

Kapitel 7

Codegenerierung, Registerzuteilung und Optimierung

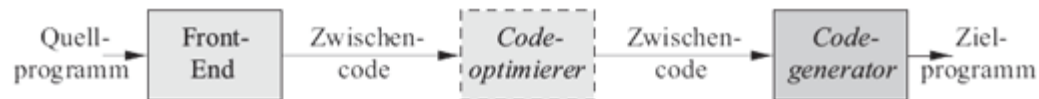


Abbildung 8.1: Die Position des Codegenerators

Adressen im Zielcode

- Statisch festgelegter Bereich Code
- Statisch festgelegter Bereich Static für globale Konstanten und andere vom Compiler erzeugte Daten
- Dynamisch verwalteter Heap
- Dynamisch verwalteter Stack für Aktivierungseinträge bei Aufrufen von Prozeduren

Speicherverwaltung im Zielcode

*(ganz kurze Exkursion;
komplette Details in Kapitel 7)*

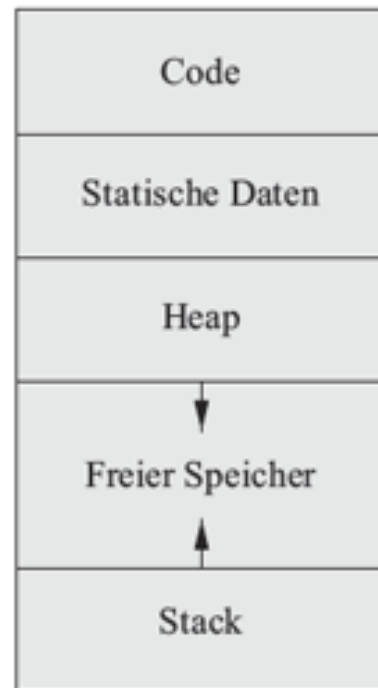


Abbildung 7.1: Gewöhnliche Unterteilung des Laufzeitspeichers in Code- und Datenbereiche

Speicherverwaltung II

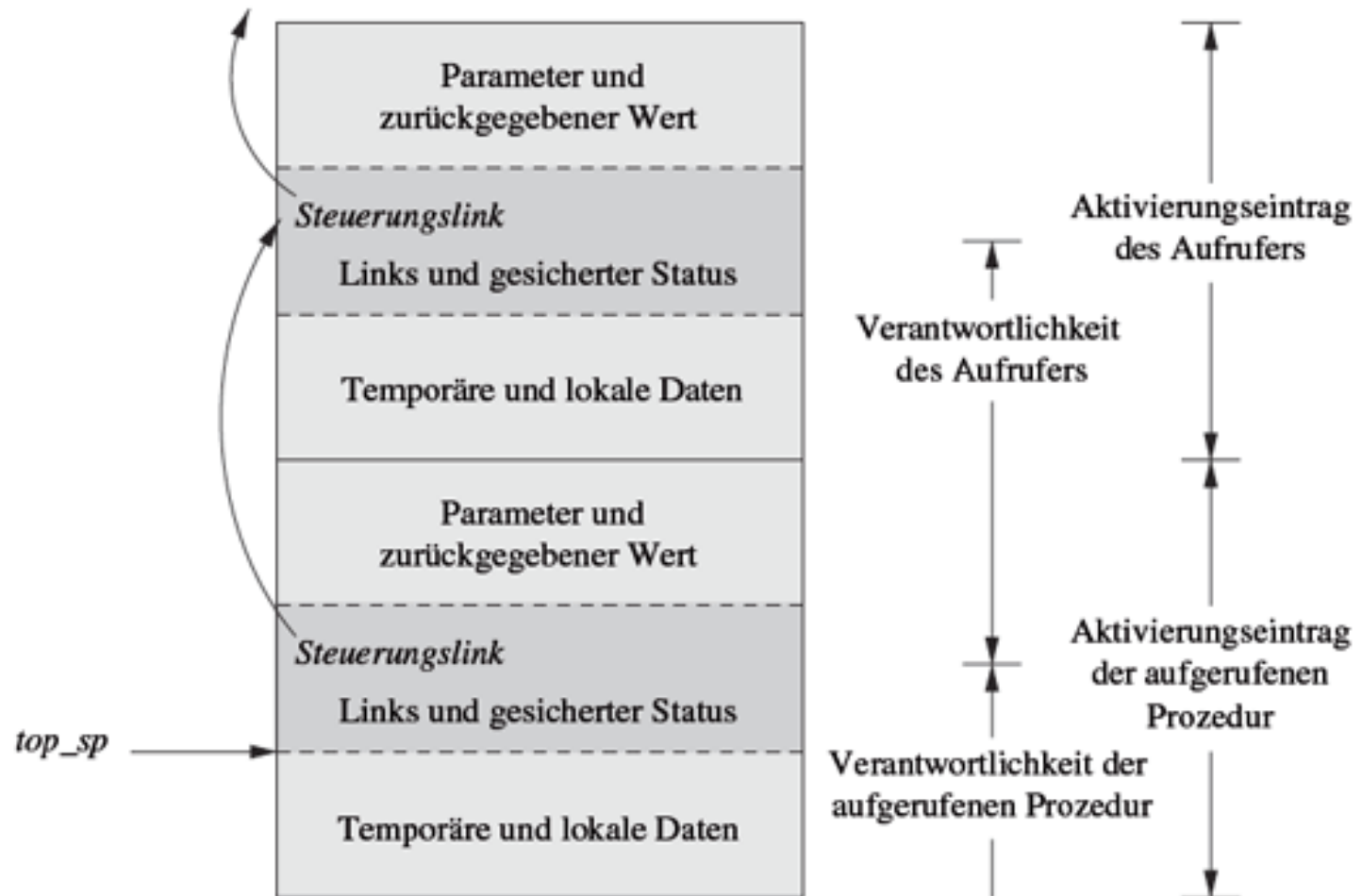


Abbildung 7.7: Aufgabenteilung zwischen Aufrufer und aufgerufener Prozedur

Wiederholung:

Format des **Zwischencodes**

- Befehle:
 - $x = y \text{ op } z, \quad x = \text{op } y, \quad x = y$
 - goto L
 - if x goto L und ifFalse x goto L
 - if x relop y goto L
 - $x = y[i] \quad \text{und} \quad x[i] = y$
 - $x = \&y, \quad x = *y, \quad x^* = y$
- Adressen:
 - Namen, Konstanten, temporäre Variablen

Wiederholung: Prozessorarchitekturen

- RISC:

- Viele Register und nur eine Klasse von Registern
- (Arithmetische) Operationen nur zwischen Registern
- Drei-Adress Instruktionen, alle gleiche Länge
- Load und store Instruktionen der Form `Memory[Register+Konstante]`
- Ein Ergebnis/ Effekt pro Instruktion

- CISC:

- Wenige Register und unterschiedliche Klassen von Registern
- (Arithmetische) Operationen zwischen Registern und Speicher
- Zwei-Adress Instruktionen, variable Länge
- Viele Addressierungsmodi
- Instruktionen mit Seiteneffekten

Drei-Adress RISC Maschine:

Zielcode (Befehle)

- **LD** dst, addr
 - z.B. LD r,x oder LD r1,r2
- **ST** x,r
 - Speichere den Wert in r an der Adresse x
- **OP** dst, src1, src2
 - z.B. SUB r1,r2,r3: $r1 = r2 - r3$
- **BR** L
- **Bcond** r, L
 - z.B. BLTZ r, L: Sprung zu L wenn $r < 0$

Drei-Adress RISC Maschine:

Zielcode (L- und R-Werte)

- Symbole wie x und y haben unterschiedliche Bedeutungen je nach Zuweisungsseite
 - Rechte Seite einer Zuweisung (R-Wert): Wert

Wir schreiben **contents**(x) im Pseudocode und **ind** (für Indirektion) in Baumdarstellungen des Zielcodes
 - Linke Seite einer Zuweisung (L-Wert): Adresse
 - z.B: $x=y$ (nimm Wert an Stelle y und schreibe ihn in den Speicher an Stelle x)

Drei-Adress RISC Maschine:

Zielcode (Adressen)

- Variablennamen: x, y, a, b ...
- **Register:** r0, r1, r2, sp ...
- **Konstanten:** #const (LD r1, #100)
- a(r0):
 - Berchnet Adresse: $a + r0$
- **Indexed:** const(reg)
 - z.B LD r1, 100(r2) mit 100 und r2 als Wert
d.h. $r1 = \text{contents}(100 + \text{contents}(r2))$
- **Indirect:** *r,
 - z.B. contents(r)
- **Indirect indexed:** *const(r)
 - z.B. *100(r): $\text{contents}(\text{contents}(100 + \text{contents}(r)))$

Naive Codeerzeugung

- $x = y + z$

```
LD R0, y      // R0 = y      (y in Register R0 laden)
ADD R0, R0, z  // R0 = R0 + z (z zu R0 addieren)
ST x, R0       // x = R0      (R0 in x speichern)
```

- $a = b + c; d = a + e$

$a = a + 1$

```
LD R0, b      // R0 = b
ADD R0, R0, c  // R0 = R0 + c
ST a, R0       // a = R0
LD R0, a      // R0 = a      unnötig
ADD R0, R0, e  // R0 = R0 + e
ST d, R0       // d = R0
```

```
LD R0, a      // R0 = a
ADD R0, R0, #1 // R0 = R0 + 1
ST a, R0       // a = R0
```

Codeerzeugung: Beispiele

- $x = y - z$

- $b = a[i]$

Codeerzeugung: Beispiele

- $x = y - z$

LD R1, y

LD R2, z

SUB R1, R1, R2

ST x, R1

- $b = a[i]$

LD R1, i

MUL R1, R1, 8

LD R2, a(R1)

ST b, R2

Codeerzeugung: Beispiele

- $a[i]=c$
- if $x < y$ goto L

Codeerzeugung: Beispiele

- $a[i]=c$

LD R1, c

LD R2, i

MUL R2, R2, 8

ST a(R2), R1

- if $x < y$ goto L

LD R1, x

LD R2, y

SUB R1, R1, R2

BLTZ R1, L

Optimale Codeerzeugung

$t = a + b$	$t = a + b$
$t = t * c$	$t = t + c$
$t = t / d$	$t = t / d$
a	b

Abbildung 8.2: Zwei Drei-Adress-Codefolgen



	
L R1, a	L R0, a
A R1, b	A R0, b
M R0, c	A R0, c
D R0, d	SRDA R0, 32
ST R1, t	D R0, d
a	b

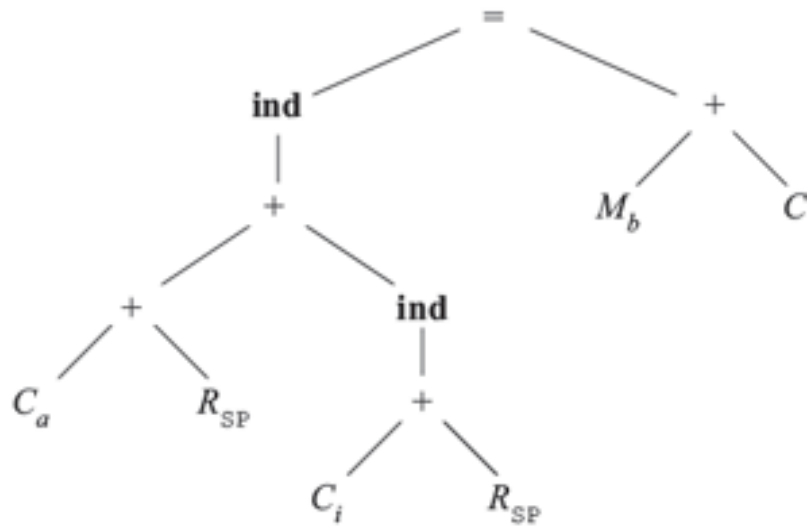
Abbildung 8.3: Optimale Maschinencodfolgen

