

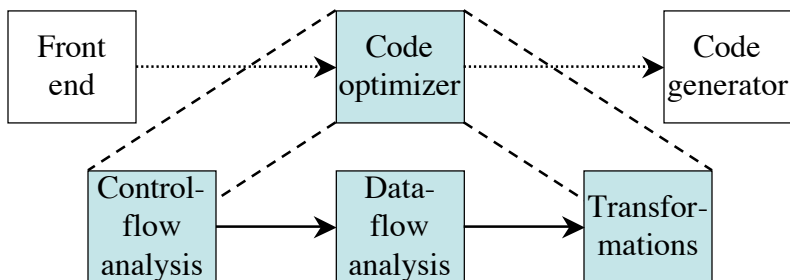
# Code Optimization

## Chapter 10

COP5621 Compiler Construction  
Copyright Robert van Engelen, Florida State University, 2005

## The Code Optimizer

- Control flow analysis: CFG (Ch. 9)
- Data-flow analysis
- Transformations



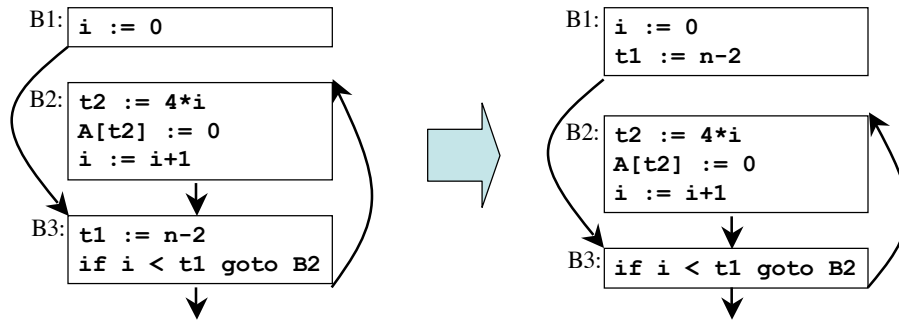
## Code Optimizations

- Local/global common subexpression elimination
- Dead-code elimination
- Instruction reordering
- Constant folding
- Algebraic transformations
- Copy propagation
- *Loop optimizations*

## Loop Optimizations

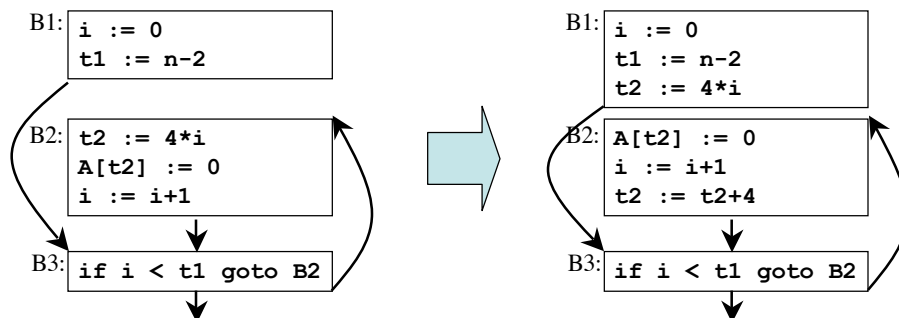
- Code motion
- Induction variable elimination
- Reduction in strength
- ... lots more

## Code Motion



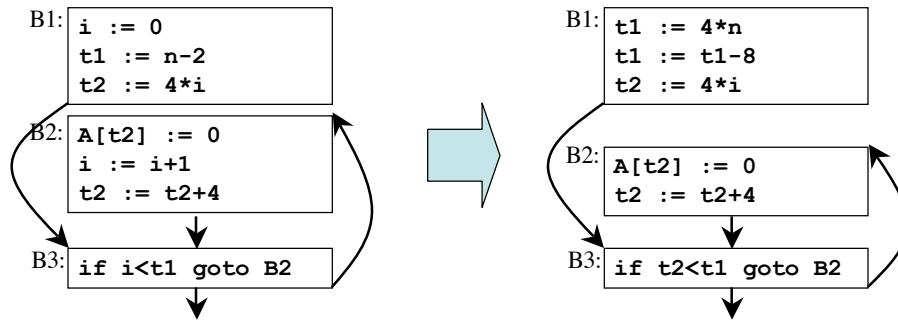
Move *loop-invariant computations* before the loop

