Kapitel 11

SSA-Form

Kapitel 11: SSA-Form

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
}
```

✖

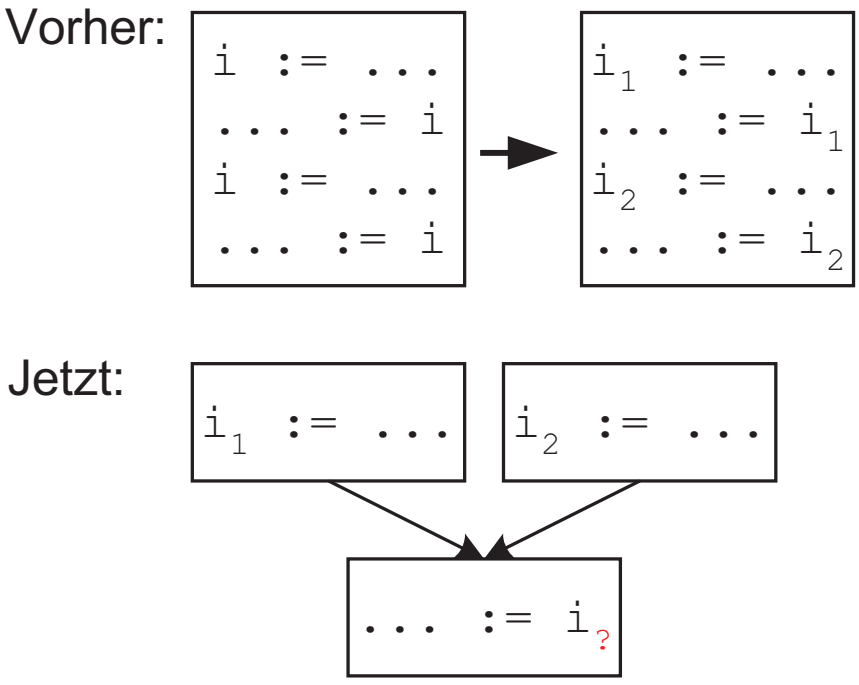
```
int f(int x0) {
   if x0 <= 1
    return x0
   res0 = f(x0−1)
   res1 = res0 + f(x0−2)
   return res1
}
```

✔

SSA Form – Konstruktion

Grundidee:

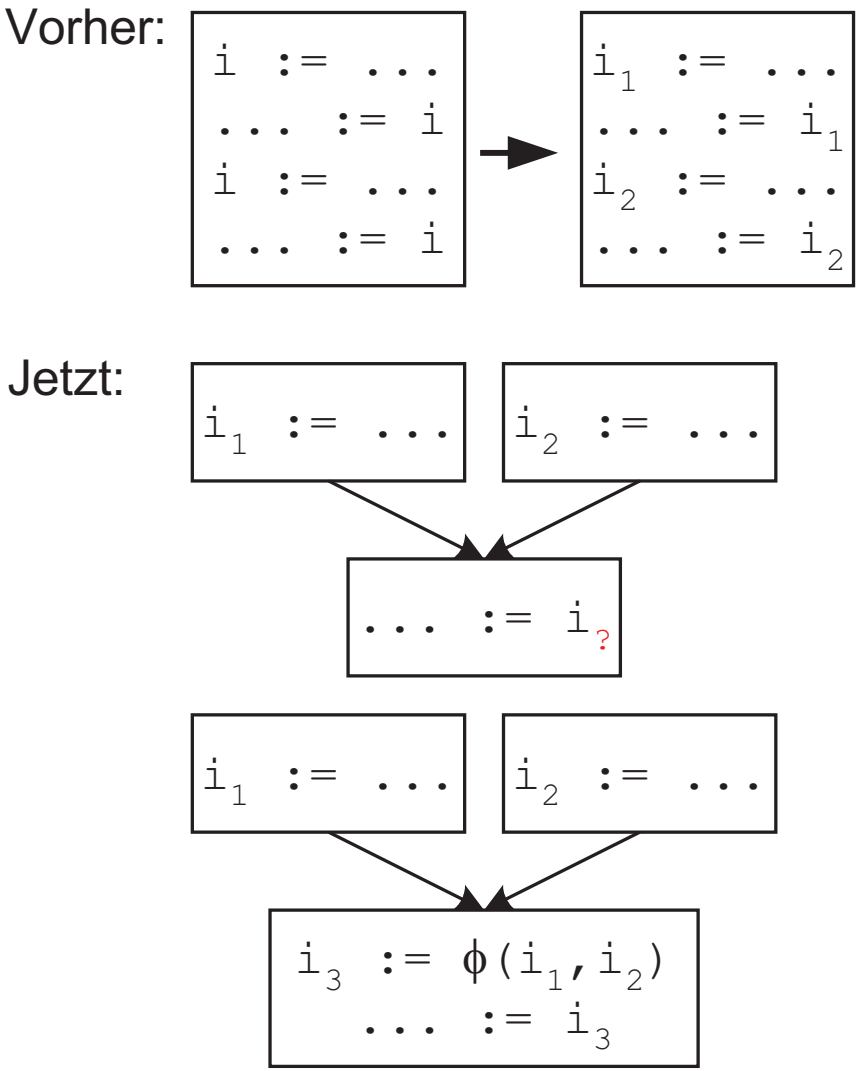
- Mehrere Zuweisungen an Variable x
  $\Rightarrow$ durchnummerieren
- Bei Verwendungen passende Variante nehmen (Sichtbare Definitionen)
- Welche Nummer nimmt man, wenn Pfade im CFG zusammenlaufen?



SSA Form – Konstruktion

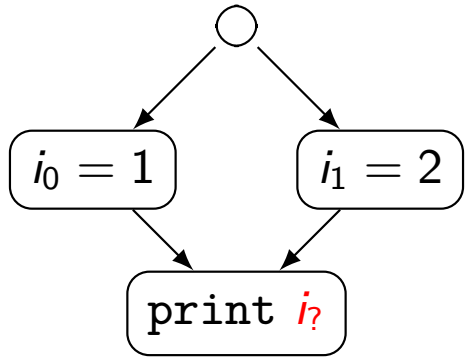
Grundidee:

- 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



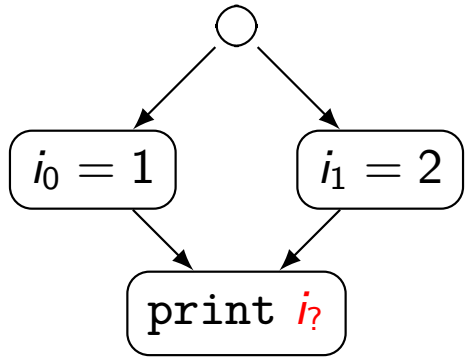
ϕ-Funktionen

Einmalzuweisung bei Zusammenfluss?

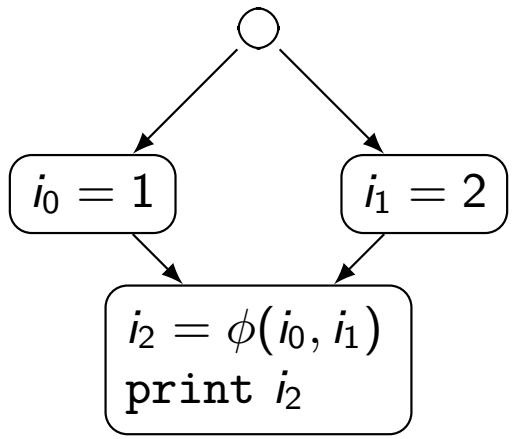


ϕ-Funktionen

Einmalzuweisung bei Zusammenfluss?



Definiere neue Variable mit ϕ-Funktion!



ϕ-Funktionen

Eine ϕ-Funktion

$$\phi(x_0, x_1, \ldots)$$

in Grundblock G (im Kontrollflussgraph)

- besitzt einen Operand pro Vorgänger von G
- wird G über k-te Kante betreten, wird x_k gewählt

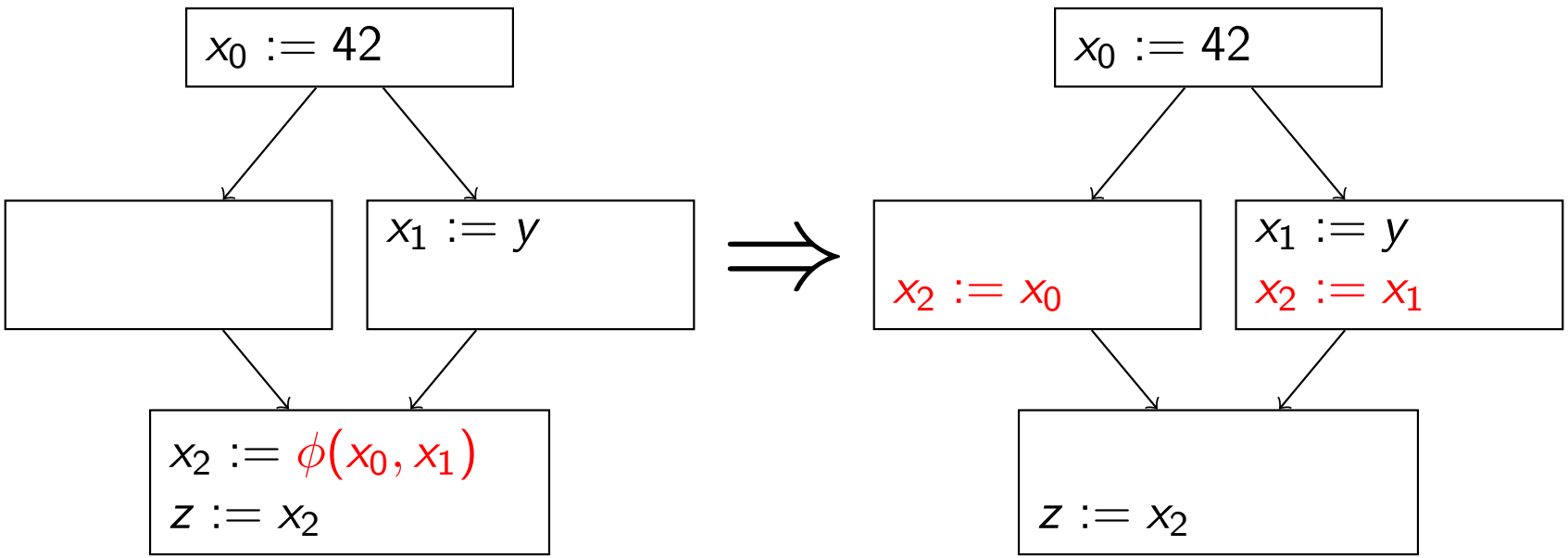
Achtung: Auswertung geschieht beim Betreten des Grundblocks:

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

- Rein theoretische Konstrukte
- Zur Codegenerierung wieder eliminieren

Codegenerierung mit ϕ-Funktionen

Für $x_i = \phi(x_1, \ldots, x_n)$, erstelle in jedem Vorgängerblock$_k$ eine Kopieroperation $x_i = x_k$.



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

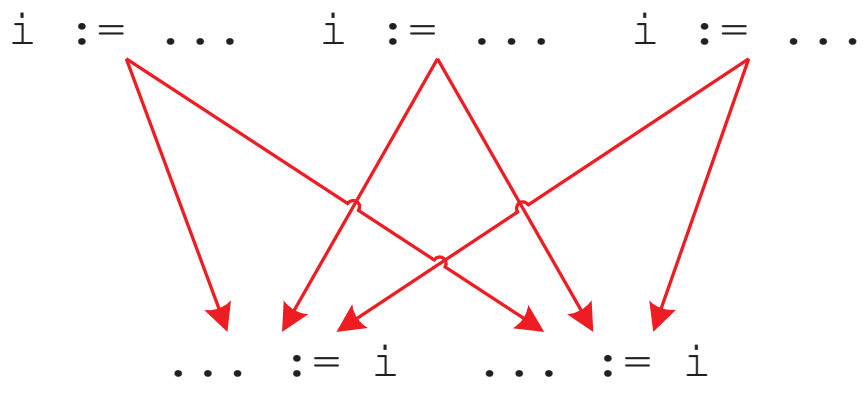
- Variable $\leftrightarrow$ Definition Beziehung ist explizit
  - Def-Use-Analyse entfällt; Def-Use-Information stets up-to-date.
  - Variablenfreie Darstellung möglich:
    Benutze Wertdefinition statt Variablen als Operanden
  - Aus syntaktischer Gleichheit folgt Wertgleichheit
    (bei arithmetischen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

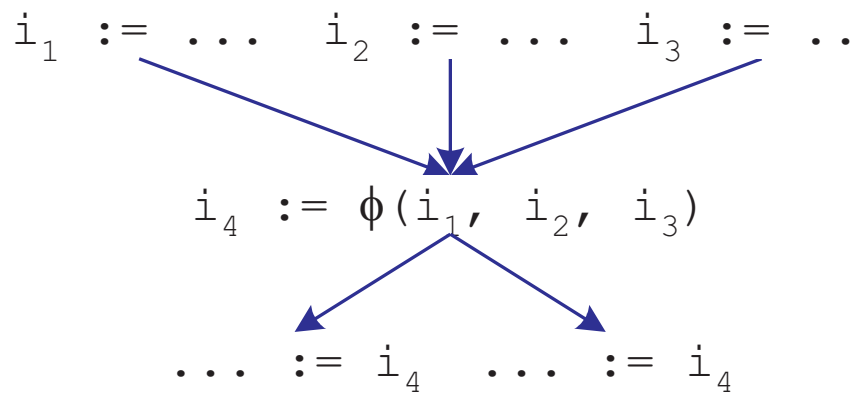
Definiert-Benutzt-Beziehungen

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

vorher: n^2

jetzt: $2n$

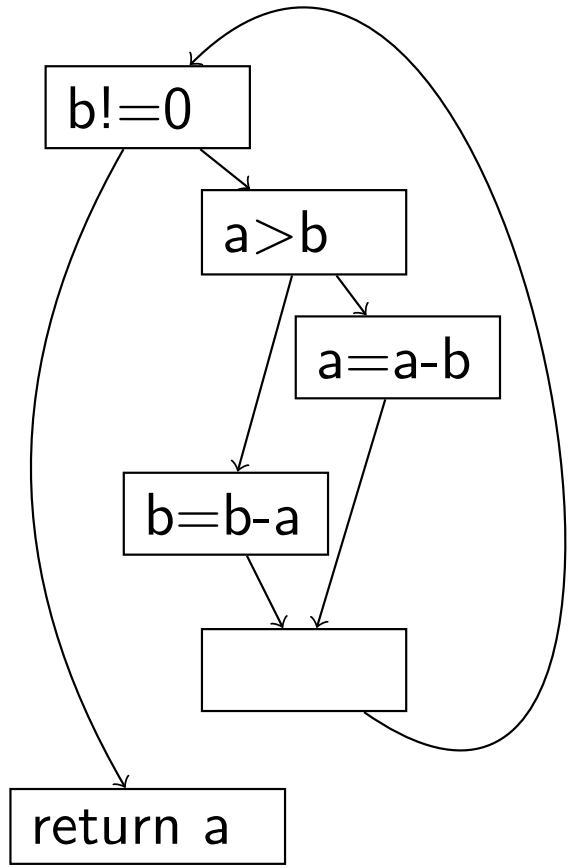




theoretisch aber bis zu n^2 ϕ-Funktionen.

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)
}
```

```
x_0 = 1
y_0 = 2
while (true) {
  x_1 = φ(x_0, y_2)
  y_1 = φ(y_0, x_2)
  t_0 = y_1;
  y_2 = x_1;
  x_2 = t_0;
  print(x_2, y_2)
}
```

```
x_0 = 1
y_0 = 2
while (true) {
  x_1 = φ(x_0, y_1)
  y_1 = φ(y_0, x_1)


  print(y_1, x_1)
}
```

Kopienfortpflanzung!

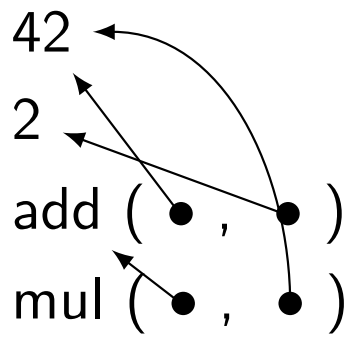
Kapitel 11: SSA-Form

Implementierungsvarianten

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

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

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



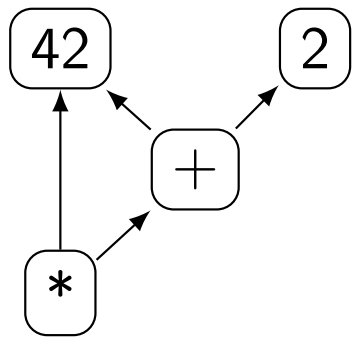
3. Als Programmgraph (Bsp.: Sun JavaVM, libFirm)

Programmgraph – Ausdrücke

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

$a_0 := 42$
$b_0 := 2$
$c_0 := \text{add}(a_0, b_0)$
$c_1 := \text{mul}(c_0, a_0)$

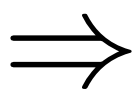
$\Rightarrow$

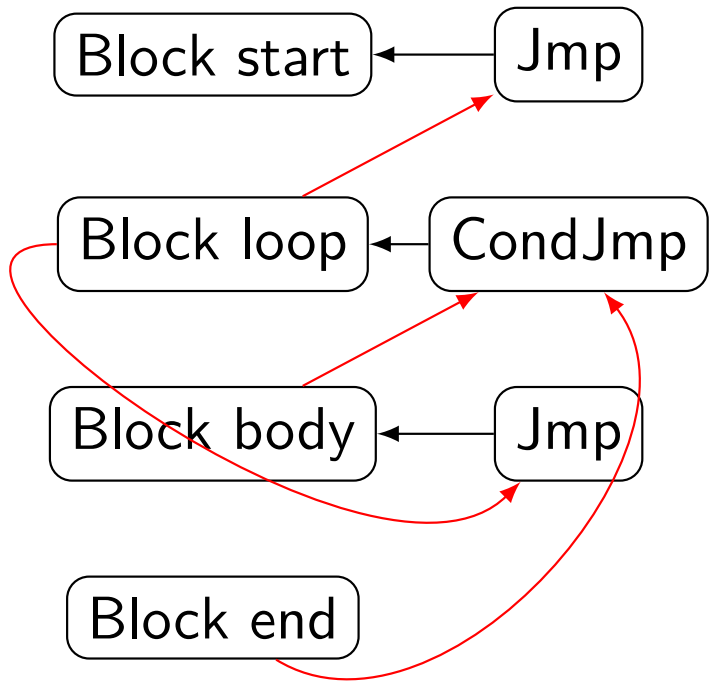


Programmgraph – Kontrollfluss

- 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:
  # ...
```



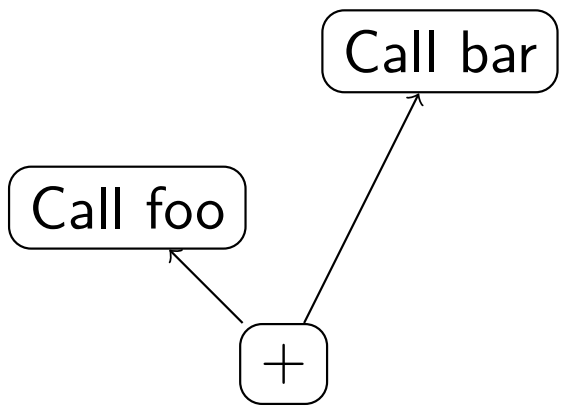


Programmgraph – Zustand

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

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

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

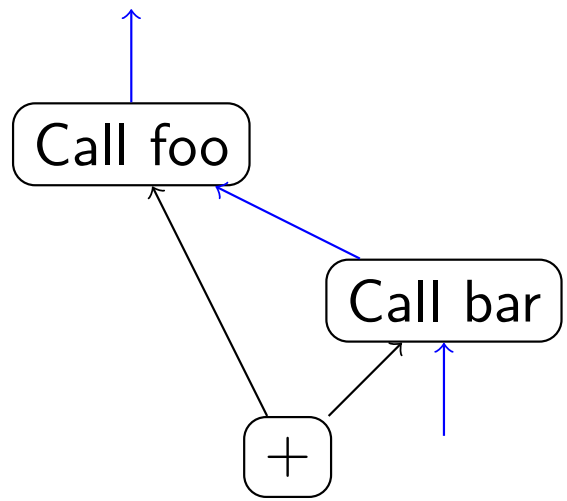


Programmgraph – Zustand

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

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

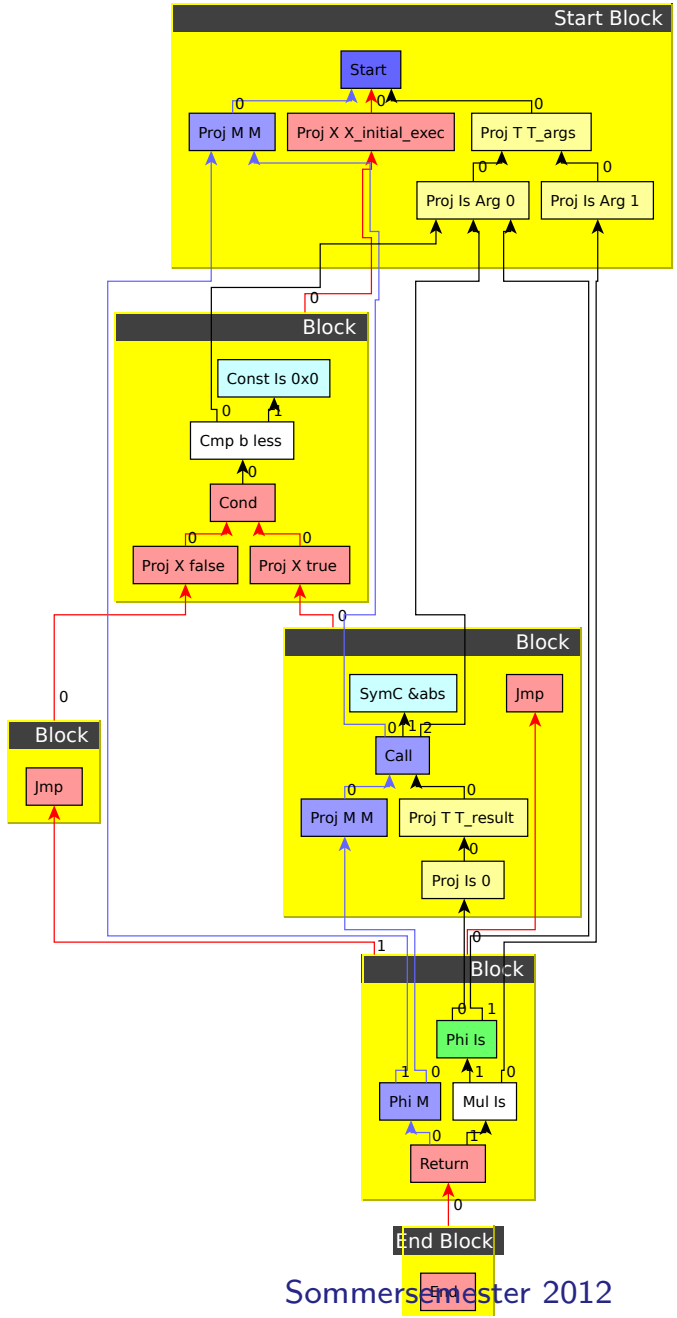
```
x = foo();
y = bar();
z = x + y;
```



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;
}
```



Kapitel 11: SSA-Form

Wdh.: Dominanz und Dominatorbäume

- *Dominanz*: $X \preceq Y$
  Auf jedem Pfad vom Startblock S im Ablaufgraph kommt X vor Y.
  $\preceq$ ist reflexiv: $X \preceq X$.
- *Strikte Dominanz*:

$$X \prec Y \Longrightarrow X \preceq Y \wedge X \neq Y.$$

- *Unmittelbare (direkte) Dominanz*: $\text{idom}(X)$

$$X = \text{idom}(Y) \Longrightarrow X \prec Y \wedge \neg\exists Z : X \prec Z \prec Y.$$

- *Postdominanz*: $X \succeq Y$ Auf jedem Pfad von Y zum Endblock E im Ablaufgraph kommt Y vor X.
- Übrige Definitionen für Postdominanz analog.

Dominanzgrenze und iterierte DG

- *Dominanzgrenze* $DG(X)$
  Menge von Blöcken die gerade nicht mehr von X dominiert werden.

$$DG(X) := \{Y | \exists P \in \text{pred}(Y) : X \,\text{dom}\, P \wedge \neg(X \,\text{dom}\, Y)\}.$$

- Dominanzgrenze einer *Menge* M von Blöcken $DG(M)$

$$DG(M) := \bigcup_{X \in M} DG(X)$$

- *Iterierte Dominanzgrenze* $DG^+(M)$ minimaler Fixpunkt von:

$$\begin{aligned} DG_0 &:= DG(M), \\ DG_{i+1} &:= DG(M \cup DG_i) \end{aligned}$$

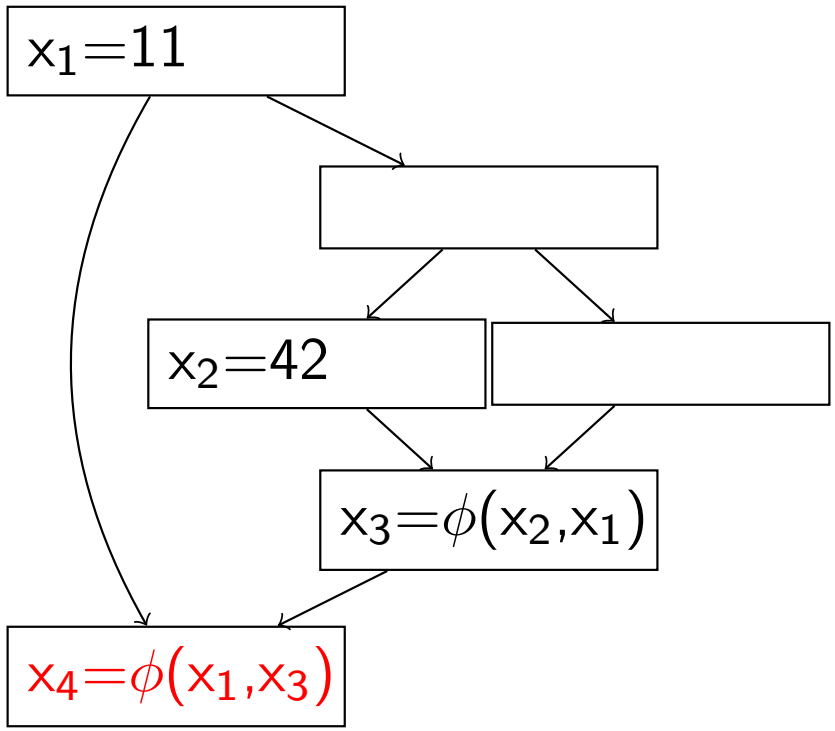
Platzierung von ϕ

- Jede SSA-Variable dominiert alle ihre Verwendungen! (Alle Variablen initialisiert)

  Beweis: angenommen es gibt Verwendung $y = x_i$, die nicht von Zuweisung $x_i = z$ dominiert wird. Dann gibt es Pfad zu $y = x_i$, der nicht über die Zuweisung läuft. Dann ist in der Verwendung x_i nicht immer initialisiert, da $x_i = z$ die einzige Zuweisung an x_i ist.

- „Herrschaftsbereich“ von x_i endet an der Dominanzgrenze
- Deshalb ϕ-Funktion an Dominanzgrenzen $DG(x_i)$ platzieren!
- Führt neue SSA-Variablen $x_j = \phi(...)$ ein
- Für diese gilt dieselbe Argumentation, deshalb ϕ-Funktion in $DG^+(x_i)$ platzieren

Beispiel: Platzierung in DG$^+$



Wie konstruiert man SSA-Form

1. Naiv: alle Grundblöcke ϕ-Funktionen für alle Variable:
   - Aufwand $O(n^2)$, $n =$ Anzahl Variablen, nicht vermeidbar
   - Aber: Praktisch alle Programme nur linear viele ϕ notwendig
2. Platzierung vorberechnen (Cytron et al. 1991):
   1. Kontrollflussgraph und iterierte Dominanzgrenzen berechnen
   2. Entsprechend ϕ-Knoten platzieren
3. On-The-Fly (Click 1995):
   - On-the-fly Aufbau des Abhängigkeitsgraphen
   - Keine Vorstufe zur Kontrollflussgraph- bzw. Dominanzgrenzenberechnung
   - Trick: Vorläufige ϕ'-Funktionen für Grundblöcke, von denen noch nicht alle Vorgänger besucht wurden.

Cytron-Verfahren

SSA-Aufbau nach Cytron et al.:

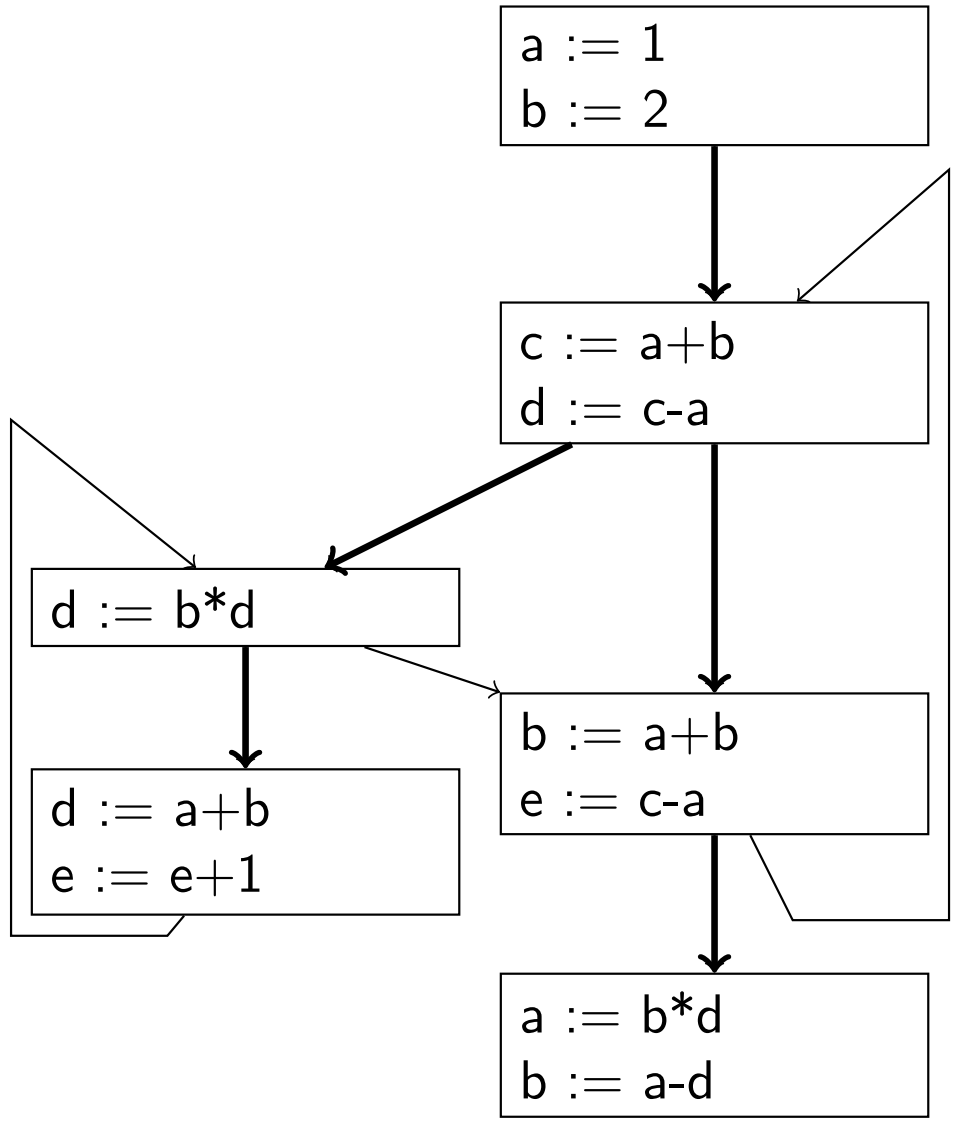
1. Dominanzgrenzen berechnen
   - Kontrollflussgraph und Dominatorbaum notwendig
2. Für jede Variable v:
   - Für jede Definition in einem Block B:
     Platziere ϕ-Funktionen in den Dominanzgrenzen von B
   - Rekursion $\Rightarrow$ Iterierte Dominanzgrenzen
3. Variablen umbenennen: $x \Rightarrow x_i$
   - Implizite Definition x_0 im Startblock

Beispiel

```
 1  a:=1;
 2  b:=2;
 3  while true {
 4     c:=a+b;
 5     if (d=c−a)
 6        while (d=b∗d) {
 7           d:=a+b;
 8           e:=e+1;
 9        }
10     b:=a+b;
11     if (e=c−a) break;
12  }
13  a:=b∗d;
14  b:=a−d;
```

Beispiel

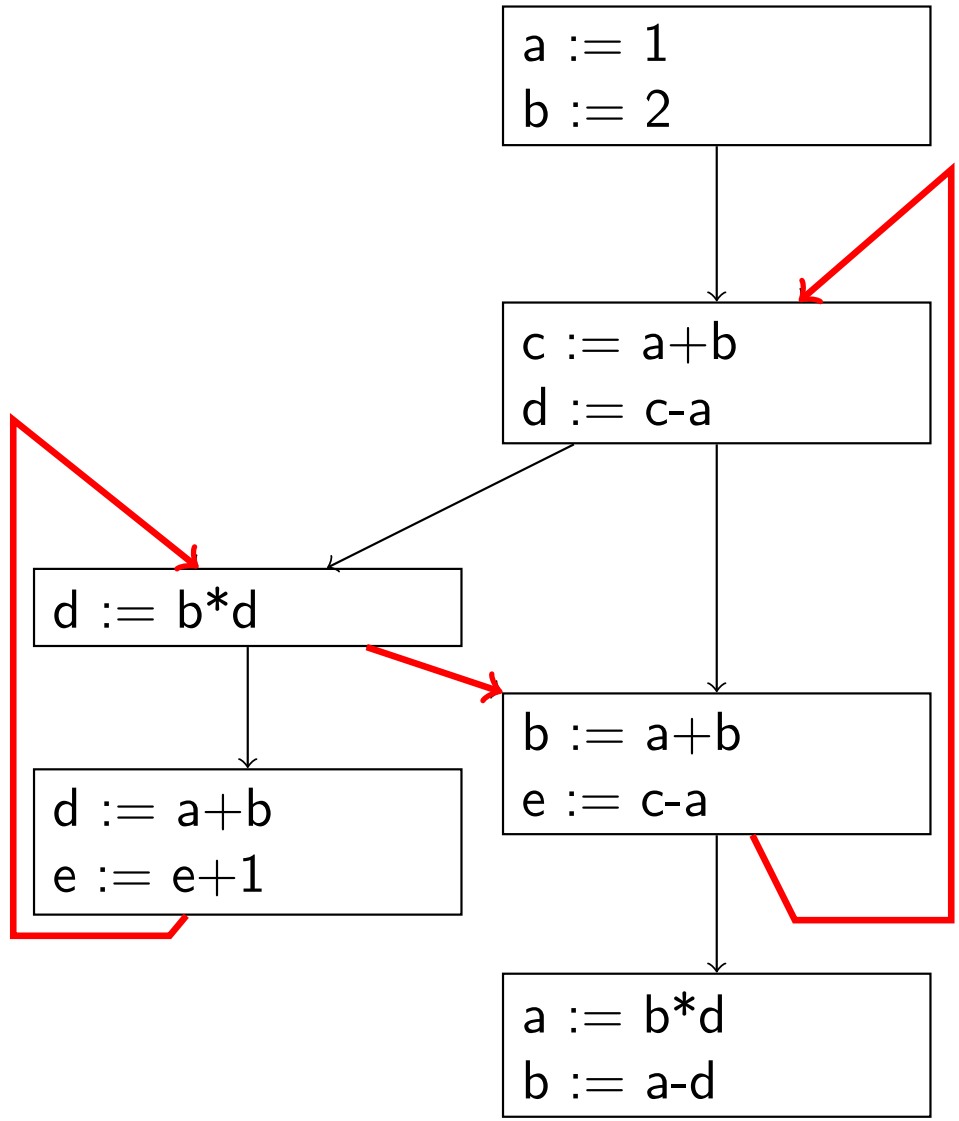
```
 1  a:=1;
 2  b:=2;
 3  while true {
 4     c:=a+b;
 5     if (d=c−a)
 6        while (d=b∗d) {
 7           d:=a+b;
 8           e:=e+1;
 9        }
10     b:=a+b;
11     if (e=c−a) break;
12  }
13  a:=b∗d;
14  b:=a−d;
```



