Loops in Flow graphs

Peephole Optimization

- Examines a short sequence of target instructions in a window (peephole) and replaces the instructions by a faster and/or shorter sequence when possible
- Applied to intermediate code or target code

Typical Optimization

- Redundant instruction elimination
- Flow-of-control optimizations
- Algebraic simplifications
- Use of machine idioms

Eliminating Redundant Loads and Stores

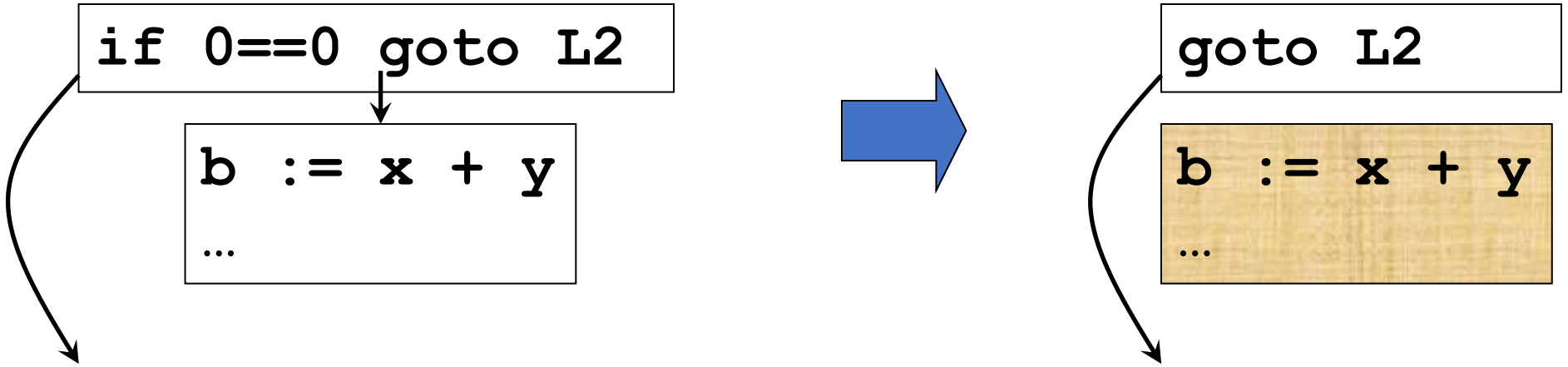
- Consider

```
MOV R0,a
MOV a,R0
```

- The second instruction can be deleted, but only if it is not labeled with a target label
  - Peephole represents sequence of instructions with at most one entry point
- The first instruction can also be deleted if *live*(**a**)=false

