Funktionale Programmierung

LVA 185.A03, VU 2.0, ECTS 3.0
WS 2020/2021

Vortrag III
Orientierung, Einordnung
27.10.2020

Jens Knoop

Technische Universität Wien
Information Systems Engineering
Compilers and Languages

Vortr. III
Teil III
Kap. 7
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Vortrag III
Orientierung, Einordnung

...zum selbstgeleiteten, eigenständigen Weiterlernen.

Teil III: Applikative Programmierung

- Kapitel 7: Rekursion
  $\rightsquigarrow$ Rekursionstypen, Aufrufgraphen, Komplexität,...
- Kapitel 8: Auswertung einfacher Ausdrücke
  $\rightsquigarrow$ Ausdrücke ohne/mit einfachen Fkt.-Termen,...
- Kapitel 9: Programmentwicklung, Programmverstehen
  $\rightsquigarrow$ Vorgehensrichtlinien zu Programmentw., Programmverst.

Vortr. III
Teil III
Kap. 7
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Teil III

Applikative Programmierung

Vortr. III
Teil III
Kap. 7
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Plakativ gesprochen:

Applikatives Programmieren ist Programmieren und Rechnen mit Funktionen über elementaren Werten (Zahlen, Zeichen, Wahrheitswerte,...):

```
type Nat0 = Int
fac :: Nat0 -> Nat0
fac n = prod [1..n]
```

```
pi = 3.14                                    -- Kreiszahl π
g  = 9.81                          -- Erdbeschleunigung in m/s2
type Pendellaenge     = Float                  -- in Metern
type Schwingungsdauer = Float                  -- in Sekunden
type Frequenz         = Float      -- in Anzahl pro Minute
```

```
fadenpendel :: Pendellaenge -> (Schwingungsdauer,Frequenz)
fadenpendel pl = (2*pi*sqr(pl/g),(1/(2*pi))*sqr(g/pl))
```

```
fac 5 ->> 120
fadenpendel 42.0 ->> (26.8869,0.0372)
```

Funktionen sind weder Argument noch Resultat v. Funktionen!

Vortr. III
Teil III
Kap. 7
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Applikatives Programmieren im strengen Sinn

...ist das

- Programmieren und Rechnen auf dem Niveau elementarwertiger Ausdrücke und Funktionen mit elementaren Werten als Argument und Resultat.
- Funktionen werden durch Abstraktion nach (unabhängig (variabel!) angesehenen) Ausdrucksoperanden gebildet.
- Funktionen werden auf Ausdrücke aus Konstanten, Variablen u. Funktionstermen elementaren Werts appliziert.

Summa summarum:

- Das tragende Prinzip applikativen Programmierens ist die Bildung von Funktionen durch Abstraktion v. Ausdrücken nach unabh. Variablen und die Applikation v. Funktionen auf elementare Werte mit elementarem Resultat; kurz: Das Rechnen mit elementaren Werten.

Wolfram-Manfred Lippe. Funktionale und Applikative Programmierung. eXamen.press, 2009, Kapitel 1.

Vortr. III
Teil III
Kap. 7
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Beispiel zur Funktionsabstraktion

Ausgangspunkt: (Viele) Ausdrücke mit gleicher Verknüpfungsstruktur:

```
(2*3.14 * sqr (1.0/9.81), (1/(2*3.14)) * sqr (9.81/1.0))
(2*3.14 * sqr (1.5/9.81), (1/(2*3.14)) * sqr (9.81/1.5))
(2*3.14 * sqr (2.0/9.81), (1/(2*3.14)) * sqr (9.81/2.0))
```

Abstraktion 1 nach variabel angesehenem grünen Ausdrucksoperand:

```
fadenpendel' :: Pendellaenge -> (Schwingungsdauer,Frequenz)
fadenpendel' pl = (2*3.14 * sqr (pl/9.81), (1/(2*3.14)) * sqr (9.81/pl))
```

Abstraktion 2 nach variabel anges. konstantwertigen Ausdrucksoperanden:

```
pi :: Float       -- 0-stellige Funktion, konstante Funktion
pi = 3.14
g :: Float        -- 0-stellige Funktion, konstante Funktion
g  = 9.81
fadenpendel:: Pendellaenge -> (Schwingungsdauer,Frequenz)
fadenpendel pl = (2*pi * sqr (pl/g), (1/(2*pi)) * sqr (g/pl))
```

Rechnen mit Funktionen: Applizieren von Funktionen auf Argumente:

```
fadenpendel' 42.0 ->> (26.8869,0.0372)
fadenpendel 42.0 ->> (26.8869,0.0372)
```

Vortr. III
Teil III
Kap. 7
Kap. 8
Kap. 9
Umgekehr
Klassen-
zim-
Hinweis
Aufgabe

Beispiel zur Funktionsabstraktion (fgs.)

Ausgangspunkt: Ausdrücke (wiederholt):

```
(2*3.14 * sqr (1.0/9.81), (1/(2*3.14)) * sqr (9.81/1.0))
(2*3.14 * sqr (1.5/9.81), (1/(2*3.14)) * sqr (9.81/1.5))
(2*3.14 * sqr (2.0/9.81), (1/(2*3.14)) * sqr (9.81/2.0))
```

Abstraktion nach allen als unabhängig angesehenen Ausdrucksoperanden:

```
fadenpendel'' :: (Pendellaenge,Kreiszahl,Erdbeschleunigung)
                                    -> (Schwingungsdauer,Frequenz)
fadenpendel'' (pl,pi,g) = (2*pi * sqr (pl/g), (1/(2*pi)) * sqr (g/pl))
```

Rechnen mit unterschiedlichen Pendellängen und unterschiedlich genauen Approximationen von Kreiszahl und Erdbeschleunigung:

```
fadenpendel'' (1.0,3.14,9.81) ->> ...          (g ∅-Wert)
fadenpendel'' (1.0,3.14159,9.78033) ->> ...    (g Äquator)
fadenpendel'' (1.0,3.14159265,9.83219) ->> ... (g Erdpole)
```

Beispiel zur Ausdrucksbildung

Zwei neu gebildete Ausdrücke (mit anschließender nicht ausgeführter Auswertung):

```
1) schwingungsdauer = first (fadenpendel (sqr (pi * g)))
2) frequenz = second (fadenpendel (sqr (pi * g)))

schwingungsdauer ->> ...
frequenz ->> ...
```

Vortr. III
Teil III
Kap. 7
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Kapitel 7

Rekursion

Vortr. III
Teil III
Kap. 7
7.1
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr
Klassen-
zim-
mer II
Hinweis
Aufgabe

Kapitel 7.1

Motivation

Vortr. III
Teil III
Kap. 7
7.1
7.1.1
7.1.2
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr
Klassen-
zim-
mer II
Hinweis
Aufgabe

Rekursion

...zentrales Mittel funktionaler Sprachen

- Wiederholungen auszudrücken (Beachte: In funktionalen Sprachen gibt es keine Anweisungen und deshalb auch keine Schleifen).

Rekursives Vorgehen

- führt oft auf sehr elegante Lösungen, die konzeptuell wesentlich einfacher und intuitiver sind als schleifenbasierte imperative Lösungen (Typische Beispiele: Quicksort, Türme von Hanoi).

Rekursion für fkt. Programmierung so wichtig, dass eine

- Klassifizierung von Rekursionstypen zweckmäßig ist.

...eine solche Klassifizierung nehmen wir in der Folge vor.

Vortr. III
Teil III
Kap. 7
7.1
7.1.1
7.1.2
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Quicksort, Türme von Hanoi

...zwei Beispiele, für die rekursives Vorgehen auf besonders

- intuitive, einfache und elegante Lösungen

führt:

1. Quicksort: Schnelles Sortieren.
2. Türme von Hanoi: Umschichten eines Turms aus 50 goldenen Scheiben nach bestimmten Regeln, womit nach einer hinterindischen Sage eine Gruppe von Mönchen seit dem Anbeginn der Zeit betraut ist.*

* Die Sage berichtet, dass das Ende der Welt gekommen ist, wenn die Mönche ihre Aufgabe vollendet haben.

Vortr. III
Teil III
Kap. 7
7.1
7.1.1
7.1.2
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Kapitel 7.1.1

Schnelles Sortieren, Quicksort

Vortr. III
Teil III
Kap. 7
7.1
7.1.1
7.1.2
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Quicksort

...bereits besprochen (s. Kap. 1.2.1):

```
quickSort :: [Integer] -> [Integer]
quickSort []     = []
quickSort (n:ns) =
 quickSort smaller   -- Alle Elemente m aus ns kleiner als n
                     -- rekursiv sortiert an den Anfang.
 ++ [n]              -- n selbst steht automatisch richtig
                     -- sortiert in der Mitte.
 ++ quickSort larger -- Alle Elemente m aus ns grösser als n
                     -- rekursiv sortiert ans Ende.
  where smaller = [m | m<-ns, m<=n]
        larger  = [m | m<-ns, m>n]
```

Vortr. III
Teil III
Kap. 7
7.1
7.1.1
7.1.2
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Kapitel 7.1.1

Türme von Hanoi

Vortr. III
Teil III
Kap. 7
7.1
7.1.1
7.1.2
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Türme von Hanoi

Aufgabe, Regelwerk:

Ausgangssituation:

Gegeben sind drei Stapelplätze A, B und C. Auf Platz A liegt ein Stapel paarweise verschieden großer goldener Scheiben, die mit von unten nach oben abnehmender Größe übereinander geschichtet sind.

Aufgabe:

Schichte den Scheibenstapel von Platz A auf Platz C um unter Ausnutzung von Platz B als Zwischenablage.

Regelwerk:

Bei jedem Umschichtungszug darf stets nur eine Scheibe bewegt werden; nie darf eine größere Scheibe oberhalb einer kleineren Scheibe auf einem der drei Plätze zu liegen kommen.

Vortr. III
Teil III
Kap. 7
7.1
7.1.1
7.1.2
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Veranschaulichung: Türme von Hanoi (1)

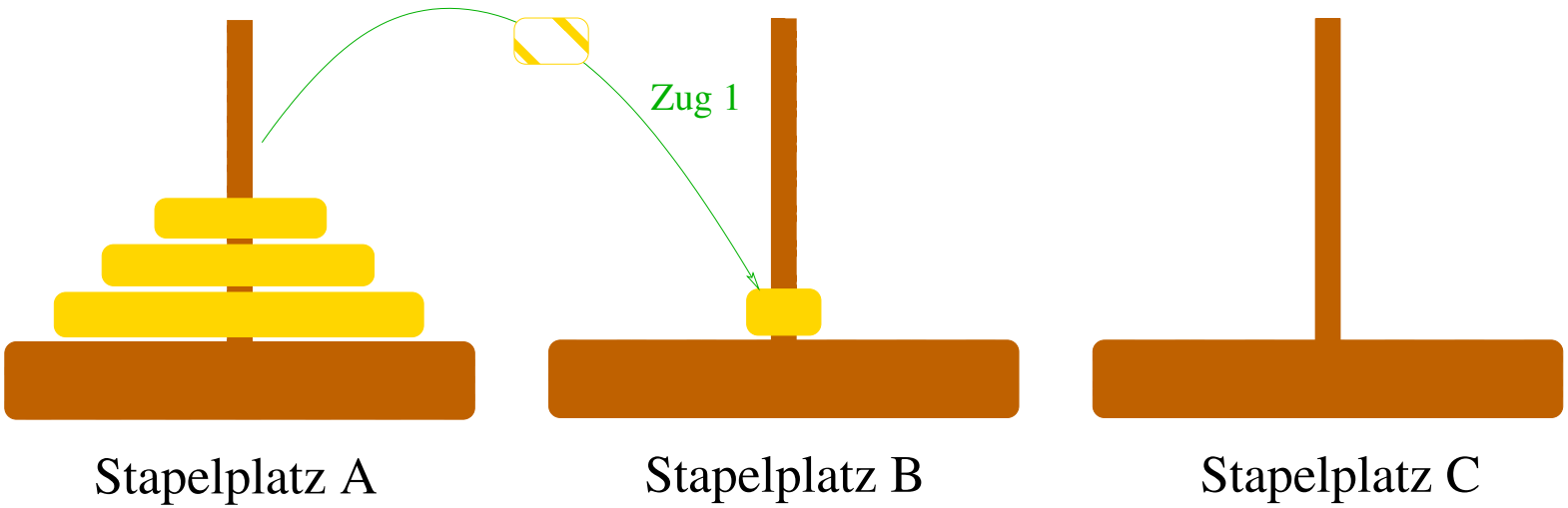
Ausgangssituation:

Ausgangsstapel Zwischenablage Zielstapel

Vortr. III
Teil III
Kap. 7
7.1
7.1.1
7.1.2
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Veranschaulichung: Türme von Hanoi (2)

Nach einem Zug:

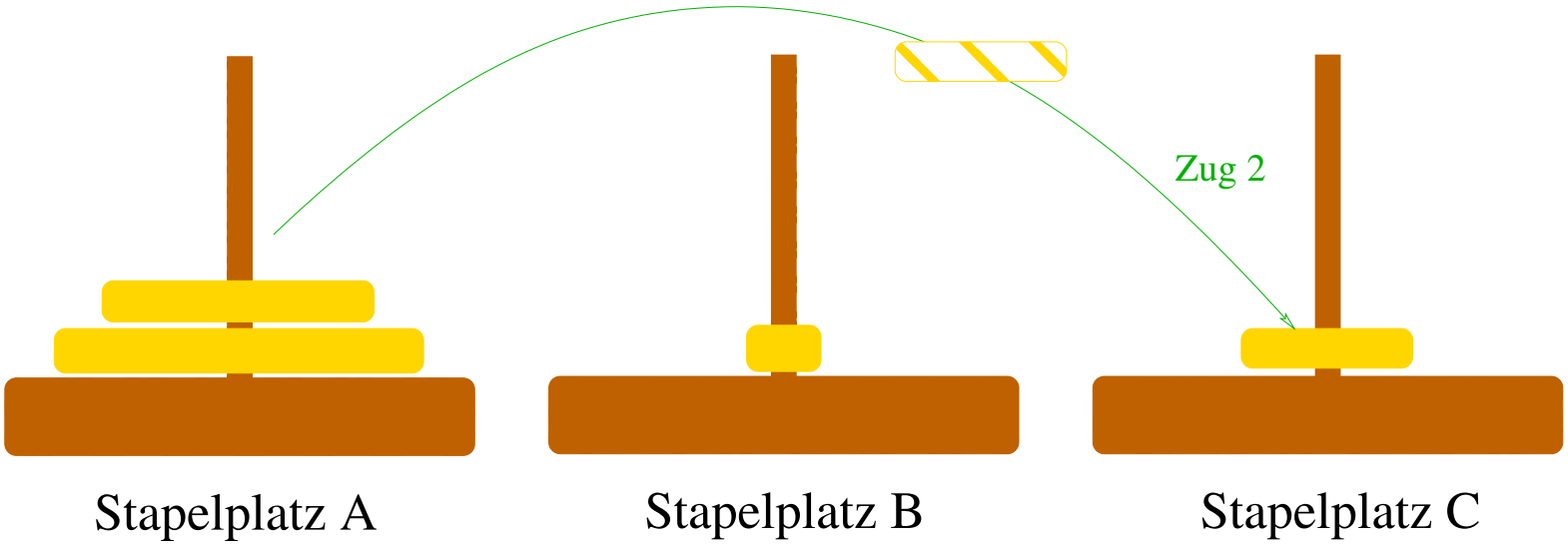


Ausgangsstapel Zwischenablage Zielstapel

Vortr. III
Teil III
Kap. 7
7.1
7.1.1
7.1.2
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Veranschaulichung: Türme von Hanoi (3)

Nach zwei Zügen:

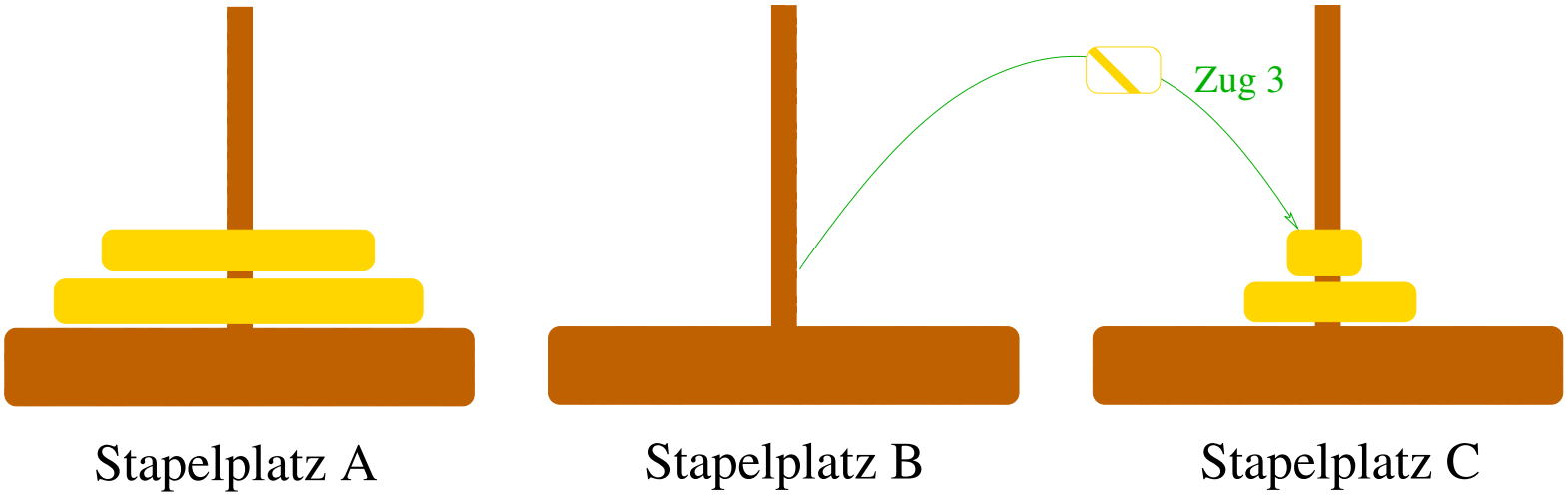


Ausgangsstapel Zwischenablage Zielstapel

Vortr. III
Teil III
Kap. 7
7.1
7.1.1
7.1.2
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Veranschaulichung: Türme von Hanoi (4)

Nach drei Zügen:

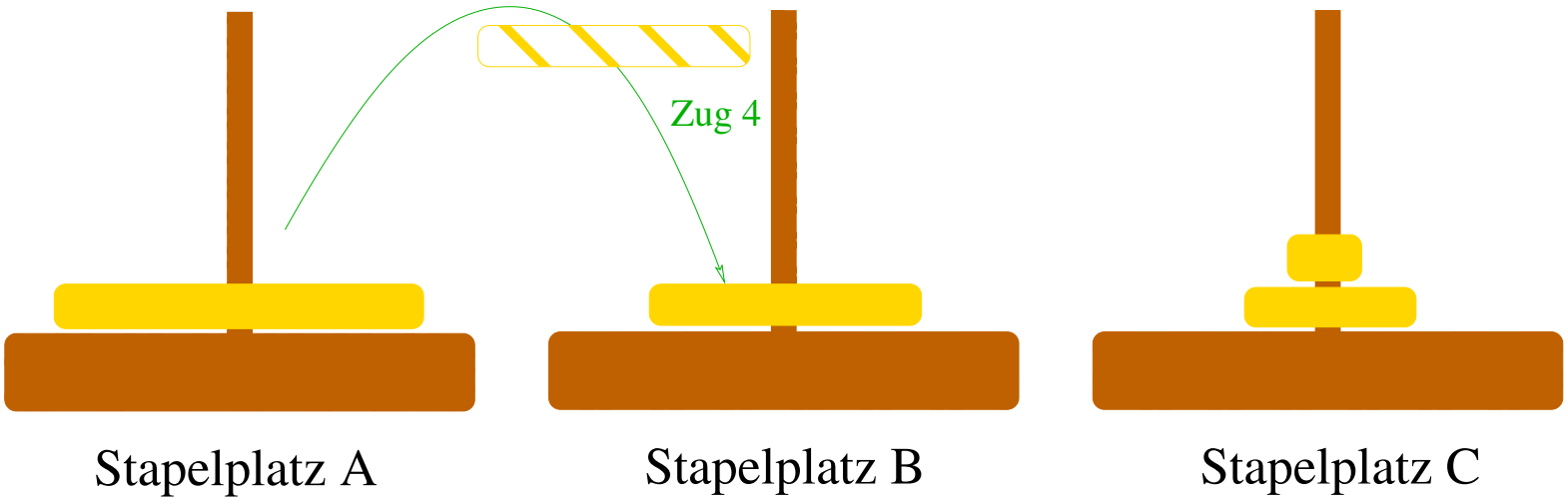


Ausgangsstapel Zwischenablage Zielstapel

Vortr. III
Teil III
Kap. 7
7.1
7.1.1
7.1.2
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Veranschaulichung: Türme von Hanoi (5)

Nach vier Zügen:

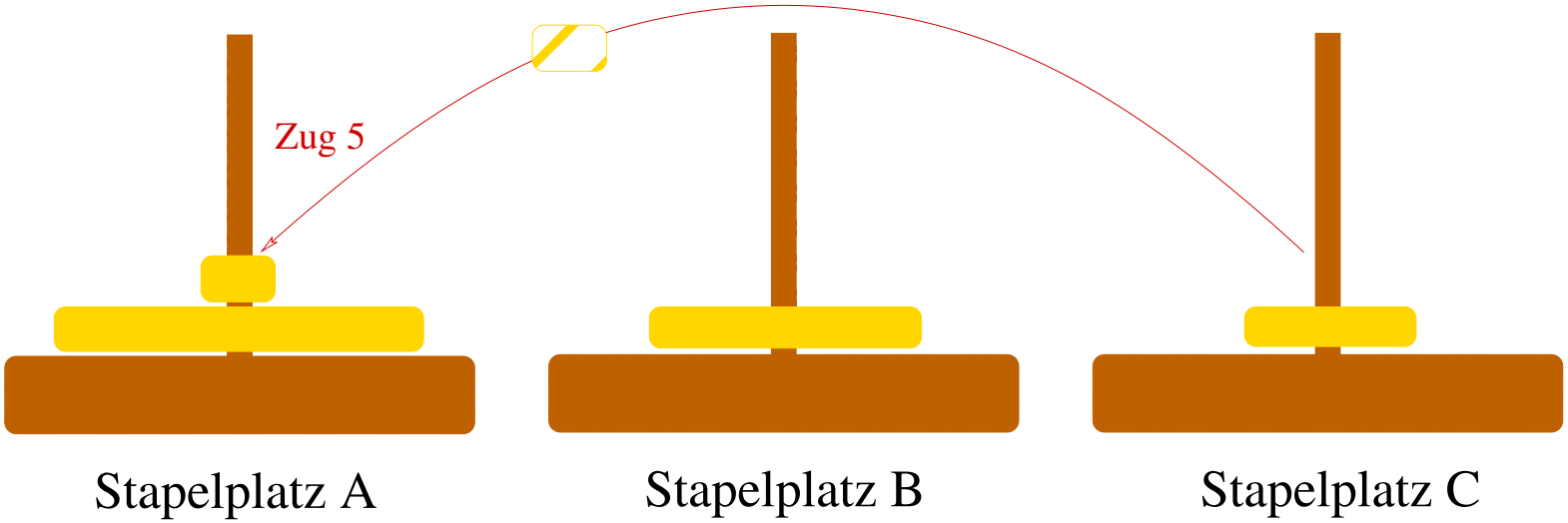


Ausgangsstapel Zwischenablage Zielstapel

Vortr. III
Teil III
Kap. 7
7.1
7.1.1
7.1.2
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Veranschaulichung: Türme von Hanoi (6)

Nach fünf Zügen:

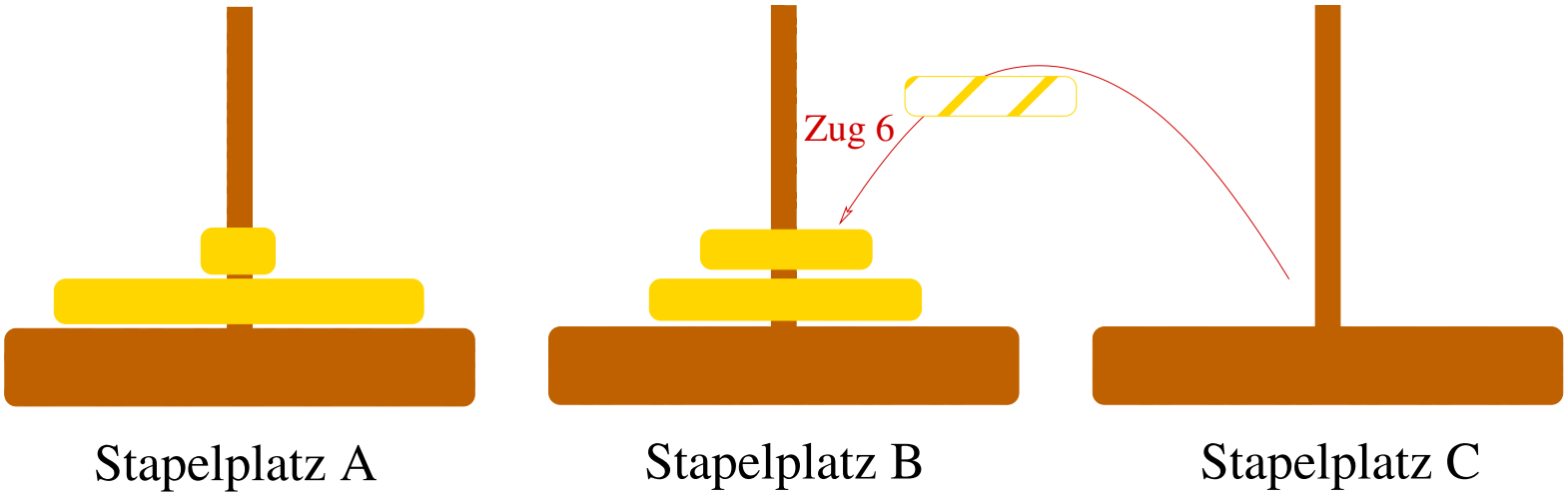


Ausgangsstapel Zwischenablage Zielstapel

Vortr. III
Teil III
Kap. 7
7.1
7.1.1
7.1.2
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Veranschaulichung: Türme von Hanoi (7)

Nach sechs Zügen:

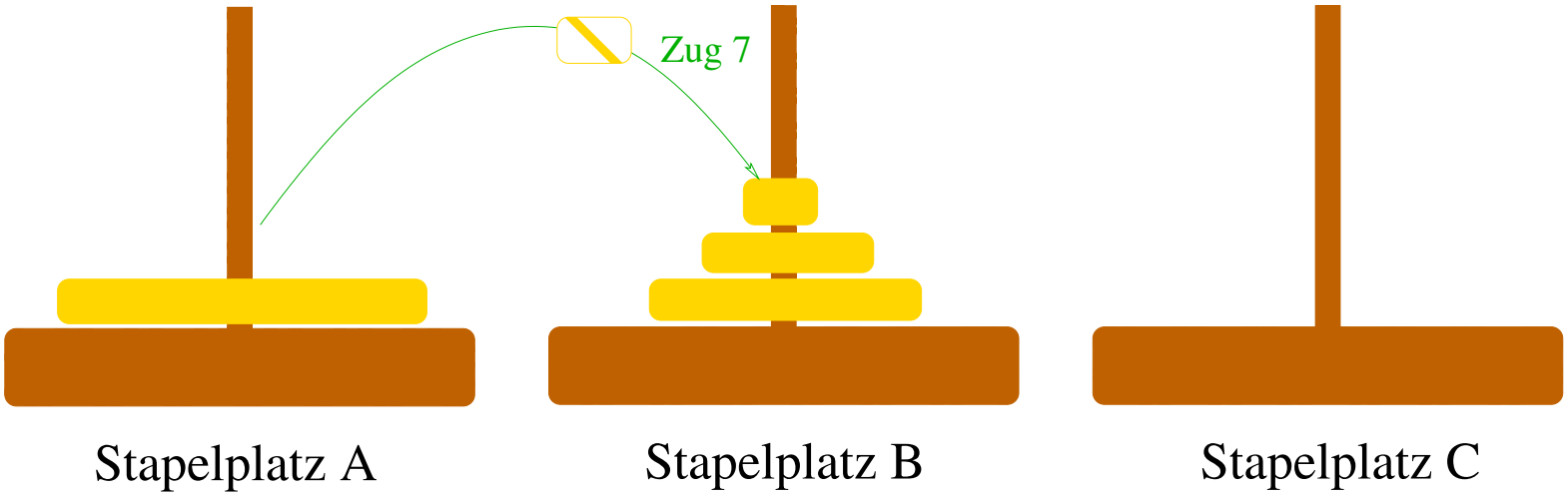


Ausgangsstapel Zwischenablage Zielstapel

Vortr. III
Teil III
Kap. 7
7.1
7.1.1
7.1.2
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Veranschaulichung: Türme von Hanoi (8)

Nach sieben Zügen:

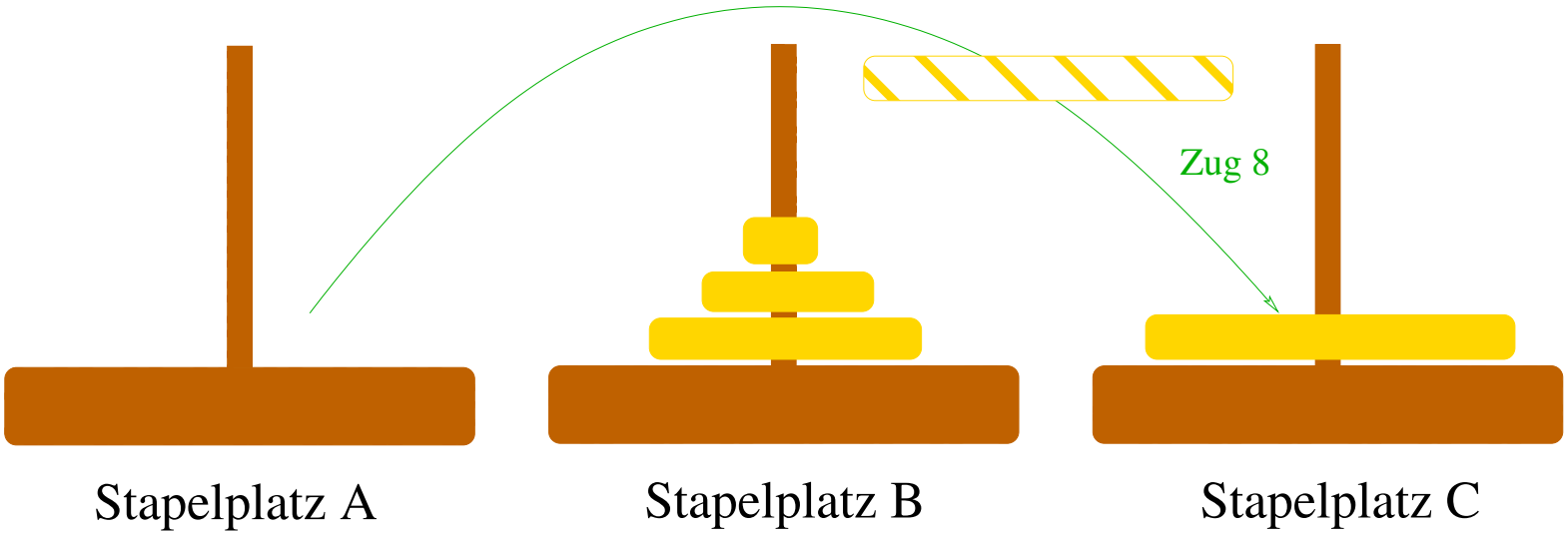


Ausgangsstapel Zwischenablage Zielstapel

Vortr. III
Teil III
Kap. 7
7.1
7.1.1
7.1.2
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Veranschaulichung: Türme von Hanoi (9)

Nach acht Zügen:

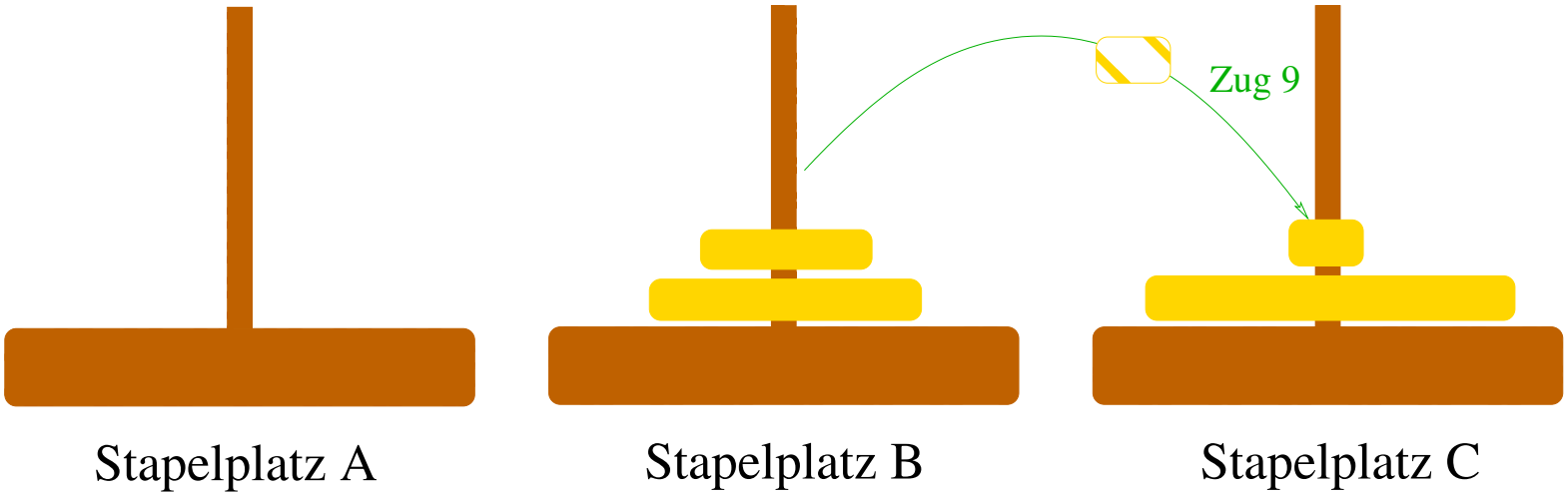


Ausgangsstapel Zwischenablage Zielstapel

Vortr. III
Teil III
Kap. 7
7.1
7.1.1
7.1.2
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Veranschaulichung: Türme von Hanoi (10)

Nach neun Zügen:

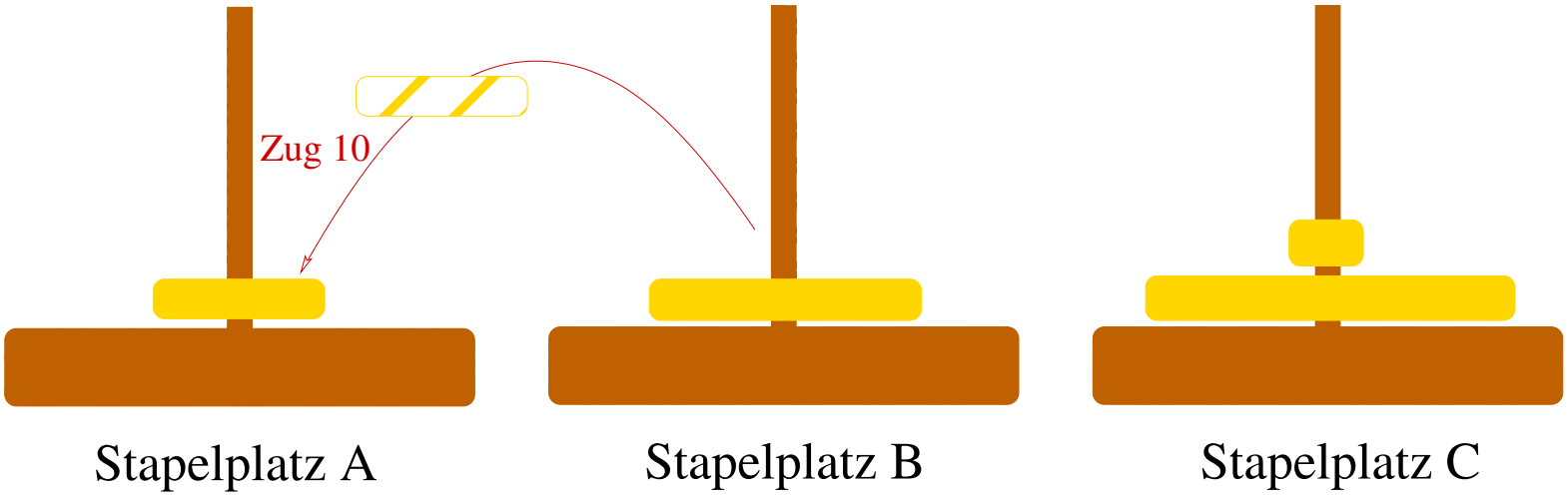


Ausgangsstapel Zwischenablage Zielstapel

Vortr. III
Teil III
Kap. 7
7.1
7.1.1
7.1.2
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Veranschaulichung: Türme von Hanoi (11)

Nach zehn Zügen:

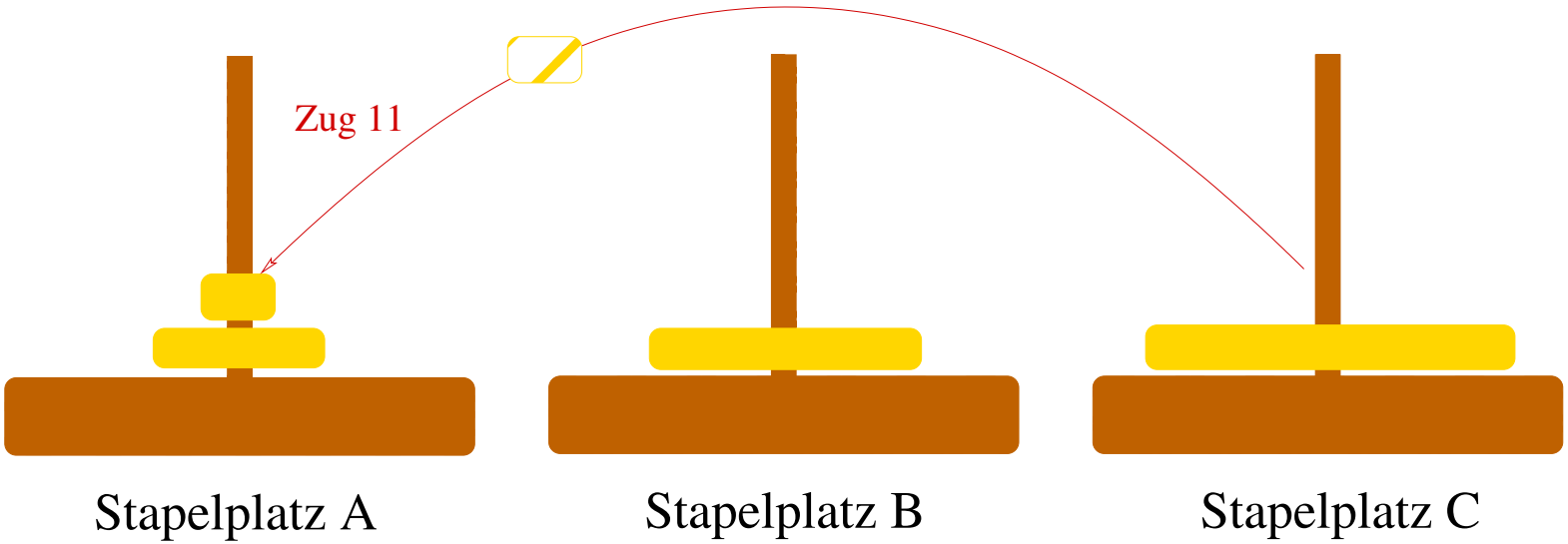


Ausgangsstapel Zwischenablage Zielstapel

Vortr. III
Teil III
Kap. 7
7.1
7.1.1
7.1.2
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Veranschaulichung: Türme von Hanoi (12)

Nach elf Zügen:

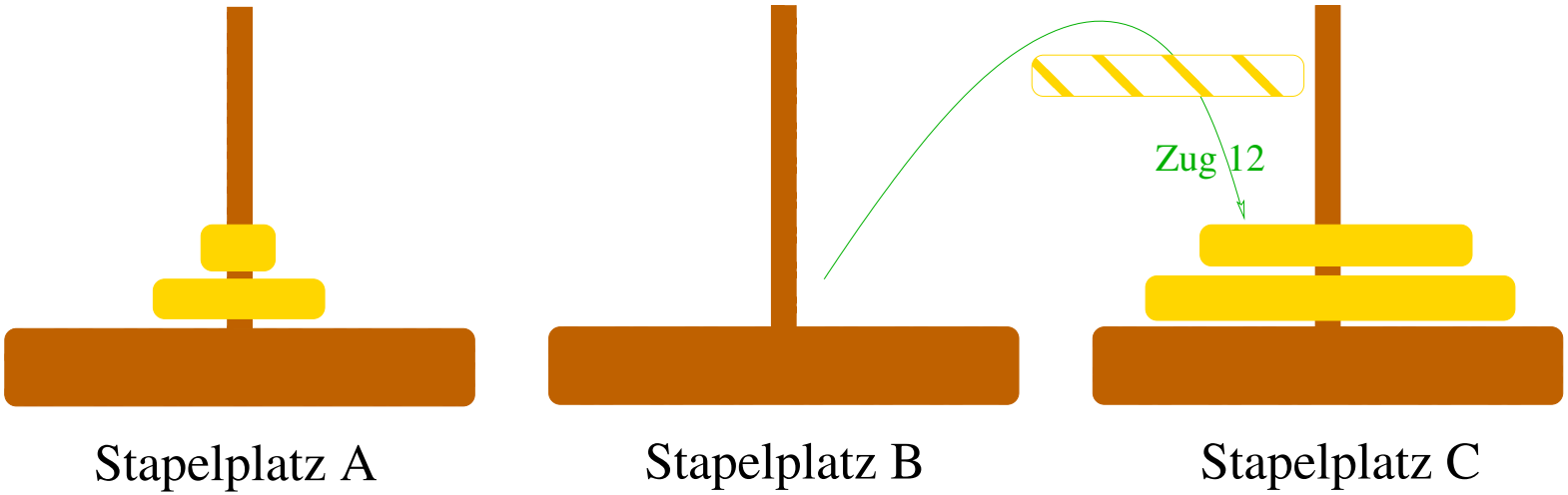


Ausgangsstapel Zwischenablage Zielstapel

Vortr. III
Teil III
Kap. 7
7.1
7.1.1
7.1.2
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Veranschaulichung: Türme von Hanoi (13)

Nach zwölf Zügen:

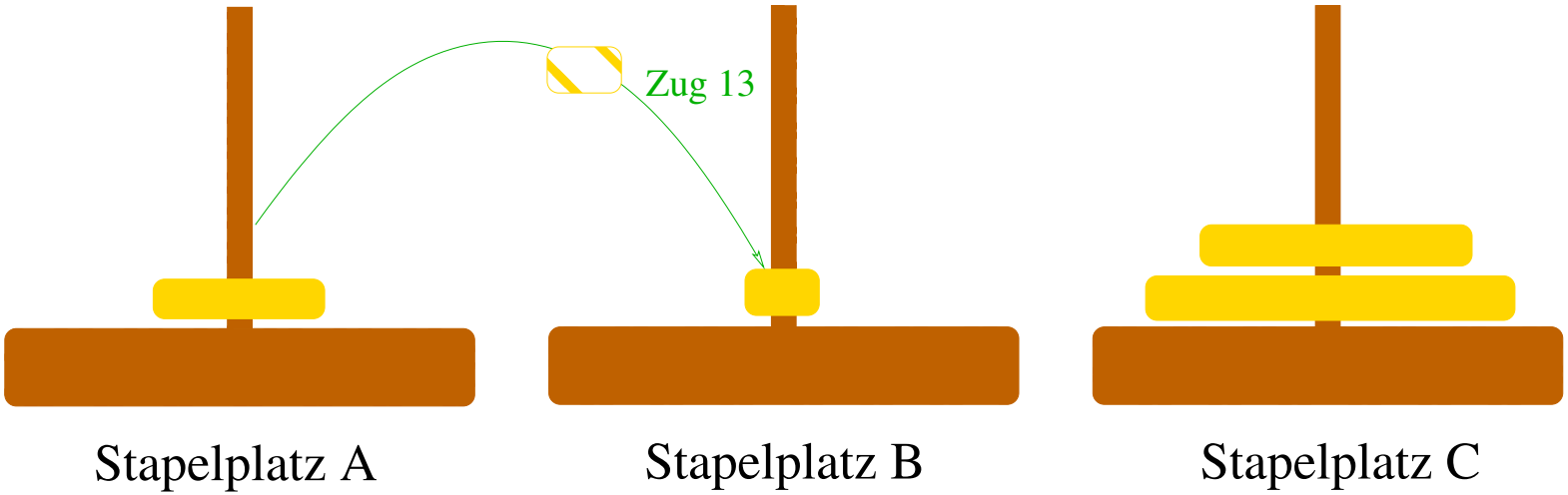


Ausgangsstapel Zwischenablage Zielstapel

Vortr. III
Teil III
Kap. 7
7.1
7.1.1
7.1.2
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Veranschaulichung: Türme von Hanoi (14)

Nach dreizehn Zügen:

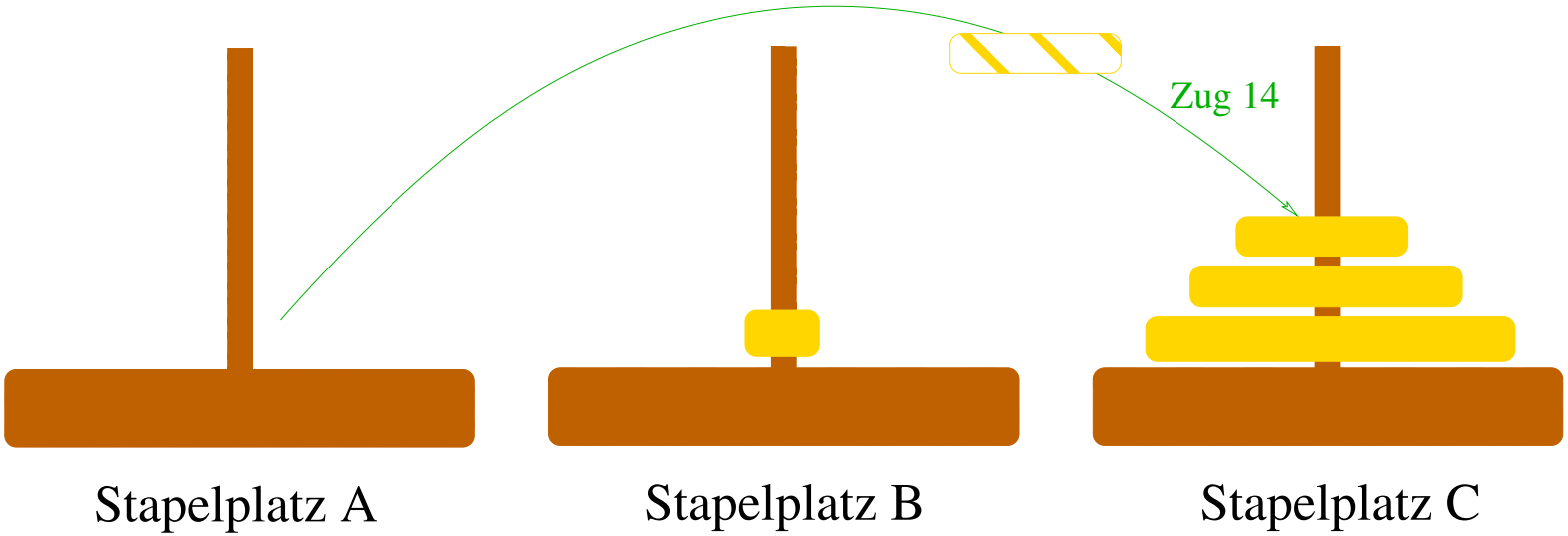


Ausgangsstapel Zwischenablage Zielstapel

Vortr. III
Teil III
Kap. 7
7.1
7.1.1
7.1.2
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Veranschaulichung: Türme von Hanoi (15)

Nach vierzehn Zügen:

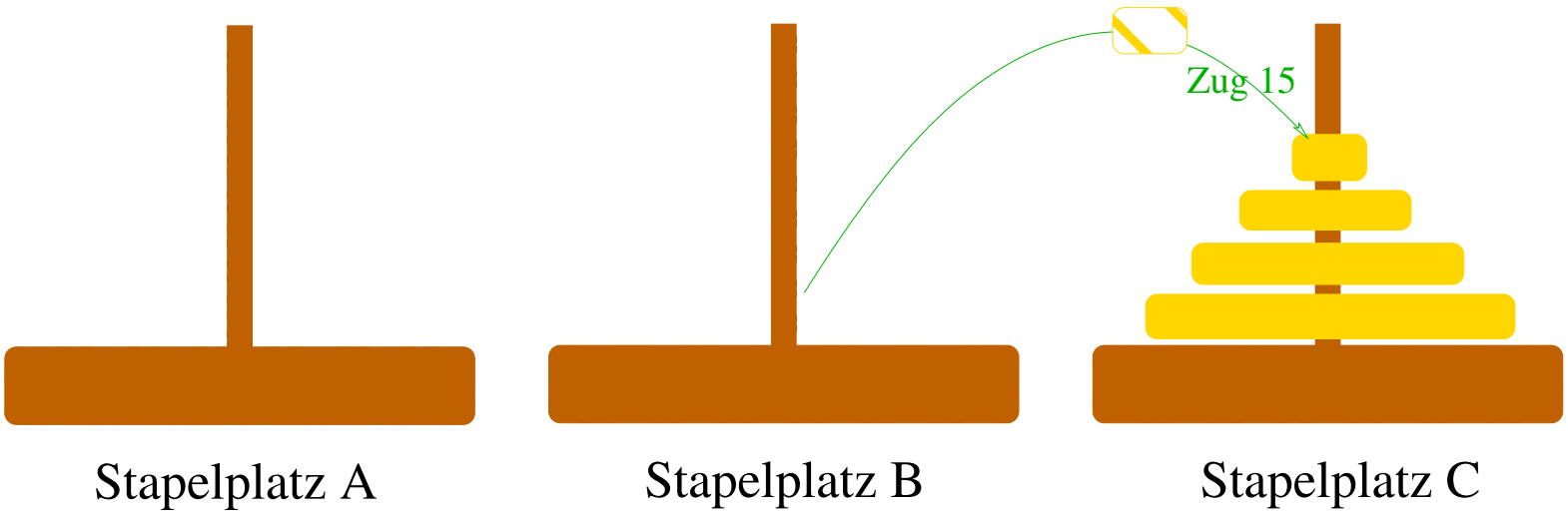


Ausgangsstapel Zwischenablage Zielstapel

Vortr. III
Teil III
Kap. 7
7.1
7.1.1
7.1.2
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Veranschaulichung: Türme von Hanoi (16)

Nach fünfzehn Zügen:



Ausgangsstapel Zwischenablage Zielstapel

Vortr. III
Teil III
Kap. 7
7.1
7.1.1
7.1.2
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Veranschaulichung der Rekursionsidee (1)

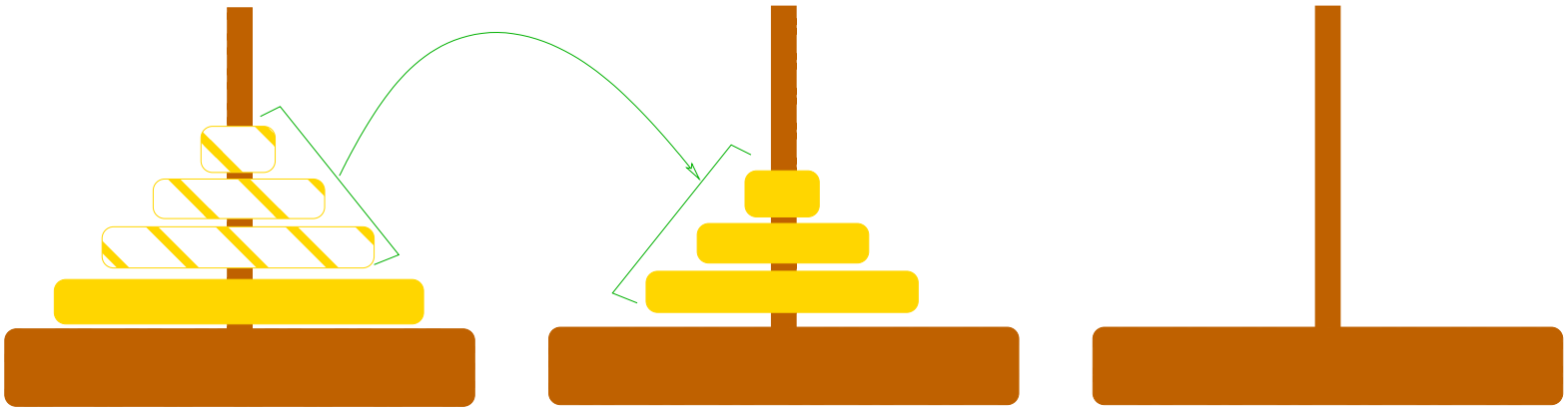
Aufgabe: Schichte Turm $[1, 2, \ldots, N]$ von Ausgangsstapel A auf Zielstapel C um unter Verwendung von Stapelplatz B als Zwischenablage.

Ausgangsstapel Zwischenablage Zielstapel

Vortr. III
Teil III
Kap. 7
7.1
7.1.1
7.1.2
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Veranschaulichung der Rekursionsidee (2)

Schritt 1 – Scheibe N freispielen:
Schichte Turm $[1, 2, \ldots, N-1]$ (rekursiv) von Ausgangsstapel A auf Zwischenablage B als temporärer Zielstapel unter Ausnutzung von Stapelplatz C als temporäre Zwischenablage um.

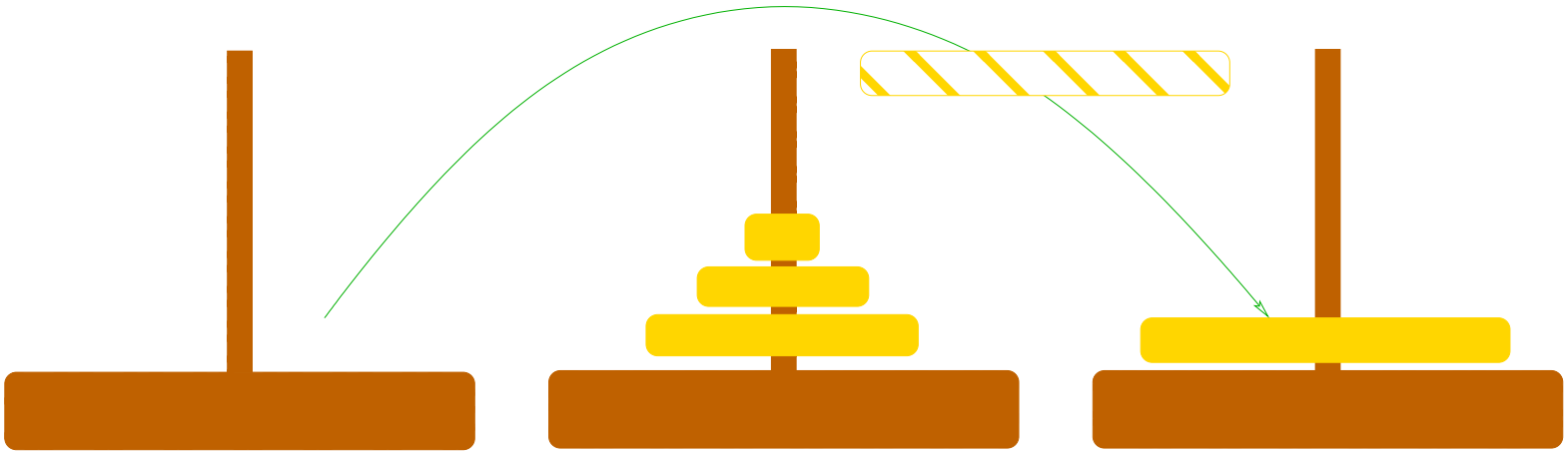


Ausgangsstapel Zielstapel Zwischenablage

Vortr. III
Teil III
Kap. 7
7.1
7.1.1
7.1.2
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Veranschaulichung der Rekursionsidee (3)

Schritt 2 – Scheibe N spielen:
Verschiebe Scheibe N von Ausgangsstapel A auf Zielstapel C.

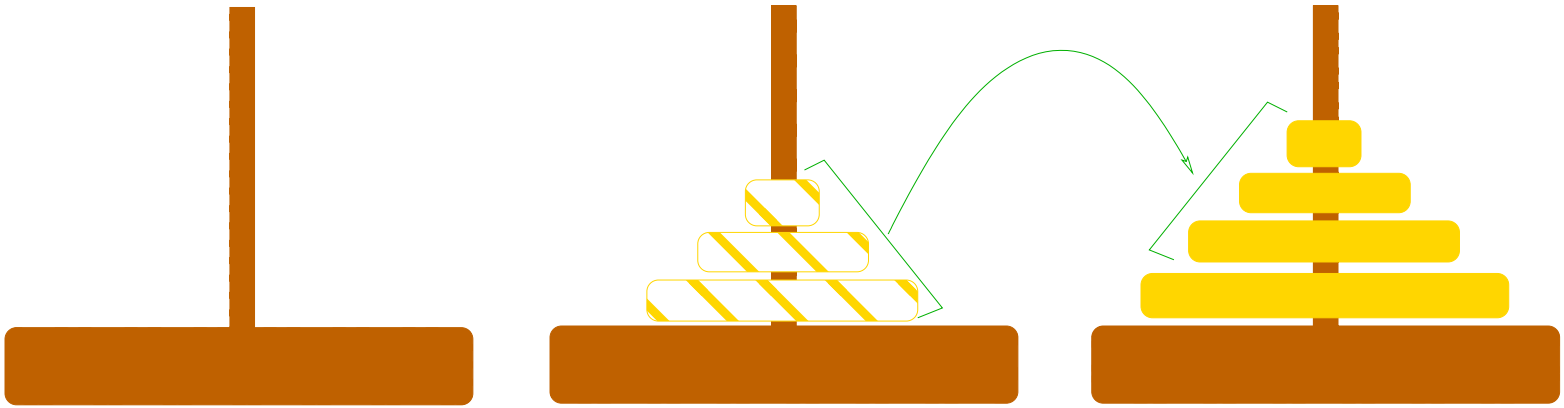


Ausgangsstapel Zwischenablage Zielstapel

Vortr. III
Teil III
Kap. 7
7.1
7.1.1
7.1.2
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Veranschaulichung der Rekursionsidee (4)

Schritt 3 – Verbleibenden Turm umschichten:
Schichte Turm $[1, 2, \ldots, N-1]$ (rekursiv) von Stapel B als temporärer Ausgangsstapel auf Zielstapel C unter Ausnutzung von Stapelplatz A als temporärer Zwischenablage um.



Zwischenablage Ausgangsstapel Zielstapel

Beachte: A, B und C tauschen für Schritt 3 gegenüber Schritt 1 ihre Rollen als Ausgangs-, Ziel- und Hilfsstapelplatz.

Vortr. III
Teil III
Kap. 7
7.1
7.1.1
7.1.2
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Zusammenfassung der Rekursionsidee

...für das Problem der Türme von Hanoi:

Der Turm $[1, 2, \ldots, N-1, N]$ mit N Scheiben, in zunehmender Größe benannt von 1 bis N, wird von Ausgangsstapel A auf Zielstapel C unter Zuhilfenahme von Stapelplatz B als Zwischenablage wie folgt umgeschichtet:

1) Schichte den Turm $[1, 2, \ldots, N-1]$ mit $(N-1)$ Scheiben rekursiv von Platz A auf Platz B unter Zuhilfenahme von Platz C als Zwischenablage um.
2) Verschiebe (die dadurch jetzt frei liegende unterste und größte) Scheibe N von Platz A auf Platz C.
3) Schichte den Turm $[1, 2, \ldots, N-1]$ mit $(N-1)$ Scheiben rekursiv von Platz B auf Platz C unter Zuhilfenahme von Platz A als Zwischenablage um.

Vortr. III
Teil III
Kap. 7
7.1
7.1.1
7.1.2
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Türme von Hanoi: Modellierung in Haskell

Modellierung:

```
type Turmhoehe = Int      -- Anzahl Scheiben
type Scheibe   = Int      -- Scheibenidentifikator

data Stapel    = A | B | C deriving Show

type A_Stapel  = Stapel   -- Ausgangsstapel A
type Z_Stapel  = Stapel   -- Zielstapel Z
type H_Stapel  = Stapel   -- Hilfsstapel H
type Von       = Stapel   -- Stapelbezeichner
type Nach      = Stapel   -- Stapelbezeichner

hanoi :: Turmhoehe -> A_Stapel -> Z_Stapel -> H_Stapel
                                  -> [(Scheibe,Von,Nach)]

hanoi n a z h = <Code, der einen n-scheibigen Turm
                 vom Platz a auf den Platz z um-
                 schichtet unter Ausnutzung von
                 Hilfplatz h als Zwischenablage>
```

Vortr. III
Teil III
Kap. 7
7.1
7.1.1
7.1.2
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Türme von Hanoi: Implementierung in Haskell

Implementierung:

```
hanoi :: Turmhoehe -> A_Stapel -> Z_Stapel -> H_Stapel
                                    -> [(Scheibe,Von,Nach)]
hanoi n a z h
 | n == 0    = [] -- Fertig, Turm ist umgeschichtet!
 | otherwise =
     {- Schritt 1: Verschiebe (n-1)-Turm rek. von
        Platz a auf Platz h über z als Hilfsplatz -}
    (hanoi (n-1) a h z)
     {- Schritt 2: Verschiebe Scheibe n von Platz a
        auf Platz z -}
    ++ [(n,a,z)]
     {- Schritt 3: Verschiebe (n-1)-Turm rek. von
        Platz h nach Platz z über a als Hilfsplatz -}
    ++ (hanoi (n-1) h z a)
```

Vortr. III
Teil III
Kap. 7
7.1
7.1.1
7.1.2
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Türme von Hanoi: Aufrufe der Funktion hanoi

```
Main>hanoi 1 A C B
[(1,A,C)]                                  (1 Zug für Schritt 2)

Main>hanoi 2 A C B
[(1,A,B),                                  (1 Zug für Schritt 1)
 (2,A,C),                                  (1 Zug für Schritt 2)
 (1,B,C)]                                  (1 Zug für Schritt 3)

Main>hanoi 3 A C B
[(1,A,C),(2,A,B),(1,C,B),                  (3 Züge für Schritt 1)
 (3,A,C),                                  (1 Zug für Schritt 2)
 (1,B,A),(2,B,C),(1,A,C)]                  (3 Züge für Schritt 1)

Main>hanoi 4 A C B
[(1,A,B),(2,A,C),(1,B,C),                  (7 Züge für Schritt 1)
 (3,A,B),(1,C,A),(2,C,B),(1,A,B),
 (4,A,C),                                  (1 Zug für Schritt 2)
 (1,B,C),(2,B,A),(1,C,A),                  (7 Züge für Schritt 3)
 (3,B,C),(1,A,B),(2,A,C),(1,B,C)]
```

Vortr. III
Teil III
Kap. 7
7.1
7.1.1
7.1.2
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Kapitel 7.2

Rekursionstypen

Vortr. III
Teil III
Kap. 7
7.1
7.2
7.2.1
7.2.2
7.2.3
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr
Klassen-
zim-
mer II
Hinweis
Aufgabe

Klassifikation der Rekursionstypen

Eine Rechenvorschrift heißt rekursiv, wenn sie

- in ihrem Rumpf (direkt oder indirekt) aufgerufen wird.

Wir unterscheiden zwischen Rekursion auf

- mikroskopischer Ebene
  ...betrachtet einzelne Rechenvorschriften und die syntaktische Gestalt der rekursiven Aufrufe.
- makroskopischer Ebene
  ...betrachtet Systeme von Rechenvorschriften und ihre wechselseitigen Aufrufe.

Vortr. III
Teil III
Kap. 7
7.1
7.2
7.2.1
7.2.2
7.2.3
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Kapitel 7.2.1

Mikroskopische Ebene

Vortr. III
Teil III
Kap. 7
7.1
7.2
7.2.1
7.2.2
7.2.3
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Rekursion auf mikroskopischer Ebene (1)

...folgende Unterscheidungen und Sprechweisen sind üblich:

1. Repetitive (schlichte, endständige) Rekursion
   $\rightsquigarrow$ pro Zweig höchstens ein rekursiver Aufruf und diesen stets als äußerste Operation.

   Beispiel:

```
ggt :: Integer -> Integer -> Integer
ggt m n
  | n == 0 = m                    -- Zweig 1
  | m >= n = ggt (m-n) n          -- Zweig 2
  | m < n  = ggt (n-m) m          -- Zweig 3
```

Vortr. III
Teil III
Kap. 7
7.1
7.2
7.2.1
7.2.2
7.2.3
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Rekursion auf mikroskopischer Ebene (2)

2. Lineare Rekursion

   $\rightsquigarrow$ pro Zweig höchstens ein rekursiver Aufruf, davon mindestens einer nicht als äußerste Operation.

   Beispiel:

```
powerThree :: Integer -> Integer
powerThree n
  | n == 0 = 1                          -- Zweig 1
  | n > 0  = 3 * powerThree (n-1)       -- Zweig 2
```

   Beachte: In Zweig 2, $n > 0$, ist "$*$" die äußerste Operation, nicht `powerThree`!

Vortr. III
Teil III
Kap. 7
7.1
7.2
7.2.1
7.2.2
7.2.3
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Rekursion auf mikroskopischer Ebene (3)

3. Baumartige (kaskadenartige) Rekursion
   $\rightsquigarrow$ pro Zweig können mehrere rekursive Aufrufe nebeneinander vorkommen.

   Beispiel:

```
binom :: Integer -> Integer -> Integer
binom n k
 | k == 0 || n == k = 1                    -- Zweig 1
 | otherwise =
    binom (n-1) (k-1) + binom (n-1) k -- Zweig 2
```

Vortr. III
Teil III
Kap. 7
7.1
7.2
7.2.1
7.2.2
7.2.3
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Rekursion auf mikroskopischer Ebene (4)

4. Geschachtelte Rekursion
   $\rightsquigarrow$ rekursive Aufrufe enthalten rekursive Aufrufe als Argumente.

   Beispiel:

```
fun91 :: Integer -> Integer
fun91 n
  | n >  100 = n - 10
  | n <= 100 = fun91 (fun91 (n+11))
```

   Übungsaufgabe: Warum heißt die Funktion wohl `fun91`?

Vortr. III
Teil III
Kap. 7
7.1
7.2
7.2.1
7.2.2
7.2.3
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Zusammenfassung

...auf mikroskopischer Ebene sprechen wir von

- direkter (oder unmittelbarer) Rekursion

und unterscheiden genauer:

- Repetitive (oder schlichte oder endständige) Rekursion
- Lineare Rekursion
- Baumartige (oder kaskadenartige) Rekursion
- Geschachtelte Rekursion

Vortr. III
Teil III
Kap. 7
7.1
7.2
7.2.1
7.2.2
7.2.3
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Kapitel 7.2.2

Makroskopische Ebene

Vortr. III
Teil III
Kap. 7
7.1
7.2
7.2.1
7.2.2
7.2.3
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Rekursion auf makroskopischer Ebene

...folgende Sprechweise ist üblich:

- Indirekte (verschränkte, wechselweise) Rekursion
  $\rightsquigarrow$ zwei oder mehr Funktionen rufen sich wechselweise auf.

Beispiel:

```
isOdd :: Integer -> Bool
isOdd n
 | n == 0 = False
 | n > 0  = isEven (n-1)

isEven :: Integer -> Bool
isEven n
 | n == 0 = True
 | n > 0  = isOdd (n-1)
```

Vortr. III
Teil III
Kap. 7
7.1
7.2
7.2.1
7.2.2
7.2.3
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Kapitel 7.2.3

Eleganz und Effizienz, Effizienzfallen

Vortr. III
Teil III
Kap. 7
7.1
7.2
7.2.1
7.2.2
7.2.3
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Eleganz, Effizienz, Effizienzfallen

Viele Probleme sind rekursiv besonders

- elegant lösbar (z.B. Quicksort, Türme von Hanoi)
- jedoch nicht immer unmittelbar effizient ($\neq$ effektiv!)
  (z.B. die naive Berechnung der Fibonacci-Zahlen)
  - Gefahr: (Unnötige) Mehrfachberechnungen
  - Besonders anfällig: Baum-/kaskadenartige Rekursion

Aus Implementierungssicht ist (s.a. Anhang E)

- repetitive Rekursion am (kosten-) günstigsten.
- geschachtelte Rekursion am ungünstigsten.

Vortr. III
Teil III
Kap. 7
7.1
7.2
7.2.1
7.2.2
7.2.3
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Die Folge der Fibonacci-Zahlen

...die unendliche Folge der Fibonacci-Zahlen $(f_i)_{i \in \mathbb{N}_0}$ ist folgendermaßen definiert:

$$f_0 = 0, \quad f_1 = 1 \quad \text{und} \quad f_n = f_{n-2} + f_{n-1} \quad \text{für alle } n \geq 2$$

Anfang der Folge der Fibonacci-Zahlen:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, \ldots$$

Vortr. III
Teil III
Kap. 7
7.1
7.2
7.2.1
7.2.2
7.2.3
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Naive Implementierung der Fibonacci-Fkt. (1)

Die naheliegende, unmittelbar an die Definition angelehnte naive Implementierung mit baumartiger Rekursion zur Berechnung der Fibonacci-Zahlen:

```
fib :: Integer -> Integer
fib n
  | n == 0      = 0
  | n == 1      = 1
  | otherwise   = fib (n-2) + fib (n-1)
```

...ist sehr, seeehr laaangsaaaaaaaaam (ausprobieren!)

Vortr. III
Teil III
Kap. 7
7.1
7.2
7.2.1
7.2.2
7.2.3
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Naive Implementierung der Fibonacci-Fkt. (2)

Veranschaulichung der durch die Mehrfachberechnung von Werten verursachten Ineffizienz durch manuelle Auswertung:

```
fib 0 ->> 0                          --  1 Aufrufe von fib

fib 1 ->> 1                          --  1 Aufrufe von fib

fib 2 ->> fib 0 + fib 1
      ->> 0 + 1
      ->> 1                          --  3 Aufrufe von fib

fib 3 ->> fib 1 + fib 2
      ->> 1 + (fib 1 + fib 0)
      ->> 1 + (1 + 0)
      ->> 2                          --  5 Aufrufe von fib
```

Vortr. III
Teil III
Kap. 7
7.1
7.2
7.2.1
7.2.2
7.2.3
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Naive Implementierung der Fibonacci-Fkt. (3)

```
fib 4 ->> fib 2 + fib 3
      ->> (fib 0 + fib 1) + (fib 1 + fib 2)
      ->> (0 + 1) + (1 + (fib 0 + fib 1))
      ->> (0 + 1) + (1 + (0 + 1))
      ->> 3                          --  9 Aufrufe von fib

fib 5 ->> fib 3 + fib 4
      ->> (fib 1 + fib 2) + (fib 2 + fib 3)
      ->> (1 + (fib 0 + fib 1)) +
             ((fib 0 + fib 1) + (fib 1 + fib 2))
      ->> (1 + (0 + 1)) +
             ((0 + 1) + (1 + (fib 0 + fib 1))
      ->> (1 + (0 + 1)) + ((0 + 1) + (1 + (0 + 1)))
      ->> 5                          -- 15 Aufrufe von fib
```

Vortr. III
Teil III
Kap. 7
7.1
7.2
7.2.1
7.2.2
7.2.3
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Naive Implementierung der Fibonacci-Fkt. (4)

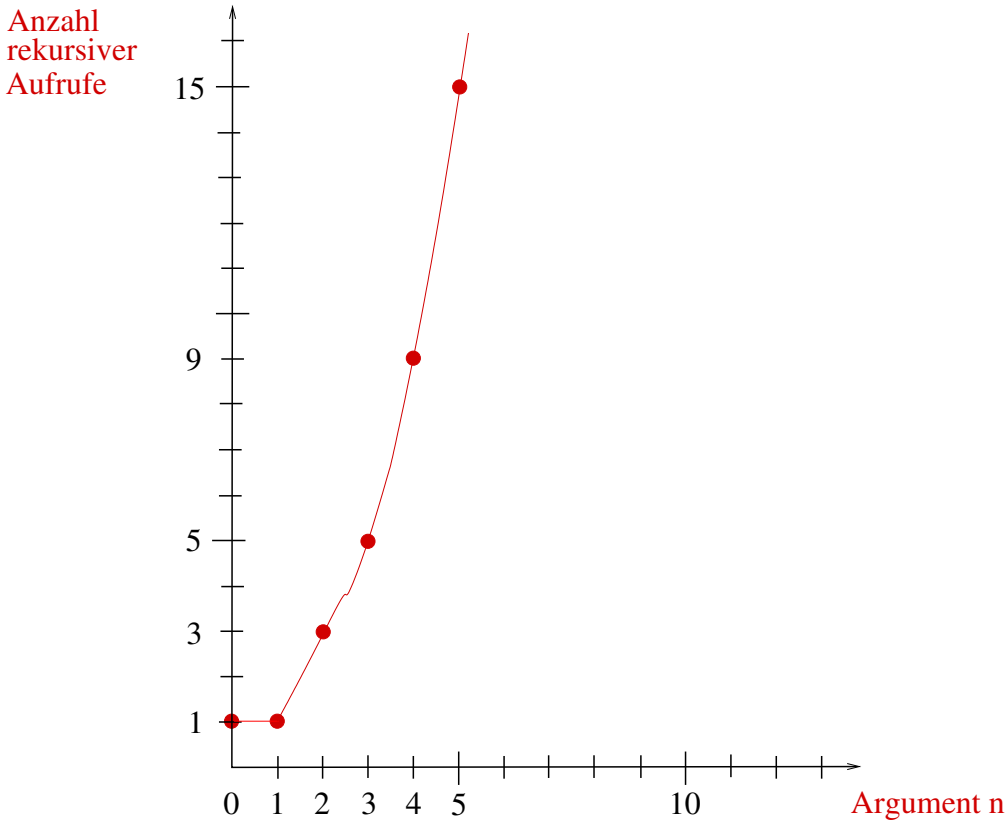
```
fib 8 ->> fib 6 + fib 7
      ->> + (fib 4 + fib 5) + (fib 5 + fib 6)
      ->> ((fib 2 + fib 3) + (fib 3 + fib 4))
          + ((fib 3 + fib 4) + (fib 4 + fib 5))
      ->> (((fib 0 + fib 1) + (fib 1 + fib 2))
           + (fib 1 + fib 2) + (fib 2 + fib 3)))
          + (((fib 1 + fib 2) + (fib 2 + fib 3))
           + ((fib 2 + fib 3) + (fib 3 + fib 4)))
      ->> ...
      ->> 21                          -- 60 Aufrufe von fib
```

Vortr. III
Teil III
Kap. 7
7.1
7.2
7.2.1
7.2.2
7.2.3
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Naive Implementierung der Fibonacci-Fkt. (5)

...die naive baumartig-rekursive Berechnung der Fibonacci-Zahlen führt zu äußerst vielen Mehrfachberechnungen.

Der Berechnungsaufwand wächst dabei exponentiell!



Vortr. III
Teil III
Kap. 7
7.1
7.2
7.2.1
7.2.2
7.2.3
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr
Klassen-
zim-
mer II
Hinweis
Aufgabe

Lsg. 1: Effiziente Berechnung der Fibonacci-Z.

Fibonacci-Zahlen lassen sich auf viele Arten effizient berechnen, z.B. mit der Idee des

- Rechnens auf Parameterposition!

Das Ergebnis wird hierbei in einem oder verteilt über mehrere zusätzliche (Akkumulations-) Parameter (hier zwei!) sukzessive akkumuliert:

```
fib :: Integer -> Integer
fib n = fib' n 0 1
 where
  fib' :: Integer -> Integer -> Integer -> Integer
  fib' 0 a b = a
  fib' n a b = fib' (n-1) b (a+b)
```

Beachte: Das Bsp. zeigt zugleich, wie eine baumartige Rekursion (in `fib` 'naiv') auf eine schlichte Rekursion (in `fib'`) zurückgeführt wird.

Vortr. III
Teil III
Kap. 7
7.1
7.2
7.2.1
7.2.2
7.2.3
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Lsg. 2: Effiziente Berechnung der Fibonacci-Z.

...die i.w. gleiche Idee verteilt auf mehrere Funktionen realisiert das System von Rechenvorschriften aus:

- `fibSchritt`, `fibPaar` und `fib`.

```
fibSchritt :: (Integer,Integer) -> (Integer,Integer)
fibSchritt (m,n) = (n,m+n)

fibPaar :: Integer -> (Integer,Integer)
fibPaar n
  | n == 0     = (0,1)
  | otherwise = fibSchritt (fibPaar (n-1))

fib :: Integer -> Integer
fib n = fst (fibPaar n)
```

Vortr. III
Teil III
Kap. 7
7.1
7.2
7.2.1
7.2.2
7.2.3
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Lsg. 3: Effiziente Berechnung der Fibonacci-Z.

...mit der Idee:

– Speichern und wiederbenutzen statt (neu) Berechnen

mithilfe sog. Memo-Funktionen; eine Idee, die zurückgeht auf:

Donald Michie. 'Memo' Functions and Machine Learning. Nature 218:19-22, 1968.

wobei hier die Memo-Funktion als (Memo-) Liste realisiert ist:

```
memo_fib :: [Int]
memo_fib = [fib' n | n <- [0..]]          -- Memo-Liste

fib' :: Int -> Int
fib' 0 = 0
fib' 1 = 1
fib' n = memo_fib !! (n-2) + memo_fib !! (n-1)

fib :: Int -> Int
fib n = memo_fib!!n
```

Hinweis: Die Listenelementzugriffsfunktion `(!!)` hat Typ `(!!) :: [a] -> Int -> a`; deshalb ist `fib` hier für `Int` definiert, nicht für `Integer`.

Vortr. III
Teil III
Kap. 7
7.1
7.2
7.2.1
7.2.2
7.2.3
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Abhilfe bei ungünstigem Rekursionsverhalten

...oft ist folgende Verbesserung möglich:

- Ersetzung aufwandsungünstiger durch günstigere Rekursionsmuster!

Zum Beispiel die Rückführung

- baumartiger Rekursion
- linearer Rekursion

auf

- repetitive Rekursion (s.a. Lsg. 1, Kap. 7.2.3, und Anh. E).

Vortr. III
Teil III
Kap. 7
7.1
7.2
7.2.1
7.2.2
7.2.3
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr
Klassen-
zim-
mer II
Hinweis
Aufgabe

Rückführung linearer auf repetitive Rek. (1)

...am Beispiel der Fakultätsfunktion:

Naheliegende Implementierung mittels linearer Rekursion:

```
fac :: Integer -> Integer
fac n
  | n == 0    = 1
  | otherwise = n * fac (n-1)
```

Vortr. III
Teil III
Kap. 7
7.1
7.2
7.2.1
7.2.2
7.2.3
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Rückführung linearer auf repetitive Rek. (2)

Günstigere Formulierung mittels repetitiver Rekursion durch

- Rechnen auf Parameterposition

...mit einem zusätzlichen (Akkumulations-) Parameter, in dem sukzessive das Ergebnis akkumuliert wird:

```
type Akkumulation = Integer

fac :: Integer -> Integer
fac n = fac_repetitiv n 1

fac_repetitiv :: Integer -> Akkumulation -> Integer
fac_repetitiv n akkumulation
  | n == 0     = akkumulation
  | otherwise = fac_repetitiv (n - 1) (n * akkumulation)
```

Beachte: Überlagerungen mit anderen Effekten sind möglich, so dass sich möglicherweise kein Effizienzgewinn realisiert!

Vortr. III
Teil III
Kap. 7
7.1
7.2
7.2.1
7.2.2
7.2.3
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Weitere Möglichkeiten zur Verbesserung

...bieten spezielle Programmiertechniken wie

- Dynamische Programmierung
- Memoization

Zentrale Idee:

- Speicherung und Wiederverwendung statt Wiederberechnung bereits berechneter (Teil-) Ergebnisse.

  (Siehe etwa die effiziente Berechnung der Fibonacci-Zahlen mithilfe einer Memo-Liste)

Hinweis: Dynamische Programmierung und Memoization werden in der LVA 185.A05 Fortgeschrittene funktionale Programmierung ausführlich behandelt.

Vortr. III
Teil III
Kap. 7
7.1
7.2
7.2.1
7.2.2
7.2.3
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Kapitel 7.3

Aufrufgraphen

Vortr. III
Teil III
Kap. 7
7.1
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr
Klassen-
zim-
mer II
Hinweis
Aufgabe

Struktur von Programmen

Programme funktionaler Programmiersprachen (auch Haskell-Programme) sind i.a.

- Systeme (wechselweiser) rekursiver Rechenvorschriften, die sich hierarchisch oder/und wechselweise aufeinander abstützen.

Aufrufgraphen erleichtern es, sich über die

- Struktur von Systemen von Rechenvorschriften

Klarheit zu verschaffen.

Vortr. III
Teil III
Kap. 7
7.1
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Aufrufgraphen

...sei S ein System von Rechenvorschriften.

Definition 7.3.1 (Aufrufgraph)

Der Aufrufgraph von S ist ein Graph, der

- für jede in S deklarierte Rechenvorschrift einen Knoten mit dem Namen der Rechenvorschrift als Beschriftung enthält,
- eine gerichtete Kante vom Knoten f zum Knoten g genau dann enthält, wenn im Rumpf der zu f gehörigen Rechenvorschrift die zu g gehörige Rechenvorschrift aufgerufen wird.

Vortr. III
Teil III
Kap. 7
7.1
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Beispiel: Aufrufgraphen (1)

...der Rechenvorschriften bzw. Systeme von Rechenvorschriften

add, add′, fac, fib, max und mx:

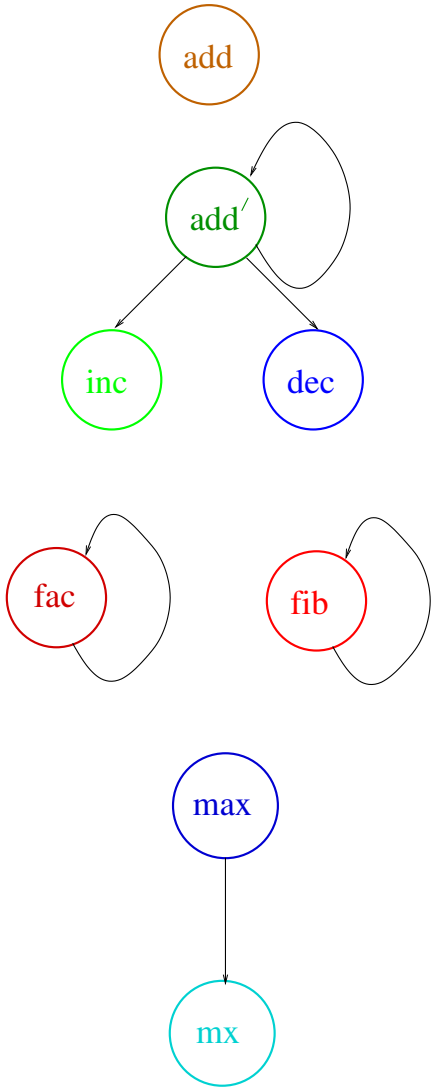
```
add :: Int -> Int -> Int
add m n = (+) m n

add'  :: Int -> Int -> Int
add' m n | n == 0    = m
         | n >0      = add' (inc m) (dec n)
         | otherwise = add' (dec m) (inc n)
 where inc :: Int -> Int
       inc n = n+1
       dec :: Int -> Int
       dec n = n-1

fac :: Integer -> Integer
fac n | n == 0    = 1
      | otherwise = n * fac (n-1)

fib :: Integer -> Integer
fib n | n == 0    = 0
      | n == 1    = 1
      | otherwise = fib (n-1) + fib (n-2)

max :: Int -> Int -> Int -> Int
max p q r
 | (mx p q == p) && (p `mx` r == p) = p
 | (mx p q == q) && (q `mx` r == q) = q
 | otherwise                        = r
 where mx :: Int -> Int -> Int
       mx p q | p >= q    = p
              | otherwise = q
```



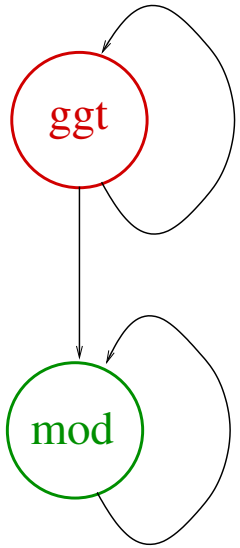
Vortr. III
Teil III
Kap. 7
7.1
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Beispiel: Aufrufgraphen (2)

...des Systems hierarchischer Rechenvorschriften der Funktionen ggt und mod:

```
ggt :: Int -> Int -> Int
ggt m n
  | n == 0 = m
  | n > 0  = ggt n (mod m n)

mod :: Int -> Int -> Int
mod m n
  | m < n  = m
  | m >= n = mod (m-n) n
```



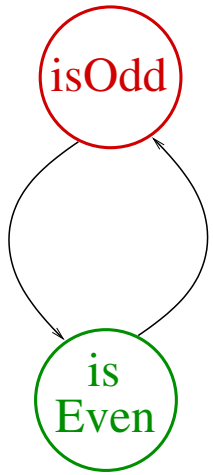
Vortr. III
Teil III
Kap. 7
7.1
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Beispiel: Aufrufgraphen (3)

...des Systems wechselweise rekursiver Rechenvorschriften der Funktionen `isOdd` und `isEven`:

```
isOdd :: Integer -> Bool
isOdd n
 | n == 0 = False
 | n > 0  = isEven (n-1)


isEven :: Integer -> Bool
isEven n
 | n == 0 = True
 | n > 0  = isOdd (n-1)
```



Vortr. III
Teil III
Kap. 7
7.1
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr
Klassen-
zim-
mer II
Hinweis
Aufgabe

Beispiel: Aufrufgraphen (4)

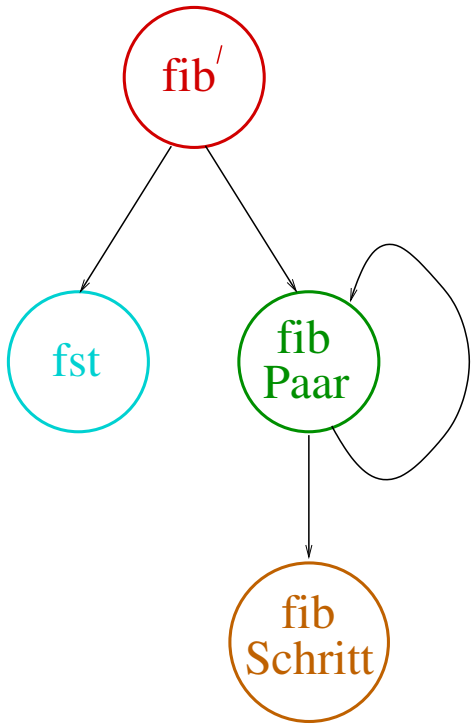
...des Systems hierarchischer Rechenvorschriften der Funktionen `fib`, `fibPaar`, `fibSchritt` und `fst`:

```
fibSchritt :: (Integer,Integer) -> (Integer,Integer)
fibSchritt (m,n) = (n,m+n)

fibPaar :: Integer -> (Integer,Integer)
fibPaar n =
 | n == 0    = (0,1)
 | otherwise = fibSchritt (fibPaar (n-1))

fib' :: Integer -> Integer
fib' n = fst (fibPaar n)

fst :: (a,b) -> a
fst (x,y) = x
```



Vortr. III
Teil III
Kap. 7
7.1
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Interpretation von Aufrufgraphen (1)

Aus den Aufrufgraphen eines Systems von Rechenvorschriften ist u.a. ablesbar:

- Direkte Rekursivität einer Funktion: 'Selbstkringel'.
  (z.B. bei den Aufrufgraphen der Funktionen `fac` und `fib`)
- Wechselweise Rekursivität zweier Funktionen: Kreise mit zwei Kanten.
  (z.B. bei den Aufrufgraphen der Funktionen `isOdd` und `isEven`)
- Direkte hierarchische Abstützung einer Funktion auf eine andere: Es gibt eine Kante von Knoten f zu Knoten g, aber nicht umgekehrt.
  (z.B. bei den Aufrufgraphen der Funktionen `max` und `mx`)

Vortr. III
Teil III
Kap. 7
7.1
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Interpretation von Aufrufgraphen (2)

- Indirekte hierarchische Abstützung einer Funktion auf eine andere: Knoten g ist von Knoten f über eine Folge von Kanten erreichbar, aber nicht umgekehrt.
  (z.B. bei den Aufrufgraphen der Funktionen `fib'`, `fibPaar` und `fibSchritt`)
- Indirekte wechselweise Abstützung: Knoten g ist von Knoten f über eine Folge von Kanten erreichbar und umgekehrt (Kreise mit mehr als zwei Kanten).
- Unabhängigkeit/Isolation einer Funktion: Knoten f hat (ggf. mit Ausnahme eines Selbstkringels) weder ein- noch ausgehende Kanten.
  (z.B. bei den Aufrufgraphen der Funktionen `add`, `fac` und `fib`)
- ...

Vortr. III
Teil III
Kap. 7
7.1
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr
Klassen-
zim-
mer II
Hinweis
Aufgabe

Kapitel 7.4

Komplexität, Komplexitätsklassen

Vortr. III
Teil III
Kap. 7
7.1
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr
Klassen-
zim-
mer II
Hinweis
Aufgabe

Rechenaufwand von Algorithmen

...als Anzahl elementarer Operationen zu (angenommenen) Einheitskosten einer imaginären Maschine.

Sei

1. $\mathcal{A} : \mathcal{P} \to \mathcal{L}$ ein Algorithmus, der Probleminstanzen p aus einer Menge $\mathcal{P}$ auf Lösungsinstanzen l aus einer Menge $\mathcal{L}$ abbildet.
2. $f_{\mathcal{A}} : \mathcal{P} \to \mathbb{N}_0$ eine Funktion, die für jede Probleminstanz $p \in \mathcal{P}$ die Zahl elementarer Operationen angibt, die $\mathcal{A}$ zur Lösung von p benötigt.
3. $\gamma : \mathcal{P} \to \mathbb{N}_0$ eine Funktion, die jeder Probleminstanz $p \in \mathcal{P}$ eine natürliche Zahl als Größe zuordnet.

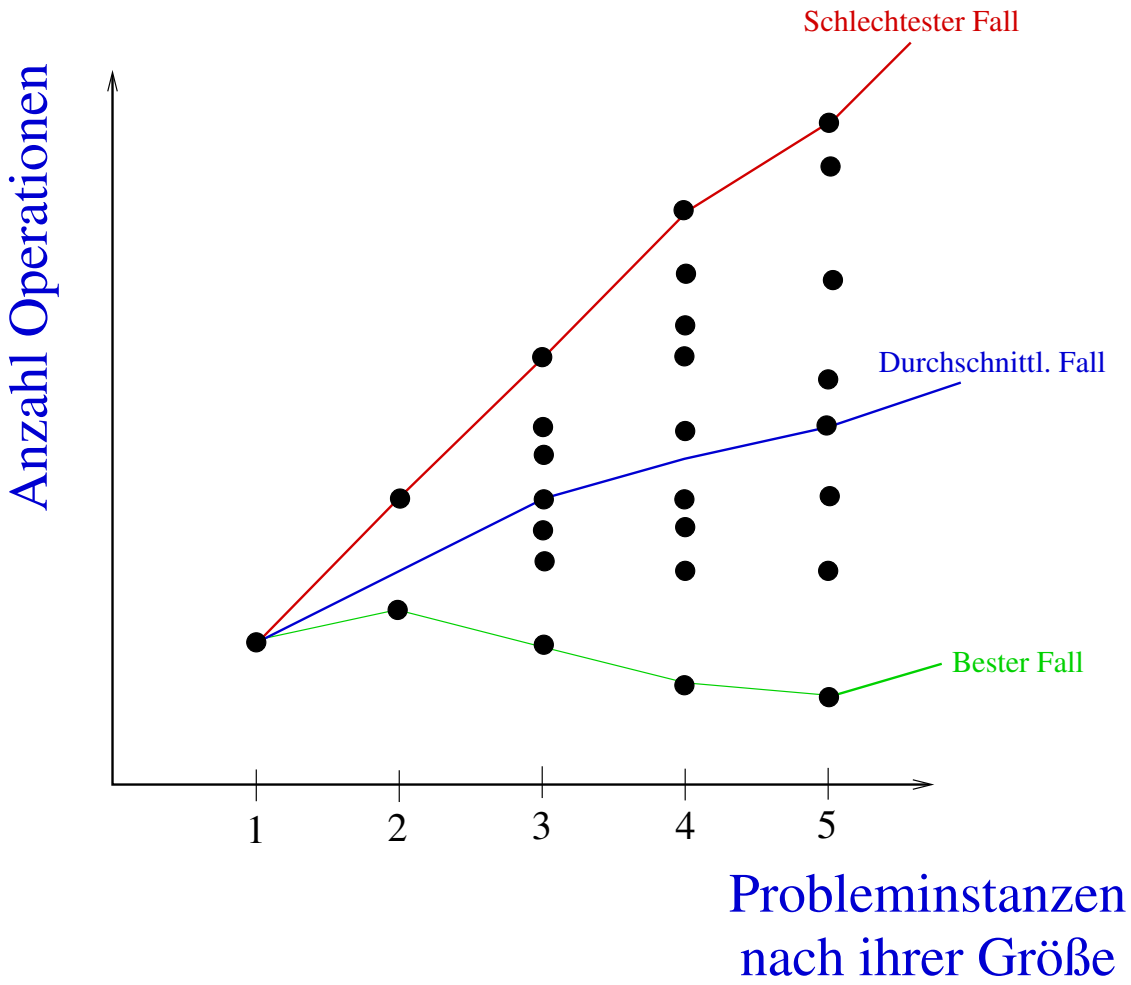
Beispiel: Sortieren ganzer Zahlen

- $\mathcal{P}$: Menge aller Listen über ganzen Zahlen.
- $\mathcal{L} \subseteq \mathcal{P}$: Menge aller aufsteigend sortierten Listen.
- $\mathcal{A}$: Quicksort
- $\gamma(p)$, $p \in \mathcal{P}$: Länge von p, d.h. Anzahl Elemente in p.

Vortr. III
Teil III
Kap. 7
7.1
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Aufwand in Abhängigkeit der Problemgröße

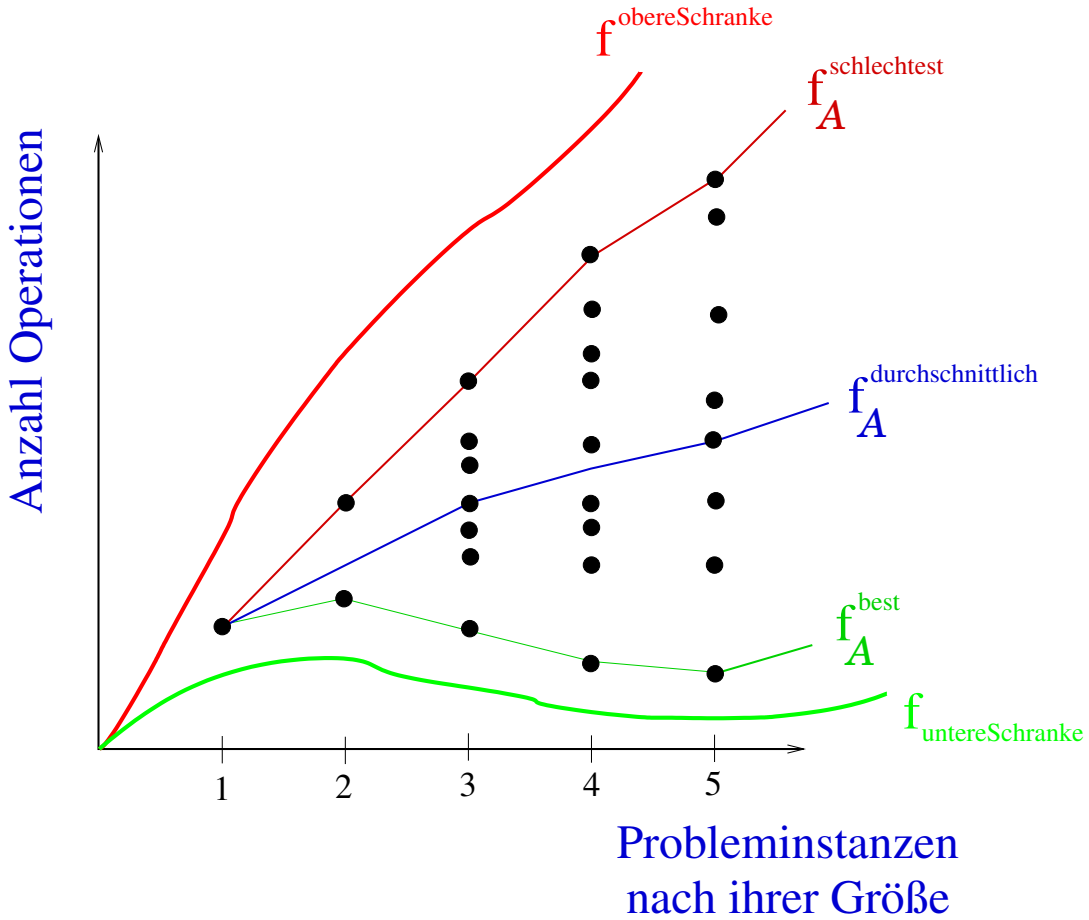
...von Algorithmus $\mathcal{A}$, wobei • Probleminstanzen darstellen:



Vortr. III
Teil III
Kap. 7
7.1
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr
Klassen-
zim-
mer II
Hinweis
Aufgabe

Aus Gründen der Praktikabilität

...betrachtet man i.a nicht die (meist schwer beschreibbaren) Aufwandsfunktionen $f_{\mathcal{A}}^{schlechtest}$, $f_{\mathcal{A}}^{durchschnittlich}$ und $f_{\mathcal{A}}^{best}$ direkt, sondern konzentriert sich auf Funktionen $f^{obereSchranke}$ und $f_{untereSchranke}$, die $f_{\mathcal{A}}^{schlechtest}$ und $f_{\mathcal{A}}^{best}$ beschränken:



Vortr. III
Teil III
Kap. 7
7.1
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Obere, untere, einhüllende Schranken

Seien $f, g : \mathbb{N}_0 \to IR_0^+$ zwei Funktionen von den natürlichen in die positiven reellen Zahlen (einschl. 0 jeweils; sonst $\mathbb{N}$, IR^+).

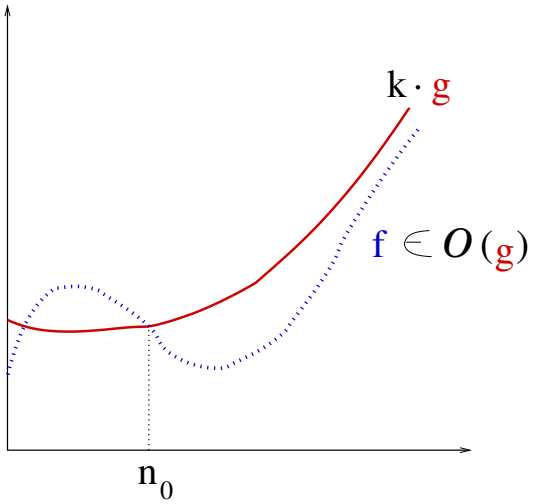
Definition 7.4.1 (Obere, untere, einhüll. Schranke)

g heißt asymptotische

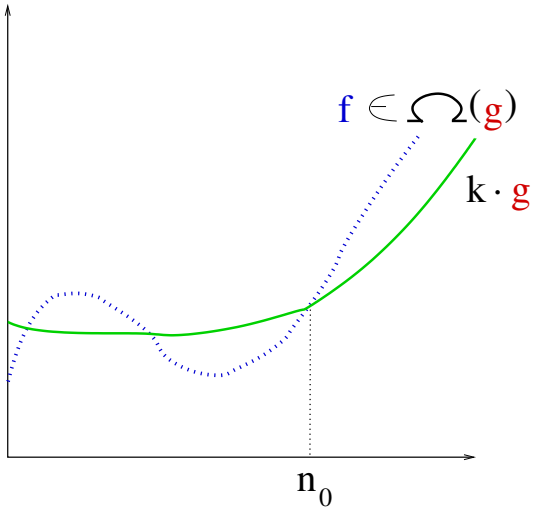
1. obere Schranke von f gdw.
   $\exists k \in IR^+. \; \exists n_0 \in \mathbb{N}_0. \; \forall n \in \mathbb{N}_0. \; n \geq n_0 \;\Rightarrow\; f(n) \leq k * g(n)$
   In diesem Fall schreiben wir: $f \in O(g)$ (oder $f = O(g)$).
2. untere Schranke von f gdw.
   $\exists k \in IR^+. \; \exists n_0 \in \mathbb{N}_0. \; \forall n \in \mathbb{N}_0. \; n \geq n_0 \;\Rightarrow\; f(n) \geq k * g(n)$
   In diesem Fall schreiben wir: $f \in \Omega(g)$ (oder $f = \Omega(g)$).
3. einhüllende Schranke von f gdw. g ist obere und untere Schranke von f: $f \in O(g) \wedge f \in \Omega(g)$.
   In diesem Fall schreiben wir: $f \in \Theta(g)$ (oder $f = \Theta(g)$).

Vortr. III
Teil III
Kap. 7
7.1
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

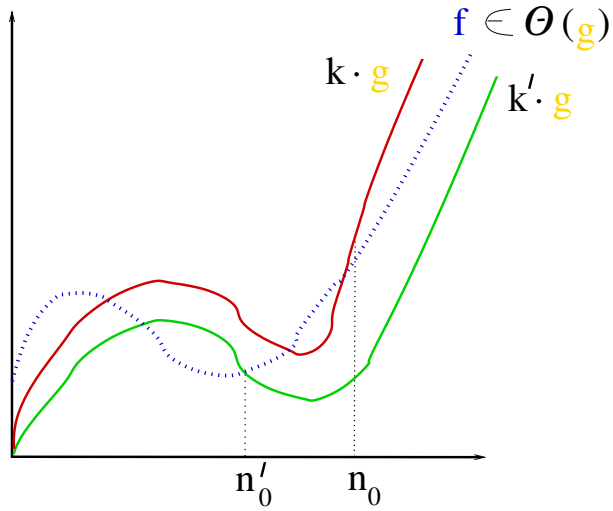
Asymptotische Schranken: Veranschaulichung



Obere Schranke



Untere Schranke



Einhüllende Schranke

1. $O(g) = \{f \mid g \text{ asympt. obere Schranke von } f\}$
2. $\Omega(g) = \{f \mid g \text{ asympt. untere Schranke von } f\}$
3. $\Theta(g) = \{f \mid g \text{ asympt. obere u. untere Schranke von } f\}$

Vortr. III
Teil III
Kap. 7
7.1
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Sprechweisen (für $f, g : \mathbb{N}_0 \to I\!R_0^+$)

Man sagt, Funktion f

- ist
  (a) höchstens
  (b) mindestens
  (c) genau

  von der Größenordnung g
- wächst von der Größenordnung (oder größenordnungsmäßig)
  (a) höchstens
  (b) mindestens
  (c) genau

  so schnell wie g, wenn gilt:
  (a) $f \in O(g)$
  (b) $f \in \Omega(g)$
  (c) $f \in \Theta(g)$

Vortr. III
Teil III
Kap. 7
7.1
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Beachte

Die Schreibweise '$=$' für '$\in$', z.B. $f = O(g)$ für $f \in O(g)$ ist nützlich und verbreitet, aber ungenau, da '$=$' in diesem Kontext weder symmetrisch noch transitiv gelesen werden kann:

- Lesen wir $f = O(g)$ gleichbedeutend mit $f \in O(g)$, so ist die Schreibweise $O(g) = f$ sinnlos und ohne Bedeutung.
- Aus $f = O(g)$ und $h = O(g)$ kann nicht geschlossen werden: $f = h$.

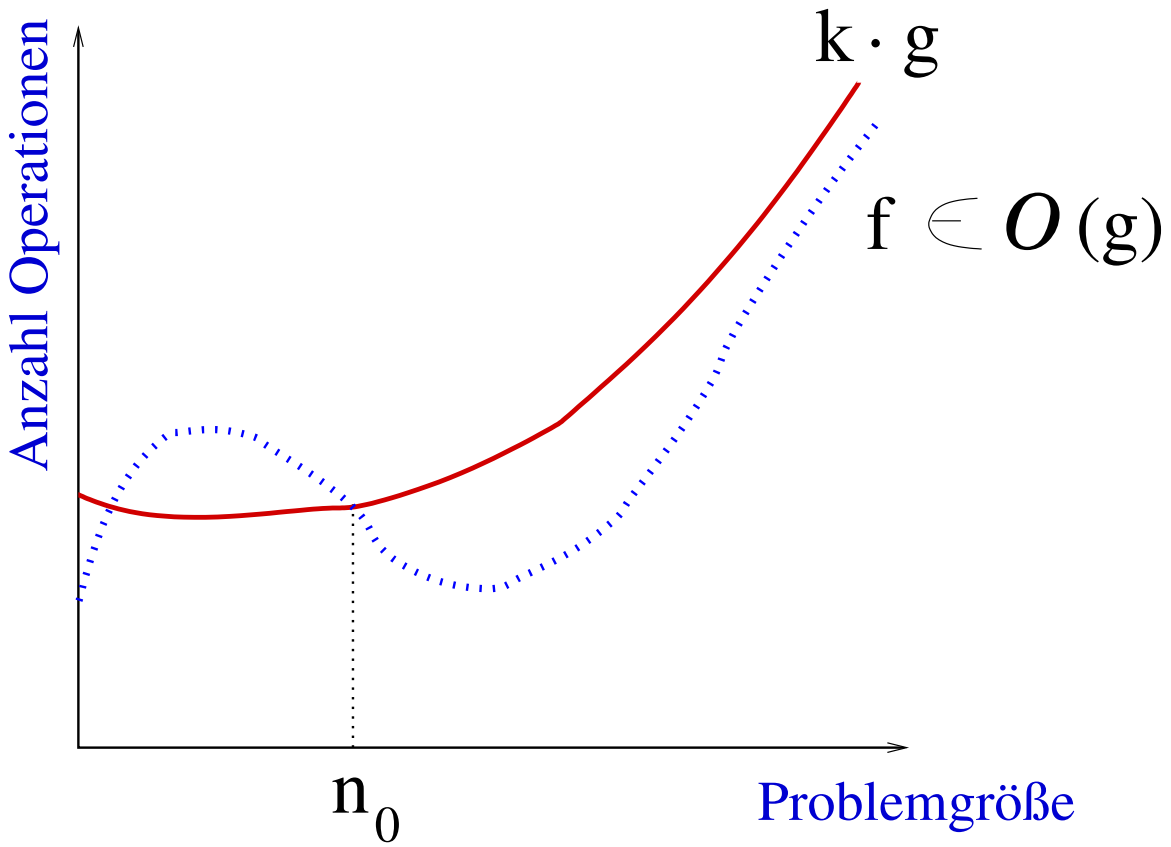
Das bedeutet:

- In Kontexten wie $f = O(g)$, $f = \Omega(g)$, $f = \Theta(g)$ ist '$=$' als Einweggleichung ausschließlich und nicht umkehrbar von links nach rechts zu lesen.

Vortr. III
Teil III
Kap. 7
7.1
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

In der Praxis bes. wichtig: Asymp. obere Schr.

...sei $f : \mathbb{N}_0 \to \mathbb{N}_0$ eine Funktion, die die max. Zahl elementarer Operationen eines Algorithmus $\mathcal{A}$ in Abhängigkeit der durch eine natürliche Zahl beschriebenen Problemgröße angibt:



Vortr. III
Teil III
Kap. 7
7.1
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr
Klassen-
zim-
mer II
Hinweis
Aufgabe

Beispiele

...in der Praxis häufig auftretender Kostenfunktionen:

Kürzel	Aufwand	Intuition: *Vertausendfachte Eingabe heißt...*
$O(c)$	konstant	gleiche Arbeit
$O(log_2\ n)$	logarithmisch	nur zehnfache Arbeit
$O(n)$	linear	auch vertausendfachte Arbeit
$O(n\ log_2\ n)$	quasi-linear	zehntausendfache Arbeit
$O(n^2)$	quadratisch	millionenfache Arbeit
$O(n^3)$	kubisch	milliardenfache Arbeit
$O(n^c)$	polynomiell	gigantisch viel Arbeit (f. großes c)
$O(2^n)$	exponentiell	hoffnungslos

Peter Pepper. Funktionale Programmierung in OPAL, ML, Haskell und Gofer. Springer-V., 2. Auflage, 2003, Kap. 11.

Anm.: Die Angabe der Basis bei (quasi-) logarithm. Komplexität entfällt üblicherw. (auch bei Pepper), da sie als Konstante in d. O-Not. aufgeht.

Vortr. III
Teil III
Kap. 7
7.1
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Veranschaulichung über Skalierbarkeit

...was das Wachstum von Probleminstanzen für die reale Ausführungszeit bedeutet (Annahme: Jede Elementarop. benötigt $1\ \mu s$; $f(n)$ Zahl max. benötigter Elementarop. für Instanzen d. Größe n):

Grösse n	Linear $f(n) = n$	Quadratisch $f(n) = n^2$	Kubisch $f(n) = n^3$	Exponentiell $f(n) = 2^n$
1	1 μs	1 μs	1 μs	2 μs
10	10 μs	100 μs	1 ms	1 ms
20	20 μs	400 μs	8 ms	1 s
30	30 μs	900 μs	27 ms	18 min
40	40 μs	2 ms	64 ms	13 Tage
50	50 μs	3 ms	125 ms	36 Jahre
60	60 μs	4 ms	216 ms	36 560 Jahre
100	100 μs	10 ms	1 sec	$4 * 10^{16}$ Jahre
1000	1 ms	1 sec	17 min	sehr, sehr lange...

Peter Pepper. Funktionale Programmierung in OPAL, ML, Haskell und Gofer. Springer-V., 2. Auflage, 2003, Kap. 11.

Vortr. III
Teil III
Kap. 7
7.1
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Veransch. über handhabbare Problemgröße (1)

...mit folgender umgekehrter Fragestellung:

- Probleminstanzen welcher Größe können gerade noch gelöst werden, so dass das Ergebnis 'gefühlt sofort', ohne merklich wahrnehmbare Berechnungsverzögerung vorliegt?

Dafür nehmen wir an:

- Ein Ergebnis liegt 'gefühlt sofort' vor, wenn die Berechnungszeit bz nicht wesentlich mehr als eine Hundertstelsekunde beträgt, d.h. wenn für die Berechnungszeit $bz(p)$ einer Probleminstanz p gilt:

$$bz(p) < (10.000 + \varepsilon)\,\mu s = (10 + \varepsilon)\,ns = (0,01 + \varepsilon)\,s$$

- Eine Elementaroperation benötigt $1\,\mu s$.

Vortr. III
Teil III
Kap. 7
7.1
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Veransch. über handhabbare Problemgröße (2)

Gegeben seien:

1. Sechs unterschiedliche Lösungsalgorithmen mit Laufzeit:
   - exponentiell ($\mathcal{A}_1$) $f_{\mathcal{A}_1} \in O(2^n)$
   - kubisch ($\mathcal{A}_2$) $f_{\mathcal{A}_2} \in O(n^3)$
   - quadratisch ($\mathcal{A}_3$) $f_{\mathcal{A}_3} \in O(n^2)$
   - quasi-linear ($\mathcal{A}_4$) $f_{\mathcal{A}_4} \in O(n \log_2 n)$
   - linear ($\mathcal{A}_5$) $f_{\mathcal{A}_5} \in O(n)$
   - logarithmisch ($\mathcal{A}_6$) $f_{\mathcal{A}_6} \in O(\log_2 n)$
2. Funktionen $f_{\mathcal{A}_i} : \mathbb{N} \to \mathbb{N}$, $1 \leq i \leq 6$, die für jeden Algorithmus angeben, wieviele Elementaroperationen $\mathcal{A}_i$ für die Lösung einer Probleminstanz der Größe n, $n \in \mathbb{N}$, höchstens benötigt.

   Informell: $f_{\mathcal{A}_i}$ übersetzt Problemgröße in (bis auf konstanten Faktor) max. Anzahl der von $\mathcal{A}_i$ benötigten Elementaroperationen zu je $1\,\mu s$.

Vortr. III
Teil III
Kap. 7
7.1
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Veransch. über handhabbare Problemgröße (3)

Algorithmus	Zeitbudget $(10.000 + \varepsilon)\,\mu s$	Handhabbare Problemgröße im Zeitbudget
Exponentiell $f_{\mathcal{A}_1}(n) = 2^n$	$f_{\mathcal{A}_1}(13) = 8.192$ $f_{\mathcal{A}_1}(14) = 16.384$	14
Kubisch $f_{\mathcal{A}_2}(n) = n^3$	$f_{\mathcal{A}_2}(21) = 9.261$ $f_{\mathcal{A}_2}(22) = 10.648$ $f_{\mathcal{A}_2}(26) = 17.576$	26
Quadratisch $f_{\mathcal{A}_3}(n) = n^2$	$f_{\mathcal{A}_3}(100) = 10.000$	100
Quasi-linear $f_{\mathcal{A}_4}(n) = n\,log_2 n$	$f_{\mathcal{A}_4}(1.000) \approx 10.000$	1.000
Linear $f_{\mathcal{A}_5}(n) = n$	$f_{\mathcal{A}_5}(10.000) = 10.000$	10.000
Logarithmisch $f_{\mathcal{A}_6}(n) = log_2 n$	$f_{\mathcal{A}_6}(2^{10.000}) = 10.000$	$2^{10.000}$

Vortr. III
Teil III
Kap. 7
7.1
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Wachstum informell gedeutet

– Exponentiell: Die Zahl der Operationen wächst im Vergleich zur Problemgröße äußerst schnell:

Problemgröße n	Anzahl Operationen 2^n
1	2
14	16.384
1.000	$2^{1.000}$
n	2^n

– Logarithmisch: Die Zahl der Operationen wächst im Vergleich zur Problemgröße äußerst langsam:

Problemgröße 2^n	Anzahl Operationen n
1	0
$16 \approx 14$	4
$1.024 \approx 1.000$	10
$2^{16.384}$	16.384
2^{2^n}	2^n

Vortr. III
Teil III
Kap. 7
7.1
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Lohnt der Kauf eines schnelleren Rechners?

