CS293S Data Flow Analysis

Yufei Ding

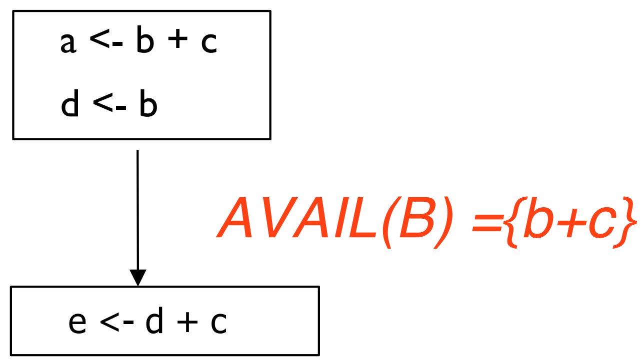
Questions from students

- Office hour by appointments ☺
- Which one is Global method?

  LVN, SVN, DVN, GCSE
- SVN and DVN: build the SSA first
- EXPRKILL and DEEXPR: check the pseudocodes in the lecture slides.
- When introducing GCSE, we focus on the program analysis part, i.e., available expression elimination. How about the program transformation part for redundancy elimination?

Replacement step in GCSE

- Limit to textually identical expressions
  (like DAG, unlike value numbering)

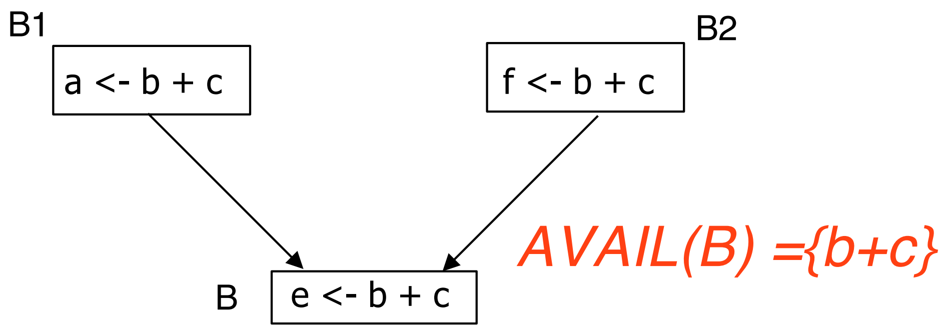


Cannot find or remove the redundancy!

Replacement step in GCSE

- Limit to textually identical expressions

  (like DAG, unlike value numbering)

