Technische Universität Braunschweig

Institute of Operating Systems and Computer Networks
Reliable System Software

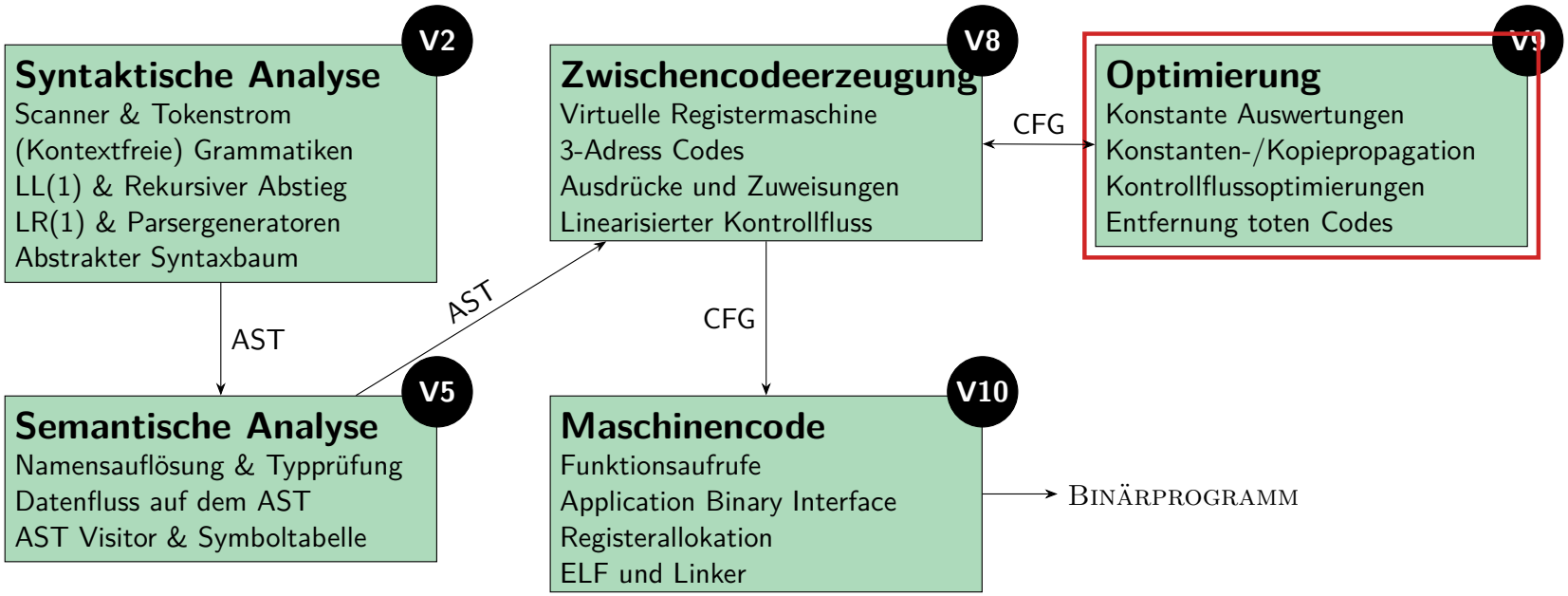
Programmiersprachen und Übersetzer

09 - Optimierung

Christian Dietrich

Sommersemester 2024

Einordnung in die Vorlesung: Optimierung



- Hardwareunabhängige Optimierung auf dem Zwischencode
  - IR-Code wurde ohne Rücksicht auf die effiziente Ausführbarkeit erstellt.
  - Effektive Entwickler: Welche Arbeiten kann er dem Optimierer überlassen?
  - Effiziente Entwickler: Wie lege ich dem Optimierer keine Steine in den Weg?

Motivation: Da geht noch was!

```
func main() : int {
     var a : int;
     a := 1 + 1;
     if (1) {
        a := a + 3;
        if (a >= 1) {
           a := 0;
        } else {
           a := -1;
        }
     }
     return a;
}
```

L0

Bei genauer Betrachtung sehen wir, dass dieses Programm **immer 0** zurückgeben wird.

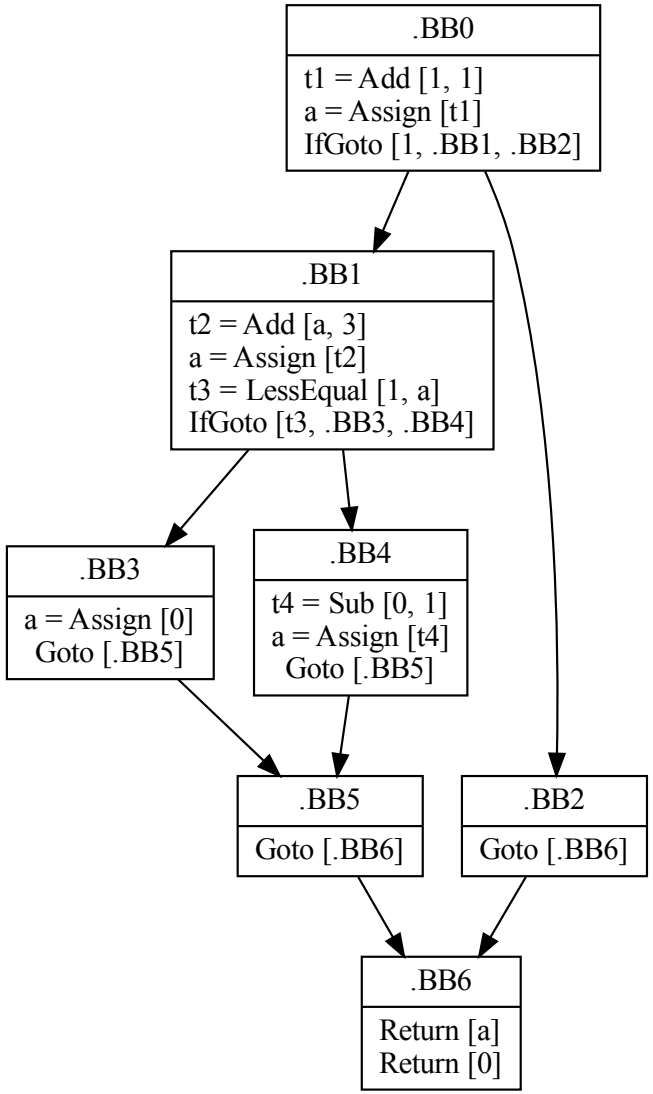
Motivation: Da geht noch was!

```
func main() : int {
     var a : int;
     a := 1 + 1;
     if (1) {
        a := a + 3;
        if (a >= 1) {
           a := 0;
        } else {
           a := -1;
        }
     }
     return a;
}
```

L0

Bei genauer Betrachtung sehen wir, dass dieses Programm **immer 0** zurückgeben wird.

Dafür wird echt viel gerechnet!

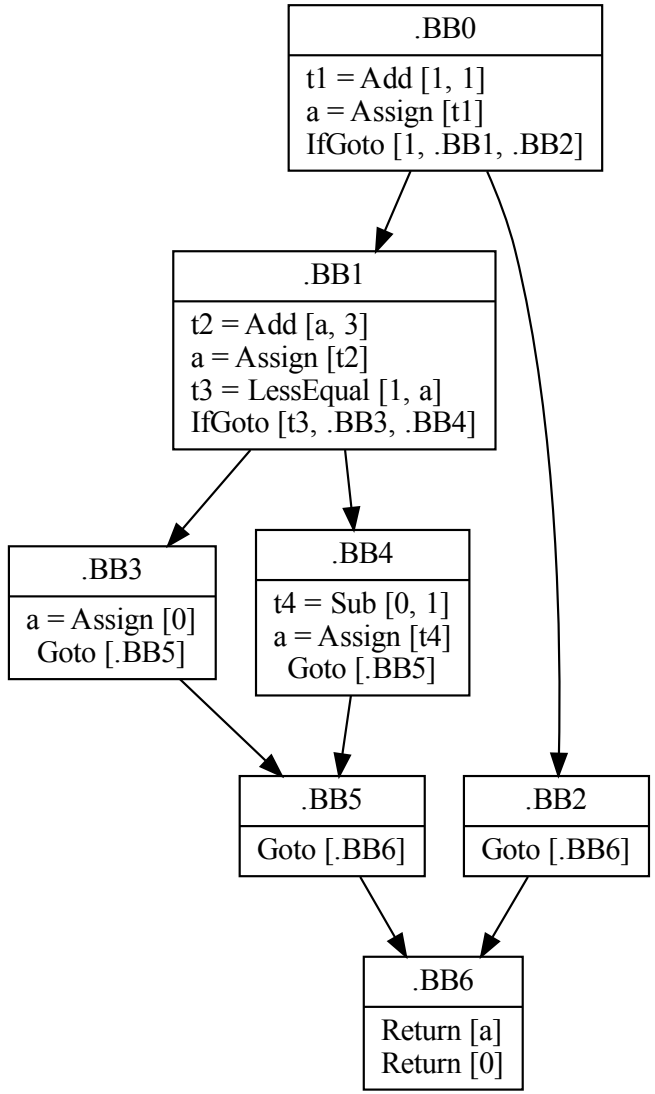


Motivation: Da geht noch was!

„Der Übersetzer schaut genau hin“

1. Konstante Auswertung

   „Bekanntes Vorausberechnen“



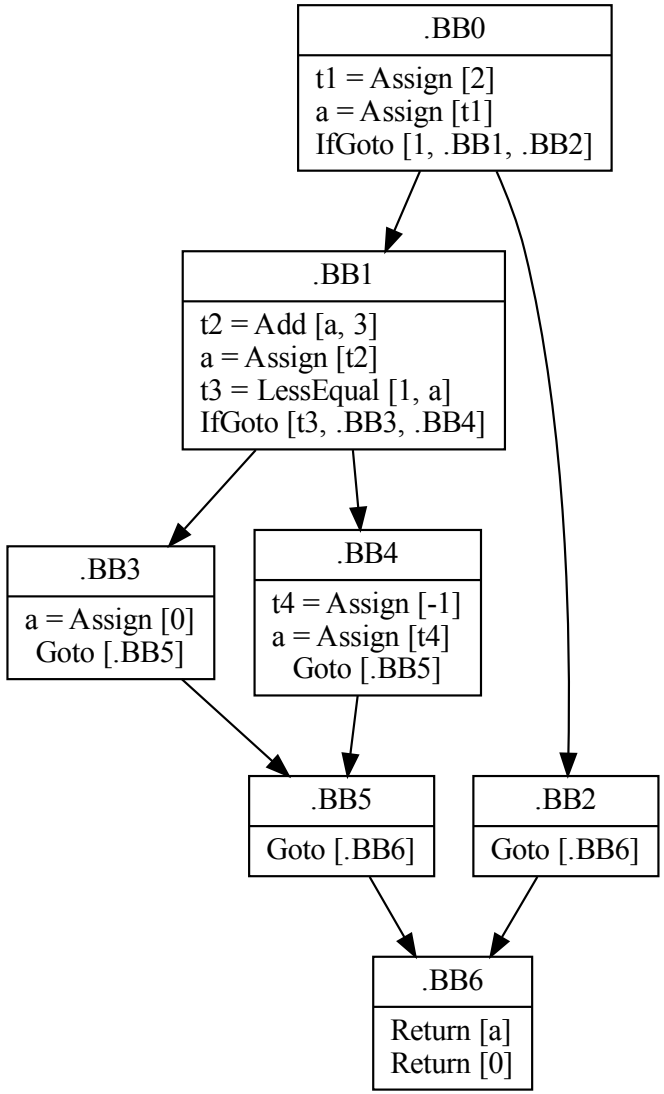
Motivation: Da geht noch was!

„Der Übersetzer schaut genau hin“

1. Konstante Auswertung

   „Bekanntes Vorausberechnen“

   - Arithmetische Berechnungen



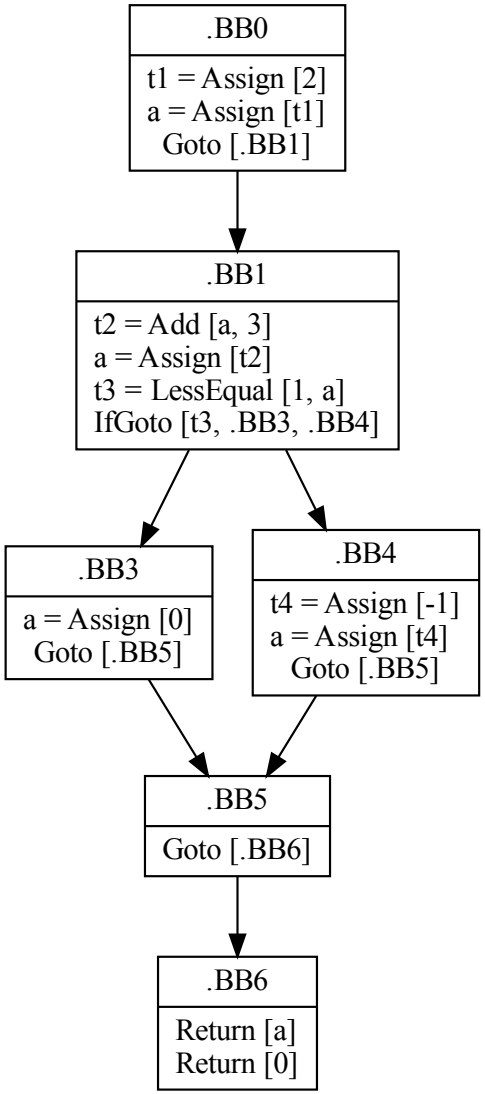
Motivation: Da geht noch was!

„Der Übersetzer schaut genau hin“

1. Konstante Auswertung

   „Bekanntes Vorausberechnen“

   - Arithmetische Berechnungen
   - Konstante Sprungbedingungen



Motivation: Da geht noch was!

