Diskussion:

- Im Beispiel terminiert die Analyse schnell :-)
- Falls $\mathbb{D}$ endliche Höhe hat, terminiert die Analyse, sofern nur jede Funktion während der Iteration nur mit endlich vielen verschiedenen Argumenten aufgerufen wird :-))
- Analoge Analyse-Algorithmen erwiesen sich bei der Analyse von Prolog als äußerst effizient und präzise :-)
- Zusammen mit einer Points-To-Analyse und Propagation selbst von negativer Konstanten-Information haben wir diesen Algorithmus äußerst erfolgreich zur Fehlersuche in C mit Posix-Threads eingesetzt :-)

(2) Der Call-String-Ansatz:

Idee:

→ Berechne die Menge aller erreichbaren Aufrufkeller!

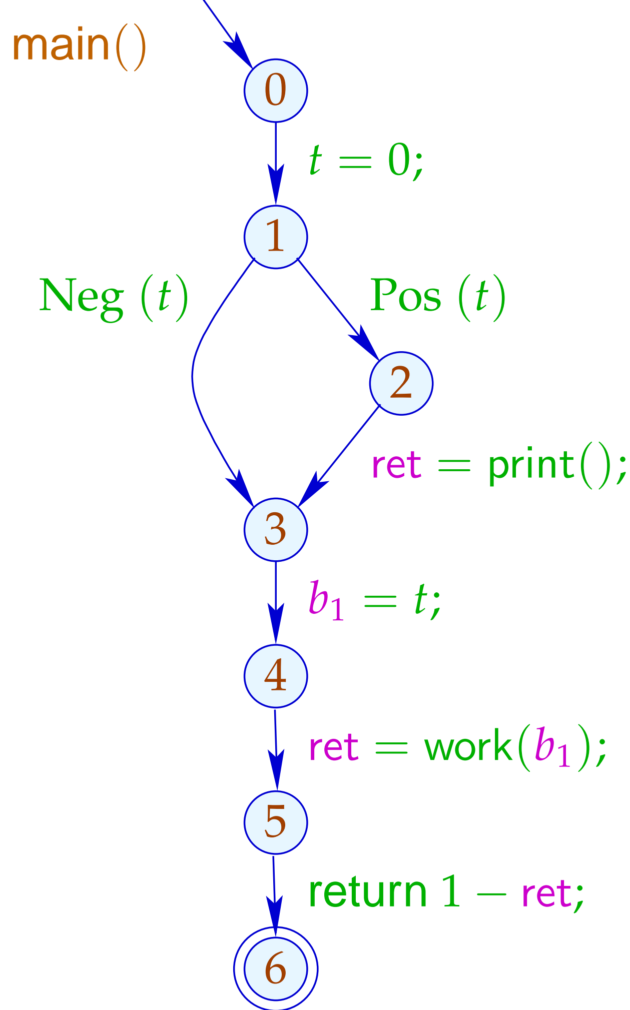
→ Diese ist i.a. unendlich :-(

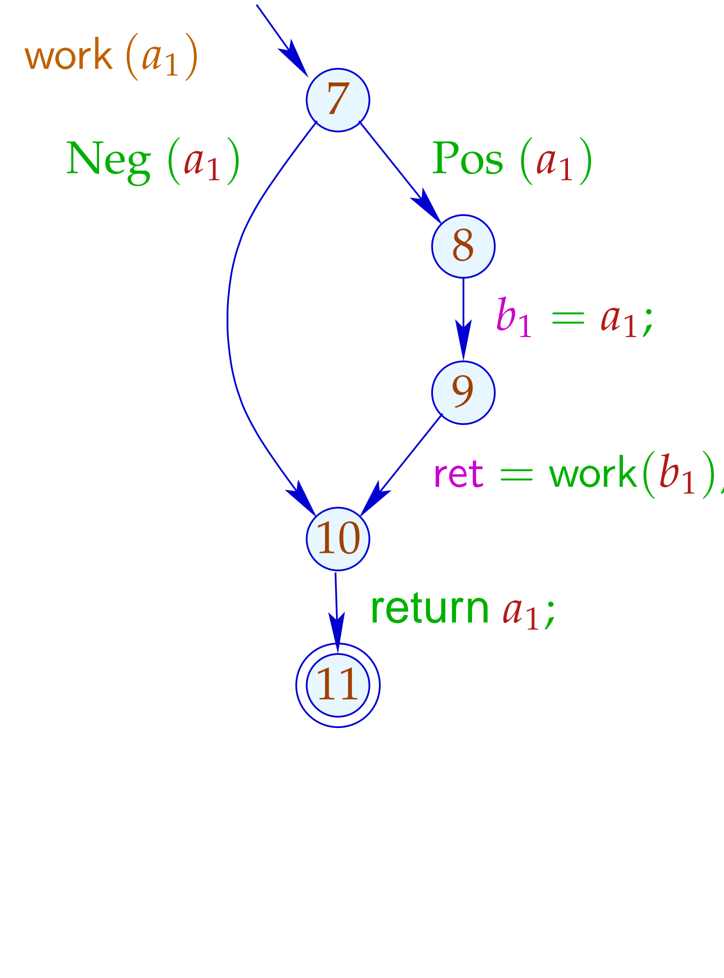
→ Behandle Keller bis zu einer festen Tiefe d exakt! Behalte von längeren Kellern nur das obere Ende der Länge d :-)

→ Wichtiger Spezialfall: $d = 0$.

