9. Aufgabenblatt zu Funktionale Programmierung vom 07.12.2010.
Fällig: Di, 14.12.2010 / Di, 21.12.2010 (jeweils 15:00 Uhr)

Themen: *Funktionen höherer Ordnung, Teile und Herrsche, Knobeleien*

Für dieses Aufgabenblatt sollen Sie Haskell-Rechenvorschriften für die Lösung der unten angegebenen Aufgabenstellungen entwickeln und für die Abgabe in einer Datei namens `Aufgabe9.hs` ablegen. 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.

Im einzelnen sollen Sie die im folgenden beschriebenen Problemstellungen bearbeiten.

1. "Teile und Herrsche" umschreibt 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` kapseln:

```
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. Der Name "ind" erinnert dabei an "indivisible".
   - `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.

   Zeigen Sie, dass sich die algorithmische Lösung folgender Probleme auf das "Teile und Herrsche"-Prinzip abstützen und sich als passende Spezialisierung des Funktionals `divAndConquer` formulieren lässt.

   (a) `maximum`: bestimme das größte Element in einer nichtleeren Liste ganzer Zahlen.

   (b) `primes`: bestimme aufsteigend geordnet die Menge der Primzahlen p mit $1 \leq m \leq p \leq n$, m, n ganzzahlig (Hinweis: 1 ist keine Primzahl; 2 ist die kleinste Primzahl).

   (c) `nodes`: bestimme die Anzahl der Knoten in einem Binärbaum.

   Geben Sie dazu problemspezifische Implementierungen der Funktionen `ind`, `solve`, `divide` und `combine` an, so dass `maximum`, `primes` und `nodes` in Ihrer Abgabedatei insgesamt wie folgt implementiert sind:

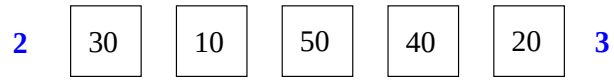
```
data Tree a = Nil |
              Node a (Tree a) (Tree a) deriving (Eq,Show)

type Von = Integer
type Bis = Integer

maximum :: [Integer] -> Integer
maximum = divAndConquer indM solveM divideM combineM

primes :: (Von,Bis) -> [Integer]
primes = divAndConquer indP solveP divideP combineP
```

```
nodes :: (Tree a) -> Integer
nodes = divAndConquer indN solveN divideN combineN
```

Dabei sollen die Funktionen `indX`, `solveX`, `divideX`, `combineX` für `X`$\in$ {`M`,`P`,`N`} global definiert sein, also nicht lokal in einer `let`- oder `where`-Klausel.

2. *Hochhauszeile* ist eine vereinfachte Variante der Knobelei *Skyline*. In einer Hochhauszeile der Länge n, $n > 1$, steht je ein Hochhaus mit genau 10, 20, 30,...,$n * 10$ Etagen. Die Zahlen am Rand der Hochhauszeile geben an, wieviele Hochhäuser man sieht, wenn man von diesem Punkt in die Hochhauszeile hineinschaut. Dabei gilt: Höhere Hochhäuser verdecken niedrigere.

   Die folgende Abbildung zeigt ein Beispiel für eine Hochhauszeile der Länge 5 (bzw. Länge $(5 + 2)$ einschließlich Sichtbarkeitsinformation).



   Wir modellieren Hochhauszeilen und Sichtbarkeitsinformation durch folgenden Datentyp:

```
type Skyscraperline = [Integer]
```

   Zusätzlich führen wir folgende Typsynonyme ein:

```
type Length       = Integer
type VisFromLeft  = Integer
type VisFromRight = Integer
```

   In einer Liste der Länge $(n + 2)$ beschreiben der 1-te und $(n + 2)$-te Wert der Liste die Anzahl der sichtbaren Hochhäuser, wenn man von links bzw. rechts aus in die Hochhauszeile hineinschaut. Das 1-te und $(n+2)$-te Feld der Liste bezeichnen wir als *Sichtbarkeitsfelder*, die die Sichtbarkeitsinformation tragen. Als *Hochhauszeile* bezeichnen wir die innere Liste der Länge n, auf denen Hochhäuser stehen ohne die Sichtbarkeitsfelder.

   (a) Schreiben Sie eine Haskell-Funktion `isValid` mit der Signatur `isValid :: Skyscraperline -> Bool`. Angewendet auf einen Wert vom Typ `Skyscraperline` überprüft `isValid`, ob das Argument eine *gültige* Hochhauszeile einschließlich Sichtbarkeitsinformation beschreibt, d.h., das Argument eine Liste der Länge $(n + 2)$ ist, auf den inneren n Elementen der Liste genau je ein Hochhaus mit 10, 20, 30,...,$n * 10$ Etagen steht und die Sichtbarkeitsangaben im 1-ten und $n + 2$-ten Listenelement korrekt sind. In diesem Fall ist das Resultat des Aufrufs `True`, sonst `False`.

   (b) Schreiben Sie eine Haskell-Funktion `computeVisibility` mit der Signatur `computeVisibility :: Skyscraperline -> Skyscraperline`, die angewendet auf eine Liste der Länge n, in der auf jedem Feld je ein Hochhaus mit genau 10, 20, 30,...,$n * 10$ Etagen steht, diese Hochhauszeile erweitert um die beiden Sichtbarkeitsfelder mit korrekter Sichtbarkeitsinformation zurückliefert. Sie können davon ausgehen, dass die Funktion nur mit solchen Listen aufgerufen wird.

   (c) Schreiben Sie eine Haskell-Funktion `buildSkyscrapers` mit der Signatur `buildSkyscrapers :: Length -> VisFromLeft -> VisFromRight -> Maybe Skyscraperline`, die angewendet auf die Länge der Hochhauszeile, die Anzahl der von links und von rechts sichtbaren Hochhäuser eine passende Hochhauszeile mit entsprechenden Sichtbarkeitsfeldern zurückliefert, also einen Wert vom Typ `Just Skyscraperline`, wenn möglich; ansonsten den Wert `Nothing`. Ist die Hochhauszeile durch Länge und Sichtbarkeitsinformation nicht eindeutig festgelegt, reicht es, wenn ihre Funktion eine den Anforderungen genügende Bebauung zurückliefert.

   (d) Schreiben Sie eine Haskell-Funktion `noOfSkyscraperLines` mit der Signatur `noOfSkyscraperLines :: Length -> VisFromLeft -> VisFromRight -> Integer`, die die Anzahl zulässiger Hochhausbebauungen für bestimmte Hochhauszeilenlänge und Sichtbarkeitsinformation berechnet.

(e) Schreiben Sie eine Haskell-Funktion `allSkyscraperLines` mit der Signatur `allSkyscraperLines :: Length -> VisFromLeft -> VisFromRight -> [Skyscraperline]`, die alle zulässigen Hochhausbebauungen einschließlich Sichtbarkeitsinformationen für gegebene Hochhauszeilenlänge und Sichtbarkeitsinformation zurückliefert. Dabei seien die verschiedenen Permutationen in der Resultatliste lexikographisch aufsteigend sortiert.

Sie können davon ausgehen, dass die Funktionen aus c), d) und e) nur für Hochhauszeilen bis zur Länge 5 einschließlich getestet werden.

Hinweis:

- Verwenden Sie *keine* Module, um frühere eigene Funktionen wiederzuwenden. Wenn Sie eigene Funktionen wiederverwenden möchten, kopieren Sie diese in die Abgabedatei `Aufgabe9.hs`. Andere Dateien als diese werden vom Abgabeskript ignoriert.

Haskell Live

Der nächste *Haskell Live*-Termin findet am Freitag, den 10.12.2010, statt.