1.8 Application: Loop-invariant Code

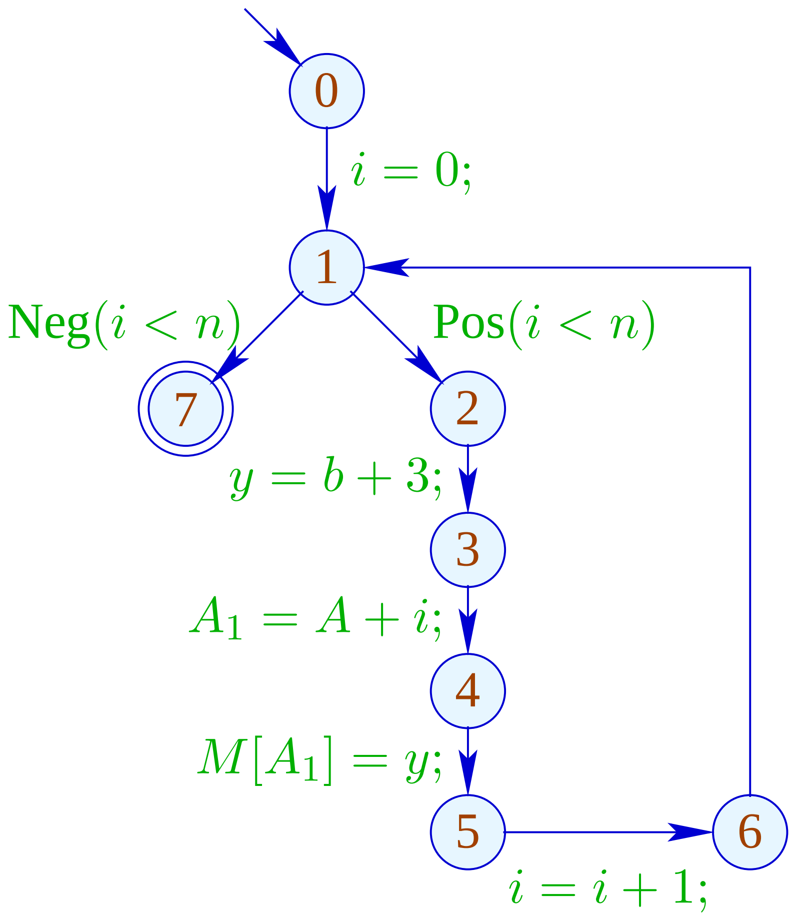
Example:

$$\text{for } (i = 0; i < n; i{+}{+})$$
$$a[i] = b + 3;$$

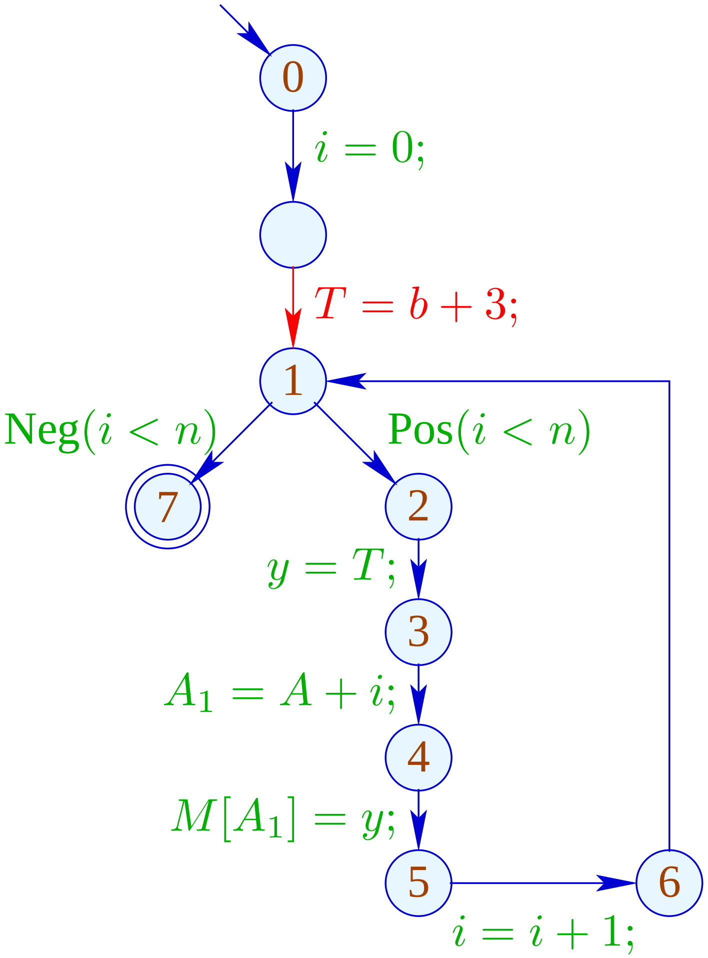
// The expression $b + 3$ is recomputed in every iteration :-(

// This should be avoided :-)

The Control-flow Graph:

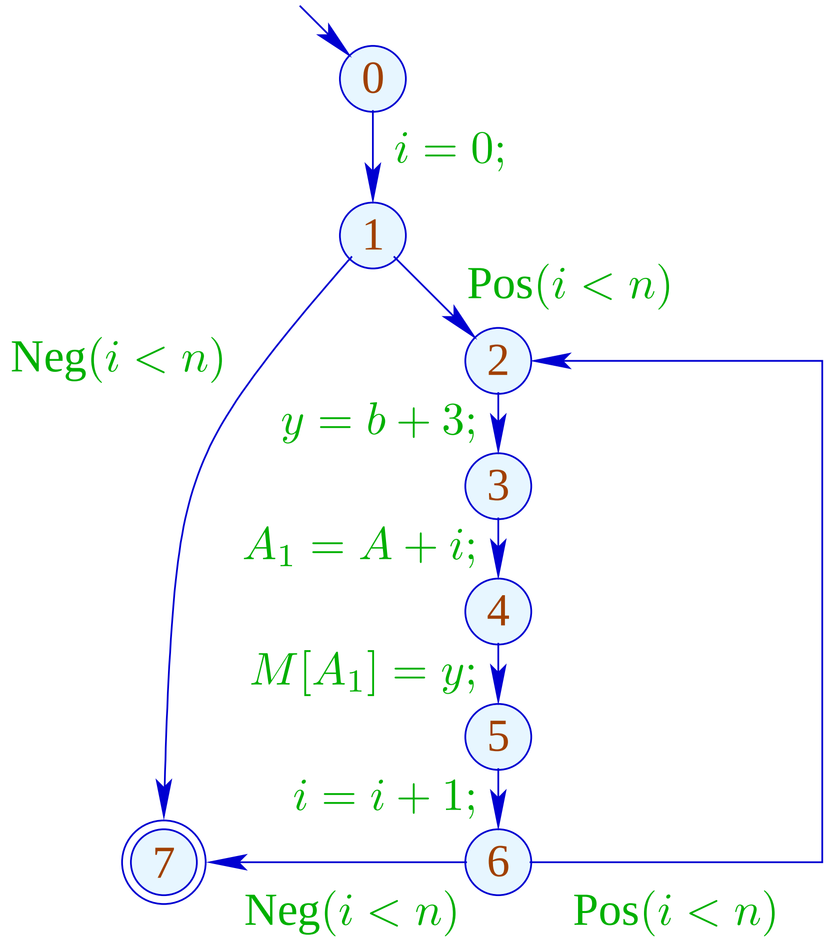


Warning: $T = b + 3;$ may not be placed before the loop :

