→ Die Auswertung startet mit einer interessierenden Variable x_i (z.B. dem Wert für *stop*)

→ Es werden automatisch alle Variablen ausgewertet, die x_i beeinflussen :-)

→ Die Anzahl der Auswertungen ist i.a. kleiner als die bei normaler Iteration ;-)

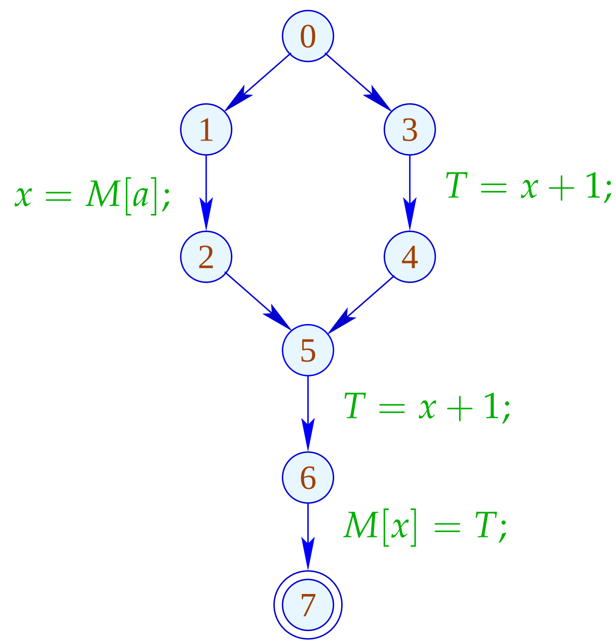
→ Der Algorithmus ist komplizierter, benötigt aber keine Vorberechnung der Variablen-Abhängigkeiten :-))

→ Er funktioniert auch, wenn die Variablen-Abhängigkeiten sich während der Iteration ändern !!!

$\Longrightarrow$ interprozedurale Analyse

1.7 Beseitigung partieller Redundanzen

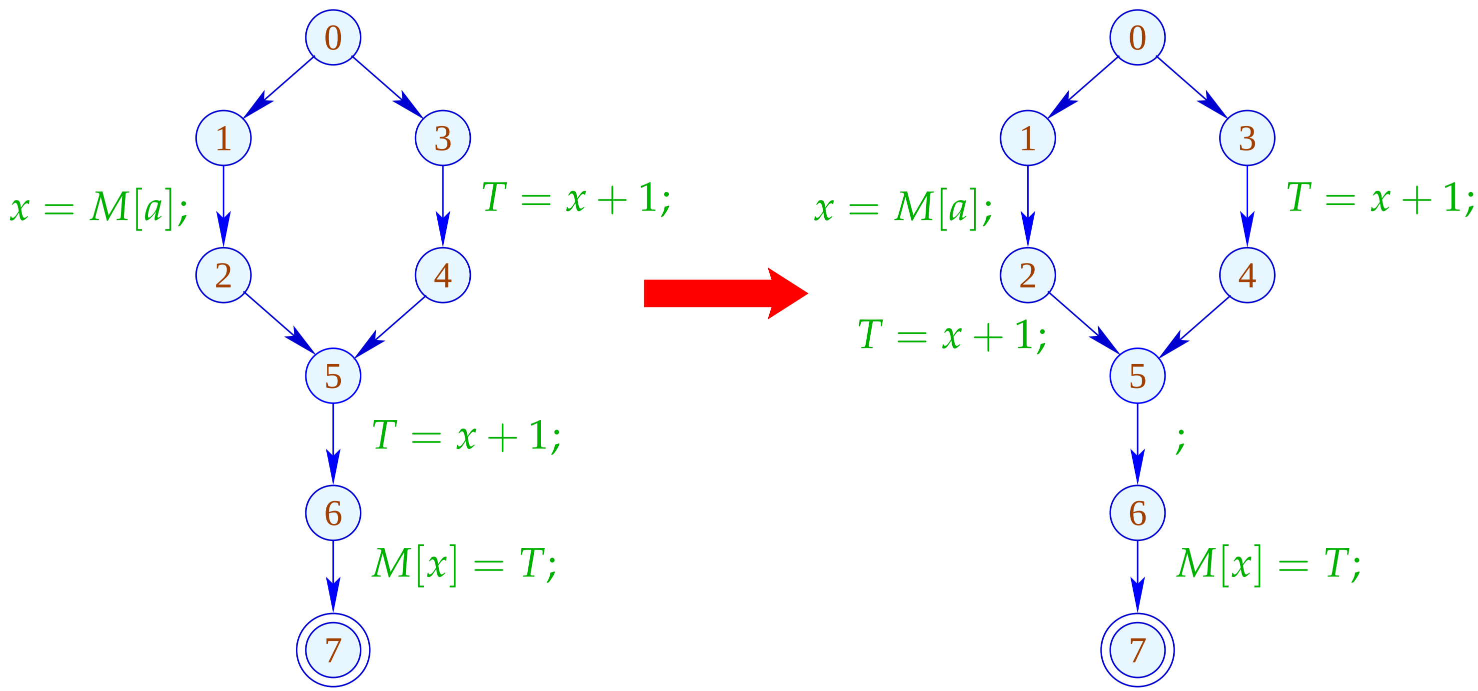
Beispiel:



// $\quad e = x + 1$ wird auf jedem Pfad ausgewertet ...

// leider auf einem Pfad sogar zweimal :-(

Ziel:



Idee:

(1) Füge so Zuweisungen $T_e = e;$ ein, dass e an allen Stellen verfügbar ist, an denen der Wert von e benötigt wird.

(2) Spare dabei die Stellen, an denen e entweder bereits verfügbar ist oder in der Zukunft sicher benötigt wird.

Ausdrücke mit der letzteren Eigenschaft nennt man sehr beschäftigt.

(3) Ersetze in die ursprünglichen Berechnungen von e durch Zugriffe auf die Variable T_e.

$\Longrightarrow$ wir benötigen eine neue Analyse :-))

Ein Ausdruck e heißt beschäftigt (busy) entlang eines Pfads π, falls der Wert von e berechnet wird, bevor eine der Variablen $x \in Vars(e)$ überschrieben wird.

$//$ Rückwärtsanalyse!

e heißt sehr beschäftigt (very busy) an u, falls e beschäftigt ist entlang jedes Pfads $\pi : u \rightarrow^* stop$.

Ein Ausdruck e heißt beschäftigt (busy) entlang eines Pfads π, falls der Wert von e berechnet wird, bevor eine der Variablen $x \in Vars(e)$ überschrieben wird.

$//$ Rückwärtsanalyse!

e heißt sehr beschäftigt (very busy) an u, falls e beschäftigt ist entlang jedes Pfads $\pi : u \to^* stop$.

Entsprechend benötigen wir:

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

wobei für $\pi = k_1 \ldots k_m$:

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

Unser vollständiger Verband ist:

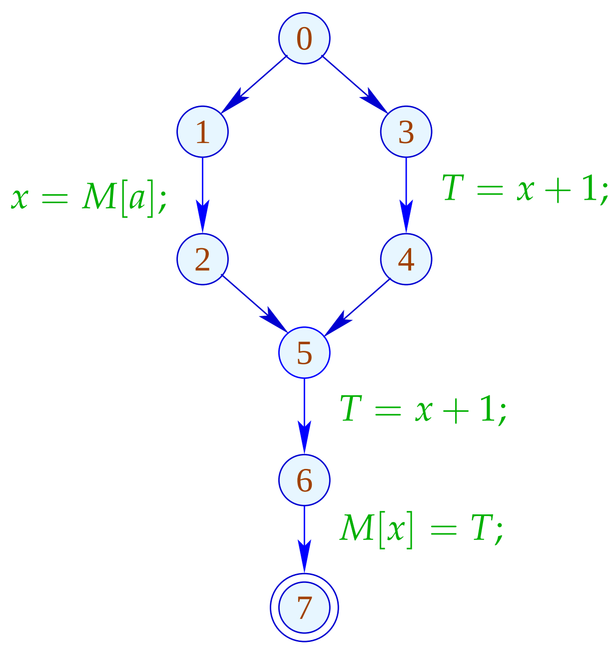
$$\mathbb{B} = 2^{Expr \setminus Vars} \qquad \text{mit} \quad \sqsubseteq \; = \; \supseteq$$

Der Effekt $[\![k]\!]^\sharp$ einer Kante $k = (u, lab, v)$ hängt nur von lab ab, d.h. $[\![k]\!]^\sharp = [\![lab]\!]^\sharp$ wobei:

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

Die Kanten-Effekte sind sämtlich distributiv. Deshalb liefert die kleinste Lösung des Ungleichungssystems exakt den MOP :-)

Beispiel:



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

Ein u heißt sicher für e, sofern $e \in \mathcal{A}[u] \cup \mathcal{B}[u]$; d.h. e ist entweder verfügbar oder sehr beschäftigt.