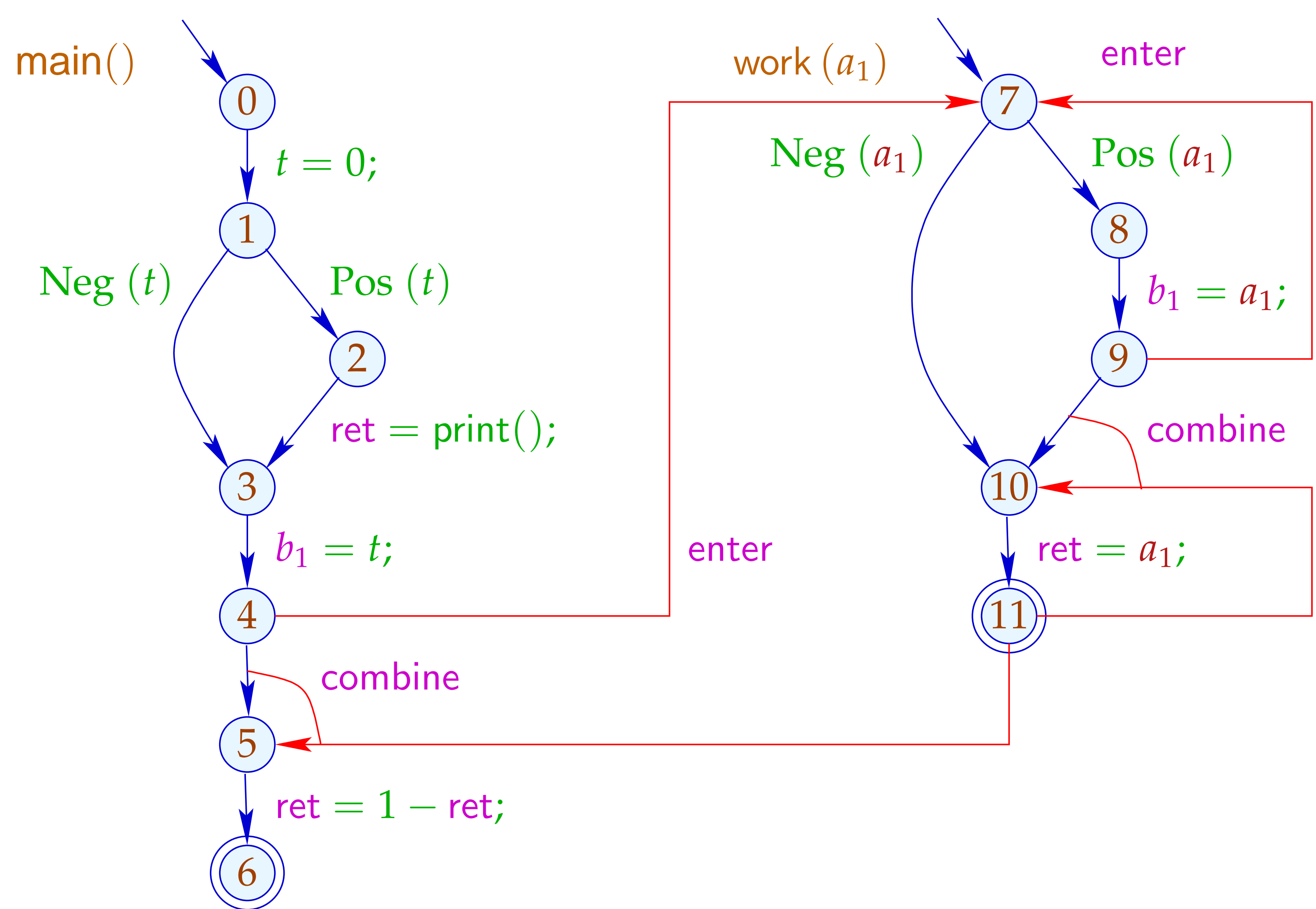
$\Longrightarrow$ Betrachte nur die obersten Kellerrahmen ...

... im Beispiel:





... im Beispiel:



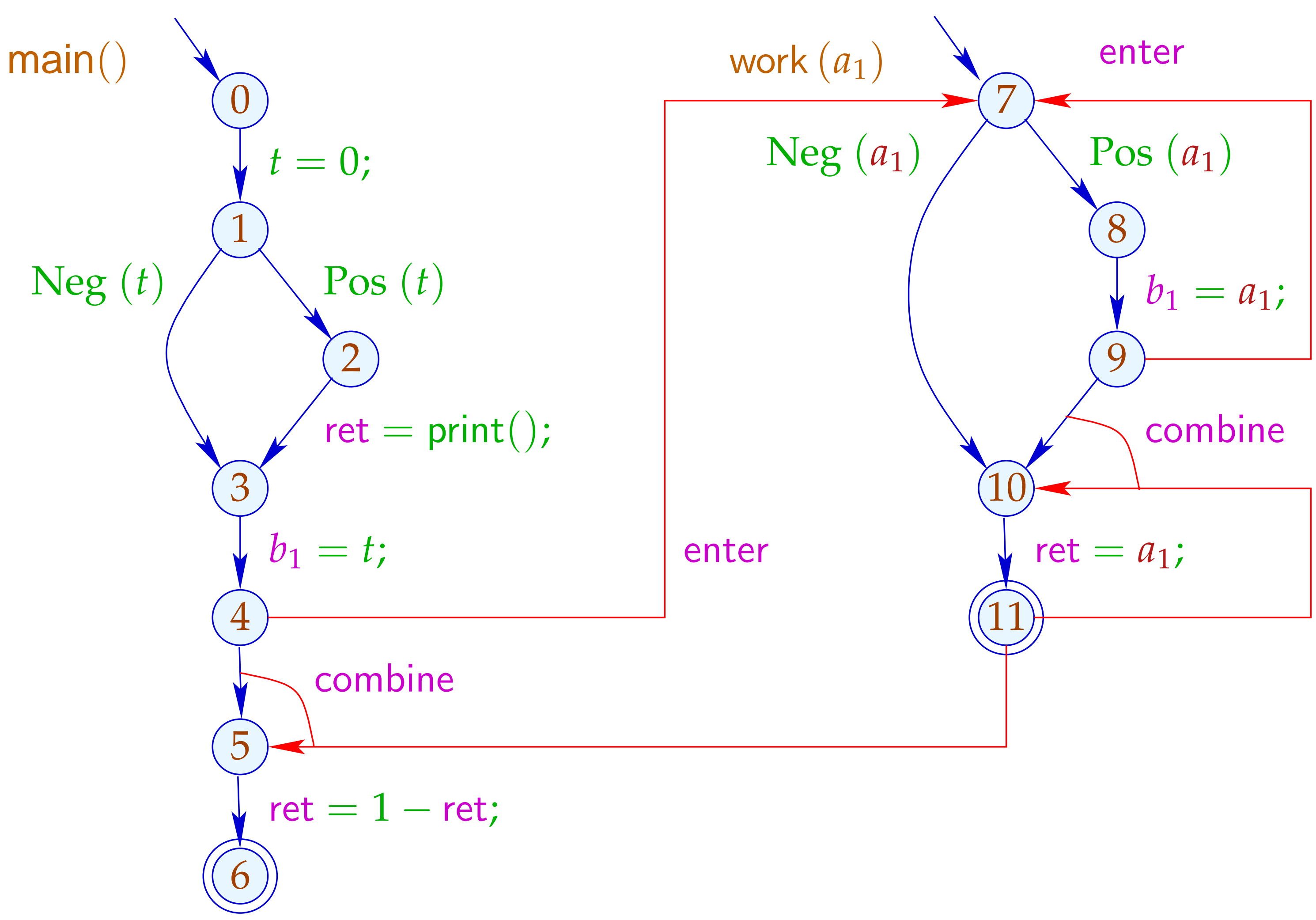
Die Bedingungen für $5, 7, 10$ sind dann etwa:

$$\begin{array}{lll} \mathcal{R}[5] & \sqsupseteq & \mathsf{combine}^\sharp\,(\mathcal{R}[4], \mathcal{R}[11]) \\ \mathcal{R}[7] & \sqsupseteq & \mathsf{enter}^\sharp_f\,(\mathcal{R}[4]) \\ \mathcal{R}[7] & \sqsupseteq & \mathsf{enter}^\sharp_f\,(\mathcal{R}[9]) \\ \mathcal{R}[10] & \sqsupseteq & \mathsf{combine}^\sharp\,(\mathcal{R}[9], \mathcal{R}[11]) \end{array}$$

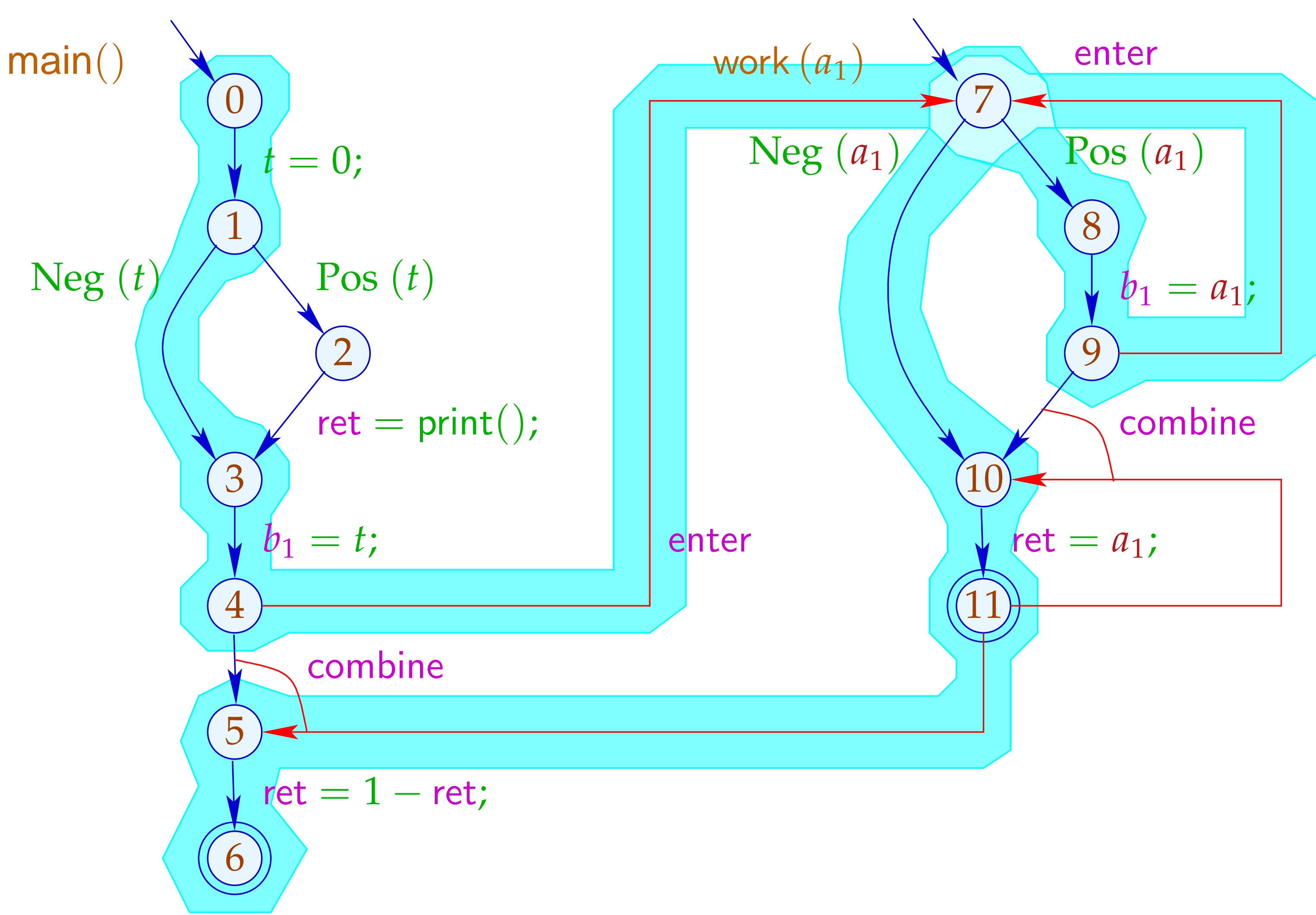
Achtung:

Der resultierende Supergraph enthält offensichtlich unmögliche Pfade ...

