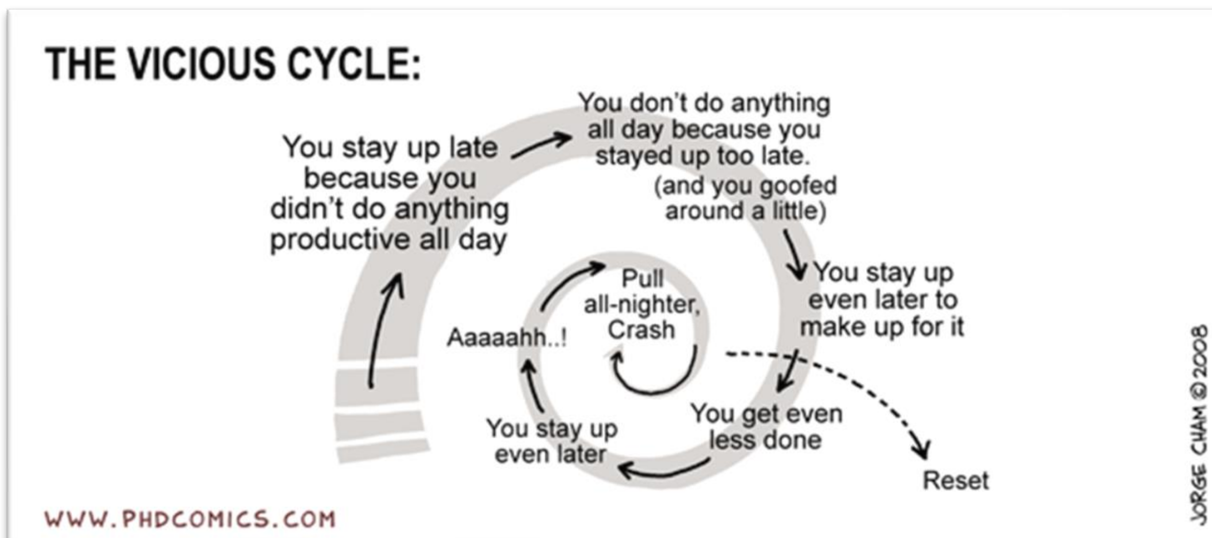
Compilers - Chapter 7:
Loop optimisations
Part 2: Dominators and natural loops

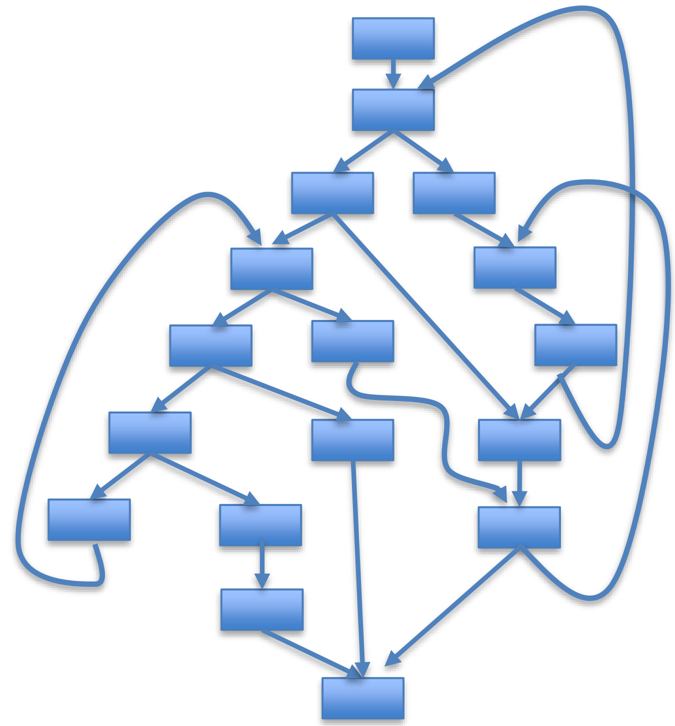
- Lecturer:
  - Paul Kelly (p.kelly@imperial.ac.uk)



http://phdcomics.com/comics/archive.php?comicid=1015, https://en.wikipedia.org/wiki/Piled_Higher_and_Deeper

Where should we move the loop-invariant instructions *to*?

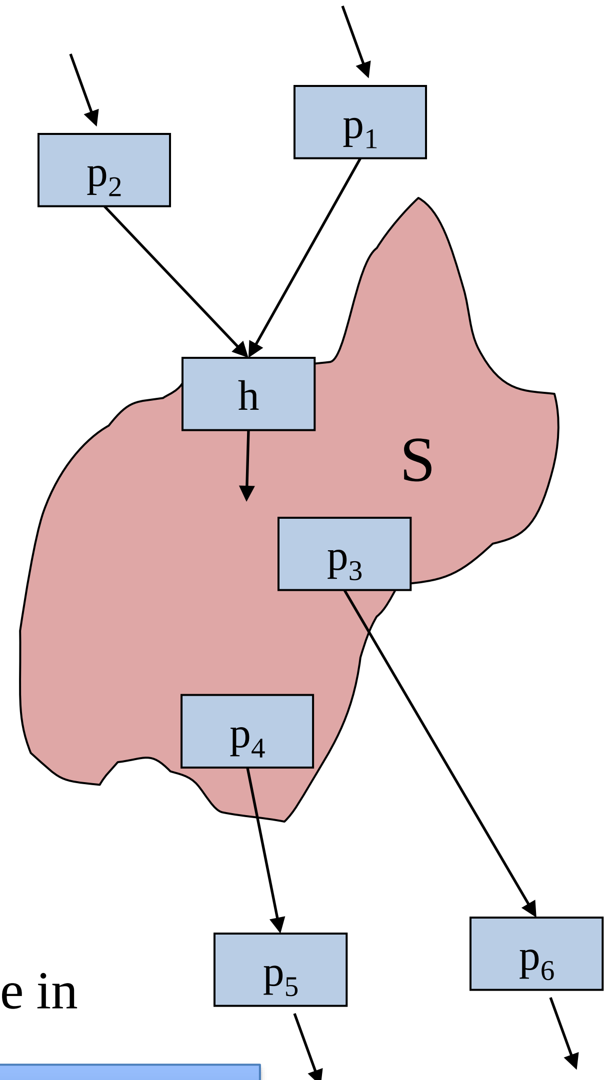
- Given control-flow graph, need to find
  - Where the loops are
  - Where the loop headers are
  - So we can find a place to put the loop's loop-invariant instructions
  - Need robust scheme that handles all loops including whatever you can do with goto



- We will develop a general framework for finding loops in control-flow graphs
  - We aim to *recover* the loop structure that came from the source program's looping constructs
  - We do not assume that the source code's structured control flow is preserved – so that we can combine different optimisations without having to track how the CFG was built

Where should we move the loop-invariant instructions *to*?

- Given control-flow graph, need to find
  - Where the loops are
  - Where the loop headers are
  - So we can find a place to put the loop's loop-invariant instructions
  - Need robust scheme that handles all loops including whatever you can do with goto



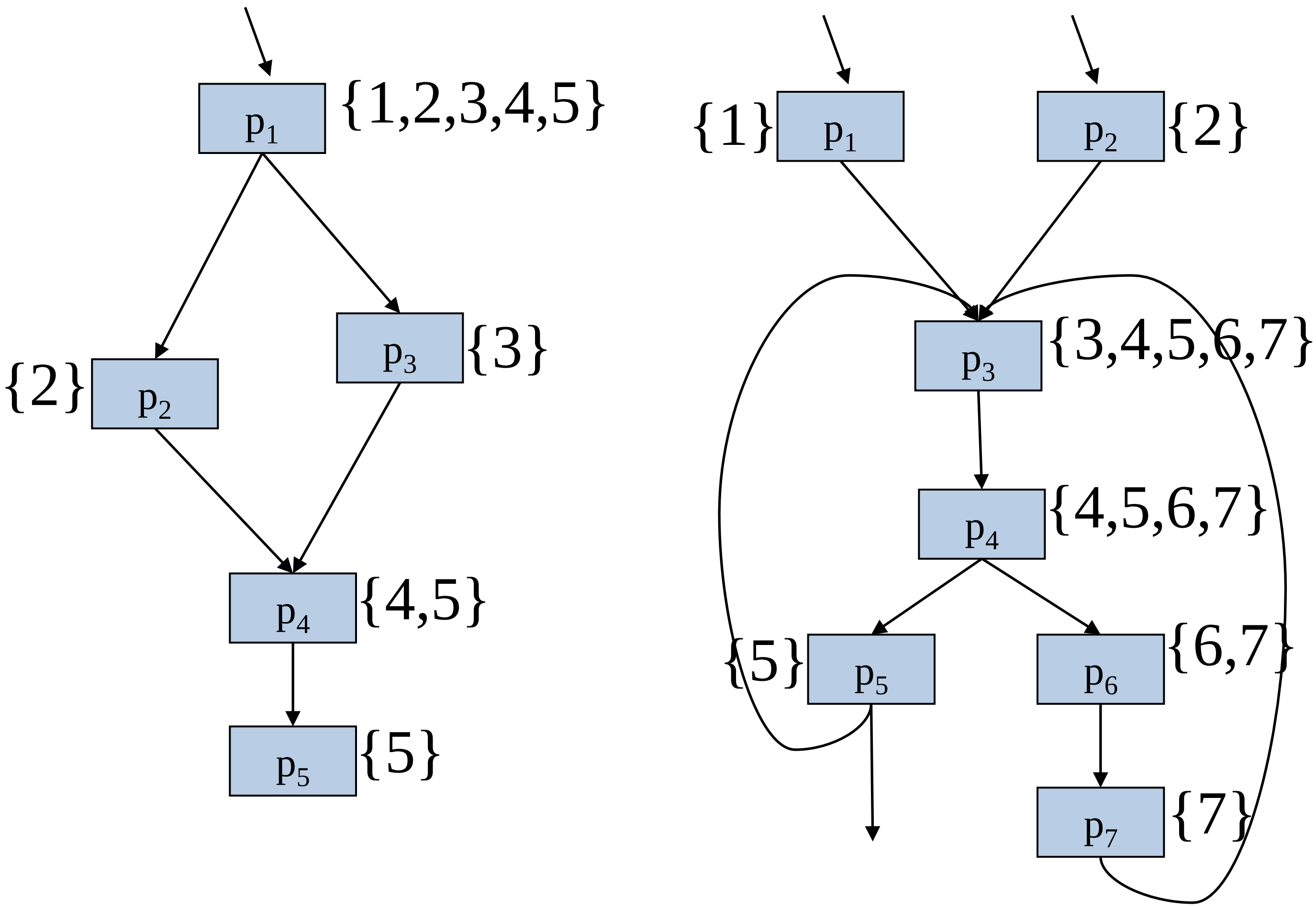
- Definition:

