LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Fortgeschrittene Funktionale Programmierung

Wintersemester 2021/22

Eine Einführung in wichtige Konstrukte von Haskell

Prof. Dr. David Sabel

LFE Theoretische Informatik

Letzte Änderung der Folien: 4. November 2021 ♦ Die Folien basieren zum Teil auf Material von Dr. Steffen Jost, dem an dieser Stelle für die Verwendungserlaubnis herzlich gedankt sei.

Ziele des Kapitels

- Übersicht über die wesentliche Konstrukte von Haskell
- Werkzeuge und Haskell, Typen, Zahlen und einfache Typen, Datentypen: Aufzählungstypen, Produkttypen, Summentypen, Record-Syntax, rekursive Datentypen: Listen und Bäume, Funktionen, Module
- Zum Teil ist dies eine Wiederholung des Stoffs aus ProMo

Haskell und Werkzeuge

Haskell und Werkzeuge

http://www.haskell.org

- ist eine rein funktionale Sprache mit verzögerter Auswertung.
- benannt nach dem US-amerikanischen Logiker Haskell B. Curry (1900-1982).
- Standards: Haskell 98 und Haskell 2010
- Interpreter / Compiler: GHCi bzw. GHC (Glasgow / Glorious Haskell Compiler)
- Dokumentation des GHC:
  `http://downloads.haskell.org/ghc/latest/docs/html/users_guide/`
- Dokumentation der Standardbibliotheken:
  `http://www.haskell.org/ghc/docs/latest/html/libraries/index.html`

GHC / GHCi installieren

Verschiedene Varianten zur Installation:

- Minimal = GHC und Build Tools
- Stack: Projektbasiertes Build-Tool, automatisches Management von Abhängigkeiten
- Haskell-Platform: GHC,Build Tools und Libraries

Wir empfehlen Stack:

`http://haskellstack.org`

GHCi verwenden

Interpreter starten und Ausdrücke eingeben und auswerten lassen:

```
> stack exec -- ghci
GHCi, version 8.6.5: http://www.haskell.org/ghc/ :? for help
Prelude> 1+2
3
Prelude> 10 * (2 + 4)
60
Prelude> 14 `mod` 6
2
```

Mit den Pfeiltasten  ,  kann man die vorherigen Eingaben durchblättern

GHCi verwenden

Steuerung des Interpreters:

- Befehle, die mit `:` beginnen, z.B. `:?` für die Hilfe.
- Befehle abkürzen, z.B. statt `:quit` auch `:q`, um den Interpreter zu verlassen

Quelltexte laden:

```
:l datei.hs -- lade Definition aus Datei datei.hs
:r          -- erneut alle offenen Dateien einlesen
```

Quellcode-Dateien

Datei-Endung:

- Normale Haskell-Quellcodedateien enden mit `.hs`
- Literate-Haskell-Dateien mit `.lhs`

Quellcode-Dateien

Datei-Endung:

- Normale Haskell-Quellcodedateien enden mit `.hs`
- Literate-Haskell-Dateien mit `.lhs`

Normale Haskell-Quellcodedateien:

- Alles ist Quellcode, außer Kommentare
- `--␣` leitet Kommentar bis zum Zeilenende ein
- Mehrzeiliger (geklammerter) Kommentar mit `{-` und `-}`

Quellcode-Dateien

Datei-Endung:

- Normale Haskell-Quellcodedateien enden mit `.hs`
- Literate-Haskell-Dateien mit `.lhs`

Normale Haskell-Quellcodedateien:

- Alles ist Quellcode, außer Kommentare
- `--␣` leitet Kommentar bis zum Zeilenende ein
- Mehrzeiliger (geklammerter) Kommentar mit `{-` und `-}`

Literate-Haskell-Programm

- Alles ist Kommentar, außer es ist als Quellcode gekennzeichnet
- Bird-Stil (nach Richard Bird): Code-Zeilen beginnen mit `>␣` und Code- und Kommentarzeilen sind durch Leerzeile getrennt.
- LaTeX-Stil: Code steht in `\begin{code} ... \end{code}`-Umgebung.

Beispiel: Normales Haskell

```
{- Hier steht beliebiger Kommentar, der
   sich auch über mehrere Zeile erstrecken
   kann.
-}
meineFunktion [] = putStrLn "Hallo Welt"
meineFunktion (_:rs) = do
                       putStrLn "Hallo Welt"
                       meineFunktion rs

-- Hier steht erneut ein Kommentar.
```

Beispiel: Literate-Haskell im Bird-Stil

```
Hier steht beliebiger Kommentar, der
sich auch über mehrere Zeile erstrecken
kann.

> meineFunktion [] = putStrLn "Hallo Welt"
> meineFunktion (_:rs) = do
>                        putStrLn "Hallo Welt"
>                        meineFunktion rs

Hier steht erneut ein Kommentar.
```

Beispiel: Literate-Haskell im LaTeX-Stil

```
Hier steht beliebiger Kommentar, der
sich auch über mehrere Zeile erstrecken
kann.
\begin{code}
meineFunktion [] = putStrLn "Hallo Welt"
meineFunktion (_:rs) = do
                       putStrLn "Hallo Welt"
                       meineFunktion rs
\end{code}
Hier steht erneut ein Kommentar.
```

Allgemeine Syntaxregeln

- Haskell beachtet die Groß-/Kleinschreibung
- Haskell ist „whitespace“-sensitiv: Einrückungen werden beachtet
- Veränderungen an Leerzeichen, Tabulatoren und Zeilenumbrüchen können Fehler verursachen.
- Statt Einrückung kann man auch geschweifte Klammern `{ }` und Semikolons `;` verwenden.

Dringende Empfehlung:

Keine Tabulatoren verwenden, sondern (am besten durch den Text-Editor) durch Leerzeichen ersetzen.

Editoren und IDEs: `https://wiki.haskell.org/IDEs`.

Allgemeine Syntaxregeln (2)

Daumenregel für Einrückungen:

Beginnt die nächste Zeile in einer Spalte

- weiter rechts als die vorherige, dann geht die vorherige Zeile weiter
- in der gleichen Spalte wie die vorherige, dann beginnt das nächste Element eines Blocks
- weiter links als die vorherige, dann ist der Block beendet.

Allgemeine Syntaxregeln (2)

Daumenregel für Einrückungen:

Beginnt die nächste Zeile in einer Spalte

- weiter rechts als die vorherige, dann geht die vorherige Zeile weiter
- in der gleichen Spalte wie die vorherige, dann beginnt das nächste Element eines Blocks
- weiter links als die vorherige, dann ist der Block beendet.

Falsch:

```
fun x = let x = 2
          y = 3
        in x+y
<interactive>: parse error on input ‘y’
```

Falsch:

```
fun x = let x = 2
              y = 3
         in x+y
<interactive>: parse error on input ‘=’
```

Richtig:

```
fun x = let x = 2
            y = 3
        in x+y
```

Typen in Haskell

Typen

- **Typ** = Menge von Werten
- Z.B. `Int` = Menge der ganzen Zahlen von -2^{63} bis $2^{63} - 1$.
- In Haskell:
  - Typnamen beginnen mit einem Großbuchstaben
  - Typvariablen beginnen mit einem Kleinbuchstaben.
- Im GHCi: `:type` *Ausdruck* zeigt Typ an

```
Prelude> :type 'a'
'a' :: Char
```

Typen: Sprechweisen

Gilt `e :: T`, so sagt man

- „Ausdruck `e` hat Typ `T`“

  oder alternativ

- „Ausdruck `e` ist vom Typ `T`“

Syntax von Typen

Die Syntax von Typen (ohne Typklassenbeschränkungen) in Haskell ist

$$\mathbf{T} ::= TV \mid TC\ \mathbf{T}_1\ \ldots\ \mathbf{T}_n \mid \mathbf{T}_1\ \texttt{->}\ \mathbf{T}_2$$

wobei TV für eine Typvariable steht und TC ein Typkonstruktor mit Stelligkeit n ist

- Typkonstruktoren beginnen mit einem Großbuchstaben
  (wenn sie infix verwendet werden, mit einem Doppelpunkt)
- $\mathbf{T}_1$ -> $\mathbf{T}_2$ ist ein **Funktionstyp**, d.h. Typ jener Funktionen,
  - deren Eingabe Werte des Typs $\mathbf{T}_1$ sind, und
  - die als Ausgabe Werte des Typs $\mathbf{T}_2$ liefern.

Sprechweisen für verschiedene Typen

Basistypen: nullstellige Typkonstruktoren
(Basistypen haben keine Typvariablen und sind nicht zusammengesetzt aus anderen Typen)

Z.B. `Int`, `Bool`

Sprechweisen für verschiedene Typen

Basistypen: nullstellige Typkonstruktoren
(Basistypen haben keine Typvariablen und sind nicht zusammengesetzt aus anderen Typen)

Z.B. `Int`, `Bool`

Grundtypen / **monomorphe Typen**: Typen, die keine Typvariablen enthalten.
zusammengesetzte Typen sind erlaubt

Z.B. `Baum Int`, `[(Int,Bool)]`, `Int -> Bool`

Sprechweisen für verschiedene Typen

Basistypen: nullstellige Typkonstruktoren
(Basistypen haben keine Typvariablen und sind nicht zusammengesetzt aus anderen Typen)

Z.B. `Int`, `Bool`

Grundtypen / **monomorphe Typen**: Typen, die keine Typvariablen enthalten.
zusammengesetzte Typen sind erlaubt

Z.B. `Baum Int`, `[(Int,Bool)]`, `Int -> Bool`

Polymorphe Typen: Typen, die auch Typvariablen enthalten dürfen

Z.B. `[a]`, `a -> b -> b`

Sprechweisen für verschiedene Typen

Basistypen: nullstellige Typkonstruktoren
(Basistypen haben keine Typvariablen und sind nicht zusammengesetzt aus anderen Typen)

Z.B. `Int`, `Bool`

Grundtypen / **monomorphe Typen**: Typen, die keine Typvariablen enthalten.
zusammengesetzte Typen sind erlaubt

Z.B. `Baum Int`, `[(Int,Bool)]`, `Int -> Bool`

Polymorphe Typen: Typen, die auch Typvariablen enthalten dürfen

Z.B. `[a]`, `a -> b -> b`

Es gilt: Basistypen $\subseteq$ Grundtypen $\subseteq$ polymorphe Typen

Zahlen und einfache Typen in Haskell

Haskell: Zahlen

Eingebaut sind:

- Ganze Zahlen beschränkter Größe: `Int` (mindestens von -2^{29} bis $2^{29}-1$)
- Ganze Zahlen beliebiger Größe: `Integer`
- Gleitkommazahlen: `Float`
- Gleitkommazahlen mit doppelter Genauigkeit: `Double`
- Rationale Zahlen: `Rational`

  (verallgemeinert `Ratio` α, wobei `Rational = Ratio Integer`)

  Darstellung x `%` y (Modul `Data.Ratio` importieren)

Arithmetische Operationen

Rechenoperationen:

- `+` für die Addition
- `-` für die Subtraktion
- `*` für die Multiplikation
- `/` für die Division
- `mod` , `div`

Die Operatoren sind überladen. Dafür gibt es Typklassen.

