Compiler und Codegenerierung

Hw-Sw-Co-Design

WS 2008/09

Rolf Drechsler

Informatik, Universität Bremen

Nach: Jürgen Teich, Universität Erlangen

Überblick

- Compiler - Aufbau
- Codegenerierung
- Codeoptimierung
- Codegenerierung für Spezialprozessoren
- Retargetable Compiler

Registervergabe

- globale Registerzuweisung
  - bestimmte Anzahl von Registern reservieren
    - für globale Variablen
    - für Schleifenvariablen
    - für Variablen in Grundblöcken
  - benutzerdefinierte Registervergabe
    - z.Bsp. in der Programmiersprache C:
      `register int i;`
- Verwendungszähler
- Registerzuweisung durch Graphfärbung

Verwendungszähler

- eine Schleife L besteht aus mehreren Grundblöcken

- wenn eine Variable **a** während der ganzen Schleife L in einem Register bleibt, ergeben sich Kosteneinsparungen von:
  - 1 Kosteneinheit bei Verwendung von **a**
    - `ADD R0, R1` (Kosten 1) statt `ADD a, R1` (Kosten 2)
  - 2 Kosteneinheiten am Ende des Grundblocks, falls **a** im Grundblock definiert wurde und nach dem Grundblock noch aktiv ist
    - kein Abspeichern `MOV R0, a`

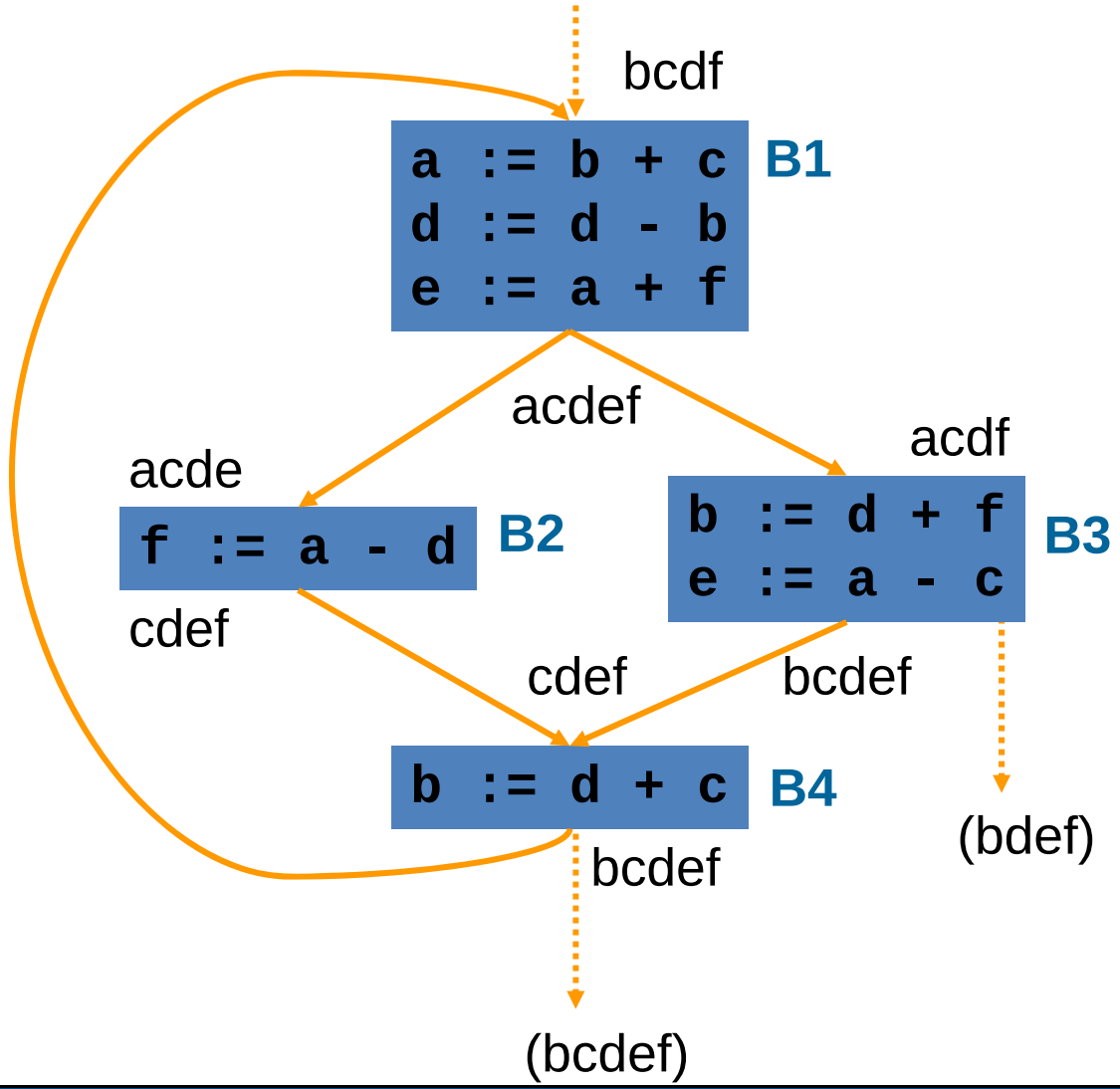
Verwendungszähler

- Kosteneinsparung für gesamte Schleife

$$\sum_{B \in L} \left(\text{verwendet}(a, B) + 2 \cdot \text{aktiv}(a, B) \right)$$

  - verwendet(***a, B***) ist die Anzahl der Verwendungen von **a** im Grundblock ***B*** vor einer eventuellen Definition von **a**
  - aktiv(***a, B***) ist 1, wenn **a** im Grundblock B definiert wurde und am Ende aktiv sein muss; sonst 0
- Näherung, da angenommen wird, dass
  - alle Blöcke gleich oft ausgeführt werden
  - die Schleife oft durchlaufen wird

