

Praktische Informatik 3

Einführung in die Funktionale Programmierung

Christoph Lüth

WS 08/09



Vorlesung vom 29.10.2008: Einführung

Personal

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

Termine

- **Vorlesung:**

Mi 13 – 15, SFG 0140

- **Tutorien:**

Di	10 – 12	MZH 7210	Klaus Hartke
	17 – 19	MZH 1380	Marcus Ermler
Mi	8 – 10	MZH 7250	Ewaryst Schulz & Dominik Dietrich
Do	8 – 10	FZB 0240	Dominik Luecke
	10 – 12	Cart 0.01	Christian Maeder

- **Fragestunde (FAQ):**

Mi 10 – 12 Berthold Hoffmann (Cartesium 2.048)

Übungsbetrieb

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

Scheinkriterien — Vorschlag:

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

Spielregeln

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

Fahrplan

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

Warum funktionale Programmierung lernen?

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

Geschichtliches

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

Referentielle Transparenz

- Programme als Funktionen

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

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

Programmieren mit Funktionen

- Programme werden durch Gleichungen definiert:

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

- Auswertung durch Reduktion von Ausdrücken:

```
addDouble (inc 5) 4
```

Definition von Funktionen

- Zwei wesentliche Konstrukte:

- Fallunterscheidung
- Rekursion

- Beispiel:

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

- Auswertung kann divergieren!

Imperativ vs. Funktional

- **Imperative** Programmierung:

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

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

- **Funktionale** Programmierung:

- Funktionen $f : E \rightarrow A$

- Kontrollstrukturen: Fallunterscheidung
Rekursion

Nichtnumerische Werte

- Rechnen mit Zeichenketten

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

- Auswertung:

```
repeat 2 "hallo "  
~> "hallo" ++ repeat 1 "hallo"  
~> "hallo "++ ("hallo " ++ repeat 0 "hallo ")  
~> "hallo "++ ("hallo " ++ "")  
~> "hallo "++ "hallo "  
~> "hallo hallo "
```

Typisierung

- **Typen** unterscheiden Arten von Ausdrücken:

repeat	n	s	=	...	n	Zahl
		s			s	Zeichenkette

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

Signaturen

- Jede Funktion hat eine **Signatur**

`fac :: Int -> Int`

`repeat :: Int -> String -> String`

- **Typüberprüfung**
 - `fac` nur auf `Int` anwendbar, Resultat ist `Int`
 - `repeat` nur auf `Int` und `String` anwendbar, Resultat ist `String`

Übersicht: Typen in Haskell

Ganze Zahlen	<code>Int</code>	<code>0 94 -45</code>
Fließkomma	<code>Double</code>	<code>3.0 3.141592</code>
Zeichen	<code>Char</code>	<code>'a' 'x' '\034' '\n'</code>
Zeichenketten	<code>String</code>	<code>"yuck" "hi\nho"\n"</code>
Wahrheitswerte	<code>Bool</code>	<code>True False</code>
Listen	<code>[a]</code>	<code>[6, 9, 20]</code> <code>["oh", "dear"]</code>
Tupel	<code>(a, b)</code>	<code>(1, 'a') ('a', 4)</code>
Funktionen	<code>a -> b</code>	

Auswertungsstrategien

- Von **außen** nach **innen** (outermost-first):
`inc (addDouble (inc 3) 4)`

~→

Auswertungsstrategien

- Outermost-first entspricht **call-by-need**, **verzögerte** Auswertung.
- Innermost-first entspricht **call-by-value**, **strikte** Auswertung
- Beispiel:

```
div :: Int-> Int-> Int
```

Ganzzahlige Division, undefiniert für `div n 0`

```
mult :: Int-> Int-> Int
```

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

- Auswertung von `mult 0 (div 1 0)`

Striktheit

Def: Funktion f ist **strikt** gdw.

Ergebnis ist undefiniert sobald ein Argument undefiniert ist

- Standard ML, Java, C etc. sind **strikt**
- Haskell ist **nicht-strikt**
- Fallunterscheidung ist **immer** nicht-strikt

Zusammenfassung

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

Vorlesung vom 05.11.2008: Funktionen und Datentypen

Organisatorisches

- **Tutorien:** Ungleichverteilung

Di 10– 12: 42

Mi 8 – 10: 32

Do 10– 12: 26

Di 17– 19: 18

Do 8– 10: 10

- **Übungsblätter:**

- Lösungen in $\text{\LaTeX}$ (siehe Webseite)

Fahrplan

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

- Definition von **Funktionen**
 - Syntaktische Feinheiten
- Definition von **Datentypen**
 - Aufzählungen
 - Produkte
 - Rekursive Datentypen
- **Basisdatentypen:**
 - Wahrheitswerte
 - numerische Typen
 - alphanumerische Typen

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**)?

Die Abseitsregel

Funktionsdefinition:

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

- **Geltungsbereich** der Definition von f :
alles, was gegenüber f **eingerrückt** 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**

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.

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 = ...
```

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 und Funktionen

- Datentypen konstruieren Werte
- Funktionen sind Berechnungen
- Konstruktion für Datentypen $\longleftrightarrow$ Definition von Funktionen

Aufzählungen

- Aufzählungen: Menge von **disjunkten** Konstanten

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

$Mon \neq Tue, Mon \neq Wed, Tue \neq Thu, Wed \neq Sun \dots$

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

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

Aufzählung und Fallunterscheidung in Haskell

- Definition

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

- Implizite Deklaration der Konstanten `Mon :: Days`

- 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**):

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

- 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:

<code>not :: Bool -> Bool</code>	Negation
<code>&& :: Bool -> Bool -> Bool</code>	Konjunktion
<code> :: Bool -> Bool -> Bool</code>	Disjunktion

- Konjunktion definiert wie

```
a && b = case a of True -> b  
                False -> False
```

- `&&`, `||` sind rechts nicht strikt

- `False && div 1 0 == 0 ~> False`

- `if then else` als syntaktischer Zucker:

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

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)

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

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

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

- 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
```

Der Allgemeine Fall: Algebraische Datentypen

Definition eines **algebraischen Datentypen** T :

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

- Konstruktoren $C_1, \dots, C_n$ sind **disjunkt**:

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

- Konstruktoren sind **injektiv**:

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

- 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.
- Entspricht **induktiver Definition**
- Rekursive Datentypen sind **unendlich**
- Beispiel **natürliche Zahlen**: Peano-Axiome
 - $0 \in \mathbb{N}$
 - wenn $n \in \mathbb{N}$, dann $Sn \in \mathbb{N}$
 - S injektiv und $Sn \neq 0$
 - Induktionsprinzip — entspricht rekursiver Funktionsdefinition
- **Induktionsprinzip** erlaubt Definition **rekursiver Funktionen**:

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

Natürliche Zahlen in Haskell

- Der Datentyp

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

- Funktionen auf **rekursiven** Typen oft **rekursiv** definiert:

```
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** und eine weitere **Zeichenkette**

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

- Was ist **ungünstig** an dieser Repräsentation:

```
data MyString' = Empty'  
               | Single Char  
               | Concat MyString' MyString'
```

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)
```

Rekursive Typen in anderen Sprachen

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

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

- **Java**: keine Entsprechung
 - Nachbildung durch Klassen, z.B. für Listen:

```
class List {  
    public List(Object theElement, ListNode n) {  
        element = theElement;  
        next    = n;  }  
    public Object element;  
    public List  next;  }
```

- **C**: Produkte, Aufzählungen, keine rekursiven Typen

Das Rechnen mit Zahlen

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

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 `==`, `/=`, `<=`, `<`, ...
- **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: **Listen** von Zeichen $\rightsquigarrow$ nächste Vorlesung

Zusammenfassung

- Funktionsdefinitionen:
 - Abseitsregel, bedingte Definition
 - Lokale Definitionen
- Datentypen und Funktionsdefinition dual
 - Aufzählungen — Fallunterscheidung
 - Produkte
 - Rekursive Typen — rekursive Funktionen
- Wahrheitswerte Bool
- Numerische Basisdatentypen:
 - Int, Integer, Rational und Double
- Alphanumerische Basisdatentypen: Char
- Nächste Vorlesung: Abstraktion über Typen

Vorlesung vom 12.11.2008: Typvariablen und Polymorphie

Fahrplan

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

Inhalt

- Letzte Vorlesung: rekursive Datentypen
- Diese Vorlesung:
 - **Abstraktion** über Typen: **Typvariablen** und **Polymorphie**
 - Typinferenz: Wie **bestimme** ich den **Typ** eines **Ausdrucks**?

