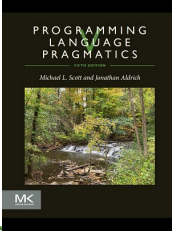
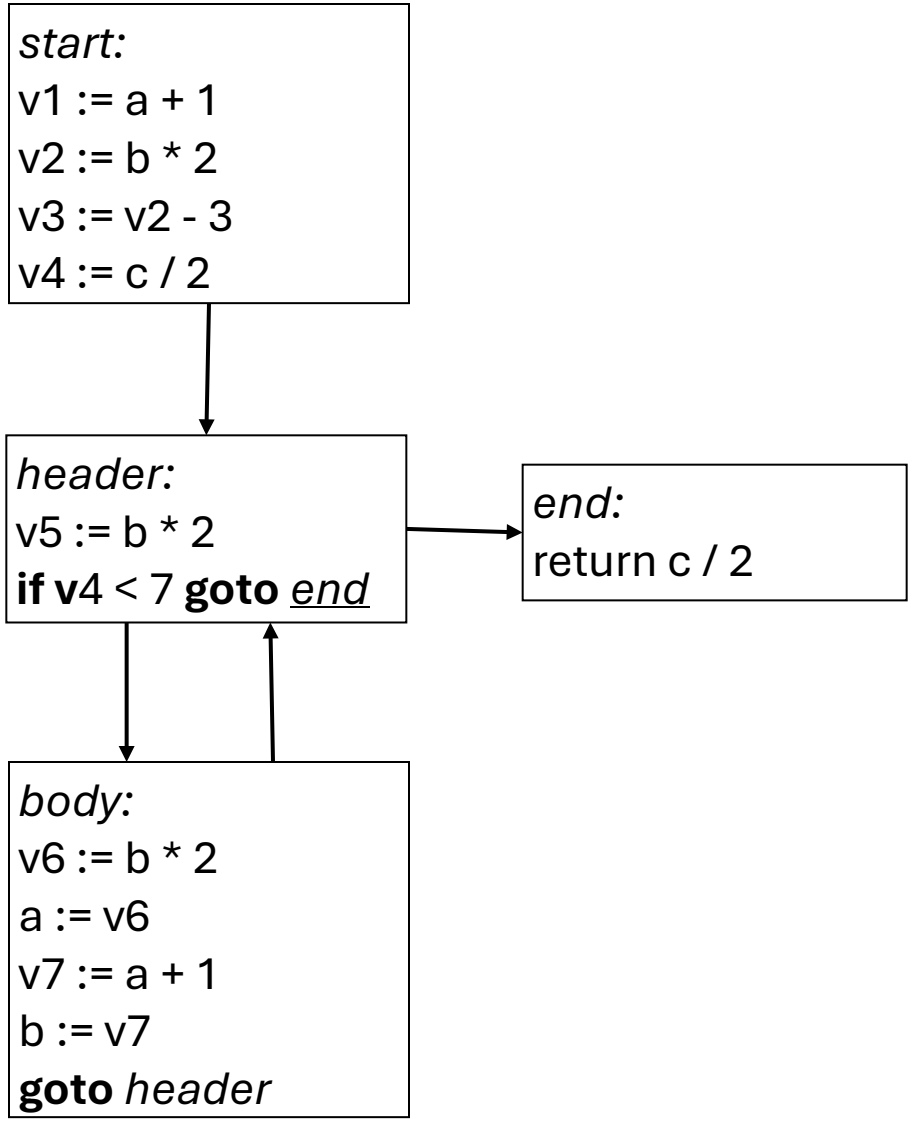


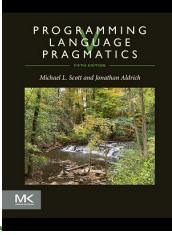
Data flow analysis

- Data flow analysis tracks the flow of information across basic block boundaries
- Examples:
  - **Available expressions:** which expressions are available in a (virtual) register?
  - **Reaching definitions:** which assignments to a variable reach a given program point?
  - **Constant propagation:** what variables hold constant values?
  - **Sign analysis:** is a variable positive, negative, zero, or of unknown sign?
  - **Range analysis:** what is the maximum and minimum value of a variable at a program point?