Algorithmus	Alter Rechner	Neuer schnellerer Rechner		
		10x	100x	1.000x
Exponentiell $f_{\mathcal{A}_1}(n) = 2^n$	14 ($2^{14} = 16.384$)	17 ($2^{17} = 131.072$)	20 ($2^{20} = 1.048.576$)	24 ($2^{24} = 16.777.216$)
Kubisch $f_{\mathcal{A}_2}(n) = n^3$	26 ($26^3 = 17.576$)	47 ($47^3 = 103.823$)	100 ($100^3 = 1.000.000$)	216 ($216^3 = 10.077.696$)
Quadratisch $f_{\mathcal{A}_3}(n) = n^2$	100 ($100^2 = 10.000$)	317 ($317^2 = 100.489$)	1.000 ($1.000^2 = 1.000.000$)	3.163 ($3.163^2 = 10.004.569$)
Quasi-linear $f_{\mathcal{A}_4}(n) = n\,log_2\,n$	1.000 ($10^3\ log_2 10^3 \approx 10^3 * 10 = 10.000$)	9.000 ($9.000\ log_2 9.000 \approx 9.000 * 13 = 117.000$)	65.000 ($65.000\ log_2 65.000 \approx 65.000 * 16 = 1.040.000$)	530.000 ($530.000\ log_2 530.000 \approx 530.000 * 19 = 10.070.000$)
Linear $f_{\mathcal{A}_5}(n) = n$	10.000	100.000	1.000.000	10.000.000
Logarithmisch $f_{\mathcal{A}_6}(n) = log_2\,n$	$2^{10.000}$ ($log_2 2^{10^4} = 10.000$)	$2^{100.000}$ ($log_2 2^{10^5} = 100.000$)	$2^{1.000.000}$ ($log_2 2^{10^6} = 1.000.000$)	$2^{10.000.000}$ ($log_2 2^{10^7} = 10.000.000$)

Beobachtung

Der Kauf eines neuen, schnelleren Rechners bringt verglichen mit dem Finden und Wechsel zu einem asymptotisch besseren Algorithmus fast nichts.

- Exponentieller Algorithmus, Rechner um Faktor 1.000 schneller, im Bsp. wächst handhabbar von 14 auf 24:

  $\rightsquigarrow$ bedeutungslos, weil noch immer weit zu wenig.
- Logarithmischer Algorithmus, Rechner um Faktor 1.000 schneller, im Bsp. wächst handbabbar von $2^{10.000}$ auf $2^{10.000.000}$:

  $\rightsquigarrow$ bedeutungslos, weil schon $2^{10.000}$ riesig (genug) ist.

Kein Königsweg:

- Das Finden eines asymptotisch besseren Algorithmus ist kein Selbstläufer; möglicherweise gibt es auch keinen.

Vortr. III
Teil III
Kap. 7
7.1
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Interpretation, Folgerungen

- Einem asymptotisch schlechten Algorithmus hat auch ein schneller(er) Rechner nichts entgegenzusetzen:

  Selbst ein 100.000-fach schnellerer Rechner mit 1.000.000.000 Operationen pro $0,01\,\mathrm{s}$ erlaubte bei exponentiellem Algorithmus ohne Verlängerung der erlaubten Antwortzeit lediglich Probleme bis zur Größe 30 zu lösen: $2^{30} = 1.073.741.824$.
- Ein asymp. schlechter Algorithmus verhält sich schlecht, auch wenn er auf einen schnelleren Rechner portiert wird.
- Asymp. schlechtes Alg.-Verhalten ist portierungsinvariant.

Faustregel:

- Ein langsamer Rechner mit asymptotisch gutem Algorithmus gewinnt gegen einen schnellen Rechner mit asymptotisch schlechtem oder auch nur schlechterem Algorithmus bei Probleminstanzen nichttrivialer Größe immer!

Vortr. III
Teil III
Kap. 7
7.1
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr
Klassen-
zim-
mer II
Hinweis
Aufgabe

Abschließend

Die Beispiele und Überlegungen dieses Abschnitts machen deutlich:

- Rekursionsmuster beeinflussen (auch) die Effizienz einer Implementierung (siehe die naive baumartig-rekursive Implementierung der Fibonacci-Funktion).
- Die Wahl eines zweckmäßigen und zweckmäßig eingesetzten Rekursionsmusters ist deshalb wichtig.

WICHTIG: Nicht bestimmte Rekursionsmuster an sich sind

- problematisch, sondern ihr unzweckmäßiger Einsatz, wenn etwa wie im Fall der Fibonacci-Funktion baumartige Rekursion zu (unnötigen) Vielfachberechnungen von Werten führt!

Zweckmäßig eingesetzt bietet z.B. baumartige Rekursion viele Vorteile, darunter zur Parallelisierung! *Stichwort*: Teile und herrsche (oder divide et impera oder divide and conquer)!

Vortr. III
Teil III
Kap. 7
7.1
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Landausche Symbole

Die Symbole O, Ω, Θ, o und ω heißen Landausche Symbole und gehen zurück auf Arbeiten von:

- Edmund Landau. Handbuch der Lehre von der Verteilung der Primzahlen, Band I und II, B. G. Teubner, 1909. (o-Notation)
- Paul Bachmann. Die Analytische Zahlentheorie. Zahlentheorie, B. G. Teubner, 1894. (O-Notation)
- Godfrey H. Hardy, John E. Littlewood. Some Problems of Diophantine Approximation. Acta Mathematica 37:155-238, 1914. (Ω-Notation; mit vom heutigen Gebrauch abweichender Bedeutung von Ω als 'o-Negation')

Für einen genaueren historischen Abriss siehe die Arbeit:

- Donald E. Knuth. Big Omicron and Big Omega and Big Theta. ACM SIGACT News 8(2):18-24, 1976.

...der für die Verwendung der Landauschen Symbole in ihrer heutigen Bedeutung maßgeblicher Einfluss zukommt.

Vortr. III
Teil III
Kap. 7
7.1
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr
Klassen-
zim-
mer II
Hinweis
Aufgabe

Kapitel 7.5

Leseempfehlungen

Vortr. III
Teil III
Kap. 7
7.1
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr
Klassen-
zim-
mer II
Hinweis
Aufgabe

Basisleseempfehlungen für Kapitel 7

Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011. (Kapitel 4, Rekursion als Entwurfstechnik; Kapitel 9, Laufzeitanalyse von Algorithmen; Kapitel 9.2, Landau-Symbole)

Peter Pepper. *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*. Springer-V., 2. Auflage, 2003. (Kapitel 5, Rekursion; Kapitel 11, Formalismen 3: Aufwand und Terminierung)

Ben Goldreich. *Invitation to Complexity Theory*. Crossroads, the ACM Magazine for Students 18(3):18-22, 2012.

Ingo Wegener. *Komplexität*. In Informatik-Handbuch, Peter Rechenberg, Gustav Pomberger (Hrsg.), Carl Hanser Verlag, 4. Auflage, 119-144, 2006. (Kapitel 5.1, Größenordnungen und die $\mathcal{O}$-Notation)

Vortr. III
Teil III
Kap. 7
7.1
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen- zim- mer II
Hinweis
Aufgabe

Weiterführende Leseempfehlungen für Kap. 7

Thomas H. Cormen, Charles E. Leiserson, Ronald Rivest, Clifford Stein. *Algorithmen – Eine Einführung*. Oldenbourg Verlag, 2004. Kapitel 3, Wachstum von Funktionen; Kapitel 3.1, Asymptotische Notation (Θ, O, Ω, o, ω); Kapitel 3.2, Standardnotationen und Standardfunktionen)

Steven S. Skiena. *The Algorithm Design Manual*. Springer-V., 1998. (Kapitel 1.3.2, Best, Worst, and Average-Case Complexity; Kapitel 1.4, The Big Oh Notation)

Donald E. Knuth. *Big Omicron and Big Omega and Big Theta*. ACM SIGACT News 8(2):18-24, 1976. (s.a. Nachdruck unter gleichem Titel in: Donald E. Knuth. *Selected Papers on Analysis of Algorithms*. CSLI Lecture Notes Number 102, CSLI Publications, 35-41, 2012.)

Vortr. III
Teil III
Kap. 7
7.1
7.2
7.3
7.4
7.5
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Kapitel 8

Auswertung einfacher Ausdrücke

Vortr. III
Teil III
Kap. 7
Kap. 8
8.1
8.2
8.3
8.4
Kap. 9
Umgekehr
Klassen-
zim-
mer II
Hinweis
Aufgabe

Auswertung von Ausdrücken

...hat das Ziel, Ausdrücke soweit zu vereinfachen wie nur irgend möglich und so ihren Wert zu berechnen.

Dafür ist das Zusammenspiel des

- Expandierens ($\rightsquigarrow$ Funktionsterme, Funktionsaufrufe)
- Simplifizierens ($\rightsquigarrow$ Funktionstermfreie Ausdrücke)

von Ausdrücken zu organisieren.

In der Folge illustrieren wir das am Beispiel funktionstermfreier und einfacher funktionstermbehafteter Ausdrücke.

Vortr. III
Teil III
Kap. 7
Kap. 8
8.1
8.2
8.3
8.4
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Kapitel 8.1

Auswertung von Ausdrücken ohne Funktionsterme

Vortr. III
Teil III
Kap. 7
Kap. 8
8.1
8.2
8.3
8.4
Kap. 9
Umgekehr
Klassen-
zim-
mer II
Hinweis
Aufgabe

Auswertung funktionstermfreier Ausdrücke

Viele (Simplifikations-) Wege führenZiel, zum stets selben (!) Ziel (s.a. Kap. 12.3, 13.3), hier zum Wert 78, der Semantik (oder: Bedeutung) des Ausdrucks 2*3-5+7*11:

1. Simplifikations-Weg:

```
          2*3-5+7*11
(S) ->>   6-5+7*11
(S) ->>   1+7*11
(S) ->>   1+77   (S) ->>  78
```

2. S-Weg:

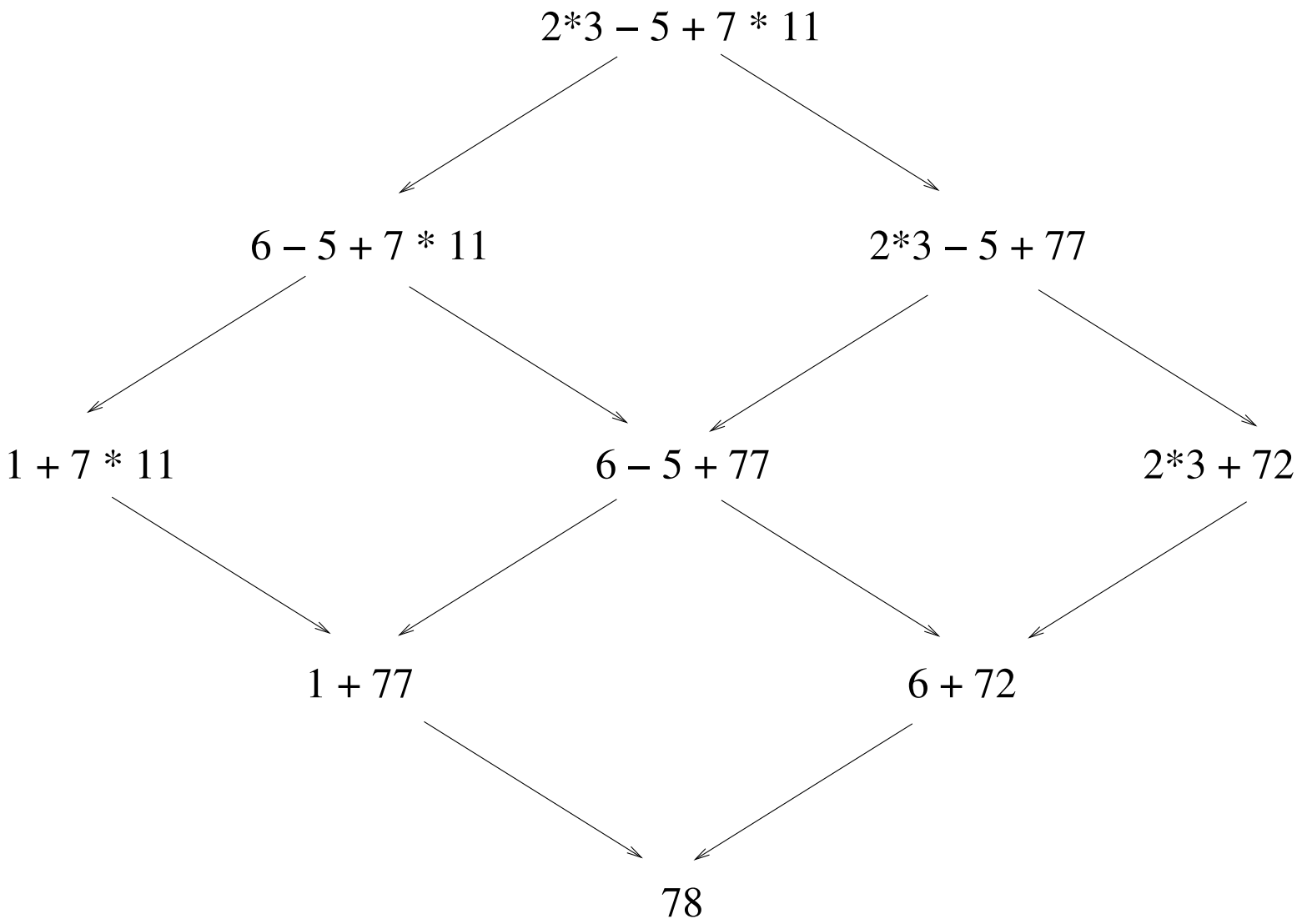
```
          2*3-5+7*11
(S) ->>   2*3-5+77
(S) ->>   6-5+77
(S) ->>   6+72   (S) ->>  78
```

3. S-Weg:

```
          2*3-5+7*11
(S) ->>   2*3-5+77
(S) ->>   2*3+72
(S) ->>   ...    (S) ->>  78
```

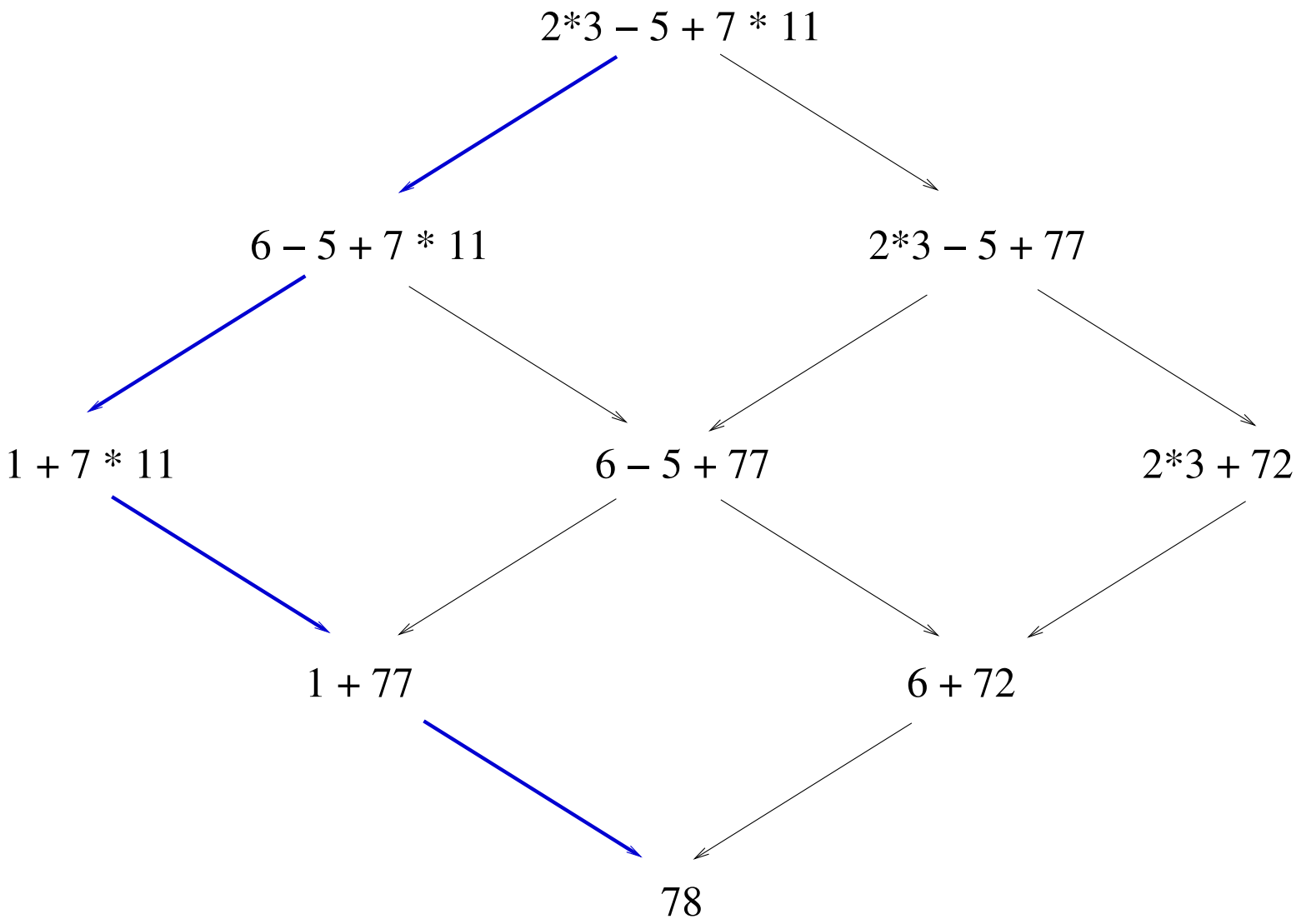
Vortr. III
Teil III
Kap. 7
Kap. 8
8.1
8.2
8.3
8.4
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Graphische Veranschaulichung (1)



Vortr. III
Teil III
Kap. 7
Kap. 8
8.1
8.2
8.3
8.4
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

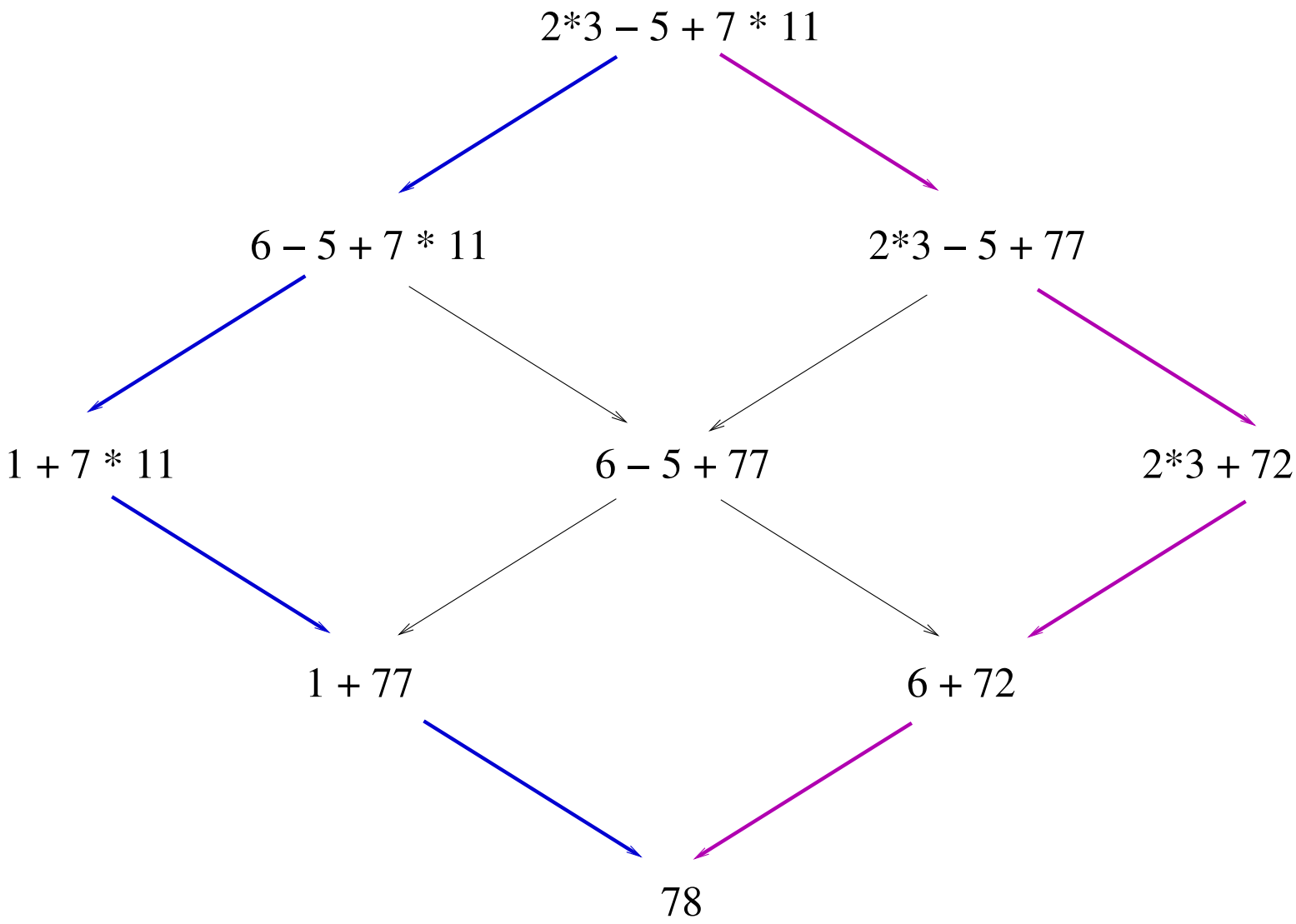
Graphische Veranschaulichung (2)



$\longrightarrow \ldots \longrightarrow$: Rechnen stets an linkestmöglicher Stelle!

