

Einführung in die funktionale Programmierung

Wintersemester 2007/2008

Aufgabenblatt Nr. 6

Abgabe: Dienstag 27. November 2007 **vor!** der Vorlesung

Aufgabe 1 (28 Punkte)

Der Datentyp `SnocList` zur Darstellungen von Listen mit „umgekehrten Cons“ (dargestellt durch den Infix-Konstruktor `:<`) sei in Haskell definiert als:

```
> data SnocList a = (SnocList a) :< a | Nil
> deriving(Show)
```

Z. B. wird die Liste `[1,2,3]` als `((Nil :< 3) :< 2) :< 1` dargestellt.

- a) Definieren Sie eine *Konstruktorklasse* `List`, welche die folgenden Operationen auf Listen-ähnlichen Datentypen überlädt:
- `emptyList`, die eine leere Liste erzeugt.
 - `mkList`, die ein Element sowie eine Liste erhält und daraus eine neue Liste erstellt, so dass das übergebene Element das Kopfelement und die übergebene Liste der Tail der Ergebnisliste ist.
 - `isEmptyList`, die für eine Liste prüft, ob diese leer ist.
 - `headList`, die das Kopfelement einer Liste liefert.
 - `tailList`, die den Tail einer Liste liefert.
 - `foldleft`, `foldright`, `foldleft1`, `foldright1`, die wie `foldl`, `foldr`, `foldl1` und `foldr1` wirken, jedoch für alle alle Listen-ähnlichen Datentypen definierbar sind.

Versuchen Sie möglichst viele default-Implementierungen der Klassenfunktionen anzugeben. (9 Punkte)

- b) Definieren Sie Instanzen der Klasse `List` *sowohl* für Haskells eingebaute Listen *als auch* für den Datentyp `SnocList`. (12 Punkte)
- c) Verwenden Sie `foldright`, um die beiden folgenden Funktionen zu definieren: (4 Punkte)
- `listToSnoc :: [a] -> SnocList a`, die eine Haskell-Liste in eine Liste vom Typ `SnocList` konvertiert.
 - `snocToList :: SnocList a -> [a]`, die eine Liste vom Typ `SnocList` in eine Haskell-Liste konvertiert.
- d) Definieren Sie eine Funktion `mapList`, die sich wie `map` verhält, aber für alle Instanzen der Klasse `List` verwendet werden kann. Welchen Typ hat `mapList`? (3 Punkte)

Aufgabe 2 (22 Punkte)

Der Datentyp `FehlerTest` sei definiert als:

```
data FehlerTest a = Fehler String | Ok a
  deriving Show
```

Er dient zur Darstellung eines erfolgreichen Resultats (`Ok a`) oder einer Fehlermeldung. Wir wollen den Datentypen benutzen, um Operationen sequentiell mithilfe der `do`-Notation auszuführen, und Fehler dabei abzufangen. Falls die Ausführung mehrerer Operationen erfolgreich ist, soll das Ergebnis `Ok a` sein, andernfalls soll `Fehler str` zurückgegeben werden, wobei `str` eine Fehlerbeschreibung ist.

- a) Implementieren Sie eine Instanz der Klasse `Monad` für den Datentyp `FehlerTest`, wobei das folgende Beispiel verdeutlicht, wie die Operationen `>>=`, `return` und `fail` zu implementieren sind.

Wir betrachten als Anwendungsbeispiel, die Funktion `tail'`, die zum Fehler führt, falls sie auf eine leeren Liste angewendet wird, und ansonsten `Ok xs` liefert, wobei `xs` der Tail der Liste ist:

```
> tail' :: [a] -> FehlerTest [a]
> tail' []      = fail "Fehler: tail auf einer leeren Liste"
> tail' (x:xs) = Ok xs
```

Die Sequenz

```
> do
>   x <- tail' [1,2,3]
>   y <- tail' x
>   return y
```

sollte als Ergebnis `Ok [3]` liefern, während

```
> do
>   x <- tail' [1]
>   y <- tail' x
>   z <- tail' y
>   return z
```

als Ergebnis `Fehler "Fehler: tail auf einer leeren Liste"` liefert. (12 Punkte)

- b) Überprüfen Sie, ob Ihre Implementierung die monadischen Gesetze erfüllt. (10 Punkte)