Typ von `(+) :: Num a => a -> a -> a`

Genaue Behandlung von Typklassen: später

Funktionsnamen / Operatoren

- Namen von **Funktionen** müssen in Haskell mit einem Kleinbuchstaben oder _ beginnen. (z.B. `mod`, `div`)
- **Operatoren** bestehen nur aus Symbolen, die keine alphanumerischen Zeichen sind. (z.B. +, $, <*>)
- Operatoren werden standardmäßig infix verwendet,
  während Funktionen standardmäßig präfix verwendet werden.
- Operatoren, die mit : beginnen, spielen eine Sonderrolle, da sie Datenkonstruktoren sind

Präfix / Infix

Anmerkung zum Minuszeichen:

- Mehr Klammern als man denkt: `5 + -6` geht nicht,
  richtig: `5 + (-6)`
- In Haskell können Funktionen durch Einfassen in Backquotes ` auch infix benutzt werden:
  `mod 5 6` ; infix durch Backquotes: `5 `mod` 6`
- Umgekehrt können infix-Operatoren durch Einklammern in () auch präfix benutzt werden:
  `5 + 6` ; präfix durch Klammern: `(+) 5 6`

Weitere Typen

`Bool` Boolesche (logische) Wahrheitswerte: `True` und `False`

`Char` Unicode Zeichen, z.B. `'q'`. Diese werden immer in Apostrophen eingeschlossen.

`String` Zeichenketten, z.B. `"Hallo!"`. Diese werden immmer in Anführungszeichen eingeschlossen.

Davon ist lediglich `Char` eingebaut, denn:

```
type String = [Char]      -- Typsynonym
data Bool = True | False
```

Einige vordefinierte Funktionen

Die Standardbibliothek (die sogenannte Prelude) definiert u.a. folgende Funktionen

- `&&` Konjunktion, logisches Und
- `||` Disjunktion, logisches Oder
- `not` Negation, Verneinung
- `==` Test auf Gleichheit
- `/=` Test auf Ungleichheit

```
Prelude> (True || False) && (not True)
False
Prelude> True == False
False
Prelude> False /= True
True
```

Vergleichsoperatoren

- `==` für den Gleichheitstest
  `(==) :: (Eq a) => a -> a -> Bool`
- `/=` für den Ungleichheitstest
- `<`, `<=`, `>`, `>=`, für kleiner, kleiner gleich, größer und größer gleich
  (der Typ ist `(Ord a) => a -> a -> Bool`.

Assoziativitäten und Prioritäten

```
infixr 9  .
infixr 8  ^, ^^, **
infixl 7  *, /, `quot`, `rem`, `div`, `mod`
infixl 6  +, -

-- The (:) operator is built-in syntax, and cannot
-- legally be given a fixity declaration; but its
-- fixity is given by:
--   infixr 5  :

infix  4  ==, /=, <, <=, >=, >
infixr 3  &&
infixr 2  ||
infixl 1  >>, >>=
infixr 1  =<<
infixr 0  $, $!, `seq`
```

Algebraische Datentypen in Haskell

Aufzählungstypen

Aufzählungstyp = Aufzählung verschiedener Werte

`data Typname = Konstante`$_1$ `|` `Konstante`$_2$ `|` $\ldots$ `|` `Konstante`$_N$

Beispiele:

```
data Bool      = True | False

data Wochentag = Montag | Dienstag | Mittwoch | Donnerstag
               | Freitag | Samstag | Sonntag
  deriving(Show)
```

`deriving(Show)` erzeugt Instanz der Typklasse `Show`,
damit der Datentyp angezeigt werden kann.

Aufzählungstypen (2)

```
istMontag :: Wochentag -> Bool
istMontag x = case x of
                Montag -> True
                Dienstag -> False
                Mittwoch -> False
                Donnerstag -> False
                Freitag -> False
                Samstag -> False
                Sonntag -> False
```

Mit Default-Alternative:

```
istMontag' :: Wochentag -> Bool
istMontag' x = case x of
                Montag -> True
                y      -> False
```

Aufzählungstypen (3)

In Haskell:

Pattern-matching in den linken Seiten der Funktionsdefinition:

```
istMontag'' :: Wochentag -> Bool
istMontag'' Montag = True
istMontag'' _      = False
```

Produkttypen

Produkttyp = Zusammenfassung verschiedener Werte

Bekanntes Beispiel: Tupel

Definition (Kartesisches Produkt)

Sind $A_1, \ldots A_n$ Mengen, so ist das kartesische Produkt definiert als
$A_1 \times \cdots \times A_n = \{(a_1, \ldots, a_n) \mid a_i \in A_i \text{ für } i = 1 \ldots n\}$

Die Elemente von $A_1 \times \cdots \times A_n$ heißen allgemein n-*Tupel*, spezieller auch Paare, Tripel, Quadrupel, ...

Produkttypen (2)

In Haskell: Tupelausdrücke und -typen mit runden Klammern und Kommas,
z.B. $\mathbb{Z} \times \mathbb{Z}$ wird zu `(Int,Int)`.

```
Prelude> :t (True, 'a', 7)
(True, 'a', 7) :: (Bool, Char, Int)
Prelude> :t (4.5,"Hi!")
(4.5,"Hi!") :: (Double, String)
```

Das 0-Tupel ist ebenfalls in Haskell erlaubt: `() :: ()`.
Der Typ `()` wird als Unit-Typ bezeichnet und hat nur den einzigen Wert `()`.

Produkttypen (3)

Mit `data` können auch Produkttypen definiert werden (die keine Tupelsyntax verwenden).

Die Syntax hierfür ist

`data` Typname = KonstruktorName Typ_1 Typ_2 ... Typ_N

Beispiel:

```
data Student = Student
                 String  -- Name
                 String  -- Vorname
                 Int     -- Matrikelnummer
```

Beachte: `Student` taucht hier auf als Typ und als Datenkonstruktor

Produkttypen (3)

```
setzeName :: Student -> String -> Student
setzeName x name' =
 case x of
  (Student name vorname mnr)
            -> Student name' vorname mnr
```

Alternativ mit Pattern auf der linken Seite der Funktionsdefinition:

```
setzeName :: Student -> String -> Student
setzeName (Student name vorname mnr) name' =
  Student name' vorname mnr
```

Produkttypen und Aufzählungstypen

Man kann beides mischen, Z.B.

```
data DreiDObjekt =
       Wuerfel Int
     | Quader Int Int Int
     | Kugel Int
```

Früchte:

```
data Frucht = Apfel (Int,Double)
            | Birne Int Double
            | Banane Int Int Double
```

Allgemein nennt man dies auch **Summentyp**, allgemein

`data Typ =` $\texttt{Typ}_1$ `|` ... `|` $\texttt{Typ}_n$

wobei $\texttt{Typ}_1,\ldots,\texttt{Typ}_n$ komplexe Typen sind

Record-Syntax: Einführung

```
data Student = Student
                 String  -- Vorname
                 String  -- Name
                 Int     -- Matrikelnummer
```

Nachteil: Nur die Kommentare verraten, was die Komponenten darstellen.

Außerdem mühsam: Zugriffsfunktionen erstellen:

```
vorname :: Student -> String
vorname (Student vorname name mnr) = vorname
```

Record-Syntax: Einführung (2)

Änderung am Datentyp:

```
data Student = Student
                 String  -- Vorname
                 String  -- Name
                 Int     -- Matrikelnummer
                 Int     -- Hochschulsemester
```

muss für Zugriffsfunktionen nachgezogen werden

```
vorname :: Student -> String
vorname (Student vorname name mnr hsem) = vorname
```

Abhilfe verschafft die Record-Syntax

Record-Syntax in Haskell

Student mit Record-Syntax:

```
data Student =  Student {
                          vorname        :: String,
                          name           :: String,
                          matrikelnummer :: Int
                        }
```

Zur Erinnerung: Ohne Record-Syntax:

```
data Student = Student String String Int
```

$\Rightarrow$ Die Komponenten werden mit Namen markiert

Beispiel

Beispiel: `Student "Hans" "Mueller" 1234567`

kann man schreiben als

```
Student{vorname="Hans",
        name="Mueller",
        matrikelnummer=1234567}
```

Reihenfolge der Komponenten egal:

```
Student{matrikelnummer=1234567,
        vorname="Hans",
        name="Mueller"}
```

Record-Syntax

Zugriffsfunktionen sind automatisch verfügbar, z.B.

```
*> matrikelnummer x
1234567
```

- Record-Syntax ist in den Pattern erlaubt
- Nicht alle Felder müssen abgedeckt werden
- bei Erweiterung der Datenstrukturen, daher kein Problem

```
nameMitA Student{name = 'A':xs} = True
nameMitA _ = False
```

Record-Syntax: Update

```
setzeName :: Student -> String -> Student
setzeName student neuername =
        student {name = neuername}
```

ist äquivalent zu

```
setzeName :: Student -> String -> Student
setzeName student neuername =
  Student {vorname        = vorname student,
           name           = neuername,
           matrikelnummer = matrikelnummer student}
```

Parametrisierte Datentypen

Datentypen in Haskell dürfen polymorph parametrisiert sein:

`data` *Typname* $par_1 \ldots par_m$ = *Konstruktor*$_1$ $arg_{1,1} \ldots arg_{1,i_1}$ | ... | *Konstruktor*$_n$ $arg_{n,1} \ldots arg_{n,i_n}$
`deriving` (*class*$_1$, ..., *class*$_l$)

- Typname muss mit einem Großbuchstaben beginnen
- Typparameter sind optional, ebenso Alternativen
- Konstruktoren müssen mit Großbuchstaben beginnen
- Konstruktoren erwarten als Argumente eine beliebige Anzahl von Argumenten: Argument = bekannter Typ oder Typparameter
- Optionale `deriving`-Klausel mit einer Liste von Typklassen

Der Datentyp Maybe

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

- ist polymorph über dem Parameter a.
- Konkrete Instanz ist z.B. `Maybe Int`.
- Verwendung: Undefinierte Fälle partieller Funktionen mit `Nothing` abbilden, dadurch totale Funktionen Erstellen

Beispiel:

```
getKruemmung :: Frucht -> Maybe Double
getKruemmung (Banane _ _ k)  = Just k
getKruemmung _               = Nothing
```

Einige vordefinierte Funktionen für Maybe

```
isJust         :: Maybe a -> Bool
isJust Nothing = False
isJust _       = True

fromMaybe :: a -> Maybe a -> a
fromMaybe standardWert Nothing  = standardWert
fromMaybe      _       (Just x) = x

catMaybes :: [Maybe a] -> [a]
catMaybes ls = [x | Just x <- ls]
```

Der Datentyp Either

Der Datentyp `Either` hat mehrere Typparameter:

```
data Either a b  =  Left a | Right b
```