## Strength Reduction



Replace expensive computations with *induction variables*

## Reduction Variable Elimination

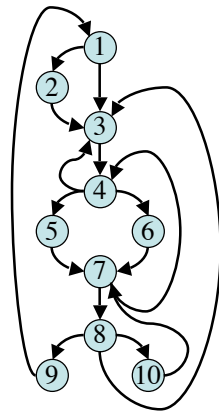


Replace induction variable in expressions with another

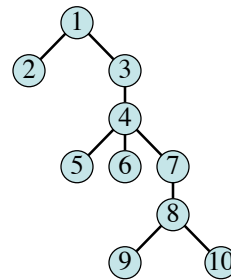
## Determining Loops in Flow Graphs: Dominators

- Dominators:  $d \text{ dom } n$ 
  - Node  $d$  of a CFG *dominates* node  $n$  if *every* path from the initial node of the CFG to  $n$  goes through  $d$
  - The loop entry dominates all nodes in the loop
- The *immediate dominator*  $m$  of a node  $n$  is the last dominator on the path from the initial node to  $n$ 
  - If  $d \neq n$  and  $d \text{ dom } n$  then  $d \text{ dom } m$

## Dominator Trees



CFG

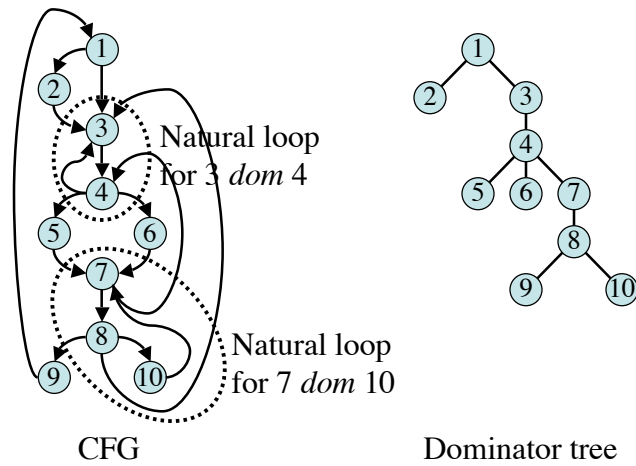


Dominator tree

## Natural Loops

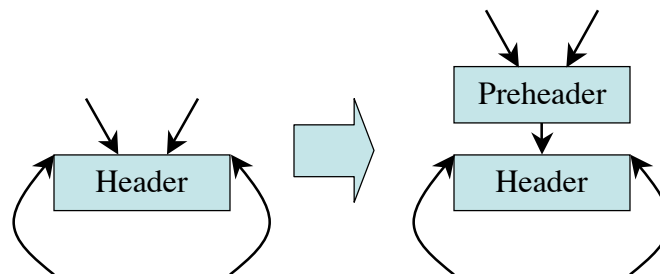
- A *back edge* is an edge  $a \rightarrow b$  whose head  $b$  dominates its tail  $a$
- Given a back edge  $n \rightarrow d$ 
  - The *natural loop* consists of  $d$  plus the nodes that can reach  $n$  without going through  $d$
  - The *loop header* is node  $d$
- Unless two loops have the same header, they are disjoint or one is nested within the other
  - A nested loop is an *inner loop* if it contains no other loops

## Natural (Inner) Loops Example



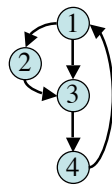
## Pre-Headers

- To facilitate loop transformations, a compiler often adds a *preheader* to a loop
- Code motion, strength reduction, and other loop transformations populate the preheader

