

1.14 Application: Loop-invariant Code

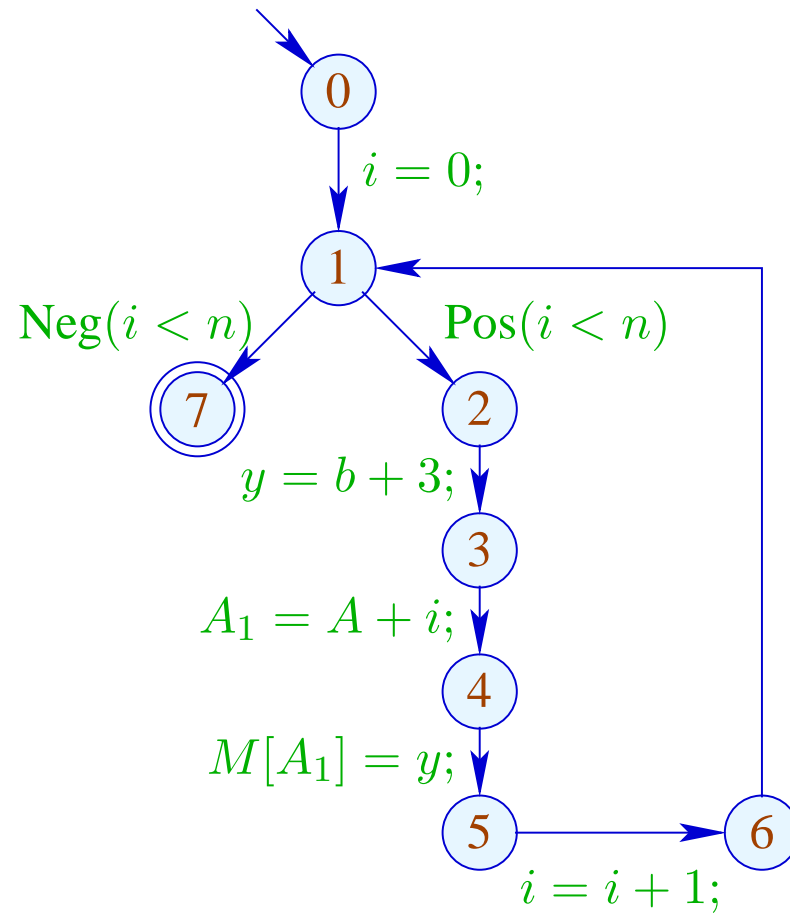
Example:

```
for ( $i = 0; i < n; i++$ )  
     $a[i] = b + 3;$ 
```

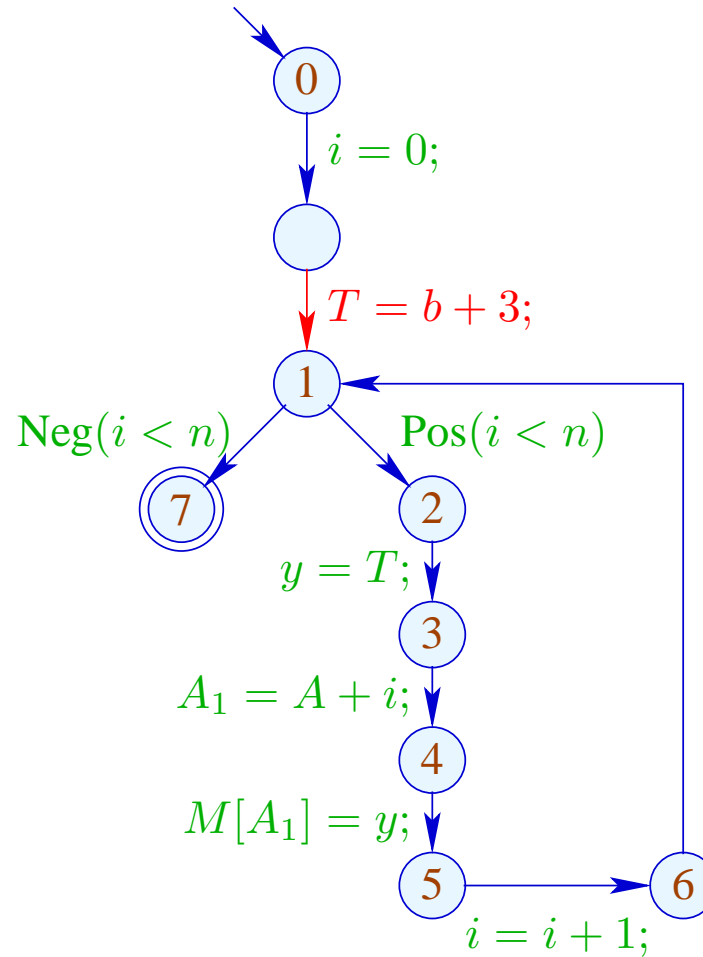
// The expression $b + 3$ is recomputed in every iteration

// This should be avoided

The Control-flow Graph:

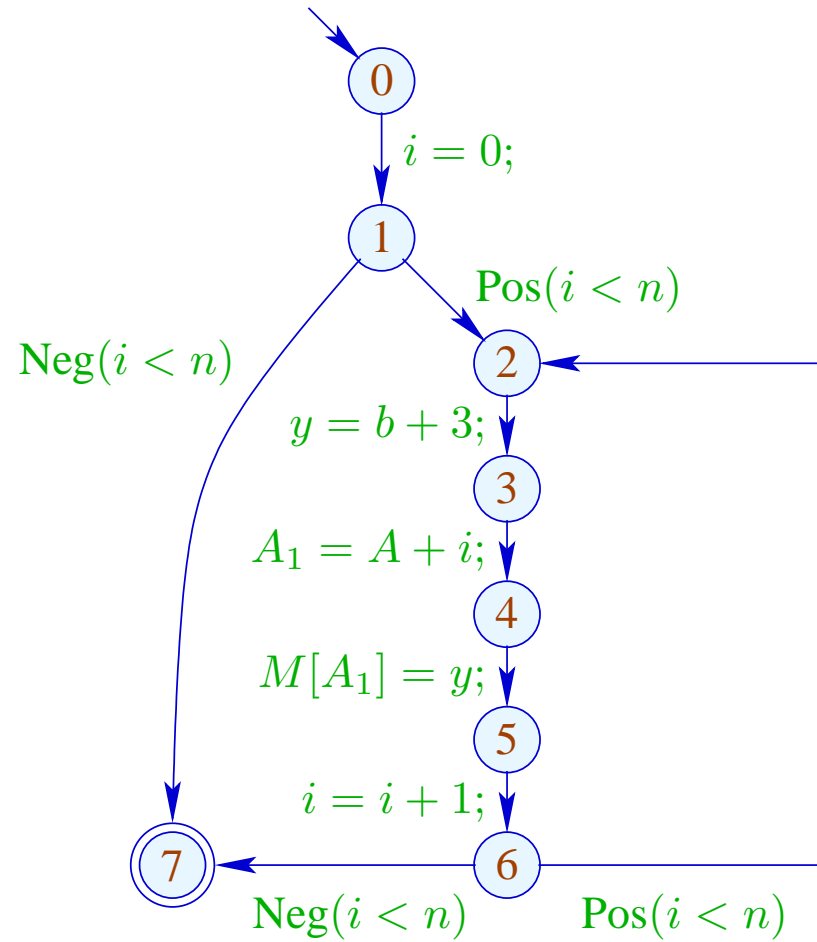


Warning: $T = b + 3;$ may not be placed **before** the loop :

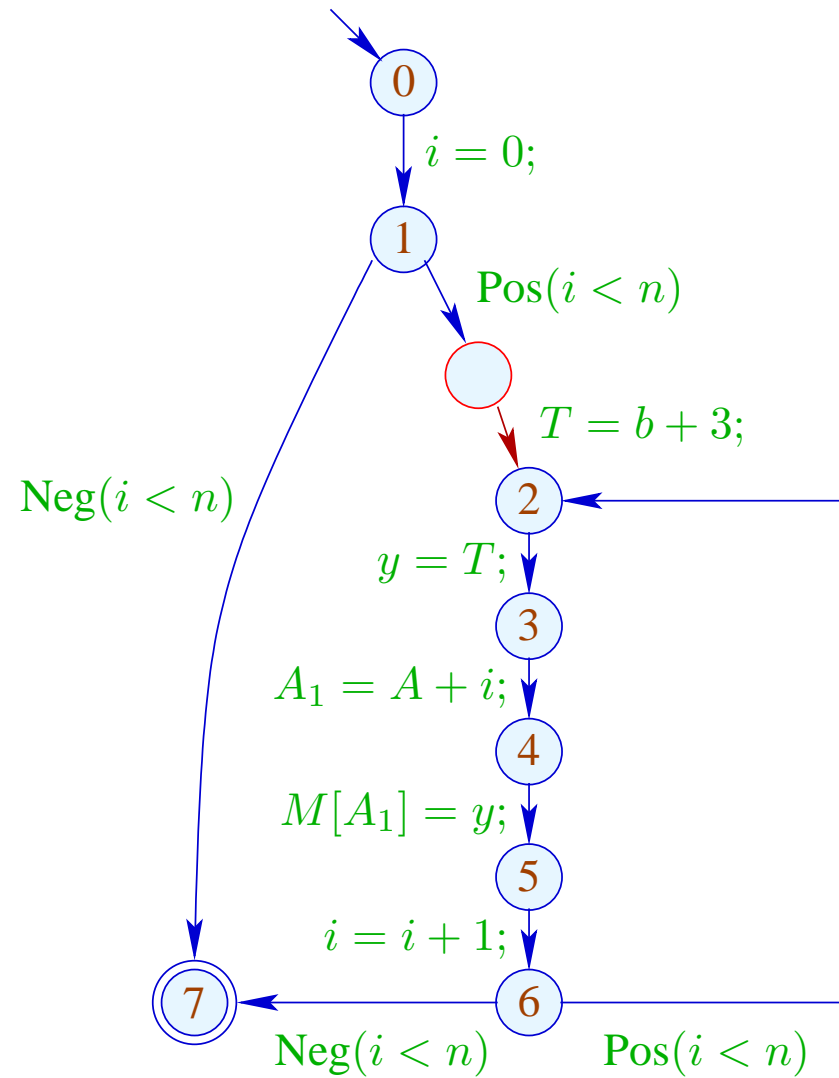


\implies There is no **safe** place to move $T = b + 3;$ to!

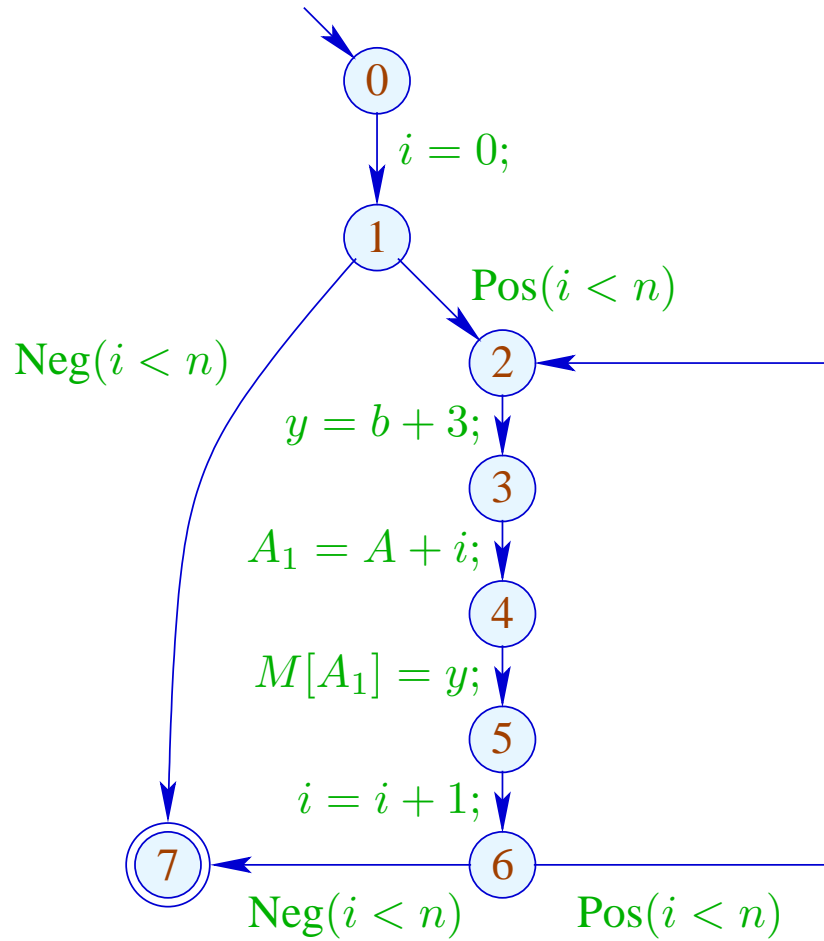
Idea: Transform into a do-while-loop ...



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

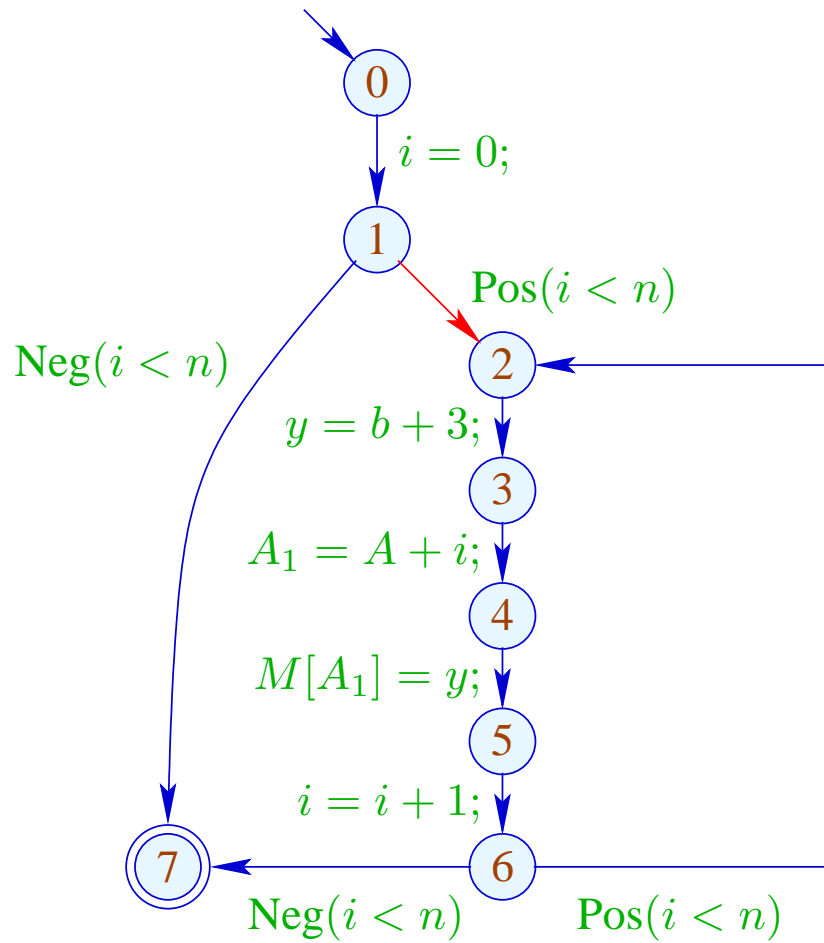


Application of PRE:



	\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 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 a loop
- This only works properly for **do-while-loops**
- To optimize other loops, we transform them into **do-while-loops** before-hand:

`while (b) stmt` \implies `if (b)`
`do stmt`
`while (b);`

\implies **Loop Rotation**

Problem:

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

\implies Pre-dominators

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

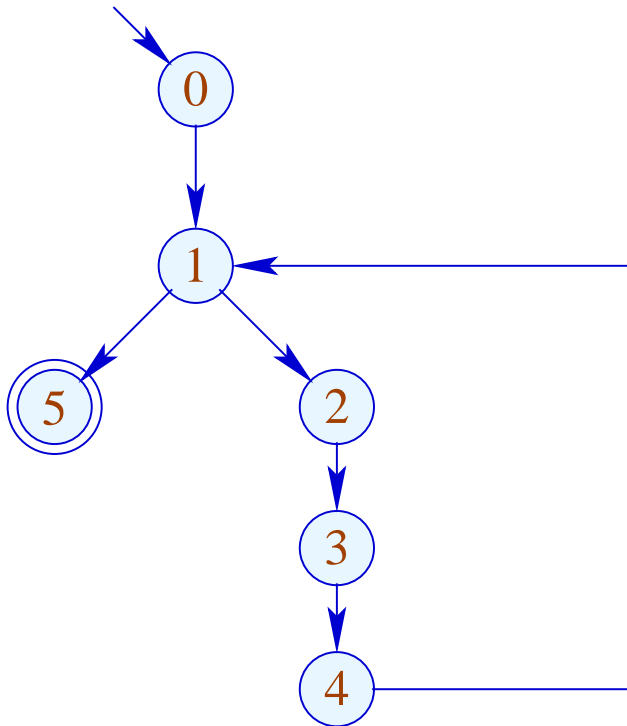
$$\mathbb{P} = 2^{\text{Nodes}}, \quad \sqsubseteq = \supseteq$$
$$\llbracket (-, -, v) \rrbracket^\# P = P \cup \{v\}$$

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

$$\mathcal{P}[v] = \bigcap \{ \llbracket \pi \rrbracket^\# \{start\} \mid \pi : start \rightarrow^* v \}$$

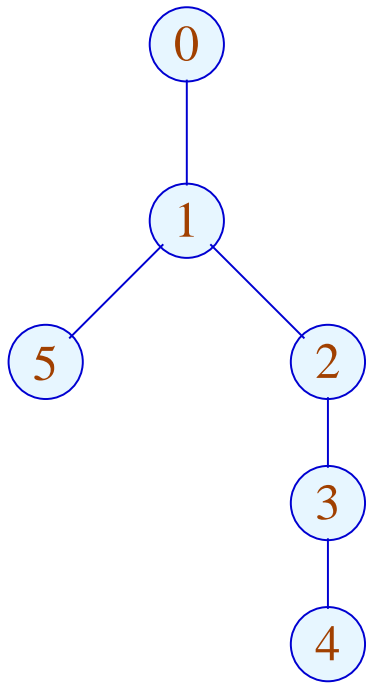
Since $\llbracket k \rrbracket^\#$ are distributive, the $\mathcal{P}[v]$ can be computed by means of fixed-point iteration

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 exist $u_1 \neq u_2$ that immediately pre-dominate v .

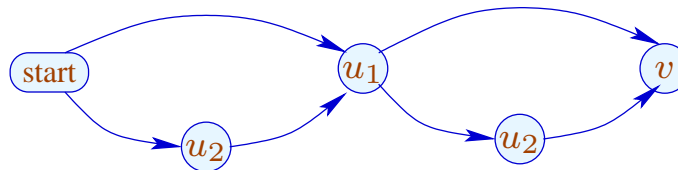
If $u_1 \Rightarrow u_2$ then u_1 not immediate.

Consequently, u_1, u_2 are incomparable

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

$$\pi = \pi_1 \pi_2 \quad \text{with} \quad \begin{aligned} \pi_1 &: \text{start} \rightarrow^* u_1 \\ \pi_2 &: u_1 \rightarrow^* v \end{aligned}$$

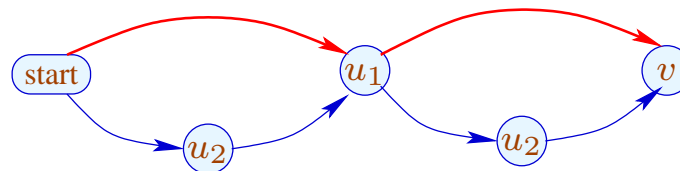
If u_1, u_2 are incomparable, then there is a path: $\text{start} \rightarrow^* v$
bypassing u_2 :



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

$$\pi = \pi_1 \pi_2 \quad \text{with} \quad \begin{aligned} \pi_1 &: \textit{start} \rightarrow^* u_1 \\ \pi_2 &: u_1 \rightarrow^* v \end{aligned}$$

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



Observation:

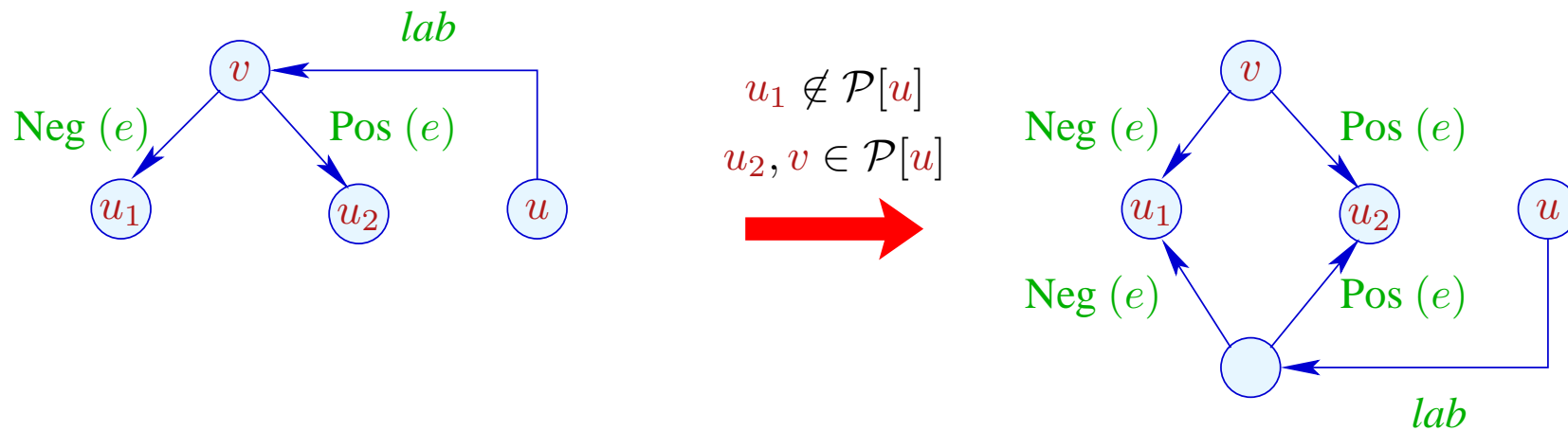
The loop head of a **while**-loop pre-dominates every node in the body.

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

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

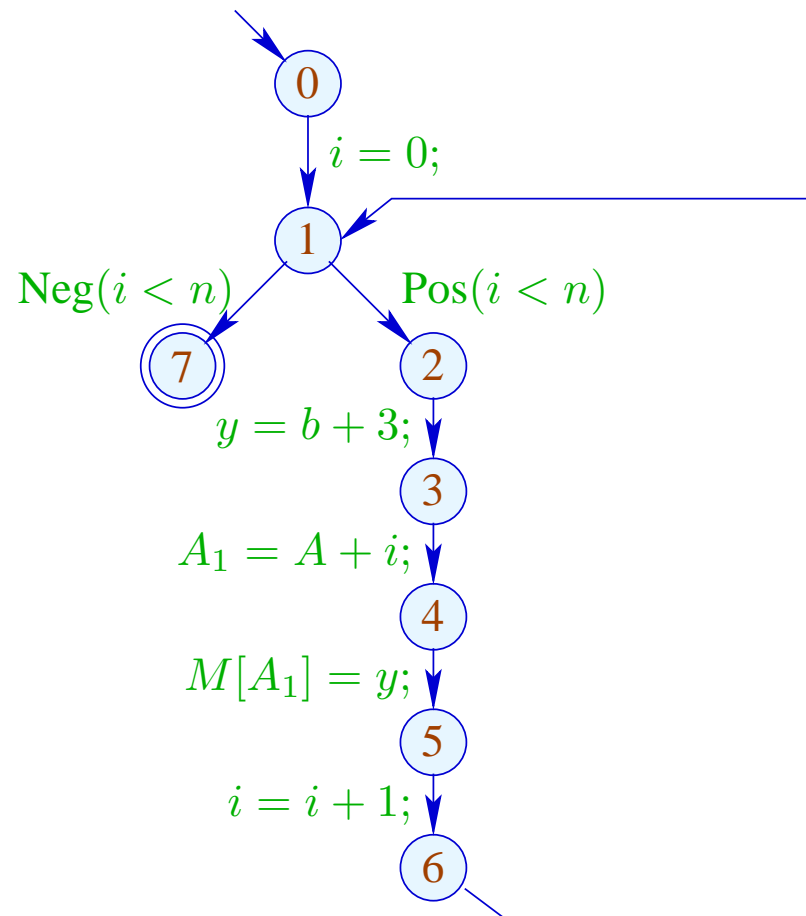
Accordingly, we define:

Transformation LR:

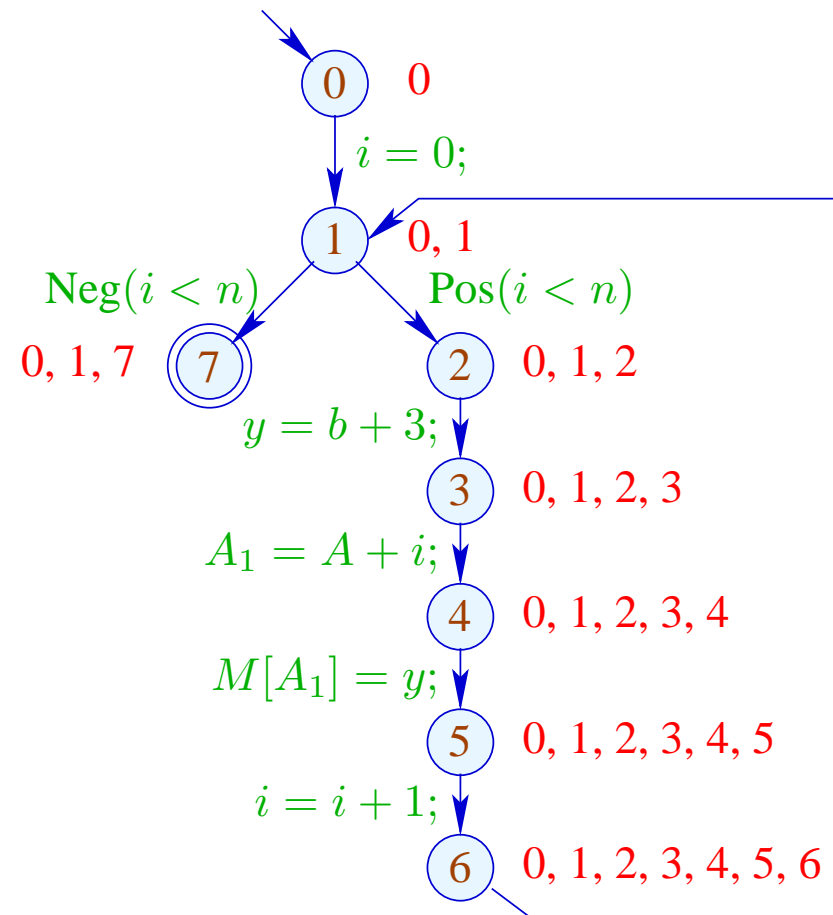


We duplicate the entry check to all back edges

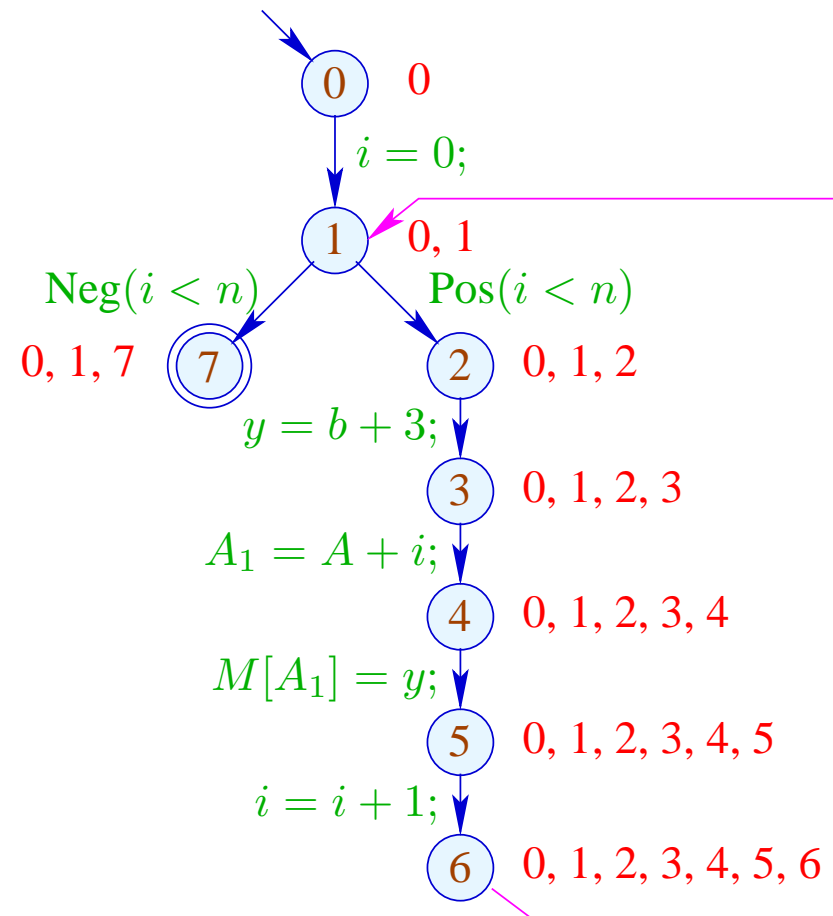
... in the Example:



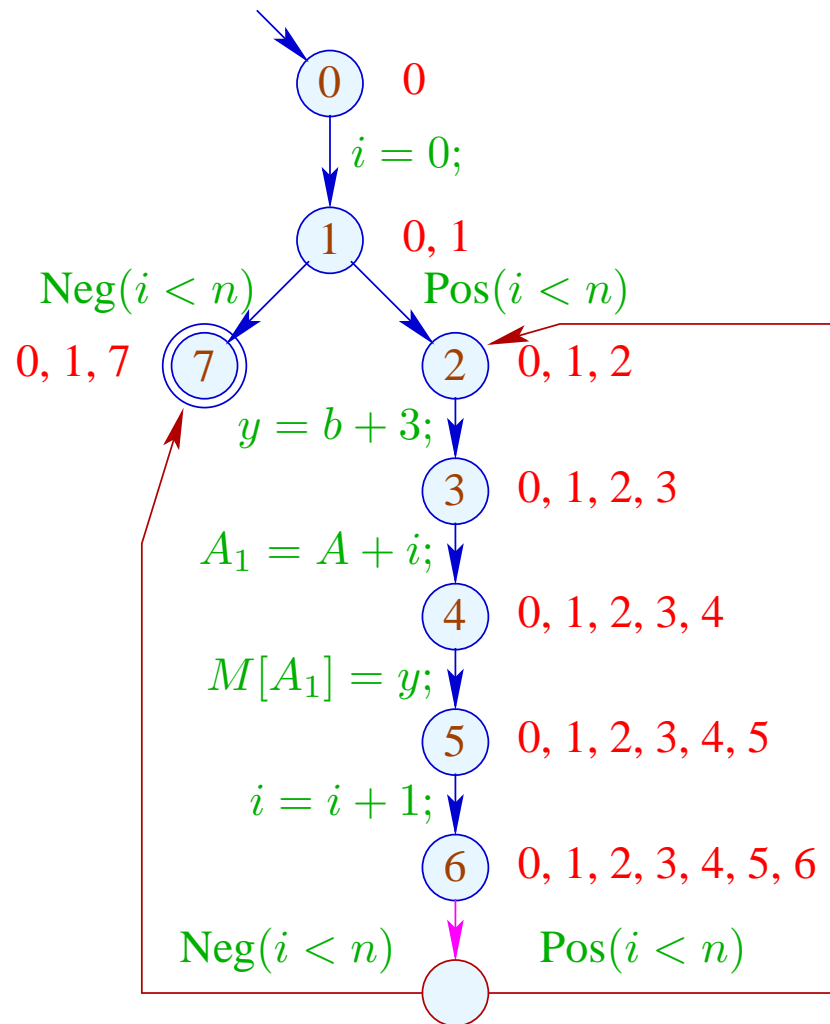
... in the Example:



... in the Example:

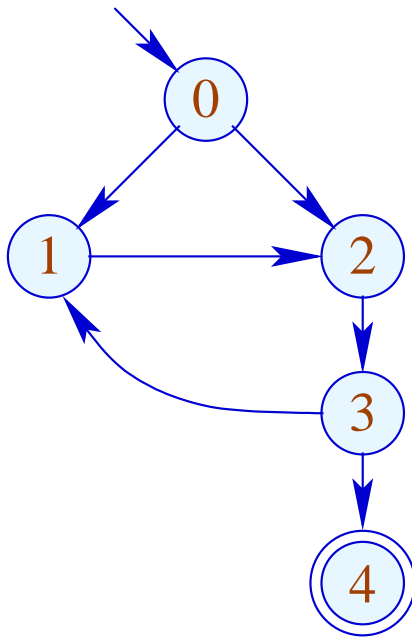


... in the Example:

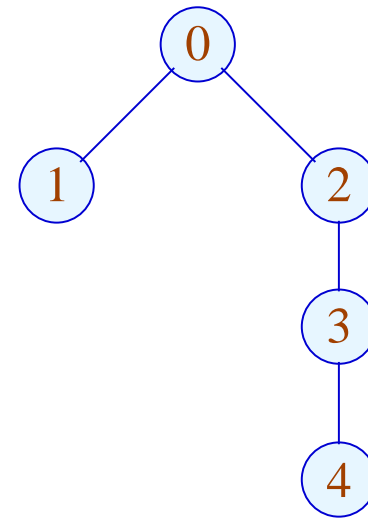


Warning:

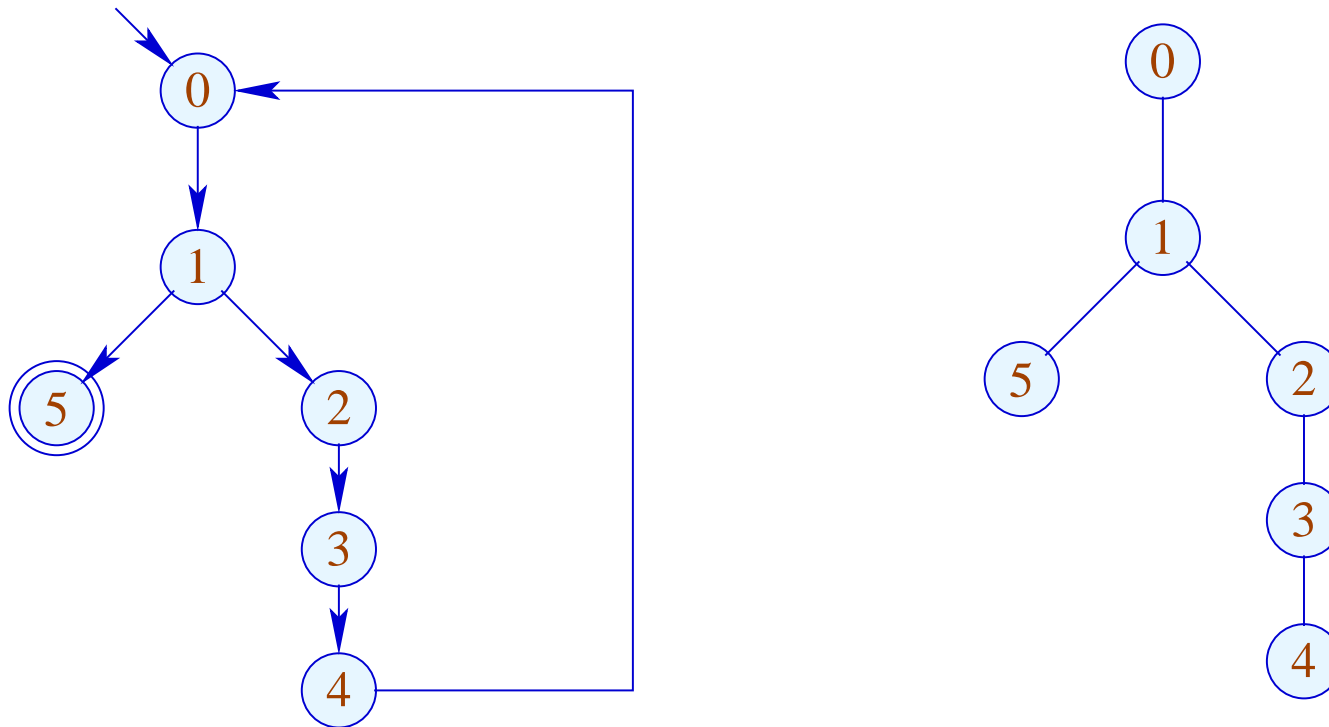
There are **unusual** loops which cannot be rotated:



Pre-dominators:

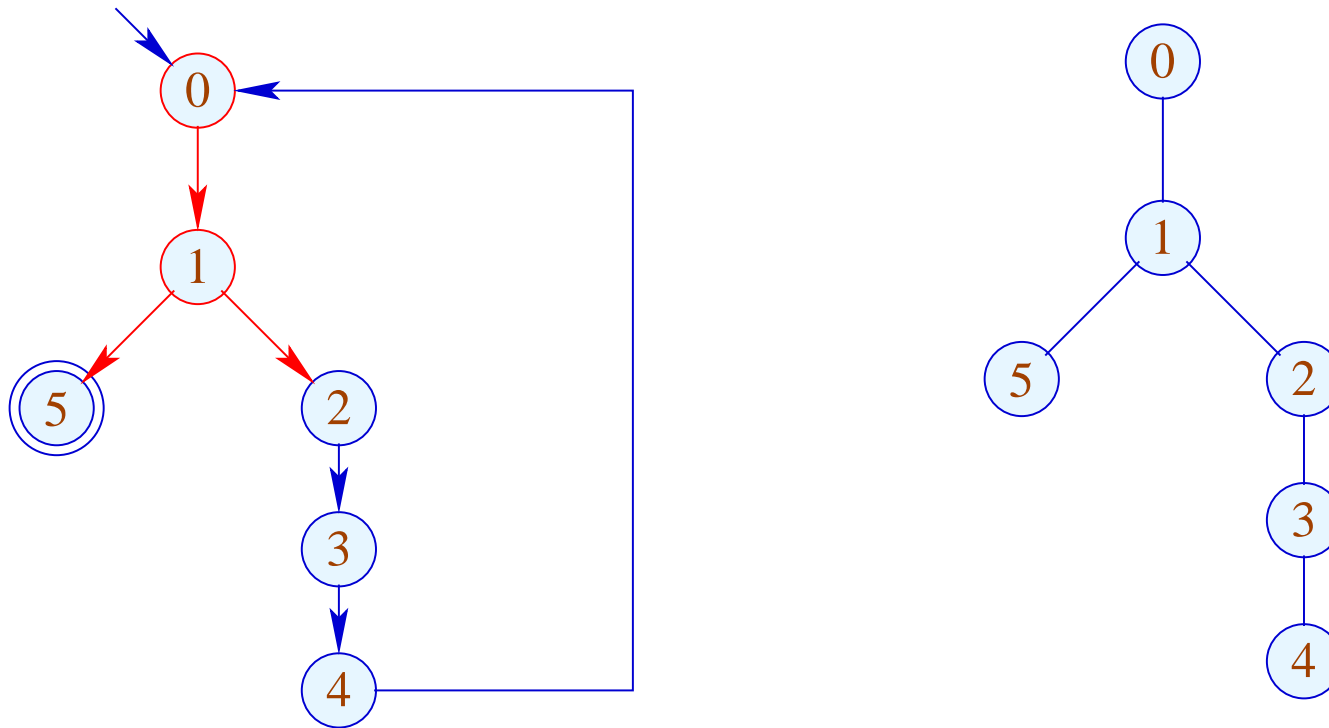


... but also **common ones** which cannot be rotated:



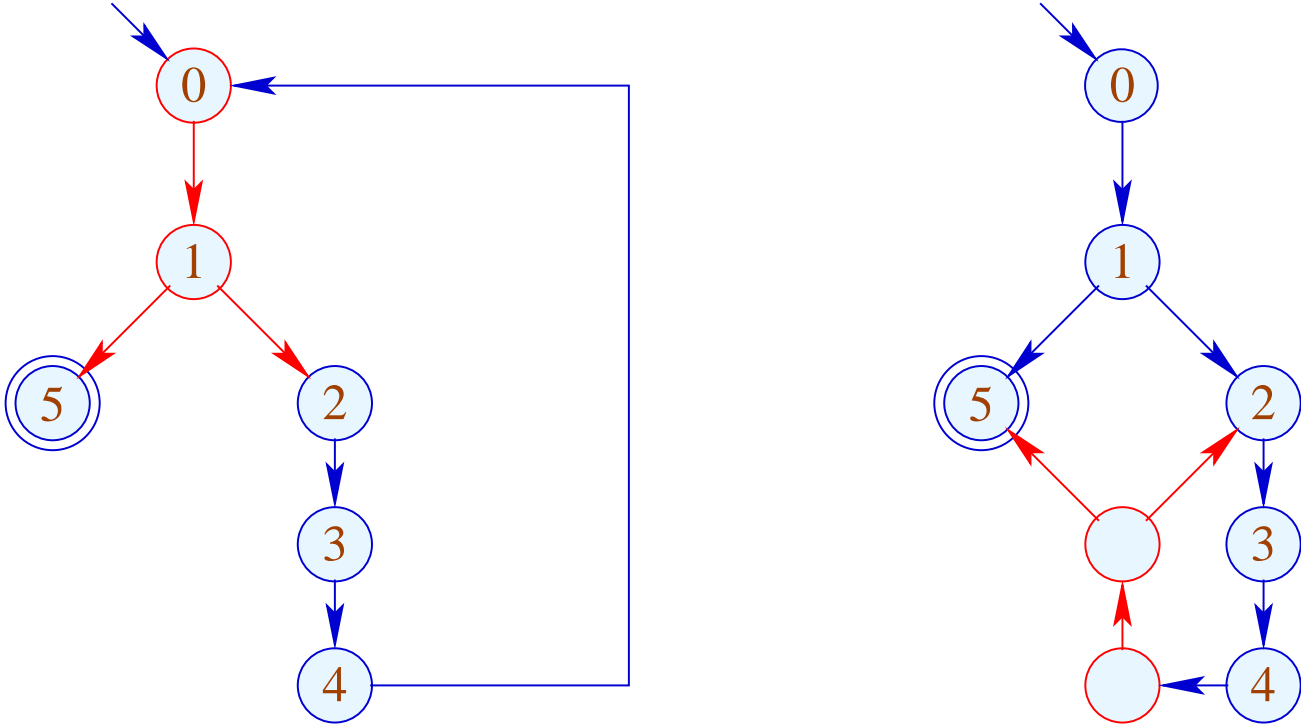
Here, the complete block between back edge and conditional jump should be duplicated

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



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

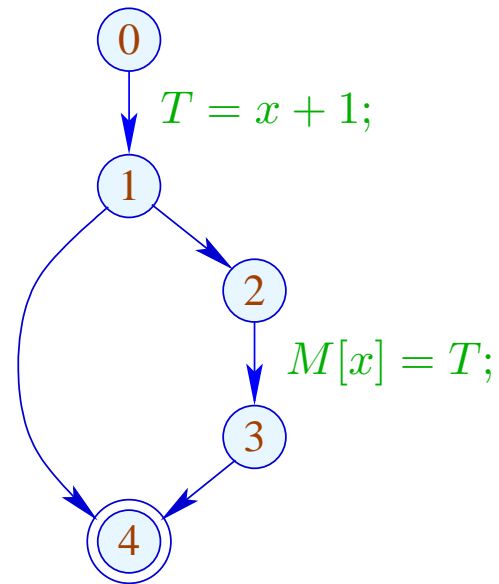
... but also **common ones** which cannot be rotated:



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

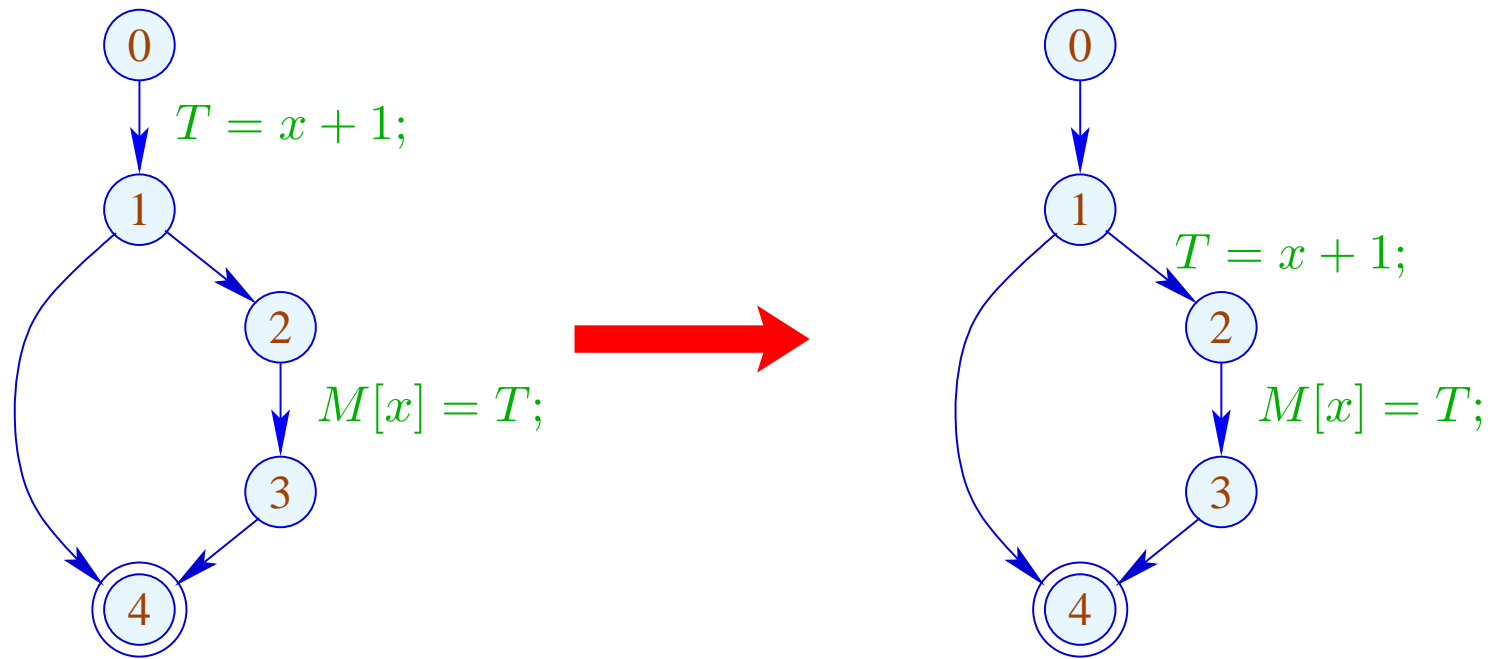
1.15 Eliminating Partially Dead Code

Example:



$x + 1$ needs 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



We define an analysis that maximally delays computations:

$$\begin{aligned} \llbracket ; \rrbracket^\# D &= D \\ \llbracket x = e; \rrbracket^\# D &= \begin{cases} D \setminus (Use_e \cup Def_x) \cup \{x = e;\} & \text{if } x \notin Vars_e \\ D \setminus (Use_e \cup Def_x) & \text{if } x \in Vars_e \end{cases} \end{aligned}$$

... where:

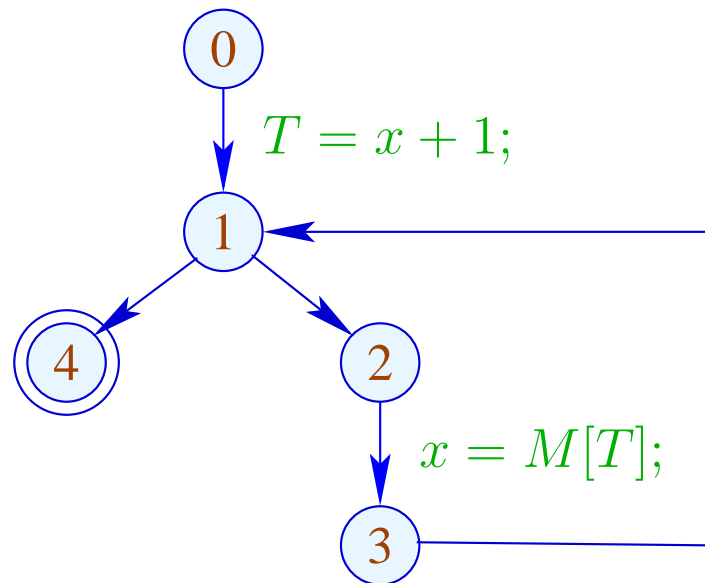
$$\begin{aligned} Use_e &= \{y = e'; \mid y \in Vars_e\} \\ Def_x &= \{y = e'; \mid y \equiv x \vee x \in Vars_{e'}\} \end{aligned}$$

For the remaining edges, we define:

$$\begin{aligned} \llbracket x = M[e]; \rrbracket^\# D &= D \setminus (Use_e \cup Def_x) \\ \llbracket M[e_1] = e_2; \rrbracket^\# D &= D \setminus (Use_{e_1} \cup Use_{e_2}) \\ \llbracket Pos(e) \rrbracket^\# D &= \llbracket Neg(e) \rrbracket^\# D = D \setminus Use_e \end{aligned}$$

Warning:

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

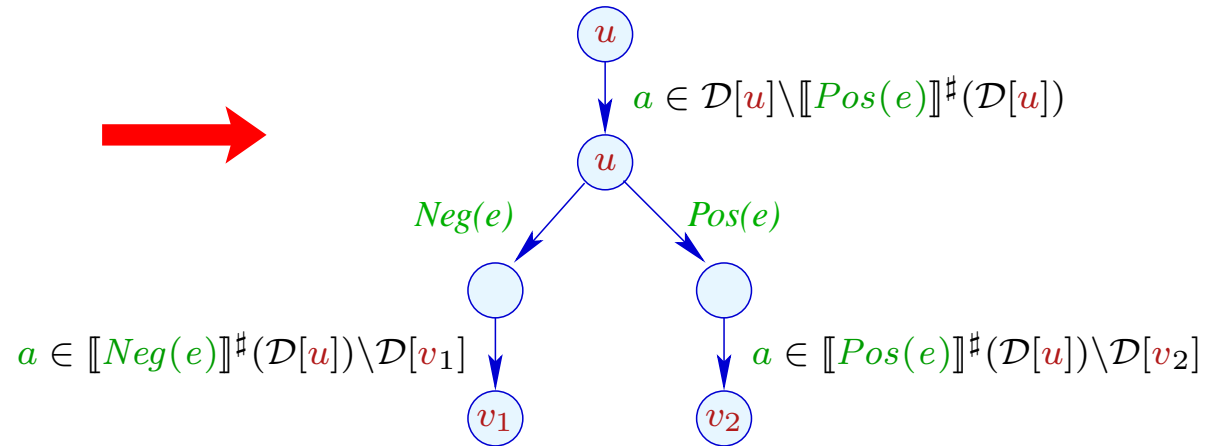
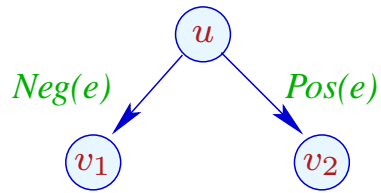
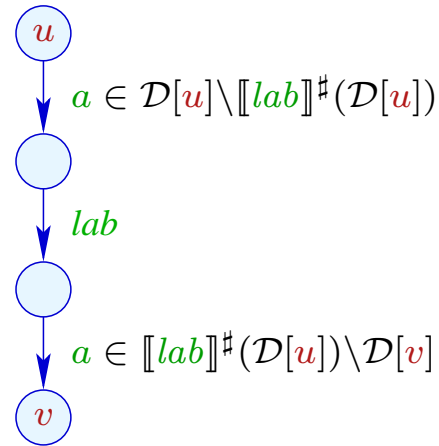
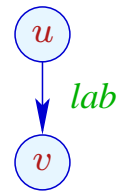


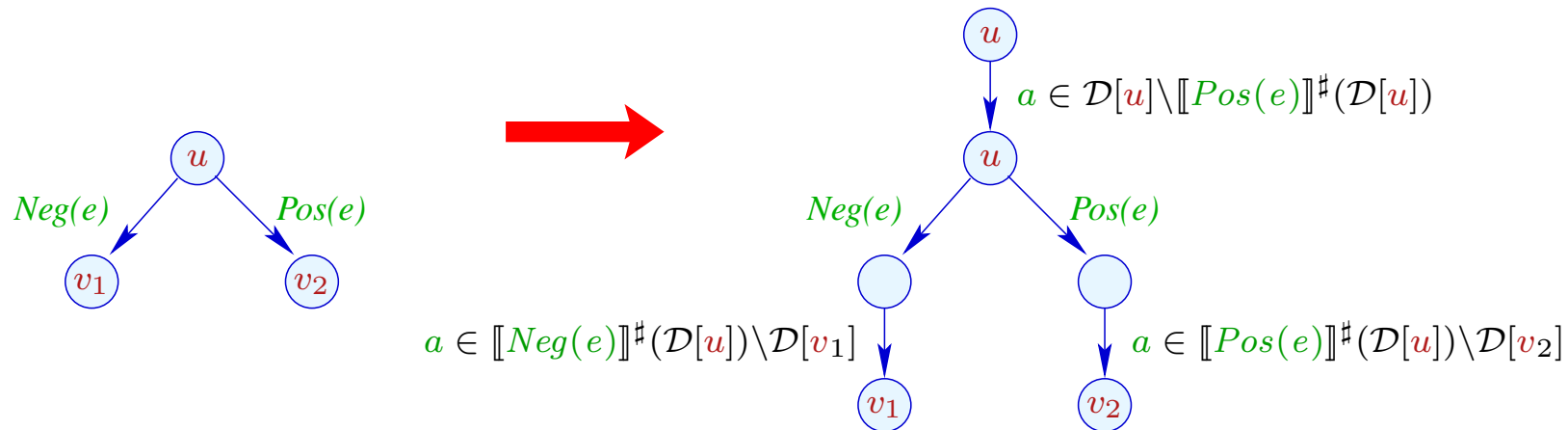
Here, $T = x + 1;$ cannot be moved beyond **1 !!!**

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 assignments delayable at u 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 PDE:

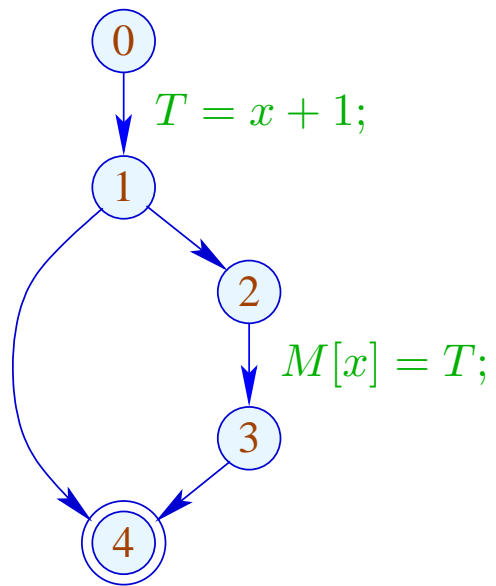




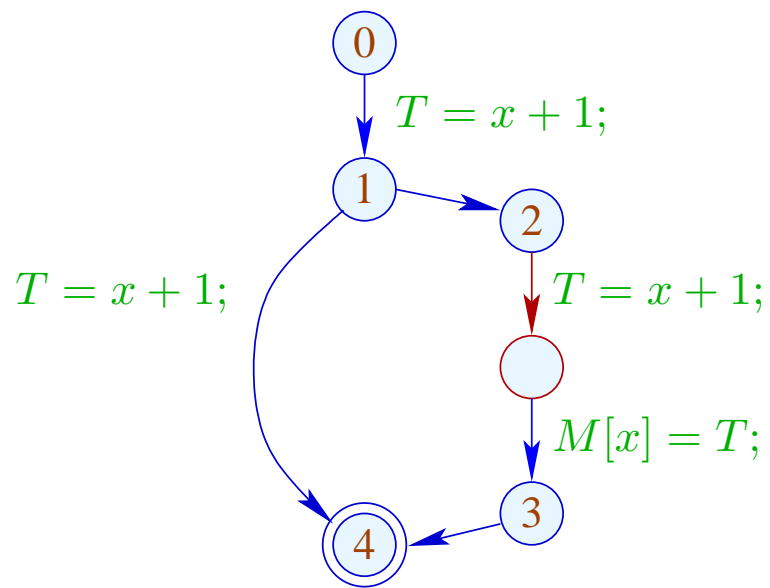
Note:

Transformation **PDE** is only meaningful, if we subsequently eliminate assignments to dead variables by means of transformation **DE**

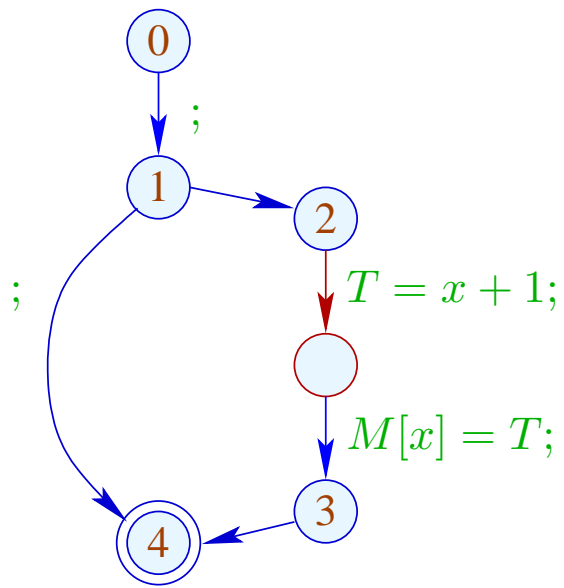
In the example, the partially dead code is eliminated:



	\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 *PDE*, all original assignments $y = e;$ with $y \notin Vars_e$ are assignments to dead variables and thus can always be eliminated
- By this, it can be proved that the transformation does not deteriorate the efficiency of the code.
- Similar to the elimination of partial redundancies, repeated application of the transformation may be profitable.

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 should be applied repeatedly.

... a meaningful ordering:

CP	Constant Propagation Interval Analysis Alias Analysis
LR	Loop Rotation
RE	Available Expressions
DE	Dead Variables
PDE	Partially Dead Code
PRE	Partially Redundant Code