Funktionale Programmierung

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

Vortrag VII
Orientierung, Einordnung
16.12.2020

Jens Knoop

Technische Universität Wien
Information Systems Engineering
Compilers and Languages

Vortr. VII
Teil VI
Kap. 18
Teil VII
Kap. 19
Anhänge
A
Umgekehr Klassen-zim-mer VI
Hinweis
Viel Erfolg!

Vortrag VII
Orientierung, Einordnung

...zum selbstgeleiteten, eigenständigen Weiterlernen.

Teil VI: Weiterführende Konzepte

– Kapitel 18: Allgemeine und fkt. Programmierprinzipien

Teil VII: Abschluss

– Kapitel 19: Rückschau, Ausschau

Anhänge

– A: Schlaglichter: Imperative vs. fkt. Programmierung
  – A.8: Problem- & Lösungssicht: Imperativ vs. funktional
  – A.9: Wer bin ich? Welcher Problemlösungstyp bin ich?

Teil VI

Weiterführende Konzepte

Vortr. VII
Teil VI
Kap. 18
Teil VII
Kap. 19
Anhänge
A
Umgekehr Klassen-zimmer VI
Hinweis
Viel Erfolg!

Kapitel 18

Allgemeine und funktionale Programmierprinzipien

Vortr. VII
Teil VI
Kap. 18
18.1
18.2
18.3
18.4
18.5
Teil VII
Kap. 19
Anhänge
A
Umgekehr Klassenzimmer VI
Hinweis
Viel Erfolg!

Kapitel 18.1

Überblick, Orientierung

Vortr. VII
Teil VI
Kap. 18
18.1
18.2
18.3
18.4
18.5
Teil VII
Kap. 19
Anhänge
A
Umgekehr Klassen-zimmer VI
Hinweis
Viel Erfolg!

Im Mittelpunkt: Drei Themen

Allgemeine Programmierprinzipien

1. Stetes Hinterfragen u. Anpassen seines Tuns (Kap. 18.2)
   – im Wege reflektiven Programmierens!

   Illustriert anhand von Schlüssel- und Leitfragen.

Funktionale Programmierprinzipien

2. Kapseln algorithmischen Vorgehens (Kap. 18.3)
   – möglich dank Funktionen höherer Ordnung!

   Illustriert anhand von Teile und Herrsche.
3. Problemorientiertes Modularisieren (Kap. 18.4)
   – möglich dank später Auswertung! (engl. lazy evaluation)

   Generatiere/anpass-Modularisierungen (selektieren, filtern, transformieren,...)

   Illustriert anhand von Stromprogrammierung (Ströme, unendliche Listen (engl. streams, lazy lists)).

Vortr. VII
Teil VI
Kap. 18
18.1
18.2
18.3
18.4
18.5
Teil VII
Kap. 19
Anhänge
A
Umgekehrt Klassenzimmer VI
Hinweis
Viel Erfolg!

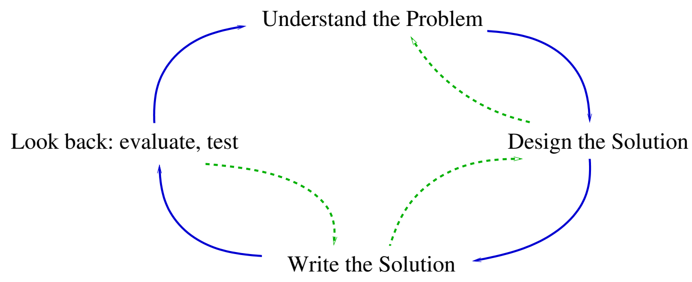
Kapitel 18.2

Reflektives Programmieren

Vortr. VII
Teil VI
Kap. 18
18.1
18.2
18.3
18.4
18.5
Teil VII
Kap. 19
Anhänge
A
Umgekehr Klassen-zim-mer VI
Hinweis
Viel Erfolg!

Reflektives Programmieren

...der Programm-Entwicklungszyklus nach Simon Thompson, Haskell: The Craft of Fuctional Programming, 2. Auflage, 1999, Kap. 11 'Reflective Programming':



...in jeder der 4 Phasen ist es nützlich, (sich) Fragen zu stellen, zu beantworten und den Lösungsweg ggf. anzupassen.

Nosce te ipsum, nosce tuum opus.
Erkenne dich selbst, erkenne dein Werk.
lat., sprichw., Apoll zugeschrieben, abgewandelt

Vortr. VII
Teil VI
Kap. 18
18.1
18.2
18.3
18.4
18.5
Teil VII
Kap. 19
Anhänge
A
Umgekehrtes Klassenzimmer VI
Hinweis
Viel Erfolg!

Schlüssel- und Leitfragen

Phase 1: Verstehen des Problems

- Welches sind die Ein- und Ausgaben des Problems?
- Welche Randbedingungen sind einzuhalten?
- Ist das Problem über- oder unterspezifiziert?
- ...

Phase 2: Entwerfen einer Lösung

- Ist das Problem verwandt zu (mir) bekannten anderen, möglicherweise einfacheren Problemen?
- Wenn ja, lassen sich deren Lösungsideen anpassen und anwenden? Ebenso deren Implementierungen, vorhandene Bibliotheken?
- Lässt sich das Problem verallgemeinern und so möglicherweise sogar einfacher lösen?
- ...

Vortr. VII
Teil VI
Kap. 18
18.1
18.2
18.3
18.4
18.5
Teil VII
Kap. 19
Anhänge
A
Umgekehrtes Klassenzimmer VI
Hinweis
Viel Erfolg!

Schlüssel- und Leitfragen

Phase 3: Ausformulieren und kodieren der Lösung

- Gibt es passende Bibliotheken, speziell geeignete polymorphe Funktionen höherer Ordnung für die Lösung von Teilproblemen?
- Können vorhandene Bibliotheksfunktionen (zumindest) als Vorbild dienen, um entsprechende Funktionen für eigene Datentypen zu definieren?
- ...

Phase 4: Blick zurück, evaluieren, testen

- Lässt sich die Lösung testen, ihre Korrektheit beweisen, auch formal?
- Erfüllt das Programm auch nichtfunktionale Eigenschaften gut wie Performanz, Speicherverbrauch, Skalierbarkeit, Verständlichkeit, Änder- und Erweiterbarkeit?
- ...

Vortr. VII
Teil VI
Kap. 18
18.1
18.2
18.3
18.4
18.5
Teil VII
Kap. 19
Anhänge
A
Umgekehrtes Klassenzimmer VI
Hinweis
Viel Erfolg!

Kapitel 18.3

Kapseln algorithmischen Vorgehens

Vortr. VII
Teil VI
Kap. 18
18.1
18.2
18.3
18.4
18.5
Teil VII
Kap. 19
Anhänge
A
Umgekehr Klassenzimmer VI
Hinweis
Viel Erfolg!

Beispiel: Teile und Herrsche

...die zugrundeliegende algorithmische Idee dieses Vorgehens:

1. Ist ein Problem einfach genug, löse es sofort.
2. Wenn nicht: Teile das Problem rekursiv in kleinere Teilprobleme, bis alle Teilprobleme einfach genug sind, sofort gelöst werden zu können.
3. Berechne die Lösung des ursprünglichen Problems aus den Lösungen der Teilprobleme.

...typisches Beispiel einer von-oben-nach-unten-Vorgehensweise (engl. top-down)!

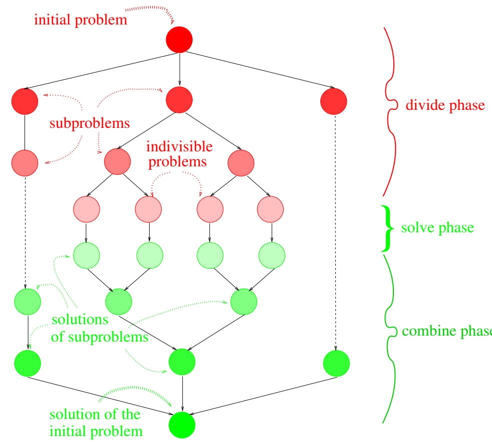
Divide et impera.
Teile und herrsche.

nach Ludwig XI von Frankreich (1423-1483)

Vortr. VII
Teil VI
Kap. 18
18.1
18.2
18.3
18.4
18.5
Teil VII
Kap. 19
Anhänge
A
Umgekehr Klassenzimmer VI
Hinweis
Viel Erfolg!