1a. Kontrollflussgraph und Dominatorbaum aufbauen

Beispiel

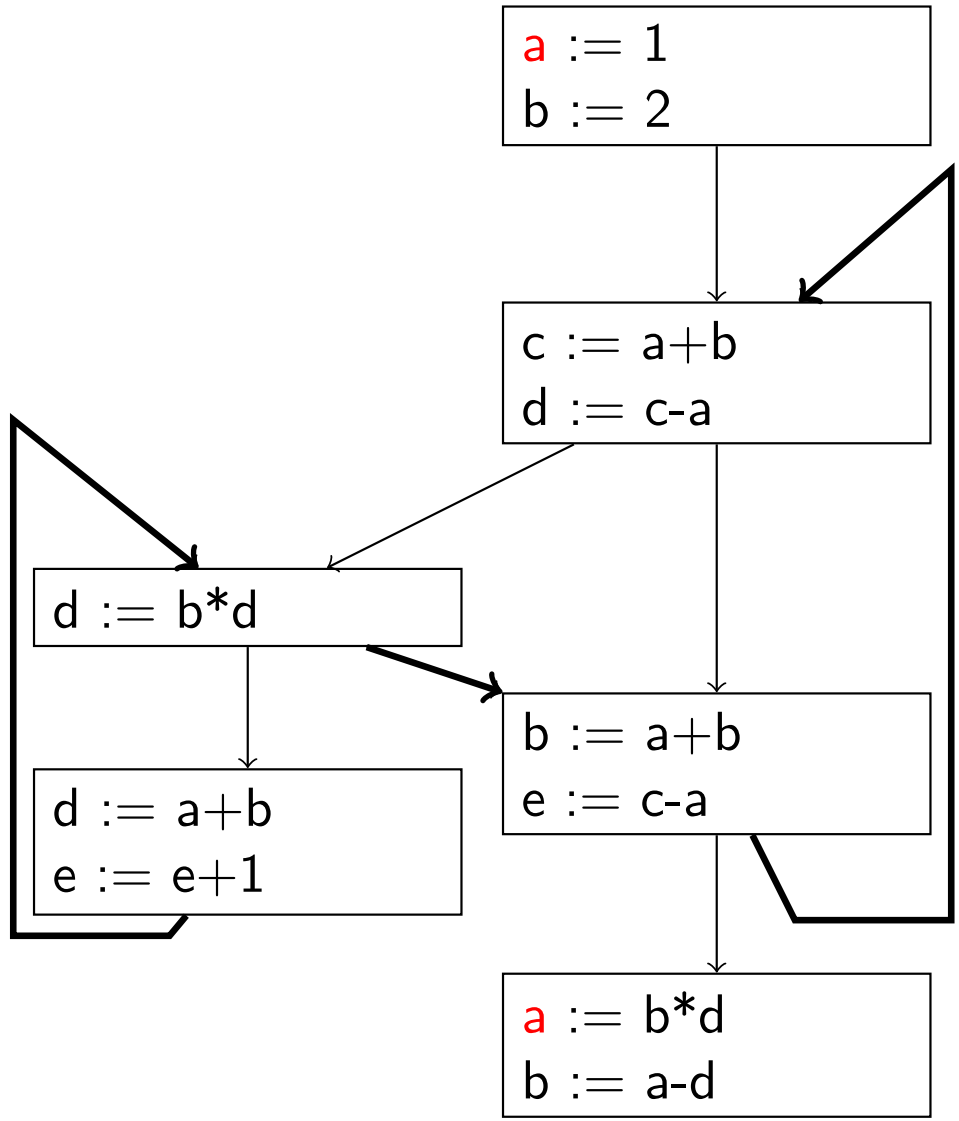
```
 1  a:=1;
 2  b:=2;
 3  while true {
 4     c:=a+b;
 5     if (d=c−a)
 6        while (d=b∗d) {
 7           d:=a+b;
 8           e:=e+1;
 9        }
10     b:=a+b;
11     if (e=c−a) break;
12  }
13  a:=b∗d;
14  b:=a−d;
```



1b. Dominanzgrenzen berechnen

Beispiel

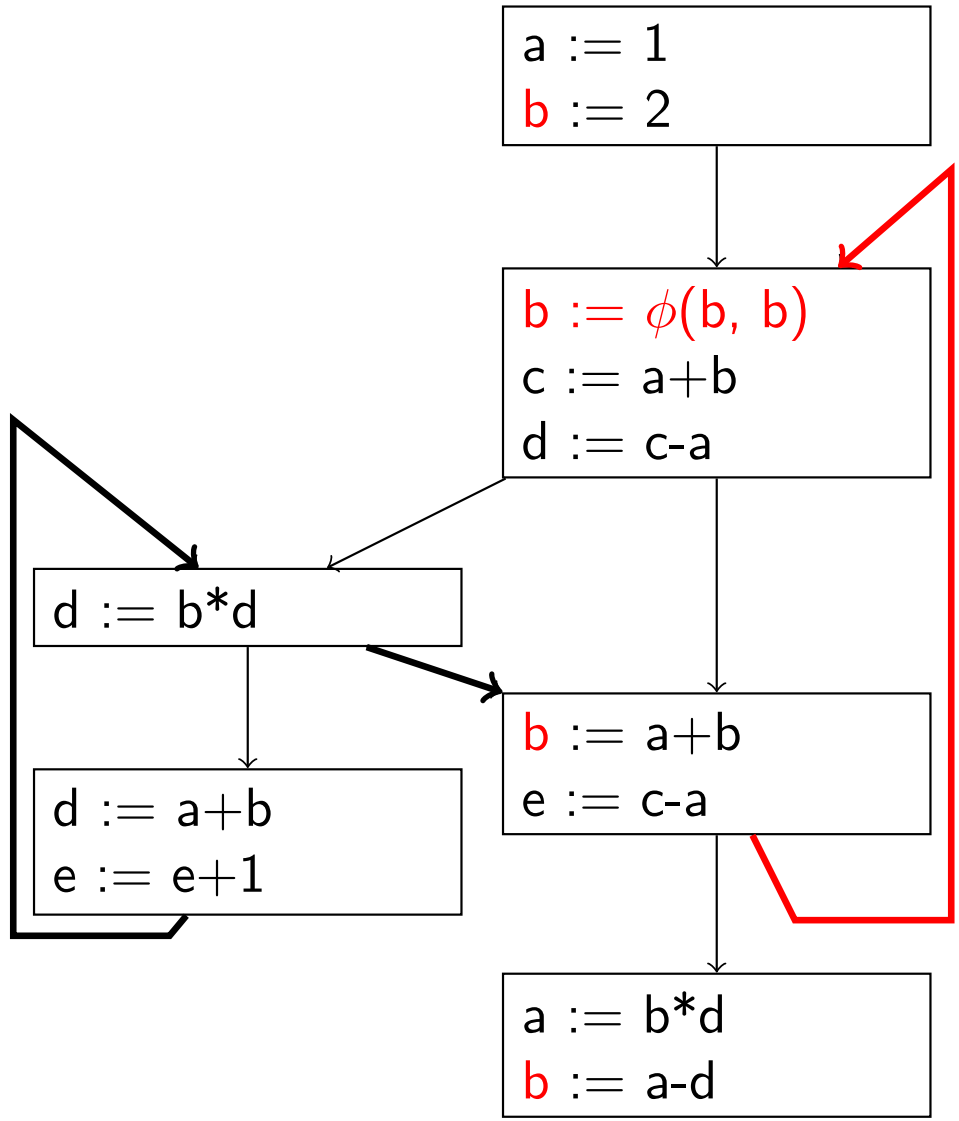
```
 1  a:=1;
 2  b:=2;
 3  while true {
 4     c:=a+b;
 5     if (d=c−a)
 6        while (d=b∗d) {
 7           d:=a+b;
 8           e:=e+1;
 9        }
10     b:=a+b;
11     if (e=c−a) break;
12  }
13  a:=b∗d;
14  b:=a−d;
```



2a. Keine ϕ für a notwendig

Beispiel

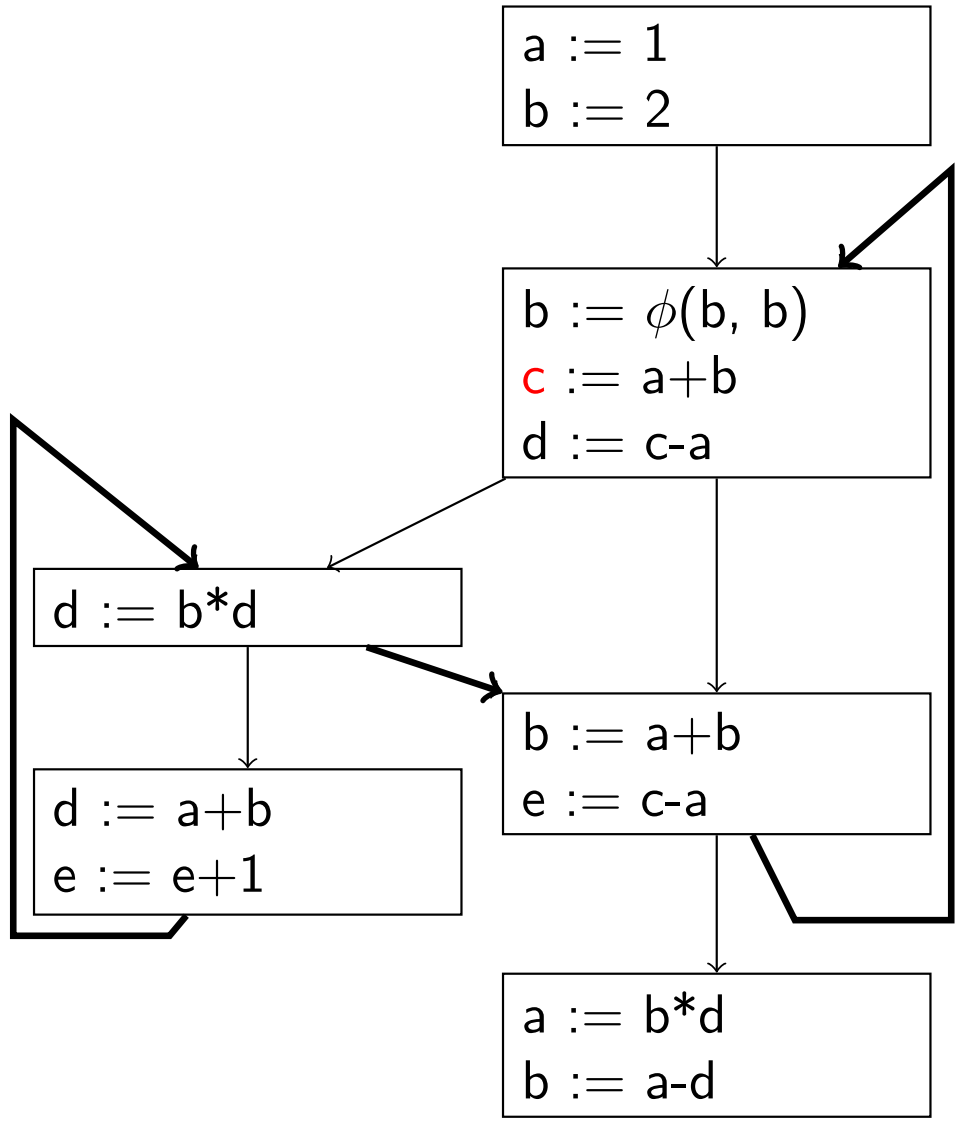
```
 1  a:=1;
 2  b:=2;
 3  while true {
 4     c:=a+b;
 5     if (d=c−a)
 6        while (d=b∗d) {
 7           d:=a+b;
 8           e:=e+1;
 9        }
10     b:=a+b;
11     if (e=c−a) break;
12  }
13  a:=b∗d;
14  b:=a−d;
```



2b. Ein ϕ für b zu platzieren

Beispiel

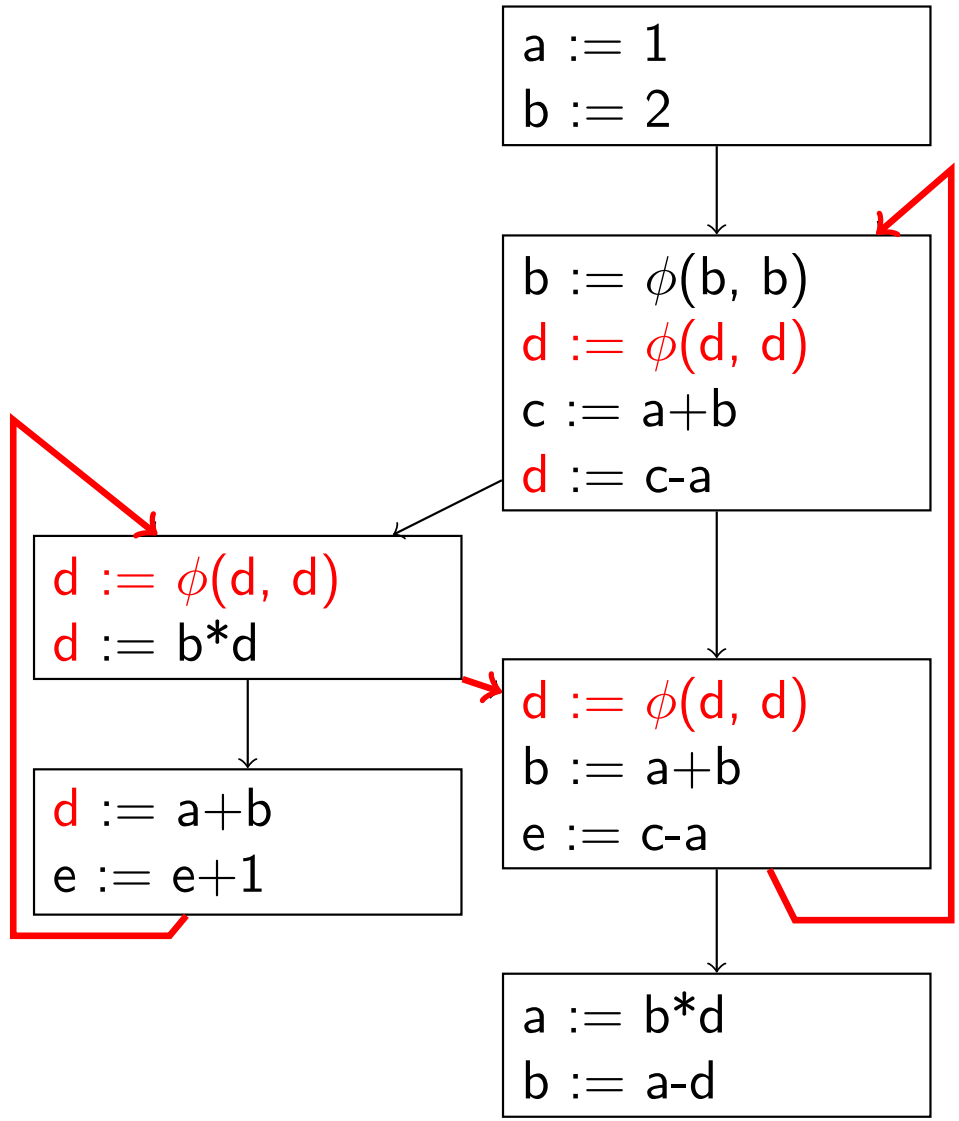
```
 1  a:=1;
 2  b:=2;
 3  while true {
 4     c:=a+b;
 5     if (d=c−a)
 6        while (d=b∗d) {
 7           d:=a+b;
 8           e:=e+1;
 9        }
10     b:=a+b;
11     if (e=c−a) break;
12  }
13  a:=b∗d;
14  b:=a−d;
```



2c. Keine ϕ für c notwendig

Beispiel

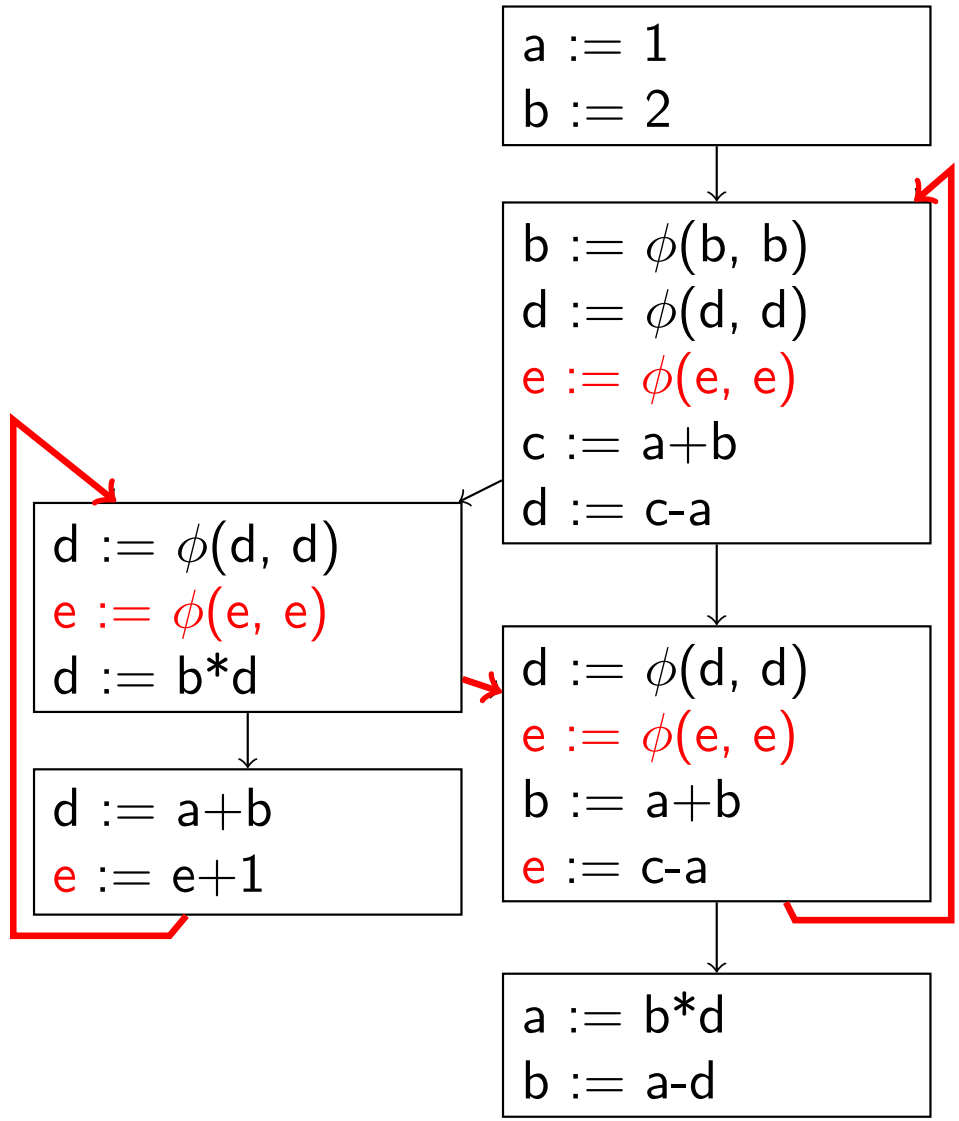
```
 1  a:=1;
 2  b:=2;
 3  while true {
 4     c:=a+b;
 5     if (d=c−a)
 6        while (d=b∗d) {
 7           d:=a+b;
 8           e:=e+1;
 9        }
10     b:=a+b;
11     if (e=c−a) break;
12  }
13  a:=b∗d;
14  b:=a−d;
```



2d. Drei ϕ für d

Beispiel

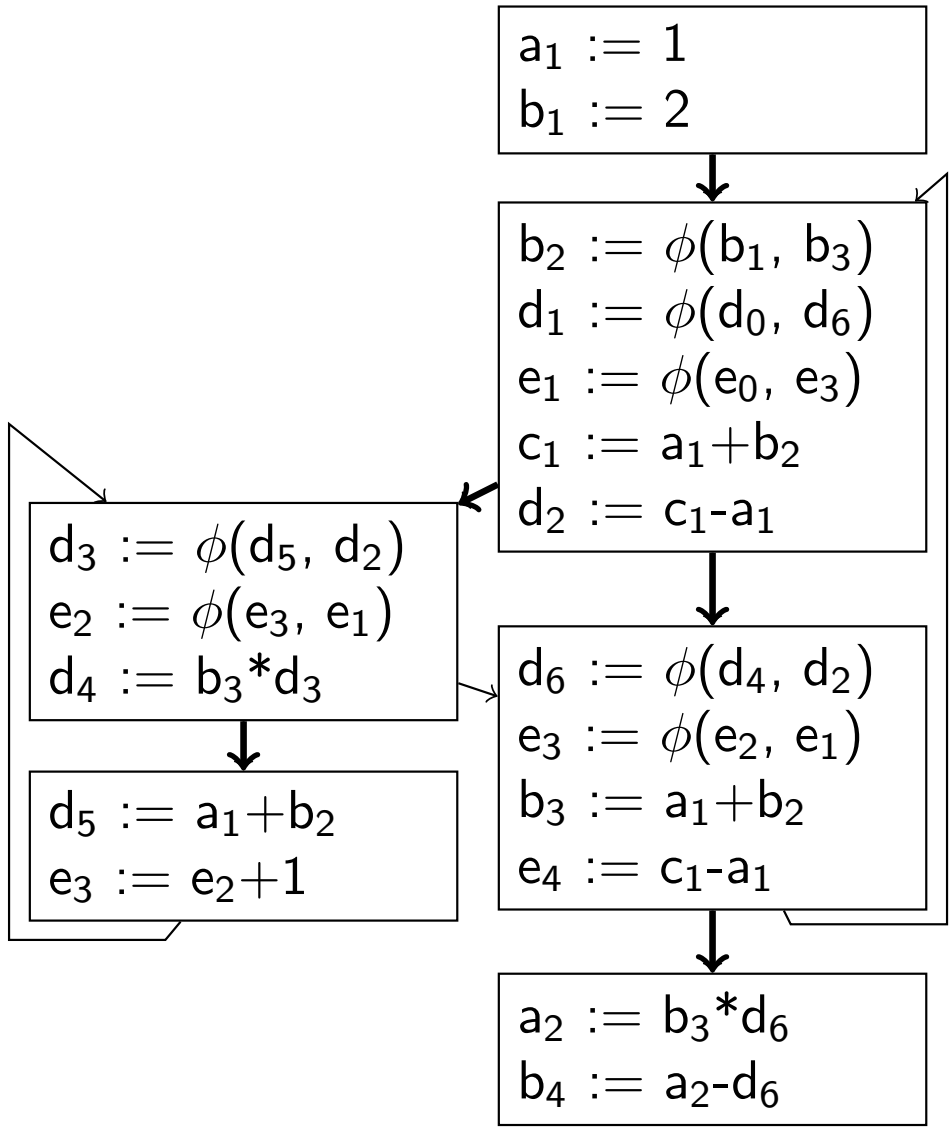
```
 1  a:=1;
 2  b:=2;
 3  while true {
 4     c:=a+b;
 5     if (d=c−a)
 6        while (d=b∗d) {
 7           d:=a+b;
 8           e:=e+1;
 9        }
10     b:=a+b;
11     if (e=c−a) break;
12  }
13  a:=b∗d;
14  b:=a−d;
```



2e. Drei ϕ für e

Beispiel

```
 1  a:=1;
 2  b:=2;
 3  while true {
 4     c:=a+b;
 5     if (d=c−a)
 6        while (d=b∗d) {
 7           d:=a+b;
 8           e:=e+1;
 9        }
10     b:=a+b;
11     if (e=c−a) break;
12  }
13  a:=b∗d;
14  b:=a−d;
```



3. Variablen umbenennen

On-The-Fly SSA-Aufbau (nach libFirm)

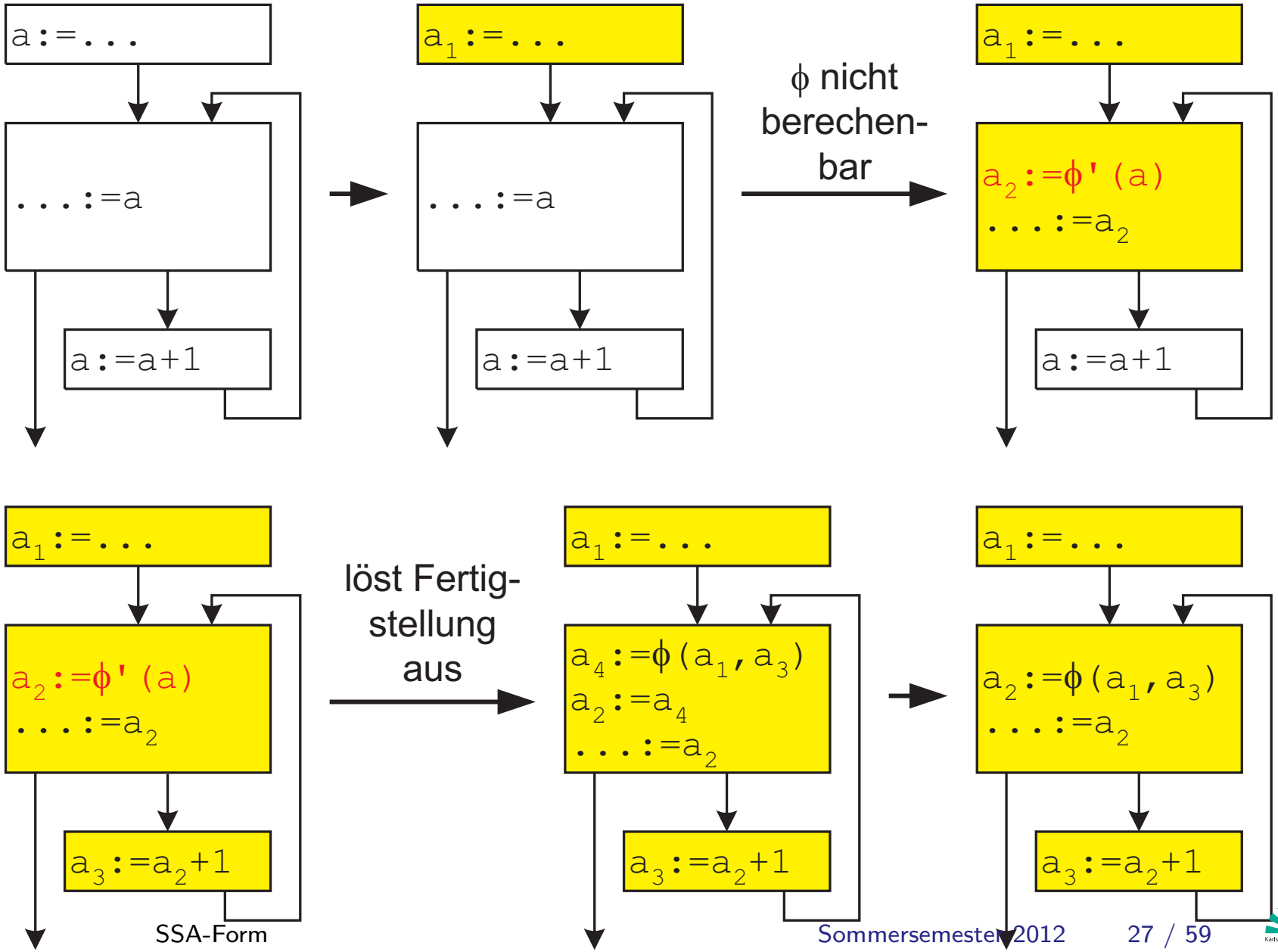
Annahme: Aufbau direkt aus dem AST:

- CFG unvollständig (wird gleichzeitig konstruiert)
- $\rightarrow$ kein Dominanzbaum (oder -grenzen)

Vorgehen: „Platziere ϕs bei Bedarf“

- Durchlaufe Block von Anfang bis Ende
- Beginne mit leerer Variable $\rightarrow$ Definition Map
- Bei Variablenzuweisung: In Map vermerken
- Bei Variablenbenutzung:
  - Lese Definition aus Map, benutze als Operand falls vorhanden
  - Falls nicht vorhanden erzeuge (vorläufiges) ϕ', Vermerke ϕ' in Map
  - Suche ϕ-Argumente (rekursiv) in Maps der Vorgängerblöcke
  - Falls keine Definition im Startblock benutze undefinierten Wert
- Am Grundblockende erweitere Argumente von ϕ-Operationen in Nachfolgeblöcken.

unbekannte Vorgänger: Beispiel



Unnötige ϕ-Funktionen eliminieren

Feststellung:

Es werden viele „unnötige“ ϕ-Funktionen erzeugt weil Vorgänger noch nicht feststehen.

Verbesserung:

Betrachte Funktionen $p = \phi(i_0, i_1, \ldots)$ nachdem alle Argumente feststehen:

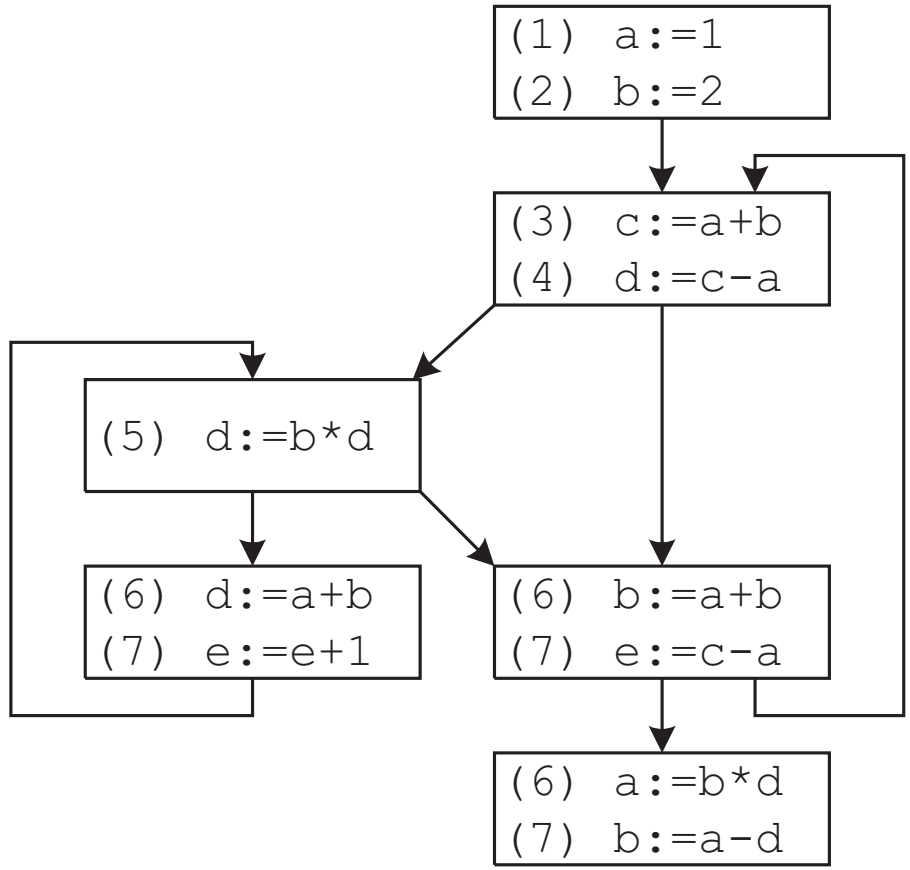
- Gibt es einen Wert x, so dass alle i_k gleich x oder gleich p sind, so ersetze p durch x.
- Rekursion: Betrachte nach Ersetzung alle (ϕ-)Verwender von x

Beispiele:

$$\begin{aligned} p = \phi(a_0) &\Rightarrow p = a_0 \\ p = \phi(b_0, b_0, b_0) &\Rightarrow p = b_0 \\ p = \phi(p, c_0, p) &\Rightarrow p = c_0 \end{aligned}$$

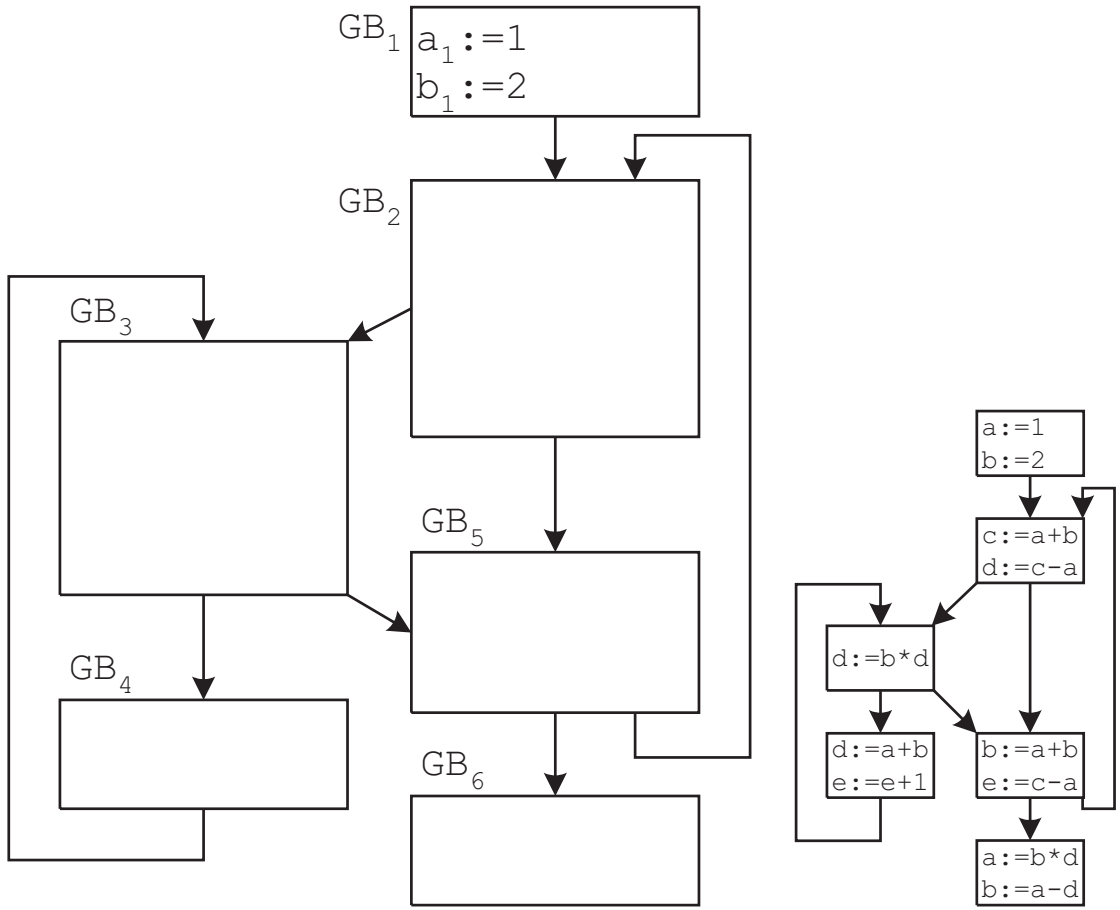
Beispielprogramm und Grundblockgraph

```
 1  a:=1;
 2  b:=2;
 3  while true {
 4     c:=a+b;
 5     if (d=c−a)
 6        while (d=b∗d) {
 7           d:=a+b;
 8           e:=e+1;
 9        }
10     b:=a+b;
11     if (e=c−a) break;
12  }
13  a:=b∗d;
14  b:=a−d;
```



SSA-Aufbau GB_1

Vermerke Definition von a und b in Map.

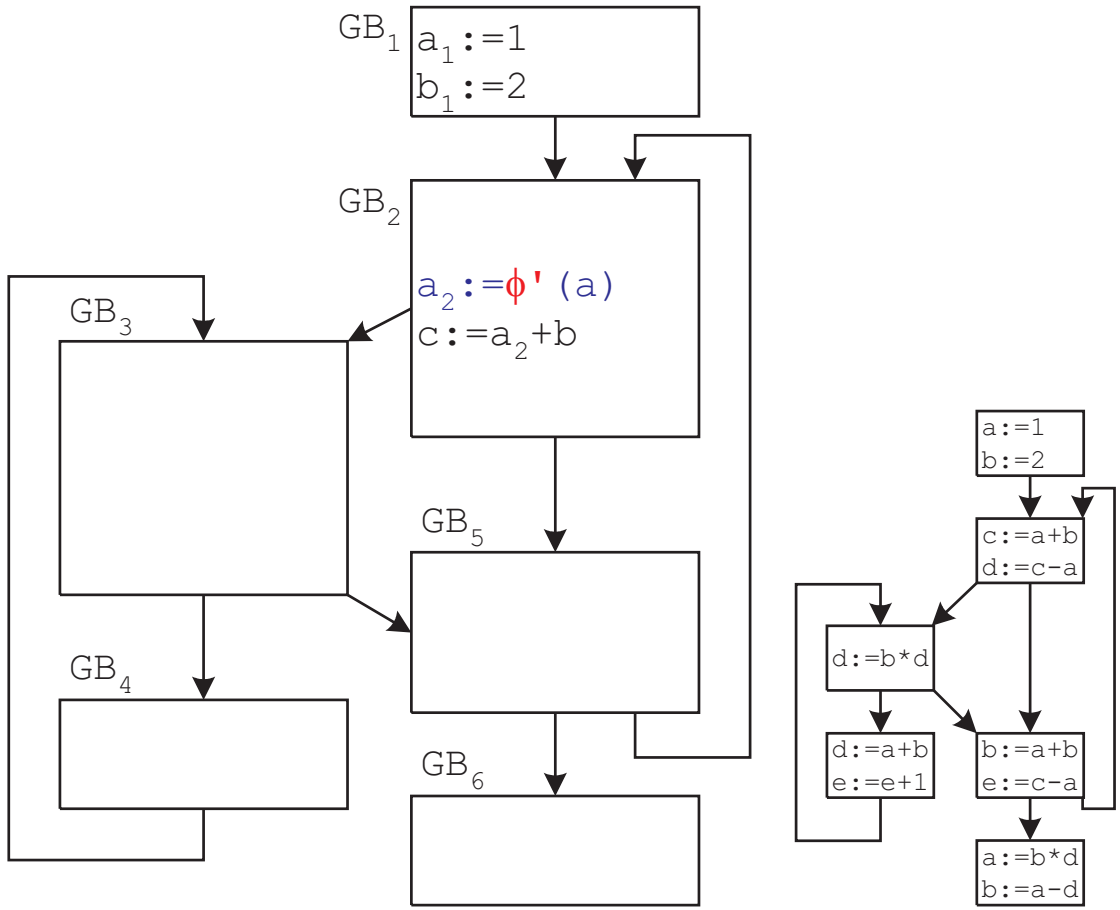


SSA-Aufbau GB_2

Lesen von a

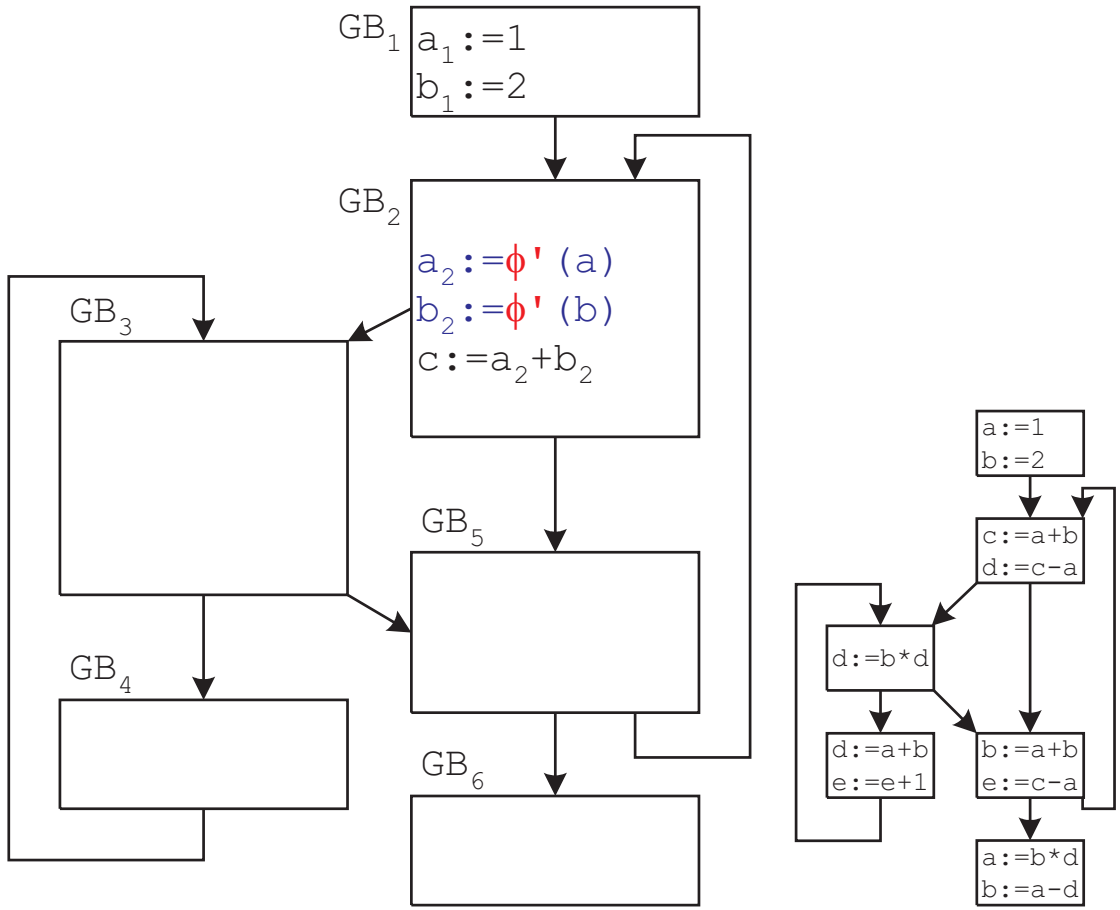
$\Rightarrow$ erzeugt ϕ' für a

...



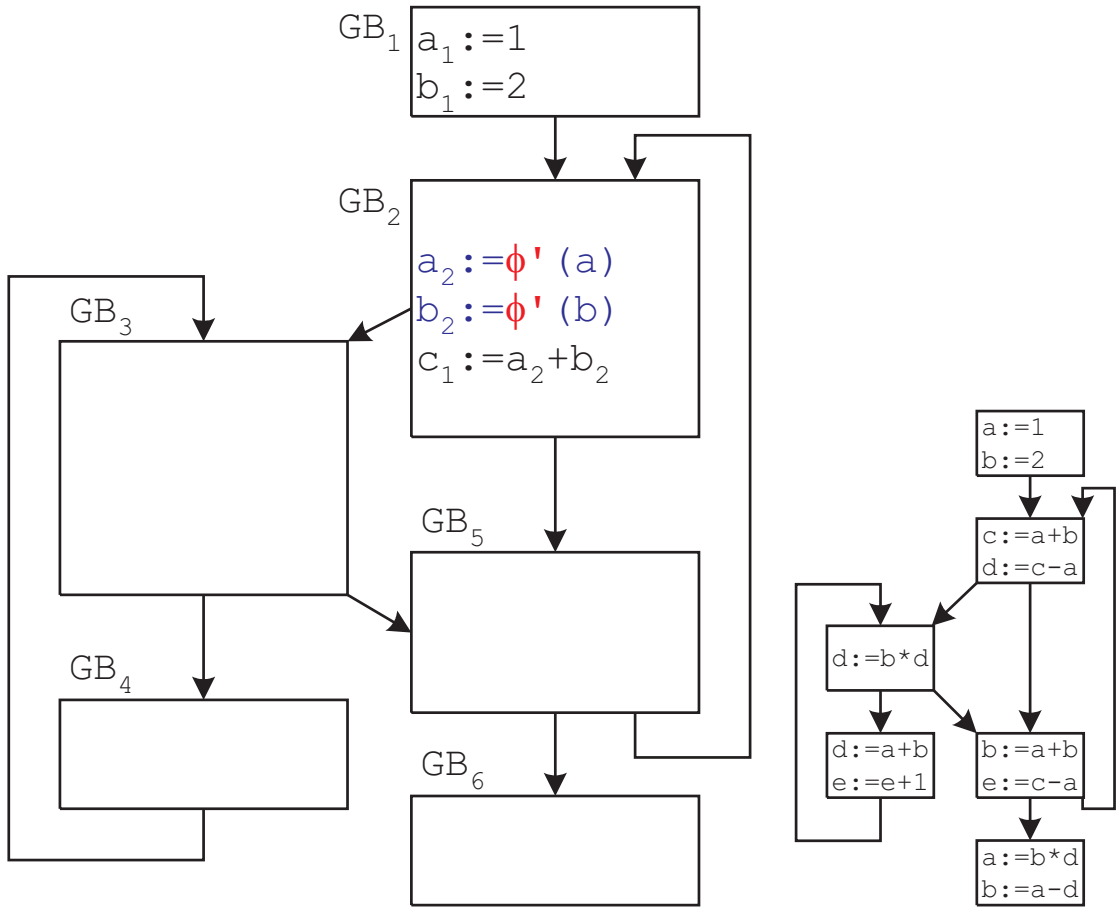
SSA-Aufbau GB_2

. . . dann für b . . .



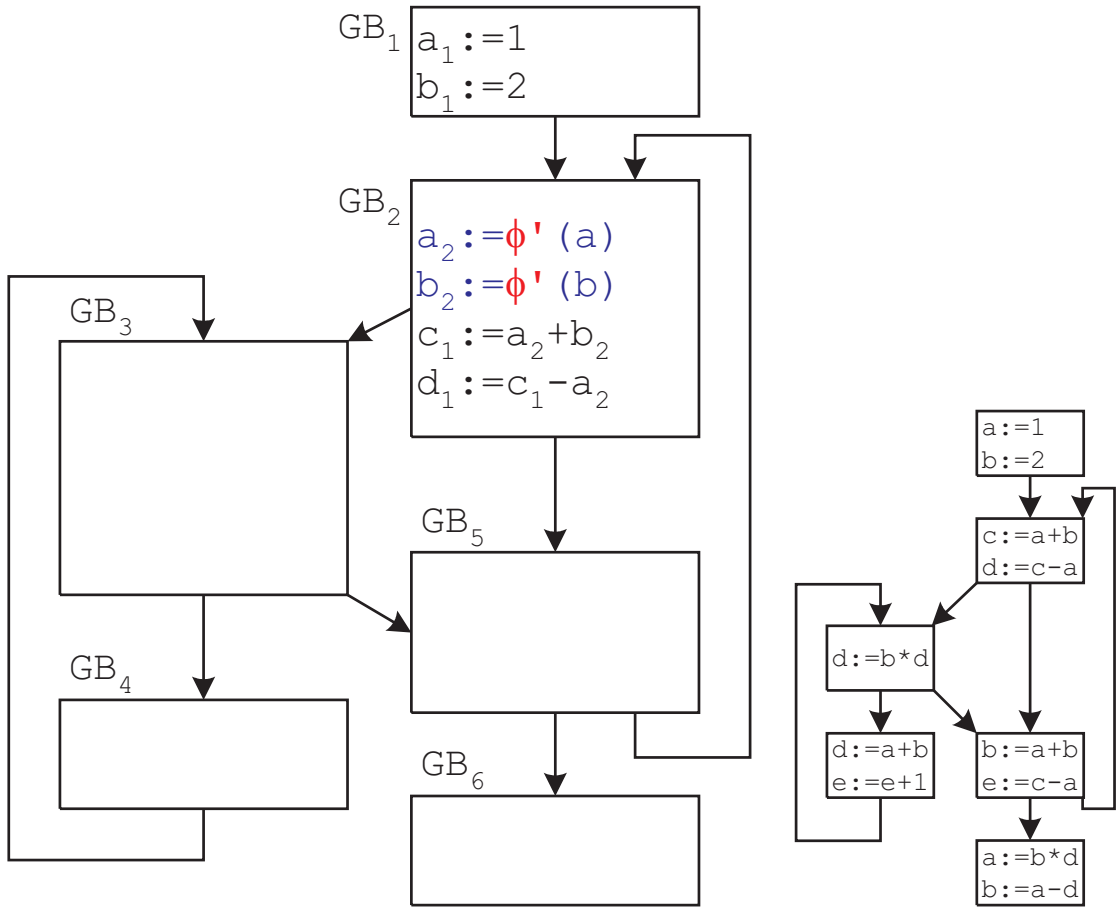
SSA-Aufbau GB_2

. . . vermerke Definition von c.