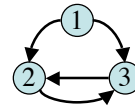


## Reducible Flow Graphs

- *Reducible graph* = disjoint partition in forward and back edges such that the forward edges form an acyclic (sub)graph



Example of a reducible CFG

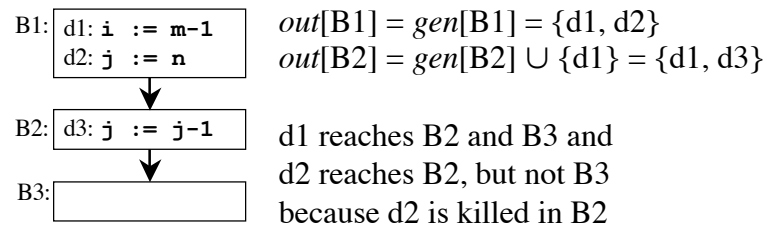


Example of a nonreducible CFG

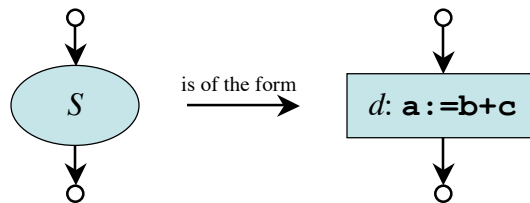
## Global Data-Flow Analysis

- To apply global optimizations on basic blocks, *data-flow information* is collected by solving systems of *data-flow equations*
- Suppose we need to determine the *reaching definitions* for a sequence of statements  $S$ 

$$out[S] = gen[S] \cup (in[S] - kill[S])$$



## Reaching Definitions

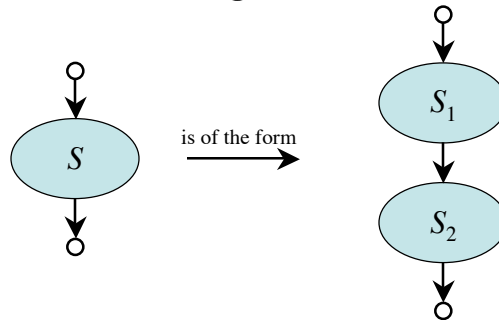


Then, the data-flow equations for  $S$  are:

$$\begin{aligned} gen[S] &= \{d\} \\ kill[S] &= D_a - \{d\} \\ out[S] &= gen[S] \cup (in[S] - kill[S]) \end{aligned}$$

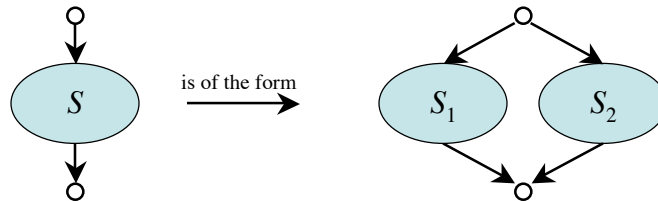
where  $D_a$  = all definitions of  $a$  in the region of code

## Reaching Definitions



$$\begin{aligned} gen[S] &= gen[S_2] \cup (gen[S_1] - kill[S_2]) \\ kill[S] &= kill[S_2] \cup (kill[S_1] - gen[S_2]) \\ in[S_1] &= in[S] \\ in[S_2] &= out[S_1] \\ out[S] &= out[S_2] \end{aligned}$$

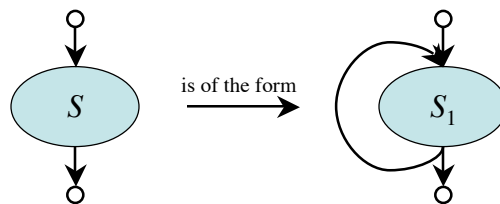
## Reaching Definitions



is of the form

$$\begin{array}{ll}
 \mathit{gen}[S] & = \mathit{gen}[S_1] \cup \mathit{gen}[S_2] \\
 \mathit{kill}[S] & = \mathit{kill}[S_1] \cap \mathit{kill}[S_2] \\
 \mathit{in}[S_1] & = \mathit{in}[S] \\
 \mathit{in}[S_2] & = \mathit{in}[S] \\
 \mathit{out}[S] & = \mathit{out}[S_1] \cup \mathit{out}[S_2]
 \end{array}$$

