2.2 Peephole Optimization

Idea:

- Slide a small window over the program.
- Optimize agressively inside the window, i.e.,

  → Eliminate redundancies!

  → Replace expensive operations inside the window by cheaper ones!

Examples:

$$x = x + 1; \qquad \Longrightarrow \qquad x{+}{+};$$

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

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

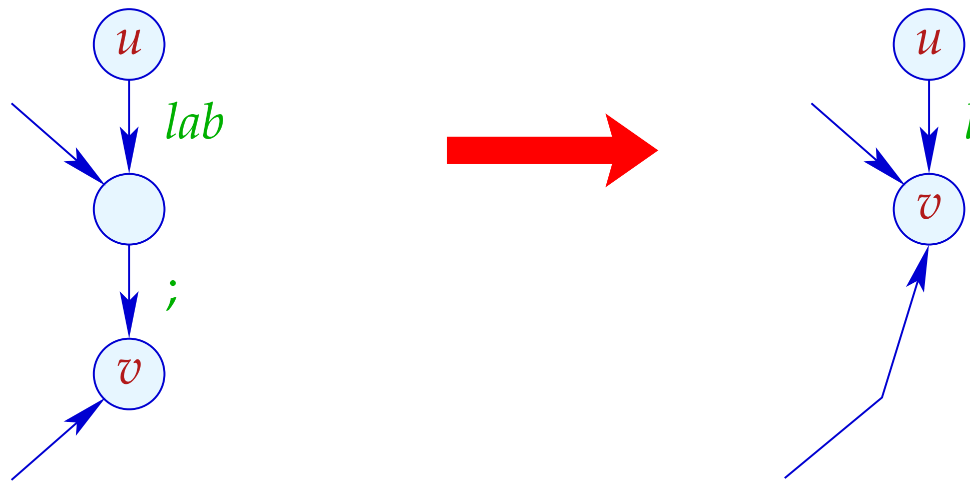
// algebraic simplifications :-)

$$x = x; \qquad \Longrightarrow \qquad ;$$

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

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

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 $\mathsf{next} : \mathit{Nodes} \rightarrow \mathit{Nodes}$ 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 ???

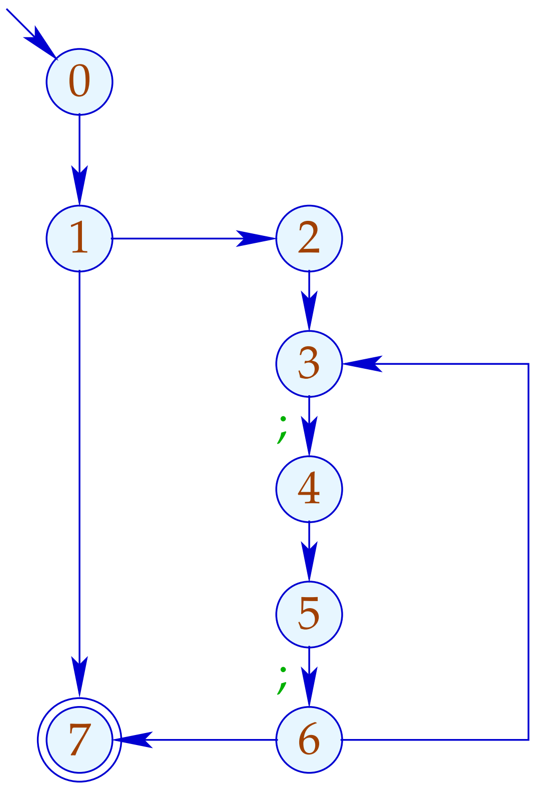
- We replace every edge:

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

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

- All ;-edges are removed ;-)

Example:

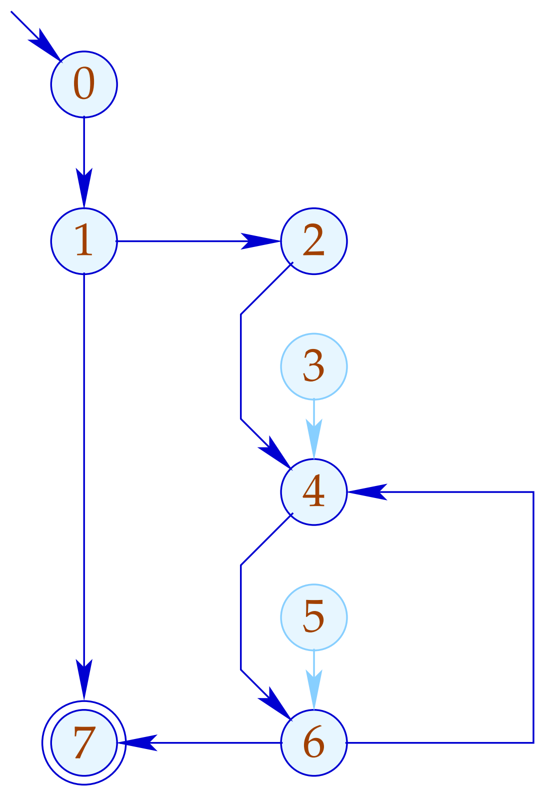


$$\text{next } 1 = 1$$

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

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

Example:



$$\text{next } 1 = 1$$
$$\text{next } 3 = 4$$
$$\text{next } 5 = 6$$

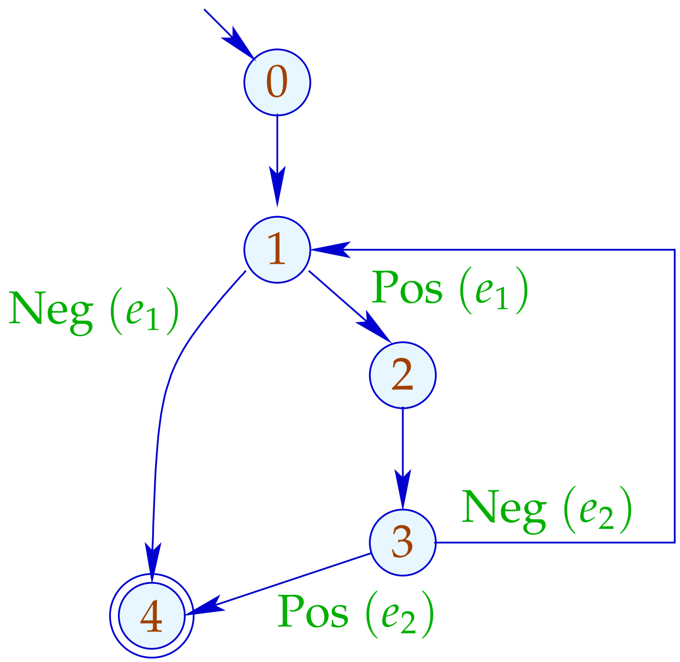
2. Subproblem: Linearization

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

Warning:

Not every linearization is equally efficient !!!

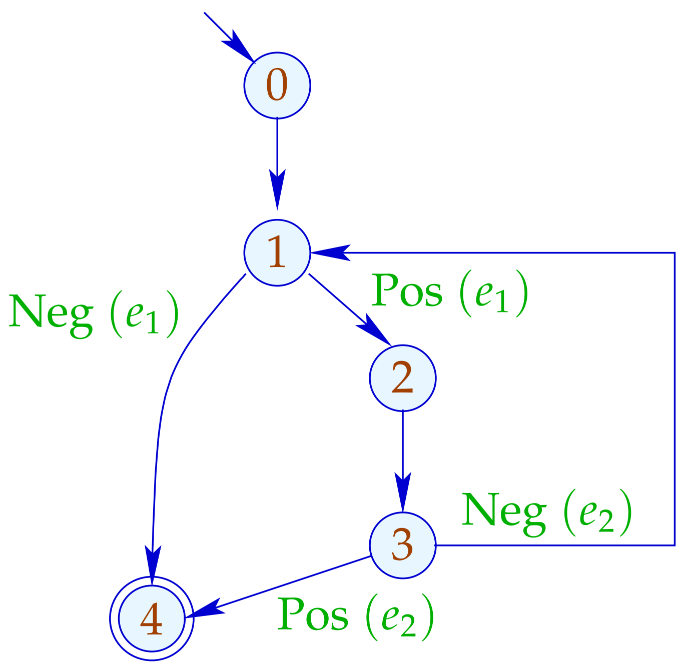
Example:



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

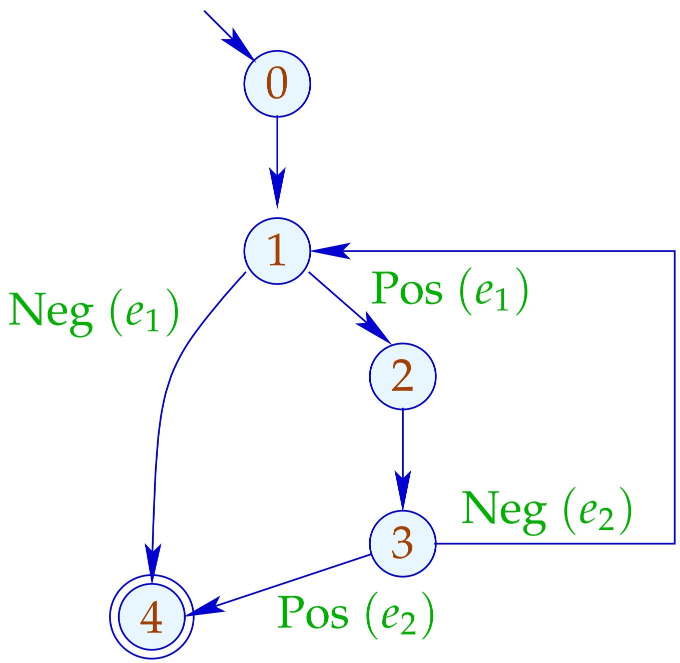
  (1) nodes which have already been handled;

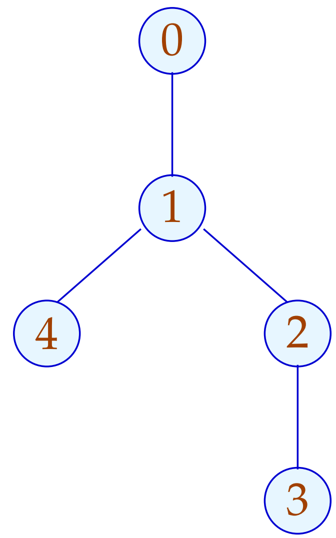
  (2) colder nodes.

