Compilers I - Chapter 7: Loop optimisations

- Lecturers:
  - Part I: Paul Kelly (phjk@doc.ic.ac.uk)
    - Office: room 304 William Penney Building
  - Part II: Naranker Dulay (nd@doc.ic.ac.uk)
    - Office: room 562
- Materials:
  - Textbook
  - Course web pages (http://www.doc.ic.ac.uk/~phjk/Compilers)
  - Piazza (http://piazza.com/imperial.ac.uk/fall2016/221)

The plan

- To optimise or not to optimise?
- High-level vs low-level; role of analysis
- Peephole optimisation
- Local, global, interprocedural
- Loop optimisations
- Where optimisation fits in the compiler
- Example: live ranges
- Live ranges as a data flow problem
- Solving the data-flow equations
- Deriving the interference graph
- Loop-invariant code and code motion optimisations
- Other data-flow analyses
- More sophisticated optimisations

Loop-invariant code motion

- Definition:
  - An instruction is loop-invariant if its operands can only arrive from outside the loop
- Objective:
  - move (“hoist”) loop-invariant instructions out of loop
- Issues:
  - Where should we move the loop-invariant instructions *to*?
  - How can we find out whether operands only arrive from outside loop
  - Other pitfalls…

Finding loop-invariant instructions

- A CFG node is a definition if it updates a temporary
- In our CFG, an instruction can update at most one temporary, *t*
- Each definition node is labelled with the Node id, *d:*

  d :   t := u1 ⊕ u2

  Or simply

  d :   t := u1   or d :   t := constant

  (where u1 and u2 are given by the Node' s "uses" field)
- This definition is loop-invariant if, for each $u_i \in$ uses(*d*),
  - All the definitions of u_i that reach *d* are outside the loop
  - Or only one definition of u_i reaches *d*, and that definition is loop invariant

Finding reaching definitions

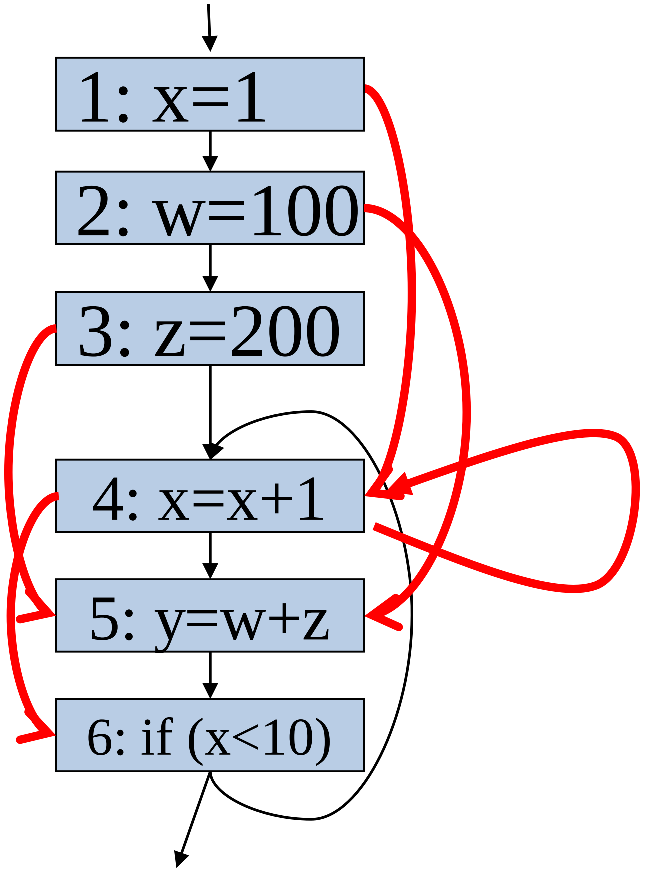
- A definition of variable *t* is a statement which may assign to *t*
- A definition *d* **reaches** a program point *p* if there exists a path from *d* to *p* such that *d* is not killed along that path
- Consider a CFG node

  n: *t* := u1 ⊕ u2 (defs(*n*)={*t*}, uses(*n*) = {u1,u2})

Define:

- **Gen(*n*)** is the set of definitions generated by node *n*, *i.e.* {*n*}
- **Kill(*n*)** is the set of all definitions of *t*, excluding *n*
- **ReachIn(*n*)** is the set of definitions reaching the point before *n*
- **ReachOut(*n*)** is the set of definitions reaching the point after *n*

Reaching definitions



- Reaching definitions link each use of a variable back to where its value was generated
- Loops and conditionals lead to multiple reaching definitions

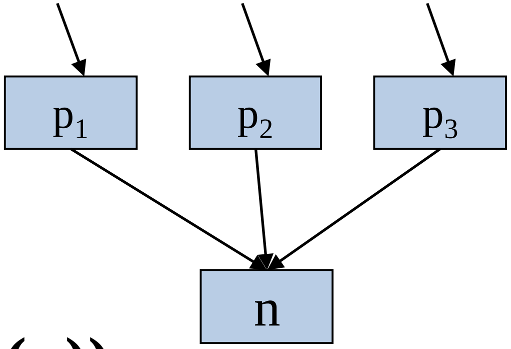
- Gen(4) = {4}
- Gen(5) = {5}
- Gen(6) = {}

- kill(4) = {1}
- kill(5) = {}

- ((In the worst case, the number of reaching definitions could be quite large))

Reaching definitions – another data flow analysis

- Dataflow equations:



$$\mathbf{ReachIn}(n) = \bigcup_{p \in \mathrm{pred}(n)} \mathbf{ReachOut}(\boldsymbol{p})$$

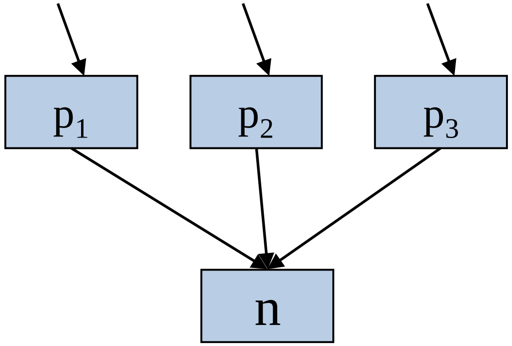
$$\mathbf{ReachOut}(\boldsymbol{n}) = \mathbf{Gen}(\boldsymbol{n}) \cup (\mathbf{ReachIn}(\boldsymbol{n}) - \mathbf{Kill}(\boldsymbol{n}))$$

(“The Gen(n) + whatever survives”)

- Many dataflow problems have “gen” and “kill”
- In the case of ReachOut(n), gen(n) is usually just its own id, {n}
- But if node n defines no value (eg it’s a jump), it will never reach anything – so gen(n) = {}

Reaching definitions – another data flow analysis

- Dataflow equations:



$$\textbf{ReachIn}(n) = \bigcup_{p \in \mathrm{pred}(n)} \textbf{ReachOut}(\boldsymbol{p})$$

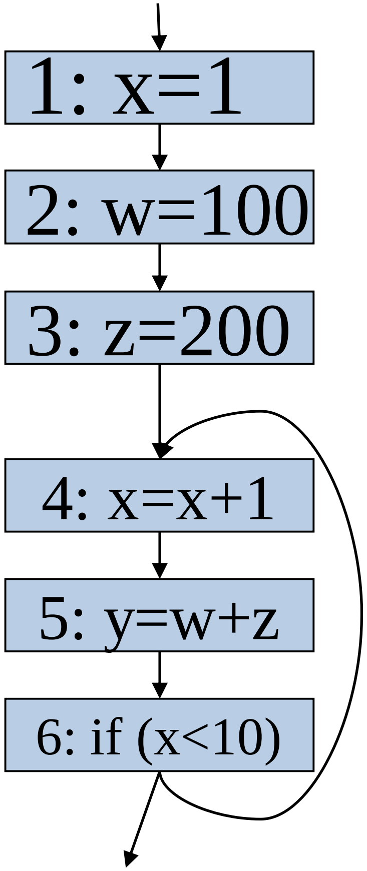
$$\textbf{ReachOut}(\boldsymbol{n}) = \textbf{Gen}(\boldsymbol{n}) \cup (\mathrm{R}\textbf{eachIn}(\boldsymbol{n}) – \textbf{Kill}(\boldsymbol{n}))$$

- Solve in the usual way: *("The Gen(n) + whatever survives")*
  - Initialise **ReachIn**(*n*) and **ReachOut**(*n*) to { }
  - Iterate, repeatedly updating **ReachIn**(*n*) and **ReachOut**(*n*) using definitions above
  - Until convergence
  - At each step, the sets increase in size

Use reaching definitions to find loop invariant instructions

- Find the set of definitions of variables used by this node
- An instruction is loop invariant if the definitions of all the values it uses are outside the loop
- Example:

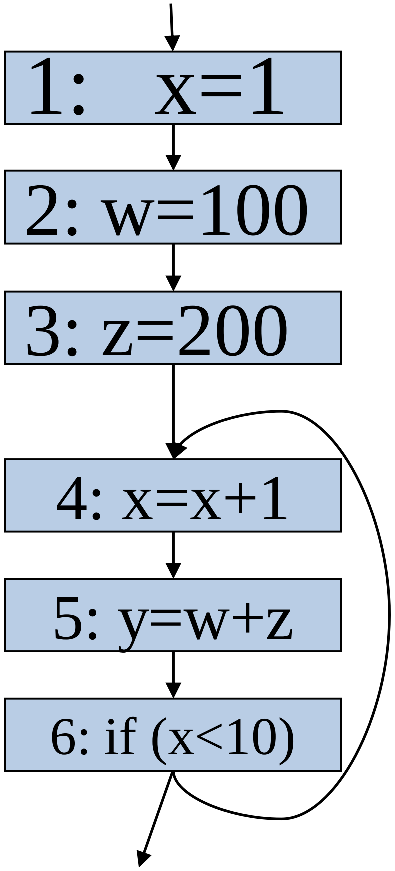
```
1    x = 1
2    w=100
3    z=200
  Here:
4    x = x+1
5    y=w+z
6    if (x<10) goto Here
```



- Reaching definitions (ReachIn):
- 1: []
- 2: [1]
- 3: [1,2]
- 4: [1,2,3,4,5]
- 5: [2,3,4,5]
- 6: [2,3,4,5]

Use reaching definitions to find loop invariant instructions

- Find the definitions which reach this node which are relevant – that is, which generate the values this node uses:



- Reaching definitions:
- 1: []
- 2: [1]
- 3: [1,2]
- 4: [1,2,3,4,5]
- 5: [2,3,4,5]
- 6: [2,3,4,5]

- “Relevant” Reaching definitions:
- 1: []
- 2: []
- 3: []
- 4: [1,4]
- 5: [2,3] ←
- 6: [4]

