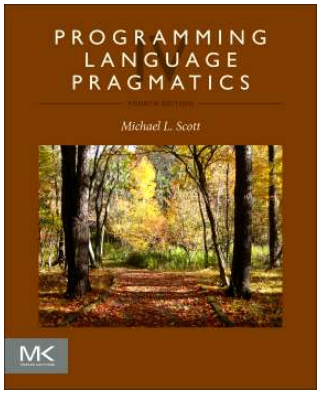
Code Improvement, Day 2: Global Optimization

17-363/17-663: Programming Language Pragmatics



Reading: PLP chapter 17

Prof. Jonathan Aldrich

Value Numbering & Aliasing

- Aliasing: x and y might refer to the same location
  - Distinguish x and y ***must*** alias from x and y ***may*** alias

- Concerns
  - If x may alias y:
    - store to x → remove knowledge of y
      → can't move below a load of y
  - If x must alias y:
    - store to x → update knowledge of y in table
    - load of x → can replace with existing load of y

Optimization Correctness

- Criterion: does the optimized program compute the same result as the original program, for all inputs?

- Soundness theorem: If p ~> p' then ∀input I, p(I) = p'(I)
  - You'll prove a version of this for a simple constant propagation analysis in Homework 8

Analysis Correctness

- Optimizations often rely on analysis information
  - Value numbering: correspondences between expressions and values in registers

- Rough guide to correctness: when you replace symbolic information in the analysis with concrete information from particular executions, does the result hold?
  - Becomes a lemma in the proof of soundness for the "client" optimization

Redundancy Elimination in Basic Blocks

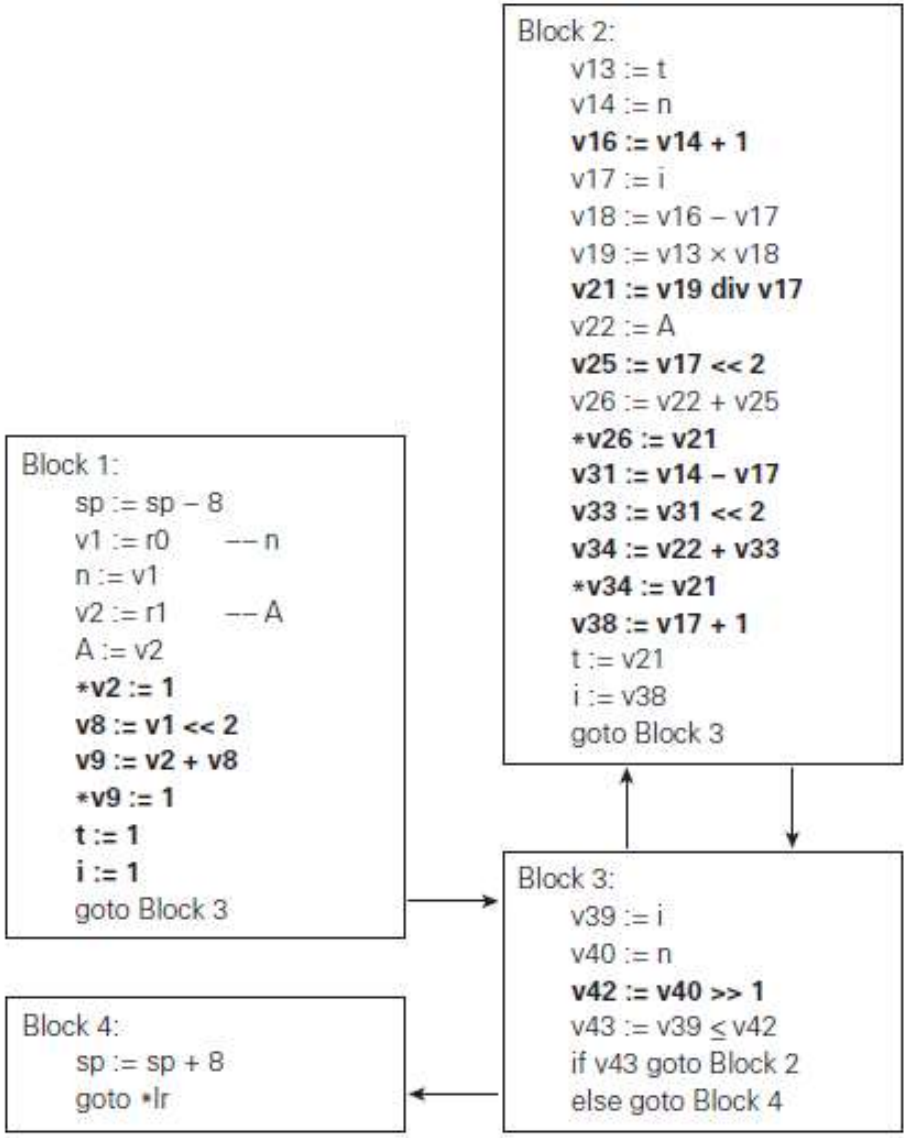


Figure 17.4 Control flow graph for the `combinations` subroutine after local redundancy elimination and strength reduction. Changes from Figure C-17.3 are shown in boldface type.

Global Redundancy and Data Flow Analysis

- We now concentrate on the elimination of redundant loads and computations across the boundaries between basic blocks
- We translate the code of our basic blocks into *static single assignment* (SSA) form, which will allow us to perform global value numbering
- Once value numbers have been assigned, we shall be able to perform
  - global common subexpression elimination
  - constant propagation
  - copy propagation

Global Redundancy and Data Flow Analysis

- In a compiler both the translation to SSA form and the various global optimizations would be driven by data flow analysis.
  - We detail the problems of identifying
    - common subexpressions
    - useless store instructions
  - We will also give data flow equations for the calculation of *reaching definitions*, used to move invariant computations out of loops
- Global redundancy elimination can be structured in such a way that it catches local redundancies as well, eliminating the need for a separate local pass

Value Numbering and SSA Form

- Value numbering, as introduced earlier, assigns a distinct virtual register name to every symbolically distinct value that is loaded or computed in a given body of code
  - It allows us to recognize when certain loads or computations are redundant.
- The first step in *global* value numbering is to distinguish among the values that may be written to a variable in different basic blocks
  - We accomplish this step using static single assignment (SSA) form

