CS 4240: Compilers

Lecture 3: Introduction to Data Flow Analysis

Instructor: Vivek Sarkar (vsarkar@gatech.edu)

January 14, 2019

Course Announcements

- Ensure that you can access the course Piazza site
  - http://piazza.com/gatech/spring2019/cs4240a
- There will be 3 homeworks and 3 projects during the semester
  - See release and due dates ton Piazza
- Forming project teams
  - We will create a 0-point (pseudo) assignment in Canvas for you to report your team members and implementation language. Deadline: Wednesday, Jan 16, 2019
  - You can use "Search for Teammates!" in Piazza if needed.

Worksheet-2 Solution

(From Lecture 2 given on 01/09/2019)

Q1

- Construct a Control Flow Graph for the IR (Intermediate Representation) segment shown below, and draw it on the right of the IR. Each vertex can be a single IR instruction, or a basic block containing of a straight-line sequence of multiple IR instructions.

```
  assign s, 0
  assign a, 4
  assign i, 0
// branch if (arg1 != arg2)
  brneq k, 0, label0
  assign, b, 1
  goto, label1
label0:
  assign, b, 2
label1:
label2:
// branch if (arg1 >= arg2)
  brgeq, i, n, label3
  mult, _t1 a, b
  add, _t2, s, _t1
  assign s, _t2
  add _t3, i, 1
  assign i, _t3
  goto, label2
label3:
  call, printi, s
  call, printi, a
```

Two answers possible for Q1

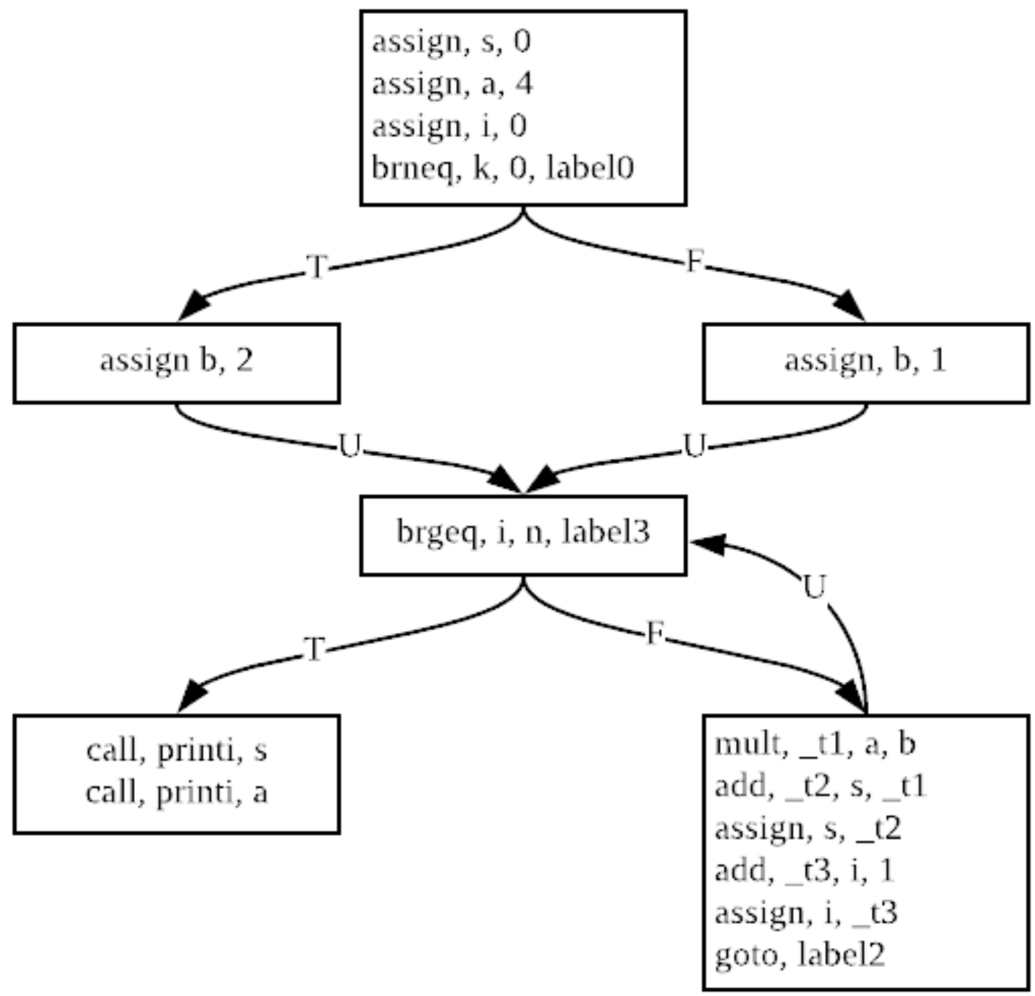
- It is a design choice whether **label**s should be considered as “no-op IR instructions” in CFG vertices, or should be excluded from CFG vertices
  - Just like basic block granularity (minimal vs. maximal) is a design choice when implementing CFGs

- Resulting Control Flow Graphs can be slightly different, depending on the assumption made for labels.

- Most students answered correctly in either case.

