intersection of OUT[P], P a predecessor of B; compute OUT[B] as union of IN[B] and GEN[B] (because KILL[B] is empty)

Iterative Data-flow Analysis (generalization)

Data-flow Framework

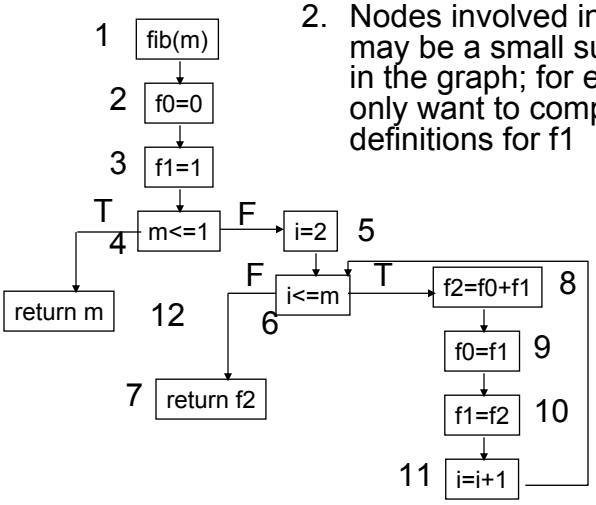
<answered in class>

Other Types of Data-flow Analysis (worklist)



1. Data-flow for nodes 1, 2, 3 never changes but is computed on every iteration of the algorithm

Other Types of Data-flow Analysis (worklist)



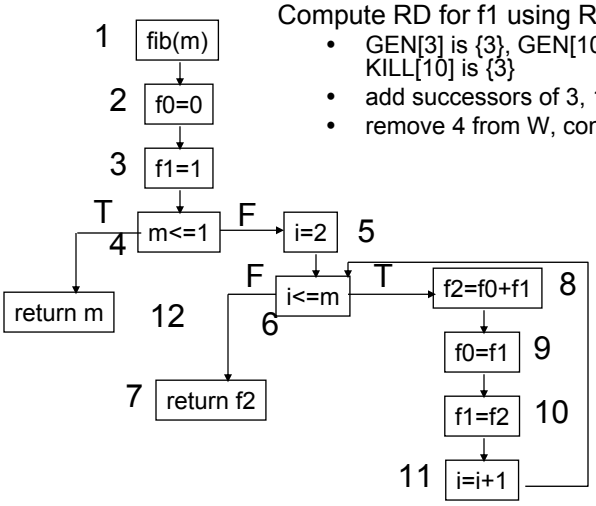
2. Nodes involved in the computation may be a small subset of the nodes in the graph; for example, what if only want to compute reaching definitions for f1

Other Types of Data-flow Analysis (worklist)

algorithm RDWorklist
Input: GEN[B], KILL[B] for all B
output reaching definitions for each B
Method:

initialize IN[B], OUT[B] for all B; add successors of B involved initially involved in computation to worklist W
repeat
remove B from W
Oldout=OUT[B]
compute IN[B], OUT[B]
if oldout != OUT[B] then add successors of B to W endif
until W is empty

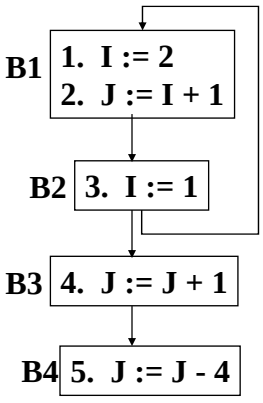
Other Types of Data-flow Analysis (worklist)



Compute RD for f1 using RDWorklist

- GEN[3] is {3}, GEN[10] is {10}, KILL[3] is {10}, KILL[10] is {3}
- add successors of 3, 10 to W
- remove 4 from W, compute IN[4], OUT[4], etc

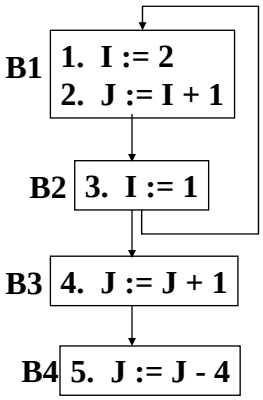
Other Types of Data-flow Analysis (incremental)



	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	1,2	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

- What if stmt 4 changes to J := I + 1?
- What if stmt 4 changes to I := J + 1;
- What if stmt 4 changes to J := J –1?
- Etc

Other Types of Data-flow Analysis (demand)

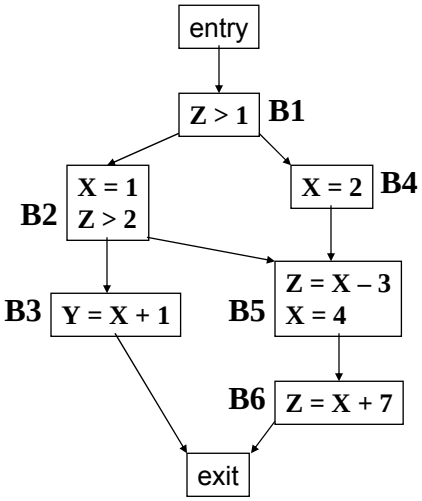


What if want data flow for one statement only—e.g., find reaching definitions for B3?

