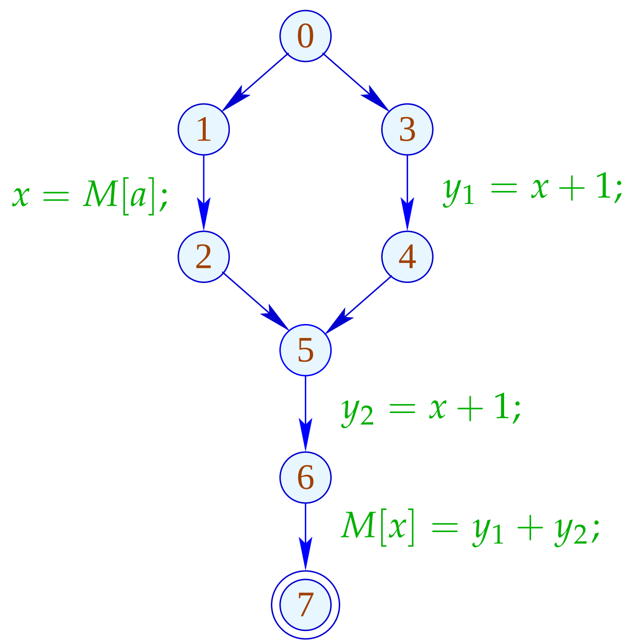
solve x_2	eval x_2 x_3	solve x_3	eval x_3 x_1	solve x_1	eval x_1 x_3	solve x_3
						stable!
					$I[x_3] = \{x_1\}$	
					$\Rightarrow \quad \emptyset$	
				$\boxed{D[x_1] = \{a\}}$		
			$I[x_1] = \{x_3\}$			
			$\Rightarrow \quad \{a\}$			
		$\boxed{D[x_3] = \{a, c\}}$				
		$I[x_3] = \emptyset$				
		solve x_1	eval x_1 x_3	solve x_3		
				stable!		
			$I[x_3] = \{x_1\}$			
			$\Rightarrow \quad \{a, c\}$			
		$\boxed{D[x_1] = \{a, c\}}$				
		$I[x_1] = \emptyset$				
		solve x_3	eval x_3 x_1	solve x_1		
				stable!		
			$I[x_1] = \{x_3\}$			
			$\Rightarrow \quad \{a, c\}$			
		$\boxed{\text{ok}}$				
	$I[x_3] = \{x_1, x_2\}$					
	$\Rightarrow \quad \{a, c\}$					
$\boxed{D[x_2] = \{a\}}$						

- $\longrightarrow$ Evaluation starts with an interesting unknown x_i (e.g., the value at *stop*)
- $\longrightarrow$ Then automatically all unknowns are evaluated which influence x_i :-)
- $\longrightarrow$ The number of evaluations is often smaller than during worklist iteration ;-)
- $\longrightarrow$ The algorithm is more complex but does not rely on pre-computation of variable dependencies :-))
- $\longrightarrow$ It also works if variable dependencies during iteration change !!!

$\Longrightarrow$ interprocedural analysis

1.7 Eliminating Partial Redundancies

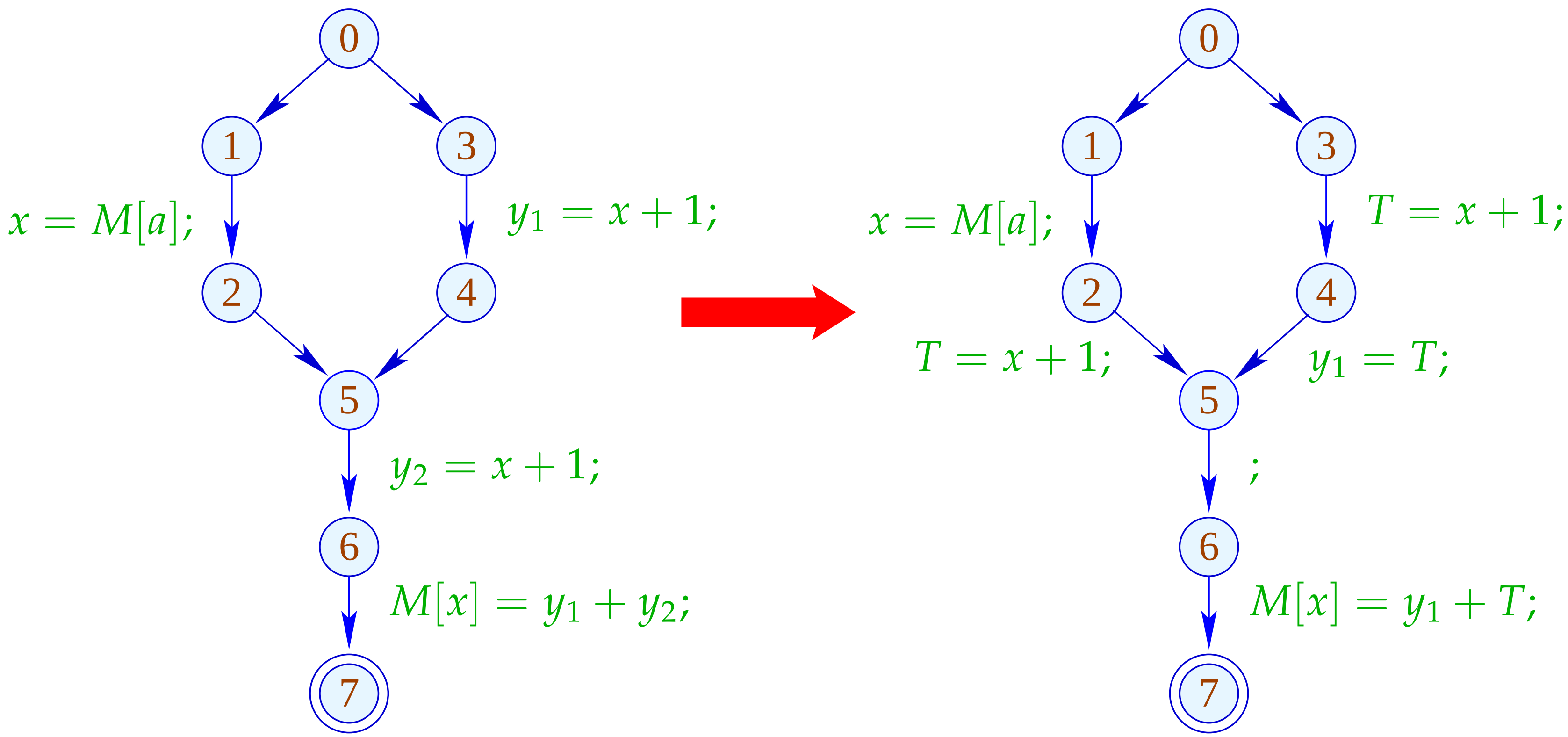
Example:



// $x + 1$ is evaluated on every path ...

// on one path, however, even twice :-(

Goal:


0
1
3
$x = M[a];$
$y_1 = x + 1;$
2
4
5
$y_2 = x + 1;$
6
$M[x] = y_1 + y_2;$
7
0
1
3
$x = M[a];$
$T = x + 1;$
2
4
$T = x + 1;$
$y_1 = T;$
5
;
6
$M[x] = y_1 + T;$
7

Idea:

(1) Insert assignments $T_e = e;$ such that e is available at all points where the value of e is required.

(2) Thereby spare program points where e either is already available or will definitely be computed in future.

Expressions with the latter property are called very busy.

(3) Replace the original evaluations of e by accesses to the variable T_e.

$\Longrightarrow$ we require a novel analysis :-))

An expression e is called busy along a path π , if the expression e is evaluated before any of the variables $x \in \mathit{Vars}(e)$ is overwritten.

$//$ backward analysis!

e is called very busy at u , if e is busy along every path $\pi : u \longrightarrow^* \mathit{stop}$.

An expression e is called busy along a path π, if the expression e is evaluated before any of the variables $x \in \mathit{Vars}(e)$ is overwriten.

