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

  $$P : \mathit{Eingabe} \to \mathit{Ausgabe}$$

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

Programmieren mit Funktionen

- Programme werden durch Gleichungen definiert:

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

- Auswertung durch Reduktion von Ausdrücken:

  `addDouble (inc 5) 4` $\rightsquigarrow$ `2*(inc 5+ 4)` $\rightsquigarrow$ `2*((5+1)+ 4)` $\rightsquigarrow$ `20`
- Nichtreduzierbare Ausdrücke sind Werte
  - Vorgebenene Basiswerte: Zahlen, Zeichen
  - Definierte Datentypen: Wahrheitswerte, Listen, ...

Definition von Funktionen

- Zwei wesentliche Konstrukte:
  - Fallunterscheidung
  - Rekursion
- Beispiel:

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

- Auswertung kann divergieren!

Imperativ vs. Funktional

- Imperative Programmierung:
  - Zustandsübergang $\Sigma \to \Sigma$, Lesen/Schreiben von Variablen
  - Kontrollstrukturen: Fallunterscheidung `if ... then ... else`
    Iteration `while ...`
- Funktionale Programmierung:
  - Funktionen $f : E \to A$
  - 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` Zeichenkette
- Verschiedene Typen:
  - Basistypen (Zahlen, Zeichen)
  - strukturierte Typen (Listen, Tupel, etc)
- Wozu Typen?
  - Typüberprüfung während Übersetzung erspart Laufzeitfehler
  - Programmsicherheit

Signaturen

- Jede Funktion hat eine Signatur

```
fac :: Int-> Int

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

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

Übersicht: Typen in Haskell

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

Auswertungsstrategien

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

Auswertungsstrategien

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

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

Ganzzahlige Division, undefiniert für `div n 0`

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

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

Striktheit

Def: Funktion f ist strikt gdw.
Ergebnis ist undefiniert sobald ein Argument undefiniert ist

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

Zusammenfassung

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

Inhalt

- 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:
$\mathtt{f}\ \mathtt{x}_1\ \mathtt{x}_2 \ldots \mathtt{x}_n = \mathtt{E}$

- Geltungsbereich der Definition von `f`:
  alles, was gegenüber `f` eingerückt 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                      f x y =
  | g = P y                 let y = M
  | otherwise = Q where         f x = N x
    y = M                   in  if g then P y
    f x = N x                       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

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

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

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

$$\begin{array}{l} f\ c_1 = e_1 \\ \ldots \\ f\ c_n = e_n \end{array} \longrightarrow \begin{array}{l} f\ x = \texttt{case}\ x\ \texttt{of}\ c_1\ \texttt{->}\ e_1, \\ \ldots \\ c_n\ \texttt{->}\ e_n \end{array}$$

- 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

$\wedge$	$True$	$False$
$True$	$True$	$False$
$False$	$False$	$False$

$$\begin{array}{lcll} True & \wedge & True & = & True \\ True & \wedge & False & = & False \\ False & \wedge & True & = & False \\ False & \wedge & False & = & False \end{array}$$

Wahrheitswerte: `Bool`

- Vordefiniert als

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

- Vordefinierte Funktionen:

```
not :: Bool-> Bool          Negation
&&  :: Bool-> Bool-> Bool   Konjunktion
||  :: Bool-> Bool-> Bool   Disjunktion
```

- Konjunktion definiert wie

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

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

```
if b then p else q  ⟶  case b of  True -> p
                                  False -> q
```

Beispiel: Ausschließende Disjunktion

- Mathematische Definiton:

```
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)

$$\begin{array}{rcl} Date & = & \{Date\ (n,\ m,\ y) \mid n \in \mathbb{N}, m \in Month, y \in \mathbb{N}\} \\ Month & = & \{Jan, Feb, Mar, \ldots\} \end{array}$$

- Funktionsdefinition:
  - Konstruktorargumente sind gebundene Variablen

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

  - 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:

$$\begin{aligned} \texttt{data T} = \; & \texttt{C}_1\ \texttt{t}_{1,1} \ldots \texttt{t}_{1,k_1} \\ & \ldots \\ \mid \; & \texttt{C}_n\ \texttt{t}_{n,1} \ldots \texttt{t}_{n,k_n} \end{aligned}$$

- Konstruktoren $\texttt{C}_1, \ldots, \texttt{C}_n$ sind disjunkt:
  $\texttt{C}_i\ \texttt{x}_1 \ldots \texttt{x}_n = \texttt{C}_j\ \texttt{y}_1 \ldots \texttt{y}_m \Longrightarrow i = j$
- Konstruktoren sind injektiv:
  $\texttt{C}\ \texttt{x}_1 \ldots \texttt{x}_n = \texttt{C}\ \texttt{y}_1 \ldots \texttt{y}_n \Longrightarrow \texttt{x}_i = \texttt{y}_i$
- Konstruktoren erzeugen den Datentyp:
  $\forall x \in T.\, x = \texttt{C}_i\ \texttt{y}_1 \ldots \texttt{y}_m$

