Prof. Dr. J. Giesl

S. Dollase, M. Hark, D. Cloerkes

Allgemeine Hinweise:

- Die **Hausaufgaben** sollen in Gruppen von je **2 Studierenden** aus der **gleichen Kleingruppenübung (Tutorium)** bearbeitet werden. **Namen und Matrikelnummern** der Studierenden sind auf jedes Blatt der Abgabe zu schreiben. **Heften bzw. tackern Sie die Blätter oben links!**
- Die **Nummer der Übungsgruppe** muss **links oben** auf das **erste Blatt** der Abgabe geschrieben werden. Notieren Sie die Gruppennummer gut sichtbar, damit wir besser sortieren können.
- Die Lösungen müssen bis **Montag, den 27.01.2020, um 12:00 Uhr** in den entsprechenden Übungskasten eingeworfen werden. Sie finden die Kästen am Eingang Halifaxstr. des Informatikzentrums (Ahornstr. 55). Alternativ können Sie die Lösungen auch vor der Abgabefrist direkt bei Ihrer Tutorin/Ihrem Tutor abgeben.
- Auf diesem Blatt müssen Sie in Haskell programmieren und `.hs`-Dateien anlegen. **Drucken** Sie diese aus **und** laden Sie sie fristgerecht im `RWTHmoodle`-Lernraum "Programmierung (Übung - Tutorium)" hoch.

  Stellen Sie sicher, dass Ihr Programm mit `GHCi` **ausgeführt werden kann**, ansonsten werden keine Punkte vergeben.
- Benutzen Sie in Ihrem Code keine Umlaute, auch nicht in Strings und Kommentaren. Diese führen oft zu Problemen, da diese Zeichen auf verschiedenen Betriebssystemen unterschiedlich behandelt werden. Dadurch kann es dazu führen, dass Ihr Programm bei Ihrer Tutorin/Ihrem Tutor bei Verwendung von Umlauten nicht mehr von `GHCi` akzeptiert wird.

Tutoraufgabe 1 (Datenstrukturen):

In dieser Aufgabe beschäftigen wir uns mit *Kindermobiles*, die man beispielsweise über das Kinderbett hängen kann. Ein Kindermobile besteht aus mehreren Figuren, die mit Fäden aneinander aufgehängt sind. Als mögliche Figuren im Mobile beschränken wir uns hier auf *Sterne*, *Seepferdchen*, *Elefanten* und *Kängurus*.
An Sternen und Seepferdchen hängt keine weitere Figur. An jedem Elefant hängt eine weitere Figur, unter jedem Känguru hängen zwei Figuren. Weiterhin hat jedes Känguru einen Beutel, in dem sich etwas befinden kann (z. B. eine Zahl).
In Abbildung 1 finden Sie zwei beispielhafte Mobiles[1].

a) Implementieren Sie in Haskell einen parametrisierten Datentyp `Mobile a` mit vier Konstruktoren (für Sterne, Seepferdchen, Elefanten und Kängurus), mit dem sich die beschriebenen Mobiles darstellen lassen. Verwenden Sie den Typparameter `a` dazu, den Typen der Känguru-Beutelinhalte festzulegen.

Modellieren Sie dann die beiden in Abbildung 1 dargestellten Mobiles als Ausdruck dieses Datentyps in Haskell. Nehmen Sie hierfür an, dass die gezeigten Beutelinhalte vom Typ `Int` sind.

```
mobileLinks :: Mobile Int          mobileRechts :: Mobile Int
mobileLinks = ...                  mobileRechts = ...
```

Hinweise:

- Für Tests der weiteren Teilaufgaben bietet es sich an, die beiden Mobiles als konstante Funktionen im Programm zu deklarieren.
- Schreiben Sie `deriving Show` an das Ende Ihrer Datentyp-Deklaration. Damit können Sie sich in `GHCi` ausgeben lassen, wie ein konkretes Mobile aussieht.

[1] Für die Grafik wurden folgende Bilder von Wikimedia Commons verwendet:
- Stern `http://commons.wikimedia.org/wiki/File:Crystal_Clear_action_bookmark.png`
- Seepferdchen `http://commons.wikimedia.org/wiki/File:Seahorse.svg`
- Elefant `http://commons.wikimedia.org/wiki/File:African_Elephant_Transparent.png`
- Känguru `http://commons.wikimedia.org/wiki/File:Kangourou.svg`

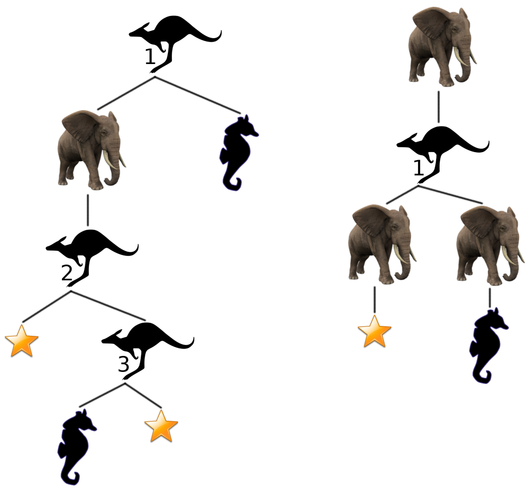


Abbildung 1: Zwei beispielhafte Mobiles.

b) Schreiben Sie in Haskell eine Funktion `count :: Mobile a -> Int`, die die Anzahl der Figuren im Mobile berechnet. Für die beiden gezeigten Mobiles soll also 8 und 6 zurückgegeben werden.

c) Schreiben Sie eine Funktion `liste :: Mobile a -> [a]`, die alle in den Känguru-Beuteln enthaltenen Elemente in einer Liste (mit beliebiger Reihenfolge) zurückgibt. Für das linke Mobile soll also die Liste `[1,2,3]` (oder eine Permutation davon) berechnet werden.

d) Schreiben Sie eine Funktion `greife :: Mobile a -> Int -> Mobile a`. Diese Funktion soll für den Aufruf `greife mobile n` die Figur mit Index `n` im Mobile `mobile` zurückgeben.

Wenn man sich das Mobile als Baumstruktur vorstellt, werden die Indizes entsprechend einer *Tiefensuche*[2] berechnet:

Wir definieren, dass die oberste Figur den Index 1 hat. Wenn ein Elefant den Index n hat, so hat die Nachfolgefigur den Index $n + 1$.

Wenn ein Känguru den Index n hat, so hat die linke Nachfolgefigur den Index $n+1$. Wenn entsprechend dieser Regeln alle Figuren im linken Teil-Mobile einen Index haben, hat die rechte Nachfolgefigur den nächsthöheren Index.

Im linken Beispiel-Mobile hat das Känguru mit Beutelinhalt `3` also den Index 5.

Hinweise:

- Benutzen Sie die Funktion `count` aus Aufgabenteil b).
- Falls der übergebene Index kleiner 1 oder größer der Anzahl der Figuren im Mobile ist, darf sich Ihre Funktion beliebig verhalten.

Aufgabe 2 (Datenstrukturen): (1 + 1 + 1 + 2.5 + 1 + 2.5 = 9 Punkte)

In dieser Aufgabe betrachten wir Binärbaume, deren Blätter einzelne Zeichen und deren sonstige Knoten einstellige arithmetische Funktionen speichern. Als Beispiel betrachten wir den Baum in Abbildung 2.

Der Beispielbaum hat fünf Blätter mit den Zeichen `'g'`, `'u'`, `'l'`, `'f'`, `'i'` und vier weitere Knoten, die eine Funktion mit Parameter `x` enthalten.
Schreiben Sie zu jeder der im Folgenden zu implementierenden Funktionen auch eine Typdeklaration.

[2] siehe auch Wikipedia: `http://de.wikipedia.org/wiki/Tiefensuche`

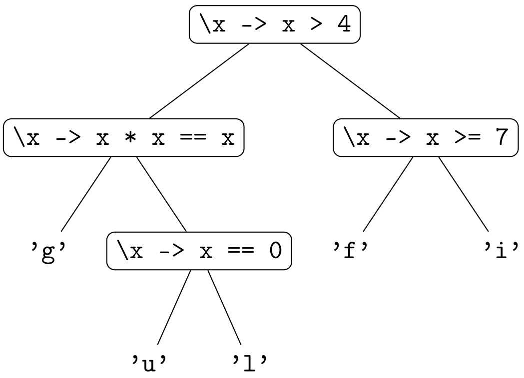


Abbildung 2: Ein beispielhafter Binärbaum.

a) Implementieren Sie in Haskell einen parametrisierten Datentyp `BinTree a b`, mit dem Binärbäume mit Werten vom Datentyp `b` in den Blättern und Werten vom Datentyp `a` in allen anderen Knoten dargestellt werden können. Dabei soll sichergestellt werden, dass jeder innere Knoten genau zwei Nachfolger hat.

Hinweise:

- Ergänzen Sie `deriving Show` am Ende der Datentyp-Deklaration, damit `GHCi` die Bäume auf der Konsole anzeigen kann: `data ... deriving Show`. Importieren Sie dazu am Anfang Ihrer Datei per `import Text.Show.Functions` das zugehörige Package.

b) Definieren Sie den Beispielbaum aus Abbildung 2 als `example`:

```
example :: BinTree (Int -> Bool) Char
example = ...
```

c) Schreiben Sie die Funktion `countInnerNodes`, die einen Binärbaum vom Typ `BinTree a b` übergeben bekommt und die Anzahl der Knoten, die keine Blätter sind, als `Int` zurückgibt.

Für den Beispielbaum (angenommen dieser ist als `example` verfügbar) soll für den Aufruf `countInnerNodes example` also 4 zurückgegeben werden.

d) Schreiben Sie die Funktion `decodeInt`. Diese bekommt als erstes Argument einen Binärbaum vom Typ `BinTree (Int -> Bool) b` und als zweites Argument einen Wert vom Typ `Int`. Der Rückgabewert dieser Funktion ist vom Typ `b`. Für einen Baum `bt` und eine Zahl `x` gibt `decodeInt bt x` das Zeichen zurück, an das man gelangt, wenn man ausgehend von der Wurzel in jedem Knoten die Funktion des jeweiligen Knotens auf die Zahl `x` anwendet, wobei das linke Kind eines Knotens als Nachfolger gewählt wird, falls die Funktion zu `False` auswertet, und das rechte Kind gewählt wird, falls sie zu `True` auswertet. Wird `decodeInt` auf einem Blatt aufgerufen, wird dessen Wert zurückgegeben.

Für den Beispielbaum `example` soll der Aufruf `decodeInt example 0` also `'l'` zurückgeben.

e) Schreiben Sie die Funktion `decode`. Diese bekommt als erstes Argument einen Binärbaum vom Typ `BinTree (Int -> Bool) b` und als zweites Argument eine Liste vom Typ `[Int]` übergeben. Für einen Baum `bt` und eine Liste `xs` sucht `decode bt xs` zu jeder Zahl `x` aus der Liste `xs` den Wert `decodeInt bt x` und fügt die so erhaltenen Werte in einer Liste zusammen. Verwenden Sie hierzu die für Listen vordefinierte Funktion `map`.

Für den Beispielbaum `example` soll der Aufruf `decode example [0,1,5,-4,7]` also den `String` `"lufgi"` zurückgeben.

f) Schreiben Sie, analog zu `map`, die Funktion `mapTree`. Diese bekommt als erstes Argument eine Funktion `f` vom Typ `b -> c` und als zweites Argument einen Binärbaum `bt` vom Typ `BinTree a b`. Der Rückgabewert von `mapTree f bt` ist der Binärbaum vom Typ `BinTree a c`, der aus `bt` entsteht, indem der Wert jedes Blattes durch `f` auf einen neuen Wert vom Typ `c` abgebildet wird.

Für den Beispielbaum `example` soll der Aufruf `mapTree (\x -> 'e') example` also den Binärbaum zurückgeben, der sich von dem obigen nur darin unterscheidet, dass jedes Blatt das Zeichen `'e'` enthält.

Tutoraufgabe 3 (Typen):

Bestimmen Sie zu den folgenden Haskell-Funktionen `f`, `g`, `h` und `i` den jeweils allgemeinsten Typ. Geben Sie den Typ an und begründen Sie Ihre Antwort. Gehen Sie hierbei davon aus, dass alle Zahlen den Typ `Int` haben und die Funktion `+` den Typ `Int -> Int -> Int` hat.

i)
```
f []     x y = y
f [z:zs] x y = f [] (z:x) y
```

ii)
```
g x 1 = 1
g x y = (\x -> (g x 0)) y
```

iii)
```
h (x:xs) y z = if x then h xs x (y:z) else h xs y z
```

iv)
```
data T a b = C0 | C1 a | C2 b | C3 (T a b) (T a b)

i (C3 (C1 x)      (C2 y)) = C2 0
i (C3 (C1 (x:xs)) (C2 y)) = i (C3 (C1 [y]) (C2 x))
```

Hinweise:

- Versuchen Sie diese Aufgabe ohne Einsatz eines Rechners zu lösen. Bedenken Sie, dass Sie in einer Prüfung ebenfalls keinen Rechner zur Verfügung haben.

Aufgabe 4 (Typen): $(1 + 1 + 2 + 3 + 2 = 9$ Punkte)

Bestimmen Sie zu den folgenden Haskell-Funktionen `f`, `g`, `h`, `i` und `j` den jeweils allgemeinsten Typ. Geben Sie den Typ an und begründen Sie Ihre Antwort. Gehen Sie hierbei davon aus, dass alle Zahlen den Typ `Int` haben und die Funktion `+` die Typen `Int -> Int -> Int` hat, die Funktion `head` die Typen `[a] -> a` und die Funktion `==` die Typen `a -> a -> Bool`.

i)
```
f xs     y []     = []
f (x:xs) y (z:zs) = if z then ((x + y) : f xs y zs) else (x : f xs y zs)
```

ii)
```
g x y = g (head y) y
g x y = (\x -> x) y
```

iii)
```
h w x []     z = if w == [] then head z else h w x [] z
h w x (y:ys) z = if w == [x] then y else (x + 1 > x)
```

iv)
```
data X a b = A a | B Int | F (a -> b -> Bool)

i (F f) x y = f x y
i (A x) y z = if (x == z) then h y else h 0
  where
    h n = i (B n) y x
```

v)
```
j x y | x > y = []
      | y == x = [x] ++ [y]
      | otherwise = y : (x <= y) : j x x
```

Hinweise:

- Versuchen Sie diese Aufgabe ohne Einsatz eines Rechners zu lösen. Bedenken Sie, dass Sie in einer Prüfung ebenfalls keinen Rechner zur Verfügung haben.

Tutoraufgabe 5 (Funktionen höherer Ordnung):

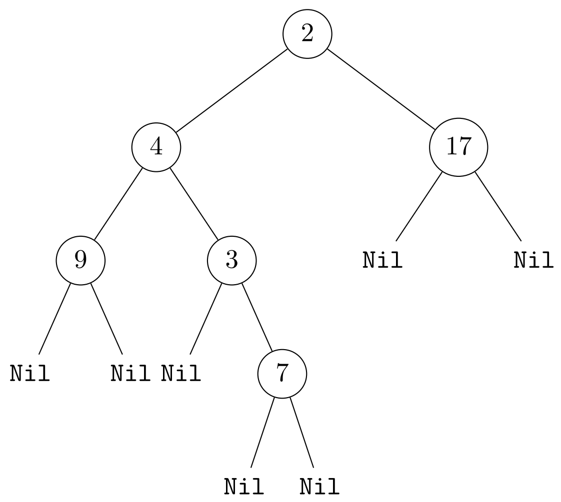
Wir betrachten Operationen auf dem Typ `Tree`, der wie folgt definiert ist:

```
data Tree = Nil | Node Int Tree Tree deriving Show
```

Ein Beispielobjekt vom Typ `Tree` ist:

```
testTree = Node 2
 (Node 4 (Node 9 Nil Nil) (Node 3 Nil (Node 7 Nil Nil)))
 (Node 17 Nil Nil)
```

Man kann den Baum auch graphisch darstellen:



Wir wollen nun eine Reihe von Funktionen betrachten, die auf diesen Bäumen arbeiten:

```
decTree :: Tree -> Tree
decTree Nil = Nil
decTree (Node v l r) = Node (v - 1) (decTree l) (decTree r)
```

Die Funktion `decTree` gibt den Baum zurück, der aus dem Eingabebaum daraus entsteht, dass in jedem Knoten der Wert um eins verringert wird.

```
sumTree :: Tree -> Int
sumTree Nil = 0
sumTree (Node v l r) = v + (sumTree l) + (sumTree r)
```

Die Funktion `sumTree` gibt die Summe aller Werte aus den Knoten des Eingabebaums zurück.

```
flattenTree :: Tree -> [Int]
flattenTree Nil = []
flattenTree (Node v l r) = v : (flattenTree l) ++ (flattenTree r)
```

Die Funktion `flattenTree` gibt die Liste der Werte aus den Knoten zuück, die entsteht, wenn man alle Knoten per Tiefensuche durchläuft.
Wir sehen, dass diese Funktionen alle in der gleichen Weise konstruiert werden: Was die Funktion mit einem Baum macht, wird anhand des verwendeten Datenkonstruktors entschieden. Der nullstellige Konstruktor `Nil` wird einfach in einen Standard-Wert übersetzt. Für den dreistelligen Konstruktor `Node` wird die jeweilige Funktion rekursiv auf den Kindern aufgerufen und die Ergebnisse werden dann weiterverwendet, z.B. um ein neues `Tree`-Objekt zu konstruieren oder ein akkumuliertes Gesamtergebnis zu berechnen. Intuitiv kann man sich vorstellen, dass jeder Konstruktor durch eine Funktion der gleichen Stelligkeit ersetzt wird. Klarer wird dies, wenn man die folgenden alternativen Definitionen der Funktionen von oben betrachtet:

```
decTree' Nil = Nil
decTree' (Node v l r) = decN v (decTree' l) (decTree' r)
decN = \v l r -> Node (v - 1) l r

sumTree' Nil = 0
sumTree' (Node v l r) = sumN v (sumTree' l) (sumTree' r)
sumN = \v l r -> v + l + r

flattenTree' Nil = []
flattenTree' (Node v l r) = flattenN v (flattenTree' l) (flattenTree' r)
flattenN = \v l r -> v : l ++ r
```

Die definierenden Gleichungen für den Fall, dass der betrachtete Baum durch den Konstruktor `Node` erzeugt wird, kann man in allen diesen Definitionen so lesen, dass die eigentliche Funktion rekursiv auf die Kinder angewandt wird und der Konstruktor `Node` durch die jeweils passende Hilfsfunktion (`decN`, `sumN`, `flattenN`) ersetzt wird. Der Konstruktor `Nil` wird analog durch eine nullstellige Funktion (also eine Konstante) ersetzt. Als Beispiel kann der Ausdruck `decTree' testTree` dienen, der dem folgenden Ausdruck entspricht:

```
decN 2
 (decN 4 (decN 9 Nil Nil) (decN 3 Nil (decN 7 Nil Nil)))
 (decN 17 Nil Nil)
```

Im Baum `testTree` sind also alle Vorkommen von `Node` durch `decN` und alle Vorkommen von `Nil` durch `Nil` ersetzt worden.
Analog ist `sumTree' testTree` äquivalent zu

```
sumN 2
 (sumN 4 (sumN 9 0 0) (sumN 3 0 (sumN 7 0 0)))
 (sumN 17 0 0)
```

Im Baum `testTree` sind also alle Vorkommen von `Node` durch `sumN` und alle Vorkommen von `Nil` durch `0` ersetzt worden.
Die *Form* der Funktionsanwendung bleibt also gleich, nur die Ersetzung der Konstruktoren durch Hilfsfunktionen muss gewählt werden. Dieses allgemeine Muster wird durch die Funktion `foldTree` beschrieben:

```
foldTree :: (Int -> a -> a -> a) -> a -> Tree -> a
foldTree f c Nil = c
foldTree f c (Node v l r) = f v (foldTree f c l) (foldTree f c r)
```

Bei der Anwendung ersetzt `foldTree` alle Vorkommen des Konstruktors `Node` also durch die Funktion `f` und alle Vorkommen des Konstruktors `Nil` durch die Konstante `c`. Unsere Funktionen von oben können dann vereinfacht wie folgt dargestellt werden:

```
decTree'' t = foldTree decN Nil t
sumTree'' t = foldTree sumN 0 t
flattenTree'' t = foldTree flattenN [] t
```

a) Implementieren Sie eine Funktion `prodTree`, die das Produkt der Einträge eines Baumes bildet. Es soll also `prodTree testTree = 2 * 4 * 9 * 3 * 7 * 17 = 25704` gelten. Verwenden Sie dazu die Funktion `foldTree`.

Hinweise:

- Das leere Produkt (= Produkt mit 0 Faktoren) ist 1.

b) Implementieren Sie eine Funktion `incTree`, die einen Baum zurückgibt, in dem der Wert jeden Knotens um 1 inkrementiert, d.h. erhöht, wurde. Verwenden Sie dazu die Funktion `foldTree`.

Aufgabe 6 (Funktionen höherer Ordnung): (1.5 + 1 + 1.5 + 1 + 2 + 2 + 2 = 11 Punkte)

Wir betrachten Operationen auf dem parametrisierten Typ `List a`, der (mit zwei Testwerten) wie folgt definiert ist:

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

Zwei Beispielobjekte vom Typ `List Int` sind:

```
list :: List Int
list = Cons (-3) (Cons 14 (Cons (-6) (Cons 7 (Cons 1 Nil))))

blist :: List Int
blist = Cons 1 (Cons 1 (Cons 0 (Cons 0 Nil)))
```

Die Liste `list` entspricht also $[-3, 14, -6, 7, 1]$.
Verwenden Sie keine vordefinierten Funktionen, wenn sie nicht explizit erwähnt sind.

a) Schreiben Sie eine Funktion `filterList :: (a -> Bool) -> List a -> List a`, die sich auf unseren selbstdefinierten Listen wie `filter` auf den vordefinierten Listen verhält. Es soll also die als erster Parameter übergebene Funktion auf jedes Element angewandt werden, um zu entscheiden, ob dieses auch im Ergebnis auftritt. Der Ausdruck `filterList (\x -> x > 10 || x < -5) list` soll dann also zu `(Cons 14 (Cons -6 Nil))` auswerten.

b) Schreiben Sie eine Funktion `divisibleBy :: Int -> List Int -> List Int`, wobei `divisibleBy x xs` die Teilliste der Werte der Liste `xs` zurückgibt, die durch `x` teilbar sind. Für `divisibleBy 7 list` soll also `Cons 14 (Cons 7 Nil)` zurückgegeben werden. Verwenden Sie dafür `filterList`.

Hinweise:

Sie dürfen die vordefinierte Funktion `rem x y` verwenden, die den Rest der Division `x / y` zurückgibt.

c) Schreiben Sie eine Funktion `foldList :: (a -> b -> b) -> b -> List a -> b`, die wie `foldTree` aus der vorhergegangen Tutoraufgabe die Datenkonstruktoren durch die übergebenen Argumente ersetzt. Der Ausdruck `foldList f c (Cons` x_1 `(Cons` x_2 `... (Cons` x_n `Nil) ... )` soll dann also äquivalent zu `(f` x_1 `(f` x_2 `... (f` x_n `c) ... )` sein.

Beispielsweise soll für `plus x y = x + y` der Ausdruck `foldList plus 0 list` zu `-3 + 14 + (-6) + 7 + 1 = 13` ausgewertet werden.

d) Schreiben Sie eine Funktion `listMaximum :: List Int -> Int`, die für eine nicht-leere Liste das Maximum berechnet. Verwenden Sie hierzu `foldList`. Auf der leeren Liste darf sich Ihre Funktion beliebig verhalten.

Hinweise:

Sie dürfen die vordefinierte Konstante `minBound :: Int` benutzen, die den kleinsten möglichen Wert vom Typ `Int` liefert.

e) Schreiben Sie eine Funktion `mapList :: (a -> b) -> List a -> List b`, die sich auf unseren selbstdefinierten Listen wie `map` auf den vordefinierten Listen verhält. Es soll also die als erster Parameter übergebene Funktion auf jedes Element angewandt werden, um die Ausgabeliste zu erzeugen. Der Ausdruck `mapList (\x -> 2*x) list` soll dann also zu `Cons (-6) (Cons 28 (Cons (-12) (Cons 14 (Cons 2 Nil))))` auswerten.

Verwenden Sie hierzu neben der Typdeklaration nur eine weitere Zeile, in der Sie `mapList` mittels `foldList` definieren.

f) Schreiben Sie eine Funktion `zipLists :: (a -> b -> c) -> List a -> List b -> List c` die aus zwei Listen eine neue erstellt. Das Element an Position i der resultierenden Liste ist das Ergebnis der Anwendung der übergebenen Funktion auf die beiden Elemente an Position i der Eingabelisten. Falls eine Liste mehr Elemente enthält als die andere, werden die überzähligen Elemente ignoriert. Die Länge der Ausgabeliste ist also gleich der Länge der kürzeren Eingabeliste.

Beispielsweise soll die Anwendung von `zipLists (>) list blist` also `Cons False (Cons True (Cons False (Cons True Nil)))` ergeben.

g) Schreiben Sie eine Funktion `skalarprodukt :: List Int -> List Int -> Int`. Diese interpretiert die übergebenen Listen als Vektoren und berechnet das Skalarprodukt. Falls eine Eingabeliste länger ist als die andere, werden die überzähligen Elemente ignoriert. Verwenden Sie hierzu `zipLists` und `foldList`.

Für den Aufruf `skalarprodukt blist list` wird also das Ergebnis $1 \cdot (-3) + 1 \cdot 14 + 0 \cdot (-6) + 0 \cdot 7 = 11$ zurückgegeben.

Tutoraufgabe 7 (Unendliche Datenstrukturen):

a) Implementieren Sie in Haskell die Funktion `odds` vom Typ `[Int]`, welche die unendliche Liste aller ungeraden natürlichen Zahlen berechnet.

b) Aus der Vorlesung ist Ihnen die Funktion `primes` bekannt, welche die Liste aller Primzahlen berechnet. Nutzen Sie diese Funktion nun, um die Funktion `primeFactors` vom Typ `Int -> [Int]` in Haskell zu implementieren. Diese Funktion soll zu einer natürlichen Zahl ihre Primfaktorzerlegung als Liste berechnen (auf Zahlen kleiner als 1 darf sich Ihre Funktion beliebig verhalten). Z. B. soll der Aufruf `primeFactors 420` die Liste `[2,2,3,5,7]` berechnen.

Hinweise:

- Die vordefinierten Funktionen `div` und `rem` vom Typ `Int -> Int -> Int`, welche die abgerundete Ganzzahldivision bzw. den Rest der Division berechnen, könnten hilfreich sein.

Aufgabe 8 (Unendliche Datenstrukturen): (2 + 2 + 3 + 2 = 9 Punkte)

Verwenden Sie keine vordefinierten Funktionen, wenn sie nicht explizit erwähnt sind.

a) Geben Sie einen Haskell-Ausdruck an, der zu einer unendlichen Liste aller Palindrome ausgewertet wird. Ein Palindrom ist ein `String`, der vorwärts und rückwärts gelesen gleich ist. Somit ist `''anna''` ein Beispiel für ein Palindrom. Wir betrachten in dieser Aufgabe ausschließlich `String`s, die aus den Zeichen `'a'` bis `'z'` bestehen. Die berechnete Liste soll bezüglich der Länge ihrer Elemente aufsteigend sortiert sein.

Sie dürfen die folgende Hilfsfunktion `strings` benutzen. Diese berechnet alle `String`s der Länge `n`, wobei `n` das erste Argument der Funktion ist.

```
strings :: Int -> [String]
strings 0 = [""]
strings n = concat (map (\x -> map (\tail -> x:tail) tails) ['a'..'z'])
  where tails = strings (n-1)
```

Hinweise:

- Die Funktion `reverse :: [a] -> [a]` dreht eine Liste um.

b) Geben Sie einen Haskell-Ausdruck an, der zu der aufsteigend sortierten Liste aller *perfekten Zahlen* ausgewertet wird. Eine Zahl $x \geq 2$ ist genau dann perfekt, wenn die Summe ihrer echten Teiler gleich x ist. Betrachten Sie als Beispiel die Zahl 6: Ihre echten Teiler sind $1, 2$ und 3 und es gilt $1 + 2 + 3 = 6$, also ist 6 eine perfekte Zahl.

Sie dürfen die folgende Hilfsfunktion `divisors` benutzen. Diese berechnet alle echten Teiler der als Argument übergebenen Zahl.

```
divisors :: Int -> [Int]
divisors x = filter (\y -> rem x y == 0) [1..div x 2]
```

Hinweise:

- Die Funktion `sum :: [Int] -> Int` berechnet die Summe aller Elemente einer Liste.
- Für jede Zahl x erzeugt `[x..]` die unendliche Liste `[x,x+1,x+2,...]`.

c) Geben Sie einen Haskell-Ausdruck an, der zu der aufsteigend sortierten Liste aller *semiperfekten Zahlen* ausgewertet wird. Eine Zahl $x \geq 2$ ist genau dann semiperfekt, wenn die Summe *aller oder einiger* ihrer echten Teiler gleich x ist. Betrachten Sie als Beispiel die Zahl 12: Ihre echten Teiler sind $1, 2, 3, 4$ und 6 und es gilt $2 + 4 + 6 = 12$, also ist 12 eine semiperfekte Zahl.

Hinweise:

- Die Funktion `any :: (a -> Bool) -> [a] -> Bool` testet, ob ein Element einer Liste das als erstes Argument übergebene Prädikat erfüllt.
- Die Funktion `subsequences :: [a] -> [[a]]` berechnet alle Teillisten der als Argument übergebenen Liste. Es gilt zum Beispiel:

  `subsequences [1,2,3] = [[],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]]`

  Damit Sie die Funktion `subsequences` nutzen können, muss die erste Zeile der Datei mit Ihrer Lösung "`import Data.List`" lauten.

d) Geben Sie einen Haskell-Ausdruck an, der zu der aufsteigend sortierten Liste aller Fibonacci-Zahlen evaluiert wird. Die Fibonacci-Zahlen haben Sie bereits auf Blatt 10 kennengelernt. Greifen Sie dafür *nicht* auf einen Ausdruck zurück, der die n-te Fibonacci-Zahl berechnet.

Hinweise:

- Überlegen Sie, wie Sie die Effizienzüberlegungen von Blatt 10 auch in dieser Aufgabe umsetzen können. Für eine ineffiziente Lösung, bei der Elemente der Liste mehrfach evaluiert werden, werden keine Punkte vergeben.
- Es bietet sich an, die Hilfsfunktion `fibInit :: Int -> Int -> [Int]` zu implementieren, die die unendliche Liste der Fibonacci-Zahlen mit beliebigen Initialwerten berechnet, vgl. hierzu Aufgabe 6 auf Blatt 10.