Instruction Selection (Befehlsauswahl)

- Generierung von Maschinencode aus einem IR-Baum (oder Zwischencode)
- Gefolgt von Registerzuweisung
- Es gibt nicht immer eine 1:1 Beziehung zwischen IR-Knoten und Instruktionen

Zwischencode nach Zielcode

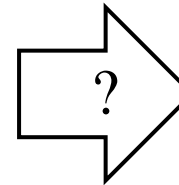
- Zwischencode (nach Adresszuweisung) in Baumdarstellung:



C_a , C_i : Offsets im Stackframe
 M_b : Adresse der globalen Variable b

ind: Indirektion: Argument ist Speicheradresse

Abbildung 8.19: Zwischencodebaum für $a[i] = b + 1$



```
LD R0, #a
ADD R0, R0, SP
ADD R0, R0, i(SP)
LD R1, b
INC R1
ST *R0, R1
```

Siehe Kapitel 8.3 (Adressen im Zielcode)

Baumersetzungsverfahren

“Kontextfreie Grammatik” auf Bäumen:



*Semantische
Aktion*

Anmerkung: i, j Parameter der Regel

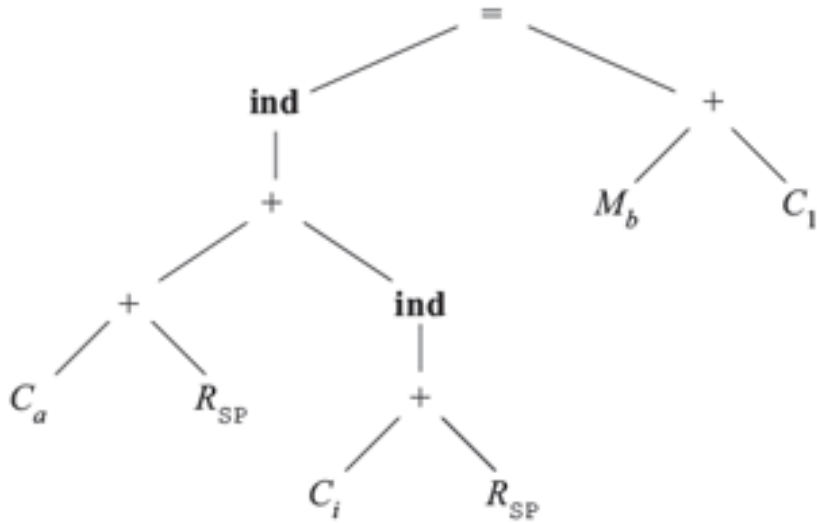
Generierung des Maschinencodes durch LR-Parsing

Komplette Regeln

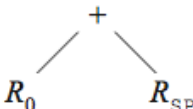
1)	$R_i \leftarrow C_a$	$\{ \text{LD } Ri, \#a \}$
2)	$R_i \leftarrow M_x$	$\{ \text{LD } Ri, x \}$
3)	$ \begin{array}{c} M \leftarrow = \\ \swarrow \quad \searrow \\ M_x \quad R_i \end{array} $	$\{ \text{ST } x, Ri \}$
4)	$ \begin{array}{c} M \leftarrow = \\ \swarrow \quad \searrow \\ \text{ind} \quad R_j \\ \\ R_i \end{array} $	$\{ \text{ST } *Ri, Rj \}$
5)	$ \begin{array}{c} R_i \leftarrow \text{ind} \\ \\ + \\ \swarrow \quad \searrow \\ C_a \quad R_j \end{array} $	$\{ \text{LD } Ri, a(Rj) \}$
6)	$ \begin{array}{c} R_i \leftarrow + \\ \swarrow \quad \searrow \\ R_i \quad \text{ind} \\ \quad \\ \quad + \\ \quad \swarrow \quad \searrow \\ \quad C_a \quad R_j \end{array} $	$\{ \text{ADD } Ri, Ri, a(Rj) \}$
7)	$ \begin{array}{c} R_i \leftarrow + \\ \swarrow \quad \searrow \\ R_i \quad R_j \end{array} $	$\{ \text{ADD } Ri, Ri, Rj \}$
8)	$ \begin{array}{c} R_i \leftarrow + \\ \swarrow \quad \searrow \\ R_i \quad C_1 \end{array} $	$\{ \text{INC } Ri \}$

Abbildung 8.20: Baumersetzungsregeln für einige Zielmaschinenbefehle

Beispiel



1) $R_0 \leftarrow C_a$ { LD $R0, \#a$ }

7) $R_0 \leftarrow$  { ADD $R0, R0, SP$ }

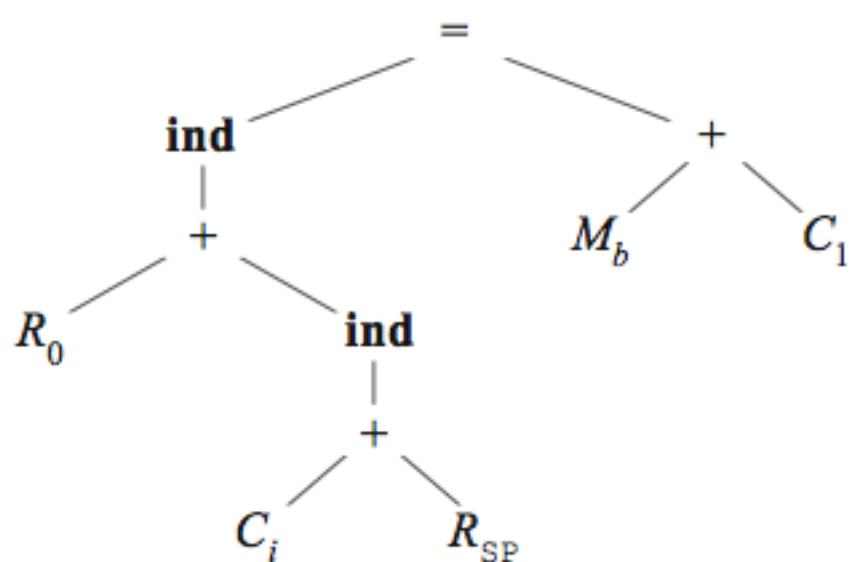
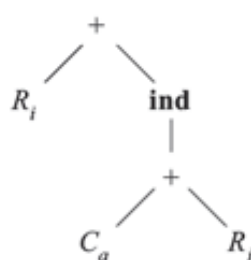
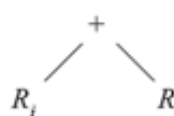
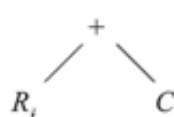
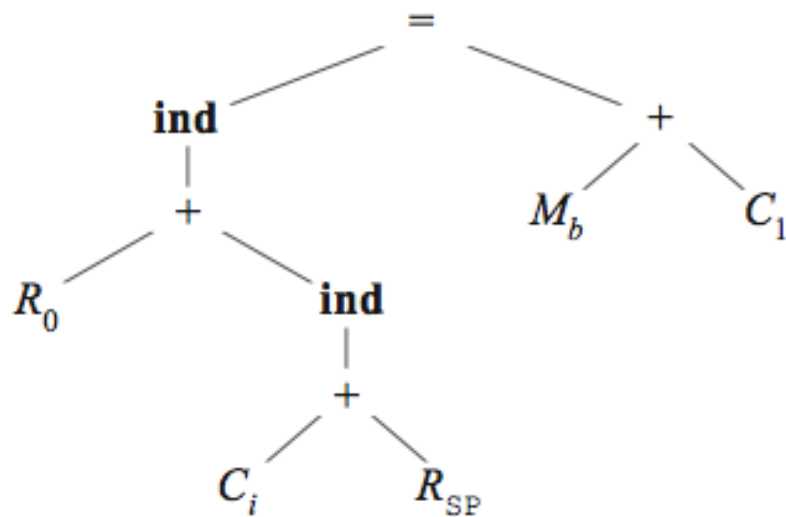
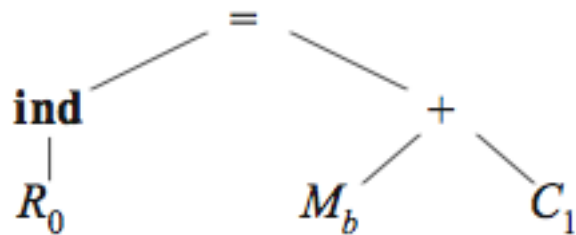
1)	$R_i \leftarrow C_a$	{ LD $Ri, \#a$ }
		
6)	$R_i \leftarrow$ 	{ ADD $Ri, Ri, a(Rj)$ }
7)	$R_i \leftarrow$ 	{ ADD Ri, Ri, Rj }
8)	$R_i \leftarrow$ 	{ INC Ri }

Abbildung 8.20: Baumersetzungsregeln für einige Zielmaschinenbefehle

Beispiel



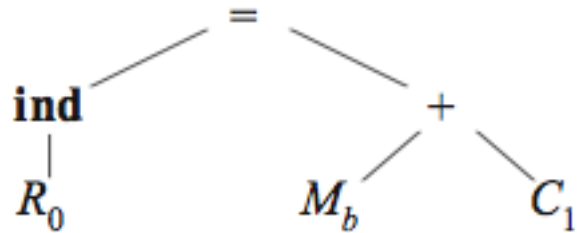
ADD R0, R0, i(SP)



1)	$R_i \leftarrow C_a$	{ LD $R_i, \#a$ }
2)	$R_i \leftarrow M_x$	{ LD R_i, x }
3)	$M \leftarrow \begin{array}{c} = \\ \swarrow \quad \searrow \\ M_x \quad R_i \end{array}$	{ ST x, R_i }
4)	$M \leftarrow \begin{array}{c} = \\ \swarrow \quad \searrow \\ \text{ind} \quad R_j \\ \\ R_i \end{array}$	{ ST $*R_i, R_j$ }
5)	$R_i \leftarrow \begin{array}{c} \text{ind} \\ \\ + \\ \swarrow \quad \searrow \\ C_a \quad R_j \end{array}$	{ LD $R_i, a(R_j)$ }
6)	$R_i \leftarrow \begin{array}{c} + \\ \swarrow \quad \searrow \\ R_i \quad \text{ind} \\ \quad \\ \quad + \\ \quad \swarrow \quad \searrow \\ \quad C_a \quad R_j \end{array}$	{ ADD $R_i, R_i, a(R_j)$ }
7)	$R_i \leftarrow \begin{array}{c} + \\ \swarrow \quad \searrow \\ R_i \quad R_j \end{array}$	{ ADD R_i, R_i, R_j }
8)	$R_i \leftarrow \begin{array}{c} + \\ \swarrow \quad \searrow \\ R_i \quad C_1 \end{array}$	{ INC R_i }

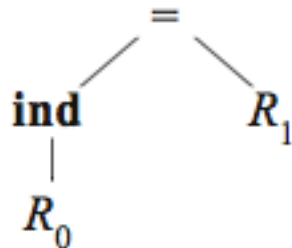
Abbildung 8.20: Baumersetzungsregeln für einige Zielmaschinenbefehle

Beispiel



Regel 2 LD R0, #a
 ADD R0, R0, SP
 ADD R0, R0, i(SP)
 LD R1, b

Regel 8 (INC R1)



