Principles of Programming Languages

http://www.di.unipi.it/~andrea/Didattica/PLP-15/

Prof. Andrea Corradini
Department of Computer Science, Pisa

Lesson 33

- Data-Flow analysis for global optimization

Data-Flow Analysis

- A data-flow analysis schema defines a value at each point in the program, IN[s] and OUT[s] for each statement s
- Values are abstractions of all program states reachable in that point with an arbitrary computation path
- Statements of the program have associated *transfer functions* that relate the value before the statement to the value after
  - Forward OUT[s] = *f* (IN[s]) or backward IN[s] = *f* (OUT[s])
- Statements with more than one predecessor must have their value defined by combining the values at the predecessors, using a *meet* operator.
- Often *basic blocks* are annotated with values instead of individual statements: OUT[B] and IN[B]
- Useful for annotating the code with info needed for local or global optimization.

Data-Flow Analysis Framework

- A *Data-Flow Analysis Framework* (D, V, $\wedge$, F) consists of:
  - A *direction* D *in {FORWARDS, BACKWARDS}*
  - A *domain of values* (V, $\wedge$) which forms a *meet semilattice*:
    - A partial order with a *top element* and a binary operation *meet (*$\wedge$*, greatest lower bound*) such that

$$x \wedge y \leq x \ \ and \ \ x \wedge y \leq y \ and \ \ (\forall z. z \leq x \ \ and \ \ z \leq y \Rightarrow z \leq x \wedge y)$$

  - A family *F* of *transfer functions* from V to V, including the identity function and closed under composition

- A framework is monotone if if for all *f* in *F* $\quad x \leq y \Rightarrow f(x) \leq f(y)$

- It is distributive if for all *f* in *F* $\quad f(x \wedge y) = f(x) \wedge f(y)$

Data-Flow Iterative Algorithm

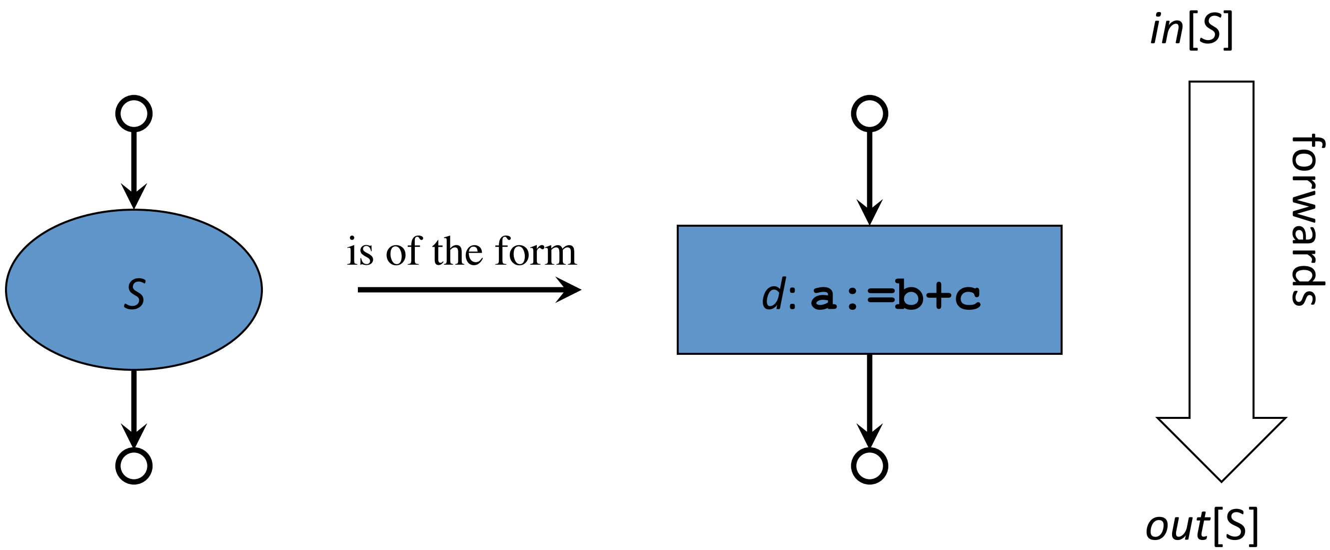
- [Forward] Given:
  - a data-flow graph with ENTRY and EXIT nodes
  - one transfer function f_B for each basic block *B*
  - A "boundary condition" v_{ENTRY}
- Computes values IN[B] and OUT[B] for all blocks

```
1)  OUT[ENTRY] = v_ENTRY;
2)  while (changes to any OUT occur)
3)      for (each basic block B other than ENTRY){
4)          IN[B] = ∧_{p a predecessor of B} OUT[P];
5)          OUT[B] = f_B(IN[B]);
    }
```

Example: Dataflow analysis for Reaching Definitions

- Each point in the program is associated with the set of definitions that are active at that point
- Semilattice:
  - Powerset of definitions (assignments)
  - Meet operator: union. Top element: empty set
- The *transfer function* for a block kills definitions of variables that are redefined in the block and adds definitions of variables that occur in the block: $f_B(x) = gen_B \cup (x - kill_B)$
- The confluence operator is union.

Reaching Definitions



Then, the data-flow equations for *S* are:

applies transfer function: $f_{[S]}(x) = gen_{[S]} \cup (x - kill_{[S]})$

$$gen[S] = \{d\}$$

$$kill[S] = D_{\mathtt{a}} - \{d\}$$

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

where $D_{\mathtt{a}}$ = all definitions of **`a`** in the region of code

Reaching Definitions: Iterative solution

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

- Visiting order in line 4) influences convergence
- Very efficient implementations with bit vectors
- Non-iterative solutions possible: Syntax-directed, and region-based

Dataflow analysis for Reaching Definitions towards a syntax directed algorithm

```
d1: i  := m-1;
d2: j  := n;
d3: a  := u1;
    do
d4:    i  := i+1;
d5:    j  := j-1;
       if e1 then
d6:       a  := u2
       else
d7:       i  := u3
    while e2
```

