Compilerconstructie

najaar 2014

http://www.liacs.nl/home/rvvliet/coco/

Rudy van Vliet
kamer 124 Snellius, tel. 071-527 5777
rvvliet(at)liacs(dot)nl

college 9, dinsdag 25 november 2014
\+ 'werkcollege'

Code Optimization

9.2 Introduction to Data-Flow Analysis

- Optimizations depend on data-flow analysis, e.g.,
  - Global common subexpression elimination
  - Dead-code elimination

- Execution path yields program state at program point

- Extract information from program state for data-flow analysis

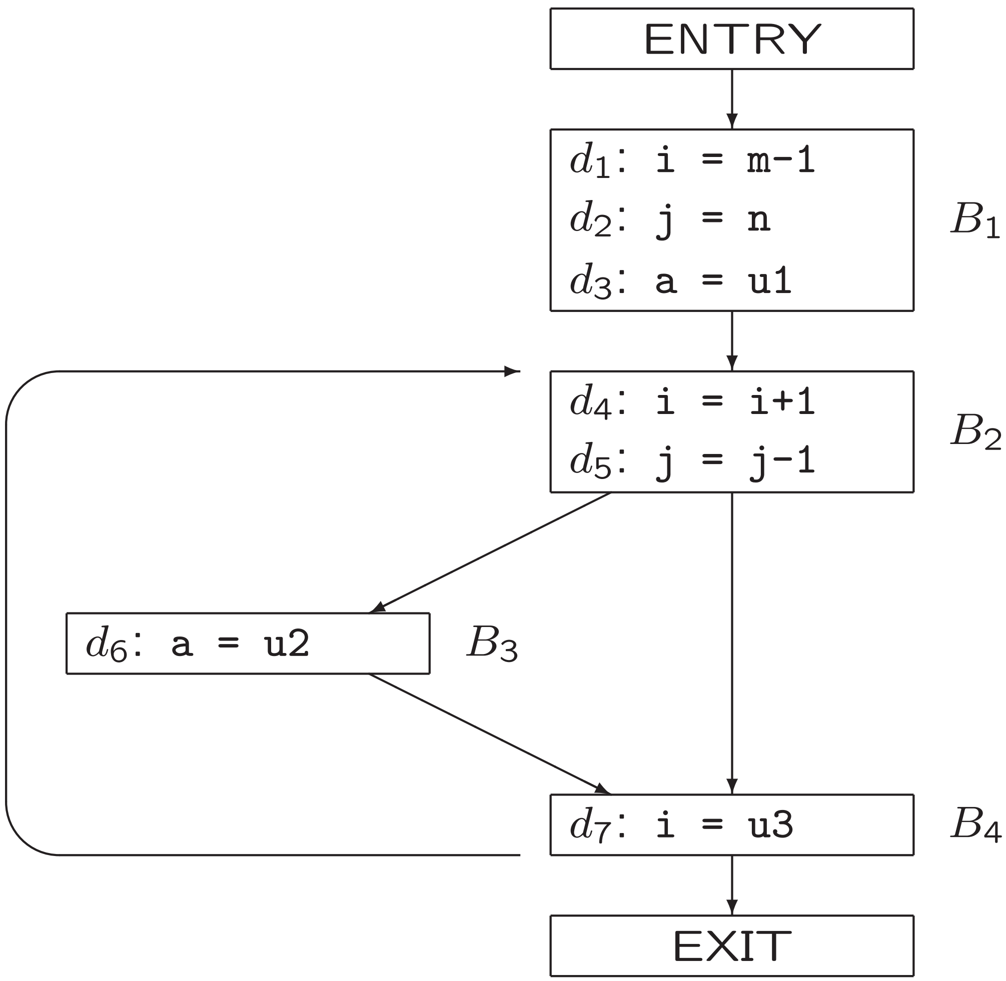
- Usually infinite number of execution paths / program states

- Different analyses extract different information

Data-Flow Analysis (Examples)

- Reaching definitions: which definitions (assignments of values) of variable x may reach program point?
  - Useful for debugging:
    May variable x be undefined?
  - Useful for constant folding:
    Can variable x only have one constant value at program point?

9.2.4 Computing Reaching Definitions



Reaching definitions

- Before B_1: $\emptyset$
- After B_1: $\{d_1, d_2, d_3\}$
- Before B_2: ...

9.2.2 The Data Flow Analysis Schema

Data flow values

- IN[s]: before statement s
- OUT[s]: after statement s
- Transfer function f_s
  - forward: OUT[s] $= f_s$(IN[s])
  - backward: IN[s] $= f_s$(OUT[s])

Computing Reaching Definitions

- Effect of single definition $d : u = v$ *op* w:
  - $\text{OUT}[d] = \{d\} \cup (\text{IN}[d] - \ldots)$

Computing Reaching Definitions

Effect of single definition $d : u = v \textit{ op } w$:

- $\text{OUT}[d] = \{d\} \cup (\text{IN}[d] - \{\text{all other definitions of } u \text{ in program}\})$

- Hence,

$$
\begin{aligned}
f_d(x) &= \{d\} \cup (x - \{\text{all other definitions of } u \text{ in program}\}) \\
&= \textit{gen}_d \cup (x - \textit{kill}_d)
\end{aligned}
$$

where

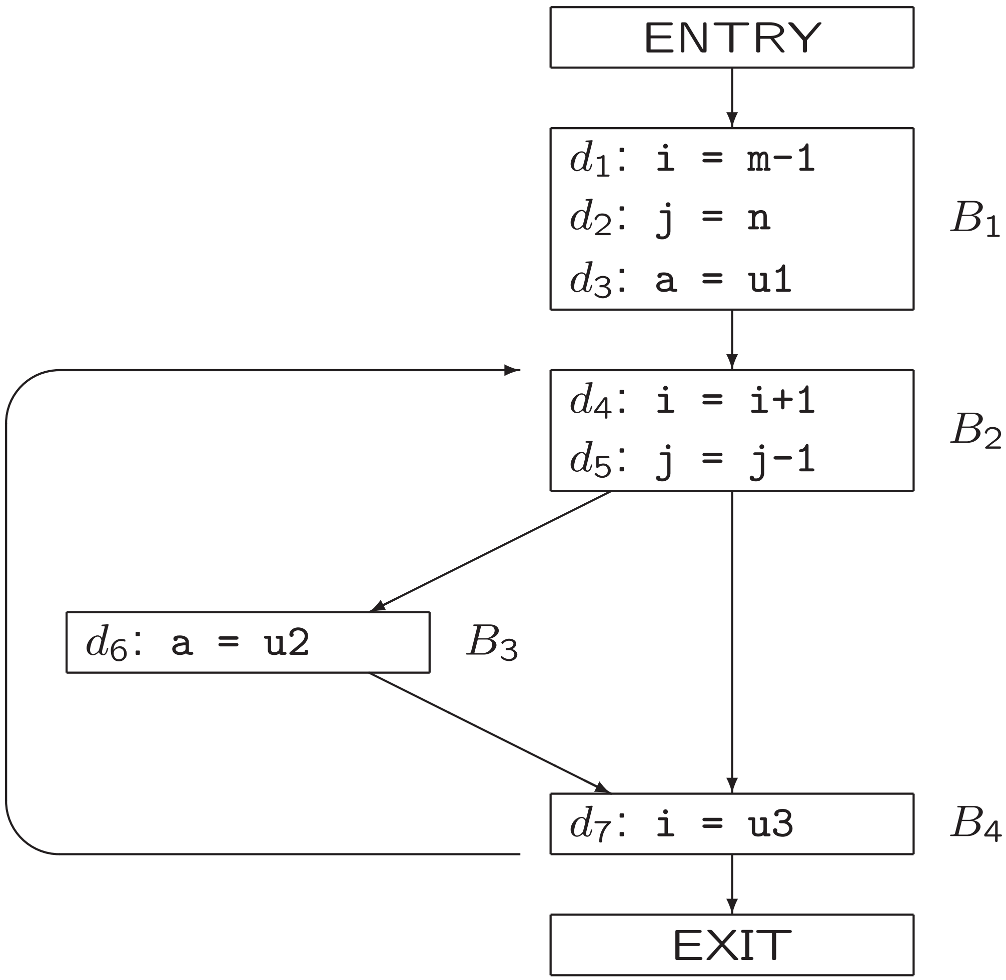
$$
\begin{aligned}
\textit{gen}_d &= \{d\} \\
\textit{kill}_d &= \{\text{all other definitions of } u \text{ in program}\}
\end{aligned}
$$