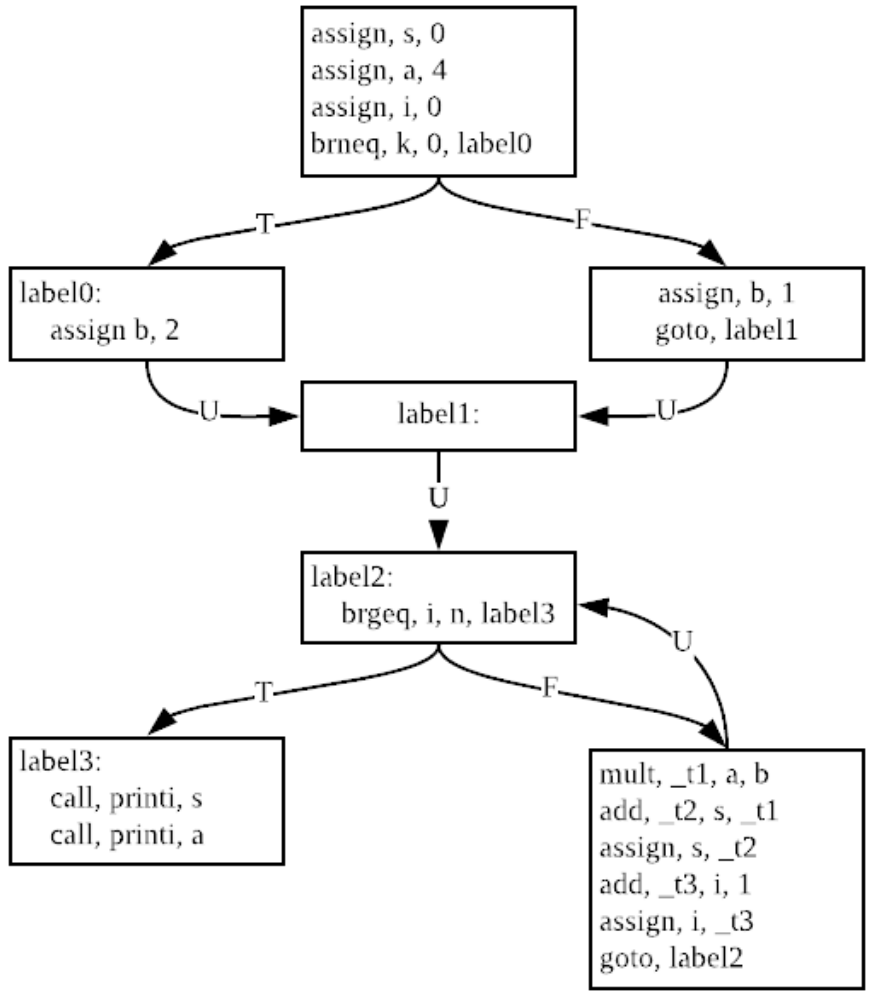
Q1 Sample Solution1: labels are not instructions

```
  assign s, 0
  assign a, 4
  assign i, 0
// branch if (arg1 != arg2)
  brneq k, 0, label0
  assign, b, 1
  goto, label1
label0:
  assign, b, 2
label1:
label2:
// branch if (arg1 >= arg2)
  brgeq, i, n, label3
  mult, _t1 a, b
  add, _t2, s, _t1
  assign s, _t2
  add _t3, i, 1
  assign i, _t3
  goto, label2
label3:
  call, printi, s
  call, printi, a
```



Q1 Sample Solution2: labels are no-op instructions

```
 1    assign s, 0
 2    assign a, 4
 3    assign i, 0
 // branch if (arg1 != arg2)
 4    brneq k, 0, label0
 5    assign, b, 1
 6    goto, label1
 7  label0:
 8    assign, b, 2
 9  label1:
10  label2:
// branch if (arg1 >= arg2)
11    brgeq, i, n, label3
12    mult, _t1 a, b
13    add, _t2, s, _t1
14    assign s, _t2
15    add _t3, i, 1
16    assign i, _t3
17    goto, label2
18  label3:
19    call, printi, s
20    call, printi, a
```



Q2, Q3

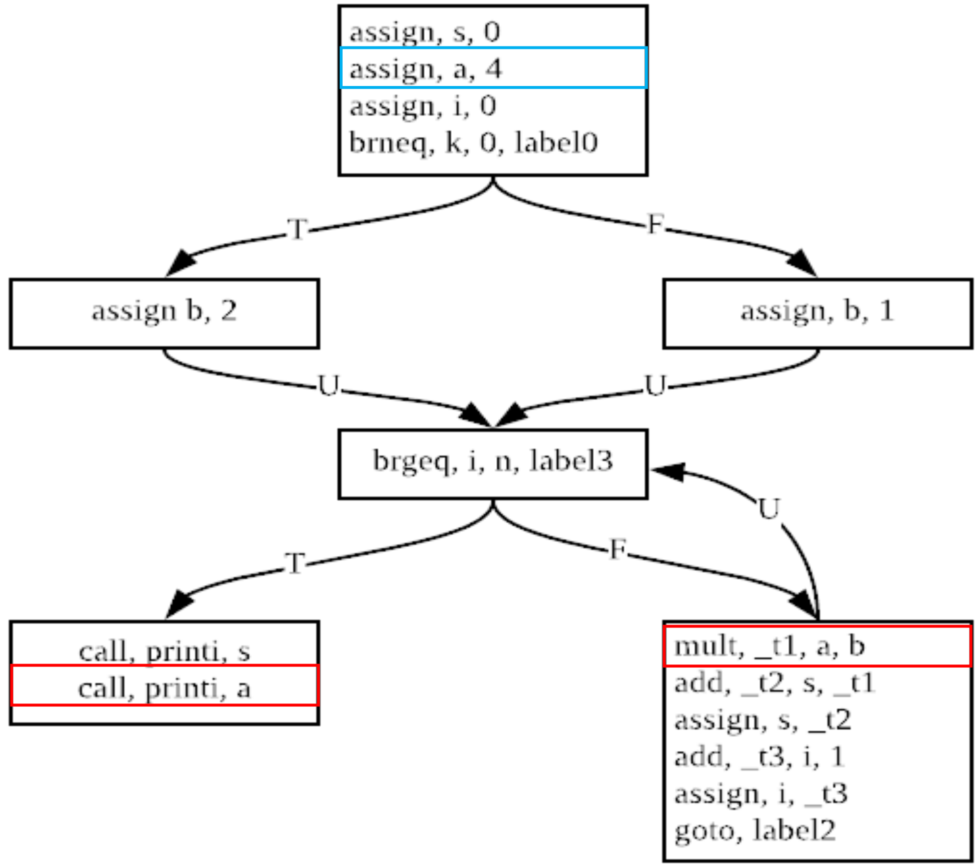
- Q2
  Which uses of ***a*** are reached by the def of ***a*** in line 2?