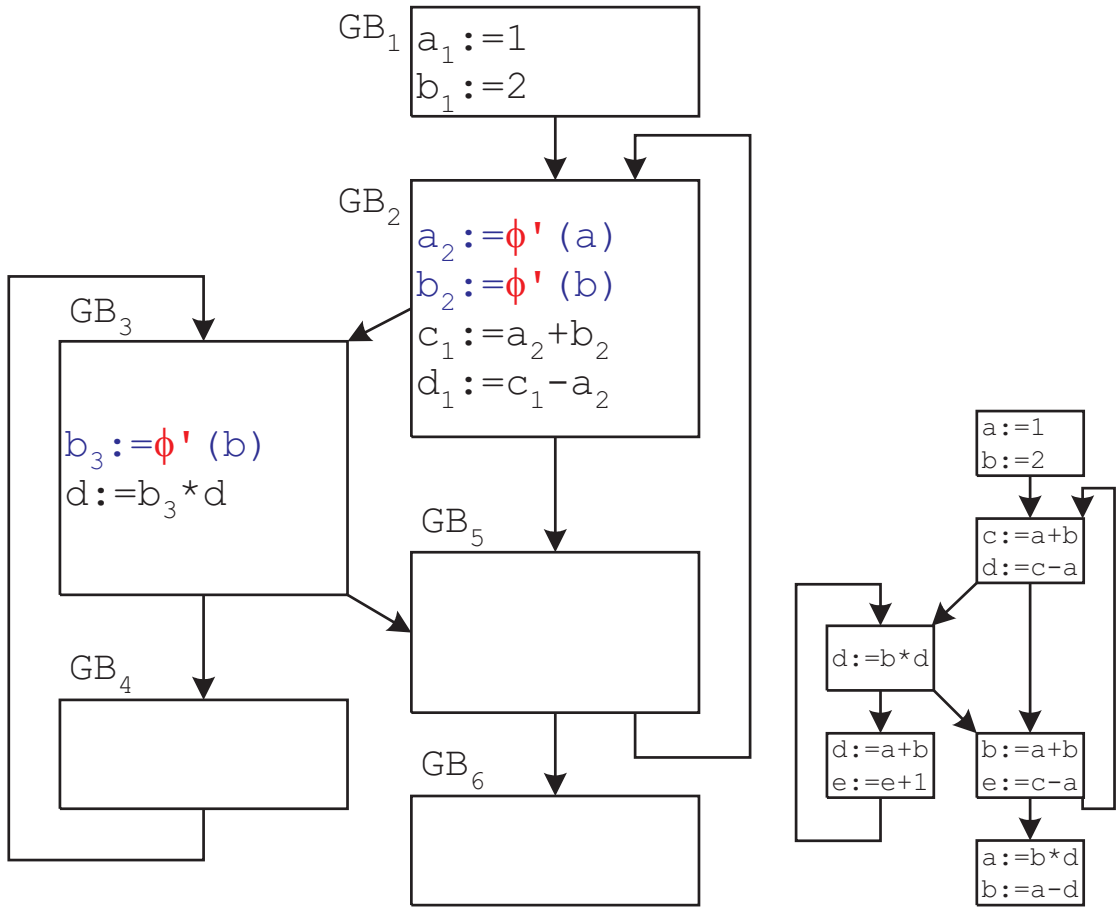


SSA-Aufbau GB_2

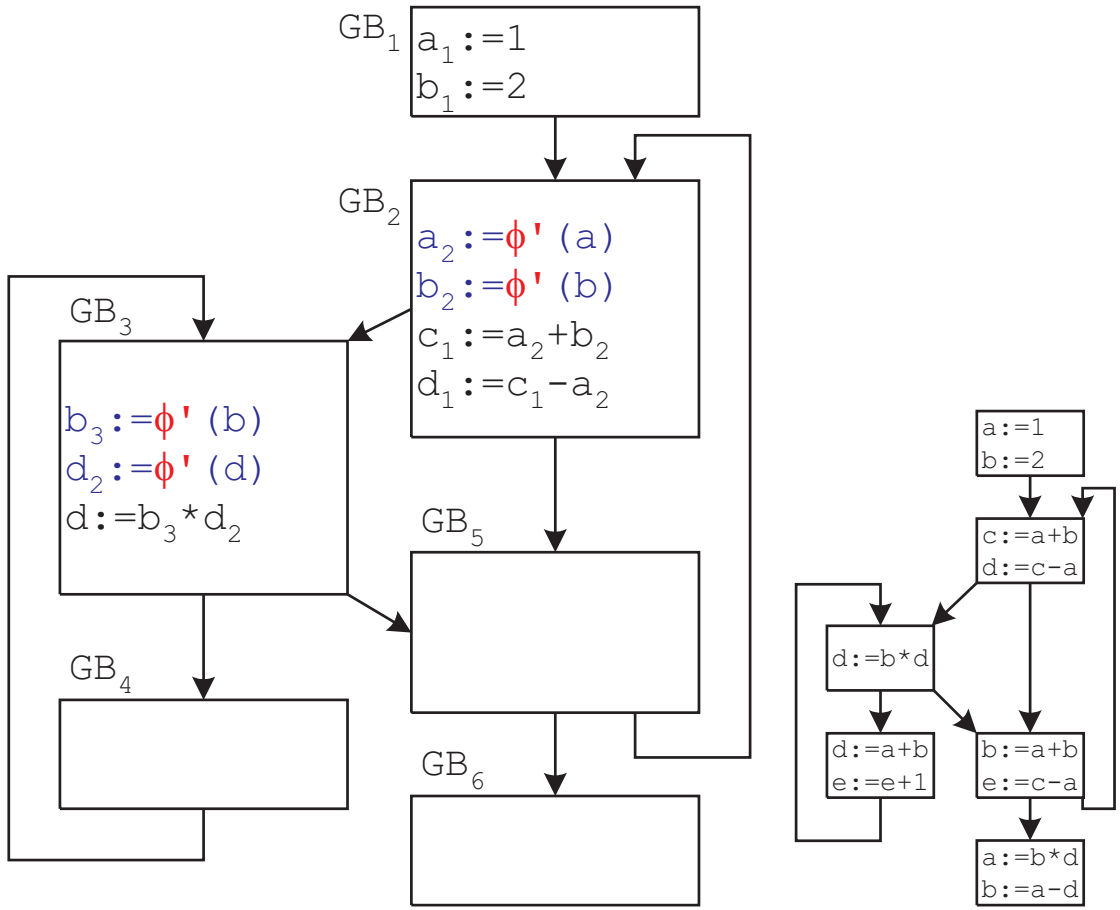
Aufbau für $d := c - a$ analog



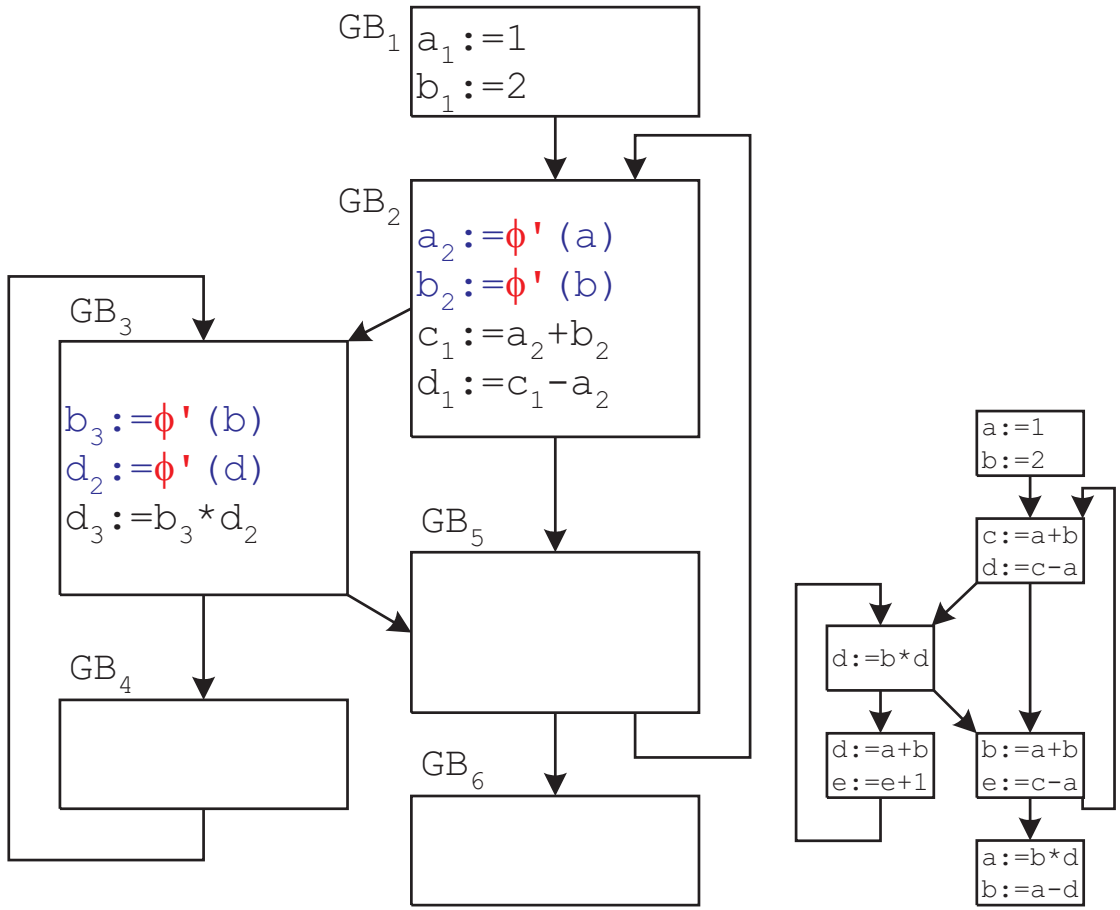
SSA-Aufbau GB_3



SSA-Aufbau GB_3

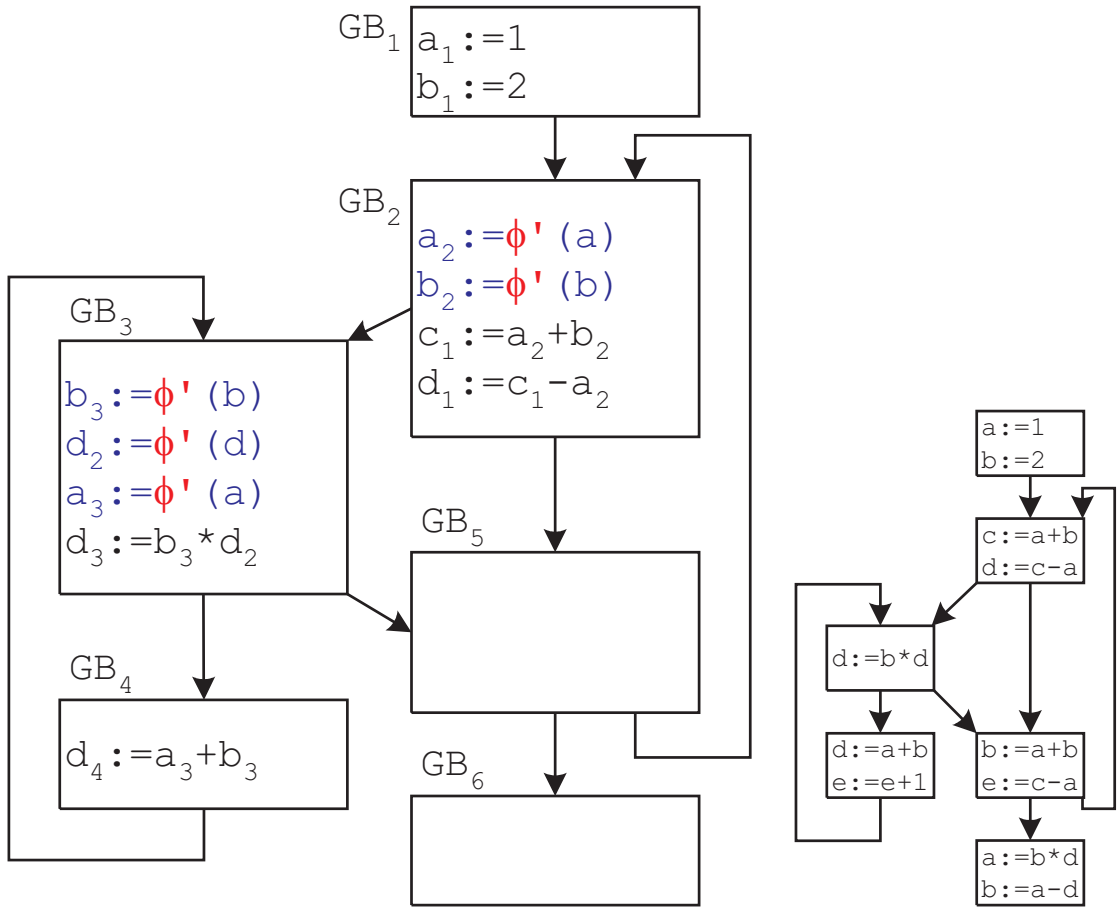


SSA-Aufbau GB_3

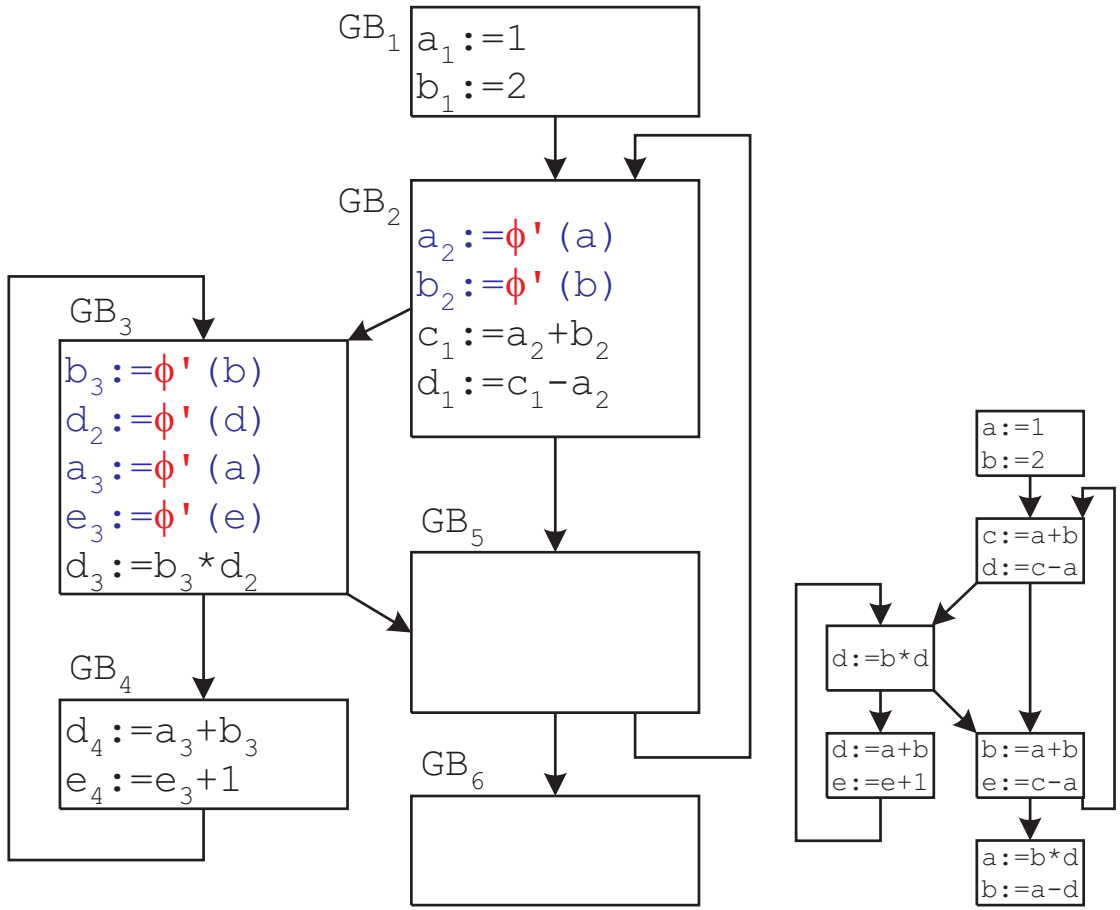


SSA-Aufbau GB_4

Lesen von a in GB_4 führt zu rekursiver Suche in GB_3.
Dort wird neue ϕ'-Funktion für *a* erzeugt.



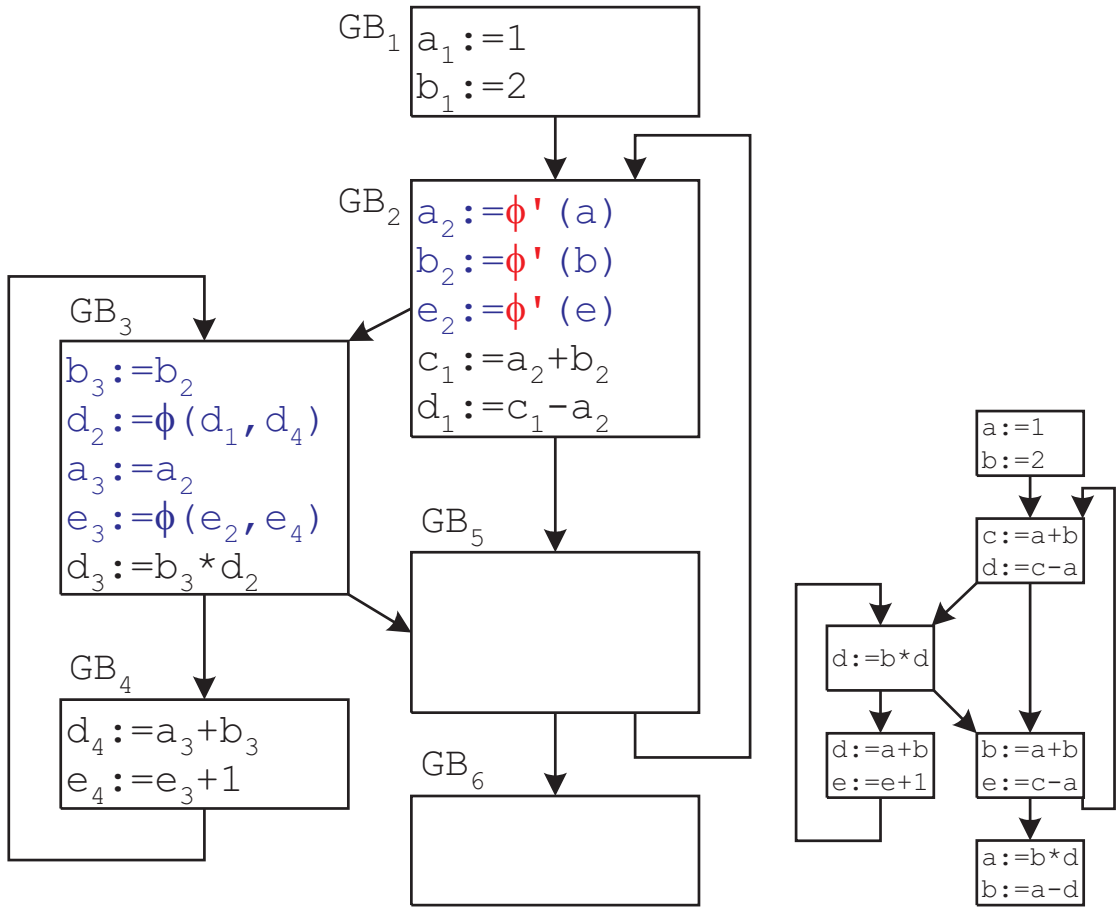
SSA-Aufbau GB_4



SSA-Aufbau GB_4

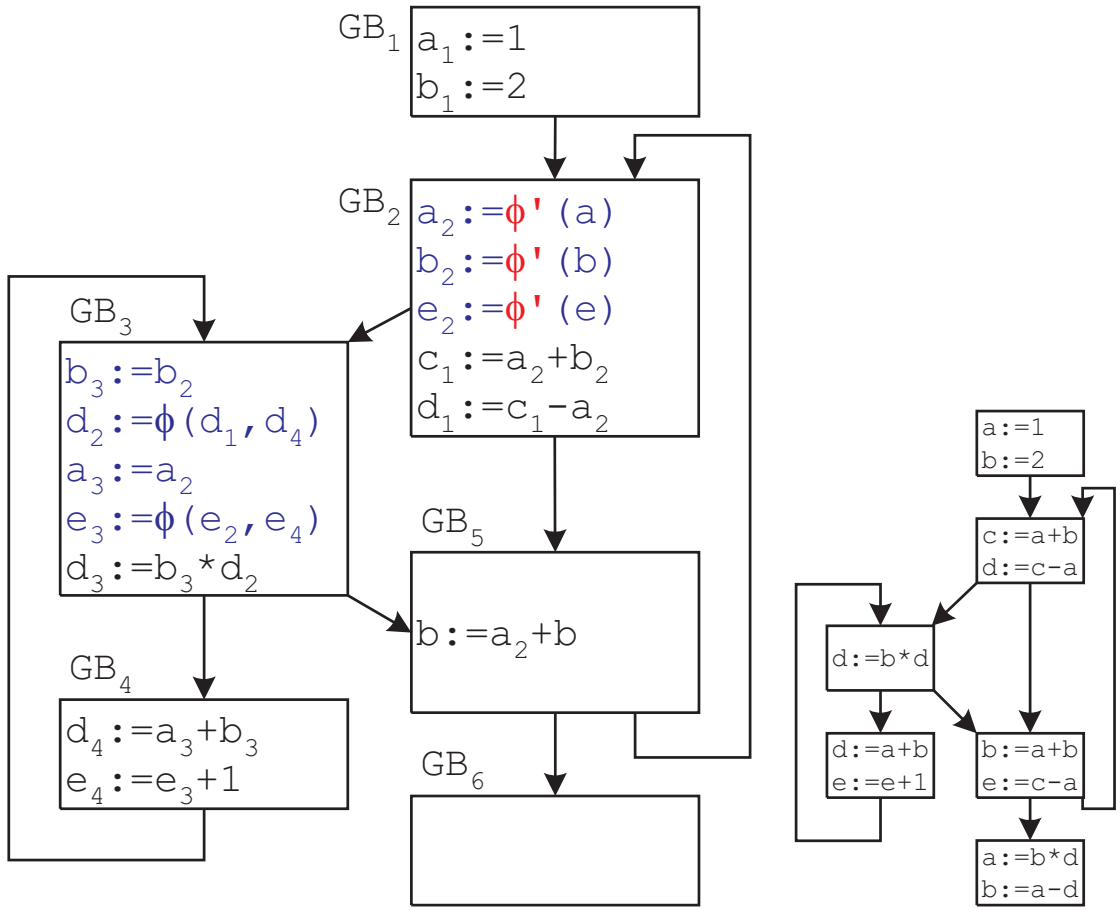
Jetzt alle Vorgänger von GB_3 in SSA-Form: ϕ-Funktionen werden berechnet.

Für e wird rekursiv eine ϕ'-Funktion in GB_2 eingesetzt.

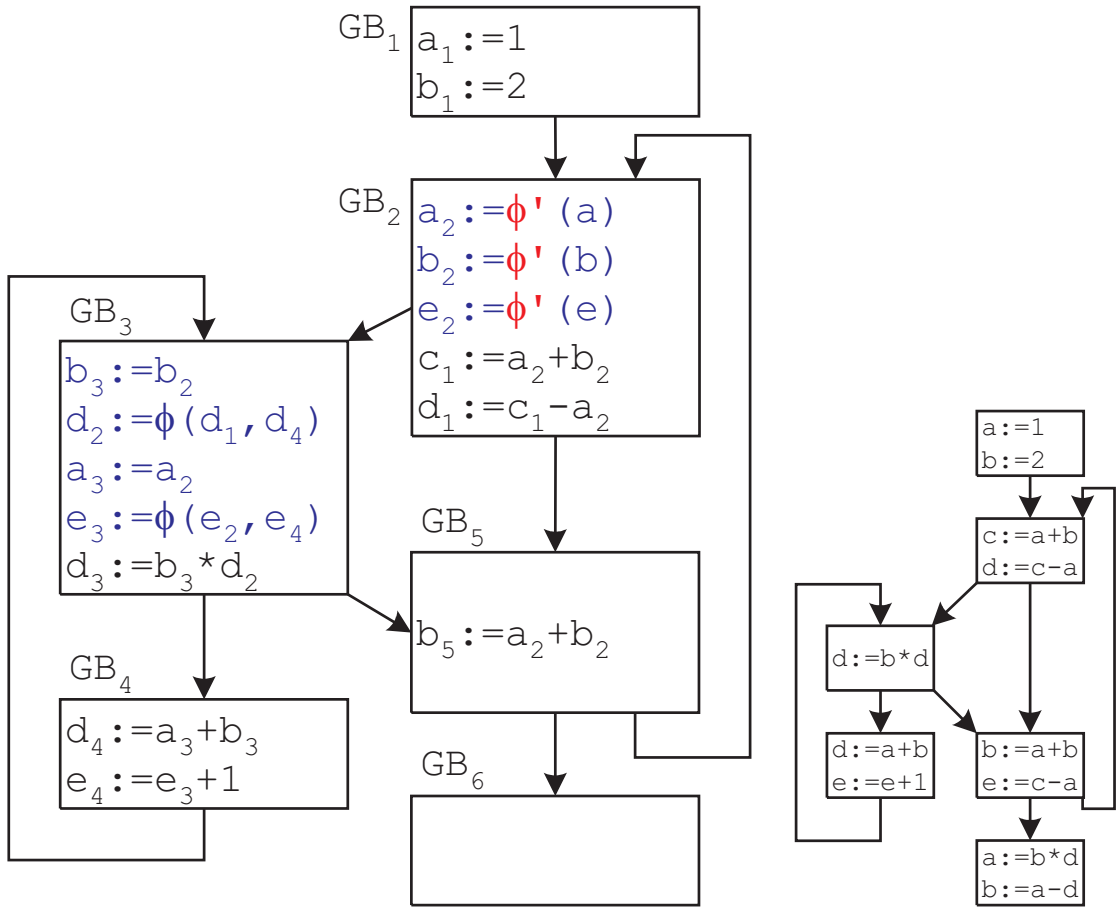


SSA-Aufbau GB_5

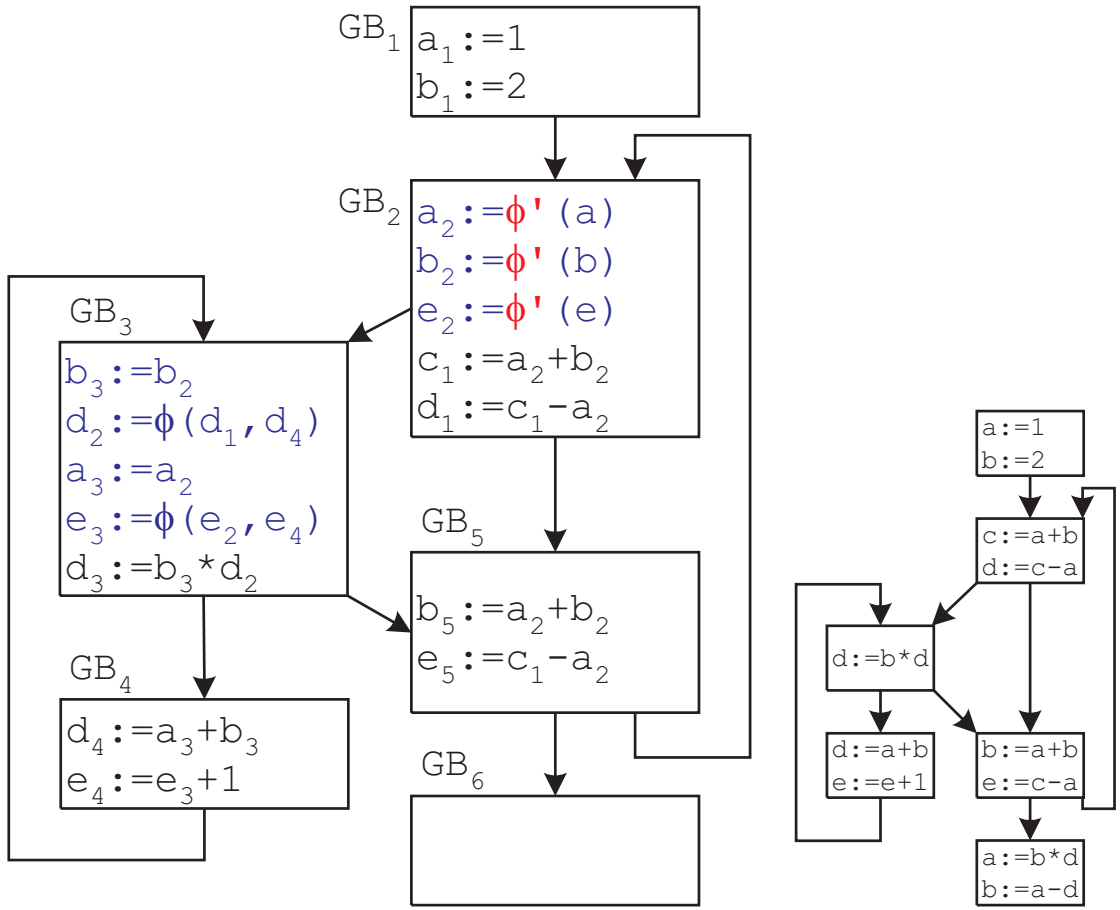
Lesen von a in GB_5 überspringt Kopien, findet eindeutige Definition:
keine ϕ-Funktion nötig.



SSA-Aufbau GB_5



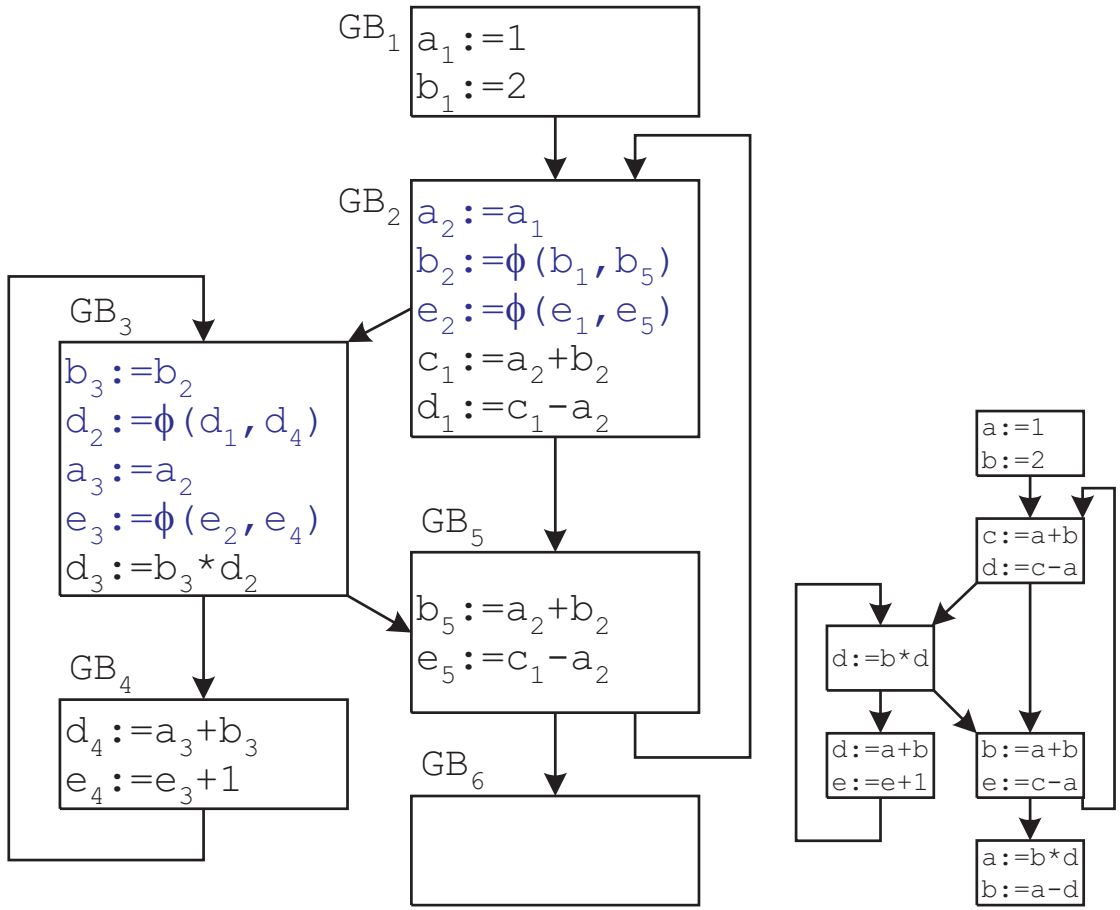
SSA-Aufbau GB_5



SSA-Aufbau GB_5

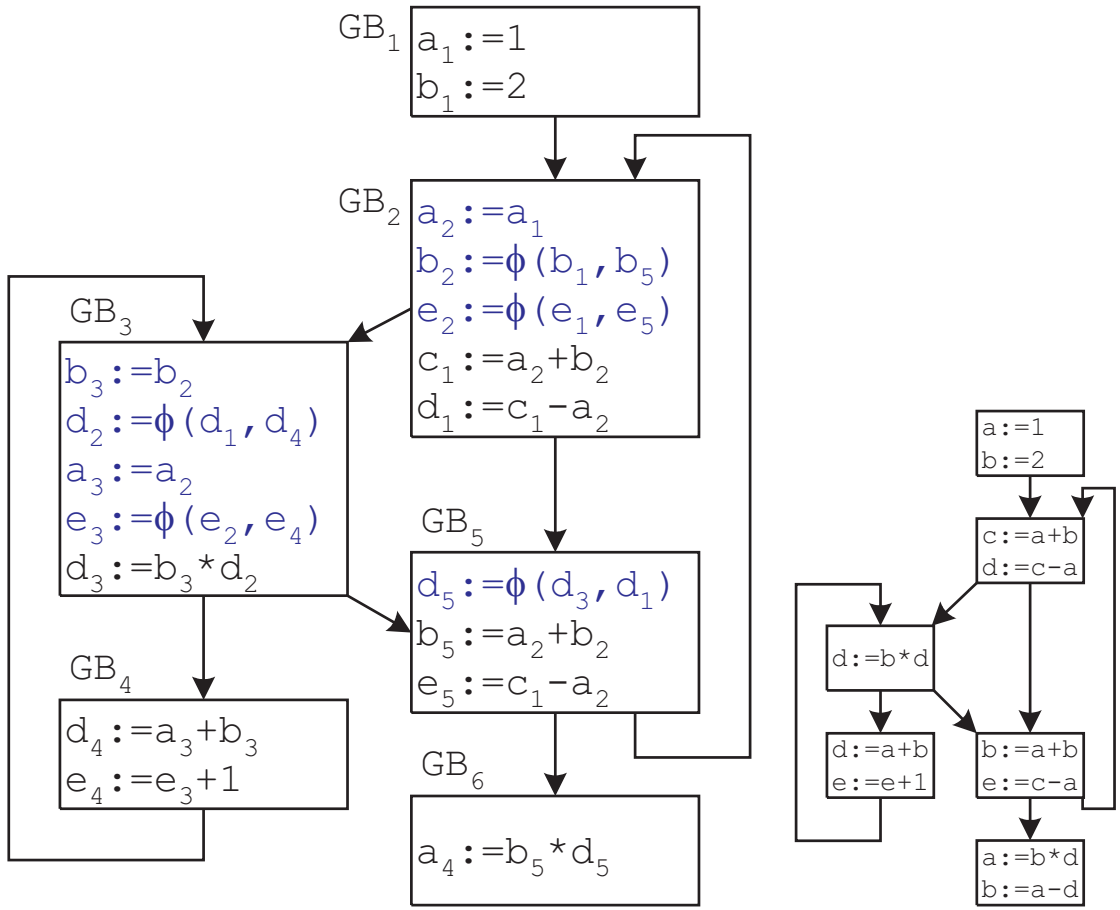
Jetzt alle Vorgänger von GB_2 in SSA-Form: ϕ-Funktionen werden berechnet.

Algorithmus bemerkt:
e ist uninitialisiert! Annahme: Wert e_1

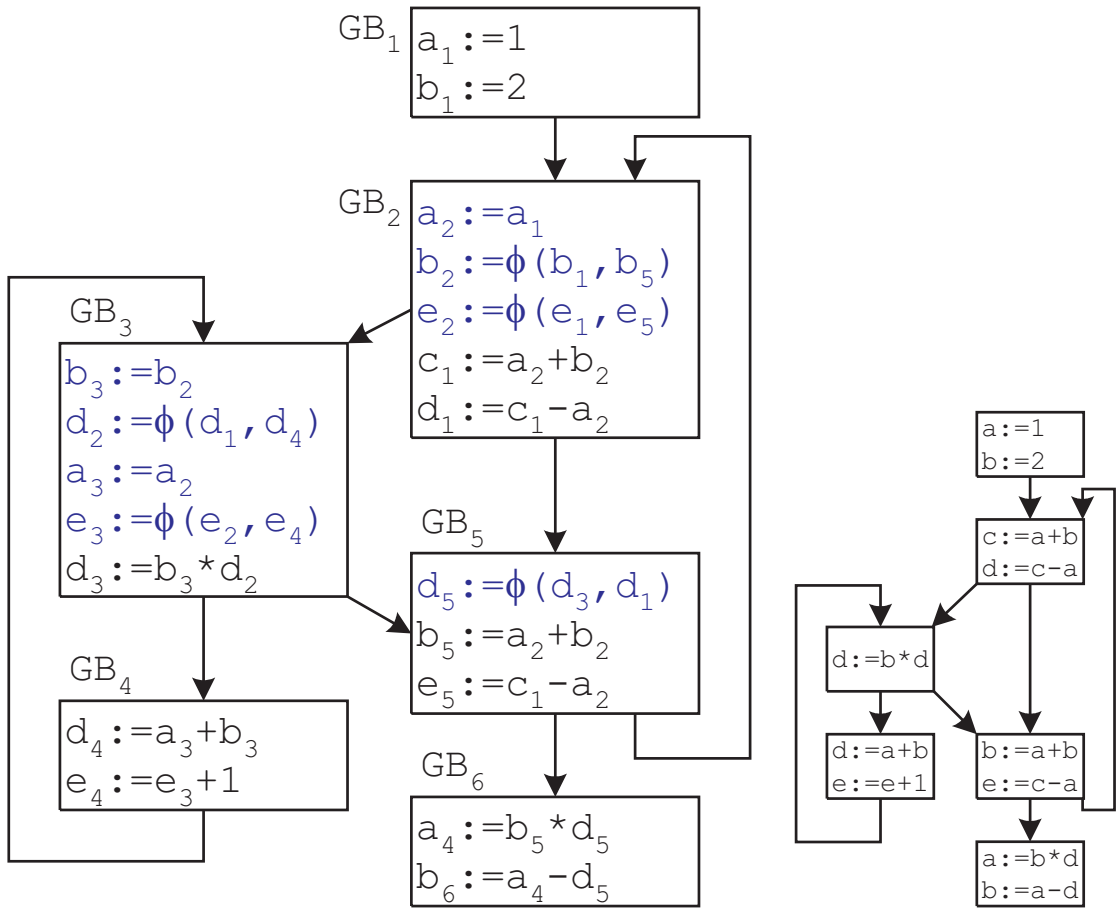


SSA-Aufbau GB_6

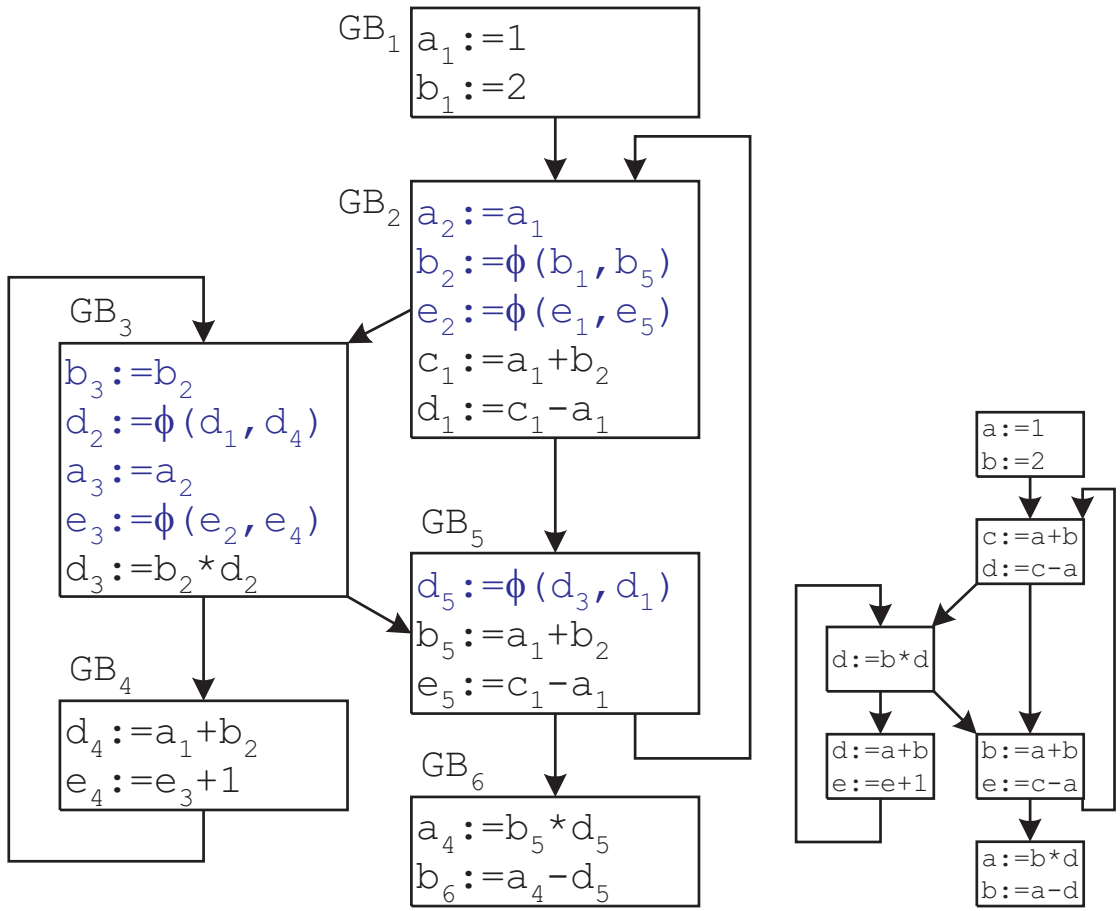
Rekursiver Suchen nach d in GB_6 setzt komplette ϕ-Funktion d_5 in GB_5 ein



