Lecture O:

Dataflow Analaysis

CSE401/501m:

Introduction to Compiler Construction

Instructor: Gilbert Bernstein

Outline

Constant Analysis Revisited

Example — Available Expressions for CSE

Dataflow in General

Liveness Analysis

Other Analyses

Optimize! CSE, Copy Propagation, DCE

Outline

Constant Analysis Revisited

Example — Available Expressions for CSE

Dataflow in General

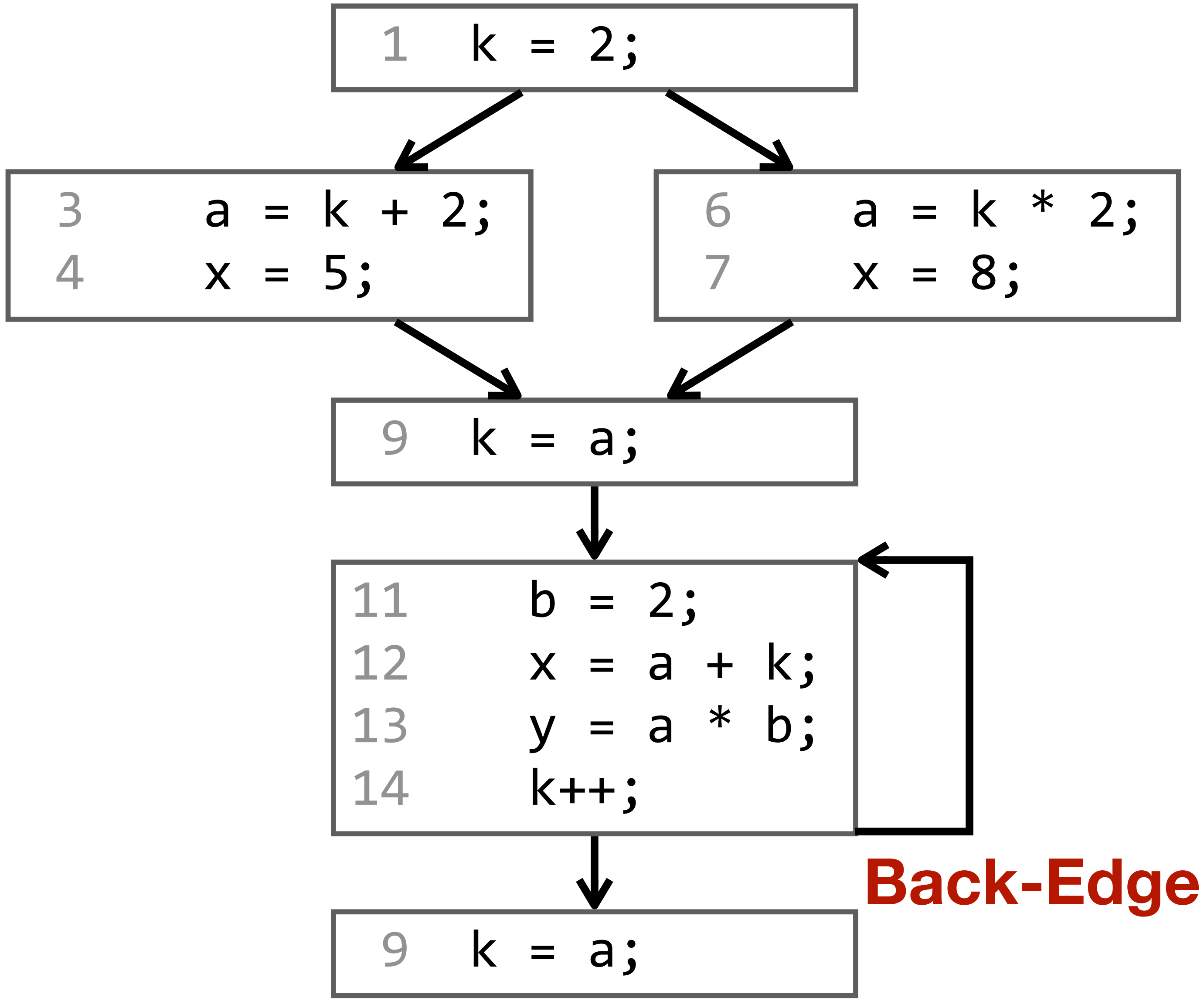
Liveness Analysis

Other Analyses

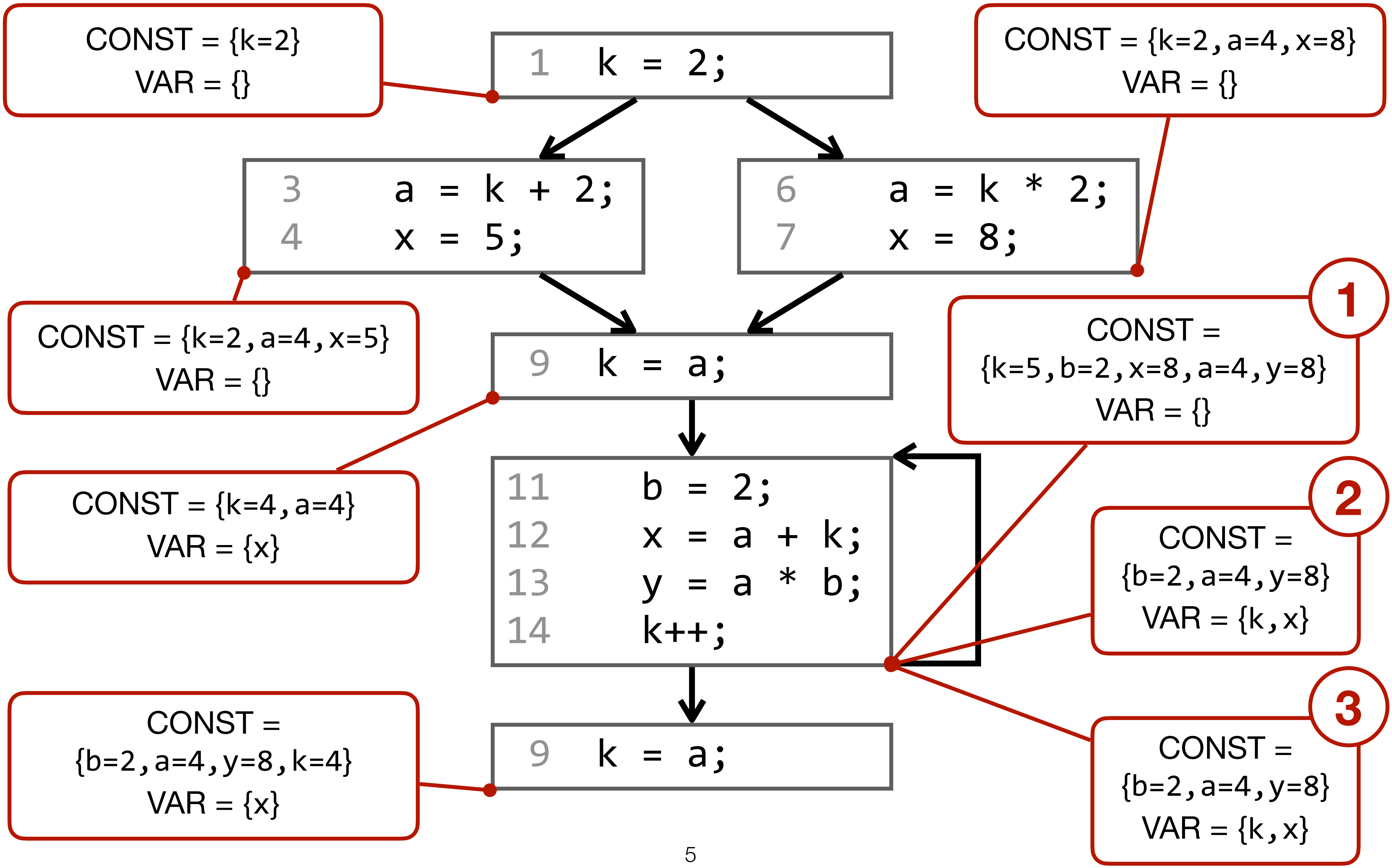
Optimize! CSE, Copy Propagation, DCE

Example — Constant Propagation

```
k = 2;
if (...) {
  a = k + 2;
  x = 5;
} else {
  a = k * 2;
  x = 8;
}
k = a;
while (...) {
  b = 2;
  x = a + k;
  y = a * b;
  k++;
}
print(a+x);
```



Example — Constant Propagation



Dataflow Basics — Facts

- Keep track of facts that are true ***after*** (or ***before***) a basic block is done executing
  - CONST — a set of all variables that are constant along with the constant value they have
  - VAR — a set of all variables that could have more than one value

Dataflow Basics — Transitions

- At each basic block, specify how to compute facts that are true ***after*** the block based on facts that are true ***before*** the block
  - Can extrapolate from a single statement
  - e.g. for `x = RHS` if *every* variable in the `RHS` is CONST, then x becomes CONST with the computed value. If any variable in the `RHS` is VAR, then x becomes VAR.
  - *(We'll show more precise equations for other analyses)*

Dataflow Basics — Merging

- Whenever we have facts ***incoming*** from more than one other basic block, we have to ***merge*** the facts
  - In our example, the constants after merging paths must be constant with the same value on all incoming paths

$$CONST_{merge} = \bigcap_{b} CONST(b)$$

  - in our example, any variable with potentially different values becomes VAR

$$VAR_{merge} = \left(\bigcup_{b} VAR(b) \cup \underbrace{\left(CONST(b) - CONST_{merge} \right)}_{} \right)$$

Note: we need to convert from a set of entries like "x=3" to a set of entries like "x" here

Dataflow Basics — Iterating

- The **master algorithm**
  - Just keep propagating facts around the CFG
  - Transform facts ***before*** basic blocks into facts ***after*** basic blocks using the transition rules
  - Merge ***incoming*** facts (i.e. facts that are true ***after*** preceding basic blocks) to update facts true ***before*** a basic block
  - Repeat in any order until facts stop changing

