Iteration Variable:

i is an iteration variable if the only definition of i inside the loop occurs at an edge which separates the body and is of the form:

$$i = i + h;$$

for some loop constant h .

A loop constant is simply a constant (e.g., 42), or slightly more libaral, an expression which only depends on variables which are not modified during the loop :-)

(3) Differences for Sets

Consider the fixpoint computation:

$$
\begin{array}{l}
x = \emptyset; \\
\mathsf{for}\ \ (t = F\,x; t \not\subseteq x; \boxed{t = F\,x;}) \\
\qquad\qquad x = x \cup t;
\end{array}
$$

If F is distributive, it could be replaced by:

$$
\begin{array}{l}
x = \emptyset; \\
\mathsf{for}\ \ (\Delta = F\,x; \Delta \neq \emptyset; \boxed{\Delta = (F\,\Delta) \setminus x;}) \\
\qquad\qquad x = x \cup \Delta;
\end{array}
$$

The function F must only be computed for the smaller sets Δ :-)
semi-naive iteration

Instead of the sequence: $\emptyset \subseteq F(\emptyset) \subseteq F^2(\emptyset) \subseteq \ldots$

we compute: $\Delta_1 \cup \Delta_2 \cup \ldots$

where:

$$\begin{aligned} \Delta_{i+1} &= F(F^i(\emptyset)) \backslash F^i(\emptyset) \\ &= F(\Delta_i) \backslash (\Delta_1 \cup \ldots \cup \Delta_i) \quad \text{with } \Delta_0 = \emptyset \end{aligned}$$

Assume that the costs of $F\ x$ is $1 + \#x$.

Then the costs may sum up to:

naive	$1 + 2 + \ldots + n + n$	$=$	$\frac{1}{2}n(n+3)$
semi-naive			$2n$

where n is the cardinality of the result.

$\Longrightarrow$ A linear factor is saved :-)

2.2 Peephole Optimization

Idea:

- Slide a small window over the program.
- Optimize agressively inside the window, i.e.,
  - $\rightarrow$ Eliminate redundancies!
  - $\rightarrow$ Replace expensive operations inside the window by cheaper ones!

Examples:

$y = M[x]; x = x + 1; \quad \Longrightarrow \quad y = M[x{+}{+}];$

// given that there is a specific post-increment instruction :-)

$z = y - a + a; \quad \Longrightarrow \quad z = y;$

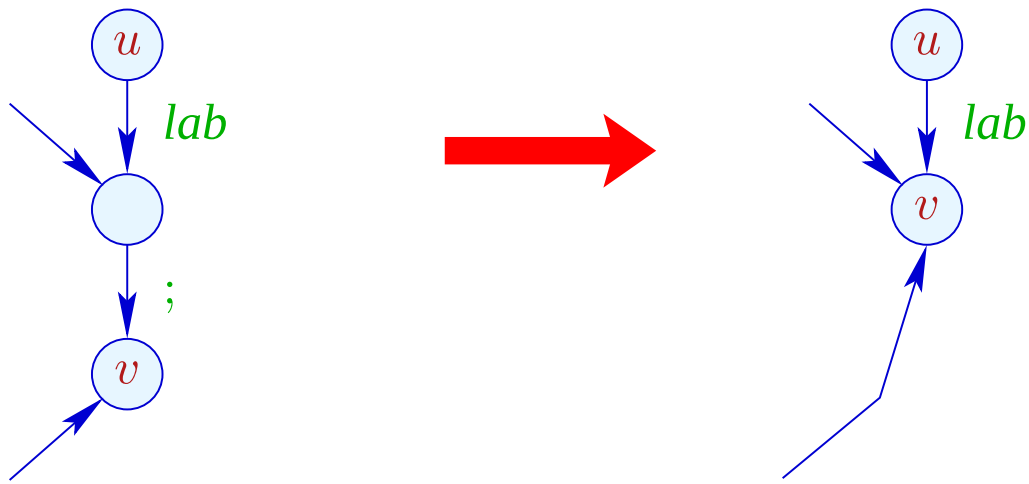
// algebraic simplifications :-)

$x = x; \quad \Longrightarrow \quad ;$

$x = 0; \quad \Longrightarrow \quad x = x \oplus x;$

$x = 2 \cdot x; \quad \Longrightarrow \quad x = x + x;$

Important Subproblem: *nop*-Optimization



→ If $(v_1, ;, v)$ is an edge, v_1 has no further out-going edge.