Z.B.: Verwende `Either a String` anstelle von `Maybe a`
für die Rückgabe von Fehlermeldungen.

```
getKruemmung' :: Frucht -> Either String Double
getKruemmung' (Banane _ _ k)  = Right k
getKruemmung' _               = Left "Eingabe ist keine Banane."

myDiv :: Double -> Double -> Either String Double
myDiv x 0 = Left "Error: Division by 0"
myDiv x y = Right $ x / y
```

Der Datentyp Either (2)

`Either` ist der Typ für disjunkte Vereinigungen:

$$A_1 \dot\cup A_2 = \big\{(i, a) \;\big|\; a \in A_i\big\}$$

Zum Beispiel:

$$\{\diamondsuit, \heartsuit\} \dot\cup \{\heartsuit, \clubsuit, \spadesuit\} = \{(1, \diamondsuit), (1, \heartsuit), (2, \heartsuit), (2, \clubsuit), (2, \spadesuit)\}$$

Hilfreiche Funktion für Either

```
isRight :: Either a b -> Bool
isRight (Left  _) = False
isRight (Right _) = True

lefts   :: [Either a b] -> [a]
lefts x = [a | Left a <- x]

partitionEithers :: [Either a b] -> ([a],[b])
partitionEithers []      = ([],[])
partitionEithers (h : t)
   | Left  l <- h = (l:ls,   rs)
   | Right r <- h = (  ls, r:rs)
  where (ls,rs) = partitionEithers t
```

Rekursive Datentypen

Rekursiver Datentyp: Der definierte Typ kommt rechts vom = wieder vor

```
data Typ = ... Konstruktor Typ ...
```

Listen könnte man definieren als:

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

In Haskell, eher Spezialsyntax:

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

In Haskell: $[a_1, \ldots, a_n]$ als Abkürzung für $a_1 : (a_2 : (\ldots : []) \ldots)$

Beispiele:

```
[1,2,3] :: [Int]
[1,2,2,3,3,3] :: [Int]
["Hello","World","!"] :: [[Char]]
```

Listen vs. Tupel

	Listen	Tupel
Anzahl an Elementen	variabel	fest
Typ der Elemente	identisch	auch verschieden erlaubt

Man kann Listen und Tupel man beliebig ineinander verschachteln:

```
[(1,'a'),(2,'z'),(-4,'w')] :: [(Integer, Char)]

[[1,2,3],[],[4]] :: [[Integer]]

(4.5,[(True,'a',[5,7],())]) :: (Double,[(Bool,Char,[Integer],())])
```

Beachte: `[]` `[[]]` `[[],[]]` `[[[]]]` `[[],[[]]]`

sind alles **verschiedene** Werte

Haskell: Geschachtelte Pattern

Haskell erlaubt geschachtelte Pattern, damit kann man u.a. Listen zerlegen:

```
viertesElement (x1:(x2:(x3:(x4:xs)))) = Just x4
viertesElement _                      = Nothing
```

Rekursive Datenstrukturen: Listen

Vorbemerkung

- In aktuellen Haskell-Bibliotheken sind viele Listenfunktionen **verallgemeinert** für Instanzen von `Foldable`
- Listen sind Instanzen von `Foldable`
- Zur Verständlichkeit geben wir in diesem Kapitel die auf Listen spezialisierten Implementierungen und Typen an

Beispiel:

```
length :: [a] -> Int
length [] = 0
length (_:xs) = 1+(length xs)
```

Aber:

```
> stack exec -- ghci
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Prelude> :t length
length :: Foldable t => t a -> Int
```

Listen von Zahlen: Syntaktischer Zucker in Haskell

- [*start*..*end*]
  erzeugt die Liste der Zahlen von *start* bis *end*
  z.B. ergibt `[10..15]` die Liste `[10,11,12,13,14,15]`.
- [*start*..]
  erzeugt die unendliche Liste ab dem Wert *start*,
  z.B. erzeugt `[1..]` die Liste aller natürlichen Zahlen.
- [*start*,*next*..*end*]
  erzeugt die Liste $[start, start + \delta, start + 2 * \delta, \ldots, start + m \cdot \delta]$,
  wobei $\delta = next - start$ und solange Vielfache von δ dazuaddiert werden,
  bis der Endwert erreicht ist (überschritten, falls $\delta > 0$, unterschritten falls $\delta < 0$).
  Z.B. `[1,3..8]` erzeugt `[1,3,5,7]` und `[7,5..2]` erzeugt `[7,5,3]`.
- [*start*,*next*..]
  erzeugt unendlich lange Liste mit der Schrittweite $next - start$.

Implementierung für Integer

```
-- [start..]
from :: Integer -> [Integer]
from start = fromThenDelta start 1

-- [start,next..]
fromThen :: Integer -> Integer -> [Integer]
fromThen start next = fromThenDelta start (next-start)

fromThenDelta :: Integer -> Integer  -> [Integer]
fromThenDelta start delta = start:(fromThenDelta (start+delta) delta)
```

Implementierung für Integer (2)

```
-- [start..end]
fromTo :: Integer -> Integer -> [Integer]
fromTo start end = fromThenDeltaTo start 1 end

-- [start,next..end]
fromThenTo :: Integer -> Integer -> Integer  -> [Integer]
fromThenTo start next end  = fromThenDeltaTo start (next-start) end

fromThenDeltaTo :: Integer -> Integer -> Integer  -> [Integer]
fromThenDeltaTo start delta end = go start
 where go val
        | test val     = []
        | otherwise    = val:(go (val+delta))
       test val
        | delta >= 0 = val > end
        | otherwise  = val < end
```

Guards

Verwendungen von Guards (Wächter):

```
f pat_1 ... pat_n
   | guard_1 = e_1
   | ...
   | guard_n = e_n
```

- $\texttt{guard}_1$ bis $\texttt{guard}_n$ sind Boolesche Ausdrücke, die die Variablen der Pattern $\texttt{pat}_1,\ldots,\texttt{pat}_n$ benutzen dürfen.
- Guards werden von oben nach unten ausgewertet.
- Wird ein Guard zu `True`, so wird die entsprechende rechte Seite $\texttt{e}_i$ als Resultat übernommen.
- `otherwise` vordefinert als `otherwise = True`

where-Klauseln

- Nachgestellte Definitionen mit `where`
- Analog zu `let` (vorgestellte Definitionen)
- `where`-Klauseln formen keinen Ausdruck, `let`-Ausdrücke schon.

Obige Definition mit `let`:

```
fromThenDeltaTo' :: Integer -> Integer -> Integer  -> [Integer]
fromThenDeltaTo' start delta end =
   let
      go val
       | test val     = []
       | otherwise    = val:(go (val+delta))
      test val
       | delta >= 0 = val > end
       | otherwise  = val < end
   in go start
```

`let` und `where` werden später nochmal behandelt!

Zeichen und Zeichenketten

- Eingebauter Typ `Char` für Zeichen
- Darstellung: Einfaches Anführungszeichen, z.B. `'A'`
- Steuersymbole beginnen mit \, z.B. `\n`, `\t`
- Spezialsymbole `\\` und `\"`

Zeichen und Zeichenketten

- Eingebauter Typ `Char` für Zeichen
- Darstellung: Einfaches Anführungszeichen, z.B. `'A'`
- Steuersymbole beginnen mit \, z.B. `\n`, `\t`
- Spezialsymbole `\\` und `\"`

Strings

- Vom Typ `String` = `[Char]` (Listen von Zeichen)
- Spezialsyntax `"Hallo"` ist gleich zu

  `['H','a','l','l','o']` bzw. `'H':('a':('l':('l':('o':[]))))`.

Zeichen und Zeichenketten (2)

Nützliche Funktionen für `Char`: In der Bibliothek `Data.Char`
Z.B.:

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

isLower :: Char -> Bool
isUpper :: Char -> Bool
isAlpha :: Char -> Bool

toUpper :: Char -> Char
toLower :: Char -> Char
```

Beachte: Strings als Listen von Char sind ineffizient.

Für RealWorld-Anwendungen: Verwende `ByteStrings` (siehe `Data.ByteString`).

Standard-Listenfunktionen

Einige vordefinierte Listenfunktionen, fast alle in `Data.List`

Standard-Listenfunktionen (1)

Append: ++, Listen zusammenhängen

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

Beispiele:

```
*> [1..10] ++ [100..109]
[1,2,3,4,5,6,7,8,9,10,100,101,102,103,104,105,106,107,108,109]
*> [[1,2],[2,3]] ++ [[3,4,5]]
[[1,2],[2,3],[3,4,5]]
*> "Infor" ++ "matik"
"Informatik"
```

Laufzeitverhalten: linear in der Länge der ersten Liste

Standard-Listenfunktionen (2)

Zugriff auf Listenelement per Index: `!!`

```
(!!) :: [a] -> Int -> a
[]     !! _ = error "Index too large"
(x:xs) !! 0 = x
(x:xs) !! i = xs !! (i-1)
```

Beispiele:

```
*> [1,2,3,4,5]!!3
4
*> [0,1,2,3,4,5]!!3
3
*> [0,1,2,3,4,5]!!5
5
*> [1,2,3,4,5]!!5
*** Exception: Prelude.!!: index too large
```

Standard-Listenfunktionen (3)

Index eines Elements berechnen: `elemIndex`

```
elemIndex :: (Eq a) => a -> [a] -> Maybe Int
elemIndex a xs = findInd 0 a xs
  where findInd i a [] = Nothing
        findInd i a (x:xs)
          | a == x    = Just i
          | otherwise = findInd (i+1) a xs
```

Beispiele:

```
*> elemIndex 1 [1,2,3]
Just 0
*> elemIndex 1 [0,1,2,3]
Just 1
*> elemIndex 1 [5,4,3,2]
Nothing
*> elemIndex 1 [1,4,1,2]
Just 0
```

Standard-Listenfunktionen (4)

Map: Funktion auf Listenelemente anwenden

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = (f x):(map f xs)
```

Beispiele:

```
*> map (*3) [1..20]
[3,6,9,12,15,18,21,24,27,30,33,36,39,42,45,48,51,54,57,60]
*> map not [True,False,False,True]
[False,True,True,False]
*> map (^2) [1..10]
[1,4,9,16,25,36,49,64,81,100]
*> map toUpper "Informatik"
"INFORMATIK"
```

Standard-Listenfunktionen (5)

Filter: Elemente heraus filtern

```
filter :: (a -> Bool) -> [a] -> [a]
filter f [] = []
filter f (x:xs)
 | f x       = x:(filter f xs)
 | otherwise = filter f xs
```

Beispiele:

```
*> filter (> 15) [10..20]
[16,17,18,19,20]
*> filter isAlpha "FFP 2021"
"FFP"
*> filter (\x -> x > 5) [1..10]
[6,7,8,9,10]
```

Standard-Listenfunktionen (6)

Analog zu filter: **remove**: Listenelemente entfernen

```
remove p xs = filter (not . p) xs
```