Outline

Constant Analysis Revisited

Example — Available Expressions for CSE

Dataflow in General

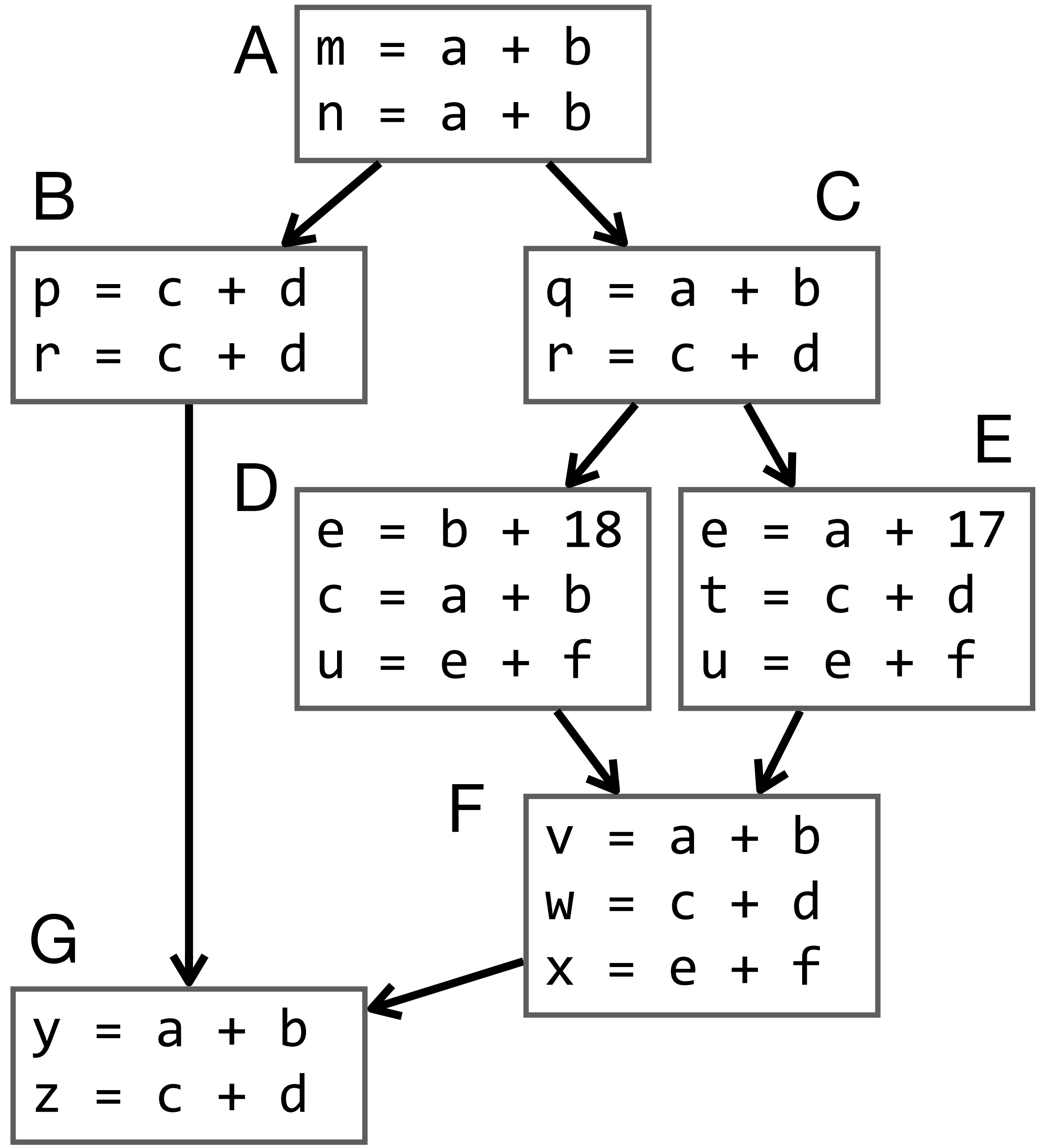
Liveness Analysis

Other Analyses

Optimize! CSE, Copy Propagation, DCE

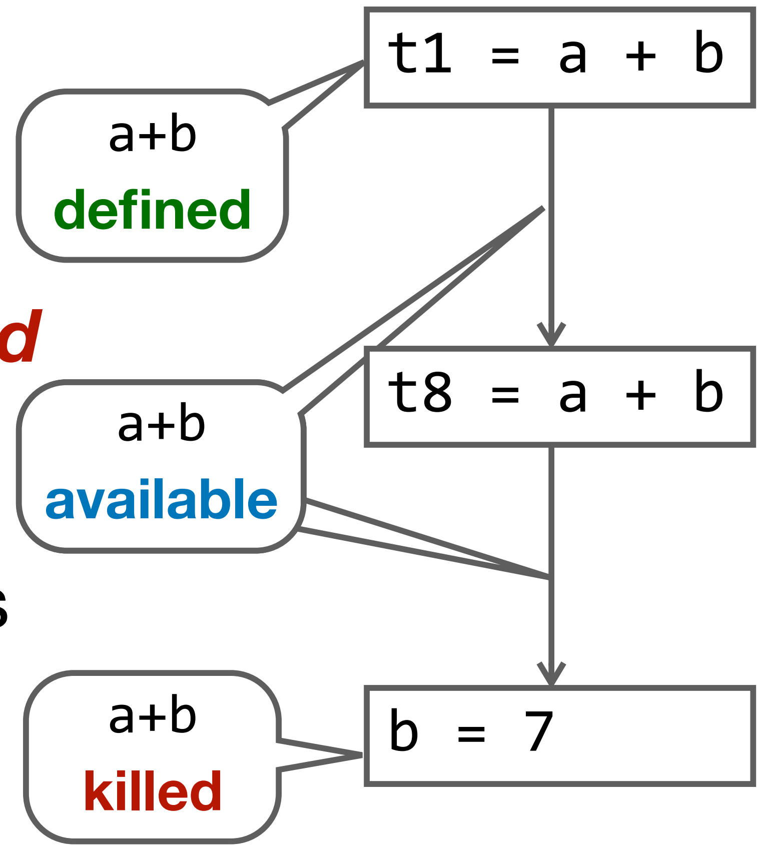
Common Subexpression Elimination

- Goal — use dataflow analysis to find common subexpressions
- Idea — calculate ***available expressions*** at the beginning of each basic block
- If an expression is ***available***, then we can replace it by copying the already computed value…



"Available" & Other Terms

- We say an expression *e* is ***defined*** at a program point *p* (aka. ***definition site***) immediately following its computation
- We say that an expression e is ***killed*** at a program point *p* (aka. ***kill site***) immediately following the (re-)definition of one of its operands
- We say that an expression e is ***available*** at a program point *p* if every path leading to p contains a prior definition of *e*, and *e* is not killed between that definition site and *p*.



Available Facts

- For each block b, we define
  - ✦ $AVAIL(b)$ — the set of expressions ***available***
    - $AVAIL_{in}(b)$ on entry to b, $AVAIL_{out}(b)$ on exit from b
  - ✦ $KILL(b)$ — the set of expressions ***killed*** in b
    - i.e. all expressions in the program that contain an operand whose value is changed/set by b
  - ✦ $DEF(b)$ — the set of expressions ***defined*** in b and not subsequently killed in b

Facts for Basic Blocks

- Consider one statement $s = \left(x \leftarrow y \ op \ z\right)$
  - ✦ $DEF(s) = \{y \ op \ z\}$
  - ✦ $KILL(s)$ — set of all expressions with x as an operand
  - ✦ Note — both of these are defined entirely locally
- $AVAIL_{out}(s) = DEF(s) \cup \left(AVAIL_{in}(s) - KILL(s)\right)$
  - ✦ The equation for the available set links before and after the statement
- For a whole basic block, just apply this idea iteratively in the block to define $DEF(b)$, $KILL(b)$, and $AVAIL_{out}(b)$

Computing KILL and DEF (1)

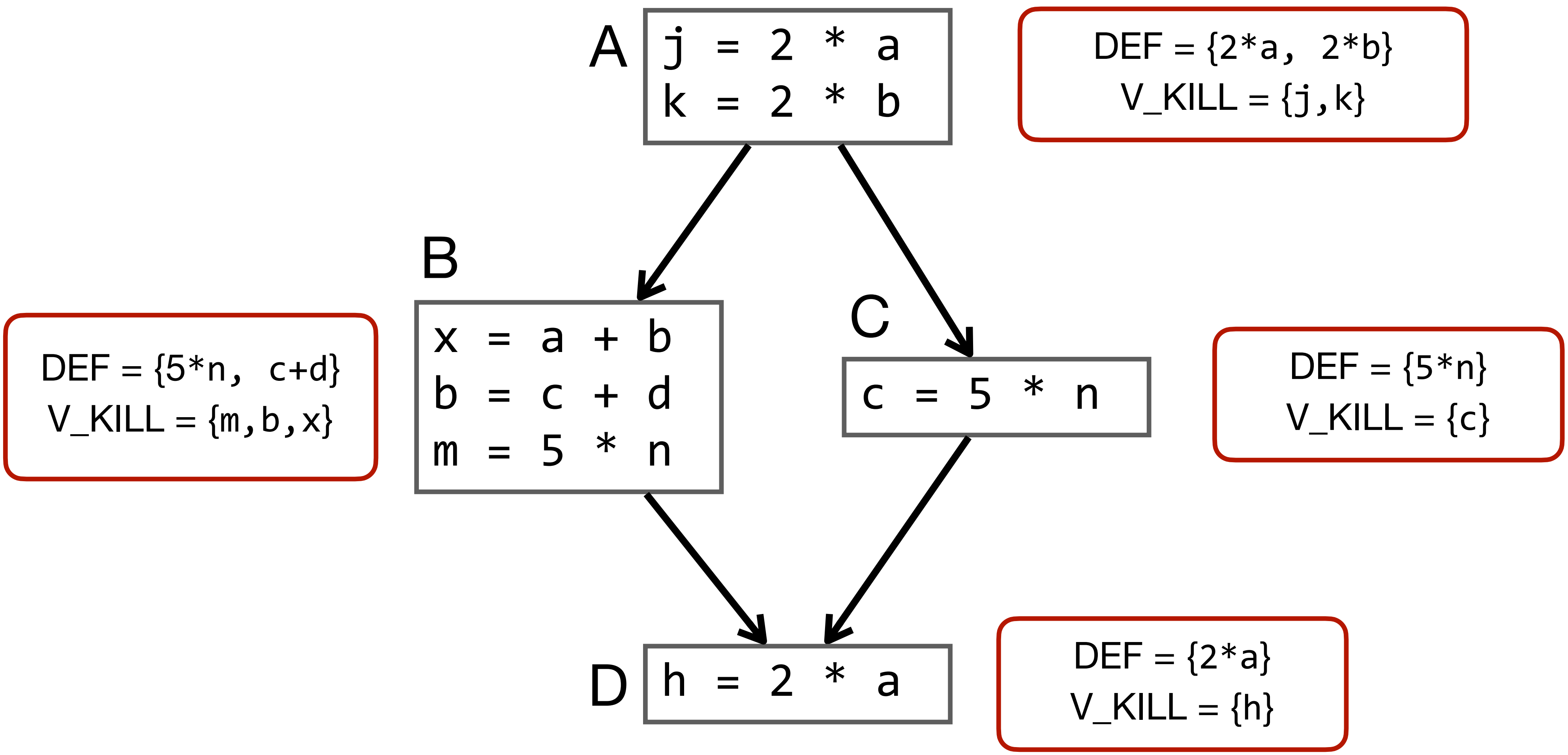
- First, figure out which variables are killed, and which expressions are defined
- For block $b = s_1; s_2; \cdots; s_n$

```
V_KILL(b) = ∅     // variables killed in b, not expr
DEF(b)    = ∅
for k = n to 1   // note — iterating backwards
    let (x ← y op z) = s
    V_KILL(b) = V_KILL(b) ∪ {x}
    if (y ∉ V_KILL(b) and z ∉ V_KILL(b))
        DEF(b) = DEF(b) ∪ {y op z}
        // i.e. neither y nor z is killed after this point in b
```

