

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:

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 $f :: a \rightarrow b$

Praktische Informatik 3: Einführung in die Funktionale
Programmierung
Vorlesung vom 03.11.2010: Funktionen und Datentypen

Christoph Lüth & Dennis Walter

Universität Bremen

Wintersemester 2010/11

Inhalt

- ▶ Auswertungsstrategien
 - ▶ Striktheit
- ▶ Definition von **Funktionen**
 - ▶ Syntaktische Feinheiten
- ▶ Definition von **Datentypen**
 - ▶ Aufzählungen
 - ▶ Produkte
- ▶ **Basisdatentypen:**
 - ▶ Wahrheitswerte, numerische Typen, alphanumerische Typen

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

Auswertungsstrategien

```
inc :: Int → Int  
inc x = x + 1
```

```
double :: Int → Int  
double x = 2 * x
```

- ▶ Reduktion von `inc (double (inc 3))`
- ▶ Von **außen** nach **innen** (outermost-first):
`inc (double (inc 3))` \rightsquigarrow

Auswertungsstrategien

```
inc :: Int → Int
inc x = x + 1
```

```
double :: Int → Int
double x = 2 * x
```

▶ Reduktion von `inc (double (inc 3))`

▶ Von **außen** nach **innen** (outermost-first):

$$\begin{aligned} \text{inc (double (inc 3))} &\rightsquigarrow \text{double (inc 3) + 1} \\ &\rightsquigarrow \end{aligned}$$

Auswertungsstrategien

```
inc :: Int → Int
inc x = x + 1
```

```
double :: Int → Int
double x = 2 * x
```

▶ Reduktion von `inc (double (inc 3))`

▶ Von **außen** nach **innen** (outermost-first):

`inc (double (inc 3))` \rightsquigarrow `double (inc 3) + 1`
 \rightsquigarrow `2 * (inc 3) + 1`
 \rightsquigarrow

Auswertungsstrategien

```
inc :: Int → Int
inc x = x + 1
```

```
double :: Int → Int
double x = 2 * x
```

▶ Reduktion von `inc (double (inc 3))`

▶ Von **außen** nach **innen** (outermost-first):

$$\begin{aligned} \text{inc (double (inc 3))} &\rightsquigarrow \text{double (inc 3) + 1} \\ &\rightsquigarrow 2 * (\text{inc 3}) + 1 \\ &\rightsquigarrow 2 * (3 + 1) + 1 \\ &\rightsquigarrow 2 * 4 + 1 \rightsquigarrow 9 \end{aligned}$$

Auswertungsstrategien

```
inc :: Int → Int
inc x = x + 1
```

```
double :: Int → Int
double x = 2 * x
```

▶ Reduktion von `inc (double (inc 3))`

▶ Von **außen** nach **innen** (outermost-first):

$$\begin{aligned} \text{inc (double (inc 3))} &\rightsquigarrow \text{double (inc 3) + 1} \\ &\rightsquigarrow 2 * (\text{inc 3}) + 1 \\ &\rightsquigarrow 2 * (3 + 1) + 1 \\ &\rightsquigarrow 2 * 4 + 1 \rightsquigarrow 9 \end{aligned}$$

▶ Von **innen** nach **außen** (innermost-first):

$$\text{inc (double (inc 3))} \rightsquigarrow$$

Auswertungsstrategien

```
inc :: Int → Int
inc x = x + 1
```

```
double :: Int → Int
double x = 2 * x
```

- ▶ Reduktion von `inc (double (inc 3))`

- ▶ Von **außen** nach **innen** (outermost-first):

$$\begin{aligned} \text{inc (double (inc 3))} &\rightsquigarrow \text{double (inc 3) + 1} \\ &\rightsquigarrow 2 * (\text{inc 3}) + 1 \\ &\rightsquigarrow 2 * (3 + 1) + 1 \\ &\rightsquigarrow 2 * 4 + 1 \rightsquigarrow 9 \end{aligned}$$

- ▶ Von **innen** nach **außen** (innermost-first):

$$\begin{aligned} \text{inc (double (inc 3))} &\rightsquigarrow \text{inc (double (3+1))} \\ &\rightsquigarrow \end{aligned}$$

Auswertungsstrategien

```
inc :: Int → Int
inc x = x + 1
```

```
double :: Int → Int
double x = 2 * x
```

▶ Reduktion von `inc (double (inc 3))`

▶ Von **außen** nach **innen** (outermost-first):

$$\begin{aligned} \text{inc (double (inc 3))} &\rightsquigarrow \text{double (inc 3) + 1} \\ &\rightsquigarrow 2 * (\text{inc 3}) + 1 \\ &\rightsquigarrow 2 * (3 + 1) + 1 \\ &\rightsquigarrow 2 * 4 + 1 \rightsquigarrow 9 \end{aligned}$$

▶ Von **innen** nach **außen** (innermost-first):

$$\begin{aligned} \text{inc (double (inc 3))} &\rightsquigarrow \text{inc (double (3 + 1))} \\ &\rightsquigarrow \text{inc (2 * (3 + 1))} \\ &\rightsquigarrow \end{aligned}$$

Auswertungsstrategien

```
inc :: Int → Int
inc x = x + 1
```

```
double :: Int → Int
double x = 2 * x
```

- ▶ Reduktion von `inc (double (inc 3))`

- ▶ Von **außen** nach **innen** (outermost-first):

$$\begin{aligned} \text{inc (double (inc 3))} &\rightsquigarrow \text{double (inc 3) + 1} \\ &\rightsquigarrow 2 * (\text{inc 3}) + 1 \\ &\rightsquigarrow 2 * (3 + 1) + 1 \\ &\rightsquigarrow 2 * 4 + 1 \rightsquigarrow 9 \end{aligned}$$

- ▶ Von **innen** nach **außen** (innermost-first):

$$\begin{aligned} \text{inc (double (inc 3))} &\rightsquigarrow \text{inc (double (3 + 1))} \\ &\rightsquigarrow \text{inc (2 * (3 + 1))} \\ &\rightsquigarrow (2 * (3 + 1)) + 1 \\ &\rightsquigarrow \end{aligned}$$

Auswertungsstrategien

```
inc :: Int → Int
inc x = x + 1
```

```
double :: Int → Int
double x = 2 * x
```

► Reduktion von `inc (double (inc 3))`

► Von **außen** nach **innen** (outermost-first):

$$\begin{aligned} \text{inc (double (inc 3))} &\rightsquigarrow \text{double (inc 3)} + 1 \\ &\rightsquigarrow 2 * (\text{inc 3}) + 1 \\ &\rightsquigarrow 2 * (3 + 1) + 1 \\ &\rightsquigarrow 2 * 4 + 1 \rightsquigarrow 9 \end{aligned}$$

► Von **innen** nach **außen** (innermost-first):

$$\begin{aligned} \text{inc (double (inc 3))} &\rightsquigarrow \text{inc (double (3+1))} \\ &\rightsquigarrow \text{inc (2*(3+1))} \\ &\rightsquigarrow (2*(3+1)) + 1 \\ &\rightsquigarrow 2*4 + 1 \rightsquigarrow 9 \end{aligned}$$

Auswertungsstrategien und Konfluenz

Theorem (Konfluenz)

*Funktionale Programme sind für jede Auswertungsstrategie **konfluent**.*

Theorem (Normalform)

***Terminierende** funktionale Programme werten unter jeder Auswertungsstrategie jeden Ausdruck zum gleichen Wert aus (der **Normalform**).*

- ▶ Auswertungsstrategie für **nicht-terminierende** Programme relevant.
- ▶ Nicht-Termination nötig (Turing-Mächtigkeit)

Auswirkung der Auswertungsstrategie

- ▶ 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 $\text{div } n \ 0$

```
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

Definition (Striktheit)

Funktion f ist **strikt** \iff Ergebnis ist undefiniert
sobald ein Argument undefiniert ist

- ▶ **Semantische** Eigenschaft (nicht operational)
- ▶ Standard ML, Java, C etc. sind **strikt** (nach Sprachdefinition)
- ▶ Haskell ist **nicht-strikt** (nach Sprachdefinition)
 - ▶ Meisten **Implementationen** nutzen **verzögerte Auswertung**
- ▶ Fallunterscheidung ist **immer** nicht-strikt

Wie definiere ich eine Funktion?

Generelle Form:

▶ **Signatur:**

```
max :: Int → Int → Int
```

▶ **Definition**

```
max x y = if x < y then y else x
```

- ▶ **Kopf**, mit Parametern
- ▶ **Rumpf** (evtl. länger, mehrere Zeilen)
- ▶ Typisches **Muster**: Fallunterscheidung, dann rekursiver Aufruf
- ▶ Was gehört zum Rumpf (**Geltungsbereich**)?

Haskell-Syntax: Charakteristika

- ▶ Leichtgewichtig
 - ▶ Wichtigstes Zeichen:
- ▶ Funktionsapplikation: `f a`
 - ▶ Keine Klammern
- ▶ Abseitsregel: Gültigkeitsbereich durch Einrückung
 - ▶ Keine Klammern
- ▶ Auch in anderen Sprachen (Python, Ruby)

Haskell-Syntax I: Die Abseitsregel

Funktionsdefinition:

$f\ x_1\ x_2\ \dots\ x_n = E$

- ▶ **Geltungsbereich** der Definition von f :
alles, was gegenüber f **engerückt** ist.
- ▶ Beispiel:

```
f x = hier faengts an
    und hier gehts weiter
      immer weiter
g y z = und hier faengt was neues an
```

- ▶ Gilt auch **verschachtelt**.
- ▶ Kommentare sind **passiv**

Haskell-Syntax II: Kommentare

- ▶ Pro Zeile: Ab `--` bis Ende der Zeile

```
f x y = irgendwas  -- und hier der Kommentar!
```

- ▶ Über mehrere Zeilen: Anfang `{-`, Ende `-}`

```
{-  
  Hier fängt der Kommentar an  
  erstreckt sich über mehrere Zeilen  
  bis hier                               -}  
f x y = irgendwas
```

- ▶ Kann geschachtelt werden.

Haskell-Syntax III: Bedingte Definitionen

- ▶ Statt verschachtelter Fallunterscheidungen ...

```
f x y = if B1 then P else  
        if B2 then Q else ...
```

... **bedingte Gleichungen**:

```
f x y  
  | B1 = ...  
  | B2 = ...
```

- ▶ Auswertung der Bedingungen von oben nach unten
- ▶ Wenn keine Bedingung wahr ist: **Laufzeitfehler!** Deshalb:

```
| otherwise = ...
```

Haskell-Syntax IV: Lokale Definitionen

- ▶ Lokale Definitionen mit `where` oder `let`:

```
f x y
  | g = P y
  | otherwise =
Q where
  y = M
  f x = N x
```

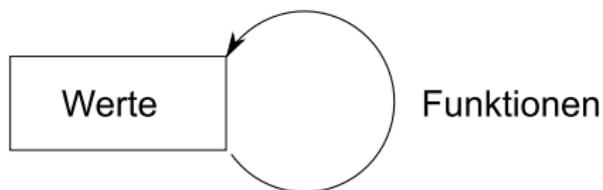
```
f x y =
  let y = M
      f x = N x
  in  if g then P y
      else Q
```

- ▶ `f`, `y`, ... werden **gleichzeitig** definiert (Rekursion!)
- ▶ Namen `f`, `y` und Parameter (`x`) **überlagern** andere
- ▶ Es gilt die **Abseitsregel**
 - ▶ Deshalb: Auf **gleiche Einrückung** der lokalen Definition achten!

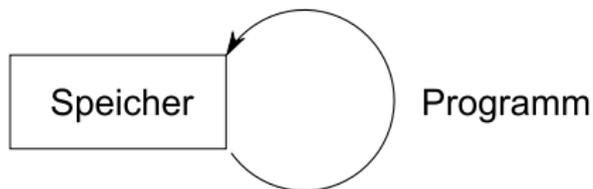
Datentypen als Modellierungskonstrukt

Programme **manipulieren** ein **Modell** (der Umwelt)

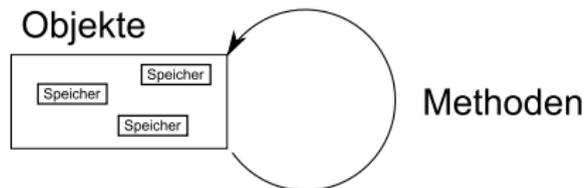
▶ **Funktionale** Sicht:



▶ **Imperative** Sicht:



▶ **Objektorientierte** Sicht:



Datentypen, Funktionen und Beweise

- ▶ Datentypen konstruieren **Werte**
- ▶ Funktionen definieren **Berechnungen**
- ▶ Berechnungen haben **Eigenschaften**
- ▶ Dualität:

Datentypkonstruktor \longleftrightarrow Definitionskonstrukt \longleftrightarrow Beweiskonstrukt

Typkonstruktoren

- ▶ Aufzählungen
- ▶ Produkt
- ▶ Rekursion
- ▶ Funktionsraum

Aufzählungen

- ▶ Aufzählungen: Menge von **disjunkten** Konstanten

$$\text{Days} = \{\text{Mon}, \text{Tue}, \text{Wed}, \text{Thu}, \text{Fri}, \text{Sat}, \text{Sun}\}$$

$$\text{Mon} \neq \text{Tue}, \text{Mon} \neq \text{Wed}, \text{Tue} \neq \text{Thu}, \text{Wed} \neq \text{Sun} \dots$$

- ▶ Genau sieben **unterschiedliche** Konstanten
- ▶ Funktion mit **Wertebereich** *Days* muss sieben Fälle unterscheiden
- ▶ Beispiel: *weekend* : *Days* \rightarrow *Bool* mit

$$\text{weekend}(d) = \begin{cases} \text{true} & d = \text{Sat} \vee d = \text{Sun} \\ \text{false} & d = \text{Mon} \vee d = \text{Tue} \vee d = \text{Wed} \vee \\ & d = \text{Thu} \vee d = \text{Fri} \end{cases}$$

Aufzählung und Fallunterscheidung in Haskell

► Definition

```
data Days = Mon | Tue | Wed | Thu | Fri | Sat | Sun
```

- Implizite Deklaration der **Konstruktoren** Mon :: Days als Konstanten
- Großschreibung der Konstruktoren

► Fallunterscheidung:

```
weekend :: Days → Bool  
weekend d = case d of  
  Sat → True  
  Sun → True  
  Mon → False  
  Tue → False  
  Wed → False  
  Thu → False  
  Fri → False
```

Aufzählung und Fallunterscheidung in Haskell

► Definition

```
data Days = Mon | Tue | Wed | Thu | Fri | Sat | Sun
```

- Implizite Deklaration der **Konstruktoren** Mon :: Days als Konstanten
- Großschreibung der Konstruktoren

► Fallunterscheidung:

```
weekend :: Days → Bool  
weekend d = case d of  
  Sat → True  
  Sun → True  
  Mon → False  
  Tue → False  
  Wed → False  
  Thu → False  
  Fri → False
```

```
weekend d =  
case d of  
  Sat → True  
  Sun → True  
  _   → False
```

Fallunterscheidung in der Funktionsdefinition

- ▶ Abkürzende Schreibweise (**syntaktischer Zucker**):

$$\begin{array}{ccc} f\ c_1 == e_1 & & f\ x == \mathbf{case\ } x \mathbf{ of\ } c_1 \rightarrow e_1, \\ \dots & \longrightarrow & \dots \\ f\ c_n == e_n & & c_n \rightarrow e_n \end{array}$$

- ▶ Damit:

```
weekend :: Days → Bool
weekend Sat = True
weekend Sun = True
weekend _   = False
```

Der einfachste Aufzählungstyp

- ▶ **Einfachste** Aufzählung: Wahrheitswerte

$$Bool = \{ True, False \}$$

- ▶ Genau zwei unterschiedliche Werte
- ▶ **Definition** von Funktionen:
 - ▶ Wertetabellen sind explizite Fallunterscheidungen

\wedge	<i>true</i>	<i>false</i>
<i>true</i>	<i>true</i>	<i>false</i>
<i>false</i>	<i>false</i>	<i>false</i>

$$\begin{aligned} true \wedge true &= true \\ true \wedge false &= false \\ false \wedge true &= false \\ false \wedge false &= false \end{aligned}$$

Wahrheitswerte: Bool

- ▶ **Vordefiniert** als

```
data Bool = True | False
```

- ▶ Vordefinierte **Funktionen**:

```
not  :: Bool → Bool      — Negation
&&   :: Bool → Bool → Bool — Konjunktion
||   :: Bool → Bool → Bool — Disjunktion
```

- ▶ **Konjunktion** definiert als

```
a && b = case a of True  → b
                False → False
```

- ▶ `&&`, `||` sind rechts nicht strikt
 - ▶ `1 == 0 && div 1 0 == 0` \rightsquigarrow False
- ▶ if then else als syntaktischer Zucker:

$$\text{if } b \text{ then } p \text{ else } q \longrightarrow \text{case } b \text{ of True } \rightarrow p \\ \text{False } \rightarrow q$$

Beispiel: Ausschließende Disjunktion

- ▶ Mathematische Definition:

```
exOr :: Bool → Bool → Bool
exOr x y = (x || y) && (not (x && y))
```

- ▶ Alternative 1: explizite Wertetabelle:

```
exOr False False = False
exOr True  False = True
exOr False True  = True
exOr True  True  = False
```

- ▶ Alternative 2: Fallunterscheidung auf ersten Argument

```
exOr True  y = not y
exOr False y = y
```

- ▶ Was ist am besten?
 - ▶ Effizienz, Lesbarkeit, Striktheit

Produkte

- ▶ Konstruktoren können **Argumente** haben
- ▶ Beispiel: Ein **Datum** besteht aus **Tag**, **Monat**, **Jahr**
- ▶ Mathematisch: Produkt (Tupel)

$$\begin{aligned} \text{Date} &= \{ \text{Date}(n, m, y) \mid n \in \mathbb{N}, m \in \text{Month}, y \in \mathbb{N} \} \\ \text{Month} &= \{ \text{Jan}, \text{Feb}, \text{Mar}, \dots \} \end{aligned}$$

- ▶ **Funktionsdefinition:**
 - ▶ Konstruktorargumente sind **gebundene Variablen**

$$\begin{aligned} \text{year}(D(n, m, y)) &= y \\ \text{day}(D(n, m, y)) &= n \end{aligned}$$

- ▶ Bei der **Auswertung** wird **gebundene Variable** durch **konkretes Argument** ersetzt

Produkte in Haskell

- ▶ Konstruktoren mit Argumenten

```
data Date = Date Int Month Int
data Month = Jan | Feb | Mar | Apr | May | Jun
           | Jul | Aug | Sep | Oct | Nov | Dec
```

- ▶ Beispielwerte:

```
today      = Date 5 Nov 2008
bloomsday  = Date 16 Jun 1904
```

- ▶ Über **Fallunterscheidung** Zugriff auf Argumente der Konstruktoren:

```
day  :: Date → Int
year :: Date → Int
day  d = case d of Date t m y → t
year (Date d m y) = y
```

Beispiel: Tag im Jahr

- ▶ Tag im Jahr: Tag im laufenden Monat plus Summe der Anzahl der Tage der vorherigen Monate

```
yearDay :: Date → Int
yearDay (Date d m y) = d + sumPrevMonths m where
  sumPrevMonths :: Month → Int
  sumPrevMonths Jan = 0
  sumPrevMonths m   = daysInMonth (prev m) y +
    sumPrevMonths (prev m)
```

- ▶ Tage im Monat benötigt Jahr als Argument (Schaltjahr!)

```
daysInMonth :: Month → Int → Int
prev :: Month → Month
```

- ▶ Schaltjahr: Gregorianischer Kalender

```
leapyear :: Int → Bool
leapyear y = if mod y 100 == 0 then mod y 400 == 0
             else mod y 4 == 0
```

Fallunterscheidung und Produkte

- ▶ Beispiel: geometrische Objekte

- ▶ Dreieck, gegeben durch Kantenlänge

- ▶ Kreis, gegeben durch Radius

- ▶ Rechteck, gegeben durch zwei Kantenlängen

$$O = \{Tri(a) \mid a \in \mathbb{R}\} \cup \{Circle(r) \mid r \in \mathbb{R}\} \cup \{Rect(a, b) \mid a, b \in \mathbb{R}\}$$

```
data Obj = Tri Double | Circle Double | Rect Double
```

Fallunterscheidung und Produkte

- ▶ Beispiel: geometrische Objekte

- ▶ Dreieck, gegeben durch Kantenlänge

- ▶ Kreis, gegeben durch Radius

- ▶ Rechteck, gegeben durch zwei Kantenlängen

$$O = \{Tri(a) \mid a \in \mathbb{R}\} \cup \{Circle(r) \mid r \in \mathbb{R}\} \cup \{Rect(a, b) \mid a, b \in \mathbb{R}\}$$

```
data Obj = Tri Double | Circle Double | Rect Double
```

- ▶ Berechnung des Umfangs:

```
circ :: Obj -> Double
circ (Tri a) = 3 * a
circ (Circle r) = 2 * pi * r
circ (Rect a b) = 2 * (a + b)
```

Der Allgemeine Fall: Algebraische Datentypen

Definition eines **algebraischen Datentypen** T :

$$\begin{array}{l} \text{data } T = C_1 t_{1,1} \dots t_{1,k_1} \\ \quad \quad \quad \dots \\ \quad \quad \quad | C_n t_{n,1} \dots t_{n,k_n} \end{array}$$

► Konstruktoren C_1, \dots, C_n sind **disjunkt**:

$$C_i x_1 \dots x_n = C_j y_1 \dots y_m \longrightarrow i = j$$

► Konstruktoren sind **injektiv**:

$$C x_1 \dots x_n = C y_1 \dots y_n \longrightarrow x_i = y_i$$

► Konstruktoren **erzeugen** den Datentyp:

$$\forall x \in T. x = C_i y_1 \dots y_m$$

Diese Eigenschaften machen **Fallunterscheidung** möglich.

Beweis von Eigenschaften

- ▶ Eigenschaften von Programmen: **Prädikate**
 - ▶ Haskell-Ausdrücke vom Typ Bool
 - ▶ Allquantifizierte Aussagen:
wenn $P(x)$ Prädikat, dann ist $\forall x.P(x)$ auch ein Prädikat
 - ▶ Sonderfall Gleichungen $s == t$
 - ▶ Müssen nicht **ausführbar** sein

Wie beweisen?

- ▶ Gleichungsumformung (equational reasoning)
- ▶ Fallunterscheidungen
- ▶ Induktion
- ▶ Wichtig: formale Notation

Ein ganz einfaches Beispiel

```
addTwice :: Int → Int → Int
addTwice x y = 2*(x+ y)
```

Lemma: $\frac{\text{addTwice } x \text{ (} y + z \text{)} = \text{addTwice (} x + y \text{) } z}{\text{addTwice } x \text{ (} y + z \text{)}}$

Ein ganz einfaches Beispiel

```
addTwice :: Int → Int → Int
addTwice x y = 2*(x+ y)
```

$$\begin{array}{l} \textbf{Lemma: } \text{addTwice } x \ (y + z) = \text{addTwice } (x + y) \ z \\ \hline \text{addTwice } x \ (y + z) \\ = 2 * (x + (y + z)) \quad \text{— Def. addTwice} \end{array}$$

Ein ganz einfaches Beispiel

```
addTwice :: Int → Int → Int
addTwice x y = 2*(x+ y)
```

$$\begin{array}{l} \textbf{Lemma: } \text{addTwice } x \ (y + z) = \text{addTwice } (x + y) \ z \\ \hline \text{addTwice } x \ (y + z) \\ = 2 * (x + (y + z)) \quad \text{— Def. addTwice} \\ = 2 * ((x + y) + z) \quad \text{— Assoziativität von } + \end{array}$$

Ein ganz einfaches Beispiel

```
addTwice :: Int → Int → Int
addTwice x y = 2*(x+ y)
```

$$\begin{aligned} \text{Lemma: } & \text{addTwice } x \ (y + z) = \text{addTwice } (x + y) \ z \\ \hline & \text{addTwice } x \ (y + z) \\ & = 2 * (x + (y + z)) \quad \text{— Def. addTwice} \\ & = 2 * ((x + y) + z) \quad \text{— Assoziativität von } + \\ & = \text{addTwice } (x + y) \ z \quad \text{— Def. addTwice} \\ & \square \end{aligned}$$

Fallunterscheidung

```
max, min :: Int → Int → Int
max x y = if x < y then y else x
min x y = if x < y then x else y
```

Lemma: $\max x y - \min x y = |x - y|$

$\max x y - \min x y$

Fallunterscheidung

```
max, min :: Int → Int → Int
max x y = if x < y then y else x
min x y = if x < y then x  else y
```

Lemma: $\max x y - \min x y = |x - y|$

$\max x y - \min x y$

• Fall: $x < y$

$= y - \min x y$ — Def. max

$= y - x$ — Def. min

$= |x - y|$ — Wenn $x < y$, dann $y - x = |x - y|$

Fallunterscheidung

```
max, min :: Int → Int → Int
max x y = if x < y then y else x
min x y = if x < y then x  else y
```

Lemma: $\max x y - \min x y = |x - y|$

$\max x y - \min x y$

- Fall: $x < y$

$= y - \min x y$ — Def. max

$= y - x$ — Def. min

$= |x - y|$ — Wenn $x < y$, dann $y - x = |x - y|$

- Fall: $x \geq y$

$= x - \min x y$ — Def. max

$= x - y$ — Def. min

$= |x - y|$ — Wenn $x \geq y$, dann $x - y = |x - y|$

Fallunterscheidung