Vortr. III
Teil III
Kap. 7
Kap. 8
8.1
8.2
8.3
8.4
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

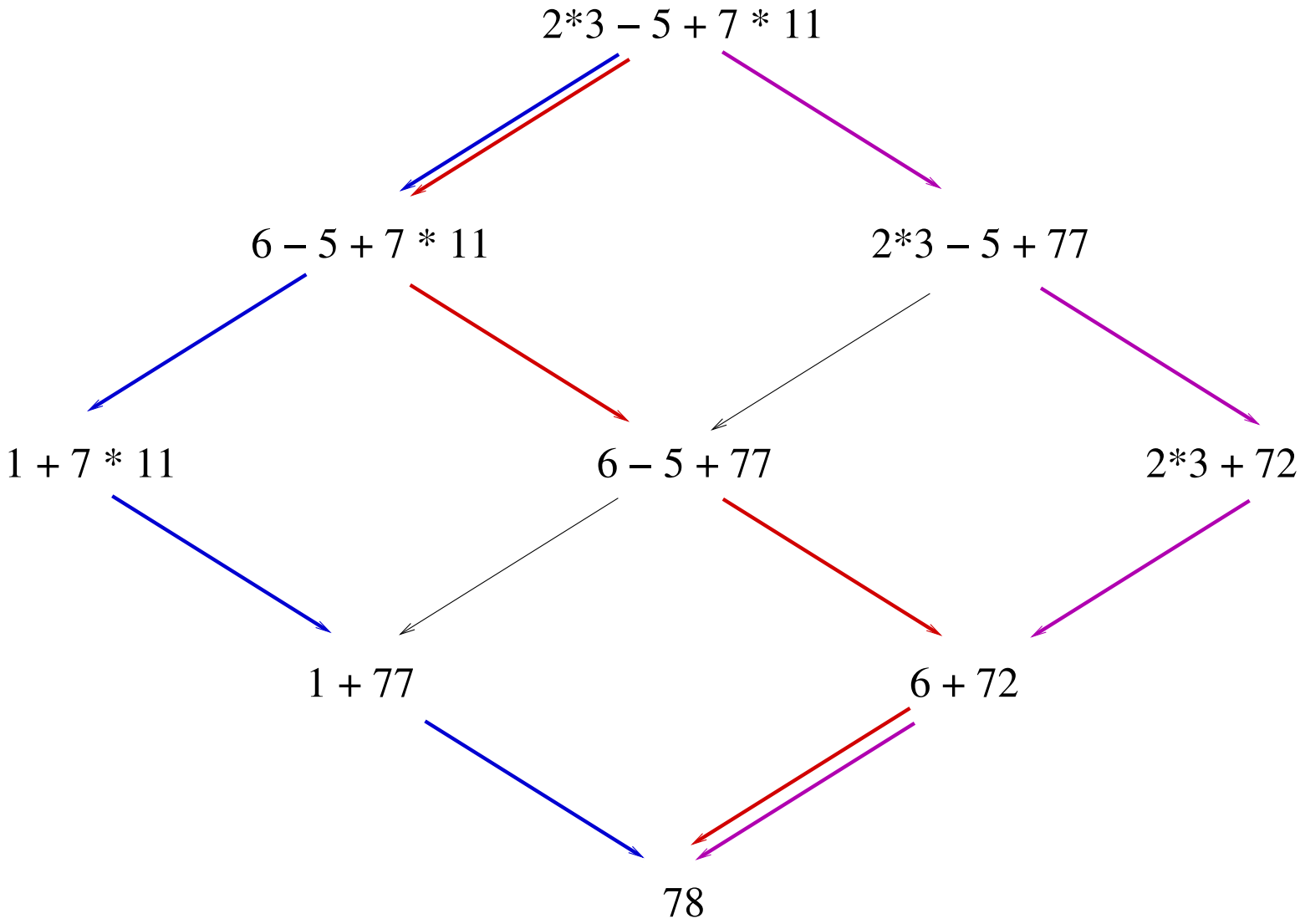
Graphische Veranschaulichung (3)



$\longrightarrow \ldots \longrightarrow$: Rechnen stets an rechtestmöglicher Stelle!

Vortr. III
Teil III
Kap. 7
Kap. 8
8.1
8.2
8.3
8.4
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

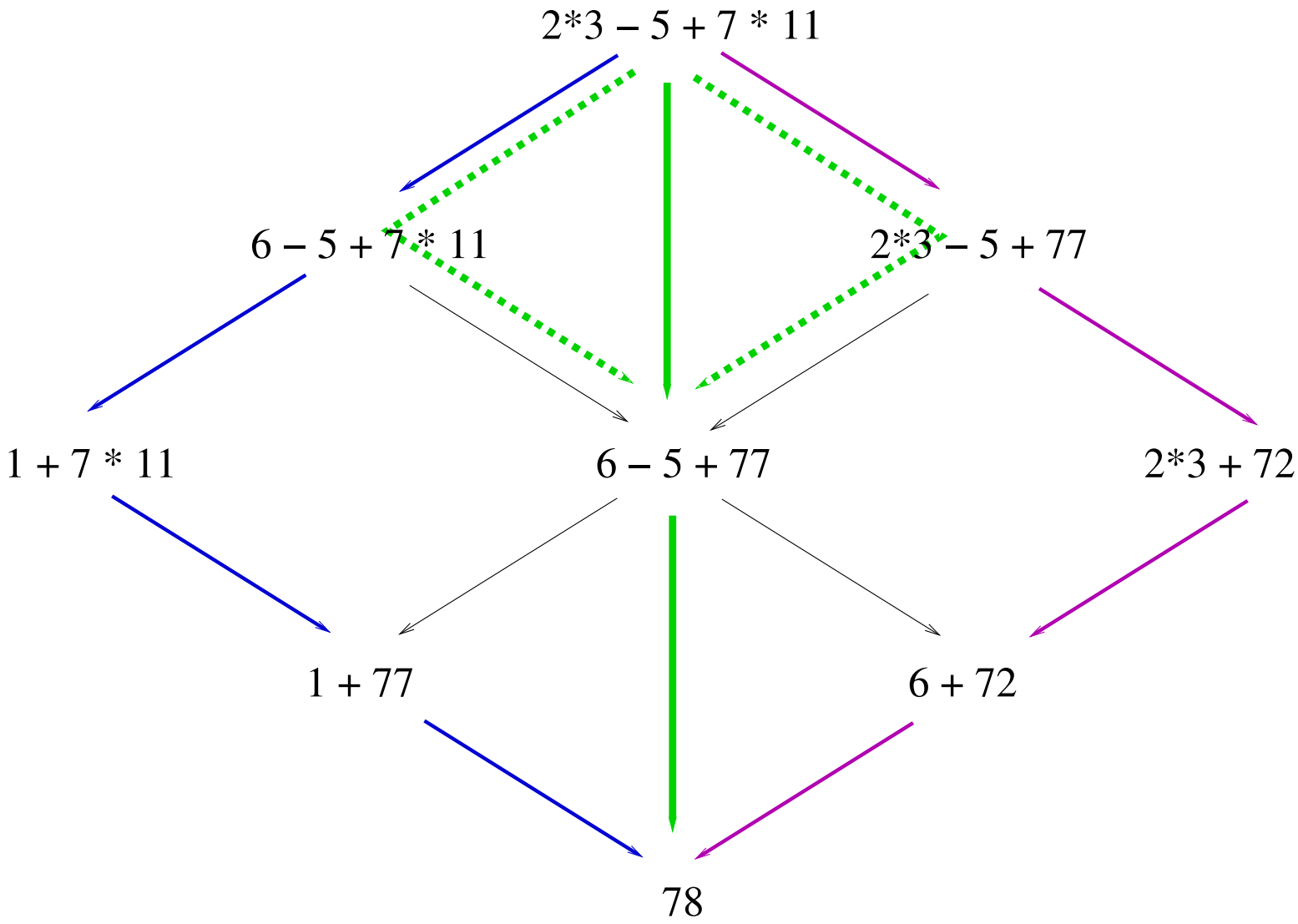
Graphische Veranschaulichung (4)



$\longrightarrow \ldots \longrightarrow$: Rechnen stets an willkürlich gewählter Stelle!

Vortr. III
Teil III
Kap. 7
Kap. 8
8.1
8.2
8.3
8.4
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Graphische Veranschaulichung (5)



...stets ist das Ergebnis gleich: Die sog. **Church-Rosser-** oder **Diamant-/Rauteneigenschaft!** (vgl. Theorem 8.3.1).

Vortr. III
Teil III
Kap. 7
Kap. 8
8.1
8.2
8.3
8.4
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Kapitel 8.2

Auswertung einfacher Ausdrücke mit Funktionstermen

Vortr. III
Teil III
Kap. 7
Kap. 8
8.1
8.2
8.3
8.4
Kap. 9
Umgekehr
Klassen-
zim-
mer II
Hinweis
Aufgabe

Auswertung von Funktionstermen

```
fac :: Integer -> Integer
fac n = if n == 0 then 1 else (n * fac (n - 1))
```

Der Funktionsterm `fac 2` hat als Semantik (oder: Bedeutung) den Wert `2`:

```
                          fac 2
(Expandieren)        ->>  if 2 == 0 then 1
                          else (2 * fac (2 - 1))
(Simplifizieren)     ->>  2 * fac (2 - 1)
                     ->>  ...
```

Für die Fortführung der Berechnung mit dem Funktionsterm `fac (2 - 1)` gibt es jetzt

- Freiheitsgrade und damit verschiedene Möglichkeiten.

...die beiden in der Praxis wichtigsten führen wir genauer aus.

Vortr. III
Teil III
Kap. 7
Kap. 8
8.1
8.2
8.3
8.4
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Zwei Hauptauswertungsvorgehensweisen

A): Applikativ – Argumentauswertung vor Expansion

```
                    2 * fac (2 - 1)
(Simplifizieren) ->> 2 * fac 1
(Expandieren)    ->> 2 * (if 1 == 0 then 1
                             else (1 * fac (1-1)))
                 ->> ... in diesem Stil fortfahren
```

B): Normal – Argumentauswertung nach Expansion

```
                    2 * fac (2 - 1)
(Expandieren)    ->> 2 * (if (2-1) == 0 then 1
                       else ((2-1) * fac ((2-1)-1)))
(Simplifizieren) ->> 2 * ((2-1) * fac ((2-1)-1))
                 ->> ... in diesem Stil fortfahren
```

Vortr. III
Teil III
Kap. 7
Kap. 8
8.1
8.2
8.3
8.4
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Vollständige Auswertung gemäß A)

```
fac n = if n == 0 then 1 else (n * fac (n - 1))

        fac 2
 (E) ->> if 2 == 0 then 1 else (2 * fac (2 - 1))
 (S) ->> if False then 1 else (2 * fac (2 - 1))
 (S) ->> (2 * fac (2 - 1))
 (S) ->> 2 * fac 1
 (E) ->> 2 * (if 1 == 0 then 1 else (1 * fac (1 - 1)))
 (S) ->> 2 * (if False then 1 else (1 * fac (1 - 1)))
 (S) ->> 2 * ((1 * fac (1 - 1)))
 (S) ->> 2 * (1 * fac 0)
 (E) ->> 2 * (1 * (if 0 == 0 then 1
                      else (0 * fac (0 - 1))))
 (S) ->> 2 * (1 * (if True then 1
                      else (0 * fac (0 - 1))))
 (S) ->> 2 * (1 * 1)
 (S) ->> 2 * 1
 (S) ->> 2
```

$\rightsquigarrow$ sog. applikativer Auswertungstil.

Vortr. III
Teil III
Kap. 7
Kap. 8
8.1
8.2
8.3
8.4
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Vollständige Auswertung gemäß B)

```
fac n = if n == 0 then 1 else (n * fac (n - 1))

        fac 2
 (E) ->> if 2 == 0 then 1 else (2 * fac (2 - 1))
 (S) ->> if False then 1 else (2 * fac (2 - 1))
 (S) ->> (2 * fac (2 - 1))
 (E) ->> 2 * (if (2-1) == 0 then 1
                 else ((2-1) * fac ((2-1)-1)))
(2S) ->> 2 * (if False then 1
                 else ((2-1) * fac ((2-1)-1)))
 (S) ->> 2 * (((2-1) * fac ((2-1)-1)))
 (S) ->> 2 * (1 * fac ((2-1)-1))
 (E) ->> 2 * (1 * (if ((2-1)-1) == 0 then 1
                   else ((2-1)-1) * fac (((2-1)-1)-1)))
(3S) ->> 2 * (1 * (if True then 1
                   else ((2-1)-1) * fac (((2-1)-1)-1)))
 (S) ->> 2 * (1 * 1)
(2S) ->> 2
```

$\rightsquigarrow$ sog. normaler Auswertungsstil.

Vortr. III
Teil III
Kap. 7
Kap. 8
8.1
8.2
8.3
8.4
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Kapitel 8.3

Zusammenfassung

Vortr. III
Teil III
Kap. 7
Kap. 8
8.1
8.2
8.3
8.4
Kap. 9
Umgekehr
Klassen-
zim-
mer II
Hinweis
Aufgabe

Zusammenfassung

...in Kapitel 8.1 und 8.2 haben wir anhand von Beispielen die Auswertung einfacher Ausdrücke

- ohne Funktionsterme
- mit Funktionstermen im
  - applikativen
  - normalen

  Auswertungsstil

als maximale (d.h. nicht mehr verlängerbare) Folgen von Simplifikations- und Expansions-Schritten vorgeführt.

Für alle Beispiele hat gegolten, dass die konkret gewählte Folge von Expansions- und Simplifikations-Schritten keinen Einfluss auf den Wert des jeweiligen Ausdrucks gehabt hat.

In Kapitel 12.3 und 13.3 werden wir sehen, dass dies stets gilt.

Vortr. III
Teil III
Kap. 7
Kap. 8
8.1
8.2
8.3
8.4
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Ausblick

...in Kapitel 12 und 13 werden wir weitergehend zeigen, dass dies auch dann stets gilt, wenn z.B.:

- Funktionsterme in komplexe Ausdrücke eingebettet sind
- applikativer und normaler Auswertungsstil gemischt werden
- in irgendeiner, weder applikativen noch normalen, terminierenden Reihenfolge ausgewertet wird.

Dieses für die Wohldefinierbarkeit der Semantik funktionaler Programmiersprachen zentrale Resultat geht auf Alonzo Church und John Barkley Rosser zurück, das wir hier im Vorgriff auf Kap. 12.3.4 und Kap. 13.3 anführen:

Theorem 8.3.1 (Church/Rosser, 1936)

Jede terminierende maximale Folge von Expansions- und Simplifikations-Schritten endet mit demselben Wert.

Vortr. III
Teil III
Kap. 7
Kap. 8
8.1
8.2
8.3
8.4
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Kapitel 8.4

Leseempfehlungen

Vortr. III
Teil III
Kap. 7
Kap. 8
8.1
8.2
8.3
8.4
Kap. 9
Umgekehr
Klassen-
zim-
mer II
Hinweis
Aufgabe

Basisleseempfehlungen für Kapitel 8

Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011. (Kapitel 4, Rekursion als Entwurfstechnik)

Paul Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, 2000. (Kapitel 1, Problem Solving, Programming, and Calculation)

Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 3. Auflage, 2011. (Kapitel 1, Introducing functional programming)

Vortr. III
Teil III
Kap. 7
Kap. 8
8.1
8.2
8.3
8.4
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Weiterführende Leseempfehlungen für Kap. 8

Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2007. (Kapitel 1, Introduction)

Peter Pepper. *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*. Springer-V., 2. Auflage, 2003. (Kapitel 9, Formalismen 1: Zur Semantik von Funktionen)

Vortr. III
Teil III
Kap. 7
Kap. 8
8.1
8.2
8.3
8.4
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Kapitel 9

Programmentwicklung, Programmverstehen

Vortr. III
Teil III
Kap. 7
Kap. 8
Kap. 9
9.1
9.2
9.3
Umgekehr
Klassen-
zim-
mer II
Hinweis
Aufgabe

Kapitel 9.1

Programmentwicklung

Exercitatio artem parat.
Übung verschafft Geschicklichkeit.

Tacitus (um 55 - um 120 n.Chr.)
röm. Geschichtsschreiber

Vortr. III
Teil III
Kap. 7
Kap. 8
Kap. 9
9.1
9.2
9.3
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Motivation

Das Finden eines algorithmischen Lösungsverfahrens ist

- kreativer Prozess
- nicht vollständig automatisierbar (siehe Verfahren für automatische Programmsynthese)

Es gibt jedoch

- Vorgehensweisen, Faustregeln

die die Aussicht erhöhen, erfolgreich zu sein.

Eine von Graham Hutton vorgeschlagene Vorgehensweise zur

- systematischen Entwicklung rekursiver Programme

betrachten wir hier genauer.

Vortr. III
Teil III
Kap. 7
Kap. 8
Kap. 9
9.1
9.2
9.3
Umgekehr
Klassen-
zim-
mer II
Hinweis
Aufgabe

Systematische Programmentwicklung

...für rekursive Programme als 5-schrittiger Prozess.

5-schrittiger Entwicklungsprozess (Graham Hutton, 2007):

1. Lege die (Daten-) Typen fest.
2. Führe alle relevanten Fälle auf.
3. Lege die Lösung für die einfachen (Basis-) Fälle fest.
4. Lege die Lösung für die übrigen Fälle fest.
5. Verallgemeinere und vereinfache das Lösungsverfahren.

Dieses Vorgehen werden wir an einem Beispiel demonstrieren.

Repetitio est mater studiorum.
Wiederholung ist die Mutter der Studien.
lat., sprichwörtl.

Vortr. III
Teil III
Kap. 7
Kap. 8
Kap. 9
9.1
9.2
9.3
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Bsp.: Aufsummieren

...der Elemente einer Liste ganzer Zahlen.

- Schritt 1: Lege die (Daten-) Typen fest

```
sum :: [Integer] -> Integer
```

- Schritt 2: Führe alle relevanten Fälle auf

```
sum []     =
sum (n:ns) =
```

- Schritt 3: Lege die Lösung für die Basisfälle fest

```
sum []     = 0
sum (n:ns) =
```

Vortr. III
Teil III
Kap. 7
Kap. 8
Kap. 9
9.1
9.2
9.3
Umgekehr
Klassen-
zim-
mer II
Hinweis
Aufgabe

Bsp. 1: Aufsummieren (fgs.)

- Schritt 4: Lege die Lösung für die übrigen Fälle fest

```
sum []     = 0
sum (n:ns) = n + sum ns
```

- Schritt 5: Verallgemeinere u. vereinfache das Lösungsverf.

```
5a) sum :: Num a => [a] -> a
```

```
5b) sum = foldr (+) 0
```

Gesamtlösung nach Schritt 5:

```
sum :: Num a => [a] -> a
sum = foldr (+) 0
```

Vortr. III
Teil III
Kap. 7
Kap. 8
Kap. 9
9.1
9.2
9.3
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Kapitel 9.2

Programmverstehen

Exercitatio optimus magister.
Übung ist der beste Lehrmeistert.

lat., sprichwörtl.

Vortr. III
Teil III
Kap. 7
Kap. 8
Kap. 9
9.1
9.2
9.3
Umgekehr
Klassen-
zim-
mer II
Hinweis
Aufgabe

Motivation

...eine Binsenweisheit:

- Programme werden häufiger gelesen als geschrieben!

Deshalb ist es wichtig

- Strategien

zu besitzen, die durch geeignete Vorgehensweisen und Fragen an ein Programm helfen, Programme

- zu lesen und zu verstehen, besonders fremde Programme.

Vortr. III
Teil III
Kap. 7
Kap. 8
Kap. 9
9.1
9.2
9.3
Umgekehr
Klassen-
zim-
mer II
Hinweis
Aufgabe

Vier Vorgehensweisen

...im Überblick:

1) Lesen des Programmtexts.
2) Nachdenken über das Programm, Ziehen von Schlussfolgerungen (z.B. Verhaltenshypothesen).

Zur Überprüfung von Verhaltenshypothesen, aber auch zu deren Auffinden kann hilfreich sein:

3) Gedankliche und/oder 'Papier- und Bleistift'-Programmausführung.

Auf einer konzeptuell anderen Ebene kann das Verständnis des Ressourcenbedarfs helfen, ein Programm zu verstehen:

4) Analyse des Zeit- und Speicherplatzverhaltens eines Programms.

Vortr. III
Teil III
Kap. 7
Kap. 8
Kap. 9
9.1
9.2
9.3
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Zur Illustration

...ein Beispiel:

```
mapWhile :: (a -> b) -> (a -> Bool) -> [a] -> [b]
mapWhile f p [] = []                                    (mW1)
mapWhile f p (x:xs)
  | p x         = f x : mapWhile f p xs                 (mW2)
  | otherwise = []                                      (mW3)
```

Vortr. III
Teil III
Kap. 7
Kap. 8
Kap. 9
9.1
9.2
9.3
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

1) Lesen des Programmtexts

Lesen der Funktionssignatur liefert bereits Einsichten in Art und Typ von Argumenten und Resultat. Im Beispiel:

- `mapWhile` erwartet als Argumente eine
  - Funktion `f` eines nicht weiter eingeschränkten Typs `(a -> b)`
  - Eigenschaft `p` von Objekten vom Typ `a`, genauer ein Prädikat (oder Wahrheitswertfunktion) vom Typ `(a -> Bool)`
  - Liste `l` von Elementen vom Typ `a`
- `mapWhile` liefert als Resultat
  - eine Liste l' von Elementen vom Typ `b`

...Lesen zusätzlich eingestreuter Programmkommentare, möglicherweise auch in Form von Vor- und Nachbedingungen ermöglicht (hoffentlich)

- weitere und tiefergehende Einsichten.

Vortr. III
Teil III
Kap. 7
Kap. 8
Kap. 9
9.1
9.2
9.3
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

1) Lesen des Programmtexts (fgs.)

Lesen der Funktionsdefinition liefert erste weitere Einsichten in Verhalten und Bedeutung des Programms. Im Beispiel:

- Angewendet auf die leere Liste `[]`, ist gemäß (mW1) das Resultat die leere Liste `[]`.
- Angewendet auf eine nichtleere Liste, deren Kopfelement `x` Eigenschaft `p` erfüllt, ist gemäß (mW2) das Element `(f x)` vom Typ `b` das Kopfelement der Resultatliste, deren Rest sich durch den rekursiven Aufruf auf die Restliste `xs` ergibt.
- Erfüllt Kopfelement `x` die Eigenschaft `p` nicht, bricht gemäß (mW3) die Berechnung ab und liefert als Resultat die leere Liste `[]` zurück.

Vortr. III
Teil III
Kap. 7
Kap. 8
Kap. 9
9.1
9.2
9.3
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

2) Nachdenken über das Programm

Nachdenken liefert tiefere Einsichten über Programmverhalten und -bedeutung, auch durch den Beweis von Eigenschaften, die das Programm besitzt. Im Beispiel:

- Für alle Funktionen `f`, Prädikate `p` und endliche Listen `xs` können wir folgende Gleichheiten beweisen:

  ```
  mapWhile f p xs = map f (takeWhile p xs)        (mW4)
  mapWhile f (const True) xs = map f xs           (mW5)
  mapWhile id p xs = takeWhile p xs               (mW6)
  ```

  wobei (mW5) und (mW6) Folgerungen aus (mW4) sind.

Vortr. III
Teil III
Kap. 7
Kap. 8
Kap. 9
9.1
9.2
9.3
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

3) Gedankliche, Papier- u. Bleistiftausführung

...hilft, Verhaltenshypothesen zu validieren oder zu generieren durch Berechnung der Funktionswerte für ausgewählte Argumente. Im Beispiel:

```
mapWhile (+1) (>=7) [8,12,7,3,16]
 ->> (8+1) : mapWhile (+1) (>=7) [12,7,3,16]    wg. (mW2)
 ->> 9 : (12+1) : mapWhile (+1) (>=7) [7,3,16]  wg. (mW2)
 ->> 9 : 13 : (7+1) : mapWhile (+1) (>=7) [3,16] wg. (mW2)
 ->> 9 : 13 : 8 : []                            wg. (mW3)
 ->> [9,13,8]

mapWhile (+1) (>=0) [8,12,7,3,16]
 ->> ...
 ->> [9,13,8,4,17]
```

Vortr. III
Teil III
Kap. 7
Kap. 8
Kap. 9
9.1
9.2
9.3
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

4) Analyse des Ressourcenverbrauchs

...des Programms liefert:

- Für das Zeitverhalten: Unter der Annahme, dass `f` und `p` jeweils in konstanter Zeit ausgewertet werden können, ist die Auswertung von `mapWhile` linear in der Länge der Argumentliste, da im schlechtesten Fall die gesamte Liste durchgegangen wird.
- Für das Speicherverhalten: Der Platzbedarf ist konstant, da das Kopfelement stets schon 'ausgegeben' werden kann, sobald es berechnet ist (siehe unterstrichene Resultatteile):

```
mapWhile (+1) (>=7) [8,12,7,3,16]
 ->> (8+1) : mapWhile (+1) (>=7) [12,7,3,16]
 ->> 9 : (12+1) : mapWhile (+1) (>=7) [7,3,16]
 ->> 9 : 13 : (7+1) : mapWhile (+1) (>=7) [3,16]
 ->> 9 : 13 : 8 : []
 ->> [9,13,8]
```

Vortr. III
Teil III
Kap. 7
Kap. 8
Kap. 9
9.1
9.2
9.3
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Zusammenfassung (1)

...jede der vorgestellten 4 Vorgangsweisen

- bietet einen anderen Zugang zum Verstehen eines Programms.
- liefert für sich einen Mosaikstein zu seinem Verstehen, aus denen sich durch Zusammensetzen ein vollständig(er)es Gesamtbild ergibt.
- kann 'von unten nach oben' auch auf Systeme von auf sich wechselweise abstützenden Funktionen angewendet werden.
- bietet mit Vorgangsweise (3) der gedanklichen oder Papier- und Bleistiftausführung eines Programms einen stets anwendbaren (Erst-) Zugang zum Erschließen der Programmbedeutung an.