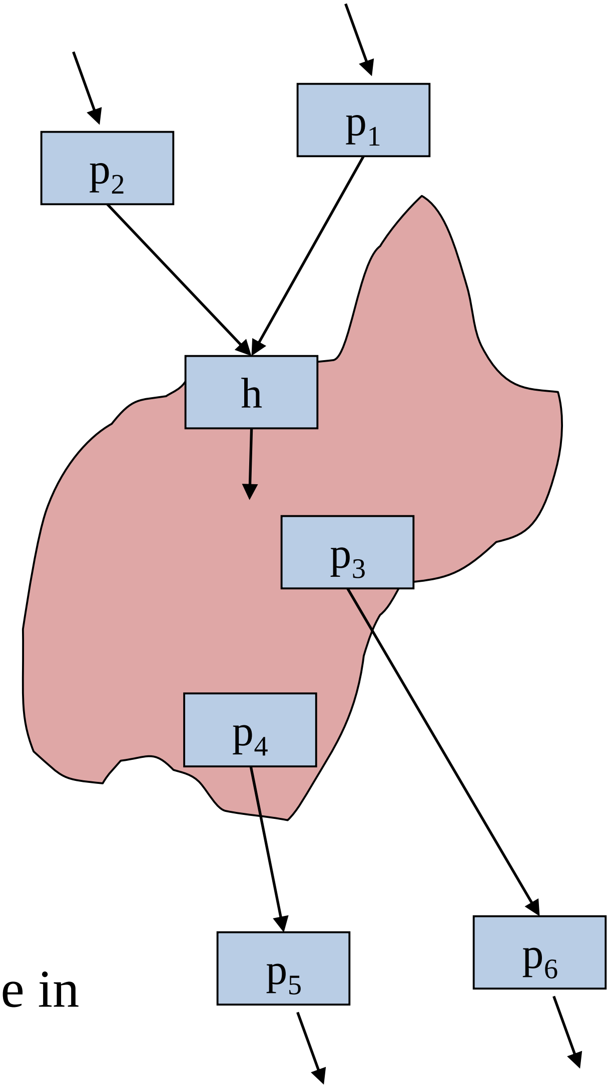
All the definitions of the values used by node 5 lie outside the loop

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 goto

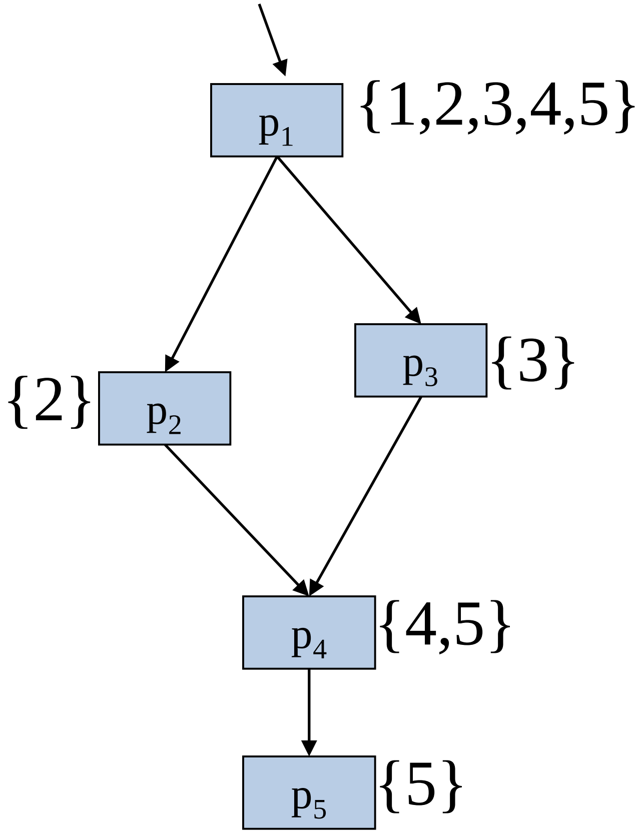
- Definition:

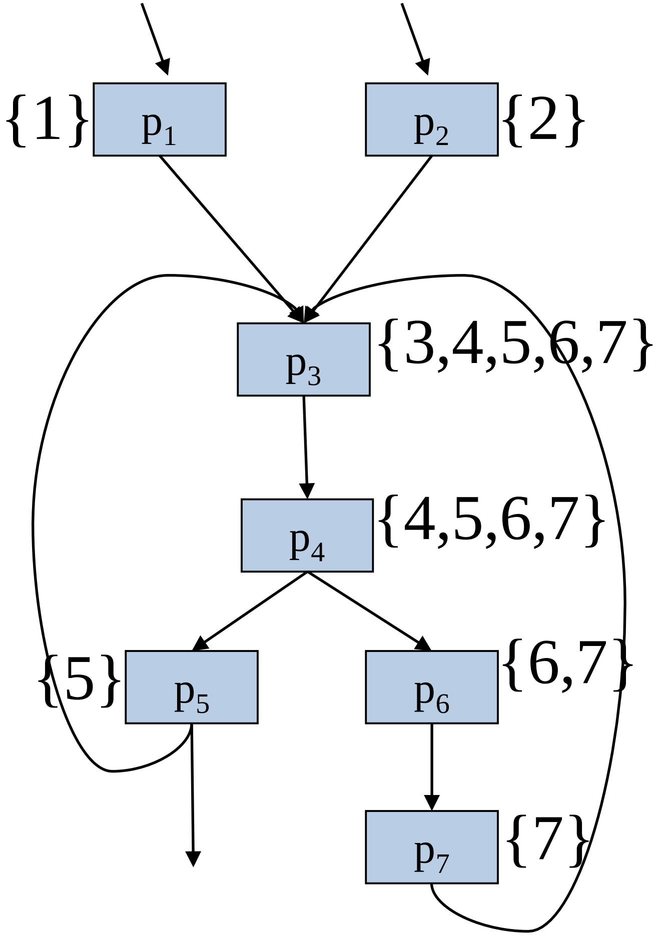
  A *loop* in a control flow graph is a set of nodes S including a *header* node h, with the following properties:
  - From any node in S there is a path leading to h
  - There is a path from h to any node in S
  - There is no edge from any node outside S to any node in S other than h



- **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

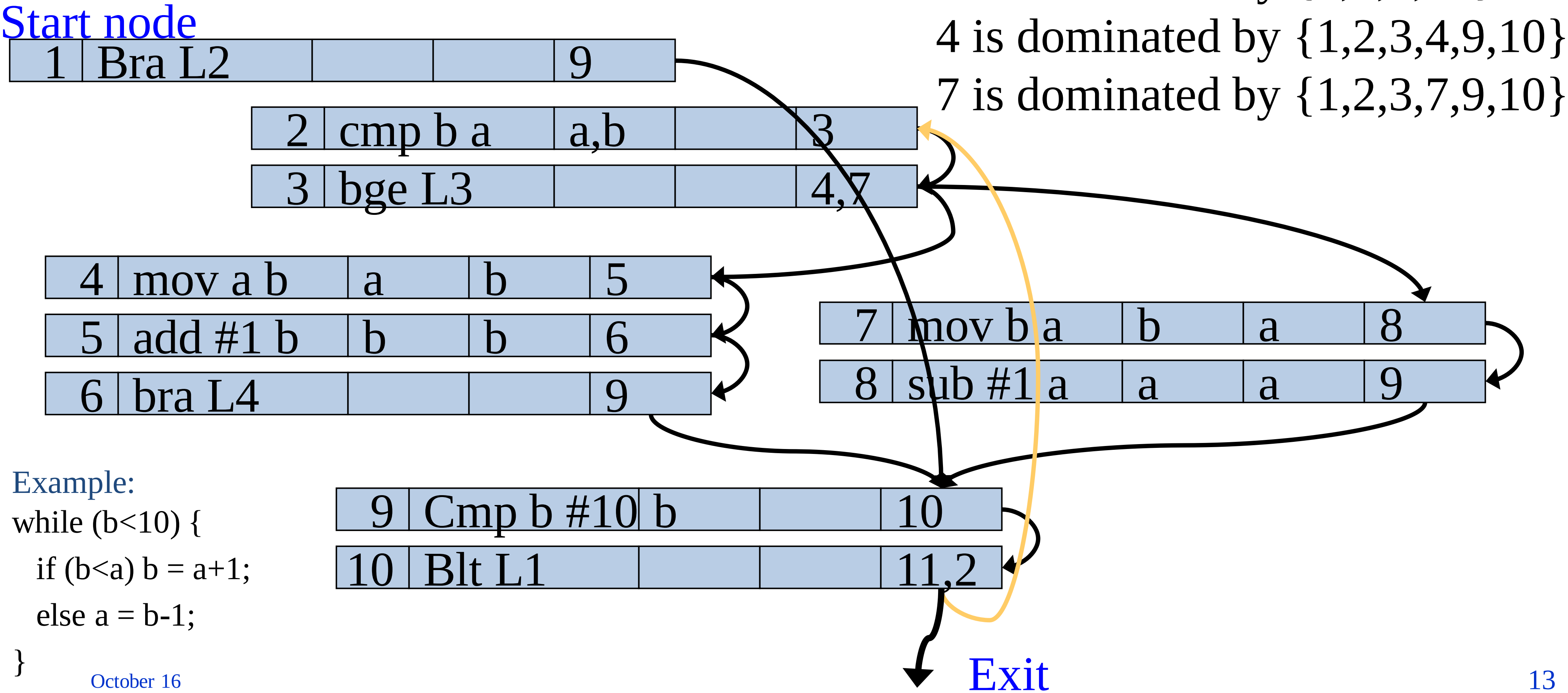




- **Definition: dominator**

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}



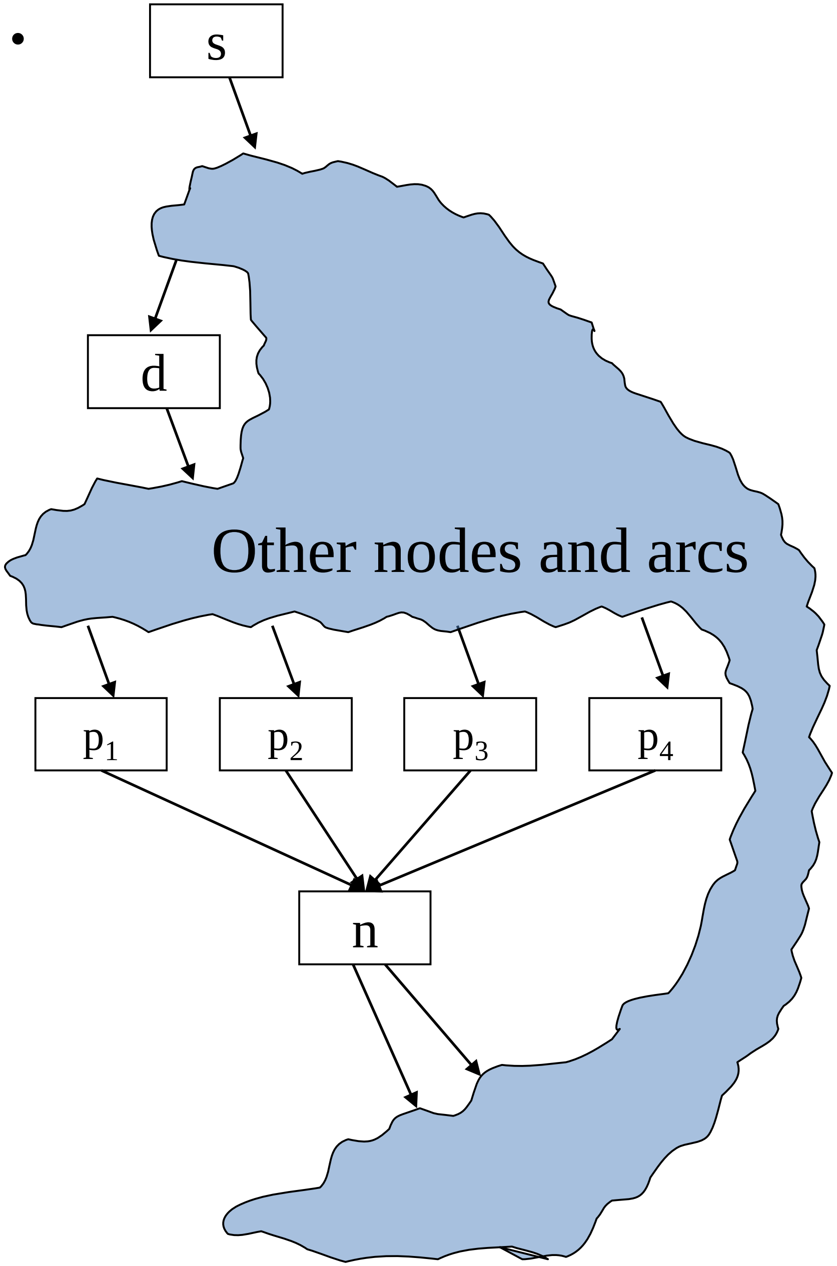
1	Bra L2			9
2	cmp b a	a,b		3
3	bge L3			4,7
4	mov a b	a	b	5
5	add #1 b	b	b	6
6	bra L4			9
7	mov b a	b	a	8
8	sub #1 a	a	a	9
9	Cmp b #10	b		10
10	Blt L1			11,2