Computing Reaching Definitions

Effect of block B, with definitions $1, 2, \ldots, n$:

$$\begin{aligned} \mathit{gen}_B &= \{n, n-1, \ldots, 1\} - \{ \text{ definitions killed afterwards } \} \\ &= \mathit{gen}_n \cup (\mathit{gen}_{n-1} - \mathit{kill}_n) \cup (\mathit{gen}_{n-2} - \mathit{kill}_{n-1} - \mathit{kill}_n) \ldots \\ \mathit{kill}_B &= \mathit{kill}_1 \cup \mathit{kill}_2 \cup \ldots \cup \mathit{kill}_n \end{aligned}$$

Computing Reaching Definitions



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

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

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

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

Iterative Algorithm for Computing Reaching Definitions

OUT[ENTRY] $= \emptyset$
for each basic block B other than ENTRY
    OUT[B] $= \emptyset$

while (changes to any OUT occur)
    for each basic block B other than ENTRY
    $\{$ IN[B] $= \cup_{\text{predecessors } P \text{ of } B}$ OUT[P]

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

Typical form of algorithm for forward data-flow analysis

$\cup$ is meet operator

Example with $B = B_1, B_2, B_3, B_4,$ EXIT...

Implementation of Iterative Algorithm for Computing Reaching Definitions

With bit vectors

Block B	OUT$[B]^0$	IN$[B]^1$	OUT$[B]^1$	IN$[B]^2$	OUT$[B]^2$
B_1	000 0000	000 0000	111 0000	000 0000	111 0000
B_2	000 0000	111 0000	001 1100	111 0111	001 1110
B_3	000 0000	001 1100	000 1110	001 1110	000 1110
B_4	000 0000	001 1110	001 0111	001 1110	001 0111
EXIT	000 0000	000 0000	001 0111	001 0111	001 0111

9.2.5 Live-Variable Analysis

- Variable x is live at program point p,
  if value of x at p could be used later along *some* path

- Otherwise x is dead at p

- Information useful for register allocation (see lecture 7)

- Information about later use must be propagated backwards

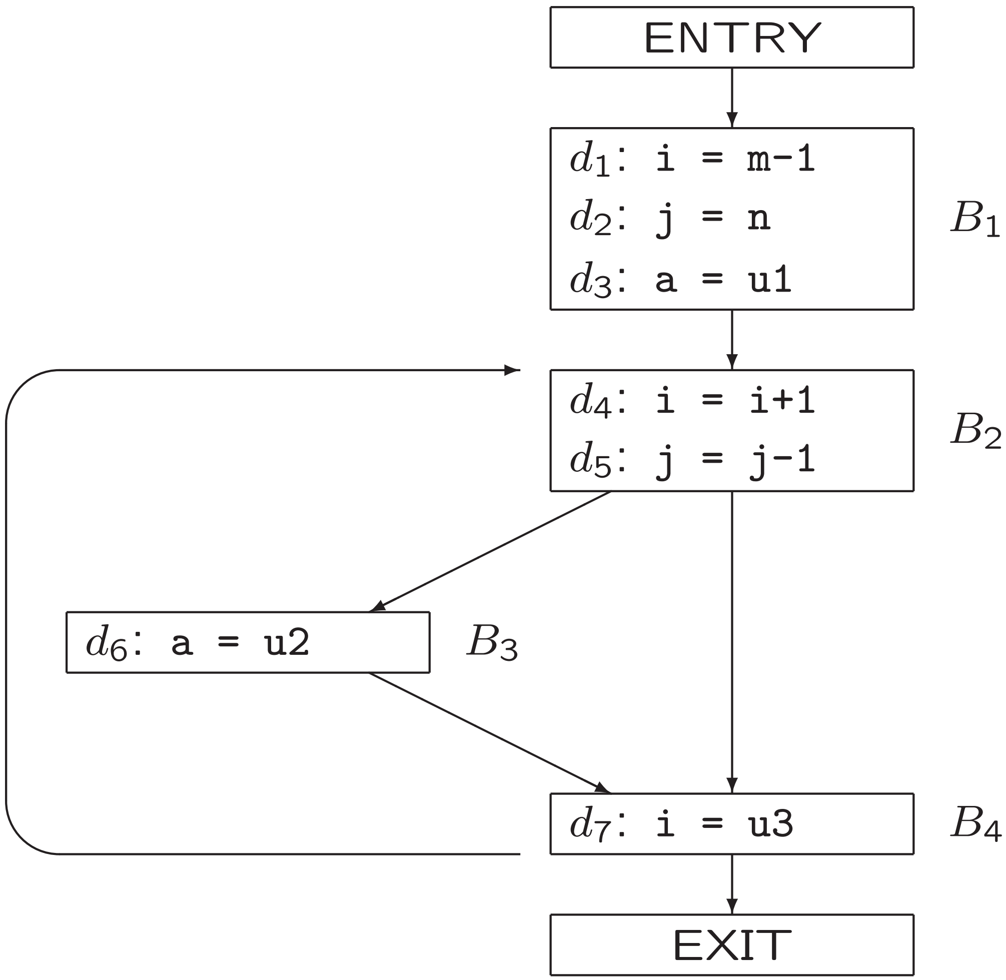
Live-Variable Analysis

Effect of block B on live variables

- *use*$_B$:
  variables that may be *used* in B prior to any definition in B
  ($\approx$ *gen*)

