7. Aufgabenblatt zu Funktionale Programmierung vom 27.11.2006. Fällig: 05.12.2006 / 12.12.2006 (jeweils 15:00 Uhr)

Themen: *Typklassen und überladene Funktionen, Funktionale*

Für dieses Aufgabenblatt sollen Sie Haskell-Rechenvorschriften für die Lösung der unten angegebenen Aufgabenstellungen entwickeln und für die Abgabe in einer Datei namens `Aufgabe7.hs` ablegen. Sie sollen für die Lösung dieses Aufgabenblats also wieder ein "gewöhnliches" Haskell-Skript schreiben. Versehen Sie wieder wie auf den bisherigen Aufgabenblättern alle Funktionen, die Sie zur Lösung brauchen, mit ihren Typdeklarationen und kommentieren Sie Ihre Programme aussagekräftig. Benutzen Sie, wo sinnvoll, Hilfsfunktionen und Konstanten. Denken Sie bei der Kommentierung daran, dass Ihre Lösungen auch Grundlage für das Abgabegespräch am Semesterende sind.

Im einzelnen sollen Sie die im folgenden beschriebenen Problemstellungen bearbeiten.

1. In dieser Aufgabe greifen wir noch einmal die zweite Teilaufgabe von Aufgabenblatt 5 auf. An den Anfang stellen wir dabei das folgende Exzerpt aus Abschnitt 4.3.2 aus *Haskell 98: Language and Libraries – The Revised Report*, herausgegeben von Simon Peyton Jones, verlegt von Cambridge University Press, erschienen 2003:

   "The type $(T\ u_1 \ldots u_k)$ must take the form of a type constructor T applied to simple type variables $u_1, \ldots, u_k$; furthermore, T must not be a type synonym, and the u_i must all be distinct.

   This prohibits instance declarations such as

   ```
   instance C (a,a) where ...
   instance C (Int,a) where ...
   instance C [[a]] where ...
   ```

   The declarations may contain..."

   Dies zeigt, dass die zweite Teilaufgabe von Aufgabenblatt 5 in ihrer ursprünglichen Formulierung über den aktuellen Haskell-Report hinausgeht. Wir ersetzen deshalb die dort angesprochenen Typsynonyme durch die folgenden algebraischen Datentypen:

   ```
   data Graph    = Gr [[Bool]]
   data Relation = Re [[Int]]
   ```

   Lösen Sie jetzt für diese Datentypen die zweite Teilaufgabe von Aufgabenblatt 5 ihrem Sinn gemäß.

2. "Teile und Herrsche" beschreibt ein wichtiges Organisationsprinzip von Algorithmen. Wenn wir zwei Typen `p` und `s` annehmen, deren Werte Probleme bzw. Lösungen dieser Probleme beschreiben, läßt sich das diesem Prinzip zugrundeliegende Organisationsschema in einem Funktional `divAndConquer` kondensieren:

   ```
   divAndConquer :: (p -> Bool) -> (p -> s) -> (p -> [p]) -> (p -> [s] -> s) -> p -> s
   divAndConquer ind solve divide combine initProblem
       = dac initProblem
       where dac problem
                 | ind problem = solve problem
                 | otherwise   = combine problem (map dac (divide problem))
   ```

   Dabei stützt sich das Funktional `divAndConquer` auf folgende Argumentfunktionen:

   - `ind :: p -> Bool`: ...liefert `True`, wenn die als Argument übergebene Probleminstanz nicht mehr teilbar ist, `False` sonst.
   - `solve :: p -> s`: ...liefert die Lösung zu einer nicht mehr teilbaren Probleminstanz.
   - `divide :: p -> [p]`: ...liefert eine Liste von Instanzen von Teilproblemen, wenn die als Argument übergebene Probleminstanz teilbar ist.
   - `combine :: p -> [s] -> s`: ...konstruiert aus der Instanz des Ausgangsproblems und den Lösungen seiner Teilprobleme die Lösung des Ausgangsproblems.

(a) Zeigen Sie, dass die in der Vorlesung besprochene Funktion `quickSort` dem "Teile und Herrsche"-Prinzip folgt und sich als Spezialisierung des Funktionals `divAndConquer` ergibt. Geben Sie dazu geeignete Implementierungen der Funktionen `ind`, `solve`, `divide` und `combine` (als lokale Rechenvorschriften von `quickSort`) an, so dass `quickSort` in Ihrer Abgabedatei insgesamt wie folgt implementiert ist:

```
quickSort :: Ord a => [a] -> [a]
quickSort = divAndConquer ind solve divide combine
```

(b) Betrachten Sie noch einmal die Folge der Fibonacci-Zahlen $f_0, f_1, \ldots$ definiert durch

$$f_0 = 0, f_1 = 1 \quad \text{und} \quad f_n = f_{n-1} + f_{n-2} \quad \text{für alle } n \geq 2$$

und zeigen Sie, dass sich die Rechenvorschrift `fib :: Integer -> Integer` ebenfalls als Spezialisierung des Funktionals `divAndConquer` angeben läßt, indem Sie wieder geeignete Implementierungen der Funktionen `ind`, `solve`, `divide` und `combine`, dieses Mal als lokale Rechenvorschriften von `fib` angeben, so dass `fib` in Ihrer Abgabedatei insgesamt wie folgt implementiert ist:

```
fib :: Integer -> Integer
fib = divAndConquer ind solve divide combine
```

3. Erweitern Sie die Implementierungen der Rechenvorschriften `quickSort` und `fib` aus der vorherigen Teilaufgabe um eine Fehlerbehandlung im Sinne von Variante 3 aus Vorlesungsteil 8 vom 23.11.2006. Schreiben Sie dazu neue Haskell-Rechenvorschriften `quickSortE` und `fibE` mit den Signaturen `quickSortE :: Ord a => [a] -> Maybe [a]` und `fibE :: Integer -> Maybe Integer`. Der Aufruf `fibE n` soll dabei den Wert `Nothing` liefern, wenn das Argument `n` negativ ist, ansonsten den Wert `Just f`, wobei `f` der entsprechende Wert der Fibonacci-Funktion ist. Der Aufruf der Funktion `quickSortE` soll den Wert `Nothing` liefern, wenn das Argument die vorgegebene Sortiergrenze 5 übersteigt, d.h. angewendet auf Listen `l` mit 6 oder mehr Elementen liefert der Aufruf `quickSortE l` das Resultat `Nothing`, ansonsten den Wert `Just s`, wobei `s` die Elemente von `l` sortiert enthält. Im Nichtfehlerfall sollen beide Funktionen `quickSortE` und `fibE` wieder mithilfe des Funktionals `divAndConquer` definiert werden.

Hinweis:

- Verwenden Sie *keine* Module. Wenn Sie Funktionen wiederverwenden möchten, kopieren Sie diese in die Abgabedatei `Aufgabe7.hs`. Andere als diese Datei werden vom Abgabeskript ignoriert.
- Bitte testen Sie Ihre Lösungen mit der aktuellen Version von Hugs auf der b1: `/usr/local/bin/hugs`