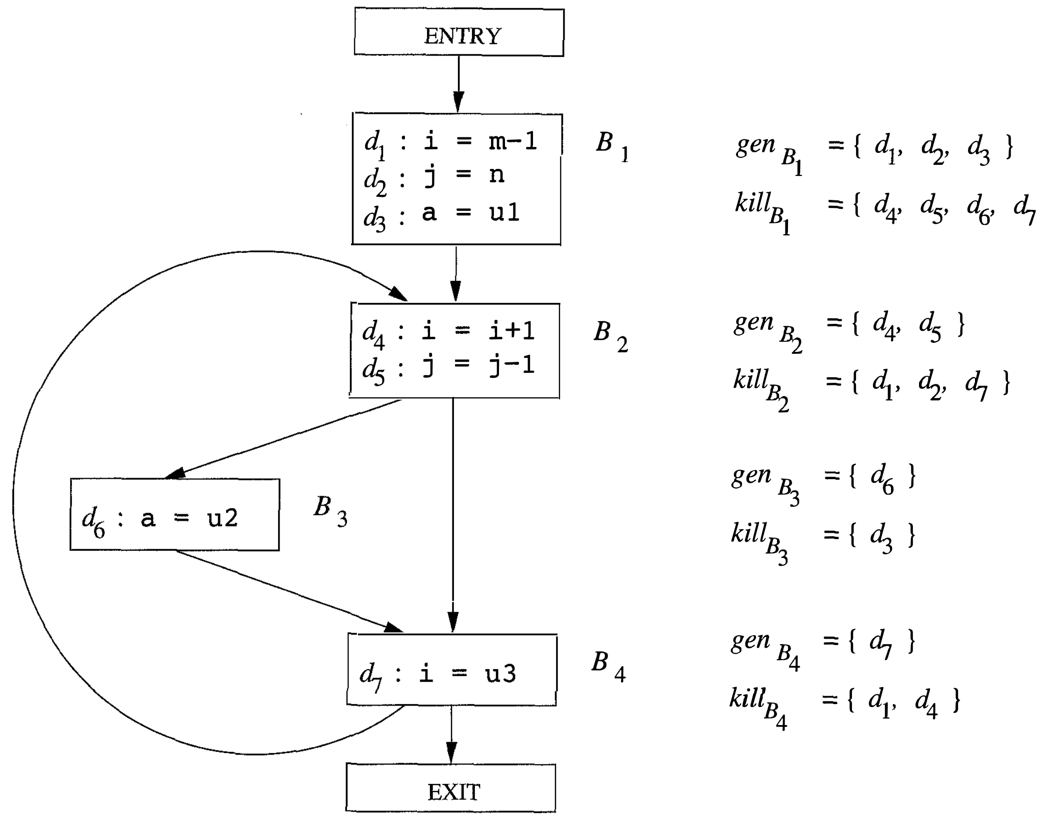
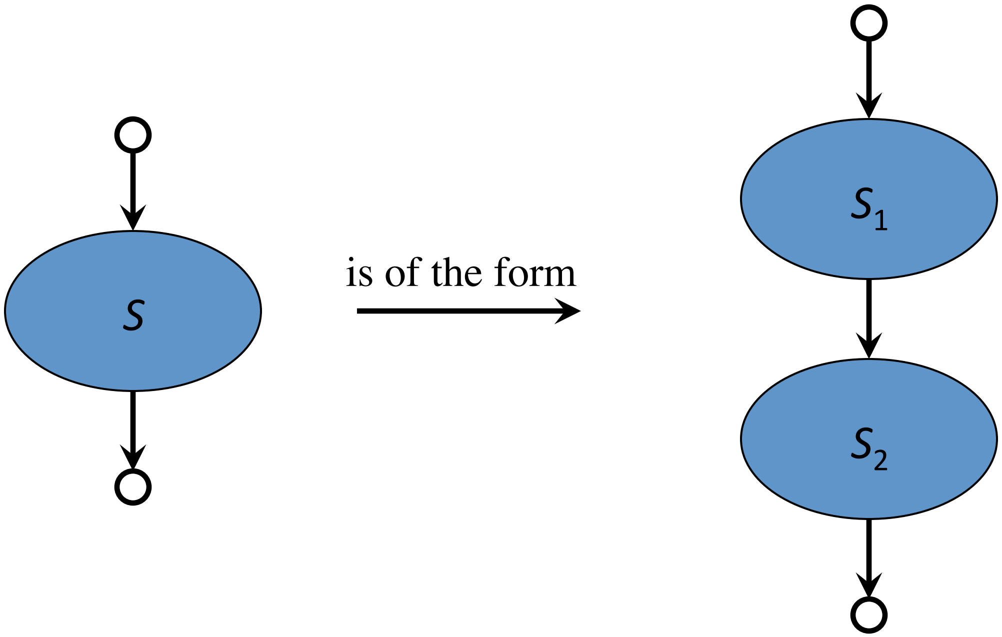


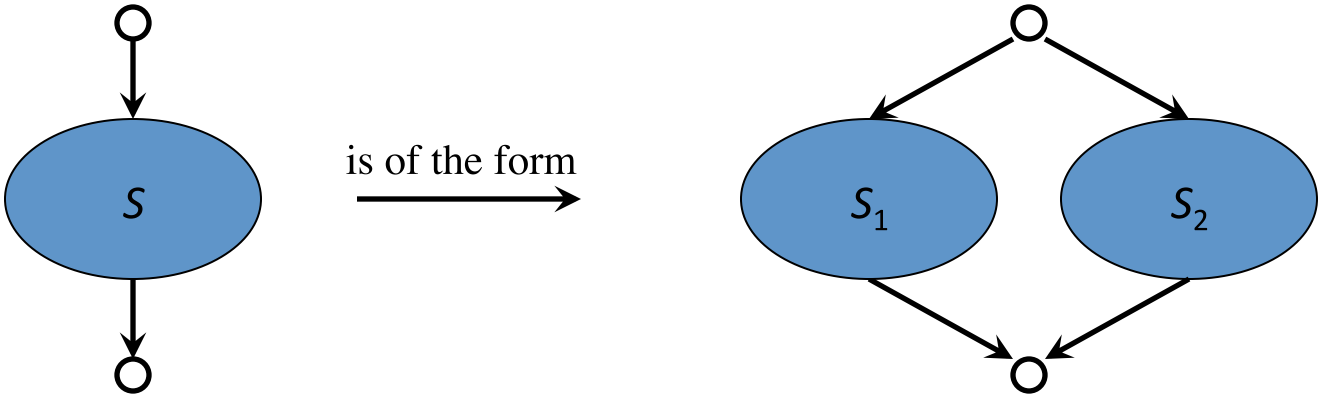
Figure 9.13: Flow graph for illustrating reaching definitions