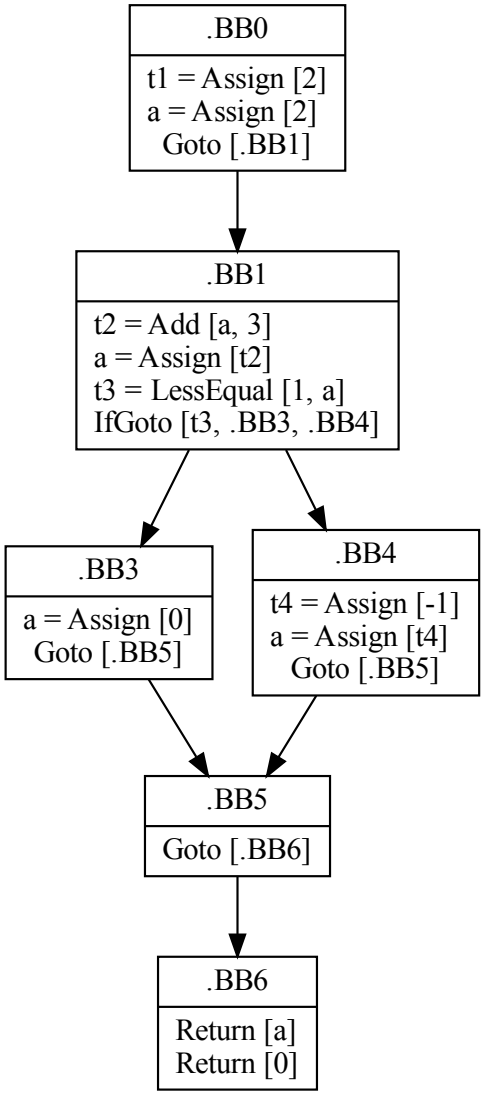
„Der Übersetzer schaut genau hin“

1. Konstante Auswertung

   „Bekanntes Vorausberechnen“

   - Arithmetische Berechnungen
   - Konstante Sprungbedingungen

2. Konstanten- und Kopiepropagation



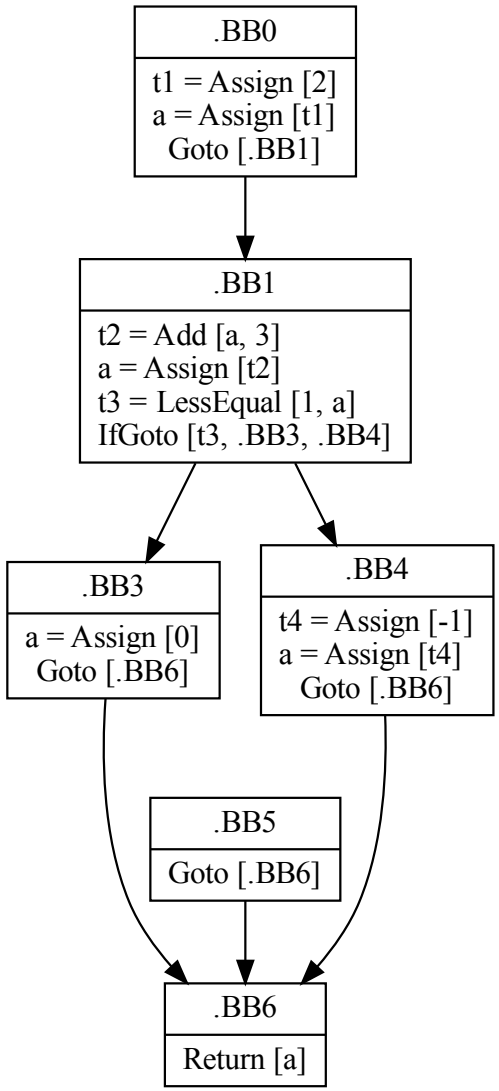
Motivation: Da geht noch was!

„Der Übersetzer schaut genau hin“

1. Konstante Auswertung

   „Bekanntes Vorausberechnen“

   - Arithmetische Berechnungen
   - Konstante Sprungbedingungen
2. Konstanten- und Kopiepropagation
3. Goto-auf-Goto eliminieren



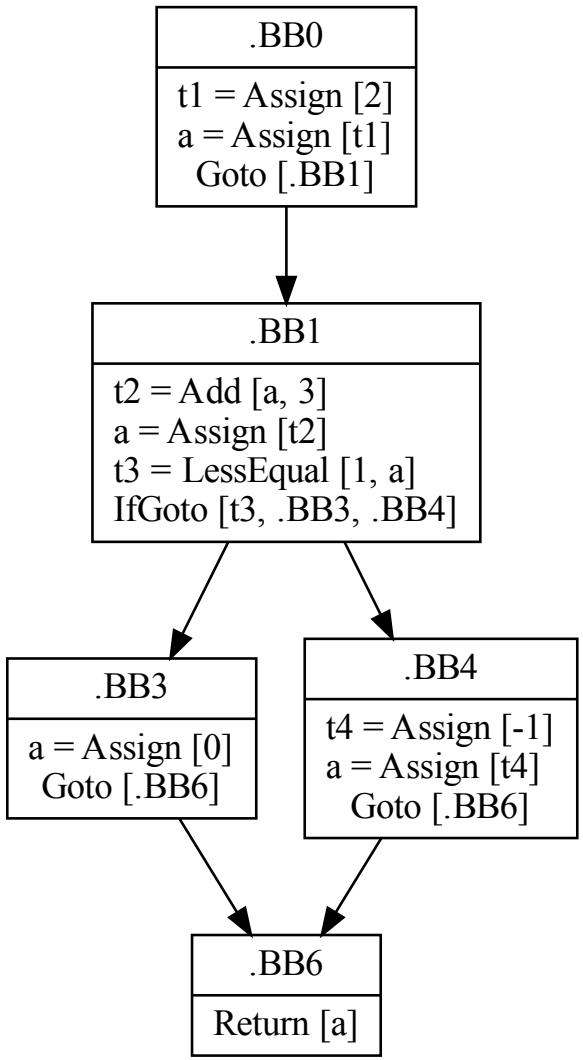
Motivation: Da geht noch was!

„Der Übersetzer schaut genau hin“

1. Konstante Auswertung

   „Bekanntes Vorausberechnen“

   - Arithmetische Berechnungen
   - Konstante Sprungbedingungen
2. Konstanten- und Kopiepropagation
3. Goto-auf-Goto eliminieren
4. Dead-Code Elimination



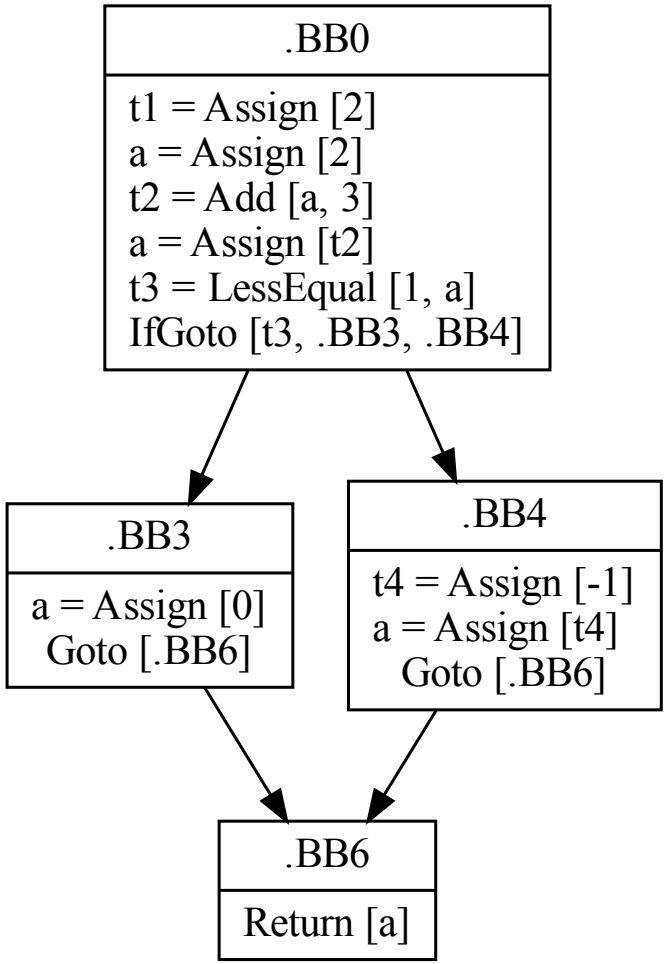
Motivation: Da geht noch was!

„Der Übersetzer schaut genau hin“

1. Konstante Auswertung

   „Bekanntes Vorausberechnen“

   - Arithmetische Berechnungen
   - Konstante Sprungbedingungen
2. Konstanten- und Kopiepropagation
3. Goto-auf-Goto eliminieren
4. Dead-Code Elimination
5. Basic-Block Merge



Motivation: Da geht noch was!

„Der Übersetzer schaut genau hin“

.BB0
Return [0]

1. Konstante Auswertung

   „Bekanntes Vorausberechnen“

   - Arithmetische Berechnungen
   - Konstante Sprungbedingungen
2. Konstanten- und Kopiepropagation
3. Goto-auf-Goto eliminieren
4. Dead-Code Elimination
5. Basic-Block Merge

6. …repeat, until no change.

Ziele bei IR-Level Optimierung

Allgemeine Optimierungsziele

Ziel jeder Programm-Optimierung ist es, die **nicht-funktionalen** Eigenschaften, unter Beibehaltung der **erwarteten Funktionalität**, zu verbessern.

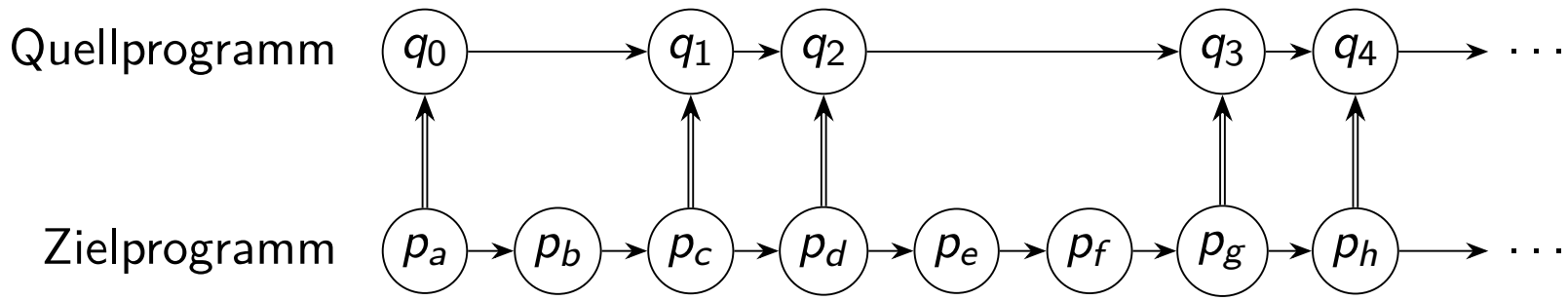
- Nicht-Funktionale Eigenschaften nach denen wir optimieren
  - Ausführungszeit: Abbildung auf weniger oder schnellere Instruktionen
  - Programmgröße: Verringerung der Binärgröße (eingebettete Systeme)
  - Energieverbrauch: Verlängerung der Batterielaufzeit
- Auf der Hardware-unabhängigen IR-Ebene fehlt das Kostenmodell
  - Wir wissen nicht, wie schnell, groß oder Energie-hungrig einzelne Instruktionen auf der Zielplattform sein werden.
  - Heuristik: „Speicherzugriffe und Sprünge sind teuer."

$\Rightarrow$ Generelle Reduktion der Anzahl der Instruktionen und Basisblöcke

Wiederholung (1. Vorlesung): Korrekte Übersetzung

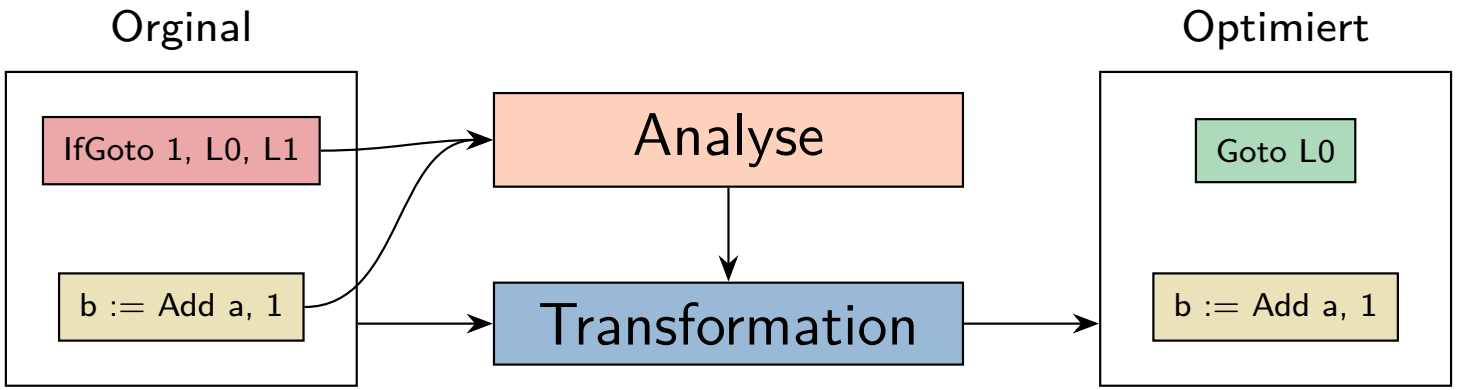
- Die Programmiersprache definiert die beobachtbaren Zustände.
  - Ein- und Ausgabe ist immer beobachtbar, oft aber auch Teile des Speichers
  - Bei C: Nur das Schreiben von globalen Variablen, nicht aber von lokalen

- Ein korrekter Übersetzer erhält alle beobachtbaren Zustände.
  - Parsing+Code-Erzeugung ist eine **Abbildungsfunktion**:



⇒ Funktionen dürfen verändert werden, solange sie außen gleich aussehen.
  - Im IR-Code: lokale und temporäre Variablen sind nicht sichtbar
  - Von Außen sichtbar sind: `Store`, `Call`, `Return`

Phasen und Umfang der Optimierung



- Optimierung besteht immer aus **zwei Schritten**
  - Analyse des Programms generiert Wissen, anhand dessen wir entscheiden können, ob wir eine Optimierung anwenden dürfen.
  - Transformation in ein semantisch äquivalentes, aber „besseres“ Programm.
- Optimierungen analysieren und transformieren **Programmausschnitte**
  - Wie groß ist der Ausschnitt, über den wir Wissen extrahieren?
  - Instruktionen, einzelne Basisblöcke, eine ganze Funktion, eine Aufrufhierarchie, mehrere interagierende Threads

Algebraische Identitäten

Technische Universität Braunschweig

Ersetzung von Instruktionen durch algebraisch äquivalente Instruktionen

- Konstantenfaltung ist eine Instruktions-lokale Optimierung

```
a := Add 1, 2
```
$\Rightarrow$
```
a := Assign 3
```

  - **Ziel**: Ersetzung von Instruktionen mit ausschließlich konstanten Operanden
  - Sprach-Semantik muss erhalten (32-Bit Programm vs. 64-Bit Übersetzer)

- Strength-Reduction verringert die Ausdrucksstärke einer Instruktion

```
a := Mul x, 2
```
$\Rightarrow$
```
a := Add x, x
```

  - Arithmetische Operationen haben unterschiedliche theoretische Komplexitäten
  - Multiplikation: $\mathcal{O}(n \cdot \log(n) \cdot \log(\log(n)))$ (FFT) vs. Addition: $\mathcal{O}(n)$
  - Ausdrucksstärke: Potenzierung $\gg$ Multiplikation $\gg$ Addition $\gg$ Bit-Shift

- Technische Umsetzung: Wende Ersetzungsregeln auf jede Instruktion an
  - Ersetzungsmuster gibt Bedingungen vor, die Operanden erfüllen müssen
  - Ersetze die Instruktion im umgebenden Basisblock

Algebraische Identitäten: Einige Ersetzungsmuster

Variablen: A, Konstanten: x, Labels: L

Orginal	Bedingungen	Ersetzung
Null-Operationen		
A := Add B, 0	--	A := Assign B
A := Mul B, 1	--	A := Assign B
A := Sub B, B	--	A := Assign 0
Konstantenfaltung		
A := Add b, c	--	A := Assign b+c
IfGoto x, A, B	x !=0	Goto A
IfGoto x, A, B	x ==0	Goto B
Strength-Reduction		
A := Pow B, 2	--	A := Mul B, B
A := Mul B, n	$n == 2^x$	A := LShift B, x

Konstanten- und Kopiepropagation

```
t1 := Add 1, 1
a := Assign t1
Goto .BB1                    BB0
```

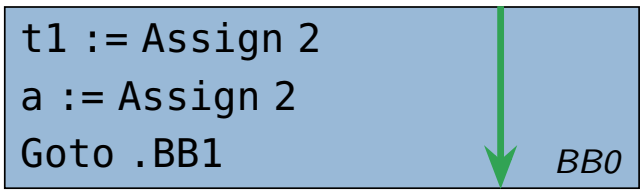
Durch die einfache Codeerzeugung und bei der Optimierung entstehen konstante oder redundante Zuweisungen.

