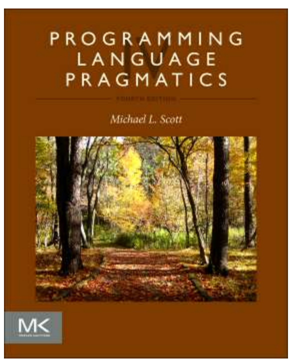
Code Improvement

17-363/17-663: Programming Language Pragmatics



Reading: PLP chapter 17

Prof. Jonathan Aldrich

Introduction

- We discussed target code generation
  - Typically produces correct but highly suboptimal code
    - redundant computations
    - inefficient use of the registers, multiple functional units, and cache
- This chapter takes a look at *code improvement*: the phases of compilation devoted to generating *good* code
  - we interpret "good" to mean *fast*
  - occasionally we also consider program transformations to decrease memory requirements
  - we sometimes say "optimization," but the code produced is rarely truly optimal

Introduction

- In a very simple compiler, we can use a *peephole optimizer* to peruse already-generated target code for obviously suboptimal sequences of adjacent instructions
- At a slightly higher level, we can generate near-optimal code for *basic blocks*
  - a basic block is a maximal-length sequence of instructions that will always execute in its entirety (assuming it executes at all)
  - in the absence of hardware exceptions, control never enters a basic block except at the beginning, and never exits except at the end

Introduction

- Code improvement at the level of basic blocks is known as *local* optimization
  - elimination of redundant operations (unnecessary loads, common sub-expression calculations)
  - effective instruction scheduling and register allocation
- At higher levels of aggressiveness, compilers employ techniques that analyze entire subroutines for further speed improvements
- These techniques are known as *global* optimization
  - multi-basic-block versions of redundancy elimination
  - instruction scheduling, and register allocation
  - code modifications designed to improve the performance of loops

ELSEVIER

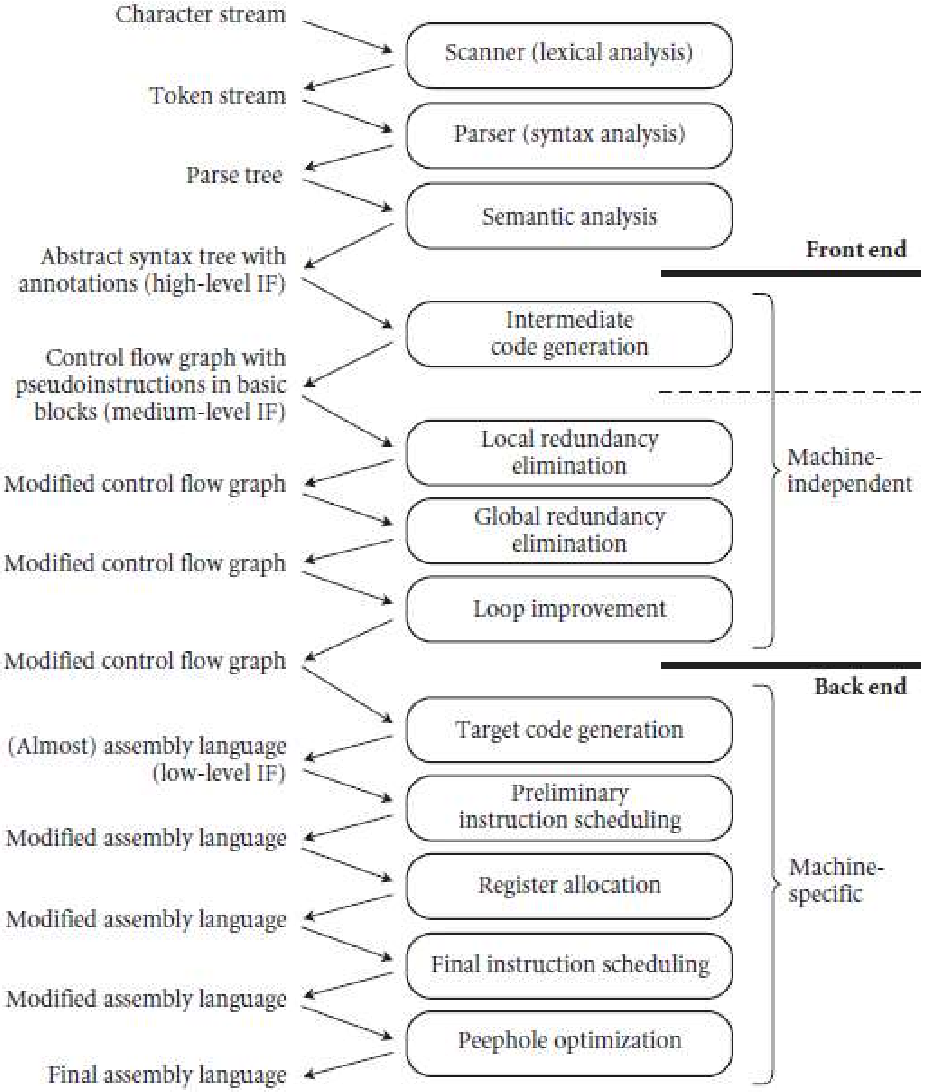
Introduction

- Both global redundancy elimination and loop improvement typically employ a *control flow graph* representation of the program
  - Use a family of algorithms known as *data flow analysis* (flow of information between basic blocks)
- Recent compilers perform various forms of *interprocedural* code improvement
- Interprocedural improvement is difficult
  - subroutines may be called from many different places
    - hard to identify available registers, common subexpressions, etc.
  - subroutines are separately compiled

Phases of Code Improvement

- We will concentrate in our discussion on the forms of code improvement that tend to achieve the largest increases in execution speed, and are most widely used
  - Compiler phases to implement these improvements is shown in Figure 17.1

Phases of Code Improvement



Phases of Code Improvement

- The *machine-independent* part of the back end begins with intermediate code generation
  - identifies fragments of the syntax tree that correspond to basic blocks
  - creates a control flow graph in which each node contains a sequence of three-address instructions for an idealized machine (unlimited supply of *virtual registers*)
- The *machine-specific* part of the back end begins with target code generation
  - strings the basic blocks together into a linear program
    - translates each block into the instruction set of the target machine and generating branch instructions that correspond to the arcs of the control flow graph

Phases of Code Improvement

- Machine-independent code improvement has three separate phases
  1. Local redundancy elimination: identifies and eliminates redundant loads, stores, and computations within each basic block
  2. Global redundancy elimination: identifies similar redundancies across the boundaries between basic blocks (but within the bounds of a single subroutine)
  3. Loop improvement: effects several improvements specific to loops
     - these are particularly important, since most programs spend most of their time in loops.
     - Global redundancy elimination and loop improvement may actually be subdivided into several separate phases