- Q3
  Which defs of ***b*** reach the use of ***b*** in line 12?

```
1     assign s, 0
2     assign a, 4
3     assign i, 0
// branch if (arg1 != arg2)
4     brneq k, 0, label0
5     assign, b, 1
6     goto, label1
7   label0:
8     assign, b, 2
9   label1:
10  label2:
// branch if (arg1 >= arg2)
11    brgeq, i, n, label3
12    mult, _t1 a, b
13    add, _t2, s, _t1
14    assign s, _t2
15    add _t3, i, 1
16    assign i, _t3
17    goto, label2
18  label3:
19    call, printi, s
20    call, printi, a
```

Q2 Solution

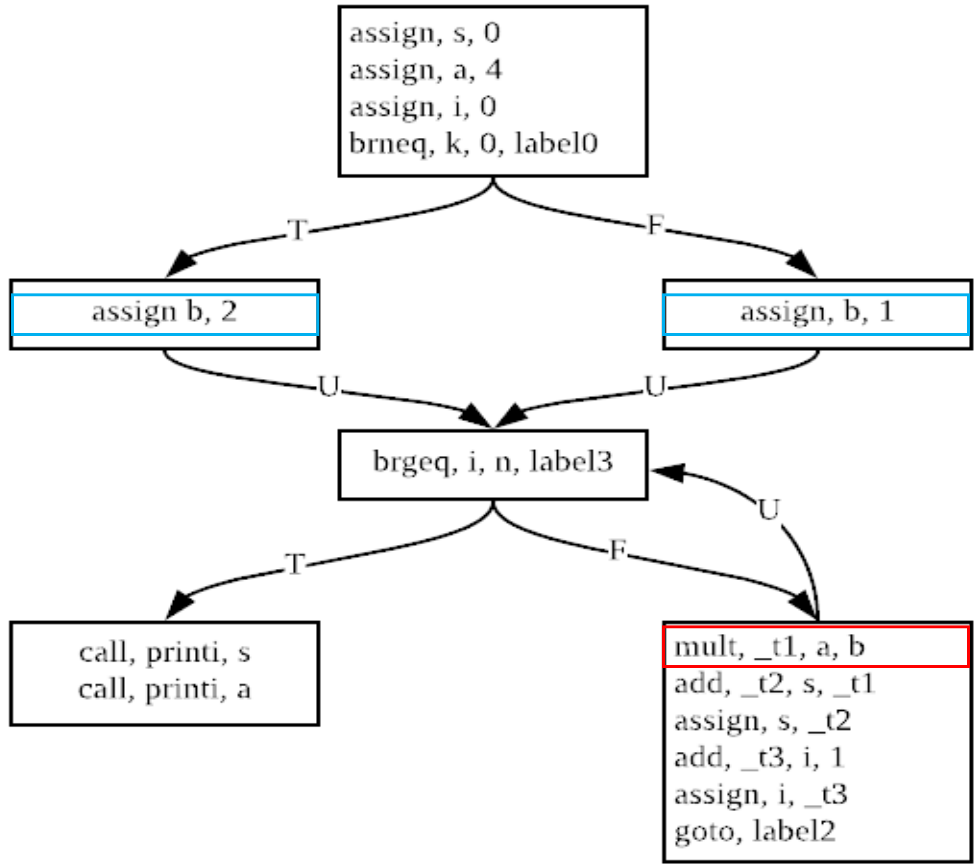
```
1    assign s, 0
2    assign a, 4
3    assign i, 0
// branch if (arg1 != arg2)
4    brneq k, 0, label0
5    assign, b, 1
6    goto, label1
7  label0:
8    assign, b, 2
9  label1:
10 label2:
// branch if (arg1 >= arg2)
11   brgeq, i, n, label3
12   mult, _t1 a, b
13   add, _t2, s, _t1
14   assign s, _t2
15   add _t3, i, 1
16   assign i, _t3
17   goto, label2
18 label3:
19   call, printi, s
20   call, printi, a
```



In line 2, variable ‘a’ is defined as 4. <u>The definition of ‘a’ in line 2 reaches line **12 & 20** without any intervening ‘def of a’.</u>

Q3 Solution

```
1    assign s, 0
2    assign a, 4
3    assign i, 0
// branch if (arg1 != arg2)
4    brneq k, 0, label0
5    assign, b, 1
6    goto, label1
7  label0:
8    assign, b, 2
9  label1:
10 label2:
// branch if (arg1 >= arg2)
11   brgeq, i, n, label3
12   mult, _t1 a, b
13   add, _t2, s, _t1
14   assign s, _t2
15   add _t3, i, 1
16   assign i, _t3
17   goto, label2
18 label3:
19   call, printi, s
20   call, printi, a
```



There is no intervening ‘def of b’ in the control flow between line 5 and line 12. Same for line 8 and line 12. The defs of b in lines **5 & 8** both reach the use of b in line 12.

Comments about Q2, Q3

- Most students answered these questions correctly.
- Some students were not sure about the meaning of the terms **‘def’** and **‘use’**.