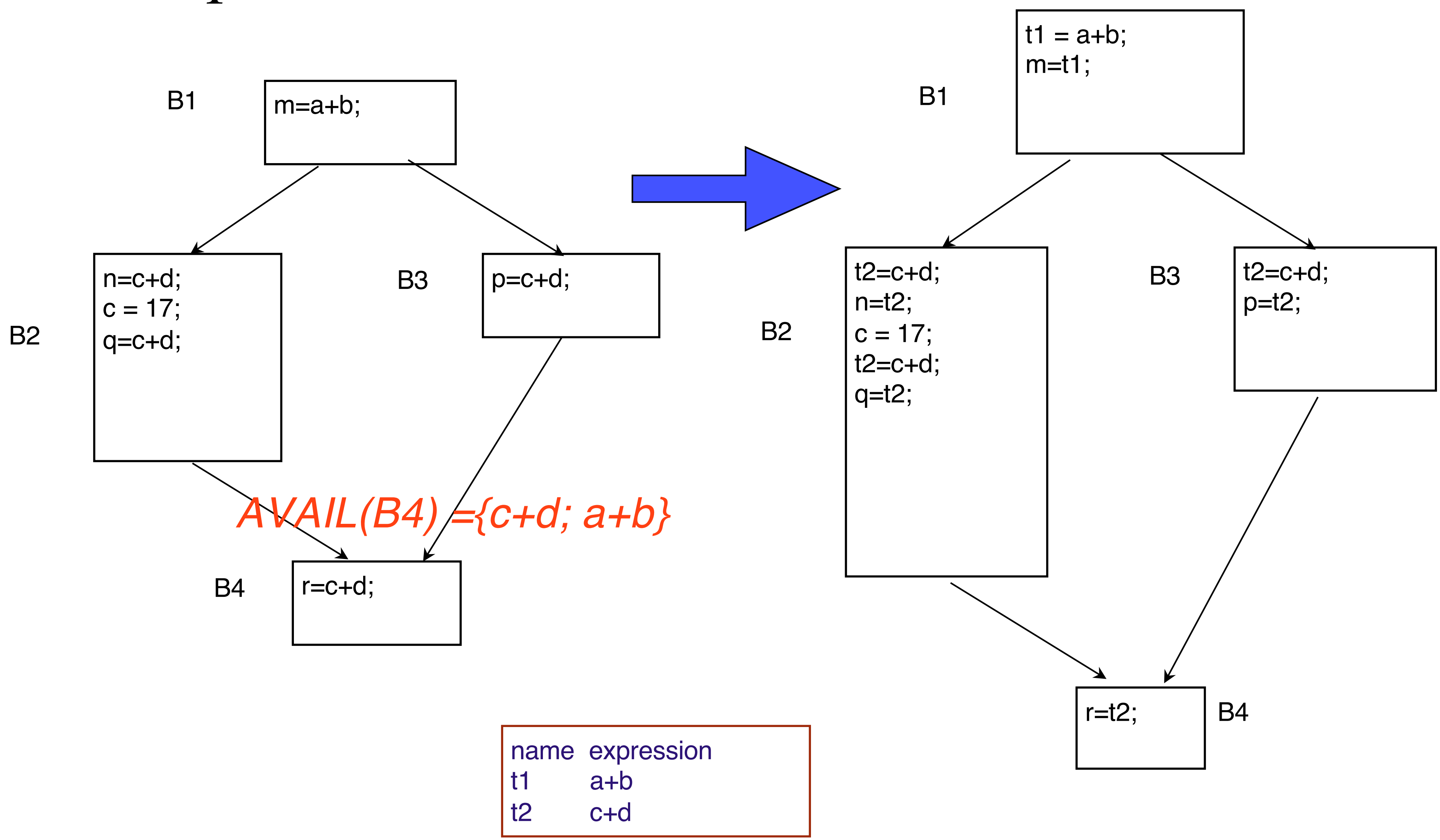


Should replace b+c with ?

GCSE (replacement step)

- Compute a static mapping from expression to name
  - After analysis & before transformation
  - $\forall$ block b, $\forall$ expression e$\in$AVAIL(b), assign e a global name by hashing on e
- During transformation step
  - Evaluation of e $\Rightarrow$ insert copy name(e) $\leftarrow$ e
    - (e is not available and needs to be evaluated)
  - Reference to e $\Rightarrow$ replace e with name(e)
    - (e is available and should be replaced)

Example


B1
m=a+b;
B2
n=c+d;
c = 17;
q=c+d;
B3
p=c+d;
AVAIL(B4) ={c+d; a+b}
B4
r=c+d;
B1
t1 = a+b;
m=t1;
B2
t2=c+d;
n=t2;
c = 17;
t2=c+d;
q=t2;
B3
t2=c+d;
p=t2;
B4
r=t2;
name expression
t1 a+b
t2 c+d

GCSE (replacement step)

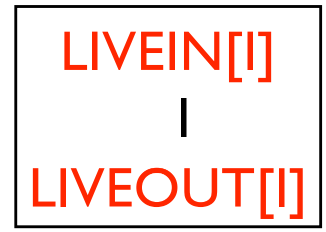
- The major problem with this approach
  - Inserts extraneous copies
  - At all definitions and uses of any $e \in AVAIL(b)$, $\forall$ b
- Not a big issue
  - Those extra copies are dead and easy to remove

Review of Last Class

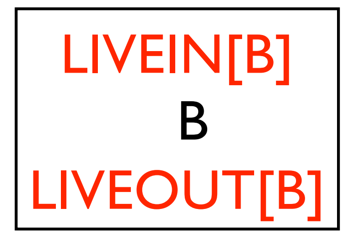
- Global Common Subexpression Elimination (GCSE)
  - First data/control flow analysis
- Live Variable Analysis

How to Compute Liveness?

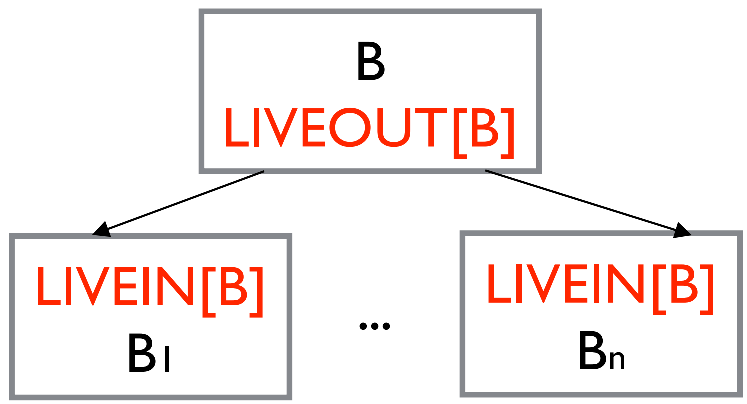
- Question 1: for each instruction I, what is the relation between LIVEIN[I] and LIVEOUT[I]?



- Question 1: for each block B, what is the relation between LIVEIN[B] and LIVEOUT[B]?



- Question 2: for each basic block B with successor blocks B1, ..., Bn, what is the relation between LIVEOUT[B] and LIVEIN[B1], ..., LIVEIN[Bn]?



Analyze CFG

- Mathematically:

$$LIVEOUT[B] = \bigcup_{B' \in succ(B)} ((LIVEOUT[B'] - VARKILL(B')) \bigcup UEVAR(B'))$$

- The information flows backward: from successors B' of B to basic block
  - LIVEOUT(B) contains the name of every variable that is live at the exit point of basic block B.
  - UEVAR(B) contains the upward-exposed variables in B, i.e. those that are used in n before any redefinition in B.
  - VARKILL(B) contains all the variables that are defined in B.