  A *loop* in a control flow graph is a set of nodes S including a *header* node h, with the following properties:
  - From any node in S there is a path leading to h
  - There is a path from h to any node in S
  - There is no edge from any node outside S to any node in S other than h

So there is only one way in!

- # Definition: dominator

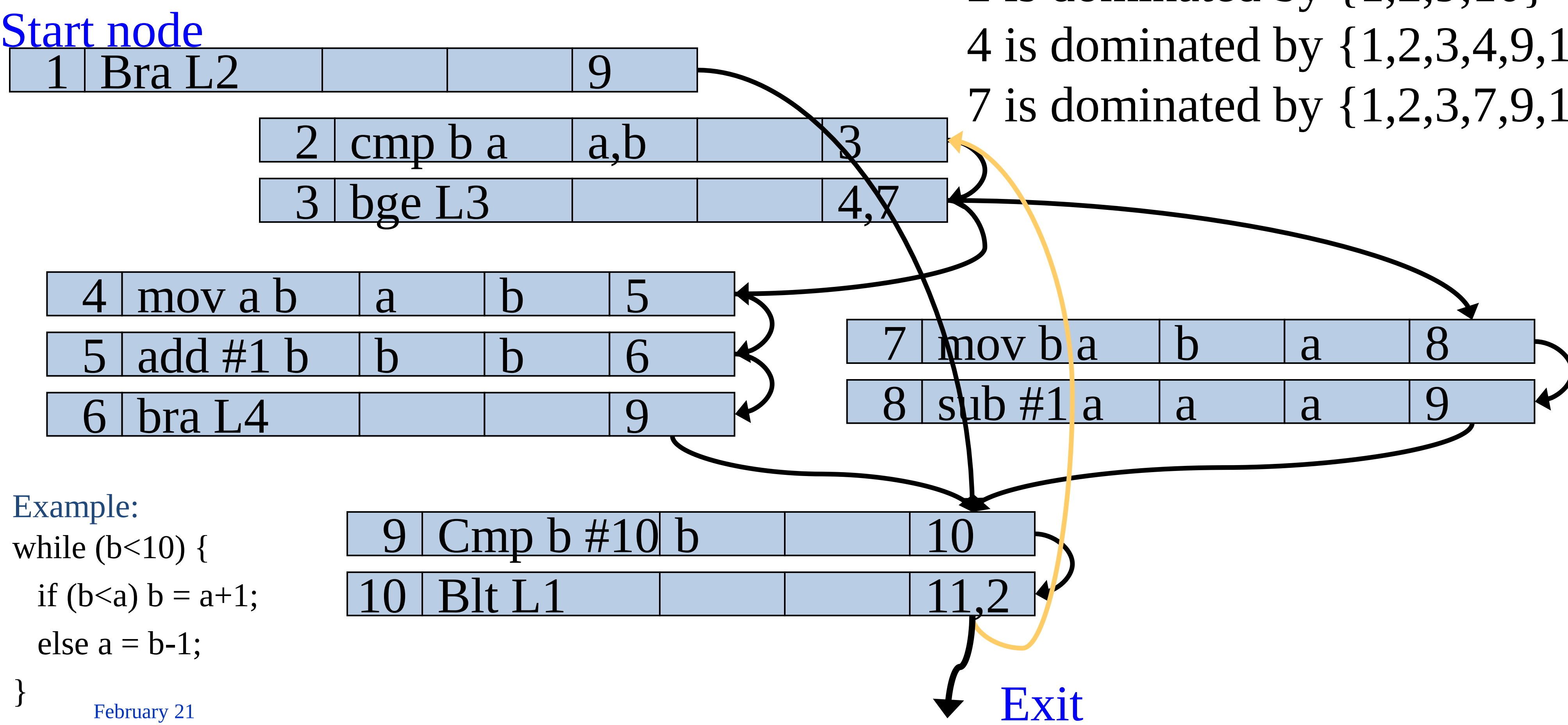
A node d *dominates* a node n if every path from the CFG's start node to n must go through d. Every node dominates itself



• Definition: dominator

A node d *dominates* a node n if every path from the CFG's start node to n must go through d. Every node dominates itself

1 is dominated by {1}
9 is dominated by {1,9}
2 is dominated by {1,2,9,10}
4 is dominated by {1,2,3,4,9,10}
7 is dominated by {1,2,3,7,9,10}

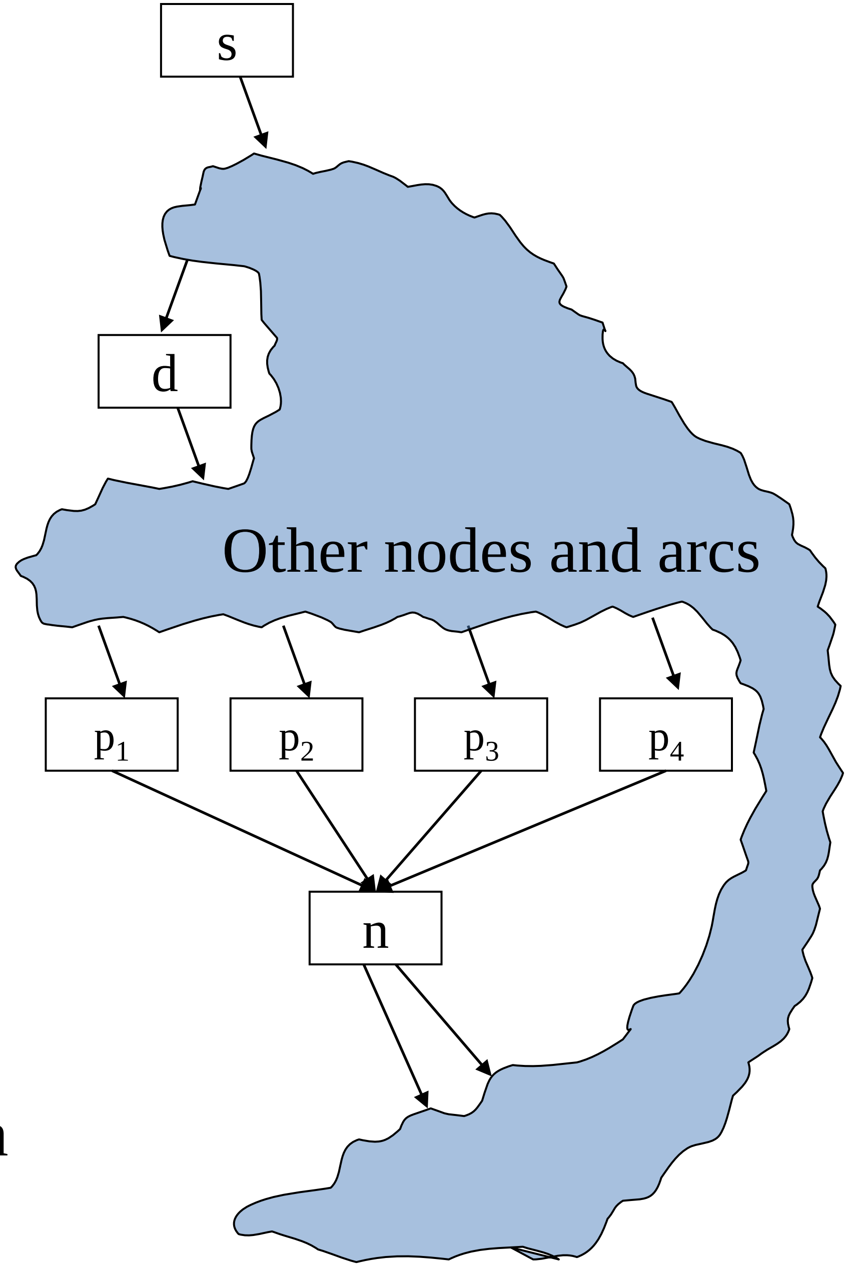


Example:

```
while (b<10) {
  if (b<a) b = a+1;
  else a = b-1;
}
```

Dominators…

- Finding the nodes dominated by a node d:
  - Consider another node n with predecessors $p_1 \ldots p_k$
  - If d dominates each one of the p_i then it must dominate n
  - Because:
    - Every path from the start node to n must go through one of the p_i
    - And every path from the start node to a p_i must go through d
  - Conversely,
    - If d dominates n, it must dominate all the p_i
    - Otherwise there would be a path from the start node to n going through the predecessor not dominated by d



