3 Ausnutzung von Hardware-Einrichtungen

Frage: Wie nutzt man optimal

... Register

... Instruktionen

... Pipelines

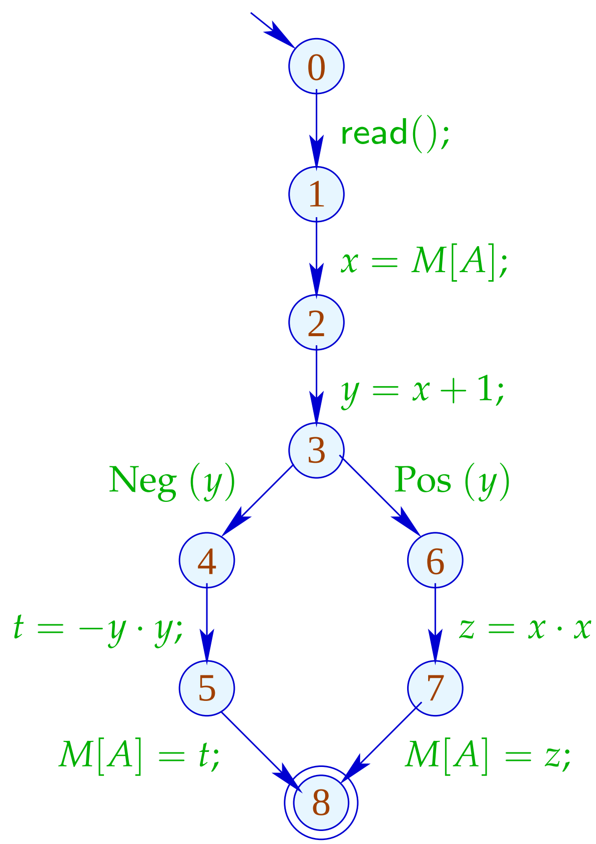
... Caches

... Prozessoren ???

3.1 Register

Beispiel:

$$
\begin{array}{l}
\mathsf{read}(); \\
x = M[A]; \\
y = x + 1; \\
\mathsf{if}\ \ (y)\ \ \{ \\
\qquad z = x \cdot x; \\
\qquad M[A] = z; \\
\}\ \ \mathsf{else}\ \ \{ \\
\qquad t = -y \cdot y; \\
\qquad M[A] = t; \\
\}
\end{array}
$$



Das Programm benötigt 5 Variablen ...

Problem:

Was tun, wenn das Programm benutzt mehr Variablen als Register da sind :-(

Idee:

Benutze ein Register für mehrere Variablen :-)

Im Beispiel etwa eines für x, t, z ...

$\mathsf{read}();$

$x = M[A];$

$y = x + 1;$

$\mathsf{if}\ \ (y)\ \ \{$

$\qquad z = x \cdot x;$

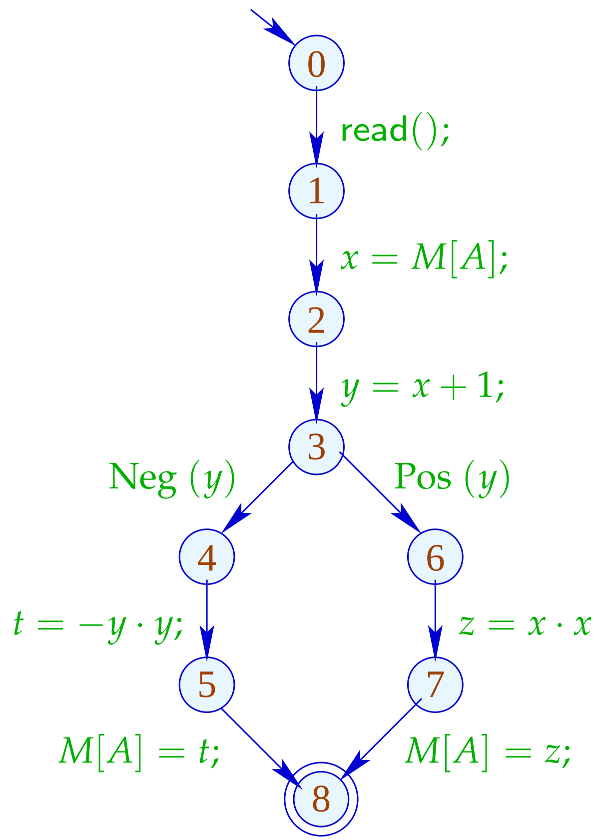
$\qquad M[A] = z;$

$\}\ \ \mathsf{else}\ \ \{$

$\qquad t = -y \cdot y;$

$\qquad M[A] = t;$

$\}$



read$()$;

$R = M[A]$;

$y = R + 1$;

if (y) $\{$

$\quad R = R \cdot R$;

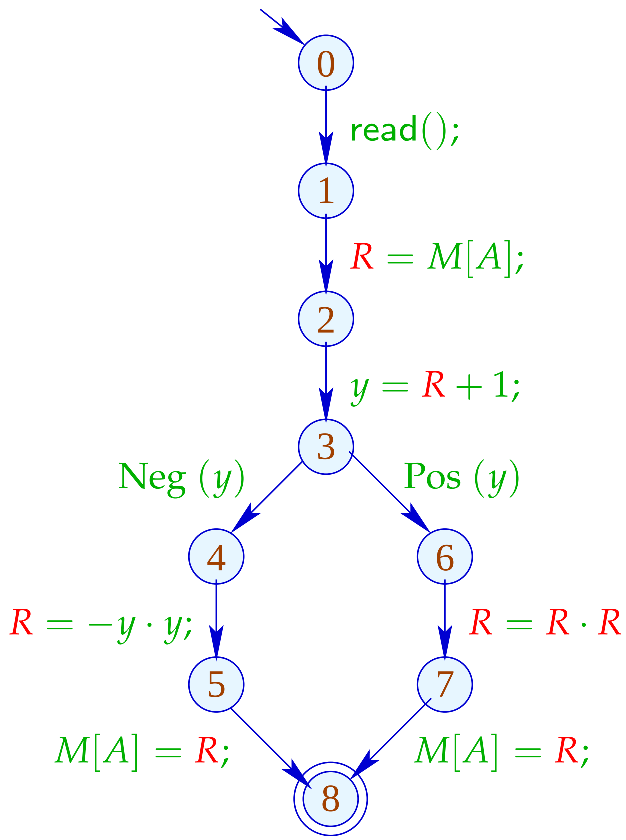
$\quad M[A] = R$;

