Institut für Informatik **Ludwig-Maximilians-Universität München**
Prof. Dr. François Bry

Programmierung und Modellierung, SoSe 16
Übungsblatt 8

Abgabe: bis Mo 06.06.2016 10:00 Uhr

Besprechung: ab Di 07.06.2016

Aufgabe 8-1 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 8-2 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 8-3 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")`.