

(3) Differenzen für Mengen

Betrachte die Fixpunkt-Berechnung:

$$\begin{aligned} x &= \emptyset; \\ \text{for } (t = F x; t \not\subseteq x; &\boxed{t = F x;}) \\ x &= x \cup t; \end{aligned}$$

Ist F **distributiv**, könnte man sie ersetzen durch:

$$\begin{aligned} x &= \emptyset; \\ \text{for } (\Delta = F x; \Delta \neq \emptyset; &\boxed{\Delta = (F \Delta) \setminus x;}) \\ x &= x \cup \Delta; \end{aligned}$$

Die Funktion F muss jetzt nur noch für die **kleineren** Mengen Δ ausgerechnet werden :-)
semi-naive Iteration

Statt der Folge: $\emptyset \subseteq F(\emptyset) \subseteq F^2(\emptyset) \subseteq \dots$

berechnen wir: $\Delta_1 \cup \Delta_2 \cup \dots$

wobei:
$$\begin{aligned} \Delta_{i+1} &= F(F^i(\emptyset)) \setminus F^i(\emptyset) \\ &= F(\Delta_i) \setminus (\Delta_1 \cup \dots \cup \Delta_i) \quad \text{mit } \Delta_0 = \emptyset \end{aligned}$$

Nehmen wir an, die Kosten von $F x$ seien $1 + \#x$.

Dann summieren sich die Kosten zu:

naiv	$1 + 2 + \dots + n + n = \frac{1}{2}n(n + 3)$
semi-naiv	$2n$

wobei n die Kardinalität des Ergebnisses ist.

\implies Man spart einen linearen Faktor :-)

2.2 Peephole Optimierung

Idee:

- Schiebe ein **kleines** Fenster über das Programm.
- Optimiere aggressiv innerhalb des Fensters. D.h.:
 - Beseitige Redundanzen!
 - Ersetze innerhalb des Fensters teure Operationen durch billige!

Beispiele:

$$x = x + 1; \quad \Longrightarrow \quad x++;$$

// sofern es dafür eine spezielle Instruktion gibt :-)

$$z = y - a + a; \quad \Longrightarrow \quad z = y;$$

// algebraische Umformungen :-)

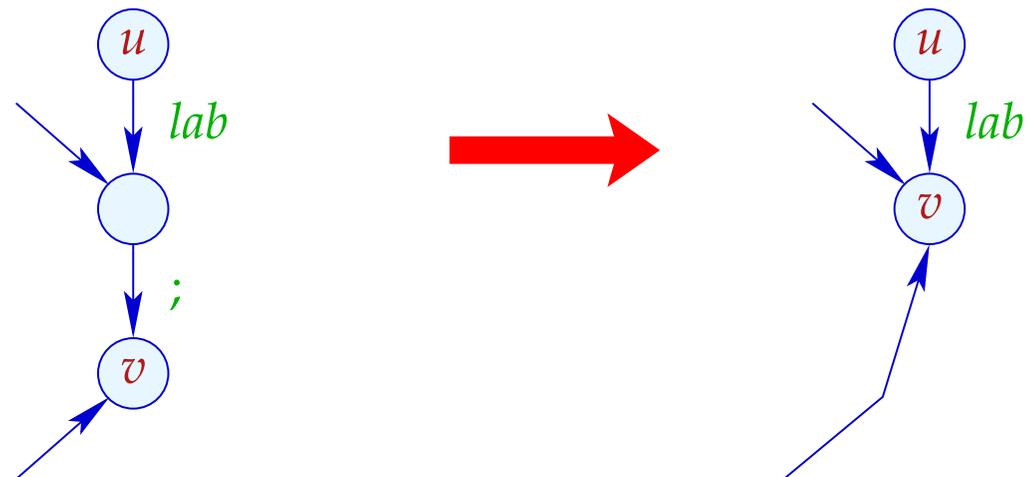
$$x = x; \quad \Longrightarrow \quad ;$$

$$x = 0; \quad \Longrightarrow \quad x = x \oplus x;$$

$$x = 2 \cdot x; \quad \Longrightarrow \quad x = x + x;$$

Wichtiges Teilproblem:

nop-Optimierung



- Ist $(v_1, ;, v)$ eine Kante, hat v_1 keine weitere ausgehende Kante.
- Folglich dürfen wir v_1 und v identifizieren :-)
- Die Reihenfolge der Identifizierungen ist egal :-))

Implementierung:

- Wir konstruieren eine Funktion $\text{next} : \text{Nodes} \rightarrow \text{Nodes}$ mit:

$$\text{next } u = \begin{cases} \text{next } v & \text{falls } (u, ;, v) \text{ Kante} \\ u & \text{sonst} \end{cases}$$

Achtung: Diese Definition ist nur rekursiv, wenn es ;-Schleifen gibt ???

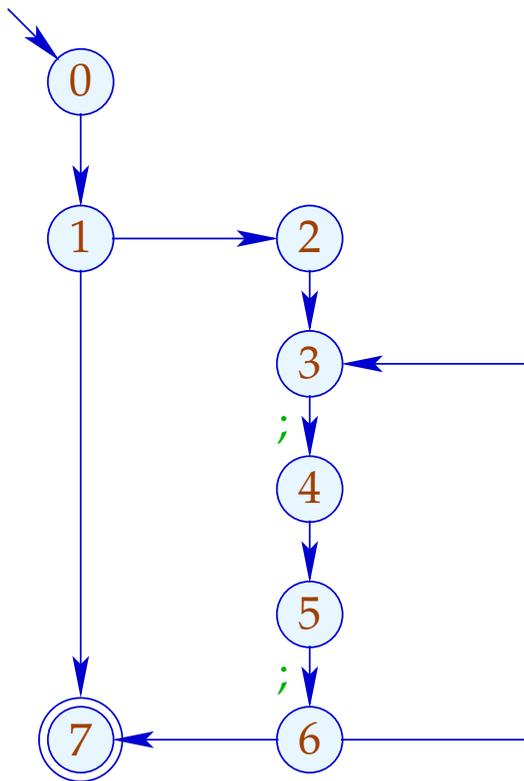
- Wir ersetzen jede Kante:

$$(u, \text{lab}, v) \implies (u, \text{lab}, \text{next } v)$$

... sofern $\text{lab} \neq ;$

- Alle ;-Kanten werfen wir weg ;-)