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)")

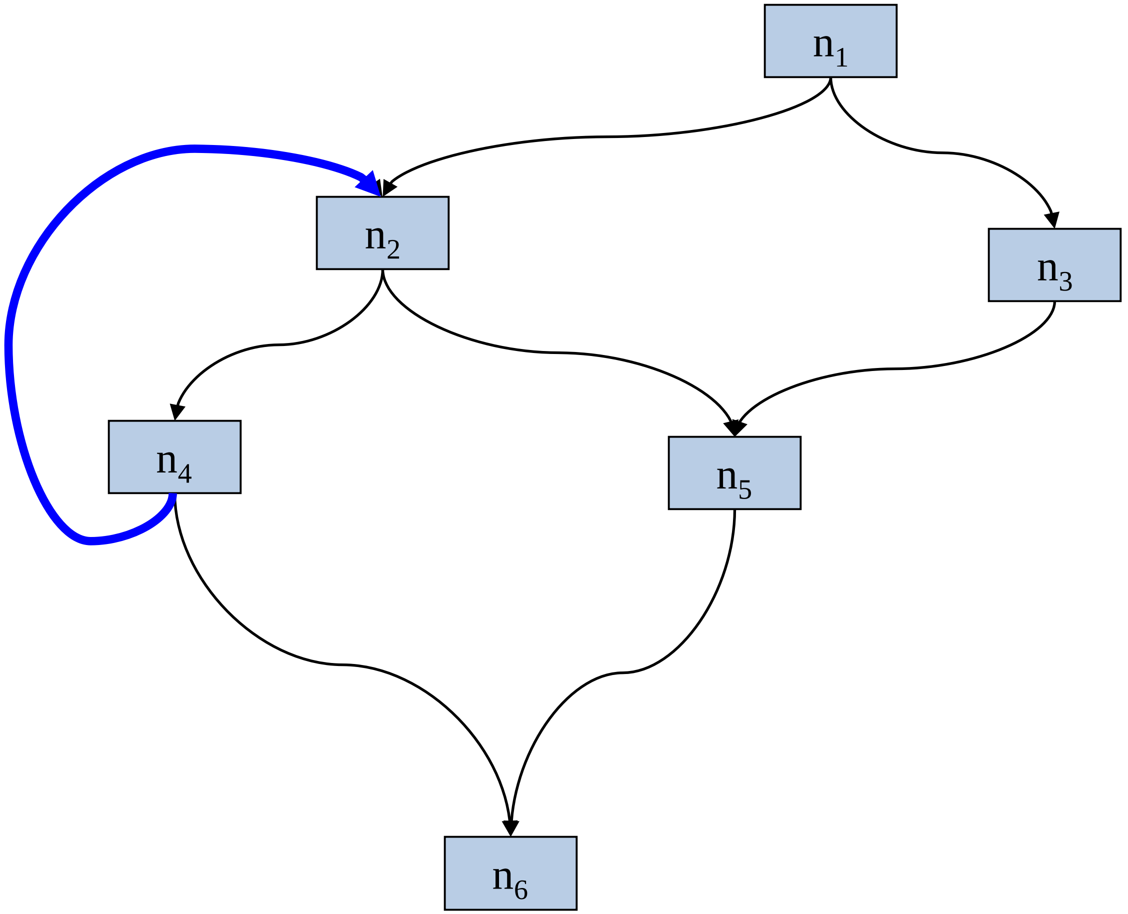
- Construct a system of simultaneous set equations:
- Doms(s) = { s } *(s = start node)*
- $\text{Doms}(n) = \{n\} \cup \left(\bigcap_{p \in \text{preds}(n)} \text{Doms}(p) \right)$ *(otherwise)*

  ("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

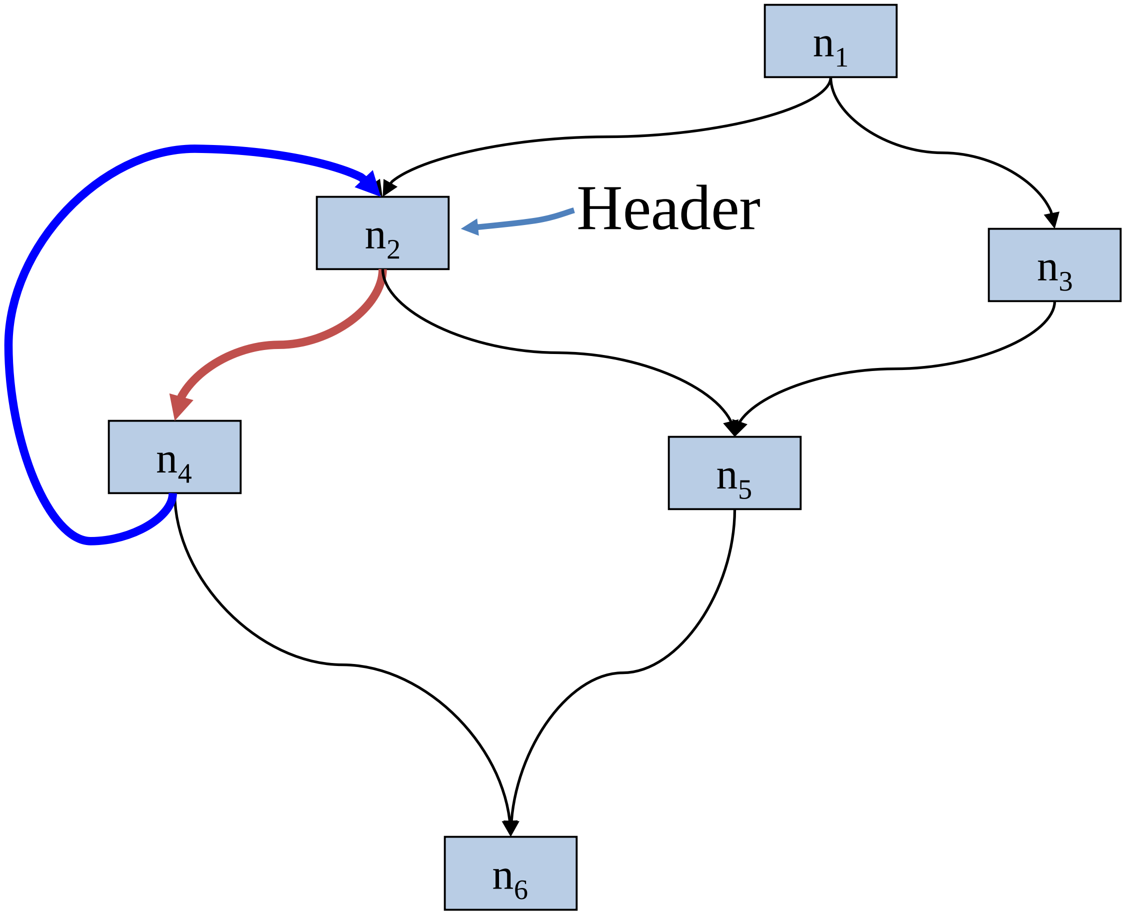
- A control flow graph edge from a node *n* to a node *h* that dominates *n* is called a ***back edge***.



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)



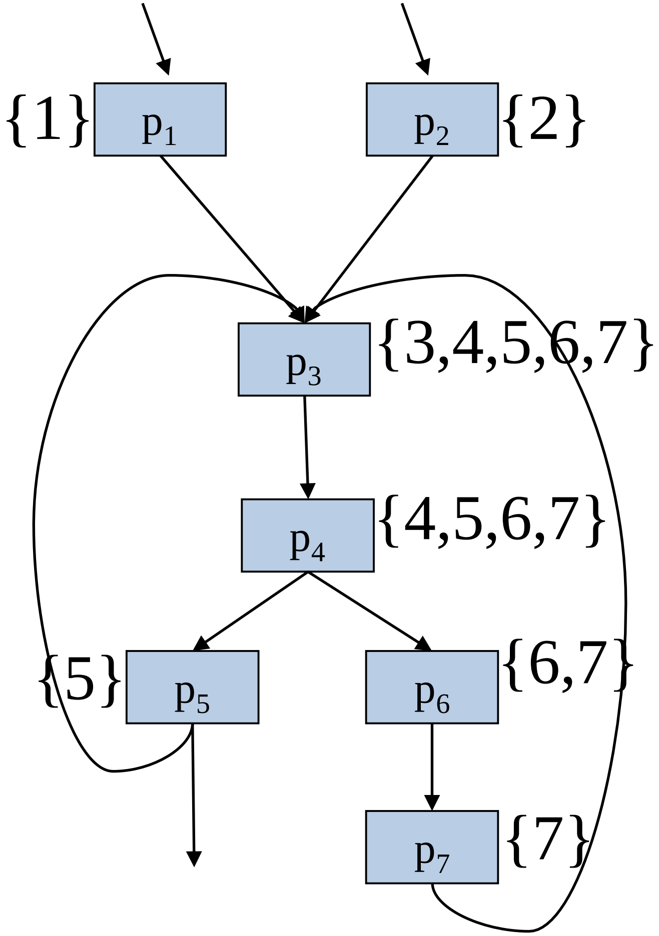
Definition:
The ***natural loop*** of a backedge (n,h), where h dominates n, is

- 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)