- *def*$_B$:
  variables *defined* in B prior to any use of that variable in B
  ($\approx$ *kill*)

Computing Live Variables



$def_{B_1} = \{i, j, a\}$

$use_{B_1} = \{m, n, u1\}$

$def_{B_2} = \emptyset$

$use_{B_2} = \{i, j\}$

$def_{B_3} = \{a\}$

$use_{B_3} = \{u2\}$

$def_{B_4} = \{i\}$

$use_{B_4} = \{u3\}$

Iterative Algorithm to Compute Live Variables

IN[EXIT] $= \emptyset$
for each basic block B other than EXIT
    IN[B] $= \emptyset$

while (changes to any IN occur)
    for each basic block B other than EXIT
    $\{$ OUT[B] $= \cup_{\text{successors } S \text{ of } B}$ IN[S]

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

Typical form of algorithm for backward data-flow analysis

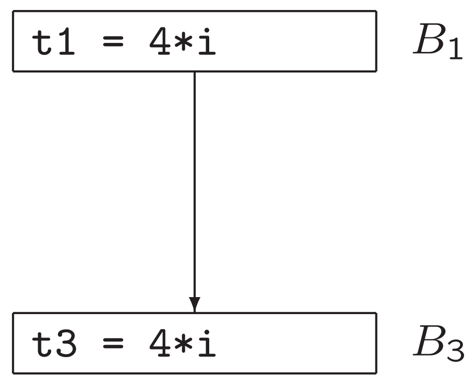
9.2.6 Available expressions

- Is (value of) expression x *op* y available?

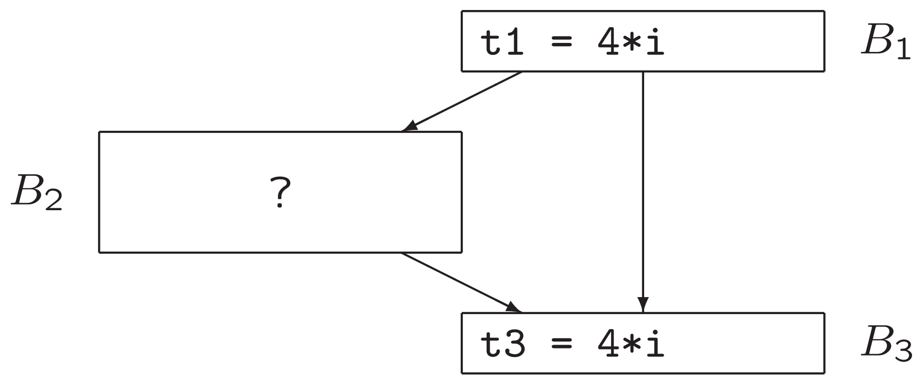
- Useful for global common subexpression elimination

- Can be decided with data-flow analysis

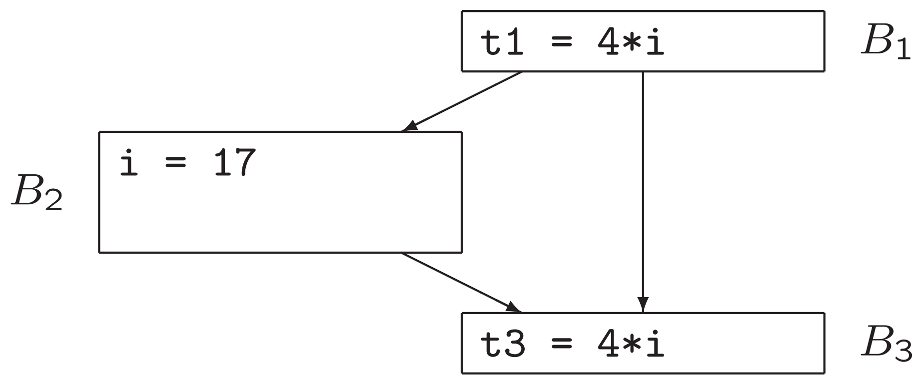
Available Expressions (Example)



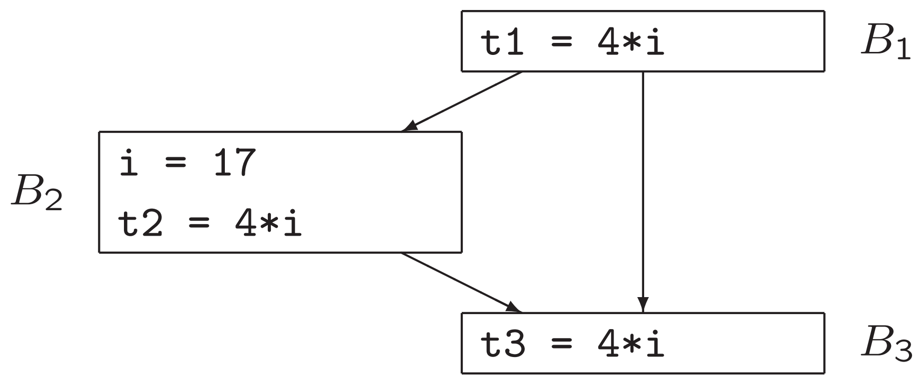
Available Expressions (Example)



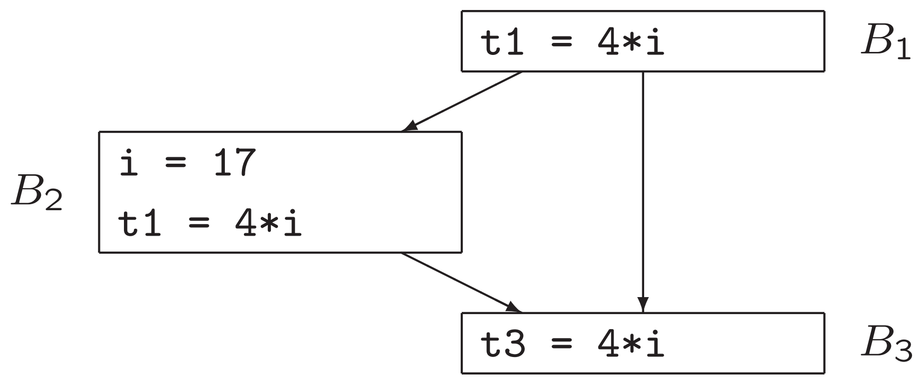
Available Expressions (Example)



Available Expressions (Example)



Available Expressions (Example)



Computing Available Expressions

Effect of block B on available expressions

- *e_gen*$_B$:
  expressions y *op* z that are computed in B,
  and for which y and z are not subsequently redefined

- *e_kill*$_B$:
  expressions y *op* z for which y and/or z are defined in B,
  and that are not subsequently recomputed

Computing e_gen_B (Example)

$S = \emptyset$
For each statement $x = y$ *op* z in block B (forwards)

