Prof. Dr. J. Giesl
S. Dollase, M. Hark, D. Korzeniewski

Aufgabe 2 (Datenstrukturen in Haskell): (2 + 1 + 2 + 2.5 + 3.5 = 11 Punkte)

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 trägt. 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.

Aufgabe 4 (Typen in Haskell): (2 + 2 + 2 = 6 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 Funktion `length` den Typ `[a] -> Int` hat.

i)
```
f [] x ys = x:ys
f s _ ys = s:ys
```

ii)
```
g x y [] = if length (x:[[]]) == 2 then y:[] else (length x):[]
g x y (z:zs) = length (y:z:[]) : []
```

iii)
```
h f x (y:ys) = f y : (h f x ys)
h _ x []     = [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.

Aufgabe 6 (Funktionen höherer Ordnung): (1 + 1 + 1 + 1.5 + 3.5 = 8 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 den leeren Wert enthalten 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 einem `Present` gewrappt 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, welche 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 vielleicht 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.

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` und `adjustPrice` nutzt, um folgende Funktionalität zu implementieren.

Das Geschäft verkauft ausschließlich für Menschen genießbare Artikel. Hundefutter kann also nicht verkauft werden. Ein Artikel mit dem Namen `Dog Food` erhält 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` gewrappt werden.

Implementieren Sie die Funktion `adjustPrice0 :: Optional Product -> Optional Product`, welche die Funktion `mapOptional` nutzt, um `adjustPrice` auf ihr Argument anzuwenden.

Implementieren Sie die Funktion `stringify0 :: 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 `adjustPrice0` übergibt, dessen Rückgabewert an `stringify0` übergibt und dessen Rückgabewert selbst zurückgibt.

Der Ausdruck `toPriceTag (Article "Dog Food" 1000)` sollte 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.

Aufgabe 8 (Unendliche Datenstrukturen): (1 + 2 = 3 Punkte)

Vor langer langer Zeit haben wir in unserem Garten einen schönen Rosenbusch gepflanzt. Leider ist dieser inzwischen sehr groß gewachsen. Schlimmer noch: Wann immer wir versuchen an sein Ende zu gelangen, wächst er unaufhörlich weiter. Um unser Haus verlassen zu können haben wir nur eine Wahl. Wir müssen ihn auf eine feste Größe zurecht schneiden.

a) Deklarieren Sie einen Datentypen `Rosebush`, welcher entweder eine Rose (`Rose`), ein abgeschnittener Ast (`Cut`), ein Ast mit weiterem Rosenbusch (`Stalk`) oder eine Gabelung mit zwei weiteren Rosenbüschen (`Fork`) ist.

Implementieren Sie eine Funktion `generate :: Rosebush`, welche folgenden unendlichen Rosenbusch generiert. Er wird dadurch unendlich, dass an der Stelle der `O`s jeweils der selbe Rosenbusch erneut eingefügt wird. Ein `|` ist dabei ein `Stalk`, ein `+--+-+` ist ein `Fork` und ein `*` ist eine `Rose`.

```
       O
       |
     * |
O * | |
| | +++
+++  |
 |   |
 +-+-+
   |
```

Die Funktion beginnt mit `generate = Stalk (Fork (Stalk (...)) (Stalk (...)))`.

Hinweise:

- Die Anzahl der Striche (`-`) pro `Fork` kann 0 sein, aber ebenso beliebig groß werden. Sie ist davon abhängig, welche Form die beiden weiteren Rosenbüsche haben.
- Mit der Funktion `showMe`, welche in der auf unserer Website verfügbaren Datei `rosebush.hs` definiert ist, können Sie einen `Rosebush r` wie folgt ausgeben: `showMe r`
- In der Ausgabe ist die Wurzel des Rosenbuschs unten.
- Wenn Sie `showMe generate` ausführen, so terminiert die Auswertung nicht, da `generate` eine unendliche Datenstruktur generiert.

b) Implementieren Sie die Funktion `cut :: Int -> Rosebush -> Rosebush`, um einen gegebenen Rosenbusch auf eine gegebene Höhe zurechtzuschneiden. Hierbei wird alles, was über die gegebene Höhe hinaus wächst, abgeschnitten, also durch `Cut` ersetzt.

Der durch `generate` generierte Rosenbusch, auf die Höhe 10 geschnitten, sieht wie folgt aus, wobei `X` ein `Cut` ist. Er kann mit `showMe (cut 10 generate)` ausgegeben werden.

```
X X X X     X X
| | +++     +++
+++  |       |
 |   |       |
 +-+-+     *  |
   |     * |  |
   |     | ++-+
   +---++  |
       |   |
       +-+-+
         |
```