CS293S GCSE and Data Flow Analysis

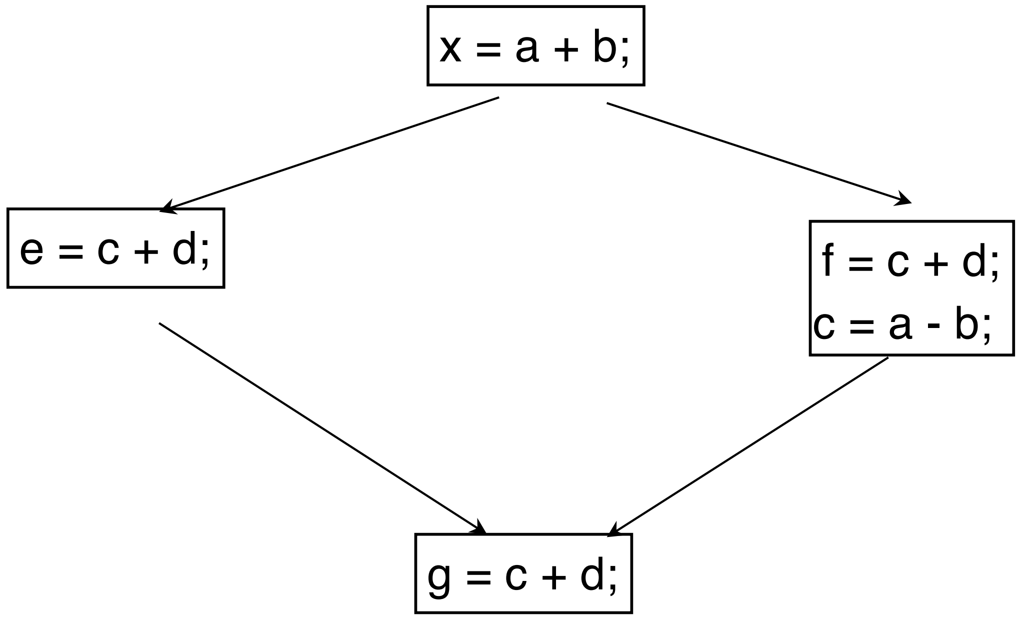
Yufei Ding

Review of Last Class

- Scope of optimization for redundancy elimination
  - Basic block -> Local value numbering
  - Extended basic block -> Superlocal value numbering (SVN)
  - Dominator -> Dominator-based value numbering (DVN)

- HW1 was out, due on 22-Oct.
- Paper Assignment out today. First review due in 3 weeks (3-Nov)

Examples with redundancy can not be eliminated?



Topics of This Class

- Global Common Subexpression Elimination (GCSE)
  - More close to DAG-based methods
  - Work on lexical notation instead of expression values.
  - Our first data flow analysis
- Other data flow analysis
  - The general framework
  - Live variable analysis
  - Reaching definition analysis

Global Common Subexpression Elimination (GCSE)

- The first **data-flow problem**
- A global method

Some Expression Sets

For each block b

Let **Avail(b)** be the set of expressions available on entry to b.

Let **ExprKill(b)** be the set of expressions killed in b.

i.e. one or more operands of the expression are redefined in b.

!!!! Must consider all expressions in the whole graph.

Let **DEExpr(b)** include the downward exposed expressions in b.

i.e. expressions defined in b and not subsequently killed in b

Formula to Compute AVAIL

- Now, AVAIL(b) can be defined as:

$$\text{AVAIL}(b) = \cap_{x \in pred(b)} (\text{DEEXPR}(x) \cup (\text{AVAIL}(x) \cap \overline{\text{EXPRKILL}(x)}\,))$$

- preds(b) is the set of b's predecessors in the control-flow graph. (Again, a predecessor is an immediate parent, not including other ancestors.)

