CSE 401/M501 – Compilers

Dataflow Analysis

Spring 2022

Administrivia - Schedule

- Codegen project due Thursday, 5/26
- Project Report due Tuesday, 5/31

Except M501

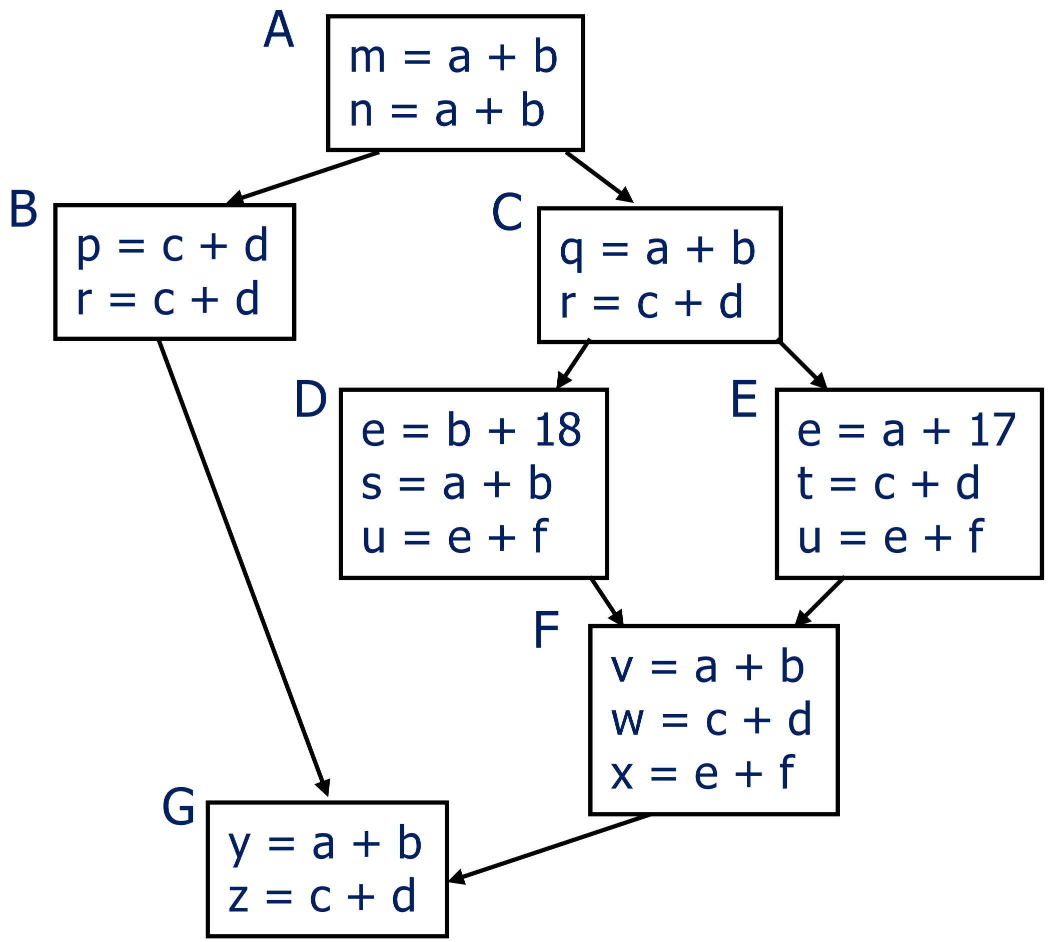
- HW4 (data flow/SSA) out soon, due 6/2
  - *Limited late days!*
- Final exam is Tue. 6/7, 2:30 12/14

Agenda

- Dataflow analysis: a framework and algorithm for many common compiler analyses
- Initial example: dataflow analysis for common subexpression elimination
- Then: other analysis problems that work in the same framework
- Some of these are the same analysis and optimizations we've seen, but more formally and with details

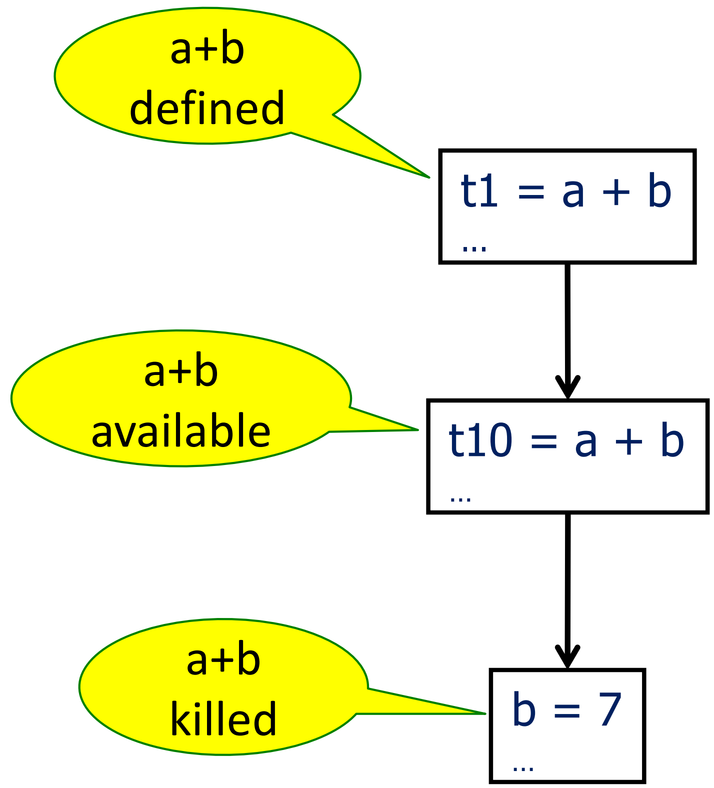
Common Subexpression Elimination

- Goal: use dataflow analysis to find common subexpressions
- Idea: calculate *available expressions* at beginning of each basic block
- Avoid re-evaluation of an available expression – use a copy operation
  - Simple inside a single block; more complex dataflow analysis used across bocks



“Available” and Other Terms

- An expression *e* is *defined* at point *p* in the CFG if its value is computed at *p*
  - Sometimes called *definition site*
- An expression *e* is *killed* at point *p* if one of its operands is defined at *p*
  - Sometimes called *kill site*
- An expression *e* is *available* at point *p* if every path leading to *p* contains a prior definition of *e* and *e* is not killed between that definition and *p*



Available Expression Sets

To compute available expressions, for each block *b*, define

- AVAIL(*b*) – the set of expressions available on entry to *b*
- NKILL(*b*) – the set of expressions <u>not killed</u> in *b*
  - all expressions in the program *except* those killed in *b*
- DEF(*b*) – the set of expressions defined in *b* and not subsequently killed in *b*

Computing Available Expressions

- The set of expressions available on *entry* to b is the set of expressions that are available on *exit* from *every* predecessor x of b

  $$\text{AVAIL}_{in}(b) = \bigcap_{x \in \text{preds}(b)} \text{AVAIL}_{out}(x),$$

  where preds(b) is the set of b's predecessors in the CFG
- The expressions available on exit from block b are those defined in b or available on entry to b and not killed in b, so

  $$\text{AVAIL}(b) = \bigcap_{x \in \text{preds}(b)} (\text{DEF}(x) \cup (\text{AVAIL}(x) \cap \text{NKILL}(x)))$$
- This gives a system of simultaneous equations – a dataflow problem

Computing Available Expressions

- Big Picture
  - Build control-flow graph
  - Calculate initial local data – DEF(*b*) and NKILL(*b*)
    - This only needs to be done once for each block *b* and depends only on the statements in *b*
  - Iteratively calculate AVAIL(*b*) by repeatedly evaluating equations until nothing changes
    - Another fixed-point algorithm