SSA Form

- For example, if the instruction `v2 := x` is guaranteed to read the value of `x` written by the instruction `x3 := v1`, then we replace `v2 := x` with `v2 := x3`
- If we cannot tell which version of x will be read, we use a hypothetical function ϕ to choose among the possible alternatives
  - we won't actually have to compute ϕ-functions at run time
    - the only purpose is to help us identify possible code improvements
  - we will drop them (and the subscripts) prior to target code generation

SSA Form - Example

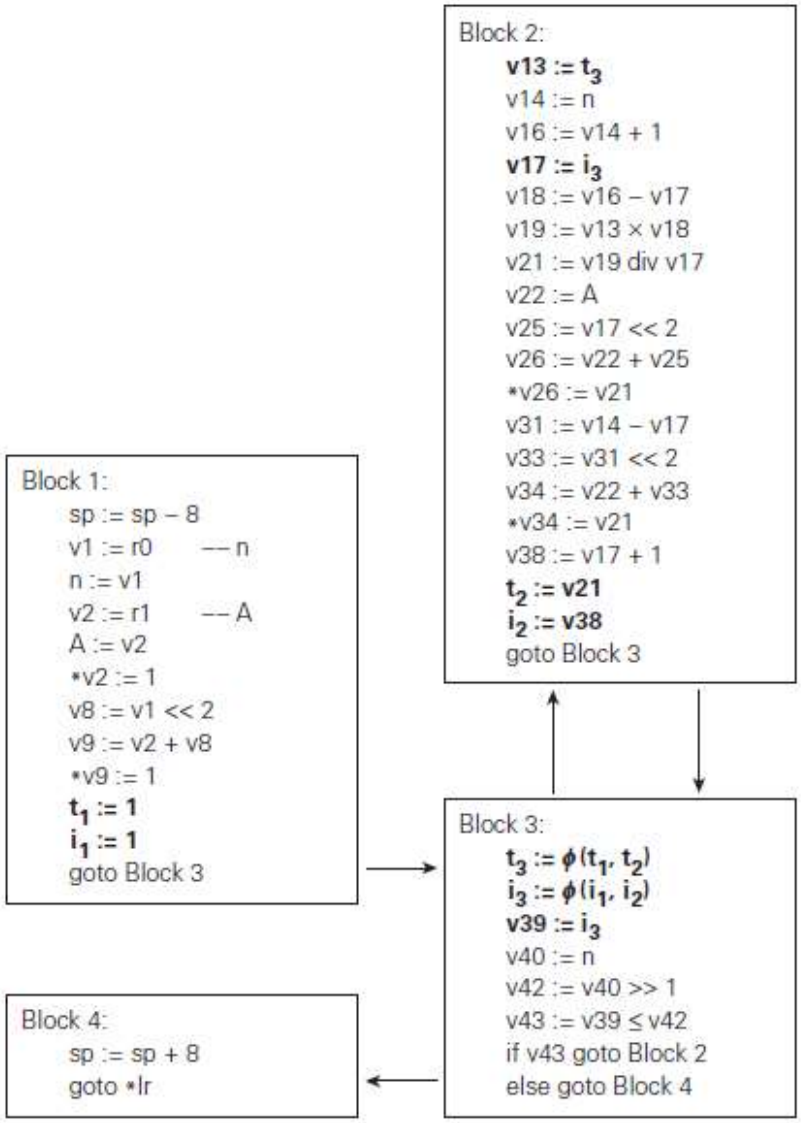


Figure 17.5 Control flow graph for the `combinations` subroutine, in static single assignment (SSA) form. Changes from Figure C-17.4 are shown in boldface type.

Local variables t and i are used across blocks.

We rename the variable at each load or assignment with an index.

At control flow merges, we add a ϕ-*function* merging them, and use the index for the merged variable.

Global Value Numbering

- With flow-dependent values determined by ϕ-functions, we are now in a position to perform global value numbering
  - As in local value numbering, the goal is to merge any virtual registers that are guaranteed to hold symbolically equivalent expressions
  - In the local case, we were able to perform a linear pass over the code
  - We kept a dictionary that mapped loaded and computed expressions to the names of virtual registers that contained them

Global Value Numbering

- This approach does not suffice in the global case, because the code may have cycles
  - The general solution can be formulated using data flow
  - It can also be obtained with a simpler algorithm that begins by unifying all expressions with the same top-level operator
    - In the end, repeatedly separates expressions whose operands are distinct
    - It is quite similar to the DFA minimization algorithm of Chapter 2
- We perform this analysis for our running example informally