Veranschaulichung

...der Phasenabfolge des 'Teile und Herrsche'-Vorgehens:



Fethi Rabhi, Guy Lapalme.
Algorithms: A Functional Programming Approach.
Addison-Wesley, 1999, Seite 156.

Vortr. VII
Teil VI
Kap. 18
18.1
18.2
18.3
18.4
18.5
Teil VII
Kap. 19
Anhänge
A
Umgekehr
Klassen-
zim-
mer VI
Hinweis
Viel
Erfolg!

Typische Anwendungsfelder, Anwendungen

...für ein 'Teile und Herrsche'-Vorgehen:

- Sortieren (Quicksort, Mergesort, etc.)
- Numerische Analyse(verfahren)
- Kryptographie
- Bildverarbeitung
- Binomialkoeffizientenberechnung
- ...

Vortr. VII
Teil VI
Kap. 18
18.1
18.2
18.3
18.4
18.5
Teil VII
Kap. 19
Anhänge
A
Umgekehr Klassen-zim-mer VI
Hinweis
Viel Erfolg!

Vorbereitung zur Vorgehenskapselung

...in einer Funktion höherer Ordnung.

Gegeben:

- Ein Problem mit Probleminstanzen eines generischen Typs, beschrieben durch die Typvariable `pb`.

Gesucht:

- Eine Lösung aus einer Menge von Lösungsinstanzen eines generischen Typs, beschrieben durch die Typvariable `lsg`.

Kapselungsziel:

Eine Funktion höherer Ordnung `teile_und_herrsche`, die geeignet parametrisiert für

- Probleminstanzen vom Typ `pb` gemäß des 'Teile und Herrsche'-Prinzips eine Lösungsinstanz vom Typ `lsg` berechnet.

Vortr. VII
Teil VI
Kap. 18
18.1
18.2
18.3
18.4
18.5
Teil VII
Kap. 19
Anhänge
A
Umgekehr Klassen-zim-mer VI
Hinweis
Viel Erfolg!

Die Bedeutung der Parameter

...der Funktion höherer Ordnung `teile_und_herrsche`:

- `einfach_genug :: pb -> Bool`: ...liefert `True`, falls die Probleminstanz einfach genug ist, um sofort gelöst werden zu können.
- `loese :: pb -> lsg`: ...liefert die Lösungsinstanz einer unmittelbar lösbaren Probleminstanz.
- `teile :: pb -> [pb]`: ...teilt eine nicht unmittelbar lösbare Probleminstanz in eine Liste von Teilprobleminstanzen auf.
- `herrsche :: pb -> [lsg] -> lsg`: ...liefert angewendet auf eine Ausgangsprobleminstanz und eine Liste von Lösungen von Teilprobleminstanzen die Lösung der Ausgangsprobleminstanz.

Einführung sprechender Typsynonyme

...für die vorkommenden Funktionstypen:

```
type Einfach_genug pb     = pb -> Bool
type Loese pb lsg         = pb -> lsg
type Teile pb             = pb -> [pb]
type Herrsche pb lsg      = pb -> [lsg] -> lsg
```


Vortr. VII
Teil VI
Kap. 18
18.1
18.2
18.3
18.4
18.5
Teil VII
Kap. 19
Anhänge
A
Umgekehrt Klassenzimmer VI
Hinweis
Viel Erfolg!

Das Funktional teile_und_herrsche

```
teile_und_herrsche :: (Einfach_genug pb) -> (Loese pb lsg)
     -> (Teile pb) -> (Herrsche pb lsg) -> pb -> lsg
teile_und_herrsche einfach_genug loese teile herrsche
                   pb_instanz
 = loesung_von pb_instanz
   where
    loesung_von p
     | einfach_genug p = loese p
     | otherwise
        = herrsche p (map loesung_von (teile p))
```

(map loesung_von (teile p)): Löse rekursiv alle durch die Teilung entstehenden Probleme.

Vortr. VII
Teil VI
Kap. 18
18.1
18.2
18.3
18.4
18.5
Teil VII
Kap. 19
Anhänge
A
Umgekehrtes Klassenzimmer VI
Hinweis
Viel Erfolg!

Anwendung: Teile und Herrsche für Quicksort

```
quickSort :: Ord a => [a] -> [a]
quickSort liste
 = teile_und_herrsche einfach_genug loese teile
                      herrsche liste
 where
  einfach_genug ls          = length ls <= 1
  loese                     = id
  teile (l:ls)              = [[ x | x <- ls, x <= l],
                                [ x | x <- ls, x > l]]
  herrsche (l:_) [ls1,ls2] = ls1 ++ [l] ++ ls2
```


Vortr. VII
Teil VI
Kap. 18
18.1
18.2
18.3
18.4
18.5
Teil VII
Kap. 19
Anhänge
A
Umgekehr Klassen-zim-mer VI
Hinweis
Viel Erfolg!

Aber Achtung!

...nicht jedes Problem, das dem 'teile und herrsche'-Vorgehen in natürlicher Weise zugänglich ist, ist auch dafür geeignet naiv umgesetzt.

Betrachte dazu:

```
fib :: Integer -> Integer
fib n = teile_und_herrsche einfach_genug loese
                           teile herrsche n
   where
    einfach_genug n     = (n == 0) || (n == 1)
    loese               = id
    teile n             = [n-2,n-1]
    herrsche _ [m1,m2]  = m1 + m2
```

...besitzt exponentielles Laufzeitverhalten!

Vortr. VII
Teil VI
Kap. 18
18.1
18.2
18.3
18.4
18.5
Teil VII
Kap. 19
Anhänge
A
Umgekehr Klassen-zim-mer VI
Hinweis
Viel Erfolg!

Kapitel 18.4

Problemorientiertes Modularisieren

Vortr. VII
Teil VI
Kap. 18
18.1
18.2
18.3
18.4
18.5
Teil VII
Kap. 19
Anhänge
A
Umgekehr
Klassen-
zim-
mer VI
Hinweis
Viel
Erfolg!

Beispiel: Generiere/anpass-Modularisierungen

...in fkt. Programmiersprachen mit später Auswertung.

Darunter:

- Generiere/selektiere-Modularisierungen
- Generiere/filtere-Modularisierungen
- Generiere/transformiere-Modularisierungen
- Generiere/... -Modularisierungen

mit denen sich viele Probleme elegant, knapp und effizient lösen lassen.

...illustriert im folgenden am Beispiel der Programmierung mit Strömen, programmiersprachlichem Jargon für die Programmierung mit (potentiell)

- unendlichen Listen (engl. streams, lazy lists).

Generator- und Anpassmodule

...am Beispiel des Siebs des Eratosthenes zur Berechnung des Stroms der Primzahlen:

1. Schreibe alle natürlichen Zahlen ab 2 hintereinander auf.
2. Die kleinste nicht gestrichene Zahl in dieser Folge ist eine Primzahl. Streiche alle Vielfachen dieser Zahl.
3. Wiederhole Schritt 2 mit der kleinsten jeweils noch nicht gestrichenen Zahl.

```
Nach Schritt 1:
 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19...
Nach Schritt 2 für Zahl 2:
 2 3   5   7   9    11    13    15    17    19...
Nach Schritt 2 für Zahl 3:
 2 3   5   7        11    13          17    19...
usw.
```

Vortr. VII
Teil VI
Kap. 18
18.1
18.2
18.3
18.4
18.5
Teil VII
Kap. 19
Anhänge
A
Umgekehrtes Klassenzimmer VI
Hinweis
Viel Erfolg!

primes: Das Generatormodul

...für den Strom der Primzahlen.

```
primes :: [Integer]
primes = sieve [2..]

sieve :: [Integer] -> [Integer]
sieve (x:xs) = x : sieve [y | y <- xs, mod y x > 0]
```

Die (0-stellige) Generatorfunktion primes liefert

- den Strom der (unendlich vielen) Primzahlen.

