

Praktische Informatik 3
Einführung in die Funktionale Programmierung
Vorlesung vom 29.10.2008:
Einführung

Christoph Lüth

WS 08/09



Personal

- **Vorlesung:** Christoph Lüth <cxl>, Cartesium 2.046, Tel. 64223
- **Tutoren:** Dominik Luecke <luecke>
Klaus Hartke <hartke>
Marcus Ermler <maermLer>
Christian Maeder <maeder>
Ewaryst Schulz & Dominik Dietrich
- **Fragestunde:** Berthold Hoffmann <hof>
- **Website:** www.informatik.uni-bremen.de/~cxl/lehre/pi3.ws08.

Termine

- **Vorlesung:**
Mi 13 – 15, SFG 0140
- **Tutorien:**

Di	10 – 12	MZH 7210	Klaus Hartke
	17 – 19	MZH 1380	Marcus Ermler
Mi	8 – 10	MZH 7250	Ewaryst Schulz & Dominik Dietrich
Do	8 – 10	FZB 0240	Dominik Luecke
	10 – 12	Cart 0.01	Christian Maeder
- **Fragestunde (FAQ):**
Mi 10 – 12 Berthold Hoffmann (Cartesium 2.048)

Übungsbetrieb

- Ausgabe der Übungsblätter über die Webseite **Montag vormittag**
- Besprechung der Übungsblätter in den Tutorien
- **Bearbeitungszeit** zwei Wochen
- **Abgabe** elektronisch bis Montag um 10:00
- **Sechs** Übungsblätter (und ein Bonusblatt)
- **Übungsgruppen:** max. **drei** Teilnehmer (nur in Ausnahmefällen vier)

Scheinkriterien — Vorschlag:

- **Alle Übungsblätter** sind zu bearbeiten.
- **Pro Übungsblatt** mind. 50% aller Punkte
- Es gibt ein **Bonusübungsblatt**, um Ausfälle zu kompensieren.
- **Prüfungsgespräch** (Individualität der Leistung)

Spielregeln

- **Quellen angeben** bei
 - Gruppenübergreifender Zusammenarbeit;
 - Internetrecherche, Literatur, etc.
- **Erster Täuschungsversuch:**
 - **Null** Punkte
- **Zweiter Täuschungsversuch:** **Kein Schein.**
- **Deadline verpaßt?**
 - **Vorher** ankündigen, sonst **null** Punkte.

Fahrplan

- **Teil I: Grundlagen**
 - **Rekursion als Berechnungsmodell**
 - Rekursive Datentypen, rekursive Funktionen
 - Typvariablen und Polymorphie
 - Funktionen höherer Ordnung
 - Standarddatentypen
- **Teil II: Abstraktion**
- **Teil III: Beispiele, Anwendungen, Ausblicke**

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?

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), später SML und CAML; Hope (Burstall); Miranda (Turner)
- **Konsolidierung** 1990
 - CAML, Formale Semantik für Standard ML
 - Haskell als **Standardsprache**



Referentielle 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 (**referentielle Transparenz**)
- Alle **Abhängigkeiten** explizit



Programmieren mit Funktionen

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

```
inc x = x + 1
addDouble x y = 2*(x + y)
```

- Auswertung durch **Reduktion** von **Ausdrücken**:
addDouble (inc 5) 4 \rightsquigarrow 2*(inc 5 + 4) \rightsquigarrow 2*((5+1) + 4) \rightsquigarrow 20
- Nichtreduzierbare Ausdrücke sind **Werte**
 - Vorgegebene **Basiswerte**: Zahlen, Zeichen
 - Definierte **Datentypen**: Wahrheitswerte, Listen, ...



Definition von Funktionen

- Zwei wesentliche **Konstrukte**:

- Fallunterscheidung
- Rekursion

- **Beispiel**:

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

- Auswertung kann **divergieren!**



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



Nichtnumerische Werte

- Rechnen mit **Zeichenketten**

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

- **Auswertung**:

```
repeat 2 "hallo "
 $\rightsquigarrow$  "hallo " ++ repeat 1 "hallo "
 $\rightsquigarrow$  "hallo " ++ ("hallo " ++ repeat 0 "hallo ")
 $\rightsquigarrow$  "hallo " ++ ("hallo " ++ "")
 $\rightsquigarrow$  "hallo " ++ "hallo "
 $\rightsquigarrow$  "hallo hallo "
```



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

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
Listen	[a]	[6, 9, 20] ["oh", "dear"]
Tupel	(a, b)	(1, 'a') ('a', 4)
Funktionen	a-> b	



Auswertungsstrategien

- Von **außen** nach **innen** (outermost-first):
`inc (addDouble (inc 3) 4)`
 \rightsquigarrow `(addDouble (inc 3) 4) + 1`
 \rightsquigarrow `2 * (inc 3 + 4) + 1`
 \rightsquigarrow `2 * (3 + 1 + 4) + 1`
 \rightsquigarrow `2 * 8 + 1` \rightsquigarrow 17
- Von **innen** nach **außen** (innermost-first):
`inc (addDouble (inc 3) 4)`
 \rightsquigarrow `inc (addDouble (3 + 1) 4)`
 \rightsquigarrow `inc (2 * ((3 + 1) + 4))`
 \rightsquigarrow `(2 * ((3 + 1) + 4)) + 1`
 \rightsquigarrow `2 * 8 + 1` \rightsquigarrow 17



Auswertungsstrategien

- Outermost-first entspricht **call-by-need**, verzögerte Auswertung.
- Innermost-first entspricht **call-by-value**, strikte Auswertung
- Beispiel:
`div :: Int -> Int -> Int`
Ganzzahlige Division, undefiniert für `div n 0`

`mult :: Int -> Int -> Int`
`mult n m = if n == 0 then 0`
 `else (mult (n - 1) m) * m`

• Auswertung von `mult 0 (div 1 0)`



Striktheit

- Def: Funktion f ist **strikt** gdw.
Ergebnis ist undefiniert sobald ein Argument undefiniert ist
- Standard ML, Java, C etc. sind **strikt**
 - Haskell ist **nicht-strikt**
 - Fallunterscheidung ist **immer** nicht-strikt



Zusammenfassung

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

