

## Übungsblatt zu Haskell

Learn You a Haskell for Great Good!

### 1 Aufwärmübungen in GHCi

#### Aufgabe 1. Verschachtelte Tupel

Benutze die vordefinierten Funktionen `fst :: (a, b) -> a` und `snd :: (a, b) -> b`, um das Textzeichen aus `(1, ('a', "foo"))` zu extrahieren.

#### Aufgabe 2. Medianbestimmung

Sei `xs` eine unsortierte Liste von ungerade vielen Zahlen, z. B. `let xs = [3, 7, -10, 277, 89, 13, 22, -100, 1]`. Schreibe einen Ausdruck, der den Median (das Element in der Mitte in einer Sortierung) von `xs` berechnet. Verwende dazu die Funktionen `Data.List.sort`, `length`, `div` und `!!`.

#### Aufgabe 3. Der Eulen-Operator erster Ordnung

Was könnte der Ausdruck `(.) . (.)` bewirken? Finde es heraus mit Hilfe von GHCi!

*Hinweis.* Du brauchst nicht verstehen, warum der Eulen-Operator funktioniert. Nur, was er tut.

### 2 Spiel und Spaß mit Listenfunktionen

#### Aufgabe 4. Groß- und Kleinschreibung

- a) Verwende `map`, um einen String in eine Liste von Booleans zu verwandeln, die angeben, ob das entsprechende Zeichen im String ein Groß- oder Kleinbuchstabe war. Beispielsweise soll bei `"aBCde"` das Ergebnis `[True, False, False, True, True]` sein.

*Hinweis.* Verwende `isLower :: Char -> Bool` aus `Data.Char`.

- b) Verwende `and :: [Bool] -> Bool`, um zu prüfen, ob ein String nur aus Kleinbuchstaben besteht.
- c) Berechne die Anzahl Kleinbuchstaben in einem gegebenen String.

#### Aufgabe 5. Typfehler

Erkläre den Typfehler, der bei folgendem Ausdruck auftritt: `\x -> x x`

*Bemerkung.* Es gibt einen tieferen und guten Grund dafür, wieso man die dabei auftretenden „unendlichen Typen“ ablehnt. <http://copilotco.com/mail-archives/haskell-cafe.2006/msg05831.html>

### 3 Funktionsdefinitionen

#### Aufgabe 6. *Fizz buzz*

Beim Spiel *Fizz buzz* stehen alle Spieler im Kreis. Reihum wird nun von eins hoch gezählt. Anstatt von Zahlen, die durch drei teilbar sind, muss man jedoch „fizz“ sagen und „buzz“ bei Zahlen, die durch fünf teilbar sind. Ist eine Zahl sowohl durch drei als auch durch fünf teilbar, so sagt man „fizz buzz“. Wer einen Fehler macht, scheidet aus.

Implementiere die unendliche Liste

```
fizzbuzz :: [String]
fizzbuzz = [ "1", "2", "fizz", "4", "buzz", "fizz", "7", "8", "fizz", "buzz"
            , "11", "fizz", "13", "14", "fizz buzz", "16", ... ]
```

#### Aufgabe 7. *Origami*

Implementiere eine Funktion `maximum' :: [Int] -> Int`, die die größte Zahl in einer Liste zurückliefert oder `0`, falls die Liste leer ist oder alle Zahlen negativ sind. Verwende dazu `foldl :: (b -> a -> b) -> b -> [a] -> b`.

#### Aufgabe 8. *Fibonacci-Zahlen*

Die Fibonacci-Folge  $0, 1, 1, 2, 3, 5, 8, \dots$  ist bekanntermaßen rekursiv definiert durch:

*Die nullte Fibonacci-Zahl ist null, die erste Fibonacci-Zahl ist eins,  
jede weitere Fibonacci-Zahl ist die Summe ihrer beiden Vorgänger.*

- Verwende diese Definition direkt, um eine Haskell-Funktion `fib :: Int -> Int` zu schreiben, die die  $n$ -te Fibonacci-Zahl berechnet.
- Berechne `fib 35`. Was ist das Problem?
- Implementiere `fibs :: [Int]`, eine unendliche Liste aller Fibonacci-Zahlen. Lass dir mit `take 100 fibs` die ersten hundert Folgenglieder in GHCi ausgeben.  
*Hinweis.* Du bekommst massig Bonuspunkte, wenn du `zipWith` verwendest.

#### Aufgabe 9. *Die Collatz-Vermutung*

- Beginne mit irgendeiner natürlichen Zahl  $n > 0$ .
- Ist  $n$  gerade, so nimm als Nächstes  $\frac{n}{2}$ ,
- Ist  $n$  ungerade, so nimm als Nächstes  $3n + 1$ .
- Wiederhole die Vorgehensweise mit der erhaltenen Zahl.

Zum Beispiel erhält man für  $n = 19$  die Folge

19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1, 4, 2, 1, ...

- Schreibe eine Funktion `collNext :: Int -> Int`, welche die Collatz-Iteration durchführt.
- Implementiere die Funktion `collSeq :: Int -> [Int]`, die die Folge der Collatz-Iterierten berechnet: `collSeq 10 = [10, 5, 16, 8, 4, 2, 1, 4, 2, 1, 4, 2, 1, ...]`

Die bisher ungelöste Collatz-Vermutung besagt, dass man ausgehend von jeder beliebigen Zahl  $n$  irgendwann bei 1 landet.

- Schreibe die Funktion `collTest :: Int -> Bool`, welche die Collatz-Vermutung für eine Eingabe  $n$  testet. Falls die Collatz-Vermutung für  $n$  falsch sein sollte, so muss die Funktion nicht terminieren.

d) Überprüfe die Collatz-Vermutung für alle natürlichen Zahlen kleiner als 100000.

### Aufgabe 10. *Die Prelude*

Informiere dich, was folgende Funktionen aus der Standardbibliothek tun, und implementiere so viele wie du willst neu:

`head`, `tail`, `init`, `last`, `length`, `reverse`, `(++)`, `iterate`, `map`, `filter`, `intersperse`, `concat`, `zipWith`, `repeat`, `and`, `takeWhile`, `dropWhile`, `maximum`

### Aufgabe 11. *Pointless/pointfree programming*

Vereinfache folgende Funktionsdefinitionen:

```
multMany a xs = map (\x -> a*x) xs
filterMap f g xs = filter f (map g xs)
```

### Aufgabe 12. *Run-Length-Encoding*

Zur effizienten Speicherung von Daten mit vielen Zeichenwiederholungen bietet es sich an, nicht jedes einzelnes Zeichen, sondern Zeichen mit der jeweiligen Anzahl Wiederholungen zu speichern:

```
Prelude> encode ['a', 'a', 'a', 'a', 'b', 'c', 'c', 'a', 'a', 'd', 'e', 'e', 'e', 'e']
[(4, 'a'), (1, 'b'), (2, 'c'), (2, 'a'), (1, 'd'), (4, 'e')]
```

Implementiere die Funktion `encode :: String -> [(Int, Char)]` und die Umkehrfunktion `decode :: [(Int, Char)] -> String!`

### Aufgabe 13. *Längste Teilfolge*

Schreibe eine Funktion `longestSubsequence :: (a -> Bool) -> [a] -> [a]`. Diese soll die längste zusammenhängende Unterliste in einer Liste berechnen, für die das übergebene Prädikat vom Typ `a -> Bool` den Wert `True` liefert. Wenn beispielsweise `xs :: [Date]` die Liste der letzten 365 Tage ist, und `p :: Date -> Bool` angibt, ob eine gewisse Benutzerin an einem gegebenen Tag auf GitHub aktiv war, so berechnet `longestSubsequence p xs` die längste Strähne auf GitHub.

### Aufgabe 14. *(Für Gelangweilte.) Die State-Monade*

(Die folgende Aufgabe nutzt viele Konzepte, die bisher nicht eingeführt wurden und ist daher nur für Leute gedacht, die sich gerade langweilen und sich selbst in diese Themen einlesen wollen.)

- Führe in deinem ghci `import Control.Monad.State` aus, damit die Funktionen, die du für diese Aufgabe brauchen wirst, geladen werden.
- Informiere dich, was `(>>=) :: Monad m => m a -> (a -> m b) -> m b` und `return :: Monad m => a -> m a` tun. (Etwa auf <https://curry-club-augsburg.de/posts/2015-10-27-monaden-in-haskell.html>)
- Die oben genannten Funktionen kann man für beliebige Monaden verwenden. Ein Beispiel für eine Monade ist die Listen-Monade. Die obigen Typsignaturen werden dann zu `(>>=) :: [a] -> (a -> [b]) -> [b]` und `return :: a -> [a]`. Verstehe, was folgender Haskell-Code tut:

```
and10more :: Int -> [Int]
and10more x = [x, x + 10]
onlyifodd :: Int -> [Int]
onlyifodd x = if x `mod` 2 /= 0 then [x] else []
[1, 2, 3] >>= and10more >>= onlyifodd
```

- d) Informiere dich, was `get :: MonadState s m => m s`, `put :: MonadState s m => s -> m ()`, `state :: MonadState s m => (s -> (a, s)) -> m a` und `runState :: State s a -> s -> (a, s)` tun. (Etwa auf obigem Link.)
- e) Verstehe, was folgender Haskell-Code tut:

```
import Control.Monad.State

factStep :: (Int, Int) -> ((), (Int, Int))
factStep (result, counter) = ((), (result * counter, counter - 1))

factContinueCondition :: (Int, Int) -> Bool
factContinueCondition (result, counter) = counter >= 1

factMonadic :: State (Int, Int) Int
factMonadic = do
    bool <- fmap factContinueCondition get
    if bool
    then do
        state factStep
        factMonadic
    else fmap fst get

fact :: Int -> Int
fact n = fst $ runState factMonadic (1, n)
```

- f) Wähle eine Aufgabe deiner Wahl, die du bereits gelöst hast und löse sie mit Hilfe der State-Monade auf imperative Weise (d.h. in ähnlichem Stil wie obiges Beispiel).

### Aufgabe 15. Kettenbrüche

Jede reelle Zahl  $r \in \mathbb{R}$  kann als *unendlicher Kettenbruch*

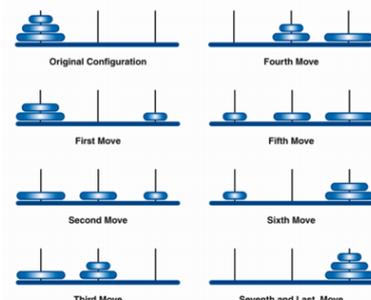
$$r = b_0 + \frac{1}{b_1 + \frac{1}{b_2 + \frac{1}{b_3 + \frac{1}{\ddots}}}}$$

mit  $b_0 \in \mathbb{Z}$  und  $b_i \in \mathbb{N}$ ,  $i \geq 1$  geschrieben werden. Überlege dir einen Algorithmus, der die unendliche Folge der  $b_i$ 's berechnet und implementiere ihn in Haskell. Überprüfe deinen Algorithmus anhand der Kettenbruchentwicklung von  $\pi$  (in Haskell: `pi`).

*Hinweis.* Du wirst die Funktionen `floor :: Double -> Int` und `fromIntegral :: Int -> Double` brauchen.

### Aufgabe 16. Türme von Hanoi

Die Aufgabe bei Hanoi ist es, einen Turm von Scheiben von einem Steckplatz zu einem anderen zu transportieren. Dabei darf man nur je eine Scheibe einzeln bewegen und es darf niemals eine größere Scheibe auf einer kleineren liegen. Man hat einen Steckplatz als Zwischenlager zur Verfügung. Implementiere eine Funktion `toh :: Int -> [(Int, Int)]`, welche die Hanoi-Züge (Bewegungen von einzelnen Scheiben) berechnet, mit denen man einen Turm einer gewissen Größe von Steckplatz 1 zu Steckplatz 3 mit Steckplatz 2 als Zwischenlager bewegen kann.



```
Prelude> toh 3
[(1,3), (1,2), (3,2), (1,3), (2,1), (2,3), (1,3)]
```

*Tip.* Definiere eine Hilfsfunktion `moveTower :: Int -> (Int, Int, Int) -> [(Int, Int)]`, sodass `moveTower n (x, y, z)` die nötigen Schritte ausgibt, um  $n$  Scheiben von  $x$  nach  $y$  unter Verwendung von  $z$  als Zwischenlager zu bewegen.

### Aufgabe 17. Mirpzahlen

Eine *Mirpzahl* ist eine Zahl, die sowohl von vorne nach hinten als auch hinten nach vorne gelesen eine Primzahl ist. Zum Beispiel sind 13 und 17 Mirpzahlen. Die Primzahl 19 ist dagegen keine Mirpzahl.

Schreibe ein Programm, das die Liste aller Mirpzahlen berechnet.

*Hinweis.* Definiere eine Hilfsfunktion `isPrime :: Integer -> Bool` und verwende die Funktionen `show :: Integer -> [Char]`, `reverse :: [Char] -> [Char]` und `read :: [Char] -> Integer`.

### Aufgabe 18. Primzahlen mit dem Sieb des Eratosthenes

- Schreibe ein Haskell-Programm, das die unendliche Liste aller Primzahlen berechnet. Verwende dazu zunächst deine Funktion `isPrime :: Integer -> Bool` aus der vorherigen Aufgabe und die allgemeine Listenfunktion `filter`.
- Stelle nun deinen Algorithmus auf das Sieb des Eratosthenes um. Dabei geht man wie folgt vor: Zunächst betrachtet man die Liste `[2..]`. Das erste Element notiert man als Primzahl. Dann streicht man alle Vielfachen dieses Elements. Die erste jetzt noch stehende Zahl ist die nächste Primzahl. Dann streicht man deren Vielfache; und so weiter.

### Aufgabe 19. Strikte und nicht-strikte Funktionen

Eine Haskell-Funktion `f :: A -> B` heißt genau dann *strikt*, wenn sie, angewendet auf einen nicht-terminierenden Ausdruck, selbst nicht terminiert. In Symbolen schreibt man das gerne so:  $f \perp = \perp$ .

Nicht-terminierende Ausdrücke sind zum Beispiel:

- `undefined`
- `error "Division durch Null"`
- `last [0..]`

*Keine* Beispiele für nicht-terminierende Ausdrücke sind:

- `"abc"`
- `[0..]`
- `['a', undefined, 'b']`

Dass auch die letzten beiden Ausdrücke keine nicht-terminierenden Ausdrücke sind, kann man sich so klarmachen: Mit ihnen kann man prima arbeiten, ohne Endlosschleifen zu produzieren. Zum Beispiel ist `length ['a', undefined, 'b']` kein Fehler, sondern einfach `3` (in das mittlere Element der Liste muss nicht hineingeschaut werden).

Die Funktion `id :: Char -> Char` ist strikt. Der *Robot Monkey Operator*, die Funktion `(:[]) :: Char -> [Char]`, ist es dagegen nicht. In den meisten Programmiersprachen ist jede Funktion strikt, in Haskell jedoch nicht.

Welche der folgenden Funktionen sind strikt, welche nicht?

- `reverse :: [a] -> [a]`
- `("abc" ++)` :: `[Char] -> [Char]`

c) `(++ "abc") :: [Char] -> [Char]`

### Aufgabe 20. Der Fixpunktoperator

Im Modul `Control.Monad.Fix` ist folgende Funktion vordefiniert (welche noch nichts mit Monaden zu tun hat):

```
fix :: (a -> a) -> a
fix f = let x = f x in x
-- alternative Schreibweise:
-- fix f = x where x = f x
```

Diese Funktion berechnet von einer gegebenen Funktion `f` ihren *kleinsten Fixpunkt* – das ist ein Wert `x :: a`, sodass `f x` gleich `x` ist. Gibt es mehrere Fixpunkte, so berechnet `fix` den „am wenigsten definierten“ (das ist mit „kleinstem“ gemeint). Anschaulich kann man sich `fix f` als `f (f (f (...)))` vorstellen.

- Was ist `fix ('a':)`?
- Was ist `fix ("abc" ++)`?
- Was ist `fix id`? Was ist allgemeiner der kleinste Fixpunkt einer strikten Funktion?
- Was ist `fix $ \xs -> 0 : 1 : zipWith (+) xs (tail xs)`?
- Wieso berechnet folgender Code die Fibonacci-Zahlen?

```
fib :: Integer -> Integer
fib = fix $ \fib' n ->
  case n of
    0      -> 0
    1      -> 1
    _      -> fib' (n-1) + fib' (n-2)
```

- Wenn du Spaß an solchen Dingen findest, entziffere auch noch folgenden Code:

```
fix $ (0:) . scanl (+) 1
```

- Erkläre, wie die Ausgabe von `fix show` zustande kommt. Dabei ist `show :: String -> String` die Funktion, die eine Zeichenkette als validen Haskell-Code formatiert. Zum Beispiel ist `show "foo"` die aus fünf Zeichen bestehende Zeichenkette `"\"foo\""`.

## 4 Eigene Datentypen

### Aufgabe 21. Binäre Bäume

Im Folgenden verwenden wir folgende Definition für binäre Bäume, deren Verzweigungsknoten mit Werten vom Typ `Int` dekoriert sind.

```
data Tree = Nil | Fork Int Tree Tree
  deriving (Show)
```

- Schreibe eine Funktion, die die Gesamtzahl Blätter eines Baums berechnet:  
`numberOfLeaves :: Tree -> Int`.
- Schreibe eine Funktion, die die Höchsttiefe eines Baums berechnet.
- Schreibe eine Funktion, die die Int-Werte der Verzweigungsknoten in einer Reihenfolge deiner Wahl als Liste zurückgibt.

### Aufgabe 22. Binäre Bäume bilden einen Funktor

- a) Verallgemeinere die vorherige Aufgaben auf Bäume, die Werte von einem beliebigen Typ `a` statt `Int` tragen. Vervollständige dazu zunächst folgende Definition:

```
data Tree a = Nil | ...
    deriving (Show)
```

- b) Implementiere eine Funktion `tmap :: (a -> b) -> Tree a -> Tree b`.

### Aufgabe 23. Unendliche Bäume

- a) Schreibe eine Funktion `cutOff :: Int -> Tree a -> Tree a`, die eine Maximaltiefe und einen Baum als Argumente nimmt und einen neuen Baum zurückgibt, der sich aus dem gegebenen durch Abschneidung bei der gegebenen Maximaltiefe ergibt.
- b) Definiere eine Funktion, die eine unendliche Liste von Werten nimmt und einen Baum zurückgibt, auf dessen Verzweigungsknoten die Elemente der Liste sitzen. Suche dir selbst aus, in welcher Reihenfolge die Listenelemente auf dem Baum platziert werden sollen.

### Aufgabe 24. Der Stern-Brocot-Baum (für Fans von Kettenbrüchen)

Informiere dich auf Wikipedia über den Stern-Brocot-Baum und implementiere ein Haskell-Programm, das diesen unendlichen Baum berechnet. Hole dir gegebenenfalls einen (stark spoilernden) Tipp ab.

### Aufgabe 25. Termbäume

- a) Implementiere einen Datentyp für Funktionsterme. Zum Beispiel soll

$$(x \cdot x + 3) - x$$

so repräsentiert werden: `Sub (Add (Mul Var Var) (Lit 3)) Var`.

