2.2 Peephole Optimierung

Idee:

- Schiebe ein kleines Fenster über das Programm.
- Optimiere agressiv innerhalb des Fensters. D.h.:

  → Beseitige Redundanzen!

  → Ersetze innerhalb des Fensters teure Operationen durch billige!

Beispiele:

$x = x + 1;$ $\Longrightarrow$ $x{+}{+};$

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

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

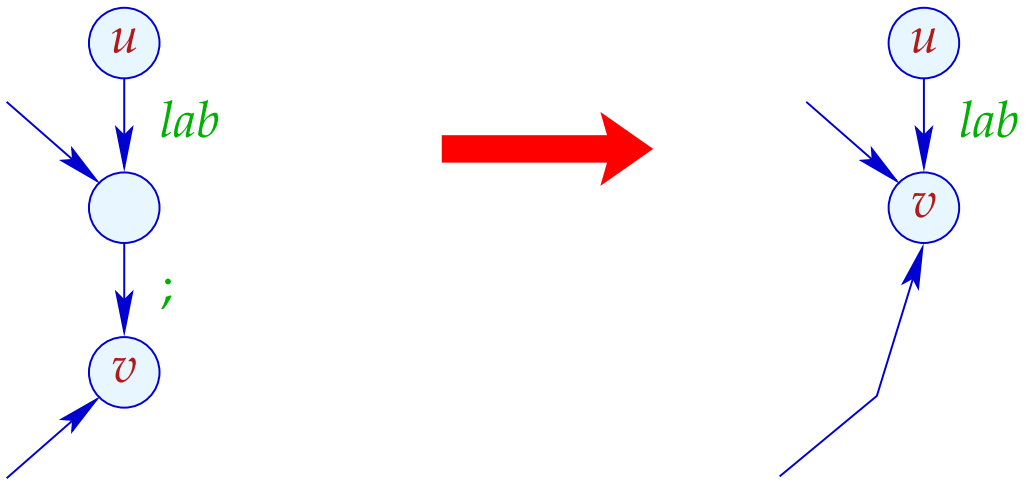
// algebraische Umformungen :-)

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

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

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

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 $\mathsf{next} : \mathit{Nodes} \to \mathit{Nodes}$ mit:

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

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

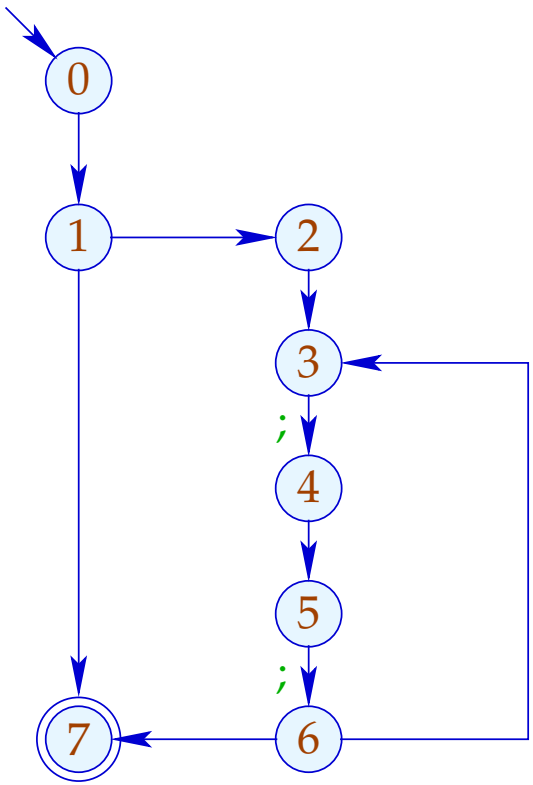
- Wir ersetzen jede Kante:

$$(u, lab, v) \quad \Longrightarrow \quad (u, lab, \mathsf{next}\ v)$$

  ... sofern $lab \neq ;$

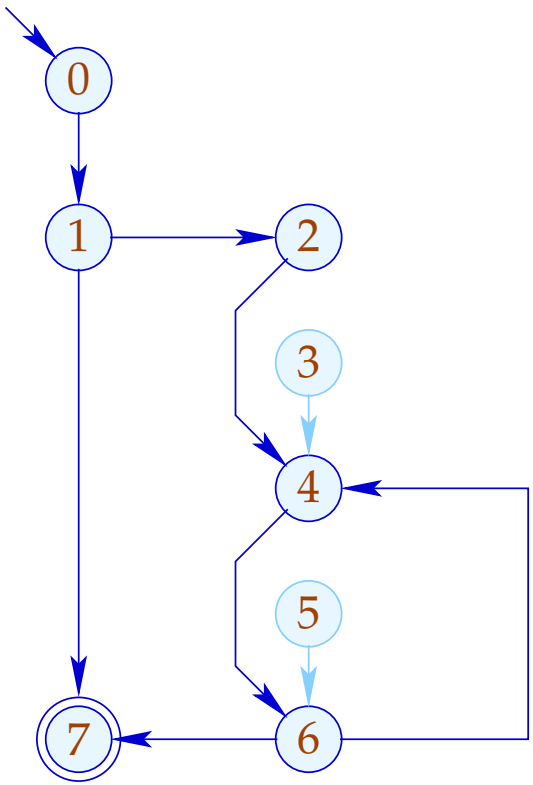
- Alle ;-Kanten werfen wir weg ;-)

Beispiel:



$$\begin{aligned} \text{next } 1 &= 1 \\ \text{next } 3 &= 4 \\ \text{next } 5 &= 6 \end{aligned}$$

Beispiel:



$$\begin{aligned} \text{next } 1 &= 1 \\ \text{next } 3 &= 4 \\ \text{next } 5 &= 6 \end{aligned}$$