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

Haskell bietet die Auswahl:

- `Int` - ganze Zahlen als Maschinenworte ($\geq$ 31 Bit)
- `Integer` - beliebig große ganze Zahlen
- `Rational` - beliebig genaue rationale Zahlen
- `Float` - 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 `(div x y)*y + 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, Exponentation, 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

  $\rightsquigarrow$ 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 = `\x -> E` für `f x` = E

Typinferenzregeln

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

$$\frac{A, x :: s \vdash e :: t}{A \vdash \backslash x \text{ -> } e :: s \text{ -> } t}\ Abs$$

$$\frac{A \vdash e :: s \text{ -> } t \quad A \vdash e' :: s}{A \vdash e\ e' :: t}\ App$$

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

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

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

`++`	`[a]-> [a]-> [a]`	Verketten
`!!`	`[a]-> Int-> a`	n-tes Element selektieren
`concat`	`[[a]]-> [a]`	"flachklopfen"
`length`	`[a]-> Int`	Länge
`head, last`	`[a]-> a`	Erster/letztes Element
`tail, init`	`[a]-> [a]`	(Hinterer/vorderer) Rest
`replicate`	`Int-> a-> [a]`	Erzeuge `n` Kopien
`take`	`Int-> [a]-> [a]`	Nimmt ersten `n` Elemente
`drop`	`Int-> [a]-> [a]`	Entfernt erste `n` Elemente
`splitAt`	`Int-> [a]-> ([a], [a])`	Spaltet an `n`-ter Position
`reverse`	`[a]-> [a]`	Dreht Liste um
`zip`	`[a]-> [b]-> [(a, b)]`	Paare zu Liste von Paaren
`unzip`	`[(a, b)]-> ([a], [b])`	Liste von Paaren zu Paaren
`and, or`	`[Bool]-> Bool`	Konjunktion/Disjunktion
`sum`	`[Int]-> Int` (überladen)	Summe
`product`	`[Int]-> Int` (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 = letzer 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` $\leadsto 7$

Funktionen Höherer Ordnung

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

Funktionen als Argumente: Funktionskomposition

- Funktionskomposition

```
(.) :: (b-> c) -> (a-> b)-> a-> c
(f . g) x = f (g x)
```

  - Vordefiniert
  - Lies: `f` nach `g`
- Funktionskomposition vorwärts:

```
(>.>) :: (a-> b)-> (b-> c)-> a-> 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 `\n-> mod n p /= 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 -> b-> c` ≡ `a-> (b-> c)`

- Inbesondere:

  `(a -> b)-> c` ≠ `a-> (b-> c)`

- Partielle Anwendung von Funktionen:
  - Für `f :: a-> b-> c`, `x :: a` ist `f x :: b-> c`
- Beispiele:
  - `map toLower :: String-> String`
  - `3 == :: Int-> Bool`
  - `concat . map (replicate 2) :: String-> String`

Die Kürzungsregel

Bei Anwendung der Funktion

$$\texttt{f :: } t_1 \texttt{ -> } t_2 \texttt{ -> ... -> } t_n \texttt{-> t}$$

auf k Argumente mit $k \leq n$

$$e_1 \texttt{ :: } t_1\texttt{, } e_2 \texttt{ :: } t_2\texttt{, ..., } e_k \texttt{ :: } t_k$$

werden die Typen der Argumente gekürzt:

$$\begin{aligned}&\texttt{f :: } \not{t}_1 \texttt{ -> } \not{t}_2 \texttt{ -> ... -> } \not{t}_k \texttt{ -> } t_{k+1} \texttt{ -> ...-> } t_n \texttt{ -> t}\\ &\qquad\texttt{f } e_1 \texttt{ ...} e_k \texttt{ :: } t_{k+1} \texttt{ -> ...-> } t_n \texttt{ -> t}\end{aligned}$$

- 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:

```
f []     = A
f (x:xs) = x ⊗ f xs
```

- Parameter der Definition:
  - Startwert (für die leere Liste) `A :: b`
  - Rekursionsfunktion ⊗ `:: a -> b-> b`
- Auswertung:

  `f[x1, ..., xn]` = `x1` ⊗ `x2` ⊗ ... ⊗ `xn` ⊗ `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:
  $\texttt{foldr} \otimes [x_1, \ldots, x_n]\ A = x_1 \otimes (x_2 \otimes (\ldots (x_n \otimes A)))$
- Warum nicht andersherum?
  $\texttt{foldl} \otimes [x_1, \ldots, x_n]\ A = (((A \otimes x_1) \otimes x_2) \ldots) \otimes x_n$
- 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

- $\texttt{f} = \texttt{foldr} \otimes A$ entspricht

```
f []     =  A
f (x:xs) =  x ⊗ f xs
```

  - Kann nicht-strikt in `xs` sein
- $\texttt{f} = \texttt{foldl} \otimes A$ entspricht