Beispiel:

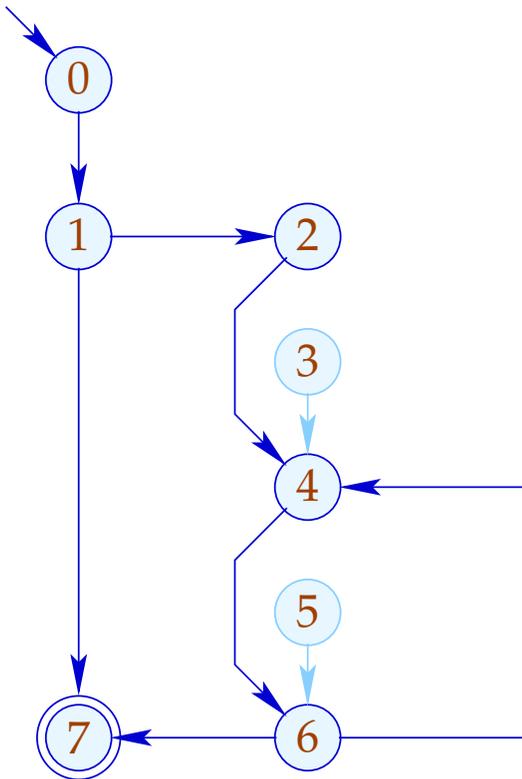


next 1 = 1

next 3 = 4

next 5 = 6

Beispiel:



next 1 = 1

next 3 = 4

next 5 = 6

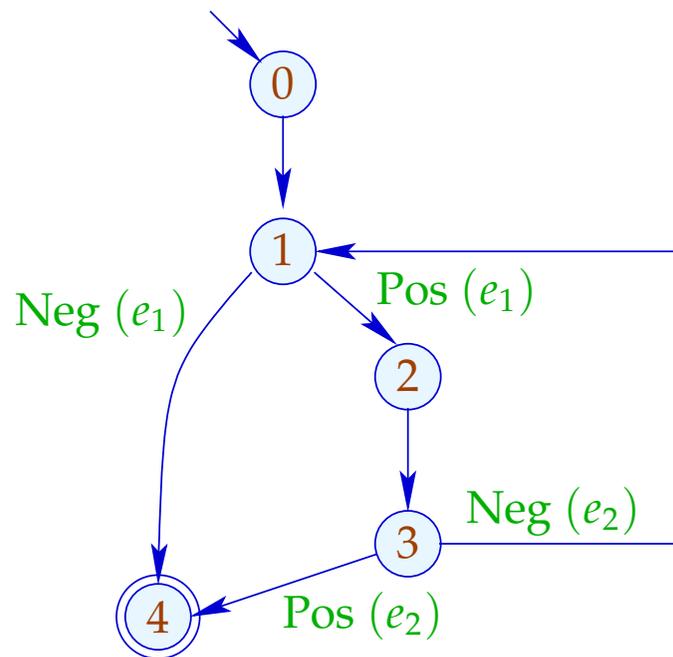
2. Teilproblem: Linearisierung

Der CFG muss nach der Optimierung wieder in eine **lineare Abfolge** von Instruktionen gebracht werden :-)

Achtung:

Nicht jede Linearisierung ist gleich gut !!!

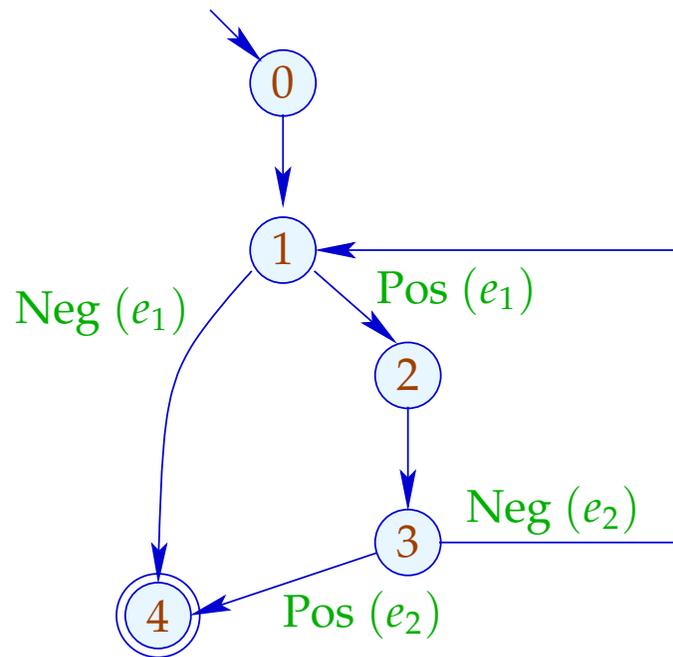
Beispiel:



0:
1: if (e_1) goto 2;
4: halt
2: Rumpf
3: if (e_2) goto 4;
goto 1;

Schlecht: Der Schleifen-Rumpf wird angesprungen :-)

Beispiel:



```
0:  
1:  if (!e1) goto 4;  
2:  Rumpf  
3:  if (!e2) goto 1;  
4:  halt
```