Der Kompositionsoperator `(.)` ist definiert als:

```
(f . g) x = f (g x)
```

Beispiele:

```
*> remove p xs = filter (not . p) xs
*> remove (> 5) [1..10]
[1,2,3,4,5]
*> remove isAlpha "FFP 2021"
" 2021"
```

Standard-Listenfunktionen (7)

Length: Länge einer Liste

```
length :: [a] -> Int
length [] = 0
length (_:xs) = 1+(length xs)
```

Beispiele:

```
*> length "Informatik"
10
*> length [2..20002]
20001
*> length [1..]
^CInterrupted.
```

Standard-Listenfunktionen (8)

Length: Bessere Variante (konstanter Platz)

```
length :: [a] -> Int
length xs = length_it xs 0

length_it []     acc = acc
length_it (_:xs) acc = let acc' = 1+acc
                       in seq acc' (length_it xs acc')
```

Standard-Listenfunktionen (9)

Reverse: Umdrehen einer Liste

Schlechte Variante: Laufzeit quadratisch!

```
reverse1 :: [a] -> [a]
reverse1 [] = []
reverse1 (x:xs) = (reverse1 xs) ++ [x]
```

Standard-Listenfunktionen (9)

Reverse: Umdrehen einer Liste

Schlechte Variante: Laufzeit quadratisch!

```
reverse1 :: [a] -> [a]
reverse1 [] = []
reverse1 (x:xs) = (reverse1 xs) ++ [x]
```

Besser mit Stack: Laufzeit linear

```
reverse :: [a] -> [a]
reverse xs = rev xs []
 where rev []     acc = acc
       rev (x:xs) acc = rev xs (x:acc)
```

Standard-Listenfunktionen (9)

Reverse: Umdrehen einer Liste

Schlechte Variante: Laufzeit quadratisch!

```
reverse1 :: [a] -> [a]
reverse1 [] = []
reverse1 (x:xs) = (reverse1 xs) ++ [x]
```

Beispiele:

```
*> reverse [1..10]
[10,9,8,7,6,5,4,3,2,1]
*> reverse [1..]
^CInterrupted.
*> reverse "neffen neppen neffen"
"neffen neppen neffen"
```

Besser mit Stack: Laufzeit linear

```
reverse :: [a] -> [a]
reverse xs = rev xs []
 where rev []     acc = acc
       rev (x:xs) acc = rev xs (x:acc)
```

Standard-Listenfunktionen (10)

Repeat und Replicate

```
repeat :: a -> [a]
repeat x = x:(repeat x)

replicate :: Int -> a -> [a]
replicate 0 x = []
replicate i x = x:(replicate (i-1) x)
```

Beispiele

```
*> replicate 10 [1,2]
[[1,2],[1,2],[1,2],[1,2],[1,2],[1,2],[1,2],[1,2],[1,2],[1,2]]
*> replicate 20 'A'
"AAAAAAAAAAAAAAAAAAAA"
```

Standard-Listenfunktionen (11)

Take und Drop: n Elemente nehmen / verwerfen

```
take :: Int -> [a] -> [a]
take i [] = []
take 0 xs = []
take i (x:xs) = x:(take (i-1) xs)
```

```
drop :: Int -> [a] -> [a]
drop i []     = []
drop 0 xs     = xs
drop i (x:xs) = drop (i-1) xs
```

Beispiele:

```
*> take 10 [1..]
[1,2,3,4,5,6,7,8,9,10]
*> drop 5 "Informatik"
"matik"
*> take 5 (drop 4 [1..])
[5,6,7,8,9]
```

Standard-Listenfunktionen (12)

TakeWhile und DropWhile

```
takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile p [] = []
takeWhile p (x:xs)
 | p x       = x:(takeWhile p xs)
 | otherwise = []
```

```
dropWhile :: (a -> Bool) -> [a] -> [a]
dropWhile p [] = []
dropWhile p  (x:xs)
 | p x        = dropWhile p xs
 | otherwise  = x:xs
```

Beispiele:

```
*> takeWhile (> 5) [5,6,7,3,6,7,8]
[]
*> takeWhile (> 5) [7,6,7,3,6,7,8]
[7,6,7]
*> dropWhile (< 10) [1..20]
[10,11,12,13,14,15,16,17,18,19,20]
```

Standard-Listenfunktionen (13)

Zip und Unzip

```
zip :: [a] -> [b] -> [(a,b)]
zip [] ys = []
zip xs [] = []
zip (x:xs) (y:ys) = (x,y):(zip xs ys)
```

```
unzip :: [(a, b)] -> ([a], [b])
unzip [] = ([],[])
unzip ((x,y):xs) =
      let (xs',ys') = unzip xs
      in (x:xs',y:ys')
```

Beispiele:

```
*> zip [1..10] "Informatik"
[(1,'I'),(2,'n'),(3,'f'),(4,'o'),(5,'r'),(6,'m'),(7,'a'),(8,'t'),(9,'i'),(10,'k')]
*> unzip [(1,'I'),(2,'n'),(3,'f'),(4,'o'),(5,'r'),
          (6,'m'),(7,'a'),(8,'t'),(9,'i'),(10,'k')]
([1,2,3,4,5,6,7,8,9,10],"Informatik")|
```

Standard-Listenfunktionen (14)

Bemerkung zu `zip`:

- Man kann zwar `zip3`, `zip4` etc. definieren um 3, 4, ..., Listen in 3-Tupel, 4-Tupel, etc. einzupacken, aber:

  Man kann keine Funktion `zipN` für n Listen definieren

- Grund: diese Funktion wäre nicht getypt.
- Abhilfe liefert: Template Haskell (siehe späteres Kapitel)

Standard-Listenfunktionen (15)

Kombination von `zip` und `map`:

```
zipWith :: (a -> b -> c) -> [a]-> [b] -> [c]
zipWith f (x:xs) (y:ys) = (f x y) : (zipWith f xs ys)
zipWith _ _      _      = []
```

Damit kann man `zip` definieren:

```
zip = zipWith (\x y -> (x,y))
```

Anderes Beispiel:

```
vectorAdd :: (Num a) => [a] -> [a] -> [a]
vectorAdd = zipWith (+)
```

Standard-Listenfunktionen (16)

Die Fold-Funktionen:

- `foldl` $\otimes$ e $[a_1, \ldots, a_n]$ ergibt $(\ldots((e \otimes a_1) \otimes a_2)\ldots) \otimes a_n$
- `foldr` $\otimes$ e $[a_1, \ldots, a_n]$ ergibt $a_1 \otimes (a_2 \otimes (\ldots \otimes (a_n \otimes e)\ldots))$

Implementierung:

```
foldl ::  (a -> b -> a) -> a -> [b] -> a
foldl f e  []     = e
foldl f e (x:xs) = foldl f (e `f` x) xs

foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f e []     = e
foldr f e (x:xs) = x `f` (foldr f e xs)
```

foldl und foldr sind identisch, wenn die Elemente und der Operator $\otimes$ ein Monoid mit neutralem Element e bilden.

Standard-Listenfunktionen (17)

Concat:

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

Beachte: `foldl` bei append wäre ineffizienter!

```
sum     = foldl (+) 0
product = foldl (*) 1
```

haben schlechten Platzbedarf, besser strikte Variante von `foldl`:

```
foldl' :: (a -> b -> a) -> a -> [b] -> a
foldl' f e []     = e
foldl' f e (x:xs) = let e' = e `f` x in e' `seq` foldl' f e' xs
```

Standard-Listenfunktionen (18)

Beachte die Allgemeinheit der Typen von foldl / foldr

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

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

z.B. sind alle Elemente ungerade?

```
foldl (\xa xb -> xa && (odd xb)) True
```

`xa` und `xb` haben verschiedene Typen!

Analog mit foldr:

```
foldr (\xa xb -> (odd xa) && xb) True
```

Standard-Listenfunktionen (19)

Varianten von foldl, foldr:

```
foldr1                  :: (a -> a -> a) -> [a] -> a
foldr1 _ []             =  error "foldr1 on an empty list"
foldr1 _ [x]            =  x
foldr1 f (x:xs)         =  f x (foldr1 f xs)

foldl1                  :: (a -> a -> a) -> [a] -> a
foldl1 f (x:xs)         =  foldl f x xs
foldl1 _ []             =  error "foldl1 on an empty list"
```

Beispiele:

```
maximum :: (Ord a) => [a] -> a
maximum xs = foldl1 max xs

minimum :: (Ord a) => [a] -> a
minimum xs = foldl1 min xs
```

Standard-Listenfunktionen (20)

Scanl, Scanr: Zwischenergebnisse von foldl, foldr

- `scanl` $\otimes$ e $[a_1, a_2, ..., a_n] = [e, e \otimes a_1, (e \otimes a_1) \otimes a_2, \ldots]$
- `scanr` $\otimes$ e $[a_1, a_2, ..., a_n] = [\ldots, a_{n-1} \otimes (a_n \otimes e), a_n \otimes e, e]$

Es gilt:

- `last (scanl f e xs) = foldl f e xs`
- `head (scanr f e xs) = foldr f e xs`.

Standard-Listenfunktionen (21)

```
scanl :: (a -> b -> a) -> a -> [b] -> [a]
scanl f e xs = e:(case xs of
                    [] ->  []
                    (y:ys) -> scanl f (e `f` y) ys)

scanr :: (a -> b -> b) -> b -> [a] -> [b]
scanr _ e []           =  [e]
scanr f e (x:xs)       =  f x q : qs
                           where qs@(q:_) = scanr f e xs
```

Anmerkung: „As“-Pattern `Var@Pat`

```
*> scanr (++) [] [[1,2],[3,4],[5,6],[7,8]]
[[1,2,3,4,5,6,7,8],[3,4,5,6,7,8],[5,6,7,8],[7,8],[]]
*> scanl (++) [] [[1,2],[3,4],[5,6],[7,8]]
[[],[1,2],[1,2,3,4],[1,2,3,4,5,6],[1,2,3,4,5,6,7,8]]
*> scanl (+) 0 [1..10]
[0,1,3,6,10,15,21,28,36,45,55]
```

Standard-Listenfunktionen (22)

Beispiele zur Verwendung von `scan`:

Fakultätsfolge:

```
faks  = scanl (*) 1  [1..]
```

Z.B.

```
*> take 5 faks
[1,1,2,6,120]
```

Standard-Listenfunktionen (22)

Beispiele zur Verwendung von `scan`:

Fakultätsfolge:

```
faks  = scanl (*) 1  [1..]
```

Z.B.

```
*> take 5 faks
[1,1,2,6,120]
```

Funktion, die alle Restlisten einer Liste berechnet:

```
tails xs = scanr (:) []  xs
```

Z.B.

```
*> tails [1,2,3]
[[1,2,3],[2,3],[3],[]]
```

Standard-Listenfunktionen (23)

Partitionieren einer Liste

```
partition p xs = (filter p xs, remove p xs)
```

Effizienter:

```
partition :: (a -> Bool) -> [a] -> ([a], [a])
partition p [] = ([],[])
partition p (x:xs)
  | p x       = (x:r1,r2)
  | otherwise = (r1,x:r2)
  where (r1,r2) = partition p xs
```

