

2. Übung zur Vorlesung Programmierung und Modellierung

Hinweis: Für die Bearbeitung der Aufgaben und Hausaufgaben sind bis auf weiteres die Verwendung von Funktionen der Standardbibliothek `tabu`, abgesehen von den Grundoperationen wie `(:)`, `(++)`, `(>)`, `(>=)`, `(<)`, `(<=)`, `div`, `mod`, `not`, `(&&)`, `max`, `min`, etc. Von Ihnen implementierte Funktionen aus vorangegangenen Aufgaben sind zulässig.

A2-1 *Replicate* Die Funktion `replicate :: Int -> a -> [a]` aus der Standardbibliothek wiederholt einen gegebenen Wert n -mal:

```
> replicate 3 "Ha!"  
["Ha!", "Ha!", "Ha!"]  
  
> replicate 7 5  
[5,5,5,5,5,5,5]
```

- a) Implementieren Sie diese Funktion mit Hilfe von Rekursion. Siehe Folien 3-27 & 3-28.
- b) Überprüfen Sie, welche Art der Rekursion Ihre Lösung zur vorherigen Teilaufgabe verwendet! Formulieren Sie ggf. jetzt noch eine *endrekursive* Version.
- c) Implementieren Sie die Funktion erneut, aber dieses mal ohne Rekursion unter Verwendung einer List-Comprehension!

A2-2 *Quicksort* Das Quicksort-Verfahren ist ein rekursiver Sortier-Algorithmus:

Schritt 1: Man wähle irgendein Element der sortierenden Liste, z.B. das erste Element.

Schritt 2: Teile die Restliste in zwei Teillisten auf: eine Teilliste enthalte alle Elemente, welche kleiner oder gleich sind als das gewählte Element; die andere alle größeren Elemente.

Schritt 3: Sortiere beide Teillisten durch rekursive Verwendung des Quicksort-Verfahrens.

Schritt 4: Füge die beiden sortierten Teillisten wieder zusammen, wobei das anfangs ausgewählte Element in die Mitte dazwischen gesteckt wird.

Implementieren Sie diesen Algorithmus in Haskell, indem Sie folgende Funktionen definieren:

```
quicksort :: [Int] -> [Int]  
splitBy   :: Int -> [Int] -> ([Int],[Int])
```

Die Funktion `quicksort` sortiert eine Liste. Die Funktion `splitBy` teilt eine Liste von Zahlen wie angegeben in zwei Teillisten auf. *Beispiele:*

```
> splitBy0 6 ([1..10] ++ [12,10..0])
([1,2,3,4,5,6,6,4,2,0],[7,8,9,10,12,10,8])
```

```
> quicksort ([1..10] ++ [12,10..0])
[0,1,2,2,3,4,4,5,6,6,7,8,8,9,10,10,12]
```

- a) Implementieren Sie zuerst die Funktion `quicksort` gemäß des angegebenen Algorithmus, da Ihnen dies leichter fallen könnte. Verwenden Sie dabei bereits die Funktion `splitBy`, ohne diese zu implementieren. Falls Sie Ihr Programm bereits auf Typfehler prüfen möchten, können Sie folgende temporäre Definition für `splitBy` verwenden:

```
splitBy :: Int -> [Int] -> ([Int],[Int])
splitBy _ _ = error "Code not written yet."
```

- b) Implementieren Sie nun `splitBy` und testen Sie anschließend beide Funktionen.

Hinweise: Wem gar nichts einfällt, der kann einfach zwei List-Comprehensions verwenden. Das funktioniert, ist aber nicht besonders effizient, da die Eingabeliste dabei zwei Mal durchlaufen werden muss.

Schöner ist es, wenn die Eingabeliste nur einmal durchlaufen wird, z.B. durch eine Rekursion über die Eingabeliste mit anschließendem einordnen des aktuellen Elementes in die richtige Ergebnisliste. Da die Reihenfolge der Elemente in den beiden Ergebnislisten nicht spezifiziert wurde, läßt sich dann auch schnell eine endrekursive Version schreiben.

A2-3 List Comprehension Mia die modische Mathematikerin hat folgende Klamotten:

	Schwarz	Weiß	Pink	Flieder
Tops	3	3	3	4
Hosen	4	3	2	1
Schuhe	3	2	1	1

Mia möchte wissen, wie viele verschiedene Kombinationen Ihr damit zum Anziehen zur Verfügung stehen. Allerdings zieht Mia keine einfarbigen Outfits an; und die Farbe Ihrer Schuhe müssen zur Farbe Ihres Tops oder zu Ihrer Hose passen (gleiche Farbe). Als Mathematikerin kann Mia die Menge aller akzeptablen Kombinationen fix hinschreiben:

$$\left\{ (t, h, s) \mid t \in M_t, h \in M_h, s \in M_s, \text{ mit } f(t) \neq f(h) \text{ und } (f(t) = f(s) \text{ oder } f(h) = f(s)) \right\}$$

wobei M_t , M_h und M_s jeweils die Menge ihrer Tops, Hosen und Schuhe bezeichnen, und die Funktion f ein Kleidungsstück auf seine Farbe abbildet.

Typisch für Mathematiker ist diese Antwort zweifellos richtig, aber irgendwie nutzlos. Rechnen Sie die Anzahl der Kombinationen mit einer List-Comprehension fix aus!