- b) Schreibe eine Funktion `eval :: Exp -> Double -> Double`, die in einem gegebenen Term für die Variable  $x$  einen konkreten Wert einsetzt.
- c) Schreibe eine Funktion `diff :: Exp -> Exp`, die einen gegebenen Funktionsterm ableitet. Zum Beispiel soll `diff (Mul Var Var)` im Wesentlichen äquivalent sein zu `Mul (Lit 2) Var`.

### Aufgabe 26. Isomorphe Typen

Manche Typen lassen sich verlustfrei ineinander umwandeln, zum Beispiel die folgenden beiden:

```
data Bool    = False | True  -- schon vordefiniert
data Aussage = Falsch | Wahr
```

Man spricht in einem solchen Fall von *zueinander isomorphen Typen*. Die Umwandlungsfunktionen heißen *Isomorphismen* und können in diesem Fall wie folgt definiert werden:

```
iso :: Bool -> Aussage
iso False = Falsch
iso True  = Wahr
```

```
osi :: Aussage -> Bool
osi Falsch = False
osi Wahr   = True
```

Das charakteristische an diesen beiden Funktionen ist, dass `osi . iso == id` und `iso . osi == id`.

Folgende Typen sind jeweils zueinander isomorph. Implementiere auf analoge Weise Funktionen `iso` und `osi`, die das bezeugen!

- `(a, b)` versus `(b, a)`
- `((a, b), c)` versus `(a, (b, c))`
- `(a, Either b c)` versus `Either (a, b) (a, c)`
- `a -> (b, c)` versus `(a -> b, a -> c)`
- `(a, b) -> c` versus `a -> b -> c` (nach dieser Isomorphie ist der Curry Club benannt!)

## 5 Typklassen

### Aufgabe 27. Eigene Show-Instanzen

Für Debugging-Zwecke oder auch zum Datenaustausch ist die Show-Klasse nützlich, deren Definition in etwa die folgende ist:

```
class Show a where
  show :: a -> String
```

Bei der Deklaration eines neuen Datentyps hat man die Möglichkeit, mit einer `deriving`-Klausel den Compiler anzuweisen, automatisch eine geeignete Show-Instanz zu generieren:

```
data Tree a = Nil | Fork a (Tree a) (Tree a)
  deriving (Show)
```

In dieser Aufgabe aber sollst du den dafür nötigen Boilerplate-Code von Hand schreiben. Such dir einen Datentyp deiner Wahl aus und schreibe eine individuelle Show-Instanz für ihn.

### Aufgabe 28. Die Monoid-Typklasse

Das Modul `Data.Monoid` definiert die Monoid-Typklasse:

```
class Monoid a where
  mempty  :: a
  mappend :: a -> a -> a
  mconcat :: [a] -> a
```

Ihr gehören solche Typen an, die eine sog. *Monoidstruktur* tragen. (Wenn du diesen Begriff nicht kennst, dann frag kurz nach!) Das neutrale Element soll durch `mempty` angegeben werden, die Monoidoperation durch `mappend`. Die Funktion `mconcat` soll gleich mehrere Elemente miteinander verknüpfen.<sup>1</sup>

- Gebe einer Nachimplementierung des Listen-Datentyps, etwa `data List a = Nil | Cons a (List a)` eine Monoid-Instanz. Vergiss nicht, zu Beginn deines Programmtexts mit `import Data.Monoid` die Definition der Monoid-Klasse zu laden.

<sup>1</sup>Die Funktionen `mappend` und `mconcat` lassen sich gegenseitig ausdrücken. Fällt dir ein Grund ein, wieso trotzdem beide Funktionen Teil der Klasse sind? Hätte man nicht auch einfach `mconcat` außerhalb der Klasse definieren können?

b) Implementiere folgende Funktion:

```
cata :: (Monoid m) => (a -> m) -> ([a] -> m)
```

*Bemerkung.* Das ist der Herzstück des Beweises, dass der Monoid der endlichen Listen mit Einträgen aus `a` der freie Monoid über `a` ist.

### Aufgabe 29. Sortierung nach mehreren Kriterien

Oft steht man vor folgendem Problem: Eine Liste von Dingen soll nach mehreren Kriterien sortiert werden. Etwa zunächst nach Nachname, unter gleichen Nachnamen aber nach Vorname und unter gleichem Namen nach Geburtsdatum. Die in Haskell idiomatische Herangehensweise an dieses Problem verwendet ... Monoide!

a) Schlage den `Ordering`-Typ nach.

b) Reimplementiere die Funktion

```
comparing :: (Ord a) => (b -> a) -> b -> b -> Ordering
```

aus dem Modul `Data.Ord`. Sie kann zum Beispiel so verwendet werden:

```
import Data.List
```

```
data Person = MkPerson
  { lastName  :: String
  , givenName :: String
  , birthday  :: String
  }
  deriving (Show)
```

```
sortPersons :: [Person] -> [Person]
sortPersons = sortBy (comparing lastName)
-- sortBy hat den Typ (a -> a -> Ordering) -> [a] -> [a].
```

c) Trägt ein Typ `a` eine Monoidstruktur, so auch der Typ `e -> a` der Funktionen von `e` nach `a`. Bestätige das, indem du folgenden Code vervollständigst:

```
instance (Monoid a) => Monoid (e -> a) where
  -- ...
```

