

Praktische Informatik 3: Funktionale Programmierung

Vorlesung 1 vom 18.10.2016: Einführung

Christoph Lüth

Universität Bremen

Wintersemester 2016/17

Personal

- ▶ **Vorlesung:**

Christoph Lüth <cxl@informatik.uni-bremen.de>

www.informatik.uni-bremen.de/~cxl/ (MZH 4186, Tel. 59830)

- ▶ **Tutoren:**

Tobias Brandt <to_br@uni-bremen.de>

Tristan Bruns <tbruns@informatik.uni-bremen.de>

Johannes Ganser <ganser@uni-bremen.de>

Alexander Kurth <kurth1@uni-bremen.de>

Berthold Hoffmann <hof@informatik.uni-bremen.de>

- ▶ **“Fragestunde”:** Berthold Hoffmann n.V. (Cartesium 1.54, Tel. 64 222)

- ▶ **Webseite:** www.informatik.uni-bremen.de/~cxl/lehre/pi3.ws16

Termine

- ▶ **Vorlesung:** Di 16 – 18 NW1 H 1 – H0020

- ▶ **Tutorien:**

| | | | |
|----|---------|-----------|-------------------|
| Mi | 08 – 10 | GW1 A0160 | Berthold Hoffmann |
| | 10 – 12 | GW1 A0160 | Johannes Ganser |
| | 12 – 14 | MZH 1110 | Johannes Ganser |
| | 14 – 16 | GW1 B2070 | Alexander Kurth |
| Do | 08 – 10 | MZH 1110 | Tobias Brandt |
| | 10 – 12 | GW1 B2130 | Tristan Bruns |

- ▶ **Anmeldung** zu den Übungsgruppen über stud.ip
 - ▶ Duale Studierende sollten im Tutorium Do 10– 12 registriert sein.

Übungsbetrieb

- ▶ Ausgabe der Übungsblätter über die Webseite **Dienstag morgen**
- ▶ Besprechung der Übungsblätter in den Tutorien
- ▶ **Bearbeitungszeit:** eine Woche
- ▶ **Abgabe:** elektronisch bis **Freitag** nächste Woche **12:00**
- ▶ **Zehn** Übungsblätter (voraussichtlich) plus 0. Übungsblatt
- ▶ Übungsgruppen: max. **drei Teilnehmer**
- ▶ **Bewertung:** Quellcode 50%, Tests 25%, Dokumentation 25%
 - ▶ Nicht übersetzender Quellcode: **0 Punkte**

Scheinkriterien

- ▶ Geplant: $n = 10$ Übungsblätter
- ▶ Mind. 50% in **allen** und in den **ersten $n/2$** Übungsblättern
- ▶ **Notenspiegel** (in Prozent aller Punkte):

| Pkt.% | Note | Pkt.% | Note | Pkt.% | Note | Pkt.% | Note |
|-----------|------|---------|------|---------|------|---------|------|
| | | 89.5-85 | 1.7 | 74.5-70 | 2.7 | 59.5-55 | 3.7 |
| ≥ 95 | 1.0 | 84.5-80 | 2.0 | 69.5-65 | 3.0 | 54.5-50 | 4.0 |
| 94.5-90 | 1.3 | 79.5-75 | 2.3 | 64.5-60 | 3.3 | 49.5-0 | n/b |

- ▶ **Fachgespräch** (Individualität der Leistung) am Ende
- ▶ Alternative: **Modulprüfung** (mündlich)

Spielregeln

- ▶ **Quellen angeben** bei
 - ▶ Gruppenübergreifender Zusammenarbeit;
 - ▶ Internetrecherche, Literatur, etc.
- ▶ **Täuschungsversuch:**
 - ▶ Null Punkte, **kein** Schein, **Meldung** an das **Prüfungsamt**
- ▶ **Deadline verpaßt?**
 - ▶ **Triftiger** Grund (z.B. Krankheit mehrerer Gruppenmitglieder)
 - ▶ **Vorher** ankündigen, sonst **null** Punkte.

Fahrplan

- ▶ Teil I: Funktionale Programmierung im Kleinen
 - ▶ Einführung
 - ▶ Funktionen und Datentypen
 - ▶ Algebraische Datentypen
 - ▶ Typvariablen und Polymorphie
 - ▶ Funktionen höherer Ordnung I
 - ▶ Funktionen höherer Ordnung II und Effizienzaspekte
- ▶ Teil II: Funktionale Programmierung im Großen
- ▶ Teil III: Funktionale Programmierung im richtigen Leben

Warum funktionale Programmierung lernen?