$\}$ else $\{$

$\quad R = -y \cdot y$;

$\quad M[A] = R$;

$\}$



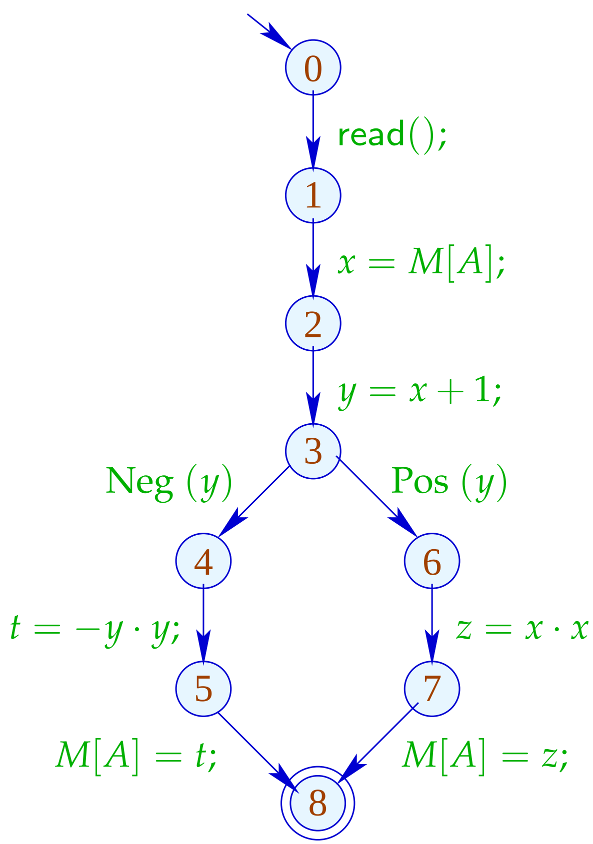
Achtung:

Das geht nur, wenn sich die Lebendigkeitsbereiche nicht überschneiden :-)

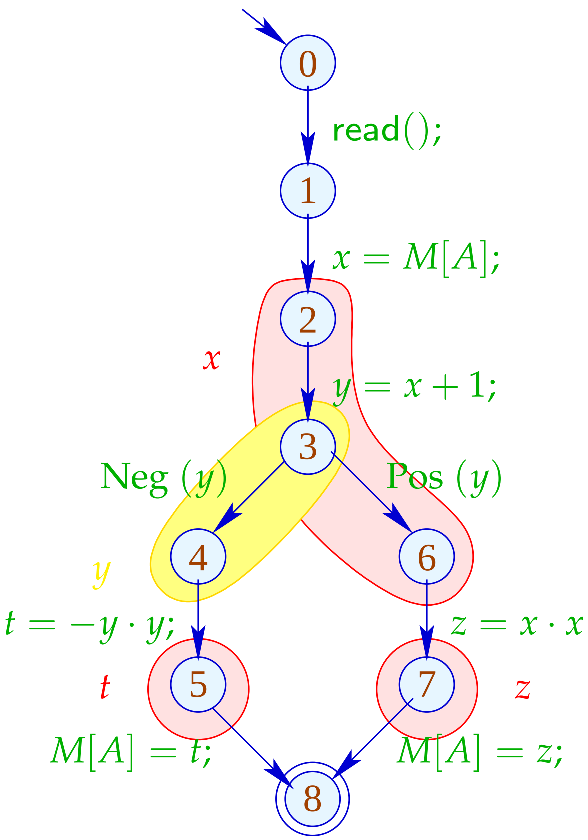
Der (wahre) Lebendigkeitsbereich von x ist:

$$\mathcal{L}[x] = \{u \mid x \in \mathcal{L}[u]\}$$

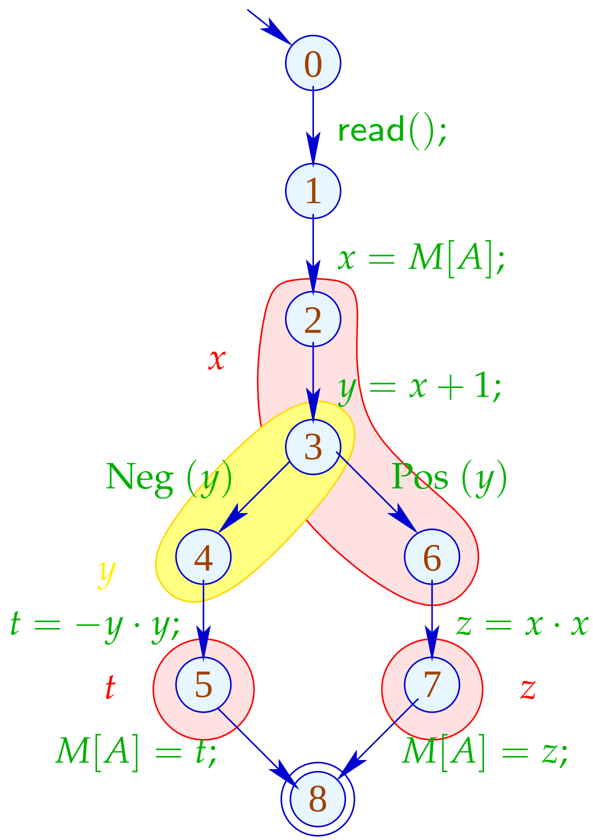
... im Beispiel:



	$\mathcal{L}$
8	$\emptyset$
7	$\{A, z\}$
6	$\{A, x\}$
5	$\{A, t\}$
4	$\{A, y\}$
3	$\{A, x, y\}$
2	$\{A, x\}$
1	$\{A\}$
0	$\emptyset$