- add y *op* z to S
- delete from S any expression involving x

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

Computing Available Expressions

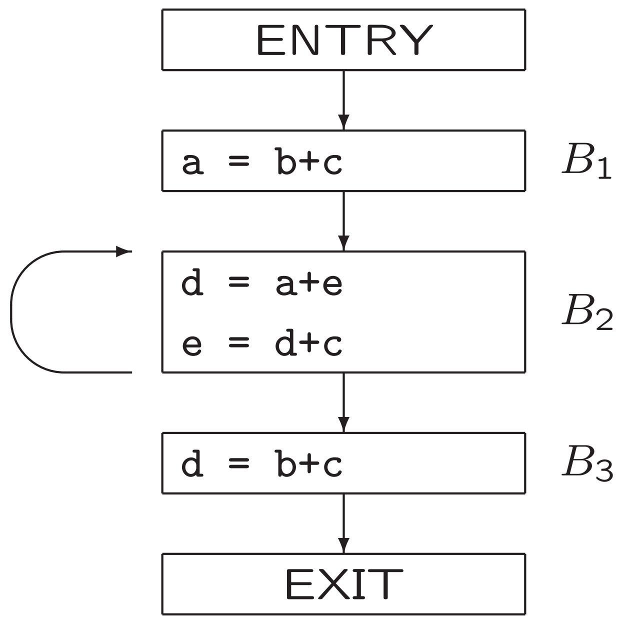
OUT[ENTRY] $= \emptyset$
for each basic block B other than ENTRY
    OUT[B] $= U$

while (changes to any OUT occur)
    for each basic block B other than ENTRY
    $\{$ IN[B] $= \cap_{\text{predecessors } P \text{ of } B}$ OUT[P]

        OUT[B] $=$ *e_gen*$_B \cup$ (IN[B] $-$ *e_kill*$_B$)
    $\}$

Why U . . .

Available Expressions (Example)



Efficient Iterative Data-Flow Analysis

Example: computing reaching definitions

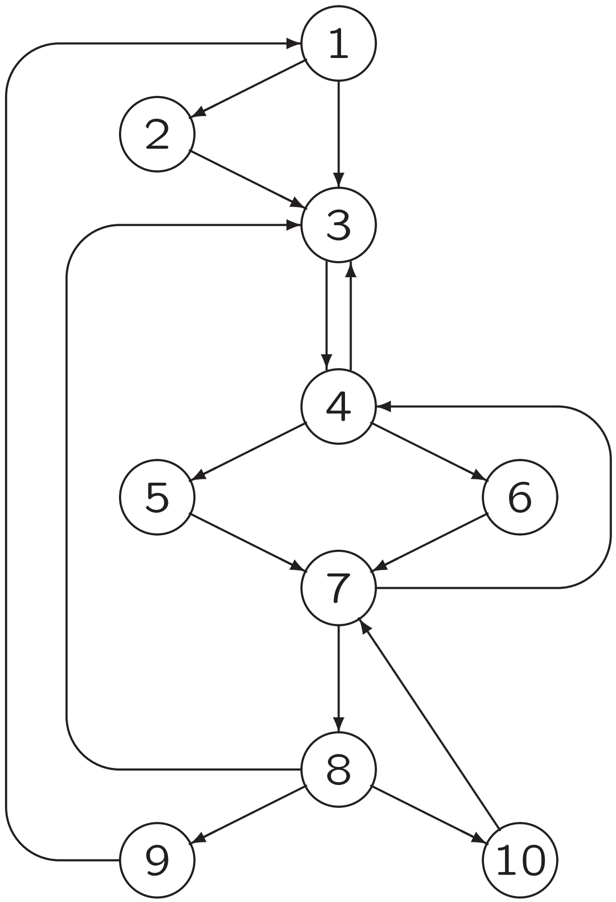
OUT[ENTRY] $= \emptyset$
for each basic block B other than ENTRY
    OUT[B] $= \emptyset$

while (changes to any OUT occur)
    for each basic block B other than ENTRY
    $\{$ IN[B] $= \cup_{\text{predecessors } P \text{ of } B}$ OUT[P]