Computing DEF

For each block *b* with operations $o_1, o_2, \ldots, o_n$

KILLED = ∅ // *variables* killed (later) in *b*, not expressions

DEF(*b*) = ∅

for *k* = *n* to 1 // note: working back to front

assume o_k is "x = y + z"

add x to KILLED

if (y ∉ KILLED and z ∉ KILLED)

add "y + z" to DEF(b) // i.e., neither y nor z killed
// after this point in *b*

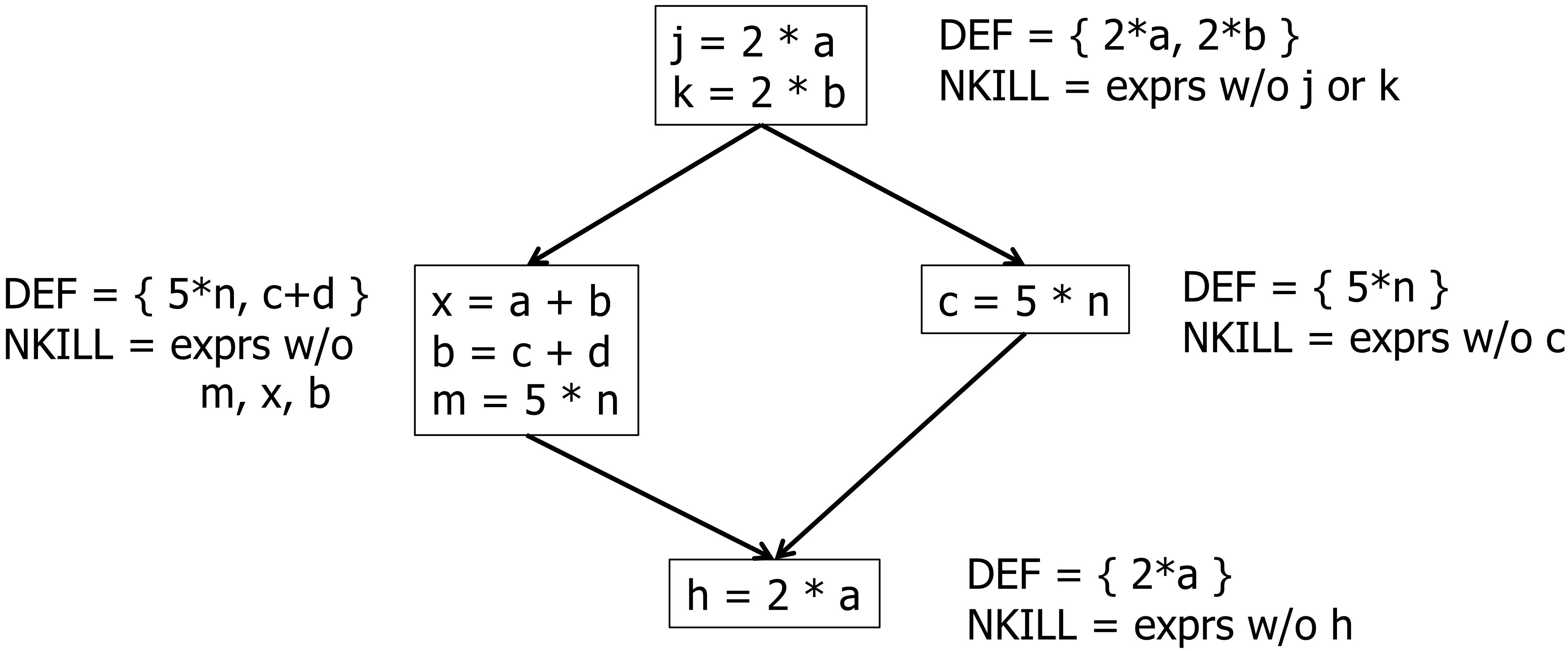
…

Computing NKILL

After computing DEF and KILLED for a block *b*, compute set of all expressions in the program not killed in *b*

NKILL(*b*) = { all expressions }
for each expression *e*
  for each variable $v \in e$
    if $v \in$ KILLED(*b*) then
      NKILL(*b*) = NKILL(*b*) – { *e* }

Example: Compute DEF and NKILL



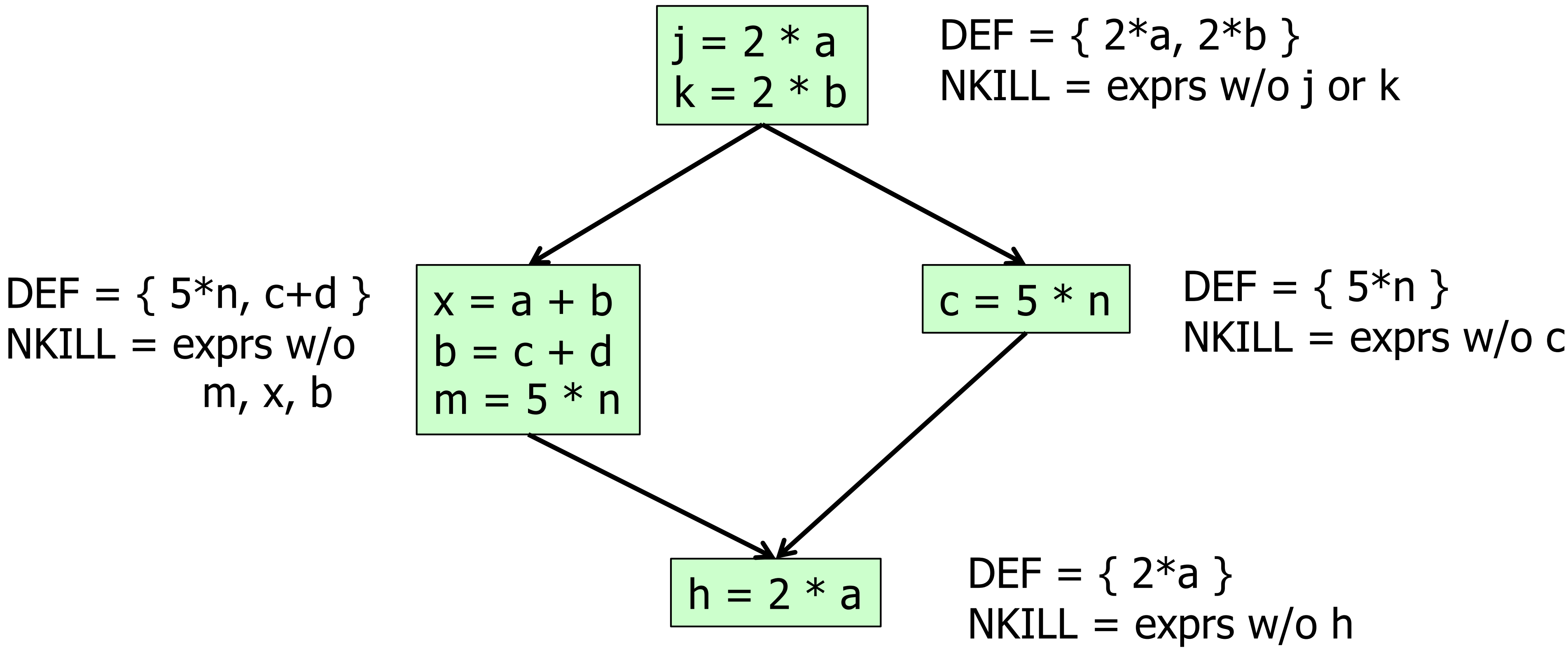
Computing Available Expressions

Once DEF(b) and NKILL(b) are computed for all blocks b

Worklist = { all blocks b_k }
AVAIL(b_k) = $\emptyset$ for all blocks b_k
while (Worklist $\neq \emptyset$)
    remove a block b from Worklist
    recompute AVAIL(b)
    if AVAIL(b) changed
        Worklist = Worklist $\cup$ successors(b)

Example: Find Available Expressions

$$\mathrm{AVAIL}(b) = \bigcap_{x \in \mathrm{preds}(b)} \left(\mathrm{DEF}(x) \cup \left(\mathrm{AVAIL}(x) \cap \mathrm{NKILL}(x)\right)\right)$$