Algorithm for finding dominators

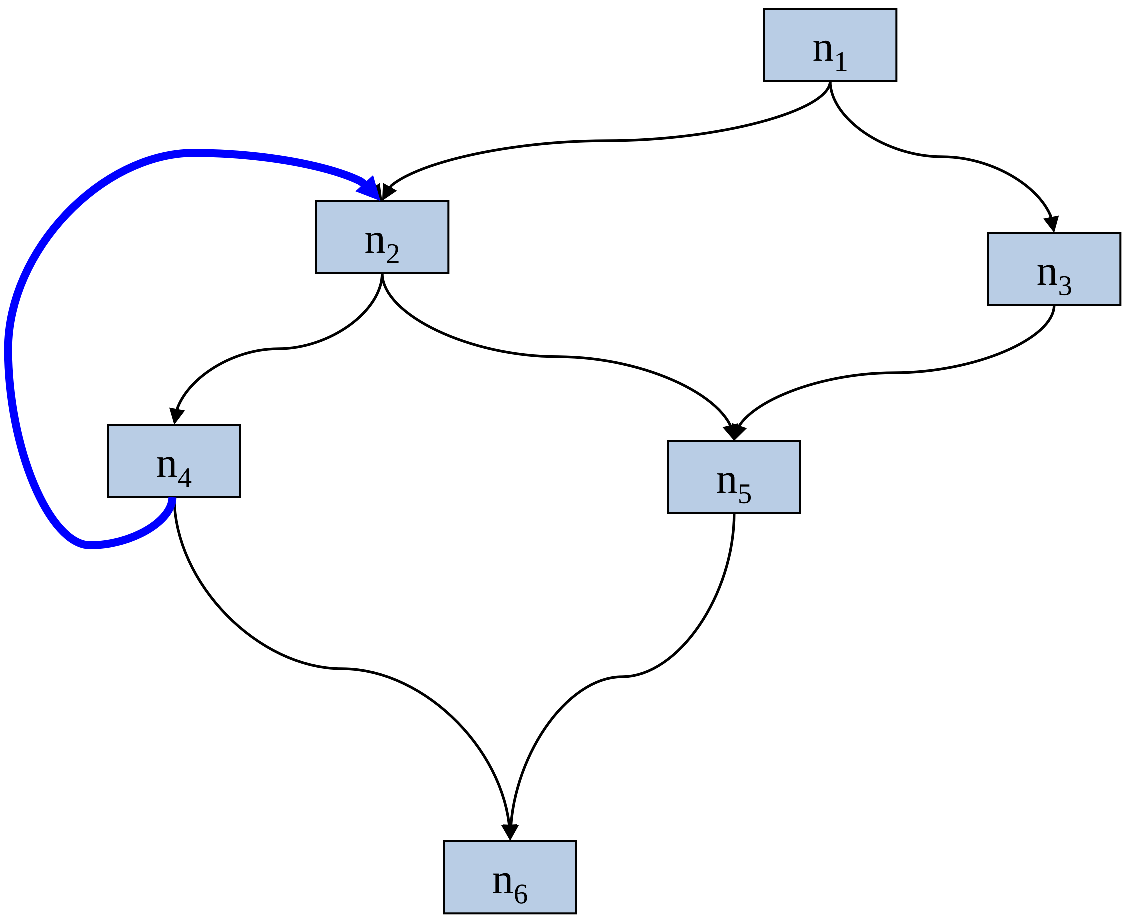
- Let Doms(n) be the set of nodes that dominate n

  ("n is dominated by Doms(n)")
- Construct a system of simultaneous set equations:
- Doms(s) = { s } *(s = start node)*
- $\text{Doms}(n) = \{n\} \cup \left(\bigcap_{p \in \text{preds}(n)} \text{Doms}(p) \right)$ *(otherwise)*

  ("which dominators are common to all our preds? ")
- Solve this system iteratively
- Initially, each Doms(n) starts as the set of all nodes in the graph
- Each assignment makes Doms(n) smaller, until it stops changing

Back edges

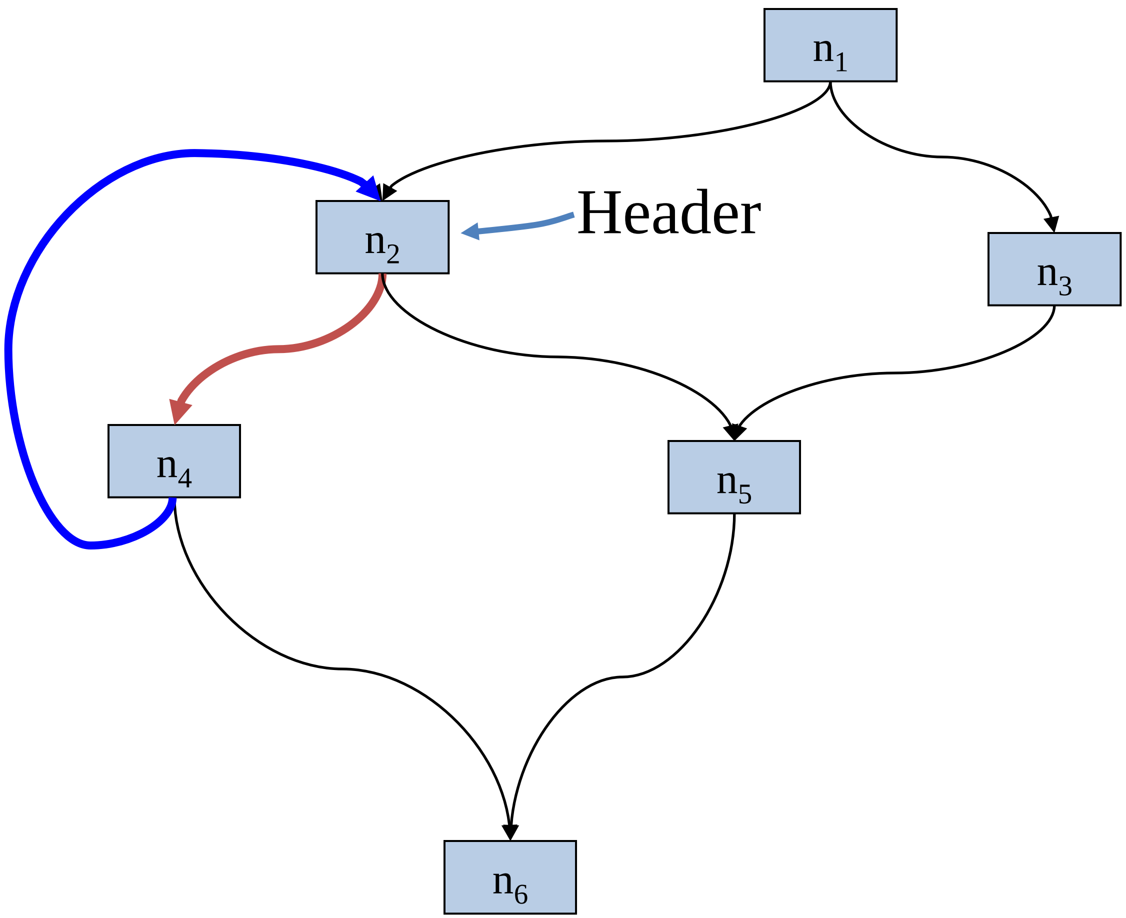
- A control flow graph edge from a node *n* to a node *h* that dominates *n* is called a ***back edge***.



n1 dominates all nodes
n2 dominates n2,n4
n3 dominates only n3
n4 dominates only n4
n5 dominates only n5
n6 dominates only n6

Back edges…

- For every back edge, there is a corresponding subgraph of the CFG that is a loop (by our definition earlier)



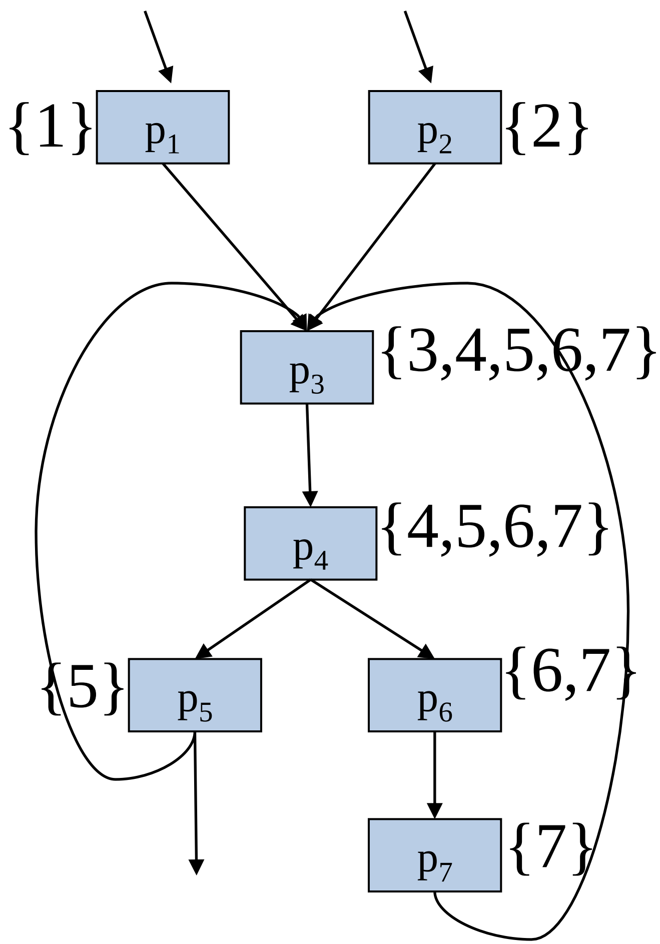
Definition:
The ***natural loop*** of a backedge (n,h), where h dominates n, is

- the set of nodes x such that h dominates x and
- there is a path from x to n not containing h.