- Temperature $\approx$ nesting-depth

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

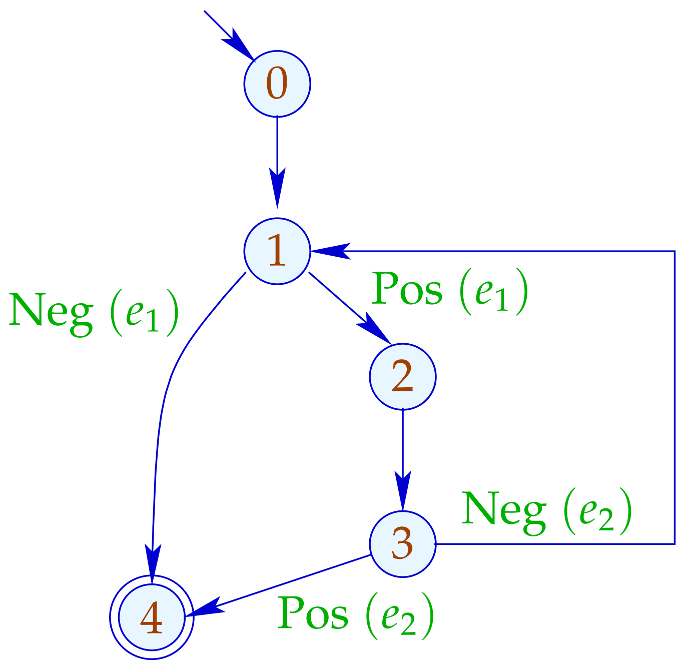
... in the Example:

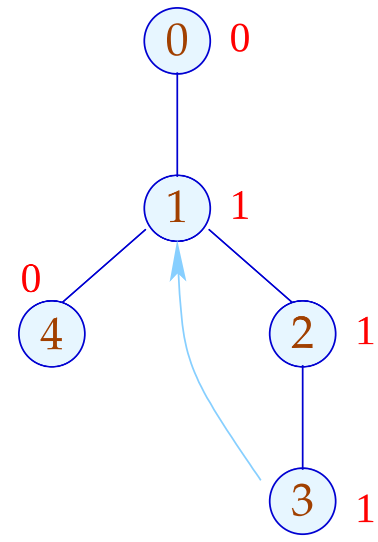




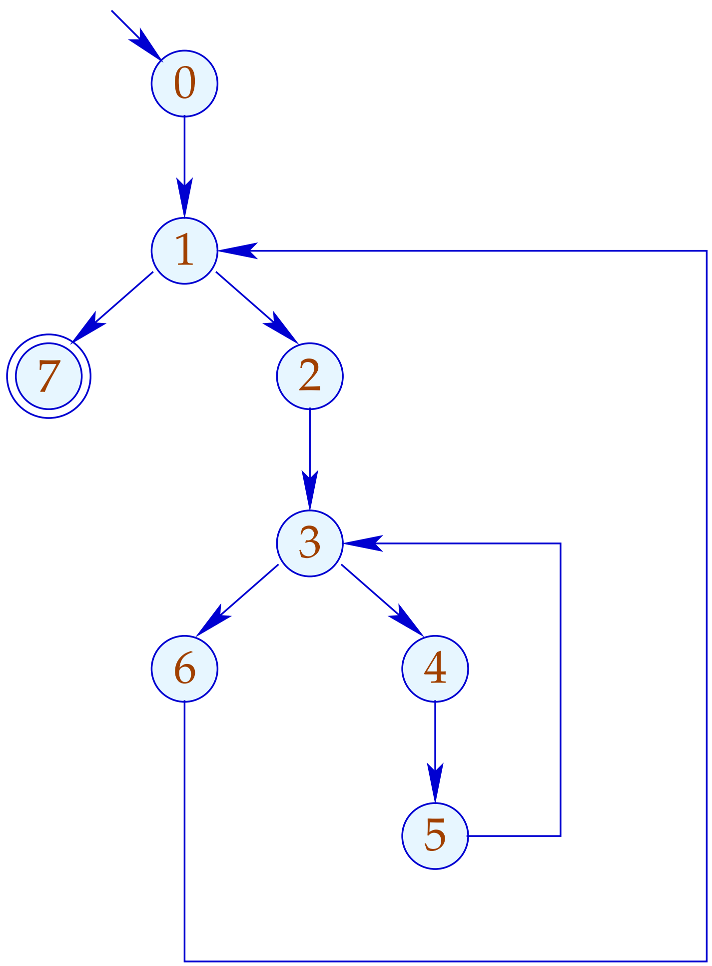
The sub-tree with back edge is hotter ...

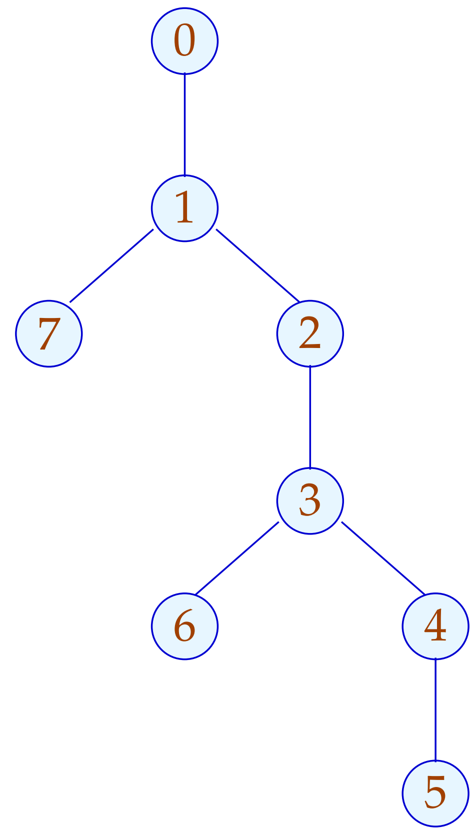
... in the Example:



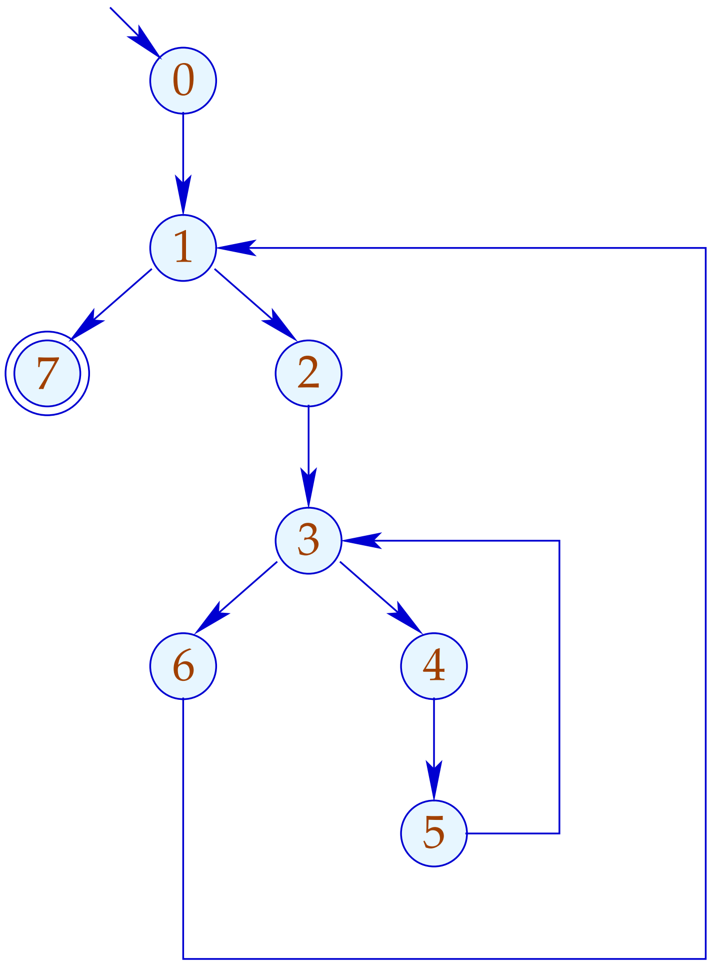


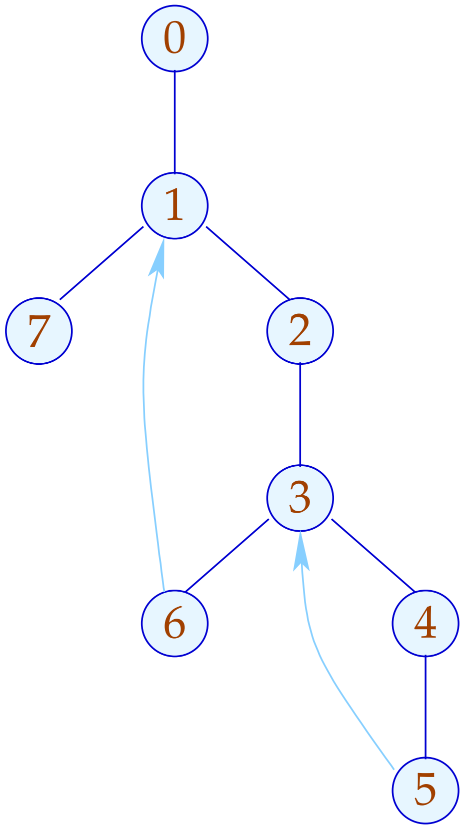
More Complicated Example:



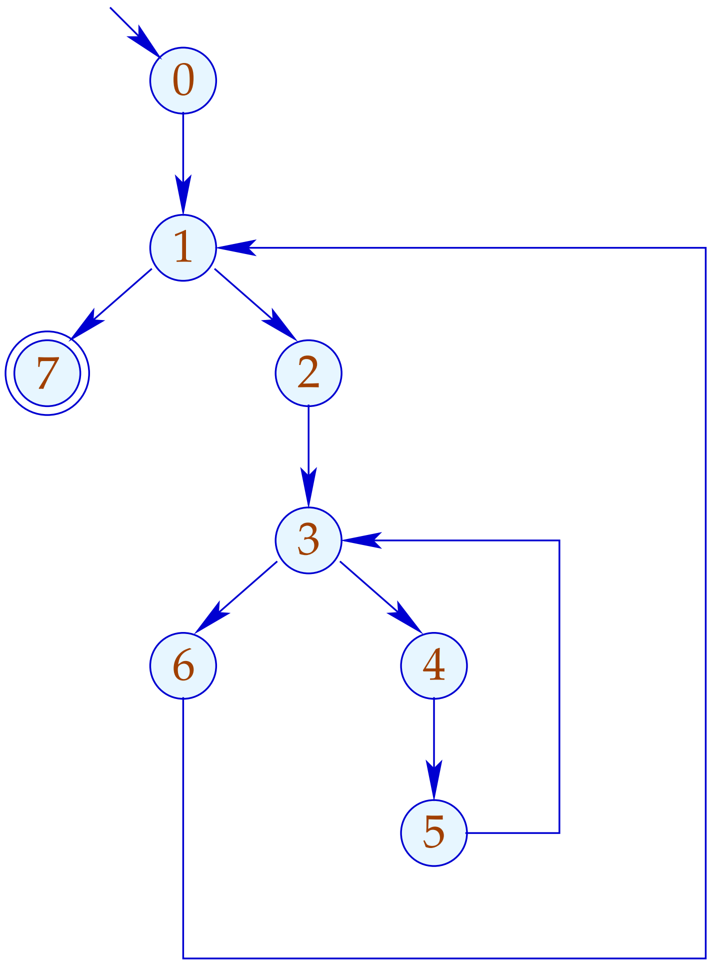


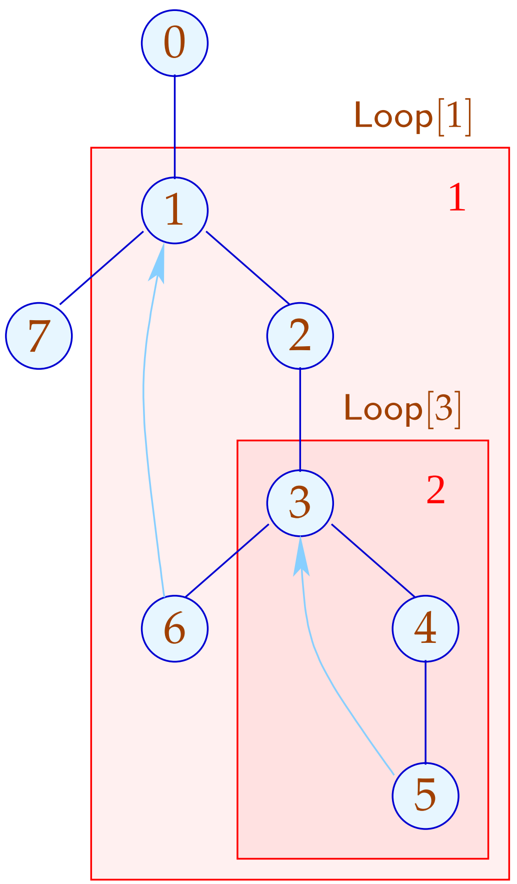
More Complicated Example:





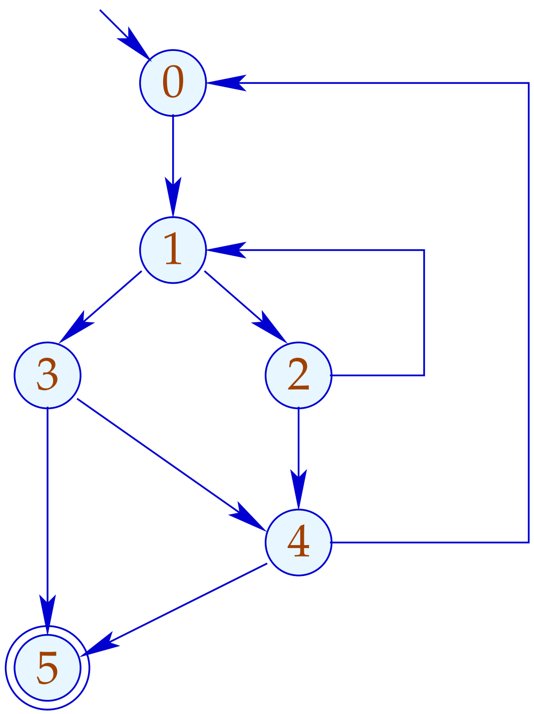
More Complicated Example:

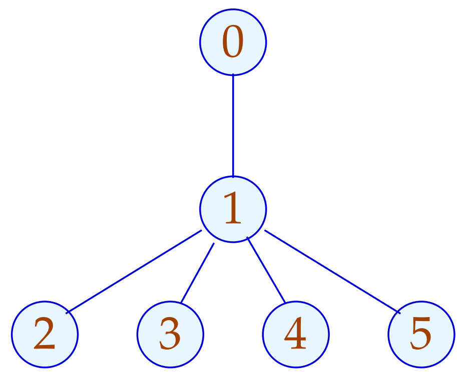




Our definition of Loop implies that (detected) loops are necessarily nested :-)

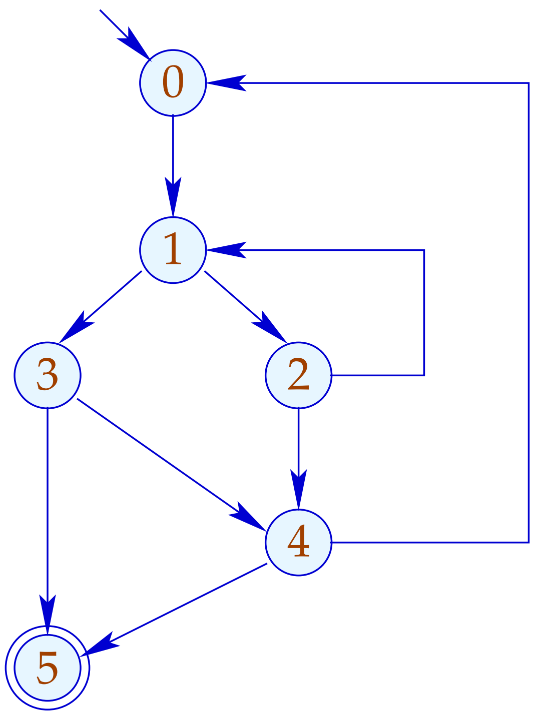
Is is also meaningful for do-while-loops with breaks ...

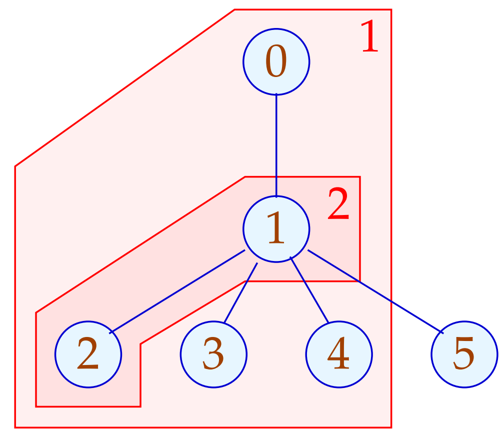




Our definition of Loop implies that (detected) loops are necessarily nested :-)

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 ;-)

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.