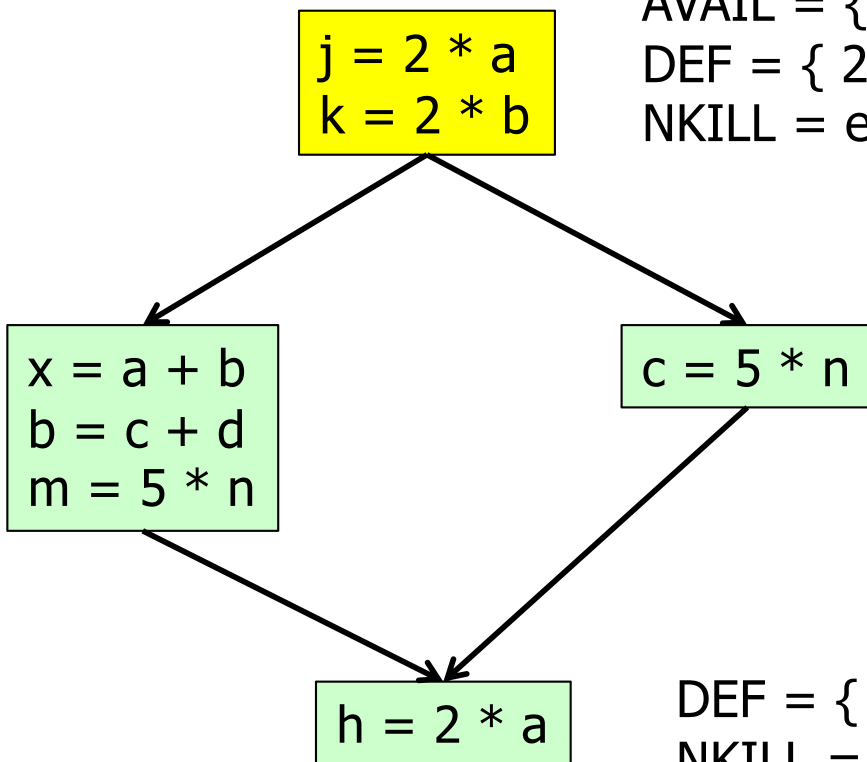


= in worklist

= processing

Example: Find Available Expressions

$$AVAIL(b) = \bigcap_{x \in preds(b)} (DEF(x) \cup (AVAIL(x) \cap NKILL(x)))$$



AVAIL = { }
DEF = { 2*a, 2*b }
NKILL = exprs w/o j or k

DEF = { 5*n, c+d }
NKILL = exprs w/o m, x, b

DEF = { 5*n }
NKILL = exprs w/o c

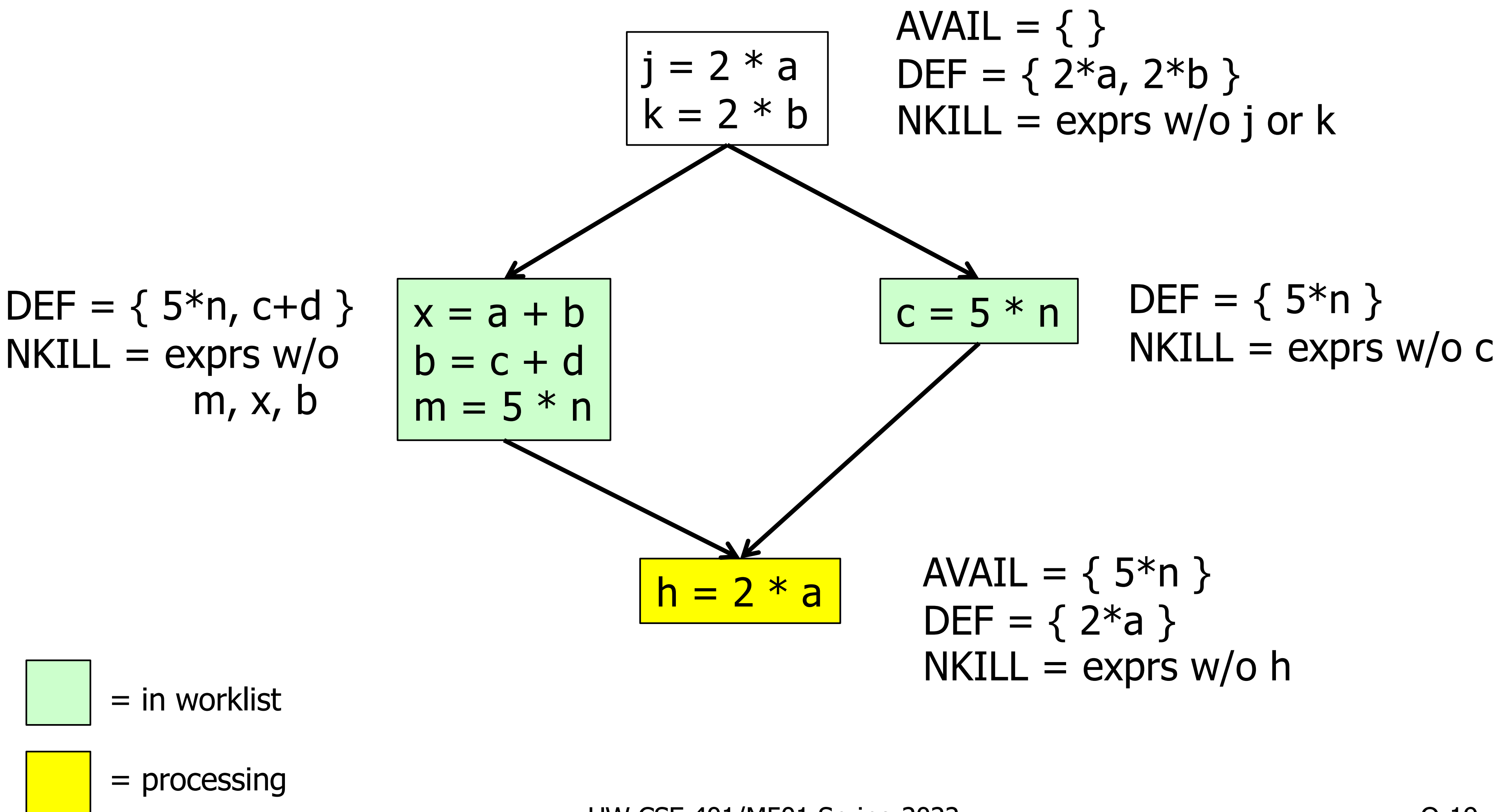
DEF = { 2*a }
NKILL = exprs w/o h

= in worklist

= processing

Example: Find Available Expressions

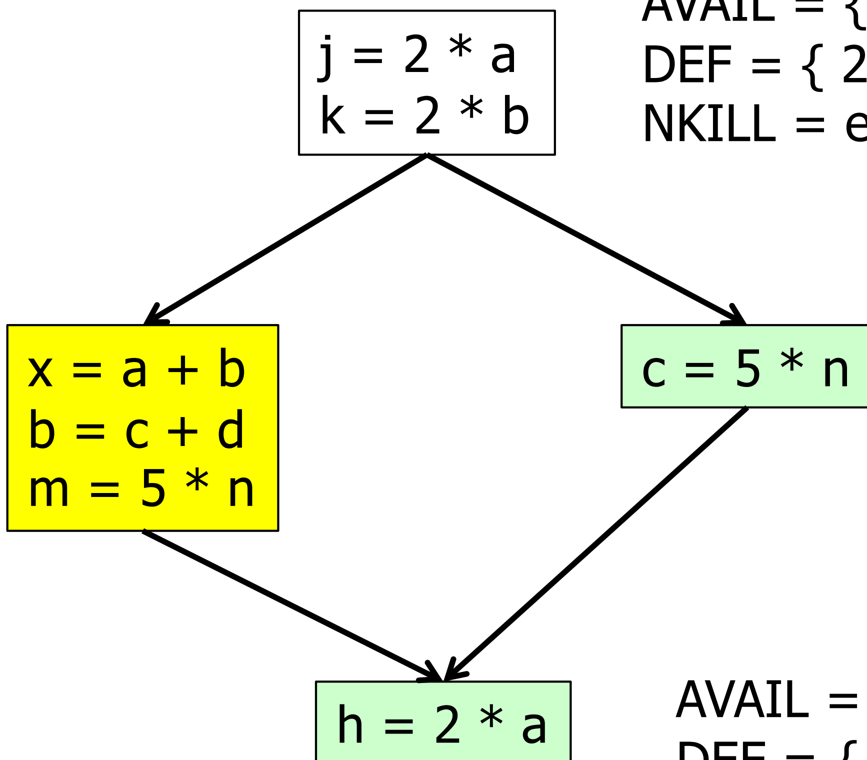
$$\text{AVAIL}(b) = \cap_{x \in \text{preds}(b)} (\text{DEF}(x) \cup (\text{AVAIL}(x) \cap \text{NKILL}(x)))$$