Quicksort mit `partition`

```
quicksort :: (Ord a) => [a] -> [a]
quicksort []  = []
quicksort [x] = [x]
quicksort (x:xs) = let (kleiner,groesser) = partition (<x) xs
                   in quicksort kleiner ++ (x:(quicksort groesser))
```

Listen als Ströme (1)

- Listen in Haskell können unendlich lang sein
- Daher kann man Listen auch als Ströme auffassen

Listen als Ströme (1)

- Listen in Haskell können unendlich lang sein
- Daher kann man Listen auch als Ströme auffassen
- Bei der Stromverarbeitung muss man aufpassen:
  Nie versuchen die gesamte Liste auszuwerten
- D.h. Funktionen sollen strom-produzierend sein.
- Grobe Regel:
  Funktion `f ::[Int]->[Int]` ist strom-produzierend, wenn `take n (f list)`
  für jede unendliche Liste `list` und jedes `n` terminiert

Listen als Ströme (1)

- Listen in Haskell können unendlich lang sein
- Daher kann man Listen auch als Ströme auffassen
- Bei der Stromverarbeitung muss man aufpassen:
  Nie versuchen die gesamte Liste auszuwerten
- D.h. Funktionen sollen strom-produzierend sein.
- Grobe Regel:
  Funktion `f ::[Int]->[Int]` ist strom-produzierend, wenn `take n (f list)`
  für jede unendliche Liste `list` und jedes `n` terminiert
- Ungeeignet daher: `reverse`, `length`, `fold`
- Geeignet: `map`, `filter`, `zipWith`, `take`, `drop`

Listen als Ströme (2)

Einige Stromfunktionen für Strings:

- `words :: String -> [String]`
  Zerlegen einer Zeichenkette in eine Liste von Wörtern
- `lines :: String -> [String]`
  Zerlegen einer Zeichenkette in eine Liste der Zeilen
- `unlines :: [String] -> String`
  Einzelne Zeilen in einer Liste zu einem String zusammenfügen (mit Zeilenumbrüchen)

Beispiele:

```
*> words "Haskell ist eine funktionale Programmiersprache"
["Haskell","ist","eine","funktionale","Programmiersprache"]
*> lines "1234\n5678\n90"
["1234","5678","90"]
*> unlines ["1234","5678","90"]
"1234\n5678\n90\n"
```

Listen als Ströme (2)

Mischen zweier sortierter Ströme

```
merge :: (Ord t) => [t] -> [t] -> [t]
merge []       ys  = ys
merge xs       []  = xs
merge a@(x:xs) b@(y:ys)
  | x <= y     = x:merge xs b
  | otherwise = y:merge a ys
```

Beispiel:

```
*> merge [1,3,5,6,7,9] [2,3,4,5,6]
[1,2,3,3,4,5,5,6,6,7,9]
```

Listen als Ströme (3)

Doppelte Elemente entfernen

```
nub xs  = nub' xs []
  where
    nub' [] _           = []
    nub' (x:xs) seen
        | x `elem` seen = nub' xs seen
        | otherwise     = x : nub' xs (x:seen)
```

Anmerkungen:

- `seen` merkt sich die bereits gesehenen Elemente
- Laufzeit von `nub` ist quadratisch

```
elem e []    = False
elem e (x:xs)
  | e == x    = True
  | otherwise = elem e xs
```

Listen als Ströme (4)

Doppelte Elemente aus sortierter Liste entfernen:

```
nubSorted (x:y:xs)
  | x == y    = nubSorted (y:xs)
  | otherwise = x:(nubSorted (y:xs))
nubSorted y = y
```

ist linear in der Länge der Liste.

Listen als Ströme (5)

Mischen der Vielfachen von 3,5 und 7:

```
*> nubSorted $ merge (map (3*) [1..])
*>              (merge (map (5*) [1..]) (map (7*) [1..]))
 [3,5,6,7,9,10,12,14,15,18,20,..
```

Listen als Wörterbuch

Lookup

```
lookup                  :: (Eq a) => a -> [(a,b)] -> Maybe b
lookup  key []          =  Nothing
lookup  key ((x,y):xys)
    | key == x          =  Just y
    | otherwise         =  lookup key xys
```

Beispiele:

```
*> lookup 5 [(1,'A'), (2,'B'), (4,'C'), (5,'F')]
Just 'F'
*> lookup 3 [(1,'A'), (2,'B'), (4,'C'), (5,'F')]
Nothing
```

Listen als Mengen (1)

Any und All: Wie Quantoren

```
any _ []     = False
any p (x:xs)
 | (p x)     = True
 | otherwise = any xs
```

```
all _ []     = True
all p (x:xs)
 | (p x)     = all xs
 | otherwise = False
```

Beispiele:

```
*> all even [1,2,3,4]
False
*> all even [2,4]
True
*> any even [1,2,3,4]
True
```

Listen als Mengen (2)

Delete: Löschen eines Elements

```
delete :: (Eq a) => a -> [a] -> [a]
delete e (x:xs)
 | e == x    = xs
 | otherwise = x:(delete e xs)
```

Mengendifferenz: \\

```
(\\) :: (Eq a) => [a] -> [a] -> [a]
(\\) =  foldl (flip delete)
```

dabei dreht `flip` dreht die Argumente einer Funktion um:

```
flip :: (a -> b -> c) -> b -> a -> c
flip f a b = f b a
```

Listen als Mengen (2b)

Beispiele:

```
*> delete 3 [1,2,3,4,5,3,4,3]
[1,2,4,5,3,4,3]
*> [1,2,3,4,4] \\ [9,6,4,4,3,1]
[2]
*> [1,2,3,4] \\ [9,6,4,4,3,1]
[2]
```

Listen als Mengen (3)

Vereinigung und Schnitt

```
union :: (Eq a) => [a] -> [a] -> [a]
union xs ys =  xs ++ (ys \\ xs)

intersect :: (Eq a) => [a] -> [a] -> [a]
intersect xs ys = filter (\y -> any (== y) ys) xs
```

```
*> union [1,2,3,4,4] [9,6,4,3,1]
[1,2,3,4,4,9,6]
*> union [1,2,3,4,4] [9,6,4,4,3,1]
[1,2,3,4,4,9,6]
*> union [1,2,3,4,4] [9,9,6,4,4,3,1]
[1,2,3,4,4,9,6]
*> intersect [1,2,3,4,4] [4,4]
[4,4]
*> intersect [1,2,3,4] [4,4]
[4]
```

ConcatMap

Konkateniert die Ergebnislisten: concatMap

```
concatMap :: (a -> [b]) -> [a] -> [b]
concatMap f = concat . map f
```

```
*> concatMap (\x-> take x [1..]) [3..7]
[1,2,3,1,2,3,4,1,2,3,4,5,1,2,3,4,5,6,1,2,3,4,5,6,7]
```

List Comprehensions

- Spezielle Syntax zur Erzeugung und Verarbeitung von Listen
- ZF-Ausdrücke (nach der Zermelo-Fränkel Mengenlehre)

Syntax: `[Expr | qual1,...,qualn]`

- `Expr`: ein Ausdruck
- $FV($`Expr`$)$ sind durch `qual1,...,qualn` gebunden
- `quali` ist:
  - ein Generator der Form `pat <- Expr`, oder
  - ein Guard, d.h. ein Ausdruck booleschen Typs,
  - oder eine Deklaration lokaler Bindungen der Form `let x1=e1,...,xn=en` (ohne `in`-Ausdruck!) ist.

List Comprehensions: Beispiele

- Liste der natürlichen Zahlen: `[x | x <- [1..]]`
- Kartesisches Produkt: `[ (wert,name) | wert <-[1..3], name <- ['a'..'b']]` erzeugt `[(1,'a'),(1,'b'),(2,'a'),(2,'b'),(3,'a'),(3,'b')]`.
- Reihenfolge wichtig: `[ (wert,name) | name <- ['a'..'b'], wert <-[1..3]]` erzeugt `[(1,'a'),(2,'a'),(3,'a'),(1,'b'),(2,'b'),(3,'b')]`.
- `[(x,y) | x <- [1..], y <- [1..]]` erzeugt $(\mathbb{N} \times \mathbb{N})$

Beispiele (2)

```
*> take 10 [(x,y) | x <- [1..], y <- [1..]]
[(1,1),(1,2),(1,3),(1,4),(1,5),(1,6),(1,7),(1,8),(1,9),(1,10)]|
*> [x | x <- [1..], odd x]
[1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31, ^CInterrupted
*> take 20 [y | x <- [1..], let y = x*x, y > 100, y < 1000]
[121,144,169,196,225,256,289,324,361,400,441,484,529,576,625,676,729,784,841,900]
*> [a | (a,_,_,_) <- [(x,x,y,y) | x <- [1..3], y <- [1..3]]]
[1,1,1,2,2,2,3,3,3]
```

Map, Filter, Concat mit List Comprehensions

```
map f xs = [f x | x <- xs]

filter p xs = [x | x <- xs, p x]

concat xss = [y | xs <- xss, y <- xs]
```

List Comprehensions: Quicksort

```
qsort (x:xs) =    qsort [y | y <- xs, y <= x]
               ++ [x]
               ++ qsort [y | y <- xs, y > x]
qsort x = x
```

Übersetzung in ZF-freies Haskell

```
[ e | True ]          =   [e]

[ e | q ]             =   [ e | q, True ]

[ e | b, Q ]          =   if b then [ e | Q ] else []

[ e | p <- l, Q ]     =   let ok p = [ e | Q ]
                              ok _ = []
                          in concatMap ok l

[ e | let decls, Q ]  =   let decls in [ e | Q ]
```

wobei

- `ok` eine neue Variable,
- `b` ein Guard,
- `q` ein Generator, eine lokale Bindung oder ein Guard (nicht `True`)
- `Q` eine Folge von Generatoren, Deklarationen und Guards.

