2 Replacing Expensive Operations by Cheaper Ones

2.1 Reduction of Strength

(1) Evaluation of Polynomials

$$f(x) = a_n \cdot x^n + a_{n-1} \cdot x^{n-1} + \ldots + a_1 \cdot x + a_0$$

	Multiplications	Additions
naive	$\frac{1}{2}n(n+1)$	n
re-use	$2n-1$	n
Horner-Scheme	n	n

Idea:

$$f(x) = (\ldots((a_n \cdot x + a_{n-1}) \cdot x + a_{n-2}) \ldots) \cdot x + a_0$$

(2) Tabulation of a polynomial $f(x)$ of degree n :

→ To recompute $f(x)$ for every argument x is too expensive :-)

→ Luckily, the n-th differences are constant !!!

Example: $f(x) = 3x^3 - 5x^2 + 4x + 13$

n	$f(n)$	Δ	Δ^2	Δ^3
0	13	2	8	$\boxed{18}$
1	15	10	$\boxed{26}$	
2	25	$\boxed{36}$		
3	$\boxed{61}$			
4	...			

Here, the n-th difference is always

$$\Delta_h^n(f) = n! \cdot a_n \cdot h^n \qquad (h \text{ step width})$$

Costs:

- n times evaluation of f ;
- $\frac{1}{2} \cdot (n-1) \cdot n$ subtractions to determine the Δ^k ;
- n additions for every further value :-)

$$\Longrightarrow$$

Number of multiplications only depends on n :-))

Simple Case: $f(x) = a_1 \cdot x + a_0$

- ... naturally occurs in many numerical loops :-)
- The first differences are already constant:

$$f(x+h) - f(x) = a_1 \cdot h$$

- Instead of the sequence: $y_i = f(x_0 + i \cdot h), \quad i \geq 0$

  we compute: $y_0 = f(x_0), \quad \Delta = a_1 \cdot h$

  $y_i = y_{i-1} + \Delta, \quad i > 0$

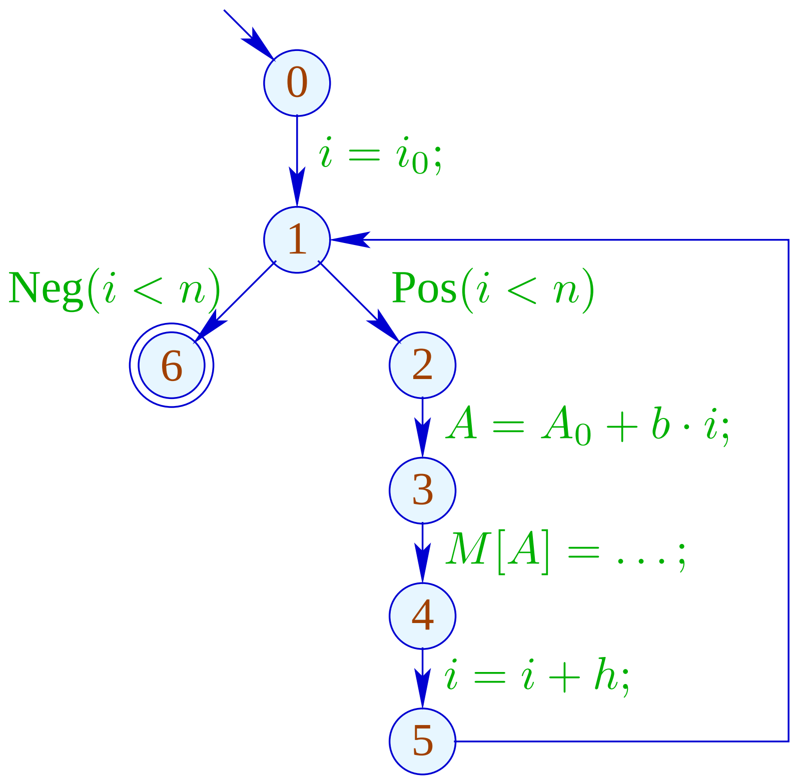
Example:

for $(i = i_0; i < n; i = i + h)\ \{$

$$A = A_0 + b \cdot i;$$

$$M[A] = \ldots;$$

$\}$



... or, after loop rotation:

$i = i_0;$

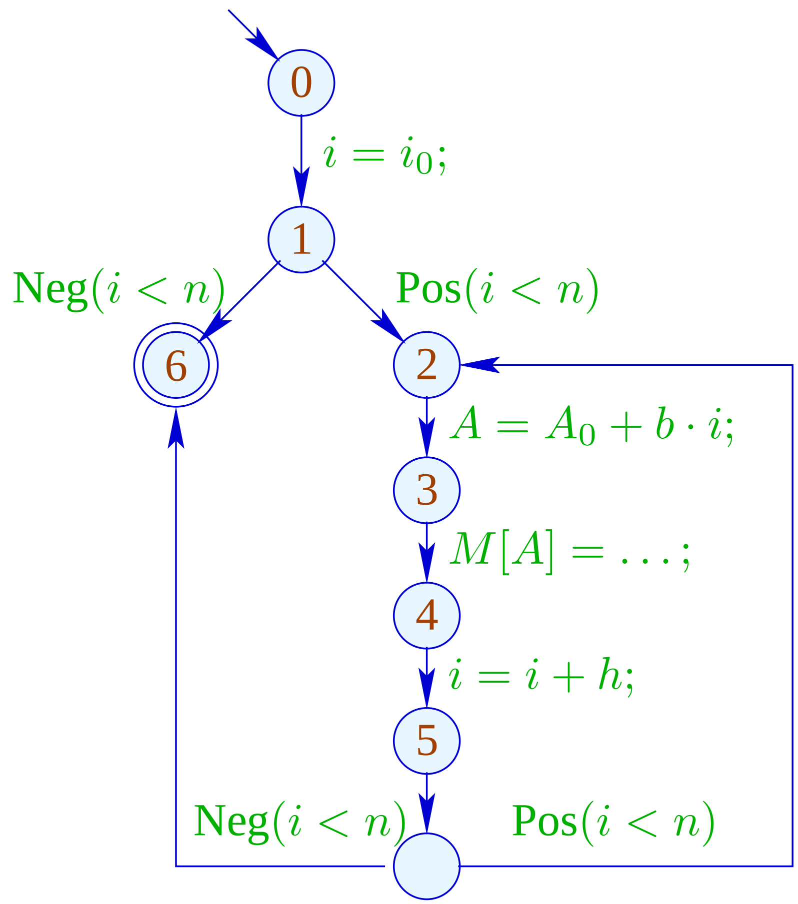
if $(i < n)$ do $\{$

$A = A_0 + b \cdot i;$

$M[A] = \ldots;$

$i = i + h;$

$\}$ while $(i < n);$



... and reduction of strength:

$i = i_0;$

if $(i < n)$ {

$\quad \Delta = b \cdot h;$

$\quad A = A_0 + b \cdot i_0;$

$\quad$ do {

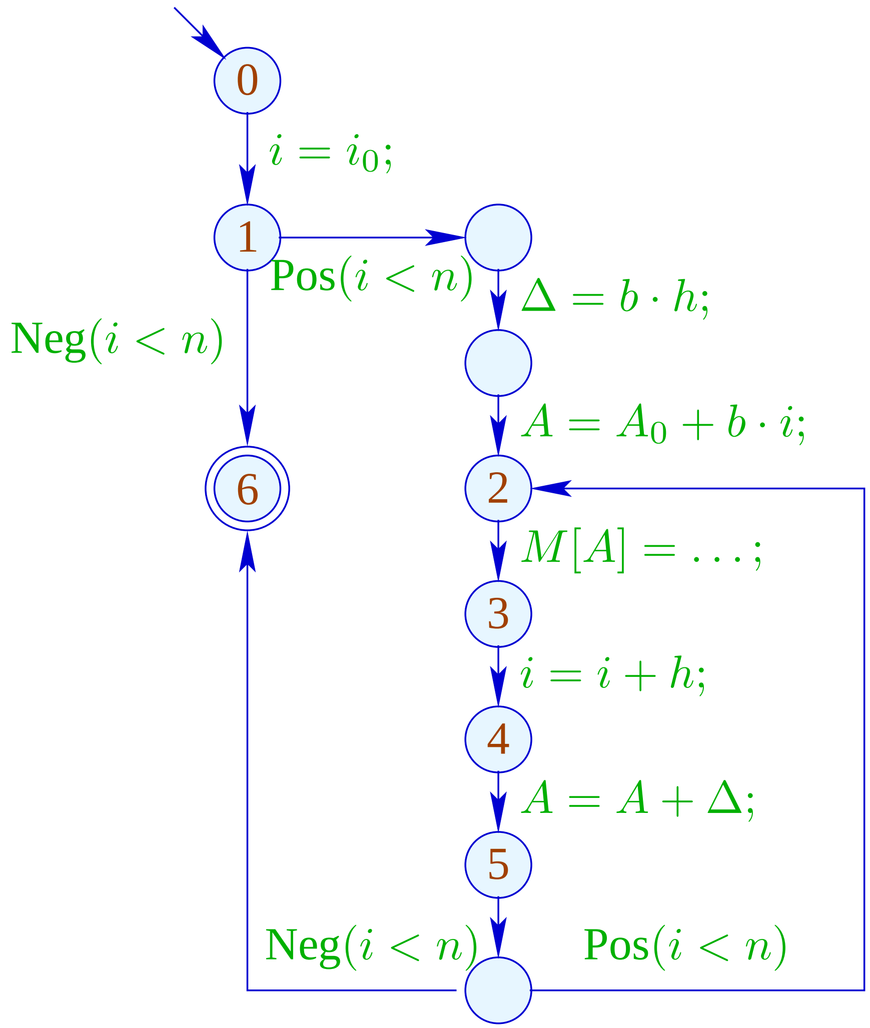
$\quad\quad M[A] = \ldots;$

$\quad\quad i = i + h;$

$\quad\quad A = A + \Delta;$

$\quad$ } while $(i < n);$

}



Warning:

- The values b, h, A_0 must not change their values during the loop.
- i, A may be modified at exactly one position in the loop :-(
- One may try to eliminate the variable i altogether :

  $\rightarrow$ i may not be used else-where.

  $\rightarrow$ The initialization must be transformed into:
  $A = A_0 + b \cdot i_0$.

  $\rightarrow$ The loop condition $i < n$ must be transformed into:
  $A < N$ for $N = A_0 + b \cdot n$.

  $\rightarrow$ b must always be different from zero !!!

Approach:

Identify

... loops;

... iteration variables;

... constants;

... the matching use structures.

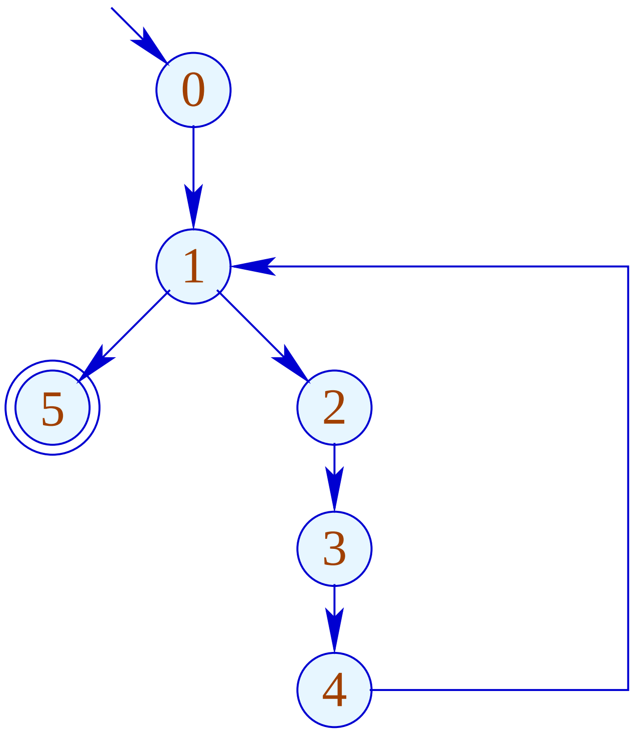
Loops:

... are identified through the node v with back edge $(\_, \_, v)$:-)

For the sub-graph G_v of the cfg on $\{w \mid v \Rightarrow w\}$, we define:

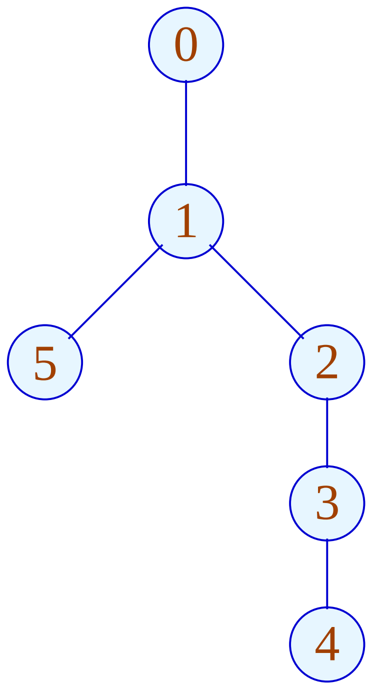
$$\mathsf{Loop}[v] \quad = \quad \{w \mid w \to^* v \;\; \text{in} \;\; G_v\}$$

Example:



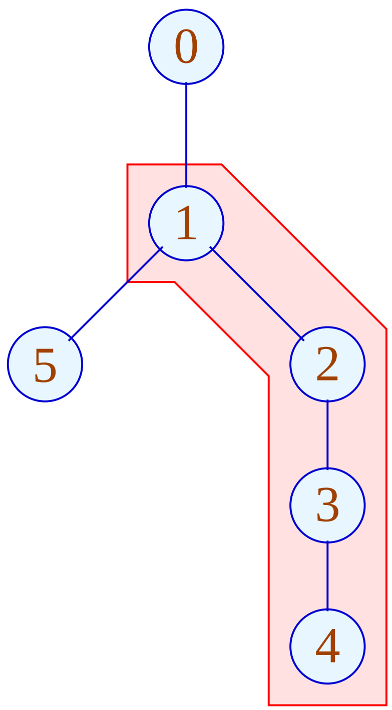
	$\mathcal{P}$
0	$\{0\}$
1	$\{0, 1\}$
2	$\{0, 1, 2\}$
3	$\{0, 1, 2, 3\}$
4	$\{0, 1, 2, 3, 4\}$
5	$\{0, 1, 5\}$

Example:



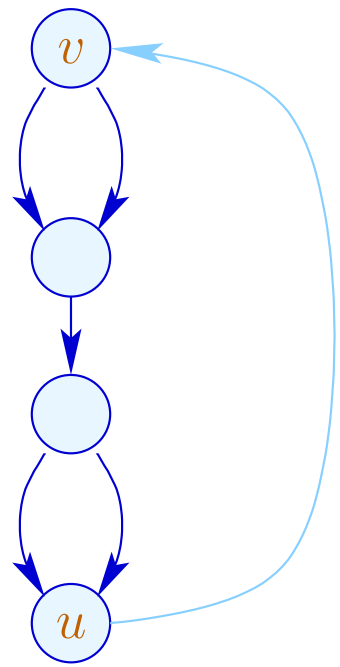
	$\mathcal{P}$
0	$\{0\}$
1	$\{0, 1\}$
2	$\{0, 1, 2\}$
3	$\{0, 1, 2, 3\}$
4	$\{0, 1, 2, 3, 4\}$
5	$\{0, 1, 5\}$

Example:



	$\mathcal{P}$
0	$\{0\}$
1	$\{0, 1\}$
2	$\{0, 1, 2\}$
3	$\{0, 1, 2, 3\}$
4	$\{0, 1, 2, 3, 4\}$
5	$\{0, 1, 5\}$

We are interested in edges which during each iteration are executed exactly once:



This property can be expressed by means of the pre-dominator relation ...

Assume that $(u, \_, v)$ is the back edge.

Then edges $\quad k = (u_1, \_, v_1) \quad$ could be selected such that:

- v pre-dominates u_1;
- u_1 pre-dominates v_1;
- v_1 predominates u.

Assume that $(u, \_, v)$ is the back edge.

Then edges $k = (u_1, \_, v_1)$ could be selected such that:

- v pre-dominates u_1;
- u_1 pre-dominates v_1;
- v_1 predominates u.

On the level of source programs, this is trivial:

$$\begin{array}{l} \mathsf{do}\ \{\ s_1 \ldots s_k \\ \quad \}\ \mathsf{while}\ (e); \end{array}$$

The desired assignments must be among the s_i :-)

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:

$$x = \emptyset;$$
$$\text{for } (t = F\,x; t \not\subseteq x; \boxed{t = F\,x;})$$
$$x = x \cup t;$$

If F is distributive, it could be replaced by:

$$x = \emptyset;$$
$$\text{for } (\Delta = F\,x; \Delta \neq \emptyset; \boxed{\Delta = (F\,\Delta) \setminus x;})$$
$$x = x \cup \Delta;$$

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	$1 + 2 + \ldots + n + n \quad = \quad \frac{1}{2}n(n+3)$
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.,
  - → Eliminate redundancies!
  - → 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 :-)

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

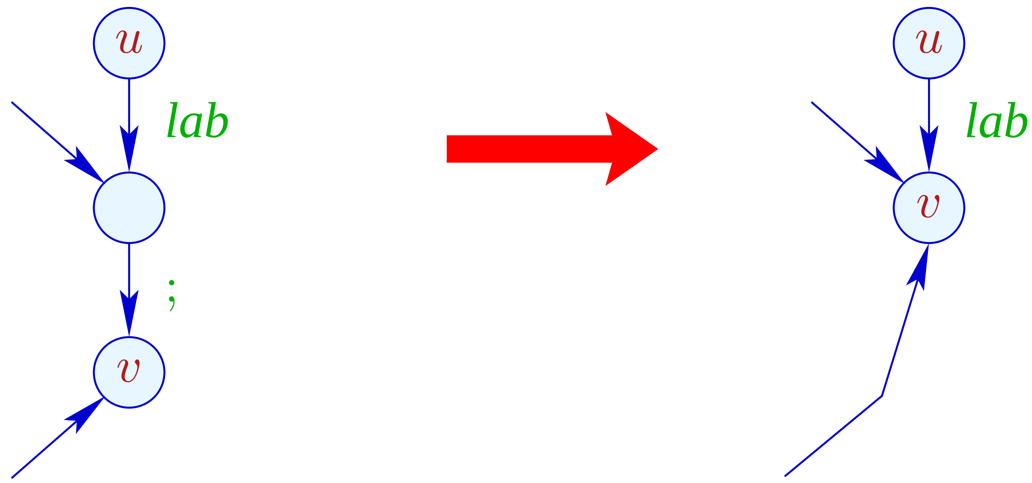
// algebraic simplifications :-)

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

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

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

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 $\textsf{next} : \mathit{Nodes} \to \mathit{Nodes}$ with:

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

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

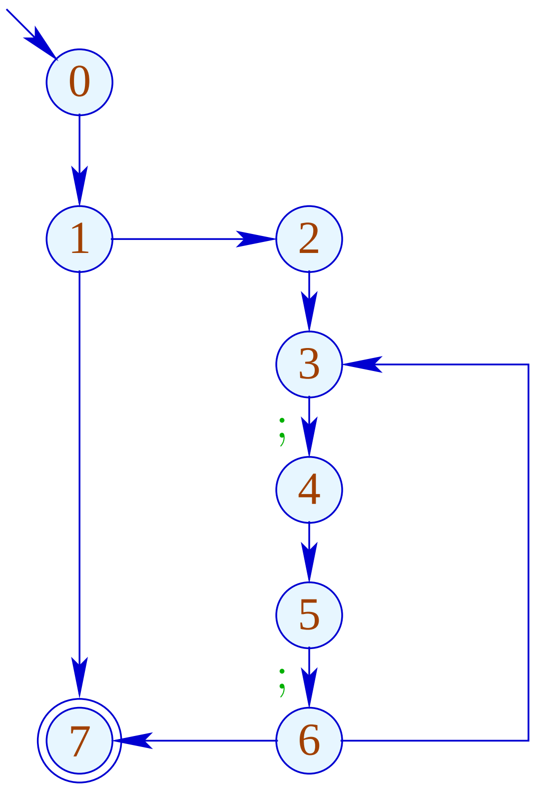
- We replace every edge:

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

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

- All ;-edges are removed ;-)

Example:

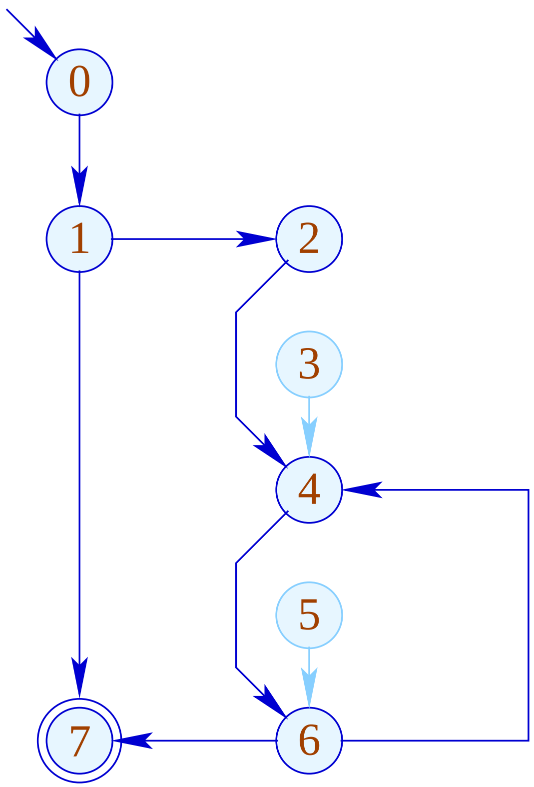


$$\text{next } 1 = 1$$

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

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

Example:



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

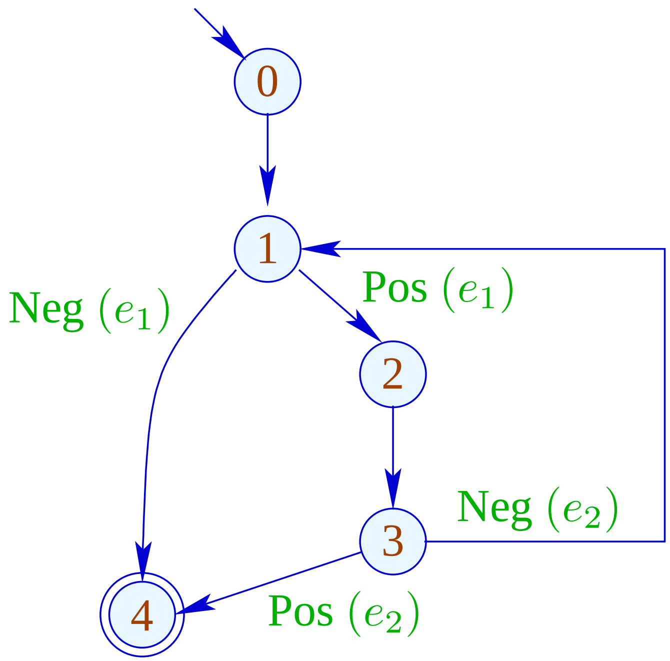
2. Subproblem: Linearization

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

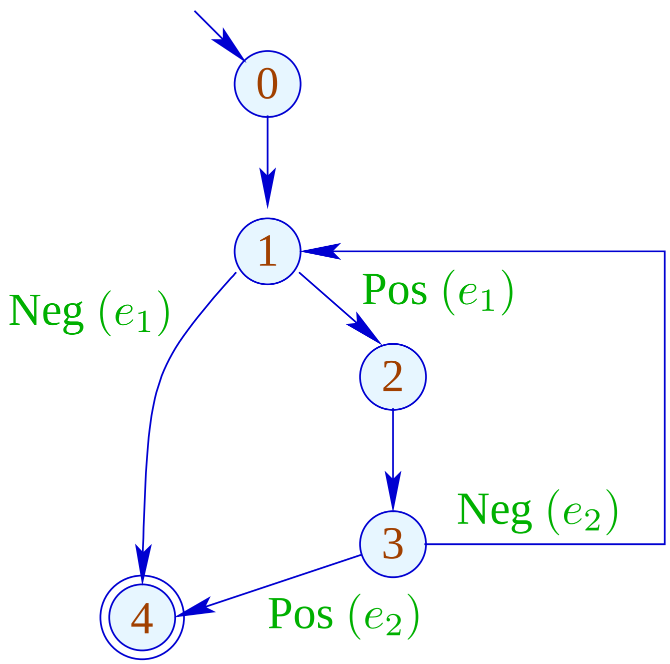
2: $\boxed{\text{Rumpf}}$

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

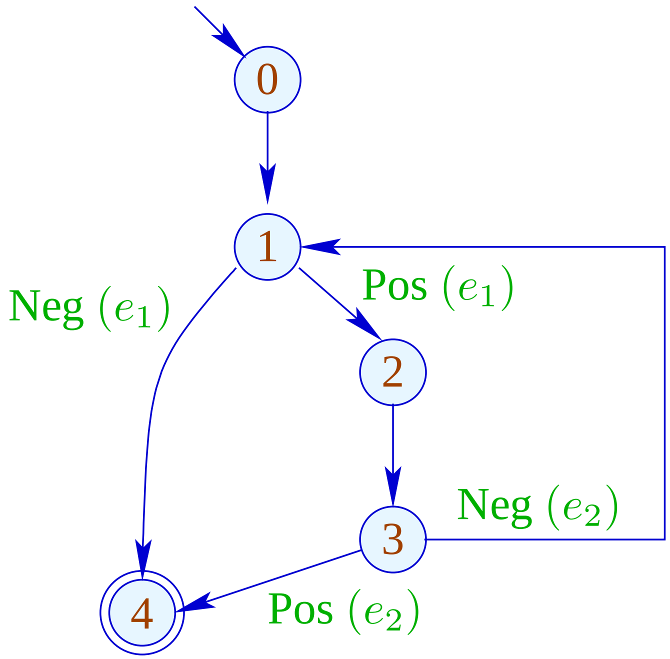
  (1) nodes which have already been handled;

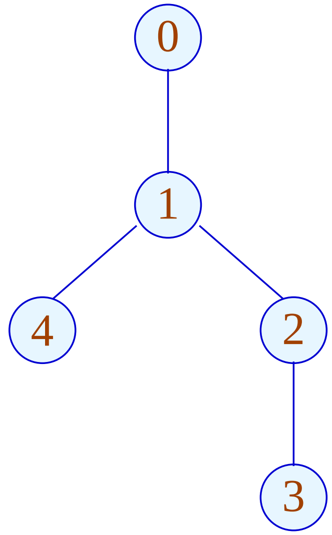
  (2) colder nodes.

- Temperature $\approx$ nesting-depth

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

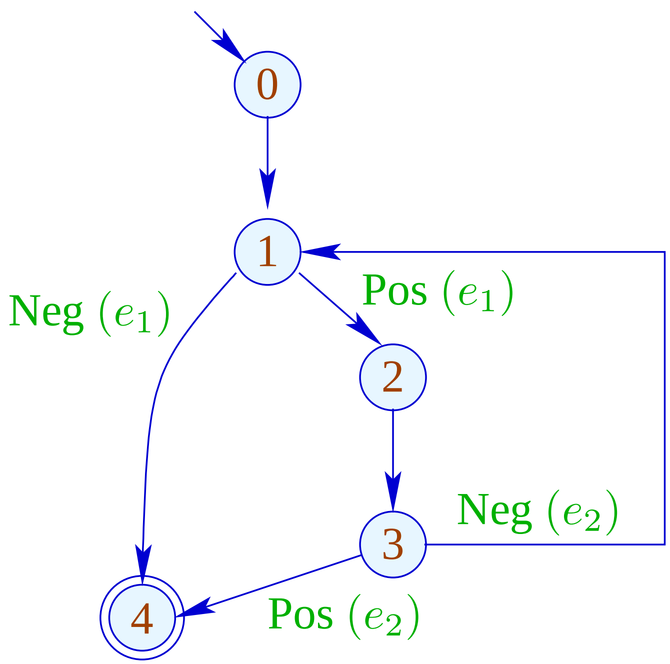
... in the Example:

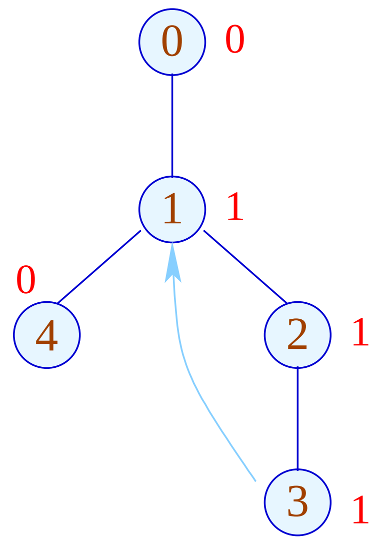




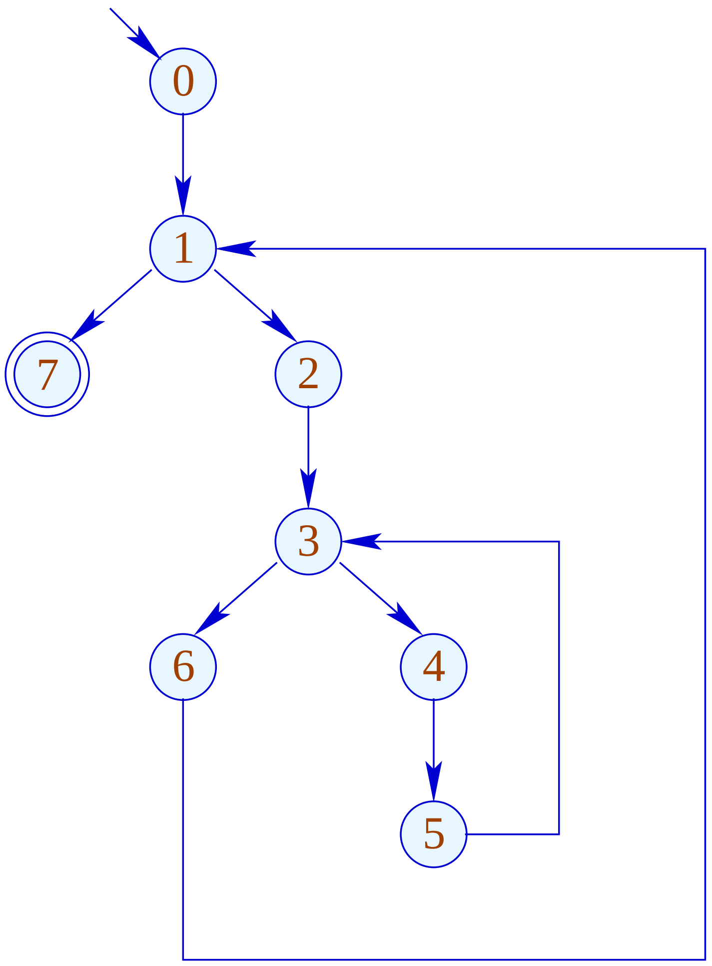
The sub-tree with back edge is hotter ...

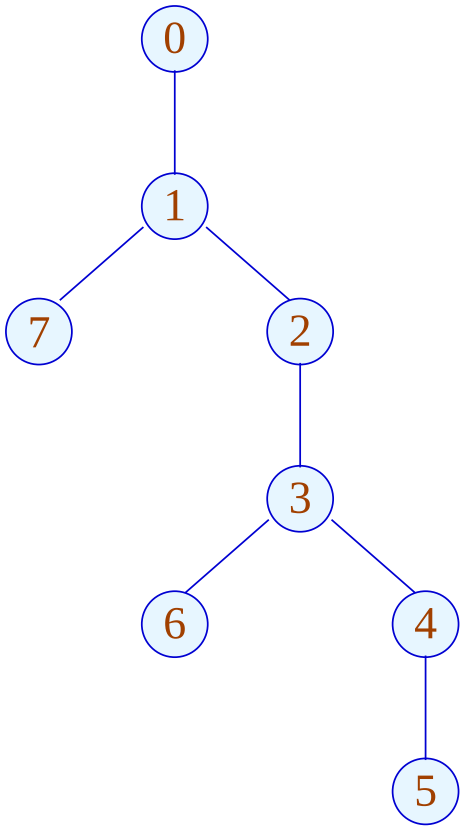
... in the Example:



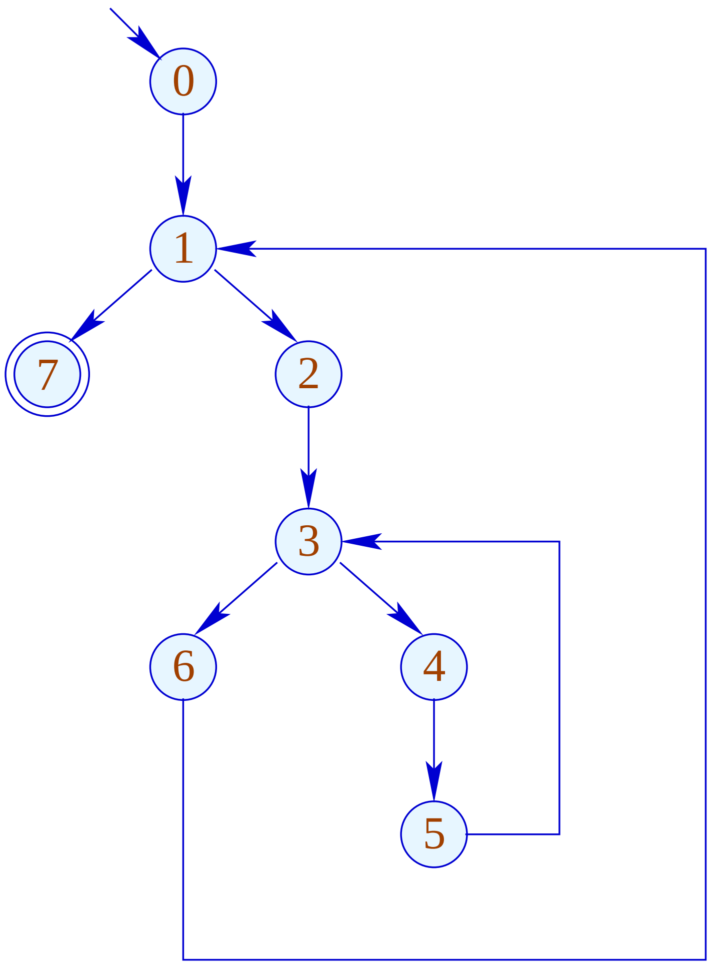


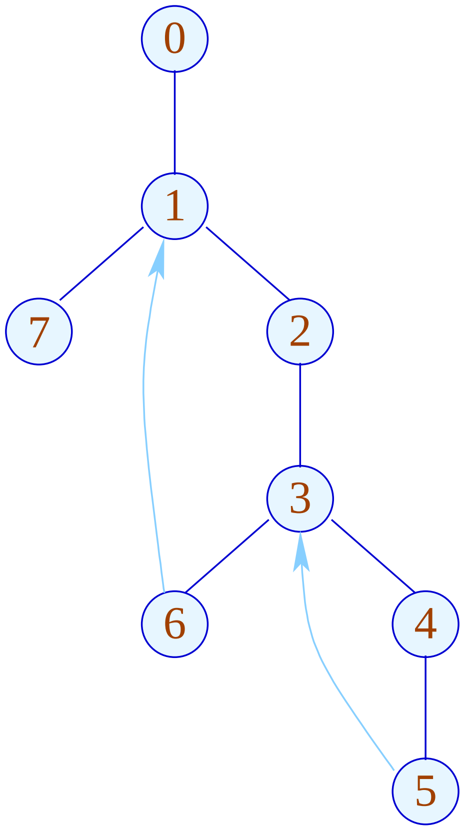
More Complicated Example:



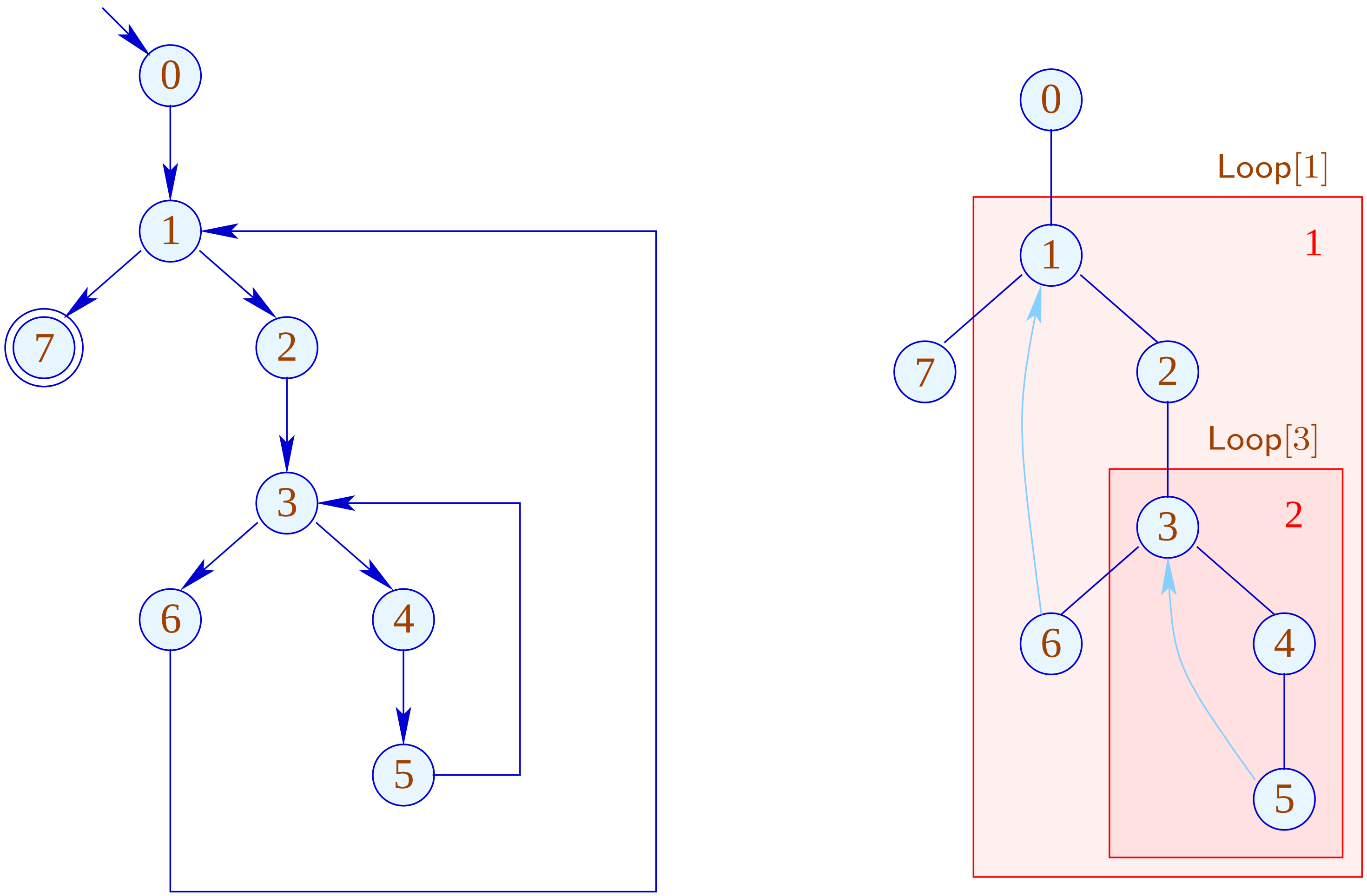


More Complicated Example:



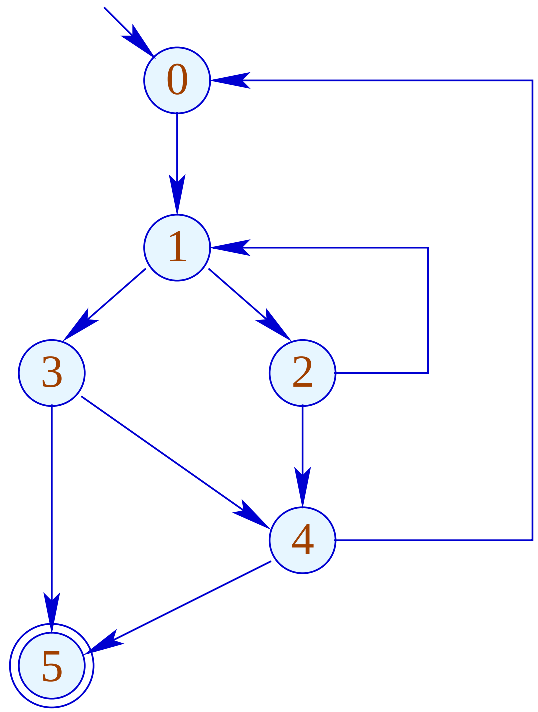


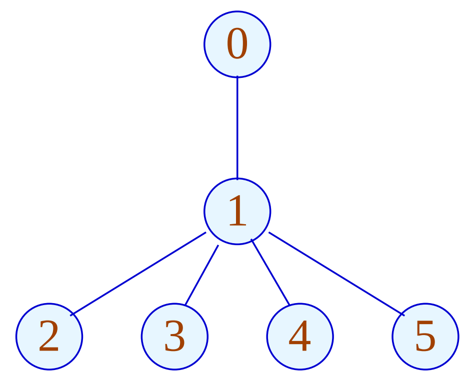
More Complicated Example:



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

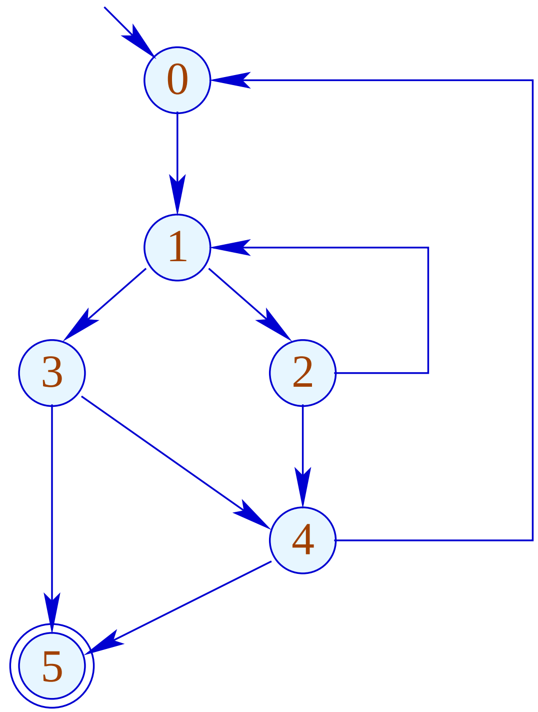
Is is also meaningful for do-while-loops with breaks ...

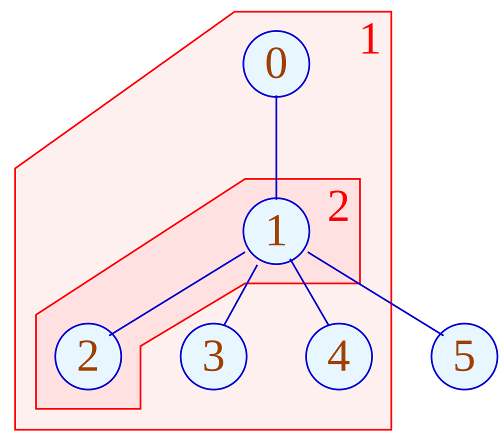




Our definition of Loop implies that (detected) loops are necessarily nested :-)

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;

- $\rightarrow$ If an edge leads to a node we already have generated code for, then we insert a jump.
- $\rightarrow$ If a node has two successors with different temperature, then we insert a jump to the colder of the two.
- $\rightarrow$ If both successors are equally warm, then it does not matter ;-)