

# SSA Form



## 1 Einführung – Motivation

- FIRM

## 2 SSA Aufbau

- Theorie
- Beispiel
- Konstruktion aus dem AST



# Statische Einmalzuweisung – SSA

Ziel:

- Effizienz prozedurglobaler Optimierungen steigern
- Datenflußanalysen beschleunigen
- Definiert-Benutzt-Beziehungen explizit darstellen

Definition SSA (Static Single Assignment, statische Einmalzuweisung): Ein Programm ist in SSA Form, wenn an jede Variable jeweils an genau einer Programmstelle zugewiesen wird.

- Programm bedeutet hier zunächst Prozedur
- verlangt: Ablauf des Programms ist reduzibel
- Variable bedeutet hier zunächst aliasfreie, lokale Variable
- beachte: SSA bedeutet nicht, daß jeder Wert nur einmal berechnet wird.



# SSA intuitiv: Erkenntnisse

- Maschinenbefehle verarbeiten Werte im Speicher/Registern. **Es kommt nicht darauf an, wie die zugehörigen Variablen im Quellprogramm heißen.** (Viele Werte entstammen der Adreßrechnung und sind namenlos.)
- Wenn zweimal dieselbe Operation auf die gleichen Operanden angewandt wird, kann man eine Operation weglassen (Idee der Wertnumerierung)
  - gleicher Operand heißt hier: gleicher Wert zur Laufzeit
- **Ob zwei Operanden gleichen Wert besitzen, kann von der Vorgeschichte, dem Ablaufpfad, der zu der Berechnung führt, abhängen.**



# SSA intuitiv: schematische Konstruktion

Konstruktionsidee:

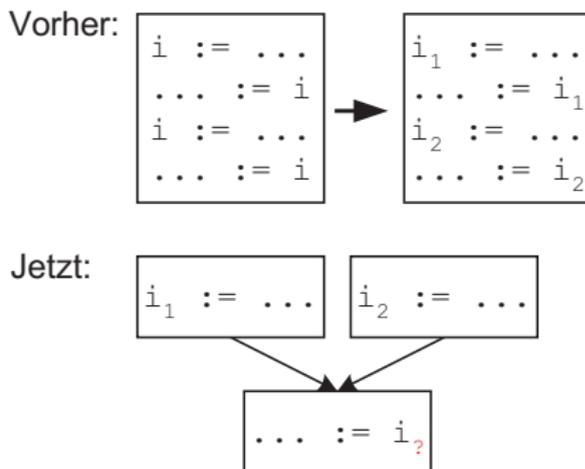
- Ersetze alle Zuweisungen durch Vereinbarungen (Definitionen) dynamischer Konstanter, an die kein zweites Mal zugewiesen werden kann (daher SSA: statische Einmalzuweisung)
- Verschiedene Zuweisungen an eine Programmvariable  $a$  führen zu Vereinbarungen verschiedener Konstanter  $a_1, a_2, \dots$
- Überall, wo  $a$  als Operand benutzt wird, setze die dort gültige Definition  $a_i$  ein
- **Problem:** Was tun, wenn die gültige Definition vom Ablaufpfad abhängt, auf dem die Benutzung als Operand erreicht wird?
- **Lösung:** Setze zuvor eine Auswahlfunktion ( $\phi$ -Funktion) ein, die die gültige Definition abhängig vom Ablaufpfad selektiert.
- Nebenbei: Diese Idee löst das Datenflußproblem „ordne der Verwendung einer Variablen ihre letzte Definition zu“!



# Abstrakte Werte

Problem: Wie verfährt man mit verschiedenen abstrakten Werten bei Ablauf-Vereinigung?

Welches  $i$  ist hier gültig?

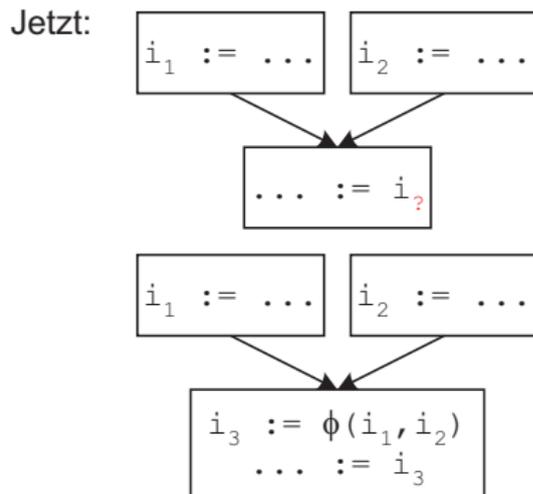
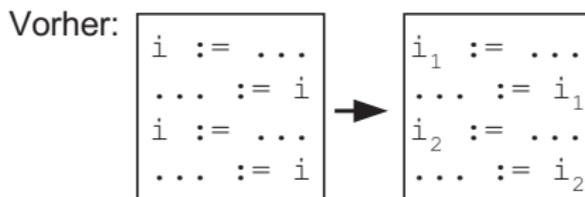


# Abstrakte Werte II

Problem: Wie verfährt man mit verschiedenen abstrakten Werten bei Ablauf-Vereinigung?

Welches  $i$  ist hier gültig?

Bei Zusammenführungen des Ablaufs den Wert durch eine Pseudooperation  $i_3 := \phi(i_1, i_2)$ , eine  $\phi$ -Funktion, auswählen.

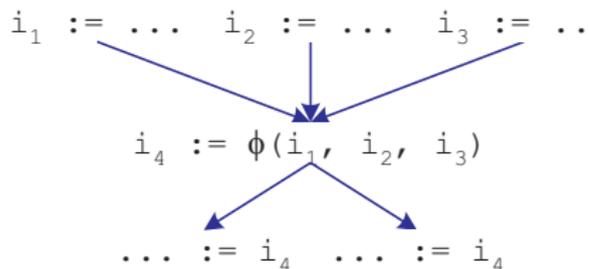
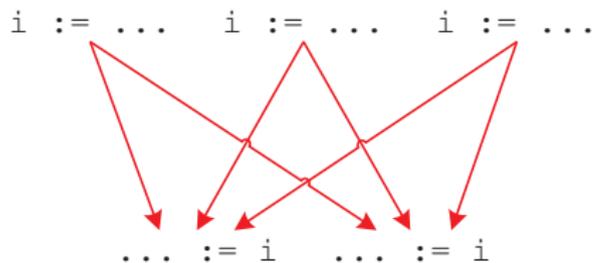


# Explizite Definiert-Benutzt-Beziehungen

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

**vorher:** #Defs  $\times$  #Bens

**jetzt:** #Defs + #Bens

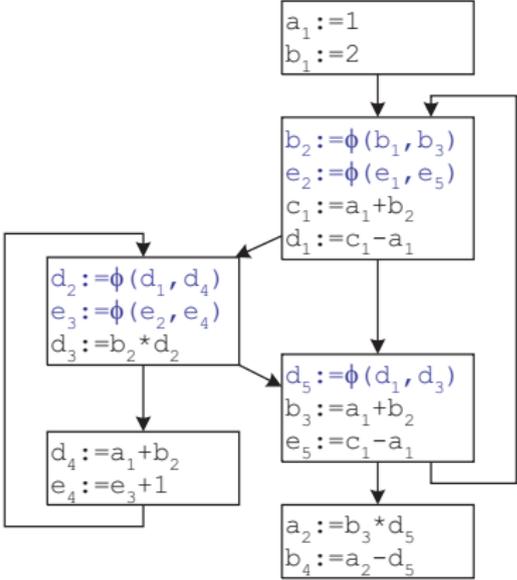
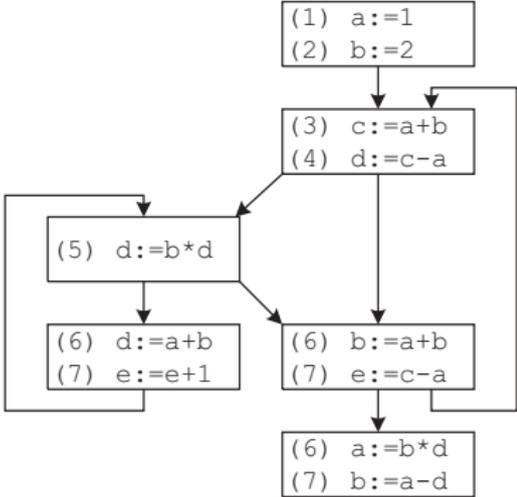


# $\phi$ -Funktionen

- Eine  $\phi$ -Funktion  $i_3 := \phi(i_1, i_2)$  wählt abhängig vom Programmablauf einen der Werte  $i_1, i_2$  aus und benutzt ihn als Wert  $i_3$ .
- Eine  $\phi$ -Funktion hat genau so viele Operanden, wie der zugehörige Grundblock Vorgänger im Ablaufgraph.
- Das  $k$ -te Argument einer  $\phi$ -Funktion ist eineindeutig dem  $k$ -ten Vorgänger zugeordnet.
- Das Ergebnis einer  $\phi$ -Funktion ist das Argument, das dem Pfad, auf dem die  $\phi$ -Funktion erreicht wurde, zugeordnet ist.
- $\phi$ -Funktionen stehen immer am Blockanfang.
- Alle  $\phi$ -Funktionen eines Blocks werden **simultan** ausgewertet.



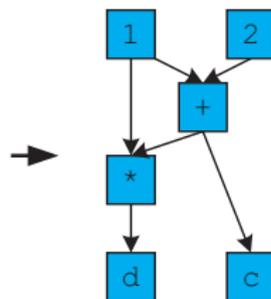
# Beispiel: Grundblock- und SSA-Grundblockgraph



# Implementierung

- Optimierungen benötigen Definiert-Benutzt-Beziehungen.
- Dafür Operationen in Tripelform als Ecken eines Graphen reinterpretieren:
- Graph-Interpretation:
  - Jeder abstrakte Wert (Wertnummer) ist eine Ecke.
  - Jede Ecke enthält Operation oder Konstante.
  - Def.-Ben. Beziehungen sind Kanten (**Datenflußkanten**).
  - Umkehrung der Pfeile entspricht Datenabhängigkeiten.
  - **Ablaufkante**: Welchen Grundblock erreicht ein Sprung?  
Zu welchem Grundblock gehört eine Operation?
  - Weitere Kantenarten später

```
a := 1  
b := 2  
c := a + b  
d := c * a  
...
```



# Implementierung: Firm

- Firm ist eine moderne SSA-Zwischendarstellung in Graphform; entstanden aus Diplomarbeiten, Dissertationen, Industriekooperationen des Lehrstuhls 1995-2006.
- Ablauf, Datenfluß und Sequentialisierung von Operationen werden in einem Graphen dargestellt.
- Firm ist ein Abhängigkeitsgraph, die Kantenrichtung ist also umgekehrt zur Datenflußrichtung.
- Zur Unterscheidung der Konzepte sind die Kanten des Graphen typisiert, die Typen heißen Modi.
- Grundblöcke sind auch Ecken, jede andere Ecke hat eine Grundblockecke als Vorgänger (ermöglicht einheitlichen ADT)
- Firm ist durch Angabe der Operationen (Ecken), Kantenmodi und der Signatur der Operationen definiert.



## 1 Einführung – Motivation

- FIRM

## 2 SSA Aufbau

- Theorie
- Beispiel
- Konstruktion aus dem AST



## 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*:

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

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

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

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



## Dominanzgrenze und iterierte DG

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

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

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

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

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

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



## Plazierung der $\phi$ -Funktionen

Wo müssen die  $\phi$ -Funktionen optimal plaziert werden?

- SSA-Eigenschaft muß erfüllt sein (nur eine Definition)
- Programm muß korrekt dargestellt sein
- Minimale Anzahl von  $\phi$ -Funktionen

Satz *Plazierung  $\phi$ -Funktionen*:

Sei  $P$  eine Prozedur in SSA Form mit minimaler Anzahl  $\phi$ -Funktionen. Seien  $X, Y$  Grundblöcke in  $P$  mit einer Definition von  $v$  und Ablaufpfaden  $X \rightarrow^+ Z, Y \rightarrow^+ Z$ , wobei  $Z$  der erste gemeinsame GB dieser Pfade ist.

Dann enthält  $Z$  eine  $\phi$ -Funktion für  $v$ , falls noch ein Gebrauch von  $v$  folgt.

**Beweisidee**

- $\phi$ -Funktion kann nicht früher eingesetzt werden.
- $\phi$ -Funktion darf nicht in einen späteren Block  $Z'$  eingesetzt werden:

Die Wege  $Z \rightarrow^+ Z'$  enthalten keine Möglichkeit, die ursprünglichen Definitionen von  $v$  zu differenzieren.



# Folgerungen

- $X$  bzw.  $Y$  müssen alle direkten Vorgänger von  $Z$  dominieren, sonst gäbe es einen Gebrauch von  $v$  ohne vorherige Definition. Daher gilt  $Z \in DG(X, Y)$ .  
Nebenbei: beim Einsetzen von  $\phi$ -Funktionen werden alle nicht-initialisierten, aber benutzten einfachen Variablen entdeckt!
- Da die  $\phi$ -Funktion in  $Z$  eine neue Definition von  $v$  ist, werden  $\phi$ -Funktionen in den iterierten Dominanzgrenzen der ursprünglichen Definitionen eingesetzt. Hier werden weitere Definitionen von  $v$ , die erst in einem Block  $\notin DG(X, Y)$  hinzugenommen werden, vereinigt.

## Achtung:

Dominanzgrenzen sind als Konstruktionsvorschrift weniger geeignet!



# Wie konstruiert man SSA-Form

- mehrere Konstruktionsverfahren möglich
- im schlimmsten Fall enthalten alle Grundblöcke  $\phi$ -Funktionen für alle Variable:
  - Aufwand  $O(n^2)$ ,  $n =$  Anzahl Variable, nicht vermeidbar
  - praktisch ist der Aufwand linear
- Grundidee unseres Verfahrens (Click 1995):
  - während eines Durchlaufs des Strukturbaums (AST) führe erweiterte Wertnumerierung durch:
    - bei nur einem Vorgängerblock Wertnummern übernehmen
    - bei mehreren Vorgängern vorläufige  $\phi'$ -Funktionen  $\phi'(\dots)$  einsetzen
    - Argumentliste der  $\phi'$ -Funktionen erweitern, wenn weitere Vorgänger gefunden werden
    - am Ende  $\phi'$ -Funktionen in  $\phi$ -Funktionen umwandeln oder streichen (wenn Wert nicht mehr benötigt)



# SSA Aufbau mit Wertnumerierung

- Ausgangspunkt:
  - AST oder Grundblockgraph mit Zuweisungen der Form  $x := \tau(y, z)$ ;  $x, y, z$  lokale Variable (also aliasfrei)
  - Anzahl in Prozedur verwendeter lokaler Variablen bekannt ( $n$ )
- Vorgehen:
  - Merke in jedem Grundblock aktuellen Wert jeder Variablen, d.h. den definierenden Ausdruck (Reihung der Größe  $n$ )
  - Bei Verwendung einer Variablen benutze Wertnummer dieses Ausdrucks
  - Funktionen „hole Wertnummer“  $HW(v)$ , „merke Wertnummer“  $MW(v, wn)$
  - $HW(v)$  fügt  $\phi$ -Funktion ein, wenn Variable in Vorgängerblock definiert  
**praktisch:**  $\phi$ -Funktionen werden nur generiert, wenn Wert noch lebendig!
  - Berechnen einer Wertnummer für Ausdrücke  $t$ ,  $t = \tau(y, z)$  mit  $WN(t)$  wie gehabt



# SSA Aufbau mit Wertnumerierung

Für jede Zuweisung  $x := \tau(y, z)$ :

- hole Wert für  $y, z \mapsto HW(y), HW(z)$
- berechne Wertnummer  $WN(\tau, y, z)$  für  $\tau(y, z)$
- Falls Wertnummer neu, füge Zuweisung  $WN(\tau, y, z) := \tau(HW(y), HW(z))$  in Grundblock ein.
- Merke Wert für  $x$ :  $MW(x, WN(\tau, y, z))$

**Bemerkung:** Aufruf von  $WN$  eliminiert gemeinsame Teilausdrücke!



# SSA Aufbau mit Wertnumerierung

Vorgehen von  $HW(v)$ :

- Wenn in aktuellem Grundblock Wert  $w$  für Variable  $v$  bereits gemerkt, verwende diesen
- Wenn genau ein Vorgängerblock rufe  $HW(v)$  dort auf
- Wenn zwei (mehrere) Vorgängerblöcke:
  - rufe  $HW(v)$  in jedem dieser Blöcke auf  $\Rightarrow$  liefert Werte  $w_1, w_2$
  - füge Zuweisung  $WN(\phi, v, v) := \phi(w_1, w_2, \dots)$  in aktuellem Grundblock ein.
  - merke neuen Wert für  $v$  :  $MW(v, WN(\phi, v, v))$
  - gebe neuen Wert als Ergebnis zurück



# SSA Aufbau und unbekannte Vorgänger

**Beobachtung:** Beim Aufbau von Schleifen sind die Wertnummern aus den Vorgängerblöcken u.U. nicht bekannt,  $HW(v)$  also (noch) nicht definiert

**Abhilfe:** Zweistufiges Vorgehen:

- Markiere Blöcke in SSA-Form
- Wenn alle Vorgänger in SSA-Form, berechne  $\phi$ -Funktion wie gehabt.
- Wenn Vorgänger nicht in SSA-Form, füge  $\phi'$  ein und merke Operand des  $\phi'$  zum späteren Fertigstellen (passiert nur an Dominanzgrenzen)
- Wenn Grundblock in SSA-Form gebracht, teste, ob schon vorhandene Nachfolger fertig gestellt werden können, und stelle sie fertig.



# SSA Aufbau und unbekannte Vorgänger

## Beobachtungen:

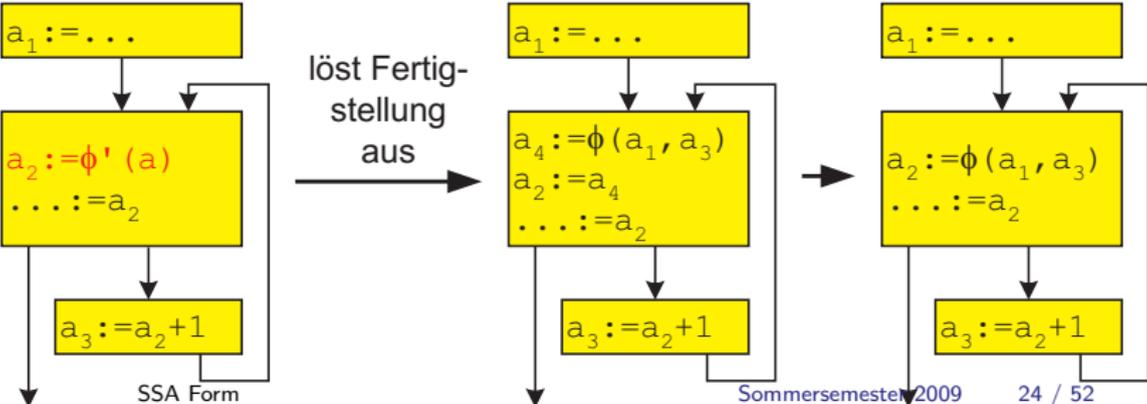
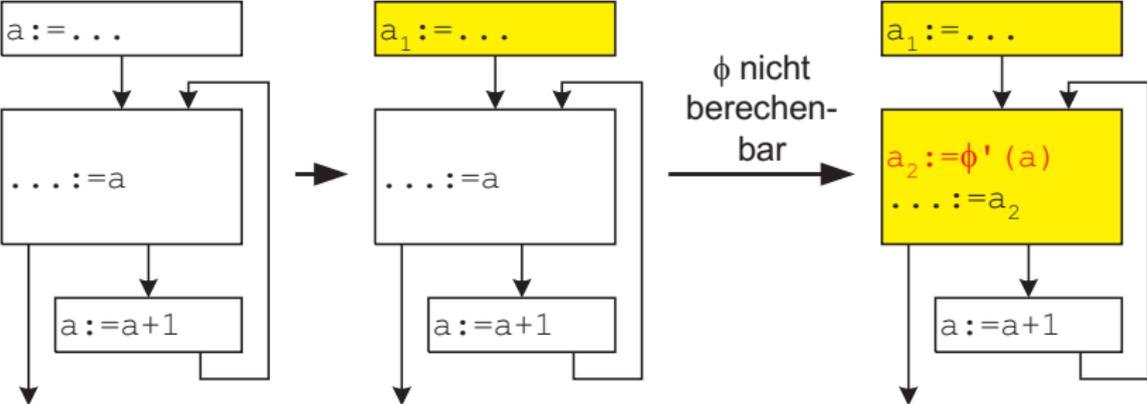
- Geht man beim Aufbau soweit möglich in Ablafrichtung vor, sind eindeutige Vorgänger immer vor ihren Nachfolgern in SSA-Form: Dominatoren werden immer zuerst aufgebaut.
- Bei Aufbau aus dem AST ist bekannt, wann alle Vorgänger aufgebaut sind (außer bei expliziten Sprüngen mit *goto*).

## Folgerung:

- Aufbau bei reduzierbarem Ablauf effizient!
- Wenn Dominatoren bekannt, ist globale Eliminierung gemeinsamer Teilausdrücke effizient.



# Unbekannte Vorgänger: Beispiel



# SSA Aufbau - Abbruchkriterien für HW

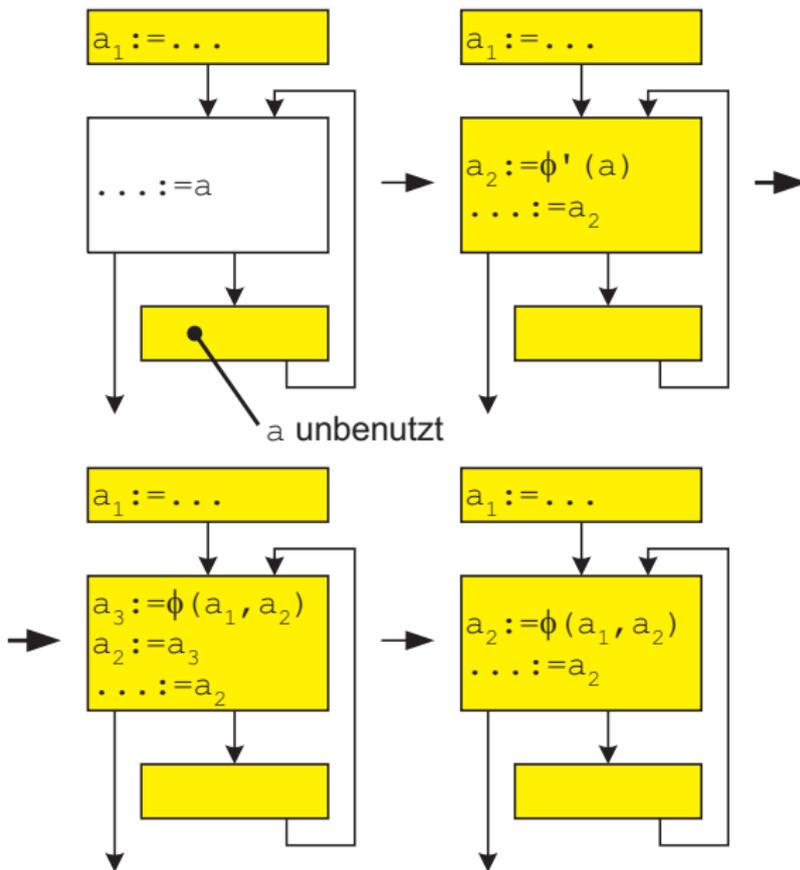
Beobachtung:

Algorithmus *HW* kann rekursiv über alle Vorgängerblöcke iterieren.

- Erreicht er den Startblock, wird ein undefinierter Wert verwendet (Fehlermeldung: nicht initialisierte Variable)
- Bei zyklischem Ablauf ohne Definitionen wird Endlosrekursion durch  $\phi'$  unterbunden  
(dies ist die zweite wichtige Aufgabe der  $\phi'$ -Funktionen!)