Übersetzung in ZF-freies Haskell: Beispiel

```
[x*y | x <- xs, y <- ys, x > 2, y < 3]
```

```
=  let ok x = [x*y | y <- ys, x > 2, y < 3]
       ok _ = []
   in concatMap ok xs

=  let ok x = let ok' y = [x*y | x > 2, y < 3]
                  ok' _ = []
              in concatMap ok' ys
       ok _ = []
   in concatMap ok xs
```

Übersetzung in ZF-freies Haskell: Beispiel

```
[x*y | x <- xs, y <- ys, x > 2, y < 3]

=  let ok x = [x*y | y <- ys, x > 2, y < 3]
       ok _ = []
   in concatMap ok xs

=  let ok x = let ok' y = [x*y | x > 2, y < 3]
                  ok' _ = []
              in concatMap ok' ys
       ok _ = []
   in concatMap ok xs
```

Übersetzung in ZF-freies Haskell: Beispiel

```
[x*y | x <- xs, y <- ys, x > 2, y < 3]

=  let ok x = [x*y | y <- ys, x > 2, y < 3]
       ok _ = []
   in concatMap ok xs

=  let ok x = let ok' y = [x*y | x > 2, y < 3]
                  ok' _ = []
              in concatMap ok' ys
       ok _ = []
   in concatMap ok xs
```

Übersetzung in ZF-freies Haskell: Beispiel

```
=  let ok x = let ok' y = if x > 2 then [x*y | y < 3] else []
                  ok' _ = []
              in concatMap ok' ys
       ok _ = []
   in concatMap ok xs
=  let ok x = let ok' y = if x > 2 then [x*y | y < 3, True] else []
                  ok' _ = []
              in concatMap ok' ys
       ok _ = []
   in concatMap ok xs
=  let ok x = let ok' y = if x > 2 then
                           (if y < 3 then [x*y | True] else [])
                         else []
                  ok' _ = []
              in concatMap ok' ys
       ok _ = []
   in concatMap ok xs
```

Übersetzung in ZF-freies Haskell: Beispiel

```
=  let ok x = let ok' y = if x > 2 then [x*y | y < 3] else []
                  ok' _ = []
              in concatMap ok' ys
       ok _ = []
   in concatMap ok xs
=  let ok x = let ok' y = if x > 2 then [x*y | y < 3, True] else []
                  ok' _ = []
              in concatMap ok' ys
       ok _ = []
   in concatMap ok xs
=  let ok x = let ok' y = if x > 2 then
                            (if y < 3 then [x*y | True] else [])
                          else []
                  ok' _ = []
              in concatMap ok' ys
       ok _ = []
   in concatMap ok xs
```

Übersetzung in ZF-freies Haskell: Beispiel

```
=  let ok x = let ok' y = if x > 2 then [x*y | y < 3] else []
                  ok' _ = []
              in concatMap ok' ys
       ok _ = []
   in concatMap ok xs
=  let ok x = let ok' y = if x > 2 then [x*y | y < 3, True] else []
                  ok' _ = []
              in concatMap ok' ys
       ok _ = []
   in concatMap ok xs
=  let ok x = let ok' y = if x > 2 then
                            (if y < 3 then [x*y | True] else [])
                          else []
                  ok' _ = []
              in concatMap ok' ys
       ok _ = []
   in concatMap ok xs
```

Übersetzung in ZF-freies Haskell: Beispiel

```
=  let ok x = let ok' y = if x > 2 then
                            (if y < 3 then [x*y] else [])
                          else []
                   ok' _ = []
               in concatMap ok' ys
       ok _ = []
   in concatMap ok xs
```

Die Übersetzung ist nicht optimal, da Listen generiert und wieder abgebaut werden und bei `x <- xs` unnötige Pattern-Fallunterscheidung

Rekursive Datenstrukturen: Bäume in Haskell

Rekursive Datenstrukturen: Bäume

Binäre Bäume mit (polymorphen) Blattmarkierungen:

```
data BBaum a = Blatt a | Knoten (BBaum a) (BBaum a)
 deriving(Eq,Show)
```

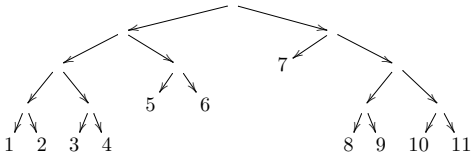
`BBaum` ist Typkonstruktor, `Blatt` und `Knoten` sind Datenkonstruktoren

Rekursive Datenstrukturen: Bäume

Binäre Bäume mit (polymorphen) Blattmarkierungen:

```
data BBaum a = Blatt a | Knoten (BBaum a) (BBaum a)
 deriving(Eq,Show)
```

`BBaum` ist Typkonstruktor, `Blatt` und `Knoten` sind Datenkonstruktoren




```
beispielBaum =
  Knoten (Knoten (Knoten
                   (Knoten (Blatt 1) (Blatt 2))
                   (Knoten (Blatt 3) (Blatt 4)))
                 (Knoten (Blatt 5) (Blatt 6)))
         (Knoten (Blatt 7)
                 (Knoten
                   (Knoten (Blatt 8) (Blatt 9))
                   (Knoten (Blatt 10) (Blatt 11))))
```

Funktionen auf Bäumen (1)

Summe aller Blattmarkierungen

```
bSum (Blatt a) = a
bSum (Knoten links rechts) = (bSum links) + (bSum rechts)
```

Ein Beispielaufruf:

```
*> bSum beispielBaum
66
```

Funktionen auf Bäumen (1)

Summe aller Blattmarkierungen

```
bSum (Blatt a) = a
bSum (Knoten links rechts) = (bSum links) + (bSum rechts)
```

Ein Beispielaufruf:

```
*> bSum beispielBaum
66
```

Liste der Blätter

```
bRand (Blatt a) = [a]
bRand (Knoten links rechts) = (bRand links) ++ (bRand rechts)
```

Test:

```
*> bRand beispielBaum
[1,2,3,4,5,6,7,8,9,10,11]
```

Funktionen auf Bäumen (2)

Map auf Bäumen

```
bMap f (Blatt a) = Blatt (f a)
bMap f (Knoten l r) = Knoten (bMap f l) (bMap f r)
```

Beispiel:

```
 *> bMap (^2) beispielBaum
 Knoten (Knoten (Knoten (Knoten (Blatt 1) (Blatt 4))
  (Knoten (Blatt 9) (Blatt 16))) (Knoten (Blatt 25) (Blatt 36)))
  (Knoten (Blatt 49) (Knoten (Knoten (Blatt 64) (Blatt 81))
  (Knoten (Blatt 100) (Blatt 121))))
```

Die Anzahl der Blätter eines Baumes:

```
anzahlBlaetter = bSum . bMap (\x -> 1)
```

Funktionen auf Bäumen (3)

Element-Test

```
bElem e (Blatt a)
 | e == a            = True
 | otherwise         = False
bElem e (Knoten l r) = (bElem e l) || (bElem e r)
```

Einige Beispielaufrufe:

```
*> 11 `bElem` beispielBaum
 True
*> 1 `bElem` beispielBaum
 True
*> 20 `bElem` beispielBaumm
 False
*> 0 `bElem` beispielBaum m
 False
```

Funktionen auf Bäumen (4)

Fold auf Bäumen

```
bFold op (Blatt a) = a
bFold op (Knoten a b) = op (bFold op a) (bFold op b)
```

Damit kann man z.B. die Summe und das Produkt berechnen:

```
*> bFold (+) beispielBaum
66
*> bFold (*) beispielBaum
39916800
```

Funktionen auf Bäumen (5)

Allgemeineres Fold auf Bäumen:

```
foldbt :: (a -> b -> b) -> b -> BBaum a  -> b
foldbt op a (Blatt x)    = op x a
foldbt op a (Knoten x y) = (foldbt op (foldbt op a y) x)
```

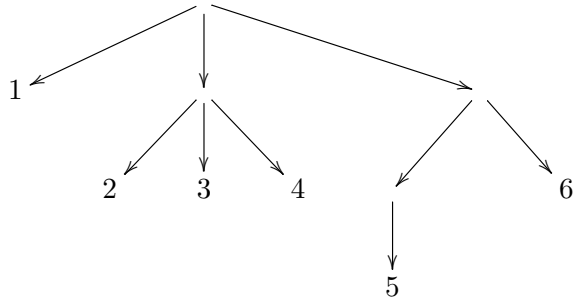
Der Typ des Ergebnisses kann anders sein als der Typ der Blattmarkierung
Zum Beispiel: Rand eines Baumes:

```
*> foldbt (:) [] beispielBaum
[1,2,3,4,5,6,7,8,9,10,11]
```

N-äre Bäume

```
data NBaum a = NBlatt a | NKnoten [NBaum a]
 deriving(Eq,Show)

beispiel = NKnoten [NBlatt 1,
                    NKnoten [NBlatt 2, NBlatt 3, NBlatt 4],
                    NKnoten [NKnoten [NBlatt 5], NBlatt 6]]
```



Bäume mit Knotenmarkierungen

Beachte: `BBaum` und `NBaum` haben nur Markierungen der Blätter!

Bäume mit Markierung aller Knoten

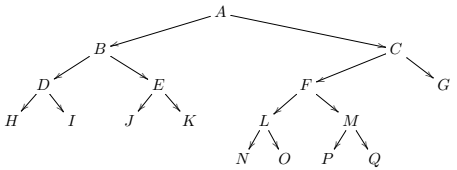
```
data BinBaum a = BinBlatt a | BinKnoten a (BinBaum a) (BinBaum a)
 deriving(Eq,Show)
```

Bäume mit Knotenmarkierungen

Beachte: `BBaum` und `NBaum` haben nur Markierungen der Blätter!

Bäume mit Markierung aller Knoten

```
data BinBaum a = BinBlatt a | BinKnoten a (BinBaum a) (BinBaum a)
 deriving(Eq,Show)
```



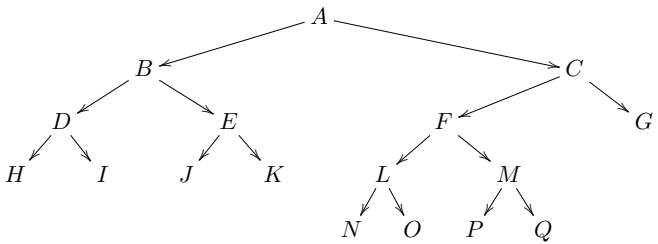

```
beispielBinBaum =
 BinKnoten 'A'
  (BinKnoten 'B'
     (BinKnoten 'D' (BinBlatt 'H') (BinBlatt 'I'))
     (BinKnoten 'E' (BinBlatt 'J') (BinBlatt 'K'))
  )
  (BinKnoten 'C'
     (BinKnoten 'F'
        (BinKnoten 'L' (BinBlatt 'N') (BinBlatt 'O'))
        (BinKnoten 'M' (BinBlatt 'P') (BinBlatt 'Q'))
     )
     (BinBlatt 'G')
  )
```

Funktionen auf BinBaum (1)

Knoten in Preorder-Reihenfolge (Wurzel, links, rechts):

```
preorder :: BinBaum t -> [t]
preorder (BinBlatt a)      = [a]
preorder (BinKnoten a l r) = a:(preorder l) ++ (preorder r)
```

```
*> preorder beispielBinBaum
"ABDHIEJKCFLNOMPQG"
```

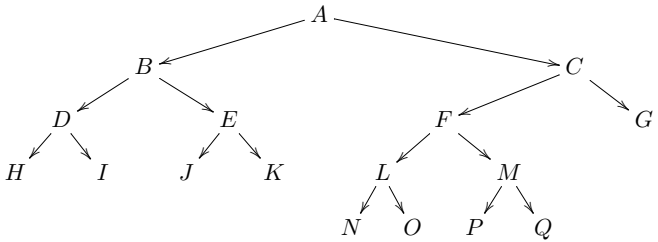


Funktionen auf BinBaum (2)