Reaching Definitions



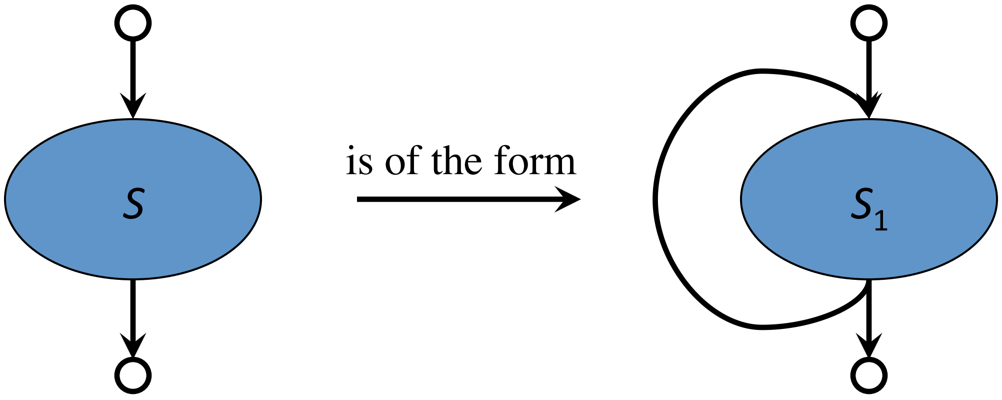
$$
\begin{aligned}
gen[S] &= gen[S_2] \cup (gen[S_1] - kill[S_2]) \\
kill[S] &= kill[S_2] \cup (kill[S_1] - gen[S_2]) \\
in[S_1] &= in[S] \\
in[S_2] &= out[S_1] \\
out[S] &= out[S_2]
\end{aligned}
$$

Reaching Definitions



$$
\begin{aligned}
gen[S] &= gen[S_1] \cup gen[S_2] \\
kill[S] &= kill[S_1] \cap kill[S_2] \\
in[S_1] &= in[S] \\
in[S_2] &= in[S] \\
out[S] &= out[S_1] \cup out[S_2]
\end{aligned}
$$