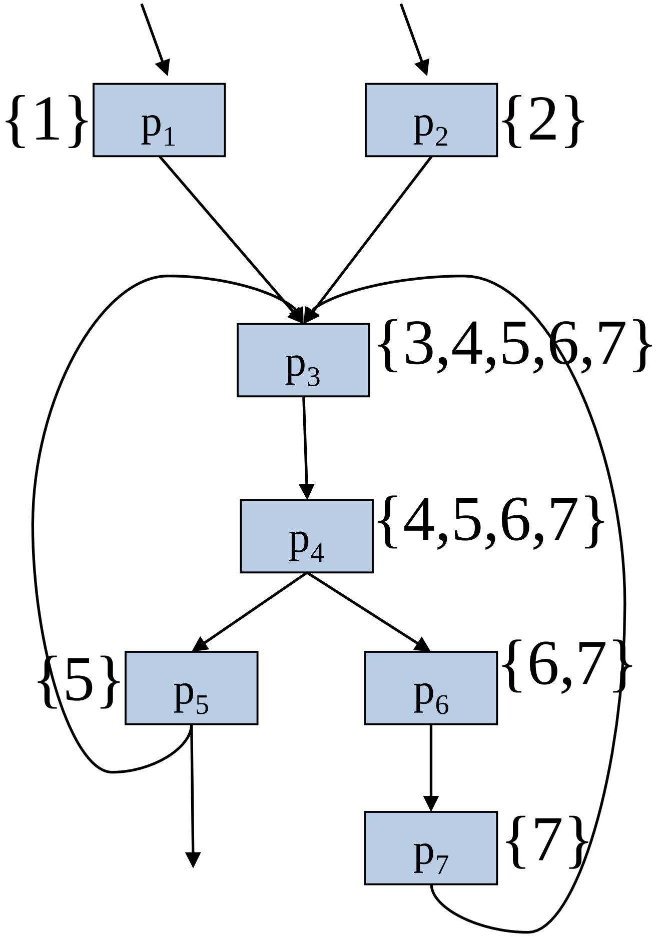
Definition:
The ***natural loop*** of a backedge (n,h), where h dominates n, is

- 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

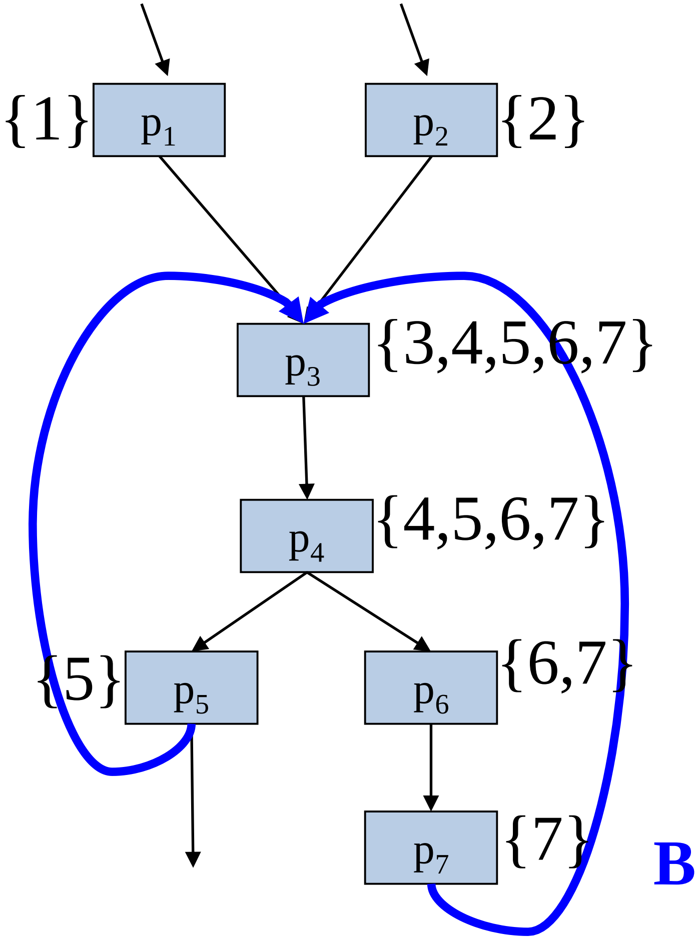
Multiple loops

- It is possible for two loops to share the same header
- This example has two back edges, (5,3) and (7,3)
- The easiest thing to do in this case is to treat them as one loop



Multiple loops

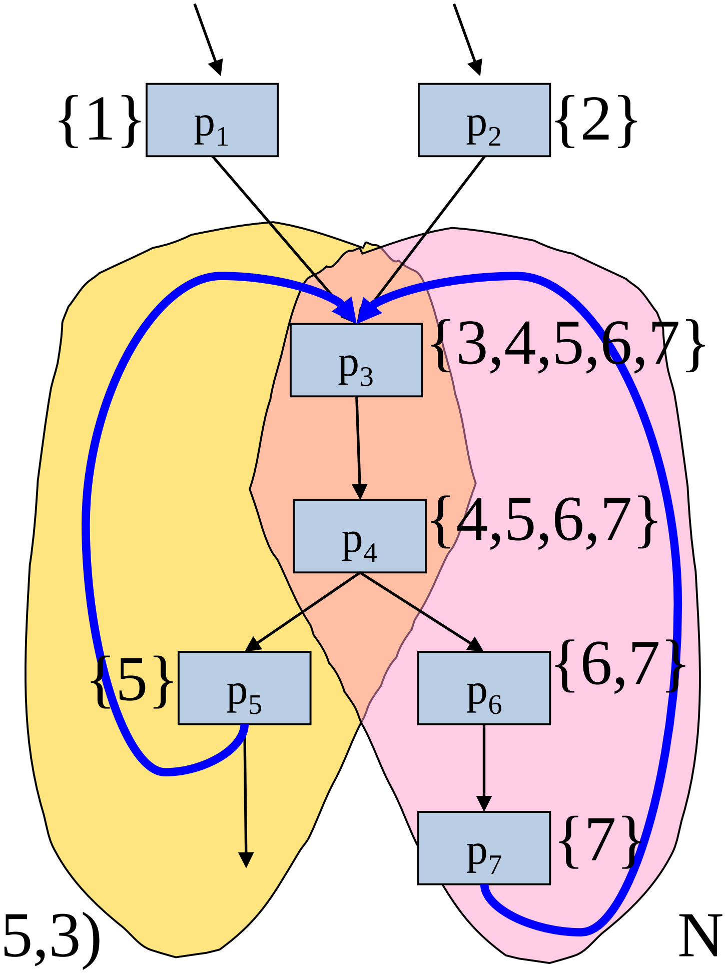
- It is possible for two loops to share the same header
- This example has two back edges, (5,3) and (7,3)
- The easiest thing to do in this case is to treat them as one loop



Back edges

Multiple loops

- It is possible for two loops to share the same header
- This example has two back edges, (5,3) and (7,3)
- The easiest thing to do in this case is to treat them as one loop

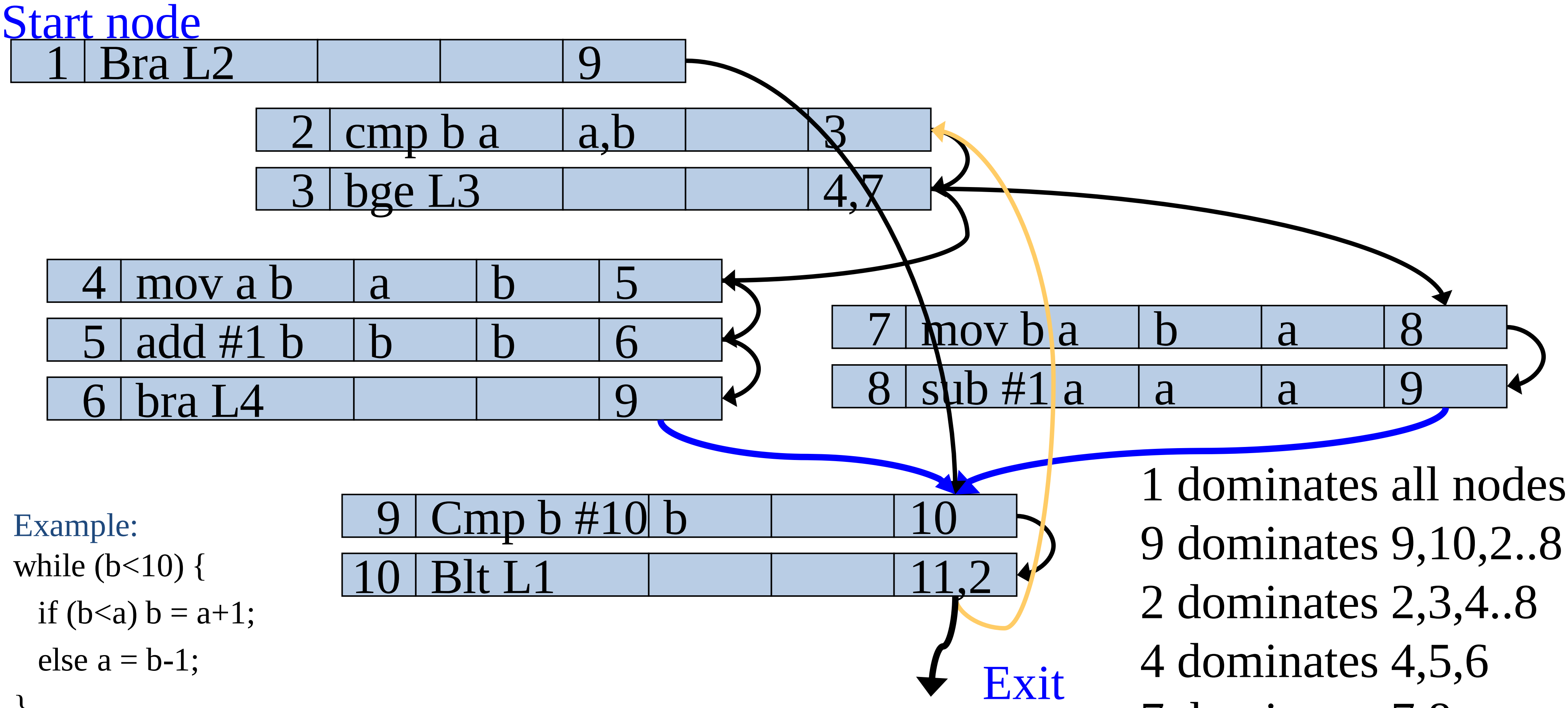


Natural loop of (5,3)

Natural loop of (7,3)

Multiple loops

- It is possible for two loops to share the same header
- This example has two back edges, (6,9) and (8,9)
- The easiest thing to do in this case is to treat them as one loop



1	Bra L2			9
2	cmp b a	a,b		3
3	bge L3			4,7
4	mov a b	a	b	5
5	add #1 b	b	b	6
6	bra L4			9
7	mov b a	b	a	8
8	sub #1 a	a	a	9
9	Cmp b #10	b		10
10	Blt L1			11,2

