Chapter 7: Introduction to Data-Flow Analysis

Yepang Liu

liuyp1@sustech.edu.cn

Outline

- Data-Flow Analysis

- Classic Data-Flow Problems
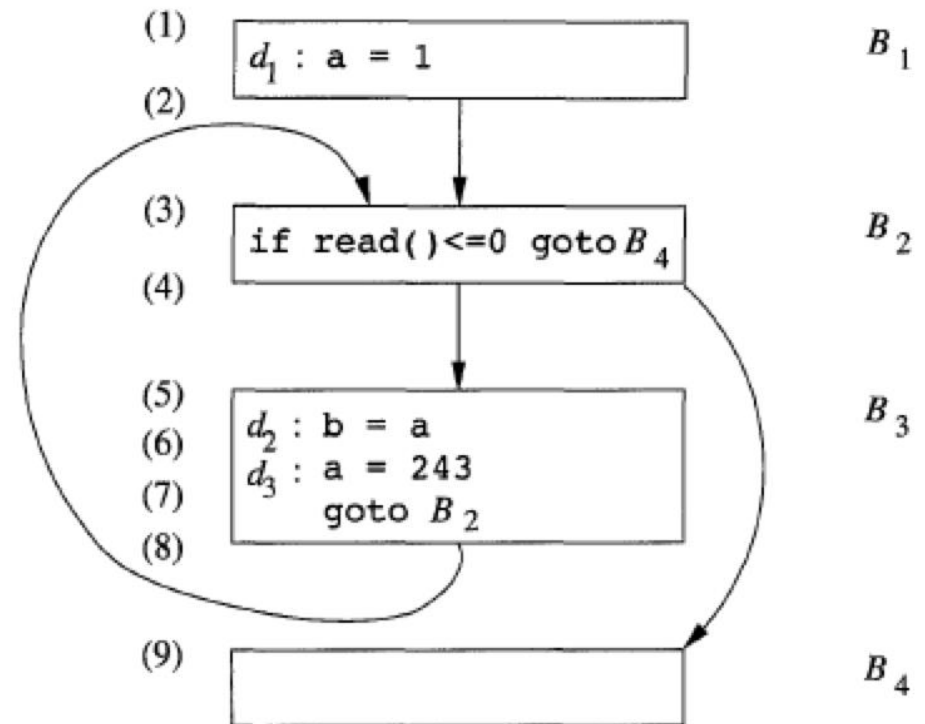
Introduction

- Code optimization (or code improvement)

  - Eliminating unnecessary instructions in object code

  - Replacing one sequence of instructions by a faster sequence of instructions that <u>does the same thing</u>

- Global optimizations

  - Performed on a flow graph as a whole, involving multiple basic blocks

  - Most global optimizations are based on data-flow analysis

- A compilers knows only how to apply relatively low-level semantic transformations to optimize code

  - High-level optimizations: architectural/algorithmic changes, refactoring, etc.

What Is Data-Flow Analysis?

- A body of techniques that <u>derive information about the flow of data</u> along program execution paths

  - **Example 1:** Whether two textually identical expressions evaluate to the same value along any execution path? (Identify common subexpressions)

  - **Example 2:** Whether the result of an assignment is used on subsequent execution paths? (Eliminate dead code)

- Data-flow analysis is the foundation of many optimizations
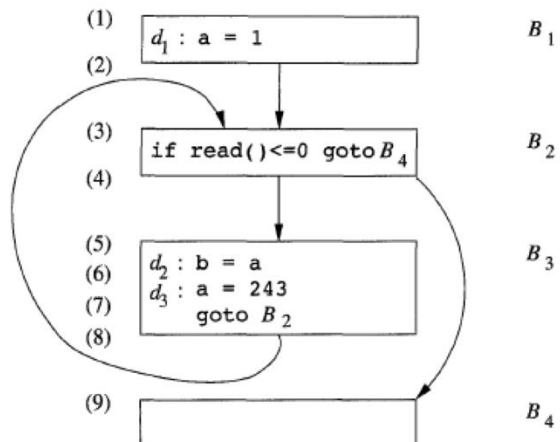
The Data-Flow Abstraction

- **Program points**: points before and after each statement

  - Within one block, the program point after a statement is the same as the program point before the next statement (e.g., 6)

  - If there is an edge from block *B* to block *C*, then the program point after the last statement of *B* may be followed immediately by the program point before the first statement of *C* (e.g., 8 and 3)

(1)

$d_1 : a = 1$ B_1

(2)

(3)

if read()<=0 goto B_4 B_2

(4)

(5)
(6) $d_2 : b = a$ B_3
(7) $d_3 : a = 243$
(8)   goto B_2

(9) B_4

The Data-Flow Abstraction

- An **execution path** from point p_1 to point p_n is a sequence of points $p_1, p_2, \ldots, p_n$ such that for each $i = 1, 2, \ldots, n\text{-}1$:
  - either p_i is the point immediately preceding a statement and p_{i+1} is the point immediately follow that statement (the "within block" case)
  - or p_i is the end of a block and p_{i+1} is the beginning of a successor block (the "across block" case)
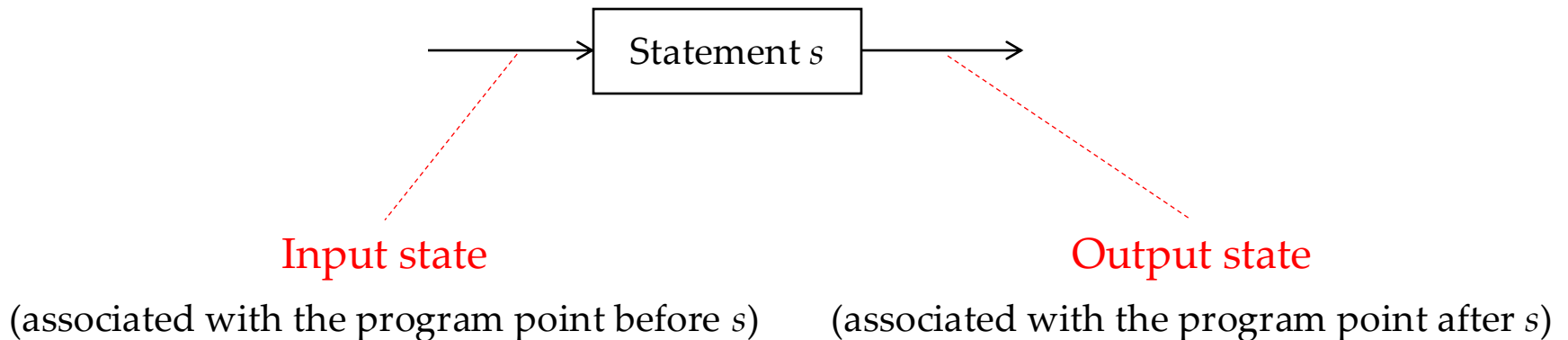

The shortest complete execution path: (1, 2, 3, 4, 9)

The next shortest path executing one iteration of the loop: (1, 2, 3, 4, 5, 6, 7, 8, 3, 4, 9)

The Data-Flow Abstraction

- Program execution can be viewed as *a series of transformations of the program state* (the values of all variables)

  - Each execution of a statement transforms an input state to a new output state

```
           ┌─────────────┐
  ─────────→│ Statement s │─────────→
           └─────────────┘
```

Input state

(associated with the program point before s)

Output state

(associated with the program point after s)

The Data-Flow Abstraction

- When analyzing a program, we must consider all the possible execution paths through a flow graph

    - In general, there is an infinite number of possible paths due to the existence of loops and recursions

- Data-flow analyses summarize all the possible program states that can occur at a program point with a finite set of facts

    - Different analyses may choose to abstract out different information (i.e., obtaining approximations)