- ▶ Funktionale Programmierung macht aus Programmierern Informatiker
- ▶ Blick über den Tellerrand — was kommt in 10 Jahren?
- ▶ Herausforderungen der Zukunft
- ▶ Enthält die wesentlichen Elemente moderner Programmierung

Zukunft eingebaut

Funktionale Programmierung ist bereit für die Herausforderungen der Zukunft:

- ▶ Nebenläufige Systeme (Mehrkernarchitekturen)
- ▶ Massiv verteilte Systeme („Internet der Dinge“)
- ▶ Große Datenmengen („Big Data“)

The Future is Bright — The Future is Functional

- ▶ Funktionale Programmierung enthält die **wesentlichen** Elemente moderner Programmierung:
 - ▶ Datenabstraktion und Funktionale Abstraktion
 - ▶ Modularisierung
 - ▶ Typisierung und Spezifikation
- ▶ Funktionale Ideen jetzt im Mainstream:
 - ▶ Reflektion — LISP
 - ▶ Generics in Java — Polymorphie
 - ▶ Lambda-Fkt. in Java, C++ — Funktionen höherer Ordnung

Warum Haskell?



- ▶ **Moderne** Sprache
- ▶ Standardisiert, mehrere **Implementationen**
 - ▶ **Interpreter**: ghci, hugs
 - ▶ **Compiler**: ghc, nhc98
- ▶ **Rein** funktional
 - ▶ **Essenz** der funktionalen Programmierung

Geschichtliches: Die Anfänge

- ▶ **Grundlagen** 1920/30
 - ▶ Kombinatorlogik und λ -Kalkül (Schönfinkel, Curry, Church)
- ▶ Erste funktionale **Programmiersprachen** 1960
 - ▶ LISP (McCarthy), ISWIM (Landin)
- ▶ **Weitere** Programmiersprachen 1970– 80
 - ▶ FP (Backus); ML (Milner, Gordon); Hope (Burstall); Miranda (Turner)



Moses Schönfinkel Haskell B. Curry Alonzo Church John McCarthy John Backus Robin Milner Mike Gordon

Geschichtliches: Die Gegenwart

- ▶ **Konsolidierung** 1990
 - ▶ CAML, Formale Semantik für Standard ML
 - ▶ Haskell als Standardsprache

- ▶ **Kommerzialisierung** 2010
 - ▶ OCaml
 - ▶ Scala, Clojure (JVM)
 - ▶ F# (.NET)

Programme als Funktionen

- ▶ Programme als Funktionen:

$$P : \text{Eingabe} \rightarrow \text{Ausgabe}$$

- ▶ Keine veränderlichen **Variablen** — kein versteckter **Zustand**
- ▶ Rückgabewert hängt ausschließlich von Werten der Argumente ab, nicht vom Aufrufkontext (**referentielle Transparenz**)
- ▶ Alle **Abhängigkeiten explizit**

Beispiel: Programmieren mit Funktionen

- ▶ **Programme** werden durch **Gleichungen** definiert:

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

- ▶ Auswertung durch **Reduktion** von **Ausdrücken**:

fac 2

Beispiel: Programmieren mit Funktionen

- ▶ **Programme** werden durch **Gleichungen** definiert:

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

- ▶ Auswertung durch **Reduktion** von **Ausdrücken**:

fac 2 → **if** 2 == 0 **then** 1 **else** 2* fac (2-1)

Beispiel: Programmieren mit Funktionen

- ▶ **Programme** werden durch **Gleichungen** definiert:

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

- ▶ Auswertung durch **Reduktion** von **Ausdrücken**:

```
fac 2  → if 2 == 0 then 1 else 2* fac (2-1)  
       → if False then 1 else 2* fac 1
```

Beispiel: Programmieren mit Funktionen

- ▶ **Programme** werden durch **Gleichungen** definiert:

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

- ▶ Auswertung durch **Reduktion** von **Ausdrücken**:

```
fac 2  → if 2 == 0 then 1 else 2* fac (2-1)  
       → if False then 1 else 2* fac 1  
       → 2* fac 1
```

Beispiel: Programmieren mit Funktionen

- ▶ **Programme** werden durch **Gleichungen** definiert:

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

- ▶ Auswertung durch **Reduktion** von **Ausdrücken**:

```
fac 2  → if 2 == 0 then 1 else 2* fac (2-1)  
       → if False then 1 else 2* fac 1  
       → 2* fac 1  
       → 2* if 1 == 0 then 1 else 1* fac (1-1)
```

Beispiel: Programmieren mit Funktionen

- ▶ **Programme** werden durch **Gleichungen** definiert:

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

- ▶ Auswertung durch **Reduktion** von **Ausdrücken**:

```
fac 2 → if 2 == 0 then 1 else 2* fac (2-1)  
      → if False then 1 else 2* fac 1  
      → 2* fac 1  
      → 2* if 1 == 0 then 1 else 1* fac (1-1)  
      → 2* if False then 1 else 1* fac 0
```

Beispiel: Programmieren mit Funktionen

- ▶ **Programme** werden durch **Gleichungen** definiert:

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

- ▶ Auswertung durch **Reduktion** von **Ausdrücken**:

```
fac 2 → if 2 == 0 then 1 else 2* fac (2-1)  
      → if False then 1 else 2* fac 1  
      → 2* fac 1  
      → 2* if 1 == 0 then 1 else 1* fac (1-1)  
      → 2* if False then 1 else 1* fac 0  
      → 2* 1* fac 0
```

Beispiel: Programmieren mit Funktionen

- ▶ **Programme** werden durch **Gleichungen** definiert:

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

- ▶ Auswertung durch **Reduktion** von **Ausdrücken**:

```
fac 2  → if 2 == 0 then 1 else 2* fac (2-1)  
       → if False then 1 else 2* fac 1  
       → 2* fac 1  
       → 2* if 1 == 0 then 1 else 1* fac (1-1)  
       → 2* if False then 1 else 1* fac 0  
       → 2* 1* fac 0  
       → 2* 1* if 0 == 0 then 1 else 1* fac (0-1)
```

Beispiel: Programmieren mit Funktionen

- ▶ **Programme** werden durch **Gleichungen** definiert:

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

- ▶ Auswertung durch **Reduktion** von **Ausdrücken**:

```
fac 2 → if 2 == 0 then 1 else 2* fac (2-1)  
      → if False then 1 else 2* fac 1  
      → 2* fac 1  
      → 2* if 1 == 0 then 1 else 1* fac (1-1)  
      → 2* if False then 1 else 1* fac 0  
      → 2* 1* fac 0  
      → 2* 1* if 0 == 0 then 1 else 1* fac (0-1)  
      → 2* 1* if True then 1 else 1* fac (-1)
```

Beispiel: Programmieren mit Funktionen

- ▶ **Programme** werden durch **Gleichungen** definiert:

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

- ▶ Auswertung durch **Reduktion** von **Ausdrücken**:

```
fac 2 → if 2 == 0 then 1 else 2* fac (2-1)  
      → if False then 1 else 2* fac 1  
      → 2* fac 1  
      → 2* if 1 == 0 then 1 else 1* fac (1-1)  
      → 2* if False then 1 else 1* fac 0  
      → 2* 1* fac 0  
      → 2* 1* if 0 == 0 then 1 else 1* fac (0-1)  
      → 2* 1* if True then 1 else 1* fac (-1)  
      → 2* 1* 1 → 2
```


Beispiel: Nichtnumerische Werte

- ▶ Rechnen mit Zeichenketten

```
repeat n s = if n == 0 then ""  
             else s ++ repeat (n-1) s
```

- ▶ Auswertung:
 repeat 2 "hallo_"

Beispiel: Nichtnumerische Werte

- ▶ Rechnen mit Zeichenketten

```
repeat n s = if n == 0 then ""  
           else s ++ repeat (n-1) s
```

- ▶ Auswertung:

```
repeat 2 "hallo_"
```

```
→ if 2 == 0 then "" else "hallo_" ++ repeat (2-1) "hallo_"
```

Beispiel: Nichtnumerische Werte

- ▶ Rechnen mit Zeichenketten

```
repeat n s = if n == 0 then ""  
           else s ++ repeat (n-1) s
```

- ▶ Auswertung:

```
repeat 2 "hallo_"
```

```
→ if 2 == 0 then "" else "hallo_" ++ repeat (2-1) "hallo_"
```

```
→ "hallo_" ++ repeat 1 "hallo_"
```