Konstanten- und Kopiepropagation

```
t1 := Assign 2
a := Assign t1
Goto .BB1                    BB0
```

Durch die einfache Codeerzeugung und bei der Optimierung entstehen konstante oder redundante Zuweisungen.

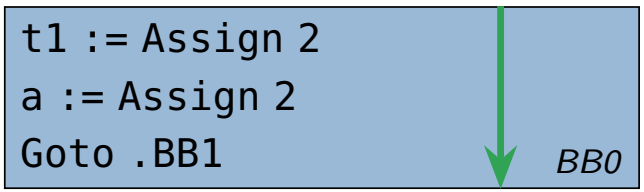
Konstanten- und Kopiepropagation



Durch die einfache Codeerzeugung und bei der Optimierung entstehen konstante oder redundante Zuweisungen.

- Propagation bekannter Werte durch einen Basisblock
  - Zum Zeitpunkt einer Zuweisung werden rechte und linke Seite äquivalent.
  - Operanden-Ersetzung in den folgenden Instruktionen
  - **Aber**: Wir dürfen nur solange ersetzen, wie die Äquivalenz sicher ist.

Konstanten- und Kopiepropagation



Durch die einfache Codeerzeugung und bei der Optimierung entstehen konstante oder redundante Zuweisungen.

- Propagation bekannter Werte durch einen Basisblock
  - Zum Zeitpunkt einer Zuweisung werden rechte und linke Seite äquivalent.
  - Operanden-Ersetzung in den folgenden Instruktionen
  - **Aber**: Wir dürfen nur solange ersetzen, wie die Äquivalenz sicher ist.
- Intuition für die Propagation von Konstanten und Werten
  - Iteration über die Instruktionen eines BBs, Mitführen von Äquivalenzmengen
  - Zuweisungen etablieren eine neue Äquivalenz zwischen LHS und RHS
  - Andere Schreibzugriffe, die eine Variable verändern, löschen Äquivalenzen
  - Bei Lesezugriffen wird die Äquivalenzliste befragt und ggf. ersetzt.

Vorgehen für Wertpropagation

Äquivalenzmengen

```
x := Assign 2
y := Assign x
x := Add x, y
t0 := Assign x
t1 := Call f, t0, y
x := Add y, t1
```

- Äquivalenzmengen werden „durchgeschoben“, modifiziert und befragt

Vorgehen für Wertpropagation

Äquivalenzmengen

```
x := Assign 2          ← []
y := Assign x
x := Add x, y
t0 := Assign x
t1 := Call f, t0, y
x := Add y, t1
```

- Äquivalenzmengen werden „durchgeschoben“, modifiziert und befragt
  1. Zu Beginn des Blocks ist nichts zu nichts anderem äquivalent

Vorgehen für Wertpropagation

Äquivalenzmengen

```
x := Assign 2          ← []
y := Assign x          ← [{x, 2}]
x := Add x, y
t0 := Assign x
t1 := Call f, t0, y
x := Add y, t1
```

- Äquivalenzmengen werden „durchgeschoben“, modifiziert und befragt
  1. Zu Beginn des Blocks ist nichts zu nichts anderem äquivalent
  2. Bei Zuweisungen werden linke und rechte Seite äquivalent

Vorgehen für Wertpropagation

Äquivalenzmengen

```
x := Assign 2          ← []
y := Assign 2          ← [{x, 2}]
x := Add x, y          ← [{x, y, 2}]
t0 := Assign x
t1 := Call f, t0, y
x := Add y, t1
```

- Äquivalenzmengen werden „durchgeschoben“, modifiziert und befragt
  1. Zu Beginn des Blocks ist nichts zu nichts anderem äquivalent
  2. Bei Zuweisungen werden linke und rechte Seite äquivalent
  3. Quelloperanden werden durch äquivalente Konstanten oder Variablen ersetzt
  4. Äquivalenzmengen können mehr als 2 Elemente haben

Vorgehen für Wertpropagation

Äquivalenzmengen

```
x := Assign 2          ← []
y := Assign 2          ← [{x, 2}]
x := Add 2, 2          ← [{x, y, 2}]
t0 := Assign x         ← [{y, 2}]
t1 := Call f, t0, y
x := Add y, t1
```

- Äquivalenzmengen werden „durchgeschoben“, modifiziert und befragt
  1. Zu Beginn des Blocks ist nichts zu nichts anderem äquivalent
  2. Bei Zuweisungen werden linke und rechte Seite äquivalent
  3. Quelloperanden werden durch äquivalente Konstanten oder Variablen ersetzt
  4. Äquivalenzmengen können mehr als 2 Elemente haben
  5. Nicht-`Assign` Schreiboperationen entfernen Elemente aus ihrer Menge

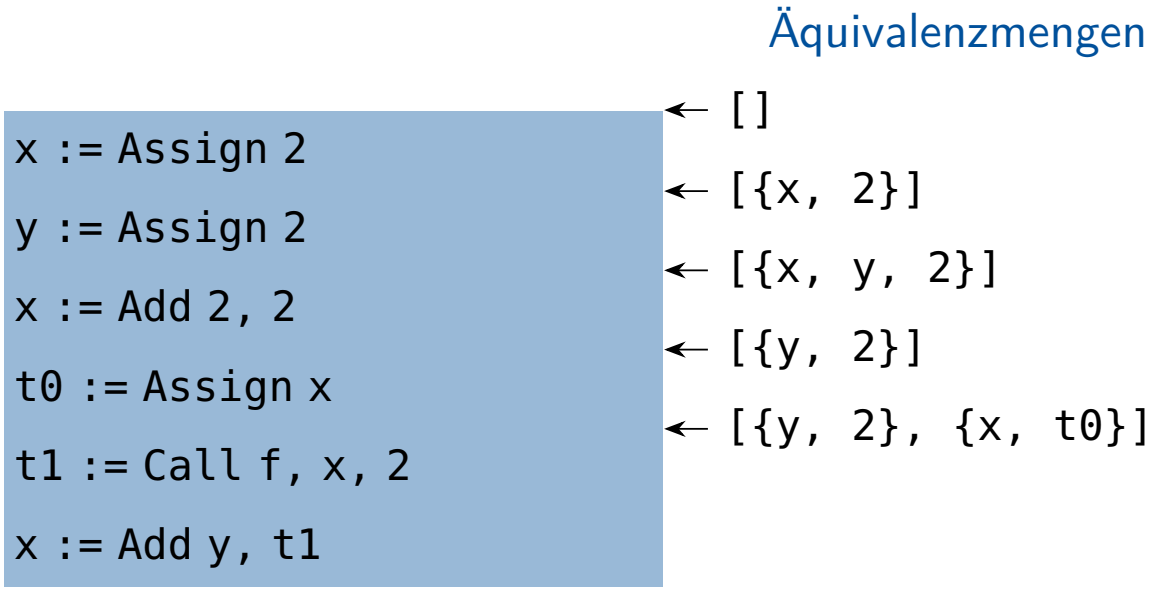
Vorgehen für Wertpropagation

Äquivalenzmengen

```
x := Assign 2          ← []
y := Assign 2          ← [{x, 2}]
x := Add 2, 2          ← [{x, y, 2}]
t0 := Assign x         ← [{y, 2}]
t1 := Call f, t0, y    ← [{y, 2}, {x, t0}]
x := Add y, t1
```

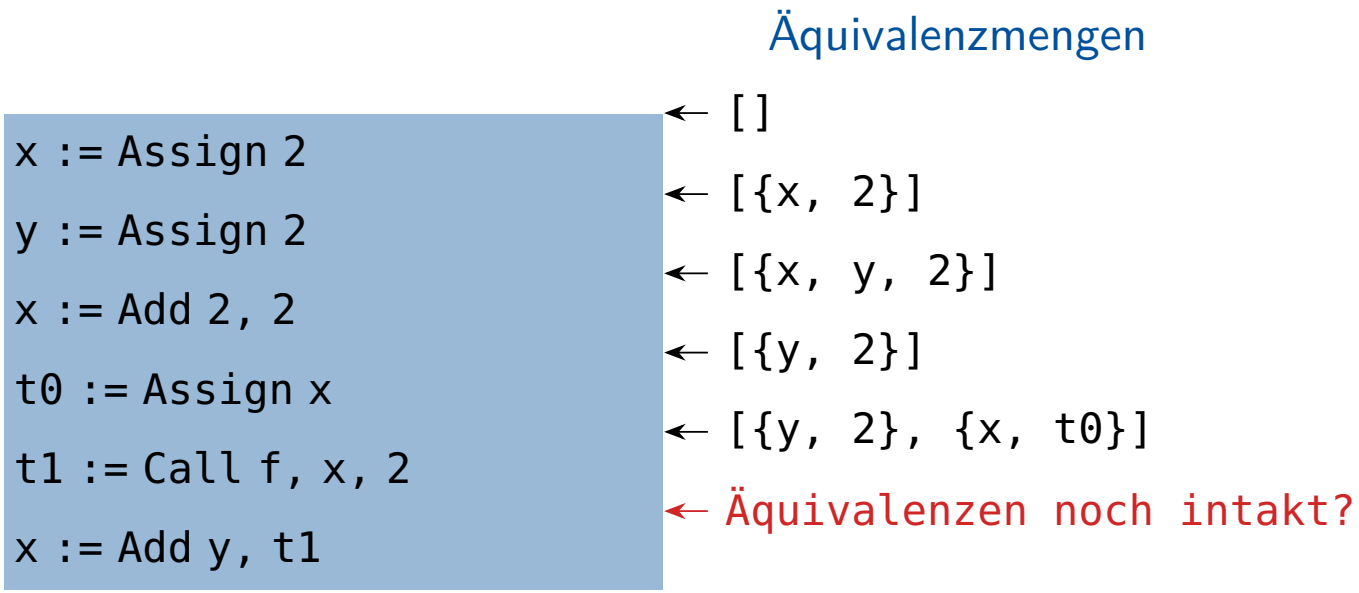
- Äquivalenzmengen werden „durchgeschoben“, modifiziert und befragt
  1. Zu Beginn des Blocks ist nichts zu nichts anderem äquivalent
  2. Bei Zuweisungen werden linke und rechte Seite äquivalent
  3. Quelloperanden werden durch äquivalente Konstanten oder Variablen ersetzt
  4. Äquivalenzmengen können mehr als 2 Elemente haben
  5. Nicht-`Assign` Schreiboperationen entfernen Elemente aus ihrer Menge

Vorgehen für Wertpropagation



- Äquivalenzmengen werden „durchgeschoben“, modifiziert und befragt
  1. Zu Beginn des Blocks ist nichts zu nichts anderem äquivalent
  2. Bei Zuweisungen werden linke und rechte Seite äquivalent
  3. Quelloperanden werden durch äquivalente Konstanten oder Variablen ersetzt
  4. Äquivalenzmengen können mehr als 2 Elemente haben
  5. Nicht-`Assign` Schreiboperationen entfernen Elemente aus ihrer Menge

Vorgehen für Wertpropagation



- Äquivalenzmengen werden „durchgeschoben“, modifiziert und befragt
  1. Zu Beginn des Blocks ist nichts zu nichts anderem äquivalent
  2. Bei Zuweisungen werden linke und rechte Seite äquivalent
  3. Quelloperanden werden durch äquivalente Konstanten oder Variablen ersetzt
  4. Äquivalenzmengen können mehr als 2 Elemente haben
  5. Nicht-`Assign` Schreiboperationen entfernen Elemente aus ihrer Menge

Kann die Funktion f, die Variable x, y, oder t0 verändern?

Alias-Problematik bei der Datenflussanalyse

- Das Problem sind Zeiger, die unsere IR-Variablen verändern könnten.
  - Wenn `ptr` auf `y` zeigt, kann sich `y` ändern, ohne explizit Operand zu sein.
  - Im Funktions-lokalen Scope wissen wir nicht, was `func()` mit `ptr` macht.
  - Die `Store`-Instruktion könnte `y` auf 3 setzen.
- Dies ist das Alias-Problem für Zeiger.
  - Wir müssten genau wissen, auf welche Objekte ein Zeiger verweisen kann.
  - Dies erfordert aufwendige, teils inter-prozedurale, Analysen.
  - Beim geringsten Zweifel über die Alias-Relation **wirft der Optimierer hin**.
- Einfache Heuristik für unsere Wertepropagation
  - Bei jedem `Store` und bei jedem `Call` werfen wir die Äquivalenzmengen weg. Better safe, than sorry.

Alias-Problematik bei der Datenflussanalyse

Take-Away für den effizienten Programmierer

- Wildes Zeiger-rum-gefuchtel macht dem Übersetzer das Leben schwer.
- Variablen sind umsonst, Wertepropagation eliminiert Zuweisungen.

⇒ Programmieren Sie **verständlich**, nicht „**optimiert**“!

- Wir mussten genau wissen, auf welche Objekte ein Zeiger verweisen kann.
- Dies erfordert aufwendige, teils inter-prozedurale, Analysen.
- Beim geringsten Zweifel über die Alias-Relation **wirft der Optimierer hin**.

- Einfache Heuristik für unsere Wertepropagation
  - Bei jedem `Store` und bei jedem `Call` werfen wir die Äquivalenzmengen weg. Better safe, than sorry.

Wertepropagation über Basisblockgrenzen

Technische Universität Braunschweig

Bisher: Wertepropagation (CVP) innerhalb eines Basisblocks

```
CVP :: BasicBlock ↦ ()
```

- Wertepropagation ist eine Datenflussanalyse
  - Wir verfolgen, wie Datenwerte entlang des Kontrollflusses „fließen“.
  - In einem Basisblock ist der Kontrollfluss determiniert linear.
  - Die CVP propagiert eine Äquivalenzmenge durch einen Block.