// backward analysis!

e is called very busy at u, if e is busy along every path $\pi : u \to^* \mathit{stop}$.

Accordingly, we require:

$$\mathcal{B}[u] \;=\; \bigcap \{ [\![\pi]\!]^\sharp\, \emptyset \mid \pi : u \to^* \mathit{stop} \}$$

where for $\pi = k_1 \ldots k_m$:

$$[\![\pi]\!]^\sharp \;=\; [\![k_1]\!]^\sharp \circ \ldots \circ [\![k_m]\!]^\sharp$$

Our complete lattice is given by:

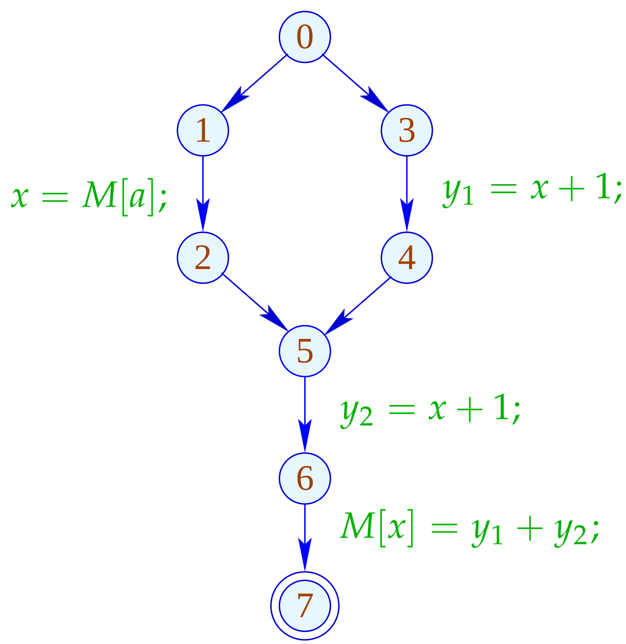
$$\mathbb{B} = 2^{Expr \setminus Vars} \qquad \text{with} \quad \sqsubseteq \;=\; \supseteq$$

The effect $[\![k]\!]^\sharp$ of an edge $k = (u, lab, v)$ only depends on lab, i.e., $[\![k]\!]^\sharp = [\![lab]\!]^\sharp$ where:

$$\begin{array}{lll}
[\![;]\!]^\sharp\, B & = & B \\
[\![Pos(e)]\!]^\sharp\, B & = & [\![Neg(e)]\!]^\sharp\, B \qquad = \quad B \cup \{e\} \\
[\![x = e;]\!]^\sharp\, B & = & (B \setminus Expr_x) \cup \{e\} \\
[\![x = M[e];]\!]^\sharp\, B & = & (B \setminus Expr_x) \cup \{e\} \\
[\![M[e_1] = e_2;]\!]^\sharp\, B & = & B \cup \{e_1, e_2\}
\end{array}$$

These effects are all distributive. Thus, the least solution of the constraint system yields precisely the MOP — given that *stop* is reachable from every program point :-)

Example:



7	$\emptyset$
6	$\emptyset$
5	$\{x + 1\}$
4	$\{x + 1\}$
3	$\{x + 1\}$
2	$\{x + 1\}$
1	$\emptyset$
0	$\emptyset$

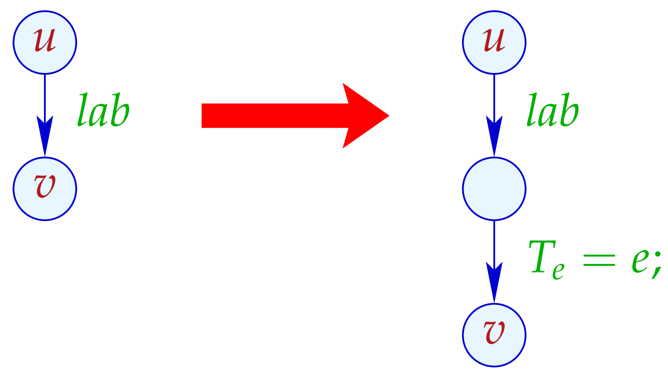
A point u is called safe for e, if $e \in \mathcal{A}[u] \cup \mathcal{B}[u]$, i.e., e is either available or very busy.

Idea:

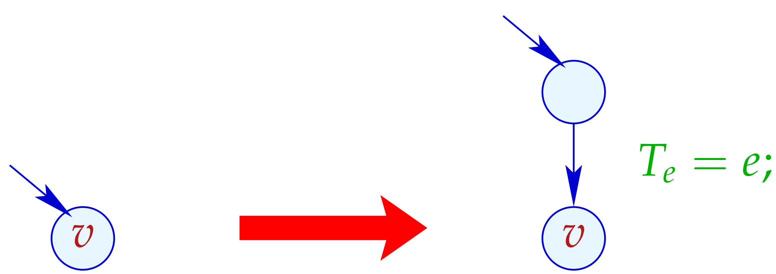
- We insert computations of e such that e becomes available at all safe program points :-)
- We insert $T_e = e;$ after every edge (u, lab, v) with

$$e \in \mathcal{B}[v] \backslash [\![lab]\!]^\sharp_{\mathcal{A}}(\mathcal{A}[u] \cup \mathcal{B}[u])$$

Transformation 5.1:



$$(e \in \mathcal{B}[v] \backslash [\![lab]\!]^\sharp_{\mathcal{A}} (\mathcal{A}[u] \cup \mathcal{B}[u]))$$



$$(e \in \mathcal{B}[v])$$

Transformation 5.2:

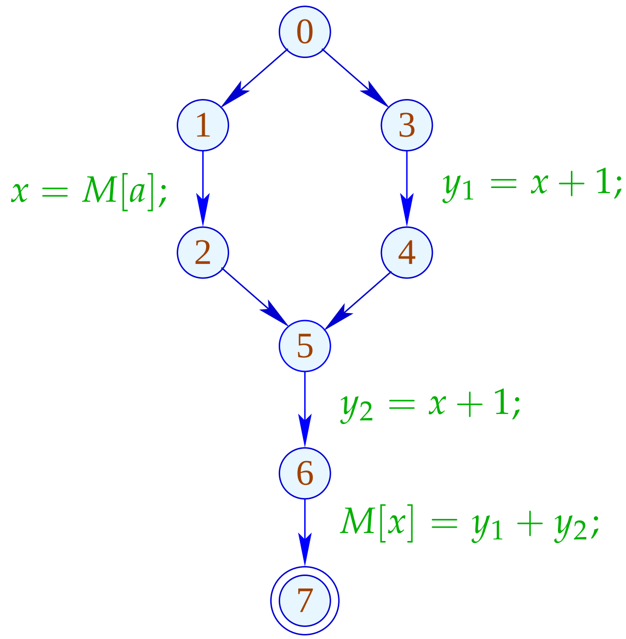
$//$ analogously for the other uses of e

$//$ at old edges of the program.

