Compilerconstructie

najaar 2012

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

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

college 8, dinsdag 13 november 2012

Code Optimization

9.1 The Principal Sources of Optimization

Causes of redundancy

- At source level
- Side effect of high-level programming language, e.g., $A[i][j]$

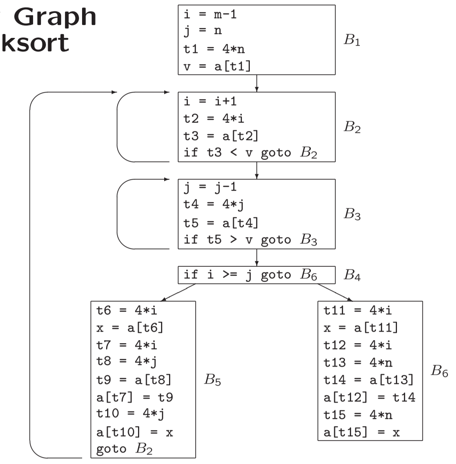
A Running Example: Quicksort

```
void quicksort (int m, int n)
    /* recursively sorts a[m] through a[n] */
{
    int i, j;
    int v, x;

    if (n <= m) return;

    i = m-1; j = n; v = a[n];
    while (1)
    {   do i = i+1; while (a[i] < v);
        do j = j-1; while (a[j] > v);
        if (i >= j) break;
        x = a[i]; a[i] = a[j]; a[j] = x; /* swap a[i], a[j] */
    }
    x = a[i]; a[i] = a[n]; a[n] = x; /* swap a[i], a[n] */

    quicksort(m,j); quicksort(i+1,n);
}
```

Three-Address Code Quicksort

```
⟶   (1)  i = m-1                 (16)  t7 = 4*i
    (2)  j = n                   (17)  t8 = 4*j
    (3)  t1 = 4*n                (18)  t9 = a[t8]
    (4)  v = a[t1]               (19)  a[t7] = t9
⟶   (5)  i = i+1                 (20)  t10 = 4*j
    (6)  t2 = 4*i                (21)  a[t10] = x
    (7)  t3 = a[t2]              (22)  goto (5)
    (8)  if t3<v goto (5)     ⟶  (23)  t11 = 4*i
⟶   (9)  j = j-1                 (24)  x = a[t11]
   (10)  t4 = 4*j                (25)  t12 = 4*i
   (11)  t5 = a[t4]              (26)  t13 = 4*n
   (12)  if t5>v goto (9)        (27)  t14 = a[t13]
⟶  (13)  if i>=j goto (23)       (28)  a[t12] = t14
⟶  (14)  t6 = 4*i                (29)  t15 = 4*n
   (15)  x = a[t6]               (30)  a[t15] = x
```

Flow Graph Quicksort



B_1

```
i = m-1
j = n
t1 = 4*n
v = a[t1]
```

B_2

```
i = i+1
t2 = 4*i
t3 = a[t2]
if t3 < v goto B2
```

B_3

```
j = j-1
t4 = 4*j
t5 = a[t4]
if t5 > v goto B3
```

B_4

```
if i >= j goto B6
```

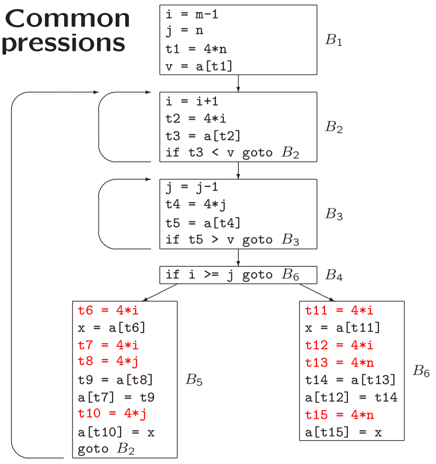
B_5

```
t6 = 4*i
x = a[t6]
t7 = 4*i
t8 = 4*j
t9 = a[t8]
a[t7] = t9
t10 = 4*j
a[t10] = x
goto B2
```

B_6

```
t11 = 4*i
x = a[t11]
t12 = 4*i
t13 = 4*n
t14 = a[t13]
a[t12] = t14
t15 = 4*n
a[t15] = x
```