	$\mathcal{L}$
8	$\emptyset$
7	$\{A, z\}$
6	$\{A, x\}$
5	$\{A, t\}$
4	$\{A, y\}$
3	$\{A, x, y\}$
2	$\{A, x\}$
1	$\{A\}$
0	$\emptyset$



Lebendigkeitsbereiche:

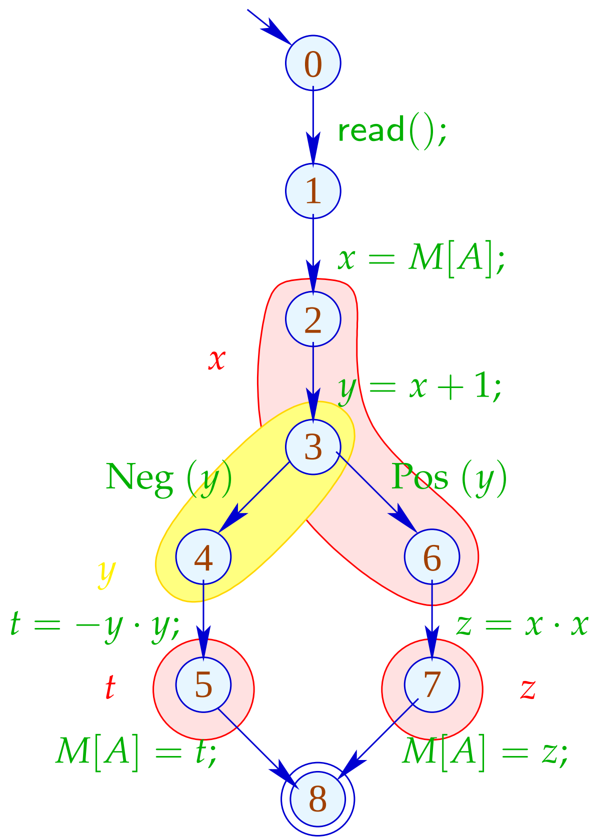
A	$\{1, \ldots, 7\}$
x	$\{2, 3, 6\}$
y	$\{2, 4\}$
t	$\{5\}$
z	$\{7\}$

Um Mengen kompatibler Variablen zu finden, konstruieren wir den Interferenz-Graphen $I = (Vars, E_I)$, wobei:

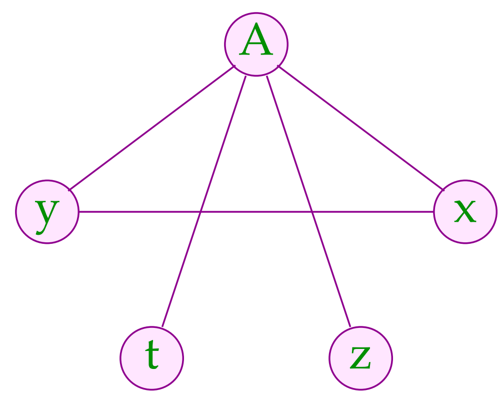
$$E_I = \{\{x, y\} \mid x \neq y, \mathcal{L}[x] \cap \mathcal{L}[y] \neq \emptyset\}$$

E_I enthält eine Kante für $x \neq y$ genau dann wenn x, y an einem gemeinsamen Punkt lebendig sind :-)

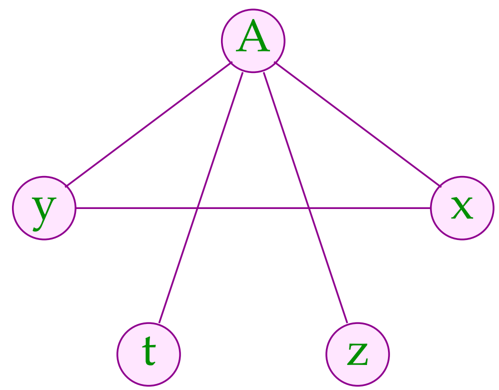
... im Beispiel:



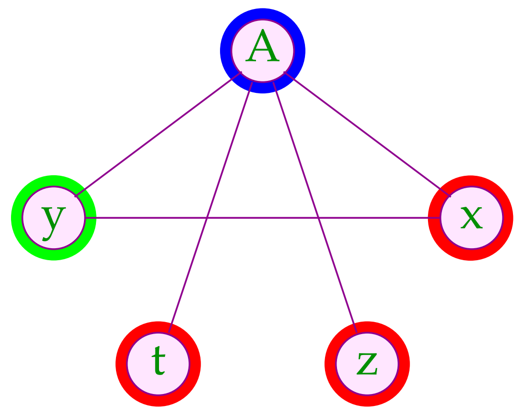
Interferenz-Graph:



Variablen, die nicht mit einer Kante verbunden sind, dürfen dem gleichen Register zugeordnet werden :-)

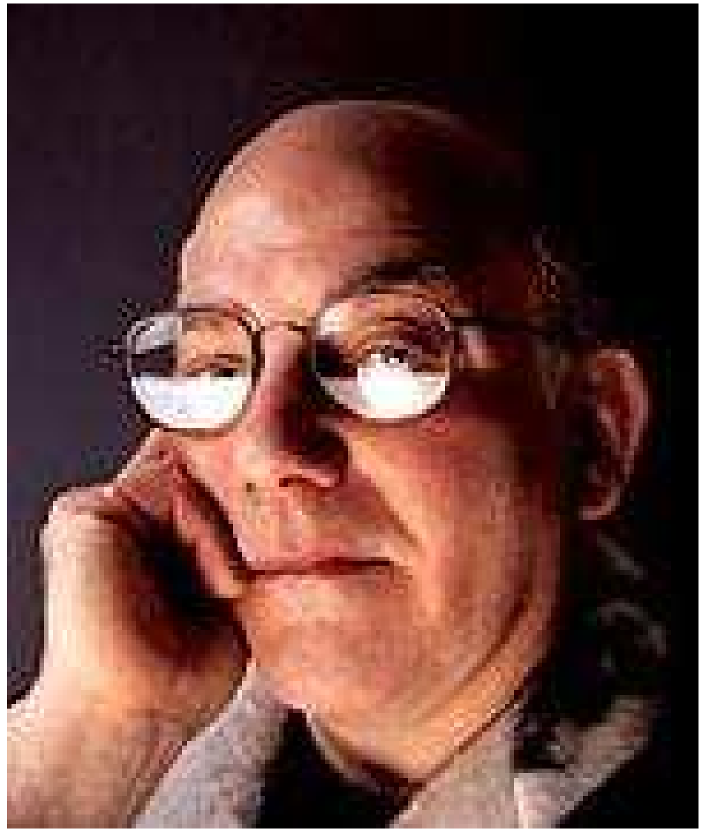


Variablen, die nicht mit einer Kante verbunden sind, dürfen dem gleichen Register zugeordnet werden :-)