```
max, min :: Int → Int → Int
max x y = if x < y then y else x
min x y = if x < y then x  else y
```

Lemma: $\max x y - \min x y = |x - y|$

$\max x y - \min x y$

• Fall: $x < y$

$= y - \min x y$ — Def. max

$= y - x$ — Def. min

$= |x - y|$ — Wenn $x < y$, dann $y - x = |x - y|$

• Fall: $x \geq y$

$= x - \min x y$ — Def. max

$= x - y$ — Def. min

$= |x - y|$ — Wenn $x \geq y$, dann $x - y = |x - y|$

$= |x - y|$

□

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

- ▶ **Striktheit**
 - ▶ Haskell ist **spezifiziert** als nicht-strikt
- ▶ Datentypen und Funktionsdefinition **dual**
 - ▶ **Aufzählungen** — **Fallunterscheidung**
 - ▶ **Produkte**
- ▶ Funktionsdefinition und Beweis **dual**
 - ▶ Beweis durch **Gleichungsumformung**
 - ▶ Programmeigenschaften als **Prädikate**
 - ▶ **Fallunterscheidung** als Beweiskonstrukt
- ▶ Wahrheitswerte Bool
- ▶ Numerische Basisdatentypen:
 - ▶ Int, Integer, Rational und Double
- ▶ Alphanumerische Basisdatentypen: Char
- ▶ **Nächste Vorlesung**: Rekursive Datentypen

Praktische Informatik 3: Einführung in die Funktionale
Programmierung
Vorlesung vom 10.11.2010: Rekursive Datentypen

Christoph Lüth & Dennis Walter

Universität Bremen

Wintersemester 2010/11

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

Inhalt

- ▶ **Rekursive** Datentypen
 - ▶ Formen der Rekursion
 - ▶ Rekursive **Definition**
 - ▶ Rekursive Datentypen in anderen Sprachen
- ▶ **Induktiver** Beweis
 - ▶ Schluss vom **kleineren** aufs **größere**

Der Allgemeine Fall: Algebraische Datentypen

Definition eines **algebraischen Datentypen** T :

$$\begin{array}{l} \text{data } T = C_1 t_{1,1} \dots t_{1,k_1} \\ \quad \quad \quad \dots \\ \quad \quad \quad | C_n t_{n,1} \dots t_{n,k_n} \end{array}$$

► Konstruktoren C_1, \dots, C_n sind **disjunkt**:

$$C_i x_1 \dots x_n = C_j y_1 \dots y_m \longrightarrow i = j$$

► Konstruktoren sind **injektiv**:

$$C x_1 \dots x_n = C y_1 \dots y_n \longrightarrow x_i = y_i$$

► Konstruktoren **erzeugen** den Datentyp:

$$\forall x \in T. x = C_i y_1 \dots y_m$$

Diese Eigenschaften machen **Fallunterscheidung** möglich.

Rekursive Datentypen

- ▶ Der definierte Typ T kann **rechts** benutzt werden.
- ▶ Rekursive Datentypen sind **unendlich**
- ▶ Entspricht **induktiver Definition**

Induktive Definitionen

- ▶ Beispiel natürliche Zahlen: Peano-Axiome

- ▶ $0 \in \mathbb{N}$

- ▶ wenn $n \in \mathbb{N}$, dann $S n \in \mathbb{N}$

- ▶ S injektiv und $S n \neq 0$

- ▶ Induktionsprinzip: $\phi(0), \phi(x) \longrightarrow \phi(S x)$, dann $\forall n \in \mathbb{N}.\phi(n)$

- ▶ Induktionsprinzip erlaubt Definition rekursiver Funktionen:

$$\begin{aligned}n + 0 &= n \\n + S m &= S(n + m)\end{aligned}$$

Natürliche Zahlen in Haskell

- ▶ Direkte Übersetzung der Peano-Axiome

- ▶ Der **Datentyp**:

```
data Nat = Zero  
         | S Nat
```

- ▶ Rekursive **Funktionsdefinition**:

```
add :: Nat → Nat → Nat  
add n Zero   = n  
add n (S m)  = S (add n m)
```

Beispiel: Zeichenketten selbstgemacht

- ▶ Eine **Zeichenkette** ist
 - ▶ entweder **leer** (das leere Wort ϵ)
 - ▶ oder ein **Zeichen** c und eine weitere **Zeichenkette** xs

```
data MyString = Empty  
              | Cons Char MyString
```

- ▶ **Lineare** Rekursion
 - ▶ Genau ein rekursiver Aufruf

Rekursive Definition

- ▶ Typisches Muster: Fallunterscheidung
 - ▶ Ein Fall pro Konstruktor
- ▶ Hier:
 - ▶ Leere Zeichenkette
 - ▶ Nichtleere Zeichenkette

Funktionen auf Zeichenketten

► Länge:

```
len :: MyString → Int
len Empty          = 0
len (Cons c str)  = 1 + len str
```

Funktionen auf Zeichenketten

► Länge:

```
len :: MyString → Int
len Empty          = 0
len (Cons c str)  = 1 + len str
```

► Verkettung:

```
cat :: MyString → MyString → MyString
cat Empty t      = t
cat (Cons c s) t = Cons c (cat s t)
```

Funktionen auf Zeichenketten

► Länge:

```
len :: MyString → Int
len Empty          = 0
len (Cons c str) = 1 + len str
```

► Verkettung:

```
cat :: MyString → MyString → MyString
cat Empty t      = t
cat (Cons c s) t = Cons c (cat s t)
```

► Umkehrung:

```
rev :: MyString → MyString
rev Empty          = Empty
rev (Cons c t) = cat (rev t) (Cons c Empty)
```

Baumartige Rekursion: Binäre Bäume

- ▶ Datentyp:

```
data BTree = MtBTree  
          | BNode Int BTree BTree
```

- ▶ Funktion, bsp. Höhe:

```
height :: BTree → Int  
height MtBTree = 0  
height (BNode j l r) = max (height l) (height r) + 1
```

- ▶ Baumartige Rekursion
 - ▶ Doppelter rekursiver Aufruf

Wechselseitige Rekursion: Variadische Bäume

- ▶ Variable Anzahl Kinderknoten

```
data VTree = MtVTree
          | VNode String VNodes

data VNodes = MtVNodes
            | VMore VTree VNodes
```

- ▶ VNodes: Liste von Kinderbäumen

Wechselseitige Rekursion: Variadische Bäume

▶ Hauptfunktion:

```
count :: VTree → Int
count MtVTree = 0
count (VNode _ ns) = 1 + count_nodes ns
```

▶ Hilfsfunktion:

```
count_nodes :: VNodes → Int
count_nodes MtVNodes = 0
count_nodes (VMore t ns) = count t + count_nodes ns
```

Rekursive Typen in anderen Sprachen

- ▶ **Standard ML**: gleich
- ▶ **Lisp**: keine Typen, aber alles ist eine S-Expression

```
data SExpr = Quote Atom | Cons SExpr SExpr
```

- ▶ **Python, Ruby**:
 - ▶ Listen (Sequenzen) vordefiniert
 - ▶ Keine anderen Typen

Rekursive Typen in Java

- ▶ Nachbildung durch Klassen, z.B. für Listen:

```
class List {  
    public List(Object el, List tl) {  
        this.elem = el;  
        this.next = tl;  
    }  
    public Object elem;  
    public List next;  
}
```

- ▶ Länge (iterativ):

```
int length() {  
    int i = 0;  
    for (List cur = this; cur != null; cur = cur.next)  
        i++;  
    return i;  
}
```

Rekursive Typen in C

- ▶ **C**: Produkte, Aufzählungen, keine rekursiven Typen
- ▶ Rekursion durch **Zeiger**

```
typedef struct list_t {  
    void          *elem;  
    struct list_t *next;  
} *list;
```

- ▶ Konstruktoren **nutzerimplementiert**

```
list cons(void *hd, list tl)  
{  
    list l;  
    if ((l = (list) malloc(sizeof(struct list_t))) == NULL)  
        printf("Out of memory\n"); exit(-1);  
}  
l->elem = hd; l->next = tl;  
return l;  
}
```

Rekursive Definition, induktiver Beweis

- ▶ Definition durch **Rekursion**

- ▶ Basisfall (leere Zeichenkette)

- ▶ Rekursion (nicht-leere Zeichenkette)

```
rev :: MyString → MyString
rev Empty      = Empty
rev (Cons c t) = cat (rev t) (Cons c Empty)
```

- ▶ Reduktion der Eingabe (vom größeren aufs kleinere)

- ▶ **Beweis** durch Induktion

- ▶ Schluß vom kleineren aufs größere

Beweis durch vollständige Induktion

Zu zeigen:

Für alle natürlichen Zahlen x gilt $P(x)$.

Beweis:

- ▶ Induktionsbasis: $P(0)$
- ▶ Induktionsschritt:
 - ▶ Induktionsvoraussetzung $P(x)$
 - ▶ zu zeigen $P(x + 1)$

Beweis durch strukturelle Induktion (Zeichenketten)

Zu zeigen:

Für alle (endlichen) Zeichenketten xs gilt $P(xs)$

Beweis:

- ▶ Induktionsbasis: $P(\epsilon)$
- ▶ Induktionsschritt:
 - ▶ Induktionsvoraussetzung $P(xs)$
 - ▶ zu zeigen $P(x xs)$

Beweis durch strukturelle Induktion (Allgemein)

Zu zeigen:

Für alle x in T gilt $P(x)$

Beweis:

- ▶ Für jeden Konstruktor C_i :
 - ▶ Voraussetzung: für alle $t_{i,j}$ gilt $P(t_{i,j})$
 - ▶ zu zeigen $P(C_i t_{i,1} \dots t_{i,k_i})$

Beispielbeweise

$$\text{len } s \geq 0 \quad (1)$$

$$\text{len } (\text{cat } s \ t) = \text{len } s + \text{len } t \quad (2)$$

$$\text{len } (\text{rev } s) = \text{len } s \quad (3)$$

Spezifikation und Korrektheit

- ▶ Die ersten n Zeichen einer Zeichenkette ($n \geq 0$)

```
takeN :: Int → MyString → MyString
takeN n Empty = Empty
takeN n (Cons c s) = if n == 0 then Empty
                    else Cons c (takeN (n-1) s)
```

Spezifikation und Korrektheit

- ▶ Die ersten n Zeichen einer Zeichenkette ($n \geq 0$)

```
takeN :: Int → MyString → MyString
takeN n Empty = Empty
takeN n (Cons c s) = if n == 0 then Empty
                    else Cons c (takeN (n-1) s)
```

- ▶ Zeichenkette ohne die ersten n Zeichen ($n \geq 0$)

```
dropN :: Int → MyString → MyString
dropN n Empty = Empty
dropN n (Cons c s) = if n == 0 then Cons c s
                    else dropN (n-1) s
```

Spezifikation und Korrektheit

- ▶ Die ersten n Zeichen einer Zeichenkette ($n \geq 0$)

```
takeN :: Int → MyString → MyString
takeN n Empty = Empty
takeN n (Cons c s) = if n == 0 then Empty
                    else Cons c (takeN (n-1) s)
```

- ▶ Zeichenkette ohne die ersten n Zeichen ($n \geq 0$)

```
dropN :: Int → MyString → MyString
dropN n Empty = Empty
dropN n (Cons c s) = if n == 0 then Cons c s
                    else dropN (n-1) s
```

- ▶ Eigenschaften:

$$\text{len (takeN } n \text{ } s) \leq n \quad (4)$$

$$\text{len (dropN } n \text{ } s) \geq \text{len } s - n \quad (5)$$

$$\text{cat (takeN } n \text{ } s) \text{ (dropN } n \text{ } s) = s \quad (6)$$

Zusammenfassung

- ▶ Datentypen können **rekursiv** sein
- ▶ Rekursive Datentypen sind **unendlich** (induktiv)
- ▶ Funktionen werden **rekursiv** definiert
 - ▶ **Formen** der Rekursion: linear, baumartig, wechselseitig
- ▶ **Rekursive** Definition ermöglicht **induktiven** Beweis
- ▶ **Nächste** Woche: Abstraktion über Typen (Polymorphie)

Praktische Informatik 3: Einführung in die Funktionale
Programmierung
Vorlesung vom 17.11.2010: Typvariablen und Polymorphie

Christoph Lüth & Dennis Walter

Universität Bremen

Wintersemester 2010/11

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

Inhalt

- ▶ Letzte Vorlesung: rekursive Datentypen
- ▶ Diese Vorlesung:
 - ▶ **Abstraktion** über Typen: **Typvariablen** und **Polymorphie**

Zeichenketten und Listen von Zahlen

- ▶ Letzte VL: Eine **Zeichenkette** ist
 - ▶ entweder **leer** (das leere Wort ϵ)
 - ▶ oder ein **Zeichen** und eine weitere **Zeichenkette**

```
data MyString = Empty  
              | Cons Char MyString
```

Zeichenketten und Listen von Zahlen

- ▶ Letzte VL: Eine **Zeichenkette** ist
 - ▶ entweder **leer** (das leere Wort ϵ)
 - ▶ oder ein **Zeichen** und eine weitere **Zeichenkette**

```
data MyString = Empty
              | Cons Char MyString
```

- ▶ Eine **Liste von Zahlen** ist
 - ▶ entweder **leer**
 - ▶ oder eine **Zahl** und eine weitere **Liste**

```
data IntList = Empty
             | Cons Int IntList
```

Zeichenketten und Listen von Zahlen

- ▶ Letzte VL: Eine **Zeichenkette** ist
 - ▶ entweder **leer** (das leere Wort ϵ)
 - ▶ oder ein **Zeichen** und eine weitere **Zeichenkette**

```
data MyString = Empty
              | Cons Char MyString
```

- ▶ Eine **Liste von Zahlen** ist
 - ▶ entweder **leer**
 - ▶ oder eine **Zahl** und eine weitere **Liste**

```
data IntList = Empty
             | Cons Int IntList
```

- ▶ Strukturell **gleiche** Definition
 \rightsquigarrow Zwei Instanzen einer **allgemeineren** Definition.

Typvariablen

- ▶ **Typvariablen** abstrahieren über Typen

```
data List  $\alpha$  = Empty  
          | Cons  $\alpha$  (List  $\alpha$ )
```

- ▶ α ist eine **Typvariable**
- ▶ α kann mit Char oder Int **instantiert** werden
- ▶ List α ist ein **polymorpher** Datentyp
- ▶ Typvariable α wird bei Anwendung instantiiert
- ▶ **Signatur** der Konstruktoren

```
Empty :: List  $\alpha$   
Cons  ::  $\alpha \rightarrow$  List  $\alpha \rightarrow$  List  $\alpha$ 
```

Polymorphe Datentypen

► **Typkorrekte** Terme:

Empty

Typ

Polymorphe Datentypen

► Typkorrekte Terme:

Empty

Typ

List α

Polymorphe Datentypen

► **Typkorrekte** Terme:

Empty

Cons 57 Empty

Typ

List α

Polymorphe Datentypen

► **Typkorrekte** Terme:

Empty

Cons 57 Empty

Typ

List α

List Int

Polymorphe Datentypen

► **Typkorrekte** Terme:

Empty

Cons 57 Empty

Cons 7 (Cons 8 Empty)

Typ

List α

List Int

Polymorphe Datentypen

► **Typkorrekte** Terme:

Empty

Cons 57 Empty

Cons 7 (Cons 8 Empty)

Typ

List α

List Int

List Int

Polymorphe Datentypen

► **Typkorrekte** Terme:

Empty

Cons 57 Empty

Cons 7 (Cons 8 Empty)

Cons 'p' (Cons 'i' (Cons '3' Empty))

Typ

List α

List Int

List Int

Polymorphe Datentypen

► **Typkorrekte** Terme:

Empty

Cons 57 Empty

Cons 7 (Cons 8 Empty)

Cons 'p' (Cons 'i' (Cons '3' Empty))

Typ

List α

List Int

List Int

List Char

Polymorphe Datentypen

► **Typkorrekte** Terme:

Empty

Cons 57 Empty

Cons 7 (Cons 8 Empty)

Cons 'p' (Cons 'i' (Cons '3' Empty))

Cons True Empty

Typ

List α

List Int

List Int

List Char

Polymorphe Datentypen

► **Typkorrekte** Terme:

Empty

Cons 57 Empty

Cons 7 (Cons 8 Empty)

Cons 'p' (Cons 'i' (Cons '3' Empty))

Cons True Empty

Typ

List α

List Int

List Int

List Char

List Bool

► Nicht **typ-korrekt**:

Cons 'a' (Cons 0 Empty)

Cons True (Cons 'x' Empty)

wegen **Signatur** des Konstruktors:

```
Cons ::  $\alpha \rightarrow$  List  $\alpha \rightarrow$  List  $\alpha$ 
```

Polymorphe Funktionen

- ▶ Verkettung von MyString:

```
cat :: MyString → MyString → MyString
cat Empty t      = t
cat (Cons c s) t = Cons c (cat s t)
```

Polymorphe Funktionen

- ▶ Verkettung von MyString:

```
cat :: MyString → MyString → MyString
cat Empty t      = t
cat (Cons c s) t = Cons c (cat s t)
```

- ▶ Verkettung von IntList:

```
cat :: IntList → IntList → IntList
cat Empty t      = t
cat (Cons c s) t = Cons c (cat s t)
```

Polymorphe Funktionen

- ▶ Verkettung von MyString:

```
cat :: MyString → MyString → MyString
cat Empty t      = t
cat (Cons c s) t = Cons c (cat s t)
```

- ▶ Verkettung von IntList:

```
cat :: IntList → IntList → IntList
cat Empty t      = t
cat (Cons c s) t = Cons c (cat s t)
```

- ▶ Gleiche Definition, unterschiedlicher Typ

↪ Zwei Instanzen einer allgemeineren Definition.

Polymorphe Funktionen

- ▶ Polymorphie auch für **Funktionen**:

```
cat :: List α → List α → List α
cat Empty ys          = ys
cat (Cons x xs) ys = Cons x (cat xs ys)
```

- ▶ Typvariable α wird bei Anwendung instantiiert:

```
cat (Cons 3 Empty) (Cons 5 (Cons 57 Empty))
cat (Cons 'p' (Cons 'i' Empty)) (Cons '3' Empty)
```

aber **nicht**

```
cat (Cons True Empty) (Cons 'a' (Cons 0 Empty))
```

- ▶ Typvariable: vergleichbar mit Funktionsparameter

Polymorphe Datentypen: Bäume

Datentyp:

```
data BTree  $\alpha$  = MtBTree  
          | BNode  $\alpha$  (BTree  $\alpha$ ) (BTree  $\alpha$ )
```

Höhe des Baumes:

```
height :: BTree  $\alpha$   $\rightarrow$  Int  
height MtBTree = 0  
height (BNode j l r) = max (height l) (height r) + 1
```

Traversion — erzeugt Liste aus Baum:

```
inorder :: BTree  $\alpha$   $\rightarrow$  List  $\alpha$   
inorder MtBTree = Empty  
inorder (BNode j l r) =  
  cat (inorder l) (Cons j (inorder r))
```

Tupel

- ▶ Mehr als **eine** Typvariable möglich
- ▶ Beispiel: **Tupel** (kartesisches Produkt, Paare)

```
data Pair  $\alpha$   $b$  = Pair  $\alpha$   $b$ 
```

- ▶ Signatur des Konstruktors:

```
Pair ::  $\alpha \rightarrow \beta \rightarrow$  Pair  $\alpha$   $\beta$ 
```

Tupel

- ▶ Mehr als **eine** Typvariable möglich
- ▶ Beispiel: **Tupel** (kartesisches Produkt, Paare)

```
data Pair  $\alpha$  b = Pair  $\alpha$  b
```

- ▶ Signatur des Konstruktors:

```
Pair ::  $\alpha \rightarrow \beta \rightarrow$  Pair  $\alpha$   $\beta$ 
```

- ▶ Beispielterm Typ
Pair 4 'x'

Tupel

- ▶ Mehr als **eine** Typvariable möglich
- ▶ Beispiel: **Tupel** (kartesisches Produkt, Paare)

```
data Pair  $\alpha$  b = Pair  $\alpha$  b
```

- ▶ Signatur des Konstruktors:

```
Pair ::  $\alpha \rightarrow \beta \rightarrow$  Pair  $\alpha$   $\beta$ 
```

- ▶ Beispielterm Typ
Pair 4 'x' Pair Int Char

Tupel

- ▶ Mehr als **eine** Typvariable möglich
- ▶ Beispiel: **Tupel** (kartesisches Produkt, Paare)

```
data Pair  $\alpha$   $\beta$  = Pair  $\alpha$   $\beta$ 
```

- ▶ Signatur des Konstruktors:

```
Pair ::  $\alpha \rightarrow \beta \rightarrow$  Pair  $\alpha$   $\beta$ 
```

- | ▶ Beispielterm | Typ |
|----------------------------|---------------|
| Pair 4 'x' | Pair Int Char |
| Pair (Cons True Empty) 'a' | |

Tupel

- ▶ Mehr als **eine** Typvariable möglich
- ▶ Beispiel: **Tupel** (kartesisches Produkt, Paare)

```
data Pair  $\alpha$  b = Pair  $\alpha$  b
```

- ▶ Signatur des Konstruktors:

```
Pair ::  $\alpha \rightarrow \beta \rightarrow$  Pair  $\alpha$   $\beta$ 
```

- | ▶ Beispielterm | Typ |
|----------------------------|-----------------------|
| Pair 4 'x' | Pair Int Char |
| Pair (Cons True Empty) 'a' | Pair (List Bool) Char |

Tupel

- ▶ Mehr als **eine** Typvariable möglich
- ▶ Beispiel: **Tupel** (kartesisches Produkt, Paare)

```
data Pair  $\alpha$   $\beta$  = Pair  $\alpha$   $\beta$ 
```

- ▶ Signatur des Konstruktors:

```
Pair ::  $\alpha \rightarrow \beta \rightarrow$  Pair  $\alpha$   $\beta$ 
```

- | ▶ Beispielterm | Typ |
|------------------------------|-----------------------|
| Pair 4 'x' | Pair Int Char |
| Pair (Cons True Empty) 'a' | Pair (List Bool) Char |
| Pair (3+ 4) (Cons 'a' Empty) | |

Tupel

- ▶ Mehr als **eine** Typvariable möglich
- ▶ Beispiel: **Tupel** (kartesisches Produkt, Paare)

```
data Pair  $\alpha$   $\beta$  = Pair  $\alpha$   $\beta$ 
```

- ▶ Signatur des Konstruktors:

```
Pair ::  $\alpha \rightarrow \beta \rightarrow$  Pair  $\alpha$   $\beta$ 
```

- | ▶ Beispielterm | Typ |
|------------------------------|-----------------------|
| Pair 4 'x' | Pair Int Char |
| Pair (Cons True Empty) 'a' | Pair (List Bool) Char |
| Pair (3+ 4) (Cons 'a' Empty) | Pair Int (List Char) |

Tupel

- ▶ Mehr als **eine** Typvariable möglich
- ▶ Beispiel: **Tupel** (kartesisches Produkt, Paare)

```
data Pair  $\alpha$   $\beta$  = Pair  $\alpha$   $\beta$ 
```

- ▶ Signatur des Konstruktors:

```
Pair ::  $\alpha \rightarrow \beta \rightarrow$  Pair  $\alpha$   $\beta$ 
```

- | ▶ Beispielterm | Typ |
|------------------------------|-----------------------|
| Pair 4 'x' | Pair Int Char |
| Pair (Cons True Empty) 'a' | Pair (List Bool) Char |
| Pair (3+ 4) (Cons 'a' Empty) | Pair Int (List Char) |
| Cons (Pair 7 'x') Empty | |

Tupel

- ▶ Mehr als **eine** Typvariable möglich
- ▶ Beispiel: **Tupel** (kartesisches Produkt, Paare)

```
data Pair  $\alpha$   $\beta$  = Pair  $\alpha$   $\beta$ 
```

- ▶ Signatur des Konstruktors:

```
Pair ::  $\alpha \rightarrow \beta \rightarrow$  Pair  $\alpha$   $\beta$ 
```

- | ▶ Beispielterm | Typ |
|------------------------------|-----------------------|
| Pair 4 'x' | Pair Int Char |
| Pair (Cons True Empty) 'a' | Pair (List Bool) Char |
| Pair (3+ 4) (Cons 'a' Empty) | Pair Int (List Char) |
| Cons (Pair 7 'x') Empty | List (Pair Int Char) |

Vordefinierte Datentypen: Tupel und Listen

- ▶ Eingebauter **syntaktischer Zucker**
- ▶ **Tupel** sind das kartesische Produkt

```
data ( $\alpha$ ,  $\beta$ ) = ( $\alpha$ ,  $\beta$ )
```

- ▶ (a, b) = alle Kombinationen von Werten aus a und b
- ▶ Auch n-Tupel: (a,b,c) etc.

- ▶ **Listen**

```
data [ $\alpha$ ] = [] |  $\alpha$  : [ $\alpha$ ]
```

- ▶ Weitere **Abkürzungen**: $[x]$ = $x:[]$, $[x,y]$ = $x:y:[]$ etc.

Übersicht: vordefinierte Funktionen auf Listen I

$(++)$	$::$	$[\alpha] \rightarrow [\alpha] \rightarrow [\alpha]$	— Verketteten
$(!!)$	$::$	$[\alpha] \rightarrow \text{Int} \rightarrow \alpha$	— n -tes Element selektieren
concat	$::$	$[[\alpha]] \rightarrow [\alpha]$	— “flachklopfen”
length	$::$	$[\alpha] \rightarrow \text{Int}$	— Länge
head, last	$::$	$[\alpha] \rightarrow \alpha$	— Erstes/letztes Element
tail, init	$::$	$[\alpha] \rightarrow [\alpha]$	— Hinterer/vorderer Rest
replicate	$::$	$\text{Int} \rightarrow \alpha \rightarrow [\alpha]$	— Erzeuge n Kopien
take	$::$	$\text{Int} \rightarrow [\alpha] \rightarrow [\alpha]$	— Erste n Elemente
drop	$::$	$\text{Int} \rightarrow [\alpha] \rightarrow [\alpha]$	— Rest nach n Elementen
splitAt	$::$	$\text{Int} \rightarrow [\alpha] \rightarrow ([\alpha], [\alpha])$	— Spaltet an Index n
reverse	$::$	$[\alpha] \rightarrow [\alpha]$	— Dreht Liste um
zip	$::$	$[\alpha] \rightarrow [\beta] \rightarrow [(\alpha, \beta)]$	— Erzeugt Liste v. Paaren
unzip	$::$	$[(\alpha, \beta)] \rightarrow ([\alpha], [\beta])$	— Spaltet Liste v. Paaren
and, or	$::$	$[\text{Bool}] \rightarrow \text{Bool}$	— Konjunktion/Disjunktion
sum	$::$	$[\text{Int}] \rightarrow \text{Int}$	— Summe (überladen)
product	$::$	$[\text{Int}] \rightarrow \text{Int}$	— Produkt (überladen)

Zeichenketten: String

- ▶ String sind Listen von Zeichen:

```
type String = [Char]
```

- ▶ Alle vordefinierten Funktionen auf Listen verfügbar.
- ▶ Syntaktischer Zucker zur Eingabe:

```
['y', 'o', 'h', 'o'] == "yoho"
```

- ▶ Beispiel:

```
cnt :: Char → String → Int
cnt c []      = 0
cnt c (x:xs) = if (c == x) then 1 + cnt c xs
               else cnt c xs
```

Variadische Bäume

- ▶ Variable Anzahl Kinderknoten: Liste von Kinderknoten

```
data VTree  $\alpha$  = MtVTree
                | VNode  $\alpha$  [VTree  $\alpha$ ]
```

- ▶ Anzahl Knoten zählen:

```
count :: VTree  $\alpha$   $\rightarrow$  Int
count MtVTree = 0
count (VNode _ ns) = 1 + count_nodes ns
```

```
count_nodes :: [VTree  $\alpha$ ]  $\rightarrow$  Int
count_nodes [] = 0
count_nodes (t:ts) = count t + count_nodes ts
```

Berechnungsmuster für Listen

- ▶ **Primitiv rekursive** Definitionen:
 - ▶ Eine Gleichung für leere Liste
 - ▶ Eine Gleichung für nicht-leere Liste, **rekursiver** Aufruf
- ▶ **Komprehensionsschema:**
 - ▶ Jedes **Element** der Eingabeliste
 - ▶ wird **getestet**
 - ▶ und gegebenenfalls **transformiert**

Listenkomprension

- ▶ Ein einfaches **Beispiel**: Zeichenkette in **Kleinbuchstaben** wandeln

```
toL :: String → String  
toL s = [ toLower c | c ← s ]
```

- ▶ **Buchstaben** herausfiltern:

```
letters :: String → String  
letters s = [ c | c ← s, isAlpha c ]
```

- ▶ **Kombination**: alle Buchstaben kanonisch kleingeschrieben

```
toLL :: String → String  
toLL s = [ toLower c | c ← s, isAlpha c ]
```

Listenkomprension

- ▶ **Allgemeine** Form:

```
[ E c | c ← L, test c ]
```

- ▶ Ergebnis: E c für alle Werte c in L, so dass test c wahr ist
- ▶ Typen: L :: [α], c :: α, test :: α → Bool, E :: α → β, Ergebnis [β]

- ▶ Auch **mehrere** Generatoren und Tests möglich:

```
[ E c1 ... cn | c1 ← L1, test1 c1,  
                c2 ← L2 c1, test2 c1 c2, ... ]
```

- ▶ E vom Typ $\alpha_1 \rightarrow \alpha_2 \dots \rightarrow \beta$

Variadische Bäume II

- ▶ Die Zähl-Funktion vereinfacht:

```
count' :: VTree α → Int
count' MtVTree = 0
count' (VNode _ ts) =
  1 + sum [count' t | t ← ts]
```

- ▶ Die Höhe:

```
height' :: VTree α → Int
height' MtVTree = 0
height' (VNode _ ts) =
  1 + maximum (0: [height' t | t ← ts])
```

Beispiel: Permutation von Listen

```
perms :: [ $\alpha$ ] → [[ $\alpha$ ]]
```

- ▶ Permutation der **leeren** Liste
- ▶ Permutation von $x:xs$
- ▶ x an allen Stellen in alle Permutationen von xs eingefügt.

```
perms [] = [ [] ] — Wichtig!  
perms (x:xs) = [ ps ++ [x] ++ qs  
                | rs ← perms xs,  
                  (ps, qs) ← splits rs ]
```

- ▶ Dabei `splits`: alle möglichen Aufspaltungen

```
splits :: [ $\alpha$ ] → [[ $\alpha$ ], [ $\alpha$ ]]  
splits [] = [ ([], []) ]  
splits (y:ys) = ( [], y:ys ) :  
                [ (y:ps, qs) | (ps, qs) ← splits ys ]
```

Beispiel: Quicksort

- ▶ Zerlege Liste in Elemente kleiner, gleich und größer dem ersten,
- ▶ sortiere Teilstücke,
- ▶ konkateniere Ergebnisse.

Beispiel: Quicksort

- ▶ Zerlege Liste in Elemente kleiner, gleich und größer dem ersten,
- ▶ sortiere Teilstücke,
- ▶ konkateniere Ergebnisse.

```
qsort :: [α] → [α]
```

```
qsort [] = []  
qsort (x:xs) =  
  qsort smaller ++ x:equals ++ qsort larger where  
  smaller = [ y | y ← xs, y < x ]  
  equals  = [ y | y ← xs, y == x ]  
  larger  = [ y | y ← xs, y > x ]
```

Überladung und Polymorphie

- ▶ Fehler: qsort nur für Datentypen mit Vergleichsfunktion
- ▶ **Überladung**: Funktion $f :: a \rightarrow b$ existiert für einige, aber nicht für alle Typen
- ▶ Beispiel:
 - ▶ Gleichheit: $(==) :: a \rightarrow a \rightarrow \text{Bool}$
 - ▶ Vergleich: $(<) :: a \rightarrow a \rightarrow \text{Bool}$
 - ▶ Anzeige: $\text{show} :: a \rightarrow \text{String}$
- ▶ Lösung: **Typklassen**
 - ▶ Typklasse Eq für $(==)$
 - ▶ Typklasse Ord für $(<)$ (und andere Vergleiche)
 - ▶ Typklasse Show für show
- ▶ Auch **Ad-hoc Polymorphie** (im Ggs. zur **parametrischen Polymorphie**)

Typklassen in polymorphen Funktionen

- ▶ qsort, korrekte Signatur:

```
qsort :: Ord α ⇒ [α] → [α]
```

- ▶ Element einer Liste (vordefiniert):

```
elem :: Eq α ⇒ α → [α] → Bool  
elem e [] = False  
elem e (x:xs) = e == x || elem e xs
```

- ▶ Liste ordnen und anzeigen:

```
showsorted :: (Eq α, Show α) ⇒ [α] → String  
showsorted x = show (qsort x)
```

Polymorphie in anderen Programmiersprachen: Java

- ▶ Polymorphie in **Java**: Methode auf alle Subklassen anwendbar
 - ▶ Manuelle **Typkonversion** nötig, fehleranfällig
- ▶ Neu ab Java 1.5: **Generics**
 - ▶ Damit **parametrische Polymorphie** möglich

```
class AbsList<T> {  
    public AbsList(T el, AbsList<T> tl) {  
        this.elem= el;  
        this.next= tl;  
    }  
    public T elem;  
    public AbsList<T> next;  
}
```

Polymorphie in anderen Programmiersprachen: Java

Typkorrekte Konkatenation:

```
void concat(AbsList<T> o)
{
    AbsList<T> cur= this;
    while (cur.next  $\neq$  null) cur= cur.next;
    cur.next= o;
}
```

Nachteil: Benutzung umständlich, weil keine Typherleitung

```
AbsList<Integer> l=
    new AbsList<Integer>(new Integer(1),
        new AbsList<Integer>(new Integer(2), null));
```

Polymorphie in anderen Programmiersprachen: C

- ▶ “Polymorphie” in C: void *

```
struct list {  
    void      *head;  
    struct list *tail;  
}
```

- ▶ Gegeben:

```
int x = 7;  
struct list s = { &x, NULL };
```

- ▶ s.head hat Typ void *:

```
int y;  
y = *(int *)s.head;
```

- ▶ Nicht möglich: head direkt als Skalar (e.g. int)
- ▶ C++: [Templates](#)

Zusammenfassung

- ▶ **Typvariablen** und (parametrische) **Polymorphie**: **Abstraktion** über Typen
- ▶ Vordefinierte Typen: Listen $[a]$ und Tupel (a,b)
- ▶ **Berechnungsmuster** über Listen: **primitive Rekursion**, **Listenkompensation**
- ▶ **Überladung** durch **Typklassen**
- ▶ Nächste Woche: Funktionen höherer Ordnung

Praktische Informatik 3: Einführung in die Funktionale
Programmierung
Vorlesung vom 24.11.2010: Funktionen Höherer Ordnung

Christoph Lüth & Dennis Walter

Universität Bremen

Wintersemester 2010/11

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

Inhalt

- ▶ Funktionen **höherer Ordnung**
- ▶ Funktionen als **gleichberechtigte Objekte**
- ▶ Funktionen als **Argumente**
- ▶ Spezielle Funktionen: **map, filter, fold** und Freunde
- ▶ **foldr vs foldl**

Funktionen als Werte

- ▶ Argumente können auch **Funktionen** sein.
- ▶ Beispiel: Funktion **zweimal** anwenden

```
twice :: ( $\alpha \rightarrow \alpha$ )  $\rightarrow$   $\alpha \rightarrow \alpha$   
twice f x = f (f x)
```

- ▶ Auswertung wie vorher:
twice inc 3

Funktionen als Werte

- ▶ Argumente können auch **Funktionen** sein.
- ▶ Beispiel: Funktion **zweimal** anwenden

```
twice :: ( $\alpha \rightarrow \alpha$ )  $\rightarrow$   $\alpha \rightarrow \alpha$   
twice f x = f (f x)
```

- ▶ Auswertung wie vorher:

```
twice inc 3  $\rightsquigarrow$  5
```

```
twice (twice inc) 3
```

Funktionen als Werte

- ▶ Argumente können auch **Funktionen** sein.
- ▶ Beispiel: Funktion **zweimal** anwenden

```
twice :: ( $\alpha \rightarrow \alpha$ )  $\rightarrow$   $\alpha \rightarrow \alpha$   
twice f x = f (f x)
```

- ▶ Auswertung wie vorher:

```
twice inc 3  $\rightsquigarrow$  5
```

```
twice (twice inc) 3  $\rightsquigarrow$  7
```

- ▶ Beispiel: Funktion ***n*-mal hintereinander** anwenden:

```
iter :: Int  $\rightarrow$  ( $\alpha \rightarrow \alpha$ )  $\rightarrow$   $\alpha \rightarrow \alpha$   
iter 0 f x = x  
iter n f x | n > 0 = f (iter (n-1) f x)  
           | otherwise = x
```

- ▶ Auswertung:

```
iter 3 inc
```

Funktionen als Werte

- ▶ Argumente können auch **Funktionen** sein.
- ▶ Beispiel: Funktion **zweimal** anwenden

```
twice :: ( $\alpha \rightarrow \alpha$ )  $\rightarrow$   $\alpha \rightarrow \alpha$   
twice f x = f (f x)
```

- ▶ Auswertung wie vorher:

```
twice inc 3  $\rightsquigarrow$  5
```

```
twice (twice inc) 3  $\rightsquigarrow$  7
```

- ▶ Beispiel: Funktion ***n*-mal hintereinander** anwenden:

```
iter :: Int  $\rightarrow$  ( $\alpha \rightarrow \alpha$ )  $\rightarrow$   $\alpha \rightarrow \alpha$   
iter 0 f x = x  
iter n f x | n > 0 = f (iter (n-1) f x)  
           | otherwise = x
```

- ▶ Auswertung:

```
iter 3 inc  $\rightsquigarrow$  6
```

Funktionen Höherer Ordnung

Slogan

“Functions are first-class citizens.”

- ▶ Funktionen sind **gleichberechtigt**: Werte wie **alle anderen**
- ▶ **Grundprinzip** der funktionalen Programmierung
- ▶ Reflektion
- ▶ Funktionen als Argumente

Funktionen als Argumente: Funktionskomposition

▶ Funktionskomposition (mathematisch)

$$\begin{aligned}(\circ) &:: (\beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \gamma \\(f \circ g) \ x &= f \ (g \ x)\end{aligned}$$

▶ Vordefiniert

▶ Lies: f nach g

▶ Funktionskomposition **vorwärts**:

$$\begin{aligned}(>.>) &:: (\alpha \rightarrow \beta) \rightarrow (\beta \rightarrow \gamma) \rightarrow \alpha \rightarrow \gamma \\(f >.> g) \ x &= g \ (f \ x)\end{aligned}$$

▶ **Nicht** vordefiniert!

Funktionen als Argumente: Funktionskomposition

- ▶ Vertauschen der **Argumente** (vordefiniert):

```
flip :: (α → β → γ) → β → α → γ
flip f b a = f a b
```

- ▶ Damit Funktionskomposition vorwärts:

```
(>.>) :: (α → β) → (β → γ) → α → γ
(>.>) f g x = flip (o) f g x
```

- ▶ Operatorennotation

η -Kontraktion

- ▶ Alternative Definition der Vorwärtskomposition: **Punktfreie** Notation

$$\begin{aligned} (>.>) &:: (\alpha \rightarrow \beta) \rightarrow (\beta \rightarrow \gamma) \rightarrow \alpha \rightarrow \gamma \\ (>.>) &= \text{flip } (\circ) \end{aligned}$$

- ▶ **Da fehlt doch was?!**

η -Kontraktion

- ▶ Alternative Definition der Vorwärtskomposition: **Punktfreie** Notation

$$\begin{aligned}(>.>) &:: (\alpha \rightarrow \beta) \rightarrow (\beta \rightarrow \gamma) \rightarrow \alpha \rightarrow \gamma \\(>.>) &= \text{flip } (\circ)\end{aligned}$$

- ▶ **Da fehlt doch was?!** Nein:

$$(>.>) = \text{flip } (\circ) \equiv (>.>) f g a = \text{flip } (\circ) f g a$$

- ▶ η -Kontraktion (η -Äquivalenz)

- ▶ Bedingung: $E :: \alpha \rightarrow \beta$, $x :: \alpha$, E darf x nicht enthalten

$$\lambda x \rightarrow E x \equiv E$$

- ▶ Syntaktischer Spezialfall **Funktionsdefinition**:

$$f x = E x \equiv f = E$$

- ▶ Warum? **Extensionale** Gleichheit von Funktionen

Funktionen als Argumente: map

- ▶ Funktion **auf alle Elemente anwenden**: map

- ▶ Signatur:

```
map :: ( $\alpha \rightarrow \beta$ )  $\rightarrow$  [ $\alpha$ ]  $\rightarrow$  [ $\beta$ ]
```

- ▶ Definition

```
map f [] = []  
map f (x:xs) = f x : map f xs
```

- ▶ Beispiel:

```
toL :: String  $\rightarrow$  String  
toL = map toLower
```

Funktionen als Argumente: filter

- ▶ Elemente **filtern**: filter

- ▶ Signatur:

```
filter :: ( $\alpha \rightarrow \text{Bool}$ )  $\rightarrow$  [ $\alpha$ ]  $\rightarrow$  [ $\alpha$ ]
```

- ▶ Definition

```
filter p [] = []  
filter p (x:xs)  
  | p x      = x: filter p xs  
  | otherwise = filter p xs
```

- ▶ Beispiel:

```
letters :: String  $\rightarrow$  String  
letters = filter isAlpha
```

Beispiel: Primzahlen

▶ Sieb des Erathostenes

- ▶ Für jede gefundene Primzahl p alle Vielfachen heraussieben
- ▶ Dazu: **filtern** mit $\lambda n \rightarrow \text{mod } n \ p \neq 0!$
- ▶ Namenlose (anonyme) Funktion

▶ Primzahlen im Intervall $[1.. n]$:

```
sieve :: [Integer] → [Integer]
sieve [] = []
sieve (p:ps) =
  p: sieve (filter (\n → mod n p /= 0) ps)

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

- ▶ NB: Mit 2 anfangen!
- ▶ Listengenerator $[n.. m]$

Partielle Applikation

- ▶ Funktionskonstruktor **rechtsassoziativ**:

$$a \rightarrow b \rightarrow c \equiv a \rightarrow (b \rightarrow c)$$

- ▶ **Inbesondere**: $(a \rightarrow b) \rightarrow c \neq a \rightarrow (b \rightarrow c)$
- ▶ Funktionsanwendung ist **linksassoziativ**:

$$f \ a \ b \equiv (f \ a) \ b$$

- ▶ **Inbesondere**: $f \ (a \ b) \neq (f \ a) \ b$

Partielle Applikation

- ▶ Funktionskonstruktor **rechtsassoziativ**:

$$a \rightarrow b \rightarrow c \equiv a \rightarrow (b \rightarrow c)$$

- ▶ **Inbesondere**: $(a \rightarrow b) \rightarrow c \neq a \rightarrow (b \rightarrow c)$
- ▶ Funktionsanwendung ist **linksassoziativ**:
$$f\ a\ b \equiv (f\ a)\ b$$
- ▶ **Inbesondere**: $f\ (a\ b) \neq (f\ a)\ b$
- ▶ **Partielle** Anwendung von Funktionen:
 - ▶ Für $f :: a \rightarrow b \rightarrow c$, $x :: a$ ist $f\ x :: b \rightarrow c$ (**closure**)
- ▶ Beispiele:
 - ▶ `map toLower :: String → String`
 - ▶ `(3 ==) :: Int → Bool`
 - ▶ `concat ∘ map (replicate 2) :: String → String`

Einfache Rekursion

- ▶ **Einfache Rekursion:** gegeben durch
 - ▶ eine Gleichung für die leere Liste
 - ▶ eine Gleichung für die nicht-leere Liste
- ▶ Beispiel: sum, concat, length, (+), ...
- ▶ Auswertung:

sum [4,7,3] \rightsquigarrow

Einfache Rekursion

- ▶ **Einfache Rekursion:** gegeben durch
 - ▶ eine Gleichung für die leere Liste
 - ▶ eine Gleichung für die nicht-leere Liste
- ▶ Beispiel: `sum`, `concat`, `length`, `(++)`, ...
- ▶ Auswertung:

`sum [4,7,3]` \rightsquigarrow `4 + 7 + 3 + 0`
`concat [A, B, C]` \rightsquigarrow

Einfache Rekursion

- ▶ **Einfache Rekursion:** gegeben durch
 - ▶ eine Gleichung für die leere Liste
 - ▶ eine Gleichung für die nicht-leere Liste
- ▶ Beispiel: sum, concat, length, ($++$), ...
- ▶ Auswertung:

sum [4,7,3] \rightsquigarrow 4 + 7 + 3 + 0

concat [A, B, C] \rightsquigarrow A ++ B ++ C ++ []

length [4, 5, 6] \rightsquigarrow

Einfache Rekursion

- ▶ **Einfache Rekursion:** gegeben durch
 - ▶ eine Gleichung für die leere Liste
 - ▶ eine Gleichung für die nicht-leere Liste
- ▶ Beispiel: `sum`, `concat`, `length`, `(++)`, ...
- ▶ Auswertung:

`sum [4,7,3]` \rightsquigarrow `4 + 7 + 3 + 0`

`concat [A, B, C]` \rightsquigarrow `A ++ B ++ C ++ []`

`length [4, 5, 6]` \rightsquigarrow `1 + 1 + 1 + 0`

Einfache Rekursion

- ▶ **Allgemeines Muster:**

$$\begin{aligned}f [] &= A \\f (x:xs) &= x \otimes f xs\end{aligned}$$

- ▶ **Parameter** der Definition:

- ▶ Startwert (für die leere Liste) $A :: b$
- ▶ Rekursionsfunktion $\otimes :: a \rightarrow b \rightarrow b$

- ▶ **Auswertung:**

$$f [x_1, \dots, x_n] = x_1 \otimes x_2 \otimes \dots \otimes x_n \otimes A$$

- ▶ **Terminiert** immer

- ▶ Entspricht einfacher **Iteration** (while-Schleife)

Einfach Rekursion durch foldr

- ▶ **Einfache** Rekursion

- ▶ **Basisfall**: leere Liste

- ▶ **Rekursionsfall**: Kombination aus Listenkopf und Rekursionswert

- ▶ **Signatur**

```
foldr :: ( $\alpha \rightarrow \beta \rightarrow \beta$ )  $\rightarrow \beta \rightarrow [\alpha] \rightarrow \beta$ 
```

- ▶ **Definition**

```
foldr f e [] = e  
foldr f e (x:xs) = f x (foldr f e xs)
```

Beispiele: foldr

- ▶ Summieren von Listenelementen.

```
sum :: [Int] → Int
sum xs = foldr (+) 0 xs
```

- ▶ Flachklopfen von Listen.

```
concat :: [[a]] → [a]
concat xs = foldr (++) [] xs
```

- ▶ Länge einer Liste

```
length :: [a] → Int
length xs = foldr (λx n → n + 1) 0 xs
```

Noch ein Beispiel: rev

- ▶ Listen **umdrehen**:

```
rev :: [a] → [a]
rev []      = []
rev (x:xs) = rev xs ++ [x]
```

- ▶ Mit fold:

```
rev xs = foldr snoc [] xs

snoc :: a → [a] → [a]
snoc x xs = xs ++ [x]
```

- ▶ Unbefriedigend: doppelte Rekursion

Einfache Rekursion durch foldl

- ▶ foldr faltet von rechts:

$$\text{foldr } \otimes [x_1, \dots, x_n] A = x_1 \otimes (x_2 \otimes (\dots (x_n \otimes A)))$$

- ▶ Warum nicht andersherum?

$$\text{foldl } \otimes [x_1, \dots, x_n] A = (((A \otimes x_1) \otimes x_2) \dots) \otimes x_n$$

- ▶ Definition von foldl:

```
foldl :: (α → β → α) → α → [β] → α
foldl f a [] = a
foldl f a (x:xs) = foldl f (f a x) xs
```

Beispiel: rev revisited

- ▶ Listenumkehr ist falten **von links**:

```
rev ' xs = foldl (flip (:)) [] xs
```

- ▶ Nur noch **eine** Rekursion

foldr vs. foldl

- ▶ $f = \text{foldr } \otimes A$ entspricht

$$\begin{aligned}f [] &= A \\f (x:xs) &= x \otimes f xs\end{aligned}$$

- ▶ Kann nicht-strikt in xs sein, z.B. `and`, `or`

- ▶ $f = \text{foldl } \otimes A$ entspricht

$$\begin{aligned}f xs &= g A xs \\g a [] &= a \\g a (x:xs) &= g (a \otimes x) xs\end{aligned}$$

- ▶ Endrekursiv (effizient), aber strikt in xs

foldl = foldr

Definition (Monoid)

(\otimes, A) ist ein **Monoid** wenn

$$A \otimes x = x \quad (\text{Neutrales Element links})$$

$$x \otimes A = x \quad (\text{Neutrales Element rechts})$$

$$(x \otimes y) \otimes z = x \otimes (y \otimes z) \quad (\text{Assoziativität})$$

Theorem

Wenn (\otimes, A) **Monoid**, dann für alle A, xs

$$\text{foldl } \otimes A xs = \text{foldr } \otimes A xs$$

- ▶ Beispiele: length, concat, sum
- ▶ Gegenbeispiel: rev

Funktionen Höherer Ordnung: Java

- ▶ **Java**: keine direkte Syntax für Funktionen höherer Ordnung
- ▶ Folgendes ist **nicht** möglich:

```
interface Collection {  
    Object fold(Object f(Object a, Collection c),  
                Object a) }
```

- ▶ Aber folgendes:

```
interface Foldable {  
    Object f (Object a); }  
  
interface Collection {  
    Object fold(Foldable f, Object a); }
```

- ▶ Vergleiche Iterator aus Collections Framework (Java SE 6):

```
public interface Iterator<E>  
    boolean hasNext();  
    E next(); }
```

Funktionen Höherer Ordnung: C

- ▶ Implizit vorhanden: Funktionen = Zeiger auf Funktionen

```
typedef struct list_t {  
    void          *elem;  
    struct list_t *next;  
} *list;
```

```
list filter(int f(void *x), list l);
```

- ▶ Keine direkte Syntax (e.g. namenlose Funktionen)
- ▶ Typsystem zu schwach (keine Polymorphie)
- ▶ Benutzung: signal (C-Standard 7.14.1)

```
#include <signal.h>
```

```
void (*signal(int sig, void (*func)(int)))(int);
```

Funktionen Höherer Ordnung: C

Implementierung von filter:

```
list filter(int f(void *x), list l)
{ if (l == NULL) {
    return NULL;
  }
  else {
    list r;
    r = filter(f, l->next);
    if (f(l->elem)) {
      l->next = r;
      return l;
    }
    else {
      free(l);
      return r;
    }
  }
}
```

Übersicht: vordefinierte Funktionen auf Listen II

```
map      :: ( $\alpha \rightarrow \beta$ )  $\rightarrow$   $[\alpha] \rightarrow [\beta]$       — Auf alle anwenden
filter  :: ( $\alpha \rightarrow \text{Bool}$ )  $\rightarrow$   $[\alpha] \rightarrow [\alpha]$  — Elemente filtern
foldr   :: ( $\alpha \rightarrow \beta \rightarrow \beta$ )  $\rightarrow$   $\beta \rightarrow [\alpha] \rightarrow \beta$  — Falten v. rechts
foldl   :: ( $\beta \rightarrow \alpha \rightarrow \beta$ )  $\rightarrow$   $\beta \rightarrow [\alpha] \rightarrow \beta$  — Falten v. links
takeWhile :: ( $\alpha \rightarrow \text{Bool}$ )  $\rightarrow$   $[\alpha] \rightarrow [\alpha]$ 
dropWhile :: ( $\alpha \rightarrow \text{Bool}$ )  $\rightarrow$   $[\alpha] \rightarrow [\alpha]$ 
    — takeWhile ist längster Prefix so dass p gilt, dropWhile der Rest
any     :: ( $\alpha \rightarrow \text{Bool}$ )  $\rightarrow$   $[\alpha] \rightarrow \text{Bool}$  — p gilt mind. einmal
all     :: ( $\alpha \rightarrow \text{Bool}$ )  $\rightarrow$   $[\alpha] \rightarrow \text{Bool}$  — p gilt für alle
elem    :: ( $\text{Eq } \alpha$ )  $\Rightarrow$   $\alpha \rightarrow [\alpha] \rightarrow \text{Bool}$  — Ist enthalten?
zipWith :: ( $\alpha \rightarrow \beta \rightarrow \gamma$ )  $\rightarrow$   $[\alpha] \rightarrow [\beta] \rightarrow [\gamma]$ 
    — verallgemeinertes zip
```

Allgemeine Rekursion

- ▶ Einfache Rekursion ist **Spezialfall** der **allgemeinen** Rekursion
- ▶ **Allgemeine** Rekursion:
 - ▶ Rekursion über **mehrere Argumente**
 - ▶ Rekursion über **andere Datenstruktur**
 - ▶ **Andere Zerlegung** als Kopf und Rest

Beispiele für allgemeine Rekursion: Sortieren

▶ Quicksort:

- ▶ zerlege Liste in Elemente **kleiner**, **gleich** und **größer** dem ersten,
- ▶ **sortiere** Teilstücke, **konkatenierte** Ergebnisse

▶ Mergesort:

- ▶ **teile** Liste in der **Hälfte**,
- ▶ **sortiere** Teilstücke, füge **ordnungserhaltend** zusammen.

