5. Aufgabenblatt zu Funktionale Programmierung vom 09.11.2010.
Fällig: 16.11.2010 / 23.11.2010 (jeweils 15:00 Uhr)

Themen: *Funktionen auf algebraischen Datentypen*

Für dieses Aufgabenblatt sollen Sie Haskell-Rechenvorschriften für die Lösung der unten angegebenen Aufgabenstellungen entwickeln und für die Abgabe in einer Datei namens `Aufgabe5.hs` ablegen. Sie sollen für die Lösung dieses Aufgabenblatts 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.

Im einzelnen sollen Sie die im folgenden beschriebenen Problemstellungen bearbeiten.

Wir betrachten für dieses Aufgabenblatt gerichtete Graphen mit kostenbenannten Kanten. Dabei nehmen wir an, dass es zwischen je zwei Knoten höchstens eine Kante je Richtung gibt, dass alle Knoten beginnend mit 0 ohne Lücken fortlaufend durchnummeriert sind und dass alle Kantenkosten echt größer als 0 sind. Knoten ohne eingehende und ausgehende Kanten nennen wir *isoliert*. Graphen, die diese Bedingungen erfüllen, nennen wir *gültig*. Für gültige Graphen gibt es unterschiedliche Repräsentationsformalismen, darunter in Form von *Adjazenzlisten*, *Adjazenzmatrizen* und *Kantenlisten*. In Haskell können diese Varianten durch die algebraischen Datentypen `ALgraph`, `AMgraph` und `ELgraph` implementiert werden.

```
type Cost           = Integer
type Vertex         = Integer
type MaxVertexNo    = Integer
type Edge           = (Vertex,Cost,Vertex)
type Row            = [Integer]

data ALgraph        = ALg [(Vertex,[(Vertex,Cost)])] deriving (Eq,Show)
data ELgraph        = ELg MaxVertexNo [Edge] deriving (Eq,Show)
data AMgraph        = AMg [Row] deriving (Eq,Show)

type Inp            = (MaxVertexNo,[(Vertex,Vertex,Cost)])
```

Adjazenzlistengraphen enthalten für jeden Knoten des Graphen genau einen Eintrag mit der zugehörigen (möglicherweise leeren) Nachfolgerliste des Knoten. Kantenlistengraphen repräsentieren einen Graphen durch eine listenförmige Aufzählung aller seiner jeweils mit ihren Kosten benannten gerichteten Kanten. Da die Knoten des Graphen so nur implizit spezifiziert sind, enthalten Kantenlistengraphen zusätzlich die Information über die Anzahl der Knoten im Graphen. Somit ist implizit auch die Information über isolierte Knoten abgebildet. Adjazenzmatrizengraphen schließlich ordnen jedem Knoten v, $0 \leq v \leq$ `MaxVertexNo` die v-te Zeilen- und Spaltennummer der Adjazenzmatrix zu. Gibt es eine gerichtete Kante vom Knoten i zum Knoten j, so enthält das j-te Element der i-ten Zeile den Wert `c`, wobei `c` > 0 die Kosten der entsprechenden Kante angibt. Gibt es eine solche Kante nicht, so enthält das entsprechende Element den Wert 0.

1. Schreiben Sie eine Testfunktion `isValid :: Inp -> Bool`, die wahr ist, wenn die Eingabeliste einen gültigen Graphen beschreibt, d.h. Anfangs- und Endpunkt aller Kanten mit Werten zwischen 0 und `MaxVertexNo`, `MaxVertexNo`≥ 0, benannt sind, es höchstens eine Kante zwischen zwei Knoten gibt, und alle Kantenkosten echt größer als 0 sind, ansonsten falsch.

2. Schreiben Sie folgende Konversionsfunktionen, die ihre Argumente in bedeutungsgleiche Ausgaben entsprechenden Typs konvertieren. Das Resultat ist i.a. nicht eindeutig festgelegt. Es reicht, wenn Ihre Konversionsfunktionen jeweils eine gültige Repräsentation bestimmen.