Global Value Numbering

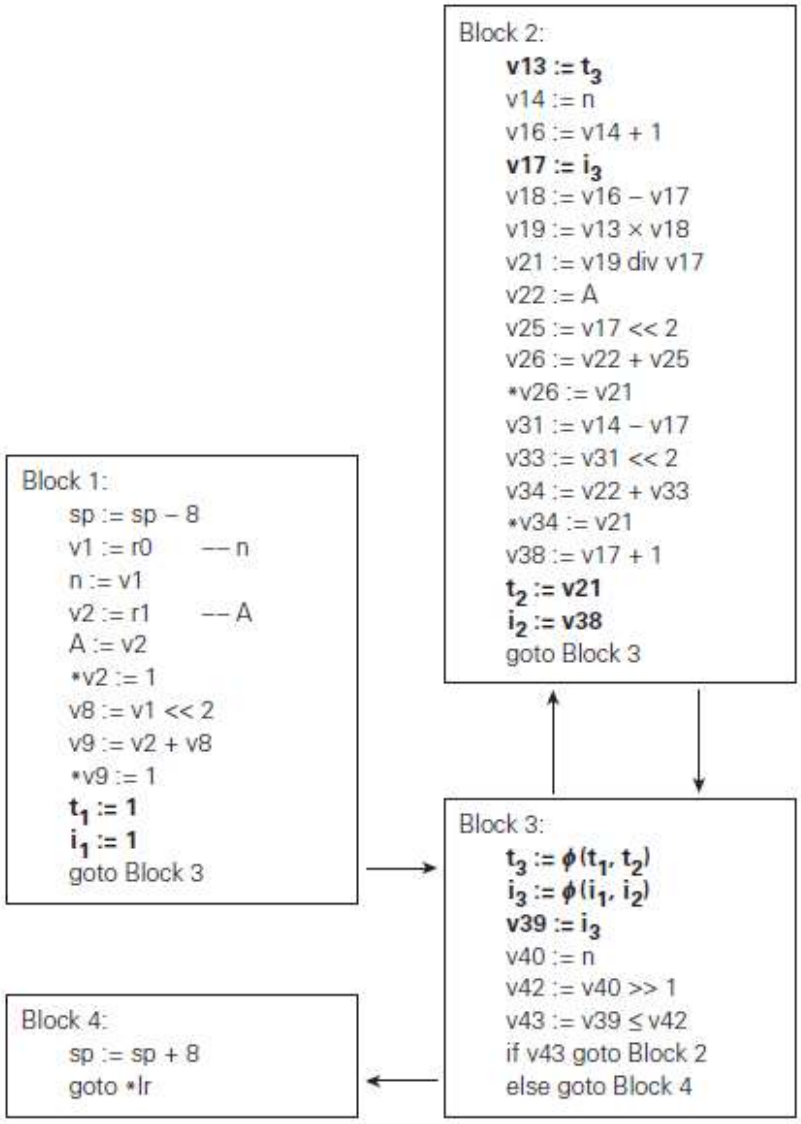
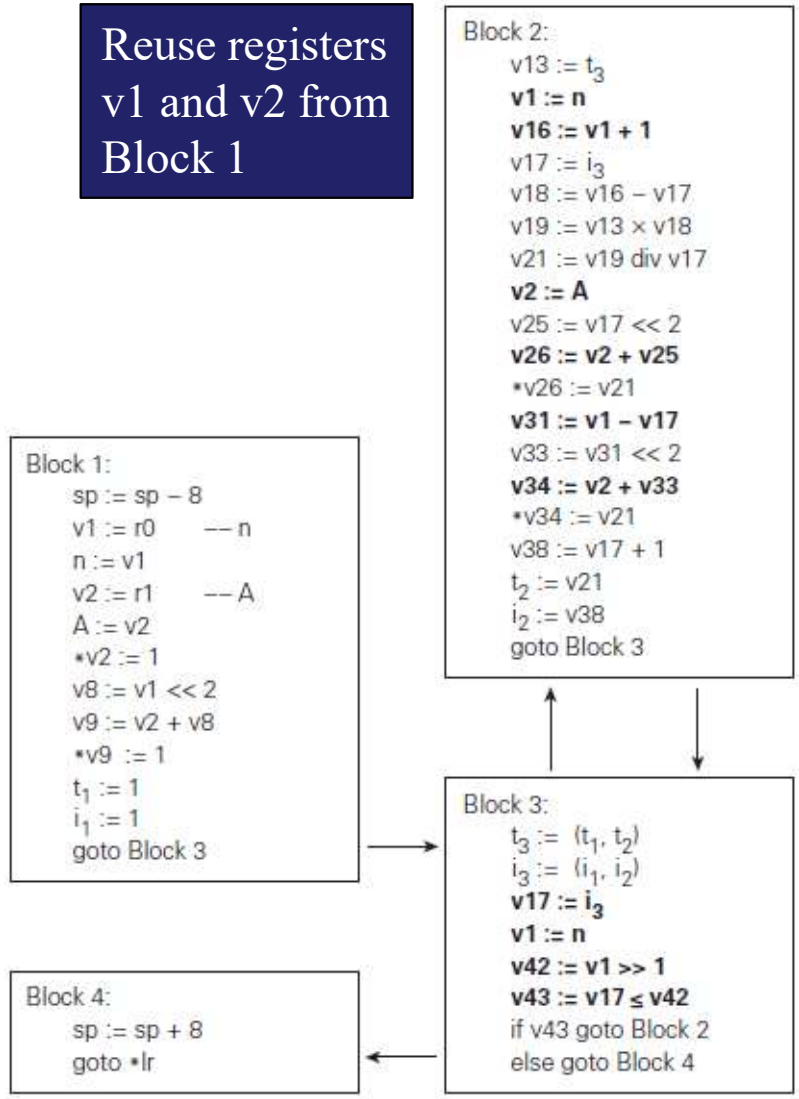


Figure 17.5 Control flow graph for the `combinations` subroutine, in static single assignment (SSA) form. Changes from Figure C-17.4 are shown in boldface type.

Reuse registers v1 and v2 from Block 1



Reuse v1 and v17

Figure 17.6 Control flow graph for the `combinations` subroutine after global value numbering. Changes from Figure C-17.5 are shown in boldface type.

Global Redundancy and Data Flow Analysis

- 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

Global Redundancy and Data Flow Analysis

- 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 Redundancy and Data Flow Analysis

- 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

Global Redundancy and Data Flow Analysis

- The data flow equations for available expression analysis are:

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

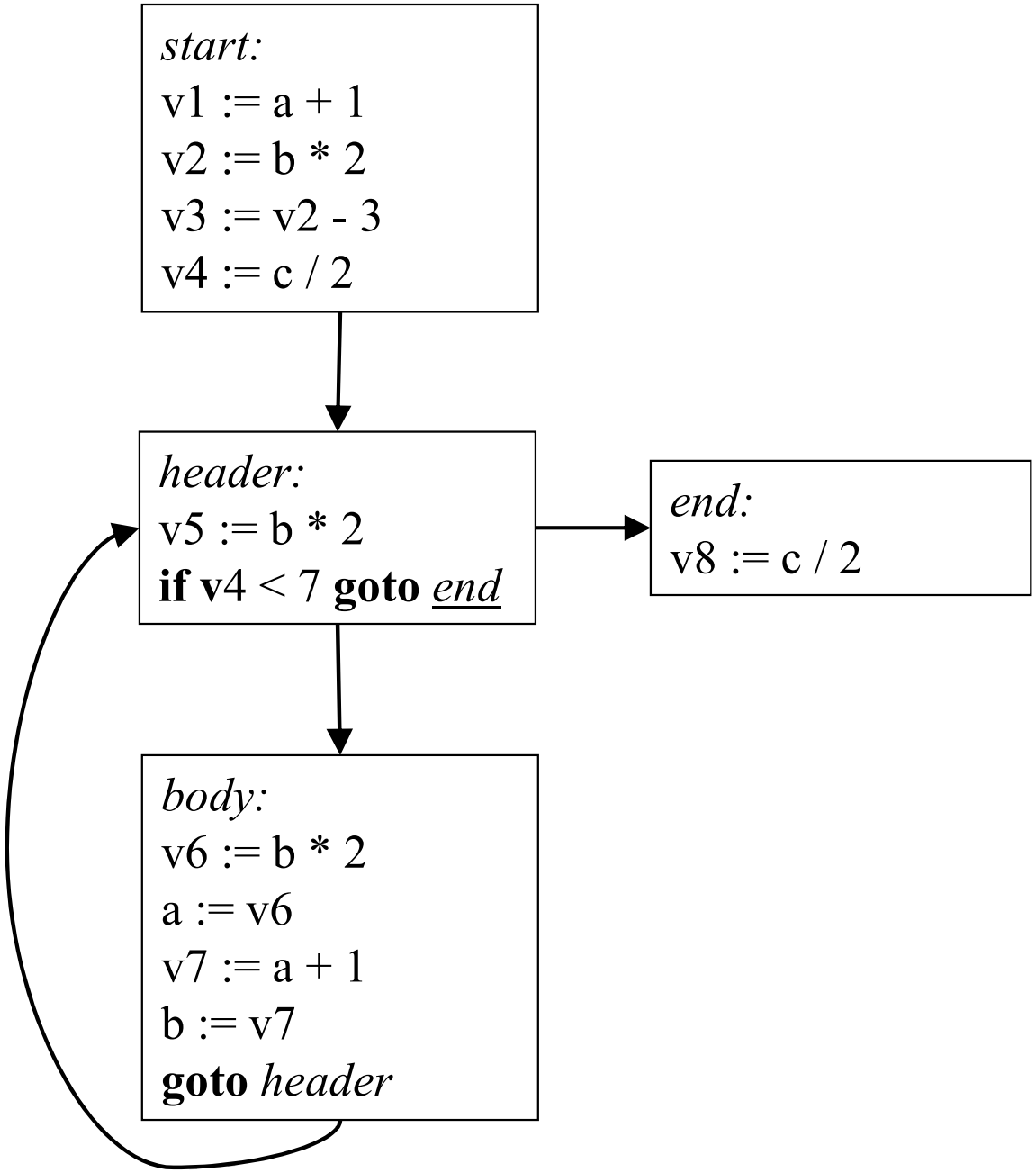
$$In_B = \bigcap_{\text{predecessors } A \text{ of } B} Out_A$$