  ‘def’ :
  a write operation on a variable (short for “definition”)

  ‘use’ :
  a read operation on a variable

Formalizing a Solution to the Reaching Definitions Problem (Recap)

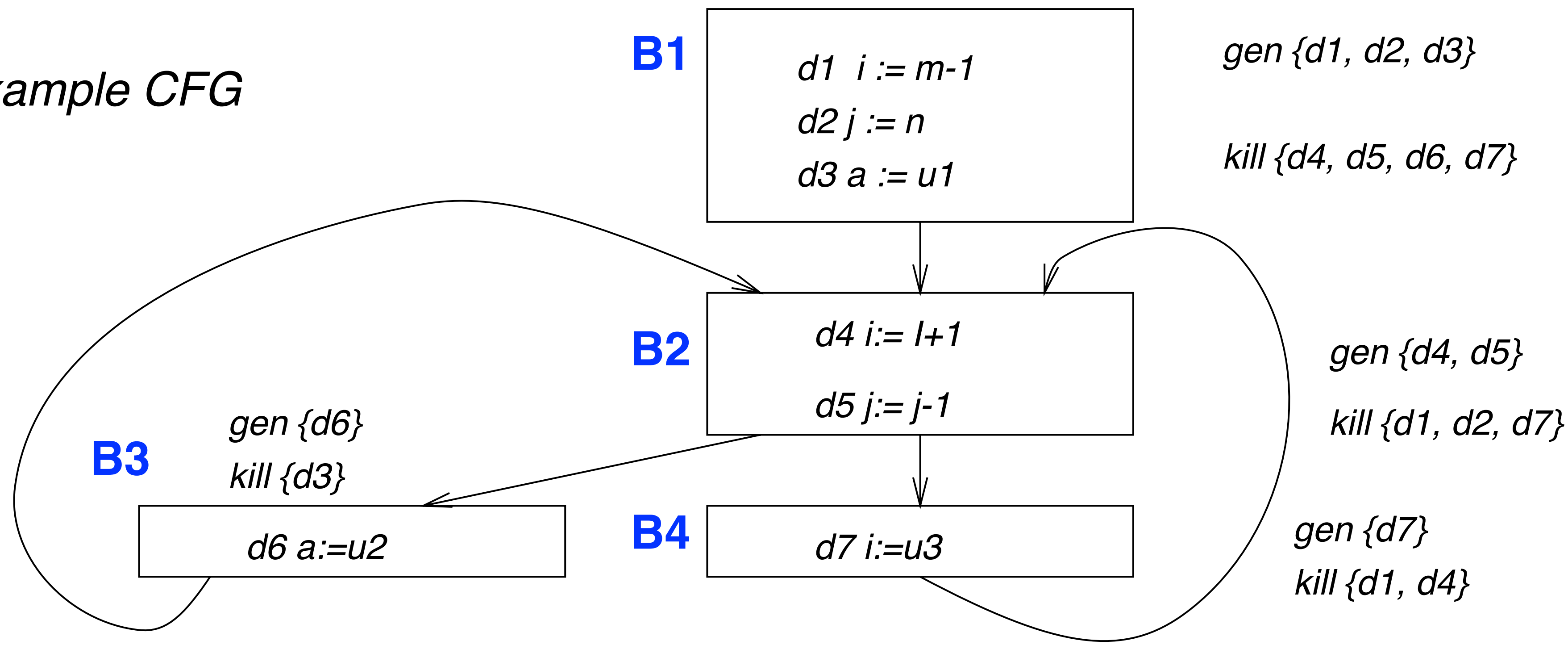
- Given a statement/instruction S, define
  - Local sets that can be extracted from S
    - GEN[S] = set of definitions in S ("generated" by S)
    - KILL[S] = set of definitions that may be overwritten by S (e.g., all definitions in program that write to S's lval, whether or not they reach S)
  - Global sets to be computed using CFG
    - IN[S] = set of definitions that reach the entry point of S
    - OUT[S] = set of definitions in S as well as definitions from IN[S] that go beyond S (are not "killed" by S)
- Data flow equations (invariants) for these sets

$$\text{OUT[S]} = \text{GEN[S]} \cup (\text{IN[S]} - \text{KILL[S]})$$

$$\text{IN[S]} = \cup_{p \in predecessors} \text{OUT[p]}$$

Data Flow Equations are Recursive!

Example CFG



OUT[B1] = GEN[B1] U (IN[B1] - KILL[B1]) IN[B1] = { } // empty set OUT[B2] = GEN[B2] U (IN[B2] - KILL[B2]) IN[B1] = OUT[B1] U OUT[B3] U OUT[B4]	OUT[B3] = GEN[B3] U (IN[B3] - KILL[B3]) IN[B3] = OUT[B2] OUT[B4] = GEN[B4] U (IN[B4] - KILL[B4]) IN[B4] = OUT[B2]

FIXED POINT ITERATION METHOD

Fixed point : A point, say, **s** is called a fixed point if it satisfies the equation $\mathbf{x = g(x)}$.

Fixed point Iteration : The transcendental equation $\mathbf{f(x) = 0}$ can be converted algebraically into the form $\mathbf{x = g(x)}$ and then using the iterative scheme with the recursive relation

$$x_{i+1} = g(x_i), \qquad i = 0, 1, 2, \ldots,$$

with some initial guess $\mathbf{x_0}$ is called the fixed point iterative scheme.