Local Common Subexpressions



B_1

```
i = m-1
j = n
t1 = 4*n
v = a[t1]
```

B_2

```
i = i+1
t2 = 4*i
t3 = a[t2]
if t3 < v goto B2
```

B_3

```
j = j-1
t4 = 4*j
t5 = a[t4]
if t5 > v goto B3
```

B_4

```
if i >= j goto B6
```

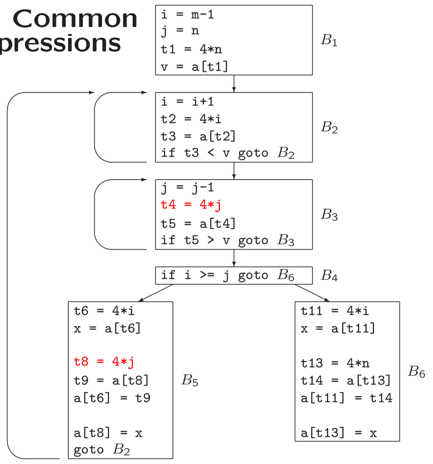
B_5

```
t6 = 4*i
x = a[t6]
t7 = 4*i
t8 = 4*j
t9 = a[t8]
a[t7] = t9
t10 = 4*j
a[t10] = x
goto B2
```

B_6

```
t11 = 4*i
x = a[t11]
t12 = 4*i
t13 = 4*n
t14 = a[t13]
a[t12] = t14
t15 = 4*n
a[t15] = x
```

Global Common Subexpressions



B_1

```
i = m-1
j = n
t1 = 4*n
v = a[t1]
```

B_2

```
i = i+1
t2 = 4*i
t3 = a[t2]
if t3 < v goto B2
```

B_3

```
j = j-1
t4 = 4*j
t5 = a[t4]
if t5 > v goto B3
```

B_4

```
if i >= j goto B6
```

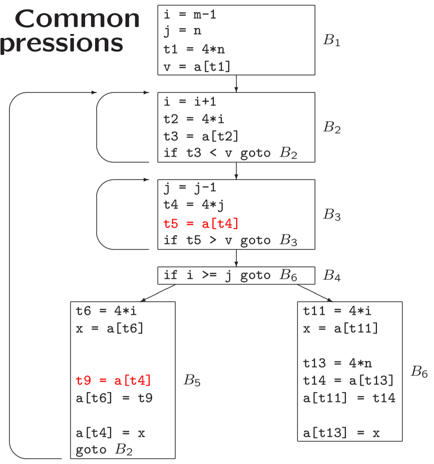
B_5

```
t6 = 4*i
x = a[t6]

t8 = 4*j
t9 = a[t8]
a[t6] = t9

a[t8] = x
goto B2
```

B_6

```
t11 = 4*i
x = a[t11]

t13 = 4*n
t14 = a[t13]
a[t11] = t14

a[t13] = x
```

Global Common Subexpressions



B_1

```
i = m-1
j = n
t1 = 4*n
v = a[t1]
```

B_2

```
i = i+1
t2 = 4*i
t3 = a[t2]
if t3 < v goto B2
```

B_3

```
j = j-1
t4 = 4*j
t5 = a[t4]
if t5 > v goto B3
```

B_4

```
if i >= j goto B6
```

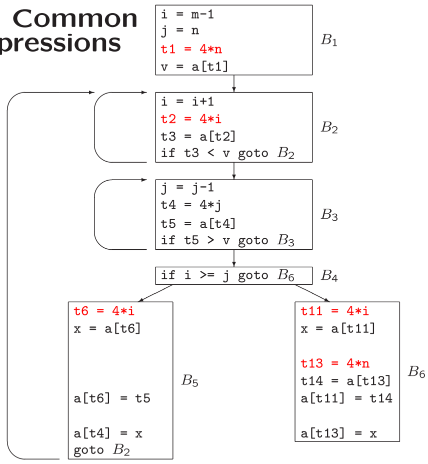
B_5

```
t6 = 4*i
x = a[t6]

t9 = a[t4]
a[t6] = t9

a[t4] = x
goto B2
```