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

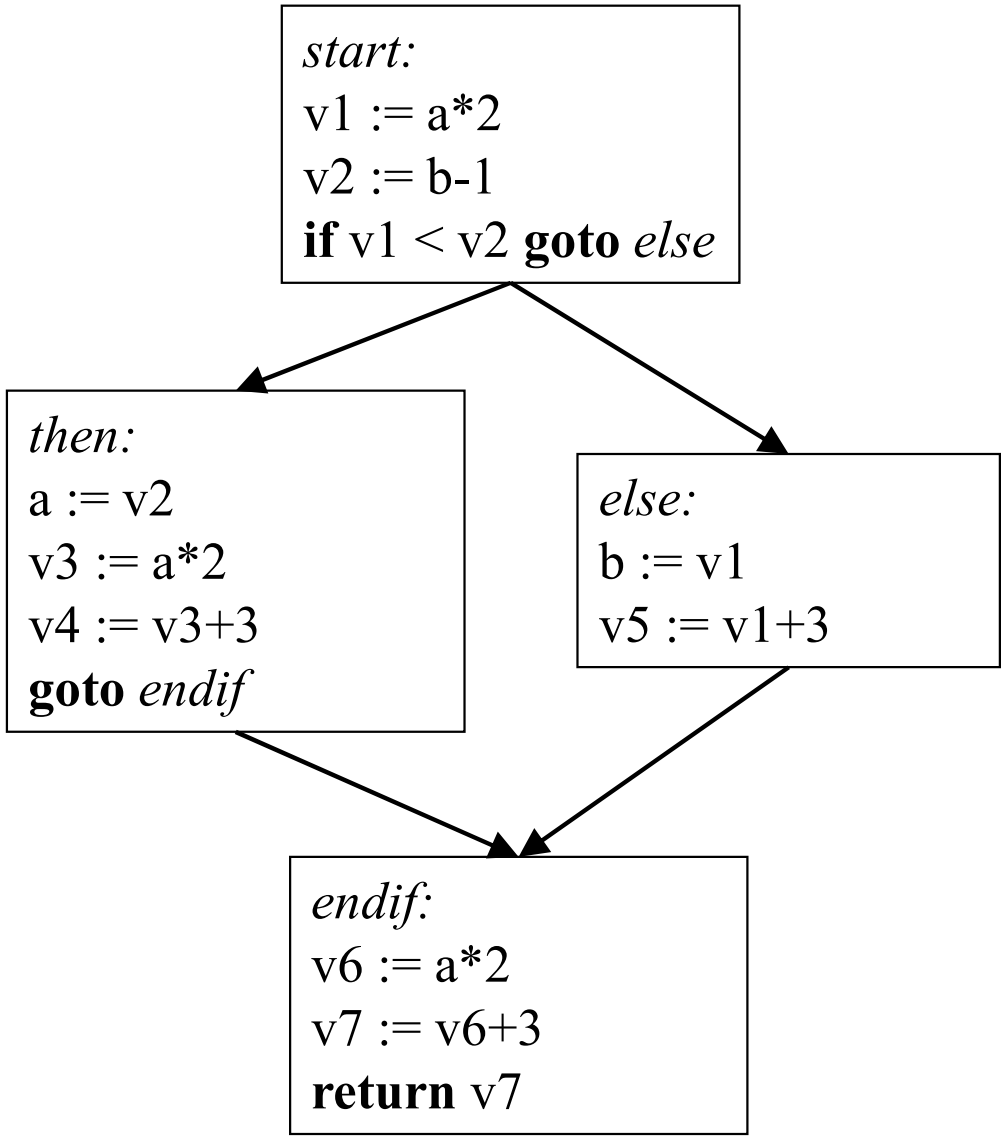
Global Redundancy and Data Flow Analysis

- 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, much as we computed first and follow sets in Chapter 2
- 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