Verwendungszähler - Beispiel



aktiv(a, B1) = 1
verwendet(a, B2) = 1
verwendet(a, B3) = 1

Kosteneinsparung

a :	4
b :	6
c :	3
d :	6
e :	4
f :	4

bei 3 Registern:
b → R0, d → R1, a → R2

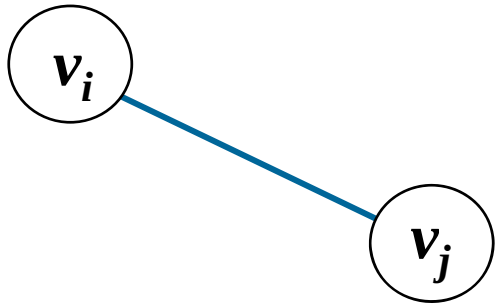
Registervergabe durch Graphfärbung

- Ablauf
  1. Codegenerierung mit unbeschränkter Anzahl von Registern (symbolische Register), d.h. jeder Variablenname ist ein symbolisches Register
  2. Bestimmung der Lebenszeit der Variablen (symbolischen Register)
  3. Konstruktion des Registerkonfliktgraphen
  4. Abbildung der symbolischen Register auf physikalische Register durch Graphfärbung

Registerkonfliktgraph

- Definition: Ein Registerkonfliktgraph $G\ (V, E)$ ist ein ungerichteter Graph, in dem die Knoten V die Variablen (symbolische Register) darstellen und die Kanten E die Konflikte zwischen den Variablen.

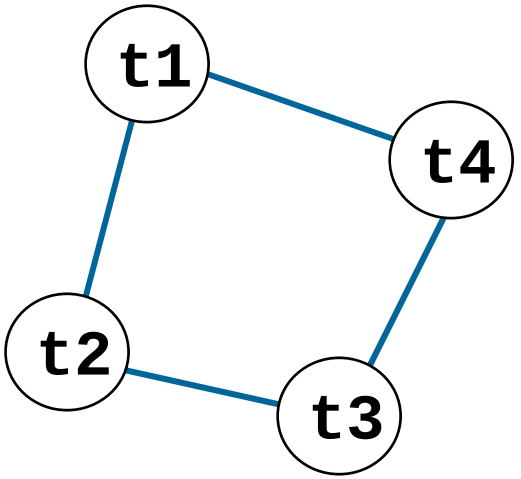
  – Eine Kante zwischen zwei Knoten v_i und v_j bedeutet, dass sich die Lebenszeiten von v_i und v_j überlappen. Die Variablen v_i und v_j können sich nicht ein Register teilen.

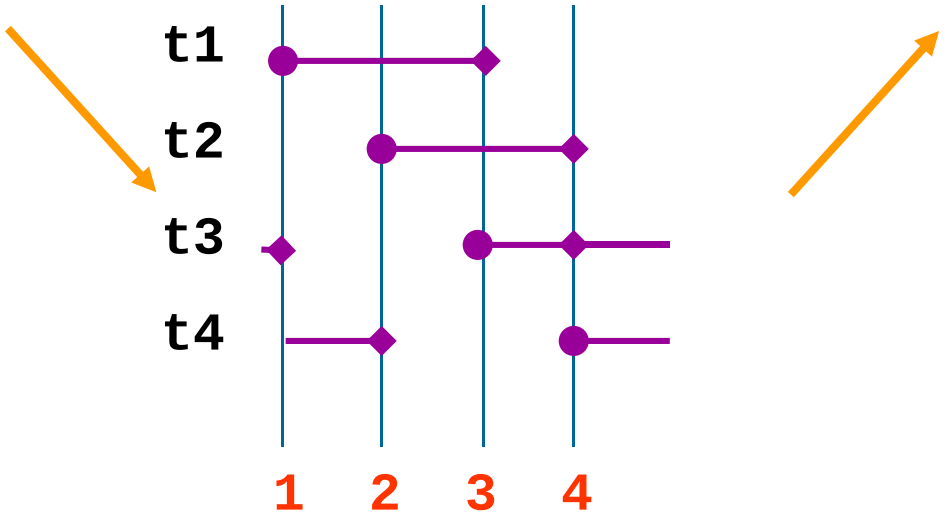


Registerkonfliktgraph - Beispiel

Schleifenrumpf

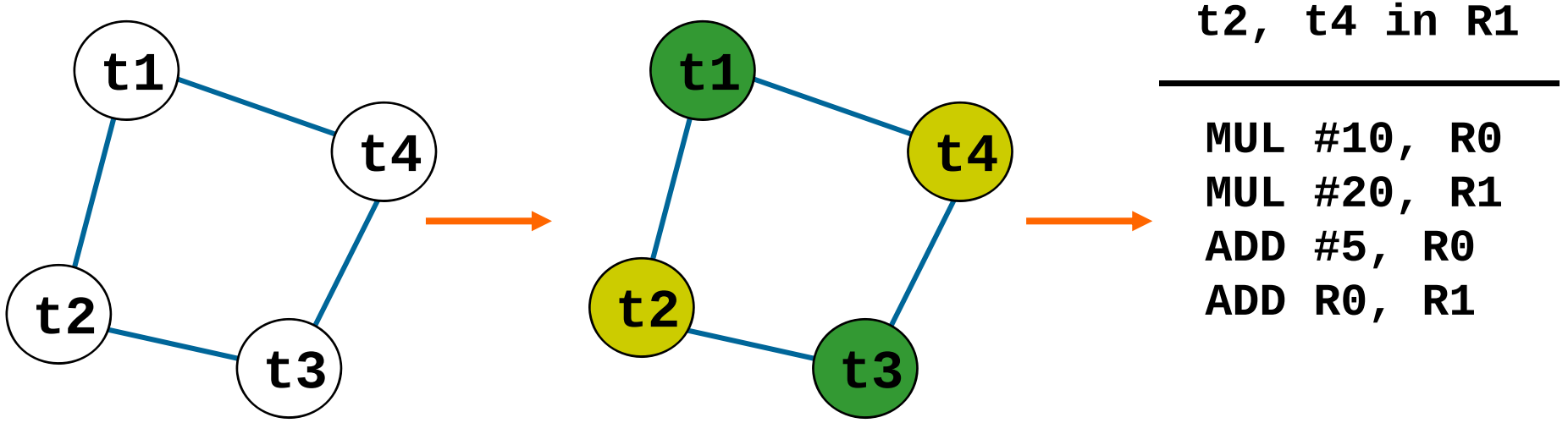
```
(1)   t1 := t3 * 10
(2)   t2 := t4 * 20
(3)   t3 := t1 + 5
(4)   t4 := t2 + t3
```





Graphfärbung (1)

- Färbe die Knoten von G mit l Farben derart, dass niemals zwei benachbarte Knoten die gleiche Farbe bekommen.
  - Graphfärbung ist NP-vollständig.