DU-Chains, UD-Chains, Webs

- A definition-use chain or DU-chain for a definition D of variable v connects the D to all uses of v that it can reach
- A use-definition chain or UD-chain for a use U of variable v connects U to all definitions of v that reach it

DU-Chains, UD-Chains, Webs



DU-chain(X,2)

DU-chain(X,4)

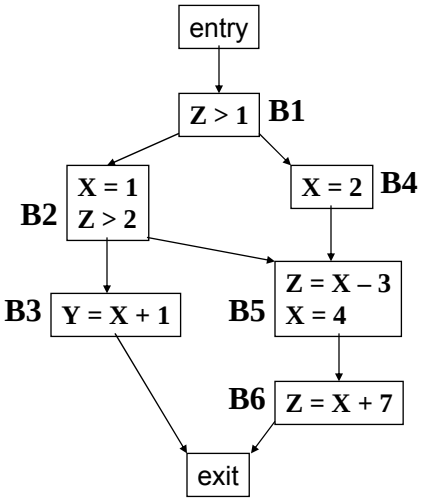
DU-chain(X,5)

DU-chain(Y,3)

DU-chain(Z,5)

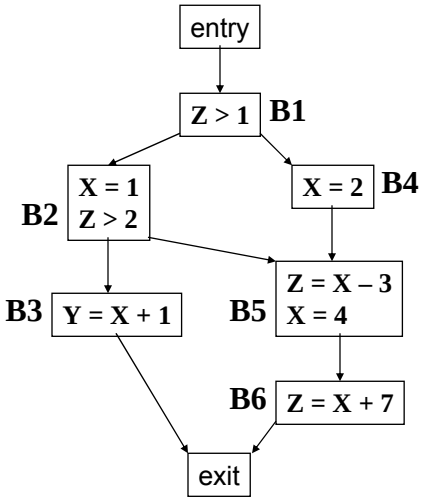
DU-chain(Z,6)

DU-Chains, UD-Chains, Webs



DU-chain(X,2)
{(X,3), (X,5)}
DU-chain(X,4)
{(X,5)}
DU-chain(X,5)
{(X,6)}
DU-chain(Y,3)
{}
DU-chain(Z,5)
{}
DU-chain(Z,6)
{}

DU-Chains, UD-Chains, Webs



UD-chain(Z,1)

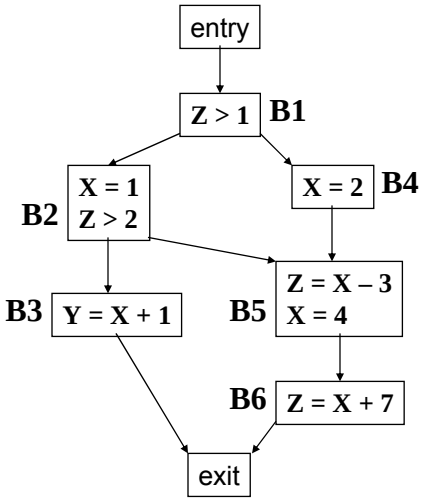
UD-chain(Z,2)

UD-chain(X,3)

UD-chain(X,5)

UD-chain(X,6)

DU-Chains, UD-Chains, Webs



UD-chain(Z,1)
{}
UD-chain(Z,2)
{}
UD-chain(X,3)
{(X,2)}
UD-chain(X,5)
{(X,2),(X,4)}
UD-chain(X,6)
{(X,5)}

DU-Chains, UD-Chains, Webs

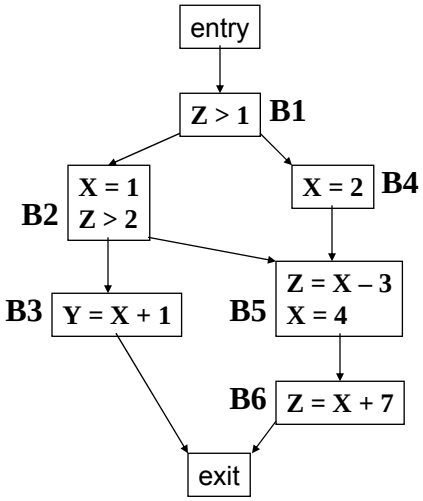
- How can we compute DU-chains?

- How can we compute UD-chains?