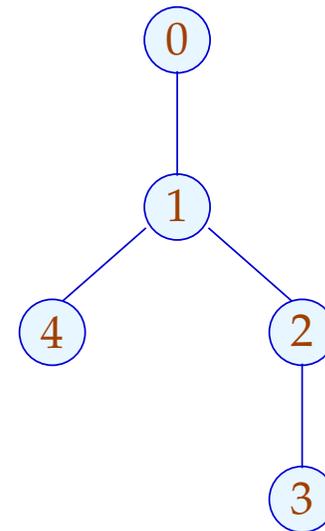
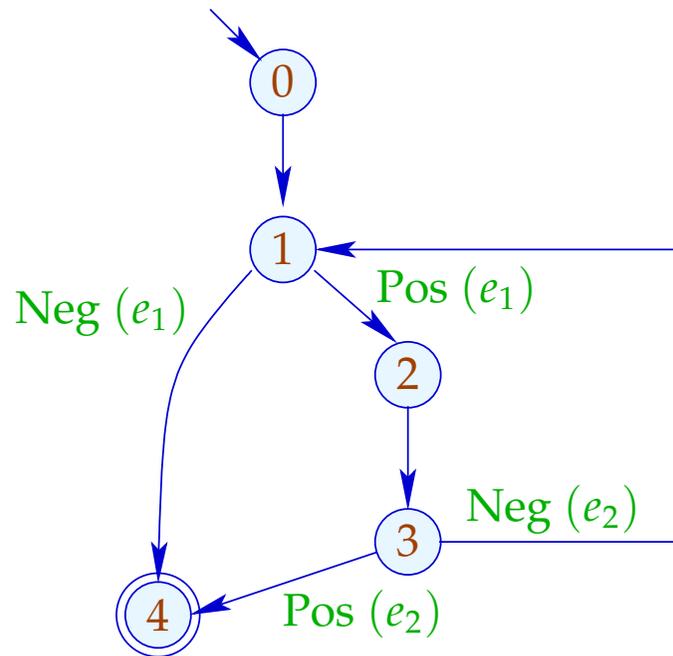
// besseres Cache-Verhalten :-)

Idee:

- Gib jedem Knoten eine **Temperatur!**
- Springe stets zu
 - (1) bereits behandelten Knoten;
 - (2) **kälteren** Knoten.
- **Temperatur** \approx Schachtelungstiefe

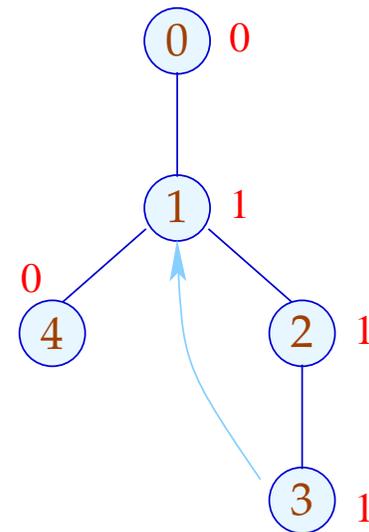
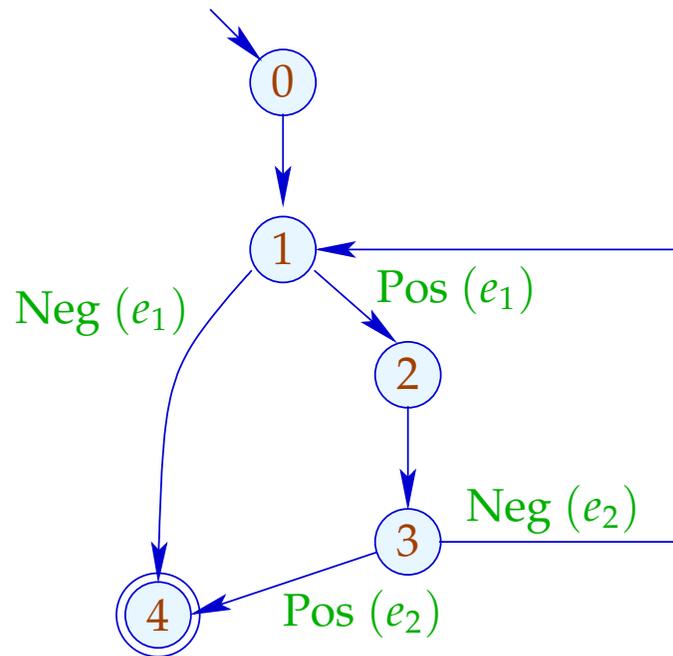
Zur Berechnung benutzen wir den Prädominator-Baum und starke Zusammenhangskomponenten ...

... im Beispiel:

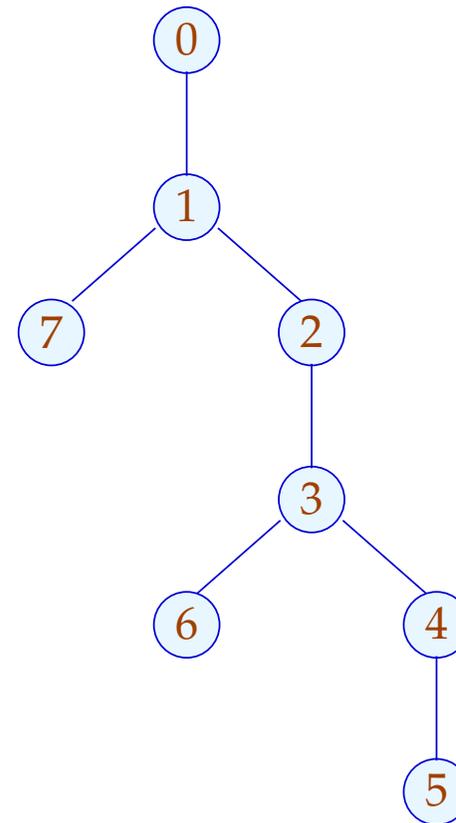
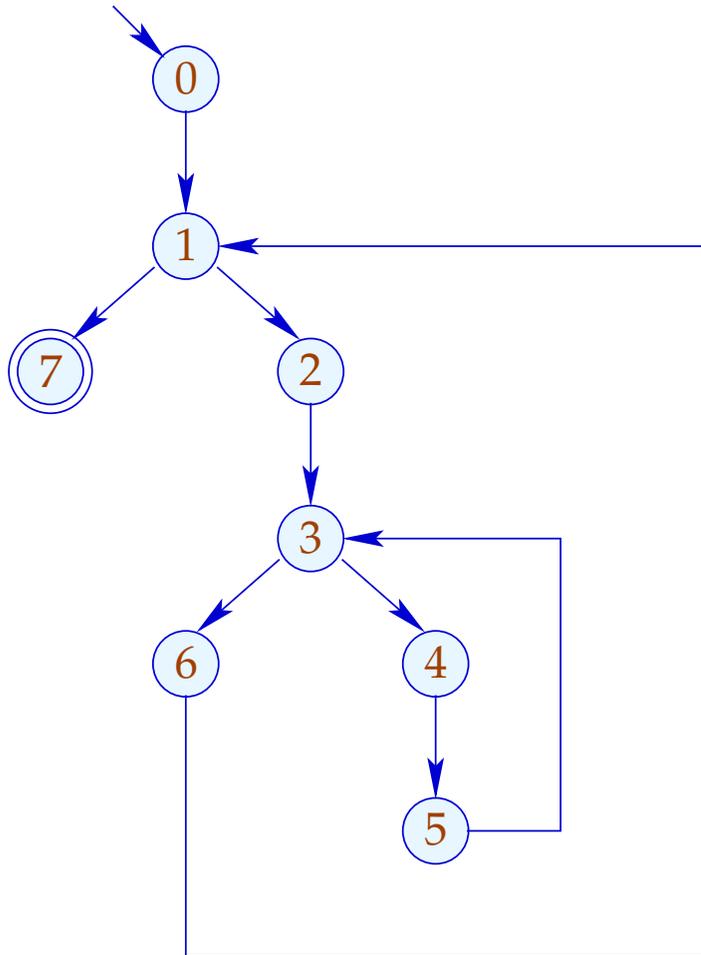