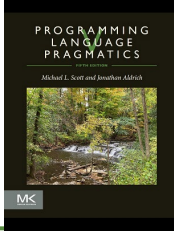
Example of available expressions



Data flow analysis frameworks



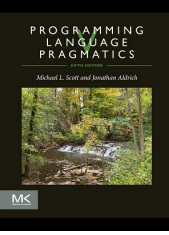
- Many instances of data flow analysis can be cast in the following framework:
  1. four sets for each basic block B, called In_B, Out_B, Gen_B, and $Kill_B$;
  2. values for the *Gen* and *Kill* sets;
  3. an equation relating the sets for any given block B;
  4. an equation relating the *Out* set of a given block to the *In* sets of its successors, or relating the *In* set of the block to the *Out* sets of its predecessors; and (often)
  5. certain initial conditions



Solving data flow analysis equations

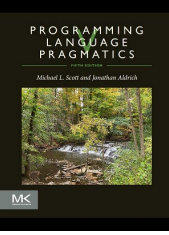
- The goal of the analysis is to find a *fixed point* of the equations: a consistent set of *In* and *Out* sets (usually the smallest or the largest) that satisfy both the equations and the initial conditions
  - Some problems have a single fixed point
  - Others may have more than one
    - we usually want either the least or the greatest fixed point (smallest or largest sets)

Global common subexpression elimination



- In the case of *global common subexpression elimination*, In_B is the set of expressions (virtual registers) guaranteed to be available at the beginning of block *B*
  - These *available expressions* will all have been set by predecessor blocks
  - *Out_B is the set of expressions guaranteed to be available at the end of B*
  - *$Kill_B$ is the set of expressions killed in B: invalidated by assignment to one of the variables used to calculate the expression, and not subsequently recalculated in B*
  - *Gen_B is the set of expressions calculated in B and not subsequently killed in B*

Available expression data flow equations



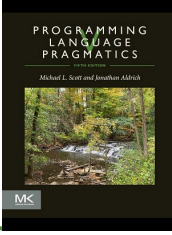
- The data flow equations for available expression analysis are:

$$Out_B = Gen_B \cup (In_B \setminus Kill_B)$$

$$In_B = \bigcap_{\text{predecessors } A \text{ of } B} Out_A$$

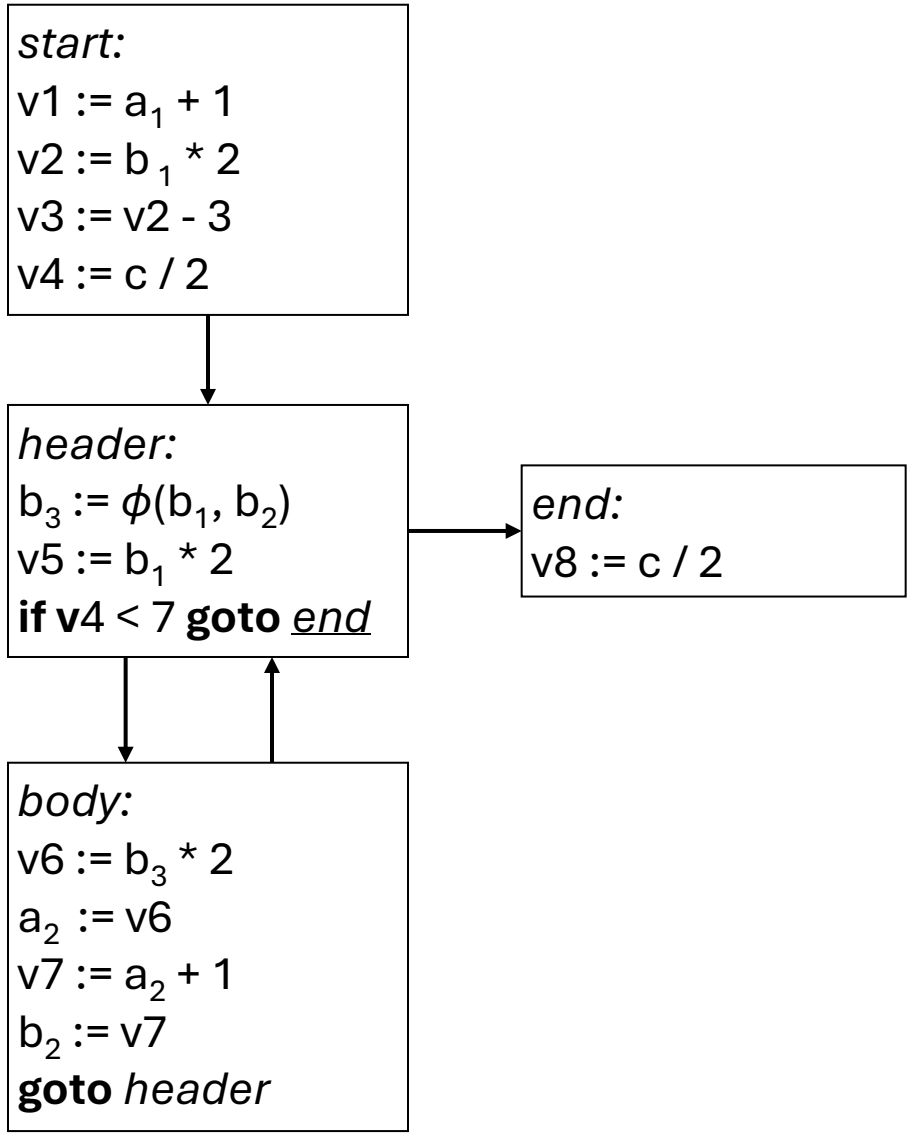
- Our initial condition is $In_1 = \varnothing$: no expressions are available at the beginning of execution

Solving data flow analysis equations



- Available expression analysis is known as a *forward* data flow problem, because information flows forward across branches: the *In* set of a block depends on the *Out* sets of its predecessors
  - We will see an example of a *backward* data flow problem later
- We calculate the desired fixed point of our equations in an inductive (iterative) fashion
- Our equation for In_B uses intersection to insist that an expression be available on all paths into *B*
  - In our iterative algorithm, this means that In_B can only shrink with subsequent iterations

Example of available expressions (revisited)