The ***header*** of this loop will be h

Back edges…

- For every back edge, there is a corresponding subgraph of the CFG that is a loop (by our definition earlier)



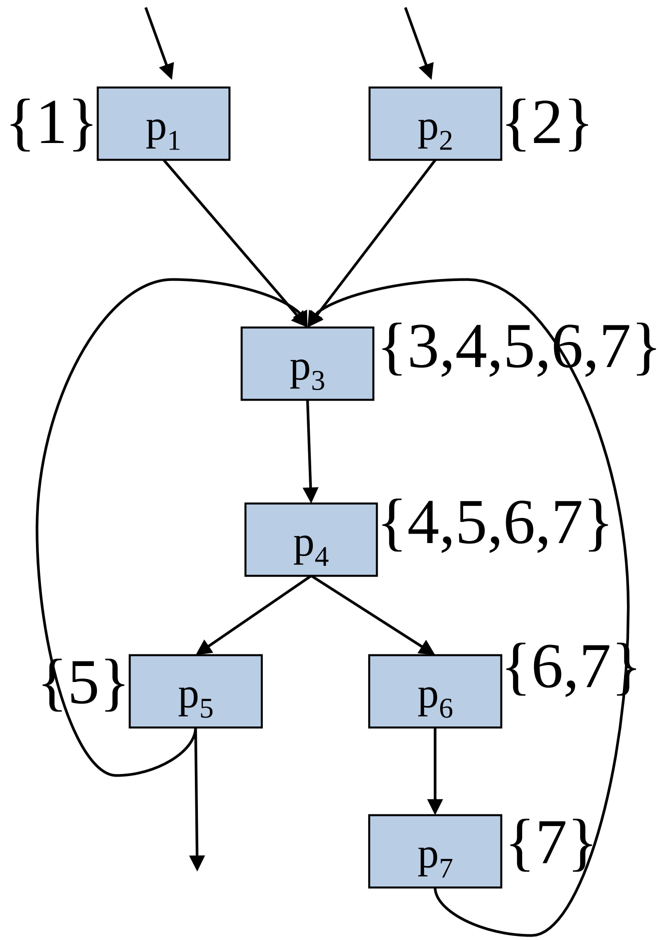
Definition:
The ***natural loop*** of a backedge (n,h), where h dominates n, is

- the set of nodes x such that h dominates x and
- there is a path from x to n not containing h.

The ***header*** of this loop will be h

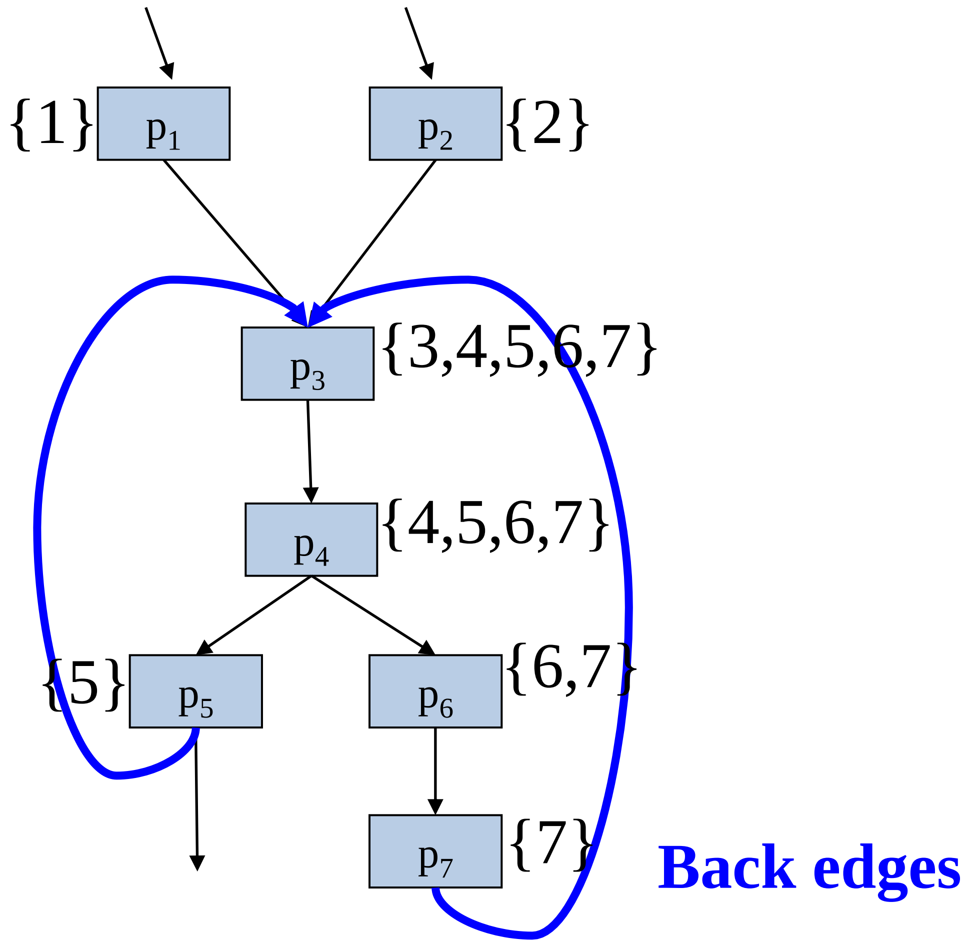
Multiple loops

- It is possible for two loops to share the same header
- This example has two back edges, (5,3) and (7,3)
- In many cases these two natural loops arise from one source-code loop