Computing Available Expressions

The Big Picture

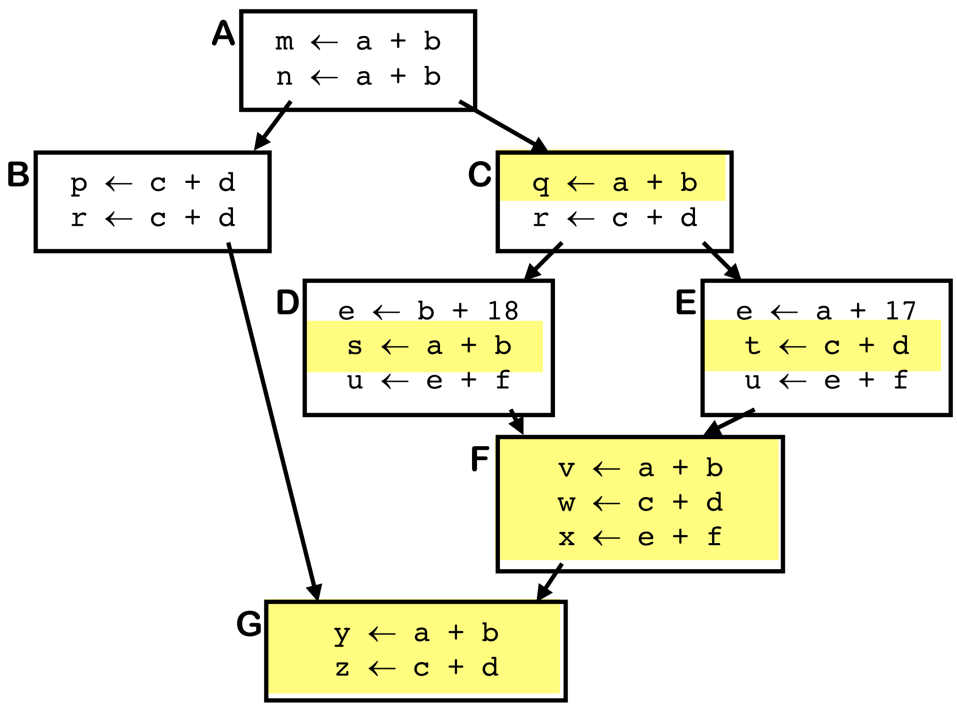
1. Build a control-flow graph
2. Gather the initial data: DEEXPR(b) & EXPRKILL(b)
3. Propagate information around the graph, evaluating the equation

Works for loops through an iterative algorithm: finding the fixed-point.

All data-flow problems are solved, essentially, this way.

Making Theory Concrete

Computing AVAIL for the example



	A	B	C	D	E	F	G
DEEXPR	a+b	c+d	a+b,c+d	b+18,a+b,e+f	a+17,c+d,e+f	a+b,c+d,e+f	a+b,c+d
EXPRKILL	{ }	{ }	{ }	e+f	e+f	{ }	{ }

AVAIL(A) = Ø

AVAIL(B) = {a+b} ∪ (Ø ∩ all)
= {a+b}

AVAIL(C) = {a+b}

AVAIL(D) = {a+b,c+d} ∪ ({a+b} ∩ all)
= {a+b,c+d}

AVAIL(E) = {a+b,c+d}

AVAIL(F) = [{b+18,a+b,e+f} ∪
({a+b,c+d} ∩ {all - e+f})]
∩ [{a+17,c+d,e+f} ∪
({a+b,c+d} ∩ {all - e+f})]
= {a+b,c+d,e+f}

AVAIL(G) = [{c+d} ∪ ({a+b} ∩ all)]
∩ [{a+b,c+d,e+f} ∪
({a+b,c+d,e+f} ∩ all)]
= {a+b,c+d}

Computing Available Expressions

- First step is to compute DEEXPR & EXPRKILL

assume a block b with operations $o_1, o_2, \ldots, o_k$

Many data-flow problems have initial information that costs less to compute

```
VARKILL ← Ø
DEEXPR(b) ← Ø

for i = k to 1
    assume oi is "x ← y + z"
    add x to VARKILL
    if (y ∉ VARKILL) and (z ∉ VARKILL) then
        add "y + z" to DEEXPR(b)
```

Backward through block

O(k) steps

```
EXPRKILL(b) ← Ø

For each expression e
    for each variable v ∈ e
        if v ∈ VARKILL(b) then
            EXPRKILL(b) ← EXPRKILL(b) ∪ {e}
```

O(N) steps

N is # operations

Computing Available Expressions

The worklist iterative algorithm

Worklist $\leftarrow$ { all blocks, b_i }

while (Worklist $\neq$ Ø)
remove a block b from Worklist
recompute AVAIL(b) as

$$\text{AVAIL}(b) = \cap_{x \in pred(b)} (\text{DEEXPR}(x) \cup (\text{AVAIL}(x) \cap \overline{\text{EXPRKILL}(x)}))$$

if ??? then
Worklist $\leftarrow$???