Ist u sicher, können wir dort e gefahrlos berechnen :-)

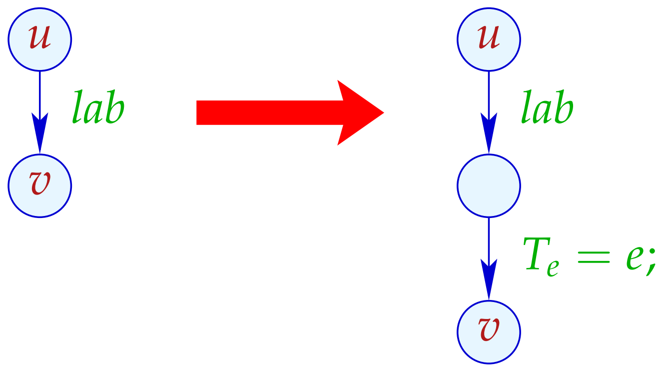
Idee:

- Wir berechnen e zum frühesten sicheren Zeitpunkt :-)
- Wir platzieren $T_e = e;$ hinter einer Kante (u, lab, v) mit

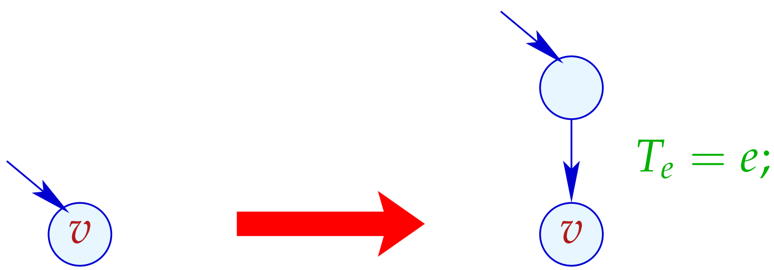
$$e \in \mathcal{B}[v] \backslash [\![lab]\!]^\sharp_{\mathcal{A}}(\mathcal{A}[u] \cup \mathcal{B}[u])$$

- Folge: e ist nun an u verfügbar, wannimmer $e \in \mathcal{A}[u] \cup \mathcal{B}[u]$:-).

Transformation 5.1:

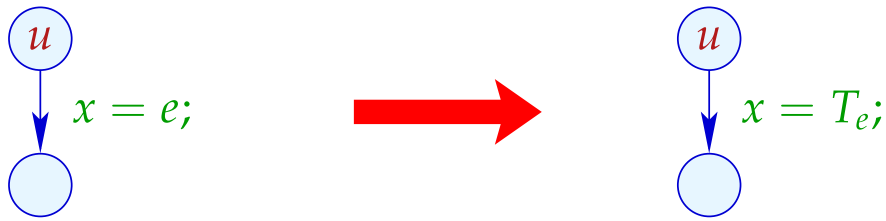


$$(e \in \mathcal{B}[v] \backslash [\![lab]\!]^\sharp_{\mathcal{A}} (\mathcal{A}[u] \cup \mathcal{B}[u]))$$



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

Transformation 5.2:



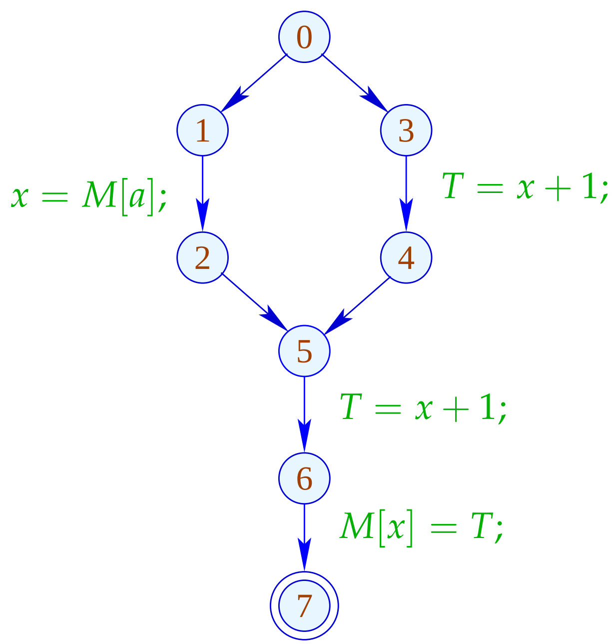
$//$ analog für die anderen Benutzungen von e

$//$ an alten Kanten des Programms.

Bernhard Steffen, Dortmund

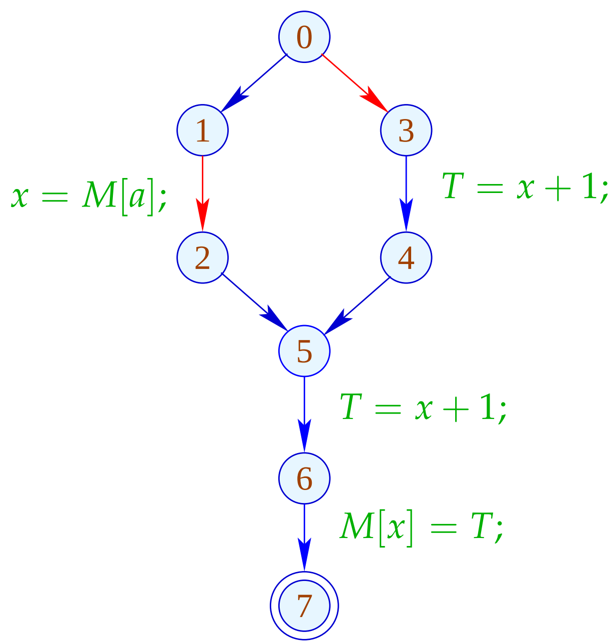
Jens Knoop, Wien

Im Beispiel:



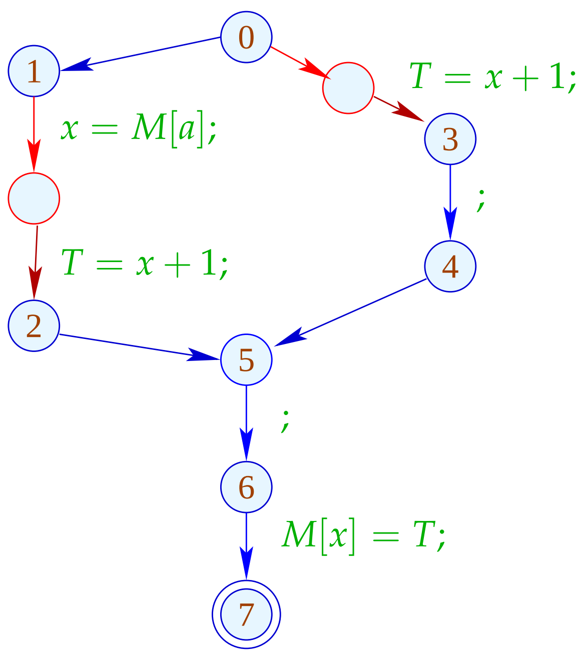
	$\mathcal{A}$	$\mathcal{B}$
0	$\emptyset$	$\emptyset$
1	$\emptyset$	$\emptyset$
2	$\emptyset$	$\{x+1\}$
3	$\emptyset$	$\{x+1\}$
4	$\{x+1\}$	$\{x+1\}$
5	$\emptyset$	$\{x+1\}$
6	$\{x+1\}$	$\emptyset$
7	$\{x+1\}$	$\emptyset$

Im Beispiel:



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



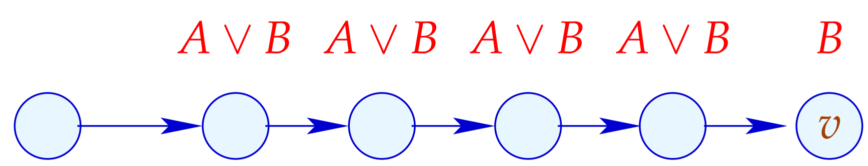
	$\mathcal{A}$	$\mathcal{B}$
0	$\emptyset$	$\emptyset$
1	$\emptyset$	$\emptyset$
2	$\emptyset$	$\{x+1\}$
3	$\emptyset$	$\{x+1\}$
4	$\{x+1\}$	$\{x+1\}$
5	$\emptyset$	$\{x+1\}$
6	$\{x+1\}$	$\emptyset$
7	$\{x+1\}$	$\emptyset$

Zur Korrektheit:

Sei π ein Pfad, der nach v führt, hinter dem eine Benutzung von e erfolgt.