Three Steps in Data-Flow Analysis

- Build a CFG
- Gather the initial information for each block (i.e., (UEVAR and VARKILL))
- Use an iterative fixed-point algorithm to propagate information around the CFG

Algorithm

// Get initial sets

```
for each block b
   UEVAR(b) = Ø
   VARKILL(b) = Ø
   for i=1 to number of instr in b
      (assuming inst I is “x= y op z”)
        if y ∉VARKILL(b) then
           UEVAR(b) = UEVAR(b) ∪ {y}
        if z ∉VARKILL(b) then
           UEVAR(b) = UEVAR(b) ∪ {z}
        VARKILL(b) = VARKILL(b) ∪ {x}
```

// update LiveOut version 1

```
set LIVEOUT(bi) to Ø for all blocks
Worklist ← {all blocks}
while (Worklist ≠ Ø)
          remove a block b from Worklist
          recompute LIVEOUT(b)
          if LIVEOUT(b) changed then
             Worklist ← Worklist ∪ pred(b)
```

$$LIVEOUT[B] = \bigcup_{B' \in succ(B)} ((LIVEOUT[B'] - VARKILL(B')) \bigcup UEVAR(B'))$$

Algorithm

// Get initial sets

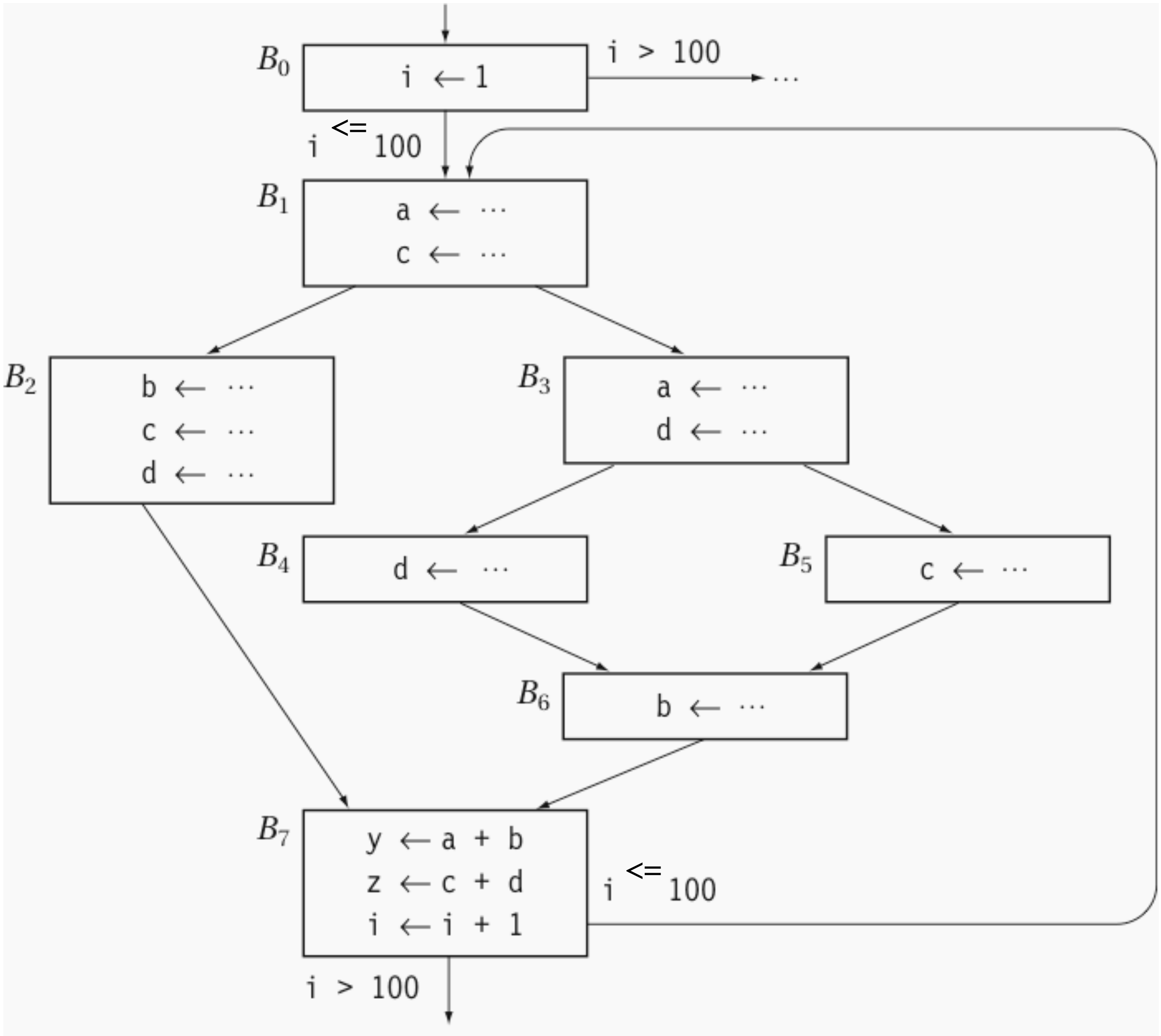
```
for each block b
   UEVAR(b) = Ø
   VARKILL(b) = Ø
   for i=1 to number of instr in b
      (assuming inst I is "x= y op z")
        if y ∉VARKILL(b) then
           UEVAR(b) = UEVAR(b) ∪ {y}
        if z ∉VARKILL(b) then
           UEVAR(b) = UEVAR(b) ∪ {z}
        VARKILL(b) = VARKILL(b) ∪ {x}
```

