Lecture 9:

Loop Invariant Computation and Code Motion

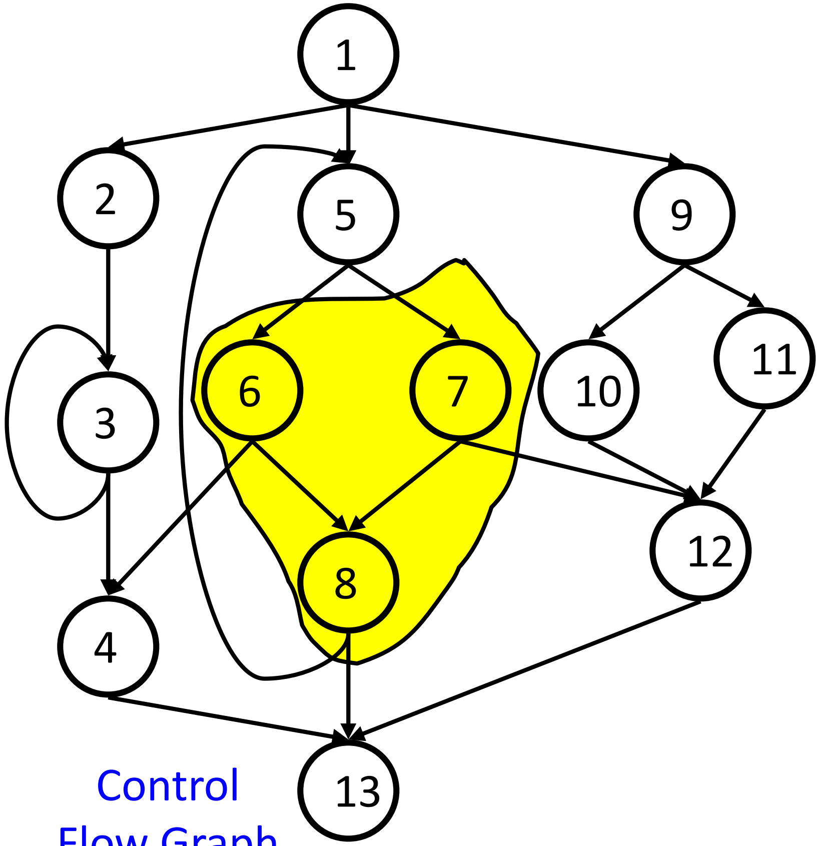
I. Loop-invariant computation

II. Algorithm for code motion

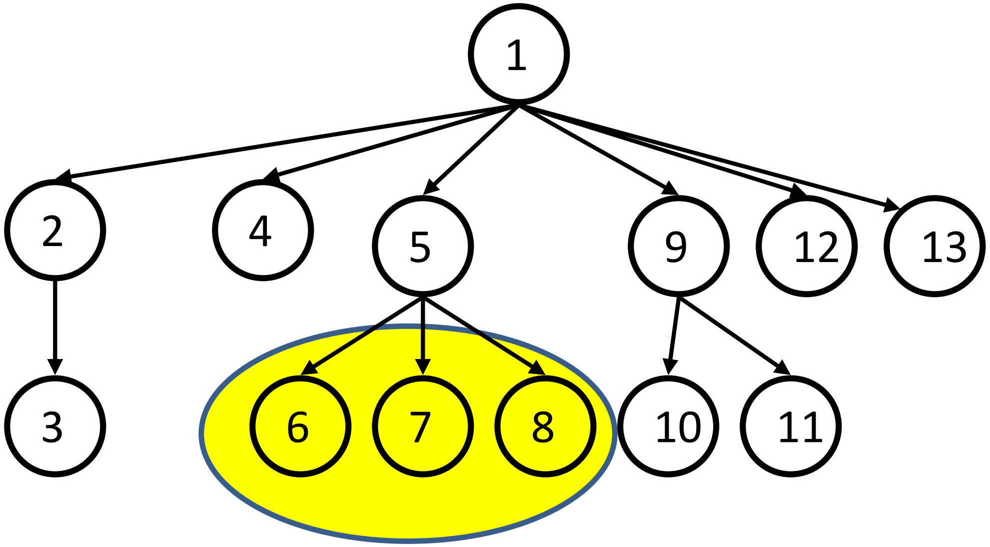
III. Partial redundancy elimination

ALSU 9.5-9.5.2

Review: Dominators



Control Flow Graph



Dominance Tree (D-Tree)