Hinweis: Zur Vereinfachung modellieren wir Klamottenmengen durch Listen von ganzen Zahlen, wobei jeder Zahlwert für eine der möglichen Farben steht. Zum Beispiel kodieren wir M_s durch die Liste `[0,0,0,1,1,2,3]`. Listen können Elemente mehrfach enthalten; Schuhe gleicher Farbe brauchen wir ja hier nicht zu unterscheiden. Die Länge einer Liste berechnet man mit der Bibliotheksfunktion `length :: [a] -> Int`

H2-1 *Rekursion* (Datei H2-1.hs als Lösung abgeben)

Schreiben Sie eine Funktion `seekMaxMin :: [Double] -> (Double,Double)` welche gleichzeitig das kleinste und das größte Element einer Liste von Fließkommazahlen berechnet. Falls die Eingabeliste leer ist, so soll einfach das Paar `(0,0)` zurückgegeben werden.

```
> seekMaxMin [-2.5,0,3,-7.7,3.001,2.7]
(-7.7,3.001)
```

Um die volle Punktzahl zu erreichen, darf die Eingabeliste nur einmal durchgegangen werden.

H2-2 *Türme von Hanoi* (5 Punkte; Datei H2-2.hs als Lösung abgeben)

In der Vorlesung wurde eine rekursive Lösung des Puzzles “Türme von Hanoi” behandelt:

```
type Position = Int
type Move      = (Position,Position)
type Towers    = ([Int],[Int],[Int])

hanoi :: Int -> Position -> Position -> [Move]
```

Die behandelte rekursive Funktion `hanoi n i j` berechnet eine Liste von Spielzügen, um einen Turm der Größe `n` von Position `i` nach `j` zu versetzen.

- a) Schreiben Sie eine Funktion `move :: ([Move],Towers) -> ([Move],Towers)` welche eine Liste von Spielzügen und eine Spielkonfiguration bekommt, den *ersten* Spielzug überprüft und dann ausführt. Ein ungültiger Spielzug soll einen Ausnahmefehler durch einen Aufruf der Funktion `error :: String -> a` auslösen. *Beispiel:*

```
> hanoi 3 1 3
[(1,3),(1,2),(3,2),(1,3),(2,1),(2,3),(1,3)]
> move it, ([1,2,3],[],[])
([(1,2),(3,2),(1,3),(2,1),(2,3),(1,3)],([2,3],[],[1]))
> move it
([(3,2),(1,3),(2,1),(2,3),(1,3)],([3],[2],[1]))
> move it
([(1,3),(2,1),(2,3),(1,3)],([3],[1,2],[]))
> move ([1,4],[3],[1,2],[])
([],*** Exception: Ungültiger Zug!
> move ([1,2),(1,3)],([3,4],[1,2],[]))
([(2,3)],([4],*** Exception: Scheibe ist zu groß!)
```

Hinweis: GHCi bindet den Wert der letzten Berechnung immer an den Bezeichner `it`

```
Prelude> 1 + 3
4
Prelude> it * 2
8
Prelude> it * 2
16
```

- b) Schreiben Sie eine rekursive Funktion `game :: ([Move], Towers) -> Towers`, welche alle gegebenen Spielzüge ausführt, falls keine Fehler auftreten. Sie dürfen dabei `move` aus der vorangegangenen Teilaufgabe verwenden.

```
> game (hanoi 4 2 1, ([],[1..6],[]))
([1,2,3,4],[5,6],[])
> game (hanoi 4 2 1 ++ [(2,3)], ([],[1..5],[]))
([1,2,3,4],[],[5])
> game (hanoi 4 2 1 ++ [(2,3),(2,3)], ([],[1..6],[]))
([1,2,3,4],[],*** Exception: Scheibe ist zu groß!
```

Hinweis: In einigen Übungen sind deutlich weniger Teilnehmer erschienen als angemeldet, weshalb wir nächste Woche Anwesenheitslisten in den Übungen auslegen werden, um dies in UniworX sichtbar zu machen. Das Ziel ist die gleichmässige Auslastung aller Übungsgruppen. In folgenden Übungen sind offenbar noch Plätze frei: Di 16, 18; Mi 12, 14, 18; Fr 10, 12.

Abgabe: Lösungen zu den Hausaufgaben können bis Dienstag, den 5.05.2015, 11:00 Uhr mit UniworX abgegeben werden. Aufgrund des Klausurbonus müssen die Hausaufgaben von Ihnen alleine gelöst werden. Abschreiben bei den Hausaufgaben gilt als Betrug und kann zum Ausschluss von der Klausur zur Vorlesung führen.

Achten Sie darauf, dass Sie die geforderten Dateinamen einhalten. Mißachtung der Dateinamen führt zu Punktabzug! Für die Bearbeitung der Hausaufgaben sind bis auf weiteres die Verwendung von Funktionen der Standardbibliothek tabu, abgesehen von den Grundoperationen wie `(:)`, `(++)`, `(>)`, `(>=)`, `(<)`, `(<=)`, `div`, `mod`, `not`, `(&&)`, `(||)`, `max`, `min`, etc. Von Ihnen implementierte Funktionen aus vorangegangenen Aufgaben sind zulässig.