Computing Available Expressions

The worklist iterative algorithm

Worklist $\leftarrow$ { all blocks, b_i }

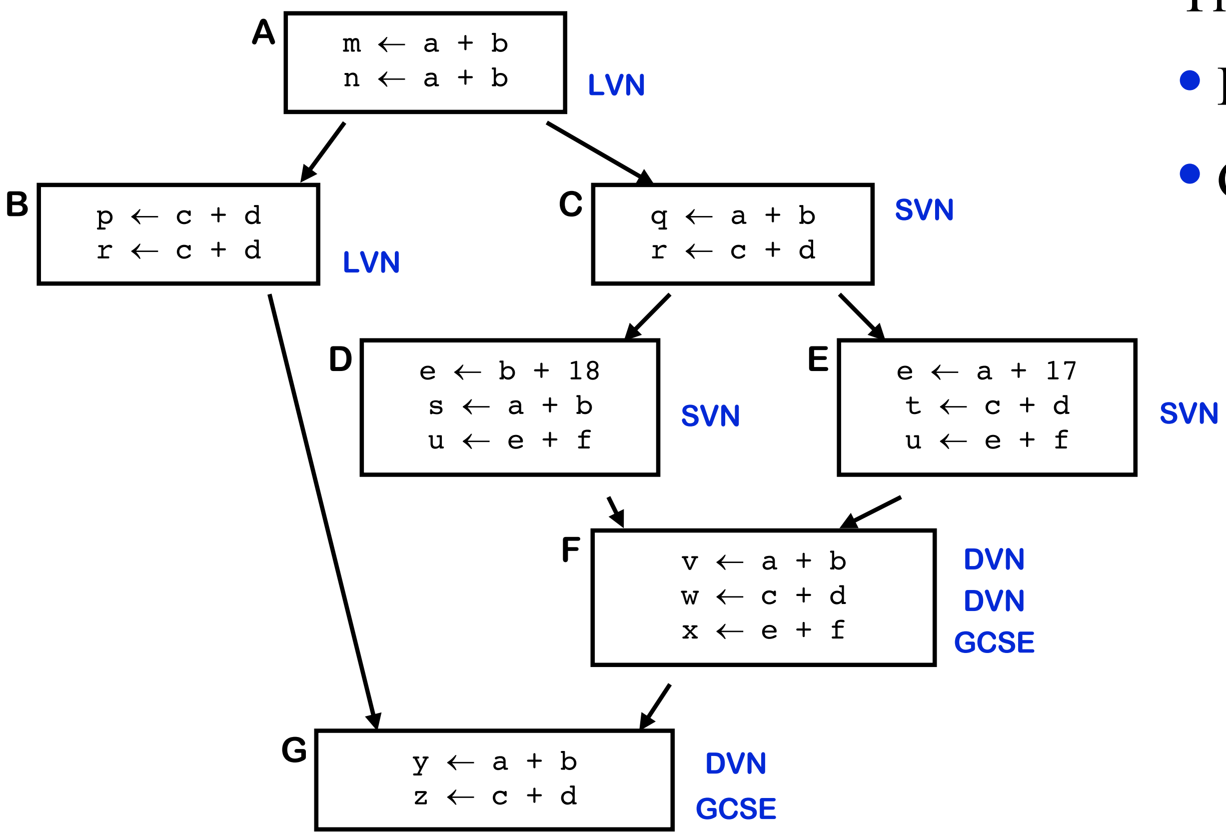
while (Worklist $\neq$ Ø)
remove a block b from Worklist
recompute AVAIL(b) as

$$\text{AVAIL}(b) = \cap_{x \in pred(b)} \left(\text{DEEXPR}(x) \cup \left(\text{AVAIL}(x) \cap \overline{\text{EXPRKILL}(x)}\right)\right)$$

if AVAIL(b) changed then
Worklist $\leftarrow$ Worklist $\cup$ successors(b)

- **Finds fixed point solution to equation for AVAIL**
- **That solution is unique**

Comparison



The VN methods are ordered

- LVN $\leq$ SVN $\leq$ DVN
- GCSE is different
  - Based on names, not value
  - But for this particular example: DVN $\leq$ GCSE
    - Not always!!!!

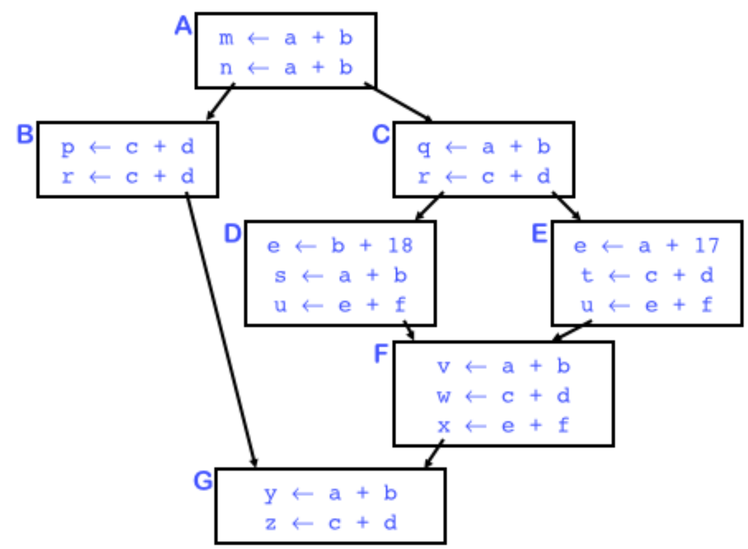
Redundancy Elimination Wrap-up

Conclusions

- Redundancy elimination has some depth & subtlety
- Various algorithms and optimization scopes