x sdom w iff x is a proper ancestor of w

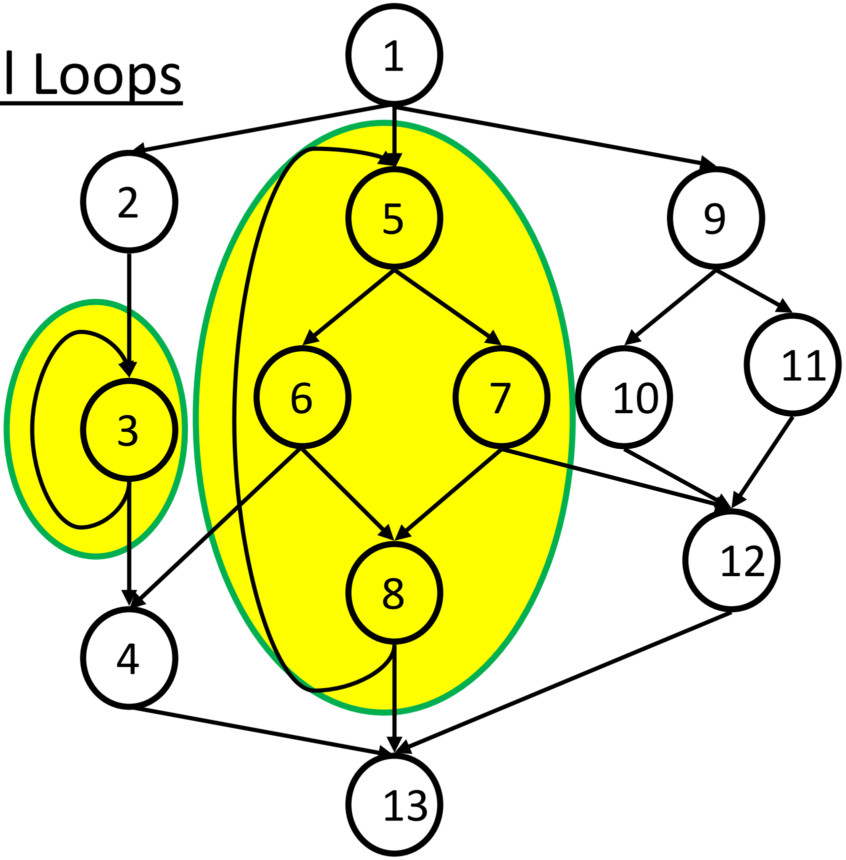
All paths to 6, 7, or 8 must visit 5 first

x strictly dominates w (x sdom w) iff impossible to reach w without passing through x first

x dominates w (x dom w) iff x sdom w OR x = w

Review: Natural Loops

- Single entry-point: ***header***
  - a header dominates all nodes in the loop
- A ***back edge*** is an arc t->h whose head h dominates its tail t
  - a back edge must be a part of at least one loop
- The **natural loop of a back edge** t->h is the smallest set of nodes that includes t and h, and has no predecessors outside the set, except for the predecessors of the header h.



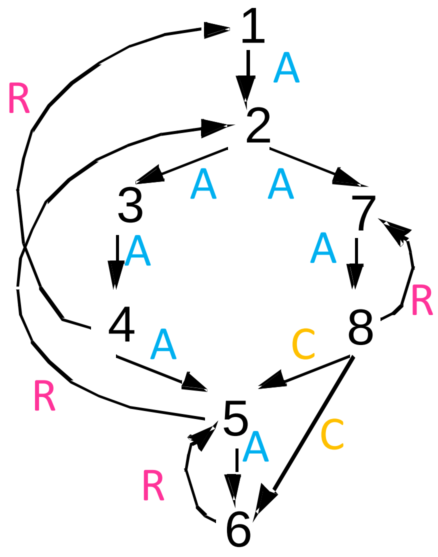
What are the back edges?

3->3 and 8->5

What are the natural loops?

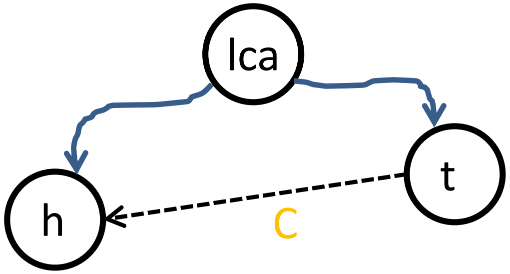
highlighted in yellow above

Recall: Finding Back Edges



1. **Construct a depth-first spanning tree of the CFG**
   - Edges traversed in a depth-first search of the CFG form a depth-first spanning tree
   - Advancing edges (A): from ancestor to proper descendant
   - Cross edges (C): from right to left
   - Retreating edges (R): from descendant to ancestor
2. **Determine which Retreating edges are Back edges** (t->h, h dominates t)