        OUT[B] $=$ *gen*$_B \cup$ (IN[B] $-$ *kill*$_B$)
    $\}$

Order of blocks in second for-loop matters

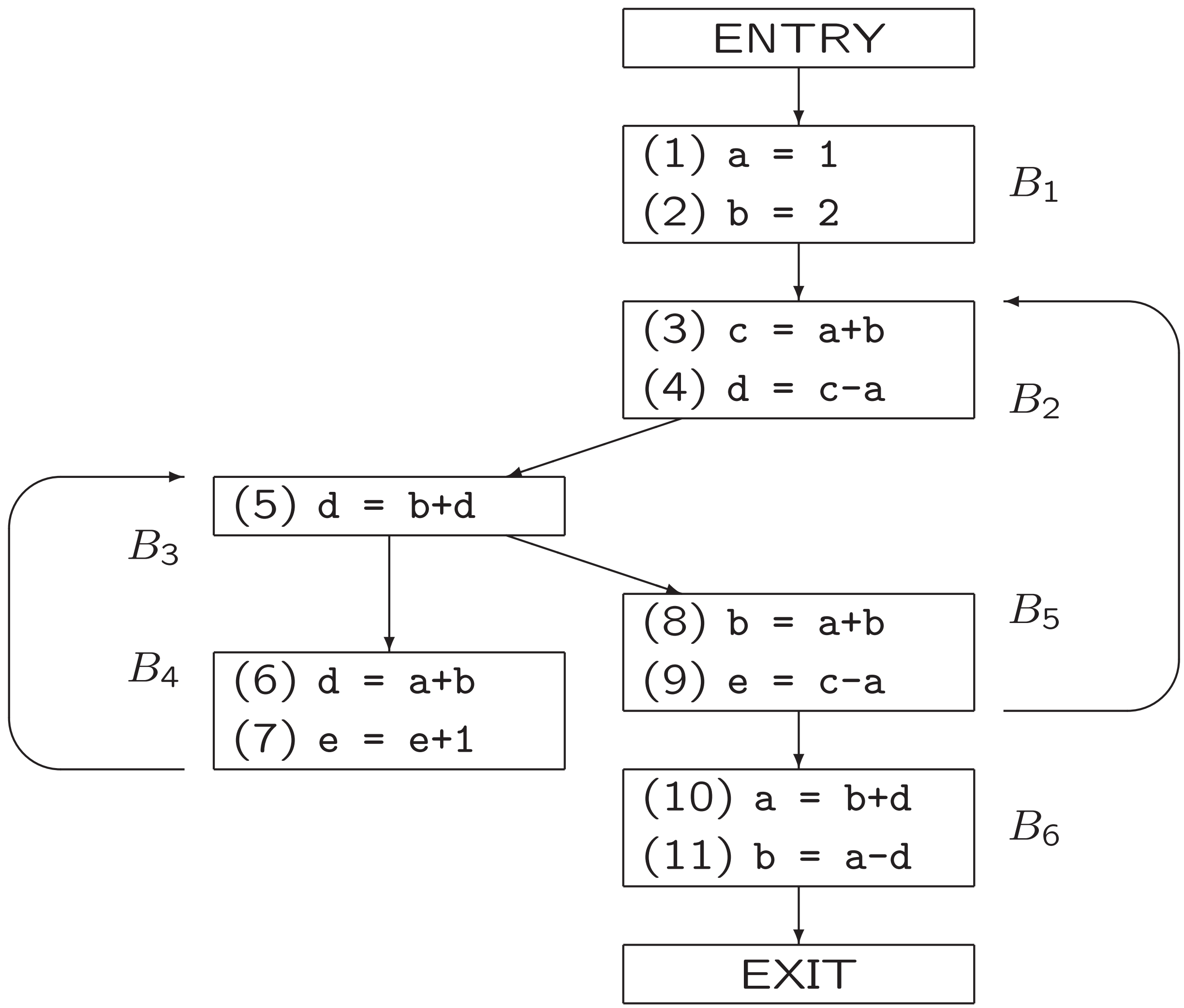
Efficient Iterative Data-Flow Analysis



Order of blocks in second for-loop matters

Exercises

Flow Graph For Data Flow Analysis



9.6 Loops in Flow Graphs

- Optimizations of loops have significant impact

- Loops affect speed of convergence of iterative DFA

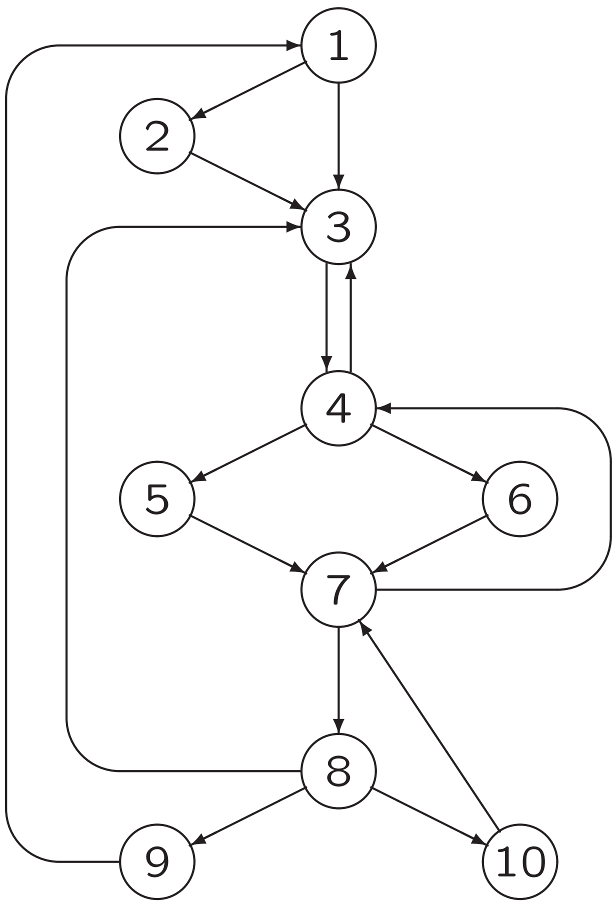
- Essential to identify loops

- Used in region based analysis (not for exam)

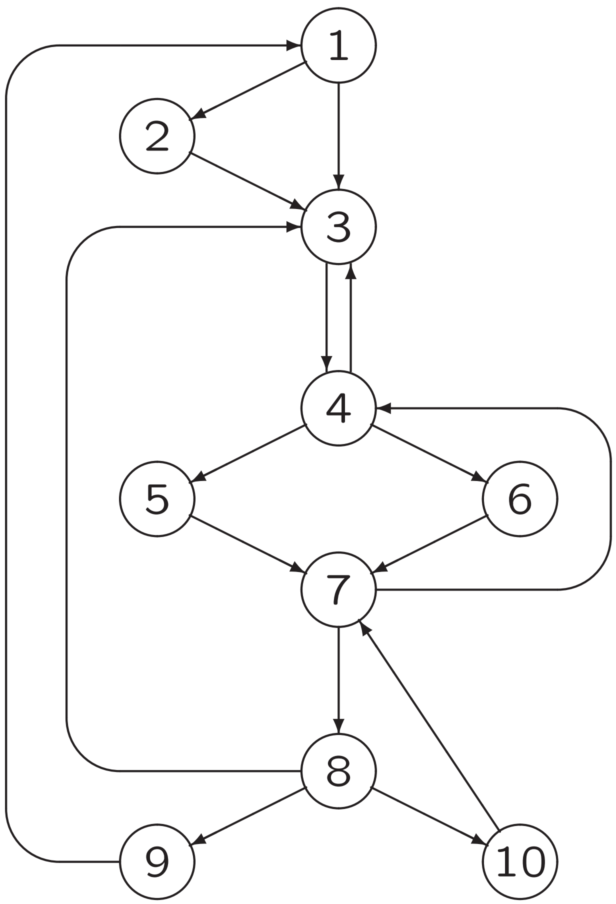
9.6.1 Dominators

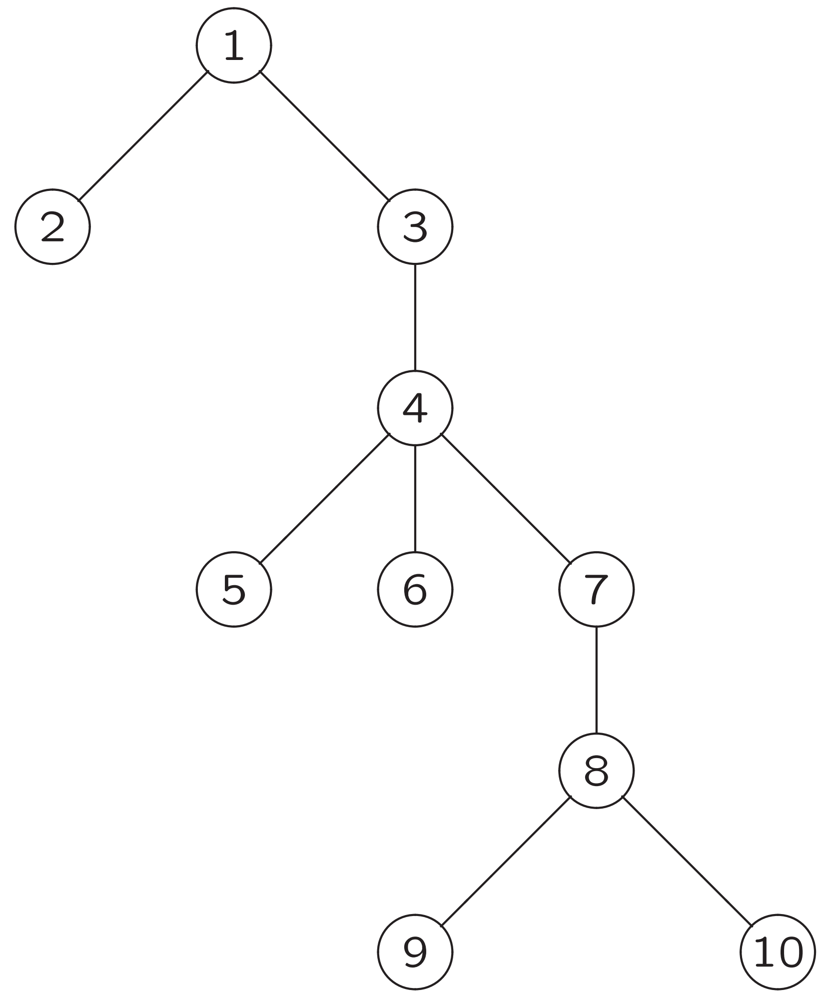
- Dominators:
  - Node d dominates node n if every path from ENTRY node to n goes through d: d *dom* n
  - Node n dominates itself
  - Loop entry dominates all nodes in loop
- Immediate dominator m of n:
  last dominator on (any) path from ENTRY node to n
  - if $d \neq n$ and d *dom* n, then d *dom* m