Algorithm - Fixed Point Iteration Scheme

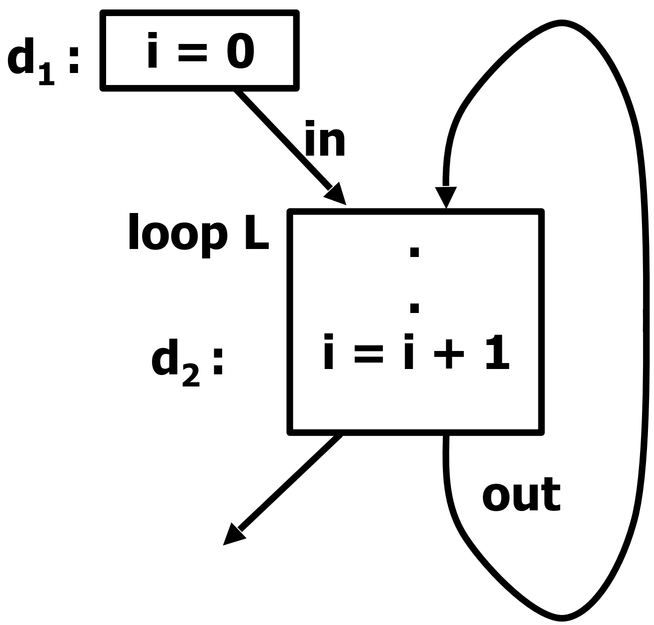
Given an equation $f(x) = 0$
Convert $f(x) = 0$ into the form $x = g(x)$
Let the initial guess be x_0
Do
    $\mathbf{x_{i+1} = g(x_i)}$
while (none of the convergence criterion C1 or C2 is met)

- C1. Fixing apriori the total number of iterations **N** .
- C2. By testing the condition $\mathbf{| x_{i+1} - g(x_i) |}$ (where **i** is the iteration number) less than some tolerance limit, say epsilon, fixed apriori.

Source: https://mat.iitm.ac.in/home/sryedida/public_html/caimna/transcendental/iteration%20methods/fixed-point/iteration.html

Reaching Definitions as an example of Data Flow Analysis

Data Flow Analysis = finding solution to recursive data flow equations



Question:

What is the set of reaching definitions at the exit of the loop L?

in [L] = {d_1} ∪ out[L]
gen [L] = {d_2}
kill [L] = {d_1}
out [L] = gen [L] ∪ {in [L] - kill[L]}

in[L] depends on out[L], and out[L] depends on in[L]!!

Solution?

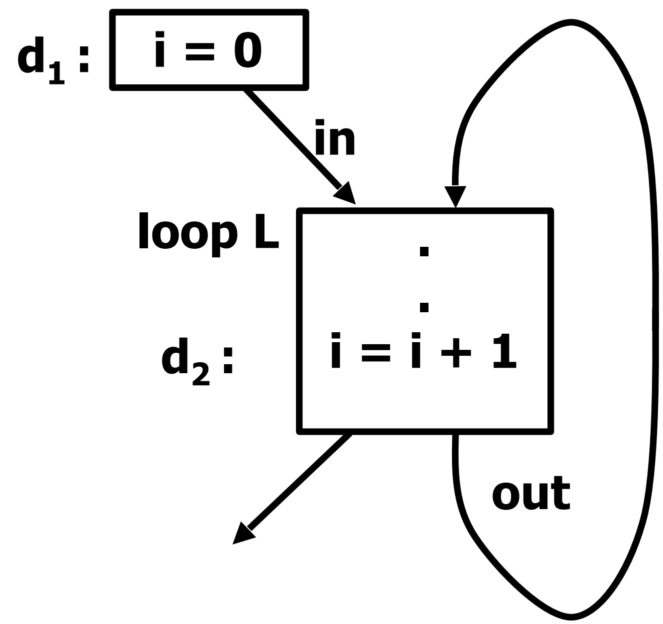
Initialization

$out[L] = \varnothing$

First iteration

$in[L] = \{d_1\} \cup out[L]$
$= \{d_1\}$

$out[L] = gen\ [L]\ \cup\ (in\ [L] - kill\ [L])$
$= \{d_2\} \cup (\{d_1\} - \{d_1\})$
$= \{d_2\}$



in [L] = $\{d_1\} \cup$ out[L]
gen [L] = $\{d_2\}$
kill [L] = $\{d_1\}$
out [L] = gen [L] $\cup$ {in [L] - kill[L]}

Solution

First iteration

out[L] = $\{d_2\}$

Second iteration

in[L] = $\{d_1\} \cup$ out[L]

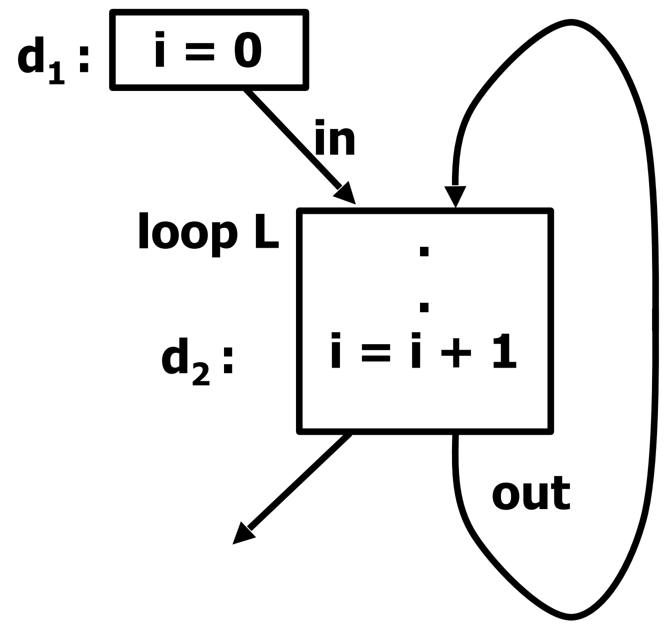
$= \{d_1, d_2\}$

out[L] = *gen* [L] $\cup$ (in [L] - kill [L])