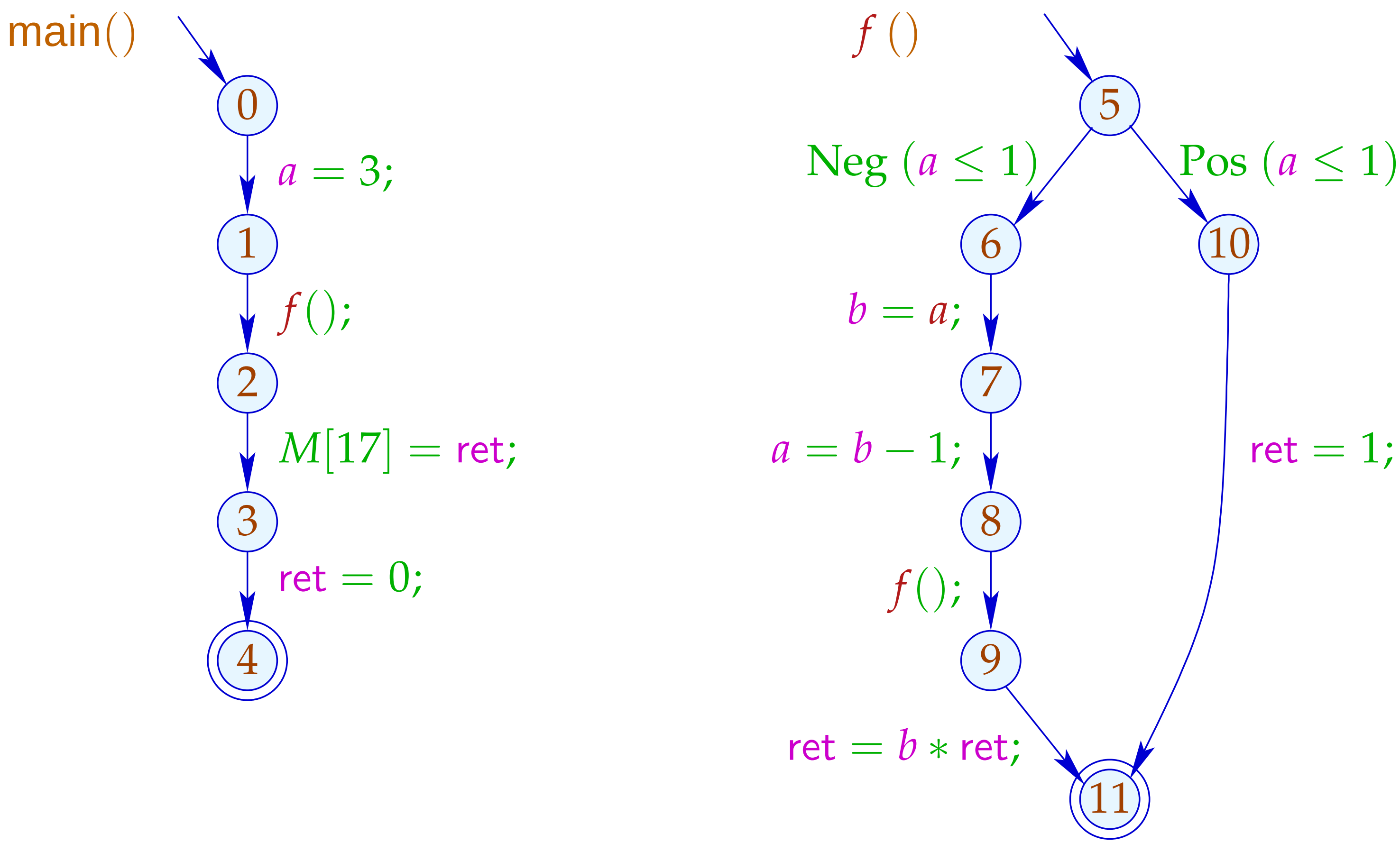
Program execution starts with the call of a procedure $\mathsf{main}\,()$.

Example:

```
int a, ret;                    f () {
main () {                           int b;
        a = 3;                      if (a ≤ 1) {ret = 1; goto exit; }
        f();                        b = a;
        M[17] = ret;                a = b − 1;
        ret = 0;                    f();
}                                   ret = b · ret;
                               exit :
                               }
```

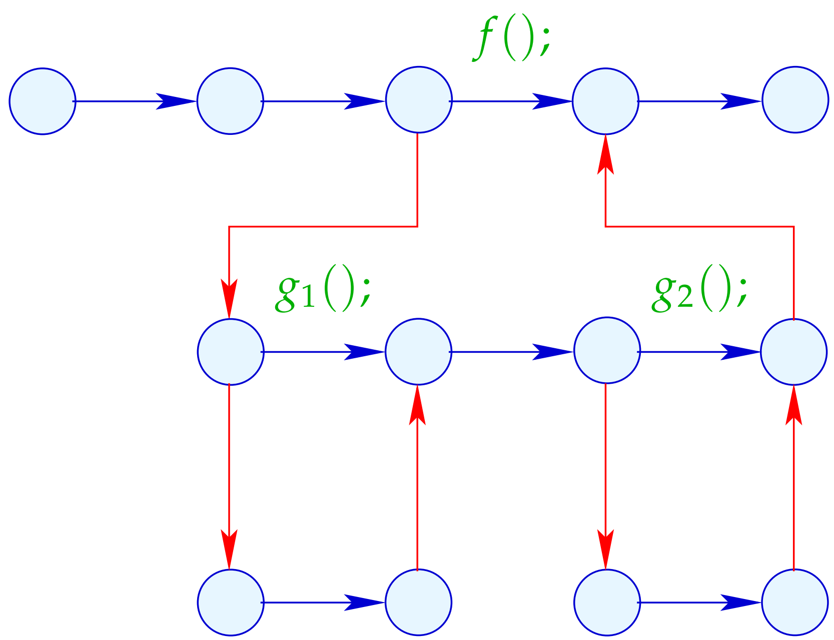
Such programs can be represented by a set of CFGs: one for each procedure ...