Letzte Vorlesung: Zeichenketten

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

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

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)
```

Weiteres Beispiel: Liste von Zahlen

- Eine **Liste von Zahlen** ist
 - entweder **leer** (das leere Wort ϵ)
 - oder eine **Zahl** und eine weitere **Liste**

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

Funktionen auf Zahlenlisten

- Länge:

```
len :: IntList-> Int
len Empty          = 0
len (Cons c str) = 1+ len str
```

- Verkettung:

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

- Umkehrung:

```
rev :: IntList-> IntList
rev Empty          = Empty
rev (Cons c t) = cat (rev t) (Cons c Empty)
```

Typvariablen

- **Typvariablen** abstrahieren über Typen

```
data List a = Empty  
            | Cons a (List a)
```

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

```
Empty :: List a  
Cons  :: a -> List a -> List a
```

Polymorphe Datentypen

- **Typkorrekte** Terme:

Typ

Empty

List a

Cons 57 Empty

List Int

Cons 7 (Cons 8 Empty)

List Int

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

List Char

Cons True Empty

List Bool

- Nicht typ-korrekt:

Cons 'a' (Cons 0 Empty)

Cons True (Cons 'x' Empty)

wegen **Signatur** des Konstruktors:

Cons :: a -> List a -> List a

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 erlaubt **Parametrisierung über Typen**:

```
cat :: List a -> List a -> List a
cat Empty ys          = ys
cat (Cons x xs) ys = Cons x (cat xs ys)
```

- **Typvariable** *a* 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

Tupel

- Mehr als **eine** Typvariable:

```
data Pair a b = Pair a b
```

- Konstruktorname = Typname
- Beispielterme:

```
Pair 4 "fünf"
```

```
Pair (Cons True Empty) 'a'
```

```
Pair (3+ 4) (Cons 'a' Empty)
```

Typinferenz

- Bestimmung des Typen durch Typinferenz
- Formalismus: Typableitungen der Form

$$A \vdash x :: t$$

- A — Typumgebung (Zuordnung Symbole zu Typen)
- x — Term
- t — Typ
- Herleitung durch fünf Basisregeln
 - Notation: $t \left[\begin{smallmatrix} s \\ x \end{smallmatrix} \right]$ x in t durch s ersetzt
 - Lambda-Abstraktion: $f = \lambda x \rightarrow E$ für $f \ x = E$

Typinferenzregeln

$$\frac{}{A, x :: t \vdash x :: t} Ax$$

$$\frac{A, x :: s \vdash e :: t}{A \vdash \lambda x \rightarrow e :: s \rightarrow t} Abs$$

$$\frac{A \vdash e :: s \rightarrow t \quad A \vdash e' :: s}{A \vdash e e' :: t} App$$

$$\frac{A \vdash e :: t, \text{Typvariable } \alpha \text{ nicht frei in } A}{A \vdash e :: t \left[\begin{smallmatrix} s \\ \alpha \end{smallmatrix} \right]} Spec$$

$$\frac{A \vdash f :: s \quad A \vdash c_i :: s \quad A \vdash e_i :: t}{A \vdash \text{case } f \text{ of } c_i \rightarrow e_i :: t} Cases$$

Polymorphie in anderen Programmiersprachen: Java

- Polymorphie in **Java**: Methode auf alle Subklassen anwendbar

```
class List {  
    public List(Object theElement, List n) {  
        element = theElement;  
        next    = n;  }  
    public Object element;  
    public List  next; }  

```

- Keine Typvariablen:

```
String s = "abc";  
List l   = new List(s, null);
```

- l.element hat Typ Object, nicht String

```
String e = (String)l.element;
```

- Neu ab Java 1.5: **Generics** — damit echte Polymorphie möglich

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`

Vordefinierte Datentypen: Tupel und Listen

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

`data (a, b) = (a, b)`

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

- **Listen**

`data [a] = [] | a : [a]`

- Weitere Abkürzungen: `[x] = x : []`, `[x,y] = x:y:[]` etc.

Übersicht: vordefinierte Funktionen auf Listen I

<code>++</code>	<code>[a] -> [a] -> [a]</code>	Verketten
<code>!!</code>	<code>[a] -> Int -> a</code>	<code>n</code> -tes Element selektieren
<code>concat</code>	<code>[[a]] -> [a]</code>	“flachklopfen”
<code>length</code>	<code>[a] -> Int</code>	Länge
<code>head, last</code>	<code>[a] -> a</code>	Erster/letztes Element
<code>tail, init</code>	<code>[a] -> [a]</code>	(Hinterer/vorderer) Rest
<code>replicate</code>	<code>Int -> a -> [a]</code>	Erzeuge <code>n</code> Kopien
<code>take</code>	<code>Int -> [a] -> [a]</code>	Nimmt ersten <code>n</code> Elemente
<code>drop</code>	<code>Int -> [a] -> [a]</code>	Entfernt erste <code>n</code> Elemente
<code>splitAt</code>	<code>Int -> [a] -> ([a], [a])</code>	Spaltet an <code>n</code> -ter Position
<code>reverse</code>	<code>[a] -> [a]</code>	Dreht Liste um
<code>zip</code>	<code>[a] -> [b] -> [(a, b)]</code>	Paare zu Liste von Paaren
<code>unzip</code>	<code>[(a, b)] -> ([a], [b])</code>	Liste von Paaren zu Paaren
<code>and, or</code>	<code>[Bool] -> Bool</code>	Konjunktion/Disjunktion
<code>sum</code>	<code>[Int] -> Int</code> (überladen)	Summe
<code>product</code>	<code>[Int] -> Int</code> (überladen)	Produkt

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:

```
count :: Char -> String -> Int
count c []      = 0
count c (x:xs) = if (c == x) then 1 + count c xs
                  else count c xs
```

Beispiel: Palindrome

- **Palindrom:** vorwärts und rückwärts gelesen gleich (z.B. Otto, Reliefpfeiler)

- Signatur:

`palindrom :: String -> Bool`

- Entwurf:
 - **Rekursive Formulierung:**
erster Buchstabe = letzter Buchstabe, und Rest auch Palindrom
 - **Termination:**
Leeres Wort und monoliterales Wort sind Palindrome
 - **Hilfsfunktionen:**
`last: String -> Char, init: String -> String`

Beispiel: Palindrome

- Implementierung:

```
palindrom :: String-> Bool
palindrom []      = True
palindrom [x]     = True
palindrom (x:xs) = (x == last xs)
                  && palindrom (init xs)
```

- Kritik:

- Unterschied zwischen Groß- und kleinschreibung

```
palindrom (x:xs) = (toLower x == toLower (last xs))
                  && palindrom (init xs)
```

- Nichtbuchstaben sollten nicht berücksichtigt werden.

Zusammenfassung

- Typvariablen und Polymorphie: Abstraktion über Typen
- Typinferenz (Hindley-Damas-Milner): Herleitung des Typen eines Ausdrucks
- Vordefinierte Typen: Listen $[a]$ und Tupel (a,b)
- Nächste Woche: Funktionen höherer Ordnung

Vorlesung vom 19.11.2008: Funktionen höherer Ordnung

Fahrplan

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

Inhalt

- Funktionen **höherer Ordnung**
 - Funktionen als **gleichberechtigte Objekte**
 - Funktionen als **Argumente**
 - Spezielle Funktionen: **map**, **filter**, **fold** und Freunde
- Formen der **Rekursion**:
 - **Einfache** und **allgemeine** Rekursion
- Typklassen

Funktionen als Werte

- Rekursive Definitionen, z.B. über Listen:

```
concat :: [[a]] -> [a]
concat []      = []
concat (x:xs) = x ++ concat xs
```

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

```
twice :: (a -> a) -> (a -> a)
twice f x = f (f x)
```

- Auswertung wie vorher: `twice (twice inc) 3` $\rightsquigarrow$ 7

Funktionen Höherer Ordnung

- Funktionen sind **gleichberechtigt**: Werte wie **alle anderen**
- **Grundprinzip** der funktionalen Programmierung
- Funktionen als **Argumente**.
- Vorzüge:
 - Modellierung **allgemeiner** Berechnungsmuster
 - Höhere **Wiederverwendbarkeit**
 - Größere **Abstraktion**

Funktionen als Argumente: Funktionskomposition

- Funktionskomposition

$(.) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c$

$(f \ . \ g) \ x = f \ (g \ x)$

- Vordefiniert
- Lies: f nach g

- Funktionskomposition **vorwärts**:

$(>.) :: (a \rightarrow b) \rightarrow (b \rightarrow c) \rightarrow a \rightarrow c$

$(f \ > . \ g) \ x = g \ (f \ x)$

- **Nicht** vordefiniert!

Funktionen als Argumente: map

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

`map :: (a -> b) -> [a] -> [b]`

- Definition

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

- Beispiel:

`lowercase :: String -> String`
`lowercase str = map toLower str`

Funktionen als Argumente: filter

- Elemente **filtern**: filter

- Signatur:

```
filter :: (a-> Bool)-> [a]-> [a]
```

- Definition

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

- Beispiel:

```
qsort :: [a]-> [a]  
  
qsort []        = []  
qsort (x:xs)    = qsort (filter (\y-> y < x) xs) ++  
                  filter (\y-> y == x) (x:xs) ++  
                  qsort (filter (\y-> x < y) xs)
```

Beispiel: Primzahlen

- Sieb des Erathostenes

- Für jede gefundene Primzahl p alle Vielfachen heraussieben
- Dazu: **filtern** mit $\backslash n \rightarrow \text{mod } n \ p \neq 0$

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

- Primzahlen im Intervall $[1.. n]$:

```
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**:

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

- **Partielle** Anwendung von Funktionen:

- Für $f :: a \rightarrow b \rightarrow c$, $x :: a$ ist $f\ x :: b \rightarrow c$

- Beispiele:

- `map toLower :: String -> String`
- `3 == :: Int -> Bool`
- `concat . map (replicate 2) :: String -> String`

Die Kürzungsregel

Bei **Anwendung** der Funktion

$$f :: t_1 \rightarrow t_2 \rightarrow \dots \rightarrow t_n \rightarrow t$$

auf k **Argumente** mit $k \leq n$

$$e_1 :: t_1, e_2 :: t_2, \dots, e_k :: t_k$$

werden die **Typen der Argumente gekürzt**:

$$\begin{aligned} f &:: \cancel{t_1} \rightarrow \cancel{t_2} \rightarrow \dots \rightarrow \cancel{t_k} \rightarrow t_{k+1} \rightarrow \dots \rightarrow t_n \rightarrow t \\ f \ e_1 \ \dots e_k &:: t_{k+1} \rightarrow \dots \rightarrow t_n \rightarrow t \end{aligned}$$

- Beweis: Regel **App** (letzte VL)

Einfache Rekursion

- **Einfache Rekursion:** gegeben durch
 - eine Gleichung für die leere Liste
 - eine Gleichung für die nicht-leere Liste
- Beispiel:

```
sum :: [Int] -> Int
sum []      = 0
sum (x:xs)  = x + sum xs
```

- Weitere Beispiele: `length`, `concat`, `(++)`, ...
- Auswertung:

```
sum [4,7,3]      ~> 4 + 7 + 3 + 0
concat [A, B, C] ~> A ++ B ++ C++ []
```

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 :: (a -> b -> b) -> b -> [a] -> b`

- Definition

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

Beispiele: foldr

- **Beispiel:** Summieren von Listenelementen.