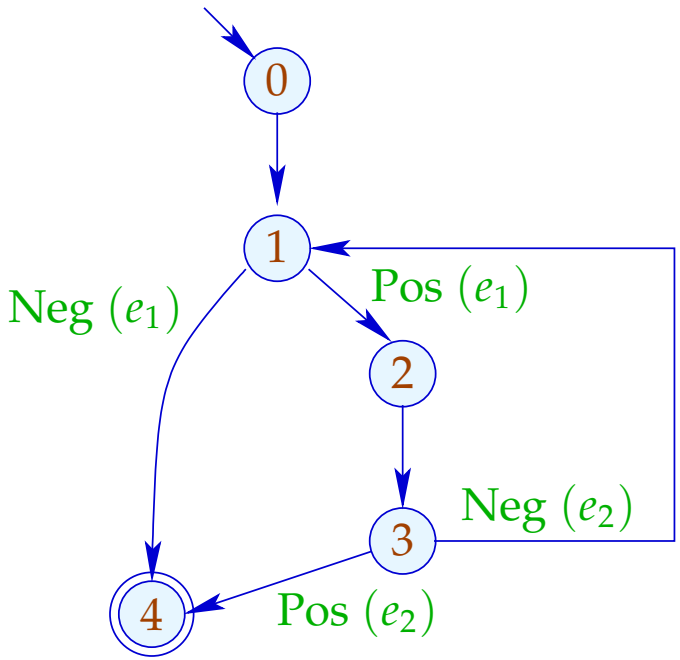
2. Teilproblem: Linearisierung

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

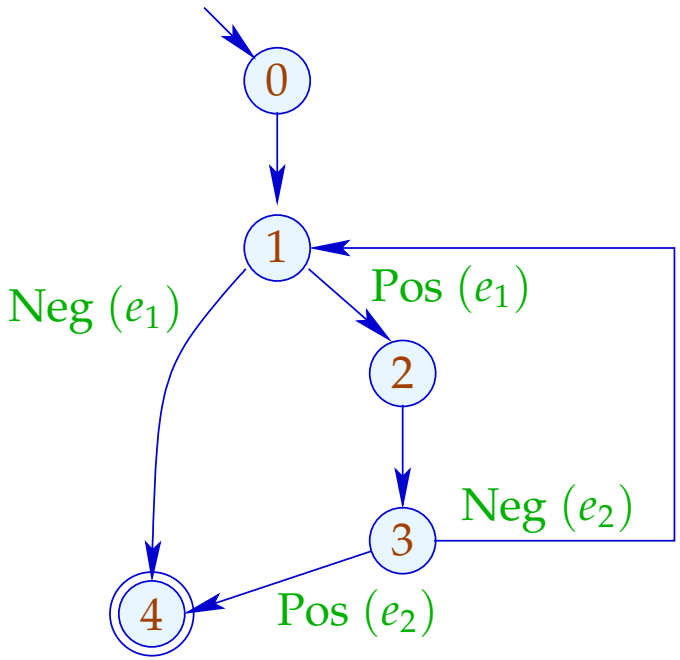
2: Rumpf

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
```

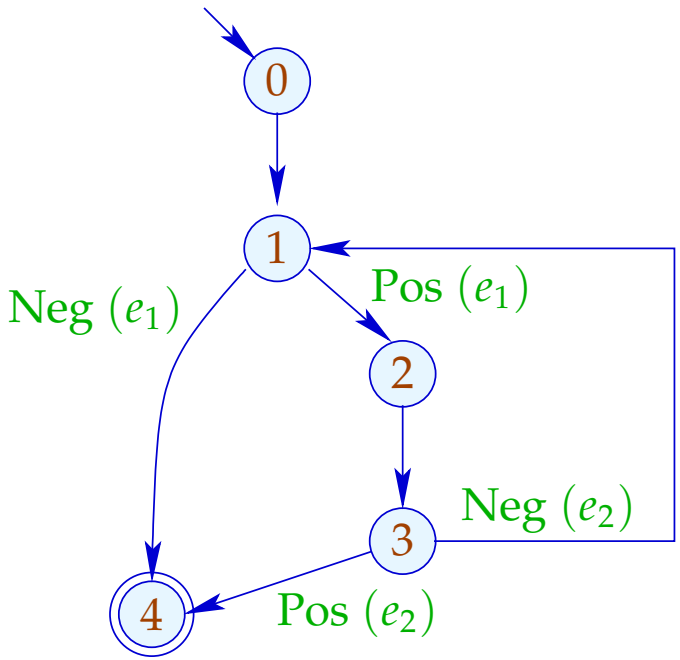
// besseres Cache-Verhalten :-)

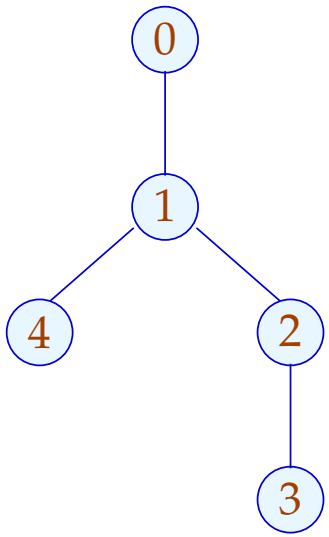
Idee:

- 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 ...

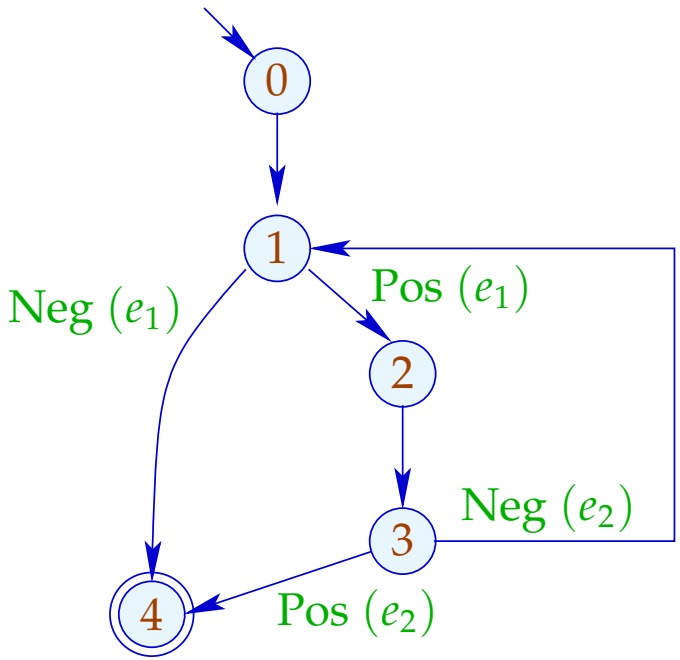