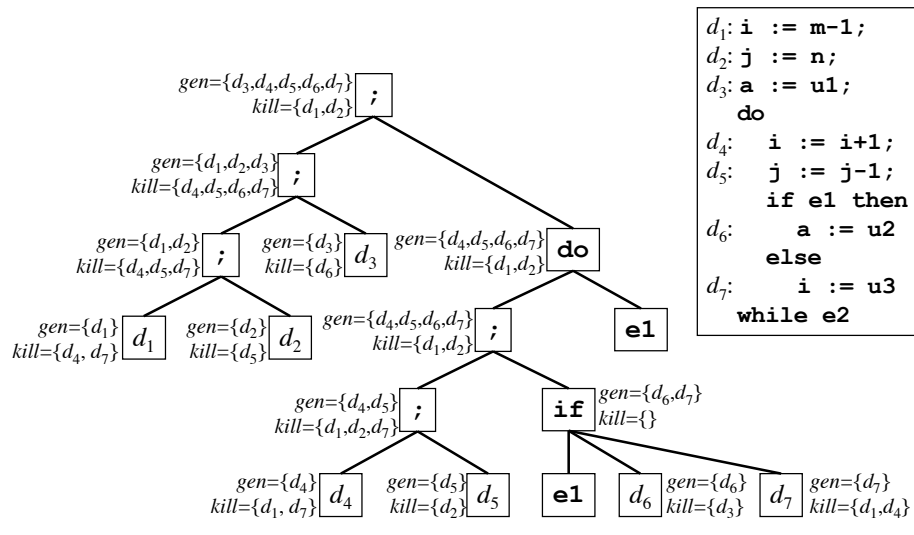
## Reaching Definitions



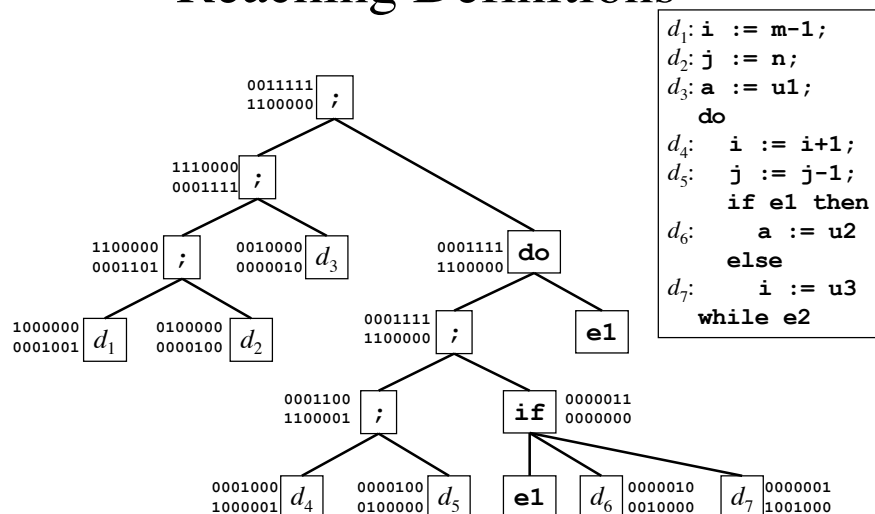
is of the form

$$\begin{array}{ll}
 \mathit{gen}[S] & = \mathit{gen}[S_1] \\
 \mathit{kill}[S] & = \mathit{kill}[S_1] \\
 \mathit{in}[S_1] & = \mathit{in}[S] \cup \mathit{gen}[S_1] \\
 \mathit{out}[S] & = \mathit{out}[S_1]
 \end{array}$$