DVN is probably the method of choice

- Results quite close to the global methods (± 1%)
- Cost is low



Data-flow Analysis

- Data-flow analysis is a collection of techniques for compile-time reasoning about run-time flow of values
- Almost always involves building a graph
  - Problems are trivial on a basic block
  - Global problems -> control-flow graph (or derivative)
  - Whole program problems -> call graph (or derivative)
- Usually formulated as a set of simultaneous equations

GCSE: Computing Available Expressions

The Big Picture

1. Gather the initial data: DEEXPR(b) & EXPRKILL(b)
2. Propagate information around the graph, evaluating the equation

$$\underline{\text{AVAIL}(b)} = \cap_{x \in pred(b)} \underline{(\text{DEEXPR}(x) \cup (\text{AVAIL}(x) \cap \overline{\text{EXPRKILL}(x)}))}$$

Entry point of block b **Exit point of block x**

Works for loops through an iterative algorithm: finding the fixed-point.

All data-flow problems are solved, essentially, this way.

Other 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.

Live Variables

- A variable v is live at a point p if there is a path from p to a use of v, and that path does not contain a redefinition of v

- Example: I: *a* <- *b* + *c*
  - A statement/instruction I is a definition of a variable *v* if it may write to *v*. def[I] = *a*
  - A statement is a use of variable *v* if it may read from *v*. use[I] = {*b*, *c*}

Usage of Live Variables

- Detect references to uninitialized variables
- Detect defined but not used variables
  - Global register allocation
  - useless-store elimination
  - Improve SSA construction

Live Variables at Special Points

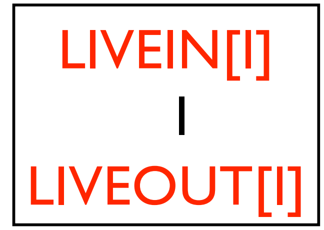
- For an instruction I
  - LIVEIN[I]: live variables at program point before I
  - LIVEOUT[I]: live variables at program point after I

- For a basic block B
  - LIVEIN[B]: live variables at the entry point of B
  - LIVEOUT[B]: live variables at the exit point of B

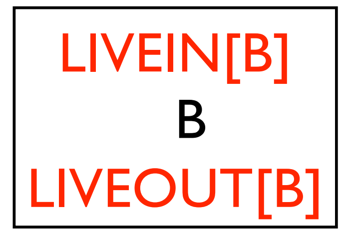
- If I = first instruction in B, then LIVEIN[B] = LIVEIN[I]
- If I = last instruction in B, then LIVEOUT[B] = LIVEOUT[I]

How to Compute Liveness?

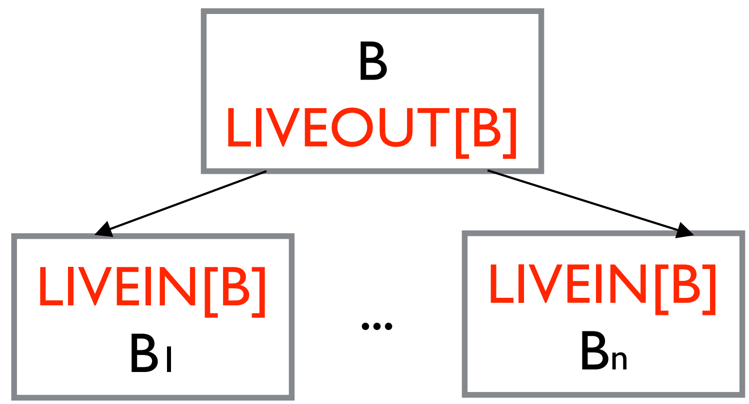
- Question 1: for each instruction I, what is the relation between LIVEIN[I] and LIVEOUT[I]?



- 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]?



Part 1: Analyze Instructions

- Question: what is the relation between the sets of live variables before and after an instruction I?