    - In general, no analysis is necessarily a perfect representation of the state

Example (1)
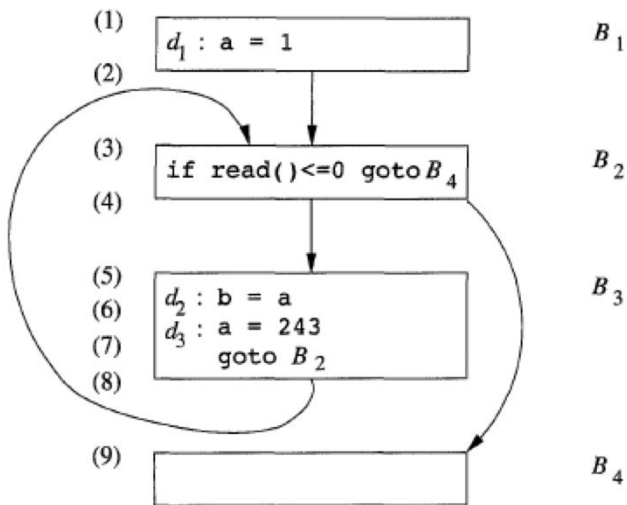
Reaching definitions:

- The first time point (5) is executed, the value of a is 1, i.e., definition d_1 reaches (5) in the first iteration

- In subsequent iterations, d_3 reaches point (5) and the value of a is 243

- So, we may summarize all the program states at (5) by saying that <u>the value of a is one of {1, 243} and defined by one of {d_1, d_3}</u>

Static approximation.
The exact value depends on the execution path at runtime.

Example (2)
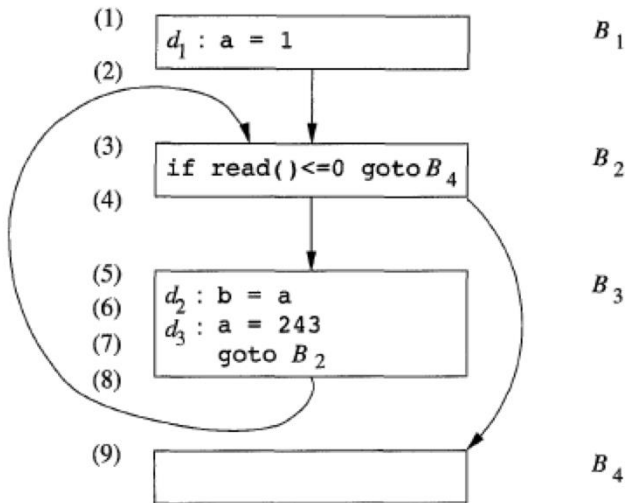
Constant folding:

- Find those definitions that are the unique definition of their variable to reach a given program point, regardless of the execution paths

- For this task, we describe some variables as "constant" or "not a constant"

  - *a* is not a constant at point (5) and thus cannot be replaced by a constant value

The Data-Flow Analysis Schema

- We associate with every program point a *data-flow value* that represents an abstraction of program states observed for that point[*]
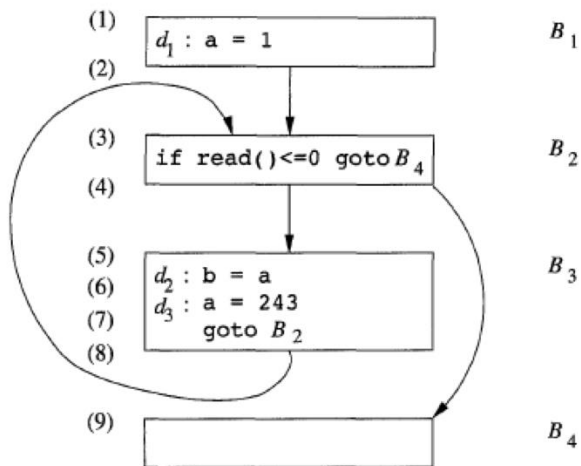
Reaching definitions:

The data-flow value at (5) is the set of a's definitions that can reach (5), that is, $\{d_1, d_3\}$

Constant folding:

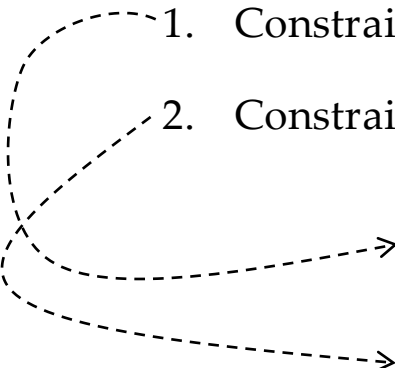The data-flow value for variable a at (5) is NAC (not a constant)

[*] In each application of data-flow analysis, the set of possible data-flow values is the domain for the application.

The Data-Flow Analysis Schema

- IN[*s*]: The data-flow value **before** the statement *s*

- OUT[*s*]: The data-flow value **after** the statement *s*

- The *data-flow problem* is to find a solution to a set of constraints on the IN[*s*]'s and OUT[*s*]'s for all statements *s*

  1. Constraints based on the semantics of the statements ("*transfer functions*")

  2. Constraints based on the flow of control

Statement exeuctions may alter data-flow values

Control flows propagate data-flow values

Transfer Functions (传递函数)

- The relationship between the data-flow values before and after each statement is known as a *transfer function*

- In a forward-flow problem (information propagate forward along execution paths), the transfer function f_s takes the data-flow value before the statement s and produces a new data-flow value after s

  - OUT[s] = f_s(IN[s])

- In a backward-flow problem (information flow backwards up the execution paths), the transfer function f_s takes the data-flow value after the statement s and produces a new data-flow value before s

  - IN[s] = f_s(OUT[s])

Control-Flow Constraints

1. **Within a basic block** of statements s_1, s_2, …, s_n, the data-flow value out of s_i is the same as that into s_{i+1}

   - IN[s_{i+1}] = OUT[s_i], for all i = 1, 2, …, n-1

2. **Control-flow edges between basic blocks may create more complex constraints**

   - For example, in reaching definitions analysis, the set of definitions reaching the leader statement of a block should be <u>the union of the definitions</u> after the last statements of each of the predecessor blocks

Data-Flow Schemas on Basic Blocks (The Intra-Block Case)

- Within a basic block, control flows from the beginning to the end without interruption or branching

- For a block B consisting of statements s_1, s_2, …, s_n, we have[*]

    - $\text{IN}[B] = \text{IN}[s_1]$

    - $\text{OUT}[B] = \text{OUT}[s_n]$

    - $\text{OUT}[B] = f_B(\text{IN}[B])$, where $f_B = f_{s_n} \circ \dots \circ f_{s_2} \circ f_{s_1}$ (composing statement-level transfer functions)
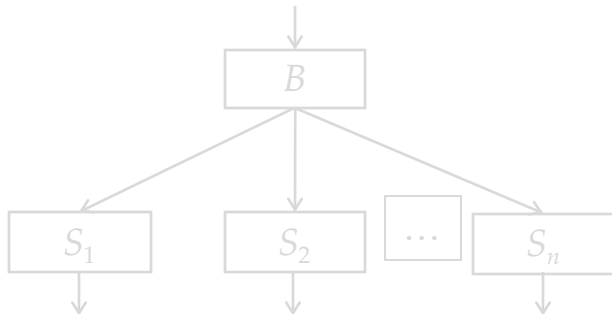
[*] For backward-flow problem, $\text{IN}[B] = f_B(\text{OUT}[B])$, where $f_B = f_{s_1} \circ f_{s_2} \circ \dots \circ f_{s_n}$

Data-Flow Schemas on Basic Blocks (The Inter-Block Case)
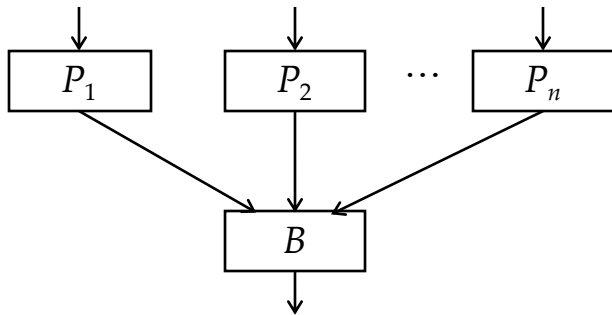
Forward-flow problem:

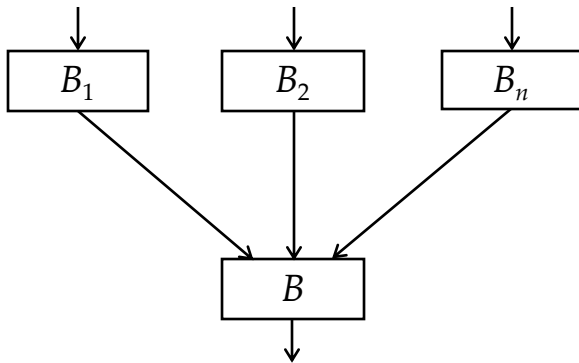$$\text{IN}[B] = \bigcup_{P \text{ a predecessor of } B} \text{OUT}[P]$$

Backward-flow problem:

$$\text{OUT}[B] = \bigcup_{S \text{ a successor of } B} \text{IN}[S]$$

$\cup$ is a generic *meet operator* depending on specific problems.

