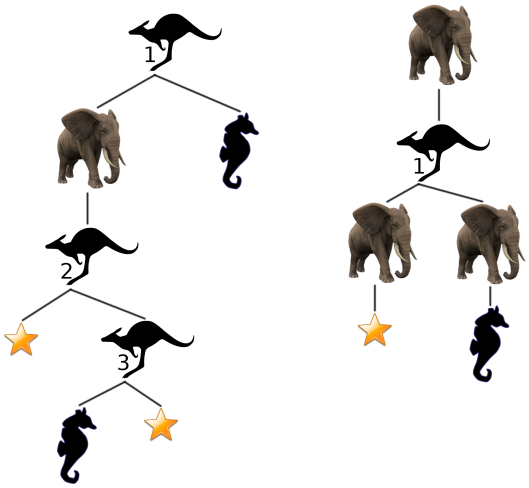
Prof. Dr. J. Giesl
M. Brockschmidt, F. Emmes, C. Otto, T. Ströder

Tutoraufgabe 1 (Datenstrukturen in Haskell):

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 der folgenden Grafik finden Sie zwei beispielhafte Mobiles[1].



a) Entwerfen Sie einen parametrischen 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 oben dargestellten Mobiles als Ausdruck dieses Datentyps in Haskell. Nehmen Sie hierfür an, dass die gezeigten Beutelinhalte vom Typ `Int` sind.

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.

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

b) Schreiben Sie 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.

[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`

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.

Lösung:

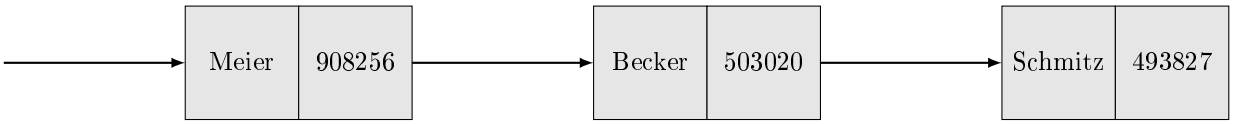
```
data Mobile a = Stern | Seepferdchen | Elefant (Mobile a)
              | Kaenguru a (Mobile a) (Mobile a) deriving Show

mobileLinks :: Mobile Int
mobileLinks = Kaenguru 1
                  (Elefant (Kaenguru 2
                        Stern
                        (Kaenguru 3
                            Seepferdchen
                            Stern
                        )
                  ))
                  Seepferdchen

mobileRechts :: Mobile Int
mobileRechts = Elefant (Kaenguru 1 (Elefant Stern) (Elefant Seepferdchen))

count :: Mobile a -> Int
count Stern                 = 1
count Seepferdchen          = 1
count (Elefant         a  ) = 1 + count a
count (Kaenguru inhalt b c) = 1 + count b + count c

liste :: Mobile a -> [a]
liste Stern                 = []
liste Seepferdchen          = []
liste (Elefant a)           = liste a
liste (Kaenguru inhalt a b) = inhalt : liste a ++ liste b


greife :: Mobile a -> Int -> Mobile a
greife x                     1 = x
greife (Elefant a)           x = greife a (x-1)
```

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

```
greife (Kaenguru inhalt a b) x | (x-1) <= count a = greife a (x-1)
                               | otherwise = greife b (x-1 - (count a))
```

Aufgabe 2 (Datenstrukturen in Haskell): (1 + 1 + 1 + 2 + 2 = 7 Punkte)

In dieser Aufgabe betrachten wir eine einfache assoziative Datenstruktur in Form einer Liste.
Bei einer assoziativen Datenstruktur ist es möglich, einzelnen Schlüsselwerten jeweils einen Wert zuzuordnen. Beispielsweise kann jedes Telefonbuch als assoziative Datenstruktur verstanden werden, bei der zu jedem Namen (dem Schlüssel) im Buch die dazugehörige Telefonnummer (der Wert) angegeben ist.
Wir speichern in dieser Aufgabe die Paare aus Schlüssel und Wert in einer Liste. In der Beispielgrafik erkennt man eine solche Liste, die zum Namen `"Meier"` die Nummer `908256`, zum Namen `"Becker"` die Nummer `503020` und zum Namen `"Schmitz"` die Nummer `493827` gespeichert hat.