LIVEIN[I]
I
LIVEOUT[I]

Examples:

LIVEIN[I] = {y,z} x = y+z; LIVEOUT[I] = {z}	LIVEIN[I] = {y,z,t} x = y+z; LIVEOUT[I] = {x,t}	LIVEIN[I] = {x,t} x = x+1; LIVEOUT[I] = {x,t}

… is there a general rule?

Analyze Instructions

- Two Rules:
  - Each variable live after I is also live before I, unless I defines (writes) it.
  - Each variable that I uses (reads) is also live before instruction I

- Mathematically:
  LIVEIN[I] = (LIVEOUT[I] – def[I]) ∪ use[I]
  where: def[I] = variables defined (written) by instruction I
  use[I] = variables used (read) by instruction I

  - The information flows **backward!**

Analyze block

- Example: block B with three instructions I1, I2, I3:
  - Live1 = LIVEIN[B] = LIVEIN[I1]
  - Live2 = LIVEOUT[I1] = LIVEIN[I2]
  - Live3 = LIVEOUT[I2] = LIVEIN[I3]
  - Live4 = LIVEOUT[I3] = LIVEOUT[B]
- Relation between Live sets:
  - Live1 = (Live2-{x}) ∪ {y}
  - Live2 = (Live3-{y}) ∪ {x, z}
  - Live3 = (Live4-{t}) ∪ {d}

Block B

	Live1
I1	x = y + 1
	Live 2
I2	y = x * z
	Live 3
I3	t = d
	Live 4

Live1=(Live4-{x, y, t}) ∪ {d, z, y}

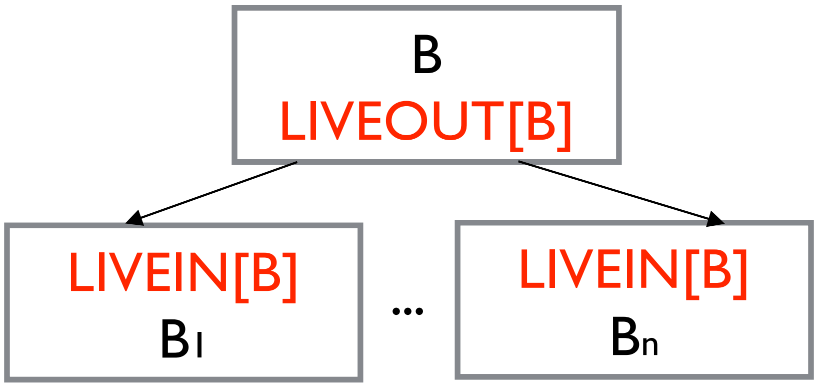
Analyze Block

- Two Rules:
  - Each variable live after B is also live before B, unless B defines (writes) it.
  - Each variable v that B uses (reads) before any redefinition in B is also live before B

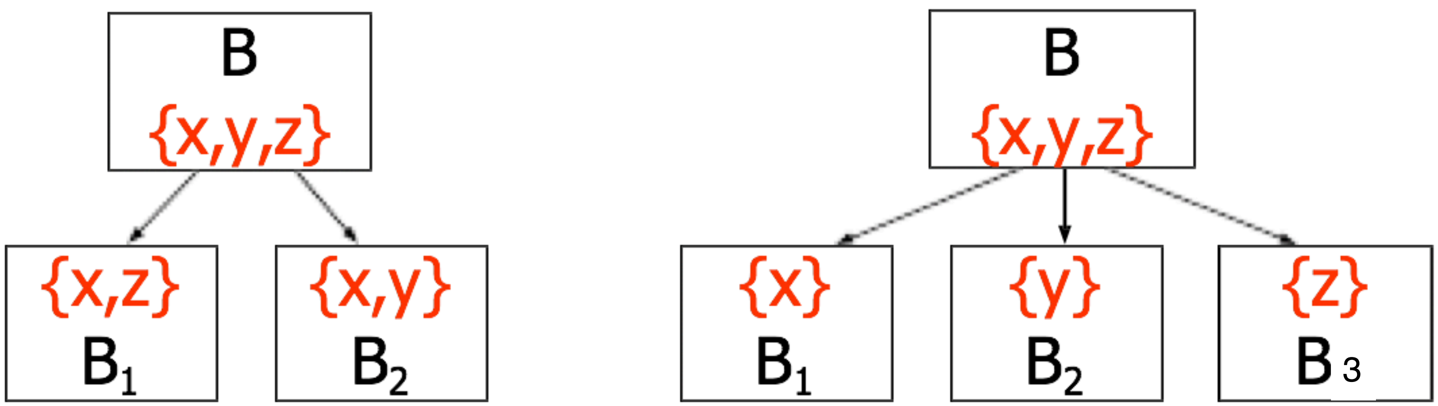
- Mathematically:
  LIVEIN[B] = (LIVEOUT[B] – VARKILL(B)) ∪ UEVAR(B)
  where:
  - VARKILL(B) = variables that are defined in B
  - UEVAR(B) variables that are used in B before any redefinition in B, i.e., <u>u</u>pward-<u>e</u>xposed <u>var</u>iables

Analyze CFG

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



- Example:



- General rule?

