1. Rekursion über Listen

Die folgenden Teilaufgaben sollen ohne Verwendung vordefinierter Bibliotheksfunktionen gelöst werden. Berücksichtigen Sie, dass die zu programmierenden Funktionen nicht für alle Eingaben sinnvoll definiert sind. In diesen Fällen sollen entsprechende Fehlermeldungen ausgegeben werden.

i).

Programmieren Sie eine Haskell-Funktion `elementAnIdx :: Integer -> [el] -> el`, die zu einem Index `i` und einer Liste das i-te Element dieser Liste bestimmt. Das erste Element einer nichtleeren Liste habe dabei den Index 0. Es ergibt sich zum Beispiel:

```
elementAnIdx 2 [3, 4, 5, 6] = 5
```

ii).

Programmieren Sie eine Haskell-Funktion `snoc :: [el] -> el -> [el]`, die an eine Liste ein weiteres Element anhängt. Es ergibt sich zum Beispiel:

```
snoc [1, 2, 3] 5 = [1, 2, 3, 5]
```

iii).

Programmieren Sie eine Haskell-Funktion `rueckwaerts :: [el] -> [el]`, welche eine Liste umdreht, die Elemente der Liste also in umgekehrter Reihenfolge zurück gibt.

iv).

Programmieren Sie eine Haskell-Funktion `praefix :: Integer -> [el] -> [el]`, die für eine gegebene Zahl n und eine Liste l die Liste der ersten n Elemente von l zurück gibt.

v).

Programmieren Sie eine Haskell-Funktion `suffix :: Integer -> [el] -> [el]`, die für eine Zahl n und eine Liste l die Liste der letzten n Elemente von l zurückgibt.

2. Mengen als Listen

Mengen können in Haskell durch Listen repräsentiert werden. Allerdings unterscheiden sich Mengen und Listen in zwei Punkten:

- Die Elemente einer Liste besitzen eine Reihenfolge, die Elemente von Mengen nicht.
- Derselbe Wert darf in einer Liste mehrfach vorkommen, in einer Menge nicht.

Zur Darstellung von Mengen verwenden wir daher nur solche Listen, deren Elemente streng monoton wachsen. Die Reihenfolge der Elemente ist damit festgelegt und eine Dopplung von Werten ist nicht möglich.

Es sollen nun Mengenoperationen auf Basis von Listen implementiert werden. Geben Sie für folgende Funktionen deren allgemeinste Typen an. Implementieren Sie anschließend die Funktionen. Sie können davon ausgehen, dass als Argumente übergebene Mengendarstellungen immer die oben genannte Eigenschaft erfüllen. Sie müssen aber sicherstellen, dass Sie keine ungültigen Mengendarstellungen als Resultate generieren.

i).

Die Funktion `istElement` entscheidet, ob eine Wert in einer Menge enthalten ist und implementiert damit die Relation $\in$. Es ergibt sich z.B.

```
4 `istElement` [1, 3, 5, 7]   ⇝ False
5 `istElement` [1, 3, 5, 7]   ⇝ True
```

ii).

Die Funktion `istTeilmenge` entscheidet, ob eine Menge Teilmenge einer anderen Menge ist und implementiert damit die Relation $\subseteq$. Es ergibt sich z.B.

```
[1, 3] `istTeilmenge` [1, 2, 3, 4, 5]      ⇝ True
[1, 3, 6] `istTeilmenge` [1, 2, 3, 4, 5]   ⇝ False
```

iii).

Die Funktion `vereinigung` bestimmt die Vereinigungsmenge zweier Mengen und implementiert damit die Funktion $\cup$. Es ergibt sich z.B.

```
[1, 2, 4] `vereinigung` [3, 4, 5] ⇝ [1, 2, 3, 4, 5]
```

iv).

Die Funktion `schnitt` bestimmt die Schnittmenge zweier Mengen und implementiert damit die Funktion $\cap$. Es ergibt sich z.B.

```
[1, 2, 4] `schnitt` [3, 4, 5] ⇝ [4]
```

3. Mengen mit Currying

Anders als in der letzten Aufgabe wollen wir nun Mengen nicht als Listen sondern durch ihre Eigenschaften definieren. Gegeben seinen die Typdefinition `Set` sowie die Funktion `myelem`:

```
type Set = (Integer -> Bool)

myElem :: Set -> Integer  -> Bool
myElem    s      e         = s e
```

Der Typ `Set` ist eine Funktion von Integer nach Bool, die entscheidet ob ein Element in der Menge liegt oder nicht. Die Funktion `myElem` werdet den Typ `Set` aus.

Denken Sie für die folgenden Teilaufgaben an Lambda-Ausdrücke und das Currying!

Damit Sie ihre Menge Ausgeben können, wird Ihnen die folgende Funktion `myPrint` gestellt:

```
myPrint :: Set -> Integer -> Integer -> [Integer]
myPrint s min max | min > max    = []
                  | myElem s min = min : (myPrint s (min + 1) max)
                  | otherwise    = myPrint s (min + 1) max
```

Ihr wird eine Menge sowie ein Start- und Endwert übergeben und gibt eine Liste von Integer-Werten zurück.

i). Einelementige Mengen

Schreiben Sie die Funktion `mySingleSet :: Integer -> Set`, welche einen Integerwert erhält und aus diesem eine Menge erstellt.

ii). Mengen durch Eigenschaften

Schreiben Sie die Funktion `myBoolSet :: (Integer -> Bool)-> Set`, welche eine die Eigenschaften einer Menge als Lambda-Ausdrücke erhält und eine Menge zurück gibt.

iii). Vereinigung

Schreiben Sie die Funktion `myUnion :: Set -> Set -> Set`, welche zwei Mengen logisch vereinigt. Denken Sie an die mathematische Definition der Vereinigung.

iv). Hinzufügen

Schreiben Sie die Funktion `myAdd :: Set -> Integer -> Set`, welche einen Integer-Wert zu einer Menge hinzufügt.

v). Durchschnitt

Schreiben Sie die Funktion `myIntersect :: Set -> Set -> Set`, welche zwei Mengen logisch schneidet. Denken Sie an die mathematische Definition des Durchschnitts.

vi). Differenz

Schreiben Sie die Funktion `myDiff :: Set -> Set -> Set`, welche zwei Mengen erhält und die zweite von der ersten Menge abzieht. Denken Sie an die mathematische Definition der Differenz.

vii). Mehrelementige Mengen (nicht so einfach)

Schreiben Sie die Funktion `myListSet :: [Integer] -> Set`, welche eine Liste von Integer-Werten erhält und aus dieser eine Funktion erstellt.

viii). Quantoren

Da wir nun Mengen besitzen können wir nun auch Beweise auf ihnen führen. Schreiben Sie dazu die Funktion `myForAll :: Set -> (Integer -> Bool)-> Integer -> Integer -> Bool`, welche eine Menge auf eine Eigenschaft prüft und nur dann wahr ausgibt, wenn alle Elemente diese Eigenschaft besitzen also den ∀ Quantor. Die letzten beiden Integer-werte sind das Intervall. Die können die Funktion in Anlehnung an die Funktion `myprint` erstellen.

Nach dem Sie den ∀ Quantor erstellt haben, erstellen Sie nun die Funktion `myExists :: Set -> (Integer -> Bool)-> Integer -> Integer -> Bool` die den ∃ Quantor repräsentiert. Denken Sie an die mathematischen Zusammenhang zwischen den beiden Quantoren.

ix). Filter

Schreiben Sie die Funktion `myFilter :: Set -> (Integer -> Bool)-> Set`, welche ein positiv Filter für eine Menge realisiert. Somit wenn ein Element in der Menge ist und den Filter erfüllt, dann ist dies auch in der zurück gegebenen Menge.

x). Teilmenge

Schreiben Sie die Funktion `mySubset :: Set -> Set -> Bool`, welche bestimmt ob die erste Menge eine Teilmenge der zweiten ist. Die Intervallgrenzen können Sie beliebig festlegen.

xi). Äquivalenz

Schreiben Sie die Funktion `myEquals :: Set -> Set -> Bool`, welche bestimmt ob zwei Mengen identisch sind.