→ Consequently, we can identify v_1 and v :-)

→ The ordering of the identifications does not matter :-))

Implementation:

- We construct a function $\mathsf{next} : \mathit{Nodes} \to \mathit{Nodes}$ with:

$$\mathsf{next}\ u = \begin{cases} \mathsf{next}\ v & \text{if} \quad (u, ;, v) \quad \text{edge} \\ u & \text{otherwise} \end{cases}$$

  Warning: This definition is only recursive if there are ;-loops ???

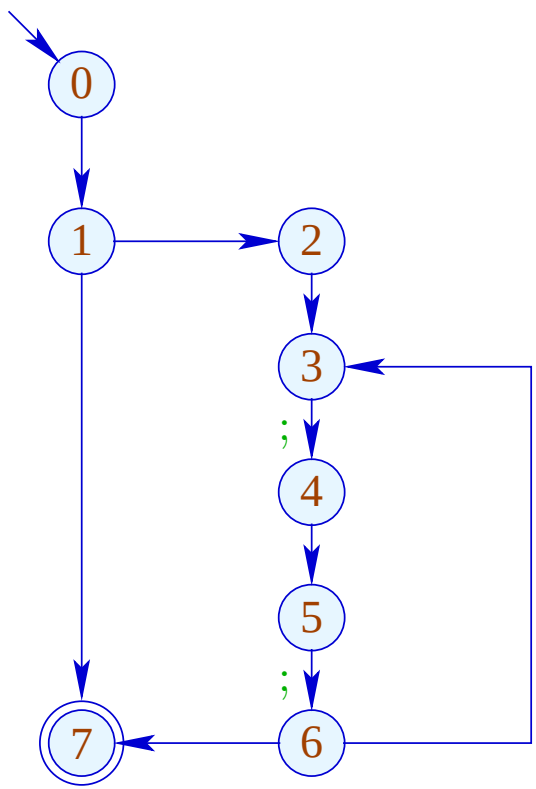
- We replace every edge:

$$(u, \mathit{lab}, v) \quad \Longrightarrow \quad (u, \mathit{lab}, \mathsf{next}\ v)$$

  ... whenever $\mathit{lab} \neq ;$

- All ;-edges are removed ;-)

Example:

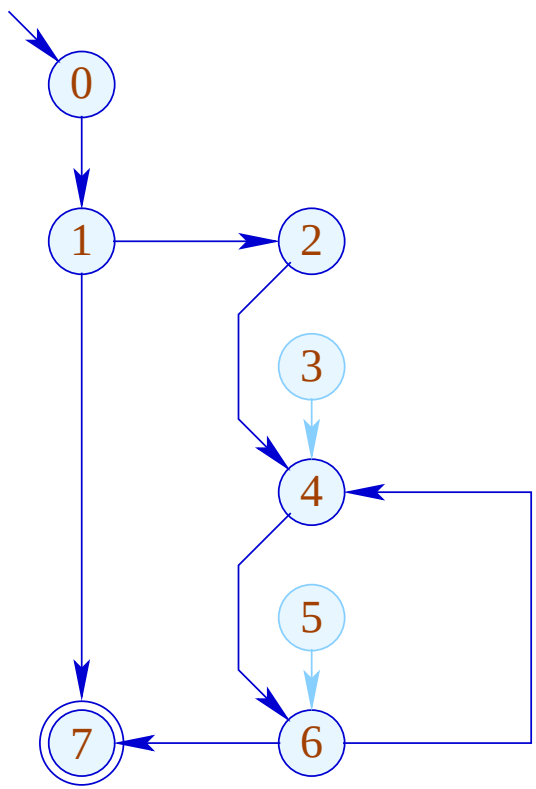


$$\text{next } 1 = 1$$

$$\text{next } 3 = 4$$

$$\text{next } 5 = 6$$

Example:



$$\text{next } 1 = 1$$

$$\text{next } 3 = 4$$

$$\text{next } 5 = 6$$

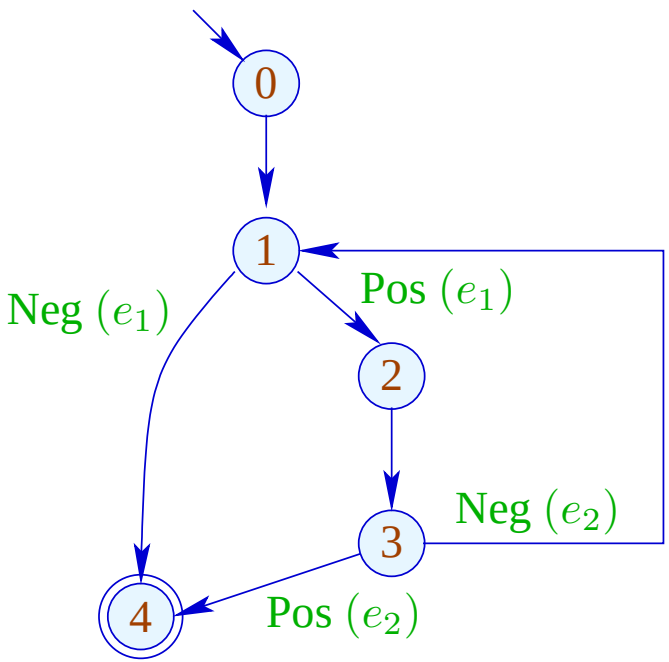
2. Subproblem: Linearization

After optimization, the CFG must again be brought into a linearly arrangement of instructions :-)

Warning:

Not every linearization is equally efficient !!!

Example:



0:

1: if (e_1) goto 2;

4: halt

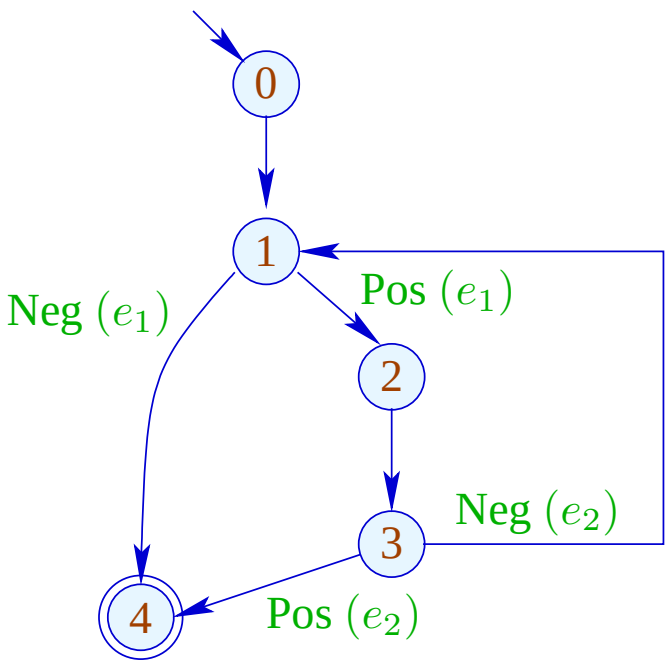
2: $\boxed{\text{Rumpf}}$

3: if (e_2) goto 4;

goto 1;

Bad: The loop body is jumped into :-(

Example:



0:

1: if $(!e_1)$ goto 4;

2: $\boxed{\text{Rumpf}}$

3: if $(!e_2)$ goto 1;

4: halt

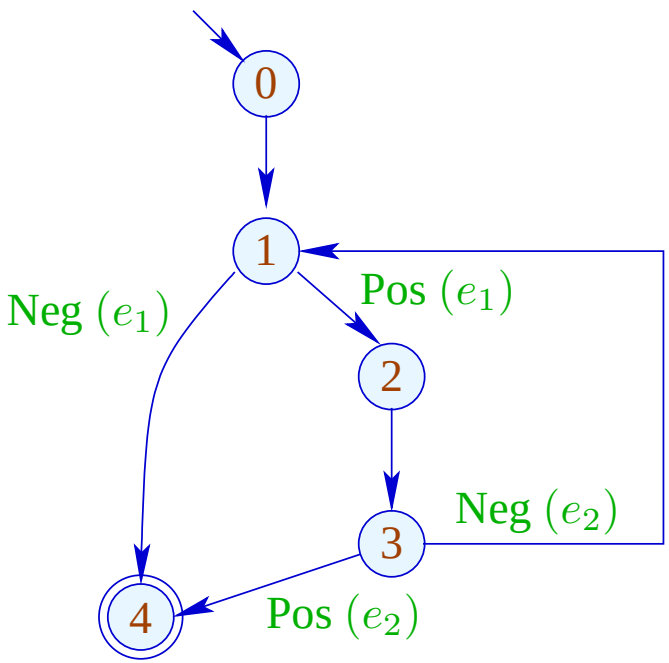
// better cache behavior :-)

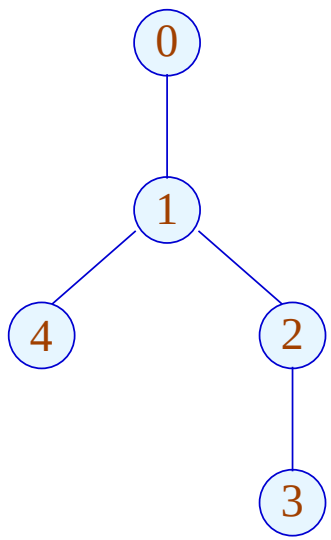
Idea:

- Assign to each node a temperature!
- always jumps to
  - (1) nodes which have already been handled;
  - (2) colder nodes.
- Temperature $\approx$ nesting-depth

For the computation, we use the pre-dominator tree and strongly connected components ...