```
f xs       =  g A xs
g a []     =  a
g a (x:xs) =  g (a ⊗ x) xs
```

  - 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

$$A \otimes x = x \quad \text{(Neutrales Element links)}$$
$$x \otimes A = x \quad \text{(Neutrales Element rechts)}$$
$$(x \otimes y) \otimes z = x \otimes (y \otimes z) \quad \text{(Assoziativität)}$$

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

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

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-> tl);
    if (f(l-> hd)) {
       l->tl= r;
       return l;
       }
    else {
       free(l);
       return r;
} }    }
```


Übersicht: vordefinierte Funktionen auf Listen II

`map`	`(a-> b)-> [a]-> [b]`	Auf alle anwenden
`filter`	`(a-> Bool)-> [a]-> [a]`	Elemente filtern
`foldr`	`(a -> b -> b) -> b -> [a] -> b`	Falten von rechts
`foldl`	`(a -> b -> a) -> a -> [b] -> a`	Falten von links
`takeWhile`	`(a -> Bool) -> [a] -> [a]`	Längster Prefix s.t. `p` gilt
`dropWhile`	`(a -> Bool) -> [a] -> [a]`	Rest davon
`any`	`(a-> Bool)-> [a]-> Bool`	`any p = or . map p`
`all`	`(a-> Bool)-> [a]-> Bool`	`all p = and . map p`
`elem`	`Eq a=> a-> [a]-> Bool`	`elem x = any (x ==)`
`zipWith`	`(a -> b -> c) -> [a] -> [b] -> [c]`	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
2. Programmentwurf:
   - Wie kann das Problem in Teilprobleme zerlegt werden?
   - Wie können Teillösungen zusammengesetzt werden?
   - Gibt es ein ähnliches (gelöstes) Problem?

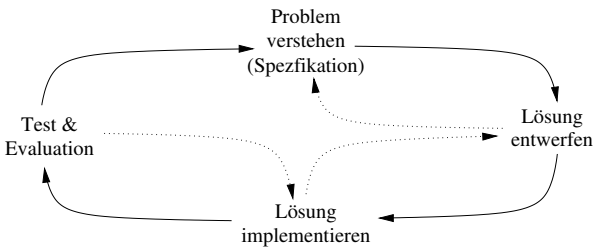
   ⇝ Erster Entwurf

Funktionaler Entwurf und Entwicklung

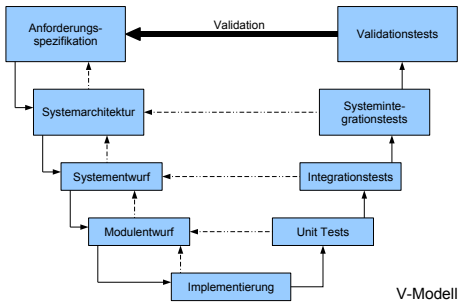
3. Implementierung:
   - Effizienz
   - Wie würde man Korrektheitheit zeigen?
   - Termination
   - Gibt es hilfreiche Büchereifunktionen
   - Refaktorierung: mögliche Verallgemeinerungen, shared code

   ⇝ 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



V-Modell

1. Beispiel: größter gemeinsame Teiler

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

ggT: Spezifikation

- Signatur

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

- Eigenschaften (ausführbare Spezifikationen) formulieren
  - Problem: Existenzquantor — besser: $a \mid b \iff b \mod a = 0$

```
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: $a < b \rightsquigarrow \text{gcd}(a, b) = \text{gcd}(a, b - a)$
- Besser: $a < b \rightsquigarrow \text{gcd}(a, b) = \text{gcd}(b - a, a)$
- Implementierung:

```
gcd a b
  | a == b    = a
  | a < b     = gcd (b- a) a
  | otherwise = gcd 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 gcd$(a, b) =$ gcd$(b, a \mod b)$ (Euklid'scher Algorithmus)
- Es gibt eine Büchereifunktion.

2. Beispiel: das n-Königinnen-Problem

- Problem: n Königinnen auf $n \times n$-Schachbrett sicher plazieren
- 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
```