Analyze CFG

- Rule: A variables is live at end of block B if it is live at the beginning of one (or more) successor blocks
- Mathematically:

$$LIVEOUT[B] = \bigcup_{B' \in succ(B)} LIVEIN[B']$$

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

- Again, information flows backward: from successors B' of B to basic block

Equations for Live Variables

- 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.

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

Note: A-B = A ∩ $\bar{B}$

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

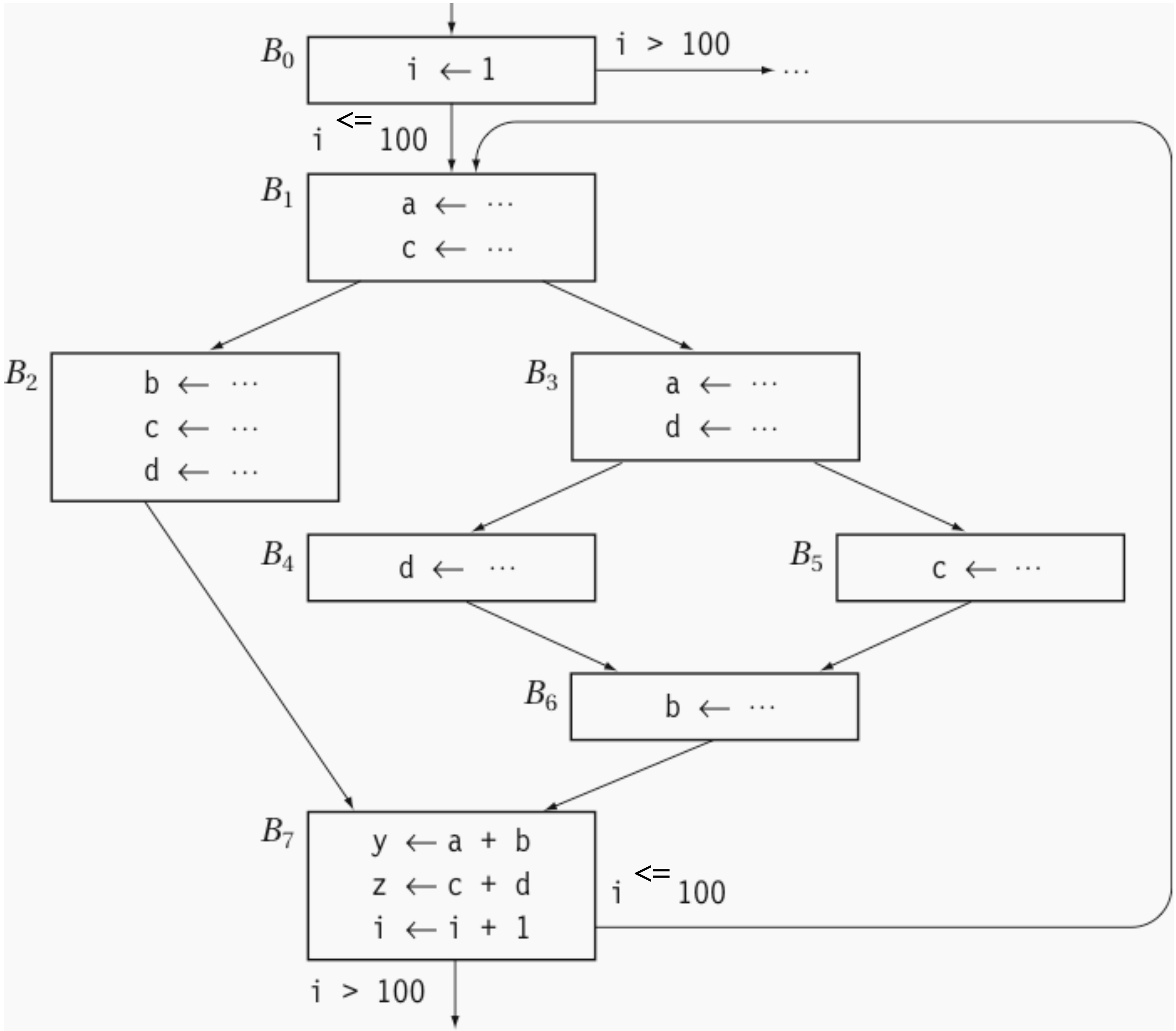