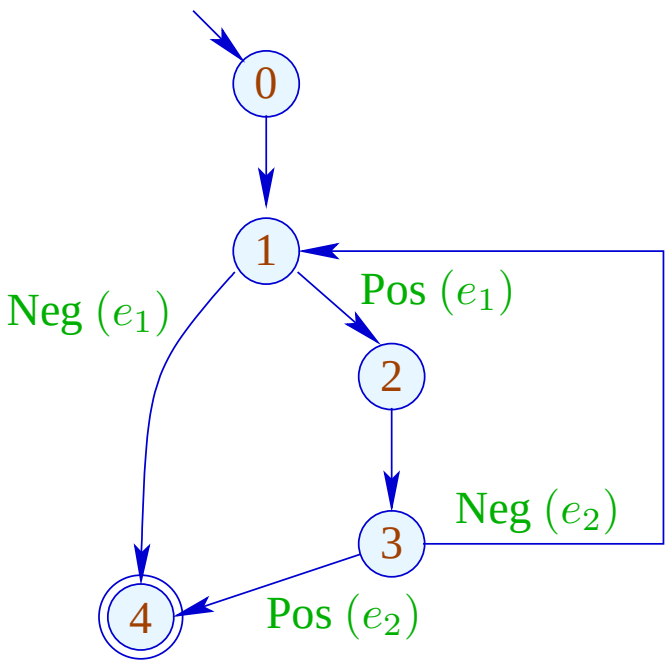
... in the Example:

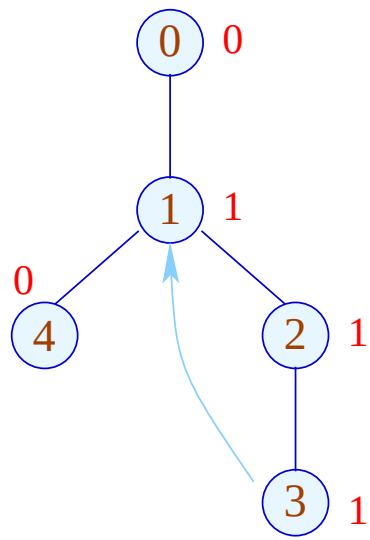




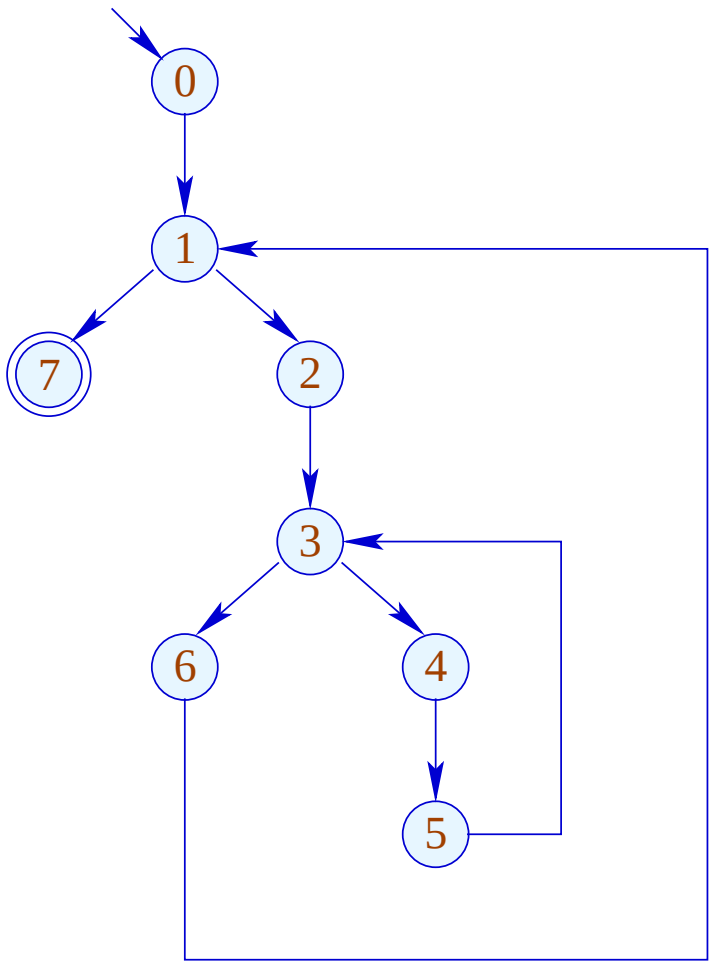
The sub-tree with back edge is hotter ...