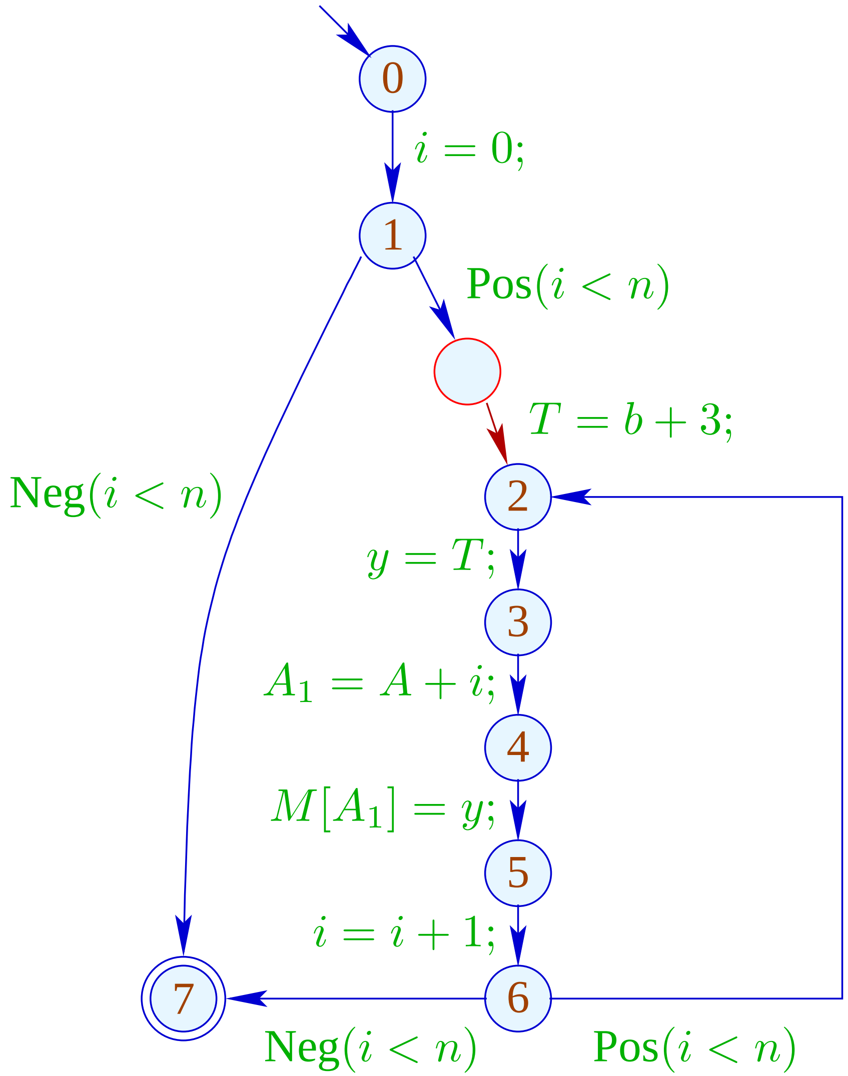


$\Longrightarrow$ There is no decent place for $T = b + 3;$:-(

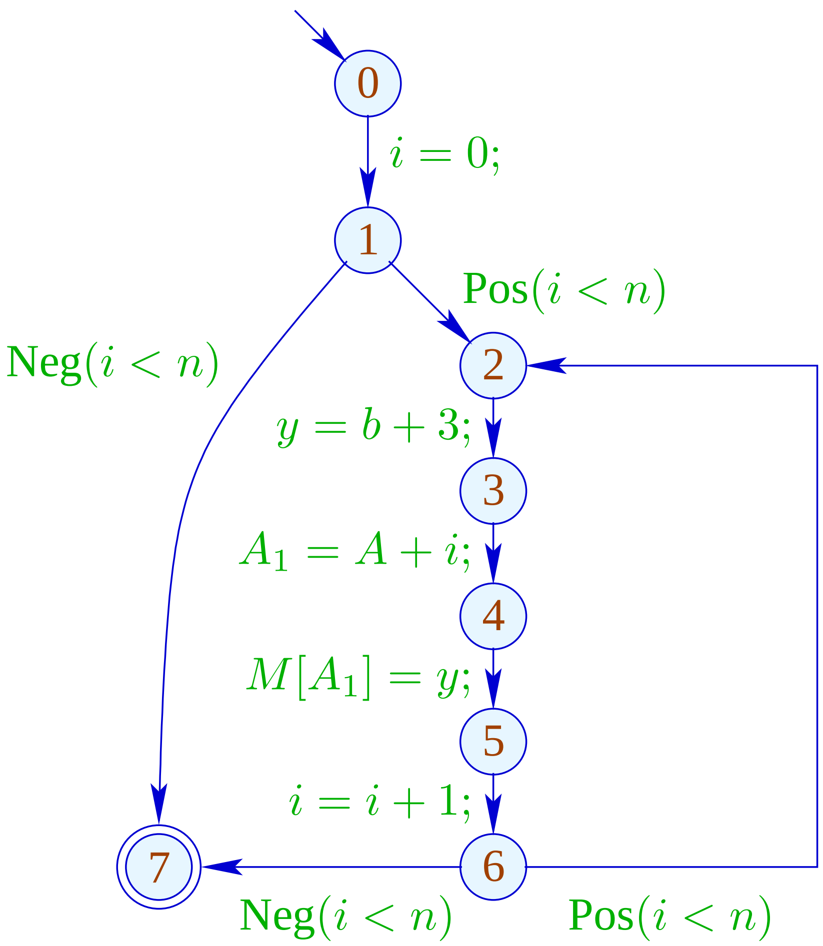
Idea: Transform into a do-while-loop ...



... now there is a place for $T = e;$:-)

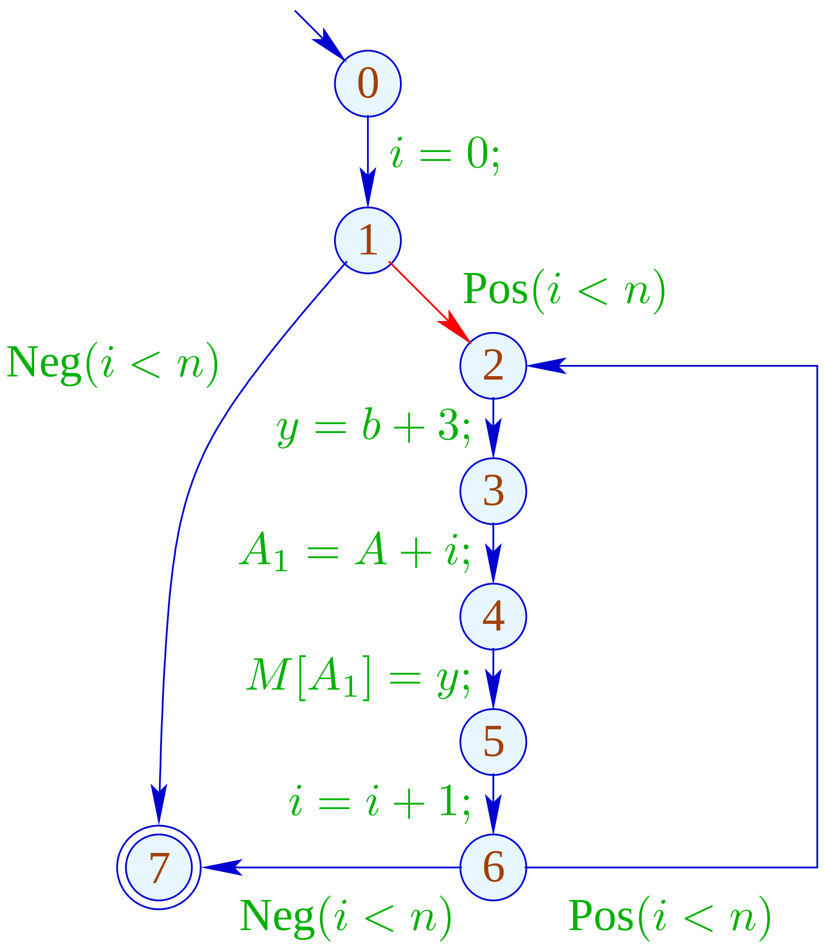


Application of T5 (PRE) :



	$\mathcal{A}$	$\mathcal{B}$
0	$\emptyset$	$\emptyset$
1	$\emptyset$	$\emptyset$
2	$\emptyset$	$\{b+3\}$
3	$\{b+3\}$	$\emptyset$
4	$\{b+3\}$	$\emptyset$
5	$\{b+3\}$	$\emptyset$
6	$\{b+3\}$	$\emptyset$
6	$\emptyset$	$\emptyset$
7	$\emptyset$	$\emptyset$

Application of T5 (PRE) :



	$\mathcal{A}$	$\mathcal{B}$
0	$\emptyset$	$\emptyset$
1	$\emptyset$	$\emptyset$
2	$\emptyset$	$\{b + 3\}$
3	$\{b + 3\}$	$\emptyset$
4	$\{b + 3\}$	$\emptyset$
5	$\{b + 3\}$	$\emptyset$
6	$\{b + 3\}$	$\emptyset$
6	$\emptyset$	$\emptyset$
7	$\emptyset$	$\emptyset$

Conclusion:

- Elimination of partial redundancies may move loop-invariant code out of the loop :-))
- This only works properly for do-while-loops :-(
- To optimize other loops, we transform them into do-while-loops before-hand:

$$\textsf{while}\ (b)\ stmt \quad \Longrightarrow \quad \begin{array}{l} \textsf{if}\ (b) \\ \qquad \textsf{do}\ stmt \\ \qquad \textsf{while}\ (b); \end{array}$$

$\Longrightarrow$ Loop Rotation

Problem:

If we do not have the source program at hand, we must re-construct potential loop headers ;-)

$\Longrightarrow$ Pre-dominators

u pre-dominates v , if every path $\pi : start \to^* v$ contains u. We write: $u \Rightarrow v$.