Farbe $=$ Register

Sviatoslav Sergeevich Lavrov,
Russische Akademie der Wissenschaften (1962)



Gregory J. Chaitin, University of Maine (1981)

Abstraktes Problem:

Gegeben: Ungerichteter Graph (V, E) .

Gesucht: Minimale Färbung, d.h. Abbildung $c : V \to \mathbb{N}$ mit

(1) $c(u) \neq c(v)$ für $\{u, v\} \in E$;

(2) $\bigsqcup\{c(u) \mid u \in V\}$ minimal!

- Im Beispiel reichen 3 Farben :-) Aber Achtung:
- Die minimale Färbung ist i.a. nicht eindeutig :-(
- Es ist NP-vollständig herauszufinden, ob eine Färbung mit maximal k Farben möglich ist :-((

$\Longrightarrow$

Wir sind auf Heuristiken angewiesen oder Spezialfälle :-)

Greedy-Heuristik:

- Beginne irgendwo mit der Farbe 1;
- Wähle als jeweils neue Farbe die kleinste Farbe, die verschieden ist von allen bereits gefärbten Nachbarn;
- Ist ein Knoten gefärbt, färbe alle noch nicht gefärbten Nachbarn;
- Behandle eine Zusammenhangskomponente nach der andern

  ...

... etwas konkreter:

```
forall (v ∈ V) c[v] = 0;
forall (v ∈ V) color (v);

void color (v) {
    if (c[v] ≠ 0) return;
    neighbors = {u ∈ V | {u,v} ∈ E};
    c[v] = ⨅{k > 0 | ∀ u ∈ neighbors : k ≠ c(u)};
    forall (u ∈ neighbors)
        if (c(u) == 0) color (u);
}
```

Die neue Farbe lässt sich leicht berechnen, nachdem die Nachbarn nach ihrer Farbe geordnet wurden :-)

Diskussion:

$\rightarrow$ Im wesentlichen ist das Prä-order DFS :-)

$\rightarrow$ In der Theorie kann das Ergebnis beliebig weit vom Optimum entfernt sein :-(

$\rightarrow$ ... ist aber in der Praxis ganz gut :-)

$\rightarrow$ ... Achtung: verschiedene Varianten sind patentiert !!!

Diskussion:

→ Im wesentlichen ist das Prä-order DFS :-)

→ In der Theorie kann das Ergebnis beliebig weit vom Optimum entfernt sein :-(

→ ... ist aber in der Praxis ganz gut :-)

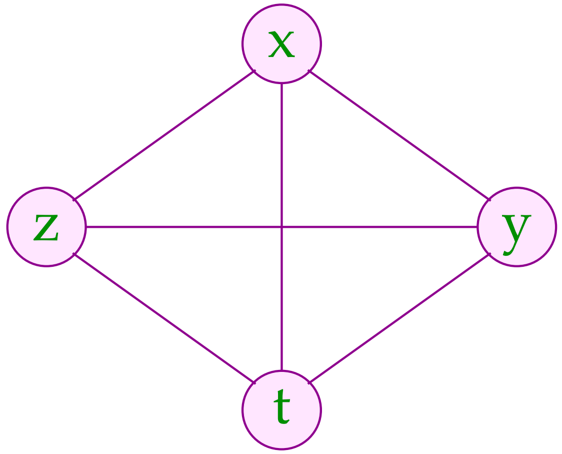
→ ... Achtung: verschiedene Varianten sind patentiert !!!

Der Algorithmus funktioniert umso besser, je kleiner die Lebendigkeitsbereiche sind ...

Idee: Life range splitting

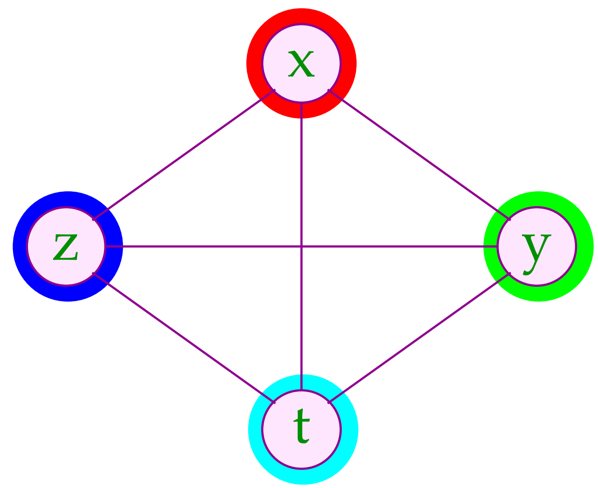
Beispiel:

	$\mathcal{L}$
	x, y, z
$A_1 = x + y;$	x, z
$M[A_1] = z;$	x
$x = x + 1;$	x
$z = M[A_1];$	x, z
$t = M[x];$	x, z, t
$A_2 = x + t;$	x, z, t
$M[A_2] = z;$	x, t
$y = M[x];$	y, t
$M[y] = t;$	