Partial redundancy elimination

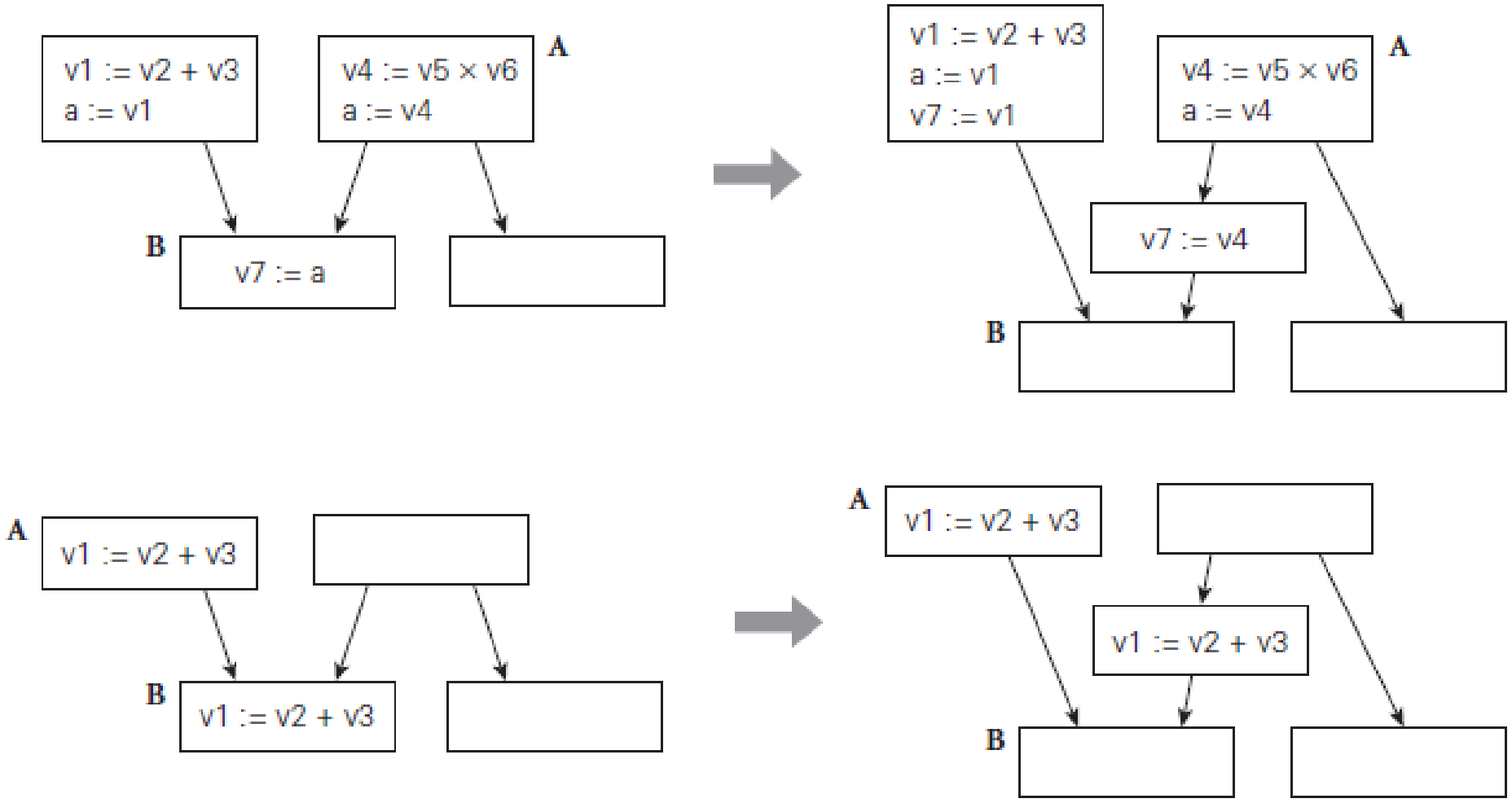
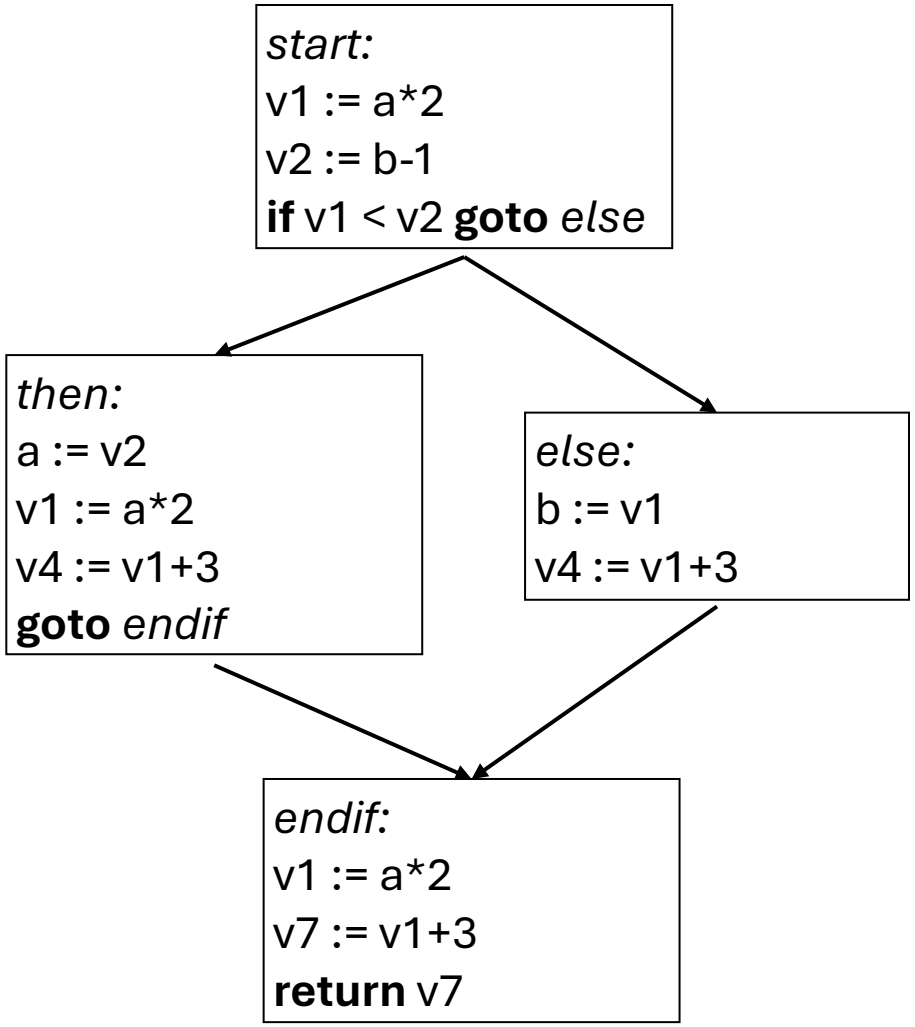