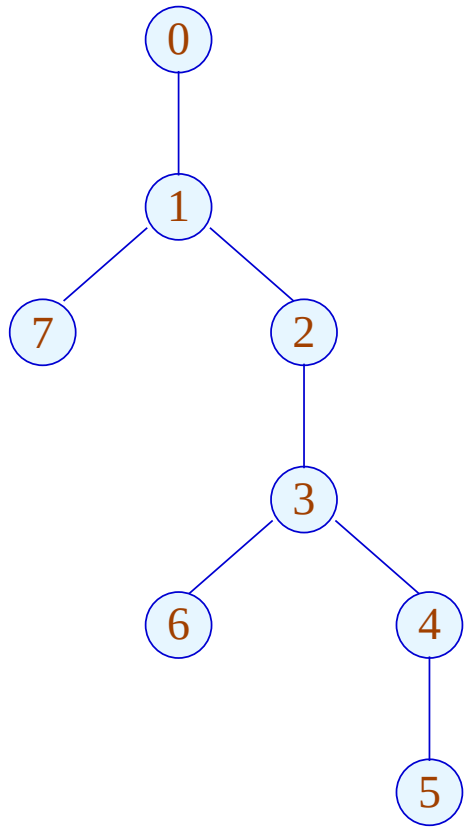
... in the Example:



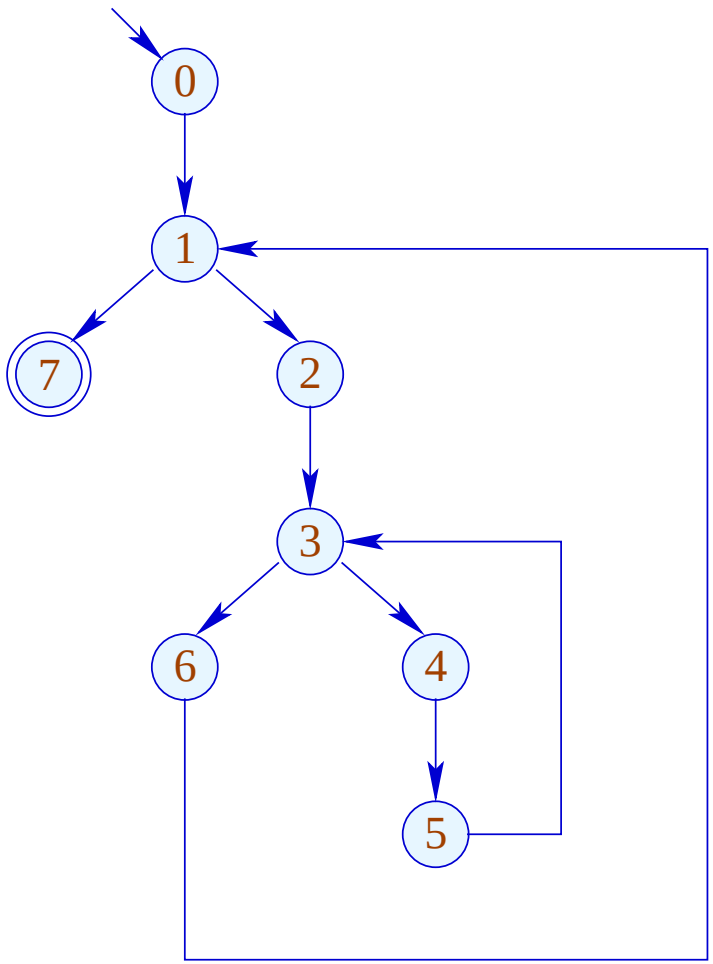


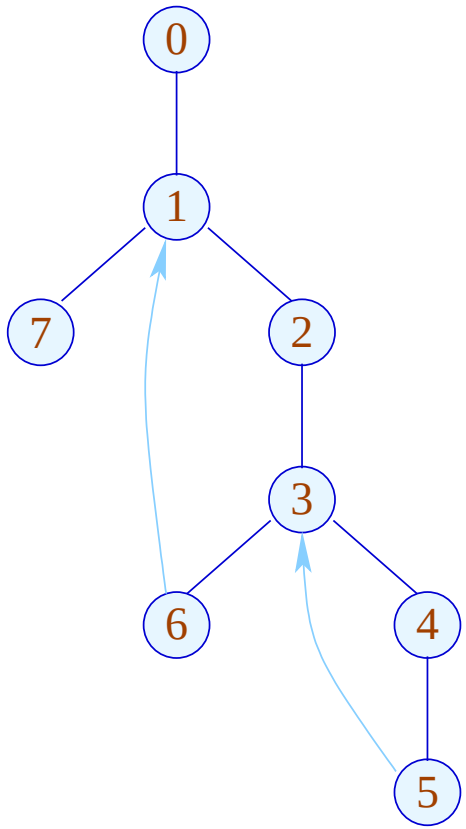
More Complicated Example:



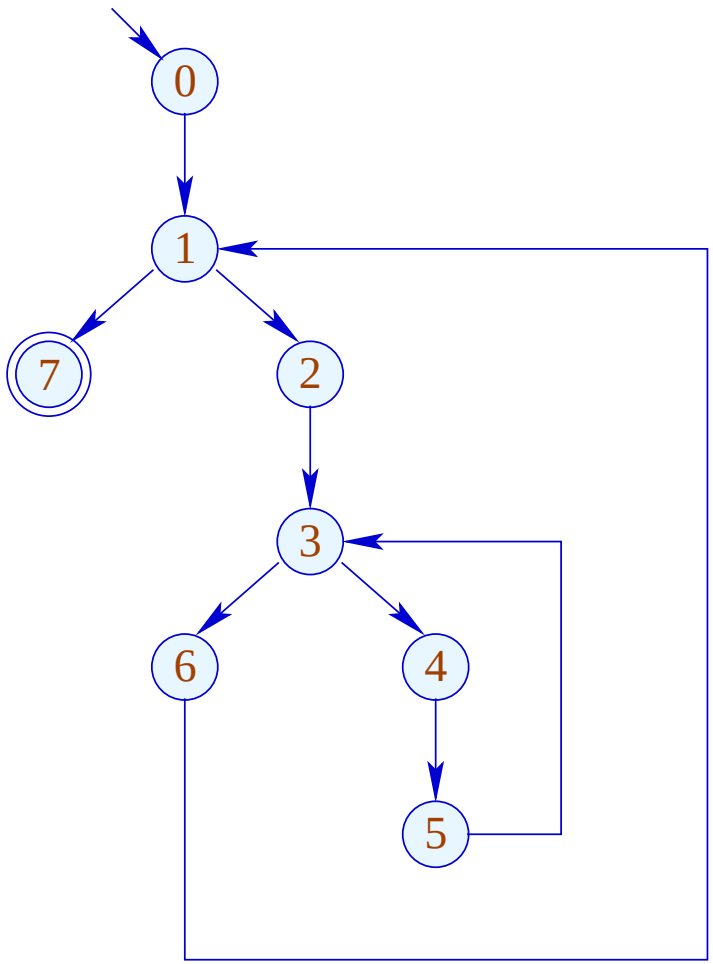


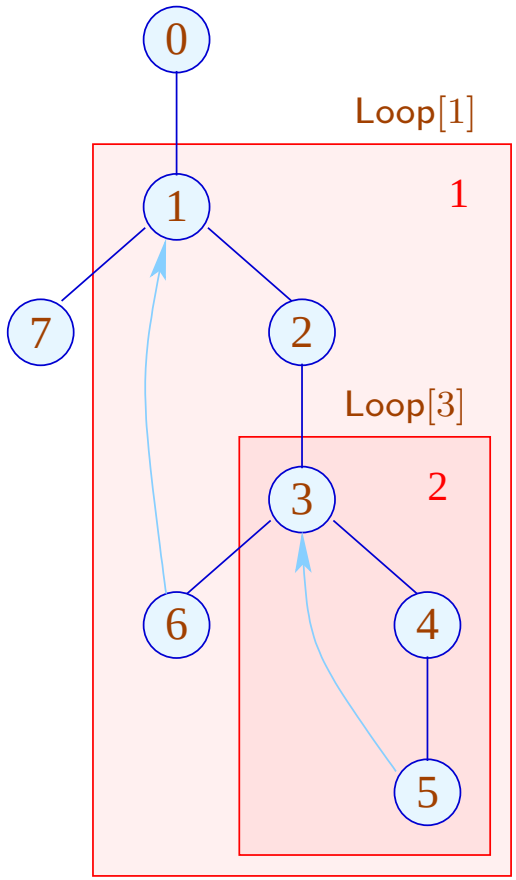
More Complicated Example:





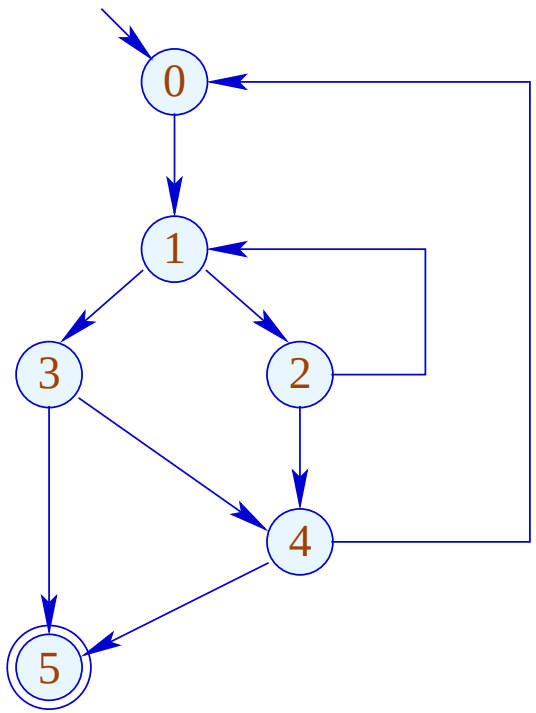
More Complicated Example:

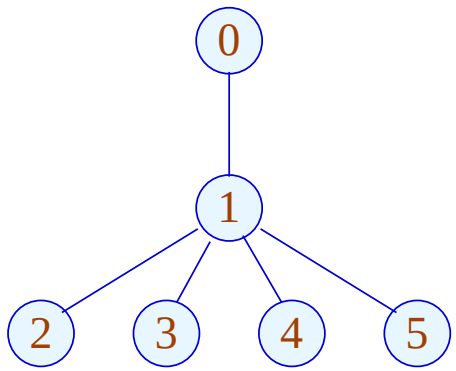




Our definition of Loop implies that (detected) loops are necessarily nested :-)

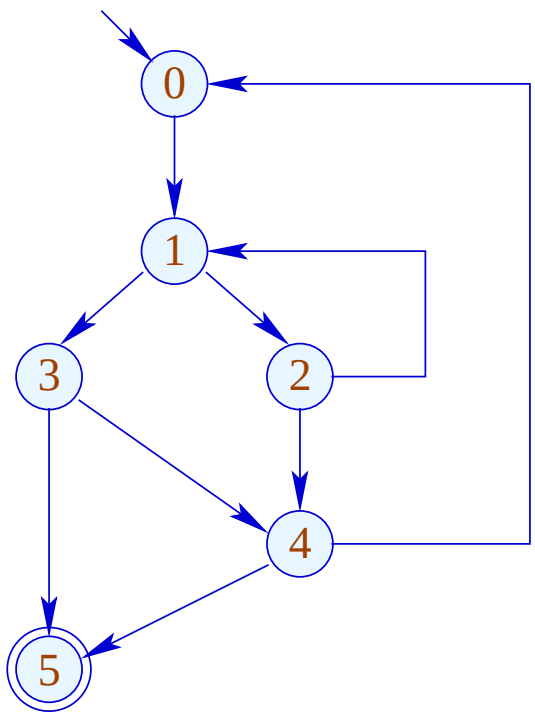
Is is also meaningful for do-while-loops with breaks ...

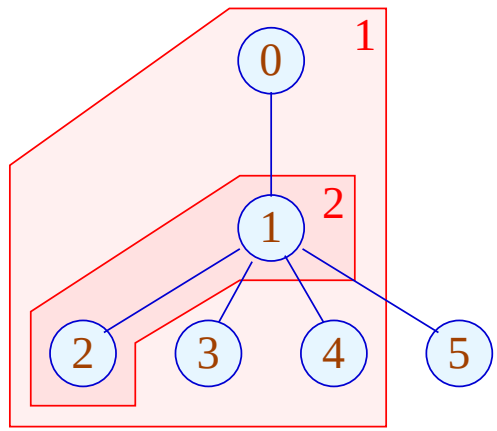




Our definition of Loop implies that (detected) loops are necessarily nested :-)

Is is also meaningful for do-while-loops with breaks ...





Summary: The Approach

(1) For every node, determine a temperature;

(2) Pre-order-DFS over the CFG;

- → If an edge leads to a node we already have generated code for, then we insert a jump.
- → If a node has two successors with different temperature, then we insert a jump to the colder of the two.
- → If both successors are equally warm, then it does not matter ;-)
- → At a join point, detect whether the current edge is a back edge (corresponding to a loop) or a cross edge (corresponding to a conditional).

2.3 Procedures

We extend our mini-programming language by procedures without parameters and procedure calls.