Phases of Code Improvement

- Machine-specific code improvement has four separate phases
  - Preliminary and final instruction scheduling are essentially identical (Phases 1 & 3)
  - Register allocation (Phase 2) and instruction scheduling tend to interfere with one another
    - the instruction schedules minimize pipeline stalls which tend to increase the demand for architectural registers (*register pressure*)
    - we schedule instructions first, then allocate architectural registers, then schedule instructions again
      - If it turns out that there aren't enough architectural registers, the register allocator will generate additional load and store instructions to *spill* registers temporarily to memory
      - the second round of instruction scheduling attempts to fill any delays induced by the extra loads

ELSEVIER

Peephole Optimization

- A relatively simple way to significantly improve the quality of naive code is to run a *peephole optimizer* over the target code
  - works by sliding a several instruction window (a peephole) over the target code, looking for suboptimal patterns of instructions
  - the patterns to look for are heuristic
    - patterns to match common suboptimal idioms produced by a particular front end
    - patterns to exploit special instructions available on a given machine
- A few examples are presented in what follows

Peephole Optimization

- ***Elimination of redundant loads and stores***
  - The peephole optimizer can often recognize that the value produced by a load instruction is already available in a register

```
r2 := r1 + 5
i := r2
r3 := i
r3 := r3 × 3
```

becomes

```
r2 := r1 + 5
i := r2
r3 := r2 × 3
```

Peephole Optimization

- *Constant folding*
- A naive code generator may produce code that performs calculations at run time that could actually be performed at compile time
  - A peephole optimizer can often recognize such code

```
r2 := 3 × 2
```

becomes

```
r2 := 6
```

Peephole Optimization

- ***Constant propagation***
  - Sometimes we can tell that a variable will have a constant value at a particular point in a program
  - We can then replace occurrences of the variable with occurrences of the constant

```
r2 := 4
r3 := r1 + r2
r2 := . . .
```

becomes

```
r2 := 4
r3 := r1 + 4
r2 := . . .
```

and then

```
r3 := r1 + 4
r2 := . . .
```

ELSEVIER

Peephole Optimization

- ***Common subexpression elimination***
  - When the same calculation occurs twice within the peephole of the optimizer, we can often eliminate the second calculation:

```
r2 := r1 × 5
r2 := r2 + r3
r3 := r1 × 5
```

becomes

```
r4 := r1 × 5
r2 := r4 + r3
r3 := r4
```

  - Often, as shown here, an extra register will be needed to hold the common value

Peephole Optimization

- It is natural to think of common subexpressions as something that could be eliminated at the source code level, and programmers are sometimes tempted to do so
- The following, for example,

```
x = a + b + c;
y = a + b + d;
```

could be replaced with

```
t = a + b;
x = t + c;
y = t + d;
```

Peephole Optimization

- ***Copy propagation***
  - Even when we cannot tell that the contents of register *b* will be constant, we may sometimes be able to tell that register *b* will contain the same value as register *a*
    - replace uses of *b* with uses of *a*, so long as neither *a* nor *b* is modified

```
r2 := r1
r3 := r1 + r2
r2 := 5
```

becomes

```
r2 := r1
r3 := r1 + r1
r2 := 5
```

and then

```
r3 := r1 + r1
r2 := 5
```

Peephole Optimization

- ***Strength reduction***
  - Numeric identities can sometimes be used to replace a comparatively expensive instruction with a cheaper one
    - In particular, multiplication or division by powers of two can be replaced with adds or shifts:

```
r1 := r2 × 2
```

becomes

```
r1 := r2 + r2 or r1 := r2 << 1
```

```
r1 := r2 / 2
```

becomes

```
r1 := r2 >> 1
```

Peephole Optimization

- ***Elimination of useless instructions***
  - Instructions like the following can be dropped entirely:

```
r1 := r1 + 0
r1 := r1 × 1
```

- ***Filling of load and branch delays***
  - Several examples of delay-filling transformations are presented in Chapter 5 of the textbook
- ***Exploitation of the instruction set***
  - Particularly on CISC machines, sequences of simple instructions can often be replaced by a smaller number of more complex instructions

Redundancy Elimination in Basic Blocks

- Let's look at improving intermediate code generated from this C program:

```
combinations(int n, int *A) {
    int i, t;
    A[0] = 1;
    A[n] = 1;
    t = 1;
    for (i = 1; i <= n/2; i++) {
        t = (t * (n+1-i)) / i;
        A[i] = t;
        A[n-i] = t;
    }
}
```

Redundancy Elimination in Basic Blocks

- We employ a medium level intermediate form (IF) for control flow
  - Every calculated value is placed in a separate register
  - To emphasize virtual registers (of which there is an unlimited supply), we name them v1, v2, . . .
  - We use r1, r2, . . . to represent architectural registers in Section 17.8.

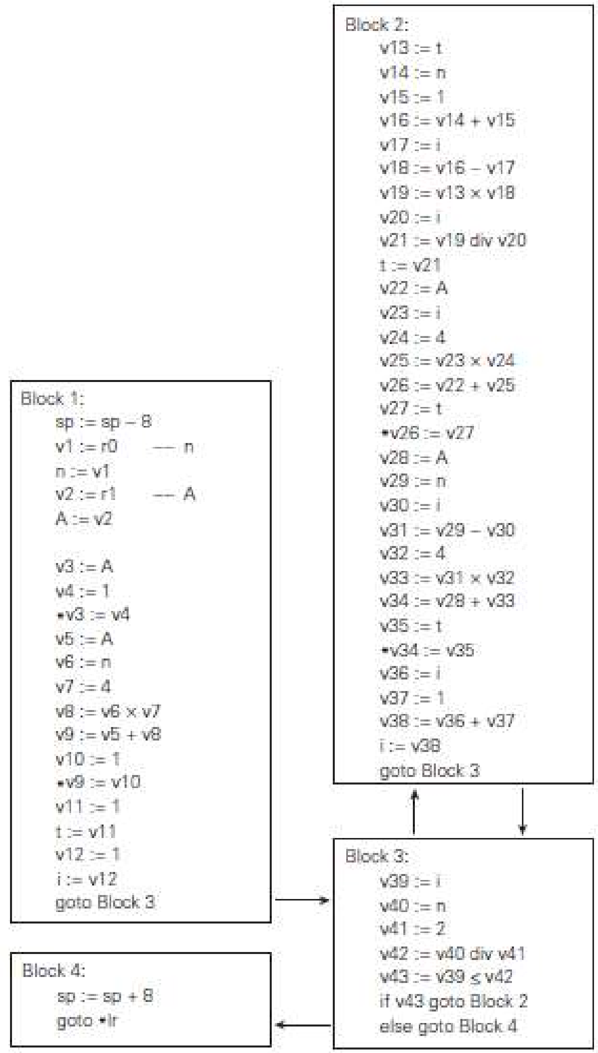


Figure 17.3 Naive control flow graph for the combinations subroutine. Note that reference parameter A contains the address of the array into which to write results; hence we write v3 := A instead of v3 := &A.