... im Beispiel ist das etwa:



... im Beispiel ist das etwa:



Beachte:

- $\longrightarrow$ Im Beispiel finden wir zwar die gleichen Ergebnisse: Mehr Pfade machen die Ergebnissse evt. weniger präzise.

  Insbesondere analysieren wir jede Funktion nur für ein (evt. sehr nichtssagendes) Argument-Tupel :-(

- $\longrightarrow$ Die Analyse terminiert — sofern nur $\mathbb{D}$ keine unendlichen echt aufsteigenden Ketten besitzt :-)

- $\longrightarrow$ Die Korrektheit zeigt man relativ zur operationellen Semantik mit den Stacks.

- $\longrightarrow$ Für die Korrektheit des funktionalen Ansatzes ist die Semantik über Berechnungswälder besser geeignet :-)

3 Ausnutzung von Hardware-Einrichtungen

Frage: Wie nutzt man optimal

... Register

... Instruktionen

... Pipelines

... Caches

... Prozessoren ???

3.1 Register

Beispiel:

$$A = \text{read}();$$
$$x = M[A];$$
$$y = x + 1;$$
$$\text{if } (y) \; \{$$
$$\quad z = x \cdot x;$$
$$\quad M[A] = z;$$
$$\} \text{ else } \{$$
$$\quad t = -y \cdot y;$$
$$\quad M[A] = t;$$
$$\}$$

Das Programm benötigt 5 Variablen ...

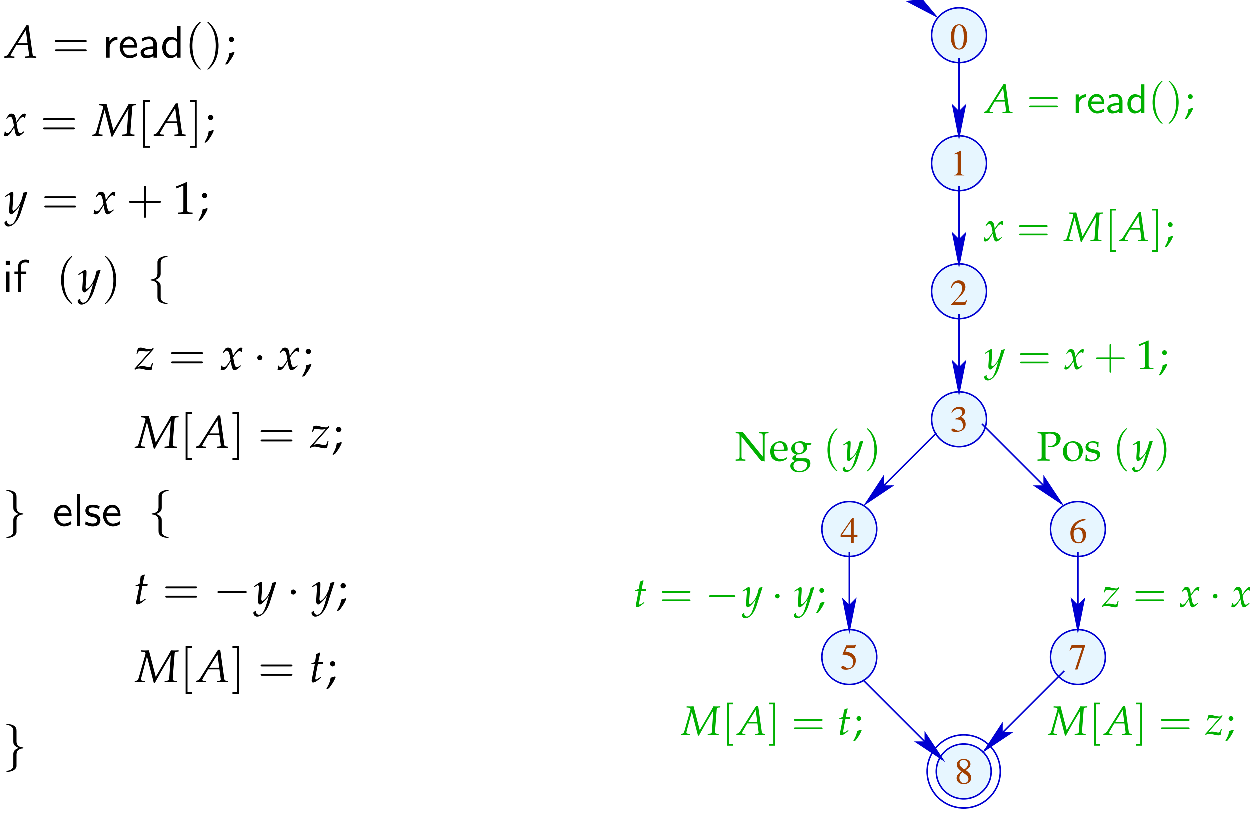
Problem:

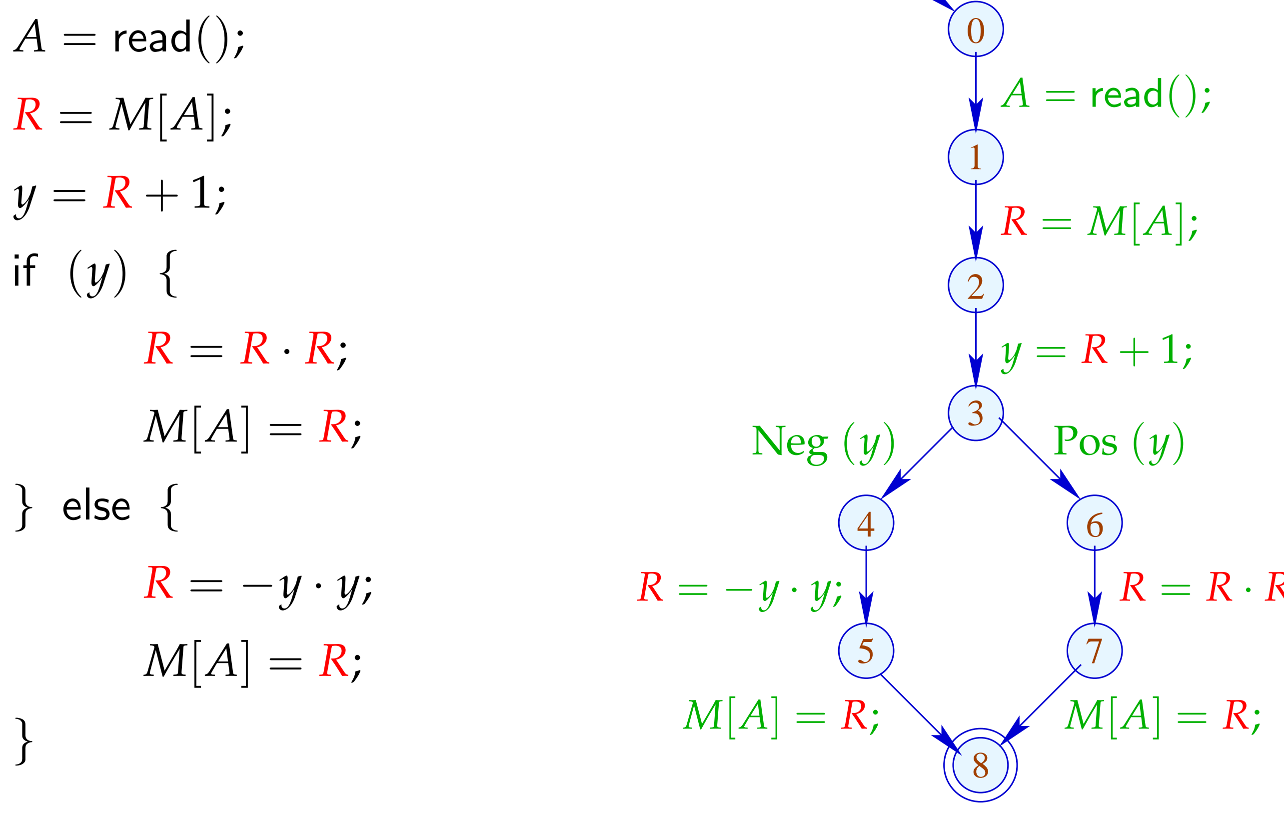
Was tun, wenn das Programm benutzt mehr Variablen als Register da sind :-(

Idee:

Benutze ein Register für mehrere Variablen :-)

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


A = read();
x = M[A];
y = x + 1;
if (y) {
z = x · x;
M[A] = z;
} else {
t = −y · y;
M[A] = t;
}
0
A = read();
1
x = M[A];
2
y = x + 1;
3
Neg (y)
Pos (y)
4
6
t = −y · y;
z = x · x
5
7
M[A] = t;
M[A] = z;
8


A = read();
R = M[A];
y = R + 1;
if (y) {
R = R · R;
M[A] = R;
} else {
R = −y · y;
M[A] = R;
}
0
A = read();
1
R = M[A];
2
y = R + 1;
3
Neg (y)
Pos (y)
4
6
R = −y · y;
R = R · R
5
7
M[A] = R;
M[A] = R;
8

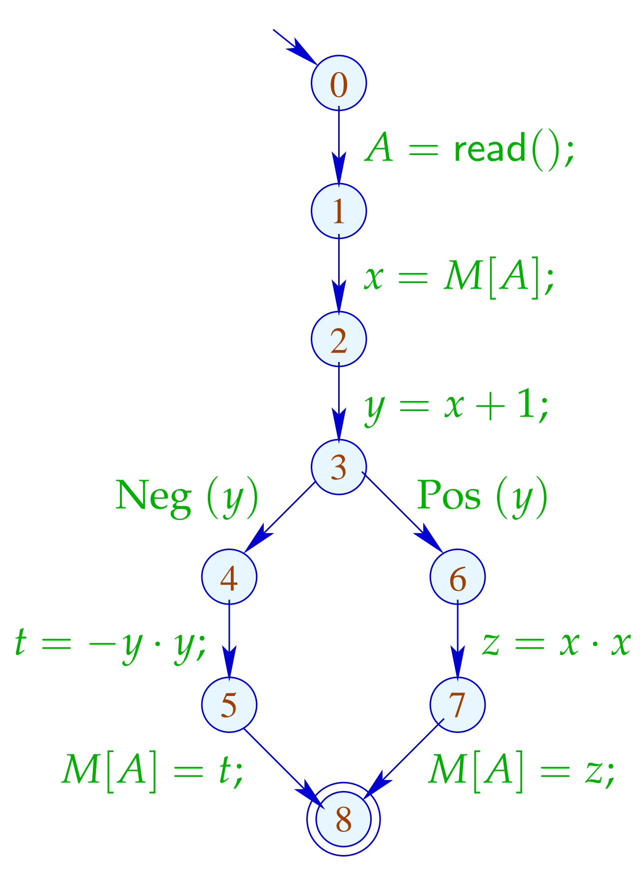
Achtung:

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

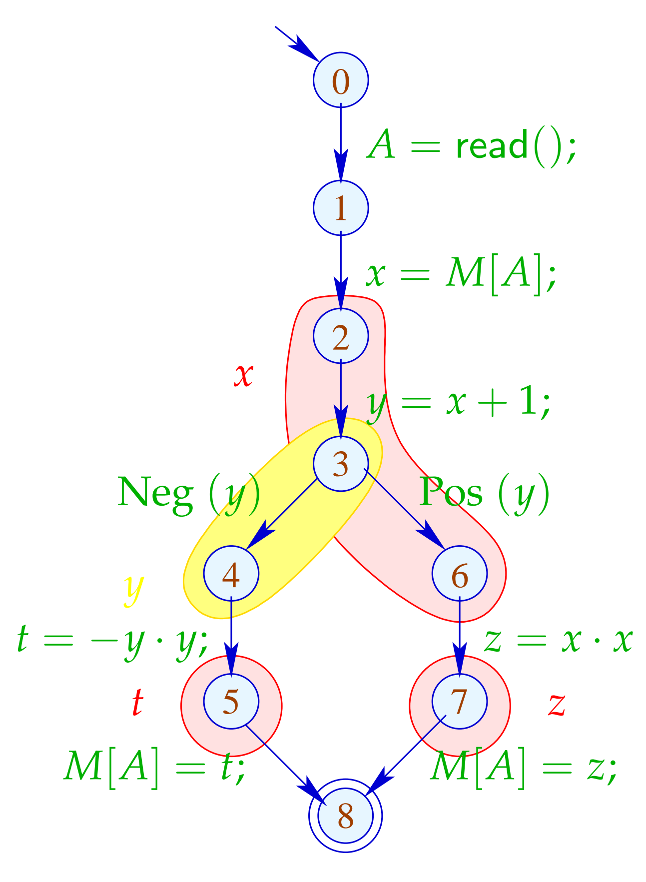
Der (wahre) Lebendigkeitsbereich von x ist:

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

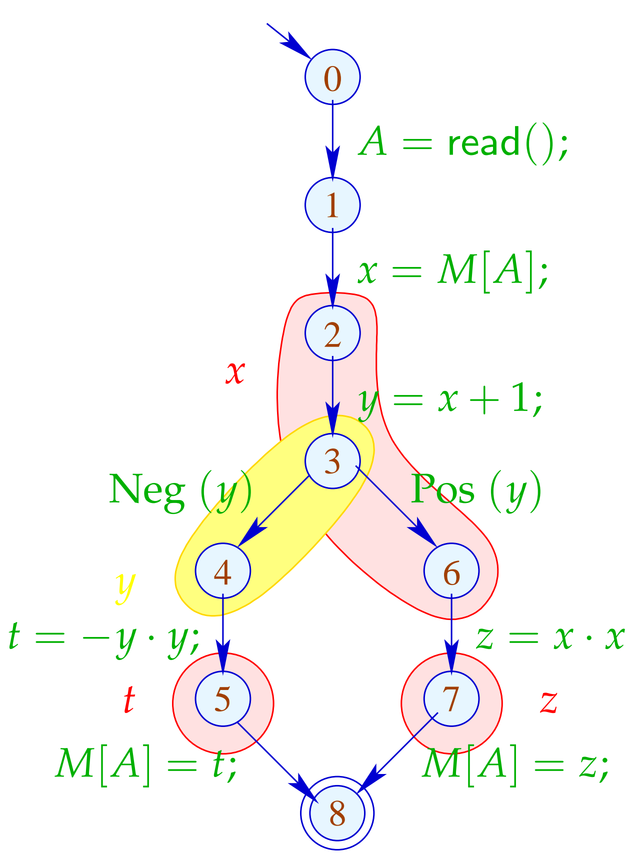
... im Beispiel:



	$\mathcal{L}$
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:

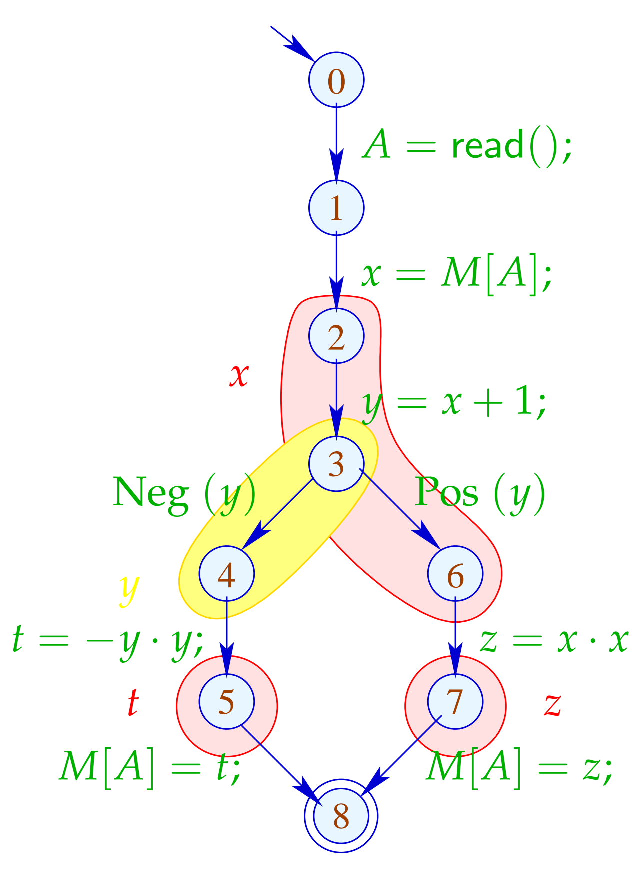
A	$\{1, \ldots, 7\}$
x	$\{2, 3, 6\}$
y	$\{2, 4\}$
t	$\{5\}$
z	$\{7\}$

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

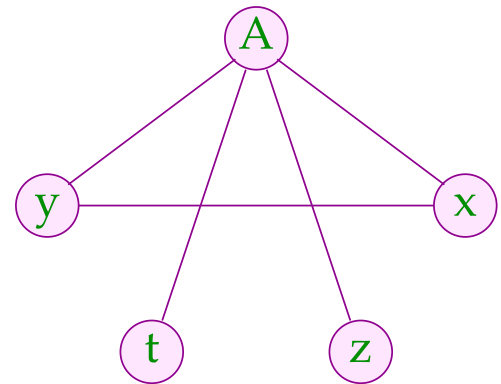
$$E_I = \{\{x, y\} \mid x \neq y, \mathcal{L}[x] \cap \mathcal{L}[y] \neq \emptyset\}$$