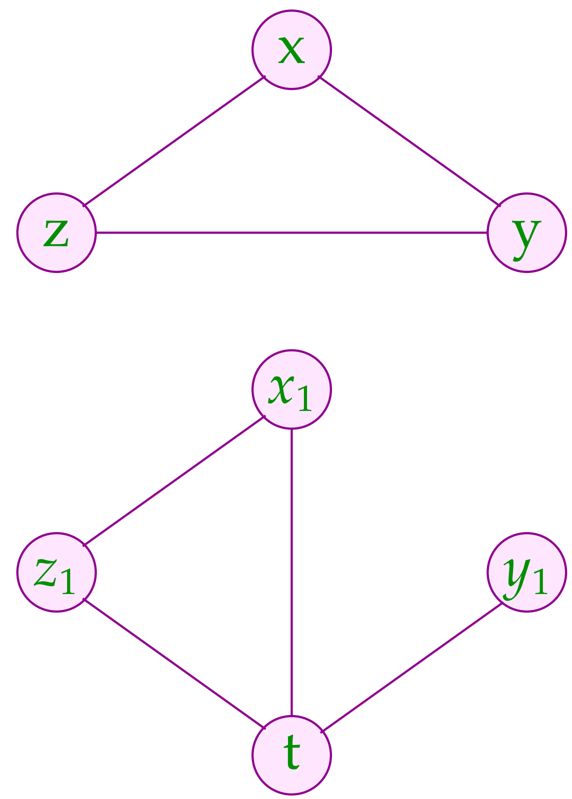
Beispiel:

	$\mathcal{L}$
	x, y, z
$A_1 = x + y;$	x, z
$M[A_1] = z;$	x
$x = x + 1;$	x
$z = M[A_1];$	x, z
$t = M[x];$	x, z, t
$A_2 = x + t;$	x, z, t
$M[A_2] = z;$	x, t
$y = M[x];$	y, t
$M[y] = t;$	



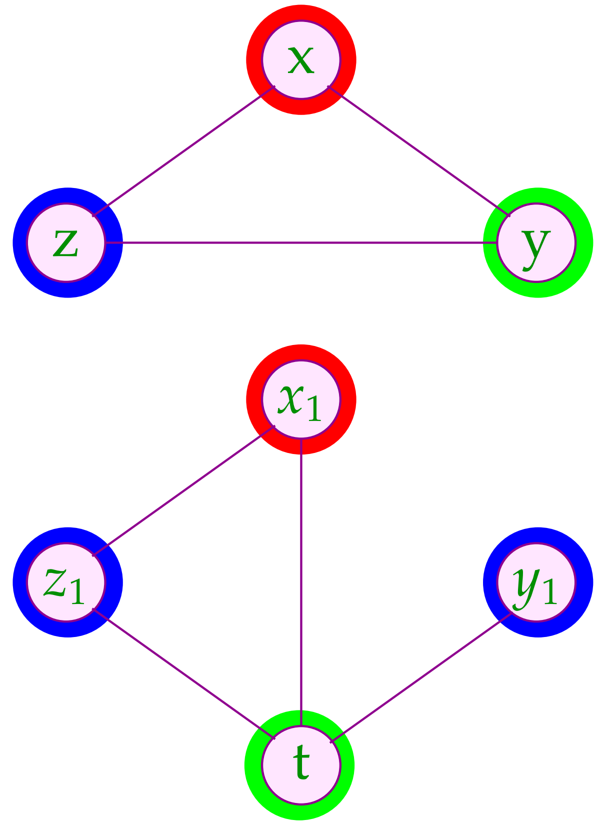
Die Lebendigkeitsbereiche von x und z können wir aufteilen:

	$\mathcal{L}$
	x, y, z
$A_1 = x + y;$	x, z
$M[A_1] = z;$	x
$x_1 = x + 1;$	x_1
$z_1 = M[A_1];$	x_1, z_1
$t = M[x_1];$	x_1, z_1, t
$A_2 = x_1 + t;$	x_1, z_1, t
$M[A_2] = z_1;$	x_1, t
$y_1 = M[x_1];$	y_1, t
$M[y_1] = t;$	



Die Lebendigkeitsbereiche von x und z können wir aufteilen:

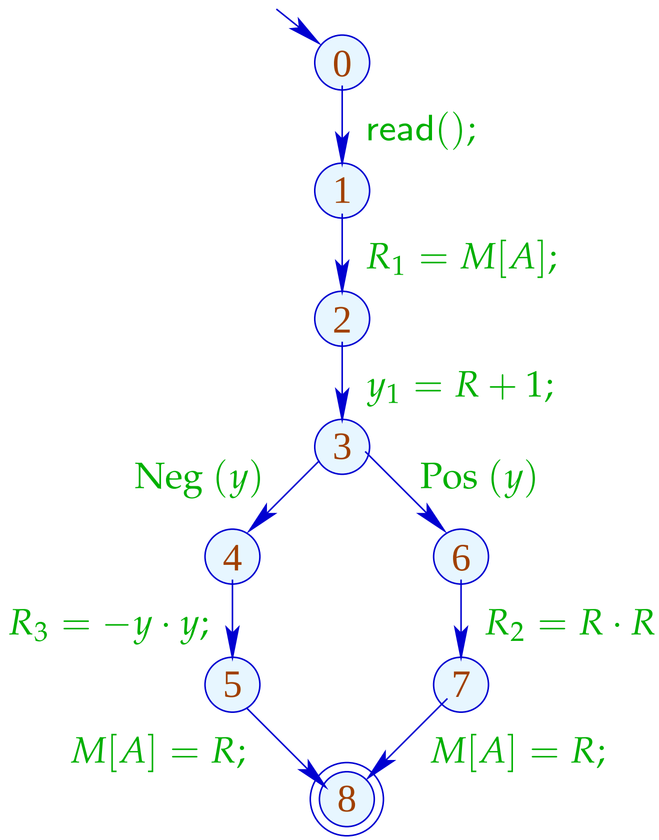
	$\mathcal{L}$
	x, y, z
$A_1 = x + y;$	x, z
$M[A_1] = z;$	x
$x_1 = x + 1;$	x_1
$z_1 = M[A_1];$	x_1, z_1
$t = M[x_1];$	x_1, z_1, t
$A_2 = x_1 + t;$	x_1, z_1, t
$M[A_2] = z_1;$	x_1, t
$y_1 = M[x_1];$	y_1, t
$M[y_1] = t;$	

