

Praktische Informatik 3: Einführung in die Funktionale
Programmierung
Vorlesung vom 27.10.2010: Einführung

Christoph Lüth & Dennis Walter

Universität Bremen

Wintersemester 2010/11

Personal

- ▶ **Vorlesung:** Christoph Lüth <cxl@informatik.uni-bremen.de>
Cartesium 2.046, Tel. 64223
- ▶ **Tutoren:**
Diedrich Wolter <dwolter@informatik.uni-bremen.de>
Bernd Gersdorf <Bernd.Gersdorf@dfki.de>
Rene Wagner <Rene.Wagner@dfki.de>
Christian Maeder <Christian.Maeder@dfki.de>
Simon Ulbricht <tekknix@informatik.uni-bremen.de>
- ▶ **Fragestunde:** Berthold Hoffmann
<hof@informatik.uni-bremen.de>
- ▶ **Website:** www.informatik.uni-bremen.de/~cxl/lehre/pi3.ws10.

Termine

- ▶ **Vorlesung:** Mi 12 – 14, NW1 H 1 – H0020
- ▶ **Tutorien:**

Mo 10-12	MZH 5210	Christian Maeder
Mo 16-18	MZH 1380	Rene Wagner
Di 8-10	MZH 1100	Diedrich Wolter
Di 10-12	MZH 1380	Diedrich Wolter
Di 10-12	MZH 1400	Bernd Gersdorf
Di 12-14	MZH 1450	Simon Ulbricht
- ▶ **Fragestunde** : Do 9 – 11 Berthold Hoffmann (Cartesium 2.048)
- ▶ **Anmeldung** zu den Übungsgruppen über stud.ip

Übungsbetrieb

- ▶ Ausgabe der Übungsblätter über die Webseite **Donnerstag Mittag**
- ▶ Besprechung der Übungsblätter in den Tutorien
- ▶ **Bearbeitungszeit**: eine Woche
- ▶ **Abgabe**: elektronisch bis Montag um 10:00
- ▶ **Elf** Übungsblätter (voraussichtlich) plus 0. Übungsblatt
- ▶ Übungsgruppen: max. **drei Teilnehmer** (nur in Ausnahmefällen vier)

Scheinkriterien

- ▶ Von n Übungsblättern werden $n - 1$ bewertet (geplant $n = 11$)
- ▶ **Insgesamt** mind. 50% aller Punkte
- ▶ **Fachgespräch** (Individualität der Leistung) am Ende

Spielregeln

- ▶ **Quellen angeben** bei
 - ▶ Gruppenübergreifender Zusammenarbeit;
 - ▶ Internetrecherche, Literatur, etc.
- ▶ Erster Täuschungsversuch: **Null** Punkte
- ▶ Zweiter Täuschungsversuch: **Kein Schein**.
- ▶ **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
 - ▶ Rekursive Datentypen
 - ▶ Typvariablen und Polymorphie
 - ▶ Funktionen höherer Ordnung
 - ▶ Typinferenz
- ▶ Teil II: Funktionale Programmierung im Großen
- ▶ Teil III: Funktionale Programmierung im richtigen Leben

Warum funktionale Programmierung lernen?

- ▶ Denken in **Algorithmen**, nicht in **Programmiersprachen**
- ▶ **Abstraktion**: Konzentration auf das Wesentliche
- ▶ **Wesentliche** Elemente moderner Programmierung:
 - ▶ Datenabstraktion und Funktionale Abstraktion
 - ▶ Modularisierung
 - ▶ Typisierung und Spezifikation
- ▶ Blick über den Tellerrand — Blick in die Zukunft
- ▶ Studium \neq Programmierkurs — was kommt in 10 Jahren?

Warum Haskell?

- ▶ **Moderne** Sprache
- ▶ Standardisiert, mehrere **Implementationen**
 - ▶ **Interpreter**: hugs
 - ▶ **Compiler**: ghc, nhc98
- ▶ **Rein** funktional

Geschichtliches

- ▶ **Grundlagen** 1920/30
 - ▶ Kombinatorlogik und λ -Kalkül (Schönfinkel, Curry, Church)
- ▶ Erste **Programmiersprachen** 1960
 - ▶ LISP (McCarthy), ISWIM (Landin)
- ▶ **Weitere** Programmiersprachen 1970– 80
 - ▶ FP (Backus); ML (Milner, Gordon); Hope (Burstall); Miranda (Turner)
- ▶ **Konsolidierung** 1990
 - ▶ CAML, Formale Semantik für Standard ML
 - ▶ Haskell als Standardsprache
- ▶ **Hype** 2010
 - ▶ Scala, F#, Clojure

Referenzielle Transparenz

- ▶ 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 (referenzielle Transparenz)
- ▶ Alle Abhängigkeiten explizit

Definition von Funktionen

- ▶ Zwei wesentliche Konstrukte:
 - ▶ Fallunterscheidung
 - ▶ Rekursion

Satz

Fallunterscheidung und Rekursion auf natürlichen Zahlen sind **turing-mächtig**.

- ▶ Beispiel:

$$fac(n) = \begin{cases} 1 & \text{wenn } n = 0 \\ n \cdot fac(n - 1) & \text{sonst} \end{cases}$$

- ▶ Funktion kann **partiell** sein.

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**!