For that, we introduce a new statement:

$$f\,();$$

Every procedure f has a definition:

$$f\ ()\ \{\ \ stmt^*\ \ \}$$

Additionally, we distinguish between global and local variables.

Program execution starts with the call of a procedure $\mathsf{main}\,()$.

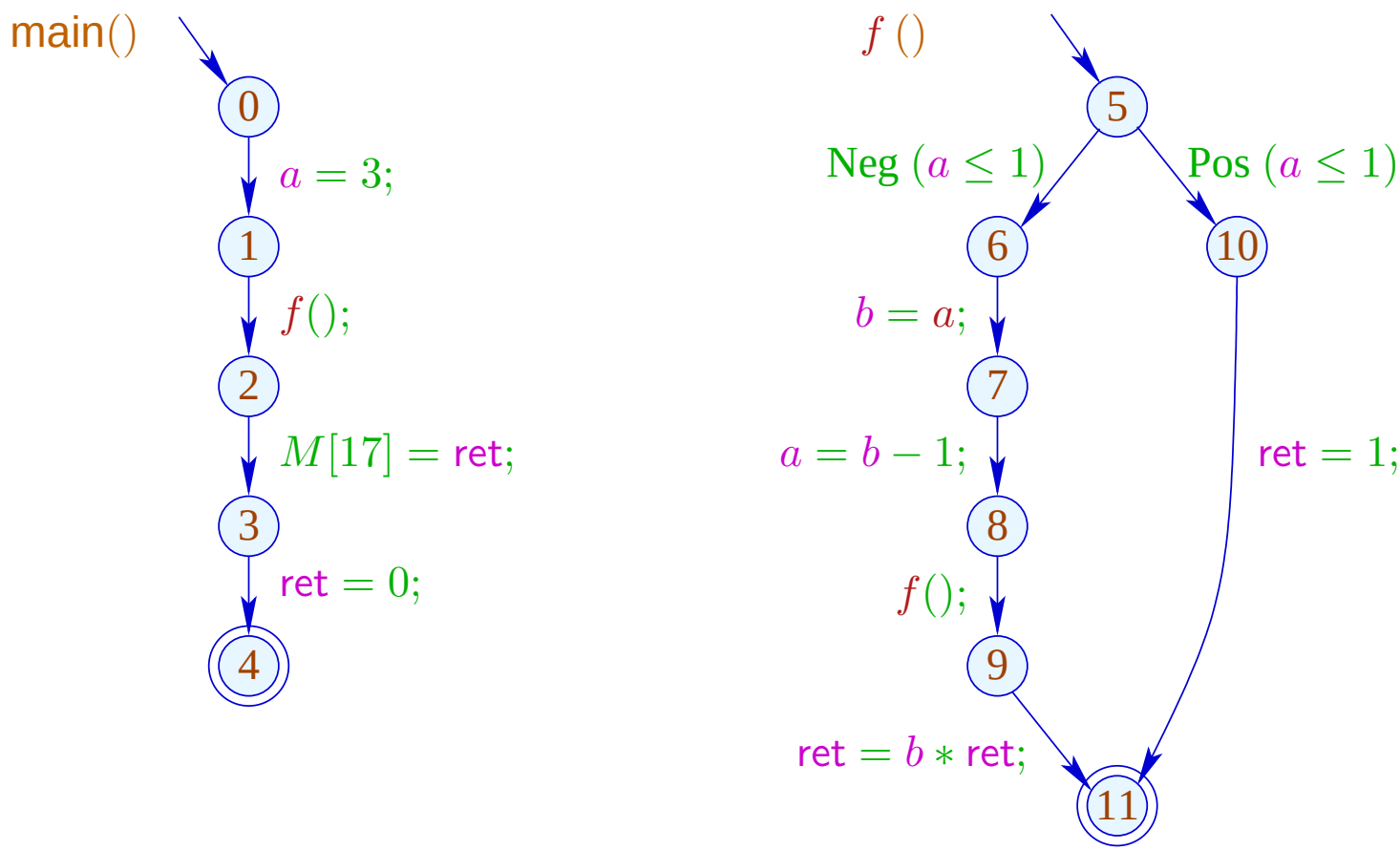
Example:

```
int a, ret;
main () {
    a = 3;
    f();
    M[17] = ret;
    ret = 0;
}
```

```
f () {
    int b;
    if (a ≤ 1) {ret = 1; goto exit;}
    b = a;
    a = b − 1;
    f();
    ret = b · ret;
exit :
}
```

Such programs can be represented by a set of CFGs: one for each procedure ...

... in the Example:



In order to optimize such programs, we require an extended operational semantics ;-)

Program executions are no longer paths, but forests: