Karlsruhe Institute of Technology

Sprachtechnologie und Compiler

Prof. Dr.-Ing. Gregor Snelting

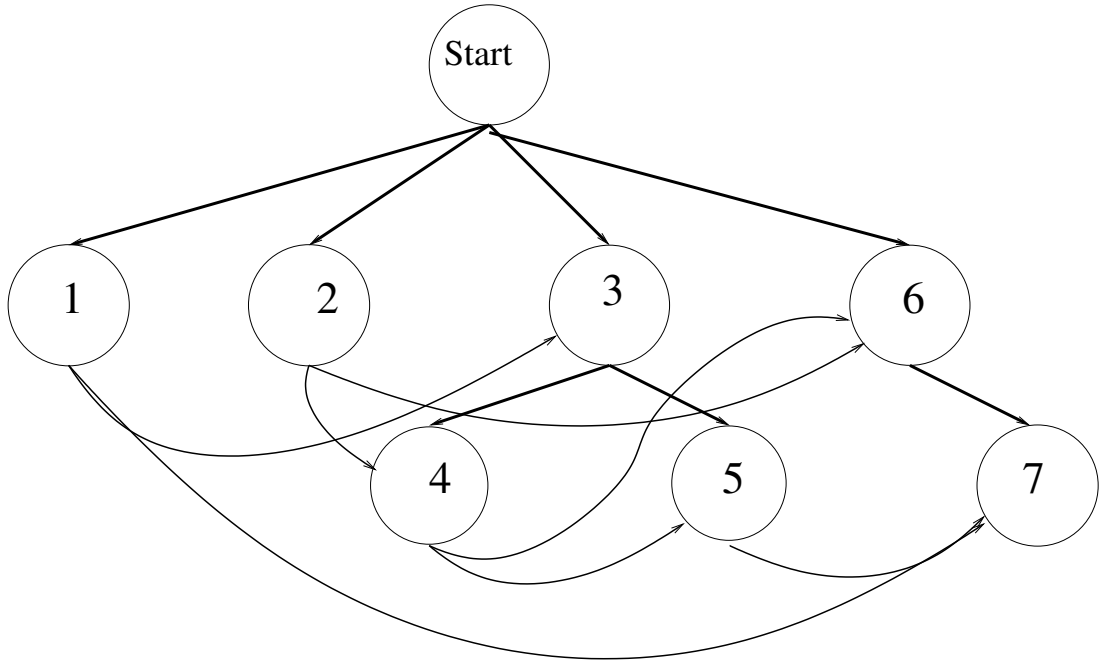
Überarbeitung: Matthias Braun, Sebastian Buchwald, Manuel Mohr, Andreas Zwinkau
Einige Teile basieren auf Unterlagen von Prof. em. Dr. Dr. h.c. Gerhard Goos

Institut für Programmstrukturen und Datenorganisation, Karlsruher Institut für Technologie (KIT)

Programmabhängigkeitsgraph – Beispiel

```
(1) x := 42;
(2) y := 17;
(3) IF x<0 THEN
(4)      read(y);
(5)      x := y;
    END;
(6) IF y>0 THEN
(7)      p(x);
```

$\Longrightarrow$



- Knoten: Anweisungen und Ausdrücke
- Datenabhängkeit $x \rightarrow y$: in x definierte Variable wird in y verwendet
- Kontrollabhängigkeit $x \rightarrow y$: x entscheidet, ob y ausgeführt wird
- Falls es Funktionen, Adressen (Aliasing) oder komplexe Datenstrukturen gibt, wird der PDG wesentlich komplizierter

Programmabhängigkeitsgraph (1/2)

Seien x, y Anweisungen bzw. Grundblöcke; $gen(x)$, $in(x)$ wie vorher, $var(d)$ sei die in einer Definition definierte Variable; $uses(x)$ seien die in x benutzten Variablen.

Definition *(Datenabhängigkeit)*

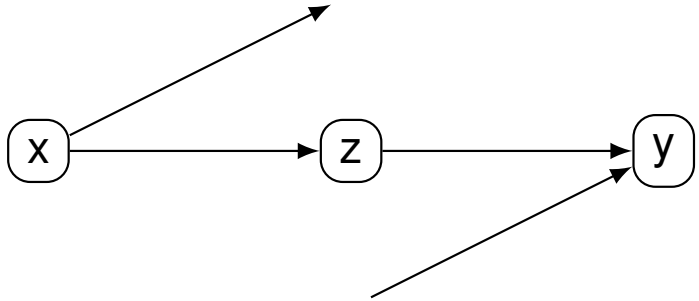
$$x \to y \iff \exists d \in gen(x) : d \in in(y) \wedge var(d) \in uses(y)$$

Definition *(Dominanz)*
x dominiert y, wenn jeder CFG-Pfad von *Start* zu y über x führt.

Definition *(Postdominanz)*
y postdominiert x, wenn jeder CFG-Pfad von x zu *Stop* über y führt.

Programmabhängigkeitsgraph (2/2)

Definition *(Kontrollabhängigkeit)*
$x \to y \iff \exists$ Pfad P von x nach y im CFG $\forall z \in P, z \neq x, z \neq y$:
y postdominiert $z \ \wedge \ y$ postdominiert nicht x



Bemerkung: In strukturierten Programmen sind kontrollabhängige Anweisungen Y gerade jene im Rumpf eines `if`, `while` usw; sie hängen von der regierenden Bedingung x ab

Definition Der PDG $(S, \to)$ hat alle Anweisungen als Knoten und Daten-/Kontrollabhängigkeiten als Kanten.

Beispiel: Programm für elektronische Waage (PTB)

Karlsruhe Institute of Technology

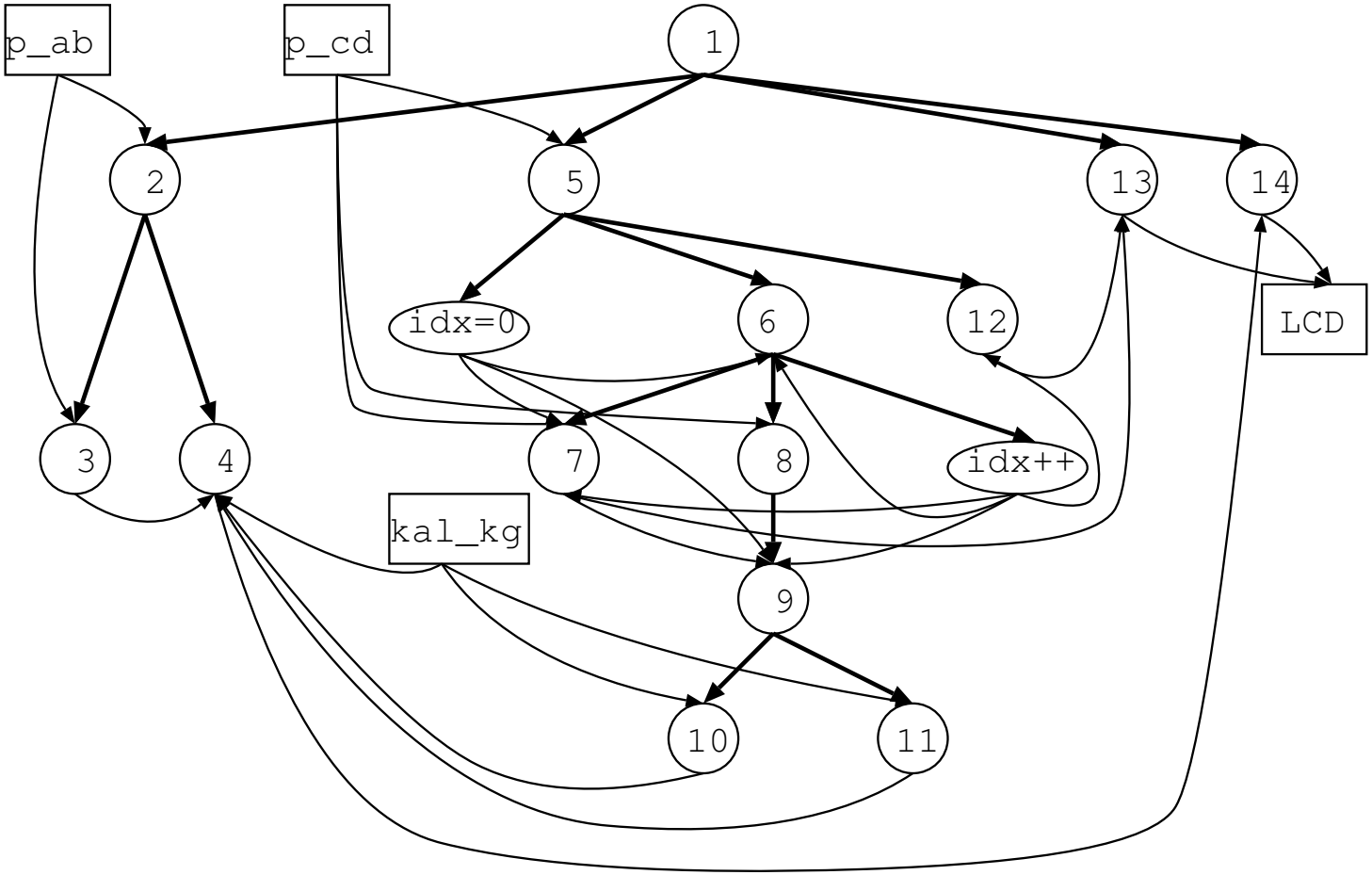
```
    void main() {
        int p_ab[2] = {0, 1};
        int p_cd[1] = {0};
        char e_puf[8];
        int u; int idx;
        float u_kg; float kal_kg = 1.0;

(1)     while(TRUE) {
(2)         if ((p_ab[CTRL2] & 0x10)==0) {
(3)             u = ((p_ab[PB] & 0x0f) << 8) + (unsigned int)p_ab[PA];
(4)             u_kg = u * kal_kg;
            }
(5)         if ((p_cd[CTRL2] & 0x01) != 0) {
(6)             for (idx=0;idx<7;idx++) {
(7)                 e_puf[idx] = p_cd[PA];
(8)                 if ((p_cd[CTRL2] & 0x10) != 0) {
(9)                     switch(e_puf[idx]) {
(10)                        case '+': kal_kg *= 1.01; break; /* unerlaubter */
(11)                        case '-': kal_kg *= 0.99; break; /* Datenfluss */
                }   }   }
(12)            e_puf[idx] = '\0';
            }
(13)        printf("Artikel: %07.7s\n,e_puf);
(14)        printf("   %6.2f kg   ",u_kg);
      }   }
```

p_cd: Keyboard-Eingaberegister; Kontrollbits

p_ab: Messwert-Eingaberegister

Beispiel: Programm für elektronische Waage – PDG



fett: Kontrollabhängigkeiten
dünn: Datenabhängigkeiten

Program Slicing (1/2)

Welche Anweisungen y können Programmpunkt x beeinflussen, und welche tun dies garantiert nicht?

Definition: Der Rückwärtsslice $BS(x)$ enthält alle Anweisungen, die x beeinflussen können.
Im PDG ist

$$BS(x) = \{y \mid y \to^* x\}$$

Bemerkung: $BS(x)$ darf zu groß sein, aber niemals zu klein (Prinzip der konservativen Approximation). In der Praxis will man natürlich möglichst kleine $BS(x)$

Beispiel: Für die elektronische Waage ist $(5) \in BS((14))$. Deshalb potentielle Beeinflussung des angezeigten Messwertes durch das Keyboard.

Program Slicing (2/2)

Karlsruhe Institute of Technology

Definition: Vorwärtsslice

$$FS(x) = \{y \mid x \rightarrow^* y\}$$

Definition: Chop

$$CH(x, y) = \{z \mid x \rightarrow^* z \rightarrow^* y\} = BS(y) \cap FS(x)$$

Bemerkung: Diese einfachen Formeln gelten nicht mehr im interprozeduralen Fall sowie für komplexe Datenstrukturen.

Slicing-Theorem: Wenn es keinen Pfad $x \rightarrow^* y$ im PDG gibt, so kann x garantiert nicht y beeinflussen.

Statische Einmalzuweisung

Statische Einmalzuweisung (engl. static single assignment)

Ein Programm ist in **SSA-Form**, wenn es für jede Variable genau eine Zuweisung gibt.

Beispiel:

```
in = STDIN
while (!eof(in)) {
  c = read(in)
  e = (c+13) % 26
  print(e)
}
```

```
int f(int x) {
  if x <= 1
   return x
  res = f(x-1)
  res = res + f(x-2)
  return res
}
```

```
int f(int x0) {
  if x0 <= 1
   return x0
  res0 = f(x0-1)
  res1 = res0 + f(x0-2)
  return res1
}
```

SSA?

Statische Einmalzuweisung

Statische Einmalzuweisung (engl. static single assignment)

Ein Programm ist in **SSA-Form**, wenn es für jede Variable genau eine Zuweisung gibt.