# Abbruchkriterien für *HW*: Beispiel



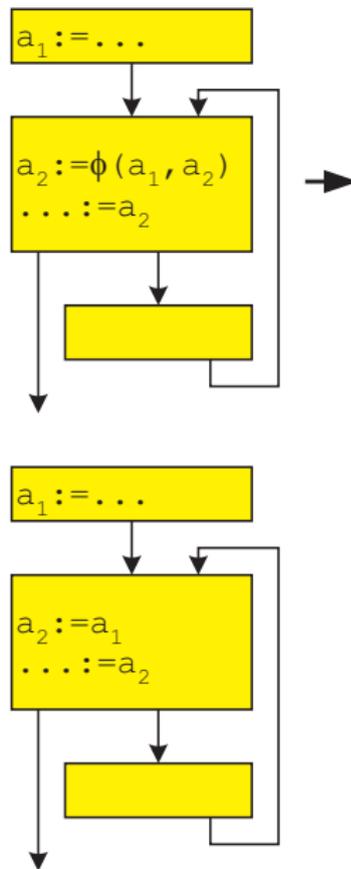
# SSA Aufbau und unnötige $\phi$ -Funktionen

Beobachtung:

Es entstehen unnötige  $\phi$ -Funktionen:

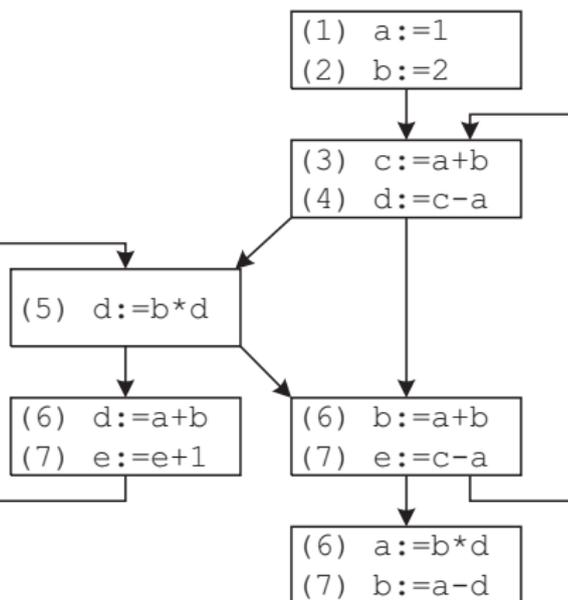
- Die Operanden sind das Ergebnis der  $\phi$ -Funktion
- Da jede Schleife mindestens einen Vorgänger außerhalb der Schleife hat, existiert mindestens ein sinnvoller Operand
- Gibt es nur einen Operanden, so ersetze die  $\phi$ -Funktion durch diesen.

(Bei nicht reduzierbarem Ablauf gibt es Komplikationen.)

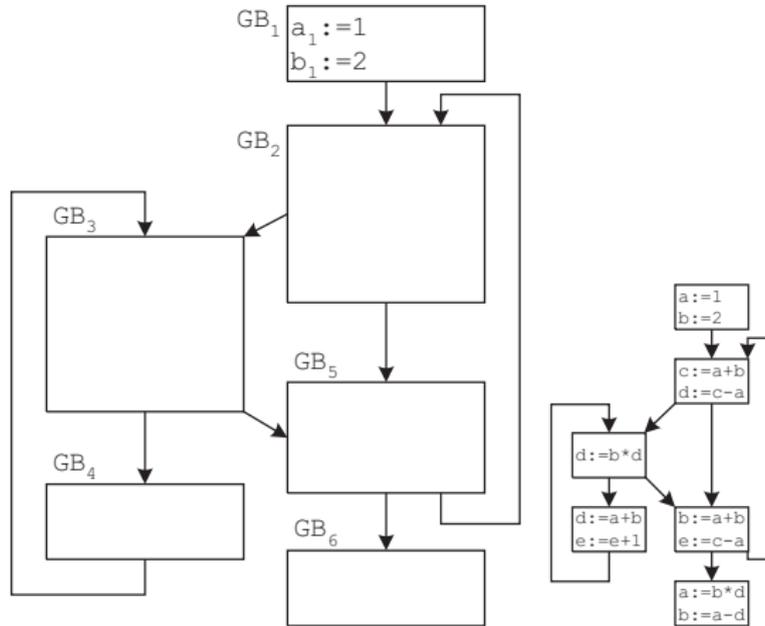


# Beispielprogramm und Grundblockgraph

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

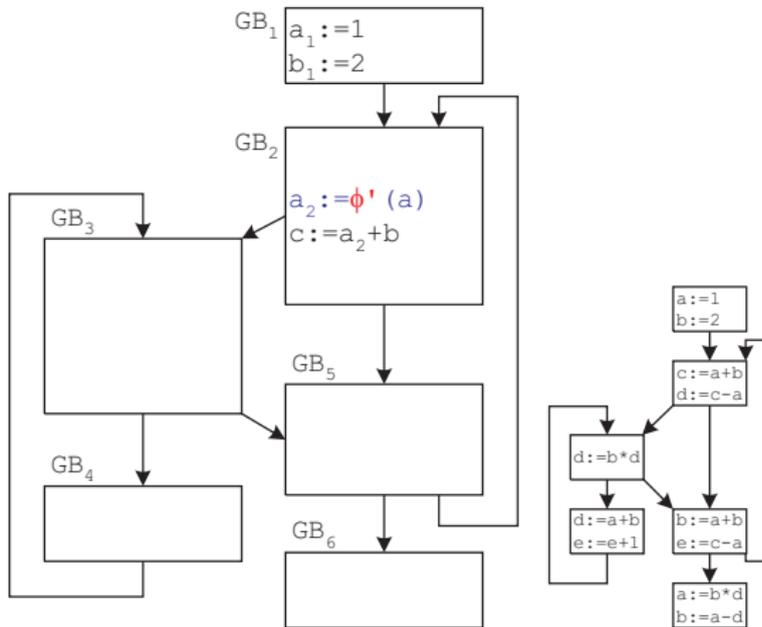


# SSA Aufbau Block 1



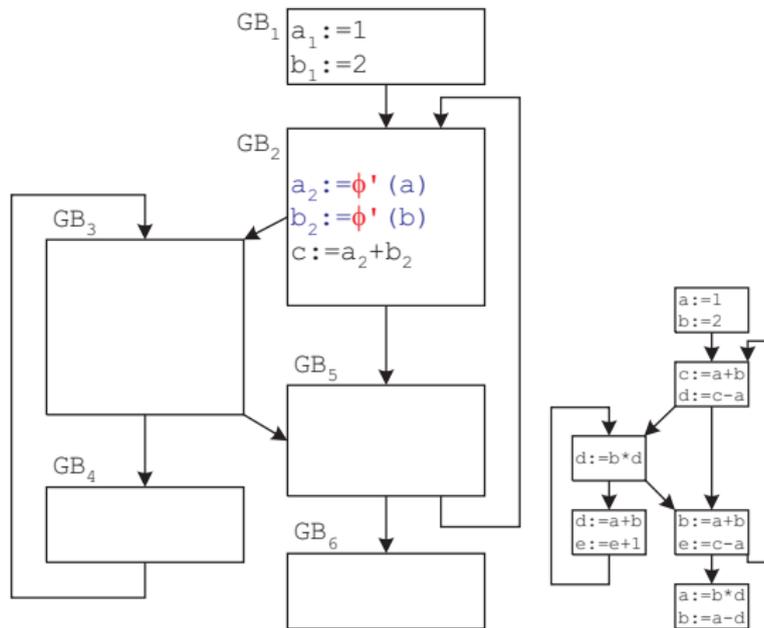
# SSA Aufbau Block 2

Holen der  $wn$  für  $a$  erzeugt zuerst  $\phi'$  für  $a$  ...