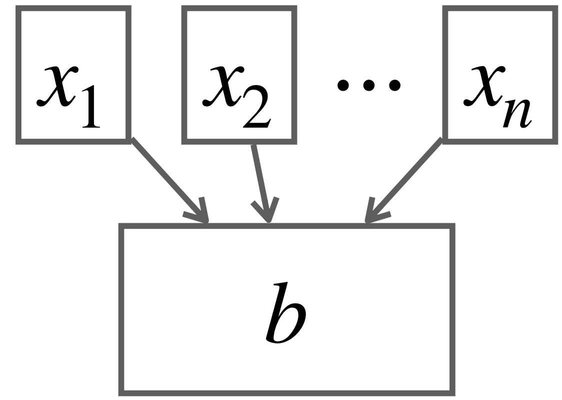
Computing KILL and DEF (2)

- How do we compute KILL? (rather than V_KILL)
  - Collect all expressions in the program (really, in the method since this is an ***intra***-procedural analysis)
  - If x is in V_KILL, then add every expression with x as an operand to KILL; repeat for all variables in V_KILL
- *Alternately*, we can just store V_KILL, and then compute $\left(AVAIL_{in}(b) - KILL(b)\right)$ by removing all expressions from $AVAIL_{in}(b)$ that have an operand in KILL(b)

Example – Compute DEF & KILL



Merging Available Facts



- Now suppose we have multiple blocks x that all have CFG edges pointing at a basic block b
- We need to give a rule for how to correctly combine the facts about what's $AVAIL_{out}(x)$ into what's $AVAIL_{in}(b)$
- The rule (see def. of ***available***) is that an expression is only available at a point p if it is available on all paths to that point. Or in terms of set equations, we have

$$AVAIL_{in}(b) = \bigcap_{x \in \text{pred}(b)} AVAIL_{out}(x)$$

Putting it all together

- Step 1 — Compute a CFG for your code
- Step 2 — Compute **DEF** & **KILL** sets for basic blocks
- Step 3 — iteratively compute **AVAIL** sets using two rules

$$AVAIL_{out}(b) = DEF(b) \cup \left(AVAIL_{in}(b) - KILL(b)\right)$$

$$AVAIL_{in}(b) = \bigcap_{x \in \mathrm{pred}(b)} AVAIL_{out}(x)$$

  - ✦ ***or*** we can combine the two rules

$$AVAIL_{in}(b) = \bigcap_{x \in \mathrm{pred}(b)} DEF(x) \cup \left(AVAIL_{in}(x) - KILL(x)\right)$$

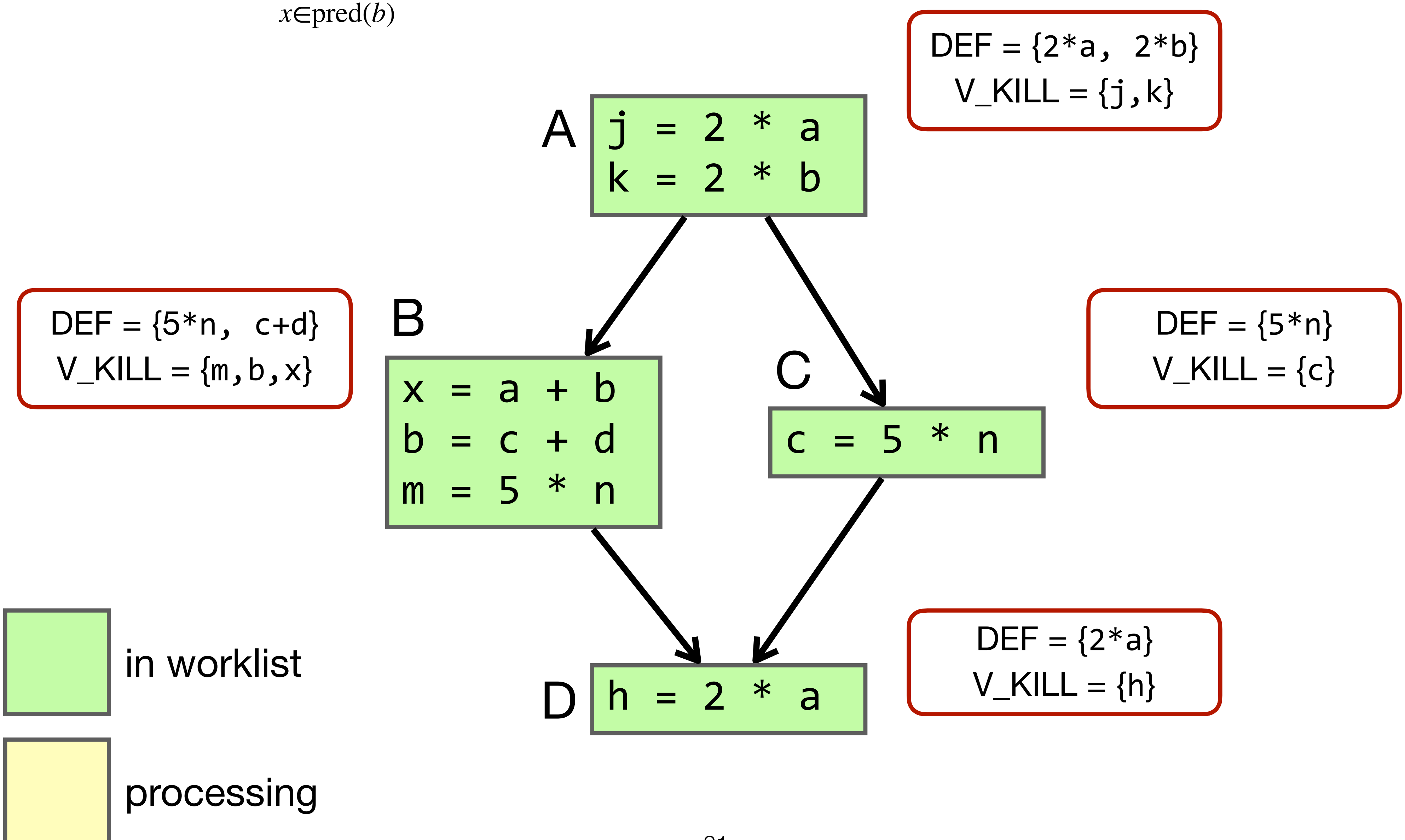
- Terminate when **AVAIL** sets stop changing

Iteratively Compute?

```
worklist = { all blocks b_k }
while (worklist ≠ ∅)
    remove some block b from Worklist
    new_avail = …                  // see prev. slide
    if (new_avail ≠ AVAIL(b)) // i.e. AVAIL(b) changed
        worklist = worklist ∪ successors(b)
    AVAIL(b) = new_avail
```

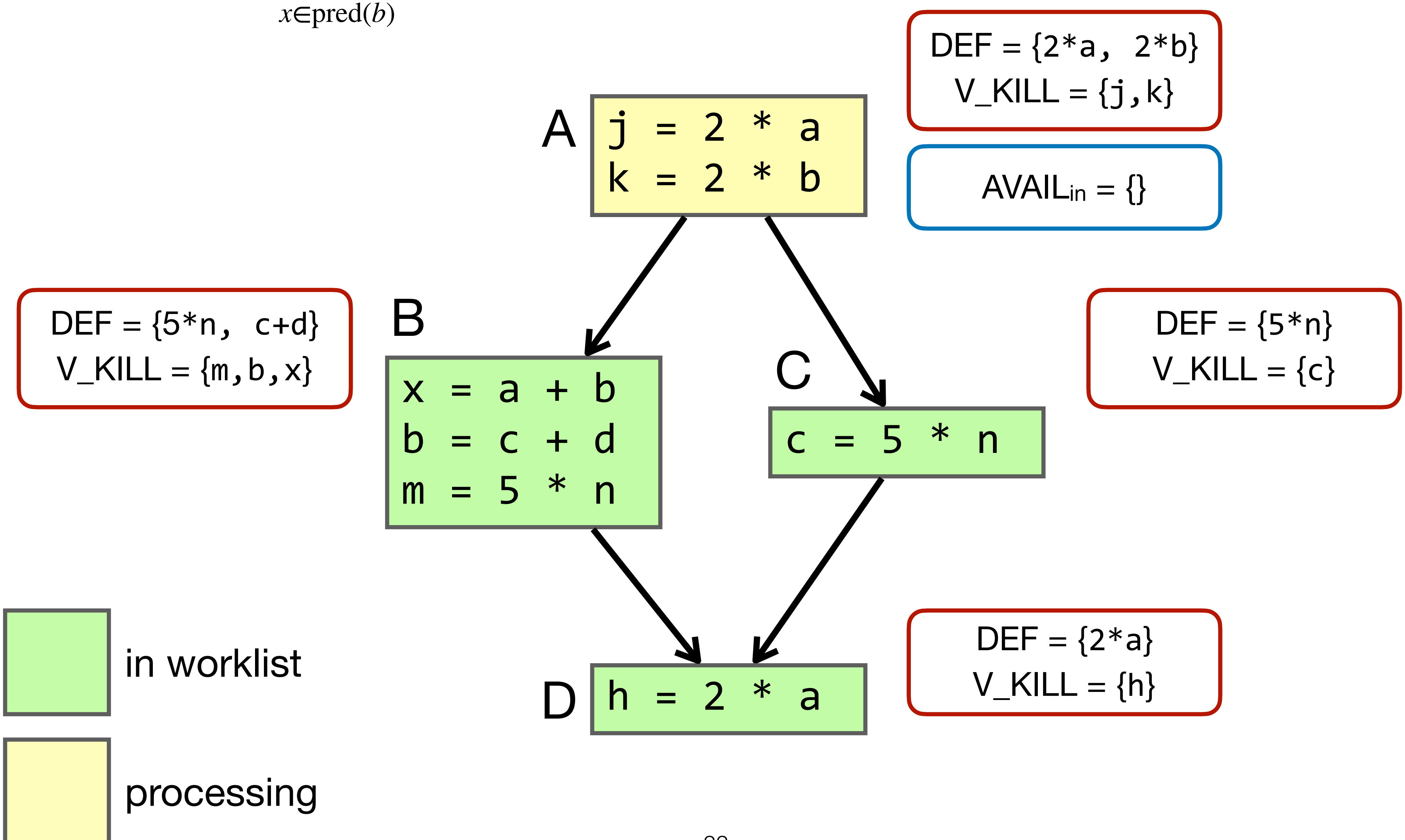
Example – Available Expressions

$$AVAIL_{in}(b) = \bigcap_{x \in \text{pred}(b)} DEF(x) \cup \left(AVAIL_{in}(x) - KILL(x)\right)$$