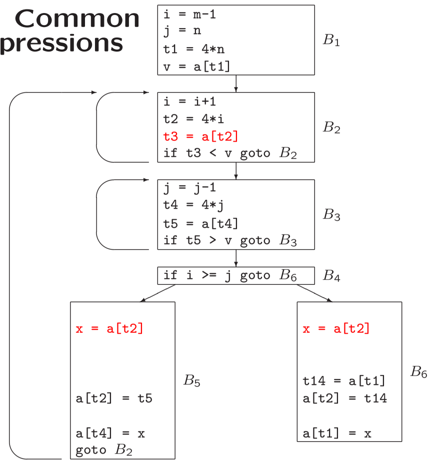
B_6

```
t11 = 4*i
x = a[t11]

t13 = 4*n
t14 = a[t13]
a[t11] = t14

a[t13] = x
```

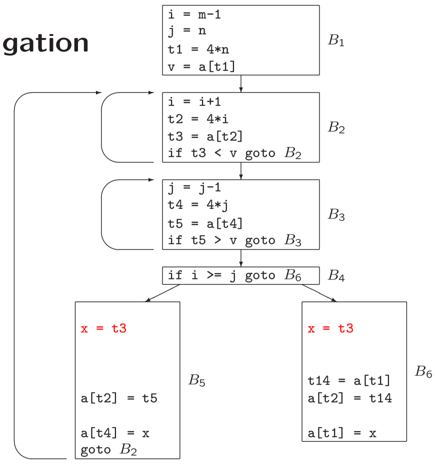
Global Common Subexpressions



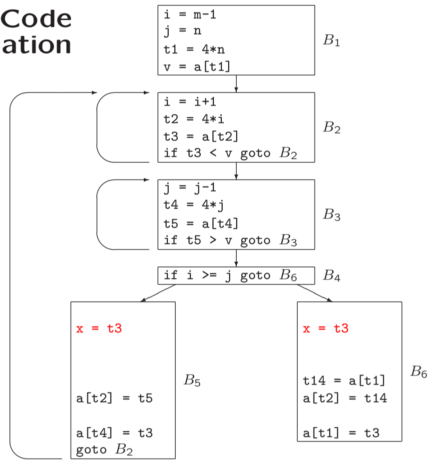
Global Common Subexpressions



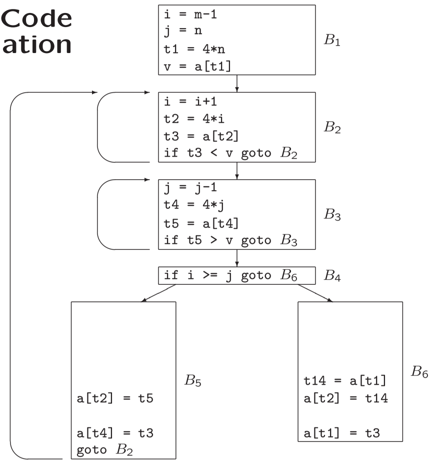
Copy Propagation



Dead-Code Elimination



Dead-Code Elimination



Code Motion

- loop-invariant computation
- compute before loop
- Example:

```
while (i <= limit-2)  /* statement does not change limit */
```

After code-motion

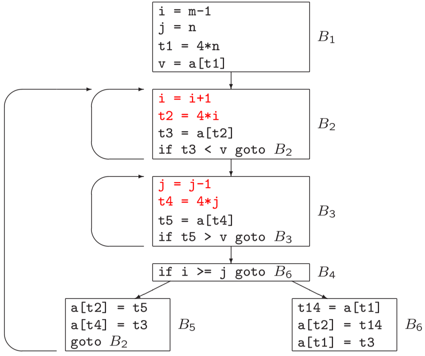
```
t = limit-2
while (i <= t)  /* statement does not change limit or t */
```

Induction Variables and Reduction in Strength

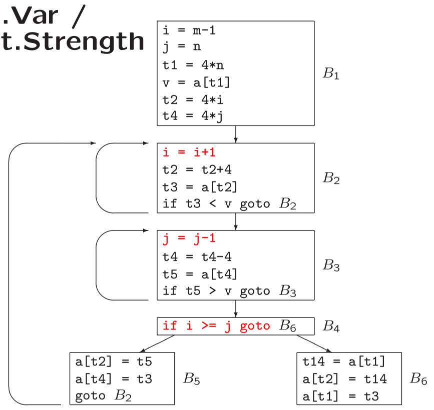
- Induction variable: each assignment to x of form $x = x + c$
- Reduction in strength: replace expensive operation by cheaper one