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

- **Beispiel:** Flachklopfen von Listen.

```
concat :: [[a]] -> [a]
concat xs = foldr (++) [] 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 :: (a -> b -> a) -> a -> [b] -> a
```

```
foldl f a [] = a
```

```
foldl f a (x:xs) = foldl f (f a x) xs
```

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

Noch ein Beispiel: rev revisited

- Listenumkehr ist falten **von links**:

```
rev' xs = foldl cons [] xs  
cons :: [a] -> a -> [a]  
cons xs x = x : xs
```

- Nur noch **eine** Rekursion

`foldl = foldr`

- Def: $(\otimes, A)$ ist ein **Monoid** wenn

$$A \otimes x = x$$

(Neutrales Element links)

$$x \otimes A = x$$

(Neutrales Element rechts)

$$(x \otimes y) \otimes z = x \otimes (y \otimes z)$$

(Assoziativität)

- Satz: Wenn $(\otimes, A)$ **Monoid**, dann

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

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:

```
struct listel {  
    void *hd;  
    struct listel *tl;  
};
```

```
typedef struct listel *list;
```

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

- Keine direkte Syntax (e.g. namenlose Funktionen)
- Typsystem zu schwach (keine Polymorphie)
- Funktionen = Zeiger auf Funktionen
- 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-> t1);
    if (f(l-> hd)) {
      l->t1= r;
      return l;
    }
    else {
      free(l);
      return r;
    }
  } }
```

Übersicht: vordefinierte Funktionen auf Listen II

<code>map</code>	<code>(a -> b) -> [a] -> [b]</code>	Auf alle anwenden
<code>filter</code>	<code>(a -> Bool) -> [a] -> [a]</code>	Elemente filtern
<code>foldr</code>	<code>(a -> b -> b)</code> <code>-> b -> [a] -> b</code>	Falten von rechts
<code>foldl</code>	<code>(a -> b -> a)</code> <code>-> a -> [b] -> a</code>	Falten von links
<code>takeWhile</code>	<code>(a -> Bool) -> [a] -> [a]</code>	Längster Prefix s.t. <code>p</code> gilt
<code>dropWhile</code>	<code>(a -> Bool) -> [a] -> [a]</code>	Rest davon
<code>any</code>	<code>(a -> Bool) -> [a] -> Bool</code>	<code>any p = or . map p</code>
<code>all</code>	<code>(a -> Bool) -> [a] -> Bool</code>	<code>all p = and . map p</code>
<code>elem</code>	<code>Eq a => a -> [a] -> Bool</code>	<code>elem x = any (x ==)</code>
<code>zipWith</code>	<code>(a -> b -> c)</code> <code>-> [a] -> [b] -> [c]</code>	Verallgemeinertes zip

Typklassen

- Allgemeiner Typ für elem:

`elem :: a -> [a] -> Bool`

zu `allgemein` wegen `c ==`

- `(==)` kann **nicht** für **alle** Typen definiert werden:
- Gleichheit auf Funktionen **nicht entscheidbar**.
 - z.B. `(==) :: (Int -> Int) -> (Int -> Int) -> Bool`
 - Extensionale vs. intensionale Gleichheit

Typklassen

- Lösung: Typklassen

```
elem :: Eq a => a -> [a] -> Bool  
elem c = any (c ==)
```

- Für `a` kann jeder Typ eingesetzt werden, für den `(==)` definiert ist.
- Typklassen erlauben systematisches **Überladen** (ad-hoc Polymorphie)
 - **Polymorphie**: auf allen Typen gleich definiert
 - **ad-hoc Polymorphie**: unterschiedliche Definition für jeden Typ möglich

Standard-Typklassen

- `Eq a` für `== :: a -> a -> Bool` (Gleichheit)
- `Ord a` für `<= :: a -> a -> Bool` (Ordnung)
 - Alle Basisdatentypen
 - Listen, Tupel
 - **Nicht** für Funktionen
 - Damit auch Typ für `qsort` oben:
`qsort :: Ord a => [a] -> [a]`
- `Show a` für `show :: a -> String`
 - Alle Basisdatentypen
 - Listen, Tupel
 - **Nicht** für Funktionen
- `Read a` für `read :: String -> a`
 - Siehe Show

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, **konkateniere** Ergebnisse
- **Mergesort:**
 - **teile** Liste in der **Hälfte**,
 - **sortiere** Teilstücke, füge **ordnungserhaltend** zusammen.

Beispiel für allgemeine Rekursion: Mergesort

- **Hauptfunktion:**

```
msort :: [Int] -> [Int]
msort xs
  | length xs <= 1 = xs
  | otherwise = merge (msort front) (msort back) where
    (front, back) = splitAt ((length xs) `div` 2) xs
```

- `splitAt :: Int -> [a] -> ([a], [a])` spaltet Liste auf
- **Hilfsfunktion:** ordnungserhaltendes Zusammenfügen

```
merge :: [Int] -> [Int] -> [Int]
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**
 - Spezielle Funktionen höherer Ordnung: **map**, **filter**, **fold** und Freunde
 - Partielle Applikation, Kürzungsregel
- Formen der **Rekursion**:
 - **Einfache** und **allgemeine** Rekursion
 - **Einfache** Rekursion entspricht **fold**
- Typklassen
 - **Überladen** von Bezeichnern

Vorlesung vom 26.11.08: Funktionaler Entwurf & Standarddatentypen

Fahrplan

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

Inhalt

- Funktionaler Entwurf und Entwicklung
 - Spezifikation
 - Programmentwurf
 - Implementierung
 - Testen
- Beispiele
- Standarddatentypen: `Maybe`, Bäume

Funktionaler Entwurf und Entwicklung

① Spezifikation:

- Definitionsbereich (Eingabewerte)
- Wertebereich (Ausgabewerte)
- Anforderungen definieren
- Anforderungen als **Eigenschaften** formulieren

~> **Signatur**

Funktionaler Entwurf und Entwicklung

① Spezifikation:

- Definitionsbereich (Eingabewerte)
- Wertebereich (Ausgabewerte)
- Anforderungen definieren
- Anforderungen als **Eigenschaften** formulieren

~> **Signatur**

② Programmmentwurf:

- Wie kann das Problem in **Teilprobleme zerlegt** werden?
- Wie können **Teillösungen zusammengesetzt** werden?
- Gibt es ein ähnliches (gelöstes) Problem?

~> **Erster Entwurf**

3 Implementierung:

- Effizienz
 - Wie würde man Korrektheit zeigen?
 - Termination
 - Gibt es hilfreiche Büchereifunktionen
 - Refaktorisierung: mögliche Verallgemeinerungen, `shared code`
- ⇒ Lauffähige Implementierung

3 Implementierung:

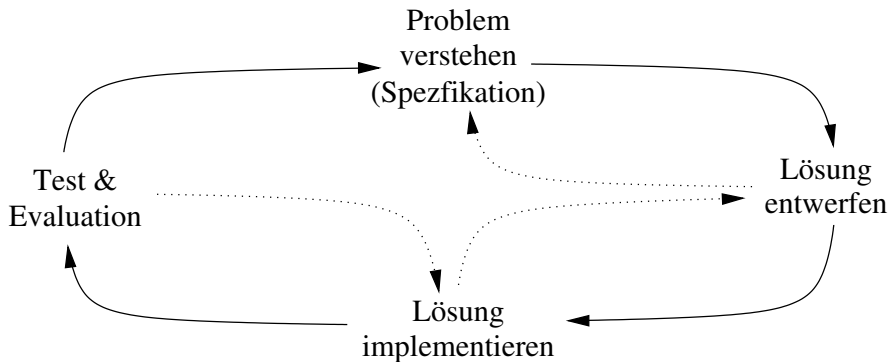
- Effizienz
- Wie würde man Korrektheit zeigen?
- Termination
- Gibt es hilfreiche Büchereifunktionen
- Refaktorisierung: mögliche Verallgemeinerungen, shared code

↪ Lauffähige Implementierung

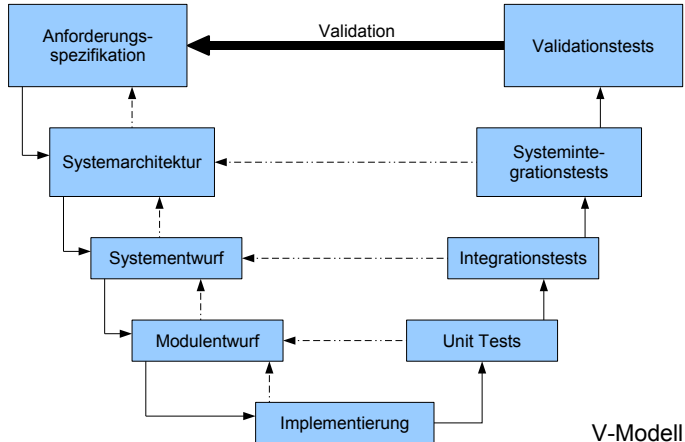
4 Test:

- Black-box Test: Testdaten aus der Spezifikation
- White-box Test: Testdaten aus der Implementierung
- Testdaten: hohe Abdeckung, Randfälle beachten.
- quickcheck: automatische Testdatenerzeugung

Der Programmentwicklungszyklus im kleinen



Vorgehensmodelle im Großen



1. Beispiel: größter gemeinsame Teiler

- Definitionsbereich: $\text{Int} \times \text{Int}$
- Wertebereich: Int
- Spezifikation:
 - Teiler: $a \mid b \iff \exists n. a \cdot n = b$

1. Beispiel: größter gemeinsame Teiler

- Definitionsbereich: $\text{Int} \times \text{Int}$
- Wertebereich: Int
- Spezifikation:
 - Teiler: $a \mid b \iff \exists n. a \cdot n = b$
 - Gemeinsamer Teiler: $\text{is_gcd}(x, y, z) \iff z \mid x \wedge z \mid y$

1. Beispiel: größter gemeinsame Teiler

- Definitionsbereich: $\text{Int} \times \text{Int}$
- Wertebereich: Int
- Spezifikation:
 - Teiler: $a \mid b \iff \exists n. a \cdot n = b$
 - Gemeinsamer Teiler: $\text{is_gcd}(x, y, z) \iff z \mid x \wedge z \mid y$
 - Grenzen: $\text{gcd}(x, y) \leq x, \text{gcd}(x, y) \leq y$ damit $\text{gcd}(x, y) \leq \min(x, y)$

1. Beispiel: größter gemeinsame Teiler

- Definitionsbereich: $\text{Int} \times \text{Int}$
- Wertebereich: Int
- Spezifikation:
 - Teiler: $a \mid b \iff \exists n. a \cdot n = b$
 - Gemeinsamer Teiler: $\text{is_cd}(x, y, z) \iff z \mid x \wedge z \mid y$
 - Grenzen: $\text{gcd}(x, y) \leq x, \text{gcd}(x, y) \leq y$ damit $\text{gcd}(x, y) \leq \min(x, y)$
 - größter gemeinsamer Teiler: $\forall i. \text{gcd}(x, y) < i \leq \min(x, y) \implies \neg \text{cd}(x, y, i)$

- **Signatur**

`gcd :: Int-> Int-> Int`

- **Eigenschaften** (ausführbare Spezifikationen) formulieren

- Problem: **Existenzquantor** — besser: $a \mid b \iff b \bmod a = 0$

`divides :: Int-> Int-> Bool`

`divides a b = mod b a == 0`

- Gemeinsamer Teiler:

`is_cd :: Int-> Int-> Int-> Bool`

`is_cd x y a = divides a x && divides a y`

- Größter gemeinsamer Teiler:

`no_larger :: Int-> Int-> Int-> Bool`

`no_larger x y g = all (\i-> not (is_cd x y i)) [g .. min x y]`

ggT: Analyse

- Reduktion auf kleineres Teilproblem: $a \mid b \iff \exists n. a \cdot n = b$
- Fallunterscheidung:
 - $n = 1$ dann $a = b$
 - $n = m + 1$, dann $a(m + 1) = am + a = b$, also $am = b - a \iff a \mid b - a$
- Damit Abbruchbedingung: beide Argumente gleich
- Reduktion: $a < b \rightsquigarrow \gcd(a, b) = \gcd(a, b - a)$

Kritik der Lösung

- Terminiert nicht bei negativen Zahlen oder 0.

```
gcd2 :: Int-> Int-> Int
gcd2 a b = gcd' (abs a) (abs b) where
  gcd' a b | a == 0 && b == 0 = error "gcd 0 0 undefined"
           | a == b || b == 0 = a
           | a < b             = gcd' (b- a) a
           | otherwise         = gcd' b a
```

- Ineffizient — es gilt auch $\text{gcd}(a, b) = \text{gcd}(b, a \bmod b)$ (Euklid'scher Algorithmus)
- Es gibt eine Bückereifunktion.

2. Beispiel: das n -Königinnen-Problem

- **Problem:** n Königinnen auf $n \times n$ -Schachbrett sicher platzieren
- **Spezifikation:**
 - Position der Königinnen

`type Pos = (Int, Int)`

- **Eingabe:** Anzahl Königinnen, **Rückgabe:** Positionen

`queens :: Int -> [[Pos]]`

n -Königinnen: Spezifikation

- **Sicher** gdw. kein gegenseitiges **Schlagen**.
- **Diagonalen**: $x - y = c$, $x + y = c'$
- $(x, y) \sim (p, q) \iff x \neq p \wedge y \neq q \wedge x - y \neq p - q \wedge x + y \neq p + q$
- Spezifikation:
 - Alle **Lösungen** sind auf dem Feld,
 - alle **Lösungen** haben n **Positionen**,
 - und sind gegenseitig **sicher**:

$$\forall Q \in \text{queens}(n). \forall (x, y) \in Q. 1 \leq x \leq n \wedge 1 \leq y \leq n$$

$$\forall Q \in \text{queens}(n). |Q| = n$$

$$\forall Q \in \text{queens}(n). \forall p_1, p_2 \in Q. p_1 = p_2 \vee p_1 \sim p_2$$

n -Königinnen: Eigenschaften

- Eigenschaften (ausführbare Spezifikation):

```
inRange :: Int-> Pos-> Bool
```

```
inRange n (x, y) = 1 <= x && x <= n && 1 <= y && y <= n
```

```
enough :: Int-> [Pos]-> Bool
```

```
enough n q = length q == n
```

```
isSafe :: Pos-> Pos-> Bool
```

```
isSafe (x, y) (p, q) =
```

```
  x /= p && y /= q && x- y /= p- q && x+ y /= p+ q
```

```
allSafe :: [Pos]-> Bool
```

```
allSafe q =
```

```
  all (\p-> all (\r-> (p == r || isSafe p r)) q) q
```

```
isSolution :: Int-> [[Pos]]-> Bool
```

```
isSolution n q = all (all (inRange n)) q && all (en
```



n -Königinnen: Rekursive Formulierung

- Rekursive Formulierung:
 - Keine Königin— kein Problem.
 - Lösung für n Königinnen: Lösung für $n - 1$ Königinnen, n -te Königin so stellen, dass sie keine andere bedroht.
- Vereinfachung: n -te Königin muß in n -ter Spalte platziert werden.
 - Limitiert kombinatorische Explosion

n -Königinnen: Hauptfunktion

- Hauptfunktion:

- Sei p neue Zeile
- $\text{cand } p$ bisherige Teillösungen, die mit (n, p) sicher sind
- $\text{put } p \ q$ fügt neue Position p zu Teillösung q hinzu

```
queens num = putqueens num where
  putqueens :: Int -> [[Pos]]
  putqueens n =
    if n == 0 then [[]]
    else let cand p  = filter (\q -> safe q (n, p))
                               (putqueens (n-1))
          put  p q = q ++ [(n, p)]
    in   concatMap (\p -> map (put p) (cand p))
          [1.. num]
```

- Rekursion über Anzahl der Königinnen
- Daher Termination

Das n -Königinnen-Problem

- Sichere neue Position: durch keine andere bedroht

```
safe :: [Pos] -> Pos -> Bool
```

```
safe others p = all (not . threatens p) others
```

- Gegenseitige Bedrohung:

- Bedrohung wenn in gleicher Zeile, Spalte, oder Diagonale.

```
threatens :: Pos -> Pos -> Bool
```

```
threatens (i, j) (m, n) =
```

```
    (j == n) || (i + j == m + n) || (i - j == m - n)
```

- Test auf gleicher Spalte $i == m$ unnötig.

Das n -Königinnen-Problem: Testen

- Testdaten (**manuell**):
 - `queens 0, queens 1, queens 2, queens 3, queens 4`
- Test (**automatisiert**):
 - `all (\n-> is\_solution n (queens n)) [1.. 8]`

3. Beispiel: Der Index

- **Problem:**

- Gegeben ein **Text**

```
brösel fasel\nbrösel brösel\nfasel brösel blubb
```

- Zu erstellen ein **Index**: für **jedes Wort** Liste der **Zeilen**, in der es **auftritt**

```
brösel [1, 2, 3]          blubb [3]          fasel [1, 3]
```

- **Spezifikation** der Lösung

```
type Doc = String
type Word= String
makeIndex :: Doc-> [[Int], Word]
```

- Keine **Leereinträge**
- Alle Wörter im **Index** müssen im **Text** in der **angegebenen Zeile** auftreten

Der Index: Eigenschaften

- Keine **Leereinträge**

```
notEmpty :: [(Int], Word)] -> Bool  
notEmpty idx = all (\ (l, w)-> not (null l)) idx
```

- Alle Wörter im **Index** im Text in der **angegebenen Zeile**
 - NB. Index erster Zeile ist 1.

```
occursInLine :: Word-> Int-> Doc-> Bool  
occursInLine w l txt = isInfixOf w (lines txt !! (l-1))
```