... in the Example:

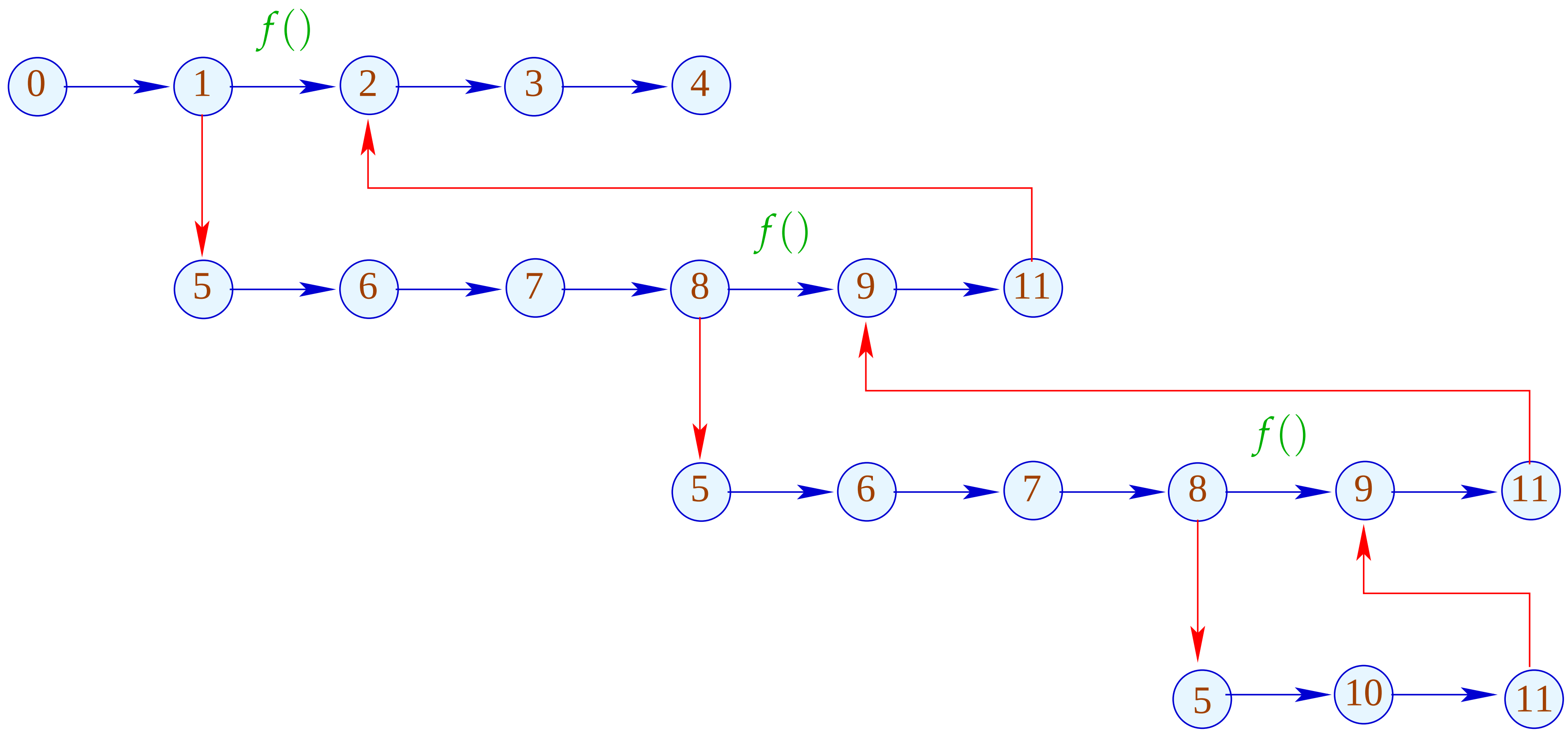


In order to optimize such programs, we require an extended operational semantics   ;-)

Program executions are no longer paths, but forests:



... in the Example:



The function $[\![.]\!]$ is extended to computation forests: w :

$$[\![w]\!] : (\mathit{Vars} \to \mathbb{Z}) \times (\mathbb{N} \to \mathbb{Z}) \to (\mathit{Vars} \to \mathbb{Z}) \times (\mathbb{N} \to \mathbb{Z})$$

For a call $k = (u, f();, v)$ we must:

- determine the initial values for the locals:

$$\textsf{enter}\ \rho = \{x \mapsto 0 \mid x \in \mathit{Locals}\} \oplus (\rho|_{\mathit{Globals}})$$

- ... combine the new values for the globals with the old values for the locals:

$$\textsf{combine}\ (\rho_1, \rho_2) = (\rho_1|_{\mathit{Locals}}) \oplus (\rho_2|_{\mathit{Globals}})$$

- ... evaluate the computation forest inbetween:

$$\begin{array}{lll} [\![k\ \langle w \rangle]\!]\ (\rho, \mu) & = & \textsf{let}\ \ (\rho_1, \mu_1) = [\![w]\!]\ (\textsf{enter}\ \rho, \mu) \\ & & \textsf{in}\ \ \ (\textsf{combine}\ (\rho, \rho_1), \mu_1) \end{array}$$

Warning:

- In general, $[\![w]\!]$ is only partially defined :-)
- Dedicated global/local variables a_i, b_i, ret can be used to simulate specific calling conventions.
- The standard operational semantics relies on configurations which maintain a call stack.
- Computation forests are better suited for the construction of analyses and correctness proofs :-)
- It is an awkward (but useful) exercise to prove the equivalence of the two approaches ...