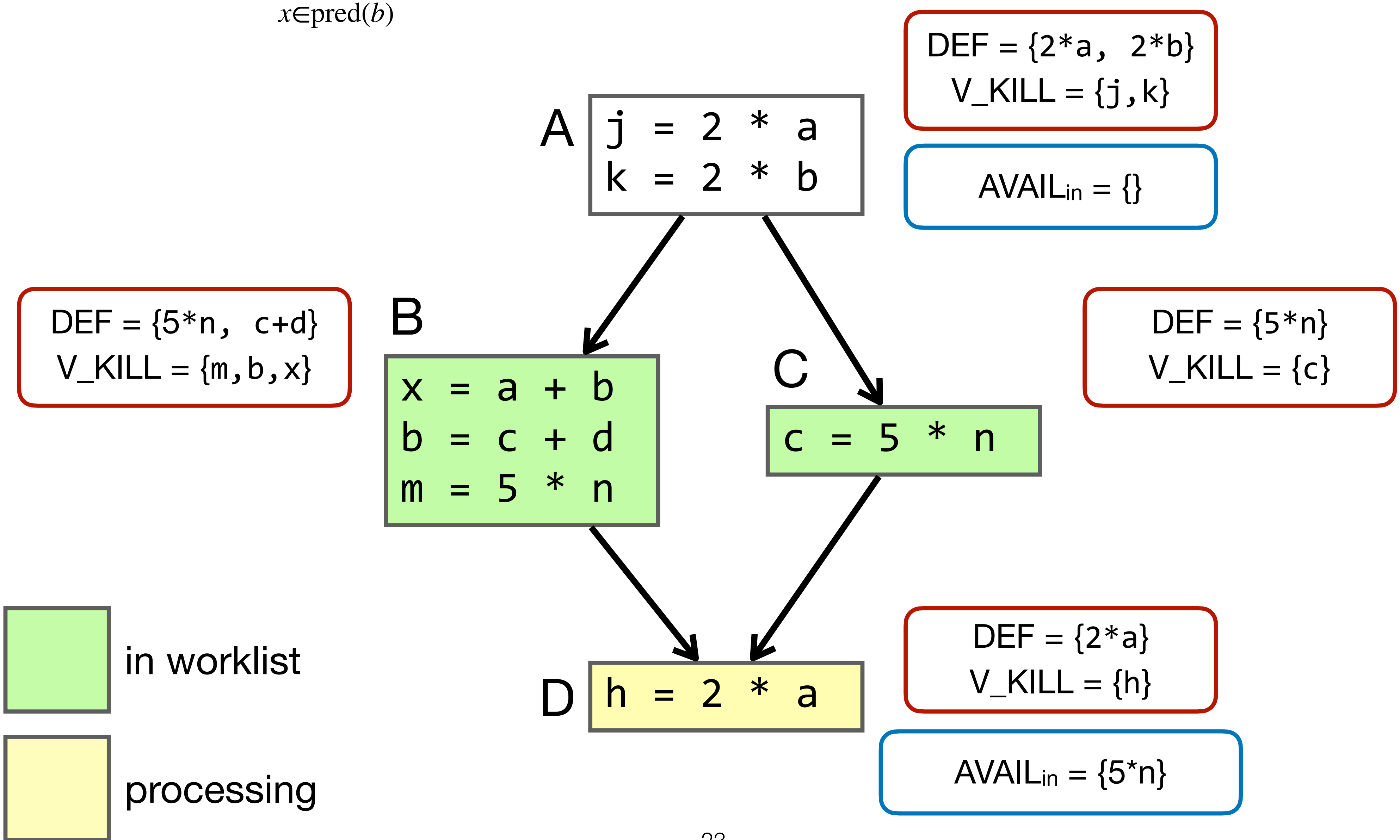
Example – Available Expressions

$$AVAIL_{in}(b) = \bigcap_{x \in \text{pred}(b)} DEF(x) \cup \left(AVAIL_{in}(x) - KILL(x)\right)$$



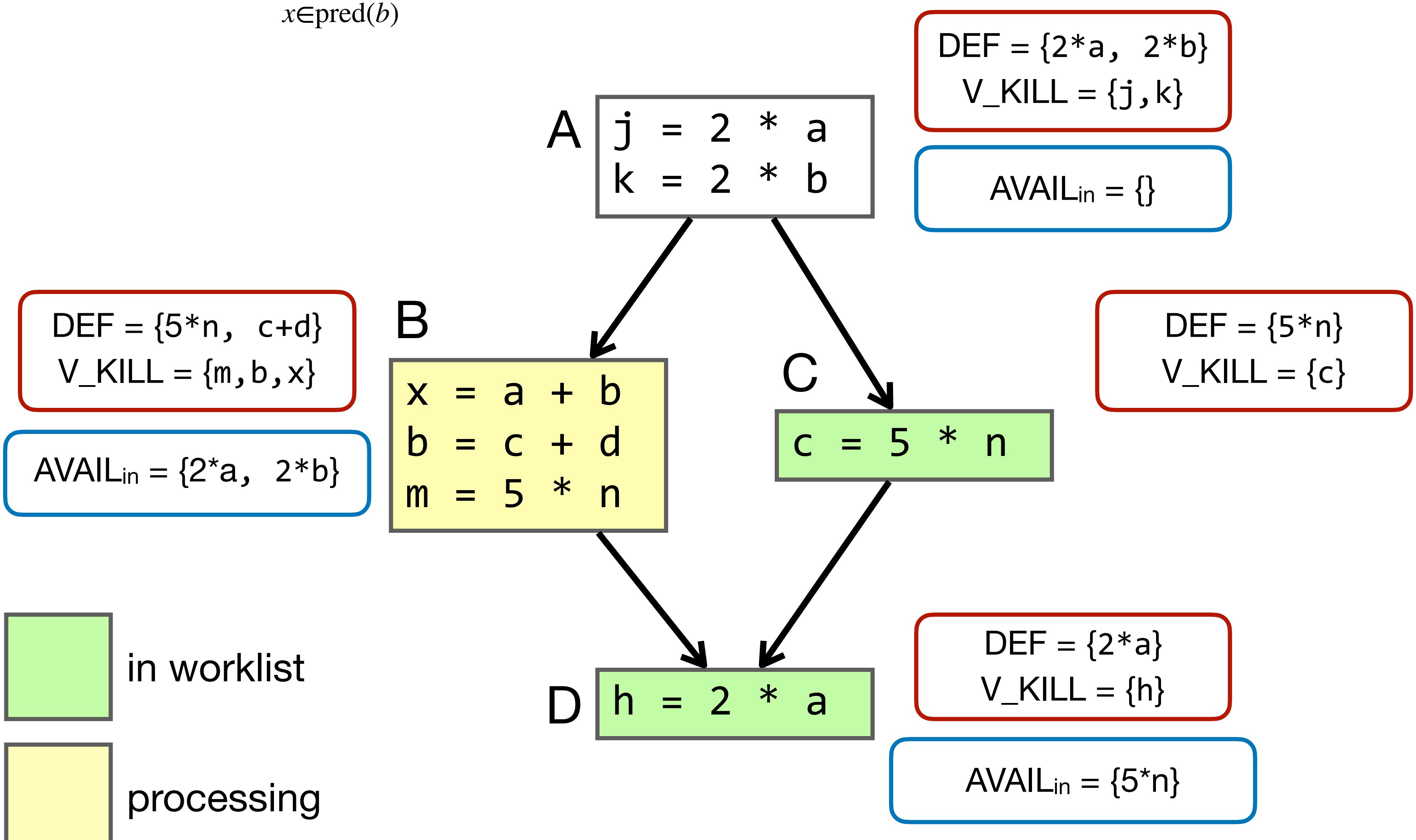
Example — Available Expressions

$$AVAIL_{in}(b) = \bigcap_{x \in \text{pred}(b)} DEF(x) \cup \left(AVAIL_{in}(x) - KILL(x)\right)$$