Vortr. III
Teil III
Kap. 7
Kap. 8
Kap. 9
9.1
9.2
9.3
Umgekehr
Klassen-
zim-
mer II
Hinweis
Aufgabe

Zusammenfassung (2)

Lesbarkeit und Verständlichkeit eines Programms sollten

- immer schon beim Schreiben des Programms bedacht werden

...nicht zuletzt im eigenen Interesse!

Programme können grundsätzlich
auf zwei Arten geschrieben werden:
So einfach, dass sie offensichtlich keinen Fehler enthalten;
so kompliziert, dass sie keinen offensichtlichen Fehler enthalten.

C.A.R. 'Tony' Hoare (* 1934)

Turing Award Preisträger 1980

Vortr. III
Teil III
Kap. 7
Kap. 8
Kap. 9
9.1
9.2
9.3
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Kapitel 9.3

Leseempfehlungen

Vortr. III
Teil III
Kap. 7
Kap. 8
Kap. 9
9.1
9.2
9.3
Umgekehr
Klassen-
zim-
mer II
Hinweis
Aufgabe

Basisleseempfehlungen für Kapitel 9

Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2007. (Kapitel 6.6, Advice on Recursion)

Matthias Felleisen, Rober B. Findler, Matthew Flatt, Shriram Krishnamurthi. *How to Design Programs: An Introduction to Programming and Computing*. MIT Press, 2001.

Norman Ramsey. *On Teaching How to Design Programs*. In Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP 2014), 153-166, 2014.

Vortr. III
Teil III
Kap. 7
Kap. 8
Kap. 9
9.1
9.2
9.3
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Weiterführende Leseempfehlungen für Kap. 9

Hugh Glaser, Pieter H. Hartel, Paul W. Garrat. *Programming by Numbers: A Programming Method for Novices*. The Computer Journal 43(4):252-265, 2000.

Miran Lipovača. *Learn You a Haskell for Great Good! A Beginner's Guide*. No Starch Press, 2011. (Kapitel 10, Functionally Solving Problems)

Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 3. Auflage, 2011. (Kapitel 4, Designing and writing programs; Kapitel 7.4, Finding primitive recursive definitions; Kapitel 9.1, Understanding definitions; Kapitel 12.7, Understanding programs)

Vortr. III
Teil III
Kap. 7
Kap. 8
Kap. 9
9.1
9.2
9.3
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Umgekehrtes Klassenzimmer II

...zur Übung, Vertiefung

...nach Eigenstudium von Teil II 'Grundlagen':

- Zwar weiß ich viel...

Als Bonusthema, so weit die Zeit erlaubt:

- Programmierparadigmen: Fakt oder Fake?

Vortr. III
Teil III
Kap. 7
Kap. 8
Kap. 9
Umgekehrtes Klassenzimmer II
Zwar weiß ich viel...
Bonusthema: Programmierparadigmen: Fakt oder Fake?
Hinweis
Aufgabe

Zwar weiß ich viel...

doch möcht ich alles wissen.

Wagner, Assistent von Faust
Johann Wolfgang von Goethe (1749-1832)
dt. Dichter und Naturforscher

Vortr. III
Teil III
Kap. 7
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Zwar weiß ich viel...
Bonusthema: Program-mier-paradigmen: Fakt oder Fake?
Hinweis
Aufgabe

Zeit für Ihren Zweifel, Ihre Fragen!

Der Zweifel ist der Beginn der Wissenschaft.
Wer nichts anzweifelt, prüft nichts.
Wer nichts prüft, entdeckt nichts.
Wer nichts entdeckt, ist blind und bleibt blind.

Pierre Teilhard de Chardin (1881-1955)
franz. Jesuit, Theologe, Geologe und Paläontologe

Die großen Fortschritte in der Wissenschaft
beruhen oft, vielleicht stets, darauf, dass man
eine zuvor nicht gestellte Frage doch,
und zwar mit Erfolg, stellt.

Carl Friedrich von Weizsäcker (1912-2007)
dt. Physiker und Philosoph

...entdecken Sie den Wagner in sich!

Vortr. III
Teil III
Kap. 7
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Zwar weiß ich viel...
Bonusthema: Program-mier-paradigmen: Fakt oder Fake?
Hinweis
Aufgabe

Bonusthema

Programmierparadigmen: Fakt oder Fake?

Vortr. III
Teil III
Kap. 7
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Zwar weiß ich viel...
Bonusthema: Program-mier-paradigmen: Fakt oder Fake?
Hinweis
Aufgabe

Programmierparadigmen: Fakt?

...ein oft und meist mit großer Selbstverständlichkeit gebrauchter Begriff (s.a. Kapitel 1.2.1):

- Präskriptives Programmierparadigma
  - Prozedural (imperativ) (Algol, Pascal, Modula, C,...)
  - Objektorientiert (Simula, Smalltalk, Java, C++, Eiffel,...)
  - Parallel, datenparallel, verteilt (Ada, HPF, MesaF,...)
- Deklaratives Programmierparadigma
  - Funktional (Lisp, Miranda, Haskell, Gofer, ML,...)
  - Logisch (Prolog, Datalog, Gödel,...)
  - Beschränkung (Oz, Curry, Bertrand, Kaleidoscope,...)
- Multiprogrammierparadigmen
  - Funktional-logisch (Curry, POPLOG, TOY, Mercury,...),
  - Funktional-objektorientiert (Scala, Haskell++, OCaml,...)
  - Funktional-prozedural (Scheme,...)
  - Funktional-parallel (Erlang,...)
  - ...
- Datenflussprogrammierparadigma
- ...

Vortr. III
Teil III
Kap. 7
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Zwar weiß ich viel...
Bonusthema: Program-mier-paradigmen: Fakt oder Fake?
Hinweis
Aufgabe

Oder Fake?

Robert Harper. *What, if Anything, is a Programming Paradigm?* fifteeneightyfour bloc, CUP, 2017. `http://www.cambridgeblog.org/2017/05/what-if-anything-is-a-programming-paradigm`

Greg Michaelson. *Are There Programming Paradigms?* Hello World 4:32-33, 2018. `https://helloworld.raspberrypi.org/issues/4`

Vortr. III
Teil III
Kap. 7
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Zwar weiß ich viel...
Bonusthema: Program-mier-paradigmen: Fakt oder Fake?
Hinweis
Aufgabe

Paradigmen: Der Wunsch, Ordnung in eine

...komplexe und sich rasch entwickelnde Welt zu bringen:

Peter J. Landin. *The next 700 Programming Languages*. Communications of the ACM 9(3):157-166, 1966.

um folgendes Problem möglichst zum Abschluss zu bringen:

> For at least the last 60 years, programmers have been faced with the question: What programming language should I use?
>
> Myriad languages have been developed in the last six decades, with at least a few dozen in common usage today.

Jeffrey S. Foster
Shedding new Light on an old Language Debate.
Communications of the ACM 60(10):90, 2017.

Vortr. III
Teil III
Kap. 7
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Zwar weiß ich viel...
Bonusthema: Program-mier-paradigmen: Fakt oder Fake?
Hinweis
Aufgabe

Methode: Die Suche nach passenden

...Ordnungsprinzipien, die zu erklären und verstehen erlauben

- wie Programmiersprachen ihrem Kern nach unterschieden werden können
- auf welche Weise sie miteinander verbunden sind
- die für eine Aufgabe bestgeeignete Sprache auszuwählen.

Programmierparadigmen werden weithin als geeignetes Ordnungsprinzip angesehen. So etwa das

- objektorientierte Paradigma als theoretischer Überbau objektorientierter Programmiersprachen mit gemeinsamer objektorientierter Methodologie zur Identifikation und Manipulation von Objekten.
- funktionale Paradigma als theoretischer Überbau funktionaler Programmiersprachen mit gemeinsamer funktionaler Methodologie aus funktionaler Abstraktion, Funktionen höherer Ordnung, rekursiven Datenstrukturen, etc.
- ...

Vortr. III
Teil III
Kap. 7
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Zwar weiß ich viel...
Bonusthema: Program-mier-paradigmen: Fakt oder Fake?
Hinweis
Aufgabe

Greg Michaelson

...greift in einer aktuellen Arbeit:

Greg Michaelson. *Programming Paradigms, Turing Completeness and Computational Thinking*. The Art, Science, and Engineering of Programming 4(3), Article 4, 21 pages, 2020.

die Eignung des Programmierparadigmen-Ansatzes als Ordnungsprinzip und Erklärungsmodell systematisch auf; Anknüpfpunkt sind die grundlegenden Arbeiten von Thomas S. Kuhn zum Paradigmenbegriff in der Wissenschaft.

Der Zweifel ist der Beginn der Wissenschaft.
Wer nichts anzweifelt, prüft nichts.
Wer nichts prüft, entdeckt nichts.
Wer nichts entdeckt, ist blind und bleibt blind.

Pierre Teilhard de Chardin (1881-1955)
franz. Jesuit, Theologe, Geologe und Paläontologe

Vortr. III
Teil III
Kap. 7
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Zwar weiß ich viel...
Bonusthema: Program-mier-paradigmen: Fakt oder Fake?
Hinweis
Aufgabe

Thomas S. Kuhn

...hat die heutige Bedeutung des Paradigmenbegriffs als Ordnungsprinzip und Erkärungsmodell wissenschaftlicher Arbeit und Umbrüche maßgeblich geprägt, ausgeführt in:

Thomas S. Kuhn. *The Structure of Scientific Revolutions*. Chicago University Press, 1962.

Vortr. III
Teil III
Kap. 7
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Zwar weiß ich viel...
Bonusthema: Program-mier-paradigmen: Fakt oder Fake?
Hinweis
Aufgabe

Zwei Grundgedanken von Kuhn

1. Ein wissenschaftliches Paradigma ist
   - ein Korpus aus Theorie und Praxis zur Konzeptualisierung, Beschreibung und Erklärung der Welt.
   - gekennzeichnet und ausgezeichnet durch seine Dominanz in Theorie und Praxis zu einer Zeit.
2. Wissenschaft verläuft in (zwei Arten von) Phasen:
   - Stetig als sog. 'normale' Wissenschaft als Anwendung und Weiterentwicklung des und innerhalb des vorherrschenden Paradigmas ('Normalphasen').
   - Sprunghaft als Übergang von einem alten zu einem neuen Paradigma im Falle von Erklärungsfehlschlägen innerhalb des bisherigen Paradigmas ('Sprungphasen').

Vortr. III
Teil III
Kap. 7
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Zwar weiß ich viel...
Bonusthema: Program-mier-paradigmen: Fakt oder Fake?
Hinweis
Aufgabe

Beispiel aus der Physik/Astronomie

Der Übergang vom

- geozentrischen Ptolemäusschen Weltbild

zum

- heliozentrischen Kopernikanischen Weltbild

im 16. und 17. Jahrhundert aufgrund neuer Beobachtungen dank neuer Instrumente (Galileiische Jupitermonde, Fernrohr).

Vortr. III
Teil III
Kap. 7
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Zwar weiß ich viel...
Bonusthema: Program-mier-paradigmen: Fakt oder Fake?
Hinweis
Aufgabe

Nach Kuhn

...zeichnen sich neuere gegenüber älteren Paradigmen aus durch:

- höhere Erklärungskraft: Beobachtungen können erklärt werden, die in älteren Paradigmen nicht erklärt werden können.

  Beispiele:

  - Die Existenz und Laufbahnen von Jupitermonden im Ptolemäusschen und Kopernikanischen Weltbild.
  - Der Schwerkrafteinfluss auf Licht in Newtonscher (mechanischer) und Einsteinscher (relativistischer) Physik.

Folgerungen:

- Verschiedene Paradigmen können in Phasen 'normaler' Wissenschaft nicht (sinnvoll) gleichzeitig co-existieren, jedoch von einem gemeinsamen Ursprung aus unterschiedlich weiter entwickelte Ausformungen, sog. 'Traditionen'.
- Paradigmen lösen sich zeitlich aufeinanderfolgend ab.

Vortr. III
Teil III
Kap. 7
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Zwar weiß ich viel...
Bonusthema: Program-mier-paradigmen: Fakt oder Fake?
Hinweis
Aufgabe

Michaelsons zentrale Frage

...ist es im Kontext von

- Programmierung, Programmiersprachen

im Sinn des

- Kuhnschen Paradigmenbegriffs

angemessen von

- Programmierparadigmen

zu sprechen? Oder ist es unangemessen? Eine Überhöhung?

Vortr. III
Teil III
Kap. 7
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Zwar weiß ich viel...
Bonusthema: Program-mier-paradigmen: Fakt oder Fake?
Hinweis
Aufgabe

Michaelsons Frage aufgebrochen

Zeitliche Aufeinanderfolge von Programmierparadigmen?

- Nein. Imperative, objektorientierte, funktionale, logische, etc. Programmierung existieren nebeneinander (und sind etwa zeitgleich erdacht worden).

Höhere Erklärungskraft einzelner Programmierparadigmen?

- Nein. Imperative, objektorientierte, funktionale, logische, etc. Programmierung sind Turing-vollständig (s.a. Kapitel 12.1) und sind in diesem Sinn von selber Erklärungskraft, erklären dieselbe Welt des Berechenbaren, sind Operationalisierungen desselben Berechenbarkeitsbegriffs.

Dominanz eines Programmierparadigmas?

- Nein. Allenfalls in Teilbereichen (Wirtschaft, Wissenschaft, etc.) oder Teilgebieten (Systemprogrammierung, Webprogrammierung, etc.) bestimmte Stile weiter verbreitet als andere.

Vortr. III
Teil III
Kap. 7
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Zwar weiß ich viel...
Bonusthema: Program-mier-paradigmen: Fakt oder Fake?
Hinweis
Aufgabe

Michaelsons Folgerung

...ausgehend von der gleichen Erklärungskraft für die Welt des Berechenbaren als

- Turing-Berechenbares

ist statt von Programmierparadigmen (wie auch schon vor ihm oft gemacht) besser von

- Programmierstilen oder (i.S.v. Kuhn) Programmiertraditionen

zu sprechen, die als

- spezielle Ausformungen des Turing-Berechenbaren

innerhalb verschiedener informatischer Teilgemeinden ausgeformt worden sind und tradiert werden mit

- Rechnerisches Denken (engl. computational thinking)

als gemeinsamer überspannender (Programmier-) Methodologie.

Vortr. III
Teil III
Kap. 7
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Zwar weiß ich viel...
Bonusthema: Program-mier-paradigmen: Fakt oder Fake?
Hinweis
Aufgabe

Daraus abgeleiteter Vorschlag von Michaelson

Ist nach Kuhn ein wissenschaftliches Paradigma gegeben durch:

- Korpus aus Theorie und Praxis

dann ist das heutige

- wissenschaftliche Paradigma der Programmierung

gegeben durch folgenden Korpus aus Theorie und Praxis:

- Theoretischer Korpus: Theorie der Turing-Vollständigkeit als (programmier-) theoretische Erklärung des Berechenbaren.
- Praktischer Korpus: Rechnerisches Denken (computational thinking) als Programmiermethodologie (als derzeit bestem Kandidaten für den praktischen Korpus).

Vortr. III
Teil III
Kap. 7
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Zwar weiß ich viel...
Bonusthema: Program-mier-paradigmen: Fakt oder Fake?
Hinweis
Aufgabe

Innerhalb dieses einen Programmierparadigmas

...unterschiedliche Programmierstile oder Programmiertraditionen, gleichsam

- Polarisationsbrillen

durch die nach Aufsetzen die Probleme dieser Welt so ausschauen, als ließen sie sich am besten

- prozedural
- objektorientiert
- funktional
- logisch
- ...

modellieren und lösen (s.a. Anhang A.9).

Vortr. III
Teil III
Kap. 7
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Zwar weiß ich viel...
Bonusthema: Program-mier-paradigmen: Fakt oder Fake?
Hinweis
Aufgabe

Programmierparadigmen: Eine Spurensuche

...nach Michaelson (hier nur ausgewählte Spuren).

Robert W. Floyd. *The Paradigms of Programming*. Turing Award Lecture. Computer Journal 22(8):455-460, 1979. (doi:10.1145/359138.359140)

...expliziter Bezug auf die Arbeiten von Kuhn; identifiziert strukturierte Programmierung als systematische schrittweise Verfeinerung von grober Spezifikation zu schließlicher Implementierung als dominantes 'Informatik-Paradigma'.

Norman E. Gibbs, Allen B. Tucker. *A Model Curriculum for a Liberal Arts Degree in Computer Science*. Communications of the ACM 29(3):202-210, 1986. (doi:10.1145/5666.5667)

...untersuchen Kernprinzipien der Programmierung, darunter prozedurale, funktionale, objektorientierte, logische, modulare und Datenfluss-Programmierung als Ausprägungen sog. Programmierstile.

Vortr. III
Teil III
Kap. 7
Kap. 8
Kap. 9
Umgekehrte Klassenzimmer II
Zwar weiß ich viel...
Bonusthema: Programmierparadigmen: Fakt oder Fake?
Hinweis
Aufgabe

Spurensuche fortgesetzt

David A. Watt. *Programming Language Concepts and Paradigms*. Prentice Hall, 1990.

...eines der ersten Lehrbücher, das den Paradigmenbegriff aufgreift.

Nach wie vor gilt die Aussage von:

Peter Wegner. Guest Editor's Introduction to Special Issue of Computing Surveys on Programming Language Paradigms. ACM Computing Surveys 21(3):253-258, 1989.

Zitat: "Computer Science has grown so fast and fashions within computer science change so rapidly, that it is more difficult to identify its dominant paradigms or characterise its essence than in established scientific disciplines like physics or mathematics." (S. 257)

Vortr. III
Teil III
Kap. 7
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Zwar weiß ich viel...
Bonusthema: Program-mier-paradigmen: Fakt oder Fake?
Hinweis
Aufgabe

Übungsaufgabe 1

1. Welche anderen Ordnungsprinzipien neben Paradigmen und Stilen für Programmiersprachen und/oder Programmierung kennen Sie?
2. Führen Sie diese Ordnungsprinzipien in einer gewissen Detaillierung aus, unterlegt auch mit geeigneten Beispielen.

Lassen Sie sich ggf. vom eingangs zitierten Artikel von Greg Michaelson inspirieren.

3. Welcher von Turing-Vollständigkeit abgelöste Berechenbarkeitsbegriff könnte als Paradigmenwechsel in der Welt der Berechenbarkeit im Sinn von Kuhn gesehen werden?

Tipp: Siehe Kapitel 12.1.

Vortr. III
Teil III
Kap. 7
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Zwar weiß ich viel...
Bonusthema: Program-mier-paradigmen: Fakt oder Fake?
Hinweis
Aufgabe

Übungsaufgabe 2

1. Die Analytische Maschine von Charles Babbage
2. Analogrechner, analoges Rechnen
3. Quantum-Computer, Quantum-Rechnen

sind naheliegende Kandidaten für schwächere oder stärkere Berechenbarkeitsbegriffe als Turing-Vollständigkeit.

Was wissen/vermuten Sie, ob tatsächlich schwächere oder stärkere Berechenbarkeitsbegriffe induziert werden? Überprüfen Sie Ihr Wissen/Vermutung in:


Charles Babbage. *Passages from the Life of a Philosopher*. In Phillip Morrison und Emily Morrison (Hrsg.). *Charles Babbage and his Calculating Engines*. Dover, 1961.


Daniel S. Graca, Jose S. Costa. *Analog Computers and Recursive Functions over the Reals*. Journal of Complexity 19(5):644-664, 2003. doi:10.1016/S0885-064X(03)00034-7


David Deutsch. *Quantum Theory, the Church-Turing Principle and the Universal Quantum Computer*. Proceedings of the Royal Society A, 400(1818), 1985. doi:10.1098/rspa.1985.0070

Vortr. III
Teil III
Kap. 7
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Zwar weiß ich viel...
Bonusthema: Program-mier-paradigmen: Fakt oder Fake?
Hinweis
Aufgabe

Übungsaufgabe 3

Strukturierte Programmierung wie von Floyd in

Robert W. Floyd. *The Paradigms of Programming*. Turing Award Lecture. Computer Journal 22(8):455-460, 1979. (doi:10.1145/359138.359140)

als seinerzeit dominantes Paradigma vorgeschlagen, fußt auf

- sequentieller Komposition
- Auswahl
- Wiederholung

und gilt im Kern prozeduraler/imperativer Programmiermethodik zugehörig.

Zeigen Sie, dass sich Vorgehen und konstituierende Bestandteile strukturierter Programmierung auch in der Methodologie

- objektorientierter
- funktionaler
- ...

Programmierung wiederfinden.

Vortr. III
Teil III
Kap. 7
Kap. 8
Kap. 9
Umgekehrte Klassenzimmer II
Zwar weiß ich viel...
Bonusthema: Programmierparadigmen: Fakt oder Fake?
Hinweis
Aufgabe

Hinweis

...für das Verständnis von Vorlesungsteil III ist eine über den unmittelbaren Inhalt von Vortrag III hinausgehende weitergehende und vertiefende Beschäftigung mit dem Stoff nötig; siehe:

- vollständige Lehrveranstaltungsunterlagen

...verfügbar auf der Webseite der Lehrveranstaltung:

`http://www.complang.tuwien.ac.at/knoop/fp185A05_ws2021.html`

Vortr. III
Teil III
Kap. 7
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe

Aufgabe bis Mittwoch, 04.11.2020

...selbstständiges Durcharbeiten von Teil III 'Applikative Programmierung', Kap. 7 bis Kap. 9 und von Leit- und Kontrollfragenteil III zur Selbsteinschätzung und als Grundlage für die umgekehrte Klassenzimmersitzung am 04.11.2020:

Vortrag, umgek. Klassenz.	Thema Vortrag	Thema umgek. Klassenz.
Di, 06.10.2020, 08:15-09:45	Teil I	n.a. / Vorbesprechung
Di, 13.10.2020, 08:15-09:45	Teil II	Teil I
Di, 27.10.2020, 08:15-09:45	**Teil III**	**Teil II**
Mi, 04.11.2020, 08:15-09:45	**Teil IV**	**Teil III**
Mi, 18.11.2020, 08:15-09:45	Teil V	Teil IV
Mi, 02.12.2020, 08:15-09:45	Teil VI	Teil V
Mi, 16.12.2020, 08:15-09:45	Teil VII	Teil VI

Vortr. III
Teil III
Kap. 7
Kap. 8
Kap. 9
Umgekehr Klassen-zim-mer II
Hinweis
Aufgabe