```
for each block b
    UEVAR(b) = Ø
    VARKILL(b) = Ø
    for i=1 to number of instr in b
        (assuming inst I is "x= y op z")
          if y ∉VARKILL(b) then
              UEVAR(b) = UEVAR(b) ∪ {y}
          if z ∉VARKILL(b) then
              UEVAR(b) = UEVAR(b) ∪ {z}
          VARKILL(b) =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

Can the algorithm converge in fewer iterations?

LiveOut (b)

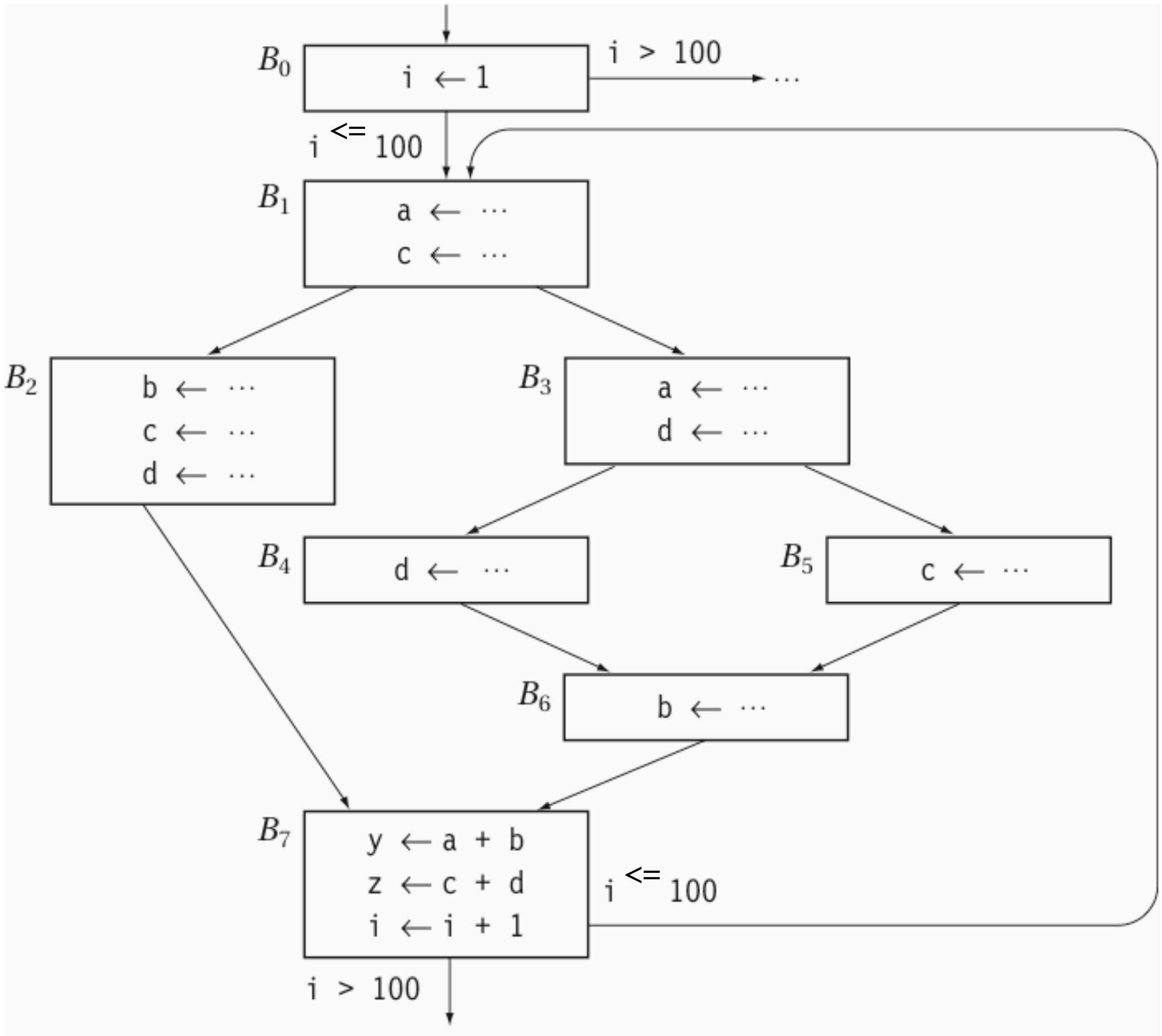
iteration	B0	B1	B2	B3	B4	B5	B6	B7
0	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø
1	Ø	Ø	a,b,c,d,i	Ø	Ø	Ø	a,b,c,d,i	Ø