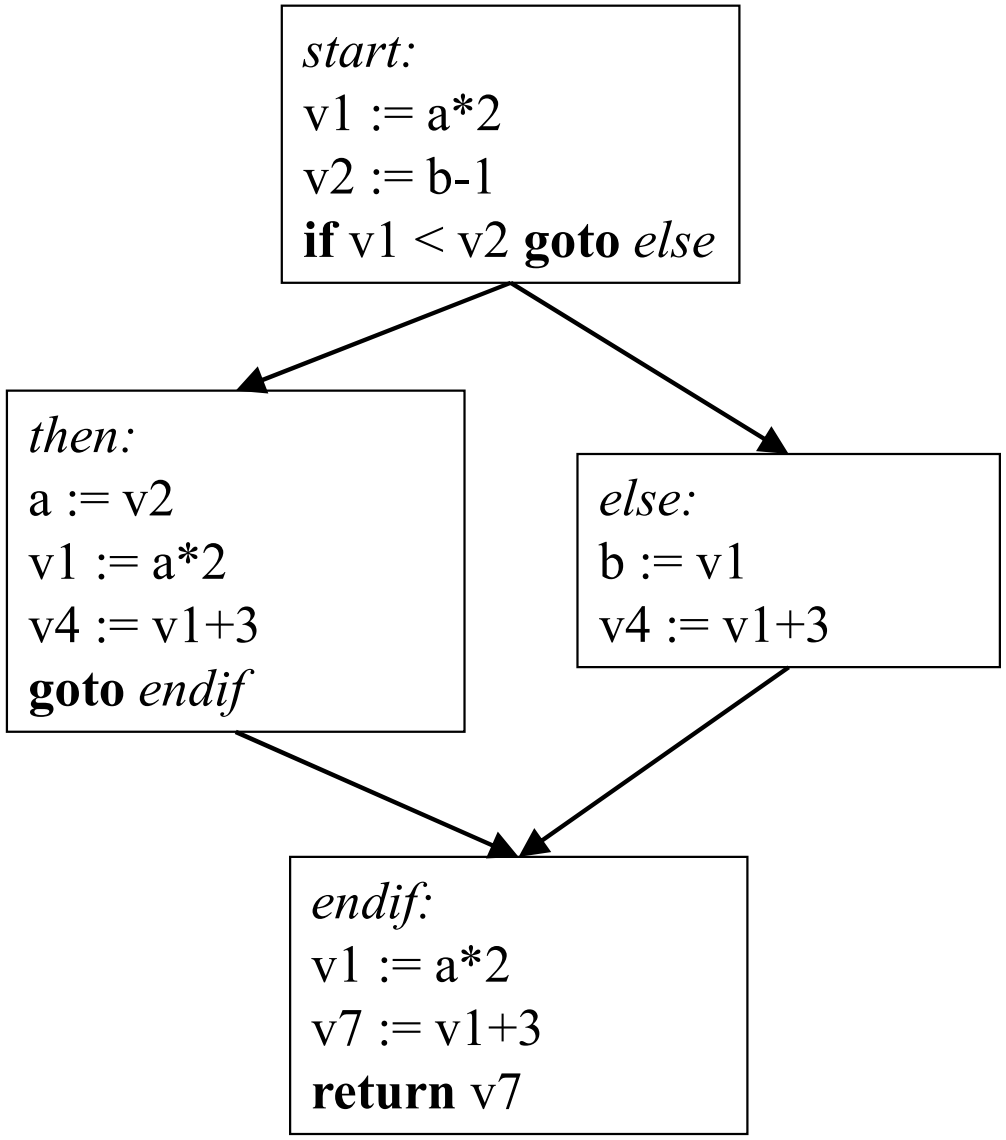
Example of Available Expressions Analysis



Exercise: Apply global value numbering and available expressions to this program



Exercise: Apply global value numbering and available expressions to this program



after renamings

Global Redundancy and Data Flow Analysis

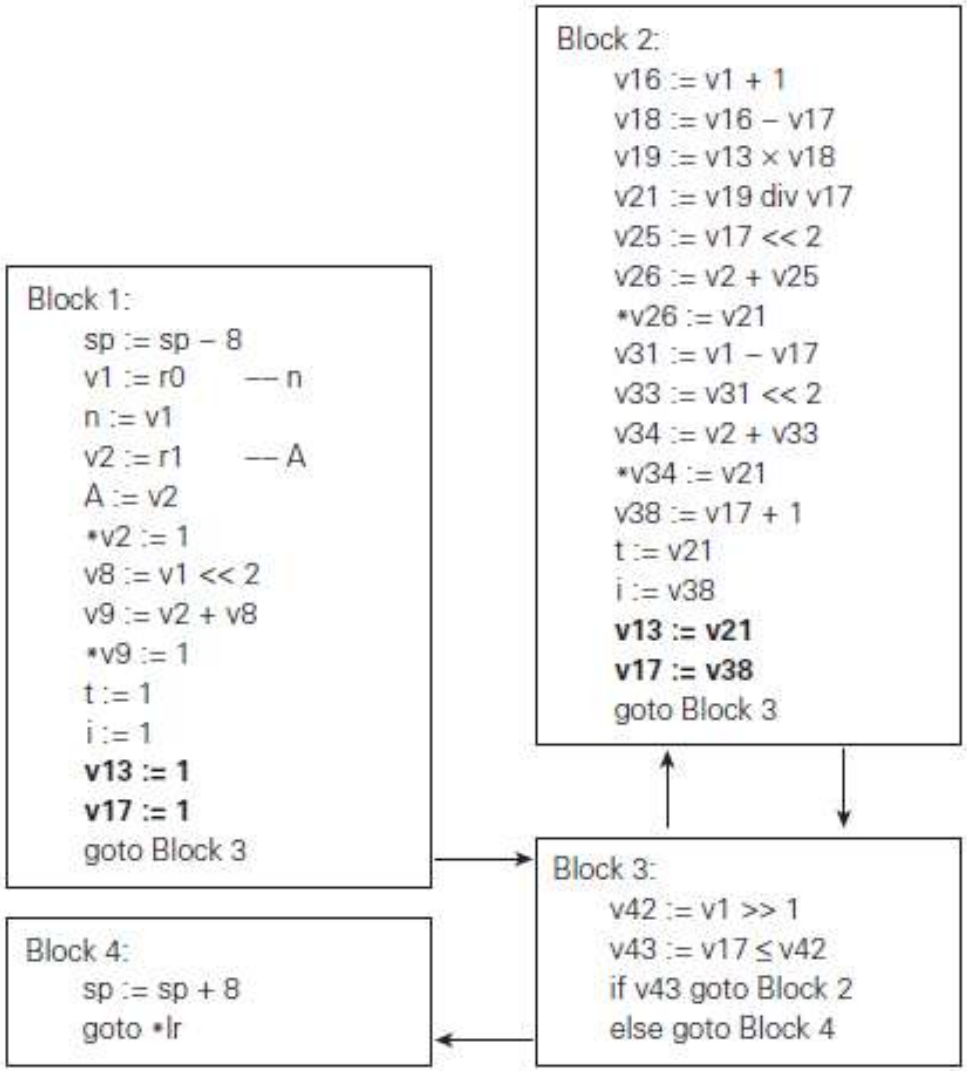


Figure 17.7 Control flow graph for the `combinations` subroutine after performing global common subexpression elimination. Note the absence of the many load instructions of Figure C-17.6. Compensating register–register moves are shown in boldface type.