Beispiel:

```
in = STDIN
while (!eof(in)) {
  c = read(in)
  e = (c+13) % 26
  print(e)
}
```

SSA? ✔

```
int f(int x) {
  if x <= 1
   return x
  res = f(x-1)
  res = res + f(x-2)
  return res
}
```

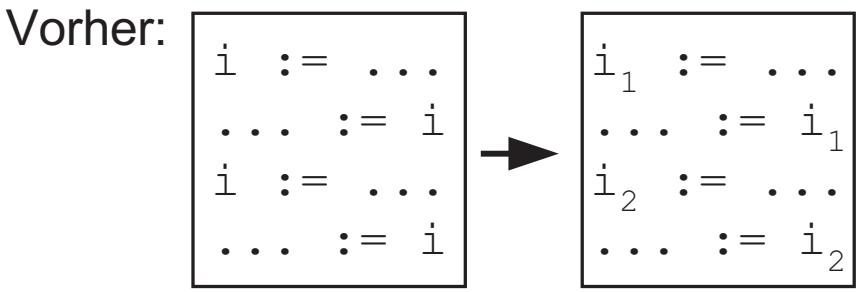
✖

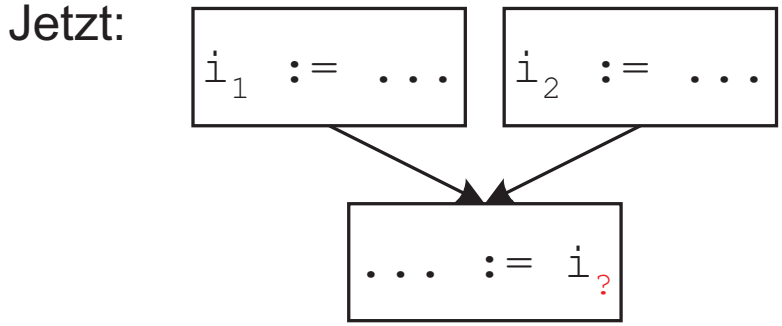
```
int f(int x0) {
  if x0 <= 1
   return x0
  res0 = f(x0-1)
  res1 = res0 + f(x0-2)
  return res1
}
```

✔

SSA-Form – Konstruktion

Grundidee:





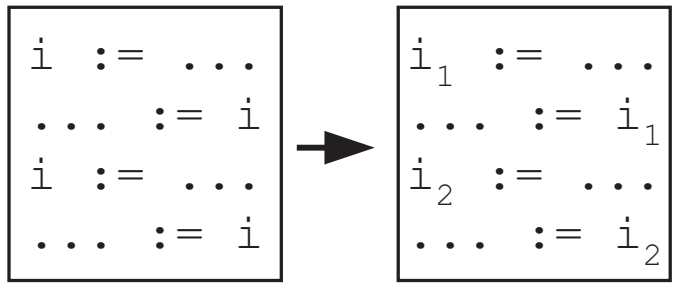
- Mehrere Zuweisungen an Variable x
  $\Rightarrow$ durchnummerieren
- Bei Verwendungen passende Variante nehmen (Sichtbare Definitionen)
- Welche Nummer nimmt man, wenn Pfade im CFG zusammenlaufen?

SSA-Form – Konstruktion

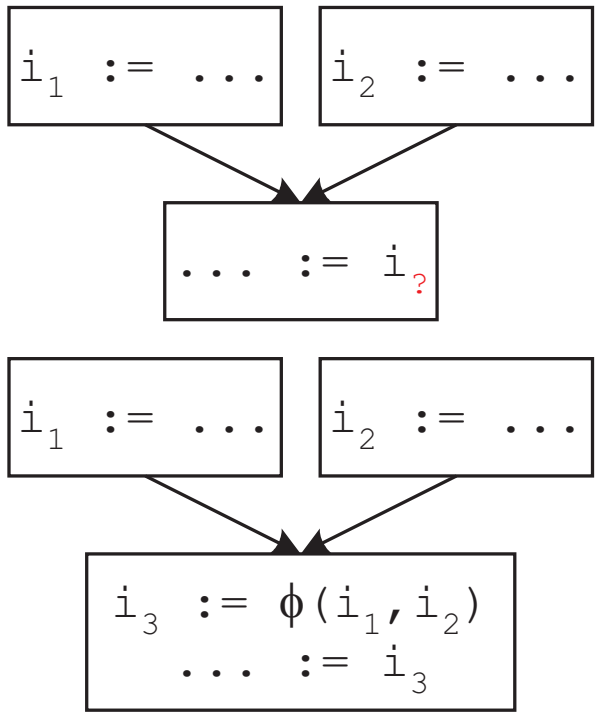
Grundidee:

- Mehrere Zuweisungen an Variable x
  $\Rightarrow$ durchnummerieren
- Bei Verwendungen passende Variante nehmen (Sichtbare Definitionen)
- Welche Nummer nimmt man, wenn Pfade im CFG zusammenlaufen?
- ϕ-Funktion

Vorher:

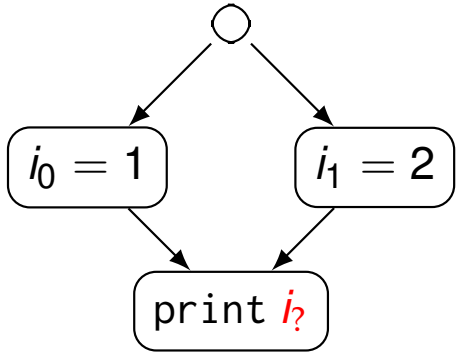


Jetzt:



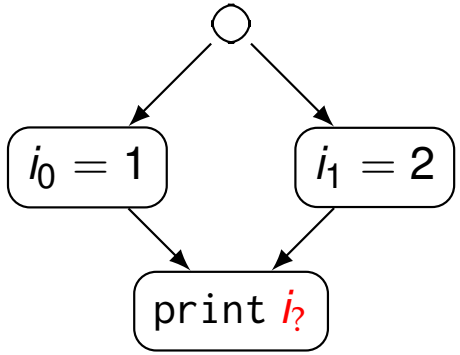
ϕ-Funktionen

Einmalzuweisung bei Zusammenfluss?

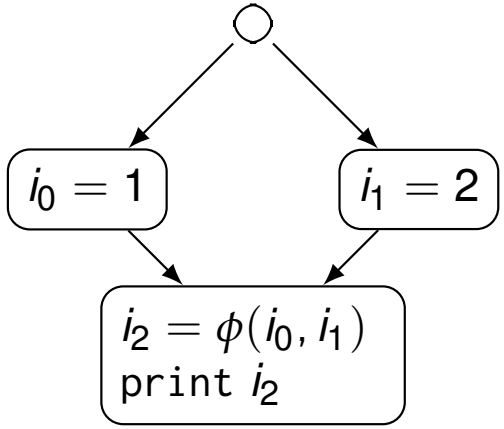


ϕ-Funktionen

Einmalzuweisung bei Zusammenfluss?



Definiere neue Variable mit ϕ-Funktion!



ϕ-Funktionen

Eine ϕ-Funktion

$$\phi(\alpha_0, \alpha_1, \ldots)$$

in Grundblock G (im Kontrollflussgraph)

- besitzt einen Operand pro Vorgänger von G
- wählt α_k aus, wenn G über k-te Kante betreten wird

Achtung: Auswertung geschieht beim Betreten des Grundblocks:

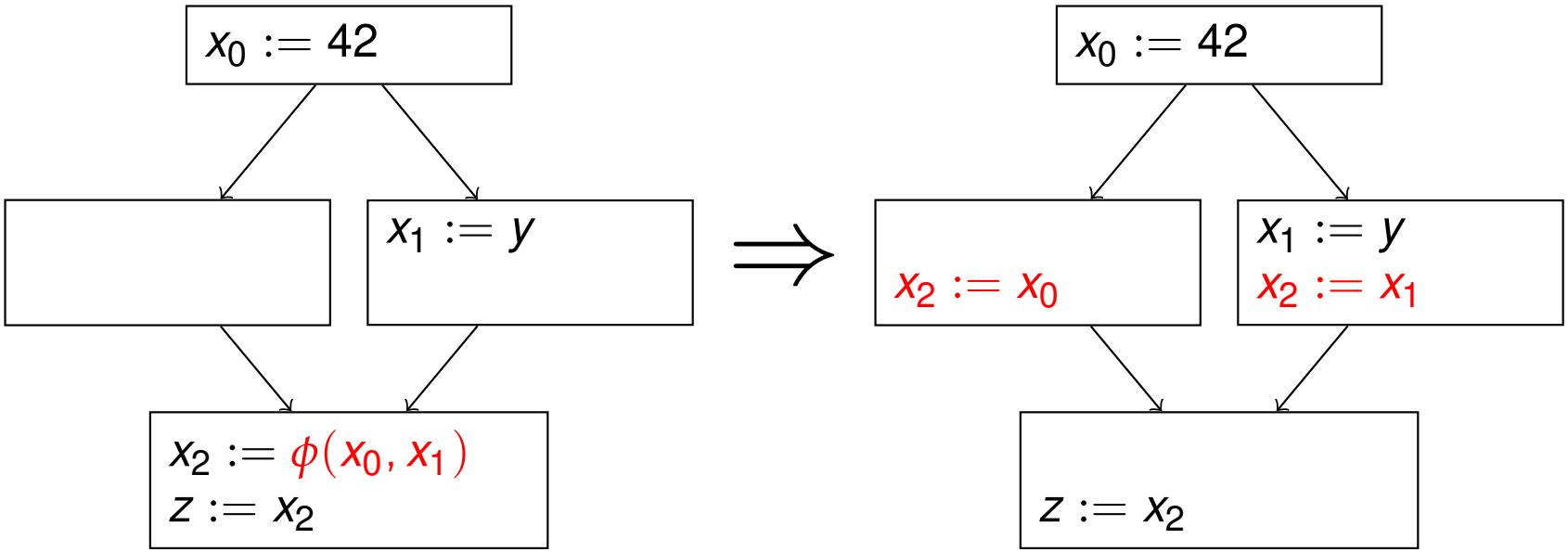
- Gleichzeitige Auswertung aller ϕ-Funktionen des Grundblocks
- Müssen am Anfang des Grundblocks stehen