Beispiel: Nichtnumerische Werte

- ▶ Rechnen mit Zeichenketten

```
repeat n s = if n == 0 then ""  
             else s ++ repeat (n-1) s
```

- ▶ Auswertung:

```
repeat 2 "hallo_"  
→ if 2 == 0 then "" else "hallo_" ++ repeat (2-1) "hallo_"  
→ "hallo_" ++ repeat 1 "hallo_"  
→ "hallo_" ++ if 1 == 0 then ""  
                else "hallo_" ++ repeat (1-1) "hallo_"
```

Beispiel: Nichtnumerische Werte

- ▶ Rechnen mit Zeichenketten

```
repeat n s = if n == 0 then ""  
           else s ++ repeat (n-1) s
```

- ▶ Auswertung:

```
repeat 2 "hallo_"  
→ if 2 == 0 then "" else "hallo_" ++ repeat (2-1) "hallo_"  
→ "hallo_" ++ repeat 1 "hallo_"  
→ "hallo_" ++ if 1 == 0 then ""  
                else "hallo_" ++ repeat (1-1) "hallo_"  
→ "hallo_" ++ ("hallo_" ++ repeat 0 "hallo_")
```

Beispiel: Nichtnumerische Werte

- ▶ Rechnen mit Zeichenketten

```
repeat n s = if n == 0 then ""  
             else s ++ repeat (n-1) s
```

- ▶ Auswertung:

```
repeat 2 "hallo_"  
→ if 2 == 0 then "" else "hallo_" ++ repeat (2-1) "hallo_"  
→ "hallo_" ++ repeat 1 "hallo_"  
→ "hallo_" ++ if 1 == 0 then ""  
                else "hallo_" ++ repeat (1-1) "hallo_"  
→ "hallo_" ++ ("hallo_" ++ repeat 0 "hallo_")  
→ "hallo_" ++ ("hallo_" ++ if 0 == 0 then ""  
                else repeat (0-1) "hallo_")
```

Beispiel: Nichtnumerische Werte

- ▶ Rechnen mit Zeichenketten

```
repeat n s = if n == 0 then ""  
           else s ++ repeat (n-1) s
```

- ▶ Auswertung:

```
repeat 2 "hallo_"  
→ if 2 == 0 then "" else "hallo_" ++ repeat (2-1) "hallo_"  
→ "hallo_" ++ repeat 1 "hallo_"  
→ "hallo_" ++ if 1 == 0 then ""  
                else "hallo_" ++ repeat (1-1) "hallo_"  
→ "hallo_" ++ ("hallo_" ++ repeat 0 "hallo_")  
→ "hallo_" ++ ("hallo_" ++ if 0 == 0 then ""  
                    else repeat (0-1) "hallo_")  
→ "hallo_" ++ ("hallo_" ++ "")
```

Beispiel: Nichtnumerische Werte

- ▶ Rechnen mit Zeichenketten

```
repeat n s = if n == 0 then ""  
             else s ++ repeat (n-1) s
```

- ▶ Auswertung:

```
repeat 2 "hallo_"  
→ if 2 == 0 then "" else "hallo_" ++ repeat (2-1) "hallo_"  
→ "hallo_" ++ repeat 1 "hallo_"  
→ "hallo_" ++ if 1 == 0 then ""  
                else "hallo_" ++ repeat (1-1) "hallo_"  
→ "hallo_" ++ ("hallo_" ++ repeat 0 "hallo_")  
→ "hallo_" ++ ("hallo_" ++ if 0 == 0 then ""  
                else repeat (0-1) "hallo_")  
→ "hallo_" ++ ("hallo_" ++ "")  
→ "hallo_hallo_"
```


Auswertung als Ausführungsbegriff

- ▶ **Programme** werden durch **Gleichungen** definiert:

$$f(x) = E$$

- ▶ **Auswertung** durch **Anwenden** der Gleichungen:

- ▶ Suchen nach **Vorkommen** von f , e.g. $f(t)$

- ▶ $f(t)$ wird durch $E \begin{bmatrix} t \\ x \end{bmatrix}$ ersetzt

- ▶ Auswertung kann **divergieren**!