ELSEVIER

Global Redundancy and Data Flow Analysis

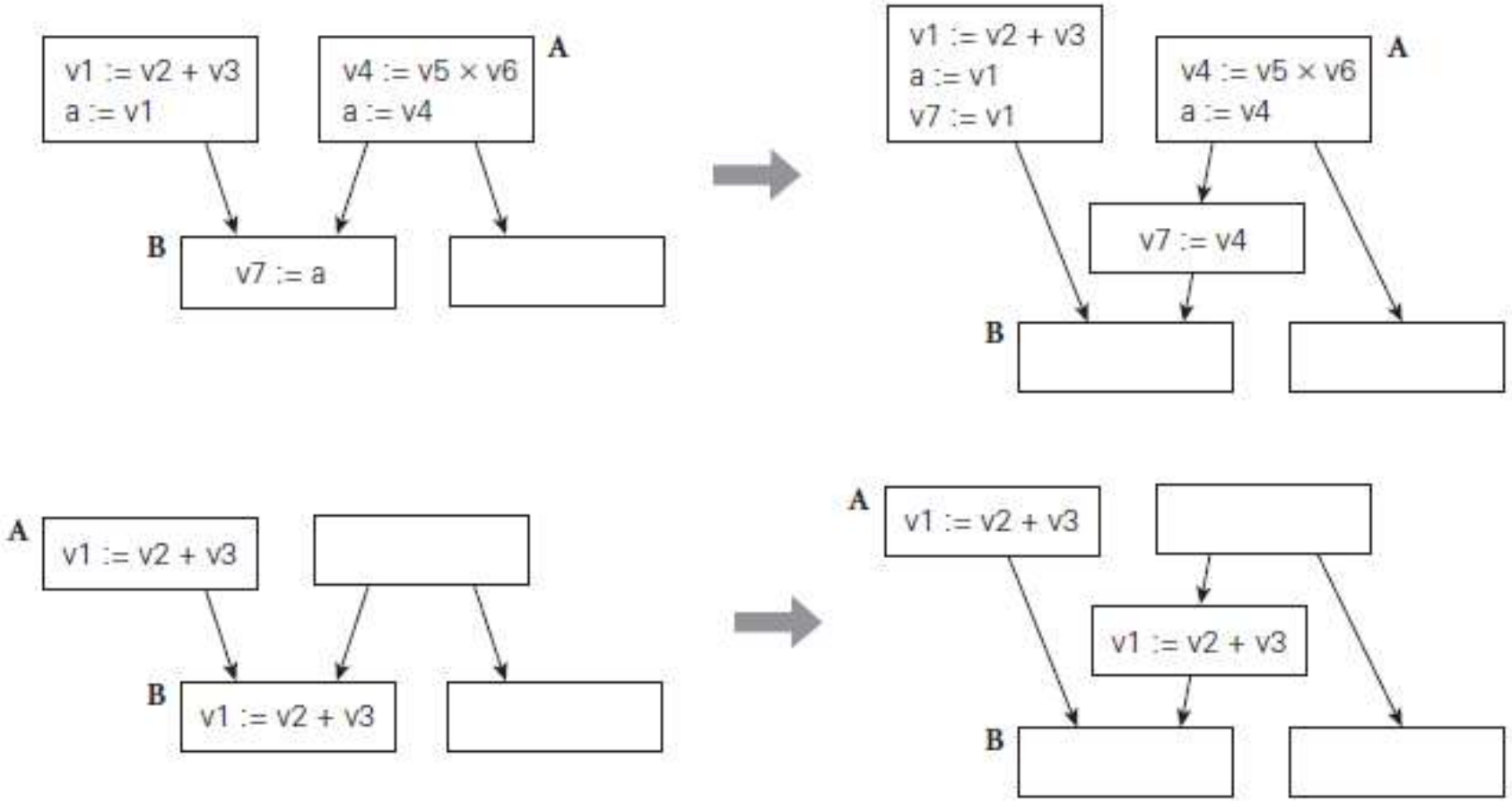
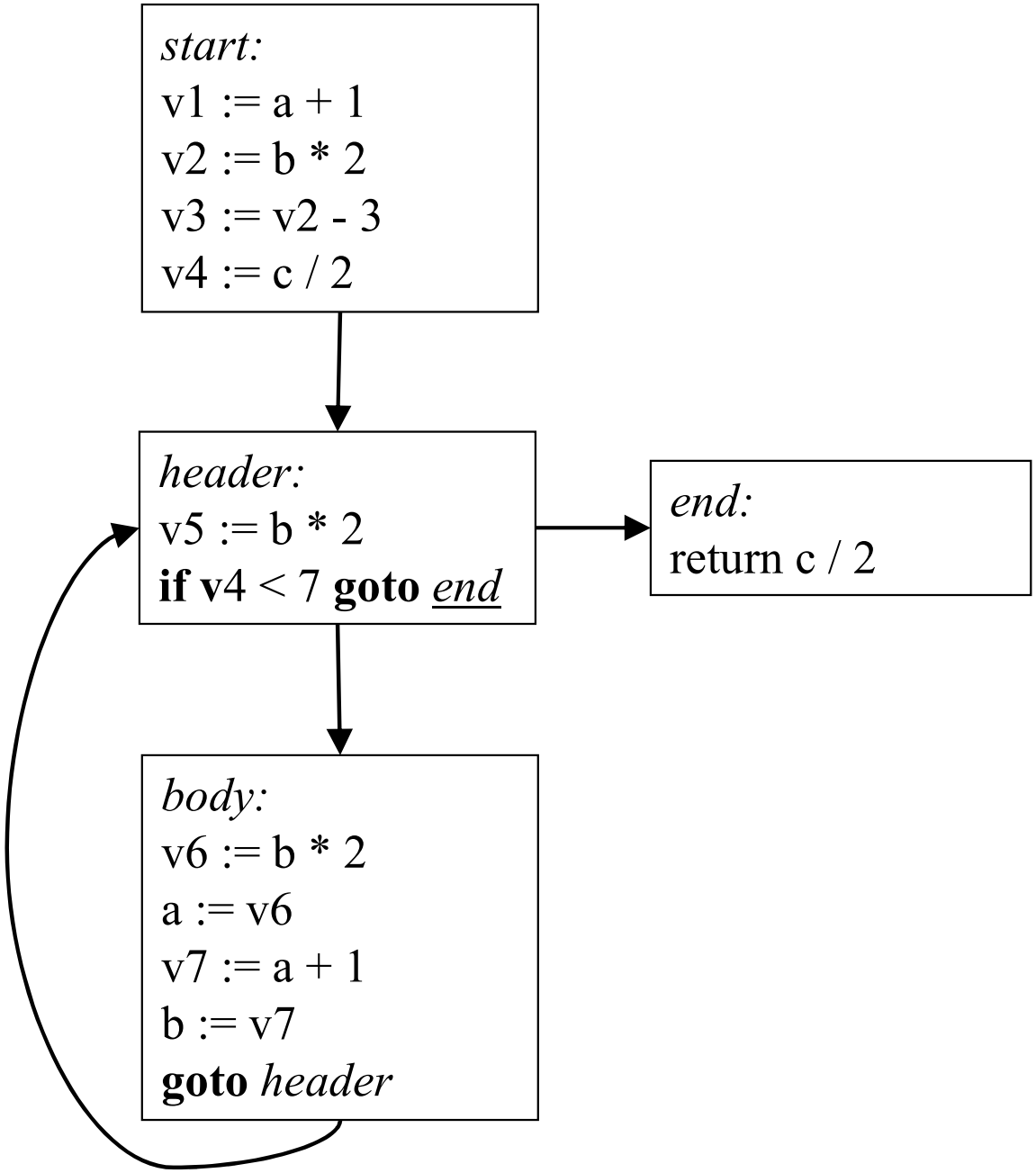