$= \{d_2\} \cup \{\{d_1, d_2\} - \{d_1\}\}$

$= \{d_2\} \cup \{d_2\}$

$= \{d_2\}$

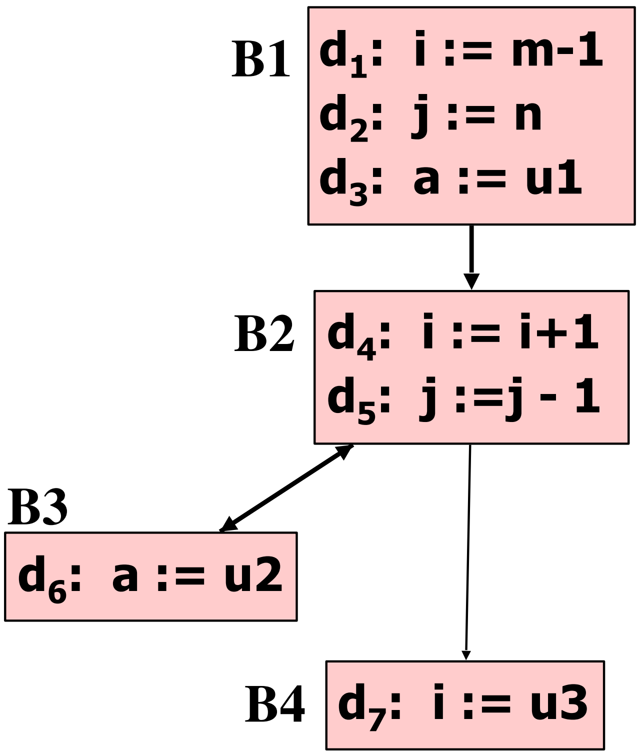


in [L] = $\{d_1\} \cup$ out[L]
gen [L] = $\{d_2\}$
kill [L] = $\{d_1\}$
out [L] = gen [L] $\cup$ {in [L] - kill[L]}

We reached the fixed point!

Iterative Algorithm for Reaching Definitions

Step 1: Compute gen and kill for each basic block



gen[B1] = {d_1, d_2, d_3}
kill[B1] = {d_4, d_5, d_6, d_7}

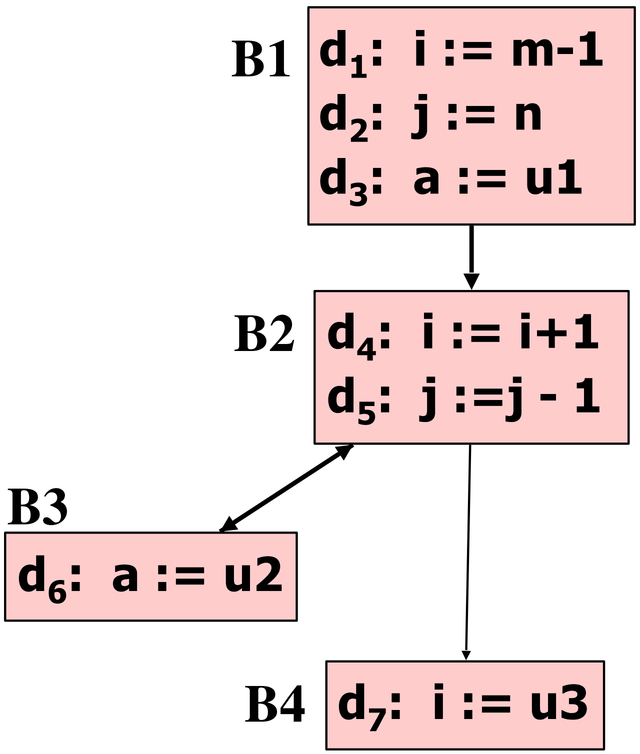
gen[B2] = {d_4, d_5}
kill [B2] = {d_1, d_2, d_7}

gen[B3] = {d_6}
kill [B3] = {d_3}

gen[B4] = {d_7}
kill [B4] = {d_1, d_4}

(AhoSethiUllman, pp. 626)

Iterative Algorithm for Reaching Definitions



Step 2: For every basic block, make:

out[B] = gen[B]

Initialization:

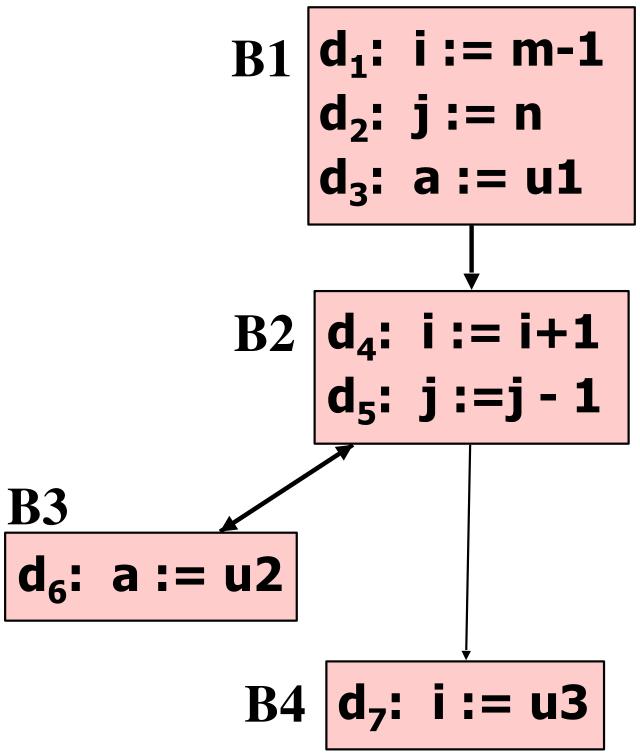
in[B1] = ∅
out[B1] = {d_1, d_2, d_3}

in[B2] = ∅
out[B2] = {d_4, d_5}

in[B3] = ∅
out[B3] = {d_6}

in[B4] = ∅
out[B4] = {d_7}

Iterative Algorithm for Reaching Definitions



To simplify the representation, the in[B] and out[B] sets are represented by bit strings. Assuming the representation $d_1d_2d_3\ d_4d_5d_6d_7$ we obtain:

Initialization:

in[B1] = ∅
out[B1] = {d_1, d_2, d_3}

in[B2] = ∅
out[B2] = {d_4, d_5}

in[B3] = ∅
out[B3] = {d_6}

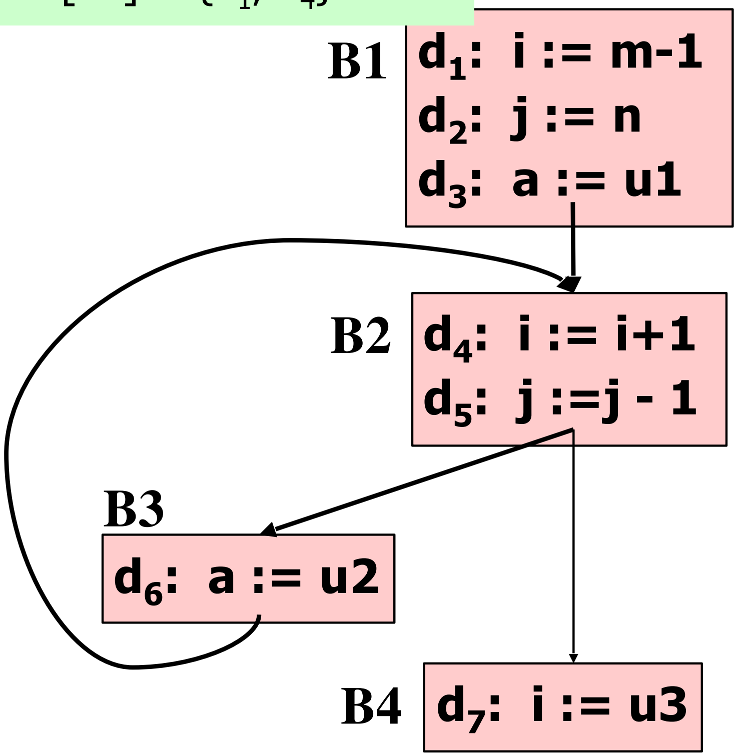
in[B4] = ∅
out[B4] = {d_7}

Block	Initial	
	in[B]	out[B]
B_1	000 0000	111 0000
B_2	000 0000	000 1100
B_3	000 0000	000 0010
B_4	000 0000	000 0001

Notation: $d_1d_2d_3\ d_4d_5d_6d_7$

Algorithm for Reaching Definitions

gen[B1] = {d_1, d_2, d_3}
kill[B1] = {d_4, d_5, d_6, d_7}
gen[B2] = {d_4, d_5}
kill [B2] = {d_1, d_2, d_7}
gen[B3] = {d_6}
kill [B3] = {d_3}
gen[B4] = {d_7}
kill [B4] = {d_1, d_4}



while a fixed point is not found:

in[B] = ∪ out[P] where P is a predecessor of B

out[B] = gen[B] ∪ (in[B]-kill[B])

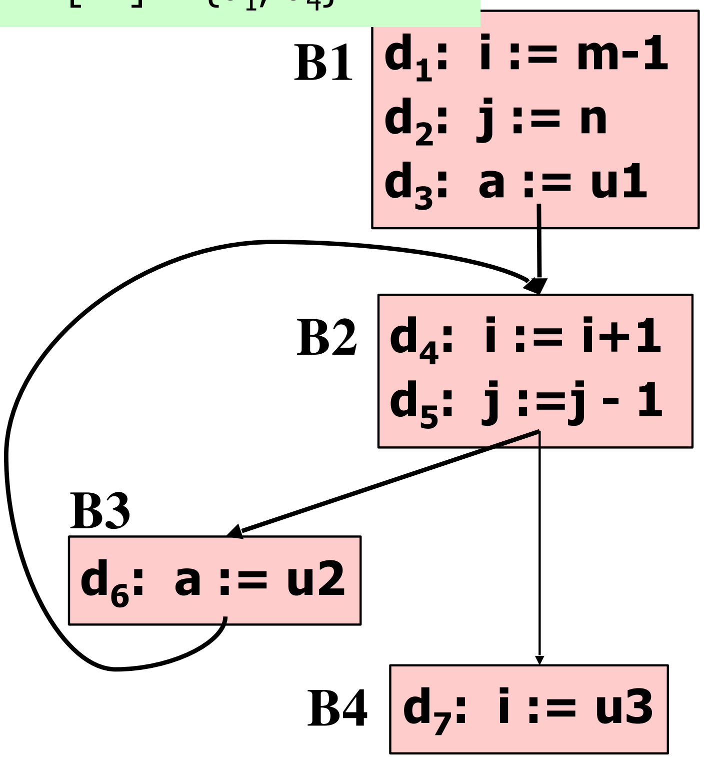
Block	Initial	
	in[B]	out[B]
B_1	000 0000	111 0000
B_2	000 0000	000 1100
B_3	000 0000	000 0010
B_4	000 0000	000 0001

Block	First Iteration	
	in[B]	out[B]
B_1	000 0000	111 0000
B_2	111 0010	001 1110
B_3	000 1100	000 1110
B_4	000 1100	000 0101