Der Teilbaum mit Rücksprung ist **heißer** ...

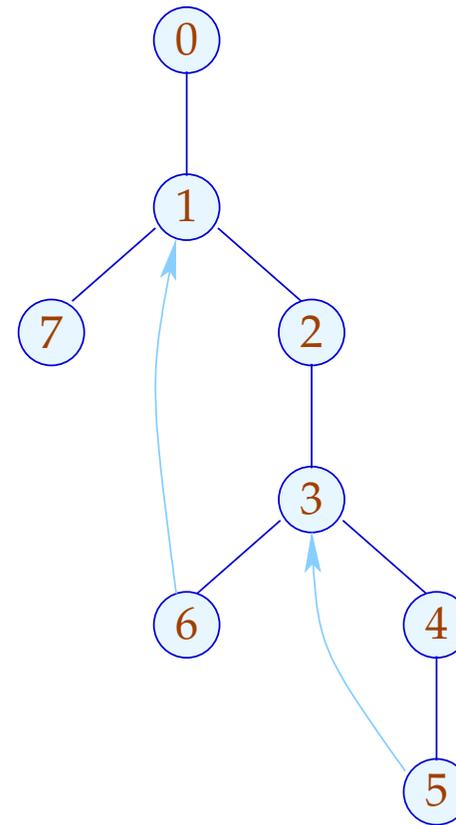
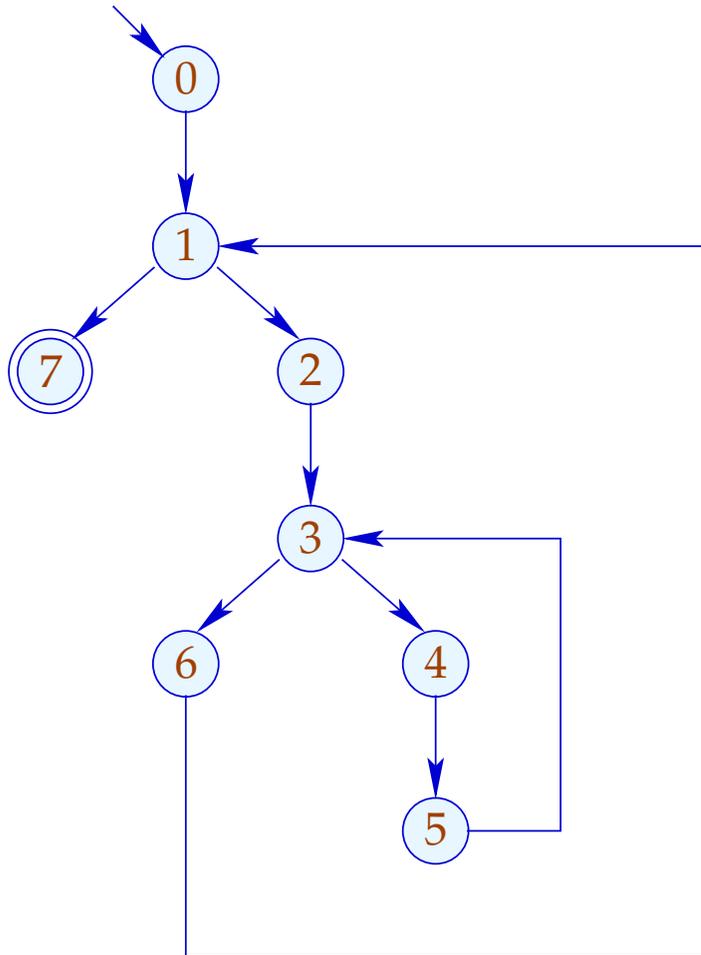
... im Beispiel:



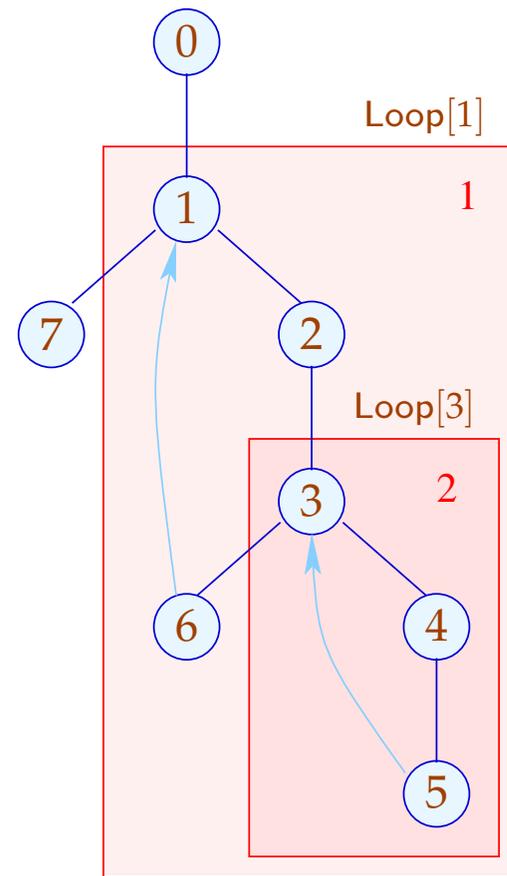
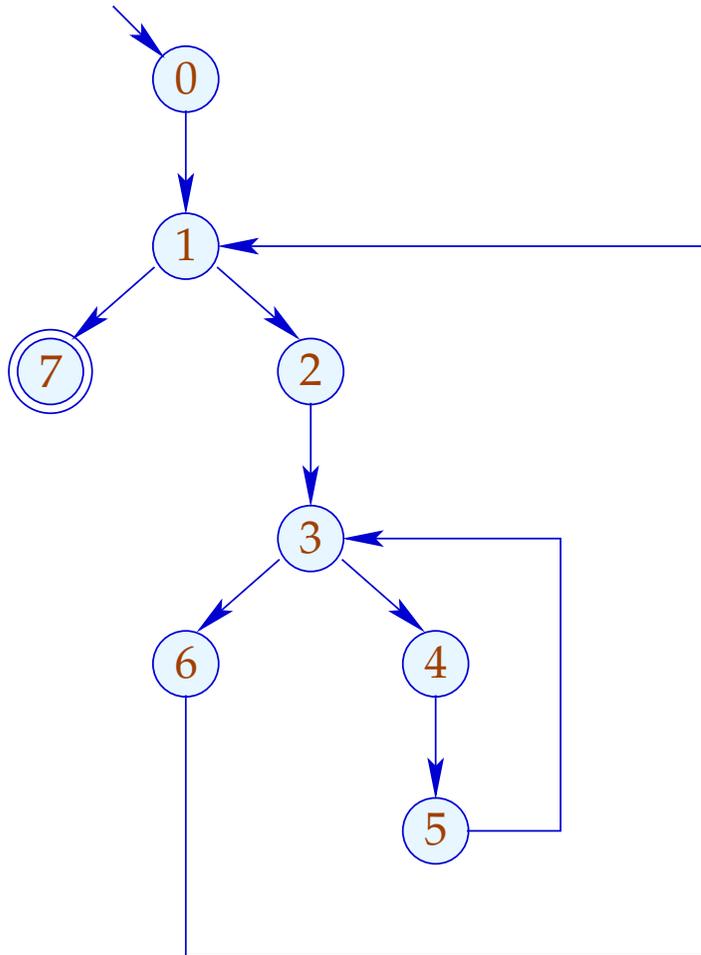
Komplizierteres Beispiel:



Komplizierteres Beispiel:

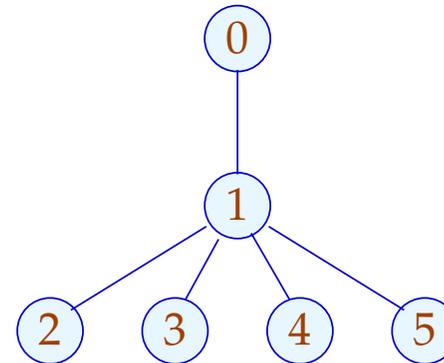
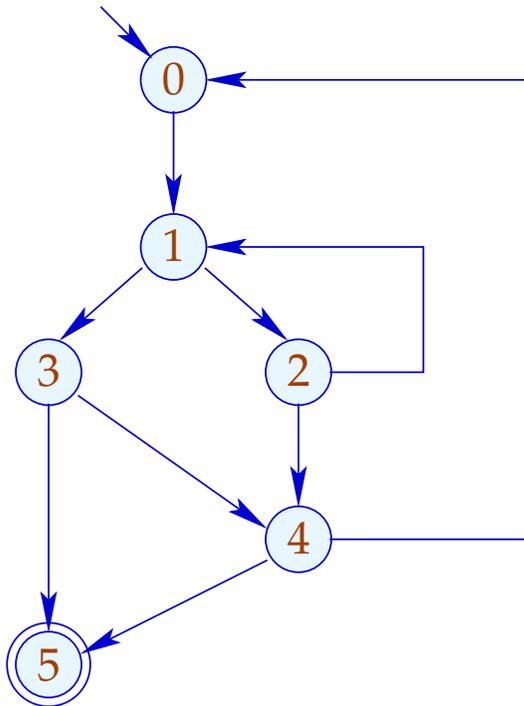


Komplizierteres Beispiel:



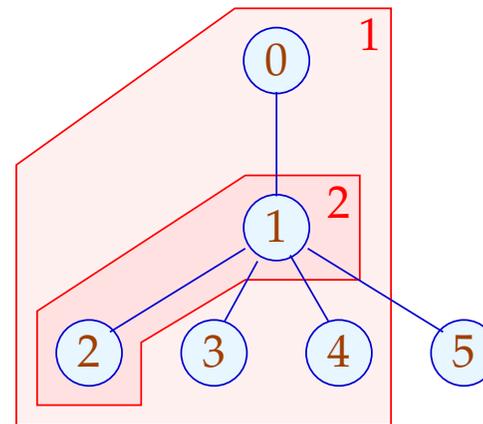
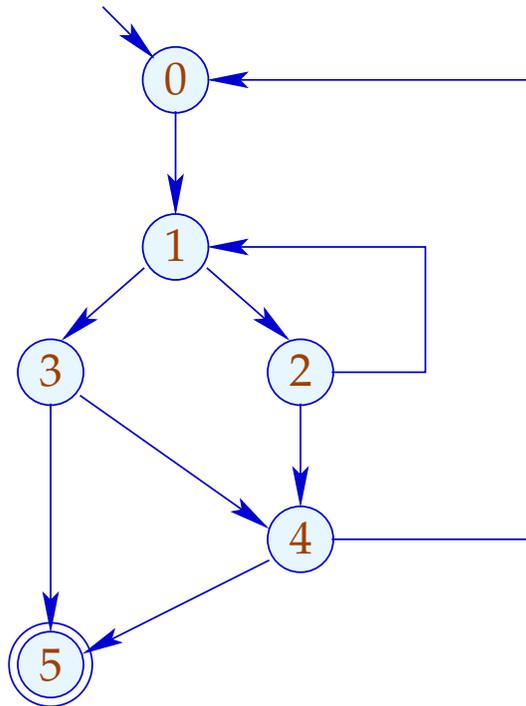
Unsere Definition von Loop sorgt dafür, dass (erkannte) Schleifen geschachtelt auftreten :-)

Sie ist auch für do-while-Schleifen mit breaks vernünftig...