Figure 17.8 Splitting an edge of a control flow graph to eliminate a redundant load (*top*) or a partially redundant computation (*bottom*).

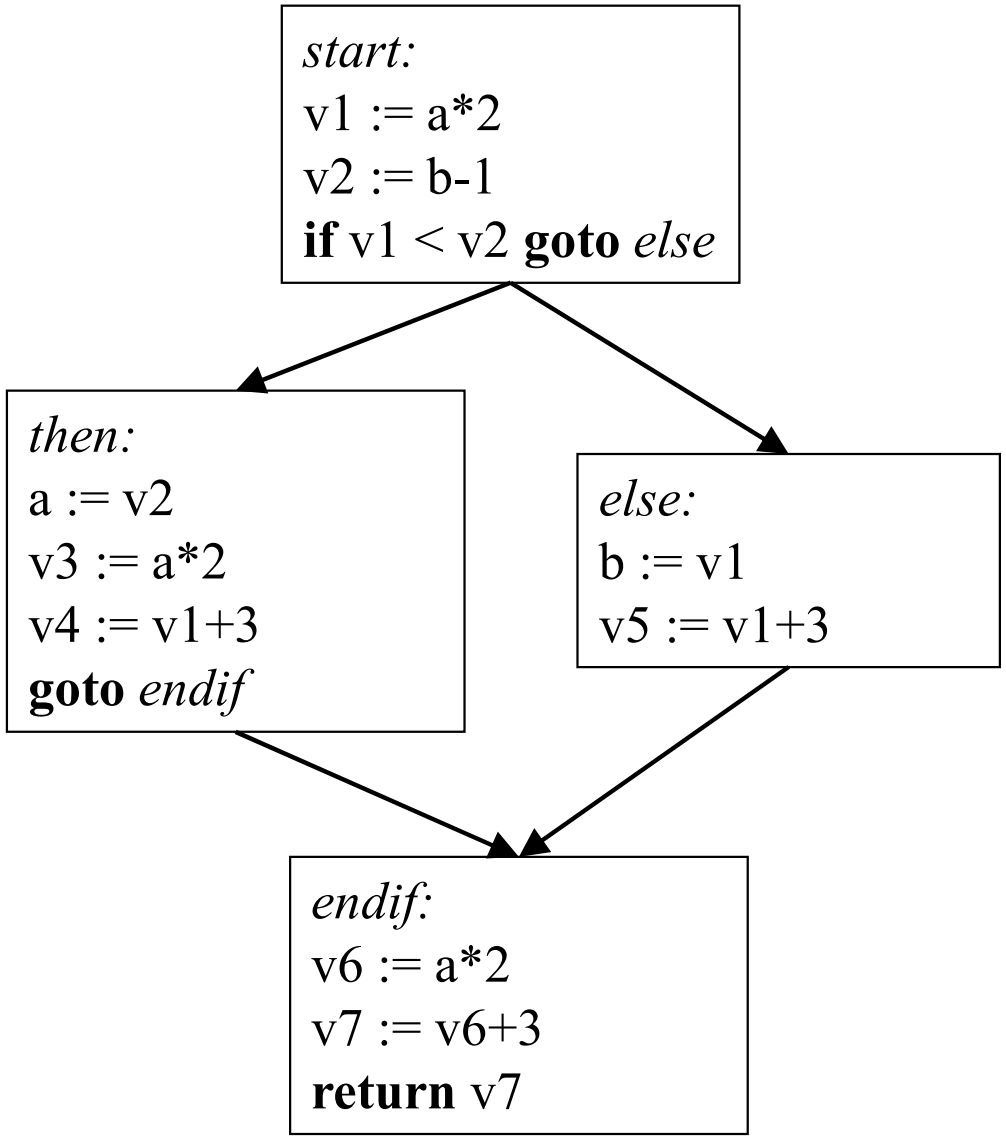
Global Redundancy and Data Flow Analysis