- Note: h can never dominate t for an advancing or cross edge t->h

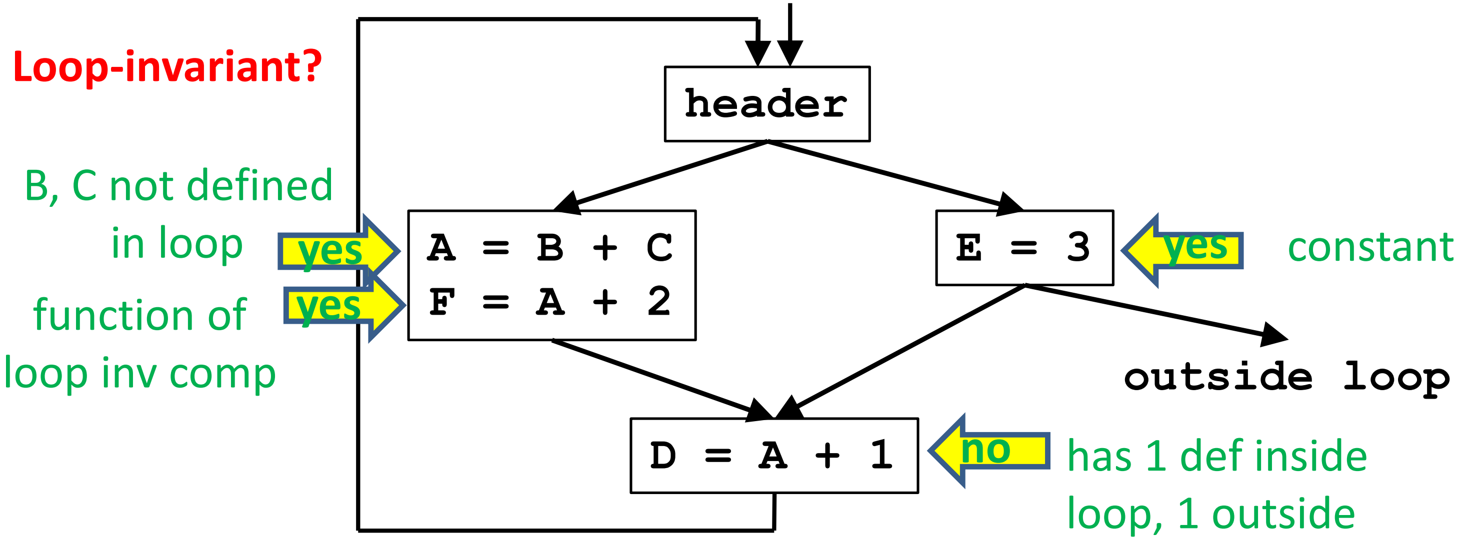


- **Cross edge:** t is not ancestor/descendent of h
- Thus, there is a least common ancestor, lca, of h and t in the tree
- Thus, entry->lca->t is a path without h

- Could apply step 2 to all edges, skipping step 1---but rPostOrder uses step 1

I. Loop-Invariant Computation and Code Motion

- **A loop-invariant computation:**
  - a computation whose value does not change as long as control stays within the loop
- **Code motion:**
  - to move a statement within a loop to the preheader of the loop



Loop-invariant?

`header`

`A = B + C`
`F = A + 2`

`E = 3`

`D = A + 1`

`outside loop`

B, C not defined in loop — yes

function of loop inv comp — yes

yes — constant

no — has 1 def inside loop, 1 outside

Algorithm

- **Observations**
  - Loop invariant
    - operands are defined outside loop or invariant themselves
  - Code motion
    - not all loop invariant instructions can be moved to preheader

- **Algorithm**
  - Find invariant expressions
  - Conditions for code motion
  - Code transformation