Example: Find Available Expressions

$$\text{AVAIL}(b) = \bigcap_{x \in \text{preds}(b)} (\text{DEF}(x) \cup (\text{AVAIL}(x) \cap \text{NKILL}(x)))$$

AVAIL = { }
DEF = { 2*a, 2*b }
NKILL = exprs w/o j or k



AVAIL = { 2*a, 2*b }
DEF = { 5*n, c+d }
NKILL = exprs w/o m, x, b

DEF = { 5*n }
NKILL = exprs w/o c

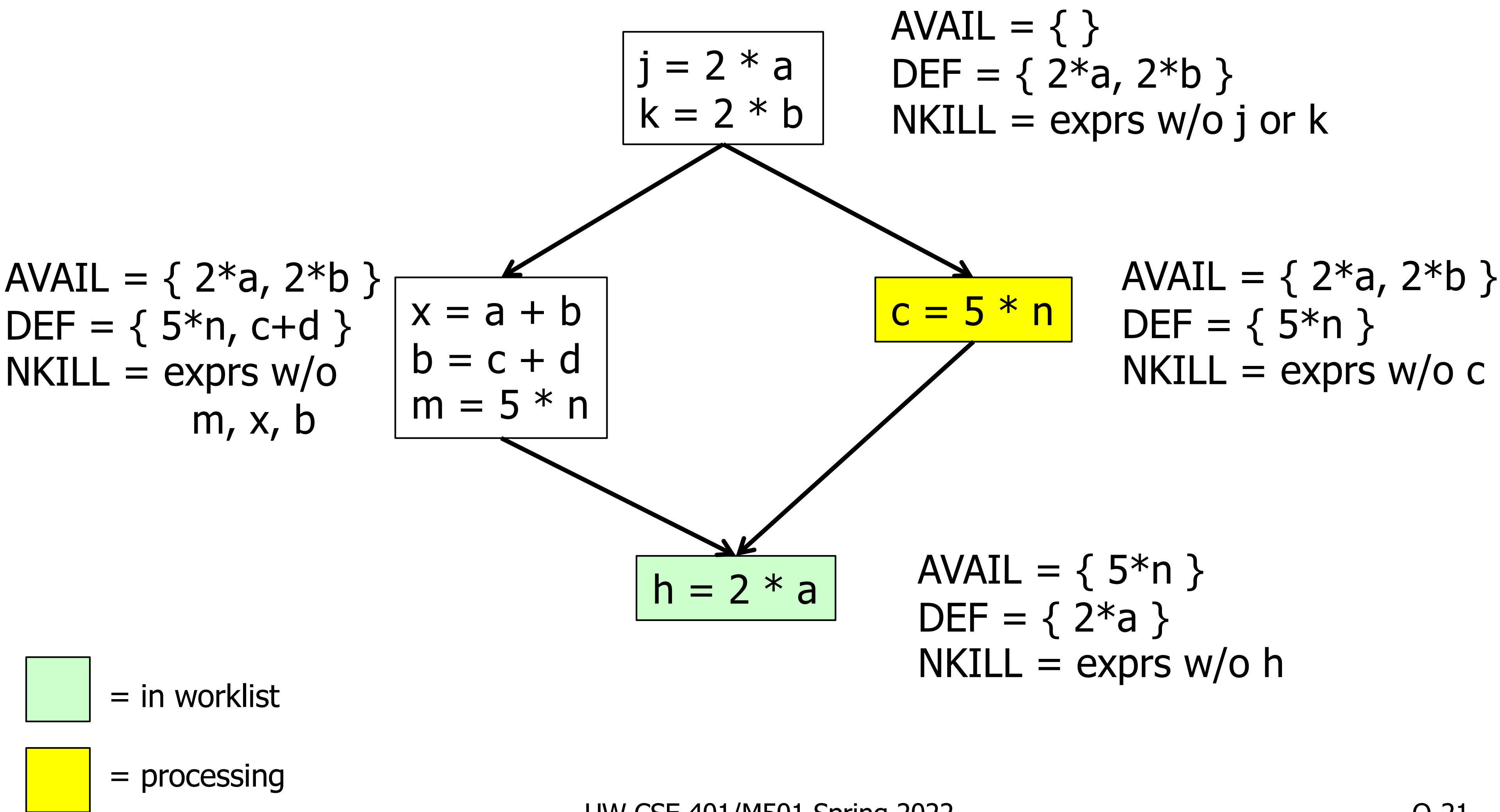
AVAIL = { 5*n }
DEF = { 2*a }
NKILL = exprs w/o h

= in worklist

= processing

Example: Find Available Expressions

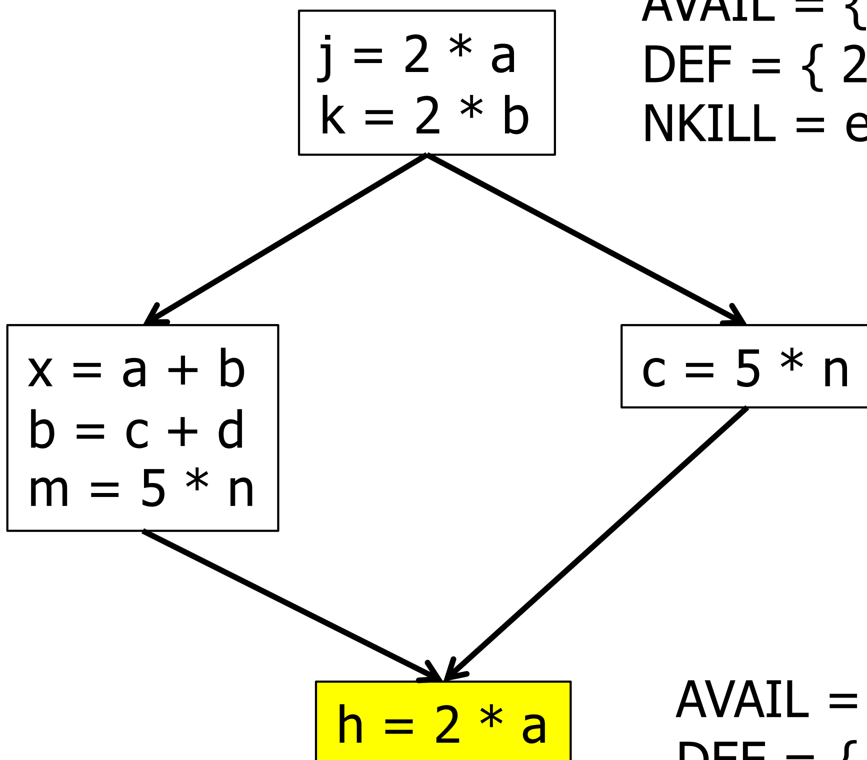
$$AVAIL(b) = \cap_{x \in preds(b)} (DEF(x) \cup (AVAIL(x) \cap NKILL(x)))$$



Example: Find Available Expressions

$\mathrm{AVAIL}(b) = \bigcap_{x \in \mathrm{preds}(b)} (\mathrm{DEF}(x) \cup (\mathrm{AVAIL}(x) \cap \mathrm{NKILL}(x)))$

AVAIL = { }
DEF = { 2*a, 2*b }
NKILL = exprs w/o j or k



AVAIL = { 2*a, 2*b }
DEF = { 5*n, c+d }
NKILL = exprs w/o m, x, b

AVAIL = { 2*a, 2*b }
DEF = { 5*n }
NKILL = exprs w/o c

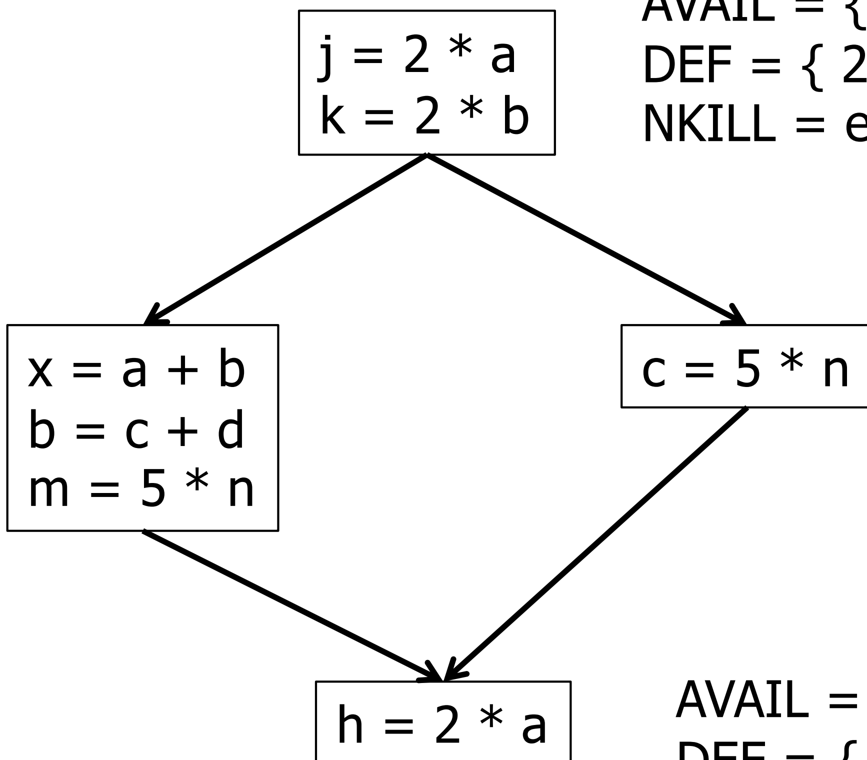
AVAIL = { 5*n, 2*a }
DEF = { 2*a }
NKILL = exprs w/o h

= in worklist

= processing

Example: Find Available Expressions

AVAIL(b) = $\cap_{x \in preds(b)}$ (DEF(x) $\cup$ (AVAIL(x) $\cap$ NKILL(x)))



AVAIL = { }
DEF = { 2*a, 2*b }
NKILL = exprs w/o j or k

AVAIL = { 2*a, 2*b }
DEF = { 5*n, c+d }
NKILL = exprs w/o m, x, b

AVAIL = { 2*a, 2*b }
DEF = { 5*n }
NKILL = exprs w/o c

AVAIL = { 5*n, 2*a }
DEF = { 2*a }
NKILL = exprs w/o h

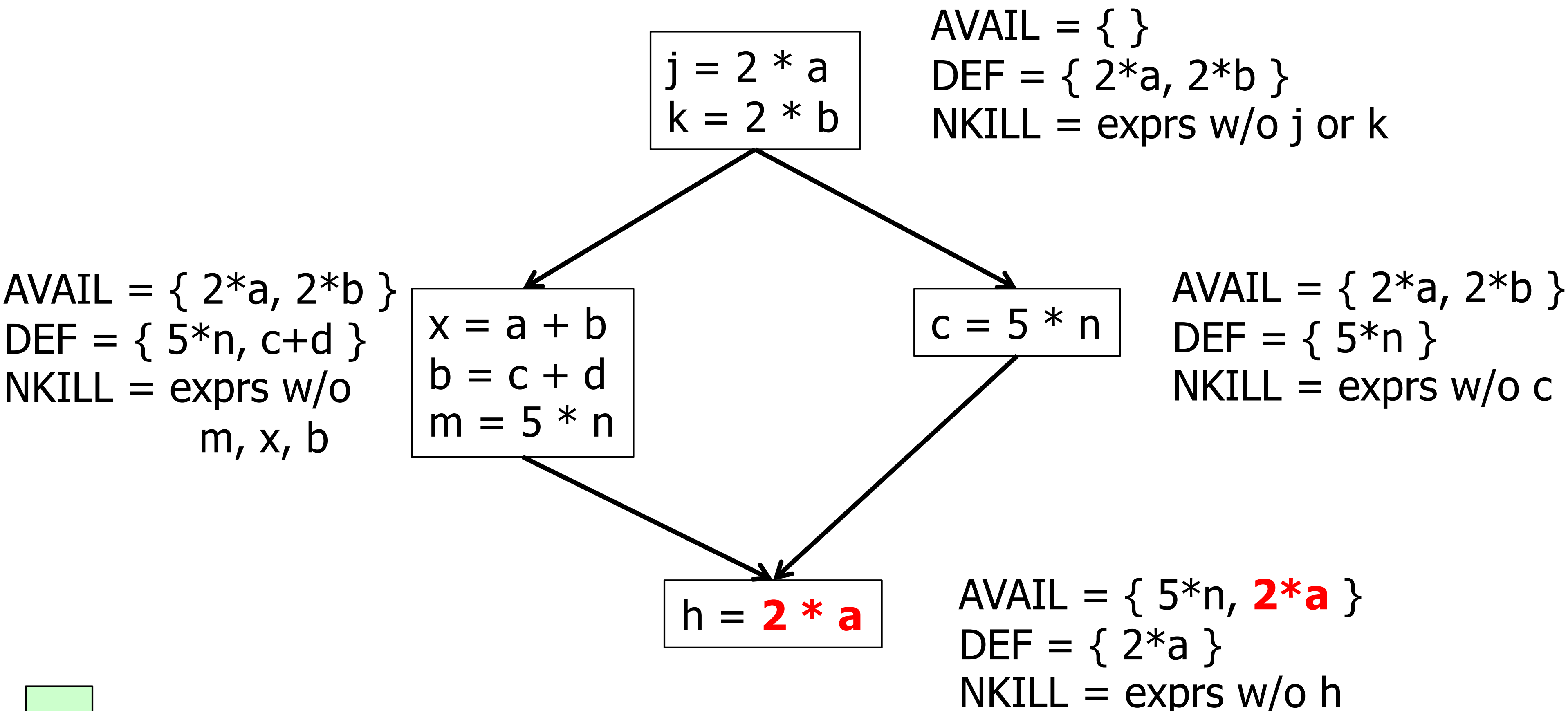
= in worklist

= processing

And the common subexpression is???

Example: Find Available Expressions

$$AVAIL(b) = \bigcap_{x \in preds(b)} (DEF(x) \cup (AVAIL(x) \cap NKILL(x)))$$



= in worklist

= processing

Example: Find Available Expressions

AVAIL(b) = $\cap_{x \in \text{preds}(b)}$ (DEF(x) $\cup$ (AVAIL(x) $\cap$ NKILL(x)))

- Termination?
  - Always
  - AVAIL(b) initially all empty
  - In equation above, DEF & NKILL are unchanging, and adding to AVAIL(x) can't shrink AVAIL(b)
  - Only a finite number of exprs in the program, so the alg is again climbing a finite n-cube; can't climb forever
- Order of worklist removals?
  - Any will work
  - Some are faster than others; e.g., if CFG is a DAG, then go in topological order (which would have been faster on the example above)

Dataflow analysis

- *Available expressions* is an example of a *dataflow analysis* problem
- Many similar problems can be expressed in a similar framework
- Only the first part of the story – once we've discovered facts, we then need to use them to improve code

Characterizing Dataflow Analysis

- All of these algorithms involve sets of facts about each basic block b

  IN(b) – facts true on entry to b

  OUT(b) – facts true on exit from b

  GEN(b) – facts created and not killed in b

  KILL(b) – facts killed in b

- These are related by the equation

  $$\text{OUT}(b) = \text{GEN}(b) \cup (\text{IN}(b) - \text{KILL}(b))$$

  – Solve this iteratively for all blocks

  – Sometimes information propagates forward; sometimes backward

Example:Live Variable Analysis

- A variable *v* is *live* at point *p* iff there is *any* path from *p* to a use of *v* along which *v* is not redefined
- Some uses:
  - Register allocation – only live variables need a register
  - Dead Store Elimination – if variable not live at store, then stored variable will never be used
  - Detecting uses of uninitialized variables – if live at declaration (before initialization) then it *might* be used uninitialized
  - Improve SSA construction – only need Φ-function for variables that are live in a block (later)

Liveness Analysis Sets

- For each block *b*, define
  - use[*b*] = variable used in *b* before any def
  - def[*b*] = variable defined in *b* and not killed
  - in[*b*] = variables live on entry to *b*
  - out[*b*] = variables live on exit from *b*

Equations for Live Variables

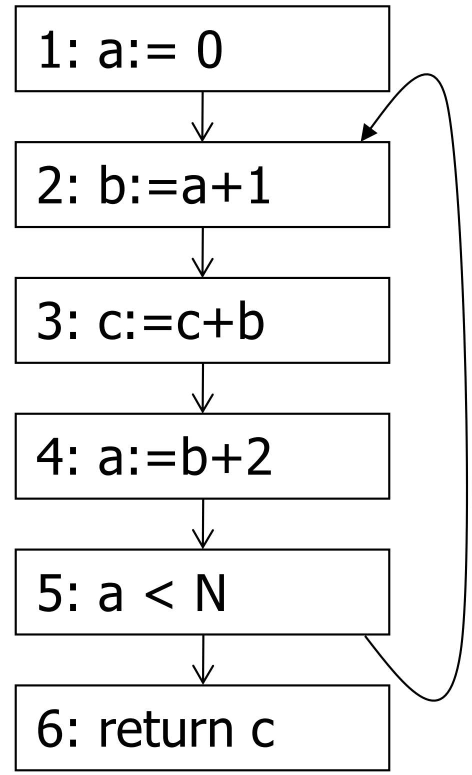
- Given the preceding definitions, we have
  - $\text{out}[b] = \cup_{s \in \text{succ}[b]} \text{in}[s]$
  - $\text{in}[b] \quad = \text{use}[b] \cup (\text{out}[b] - \text{def}[b])$
- Algorithm
  - For all b set $\text{in}[b] = \text{out}[b] = \emptyset$
  - Update out, in until no change

Example (1 stmt per block)

- Code

```
        a := 0
L:      b := a+1
        c := c+b
        a := b*2
        if a < N goto L
        return c
```

N assumed const

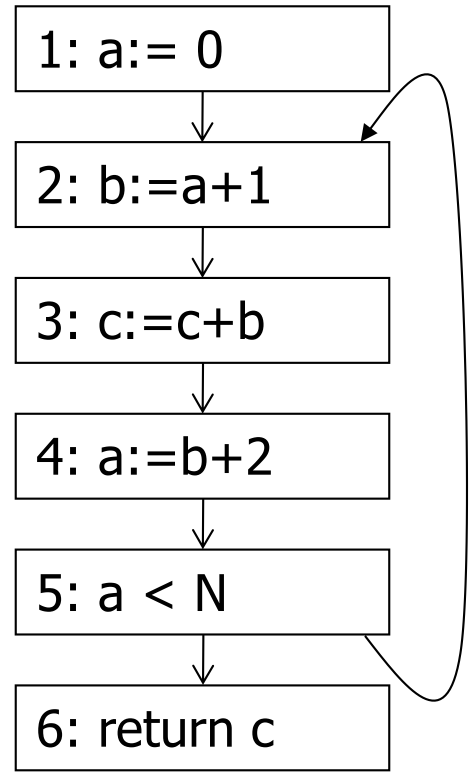


$$\text{out}[b] = \cup_{s \in \text{succ}[b]} \text{in}[s]$$

$$\text{in}[b] = \text{use}[b] \cup (\text{out}[b] - \text{def}[b])$$

Calculation

			I		II		III	
block	use	def	out	in	out	in	out	in
6								
5								
4								
3								
2								
1								



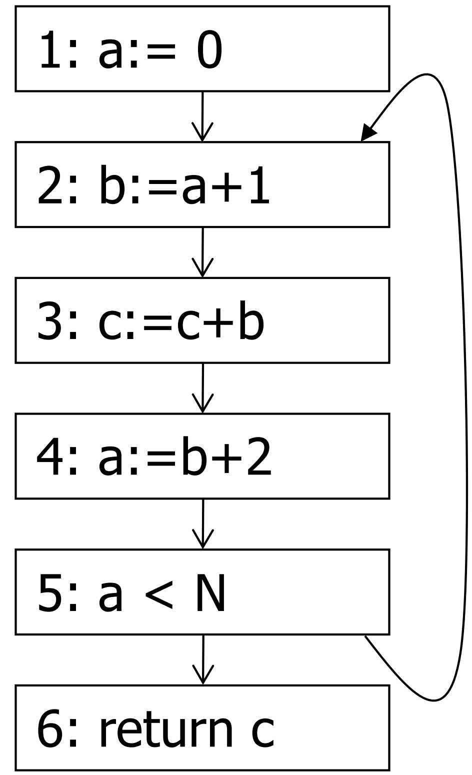
use[*b*] = variable used in *b* before any def
def[*b*] = variable defined in *b* and not killed
in[*b*] = variables live on entry to *b*
out[*b*] = variables live on exit from *b*

$\text{out}[b] = \cup_{s \in \text{succ}[b]} \text{in}[s]$

$\text{in}[b] = \text{use}[b] \cup (\text{out}[b] - \text{def}[b])$

Calculation

			I		II		III	
block	use	def	out	in	out	in	out	in
6	c	–						
5	a	–						
4	b	a						
3	b,c	c						
2	a	b						
1	–	a						



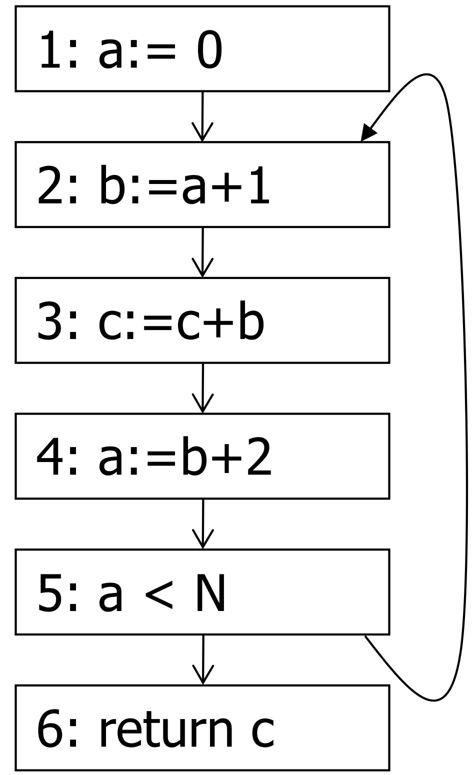
use[*b*] = variable used in *b* before any def
def[*b*] = variable defined in *b* and not killed
in[*b*] = variables live on entry to *b*
out[*b*] = variables live on exit from *b*

$$\text{out}[b] = \cup_{s \in \text{succ}[b]} \text{in}[s]$$
$$\text{in}[b] = \text{use}[b] \cup (\text{out}[b] - \text{def}[b])$$

Calculation

Last change

			I		II		III	
block	use	def	out	in	out	in	out	in
6	c	–	–	c	–	c		
5	a	–	c	a,c	a,c	a,c		
4	b	a	a,c	b,c	a,c	b,c	SAME	
3	b,c	c	b,c	b,c	b,c	b,c		
2	a	b	b,c	a,c	b,c	a,c		
1	–	a	a,c	c	a,c	c		



use[*b*] = variable used in *b* before any def
def[*b*] = variable defined in *b* and not killed
in[*b*] = variables live on entry to *b*
out[*b*] = variables live on exit from *b*

$$\text{out}[b] = \cup_{s \in \text{succ}[b]} \text{in}[s]$$
$$\text{in}[b] = \text{use}[b] \cup (\text{out}[b] - \text{def}[b])$$

Equations for Live Variables v2

- Many problems have more than one formulation. For example, Live Variables…
- Sets
  - USED(b) – variables used in b before being defined in b
  - NOTDEF(b) – variables not defined in b
  - LIVE(b) – variables live on *exit* from b
- Equation

$$\mathrm{LIVE}(b) = \cup_{s \in \mathrm{succ}(b)} \mathrm{USED}(s) \cup (\mathrm{LIVE}(s) \cap \mathrm{NOTDEF}(s))$$

Efficiency of Dataflow Analysis

- The algorithms eventually terminate, but the expected time needed can be reduced by picking a good order to visit nodes in the CFG
  - Forward problems – reverse postorder
  - Backward problems – postorder

Example: Reaching Definitions

- A definition *d* of some variable *v* *reaches* operation *i* iff *i* reads the value of *v* and there is a path from *d* to *i* that does not define *v*
- Uses
  - Find all of the possible definition points for a variable in an expression

Equations for Reaching Definitions

- Sets
  - DEFOUT(b) – set of definitions in b that reach the end of b (i.e., not subsequently redefined in b)
  - SURVIVED(b) – set of all definitions not obscured by a definition in b
  - REACHES(b) – set of definitions that reach b
- Equation

$$\text{REACHES}(b) = \bigcup_{p \in \text{preds}(b)} (\text{DEFOUT}(p) \cup (\text{REACHES}(p) \cap \text{SURVIVED}(p)))$$

Example: Very Busy Expressions

- An expression *e* is considered *very busy* at some point *p* if *e* is evaluated and used along every path that leaves *p*, and evaluating *e* at *p* would produce the same result as evaluating it at the original locations
- Uses
  - Code hoisting – move *e* to *p* (at a minimum, it reduces code size; and faster if some *e*'s are more deeply nested in loops than *p,* tho other transforms often preclude this case.)

Equations for Very Busy Expressions

- Sets
  - USED(b) – expressions used in b before they are killed
  - KILLED(b) – expressions redefined in b before they are used
  - VERYBUSY(b) – expressions very busy on exit from b
- Equation

$$\text{VERYBUSY}(b) = \bigcap_{s \in \text{succ}(b)} (\text{USED}(s) \cup (\text{VERYBUSY}(s) - \text{KILLED}(s)))$$

Using Dataflow Information

- A few examples of possible transformations…

Classic Common-Subexpression Elimination (CSE)

- In a statement s: *z* := *x* op *y*, if *x* op *y* is *available* at s then it need not be recomputed
- Where was it computed?
- Analysis: compute *reaching expressions* i.e., statements n: *v* := *x* op *y* such that the path from n to s does not compute *x* op *y* or define *x* or *y.* (How? Like reaching definitions, but for expressions.)

Classic CSE Transformation

- If x op y is defined at n and reaches s
  - Create new temporary t_i
  - Rewrite n: v := x op y as

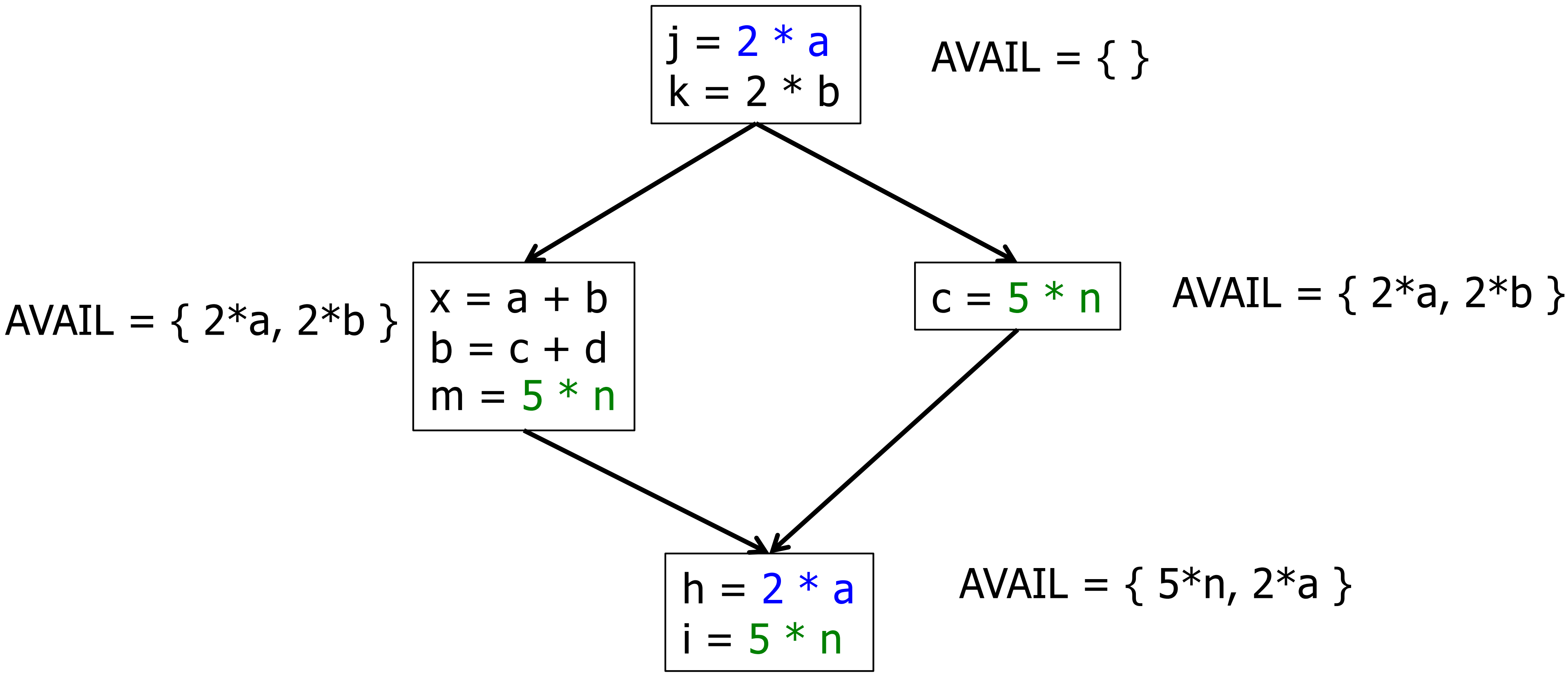
    n: t_i := x op y *// t_i is a new temporary*

    n': v := t_i
  - Rewrite statement s: z := x op y to be

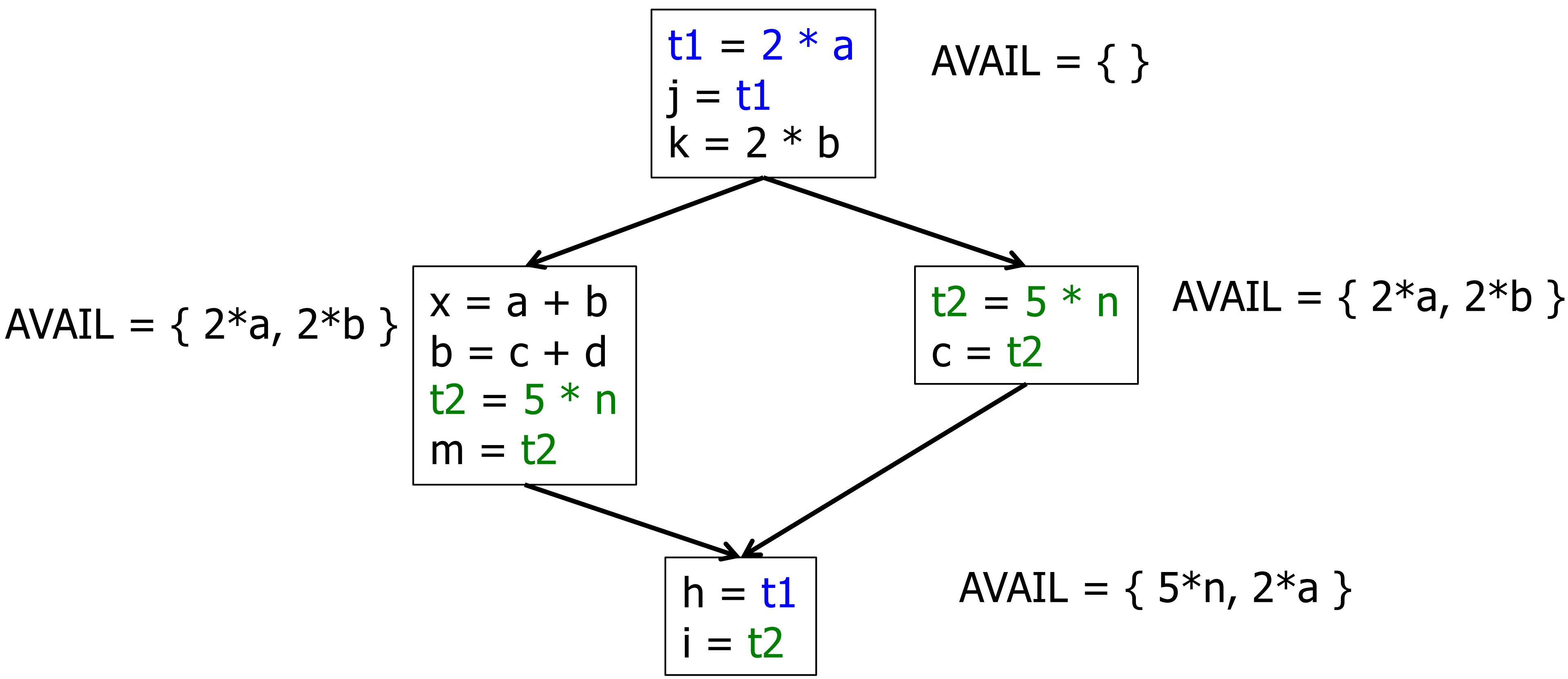
    s: z := t_i

  - (Rely on copy propagation to remove extra assignments if not really needed)

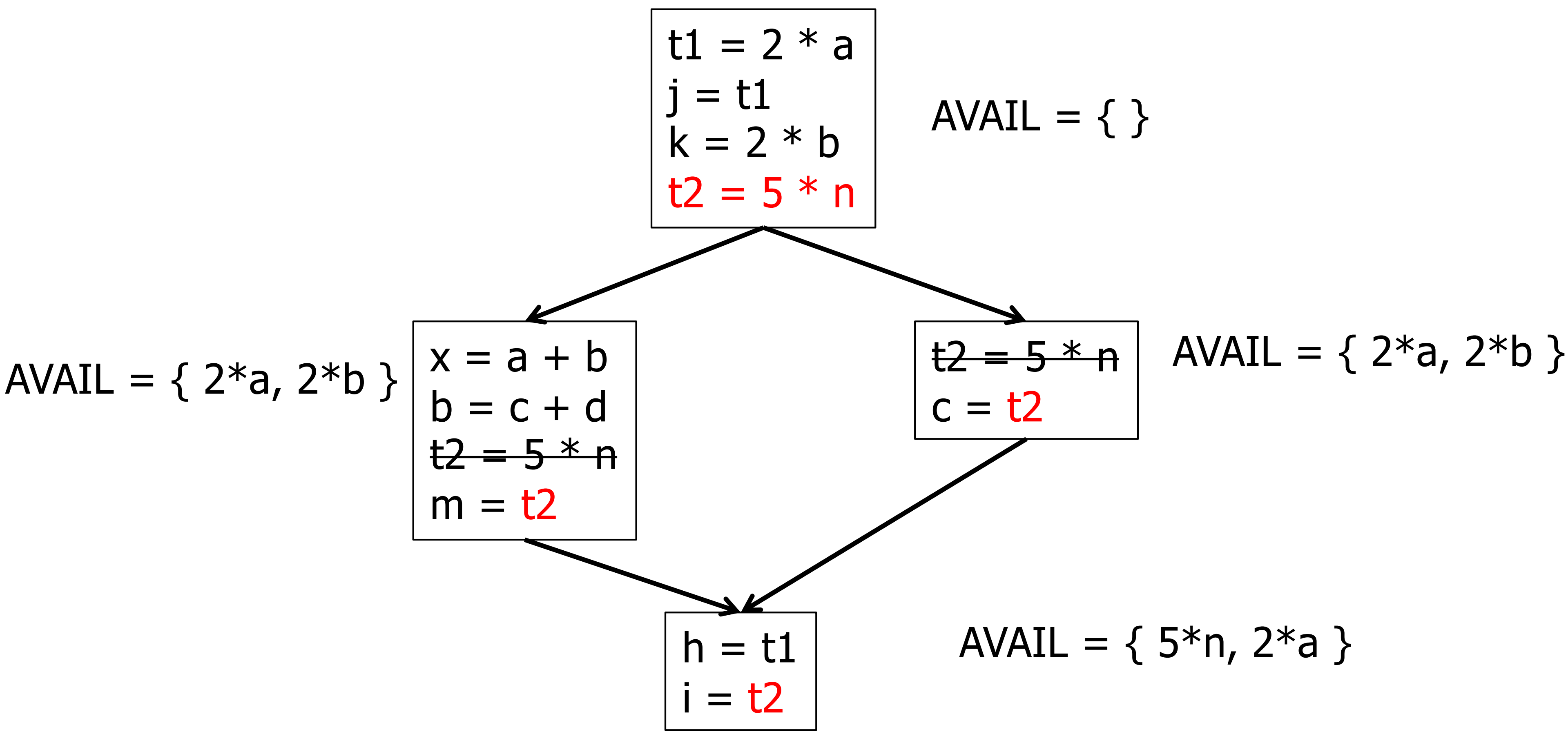
Revisiting Example (w/small change)



Revisiting Example (w/small change)



Then Apply Very Busy…



Constant Propagation

- Suppose we have
  - Statement d: $t := c$, where c is constant
  - Statement n that uses t
- If d reaches n and no other definitions of t reach n, then rewrite n to use c instead of t

Copy Propagation

- Similar to constant propagation
- Setup:
  - Statement d: $t := z$
  - Statement n uses t
- If d reaches n and no other definition of t reaches n, and there is no definition of z on any path from d to n, then rewrite n to use z instead of t
  - Recall that this can help remove dead assignments

Copy Propagation Tradeoffs

- Downside is that this can increase the lifetime of variable *z* and increase need for registers or memory traffic
- But it can expose other optimizations, e.g.,

  a := y + z

  u := y

  c := u + z   // copy propagation makes this y + z

  – After copy propagation we can recognize the common subexpression

Dead Code (Assignment) Elimination

- If we have an instruction

  s: a := b op c

  and a is not live-out after s, then s can be eliminated
  - Provided it has no implicit side effects that are visible (output, exceptions, etc.)
    - If b or c are function calls, they have to be assumed to have unknown side effects unless the compiler can prove otherwise

Dataflow…

- General framework for discovering facts about programs
  - Although not the only possible story
- And then: facts open opportunities for code improvement
- Next time: SSA (static single assignment) form – transform program to a new form where each variable has only *one* single definition
  - Can make many optimizations/analysis more efficient