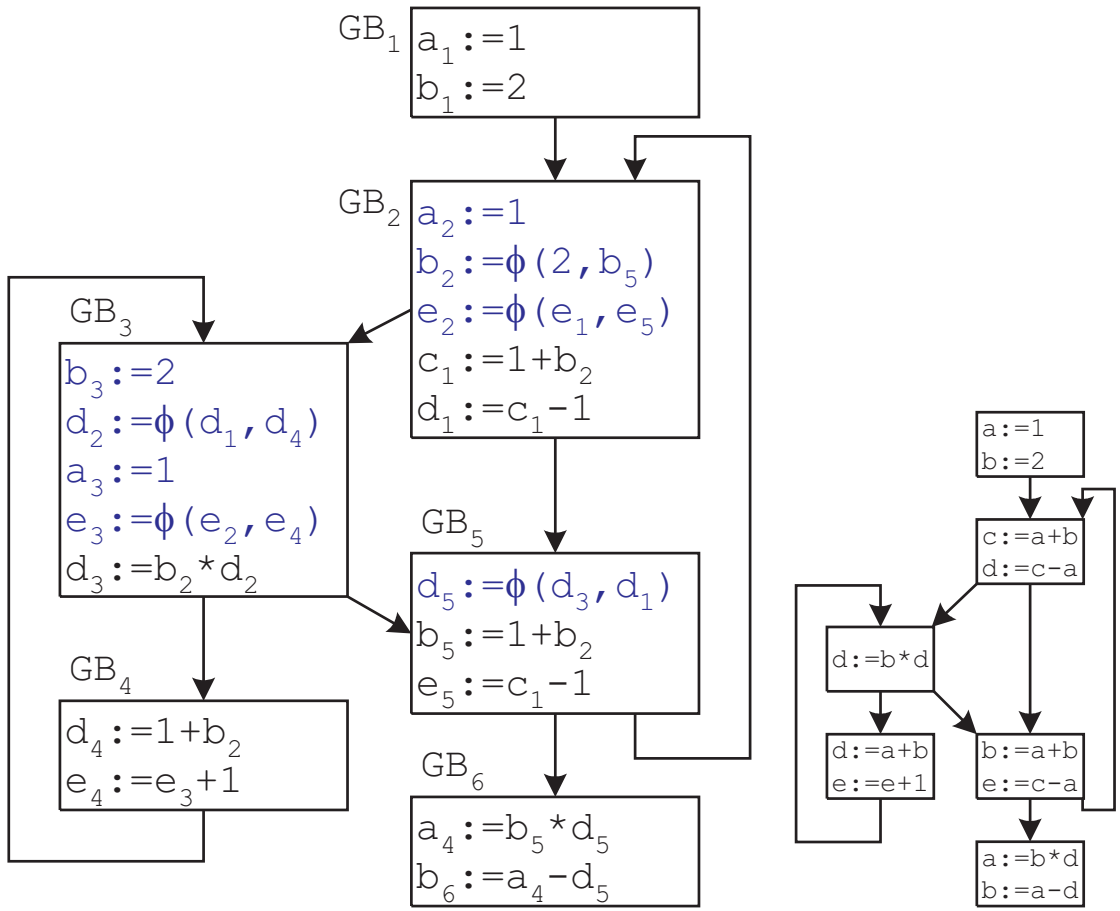
SSA-Aufbau GB_6



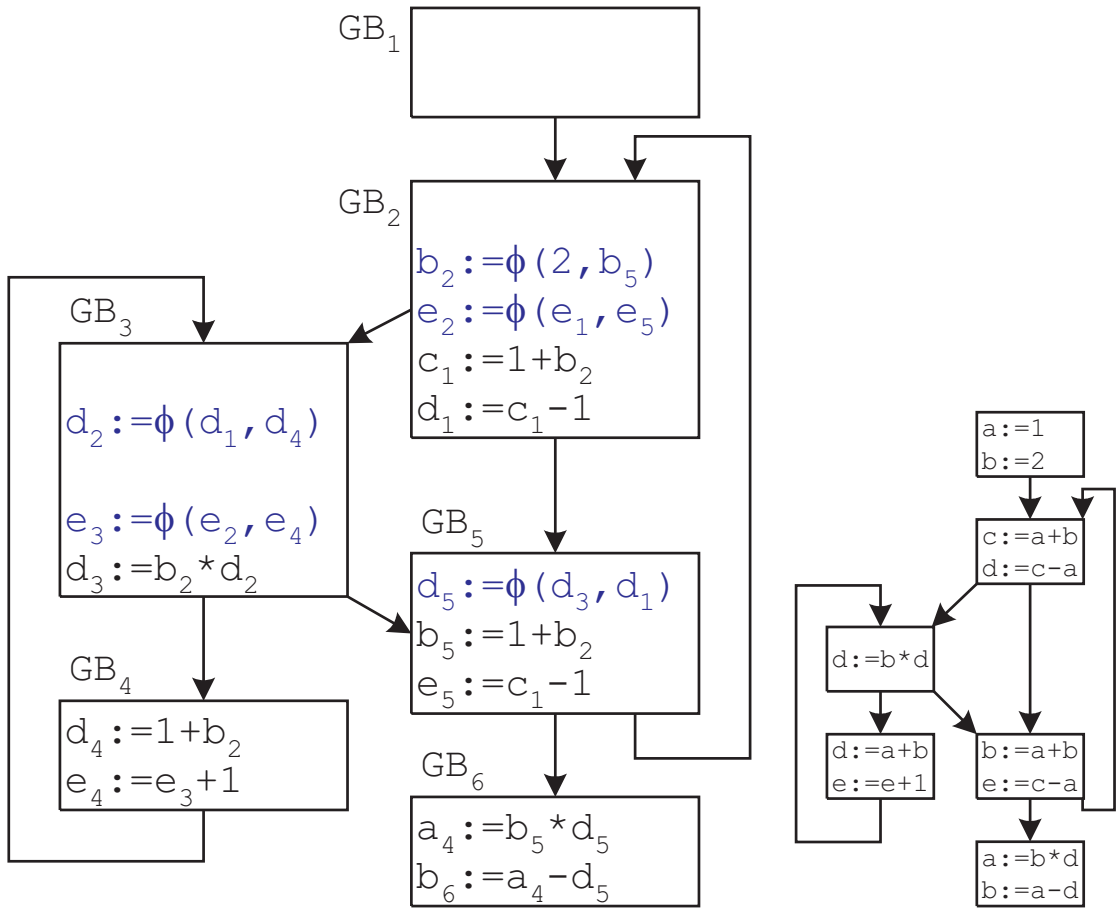
Vereinfachungen: Kopienfortpflanzung



Vereinfachungen: Konstantenfortpflanzung

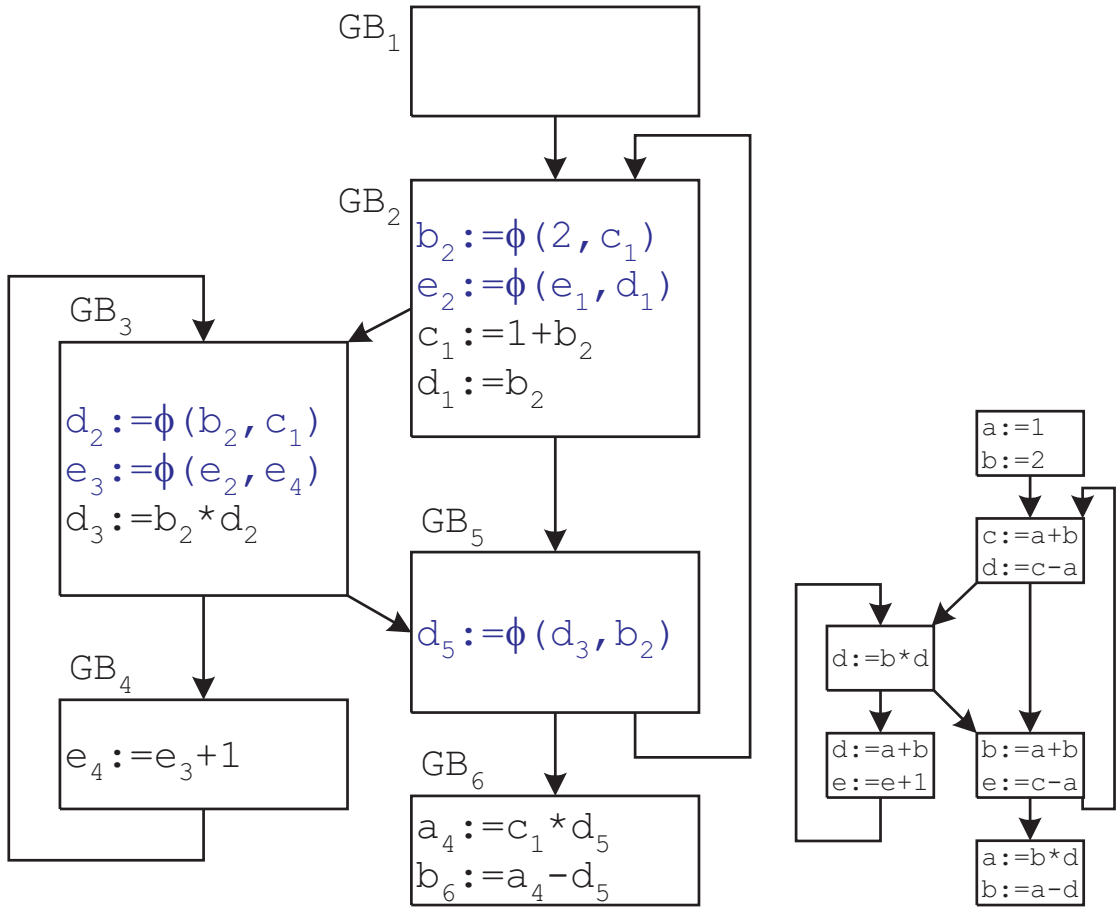


Vereinfachungen: Toten Code eliminieren



Weitere Vereinfachungen

- Gemeinsame Teilausdrücke
- Reassoziation
- konstante Ausdrücke auswerten
- Kopien fortschreiben
- Toten Code eliminieren



Firm-Algorithmus Pseudocode

```
proc writeVariable(block, varnum, node):
    block.variables[varnum] = node

internal proc setPhiArguments(phi):
    for pred in phi.block.preds:
        phi.args += readVariable(pred, phi.varnum)
    RemoveUnnecessaryPhi(phi)

proc readVariable(block, varnum):
    if varnum in block.variables:
        return block.variables[varnum]
    if block.matured:
        if startblock: uninitialized variable!
        if single pred: return readVariable(pred, varnum)
    phi = new Phi(block, varnum)
    writeVariable(block, varnum, phi);
    if block.matured: setPhiArguments(phi)
    return phi

proc matureBlock(block):
    for phi in block: setPhiArguments(phi)
    block.matured = true
```

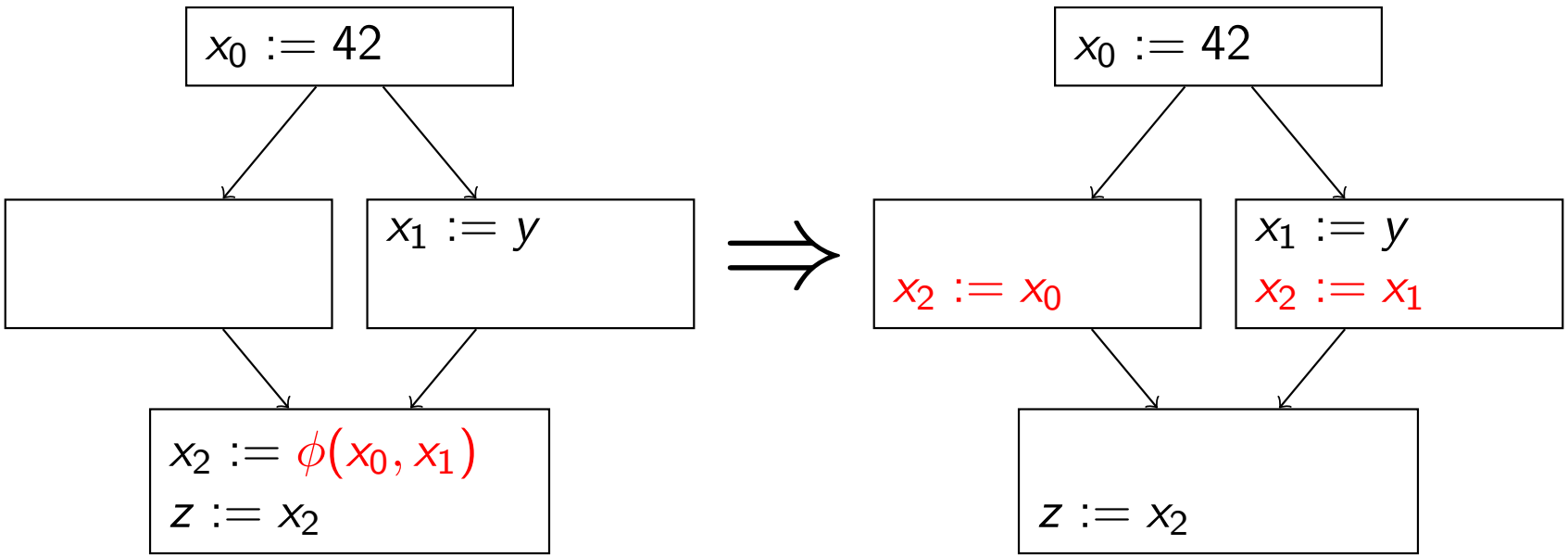
Eigenschaften

- Erzeugt niemals unbenutzte ϕ-Funktionen (pruned SSA-Form)
- Erzeugt bei reduziblem Kontrollfluss minimale Anzahl ϕ-Funktionen (minimal SSA-Form)
- Einfach
- Kommt ohne Dominanzbaum aus
- SSA-basierte Optimierungen schon während des Aufbaus anwendbar (Konstantenfaltung, Copy-Propagation, CSE, ...)

Kapitel 11: SSA-Form

Codegenerierung mit ϕ-Funktionen

Für $x_i = \phi(x_1, \ldots, x_n)$, erstelle in jedem Vorgängerblock$_k$ eine Kopieroperation $x_i = x_k$.



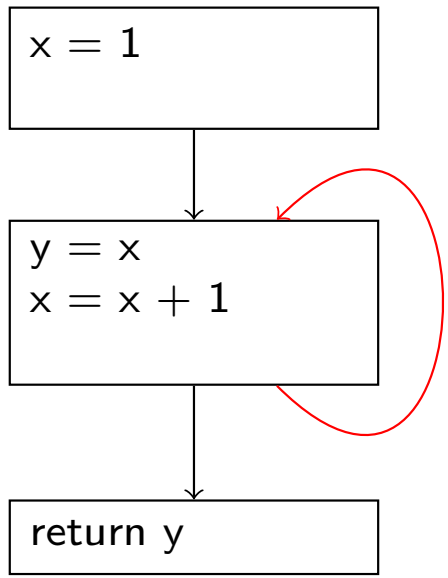
Angebracht: Anschließende Kopienminimierung

Alternative: Registerzuteilung, dann Registerpermutation zur Umsetzung der ϕ-Funktionen (hier nicht behandelt).

Stolperfalle: „Lost Copy“

Kritische Kanten können beim SSA-Abbau zu Problemen führen:

```
x = 1
do {
   y = x
   x = x + 1
} while (!p)
return y
```

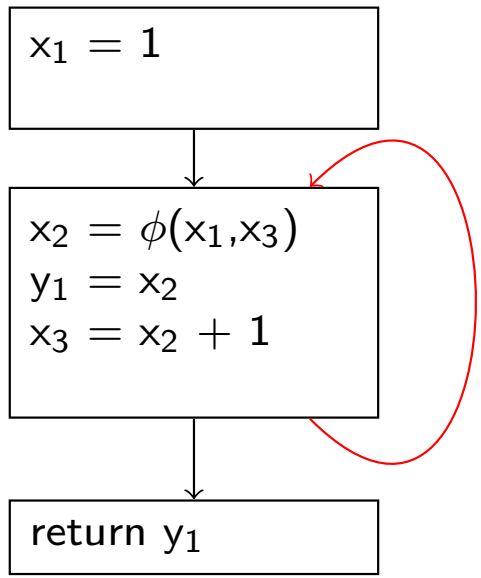


Kontrollflussgraph mit kritischer Kante

Stolperfalle: „Lost Copy“

Kritische Kanten können beim SSA-Abbau zu Problemen führen:

```
x = 1
do {
   y = x
   x = x + 1
} while (!p)
return y
```

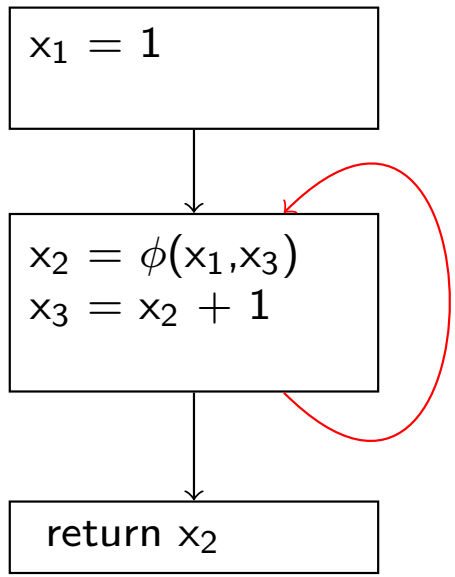


In SSA-Form

Stolperfalle: „Lost Copy“

Kritische Kanten können beim SSA-Abbau zu Problemen führen:

```
x = 1
do {
   y = x
   x = x + 1
} while (!p)
return y
```

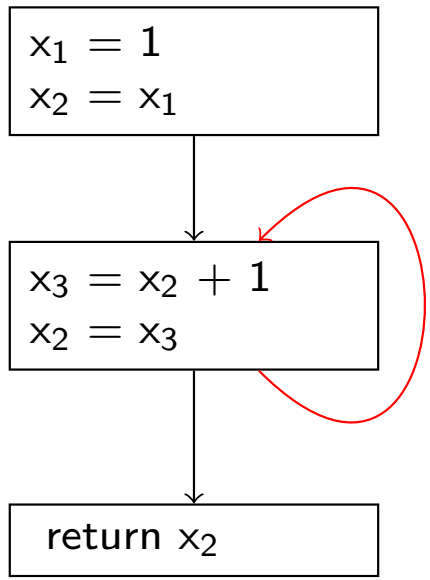


Kopienminimierung: Kein y mehr

Stolperfalle: „Lost Copy“

Kritische Kanten können beim SSA-Abbau zu Problemen führen:

```
x = 1
do {
   y = x
   x = x + 1
} while (!p)
return y
```

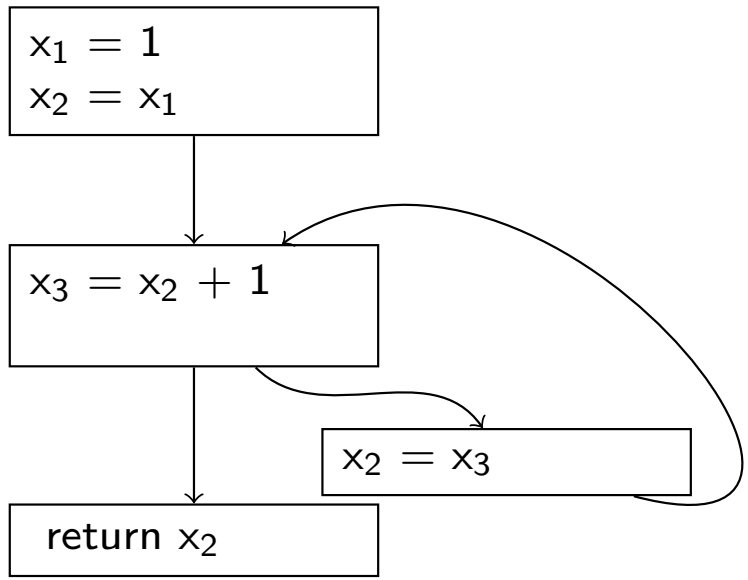


SSA abgebaut: Falscher Rückgabewert

Stolperfalle: „Lost Copy“

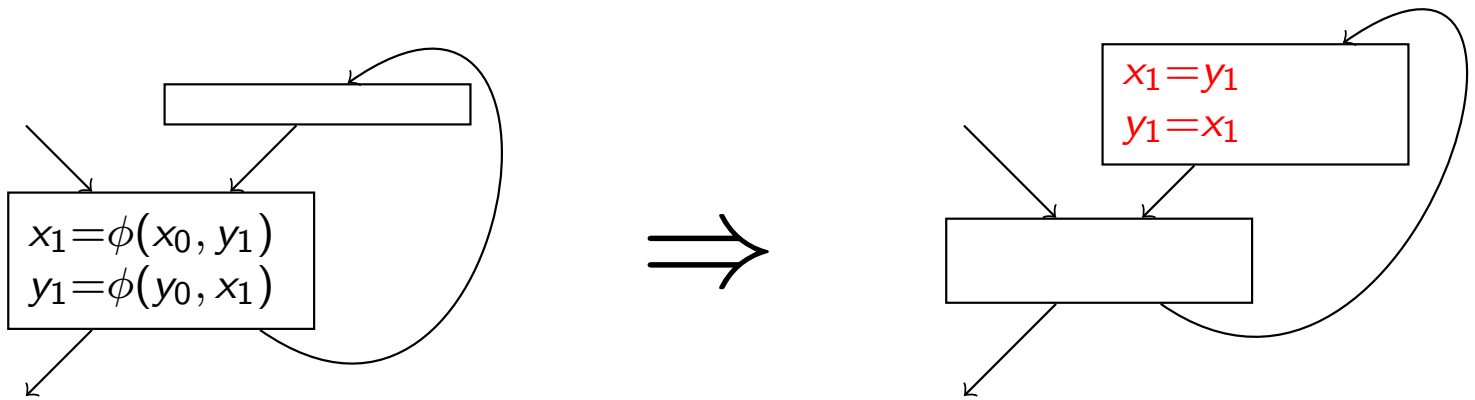
Kritische Kanten können beim SSA-Abbau zu Problemen führen:

```
x = 1
do {
   y = x
   x = x + 1
} while (!p)
return y
```

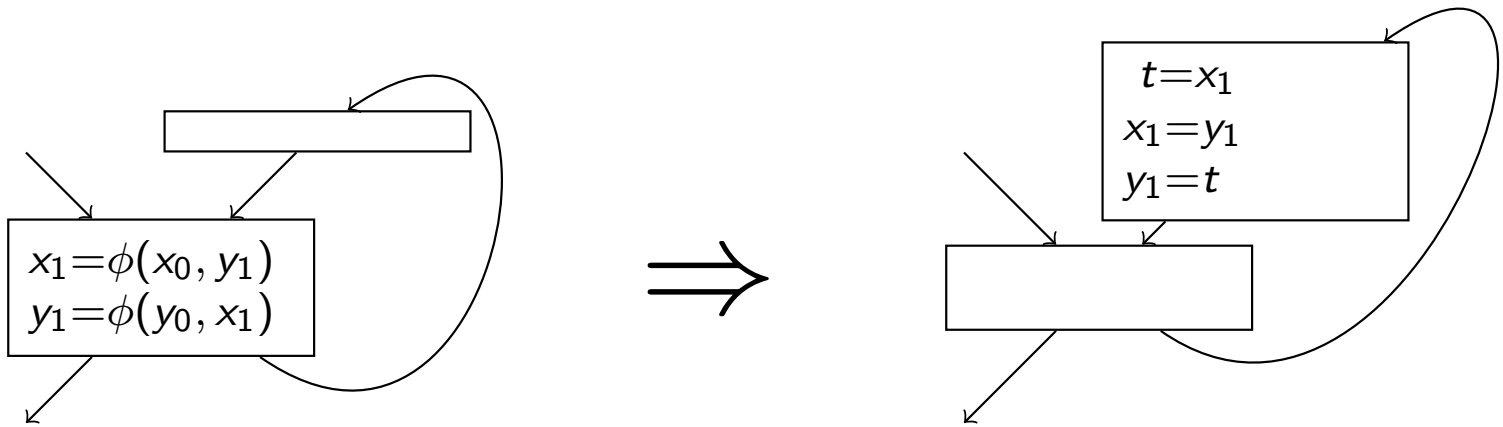


SSA korrekt abgebaut: Kritische Kante entfernt

Stolperfalle: Swap Problem



Stolperfalle: Swap Problem



Simultan-Auswertung manchmal nur durch zusätzliche Variable formulierbar

Kapitel 11: SSA-Form

Nielsen Optimierungen auf SSA

- **Common Subexpression Elimination**: Available Expression Analyse wird zur Identifikation isomorpher Expression-ASTs
- **Invariant Code Motion**: Identifiziere Expression-Sub-ASTs, die nicht von der Laufvariablen abhängen
- **Strength Reduction**: Finde Ausdrücke der Form $i * X$ wobei $X \in \text{ICM}$, und ersetze durch neue Laufvariable $j = j + X$
- **Copy Propagation**: Finde Zuweisungen $x_i = y_j$, wobei y_j SSA Wert ist
- **Dead Code Elimination**: Live Variable Analysis genau wie bei Nicht-SSA-Form

Nielsen Optimierungen in Firm

- **Common Subexpression Elimination**: Alle Knoten werden in Hashmap gespeichert. Bevor Knoten erstellt wird, erst nach äquivalentem Knoten in der Hashmap sehen.
- **(Invariant) Code Motion**: Platziere Knoten in einem Vorgängerblock, falls dieser auch von seinen Operanden dominiert wird. Nicht-invariante Knoten haben mindestens einen Operanden in der Schleife (ϕ!).
- **Strength Reduction**: Wie gehabt.
- **Copy Propagation**: Kopierzuweisungen existieren in Firm per Konstruktion nicht. (siehe `readVariable`)
- **Dead Code Elimination**: Firmgraphen sind Abhängigkeitsgraphen. Toter Code hat keine Abhängigkeiten. Ergo gibt es per Definition keinen toten Code in Firm.