Algorithm: Detecting Loop Invariant Computation

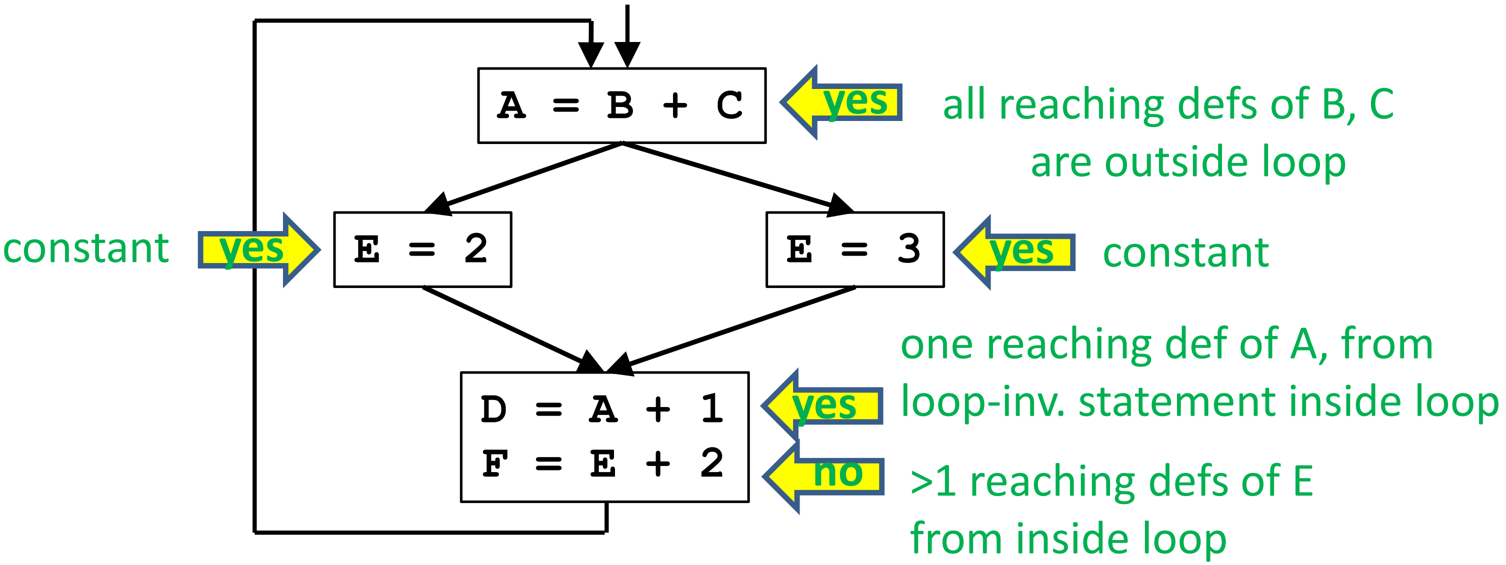
- Compute reaching definitions

- Mark INVARIANT if
  all the definitions of B and C that reach a statement A=B+C
  are outside the loop
  - What about a constant B, C? invariant

- Repeat: Mark INVARIANT if
  - (all reaching definitions of B are outside the loop OR
    there is exactly one reaching definition for B and it is from a loop-invariant statement inside the loop)
  - AND (similarly for C)

  until no changes to the set of loop-invariant statements occur.

Which Statements are Loop Invariant?



all reaching defs of B, C are outside loop

constant

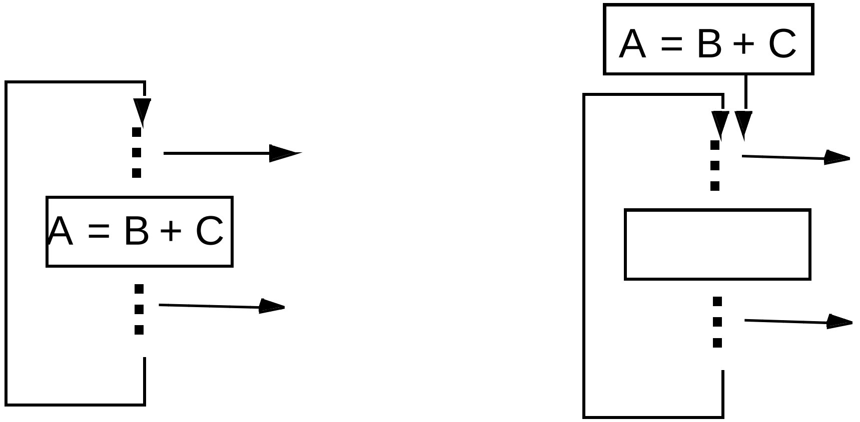
constant

one reaching def of A, from loop-inv. statement inside loop

>1 reaching defs of E from inside loop

II. Conditions for Code Motion

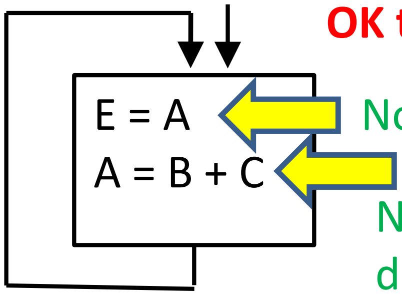
- **Correctness: Movement does not change semantics of program**
- **Performance: Code is not slowed down**



OK to move?

No (moved past exit)

- **Basic idea: defines once and for all**
  - control flow: once?
    - Code dominates all exits
  - other definitions: for all?
    - No other definition
  - uses of the definition: for all?
    - Dominates use or no other reaching defs to use



OK to move?

Not loop inv.

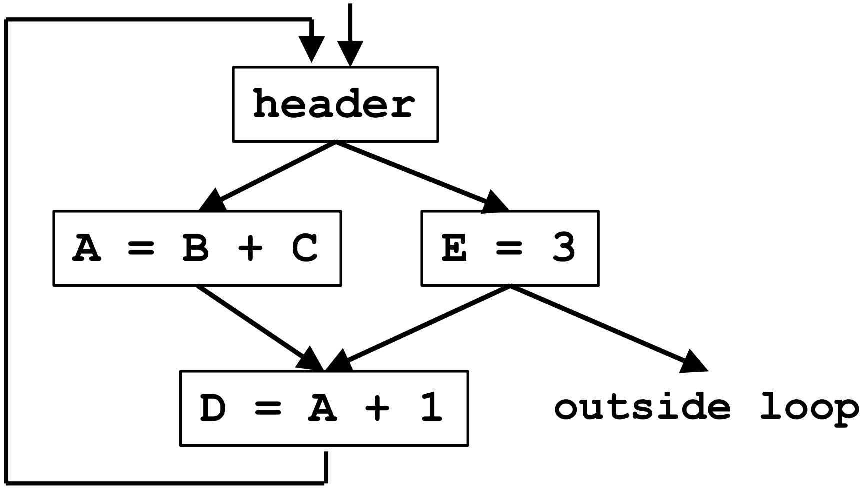
No (doesn't dominate use)

Code Motion Algorithm

Given: a set of nodes in a loop

- **Compute reaching definitions**
- **Compute loop invariant computation**
- **Compute dominators**
- **Find the exits of the loop (i.e. nodes with successor outside loop)**
- **Candidate statement for code motion:**
  - loop invariant
  - in blocks that dominate all the exits of the loop
  - assign to variable not assigned to elsewhere in the loop
  - in blocks that dominate all blocks in the loop that use the variable assigned
- **Perform a depth-first search of the blocks**
  - Move the candidate to the preheader if
    all the invariant operations it depends upon have been moved