```
q && all allSafe q
```

- Schon fast eine Lösung, aber kombinatorische Explosion

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 plaziert werden.
  - Limitiert kombinatorische Explosion

n-Königinnen: Hauptfunktion

- Hauptfunktion:
  - Sei `p` neue Zeile
  - `cand p` bisherige Teillösungen, die mit `(n, p)` sicher sind
  - `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
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 unerschiedlichen 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 `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

- Traversion: 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

y

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

Signaturen

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
  - in Speicher an eine Stelle einen Wert schreiben: $S \to a \to b \to S$
  - aus Speicher an einer Stelle einen Wert lesen: $S \to a \rightharpoonup b$ (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

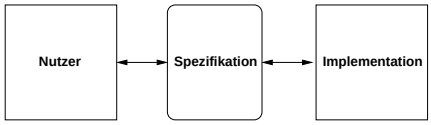
- Axiome sind Prädikate über den Operationen der Signatur
  - Elementare Prädikate P
  - Gleichheit $s == t$
  - Bedingte Prädikate: $A \Longrightarrow B$
- Beobachtbare Typen: interne Struktur bekannt
  - Vordefinierte Typen (Zahlen, Zeichen, Listen), algebraische Datentypen
- Abstrakte Typen: interne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

Beispiel: Speicher

- Beobachtbar: Adressen und Werte, abstrakt: Speicher
- Axiome für das Lesen:
  - 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:
    `a1 /= a2` $\Longrightarrow$ `get (upd s a1 v) a2 == get s a2`
- Axiome für das Schreiben:
  - Schreiben an dieselbe Stelle überschreibt alten Wert:
    `upd (upd s a v) a w == upd s a w`
  - Schreiben über verschiedene Stellen kommutiert (Reihenfolge irrelevant):
    `a1 /= a2` $\Longrightarrow$ `upd (upd s a1 v) a2 w == 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

- Typsynomyme 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 $f : A \rightharpoonup B$

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

- Invariante der Liste: für jedes a höchsten ein Paar (a, v)
- Leerer Speicher: leere Menge
- Lesen von a:
  - Wenn Paar (a, v) in Menge, `Just v`, ansonsten $\bot$
- 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 `==>` 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
  - `==>` 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)          Assoziativiät 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 ≥ 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: `map f (map g xs) == map (f. g) xs`

Ind: 1. Induktionssbasis

```
    map f (map g [])
=   map f []                  Def. map für []
=   []                        Def. map für []
=   map (f.g) []              Def. map für []
```

2. Induktionsschritt

```
    map f (map g (x:xs))
=   map f (g x: map g xs)     Def. map für x:xs
=   f (g x): map f (map g xs) Def. map für x:xs
=   f (g x): map (f. g) xs    Induktionsvoraussetzung
=   (f. g) x: map (f. ) xs    Def. .
=   map (f. g) (x:xs)         Def. map für x:xs
```

Weitere Beispiele

```
length (filter p xs)  <=  length xs                (1)

length (xs++ ys)  ==  length xs + length ys        (2)

map f (xs++ ys)  ==  map f xs ++ map f ys          (3)

sum (map length xs)  ==  length (concat xs)        (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: $P($`Null`$)$
- Induktionsschritt:
  Voraussetzung $P(s)$, $P(t)$, zu zeigen $P($`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                                   Def. fmap
=   []                                             Def. inorder
=   map f [])                                      Def. map
=   map f (inorder Null)                           Def. inorder
```

2. Induktionsschritt

```
    inorder (fmap f (Node s a t))
=   inorder (Node (fmap f s) (f a) (fmap f t))     Def. fmap
=   inorder (fmap f s) ++ [f a] ++ inorder (fmap f t)   Def. inorder
=   map f (inorder s) ++ [f a] ++ map f (inorder t)     Induktionsvoraussetzung
=   map f (inorder s) ++ map f [a] ++ map f (inorder t) Def. map
=   map f (inorder s ++ [a] ++ inorder t)           Lemma (zweimal)
=   map f (inorder (T s a t))                      Def. inorder
```

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 14.01.09: Datenmodellierung mit XML

Fahrplan

- Teil I: Grundlagen
- Teil II: Abstraktion
- Teil III: Beispiele, Anwendungen, Ausblicke
  - Datenmodellierung mit XML
  - Effizienzerwägungen
  - Grafik
  - Schluss

Inhalt

- Fallbeispiel: XML Feeds verarbeiten
- Protokolle und Standards:
  - XML
  - HTTP
  - RSS
- Haskell-Büchereien dazu

XML

Die Extensible Markup Language (engl. für "erweiterbare Auszeichnungssprache", abgekürzt XML, ist eine Auszeichnungssprache zur Darstellung hierarchisch strukturierter Daten in Form von Textdaten. XML wird u. a. für den Austausch von Daten zwischen Computersystemen eingesetzt, speziell über das Internet.

Wikipedia

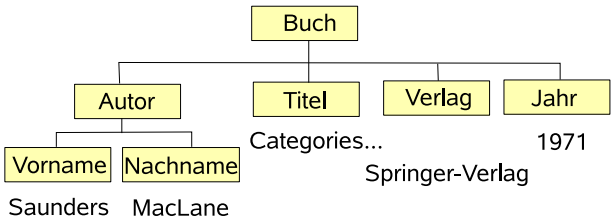
- Textbasiert
- Erweiterbar und anpassbar
- Generisch: unabhänging von Betriebssystem, Programmiersprache, Anwendung

Ein einfaches Beispiel

- Ein Buch hat
  - Autor(en) mit Vorname, Nachname (mind. einen)
  - Titel
  - Verlag, Erscheinungsjahr
  - Zusammenfassung (optional)
- Als XML:

```
<buch>
  <autor><vorname>Saunders</vorname>
         <nachname>MacLane</nachname></autor>
  <titel>Categories for the working mathematician</titel>
  <verlag>Springer-Verlag</verlag><jahr>1971</jahr>
</buch>
```

XML-Dokumente als Bäume



Mixed content

- Beispiel: Zusammenfassung mit ausgezeichneten Schlüsselwörtern