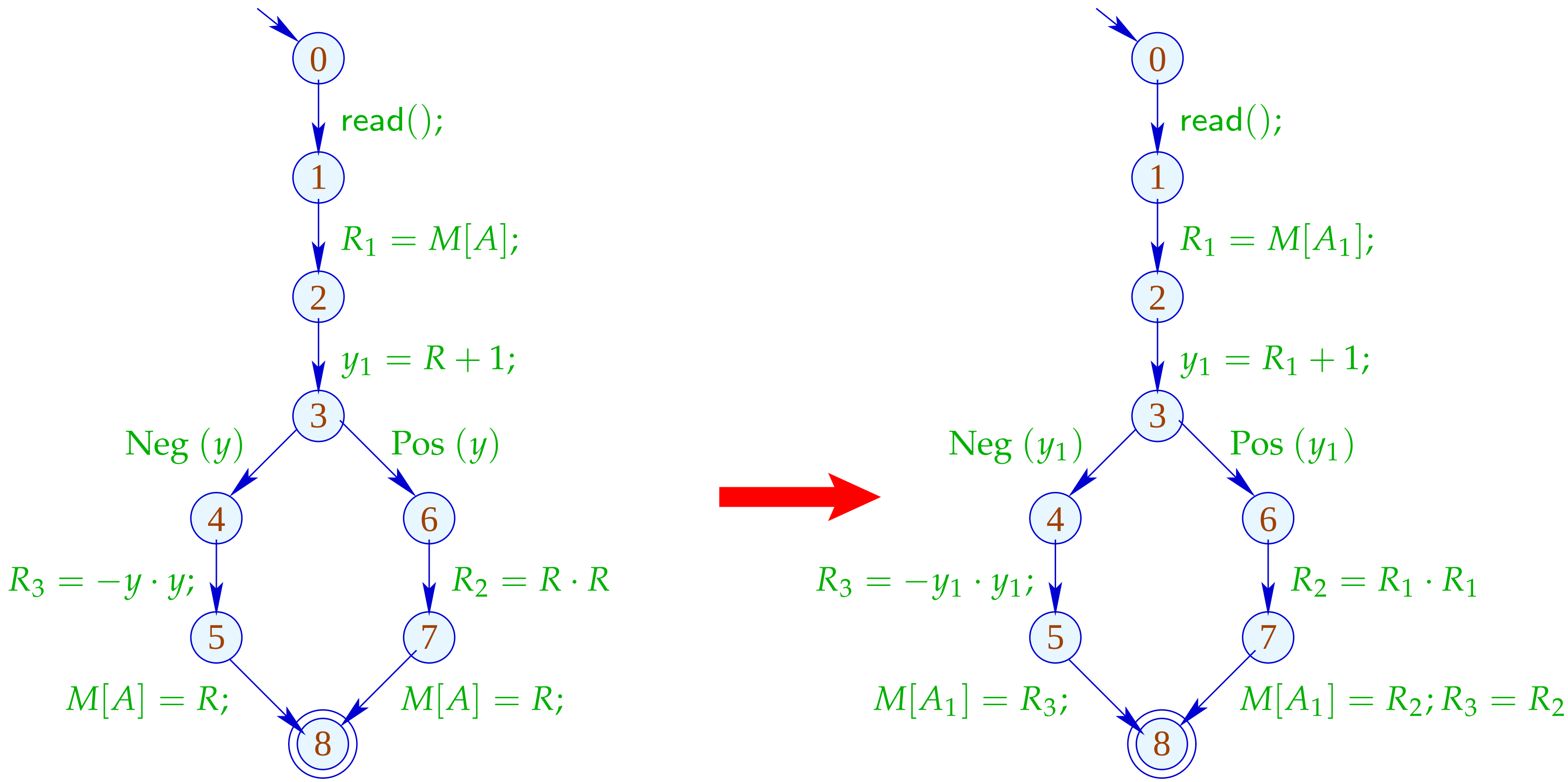


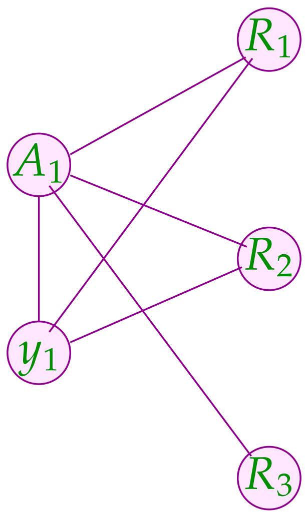
Idee:

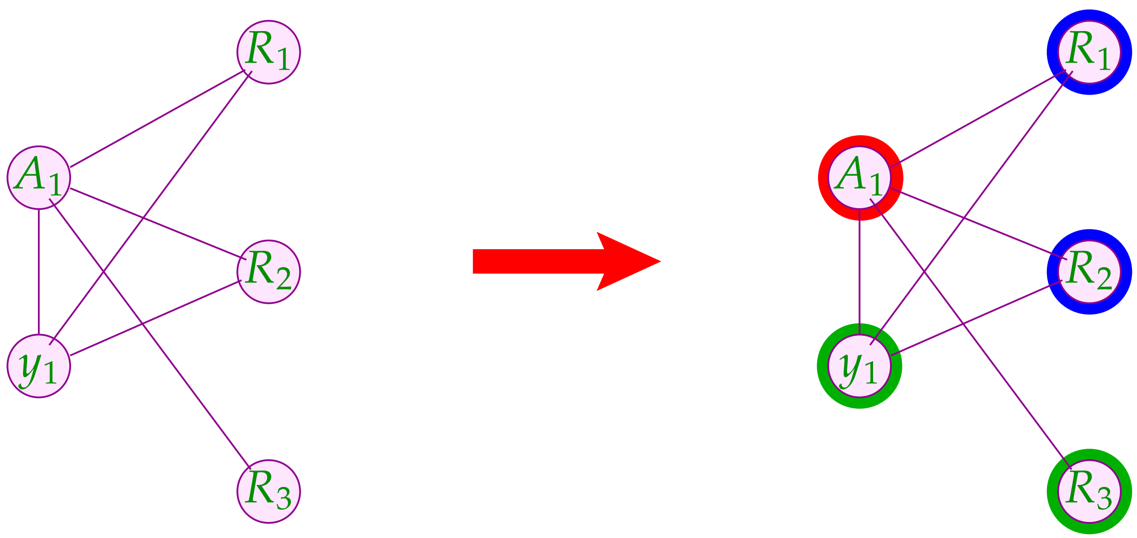
- Führe eine neue Variante x_i für jedes Vorkommen der Definition der Variable x ein!
- Berechne für jeden Programmpunkt die Menge der ankommenden Varianten!
- An Zusammenflüssen der Kontrolle wähle jeweils eine als Hauptvariante x_h aus!
- Füge an allen zusammenlaufenden Kanten Zuweisungen $x_h = x_j;$ an die Hauptvariante ein!
- Ersetze die Benutzung von x durch die Benutzung der entsprechenden Variante x_h ...