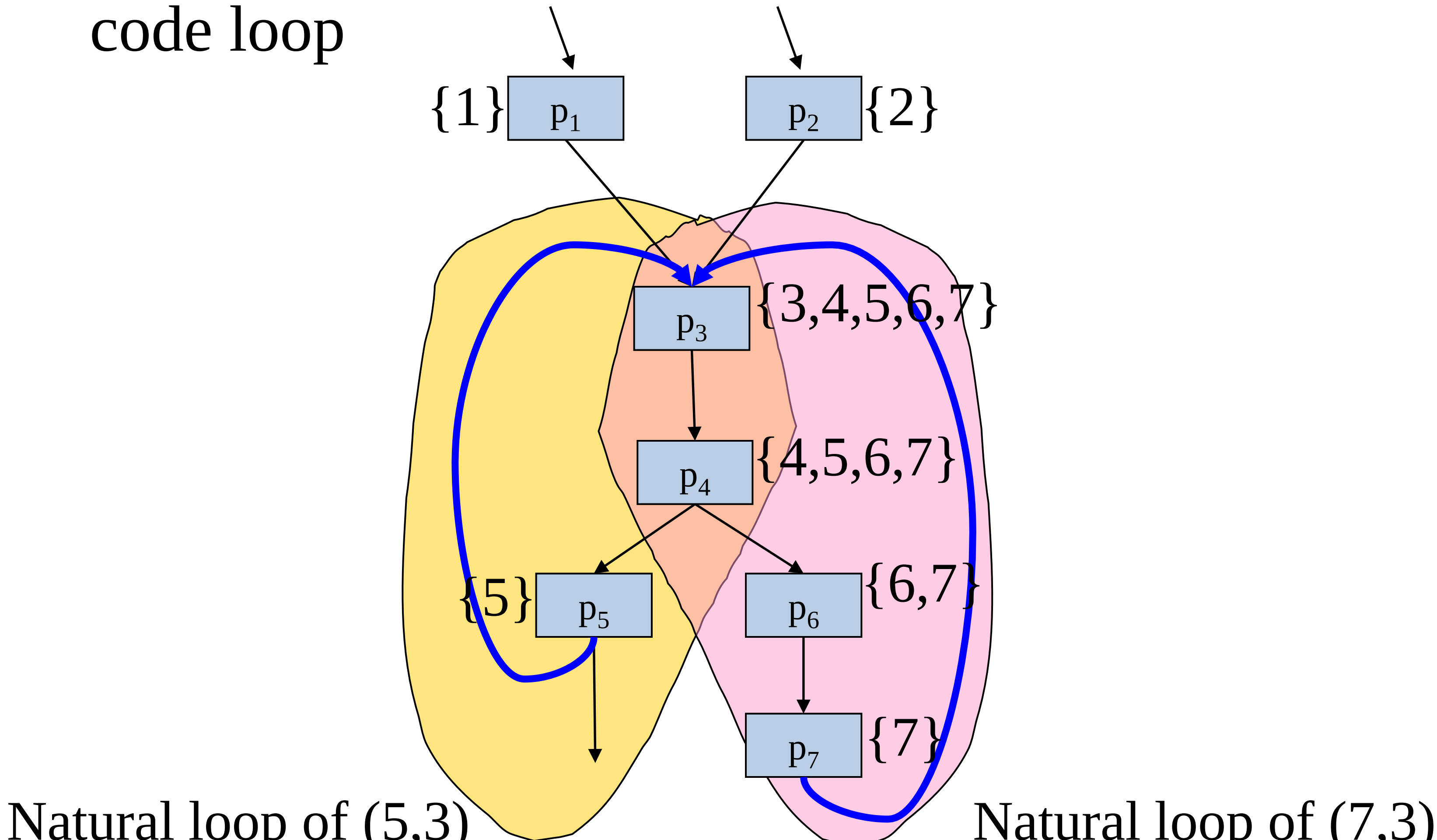
Multiple loops

- It is possible for two loops to share the same header
- This example has two back edges, (5,3) and (7,3)
- In many cases these two natural loops arise from one source-code loop



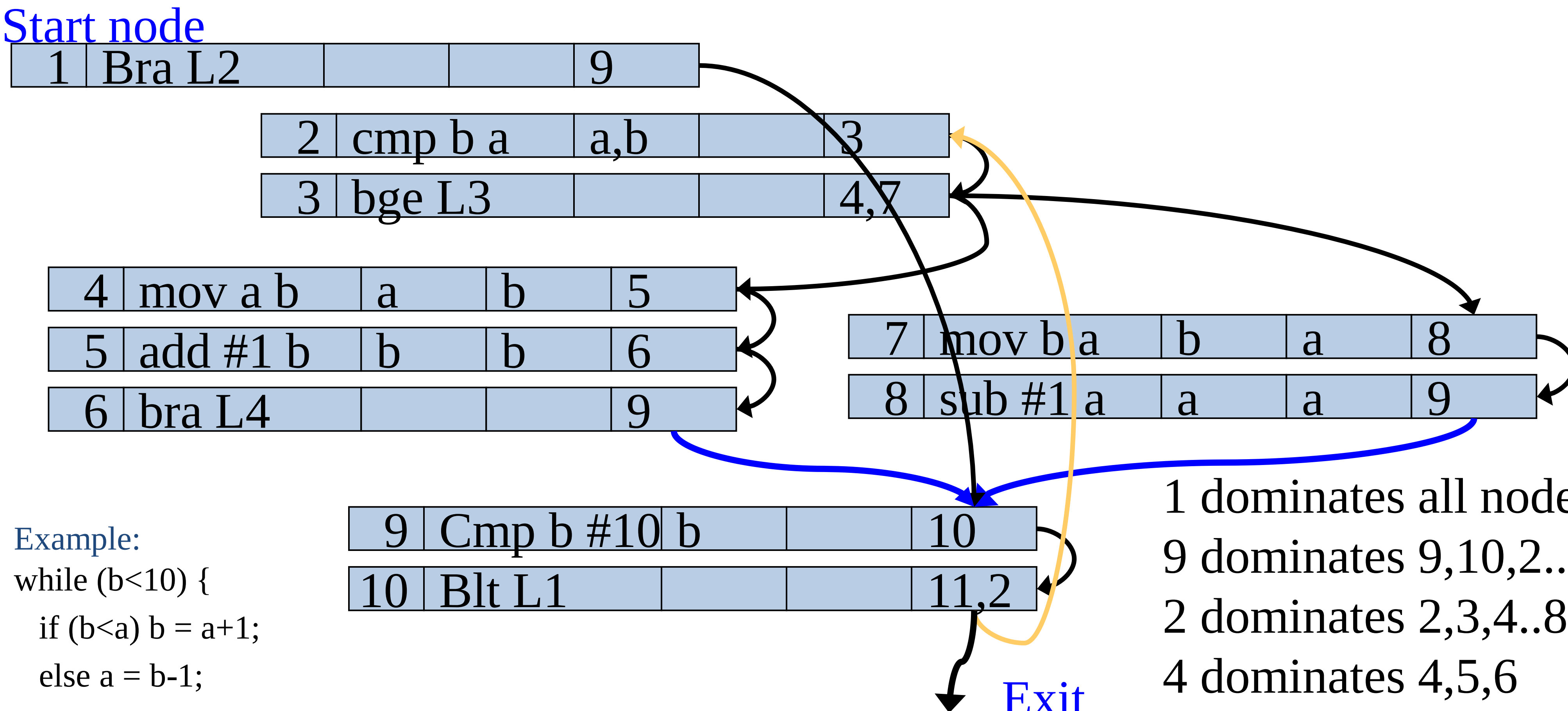
Multiple loops

- It is possible for two loops to share the same header
- This example has two back edges, (5,3) and (7,3)
- In many cases these two natural loops arise from one source-code loop



Multiple loops

- It is possible for two loops to share the same header
- This example has two back edges, (6,9) and (8,9)
- E.g. here two natural loops arise from one source-code loop



1 dominates all nodes
9 dominates 9,10,2..8
2 dominates 2,3,4..8
4 dominates 4,5,6
7 dominates 7,8

Example:

```
while (b<10) {
  if (b<a) b = a+1;
  else a = b-1;
}
```

Two natural loops sharing the same header

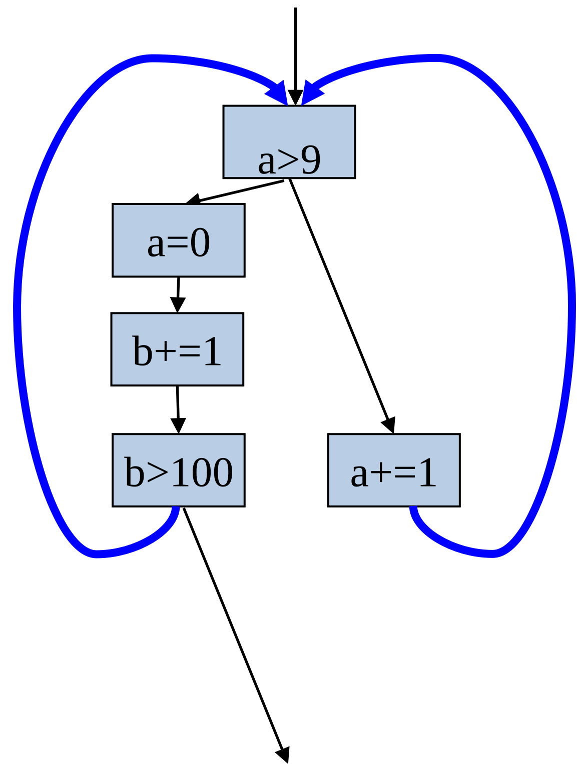
- Consider these two code fragments:

One loop:

```
while true {
  if (a<10) {
    a += 1;
  } else {
    a = 0;
    b += 1;
    if (b>100) break;
  }
}
```

Two loops:

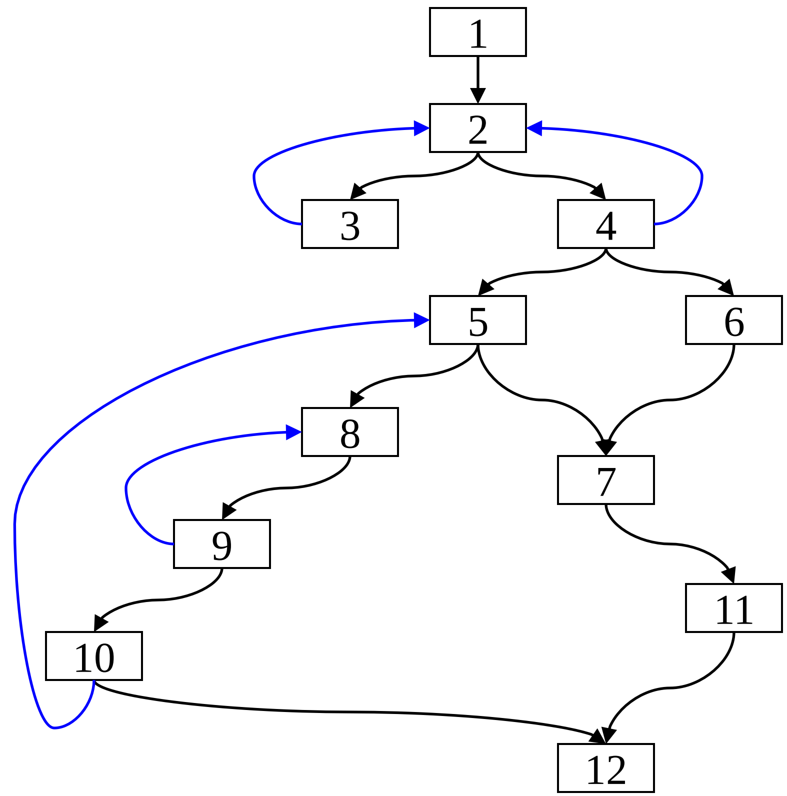
```
do {
  do {
    if (a>9) break;
    a += 1;
  } while true;
  a = 0;
  b += 1;
  if (b>100) break;
} while true;
```