Configurations:

$$
\begin{array}{lcl}
\textit{configuration} & == & \textit{stack} \times \textit{store} \\
\textit{store} & == & \textit{globals} \times \mathbb{N} \to \mathbb{Z} \\
\textit{locals} & == & (\textit{Globals} \to \mathbb{Z}) \\
\textit{stack} & == & \textit{frame} \cdot \textit{frame}^* \\
\textit{frame} & == & \textit{point} \times \textit{locals} \\
\textit{locals} & == & (\textit{Locals} \to \mathbb{Z})
\end{array}
$$

A *frame* specifies the local state of computation inside a procedure call :-)

The leftmost frame corresponds to the current call.

Computation steps refer to the current call :-)

The novel kinds of steps:

call $k = (u, f\,();, v)$:

$$(\boxed{(u,\rho)} \cdot \sigma, \langle\gamma, \mu\rangle) \quad \Longrightarrow \quad (\boxed{(u_f, \{x \to 0 \mid x \in \mathit{Locals}\}) \cdot (v, \rho)} \cdot \sigma, \langle\gamma, \mu\rangle)$$

u_f entry point of f

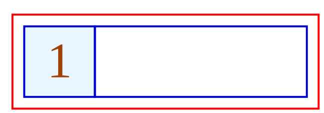
return:

$$(\boxed{(r_f, \_)} \cdot \sigma, \langle\gamma, \mu\rangle) \quad \Longrightarrow \quad (\sigma, \langle\gamma, \mu\rangle)$$

r_f return point of f

The call stack explicitly implements the DFS traversal through the computation forest :-)

... in the Example:



The call stack explicitly implements the DFS traversal through the computation forest :-)

... in the Example:

5	$b \mapsto 0$
2	

The call stack explicitly implements the DFS traversal through the computation forest :-)

... in the Example:

7	$b \mapsto 3$
2	

The call stack explicitly implements the DFS traversal through the computation forest :-)

... in the Example:

5	$b \longmapsto 0$
9	$b \longmapsto 3$
2	

The call stack explicitly implements the DFS traversal through the computation forest :-)

... in the Example:

7	$b \mapsto 2$
9	$b \mapsto 3$
2	

The call stack explicitly implements the DFS traversal through the computation forest :-)

... in the Example:

5	$b \longmapsto 0$
9	$b \longmapsto 2$
9	$b \longmapsto 3$
2	

The call stack explicitly implements the DFS traversal through the computation forest :-)

... in the Example:

11	$b \mapsto 0$
9	$b \mapsto 2$
9	$b \mapsto 3$
2	

The call stack explicitly implements the DFS traversal through the computation forest :-)

... in the Example:

9	$b \mapsto 2$
9	$b \mapsto 3$
2	

The call stack explicitly implements the DFS traversal through the computation forest :-)

... in the Example:

11	$b \mapsto 2$
9	$b \mapsto 3$
2	

The call stack explicitly implements the DFS traversal through the computation forest :-)

... in the Example:

9	$b \mapsto 3$
2	

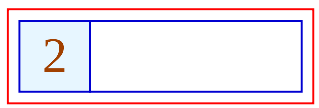
The call stack explicitly implements the DFS traversal through the computation forest :-)

... in the Example:

11	$b \mapsto 3$
2	

The call stack explicitly implements the DFS traversal through the computation forest :-)

... in the Example:



This operational semantics is quite realistic :-)

Costs for a Procedure Call:

Before entering the body:

- Creating a stack frame;
- assing of the parameters;
- Saving the registers;
- Saving the return address;
- Jump to the body.

At procedure exit:

- Freeing the stack frame.
- Restoring the registers.
- Passing of the result.
- Return behind the call.

$\Longrightarrow$ ... quite expensive !!!

1. Idea: Inlining

Copy the procedure body at every call site !!!

Example:

$$
\begin{array}{ll}
abs\;()\;\{ & max\;()\;\{ \\
\quad a_2 = -a_1; & \quad \text{if}\;(a_1 < a_2)\;\{\;\text{ret} = a_2;\;\text{goto}\;\_exit;\;\} \\
\quad max\;(); & \quad \text{ret} = a_1; \\
\} & \_exit\;: \\
 & \}
\end{array}
$$

... yields:

$$
\begin{array}{l}
abs\ ()\ \{ \\
a_2 = -a_1; \\
\boxed{\begin{array}{l}
\qquad \mathsf{if}\ (a_1 < a_2)\ \{\ \mathsf{ret} = a_2;\ \mathsf{goto}\ \_exit;\ \} \\
\qquad \mathsf{ret} = a_1; \\
\_exit :
\end{array}} \\
\}
\end{array}
$$

Problems:

- The copied block may modify the locals of the calling procedure ???
- More general: Multiple use of local variable names may lead to errors.
- Multiple calls of a procedure may lead to code duplication :-((
- How can we handle recursion ???

Detection of Recursion:

We construct the call-graph of the program.

In the Examples: