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?

Lösung:

a) Daraus ergibt sich eine Vielzahl von Vorteilen. Es wird deutlich einfacher, über das Programm nachzudenken, da man sich sicher sein kann, dass die Funktion keine unerwarteten Nebeneffekte hat. Das macht es auch simpler, die Korrektheit eines Programms zu beweisen. Außerdem erleichtert eine Garantie der Seiteneffektfreiheit die Parallelisierung von Programmcode, da sich so der Code nicht gegenseitig beeinflusst.

b) Wenn Seiteneffekte erlaubt wären, so würde die Bedarfsevaluation (lazy evaluation) nicht mehr einfach umsetzbar sein. Jeder spätere Ausdruck könnte Seiteneffekte haben, so dass entweder jede*r Programmierer*in genau Bescheid wissen muss, wann welcher Ausdruck ausgewertet werden kann/soll oder das Programm wird nahezu unmöglich vorhersehbar.

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].

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:

- 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 (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`

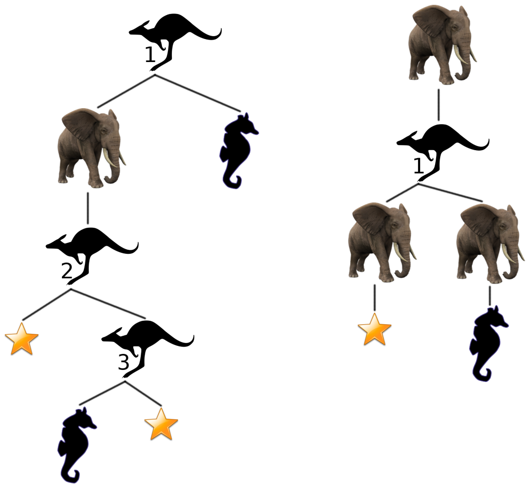

Abbildung 1: Zwei beispielhafte Mobiles.

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. 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.

Lösung:

a)
```
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
```

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

```
                            Stern
                            (Kaenguru 3
                                Seepferdchen
                                Stern
                            )
                      ))
                      Seepferdchen
```

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

b)
```
count :: Mobile a -> Int
count Stern                = 1
count Seepferdchen         = 1
count (Elefant     m)      = 1 + count m
count (Kaenguru _  m1 m2)  = 1 + count m1 + count m2
```

c)
```
liste :: Mobile a -> [a]
liste Stern                    = []
liste Seepferdchen             = []
liste (Elefant m)              = liste m
liste (Kaenguru inhalt m1 m2)  = inhalt : liste m1 ++ liste m2
```

d)
```
greife :: Mobile a -> Int -> Mobile a
greife x                    1 = x
greife (Elefant m)          x = greife m (x-1)
greife (Kaenguru _ m1 m2)   x
  | x-1 <= count m1   = greife m1 (x-1)
  | otherwise         = greife m2 (x-1 - count m1)
greife _                    _ = Stern -- Gesuchte Figur existiert nicht
```

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.

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.

Lösung:

a)
```
data VariableName = X | Y deriving Show

getValue :: VariableName -> Int
getValue X = 5
getValue Y = 13
```

b)
```
data Expression = Constant  Int
                | Variable  VariableName
                | Add       Expression Expression
                | Multiply  Expression Expression
                deriving Show
```

c)
```
evaluate :: Expression -> Int
evaluate (Constant c)       = c
evaluate (Variable v)       = getValue v
evaluate (Add e1 e2)        = evaluate e1 + evaluate e2
evaluate (Multiply e1 e2)   = evaluate e1 * evaluate e2
```

d)
```
tryOptimize :: Expression -> Expression
tryOptimize (Add      (Constant c1) (Constant c2)) = Constant (c1 + c2)
tryOptimize (Multiply (Constant c1) (Constant c2)) = Constant (c1 * c2)
tryOptimize e = e
```

e)
```
evaluatePartially :: Expression -> Expression
evaluatePartially (Add e1 e2)      = tryOptimize (Add
                                                   (evaluatePartially e1)
                                                   (evaluatePartially e2))
evaluatePartially (Multiply e1 e2) = tryOptimize (Multiply
                                                   (evaluatePartially e1)
                                                   (evaluatePartially e2))
evaluatePartially e                = e
```

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.

Lösung:

a)
```
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 `z` 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`.

b)
```
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`.

c)
```
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 `c = [Bool]`.

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