Dominators (Example)



Dominator Trees (Example)





Finding Dominators

Forward data-flow analysis

N is set of all nodes

OUT[ENTRY] $= \{$ENTRY$\}$
for each node n other than ENTRY
    OUT[n] $= N$

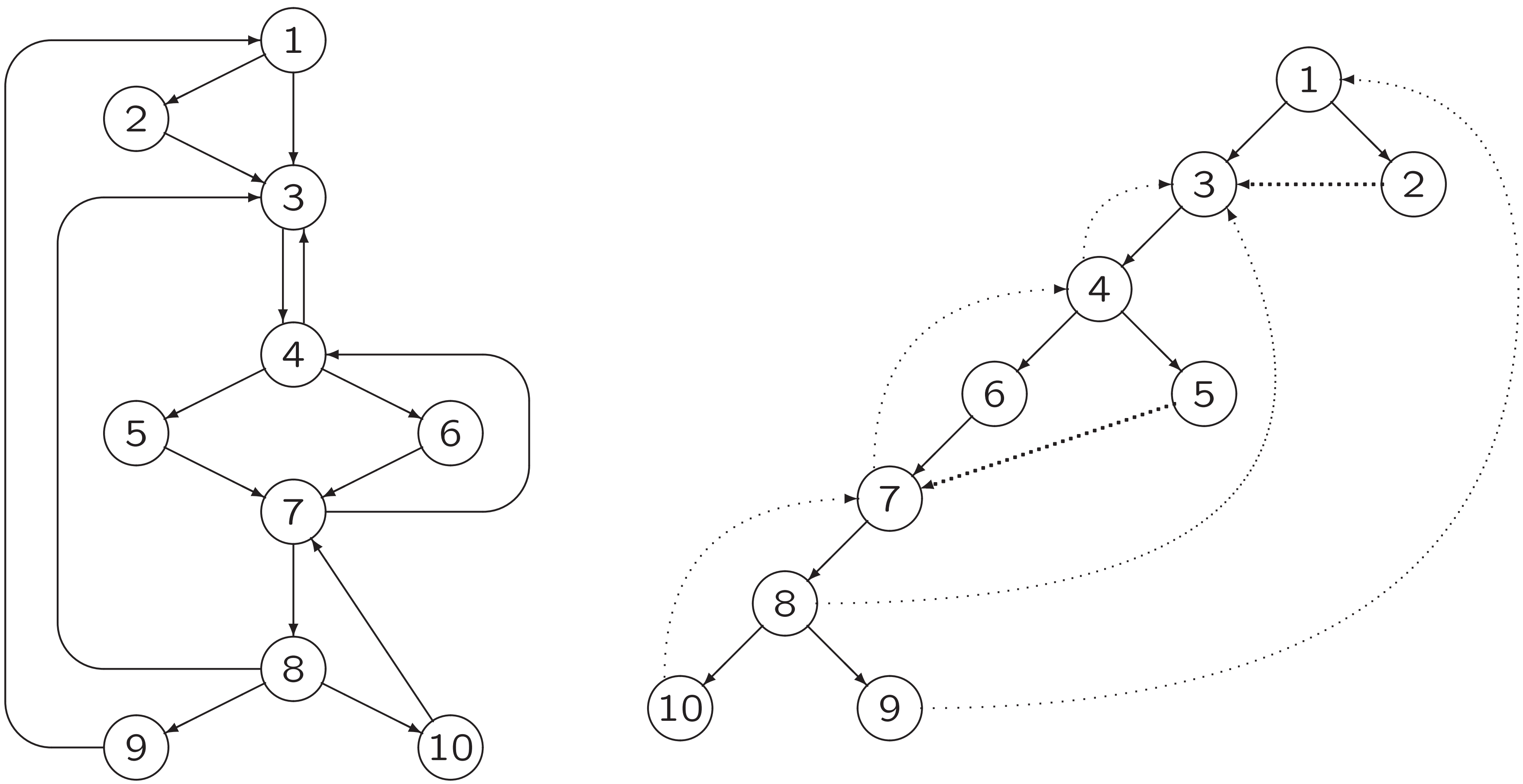
while (changes to any OUT occur)
    for each node n other than ENTRY
    $\{$ IN[n] $= \cap_{\text{predecessors } m \text{ of } n}$ OUT[m]