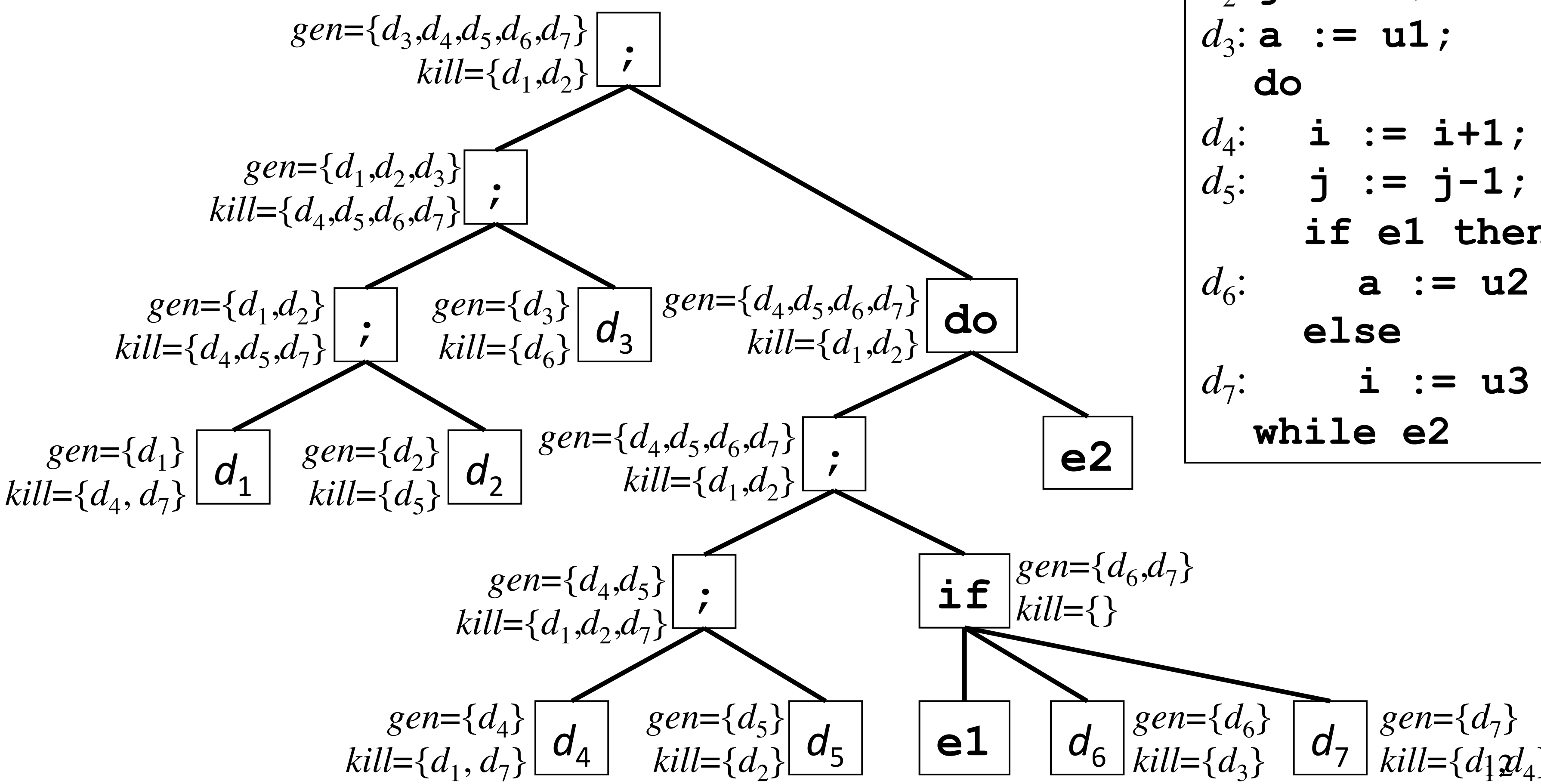
Reaching Definitions



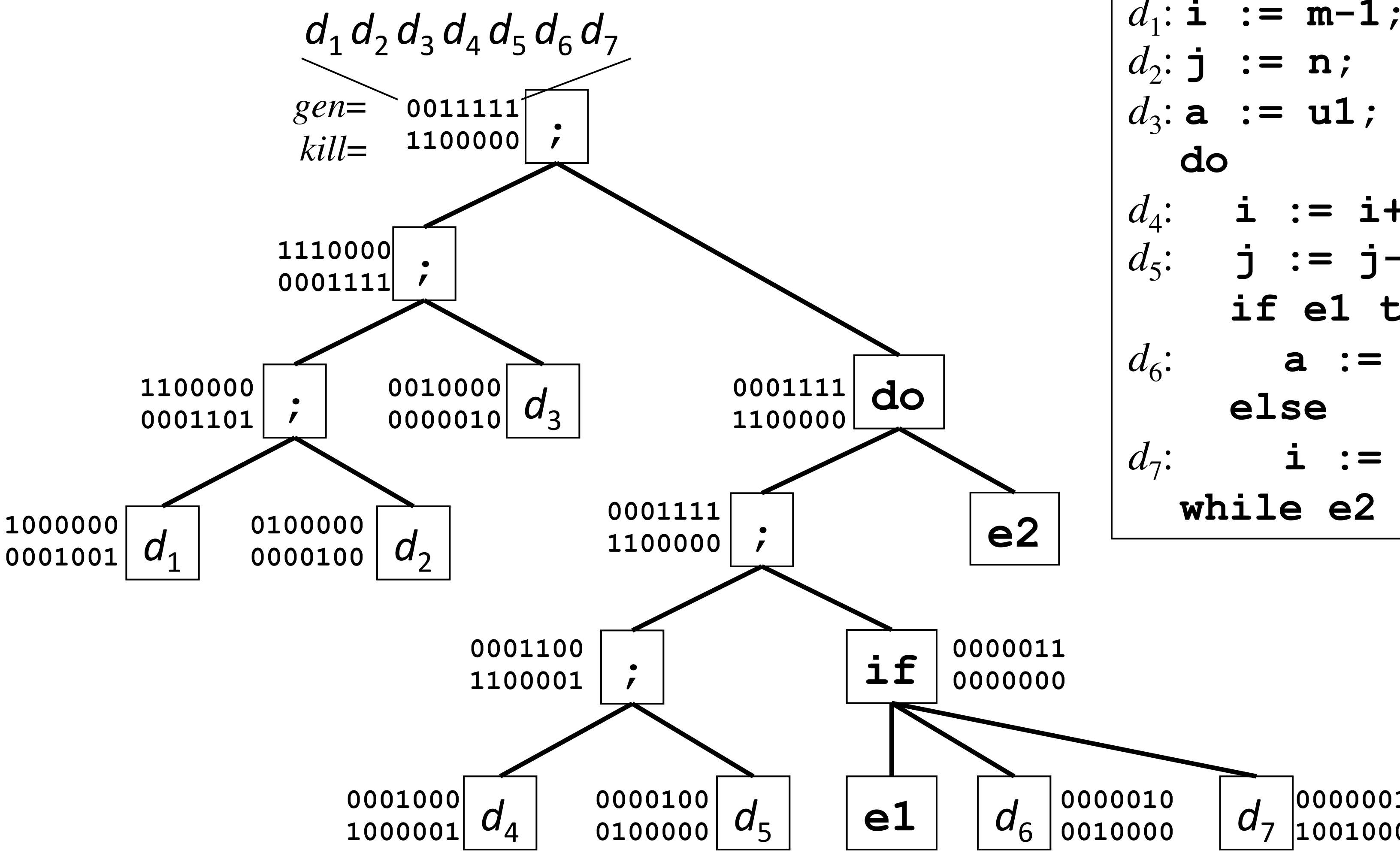
$$
\begin{aligned}
gen[S] &= gen[S_1] \\
kill[S] &= kill[S_1] \\
in[S_1] &= in[S] \cup gen[S_1] \\
out[S] &= out[S_1]
\end{aligned}
$$

Reaching Definitions: Computing Gen/Kill

```
d1: i := m-1;
d2: j := n;
d3: a := u1;
    do
d4:     i := i+1;
d5:     j := j-1;
        if e1 then
d6:         a := u2
        else
d7:         i := u3
    while e2
```



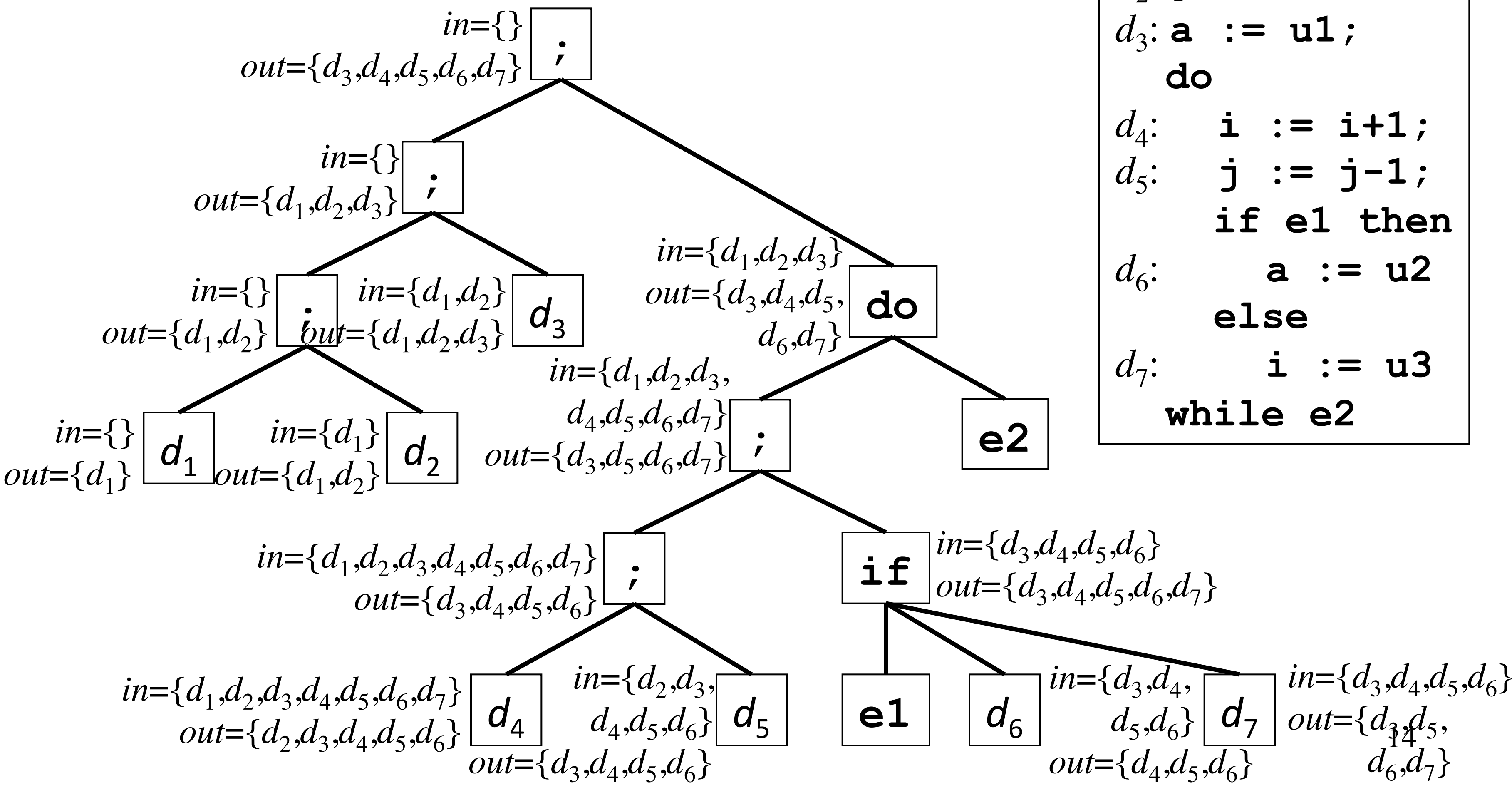
Using Bit-Vectors to Compute Reaching Definitions




```
d1: i := m-1;
d2: j := n;
d3: a := u1;
    do
d4:    i := i+1;
d5:    j := j-1;
       if e1 then
d6:       a := u2
       else
d7:       i := u3
    while e2
```