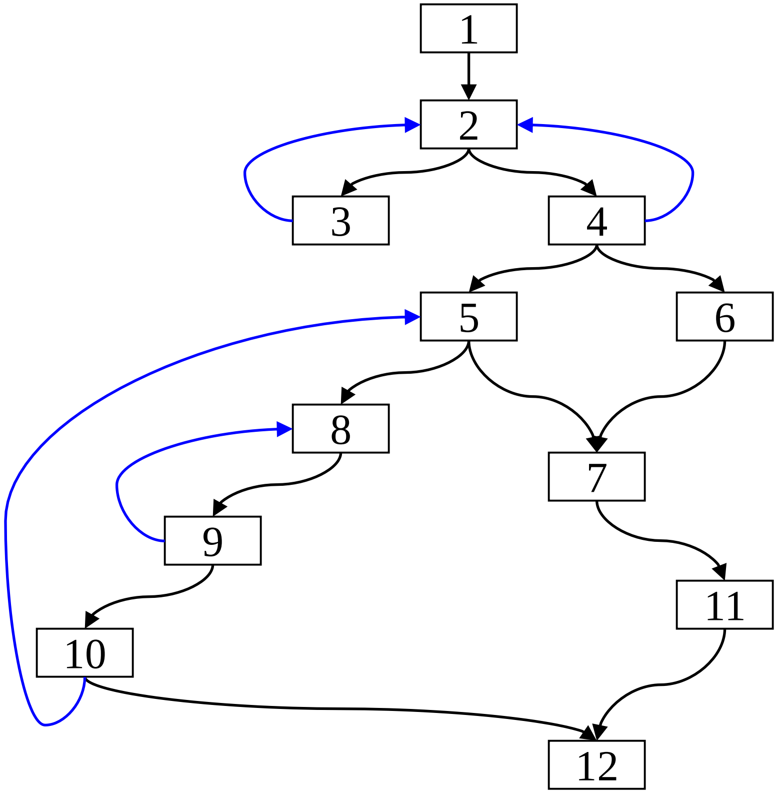
1 dominates all nodes
9 dominates 9,10,2..8
2 dominates 2,3,4..8
4 dominates 4,5,6
7 dominates 7,8

Example:

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

Nested loops

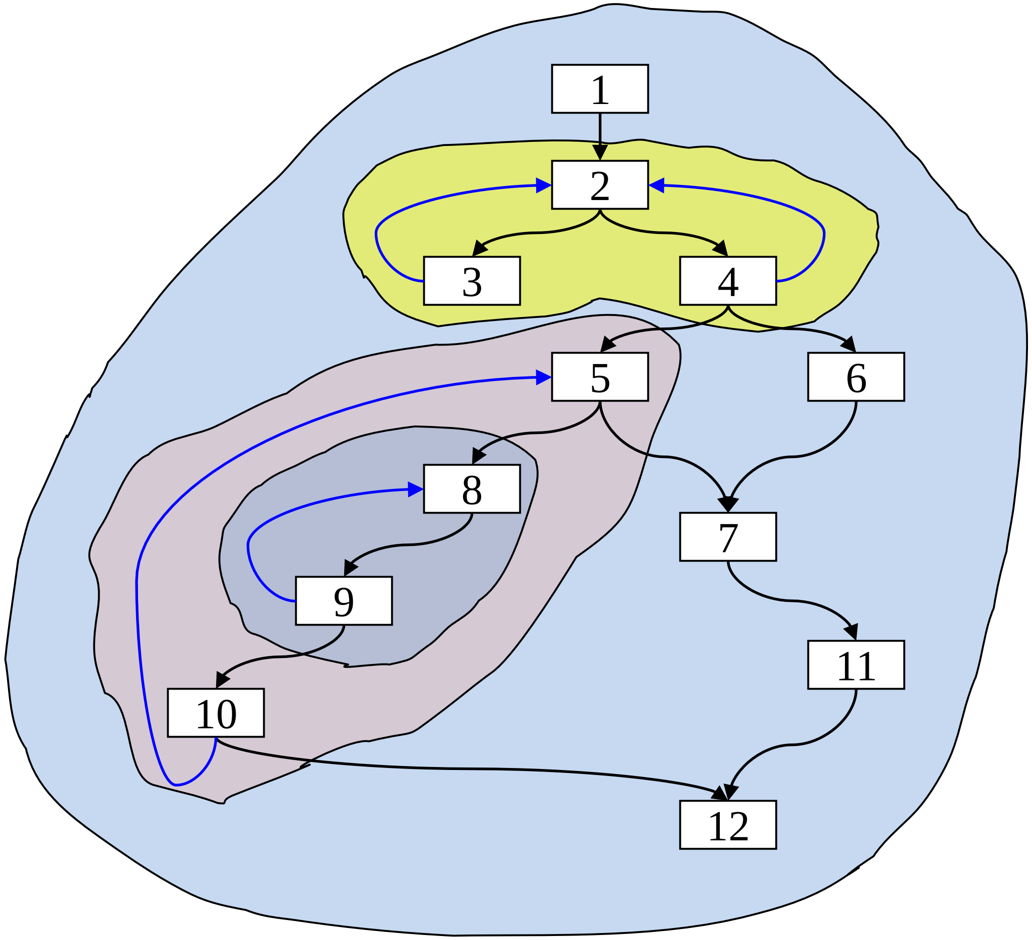
- Suppose:
  - A and B are loops with headers a and b, such that a ≠ b, and b is in A
- Then
  - The nodes of B must be a proper subset of the nodes of A
  - We say that loop B is nested within A
  - B is the inner loop



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

Nested loops

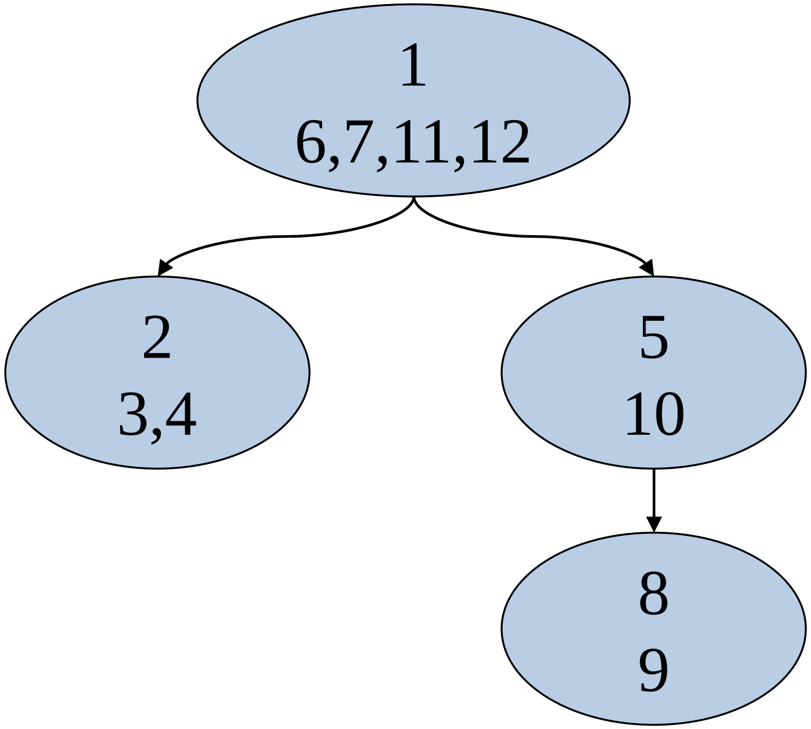
- Suppose:
  - A and B are loops with headers a and b, such that a ≠ b, and b is in A
- Then
  - The nodes of B must be a proper subset of the nodes of A
  - We say that loop B is nested within A
  - B is the inner loop

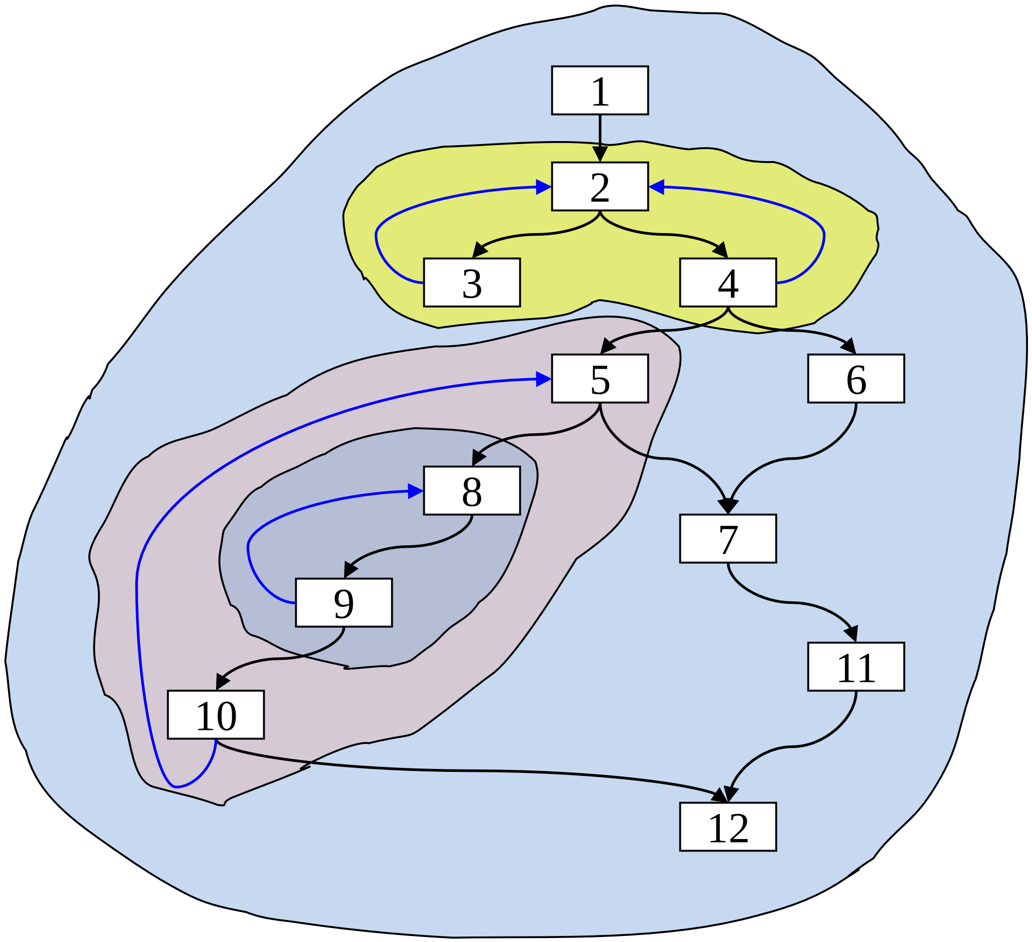


Back edges: (3,2), (4,2), (10,5), (9,8)

The Control Tree

- Loops form a tree
- Example:



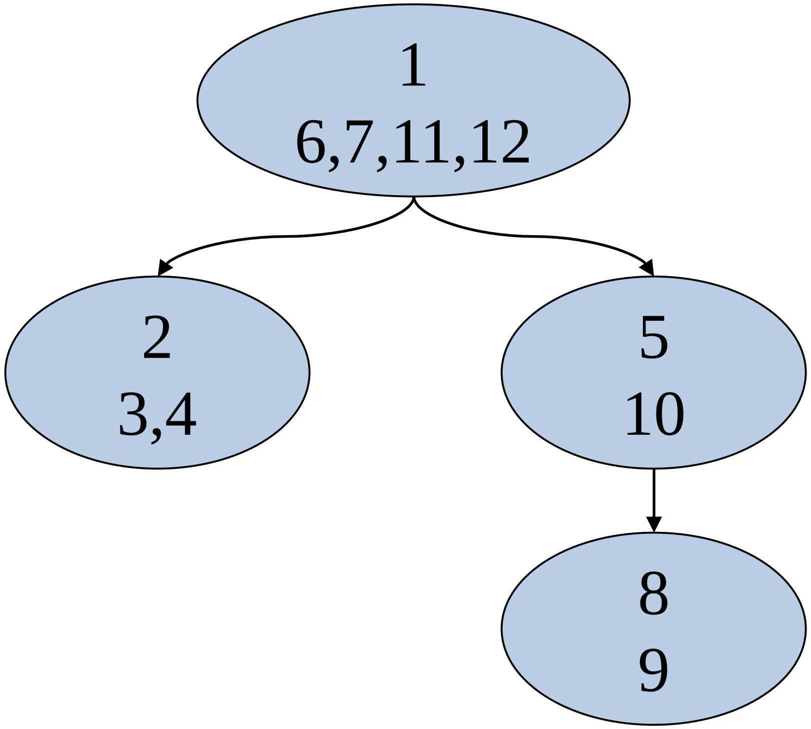


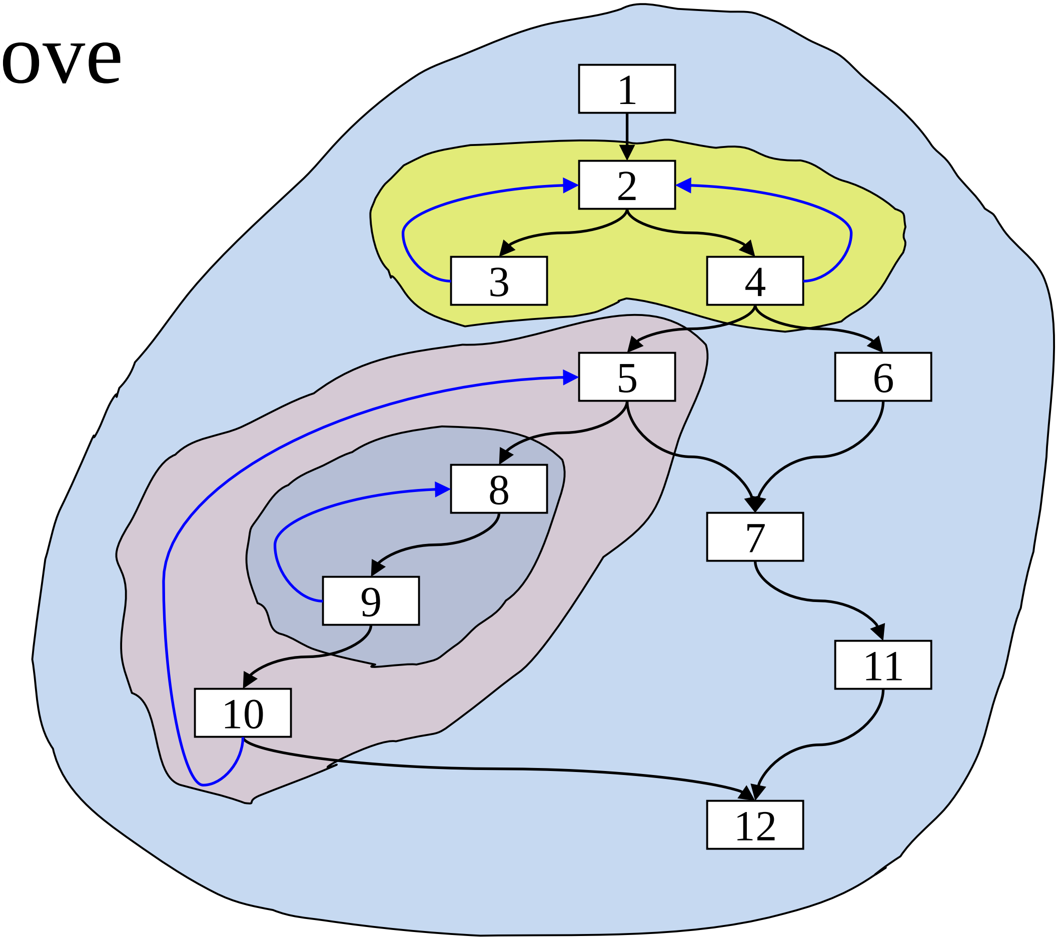
Back edges: (3,2), (4,2), (10,5), (9,8)

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

Pre-headers

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

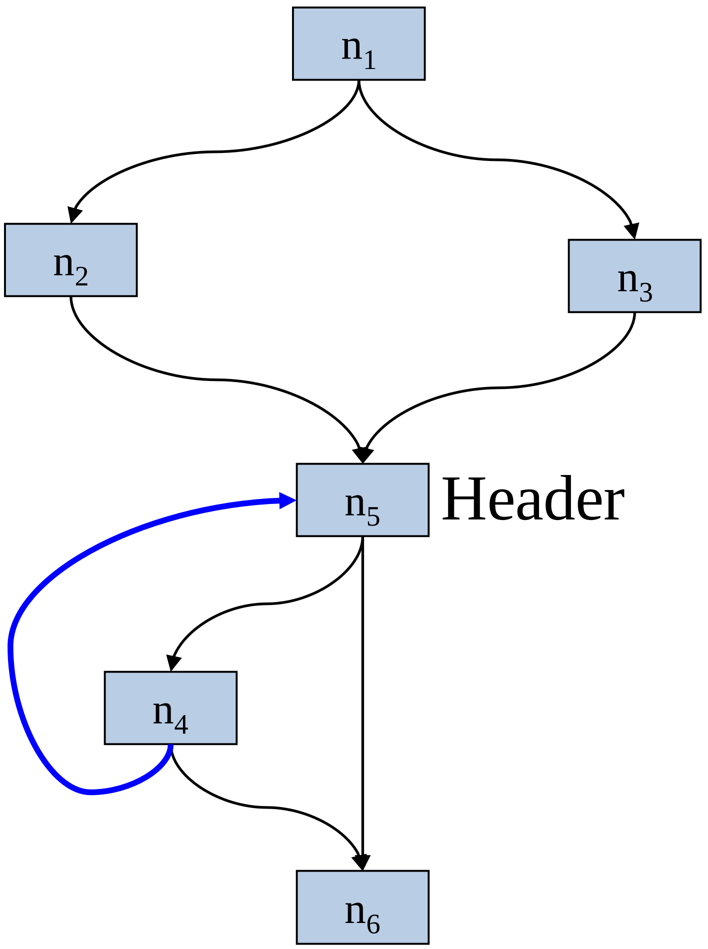




- We can’t move them to the header
- We want to move them to the node preceding the header

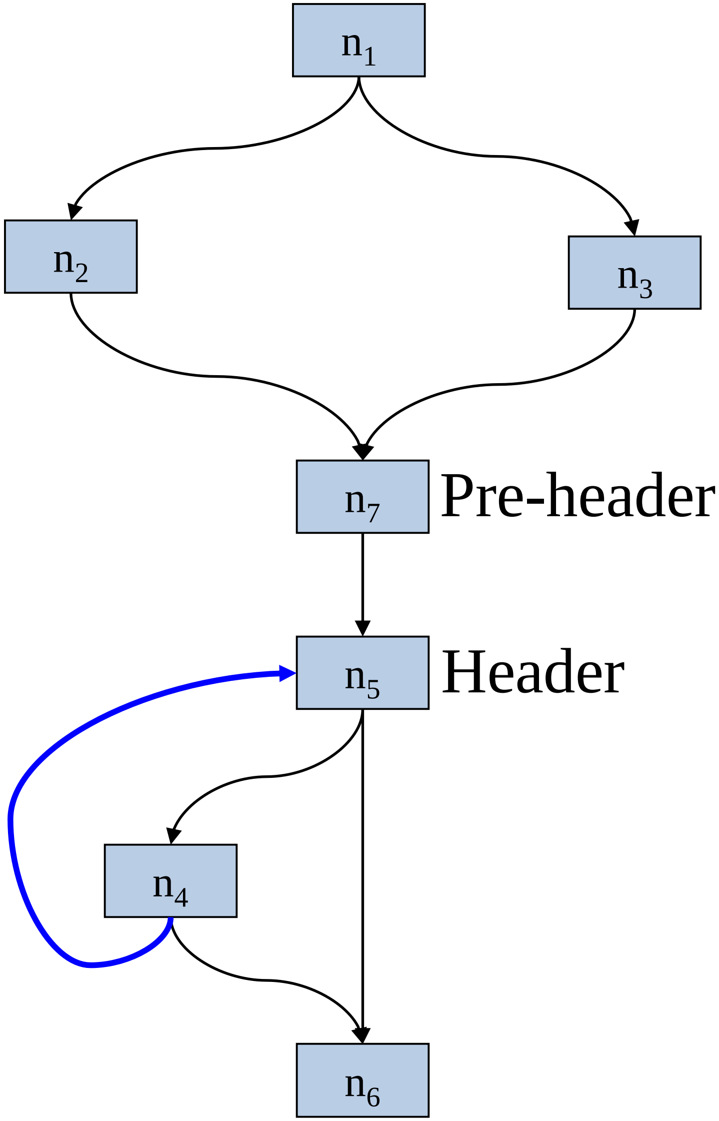
Pre-headers

- 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**



Which instructions can we move out of a loop?

- The next question is exactly which loop-invariant instructions we can move to the pre-header
- It’s easy to get it wrong….

A

```
$L_0$:
 t = 0

$L_1$:
 i = i+1
 t = a ⊕ b
 M[i] = t
 if i<N goto $L_1$
$L_2$:
 x = t
```

B

```
$L_0$:
 t = 0

$L_1$:
 if i<N goto $L_2$
 i = i+1
 t = a ⊕ b
 M[i] = t
 goto L1
$L_2$:
 x = t
```

Which instructions can we move out of a loop?

- The next question is exactly which loop-invariant instructions we can move to the pre-header
- It’s easy to get it wrong….

A

```
$L_0$:
 t = 0
 t = a ⊕ b
$L_1$:
 i = i+1
 ~~t = a ⊕ b~~
 M[i] = t
 if i<N goto $L_1$
$L_2$:
 x = t
```

B

```
$L_0$:
 t = 0
 t = a ⊕ b
$L_1$:
 if i<N goto $L_2$
 i = i+1
 ~~t = a ⊕ b~~
 M[i] = t
 goto L1
$L_2$:
 x = t
```

Which instructions can we move out of a loop?

- It’s easy to get it wrong….

C

```
L0:
 t = 0


L1:
 i = i+1
 t = a ⊕ b
 M[i] = t
 t = 0
 M[j] = t
 if i<N goto L1
L2:
```

D

```
L0:
 t = 0


L1:
 M[j] = t
 i = i+1
 t = a ⊕ b
 M[i] = t
 if i<N goto L1
L2:
 x = t
```

Which instructions can we move out of a loop?

- It’s easy to get it wrong….

C

L_0:
t = 0
t = a ⊕ b
L_1:
i = i+1
~~t = a ⊕ b~~
M[i] = t
t = 0
M[j] = t
if i<N goto L_1
L_2:

D

L_0:
t = 0
t = a ⊕ b
L_1:
M[j] = t
i = i+1
~~t = a ⊕ b~~
M[i] = t
if i<N goto L_1
L_2:

Which instructions can we move out of a loop?