        OUT[n] $=$ IN[n] $\cup \{n\}$
    $\}$

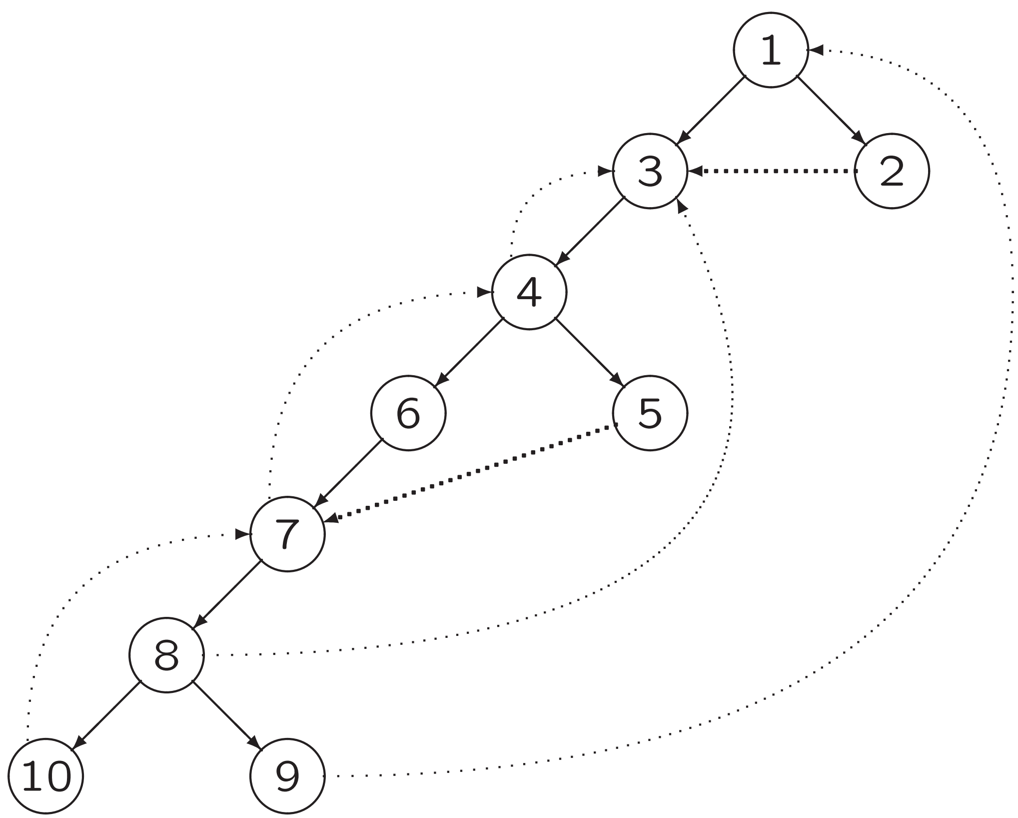
9.6.2 Depth-First Ordering

- Depth-first traversal of graph
  - Start from entry node
  - Recursively visit neighbours (in any order)
  - Hence, visit nodes far away from the entry node as quickly as it can (DF)

A Depth-First Spanning Tree



9.6.3 Edges in Depth-First Spanning Tree

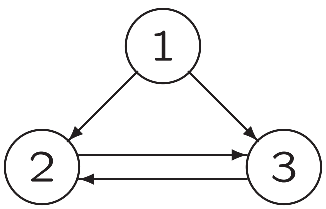


- Advancing edges
- Retreating edges
- Cross edges
- Back edge $a \to b$, if b dominates a (regardless of DFST)
- Each back edge is retreating edge in DFST
- Flow graph is reducible, if each retreating edge in any DFST is back edge

9.6.4 Reducibility

- In practice, almost every flow graph is reducible

- Example of nonreducible flow graph
  (with advancing edges)



- To decide on reducibility:

  1. Remove back edges

  2. Is remaining graph acyclic?

9.6.6 Natural loops

- If loop has single-entry node, then compiler can assume certain initial conditions

- Natural loop

  1. Has single-entry node: header

  2. Has back edge to header

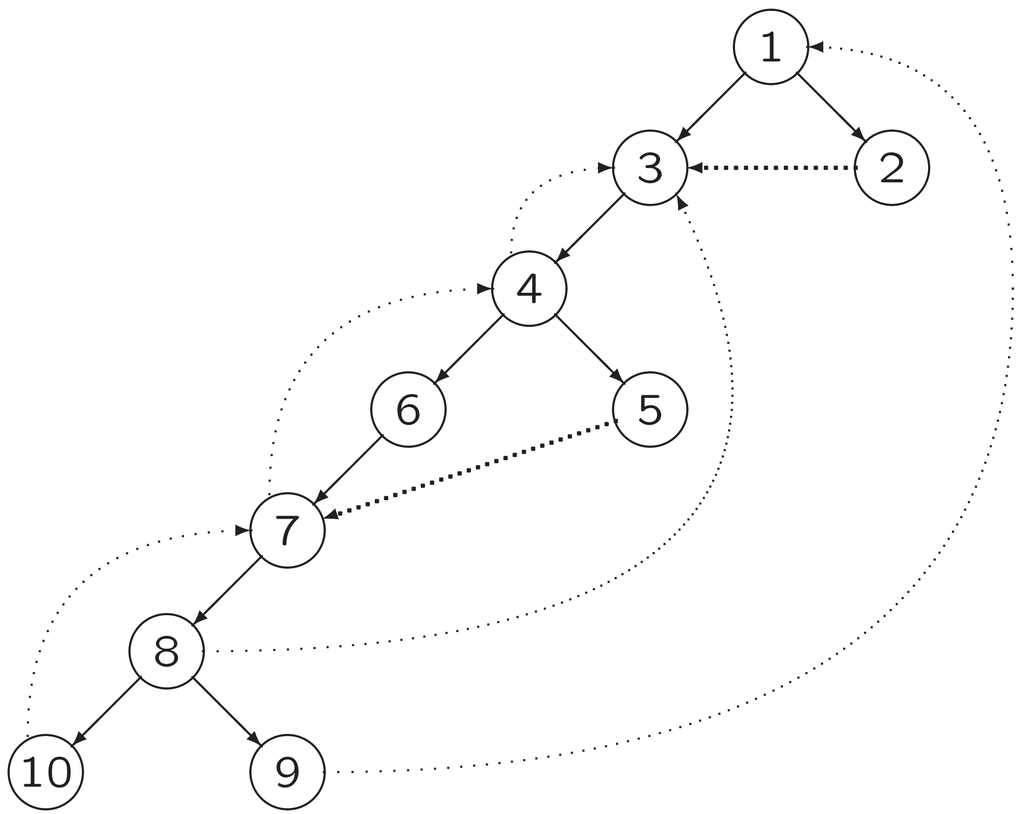
- Each back edge $n \to d$ determines natural loop, consisting of

  – d

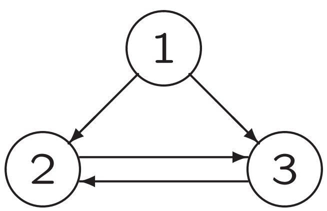
  – all nodes that can reach n without going through d

- Constructing natural loop of back edge...

Natural Loops (Example)