Verwenden Sie in dieser Aufgabe keine vordefinierten Listen oder Tupel!

a) Entwerfen Sie einen parametrisierten Datentyp `AList a`, mit dem assoziative Listen über Schlüssel vom Datentyp `String` und Werte vom Datentyp `a` dargestellt werden können.

Hinweise:

- Für die nachfolgenden Aufgaben ist es sehr hilfreich, die obige Beispielliste unter dem Namen `bsp` zur Verfügung zu haben:

```
bsp :: AList Int
bsp = ...
```

- Ergänzen Sie `deriving Show` am Ende der Datenstruktur, damit `GHCi` die Listen auf der Konsole anzeigen kann: `data ... deriving Show`

b) Schreiben Sie die Funktion `size`, die eine assoziative Liste vom Typ `AList a` übergeben bekommt und als Rückgabe die Anzahl der Einträge in dieser Liste (also die Länge der Liste) als `Int` zurückgibt.

Für die Beispielliste (angenommen diese ist als `bsp` verfügbar) soll für den Aufruf `size bsp` also `3` zurückgegeben werden.

c) Schreiben Sie die Funktion `contained`. Diese bekommt als erstes Argument eine assoziative Liste vom Typ `AList a` und als zweites Argument einen `String`. Der Rückgabewert dieser Methode ist vom Typ `Bool`. Die Methode gibt `True` zurück, wenn in der Liste ein Element enthalten ist, das als Schlüssel den `String` des zweiten Arguments hat. Ansonsten wird `False` zurückgegeben.

Für die Beispielliste `bsp` soll der Aufruf `contained bsp "Becker"` also `True` zurückgeben, `contained bsp "Meyer"` gibt `False`.

d) Schreiben Sie die Funktion `put`. Diese bekommt als erstes Argument eine assoziative Liste vom Typ `AList a`, als zweites Argument einen `String` und als drittes Argument einen Wert vom Typ `a`. Der Rückgabewert der Funktion ist die wie folgt modifizierte assoziative Liste vom Typ `AList a`: Falls für den im zweiten Argument übergebenen Schlüssel bereits ein Element in der Liste existiert, wird dessen Wert mit dem im dritten Argument übergebenen Wert überschrieben. Falls kein solches Element existiert, wird dieses hinzugefügt.

Gehen Sie hierbei davon aus, dass zu jedem Schlüssel in der Liste auch nur genau ein Listenelement existiert.

Für die Beispielliste `bsp` soll der Aufruf `put bsp "Becker" 1` also die assoziative Liste zurückgeben, bei der statt `503020` nun `1` für den Schlüssel `"Becker"` gespeichert ist.

e) Schreiben Sie, analog zu `map`, die Funktion `mapAList`. Diese bekommt als erstes Argument eine Funktion vom Typ `a -> b` und als zweites Argument eine assoziative Liste vom Typ `AList a`. Der Rückgabewert der Funktion ist die wie folgt definierte assoziative Liste vom Typ `AList b`: Für jedes Paar `(x, y)` in der Eingabeliste (des zweiten Arguments) enthält die Ergebnisliste ein Paar `(x, z)`. Wenn `f` die Funktion des ersten Arguments ist, ist `z` das Ergebnis von `f y`. Die Ergebnisliste enthält keine weiteren Paare.

Für die Beispielliste `bsp` soll der Aufruf `mapAList (* (-1)) bsp` also die assoziative Liste zurückgeben, bei der Elemente für die Paare `(Meier, -908256)`, `(Becker, -503020)` und `(Schmitz, -493827)` vorhanden sind.