- Rein theoretische Konstrukte
- Zur Codegenerierung wieder eliminieren

Bemerkung: α_i sind nicht auf eine Variable eingeschränkt: z. B. $\phi(x_{42}, y_{11})$ möglich

Codegenerierung mit ϕ-Funktionen

Für $x_i = \phi(\alpha_1, \ldots, \alpha_n)$, erstelle in jedem Vorgängerblock$_k$ eine Kopieroperation $x_i = \alpha_k$.



Eigenschaften, Vorteile

Eigenschaften

- SSA ist eine Eigenschaft
- Eine Zuweisung pro Variable, aber mehrfach ausführbar
- Nur sinnvoll für aliasfreie lokale Variablen
- Transformation in SSA-Form praktisch linear
  (theoretisch quadratisch)

Vorteile

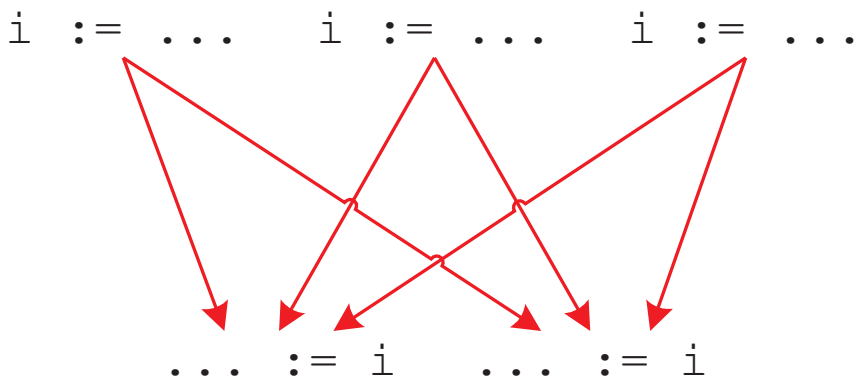
- Beziehung Variable $\leftrightarrow$ Definition ist explizit
  - Def-Use-Analyse entfällt; Def-Use-Information stets aktuell
  - Variablenfreie Darstellung möglich:
    Benutze Wertdefinition statt Variablen als Operanden
  - Aus syntaktischer Gleichheit folgt Wertgleichheit
    (bei arithmetischen Ausdrücken)
- Viele Analysen vereinfachen sich drastisch

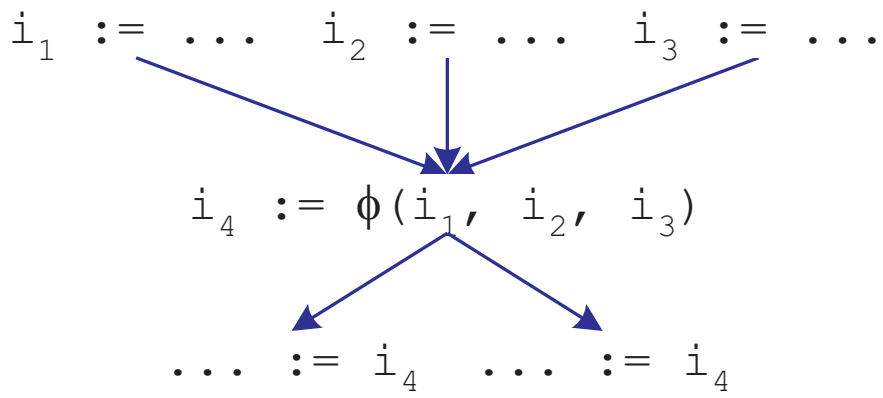
Definiert-Benutzt-Beziehungen

Die SSA-Form verringert den Aufwand zur Darstellung von Definiert-Benutzt-Beziehungen:

vorher: n^2

jetzt: $2n$



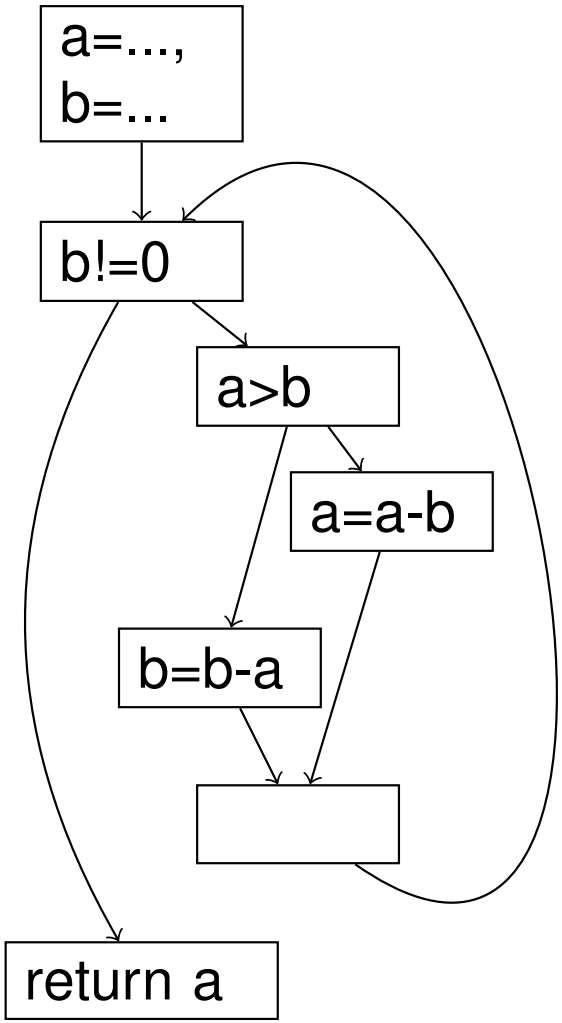


theoretisch aber bis zu n^2 ϕ-Funktionen.

Beispiel: Schleifen, Bedingter Code

```
int gcd(a, b) {
  while (
          b != 0) {
    if (a > b) {
      a = a - b
    } else {
      b = b - a
    }


  }
  return a
}
```




```
int gcd(a0, b0) {
  while (a1 = φ(a0, a3),
          b1 = φ(b0, b3),
          b1 != 0) {
    if (a1 > b1) {
      a2 = a1 - b1
    } else {
      b2 = b1 - a1
    }
    a3 = φ(a2, a1)
    b3 = φ(b1, b2)
  }
  return a1
}
```

Beispiel: Werte vertauschen

```
x = 1
y = 2
while (true) {

  t = y;
  y = x;
  x = t;
  print(x, y)
}
```

```
x0 = 1
y0 = 2
while (true) {
  x1 = φ(x0, x2)
  y1 = φ(y0, y2)
  t0 = y1;
  y2 = x1;
  x2 = t0;
  print(x2, y2)
}
```

```
x0 = 1
y0 = 2
while (true) {
  x1 = φ(x0, y1)
  y1 = φ(y0, x1)


  print(y1, x1)
}
```

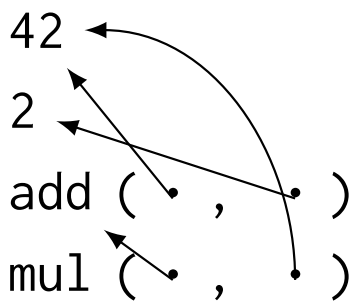
Copy Propagation!

Implementierungsvarianten

1. Tripelform mit Variablen (Bsp.: Gnu Compiler Collection)

$$
\begin{aligned}
a_0 &:= 42 \\
b_0 &:= 2 \\
c_0 &:= \mathtt{add}(a_0, b_0) \\
c_1 &:= \mathtt{mul}(c_0, a_0)
\end{aligned}
$$

2. Variablenfrei: Operanden sind Zeiger auf Definitionen (Bsp.: Low Level Virtual Machine)

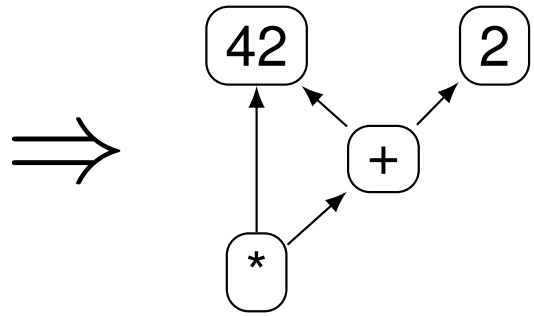


3. Als Programmgraph (Bsp.: Sun JavaVM, libFirm)

Programmgraph – Ausdrücke

- Jede Operation ist ein Knoten
- (geordnete) Datenabhängigkeitskanten zu Operanden
- Keine Totalordnung