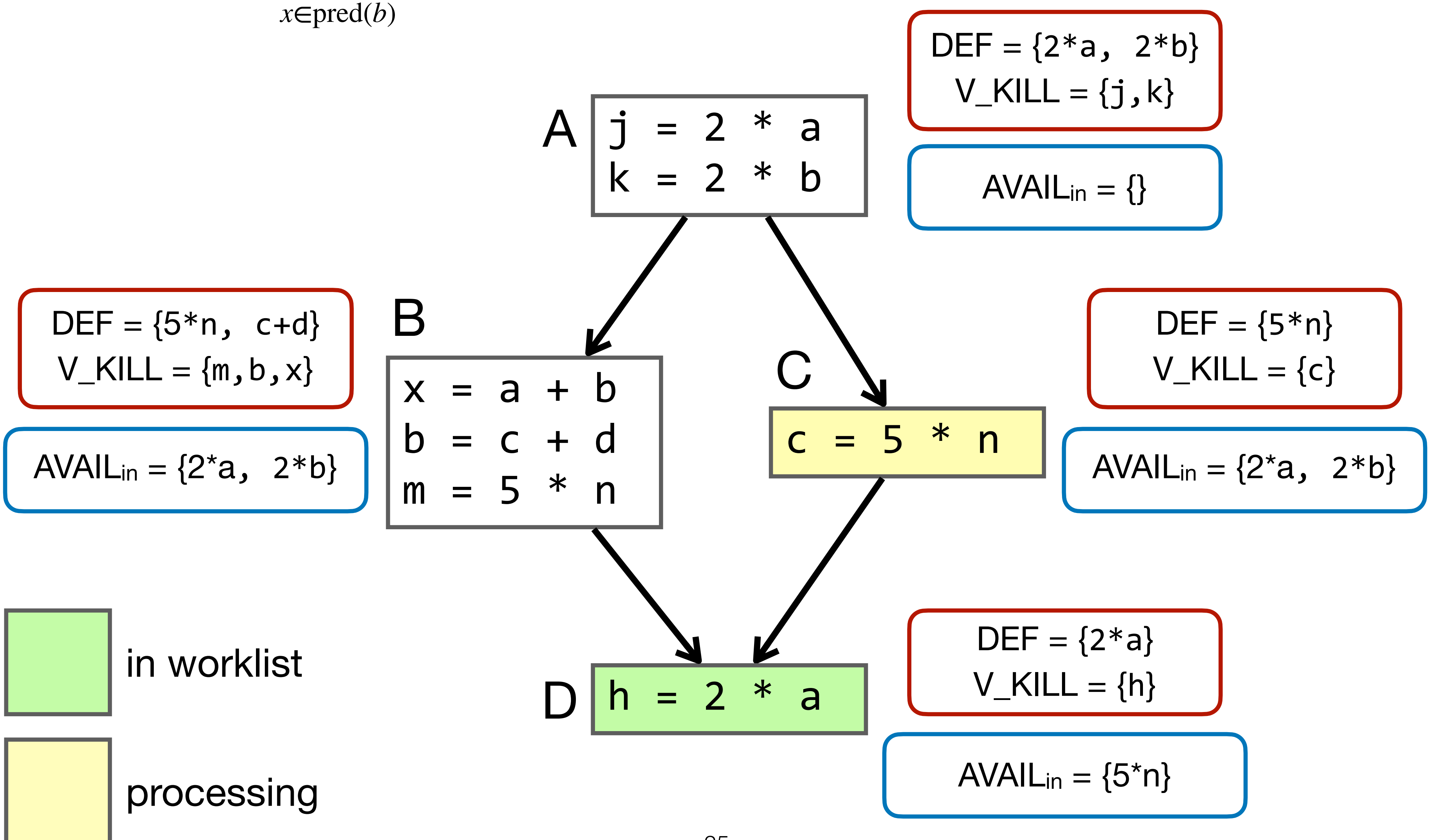
Example – Available Expressions

$$AVAIL_{in}(b) = \bigcap_{x \in \text{pred}(b)} DEF(x) \cup \left(AVAIL_{in}(x) - KILL(x)\right)$$



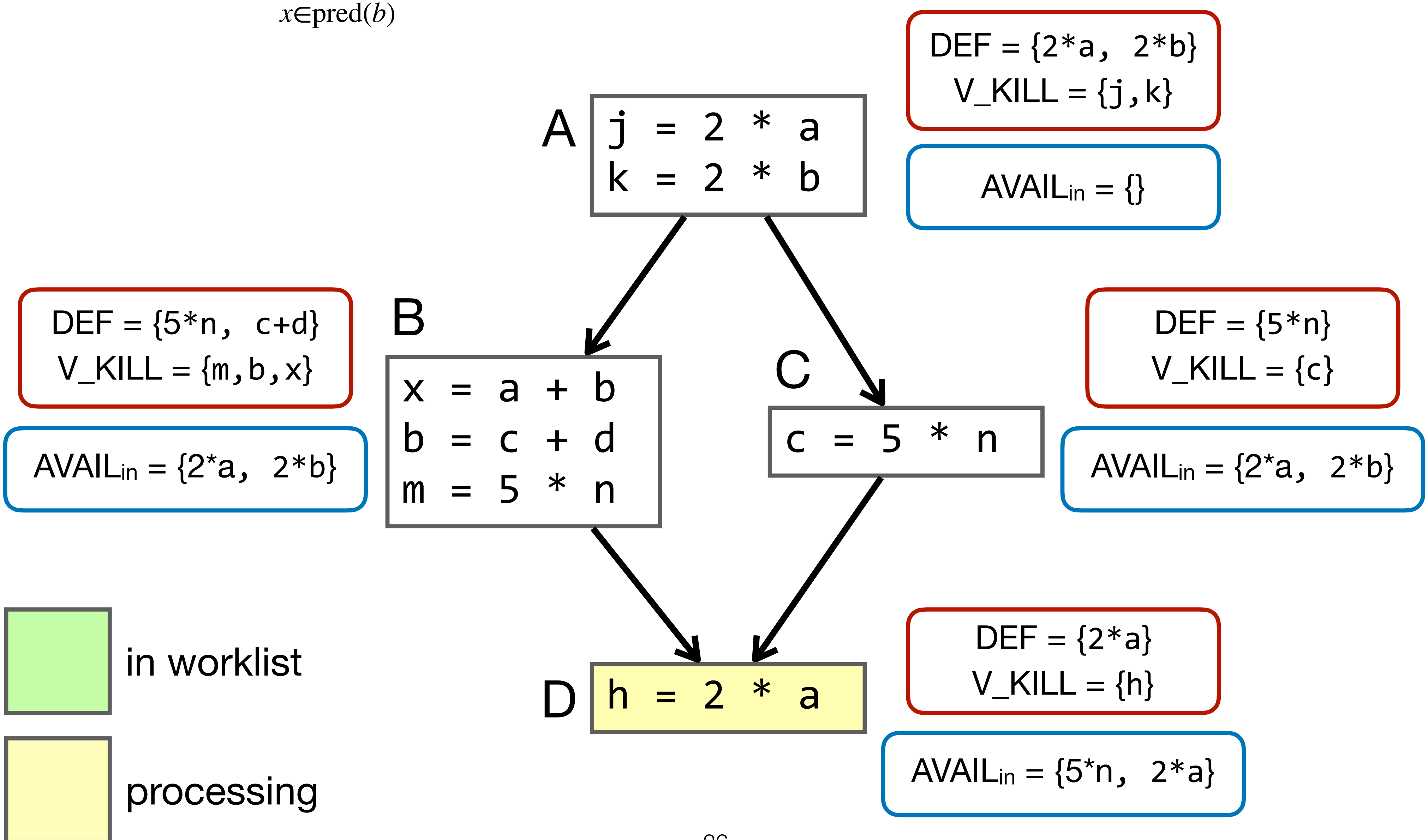
Example — Available Expressions

$$AVAIL_{in}(b) = \bigcap_{x \in \mathrm{pred}(b)} DEF(x) \cup \left(AVAIL_{in}(x) - KILL(x)\right)$$



Example — Available Expressions

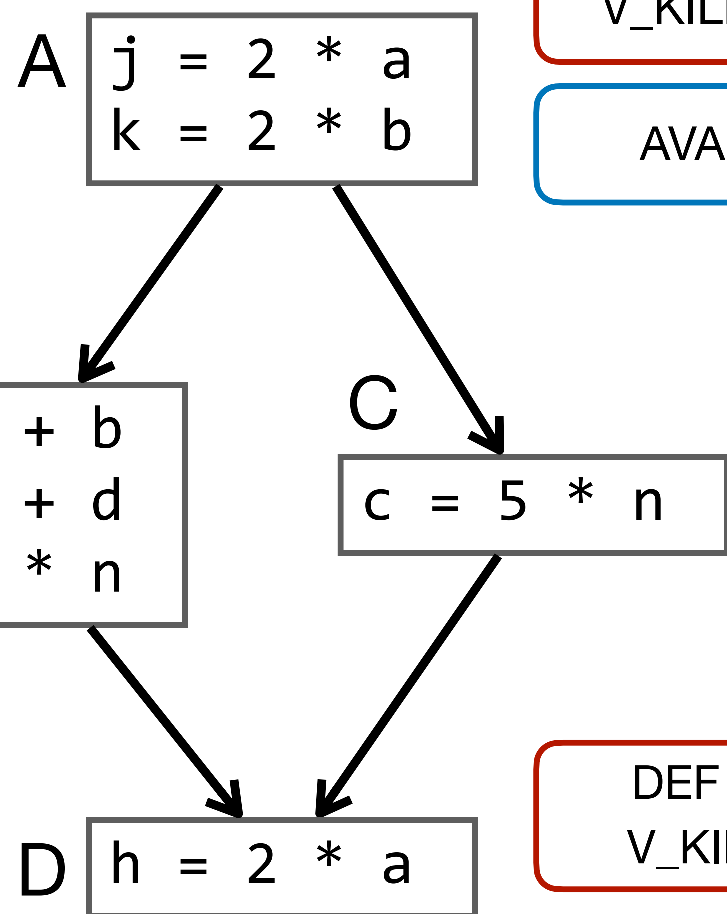
$$AVAIL_{in}(b) = \bigcap_{x \in \text{pred}(b)} DEF(x) \cup \left(AVAIL_{in}(x) - KILL(x)\right)$$



Example — Available Expressions

$$AVAIL_{in}(b) = \bigcap_{x \in \text{pred}(b)} DEF(x) \cup \left(AVAIL_{in}(x) - KILL(x)\right)$$

So, the common subexpressions are…?