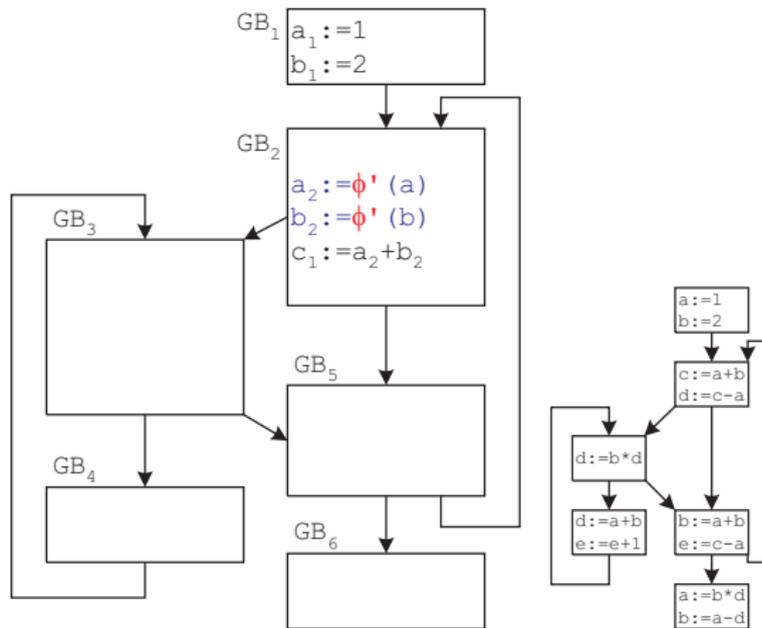
# SSA Aufbau Block 2

... dann für  $b$  ...



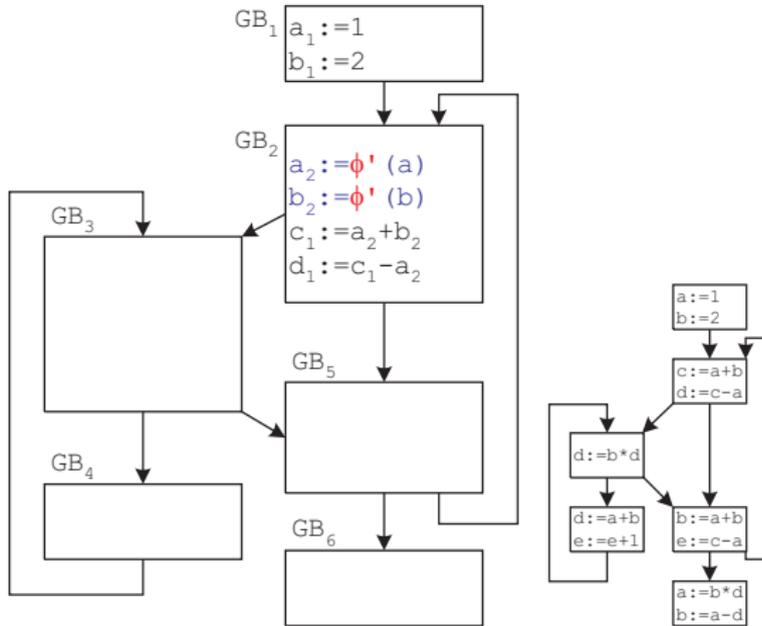
# SSA Aufbau Block 2

... und schließlich eine *wn* für *c*.

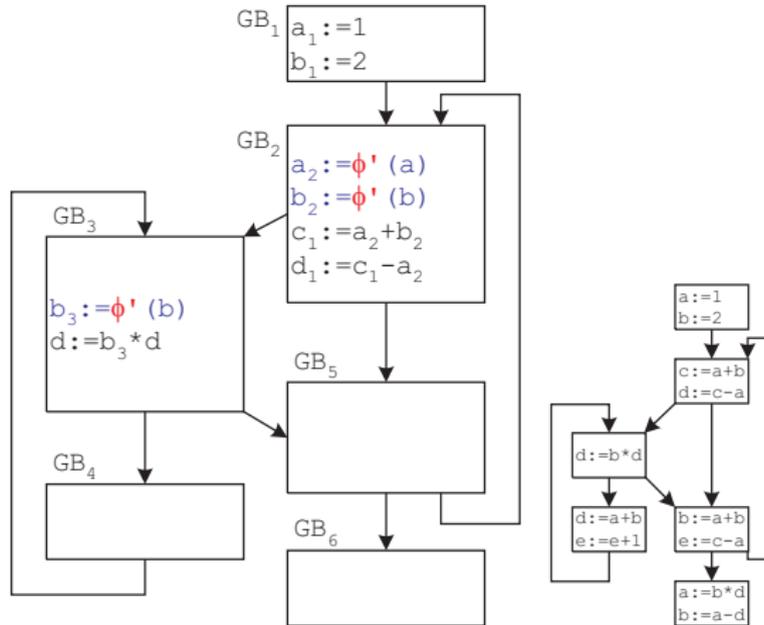


# SSA Aufbau Block 2

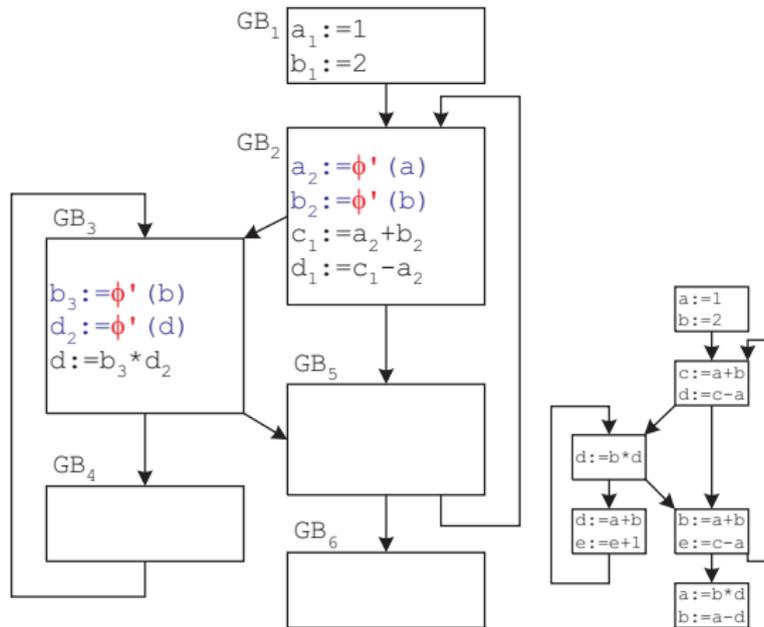
Der Aufbau für  $d := c - a$  funktioniert wie normale Wertnumerierung.



