Funktionales Programmieren
(Praktische Informatik 3)

Berthold Hoffmann
Studiengang Informatik
Universität Bremen

Winter 03/04

Vorlesung vom 27.10.2003: Funktionen und Typen

Inhalt

- Wie definiere ich eine Funktion?
  - Einige syntaktische Feinheiten
  - Von der Spezifikation zum Programm
- Basisdatentypen:
  Wahrheitswerte, Zahlen und Zeichen (-ketten)
- Strukturierte Datentypen:
  Tupel und Listen

Dank

We are great only because we are standing on the shoulders of giants. Isaac Newton

Diese Vorlesung wurde von Christoph Lüth konzipiert.

- Alles Gute stammt von Christoph.
- Fehler usw. stammen von mir.
  (Fehlermeldungen und Kritik an `hof@tzi.de`)

Danke, Riese Christoph!

Korrektur: Termine

Tutorien:

. . .

Mi 13-15 MZH 1380 Pascal Schmidt

. . .

Korrektur: Termine

Tutorien:

. . .

Mi 13-15 MZH 1380 Pascal Schmidt

. . .

SIE oder DU?

- Wer mich duzt, den duze ich zurück!

Definition von Funktionen

Funktionsdefinition

- Signatur (Kopf, Profil)

```
fac :: Int-> Int
```

- Definition (Rumpf, Gleichung)

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

- Zum Vergleich die mathematische Schreibweise:

$$\mathrm{fac} \quad : \quad \mathbb{Z} \to \mathbb{Z}$$

$$\forall n \in \mathbb{Z} : \mathrm{fac}(n) \quad = \quad \begin{cases} 1 & \text{wenn } n = 0 \\ n \times \mathrm{fac}(n-1) & \text{sonst} \end{cases}$$

Funktionsdefinition mit bedingten Gleichungen

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

```
fac n | n <  0    = - fac(-n)
      | n == 0    = 1
      | otherwise = n * fac(n-1)
```

- Bedingungen werden von oben nach unten ausgewertet
- `otherwise` (= `True`) vermeidet Laufzeitfehler (Wenn keine Bedingung wahr ist)

Abseits! offside rule

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

- Alles, was gegenüber `f` eingerückt ist, gehört noch zur Definition von `f`.
- **Beispiel:**

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

- Gilt auch bei verschachtelten Definitionen.

Kommentare

```
{- Fakultaetsfunktion fuer ganze Zahlen
   (c) 2003 Berthold Hoffmann                    -}
fac :: Int -> Int                    -- Signatur
fac n
   | n <  0    = error "negative fac-argument"     -
   | n == 0    = 1
   | otherwise = n * fac (n-1)
```

Kommentare

```
{- Fakultaetsfunktion fuer ganze Zahlen
   (c) 2003 Berthold Hoffmann                    -}
fac :: Int -> Int                    -- Signatur
fac n
   | n <  0    = error "negative fac-argument"    -
   | n == 0    = 1
   | otherwise = n * fac (n-1)
```

- zeilenweise: von `--` bis zum Zeilenende

Kommentare

```
{- Fakultaetsfunktion fuer ganze Zahlen
   (c) 2003 Berthold Hoffmann                    -}
fac :: Int -> Int                    -- Signatur
fac n
   | n <  0    = error "negative fac-argument"    -
   | n == 0    = 1
   | otherwise = n * fac (n-1)
```

- zeilenweise: von `--` bis zum Zeilenende
- mehrzeilig: zwischen `{-` und `-}`, auch geschachtelt

Funktionaler Entwurf und Entwicklung

- Spezifikation:
  - Definitionsbereich (Eingabewerte)
  - Wertebereich (Ausgabewerte)
  - Vor/Nachbedingungen?

  $\rightsquigarrow$ Signatur

Funktionaler Entwurf und Entwicklung

- Spezifikation:
  - Definitionsbereich (Eingabewerte)
  - Wertebereich (Ausgabewerte)
  - Vor/Nachbedingungen?

  $\rightsquigarrow$ Signatur
- Programmentwurf:
  - Gibt es ein ähnliches (gelöstes) Problem?
  - Wie kann das Problem in Teilprobleme zerlegt werden?
  - Wie können Teillösungen zusammengesetzt werden?

  $\rightsquigarrow$ Erster Entwurf

- Implementierung:
  - Termination?
  - Effizienz? Geht es besser? Oder allgemeiner?
  - Gibt es hilfreiche Bibliotheksfunktionen?
  - Wie könnte man die Korrektheitheit zeigen?

  $\rightsquigarrow$ Lauffähige Implementierung

- Implementierung:
  - Termination?
  - Effizienz? Geht es besser? Oder allgemeiner?
  - Gibt es hilfreiche Bibliotheksfunktionen?
  - Wie könnte man die Korrektheitheit zeigen?

  $\rightsquigarrow$ Lauffähige Implementierung
- Test:
  - Black-box: Daten aus der Spezifikation
  - White-box: Daten aus der Implementierung
  - Testfälle: hohe Abdeckung, Randfälle beachten.