// update LiveOut version2

```
set LIVEOUT(bi) to Ø for all blocks
changed = true
while (changed)
   changed = false
   for i = 1 to N (number of blocks)
      recompute LIVEOUT(i)
      if LIVEOUT(i) changed then
         changed = true
```

$$LIVEOUT[B] = \bigcup_{B' \in succ(B)} ((LIVEOUT[B'] - VARKILL(B')) \bigcup UEVAR(B'))$$

Example



B0
i ← 1
i > 100
...
i <= 100
B1
a ← ...
c ← ...
B2
b ← ...
c ← ...
d ← ...
B3
a ← ...
d ← ...
B4
d ← ...
B5
c ← ...
B6
b ← ...
B7
y ← a + b
z ← c + d
i ← i + 1
i <= 100
i > 100
...

Example (cont.)

	B0	B1	B2	B3	B4	B5	B6	B7
UEVar	Ø	Ø	Ø	Ø	Ø	Ø	Ø	a,b,c,d,i
VarKill	i	a, c	b, c, d	a, d	d	c	b	y, z, i

Example (with update LiveOut version2)

Can the algorithm converge in fewer iterations?

LiveOut (b)

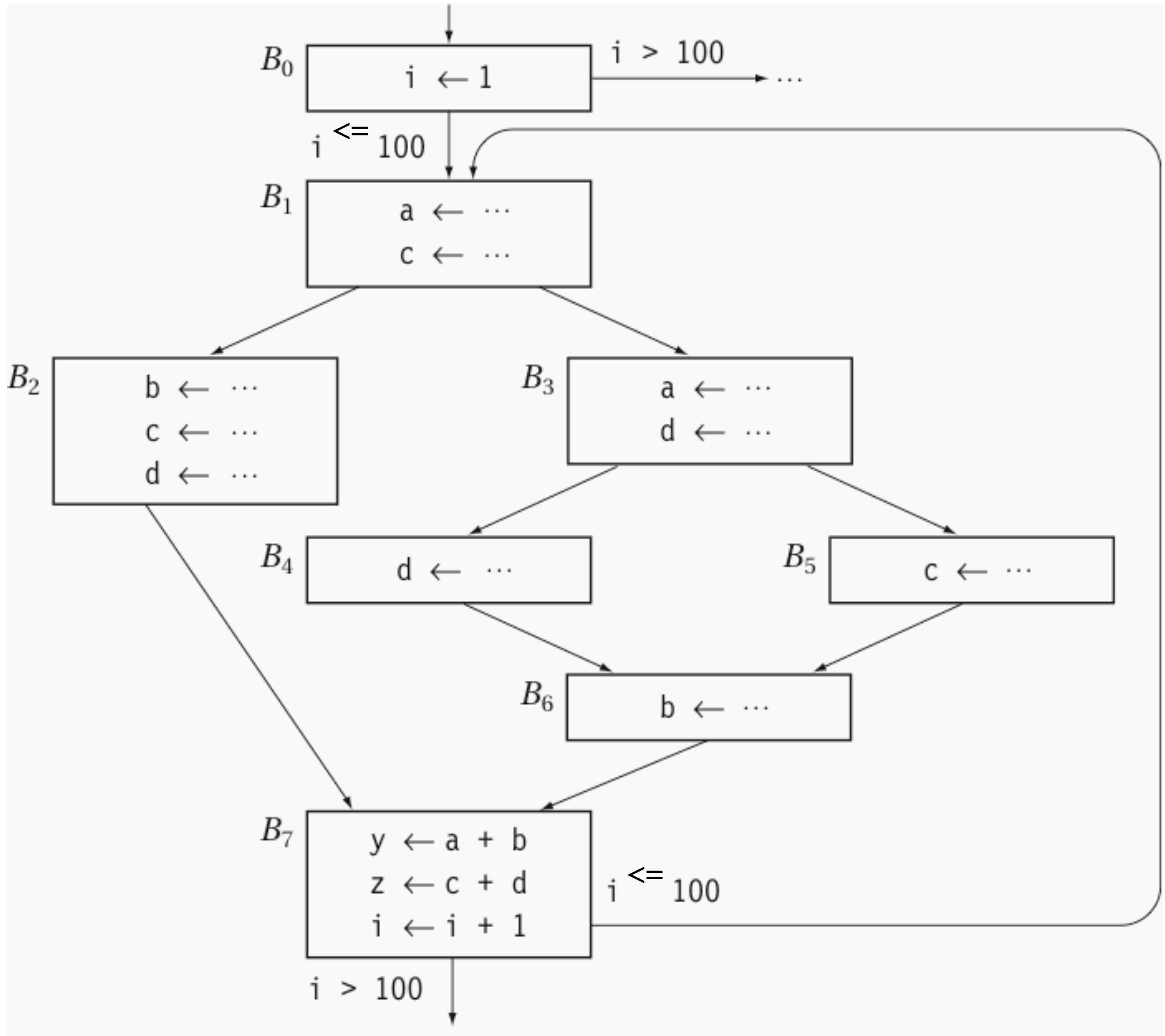
iteration	B0	B1	B2	B3	B4	B5	B6	B7
0	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø
1	Ø	Ø	a,b,c,d,i	Ø	Ø	Ø	a,b,c,d,i	Ø
2	Ø	a,i	a,b,c,d,i	Ø	a,c,d,i	a,c,d,i	a,b,c,d,i	i
3	i	a,i	a,b,c,d,i	a,c,d,i	a,c,d,i	a,c,d,i	a,b,c,d,i	i
4	i	a,c,i	a,b,c,d,i	a,c,d,i	a,c,d,i	a,c,d,i	a,b,c,d,i	i
5	i	a,c,i	a,b,c,d,i	a,c,d,i	a,c,d,i	a,c,d,i	a,b,c,d,i	i