Redundancy Elimination in Basic Blocks

- To improve the code within basic blocks, we need to
  - minimize loads and stores
  - identify redundant calculations
- There are two techniques usually employed
  1. translate the syntax tree for a basic block into an *expression DAG* (directed acyclic graph) in which redundant loads and computations are merged into individual nodes with multiple parents
  2. similar functionality can also be obtained without an explicitly graphical program representation, through a technique known as local *value numbering*
- We describe the last technique below

Redundancy Elimination in Basic Blocks

- Value numbering assigns the same name (a "number") to any two or more symbolically equivalent computations ("values"), so that redundant instances will be recognizable by their common name
- Our names are virtual registers, which we merge whenever they are guaranteed to hold a common value
- While performing local value numbering, we will also implement
  - local constant folding
  - constant propagation, copy propagation
  - common subexpression elimination
  - strength reduction
  - useless instruction elimination

Redundancy Elimination in Basic Blocks

- Let’s do value numbering for a simpler example:

v1 := x

v2 := 1

v3 := v1 + v2

y := v3

v4 := x

v5 := 1

v6 := v4 + v5

x := v6

v7 := x

v8 := 3

v9 := 1

v10 := v8 + v9

v11 := v7 * v10

v12 := v11 * v9

What the source might look like:

y := x + 1;

x := x + 1;

return x * (3+1) * 1;

Your Turn: Value Numbering

- Perform value numbering optimization on the following:

v1 := x

v2 := 3

v3 := v1 + v2

v4 := 1

v5 := x

v6 := 2

v7 := v4 + v6

v8 := v5 + v7

v9 := v8 - v3

Redundancy Elimination in Basic Blocks

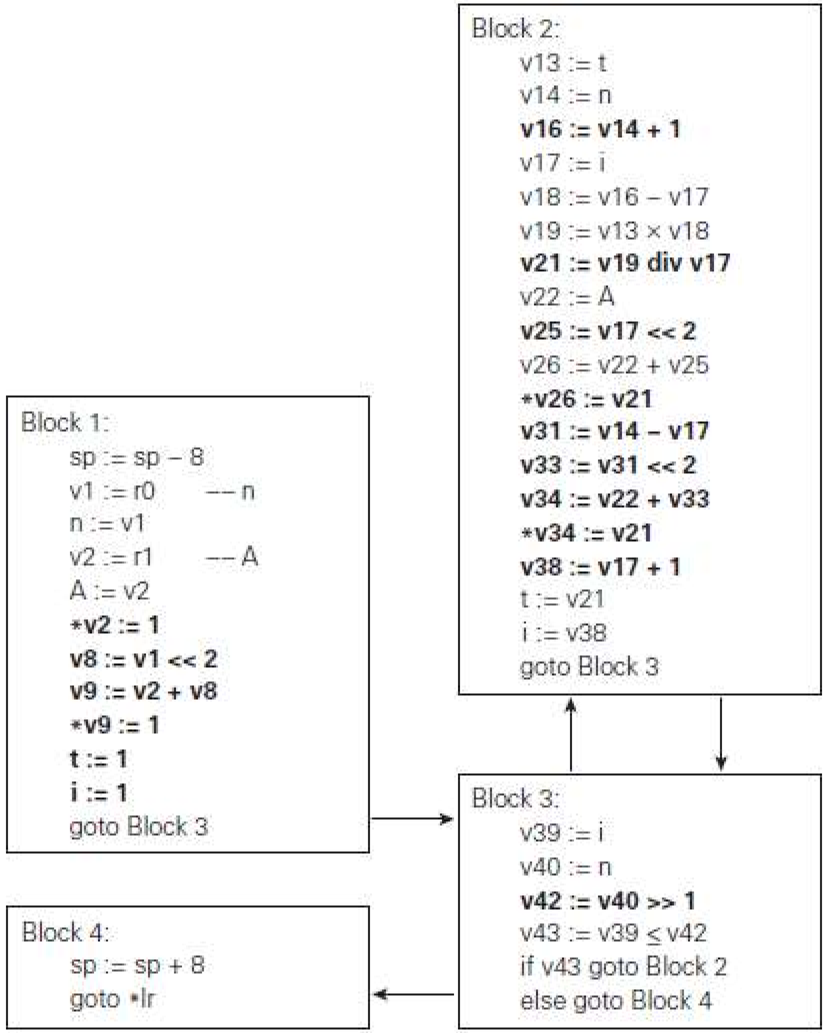


Figure 17.4 Control flow graph for the `combinations` subroutine after local redundancy elimination and strength reduction. Changes from Figure C-17.3 are shown in boldface type.