```
<zusammenfassung>
Das <keyword>Lehrbuch</keyword> über
<keyword>Kategorientheorie</keyword>,
eine abstrakte Disziplin der <keyword>Mathematik</keyword>
</zusammenfassung>
```

- Leerzeichen relevant (nur in mixed content)

Namen und Attribute

- Namen:
  - Folge von Buchstaben, Ziffern, Zeichen (keine Sonderzeichen)
  - Fängt an mit Buchstabe, Unterstrich, Zeichen
- Attribute:
  - Paar aus Namen und Werten
  - E.g. `<titel sprache="englisch">...</titel>`

XML-Schichten

1. Wohlgeformtheit (well-formedness)
   - Start-tag $\longleftrightarrow$ end-tag
   - Keine überlappenden Elemente
   - Genau ein Wurzelelement
   - Attribut-Werte in Anführungszeichen
   - Keine zwei Attribute mit gleichen Werten
   - $\Longrightarrow$ Text entspricht Grammatik, parsierbar
2. Gültigkeit (validity)
   - Dokument entspricht Dokumententyp
3. Spezifische Dokumemtentypen
   - E.g. `XHTML`, `SVG`, `XSD`

Dokumententypen

- Gibt Produktionsregeln für Elemente an
- Verschiedene Formate: DTDs, XML-Schema, Relax NG

DTD für das Buch

```
<!DOCTYPE buch [
  <!ELEMENT buch (autor+, titel, verlag, jahr, zusfassung?)>
  <!ELEMENT autor (vorname, nachname)>
  <!ELEMENT vorname (#PCDATA)>
  <!ELEMENT nachname (#PCDATA)>
  <!ELEMENT title (#PCDATA)>
  <!ATTLIST title language CDATA #IMPLIED>
  <!ELEMENT zusfassung (#PCDATA|keyword)*>
  <!ELEMENT keyword (#PCDATA)>
]>
```

RelaxNG-Schema für das Buch

```
buch    = element buch    { autor+, titel, verlag, jahr
                          , zusfassung? }
autor   = element autor   { element vorname { text }
                  , element nachname { text } }
titel   = element titel   { attribute language { text }?
                          , text }
verlag = element verlag { text }
jahr    = element jahr    { xsd:integer }
zusfassung = element zusfassung { (text
                                 | element keyword {text})* }
```

XML-Schema für das Buch (Auszug)

```
<xs:element name="buch">
  <xs:complexType>
    <xs:sequence>
      <xs:element maxOccurs="unbounded" ref="autor"/>
      <xs:element ref="titel"/>
      <xs:element ref="verlag"/>
      <xs:element ref="jahr"/>
      <xs:element minOccurs="0" ref="zusfassung"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="autor">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="vorname"/>
      <xs:element ref="nachname"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Modellierung von XML in Haskell

- Ungetypt (für wohlgeformte Dokumente)
  - Generischer Dokumententyp
- Getypt (bezüglich eines Dokumententyps)
  - DTD/RNC ⟶ Haskell-Typen

HaXml

- Getypte Einbettung in Haskell
  - Generischer XML-Parser
  - DTD → algebraischen Datentyp
  - Klasse `XmlContent`
  - Funktionen

    ```
    readXml :: XmlContent a => String -> Maybe a
    showXml :: XmlContent a => a -> String
    ```

  - Jedes Element einen Typ, Instanz von `XmlContent`

HaXml: Übersetzte DTD

```
data Buch = Buch (List1 Autor) Titel Verlag Jahr
                 (Maybe Zusfassung)
data Autor = Autor Vorname Nachname
data Vorname = Vorname String
data Nachname = Nachname String
data Titel = Titel Titel_Attrs String
data Titel_Attrs = Titel_Attrs
                   { titelLanguage :: (Maybe String) }
data Zusfassung = Zusfassung [Zusfassung_]
data Zusfassung_ = Zusfassung_Str String
                 | Zusfassung_Keyword Keyword
data Keyword = Keyword String
```

Einschub: Labelled Records

- Algebraischer Datentyp mit benannten Feldern
- Beispiel:

  ```
  data Book = Book { author :: String
                   , title  :: String
                   , publisher :: String }
  ```

- Konstruktion (Reihenfolge der Argumente irrelevant)

  ```
  b = Book { author = "S. MacLane"
           , publisher = "Springer-Verlag"
           , title  = "Categories" }
  ```

Selektion, Update und Patternmatching

- Selektion durch Feldnamen:

```
publisher b --> "Springer-Verlag"
author b    --> "S. MacLane"
```

- Update:

```
b{publisher = "Rowohlt Verlag"}
```

  - Rein funktional! (b bleibt unverändert)

- Patternmatching:

```
print :: Book-> IO ()
print (Book{author= a, publisher= p, title= t}) =
  putStrLn (a++ " schrieb "++ t ++ " und "++
            p++ " veroeffentlichte es.")
```

Ein einfacher XML-Parser

- Einfacher Parser:

```
parseXMLDoc  :: String -> Maybe Element
```

- Generisches Content-Modell:
  - Element hat
    - Namen
    - Liste von Attributen,
    - mehrere Inhalte;
  - Inhalt kann sein
    - ein Element,
    - Text,
    - Entitätenreferenzen.