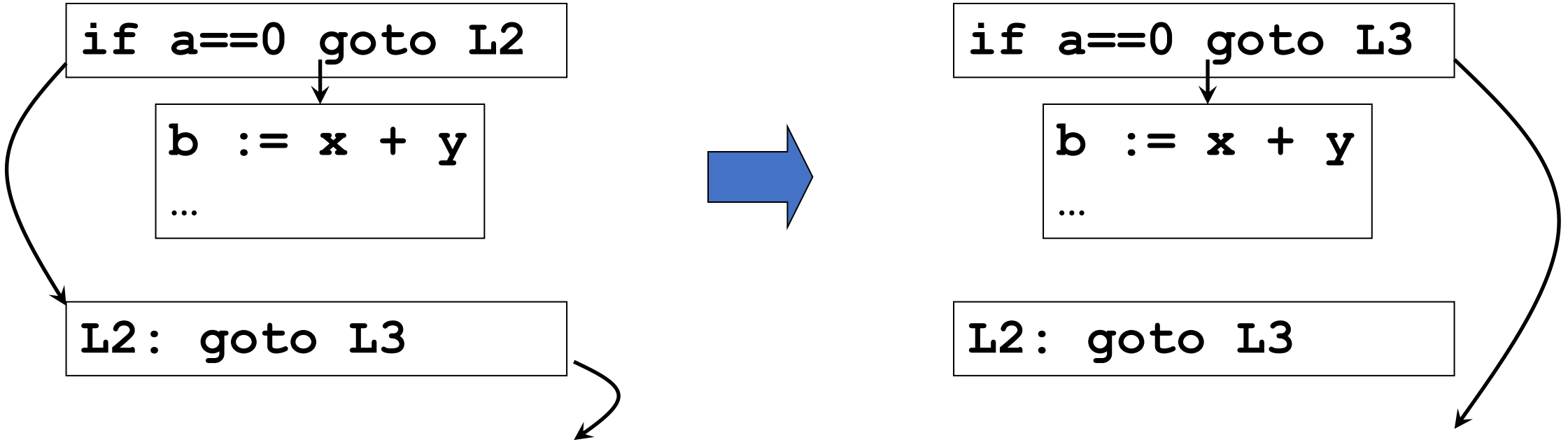
Deleting Unreachable Code

- Unlabeled blocks can be removed



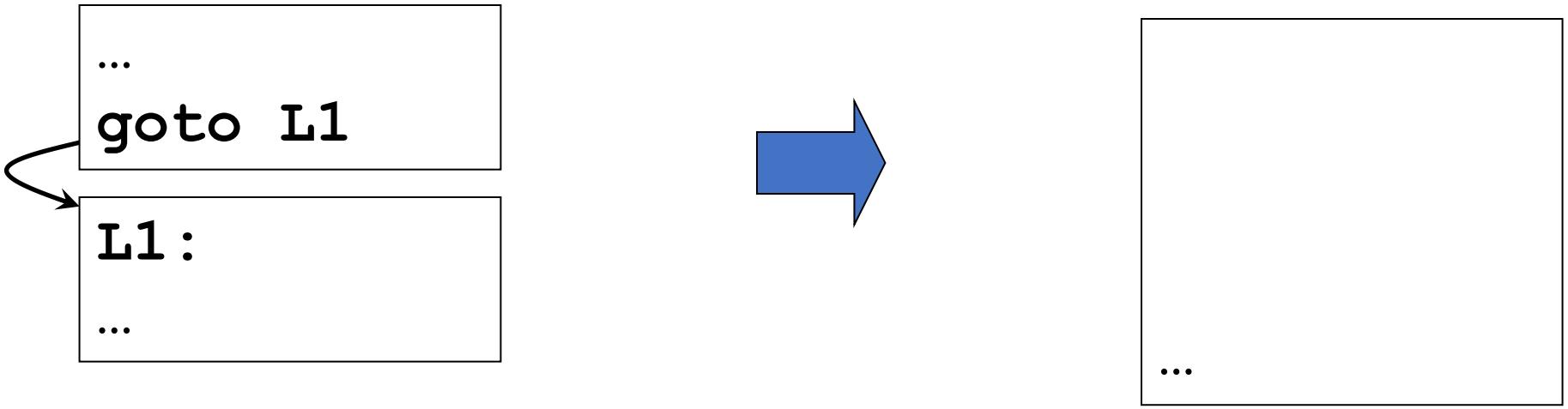
Branch Chaining

- Shorten the length of branches by modifying target labels



Flow-of-Control Optimizations

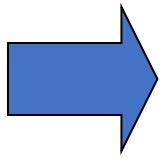
- Remove redundant jumps



Peephole Optimizations

- *Reduction in strength*: replace expensive arithmetic operations with cheaper ones

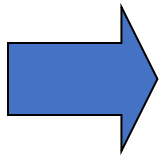
```
…
a := x ^ 2
b := y / 8
```



```
…
a := x * x
b := y >> 3
```