Aufruf von primes:

```
primes ->> [2,3,5,7,11,13,17,19,23,29,31,37,41,...
```

Vortr. VII
Teil VI
Kap. 18
18.1
18.2
18.3
18.4
18.5
Teil VII
Kap. 19
Anhänge
A
Umgekehr Klassen-zim-mer VI
Hinweis
Viel Erfolg!

Veranschaulichung

...der Stromberechnung durch händische Auswertung:

```
    primes
->> sieve [2..]
->> 2 : sieve [y | y <- [3..], mod y 2 > 0]
->> 2 : sieve (3 : [y | y <- [4..], mod y 2 > 0]
->> 2 : 3 : sieve [z | z <- [ y | y <- [4..],
                                   mod y 2 > 0],
                                   mod z 3 > 0]
->> ...
->> 2 : 3 : sieve [z | z <- [5, 7, 9..],
                         mod z 3 > 0]
->> ...
->> 2 : 3 : sieve [5, 7, 11,...
->> ...
```

Vortr. VII
Teil VI
Kap. 18
18.1
18.2
18.3
18.4
18.5
Teil VII
Kap. 19
Anhänge
A
Umgekehrt Klassenzimmer VI
Hinweis
Viel Erfolg!

Generatormodule

...kombiniert mit Anpassmodulen ermöglichen neue, problem-orientierte Modularisierungen:

Insbesondere:

- Generiere/selektiere- (G/S-) Modularisierungen
- Generiere/filtere- (G/F-) Modularisierungen
- Generiere/transformiere- (G/T-) Modularisierungen
- ...

sowie in natürlicher Weise Kombinationen davon wie G/T/S-, G/T/F-Modularisierungen, etc.

Vortr. VII
Teil VI
Kap. 18
18.1
18.2
18.3
18.4
18.5
Teil VII
Kap. 19
Anhänge
A
Umgekehr Klassen-zim-mer VI
Hinweis
Viel Erfolg!

G/S-Modul. am Bsp. des Primzahlstroms (1)

Ein Generatormodul (G):

- ```
  gen_Primes :: [Integer]
  gen_Primes = primes
  ```

Viele Selektormodule (S):

- Nimm die ersten n Elemente einer Liste:
  ```
  take :: Int -> [a] -> [a]
  take n ls = ...
  ```
- Nimm das $(n+1)$-te Element einer Liste, $n \geq 0$:
  ```
  !! :: [a] -> Int -> a
  (!!) ls n = ...
  ```
- Nimm alle ab dem $(n+1)$-ten Element einer Liste:
  ```
  drop :: Int -> [a] -> [a]
  drop n ls = ...
  ```
- ...

Vortr. VII
Teil VI
Kap. 18
18.1
18.2
18.3
18.4
18.5
Teil VII
Kap. 19
Anhänge
A
Umgekehrt
Klassen-zimmer VI
Hinweis
Viel Erfolg!
  ```

G/S-Modul. am Bsp. des Primzahlstroms (2)

Zusammenfügen der G/S-Module zum Gesamtprogramm:

- Anwendung der G/S-Modularisierung:

  Die ersten 5 Primzahlen:

  ```
  take 5 gen_Primes ->> [2,3,5,7,11]
  ```

- Anwendung der G/S-Modularisierung:

  Die 5-te Primzahl:

  ```
  (!!) 4 gen_Primes ->> gen_Primes!!4 ->> 11
  ```

- Anwendung der G/S-Modularisierung:

  Die 6-te bis 10-te Primzahl:

  ```
  take 5 (drop 5 gen_Primes) ->> [13,17,19,23,29]
  ```


Vortr. VII
Teil VI
Kap. 18
18.1
18.2
18.3
18.4
18.5
Teil VII
Kap. 19
Anhänge
A
Umgekehrtes Klassenzimmer VI
Hinweis
Viel Erfolg!

G/F-Modul. am Bsp. des Primzahlstroms (1)

Ein Generatormodul (G):

- ```
  gen_Primes :: [Integer]
  gen_Primes = primes
  ```

Viele Filtermodule (F):

- Alle Listenelemente größer als 1000:
  ```
  filter (>1000) :: [Integer] -> [Integer]
  filter (>1000) ls = ...
  ```
- Ist Zahl mit genau drei Einsen in der Dezimaldarstellung:
  ```
  hat_drei_Einsen :: Integer -> Bool
  hat_drei_Einsen n = ...
  ```
- Ist Zahl mit Palindromdezimaldarstellung:
  ```
  ist_Palindrom :: Integer -> Bool
  ist_Palindrom n = ...
  ```
- ...

Vortr. VII
Teil VI
Kap. 18
18.1
18.2
18.3
18.4
18.5
Teil VII
Kap. 19
Anhänge
A
Umgekehr Klassen-zim-mer VI
Hinweis
Viel Erfolg!
  ```

G/F-Modul. am Bsp. des Primzahlstroms (2)

Zusammenfügen der G/F-Module zum Gesamtprogramm:

- Anwendung der G/F-Modularisierung:

  Alle Primzahlen größer als 1000:

  ```
  filter (>1000) gen_Primes
    ->> [1009,1013,1019,1021,1031,1033,1039,...
  ```

- Anwendung der G/F-Modularisierung:

  Alle Primzahlen mit genau drei Einsen in der Dezimaldarstellung:

  ```
  [ n | n <- gen_Primes, hat_drei_Einsen n]
    ->> [1117,1151,1171,1181,1511,1811,2111,...
  ```

- Anwendung der G/F-Modularisierung:

  Alle Primzahlen mit Palindromdezimaldarstellung:

  ```
  [ n | n <- gen_Primes, ist_Palindrom n]
    ->> [2,3,5,7,11,101,131,151,181,191,313,...
  ```

G/T-Modul. am Bsp. des Primzahlstroms (1)

Ein Generator (G):

- ```
  gen_Primes :: [Integer]
  gen_Primes = primes
  ```

Viele Transformatoren (T):

- Quadrieren (für den Strom der Quadratprimzahlen):
  ```
  square :: Integer -> Integer
  square n = ...
  ```
- Dekrementieren (für den Strom der Primzahlvorgänger):
  ```
  decrement :: Integer -> Integer
  decrement n = n-1
  ```
- Summieren (für den Strom der partiellen Primzahlsummen (den Strom d. Summen d. Primzahlen von 2 bis n)):
  ```
  sum :: [Integer] -> Integer
  sum ls = ...
  ```
- ...

Vortr. VII
Teil VI
Kap. 18
18.1
18.2
18.3
18.4
18.5
Teil VII
Kap. 19
Anhänge
A
Umgekehr Klassenzimmer VI
Hinweis
Viel Erfolg!
  ```

G/T-Modul. am Bsp. des Primzahlstroms (2)

Zusammenfügen der G/T-Module zum Gesamtprogramm:

- Anwendung der G/T-Modularisierung:

  Der Strom der Quadratprimzahlen:

  ```
  [ square n | n <- gen_Primes ]
     ->> [4,9,25,49,121,169,289,361,529,841,...
  ```

- Anwendung der G/T-Modularisierung:

  Der Strom der Primzahlvorgänger:

  ```
  [ decrement n | n <- gen_Primes ]
     ->> [1,2,4,6,10,12,16,18,22,28,...
  ```

- Anwendung der G/T-Modularisierung:

  Der Strom der partiellen Primzahlsummen:

  ```
  [ sum [2..n] | n <- gen_Primes ]
     ->> [2,5,14,27,65,90,152,189,275,434,...
  ```

Vortr. VII
Teil VI
Kap. 18
18.1
18.2
18.3
18.4
18.5
Teil VII
Kap. 19
Anhänge
A
Umgekehr Klassen-zim-mer VI
Hinweis
Viel Erfolg!

Typische Anwendungen

...für G/S-, G/F-, G/T-Modularisierungen:

- Rucksackprobleme
- Potenzreihen
- Fibonacci-Zahlen
- Pascalsches Dreieck
- Goldenes Verhältnis
- ...

Vortr. VII
Teil VI
Kap. 18
18.1
18.2
18.3
18.4
18.5
Teil VII
Kap. 19
Anhänge
A
Umgekehrtes Klassenzimmer VI
Hinweis
Viel Erfolg!

Anmerkung zur Terminierung

Auf Terminierung ist bei Generiere/anpass-Modularisierungen

- stets besonders zu achten.

So terminiert der Aufruf:

```
filter (<10) gen_Primes ->> [2,3,5,7,
```

nicht; der Aufruf:

```
takeWhile (<10) gen_Primes ->> [2,3,5,7]
```

hingegen schon.

Vortr. VII
Teil VI
Kap. 18
18.1
18.2
18.3
18.4
18.5
Teil VII
Kap. 19
Anhänge
A
Umgekehr Klassen-zim-mer VI
Hinweis
Viel Erfolg!

Rechnen mit Strömen: Naiv vs. intelligent

...am Beispiel des Stroms der Fibonacci-Zahlen:

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

Naiv – einfach, aber ineffizient:

Generator

```
fibs :: [Integer]  -- Generator der Fibonacci-Zahlen
fibs = map fib [0..]
                ⏟
        Strom von Argumenten
```

```
fibs ->> [0,1,1,2,3,5,8,13,21,34,55,...
```

...hat exponentielles Laufzeitverhalten.

Vortr. VII
Teil VI
Kap. 18
18.1
18.2
18.3
18.4
18.5
Teil VII
Kap. 19
Anhänge
A
Umgekehr
Klassen-zimmer VI
Hinweis
Viel Erfolg!

Rechnen mit Strömen: Intelligent, effizient

Intelligent – gleichfalls einfach, aber effizient:

G1, Strom der Fib.-Zahlen:	0 1 1 2 3 5 8 13 21...
G2, Rest d. Stroms d. Fib.-Z.:	1 1 2 3 5 8 13 21 34...
Summiere G1 u. G2, 'G1+G2':	+ + + + + + + + +...
Rest des Restes des Stroms der Fibonacci-Zahlen	1 2 3 5 8 13 21 34 55...

Effiziente Berechnung der Fibonacci-Z. als Summe von G1 u. G2:

```
fibs :: [Integer] -- Generator der Fibonacci-Zahlen
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

- `0 : 1` → 'Schopf'
- `zipWith (+) fibs (tail fibs)` → 'Sumpf'; `fibs` → G1; `(tail fibs)` → G2
- `zipWith (+) fibs (tail fibs)` → Rest d. Restes d. Stroms d. Fib.-Z.
- `0 : 1 : zipWith (+) fibs (tail fibs)` → Strom der Fibonacci-Zahlen

...sich wie Münchhausen 'am eigenen Schopf aus dem Sumpf ziehen'!

Vortr. VII
Teil VI
Kap. 18
18.1
18.2
18.3
18.4
18.5
Teil VII
Kap. 19
Anhänge
A
Umgekehrt Klassen-zimmer VI
Hinweis
Viel Erfolg!

Generatoranwendungen und Hilfsfunktionen

Aufruf von Generator fibs:

```
fibs ->> [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55,...
```

Aufrufe von Generator/Filter-, Selektorkombinationen mit fibs:

```
filter even fibs ->> [0,2,8,34,144,...
take 10 fibs     ->> [0,1,1,2,3,5,8,13,21,34]
fibs!!5          ->> 3
```

Verwendete Hilfsfunktionen:

```
take :: Int -> [a] -> [a]
take 0 _             = []
take _ []            = []
take n (x:xs) | n>0 = x : take (n-1) xs
take _ _ = error "PreludeList.take: negative argument"

zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith f _      _      = []
```

Vortr. VII
Teil VI
Kap. 18
18.1
18.2
18.3
18.4
18.5
Teil VII
Kap. 19
Anhänge
A
Umgekehrtes Klassenzimmer VI
Hinweis
Viel Erfolg!

Zusammenfassung

Späte Auswertung (engl. lazy evaluation) erlaubt

- die Kontrolle der Auswertungsreihenfolge von Daten

zu trennen und ermöglicht dadurch die elegante Behandlung

- unendlicher Datenwerte (genauer: nicht a priori in der Größe beschränkter Datenwerte), insbesondere
  - unendlicher Listen, sog. Ströme (engl. streams, lazy lists)

Das führt zu problemorientierten, von der Programmlogik her begründeten neuen Modularisierungsmöglichkeiten, von Generiere/anpass-Modularisierungen:

- Generiere/selektiere-Modularisierung
- Generiere/filtere-Modularisierung
- Generiere/transformiere-Modularisierung
- ...

Vortr. VII
Teil VI
Kap. 18
18.1
18.2
18.3
18.4
18.5
Teil VII
Kap. 19
Anhänge
A
Umgekehr Klassen-zim-mer VI
Hinweis
Viel Erfolg!

Kapitel 18.5

Leseempfehlungen

Vortr. VII
Teil VI
Kap. 18
18.1
18.2
18.3
18.4
18.5
Teil VII
Kap. 19
Anhänge
A
Umgekehr Klassen-zim-mer VI
Hinweis
Viel Erfolg!

Basisleseempfehlungen für Kapitel 18

- Richard Bird. *Thinking Functionally with Haskell*. Cambridge University Press, 2015. (Kapitel 9, Infinite lists)
- Antonie J. T. Davie. *An Introduction to Functional Programming Systems using Haskell*. Cambridge University Press, 1992. (Kapitel 7.2, Infinite Objects; Kapitel 7.3, Streams)
- Paul Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, 2000. (Kapitel 14, Programming with Streams)
- Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 3. Auflage, 2011. (Kapitel 12, Developing higher-order programs; Kapitel 17, Lazy programming)

Vortr. VII
Teil VI
Kap. 18
18.1
18.2
18.3
18.4
18.5
Teil VII
Kap. 19
Anhänge
A
Umgekehrtes Klassenzimmer VI
Hinweis
Viel Erfolg!

Weiterführende Leseempfehlungen für Kap. 18

- Richard Bird, Phil Wadler. *An Introduction to Functional Programming*. Prentice Hall, 1988. (Kapitel 6.4, Divide and conquer; Kapitel 7, Infinite Lists)
- Lambert Meertens. *Functional Pearl: Calculating the Sieve of Eratosthenes*. Journal of Functional Programming 14(6):759-763, 2004.
- Matti Nykänen. *A Note on the Genuine Sieve of Eratosthenes*. Journal of Functional Programming 21(6):563-572, 2011.
- Melissa E. O'Neill. *The Genuine Sieve of Eratosthenes*. Journal of Functional Programming 19(1):95-106, 2009.
- Fethi Rabhi, Guy Lapalme. *Algorithms – A Functional Programming Approach*. Addison-Wesley, 1999. (Kapitel 8.1, Divide-and-conquer)

Teil VII

Abschluss

Vortr. VII
Teil VI
Kap. 18
Teil VII
Kap. 19
Anhänge
A
Umgekehr
Klassen-
zim-
mer VI
Hinweis
Viel
Erfolg!

Kapitel 19

Rückschau, Ausschau

Vortr. VII
Teil VI
Kap. 18
Teil VII
Kap. 19
19.1
19.2
19.3
Anhänge
A
Umgekehr Klassen-zim-mer VI
Hinweis
Viel Erfolg!

Suavis est laborum praeteritorum memoria.
Süss ist die Erinnerung an vergangene Mühen.

Cicero (106 - 43 v.Chr.)
röm. Staatsmann und Schriftsteller

Kapitel 19.1

Rückschau, Rückblick

Vortr. VII
Teil VI
Kap. 18
Teil VII
Kap. 19
19.1
19.2
19.3
Anhänge
A
Umgekehr
Klassen-
zim-
mer VI
Hinweis
Viel
Erfolg!