Regel 4: INC R1
 ST *R0, R1

1)	$R_i \leftarrow C_a$	{ LD Ri, #a }
2)	$R_i \leftarrow M_x$	{ LD Ri, x }
3)	$M \leftarrow \begin{array}{c} = \\ \swarrow \quad \searrow \\ M_x \quad R_i \end{array}$	{ ST x, Ri }
4)	$M \leftarrow \begin{array}{c} = \\ \swarrow \quad \searrow \\ \text{ind} \quad R_j \\ \\ R_i \end{array}$	{ ST *Ri, Rj }
5)	$R_i \leftarrow \begin{array}{c} \text{ind} \\ \\ + \\ \swarrow \quad \searrow \\ C_a \quad R_j \end{array}$	{ LD Ri, a(Rj) }
6)	$R_i \leftarrow \begin{array}{c} + \\ \swarrow \quad \searrow \\ R_i \quad \text{ind} \\ \quad \quad \\ \quad \quad + \\ \quad \quad \swarrow \quad \searrow \\ \quad \quad C_a \quad R_j \end{array}$	{ ADD Ri, Ri, a(Rj) }
7)	$R_i \leftarrow \begin{array}{c} + \\ \swarrow \quad \searrow \\ R_i \quad R_j \end{array}$	{ ADD Ri, Ri, Rj }
8)	$R_i \leftarrow \begin{array}{c} + \\ \swarrow \quad \searrow \\ R_i \quad C_1 \end{array}$	{ INC Ri }

Abbildung 8.20: Baumersetzungsregeln für einige Zielmaschinenbefehle

8.9.3 Mustererkennung auf Bäumen mit LR-Parsing

andere Ansätze in 8.9.5 oder in Appel

- Ein (Zwischencode-) Baum lässt sich als String darstellen:
 - zum Beispiel Präfixdarstellung

$$= \text{ind} + + C_a R_{SP} \text{ind} + C_i R_{SP} + M_b C_1$$

dieser String steht für den Baum:

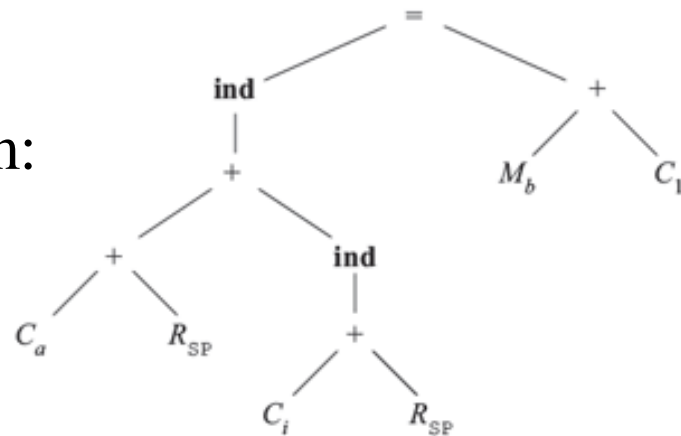
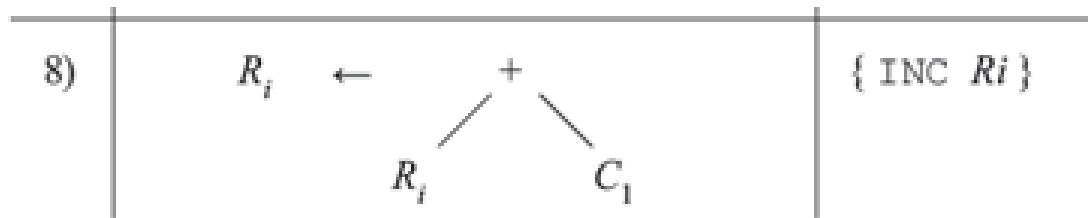


Abbildung 8.19: Zwischencodebaum für $a[i] = b + 1$

8.9.3 Mustererkennung auf Bäumen mit LR-Parsing

- Baumersetzungsregeln lassen sich als normale Grammatikregeln auf Strings darstellen

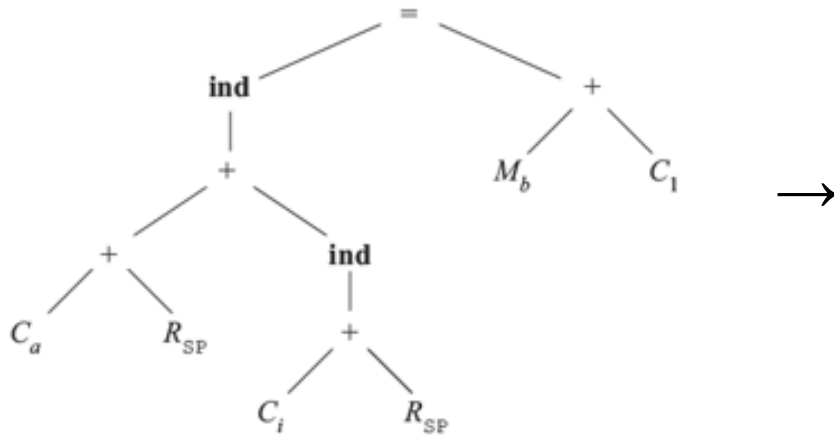


$$R_i \rightarrow + R_i C_1 \quad \{ \text{INC } Ri \}$$

Mustererkennung auf Bäumen mit LR-Parsing

andere Ansätze in 8.9.5 oder in Appel

- Baum \rightarrow String der Präfixdarstellung



$= \text{ind} + + C_a R_{SP} \text{ind} + C_i R_{SP} + M_b C_l$

Zwischencodebaum für $a[i] = b + 1$

1)	$R_i \leftarrow C_a$	{ LD Ri, #a }
2)	$R_i \leftarrow M_x$	{ LD Ri, x }
3)	$M \leftarrow \begin{array}{c} = \\ \swarrow \quad \searrow \\ M_x \quad R_i \end{array}$	{ ST x, Ri }
4)	$M \leftarrow \begin{array}{c} = \\ \swarrow \quad \searrow \\ \text{ind} \quad R_j \\ \\ R_i \end{array}$	{ ST *Ri, Rj }
5)	$R_i \leftarrow \begin{array}{c} \text{ind} \\ \\ + \\ \swarrow \quad \searrow \\ C_a \quad R_j \end{array}$	{ LD Ri, a(Rj) }
6)	$R_i \leftarrow \begin{array}{c} + \\ \swarrow \quad \searrow \\ R_j \quad \text{ind} \\ \quad \quad \\ \quad \quad + \end{array}$	{ ADD Ri, Ri, a(Rj) }



- 1) $R_i \rightarrow c_a$ { LD Ri, #a }
- 2) $R_i \rightarrow M_x$ { LD Ri, x }
- 3) $M \rightarrow = M_x R_i$ { ST x, Ri }
- 4) $M \rightarrow = \text{ind } R_i R_j$ { ST *Ri, Rj }
- 5) $R_i \rightarrow \text{ind} + c_a R_j$ { LD Ri, a(Rj) }
- 6) $R_i \rightarrow + R_i \text{ind} + c_a R_j$ { ADD Ri, Ri, a(Rj) }
- 7) $R_i \rightarrow + R_i R_j$ { ADD Ri, Ri, Rj }
- 8) $R_i \rightarrow + R_i c_1$ { INC Ri }
- 9) $R \rightarrow \text{sp}$
- 10) $M \rightarrow m$

Abbildung 8.21: Aus Abbildung 8.20 konstruiertes syntaxgesteuertes Übersetzungsverfahren

Optimale Kachelung

- *Naive Kachelung:*
 - Angrenzende Kacheln können nicht in eine Kachel mit geringern Kosten kombiniert werden
- *Optimale Kachelung:*
 - Minimum der Summe alle Kachelkosten
- CISC Prozessoren:
 - Große Kacheln, variable Instruktionskosten
 - Optimale Kachelung kann deutlich besser als naive sein.
- RISC Prozessoren:
 - Kleine Kacheln, uniforme Kosten
 - Optimal und naiv haben kaum einen Unterschied

Ein naiver Kachelungsalgorithmus

- “Maximal Munch” (Top-Down)
 - Finde die größte Kachel die passt (deckt die meisten Knoten ab) .
 - Benutze diese um den Teil des Baumes abzudecken. Dies hinterlässt i.d.R. Teilbäume.
 - Für alle Teilbäume rekursiv wiederholen
 - Willkürliche Auswahl wenn mehr als eine Kachelung möglich.
- Bedingung für Korrektheit?

Ein naiver Kachelungsalgorithmus

- “Maximal Munch” (Top-Down)
 - Finde die größte Kachel die passt (deckt die meisten Knoten ab) .
 - Benutze diese um den Teil des Baumes abzudecken. Dies hinterlässt i.d.R. Teilbäume.
 - Für alle Teilbäume rekursiv wiederholen
 - Willkürliche Auswahl wenn mehr als eine Kachelung möglich.
- Bedingung für Korrektheit:
 - Die Instruktionen werden in umgekehrter Reihenfolge erzeugt
 - Es gibt für jeden Knoten mindestens eine Kachel (also 1:1). D.h. keine Knoten bleiben „übrig“

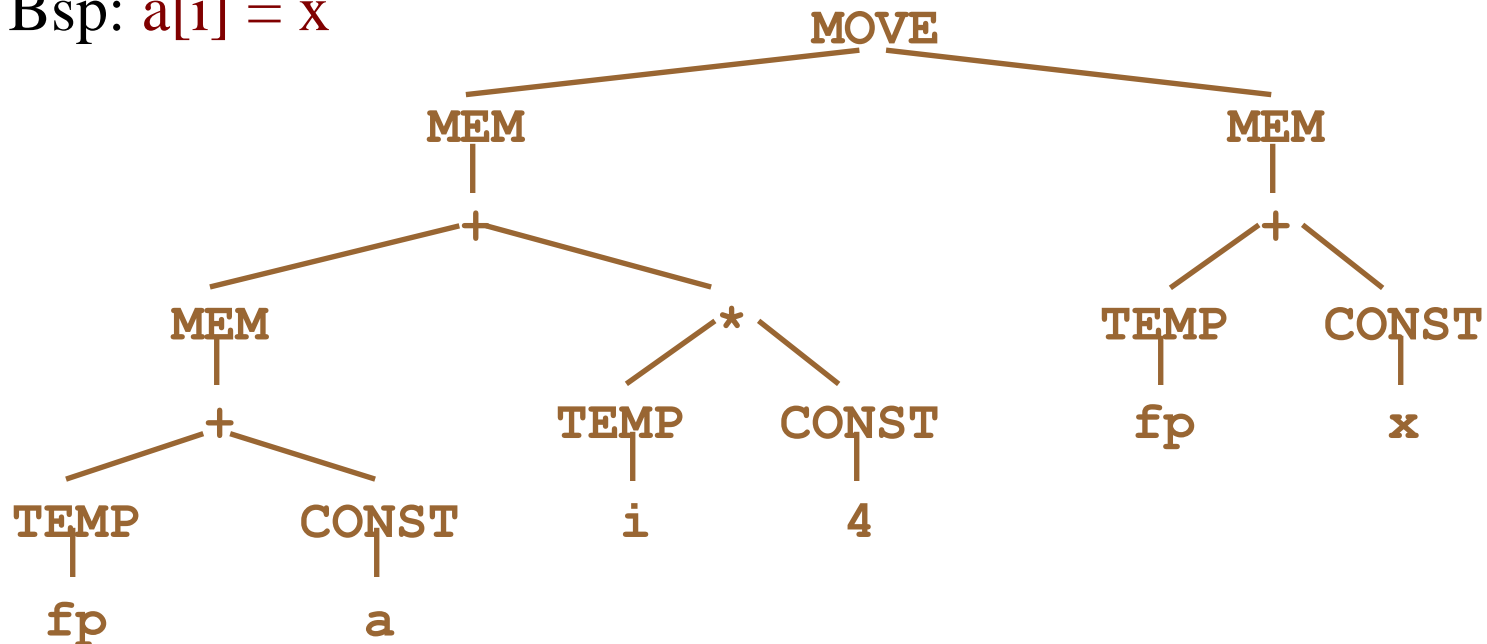
Notationsunterschiede IR von Drachenbuch und Appel