Lösung:

```
data AList value = Nil | Entry String value (AList value) deriving Show

bsp :: AList Int
bsp = Entry "Meier" 908256 (Entry "Becker" 503020 (Entry "Schmitz" 493827 Nil))

size :: AList b -> Int
size Nil            = 0
size (Entry _ _ xs) = 1 + size xs

contained :: AList a -> String -> Bool
contained Nil             _ = False
contained (Entry a _ xs) y = if a == y then True else contained xs y

put :: AList a -> String -> a -> AList a
put Nil            x y = Entry x y Nil
put (Entry a b xs) x y = if a == x then Entry x y xs else Entry a b (put xs x y)

mapAList :: (a -> b) -> AList a -> AList b
mapAList _ Nil            = Nil
mapAList f (Entry a b xs) = Entry a (f b) (mapAList f xs)
```

Tutoraufgabe 3 (Typen in Haskell):

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 [])     (C2 y)) = C1 0
i (C3 (C2 [])     (C1 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.

Lösung:

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

Wir beginnen mit einer generellen Definition `f :: a -> b -> c -> d`.

- Aus der ersten Definition ergibt sich, dass der Typ des dritten Arguments dem Typ des Rückgabewertes von `f` entspricht, d.h. `c = d`.
- Aus der zweiten Definition ergibt sich, dass der Typ des ersten Arguments von `f` eine Liste von Listen ist. Hat das daraus entnommene `x` Typ `e`, hat also das erste Argument den Typ `[[e]]`.
- Aus der zweiten Definition ergibt sich weiterhin, dass `z` in `x` eingefügt werden kann. Daher hat `x` (und damit das zweite Argument von `f`) den Typ `[e]`.

Insgesamt ergibt sich also der Typ `f :: [[e]] -> [e] -> c -> c`.

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

Wir beginnen mit einer generellen Definition `g :: a -> b -> c`.

- Aus der ersten Definition ergibt sich, dass der Typ des zweiten Arguments `Int` sein muss, da es auf `1` gematcht wird. Es gilt also `b = Int`.
- Aus der ersten Definition ergibt sich, dass der Typ des Rückgabewertes `Int` ist. Es gilt also `c = Int`.
- Aus der zweiten Definition ergibt sich durch `(\x -> (g x 0)) y = g y 0`, dass der Typ des ersten Arguments den gleichen Typ wie das zweite Argument haben muss, also `a = b`.

Insgesamt ergibt sich also der Typ `g :: Int -> Int -> Int`.

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

Wir beginnen mit einer generellen Definition `h :: a -> b -> c -> d`.

- Das erste Argument ist eine Liste, aus der `x` entnommen wird. Hat `x` den Typ `e`, gilt also `a = [e]`.
- Der Wert `x` wird als Bedingung in einem `if ... then ... else` verwendet, also ist `e = Bool`.
- Wir verwenden `x` auch als zweites Argument für `h`. Also gilt `b = e = Bool`.
- Wir fügen `y` in die Liste `z` ein. Es gilt also `d = [Bool]`.

Insgesamt ergibt sich also der Typ `h :: [Bool] -> Bool -> [Bool] -> a`.

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

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

Wir beginnen mit einer generellen Definition `i :: a -> b`.

- Aus der ersten Definition ergibt sich, dass der Rückgabewert `C1 0` ist, also den Typ `T Int e` hat.
- Aus der zweiten Definition ergibt sich, dass der Rückgabewert `C2 0` ist, also den Typ `T Int Int` hat.
- Wir matchen in allen Definitionen auf den Konstruktor `C3`. Das erste Argument ist also vom Typ `T c d`.

- Aus der Definition von `T c d` wissen wir, dass der Typ des ersten Arguments von `C3` wieder den Typ `T c d` hat. In der dritten Definition von `i` ist dies der Wert `C1 (x:xs)`, d.h. wenn `x` den Typ `e` hat, hat `(x:xs)` den Typ `[e]` und `C1 (x:xs)` daher den Typ `T [a] d`.
- In der dritten Definition von `i` verwenden wir `y` im ersten Argument von `C3 (C1 y) (C2 [x])`, welches als Argument für `i` verwendet wird. Da dieses Argument den Typ `T [a] d` hat, muss auch `C3 (C1 y) (C2 [x])` den Typen `T [a] d` haben und daher auch `(C1 y)`. Deswegen hat auch auch `y` den Typen `[a]`.