---

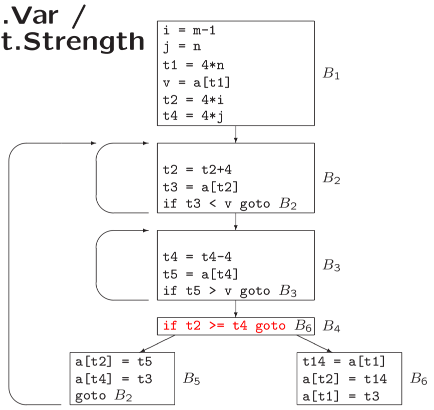
Induct.Var / Reduct.Strength



Induct.Var / Reduct.Strength



Induct.Var / Reduct.Strength



9.2 Introduction to Data-Flow Analysis

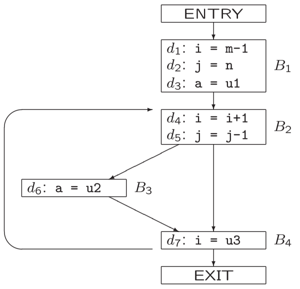
- Optimizations depend on data-flow analysis, e.g.,
  - Global common subexpression elimination
  - Dead-code elimination
- Execution path yields program state
- Extract information from program state for data-flow analysis
- Usually infinite number of execution paths / program states
- Different analyses extract different information

Data-Flow Analysis (Examples)

Extract information from program states at program point

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

Computing Reaching Definitions



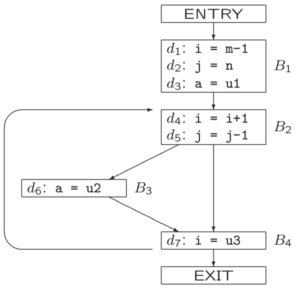
Reaching definitions
- Before B_1: $\emptyset$
- After B_1: $\{d_1, d_2, d_3\}$
- Before B_2: ...

Computing Reaching Definitions

- Effect of single definition $d : u = v$ *op* w:
  - $\mathit{gen}_d = \{d\}$
  - $\mathit{kill}_d = \{$all other definitions of u in program$\}$
- 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



$\mathit{gen}_{B_1} = \{d_1, d_2, d_3\}$
$\mathit{kill}_{B_1} = \{d_4, d_5, d_6, d_7\}$

$\mathit{gen}_{B_2} = \{d_4, d_5\}$
$\mathit{kill}_{B_2} = \{d_1, d_2, d_7\}$

$\mathit{gen}_{B_3} = \{d_6\}$
$\mathit{kill}_{B_3} = \{d_3\}$

$\mathit{gen}_{B_4} = \{d_7\}$
$\mathit{kill}_{B_4} = \{d_1, d_4\}$

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[$B$] = $\mathit{gen}_B \cup (\text{IN}[B] - \mathit{kill}_B)$
    }
```

Typical form of algorithm for forward data-flow analysis

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

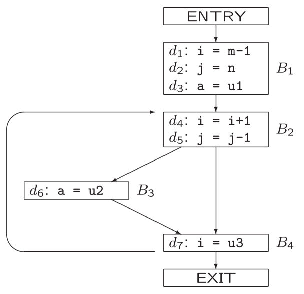
Live-Variable Analysis

- 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 college 7)
- Information about later use must be propagated backwards

Live-Variable Analysis

- Effect of block B on live variables
  - *def*$_B$: variables *defined* in B
  - *use*$_B$: variables that may be *used* in B prior to any definition in B

Computing Liveness



def$_{B_1} = \{i, j, a\}$
use$_{B_1} = \{m, n, u1\}$

def$_{B_2} = \{i, j\}$
use$_{B_2} = \{i, j\}$

def$_{B_3} = \{a\}$
use$_{B_3} = \{u2\}$

def$_{B_4} = \{i\}$
use$_{B_4} = \{u3\}$

Iterative Algorithm for Computing Liveness

```
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

Available expressions

- Is (value of) expression x *op* y available?
- Useful for global common subexpression elimination
- Can be decided with data-flow analysis (not for exam)