Example (Constant Folding Problem)

If:

OUT[B_1]: x: 3; y: 4; z: NAC

OUT[B_2]: x: 3; y: 5; z: 7

OUT[B_3]: x: 3; y: 4; z: 7

then:

IN[B]: x: 3; y: NAC; z: NAC

Solutions to Data-Flow Equations

- Data-flow equations usually do not have a unique solution

    - All data-flow schemas compute approximations to the ground truth

- Our goal is to find the most "precise" solution that satisfies both control-flow and transfer constraints

    - Being precise enables valid code improvements (in general, a higher precision leads to a better improvement)

    - Satisfying constraints guarantees the safety of transformations (does not change program semantics)

Outline

- Data-Flow Analysis

- Classic Data-Flow Problems
  - Reaching Definitions
  - Live Variables*
 - Available Expressions*

  * Optional self-study materials

Reaching Definitions

- A definition d of some variable x *reaches* a point p if there is a path from the program point after d to p, such that d is not "killed" along that path

  - d is *killed* if there is any other definition of x along the path

  - Intuitively, if d reaches the point p, then d might be the last definition of x

Example

Which definitions reach the block B_2?
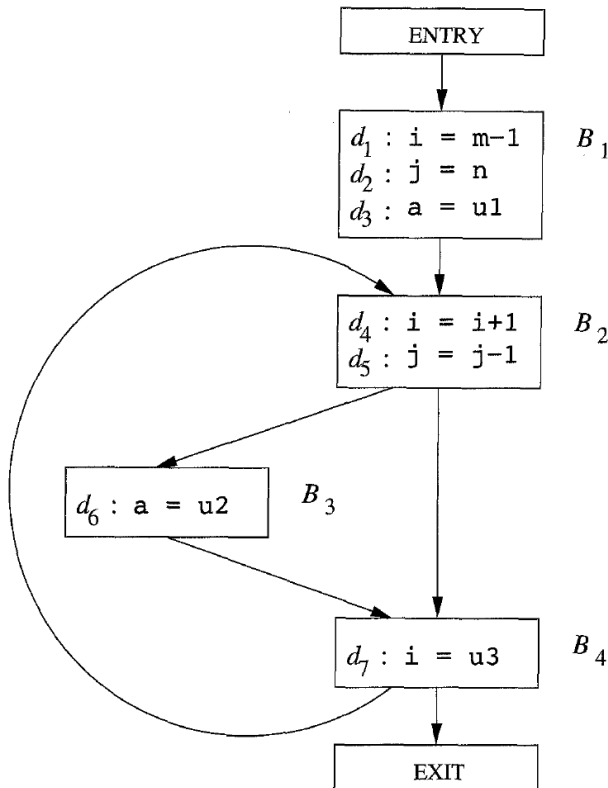
- d_1, d_2, d_3 reach B_2

- d_5 reaches B_2 since there is no other definition to j in the loop

- d_4 does not reach B_2 since i is always redefined by d_7 (i.e., d_7 reaches B_2)

- d_6 reaches B_2

Reaching Definitions

- For reaching definitions analysis, we allow inaccuracies. However, the decisions should be *"safe"*.

  - It is ok that some inferred reaching defs cannot actually reach a point

  - However, those defs that can reach a point should be identified

  - In other words, <u>over-approximations are acceptable</u>

- ***Reason of inaccuracies:*** In general, to decide whether each path in a flow graph can be taken is an undecidable problem[*]

  - We often simply assume that every path in the flow graph can be followed in some execution of the program

  * This is the well-known *path feasibility problem*.

Transfer Equations

- Consider a statement `d: u = v + w`

  - It *generates* a definition d of variable u and *kills* <u>all other definitions</u> of u

- <u>Transfer function of a statement</u> is in *gen-kill form*:

  - $f_d(x) = gen_d \cup (x - kill_d)$

    - x is the data-flow value (reaching definitions) before the statement

    - gen_d is the set of generated definitions, i.e., $\{d\}$

    - $kill_d$ is the set of killed definitions, i.e., all other definitions of u

Transfer Equations

- <u>Transfer function of a basic block</u> is also in *gen-kill* form

- Consider a block of only two statements:

$\text{IN}[1] = x$
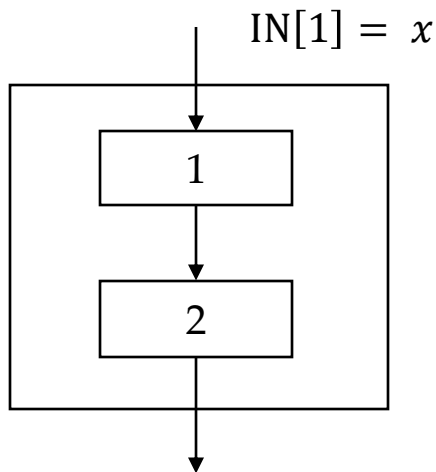
$\text{OUT}[1] = f_1(x) = gen_1 \cup (x - kill_1)$

$\text{OUT}[2] = f_2(\text{IN}[2]) = f_2(\text{OUT}[1])$

$= gen_2 \cup (\text{OUT}[1] - kill_2)$

$= gen_2 \cup ((gen_1 \cup (x - kill_1)) - kill_2)$

$= gen_2 \cup (gen_1 - kill_2) \cup (x - (kill_1 \cup kill_2))$

Transfer Equations

- Generalize to a block B with n statements, we have:

$$f_B(x) = gen_B \cup (x - kill_B)$$

where

$$kill_B = kill_1 \cup kill_2 \cup \cdots \cup kill_n$$

and

$$gen_B = gen_n \cup (gen_{n-1} - kill_n) \cup$$
$$(gen_{n-2} - kill_{n-1} - kill_n) \cup$$
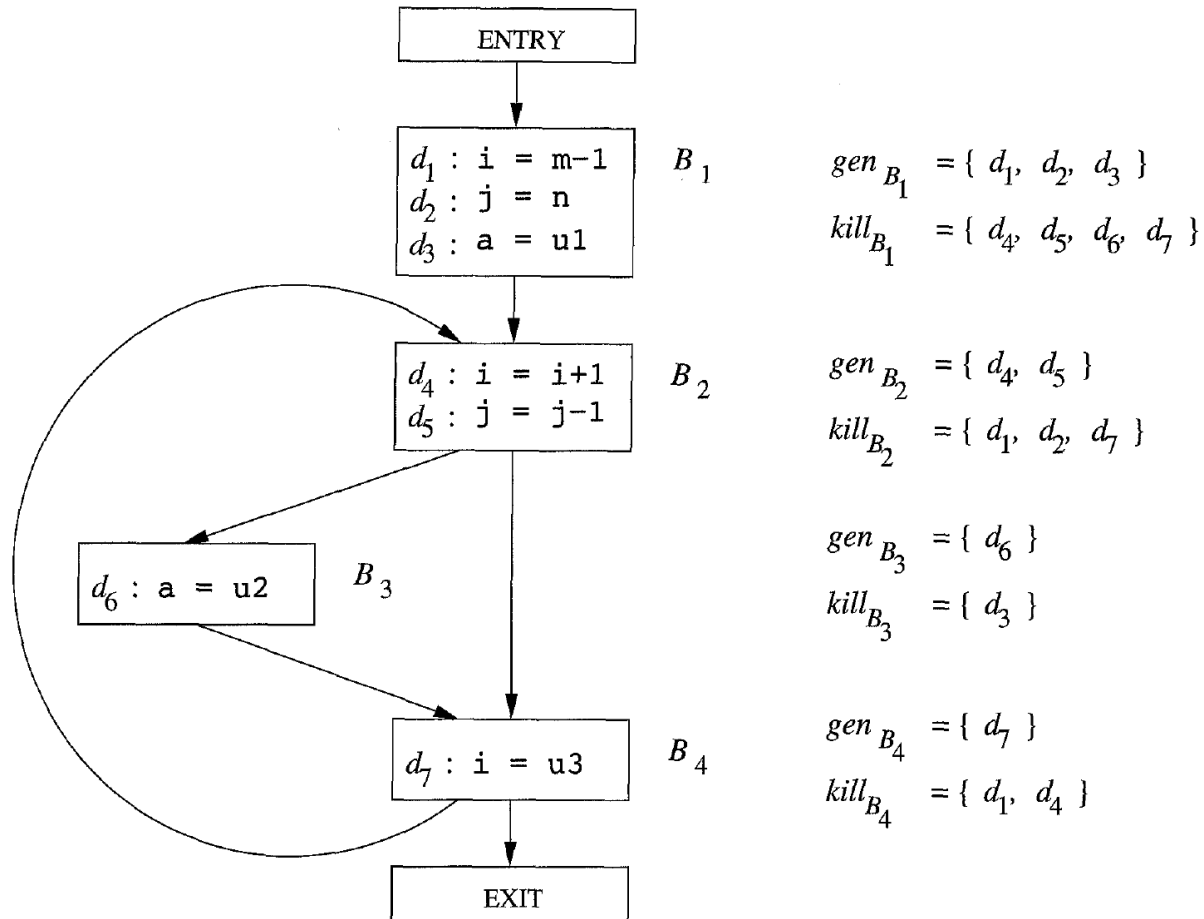$$\ldots \cup (gen_1 - kill_2 - kill_3 - \cdots - kill_n)$$

$\underline{\textbf{kill}_B}$

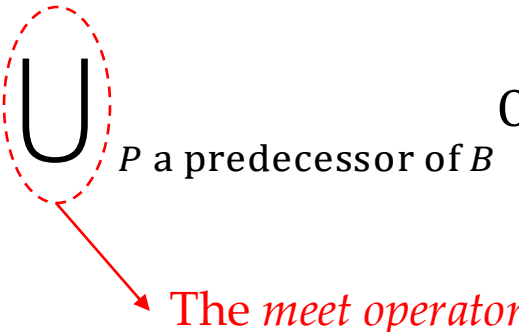The *kill* set is the union of all the defs killed by the individual statements

$\underline{\textbf{gen}_B}$

The *gen* set contains all the defs inside the block that are *downward exposed* (visible immediately after the block)

Example

$gen_{B_1} = \{ d_1, d_2, d_3 \}$

$kill_{B_1} = \{ d_4, d_5, d_6, d_7 \}$

$gen_{B_2} = \{ d_4, d_5 \}$

$kill_{B_2} = \{ d_1, d_2, d_7 \}$

$gen_{B_3} = \{ d_6 \}$

$kill_{B_3} = \{ d_3 \}$

$gen_{B_4} = \{ d_7 \}$

$kill_{B_4} = \{ d_1, d_4 \}$

Control-Flow Equations

$$\text{IN}[B] = \bigcup_{P \text{ a predecessor of } B} \text{OUT}[P]$$

The *meet operator*

- **Rationale:**

  - A definition reaches a program point as long as there exists at least one path along which the definition reaches. This explains why $\text{OUT}[P] \subseteq \text{IN}[B]$.

  - Since a definition cannot reach a point unless there is a path along which it reaches, $\text{IN}[B]$ needs to be no larger than the union of the reaching definitions of all the predecessor blocks

Problem Formulation

- The reaching definitions problem is defined by the following equations:

  - For the ENTRY block: OUT[ENTRY] = ∅

    - Since no definitions reach the beginning of a program

  - For all basic blocks B other than ENTRY:

    - $\text{OUT}[B] = gen_B \cup (\text{IN}[B] - kill_B)$

    - $\text{IN}[B] = \bigcup_{P \text{ a predecessor of } B} \text{OUT}[P]$
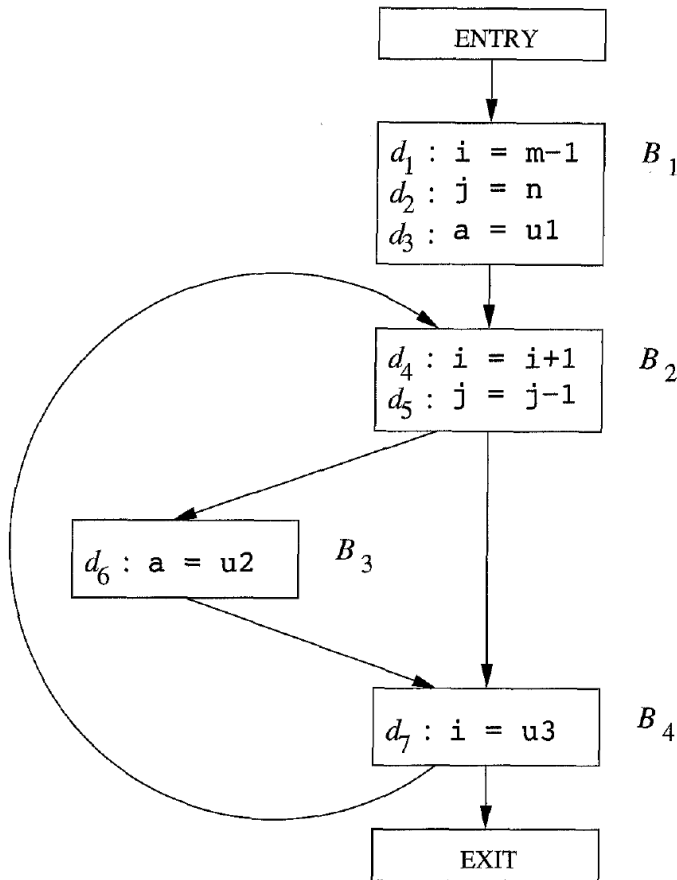
The Iterative Algorithm

- **Input:** A flow graph with $kill_B$ and gen_B computed for each block B

- **Output:** IN$[B]$ and OUT$[B]$ for each block B

$$
\begin{aligned}
&1) \quad \text{OUT}[\text{ENTRY}] = \emptyset; \\
&2) \quad \textbf{for } (\text{each basic block } B \text{ other than ENTRY}) \text{ OUT}[B] = \emptyset; \\
&3) \quad \textbf{while } (\text{changes to any OUT occur}) \\
&4) \quad\quad\quad \textbf{for } (\text{each basic block } B \text{ other than ENTRY}) \{ \\
&5) \quad\quad\quad\quad\quad \text{IN}[B] = \bigcup_{P \text{ a predecessor of } B} \text{OUT}[P]; \\
&6) \quad\quad\quad\quad\quad \text{OUT}[B] = gen_B \cup (\text{IN}[B] - kill_B); \\
&\quad\quad\quad\quad\quad\quad \}
\end{aligned}
$$

Example

1) OUT[ENTRY] = ∅;
2) **for** (each basic block B other than ENTRY) OUT[B] = ∅;
3) **while** (changes to any OUT occur)
4)   **for** (each basic block B other than ENTRY) {
5)   IN[B] = $\bigcup_{P \text{ a predecessor of } B}$ OUT[P];
6)   OUT[B] = $gen_B \cup$ (IN[B] − $kill_B$);
        }

Initialization:

OUT[ENTRY] = ∅

$gen_{B_1} = \{ d_1, d_2, d_3 \}$

$kill_{B_1} = \{ d_4, d_5, d_6, d_7 \}$

OUT[B_1] = ∅

$gen_{B_2} = \{ d_4, d_5 \}$

$kill_{B_2} = \{ d_1, d_2, d_7 \}$

OUT[B_2] = ∅

$gen_{B_3} = \{ d_6 \}$

$kill_{B_3} = \{ d_3 \}$

OUT[B_3] = ∅

$gen_{B_4} = \{ d_7 \}$

$kill_{B_4} = \{ d_1, d_4 \}$

OUT[B_4] = ∅

OUT[EXIT] = ∅

Example
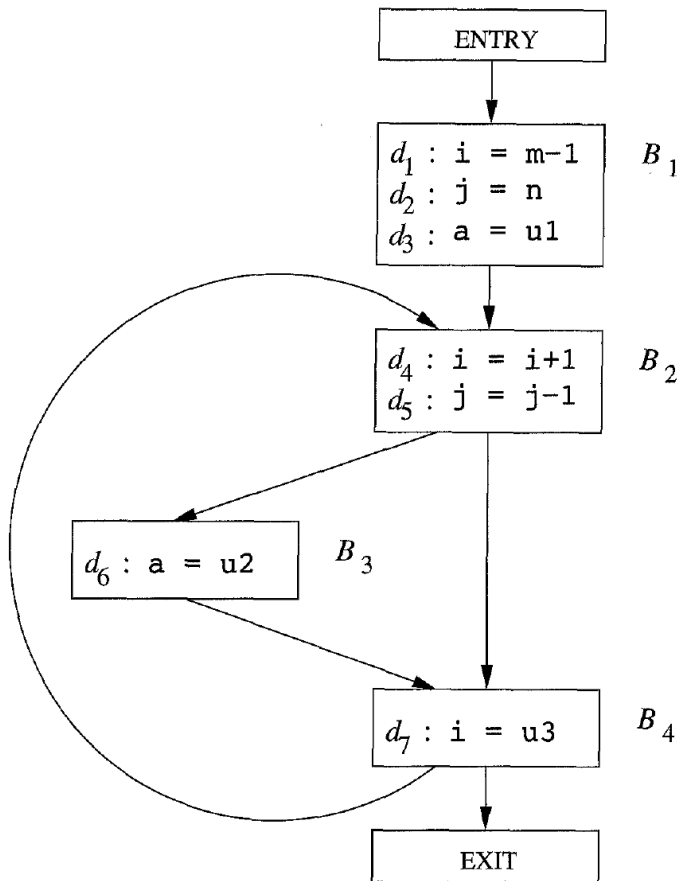
1) OUT[ENTRY] = ∅;
2) **for** (each basic block B other than ENTRY) OUT[B] = ∅;
3) **while** (changes to any OUT occur)
4)   **for** (each basic block B other than ENTRY) {
5)   IN[B] = $\bigcup_{P \text{ a predecessor of } B}$ OUT[P];
6)   OUT[B] = $gen_B \cup$ (IN[B] $- kill_B$);
        }

Iteration #1:

OUT[ENTRY] = ∅

IN[B_1] = ∅
OUT[B_1] = $\{d_1, d_2, d_3\}$ changed

IN[B_2] = $\{d_1, d_2, d_3\}$
OUT[B_2] = $\{d_3, d_4, d_5\}$ changed

IN[B_3] = $\{d_3, d_4, d_5\}$
OUT[B_3] = $\{d_4, d_5, d_6\}$ changed

IN[B_4] = $\{d_3, d_4, d_5, d_6\}$
OUT[B_4] = $\{d_3, d_5, d_6, d_7\}$ changed

IN[EXIT] = $\{d_3, d_5, d_6, d_7\}$
OUT[EXIT] = $\{d_3, d_5, d_6, d_7\}$ changed

ENTRY
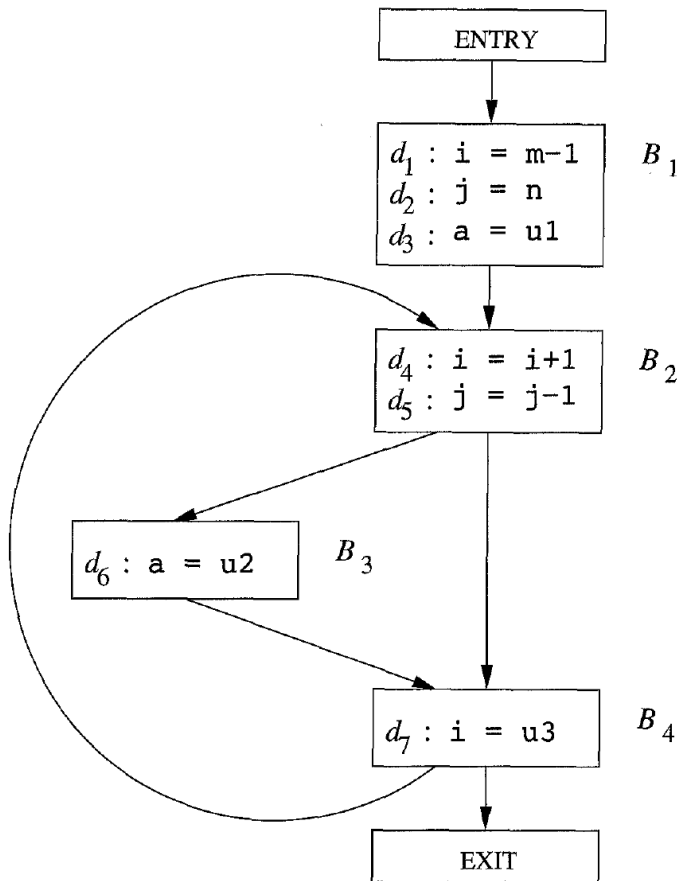
d_1 : i = m−1
d_2 : j = n
d_3 : a = u1
B_1

gen_{B_1} = { d_1, d_2, d_3 }
$kill_{B_1}$ = { d_4, d_5, d_6, d_7 }

d_4 : i = i+1
d_5 : j = j−1
B_2

gen_{B_2} = { d_4, d_5 }
$kill_{B_2}$ = { d_1, d_2, d_7 }

d_6 : a = u2
B_3

gen_{B_3} = { d_6 }
$kill_{B_3}$ = { d_3 }

d_7 : i = u3
B_4

gen_{B_4} = { d_7 }
$kill_{B_4}$ = { d_1, d_4 }

EXIT

Example

1) OUT[ENTRY] = ∅;
2) **for** (each basic block B other than ENTRY) OUT[B] = ∅;
3) **while** (changes to any OUT occur)
4)   **for** (each basic block B other than ENTRY) {
5)   IN[B] = $\bigcup_{P \text{ a predecessor of } B}$ OUT[P];
6)   OUT[B] = $gen_B \cup$ (IN[B] − $kill_B$);
    }

Iteration #2:

OUT[ENTRY] = ∅

IN[B_1] = ∅
OUT[B_1] = {d_1, d_2, d_3}

IN[B_2] = {$d_1, d_2, d_3, d_5, d_6, d_7$}
OUT[B_2] = {d_3, d_4, d_5, d_6}   changed