Insgesamt ergibt sich also der Typ `i :: T [a] [a] -> T Int Int`.

Aufgabe 4 (Typen in Haskell): (1 + 1 + 2 + 3 = 7 Punkte)

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 Funktionen `+`, `head` und `==` die Typen `Int -> Int -> Int`, `[a] -> a` und `a -> a -> Bool` haben.

i)
```
f (x : xs) y z = x + y
f xs       y z = if xs == z then y else head xs + y
```

ii)
```
g [] = g [1]
g x  = (\x -> x) x
```

iii)
```
h w x [] z = if z == [x] then w else h w x [] z
h w x y  z = if x then head y else (x, x)
```

iv)
```
data N a b = A a | F (a -> b) | I Int

i (F f) x = f x
i (A y) x = i k x
  where
    k = F (\x -> I y)
```

Hinweise:

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

Lösung:

i)
```
f (x:xs) y z = x + y
f xs     y z = if xs == z then y else head xs + y
```

Da `f` drei Parameter bekommt, hat hat der Typ die Form `f :: a -> b -> c -> d`. Aus `(x : xs)` folgt, dass `a` eine Liste sein muss. Daher ersetzen wir `a` durch `[f]`, wobei `x` den Typ `f` hat. Aus `x + y` können wir folgern, dass `x` und `y` jeweils den Typ `Int` haben und der Ergebnistyp `d` ebenfalls ein `Int` ist. Somit ersetzen wir `f`, `b` und `d` jeweils durch `Int` und erhalten den Typ `f :: [Int] -> Int -> c -> Int`. Aus `xs == z` folgt, dass `z` den gleichen Typ wie `xs` (das erste Argument der Funktion) hat. Somit ergibt sich insgesamt `f :: [Int] -> Int -> [Int] -> Int`.

ii)
```
g [] = g [1]
g x  = (\x -> x) x
```

Da `g` ein Argument hat, nehmen wir den Typ `g :: a -> b` an. Aus der ersten Gleichung folgt, dass `a` eine Liste (mit unbekanntem Element-Typ) und das Ergebnis eine `Int`-Liste ist. Somit ersetzen wir `a` durch `[c]` und `b` durch `[Int]`. Der Typ des Lambda-Ausdrucks `\x -> x` ist `d -> d`. Da dieser Lambda-Ausdruck auf den Parameter der Funktion angewendet wird, folgt aus der zweiten Gleichung, dass der Parametertyp der Funktion ihrem Ergebnistyp entspricht: `g :: [Int] -> [Int]`.

iii)
```
h w x [] z = if z == [x] then w else h w x [] z
h w x y  z = if x then head y else (x, x)
```

Die Funktion `h` hat 4 Parameter. Wir gehen also erstmal vom Typ `h :: a -> b -> c -> d -> f` aus. Aus der ersten Gleichung erfahren wir, dass der dritte Parameter (`c`) einen Listentyp hat (`[g]`). Der Vergleich `z == [x]` ergibt, dass der vierte Parameter den Typ des zweiten Parameters als Listentyp hat (`[b]`), sowie dass der erste Parameter dem Ergebnistypen entspricht. Daraus resultiert der Typ `h :: a -> b -> [g] -> [b] -> a`.

Aus der zweiten Gleichung folgt, dass der zweite Parameter (`x`) ein `Bool` ist. Der Ergebnistyp muss zum einen den Typ der Listenelemente von `y` haben (`then head y ...`), sowie ein Tupel von zwei `Bool`-Werten sein (`(else (x, x))`).