Reaching Definitions: Computing In/Out (non-iterative)

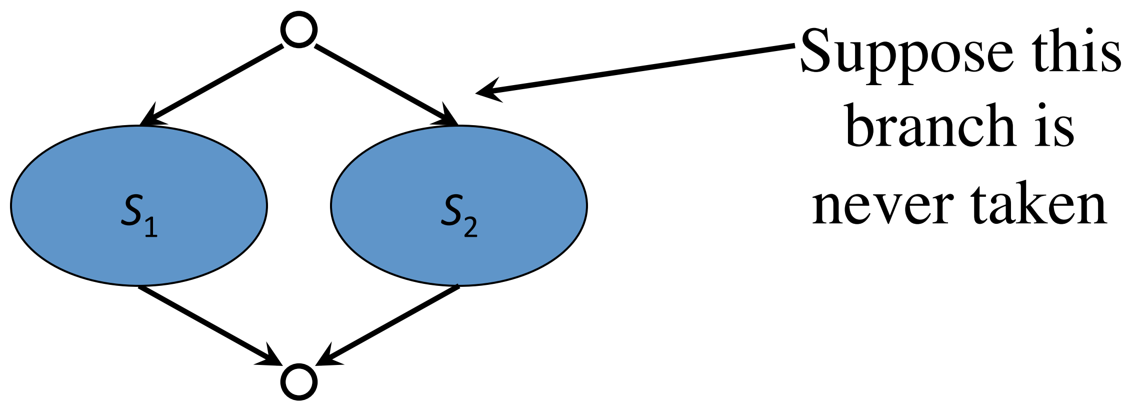
```
d1: i := m-1;
d2: j := n;
d3: a := u1;
    do
d4:   i := i+1;
d5:   j := j-1;
      if e1 then
d6:     a := u2
      else
d7:     i := u3
    while e2
```



Accuracy, Safeness, and Conservative Estimations

- *Conservative*: refers to making safe assumptions when insufficient information is available at compile time, i.e. the compiler has to guarantee not to change the meaning of the optimized code
- *Safe*: refers to the fact that a superset of reaching definitions is safe (some may have been killed)
- *Accuracy*: more and better information enables more code optimizations

Reaching Definitions are a Conservative (Safe) Estimation



Estimation:

$gen[S] = gen[S_1] \cup gen[S_2]$

$kill[S] = kill[S_1] \cap kill[S_2]$

Accurate:

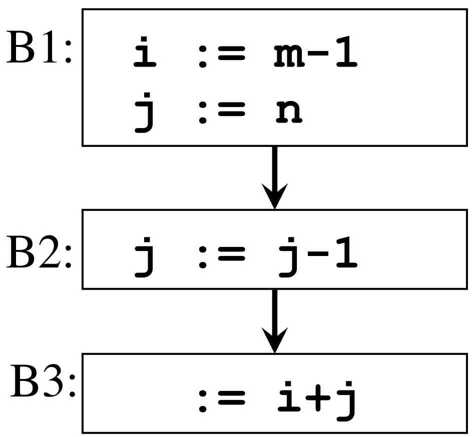
$gen'[S] = gen[S_1] \subseteq gen[S]$

$kill'[S] = kill[S_1] \supseteq kill[S]$

Example: Dataflow analysis for Live Variables [backwards!]

- Each point in the program is associated with the set of variables that are *live* at that point, i.e. such that their value will be used later
- Semilattice:
  - Powerset of variables
  - Meet operator: union. Top element: empty set
- A variable is *live* at the beginning of a block if it is either used before definition in the block or is live at the end of the block and not redefined in the block.
- The *transfer function*: $f_B(x) = use_B \cup (x - def_B)$
- The confluence operator is union.

Data-Flow Analysis for Live Variables: an example



Solution:

$in[B1] = \{\mathbf{m}, \mathbf{n}\} \cup (\{\mathbf{i}, \mathbf{j}\} - \{\mathbf{i}, \mathbf{j}\}) = \{\mathbf{m}, \mathbf{n}\}$

$out[B1] = in[B2] = \{\mathbf{i}, \mathbf{j}\}$

$in[B2] = \{\mathbf{j}\} \cup (\{\mathbf{i}, \mathbf{j}\} - \{\mathbf{j}\}) = \{\mathbf{i}, \mathbf{j}\}$

$out[B2] = in[B3] = \{\mathbf{i}, \mathbf{j}\}$

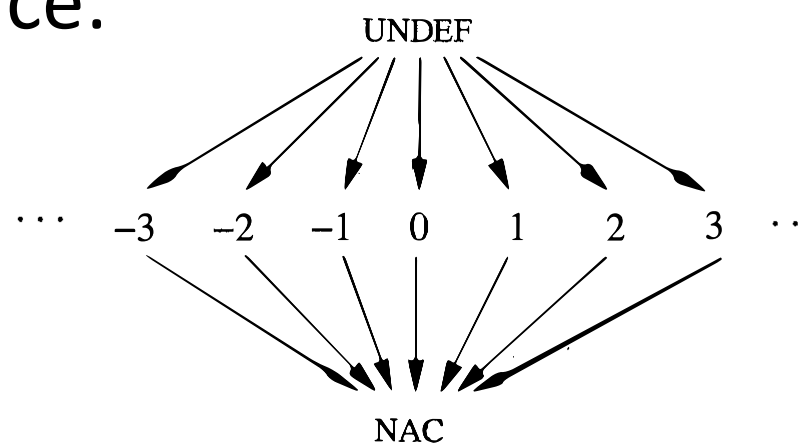
Live variables: Iterative solution

1) IN[EXIT] = { };
2) for (each basic block B) IN[B] = { }
3) while (changes to any IN occur)
4) for (each basic block B other than EXIT){
5) $\text{OUT}[B] = \text{U}_{\text{S a successor of B}}\ \text{IN}[S]$;
6) $\text{IN}[B] = use_B\ U\ (\text{OUT}[B] - def_B)$

}

Constant Propagation/Folding

- Unbounded set of values:
  - All constants for the relevant type
  - NAC: not-a-constant
  - UNDEF: no info about any value of the variable
- The semilattice:

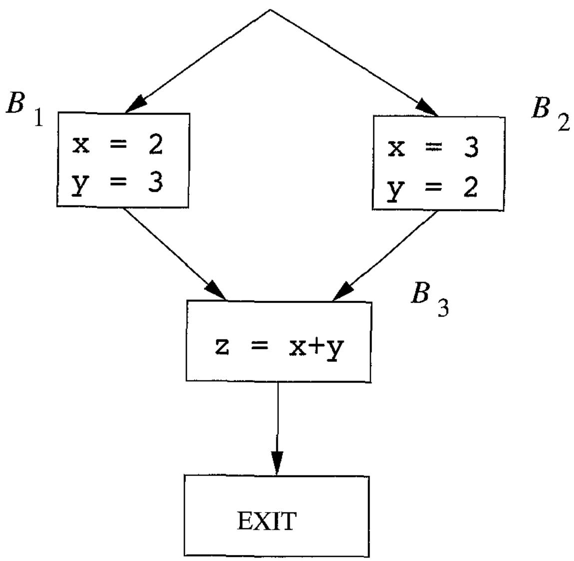


Constant Propagation/Folding

- Transfer function for statements:
  1. Identity, if it is not an assigment
  2. If it is an assigment to x:
     1. m'(v) = m(v) for v != x
     2. If the RHS is a constant c, m'(x) = c
     3. If the RHS is "y *op* z",
        1. If m(y) and m(z) are constant, m'(x) = m(y) *op* m(z)
        2. If m(y) = NAC or m(z) = NAC, then m'(x) = NAC
        3. m'(x) = UNDEF, otherwise
     4. If the RHS is anything else (e.g. function call) m'(x) = NAC

Constant Propagation/Folding

- Transfer functions are monotonic but not distributive



m	$m(x)$	$m(y)$	$m(z)$
m_0	UNDEF	UNDEF	UNDEF
$f_1(m_0)$	2	3	UNDEF
$f_2(m_0)$	3	2	UNDEF
$f_1(m_0) \wedge f_2(m_0)$	NAC	NAC	UNDEF
$f_3(f_1(m_0) \wedge f_2(m_0))$	NAC	NAC	NAC
$f_3(f_1(m_0))$	2	3	5
$f_3(f_2(m_0))$	3	2	5
$f_3(f_1(m_0)) \wedge f_3(f_2(m_0))$	NAC	NAC	5

$$f_3\big(f_1(m_0) \wedge f_2(m_0)\big) < f_3\big(f_1(m_0)\big) \wedge f_3\big(f_2(m_0)\big)$$

On Partial-Redundancy Elimination

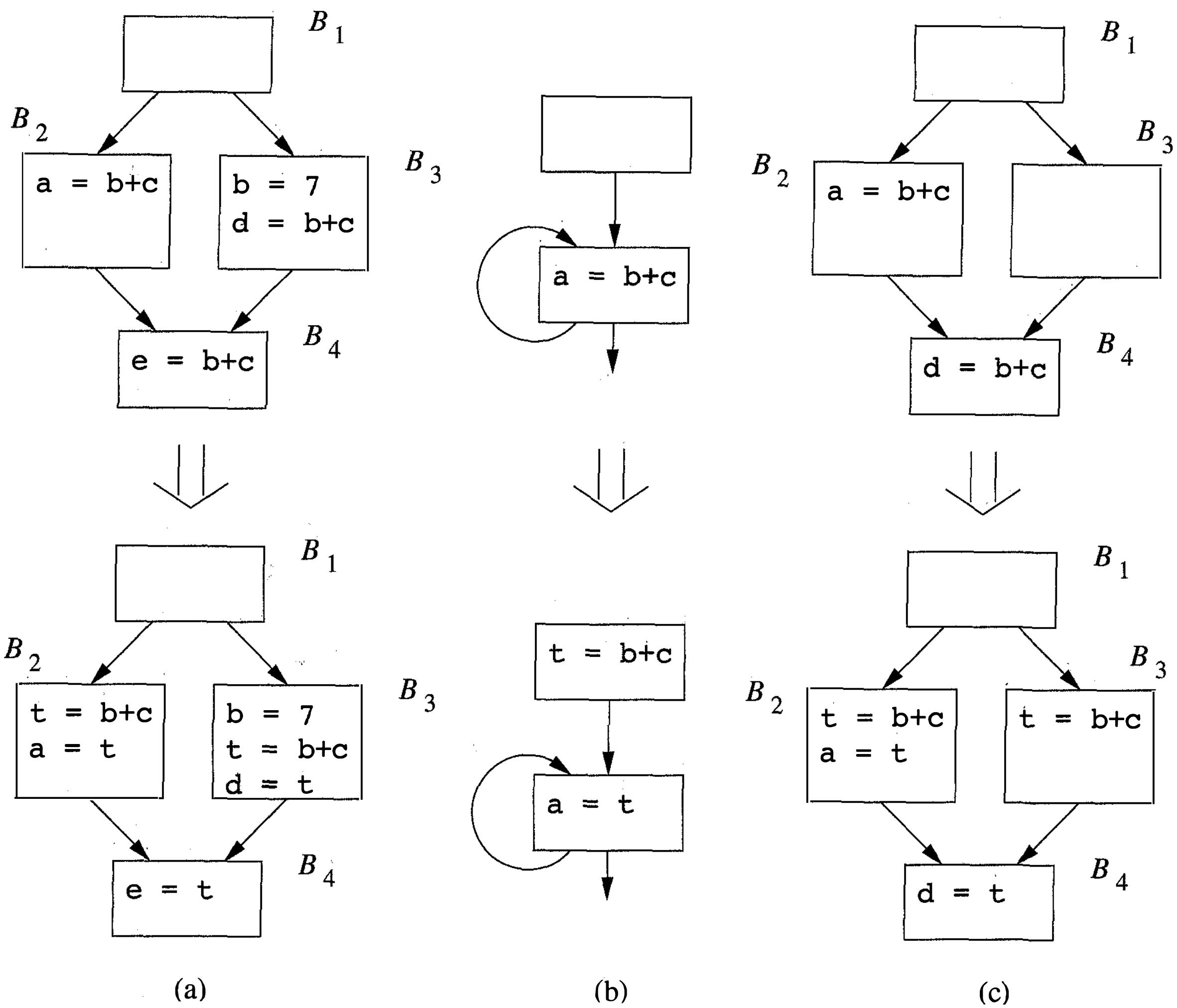


Figure 9.30: Examples of (a) global common subexpression, (b) loop-invariant code motion, (c) partial-redundancy elimination.

