Prof. Dr. J. Giesl
D. Cloerkes, S. Dollase, D. Meier

Aufgabe 3 (Datenstrukturen): (6 + 8 + 6 + 20 + 6 = 46 Punkte)

In dieser Aufgabe betrachten wir eine spezielle Art von Mehrwegbäumen. In diesen Bäumen gibt es zwei Arten von Knoten. Die erste Art sind *Index*-Knoten. Diese speichern zwei Werte und können eine beliebige Anzahl Nachfolger haben (auch 0 Nachfolger sind möglich). Die zweite Art sind *Daten*-Knoten. Diese speichern nur einen Wert und sind immer Blätter.
Der Baum `t1` in Abbildung 1 ist ein solcher Baum. Im Folgenden dürfen Sie keine vordefinierten Funktionen nutzen, die nicht explizit in den Teilaufgaben angegeben sind.

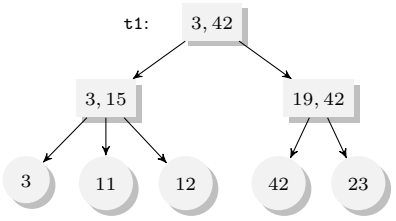

Abbildung 1: Ein Mehrwegbaum

a) Entwerfen Sie einen parametrischen Datentyp `MultTree a`, der zur Darstellung der oben beschriebenen Bäume genutzt werden kann.

Hinweise:

- Für die nachfolgenden Aufgaben ist es hilfreich, sich den Baum `t1` als Konstante zu definieren:
```
t1 :: MultTree Int
t1 = ...
```
- Ergänzen Sie `deriving Show` am Ende der Datenstruktur, damit `GHCi` die Bäume auf der Konsole anzeigen kann: `data ... deriving Show`

b) Schreiben Sie eine Funktion `verzweigungsgrad`. Diese bekommt einen Mehrwegbaum vom Typ `MultTree a` übergeben und berechnet, wie viele Nachfolger ein Knoten in diesem Baum maximal hat.

Für den Beispielbaum würde der Aufruf `verzweigungsgrad t1` also den Rückgabewert 3 liefern. Sie dürfen die Funktion `max` benutzen, die das Maximum zweier Werte zurückgibt.

c) Schreiben Sie eine Funktion `datenListe`, die alle in den Daten-Knoten des übergebenen Baumes gespeicherten Werte in einer Liste zurück gibt. Die Werte dürfen in beliebiger Reihenfolge ausgegeben werden und Werte, die mehrfach vorkommen, sollen auch mehrfach ausgegeben werden.

Der Aufruf `datenListe t1` könnte also z.B. `[3, 11, 12, 42, 23]` ergeben.

d) Schreiben Sie eine Funktion `datenIntervalle`, die einen Baum des Typs `MultTree Int` übergeben bekommt. Diese verändert die Index-Knoten so, dass der kleinere der beiden darin gespeicherten Werte dem kleinsten Wert entspricht, der in einem Daten-Knoten in diesem Teilbaum gespeichert ist. Analog soll der größere der beiden Werte dem größten Wert in einem Daten-Knoten in diesem Teilbaum entsprechen.

Für einen (Teil-)Baum ohne Daten-Knoten soll das Intervall `maxBound, minBound` in die Index-Knoten geschrieben werden.

Der Aufruf `datenIntervalle t1` würde also einen Baum zurückgeben, der sich von `t1` nur dadurch unterscheidet, dass `15` und `19` durch `12` bzw. `23` ersetzt sind.

Sie dürfen die vordefinierten Konstanten `minBound :: Int` und `maxBound :: Int` benutzen, die den kleinsten und den größten möglichen Wert vom Typ `Int` liefern. Außerdem dürfen Sie die vordefinierten Funktionen `min` und `max` nutzen, die das Minimum bzw. Maximum von 2 übergebenen Werten berechnen.

e) Schreiben Sie eine Funktion `contains`, die überprüft, ob ein gegebener Wert in einem Baum vom Typ `MultTree Int` enthalten ist. Gehen Sie dabei davon aus, dass alle Bäume, die der Funktion übergeben werden, bereits das Format haben, das die Funktion `datenIntervalle` erzeugt. Ein Knoten mit den Werten x, y hat also nur solche Blätter als Nachfolger, deren Wert im Intervall $[x, y]$ liegt. Nutzen Sie dies, um ihre Implementierung effizienter zu gestalten!