Beispiel für allgemeine Rekursion: Mergesort

► Hauptfunktion:

```
msortBy :: Ord a => [a] -> [a]
msortBy xs
  | length xs <= 1 = xs
  | otherwise = merge (msortBy f) (msortBy b) where
    (f, b) = splitAt ((length xs) `div` 2) xs
```

► `splitAt :: Int -> [a] -> ([a], [a])` spaltet Liste auf

► Hilfsfunktion: ordnungserhaltendes Zusammenfügen

```
merge :: Ord a => [a] -> [a] -> [a]
merge [] x = x
merge y [] = y
merge (x:xs) (y:ys)
  | x <= y = x:(merge xs (y:ys))
  | otherwise = y:(merge (x:xs) ys)
```

Zusammenfassung

- ▶ Funktionen **höherer Ordnung**
 - ▶ Funktionen als **gleichberechtigte Objekte** und **Argumente**
 - ▶ Partielle Applikation, η -Kontraktion, namenlose Funktionen
 - ▶ Spezielle Funktionen höherer Ordnung: **map**, **filter**, **fold** und **Freunde**
- ▶ Formen der **Rekursion**:
 - ▶ **Einfache** und **allgemeine** Rekursion
 - ▶ **Einfache** Rekursion entspricht **foldr**

Praktische Informatik 3: Einführung in die Funktionale
Programmierung
Vorlesung vom 08.12.2010: Abstrakte Datentypen

Christoph Lüth & Dennis Walter

Universität Bremen

Wintersemester 2010/11

Fahrplan

- ▶ Teil I: Funktionale Programmierung im Kleinen
- ▶ Teil II: Funktionale Programmierung im Großen
 - ▶ Abstrakte Datentypen
 - ▶ Signaturen und Eigenschaften
 - ▶ Aktionen und Zustände
- ▶ Teil III: Funktionale Programmierung im richtigen Leben

Inhalt

- ▶ Abstrakte Datentypen
 - ▶ Allgemeine Einführung
 - ▶ Realisierung in Haskell
 - ▶ Beispiele

Einfache Bäume

- ▶ Schon bekannt: Bäume

```
data Tree  $\alpha$  = Null
           | Node (Tree  $\alpha$ )  $\alpha$  (Tree  $\alpha$ )
```

- ▶ Dazu Test auf Enthaltensein:

```
member' :: Eq  $\alpha$   $\Rightarrow$   $\alpha$   $\rightarrow$  Tree  $\alpha$   $\rightarrow$  Bool
member' _ Null = False
member' b (Node l a r) =
  a == b || member' b l || member' b r
```

- ▶ Problem: Suche **aufwändig** (Backtracking)
- ▶ Besser: Baum **geordnet**
 - ▶ Noch besser: Baum **balanciert**

Geordnete Bäume

- ▶ Voraussetzung:

- ▶ Ordnung auf a ($\text{Ord } a$)

- ▶ Es soll für alle Bäume gelten:

$$\forall x \ t. \ t = \text{Node } l \ a \ r \longrightarrow \begin{aligned} &(\text{member } x \ l \longrightarrow x < a) \wedge \\ &(\text{member } x \ r \longrightarrow a < x) \end{aligned}$$

- ▶ Beispiel für eine Datentyp-**Invariante**

- ▶ Test auf Enthaltensein vereinfacht:

```
member :: Ord a => a -> Tree a -> Bool
member _ Null = False
member b (Node l a r)
  | b < a = member b l
  | a == b = True
  | b > a = member b r
```

Geordnete Bäume

- ▶ Ordnungserhaltendes Einfügen

```
insert :: Ord a => a -> Tree a -> Tree a
insert a Null = Node Null a Null
insert b (Node l a r)
  | b < a = Node (insert b l) a r
  | b == a = Node l a r
  | b > a = Node l a (insert b r)
```

- ▶ **Problem:** Erzeugung ungeordneter Bäume möglich.
- ▶ **Lösung:** *Verstecken* der Konstrukturen.
- ▶ **Warum?** E.g. Implementation von geordneten Mengen

Geordnete Bäume als abstrakter Datentyp

- ▶ Es gibt einen **Typ** `Tree a`
- ▶ Es gibt **Operationen**
 - ▶ `empty :: Ord α ⇒ Tree α`
 - ▶ Nicht `Null` :: `Tree a`, sonst Konstruktor sichtbar
 - ▶ `insert :: Ord α ⇒ α → Tree α → Tree α`
 - ▶ `member :: Ord α ⇒ α → Tree α → Bool`
 - ▶ ... und **keine** weiteren!
- ▶ Beispiel für einen **abstrakten Datentypen**
- ▶ Datentyp-**Invarianten** können außerhalb des definierenden Moduls nicht verletzt werden

Abstrakte Datentypen

Definition (ADT)

Ein **abstrakter Datentyp** (ADT) besteht aus einem (oder mehreren) **Typen** und **Operationen** auf diesem.

- ▶ Werte des Typen können nur über die bereitgestellten Operationen erzeugt werden.
- ▶ Eigenschaften von Werten des Typen (insb. ihre innere Struktur) können nur über die bereitgestellten Operationen beobachtet werden.

Zur **Implementation** von ADTs in einer Programmiersprache:
Möglichkeit der **Kapselung** durch

- ▶ Module
- ▶ Objekte

ADTs in Haskell: Module

- ▶ Einschränkung der Sichtbarkeit durch **Verkapselung**
- ▶ **Modul**: Kleinste verkapselbare **Einheit**
- ▶ Ein **Modul** umfaßt:
 - ▶ **Definitionen** von Typen, Funktionen, Klassen
 - ▶ **Deklaration** der nach außen **sichtbaren** Definitionen
- ▶ **Syntax**:
`module Name (sichtbare Bezeichner) where Rumpf`
 - ▶ *sichtbare Bezeichner* können leer sein
- ▶ Gleichzeitig: Modul $\hat{=}$ Übersetzungseinheit (getrennte Übersetzung)

Beispiel: Exportliste für Bäume

- ▶ Export als **abstrakter Datentyp**
module OrdTree (Tree, insert, member, empty) **where**
 - ▶ Typ Tree extern sichtbar
 - ▶ Konstruktoren versteckt
- ▶ Export als **konkreter Datentyp**
module OrdTree (Tree(..), insert , member, empty) **where**
 - ▶ Konstruktoren von Tree sind extern sichtbar
 - ▶ Pattern Matching ist möglich
 - ▶ Erzeugung auch von ungeordneten Bäumen möglich

Benutzung von ADTs

- ▶ **Operationen** und **Typen** müssen bekannt gemacht werden (Import)
- ▶ Möglichkeiten des Imports:
 - ▶ **Alles** importieren
 - ▶ Nur bestimmte Operationen und Typen importieren
 - ▶ Bestimmte Typen und Operationen nicht importieren

Importe in Haskell

- ▶ **Schlüsselwort:** `import Name [hiding] (Bezeichner)`
- ▶ *Bezeichner* geben an, **was** importiert werden soll:
 - ▶ Ohne Bezeichner wird **alles** importiert
 - ▶ Mit `hiding` werden Bezeichner **nicht** importiert
 - ▶ Alle Importe stehen immer am **Anfang** des Moduls
- ▶ Qualifizierter Import zur Vermeidung von Namenskollisionen
 - ▶ `import qualified Name as OtherName`
 - ▶ Z. B. **import qualified** `Data.Map as M`

Beispiel: Importe von Bäumen

Import(e)	Bekannte Bezeichner
<code>import OrdTree</code>	<code>Tree, insert, member,</code> <code>empty</code>
<code>import OrdTree(Tree, empty)</code>	<code>Tree, empty</code>
<code>import OrdTree(insert)</code>	<code>insert</code>
<code>import OrdTree hiding (member)</code>	<code>Tree, empty, insert</code>
<code>import OrdTree(empty)</code>	<code>empty,</code>
<code>import OrdTree hiding (empty)</code>	<code>Tree, insert, member</code>

Baumtraversion als Funktion höherer Ordnung

- ▶ Nützlich: Traversal als generisches fold
- ▶ Dadurch Iteration über den Baum möglich, ohne Struktur offenzulegen

```
foldT :: ( $\alpha \rightarrow \beta \rightarrow \beta \rightarrow \beta$ )  $\rightarrow \beta \rightarrow$  Tree  $\alpha \rightarrow \beta$   
foldT f e Null = e  
foldT f e (Node l a r) =  
  f a (foldT f e l) (foldT f e r)
```

- ▶ Damit externe Definition von Aufzählung möglich:

```
enum :: Ord  $\alpha \Rightarrow$  Tree  $\alpha \rightarrow [\alpha]$   
enum = foldT ( $\lambda x l1 l2 \rightarrow l1 ++ x : l2$ ) []
```

Schnittstelle vs. Implementation

- ▶ Gleiche **Schnittstelle** kann unterschiedliche **Implementationen** haben

- ▶ Beispiel: (endliche) Mengen

Endliche Mengen: Typsignaturen (1)

- ▶ Abstrakter Datentyp für endliche Mengen (polymorph über Elementtyp)

```
type Set a
```

- ▶ Leere Menge:

```
empty :: Set a
```

- ▶ Einfügen in eine Menge:

```
insert :: Ord a => a -> Set a -> Set a
```

- ▶ Test auf Enthaltensein

```
member :: Ord a => a -> Set a -> Bool
```

Endliche Mengen: Typsignaturen (2)

- ▶ Test auf leere Menge

```
null :: Set a → Bool
```

- ▶ Vereinigung

```
union :: Ord a ⇒ Set a → Set a → Set a
```

- ▶ Schnittmenge

```
intersection :: Ord a ⇒ Set a → Set a → Set a
```

- ▶ Umwandlung zu Listen

```
toList :: Set a → [a]  
fromList :: Ord a ⇒ [a] → Set a
```

- ▶ Mappen und Falten:

```
map :: (Ord a, Ord b) ⇒ (a → b) → Set a → Set b  
fold :: (a → b → b) → b → Set a → b
```

Endliche Mengen: Eigenschaften

- ▶ Die leere Menge

```
empty null
not (empty (insert x s))
```

- ▶ Extensionalität

$$s1 == s2 \Leftrightarrow (\forall x. \text{member } x \text{ } s1 \Leftrightarrow \text{member } x \text{ } s2)$$

- ▶ Einfügen und Enthaltensein

```
insert x (insert y s) == insert y (insert x s)
member x (insert x s)
```

- ▶ Schnittmenge

```
member x (intersection s1 s2) ==
  member x s1 && member x s2
```

- ▶ Vereinigung

```
member x (union s1 s2) ==
  member x s1 || member x s2
```

Endliche Mengen: Implementierung

- ▶ Für den Anwender von `Data.Set` **irrelevant!**
- ▶ Wichtig aus Implementierungssicht: **Effizienz**
- ▶ Verschiedene Möglichkeiten der Repräsentation
 - ▶ Sortierte Listen: **type** `Set a = [a]`
 - ▶ Funktionen: **type** `Set a = a → Bool`
 - ▶ In der Tat verwendet: Balancierte Bäume
data `Set a = Tip | Bin Int a (Set a) (Set a)`
(`Int` gibt die Größe des Baumes an.)

Endliche Abbildungen

- ▶ Eine Sichtweise: Ersatz für Hashtables in imperativen Sprachen.
Sehr nützlich!
- ▶ Abstrakter Datentyp für endliche Abbildungen (polymorph über Schlüssel- und Werttyp)

```
type Map a b
```

- ▶ Leere Abbildung:

```
empty :: Map a b
```

- ▶ Hinzufügen eines Schlüssel/Wert-Paars

```
insert :: Ord a => a -> b -> Map a b -> Map a b
```

- ▶ Test auf Enthaltensein

```
member :: Ord a => a -> Map a b -> Bool
```

Weitere Funktionen

- ▶ Test auf leere Abbildung

```
null :: Map a b → Bool
```

- ▶ Nachschlagen eines Werts

```
lookup :: Ord a ⇒ a → Map a b → Maybe b  
(!) :: Ord a ⇒ Map a b → a → b
```

- ▶ Löschen

```
delete :: Ord a ⇒ a → Map a b → Map a b
```

- ▶ Einfügen und Duplikatkonflikte lösen

```
insertWith :: Ord a ⇒  
(b → b → b) → a → b → Map a b → Map a b
```

- ▶ Mappen und Falten:

```
map :: (b → c) → Map a b → Map a c  
fold :: (b → c → c) → c → Map a b → c
```

Endl. Abbildungen: Anwendungsbeispiele

- ▶ Anzahl von Artikeln im Warenhaus

```
type Warehouse = Data.Map String Int

nLeft :: Warehouse → String → Int → Bool
nLeft w art n =
  case s 'Data.Map.lookup' w of
    Nothing → False
    Just m → m ≥ n

addArticle :: String → Int → Warehouse →
            Warehouse
addArticle art n w =
  Data.Map.insertWith (+) art w
```

Weiterer Datentyp: Prioritätswarteschlangen

- ▶ Signatur von Prioritätswarteschlangen ähnlich Stacks und FIFO Queues:

```
type PriorityQueue k a
```

k steht für Priorität, a ist eigentlicher Wert

- ▶ Operationen:
 - ▶ `empty :: PriorityQueue k a`
 - ▶ `null :: Ord k ⇒ PriorityQueue k a → Bool`
 - ▶ `insert :: Ord k ⇒ k → a → PriorityQueue k a → PriorityQueue k a`
 - ▶ `minKeyValue :: Ord k ⇒ PriorityQueue k a → (k, a)`
 - ▶ `deleteMin :: Ord k ⇒ PriorityQueue k a → PriorityQueue k a`

Implementierung mittels Heaps

- ▶ Ein **Heap** ist eine baumartige Datenstruktur mit der **Heap-Eigenschaft**:

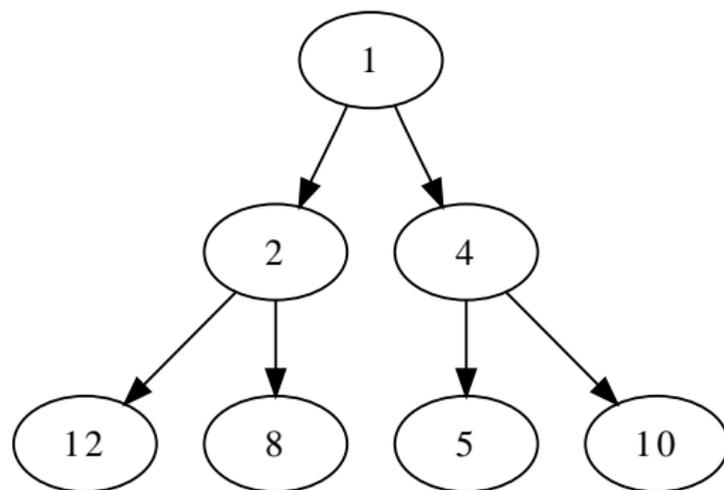
- ▶ **data** Heap k a = Nil | Branch k a (Heap k a) (Heap k a)

```
heapProp :: Ord k => Heap k a → Bool
heapProp Nil = True
heapProp (Branch k a l r) =
  k ≤ min (minH k l) (minH k r)
  && heapProp l && heapProp r
  where minH k Nil = k
        minH _ (Branch k a l r) =
          min k (min (minH k l) (minH k r))
```

- ▶ Wurzelement jedes Teilbaums ist minimales Element des Teilbaums

Beispiel: Heap-Eigenschaft

- ▶ Ein vollständiger binärer Baum mit Heap-Eigenschaft
- ▶ **Kein** geordneter Baum



Binäre Heaps

- **Vollständigkeit** zusätzlich zur Heap-Eigenschaft

```
depth :: Ord k => Heap k a -> Int
```

```
depth Nil = 0
```

```
depth (Branch _ _ l r) =  
  1 + max (depth l) (depth r)
```

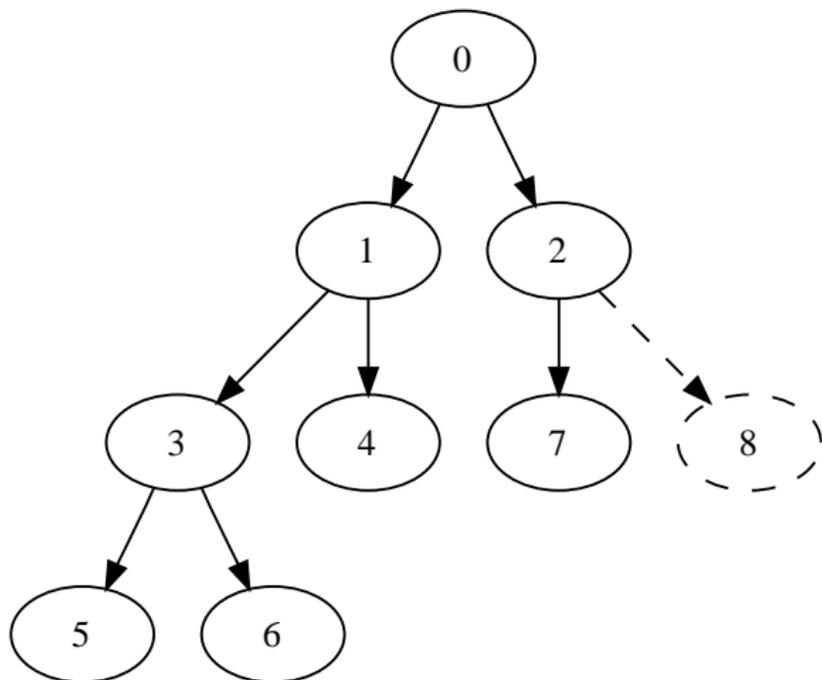
```
completeAt n Nil = False
```

```
completeAt n (Branch _ _ l r) =  
  if n > 0 then completeAt (n - 1) l &&  
                completeAt (n - 1) r  
  else not (null l) && not (null r)
```

```
complete t = d < 3 || completeAt (d - 3) t  
  where d = depth t
```

Beispiel: Vollständigkeit

- ▶ Mit Knoten 8: vollständig
- ▶ Ohne 8: höhenbalanciert, aber nicht vollständig



Operationen

► Exportierte Operationen:

```
singleton :: Ord k => k -> a -> Heap k a
singleton k a = Branch k a Nil Nil
```

```
empty :: Heap k a
empty = Nil
```

```
insert :: Ord k => k -> a -> Heap k a -> Heap k a
insert k a Nil = singleton k a
insert k a (Branch k' a' l r)
  | k < k' = Branch k a (insert k' a' r) l
  | otherwise = Branch k' a' (insert k a r) l
```

```
minKeyValue :: Ord k => Heap k a -> (k, a)
minKeyValue (Branch k a _ _) = (k, a)
```

Entfernen des minimalen Elements

- ▶ Zum Entfernen: “Hochziehen” des jeweils kleineren Kindelements

```
deleteMin :: Ord k => Heap k a -> Heap k a
deleteMin t =
  case t of
    Branch _ _ Nil r -> r
    Branch _ _ l Nil -> l
    Branch k a (l@(Branch lk la _ _))
              (r@(Branch rk ra _ _)) ->
      if lk < rk then Branch lk la (deleteMin l) r
      else Branch rk ra l (deleteMin r)
```

- ▶ Beispiele siehe ../uebung/ueb06/trees.pdf

Effizienz

- ▶ Laufzeitverhalten: $\mathcal{O}(\log(n))$ für `insert` und `deleteMin`, $\mathcal{O}(1)$ für `minKeyValue`, `singleton` und `empty`.
- ▶ **Pairing Heaps** als Alternative zu Binären Heaps
 - ▶ Worst-case Laufzeit für `deleteMin` in $\mathcal{O}(n)$
 - ▶ Aber: amortisierte Kosten in $\mathcal{O}(\log(n))$ und in der Praxis erstaunlich schnell
- ▶ Bsp.: 10^6 zufällig priorisierte Elemente einfügen und entfernen
 - ▶ Haskell: Laufzeit $\approx 7s$ (im Vergleich: $\approx 12.7s$ für Binärheap)
 - ▶ OCaml: $\approx 3.3s$
 - ▶ Java (Binärer Heap als Array): $\approx 3.6s$
- ▶ Tests auf 2GHz Intel Dual Core

Zusammenfassung

- ▶ **Abstrakte Datentypen** (ADTs):
 - ▶ Besteht aus **Typen** und **Operationen** darauf
- ▶ Realisierung in Haskell durch **Module**
- ▶ Beispieldatentypen: endliche Mengen und Abbildungen, Prioritätswarteschlangen
- ▶ Nächste Vorlesung: ADTs durch **Eigenschaften** spezifizieren

Vorlesung nächste Woche (15.12.2010) **entfällt** wegen Tag der Lehre!

Der Übungsbetrieb **findet normal statt**.

Praktische Informatik 3: Einführung in die Funktionale
Programmierung
Vorlesung vom 05.01.2011: Signaturen und Eigenschaften

Christoph Lüth & Dennis Walter

Universität Bremen

Wintersemester 2010/11

Fahrplan

- ▶ Teil I: Funktionale Programmierung im Kleinen
- ▶ Teil II: Funktionale Programmierung im Großen
 - ▶ Abstrakte Datentypen
 - ▶ Signaturen und Eigenschaften
 - ▶ Aktionen und Zustände
- ▶ Teil III: Funktionale Programmierung im richtigen Leben

Abstrakte Datentypen

- ▶ Letzte Vorlesung: **Abstrakte Datentypen**
- ▶ **Typ** plus **Operationen**
 - ▶ In Haskell: **Module**
- ▶ Heute: **Signaturen** und **Eigenschaften**

Signaturen

Definition (Signatur)

Die **Signatur** eines abstrakten Datentyps besteht aus den Typen, und der Signatur der darüber definierten Funktionen.

- ▶ Keine direkte Repräsentation in Haskell
- ▶ Signatur: `Typ` eines Moduls

Zur Erinnerung: Endliche Abbildungen

- ▶ Endliche Abbildung (FiniteMap)
- ▶ Typen: die Abbildung S , Adressen a , Werte b
- ▶ Operationen (Auszug)
 - ▶ leere Abbildung: S

Zur Erinnerung: Endliche Abbildungen

- ▶ Endliche Abbildung (FiniteMap)
- ▶ Typen: die Abbildung S , Adressen a , Werte b
- ▶ Operationen (Auszug)
 - ▶ leere Abbildung: S
 - ▶ Abbildung an einer Stelle **schreiben**: $S \rightarrow a \rightarrow b \rightarrow S$

Zur Erinnerung: Endliche Abbildungen

- ▶ Endliche Abbildung (FiniteMap)
- ▶ Typen: die Abbildung S , Adressen a , Werte b
- ▶ Operationen (Auszug)
 - ▶ leere Abbildung: S
 - ▶ Abbildung an einer Stelle **schreiben**: $S \rightarrow a \rightarrow b \rightarrow S$
 - ▶ Abbildung an einer Stelle **lesen**: $S \rightarrow a \rightarrow b$ (partiell)

Zur Erinnerung: Endliche Abbildungen

- ▶ Endliche Abbildung (FiniteMap)
- ▶ Typen: die Abbildung S , Adressen a , Werte b
- ▶ Operationen (Auszug)
 - ▶ leere Abbildung: S
 - ▶ Abbildung an einer Stelle **schreiben**: $S \rightarrow a \rightarrow b \rightarrow S$
 - ▶ Abbildung an einer Stelle **lesen**: $S \rightarrow a \rightarrow b$ (partiell)

Endliche Abbildung: Signatur

- ▶ Adressen und Werte sind Parameter

```
type Map  $\alpha$   $\beta$ 
```

- ▶ Leere Abbildung:

```
empty :: Map  $\alpha$   $\beta$ 
```

- ▶ An eine Stelle einen Wert schreiben:

```
insert :: Map  $\alpha$   $\beta$   $\rightarrow$   $\alpha$   $\rightarrow$   $\beta$   $\rightarrow$  Map  $\alpha$   $\beta$ 
```

- ▶ An einer Stelle einen Wert lesen:

```
lookup :: Map  $\alpha$   $\beta$   $\rightarrow$   $\alpha$   $\rightarrow$  Maybe  $\beta$ 
```

Signatur und Eigenschaften

- ▶ Signatur genug, um ADT **typkorrekt** zu benutzen
 - ▶ Insbesondere **Anwendbarkeit** und **Reihenfolge**
- ▶ Signatur nicht genug, um **Bedeutung** (Semantik) zu beschreiben:
 - ▶ Was wird **gelesen**?
 - ▶ Wie **verhält** sich die Abbildung?

Beschreibung von Eigenschaften

Definition (Axiome)

Axiome sind Prädikate über den Operationen der Signatur

- ▶ Elementare Prädikate P :
 - ▶ Gleichheit $s == t$
 - ▶ Ordnung $s < t$
 - ▶ Selbstdefinierte Prädikate
- ▶ Zusammengesetzte Prädikate
 - ▶ Negation $\text{not } p$
 - ▶ Konjunktion $p \ \&\& \ q$
 - ▶ Disjunktion $p \ || \ q$
 - ▶ **Implikation** $p \implies q$

Beobachtbare und Abstrakte Typen

- ▶ **Beobachtbare** Typen: interne Struktur bekannt
 - ▶ Vordefinierte Typen (**Zahlen**, **Zeichen**), algebraische Datentypen (**Listen**)
 - ▶ Viele Eigenschaften und Prädikate bekannt
- ▶ **Abstrakte** Typen: interne Struktur unbekannt
 - ▶ Wenig Eigenschaft bekannt, Gleichheit nur wenn definiert
- ▶ Beispiel Map:
 - ▶ **beobachtbar**: Adressen und Werte
 - ▶ **abstrakt**: Speicher

Axiome für Map

- ▶ Lesen aus leerer Abbildung undefiniert:

Axiome für Map

- ▶ Lesen aus leerer Abbildung undefiniert:

```
lookup empty a == Nothing
```

- ▶ Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:

Axiome für Map

- ▶ Lesen aus leerer Abbildung undefiniert:

```
lookup empty a == Nothing
```

- ▶ Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:

```
lookup (insert m a b) a == Just b
```

- ▶ Lesen an anderer Stelle liefert alten Wert:

Axiome für Map

- ▶ Lesen aus leerer Abbildung undefiniert:

```
lookup empty a == Nothing
```

- ▶ Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:

```
lookup (insert m a b) a == Just b
```

- ▶ Lesen an anderer Stelle liefert alten Wert:

```
a1 /= a2 ==> lookup (insert m a1 b) a2 ==  
              lookup m a2
```

- ▶ Schreiben an dieselbe Stelle überschreibt alten Wert:

Axiome für Map

- ▶ Lesen aus leerer Abbildung undefiniert:

```
lookup empty a == Nothing
```

- ▶ Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:

```
lookup (insert m a b) a == Just b
```

- ▶ Lesen an anderer Stelle liefert alten Wert:

```
a1 /= a2 ==> lookup (insert m a1 b) a2 ==  
              lookup m a2
```

- ▶ Schreiben an dieselbe Stelle überschreibt alten Wert:

```
insert (m a b1) a b2 == insert m a b2
```

- ▶ Schreiben über verschiedene Stellen kommutiert:

Axiome für Map

- ▶ Lesen aus leerer Abbildung undefiniert:

```
lookup empty a == Nothing
```

- ▶ Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:

```
lookup (insert m a b) a == Just b
```

- ▶ Lesen an anderer Stelle liefert alten Wert:

```
a1 /= a2 ==> lookup (insert m a1 b) a2 ==  
              lookup m a2
```

- ▶ Schreiben an dieselbe Stelle überschreibt alten Wert:

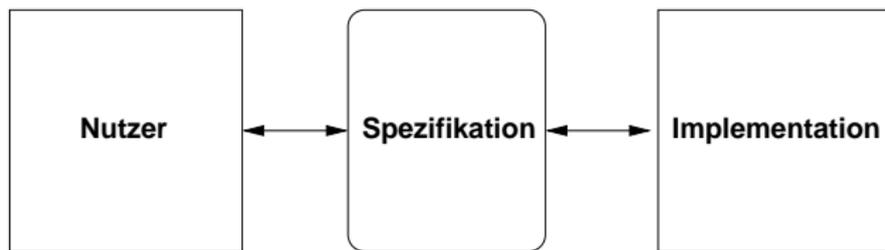
```
insert (m a b1) a b2 == insert m a b2
```

- ▶ Schreiben über verschiedene Stellen kommutiert:

```
a1 /= a2 ==> insert (insert m a1 b1) a2 b2 ==  
              insert (insert m a2 b2) a1 b1
```

Axiome als Interface

- ▶ Axiome müssen **gelten**
 - ▶ für **alle** Werte der freien Variablen zu True auswerten
- ▶ Axiome **spezifizieren**:
 - ▶ nach außen das **Verhalten**
 - ▶ nach innen die **Implementation**
- ▶ **Signatur + Axiome = Spezifikation**



- ▶ Implementation kann **getestet** werden
- ▶ Axiome können (sollten?) **bewiesen** werden

Signatur und Semantik

Stacks

Typ: $\text{St } \alpha$

Initialwert:

```
empty :: St  $\alpha$ 
```

Wert ein/auslesen:

```
push  ::  $\alpha \rightarrow \text{St } \alpha \rightarrow \text{St } \alpha$ 
```

```
top   :: St  $\alpha \rightarrow \alpha$ 
```

```
pop   :: St  $\alpha \rightarrow \text{St } \alpha$ 
```

Test auf Leer:

```
isEmpty :: St  $\alpha \rightarrow \text{Bool}$ 
```

Last in first out (**LIFO**).

Queues

Typ: $\text{Qu } \alpha$

Initialwert:

```
empty :: Qu  $\alpha$ 
```

Wert ein/auslesen:

```
enq  ::  $\alpha \rightarrow \text{Qu } \alpha \rightarrow \text{Qu } \alpha$ 
```

```
first :: Qu  $\alpha \rightarrow \alpha$ 
```

```
deq  :: Qu  $\alpha \rightarrow \text{Qu } \alpha$ 
```

Test auf Leer:

```
isEmpty :: Qu  $\alpha \rightarrow \text{Bool}$ 
```

First in first out (**FIFO**)

Gleiche **Signatur**, unterschiedliche **Semantik**.

Eigenschaften von Stack

Last in first out (LIFO):

```
top (push a s) = a
```

```
pop (push a s) = s
```

```
isEmpty empty
```

```
not (isEmpty (push a s))
```

```
push a s /= empty
```

Eigenschaften von Queue

First in first out (FIFO):

```
first (enq a empty) = a
```

```
not (isEmpty q)  $\implies$  first (enq a q) = first q
```

```
deq (enq a empty) = empty
```

```
not (isEmpty q)  $\implies$  deq (enq a q) = enq a (deq q)
```

```
isEmpty (empty)
```

```
not (isEmpty (enq a q))
```

```
enq a q  $\neq$  empty
```

Implementation von Stack: Liste

Sehr einfach: ein Stack ist eine Liste

```
data Stack a = Stack [a] deriving (Show, Eq)
```

```
empty = Stack []
```

```
push a (Stack s) = Stack (a:s)
```

```
top (Stack []) = error "Stack: top on empty stack"
```

```
pop :: Stack a → Stack a
```

Implementation von Queue

- ▶ Mit einer Liste?
 - ▶ Problem: am Ende anfügen oder abnehmen ist teuer.
- ▶ Deshalb zwei Listen:
 - ▶ Erste Liste: zu entnehmende Elemente
 - ▶ Zweite Liste: hinzugefügte Elemente rückwärts
 - ▶ Invariante: erste Liste leer gdw. Queue leer

Repräsentation von Queue

<u>Operation</u>	<u>Resultat</u>	<u>Queue</u>	<u>Repräsentation</u>
------------------	-----------------	--------------	-----------------------

Repräsentation von Queue

Operation	Resultat	Queue	Repräsentation
empty			([], [])

Repräsentation von Queue

Operation	Resultat	Queue	Repräsentation
empty			([], [])
enq 9		9	([9], [])

Repräsentation von Queue

Operation	Resultat	Queue	Repräsentation
empty			([], [])
enq 9		9	([9], [])
enq 4		4 → 9	([9], [4])

Repräsentation von Queue

Operation	Resultat	Queue	Repräsentation
empty			([], [])
enq 9		9	([9], [])
enq 4		4 → 9	([9], [4])
enq 7		7 → 4 → 9	([9], [7, 4])

Repräsentation von Queue

Operation	Resultat	Queue	Repräsentation
empty			([], [])
enq 9		9	([9], [])
enq 4		4 → 9	([9], [4])
enq 7		7 → 4 → 9	([9], [7, 4])
deq	9	7 → 4	([4, 7], [])

Repräsentation von Queue

Operation	Resultat	Queue	Repräsentation
empty			([], [])
enq 9		9	([9], [])
enq 4		4 → 9	([9], [4])
enq 7		7 → 4 → 9	([9], [7, 4])
deq	9	7 → 4	([4, 7], [])
enq 5		5 → 7 → 4	([4, 7], [5])

Repräsentation von Queue

Operation	Resultat	Queue	Repräsentation
empty			([], [])
enq 9		9	([9], [])
enq 4		4 → 9	([9], [4])
enq 7		7 → 4 → 9	([9], [7, 4])
deq	9	7 → 4	([4, 7], [])
enq 5		5 → 7 → 4	([4, 7], [5])
enq 3		3 → 5 → 7 → 4	([4, 7], [3, 5])

Repräsentation von Queue

Operation	Resultat	Queue	Repräsentation
empty			([], [])
enq 9		9	([9], [])
enq 4		4 → 9	([9], [4])
enq 7		7 → 4 → 9	([9], [7, 4])
deq	9	7 → 4	([4, 7], [])
enq 5		5 → 7 → 4	([4, 7], [5])
enq 3		3 → 5 → 7 → 4	([4, 7], [3, 5])
deq	4	3 → 5 → 7	([7], [3, 5])

Repräsentation von Queue

Operation	Resultat	Queue	Repräsentation
empty			([], [])
enq 9		9	([9], [])
enq 4		4 → 9	([9], [4])
enq 7		7 → 4 → 9	([9], [7, 4])
deq	9	7 → 4	([4, 7], [])
enq 5		5 → 7 → 4	([4, 7], [5])
enq 3		3 → 5 → 7 → 4	([4, 7], [3, 5])
deq	4	3 → 5 → 7	([7], [3, 5])
deq	7	3 → 5	([5, 3], [])

Repräsentation von Queue

Operation	Resultat	Queue	Repräsentation
empty			([], [])
enq 9		9	([9], [])
enq 4		4 → 9	([9], [4])
enq 7		7 → 4 → 9	([9], [7, 4])
deq	9	7 → 4	([4, 7], [])
enq 5		5 → 7 → 4	([4, 7], [5])
enq 3		3 → 5 → 7 → 4	([4, 7], [3, 5])
deq	4	3 → 5 → 7	([7], [3, 5])
deq	7	3 → 5	([5, 3], [])
deq	5	3	([3], [])

Repräsentation von Queue

Operation	Resultat	Queue	Repräsentation
empty			([], [])
enq 9		9	([9], [])
enq 4		4 → 9	([9], [4])
enq 7		7 → 4 → 9	([9], [7, 4])
deq	9	7 → 4	([4, 7], [])
enq 5		5 → 7 → 4	([4, 7], [5])
enq 3		3 → 5 → 7 → 4	([4, 7], [3, 5])
deq	4	3 → 5 → 7	([7], [3, 5])
deq	7	3 → 5	([5, 3], [])
deq	5	3	([3], [])
deq	3		([], [])

Repräsentation von Queue

Operation	Resultat	Queue	Repräsentation
empty			([], [])
enq 9		9	([9], [])
enq 4		4 → 9	([9], [4])
enq 7		7 → 4 → 9	([9], [7, 4])
deq	9	7 → 4	([4, 7], [])
enq 5		5 → 7 → 4	([4, 7], [5])
enq 3		3 → 5 → 7 → 4	([4, 7], [3, 5])
deq	4	3 → 5 → 7	([7], [3, 5])
deq	7	3 → 5	([5, 3], [])
deq	5	3	([3], [])
deq	3		([], [])
deq	error		([], [])

Implementation

- ▶ Datentyp:

```
data Qu  $\alpha$  = Qu [  $\alpha$  ] [  $\alpha$  ]
```

- ▶ Leere Schlange: alles leer

```
empty = Qu [] []
```

- ▶ Invariante: erste Liste leer gdw. Queue leer

```
isEmpty (Qu xs _) = null xs
```

- ▶ Erstes Element steht vorne in erster Liste

```
first (Qu [] _) = error "Queue: first of empty Q"  
first (Qu (x:xs) _) = x
```

Implementation

- ▶ Bei `enq` und `deq` Invariante prüfen

```
enq x (Qu xs ys) = check xs (x:ys)
```

```
deq (Qu [] _)      = error "Queue: deq of empty Q"  
deq (Qu (_:xs) ys) = check xs ys
```

- ▶ Prüfung der Invariante **nach** dem Einfügen und Entnehmen
- ▶ `check` **garantiert** Invariante

```
check :: [α] → [α] → Qu α  
check [] ys = Qu (reverse ys) []  
check xs ys = Qu xs ys
```

Axiome als Eigenschaften

- ▶ Axiome können **getestet** oder **bewiesen** werden
- ▶ Tests finden **Fehler**, Beweis zeigt **Korrektheit**
- ▶ **Arten** von Tests:
 - ▶ **Unit tests** (JUnit, HUnit)
 - ▶ **Black Box** vs. **White Box**
 - ▶ **Zufallsbasiertes Testen**
- ▶ Funktionale Programme eignen sich **sehr gut** zum Testen

Zufallsbasiertes Testen in Haskell

- ▶ Werkzeug: QuickCheck
- ▶ Zufällige Werte einsetzen, Auswertung auf True prüfen
- ▶ Polymorphe Variablen nicht `testbar`
- ▶ Deshalb Typvariablen `instanziieren`
 - ▶ Typ muss genug Element haben (hier `Int`)
 - ▶ Durch Signatur `Typinstanz` erzwingen
- ▶ Freie Variablen der Eigenschaft werden `Parameter` der Testfunktion

Axiome mit QuickCheck testen

- ▶ Für das Lesen:

```
prop_read_empty :: Int → Bool
prop_read_empty a =
  lookup (empty :: Map Int Int) a == Nothing
```

```
prop_read_write :: Map Int Int → Int → Int → Bool
prop_read_write s a v =
  lookup (insert s a v) a == Just v
```

- ▶ Hier: Eigenschaften direkt als [Haskell-Prädikate](#)
- ▶ Es werden N Zufallswerte generiert und getestet ($N = 100$)

Axiome mit QuickCheck testen

- ▶ **Bedingte** Eigenschaft in quickCheck:
 - ▶ $A \implies B$ mit A, B Eigenschaften
 - ▶ Typ ist Property
 - ▶ Es werden solange Zufallswerte generiert, bis N die Vorbedingung erfüllende gefunden und getestet wurden, andere werden ignoriert.

```
prop_read_write_other ::  
  Map Int Int -> Int -> Int -> Int -> Property  
prop_read_write_other s a v b =  
  a /= b ==> lookup (insert s a v) b == lookup s b
```

Axiome mit QuickCheck testen

- ▶ Schreiben:

```
prop_write_write :: Map Int Int -> Int -> Int -> Int -> Int ->
prop_write_write s a v w =
  insert (insert s a v) a w == insert s a w
```

- ▶ Schreiben an anderer Stelle:

```
prop_write_other ::
  Map Int Int -> Int -> Int -> Int -> Int -> Int -> Property
prop_write_other s a v b w =
  a /= b ==> insert (insert s a v) b w ==
             insert (insert s b w) a v
```

- ▶ Test benötigt Gleichheit auf Map a b

Zufallswerte selbst erzeugen

- ▶ Problem: **Zufällige** Werte von **selbstdefinierten** Datentypen
 - ▶ Gleichverteiltheit nicht immer erwünscht (e.g. [a])
 - ▶ Konstruktion nicht immer offensichtlich (e.g. Map)
- ▶ In QuickCheck:
 - ▶ **Typklasse class** Arbitrary a für Zufallswerte
 - ▶ Eigene **Instanziierung** kann Verteilung und Konstruktion berücksichtigen
 - ▶ E.g. **Konstruktion** einer Map:
 - ▶ Zufällige Länge, dann aus sovielen zufälligen Werten **Map** konstruieren
 - ▶ Zufallswerte in Haskell?

Zusammenfassung

- ▶ **Signatur**: Typ und Operationen eines ADT
- ▶ **Axiome**: über Typen formulierte **Eigenschaften**
- ▶ **Spezifikation** = Signatur + Axiome
 - ▶ **Interface** zwischen Implementierung und Nutzung
 - ▶ **Testen** zur Erhöhung der Konfidenz und zum Fehlerfinden
 - ▶ **Beweisen** der Korrektheit
- ▶ **QuickCheck**:
 - ▶ Freie Variablen der Eigenschaften werden **Parameter** der Testfunktion
 - ▶ \implies für **bedingte** Eigenschaften

Praktische Informatik 3: Einführung in die Funktionale
Programmierung
Vorlesung vom 12.01.2011: Aktionen und Zustände

Christoph Lüth & Dennis Walter

Universität Bremen

Wintersemester 2010/11

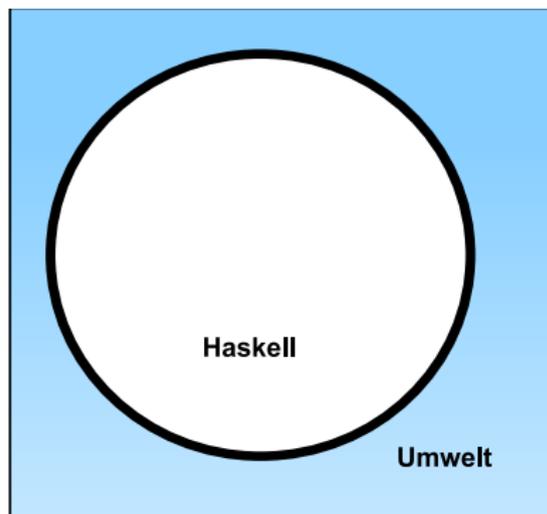
Fahrplan

- ▶ Teil I: Funktionale Programmierung im Kleinen
- ▶ Teil II: Funktionale Programmierung im Großen
 - ▶ Abstrakte Datentypen
 - ▶ Signaturen und Eigenschaften
 - ▶ Aktionen und Zustände
- ▶ Teil III: Funktionale Programmierung im richtigen Leben

Inhalt

- ▶ Ein/Ausgabe in funktionale Sprachen
- ▶ Wo ist das **Problem**?
- ▶ **Aktionen** und der Datentyp *IO*.
- ▶ Aktionen als **Werte**
- ▶ Aktionen als **Zustandstransformationen**

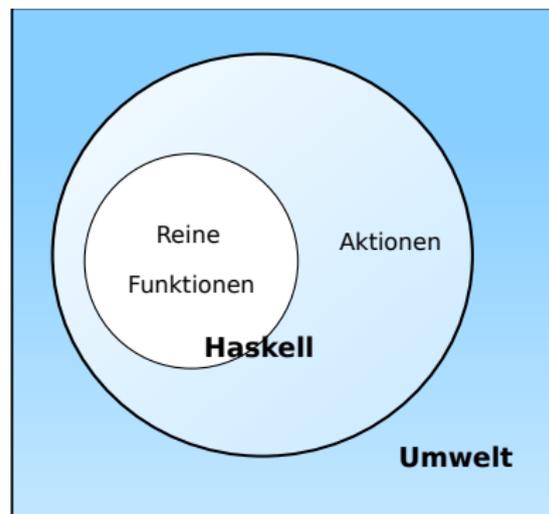
Ein- und Ausgabe in funktionalen Sprachen



Problem:

- ▶ Funktionen mit Seiteneffekten nicht referentiell transparent.
- ▶ `readString :: ... → String ??`

Ein- und Ausgabe in funktionalen Sprachen



Problem:

- ▶ Funktionen mit Seiteneffekten nicht referentiell transparent.
- ▶ `readString :: ... → String ??`

Lösung:

- ▶ Seiteneffekte am Typ erkennbar
- ▶ **Aktionen** können **nur** mit **Aktionen** komponiert werden
- ▶ „einmal Aktion, immer Aktion“

Aktionen als abstrakter Datentyp

- ▶ ADT mit Operationen **Komposition** und **Lifting**

- ▶ Signatur:

```
type IO  $\alpha$ 
```

```
( $\gg=$ )      :: IO  $\alpha \rightarrow (\alpha \rightarrow \text{IO } \beta) \rightarrow \text{IO } \beta$ 
```

```
return     ::  $\alpha \rightarrow \text{IO } \alpha$ 
```

- ▶ Plus **elementare** Operationen (lesen, schreiben etc)

Elementare Aktionen

- ▶ Zeile von `stdin` lesen:

```
getLine  :: IO String
```

- ▶ Zeichenkette auf `stdout` ausgeben:

```
putStr   :: String → IO ()
```

- ▶ Zeichenkette mit Zeilenvorschub ausgeben:

```
putStrLn :: String → IO ()
```

Einfache Beispiele

▶ Echo einfach

```
echo1 :: IO ()  
echo1 = getLine >>= putStrLn
```

▶ Echo mehrfach

```
echo :: IO ()  
echo = getLine >>= putStrLn >>= \_ → echo
```

▶ Was passiert hier?

- ▶ **Verknüpfen** von Aktionen mit $\gg=$
- ▶ Jede Aktion gibt **Wert** zurück

Noch ein Beispiel

- ▶ Umgekehrtes Echo:

```
ohce :: IO ()
ohce = getLine
      >>= \s → putStrLn (reverse s)
      >> ohce
```

- ▶ Was passiert hier?

- ▶ **Reine** Funktion `reverse` wird innerhalb von **Aktion** `putStrLn` genutzt
- ▶ Folgeaktion `ohce` benötigt **Wert** der vorherigen Aktion nicht
- ▶ Abkürzung: `>>`

```
p >> q = p >>= \_ → q
```

Die **do**-Notation

- ▶ Syntaktischer Zucker für IO:

```
echo =  
  getLine  
  >>= λs → putStrLn s  
  >> echo
```



```
echo =  
  do s ← getLine  
      putStrLn s  
      echo
```

- ▶ Rechts sind `>>=`, `>>` implizit.
- ▶ Es gilt die **Abseitsregel**.
- ▶ **Einrückung** der **ersten Anweisung** nach **do** bestimmt Abseits.

Drittes Beispiel

- ▶ Zählendes, endliches Echo

```
echo3 :: Int → IO ()
echo3 cnt = do
  putStr (show cnt ++ ":_")
  s ← getLine
  if s /= "" then do
    putStrLn $ show cnt ++ ":_" ++ s
    echo3 (cnt + 1)
  else return ()
```

- ▶ Was passiert hier?
 - ▶ Kombination aus Kontrollstrukturen und Aktionen
 - ▶ **Aktionen** als **Werte**
 - ▶ Geschachtelte **do**-Notation

Module in der Standardbücherei

- ▶ Ein/Ausgabe, Fehlerbehandlung (Modul IO)
- ▶ Zufallszahlen (Modul Random)
- ▶ Kommandozeile, Umgebungsvariablen (Modul System)
- ▶ Zugriff auf das Dateisystem (Modul Directory)
- ▶ Zeit (Modul Time)

Ein/Ausgabe mit Dateien

- ▶ Im `Prelude` vordefiniert:

- ▶ Dateien schreiben (überschreiben, anhängen):

```
type FilePath = String
writeFile      ::  FilePath → String → IO ()
appendFile    ::  FilePath → String → IO ()
```

- ▶ Datei lesen (verzögert):

```
readFile      ::  FilePath → IO String
```

- ▶ Mehr Operationen im Modul `IO` der Standardbibliothek

- ▶ `Buffered/Unbuffered`, `Seeking`, &c.
- ▶ Operationen auf `Handle`

Beispiel: Zeichen, Wörter, Zeilen zählen (wc)

```
wc :: String → IO ()
wc file =
  do cont ← readFile file
     putStrLn $ file ++ ":  
" ++
       show (length (lines cont),
            length (words cont),
            length cont)
```

- ▶ Nicht sehr effizient — Datei wird im Speicher gehalten.

Beispiel: wc verbessert.

- ▶ Effizienter: Dateiinhalt **einmal** traversieren

```
cnt :: Int → Int → Int → Bool → String
      → (Int, Int, Int)
cnt l w c _ [] = (l, w, c)
cnt l w c bl (x:xs)
  | isSpace x && not bl = cnt l' (w+1) (c+1) True xs
  | isSpace x && bl     = cnt l' w (c+1) True xs
  | otherwise         = cnt l w (c+1) False xs where
    l' = if x == '\n' then l+1 else l
```

Beispiel: wc verbessert.

- ▶ Effizienter: Dateiinhalt **einmal** traversieren

```
cnt :: Int → Int → Int → Bool → String
      → (Int, Int, Int)
cnt l w c _ [] = (l, w, c)
cnt l w c bl (x:xs)
  | isSpace x && not bl = cnt l' (w+1) (c+1) True xs
  | isSpace x && bl     = cnt l' w (c+1) True xs
  | otherwise         = cnt l w (c+1) False xs where
    l' = if x == '\n' then l+1 else l
```

- ▶ Hauptprogramm:

```
wc :: String → IO ()
wc file = do
  cont ← readFile file
  putStrLn $ file ++ ":␣" ++ show (cnt 0 0 0 False cont)
```

- ▶ Datei wird **verzögert** gelesen und **dabei** verbraucht.

Aktionen als Werte

- ▶ **Aktionen** sind **Werte** wie alle anderen.
- ▶ Dadurch **Definition** von **Kontrollstrukturen** möglich.
- ▶ Endlosschleife:

```
forever :: IO α → IO α  
forever a = a >> forever a
```

- ▶ Iteration (feste Anzahl):

```
forN :: Int → IO α → IO ()  
forN n a | n == 0    = return ()  
          | otherwise = a >> forN (n-1) a
```

- ▶ **Vordefinierte** Kontrollstrukturen (Control.Monad):
 - ▶ when, mapM, forM, sequence, ...

Fehlerbehandlung

- ▶ Fehler werden durch `IOError` repräsentiert

- ▶ Fehlerbehandlung durch `Ausnahmen` (ähnlich Java)

```
ioError  :: IOError → IO α    — "throw"  
catch    :: IO α → (IOError → IO α) → IO α
```

- ▶ Fehlerbehandlung nur in Aktionen

Fehler fangen und behandeln

- ▶ Fehlerbehandlung für wc:

```
wc2 :: String → IO ()  
wc2 file =  
    catch (wc file)  
        (λe → putStrLn $ "Fehler:␣" ++ show e)
```

- ▶ IOError kann analysiert werden (siehe Modul IO)
- ▶ read mit Ausnahme bei Fehler (statt Programmabbruch):

```
readIO :: Read a ⇒ String → IO a
```

So ein Zufall!

- ▶ Zufallswerte:

```
randomRIO :: (α, α) → IO α
```

- ▶ Warum ist randomIO **Aktion**?

So ein Zufall!

- ▶ Zufallswerte:

```
randomRIO :: (α, α) → IO α
```

- ▶ Warum ist randomIO **Aktion**?

- ▶ **Beispiel**: Aktionen zufällig oft ausführen

```
atmost :: Int → IO α → IO [α]
atmost most a =
  do l ← randomRIO (1, most)
     mapM id (replicate l a)
```

- ▶ Zufälligen String erzeugen

```
randomStr :: IO String
randomStr = atmost 40 (randomRIO ('a', 'z'))
```

Ausführbare Programme

- ▶ Eigenständiges Programm ist **Aktionen**
- ▶ **Hauptaktion**: main in Modul Main
- ▶ wc als eigenständiges Programm:

```
module Main where  
  
import System.Environment (getArgs)  
import Char(isSpace)  
  
main = do  
    args ← getArgs  
    mapM wc2 args
```

Funktionen mit Zustand

Theorem (Currying)

Folgende Typen sind *isomorph*:

$$A \times B \rightarrow C \cong A \rightarrow B \rightarrow C$$

► In Haskell: folgende Funktionen sind *invers*:

```
curry    :: (( $\alpha$ ,  $\beta$ )  $\rightarrow$   $\gamma$ )  $\rightarrow$   $\alpha \rightarrow \beta \rightarrow \gamma$   
uncurry  :: ( $\alpha \rightarrow \beta \rightarrow \gamma$ )  $\rightarrow$  ( $\alpha$ ,  $\beta$ )  $\rightarrow$   $\gamma$ 
```

Funktionen mit Zustand

- ▶ Idee: Seiteneffekt **explizit** machen
- ▶ Funktion $f : A \rightarrow B$ mit Seiteneffekt in **Zustand** S :

$$\begin{aligned} f : A \times S &\rightarrow B \times S \\ &\cong \\ f : A &\rightarrow S \rightarrow B \times S \end{aligned}$$

- ▶ Datentyp: $S \rightarrow B \times S$
- ▶ Komposition: Funktionskomposition und **uncurry**

In Haskell: Zustände **explizit**

- ▶ Datentyp: Berechnung mit Seiteneffekt in Typ Σ :

```
type State  $\Sigma$   $\alpha = \Sigma \rightarrow (\alpha, \Sigma)$ 
```

- ▶ Komposition zweier solcher Berechnungen:

```
comp :: State  $\Sigma$   $\alpha \rightarrow (\alpha \rightarrow \text{State } \Sigma \beta) \rightarrow \text{State } \Sigma \beta$   
comp f g = uncurry g  $\circ$  f
```

- ▶ Lifting:

```
lift ::  $\alpha \rightarrow \text{State } \Sigma \alpha$   
lift = curry id
```

Beispiel: Ein Zähler

- ▶ Datentyp:

```
type WithCounter  $\alpha$  = State Int  $\alpha$ 
```

- ▶ Zähler erhöhen:

```
tick :: WithCounter ()  
tick i = ((), i+1)
```

- ▶ Zähler auslesen:

```
read :: WithCounter Int  
read i = (i, i)
```

- ▶ Zähler zurücksetzen:

```
reset :: WithCounter ()  
reset i = ((), 0)
```

Implizite vs. explizite Zustände

- ▶ Nachteil: Zustand ist **explizit**
 - ▶ Kann **dupliziert** werden
- ▶ Daher: Zustand **implizit** machen
 - ▶ Datentyp **verkapseln**
 - ▶ Signatur `State`, `comp`, `lift`, elementare Operationen

Aktionen als Zustandstransformationen

- ▶ **Idee:** Aktionen sind Transformationen auf Systemzustand S
- ▶ S beinhaltet
 - ▶ Speicher als Abbildung $A \rightarrow V$ (Adressen A , Werte V)
 - ▶ Zustand des Dateisystems
 - ▶ Zustand des Zufallsgenerators
- ▶ In Haskell: Typ `RealWorld`
 - ▶ “Virtueller” Typ, Zugriff nur über elementare Operationen
 - ▶ Entscheidend nur Reihenfolge der Aktionen

Zusammenfassung

- ▶ Ein/Ausgabe in Haskell durch **Aktionen**
- ▶ **Aktionen** (Typ $\text{IO } \alpha$) sind seiteneffektbehaftete Funktionen
- ▶ **Komposition** von Aktionen durch

```
(\gg=)    :: IO \alpha \to (\alpha \to IO \beta) \to IO \beta  
return  :: \alpha \to IO \alpha
```

- ▶ **do**-Notation
- ▶ Fehlerbehandlung durch Ausnahmen (`IOError`, `catch`).
- ▶ Verschiedene Funktionen der Standardbibliothek:
 - ▶ Prelude: `getLine`, `putStr`, `putStrLn`, `readFile`, `writeFile`
 - ▶ Module: `IO`, `Random`
- ▶ Aktionen sind **implementiert** als **Zustandstransformationen**

Praktische Informatik 3: Einführung in die Funktionale
Programmierung
Vorlesung vom 19.01.2011: Effizienzaspekte

Christoph Lüth & Dennis Walter

Universität Bremen

Wintersemester 2010/11

Fahrplan

- ▶ Teil I: Funktionale Programmierung im Kleinen
- ▶ Teil II: Funktionale Programmierung im Großen
- ▶ Teil III: Funktionale Programmierung im richtigen Leben
 - ▶ Effizient Funktional Programmieren
 - ▶ Fallstudie: Kombinatoren
 - ▶ Eine Einführung in Scala
 - ▶ Rückblick & Ausblick

Inhalt

- ▶ **Zeitbedarf**: Endrekursion — **while** in Haskell
- ▶ **Platzbedarf**: Speicherlecks
- ▶ “**Unendliche**” Datenstrukturen
- ▶ Verschiedene andere **Performancefallen**:
 - ▶ Überladene Funktionen, Listen

Inhalt

- ▶ **Zeitbedarf**: Endrekursion — **while** in Haskell
- ▶ **Platzbedarf**: Speicherlecks
- ▶ “**Unendliche**” Datenstrukturen
- ▶ Verschiedene andere **Performancefallen**:
 - ▶ Überladene Funktionen, Listen
- ▶ “**Usual Disclaimers Apply**”:
 - ▶ Erste Lösung: bessere **Algorithmen**
 - ▶ Zweite Lösung: **Büchereien** nutzen

Effizienzaspekte

- ▶ Zur **Verbesserung** der Effizienz:
 - ▶ Analyse der **Auswertungsstrategie**
 - ▶ ... und des **Speichermanagement**
- ▶ Der ewige Konflikt: **Geschwindigkeit vs. Platz**

Effizienzaspekte

- ▶ Zur **Verbesserung** der Effizienz:
 - ▶ Analyse der **Auswertungsstrategie**
 - ▶ ... und des **Speichermanagement**
- ▶ Der ewige Konflikt: **Geschwindigkeit** vs. **Platz**
- ▶ Effizienzverbesserungen durch
 - ▶ **Endrekursion**: Iteration in funktionalen Sprachen
 - ▶ **Striktheit**: **Speicherlecks** vermeiden (bei verzögerter Auswertung)
- ▶ Vorteil: Effizienz **muss nicht** im Vordergrund stehen

Endrekursion

Definition (Endrekursion)

Eine Funktion ist **endrekursiv**, wenn

- (i) es genau **einen** rekursiven Aufruf gibt,
- (ii) der **nicht** innerhalb eines **geschachtelten Ausdrucks** steht.

- ▶ D.h. darüber **nur Fallunterscheidungen**: **case** oder **if**
- ▶ Entspricht **goto** oder **while** in imperativen Sprachen.
- ▶ Wird in **Sprung** oder **Schleife** übersetzt.
- ▶ Braucht **keinen Platz** auf dem Stack.

Beispiel: Fakultät

- ▶ `fac1` **nicht** endrekursiv:

```
fac1  :: Integer → Integer  
fac1 n  = if n == 0 then 1 else n * fac1 (n-1)
```

Beispiel: Fakultät

- ▶ `fac1` **nicht** endrekursiv:

```
fac1 :: Integer → Integer
fac1 n = if n == 0 then 1 else n * fac1 (n-1)
```

- ▶ `fac2` endrekursiv:

```
fac2 :: Integer → Integer
fac2 n = fac' n 1 where
  fac' :: Integer → Integer → Integer
  fac' n acc = if n == 0 then acc
               else fac' (n-1) (n*acc)
```

- ▶ `fac1` verbraucht Stack, `fac2` nicht.

Beispiel: Listen umdrehen

- ▶ Liste umdrehen, **nicht** endrekursiv:

```
rev '  :: [a] → [a]
rev ' []      = []
rev ' (x:xs) = rev ' xs ++ [x]
```

- ▶ Hängt auch noch **hinten** an — $O(n^2)$!

Beispiel: Listen umdrehen

- ▶ Liste umdrehen, **nicht** endrekursiv:

```
rev' :: [a] → [a]
rev' []      = []
rev' (x:xs) = rev' xs ++ [x]
```

- ▶ Hängt auch noch **hinten** an — $O(n^2)$!

- ▶ Liste umdrehen, **endrekursiv** und $O(n)$:

```
rev :: [a] → [a]
rev xs = rev0 xs [] where
  rev0 []      ys = ys
  rev0 (x:xs) ys = rev0 xs (x:ys)
```

- ▶ Beispiel: last (rev [1..10000])

Überführung in Endrekursion

- ▶ Gegeben Funktion

$$f' : S \rightarrow T$$

$$f' x = \text{if } B x \text{ then } H x \\ \text{else } \phi (f' (K x)) (E x)$$

- ▶ Mit $K : S \rightarrow S$, $\phi : T \rightarrow T \rightarrow T$, $E : S \rightarrow T$, $H : S \rightarrow T$.

- ▶ **Voraussetzung:** ϕ assoziativ, $e : T$ neutrales Element

- ▶ Dann ist **endrekursive** Form:

$$f : S \rightarrow T$$

$$f x = g x e \text{ where} \\ g x y = \text{if } B x \text{ then } \phi (H x) y \\ \text{else } g (K x) (\phi (E x) y)$$

Beispiel

- ▶ Länge einer Liste (nicht-endrekursiv)

```
length' :: [a] → Int
length' xs = if null xs then 0
             else 1 + length' (tail xs)
```

- ▶ Zuordnung der Variablen:

$$\begin{array}{ll} K(x) \mapsto \text{tail } x & B(x) \mapsto \text{null } x \\ E(x) \mapsto 1 & H(x) \mapsto 0 \\ \phi(x, y) \mapsto x + y & e \mapsto 0 \end{array}$$

- ▶ Es gilt: $\phi(x, e) = x + 0 = x$ (0 neutrales Element)

Beispiel

- ▶ Damit **endrekursive** Variante:

```
length :: [a] → Int
length xs = len xs 0 where
  len xs y = if null xs then y -- was: y+0
             else len (tail xs) (1+ y)
```

- ▶ Allgemeines **Muster**:

- ▶ Monoid (ϕ, e) : ϕ assoziativ, e neutrales Element.
- ▶ Zusätzlicher Parameter **akkumuliert** Resultat.

Endrekursive Aktionen

► Nicht endrekursiv:

```
getLines' :: IO String
getLines' = do str ← getLine
              if null str then return ""
              else do rest ← getLines'
                      return (str ++ rest)
```

► Endrekursiv:

```
getLines :: IO String
getLines = getit "" where
  getit res = do str ← getLine
                if null str then return res
                else getit (res ++ str)
```

Fortgeschrittene Endrekursion

- ▶ **Akkumulation** von Ergebniswerten durch **closures**
 - ▶ **closure**: partiell applizierte Funktion
- ▶ Beispiel: die Klasse `Show`
 - ▶ Nur Methode `show` wäre zu langsam ($O(n^2)$):

```
class Show a where  
  show :: a → String
```

Fortgeschrittene Endrekursion

- ▶ **Akkumulation** von Ergebniswerten durch **closures**

- ▶ **closure**: partiell applizierte Funktion

- ▶ Beispiel: die Klasse `Show`

- ▶ Nur Methode `show` wäre zu langsam ($O(n^2)$):

```
class Show a where
  show :: a → String
```

- ▶ Deshalb zusätzlich

```
showsPrec :: Int → a → String → String
show x = showsPrec 0 x ""
```

- ▶ `String` wird erst aufgebaut, wenn er ausgewertet wird ($O(n)$).

Beispiel: Mengen als Listen

```
data Set a = Set [a]
```

Zu langsam wäre

```
instance Show a ⇒ Show (Set a) where  
  show (Set elems) =  
    "{" ++ intercalate "," (map show elems) ++ "}"
```

Deshalb besser

```
instance Show a ⇒ Show (Set a) where  
  showsPrec i (Set elems) = showElems elems where  
    showElems [] = ("{" ++)  
    showElems (x:xs) = ('{' :) ∘ shows x ∘ showl xs  
    where showl [] = ('}' :)  
          showl (x:xs) = (',' :) ∘ shows x ∘ showl xs
```

Effizienz durch “unendliche” Datenstrukturen

- ▶ Listen müssen nicht **endlich repräsentierbar** sein:

- ▶ Beispiel: “unendliche” Liste [2,2,2, ...]

```
twos = 2 : twos
```

- ▶ Liste der natürlichen Zahlen:

```
nat = nats 0 where nats n = n : nats (n+ 1)
```

- ▶ Syntaktischer Zucker:

```
nat = [0 ..]
```

- ▶ Bildung von unendlichen Listen:

```
cycle :: [a] → [a]      repeat :: a → [a]  
cycle xs = xs ++ cycle xs  repeat x = x : repeat x
```

- ▶ Nützlich für Listen mit unbekannter Länge
- ▶ **Obacht:** Induktion nur für **endliche** Listen gültig.

Berechnung der ersten n Primzahlen

- ▶ Eratosthenes — aber bis wo sieben?

Berechnung der ersten n Primzahlen

- ▶ **Eratosthenes** — aber bis wo sieben?
- ▶ Lösung: Berechnung **aller** Primzahlen, davon die **ersten** n .

```
sieve :: [Integer] → [Integer]
sieve (p:ps) =
  p:(sieve (filter (\n → n `mod` p /= 0) ps))
```

- ▶ Keine Rekursionsverankerung (sieve [])

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

- ▶ Von allen Primzahlen die **ersten** n :

```
firstprimes :: Int → [Integer]
firstprimes n = take n primes
```

Fibonacci-Zahlen

- ▶ Aus der Kaninchenzucht.
- ▶ Sollte jeder Informatiker kennen.

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

- ▶ Problem: baumartige Rekursion, exponentieller Aufwand.

Fibonacci-Zahlen als Strom

- ▶ Lösung: zuvor berechnete Teilergebnisse wiederverwenden.
- ▶ Sei `fibs :: [Integer]` Strom aller Fibonaccizahlen:

```
fibs 1 1 2 3 5 8 13 21 34 55
tail fibs 1 2 3 5 8 13 21 34 55
tail (tail fibs) 2 3 5 8 13 21 34 55
```

- ▶ Damit ergibt sich:

```
fibs :: [Integer]
fibs = 1 : 1 : zipWith (+) fibs (tail fibs)
```

- ▶ n-te Fibonaccizahl mit `fibs !! n`
- ▶ Aufwand: **linear**, da `fibs` nur einmal ausgewertet wird.

Unendliche Datenstrukturen

- ▶ Endliche Repräsentierbarkeit für **beliebige** Datenstrukturen
- ▶ E.g. Bäume:

```
data Tree a = Null | Node (Tree a) a (Tree a)
           deriving Show

twoTree      = Node twoTree 2 twoTree

rightSpline n = Node Null n (rightSpline (n+1))
```

- ▶ twoTree, twos mit **Zeigern** darstellbar (e.g. Java, C)
- ▶ rightSpline 0, nat **nicht** mit darstellbar
- ▶ Damit beispielsweise auch **Graphen** modellierbar

Implementation und Repräsentation von Datenstrukturen

- ▶ Datenstrukturen werden intern durch **Objekte** in einem **Heap** repräsentiert
- ▶ Bezeichner werden an **Referenzen** in diesen Heap gebunden
- ▶ Unendliche Datenstrukturen haben zyklische Verweise
 - ▶ Kopf wird nur **einmal** ausgewertet.

```
cycle (trace "Foo!" [5])
```

- ▶ **Anmerkung:** unendlich Datenstrukturen nur sinnvoll für **nicht-strikte** Funktionen

Speicherlecks

- ▶ **Garbage collection** gibt **unbenutzten** Speicher wieder frei.
 - ▶ **Unbenutzt**: Bezeichner nicht mehr im **erreichbar**
- ▶ Verzögerte Auswertung **effizient**, weil nur bei **Bedarf** ausgewertet wird
 - ▶ Aber Obacht: **Speicherlecks!**

Speicherlecks

- ▶ **Garbage collection** gibt **unbenutzten** Speicher wieder frei.
 - ▶ **Unbenutzt**: Bezeichner nicht mehr im **erreichbar**
- ▶ Verzögerte Auswertung **effizient**, weil nur bei **Bedarf** ausgewertet wird
 - ▶ Aber Obacht: **Speicherlecks!**
- ▶ Eine Funktion hat ein **Speicherleck**, wenn Speicher **unnötig** lange im Zugriff bleibt.
 - ▶ “Echte” Speicherlecks wie in C/C++ **nicht möglich**.
- ▶ Beispiel: `getLines`, `fac2`
 - ▶ Zwischenergebnisse werden **nicht ausgewertet**.
 - ▶ Insbesondere ärgerlich bei **nicht-terminierenden Funktionen**.

Striktheit

- ▶ **Strikte Argumente** erlauben Auswertung **vor** Aufruf
 - ▶ Dadurch **konstanter** Platz bei **Endrekursion**.
- ▶ **Erzwungene Striktheit**: $\text{seq} :: \alpha \rightarrow \beta \rightarrow \beta$

$$\perp \text{ 'seq' } b = \perp$$

$$a \text{ 'seq' } b = b$$

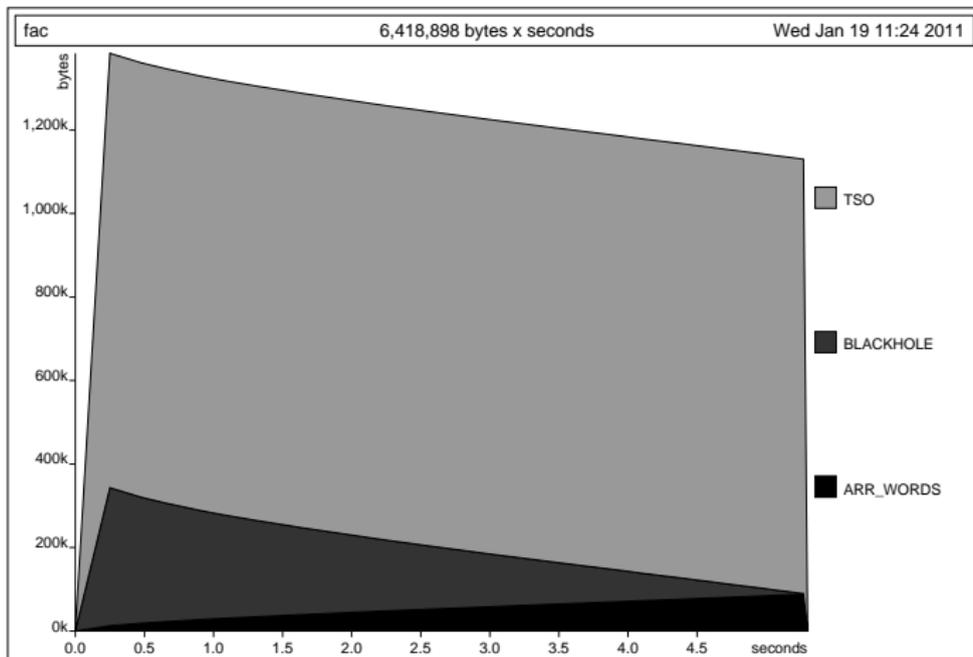
- ▶ `seq` vordefiniert (nicht in Haskell definierbar)
- ▶ $(\$!) :: (a \rightarrow b) \rightarrow a \rightarrow b$ strikte Funktionsanwendung

$$f \$! x = x \text{ 'seq' } f x$$

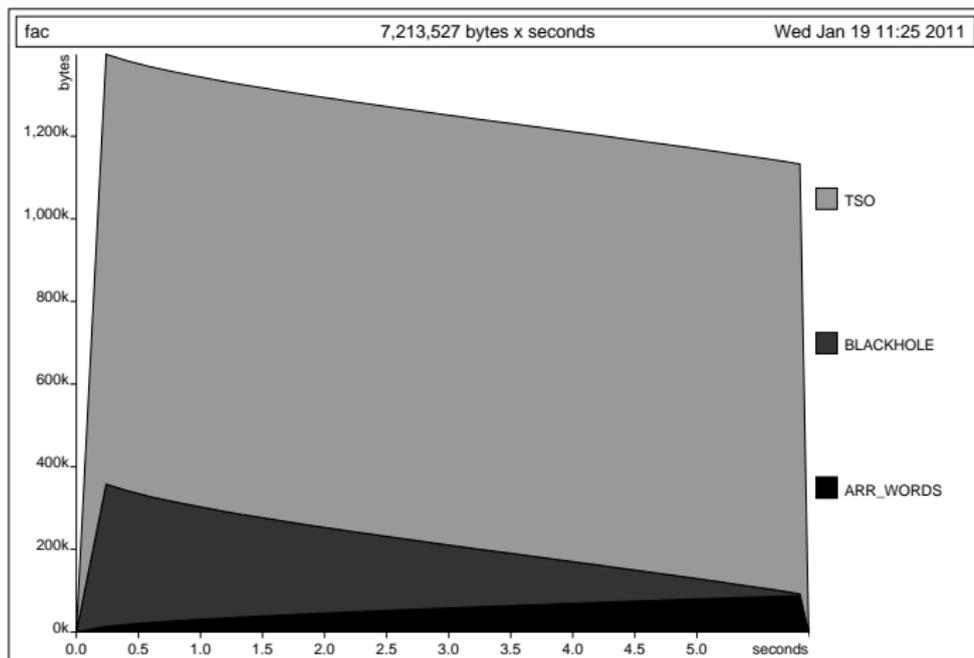
- ▶ ghc macht **Striktheitsanalyse**
- ▶ Fakultät in konstantem Platzaufwand

```
fac3 :: Integer -> Integer
fac3 n = fac' n 1 where
  fac' n acc = seq acc $ if n == 0 then acc
                    else fac' (n-1) (n*acc)
```

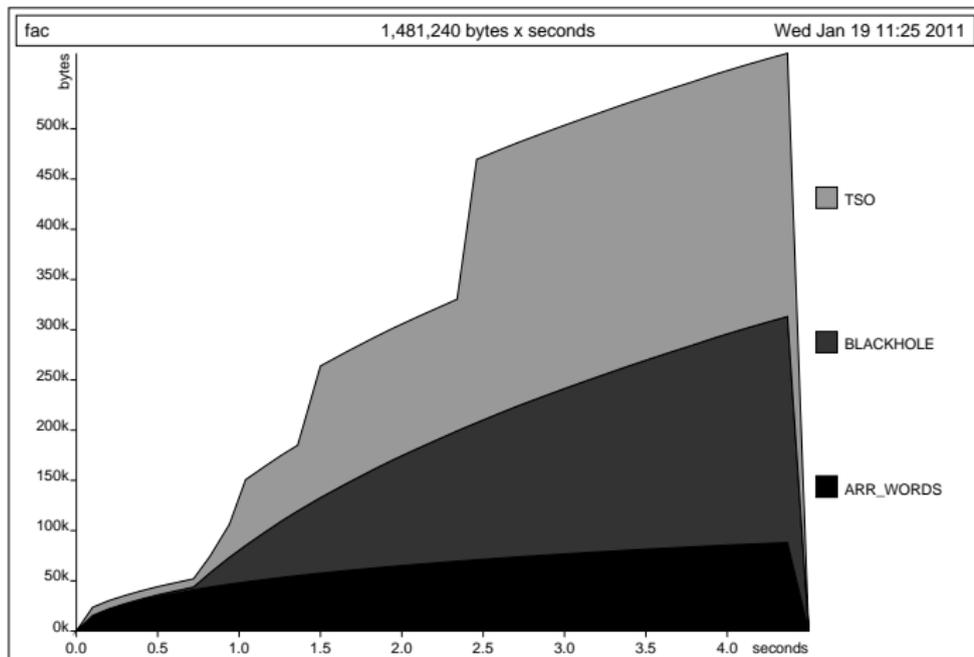
Speicherprofil: fac1 50000, nicht optimiert



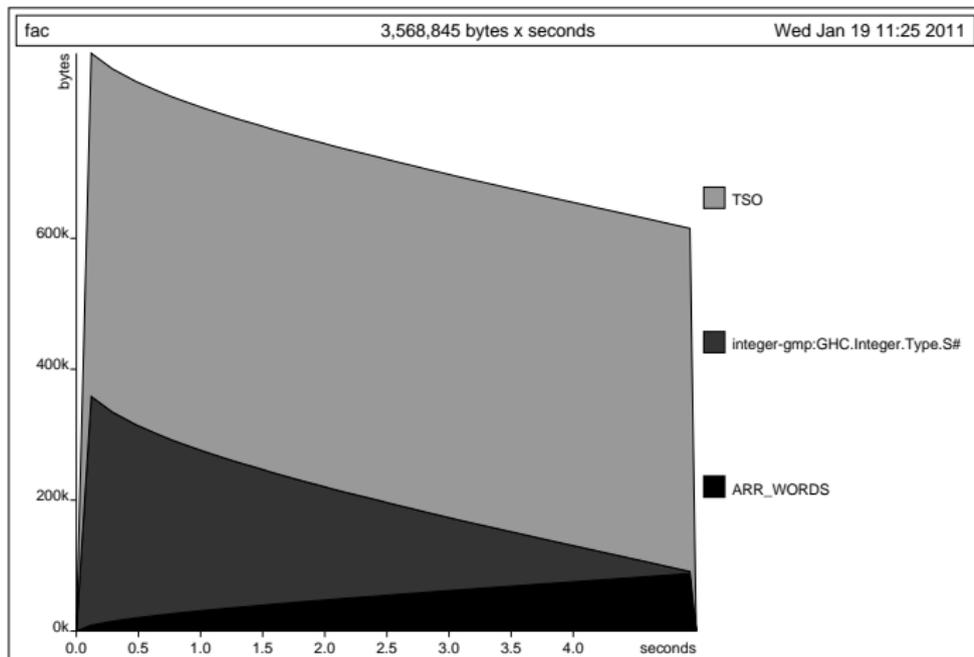
Speicherprofil: fac2 50000, nicht optimiert



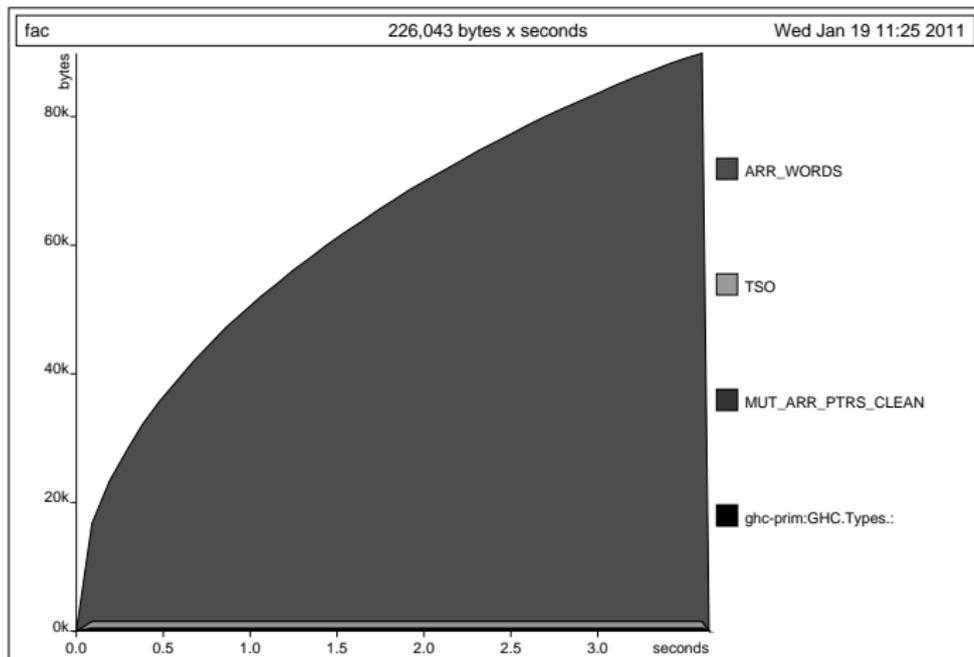
Speicherprofil: fac3 50000, nicht optimiert



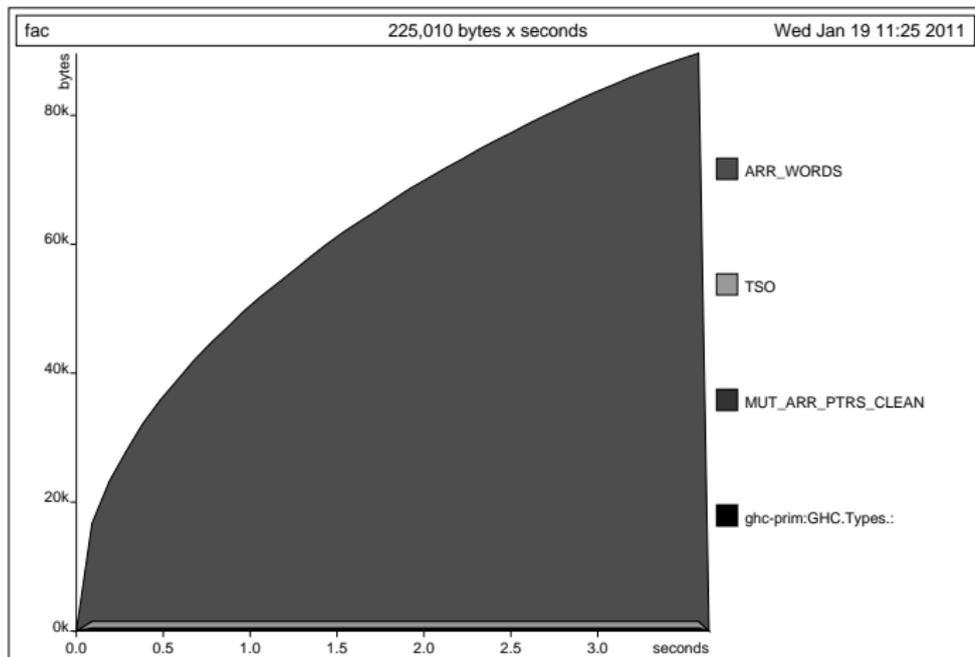
Speicherprofil: fac1 50000, optimiert



Speicherprofil: fac2 50000, optimiert



Speicherprofil: fac3 50000, optimiert



Fazit Speicherprofile

- ▶ Geschwindigkeitsgewinn durch Endrekursion **nur** mit Striktheit
- ▶ Optimierung des `ghc` meist ausreichend für Striktheitsanalyse, aber **nicht** für Endrekursion

foldr vs. foldl

- ▶ foldr ist **nicht endrekursiv**:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
```

- ▶ foldl ist **endrekursiv**:

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f z [] = z
foldl f z (x:xs) = foldl f (f z x) xs
```

- ▶ foldl' ist **strikt** und **endrekursiv**:

```
foldl' :: (a -> b -> a) -> a -> [b] -> a
foldl' f a [] = a
foldl' f a (x:xs) =
  let a' = f a x in a' `seq` foldl' f a' xs
```

- ▶ Für Monoid (ϕ, e) gilt: $\text{foldr } \phi \ e \ l = \text{foldl } (\text{flip } \phi) \ e \ l$

Wann welches fold?

- ▶ foldl endrekursiv, aber traversiert immer die **ganze** Liste.

- ▶ foldl' ferner strikt und konstanter Platzaufwand

- ▶ Wann welches fold?

- ▶ Strikte Funktionen mit foldl' falten:

```
rev2 :: [a] → [a]
rev2 = foldl' (flip (:)) []
```

- ▶ Wenn nicht die ganze Liste benötigt wird, mit foldr falten:

```
all :: (a → Bool) → [a] → Bool
all p = foldr ((&&) ∘ p) True
```

- ▶ Potenziell **unendliche** Listen **immer** mit foldr falten.

Überladene Funktionen sind langsam.

- ▶ Typklassen sind elegant aber **langsam**.
- ▶ Implementierung von Typklassen: **Verzeichnis** (dictionary) von Klassenfunktionen.
- ▶ Überladung wird zur **Laufzeit** aufgelöst

Überladene Funktionen sind langsam.

- ▶ Typklassen sind elegant aber **langsam**.
 - ▶ Implementierung von Typklassen: **Verzeichnis** (dictionary) von Klassenfunktionen.
 - ▶ Überladung wird zur **Laufzeit** aufgelöst
- ▶ Bei kritischen Funktionen: **Spezialisierung erzwingen** durch Angabe der Signatur
- ▶ NB: **Zahlen** (numerische Literale) sind in Haskell **überladen**!
 - ▶ Bsp: `facts` hat den Typ `Num a => a -> a`

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

Listen als Performance-Falle

- ▶ Listen sind **keine** Felder oder endliche Abbildungen
- ▶ Listen:
 - ▶ Beliebig lang
 - ▶ Zugriff auf n -tes Element in **linearer** Zeit.
 - ▶ Abstrakt: frei erzeugter Datentyp aus Kopf und Rest
- ▶ Felder `Array ix a` (Modul `Array` aus der Standardbibliothek)
 - ▶ Feste Größe (Untermenge von ix)
 - ▶ Zugriff auf n -tes Element in **konstanter** Zeit.
 - ▶ Abstrakt: Abbildung Index auf Daten
- ▶ Endliche Abbildung `Map k v` (Modul `Data.Map`)
 - ▶ Beliebige Größe
 - ▶ Zugriff auf n -tes Element in **sublinearer** Zeit.
 - ▶ Abstrakt: Abbildung Schlüsselbereich k auf Wertebereich v

Zusammenfassung

- ▶ **Endrekursion**: **while** für Haskell.
 - ▶ Überführung in Endrekursion meist möglich.
 - ▶ Noch besser sind **strikte Funktionen**.
- ▶ **Speicherlecks** vermeiden: **Striktheit** und **Endrekursion**
- ▶ **Compileroptimierung** nutzen
- ▶ Datenstrukturen müssen nicht **endlich repräsentierbar** sein
- ▶ **Überladene Funktionen** sind langsam.
- ▶ **Listen** sind keine Felder oder endliche Abbildungen.

Praktische Informatik 3: Einführung in die Funktionale
Programmierung
Vorlesung vom 26.01.2011: Kombinatoren

Christoph Lüth & Dennis Walter

Universität Bremen

Wintersemester 2010/11

Fahrplan

- ▶ Teil I: Funktionale Programmierung im Kleinen
- ▶ Teil II: Funktionale Programmierung im Großen
- ▶ Teil III: Funktionale Programmierung im richtigen Leben
 - ▶ Effizient Funktional Programmieren
 - ▶ Fallstudie: Kombinatoren
 - ▶ Eine Einführung in Scala
 - ▶ Rückblick & Ausblick

Kombinatoren im engeren Sinne

Definition (Kombinator)

Ein **Kombinator** ist ein punktfrei definierte Funktion höherer Ordnung.

- ▶ Herkunft: **Kombinatorlogik** (Schönfinkel, 1924)

$$K x y \triangleright x$$

$$S x y z \triangleright x z (y z)$$

$$I x \triangleright x$$

S , K , I sind **Kombinatoren**

Kombinatoren im engeren Sinne

Definition (Kombinator)

Ein **Kombinator** ist ein punktfrei definierte Funktion höherer Ordnung.

- ▶ Herkunft: **Kombinatorlogik** (Schönfinkel, 1924)

$$K x y \triangleright x$$

$$S x y z \triangleright x z (y z)$$

$$I x \triangleright x$$

S , K , I sind **Kombinatoren**

- ▶ Fun fact #1: kann alle berechenbaren Funktionen ausdrücken

Kombinatoren im engeren Sinne

Definition (Kombinator)

Ein **Kombinator** ist ein punktfrei definierte Funktion höherer Ordnung.

- ▶ Herkunft: **Kombinatorlogik** (Schönfinkel, 1924)

$$K \ x \ y \triangleright x$$

$$S \ x \ y \ z \triangleright x \ z \ (y \ z)$$

$$I \ x \triangleright x$$

S , K , I sind **Kombinatoren**

- ▶ Fun fact #1: kann alle berechenbaren Funktionen ausdrücken
- ▶ Fun fact #2: S und K sind genug: $I = S \ K \ K$

Kombinatoren als Entwurfsmuster

- ▶ **Kombination** von Basisoperationen zu komplexen Operationen
- ▶ Kombinatoren als **Muster** zur Problemlösung:
 - ▶ **Einfache** Basisoperationen
 - ▶ **Wenige** Kombinationsoperationen
 - ▶ Alle anderen Operationen **abgeleitet**
- ▶ **Kompositionalität:**
 - ▶ Gesamtproblem läßt sich **zerlegen**
 - ▶ Gesamtlösung durch **Zusammensetzen** der Einzellösungen

Beispiele

- ▶ Parserkombinatoren
- ▶ Grafikkombinatoren mit der `HGL`
- ▶ Grafikkombinatoren mit `tinySVG`

Beispiel #1: Parser

- ▶ **Parser** bilden Eingabe auf Parsierungen ab
 - ▶ Mehrere Parsierungen möglich
 - ▶ Backtracking möglich
- ▶ Kombinatoransatz:
 - ▶ **Basisparser** erkennen **Terminalsymbole**
 - ▶ **Parserkombinatoren** zur Konstruktion:
 - ▶ Sequenzierung (erst A , dann B)
 - ▶ Alternierung (entweder A oder B)
 - ▶ Abgeleitete Kombinatoren (z.B. Listen A^* , nicht-leere Listen A^+)

Modellierung in Haskell

Welcher **Typ** für Parser?

type Parse = ?

Modellierung in Haskell

Welcher **Typ** für Parser?

type Parse a b = ?

- ▶ Parametrisiert über **Eingabetyp** (Token) a und **Ergebnis** b

Modellierung in Haskell

Welcher **Typ** für Parser?

type Parse a b = [a] → b

- ▶ Parametrisiert über **Eingabetyp** (Token) a und **Ergebnis** b
- ▶ Parser übersetzt **Token** in **abstrakte Syntax**

Modellierung in Haskell

Welcher **Typ** für Parser?

type Parse a b = [a] → (b, [a])

- ▶ Parametrisiert über **Eingabetyp** (Token) a und **Ergebnis** b
- ▶ Parser übersetzt **Token** in **abstrakte Syntax**
- ▶ Muss **Rest der Eingabe** modellieren

Modellierung in Haskell

Welcher **Typ** für Parser?

type Parse a b = [a] → [(b, [a])]

- ▶ Parametrisiert über **Eingabetyp** (Token) a und **Ergebnis** b
- ▶ Parser übersetzt **Token** in **abstrakte Syntax**
- ▶ Muss **Rest der Eingabe** modellieren
- ▶ Muss **mehrdeutige Ergebnisse** modellieren

Modellierung in Haskell

Welcher **Typ** für Parser?

type Parse a b = [a] → [(b, [a])]

- ▶ Parametrisiert über **Eingabetyp** (Token) a und **Ergebnis** b
- ▶ Parser übersetzt **Token** in **abstrakte Syntax**
- ▶ Muss **Rest der Eingabe** modellieren
- ▶ Muss **mehrdeutige Ergebnisse** modellieren
- ▶ Beispiel: "a+b*c" \rightsquigarrow [
 (Var "a", "+b*c"),
 (Plus (Var "a") (Var "b") , "*c"),
 (Plus (Var "a") (Times (Var "b") (Var "c")), "")]

Basisparser

- ▶ Erkennt **nichts**:

```
none  :: Parse a b
none = const []
```

- ▶ Erkennt **alles**:

```
succeed :: b → Parse a b
succeed b inp = [(b, inp)]
```

- ▶ Erkennt **einzelne Token**:

```
spot  :: (a → Bool) → Parse a a
spot p []      = []
spot p (x:xs) = if p x then [(x, xs)] else []
```

```
token :: Eq a ⇒ a → Parse a a
token t = spot (λc → t == c)
```

- ▶ Warum nicht none, succeed durch spot? Typ!

Basiskombinatoren: alt, >*>

- ▶ **Alternierung:**

- ▶ Erste Alternative wird **bevorzugt**

```
infixl 3 'alt '  
alt  :: Parse a b → Parse a b → Parse a b  
alt p1 p2 i = p1 i ++ p2 i
```

Basiskombinatoren: alt, >*>

▶ Alternierung:

- ▶ Erste Alternative wird bevorzugt

```
infixl 3 'alt'  
alt :: Parse a b → Parse a b → Parse a b  
alt p1 p2 i = p1 i ++ p2 i
```

▶ Sequenzierung:

- ▶ Rest des ersten Parsers als Eingabe für den zweiten

```
infixl 5 >*>  
(>*>) :: Parse a b → Parse a c → Parse a (b, c)  
(>*>) p1 p2 i =  
  concatMap (\(b, r) →  
    map (\(c, s) → ((b, c), s)) (p2 r)) (p1 i)
```

Basiskombinatoren: use

- ▶ Rückgabe weiterverarbeiten:

```
infix 4 'use', 'use2'
use :: Parse a b → (b → c) → Parse a c
use p f i = map (λ(o, r) → (f o, r)) (p i)

use2 :: Parse a (b, c) → (b → c → d) → Parse a d
use2 p f = use p (uncurry f)
```

- ▶ Damit z.B. Sequenzierung rechts/links:

```
infixl 5 *>, >*
(*>) :: Parse a b → Parse a c → Parse a c
(>*) :: Parse a b → Parse a c → Parse a b
p1 *> p2 = p1 >*> p2 'use' snd
p1 >* p2 = p1 >*> p2 'use' fst
```

Abgeleitete Kombinatoren

- ▶ Listen: $A^* ::= AA^* \mid \epsilon$

```
list  :: Parse a b → Parse a [b]
list p = p >*> list p 'use2' (:)
        'alt' succeed []
```

- ▶ Nicht-leere Listen: $A^+ ::= AA^*$

```
some  :: Parse a b → Parse a [b]
some p = p >*> list p 'use2' (:)
        'alt' succeed []
```

- ▶ NB. Präzedenzen: $>*>$ (5) vor use (4) vor alt (3)

Verkapselung

▶ Hauptfunktion:

- ▶ Eingabe muß **vollständig** parsiert werden
- ▶ Auf **Mehrdeutigkeit** prüfen

```
parse :: Parse a b → [a] → Either String b
parse p i =
  case filter (null . snd) $ p i of
    []      → Left "Input does not parse"
    [(e, _)] → Right e
    _       → Left "Input is ambiguous"
```

▶ Schnittstelle:

- ▶ Nach außen nur Typ `Parse` sichtbar, plus **Operationen** darauf

Grammatik für Arithmetische Ausdrücke

$Expr ::= Term + Term \mid Term$

$Term ::= Factor * Factor \mid Factor$

$Factor ::= Variable \mid (Expr)$

$Variable ::= Char^+$

$Char ::= a \mid \dots \mid z \mid A \mid \dots \mid Z$

Abstrakte Syntax für Arithmetische Ausdrücke

- ▶ Zur Grammatik **abstrakte Syntax**

```
data Expr    = Plus  Expr Expr
              | Times Expr Expr
              | Var   String
```

- ▶ Hier Unterscheidung Term, Factor, Number unnötig.

Parsierung Arithmetischer Ausdrücke

- ▶ Token: Char
- ▶ Parsierung von Factor

```
pFactor :: Parse Char Expr
pFactor = some (spot isAlpha) 'use' Var
         'alt' token '(' *> pExpr >* token ')'
```

- ▶ Parsierung von Term

```
pTerm :: Parse Char Expr
pTerm =
  pFactor >* token '*' >*> pFactor 'use2' Times
  'alt' pFactor
```

- ▶ Parsierung von Expr

```
pExpr :: Parse Char Expr
pExpr = pTerm >* token '+' >*> pTerm 'use2' Plus
       'alt' pTerm
```

Die Hauptfunktion

- ▶ Lexing: Leerzeichen aus der Eingabe entfernen

```
parseExpr :: String → Expr
parseExpr i =
  case parse pExpr (filter (not.isSpace) i) of
    Right e → e
    Left err → error err
```

Ein kleiner Fehler

- ▶ **Mangel:** $a+b+c$ führt zu **Syntaxfehler** — Fehler in der **Grammatik**

- ▶ Behebung: **Änderung** der Grammatik

$$Expr ::= Term + Expr \mid Term$$
$$Term ::= Factor * Term \mid Factor$$
$$Factor ::= Variable \mid (Expr)$$
$$Variable ::= Char^+$$
$$Char ::= a \mid \dots \mid z \mid A \mid \dots \mid Z$$

- ▶ **Abstrakte Syntax** bleibt

Änderung des Parsers

- ▶ Entsprechende Änderung des Parsers in pTerm

```
pTerm :: Parse Char Expr
pTerm =
  pFactor >* token '*' >*> pTerm 'use2' Times
  'alt' pFactor
```

- ▶ ... und in pExpr:

```
pExpr :: Parse Char Expr
pExpr = pTerm >* token '+' >*> pExpr 'use2' Plus
  'alt' pTerm
```

- ▶ pFactor und Hauptfunktion bleiben.

Zusammenfassung Parserkombinatoren

- ▶ **Systematische Konstruktion** des Parsers aus der Grammatik.
- ▶ **Kompositional:**
 - ▶ Lokale Änderung der Grammatik führt zu lokaler Änderung im Parser
 - ▶ Vgl. Parsergeneratoren (yacc/bison, antlr, happy)
- ▶ Struktur von Parse zur Benutzung irrelevant
 - ▶ Vorsicht bei **Mehrdeutigkeiten** in der Grammatik (Performance-Falle)
 - ▶ **Einfache Implementierung** (wie oben) skaliert **nicht**
 - ▶ Effiziente Implementation mit **gleicher Schnittstelle** auch für **große Eingaben** geeignet.

Beispiel #2: Die Haskell Graphics Library HGL

- ▶ **Kompakte Grafikbücherei** für einfache Grafiken und Animationen.
- ▶ **Gleiche Schnittstelle** zu X Windows (X11) und Microsoft Windows.
- ▶ Bietet:
 - ▶ Fenster
 - ▶ verschiedene Zeichenfunktionen
 - ▶ Unterstützung für Animation

Übersicht HGL

- ▶ Grafiken

```
type Graphic
```

- ▶ Atomare Grafiken:

- ▶ Ellipsen, Linien, Polygone, ...

- ▶ Modifikation mit Attributen:

- ▶ Pinsel, Stifte und Textfarben

- ▶ Farben

- ▶ Kombination von Grafiken

- ▶ Überlagerung

Basisdatentypen

- ▶ Winkel (Grad, nicht Bogenmaß)

```
type Angle      = Double
```

- ▶ Dimensionen (Pixel)

```
type Dimension = Int
```

- ▶ Punkte (Ursprung: links oben)

```
type Point      = (Dimension , Dimension)
```

Atomare Grafiken (1)

- ▶ **Ellipse** (gefüllt) innerhalb des gegebenen Rechtecks

```
ellipse :: Point → Point → Graphic
```

- ▶ **Ellipse** (gefüllt) innerhalb des Parallelograms:

```
shearEllipse :: Point → Point → Point → Graphic
```

- ▶ **Bogenabschnitt** einer Ellipse (**math. positiven Drehsinn**):

```
arc :: Point → Point → Angle → Angle → Graphic
```

Atomare Grafiken (2)

- ▶ **Strecke, Streckenzug:**

```
line      :: Point → Point → Graphic  
polyline :: [Point] → Graphic
```

- ▶ **Polygon (gefüllt)**

```
polygon :: [Point] → Graphic
```

- ▶ **Text:**

```
text :: Point → String → Graphic
```

- ▶ **Leere Grafik:**

```
emptyGraphic :: Graphic
```

Modifikation von Grafiken

- ▶ Andere **Fonts**, **Farben**, Hintergrundfarben, ...:

<code>withFont</code>	::	<code>Font</code>	→	<code>Graphic</code>	→	<code>Graphic</code>
<code>withTextColor</code>	::	<code>RGB</code>	→	<code>Graphic</code>	→	<code>Graphic</code>
<code>withBkColor</code>	::	<code>RGB</code>	→	<code>Graphic</code>	→	<code>Graphic</code>
<code>withBkMode</code>	::	<code>BkMode</code>	→	<code>Graphic</code>	→	<code>Graphic</code>
<code>withPen</code>	::	<code>Pen</code>	→	<code>Graphic</code>	→	<code>Graphic</code>
<code>withBrush</code>	::	<code>Brush</code>	→	<code>Graphic</code>	→	<code>Graphic</code>
<code>withRGB</code>	::	<code>RGB</code>	→	<code>Graphic</code>	→	<code>Graphic</code>
<code>withTextAlignment</code>	::	<code>Alignment</code>	→	<code>Graphic</code>	→	<code>Graphic</code>

Farben

- ▶ Nützliche Abkürzung: benannte Farben

```
data Color = Black | Blue | Green | Cyan | Red  
          | Magenta | Yellow | White  
deriving (Eq, Ord, Bounded, Enum, Ix, Show, Read)
```

- ▶ Benannte Farben sind einfach `colorTable :: Array Color RGB`
- ▶ Dazu Modifikator:

```
withColor :: Color → Graphic → Graphic  
withColor c = withRGB (colorTable ! c)
```

Kombination von Grafiken

- ▶ Überlagerung (erste über zweiter):

```
overGraphic :: Graphic → Graphic → Graphic
```

- ▶ Verallgemeinerung:

```
overGraphics :: [Graphic] → Graphic  
overGraphics = foldr overGraphic emptyGraphic
```

Fenster

- ▶ **Elementare** Funktionen:

```
getGraphic  :: Window → IO Graphic
setGraphic  :: Window → Graphic → IO ()
```

- ▶ **Abgeleitete** Funktionen:

- ▶ In **Fenster** zeichnen:

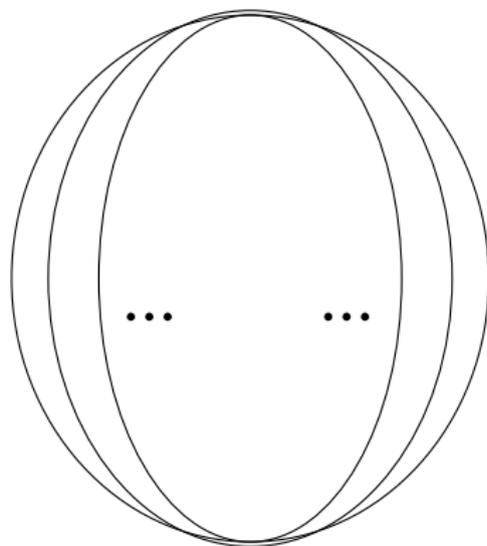
```
drawInWindow :: Window → Graphic → IO ()
drawInWindow w g = do
  old ← getGraphic w
  setGraphic w (g 'overGraphic' old)
```

- ▶ **Grafik löschen**

```
clearWindow :: Window → IO ()
clearWindow w = setGraphic w emptyGraphic
```

Ein einfaches Beispiel: der Ball

- ▶ **Ziel:** einen gestreiften Ball zeichnen
- ▶ **Algorithmus:** als Folge von konzentrischen Ellipsen
 - ▶ Start mit Eckpunkten (x_1, y_1) und (x_2, y_2) .
 - ▶ Verringerung von x um Δ_x , y bleibt gleich.
 - ▶ Dabei Farbe verändern.



Ein einfaches Beispiel: Der Ball

- ▶ Liste aller Farben cols
- ▶ Listen der x-Position (y-Position ist konstant), $\Delta_x = 25$
- ▶ Hilfsfunktion drawEllipse

```
drawBall :: Point → Point → Graphic
drawBall (x1, y1) (x2, y2) =
  let cols = cycle [Red, Green, Blue]
      midx = (x2 - x1) `div` 2
      xls  = [x1, x1 + 25 .. midx]
      xrs  = [x2, x2 - 25 .. midx]
      drawEllipse c xl xr = withColor c $
                              ellipse (xl, y1) (xr, y2)
      gs   = zipWith3 drawEllipse cols xls xrs
  in overGraphics (reverse gs)
```

Ein einfaches Beispiel: Der Ball

► Hauptprogramm ([Zeigen](#))

```
main :: IO ()
main = runGraphics $ do
  w ← openWindow "Balls!" (500,500)
  drawInWindow w $ drawBall (25, 25) (485, 485)
  getKey w
  closeWindow w
```

Animation

Alles dreht sich, alles bewegt sich...

- ▶ Animation: über der Zeit veränderliche Grafik
- ▶ Unterstützung von Animationen in HGL:
 - ▶ Timer ermöglichen getaktete Darstellung
 - ▶ Gepufferte Darstellung ermöglicht flickerfreie Darstellung
- ▶ Öffnen eines Fensters mit Animationsunterstützung:
 - ▶ Initiale Position, Grafikzwischenpuffer, Timer-Takt in Millisekunden

```
openWindowEx :: Title → Maybe Point → Size →  
              RedrawMode → Maybe Time → IO Window  
data RedrawMode  
    = Unbuffered | DoubleBuffered
```

Der springende Ball

- ▶ Ball hat Position und Geschwindigkeit:

```
data Ball = Ball { p :: Point ,  
                  v :: Point }
```

- ▶ Ball zeichnen: Roter Kreis an Position \vec{p}

```
drawBall :: Ball → Graphic  
drawBall (Ball {p= p}) =  
  withColor Red (circle p 15)
```

- ▶ Kreis zeichnen:

```
circle :: Point → Int → Graphic  
circle (px, py) r = ellipse (px- r, py- r) (px+ r, py+ r)
```

Bewegung des Balles

- ▶ Geschwindigkeit \vec{v} zu Position \vec{p} addieren
- ▶ In X-Richtung: modulo Fenstergröße 500
- ▶ In Y-Richtung: wenn Fensterrand 500 erreicht, Geschwindigkeit invertieren
- ▶ Geschwindigkeit in Y-Richtung nimmt immer um 1 ab

```
move (Ball {p= (px, py), v= (vx, vy)})=  
  Ball {p= (px', py'), v= (vx, vy')} where  
    px' = (px+ vx) 'mod' 500  
    py0 = py+ vy  
    py' = if py0> 500 then 500-(py0-500) else py0  
    vy' = (if py0> 500 then -vy else vy)+ 1
```

Der springende Ball

- ▶ Hauptschleife: Ball zeichnen, auf Tick warten, Folgeposition

```
loop w b =  
  do setGraphic w (drawBall b)  
     getWindowTick w  
     loop w (move b)
```

- ▶ **Hauptprogram**: Fenster öffnen, Starten der Hauptschleife

```
main = runGraphics $  
  do w ← openWindowEx "Bounce!" Nothing (500, 500)  
     DoubleBuffered (Just 30)  
     loop w (Ball{p=(0, 10), v=(5, 0)})
```

Zusammenfassung HGL

- ▶ Abstrakte und portable **Grafikprogrammierung**
- ▶ **Verkapselung** von **systemnaher** Schnittstelle durch Kombinatoren
- ▶ Kombinatoransatz: Kombination **elementarer** Grafiken zu komplexen Grafikprogrammen
- ▶ Rudimentäre Unterstützung von **Animation** durch Timer und Puffer
- ▶ Kombinatoransatz hier:

```
type Time = Int  
type Animation = Int → Graphic
```

Beispiel #3: tinySVG

- ▶ **Scalable Vector Graphics (SVG)**
 - ▶ XML-Standard für Vektorgrafiken
 - ▶ Unterstützt Vektorgrafiken (Pfade), Rastergrafiken und Text
 - ▶ Ein Kreis: `<circle x="20" y="30" r="50" />`
- ▶ Übersetzung in **Kombinatorbücherei** in Haskell (**tinySVG**):
 - ▶ **Elementare Operationen:**

```
circle :: Point → Double → Graphics
line   :: Point → Point → Graphics
```

- ▶ **Kombinatoren zum Transformieren:**

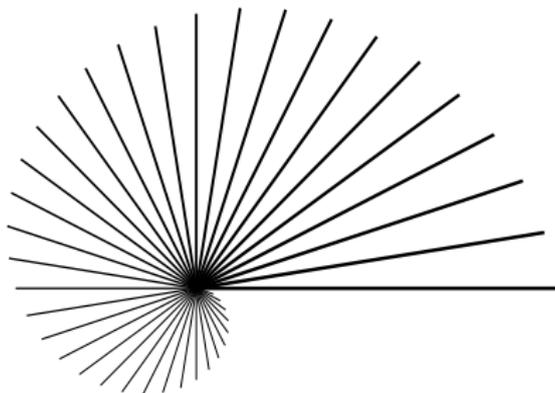
```
group :: [Graphics] → Graphics
rotate :: Double → Graphics → Graphics
scale  :: Double → Graphics → Graphics
```

- ▶ **Ausgabe:**

```
toXML :: Double → Double → Graphics → String
```

tinySVG in Aktion

```
snailShell :: Int → Graphics
snailShell n =
  let rs = [fromInt i / fromInt n | i ← [1..n]]
      theLine = line (pt' 0 0) (pt' 100 0)
      lines = [rotate (r*360) $ scale r theLine
              | r ← rs]
  in group lines
```



Zusammenfassung

- ▶ **Kombinatoransatz:**
 - ▶ Einfache Basisoperationen
 - ▶ Wenige Kombinationsoperatoren
 - ▶ Ideal in funktionalen Sprachen, generell nützlich
- ▶ **Parserkombinatoren:**
 - ▶ Von Grund auf in Haskell
 - ▶ Kombinatoren abstrahieren über Implementation
- ▶ **Grafik mit HGL:**
 - ▶ Verkapselung von Systemschnittstellen
 - ▶ Kombinatoren abstrahieren Systemschnittstellen
- ▶ **Grafik mit tinySVG:**
 - ▶ Kombinatoren abstrahieren über XML-Syntax

Praktische Informatik 3: Einführung in die Funktionale
Programmierung
Vorlesung vom 02.02.2011: The Next Big Thing — Scala

Christoph Lüth & Dennis Walter

Universität Bremen

Wintersemester 2010/11

Fahrplan

- ▶ Teil I: Funktionale Programmierung im Kleinen
- ▶ Teil II: Funktionale Programmierung im Großen
- ▶ Teil III: Funktionale Programmierung im richtigen Leben
 - ▶ Effizient Funktional Programmieren
 - ▶ Fallstudie: Kombinatoren
 - ▶ Eine Einführung in Scala
 - ▶ Rückblick & Ausblick

Scala

- ▶ A **scalable language**
- ▶ Multi-paradigm language: funktional + objektorientiert
- ▶ „Lebt im Java-Ökosystem“
- ▶ Martin Odersky, ETH Lausanne
- ▶ <http://www.scala-lang.org/>

Scala — Die Sprache

- ▶ Objekt-orientiert:
 - ▶ Veränderlicher, gekapselter **Zustand**
 - ▶ Klassen und Objekte
 - ▶ Subtypen und Vererbung
- ▶ Funktional:
 - ▶ Algebraische Datentypen
 - ▶ Unveränderliche **Werte**
 - ▶ Parametrische Polymorphie
 - ▶ Funktionen höherer Ordnung

Scala ist skalierbar

- ▶ „*A language that grows on you.*“
- ▶ Statt fertiger Sprache mit vielen Konstrukten **Rahmenwerk** zur Implementation eigener Konstrukte:
 - ▶ Einfache Basis
 - ▶ Flexible Syntax
 - ▶ Flexibles Typsystem
- ▶ Nachteil: Easy to learn but hard to master.
- ▶ Einfach zu benutzen:
 - ▶ Leichtgewichtig durch Typinferenz und Interpreter

