

Praktische Informatik 3
Einführung in die Funktionale Programmierung
Vorlesung vom 05.11.2008:
Funktionen und Datentypen

Christoph Lüth

WS 08/09



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

- 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 **ingerückt** 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

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

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

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

Aufzählung und Fallunterscheidung in Haskell

- **Definition**

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

- Implizite Deklaration der 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</code>	<code>:: Bool -> Bool</code>	Negation
<code>&&</code>	<code>:: Bool -> Bool -> Bool</code>	Konjunktion
<code> </code>	<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` \rightsquigarrow `False`

- `if then else` als syntaktischer Zucker:

```
if b then p else q  $\longrightarrow$  case b of True -> p  
                                     False -> q
```

Beispiel: Ausschließende Disjunktion

- Mathematische Definition:

```
exOr :: Bool -> Bool -> Bool
```

```
exOr x y = (x || y) && (not (x && y))
```

- Alternative 1: explizite Wertetabelle:

```
exOr False False = False
```

```
exOr True  False = True
```

```
exOr False True  = True
```

```
exOr True  True  = False
```

- Alternative 2: Fallunterscheidung auf ersten Argument

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

```
exOr False y = y
```

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

Produkte

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

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

- **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 $S n \in \mathbb{N}$
 - S injektiv und $S n \neq 0$
 - 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