Wertepropagation über Basisblockgrenzen

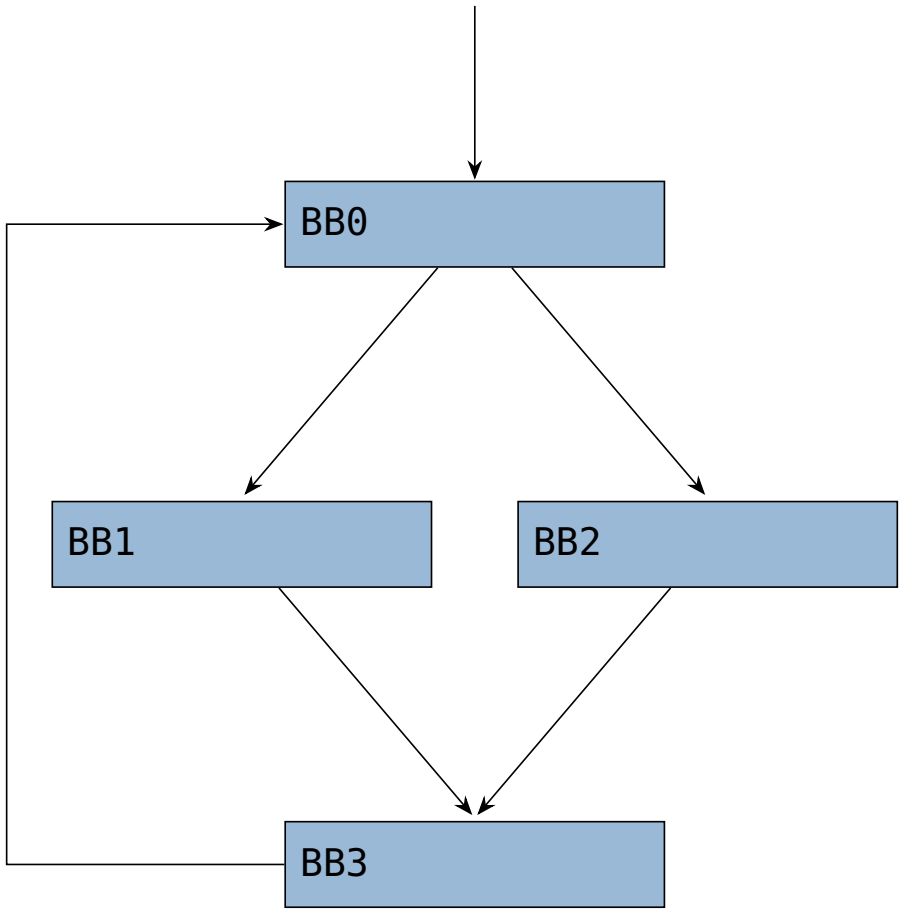
Technische Universität Braunschweig

Bisher: Wertepropagation (CVP) innerhalb eines Basisblocks

$$\text{CVP} :: \text{BasicBlock}, \text{Equivalences} \mapsto \text{Equivalences}$$

- Wertepropagation ist eine Datenflussanalyse
  - Wir verfolgen, wie Datenwerte entlang des Kontrollflusses „fließen“.
  - In einem Basisblock ist der Kontrollfluss determiniert linear.
  - Die CVP propagiert eine Äquivalenzmenge durch einen Block.
- Wir wollen die CVP auf eine ganze Funktion anwenden.
  - Blöcke werden zu Transformator-Funktionen für Äquivalenzmengen.
  - Äquivalenzmengen „fließen“ entlang der Kontrollflusskanten.
  - Fixpunkt-Iteration bis keine Änderungen mehr auftreten

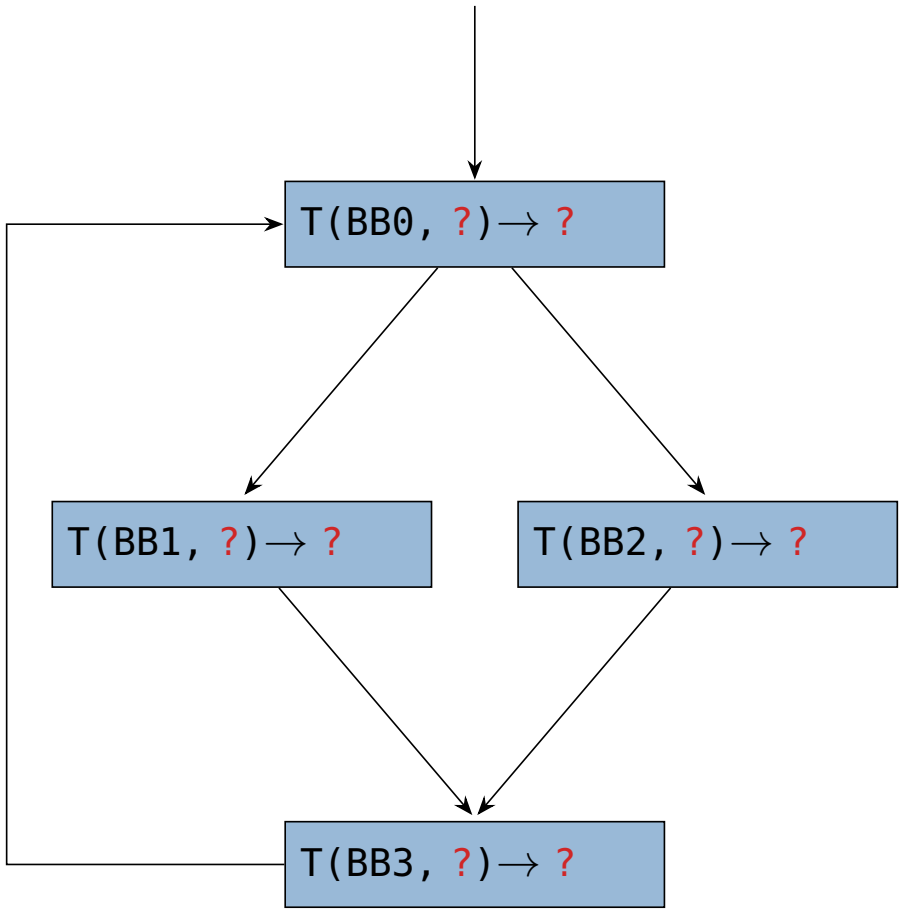
Fixpunktiteration auf einem CFG



Problem: Wie wenden wir eine Block-lokale Datenflussanalyse (z.B. CVP) auf eine ganze Funktion an?

Elemente

Fixpunktiteration auf einem CFG

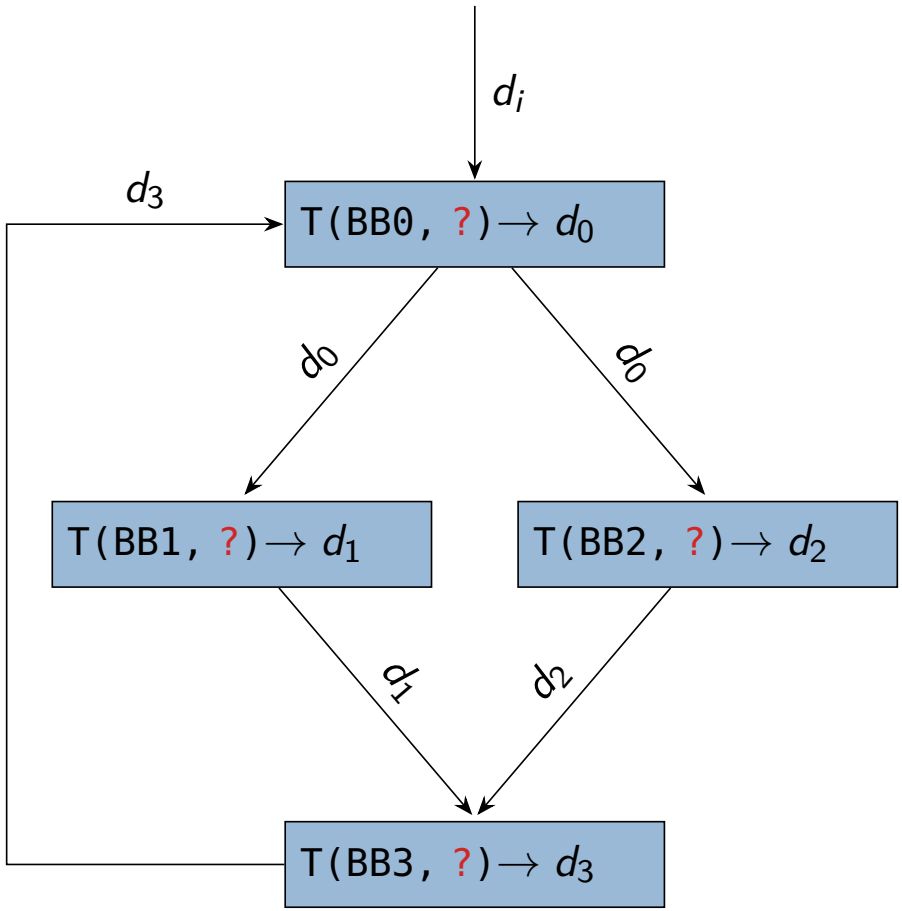


Problem: Wie wenden wir eine Block-lokale Datenflussanalyse (z.B. CVP) auf eine ganze Funktion an?

Elemente

- Transformation-Funktion: `T()`

Fixpunktiteration auf einem CFG

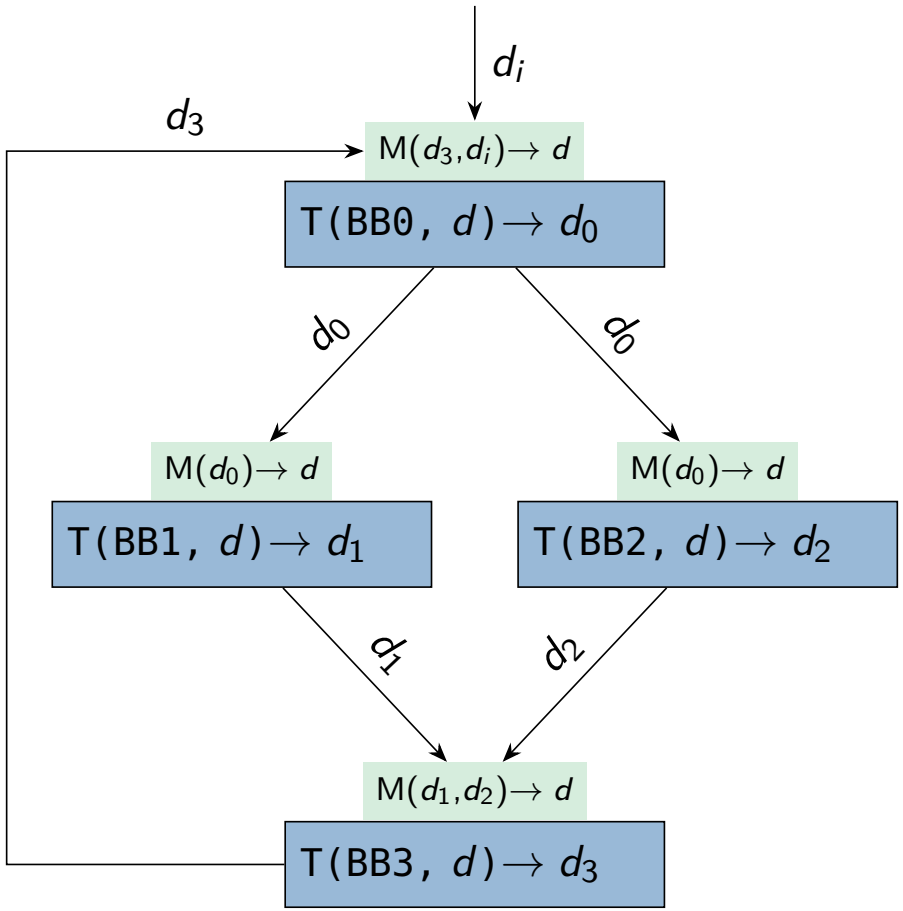


Problem: Wie wenden wir eine Block-lokale Datenflussanalyse (z.B. CVP) auf eine ganze Funktion an?

Elemente

- Transformation-Funktion: `T()`
- Propagierte Zustände: d_n

Fixpunktiteration auf einem CFG

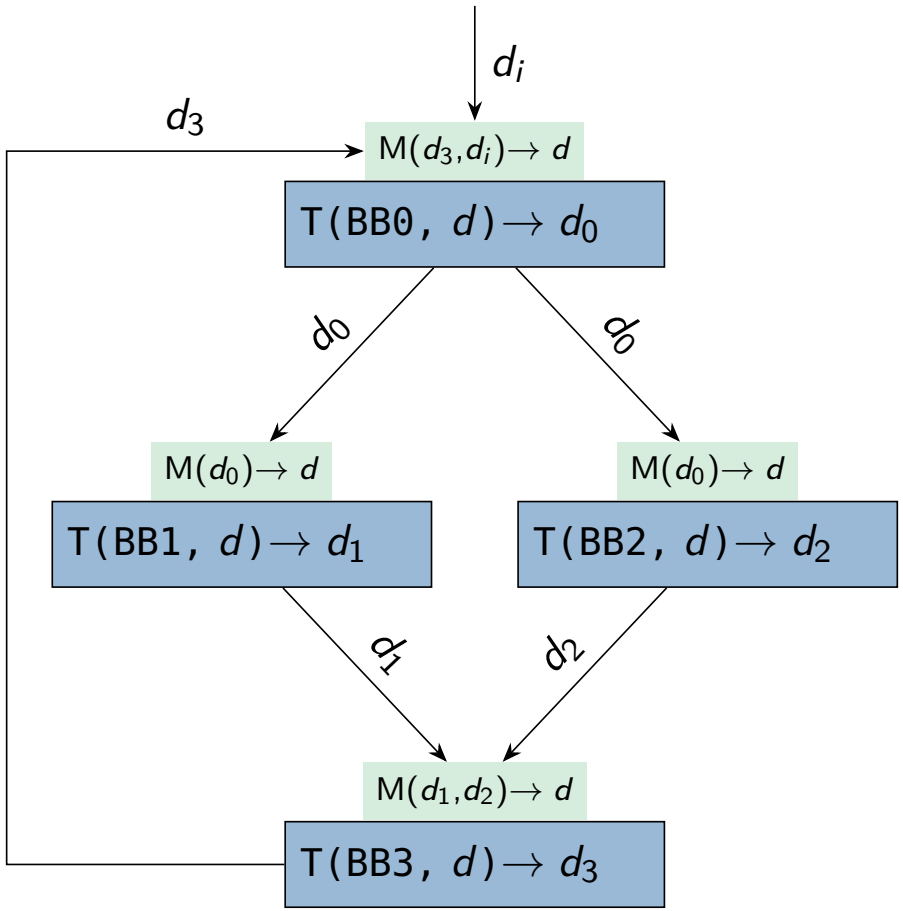


Problem: Wie wenden wir eine Block-lokale Datenflussanalyse (z.B. CVP) auf eine ganze Funktion an?

Elemente

- Transformation-Funktion: `T()`
- Propagierte Zustände: d_n
- Zustands-Merge-Funktion: `M()`

Fixpunktiteration auf einem CFG



Problem: Wie wenden wir eine Block-lokale Datenflussanalyse (z.B. CVP) auf eine ganze Funktion an?