Bernhard Steffen, Dortmund

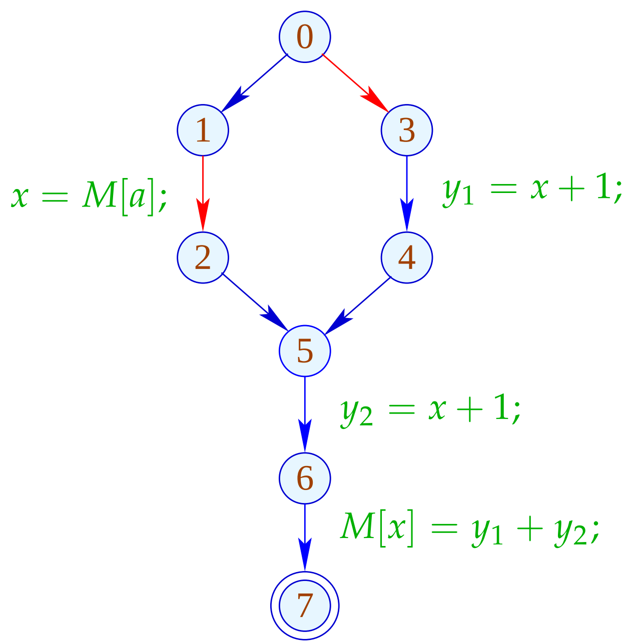
Jens Knoop, Wien

In the Example:



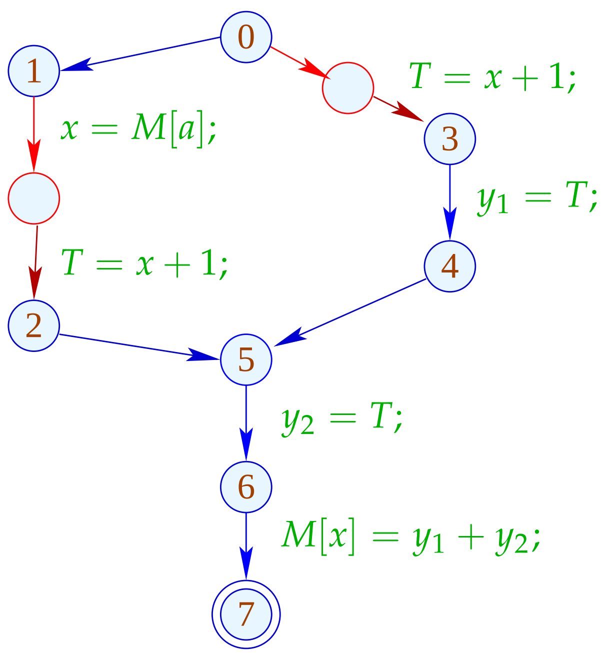
	$\mathcal{A}$	$\mathcal{B}$
0	$\emptyset$	$\emptyset$
1	$\emptyset$	$\emptyset$
2	$\emptyset$	$\{x+1\}$
3	$\emptyset$	$\{x+1\}$
4	$\{x+1\}$	$\{x+1\}$
5	$\emptyset$	$\{x+1\}$
6	$\{x+1\}$	$\emptyset$
7	$\{x+1\}$	$\emptyset$

In the Example:



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

Im Example:



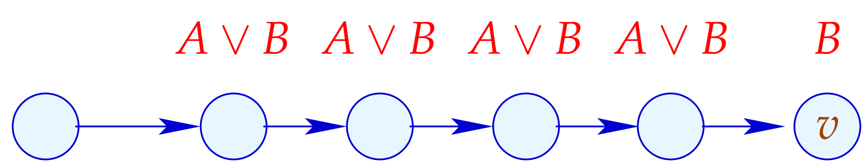
	$\mathcal{A}$	$\mathcal{B}$
0	$\emptyset$	$\emptyset$
1	$\emptyset$	$\emptyset$
2	$\emptyset$	$\{x+1\}$
3	$\emptyset$	$\{x+1\}$
4	$\{x+1\}$	$\{x+1\}$
5	$\emptyset$	$\{x+1\}$
6	$\{x+1\}$	$\emptyset$
7	$\{x+1\}$	$\emptyset$

Correctness:

Let π denote a path reaching v after which a computation of an edge with e follows.

Then there is a maximal suffix of π such that for every edge $k = (u, lab, u')$ in the suffix:

$$e \in [\![lab]\!]^\sharp_{\mathcal{A}}(\mathcal{A}[u] \cup \mathcal{B}[u])$$



Correctness:

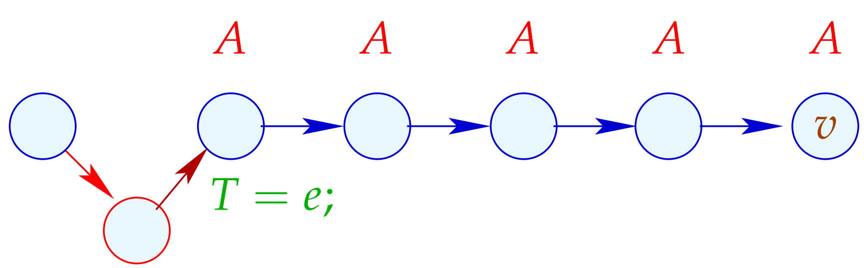
Let π denote a path reaching v after which a computation of an edge with e follows.

Then there is a maximal suffix of π such that for every edge $k = (u, lab, u')$ in the suffix:

$$e \in [\![lab]\!]^\sharp_\mathcal{A}(\mathcal{A}[u] \cup \mathcal{B}[u])$$

In particular, no variable in e receives a new value :-)

Then $T_e = e;$ is inserted before the suffix :-))



We conclude:

- Whenever the value of e is required, e is available :-)

  $\Longrightarrow$ correctness of the transformation

- Every $T = e;$ which is inserted into a path corresponds to an e which is replaced with T :-))

  $\Longrightarrow$ non-degradation of the efficiency

1.8 Application: Loop-invariant Code

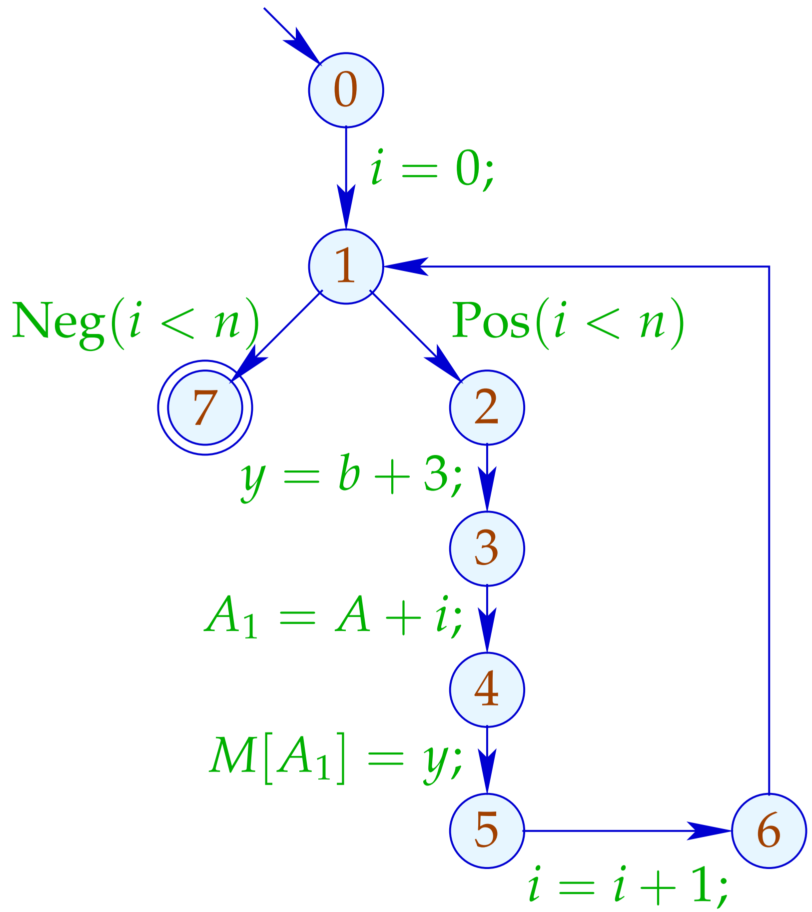
Example:

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

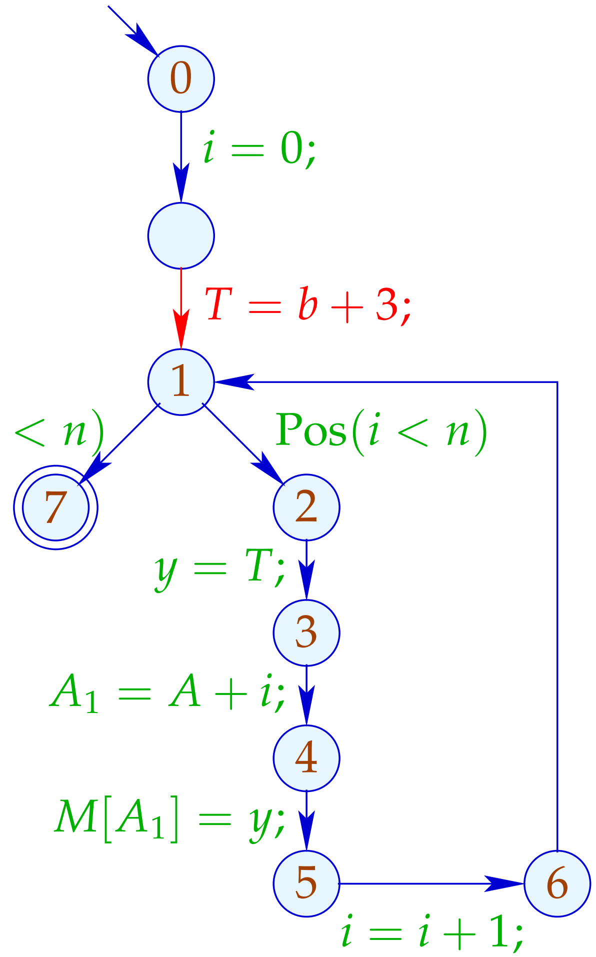
// The expression $b + 3$ is recomputed in every iteration :-(

// This should be avoided :-)

The Control-flow Graph:

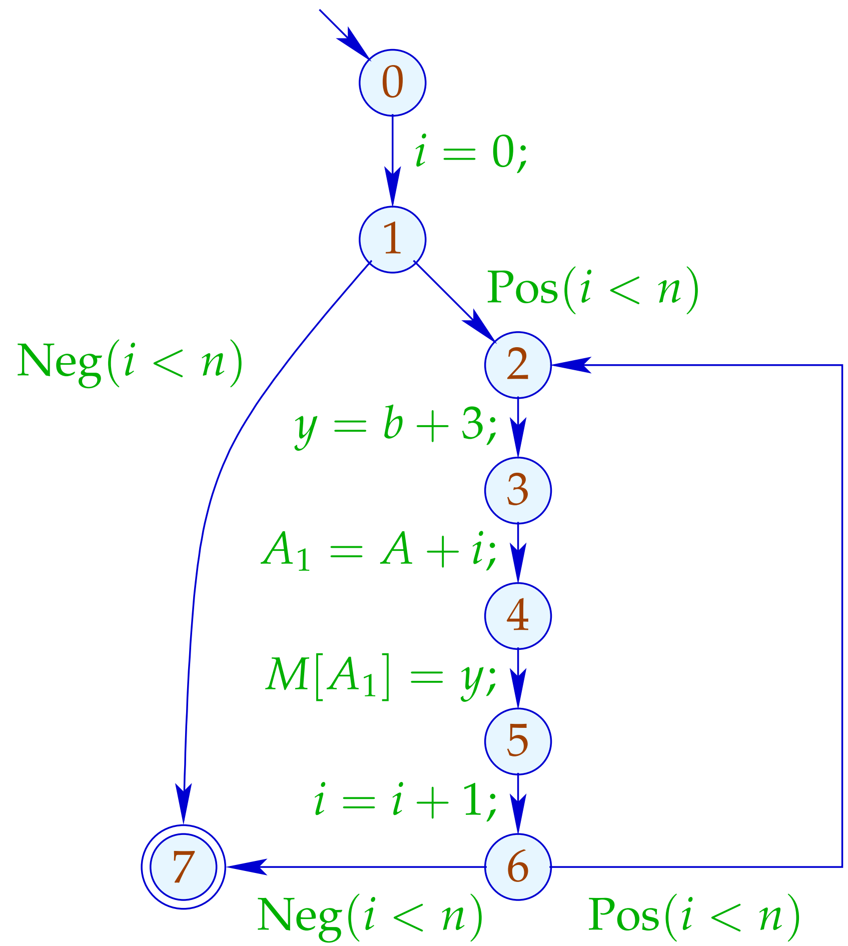


Warning: $T = b + 3;$ may not be placed before the loop :

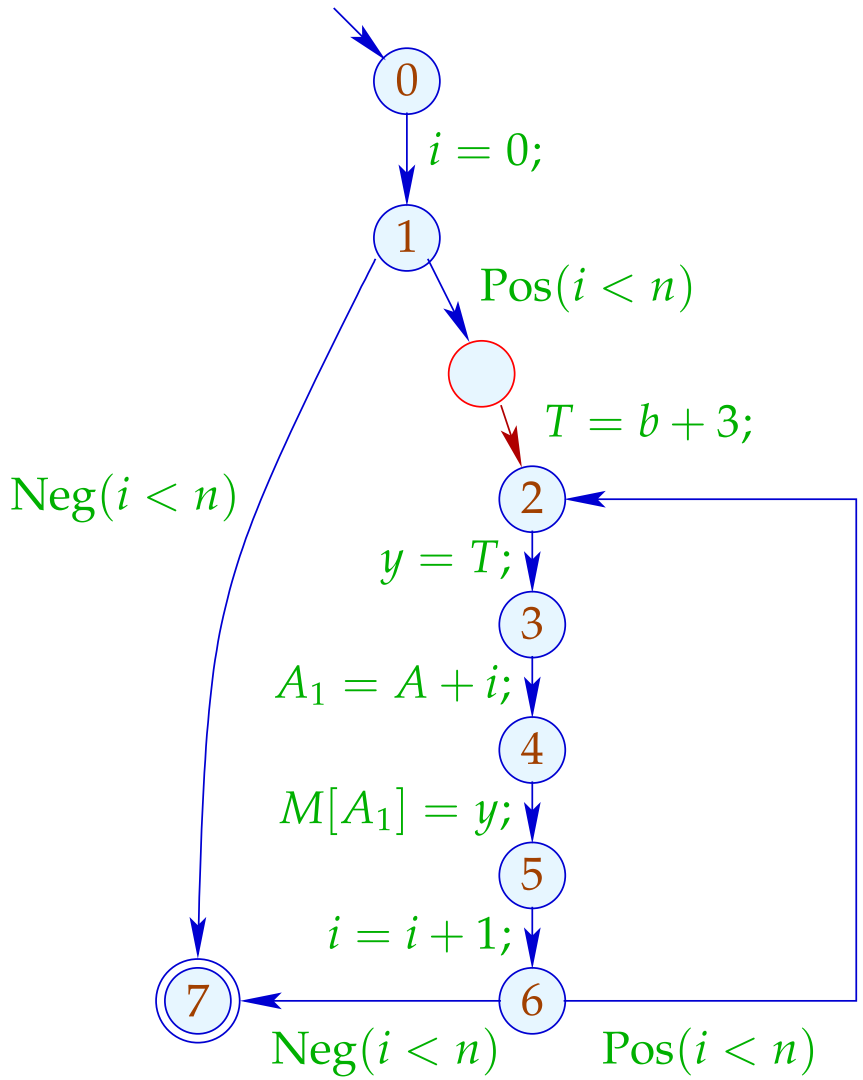


$\Longrightarrow$ There is no decent place for $T = b + 3;$:-(

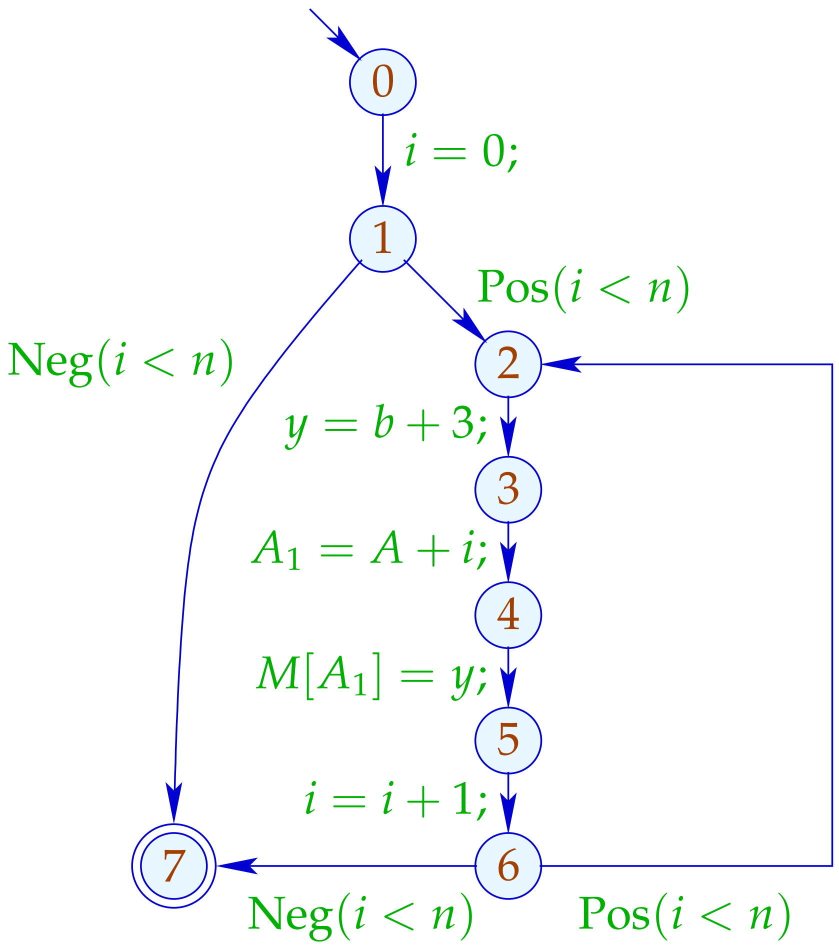
Idea: Transform into a do-while-loop ...



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

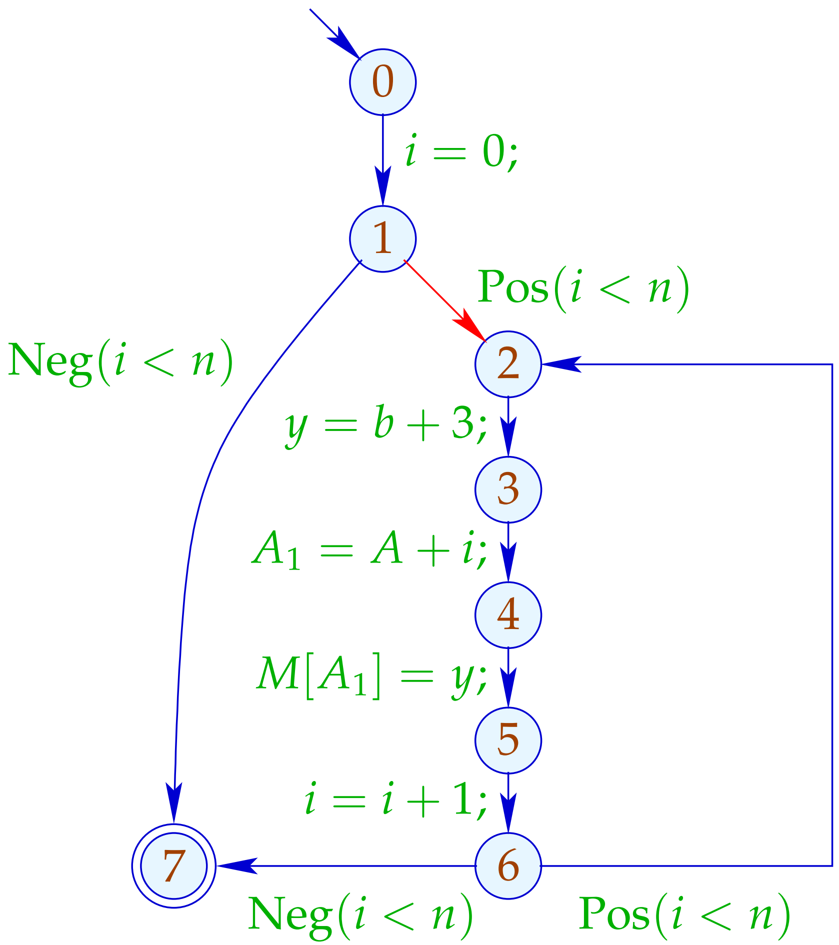


Application of T5 (PRE) :



	$\mathcal{A}$	$\mathcal{B}$
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:

$$\textsf{while}\ (b)\ stmt \quad \Longrightarrow \quad \textsf{if}\ (b)\ \textsf{do}\ stmt\ \textsf{while}\ (b);$$

$\Longrightarrow$ Loop Rotation

Problem:

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

$$\Longrightarrow \qquad \text{Pre-dominators}$$

u pre-dominates v, if every path $\pi : \textit{start} \rightarrow^* v$ 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:

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

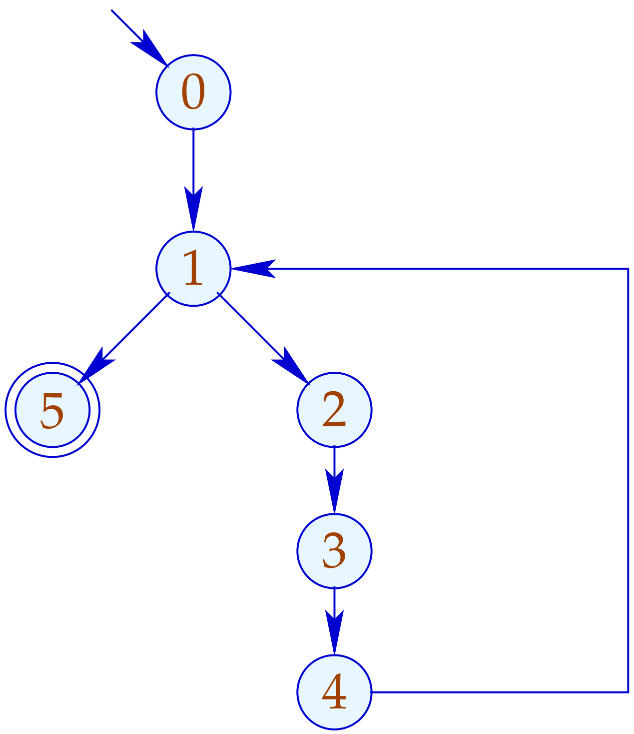
$$[\![(\_, \_, v)]\!]^\sharp \, P \;\; = \;\; P \cup \{v\}$$

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

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

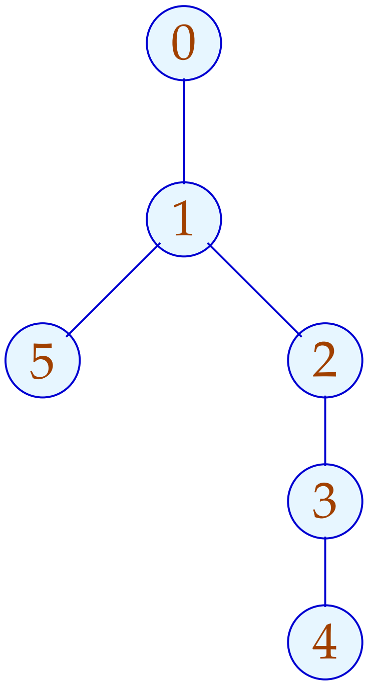
Since $[\![k]\!]^\sharp$ are distributive, the $\mathcal{P}[v]$ can computed by means of fixpoint iteration :-)

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 "$\Rightarrow$" 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.

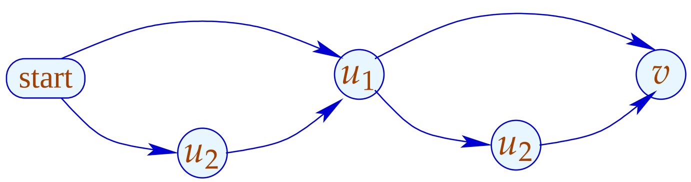
If $u_1 \Rightarrow u_2$ then u_1 not immediate.

Consequently, u_1, u_2 are incomparable :-)

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

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

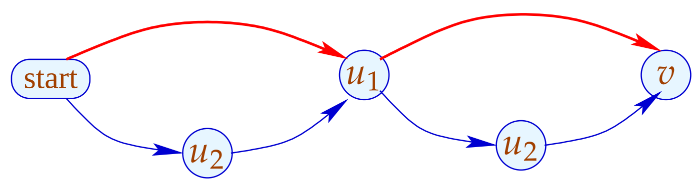
If, however, u_1, u_2 are incomparable, then there is path:
$start \longrightarrow^* v$ avoiding u_2 :



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

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

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



Observation:

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

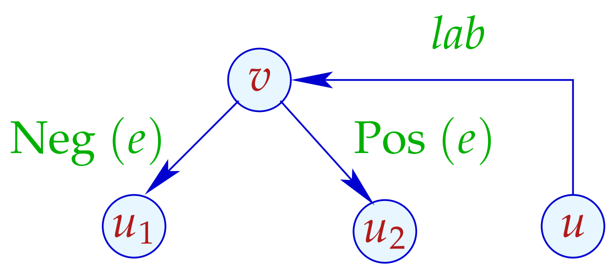
A back edge from the exit u to the loop head v can be identified through

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

:-)

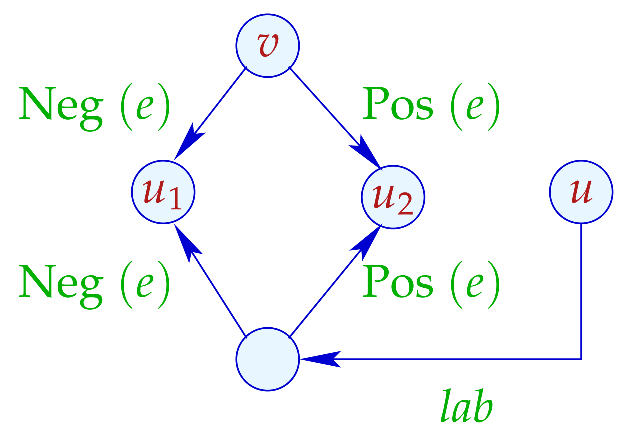
Accordingly, we define:

Transformation 6:



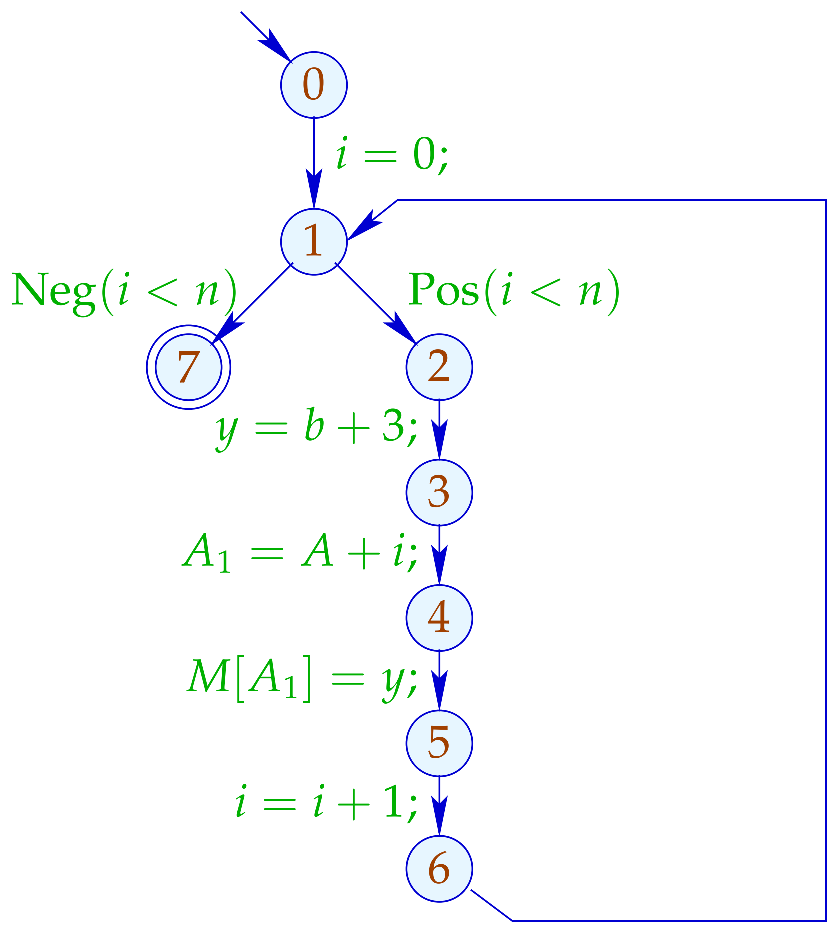
$u_1 \notin \mathcal{P}[u]$

$u_2, v \in \mathcal{P}[u]$

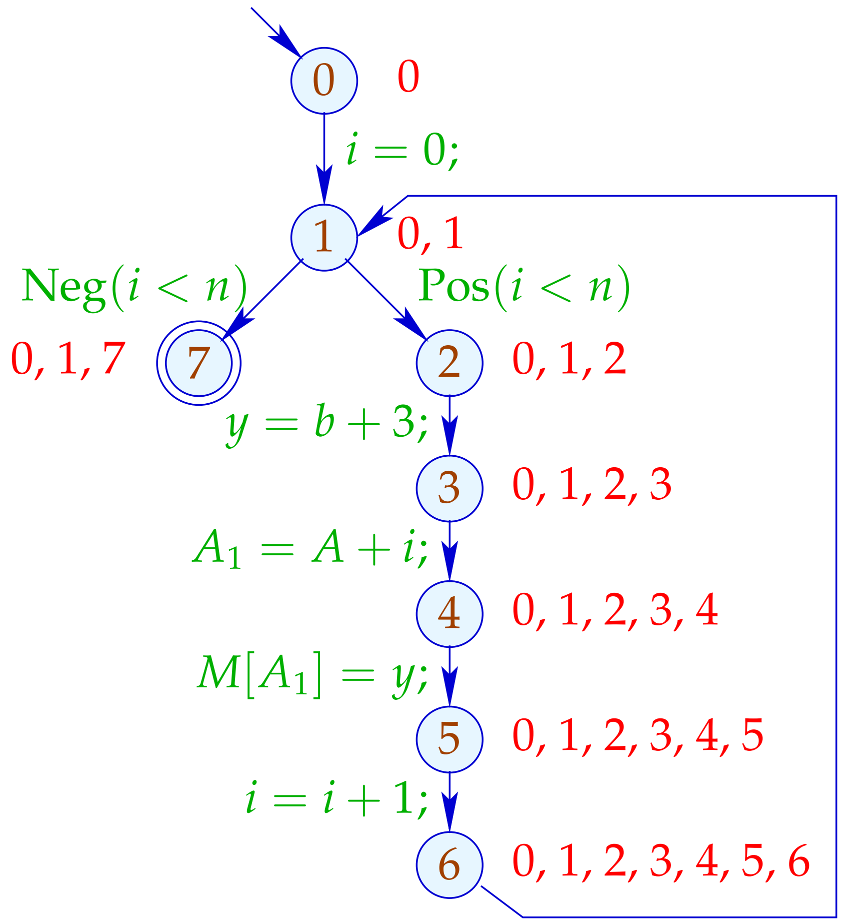


We duplicate the entry check to all back edges :-)