$\Longrightarrow$ Static Single Assignment Form


0
read();
1
$R_1 = M[A];$
2
$y_1 = R + 1;$
3
Neg (y)
Pos (y)
4
6
$R_3 = -y \cdot y;$
$R_2 = R \cdot R$
5
7
$M[A] = R;$
$M[A] = R;$
8


0
read();
1
$R_1 = M[A];$
2
$y_1 = R + 1;$
3
Neg (y)
Pos (y)
4
6
$R_3 = -y \cdot y;$
$R_2 = R \cdot R$
5
7
$M[A] = R;$
$M[A] = R;$
8
0
read();
1
$R_1 = M[A_1];$
2
$y_1 = R_1 + 1;$
3
Neg (y_1)
Pos (y_1)
4
6
$R_3 = -y_1 \cdot y_1;$
$R_2 = R_1 \cdot R_1$
5
7
$M[A_1] = R_3;$
$M[A_1] = R_2; R_3 = R_2;$
8


R_1
A_1
R_2
y_1
R_3


R_1
A_1
R_2
y_1
R_3
A_1
R_1
R_2
y_1
R_3

Large Beobachtung:

- Das Erzeugen der SSA fügt zusätzliche Register-Umspeicherungen ein :-(
- Die Interferenz-Graphen von Programmen in SSA sind besonders einfach ...
- Werden an jedem Zusammenlauf selbst stets neue Varianten eingeführt, besitzt jeder Kreis mit mehr als drei Knoten eine Sehne.

  Solche Graphen heißen chordal.

Theorem:

Jeder chordale Graph lässt sich in polynomieller Zeit optimal färben.

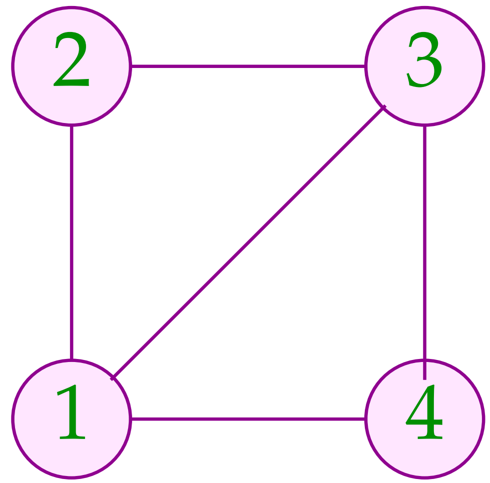
Theorem:

Jeder chordale Graph lässt sich in polynomieller Zeit optimal färben.

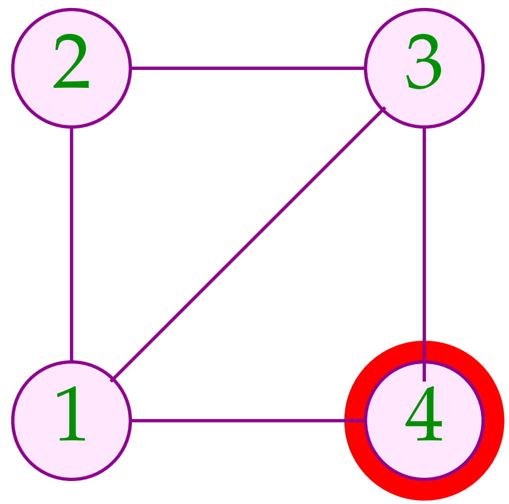
Idee:

- Entnehme einen Knoten v, dessen Nachbarn eine eine Clique bilden.
- Färbe den Graphen ohne v.
- Schließlich färbe v.

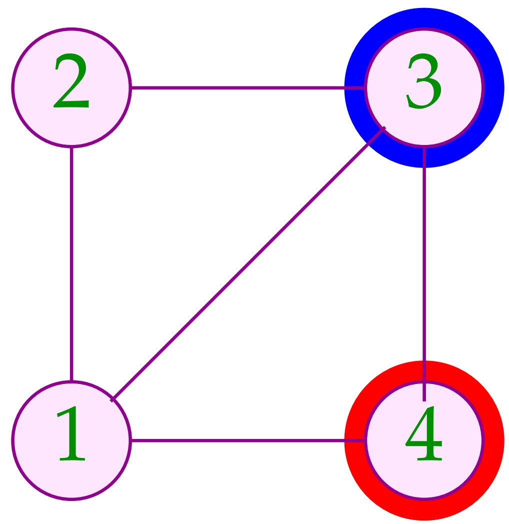
Eine solche Entnahmesequenz kann in polynomieller Zeit gefunden werden :-)