$$LIVEOUT[B] = \bigcup_{B' \in succ(B)} ((LIVEOUT[B'] - VARKILL(B')) \bigcup UEVAR(B'))$$

$$LIVEOUT[B] = \bigcup_{B' \in succ(B)} ((LIVEOUT[B'] - VARKILL(B')) \bigcup UEVAR(B'))$$

Preorder: parents first.

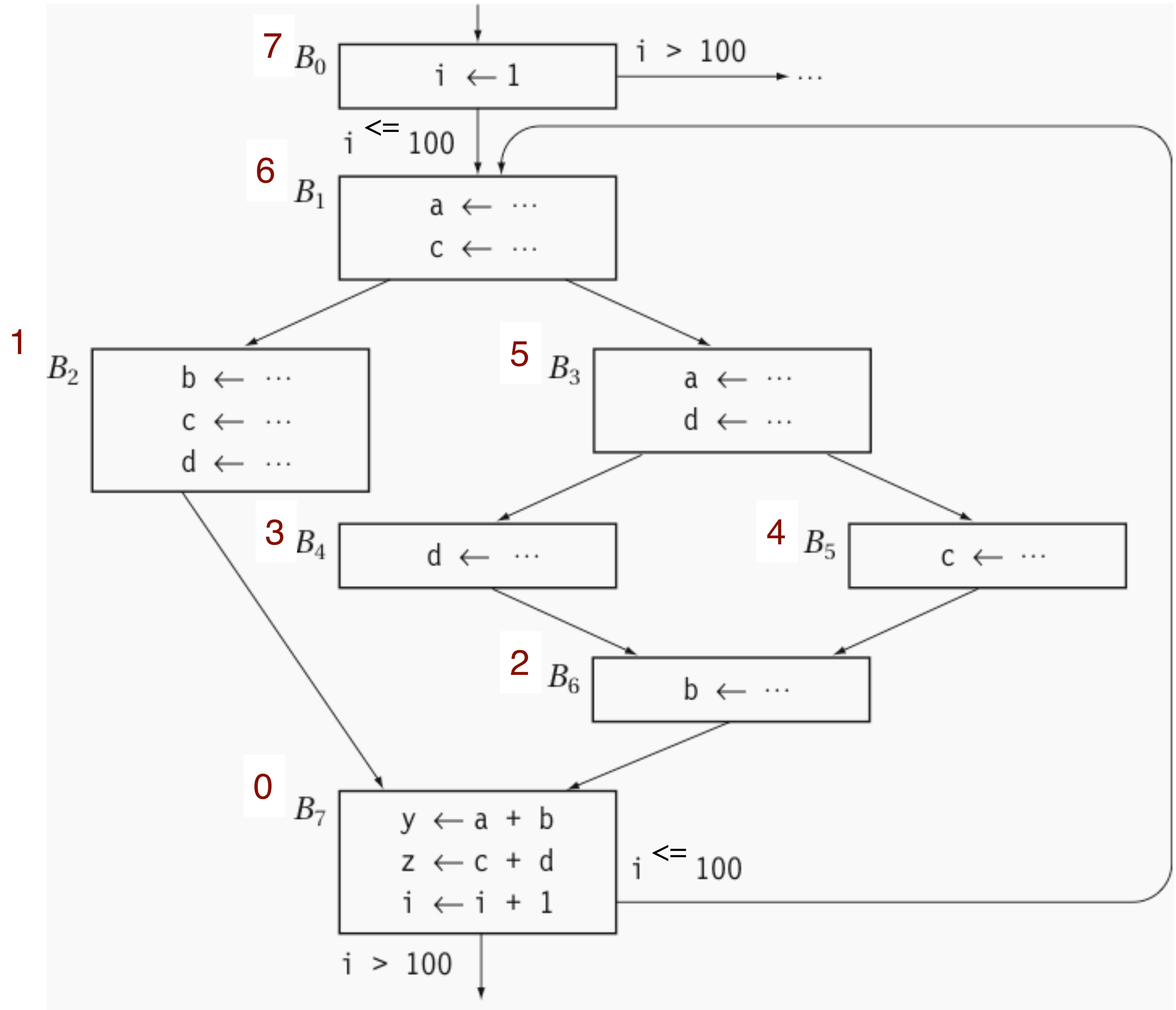
w/o considering backedges.