Code Motion Examples



header

A = B + C

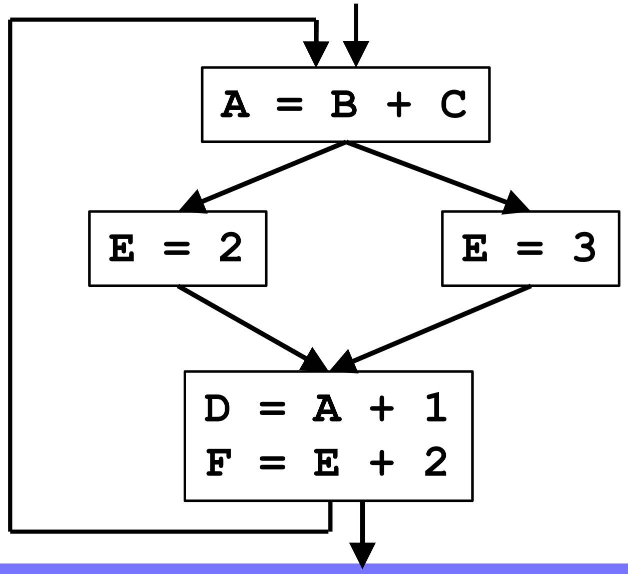
E = 3

D = A + 1

outside loop

Which statements can be moved to loop preheader?

Only E=3: only statement dominating all exits



A = B + C

E = 2

E = 3

D = A + 1
F = E + 2

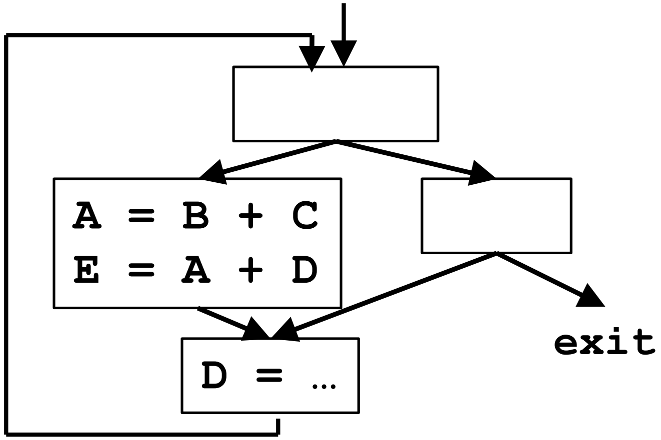
A=B+C
D=A+1

(Although E=2, E=3 are invariant, neither is only def of E)

defines once and for all

More Aggressive Optimizations

- **Gamble on: most loops get executed**
  - When can we relax constraint of dominating all exits?



Can relax if destination not live after loop
& can compute in preheader
w/o causing an exception

- **Landing pads**

```
While p do loop-body    ➔    if p {
                                preheader
                                repeat
                                   loop-body
                                until not p;
                             }
```

Ensures preheader
executes only
if enter loop

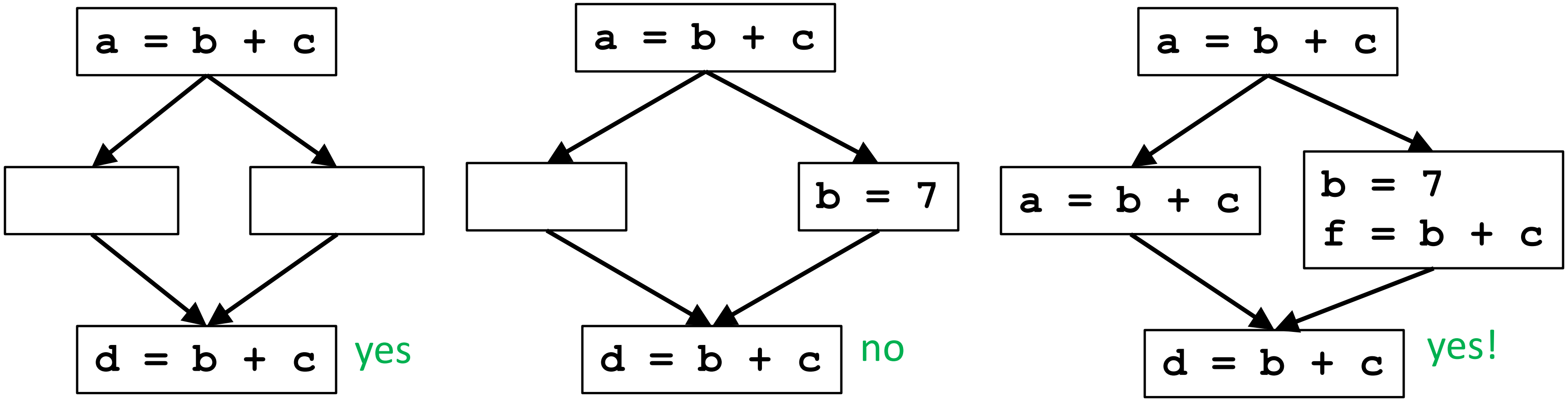
LICM Summary

- **Precise definition and algorithm for loop invariant computation**
- **Precise algorithm for code motion**
- **Use of reaching definitions and dominators in optimizations**