"$\Rightarrow$" is reflexive, transitive and anti-symmetric :-)

Computation:

We collect the nodes along paths by means of the analysis:

$$\mathbb{P} = 2^{Nodes} \quad , \qquad \sqsubseteq \; = \; \supseteq$$

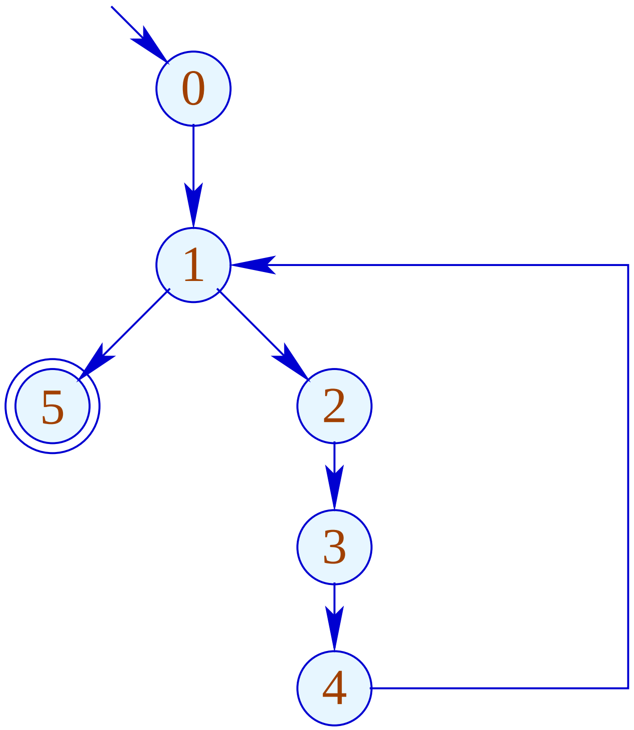
$$[\![(\_, \_, v)]\!]^\sharp \, P \;\; = \;\; P \cup \{v\}$$

Then the set $\mathcal{P}[v]$ of pre-dominators is given by:

$$\mathcal{P}[v] = \bigcap \{[\![\pi]\!]^\sharp \, \{start\} \mid \pi : start \to^* v\}$$

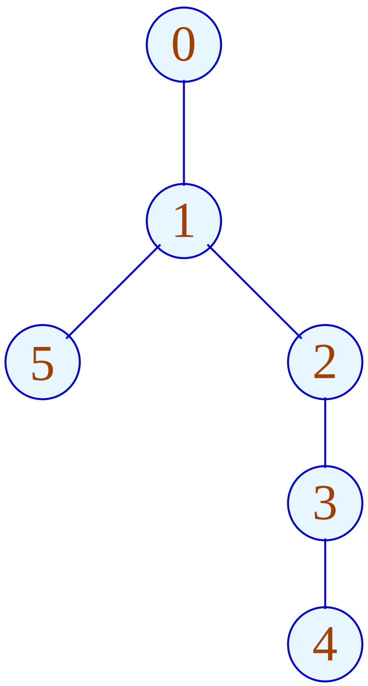
Since $[\![k]\!]^\sharp$ are distributive, the $\mathcal{P}[v]$ can computed by means of fixpoint iteration :-)

Example:



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

The partial ordering "⇒" in the example:



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

Apparently, the result is a tree :-)

In fact, we have:

Theorem:

Every node v has at most one immediate pre-dominator.

Proof:

Assume:

there are $u_1 \neq u_2$ which immediately pre-dominate v.

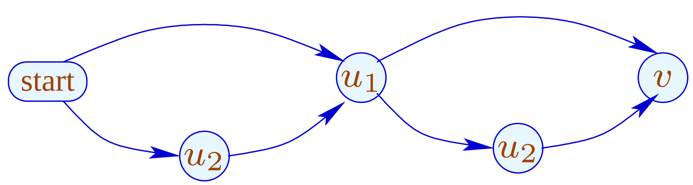
If $u_1 \Rightarrow u_2$ then u_1 not immediate.

Consequently, u_1, u_2 are incomparable :-)

Now for every $\pi : start \rightarrow^* v$:

$$\pi = \pi_1 \; \pi_2 \qquad \text{with} \qquad \begin{array}{l} \pi_1 : start \rightarrow^* u_1 \\ \pi_2 : u_1 \rightarrow^* v \end{array}$$

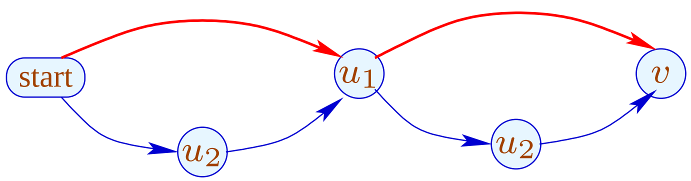
If, however, u_1, u_2 are incomparable, then there is path: $start \rightarrow^* v$ avoiding u_2 :



Now for every $\pi : start \rightarrow^* v$:

$$\pi = \pi_1 \, \pi_2 \qquad \text{with} \qquad \begin{array}{l} \pi_1 : start \rightarrow^* u_1 \\ \pi_2 : u_1 \rightarrow^* v \end{array}$$

If, however, u_1, u_2 are incomparable, then there is path: $start \rightarrow^* v$ avoiding u_2 :



Observation:

The loop head of a while-loop pre-dominates every node in the body.

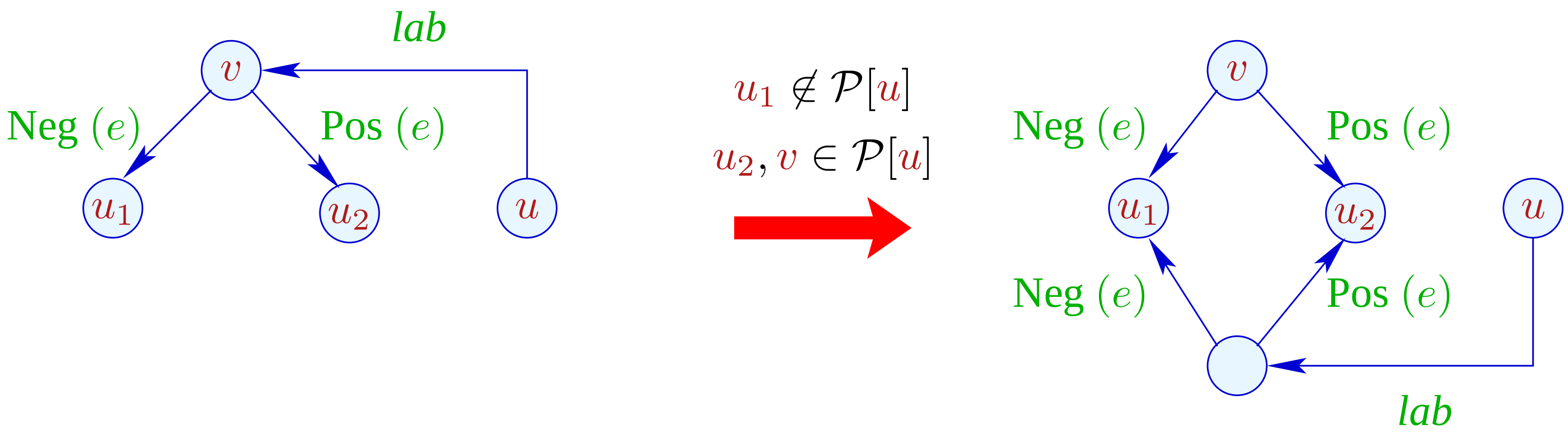
A back edge from the exit u to the loop head v can be identified through

$$v \in \mathcal{P}[u]$$

:-)