MEM statt ind

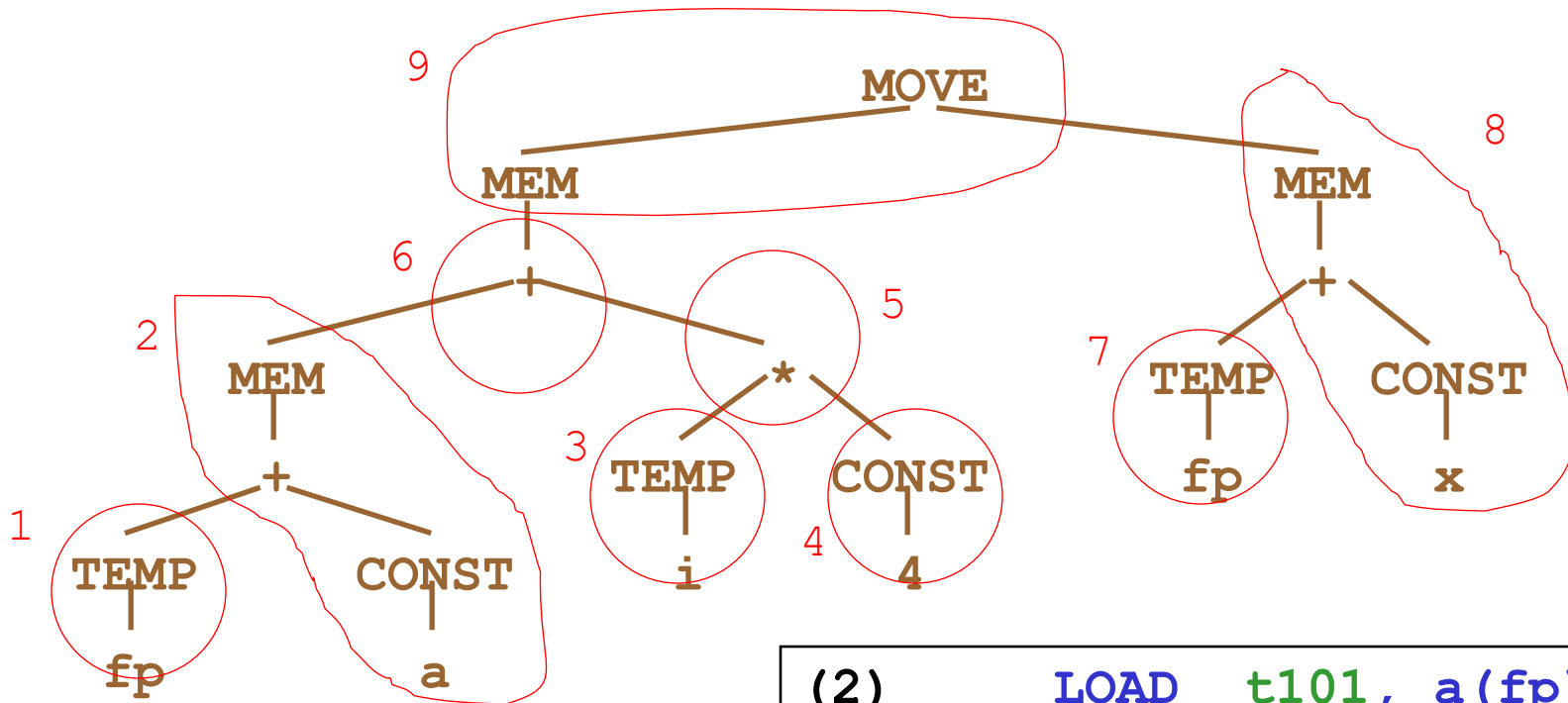
FP (wie Framepointer) statt Ci und Rsp

MOVE statt =

Bsp: $a[i] = x$



Beispiel Kachelung (aus Appel Buch)



Code der von der Befehlsauswahl
(instruction selection) generiert wird:

Variable **i** wurde schon **t7**.
zugewiesen

(2)	LOAD	t101, a(fp)
(4)	ADDI	t102, r0, 4
(5)	MUL	t103, t7, t102
(6)	ADD	t104, t101, t103
(8)	LOAD	t105, x(fp)
(9)	STORE	t105, 0(t104)

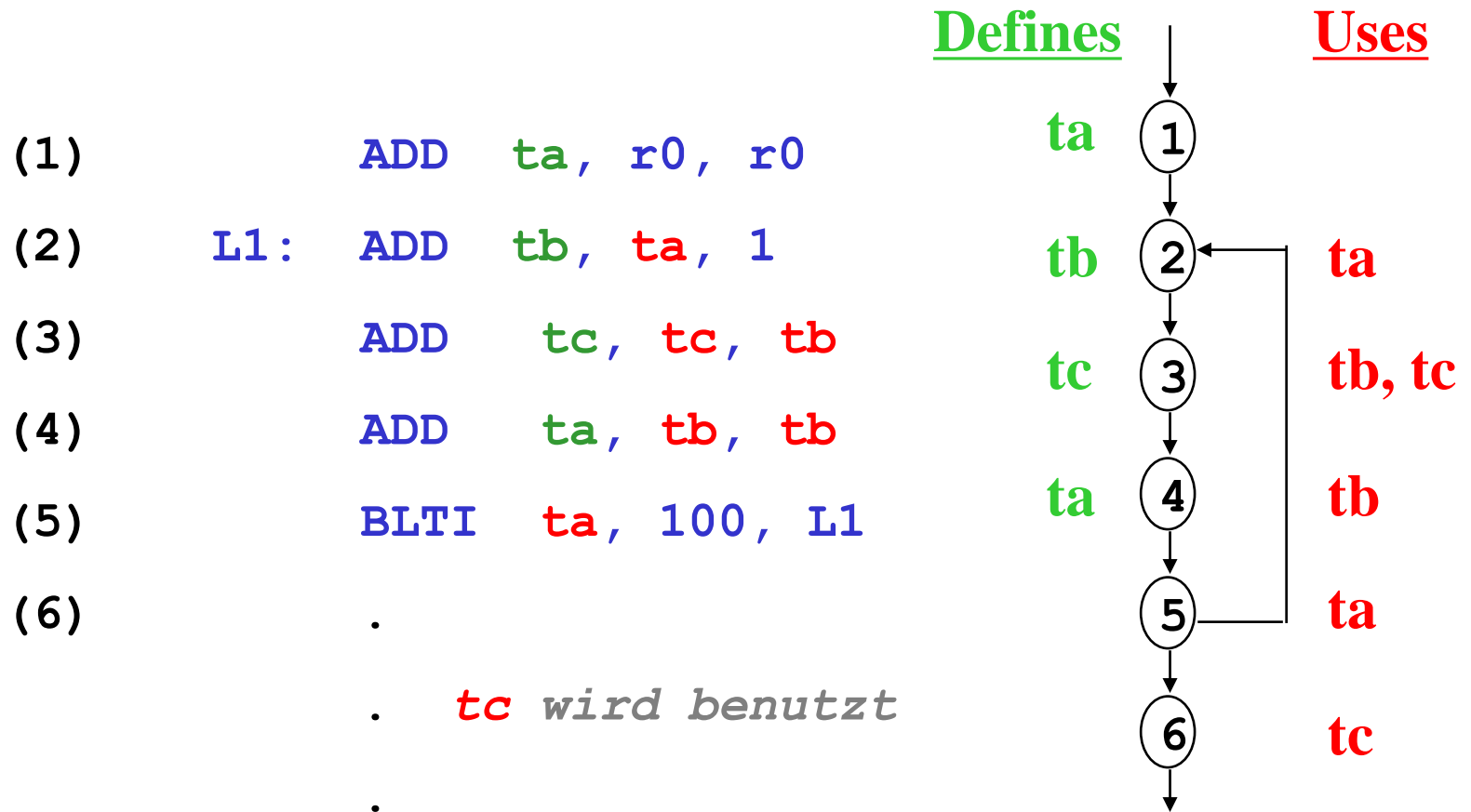
Register Allokation

Das Ergebnis der Befehlsauswahl (instruktion selection) ist ein Code der immer noch Pseudoregister (temporaries) nutzt.

- Diese müssen realen Registern zugeordnet werden
- Es können **mehrere** Pseudoregister einem realen Register zugeordnet werden wenn diese nicht gleichzeitig benutzt werden
- Spilling: Ordne einige Pseudoregister Speicherstellen (Stack) zu. Hier müssen neue Instruktionen eingefügt werden um die Werte aus dem Speicher zu holen und abzulegen

Vorgriff: Liveness Analysis

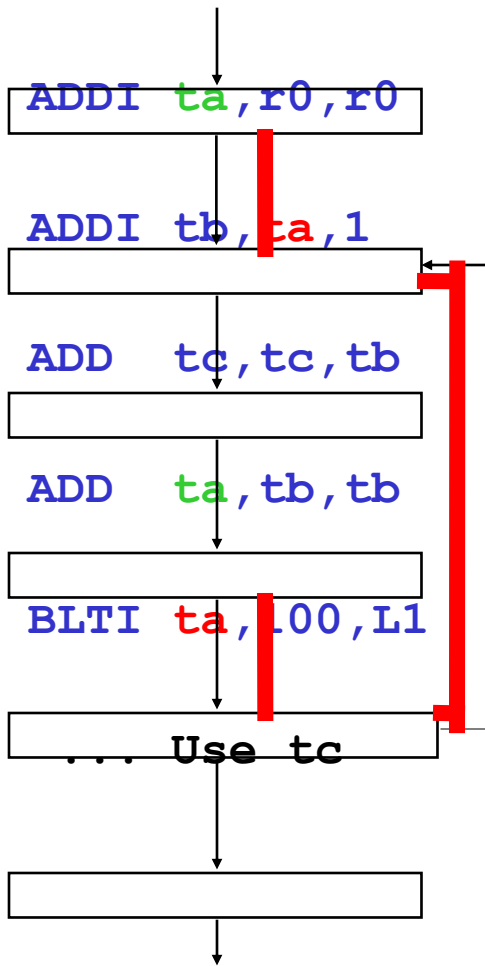
- Eine Variable ist lebendig (**live**) wenn sie einen Wert enthält der in Zukunft möglicherweise gebraucht wird.



