Institut für Informatik **Ludwig-Maximilians-Universität München**
Prof. François Bry, Thomas Prokosch
Kilian Rückschloß

Programmierung und Modellierung, SoSe 21
Übungsblatt 6

Abgabe: bis Mo 31.05.2021 08:00 Uhr

Aufgrund der Pfingstfeiertagen ist dieses Übungsblatt auf zwei Wochen Bearbeitungszeit ausgelegt. Da auch der Dienstag nach Pfingsten (25.5.) vorlesungsfrei ist, ***entfallen die Tutorien in der Woche nach Pfingsten****. Die nächsten Tutorien starten in der Woche vom 31.5.*

Aufgabe 6-1 Datentypen und Typklassen

In dieser Aufgabe sollen Sie das Kartendeck für ein sehr einfaches Kartenspiel entwickeln. Zwei Spieler mit jeweils 8 Karten aus einem französischen Kartendeck (Farben: Club, Diamond, Spade, Heart, Werte: Two ... Ace) spielen jeweils eine Karte. Es gibt nur eine Stichregel: Herz sticht immer, legen beide Spieler Herz, dann wird der Kartenwert verglichen. Bei allen anderen Farben wird nur der Kartenwert verglichen.

a) Implementieren Sie zunächst Datentypen für Farbe (`Suit`), Wert (`Value`) und Karte (`Card`).

b) Lassen Sie Ihre Typen für Farbe und Wert von sinnvollen (eingebauten) Typklassen erben. Ein Beispiel wäre die Typklasse Enum, die es erlaubt, Listen dieses Types aufzuzählen, so dass `[Club .. Diamond]` die Liste `[Club, Heart, Spade, Diamond]` erstellt.

c) Für Karten ist es nicht ganz so einfach: Implementieren Sie die Typklassen `Eq` und `Ord` für Ihren Datentyp `Card`. Als Ordnung soll dabei die oben beschriebene Stichlogik implementiert werden.

d) Implementieren Sie jetzt mit Hilfe einer List Comprehension eine Funktion, die ein komplettes Kartendeck als Liste von `Cards` zurückgibt.

Aufgabe 6-2 Rekursive Datentypen: Eigener Listentyp

Gegeben sei der folgende *rekursive* Datentyp, der dazu benutzt werden kann, einfach verkettete Listen in Haskell darzustellen.

```
data ML a = E | L a (ML a) deriving Show
```

`E` steht für die leere Liste, `L` für die Liste mit mindestens einem Element.

Bearbeiten Sie die folgenden Aufgaben aufbauend auf diesem Listentyp. Der vordefinierte Listentyp darf nicht verwendet werden.

a) Geben Sie einen Haskell-Ausdruck vom Typ `ML Int` für eine Liste, welche die Elemente `1`, `2`, `3` und `4` enthält, an.

b) Definieren Sie eine Funktion `myHead`, welche das erste Element einer Liste vom Typ `ML a` zurückgibt. Falls `myHead` auf eine leere Liste angewandt wird, soll der Haskell-Ausdruck `error "empty list"` zurückgegeben werden.

c) Definieren Sie eine Funktion `myTail`, welche alle Elemente außer das erste Element einer Liste vom Typ `ML a` zurückgibt. Falls `myTail` auf eine leere Liste angewandt wird, soll der Haskell-Ausdruck `error "empty list"` zurückgegeben werden.

d) Definieren Sie eine Funktion `myAdd`, welche zwei Listen vom Typ `ML a` übergeben bekommt und eine Liste zurückgibt, in der die jeweils korrespondierenden Listenelemente (d.h., erstes Element aus erster Liste mit erstem Element aus zweiter Liste) addiert sind. Sollten die beiden Listen nicht die gleiche Länge haben, so ist die Länge der kürzeren Liste ausschlaggebend für die Länge der Ergebnisliste.

e) Definieren Sie eine Funktion `myAppend`, welche zwei Listen vom Typ `ML a` konkateniert.

f) Definieren Sie eine Funktion `toString`, welche eine Liste vom Typ `ML a` als String der Form `"1, 2, 3, 4"` zurückgibt.

Aufgabe 6-3 Funktionen höherer Ordnung aus der Standardbibliothek

Implementieren Sie folgende Funktionen höherer Ordnung.

a) Implementieren Sie eine Funktion `any' :: (a -> Bool) -> [a] -> Bool`, die für eine Liste prüft, ob mindestens ein Element ein übergebenes Prädikat erfüllt.

b) Implementieren Sie eine Funktion `all' :: (a -> Bool) -> [a] -> Bool`, die für alle Elemente einer übergebenen Liste prüft, ob diese ein übergebenes Prädikat erfüllen.

c) Implementieren Sie das Verhalten von `map` in einer Funktion `map' :: (a -> b) -> [a] -> [b]` nur mit Hilfe der Standardbibliotheksfunktion `foldr`.

Aufgabe 6-4 Eigene Funktionen höherer Ordnung

Implementieren Sie die folgenden Funktionen.

a) Implementieren Sie eine Funktion `zipWith' :: (a -> b -> c) -> [a] -> [b] -> [c]`, die eine Funktion und zwei Listen als Parameter übergeben bekommt, die Funktion auf die korrespondierenden Elemente der Listen anwendet und eine neue Liste mit den Ergebnissen zurückgibt.

   Für die Länge der Ergebnisliste ist die kürzere der beiden übergeben Listen ausschlaggebend.

b) Implementieren Sie eine Funktion `unzipWith :: (t -> (a, b)) -> [t] -> ([a], [b])`, welche das Gegenteil von `zipWith'` aus Aufgabe a) macht: Von einer gegebenen Liste wird jedes Element mit einer gegebenen Funktion zerlegt. Als Ergebnis wird ein Tupel aus zwei Listen zurückgegeben.

   Beispiel: `unzipWith id [(0,'g'), (8,'u'), (9,'t')]` ergibt das Tupel `([0,8,9], "gut")`.