A

```
$L_0$:
 t = 0
 t = a ⊕ b
$L_1$:
 i = i+1
 ~~t = a ⊕ b~~
 M[i] = t
 if i<N goto $L_1$
$L_2$:
 x = t
```

Hoist

B

```
$L_0$:
 t = 0
 t = a ⊕ b
$L_1$:
 if i<N goto $L_2$
 i = i+1
 ~~t = a ⊕ b~~
 M[i] = t
 goto L1
$L_2$:
 x = t
```

Don’t hoist:
Loop invariant node does not dominate all loop exits

C

```
$L_0$:
 t = 0
 t = a ⊕ b
$L_1$:
 i = i+1
 ~~t = a ⊕ b~~
 M[i] = t
 t = 0
 M[j] = t
 if i<N goto $L_1$
$L_2$:
```

Don’t hoist:
More than one definition of t in the loop

D

```
$L_0$:
 t = 0
 t = a ⊕ b
$L_1$:
 M[j] = t
 i = i+1
 ~~t = a ⊕ b~~
 M[i] = t
 if i<N goto $L_1$
$L_2$:
```

Don’t hoist:
t is liveOut from the loop’s preheader

Which instructions can we move out of a loop?

- Conditions for hoisting a CFG node

d: t = a ⊕ b

1	Loop invariant: all reaching defs used by d occur outside loop	Use Reaching Definitions data flow analysis
2	Loop invariant node must dominate all loop exits	Use Dominators analysis
3	There must be just one def of t in loop	Just count them!
4	t must not be liveOut from the loop's preheader	Use Live Variables data flow analysis

What next…

- Hoisting loop invariants really helps
- But good compilers do lots more…
  - Induction variables:
    - A variable which is incremented by a loop-invariant amount
    - A variable which is a multiple of an induction variable
  - Strength reduction
    - Compute all induction variables by incrementing instead of multiplying
  - Induction variable elimination, rewriting comparisons
  - Array bounds check elimination
    - Range of all induction variables is known on entry to a for loop
  - Common sub-expressions
  - More sophisticated methods – eg partial redundancy elimination
- Now you have seen how to hoist loop-invariants, you can figure the rest out yourself!

Optimisations for high-performance computing

- “Conventional” optimisations *reduce* work done at run-time
- “restructuring” compilers improve performance by finding the *right order* in which to do the computation
- Example: Parallelisation:

Original code:

```
For (i=0;i<N;i++)
  For (j=0;j<M;j++)
   A[i,j] = (A[i,j] + A[i-1,j] + A[i+1,j])* (1/3)
```

Parallel implementation:

```
For (i=0;i<N;i++)
  ParFor (j=0;j<M;j++)
   A[i,j] = (A[i,j] + A[i-1,j] + A[i+1,j])* (1/3)
```

Better parallel implementation?

```
ParFor (j=0;j<M;j++)
  For (i=0;i<N;i++)
   A[i,j] = (A[i,j] + A[i-1,j] + A[i+1,j])* (1/3)
```

Optimisations for high-performance computing

- Another restructuring example:

Example: matrix transpose:

```
for (i=0;i<N;i++)
   for (j=0;j<M;j++)
     B[i][j] = A[j][i];
```

Cache-efficient implementation:

```
for (ii=0;ii<N;ii+=IB)
   for (jj=0;jj<M;jj+=JB)
     for (i=ii;i<ii+IB;i++)
       for (j=jj;j<jj+JB;j++)
         B[i][j] = A[j][i];
```

Using 1.4GHz AMD Athlon
N=M=2000
IB=JB=100
Original execution time: 462ms
Improved execution time: 96ms

Optimisations for high-level programming languages

- Subtype polymorphism
  - Static resolution of the type of x in x.f( ) enables inlining of method f
- Generics (aka parametric polymorphism)
  - A generic class is parameterised by a type (eg a container by its element type). When is it a good idea to generate specialised code?
- Pattern matching
  - In a language like Haskell or Prolog, pattern matching on nested data structures is very powerful. Find optimum sequence of tests.
- Dynamic object creation
  - If we allow space to be allocated, but automatically freed, can we sometimes add code to do it instead of relying on garbage collection?
- Lazy evaluation
  - Can an expression be evaluated where it is first referred to, or do we have to build a "closure" representing it?
- Arrays – overloaded arithmetic
  - If we overload arithmetic operators to work on arrays, how to avoid lots of little loops?
- Arrays – slices
  - If we allow a multidimensional array to be sliced, eg A[2:99,2:99], how do we avoid having to manipulate an array descriptor?

Textbooks

EaC

- Data flow analysis is covered in Chapter 9
  - Reaching definitions are covered in Section 9.2.4
  - Dominators are covered in Section 9.3.2
- EaC handles loop-invariant code motion somewhat differently from these slides, which are based on Appel’s presentation
  - See “Lazy Code Motion” (LCM), page 506
  - LCM resolves the hoisting conditions in a more systematic way than presented here, by combining four different data-flow analyses

Textbooks

- Appel also covers optimisation in depth
  - Chapter 10 introduces DFA through live variable analysis
  - Chapter 17 shows how DFA can be used for many other useful analyses
  - Chapter 18 deals with finding loops, finding induction variables, and implementing loop optimisations (which rely on DFAs)
  - Chapter 19 presents Static Single Assignment, a program representation which provides easy (and space-efficient) access to dependence information such as reaching definitions. This simplifies many loop optimisations
  - Chapter 20 covers instruction scheduling – finding an instruction ordering which makes optimal use of modern CPU architectures
  - Chapter 21 concerns improving cache performance – by prefetching, and by executing loops blockwise
- Another really good source if you're building an optimising compiler is "High-performance compilers for parallel computing", Michael Wolfe (Addison Wesley 1996)

- Fine print:
  - CFG would consist of basic blocks instead of individual instructions
  - For loop optimisations, we would do the DFA on the IR before instruction selection; it's simpler and it avoids complications such a having only two-address instructions
  - See Appel pg388
- Credits: in addition to Appel's book, I found it very useful to study the course notes of Liz White (George Mason University), Laurie Hendren (McGill University) and Chau-Wen Tseng (University of Maryland)

Research

- Several Imperial research groups are working on optimising compiler technology, including:
  - Wayne Luk’s Custom Computing/Silicon Compilation group
  - Alastair Donaldson’s group
  - Paul Kelly’s Software Performance Optimisation group
  - Compiler-related research: Cristian Cadar, Peter Pietzuch, Sergio Maffeis etc
  - Programming languages: Sophia Drossopoulou, Nobuko Yoshida, and others
- Opportunities: UROP summer placements, individual projects, and PhDs
- Sample projects:
  - Automatically searching for the best combination of blocking, loop fusion, unrolling, parallelisation, vectorisation
  - Work computational scientists to make their simulation of tidal turbines/Formula 1/blood flow/weather run fast on 10,000-100,000 cores
  - Efficient execution of analysis queries on results from large parallel fluid dynamics simulations
  - Design a domain-specific language and compiler to generate high-performance code for 3D robot vision and scene understanding

Implementing loop optimisations in Haskell

- The next few slides give a Haskell implementation for some of the ideas presented in this chapter
- This material is provided to provide a concrete illustration of the concepts
- It is the concepts which are important, not the code
- **Do *not* memorise the code** – spend the time reading the textbook instead
- Some of the algorithms used here are rather inefficient – in many cases we just transcribe the mathematical definitions. Efficient algorithms exist – but are considerably more complicated.

Reaching definitions – gen and kill

- Preliminaries: the Gen and Kill sets:

```
nodeGen node | nodeDefs node == []  = []
             | otherwise           = [nodeId node]

nodeKill cfg node = nodeDefSet cfg node \\ [nodeId node]
```

- Suppose t is defined in node. nodeDefSet is set of all the nodeids where t is defined:

```
nodeDefSet (ControlFlowGraph cfg) node
 = case nodeDefs node of
   [t] -> [id | Node id i ds us scs prds <- cfg,
                t `elem` ds]
   []  -> []
   otherwise -> error "nodeDefSet: multiple defs“
```

- Auxiliary functions used in solver overleaf:

```
untilConverges (a:b:rest) | a == b = a
untilConverges (a:b:rest)          = untilConverges (b:rest)

zip2 (rdsin,rdsout) = zip rdsin rdsout
bigU sets = nub (concat sets)
```

Reaching definitions - solver

- Solve the dataflow equations:

```
reachingDefinitionsOf :: CFG -> ( [ (Id,[Id]) ], [ (Id,[Id]) ] )

reachingDefinitionsOf cfg
 = untilConverges (iterate updateRDs initialRDs)
   where
   initialRDs :: ( [ (Id,[Id]) ], [ (Id,[Id]) ] )
   initialRDs = ( [(n,[]) | n<-nodesOf cfg], [(n,[]) | n<-nodesOf cfg] )

   updateRDs :: ( [(Id,[Id])], [(Id,[Id])] ) -> ( [(Id,[Id])], [(Id,[Id])] )
   updateRDs rds = unzip (map (updateRD rds) (zip2 rds))
   updateRD (rdins_sofar,rdouts_sofar) ((id,rdins), (sameid,rdouts))
    = ((id,rdins'), (id,rdouts'))
      where
      rdins' = bigU [retrieve s rdouts_sofar | s <- nodePreds node]
      rdouts' = nodeGen node `union` ((rdInsOf node) \\ nodeKill cfg node )
                where
                rdInsOf node = retrieve (nodeId node) rdins_sofar
      node = idToNode cfg id
```

- We solve the system of simultaneous set equations iteratively
- Initially each node’ s ReachIn (rdins), and ReachOut (rdouts) set is empty
- The updates successively increase the ReachIn and ReachOut sets until convergence

Use reaching definitions to find loop invariant instructions

- Find the definitions which reach this node which are relevant – that is, which generate the values this node uses:

```
relevantReachingDefinitionsOf :: CFG -> [ (Id,[Id]) ]

relevantReachingDefinitionsOf cfg
 = [(nodeId node, relevantDefs node) | node <- cfgToNodes cfg]
    where
    relevantDefs node
       = [rd | rd <- retrieve (nodeId node) rds_in,
                    nodeDefs (idToNode cfg rd) `intersect` nodeUses node /= []]
    (rds_in, rds_out) = reachingDefinitionsOf cfg
```

Use reaching definitions to find loop invariant instructions

- An instruction is loop invariant if the definitions of all the values it uses are outside the loop:

```
> externallyDependentInstructionsOf cfg loop
>  = [node | node <- [idToNode cfg id | id <- loop],
>            nodeDefs node /= [],
>            relevantDefs node `intersect` loop == [],
>            hoistable node]
>   where
>   relevantDefs node = retrieve (nodeId node)
          (relevantReachingDefinitionsOf cfg)
```

- An instruction is hoistable only if it produces a value (ie not a compare, branch, etc):

```
 hoistable (Node id i [] uses succs preds) = False
 hoistable (Node id i defs uses succs preds) = True
```

Use reaching definitions to find loop invariant instructions

- Now iteratively add instructions which are l-i because they depend only on l-i instructions. We reverse the result so that when we add them to the pre-header, they are added in dependence-order.

```
> loopInvariantInstructionsOf cfg loop
>  = reverse (untilConverges (iterate updateLIs initialLIs))
>    where
>    initialLIs = externallyDependentInstructionsOf cfg loop
>    updateLIs :: [CFGNode] -> [CFGNode]
>    updateLIs invariantsSoFar
>     = invariantsSoFar `union`
>       [n | n <- map (idToNode cfg) loop,
>            hoistable n,
>            and [hasSingleInvariantDefinition n u | u<-nodeUses n]]
>       where
>       hasSingleInvariantDefinition n u
>        = length defs == 1 && head defs `elem` map nodeId invariantsSoFar
>         where
>         defs = [d | d<-relevantDefs n, u `elem` nodeDefs (idToNode cfg d)]
```