- Use of machine idioms

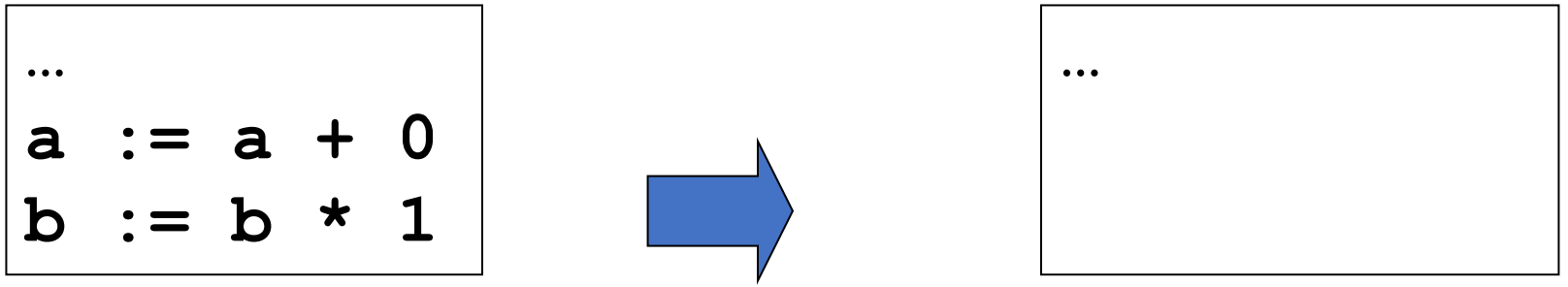
```
…
a := a + 1
```



```
…
inc a
```

Other Peephole Optimizations

- Algebraic simplifications



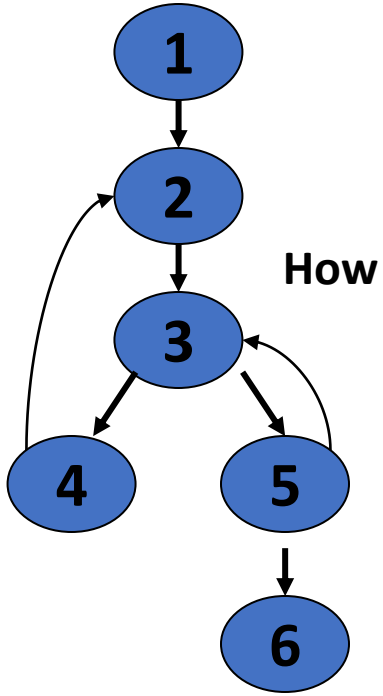
Loops

- Control Flow Graph (CFG) is a graph, which may contain loops, known as strongly-connected-components (SCC)
- Generally, a loop is a directed graph, whose nodes can reach all other nodes along some path
- This includes “unstructured” loops, with multiple entry and multiple exit points

Loops

- A structured loop which is called as a normal loop has one entry point, and (generally) a single point of exit
- Loops created by mapping high-level source programs to IR or assembly code are normal
- Goto can create any loop; break creates additional exits

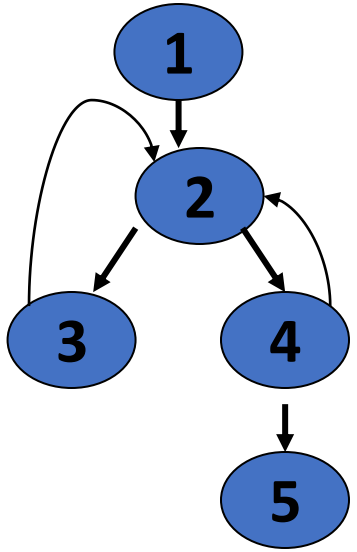
Loops



How many loops?