Graphfärbung (2)

- Heuristik: Ist ein Graph G mit l Farben färbbar?

1. finde einen Knoten v_i in G mit Grad$(v_i) < l$
2. entferne v_i und alle seine Kanten; das ergibt G'
3. wenn $G' = \{\}$:

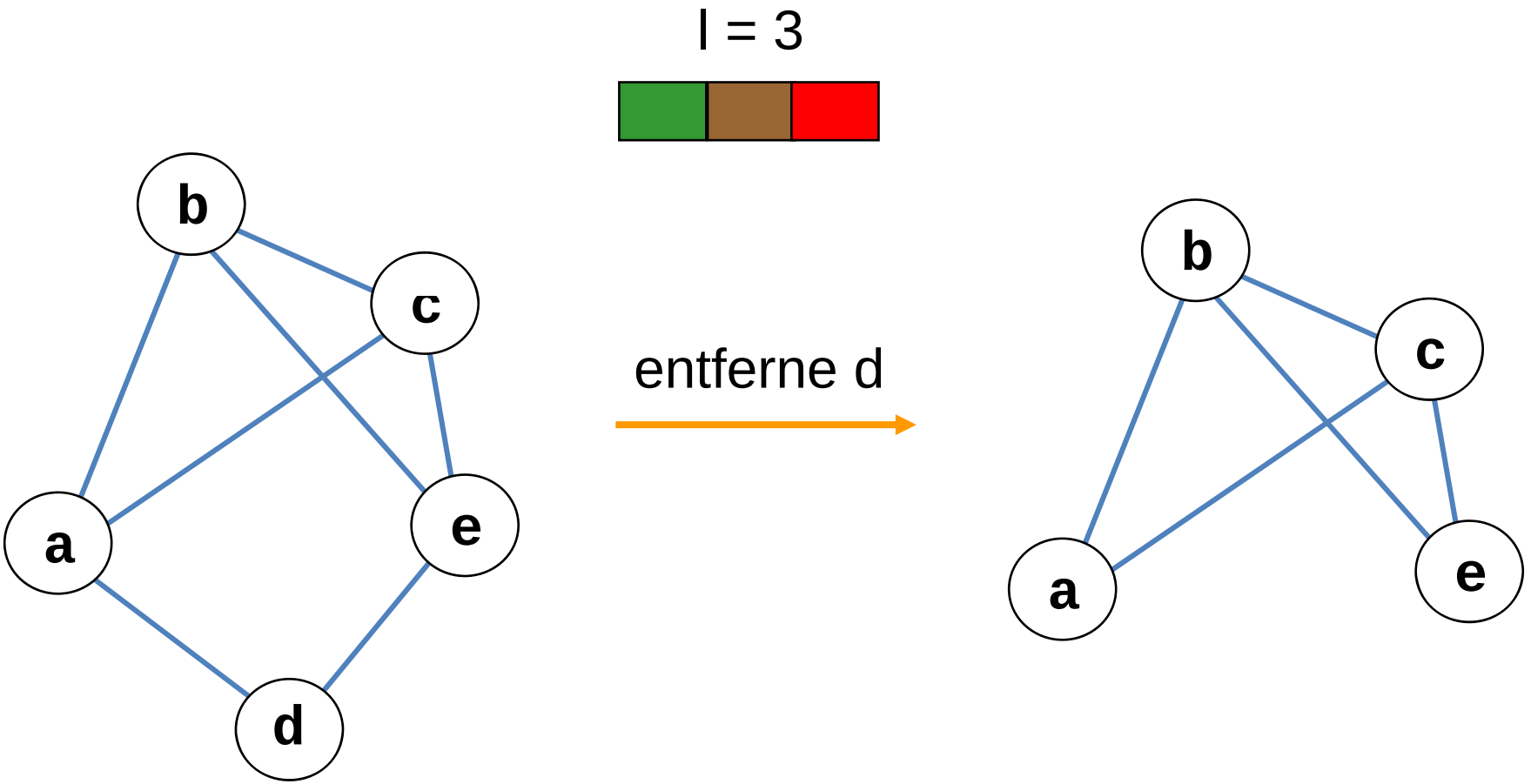
   l - Färbung möglich

   wenn alle Knoten in G' einen Grad $>= l$ haben:

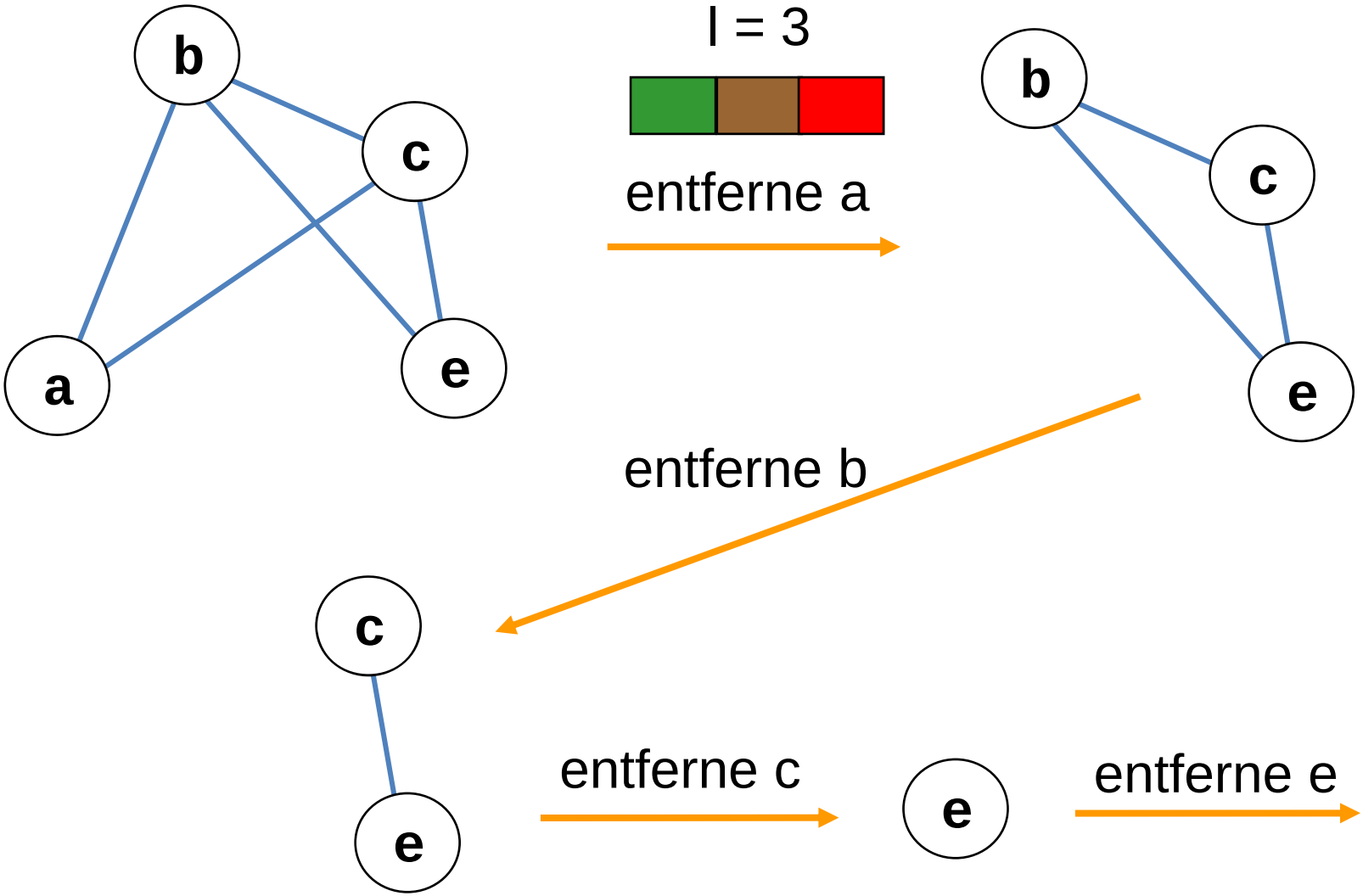
   l - Färbung nicht möglich

   $G = G'$ und gehe zu 1.

Graphfärbung - Beispiel (1)



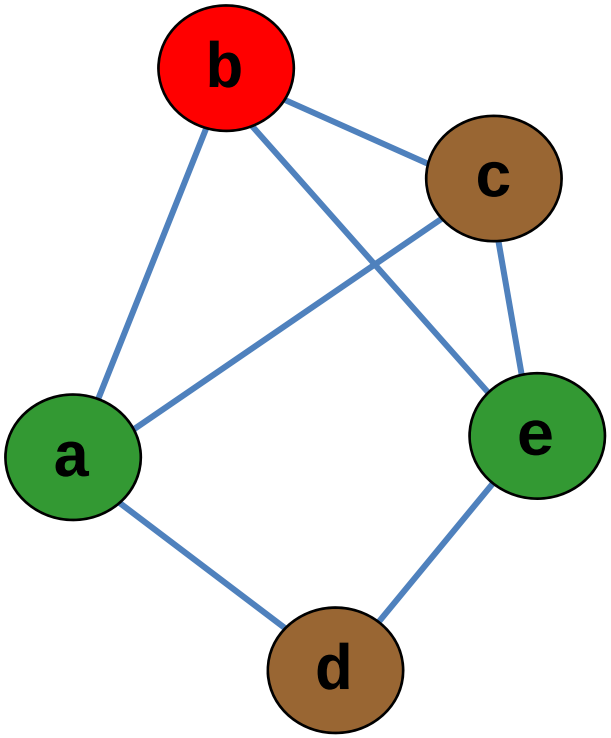
Graphfärbung - Beispiel (2)



Graphfärbung - Beispiel (3)

