Vorlesung vom 01.11.2004: Funktionen und Typen

Organisatorisches

- PI1/PI2 empfohlen aber nicht Voraussetzung für PI3.
- Tutorien: Ungleichverteilung.
- Übungsblätter:
  - Lösungen in LaTeX wie gehabt,
  - Formulare auf Webseite.

Inhalt

- Wie definiere ich eine Funktion?
  - Syntaktische Feinheiten
  - Von der Spezifikation zum Programm
- Basisdatentypen:
  - Wahrheitswerte, numerische und alphanumerische Typen
- Strukturierte Datentypen:
  - Listen und Tupel

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

Die Abseitsregel

Funktionsdefinition:

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

- Gültigkeitsbereich 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.

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 = ...
```

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.

Die Alternative: Literate Programming

- Literate Haskell (`.lhs`): Quellcode besteht hauptsächlich aus Kommentar, Programmcode ausgezeichnet.
- In Haskell zwei Stile:
  - Alle Programmzeilen mit > kennzeichnen.
  - Programmzeilen in `\begin{code}` ... `\end{code}` einschließen
- Umgebung code in LaTeX definieren:

```
\def\code{\verbatim}
\def\endcode{\endverbatim}
```

- Mit LaTeX setzen, mit Haskell ausführen: (Quelle, ausführen).

```
test x = if x == 0 then 1 else 0
```

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? Mögliche Verallgemeinerungen?
  - Gibt es hilfreiche Büchereifunktionen?
  - Wie würde man die Korrektheitheit zeigen?

  $\rightsquigarrow$ Lauffähige Implementierung
- Test:
  - Black-box Test: Testdaten aus der Spezifikation
  - White-box Test: Testdaten aus der Implementierung
  - Testdaten: 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.
- 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)
```

- 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 über, oder
  - kann nur Züge machen, bei denen es Antwort gibt, wo wir gewinnen

```
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. Testen.
- Korrektheit:
  - Vermutung: Mit $4n + 1$ Hölzchen verloren, ansonsten gewonnen.
  - Beweis durch Induktion $\rightsquigarrow$ später.

Wahrheitswerte: `Bool`

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

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

- Beispiel: ausschließende Disjunktion:

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

- Alternative: explizite Fallunterscheidung

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

- Noch besser: (`y == False` = `not y`)

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

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, rem   :: Int-> Int-> Int
mod, quot  :: Int-> Int-> Int
```

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

- Vergleich durch ==, /=, <=, <, . . .
- Achtung: Unäres Minus
  - Unterschied zum Infix-Operator -
  - Im Zweifelsfall klammern: `abs (-34)`
- Fallweise Definitionen möglich

Fließkommazahlen: `Double`

- Doppeltgenaue Fließkommazahlen (IEEE 754 und 854)
  - Logarithmen, Wurzel, Exponentation, π und e, trigonometrische Funktionen
  - siehe Thompson S. 44
- 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!

Strukturierte Datentypen: Tupel und Listen

- Strukturierte Typen: konstruieren aus bestehenden Typen neue Typen.
- Tupel sind das kartesische Produkt:

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

- Listen sind Sequenzen:

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

Beispiele

- Modellierung eines Einkaufswagens
  - Inhalt: Menge von Dingen mit Namen und Preis

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

- 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)
```

- Für Listen:
  - entweder leer
  - oder bestehend aus einem Kopf und einem Rest

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

  - Hier hat `x` den Typ `Int`, `xs` den Typ `[Int]`.

- Gesamtpreis des Einkaufs:

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

- Translation eines Polygons:

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

Einzelne Zeichen: `Char`

- Notation für einzelne Zeichen: `'a'`,. . .
  - NB. Kein Unicode.
- Nützliche Funktionen:

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

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

Zeichenketten: `String`

- `String` sind Sequenzen von Zeichenketten:

```
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`

- 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 einschließen.
- Operatoren in Nicht-Infixschreibweise (präfix):

  ```
  (&&) :: Bool-> Bool-> Bool
  ```

  - In Klammern einschließen.

Zusammenfassung

- Funktionsdefinitionen:
  - Abseitsregel, bedingte Definition, pattern matching
- Numerische Basisdatentypen:
  - `Int`, `Integer`, `Rational` und `Double`
- 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`
  - `type String = [Char]`