5. Aufgabenblatt zu Funktionale Programmierung vom 23.11.2005, fällig: 30.11.2005 (12:00 Uhr)

Themen: *Funktionen auf Listen, Bäumen und Graphen*

Für dieses Aufgabenblatt sollen Sie Haskell-Rechenvorschriften zur Lösung der unten angegebenen Aufgabenstellungen entwickeln und für die Abgabe in einer Datei namens `Aufgabe5.lhs` ablegen. Wie schon auf Aufgabenblatt 2 ist dieses Mal wieder ein "**literate Script**" gefordert.

Versehen Sie 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 folgende Problemstellungen bearbeiten:

1. Ein Graph $G = (N, E)$ ist ein Paar bestehend aus einer Menge von Knoten N und einer Menge von Kanten E mit $E \subseteq N \times N$. Dabei können die Kanten in G gerichtet oder ungerichtet sein. Entsprechend spricht man von gerichteten und ungerichteten Graphen.

   In der Folge wollen wir ungerichtete Graphen betrachten, die wir durch eine sog. *Adjazenzliste* darstellen wollen. Die Adjazenzliste enthält für jeden Knoten des Graphen einen Eintrag, der aus dem Knoten selbst und einer Liste seiner (unmittelbar) benachbarten Knoten besteht. Benennen wir die Knoten des Graphen durch ganze Zahlen, können wir die Adjazenzliste eines Graphen durch Werte des folgenden Haskell-Typs repräsentieren:

   ```
   type Graph = [(Int,[Int])]
   ```

   - Die Adjazenzmatrix eines Graphen G heißt *wohlgeformt* genau dann, wenn (1) die ersten Komponenten der Einträge der Adjazenzmatrix paarweise verschieden sind, (2) kein Knoten mehrfach in der Adjazenzliste eines Knotens auftaucht, (3) jeder Knoten, der in der Adjazenzliste eines Knotens genannt ist, einen eigenen Eintrag besitzt, d.h. als erste Komponente eines Eintrags auftritt und (4) ein in der Adjazenzliste eines Knotens n genannter Knoten m den Knoten n seinerseits in seiner eigenen Adjazenzliste aufführt und umgekehrt. Schreiben Sie eine Haskell-Rechenvorschrift `isWellformed :: Graph -> Bool`, die überprüft, ob ein vorgelegter Graph im obigen Sinn wohlgeformt ist. In diesem Fall ist das Resultat der Wahrheitswertfunktion `isWellformed True`, sonst `False`.
   - Ein wohlgeformter Graph G heißt *zusammenhängend* genau dann, wenn je zwei paarweise verschiedene Knoten des Graphen durch eine Kantenfolge verbunden sind. Schreiben Sie eine Haskell-Rechenvorschrift `isConnected :: Graph -> Bool`, die überprüft, ob ein vorgelegter wohlgeformter Graph im obigen Sinn zusammenhängend ist. In diesem Fall ist das Resultat der Wahrheitswertfunktion `isConnected True`, sonst `False`, ebenso wenn der vorgelegte Graph nicht wohlgeformt ist.

2. Viele Werkzeuge zur Zeichenreihenbearbeitung erlauben es, Zeichenreihen, die eine gewisse Regelmäßigkeit aufweisen, durch Muster zu beschreiben als alle Zeichenreihen einzeln anzugeben. Eine solche Funktionalität wollen wir in dieser Aufgabe nachbilden. Schreiben Sie dazu eine Haskell-Rechenvorschrift `patternMatching` mit der Signatur `patternMatching :: Muster -> Text -> Text`, wobei `Muster` und `Text` folgendermaßen deklariert sind: `type Muster = String` und `type Text = [String]`. In einem Element vom Typ `Muster` haben Vorkommen der Zeichen `*` und `?` eine besondere Bedeutung. Das Zeichen `*` steht als Platzhalter für eine beliebige (einschließlich der leeren Zeichenreihe), das Zeichen `?` als Platzhalter für genau ein Zeichen, das aber beliebig ist. Angewendet auf ein Muster und einen Text liefert die Funktion `patternMatching` als Resultat einen Text, der in derselben relativen Reihenfolge wie im Argumenttext genau die Zeichenreihen aus dem Argumenttext enthält, die dem gesuchten Muster entsprechen.

3. In dieser Aufgabe betrachten wir folgende algebraische Datentypen zur Darstellung von Binärbäumen:

```
data Tree = Leaf Int |
            Node Int Tree Tree

data STree = SNil |
             SLeaf Int |
             SNode Int STree STree
```

   Dabei repräsentieren Werte vom Typ `Tree` Binärbäume, deren Knoten und Blätter in beliebiger Weise mit ganzen Zahlen benannt sind, wohingegen Werte vom Typ `STree` Suchbäume über ganzen Zahlen repräsentieren. Die Benennungen in einem Suchbaum nennen wir in der Folge *Schlüssel*. Ein Baum B ist ein *Suchbaum* genau dann, wenn für jeden Knoten in B gilt, dass alle Schlüssel im linken Teilbaum einen kleineren Wert und im rechten Teilbaum einen größeren Wert haben als im betrachteten Knoten. Schreiben Sie eine Haskell-Rechenvorschrift `transformTree` mit der Signatur `transformTree :: Tree -> STree`, die den Argumentbaum in Infixordnung durchläuft und aus den gefundenen Marken einen Suchbaum aufbaut, in den die Marken des Argumentbaums in der Reihenfolge des Auffindens eingefügt werden. Duplikate von im Argumentbaum möglicherweise mehrfach vorkommender Marken werden dabei nicht berücksichtigt.

   Hinweis: In Infixordnung werden zunächst die Knoten des linken Teilbaums in Infixordnung besucht, danach der Wurzelknoten, zum Schluss die Knoten des rechten Teilbaums in Infixordnung.

Denken Sie bitte daran, dass Sie für die Lösung dieses Aufgabenblatts ein "literate" Haskell-Skript schreiben sollen!