- **Eigenschaften**, zusammengefasst:

```
prop_notempty :: String-> Bool  
prop_notempty doc = notEmpty (makeIndex doc)
```

```
prop_occurs :: String-> Bool  
prop_occurs doc =  
  all (\ (ls, w)-> all (\l-> occursInLine w l doc) ls)  
    (makeIndex doc)
```

Zerlegung des Problems: erste Näherung

- Text in **Zeilen** zerteilen
- Zeilen in **Wörter** zerteilen
- Jedes Wort mit **Zeilennummer** versehen
- Gleiche Worte **zusammenfassen**
- **Sortieren**

Zerlegung des Problems: zweite Näherung

Ergebnistyp

[Line]

- ① Text in Zeilen aufspalten:
(mit `type Line= String`)

Zerlegung des Problems: zweite Näherung

Ergebnistyp

- ① Text in Zeilen aufspalten:
(mit `type Line= String`)

[Line]

- ② Jede Zeile mit ihrer Nummer versehen:

[(Int, Line)]

Zerlegung des Problems: zweite Näherung

- | | Ergebnistyp |
|---|---------------|
| ① Text in Zeilen aufspalten:
(mit <code>type Line= String</code>) | [Line] |
| ② Jede Zeile mit ihrer Nummer versehen: | [(Int, Line)] |
| ③ Zeilen in Wörter spalten (Zeilennummer beibehalten): | [(Int, Word)] |

Zerlegung des Problems: zweite Näherung

- | | Ergebnistyp |
|---|---------------|
| ① Text in Zeilen aufspalten:
(mit <code>type Line= String</code>) | [Line] |
| ② Jede Zeile mit ihrer Nummer versehen: | [(Int, Line)] |
| ③ Zeilen in Wörter spalten (Zeilennummer beibehalten): | [(Int, Word)] |
| ④ Liste alphabetisch nach Wörtern sortieren: | [(Int, Word)] |

Zerlegung des Problems: zweite Näherung

Ergebnistyp

- ① Text in Zeilen aufspalten:
(mit `type Line= String`) [Line]
- ② Jede Zeile mit ihrer Nummer versehen: [(Int, Line)]
- ③ Zeilen in Wörter spalten (Zeilennummer beibehalten):
[(Int, Word)]
- ④ Liste alphabetisch nach Wörtern sortieren: [(Int, Word)]
- ⑤ Gleiche Wörter in unterschiedlichen Zeilen zusammenfassen:
[([Int], Word)]

Zerlegung des Problems: zweite Näherung

Ergebnistyp

- 1 Text in Zeilen aufspalten:
(mit `type Line= String`) [Line]
- 2 Jede Zeile mit ihrer Nummer versehen: [(Int, Line)]
- 3 Zeilen in Wörter spalten (Zeilennummer beibehalten): [(Int, Word)]
- 4 Liste alphabetisch nach Wörtern sortieren: [(Int, Word)]
- 5 Gleiche Wörter in unterschiedlichen Zeilen zusammenfassen: [[Int], Word)]
- 6 Alle Wörter mit weniger als vier Buchstaben entfernen: [[Int], Word)]

Erste Implementierung:

```
type Line = String
makeIndex =
  shorten .      --      -> [[Int], Word)]
  amalgamate .  --      -> [[Int],Word)]
  makeLists .   --      -> [[Int],Word)]
  sortLs .      --      -> [(Int,Word)]
  allNumWords . --      -> [(Int,Word)]
  numLines .    --      -> [(Int, Line)]
  lines        -- Doc-> [Line]
```

Implementierung von Schritt 1–2

- In Zeilen zerlegen: `lines :: String-> [String]`
- Jede Zeile mit ihrer Nummer versehen:

```
numLines :: [Line]-> [(Int, Line)]  
numLines lines = zip [1.. length lines] lines
```

- Jede Zeile in Wörter zerlegen:
 - Pro Zeile `words :: String-> [String]`
 - Berücksichtigt nur Leerzeichen.
 - Vorher alle Satzzeichen durch Leerzeichen ersetzen.

Implementierung von Schritt 3

- Zusammengenommen:

```
splitWords :: Line-> [Word]
splitWords = words . map (\c-> if isPunct c then ' '
                             else c) where

    isPunct :: Char-> Bool
    isPunct c = c `elem` ";:.,\`\"!()?{}-\\[]"
```

- Auf **alle Zeilen** anwenden, Ergebnisliste **flachklopfen**.

```
allNumWords :: [(Int, Line)]-> [(Int, Word)]
allNumWords = concatMap oneLine where
    oneLine :: (Int, Line)-> [(Int, Word)]
    oneLine (num, line) = map (\w-> (num, w))
                             (splitWords line)
```

Einschub: Ordnungen

- Generische Sortierfunktion

- Ordnung als Parameter

```
qsortBy :: (a -> a -> Bool) -> [a] -> [a]
qsortBy ord [] = []
qsortBy ord (x:xs) =
    qsortBy ord (filter (\y -> ord y x) xs) ++ [x] ++
    qsortBy ord (filter (\y -> not (ord y x)) xs)
```

- Vordefiniert (aber andere Signatur):

```
sortBy :: (a -> a -> Ordering) -> [a] -> [a]
```

Implementation von Schritt 4

- Liste alphabetisch nach Wörtern sortieren:
 - Ordnungsrelation definieren:

```
ordWord :: (Int, Word) -> (Int, Word) -> Bool
ordWord (n1, w1) (n2, w2) =
    w1 < w2 || (w1 == w2 && n1 <= n2)
```

- Sortieren mit generischer Sortierfunktion `qsortBy`

```
sortLs :: [(Int, Word)] -> [(Int, Word)]
sortLs = qsortBy ordWord
```

Implementation von Schritt 5

- **Gleiche Wörter** in **unterschiedlichen Zeilen** zusammenfassen:
 - Erster Schritt: Jede **Zeile** zu (einelementiger) **Liste von Zeilen**.

```
makeLists :: [(Int, Word)] -> [[(Int, Word)]]  
makeLists = map (\ (l, w) -> [(l, w)])
```

- Zweiter Schritt: **Gleiche Wörter** zusammenfassen.
 - Nach Sortierung sind **gleiche Wörter hintereinander**!

```
amalgamate :: [[(Int, Word)]] -> [(Int, Word)]  
amalgamate [] = []  
amalgamate [p] = [p]  
amalgamate ((l1, w1):(l2, w2):rest)  
  | w1 == w2 = amalgamate ((l1++ l2, w1):rest)  
  | otherwise = (l1, w1):amalgamate ((l2, w2):rest)
```

Implementation von Schritt 6 — Test

- Alle Wörter mit weniger als vier Buchstaben entfernen:

```
shorten :: [[Int],Word] -> [[Int],Word]  
shorten = filter \ (_, wd) -> length wd >= 4)
```

- Alternative Definition:

```
shorten = filter ((>= 4) . length . snd)
```

- Testfälle:

- `makeIndex ""`
- `makeIndex "a b a"`
- `makeIndex "abcdef abcde"`
- `makeIndex "a eins zwei\nzwei\nzwei,eins"`

Standarddatentypen

- Listen $[a]$
- Paare (a, b)
- Lifting $\text{Maybe } a$
- Bäume

Modellierung von Fehlern: Maybe a

- Typ a plus **Fehlerelement**

- Im Prelude vordefiniert.

```
data Maybe a = Just a | Nothing
```

- Nothing modelliert Fehlerfall:

```
find :: (a-> Bool)-> [a]-> Maybe a
find p []      = Nothing
find p (x:xs) = if p x then Just x
                else find p xs
```

Funktionen auf Maybe a

- Anwendung von Funktion mit Default-Wert für Fehler (**vordefiniert**):

```
maybe :: b -> (a -> b) -> Maybe a -> b  
maybe d f Nothing  = d  
maybe d f (Just x) = f x
```

- **Liften** von Funktionen ohne Fehlerbehandlung:

- Fehler bleiben **erhalten**.

```
fmap :: (a -> b) -> Maybe a -> Maybe b  
fmap f Nothing = Nothing  
fmap f (Just x) = Just (f x)
```

Binäre Bäume

Ein binärer Baum ist

- Entweder leer,
- oder ein Knoten mit genau **zwei** Unterbäumen.

Knoten tragen eine Markierung.

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

- Andere **Möglichkeit**: **Unterschiedliche** Markierungen Blätter und Knoten

```
data Tree' a b = Null'
               | Leaf' b
               | Node' (Tree' a b) a (Tree' a b)
```

Funktionen auf Bäumen

- Test auf Enthaltensein:

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

- Höhe:

```
height :: Tree a -> Int
height Null = 0
height (Node l a r) = max (height l) (height r) + 1
```

Funktionen auf Bäumen

Primitive Rekursion auf Bäumen:

- Rekursionsanfang
- Rekursionsschritt:
 - Label des Knotens,
 - **Zwei** Rückgabewerte für **linken** und **rechten** Unterbaum.

```
foldT :: (a -> b -> b -> b) -> b -> Tree a -> b
```

```
foldT f e Null = e
```

```
foldT f e (Node l a r) = f a (foldT f e l) (foldT f e r)
```

Funktionen auf Bäumen

- Damit Elementtest:

```
member' :: Eq a => Tree a -> a -> Bool
member' t x =
    foldT (\e b1 b2 -> e == x || b1 || b2) False t
```

- Höhe:

```
height' :: Tree a -> Int
height' = foldT (\ _ h1 h2 -> 1 + max h1 h2) 0
```

Funktionen auf Bäumen

- Traversal: preorder, inorder, postorder

```
preorder  :: Tree a -> [a]
inorder   :: Tree a -> [a]
postorder :: Tree a -> [a]
preorder  = foldT (\x t1 t2 -> [x] ++ t1 ++ t2) []
inorder   = foldT (\x t1 t2 -> t1 ++ [x] ++ t2) []
postorder = foldT (\x t1 t2 -> t1 ++ t2 ++ [x]) []
```

- Äquivalente Definition ohne foldT:

```
preorder' Null          = []
preorder' (Node l a r) = [a] ++ preorder' l ++ preorder' r
```

- Wie würde man **geordnete Bäume** implementieren?