$$LIVEOUT[B] = \bigcup_{B' \in succ(B)} ((LIVEOUT[B'] - VARKILL(B')) \bigcup UEVAR(B'))$$

Postorder: children first.

w/o considering backedges.



Algorithm

// Get initial sets

```
for each block b
   UEVAR(b) = Ø
   VARKILL(b) = Ø
   for i=1 to number of instr in b
      (assuming inst I is "x= y op z")
        if y ∉VARKILL(b) then
           UEVAR(b) = UEVAR(b) ∪ {y}
        if z ∉VARKILL(b) then
           UEVAR(b) = UEVAR(b) ∪ {z}
          VARKILL(b) = VARKILL(b) ∪ {x}
```

// update LiveOut version2

```
set LIVEOUT(bi) to Ø for all blocks
changed = true
while (changed)
   changed = false
   for i = 1 to N
     // different orders could be used
      recompute LIVEOUT(i)
      if LIVEOUT(i) changed then
         changed = true
```

$$LIVEOUT[B] = \bigcup_{B' \in succ(B)} ((LIVEOUT[B'] - VARKILL(B')) \bigcup UEVAR(B'))$$

Postorder (5 iterations becomes 3)

iteration	B0	B1	B2	B3	B4	B5	B6	B7
0	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø
1	i	a,c,i	a,b,c,d,i	a,c,d,i	a,c,d,i	a,c,d,i	a,b,c,d,i	Ø
2	i	a,c,i	a,b,c,d,i	a,c,d,i	a,c,d,i	a,c,d,i	a,b,c,d,i	i
3	i	a,c,i	a,b,c,d,i	a,c,d,i	a,c,d,i	a,c,d,i	a,b,c,d,i	i

Order

Parent relation does not consider backedges.

- **Preorder**: visit parents before children.
  - also called reverse postorder
- **Postorder**: visit children before parents.

- Forward problem (e.g., AVAIL):
  - A node needs the info of its predecessors.
  - Preorder on CFG.
- Backward problem (e.g., LIVEOUT):
  - A node needs the info of its successors.
  - Postorder on CFG.

Comparison with AVAIL

- Common
  - Three steps
  - Fixed-point algorithm finds solution
- Differences
  - AVAIL: domain is a set of expressions

    LIVEOUT: domain is a set of variables

    Domain
  - AVAIL: forward problem

    LIVEOUT: backward problem

    Direction
  - AVAIL: intersection of all paths (all path problem)
    - Also called Must Problem

    LIVEOUT: union of all paths (any path problem)

    Also called May Problem

    May/Must

Popular data flow analysis

	Domain	Direction	Uses
AVAIL	Expressions	Forward	GCSE
LIVEOUT	Variables	Backward	Register alloc. Detect uninit. Construct SSA Useless-store Elim.
VERYBUSY	Expressions	Backward	Hoisting
CONSTANT	Pairs <v,c>	Forward	Constant folding
REACHES	Definition Points	Forward	Def-use chain for dead code elimination etc.

Very Busy Expressions

- VERYBUSY(b) contains expressions that are very busy at <u>end</u> of b
- UEEXPR(b): up exposed expressions (i.e. expressions defined in b and not subsequently killed in b)
- EXPRKILL(b): killed expressions

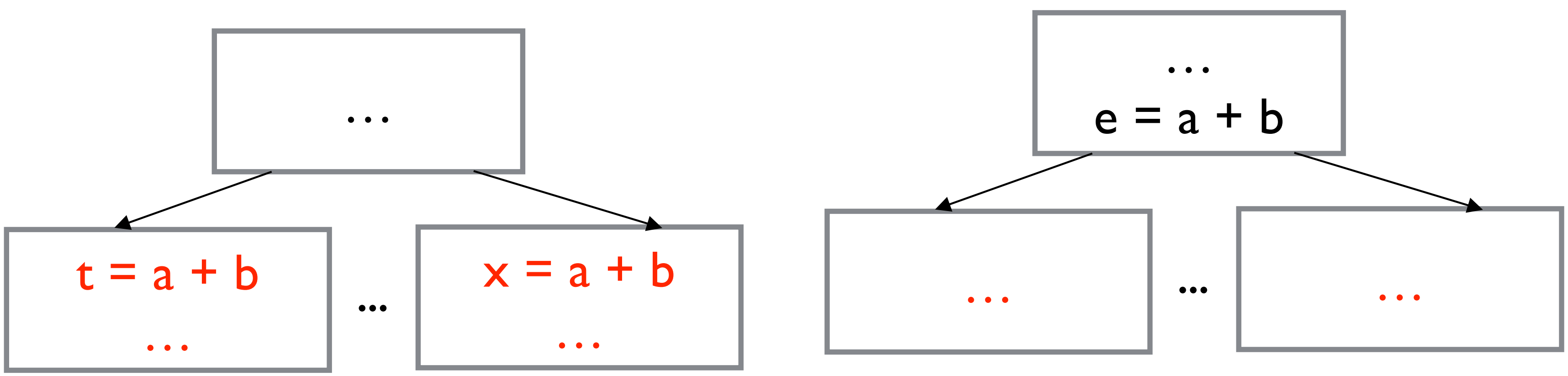
A backward flow problem, domain is the set of expressions