l = 3

Reihenfolge der entfernten Knoten:
d, a, b, c, e

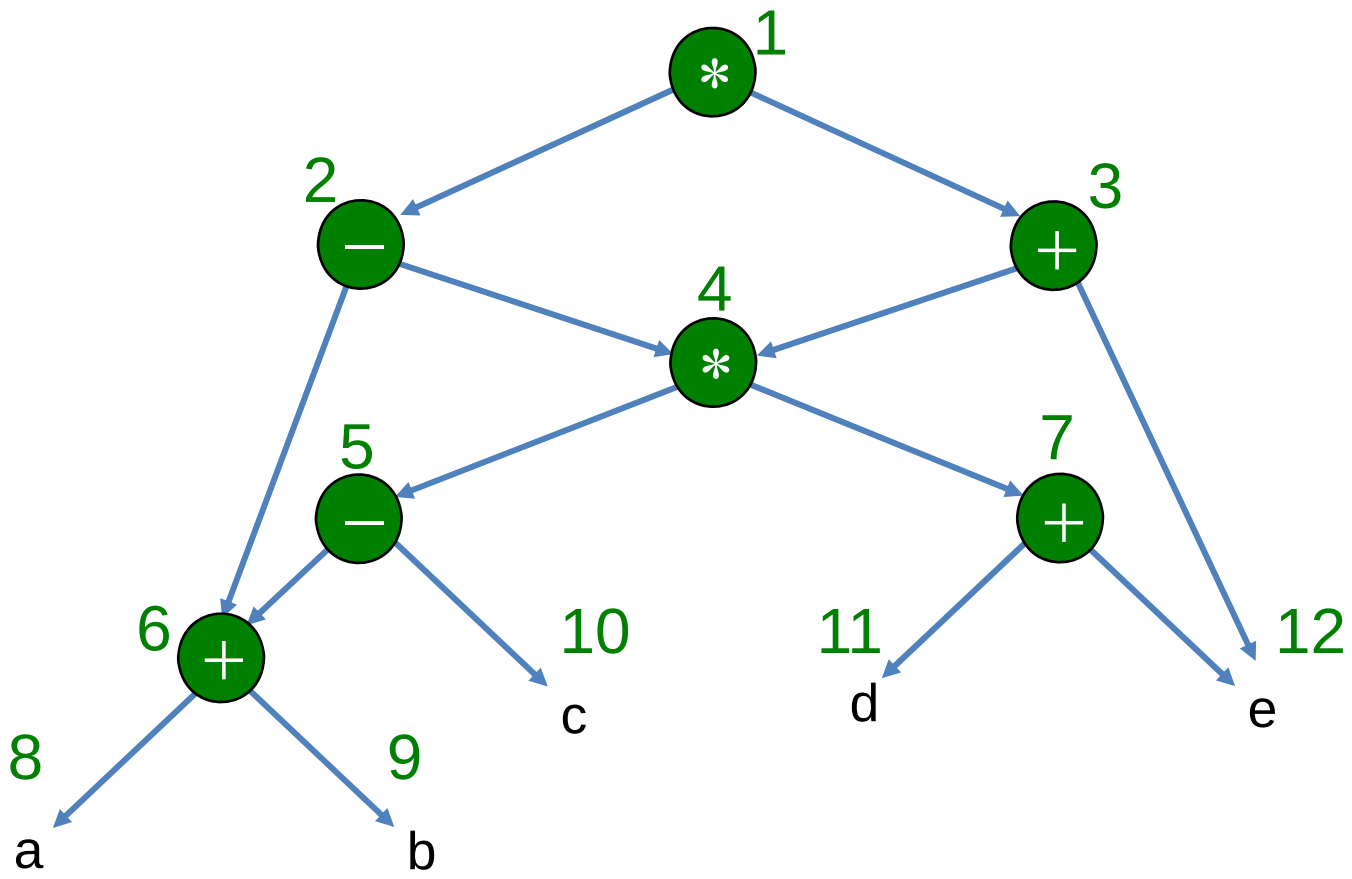


Codegenerierung für DAGs (1)

- die Berechnungsreihenfolge der Knoten des DAGs hat großen Einfluss auf die Anzahl der benötigten Instruktionen

- Heuristik für die Berechnungsreihenfolge
  - Solange es innere Knoten gibt, die noch nicht gereiht sind:
    - wähle einen Knoten n, dessen Eltern gereiht sind, und reihe ihn
    - solange das am weitesten links liegende Kind m von n keine ungereihten Eltern hat und kein Blatt ist:
      1. reihe m
      2. $n := m$

Codegenerierung für DAGs - Beispiel



verkehrte Reihenfolge: 1 2 3 4 5 6 7

Codegenerierung für DAGs (2)

- Falls ein DAG mit n Knoten ein Baum ist, kann man einen Algorithmus angeben, der in $O(n)$ Zeit optimalen Code (kürzeste Codesequenz) generiert.

- common subexpressions
  - der DAG ist kein Baum

  → DAG an den Knoten, die gemeinsame Teilausdrücke darstellen, aufspalten und für jeden Teil separat optimalen Code generieren

Dynamische Programmierung (1)

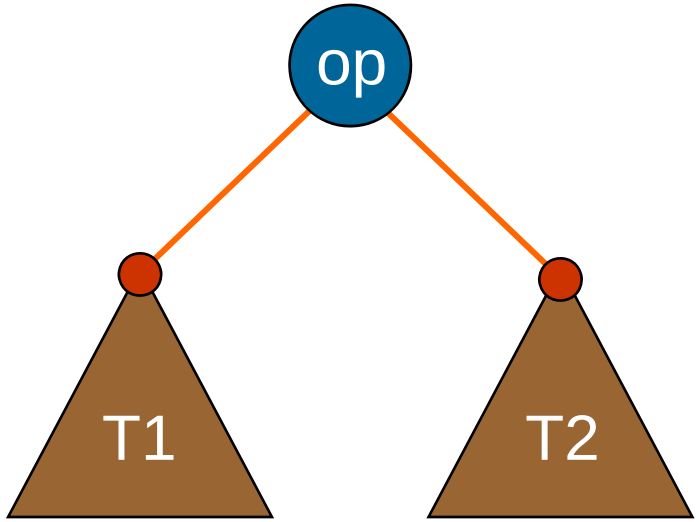
- Maschinenmodell erweitert auf komplexere Instruktionen
  - *n* Register **R0 … Rn-1**
  - Befehle **Ri := E**
    E beliebiger Ausdruck mit Registern und Speicherstellen
  - falls **E** mehrere Register hat, muss **Ri** eines davon sein
  - Load: **Ri:= M**, Store: **M:= Ri**, Kopierbefehl: **Ri:= Rj**
  - hier Vereinfachung:
    alle Befehle (alle Adressierungsarten) haben Kosten von 1
- Beispiele
  - **ADD R0, R1** → **R1 := R1 + R0**
  - **ADD *R0, R1** → **R1 := R1 + ind R0**
  - **SUB a, R0** → **R0 := R0 - a**

Dynamische Programmierung (2)

- Prinzip der dynamischen Programmierung auf die Codegenerierung für DAGs angewendet

optimaler Code für E = (T1 op T2):