- ▶ **Operational** (Ausführungsbegriff) vs. **denotational** (math. Modell)

- ▶ Nichtreduzierbare Ausdrücke sind **Werte**

- ▶ Vorgebenene **Basiswerte**: Zahlen, Zeichen

- ▶ Definierte **Datentypen**: Wahrheitswerte, Listen, ...

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)

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)
```

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)
         ~> 2 * fac(2- 1)
```


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)
         ~> 2 * fac(2- 1)
         ~> 2 * fac(1)
```

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)  
        ~> 2* fac(2- 1)  
        ~> 2* fac(1)  
        ~> 2* (if 1== 0 then 1 else 1* fac(1- 1))
```

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)
        ~> 2 * fac(2- 1)
        ~> 2 * fac(1)
        ~> 2 * (if 1== 0 then 1 else 1 * fac(1- 1))
        ~> 2 * 1 * fac(1- 1)
```

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)
        ~> 2* fac(2- 1)
        ~> 2* fac(1)
        ~> 2* (if 1== 0 then 1 else 1* fac(1- 1))
        ~> 2* 1* fac(1- 1)
        ~> 2* 1* fac(0)
```

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)
         ~> 2 * fac(2- 1)
         ~> 2 * fac(1)
         ~> 2 * (if 1== 0 then 1 else 1 * fac(1- 1))
         ~> 2 * 1 * fac(1- 1)
         ~> 2 * 1 * fac(0)
         ~> 2 * 1 * (if 0== 0 then 1 else 0 * fac(0- 1))
```

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)
        ~> 2 * fac(2- 1)
        ~> 2 * fac(1)
        ~> 2 * (if 1== 0 then 1 else 1 * fac(1- 1))
        ~> 2 * 1 * fac(1- 1)
        ~> 2 * 1 * fac(0)
        ~> 2 * 1 * (if 0== 0 then 1 else 0 * fac(0- 1))
        ~> 2 * 1 * 1 ~> 2
```

Nichtnumerische Werte

- ▶ Rechnen mit Zeichenketten

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

- ▶ Auswertung:

```
repeat 2 "hallo "
```

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 "
```


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 (2-1) "hallo "
```

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 (2-1) "hallo "  
~> "hallo " ++ if 2-1 == 0 then ""  
                else "hallo " ++ repeat ((2-1)-1) "hallo "
```

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 (2-1) "hallo "  
~> "hallo "++ if 2-1 == 0 then ""  
             else "hallo "++ repeat ((2-1)-1) "hallo "  
~> "hallo "++ ("hallo "++ repeat ((2-1)-1) "hallo ")
```

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 (2-1) "hallo "  
~> "hallo "++ if 2-1 == 0 then ""  
             else "hallo "++ repeat ((2-1)-1) "hallo "  
~> "hallo "++ ("hallo "++ repeat ((2-1)-1) "hallo ")  
~> "hallo "++ ("hallo "++ if ((2-1)-1) == 0 then ""  
              else repeat (((2-1)-1)-1) "hallo ")
```

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 (2-1) "hallo "  
↪ "hallo " ++ if 2-1 == 0 then ""  
                else "hallo " ++ repeat ((2-1)-1) "hallo "  
↪ "hallo " ++ ("hallo " ++ repeat ((2-1)-1) "hallo ")  
↪ "hallo " ++ ("hallo " ++ if ((2-1)-1) == 0 then ""  
                    else repeat (((2-1)-1)-1) "hallo ")  
↪ "hallo " ++ ("hallo " ++ "")
```

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 (2-1) "hallo "  
~> "hallo " ++ if 2-1 == 0 then ""  
             else "hallo " ++ repeat ((2-1)-1) "hallo "  
~> "hallo " ++ ("hallo " ++ repeat ((2-1)-1) "hallo ")  
~> "hallo " ++ ("hallo " ++ if ((2-1)-1) == 0 then ""  
                 else repeat (((2-1)-1)-1) "hallo ")  
~> "hallo " ++ ("hallo " ++ "")  
~> "hallo hallo "
```

Typisierung

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

<code>repeat</code>	<code>n</code>	<code>s</code>	<code>= ...</code>	<code>n</code>	Zahl
				<code>s</code>	Zeichenkette

- ▶ Verschiedene Typen:
 - ▶ **Basistypen** (Zahlen, Zeichen)
 - ▶ **strukturierte Typen** (Listen, Tupel, etc)

Typisierung

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

repeat	n	s = ...	n	Zahl
			s	Zeichenkette

- ▶ Verschiedene Typen:

- ▶ **Basistypen** (Zahlen, Zeichen)
- ▶ **strukturierte Typen** (Listen, Tupel, etc)

- ▶ **Wozu** Typen?

- ▶ Typüberprüfung während **Übersetzung** erspart **Laufzeitfehler**
- ▶ **Programmsicherheit**

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-> b			

- ▶ Später mehr. Viel mehr.

Imperativ vs. Funktional

- ▶ **Imperative** Programmierung:

- ▶ Zustandsübergang $\Sigma \rightarrow \Sigma$, Lesen/Schreiben von Variablen

- ▶ Kontrollstrukturen: Fallunterscheidung `if ... then ... else`
Iteration `while ...`

- ▶ **Funktionale** Programmierung:

- ▶ Funktionen $f : E \rightarrow A$

- ▶ Kontrollstrukturen: Fallunterscheidung
Rekursion

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
 - ▶ **Strukturierte Typen**: Listen, Tupel
 - ▶ Jede Funktion f hat eine Signatur $f :: a \rightarrow b$