Aufgabe 5 (Typen): (9 + 9 + 9 = 27 Punkte)

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

i)
```
f 1 ys _ = ys
f x (y:ys) z = if x > z then f (x - 1) (x:ys) z else (y:ys)
```

ii)
```
g x (y:ys) = g (y x) ys
g x y      = x []
```

iii)
```
h [] x y = if x == 1 then h y (length y) [] else True
h (a:as) x y = h as (x + a) 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.

Aufgabe 7 (Funktionen höherer Ordnung): (4 + 4 + 4 + 6 + 11 = 27 Punkte)

In Java können alle Referenztypen den leeren Wert (`null`) enthalten. Dies ist vor allem dann ein Problem, wenn der Entwickler einer Methode annimmt, dass ein Parameter mit Referenztyp nie den leeren Wert enthalten kann, ein Benutzer der Methode diesen jedoch trotzdem übergibt. So entstehen viele `NullPointerExceptions`. In einigen anderen Sprachen gibt es dieses Problem nicht, da der leere Wert explizit vom Typ der Variablen zugelassen werden muss. Auch in Haskell können Variablen nicht automatisch den leeren Wert annehmen. Um einen leeren Wert zuzulassen, hat Haskell den Typ `Maybe a`, welcher entweder den leeren Wert (`Nothing`) oder einen vorhandenen Wert (`Just a`) enthält.
In dieser Aufgabe werden wir eine alternative Version dieses Typs als `data Optional a = Empty | Present a deriving Show` selbst implementieren. Wenn wir also eine Variable vom Typ `Int` deklarieren wollen, welche auch den leeren Wert enthalten darf, so erhält sie den Typ `Optional Int`. Der leere Wert wird dann durch `Empty` repräsentiert und der nicht-leere Wert `10` wird durch `Present 10` dargestellt.

a) Auf einen Wert `o` des Typs `Optional a` können wir nicht direkt eine Funktion `f` des Typs `a -> b` anwenden, da diese nicht den Fall betrachtet, dass `o` auch der leere Wert sein kann. Um `f` trotzdem auf `o` anwenden zu können, definieren wir eine Funktion `mapOptional :: (a -> b) -> Optional a -> Optional b`, welche als erstes Argument die Funktion `f` und als zweites Argument den Wert `o` erhält. Zurückgegeben wird `Empty`, falls das zweite Argument `Empty` ist. Ansonsten wird auf den im zweiten Argument enthaltenen Wert das erste Argument angewendet und das Ergebnis davon in ein `Present` gekapselt und zurückgegeben. Die Auswertung von `mapOptional (\x -> 2*x) Empty` ergibt also `Empty` und die Auswertung von `mapOptional (\x -> 2*x) (Present 5)` ergibt `Present 10`.

Implementieren Sie die entsprechende Funktion `mapOptional`.

b) Ein weiterer Anwendungsfall ist, dass man einen vorhandenen Wert verwerfen möchte, falls dieser eine bestimmte Bedingung nicht erfüllt. Hierzu definieren wir die Funktion `filterOptional :: (a -> Bool) -> Optional a -> Optional a`, welche als erstes Argument eine Funktion erhält, die für einen gegebenen Wert entweder `True` oder `False` zurückgibt. Das zweite Argument ist ein vielleicht leerer Wert. Die Rückgabe ist das zweite Argument, falls dieses nicht-leer ist und das erste Argument angewendet auf den im zweiten Argument enthaltenen Wert `True` ist. Ansonsten wird `Empty` zurückgegeben. Die Auswertung von `filterOptional (\x -> x > 0) Empty` ergibt also `Empty`. Die Auswertung von `filterOptional (\x -> x > 0) (Present -5)` ergibt ebenfalls `Empty`. Die Auswertung von `filterOptional (\x -> x > 0) (Present 5)` ergibt hingegen `Present 5`.

Implementieren Sie die entsprechende Funktion `filterOptional`.

c) In dieser Teilaufgabe wollen wir die Funktion `foldOptional :: (a -> b) -> b -> Optional a -> b` definieren, welche einen vorhandenen Wert (drittes Argument) mit einer gegebenen Funktion (erstes Argument) abbildet und das Ergebnis davon zurückgibt. Ist der Wert leer (drittes Argument), so wird ein Standardwert (zweites Argument) zurückgegeben. Die Funktion `foldOptional` erlaubt uns also, im Gegensatz zu `mapOptional`, einen leeren Wert in einen nicht-leeren Wert umzuwandeln. Die Auswertung von `foldOptional (\x -> 2*x) -1 Empty` ergibt also `-1`. Die Auswertung von `foldOptional (\x -> 2*x) -1 (Present 5)` ergibt hingegen `10`.