Da diese Instanz schon in `Data.Monoid` vordefiniert ist, musst du für diese Teilaufgabe den Import von `Data.Monoid` entfernen und die Monoid-Typklasse selbst definieren.

d) Was macht folgender Code? Wieso tut er das? Informiere dich dazu über die Monoid-Instanz von `Ordering` und erinnere dich an die Monoid-Instanz von Funktionstypen.

```
sortBy $ mconcat
  [ comparing lastName
  , comparing firstName
  , comparing birthday
  ]
```

### Aufgabe 30. Endliche Typen

Manche Typen fassen nur endlich viele Werte, zum Beispiel `Bool` und `Either Bool Bool`. Für solche Typen ist es gelegentlich praktisch, eine vollständige Liste ihrer Werte zu kennen. Aus diesem Grund führen wir folgende Klasse ein:

```
class Finite a where
  elems :: [a]
```

a) Implementiere eine Finite-Instanz für `Bool`.

b) Implementiere folgende allgemeinen Instanzen:

```
instance (Finite a, Finite b) => Finite (a,b)      where ...
instance (Finite a, Finite b) => Finite (Maybe a)  where ...
instance (Finite a, Finite b) => Finite (Either a b) where ...
```

c) Wenn du Lust auf eine Herausforderung hast, dann implementiere auch folgende Instanz. Sie ist für die weiteren Teilaufgaben aber nicht nötig.

```
instance (Eq a, Finite a, Finite b) => Finite (a -> b) where ...
```

d) Implementiere eine Funktion `exhaustiveTest :: (Finite a) => (a -> Bool) -> Bool`.

e) Die Gleichheit zweier Funktionen (vom selben Typ) ist im Allgemeinen nicht entscheidbar, denn zwei Funktionen sind genau dann gleich, wenn sie auf allen Eingabewerten übereinstimmen. Um das zu überprüfen, muss man im Allgemeinen unendlich viele Fälle in Augenschein nehmen. Wenn der Quelltyp aber endlich ist, geht es doch. Implementiere also:

```
instance (Finite a, Eq b) => Eq (a -> b) where ...
```

*Bemerkung.* Allgemeiner geht es, wenn der Quelltyp *kompakt* ist. Suche nach „scheinbar unmöglichen Haskell-Programmen“.

### Aufgabe 31. Abzählbare Typen

Manche Typen sind zwar nicht endlich, aber immer noch *abzählbar*: Das heißt, dass es eine unendliche Liste gibt, in der alle Werte des Typs vorkommen. Zum Beispiel ist der Typ `Integer` abzählbar, denn in der Liste `[0, 1, -1, 2, -2, ...]` kommen alle ganzen Zahlen vor.

a) Definiere nach dem Vorbild der Finite-Typklasse aus der vorherigen Aufgabe eine Countable-Typklasse.

b) Implementiere eine Countable-Instanz von `Integer`.

c) Vervollständige folgenden Code:

```
instance (Countable a, Countable b) => Countable (a,b) where ...
```

d) Vervollständige folgenden Code (schwierig!):

```
instance (Countable a) => Countable [a] where ...
```

Dabei soll `[a]` für den Typ der endlichen Listen mit Werten in `a` stehen – obwohl der Typ `[a]` ja auch unendliche Listen enthält. Solche *sozialen Verträge* sind in Haskell leider gelegentlich nötig – man benötigt abhängige Typen und andere Entwicklungen, um sie vollständig zu vermeiden. Sauberer wäre an dieser Stelle, einen neuen Datentyp `FiniteList a` zu definieren, der isomorph zum gewöhnlichen Listentyp ist, aber den sozialen Vertrag an zentraler Stelle kundtut.

### Aufgabe 32. Überabzählbare Typen

Diese Aufgabe richtet sich nur an Leute, die das sog. *Cantorsche Diagonalargument* und die *Russelsche Antinomie* kennen. Sorry! Bei Gelegenheit suchen wir eine einführende Referenz.

Wir definieren ein Typalias für Mengen:

```
type Set a = a -> Bool
```

```
-- Ist `f :: Set a`, so soll `f x == True` bedeuten, dass `x` in
-- der Menge `f` liegt.
```

- a) Setze in diesem Modell die leere Menge, die Universalmenge (welche alle Werte überhaupt enthält) und die Menge, die nur ein bestimmtes Element enthält, um. Welche Voraussetzung an den Typ `a` musst du im letzten Teil stellen?
- b) Implementiere folgende Funktionen:
- ```
member      :: a      -> Set a -> Bool
union       :: Set a -> Set a -> Set a
intersection :: Set a -> Set a -> Set a
complement  :: Set a -> Set a
```
- c) Setze die Russellsche Antinomie in Haskell um. Definiere also eine Menge all derjenigen Mengen, die sich nicht selbst enthalten. Wie äußert sich das paradoxe Verhalten in Haskell?
- d) Setze das Cantorsche Diagonalargument in Haskell um. Definiere also eine Funktion
- ```
cantor :: (a -> Set a) -> Set a
```
- die Folgendes leistet: Für jede Funktion `f :: a -> Set a` soll `cantor f` eine Menge sein, die nicht im Bild (in der Wertemenge) von `f` enthalten ist.
- e) Bonusfrage zum Grübeln: Die vorherige Teilaufgabe zeigt, dass es in Haskell überabzählbare Typen gibt. Andererseits ist die Menge der Haskell-Programme abzählbar. Wie passt das zusammen?
- f) Literatur dazu: ein toller Blog-Post von sigfpe. <http://blog.sigfpe.com/2008/01/type-that-should-not-be.html>

## 6 Die IO-Monade und andere Monaden

### Aufgabe 33. IO-Stringmanipulation

Schreibe ein Programm, das eine Zeichenkette einliest und diese rückwärts wieder ausgibt. Die Funktion `reverse :: [a] -> [a]` könnte hilfreich sein, wenn du dir sie nicht selbst schreiben willst.

### Aufgabe 34. Überprüfung von Nutzereingaben

Schreibe ein Programm, das den Benutzer solange nach einer Zahl fragt, bis dieser eine Zahl angibt, die durch 3 teilbar ist. Erinnere dich an die Typklasse `Read`!

### Aufgabe 35. Monadische Schleifen

- a) Schreibe eine Funktion, die eine gegebene IO-Aktion (oder eine andere Art Aktion) eine gewisse Anzahl von Malen wiederholt. Die Typsignatur sollte also `replicateM :: Int -> IO a -> IO [a]` (oder allgemeiner `replicateM :: (Monad m) => Int -> m a -> m [a]` – erinnere dich, dass `IO` eine Instanz von `Monad` ist!) sein.
- b) Schreibe eine Funktion `forM :: (Monad m) => [a] -> (a -> m b) -> m [b]` die das tut, was ihr Name und ihre Typsignatur versprechen.
- c) Schreibe eine Funktion, die es erlaubt, eine IO-Aktion (oder eine andere Art Aktion) unendlich oft zu wiederholen. Die Typsignatur sollte `forever :: (Monad m) => m a -> m b` sein. Erinnere dich, dass `IO` eine Instanz von `Monad` ist!

### Aufgabe 36. Entzuckerung der do-Notation

Schreibe die folgenden Funktionen ohne Verwendung der do-Notation.

```
f :: IO String
f = do
  text <- getLine
  return text

g :: IO ()
g = do
  text <- readFile "foo.txt"
  writeFile "bar.txt" text

iterateM :: (Monad m) => (a -> m a) -> a -> m b
iterateM f x = do
  x' <- f x
  iterateM f x'

join :: (Monad m) => m (m a) -> m a
join m = do
  m' <- m
  m'

whileM :: (Monad m) => m Bool -> m a -> m [a]
whileM cond m = do
  b <- cond
  if b
    then do
      x <- m
      xs <- whileM cond m
      return (x:xs)
    else return []
```

### Aufgabe 37. Zahlenratespiel

Schreibe ein Programm, das versucht eine Zahl zwischen 0 und 100 zu erraten, die sich die Nutzerin ausgedacht hat. Die Nutzerin soll jeweils bei jedem Rateversuch angeben, ob die geratene Zahl kleiner oder größer als die tatsächliche ist. Du kannst das Problem mit einer binären Suche lösen.

### Aufgabe 38. Die Reader-Monade

Gelegentlich muss ein bestimmter Werte durch viele Funktionen gefädelt werden, zum Beispiel ein Wert, der globale Konfigurationsoptionen enthält:

```
f :: Config -> Int -> Char -> Bool -> Mumble
f config x y z = ...g config x.....h config y.....z...

g :: Config -> Int -> Gabambel
g config x     = ...x...

h :: Config -> Char -> Rainbow
h config y     = ...p config y.....y...
```

```
p :: Config -> Char -> Lollipop
p config y    = ...y...
```

Vielleicht findest du das nervig. Informiere dich über die Reader-Monade, mit der man das vermeiden kann. Die neuen Typsignaturen lauten dann:

```
f :: Int -> Char -> Bool -> Reader Config Mumble
g :: Int -> Reader Config Gabambel
h :: Char -> Reader Config Rainbow
p :: Char -> Reader Config Lollipop
```

Durch Verwendung von Typsynonymen, wie `type R = Reader Config`, könnte man den Code noch übersichtlicher gestalten. Zugriff auf den aktuellen Wert der implizit mitgeführten Konfiguration erhält man mit `ask :: Reader Config Config`:

```
foo :: FairyTale -> Reader Config MyLittlePony
foo x = do
  config <- ask
  let y = ...x...
      ...x...
  return ...
```

## 7 Ideen für kleinere und größere Projekte

### Aufgabe 39. Unicode-Smileys

Schreibe eine Funktion `replaceSmileys :: String -> String`, die alle „;-)“ durch deinen liebsten Unicode-Smiley ersetzt. Auf [unicode-table.com](http://unicode-table.com) kannst du dir jeden Unicode-Smiley kopieren.

Wenn du die `IO`-Monade schon kennst, kannst du diese in einem Programm verwenden, das einen Text einliest und mit dieser Funktion alle Smileys ersetzt und den entstandenen Text ausgibt.

### Aufgabe 40. Einfache Verschlüsselung

Implementiere einen einfachen Verschlüsselungsalgorithmus deiner Wahl, etwa:

- Die Cäsarverschlüsselung, also Verschiebung des Alphabets: `rot :: Integer -> String -> String`. Die Implementation soll folgendes Gesetz erfüllen: `rot (-n) (rot n "foobar") == "foobar"`. Benutze bei der Implementation entweder `chr :: Int -> Char` und `ord :: Char -> Int` aus `Data.Char` oder eine Liste des Alphabets, wie `['A'..'Z'] ++ ['a'..'z'] :: [Char]`. In beiden Fällen kann dir der Modulooperator `a `mod` b` helfen. Beachte die Backticks! Beispiel: `rot 3 "hallo"` evaluiert zu `"kdoor"`.
- Die Vigenèreverschlüsselung. Die funktioniert wie die Cäsarverschlüsselung, nur dass es mehrere „Schlüssel“ (`Integer`) gibt, für jeden `Char` einen: `vigenère :: [Integer] -> String -> String`. Sollten die `Integer` nicht ausreichen, wird wieder beim ersten angefangen. Nützlich ist die Funktion `cycle :: [a] -> [a]`, die aus einer Liste eine unendliche macht, indem sie sie immer wieder wiederholt. Beispiel: `vigenère [1,2,3] "hallo"` evaluiert zu `"icomq"`.

### Aufgabe 41. Häufigkeitsanalyse auf Listen

Schreibe eine Funktion `freqAn :: Eq a => [a] -> [(a, Int)]`, die eine Liste von Tupeln aus einem Element der Eingabeliste und dessen Häufigkeit zurückgibt. Zum Beispiel wäre





```

ys = True  : epsilon (f . (True:))

-- Falls `f` auf jeder Bitfolge False ist, ist `exists f` gleich `Nothing`.
-- Ansonsten ist `exists f` gleich `Just xs`, wobei `xs` eine Bitfolge ist,
-- auf der `f` True ist.
exists :: ([Bool] -> Bool) -> Maybe [Bool]
exists f = if f x then Just x else Nothing
          where x = epsilon f

```

In Aktion sieht das Programm zum Beispiel so aus:

```

Prelude> exists $ \xs -> False
Nothing
Prelude> exists $ \xs -> True
Just [False,False,False,False,...]
Prelude> exists $ \xs -> xs !! 1 && xs !! 2
Just [False,True,True,False,...]

```

#### Aufgabe 47. Ein eigenes assoziatives Datenfeld

Ein assoziatives Datenfeld oder schlicht „Map“ könntest du schon kennen. Es erlaubt innerhalb einer Datenstruktur Beziehungen zwischen einem Wert vom Typ `k` (dem Key) und einem anderem Wert vom Typ `v` (dem Value) zu speichern.

Definiere dir deinen eigenen Datentyp `Map k v` als binären Suchbaum. Jeder `Fork` enthält einen `Key` (Typ `k`) und ein `Value` (Typ `v`). Im linken Kindbaum befinden sich alle Key-Value-Paare, für die gilt `key < aktuellerKey`. Im rechten Kindbaum gilt umgekehrt: `key > aktuellerKey`. Auf Basis dieser Eigenschaften kannst du die folgende Programmierschnittstelle für `Map` implementieren.

```

-- Unser Map-Typ als binärer Baum
data Map k v = Fork k v (Map k v) (Map k v)
              | Nil deriving (Show)

-- Eine konstante Map ohne Inhalte
empty :: Map k v
empty = Nil

-- Fügt eine Beziehung zu einer Map hinzu
insert :: Ord k => k -> v -> Map k v -> Map k v

-- Entfernt eine Beziehung aus der Map
delete :: Ord k => k -> Map k v -> Map k v

-- Schaut ein Element auf Basis eines Schlüssels nach
lookup :: Ord k => k -> Map k v -> Maybe v

-- Nimmt eine Liste von Tupeln und erstellt eine Map mit diesen als Relationen
fromList :: Ord k => [(k, v)] -> Map k v

```

Implementiere nun diese Schnittstelle! Dann überlege dir, wie man Maps am Besten vergleicht. Danach implementiere `instance Eq (Map k v)`. Du musst `import Prelude hiding (lookup)` am Anfang deiner Datei hinzufügen, damit du keine Namenskonflikte bekommst.

Bonusbinärpunkt: Mache dich schlau, wie man eigene Operatoren in Haskell definiert. Dann definiere den Operator `!` als Alias für `lookup`.