A:
DEF = {2*a, 2*b}
V_KILL = {j,k}
$AVAIL_{in}$ = {}

B:
DEF = {5*n, c+d}
V_KILL = {m,b,x}
$AVAIL_{in}$ = {2*a, 2*b}

C:
DEF = {5*n}
V_KILL = {c}
$AVAIL_{in}$ = {2*a, 2*b}

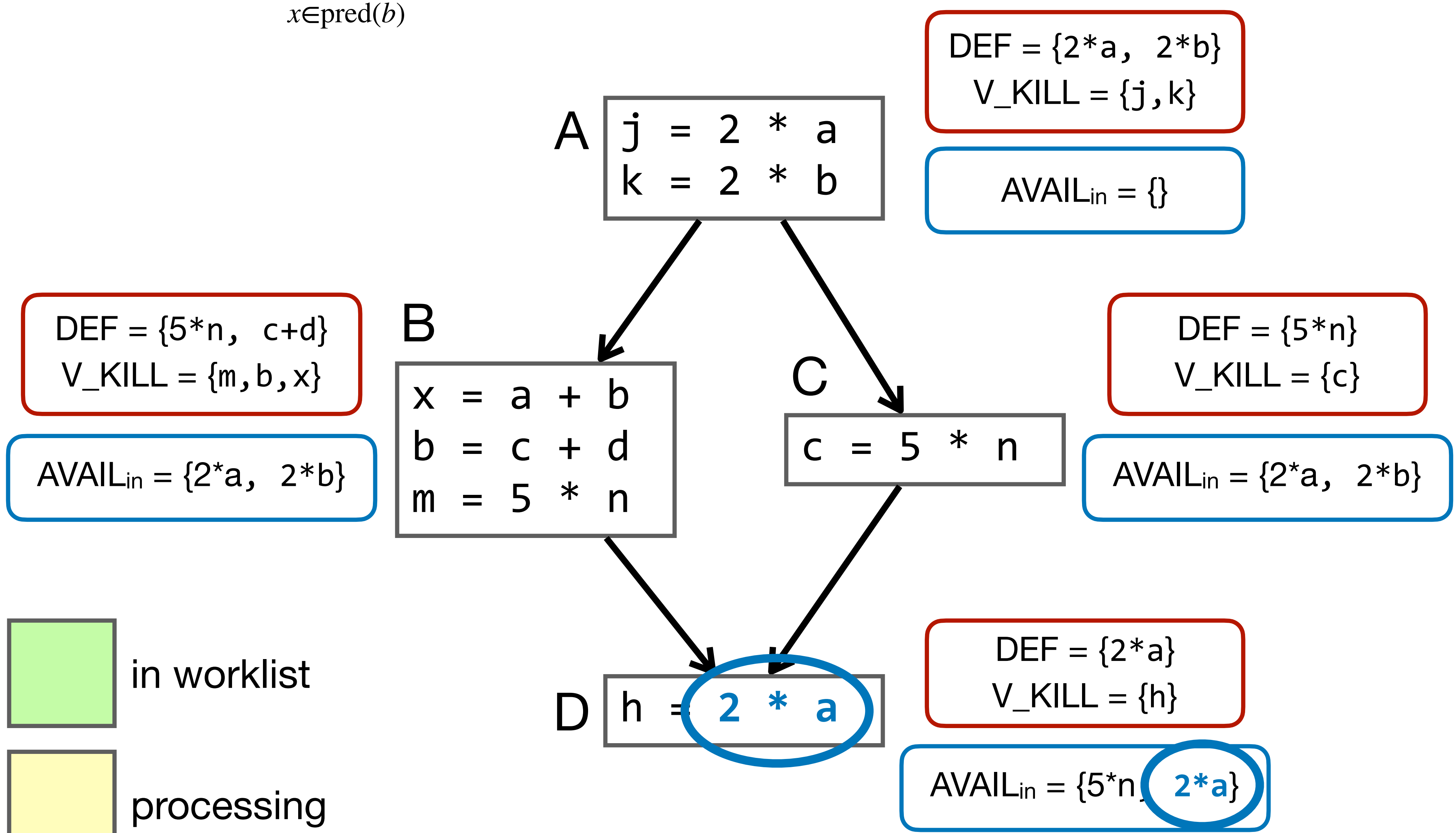
D:
DEF = {2*a}
V_KILL = {h}
$AVAIL_{in}$ = {5*n, 2*a}

in worklist

processing

Example – Available Expressions

$$AVAIL_{in}(b) = \bigcap_{x \in \mathrm{pred}(b)} DEF(x) \cup \left(AVAIL_{in}(x) - KILL(x)\right)$$



Termination

- Consider the update equation for AVAIL

$$AVAIL_{in}(b) = \bigcap_{x \in \text{pred}(b)} DEF(x) \cup \left(AVAIL_{in}(x) - KILL(x)\right)$$

- This equation has a property called ***monotonicity***
  - ✦ If $AVAIL_{in}(x)$ gets larger, then $AVAIL_{in}(b)$ can't shrink
  - ✦ As we propagate information, AVAIL sets will only get bigger
  - ✦ And there's a finite set of expressions in a method, so…
- Ahh, another fixed-point algorithm…

Outline

Constant Analysis Revisited

Example — Available Expressions for CSE

Dataflow in General

Liveness Analysis

Other Analyses

Optimize! CSE, Copy Propagation, DCE

Applications of Dataflow Analysis

- Constant Propagation & Available Expressions are examples of ***dataflow analyses***
- There are many other dataflow analyses that all share a common algorithmic framework and theoretical foundation
- Dataflow analyses can allow us to ***prove*** the conditions that are necessary to ensure optimizations are safe (i.e. preserve equivalence)
- Dataflow analyses can also be used to detect properties that ***prove absence of certain errors*** or suggest likelihood of potential problems (i.e. enhance Checking behavior)

Dataflow for Checking & Beyond

- e.g. the Checker Framework for Java (developed by UW prof. Mike Ernst, and team)
  - used on all Java code at major software companies (Google, Meta, Amazon, Uber, et al.)
- Example analyses
  - nullable/non-nullable (this variable cannot be null)
  - initialized fields (data fields should be initialized)
- More generally, dataflow analyses are subsumed by more powerful program analysis frameworks: static analysis & abstract interpretation

Dataflow, Abstractly

- 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
- Given such sets, we have an equation like
  - ✦ **OUT(b)** = **GEN(b)** ∪ (**IN(b)** - **KILL(b)**)
- We can solve these equations iteratively for all blocks by using a fixed-point algorithm

Dataflow, *Variations*

- There are both dataflow analyses that run forwards and backwards, depending on how we define our facts
  - Forward: **OUT(b)** = **GEN(b)** ∪ (**IN(b)** - **KILL(b)**)
  - Backward: **IN(b)** = **GEN(b)** ∪ (**OUT(b)** - **KILL(b)**)
- We can also merge facts using two basic methods
  - facts true on all paths: **IN(b)** = $\bigcap_{x}$ **OUT(x)**
  - facts true on any path: **IN(b)** = $\bigcup_{x}$ **OUT(x)**

Implementation Efficiency

- Encoding Sets
  - We can use `set<…>` data structures. This is simple, but rarely efficient, especially when the largest possible set is quite small. (e.g. less than 64 or 128 items)
  - Given a fixed set (e.g. of possible expressions) assign a number to each possible element of the set. Then use a bit-vector to represent the set; set operations can now be implemented by bit-logic
- Worklist priority — forward analyses should prefer to process the CFG from start to finish; backwards analyses should prefer to process the CFG in the opposite order

Outline

Constant Analysis Revisited

Example — Available Expressions for CSE

Dataflow in General

Liveness Analysis

Other Analyses

Optimize! CSE, Copy Propagation, DCE

Live Variable Analysis

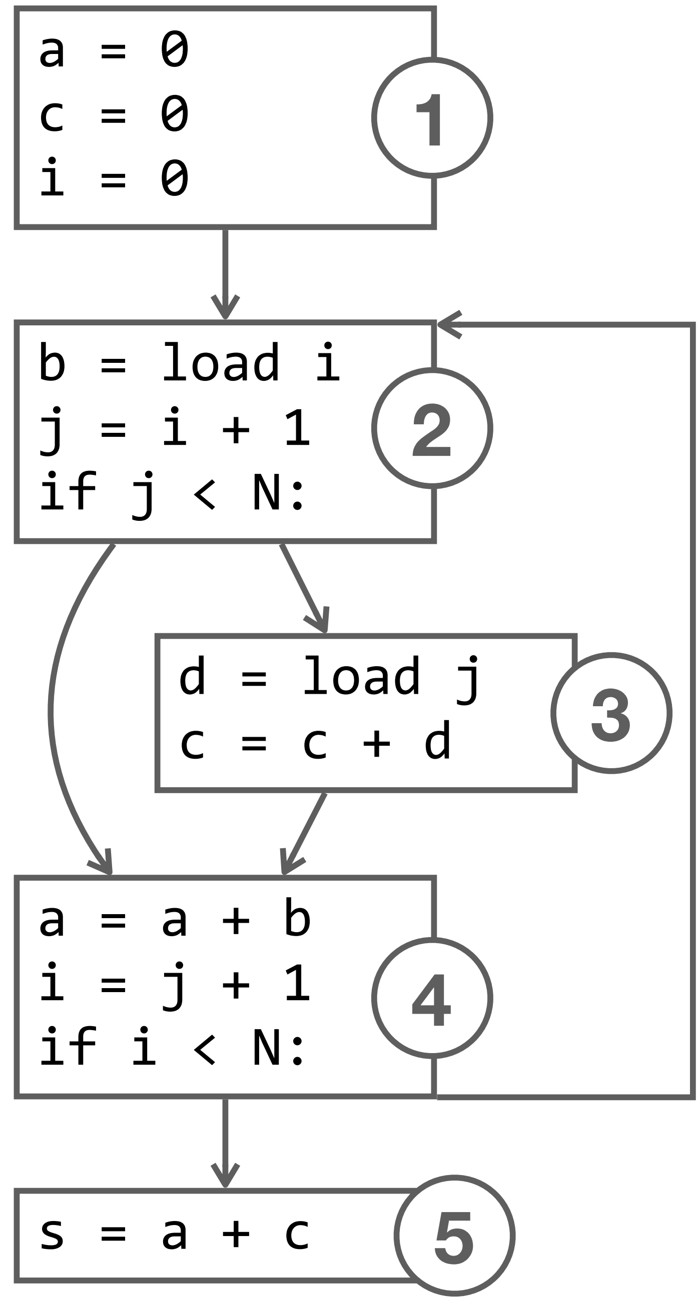
- A variable v is ***live*** at a program point p iff. there is *some* path from p to a use of v, along which v is not redefined.
- Applications
  - ✦ *Register allocation* — only live variables need to be assigned a register
  - ✦ *Dead Code Elimination* — if a variable is not live immediately after being written, then it doesn't need to be computed
  - ✦ *Detecting uninitialized variables* — if live at declaration (before initialization) then it might be used uninitialized
  - ✦ *Better SSA construction* — see next lecture

Liveness Equations

- For each block b, define
  - $USE[b]$ — variables used in b before any redefinition
  - $DEF[b]$ — variable defined in b
  - $IN[b]$ — variables live on entry to b
  - $OUT[b]$ — variables live on exit from b
- Update equations
  - $IN[b] = USE[b] \cup \left(OUT[b] - DEF[b]\right)$
  - $OUT[b] = \bigcup_{x \in \mathrm{succ}[b]} IN[x]$