Unsere Definition von Loop sorgt dafür, dass (erkannte) Schleifen geschachtelt auftreten :-)

Sie ist auch für do-while-Schleifen mit breaks vernünftig...



Zusammenfassung: Das Verfahren

- (1) Ermittlung einer Temperatur für jeden Knoten;
- (2) Prä-order-DFS über den CFG;
 - Führt eine Kante zu einem Knoten, für den wir bereits Code erzeugt haben, fügen wir einen Sprung ein.
 - Hat ein Knoten zwei Nachfolger unterschiedlicher Temperatur, fügen wir einen Sprung zum **kälteren** der beiden ein.
 - Hat ein Knoten zwei gleich warme Nachfolger, ist es egal ;-)

2.3 Funktionen

Wir erweitern unsere Mini-Programmiersprache um Funktionen und Funktions-Aufrufe. Dazu führen wir neue Statements ein:

```
ret =  $f(b_1, \dots, b_k)$ ;    return  $e$ ;
```

Jede Funktion f besitzt eine Definition:

```
 $f(a_1, \dots, a_n)$  {  $stmt^*$  }
```

```
//  $a_i$     formale Parameter
```

```
//  $b_i$     aktuelle Parameter
```

```
// ret    Register für Rückgabewert
```

Die Programm-Ausführung startet mit dem Aufruf einer (parameterlosen) Funktion `main()`.

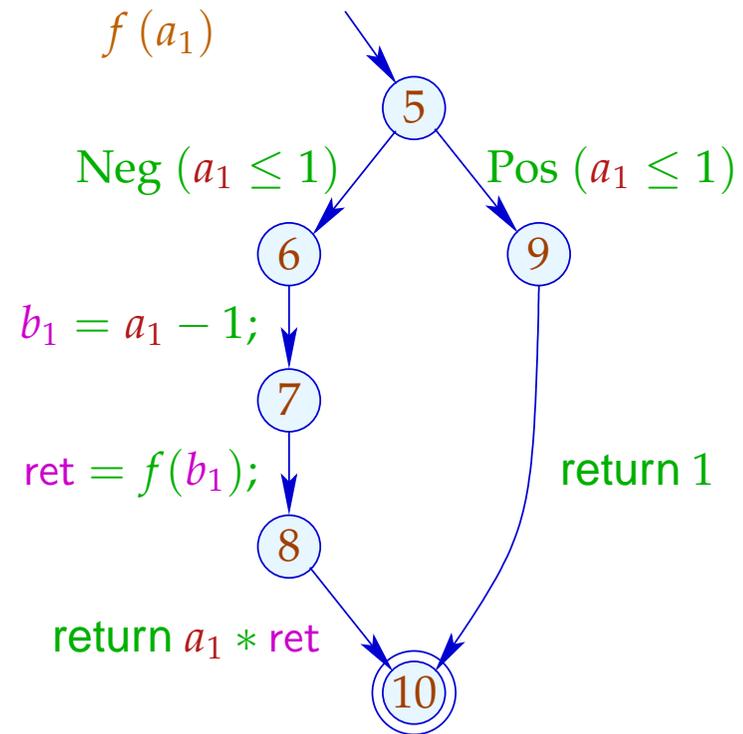
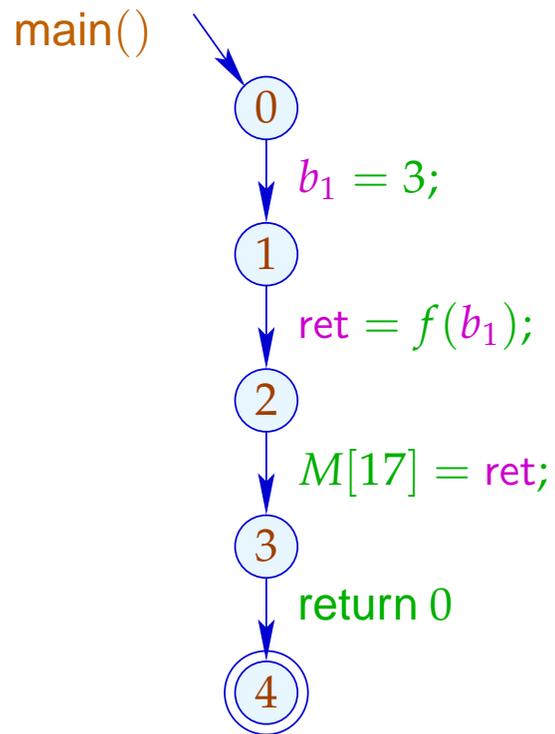
Beispiel:

```
main () {
     $b_1 = 2$ ;
     $ret = f(b_1)$ ;
     $M[17] = ret$ ;
    return 0;
}

f(a1) {
    if ( $a_1 \leq 1$ ) return 1;
     $b_1 = a_1 - 1$ ;
     $ret = f(b_1)$ ;
    return  $a_1 \cdot ret$ ;
}
```