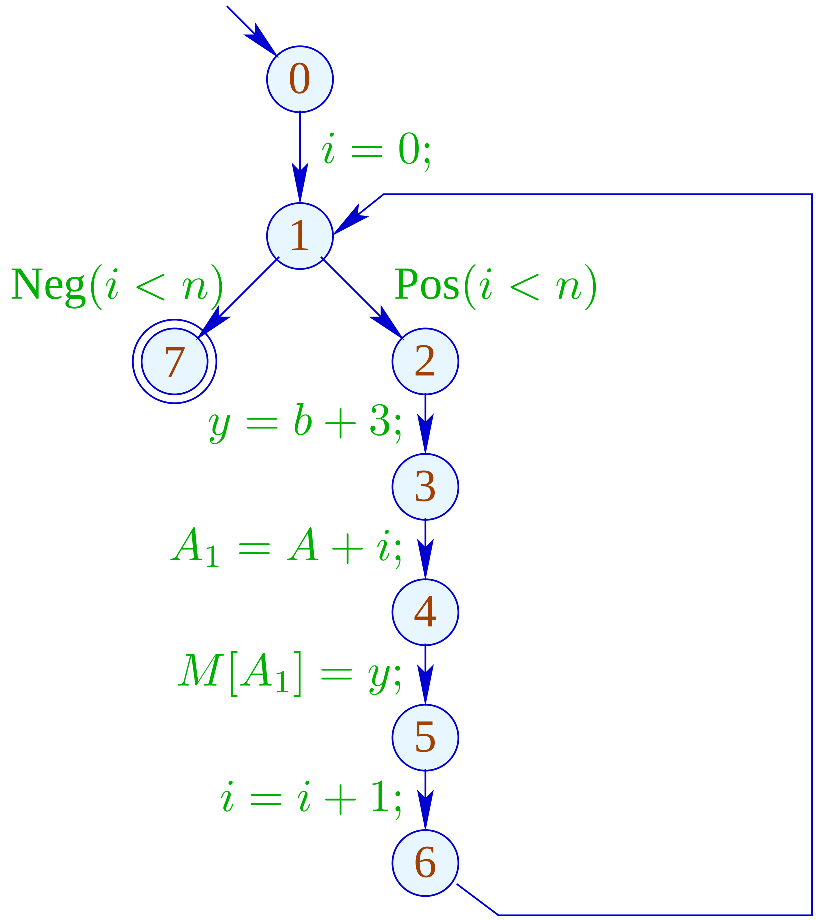
Accordingly, we define:

Transformation 6:

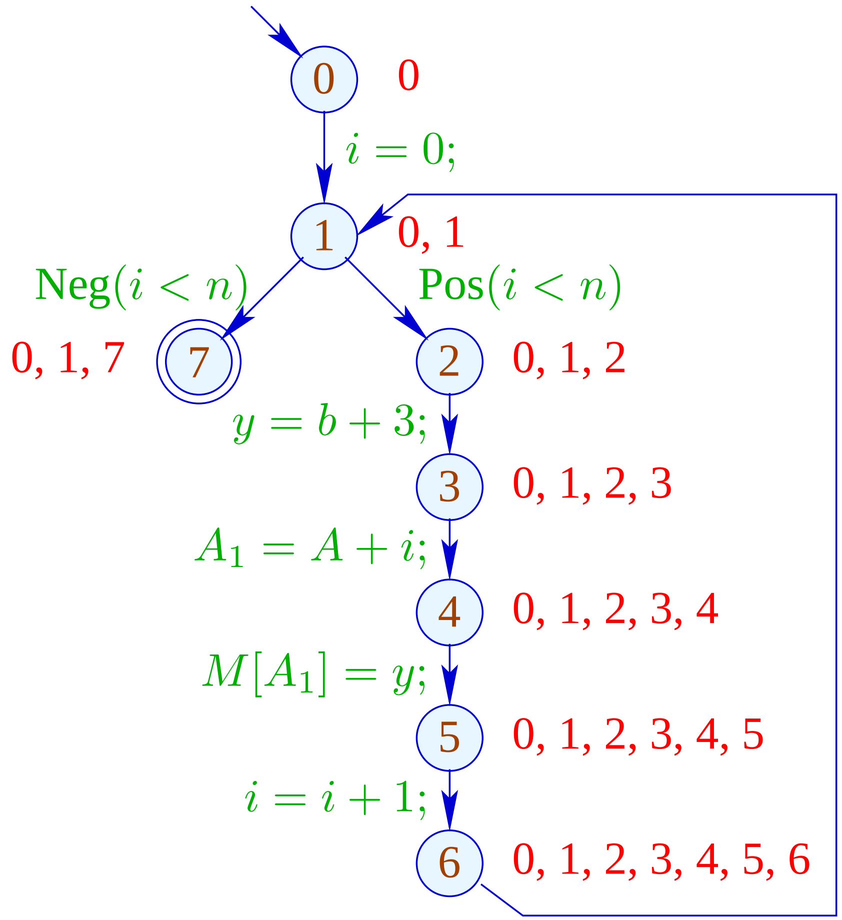


We duplicate the entry check to all back edges :-)

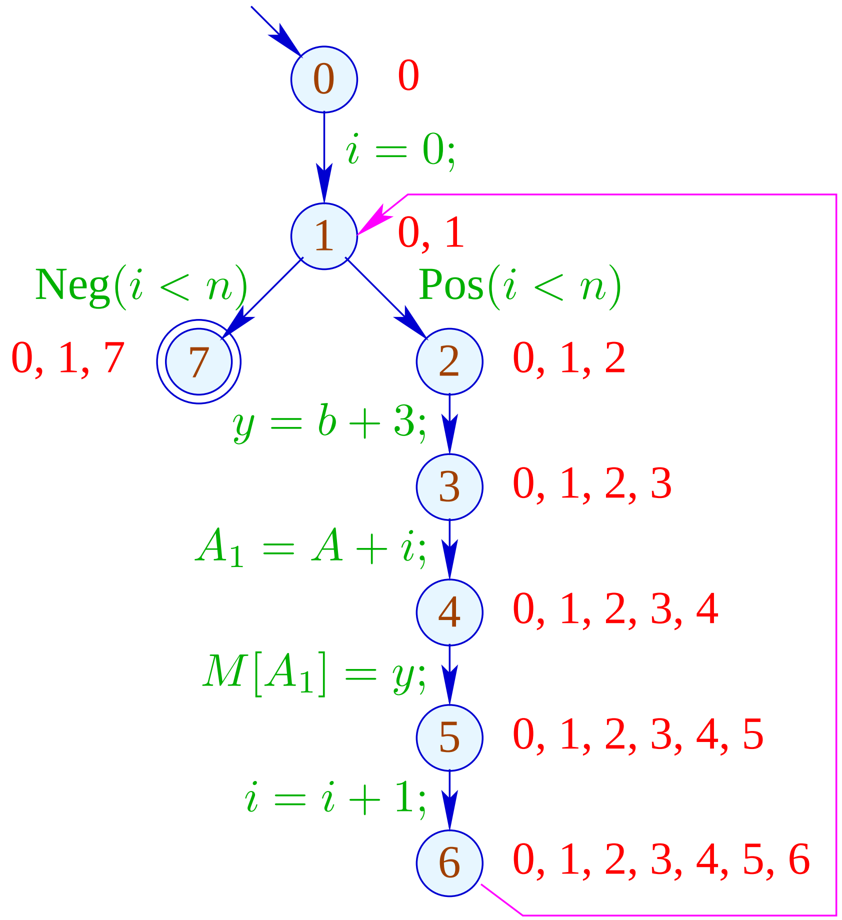
... in the Example:



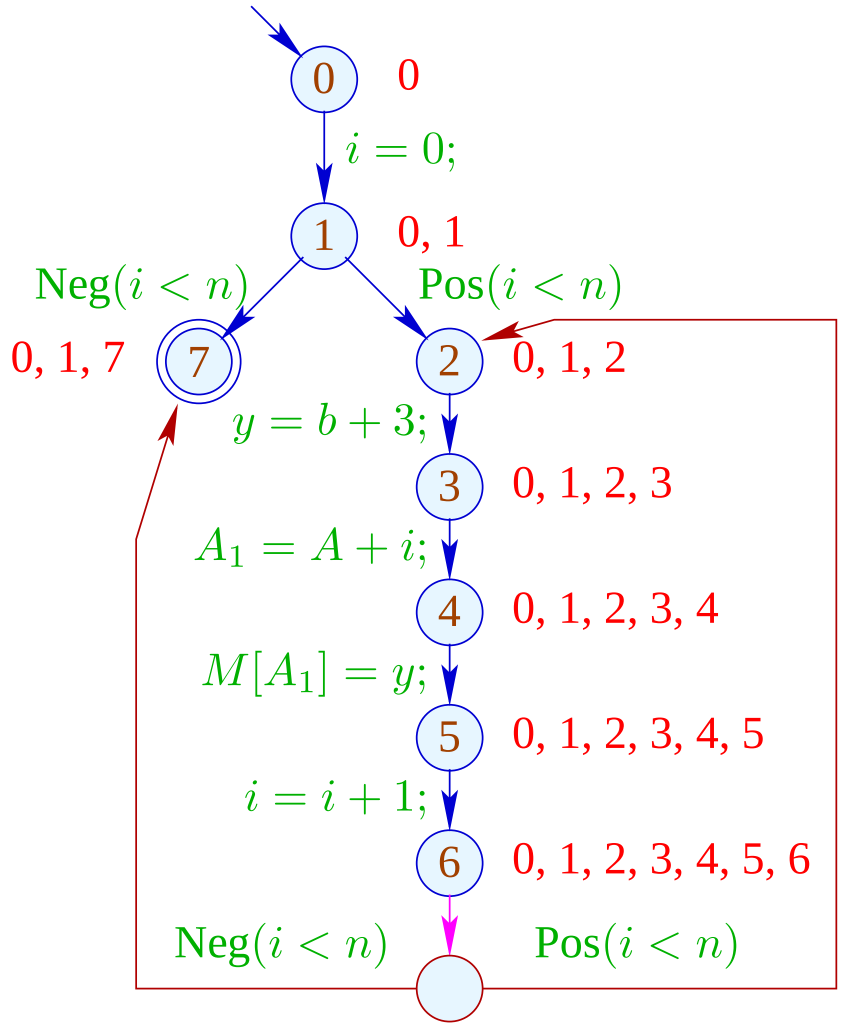
... in the Example:



... in the Example:

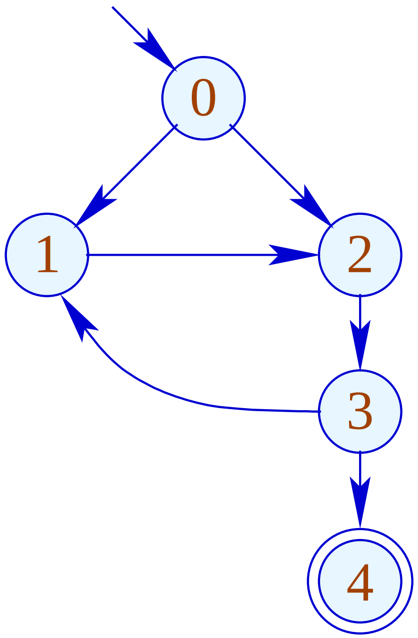


... in the Example:

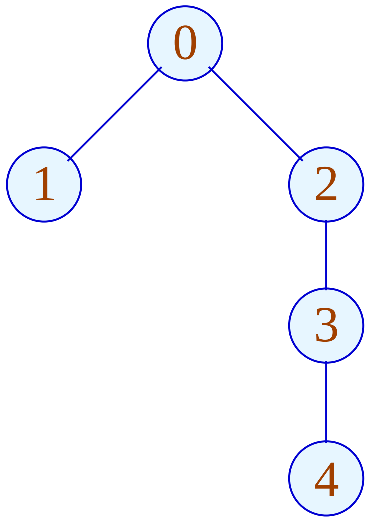


Warning:

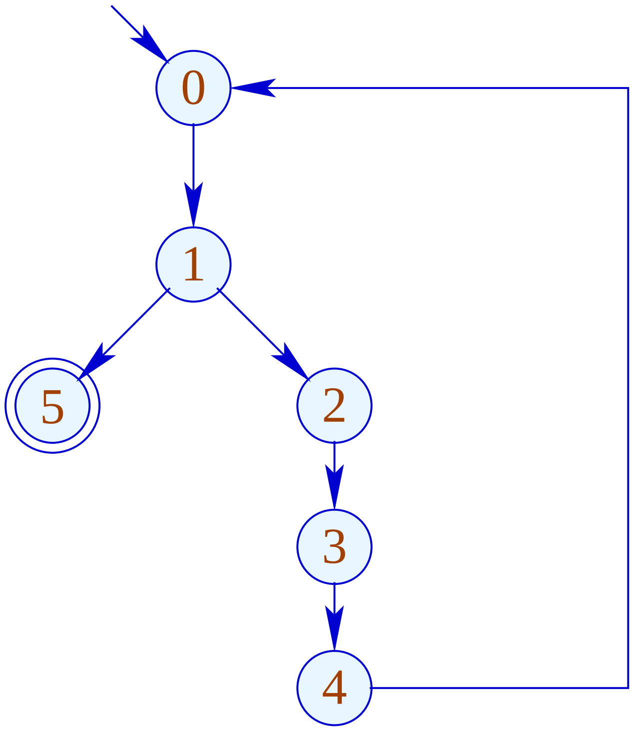
There are unusual loops which cannot be rotated:

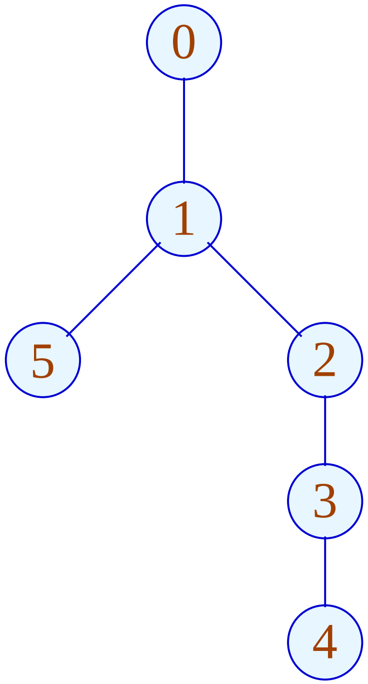


Pre-dominators:



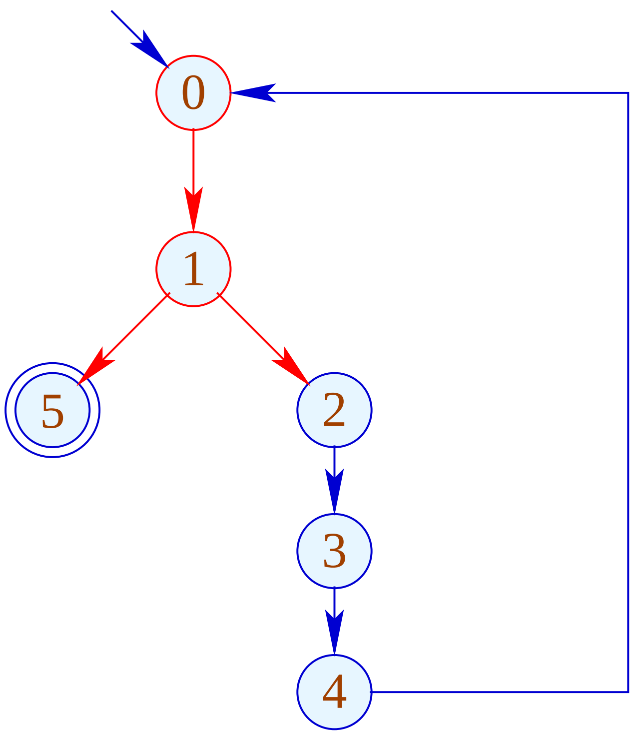
... but also common ones which cannot be rotated:

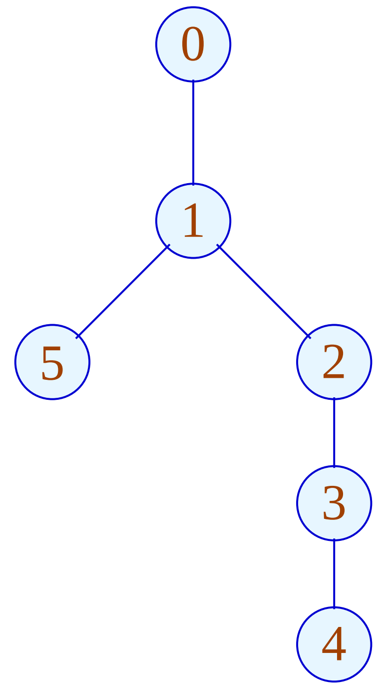




Here, the complete block between back edge and conditional jump should be duplicated :-(

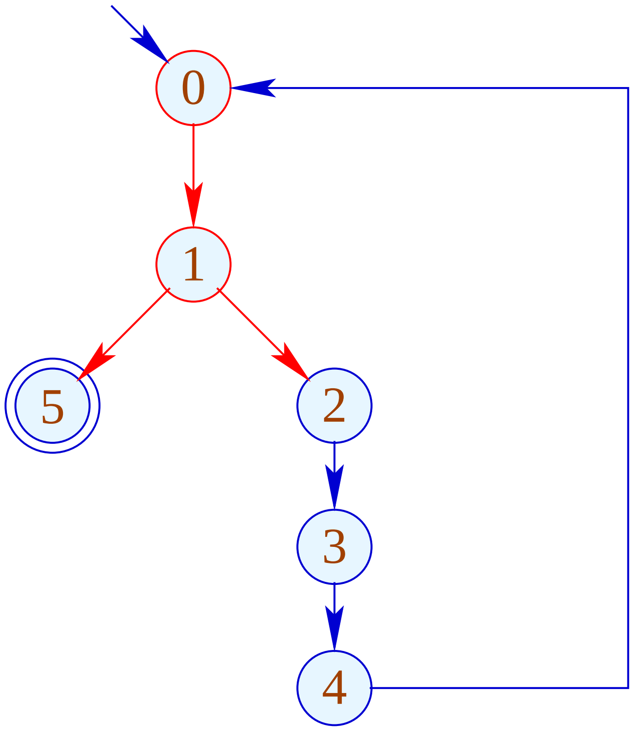
... but also common ones which cannot be rotated:





Here, the complete block between back edge and conditional jump should be duplicated :-(

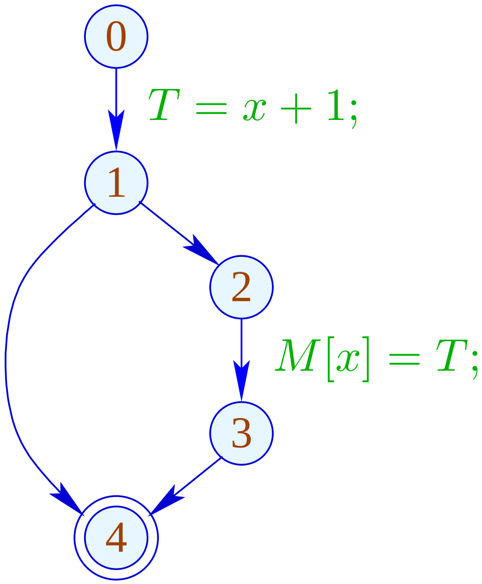
... but also common ones which cannot be rotated:



Here, the complete block between back edge and conditional jump should be duplicated :-(

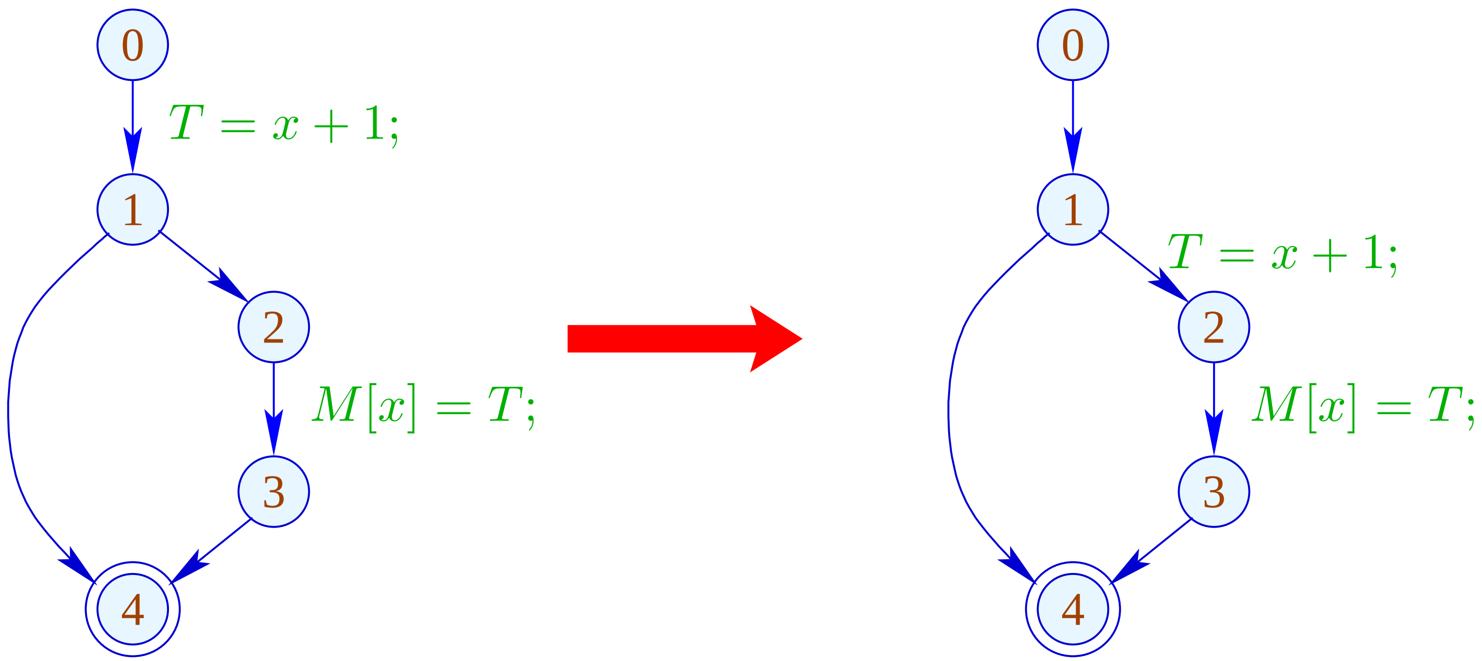
1.9 Eliminating Partially Dead Code

Example:



$x + 1$ need only be computed along one path ;-(

Idea:



Problem:

- The definition $x = e;$ $(x \notin Vars_e)$ may only be moved to an edge where e is safe ;-)
- The definition must still be available for uses of x ;-)

$\Longrightarrow$

We define an analysis which maximally delays computations:

$$
\begin{array}{lll}
[\![;]\!]^\sharp \, D & = & D \\
[\![x = e;]\!]^\sharp \, D & = & \begin{cases} D \backslash (Use_e \cup Def_x) \cup \{x = e;\} & \text{if} \quad x \notin Vars_e \\ D \backslash (Use_e \cup Def_x) & \text{if} \quad x \in Vars_e \end{cases}
\end{array}
$$

... where:

$$
\begin{aligned}
Use_e &= \{y = e'; \mid y \in Vars_e\} \\
Def_x &= \{y = e'; \mid y \equiv x \vee x \in Vars_{e'}\}
\end{aligned}
$$

... where:

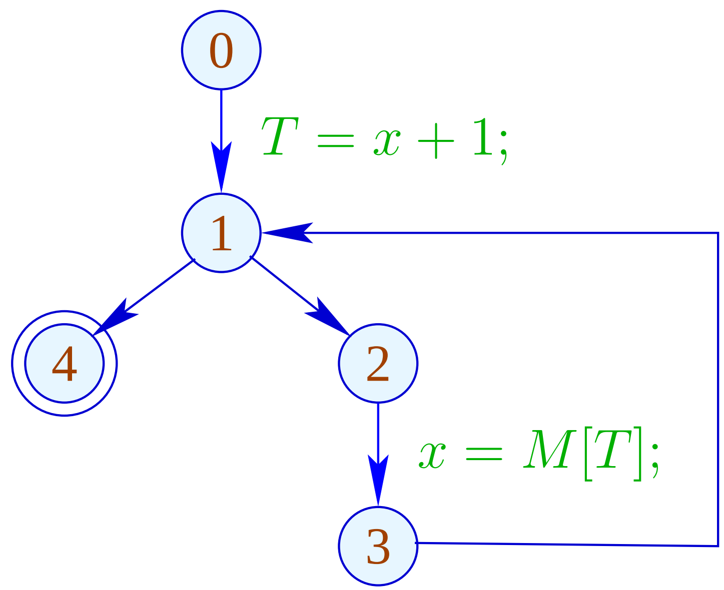
$$\begin{array}{lll} \mathit{Use}_e & = & \{y = e'; \mid y \in \mathit{Vars}_e\} \\ \mathit{Def}_x & = & \{y = e'; \mid y \equiv x \vee x \in \mathit{Vars}_{e'}\} \end{array}$$

For the remaining edges, we define:

$$\begin{array}{lll} [\![x = M[e];]\!]^\sharp\, D & = & D \backslash (\mathit{Use}_e \cup \mathit{Def}_x) \\ [\![M[e_1] = e_2;]\!]^\sharp\, D & = & D \backslash (\mathit{Use}_{e_1} \cup \mathit{Use}_{e_2}) \\ [\![\mathsf{Pos}(e)]\!]^\sharp\, D & = & [\![\mathsf{Neg}(e)]\!]^\sharp\, D \quad = \quad D \backslash \mathit{Use}_e \end{array}$$

Warning:

We may move $y = e;$ beyond a join only if $y = e;$ can be delayed along all joining edges:



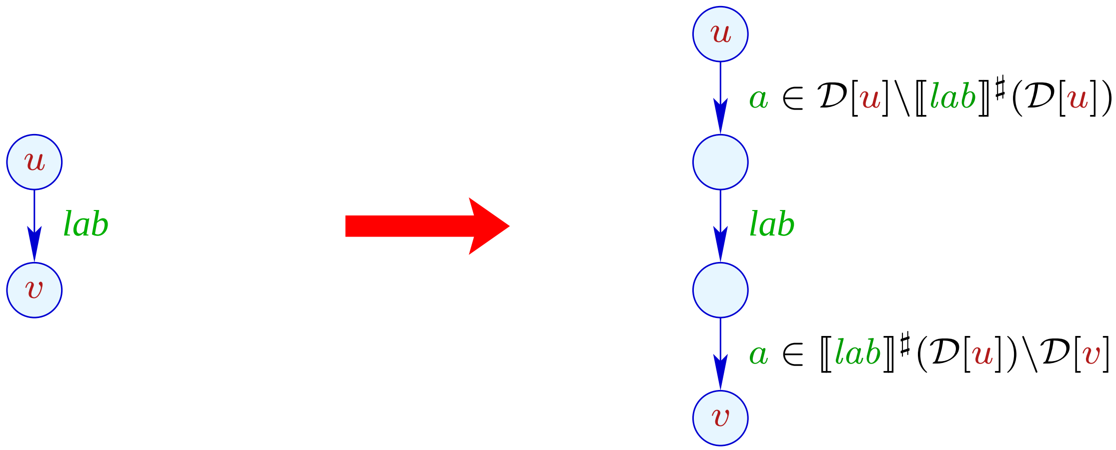
Here, $T = x + 1;$ cannot be moved beyond 1 !!!

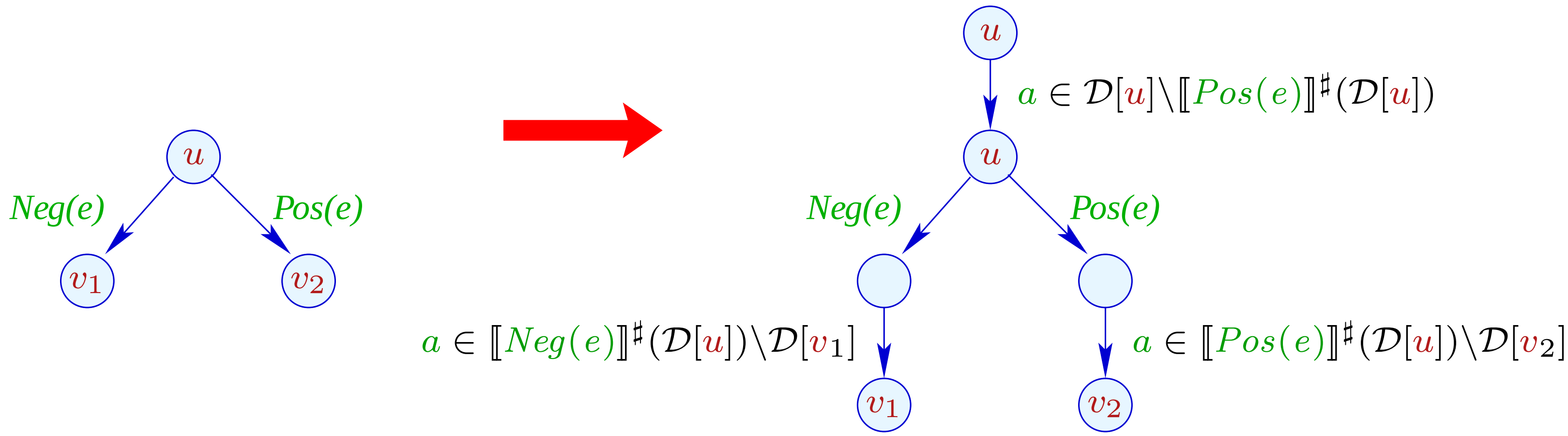
We conclude:

- The partial ordering of the lattice for delayability is given by "$\supseteq$".
- At program start: $D_0 = \emptyset$.

  Therefore, the sets $\mathcal{D}[u]$ of at u delayable assignments can be computed by solving a system of constraints.
- We delay only assignments a where $a\, a$ has the same effect as a alone.
- The extra insertions render the original assignments as assignments to dead variables ...

Transformation 7:

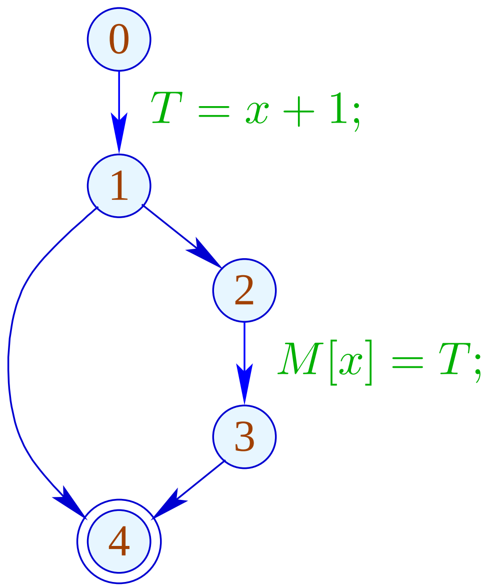




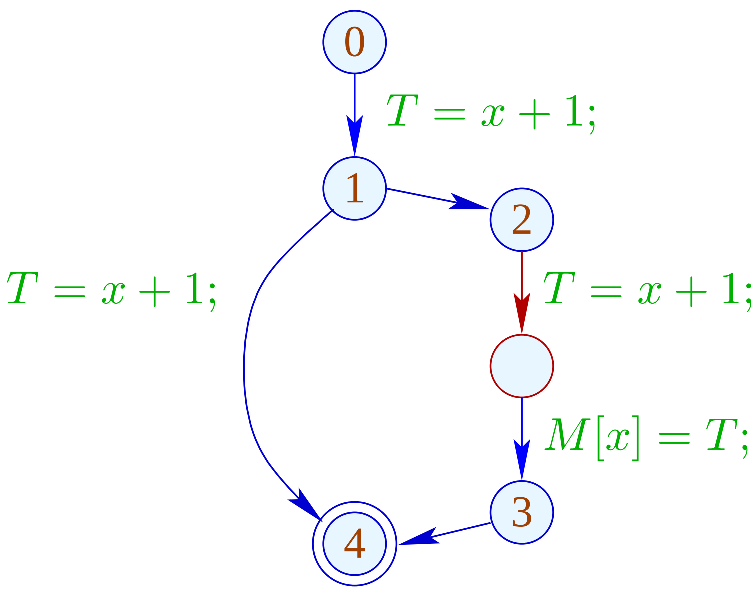
Note:

Transformation T7 is only meaningful, if we subsequently eliminate assignments to dead variables by means of transformation T2 :-)

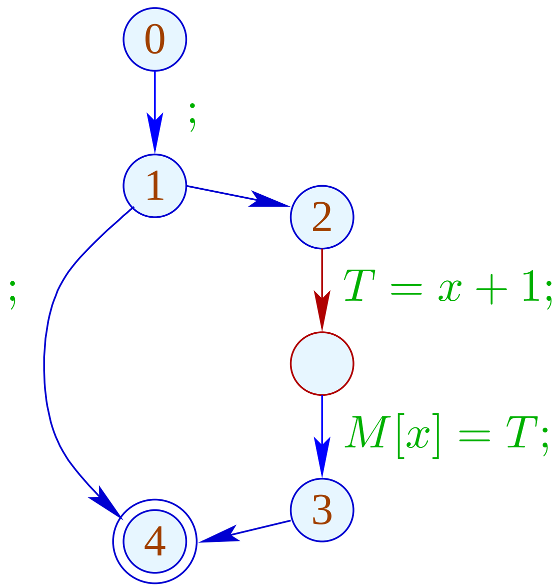
In the example, the partially dead code is eliminated:



	$\mathcal{D}$
0	$\emptyset$
1	$\{T = x + 1;\}$
2	$\{T = x + 1;\}$
3	$\emptyset$
4	$\emptyset$



	$\mathcal{D}$
0	$\emptyset$
1	$\{T = x + 1;\}$
2	$\{T = x + 1;\}$
3	$\emptyset$
4	$\emptyset$



	$\mathcal{L}$
0	$\{x\}$
1	$\{x\}$
2	$\{x\}$
2′	$\{x, T\}$
3	$\emptyset$
4	$\emptyset$

Remarks:

- After $T7$, all original assignments $y = e;$ with $y \not\in Vars_e$ are assignments to dead variables and thus can always be eliminated :-)
- By this, it can be proven that the transformation is guaranteed to be non-degradating efficiency of the code :-))
- Similar to the elimination of partial redundancies, the transformation can be repeated :-}

Conclusion:

- → The design of a meaningful optimization is non-trivial.
- → Many transformations are advantageous only in connection with other optimizations :-)
- → The ordering of applied optimizations matters !!
- → Some optimizations can be iterated !!!