Example — Liveness Analysis

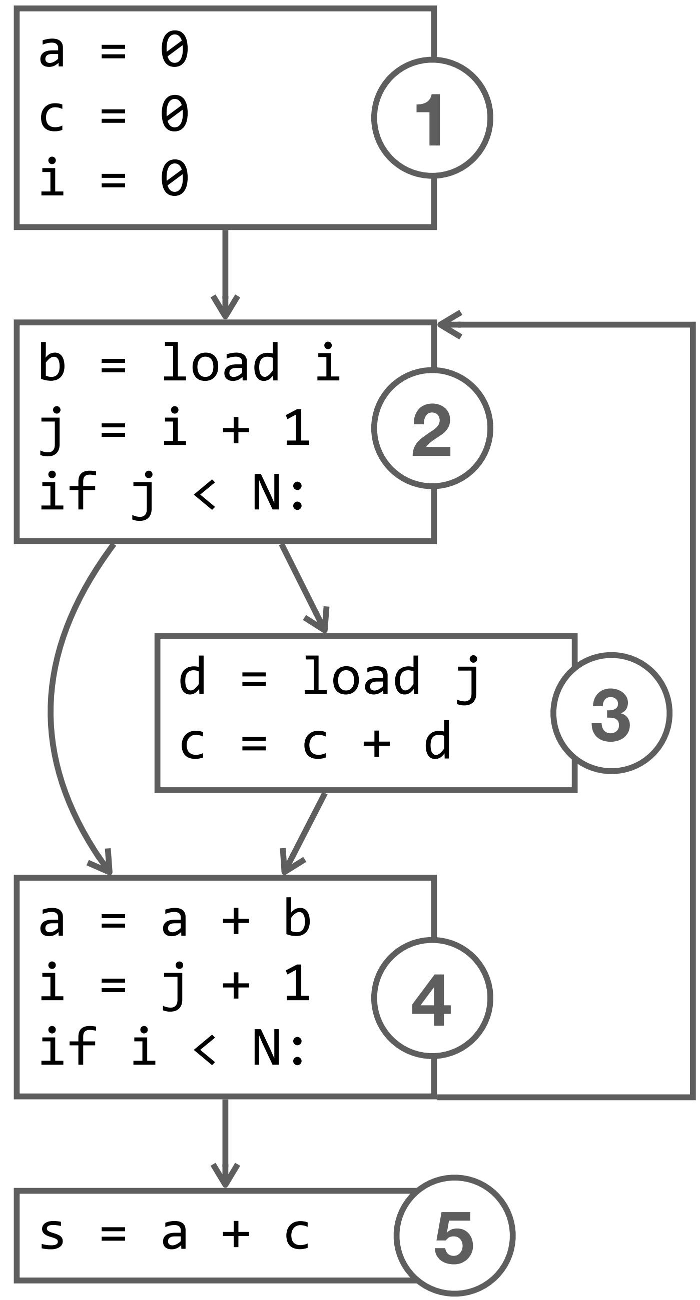
```
a = 0
c = 0
i = 0
do
  b = load i
  j = i + 1
  if j < N:
    d = load j
    c = c + d
  a = a + b
  i = j + 1
while i < N
s = a + c
```



Calculation

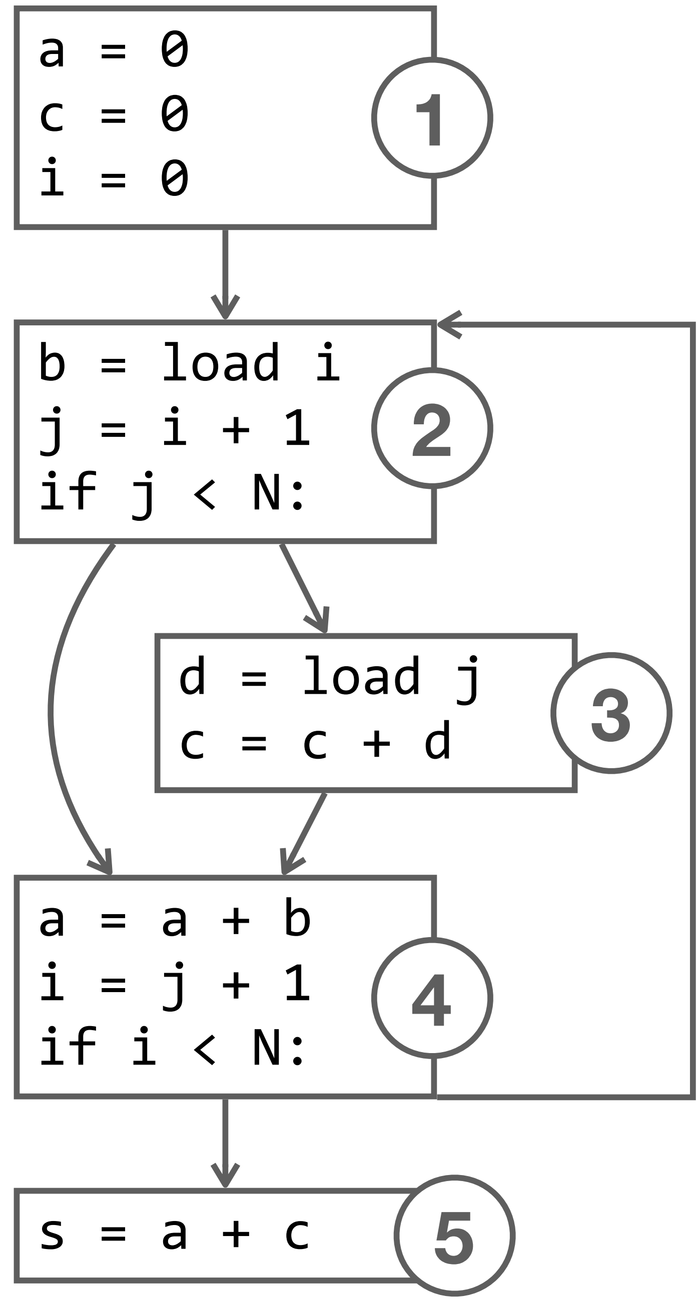
- $IN[b] = USE[b] \cup \left(OUT[b] - DEF[b]\right)$
- $OUT[b] = \bigcup_{x \in \mathrm{succ}[b]} IN[x]$

			iter 1		iter 2	
b	USE	DEF	IN	OUT	IN	OUT
1						
2						
3						
4						
5						



Calculation

			iter 1		iter 2	
b	USE	DEF	IN	OUT	IN	OUT
1		a,c,i	N	a,c, i,N	N	a,c, i,N
2	N,i	b,j	a,c, i,N	a,c,b, j,N	a,c, i,N	a,c,b, j,N
3	j,c	d,c	a,c,b, j,N	a,c,b, j,N	a,c,b, j,N	a,c,b, j,N
4	a,b, j,N	a,i	a,c,b, j,N	a,c	a,c,b, j,N	a,c,j, N
5	a,c	s	a,c		a,c	



Equivalent Formulations

- If you read the textbook & other sources, you'll find different choices of terminology. e.g. this is also ***liveness***
- Facts
  - ✦ $USED(b)$ — variables used in b prior to any def in b
  - ✦ $NOTDEF(b)$ — variables not defined in b
  - ✦ $LIVE(b)$ — variables live on *exit* from b
- Equation

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

Outline

Constant Analysis Revisited

Example — Available Expressions for CSE

Dataflow in General

Liveness Analysis

Other Analyses

Optimize! CSE, Copy Propagation, DCE

Reaching Definitions

- A definition (statement) d of some variable v ***reaches*** another statement s iff. s reads the value of v and there is a path from d to s that does not redefine v
  - Requires that we assign numbers/identifiers to each possible definition statement d
- Applications
  - Find all of the possible definition points for a variable in an expression

Facts & Eqs — Reaching Defs

- Facts
  - $DEFOUT(b)$ — set of definitions in b that reach the end of b (i.e. without being redefined)
  - $DEFKILL(b)$ — set of definitions killed by redefining a variable in b (possibly stored by variable, instead of def)
  - $REACH(b)$ — set of definitions that reach b
- Equation

$$REACH(b) = \bigcup_{p \in \mathrm{pred}(b)} DEFOUT(p) \cup \left(REACHES(p) - DEFKILL(p)\right)$$

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.
- Applications
  - Code Hoisting — If an expression e is very busy at p, then we can move the computation of e to p
  - Loop-Invariant Code Hoisting — in particular, if we can hoist computations to outside of loops then there is a strong chance we will speed up the code

Facts & Eqs — Very Busy Expr

- Facts
  - ✦ $USED(b)$ — expressions used in b before they are killed
  - ✦ $KILL(b)$ — expressions redefined in b before they are used (likely stored as redefined variables)
  - ✦ $VERYBUSY(b)$ — expressions very busy on exit from the block b
- Equation

$$VERYBUSY(b) = \bigcap_{s \in \mathrm{succ}(b)} USED(s) \cup \left(VERYBUSY(s) - KILL(s)\right)$$

Dominance

- A basic block b ***dominates*** a basic block d iff. all paths to d (from the start of the CFG) must pass through b
- Applications
  - Many!
  - Detect loops
  - help perform CSE
  - help convert to SSA

Facts & Eqs — Dominance

- Facts
  - $SELF(b) = \{b\}$
  - $DOM\_BY(b)$ — set of blocks that dominate b
- Equation

$$DOM\_BY(b) = \bigcap_{d \in \mathrm{pred}(b)} SELF(d) \cup DOM\_BY(d)$$

Reaching *Expressions*

- A mash-up of available expressions and reaching def.
- A statement $s = \left(z := x \ op \ y\right)$ ***reaches*** another statement ***as an expression*** t iff. neither x nor y are redefined along all paths from s to t
- Applications
  - We will use for CSE in a second
- Omit Equations

Outline

Constant Analysis Revisited

Example — Available Expressions for CSE

Dataflow in General

Liveness Analysis

Other Analyses

Optimize! CSE, Copy Propagation, DCE

Classic Common-Subexpression Elimination (CSE)

- In a statement $s = \left(z := x \ op \ y\right)$ if $x \ op \ y$ is ***available*** at s, then it does not need to be recomputed
- *However*, we also need to replace the common subexpression $x \ op \ y$ with the variable from some specific definition
- Fix — If there is a statement $a = (w := x \ op \ y)$ that ***reaches*** s ***as an expression***, and the block containing a dominates the block containing s, then we can perform common subexpression elimination at s using a

Classic CSE Transformation

- Suppose $a = (w := x\ op\ y)$ both dominates $s = (z := x\ op\ y)$ and reaches s as an expression.
- First, create a new temporary t_i for w, and use it to rewrite the code