DU-Chains, UD-Chains, Webs

- A web for a variable is the maximal union of intersecting du-chains

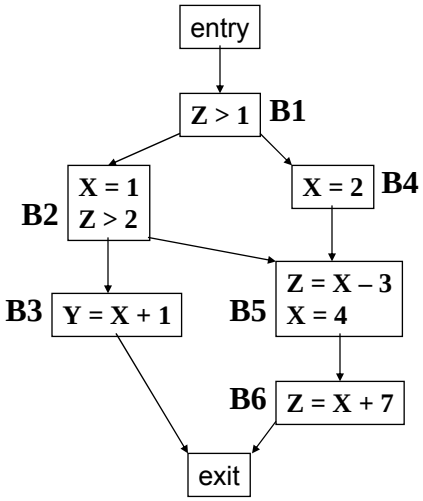
DU-Chains, UD-Chains, Webs



DU-chains

1. DU-chain(X,2) = {(X,3), (X,5)}
2. DU-chain(X,4) = {(X,5)}
3. DU-chain(X,5) = {(X,6)}
4. DU-chain(Y,3) = {}
5. DU-chain(Z,5) = {}
6. DU-chain(Z,6) = {}

DU-Chains, UD-Chains, Webs



DU-chains

1. DU-chain(X,2) = {(X,3), (X,5)}
2. DU-chain(X,4) = {(X,5)}
3. DU-chain(X,5) = {(X,6)}
4. DU-chain(Y,3) = {}
5. DU-chain(Z,5) = {}
6. DU-chain(Z,6) = {}

Intersecting: 1 and 2 → web consisting of defs 2 and 4, uses 3 and 5

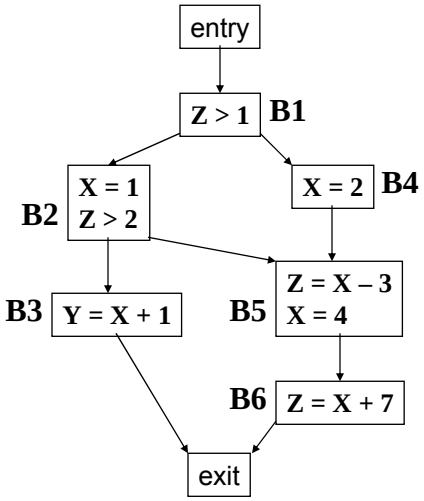
Intersecting: 3 → web consisting of def 5, use 6

Data-dependence Graph

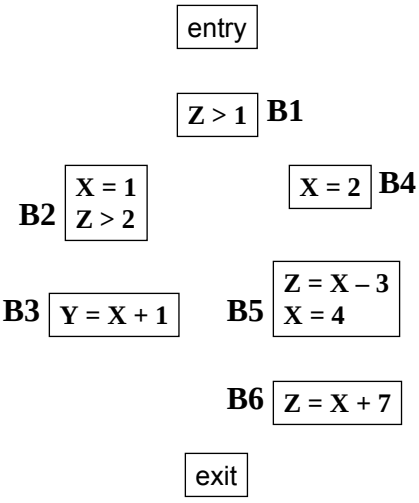
A data-dependence graph has one node for every variable (basic block) and one edge representing the flow of data between the two nodes

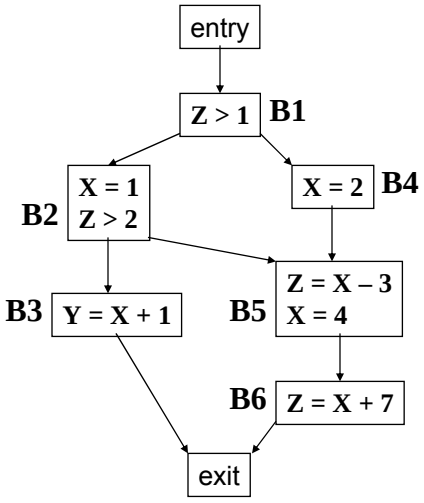
Different types of data dependence

- Flow: def to use
- Anti: use to def
- Out: def to def

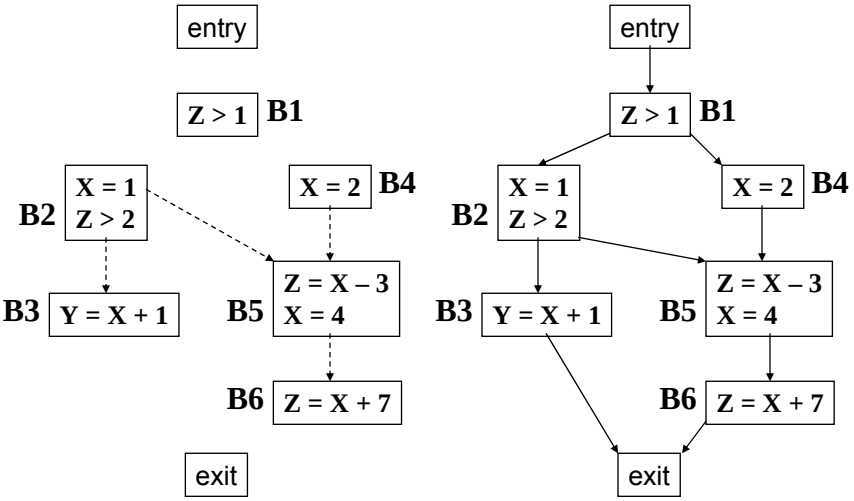


Data-dependence Graph





Data-dependence Graph



Data-flow Wrap-up (for now)

- Why is straight propagation inefficient?
- What are ways to improve it?