Notation: $d_1 d_2 d_3$ $d_4 d_5 d_6 d_7$

gen[B1] = {d_1, d_2, d_3}
kill[B1] = {d_4, d_5, d_6, d_7}
gen[B2] = {d_4, d_5}
kill [B2] = {d_1, d_2, d_7}
gen[B3] = {d_6}
kill [B3] = {d_3}
gen[B4] = {d_7}
kill [B4] = {d_1, d_4}

Algorithm for Reaching Definitions



while a fixed point is not found:
in[B] = ∪ out[P] where P is a predecessor of B
out[B] = gen[B] ∪ (in[B]-kill[B])

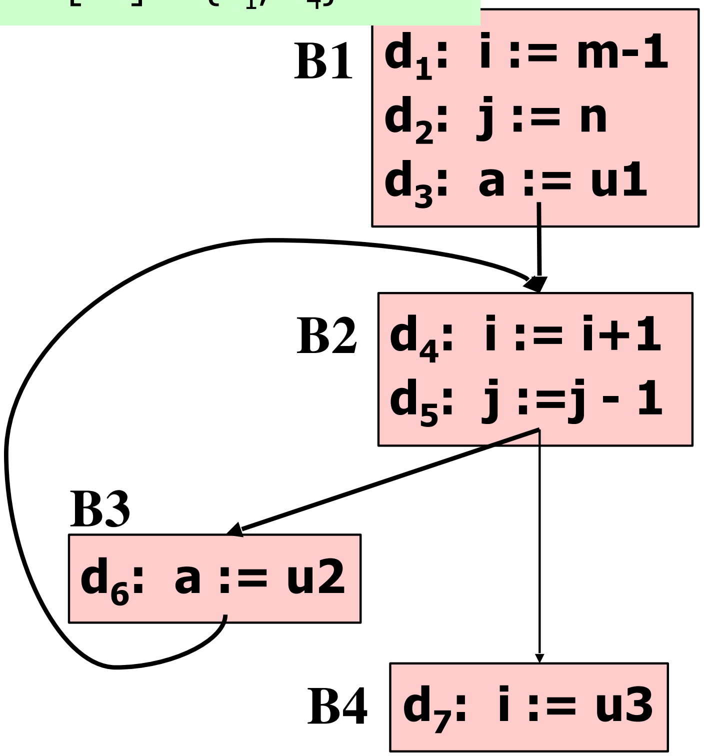
Block	First Iteration	
	in[B]	out[B]
B_1	000 0000	111 0000
B_2	111 0010	001 1110
B_3	000 1100	000 1110
B_4	000 1100	000 0101

Block	Second Iteration	
	in[B]	out[B]
B_1	000 0000	111 0000
B_2	111 1110	001 1110
B_3	001 1110	000 1110
B_4	001 1110	001 0111

Notation: $d_1d_2d_3\ d_4d_5d_6d_7$

gen[B1] = {d_1, d_2, d_3}
kill[B1] = {d_4, d_5, d_6, d_7}
gen[B2] = {d_4, d_5}
kill [B2] = {d_1, d_2, d_7}
gen[B3] = {d_6}
kill [B3] = {d_3}
gen[B4] = {d_7}
kill [B4] = {d_1, d_4}

Algorithm for Reaching Definitions



while a fixed point is not found:

in[B] = ∪ out[P] where P is a predecessor of B

out[B] = gen[B] ∪ (in[B]-kill[B])

Block	Second Iteration	
	in[B]	out[B]
B_1	000 0000	111 0000
B_2	111 1110	001 1110
B_3	001 1110	000 1110
B_4	001 1110	001 0111

Block	Third Iteration	
	in[B]	out[B]
B_1	000 0000	111 0000
B_2	001 1110	001 1110
B_3	000 1110	000 1110
B_4	001 0111	001 0111

Notation: $d_1d_2d_3\ d_4d_5d_6d_7$

Algorithm Convergence

Intuitively we can observe that the algorithm converges to a fix point because the out[B] set never decreases in size.

It can be shown that an upper bound on the number of iterations required to reach a fix point is the number of nodes in the flow graph.

> Intuitively, if a definition reaches a point, it can only reach the point through a cycle free path, and no cycle free path can be longer than the number of nodes in the graph.

Empirical evidence suggests that for real programs the number of iterations required to reach a fix point is less then five.

Algorithm Summary: Inputs and Outputs

- *Input:* A flow graph for which $kill[B]$ and $gen[B]$ have been computed for each basic block B
- *Output:* $in[B]$ and $out[B]$ for each block B
- The Idea: Use an iterative approach where the "initial" in and out information is *propagated* across edges and along the paths of the graph *until* none of the outs change

  All computation is at the granularity of basic-blocks

Algorithm Summary: Overall Steps

Reaching Definitions:

- // Initialize *out* under the assumpt1¡ion that $in = \emptyset$ by setting $out[B] := gen[B]$ for all the blocks //
- $change :=$ **true**

  // This initiates the iteration and if there is a change after the iteration in *any* of the *out* sets, then it remains true//
- While *change* remains **true** compute
  - $in[B] = \cup_{p \in P}\, out[p]$ where P is the set of all predecessors of block B
- $tempout := out[B]$
- $out[B] := gen[B] \cup (in[B] - kill[B])$
- if $out[B] \neq temoput$ $change :=$ **true**

Today's in-class Worksheet

- Worksheets can be solved collaboratively
  - All other course work must be done individually or in project groups (see syllabus for details)
- Each student should turn in their own solution, based on collaborative discussions
- Worksheets will not be graded or returned, but solutions will be provided
- Worksheets will contribute to class participation grade
- Worksheets will inform teaching staff of concepts that need to be reviewed/reinforced in future lectures