Zusammenfassung

- Funktionaler Entwurf:
 - Entwurf: Signatur, Eigenschaften
 - Implementierung: Zerlegung, Reduktion, Komposition
 - Testen: Testdaten, quickcheck
 - Ggf. wiederholen
- Standarddatentypen: Maybe a, Bäume
- Nächste Woche: abstrakte Datentypen

Vorlesung vom 10.12.08: Signaturen und Eigenschaften

Fahrplan

- Teil I: Grundlagen
- Teil II: Abstraktion
 - Abstrakte Datentypen
 - Signaturen & Axiome
 - Korrektheit von Programmen
 - Zustand und Aktionen
- Teil III: Beispiele, Anwendungen, Ausblicke

Abstrakte Datentypen

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

Signatures

Definition: Die **Signatur** eines abstrakten Datentyps besteht aus den Typen, und der Signatur der Funktionen darauf.

- Obs: Keine direkte Repräsentation in Haskell
- Signatur: Typ eines Moduls

Der Speicher: Signatur

- Ein Speicher (Store, FiniteMap, State)
- Typen: der eigentliche Speicher S , Adressen a , Werte b
- Operationen:
 - leerer Speicher: S

Der Speicher: Signatur

- Ein **Speicher** (Store, FiniteMap, State)
- Typen: der eigentliche Speicher S , Adressen a , Werte b
- Operationen:
 - **leerer** Speicher: S
 - in Speicher an eine Stelle einen Wert **schreiben**: $S \rightarrow a \rightarrow b \rightarrow S$

Der Speicher: Signatur

- Ein **Speicher** (Store, FiniteMap, State)
- Typen: der eigentliche Speicher S , Adressen a , Werte b
- Operationen:
 - **leerer** Speicher: S
 - in Speicher an eine Stelle einen Wert **schreiben**: $S \rightarrow a \rightarrow b \rightarrow S$
 - aus Speicher an einer Stelle einen Wert **lesen**: $S \rightarrow a \rightarrow b$ (partiell)

Der Speicher: Signatur

- Ein **Speicher** (Store, FiniteMap, State)
- Typen: der eigentliche Speicher S , Adressen a , Werte b
- Operationen:
 - **leerer** Speicher: S
 - in Speicher an eine Stelle einen Wert **schreiben**: $S \rightarrow a \rightarrow b \rightarrow S$
 - aus Speicher an einer Stelle einen Wert **lesen**: $S \rightarrow a \rightarrow b$ (partiell)

Der Speicher: Signatur

- Adressen und Werte sind Parameter

`type Store a b`

- Leerer Speicher:

`empty :: Store a b`

- In Speicher an eine Stelle einen Wert schreiben:

`upd :: Store a b -> a -> b -> Store a b`

- Aus Speicher an einer Stelle einen Wert lesen:

`get :: Store a b -> a -> Maybe b`

Signatur und Eigenschaften

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

Beschreibung von Eigenschaften: Axiome

- **Axiome** sind **Prädikate** über den **Operationen** der Signatur
 - **Elementare** Prädikate P
 - **Gleichheit** $s == t$
 - **Bedingte** Prädikate: $A \implies B$
- **Beobachtbare** Typen: interne Struktur bekannt
 - Vordefinierte Typen (**Zahlen**, **Zeichen**, **Listen**), algebraische Datentypen
- **Abstrakte** Typen: interne Struktur unbekannt
 - Gleichheit (wenn definiert)

Beispiel: Speicher

- Beobachtbar: Adressen und Werte, abstrakt: Speicher
- Axiome für das Lesen:

Beispiel: Speicher

- **Beobachtbar**: Adressen und Werte, abstrakt: Speicher
- **Axiome** für das **Lesen**:
 - Lesen aus dem leeren Speicher undefiniert:

Beispiel: Speicher

- **Beobachtbar**: Adressen und Werte, abstrakt: Speicher
- **Axiome** für das **Lesen**:
 - Lesen aus dem leeren Speicher undefiniert:
`get empty == Nothing`
 - Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:

Beispiel: Speicher

- **Beobachtbar**: Adressen und Werte, abstrakt: Speicher
- **Axiome** für das **Lesen**:
 - Lesen aus dem leeren Speicher undefiniert:
`get empty == Nothing`
 - Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:
`get (upd s a v) a == Just v`
 - Lesen an anderer Stelle als vorher geschrieben liefert ursprünglichen Wert:

Beispiel: Speicher

- **Beobachtbar**: Adressen und Werte, abstrakt: Speicher
- **Axiome für das Lesen**:
 - Lesen aus dem leeren Speicher undefiniert:
$$\text{get empty} == \text{Nothing}$$
 - Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:
$$\text{get (upd s a v) a} == \text{Just v}$$
 - Lesen an anderer Stelle als vorher geschrieben liefert ursprünglichen Wert:
$$a1 \neq a2 \implies \text{get (upd s a1 v) a2} == \text{get s a2}$$
- **Axiome für das Schreiben**:
 - Schreiben an dieselbe Stelle überschreibt alten Wert:

Beispiel: Speicher

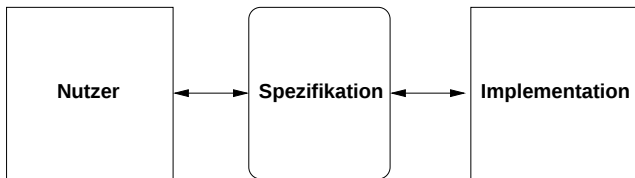
- **Beobachtbar**: Adressen und Werte, abstrakt: Speicher
- Axiome für das **Lesen**:
 - Lesen aus dem leeren Speicher undefiniert:
$$\text{get empty} == \text{Nothing}$$
 - Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:
$$\text{get (upd s a v) a} == \text{Just v}$$
 - Lesen an anderer Stelle als vorher geschrieben liefert ursprünglichen Wert:
$$a1 \neq a2 \implies \text{get (upd s a1 v) a2} == \text{get s a2}$$
- Axiome für das **Schreiben**:
 - Schreiben an dieselbe Stelle überschreibt alten Wert:
$$\text{upd (upd s a v) a w} == \text{upd s a w}$$
 - Schreiben über verschiedene Stellen kommutiert (Reihenfolge irrelevant):

Beispiel: Speicher

- **Beobachtbar**: Adressen und Werte, abstrakt: Speicher
- **Axiome für das Lesen**:
 - Lesen aus dem leeren Speicher undefiniert:
$$\text{get empty} == \text{Nothing}$$
 - Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:
$$\text{get (upd s a v) a} == \text{Just v}$$
 - Lesen an anderer Stelle als vorher geschrieben liefert ursprünglichen Wert:
$$a1 \neq a2 \implies \text{get (upd s a1 v) a2} == \text{get s a2}$$
- **Axiome für das Schreiben**:
 - Schreiben an dieselbe Stelle überschreibt alten Wert:
$$\text{upd (upd s a v) a w} == \text{upd s a w}$$
 - Schreiben über verschiedene Stellen kommutiert (Reihenfolge irrelevant):
$$a1 \neq a2 \implies \text{upd (upd s a1 v) a2 w} == \text{upd (upd s a2 w) a1 v}$$

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 mit **quickCheck** getestet werden
- Axiome können (sollten?) **bewiesen** werden

Implementation des Speicher: erster Versuch

- Speicher als **Funktion**

```
type Store a b = a-> Maybe b
```

- **Leerer** Speicher: konstant undefiniert

```
empty :: Store a b  
empty = \x-> Nothing
```

- **Lesen**: Funktion anwenden

```
get :: Eq a=> Store a b-> a-> Maybe b  
get s a = s a
```

- **Schreiben**: punktweise Funktionsdefinition

- Auf Adresstyp a muss Gleichheit existieren

```
upd :: Eq a=> Store a b-> a-> b-> Store a b  
upd s a b = \x-> if x== a then Just b else s x
```

Nachteil dieser Implementation

- **Typsynonyme** immer sichtbar
- Deshalb **Verkapselung** des Typen:

```
data Store a b = Store (a -> Maybe b)
```

Implementation des Speicher: zweiter Versuch

- Speicher als Funktion

```
data Store a b = Store (a-> Maybe b)
```

- Leerer Speicher: konstant undefiniert

```
empty :: Store a b  
empty = Store (\x-> Nothing)
```

- **Lesen:** Funktion anwenden

```
get :: Eq a=> Store a b-> a-> Maybe b  
get (Store s) a = s a
```

- **Schreiben:** punktweise Funktionsdefinition

- Auf Adresstyp a muss Gleichheit existieren

```
upd :: Eq a=> Store a b-> a-> b-> Store a b  
upd (Store s) a b =  
    Store (\x-> if x== a then Just b else s x)
```

Beweis der Axiome

- Lesen aus leerem Speicher:

```
get empty a
= get (Store (\x-> Nothing)) a
= (\x-> Nothing) a
= Nothing
```