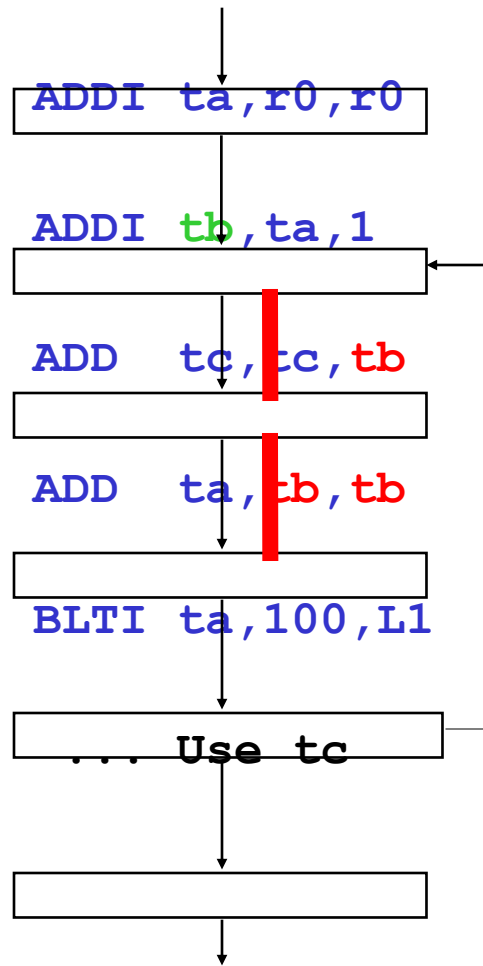
Kontrollflussgraph

Vorgriff: Liveness Analysis (2)

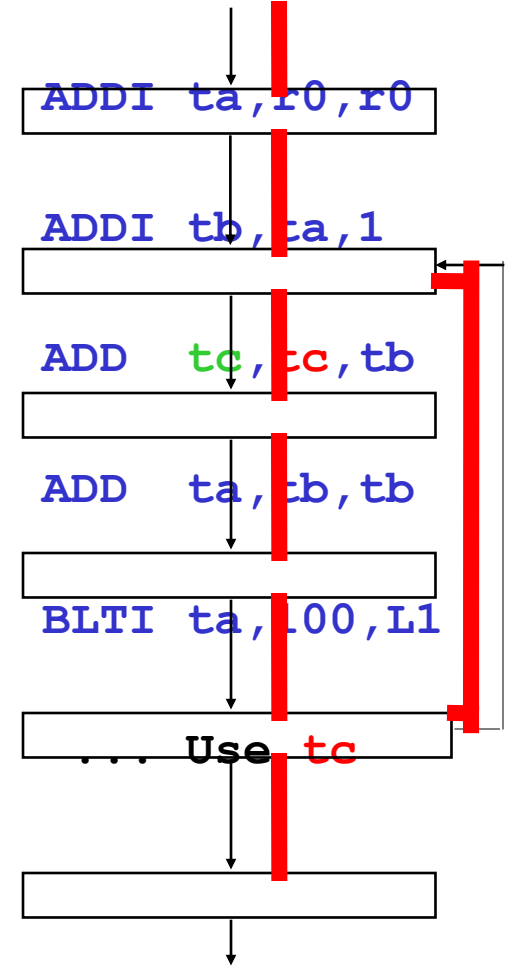
- **Liveness** von *ta*, *tb* und *tc*:



(ta)



(tb)



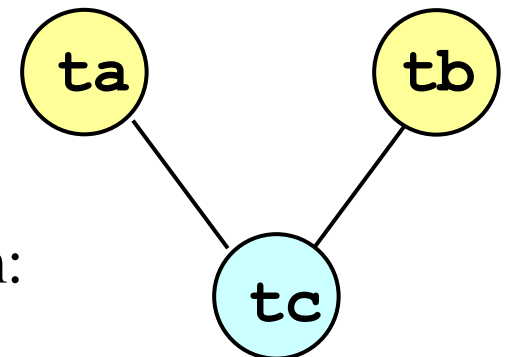
(tc)

Interferenzgraph

- **ta** und **tb** sind nie gleichzeitig lebendig, also können sie sich eine Register teilen.

Der *Interferenzgraph* drückt diese Bedingung aus

- Knoten sind temporäre Register
- Eine Kante verbindet zwei Knoten wenn diese nicht das selbe Register teilen können
- Benutze Graphfärbung um Knoten realen Registern zuzuweisen



Interferenzgraph für das Vorherige Programm:

Beispiel: Liveness Analyse

Defs

Uses

t7

t101

t102

t103

t104

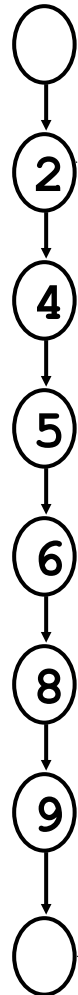
t105

t102

t101, t103

t105, t104

t7



t7

t101

t102

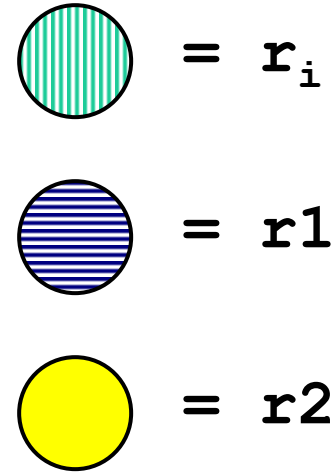
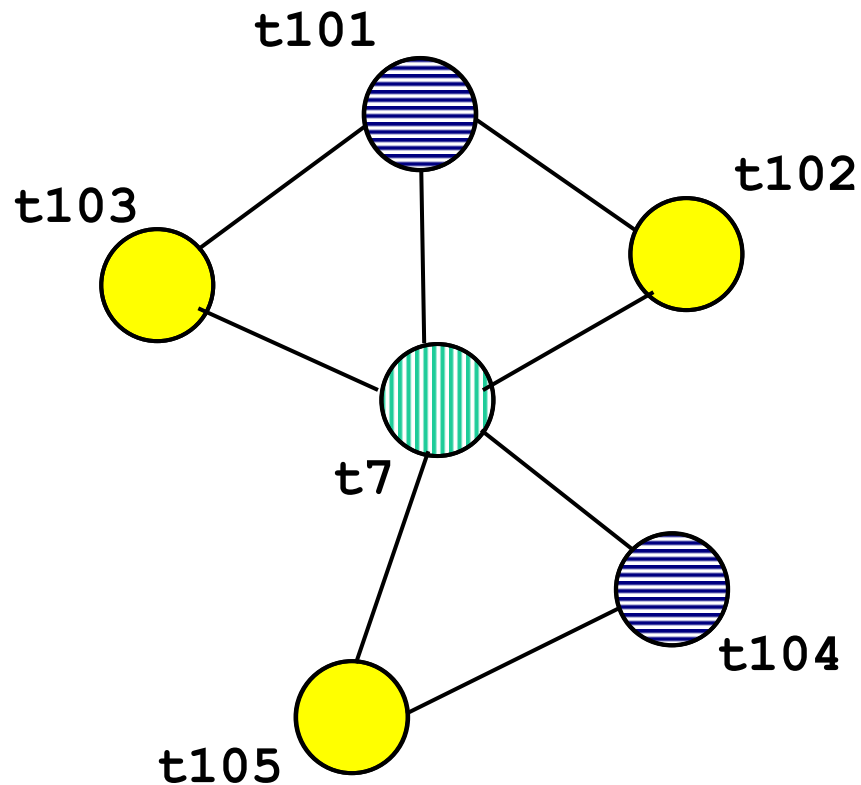
t103

t104

t105

(2) LOAD t101, a(fp)
(4) ADDI t102, r0, 4
(5) MUL t103, t7, t102
(6) ADD t104, t101, t103
(8) LOAD t105, x(fp)
(9) STORE t105, 0(t104)

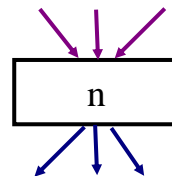
Beispiel: Interferenzgraph



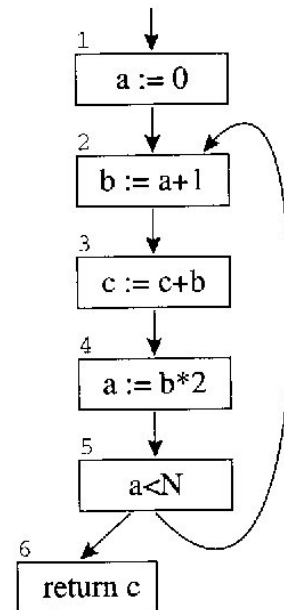
```
LOAD  r1, a(fp)
ADDI   r2, r0, 4
MUL    r2, ri, r2
ADD    r1, r1, r2
LOAD  r2, x(fp)
STORE r2, 0(r1)
```

Berechnung Liveness

- **Def**(n) = Variablen die im Knoten n definiert werden
- **Use**(n) = Variablen die im Knoten n verwendet werden
- Variable ist auf einer Kante **live** wenn es einen Pfad zu einem Knoten n gibt mit $v \in \text{Use}(n)$ der vorher keinen Knoten n' mit $v \in \text{Def}(n')$ besucht
- **in**[n] = Variablen die auf einer eingehenden Kante live sind (**live-in**)
- **out**[n] = Variable die auf einer ausgehenden Kante live sind (**live-out**)



```
a ← 0
L1 : b ← a + 1
      c ← c + b
      a ← b * 2
      if a < N goto L1
      return c
```



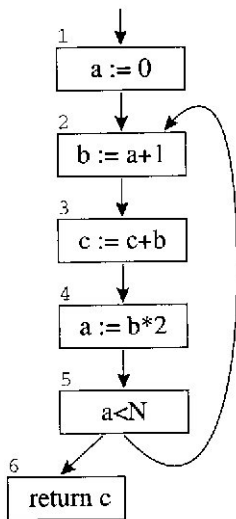
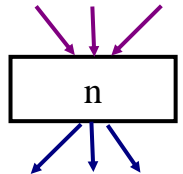
Kann für Grundblöcke erweitert werden

Berechnung Liveness: Algorithmus

- **Def**(n) = Variablen die im Knoten n definiert werden
- **Use**(n) = Variablen die im Knoten n verwendet werden
- Variable ist auf einer Kante **Live** wenn es einen Pfad zu einem Knoten n gibt mit $v \in \text{Use}(n)$ der vorher keinen Knoten n' mit $v \in \text{Def}(n')$ besucht
- **in**[n] = Variablen die auf einer eingehenden Kante live sind (**live-in**)
- **out**[n] = Variable die auf einer ausgehenden Kante live sind (**live-out**)

$$\text{in}[n] = \text{use}[n] \cup (\text{out}[n] - \text{def}[n])$$

$$\text{out}[n] = \bigcup_{s \in \text{succ}[n]} \text{in}[s]$$



EQUATIONS 10.3. Dataflow equations for liveness analysis.

```

for each  $n$ 
     $\text{in}[n] \leftarrow \{\}; \text{out}[n] \leftarrow \{\}$ 
repeat
    for each  $n$ 
         $\text{in}'[n] \leftarrow \text{in}[n]; \text{out}'[n] \leftarrow \text{out}[n]$ 
         $\text{in}[n] \leftarrow \text{use}[n] \cup (\text{out}[n] - \text{def}[n])$ 
         $\text{out}[n] \leftarrow \bigcup_{s \in \text{succ}[n]} \text{in}[s]$ 
    until  $\text{in}'[n] = \text{in}[n]$  and  $\text{out}'[n] = \text{out}[n]$  for all  $n$ 
    
```

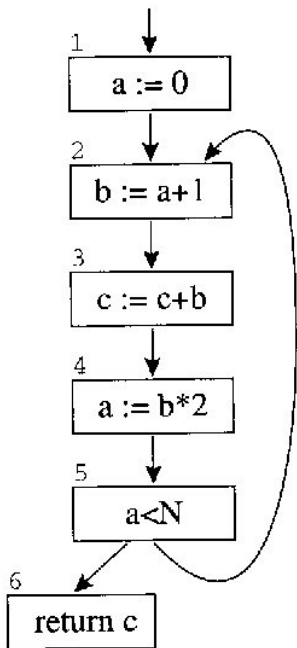
ALGORITHM 10.4. Computation of liveness by iteration.

Quelle: Appel, Modern Compiler Impl. in Java

Liveness

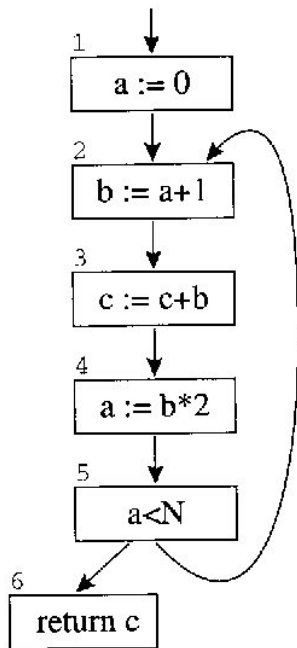
Reihenfolge der Bearbeitung hat Einfluss auf Konvergenz: möglichst in Rückwärtsrichtung berechnen und **out** vor **in** aktualisieren

Use **Def** | in out | in out | in out



Liveness

Reihenfolge der Bearbeitung hat Einfluss auf Konvergenz: möglichst in Rückwärtsrichtung berechnen und **out** vor **in** aktualisieren



			1st		2nd		3rd		4th		5th		6th		7th	
	use	def	in	out	in	out	in	out	in	out	in	out	in	out	in	out
1		a				a		a		ac	c	ac	c	ac	c	ac
2	a	b	a		a	bc	ac	bc	ac	bc	ac	bc	ac	bc	ac	bc
3	bc	c	bc		bc	b	bc	b	bc	b	bc	b	bc	bc	bc	bc
4	b	a	b		b	a	b	a	b	ac	bc	ac	bc	ac	bc	ac
5	a		a	a	a	ac	ac	ac	ac	ac	ac	ac	ac	ac	ac	ac
6	c		c		c		c		c		c		c		c	

TABLE 10.5. Liveness calculation following forward control-flow edges.

