

CSE 401/M501 – Compilers

Dataflow Analysis

Hal Perkins

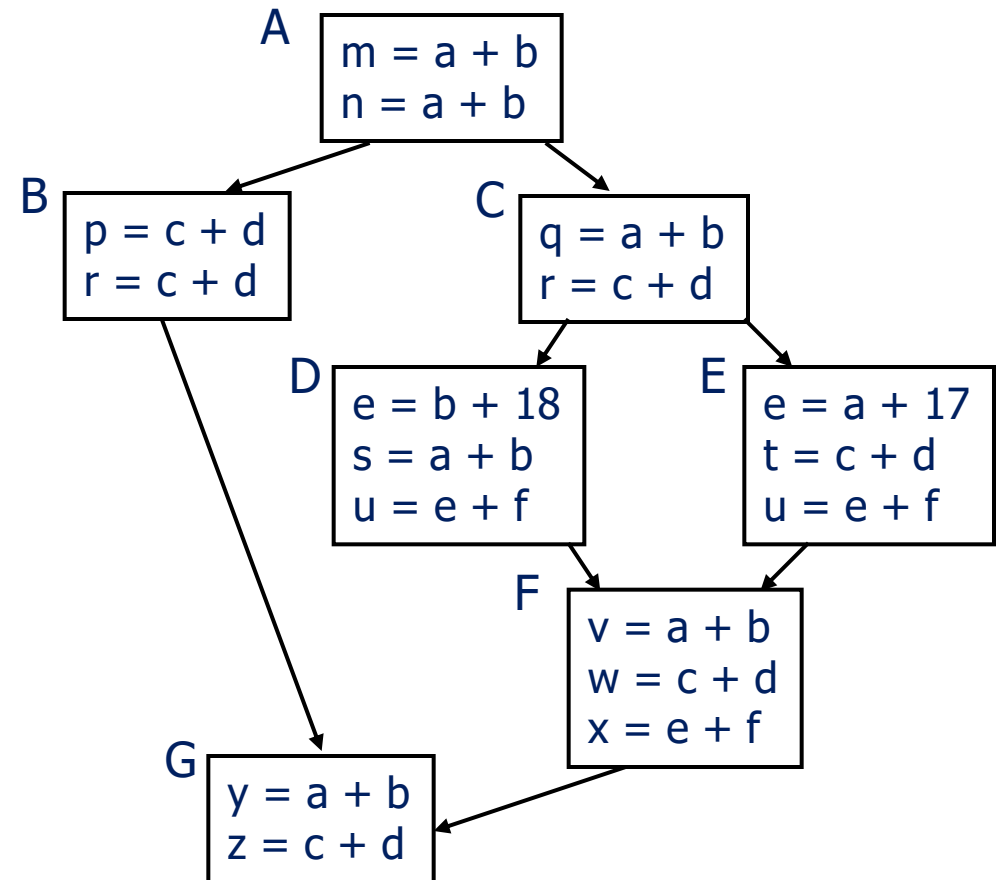
Autumn 2019

Agenda

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

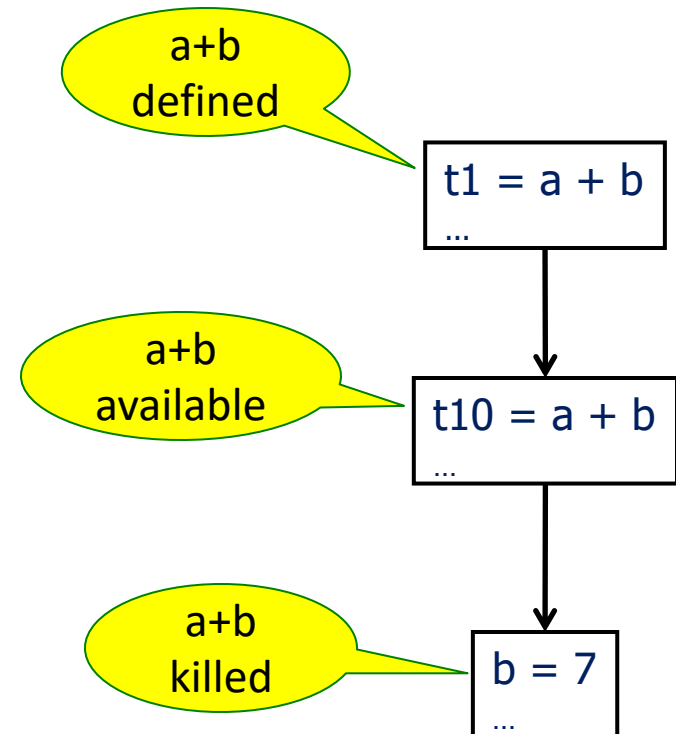
Common Subexpression Elimination

- 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 blocks



“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 not killed in b
 - i.e., all expressions in the program *except* for those killed in b
 - $DEF(b)$ – the set of expressions defined in b and not subsequently killed in b

Computing Available Expressions

- $AVAIL(b)$ is the set

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

- $\text{preds}(b)$ is the set of b 's predecessors in the CFG
 - The set of expressions available on entry to b is the set of expressions that were available at the end of *every* predecessor basic block x
 - The expressions available on exit from block b are those defined in b or available on entry to b and not killed in b
- 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 and NKILL (1)

- For each block b with operations o_1, o_2, \dots, o_n
KILLED = \emptyset // variables killed in b , not expressions
DEF(b) = \emptyset
for $k = n$ to 1 // note: working back to front
 assume o_k is “ $x = y + z$ ”
 add x to KILLED
 if ($y \notin$ KILLED and $z \notin$ KILLED)
 add “ $y + z$ ” to DEF(b) // i.e., neither y nor z killed
 // after this point in b
 ...

Computing DEF and NKILL (2)

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

$NKILL(b) = \{ \text{all expressions} \}$

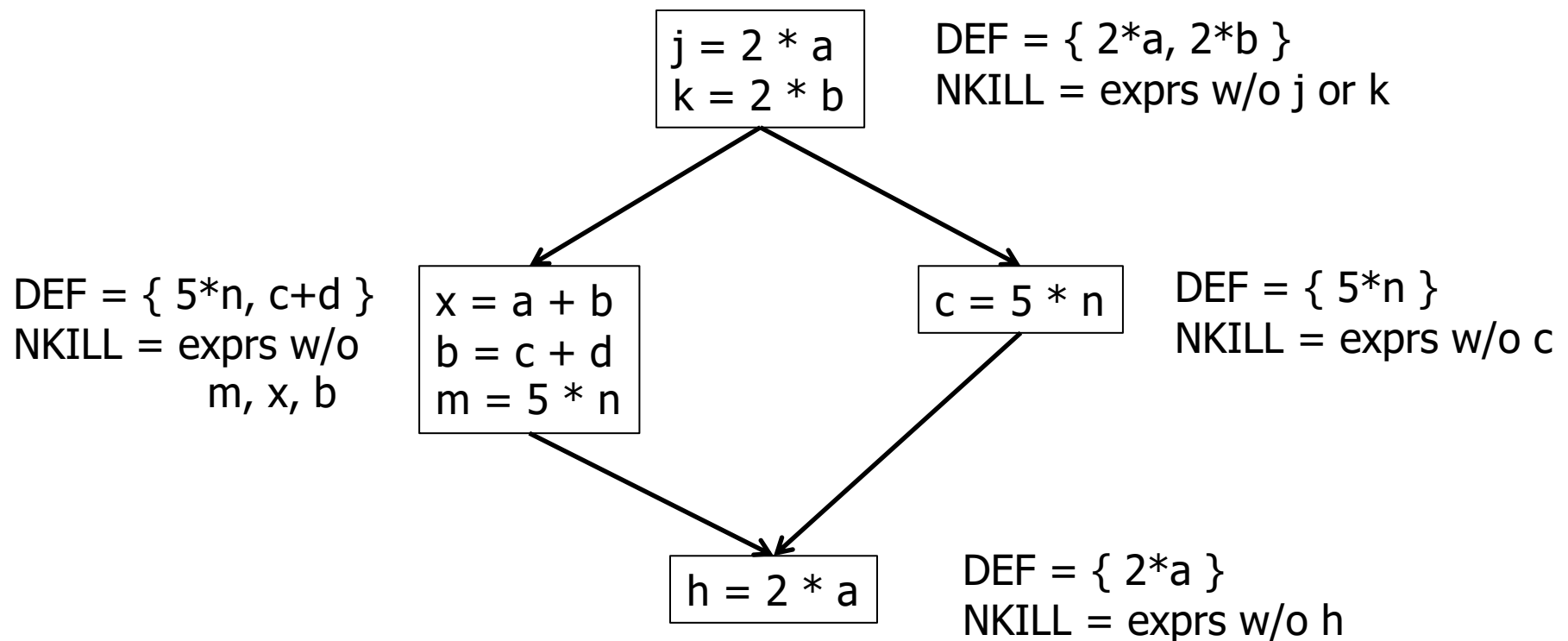
for each expression e

for each variable $v \in e$

if $v \in KILLED$ then

$NKILL(b) = NKILL(b) - e$

Example: Compute DEF and NKILL



Computing Available Expressions

Once $\text{DEF}(b)$ and $\text{NKILL}(b)$ are computed for all blocks b

Worklist = { all blocks b_k }

while (Worklist $\neq \emptyset$)

 remove a block b from Worklist

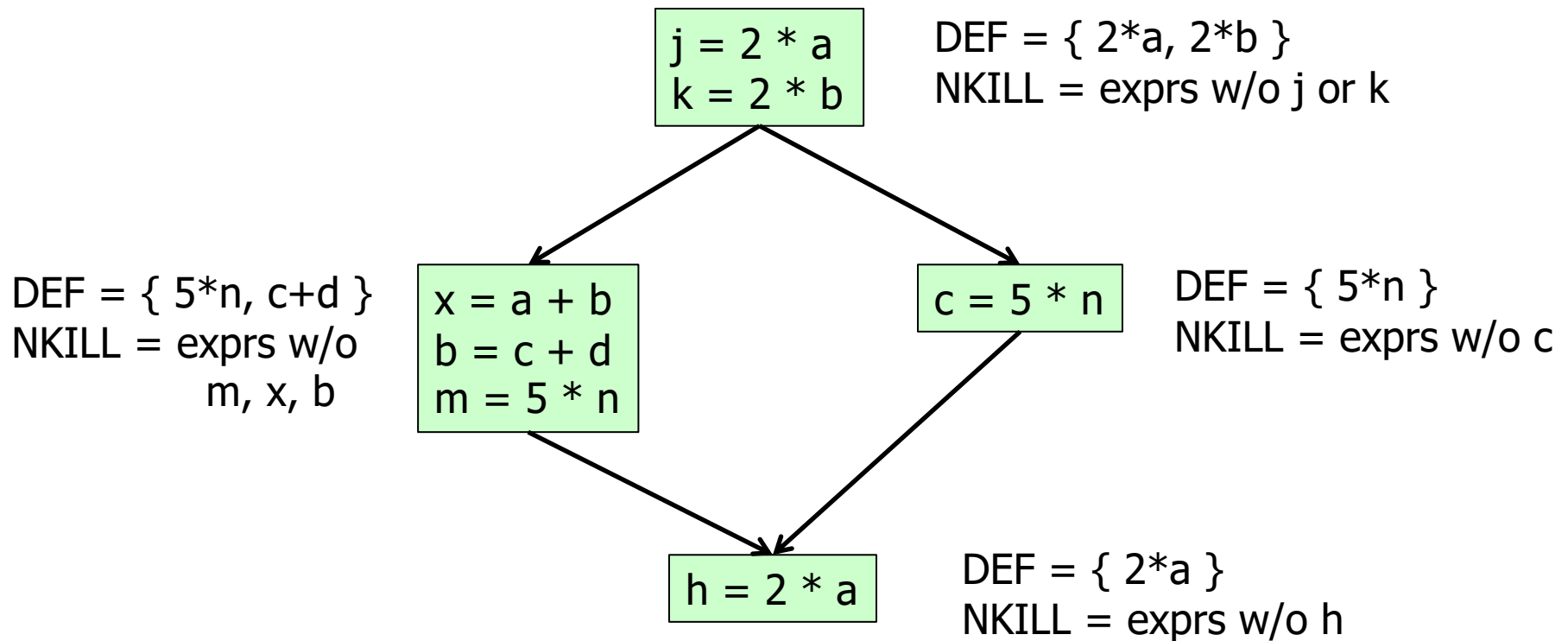
 recompute $\text{AVAIL}(b)$


 if $\text{AVAIL}(b)$ changed


 Worklist = Worklist \cup successors(b)

Example: Find Available Expressions

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

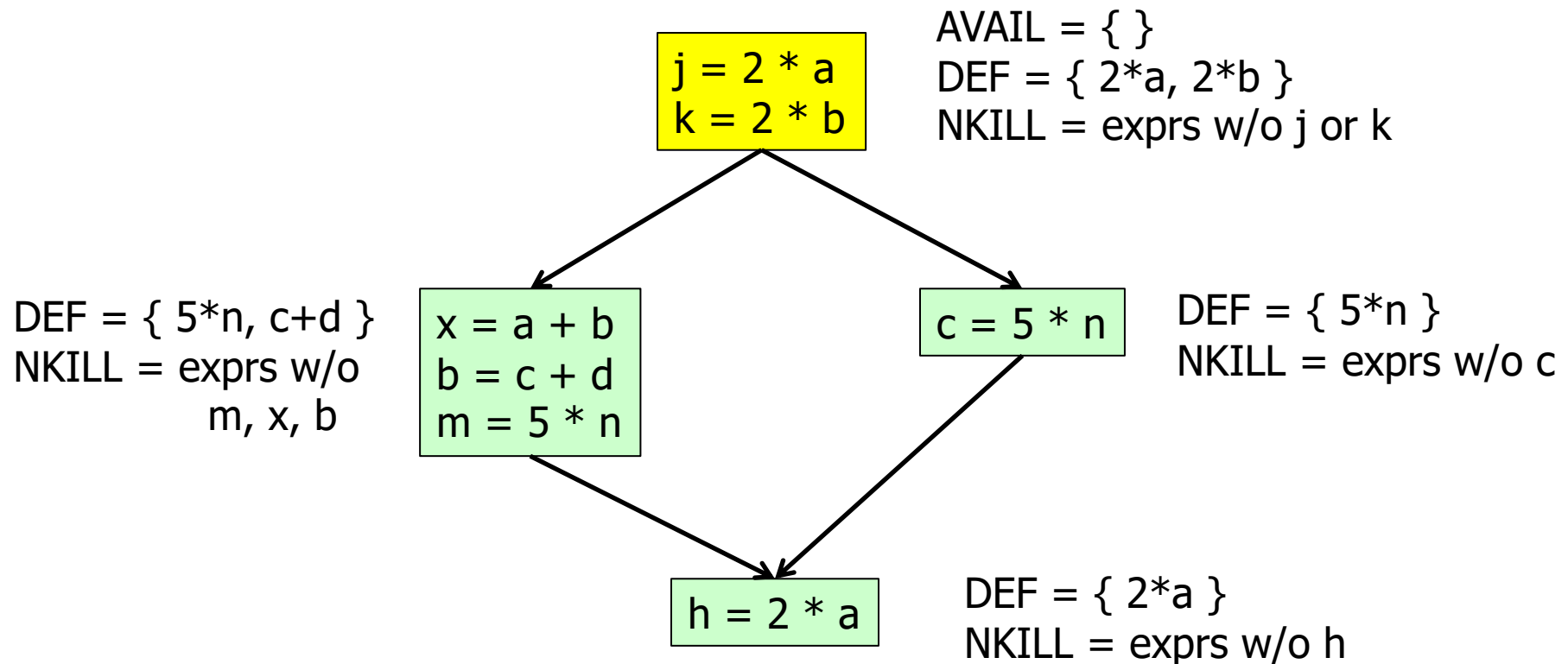


 = in worklist

 = processing

Example: Find Available Expressions

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

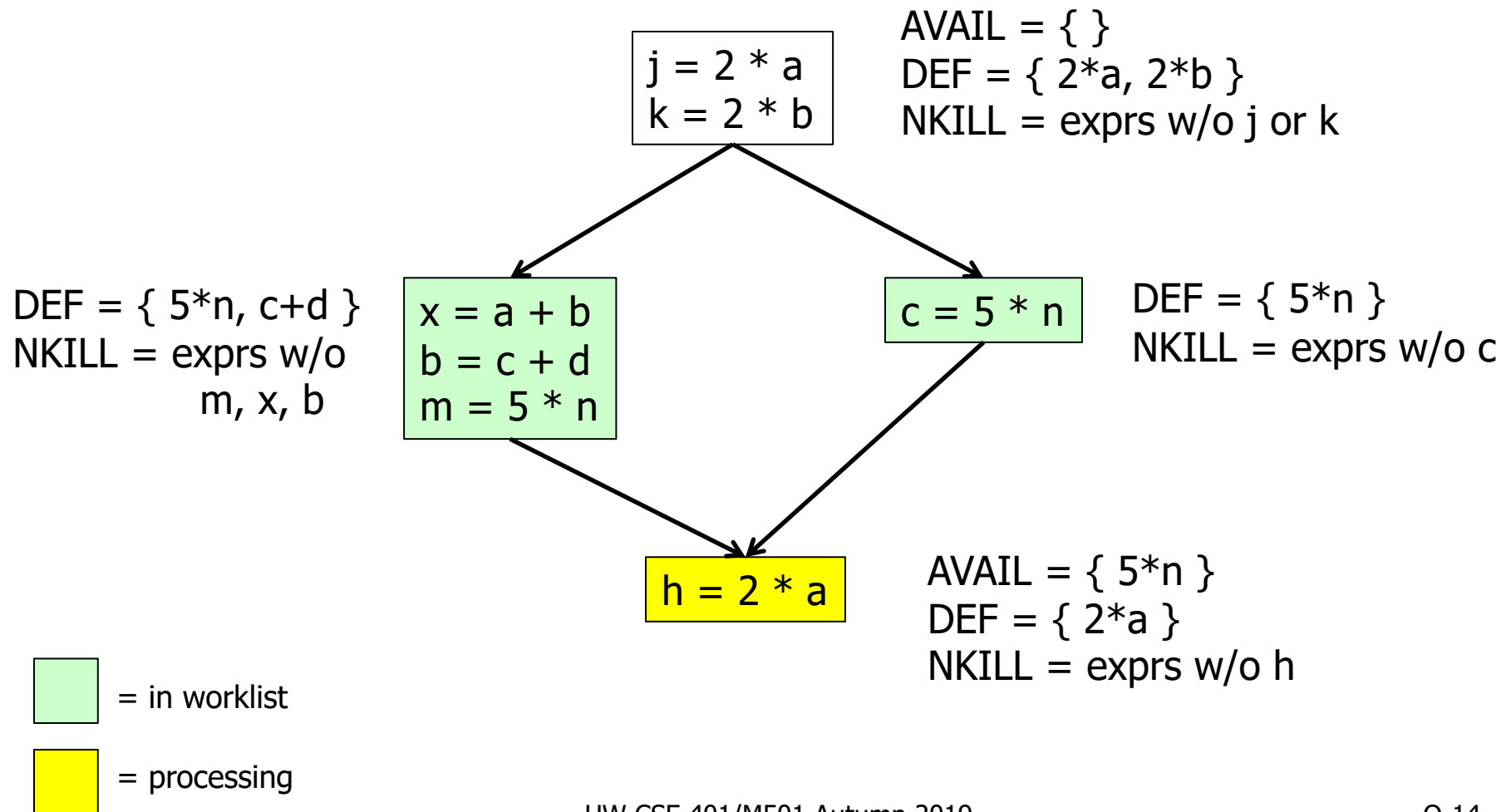


= in worklist

= processing

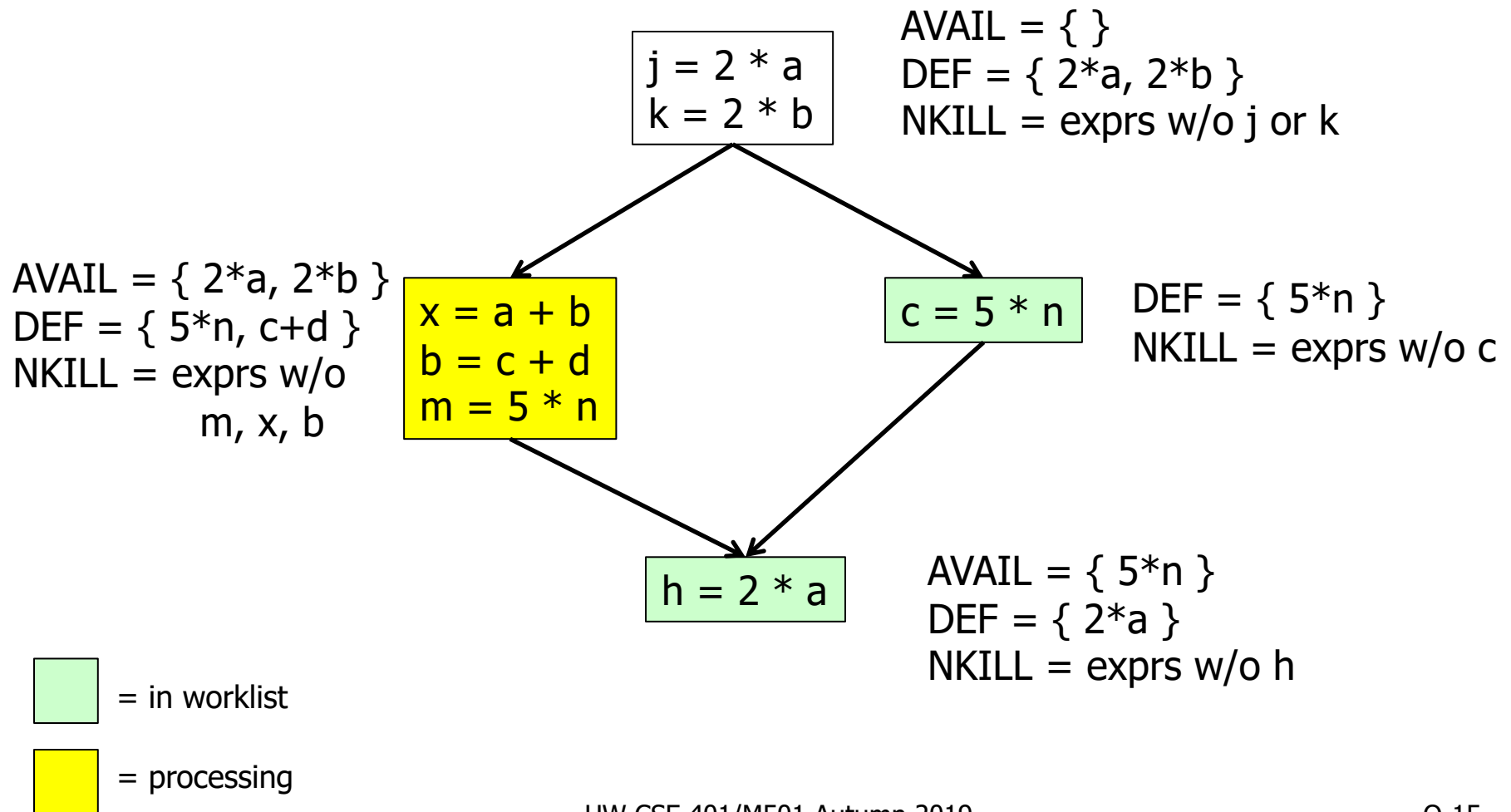
Example: Find Available Expressions

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



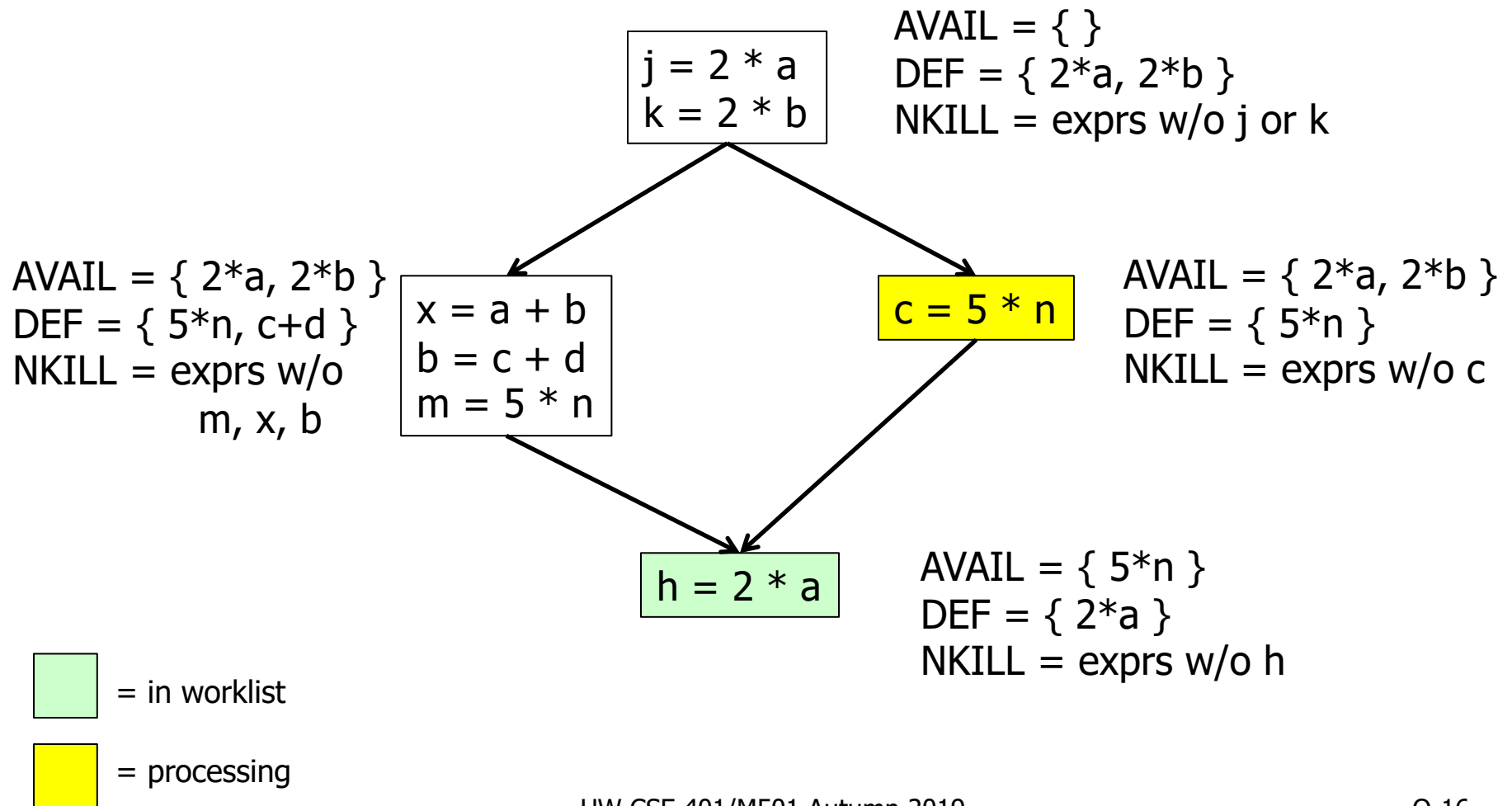
Example: Find Available Expressions

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



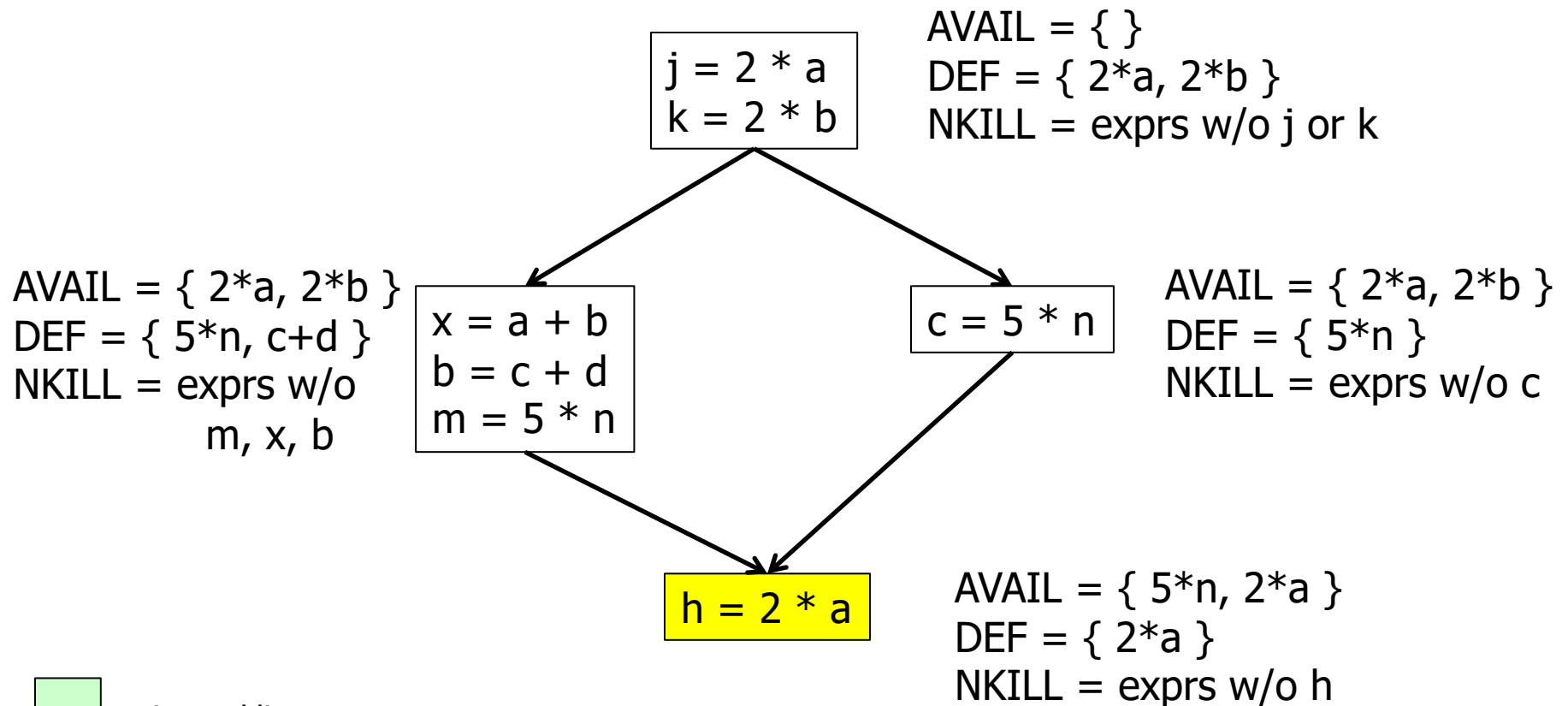
Example: Find Available Expressions


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




Example: Find Available Expressions

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

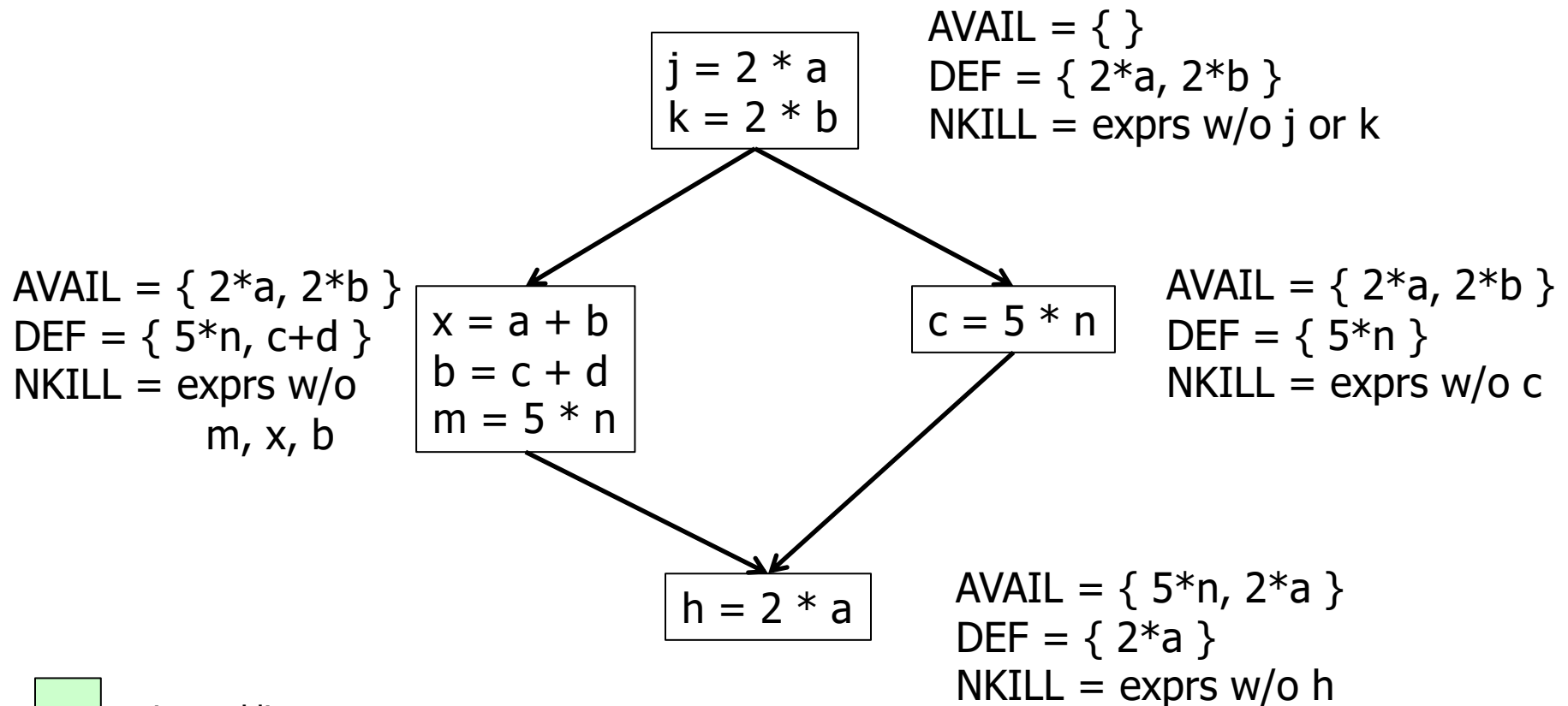



 = in worklist


 = processing

Example: Find Available Expressions

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



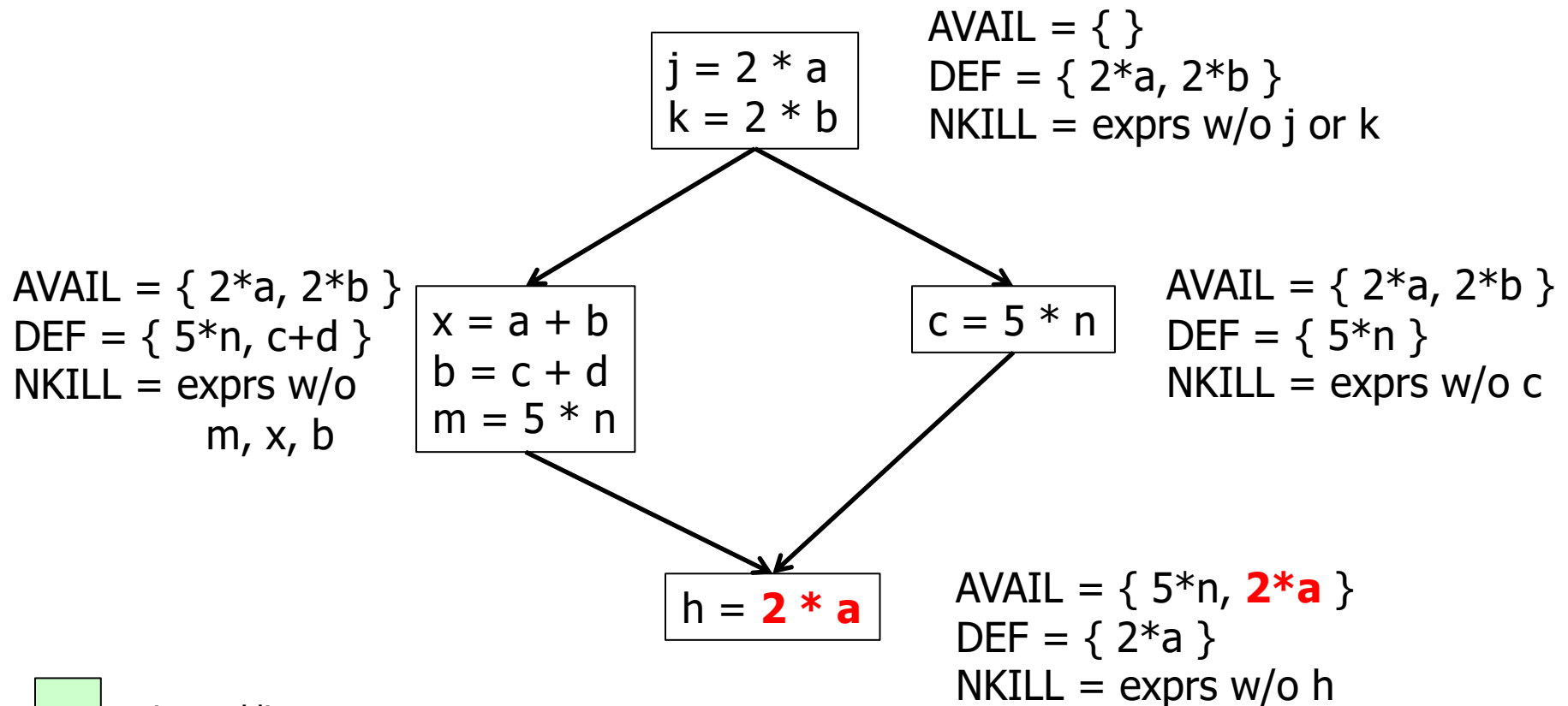
 = in worklist

 = processing

And the common subexpression is???

Example: Find Available Expressions

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



= in worklist

= processing

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
$$OUT(b) = GEN(b) \cup (IN(b) - KILL(b))$$
 - 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
 - Eliminating useless stores – 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
 - $\text{use}[b]$ = variable used in b before any def
 - $\text{def}[b]$ = variable defined in b and not killed
 - $\text{in}[b]$ = variables live on entry to b
 - $\text{out}[b]$ = variables live on exit from b

Equations for Live Variables

- Given the preceding definitions, we have

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

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

- Algorithm
 - Set $\text{in}[b] = \text{out}[b] = \emptyset$
 - Update in, out until no change

Example (1 stmt per block)

- Code

`a := 0`

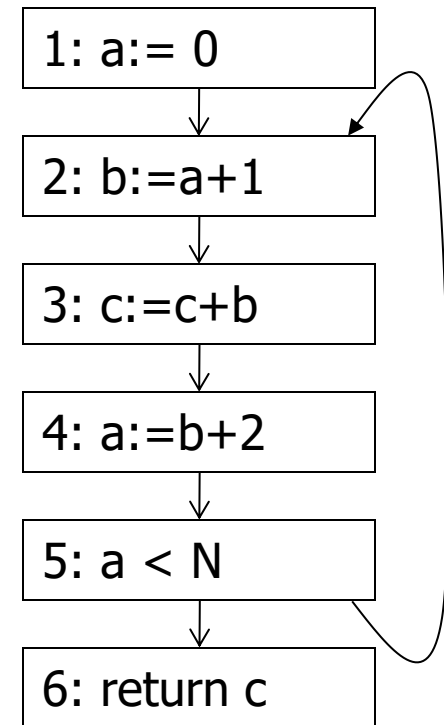
`L: b := a+1`

`c := c+b`

`a := b*2`

`if a < N goto L`

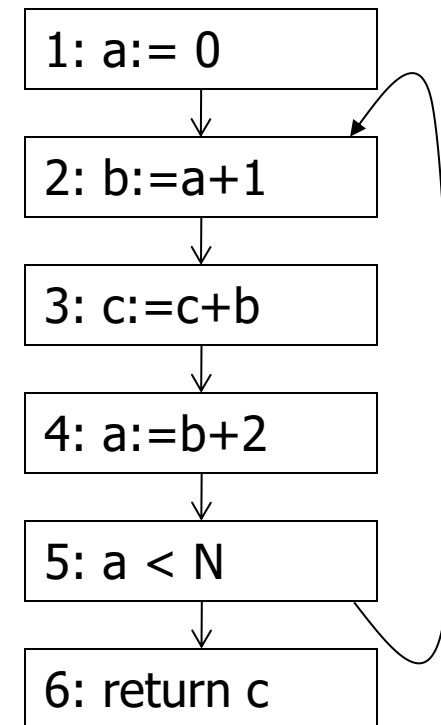
`return c`



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

Calculation

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

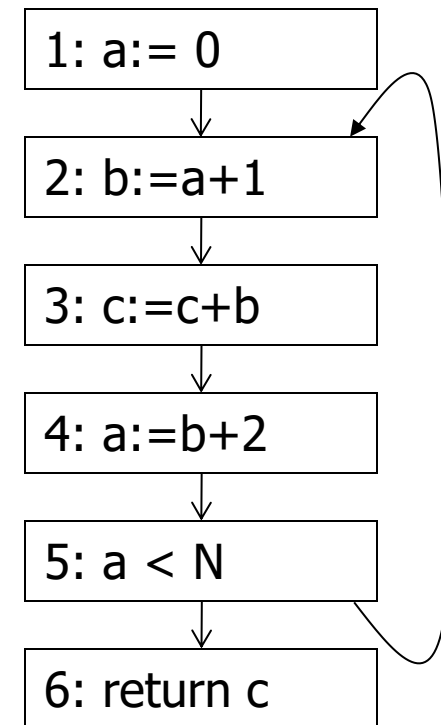


$$\text{in}[b] = \text{use}[b] \cup (\text{out}[b] - \text{def}[b])$$

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

Calculation

	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		
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		



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

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

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

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
 - $\text{DEFOUT}(b)$ – set of definitions in b that reach the end of b (i.e., not subsequently redefined in b)
 - $\text{SURVIVED}(b)$ – set of all definitions not obscured by a definition in b
 - $\text{REACHES}(b)$ – 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 (reduces code size; no effect on execution time)

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

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

Using Dataflow Information

- A few examples of possible transformations...

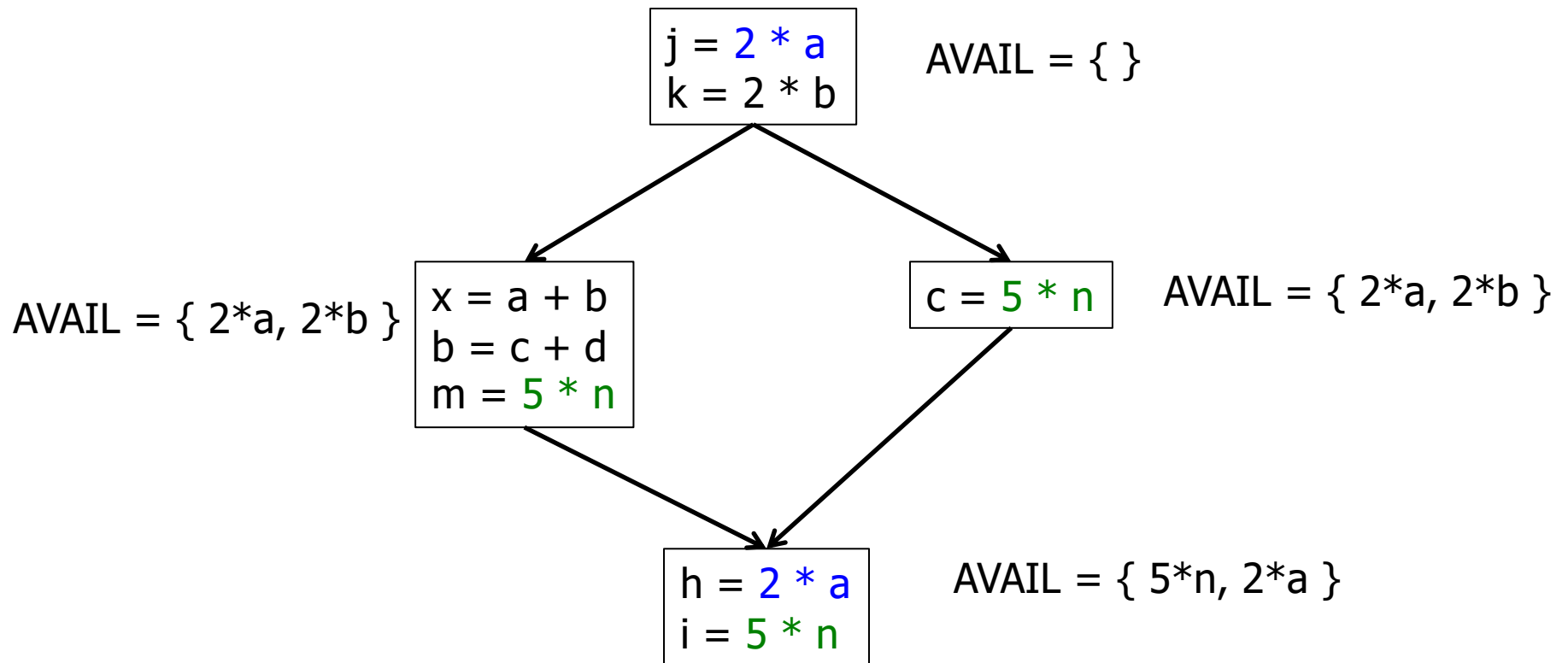
Classic Common-Subexpression Elimination (CSE)

- In a statement $s: t := x \text{ op } y$, if $x \text{ op } y$ is *available* at s then it need not be recomputed
- Analysis: compute *reaching expressions* i.e., statements $n: v := x \text{ op } y$ such that the path from n to s does not compute $x \text{ op } y$ or define x or y

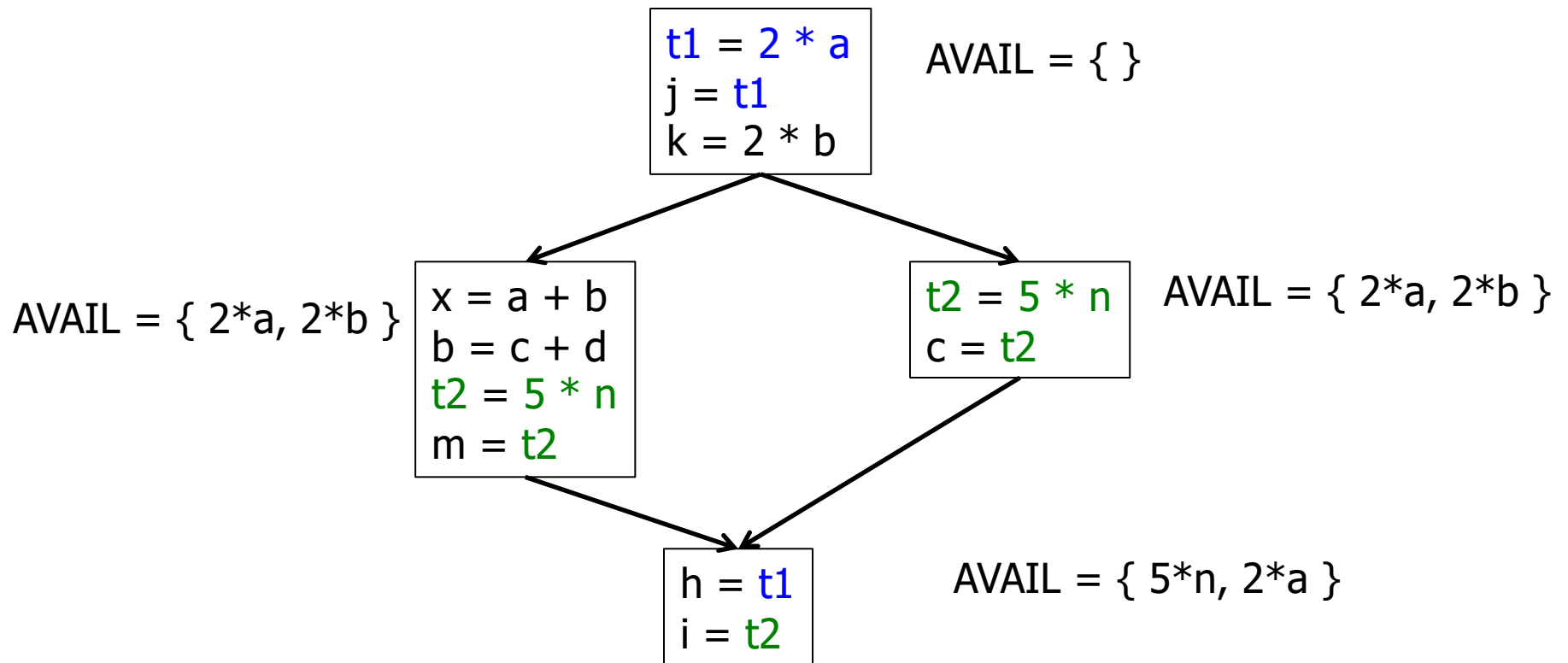
Classic CSE Transformation

- If $x \text{ op } y$ is defined at n and reaches s
 - Create new temporary w
 - Rewrite $n: v := x \text{ op } y$ as
$$n: w := x \text{ op } y$$
$$n': v := w$$
 - Modify statement s to be
$$s: t := w$$
 - (Rely on copy propagation to remove extra assignments if not really needed)

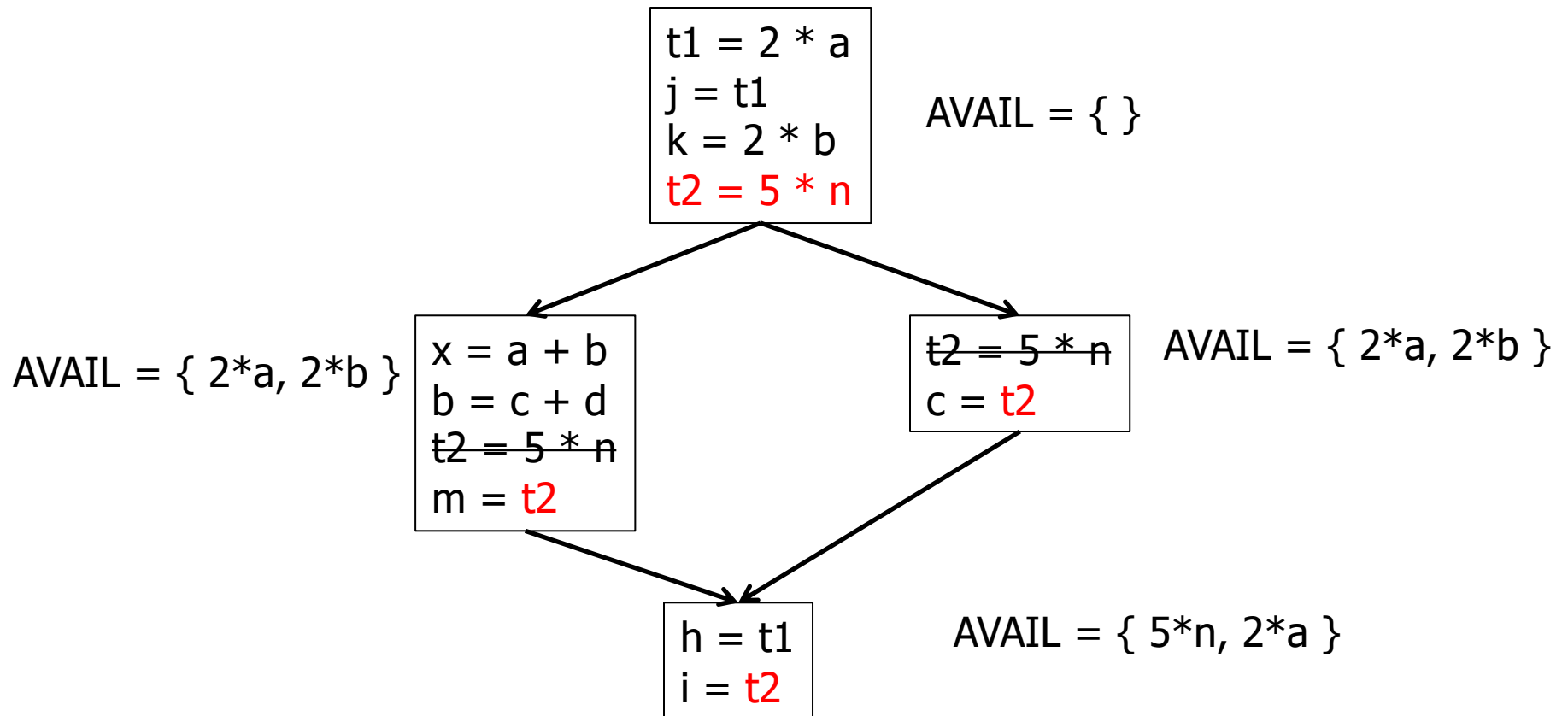
Revisiting Example (w/small change)



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 Elimination

- If we have an instruction

$s: a := b \text{ op } c$

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