Dann gibt es ein maximales Suffix von π so dass für jede Kante $k = (u, lab, u')$ in dem Sufix gilt:

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



Zur Korrektheit:

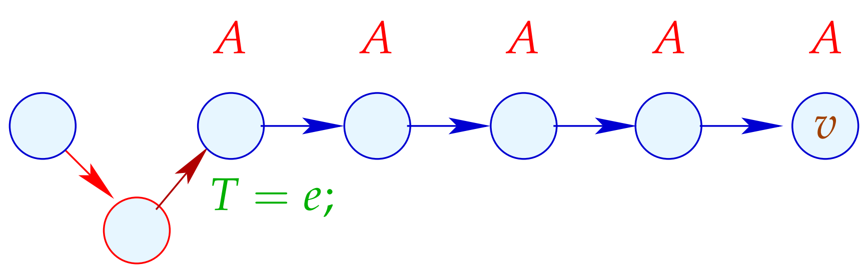
Sei π ein Pfad, der nach v führt, hinter dem eine Benutzung von e erfolgt.

Dann gibt es ein maximales Suffix von π so dass für jede Kante $k = (u, lab, u')$ in dem Sufix gilt:

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

Insbesondere kann dann keine Variable aus e entlang einer solchen Kante einen neuen Wert erhalten :-)

Und wir fügen $T_e = e;$ davor ein :-))



Wir schließen:

- Überall, wo der Wert von e benötigt wird, ist e verfügbar :-)

  $\Longrightarrow$ Korrektheit der Transformation

- Jedem $T = e;$, das wir in einen Pfad einfügen, entspricht ein $T = e;$, das wir gestrichen haben :-))

  $\Longrightarrow$ Nicht-Verschlechterung der Transformation

1.8 Anwendung: Schleifen-invarianter Code

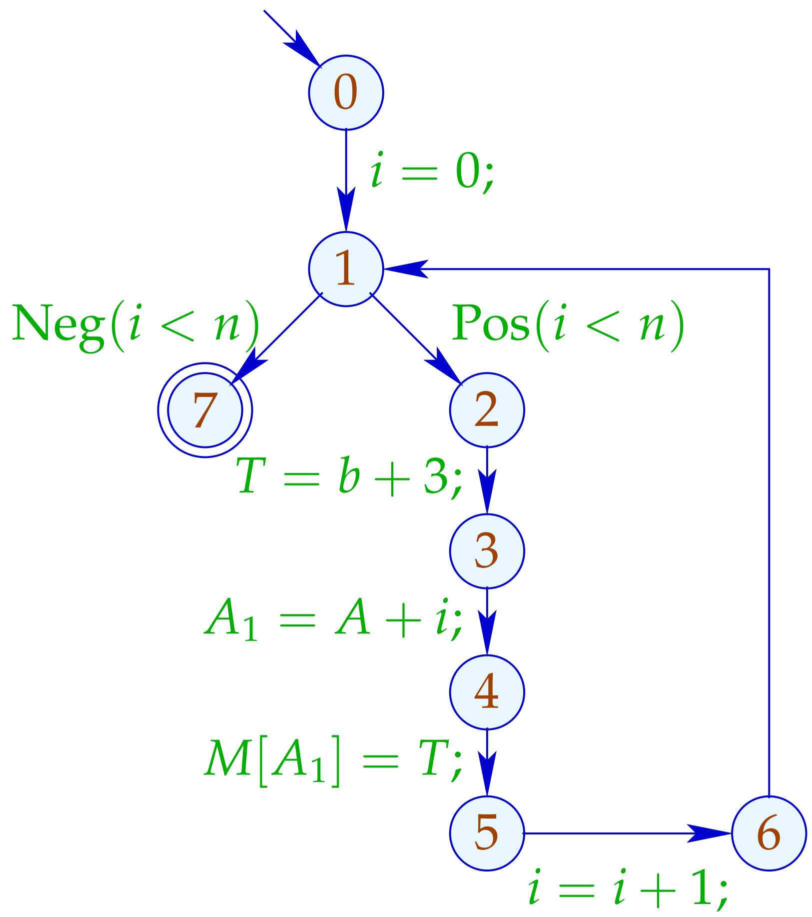
Beispiel:

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

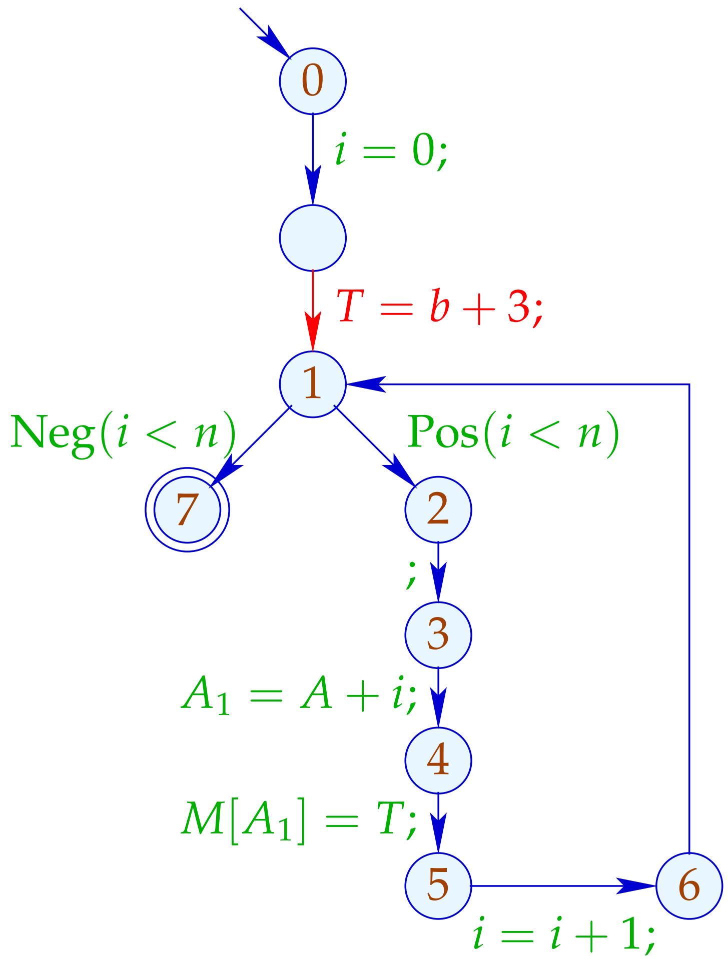
// Der Ausdruck $b + 3$ wird in jeder Iteration berechnet :-(

// Das wollen wir vermeiden :-)

Der Kontrollfluss-Graph:

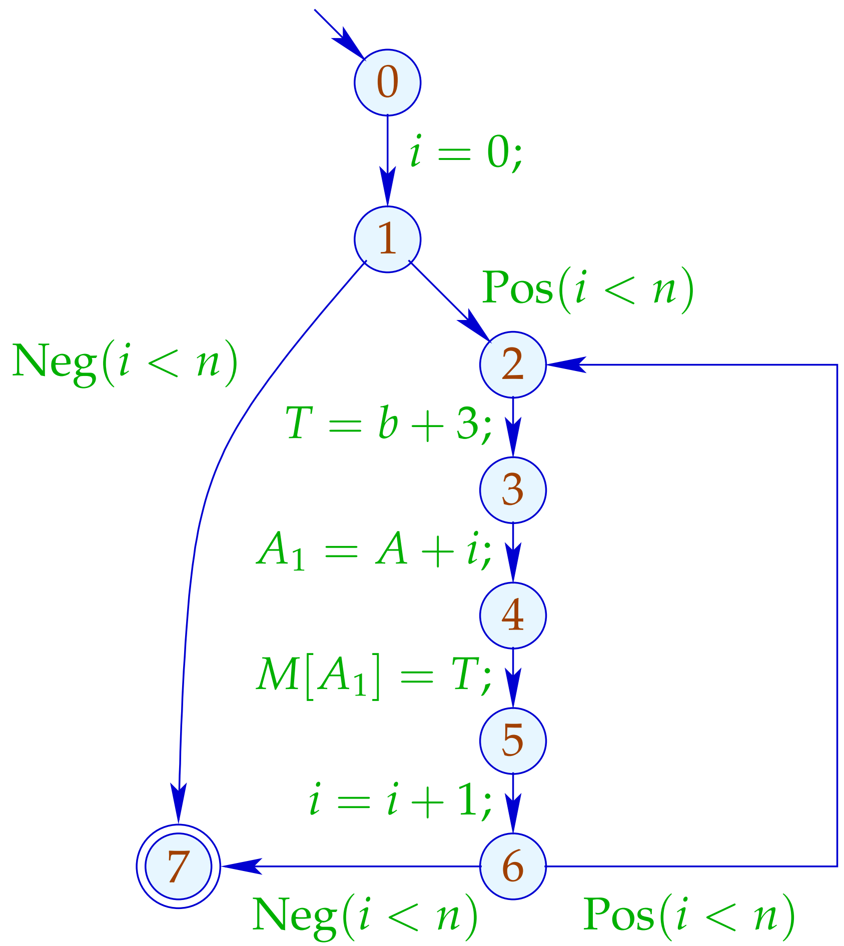


Achtung: $T = b + 3;$ darf nicht vor der Schleife stehen :

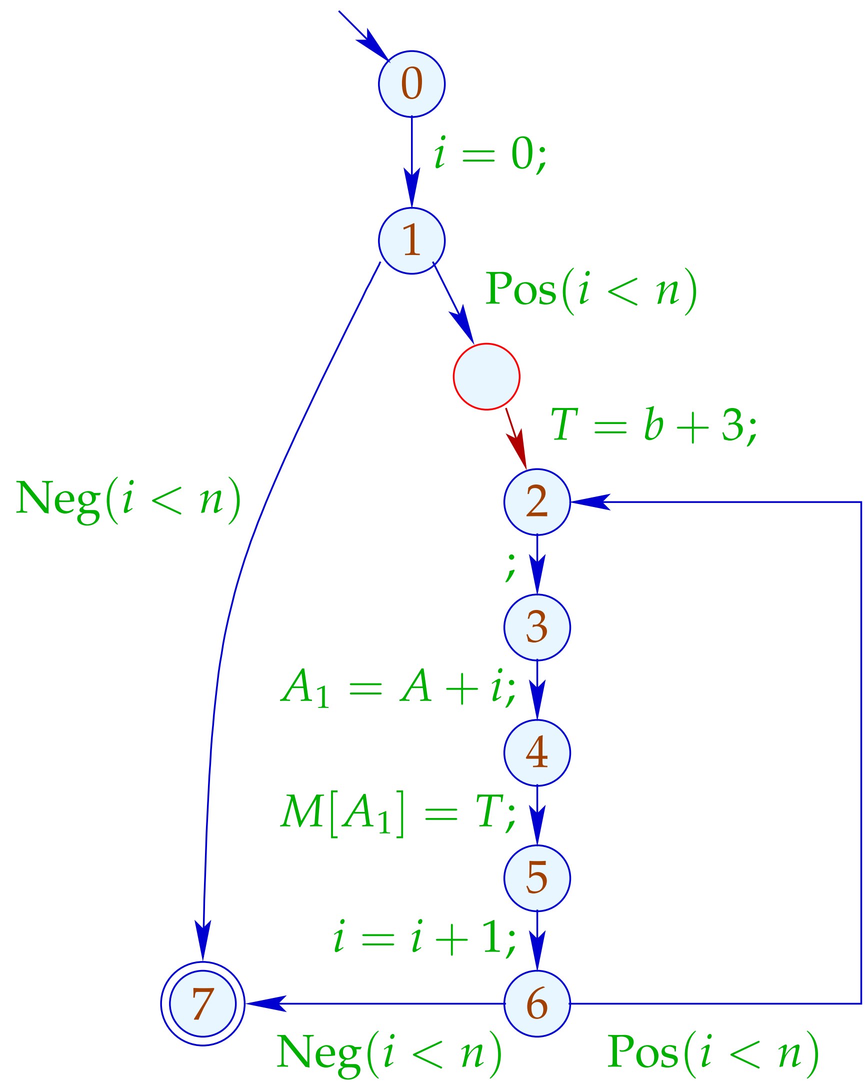


$\Longrightarrow$ Es gibt keinen guten Platz für $T = b + 3;$:-(

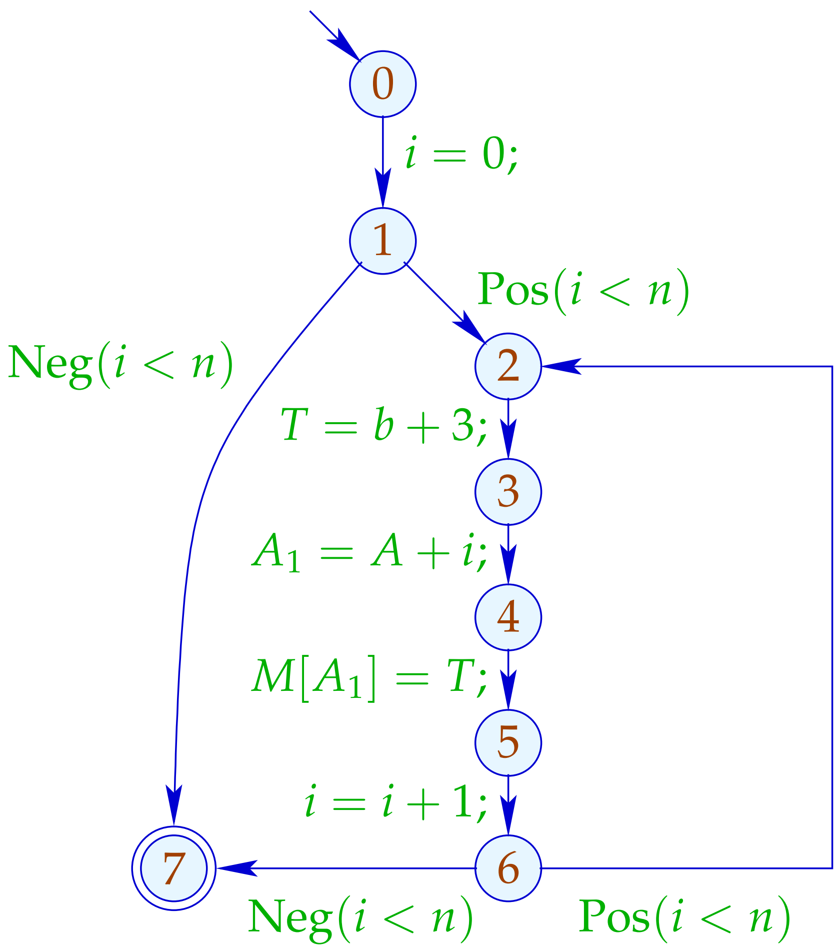
Idee: Transformiere in eine do-while-Schleife ...