1. optimaler Code für T1, T2
2. entweder T1, T2, op
   oder T2, T1, op



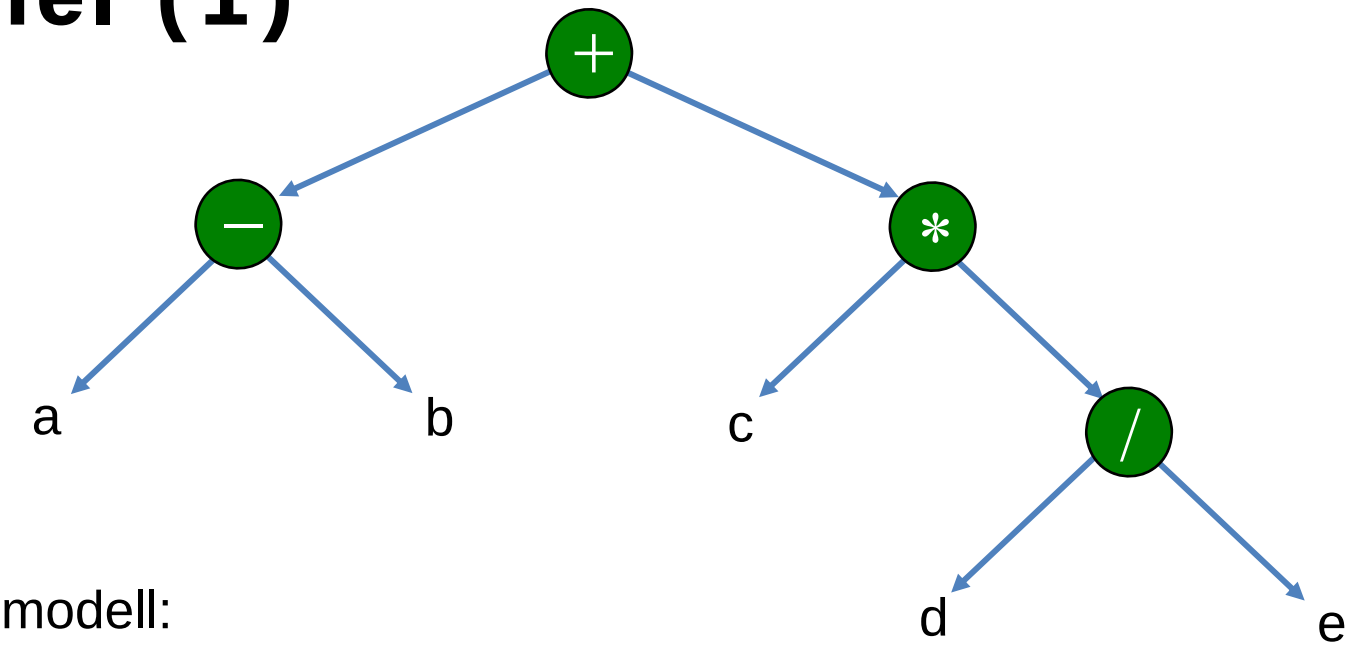
Dynamische Programmierung (3)

- das Verfahren hat 3 Phasen
  1. Berechnung von Kostenvektoren
  2. Bestimmung der Befehlsreihenfolge
  3. Generierung von Zielcode

- Berechnung von Kostenvektoren für jeden Knoten n (bottom up) :
  - C[i] optimale Kosten zur Berechnung von n mit i Registern
  - C[0] optimale Kosten zur Berechnung von n, wenn das Ergebnis in den Speicher abgelegt wird

Beispiel (1)

DAG:



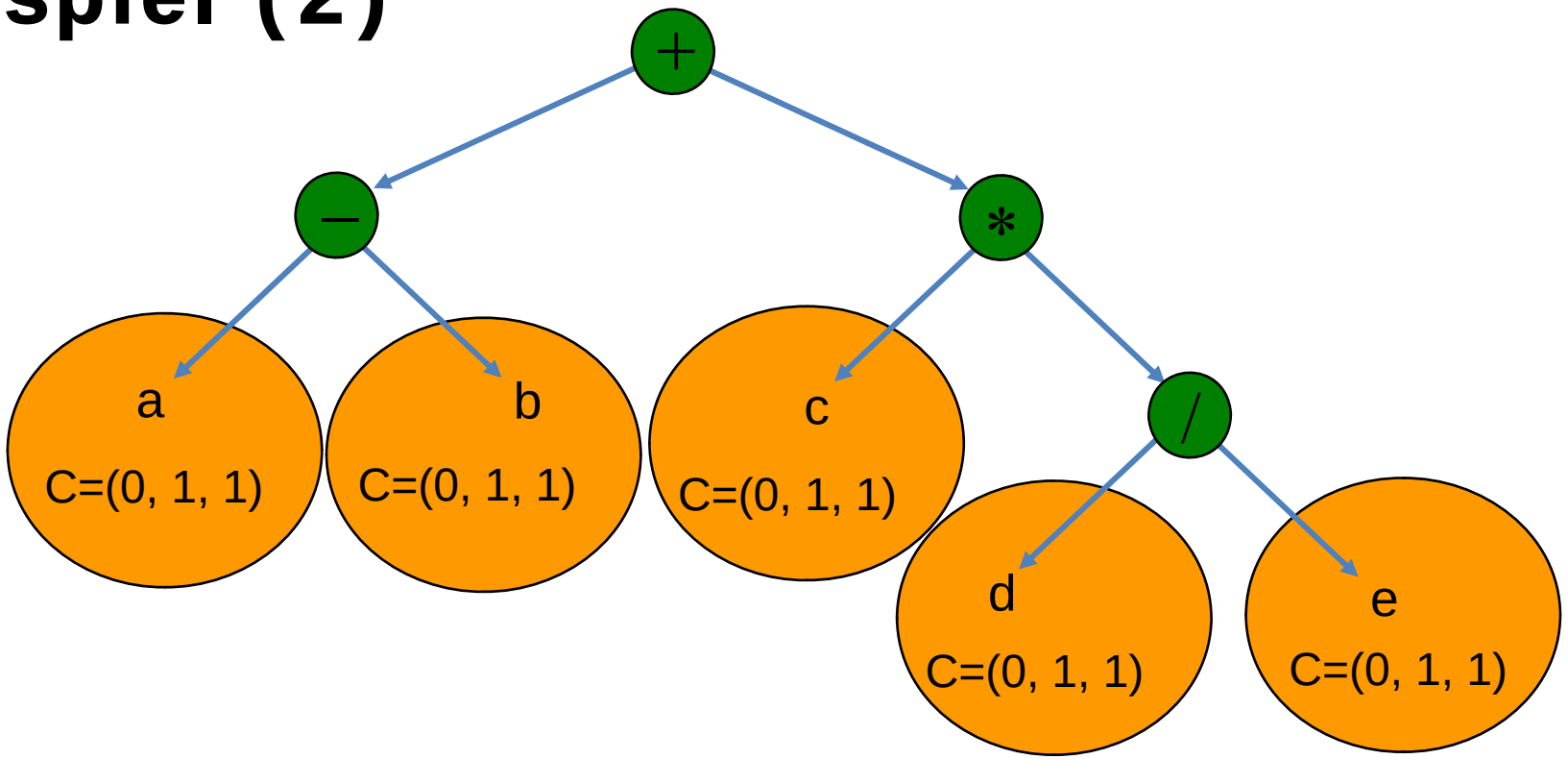
Maschinenmodell:

- Instruktionssatz

```
Ri := Mj
Ri := Ri op Rj
Ri := Ri op Mj
Ri := Rj
Mj := Ri
```

- zwei Register vorhanden

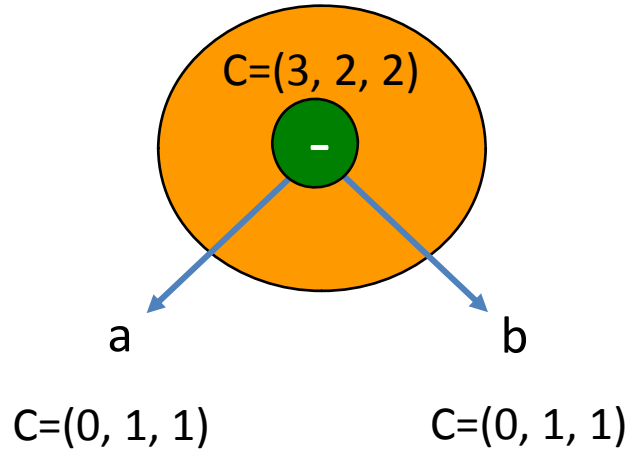
Beispiel (2)



Kosten für Berechnung von **a**:

- in den Speicher C[0] = 0 (**a** ist bereits dort)
- mit einem Register C[1] = 1 (**Ri := M**)
- mit zwei Registern C[2] = 1

Beispiel (3)



- mit <u>einem Register</u>: **Ri := Ri - M**
  linker Teilbaum mit 1 Register,
  rechter Teilbaum im Speicher: C[1] = 2
- mit <u>zwei Registern</u>: **Ri := Ri - M**

  Kosten = 2 = C[2]

  oder **Ri := Ri - Rj**

  linker Teilbaum mit 2 Registern,
  rechter Teilbaum mit 1 Register:
  Kosten = 3

  rechter Teilbaum mit 2 Registern,
  linker Teilbaum mit 1 Register:
  Kosten = 3

- in den <u>Speicher</u>: C[0] = 3

Beispiel (4)

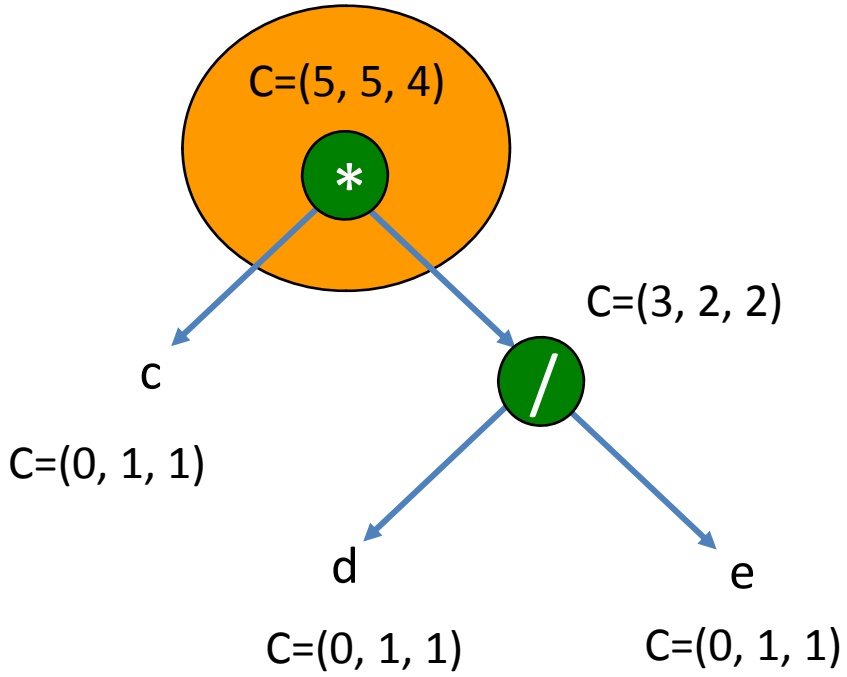
- mit einem Register: **Ri := Ri * M**
  linker Teilbaum mit 1 Register,
  rechter Teilbaum im Speicher: C[1] = 5
- mit zwei Registern: **Ri := Ri * M**