- Conclusion: we can’t always distinguish exactly what the source code’s structured control flow was

Nested loops

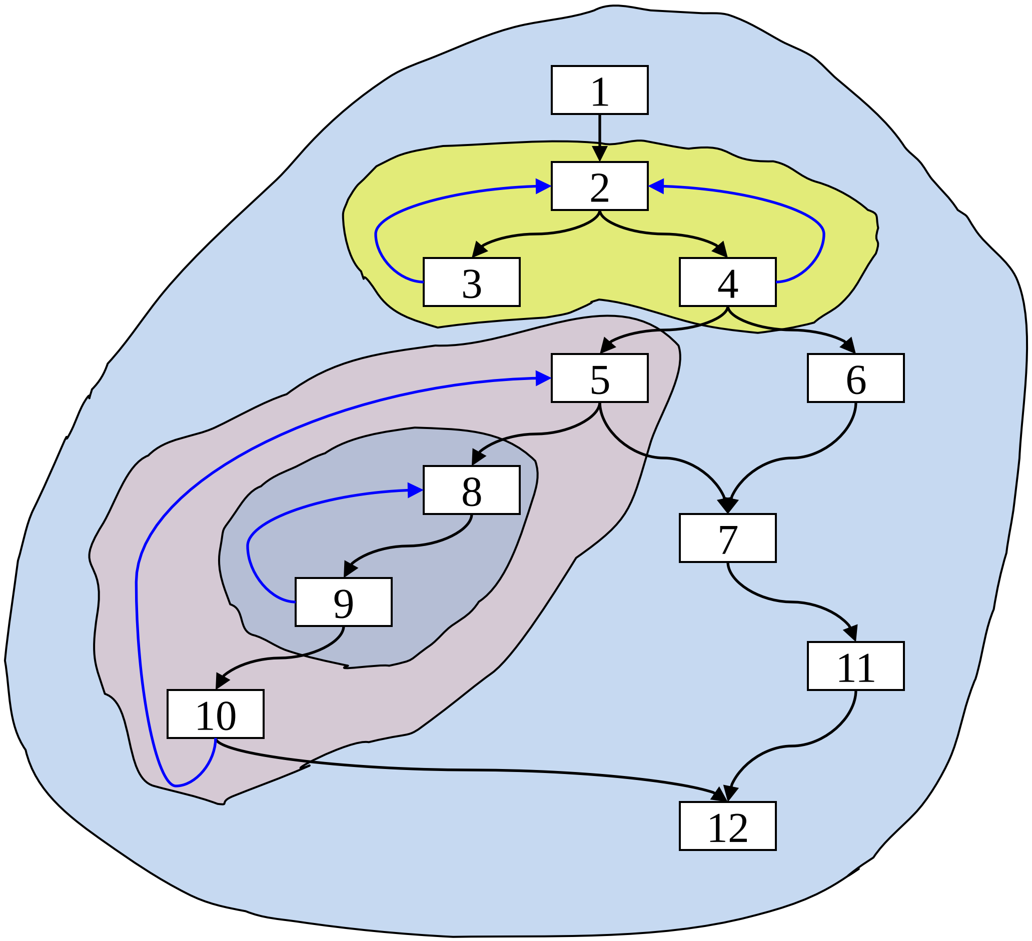
- Suppose:
  - A and B are loops with headers a and b, such that a ≠ b, and b is in A
- Then
  - The nodes of B must be a proper subset of the nodes of A
  - We say that loop B is nested within A
  - B is the inner loop



Back edges: (3,2), (4,2), (10,5), (9,8)

Nested loops

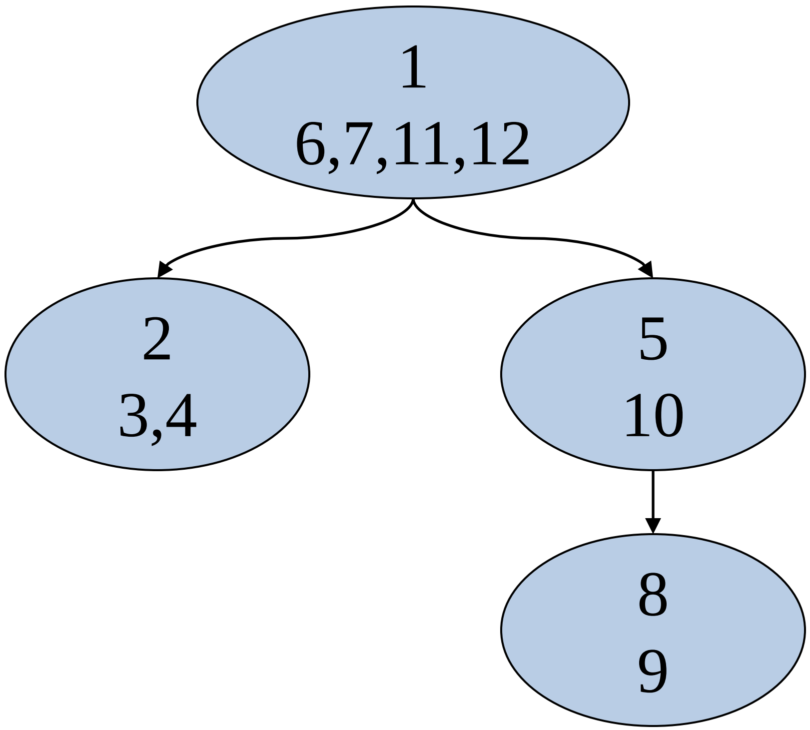
- Suppose:
  - A and B are loops with headers a and b, such that a ≠ b, and b is in A
- Then
  - The nodes of B must be a proper subset of the nodes of A
  - We say that loop B is nested within A
  - B is the inner loop

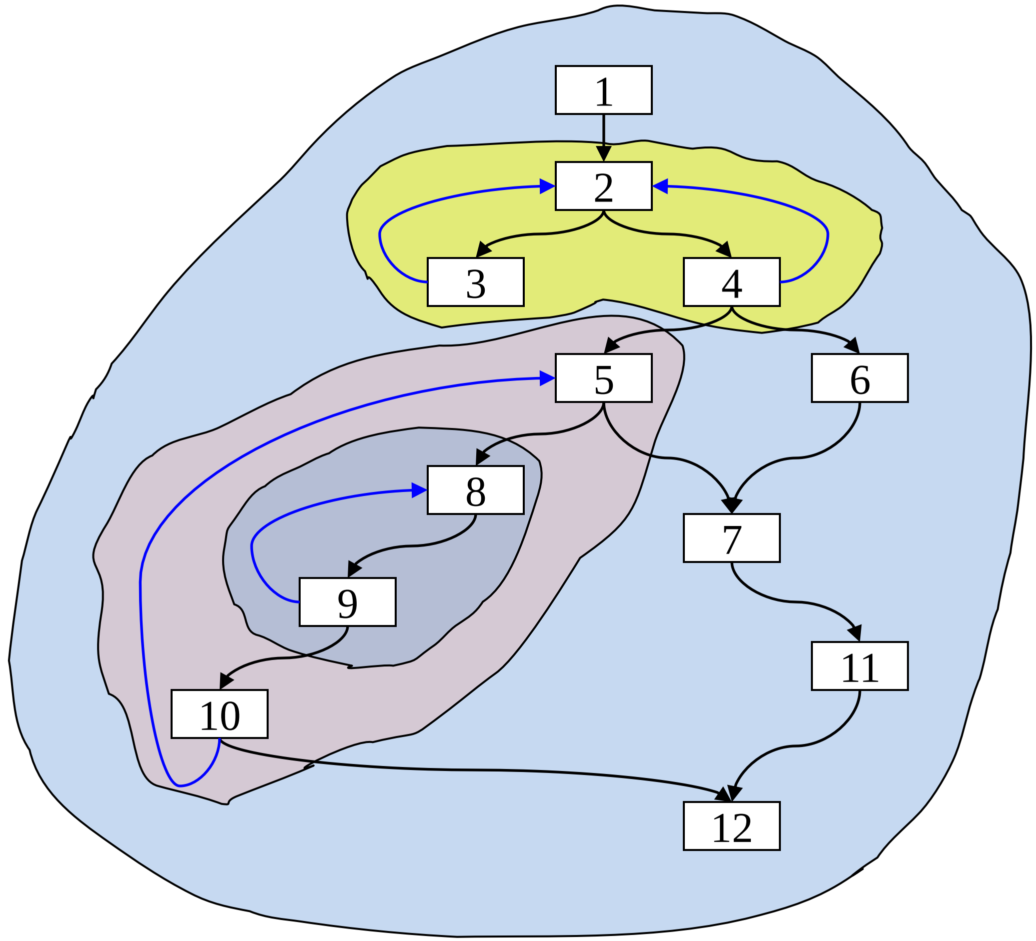


Back edges: (3,2), (4,2), (10,5), (9,8)

The Control Tree

- Loops form a tree
- Example:



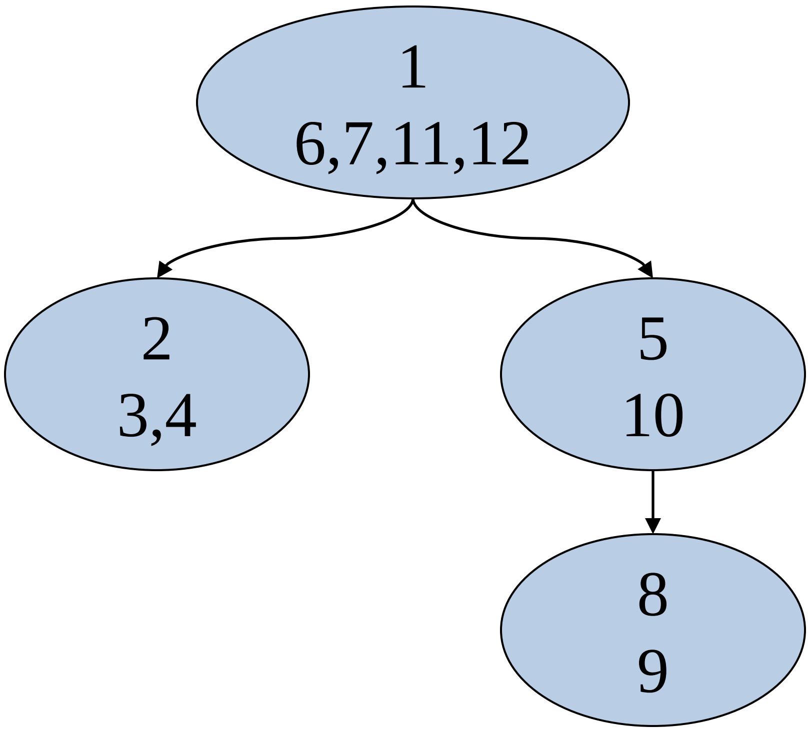


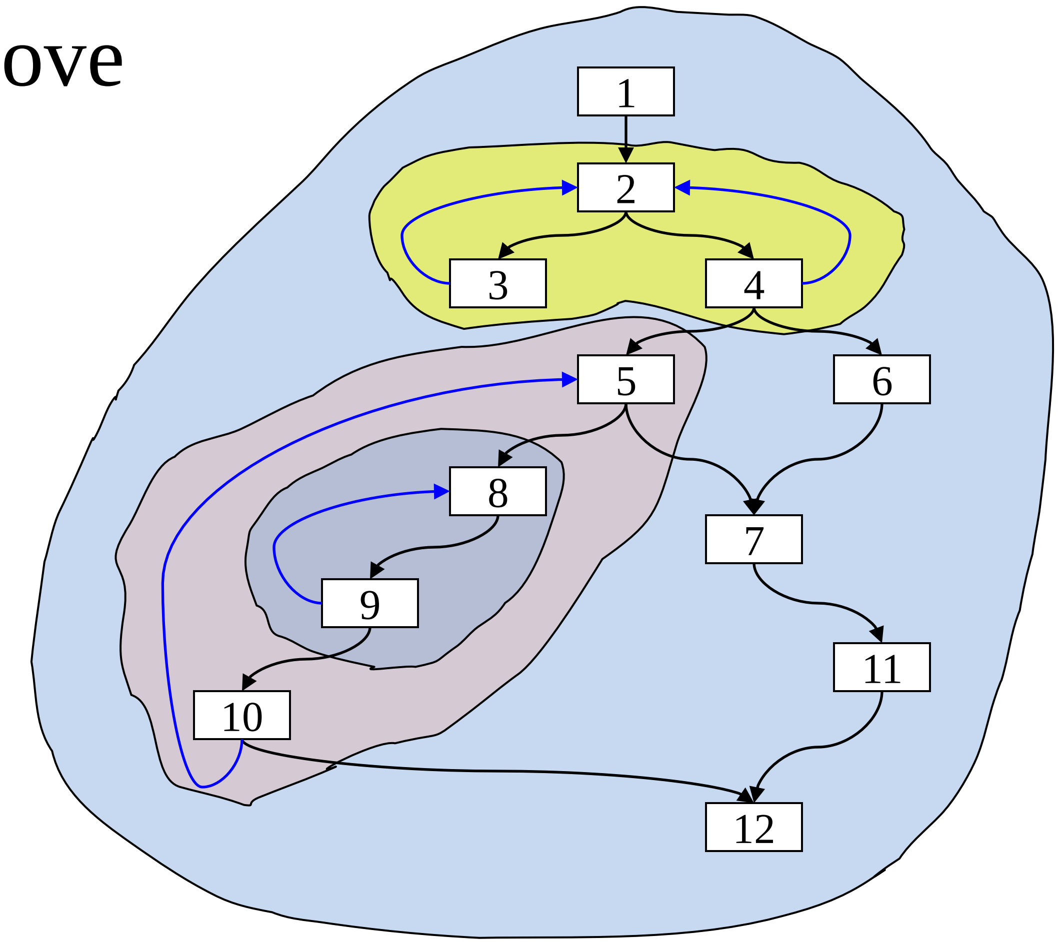
Back edges: (3,2), (4,2), (10,5), (9,8)

We have reconstructed the "structured control flow" from the control flow graph

Pre-headers

- Where should we move the loop-invariant instructions *to*?

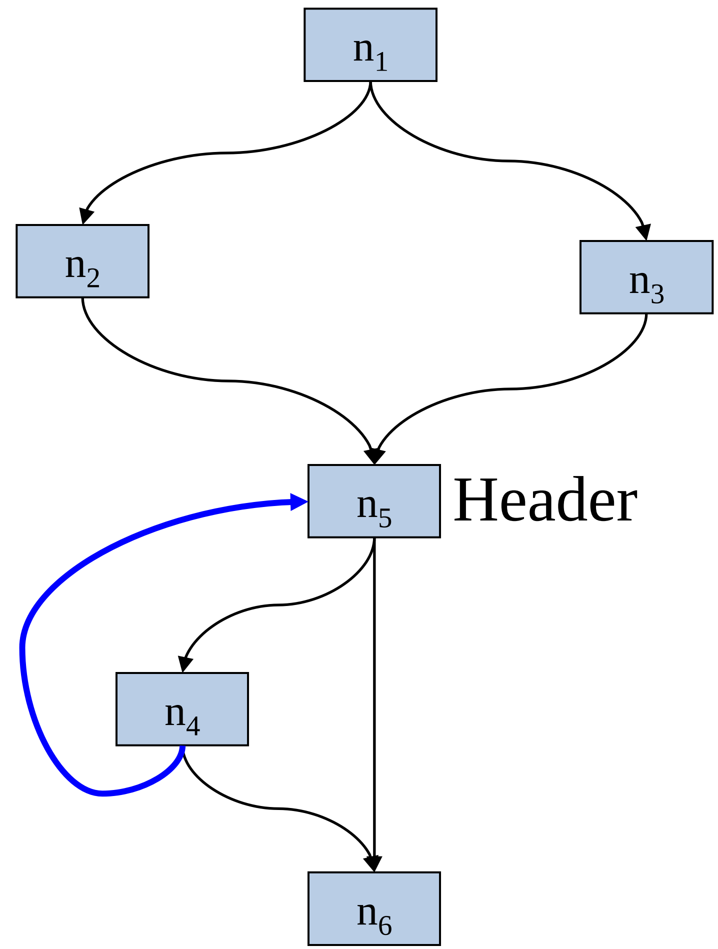




- We can't move them to the header
- We want to move them to the node preceding the header

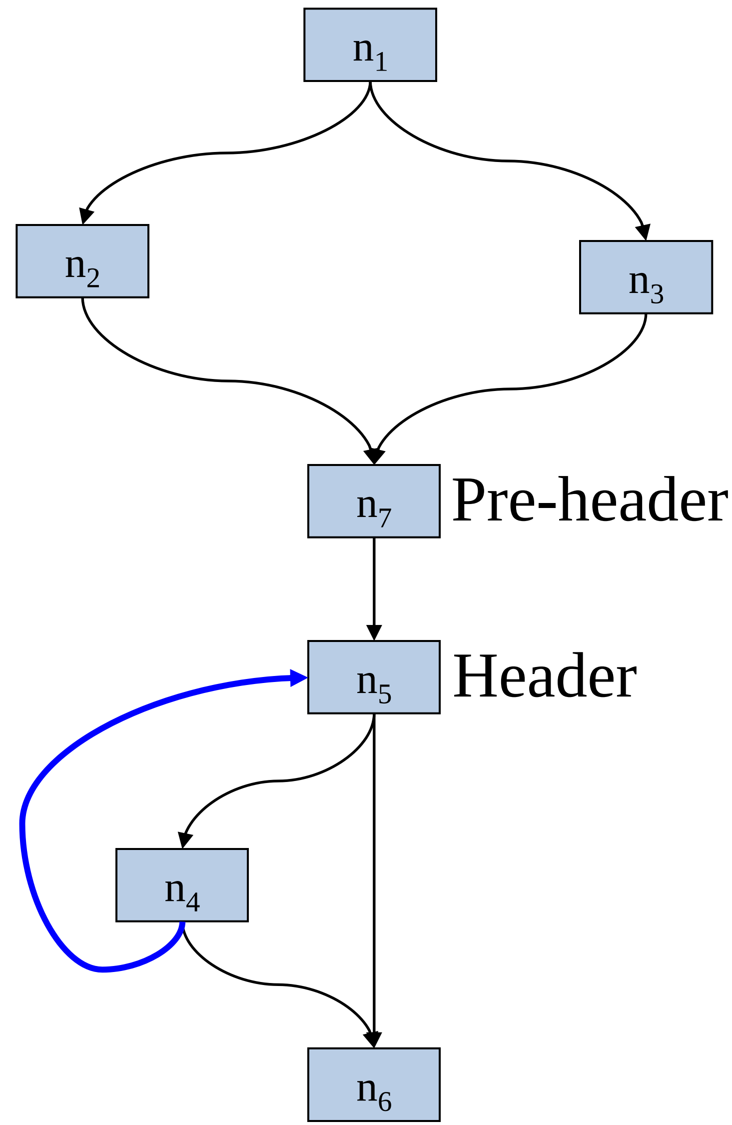
Pre-headers

- Where should we move the loop-invariant instructions *to*?
- We want to move them to the node preceding the header
- But sometimes the header has multiple predecessors
- What shall we do?