Durchgängige Objektorientierung

- ▶ **Alles** in Scala ist ein Objekt
 - ▶ Keine primitiven Typen
- ▶ Operatoren sind Methoden
 - ▶ Beliebig überladbar
- ▶ Kein **static**, sondern Singleton-Objekte (**object**)
- ▶ Beispiel: `Rational.scala`

Werte

- ▶ Veränderliche **Variablen** **var**, unveränderliche **Werte** **val**
 - ▶ Zuweisung an **Werte** nicht erlaubt
- ▶ Dadurch unveränderliche Objekte \longrightarrow referentielle Transparenz
- ▶ “Unreine” Sprache
- ▶ **lazy val**: wird nach Bedarf ausgewertet.
- ▶ Sonderbehandlung von **Endrekursion** für bessere Effizienz
 - ▶ Damit **effiziente** funktionale Programmierung möglich
- ▶ Beispiel: `Gcd.scala`

Funktionale Aspekte

- ▶ Listen mit pattern matching
- ▶ Funktionen höherer Ordnung
- ▶ Listenkomprehension
- ▶ Beispiel: `Queens.scala`

Algebraische Datentypen

- ▶ Case Classes
 - ▶ Konzise Syntax für Konstruktoren: `factory methods`, kein `new`
 - ▶ Parameter werden zu `val`-Feldern
 - ▶ Pattern Match mit `Selektoren`
- ▶ Disjunkte Vereinigung durch Vererbung
- ▶ Beispiel: `Expr.scala`

Algebraische Datentypen und Vererbung

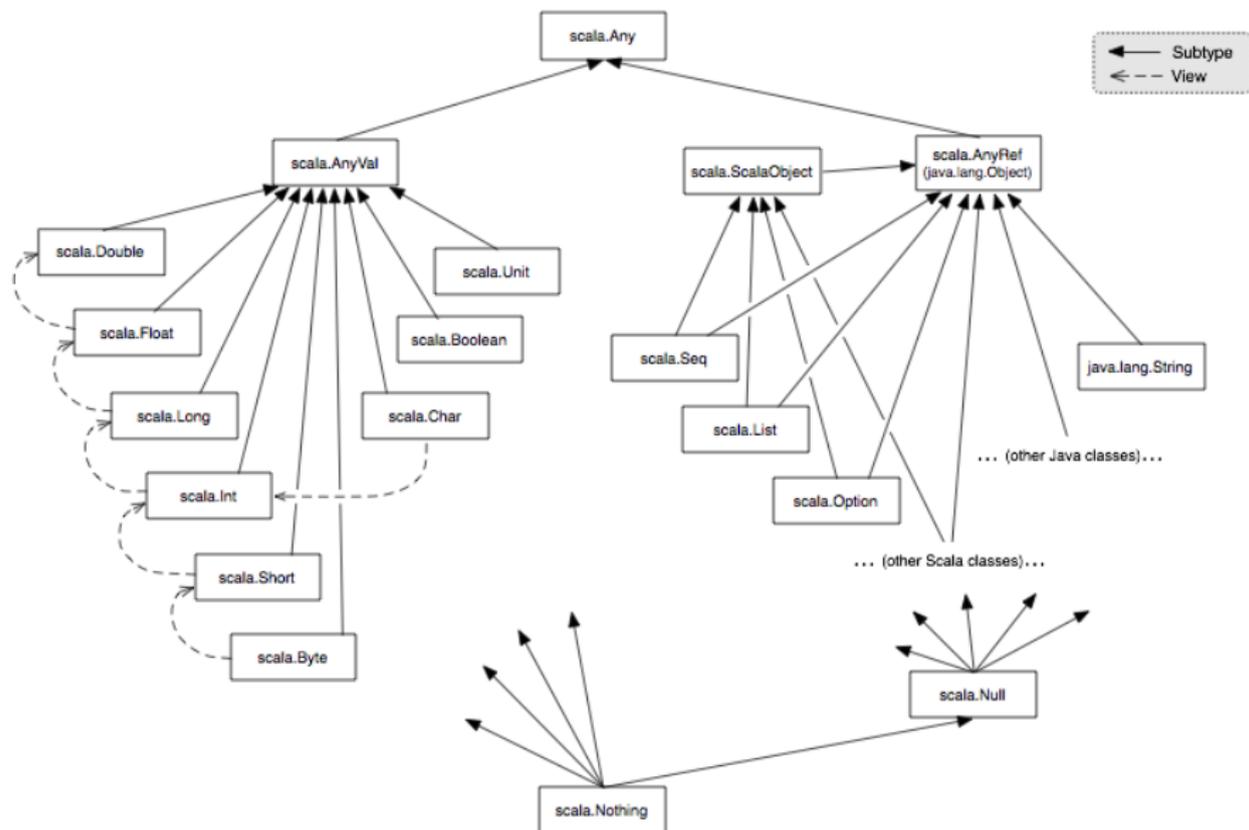
- ▶ Algebraische Datentypen können **erweitert** werden.
- ▶ Beispiel Expr:

```
case class UnOp(op: String, e: Expr) extends Expr
```

- ▶ Vorteil: **flexibel**
- ▶ Nachteil: Fehleranfällig
- ▶ Verbot der Erweiterung: **sealed classes**

```
sealed abstract class Expr  
  ...
```

Scala: Klassenhierarchie



Polymorphie und Subtypen

- ▶ Generische Typen (Scala) \cong Parametrische Polymorphie (Haskell)
 - ▶ In Scala besser als in Java wegen Typinferenz
- ▶ Problem mit generischen Typen und Polymorphie: **Varianz**
 - ▶ Gegeben `List [T]`
 - ▶ Ist `List [String] < List [AnyRef]`?
 - ▶ Gilt das immer?

Polymorphie und Subtypen

- ▶ Generische Typen (Scala) \cong Parametrische Polymorphie (Haskell)
 - ▶ In Scala besser als in Java wegen Typinferenz
- ▶ Problem mit generischen Typen und Polymorphie: **Varianz**
 - ▶ Gegeben `List [T]`
 - ▶ Ist `List [String] < List [AnyRef]`?
 - ▶ Gilt das immer? **Nein!**

Typvarianz

Gegeben folgende Klasse:

```
class Cell[T](init: T) {  
  private var curr = init  
  def get = curr  
  def set (x: T) = { curr = x }  
}
```

Problem: Ist `Cell[String] < Cell[Any]`?

```
val c1 = new Cell[String]("abc")  
val c2 : Cell[Any] = c1  
c2.set(1)  
val s : String = c1.get
```

Typvarianz

Gegeben folgende Klasse:

```
class Cell[T](init: T) {  
  private var curr = init  
  def get = curr  
  def set (x: T) = { curr = x }  
}
```

Problem: Ist `Cell[String] < Cell[Any]`?

```
val c1 = new Cell[String]("abc")  
val c2 : Cell[Any] = c1  
c2.set(1)  
val s : String = c1.get
```

Also: `Cell[String]` **kein** Untertyp von `Cell[Any]`

Java: das gleiche Problem

Gegeben:

```
String [] a1 = { "abc" };  
Object [] a2 = a1;  
a2[0]= new Integer(1);  
String s = a1[0];
```

Bei Ausführung **Laufzeitfehler**:

```
# javac Covariance.java
```

```
# java Covariance
```

```
Exception in thread "main" java.lang.ArrayStoreException: jav  
    at Covariance.main(Covariance.java:8)
```

Das Problem: Kontravarianz vs. Kovarianz

- ▶ Problem ist Typ von `set : T => Cell[T] => ()`
 - ▶ Nicht **var** und Zuweisung
- ▶ **Kovariant:**
 - ▶ Rechts des Funktionspfeils (Resultat)
 - ▶ Erhält Subtypenbeziehung
- ▶ **Kontravariant:**
 - ▶ Links des Funktionspfeils (Argument)
 - ▶ Kehrt Subtypenbeziehung
- ▶ **Position** der Typvariablen bestimmt **Varianz:**
 - ▶ Gegeben Mengen A, B, X mit $A \subseteq B$
 - ▶ Dann ist $X \rightarrow A \subseteq X \rightarrow B$
 - ▶ Aber **nicht** $A \rightarrow X \subseteq B \rightarrow X$
- ▶ Annotation der Varianz: `Set[+T]`, `Map[-T]`

Beschränkte Polymorphie

Gegeben Listen:

```
abstract class List[+T]  
case object Nil extends List[Nothing]  
case class :: [T](hd: T, tl: List[T]) extends List[T]
```

- ▶ Problem: An List [T] kann nur T gehängt werden
- ▶ Wünschenswert: beliebiger Untertyp von T
- ▶ Lösung: **bounded polymorphism**

```
case class :: [U >: T](hd: U, tl: List[T])  
                                extends List[T]
```

Traits

```
trait Ordered[A] {  
  def cmp(a: A): Int  
  
  def < (a: A): Boolean = (this cmp a) < 0  
  def > (a: A): Boolean = (this cmp a) > 0  
  def ≤ (a: A): Boolean = (this cmp a) ≤ 0  
  def ≥ (a: A): Boolean = (this cmp a) ≥ 0  
  def cmpTo(that: A): Int = cmp(that) }
```

```
class Rational extends Ordered[Rational] {  
  ...  
  def cmp(r: Rational) =  
    (this.numer * r.denom) - (r.numer * this.denom)
```

- ▶ Mächtiger als Interfaces (Java): kann Implementierung enthalten
- ▶ Mächtiger als abstrakte Klassen: Mehrfachvererbung
- ▶ Mächtiger als Typklassen (Haskell): mehr als ein Typ-Parameter

Weitere Besonderheiten: apply

- ▶ apply erlaubt Definition von Factory-Methoden und mehr:
- ▶ `f(i)` wird syntaktisch zu `f.apply(i)`
- ▶ Anderes Beispiel: Selektion aus `array`, Funktionen

Weitere Besonderheiten: Extraktoren

```
object EMail {  
  def apply(user: String, domain: String) =  
    user+ "@"+ domain  
  def unapply(str: String): Option[(String, String)] =  
  { val ps = str split "@"  
    if (ps.length == 2) Some(ps(0), ps(1)) else None  
  }  
}
```

- ▶ Extraktoren erlauben **erweitertes** pattern matching:

```
val s = EMail("cxl@dfki.de")  
s match { case EMail(user, domain) =>...}
```

- ▶ **Typgesteuertes** pattern matching:

```
val x : Any  
x match { case EMail(user, domain) =>...}
```

Weitere Besonderheiten

- ▶ Native XML support, Beispiel: `CCTherm.scala`
- ▶ Implizite Konversionen und Parameter

Zusammenfassung

- ▶ Haskell + Java = Scala (?)
- ▶ Skalierbare Sprache:
 - ▶ mächtige Basiskonstrukte
 - ▶ plus flexibles Typsystem
 - ▶ plus flexible Syntax (“syntaktischer Zucker”)
- ▶ Die Zukunft von Java?

Praktische Informatik 3: Einführung in die Funktionale
Programmierung
Vorlesung vom 09.02.11: Schlußbemerkungen

Christoph Lüth & Dennis Walter

Universität Bremen

Wintersemester 2010/11

Fahrplan

- ▶ Teil I: Funktionale Programmierung im Kleinen
- ▶ Teil II: Funktionale Programmierung im Großen
- ▶ Teil III: Funktionale Programmierung im richtigen Leben
 - ▶ Effizient Funktional Programmieren
 - ▶ Fallstudie: Kombinatoren
 - ▶ Eine Einführung in Scala
 - ▶ Rückblick & Ausblick

Organisatorisches

- ▶ Ausgefüllten **Scheinvordruck** zum Fachgespräch **mitbringen**
- ▶ Nur wer ausgefüllten Scheinvordruck abgibt, erhält auch einen.
- ▶ Evaluationsbogen ausfüllen

Beispiel

Beispiel

Definieren Sie eine Funktion `leastSpaces`, welche aus einer Liste von Zeichenketten diejenige zurückgibt, welche die wenigsten Leerzeichen enthält.

Inhalt

- ▶ Wiederholung

- ▶ Rückblick, Ausblick

Vorlesung vom 27.10.10: Grundbegriffe

- ▶ Was sind die Bestandteile einer Funktionsdefinition?

Vorlesung vom 27.10.10: Grundbegriffe

- ▶ Was sind die Bestandteile einer Funktionsdefinition?
- ▶ Was bedeutet referentielle Transparenz?

Vorlesung vom 27.10.10: Grundbegriffe

- ▶ Was sind die Bestandteile einer Funktionsdefinition?
- ▶ Was bedeutet referentielle Transparenz?
- ▶ Was ist eigentlich syntaktischer Zucker? (Beispiel?)

Vorlesung vom 03.11.10: Funktionen und Datentypen

- ▶ Welche **Auswertungsstrategien** gibt es, welche benutzt Haskell?

Vorlesung vom 03.11.10: Funktionen und Datentypen

- ▶ Welche **Auswertungsstrategien** gibt es, welche benutzt Haskell?
- ▶ Was bedeutet **Striktheit**?

Vorlesung vom 03.11.10: Funktionen und Datentypen

- ▶ Welche **Auswertungsstrategien** gibt es, welche benutzt Haskell?
- ▶ Was bedeutet **Striktheit**?
- ▶ Was ist die **Abseitsregel**?

Vorlesung vom 03.11.10: Funktionen und Datentypen

- ▶ Welche **Auswertungsstrategien** gibt es, welche benutzt Haskell?
- ▶ Was bedeutet **Striktheit**?
- ▶ Was ist die **Abseitsregel**?
- ▶ Welche vordefinierten **Basisdatentypen** gibt es in Haskell?

Vorlesung vom 10.11.10: Rekursive Datentypen

- ▶ Welches sind die **wesentlichen** Eigenschaften der **Konstruktoren** eines algebraischen Datentyps?

Vorlesung vom 10.11.10: Rekursive Datentypen

- ▶ Welches sind die **wesentlichen** Eigenschaften der **Konstruktoren** eines algebraischen Datentyps?
- ▶ Was ist das:

```
data X a = X a [X a]
```

Vorlesung vom 10.11.10: Rekursive Datentypen

- ▶ Welches sind die **wesentlichen** Eigenschaften der **Konstruktoren** eines algebraischen Datentyps?
- ▶ Was ist das:

```
data X a = X a [X a]
```

- ▶ Was bedeutet **strukturelle Induktion**?

Vorlesung vom 10.11.10: Rekursive Datentypen

- ▶ Welches sind die **wesentlichen** Eigenschaften der **Konstruktoren** eines algebraischen Datentyps?
- ▶ Was ist das:

```
data X a = X a [X a]
```

- ▶ Was bedeutet **strukturelle Induktion**?
- ▶ Wie beweist man folgende Behauptung:

```
map f (map g xs) = map (f . g) xs
```

Vorlesung vom 17.11.10: Typvariablen und Polymorphie

- ▶ Was ist **parametrische Polymorphie** in funktionalen Sprachen?

Vorlesung vom 17.11.10: Typvariablen und Polymorphie

- ▶ Was ist **parametrische Polymorphie** in funktionalen Sprachen?
- ▶ Wo ist der **Unterschied** zu Polymorphie Java, und was entspricht der parametrischen Polymorphie in Java?

Vorlesung vom 17.11.10: Typvariablen und Polymorphie

- ▶ Was ist **parametrische Polymorphie** in funktionalen Sprachen?
- ▶ Wo ist der **Unterschied** zu Polymorphie Java, und was entspricht der parametrischen Polymorphie in Java?
- ▶ Welche **Standarddatentypen** gibt es in Haskell?

Vorlesung vom 17.11.10: Typvariablen und Polymorphie

- ▶ Was ist **parametrische Polymorphie** in funktionalen Sprachen?
- ▶ Wo ist der **Unterschied** zu Polymorphie Java, und was entspricht der parametrischen Polymorphie in Java?
- ▶ Welche **Standarddatentypen** gibt es in Haskell?
- ▶ Wozu dienen **Typklassen** in Haskell?

Vorlesung vom 24.11.10: Funktionen höherer Ordnung

- ▶ Was ist eine Funktion höher Ordnung?

Vorlesung vom 24.11.10: Funktionen höherer Ordnung

- ▶ Was ist eine Funktion höher Ordnung?
- ▶ Was ist **einfache Rekursion**?

Vorlesung vom 24.11.10: Funktionen höherer Ordnung

- ▶ Was ist eine Funktion höher Ordnung?
- ▶ Was ist **einfache Rekursion**?
- ▶ Was ist **Listenkompensation**?

Vorlesung vom 24.11.10: Funktionen höherer Ordnung

- ▶ Was ist eine Funktion höher Ordnung?
- ▶ Was ist **einfache Rekursion**?
- ▶ Was ist **Listenkompensation**?
- ▶ Wie läßt sich Listenkompensation durch `map` und `filter` darstellen?

Vorlesung vom 24.11.10: Funktionen höherer Ordnung

- ▶ Was ist eine Funktion höherer Ordnung?
- ▶ Was ist **einfache Rekursion**?
- ▶ Was ist **Listenkompensation**?
- ▶ Wie läßt sich Listenkompensation durch `map` und `filter` darstellen?
- ▶ Was ist der Unterschied zwischen `foldr` und `foldl`?

Vorlesung vom 24.11.10: Funktionen höherer Ordnung

- ▶ Was ist eine Funktion höher Ordnung?
- ▶ Was ist **einfache Rekursion**?
- ▶ Was ist **Listenkompensation**?
- ▶ Wie läßt sich Listenkompensation durch `map` und `filter` darstellen?
- ▶ Was ist der Unterschied zwischen `foldr` und `foldl`?
- ▶ ... und wann benutzt man welches?

Vorlesung vom 01.12.10: Typinferenz

- ▶ Woran kann Typableitung **scheitern**?

Vorlesung vom 01.12.10: Typinferenz

- ▶ Woran kann Typableitung **scheitern**?
- ▶ Was ist der Typ von $\lambda x y \rightarrow (x, 3) : [(\text{"_"}, y)]$

Vorlesung vom 01.12.10: Typinferenz

- ▶ Woran kann Typableitung **scheitern**?
- ▶ Was ist der Typ von $\lambda x y \rightarrow (x, 3) : [(\text{"␣"}, y)]$
- ▶ ... und warum?

Vorlesung vom 01.12.10: Typinferenz

- ▶ Woran kann Typableitung **scheitern**?
- ▶ Was ist der Typ von $\lambda x y \rightarrow (x, 3) : [(\text{"_"}, y)]$
- ▶ ... und warum?
- ▶ Was ist ein Beispiel für einen Ausdruck vom Typ $[[[a], \text{Int}]]$?

Vorlesung vom 08.12.2010: ADTs

- ▶ Was ist ein **abstrakter Datentyp**?

Vorlesung vom 08.12.2010: ADTs

- ▶ Was ist ein **abstrakter Datentyp**?
- ▶ Wieso sind geordnete Bäume ein abstrakter und kein algebraischer Datentyp?

Vorlesung vom 08.12.2010: ADTs

- ▶ Was ist ein **abstrakter Datentyp**?
- ▶ Wieso sind geordnete Bäume ein abstrakter und kein algebraischer Datentyp?
- ▶ Wieso sind Listen ein algebraischer und kein abstrakter Datentyp?

Vorlesung vom 08.12.2010: ADTs

- ▶ Was ist ein **abstrakter Datentyp**?
- ▶ Wieso sind geordnete Bäume ein abstrakter und kein algebraischer Datentyp?
- ▶ Wieso sind Listen ein algebraischer und kein abstrakter Datentyp?
- ▶ Haben abstrakte Datentypen einen verkapselten **Zustand**?

Vorlesung vom 08.12.2010: ADTs

- ▶ Was ist ein **abstrakter Datentyp**?
- ▶ Wieso sind geordnete Bäume ein abstrakter und kein algebraischer Datentyp?
- ▶ Wieso sind Listen ein algebraischer und kein abstrakter Datentyp?
- ▶ Haben abstrakte Datentypen einen verkapselten **Zustand**?
- ▶ Was ist ein **Modul** in Haskell, und was sind seine **Bestandteile**?

Vorlesung vom 05.01.11: Signaturen und Eigenschaften

- ▶ Was ist eine **Signatur**?

Vorlesung vom 05.01.11: Signaturen und Eigenschaften

- ▶ Was ist eine **Signatur**?
- ▶ Was sind **Axiome**?

Vorlesung vom 05.01.11: Signaturen und Eigenschaften

- ▶ Was ist eine **Signatur**?
- ▶ Was sind **Axiome**?
- ▶ Was wäre ein ADT für **Array a** — welche Operationen, welche Eigenschaften?

Vorlesung vom 12.01.11: Aktionen und Zustände

- ▶ Was unterscheidet Aktionen (IO a) von anderen ADTs?

Vorlesung vom 12.01.11: Aktionen und Zustände

- ▶ Was unterscheidet Aktionen (IO a) von anderen ADTs?
- ▶ Was sind die Operationen des ADT IO a?

Vorlesung vom 12.01.11: Aktionen und Zustände

- ▶ Was unterscheidet Aktionen (IO a) von anderen ADTs?
- ▶ Was sind die Operationen des ADT IO a?
- ▶ Wozu dient `return`?

Vorlesung vom 12.01.11: Aktionen und Zustände

- ▶ Was unterscheidet Aktionen (IO a) von anderen ADTs?
- ▶ Was sind die Operationen des ADT IO a?
- ▶ Wozu dient `return`?
- ▶ Welche **Eigenschaften** haben die Operationen?

Vorlesung vom 12.01.11: Aktionen und Zustände

- ▶ Was unterscheidet Aktionen (IO a) von anderen ADTs?
- ▶ Was sind die Operationen des ADT IO a?
- ▶ Wozu dient `return`?
- ▶ Welche **Eigenschaften** haben die Operationen?
- ▶ Wie kann man...

Vorlesung vom 12.01.11: Aktionen und Zustände

- ▶ Was unterscheidet Aktionen (IO a) von anderen ADTs?
- ▶ Was sind die Operationen des ADT IO a?
- ▶ Wozu dient `return`?
- ▶ Welche **Eigenschaften** haben die Operationen?
- ▶ Wie kann man...
 - ▶ ... aus einer Datei lesen?

Vorlesung vom 12.01.11: Aktionen und Zustände

- ▶ Was unterscheidet Aktionen (IO a) von anderen ADTs?
- ▶ Was sind die Operationen des ADT IO a?
- ▶ Wozu dient `return`?
- ▶ Welche **Eigenschaften** haben die Operationen?
- ▶ Wie kann man...
 - ▶ ... aus einer Datei lesen?
 - ▶ ... die Kommandozeilenargumente lesen?

Vorlesung vom 12.01.11: Aktionen und Zustände

- ▶ Was unterscheidet Aktionen (IO a) von anderen ADTs?
- ▶ Was sind die Operationen des ADT IO a?
- ▶ Wozu dient `return`?
- ▶ Welche **Eigenschaften** haben die Operationen?
- ▶ Wie kann man...
 - ▶ ... aus einer Datei lesen?
 - ▶ ... die Kommandozeilenargumente lesen?
 - ▶ ... eine Zeichenkette ausgeben?

Vorlesung vom 19.01.11: Effizienzaspekte

- ▶ Was ist Endrekursion?

Vorlesung vom 19.01.11: Effizienz Aspekte

- ▶ Was ist **Endrekursion**?
- ▶ Was ist ein **Speicherleck** in Haskell?

Vorlesung vom 19.01.11: Effizienzaspekte

- ▶ Was ist **Endrekursion**?
- ▶ Was ist ein **Speicherleck** in Haskell?
- ▶ Wie vermeide ich Speicherlecks?

Zusammenfassung Haskell

Stärken:

- ▶ Abstraktion durch
 - ▶ Polymorphie und Typsystem
 - ▶ algebraische Datentypen
 - ▶ Funktionen höherer Ordnung
- ▶ Flexible Syntax
- ▶ Haskell als Meta-Sprache
- ▶ Ausgereifter Compiler
- ▶ Viele Büchereien

Schwächen:

- ▶ Komplexität
- ▶ Dokumentation
 - ▶ z.B. im Vergleich zu Java's APIs
- ▶ Büchereien
- ▶ Noch viel im Fluß
 - ▶ Tools ändern sich
 - ▶ Zum Beispiel HGL
- ▶ Entwicklungsumgebungen

Andere Funktionale Sprachen

▶ Standard ML (SML):

- ▶ Streng typisiert, strikte Auswertung
- ▶ Formal definierte Semantik
- ▶ Drei aktiv unterstützte Compiler
- ▶ Verwendet in Theorembeweisern (Isabelle, HOL)
- ▶ <http://www.standardml.org/>

▶ Caml, O'Caml:

- ▶ Streng typisiert, strikte Auswertung
- ▶ Hocheffizienter Compiler, byte code & nativ
- ▶ Nur ein Compiler (O'Caml)
- ▶ <http://caml.inria.fr/>

Andere Funktionale Sprachen

- ▶ LISP & Scheme
 - ▶ Ungetypt/schwach getypt
 - ▶ Seiteneffekte
 - ▶ Viele effiziente Compiler, aber viele Dialekte
 - ▶ Auch industriell verwendet

Funktionale Programmierung in der Industrie

▶ Erlang

- ▶ schwach typisiert, nebenläufig, strikt
- ▶ Fa. Ericsson — Telekom-Anwendungen

▶ FL

- ▶ ML-artige Sprache
- ▶ Chip-Verifikation der Fa. Intel

▶ Galois Connections

- ▶ Hochqualitätssoftware in Haskell
- ▶ Hochsicherheitswebserver, Cryptoalgorithmen

▶ Verschiedene andere Gruppen

Perspektiven

- ▶ Funktionale Programmierung in 10 Jahren?
- ▶ **Anwendungen:**
 - ▶ Integration von XML, DBS (X#/Xen, Microsoft)
 - ▶ Integration in Rahmenwerke (F# & .Net, Microsoft)
 - ▶ Eingebettete domänenspezifische Sprachen
- ▶ **Forschung:**
 - ▶ Ausdrucksstärkere Typsysteme
 - ▶ für effiziente Implementierungen
 - ▶ und eingebaute Korrektheit (Typ als Spezifikation)
 - ▶ Parallelität?

Warum funktionale Programmierung nie Erfolg haben wird

- ▶ Programmierung nur kleiner Teil der SW-Entwicklung
- ▶ Mangelnde Unterstützung:
 - ▶ Libraries, Dokumentation, Entwicklungsumgebungen
- ▶ Nicht verbreitet — funktionale Programmierer zu teuer
- ▶ Konservatives Management
 - ▶ “Nobody ever got fired for buying IBM”

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?

Hilfe!

- ▶ Haskell: primäre Entwicklungssprache am DFKI, FG SKS
 - ▶ Formale Programmentwicklung: <http://www.tzi.de/cofi/hets>
 - ▶ Sicherheit in der Robotik: <http://www.dfki.de/sks/sams>
- ▶ Wir suchen studentische Hilfskräfte für diese Projekte
- ▶ Wir bieten:
 - ▶ Angenehmes Arbeitsumfeld
 - ▶ Interessante Tätigkeit
- ▶ Wir suchen Tutoren für PI3
 - ▶ im WS 11/12 — meldet Euch bei Berthold Hoffmann!

Tschüß!