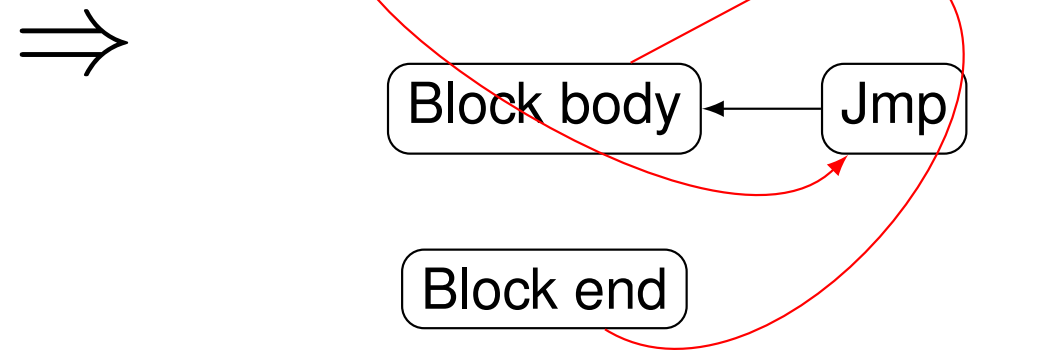
a_0 := 42
b_0 := 2
c_0 := add(a_0, b_0)
c_1 := mul(c_0, a_0)



Programmgraph – Kontrollfluss

- Grundblöcke sind Knoten
- Erster „Operand“ einer Operation ist ihr Grundblock
- Sprünge zu einem Block sind dessen Operanden (Kontrollflussabhängigkeitskanten)

```
start:
  Jmp loop
loop:
  CondJmp end
body:
  # ...
  Jmp loop
end:
  # ...
```

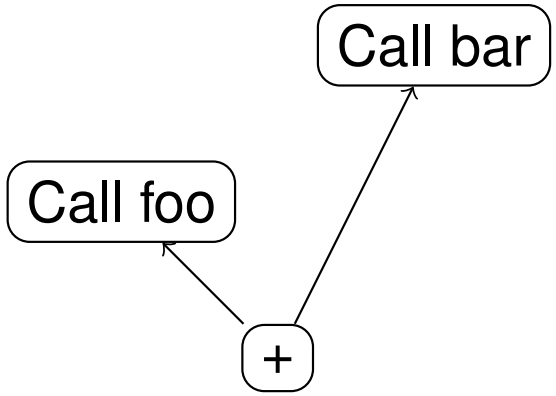


Programmgraph – Zustand

Problem: Nebenwirkungen implizieren eine Ordnung, aber keine Daten- oder Kontrollflussabhängigkeit

Beispiel: foo() vor bar(), aber keine Abhängigkeit?

```
x = foo();
y = bar();
z = x + y;
```

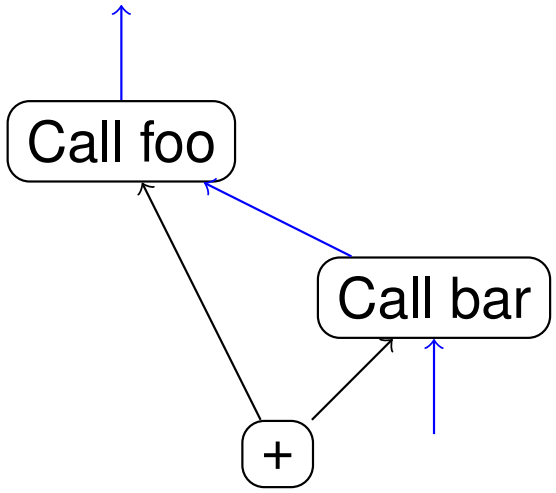


Programmgraph – Zustand

Problem: Nebenwirkungen implizieren eine Ordnung, aber keine Daten- oder Kontrollflussabhängigkeit

Beispiel: foo() vor bar(), aber keine Abhängigkeit?

```
x = foo();
y = bar();
z = x + y;
```



Modelliere den „globalen Zustand“ (RAM,IO,...) als zusätzlichen Operanden

Beispiel aus libFirm

```
int foo(int x, int y) {
    if (x < 0)
        x = abs(x);
    return x*y;
}
```