IN[B_3] = {d_3, d_4, d_5, d_6}
OUT[B_3] = {d_4, d_5, d_6}

IN[B_4] = {d_3, d_4, d_5, d_6}
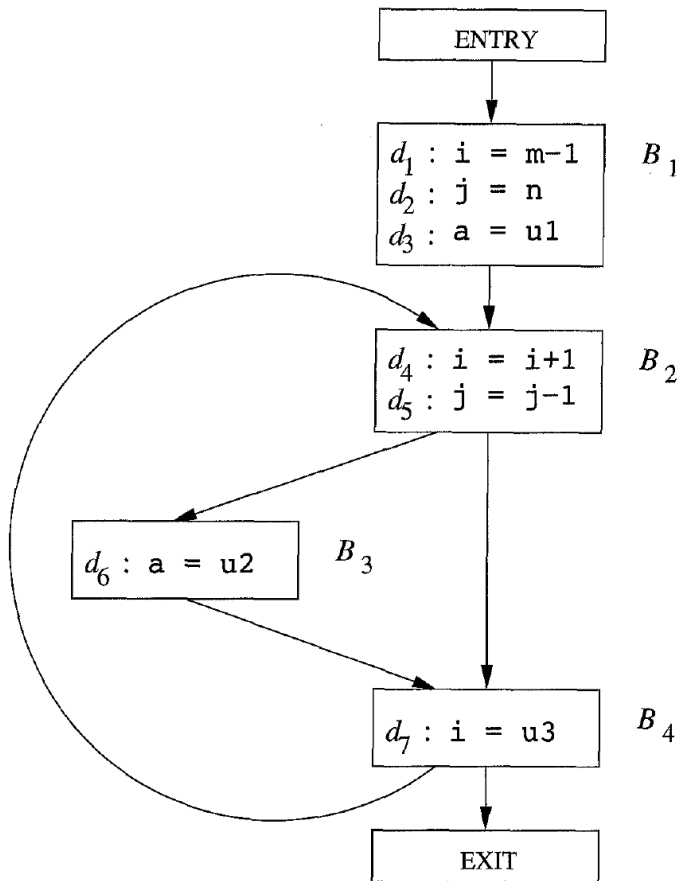OUT[B_4] = {d_3, d_5, d_6, d_7}

IN[EXIT] = {d_3, d_5, d_6, d_7}
OUT[EXIT] = {d_3, d_5, d_6, d_7}

$gen_{B_1} = \{ d_1, d_2, d_3 \}$

$kill_{B_1} = \{ d_4, d_5, d_6, d_7 \}$

$gen_{B_2} = \{ d_4, d_5 \}$

$kill_{B_2} = \{ d_1, d_2, d_7 \}$

$gen_{B_3} = \{ d_6 \}$

$kill_{B_3} = \{ d_3 \}$

$gen_{B_4} = \{ d_7 \}$

$kill_{B_4} = \{ d_1, d_4 \}$

Example

1) OUT[ENTRY] = ∅;
2) **for** (each basic block B other than ENTRY) OUT[B] = ∅;
3) **while** (changes to any OUT occur)
4)   **for** (each basic block B other than ENTRY) {
5)   IN[B] = $\bigcup_{P \text{ a predecessor of } B}$ OUT[P];
6)   OUT[B] = $gen_B \cup (IN[B] - kill_B)$;
   }

Iteration #3: fixed point!

OUT[ENTRY] = ∅

IN[B_1] = ∅

OUT[B_1] = {d_1, d_2, d_3}

IN[B_2] = {d_1, d_2, d_3, d_5, d_6, d_7}

OUT[B_2] = {d_3, d_4, d_5, d_6}

IN[B_3] = {d_3, d_4, d_5, d_6}

OUT[B_3] = {d_4, d_5, d_6}

IN[B_4] = {d_3, d_4, d_5, d_6}

OUT[B_4] = {d_3, d_5, d_6, d_7}

IN[EXIT] = {d_3, d_5, d_6, d_7}

OUT[EXIT] = {d_3, d_5, d_6, d_7}

ENTRY

d_1 : i = m−1 B_1
d_2 : j = n
d_3 : a = u1

gen_{B_1} = { d_1, d_2, d_3 }
$kill_{B_1}$ = { d_4, d_5, d_6, d_7 }

d_4 : i = i+1 B_2
d_5 : j = j−1

gen_{B_2} = { d_4, d_5 }
$kill_{B_2}$ = { d_1, d_2, d_7 }

d_6 : a = u2 B_3

gen_{B_3} = { d_6 }
$kill_{B_3}$ = { d_3 }

d_7 : i = u3 B_4

gen_{B_4} = { d_7 }
$kill_{B_4}$ = { d_1, d_4 }

EXIT

Outline

- Data-Flow Analysis

- **Classic Data-Flow Problems** ⎱
  Reaching Definitions

  Live Variables*

  Available Expressions*
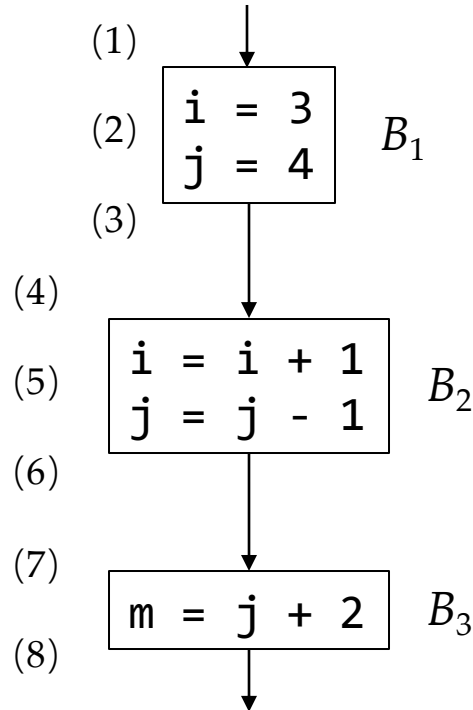
  * Optional self-study materials

Live-Variable Analysis

- Aim to know for variable x and point p whether the value of x at p could be used on some path starting at p

  - If yes, we say x is *live* at p; otherwise, x is *dead* at p.

- An important use of live-variable information is register allocation for basic blocks

  - It is not necessary to store a value computed in a register to memory at the end of a basic block

  - If all registers are used and we need another register, we should favor using a register with a dead value

- Another imporant use is to eliminate dead code

  - If a value is dead a a point, there is no need to compute it

Example

For the partial flow graph, we have:

(1)

(2) $\begin{array}{l} \texttt{i = 3} \\ \texttt{j = 4} \end{array}$ B_1

(3)

(4)

(5) $\begin{array}{l} \texttt{i = i + 1} \\ \texttt{j = j - 1} \end{array}$ B_2

(6)

(7)

$\texttt{m = j + 2}$ B_3

(8)

Point	i	j	m
(1)	Dead	Dead	Dead
(2)	Live	Dead	Dead
(3)	Live	Live	Dead
(4)	Live	Live	Dead
(5)	Dead	Live	Dead
(6)	Dead	Live	Dead
(7)	Dead	Live	Dead
(8)	Dead	Dead	Dead

Obsevation: Live-variable analysis a backward data-flow problem since the subsquent defs/uses determine the liveness of a variable at a point

Data-Flow Equations

```
i = i + 1;
j = j - 1;
```
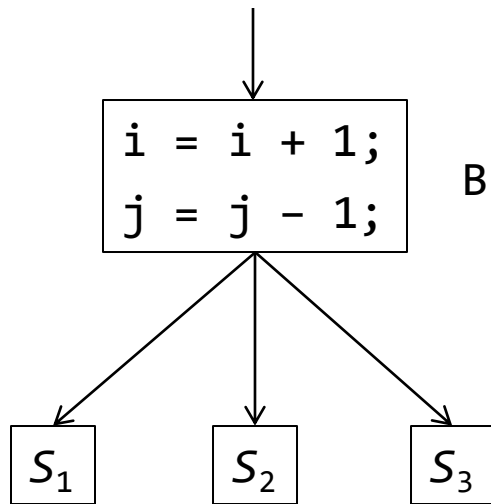B

S_1 S_2 S_3
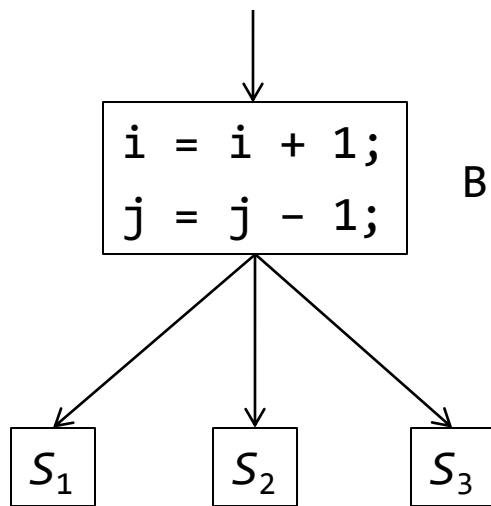
Data-flow values:

IN[B]: variables live at the entry of the block B

OUT[B]: variables live at the exit of block B

$$\text{OUT}[B] = \bigcup_{S \text{ is a successor of } B} \text{IN}[S]$$

Rationale: For safety, as long as a variable is live at the entry of any successor block of B, we consider the varible live at the exit of B

Data-Flow Equations

$$\text{IN}[B] = use_B \cup (\text{OUT}[B] - def_B)$$

```
i = i + 1;
j = j – 1;
```
B

S_1 S_2 S_3

use_B: the set of variables whose values are used in B prior to any definition of the variable (`i`, `j`)

def_B: the set of variables defined in B prior to any use of that variable in B (i.e., the first time the variable appears, it appears as a definition; def_B is $\emptyset$ for the example)

Computing use_B and def_B

- Transfer function for <u>a statement</u>:

  - $s: x = y + z$

  - $use_s = \{y, z\}$

  - $def_s = \{x\} - \{y, z\}$ // y, z might be x

- Transfer function for <u>a block of statements $s_1, s_2, \ldots, s_n$</u>:

  - $use_B = use_1 \cup (use_2 - def_1) \cup (use_3 - def_1 - def_2) \cup \cdots \cup$
    $(use_n - def_1 - def_1 - \cdots - def_{n-1})$

  - $def_B = def_1 \cup (def_2 - use_1) \cup (def_3 - use_1 - use_2) \cup \cdots \cup$
    $(def_n - use_1 - use_1 - \cdots - use_{n-1})$

Problem Formulation

- All variables are dead at the exit of a program

    - $\text{IN}[\text{EXIT}] = \emptyset$

- For all basic blocks B other than EXIT:

    - $\text{IN}[B] = use_B \cup (\text{OUT}[B] - def_B)$

    - $\text{OUT}[B] = \bigcup_{S \text{ is a successor of } B} \text{IN}[S]$

- Comparing to reaching definitions analysis:

    - Both sets of equations have union as the meet operator

    - Information flow for liveness travels "backward"

The Iterative Algorithm

- **Input:** A flow graph with def_B and use_B computed for each block B

- **Output:** IN$[B]$ and OUT$[B]$, the variables live at the entry and exit of each block B

$$\text{IN}[\text{EXIT}] = \emptyset;$$
$$\textbf{for } (\text{each basic block } B \text{ other than EXIT}) \text{ IN}[B] = \emptyset;$$
$$\textbf{while } (\text{changes to any IN occur})$$
$$\qquad \textbf{for } (\text{each basic block } B \text{ other than EXIT}) \{$$
$$\qquad\qquad \text{OUT}[B] = \bigcup_{S \text{ a successor of } B} \text{IN}[S];$$
$$\qquad\qquad \text{IN}[B] = use_B \cup (\text{OUT}[B] - def_B);$$
$$\qquad \}$$

Similar to the iterative algorithm for reaching definitions analysis, IN$[B]$ never shrinks and its size is bounded, so the algorithm will eventually halt.

Example
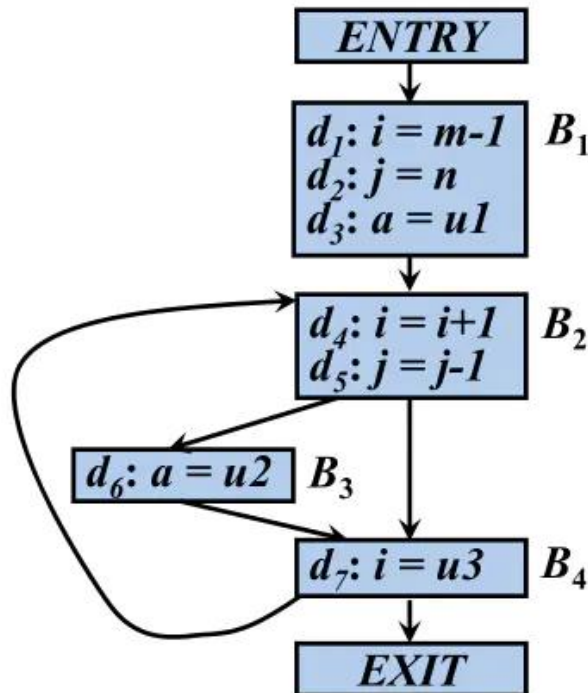
$$use_{B1} = \{\, m, n, u1\, \}$$

$$def_{B1} = \{\, i, j, a\, \}$$

$$use_{B2} = \{\, i, j\, \}$$

$$def_{B2} = \Phi$$

$$use_{B3} = \{\, u2\, \}$$

$$def_{B3} = \{\, a\, \}$$

$$use_{B4} = \{\, u3\, \}$$

$$def_{B4} = \{\, i\, \}$$

https://www.jianshu.com/p/ebc1c72b881c

Example

$use_{B1} = \{ m, n, u1 \}$
$def_{B1} = \{ i, j, a \}$
$use_{B2} = \{ i, j \}$
$def_{B2} = \Phi$
$use_{B3} = \{ u2 \}$
$def_{B3} = \{ a \}$
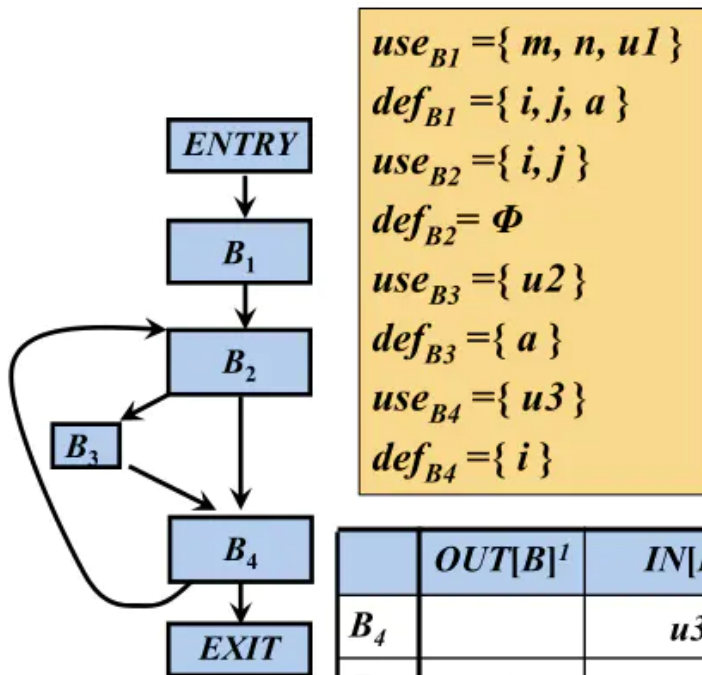$use_{B4} = \{ u3 \}$
$def_{B4} = \{ i \}$

OUT[B]	B_1	B_2	B_3	B_4
a	×	×	×	×
i	√	×	×	√
j	√	√	√	√
m	×	×	×	×
n	×	×	×	×
u_1	×	×	×	×
u_2	√	√	√	√
u_3	√	√	√	√

	$OUT[B]^1$	$IN[B]^1$	$OUT[B]^2$	$IN[B]^2$	$OUT[B]^3$	$IN[B]^3$
B_4		u3	i,j,u2,u3	j,u2,u3	i,j,u2,u3	j,u2,u3
B_3	u3	u2,u3	j,u2,u3	j,u2,u3	j,u2,u3	j,u2,u3
B_2	u2,u3	i,j,u2,u3	j,u2,u3	i,j,u2,u3	j,u2,u3	i,j,u2,u3
B_1	i,j,u2,u3	m,n,u1,u2,u3	i,j,u2,u3	m,n,u1,u2,u3	i,j,u2,u3	m,n,u1,u2,u3