Pre-headers

- Where should we move the loop-invariant instructions *to*?
- We want to move them to the node preceding the header
- But sometimes the header has multiple predecessors
- What shall we do?
  - **Insert a pre-header**



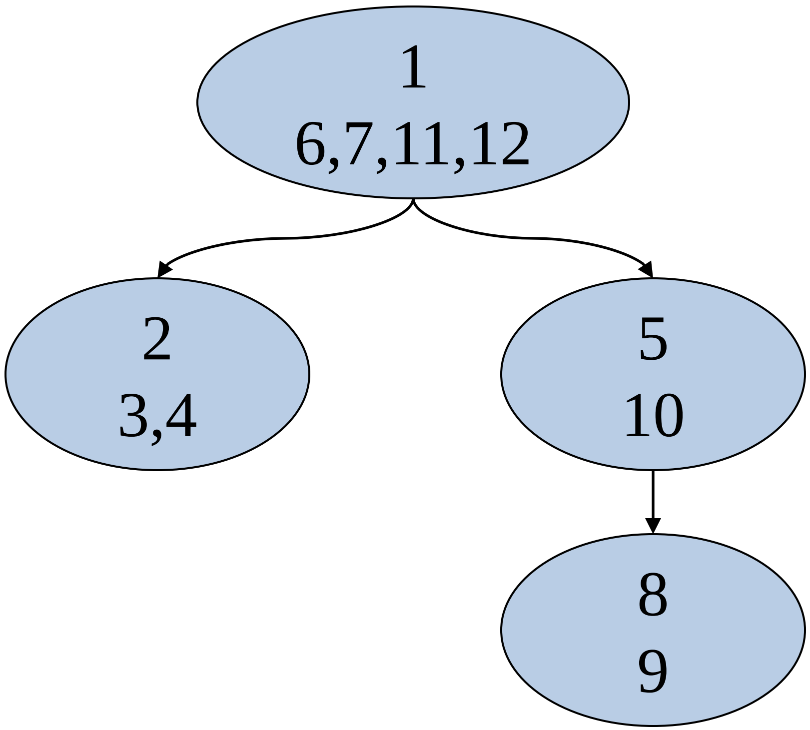
Summary

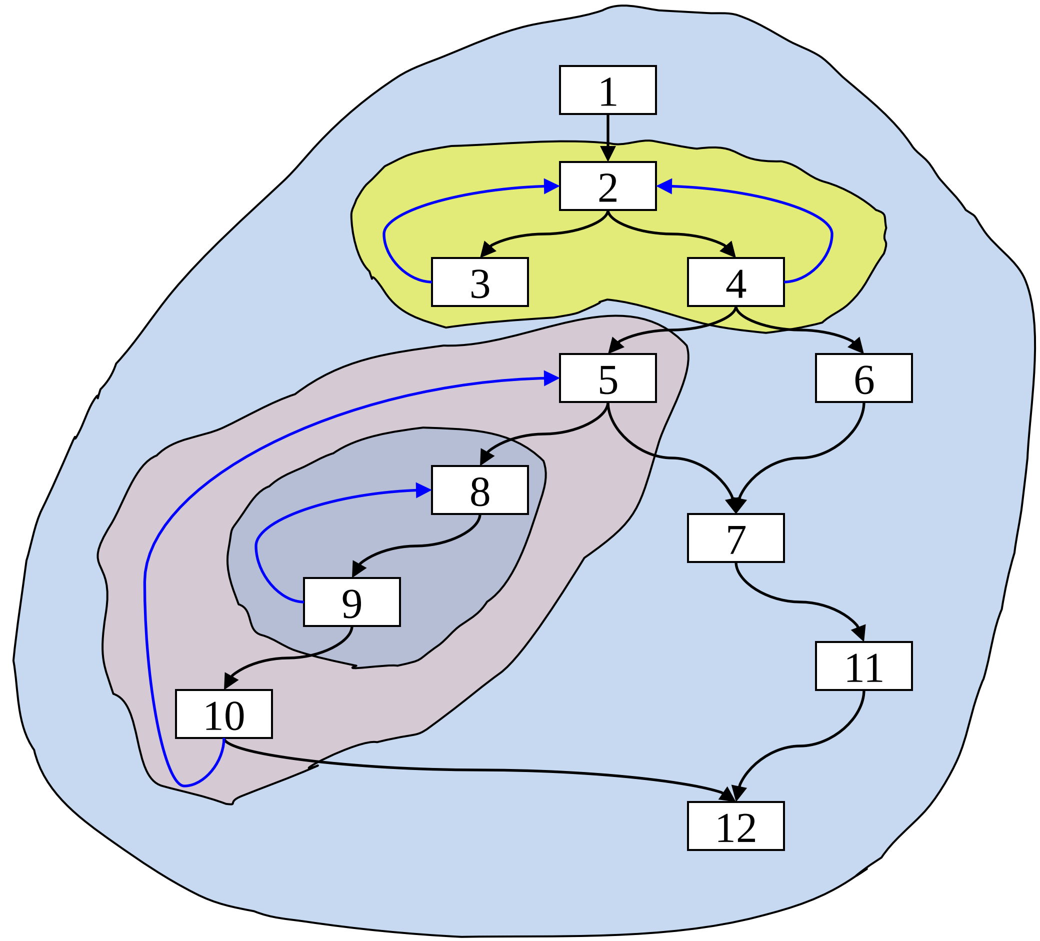
- *Dominators*
- Iterative data-flow algorithm for finding dominators
- There is a natural loop for each *back edge*
- *Natural loops*, *loop header*
  - A natural loop has just one entry path, through its *header*
  - (contrast: a natural loop is a strongly-connected region, but there are strongly-connected regions that are not natural loops)
- Natural loops that share the same header have ambiguous source-code structured control flow
- Natural loops with different headers form a loop tree
- We insert a pre-header before the header, to ensure a unique place to move loop-invariant instructions to

Piazza question:
"are 5&10 parents of 8&9?"

The Control Tree

- Loops form a tree
- Example:





Back edges: (3,2), (4,2), (10,5), (9,8)

The root node of the loop tree has seven children - two of them are loops themselves (shown in green and purple), and five of them are non-loop statements (1,6,7,11,12). The purple subloop has two non-loop children (5,10) and one loop child (in blue). That child has two non-loop children (8,9).

Feeding curiosity

- **Reducible control-flow graphs**: structured control-flow programs (goto-free) result in CFGs whose only cycles are natural loops. In particular, you can't make a loop with more than one entry path. Reducibility is a rich property that can be defined and tested in multiple ways ; see:
  - Matthew S. Hecht and Jeffrey D. Ullman. 1972. **Flow graph reducibility**. *STOC '72* https://doi.org/10.1145/800152.804919
- **Interval analysis and structural analysis**: dataflow analysis can be solved using non-iterative methods by finding the loop nesting structure – potentially leading to faster algorithms (and better behaviour with incremental updates). At least for reducible CFGs. See for example
  - M. Sharir. 1980. **Structural analysis: A new approach to flow analysis in optimizing compilers**. *Comput. Lang. 5, 3–4* (January, 1980) https://doi.org/10.1016/0096-0551(80)90007-7
- But pretty much everyone uses iterative algorithms!

Feeding curiosity

- **Reverse engineering**: recovering the source code from the binary is clearly an interesting problem – with applications from cracking license-protected software products, to reverse-engineering malware.
- **Code obfuscation:** naturally one might try to modify code to make reverse-engineering hard. Many cunning approaches exist. But they come with no guarantees (cf cryptography where we might prove that decryption is hard)
- Is it possible to have any assurance that reverse-engineering executable software is *actually* hard?
- **Impossibility:** See:
  - Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. **On the (im)possibility of obfuscating programs**. *J. ACM 59, 2, Article 6 (April 2012),* https://doi.org/10.1145/2160158.2160159
- **Possibility**: See:
  - Boaz Barak. 2016. **Hopes, fears, and software obfuscation**. *Commun. ACM 59, 3 (March 2016),* 88–96. https://doi.org/10.1145/2757276