Funktionale, imperative Programmierung (1)

Eigenschaften und Charakteristika im Vergleich.

- Funktional
  - Programm ist Ein-/Ausgaberelation.
  - Programme sind zustandsfrei und 'zeitlos'.
  - Programmformulierung auf abstraktem, mathematisch geprägten Niveau, ohne eine Maschine im Blick.
- Imperativ
  - Programm ist Arbeitsanweisung für eine Maschine.
  - Programme sind zustands- und 'zeitbehaftet'.
  - Programmformulierung mit Blick auf eine Maschine, ein Maschinenmodell (von Neumann).

Vortr. VII
Teil VI
Kap. 18
Teil VII
Kap. 19
19.1
19.2
19.3
Anhänge
A
Umgekehr Klassen-zim-mer VI
Hinweis
Viel Erfolg!

Funktionale, imperative Programmierung (2)

- Funktional
  - Die Auswertungsreihenfolge von Ausdrücken liegt nicht fest (bis auf Datenabhängigkeiten).
  - Namen werden durch Wertvereinbarungen genau einmal für immer an einen Wert gebunden.
  - Schachtelung (rekursiver) Funktionsaufrufe erlaubt neue Werte mit neuen Namen zu verbinden.
- Imperativ
  - Die Ausführungsreihenfolge von Anweisungen liegt fest; Freiheiten bestehen bei der Auswertungsreihenfolge von Ausdrücken (wie funktional).
  - Namen werden in der zeitlichen Abfolge durch Zuweisungen temporär mit Werten belegt.
  - Namen können durch wiederholte Zuweisungen beliebig oft mit neuen Werten belegt werden (in rekursiven Aufrufen, repetitiven Anweisungen wie *while*, *repeat*, *for*).

Vortr. VII
Teil VI
Kap. 18
Teil VII
Kap. 19
19.1
19.2
19.3
Anhänge
A
Umgekehr Klassen-zim-mer VI
Hinweis
Viel Erfolg!

Wenige Prinzipien, viel Kraft

Die Fülle an Möglichkeiten
[in funktionalen Programmiersprachen] erwächst
aus einer kleinen Zahl von elementaren
Konstruktionsprinzipien.

Peter Pepper, *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*. Springer-Verlag, 2. Auflage, 2003.

Im Falle von:

- Funktionen: (Fkt.-) Applikation, Fallunterscheidung, Rekursion, Polymorphie.
- Datenstrukturen: Aufzählung, Produkt-, Summenbildung, Rekursion, Polymorphie.

Das Ganze ist mehr als die Summe seiner Teile.

Aristoteles (384 - 322 v.Chr.)
griech. Philosoph

Vortr. VII
Teil VI
Kap. 18
Teil VII
Kap. 19
19.1
19.2
19.3
Anhänge
A
Umgekehr
Klassen-
zim-
mer VI
Hinweis
Viel
Erfolg!

Die Mächtigkeit und Eleganz

...funktionaler Programmierung erwächst aus diesen wenigen Prinzipien zusammen mit der durchgehenden Umsetzung der Konzepte von:

- Funktionen als erstrangige Sprachelemente (engl. first class citizens)
  - Funktionen höherer Ordnung
- Polymorphie als echte und unechte Polymorphie auf
  - Funktionen
  - Datentypen

...und ihrem nahtlosen Zusammenspiel, auf den Punkt gebracht im Slogan:

Functional Programming is Fun!

Vortr. VII
Teil VI
Kap. 18
Teil VII
Kap. 19
19.1
19.2
19.3
Anhänge
A
Umgekehr Klassenzimmer VI
Hinweis
Viel Erfolg!

Im Rückblick auf die Vorbesprechung

...betrachte dazu noch einmal die Versprechungen über die Versprechen funktionaler Programmierung:

- Konrad Hinsen. The Promises of Functional Programming. Computing in Science and Engineering 11(4): 86-90, 2009.

  ...adopting a functional programming style could make your programs more robust, more compact, and more easily parallelizable.

- Konstantin Läufer, George K. Thiruvathukal. The Promises of Typed, Pure, and Lazy Functional Programming: Part II. Computing in Science and Engineering 11(5): 68-75, 2009.

  ...this second installment picks up where Konrad Hinsen's article "The Promises of Functional Programming" [...] left off, covering static type inference and lazy evaluation in functional programming languages.

Vortr. VII
Teil VI
Kap. 18
Teil VII
Kap. 19
19.1
19.2
19.3
Anhänge
A
Umgekehrtes Klassenzimmer VI
Hinweis
Viel Erfolg!

Erfolgreiche Einsatzfelder fkt. Programmierung

- Theorembeweiser HOL und Isabelle in ML.
- Modellprüfer (z.B. Edinburgh Concurrency Workbench).
- Mobility Server von Ericson in Erlang.
- Konsistenzprüfung mit Pdiff (Lucent 5ESS) in ML.
- Übersetzer in übersetzter Sprache geschrieben.
- Datenbankabfragesprachen (z.B. CPL/Kleisli, in ML; Natural Expert, Haskell-ähnliche Abfragesprache).
- Protokollspezifikation (effiziente Fallstudien z.B. in ML).
- Expertensysteme (oft Lisp-basiert).
- ...
- `http://homepages.inf.ed.ac.uk/wadler/realworld`
- `www.haskell.org/haskellwiki/Haskell_in_industry`

...s.a. Bonusthema zu Umgek. Klassenz. III zu Vortragsteil IV.

Vortr. VII
Teil VI
Kap. 18
Teil VII
Kap. 19
19.1
19.2
19.3
Anhänge
A
Umgekehr Klassen-zimmer VI
Hinweis
Viel Erfolg!

Ich denke nie an die Zukunft.
Sie kommt früh genug.

Albert Einstein (1879-1955)
dt.-schweiz.-amerik. Physiker

Wer nicht an die Zukunft denkt,
wird bald Sorgen haben.

Konfuzius (551 - 479 v.Chr.)
chin. Ethiker und Staatslehrer

Kapitel 19.2

Ausschau, Ausblick

Die Zukunft hat schon begonnen.

Robert Jungk (1913-1994)
österr. Wissenschaftspublizist und Zukunftsforscher

Vortr. VII
Teil VI
Kap. 18
Teil VII
Kap. 19
19.1
19.2
19.3
Anhänge
A
Umgekehr Klassen-zimmer VI
Hinweis
Viel Erfolg!

Alles, was man wissen muss,
um selber weiter zu lernen,
...ist jetzt gelernt.

Dietrich Schwanitz (1940-2004)
dt. Anglistikprof. und Schriftsteller
(verkürzt und ergänzt in seinem Sinn)

Fort- und weiterführendes zu funktionaler Programmierung in TUW-Lehrveranstaltungen, insbesondere:

- LVA 185.A05 Fortgeschrittene funktionale Programmierung. VU 2.0, ECTS 3.0.
- LVA 183.653 Methodisches, industrielles Software-Engineering mit funktionalen Sprachen am Fallbeispiel von Haskell. VU 2.0, ECTS 3.0, ao.Prof. Thomas Grechenig.
- LVA 127.008 Haskell-Praxis: Programmieren mit der funktionalen Programmiersprache Haskell.
  VU 2.0, ECTS 3.0, Prof. em. Andreas Frank, Institut für Geoinformation und Kartographie.

LVA 185.A05 Fortg. fkt. Programmierung

Vorlesungsinhalte:

- Programmieren mit
  - Strömen, Funktoren, Monaden, Kombinatorbibliotheken.
  - Funktionalen Feldern, abstrakten Datentypen.
- Anwendungen
  - Funktionale Perlen, Algorithmenmuster, funktionale reaktive Programmierung, logische Programmierung funktional, Strukturanalyse (engl. Parsing).
- Qualitätssicherung
  - Programmverifikation, Programmvalidation, gleichungsbasiertes Schließen und Beweisen, automatisches Testen.
- ...

Vortr. VII
Teil VI
Kap. 18
Teil VII
Kap. 19
19.1
19.2
19.3
Anhänge
A
Umgekehrt Klassenzimmer VI
Hinweis
Viel Erfolg!

LVA 127.008 Haskell-Praxis: Progr. mit Haskell

Vorlesungsinhalte:

- Analyse und Verbesserung von gegebenem Code.
- Weiterentwicklung der Open-Source-Entwicklungsumgebung LEKSAH für Haskell, insbesondere der graphischen Benutzerschnittstelle (GUI).
- Gestaltung graphischer Benutzerschnittstellen (GUIs) mit Glade und Gtk+.
- ...

Vortr. VII
Teil VI
Kap. 18
Teil VII
Kap. 19
19.1
19.2
19.3
Anhänge
A
Umgekehr Klassen-zim-mer VI
Hinweis
Viel Erfolg!

Always look on the bright side of life

The clarity and economy of expression that the language of functional programming permits is often very impressive, and, but for human inertia, functional programming can be expected to have a brilliant future.(∗)

Edsger W. Dijkstra (1930-2002)
Turing Award Preisträger 1972

(∗) Zitat aus: Introducing a course on calculi. Ankündigung einer Lehrveranstaltung an der University of Texas at Austin, 1995.

Der größte Feind des Fortschritts ist nicht der Irrtum, sondern die Trägheit.

Henry Thomas Buckle (1821-1862)
engl. Historiker, aus "Geschichte der Zivilisation" (unvollendet)

Vortr. VII
Teil VI
Kap. 18
Teil VII
Kap. 19
19.1
19.2
19.3
Anhänge
A
Umgekehr Klassenzimmer VI
Hinweis
Viel Erfolg!

Kapitel 19.3

Leseempfehlungen

Vortr. VII
Teil VI
Kap. 18
Teil VII
Kap. 19
19.1
19.2
19.3
Anhänge
A
Umgekehr
Klassen-
zim-
mer VI
Hinweis
Viel
Erfolg!

Basisleseempfehlungen für Kapitel 19

- John Hughes. *Why Functional Programming Matters*. The Computer Journal 32(2):98-107, 1989.
- Philip Wadler. *The Essence of Functional Programming*. In Conference Record of the 19th Annual ACM Symposium on Principles of Programming Languages (POPL'92), 1-14, 1992.
- Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 3. Auflage, 2011. (Anhang A, Functional, imperative and OO programming)
- Greg Michaelson. *An Introduction to Functional Programming through Lambda Calculus*. Dover Publications, 2. Auflage, 2011. (Kapitel 1, Introduction; Kapitel 9, Functional programming in Standard ML; Kapitel 10, Functional programming and LISP)

Vortr. VII
Teil VI
Kap. 18
Teil VII
Kap. 19
19.1
19.2
19.3
Anhänge
A
Umgekehr
Klassen-
zim-
mer VI
Hinweis
Viel
Erfolg!

Weiterführ. Leseempfehlungen für Kap. 19 (1)

- Simon Peyton Jones. *16 Years of Haskell: A Retrospective on the occasion of its 15th Anniversary – Wearing the Hair Shirt: A Retrospective on Haskell*. Invited Keynote Presentation at the 30th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'03), 2003. `research.microsoft.com/users/simonpj/papers/haskell-retrospective/`
- Paul Hudak, John Hughes, Simon Peyton Jones, Philip Wadler. *A History of Haskell: Being Lazy with Class*. In Proceedings of the 3rd ACM SIGPLAN 2007 Conference on History of Programming Languages (HOPL III), 12-1 - 12-55, 2007. (ACM Digital Library `www.acm.org/dl`)

Vortr. VII
Teil VI
Kap. 18
Teil VII
Kap. 19
19.1
19.2
19.3
Anhänge
A
Umgekehr
Klassen-
zim-
mer VI
Hinweis
Viel
Erfolg!

Weiterführ. Leseempfehlungen für Kap. 19 (2)

- Anthony J. Field, Peter G. Robinson. *Functional Programming*. Addison-Wesley, 1988. (Kapitel 5, Alternative functional styles)
- Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2. Auflage, 2016. (Kapitel 1.3, Features of Haskell; Kapitel 1.4, Historical background)
- Fethi Rabhi, Guy Lapalme. *Algorithms – A Functional Programming Approach*. Addison-Wesley, 1999. (Kapitel 1.2, Functional Languages)
- Colin Runciman, David Wakeling. *Applications of Functional Programming*. UCL Press, 1995.

Vortr. VII
Teil VI
Kap. 18
Teil VII
Kap. 19
19.1
19.2
19.3
Anhänge
A
Umgekehr Klassenzimmer VI
Hinweis
Viel Erfolg!

Anhänge

Vortr. VII
Teil VI
Kap. 18
Teil VII
Kap. 19
Anhänge
A
Umgekehr
Klassen-
zim-
mer VI
Hinweis
Viel
Erfolg!

A

Schlaglichter: Imperative vs. funktionale Programmierung

Vortr. VII
Teil VI
Kap. 18
Teil VII
Kap. 19
Anhänge
A
A.8
A.9
Umgekehrtes Klassenzimmer VI
Hinweis
Viel Erfolg!

A.8

Problem- und Lösungssicht: Imperativ vs. funktional

Vortr. VII
Teil VI
Kap. 18
Teil VII
Kap. 19
Anhänge
A
A.8
A.9
Umgekehr
Klassen-
zim-
mer VI
Hinweis
Viel
Erfolg!

Die Problemsicht

...ident für imperative und funktionale Programmierung, gegeben und beschrieben durch:

1. Problemmenge $\mathcal{P}$, Lösungsmenge $\mathcal{L}$.
2. Probleminstanzen $p \in \mathcal{P}$ mit Lösungen $l_p \in \mathcal{L}$.
3. Gesucht: Ein Weg, von p zu l_p zu gelangen.

Beispiel:

1. $\mathcal{P}$: Die Menge der Listen ganzer Zahlen.

   $\mathcal{L}$: Die Menge der aufsteigend sortierten Listen ganzer Zahlen.
2. p: Die Zahlenliste $42, 4, 4711, 17$.

   l_p: Die aufsteigend sortierte Zahlenliste $4, 17, 42, 4711$.
3. Gesucht: Ein Sortierverfahren für Listen ganzer Zahlen (z.B. Quicksort).

Die imperative Lösungssicht

Ansatz:

- Schreibe zur Lösung von Probleminstanzen $p \in \mathcal{P}$ ein imperatives Programm π über einer Menge von (Programm-) Variablen.
  - Die initialen Variablenwerte beschreiben p.
  - Die Ausführung von π modifiziert diese Werte.
  - Die finalen Werte beschreiben l_p, eine Lösung von p.

Korrektheitsannahme (partielle Korrektheit):

- Terminiert π angesetzt auf initiale Variablenwerte, die die Probleminstanz p beschreiben, so beschreiben die finalen Variablenwerte eine Lösung l_p von p.

Frage, Aufforderung an ein imperatives Prg.

Ist π ein imperatives Programm und p eine Probleminstanz, die durch eine Menge initialer Werte der (Programm-) Variablen beschrieben wird, so lauten Frage und Aufforderung an π:

Frage an π: Welches sind

- die finalen Variablenwerte und die davon beschriebene Lösung l_p, wenn du mit den initialen Variablenwerten gestartet wirst?

Aufforderung an π:

- Führe deine Instruktionen beginnend mit den initialen Variablenwerten aus und liefere die Werte dieser Variablen und die davon beschriebene Lösung nach Terminierung.

Vortr. VII
Teil VI
Kap. 18
Teil VII
Kap. 19
Anhänge
A
A.8
A.9
Umgekehr
Klassen-zimmer VI
Hinweis
Viel Erfolg!

Die funktionale Lösungssicht

Ansatz:

- Schreibe zur Lösung von Probleminstanzen $p \in \mathcal{P}$ ein funktionales Programm ϕ über einer Menge von (Programm-) Namen.
  - Ein initialer Ausdruck α über den Namen beschreibt p.
  - Die Ausführung von ϕ berechnet den Wert von α.
  - Dieser Wert beschreibt l_p, eine Lösung von p.

Korrektheitsannahme (partielle Korrektheit):

- Terminiert die Auswertung des initialen Ausdrucks α durch ϕ, der die Probleminstanz p beschreibt, so beschreibt der Wert von α eine Lösung l_p von p.

Vortr. VII
Teil VI
Kap. 18
Teil VII
Kap. 19
Anhänge
A
A.8
A.9
Umgekehr
Klassen-zimmer VI
Hinweis
Viel Erfolg!

Frage, Aufforderung an ein funktionales Prg.

Ist ϕ ein funktionales Programm (also ein System von Gleichungen), $p \in \mathcal{P}$ eine Probleminstanz und α ein p beschreibender Ausdruck über der Namensmenge von ϕ, so lauten Frage und Aufforderung an ϕ:

Frage an ϕ: Welches ist

- der Wert von α und die davon beschriebene Lösung l_p, wenn du die in dir festgelegten Gleichheiten zugrundelegst?

Aufforderung an ϕ:

- Liefere den Wert des initialen Ausdrucks α unter Zugrundelegung und Lösung der in dir festgelegten Gleichungen und die davon beschriebene Lösung.

Vortr. VII
Teil VI
Kap. 18
Teil VII
Kap. 19
Anhänge
A
A.8
A.9
Umgekehrt Klassenzimmer VI
Hinweis
Viel Erfolg!

Direkte Gegenüberstellung zum Vergleich

Aufforderung an ein imperatives Programm:

- Führe deine Instruktionen beginnend mit initialen Werten deiner Variablen aus und liefere die finalen Werte dieser Variablen!

Aufforderung an ein funktionales Programm:

- Liefere den Wert eines initialen Ausdrucks unter Zugrundelegung und Lösung der in dir festgelegten Gleichungen!

Der Unterschied ist offensichtlich u. konzeptuell fundamental:

- Imperativ: Denken in Instruktionen und ihren Effekten.
- Funktional: Denken in Gleichungen und Eigenschaften ihrer Lösungen.

Vortr. VII
Teil VI
Kap. 18
Teil VII
Kap. 19
Anhänge
A
A.8
A.9
Umgekehr
Klassen-
zim-
mer VI
Hinweis
Viel
Erfolg!

Diese konzeptuell unterschiedl. Problemsicht

...hat Auswirkungen auf das Denken, Tun und Produkt

- imperativer
- funktionaler

Programmierung.

Vortr. VII
Teil VI
Kap. 18
Teil VII
Kap. 19
Anhänge
A
A.8
A.9
Umgekehr Klassen-zim-mer VI
Hinweis
Viel Erfolg!

Denken, Tun, Produkt imp. Programmierung

Denken:

- Denken in Instruktionen und ihren Effekten.

Tun:

Spezifiziere Instruktionen an den Rechner:

- Tu dies, tu das, tu jenes,...

mit dem Ziel, ihn in die Lage zu versetzen, durch Ausführung dieser Instruktionen problembeschreibende initiale Variablenwerte in

- lösungsbeschreibende finale Werte

zu überführen.

Produkt:

Ein imperatives Programm als Gesamtheit seiner Instruktionen.

Vortr. VII
Teil VI
Kap. 18
Teil VII
Kap. 19
Anhänge
A
A.8
A.9
Umgekehr
Klassen-zim-mer VI
Hinweis
Viel Erfolg!

Denken, Tun, Produkt fkt. Programmierung

Denken:

- Denken in Gleichung(ssystem)en und Eigenschaften ihrer Lösungen.

Tun:

Spezifiziere Gleichungen für den Rechner:

- Diese, jene, folgende,...

mit dem Ziel, ihn in die Lage zu versetzen, durch Lösung dieser Gleichungen einen problembeschreibenden Ausdruck in seinen

- lösungsbeschreibenden Wert

zu überführen.

Produkt:

Ein funktionales Programm als Gesamtheit seiner Gleichungen.

Vortr. VII
Teil VI
Kap. 18
Teil VII
Kap. 19
Anhänge
A
A.8
A.9
Umgekehr
Klassen-
zim-
mer VI
Hinweis
Viel
Erfolg!

Übungsaufgabe A.8.1 – Logisierung

Wie sehen die Welt und ihre Probleme durch die Brille logischer Programmierung aus? Wie sehen im Sinn dieses Abschnitts die

- logische Lösungssicht (Ansatz, Korrektheitsannahme)
- Frage, Aufforderung an ein logisches Programm

aus?

Wodurch sind Herangehens- und Denkweise gekennzeichnet, ein Problem durch logische Programmierung, z.B. ein Prolog-Programm, zu lösen?

- Logisch Programmieren heißt: Denken in ...?

Was folgt daraus für Tun und Produkt im Fall logischer Programmierung?

Vortr. VII
Teil VI
Kap. 18
Teil VII
Kap. 19
Anhänge
A
A.8
A.9
Umgekehr Klassenzimmer VI
Hinweis
Viel Erfolg!

A.9

Welcher Problemlösungstyp bin ich?

Vortr. VII
Teil VI
Kap. 18
Teil VII
Kap. 19
Anhänge
A
A.8
A.9
Umgekehr
Klassen-
zim-
mer VI
Hinweis
Viel
Erfolg!

Der imperative Problemlösungstyp

Konzeptionell:

- Denkt in Instruktionen und ihren Effekten.

Operationell:

- Ordnet haarklein jeden Schritt bis hin zum allerletzten Detail unzweideutig an.

Ich erkär's nur einmal – Sortieren geht so:

```
quickSort (L,low,high)
  if low < high
    then splitInd = partition (L,low,high)
         quickSort (L,low,splitInd-1)
         quickSort (L,splitInd+1,high) fi

partition (L,low,high)
  l = L[low]
  left = low
  for i = low+1 to high do
    if L[i] <= l then left = left+1
                      swap (L[i],L[left]) fi od
  swap (L[low],L[left])
  return left
```

Abtreten zum Sortieren! Im Laufschritt. Marsch!

In der Gefahr besteht die Schwierigkeit nie darin, Menschen zu finden, die gehorchen werden, sondern Männer, die befehlen können.

George Bernard Shaw (1854-1900)
irischer Schriftsteller

Der funktionale Problemlösungstyp

Konzeptionell:

- Denkt in Gleichung(ssystem)en und Eigenschaften ihrer Lösungen.

Operationell:

- Beschreibt präzise die Eigenschaften der Lösung.

Lieber Sesselkreis, liebe Sesselkreisler,
ich wünsche mir, dass meine Liste permutiert eine
der folgenden zwei Eigenschaften erfüllt:

```
(1) quickSort []     = []
(2) quickSort (n:ns) = quickSort [m | m <- ns, m <= n]
                        ++ [n]
                        ++ quickSort [m | m <- ns, m > n]
```

...den lästigen Rest zur Wunschverwirklichung überneh-
men ihre Sesselkreisler, begeistert, zwanglos, sofort.

Der ideale Mensch fühlt Freude,
wenn er anderen einen Dienst erweisen kann.

Aristoteles (384 - 322 v.Chr.)
griech. Philosoph

Man soll den Menschen nie sagen,
wie sie etwas tun sollen,
sondern nur, was sie tun sollen.
Dann wird ihr Einfallsreichtum einen verblüffen.

George S. Patton (1885-1945)
amerik. General

Vortr. VII
Teil VI
Kap. 18
Teil VII
Kap. 19
Anhänge
A
A.8
A.9
Umgekehr
Klassen-
zim-
mer VI
Hinweis
Viel
Erfolg!

Imperativer und fkt. Lösungstyp im Vergleich

Imperativ:

Der inspirierende Instrukteur:

> Ordnet exakt den einzuschlagenden Lösungsweg an, das 'wie' zur Lösung gelangen.