Elemente

- Transformation-Funktion: `T()`
- Propagierte Zustände: d_n
- Zustands-Merge-Funktion: `M()`

Vorgehen

- Initialisiere Zustände d_n
- Iterierte Auswertung von `M(), T()`
- Updaten der Block-Zustände
- Stop, wenn Fixpunkt erreicht ist

Fixpunktiteration als Worklist-Algorithmus

Technische Universität Braunschweig

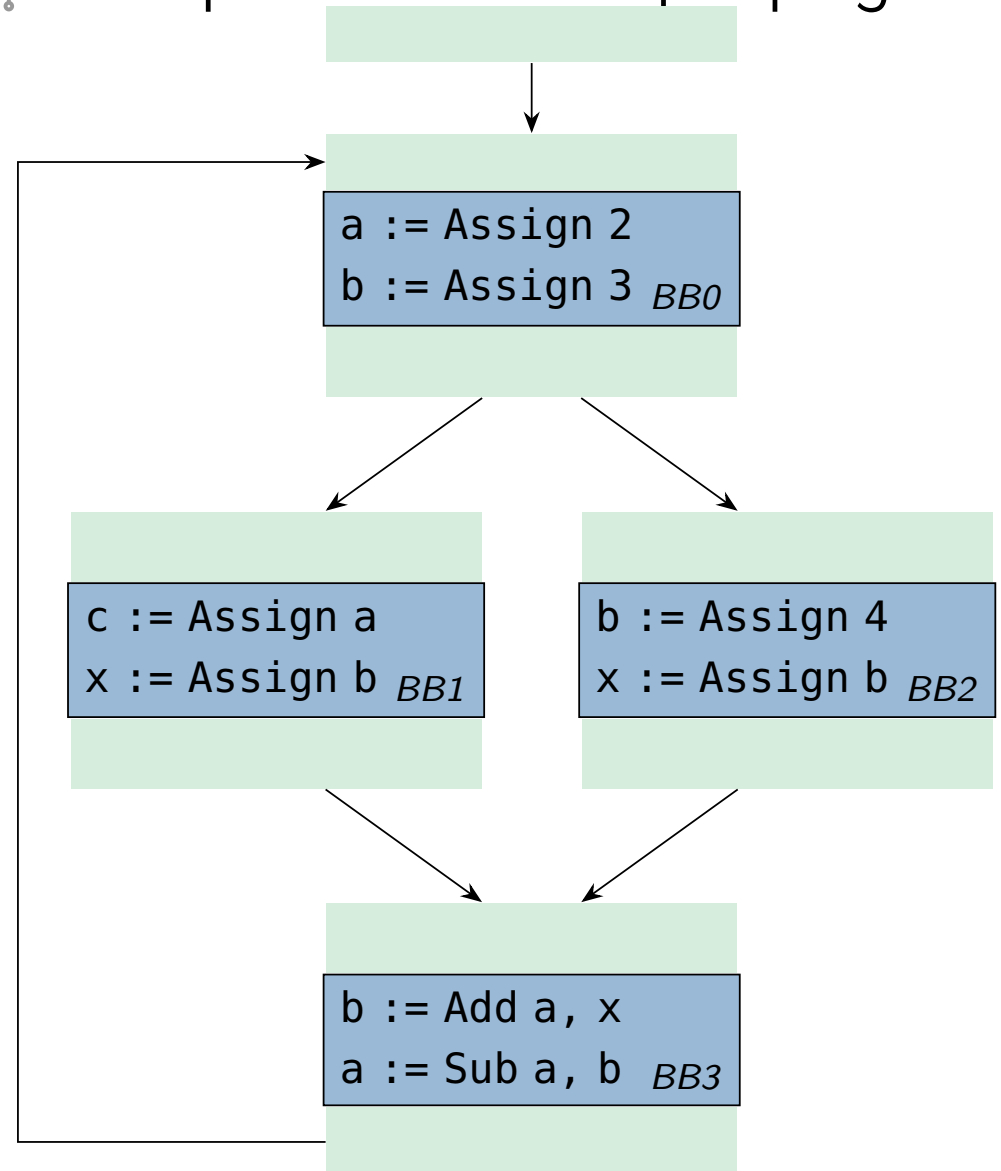
```python
# init():      Initialer Zustand für Node
# merge():     Zustands-Merge-Funktion
# transform(): Block-Transformation Funktion
# graph:       Der Graph, den wir bearbeiten wollen
def fixpoint(init, merge, transform, graph):

    states = {}                      # Verwende die Initialisierungsfunktion init(),
    for node in graph.nodes:         # um einen initialen Zustand für jeden Knoten
        states[node] = init(node)    # zu präparieren.

    worklist = list(graph.nodes)     # Worklist-Algorithmus: Laufe solange, bis
    while worklist != []:            # die Worklist leer ist. Zu Beginn besuchen
        node = worklist.pop()        # wir jeden Knoten.

        d_ins = []                                # In jedem Schritt sammeln wir die
        for pred in graph.predecessors[node]: # Zustände der Vorgängerknoten ein
            d_ins.append(states[pred])            # und kombinieren diese mit der
        d_in = merge(d_ins)                       # Zustands-Merge Funktion merge()

        d_out = transform(node, d_in)    # Wende die Transformations-Funktion an
        if d_out == states[node]:        # Ändert sich der Knoten-Zustand nicht,
            continue                     # sind wir für diesen Schritt fertig.

        states[node] = d_out                  # Ändert sich der Zustand eines
        for succ in graph.successors[node]:   # Knotens, werden alle Folge-Knoten
            worklist.append(succ)             # noch einmal besucht.
```

Python

Fixpunktiteration als Datenflussanalyse

- Voraussetzungen für die Terminierung des Worklist-Algorithmus
  - Init+Merge+Transform muss Knotenzustände monoton ändern
  - Beliebte Datenstrukturen für diesen Zustand: Bitvektoren, Mengen
  - Beispiel für eine monotone Zustands-Transformationen:
    - `init()`: Generiere eine leere Bitvektoren
    - `merge()`: Bitweises ODER
    - `transform()`: Bits werden gesetzt
- Verschiedene Geschmacksrichtungen
  - Vorwärts-Analyse: Zustände werden in Richtung der CFG-Kanten propagiert
  - Rückwärts-Analyse: Zustände werden entgegen der CFG-Kanten propagiert
- Wertepropagation als Fixpunktanalyse
  - Initialisierung mit leeren Äquivalenzmengen
  - Transformation wie gehabt mit `CVP(BasicBlock, Equivalences)`
  - Merge: Zwei Werte sind genau dann äquivalent,
    wenn sie in allen Eingabemenge äquivalent sind.

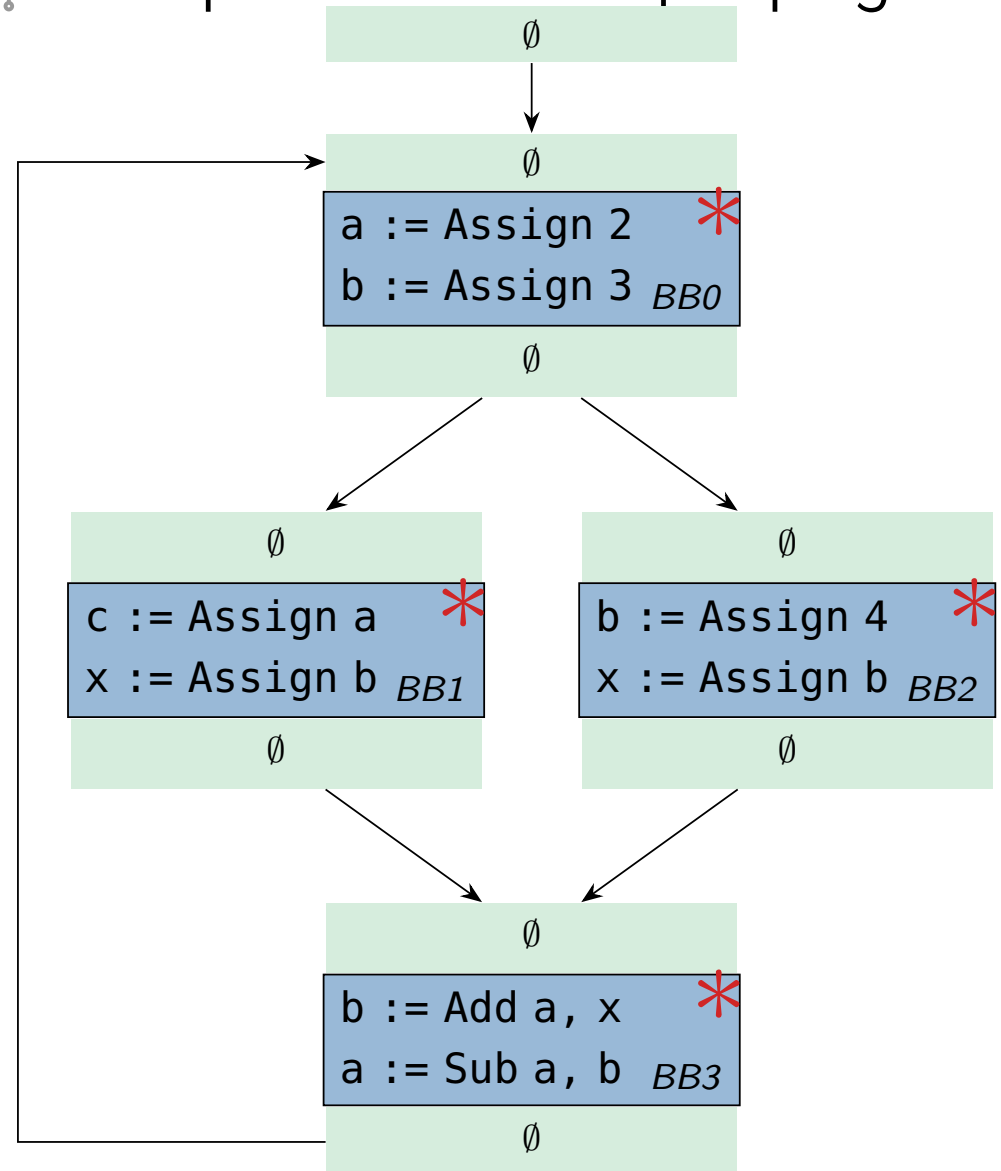
Beispiel für Wertepropagation



Worklist

[ ]