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

Global Redundancy and Data Flow Analysis

- 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

Global Redundancy and Data Flow Analysis

- 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

Global Redundancy and Data Flow Analysis

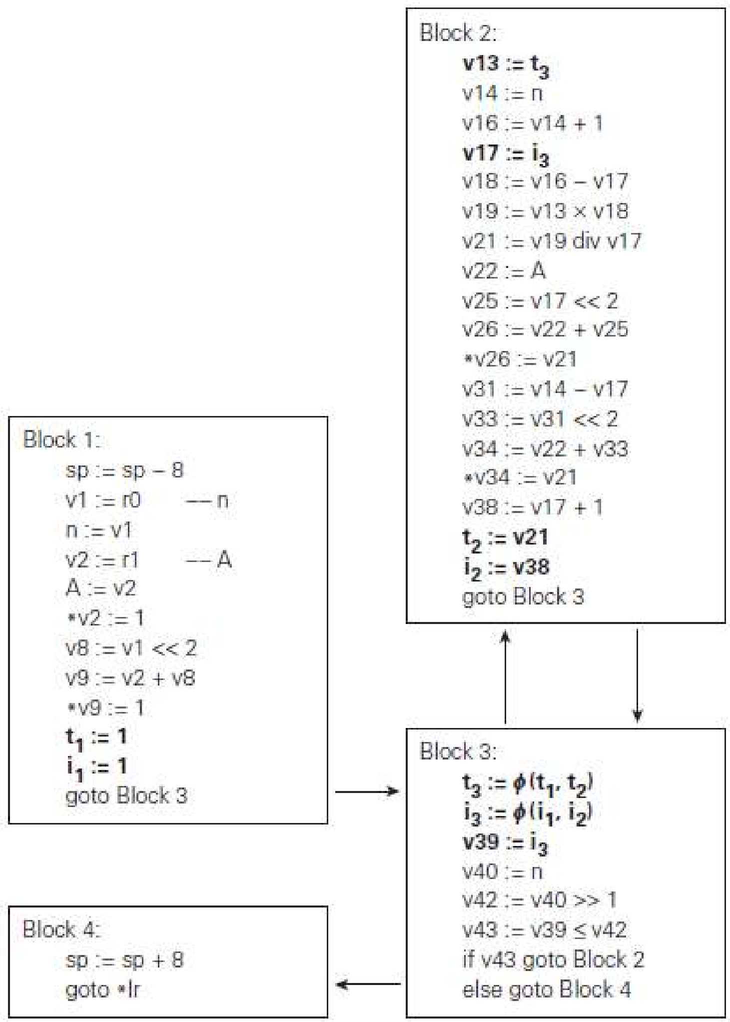


Figure 17.5 Control flow graph for the `combinations` subroutine, in static single assignment (SSA) form. Changes from Figure C-17.4 are shown in boldface type.

Global Redundancy and Data Flow Analysis

- 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 Redundancy and Data Flow Analysis

- 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 Redundancy and Data Flow Analysis

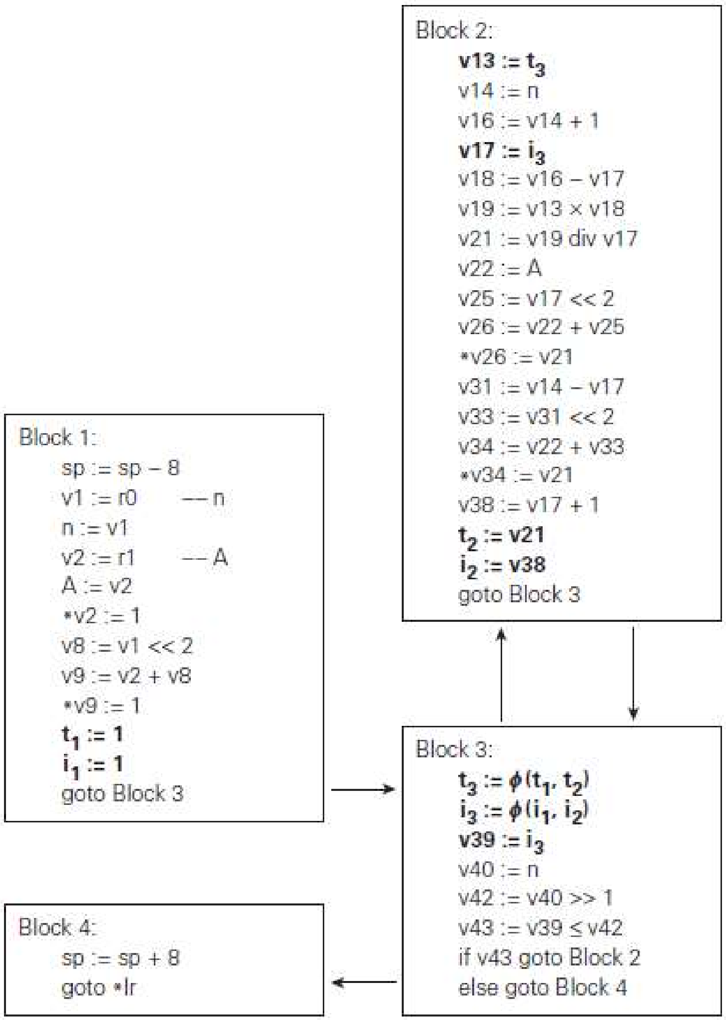
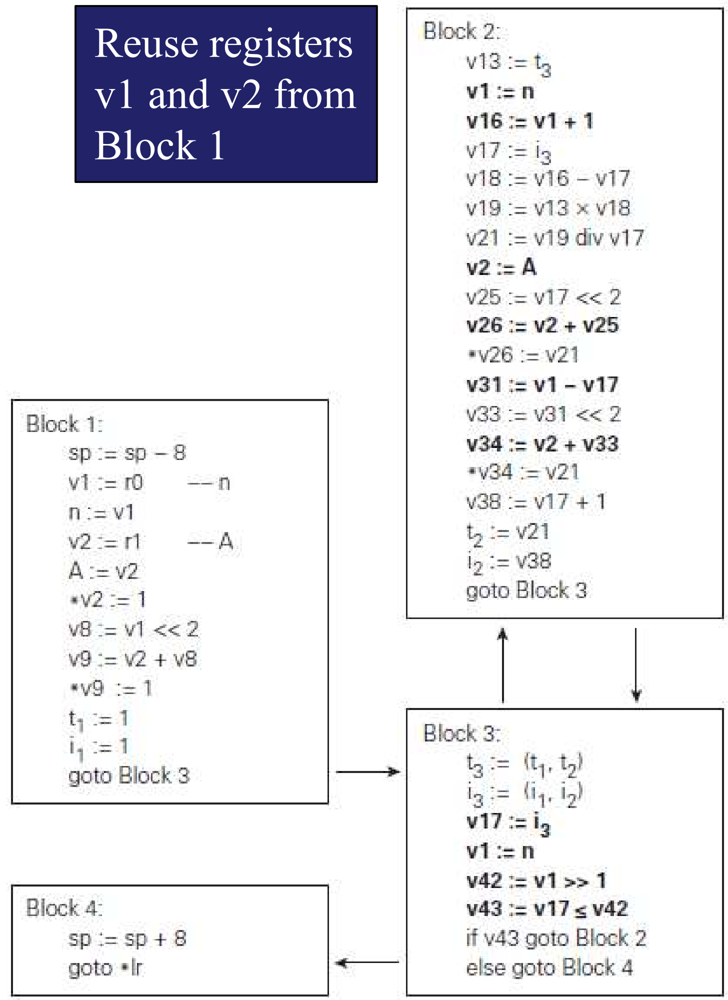


Figure 17.5 Control flow graph for the `combinations` subroutine, in static single assignment (SSA) form. Changes from Figure C-17.4 are shown in boldface type.