... im Beispiel:

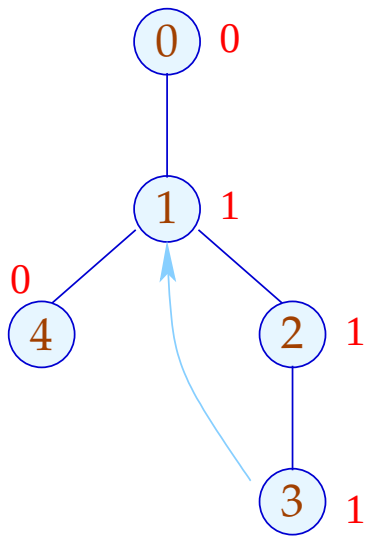




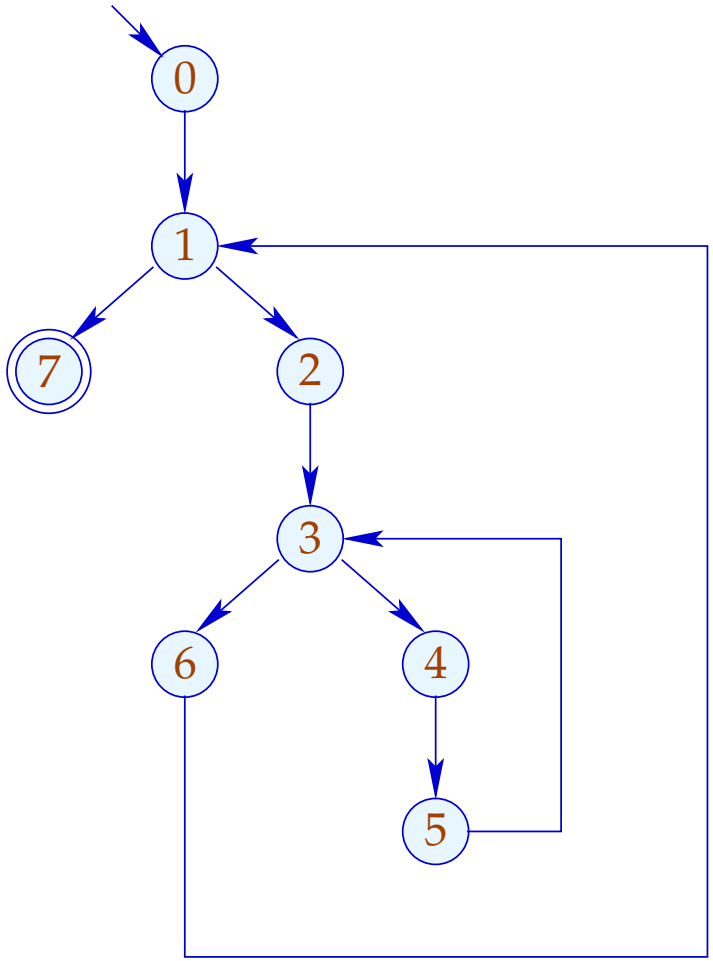
Der Teilbaum mit Rücksprung ist heißer ...

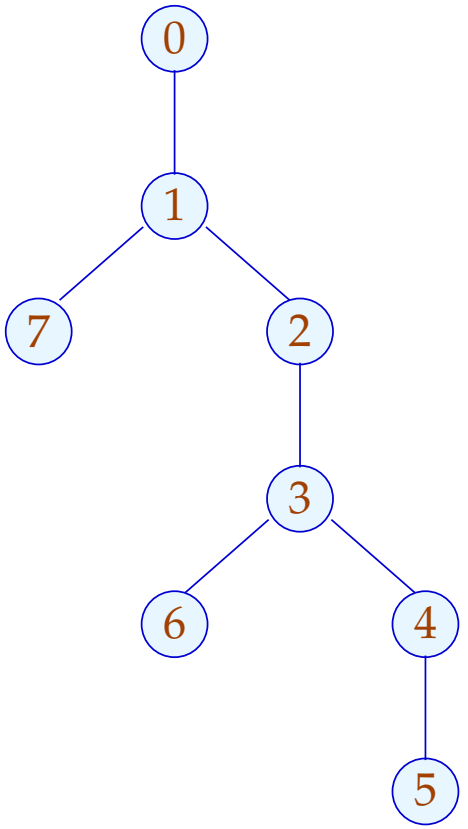
... im Beispiel:



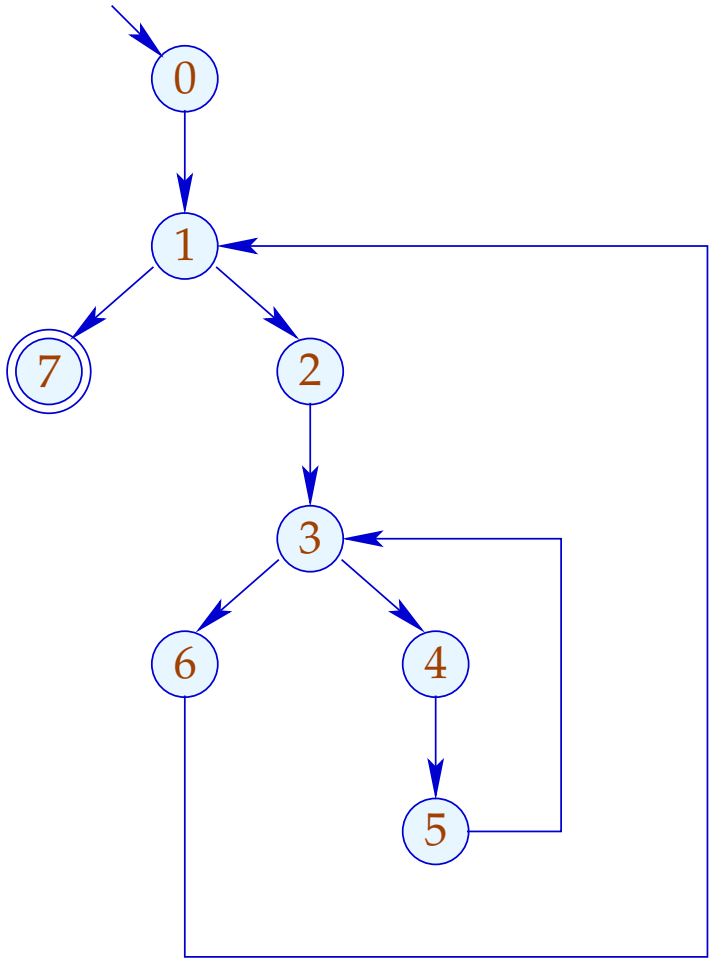


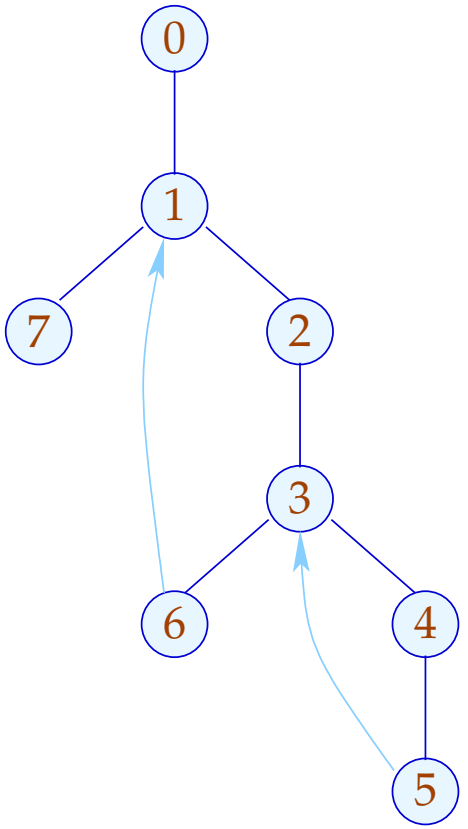
Komplizierteres Beispiel:



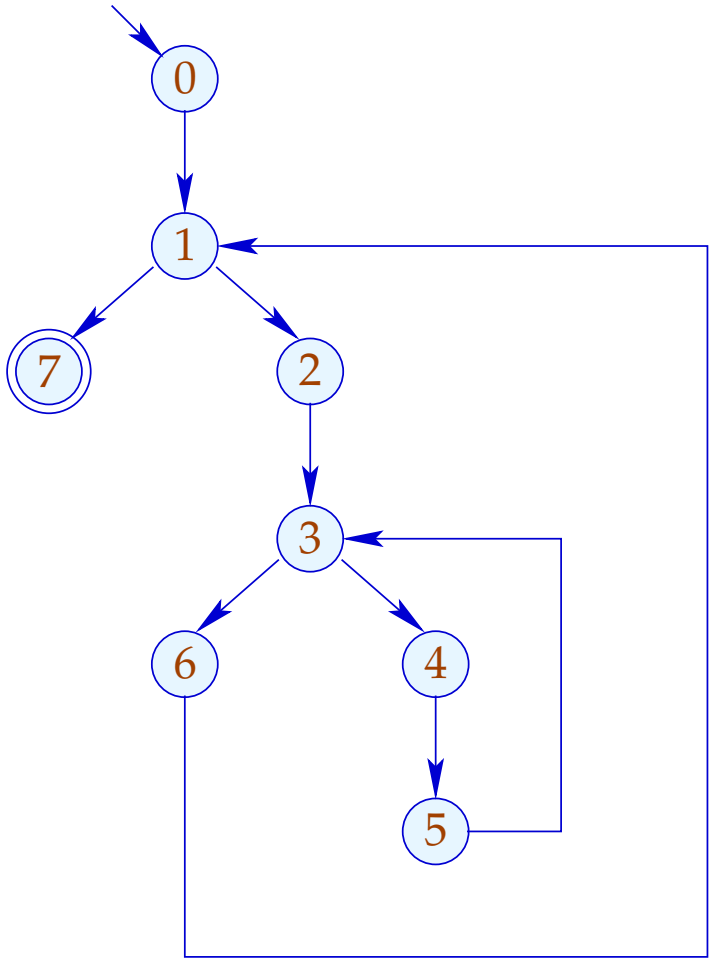


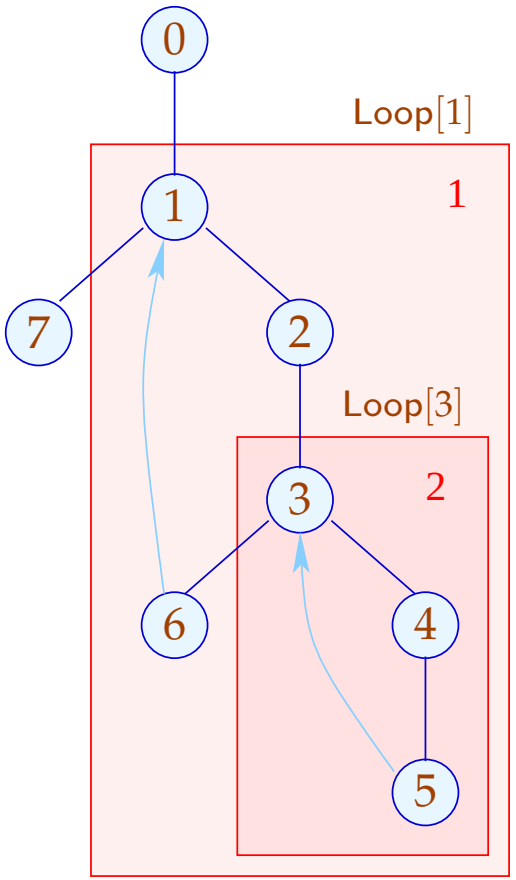
Komplizierteres Beispiel:





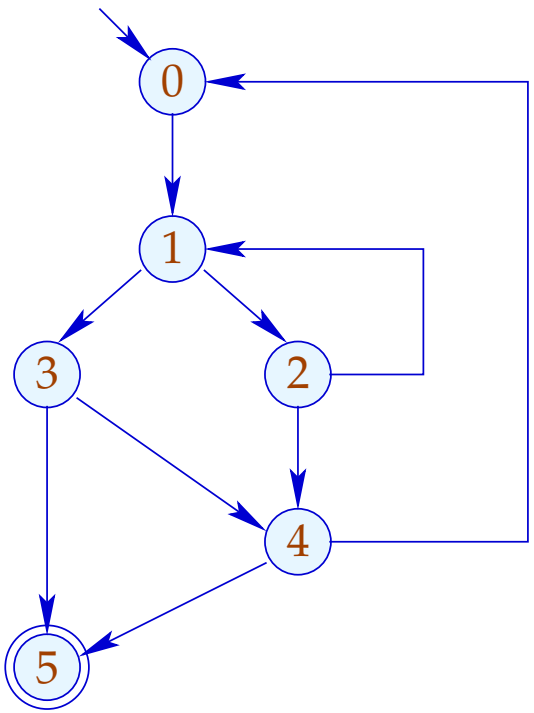
Komplizierteres Beispiel:

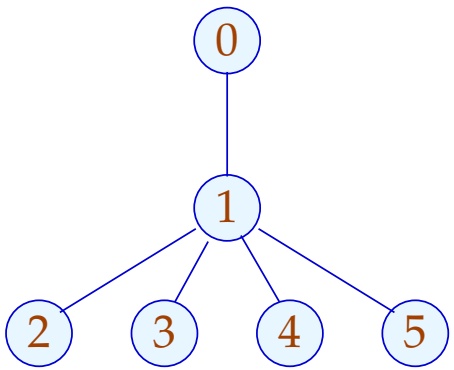




Unsere Definition von `Loop` sorgt dafür, dass (erkannte) Schleifen geschachtelt auftreten :-)

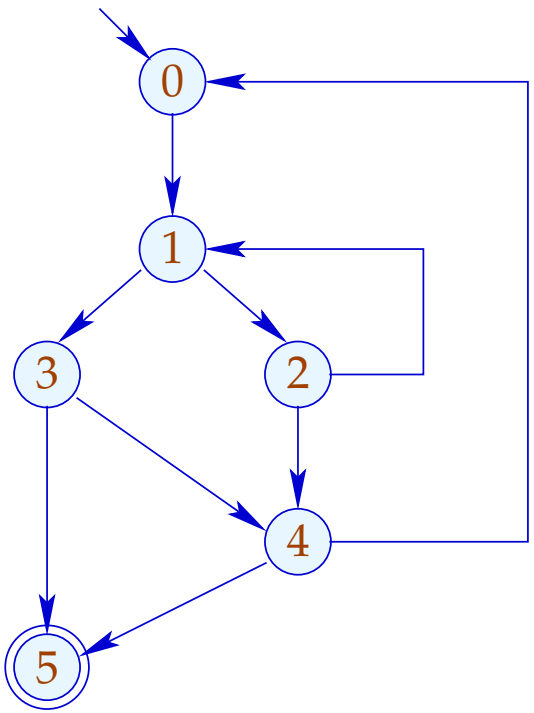
Sie ist auch für `do-while`-Schleifen mit `breaks` vernünftig...

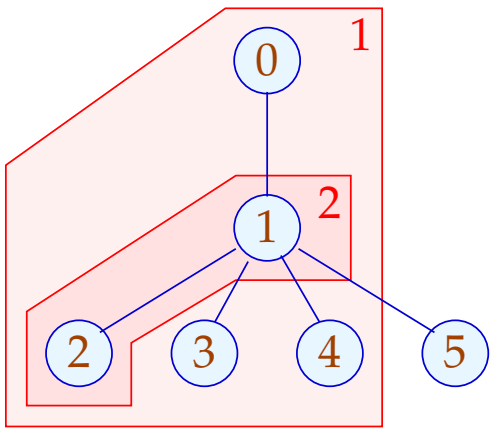




Unsere Definition von Loop sorgt dafür, dass (erkannte) Schleifen geschachtelt auftreten :-)

