6. Aufgabenblatt zu Funktionale Programmierung vom 30.11.2005, fällig: 07.12.2005 / 14.12.2005 (jeweils 12:00 Uhr)

Themen: *Polymorphe Funktionen auf Listen und Graphen*

Für dieses Aufgabenblatt sollen Sie Haskell-Rechenvorschriften zur Lösung der nachstehend beschriebenen Aufgabenstellungen entwickeln und für die Abgabe in einer Datei namens `Aufgabe6.hs` ablegen. Sie sollen also wieder ein Standard-Haskellskript schreiben. 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. In dieser Aufgabe betrachten wir noch einmal die auf Aufgabenblatt 5 eingeführten Graphen vom Typ

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

   Ein wohlgeformter Graph G heißt *kreisfrei* genau dann, wenn von keinem Knoten eine Folge paarweise verschiedener Kanten ausgeht, die wieder zu diesem Knoten zurückführt. Schreiben Sie eine Haskell-Rechenvorschrift `isAcyclic :: Graph -> Bool`, die überprüft, ob ein vorgelegter wohlgeformter Graph im obigen Sinn kreisfrei ist. In diesem Fall ist das Resultat der Wahrheitswertfunktion `isAcyclic True`, sonst `False`, ebenso wenn der vorgelegte Graph nicht wohlgeformt ist.

2. In dieser Aufgabe betrachten wir eine Variante der Graphen aus der vorigen Aufgabe, bei der wir die Kanten gewichten. Wir repräsentieren diese Graphen durch folgenden Datentyp in Haskell:

   ```
   newtype WGraph a = WG [(a,[(a,Int)])] deriving Eq
   ```

   Zusätzlich betrachten wir folgenden algebraischen Datentyp:

   ```
   data Result a = Mc Int   |
                   Nwf      |
                   Np a deriving Eq
   ```

   Unter den *Kosten einer Kantenfolge* kf in einem wohlgeformten gewichteten Graphen verstehen wir die Summe der Gewichte der Kanten in kf. Der Begriff der Wohlgeformtheit überträgt sich dabei in natürlicher Weise von den bisher betrachteten ungewichteten Graphen auf die jetzt betrachteten gewichteten Graphen. Zusätzlich wird gefordert, dass alle Kantengewichte nichtnegativ sind und in den Adjazenzlisten zweier benachbarter Knoten jeweils das gleiche Gewicht für die sie verbindende Kante angegeben ist. Schreiben Sie eine Haskell-Rechenvorschrift `minCosts` mit der Signatur `minCosts :: Eq a => WGraph a -> a -> a -> Result String`, die angewendet auf einen wohlgeformten gewichteten Graphen G und zwei Knoten m und n in G die Kosten

einer kostenminimalen Kantenfolge von m nach n bestimmt, so es eine solche gibt. In diesem Fall ist das Resultat der Funktion `minCosts` der Wert `Mc k`, wobei `k :: Int` die Kosten einer solchen kostenminimalen Kantenfolge sind. Die Konstruktorbezeichnung `Mc` erinnert dabei an "minimum costs". Ist der Argumentgraph nicht wohlgeformt, ist das Resultat der Funktionsanwendung der Wert `Nwf`, wobei dieser Konstruktorname an "not well-formed" erinnert; ist der Graph wohlgeformt, die Knoten m und n aus G, aber nicht durch eine Kantenfolge in G verbunden, dann ist das Resultat der Wert `Np "Nicht verbunden"`, ansonsten der Wert `Np "Knoten nicht im Graph"`, wobei die Konstruktorbezeichnung an "no path" erinnert.

3. In dieser Aufgabe führen wir eine Variante der Graphen aus der ersten Aufgabe ein, die wie die Graphen aus der zweiten Aufgabe gewichtete Graphen darzustellen erlauben. Wir repräsentieren diese Graphen durch folgenden Haskell-Typ:

   ```
   type SWGraph a = [(a,[(a,Int)])]
   ```

   Für Graphen vom Typ `SWGraph`, wobei das `S` an Typsynomym erinnern soll, gelten die Wohlgeformheitsregeln für Graphen der Typen `Graph` und `WGraph a` analog. Schreiben Sie zwei Konvertierungsfunktionen `conv1` und `conv2` mit den Signaturen `conv1 :: Eq a => SWGraph a -> WGraph a` und `conv2 :: Eq a => WGraph a -> SWGraph a`. Dabei soll die Funktion `conv1` angewandt auf einen wohlgeformten Graphen vom Typ `SWGraph a` diesen in einen wohlgeformten Graphen vom Typ `WGraph a` konvertieren, wobei die relativen Reihenfolgen der Knoten und auch die Reihenfolgen innnerhalb der zugehörigen Adjazenzlisten erhalten bleiben sollen. Ist der Argumentgraph nicht wohlgeformt, soll `WG []`. das Resultat sein. Entsprechend soll die Funktion `conv2` angewandt auf einen wohlgeformten Graphen vom Typ `WGraph a` diesen in einen wohlgeformten Graphen vom Typ `SWGraph a` konvertieren, wobei auch hier alle relativen Reihenfolgen erhalten bleiben sollen. Ist der Argumentgraph nicht wohlgeformt, soll die Funktion `conv2` das Resultat `[]` liefern.

4. In dieser Aufgabe betrachten wir noch einmal die Aufgabenstellung zur Zeichenreihenbearbeitung von Aufgabenblatt 5 und erweitern sie wie folgt. Zusätzlich zu den Zeichen `*` und `?` (mit derselben Bedeutung wie auf Aufgabenblatt 5) haben jetzt auch in eckige Klammern eingeschlossene Zeichenreihen in Elementen vom Typ Muster eine besondere Bedeutung: Sie stehen für das optionale (also das einmalige oder nullmalige) Vorkommen der in den eckigen Klammern eingeschlossenen Zeichenreihe in einer Zeichenreihe. Eckige Klammern ohne "passende" öffnende oder schließende Klammer haben keine spezielle Bedeutung, ebenso nicht Vorkommen von `*`, `?`, `[` innerhalb eines Paars passender eckiger Klammern. Dabei ist passend wie folgt definiert: Zu einer öffnenden eckigen Klammer ist die erste darauf folgende schließende eckige Klammer *passend*. Schreiben Sie jetzt eine Haskell-Rechenvorschrift `xPatternMatching` mit der Signatur `xPatternMatching :: Muster -> Text -> Text`, wobei `Muster` und `Text` wieder wie folgt deklariert sind: `type Muster = String` und `type Text = [String]`. Angewendet auf ein Muster und einen Text liefert die Funktion `xPatternMatching` als Resultat einen Text, der genau die Zeichenreihen

aus dem Argumenttext enthält, die dem gesuchten Muster entsprechen. Im Unterschied zur Funktion `patternMatching` von Aufgabenblatt 5 sollen die Zeichenreihen des Resultattexts dieses Mal in aufsteigender Länge sortiert sein. Enthält der Resultattext Zeichenreihen gleicher Länge, so sollen diese (innerhalb der übergeordneten Längensortierung) in derselben relativen Reihenfolge wie im Argumenttext stehen.

Hinweise:

- Verwenden Sie *keine* Module. Wenn Sie Funktionen wiederverwenden möchten, kopieren Sie diese in die Abgabedatei `Aufgabe6.hs`. Andere als diese Datei werden vom Abgabeskript ignoriert.
- Die Bedeutung des Schlüsselworts `deriving` in Typdeklarationen wird in der Vorlesung vom 01.12.2005 besprochen. Sie finden aber auch in jedem guten Buch über Haskell eine entsprechende Erklärung.