Unstructured

Loop: 2, 3, 4, 5



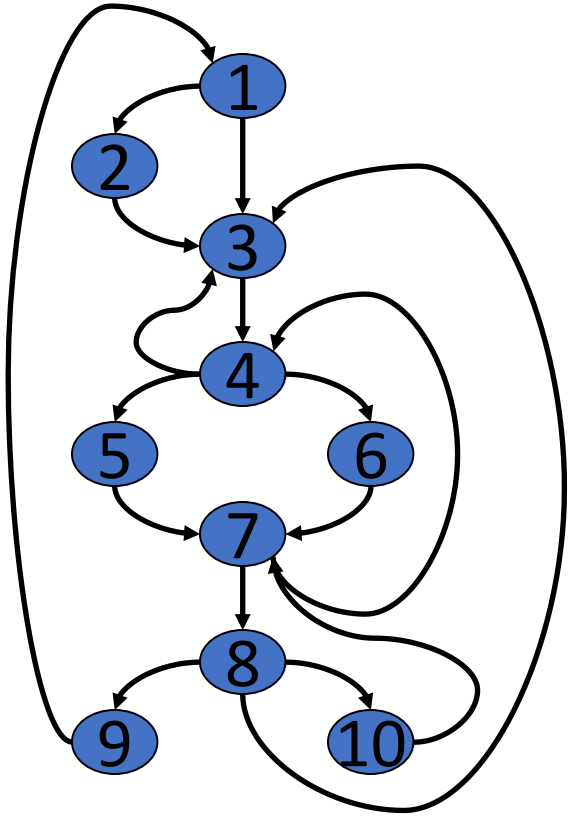
2 proper loops, one unstructured loop

Loop1: 2, 3; Loop2: 2, 4; Loop3: 2, 3, 4

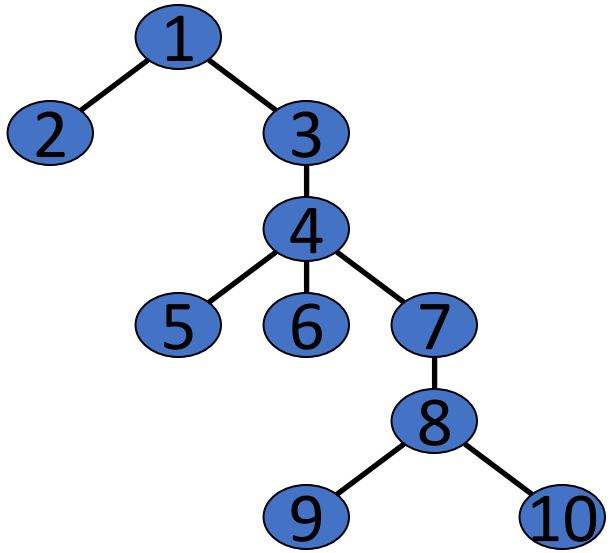
Determining Loops in Flow Graphs: Dominators

- Dominators: *d 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 dom n* then *d dom m*