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:

$$Out_B = Gen_B \cup (In_B \setminus Kill_B)$$
$$In_B = \bigcap_{\text{predecessors } A \text{ of } B} Out_A$$

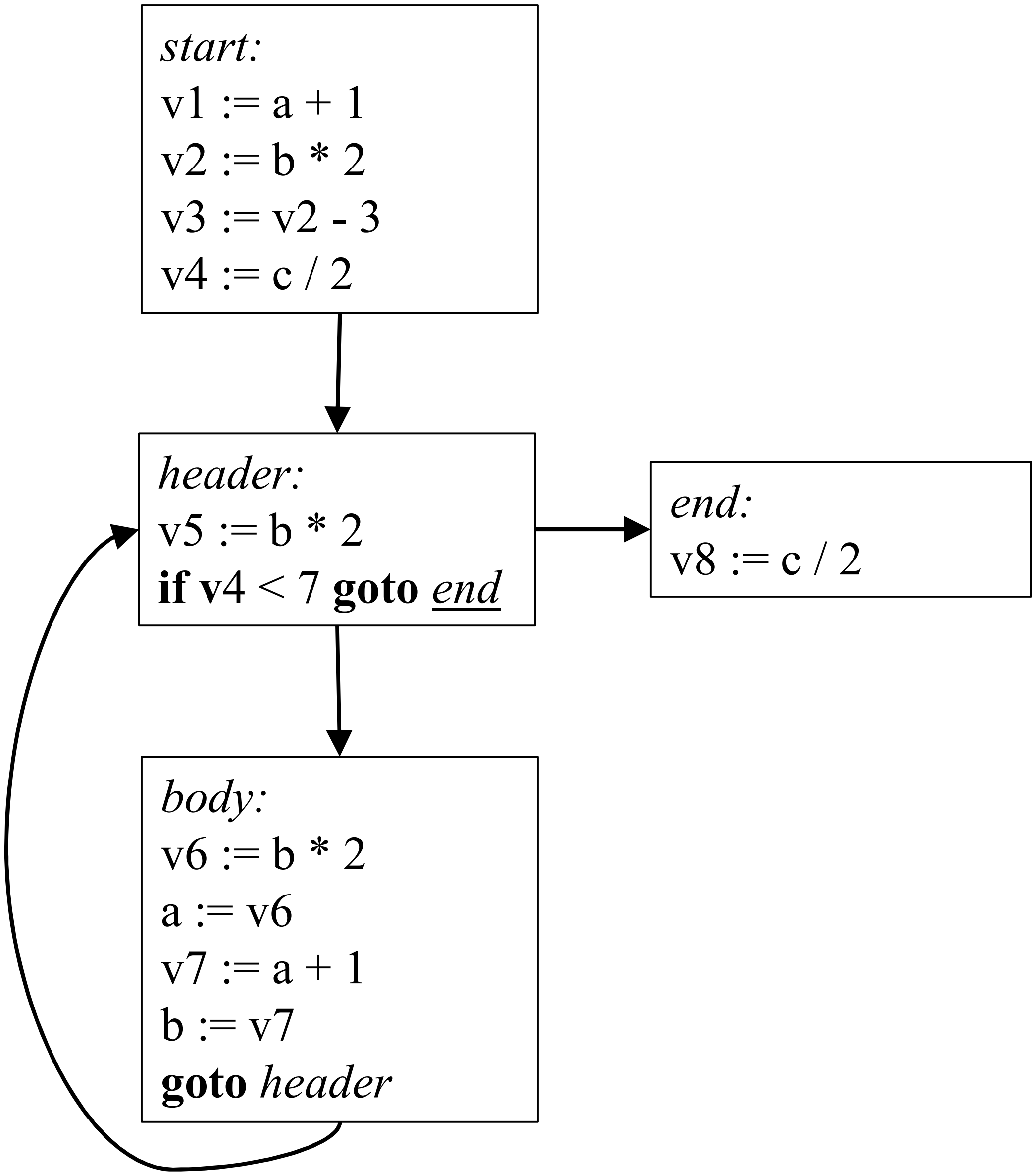
- Our initial condition is $In_1 = \emptyset$: no expressions are available at the beginning of execution

ELSEVIER

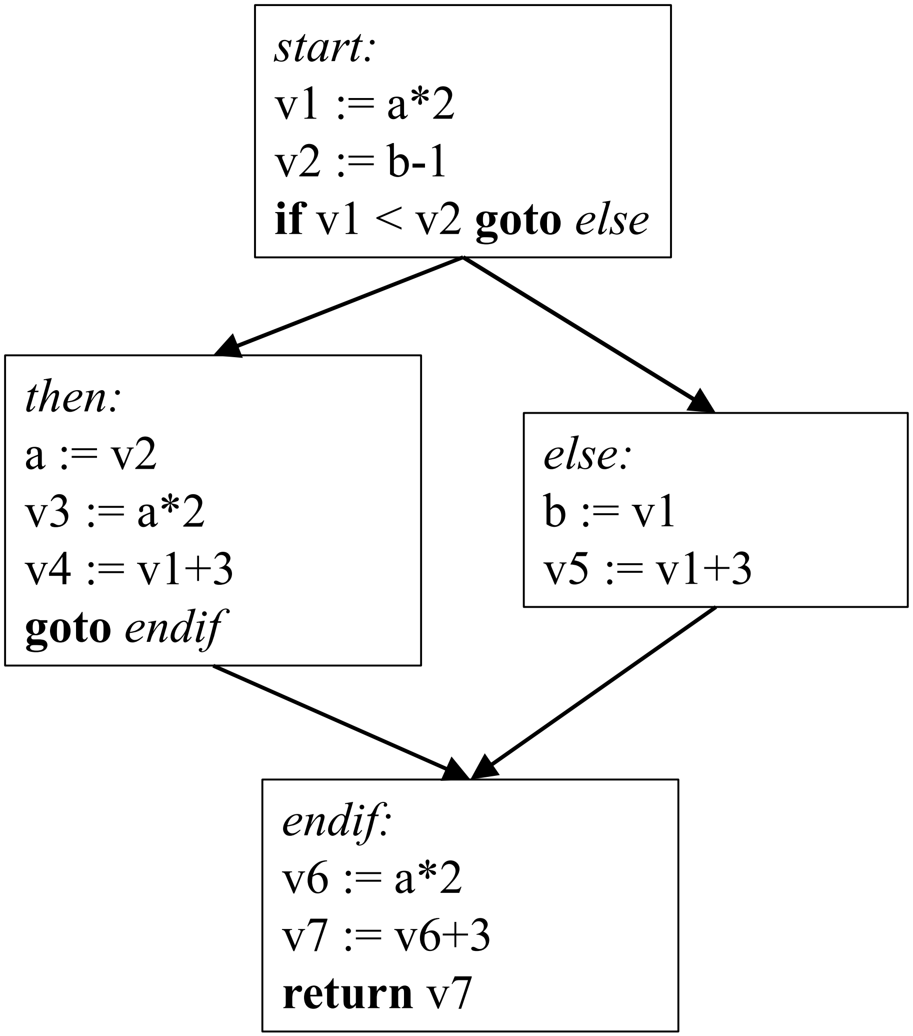
Global Redundancy and Data Flow Analysis