... a meaningful ordering:

T4	Constant Propagation Interval Analysis Alias Analysis
T6	Loop Rotation
T1, T3, T2	Available Expressions
T2	Dead Variables
T7, T2	Partially Dead Code
T5, T3, T2	Partially Redundant Code

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 :

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

$\rightarrow$ 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	18
1	15	10	26	
2	25	36		
3	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: $y_0 = f(x_0), \quad \Delta = a_1 \cdot h$

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

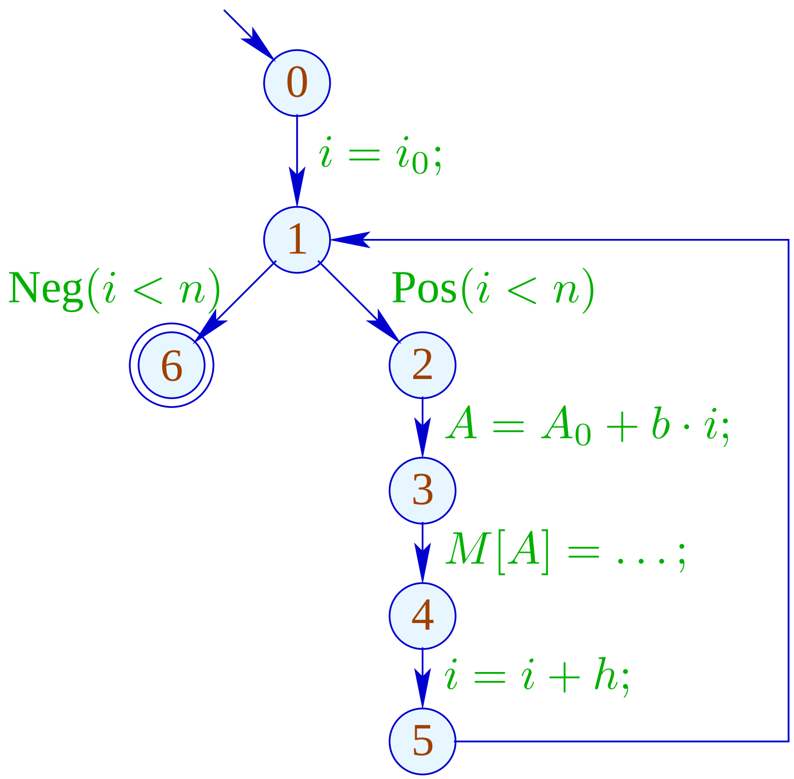
Example:

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

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

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

$\}$



... or, after loop rotation:

$i = i_0;$

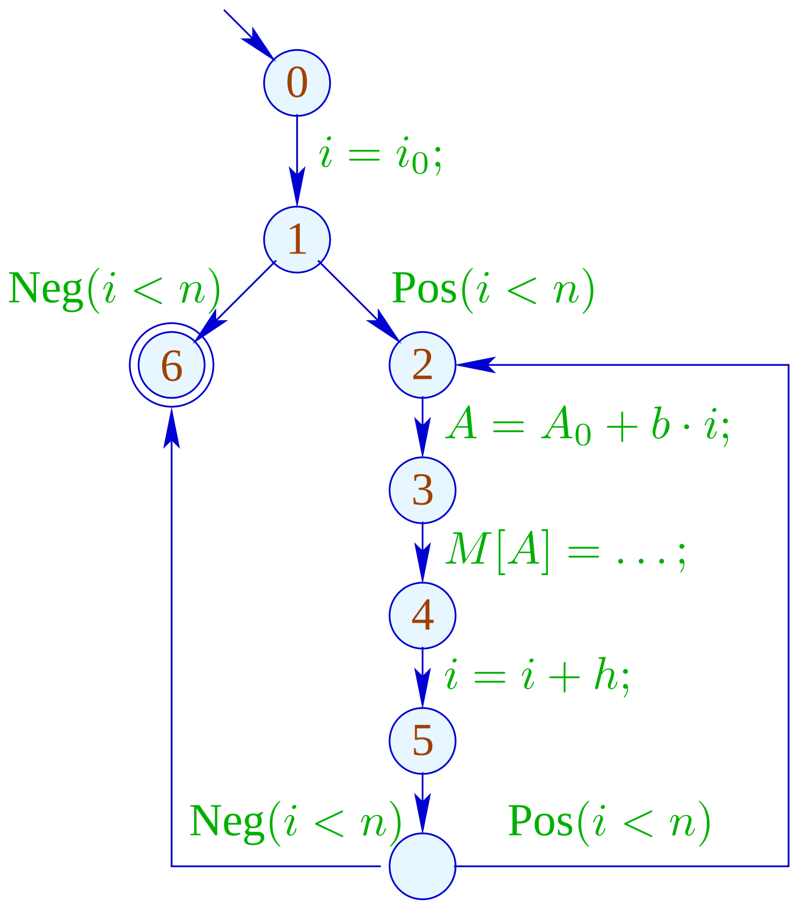
if $(i < n)$ do $\{$

$A = A_0 + b \cdot i;$

$M[A] = \ldots;$

$i = i + h;$

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



... and reduction of strength:

$i = i_0;$

if $(i < n)$ {

    $\Delta = b \cdot h;$

    $A = A_0 + b \cdot i_0;$

    do {

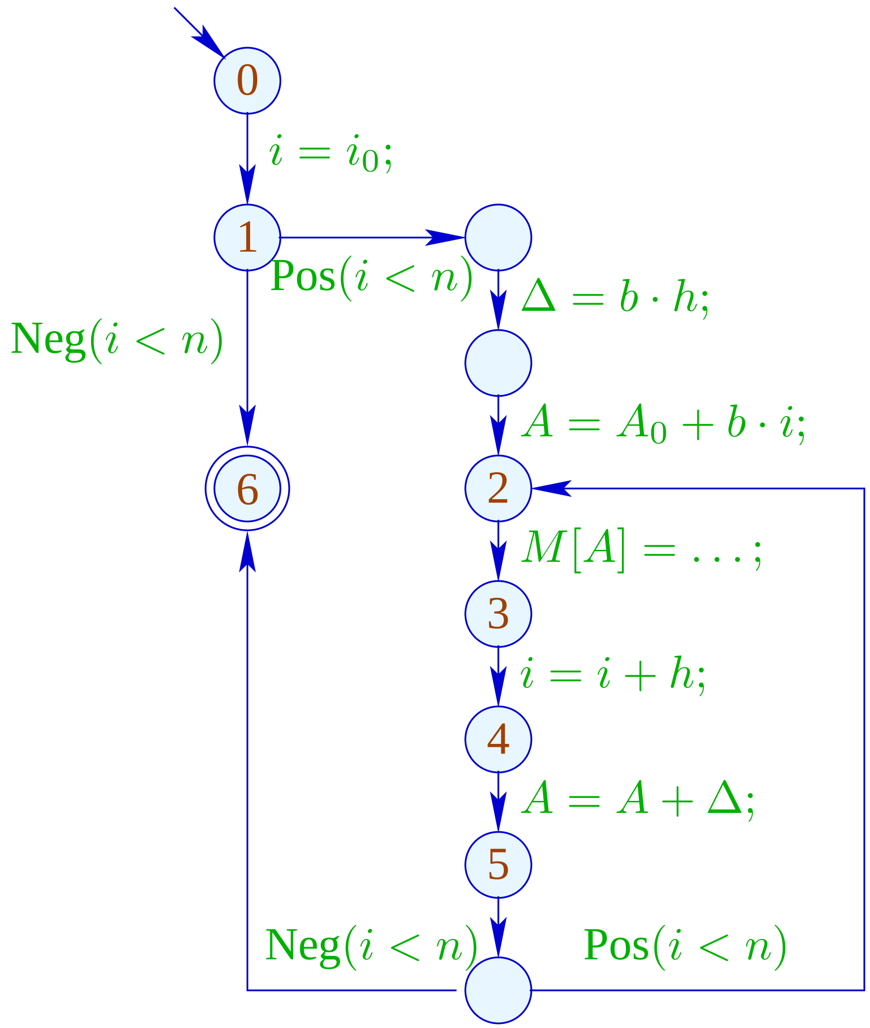
        $M[A] = \ldots;$

        $i = i + h;$

        $A = A + \Delta;$

    } 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 :

  → i may not be used else-where.

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

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

  → b must always be different from zero !!!