Knoten in Inorder-Reihenfolge (links, Wurzel, rechts):

```
inorder :: BinBaum t -> [t]
inorder (BinBlatt a) = [a]
inorder (BinKnoten a l r) = (inorder l) ++ a:(inorder r)
```

```
*> inorder beispielBinBaum
"HDIBJEKANLOFPMQCG"
```

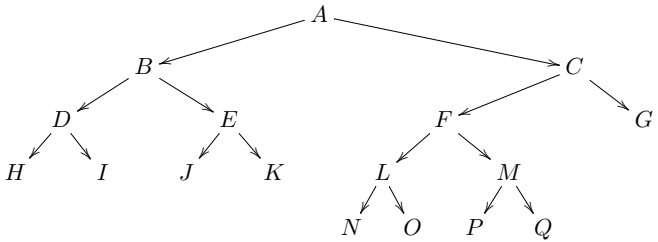


Funktionen auf BinBaum (3)

Knoten in Post-Order Reihenfolge (links, rechts, Wurzel)

```
postorder (BinBlatt a) = [a]
postorder (BinKnoten a l r) =
   (postorder l) ++ (postorder r) ++ [a]
```

```
*> postorder  beispielBinBaum
"HIDJKEBNOLPQMFGCA"
```



Funktionen auf BinBaum (2)

Level-Order (Stufenweise, wie Breitensuche)

Schlecht (alle Stufen werden von oben her neu berechnet)

```
levelorderSchlecht b =
   concat [nodesAtDepthI i b | i <- [0..depth b]]
 where
  nodesAtDepthI 0 (BinBlatt a) = [a]
  nodesAtDepthI i (BinBlatt a) = []
  nodesAtDepthI 0 (BinKnoten a l r) = [a]
  nodesAtDepthI i (BinKnoten a l r) =
         (nodesAtDepthI (i-1) l) ++ (nodesAtDepthI (i-1) r)
  depth (BinBlatt _) = 0
  depth (BinKnoten _ l r) = 1+(max (depth l) (depth r))


*> levelorderSchlecht  beispielBinBaum
"ABCDEFGHIJKLMNOPQ"
```

Funktionen auf BinBaum (3)

Level-Order (Stufenweise, wie Breitensuche)

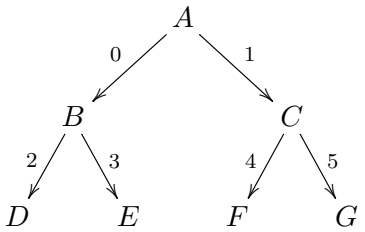
Besser:

```
levelorder b = loForest [b]
 where
  loForest xs = map root xs ++ loForest (concatMap subtrees xs)
  root (BinBlatt a) = a
  root (BinKnoten a _ _) = a
  subtrees (BinBlatt _) = []
  subtrees (BinKnoten _ l r) = [l,r]


*> levelorder  beispielBinBaum
"ABCDEFGHIJKLMNOPQ"
```

Bäume mit Knoten und Kantenmarkierungen

```
data BinBaumMitKM a b =
   BiBlatt a
 | BiKnoten a (b, BinBaumMitKM a b) (b,BinBaumMitKM a b)
 deriving(Eq,Show)
```




```
beispielBiBaum =
  BiKnoten 'A'
     (0,BiKnoten 'B'
           (2,BiBlatt 'D')
           (3,BiBlatt 'E'))
     (1,BiKnoten 'C'
           (4,BiBlatt 'F')
           (5,BiBlatt 'G'))
```

Funktion auf BinBaumMitKM

Map mit 2 Funktionen

```
biMap f g (BiBlatt a) = BiBlatt (f a)
biMap f g (BiKnoten a (kl,links) (kr,rechts) =
 BiKnoten (f a) (g kl, biMap f g links) (g kr, biMap f g rechts)
```

Beispiel

```
*> biMap toLower even beispielBiBaum
BiKnoten 'a'
  (True,BiKnoten 'b' (True,BiBlatt 'd') (False,BiBlatt 'e'))
  (False,BiKnoten 'c' (True,BiBlatt 'f') (False,BiBlatt 'g'))
```

Anmerkung zum $-Operator

Definition:

```
f $ x = f x
```

wobei Priorität ganz niedrig, z.B.

```
map (*3) $ filter (>5) $ concat [[1,1],[2,5],[10,11]]
```

wird als

```
map (*3) (filter (>5) (concat [[1,1],[2,5],[10,11]]))
```

geklammert

Syntaxbäume

Auch Syntaxbäume sind Bäume

Beispiel: Einfache arithmetische Ausdrücke:

$$\begin{array}{lcl} E & ::= & (E + E) \mid (E * E) \mid Z \\ Z & ::= & 0Z' \mid \ldots \mid 9Z' \\ Z' & ::= & \varepsilon \mid Z \end{array}$$

Als Haskell-Datentyp (infix-Konstruktoren müssen mit : beginnen)

```
data ArEx = ArEx :+: ArEx
          | ArEx :*: ArEx
          | Zahl Int
```

alternativ

```
data ArEx = Plus ArEx ArEx
          | Mult ArEx ArEx
          | Zahl Int
```

Z.B. $(3 + 4) * (5 + (6 + 7))$ als Objekt vom Typ `ArEx`:

```
((Zahl 3) :+: (Zahl 4)) :*: ((Zahl 5) :+: ((Zahl 6) :+: (Zahl 7)))
```

Interpreter für Arithmetische Ausdrücke

```
interpArEx :: ArEx -> Int
interpArEx (Zahl i) = i
interpArEx (e1 :+: e2) = (interpArEx e1) + (interpArEx e2)
interpArEx (e1 :*: e2) = (interpArEx e1) * (interpArEx e2)
```

Z.B.:

```
*Main> interpArEx (((Zahl 3) :+: (Zahl 4)) :*:
                   ((Zahl 5) :+: ((Zahl 6) :+: (Zahl 7))))
126
```

Typdefinitionen mit data, type, newtype

Typdefinitionen in Haskell

Drei syntaktische Möglichkeiten in Haskell

- `data`
- `type`
- `newtype`

Verwendung von `data` haben wir bereits ausgiebig gesehen

Typdefinitionen in Haskell (2)

`type`; Variante von Typdefinitionen.

Mit `type` definiert man Typsynonyme, d.h:

Neuer Name für bekannten Typ

Beispiele:

```
type IntCharPaar = (Int,Char)
type Studenten = [Student]
type MyList a = [a]
```

Typdefinitionen in Haskell (2)

`type`; Variante von Typdefinitionen.

Mit `type` definiert man Typsynonyme, d.h:

Neuer Name für bekannten Typ

Beispiele:

```
type IntCharPaar = (Int,Char)
```

```
type Studenten = [Student]
```

```
type MyList a = [a]
```

Sinn davon: Verständlicher, z.B.

```
alleStudentenMitA :: Studenten -> Studenten
alleStudentenMitA = map nameMitA
```

Typdefinitionen in Haskell (3)

Typdefinition mit `newtype`:

- `newtype` ist sehr ähnlich zu `type`
- Mit `newtype`-definierte Typen dürfen eigene Klasseninstanz für Typklassen haben
- Mit `type`-definierte Typen aber nicht.
- Mit `newtype`-definierte Typen haben einen neuen Konstruktor
- case und pattern match für Objekte von einem mit `newtype`-definierten Typ sind immer erfolgreich.

Typdefinitionen in Haskell (4)

Beispiel für *newtype*:

```
newtype Studenten' = St [Student]
```

Diese Definition kann man sich vorstellen als

```
data Studenten'    = St [Student]
```

Ist aber nicht semantisch äquivalent dazu, da Terminierungsverhalten anders

Vorteil `newtype` vs. `data`: Der Compiler weiß, dass es nur ein Typsynonym ist und kann optimieren: `case`-Ausdrücke dazu werden eliminiert und durch direkte Zugriffe ersetzt.

Beispiel für newtype vs. data

```
newtype Studenten' = St [Student]
data  Studenten'' = St'' [Student]
```

Compiler entfernt `St` überall, Effekt:

```
*Main> case undefined of {St _ -> 1}
1
*Main> case undefined of {St'' _ -> 1}
*** Exception: Prelude.undefined
```

Funktionen in Haskell

Funktionen

Funktionen werden durch Gleichungen definiert:

```
double1 x = x + x              -- Funktion mit 1 Argument
fun x y z = x + y * double1 z          -- mit 3 Argumenten
add       = \x y -> x + y      -- Anonyme Fkt. 2 Argumente

twice f x = f x x              -- Higher-order function
double2 y = twice (+) y
double3   = twice (+)          -- Partielle Applikation
```

- In Quellcode-Datei: Alle Top-Level Definition in gleicher Spalte
- Funktionsanwendung durch Leerzeichen: `foo 1 2 3`.
  Klammerung ist links-assoziativ: `foo 1 2 3` = `(((foo 1) 2) 3)`

Funktionen (2)

Definition (Funktion)

Eine partielle Funktion $f : A \to B$ ordnet einer Teilmenge $A' \subset A$ einen Wert aus B zu, und ist ansonsten undefiniert.
Wir bezeichnen A als Quellbereich, A' als Definitionsbereich und B als Zielbereich. Für totale Funktionen gilt $A = A'$.

Im allgemeinen: Programmiere totale Funktionen, z.B. durch Verwendung von `error` für die undefinierten Fälle.

Funktionstypen

```
foo :: (Int,Int) -> (Int,Int)
foo (x,y) = (x+y,x-y)

bar :: Int -> (Int -> (Int,Int))
bar x y = (x+y,x-y)
```

- Klammerkonvention: Funktionstypen implizit rechtsgeklammert.
  `Int -> Int -> Int` äquivalent zu `Int -> (Int -> Int)` und verschieden zu `(Int -> Int) -> Int`.
- In Haskell darf man partiell anwenden.
  Z.B. ist `(bar 1)` eine Funktion des Typs `Int -> (Int,Int)`
- Funktionen sind also normale Werte in einer funktionalen Sprache.

Funktionsdefinition in Haskell

funktionsName :: Typ_1 -> Typ_2 -> ... -> Typ_n -> *Ergebnistyp*
funktionsName var_1 var_2 ... var_n = *expr*

- *funktionsName* beginnt mit einem Kleinbuchstabe oder _
- Typdeklaration ist optional
- var_1, var_2, ..., var_n sind die formalen Parameter
- *expr* ist der Funktionsrumpf.
- Anstelle von Variablen können auch Pattern verwendet werden.
- Mehrere Definitionszeilen sind möglich, erste passende wird genommen, Abarbeitung von oben nach unten (analog bei Guards).

Beispiele

Zweites Element einer Liste:

```
second (x:y:xs) = y
second _ = error "not enough elements"
```

Falsch:

```
second' _ = error "not enough elements"
second' (x:y:xs) = y
```

Richtig:

```
second'' []       = error "not enough elements"
second'' [x]      = error "not enough elements"
second'' (x:y:xs) = y
```

Anonyme Funktionen

- Anonyme Funktion = Funktion ohne Namen (auch Abstraktion genannt)
- Herkunft aus dem Lambda-Kalkül (Alonzo Church, 1936)
- Abstraktion im Lambda-Kalkül $\lambda x_1, \ldots, x_n.e$
- In Haskell: \ statt λ, Leerzeichen statt , und `->` statt .

