# Global Optimizations

## Overview

- Last lecture
  - → Classical optimizations
  - → Basic block construction
  - → Basic block DAG construction
- This lecture
  - → Dominators and loops
  - → General code motion
  - → Global common subexpression elimination



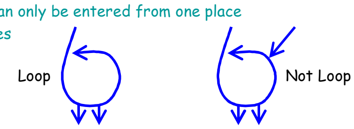
## Scope of Optimization

- Peephole
  - → Just 1-3 instructions
  - → May be in different basic blocks
- Local
  - → Within single basic block
- Global
  - → Across basic blocks
  - → Usually single procedure body
- Interprocedural
  - → Across procedures
- Multiple versions of optimizations possible
  - → Example: common subexpression elimination
    - Locally using DAG
    - Globally using available expressions dataflow analysis



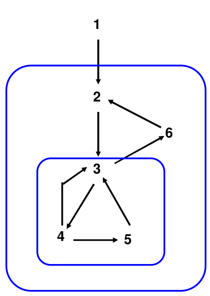
## Dominators and Loops

- Dominator
  - → A node X dominates a node Y if and only if all paths from the control flow graph (CFG) entry node to Y pass through X.
  - → Note every node dominates itself.
- Loops
  - → Let (Y, X) be a CFG edge such that X dominates Y.
  - → Then all nodes on paths from X to Y are in the loop defined by (Y, X).
- Intuitively
  - → Loops can only be entered from one place
  - → Examples






## Dominators and Loops




CFG

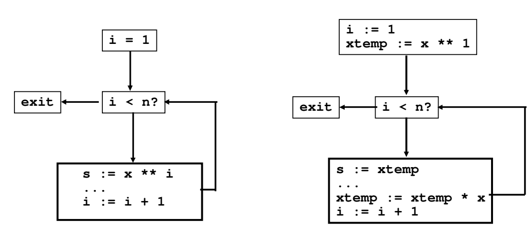
- What is the dominance relation?
  - → dom(1) = { 1,2,3,4,5,6 }
  - → dom(2) = { 2,3,4,5,6 }
  - → dom(3) = { 3,4,5,6 }
  - → dom(4) = { 4,5 }
  - → dom(5) = { 5 }
  - → dom(6) = { 6 }
- What are the loops?
  - → (5,3) = { 3,4,5 }
  - → (4,3) = { 3,4 }

  **Loops coalesced because same loop entry node (3)**

  - → (6,2) = { 2,3,4,5,6 }



## Strength Reduction






1

## General Code Motion

```
n := 1; k := 0; m := 3; read x;
while n < 10 do
  if 2 + x ≥ 5 then k := 5;
  if 3 + k = 3 then m := m + 2;
  n := n + k + m;
endwhile;
```



## General Code Motion

```
1. n := 1; 2. k := 0; 3. m := 3;
4. read x;
5. while n  < 10 do
6.    if 2 * x ≥ 5 then 7. k := 5;
8.    if 3 + k = 3 then 9. m := m + 2;
10.   n := n + k + m;
11.endwhile;
```

- Invariant within loop and therefore moveable.
- Unaffected by definitions in loop and guarded by invariant condition
- Moveable after we move statements 6 and 7.
- Not moveable because may use def of m from statement 9 on previous iteration.



## General Code Motion

```
n := 1; k := 0; m := 3; read x;
while n < 10 do
  if 2 * x ≥ 5 then k := 5;
  if 3 + k = 3 then m := m + 2;
  n := n + k + m;
endwhile;
```

```
n := 1; k := 0; m := 3; read x;
if 2 * x ≥ 5 then k := 5;
t1 := (3 + k = 3);
while n < 10 do
  if t1 then m := m + 2;
  n := n + k + m;
endwhile;
```



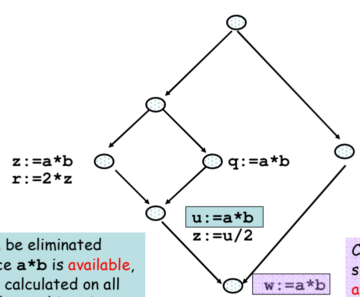
## Additional Optimization: Code Specialization

```
n := 1; k := 0; m := 3; read x;
if 2 * x ≥ 5 then k := 5;
t1 := (3 + k = 3);
if t1 then
  while n < 10 do
    m := m + 2;
    n := n + k + m;
  endwhile;
else
  while n < 10 do
    n := n + k + m;
  endwhile;
endif
```

Specialization of while loop depending on value of `t1`



## Global Common Subexpression Elimination

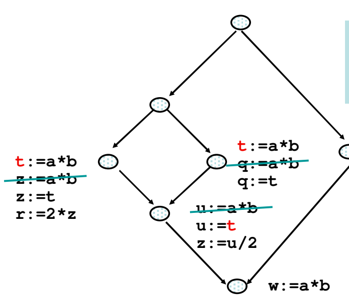



Can be eliminated since **a*b** is available, i.e., calculated on all paths to this point.

Cannot be eliminated since **a*b** is not available on all path reaching this point.



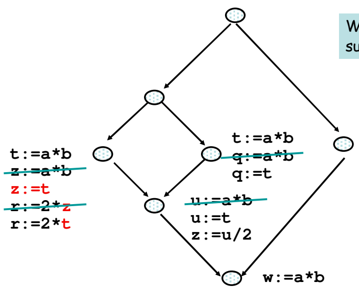
## Global Common Subexpression Elimination




Ensure a*b is assigned to the same variable **t** so it can be used for the assignment to u.

## Forward Substitution

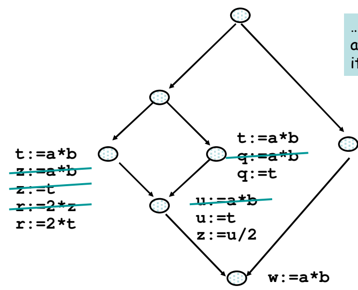
We can then forward substitute t for z…





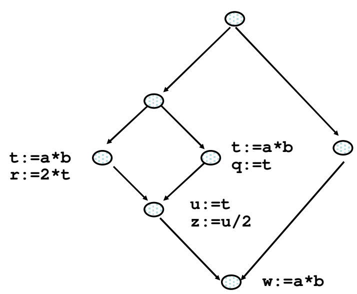

## Dead Code Elimination

…and eliminate the assignment to z since it is now dead code.





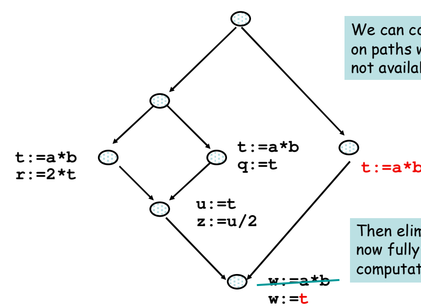

## What Else Can Be Done?






## Partial Redundancy Elimination

We can compute a*b on paths where it is not available…




Then eliminate the now fully redundant computation of a*b