Implementieren Sie die entsprechende Funktion `foldOptional`.

d) Nun wollen wir den oben deklarierten Typ sowie die dazugehörigen Funktionen anwenden. Dazu deklarieren wir zunächst einen neuen Datentyp `data Product = Article String Int deriving Show`. Ein Wert des Typs `Product` beschreibt einen zum Verkauf stehenden Artikel in einem Geschäft. Dabei ist der `String`-Wert der Artikelname und der `Int`-Wert der Preis in Cent.

Implementieren Sie die Funktion `isHumanEatable :: Product -> Bool`, welche einen Artikel erhält und genau dann `False` zurückgibt, wenn der übergebene Artikel den Namen `Dog Food` trägt. Der Ausdruck `isHumanEatable "Dog Food"` soll also zu `False` ausgewertet werden. Der Ausdruck `isHumanEatable "Pizza"` soll hingegen zu `True` ausgewertet werden.

Implementieren Sie außerdem die Funktion `adjustPrice :: Product -> Product`, welche einen Artikel erhält und einen Artikel gleichen Namens zurückgibt. Kostet der übergebene Artikel weniger als 10 Euro, d.h. weniger als 1000 Cent, so wird der Preis verdoppelt. Ansonsten bleibt der Preis gleich. Der Ausdruck `adjustPrice (Article "Pizza" 1000)` soll also zu `Article "Pizza" 1000` ausgewertet werden. Der

Ausdruck `adjustPrice (Article "Pizza" 100)` soll hingegen zu `Article "Pizza" 200` ausgewertet werden.

Implementieren Sie schließlich noch die Funktion `stringify :: Product -> String`, welche einen Artikel erhält und einen `String` zurückgibt. Der Ausdruck `stringify (Article "Pizza" 1000)` soll beispielsweise zu `"The article named 'Pizza' costs 1000 Cent. "` ausgewertet werden.

Hinweise:

- Sie können die vordefinierte Funktion `show` verwenden, um einen `Int`-Wert in einen `String` zu überführen. So ergibt `show 1000` den String `"1000"`.

e) In dieser Aufgabe wollen wir den Text für das Preisschild eines Artikels generieren. Dazu definieren wir die Funktion `toPriceTag :: Product -> String`, welche indirekt die Funktionen `mapOptional`, `filterOptional` und `foldOptional` sowie `isHumanEatable`, `adjustPrice` und `stringify` nutzt, um folgende Funktionalität zu implementieren.

Das Geschäft verkauft ausschließlich für Menschen genießbare Artikel. Hundefutter kann also nicht verkauft werden. Für einen Artikel mit dem Namen `Dog Food` liefert `toPriceTag` also das Preisschild `"This article is unavailable."`. Für alle anderen Artikel wird der Preis verdoppelt, falls er kleiner als 10 Euro ist, und anschließend der Artikel mit der `stringify`-Funktion in einen `String` umgewandelt. Hierzu definieren wir drei Hilfsfunktionen.

Implementieren Sie die Funktion `filterHumanEatable :: Product -> Optional Product`, welche die Funktion `filterOptional` nutzt, um `isHumanEatable` auf ihr Argument anzuwenden. Hierzu muss das Argument zunächst in ein `Present` gekapselt werden.

Implementieren Sie die Funktion `adjustPriceO :: Optional Product -> Optional Product`, welche die Funktion `mapOptional` nutzt, um `adjustPrice` auf ihr Argument anzuwenden.

Implementieren Sie die Funktion `stringifyO :: Optional Product -> String`, welche die Funktion `foldOptional` nutzt, um ihr Argument mit Hilfe von `stringify` in einen `String` umzuwandeln. Falls das Argument `Empty` ist, so soll `This article is unavailable.` zurückgegeben werden.

Implementieren Sie nun die Funktion `toPriceTag :: Product -> String`, welche ihr Argument an die Funktion `filterHumanEatable` übergibt, dessen Rückgabewert an `adjustPriceO` übergibt, dessen Rückgabewert an `stringifyO` übergibt und dessen Rückgabewert selbst zurückgibt.

Der Ausdruck `toPriceTag (Article "Dog Food" 1000)` sollte also zu `"This article is currently unavailable."` ausgewertet werden. Der Ausdruck `toPriceTag (Article "Pizza" 1000)` sollte zu `"The article named 'Pizza' costs 1000 Cent."` ausgewertet werden. Der Ausdruck `toPriceTag (Article "Pizza" 100)` sollte zu `"The article named 'Pizza' costs 200 Cent."` ausgewertet werden.