Dominator Trees



CFG



Dominator tree

Natural Loops

- A *back edge* is 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

Algorithm for natural loop

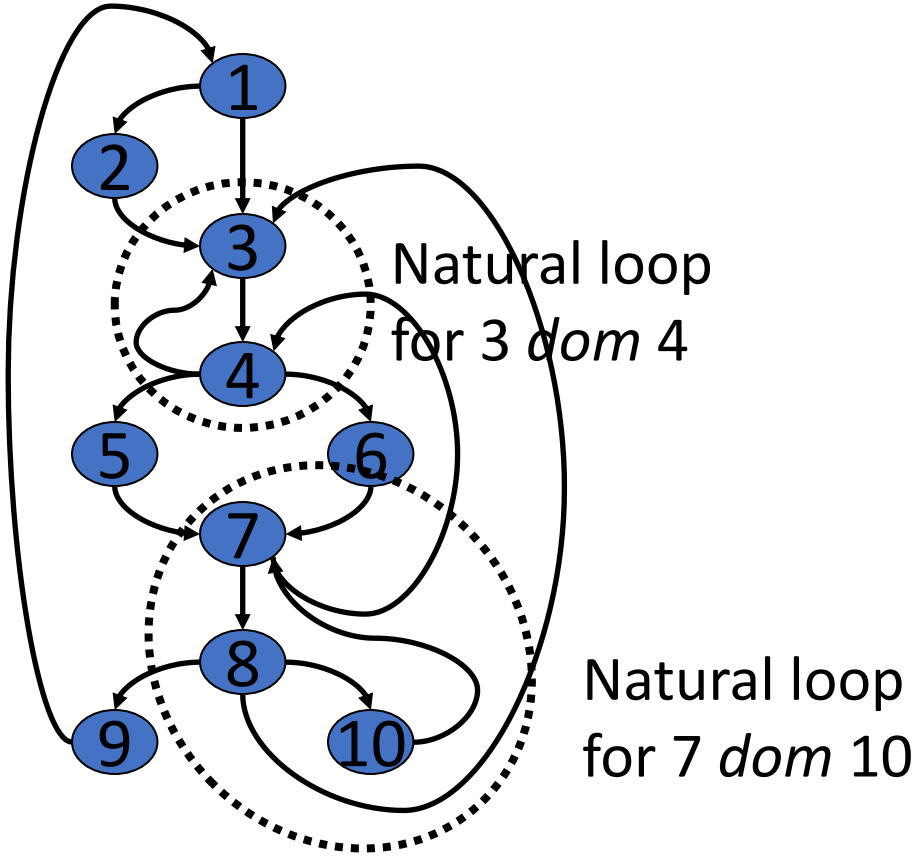
- Each node in loop except for d is placed on stack and its predecessors are examined
- Input: A flow graph G and a back edge n → d
- Output: The set *loop* consisting of all nodes in the natural loop of n→ d

```
Procedure (insert m)
{
   if m is not in loop then
            loop := loop U {m};
    push m onto stack;
}
```

Algorithm for natural loop

```
stack := empty
loop := {d};
insert(n);
while stack not empty
{
   pop m;
   for each predecessor p of m do
            insert(p);
}
```