$$\text{VERYBUSY}(b) = \cap_{s \in succ(b)} \text{UEEXPR}(s) \cup (\text{VERYBUSY}(s) \cap \overline{\text{EXPRKILL}(s)})$$

$$\text{VERYBUSY}(n_f) = \emptyset$$

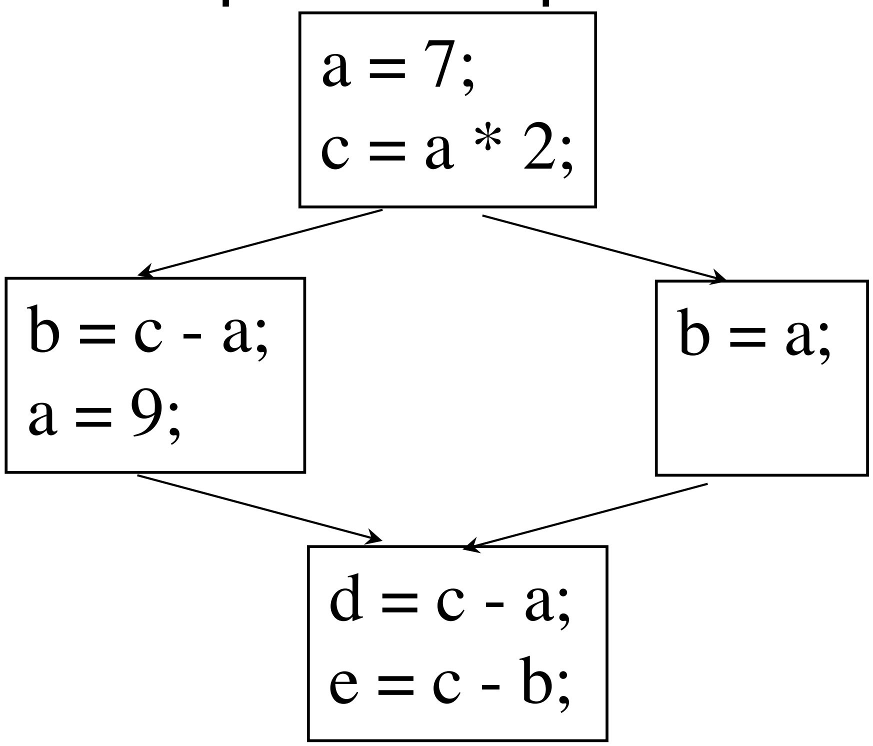
Very Busy Expressions

- Def: e is a very busy expression at the exit of block b if
  - e is evaluated and used along every path that leaves b, and
  - evaluating e at the end of b produces the same result
- useful for code hoisting
- saves code space



Constant Propagation

- Def of a constant variable v at point p:
  - Along every path to p, v has same known value
- Specialize computation at p based on v's value



Constant Propagation:

Domain is the set of pairs $<v_i,c_i>$ where v_i is a variable and $c_i \in C$

$$\text{CONSTANTS}(b) = \wedge_{p \in preds(b)} \; f_p(\text{CONSTANTS}(p))$$

- $\wedge$ performs a pairwise meet on two sets of pairs
- $f_p(x)$ is a block specific function that models the effects of block p on the $<v_i,c_i>$ pairs in x

A forward flow problem, domain is the set of pairs $<v,c>$.

C: constants or $\perp$.