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 Prozeduren

Wir erweitern unsere Mini-Programmiersprache um Prozeduren ohne Parameter und Prozedur-Aufrufe.

Dazu führen wir als neues Statement ein:

$$f\,();$$

Jede Prozedur f besitzt eine Definition:

$$f\,()\ \{\ stmt^*\ \}$$

Dabei unterscheiden wir jetzt globale von lokalen Variablen.

Die Programm-Ausführung startet mit dem Aufruf einer Prozedur main () .

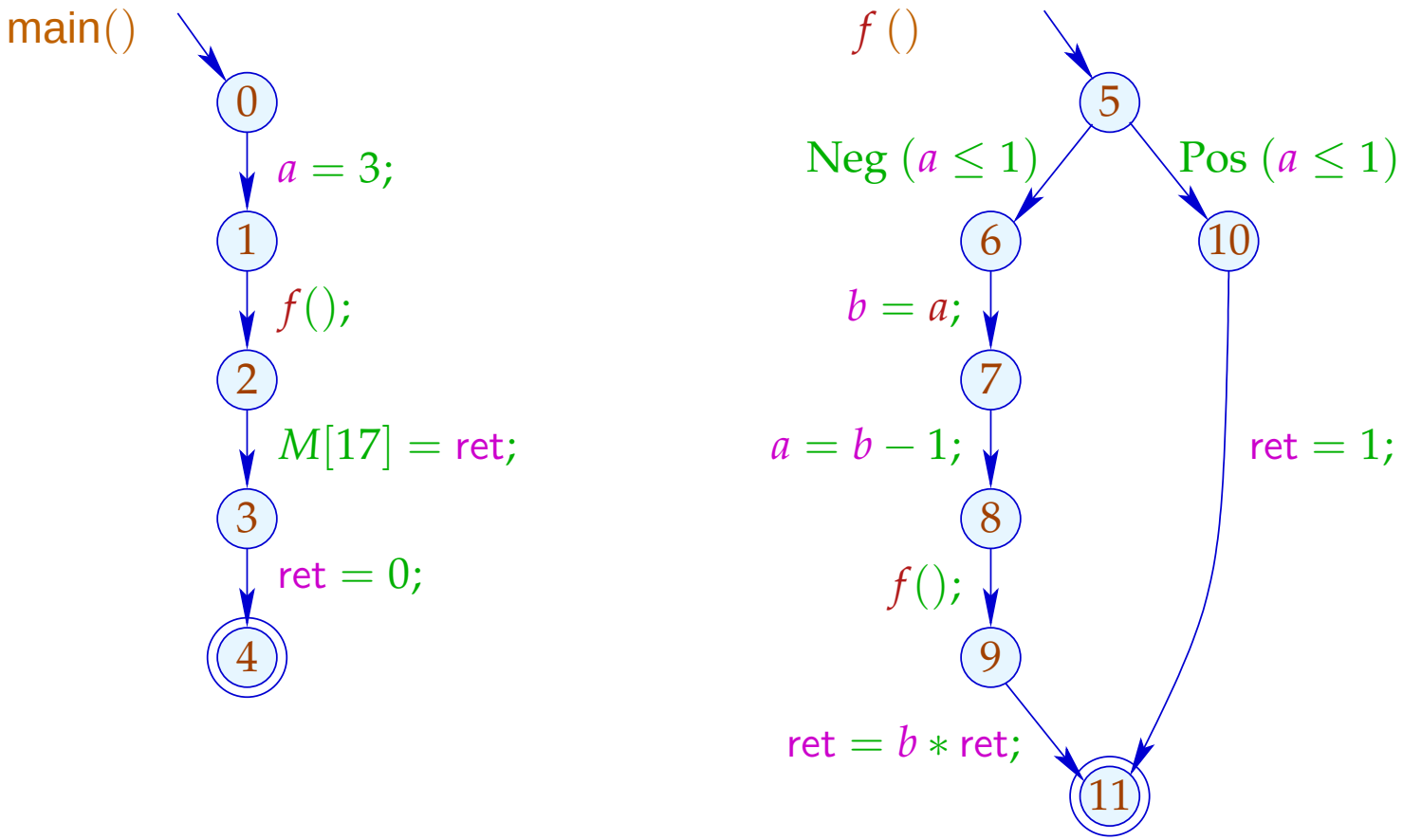
Beispiel:

```
int a, ret;
main () {
        a = 3;
        f();
        M[17] = ret;
        ret = 0;
}
```

```
f () {
        int b;
        if (a ≤ 1) ret = 1;
           b = a;
           a = b − 1;
        f();
        ret = b · ret;
}
```

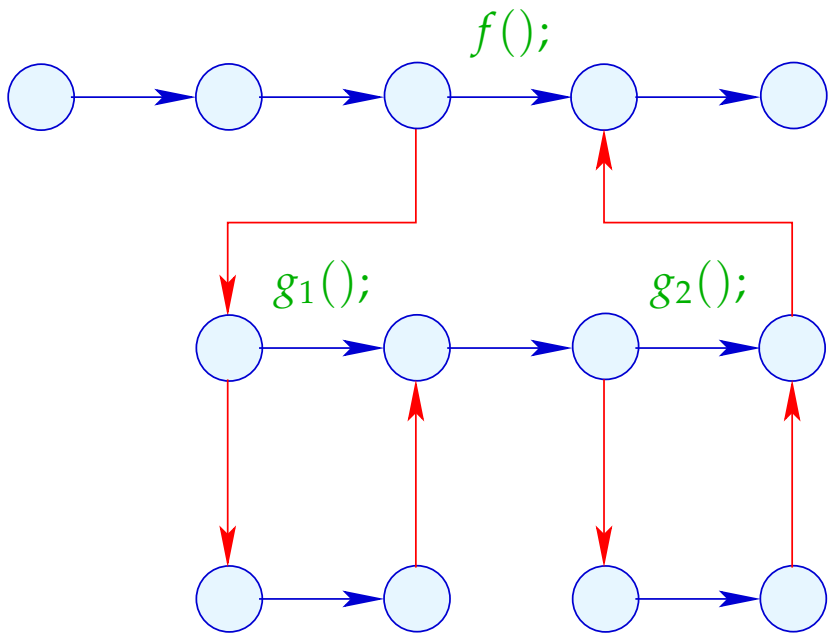
Solche Programme lassen sich durch eine Menge von CFGs darstellen: einem für jede Prozedur ...

... im Beispiel:

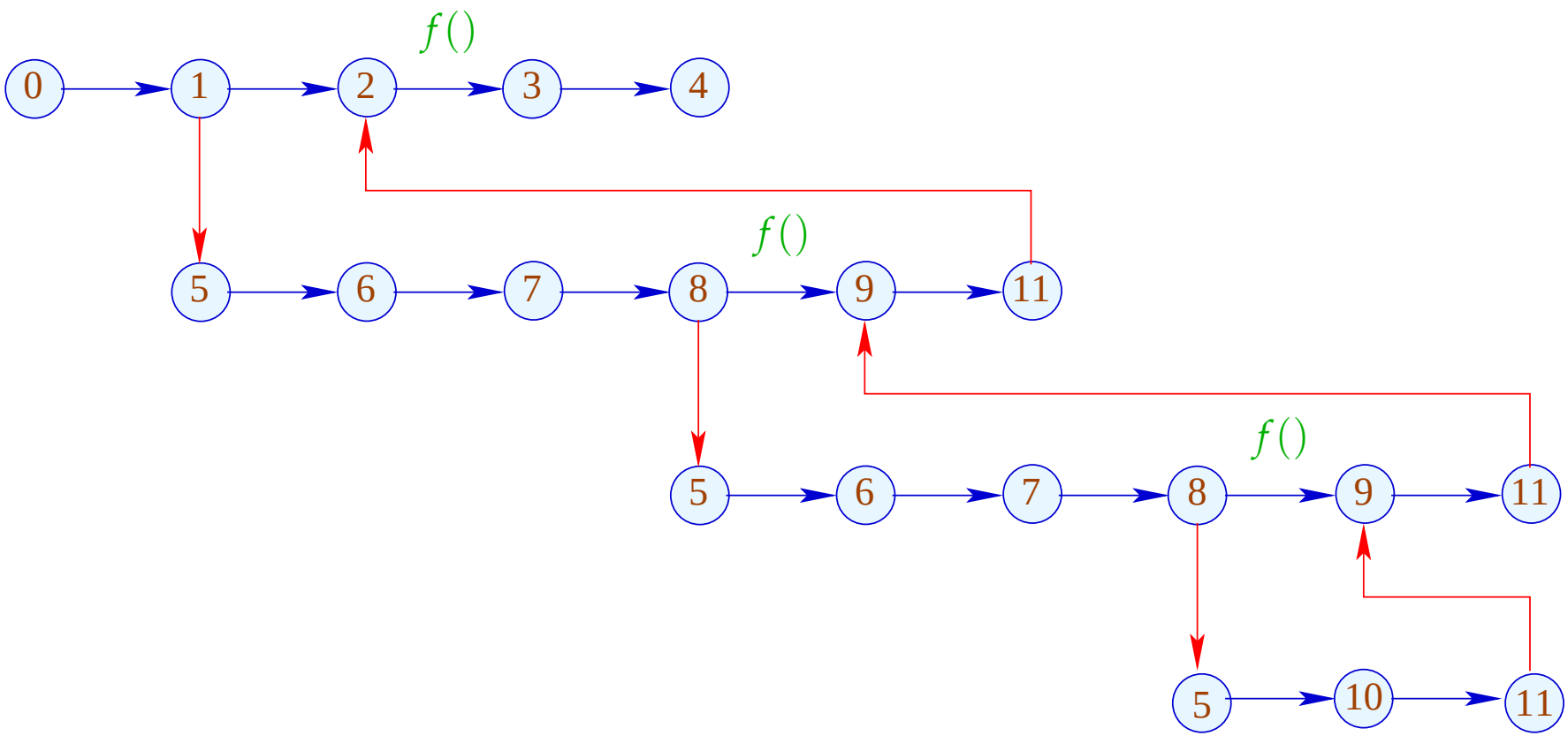

main()
0
$a = 3;$
1
$f();$
2
$M[17] = \mathsf{ret};$
3
$\mathsf{ret} = 0;$
4
$f\,()$
5
Neg $(a \leq 1)$
Pos $(a \leq 1)$
6
10
$b = a;$
7
$a = b - 1;$
$\mathsf{ret} = 1;$
8
$f();$
9
$\mathsf{ret} = b * \mathsf{ret};$
11

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

Programm-Ausführungen sind nicht mehr Pfade, sondern Wälder:



... im Beispiel:



Die Funktion $[\![.]\!]$ erweitern wir auf Berechnungs-Wälder w :

$$[\![w]\!] : (\mathit{Vars} \to \mathbb{Z}) \times (\mathbb{N} \to \mathbb{Z}) \to (\mathit{Vars} \to \mathbb{Z}) \times (\mathbb{N} \to \mathbb{Z})$$

Für einen Aufruf $k = (u, f();, v)$ müssen wir:

- die Anfangwerte der neuen lokalen Variablen ermitteln:

$$\mathsf{enter}\ \rho = \{x \mapsto 0 \mid x \in \mathit{Locals}\} \oplus (\rho|_{\mathit{Globals}})$$

- ... die neu berechneten Werte für die globalen Variablen mit den alten Werten für die lokalen kombinieren:

$$\mathsf{combine}\ (\rho_1, \rho_2) = (\rho_1|_{\mathit{Locals}}) \oplus (\rho_2|_{\mathit{Globals}})$$

- ... dazwischen den Berechnungs-Wald auswerten:

$$\begin{array}{rcl} [\![k\ \langle w \rangle]\!]\ (\rho, \mu) & = & \mathsf{let}\ \ (\rho_1, \mu_1) = [\![w]\!]\ (\mathsf{enter}\ \rho, \mu) \\ & & \mathsf{in}\ \ \ (\mathsf{combine}\ (\rho, \rho_1), \mu_1) \end{array}$$

Achtung:

- $[\![w]\!]$ ist i.a. nur partiell definiert :-)
- Spezielle globale/lokale Variablen a_i, b_i, ret können eingesetzt werden, bestimmte Aufruf-Konventionen zu simulieren.
- 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:

$$
\begin{array}{lcl}
\mathit{configuration} & = & \mathit{stack} \times \mathit{store} \\
\mathit{store} & = & \mathbb{N} \to \mathbb{Z} \\
\mathit{stack} & = & \mathit{frame} \cdot \mathit{frame}^* \\
\mathit{frame} & = & \mathit{point} \times \mathit{locals} \\
\mathit{locals} & = & (\mathit{Vars} \to \mathbb{Z})
\end{array}
$$

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 $k = (u, f\,();, v)$:

$$(\boxed{(u, \rho)} \cdot \sigma, \mu) \Longrightarrow (\boxed{(u_f, \mathsf{enter}\ \rho) \cdot (v, \rho)} \cdot \sigma, \mu)$$

u_f Anfangspunkt von f