Die Datentypen: Content

```
data Content
= Elem Element
| Text CData
| CRef String
```

- CRef "ref" ist &ref;

Die Datentypen: Text

```
data CData = CData {
   cdVerbatim :: CDataKind
   cdData :: String
   cdLine :: Maybe Line
   }
data CDataKind = CDataText
               | CDataVerbatim
               | CDataRaw
```

- CDataText ist #PCDATA oder text
- CDataVerbatim ist <![CDATA[...]]>
- CDataRaw ist <!...!>

Die Datentypen: Element

```
data Element = Element {
  elName    :: QName
  elAttribs :: [Attr]
  elContent :: [Content]
  elLine    :: Maybe Line
  }
```

- Qname: qualifizierter Name

```
data Attr = Attr {
  attrKey :: QName
  attrVal :: String
  }
```

Qualifizierte Namen: QName

- Namensräume: Abgrenzung der Namen in verschiedenen Dokumententypen
- Namensraumdeklaration bindet Präfix an URI
- Beispiel:

```
<book xmlns:mybook="http://www.informatik.uni-bremen.de/~cxl/lehre
  <autor><mybook:vorname>Saunders</mybook:vorname>
         <mybook:nachname>MacLane</mybook:nachname>
  </autor>
...
</book>
```

- In Haskell:

```
data QName = QName {     qName   :: String,
   qURI :: Maybe String, qPrefix :: Maybe String }
```

XML in Haskell: Beispiel

```
Prelude Text.XML.Light> parseXMLDoc
             "<book><title lang=\"deutsch\">Beispiel</title></book>"
Loading package xml-1.3.3 ... linking ... done.
Just (Element {elName = QName {qName = "book",
                               qURI = Nothing, qPrefix = Nothing},
               elAttribs = [], elContent =
 [Elem (Element {elName = QName {qName = "title",
                                 qURI = Nothing, qPrefix = Nothing},
                 elAttribs =
    [Attr {attrKey = QName {qName = "lang",
                            qURI = Nothing, qPrefix = Nothing},
           attrVal = "deutsch"}],
                 elContent =
    [Text (CData {cdVerbatim = CDataText,
                  cdData = "Beispiel",
                  cdLine = Just 1})],
                 elLine = Just 1})],
               elLine = Just 1})
```

Beispiel: RSS-Feeds

- Daten aus RSS-Feeds lesen, bearbeiten, ausgeben
- Eingabeformat: RSS
- Augabeformat: XHTML
- Verarbeitung: Haskell
- Transportprotokoll: HTTP

HTTP

- Hypertext Transfer Transport
- Definiert in RFC 2616
- Vier grundlegende Befehle:
  - `GET` — Ressource lesen
  - `PUT` — Ressource erzeugen
  - `POST` — Ressource schreiben
  - `DELETE` — Ressource löschen
- Verführerisch einfach
- Kann beliebigen Inhalt übertragen: HTML, XML, ...

Very Simple HTTP

- Client (e.g. Web-Browser) sendet Anforderungen (Requests):

```
$ telnet www.informatik.uni-bremen.de 80
Trying 134.102.224.17...
Connected to www.informatik.uni-bremen.de.
Escape character is '^]'.

GET http://www.informatik.uni-bremen.de/~cxl/lehre/pi3.ws08/ 1.1
```

- Server sendet Antworten (Responses):

```
HTTP/1.1 200 OK
Date: Wed, 14 Jan 2009 07:52:52 GMT
Server: Apache
Content-Type: text/html; charset=iso-8859-1
Connection: close

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
  <title>[03-05-G-700.03] Praktische Informatik 3</title>
  <link REL="SHORTCUT ICON" HREF="http://www.informatik.uni-bremen.de/ag
```

Die HTTP-Bücherei

- Einfaches `GET` mit

```
simpleHTTP :: Request -> IO (Result Response)
```

- Beispiel für Benutzung (siehe `Get.hs`)

```
readURI uri = do
  resp <- simpleHTTP (request uri)
  case resp of
    Right resp ->
      case rspCode resp of
        (2,0,0) -> do hPutStrLn stderr ("Successfully read "++ show uri)
                      return (rspBody resp)
        _ -> error (httpError resp)
    Left connerr -> error (show connerr)
```

RSS

- RSS ist ein XML-Format für Schlagzeilen (Kurznachrichten)
- Verschieden Formate: RSS 0.9x, 1.0, 2.0, Atom
- Struktur (2.0):
  - Ein Feed enthält Liste von Channels
  - Ein Channel enthält Name, Link, Liste von Items
  - Ein Item enthält Titel, Link, Beschreibung
- Ein einfaches Beispiel