Ein Beispiel: das Nim-Spiel

- Zwei Spieler nehmen abwechselnd 1–3 Hölzchen.
- Verloren hat derjenige, der das letzte Hölzchen nimmt.
- Ziel: Programm, das entscheidet, ob ein Zug gewinnt.

Ein Beispiel: das Nim-Spiel

- Zwei Spieler nehmen abwechselnd 1–3 Hölzchen.
- Verloren hat derjenige, der das letzte Hölzchen nimmt.
- Ziel: Programm, das entscheidet, ob ein Zug gewinnt.
- Eingabe: Anzahl Hölzchen gesamt, Zug
- Zug = Anzahl genommener Hölzchen
- Ausgabe: Gewonnen, ja oder nein.

```
type Move= Int
winningMove :: Int-> Move-> Bool
```

Erste Verfeinerung

- Gewonnen, wenn Zug legal & Gegner kann nicht gewinnen:

```
winningMove total move =
  legalMove total move &&
  mustLose (total-move)
```

Erste Verfeinerung

- Gewonnen, wenn Zug legal & Gegner kann nicht gewinnen:

```
winningMove total move =
  legalMove total move &&
  mustLose (total-move)
```

- Überprüfung, ob Zug legal:

```
legalMove :: Int-> Int-> Bool
legalMove total m =
   (m<= total) && (1<= m) && (m<= 3)
```

- Gegner kann nicht gewinnen, wenn
  - nur noch ein Hölzchen übrig ist, oder
  - wir bei jedem möglichen Zug von ihm gewinnen können

- Gegner kann nicht gewinnen, wenn
  - nur noch ein Hölzchen übrig ist, oder
  - wir bei jedem möglichen Zug von ihm gewinnen können

```
mustLose :: Int-> Bool
mustLose n
   | n == 1     = True
   | otherwise = canWin n 1 &&
                 canWin n 2 &&
                 canWin n 3
```

- Wir gewinnen, wenn es legalen, gewinnenden Zug gibt:

```
canWin :: Int-> Int-> Bool
canWin total move =
  winningMove (total- move) 1 ||
  winningMove (total- move) 2 ||
  winningMove (total- move) 3
```

- Analyse:
  - Effizienz: unnötige Überprüfung bei `canWin`
  - Testfälle: Gewinn, Verlust, Randfälle
- Korrektheit:
  - Vermutung: Mit $4n + 1$ Hölzchen verloren, ansonsten gewonnen.
  - Beweis durch Induktion $\rightsquigarrow$ später.

Der Basisdatentyp Bool

Wahrheitswerte: *Bool*

- Werte `True` und `False`
- Funktionen:

```
not :: Bool-> Bool              ¬
&&  :: Bool-> Bool-> Bool       ∧
||  :: Bool-> Bool-> Bool       ∨
```

Wahrheitswerte: *Bool*

- Werte `True` und `False`
- Funktionen:

```
not :: Bool-> Bool          ¬
&&  :: Bool-> Bool-> Bool   ∧
||  :: Bool-> Bool-> Bool   ∨
```

- Beispiel: exklusives Oder:

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

- Alternativ, mit bedingten Gleichungen:

```
exOr x y
  | x == True  = if y == False then True
                                else False
  | x == False = if y == True  then True
                                else False
```

- Alternativ, mit bedingten Gleichungen:

```
exOr x y
  | x == True  = if y == False then True
                                else False
  | x == False = if y == True  then True
                                else False
```

- Igitt! Besser: Definition mit pattern matching

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

Numerische Basisdatentypen

Zahlen

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

Haskell bietet die Auswahl:

Zahlen

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

Haskell bietet die Auswahl:

`Int` Zahlen (Betrag $\leq 2^{31}$)	$\longleftrightarrow$	`Integer` $\mathbb{Z}$
`Float` / `Double` Fließkommazahlen (32/64 Bit)	$\longleftrightarrow$	`Rational` $\mathbb{Q}$

Ganze Zahlen: *Int* und *Integer*

- Nützliche Funktionen (überladen, auch für `Integer`):

```
+, *, ^, -   :: Int-> Int-> Int
abs, -       :: Int-> Int
div, mod     :: Int-> Int-> Int
```

  Es gilt `(x ‘div‘ y)*y + x ‘mod‘ y == x`

  - Achtung: Unäres Minus im Zweifelsfall klammern: `abs (-34)`

- Vergleich

```
=, /=, <, >, <=, >= :: Int-> Int-> Bool
```

Fließkommazahlen: *Double*

- Doppeltgenaue Fließkommazahlen (IEEE 754 und 854)
  - Logarithmen, Wurzel, Exponentation, π und e, trigonometrische Funktionen (siehe Thompson S. 44)
- Konversion in ganze Zahlen:
  - `fromInt :: Int-> Double`
  - `fromInteger :: Integer-> Double`
  - `round, truncate :: Double-> Int` (bzw. `Integer`)
  - Überladungen mit Typannotation auflösen:

```
round (fromInt 10) :: Int
```

- Rundungsfehler!