- 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



Global Redundancy and Data Flow Analysis

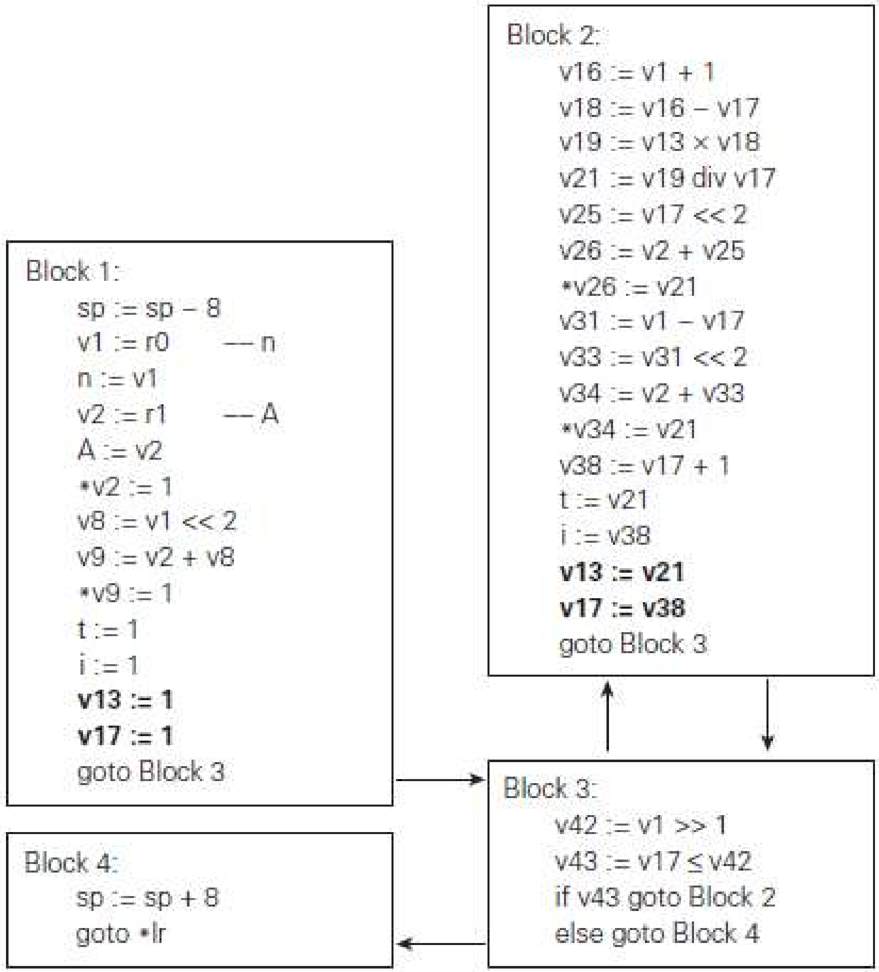


Figure 17.7 **Control flow graph for the `combinations` subroutine after performing global common subexpression elimination.** Note the absence of the many load instructions of Figure C-17.6. Compensating register–register moves are shown in boldface type.

Global Redundancy and Data Flow Analysis

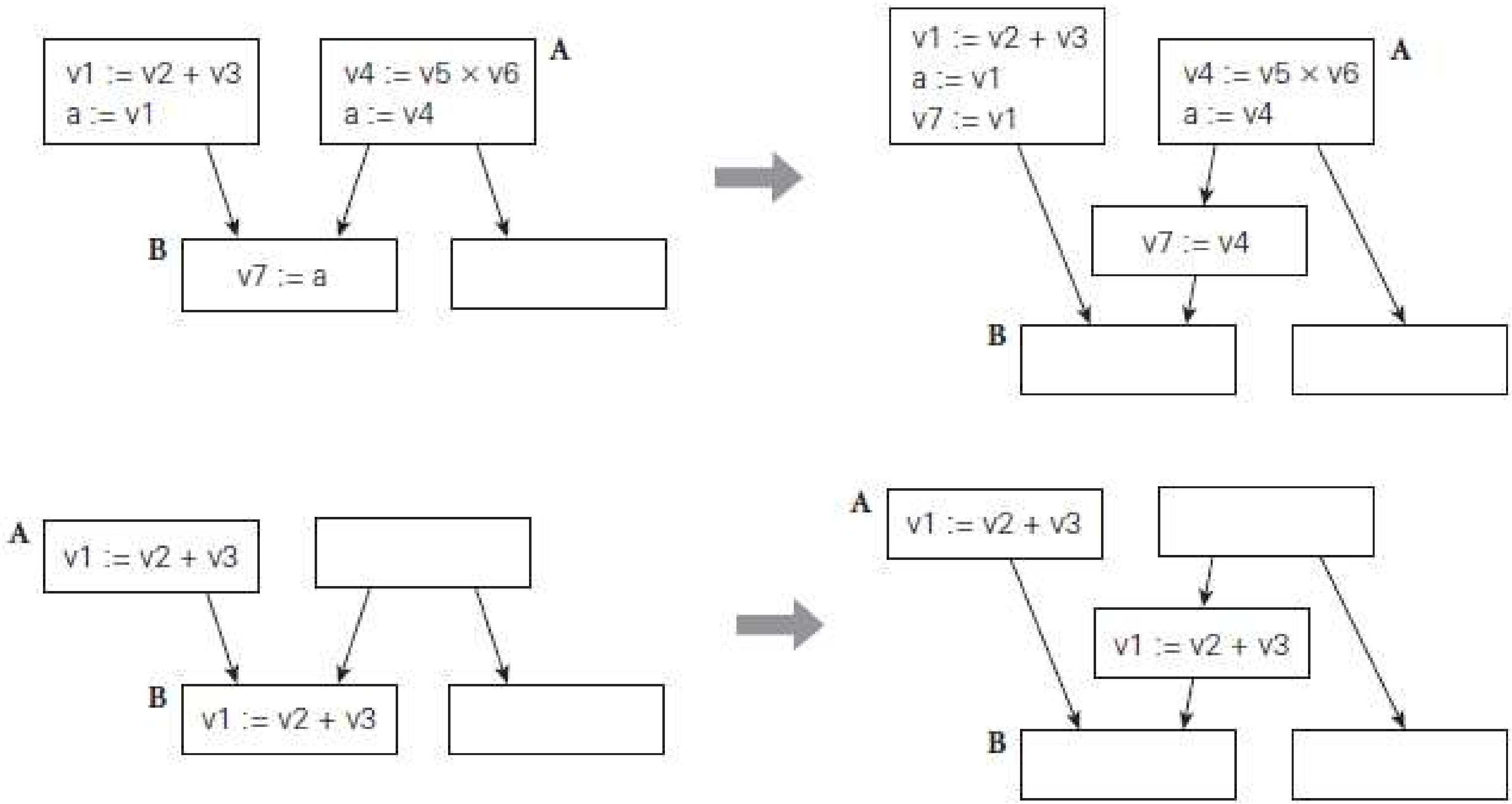


Figure 17.8 Splitting an edge of a control flow graph to eliminate a redundant load (*top*) or a partially redundant computation (*bottom*).

