Tutoraufgabe 1 (Überblickswissen):

a) In Haskell haben Funktionen keine Seiteneffekte. Welche Vorteile ergeben sich daraus?

b) Angenommen Haskell würde Seiteneffekte erlauben, sehen Sie Probleme, die bei der Auswertungsstrategie auftreten könnten?

Tutoraufgabe 2 (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].

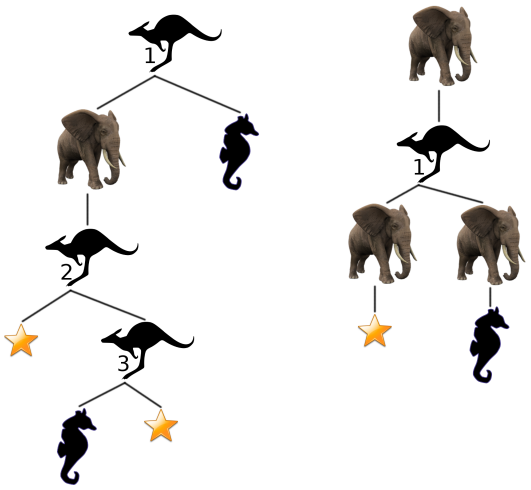

Abbildung 1: Zwei beispielhafte Mobiles.

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
mobileLinks = ...
```

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

Hinweise:

[1] Für die Grafik wurden folgende Bilder von Wikimedia Commons (2012) verwendet:

- Stern `https://commons.wikimedia.org/wiki/File:Crystal_Clear_action_bookmark.png`
- Seepferdchen `https://commons.wikimedia.org/wiki/File:Seahorse.svg`
- Elefant `https://commons.wikimedia.org/wiki/File:African_Elephant_Transparent.png`
- Känguru `https://commons.wikimedia.org/wiki/File:Kangourou.svg`

- 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.

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. Sie dürfen die vordefinierte Funktion `++` verwenden.

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 als 1 oder größer als die Anzahl der Figuren im Mobile ist, darf sich Ihre Funktion beliebig verhalten.

Tutoraufgabe 3 (Datenstrukturen (Video)):

In dieser Aufgabe geht es darum, arithmetische Ausdrücke auszuwerten. Wir betrachten Ausdrücke auf den ganzen Zahlen (`Int`) mit Variablen sowie den Operationen der Addition und Multiplikation.

a) Wir wollen Variablennamen durch den Datentyp `VariableName` darstellen. In dieser Aufgabe betrachten wir nur die Variablen `X` und `Y`.

Erstellen Sie den Datentyp `VariableName`, sodass er entweder den Wert `X` oder den Wert `Y` annehmen kann. Erstellen Sie außerdem die Funktion `getValue :: VariableName -> Int`, welche der Variablen `X` den Wert `5` und der Variablen `Y` den Wert `13` zuordnet. Die Auswertung des Ausdrucks `getValue X` soll also `5` ergeben.

b) Nun wollen wir arithmetische Ausdrücke durch den Datentyp `Expression` darstellen. Ein arithmetischer Ausdruck ist entweder ein konstanter `Int`-Wert, der Name einer Variablen (`VariableName`), die Addition zweier `Expressions` oder die Multiplikation zweier `Expressions`.

Erstellen Sie den entsprechenden Datentyp `Expression` mit den Datenkonstruktoren `Constant`, `Variable`, `Add` und `Multiply`.

Hinweise:

- Auch hier und bei `VariableName` ist es hilfreich, `deriving Show` an das Ende der Datentyp-Deklaration zu schreiben.

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

c) Um eine `Expression` zu einem `Int` auszuwerten, benötigen wir die `Expression` selbst sowie die Funktion `getValue`, welche den einzelnen Variablen Werte zuordnet. Falls die `Expression` ein konstanter `Int`-Wert ist, so ist eben dieser `Int`-Wert das Ergebnis. Falls die `Expression` eine Variable ist, so ist das Ergebnis der `Int`-Wert, welcher der Variablen von der Funktion `getValue` zugeordnet wird. Falls die `Expression` die Addition bzw. Multiplikation zweier `Expressions` ist, so werden zunächst diese beiden `Expressions` ausgewertet und die beiden `Int`-Werte anschließend miteinander addiert bzw. multipliziert, um das Ergebnis zu erhalten.

Erstellen Sie die entsprechende Funktion `evaluate :: Expression -> Int`.

Angenommen `exampleExpression :: Expression` sei wie folgt definiert.

```
exampleExpression = Add
                      (Add
                        (Constant 20)
                        (Constant 17))
                      (Add
                        (Variable X)
                        (Multiply
                          (Add
                            (Constant 14)
                            (Constant 7))
                          (Constant 2)))
```

Der Ausdruck `evaluate exampleExpression` würde dann zum Wert `84` ausgewertet.

Hinweise:

- Die `exampleExpression` entspricht dem arithmetischen Ausdruck $(20 + 17) + (x + ((14 + 7) \cdot 2))$.

d) Wird eine `Expression` mehrfach ausgewertet, so ist es wünschenswert, sie möglichst klein zu halten, damit die Auswertung möglichst schnell geht. Wir wollen nun eine gegebene `Expression` unter der Annahme unbekannter Variablenwerte optimieren. In einem ersten Schritt fassen wir dazu die Addition zweier Konstanten zu einer neuen Konstanten zusammen, welche als Wert die Summe der Werte der beiden ursprünglichen Konstanten hat. Mit der Multiplikation gehen wir analog vor. Alle anderen `Expressions` bleiben unverändert.

Erstellen Sie die entsprechende Funktion `tryOptimize :: Expression -> Expression`.

Der Ausdruck `tryOptimize (Add (Constant 20) (Constant 17))` würde beispielsweise zum Wert `Constant 37` ausgewertet. Die Ausdrücke `tryOptimize (Add (Variable X) (Constant 2))` und `tryOptimize (Multiply (Add (Constant 14) (Constant 7)) (Constant 2))` würden hingegen ihren Parameter genau so zurückgeben, d.h. sie werten zu `Add (Variable X) (Constant 2)` bzw. `Multiply (Add (Constant 14) (Constant 7)) (Constant 2)` aus.

e) In komplexen `Expressions` kann es Teilausdrücke geben, welche nur aus der Berechnung von Konstanten bestehen. Diese Teilausdrücke können durch *partielle Auswertung* vollständig durch eine neue Konstante ersetzt werden, welche als Wert die Evaluation des Teilausdrucks hat.

Erstellen Sie die entsprechende Funktion `evaluatePartially :: Expression -> Expression`. Für eine Addition werden zunächst die beiden Teilausdrücke partiell ausgewertet. Das Ergebnis ist nun die mit `tryOptimize` optimierte Addition der partiell ausgewerteten Teilausdrücke. Die Multiplikation arbeitet analog. Alle anderen Ausdrücke bleiben unverändert.

Der Ausdruck `evaluatePartially exampleExpression` würde beispielsweise zum Wert `Add (Constant 37) (Add (Variable X) (Constant 42))` ausgewertet.

Tutoraufgabe 5 (Typen):

Bestimmen Sie zu den folgenden Haskell-Funktionen `f`, `g` und `h` 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.

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

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

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

Hinweise:

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