Strukturierte Datentypen: Tupel und Listen

Tupel und Listen

- *Strukturierte Typen* : konstruieren aus bestehenden Typen neue Typen.
- *Tupeltypen* definieren kartesische Produkte:

  `(t1, t2)` = alle möglichen Kombinationen von Werten aus `t1` und `t2`.

  - Tripel und Quadrupel

- *Listen* definieren (homogene) Sequenzen:

  `[t]` = endliche Folgen von Werten aus `t`

- Beispiel: ein Einkaufskorb
  - Inhalt: Menge von Dingen mit Namen und Preis

```
type Item   = (String, Int)
type Basket = [Item]
```

- Beispiel: ein Einkaufskorb
  - Inhalt: Menge von Dingen mit Namen und Preis

```
type Item   = (String, Int)
type Basket = [Item]
```

- Beispiel: Punkte, Rechtecke, Polygone

```
type Point   = (Int, Int)
type Line    = (Point, Point)
type Polygon = [Point]
```

Funktionen über Listen und Tupeln

- Funktionsdefinition durch pattern matching:

```
add :: Point-> Point-> Point
add (a, b) (c, d) = (a+ c, b+ d)
```

- Bei Listen werden zwei Fälle unterschieden
  - Eine Liste ist entweder leer
  - oder sie besteht aus einem Kopf und einem Rest

```
sumList :: [Int]-> Int
sumList []     = 0                  -- leere Liste
sumList (x:xs) = x + sumList xs     -- nicht-leere Liste
```

  - Hier hat `x` den Typ `Int`, `xs` den Typ `[Int]`. (automatisch!)

- Gesamtpreis des Einkaufs:

```
total :: Basket -> Int
total [] = 0
total ((name, price):rest) = price + total rest
```

- Gesamtpreis des Einkaufs:

```
total :: Basket -> Int
total [] = 0
total ((name, price):rest) = price + total rest
```

- Verschiebung eines Polygons:

```
move :: Polygon-> Point-> Polygon
move [] p                       = []
move ((x, y):ps) (px, py) = (x+ px, y+ py):
                                  (move ps (px, py))
```

Alphanumerische Basisdatentypen

Einzelne Zeichen: *Char*

- Notation für einzelne Zeichen: `'a'`,. . .
  - NB. hugs verwendet 8-Bit-character, ghci Unicode.
- Nützliche Funktionen:

```
ord :: Char -> Int
chr :: Int -> Char

toLower :: Char-> Char
toUpper :: Char-> Char
isDigit :: Char-> Bool
isAlpha :: Char-> Bool
```

Zeichenketten

- *Zeichenketten* sind Sequenzen von Zeichen:

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

- Alle vordefinierten Funktionen auf Listen sind 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

- Vorwärts und rückwärts gelesen gleiche Wörter
  - Beispiele: Otto, Reliefpfeiler

Beispiel: Palindrome

- Vorwärts und rückwärts gelesen gleiche Wörter
  - Beispiele: Otto, Reliefpfeiler

- Signatur: `palindrom :: String-> Bool`

Beispiel: Palindrome

- Vorwärts und rückwärts gelesen gleiche Wörter
  - Beispiele: Otto, Reliefpfeiler

- Signatur: `palindrom :: String-> Bool`
- Entwurf:
  - Rekursive Formulierung:
    erster Buchstabe = letzer Buchstabe, und Rest auch Palindrom
    Wörter der Länge 0 und 1 sind Palindrome
  - Hilfsfunktionen (vordefiniert):
    `last: String-> Char`, `init: String-> String`

- Implementierung:

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

- 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.

Exkurs: Operatoren in Haskell

- Operatoren: Namen aus Sonderzeichen `!$%&/?+^ ...`
- Werden infix geschrieben: `x && y`
- Ansonsten normale Funktion.
- Andere Funktion infix benutzen:

  ```
  x `exOr` y
  ```

  - In Apostrophen (back quotes) einschließen.
- Operatoren in Nicht-Infixschreibweise (präfix):

  ```
  %% (&&) :: Bool-> Bool-> Bool
  (&&) True ((||) x y)
  ```

  - In Klammern einschließen.

Zusammenfassung

- Funktionsdefinitionen:
  - Abseitsregel, bedingte Definition, *pattern matching*
- Numerische Basisdatentypen:
  - `Int`, `Integer`, `Float`, `Double` und `Rational`
- Funktionaler Entwurf und Entwicklung
  - Spezifikation der Ein- und Ausgabe $\rightsquigarrow$ Signatur
  - Problem rekursiv formulieren $\rightsquigarrow$ Implementation
  - Test und Korrektheit
- Strukturierte Datentypen: Tupel und Listen
- Alphanumerische Basisdatentypen: `Char` und `String`