			1st		2nd		3rd	
	use	def	out	in	out	in	out	in
6	c			c		c		c
5	a		c	ac	ac	ac	ac	ac
4	b	a	ac	bc	ac	bc	ac	bc
3	bc	c	bc	bc	bc	bc	bc	bc
2	a	b	bc	ac	bc	ac	bc	ac
1		a	ac	c	ac	c	ac	c

TABLE 10.6. Liveness calculation following reverse control-flow edges.

8.10 Optimale Codeerzeugung für Ausdrücke

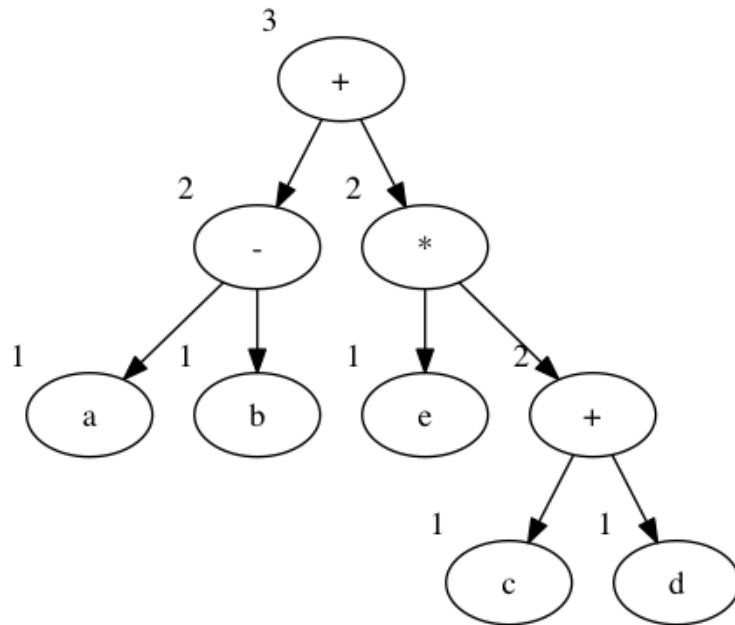
Für die Auswertung von **Ausdrücken** in Baumform kann man mit den Ershov-Zahlen berechnen wie viele Register notwendig sind wenn man **keine** temporären Variablen benutzen will

Ausdrucksbäume: Ershovzahlen

1. Fall: Blätter
→ Ershovzahl 1
2. Fall Knoten mit einem Kind:
→ Ershovzahl des Kindes
3. Fall Knoten mit zwei Kindern:
 - a. Unterschiedliche Ershovzahlen
→ Größere Ershovzahl
 - b. Gleiche Ershovzahlen
→ Ershovzahl +1

Ausdrucksbäume: Ershovzahlen

$(a-b)+e*(c+d)$



$t1 = a-b$

$t2 = c+d$

$t3 = e*t2$

$t4 = t1-t3$

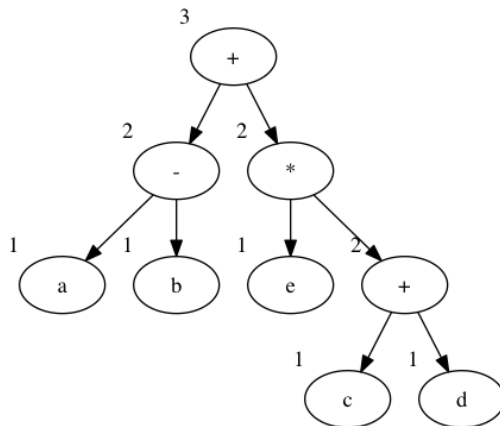
Codegen mit Ershovzahlen

Berechnung der Basis: Basis **b** bei der Wurzel auf 1 setzen

1. Fall Blatt: Basis **b**
2. Fall: zwei Kinder mit gleicher Ershovzahl **k**:
 - a. Rechter Teilbaum Basis **b+1**
 - b. Linker Teilbaum Basis **b**
3. Fall: zwei Kinder mit ungleicher Ershovzahl **k** und **m** ($m < k$)
 - a. Rechter Teilbaum Basis **b**
 - b. Linker Teilbaum Basis **b**

Ershovzahlen

Basis b?

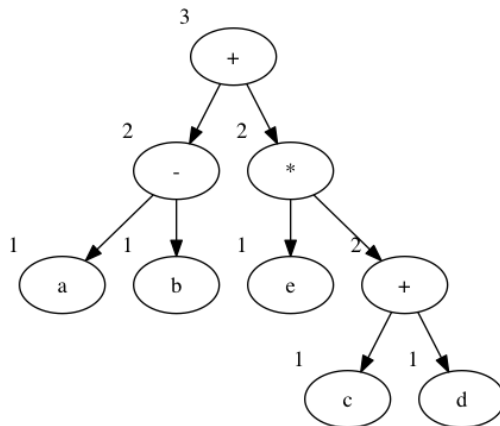


Codegen mit Ershovzahlen

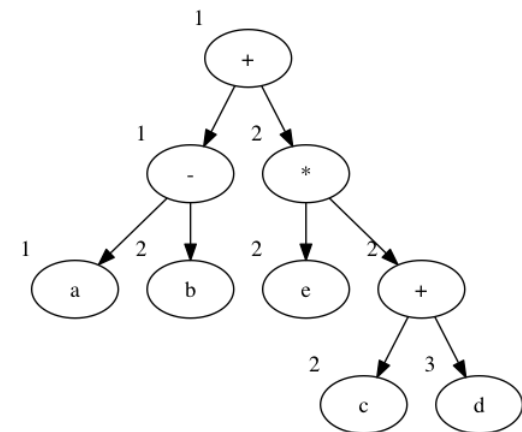
Berechnung der Basis: Basis **b** bei der Wurzel auf 1 setzen

1. Fall Blatt: Basis **b**
2. Fall: zwei Kinder mit gleicher Ershovzahl **k**:
 - a. Rechter Teilbaum Basis **b+1**
 - b. Linker Teilbaum Basis **b**
3. Fall: zwei Kinder mit ungleicher Ershovzahl **k** und **m** ($m < k$)
 - a. Rechter Teilbaum Basis **b**
 - b. Linker Teilbaum Basis **b**

Ershovzahlen



Basis b



Codegen mit Ershovzahlen

Berechnung von Registernummern mit Ershovzahlen:

Basis **b** bei der Wurzel auf 1 setzen

1. Fall Blatt x: LD R**b**, x mit Basis **b**

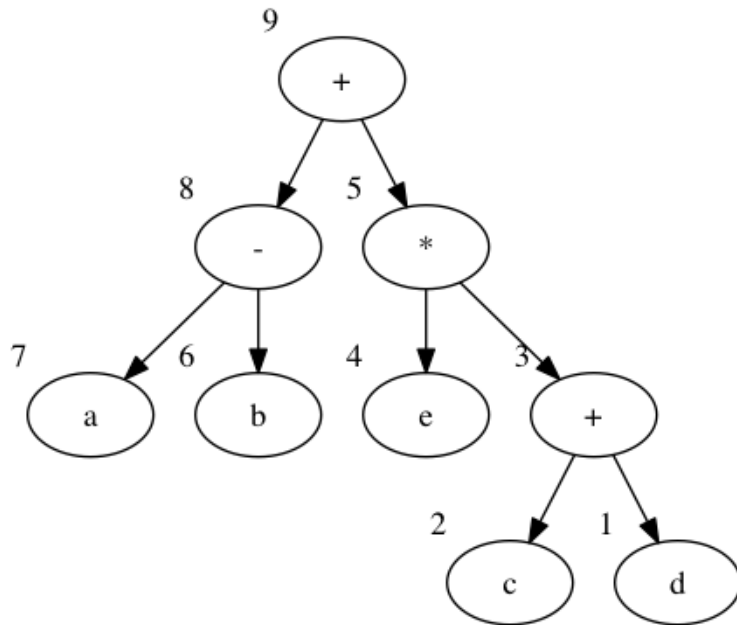
Codegen mit Ershovzahlen

- 2. Fall: zwei Kinder mit gleicher Ershovzahl $k-1$ und Knoten mit EZ k :
 - a. Codegen für rechten Teilbaum mit Basis $b+1$
Ergebnis in R_{b+k-1}
 - b. Codegen für linken Teilbaum mit Basis b
Ergebnis in R_{b+k-2}
 - c. Codegen für Knoten selbst mit OP $R_{b+k-1}, R_{b+k-2}, R_{b+k-1}$
Ergebnis in R_{b+k-1}

Codegen mit Ershovzahlen

3. Fall: zwei Kinder mit ungleicher Ershovzahl $k-1$ und m ($m < k-1$)
 - a. Codegen für Teilbaum k mit Basis b
Ergebnis in $Rb+k-1$
 - b. Codegen für Teilbaum m mit Basis b
Ergebnis in $Rb+m-1$
 - c. Codegen für Knoten selbst mit OP $Rb+k-1, Rb+m-1, Rb+k-1$
oder $Rb+k-1, Rb+k-1, Rb+m-1$ (je nachdem ob der kleinere Baum rechts oder links war)
Ergebnis in $RB+k-1$

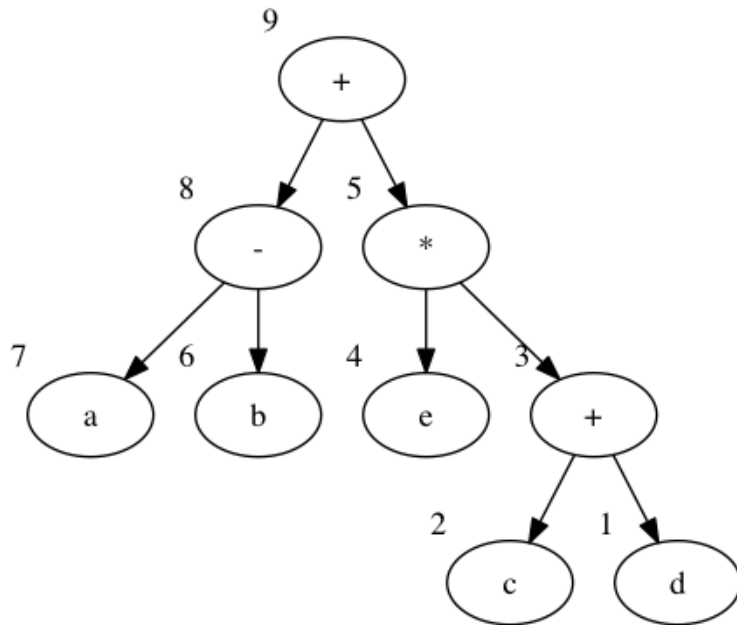
Codegen mit Ershovzahlen



```
LD R3, d
LD R2, c
ADD R3, R2, R3
LD R2, e
MUL R3, R2, R3
LD R2, b
LD R1, a
SUB R2, R1, R2
ADD R3, R2, R3
```

Codegen mit Ershovzahlen

Bsp Spilling mit nur
R=2 Registern:



```
LD R2, d
LD R1, c
ADD R2, R1, R2
LD R1, e
MUL R2, R1, R2
ST Memory[Temp], R2
LD R2, b
LD R1, a
SUB R2, R1, R2
LD R1, Memory[Temp]
ADD R2, R2, R1
```

Generiere Code für Teilbäume mit $k \leq R$ und
speichere Ergebnis in Hauptspeicher.