On Partial-Redundancy Elimination

- Four step “Lazy Code Motion” algorithm
  - Find blocks where evaluation of an expression can be anticipated (backwards)
  - Check availability of expressions along all paths leading to a block needing it (forwards)
  - Postpone the expression as much as possible (forwards)
  - Eliminate assignments to temporaries that are used only once (backwards)

Determining Loops in Flow Graphs

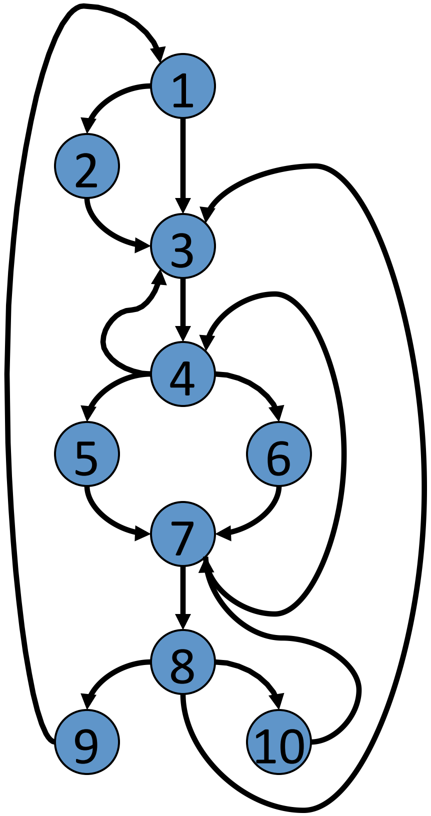
- In absence of loops data-flow analysis converges in one pass, if performed according to topological order
- Study of loops needed also to evaluate convergence speed
- For some values semi-lattices, loops do not modify values, so they can be ignored
- For others, several iterations in loops are needed: eg, constant folding

```
L:  x = y;
    y = z;
    z = 1;
    goto L
```

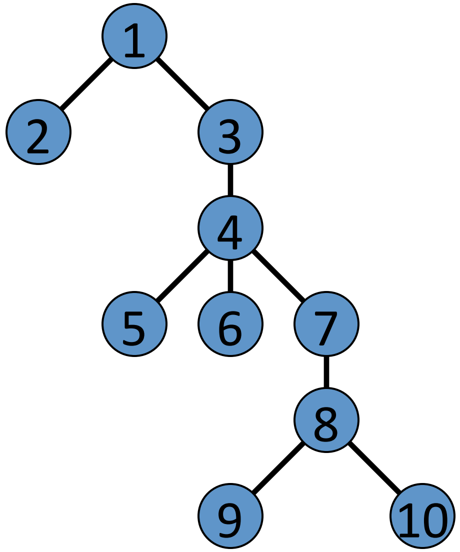
Determining Loops in Flow Graphs: Dominators

- Dominators: d dom n
  - Node d of a CFG dominates node n if every path from the initial node of the CFG to n goes through d
  - The loop entry dominates all nodes in the loop
- The immediate dominator m of a node n is the last dominator on the path from the initial node to n
  - If d $\neq$ n and d dom n then d dom m

Dominator Trees



CFG



Dominator tree

Data-Flow analysis for Dominators

- Computes *D(n)*, set of dominators for each node *n* (forwards)
- Semilattice: powerset of CFG nodes
- Transfer function: $f_B(x) = x \cup \{B\}$
- Meet operator: intersection
- Boundary: OUT[ENTRY] ={ENTRY}
- Initialization: OUT[B] = NODES

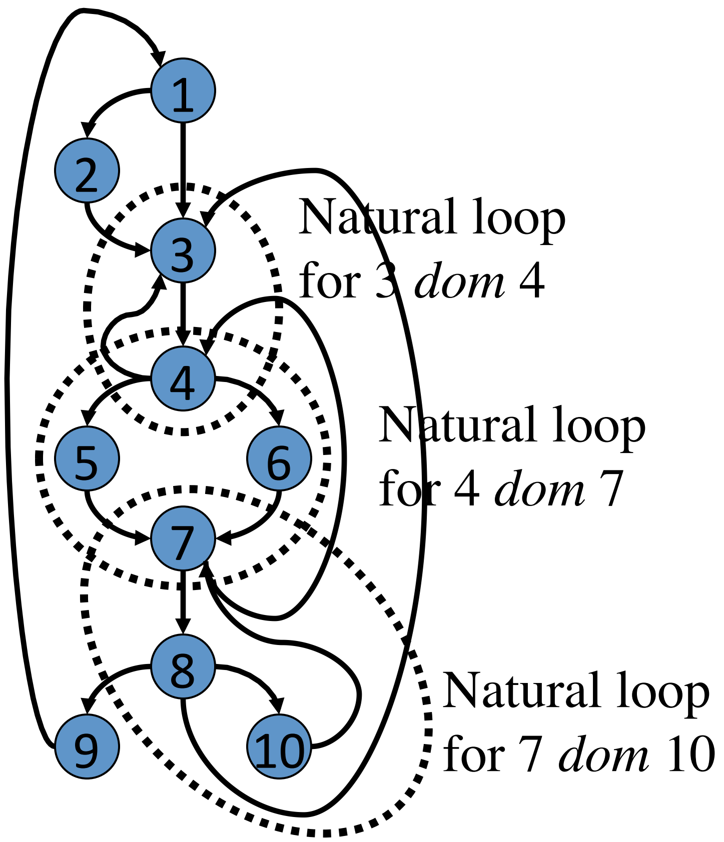
Natural Loops

- A *back edge* is an edge $a \rightarrow b$ whose head b dominates its tail a
- Given a back edge $n \rightarrow d$
  - The *natural loop* consists of d plus the nodes that can reach n without going through d
  - The *loop header* is node d
- In other words
  - A *natural loop* must have a single-entry node d
  - There must be a back edge that enters node d

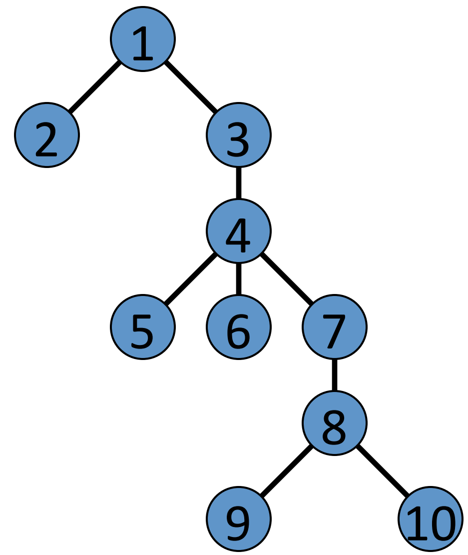
Natural Inner/Outer Loops

- Unless two loops have the same header, they are disjoint or one is nested within the other
- A nested loop is an *inner loop* if it contains no other loops
- A loop is an *outer loop* if it is not contained within another loop