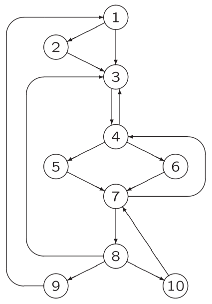
Efficient Iterative Data-Flow Analysis

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

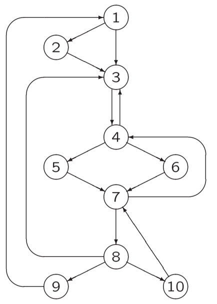
9.6 Loops in Flow Graphs

- Optimizations of loops have significant impact
- Essential to identify loops

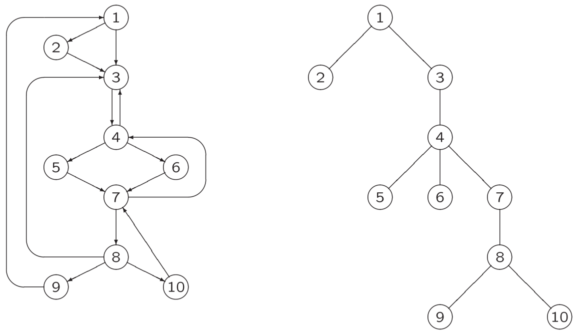
Dominators

- 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

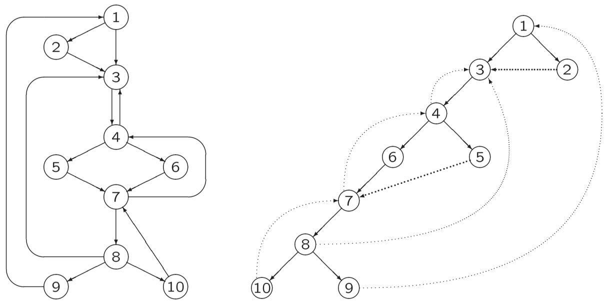
N is set of all nodes

```
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[$n$] $\cup \{n\}$
    }
```

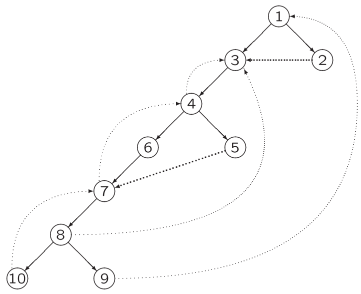
Depth-First Traversal

- 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



A 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

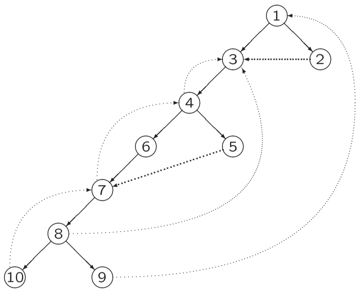
(Non)Reducible flow graphs

- 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?

Natural loops

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



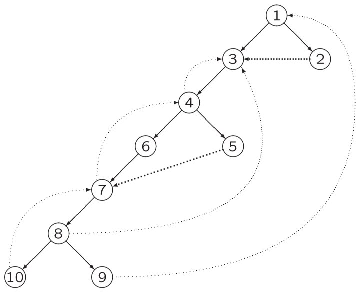
No Natural Loops

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. . .

A Depth-First Ordering

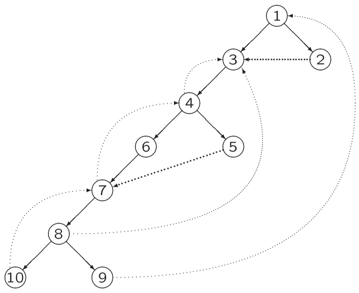


- Depth-First Ordering: nodes in DFST in WRL order ≈ reverse of postorder
- Example: 1,2,3,4,5,6,7,8,9,10
- Edge $m \to n$ is retreating, if and only if n comes before m in depth-first ordering

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 $10 \to 7 \to 4 \to 3$

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

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...

- Maandag 19 november: inleveren opdracht 3
- Dinsdag 20 november: practicum over opdracht 4
- Eerst naar 403, daarna naar 302/304
- Inleveren 10 december
- Dinsdag 27 november: werkcollege in 403
  (dus geen hoorcollege over Daedalus)
- Dinsdag 4 december: practicum over opdracht 4

Compiler constructie

college 8
Code Optimization

Chapters for reading:
9.intro, 9.1, 9.2–9.2.5, 9.6