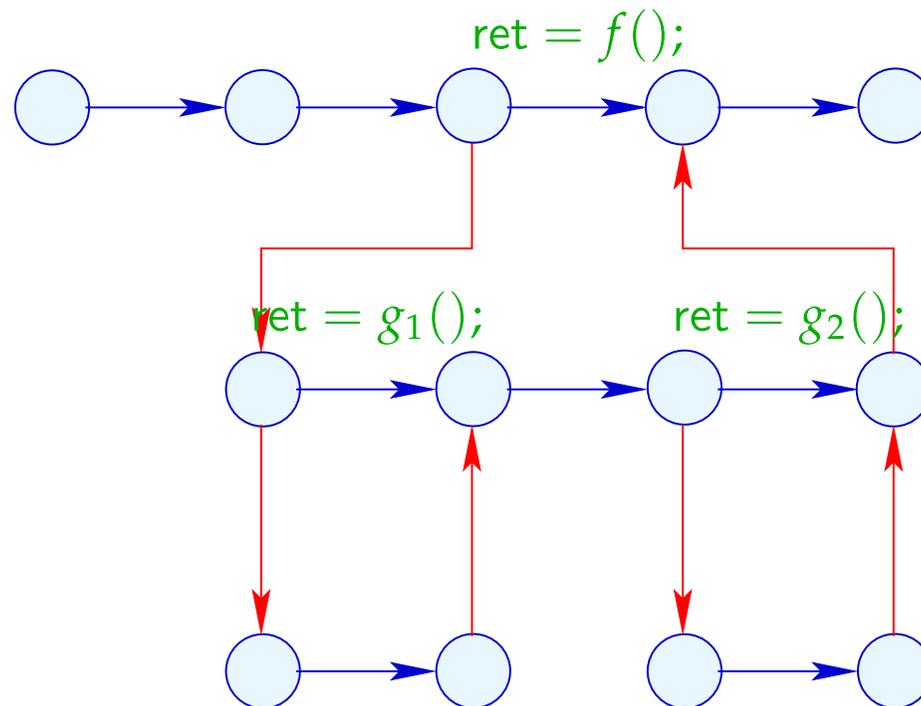
Solche Programme lassen sich durch eine **Menge** von CFGs darstellen: einem für jede Funktion ...

... im Beispiel:

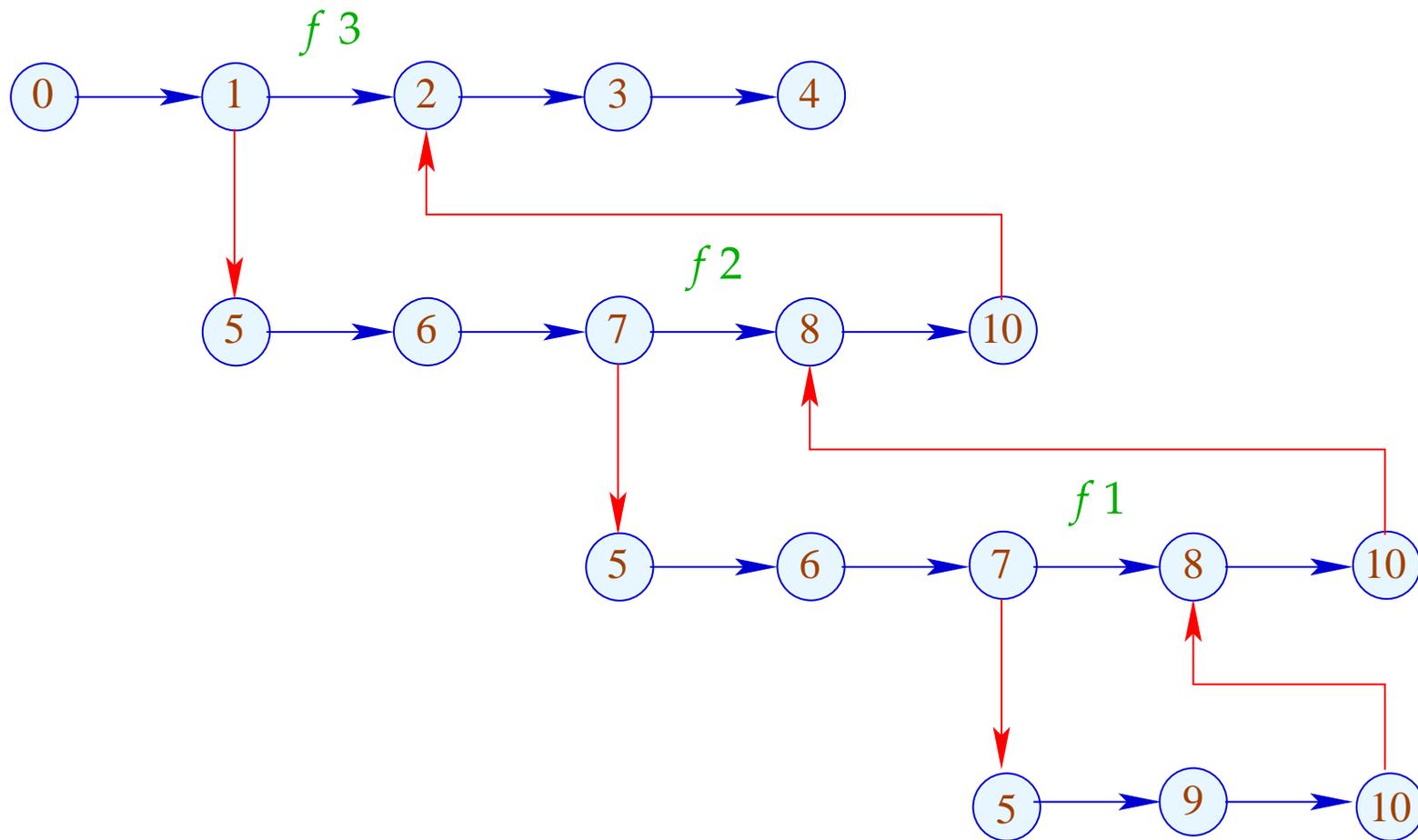


Um solche Programme zu optimieren, benötigen wir eine erweiterte operationelle Semantik ;-))

Programm-Ausführungen sind nicht mehr **Pfade**, sondern **Wälder**:



... im Beispiel:



Die Funktion $\llbracket \cdot \rrbracket$ erweitern wir auf Berechnungs-Wälder w :

$$\llbracket w \rrbracket : (\text{Vars} \rightarrow \mathbb{Z}) \times (\mathbb{N} \rightarrow \mathbb{Z}) \rightarrow (\text{Vars} \rightarrow \mathbb{Z}) \times (\mathbb{N} \rightarrow \mathbb{Z})$$

Für einen Aufruf $k = (u, \text{ret} = f(b_1, \dots, b_k);, v)$ müssen wir:

- die Anfangswerte der lokalen Variablen ermitteln:

$$\text{enter } \rho \ x = \begin{cases} \rho \ b_i & \text{falls } x = a_i \\ 0 & \text{sonst} \end{cases}$$

- ... den berechneten Rückgabe-Wert in ret ablegen:

$$\text{combine } (\rho_1, \rho_2) = \rho_1 \oplus \{\text{ret} \rightarrow \rho_2 \text{ret}\}$$

- ... dazwischen den Berechnungs-Wald der Funktion auswerten:

$$\begin{aligned} \llbracket k \langle w \rangle \rrbracket (\rho, \mu) &= \\ &\text{let } (\rho_1, \mu_1) = \llbracket w \rrbracket (\text{enter } \rho, \mu) \\ &\text{in } (\text{combine } (\rho, \rho_1), \mu_1) \end{aligned}$$

Ein Return $k = (u, \text{return } e; v)$ ist eine Zuweisung an ret :

$$\llbracket k \rrbracket (\rho, \mu) = (\rho \oplus \{\text{ret} \mapsto \llbracket e \rrbracket \rho\}, \mu)$$

Achtung:

- $\llbracket w \rrbracket$ ist i.a. nur partiell definiert :-)
- Die Benutzung von speziellen Registern a_i, b_i, ret repräsentiert eine bestimmte Aufruf-Konvention.
- Die **normale** operationelle Semantik arbeitet mit Konfigurationen, die **Aufrufkeller** verwalten.
- Berechnungs-Wälder eignen sich aber besser zur Konstruktion von Analysen und Korrektheitsbeweisen :-)
- Es ist eine lästige (aber nützliche) Aufgabe, die Äquivalenz der beiden Ansätze zu zeigen ...

Konfigurationen:

$$\begin{aligned} \text{configuration} & \quad \equiv \quad \text{stack} \times \text{store} \\ \text{store} & \quad \equiv \quad \mathbb{N} \rightarrow \mathbb{Z} \\ \text{stack} & \quad \equiv \quad \text{frame} \cdot \text{frame}^* \\ \text{frame} & \quad \equiv \quad \text{point} \times \text{locals} \\ \text{locals} & \quad \equiv \quad (\text{Vars} \rightarrow \mathbb{Z}) \end{aligned}$$

Ein *frame* (Kellerrahmen) beschreibt den lokalen Berechnungszustand innerhalb eines Funktionsaufrufs :-)

Den Rahmen des aktuellen Aufrufs schreiben wir **links**.

Berechnungsschritte beziehen sich auf den aktuellen Aufruf :-)

Zusätzlich benötigte Arten von Schritten:

Aufruf $k = (u, \text{ret} = f(b_1, \dots, b_k);, v) :$

$$\left((u, \rho) \cdot \sigma, \mu \right) \implies \left((u_f, \text{enter } \rho) \cdot (v, \rho) \cdot \sigma, \mu \right)$$

u_f Anfangspunkt von f

Rückkehr:

$$\left((r_f, \rho_2) \cdot (v, \rho_1) \cdot \sigma, \mu \right) \implies \left((v, \text{combine}(\rho_1, \rho_2)) \cdot \sigma, \mu \right)$$

r_f Endpunkt von f

Rückgabe: $k = (u, \text{return } e; , v) :$

$$\left((u, \rho) \cdot \sigma, \mu \right) \implies \left((v, \rho \oplus \{ \text{ret} \mapsto \llbracket e \rrbracket \rho \}) \cdot \sigma, \mu \right)$$

Mit dem Aufruf-Keller verwalten wir explizit den DFS-Durchlauf über den Berechnungswald :-)

... im Beispiel:

1	$b_1 \mapsto 3$
---	-----------------

Mit dem Aufruf-Keller verwalten wir explizit den DFS-Durchlauf über den Berechnungswald :-)

... im Beispiel:

5	$a_1 \mapsto 3$
2	

Mit dem Aufruf-Keller verwalten wir explizit den DFS-Durchlauf über den Berechnungswald :-)

... im Beispiel:

7	$a_1 \mapsto 3$
2	

Mit dem Aufruf-Keller verwalten wir explizit den DFS-Durchlauf über den Berechnungswald :-)

... im Beispiel:

5	$a_1 \mapsto 2$
8	$a_1 \mapsto 3$
2	

Mit dem Aufruf-Keller verwalten wir explizit den DFS-Durchlauf über den Berechnungswald :-)

... im Beispiel:

7	$a_1 \mapsto 2$
8	$a_1 \mapsto 3$
2	

Mit dem Aufruf-Keller verwalten wir explizit den DFS-Durchlauf über den Berechnungswald :-)

... im Beispiel:

5	$a_1 \mapsto 1$
8	$a_1 \mapsto 2$
8	$a_1 \mapsto 3$
2	

Mit dem Aufruf-Keller verwalten wir explizit den DFS-Durchlauf über den Berechnungswald :-)

... im Beispiel:

10	ret \mapsto 1
8	$a_1 \mapsto$ 2
8	$a_1 \mapsto$ 3
2	

Mit dem Aufruf-Keller verwalten wir explizit den DFS-Durchlauf über den Berechnungswald :-)

... im Beispiel:

8	ret \mapsto 1
8	$a_1 \mapsto$ 3
2	

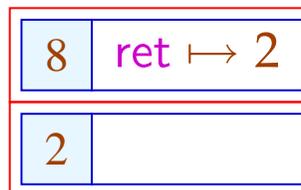
Mit dem Aufruf-Keller verwalten wir explizit den DFS-Durchlauf über den Berechnungswald :-)

... im Beispiel:

10	ret \mapsto 2
8	$a_1 \mapsto$ 3
2	

Mit dem Aufruf-Keller verwalten wir explizit den DFS-Durchlauf über den Berechnungswald :-)

... im Beispiel:



Mit dem Aufruf-Keller verwalten wir explizit den DFS-Durchlauf über den Berechnungswald :-)

... im Beispiel:

10	ret \mapsto 6
2	

Mit dem Aufruf-Keller verwalten wir explizit den DFS-Durchlauf über den Berechnungswald :-)

... im Beispiel:

2	ret \mapsto 6
---	-----------------