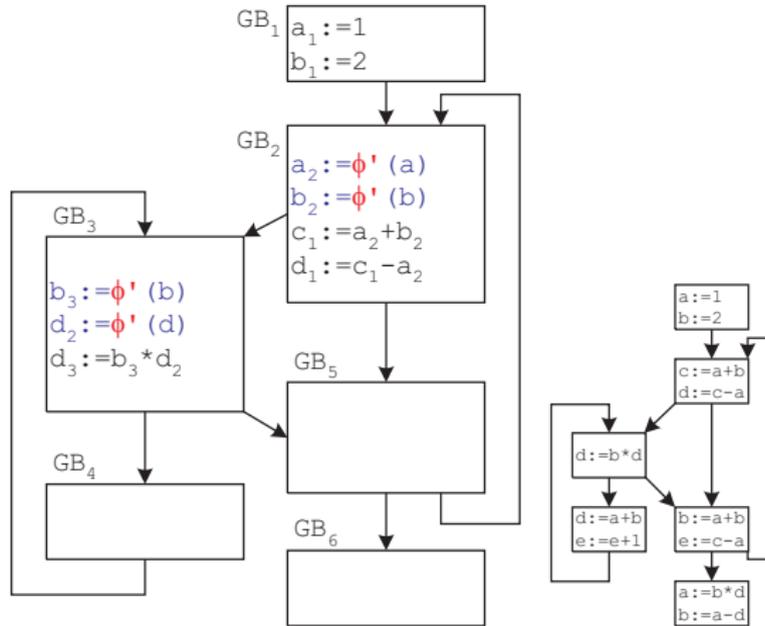
# SSA Aufbau Block 3



# SSA Aufbau Block 3

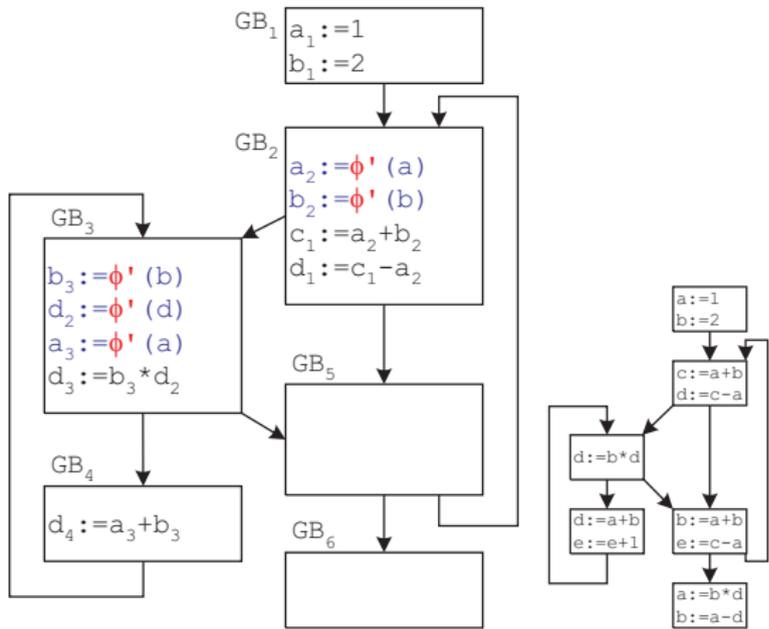


# SSA Aufbau Block 3

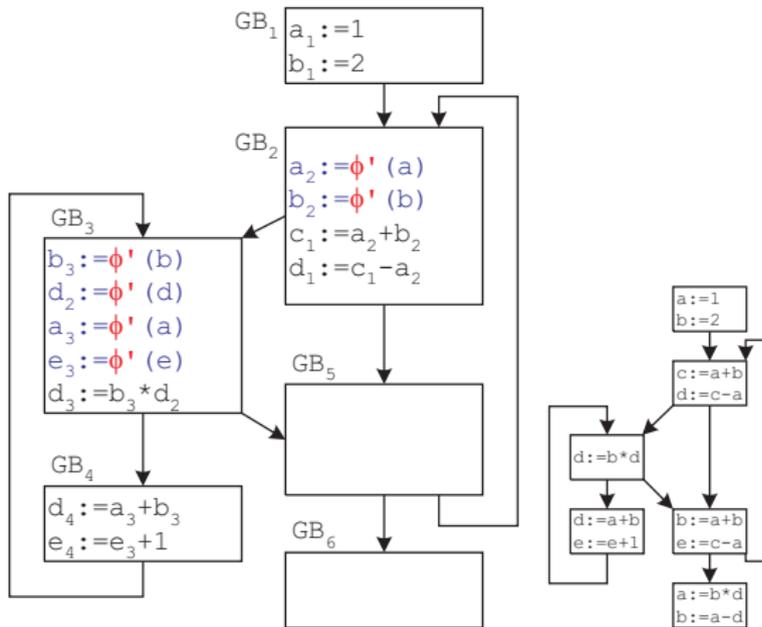


# SSA Aufbau Block 4

Der Aufruf  $HW(a)$  in 4 führt zu rekursivem Aufruf  $HW(a)$  in 3. Dieser erzeugt in 3 neuen  $\phi'$ -Funktionen für  $a$ .



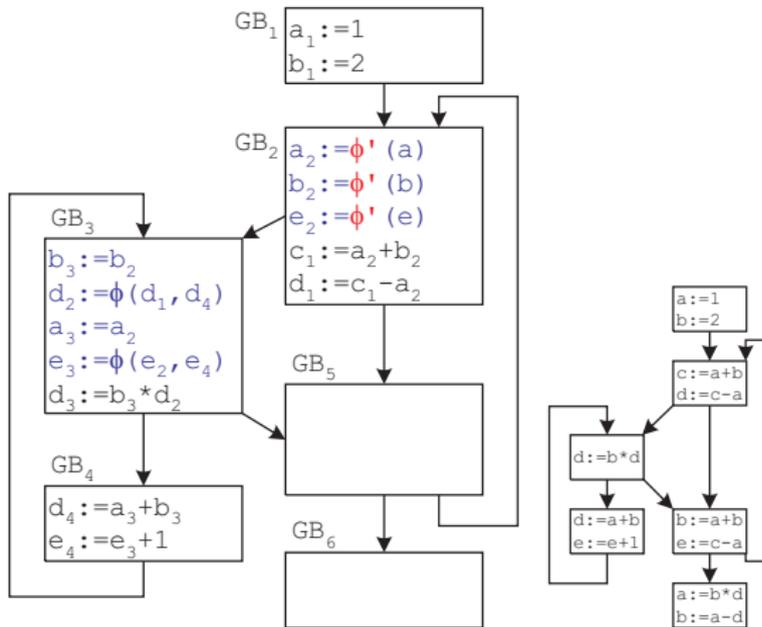
# SSA Aufbau Block 4



# SSA Aufbau Block 4

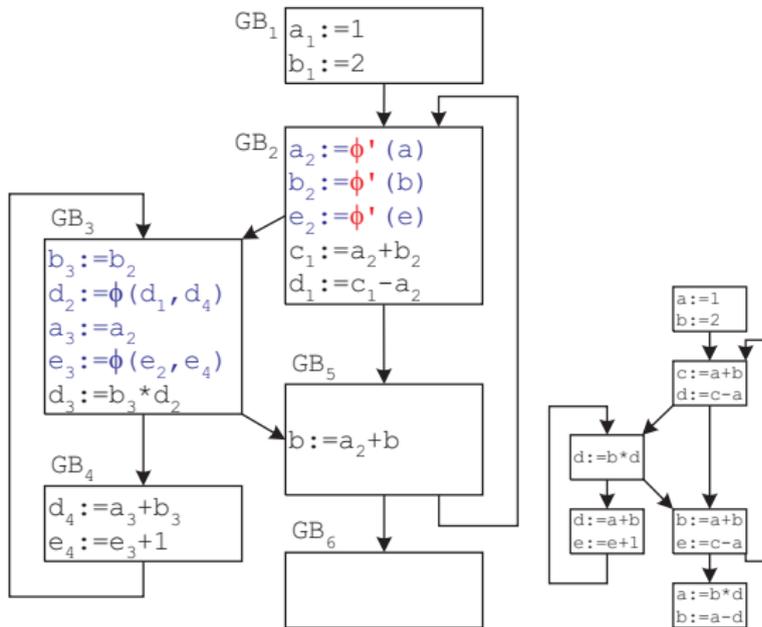
Jetzt alle Vorgänger von Block 3 in SSA Form:  $\phi$ -Funktionen werden berechnet.

Für  $e$  wird rekursiv ein  $\phi'$ -Funktionen in Block 2 eingesetzt.

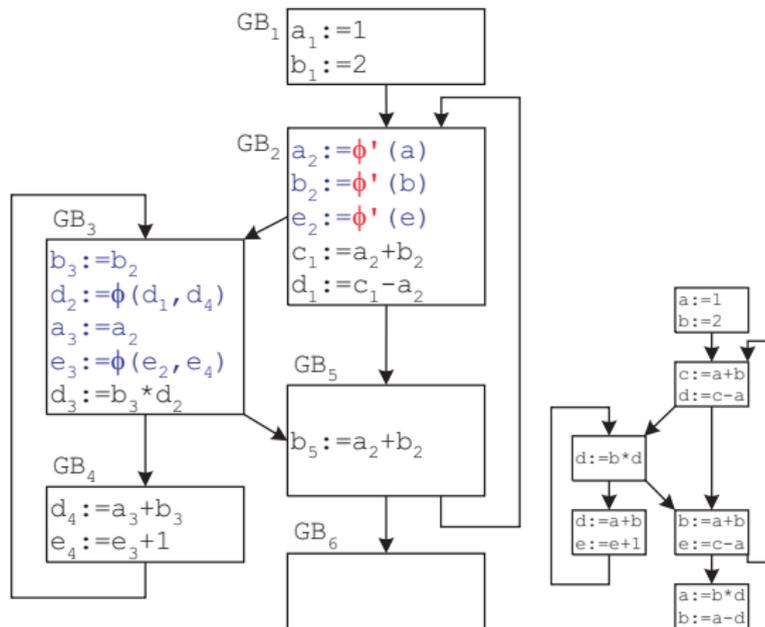


# SSA Aufbau Block 5

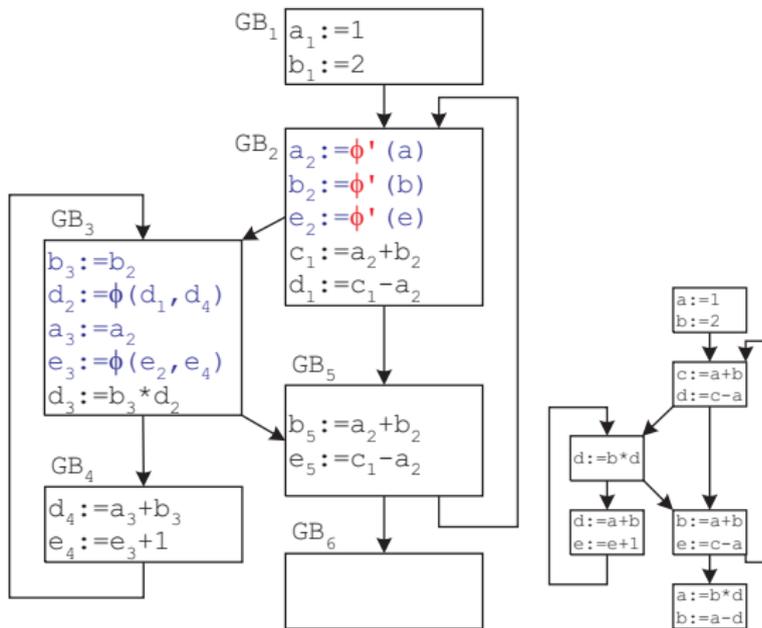
$HW(a)$  in 5 überspringt Kopien, findet eindeutige Definition:  
keine  $\phi$ -Funktion nötig.



# SSA Aufbau Block 5



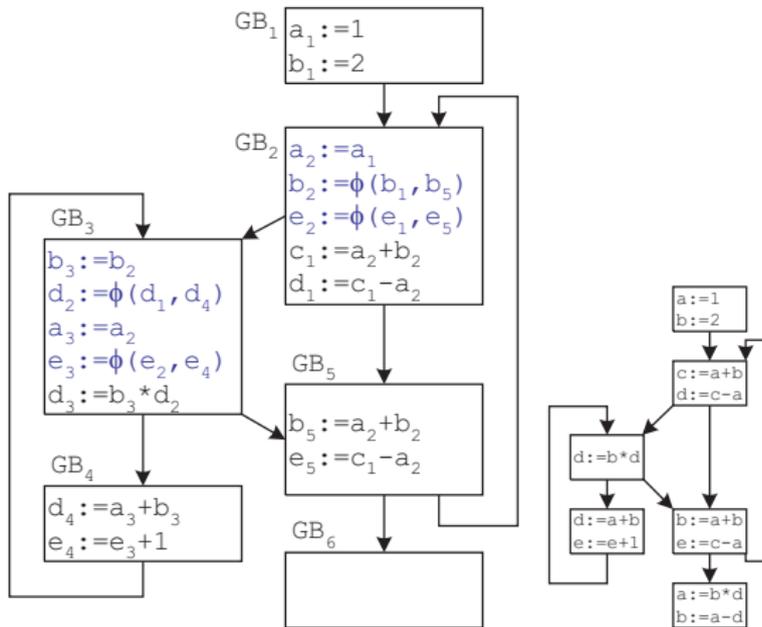
# SSA Aufbau Block 5



# SSA Aufbau Block 5

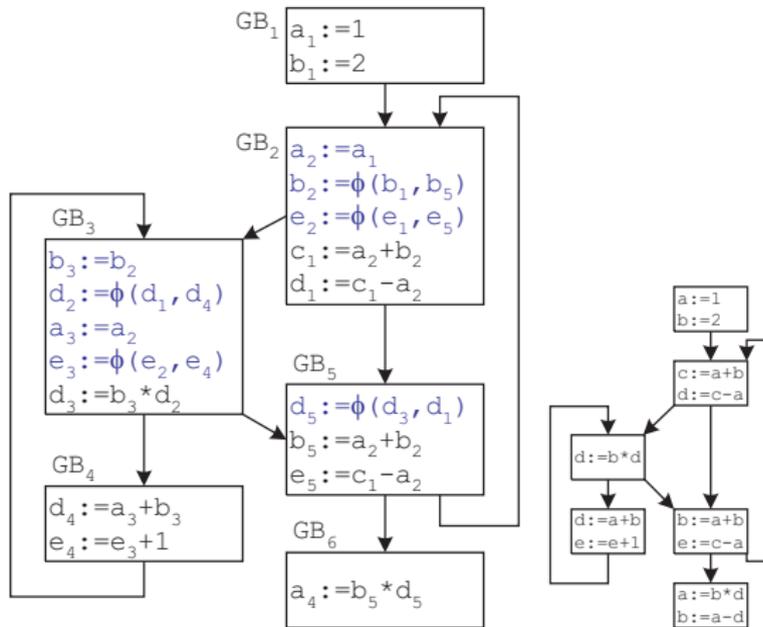
Jetzt alle Vorgänger von Block 2 in SSA Form:  $\phi$ -Funktionen werden berechnet.

Algorithmus bemerkt:  $e$  ist uninitialisiert! Annahme: Wert  $e_1$

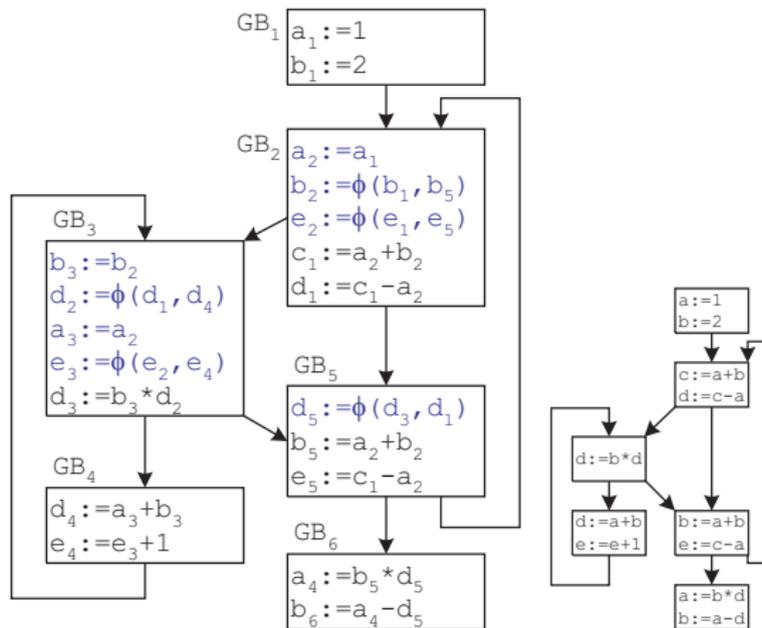


# SSA Aufbau Block 6

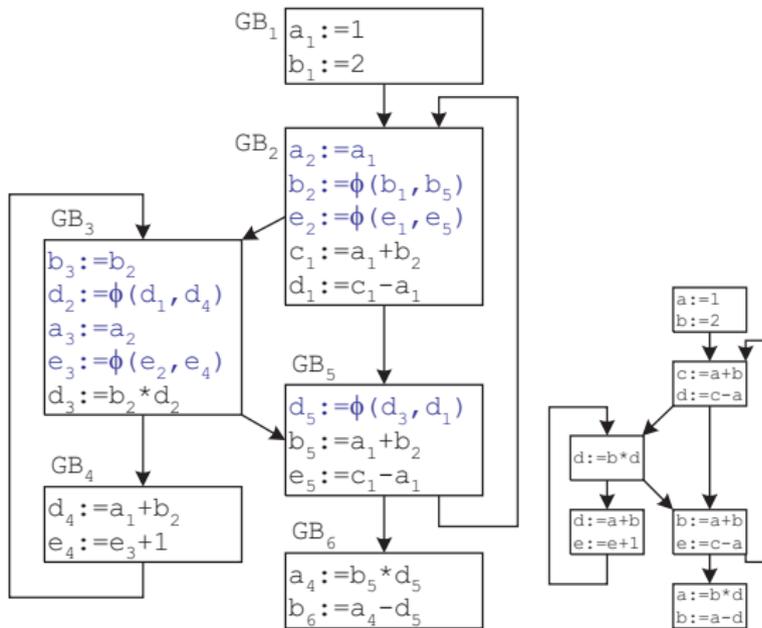
Rekursiver Aufruf von  $HW(a)$  in 5 setzt komplette  $\phi$ -Funktion  $d_5$  ein



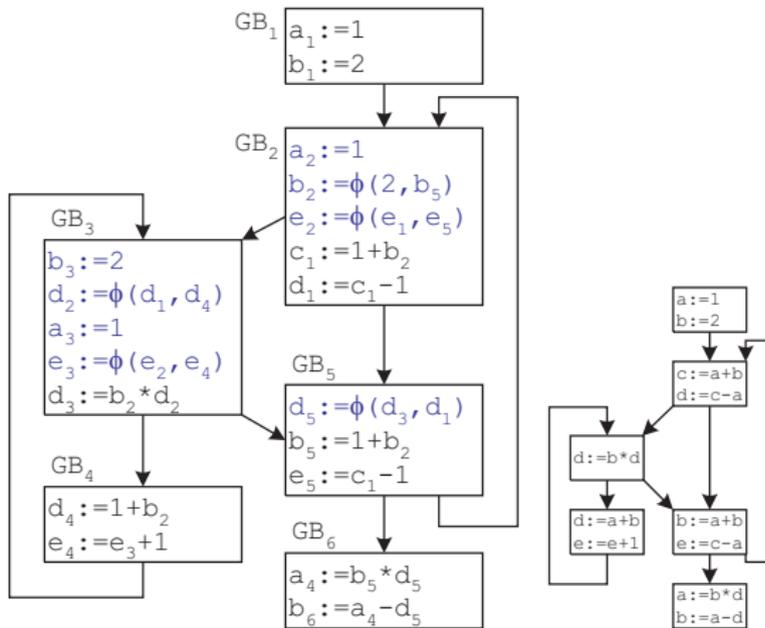
# SSA Aufbau Block 6



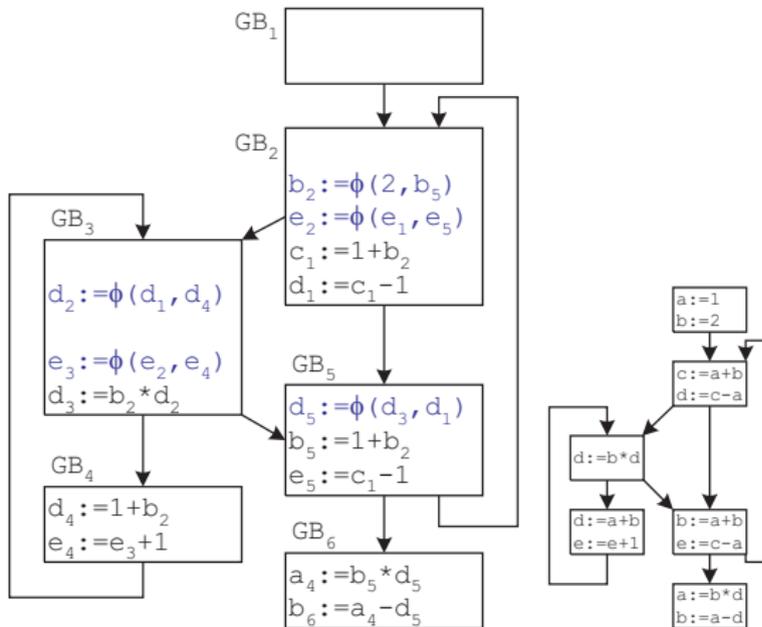
# Vereinfachungen: Kopienfortpflanzung



# Vereinfachungen: Konstantenfortpflanzung

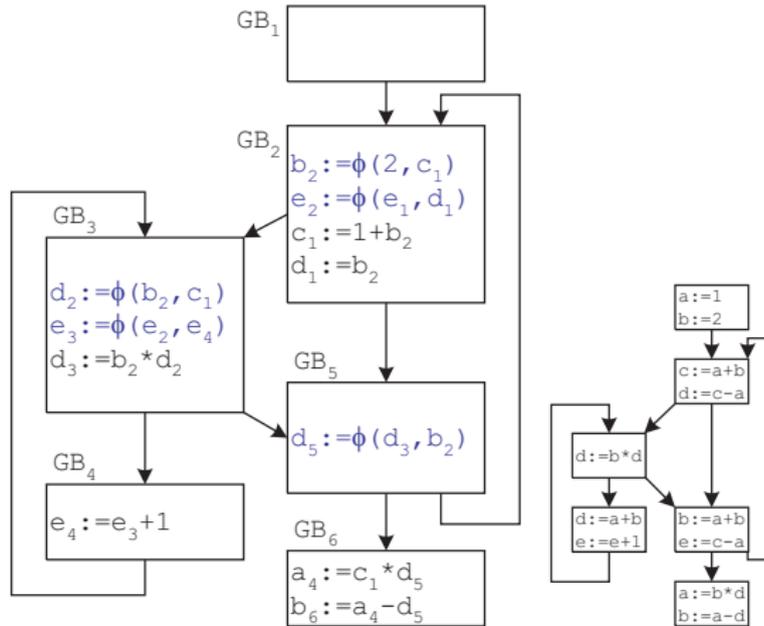


# Vereinfachungen: Toten Code eliminieren



# Weitere Vereinfachungen

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



# SSA-Aufbau aus dem AST

Ein Links-Rechts Baumdurchlauf:

- Halte aktuellen Grundblock in globaler Variablen
- Ausdrücke: Generiere SSA für aktuellen Grundblock, Zwischenergebnisse werden nur einmal verwendet, daher kein Holen/Merken von Wertnummern nötig!

Anweisungen:

- generiere Grundblöcke
- generiere Code für Grundblöcke
- Füge Ablauf der Grundblöcke zusammen
- Schließe SSA Aufbau für die Grundblöcke ab

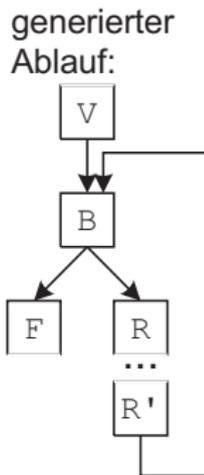
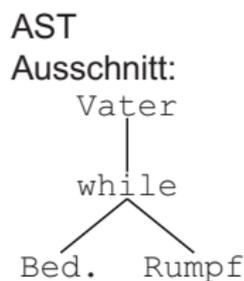
Prozeduraufrufe sind in diesem Zusammenhang Ausdrücke.

Bei Sprüngen: Schließe Grundblöcke mit Sprungmarken in einem zweiten Baumdurchlauf ab.



## Aufbau aus dem AST, Beispiel while

- Schließe SSA Aufbau für aktuellen Block ( $V$ ) ab
- Erzeuge neuen Block ( $B$ ) für Schleifenbedingung
- Füge Ablauf  $V \rightarrow B$  ein
- Erzeuge SSA-Code für Bedingung (rekursiver Abstieg)
- Erzeuge neuen aktuellen Block ( $R$ ) für Schleifenrumpf, merke Block  $B$
- Erzeuge SSA-Code für Rumpf (rekursiver Abstieg)
- Schließe SSA-Aufbau für aktuellen Block ( $R'$ ) ab
- Füge Ablauf  $R' \rightarrow B$  ein
- Schließe SSA-Aufbau für Block  $B$  ab
- Erzeuge neuen Block ( $F$ ) für Fortsetzung
- Füge Ablauf  $B \rightarrow F$  ein
- Rekursion kehrt zu Vater zurück und generiert weiteren Code in Block  $F$



# Zusammenfassung

- SSA bedeutet: statt Variablen dynamische Konstanten
- $\phi$ -Funktionen nötig bei Ablaufzusammenfluß
- $\phi$ -Funktionen werden an Dominanzgrenzen plziert
- Darstellung als Datenflußgraph
- Ermöglicht effiziente Formulierung intraprozeduraler Optimierungen die auf Datenflußanalysen aufbauen
- Aufbau der  $\phi$ -Funktionen: nur, wenn Wert verwendet; rekursiv für Vorgängerblöcke
- Endlosrekursionen vermeiden und Handhabung nicht fertiger Vorgängerblöcke mit  $\phi'$ -Funktionen