RSS in Haskell

```
data RSSChannel = RSSChannel { chTitle       :: String
                             , chLink        :: String
                             , chItems       :: [RSSItem]
                             } deriving Show

data RSSItem = RSSItem { itemTitle :: String
                       , itemLink  :: String
                       , itemDescr :: String
                       } deriving Show
```

RSS in Haskell — vereinfacht

Kanal hat nur Anzahl der Artikel

```
data RSSChannel = RSSChannel { chTitle       :: String
                             , chLink        :: String
                             , chItems       :: Int
                             } deriving Show
```

Übersetzung XML nach Haskell

- Kernfunktionalität: `String -> [RSSChannel]`
- Mit `parseXMLDoc` parsieren, gibt `Element`
- Top-Element:
  - Muss Name `RSS` haben
  - Wenn Attribut version, dann `2.0` oder `2.0.1`
  - Ansonsten Liste von Kanälen parsieren

Kanäle parsieren

```
getRSSChannels :: Element-> Either String [RSSChannel]
getRSSChannels (Element{elName= qn, elAttribs= as
                        , elContent= cont})
  | qne qn "rss" =
      case lookupAttr (qname "version") as of
        Just v | v == "2.0" || v == "2.0.1"
               -> Right (getChannels cont)
        Just v -> Left ("Wrong RSS version "++ v)
        Nothing -> Right (getChannels cont)
  | otherwise = Left "Not an RSS feed (no top RSS element)"

qne :: QName-> String-> Bool
qne qn str = qName qn == str
```

Einen Kanal übersetzen

- Kernfunktionalität: `Content-> Maybe RSSChannel`
- `Content` muss `Element` sein
- Pro `Element`:
  - Aus Elementen `title` und `link` Titel und Link extrahieren
  - Aus Element `item` Liste von Items extrahieren
- Wird dann erweitert zu

```
getChannels :: [Content]-> [RSSChannel]
```

Kanal parsieren

```
getChannel :: Content-> Maybe RSSChannel
getChannel (Elem e)
  | qne (elName e) "channel" =
       Just (foldr getChannelData mtChannel (elContent e))
getChannel _ = Nothing

mtChannel = RSSChannel "" "" 0


getChannelData :: Content -> RSSChannel-> RSSChannel
```

Kanal parsieren

```
getChannelData :: Content -> RSSChannel-> RSSChannel
getChannelData (Elem e) ch
  | qne (elName e) "title" = ch{chTitle = strContent e}
  | qne (elName e) "item"  = ch{chItems = 1+ chItems ch}
  | qne (elName e) "link"  = ch{chLink  = strContent e}
getChannelData _ ch = ch
```

Verarbeitung

- Aufgabe: Anzahl aller Artikel zählen
- Liste der Kanäle traversieren, `chItems` addieren

```
processChannels :: [RSSChannel]-> Int
processChannels chs = sum (map chItems chs)
```

Hilfsfunktionen

- Textinhalt aus mixed content extrahieren

```
strContent  :: Element-> String
```

- Element nach Name suchen

```
findElement :: QName -> Element -> Maybe Element
```

Darstellung

- Darstellung der Ergebnisse als `XHTML`
- Damit Nutzung der XML-Bücherei möglich
- Zwei Hilfsfunktionen
  - Einfaches Element (ohne Attribute) erzeugen:

    ```
    selem :: String-> [Content]-> Element
    ```

  - Einfachen Text-Content erzeugen:

    ```
    text :: String-> Content
    ```

HTML-Header erzeugen

```
htmlHeader :: String-> [Content]-> Element
htmlHeader tn cont =
  Element{ elName=qname "html"
         , elAttribs= [sattr "xmlns"
                             "http://www.w3.org/1999/xhtml"]
         , elContent= map Elem [title, body]
         , elLine= Nothing
         } where
    title = selem "head" [Elem (selem "title" [text tn])]
    body  = selem "body" cont
```

Eine ganz einfache HTML-Seite

```
showResult :: Int-> String
showResult n =
  ppTopElement (htmlHeader "Ergebnis" [
    text "Es wurden ",
    Elem (selem "b" [text (show n)]),
    text " Artikel gefunden." ])
```

Alles zusammensetzen

- Feeds aus Kommandozeile lesen (`getArgs`)

```
getArgs :: IO [String]
```

- Feeds lesen und verarbeiten

```
readURL :: String-> IO String
processFeed :: String-> IO [RSSChannel]
processChannels :: [RSSChannel]-> Int
```

- Ergebnis mit `showResult` anzeigen, ausgeben

```
showResult :: Int-> String
```

- Zusammengesetzt:

```
main = do
  feeds <- getArgs
  chs   <- mapM processFeed feeds
  putStrLn (showResult (processChannels (concat chs)))
```

Zusammenfassung

- XML modelliert Daten, Verarbeitung funktional
- XML und HTTP für Haskell:
  - `HTTP-3001.0.0`
  - `xml-1.3.3`
  - Müssen installiert werden
- Beispielprogramm zur Verarbeitung von RSS-Feeds

