Aufgabe (1) Algebraische Datentypen: Wir betrachten den Typ

```
type Color = String data CrocoFam = Alligator Color CrocoFam
| OldAlligator CrocoFam CrocoFam | Egg Color
```

(a) Schreibe eine Funktion `countEggs :: Color -> CrocoFam -> Int`, die die Anzahl der Eier in der gegebenen Farbe zählt.

(b) In welchem Sinn ist eine CrocoFam als Baum aufzufassen?

(c) Schreibe eine Funktion `inOrder :: CrocoFam -> [String]`, die den CrocoFam-Baum inorder traversiert und jedes Ei durch den String "Ei in ....", jeden alten Alligator durch "alten Alligator" und jeden Alligator durch "hungriger Alligator in Farbe" ersetzt.

Aufgabe (2) Monaden. Wir betrachten die Operation

```
(<.>) :: Monad m => (b -> m c) -> (a -> m b) -> (a -> m c)
(f <.> g) x = g x >>= f
```

Man beweise:

(a) `f <.> return = return <.> f = f`

(b) `(f <.> g) <.> h = f <.> (g <.> h)`

Aufgabe (3) Lambdas

Man werte folgende Lambdaausdrücke aus, sofern sie gültig sind

(a) $(\lambda xyz.y((xy)z))(\lambda ab.ab)$

(b) $\lambda xy.y(\lambda abc.a(abc))x)(\lambda ab.ab)(\lambda ab.a/ab))$

(c) $(\lambda x.x)\lambda y$

(d) Man gebe einen Lambdaausdruck an, der folgender Haskell-Funktion entspricht:

```
g :: Int -> Int
g 0 = 1
g 1 = 1
g n = g (n-1) + 3 * g (n-2)
```

Zahlen und Nulltest sowie +, *, T und F brauchen nicht ausgeschrieben zu werden!

Aufgabe (4) Listen

Man definiere in Haskell

(a) Die Liste aller geraden Zahlen

(b) Die Liste aller Primzahlen

Aufgabe (5) Induktion:

Man beweise oder widerlege durch Induktion

(a) `length = length . reverse`

(b) `concat . map (map f) = map f . concat`

(c) `foldr f s = foldl (flip f) s . reverse`

Aufgabe (6) Codierung

(a) Man konstruiere den Huffmann-Baum für den Satz "OTTOIST-TOLL" und gebe den codierten String an.

(b) Wie viele Fehler erkennt der Code mit Paritätsbit? Begründe Deine Behauptung!

(c) Alice hat sich folgende Codierungstabelle überlegt: `a -> 00, b -> 1, c -> 01, d -> 000` Ist das eine gute Idee? Begründung?

Aufgabe (7) Falten

Man definiere die folgenden Funktionen mit Hilfe einer Faltung

(a) `sum :: [Int] -> Int` (Summe einer Liste von Zahlen)

(b) `concatMap :: (a->[b]) -> [a] -> [b]` (Wendet Funktion auf jedes Element einer Liste an und hängt die Listen zusammen.)

Man definiere die folgenden Funktionen mit Hilfe einer Entfaltung:

(c) `map :: (a -> b) -> [a] -> [b]` (Wendet Funktion auf jedes Element einer Liste an)

(d) `zip :: [a] -> [b] -> [(a,b)]` (Klebt die Listen elementweise zusammen)

Aufgabe (8) Binärbäume

Betrachte den Typ

```
data Tree a = Node (Tree a) (Tree a) | Leaf a
```

also volle Binärbäume, an deren Blättern `as` hängen. Mache `Tree a` unter der Annahme `Eq a` zu einer Instanz von `Eq`. Dabei sollen zwei Bäume als gleich gelten, wenn sie durch eine Folge von "linkes und rechtes Kind vertauschen"-Operationen ineinander überführt werden können.

Aufgabe (9) Transposition

Schreibe eine Funktion `transpose :: [[a]] -> [[a]]`, die eine Liste von gleichlangen (das darf vorausgesetzt werden und braucht nicht überprüft zu werden) transponiert, also "Zeilen" und "Spalten" vertauscht, z. B.

```
transpose([[1,2,3],[4,5,6],[7,8,9]])=[[1,4,7],[2,5,8],[3,6,9]]
```

Aufgabe (10) Listen

Man zeige, dass für alle Listen von Listen von Listen

`concat.(map concat) = concat.concat`

Aufgabe (11) Datentypen

Man betrachte den Datentyp

```
data RumCola = Rum Int|Cola Int|Mischung RumCola RumCola
```

(a) Schreibe eine Funktion `anteile :: RumCola -> (Int,Int)` die berechnet, wieviel Rum und wieviel Cola enthalten ist.

(b) Mache `RumCola` zur Instanz der Klasse `Eq`, wobei zwei Getränke gleich sein sollen, wenn in Ihnen das Verhältnis von Rum und Cola gleich ist.

Aufgabe (12) Mehr Datentypen

Man betrachte die Datentypen

```
data Chuck a = R a
data Norris a = B a
data RBListe a b =
    Nil |
    CR (Chuck a) (RBListe a b) |
    CB (Norris b) (RBListe a b)
```

(a) Schreibe eine Funktion `leFold :: (a -> c -> c) -> (b -> c -> c) -> c -> RBListe a b -> c`

(b) Schreibe eine Funktion `leMap :: (a -> b) -> (c -> d) -> RBListe a c -> RBListe c d`, die foldRB benutzt.

Aufgabe (12) Mehr Lambdas!

Man werte aus, sofern gültig:

(a) $\big(\lambda xyz.y(xyz)\big)\big(\lambda xyz.y(xyz)\big)$

(b) $(\lambda das.ist)(\lambda to.ll)$

(c) $(\lambda x.\lambda)(\lambda y.y)$

(d) Man gebe einen Lambdaausdruck an, der für gerade natürliche Zahlen als Eingabe zu `true` und für ungerade zu `false` auswertet (Hinweis: Zum Beispiel rekursiv ...)