Outline

- Data-Flow Analysis

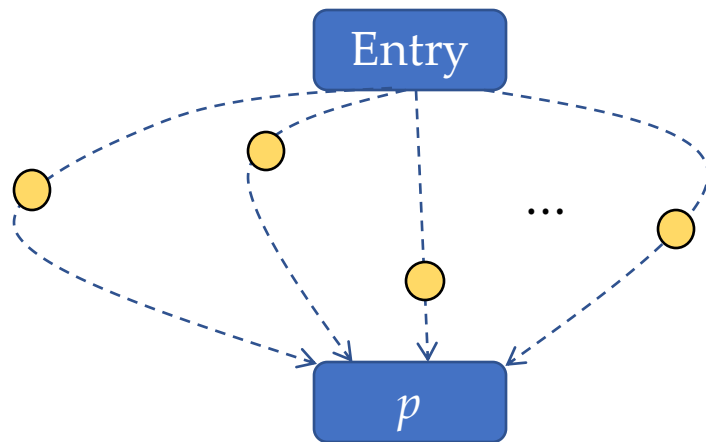- Classic Data-Flow Problems ⎰ Reaching Definitions

  Live Variables*

  Available Expressions*

  * Optional self-study materials

Available Expressions

- An expression $x + y$ is available at a point p if every path from the entry node to p evaluates $x + y$, and after the last such evaluation prior to reaching p, there are no subsequent assignments to x or y
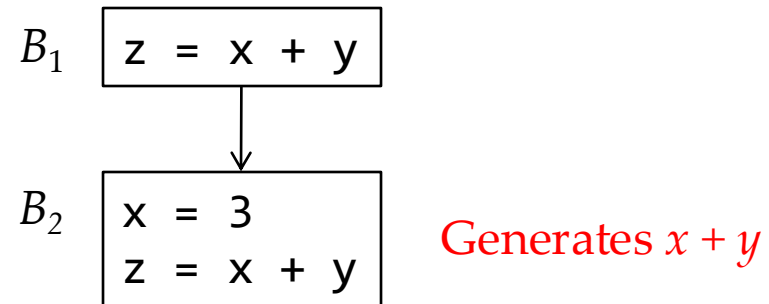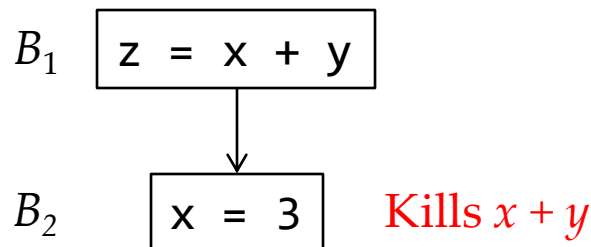

○ Point evaluating $x + y$

No assignments to x or y afterwards
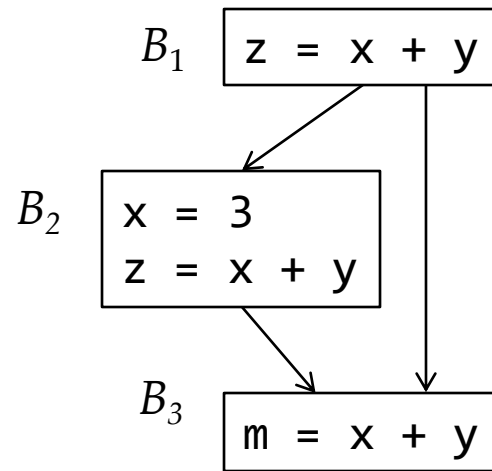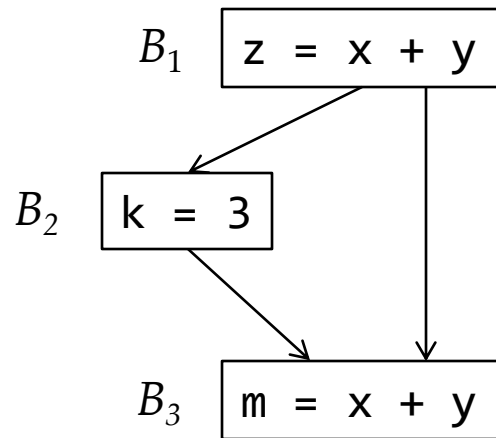
Killing / Genearting Expressions

- A block *kills* expression $x + y$ if:

  - It assigns x or y and does not subsequently recompute $x + y$

- A block *generates* expression $x + y$ if:

  - It evaluates $x + y$ and does not subsequently define x or y

B_1 | z = x + y

B_2 | x = 3   Kills $x + y$

B_1 | z = x + y

B_2 | x = 3
       z = x + y   Generates $x + y$

The Primary Use of Available Expressions

- Detecting global common expressions

Case 1: B_2 does not redefine x or y

Case 2: B_2 redefines x and recomputes $x + y$

In both cases, $x + y$ needs not to be recomputed in B_3

Computing Genereated Expressions

S: the set of expressions available before the statement

```
x = y + z
```

Compute the expressions available after the statement:

- Add to *S* the expression of $y + z$

- Delete from *S* any expression involving variable *x*

If we repeat the steps for each statement in a block *B*, we will get the set of generated expressions for *B* after we process the last statement.

Computing Killed Expressions

- For a basic block *B*, the set of *killed expressions* is all expressions, say $y + z$, such that:

    - *y* or *z* is defined in *B*

    - AND $y + z$ is not generated by *B*

Example

Statement	Available Expressions
	$\emptyset$
a = b + c	
	$\{b + c\}$
b = a - d	
	$\{a - d\}$
c = b + c	
	$\{a - d\}$
d = a – d	
	$\emptyset$

Data-Flow Equations

- At the exit of the entry node, no expressions are available:

    - OUT[ENTRY] = $\emptyset$

- For all basic blocks B other than ENTRY:

    - OUT$[B]$ = $e\_genB \cup (\text{IN}[B] - e\_killB)$

    - IN$[B]$ = $\bigcap_{P \text{ a predecessor of } B} \text{OUT}[P]$

The equations are similar to those for reaching definition analysis except that the meet opeator is intersection rather than union.[*]

[*] This is for safety reasons. Although producing a subset of the exact set of available expressions prevents certain optimizations, this does not lead to the change of program semantics

The Iterative Algorithm

- **Input:** A flow graph with *e_killB* and *e_genB* computed for each block *B*

- **Output:** IN[*B*] and OUT[*B*], the expressions available at the entry and exit of each block *B*

U: the universal set of all expressions

$$\text{OUT}[\text{ENTRY}] = \emptyset;$$
for (each basic block B other than ENTRY) $\text{OUT}[B] = U;$
while (changes to any OUT occur)
  for (each basic block B other than ENTRY) {
    $\text{IN}[B] = \bigcap_{P \text{ a predecessor of } B} \text{OUT}[P];$
    $\text{OUT}[B] = e\_gen_B \cup (\text{IN}[B] - e\_kill_B);$
  }

Summary of The Problems

	Reaching Definitions	Live Variables	Available Expressions
Domain	Sets of definitions	Sets of variables	Sets of expressions
Direction	Forwards	Backwards	Forwards
Transfer function	$gen_B \cup (x - kill_B)$	$use_B \cup (x - def_B)$	$e\_gen_B \cup (x - e\_kill_B)$
Boundary	$\text{OUT}[\text{ENTRY}] = \emptyset$	$\text{IN}[\text{EXIT}] = \emptyset$	$\text{OUT}[\text{ENTRY}] = \emptyset$
Meet ($\wedge$)	$\cup$	$\cup$	$\cap$
Equations	$\text{OUT}[B] = f_B(\text{IN}[B])$ $$\text{IN}[B] = \bigwedge_{P,\,pred(B)} \text{OUT}[P]$$	$\text{IN}[B] = f_B(\text{OUT}[B])$ $$\text{OUT}[B] = \bigwedge_{S,\,succ(B)} \text{IN}[S]$$	$\text{OUT}[B] = f_B(\text{IN}[B])$ $$\text{IN}[B] = \bigwedge_{P,\,pred(B)} \text{OUT}[P]$$
Intialize	$\text{OUT}[B] = \emptyset$	$\text{IN}[B] = \emptyset$	$\text{OUT}[B] = U$
Application	Constant folding	Dead code elimination	Global common subexpression elimination

Reading Tasks

- Chapter 9 of the dragon book
  - 9.1 The Principal Sources of Optimization (9.1.1-9.1.7)
  - 9.2 Introduction to Data-Flow Analysis (9.2.1-9.2.6)