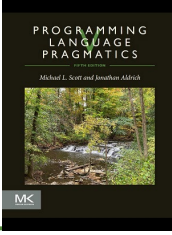
Figure 17.8 Splitting an edge of a control flow graph to eliminate a redundant load (*top*) or a partially redundant computation (*bottom*).

Check your understanding: available expressions



- Apply available expressions analysis to the control flow graph on the left

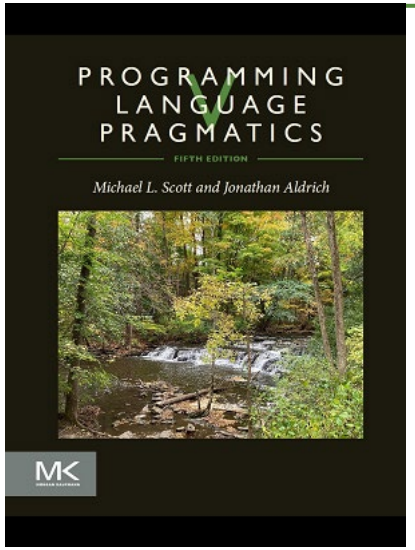
Analysis correctness



- Optimizations often rely on analysis information
  - Constant propagation: correspondences between variables and the particular known constant values that they will hold (or a token to represent that the value is unknown)

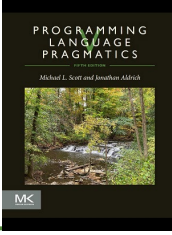
- Rough guide to correctness: when you replace symbolic information in the analysis with concrete information from particular executions, does the result hold?
  - Does the variable hold the claimed constant at run time?
  - Becomes a lemma in the proof of soundness for the “client” optimization

Section 17.4.2 part 2: Live Variable Analysis



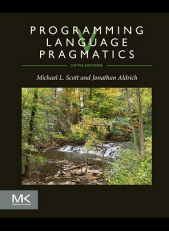
Programming Language Pragmatics, Fifth Edition

Michael L. Scott and Jonathan Aldrich



Live variable analysis

- We turn our attention to *live variable analysis*—*very* important in any subroutine in which global common subexpression analysis has eliminated load instructions
- Live variable analysis is a *backward* flow problem
- It determines which instructions produce values that will be needed in the future, allowing us to eliminate *dead* (useless) instructions
  - in our example we consider only values written to memory and with the elimination of dead stores
  - applied to values in virtual registers as well, live variable analysis can help to identify other dead instructions