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

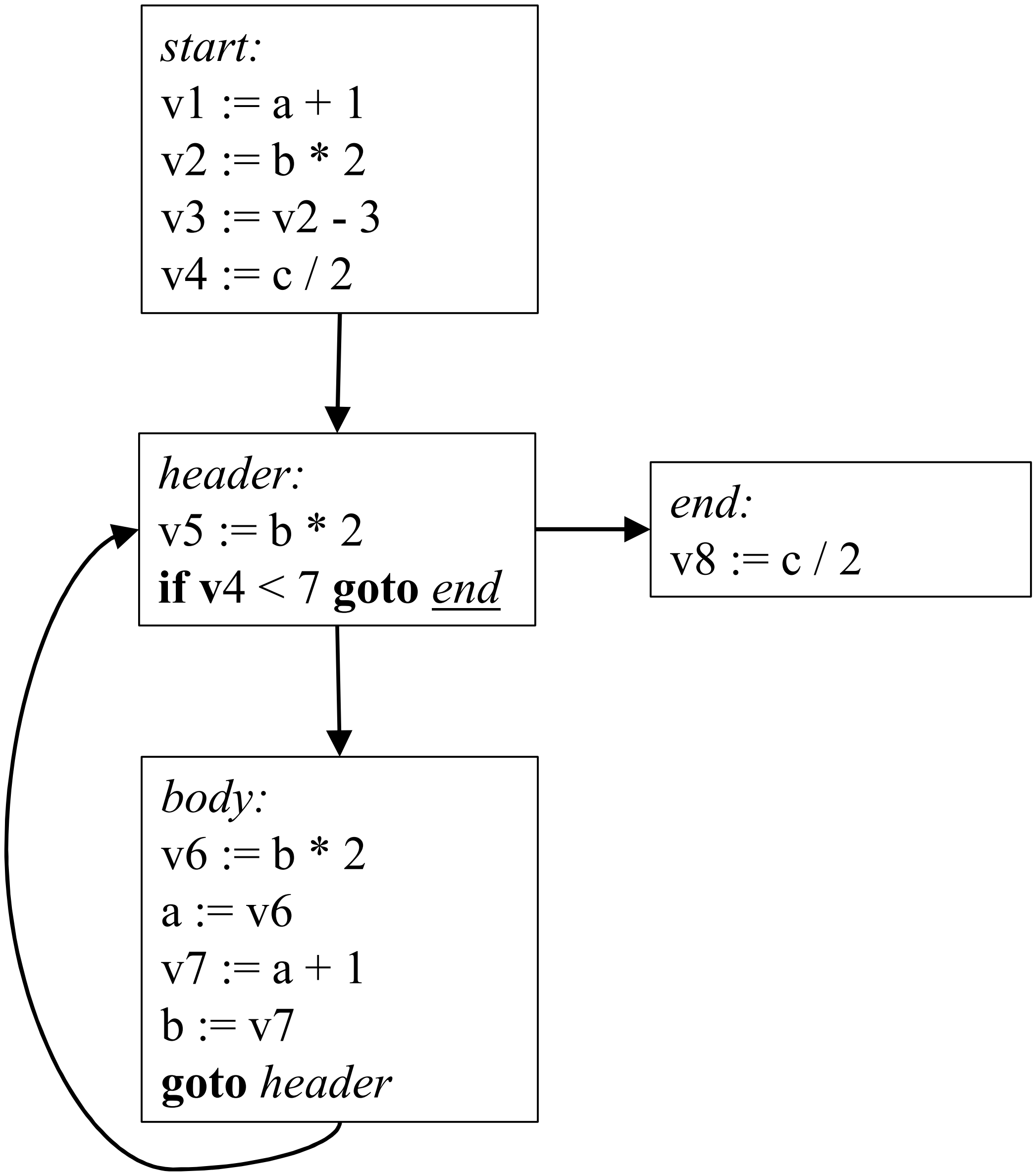
Global Redundancy and Data Flow Analysis

- 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:

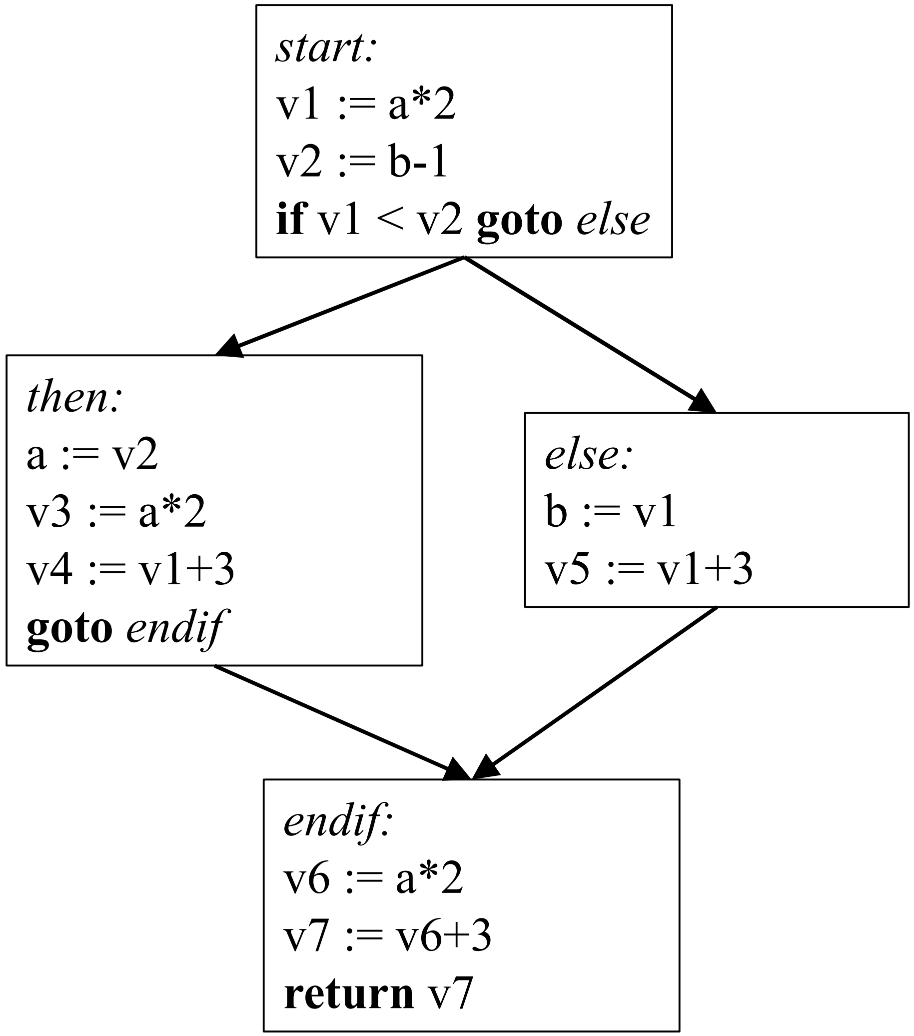
$$\begin{aligned} In_B &= Gen_B \cup (Out_B \setminus Kill_B) \\ Out_B &= \bigcup_{\text{successors } C \text{ of } B} In_C \end{aligned}$$