Funktional:

Der elegante Sesselkreis-Delegateur:

> Beschreibt präzise die essentiellen Eigenschaften der Lösung, das 'was' der Lösung, unter weitreichender Freistellung des konkreten Wegs zur Lösung für den Sesselkreis bei freilich determiniertem Ergebnis!

Vortr. VII
Teil VI
Kap. 18
Teil VII
Kap. 19
Anhänge
A
A.8
A.9
Umgekehrt Klassenzimmer VI
Hinweis
Viel Erfolg!

Instrukteur, Delegateur: Erforderliches Wissen

Imperativ:

- Wie sieht die Lösung aus? Wie komme ich zur Lösung hin?

  Bsp.: Der Instrukteur muss wissen: Wenn auf meine Liste folgende Schritte pippifein angewendet werden, ist sie sortiert.

```
quickSort (L,low,high)
  if low < high
    then splitInd = partition (L,low,high)
         quickSort (L,low,splitInd-1)
         quickSort (L,splitInd+1,high) fi
```

```
partition (L,low,high)
  l = L[low]
  left = low
  for i = low+1 to high do
    if L[i] <= l then left = left+1
                      swap (L[i],L[left]) fi od
  swap (L[low],L[left])
  return left
```

Funktional:

- Was erwarte ich von der Lösung?

  Bsp.: Der Delegateur erwartet: Meine Liste ist sortiert, wenn sie Gleichung (1) oder Gleichung (2) erfüllt (und denkt sich: Zu wissen, was ich will, war durchaus genug mit Arbeit verbunden; Wegschrittfolgen auch noch wissen zu sollen, wäre zu viel erwartet!).

```
(1) quickSort []     = []
(2) quickSort (n:ns) = quickSort [m | m <- ns, m <= n]
                       ++ [n] ++ quickSort [m | m <- ns, m > n]
```


Vortr. VII
Teil VI
Kap. 18
Teil VII
Kap. 19
Anhänge
A
A.8
A.9
Umgekehr
Klassen-zimmer VI
Hinweis
Viel Erfolg!

Imp. vs. fkt. Lsg.-Weg: Aufgabenlastverteilung

Auf wem liegt konzeptionell und operationell die größere Last?

- Dem handlungsfixierten imperativ vorgehenden Instrukteur?

- Dem ergebnisorientierten funktional vorgehenden Delegateur?

Vortr. VII
Teil VI
Kap. 18
Teil VII
Kap. 19
Anhänge
A
A.8
A.9
Umgekehr
Klassen-
zim-
mer VI
Hinweis
Viel
Erfolg!

Imperative oder funktionale Programmierung

...eine Frage (auch?!) von Typ, Persönlichkeit, Führungsstil:

- Sie bevorzugen Ordnung, klare Hierarchien und Ansagen und wissen ohnehin am besten, wie eine Aufgabe zu erledigen ist?

  Imperative Programmierung ist Ihr Ding. Probieren Sie nichts anderes.

- Sie bevorzugen ein kooperatives, harmonisches Arbeitsumfeld, in dem partizipativ, konsensual mit schließlicher Zufriedenstellung aller ohne der Rede werten eigenen Beitrags Ihr ganz persönliches Wunschergebnis erarbeitet wird?

  Funktionale Programmierung könnte Ihr Ding sein. Probieren Sie es!

  Gleichungen bilden Ihr kooperatives und harmonisches Arbeitsumfeld!

Vortr. VII
Teil VI
Kap. 18
Teil VII
Kap. 19
Anhänge
A
A.8
A.9
Umgekehr
Klassen-
zim-
mer VI
Hinweis
Viel
Erfolg!

Haben Sie eine Antwort für sich gefunden?

...ich bin (als Programmierer) ein Typ

- klarer Ansagen, denke imperativ in Instruktionen und ihren Effekten und handle als Instrukteur.

- harmonischen, konsensualen Ausgleichs, denke funktional in Gleichungssystemen und Eigenschaften ihrer Lösungen und handle als Delegateur.

- faszinierender Empathie- und Herzlosigkeit, denke logisch in Formeln und ihren logischen Konsequenzen und handle als Spockteur.

Vortr. VII
Teil VI
Kap. 18
Teil VII
Kap. 19
Anhänge
A
A.8
A.9
Umgekehr
Klassen-zim-mer VI
Hinweis
Viel Erfolg!

Umgekehrtes Klassenzimmer VI

...zur Übung, Vertiefung

...nach Eigenstudium von Teil V 'Weiterführende Konzepte':

– Zwar weiß ich viel...

Als Bonusthema, so weit die Zeit erlaubt:

– Die Versprechen funktionaler Programmierung: Erfüllt? Oder versprochen beim Versprechen?

Vortr. VII
Teil VI
Kap. 18
Teil VII
Kap. 19
Anhänge
A
Umgekehrtes Klassenzimmer VI
Zwar weiß ich viel...
Bonusthema: Versprechen fkt. Programmierung
Hinweis
Viel Erfolg!

Zwar weiß ich viel...

doch möcht ich alles wissen.

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

Vortr. VII
Teil VI
Kap. 18
Teil VII
Kap. 19
Anhänge
A
Umgekehr Klassenzimmer VI
Zwar weiß ich viel...
Bonusthema: Versprechen fkt. Programmierung
Hinweis
Viel Erfolg!

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. VII
Teil VI
Kap. 18
Teil VII
Kap. 19
Anhänge
A
Umgekehrtes Klassenzimmer VI
Zwar weiß ich viel...
Bonusthema: Versprechen fkt. Programmierung
Hinweis
Viel Erfolg!

Bonusthema

Die Versprechen funktionaler Programmierung

...im Sinne von Hinsen, Läufer, Thiruvathukal.

Vortr. VII
Teil VI
Kap. 18
Teil VII
Kap. 19
Anhänge
A
Umgekehrtes Klassenzimmer VI
Zwar weiß ich viel...
Bonusthema: Versprechen fkt. Programmierung
Hinweis
Viel Erfolg!

Welche Versprechen? Eingelöst? Nicht eingel.?

...Ihre Antwort, Ihre Einschätzung, bitte!

Auch ein Versprechen: Functional Programming is Fun!

...haben Sie solche Momente erlebt? Wann? Wobei?

...gab es Aha-Erlebnisse? Wann? Wobei?

...oder ist es beim Versprechen geblieben? Versprechen kann man sich ja einmal.

Zwei Exzerpte als Anregung

- ...adopting a functional programming style could make your programs more robust, more compact, and more easily parallelizable.

  Aus: Konrad Hinsen. The Promises of Functional Programming. Computing in Science and Engineering 11(4): 86-90, 2009.

- ...this second installment picks up where Konrad Hinsen's article "The Promises of Functional Programming" [...] left off, covering static type inference and lazy evaluation in functional programming languages.

  Aus: Konstantin Läufer, George K. Thiruvathukal. The Promises of Typed, Pure, and Lazy Functional Programming: Part II. Computing in Science and Engineering 11(5):68-75, 2009.

Vortr. VII
Teil VI
Kap. 18
Teil VII
Kap. 19
Anhänge
A
Umgekehrt Klassenzimmer VI
Zwar weiß ich viel...
Bonusthema: Versprechen fkt. Programmierung
Hinweis
Viel Erfolg!

Hinweis

...für das Verständnis von Vorlesungsteil VII ist eine über den unmittelbaren Inhalt von Vortrag VII 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. VII
Teil VI
Kap. 18
Teil VII
Kap. 19
Anhänge
A
Umgekehr Klassenzimmer VI
Hinweis
Viel Erfolg!

Zum Abschluss

Frohe Feiertage, einen guten Start ins neue Jahr und

- viel Erfolg

für alle dieses Semester für Sie noch anstehenden Prüfungen!

...einen weiteren Vortragstermin gibt es nicht!

Vortr. VII
Teil VI
Kap. 18
Teil VII
Kap. 19
Anhänge
A
Umgekehr Klassenzimmer VI
Hinweis
Viel Erfolg!