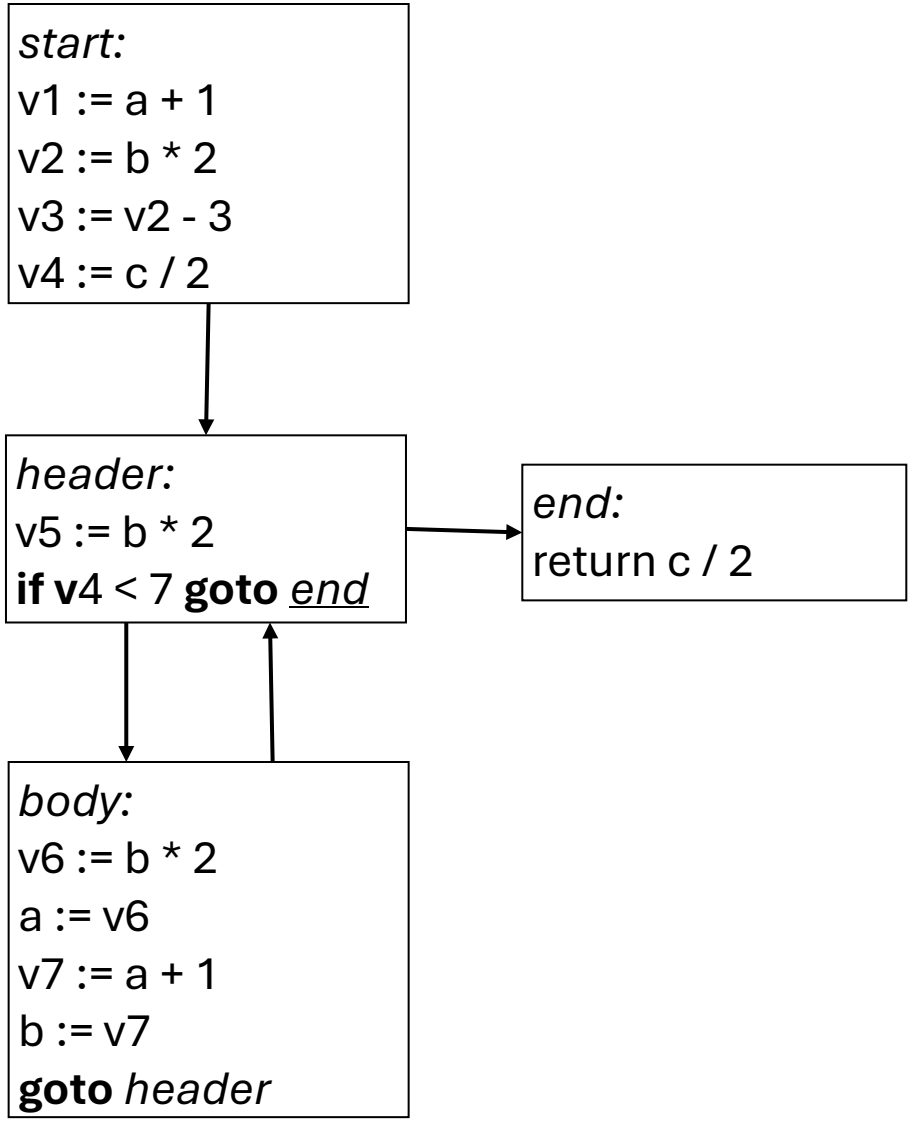
Defining live variable analysis

- For this instance of data flow analysis
  - *In_B is the set of variables live at the beginning of block B*
  - *Out_B is the set of variables live at the end of the block*
  - *Gen_B is the set of variables read in B without first being written in B*
  - *$Kill_B$ is the set of variables written in B without having been read first*
- The data flow equations are:

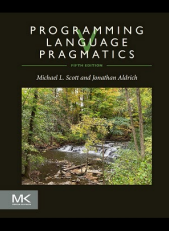
$$In_B = Gen_B \cup (Out_B \setminus Kill_B)$$