Finding dominators… implementation

```
dominatorsOf :: CFG -> [(Id,[Id])]
dominatorsOf cfg
 = untilConverges (iterate updateDs initialDs)
   where
   initialDs :: [(Id,[Id])]
   initialDs = [ (n, nodesOf cfg) | n <- (nodesOf cfg)]

   updateDs :: [(Id,[Id])] -> [(Id,[Id])]
   updateD ds_sofar (id,d)
    = (id,
         [id] `union` (bigCap [retrieve p ds_sofar | p <- nodePredsOf id])
       )
   updateDs ds = map (updateD ds) ds

   nodePredsOf id = nodePreds (idToNode cfg id)

bigCap [] = []
bigCap sets = foldr1 intersect sets

untilConverges (a:b:rest) | a == b = a
untilConverges (a:b:rest)          = untilConverges (b:rest)
```

- We solve the system of simultaneous set equations iteratively
- Initially each node’s Doms set is the set of all the nodes of the CFG
- The updates successively reduce the Doms until convergence

Finding back edges

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

```
backEdges :: CFG -> [(Id,Id)]

backEdges cfg
 = [ (n,h) | n <- nodesOf cfg, h <- nodesOf cfg, n /= h,
             flowedge n h,
             h `dominates` n]
  where
  dominators = dominatorsOf cfg
  a `dominates` b = a `elem` (retrieve b dominators)
  flowedge a b = a `elem` nodePreds (idToNode cfg b)
```

Finding natural loops

- 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.

```
naturalLoop :: CFG -> (Id,Id) -> (Id, [Id])
--                    backedge   header, nodes

naturalLoop cfg (n,header)
 = (header, real_xs)
   where
   poss_xs = [x | x <- nodesOf cfg, header `dominates` x]
   real_xs = [x | x <- poss_xs, pathExists x n]
   pathExists x n                                        (omit paths via header, and
    = [] /= [path | path <- allpaths, not (header `elem` path)] therefore paths via enclosing
      where                                              loops)
      allpaths = findControlFlowPaths cfg x n      (findControlFlowPaths defined next slide)
   dominators = dominatorsOf cfg
   a `dominates` b = a `elem` (retrieve b dominators)
```

Finding paths

- I have used a general-purpose path enumeration to find all the paths from one node to another. This is rather wasteful... Some care is needed to avoid following cycles; "mypath" below records the nodes visited so far.

```
findControlFlowPaths :: CFG -> Id -> Id -> [[Id]]

findControlFlowPaths cfg start end = findControlFlowPaths' [] start
   where
   findControlFlowPaths' mypath x
    | x == end              = [[x]]
    | x `elem` mypath       = [[]]
    | otherwise             = map (x:) restOfPath
           where
           extendedpath = x:mypath
           succs = nodeSuccs (idToNode cfg x)
           nonCycleSuccs = succs
           restOfPath = concat (map (findControlFlowPaths' extendedpath) nonCycleSuccs)
```

Building the loop nest tree (a.k.a. the control tree)

- The loop nest tree consists at each level of a loop (with its header), and the list of all its subloop trees:

```
data LoopTree = LTree (Id,[Id]) [LoopTree] deriving (Show, Eq)

loopTree :: CFG -> LoopTree

loopTree cfg
 = LTree (0, nodesOf cfg) (makeTrees theloops)
    where
    backedges = backEdges cfg
    theloops = map (naturalLoop cfg) backedges
    makeTrees loops = map makeTree (siblingloops loops)
    makeTree loop
     = LTree loop (makeTrees subloops)
       where
       subloops = [(h,nub l) | (h,l) <- theloops, containedIn (h,l) loop]
```

Building the loop nest tree…

- The children of a given loop are the immediate subloops. A subloop is an immediate subloop if it is not contained in any other loop in the list:

```
siblingloops loops
 = [l1 | l1 <- loops,
        not (any (containedIn l1) [l2 | l2<-loops, l1 /= l2]) ]
```

- To work out whether one loop l1 is strictly contained within another l2, we ask simply whether l1's header is in l2's body:

```
containedIn :: (Id,[Id]) -> (Id,[Id]) -> Bool
containedIn (h1,l1) (h2,l2) = h1 `elem` l2
```

Manipulating the control flow graph…

- To implement hoisting of loop invariants we need a few other functions:
  - Insert a pre-header before each loop header:

```
> addPreHeaders :: CFG -> LoopTree -> ([(Id,Id)], CFG)
> addPreHeaders cfg looptree =
```

  - Remove a specified list of nodes from a cfg

```
> removeNodes :: [CFGNode] -> CFG -> CFG
> removeNode node cfg = …
```

  - Insert a specified node n into a cfg after a specified node "target". This only works if the target has only one successor, as is the case with a pre-header.

```
> [CFGNode] -> CFG -> Int -> CFG
> insertNodesAfter nodes cfg target = …
```

  - Traverse the modified CFG and generate instructions:

```
> generateInstructions :: CFG -> [Instruction]
> generateInstructions cfg = …
```

Hoisting the loop-invariants…

- Finally, we bring it all together

```
> hoistLoopInvariants cfg looptree
>  = newcfg
>    where
>    newcfg = foldl hoistALoop cfgWithPreheaders loops
>    loops = [(h,l) | (h,l) <- loopsOf looptree, h /= 0]
>    (preheaders, cfgWithPreheaders) = addPreHeaders cfg looptree

>    hoistALoop cfg (header,body)
>     = insertNodesAfter invariants (removeNodes invariants cfg) preheader
>       where
>       invariants = loopInvariantInstructionsOf cfg (header:body)
>       preheader = retrieve header preheaders

> loopsOf (LTree (h,body) subloops)
>  = (h,body) : concat (map loopsOf subloops)
```

(This sketch implementation doesn’t check all the hoisting conditions…)

Example

AST

```
(Program
[(Decl "w" Integer),
 (Decl "x" Integer),
 (Decl "y" Integer),
 (Decl "z" Integer)]
[Assign (Var "x") (Const 1),
Assign (Var "w") (Const 100),
Assign (Var "z") (Const 200),
LabelStat "Here",
Assign (Var "x") (Binop Plus
          (Ref (Var "x")) (Const 1)),
Assign (Var "y") (Binop Plus
          (Ref (Var "w")) (Ref (Var "z"))),
IfThenElse (Compare CLT
          (Ref (Var "x")) (Const 10))
  [Goto "Here"] []
] )
```

Original control flow graph:

```
Node 0 (Mov (ImmNum 1) (Reg T1)) [T1] [] [1] []
Node 1 (Mov (ImmNum 100) (Reg T0)) [T0] [] [2] [0]
Node 2 (Mov (ImmNum 200) (Reg T3)) [T3] [] [3] [1]
Node 3 (Mov (Reg T1) (Reg T4)) [T4] [T1] [4] [2,15]
Node 4 (Add (ImmNum 1) (Reg T4)) [T4] [T4] [5] [3]
Node 5 (Mov (Reg T4) (Reg T1)) [T1] [T4] [6] [4]
Node 6 (Mov (Reg T3) (Reg T5)) [T5] [T3] [7] [5]
Node 7 (Mov (Reg T0) (Reg T6)) [T6] [T0] [8] [6]
Node 8 (Add (Reg T5) (Reg T6)) [T6] [T5,T6] [9] [7]
Node 9 (Mov (Reg T6) (Reg T2)) [T2] [T6] [10] [8]
Node 10 (Mov (Reg T1) (Reg T7)) [T7] [T1] [11] [9]
Node 11 (Mov (ImmNum 10) (Reg T8)) [T8] [] [12] [10]
Node 12 (Cmp (Reg T7) (Reg T8)) [] [T7,T8] [13] [11]
Node 13 (Blt "L1") [] [] [14,15] [12]
Node 14 (Bra "L2") [] [] [17] [13]
Node 15 (Bra "LHere") [] [] [3] [13]
Node 16 (Bra "L3") [] [] [17] []
Node 17 Halt [] [] [] [14,16]
```

- Relevant reaching definitions:

```
relevantReachingDefinitionsOf cfg =
        [(0,[]),
        (1,[]),
        (2,[]),
        (3,[0,5]),
        (4,[3]),
        (5,[4]),
        (6,[2]),
        (7,[1]),
        (8,[7,6]),
        (9,[8]),
        (10,[5]),
        (11,[]),
        (12,[11,10]),
        (13,[]),
        (14,[]),
        (15,[]),
        (16,[]),  (17,[])]
```

- Loop Tree:

```
loopTree cfg =
    LTree (0,[0,1,2,3,4,5,6,7,8,9,10,
              11,12,13,14,15,16,17])
     [LTree (3,[4,5,6,7,8,9,10,
                11,12,13,15])
              [] ]
```

- Loop invariants:

```
externallyDependentInstructionsOf cfg
    [3,4,5,6,7,8,9,10,11,12,13,15] =
[Node 6 (Mov (Reg T3) (Reg T5)) [T5] [T3] [7] [5],
Node 7 (Mov (Reg T0) (Reg T6)) [T6] [T0] [8] [6],
Node 11 (Mov (ImmNum 10) (Reg T8)) [T8] [] [12] [10]]

loopInvariantInstructionsOf (cfgex 15)
[3,4,5,6,7,8,9,10,11,12,13,15] =
 [Node 7 (Mov (Reg T0) (Reg T6)) [T6] [T0] [8] [6],
 Node 6 (Mov (Reg T3) (Reg T5)) [T5] [T3] [7] [5],
 Node 11 (Mov (ImmNum 10) (Reg T8)) [T8] [] [12] [10],
 Node 9 (Mov (Reg T6) (Reg T2)) [T2] [T6] [10] [8],
 Node 8 (Add (Reg T5) (Reg T6)) [T6] [T5,T6] [9] [7]]
```

- Code after loop-invariant hoisting:

```
move.l #1, T1
move.l #100, T0
move.l #200, T3
#Preheader for loop with header 3
move.l T0, T6
move.l T3, T5
move.l #10, T8
move.l T6, T2
add.l  T5, T6
```

(continued in next column…)

```
M3:
move.l T1, T4
add.l  #1, T4
move.l T4, T1
#Mov (Reg T3) (Reg T5) moved
#Mov (Reg T0) (Reg T6) moved
#Add (Reg T5) (Reg T6) moved
#Mov (Reg T6) (Reg T2) moved
move.l T1, T7
#Mov (ImmNum 10) (Reg T8) moved
cmp.l  T7, T8
blt    M15
bra    M14

M15:
bra    M3

M14:
bra    M17

M17:
halt
bra    M3
```

- Fine print:
  - For efficiency, it is better for the CFG to consist of basic blocks instead of individual instructions
  - For loop optimisations, we would do the DFA on the IR before instruction selection; it’s simpler and it avoids complications such a having only two-address instructions
  - See Appel pg388
- Credits: the primary source for these slides was Appel’s book. I also found it very useful to study the course notes of Liz White (George Mason University), Laurie Hendren (McGill University) and Chau-Wen Tseng (University of Maryland)