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

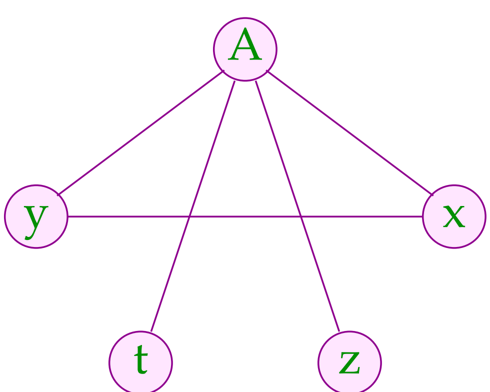
... im Beispiel:



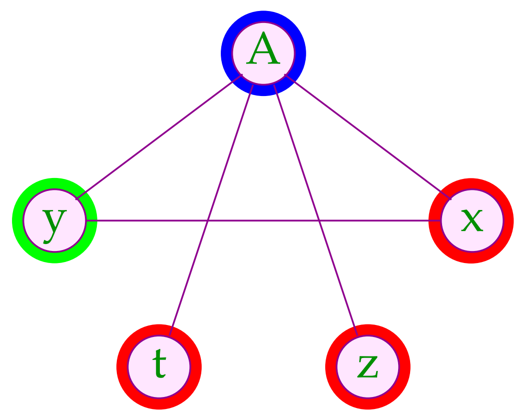
Interferenz-Graph:



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

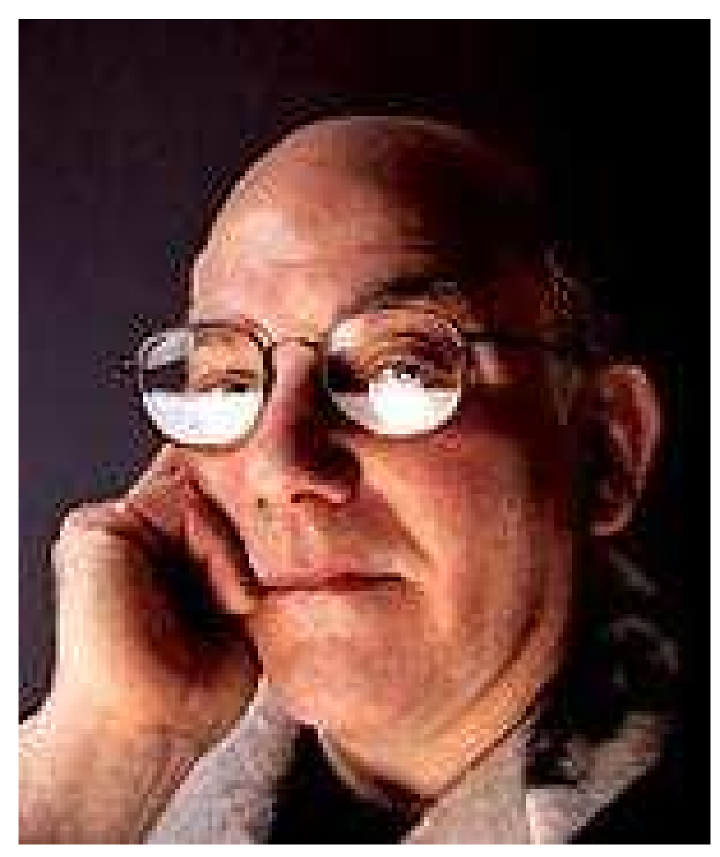


Variablen, die nicht mit einer Kante verbunden sind, dürfen dem gleichen Register zugeordnet werden :-)



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:

→ 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 !!!

Diskussion:

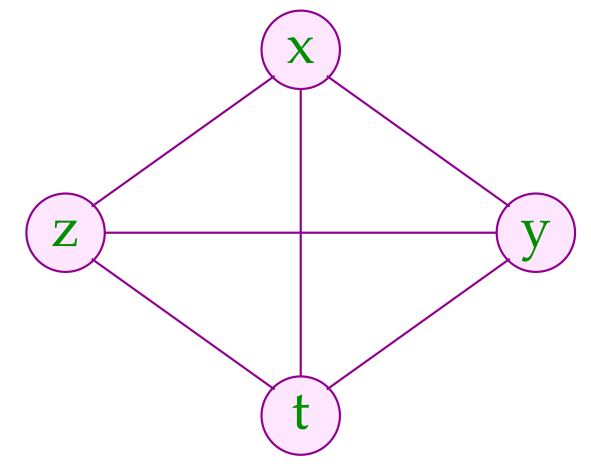
- $\rightarrow$ Im wesentlichen ist das Prä-order DFS :-)
- $\rightarrow$ In der Theorie kann das Ergebnis beliebig weit vom Optimum entfernt sein :-(
- $\rightarrow$ ... ist aber in der Praxis ganz gut :-)
- $\rightarrow$ ... Achtung: verschiedene Varianten sind patentiert !!!

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

Idee: Life range splitting

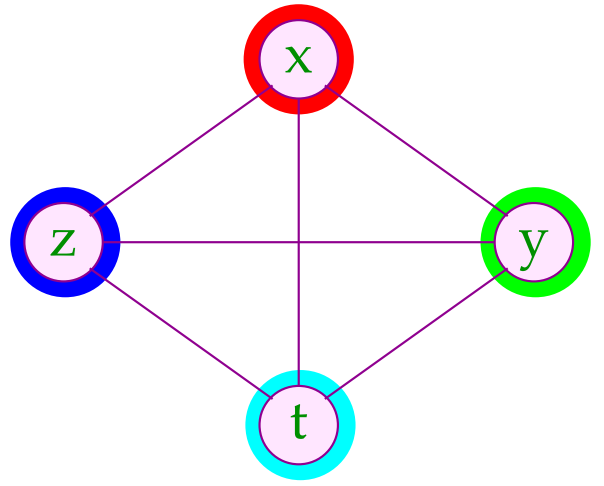
Beispiel:

	$\mathcal{L}$
	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
$M[y] = t;$	



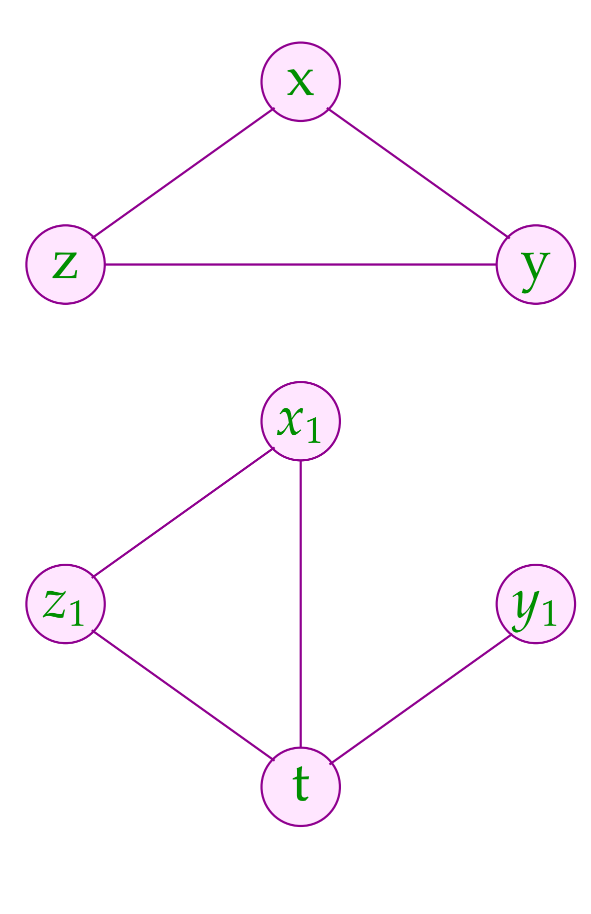
Beispiel:

	$\mathcal{L}$
	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
$M[y] = t;$	



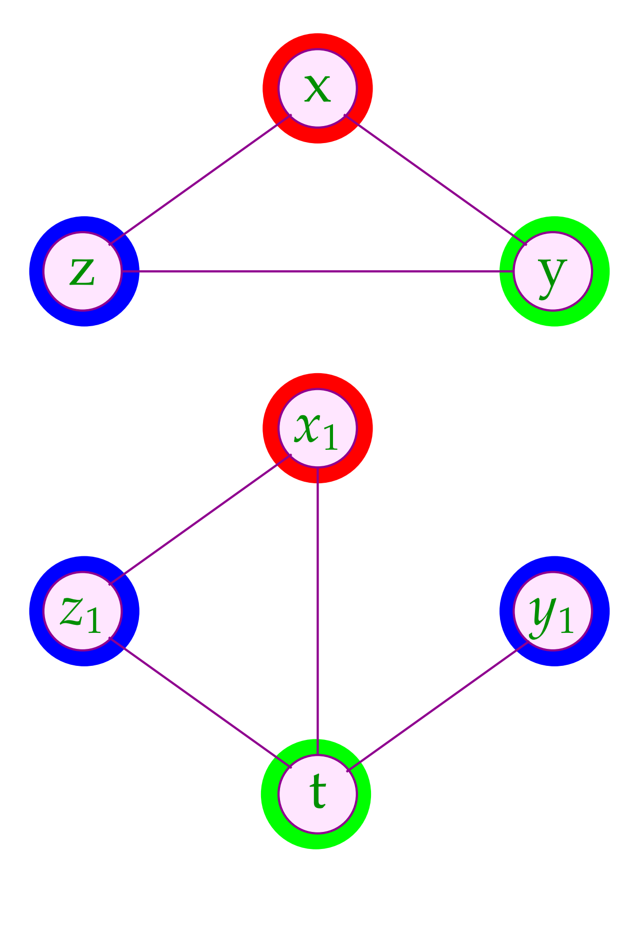
Die Lebendigkeitsbereiche von x und z können wir aufteilen:

	$\mathcal{L}$
	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
$M[y_1] = t;$	



Technisch:

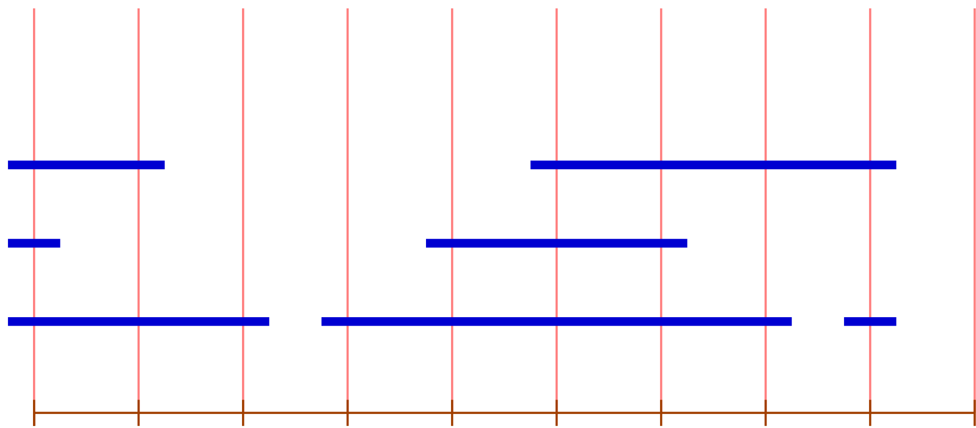
Eine Kante (u, lab, v) heißt x-transparent, falls lab keine Definition von x ist.

u, v gehören zum selben minimalen x-Lebendigkeitsbereich , falls $x \in \mathcal{L}[u] \cap \mathcal{L}[v]$ und u, v durch einen ungerichteten Pfad x-transparenter Kanten verbunden sind ...

Für jeden der minimalen x-Lebendigkeitsbereiche $L_1, \ldots, L_k$ für wir eine Variante von x ein :-)

Spezialfall: Basis-Blocks

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



Knoten $\quad = \quad$ Intervall

Kante $\quad = \quad$ gemeinsamer Punkt

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

Satz:

maximale Überdeckungszahl

$=$ Größe der maximalen Klique

$=$ maximal nötige Anzahl Farben :-)

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 :-)