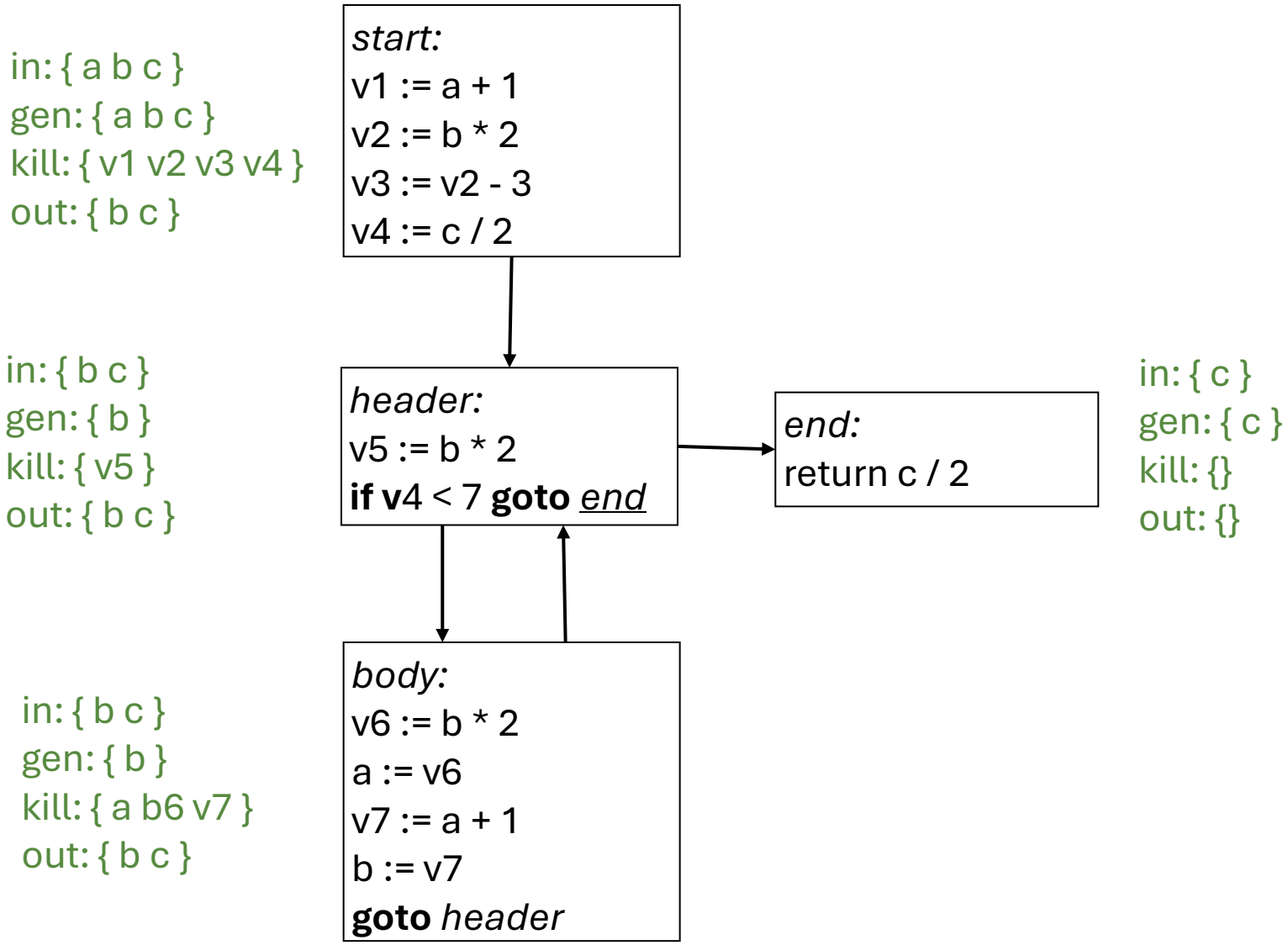
$$Out_B = \bigcup_{\text{successors } C \text{ of } B} In_C$$

Example of live variable analysis and dead code elimination



Example of live variable analysis and dead code elimination





Check your understanding: live variable analysis and dead code elimination