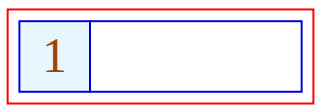
Rückkehr:

$$(\boxed{(r_f, \rho_2) \cdot (v, \rho_1)} \cdot \sigma, \mu) \Longrightarrow (\boxed{(v, \mathsf{combine}\,(\rho_1, \rho_2))} \cdot \sigma, \mu)$$

r_f Endpunkt von f

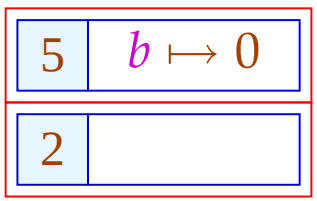
Mit dem Aufruf-Keller verwalten wir explizit den DFS-Durchlauf über den Berechnungswald :-)

... im Beispiel:



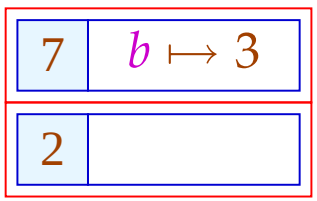
Mit dem Aufruf-Keller verwalten wir explizit den DFS-Durchlauf über den Berechnungswald :-)

... im Beispiel:



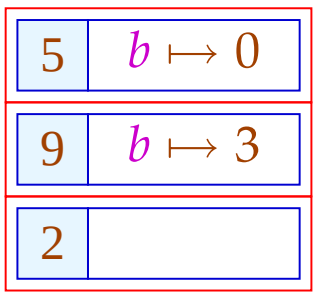
Mit dem Aufruf-Keller verwalten wir explizit den DFS-Durchlauf über den Berechnungswald :-)

... im Beispiel:



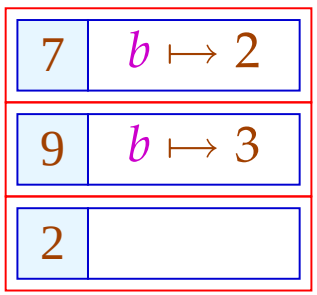
Mit dem Aufruf-Keller verwalten wir explizit den DFS-Durchlauf über den Berechnungswald :-)

... im Beispiel:



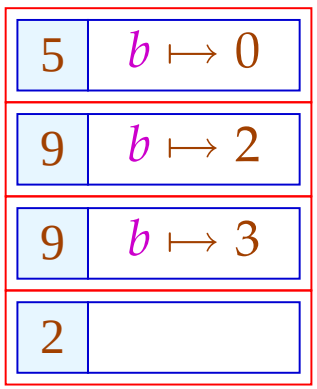
Mit dem Aufruf-Keller verwalten wir explizit den DFS-Durchlauf über den Berechnungswald :-)

... im Beispiel:



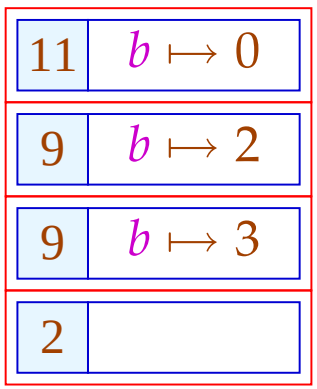
Mit dem Aufruf-Keller verwalten wir explizit den DFS-Durchlauf über den Berechnungswald :-)

... im Beispiel:



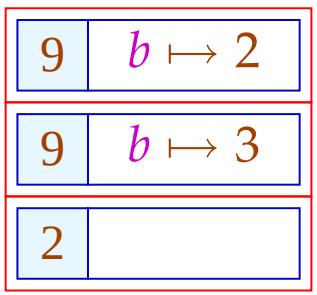
Mit dem Aufruf-Keller verwalten wir explizit den DFS-Durchlauf über den Berechnungswald :-)

... im Beispiel:



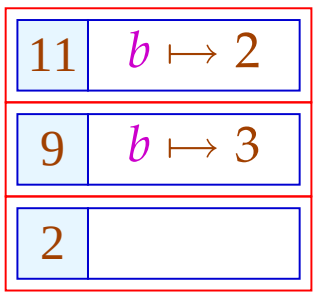
Mit dem Aufruf-Keller verwalten wir explizit den DFS-Durchlauf über den Berechnungswald :-)

... im Beispiel:



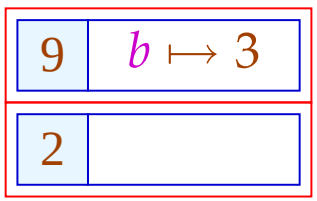
Mit dem Aufruf-Keller verwalten wir explizit den DFS-Durchlauf über den Berechnungswald :-)

... im Beispiel:



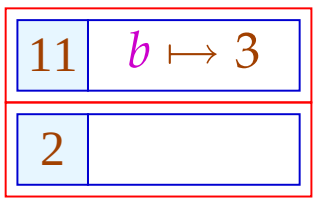
Mit dem Aufruf-Keller verwalten wir explizit den DFS-Durchlauf über den Berechnungswald :-)

... im Beispiel:



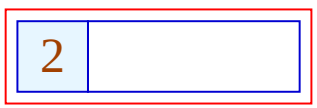
Mit dem Aufruf-Keller verwalten wir explizit den DFS-Durchlauf über den Berechnungswald :-)

... im Beispiel:



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

... im Beispiel:



Diese operationelle Semantik ist einigermaßen realistisch :-)

Kosten eines Prozedur-Aufrufs:

Vor Betreten des Rumpfs:
- Anlegen eines Kellerrahmens;
- Retten der Register;
- Retten der Fortsetzungsaddresse;
- Anspringen des Rumpfs.

Bei Beenden des Aufrufs:
- Aufgeben des Kellerrahmens;
- Restaurieren der Register;
- Übergeben des Ergebnisses;
- Rücksprung hinter die Aufrufstelle.

$\Longrightarrow$ ... ziemlich teuer !!!

1. Idee: Inlining

Kopiere den Funktionsrumpf an jede Aufrufstelle !!!

Beispiel:

```
abs () {
        a2 = −a1;
        max ();
}
```

```
max () {
        if (a1 < a2) { ret = a2; goto _exit; }
        ret = a1;
_exit :
}
```

... liefert:

```
abs () {
a2 = −a1;
        if (a1 < a2) { ret = a2; goto _exit; }
        ret = a1;
 _exit :
}
```

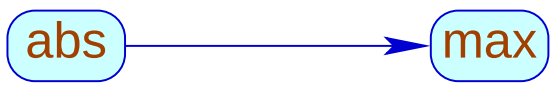
Probleme:

- Der einkopierte Block modifiziert evt. die lokalen Variablen der aufrufenden Prozedur ???
- Allgemeiner: Mehrfachbenutzung gleicher Variablennamen kann zu Fehlern führen.
- Mehrfach-Verwendung einer Prozedur führt zu Code-Duplizierung :-((
- Wie gehen wir mit Rekursion um ???

Erkennen von Rekursion:

Wir konstruieren den Aufruf-Graph des Programms.

In den Beispielen:



Aufruf-Graph:

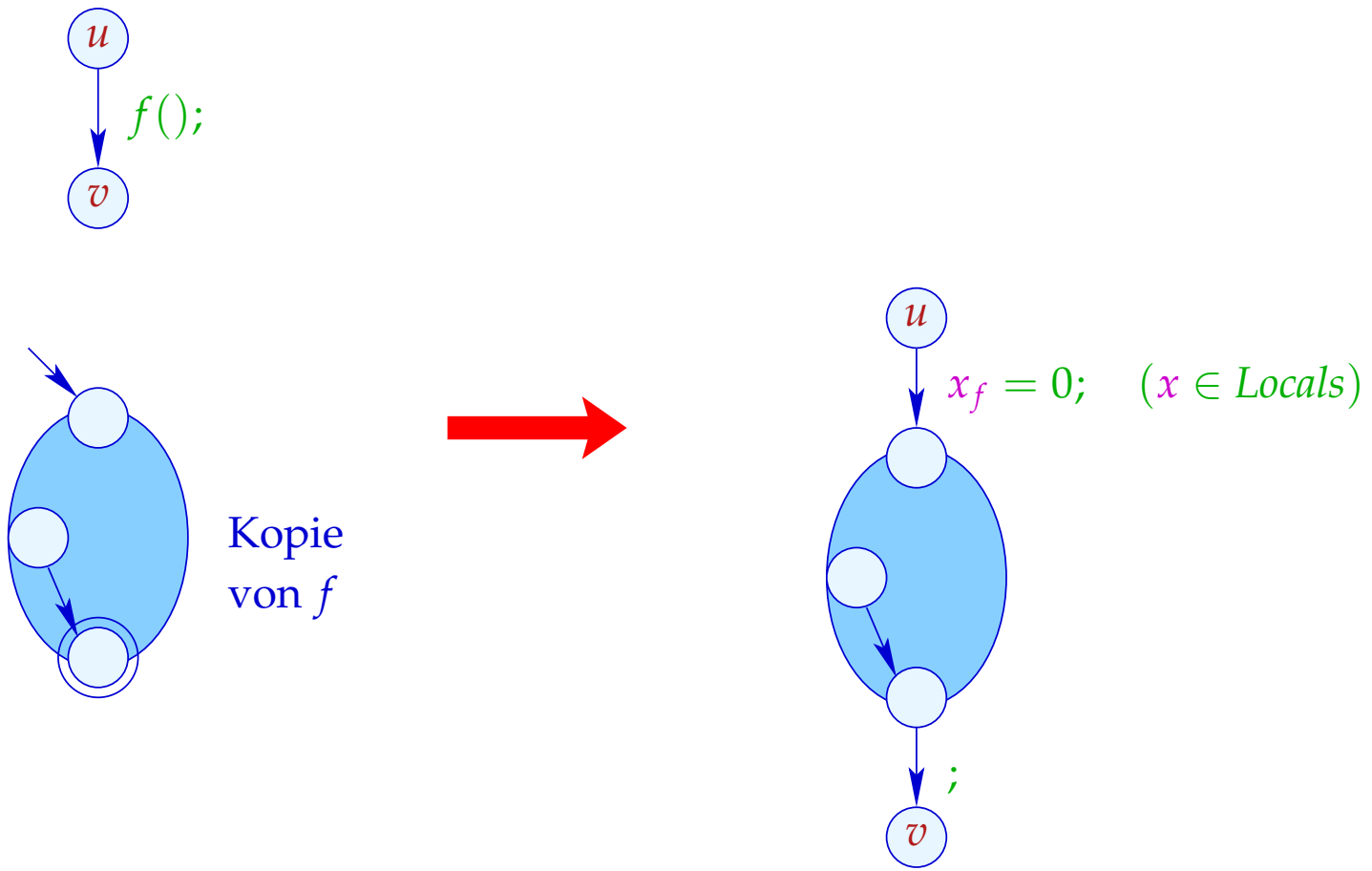
- Die Knoten sind die Prozeduren.
- Eine Kante geht von g nach h, sofern der Rumpf von g einen Aufruf von h enthält.

Strategien für Inlining:

- Kopiere nur Blatt-Prozeduren ein, d.h. solche ohne weitere Aufrufe :-)
- Kopiere sämtliche nicht-rekursiven Prozeduren ein!

... wir betrachten hier nur Blatt-Prozeduren ;-)

Transformation 9:



Beachte:

- Die Nop-Kante können wir ebenfalls einsparen, da der *stop*-Knoten von f selbst keine ausgehenden Kanten hat ...
- Die x_f sind die Kopien der lokalen Variablen der Prozedur f.
- Diese müssen gemäß unserer Semantik für Prozeduraufrufe mit 0 initialisiert werden :-)