- 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

Running live variable analysis and dead code elimination



Exercise: Apply live variable analysis and dead code elimination to this program



Global Redundancy and Data Flow Analysis

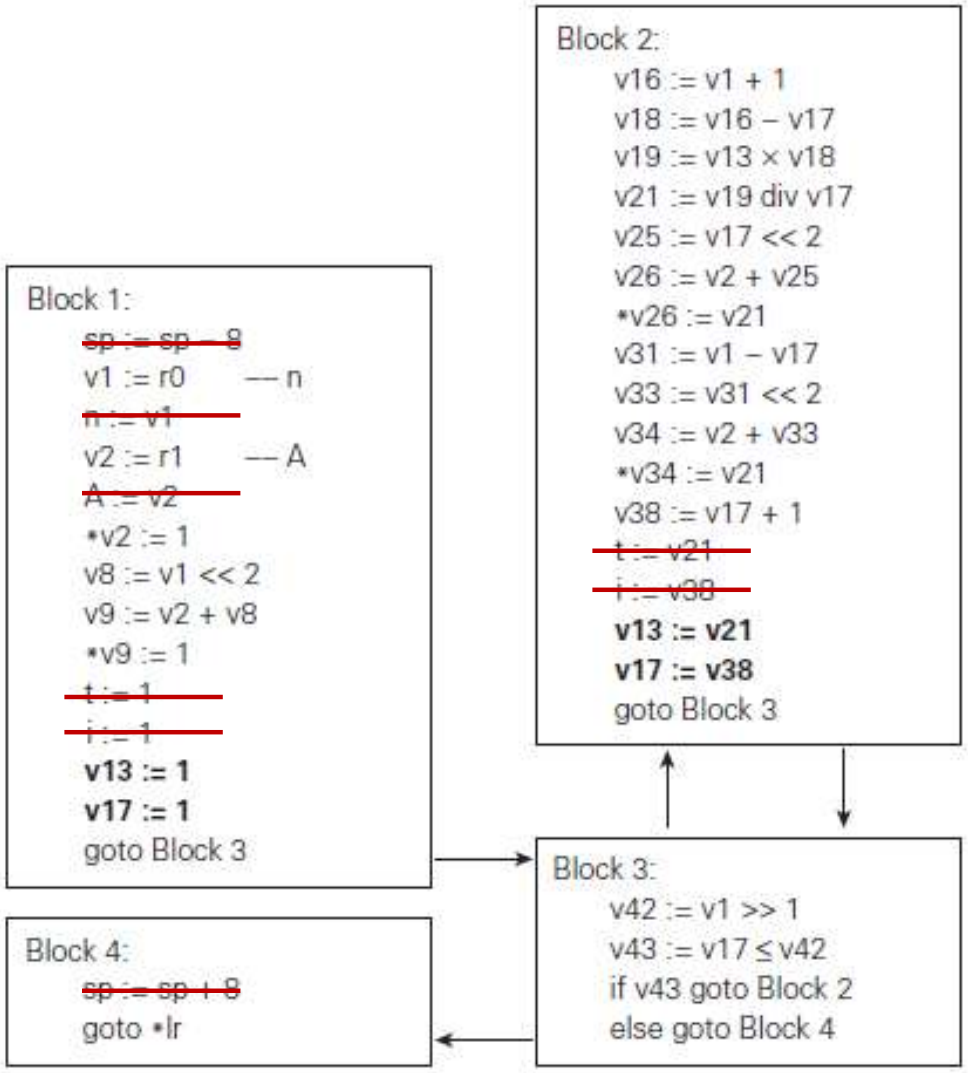


Figure 17.7 Control flow graph for the `combinations` subroutine after common subexpression elimination. Note the absence of the many load i ure C-17.6. Compensating register–register moves are shown in boldface type

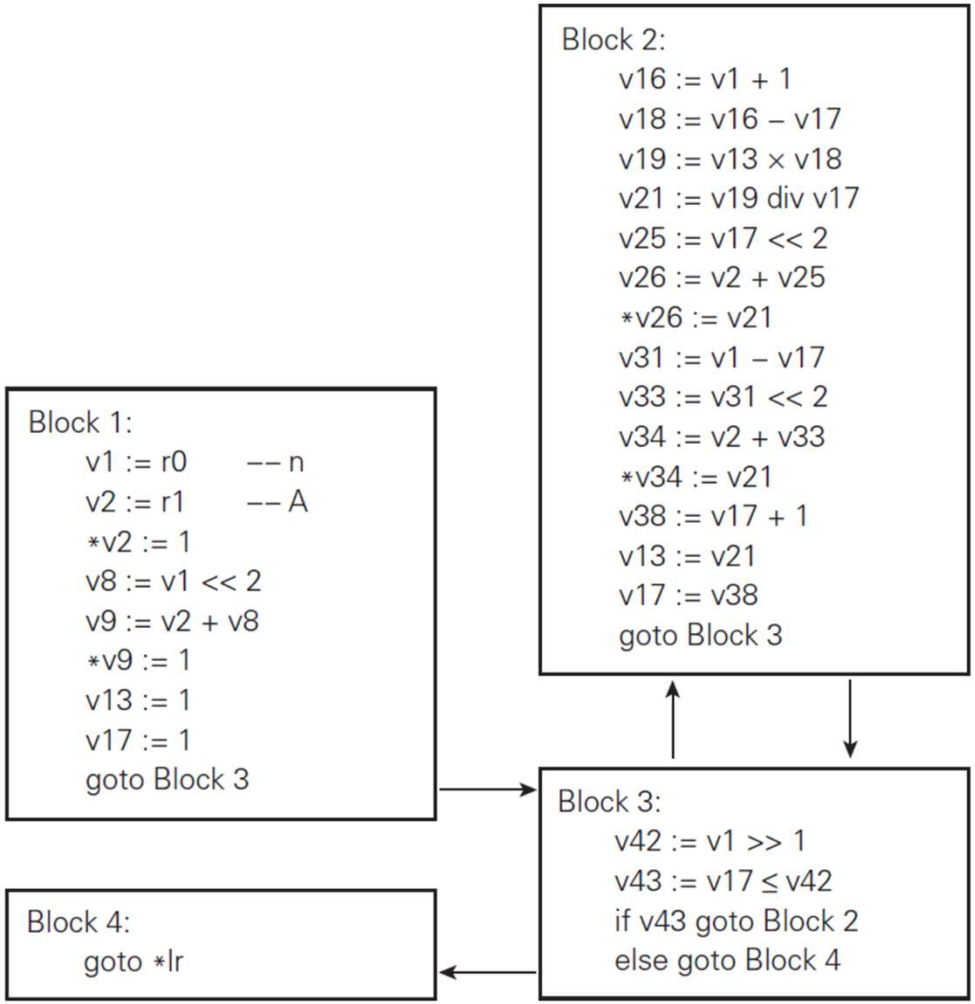


Figure 17.9 Control flow graph for the `combinations` subroutine after perform variable analysis. Starting with Figure C-17.7, the compiler has eliminated all stores to and i. It has also dropped the changes to the stack pointer that used to appear in the sub prologue and epilogue: we don't need space for local variables anymore.