## Example Reaching Definitions



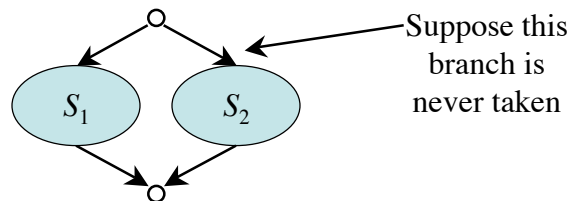
## Using Bit-Vectors to Compute Reaching Definitions



## Accuracy, Safeness, and Conservative Estimations

- *Conservative*: refers to making safe assumptions when insufficient information is available at compile time, i.e. the compiler has to guarantee not to change the meaning of the optimized code
- *Safe*: refers to the fact that a superset of reaching definitions is safe (some may have been killed)
- *Accuracy*: the larger the superset of reaching definitions, the less information we have to apply code optimizations

## Reaching Definitions are a Conservative (Safe) Estimation



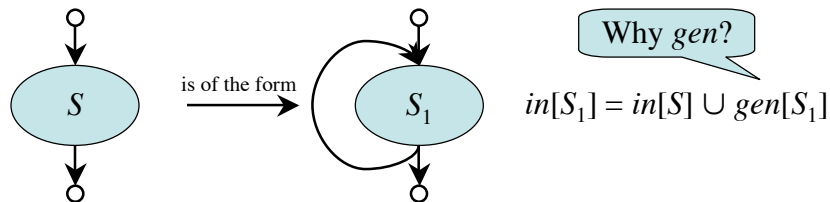
Estimation:

$$\begin{aligned} gen[S] &= gen[S_1] \cup gen[S_2] \\ kill[S] &= kill[S_1] \cap kill[S_2] \end{aligned}$$

Accurate:

$$\begin{aligned} gen'[S] &= gen[S_1] && \subseteq gen[S] \\ kill'[S] &= kill[S_1] && \supseteq kill[S] \end{aligned}$$

## Reaching Definitions are a Conservative (Safe) Estimation



The problem is that

$$in[S_1] = in[S] \cup out[S_1]$$

makes more sense, but we cannot solve this directly, because  $out[S_1]$  depends on  $in[S_1]$

## Reaching Definitions are a Conservative (Safe) Estimation

We have:

- (1)  $in[S_1] = in[S] \cup out[S_1]$
- (2)  $out[S_1] = gen[S_1] \cup (in[S_1] - kill[S_1])$

Solve  $in[S_1]$  and  $out[S_1]$  by estimating  $in^1[S_1]$  using safe but approximate  $out[S_1] = \emptyset$ , then re-compute  $out^1[S_1]$  using (2) to estimate  $in^2[S_1]$ , etc.

$$\begin{aligned} in^1[S_1] &=_{(1)} in[S] \cup out[S_1] = in[S] \\ out^1[S_1] &=_{(2)} gen[S_1] \cup (in^1[S_1] - kill[S_1]) = gen[S_1] \cup (in[S] - kill[S_1]) \\ in^2[S_1] &=_{(1)} in[S] \cup out^1[S_1] = in[S] \cup gen[S_1] \cup (in[S] - kill[S_1]) = in[S] \cup gen[S_1] \\ out^2[S_1] &=_{(2)} gen[S_1] \cup (in^2[S_1] - kill[S_1]) = gen[S_1] \cup (in[S] \cup gen[S_1] - kill[S_1]) \\ &= gen[S_1] \cup (in[S] - kill[S_1]) \end{aligned}$$

Because  $out^1[S_1] = out^2[S_1]$ , and therefore  $in^3[S_1] = in^2[S_1]$ , we conclude that  
 $in[S_1] = in[S] \cup gen[S_1]$