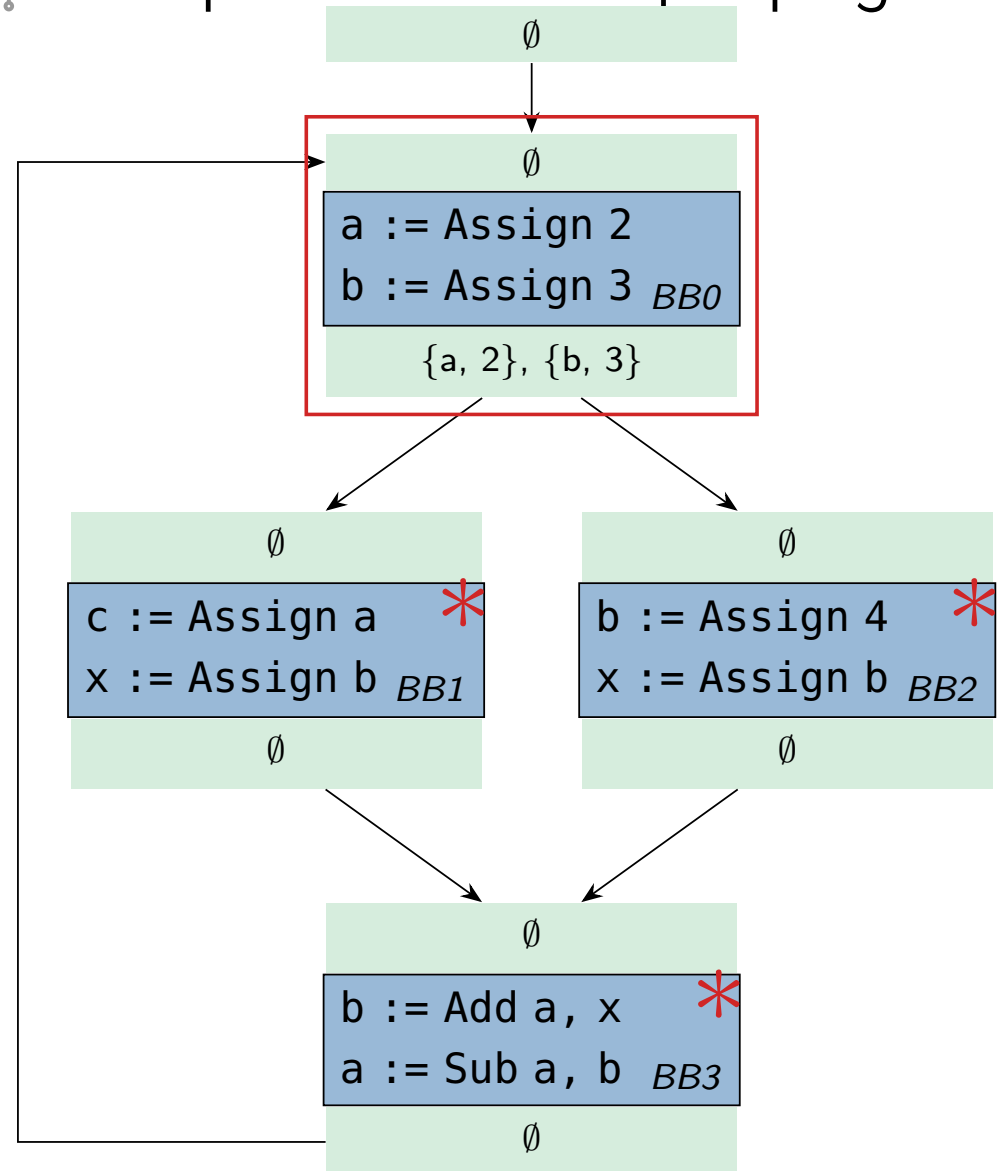
Beispiel für Wertepropagation



Worklist

```
[BB0, BB1, BB3, BB2]
```

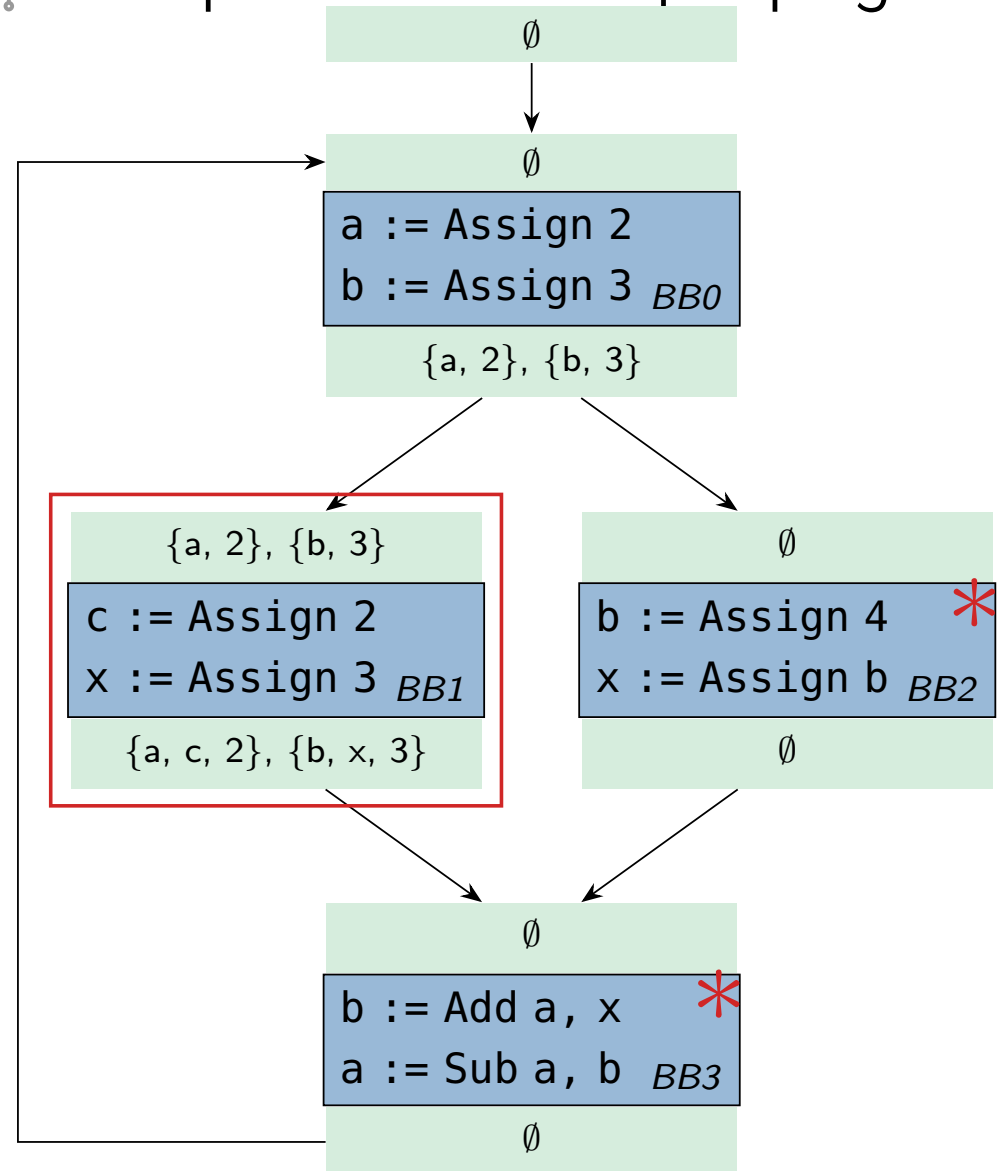
Beispiel für Wertepropagation



Worklist

```
[BB1, BB3, BB2]
```

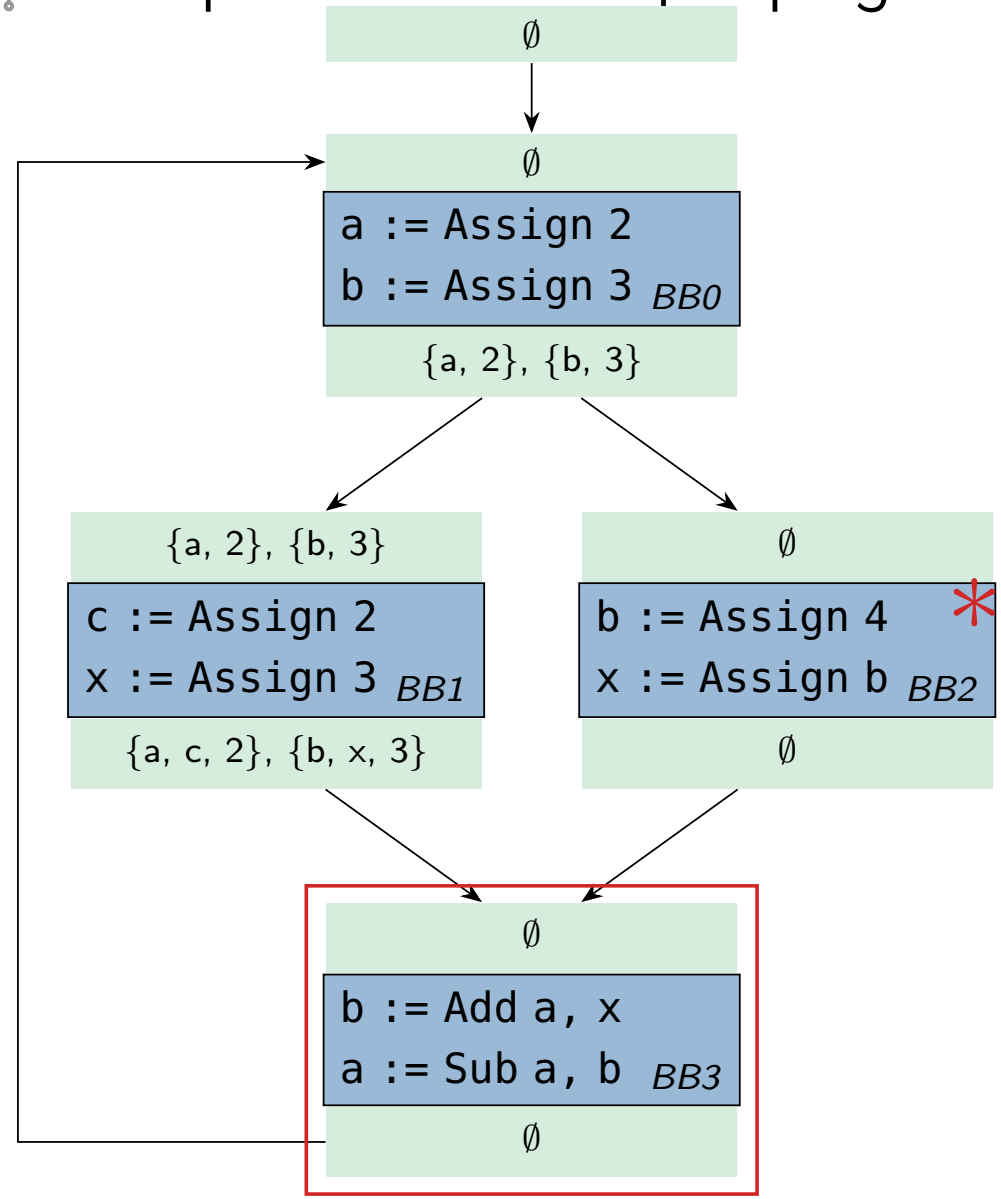
Beispiel für Wertepropagation



Worklist

```
[BB3, BB2]
```

Beispiel für Wertepropagation



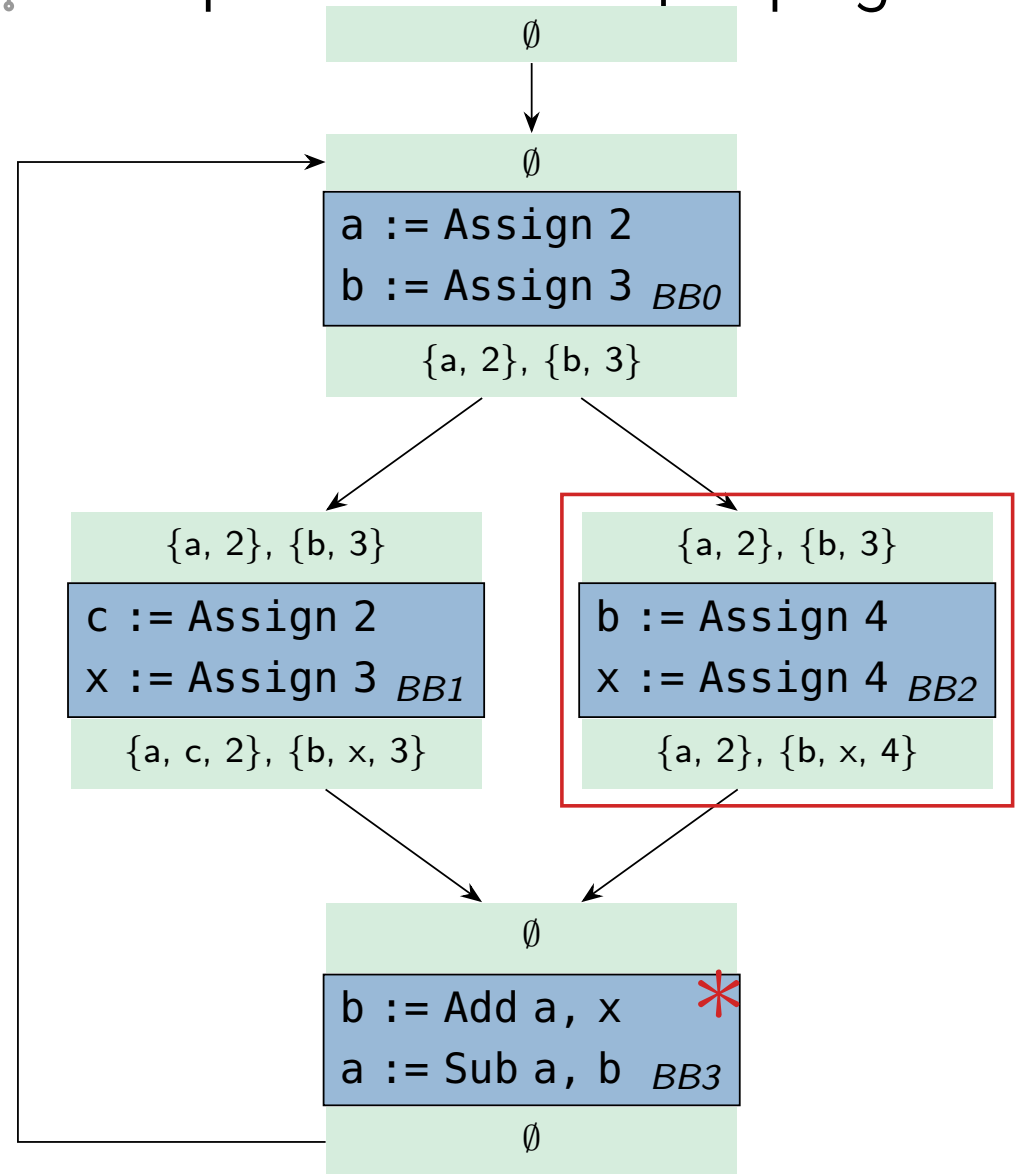
Worklist

[BB2]

Merge Logbuch

BB1	[{a, c, 2}, {b, x, 3}]
BB2	∅
BB3	∅

Beispiel für Wertepropagation



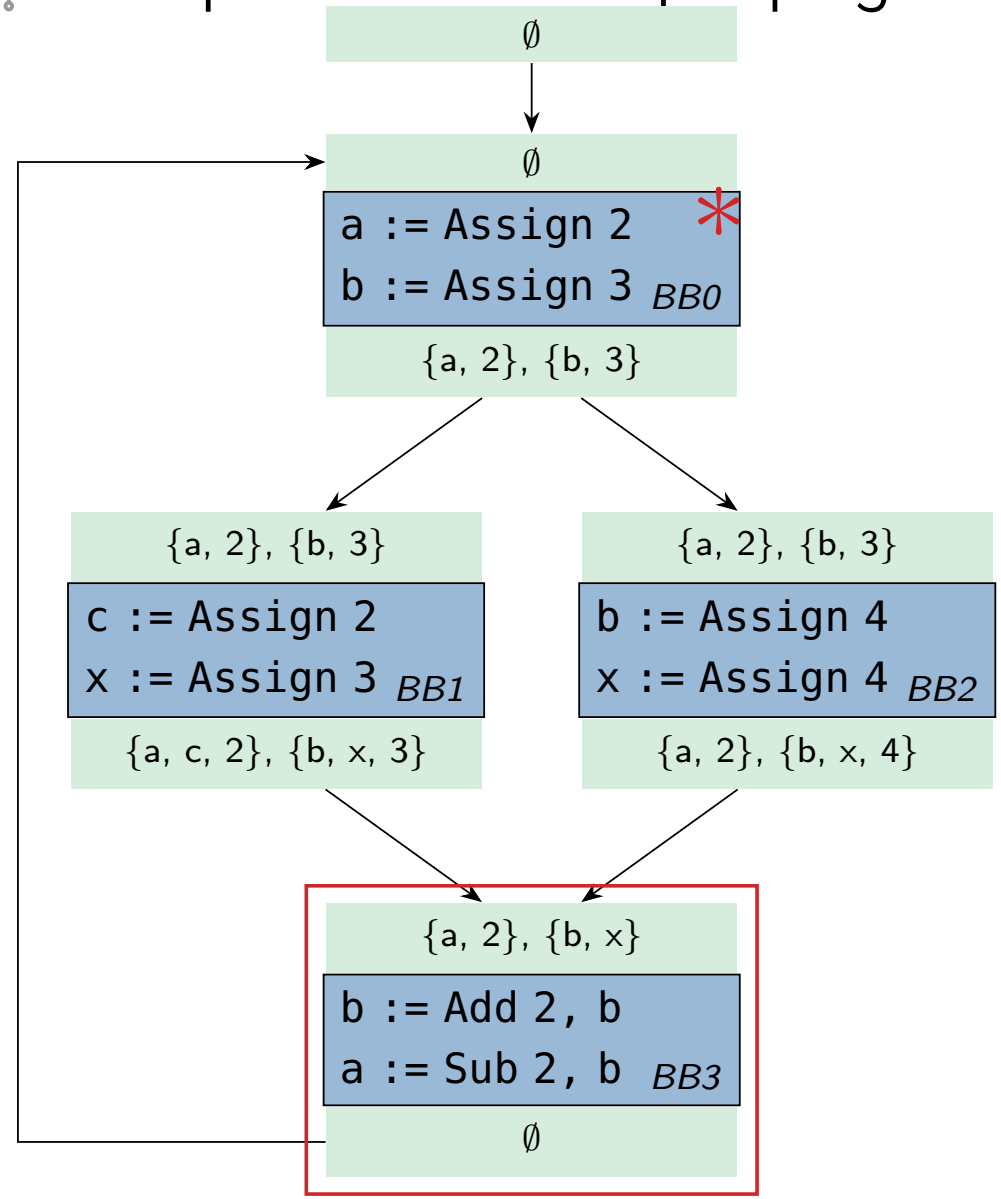
Worklist

[BB3]

Merge Logbuch

BB1	[{a, c, 2}, {b, x, 3}]
BB2	∅
BB3	∅

Beispiel für Wertepropagation



Worklist

[BB0]

Merge Logbuch

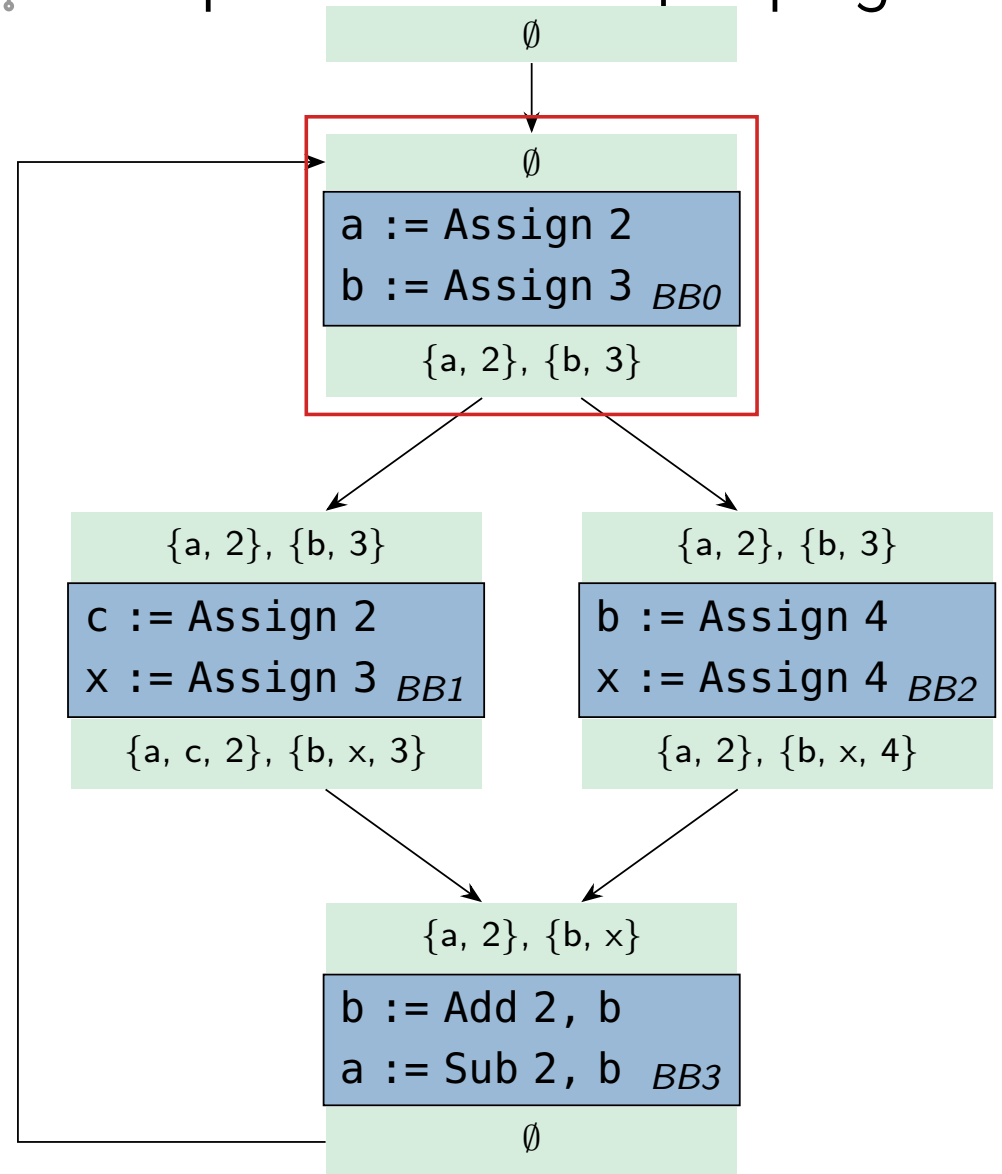
BB1	[{a, c, 2}, {b, x, 3}]
BB2	∅
BB3	∅

BB1	[{a, c, 2}, {b, x, 3}]
BB2	[{a, 2}, {b, x, 4}]
BB3	[{a, 2}, {b, x}]

Der

Merge-Code ist fummelig.