ELSEVIER

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

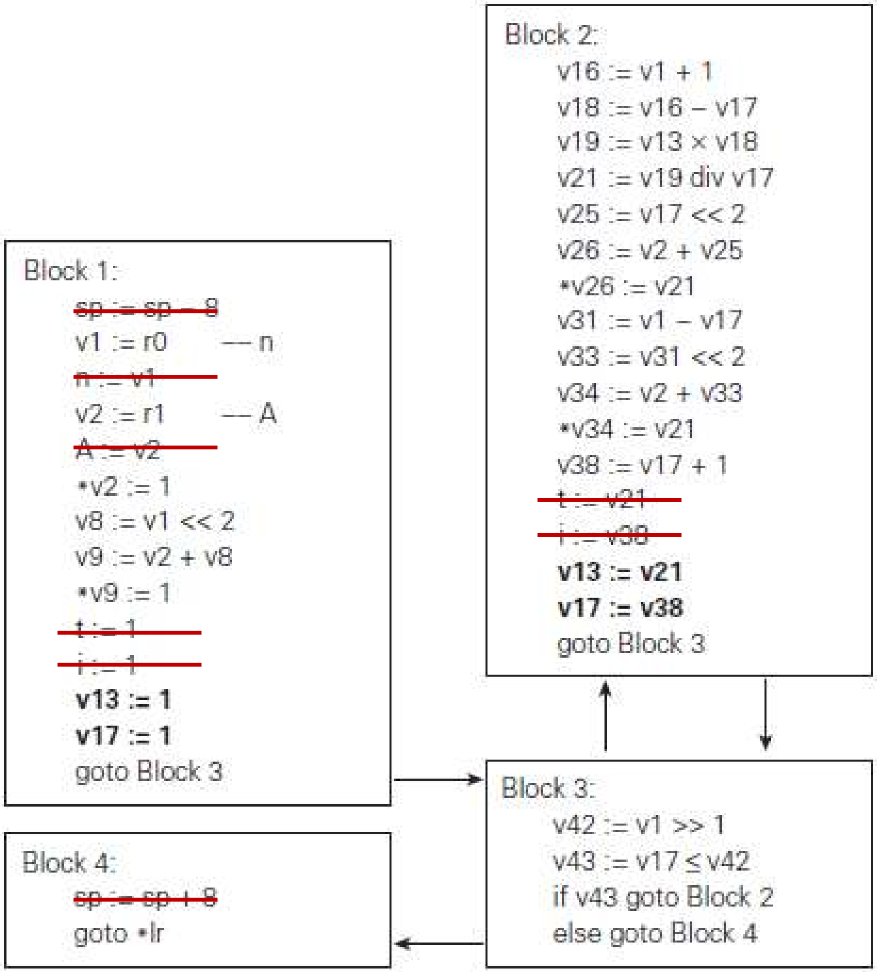


Figure 17.7 Control flow graph for the combinations subroutine after **common subexpression elimination.** Note the absence of the many load i ure C-17.6. Compensating register–register moves are shown in boldface typ

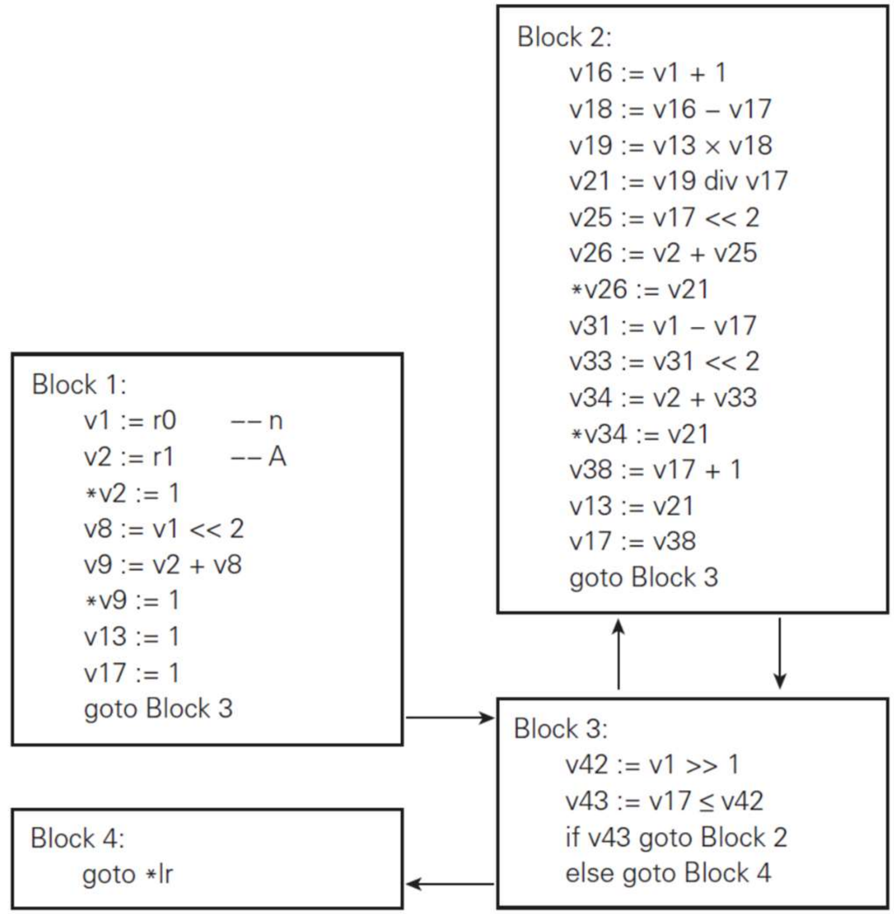


Figure 17.9 Control flow graph for the combinations subroutine after perform **variable analysis.** Starting with Figure C-17.7, the compiler has eliminated all stores to and i. It has also dropped the changes to the stack pointer that used to appear in the sub prologue and epilogue: we don't need space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