  Kosten = 5

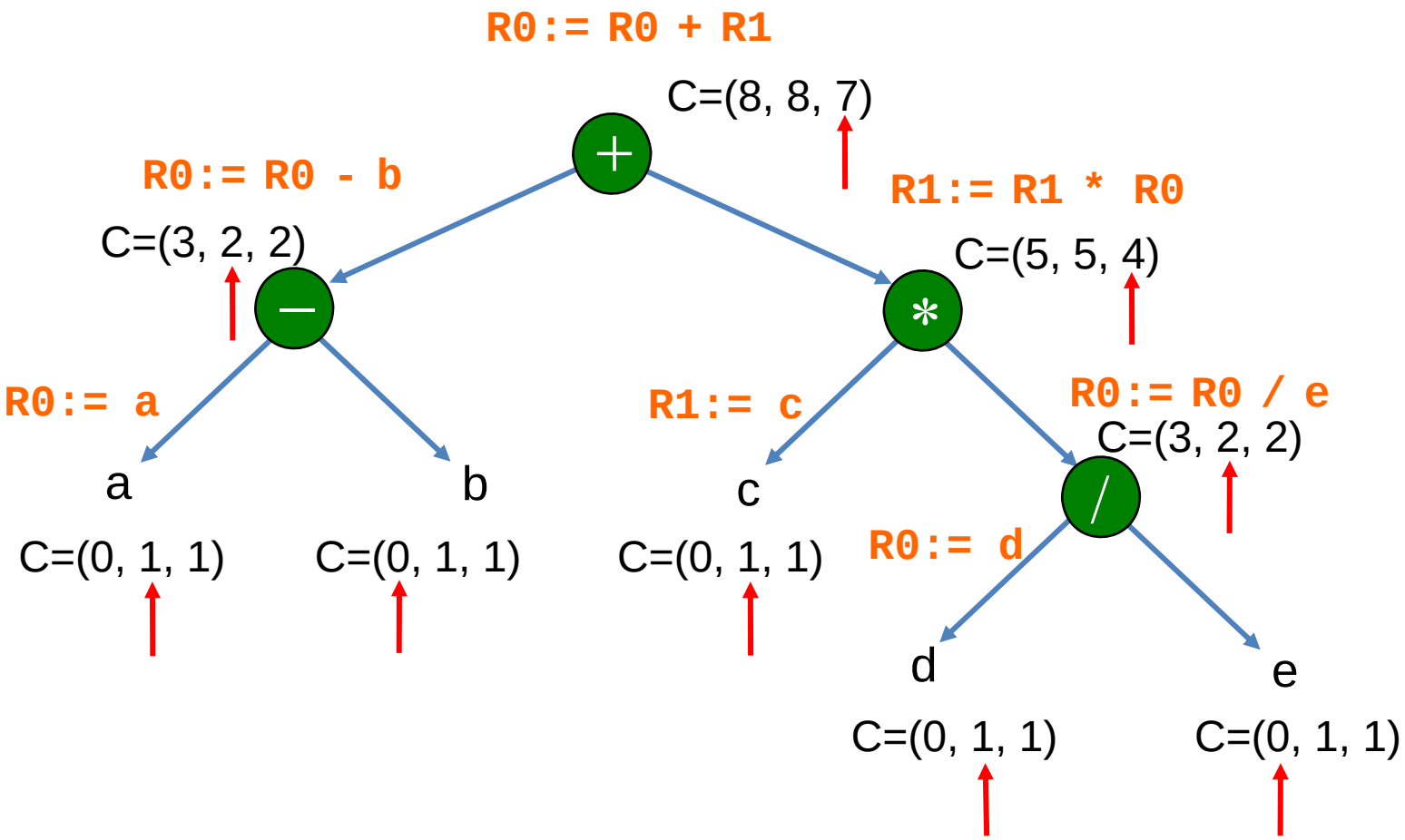
  oder **Ri := Ri * Rj**

  linker Teilbaum mit 2 Registern,
  rechter Teilbaum mit 1 Register:
  Kosten = 4 = C[2]

  rechter Teilbaum mit 2 Registern,
  linker Teilbaum mit 1 Register:
  Kosten = 4
- in den Speicher: C[0] = 5



Beispiel (5)


R0:= R0 + R1
C=(8, 8, 7)
+
R0:= R0 - b
C=(3, 2, 2)
—
R1:= R1 * R0
C=(5, 5, 4)
*
R0:= a
a
C=(0, 1, 1)
b
C=(0, 1, 1)
R1:= c
c
C=(0, 1, 1)
R0:= R0 / e
C=(3, 2, 2)
/
R0:= d
d
C=(0, 1, 1)
e
C=(0, 1, 1)

Überblick

- Compiler - Aufbau
- Codegenerierung
- Codeoptimierung
- Codegenerierung für Spezialprozessoren
- Retargetable Compiler

Codeoptimierung

- Transformationen auf dem Zwischencode und auf dem Zielcode möglich
- Peephole Optimierung
  - kleines Fenster (peephole) wird über den Code gezogen
  - mehrere Durchläufe, da eine Optimierung wieder neue Optimierungsmöglichkeiten liefern kann
- Lokale Optimierung
  - Transformationen auf Grundblöcken
- Globale Optimierung
  - Transformationen über Grundblockgrenzen hinweg

Peephole Optimierung (1)

- Entfernen überflüssiger Anweisungen

```
(1)  MOV R0, a
(2)  MOV a, R0
```

```
(1)  MOV R0, a
```

falls (1) und (2) im gleichen Grundblock sind

- Algebraische Vereinfachungen

```
x := y + 0*(z**4/(y-1));
```

```
x := y;
```

```
x := x * 1;
x := x + 0;
```

} entfernen

Peephole Optimierung (2)

- Kontrollflussoptimierungen

```
(1)       goto L1                    (1)       goto L2
          ….                                   ….
(2) L1    goto L2                    (2) L1    goto L2
```

falls dann L1 nicht mehr erreichbar ist:
(2) entfernen (dead code elimination)

- Operatorreduktionen

```
x := y*8;        ⟶    x := y << 3;

x := y**2;       ⟶    x := y * y;
```

Lokale Optimierung

- common subexpression elimination

```
(1) a := b + c          (1) a := b + c
(2) b := a - d          (2) b := a - d
(3) c := b + c          (3) c := b + c
(4) d := a - d          (4) d := b
```

- variable renaming

```
t := b + c   ⟶   u := b + c
```

Normalform eines Grundblocks: jede Variable nur einmal definiert

- instruction interchange

```
t1 := b + c          t2 := x + y
t2 := x + y   ⟶      t1 := b + c
```

Globale Optimierung (1)

- passive code elimination
  - eine Anweisung, die **x** definiert, kann entfernt werden, falls **x** danach nicht mehr verwendet wird

- copy propagation

```
(1) x := t1                    (1) x := t1
(2) a[t2] := t3      ⟶        (2) a[t2] := t3
(3) a[t4] := x                 (3) a[t4] := t1
(4) goto L                     (4) goto L
```

wenn `t1` nicht mehr definiert wird, ist (1) passive code

Globale Optimierung (2)

- code motion

```
while (i <= limit*4+2)
{
  ....
}
```

```
t = limit*4+2;
while (i <= t)
{
  ....
}
```

wenn `limit` im Schleifenrumpf nicht verändert wird

- induced variables and operator reduction

```
    j := n
(1) j := j -1
(2) t4 := 4 * j
(3) t5 := a[t4]
(4) if t5 > v goto (1)
```

```
    j := n
    t4 := 4 * j
(1) j := j -1
(2) t4 := t4 - 4
(3) t5 := a[t4]
(4) if t5 > v goto (1)
```