$\perp$: non-constant or unknown value

$\textit{CONSTANTS}(b) = \wedge_{p \in \text{preds}(b)} \; f_p(\textit{CONSTANTS}(p))$

Meet operation $<v, c_1> \wedge <v, c_2>$

- $<v, c_1>$ if $c_1 = c_2$, else $<v, \perp>$

$\perp$: non-constant or unknown value

Define fp with examples:

- If p has one statement then
  - $x \leftarrow y$ with $\text{CONSTANTS}(p) = \{\ldots<x,l_1>,\ldots<y,l_2>\ldots\}$
    then $f_p(\text{CONSTANTS}(p)) = \{\text{CONSTANTS}(p) - <x,l_1>\} \cup <x,l_2>$
  - $x \leftarrow y \text{ op } z$ with $\text{CONSTANTS}(p) = \{\ldots<x,l_1>,\ldots<y,l_2>\ldots,\ldots<z,l_3>\ldots\}$
    then $f_p(\text{CONSTANTS}(p)) = \{\text{CONSTANTS}(p) - <x,l_1>\} \cup <x,l_2 \text{ op } l_3>$
- If p has n statements then

  $f_p(\text{CONSTANTS}(p)) = f_n(f_{n-1}(f_{n-2}(\ldots f_2(f_1(\text{CONSTANTS}(p)))\ldots)))$

  where f_i is the function generated by the i^{th} statement in p

f_p interprets p over CONSTANTS

Reaching Definitions

- A definition of variable v at program point d reaches program point u if there exists a path of control flow edges from d to u that does not contain a definition of v.

$$\text{REACHES}(n) = \cup_{m \in \text{pred}(n)} \text{DEDEF}(m) \cup (\text{REACHES}(m) \cap \overline{\text{DEFKILL}(m)})$$

- REACHES(n): the set of variable definitions that reach the start of node n.
- DEDEF(n): the set of downward-exposed variable definitions in n.
  - i.e. their defined variables are not redefined before leaving n.
- DEFKILL(n): <u>all</u> definitions killed by a definition in n.
- A forward flow problem, domain is the definition point:

  the variable name + where it is defined (code position)

Def-Use Chains

Example

```
a ← 5
b ← 3
c ← b + 2
d ← a - 2
```

`d` is dead

It has no use

```
e ← a + b
e ← e + c
```

```
e ← 13
```

```
f ← 2 + e
Write f
```

Data-Flow Analysis Frameworks

- Generalizes and unifies data flow problems.
- Important components:
  - Direction D: forward or backward.
  - A *Semilattice:* a domain V and a *meet* operator $\wedge$ that captures the effect of path confluence.
  - A transfer function F(m): compute the effect of passing through a basic block and include function value at boundary conditions.

A *semilattice* is an algebra $\mathcal{S} = (S, *)$ satisfying, for all $x, y, z \in S$,

(1) $x * x = x$,
(2) $x * y = y * x$,
(3) $x * (y * z) = (x * y) * z$.

Examples

- (D, V, F, ^)
- LIVE
  - D: backward
  - V: all variables
  - F_m: $UEVAR(m) \cup (LIVEOUT(m) \cap \overline{VARKILL(m)})$; $LIVEOUT(n_f) = \phi$
  - ^ : ∪
- AVAIL
  - D: forward, V: all expressions
  - F_m: $DEEXPR(m) \cup (AVAIL(m) \cap \overline{EXPRKILL(m)})$; $AVAIL(n_o)=\phi$
  - ^: ∩

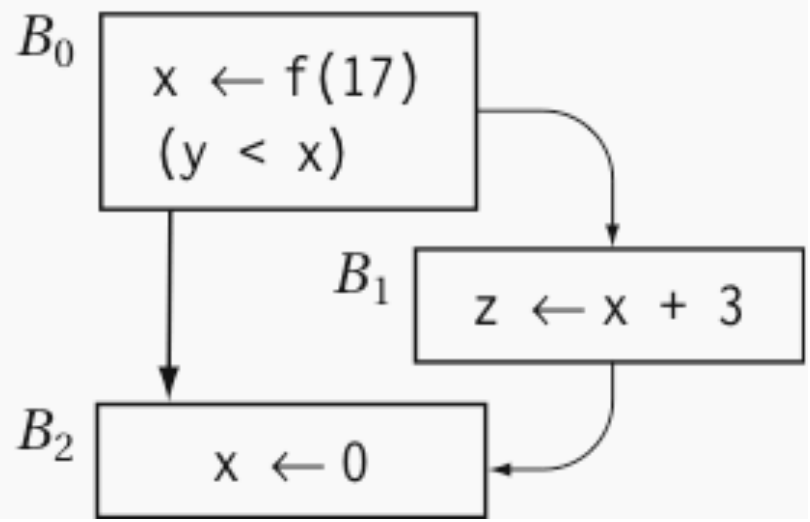
Why to Study Data Flow Analysis

- Data-flow analysis
  - A collection of techniques for compile-time reasoning about the run-time flow of values.
- Backbone of scalar optimizing compilers

Limitation of Data-Flow Analysis

- Imprecision from pointers, and procedure calls
- Assume all paths will be taken

```
x ← f(17)
if (y < x) then
   z ← x + 3
x ← 0
```



If y is always no less than x, x is not live before B2.
But data-flow analysis may not figure that out.

Summary

	Domain	Direction	Uses
AVAIL	Expressions	Forward	GCSE
LIVEOUT	Variables	Backward	Register alloc. Detect uninit. Construct SSA Useless-store Elim.
VERYBUSY	Expressions	Backward	Hoisting
CONSTANT	Pairs <v,c>	Forward	Constant folding
REACHES	Definition Points	Forward	Def-use chain for dead code elimination etc.