III. Partial Redundancy Elimination

- Sources of Redundancy
  - Global common subexpressions
  - Loop-invariant expresssions
  - Partially redundant expressions

Recall: Global Common Subexpression Elimination



`a = b + c`

`d = b + c` yes

`a = b + c`

`b = 7`

`d = b + c` no

`a = b + c`

`a = b + c`

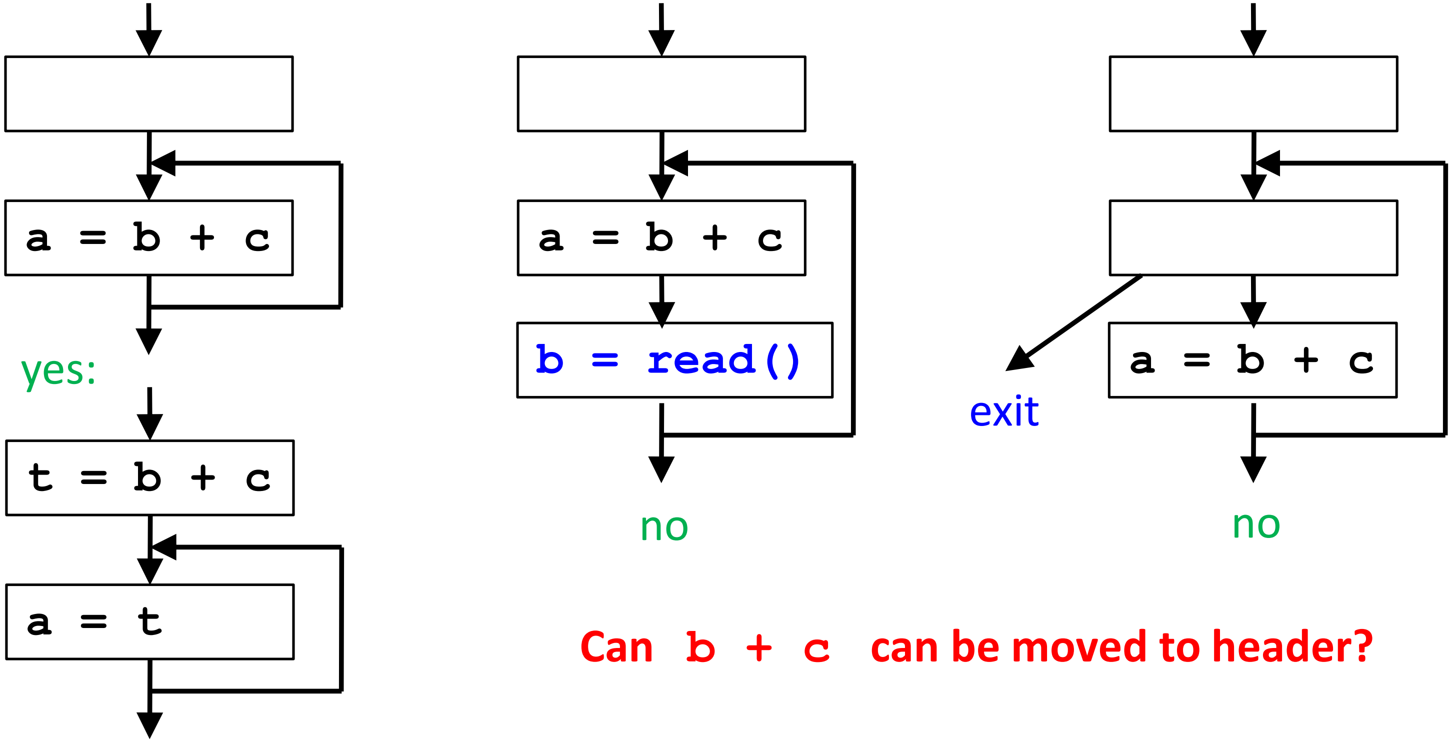
`b = 7`
`f = b + c`

`d = b + c` yes!

Which `b + c` in bottom row is a common subexpression?