Daraus ergibt sich der Typ `h :: (Bool, Bool) -> Bool -> [(Bool, Bool)] -> [Bool] -> (Bool, Bool)`.

iv)
```
data N a b = A a | F (a -> b) | I Int

i (F f) x = f x
i (A y) x = i h x
  where
    h = F (\x -> I y)
```

Die Funktion `i` hat zwei Parameter. Wir nehmen also `i :: t1 -> t2 -> t3` an. Aus der ersten Gleichung folgt, dass der erste Parameter (`t1`) vom Typ `N a b` ist und `i` den Typ `a -> b` hat. Somit folgt aus der rechten Seite `i x`, dass `x` den Typ `a` und der Ergebnistyp von `i b` ist. Somit erhalten wir `i :: N a b -> a -> b`.

In der zweiten Gleichung hat `A y` den Typ `N a b`, somit hat `y` den Typ `a`. Aus dem Ausdruck `I y` folgt zusätzlich, dass `y` den Typ `Int` hat. Somit ersetzen wir `a` durch `Int` und erhalten `i :: N Int b -> Int -> b`.

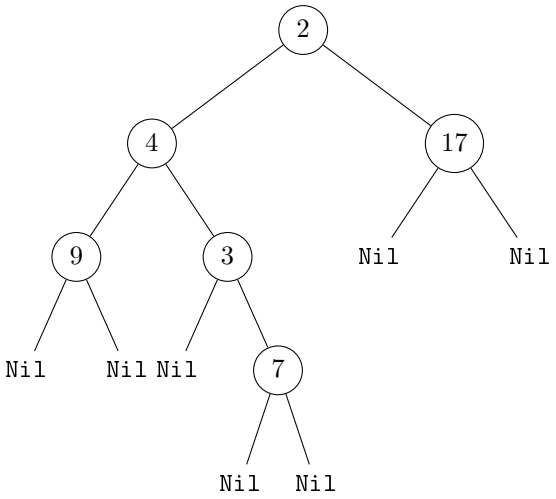
Der Ausdruck `\x -> I y` hat den Typ `c -> N d e` und somit folgt `h :: N c (N d e)`. Der Ausdruck `i h x` hat somit den Typ `N d e` und die gesamte Funktion `i :: N Int (N d e) -> Int -> N d e`.

Tutoraufgabe 5 (Higher Order):

Wir betrachten Operationen auf dem Typ `Tree` der (mit einem Testwert) wie folgt definiert ist:

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

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)

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

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

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 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`.

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

Lösung:

a)
```
prodTree = foldTree (\x y z -> x * y * z) 1
```

b)
```
incTree = foldTree (\x y z -> Node (x+1) y z) Nil
```

Aufgabe 6 (Higher Order): (1 + 0.5 + 2.5 + 1 + 1 + 2 + 2 = 10 Punkte)