Ausdrücke und Werte

- ▶ Nichtreduzierbare Ausdrücke sind **Werte**
- ▶ Vorgebenene **Basiswerte**: Zahlen, Zeichen
 - ▶ Durch **Implementation** gegeben
- ▶ Definierte **Datentypen**: Wahrheitswerte, Listen, ...
 - ▶ **Modellierung** von Daten

Typisierung

- ▶ **Typen** unterscheiden Arten von Ausdrücken und Werten:

| | | | | |
|--------|---|---------|---|--------------|
| repeat | n | s = ... | n | Zahl |
| | | | s | Zeichenkette |

- ▶ **Wozu** Typen?
 - ▶ Frühzeitiges Aufdecken “offensichtlicher” Fehler
 - ▶ Erhöhte **Programmsicherheit**
 - ▶ Hilfestellung bei **Änderungen**

Slogan

“Well-typed programs can't go wrong.”

— *Robin Milner*

Signaturen

- ▶ Jede Funktion hat eine **Signatur**

```
fac    :: Int → Int
```

```
repeat :: Int → String → String
```

- ▶ **Typüberprüfung**

- ▶ fac nur auf Int anwendbar, Resultat ist Int
- ▶ repeat nur auf Int und String anwendbar, Resultat ist String

Übersicht: Typen in Haskell

| Typ | Bezeichner | Beispiel | | |
|----------------|-------------------|----------|--------------|------|
| Ganze Zahlen | Int | 0 | 94 | -45 |
| Fließkomma | Double | 3.0 | 3.141592 | |
| Zeichen | Char | 'a' 'x' | '\034' | '\n' |
| Zeichenketten | String | "yuck" | "hi\nho\"\n" | |
| Wahrheitswerte | Bool | True | False | |
| Funktionen | $a \rightarrow b$ | | | |

- ▶ Später **mehr**. **Viel** mehr.

Das Rechnen mit Zahlen

Beschränkte Genauigkeit,
konstanter Aufwand \longleftrightarrow beliebige Genauigkeit,
wachsender Aufwand

Das Rechnen mit Zahlen

Beschränkte Genauigkeit,
konstanter Aufwand \longleftrightarrow beliebige Genauigkeit,
wachsender Aufwand

Haskell bietet die Auswahl:

- ▶ Int - ganze Zahlen als Maschinenworte (≥ 31 Bit)
- ▶ Integer - beliebig große ganze Zahlen
- ▶ Rational - beliebig genaue rationale Zahlen
- ▶ Float, Double - Fließkommazahlen (reelle Zahlen)

Ganze Zahlen: Int und Integer

- ▶ Nützliche Funktionen (**überladen**, auch für Integer):

```
+ , * , ^ , - :: Int -> Int -> Int
abs           :: Int -> Int — Betrag
div , quot   :: Int -> Int -> Int
mod , rem    :: Int -> Int -> Int
```

Es gilt: $(\text{div } x \ y) * y + \text{mod } x \ y = x$

- ▶ Vergleich durch $=$, \neq , \leq , $<$, ...
- ▶ **Achtung:** Unäres Minus
 - ▶ Unterschied zum Infix-Operator $-$
 - ▶ Im Zweifelsfall klammern: `abs (-34)`

Fließkommazahlen: Double

- ▶ Doppeltgenaue Fließkommazahlen (IEEE 754 und 854)
 - ▶ Logarithmen, Wurzel, Exponentiation, π und e , trigonometrische Funktionen
- ▶ Konversion in ganze Zahlen:
 - ▶ `fromIntegral :: Int, Integer → Double`
 - ▶ `fromInteger :: Integer → Double`
 - ▶ `round, truncate :: Double → Int, Integer`
 - ▶ Überladungen mit Typannotation auflösen:

```
round (fromInt 10) :: Int
```

- ▶ **Rundungsfehler!**

Alphanumerische Basisdatentypen: Char

- ▶ Notation für einzelne Zeichen: 'a', ...
- ▶ Nützliche Funktionen:

```
ord :: Char → Int  
chr :: Int → Char
```

```
toLower :: Char → Char  
toUpper :: Char → Char  
isDigit  :: Char → Bool  
isAlpha  :: Char → Bool
```

- ▶ Zeichenketten: String

Zusammenfassung

- ▶ **Programme** sind **Funktionen**, definiert durch **Gleichungen**
 - ▶ Referentielle Transparenz
 - ▶ kein impliziter Zustand, keine veränderlichen Variablen
- ▶ **Ausführung** durch **Reduktion** von Ausdrücken
- ▶ Typisierung:
 - ▶ **Basistypen**: Zahlen, Zeichen(ketten), Wahrheitswerte
 - ▶ Jede Funktion f hat eine Signatur $f :: a \rightarrow b$