No Natural Loops



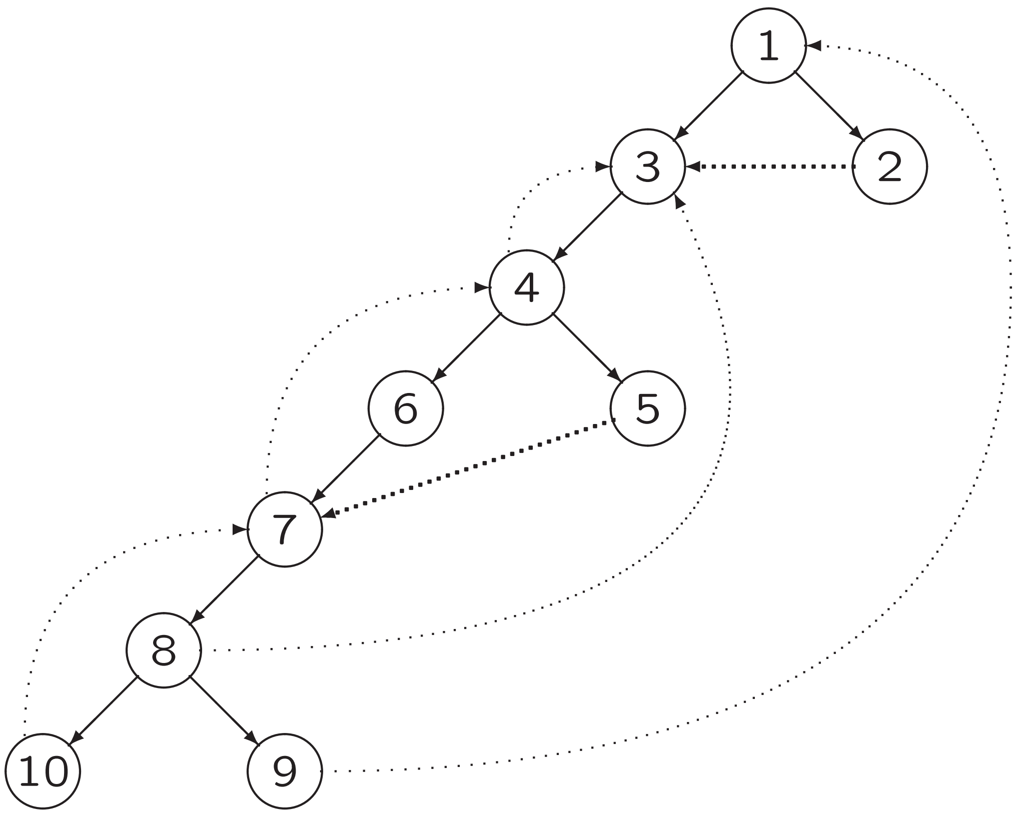
Natural Loops

- Useful property: unless two natural loops have same header
  - either they are disjoint
  - or one is nested within other

  Allows for inside-out optimization

- Assumption: if necessary, combine natural loops with same header. . .

9.6.2 A Depth-First Ordering



- Depth-First Ordering: nodes in DFST in WRL order ≈ reverse of postorder
- Example: 1,2,3,4,5,6,7,8,9,10
- Edge $m \rightarrow n$ is retreating, if and only if n comes before m in depth-first ordering

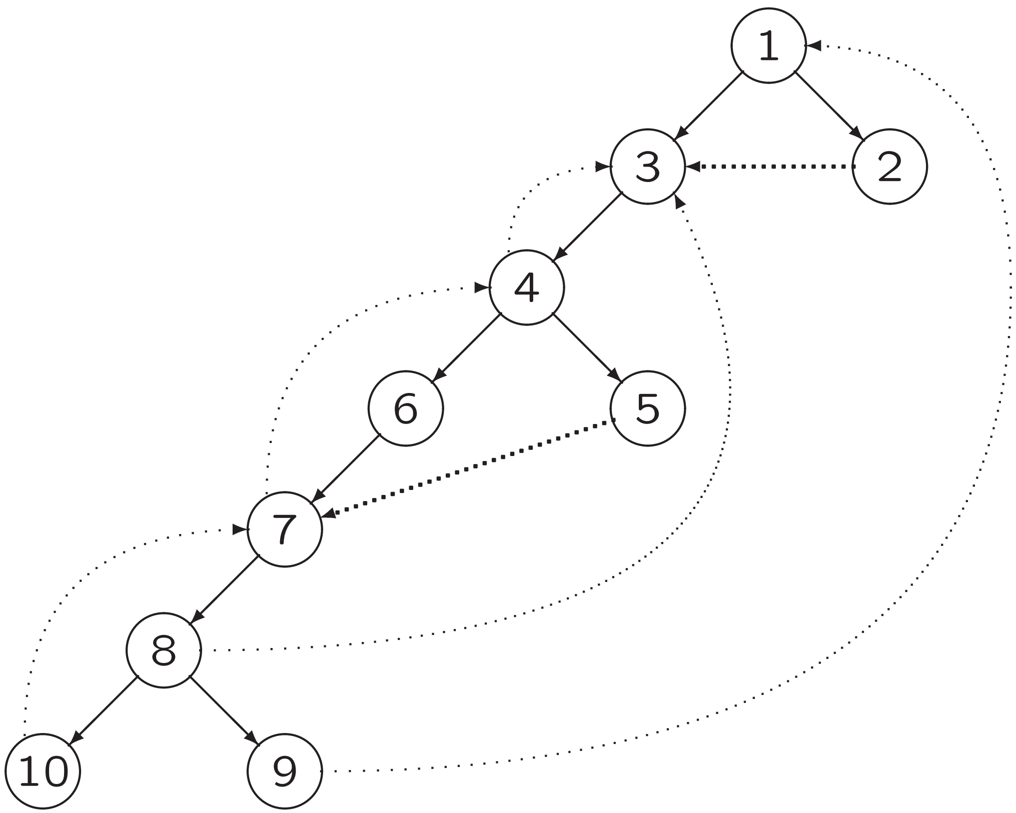
9.6.5 Depth of Flow Graph

- Depth of DFST is largest number of retreating edges on any cycle-free path

- If flow graph is reducible, then depth is independent of DFST: depth of flow graph

- Depth $\leq$ depth of loop nesting in flow graph

Depth of Flow Graph (Example)



Depth is 3, because of path
$10 \to 7 \to 4 \to 3$

9.6.7 Speed of Convergence of Iterative Data-Flow Algorithms

In data-flow analysis, can significant events be propagated to node along acyclic path?

- Yes for
  - Reaching definitions
  - Live-variable analysis
  - Available expressions
- No for
  - Copy propagation

If yes, then fast convergence possible

Efficient Iterative Data-Flow Analysis

Example: computing reaching definitions

```
OUT[ENTRY] = ∅
for each basic block B other than ENTRY
    OUT[B] = ∅

while (changes to any OUT occur)
    for each basic block B other than ENTRY
    {   IN[B] = ∪_{predecessors P of B} OUT[P]

        OUT[B] = gen_B ∪ (IN[B] − kill_B)
    }
```

Order of blocks in second for-loop matters

Fast Convergence

- Forward data-flow problem: visit nodes in depth-first-order
- Recall: edge $m \to n$ is retreating, if and only if n comes before m in depth-first ordering
- Example: path of propagation of definition d:

$$3 \to 5 \to 19 \to 35 \to 16 \to 23 \to 45 \to 4 \to 10 \to 17$$

- Number of iterations: $1 +$ depth $(+\ 1)$
- Typical flow graphs have depth 2.75
- Backward data-flow problem: visit nodes in reverse of depth-first-order

En verder. . .

- Dinsdag 2 december: practicum over opdracht 4
- Maandag 8 december: inleveren opdracht 4
- Maandag 15 december, 14:00–17:00: tentamen
- Vragenuur ?

Compiler constructie

college 9
Code Optimization

Chapters for reading:
9.2, 9.6