... jetzt gibt es eine Stelle für $T = e;$:-)

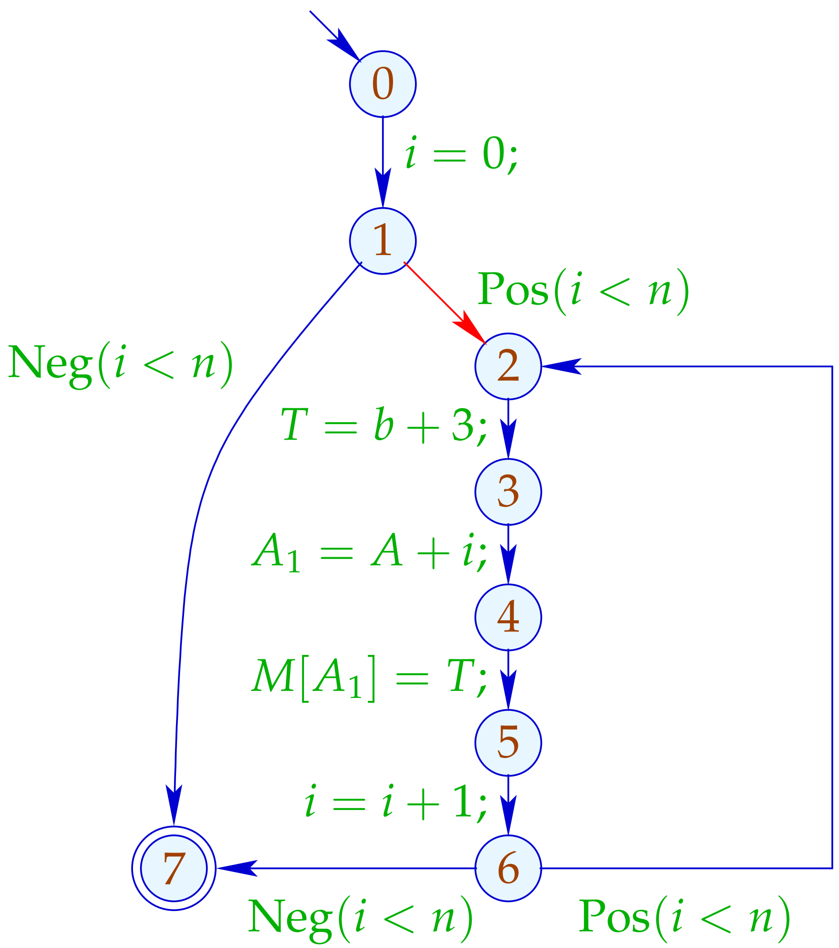


Anwendung von T5 (PRE) :



	$\mathcal{A}$	$\mathcal{B}$
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	$\emptyset$	$\emptyset$

Anwendung von 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	$\emptyset$	$\emptyset$

Fazit:

- Beseitigung partieller Redundanzen kann loop-invarianten Code aus Schleifen heraus schieben :-))
- Das funktioniert nur für do-while-Scheifen :-(
- Um andere Scheifen zu optimieren, wandeln wir sie in do-while-Scheifen um:

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

$\Longrightarrow$ Schleifen-Rotation

Problem:

Haben wir das Quell-Programm nicht (mehr) zur Verfügung, müssen wir nachträglich die Schleifen (-köpfe) identifizieren ;-)

$$\Longrightarrow \qquad \text{Prädominatoren}$$

u prädominiert v, falls jeder Pfad $\pi : \mathit{start} \rightarrow^* v$ Knoten u enthält. Wir schreiben: $u \Rightarrow v$.

"$\Rightarrow$" ist reflexiv, transitiv und anti-symmetrisch :-)

Berechnung:

Wir sammeln die Knoten entlang Pfaden auf mithilfe der Analyse:

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

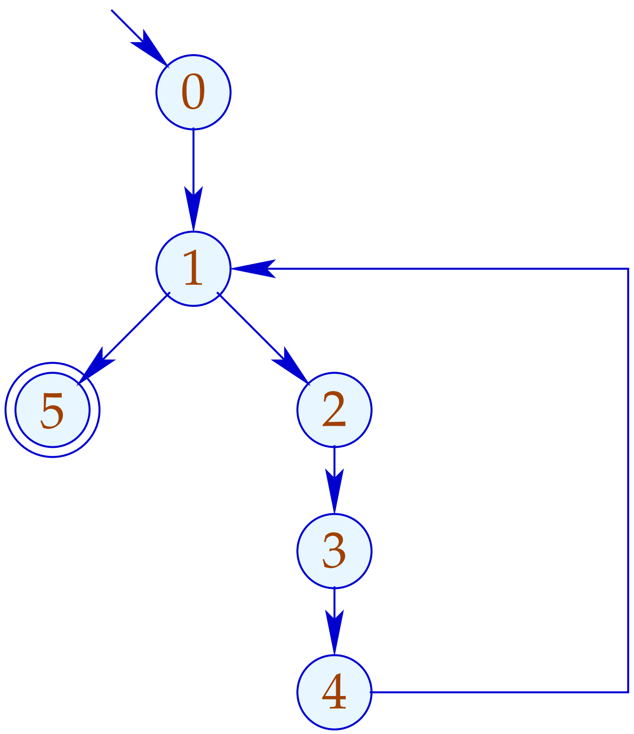
$$[\![(\_,\_,v)]\!]^\sharp\, P \;=\; P \cup \{v\}$$

Dann ist die Menge $\mathcal{P}[v]$ der Prädominatoren:

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

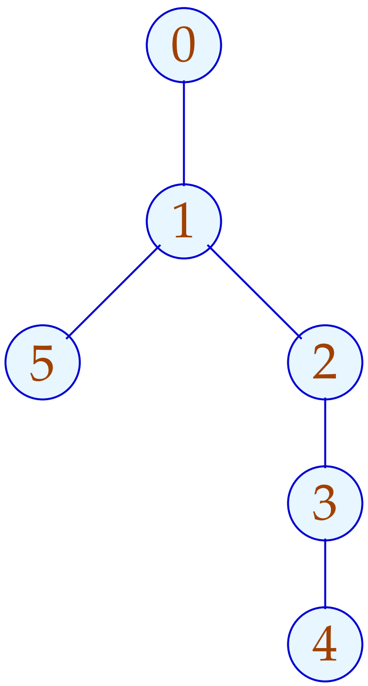
Da die $[\![k]\!]^\sharp$ distributiv sind, können wir die $\mathcal{P}[v]$ mithilfe von Fixpunkt-Iteration berechnen :-)

Beispiel:



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

Die partielle Ordnung “$\Rightarrow$” im Beispiel:



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

Offenbar ist das Ergebnis ein Baum :-)

Tatsächlich gilt:

Satz:

Jeder Punkt v hat maximal einen unmittelbaren Prädominator.

Beweis:

Annahme:

Es gäbe $u_1 \neq u_2$, die v unmittelbar prädominieren.

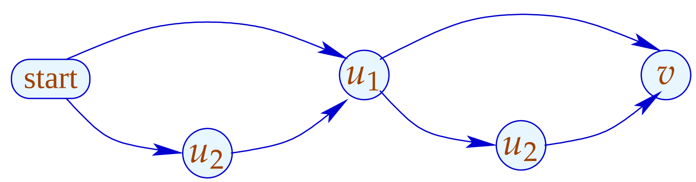
Gälte $u_1 \Rightarrow u_2$, wäre u_1 nicht unmittelbar.

Folglich müssen u_1, u_2 unvergleichbar sein :-)

Nun gilt für jedes $\pi : start \to^* v$:

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

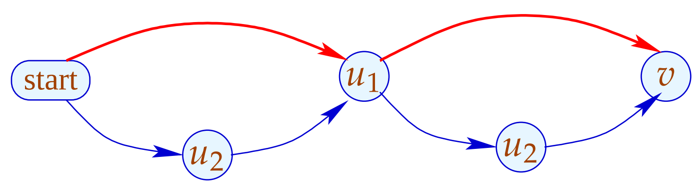
Sind u_1, u_2 aber unvergleichbar, gibt es einen Pfad: $start \to^* v$ ohne u_2 :



Nun gilt für jedes $\pi : start \longrightarrow^* v$:

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

Sind u_1, u_2 aber unvergleichbar, gibt es einen Pfad: $start \longrightarrow^* v$ ohne u_2 :



Beobachtung:

Der Schleifenkopf einer while-Schleife dominiert jeden Knoten des Rumpfs.

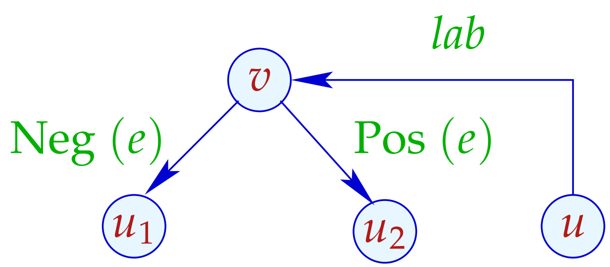
Einen Rücksprung vom Ende u zum Schleifenkopf v erkennt man daran, dass

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

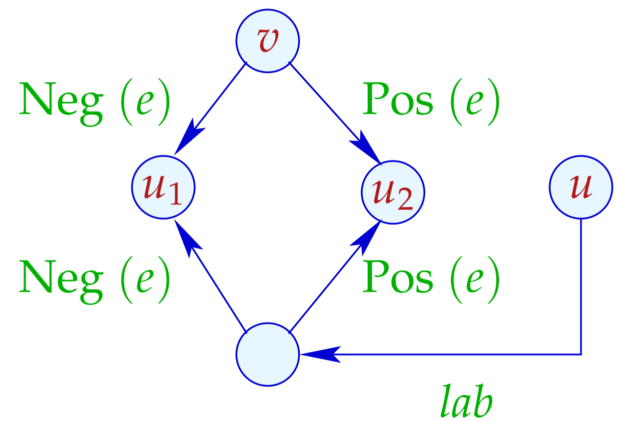
:-)

Damit definieren wir:

Transformation 6:

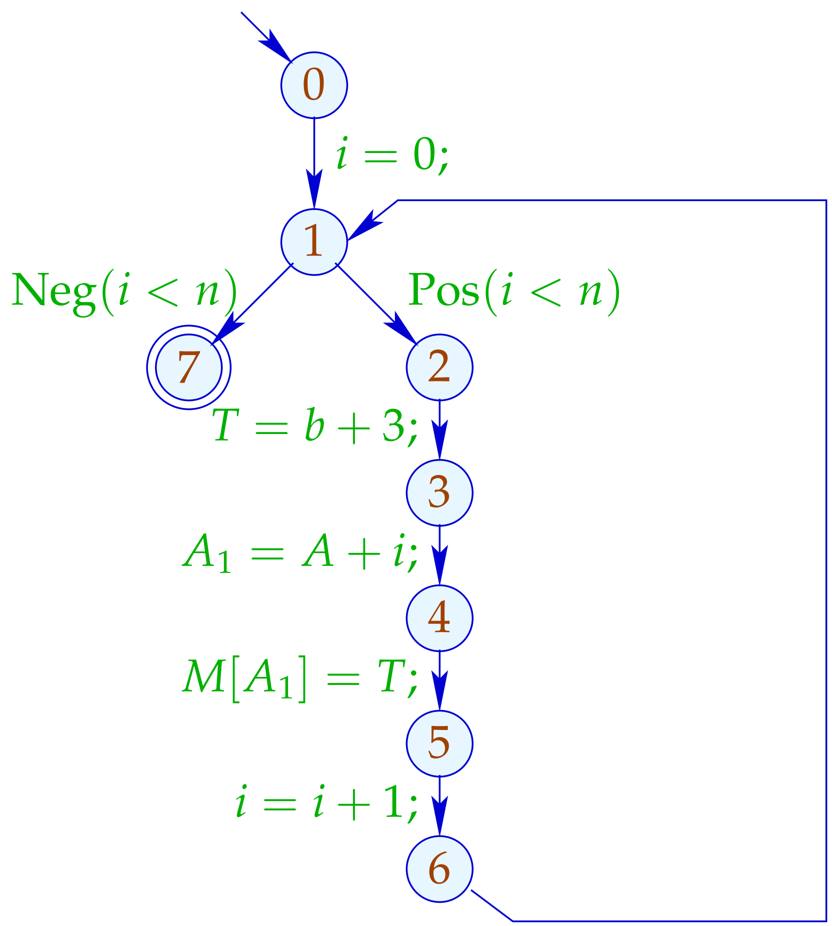


$$u_1 \notin \mathcal{P}[u]$$
$$u_2, v \in \mathcal{P}[u]$$

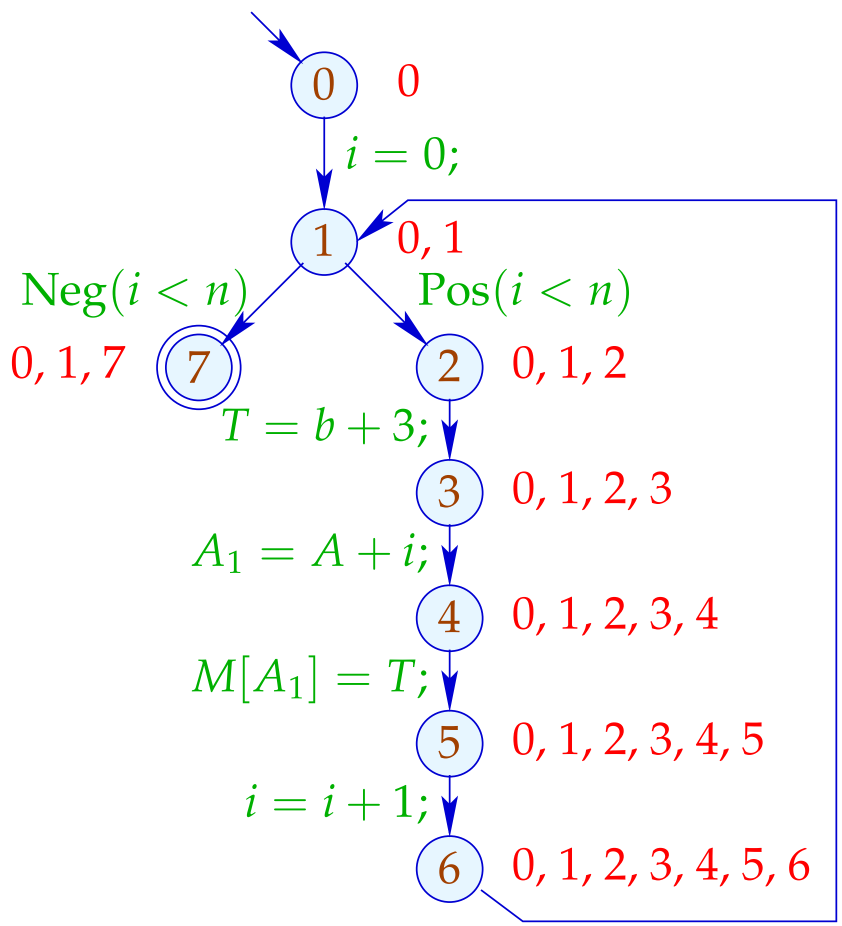


Wir duplizieren den Eintritts-Test an alle Rücksprung-Stellen :-)

... im Beispiel:

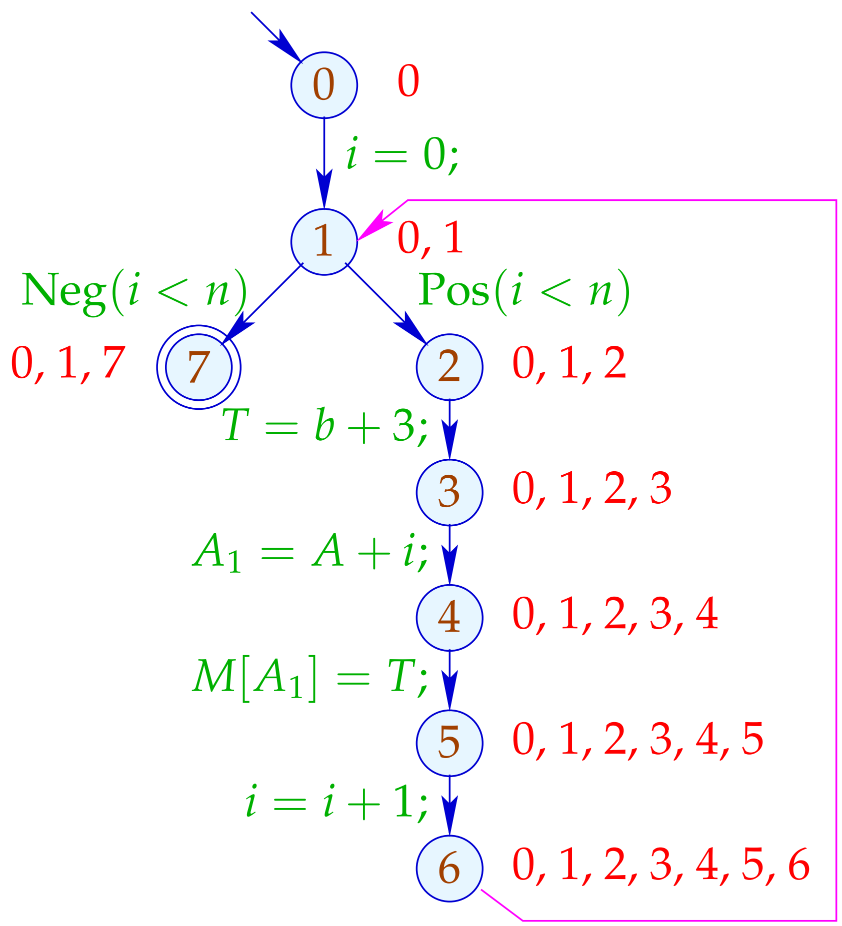


... im Beispiel:



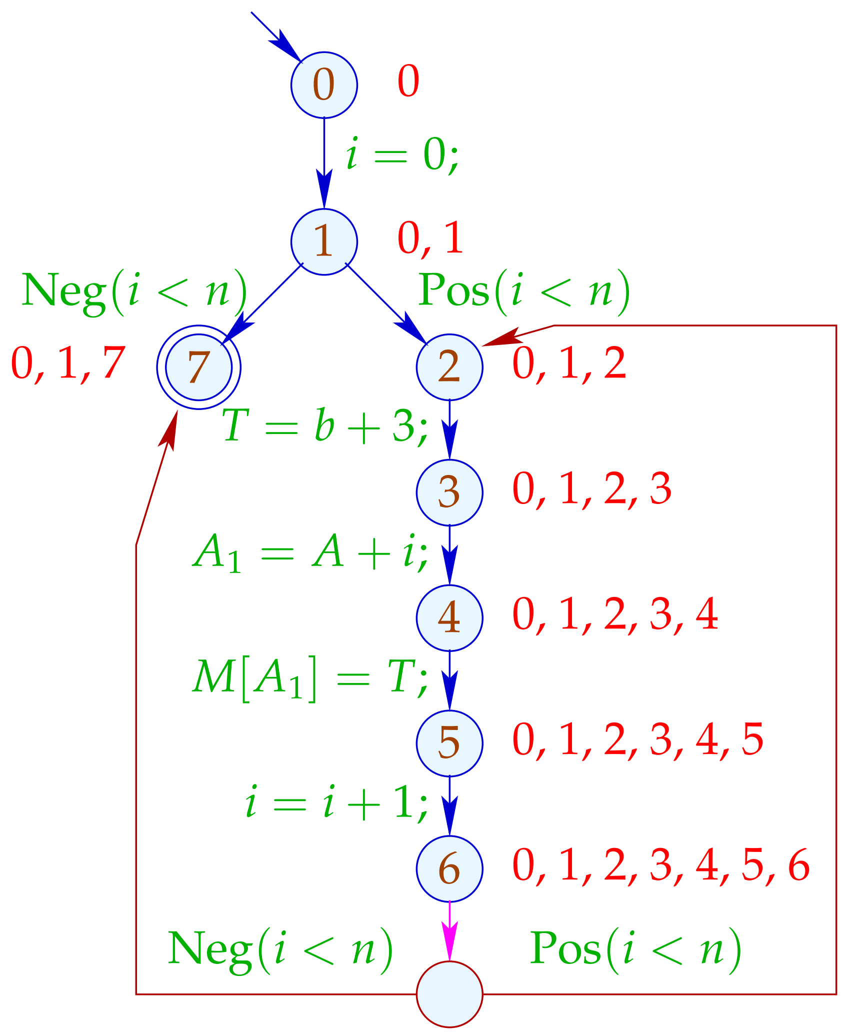
0
0
$i = 0;$
1
0, 1
Neg$(i < n)$
Pos$(i < n)$
7
0, 1, 7
2
0, 1, 2
$T = b + 3;$
3
0, 1, 2, 3
$A_1 = A + i;$
4
0, 1, 2, 3, 4
$M[A_1] = T;$
5
0, 1, 2, 3, 4, 5
$i = i + 1;$
6
0, 1, 2, 3, 4, 5, 6

... im Beispiel:



0
$i = 0;$
1
0, 1
$\text{Neg}(i < n)$
$\text{Pos}(i < n)$
0, 1, 7
7
2
0, 1, 2
$T = b + 3;$
3
0, 1, 2, 3
$A_1 = A + i;$
4
0, 1, 2, 3, 4
$M[A_1] = T;$
5
0, 1, 2, 3, 4, 5
$i = i + 1;$
6
0, 1, 2, 3, 4, 5, 6

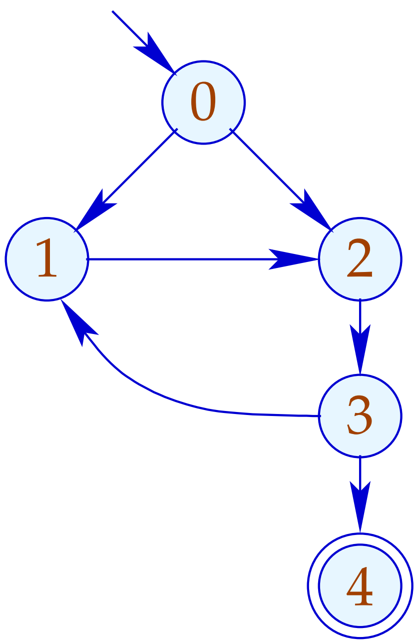
... im Beispiel:



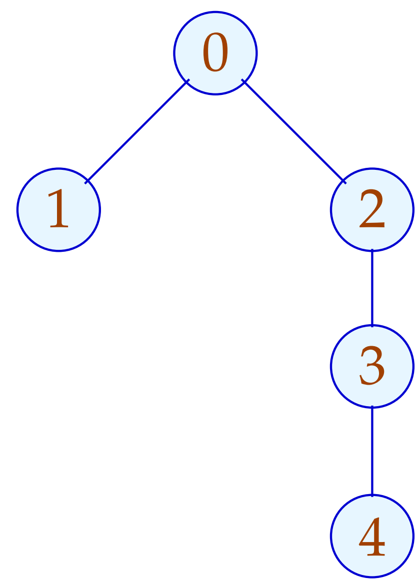
0
$i = 0;$
1
0, 1
Neg$(i < n)$
Pos$(i < n)$
7
0, 1, 7
2
0, 1, 2
$T = b + 3;$
3
0, 1, 2, 3
$A_1 = A + i;$
4
0, 1, 2, 3, 4
$M[A_1] = T;$
5
0, 1, 2, 3, 4, 5
$i = i + 1;$
6
0, 1, 2, 3, 4, 5, 6
Neg$(i < n)$
Pos$(i < n)$

Achtung:

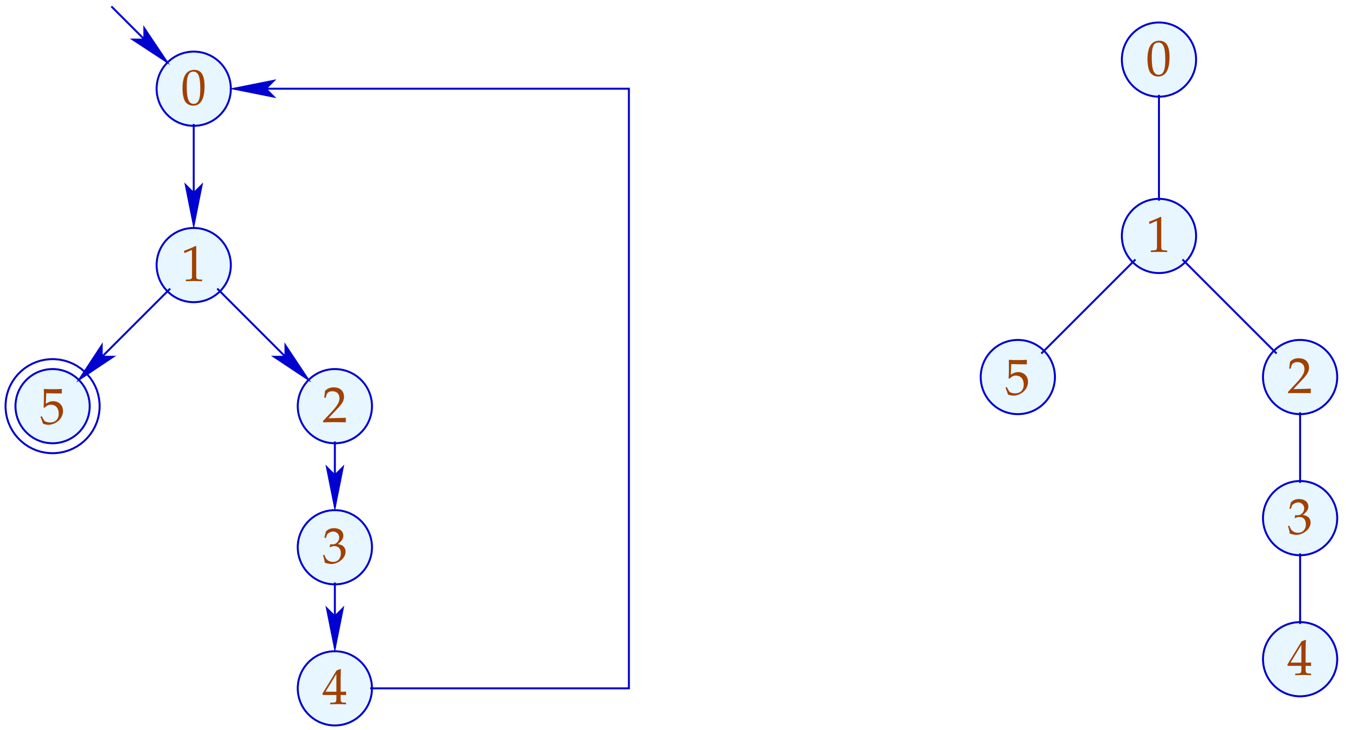
Es gibt ungewöhnliche Schleifen, die so nicht rotiert werden:



Prädominatoren:

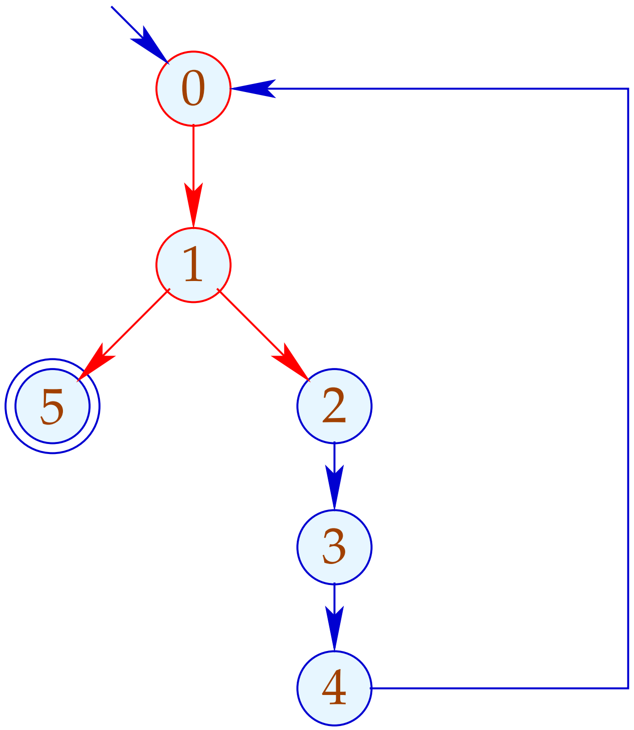


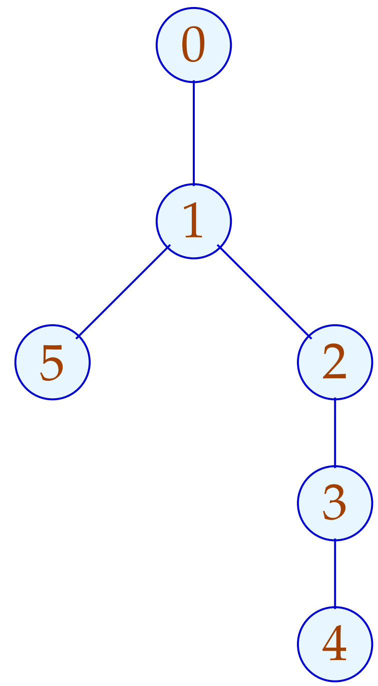
... leider aber auch gewöhnliche, die nicht rotiert werden:



Hier müsste man den ganzen Pfad zwischen Rücksprung und Bedingung duplizieren :-(

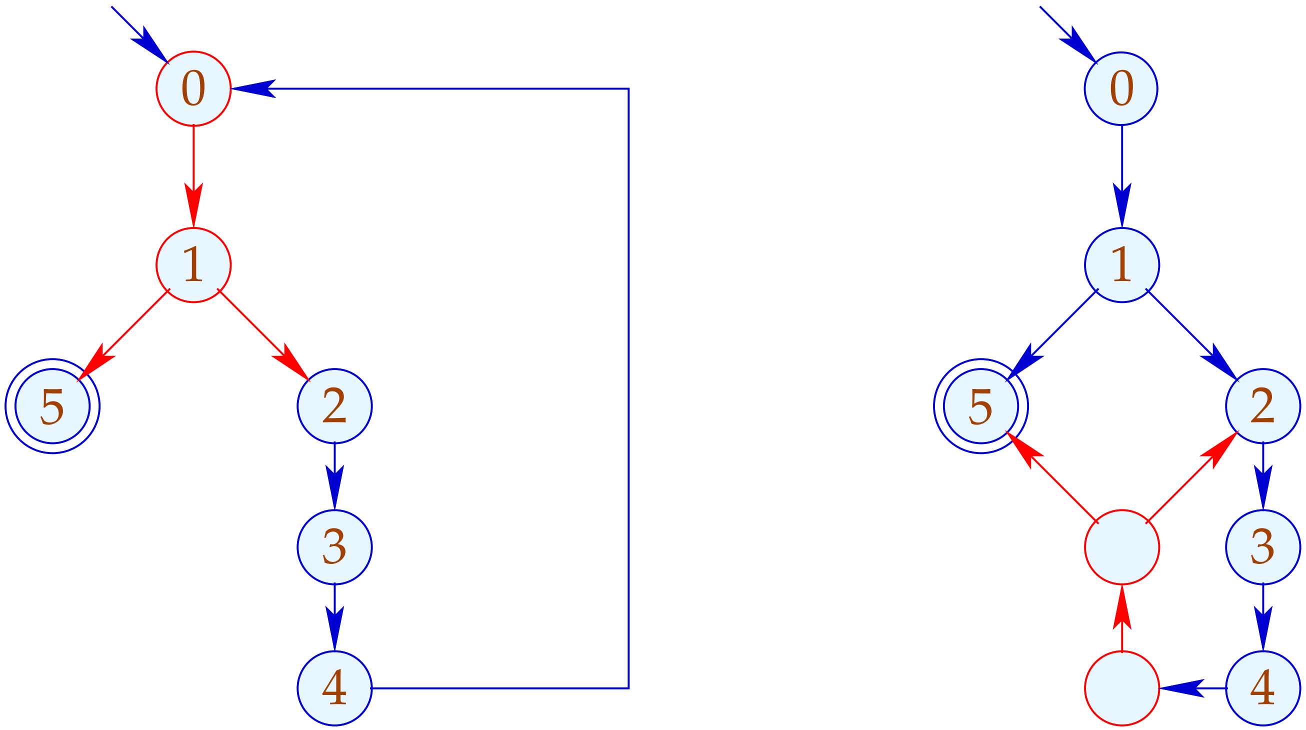
... leider aber auch gewöhnliche, die nicht rotiert werden:





Hier müsste man den ganzen Pfad zwischen Rücksprung und Bedingung duplizieren :-(

... leider aber auch gewöhnliche, die nicht rotiert werden:



Hier müsste man den ganzen Pfad zwischen Rücksprung und Bedingung duplizieren :-)