Beispiel für Wertepropagation



Worklist

[ ]

Merge Logbuch

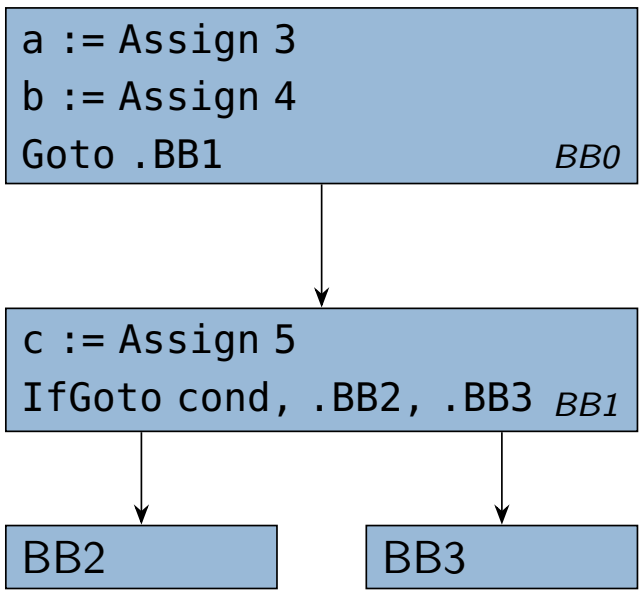
BB1	[{a, c, 2}, {b, x, 3}]
BB2	∅
BB3	∅

BB1	[{a, c, 2}, {b, x, 3}]
BB2	[{a, 2}, {b, x, 4}]
BB3	[{a, 2}, {b, x}]

Der

Merge-Code ist fummelig.

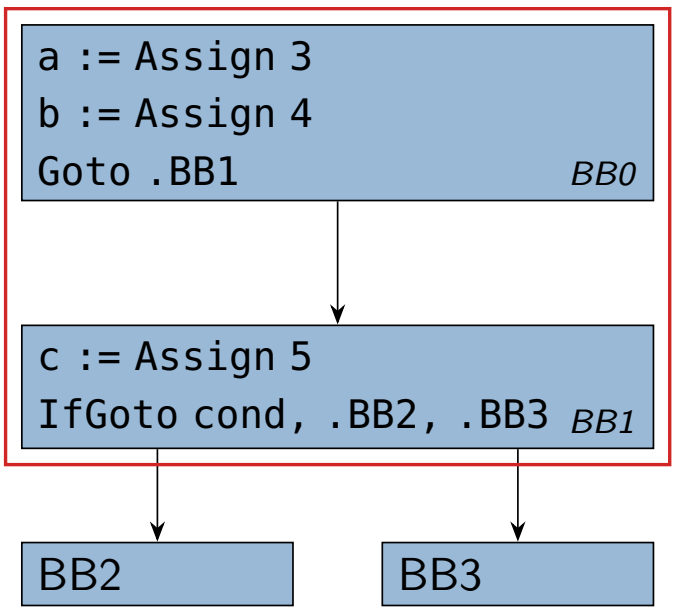
Verschmelzen von konsekutiven Basisblöcken



- **Problem**: Sprung zu BB1 ist überflüssig
  - BB0 hat genau einen Nachfolger
  - BB1 hat genau einen Vorgänger

  $\Rightarrow$ Verschmelze BB0 und BB1

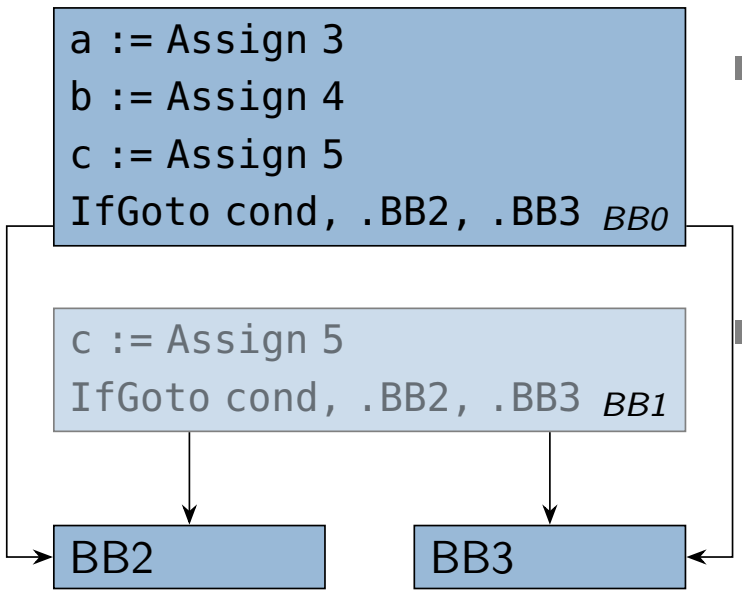
Verschmelzen von konsekutiven Basisblöcken



- **Problem**: Sprung zu BB1 ist überflüssig
  - BB0 hat genau einen Nachfolger
  - BB1 hat genau einen Vorgänger

  $\Rightarrow$ Verschmelze BB0 und BB1
- **Analyse**: Finde Block-Paar (x, y), sodass...
  - y ist Nachfolger von x
  - x hat genau einen Nachfolger
  - y hat genau einen Vorgänger

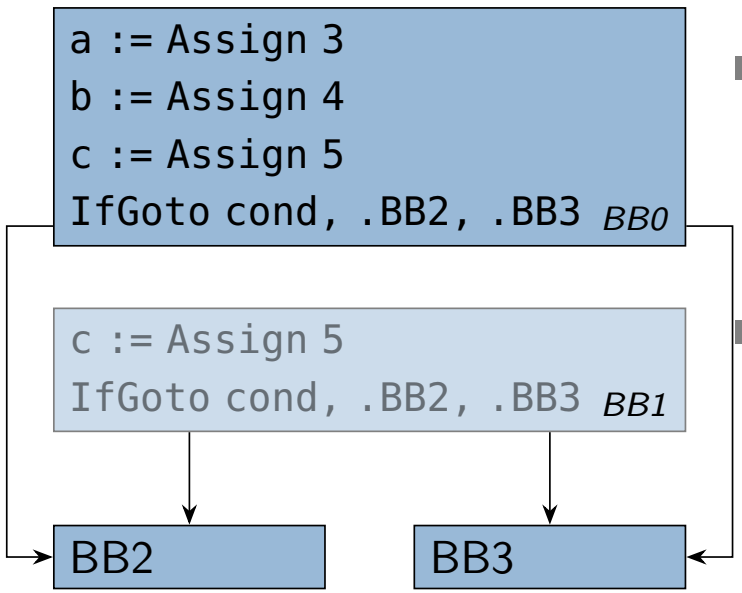
Verschmelzen von konsekutiven Basisblöcken



- **Problem**: Sprung zu BB1 ist überflüssig
  - BB0 hat genau einen Nachfolger
  - BB1 hat genau einen Vorgänger

  $\Rightarrow$ Verschmelze BB0 und BB1
- **Analyse**: Finde Block-Paar (x, y), sodass...
  - y ist Nachfolger von x
  - x hat genau einen Nachfolger
  - y hat genau einen Vorgänger
- **Transformation**: Verschmelze $x \rightarrow y$
  - Lösche `Goto` aus x
  - Hänge Instruktionen von y an x
  - Block y wird später gelöscht

Verschmelzen von konsekutiven Basisblöcken



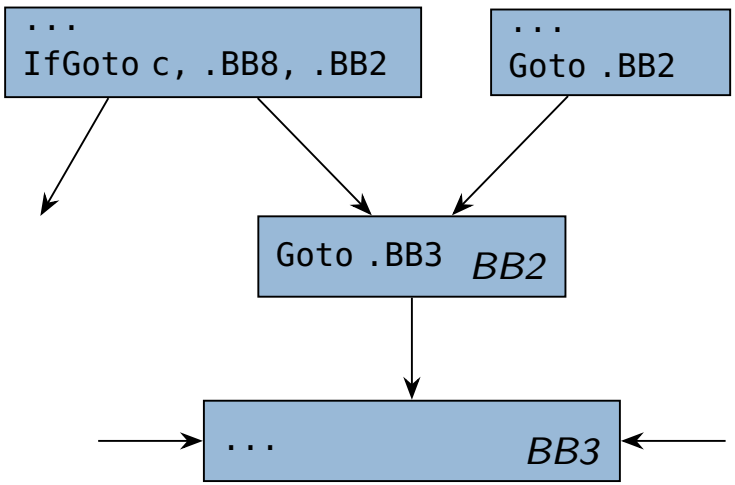
- **Problem**: Sprung zu BB1 ist überflüssig
  - BB0 hat genau einen Nachfolger
  - BB1 hat genau einen Vorgänger

  $\Rightarrow$ Verschmelze BB0 und BB1
- **Analyse**: Finde Block-Paar (x, y), sodass...
  - y ist Nachfolger von x
  - x hat genau einen Nachfolger
  - y hat genau einen Vorgänger
- **Transformation**: Verschmelze $x \rightarrow y$
  - Lösche `Goto` aus x
  - Hänge Instruktionen von y an x
  - Block y wird später gelöscht

Trennung der Belange

Das Aufräumen des toten Blocks (BB1) übernimmt eine andere Optimierung.

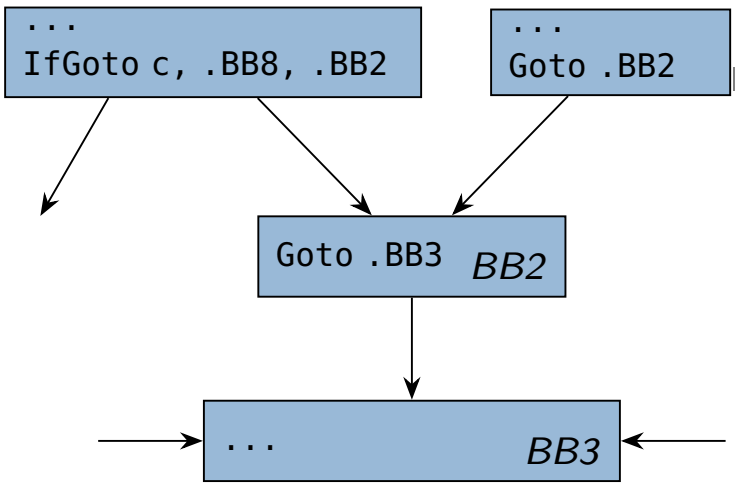
Elimination von unbedingten Doppelsprüngen



- **Problem**: BB2 ist überflüssig
  - BB2 enthält **nur** ein `Goto`
  - Jeder Sprung zu BB2 ist ein Sprung zu BB3

  ⇒ **Ziel**: Vermeide redundante Sprünge

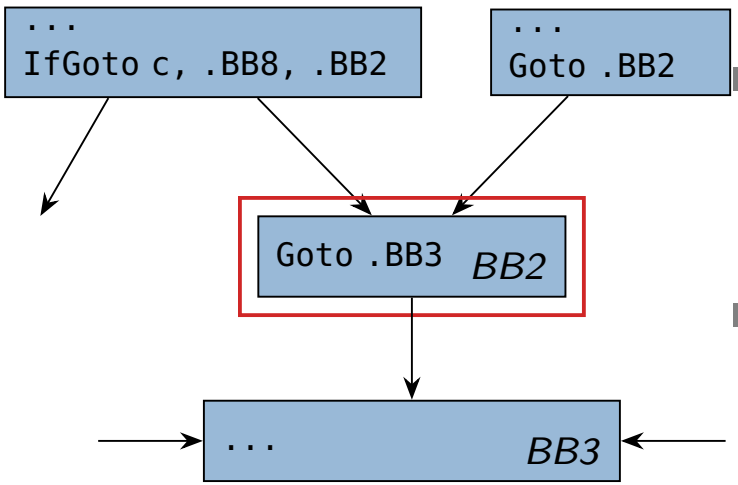
Elimination von unbedingten Doppelsprüngen



- **Problem**: BB2 ist überflüssig
  - BB2 enthält **nur** ein `Goto`
  - Jeder Sprung zu BB2 ist ein Sprung zu BB3

  ⇒ **Ziel**: Vermeide redundante Sprünge
- **Hinweis**: Verschmelzen hier unmöglich
  - BB2 hat mehrere Vorgänger
  - Verschmelzen würde BB-Bedingung verletzen

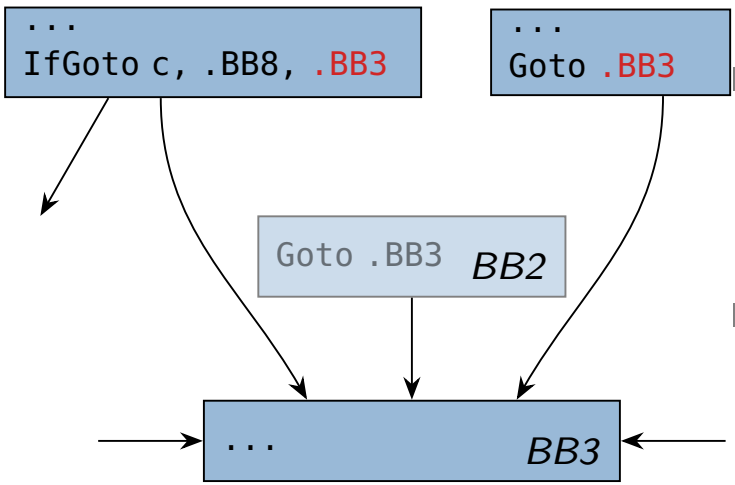
Elimination von unbedingten Doppelsprüngen



- **Problem**: BB2 ist überflüssig
  - BB2 enthält **nur** ein `Goto`
  - Jeder Sprung zu BB2 ist ein Sprung zu BB3

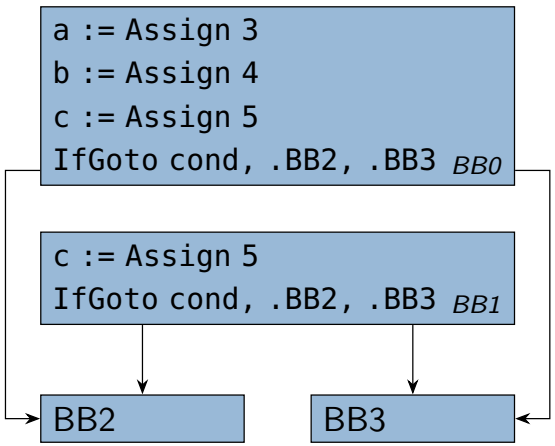
  ⇒ **Ziel**: Vermeide redundante Sprünge
- **Hinweis**: Verschmelzen hier unmöglich
  - BB2 hat mehrere Vorgänger
  - Verschmelzen würde BB-Bedingung verletzen
- **Analyse**: Finde Block x sodass…
  - x nur ein `Goto` enthält
  - x hat y als einzigen Nachfolger

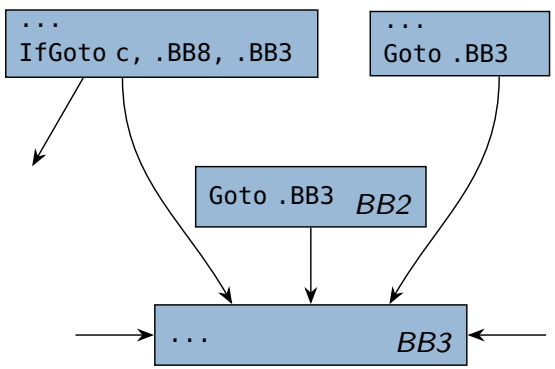
Elimination von unbedingten Doppelsprüngen



- **Problem**: BB2 ist überflüssig
  - BB2 enthält **nur** ein `Goto`
  - Jeder Sprung zu BB2 ist ein Sprung zu BB3
  - ⇒ **Ziel**: Vermeide redundante Sprünge
- **Hinweis**: Verschmelzen hier unmöglich
  - BB2 hat mehrere Vorgänger
  - Verschmelzen würde BB-Bedingung verletzen
- **Analyse**: Finde Block x sodass…
  - x nur ein `Goto` enthält
  - x hat y als einzigen Nachfolger
- **Transformation**:
  - Besuche alle Vorgänger von x
  - Ersetze jedes x.label durch y.label
  - Block x wird später gelöscht

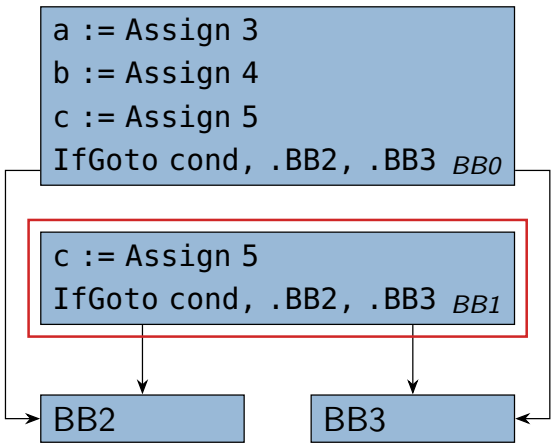
Elimination von toten Basisblöcken

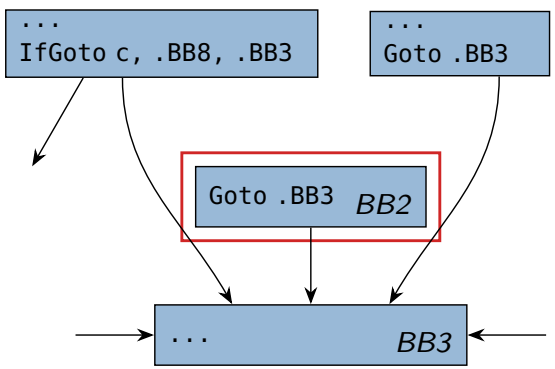




- **Problem**: Optimierungen hinterlassen Müll
  - Aufräumen von totem Code zentralisieren
  - Blöcke, die nie angesprungen werden

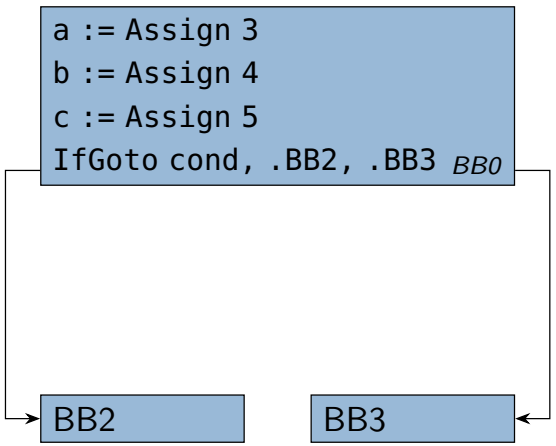
Elimination von toten Basisblöcken

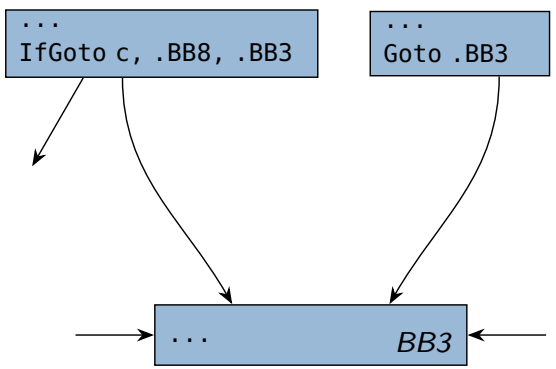




- **Problem**: Optimierungen hinterlassen Müll
  - Aufräumen von totem Code zentralisieren
  - Blöcke, die nie angesprungen werden
- **Analyse**: Finde tote Blöcke
  - Blöcke ohne Vorgänger
  - Keine Label-Adresse im Umlauf
- **Transformation**: Lösche Block
  - Entfernung aller ausgehenden Kanten
  - Block bei umgebender Funktion entfernen

Elimination von toten Basisblöcken





- **Problem**: Optimierungen hinterlassen Müll
  - Aufräumen von totem Code zentralisieren
  - Blöcke, die nie angesprungen werden
- **Analyse**: Finde tote Blöcke
  - Blöcke ohne Vorgänger
  - Keine Label-Adresse im Umlauf
- **Transformation**: Lösche Block
  - Entfernung aller ausgehenden Kanten
  - Block bei umgebender Funktion entfernen

Elimination von toten Variablen

```
return (1+2)*(3/4)*(5+6)
```

```
t0 := Add 1, 2
t1 := Div 3, 4
t2 := Mul t0, t1
t3 := Add 5, 6
t4 := Mul t2, t3
Return t4                    BB0
```

- **Problem**: Vorangegangene Optimierungen machen Variablen überflüssig
  - Konstantenfaltung ersetzt Operationen durch ihr konstantes Ergebnis
  - Durch Wertepropagation werden äquivalente Quelloperanden vereinigt

Elimination von toten Variablen

Constant-Folding!

```
t0 := Assign 3
t1 := Assign 0
t2 := Mul t0, t1
t3 := Assign 11
t4 := Mul t2, t3
Return t4                    BB0
```

- **Problem**: Vorangegangene Optimierungen machen Variablen überflüssig
  - Konstantenfaltung ersetzt Operationen durch ihr konstantes Ergebnis
  - Durch Wertepropagation werden äquivalente Quelloperanden vereinigt

Elimination von toten Variablen

Constant-Value Propagation!

```
t0 := Assign 3
t1 := Assign 0
t2 := Mul 3, 0
t3 := Assign 11
t4 := Mul t2, 11
Return t4                    BB0
```

- **Problem**: Vorangegangene Optimierungen machen Variablen überflüssig
  - Konstantenfaltung ersetzt Operationen durch ihr konstantes Ergebnis
  - Durch Wertepropagation werden äquivalente Quelloperanden vereinigt

Elimination von toten Variablen

Constant-Folding!

```
t0 := Assign 3
t1 := Assign 0
t2 := Assign 0
t3 := Assign 11
t4 := Mul t2, 11
Return t4                    BB0
```

- **Problem**: Vorangegangene Optimierungen machen Variablen überflüssig
  - Konstantenfaltung ersetzt Operationen durch ihr konstantes Ergebnis
  - Durch Wertepropagation werden äquivalente Quelloperanden vereinigt

Elimination von toten Variablen

Constant-Value Propagation!

```
t0 := Assign 3
t1 := Assign 0
t2 := Assign 0
t3 := Assign 11
t4 := Mul 0, 11
Return t4                    BB0
```

- **Problem**: Vorangegangene Optimierungen machen Variablen überflüssig
  - Konstantenfaltung ersetzt Operationen durch ihr konstantes Ergebnis
  - Durch Wertepropagation werden äquivalente Quelloperanden vereinigt

Elimination von toten Variablen

Constant-Folding!

```
t0 := Assign 3
t1 := Assign 0
t2 := Assign 0
t3 := Assign 11
t4 := Assign 0
Return t4                    BB0
```

- **Problem**: Vorangegangene Optimierungen machen Variablen überflüssig
  - Konstantenfaltung ersetzt Operationen durch ihr konstantes Ergebnis
  - Durch Wertepropagation werden äquivalente Quelloperanden vereinigt

Elimination von toten Variablen

Constant-Value Propagation!

```
t0 := Assign 3
t1 := Assign 0
t2 := Assign 0
t3 := Assign 11
t4 := Assign 0
Return 0                    BB0
```

- **Problem**: Vorangegangene Optimierungen machen Variablen überflüssig
  - Konstantenfaltung ersetzt Operationen durch ihr konstantes Ergebnis
  - Durch Wertepropagation werden äquivalente Quelloperanden vereinigt

Elimination von toten Variablen

```
t0 := Assign 3      ← dead var t0
t1 := Assign 0      ← dead var t1
t2 := Assign 0      ← dead var t2
t3 := Assign 11     ← dead var t3
t4 := Assign 0      ← dead var t4
Return 0                    BB0
```

- **Problem**: Vorangegangene Optimierungen machen Variablen überflüssig
  - Konstantenfaltung ersetzt Operationen durch ihr konstantes Ergebnis
  - Durch Wertepropagation werden äquivalente Quelloperanden vereinigt
- **Analyse**: Finde alle lokalen Variablen, die nie gelesen werden
  - Vorsicht: Zeiger können Variablen indirekt lesen (Alias-Problem)
  - Heuristik: Markiere alle Variablen, deren Adresse berechnet wird, als „gelesen“

Elimination von toten Variablen

```
t0 := Assign 3
t1 := Assign 0
t2 := Assign 0
t3 := Assign 11
t4 := Assign 0
Return 0                    BB0
```

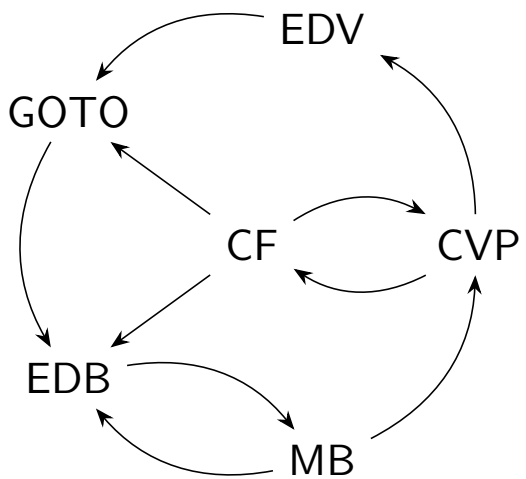
- **Problem**: Vorangegangene Optimierungen machen Variablen überflüssig
  - Konstantenfaltung ersetzt Operationen durch ihr konstantes Ergebnis
  - Durch Wertepropagation werden äquivalente Quelloperanden vereinigt
- **Analyse**: Finde alle lokalen Variablen, die nie gelesen werden
  - Vorsicht: Zeiger können Variablen indirekt lesen (Alias-Problem)
  - Heuristik: Markiere alle Variablen, deren Adresse berechnet wird, als „gelesen“
- **Transformation**: Entferne niemals gelesene lokale Variable x
  - Lösche Instruktionen, die x schreiben und kein `Call` sind (Seiteneffekte!)
  - Lösche x aus der umgebenden Funktion

Optimieren einer Funktion

```
ConstantFolding(Function);
ValuePropagation(Function);
MergeBasicBlocks(Function);
```

```
EliminateRedundantGotos(Function);
EliminateDeadBlocks(Function);
EliminateDeadVariables(Function);
```

- Wir haben sechs Optimierungsverfahren für Zwischencode kennengelernt
  - Optimierungen ermöglichen sich gegenseitig Optimierungen
  - In welcher Reihenfolge sollen wir die Optimierungen anwenden?

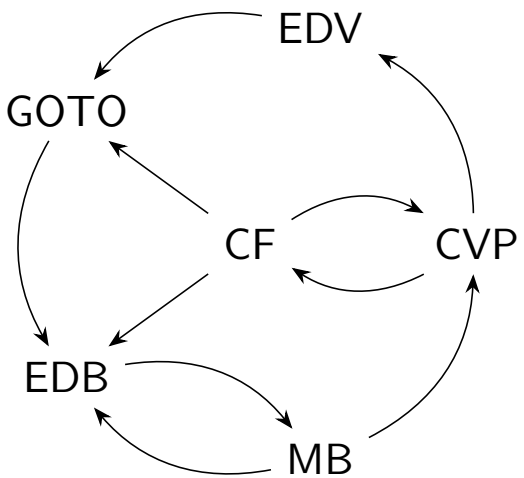


Optimieren einer Funktion

```
bool ConstantFolding(Function);
bool ValuePropagation(Function);
bool MergeBasicBlocks(Function);
```

```
bool EliminateRedundantGotos(Function);
bool EliminateDeadBlocks(Function);
bool EliminateDeadVariables(Function);
```

- Wir haben sechs Optimierungsverfahren für Zwischencode kennengelernt
  - Optimierungen ermöglichen sich gegenseitig Optimierungen
  - In welcher Reihenfolge sollen wir die Optimierungen anwenden?
- Optimierung als Fixpunktiteration
  - Jedes Verfahren gibt zurück, ob die Funktion verändert wurde
  - Bei Veränderung führen wir alle anderen Optimierungen **nochmals aus**




```
def optimize(func):
  changed = True
  while changed:
    changed = False

    for optimizer in optimizers:
      ret = optimizer(func)
      changed = changed or ret
```

Meilensteine der optimierenden Übersetzer

Technische Universität Braunschweig

Optimierende Übersetzer sind seit vielen Jahrzehnten ein aktives Forschungsebiet. Hier einige Arbeiten, die **ich** für Meilensteine halte:

1973	Datenflussanalyse als Fixpunktiteration	Kildall
1977	Abstrakte Interpretation	Cousot und Cousot
1979	Portable Peephole Optimizer	Fraser
1989	Single-Static Assignment wird effizient	Cytron
1993	Verschmelzen von Schleifen	Kelly and Pugh
1996	Effiziente Interprozedurale Alias Analyse	Steensgaard
2004	Low-Level Virtual Machine (LLVM)	Lattner
2010	Polyhedral Loop Optimization in GCC	Trifunovic

Zusammenfassung

- Optimierungen nur von nicht-funktionalen Programmeigenschaften
  - Aus Sicht der Sprache bleibt das beobachtbare Verhalten gleich.
  - Verschiedene Optimierungsziele, wobei es meist um Ausführungszeit geht
- Optimierungen sind immer zweigeteilte Übersetzungsschritte.
  - Analyse des Programms berechnet Informationen, die wir brauchen, um zu entscheiden, **ob und wie** eine Optimierung angewendet wird.
  - Transformation der Zwischenrepräsentation des Programms setzt die Optimierung durch.
- Auf IR-Ebene reduzieren wir die Anzahl der Instruktionen und Sprünge
  - Konstantenfaltung ersetzt konstante Berechnungen durch ihr Ergebnis
  - Wertepropagation ersetzt Operanden durch äquivalente Werte/Konstanten
  - Verschmelzen von streng konsekutiv ausgeführte Blöcke
  - Redundante Doppelsprünge werden aufgelöst
  - Tote Blöcke und tote Variablen werden entfernt