Natural Inner Loops Example

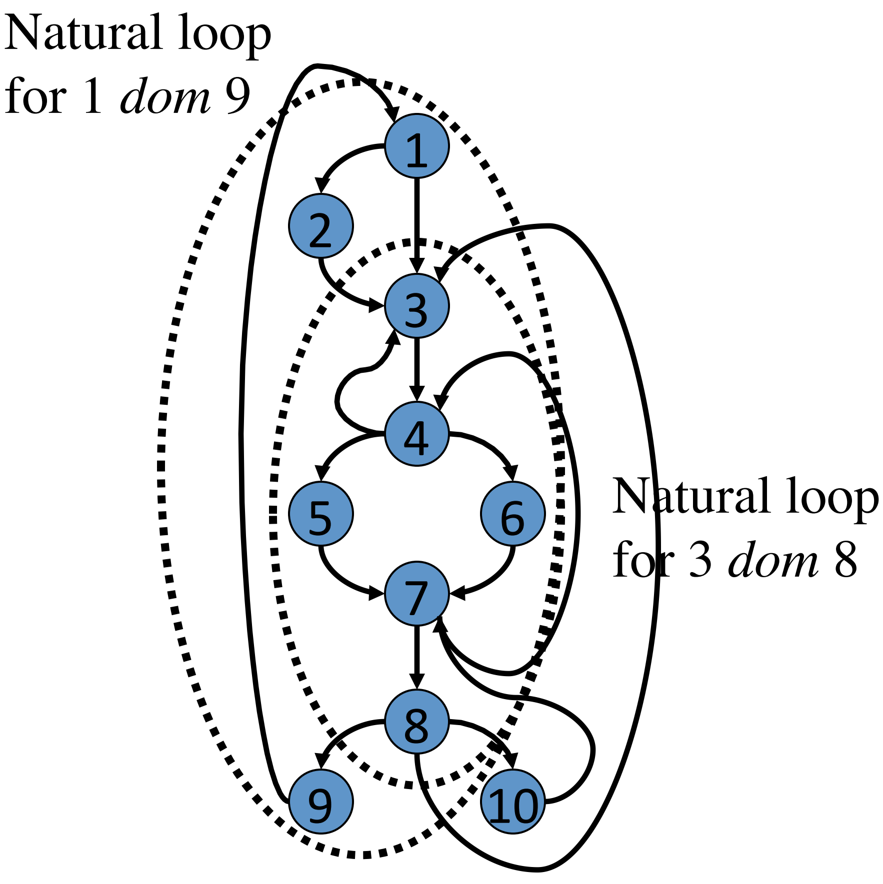


CFG

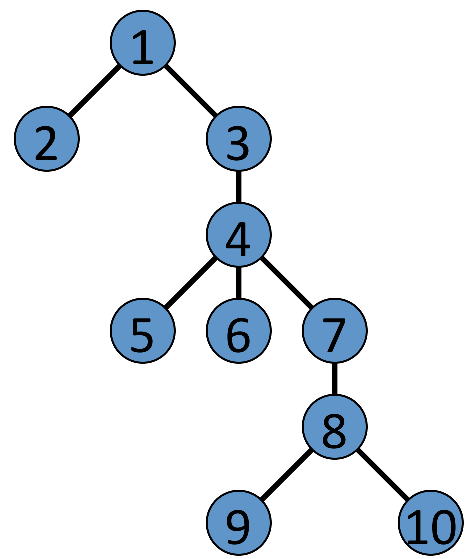


Dominator tree

Natural Outer Loops Example



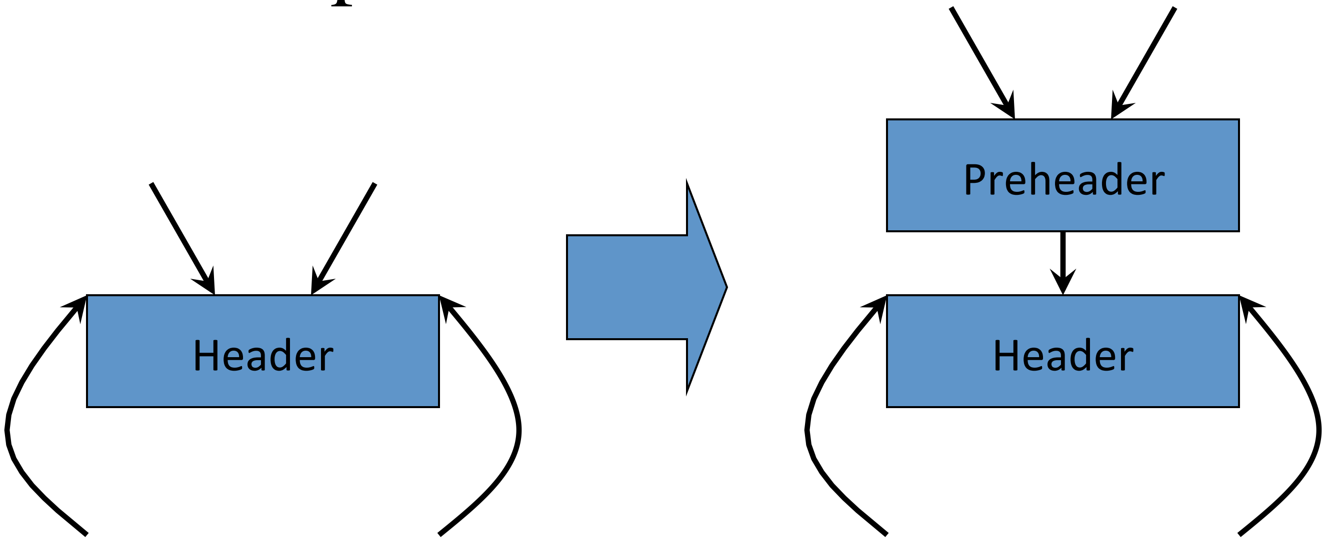
CFG



Dominator tree

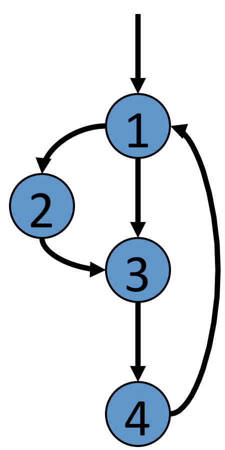
Pre-Headers

- To facilitate loop transformations, a compiler often adds a *preheader* to a loop
- Code motion (of loop invariant code), strength reduction, and other loop transformations populate the preheader

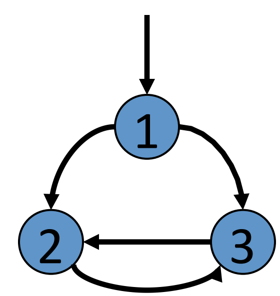


Reducible Flow Graphs

- *Reducible graph* = disjoint partition in forward and back edges such that the forward edges form an acyclic (sub)graph



Example of a reducible CFG



Example of a nonreducible CFG
(not a natural loop: no back edge to dominator 1)

Speed of convergence of data-flow analysis

- Maximum number of iterations: (height of the lattice) x (number of nodes)
- If value of interest can be propagated along acyclic path (*reaching definitions,available expressions, live variables*), few passes are sufficient in general, depending on the depth of the graph (~ number of loop nesting).