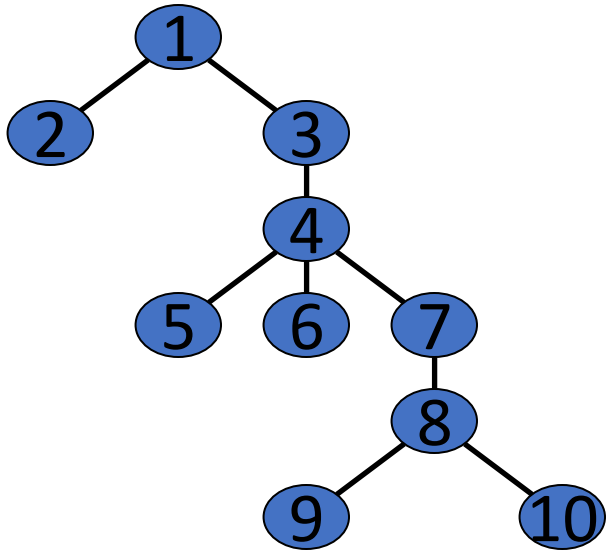
Natural (Inner) Loops Example



Natural loop
for 3 *dom* 4

Natural loop
for 7 *dom* 10

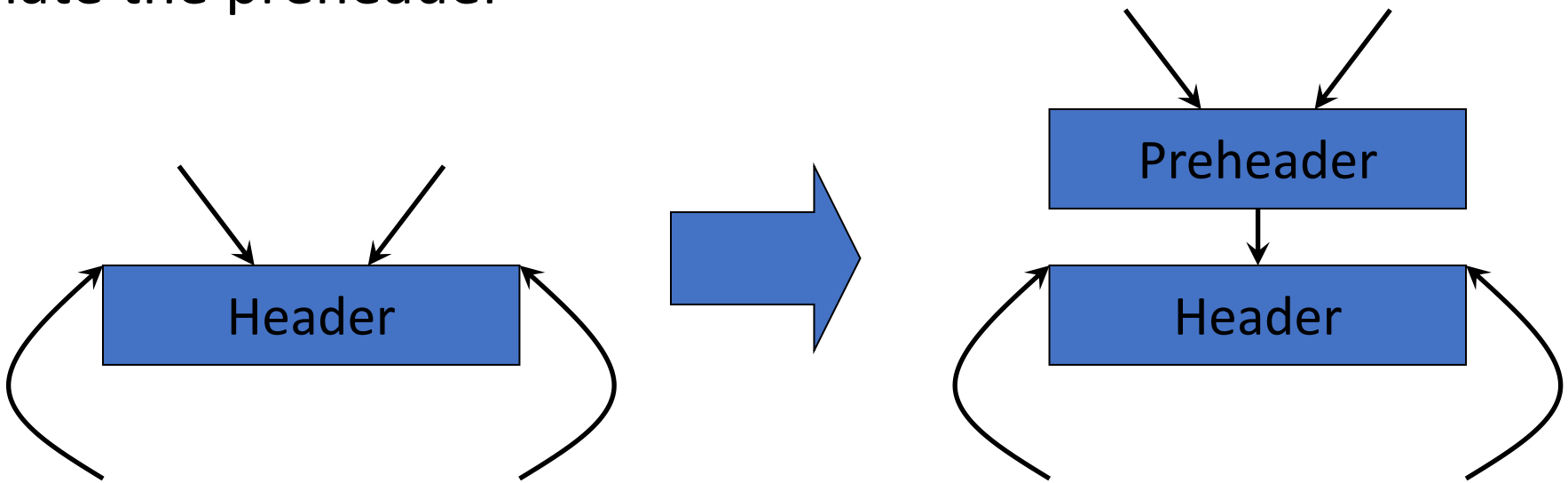
CFG



Dominator tree

Pre-Headers

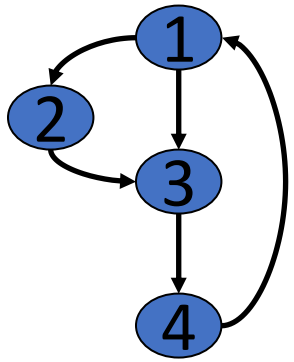
- To facilitate loop transformations, a compiler often adds a *preheader* to a loop
- Code motion, strength reduction, and other loop transformations populate the preheader



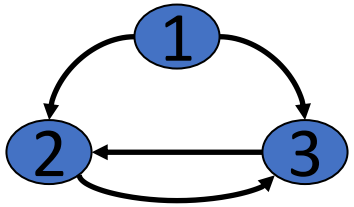
Reducible flow graphs

- Forward edges form an acyclic graph where every node can be reached from the initial node G
- Back edges consists of edges whose heads dominate their tails

Reducible flow graphs



Example of a reducible CFG



Example of a irreducible CFG

Structured Control flow graph

- CFG is well-structured and reducible iff all its loops are natural loops characterized by their back edges.
- In a well-structured control-flow graph there are no jumps into the middles of loops. Each loop is entered only through its header
- CFG derived from programs using structured flow-of-control statements such as if-then-else, while-do, continue, and break statements are always well-structured
- Many dataflow analysis algorithms work only on well-structured CFGs.

CFG Synthesis

- A Control Flow Graph (CFG) of some program p , named CFG(p), is a static abstraction of p, in which each node represents a Basic Block (B).
- Edges connecting the nodes in CFG represent the control flow from any one basic block to its successors
- A *CFG* only represents the static control flow
- Not mandatory to store, which of 2 successors in an *If Expression* is connected by the true condition.
- Just have to show 2 successors

The CFG Algorithm cfg_build(pc)

- Aside from its parameter pc, input to the CFG Algorithm cfg_build() is a list of instructions I broken into Basic Blocks. One of these blocks holds the select entry instruction at address: pc
- The CFG Algorithm creates a new cfg node for each basic block

CFG algorithm

- For each successor *s* of a Basic Block(n) cfg_build() installs a directed edge from Basic Block(n) to *s*.
- During this process each BB is labeled as *reached*. At completion, all BBs are inspected; those not *reached* are filtered out as *unreachable blocks*, hence each of its instructions is *unreachable code*.
- Output of cfg_build() is a pointer to the *cfg* node associated with address pc