2	Ø	a,i	a,b,c,d,i	Ø	a,c,d,i	a,c,d,i	a,b,c,d,i	i
3	i	a,i	a,b,c,d,i	a,c,d,i	a,c,d,i	a,c,d,i	a,b,c,d,i	i
4	i	a,c,i	a,b,c,d,i	a,c,d,i	a,c,d,i	a,c,d,i	a,b,c,d,i	i
5	i	a,c,i	a,b,c,d,i	a,c,d,i	a,c,d,i	a,c,d,i	a,b,c,d,i	i

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

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

Preorder: parents first.

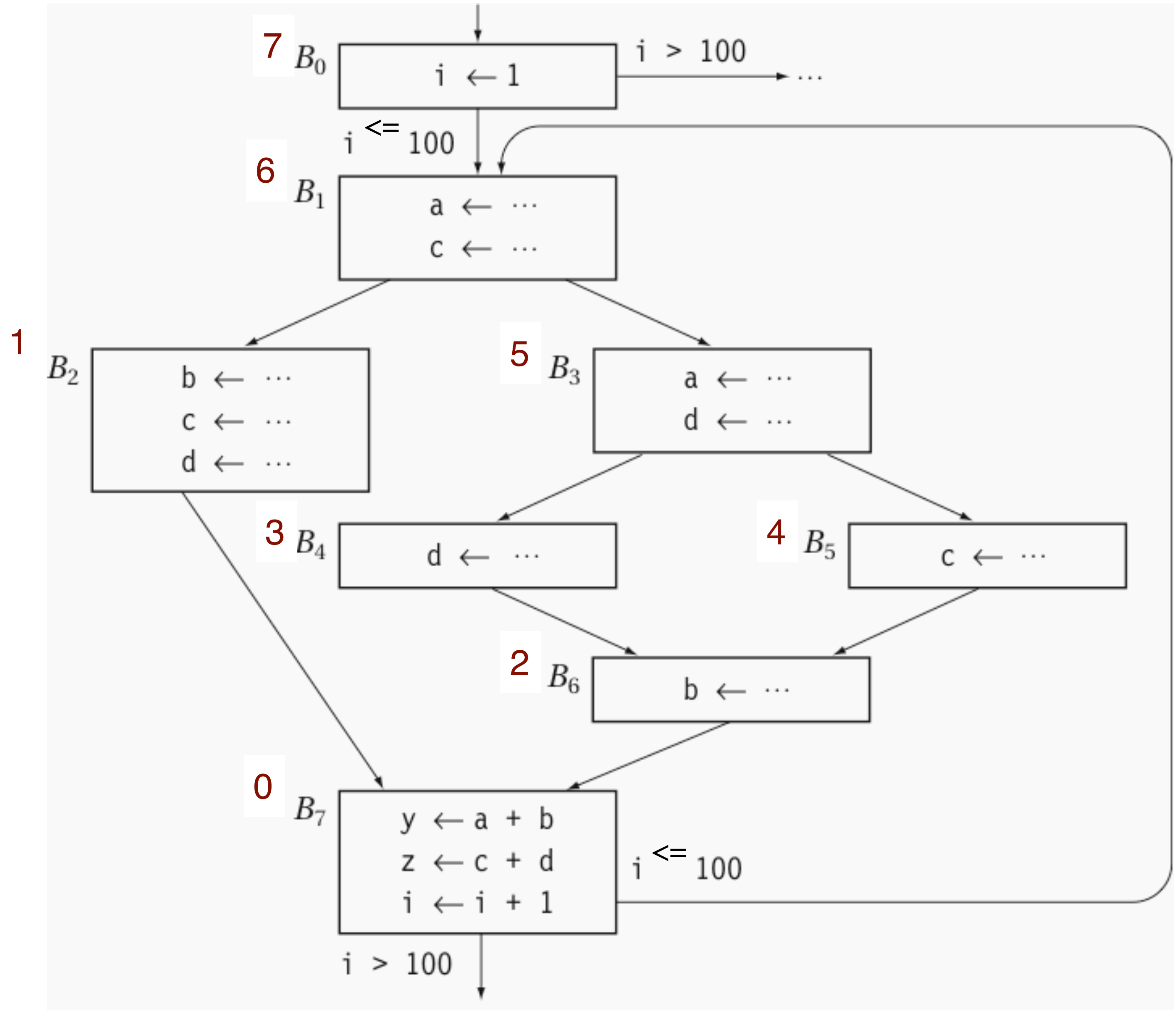
w/o considering backedges.



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

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(b_i) 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