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

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

testList, testList2 :: List Int
testList = Cons (-23) (Cons 42 (Cons 19 (Cons (-38) Nil)))
testList2 = Cons 1 (Cons 2 Nil)
```

Die Liste `testList` entspricht also $-23, 42, 19, -38$ und `testList2` der Liste $1, 2$.

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 auftaucht. Der Ausdruck `filterList (\x -> x > 30 || x < -30) testList` soll dann also zu `(Cons 42 (Cons -38 Nil))` auswerten.

b) Schreiben Sie eine Funktion `posList :: List Int -> List Int`, die die Teilliste der positiven Werte zurückgibt. Für `testList` soll also `Cons 42 (Cons 19 Nil)` zurückgegeben werden. Verwenden Sie dafür `filterList`.

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 `times x y = x * y` der Ausdruck `foldList times 1 testList` zu `-23 * 42 * 19 * -38 = 697452` ausgewertet werden.

d) Schreiben Sie eine Funktion `maxList :: List Int -> Int`, die das Maximum der Liste bestimmt. Für `testList` soll also `42` zurückgegeben werden. Verwenden Sie dafür `foldList` und die vordefinierte Funktion `max :: Int -> Int -> Int`, die das Maximum zweier Zahlen zurückgibt. Dabei ist das Maximum der leeren Liste undefiniert.

e) Schreiben Sie eine Funktion `appendList :: List a -> List a -> List a`, die zwei Listen aneinander hängt. Der Ausdruck `appendList testList testList2` soll also zu `Cons (-23) (Cons 42 (Cons 19 (Cons (-38) (Cons 1 (Cons 2 Nil)))))` auswerten. Verwenden Sie dafür `foldList`.

f) Schreiben Sie eine Funktion `reverse :: List a -> List a`, die eine Liste umkehrt. Für `testList` soll also `Cons (-38) (Cons 19 (Cons 42 (Cons (-23) Nil)))` berechnet werden. Verwenden Sie dafür `foldList` und `appendList`.

g) Implementieren Sie die Funktion `filterList` unter dem Namen `filterList'` erneut, verwenden Sie diesmal aber `foldList`, statt direkt Rekursion in der Definition von `filterList'` zu verwenden.

Lösung:

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

--testList, testList2 :: List Int
testList = Cons (-23) (Cons 42 (Cons 19 (Cons (-38) Nil)))
testList2 = Cons (-1) (Cons 2 (Cons 3 (Cons (-4) Nil)))

-- a)
filterList :: (a -> Bool) -> List a -> List a
filterList _ Nil = Nil
filterList f (Cons x xs) | f x  = Cons x rest
                         | True = rest
 where rest = filterList f xs

-- b)
posList :: List Int -> List Int
posList = filterList (\x -> x > 0)

-- c)
foldList :: (a -> b -> b) -> b -> List a -> b
foldList _ c Nil = c
foldList f c (Cons x xs) = f x (foldList f c xs)

-- d)
maxList :: List Int -> Int
maxList (Cons c xs) = foldList (\x y -> max x y) c xs

-- e)
```

```
appendList :: List a -> List a -> List a
appendList xs ys = foldList (\x ys -> Cons x ys) ys xs

-- f)
reverse :: List a -> List a
reverse = foldList (\x y -> appendList y (Cons x Nil)) Nil

-- g)
filterList' :: (a -> Bool) -> List a -> List a
filterList' f = foldList (\x xs -> if f x then Cons x xs else xs) Nil
```

Tutoraufgabe 7 (Unendliche Listen):

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

b) Implementieren Sie in Haskell die Funktion `squares` vom Typ `[Int]`, welche die Liste aller Quadratzahlen von natürlichen Zahlen berechnet (also die unendliche Liste `[1,4,9,16,25,...]`). Benutzen Sie dafür ausschließlich die Funktion `odds` aus der vorigen Teilaufgabe und die in den Hinweisen vorgegebene Funktion `sumList` (d. h. keine anderen vordefinierten oder Hilfsfunktionen und auch keine Deklarationen mit `where` oder `let`).

Hinweise:

- Es gilt $n^2 = \Sigma_{k=0}^{n-1} 2 \cdot k + 1$ für alle $n \in \mathbb{N}$.
- Die Funktion `sumList` berechnet zu einer (endlichen oder unendlichen) Liste ganzer Zahlen eine Liste gleicher Länge, wobei das i-te Element der berechneten Liste die Summe der Elemente bis zum i-ten Element aus der ursprünglichen Liste ist. So wertet z. B. der Ausdruck `sumList [1,2,3,4]` zum Ergebnis `[1,3,6,10]` aus. Nachfolgend ist der Haskell Code dieser Funktion angegeben:

```
sumList :: [Int] -> [Int]
sumList = sumListHelper 0
   where
   sumListHelper _ []     = []
   sumListHelper x (y:ys) = let z = x + y in z : sumListHelper z ys
```

c) 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 `quot` und `mod` vom Typ `Int -> Int -> Int`, welche die abgerundete Ganzzahldivision bzw. die modulo Operation (also den Rest bei der Ganzzahldivision) berechnen, könnten hilfreich sein.

Lösung:

a)
```
odds :: [Int]
odds = 1 : map (+ 2) odds
```

b)
```
squares :: [Int]
squares = sumList odds
```

c)
```
primeFactors :: Int -> [Int]
primeFactors = pHelper primes
   where
   pHelper (x:xs) y | x*x > y = [y]
                    | mod y x == 0 = x : pHelper (x:xs) (quot y x)
                    | otherwise = pHelper xs y
```

Aufgabe 8 (Unendliche Listen): (2 + 4 = 6 Punkte)

a) Implementieren Sie in Haskell die Funktion `fibs` vom Typ `[Int]`, welche die Liste aller Fibonacci-Zahlen berechnet. Die ersten beiden Fibonacci-Zahlen sind 0 und 1. Alle weiteren Fibonacci-Zahlen sind die Summe ihrer beiden Vorgänger. Es soll sich also die unendliche Liste `[0,1,1,2,3,5,8,...]` ergeben.

Hinweise:

- Die vordefinierten Funktionen `zipWith` vom Typ `(a -> b -> c) -> [a] -> [b] -> [c]` und `tail` vom Typ `[a] -> [a]` könnten hilfreich sein. Erstere berechnet zu einer Funktion `f` vom Typ `a -> b -> c` und zwei gleich langen Listen `xs` vom Typ `[a]` und `ys` vom Typ `[b]` eine Liste `zs` vom Typ `[c]`, wobei für die i-ten Elemente `x` der Liste `xs`, `y` der Liste `ys` und `z` der Liste `zs` gilt, dass `z == f x y`. Die Funktion `tail` berechnet zu einer Liste die Liste ohne ihr erstes Element.

b) Das sogenannte Zeckendorf Theorem (nach Edouard Zeckendorf) besagt, dass jede natürliche Zahl eindeutig als Summe von verschiedenen Fibonacci-Zahlen, welche in der Folge der Fibonacci-Zahlen nicht direkt aufeinander folgen, darstellbar ist. Z. B. lässt sich die Zahl 100 als $89 + 8 + 3$ darstellen. Es gibt noch weitere Möglichkeiten, diese Zahl als Summe von Fibonacci-Zahlen darzustellen (z. B. $89+8+2+1$ oder $55 + 34 + 8 + 3$), aber diese anderen Zerlegungen verletzen die Bedingung, dass keine zwei aufeinanderfolgenden Fibonacci-Zahlen verwendet werden dürfen (im Beispiel sind dies 2 und 1 bzw. 55 und 34).

Diese eindeutige Zerlegung lässt sich durch einen Greedy-Algorithmus berechnen, welcher stets die größtmögliche Fibonacci-Zahl, welche die Bedingungen des Zeckendorf Theorems erfüllt, von der noch zu zerlegenden Zahl abzieht, bis diese vollständig zerlegt ist.

Implementieren Sie in Haskell die Funktion `zeckendorf` vom Typ `Int -> [Int]`, welche zu einer natürlichen Zahl n die Liste der Fibonacci-Zahlen berechnet, welche für die Summendarstellung von n nach dem Zeckendorf Theorem verwendet werden (auf Zahlen kleiner als 1 darf sich Ihre Funktion beliebig verhalten). So soll z. B. der Aufruf `zeckendorf 100` das Ergebnis `[89,8,3]` berechnen.

Hinweise:

- Die vordefinierten Funktionen `reverse` und `tail` vom Typ `[a] -> [a]`, welche zu einer Liste die Liste in umgekehrter Reihenfolge bzw. die Liste ohne ihr erstes Element berechnen, die vordefinierte Funktion `takeWhile` vom Typ `(a -> Bool) -> [a] -> [a]`, welche zu einer Funktion `f` vom Typ `a -> Bool` und einer Liste `xs` vom Typ `[a]` eine Liste berechnet, welche die ersten Elemente `x` von `xs` enthält, für die der Ausdruck `f x` zu `True` auswertet, und die Funktion `fibs` aus der vorigen Teilaufgabe könnten hilfreich sein.

Lösung:

a)
```
fibs :: [Int]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

b)
```
zeckendorf :: Int -> [Int]
zeckendorf x = zHelper x (reverse (takeWhile (<= x) fibs))
   where
   zHelper 0 _      = []
```

```
zHelper x (y:ys) | y > x = zHelper x ys
                 | otherwise = y : (zHelper (x - y) (tail ys))
```