Registerallokation – Heuristik 8.8.4

- Sei R die Anzahl der verfügbaren Register
- Lege Knoten mit Grad $g < R$ auf den Stack
- Entferne Knoten und seine Kanten: wir bekommen einen neuen Graphen G' : eine R -Färbung von G' kann immer zu einer k -Färbung von G erweitert werden (da $g < R$)
- Sind irgendwann alle Knoten entfernt ist eine Färbung mit R Registern möglich, sonst ist **spilling** notwendig

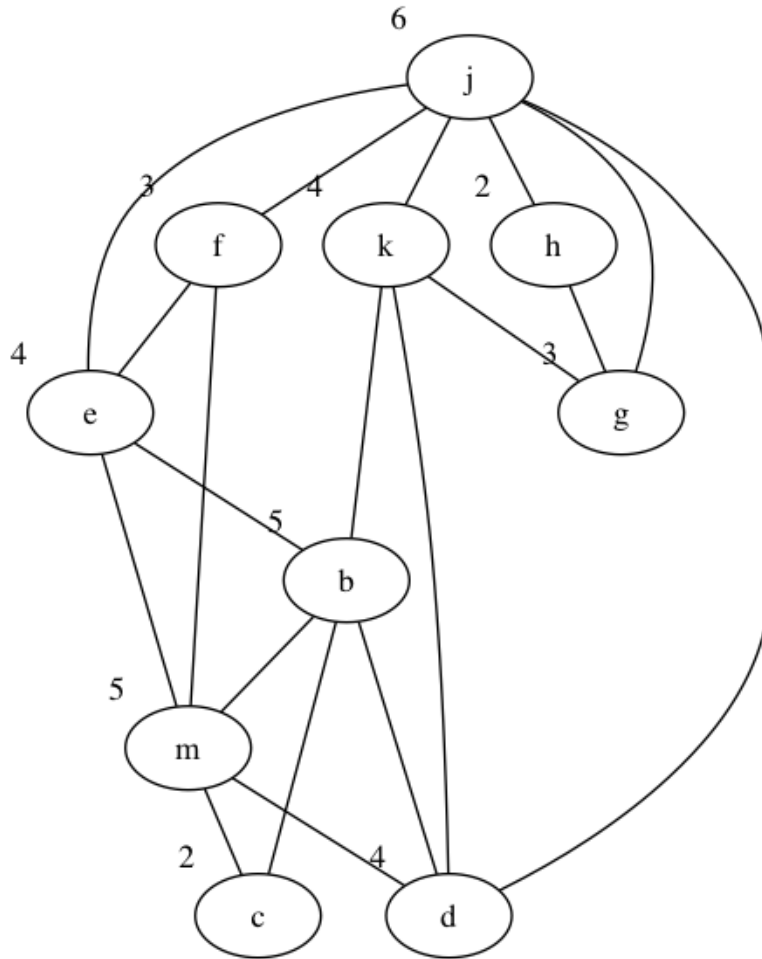
Registerallokation mit fester Registeranzahl

```
Live-in: k j
g := mem[j+12]
h := k - 1
  f := g * h
e := mem[j+8]
  m := mem[j+16]
  b := mem[f]
  c := e + 8
  d := c
  k := m + 4
  j := b
Live-out: d k j
```

Beispiel: Färbung mit
R=4 Registern.

Build:
Baue den Graph auf

Registerallokation mit fester Registeranzahl



Zahlen: Anzahl Kanten (Knotengrad)

Beispiel: Färbung mit $R=4$ Registern.

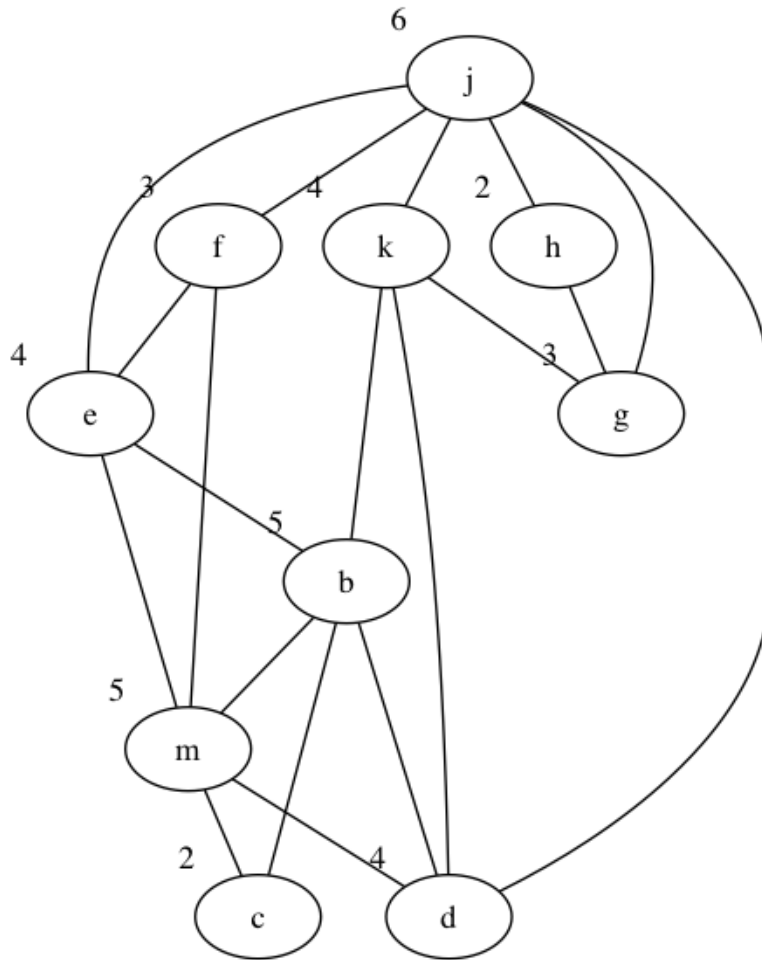
Simplify:

- Lege Knoten mit Grad $g < R$ auf den Stack
- Entferne Knoten und seine Kanten.

Select:

Sind alle Knoten entfernt ist eine Färbung mit R Registern möglich, sonst ist spilling notwendig.

Registerallokation mit fester Registeranzahl



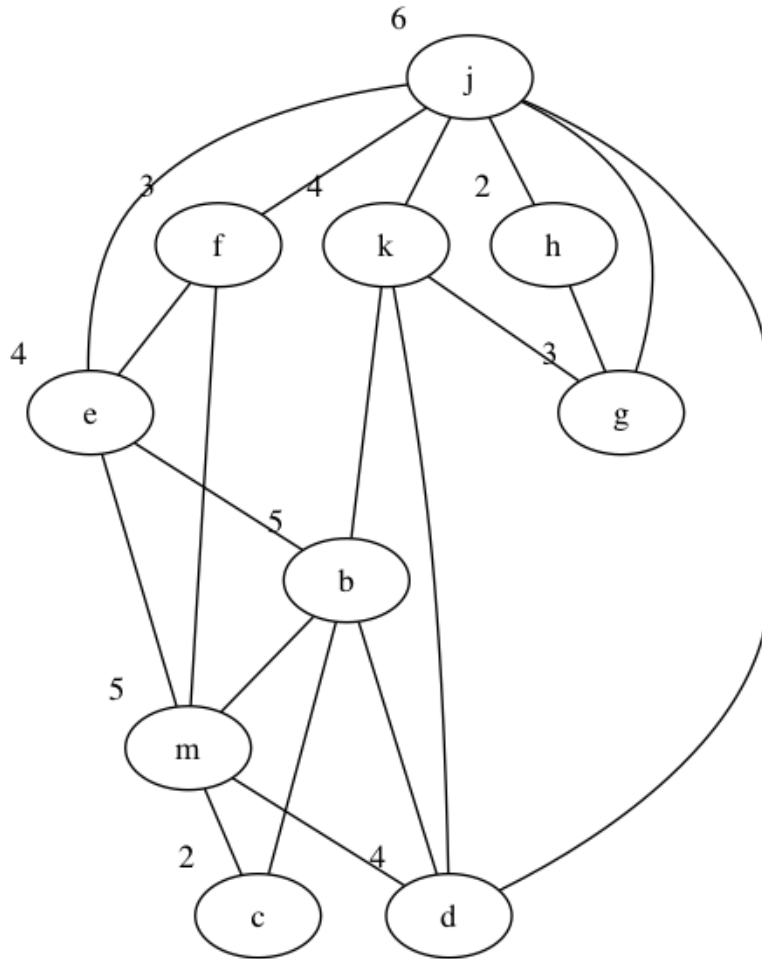
Zahlen: Anzahl Kanten (Knotengrad)

Beispiel: Färbung mit $R=4$ Registern.

Select:

- Sind alle Knoten entfernt, Graph umgekehrt wieder aufbauen indem alle Knoten vom Stack gepushed werden.
- Pro Knoten: Farbe wählen, die keiner seiner Nachbarn hat.
- Ist keine Färbung mit R Registern möglich, so ist spilling notwendig.

Registerallokation mit fester Registeranzahl



Beispiel: Färbung mit
 $R=4$ Registern.

Beispiel: g, h, k, d, j, e, f, b, c, m

Registerallokation mit fester Registeranzahl

Beispiel: Färbung mit
R=4 Registern.

Live-in: k j
g := mem[j+12]
h := k - 1
 f := g * h
e := mem[j+8]
 m := mem[j+16]
 b := mem[f]
 c := e + 8
 d := c
 k := m + 4
 j := b
Live-out: d k j

Live-in: k j
R4 := mem[R3+12]
R2 := R1 - 1
 R2 := R4 * R2
R4 := mem[R3+8]
 R1 := mem[R3+16]
 R2 := mem[R2]
 R3 := R4 + 8
 R4 := R3
 R1 := R1 + 4
 R3 := R2
Live-out: d k j

Beispiel: g, h, k, d, j, e, f, b, c, m

Verschmelzen von Knoten

Optimierung: In Instruktionen wie $x:=y$ sollten beide Variablen wenn möglich die selbe Farbe erhalten.

Instruktionen wie $R1:=R1$ können dann entfernt werden!

Live-in: k j

$g := \text{mem}[j+12]$

$h := k - 1$

$f := g * h$

$e := \text{mem}[j+8]$

$m := \text{mem}[j+16]$

$b := \text{mem}[f]$

$c := e + 8$

$d := c$

$k := m + 4$

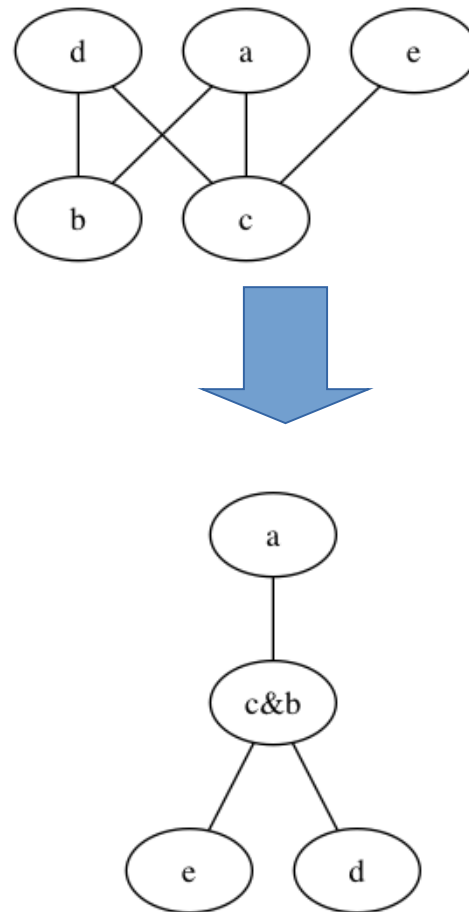
$j := b$

Live-out: d k j

Verschmelzen von Knoten

Zwei Knoten in einem Interferenzgraph die nicht interferieren können zu einem Knoten verschmolzen werden. Der neue Knoten hat die Vereinigung beider Kantenmengen.