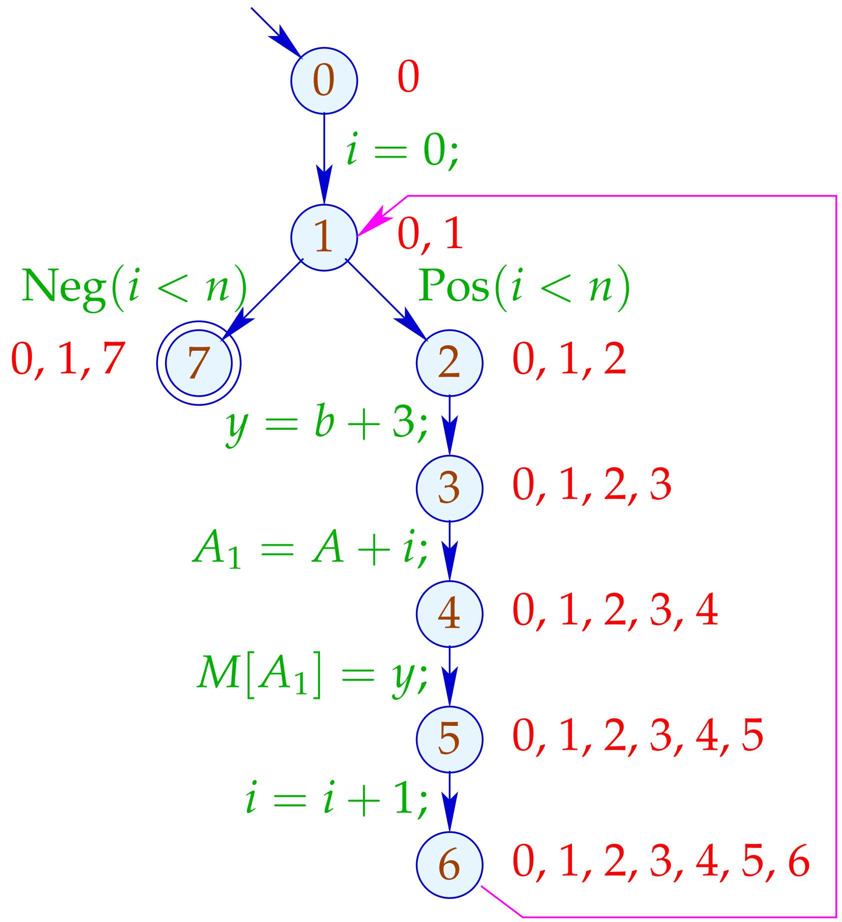
... in the Example:



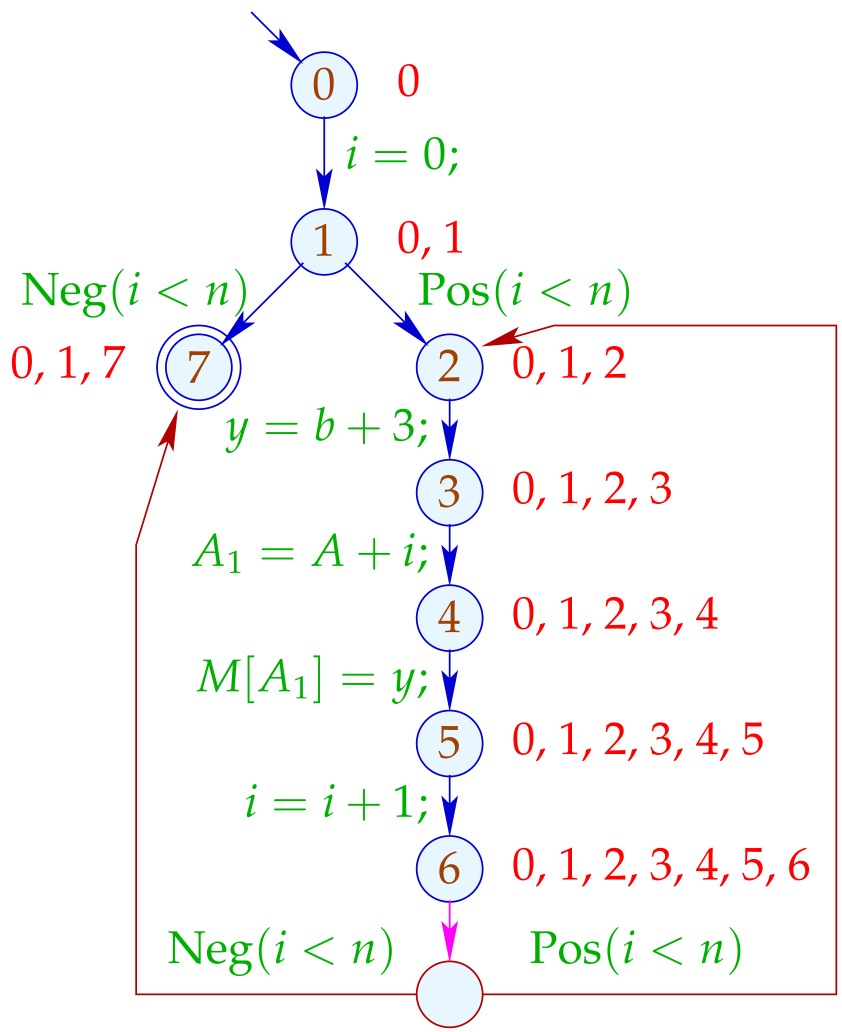
... in the Example:



... in the Example:

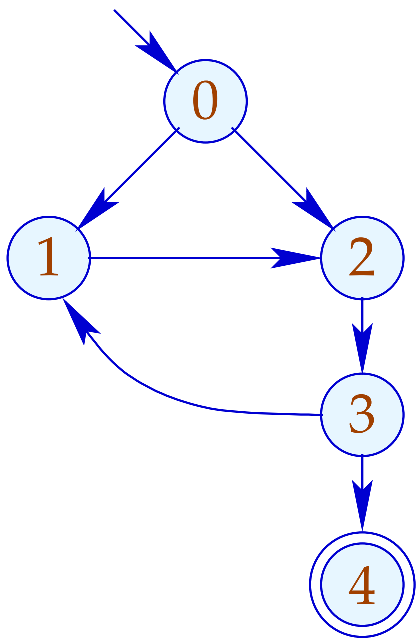


... in the Example:

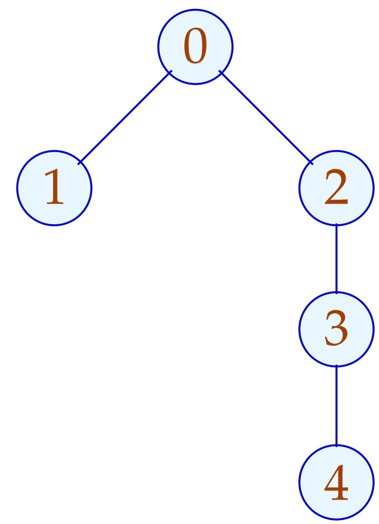


Warning:

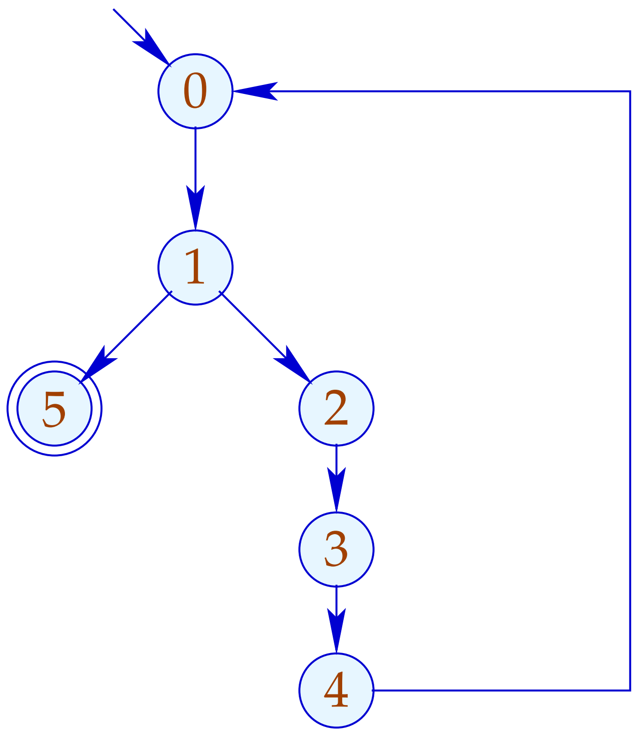
There are unusual loops which cannot be rotated:

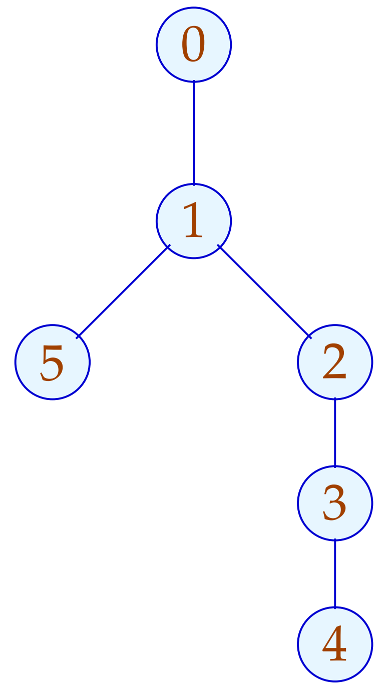


Pre-dominators:



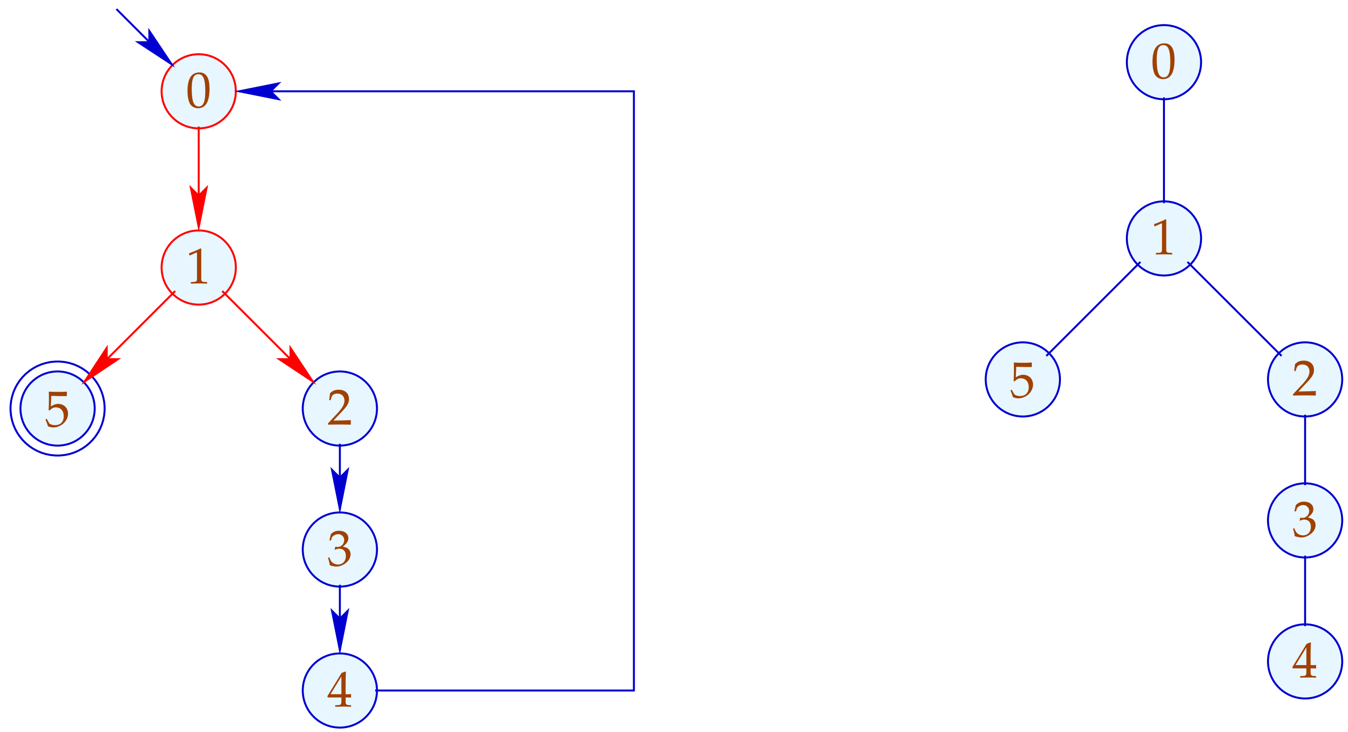
... but also common ones which cannot be rotated:





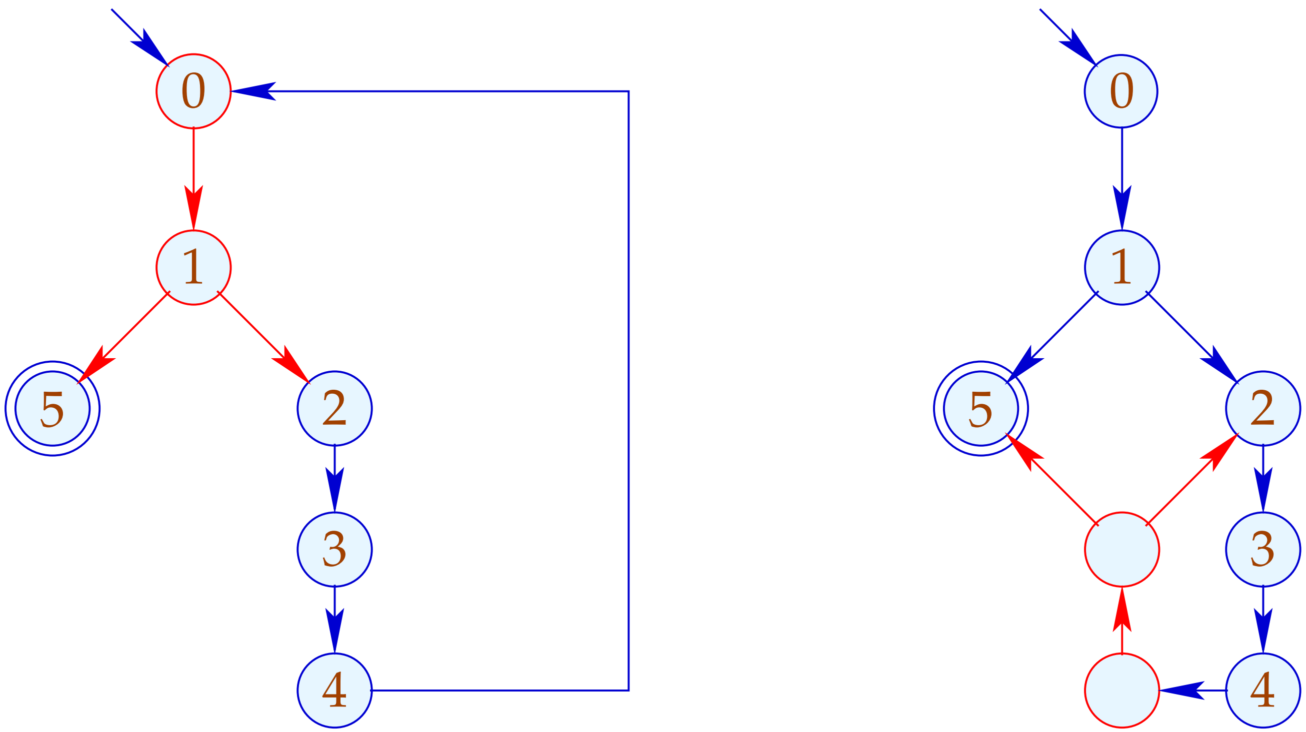
Here, the complete block between back edge and conditional jump should be duplicated :-(

... 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 :-(