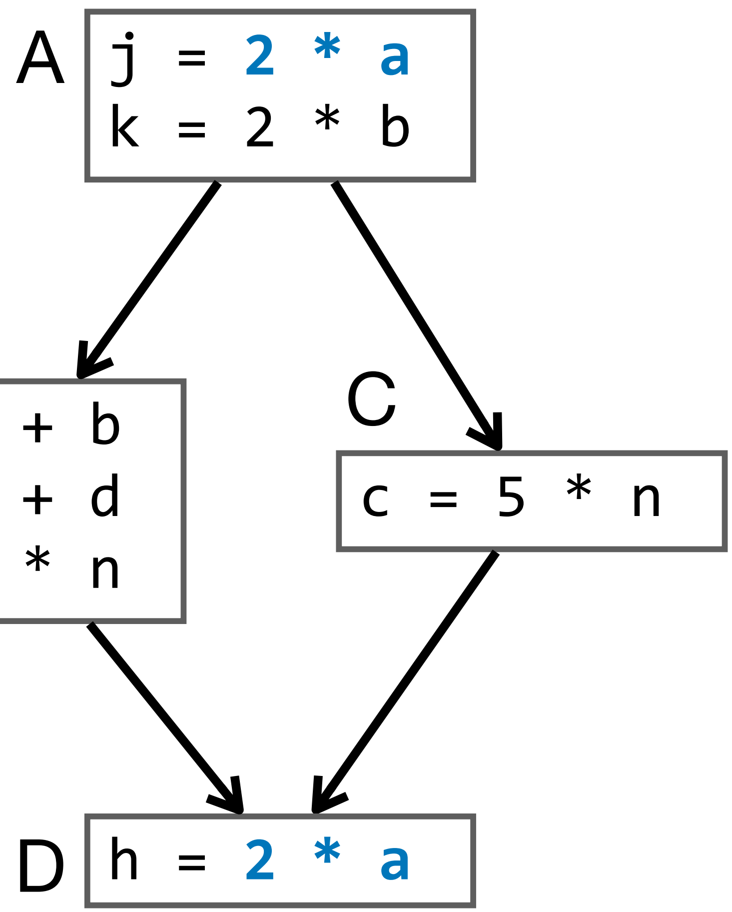
```
w := x op y
…
z := x op y
```

$\rightarrow$

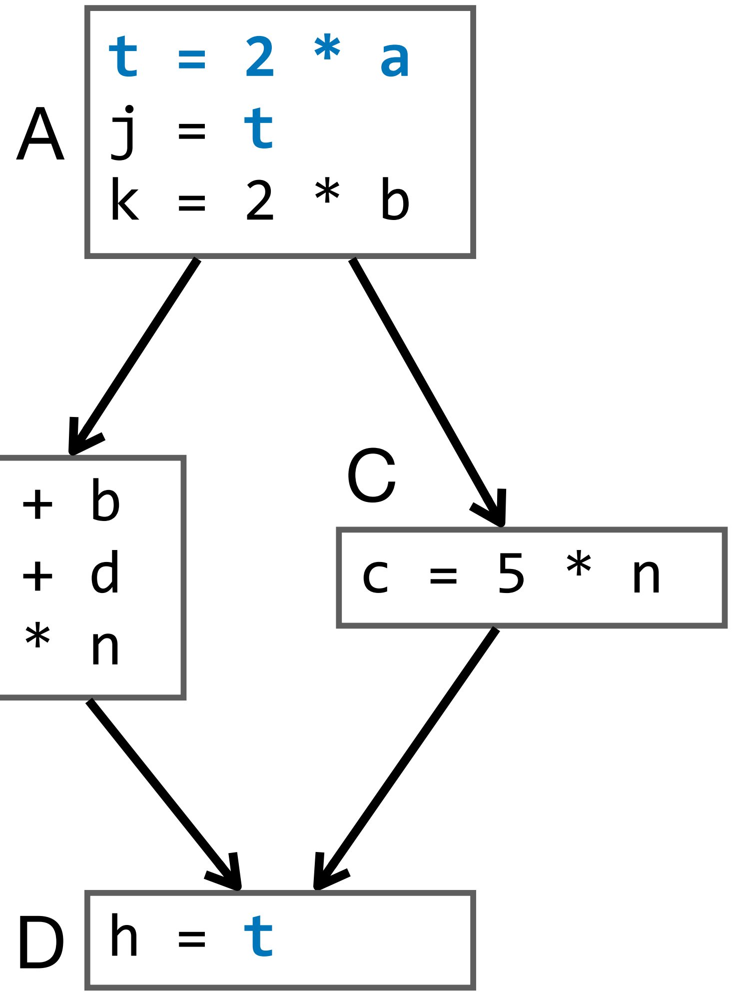
```
ti := x op y
w  := ti
…
z  := ti
```

- rely on copy propagation to remove extra assignments

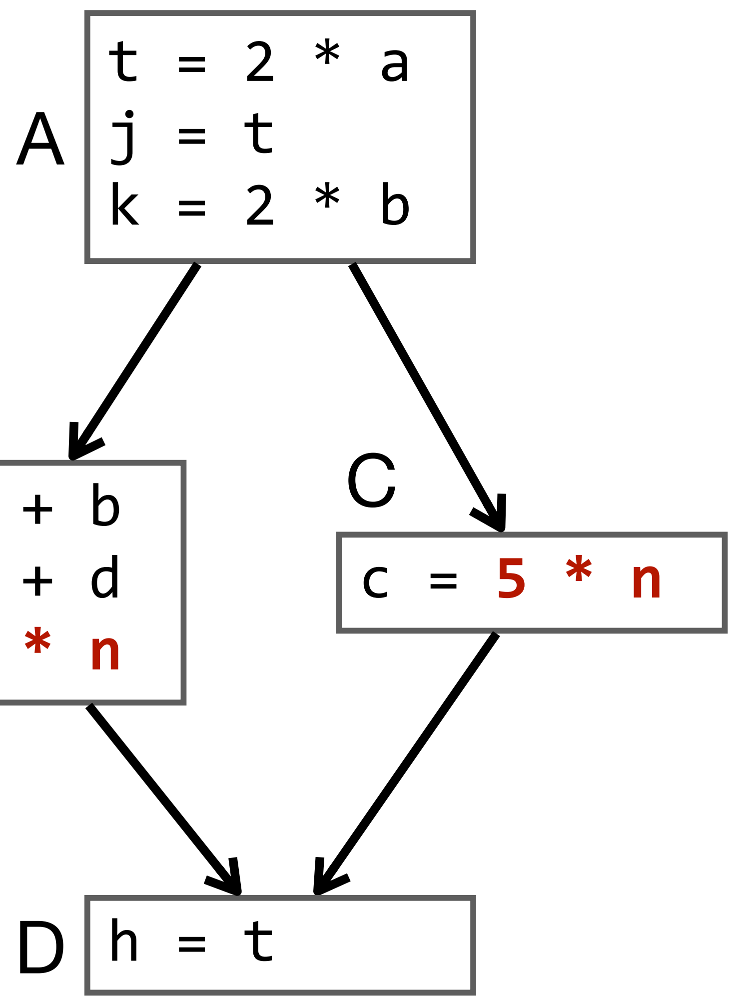
Revisiting the Example



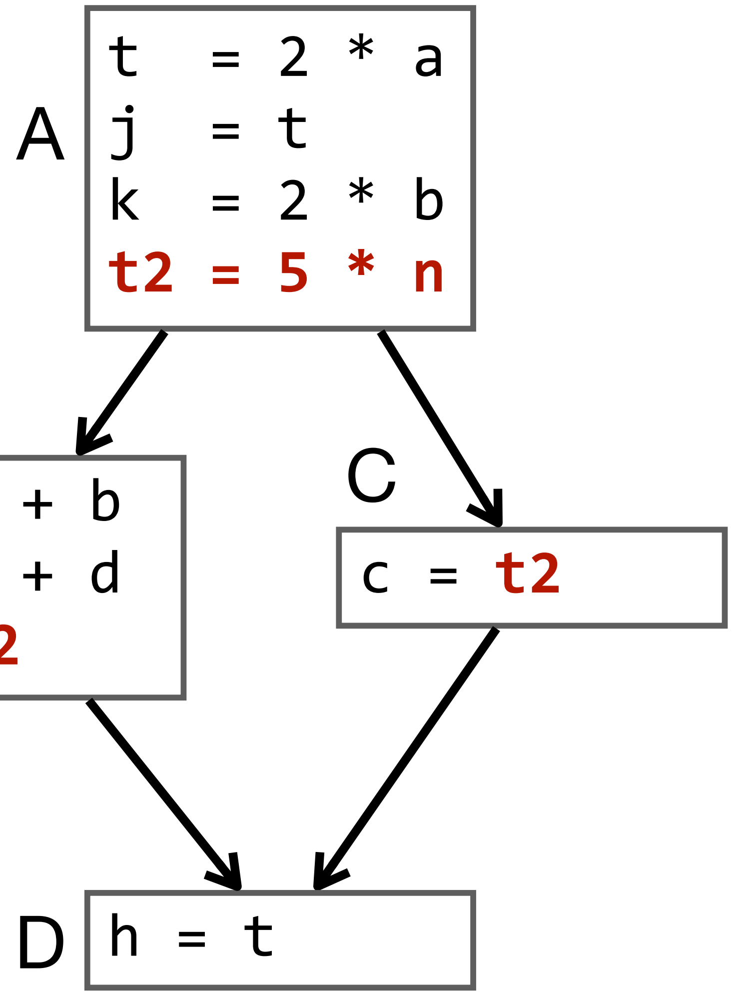
Revisiting the Example



Apply Very Busy



Apply Very Busy



Constant Propagation

an alternative method

- Suppose we have
  - statement $s = (x := c)$, where c is some literal
  - statement t that uses x
- Then if
  - the definition s ***reaches*** t and
  - no other definitions of x ***reach*** t,
- we can rewrite t to use c instead of x

Copy Propagation

- Similar idea to constant propagation
- Suppose we have
  - statement $s = (x := y)$
  - statement t that uses x
- Then if
  - the definition s ***reaches*** t and
  - no other definitions of x ***reach*** t,
  - there are no other definitions of y on any path from s to t
- we can rewrite t to use y instead of x

Copy Propagation Tradeoffs

- Copy propagation can expose opportunities for dead-code elimination (great!)
- ***But*** if copy propagation doesn't result in dead-code, it can increase the lifetime of a variable (and thus increase demand for registers and/or memory traffic — uh oh!)
- ***But*** copy propagation can expose other optimizations

```
a := y + z
u := y
c := u + z   // w/ copy propagation becomes y + z
```

  - Here, copy propagation exposes a CSE opportunity

Dead Code Elimination

- Suppose we have a statement $s = \left(a := b\ op\ c\right)$
- Then if
    - ✦ a is not ***live*** after s
- we can eliminate s
    - ✦ *provided it has no implicit side effects that are visible*
        - e.g. if $s = \left(a := foo(b, c)\right)$ then we can't eliminate the statement unless the compiler can somehow prove that foo definitely doesn't have side effects

In Summary…

- **Dataflow Analysis** is a general framework for discovering facts about programs (i.e. is *purely* analysis)
  - Can be generalized to more powerful frameworks too
- Then, the discovered facts open up opportunities for code optimization
- Another Approach is to *normalize* code into certain forms without changing its meaning. Doing this can accomplish many optimizations and simplify analyses
- Next time…
  - We'll study one of the most important normal forms for code — Static Single Assignment (SSA)