Live-in: k j
g := mem[j+12]
h := k - 1
 f := g * h
e := mem[j+8]
 m := mem[j+16]
 b := mem[f]
 c := e + 8
 d := c
 k := m + 4
 j := b
Live-out: d k j



Verschmelzen von Knoten

Das Verschmelzen von zwei Knoten **a** und **b** kann einen K-färbaren Graphen in einen L-färbaren ($L > K$) verwandeln. Wir wollen jedoch in keinem Fall mehr Register verbrauchen als notwendig. Unter folgenden Kriterien verschlechtert die Verschmelzung die K-färbbarkeit nicht:

Kriterium von Briggs:

Der neue Knoten $a \& b$ hat weniger als K Nachbarn mit
 $\text{Grad} \geq K$

Kriterium von George:

Wenn jeder Nachbar von a

Entweder: schon mit b interferiert

Oder: einen $\text{Grad} < K$ hat

Knoten mit Vorfärbung

Manche Knoten können schon eine Vorfärbungen haben.

Caller-Save Register:

Der Aufrufer muss das Register sichern.

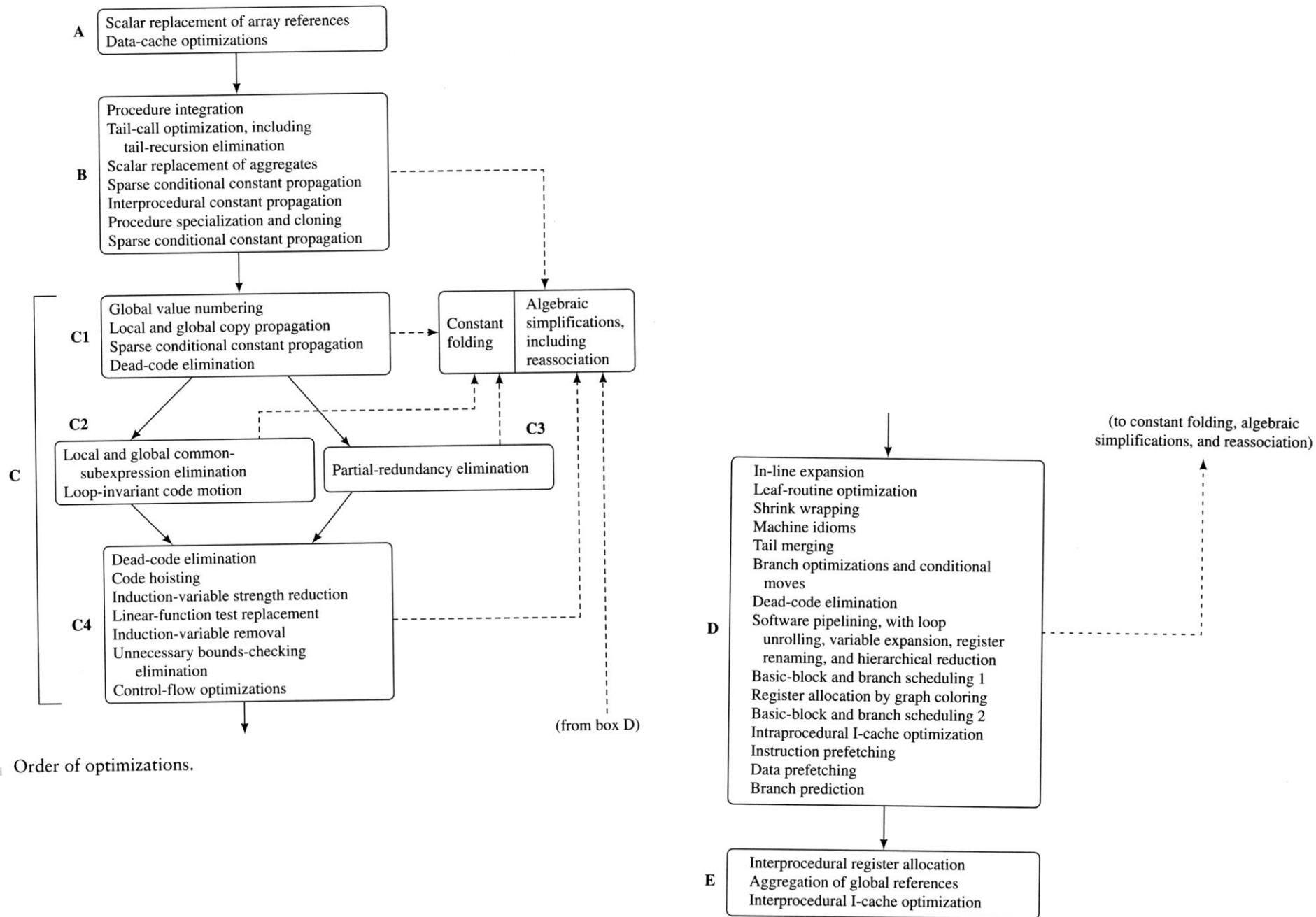
Callee-Save Register:

Der Aufgerufene muss das Register sichern. Diese Register sind beim Eintritt und Verlassen einer Funktion live. Register kommen damit explizit schon im Zwischencode vor. Damit diese dennoch frei zur Verwendung sind werden diese i.d.R. einer neuen Variable am Anfang und Ende zugewiesen.

Vorgehen: Füge die Register in den Interferenzgraph ein. Lass diese jedoch nicht auf den Stack wandern. Berücksichtige die Register auch beim Verschmelzen von Knoten.

Code Optimierung

- Techniques for improving quality of generated code.
- Law of diminishing returns applies.
- Most important is good register allocation.
- *Peephole optimization* applied locally to generated target code: remove redundant code:
 - `MOV R1,R1` (or `ADD r1,r0,r1`)
 - `JUMP L1`
 - `L1: ...`



Order of optimizations.

(continued)

Gemeinsame Teilausdrücke

- Eliminate repeated evaluation of expressions whose value does not change.

- E.g.

`a[i+j] = a[i+j] + 1;`

calculates

`(base address of a) + (i + j)*(element size)`

twice; save in a register and re-use.

- Usually done by transformations on AST
(or IR tree)

Constant Folding

- Evaluate expressions as much as possible at compile time.
- Obvious: $2 + 5$
- Less obvious: $((2 * x) + 1) * 3$
- Constant propagation:

```
double pi = 3.141592654;  
double twopi = 2*pi;
```
- Copy propagation:

```
a = y + z;  
b = y;  
c = b + z;    -- common subexpression
```


Dead-Code Elimination

- Remove code that is unreachable, or has no effect.

- Common case:

```
#define DEBUG 0  
  
.  
.  
    if (DEBUG) { ... }
```

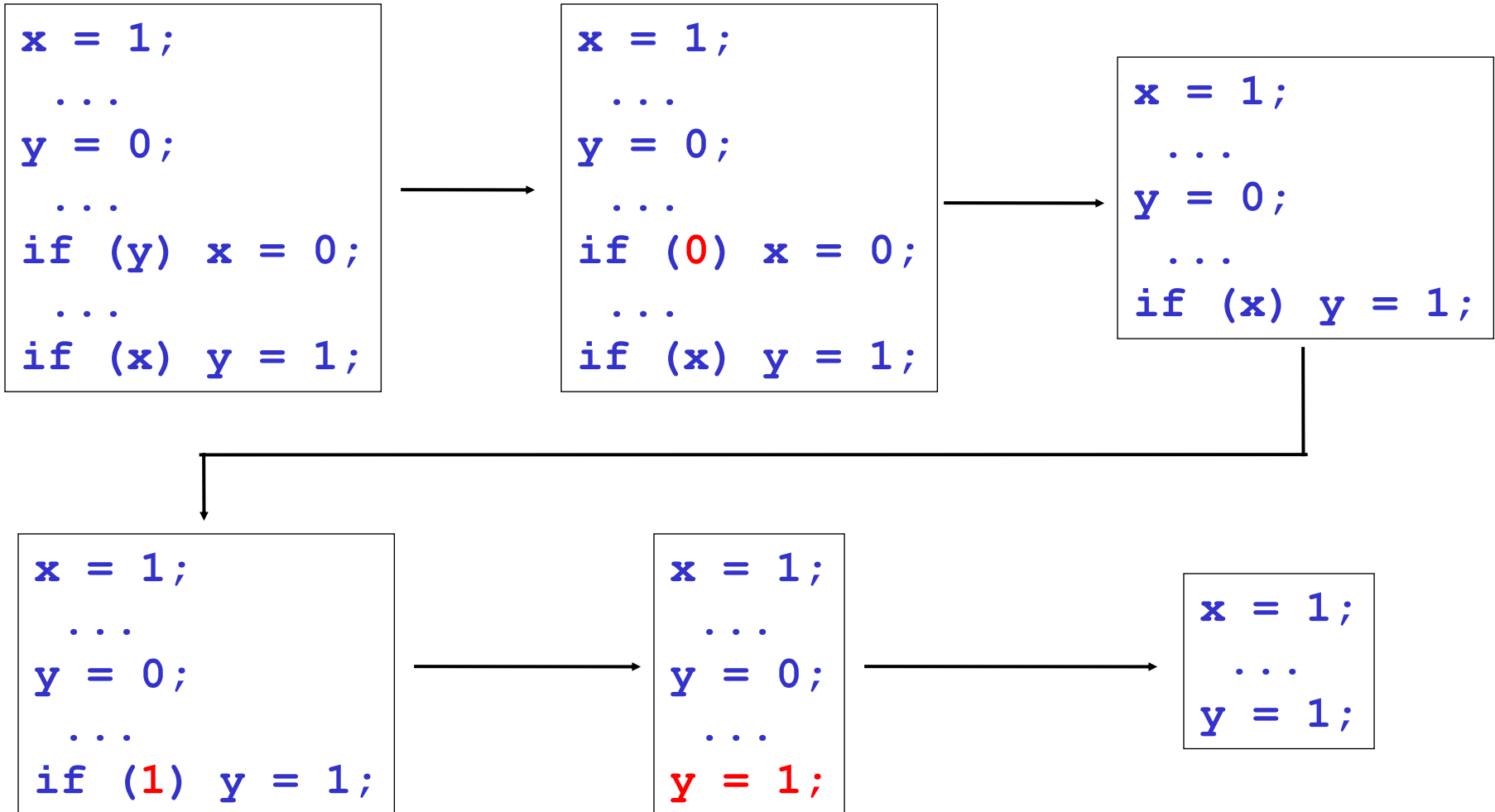
- But be careful:

```
.  
x = y / z;  
return y;
```

- Optimization must preserve program behaviour.

Reihenfolge von Optimierungen

- Order of applying transformations matters: may have to iterate.



Zusammenfassung

(Was ist wichtig für die Klausur)

- Zwischencode
 - Null-Address + Drei-Address Code, IR Bäume
- Generierung von 0-, 3-Addresscode vom AST
- Grundblöcke, Kontrollflussgraph
- Instruction Selection*
- Register Allokation
 - Liveness Analyse, Interferenzgraph
- Code Optimierung*

*: Grundlagen, keine Details

Was haben wir gelernt