- Lesen und Schreiben an gleicher Stelle:

```
get (upd (Store s) a v) a
= get (\x-> if x == a then Just v else s x) a
= (\x-> if x == a then Just v else s x) a
= if a == a then Just v else s a
= if True then Just v else s a
= Just v
```

Beweis der Axiome

- Lesen an anderer Stelle:

```
get (upd (Store s) a v) b
= get (\x-> if x == a then Just v else s x) b
= (\x-> if x == a then Just v else s x) b
= if a == b then Just v else s b
= if False then Just v else s b
= s b
= get (Store s) b
```

Bewertung der Implementation

- Vorteil: **effizient** (keine Rekursion!)
- Nachteile:
 - Keine Gleichheit auf `Store a b` — Axiome nicht **erfüllbar**
 - Speicherleck — überschriebene Werte bleiben im Zugriff

Der Speicher als Graph

- Typ `Store a b`: Liste von Paaren (a, b)
- Graph $G(f)$ der partiellen Abbildung $f : A \rightarrow B$

$$G(f) = \{(a, f(a)) \mid a \in A, f(a) \neq \perp\}$$

- **Invariante** der Liste: für jedes a höchstens ein Paar (a, v)
- **Leerer** Speicher: leere Menge
- **Lesen** von a :
 - Wenn Paar (a, v) in Menge, **Just** v , ansonsten $\perp$
- **Schreiben** von a an der Stelle v :
 - Existierende (a, w) für alle w entfernen, dann (a, v) hinzufügen

Der Speicher als Funktionsgraph

- **Datentyp** (verkapselt):

```
data Store a b = Store [(a, b)]
```

- Leerer **Speicher**:

```
empty :: Store a b  
empty = Store []
```

Operationen (rekursiv)

- **Lesen:** rekursive Formulierung

```
get :: Eq a => Store a b -> a -> Maybe b
get (Store s) a = get' s where
    get' []           = Nothing
    get' ((b, v):s) =
        if a == b then Just v else get' s
```

- **Schreiben**

```
upd :: Eq a => Store a b -> a -> b -> Store a b
upd (Store s) a v = Store (upd' s) where
    upd' []           = [(a, v)]
    upd' ((b, w):s) =
        if a == b then (a, v):s
        else (b,w): upd' s
```

Operationen (kürzer)

- Lesen: kürzere Alternative

```
get (Store s) a =  
  case filter ((a ==). fst) s of  
    ((b, v):_) -> Just v  
    []          -> Nothing
```

- Schreiben:

```
upd (Store s) a v =  
  Store ((a, v): filter ((a ==) . fst) s)
```

Axiome als Testbare Eigenschaften

- **Test**: zufällige Werte einsetzen, Auswertung auf **True** prüfen
- Polymorphe Variablen nicht **testbar**
- Deshalb Typvariablen **instantiieren**
 - Typ muss genug Element haben (**Int**)
 - Durch Signatur Typinstanz erzwingen
- **Freie Variablen** der Eigenschaft werden **Parameter** der Testfunktion

Axiome als Testbare Eigenschaften

- Für das Lesen:

```
prop_read_empty :: Int-> Bool
```

```
prop_read_empty a =
```

```
  get (empty :: Store Int Int) a == Nothing
```

```
prop_read_write :: Store Int Int-> Int-> Int-> Bool
```

```
prop_read_write s a v=
```

```
  get (upd s a v) a == Just v
```

Axiome als Testbare Eigenschaften

- **Bedingte** Eigenschaft in `quickCheck`:

- $A \implies B$ mit A, B Eigenschaften
- Typ ist `Property`

```
prop_read_write_other ::  
  Store Int Int -> Int -> Int -> Int -> Property
```

```
prop_read_write_other s a v b =  
  a /= b ==> get (upd s a v) b == get s b
```

Axiome als Testbare Eigenschaften

- Schreiben:

```
prop_write_write :: Store Int Int-> Int-> Int-> Int-> Bool
prop_write_write s a v w =
  upd (upd s a v) a w == upd s a w
```

- Schreiben an anderer Stelle:

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

- Test benötigt Gleichheit auf Store a b
 - Mehr als Gleichheit der Listen — Reihenfolge irrelevant

Beweis der Eigenschaften

- Problem: Rekursion

```
read (upd (Store s) a v) a  
= read (Store (upd' s a v)) a  
= read (Store ( ... ?)) a
```

- Lösung: nächste Vorlesung

ADTs vs. Objekte

- ADTs (z.B. Haskell): **Typ** plus **Operationen**
- Objekte (z.B. Java): **Interface**, **Methoden**.
- **Gemeinsamkeiten**: **Verkapselung** (information hiding) der Implementation
- **Unterschiede**:
 - Objekte haben **internen Zustand**, ADTs sind **referentiell transparent**;
 - Objekte haben **Konstruktoren**, ADTs nicht (Konstruktoren nicht unterscheidbar)
 - **Vererbungsstruktur** auf Objekten (**Verfeinerung** für ADTs)
 - Java: **interface** eigenes Sprachkonstrukt, Haskell: Signatur eines Moduls nicht (aber z.B. SML).

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
 - **Beweisen** der Korrektheit
- **quickCheck**:
 - Freie Variablen der Eigenschaften werden **Parameter** der Testfunktion
 - $\Rightarrow$ für **bedingte** Eigenschaften

Vorlesung vom 17.12.08: Verifikation und Beweis

Fahrplan

- Teil I: Grundlagen
- Teil II: Abstraktion
 - Abstrakte Datentypen
 - Signaturen & Axiome
 - Korrektheit von Programmen
 - Zustand und Aktionen
- Teil III: Beispiele, Anwendungen, Ausblicke

Inhalt

- **Verifikation:** Wann ist ein Programm korrekt?
- **Beweis:** Wie beweisen wir Korrektheit und andere Eigenschaften?
- Techniken:
 - Vollständige Induktion
 - Strukturelle Induktion
 - Fixpunktinduktion
- Beispiele

Warum beweisen?

- Test findet Fehler
- Beweis zeigt Korrektheit
- Formaler Beweis
 - Beweis nur durch Regeln der Logik
 - Maschinell überprüfbar (Theorembeweiser)
 - Hier: Aussagenlogik, Prädikatenlogik

Was beweisen?

- Prädikate:
 - Haskell-Ausdrücke vom Typ `Bool`
 - Allquantifizierte Aussagen:
wenn $P(x)$ Prädikat, dann ist $\forall x.P(x)$ auch ein Prädikat
 - Sonderfall Gleichungen $s == t$

Wie beweisen?

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

Ein ganz einfaches Beispiel

`addTwice :: Int -> Int -> Int`

`addTwice x y = 2*(x + y)`

zz: `addTwice x (y+z) == addTwice (x+y) z`

`addTwice x (y + z)`

`= 2*(x+(y+z))` Def. `addTwice`

`= 2*((x+y)+z)` Assoziativität von `+`

`= addTwice (x+y) z` Def. `addTwice`

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
```

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

$\max x y - \min x y$

Cases: 1. $x < y$

$= y - \min x y$ Def. max

$= y - x$ Def. min

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

2. $x \geq y$

$= x - \min x y$ Def. max

$= x - y$ Def. min

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

Rekursive Definition, induktiver Beweis

- Definition ist **rekursiv**

- Basisfall (leere Liste)
- Rekursion ($x:xs$)

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

- 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

Zu zeigen:

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

Beweis:

- Induktionsbasis: $P([])$
- Induktionsschritt:
Induktionsvoraussetzung $P(xs)$, zu zeigen $P(x : xs)$

Induktion: ein einfaches Beispiel

zz: $\text{map } f (\text{map } g \text{ xs}) == \text{map } (f. g) \text{ xs}$

Ind: 1. Induktionssbasis

$\text{map } f (\text{map } g [])$	
$= \text{map } f []$	Def. map für []
$= []$	Def. map für []
$= \text{map } (f.g) []$	Def. map für []

2. Induktionsschritt

$\text{map } f (\text{map } g (x:xs))$	
$= \text{map } f (g x: \text{map } g \text{ xs})$	Def. map für x:xs
$= f (g x): \text{map } f (\text{map } g \text{ xs})$	Def. map für x:xs
$= f (g x): \text{map } (f. g) \text{ xs}$	Induktionsvoraussetzung
$= (f. g) x: \text{map } (f.) \text{ xs}$	Def. .
$= \text{map } (f. g) (x:xs)$	Def. map für x:xs

Weitere Beispiele

$$\text{length (filter p xs)} \leq \text{length xs} \quad (1)$$

$$\text{length (xs ++ ys)} == \text{length xs} + \text{length ys} \quad (2)$$

$$\text{map f (xs ++ ys)} == \text{map f xs} ++ \text{map f ys} \quad (3)$$

$$\text{sum (map length xs)} == \text{length (concat xs)} \quad (4)$$

Strukturelle Induktion über anderen Datentypen

Gegeben binäre Bäume:

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

Zu zeigen:

Für alle (endlichen) Bäume t gilt $P(t)$

Beweis:

- Induktionsbasis: $P(\text{Null})$
- Induktionsschritt:
Voraussetzung $P(s)$, $P(t)$, zu zeigen $P(\text{Node } s \ a \ t)$.

Ein einfaches Beispiel

- Gegeben: `map` für Bäume:

```
fmap :: (a -> b) -> Tree a -> Tree b
```

```
fmap f Null = Null
```

```
fmap f (Node s a t) = Node (fmap f s) (f a) (fmap f t)
```

- Sowie **Aufzählung** der Knoten:

```
inorder :: Tree a -> [a]
```

```
inorder Null = []
```

```
inorder (Node s a t) = inorder s ++ [a] ++ inorder t
```

- Zu zeigen: `inorder (fmap f t) = map f (inorder t)`

Ein einfaches Beispiel

zz: `inorder (fmap f t) = map f (inorder t)`

Ind: 1. Induktionssbasis

```
    inorder (fmap f Null)
=  inorder Null
=  []
=  map f []
=  map f (inorder Null)
```

2. Induktionsschritt

```
    inorder (fmap f (Node s a t))
=  inorder (Node (fmap f s) (f a) (fmap f t))
=  inorder (fmap f s) ++ [f a] ++ inorder (fmap f t)
=  map f (inorder s) ++ [f a] ++ map f (inorder t)
=  map f (inorder s) ++ map f [a] ++ map f (inorder t)
=  map f (inorder s ++ [a] ++ inorder t)
=  map f (inorder (T s a t))
```

Eine Einfache Beweistaktik

- Induktionssbasis: einfach ausrechnen
 - Ggf. für zweite freie Variable zweite Induktion nötig
- Induktionsschritt:
 - 1 Definition der angewendeten Funktionen links nach rechts anwenden (auffalten)
 - 2 Ausdruck so umformen, dass Induktionssvoraussetzung anwendbar
 - 3 Definition der angewendeten Funktionen rechts nach links anwenden (einfalten)
- Schematisch: $P(x:xs) \rightsquigarrow E\ x\ (P\ xs) \rightsquigarrow E\ x\ (Q\ xs) \rightsquigarrow Q(x:xs)$

Fallbeispiel: Der Speicher

- Zur Erinnerung:

```
data Store a b = Store [(a, b)]
```

```
empty :: Store a b
```

```
empty = Store []
```

Der Speicher

- Hilfsfunktionen auf Dateiebene liften

- Lesen:

`get :: Eq a => Store a b -> a -> Maybe b`

`get (Store s) a = get' s a`

`get' :: Eq a => [(a, b)] -> a -> Maybe b`

`get' [] a = Nothing`

`get' ((b, v):s) a =`

`if a == b then Just v else get' s a`

- Schreiben:

`upd :: Eq a => Store a b -> a -> b -> Store a b`

`upd (Store s) a v = Store (upd' s a v) where`

`upd' :: Eq a => [(a, b)] -> a -> b -> [(a, b)]`

`upd' [] a v = [(a, v)]`

`upd' ((b, w):s) a v =`

`if a == b then (a, v):s else (b,w): upd' s a v`



Zusammenfassung

- Formaler Beweis vs. Testen:
 - Testen: einfach (automatisch), findet Fehler
 - Beweis: mühsam (nicht automatisierbar), zeigt Korrektheit
- Formaler Beweis hier:
 - Aussagenlogik, einfache Prädikate
- Beweismittel:
 - Gleichungen (Funktionsdefinitionen)
 - Fallunterscheidung
 - Strukturelle Induktion (für alle algebraischen Datentypen)

Vorlesung vom 07.01.09: Ein/Ausgabe in Funktionalen Sprachen

Fahrplan

- Teil I: Grundlagen
- Teil II: Abstraktion
 - Abstrakte Datentypen
 - Signaturen & Axiome
 - Korrektheit von Programmen
 - Zustand und Aktionen
- Teil III: Beispiele, Anwendungen, Ausblicke

- Ein/Ausgabe in funktionale Sprachen
- Wo ist das **Problem**?
- **Aktionen** und der Datentyp *IO*.
- **Beispiel**: Worte zählen (**wc**)
- **Aktionen** als **Werte**

Ein- und Ausgabe in funktionalen Sprachen

Problem:

- Funktionen mit Seiteneffekten nicht referentiell transparent.
- z. B. `readString :: ... -> String ??`

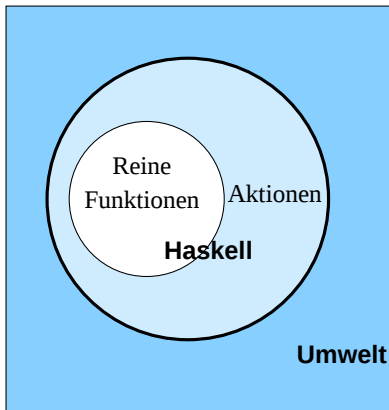
Ein- und Ausgabe in funktionalen Sprachen

Problem:

- Funktionen mit Seiteneffekten nicht referentiell transparent.
- z. B. `readString :: ... -> String ??`

Lösung:

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



Modellierung des Systemzustands

- **Idee:** Aktionen sind Transformationen auf Systemzustand Σ
- Operationen:
 - Adressen A , Werte V
 - Lesen: $\text{read} : A \rightarrow \Sigma \rightarrow V \times \Sigma$
 - Schreiben: $\text{write} : A \rightarrow V \rightarrow \Sigma \rightarrow \Sigma$,
 - Äquivalent, aber besser $\text{write} : A \rightarrow V \rightarrow \Sigma \rightarrow () \times \Sigma$
- Eine Aktion ist $I \rightarrow \Sigma \rightarrow O \times \Sigma \cong I \times \Sigma \rightarrow O \times \Sigma$
 - Einheitliche Signatur für alle Aktionen
 - Parametrisiert über Ergebnistyp O

Komposition von Aktionen

- Komposition ist **Funktionskomposition**
- Wenn **Rückgabewert** der ersten Aktion p **Eingabewert** der zweiten q :

$$\begin{array}{l} p : A \rightarrow \Sigma \rightarrow B \times \Sigma \\ q : B \rightarrow \Sigma \rightarrow C \times \Sigma \\ \hline p; q : A \rightarrow \Sigma \rightarrow C \times \Sigma \\ (p; q) \ a \ S = \text{let } (b, S_1) = p \ a \ S \\ \text{in } q \ b \ S_1 \end{array}$$

- **Lifting** $\hat{a} : \Sigma \rightarrow A \times \Sigma$ für jedes $a : A$ definiert als $\hat{a}(S) = (a, S)$

Aktionen als abstrakter Datentyp

- Idee: IO a ist $\Sigma \rightarrow a \times \Sigma$
- ADT mit Operationen **Komposition** und **Lifting**
- Signatur:

```
type IO t
```

```
(>>=)  :: IO a -> (a-> IO b) -> IO b
```

```
return :: a-> IO a
```

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

Vordefinierte 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 >>=
- 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  
  >>= \s-> 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.

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 c <- readFile file
     putStrLn (show (length (lines c),
                     length (words c),
                     length c) ++
               " lines, words, characters.")
```

- Nicht sehr effizient — Datei wird im Speicher gehalten.

Beispiel: wc verbessert.

- Effizienter: Funktion `cnt`, die Dateiinhalt **einmal** traversiert

```
cnt :: Int-> Int-> Int-> Bool-> String-> (Int, Int, Int)
cnt l w c _ [] = (l, w, c)
cnt l w c blank (x:xs)
  | isSpace x && not blank = cnt l' (w+1) (c+1) True xs
  | isSpace x && blank      = 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.

- Mit `cnt` besseres `wc`:

```
wc' :: String-> IO ()  
wc' file =  
    do c <- readFile file  
        putStrLn (show (cnt 0 0 0 False c)++  
                    " lines, words, characters.")
```

- Datei wird verzögert gelesen und dabei verbraucht.

Aktionen als Werte

- Aktionen sind Werte wie alle anderen.
- Dadurch Definition von Kontrollstrukturen möglich.
- Beispiele:

- Endlosschleife:

```
forever :: IO a -> IO a
forever a = a >> forever a
```

- Iteration (feste Anzahl):

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

Fehlerbehandlung

- Fehler werden durch `IOError` repräsentiert
- Fehlerbehandlung durch `Ausnahmen` (ähnlich Java)

```
ioError :: IOError -> IO a    -- "throw"  
catch   :: IO a -> (IOError -> IO a) -> IO a
```

- Fehlerbehandlung nur in Aktionen

Fehler fangen und behandeln

- Fehlerbehandlung für `wc`:

```
wc2 :: String -> IO ()  
wc2 file =  
    catch (wc' file)  
        (\e -> putStrLn ("Fehler beim Lesen: " ++ show e))
```

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

```
readIO :: Read a => String -> IO a
```

Map und Filter für Aktionen

- Map für Aktionen:

`mapM :: (a -> IO b) -> [a] -> IO [b]`

`mapM_ :: (a -> IO ()) -> [a] -> IO ()`

- Filter für Aktionen

- Importieren mit `import Monad (filterM).`

`filterM :: (a -> IO Bool) -> [a] -> IO [a]`

So ein Zufall!

- Zufallswerte:

```
randomRIO :: (a, a) -> IO a
```

- Warum ist randomIO Aktion?

- Beispiel: Aktionen zufällig oft ausführen

```
atmost :: Int -> IO a -> IO [a]
```

```
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)
```

```
main = do
```

```
    args <- getArgs
```

```
    mapM wc2 args
```

Zusammenfassung

- Ein/Ausgabe in Haskell durch **Aktionen**
- **Aktionen** (Typ `IO a`) sind seiteneffektbehaftete Funktionen
- Komposition von Aktionen durch

```
(>>=)  :: IO a -> (a -> IO b) -> IO b  
return :: a -> IO a
```

- do-Notation
- Fehlerbehandlung durch Ausnahmen (`IOError`, `catch`).
- Verschiedene Funktionen der Standardbibliothek:
 - Prelude: `getLine`, `putStr`, `putStrLn`, `readFile`, `writeFile`
 - Module: `IO`, `Random`