Vorlesung vom 21.01.09: Effizienzaspekte

Fahrplan

- Teil I: Grundlagen
- Teil II: Abstraktion
- Teil III: Beispiele, Anwendungen, Ausblicke
  - Datenmodellierung mit XML
  - Effizienzerwägungen
  - Grafik
  - Schluss

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
- Effizenzverbesserungen durch
  - Endrekursion: Iteration in funktionalen Sprachen
  - Striktheit: Speicherlecks vermeiden (bei verzögerter Auswertung)
- Vorteil: Effizienz muss nicht im Vordergrund stehen

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: `if` oder `case`
- Entspricht `goto` oder `while` in imperativen Sprachen.
- Wird in Sprung oder Schleife übersetzt.
- Nur nicht-endrekursive Funktionen brauchen Platz auf dem Stack.

Beispiele

- `fac'` nicht endrekursiv:

```
fac' :: Integer-> Integer
fac' n  = if n == 0 then 1 else n * fac' (n-1)
```

- `fac` endrekursiv:

```
fac :: Integer-> Integer
fac n     = fac0 n 1 where
  fac0 :: Integer-> Integer-> Integer
  fac0 n acc = if n == 0 then acc
               else fac0 (n-1) (n*acc)
```

- `fac'` verbraucht Stackplatz, `fac` nicht.

Beispiel: Listen umdrehen

- Liste umdrehen, nicht endrekursiv:

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

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

- 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..20000])`

Überführung in Endrekursion

- Gegeben Funktion

```
f': S-> T
f' x = if B x  then H x
               else φ (f' (K x)) (E x)
```

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

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

- Dann ist endrekursive Form:

```
f: S-> T
f x = g x e where
   g x y = if B x  then φ (H x) y
                   else g (K x) (φ (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:

$K(x)$	$\mapsto$	`tail`	$B(x)$	$\mapsto$	`null` x
$E(x)$	$\mapsto$	1	$H(x)$	$\mapsto$	0
$\phi(x,y)$	$\mapsto$	$x+y$	e	$\mapsto$	0

- 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)
```

"Unendliche" Listen

- Listen müssen nicht endlich repräsentierbar sein:
  - Beispiel: definiert "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]
    cycle xs = xs ++ cycle xs
    ```

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

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 (vgl. alte Version)

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

- Von allen Primzahlen die ersten:

```
nprimes :: Int-> [Integer]
nprimes 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: exponentieller Aufwand.

Bsp: Fibonacci-Zahlen

- Lösung: zuvor berechnete Teilergebnisse wiederverwenden.
- Sei `fibs :: [Integer]` Strom aller Fib'zahlen:

```
             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`, `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!
- Eine Funktion hat ein Speicherleck, wenn Speicher unnötig lange im Zugriff bleibt.
  - "Echte" Speicherlecks wie in C/C++ nicht möglich.
- Beispiel: `getLines`, `fac`
  - Zwischenergebnisse werden nicht auswertet.
  - Insbesondere ärgerlich bei nicht-terminierenden Funktionen.

Striktheit

- Strikte Argumente erlauben Auswertung vor Aufruf
  - Dadurch konstanter Platz bei Endrekursion.
- Erzwungene Striktheit:
  - `seq :: a-> b-> b` erzwingt Striktheit im ersten Argument:

```
⊥ `seq` b = ⊥
a `seq` b = b
```

  - `($!) :: (a-> b)-> a-> b` strikte Funktionsanwendung
  - `ghc` macht Striktheitsanalyse

```
f $! x = x `seq` f x
```

- Fakultät in konstantem Platzaufwand

```
fac2 n = fac0 n 1 where
  fac0 n acc = seq acc $ if n == 0 then acc
                         else fac0 (n-1) (n*acc)
```


`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' :: (a-> b-> a)-> a-> [b]-> a` ist strikt, endrekursiv.
- Für Monoid (ϕ, e) gilt

$$\texttt{foldr}\ \phi\ e\ l = \texttt{foldl}\ (\texttt{flip}\ \phi)\ e\ l$$

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
```

  - Unendliche Listen immer mit `foldr` falten.

Gemeinsame Teilausdrücke

- Ausdrücke werden intern durch Termgraphen dargestellt.
- Argument wird nie mehr als einmal ausgewertet:

```
f :: Int-> Int-> Int
f x y = x+ x
```

  - Beispiel: `f (trace "Eins" (3+2)) (trace "Zwei" (2+7))}`
- Sharing von Teilausdrücken
  - Explizit mit `where` und `let`
  - Implizit (`ghc`, optimierend):

```
double :: Int-> Int
double x = trace "Foo!" 2*x
```

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: `facs` hat den Typ `Num a=> a-> a`

```
facs n = if n == 0 then 1 else n* facs (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 Standardbücherei)
  - 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
- Datenstrukturen müssen nicht endliche repräsentierbar sein
- Überladene Funktionen sind langsam.
- Listen sind keine Felder oder endliche Abbildungen.
- Effizienz muss nicht immer im Vordergrund stehen.