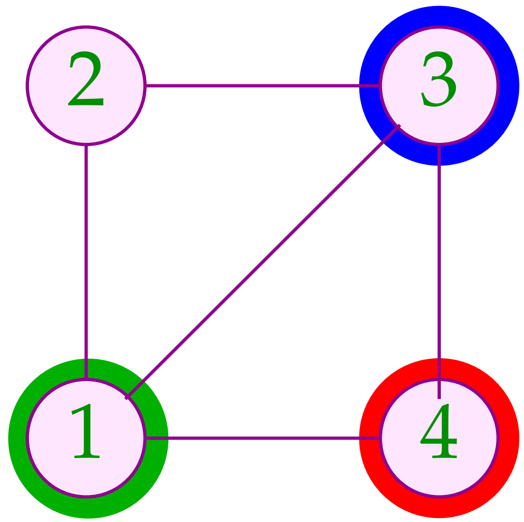
Entnahmefolge: 2, 1, 3, 4



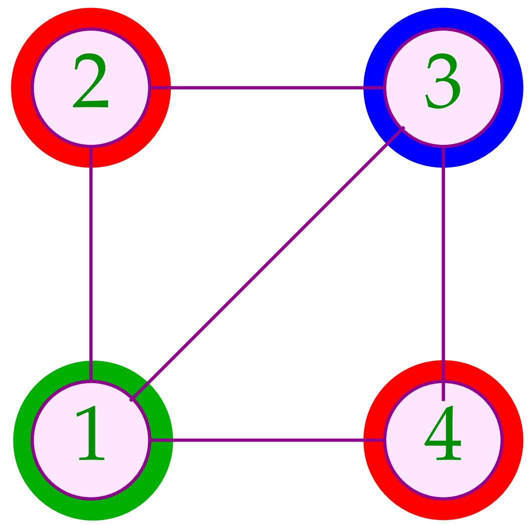
Entnahmefolge: 2, 1, 3, 4



Entnahmefolge: 2, 1, 3, 4



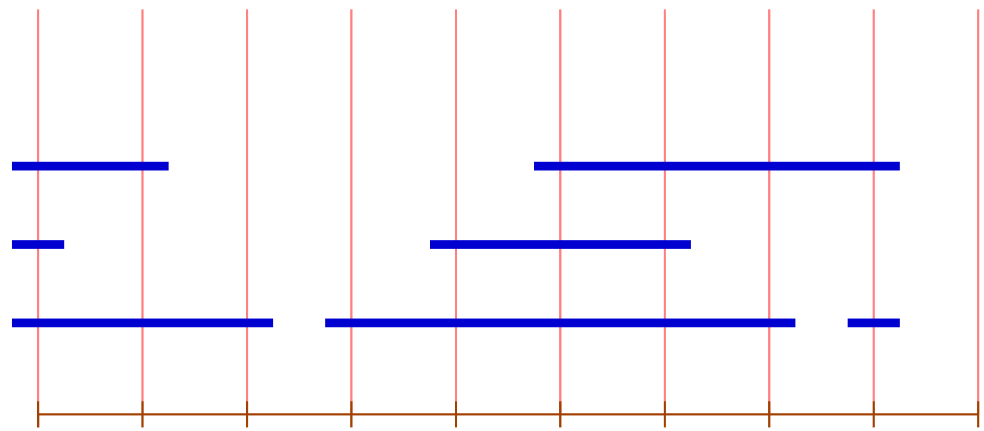
Entnahmefolge: 2, 1, 3, 4



Entnahmefolge: 2, 1, 3, 4

Spezialfall: Basis-Blöcke

Die Interferenzgraphen für minimale Lebendigkeitsbereiche auf Folgen von Zuweisungen sind Intervall-Graphen:



Knoten	$=$	Intervall
Kante	$=$	gemeinsamer Punkt

Zu jedem Punkt können wir die Überdeckungszahl der inzidenten Intervalle angeben.

Satz:

$$
\begin{aligned}
&\text{maximale Überdeckungszahl} \\
&\qquad\qquad = \text{Größe der maximalen Klique} \\
&\qquad\qquad = \text{maximal nötige Anzahl Farben} \quad :\text{-})
\end{aligned}
$$

Graphen mit dieser Eigenschaft heißen perfekt ...

Eine minimale Färbung kann in polynomieller Zeit berechnet werden :-))

Idee:

- $\longrightarrow$ Iteriere (konzeptuell) über die Punkte $0, \ldots, m-1$!
- $\longrightarrow$ Verwalte eine Liste der aktuell freien Farben.
- $\longrightarrow$ Beginnt ein neues Intervall, vergib die nächste freie Farbe.
- $\longrightarrow$ Endet ein Intervall, gib seine Farbe frei.

Damit ergibt sich folgender Algorithmus:

```
free = [1, ..., k];
for (i = 0; i < m; i++) {
    init[i] = []; exit[i] = [];
}
forall (I = [u, v] ∈ Intervals) {
    init[u] = (I :: init[u]); exit[i] = (I :: exit[v]);
}
    for (i = 0; i < m; i++) {
    forall (I ∈ exit[i]) free = color[I] :: free;
    forall (I ∈ init[i]) {
        color[I] = hd free; free = tl free;
    }
}
```

Diskussion:

- → Für Basis-Blöcke können wir eine optimale Aufteilung der Variablen auf eine Register ermitteln :-)
- → Das gleiche Problem ist bereits für einfache Schleifen (circular arc graphs) NP-schwierig :-(
- → Für beliebige Programme wird man deshalb eine Heuristik zum Graph-Färben einsetzen ...
- → Dieses Verfahren funktioniert besser, wenn wir die Lebendigkeitsbereiche maximal unterteilen :-)
- → Reicht die Anzahl der realen Register nicht aus, lagert man die überzähligen in einen festen Speicherbereich aus.
- → Man bemüht sich dabei, zumindest die in innersten Schleifen benutzten Variablen in Registern zu halten.

Interprozedurale Registerverteilung:

→ Für jede lokale Variable ist ein Eintrag im Kellerrahmen reserviert.

→ Vor dem Aufruf einer Funktion müssen die Register in den Kellerrahmen gerettet und danach restauriert werden.

→ Gelegentlich gibt es dafür Hardware-Unterstützung :-)

Dann ist ein Aufruf für alle Register transparent.

→ Verwalten wir Retten / Restaurieren selbst, können wir ...

- nur Register retten, deren Inhalte nach dem Aufruf noch benötigt werden :-)
- Register erst bei Bedarf restaurieren — und dann evt. in andere Register $\Longrightarrow$ Verkleinerung der Lebendigkeitsbereiche :-)