Beispiele:

	Lambda-Kalkül	Haskell
Identitätsfunktion	$\lambda x.x$	`\x -> x`
Nachfolgerfunktion	$\lambda x.x + 1$	`\x -> x+1`
Wurzelfunktion	$\lambda y.\sqrt{y}$	`\y -> sqrt y`
Dreistelliges Plus	$\lambda x, y, z.x + y + z$	`\x y z -> x + y + z`

Ausdrücke

Ausdrücke werden gebildet durch:

- Anwendungen
- `if-then-else`-Ausdrücke:

  `if` *Ausdruck* `then` *Ausdruck* `else` *Ausdruck*
- `case`-Ausdrücke zum Zerlegen von Daten:

  `case` *Ausdruck* `of` *Alternativen*

  wobei *Alternativen* `case`-Alternativen der Form

  Pattern -> *Ausdruck* oder *Pattern Match*
  - *Pattern* z.B. `((x,y):xs)`
  - *Match* bezeichnet eine Folge der Form

    | $Guard_1$ -> $Ausdruck_1$

    | ...

    | $Guard_n$ -> $Ausdruck_n$
  - Wert des `case`: Erstes *Pattern*, das passt und erster Guard, der zu `True` auswertet

Beispiel

```
myFun frucht p personendb =
  case frucht of
    Banane l h kruemmung
      | Just x <- lookup p personendb -> Right x
    _  -> Left "Fehler"
```

Aufrufe:

```
*Main> myFun (Banane 1 2 3) 1 [(1,"Horst")]
Right "Horst"
*Main> myFun (Apfel (1,2)) 1 [(1,"Horst")]
Left "Fehler"
*Main> myFun (Banane 1 2 3) 10 [(1,"Horst")]
Left "Fehler"
```

Ausdrücke (2)

- `let`-Ausdrücke der Form

  `let` *Bindungen* `in` *Ausdruck*

  wobei
  - *Bindungen* sind Bindungen der Form

    Funktionsname $par_1 \ldots par_n$ = *Ausdruck*
  - Die Bindungen in `let`-Ausdrücken sind rekursiv: Gültigkeitsbereich der Funktionsnamen sind alle rechten Seiten der Bindungen sowie der `in`-Ausdruck.
- `do`-Notation: Diese erläutern wir später
- Datenkonstruktoren, Listen, List-Comprehensions

Programmieren mit Let

Lokale Funktionsdefinitionen mit let:

$$\begin{aligned}
&\texttt{let} \quad f_1\ x_{1,1}\ \ldots\ x_{1,n_1} = e_1 \\
&\qquad\ \ f_2\ x_{2,1}\ \ldots\ x_{2,n_2} = e_2 \\
&\qquad\ \ \ldots \\
&\qquad\ \ f_m\ x_{m,1}\ \ldots\ x_{m,n_m} = e_m \\
&\texttt{in} \ldots
\end{aligned}$$

Z.B.

```
f x y = let quadrat z = z*z in quadrat x + quadrat y

quadratfakultaet x = let quadrat z = z*z
                         fakq    0 = 1
                         fakq    x = (quadrat x)*fakq (x-1)
                     in fakq x
```

Sharing mit Let

Bindungen der Form $var = e$ werden nur einmal berechnet (es sei denn, der Compiler optimiert dies durch inlining). Z.B.

```
verdopplefak x =
   let fak 0  = 1
       fak x  = x*fak (x-1)
       fakx   = fak x
   in fakx + fakx
```

Ein Aufruf `verdopplefak 100` wird nur einmal `fak 100` berechnen.
Im Gegensatz dazu:

```
verdopplefakLangsam x =
   let fak 0  = 1
       fak x  = x*fak (x-1)
   in fak x + fak x
```

Pattern-Matching mit `let`

Auf einer linken Seite kann auch ein Pattern stehen: z.B. $(a, b) = \ldots$, damit kann man komfortabel auf einzelne Komponenten zugreifen.
Z.B. Summe und Produkt berechnen:

```
sumprod 1 = (1,1)
sumprod n =
   let (s',p') = sumprod (n-1)
   in (s'+n,p'*n)
```

Memoization

Beispiel:

```
fib 0 = 0
fib 1 = 1
fib i = fib (i-1) + fib (i-2)
```

Mit Memoization:

```
-- Fibonacci mit Memoization

fibM i =
  let fibs = [0,1] ++ [fibs!!(i-1) + fibs!!(i-2) | i <- [2..]]
  in fibs!!i -- i-tes Element der Liste fibs
```

Memoization (2)

Variante nach let-over-lambda-shifting:

```
fibM' =
  let fibs = [0,1] ++ [fibs!!(i-1) + fibs!!(i-2) | i <- [2..]]
  in \i -> fibs!!i -- i-tes Element der Liste fibs
```

```
*Main> fibM' 20000
...
(7.27 secs, 29757856 bytes)
*Main> fibM' 20000
...
(0.06 secs, 2095340 bytes)
```

where-Klauseln

Beispiel:

```
sumprod' 1 = (1,1)
sumprod' n = (s'+n,p'*n)
  where (s',p') = sumprod' (n-1)
```

Das `where` ist gültig für die Funktionsdefinition kann und daher für alle Guards wirken, z.B.

```
f x
 | x == 0    = a
 | x == 1    = a*a
 | otherwise = a*f (x-1)
 where a = 10
```

Falsch:

```
f x = \y -> mul
   where mul = x * y
```

da `where` kein Ausdruck ist und `y` daher nicht bekannt in der `where`-Klausel ist

Haskells hierarchisches Modulsystem

Haskells hierarchisches Modulsystem

Aufgaben von Modulen

- Strukturierung / Hierarchisierung
- Kapselung
- Wiederverwendbarkeit

Moduldefinition in Haskell

$$\texttt{module}\ \mathit{Modulname}(\mathit{Exportliste})\ \texttt{where}$$
$$\left.\begin{array}{l}\mathit{Modulimporte},\\ \mathit{Datentypdefinitionen},\\ \mathit{Funktionsdefinitionen},\ldots\end{array}\right\}\ \mathit{Modulrumpf}$$

- Name des Moduls muss mit Großbuchstaben beginnen
- Exportliste enthält die nach außen sichtbaren Funktionen und Datentypen
- Dateiname = Modulname.hs (bei hierarchischen Modulen Pfad+Dateiname = Modulname.hs)
- Ausgezeichnetes Modul `Main` muss Funktion `main :: IO ()` exportieren

Import / Export

Beispiel:

```
module Spiel where
  data Ergebnis = Sieg | Niederlage | Unentschieden
  berechneErgebnis a b = if a > b then Sieg
                         else if a < b then Niederlage
                              else Unentschieden
  istSieg Sieg = True
  istSieg _    = False
  istNiederlage Niederlage = True
  istNiederlage _          = False
```

- Da keine Exportliste: Es werden alle Funktionen und Datentypen exportiert
- Außer importierte

Exporte

Exportliste kann enthalten

- Funktionsname (in Präfix-Schreibweise) Z.B.: ausschließlich `berechneErgebnis` wird exportiert

```
module Spiel(berechneErgebnis) where
```

- Datentypen mittels `data` oder `newtype`, drei Möglichkeiten
  - Nur `Ergebnis` in die Exportliste:

    ```
    module Spiel(Ergebnis) where
    ```

    Typ `Ergebnis` wird exportiert, die Datenkonstruktoren, d.h. `Sieg`, `Niederlage`, `Unentschieden` jedoch nicht
  - ```
    module Spiel(Ergebnis(Sieg, Niederlage))
    ```

    Typ `Ergebnis` und die Konstruktoren `Sieg` und `Niederlage` werden exportiert, nicht jedoch `Unentschieden`.
  - Durch den Eintrag `Ergebnis(..)`, wird der Typ mit sämtlichen Konstruktoren exportiert.
    ```

Exporte (2)

Exportliste kann enthalten

- Typsynonyme, die mit `type` definiert wurden

```
module Spiel(Result) where
  ... wie vorher ...
  type Result = Ergebnis
```

- Importierte Module:

```
module Game(module Spiel, Result) where
  import Spiel
  type Result = Ergebnis
```

`Game` exportiert alle Funktionen, Datentypen und Konstruktoren, die auch `Spiel` exportiert sowie zusätzlich noch den Typ `Result`.

Importe

Einfachste Form: Importiert alle Einträge der Exportliste

`import` *Modulname*

Andere Möglichkeiten:

- Explizites Auflisten der zu importierenden Einträge:

```
module Game where
  import Spiel(berechneErgebnis, Ergebnis(..))
  ...
```

- Explizites Ausschließen einzelner Einträge:

```
module Game where
  import Spiel hiding(istSieg,istNiederlage)
  ...
```

Importe (2)

So importierte Funktionen und Datentypen sind mit ihrem unqualifizierten und ihrem qualifizierten Namen ansprechbar.
Qualifizierter Name: *Modulname*.*unqualifizierter Name*

```
module A(f) where
    f a b = a + b
module B(f) where
    f a b = a * b
module C where
    import A
    import B
    g = f 1 2 + f 3 4 -- funktioniert nicht!


Prelude> :l C.hs
ERROR C.hs:4 - Ambiguous variable occurrence "f"
*** Could refer to: B.f A.f
```

Importe (3)

Mit qualifizierten Namen funktioniert es:

```
module C where
  import A
  import B
  g = A.f 1 2 + B.f 3 4
```

Mit Schlüsselwort `qualified` kann man nur die qualifizierten Namen importieren:

```
module C where
  import qualified A
  g = f 1 2   -- f ist nicht sichtbar


Prelude> :l C.hs
ERROR C.hs:3 - Undefined variable "f"
```

Importe (4)

Lokale Aliase: Schlüsselwort as:

```
import LangerModulName as C
```

Importe (5)

Übersicht:

Import-Deklaration	definierte Namen
`import M`	`f, g, M.f, M.g`
`import M()`	keine
`import M(f)`	`f, M.f`
`import qualified M`	`M.f, M.g`
`import qualified M()`	keine
`import qualified M(f)`	`M.f`
`import M hiding ()`	`f, g, M.f, M.g`
`import M hiding (f)`	`g, M.g`
`import qualified M hiding ()`	`M.f, M.g`
`import qualified M hiding (f)`	`M.g`
`import M as N`	`f, g, N.f, N.g`
`import M as N(f)`	`f, N.f`
`import qualified M as N`	`N.f, N.g`

Hierarchische Modulstruktur

- Modulnamen mit Punkten versehen geben Hierarchie an: z.B.

  ```
  module A.B.C
  ```

- Allerdings ist das rein syntaktisch
  (es gibt keine Beziehung zwischen Modul `A.B` und `A.B.C`)

- Beim Import:

  ```
  import A.B.C
  ```

  sucht der Compiler nach dem File namens `C.hs` im Pfad `A/B/`