- **On every path reaching p,**
  - expression b+c has been computed
  - b, c not overwritten after the expression
- **A common expression may have different values on different paths!**

Loop Invariant Code Motion

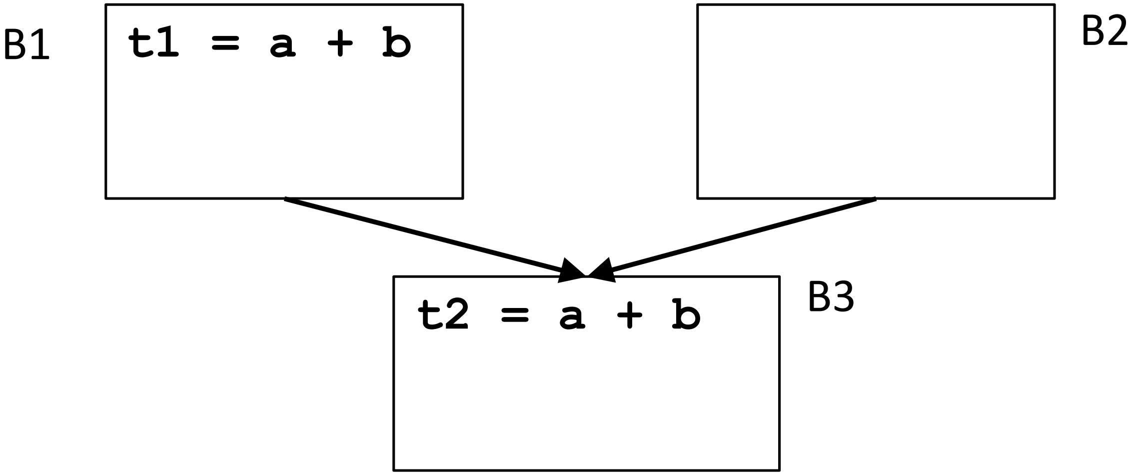


Can `b + c` can be moved to header?

- Given an expression (b+c) inside a loop,
  - does the value of b+c change inside the loop?
  - is the code executed at least once?

Partial Redundancy

- Partially Redundant Computation



B1 `t1 = a + b`

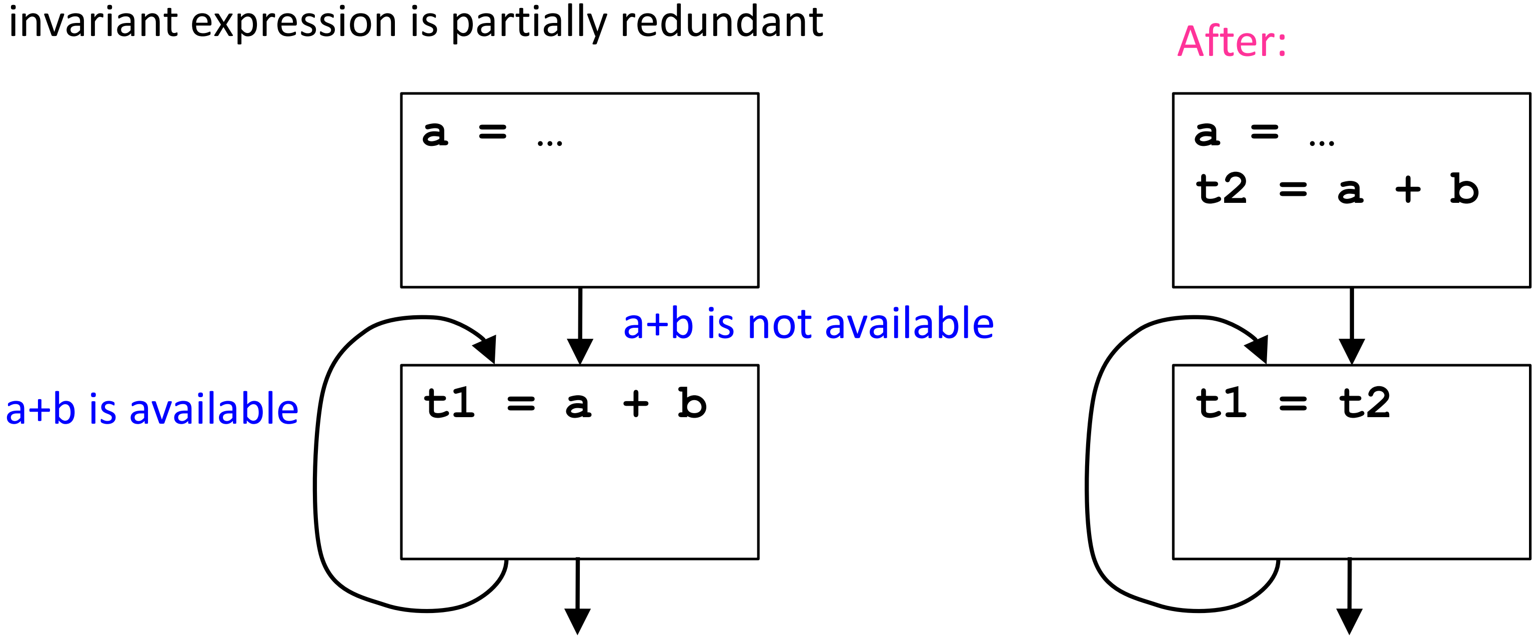
B2

B3 `t2 = a + b`

- Occurrence of expression E at P is **partially redundant** if E is partially available there:
  - E is evaluated along at least one path to P, with no operands redefined since.
- Partially redundant expression can be eliminated if we can insert computations to make it fully redundant.
  - E.g., insert **`t1 = a + b`** in B2

Loop Invariants are Partial Redundancies

- Loop invariant expression is partially redundant



After:

`a = …`

`t2 = a + b`

`t1 = t2`

`a = …`

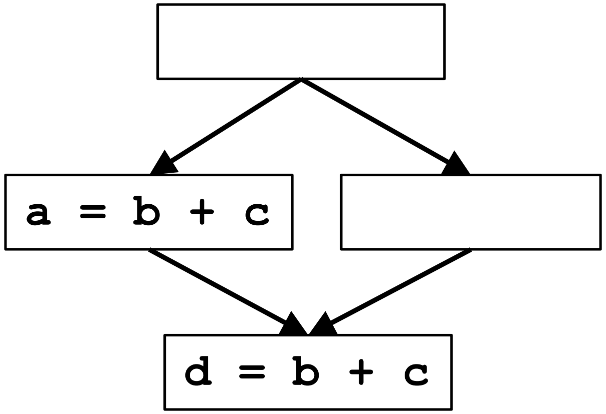
a+b is not available

a+b is available

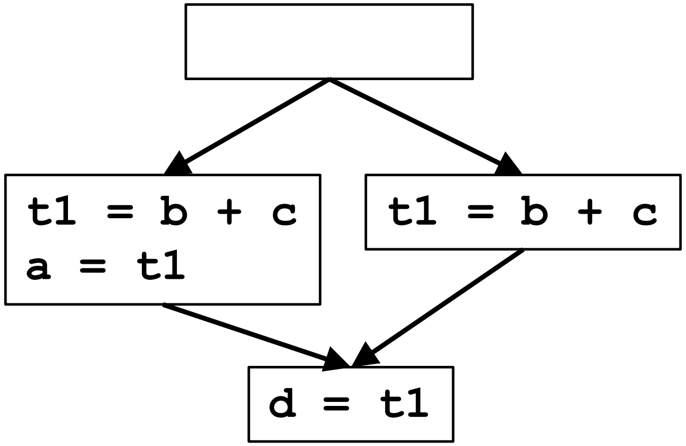
`t1 = a + b`

- As before, partially redundant computation can be eliminated if we insert computations to make it fully redundant.
- Remaining copies can be eliminated through copy propagation or more complex analysis of partially redundant assignments.

Partial Redundancy Elimination



```
a = b + c
d = b + c
```



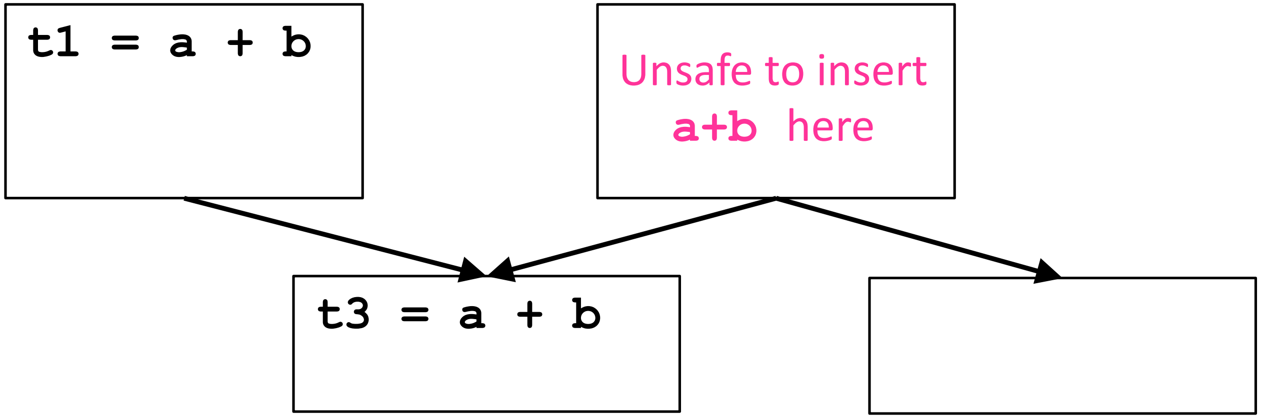
```
t1 = b + c
a = t1
t1 = b + c
d = t1
```

- Can we place calculations of b+c
  such that no path re-executes the same expression?

- Partial Redundancy Elimination (PRE)
  - subsumes:
    - global common subexpression (full redundancy)
    - loop invariant code motion (partial redundancy for loops)

Where Can We Insert Computations?

- **Safety: never introduce a new expression along any path.**



`t1 = a + b`

Unsafe to insert **`a+b`** here

`t3 = a + b`

  - Insertion could introduce exception, change program behavior.
  - If we can add a new basic block, can insert safely in most cases.
  - Solution: insert expression only where it is **anticipated**, i.e., its value computed at point p will be used along ALL subsequent paths (more in next lecture)

- **Performance: never increase the # of computations on any path.**
  - Under simple model, guarantees program won't get worse.
  - Reality: might increase register lifetimes, add copies, lose.

Today's Class

I. Loop-invariant computation

II. Algorithm for code motion

III. Partial redundancy elimination

Friday's Class

- Lazy Code Motion
  - ALSU 9.5.3-9.5.5