   (a) `inp2el :: Inp -> ELgraph`

   (b) `al2am :: ALgraph -> AMgraph`

   (c) `al2el :: ALgraph -> ELgraph`

   (d) `am2al :: AMgraph -> ALgraph`

(e) `am2el :: AMgraph -> ELgraph`

(f) `el2al :: ELgraph -> ALgraph`

(g) `el2am :: ELgraph -> AMgraph`

Überlegen Sie sich, wie sie Implementierung einiger Konversionsfunktionen auf andere abstützen können, um sich Implementierungsarbeit zu sparen.

3. Schreiben Sie Haskell-Rechenvorschriften für die folgenden Aufgaben. Überlegen Sie sich jeweils, ob möglicherweise eine der Repräsentationsformen für die Implementierung besonders geeignet ist und wandeln Sie dann Argumente, die in weniger geeigneter Form vorliegen ggf. vorher um.

   (a) `isNeighbourOf :: ELgraph -> Vertex -> Vertex -> Bool`

   (b) `allNeighboursOf :: ELgraph -> Vertex -> [Vertex]`

   (c) `numberOfEdges :: AMgraph -> Integer`

   (d) `isOnCycle :: ALgraph -> Vertex -> Cost -> Bool`

   Der Funktionswert von `isNeighbourOf g v1 v2` ist `True`, falls es in `g` eine gerichtete Kante von `v1` nach `v2` gibt, sonst `False`. Der Funktionswert `allNeighboursOf g v` ist eine aufsteigend sortierte (möglicherweise leere) Liste aller Nachbarknoten von `v`, d.h. die Menge aller Knoten, die durch eine gerichtete Kante von `v` aus erreichbar sind. Ist die Nachbarknotenmenge leer oder `v` kein Knoten aus `g`, so ist die Resultatliste leer. Der Funktionswert `numberOfEdges g` ist die Anzahl der Kanten in `g`. Der Funktionswert `isOnCycle g v c` ist `True`, falls Knoten `v` im Graphen `g` auf einem Kreis mit Kosten kleiner oder gleich `c`, `c`> 0, liegt, d.h. ob `v` Anfangs- und Endpunkt eines Pfades in `g` ist, dessen aufsummierte Kantenkosten den Wert `c` nicht übersteigen. Gibt es einen solchen Kreis nicht oder ist `v` nicht in `g`, so ist der Funktionswert `False`.

Hinweis: Keine Module importieren!

Wenn Sie zur Lösung einzelne Funktionen früherer Lösungen wiederverwenden möchten, so kopieren Sie diese unbedingt explizit in Ihre neue Programmdatei ein. Importieren schlägt im Rahmen der automatischen Programmauswertung fehl. Es wird nicht nachgebildet. Deshalb: Wiederverwendung ja, aber durch kopieren, nicht durch importieren!

Haskell Live

An einem der kommenden *Haskell Live*-Termine, der nächste ist am Freitag, den 12.11.2010, werden wir uns u.a. mit der Aufgabe *City-Maut* beschäftigen.

City-Maut

Viele Städte überlegen die Einführung einer City-Maut, um die Verkehrsströme kontrollieren und besser steuern zu können. Für die Einführung einer City-Maut sind verschiedene Modelle denkbar. In unserem Modell liegt ein besonderer Schwerpunkt auf innerstädtischen Nadelöhren. Unter einem *Nadelöhr* verstehen wir eine Verkehrsstelle, die auf dem Weg von einem Stadtteil A zu einem Stadtteil B in der Stadt passiert werden muss, für den es also keine Umfahrung gibt. Um den Verkehr an diesen Nadelöhren zu beeinflussen, sollen an genau diesen Stellen Mautstationen eingerichtet und Mobilitätsgebühren eingehoben werden.

In einer Stadt mit den Stadtteilen A, B, C, D, E und F und den sieben in beiden Richtungen befahrbaren Routen B–C, A–B, C–A, D–C, D–E, E–F und F–C führt jede Fahrt von Stadtteil A in Stadtteil E durch Stadtteil C. C ist also ein Nadelöhr und muss demnach mit einer Mautstation versehen werden.

Schreiben Sie ein Programm in Haskell oder in einer anderen Programmiersprache Ihrer Wahl, das für eine gegebene Stadt und darin vorgegebene Routen Anzahl und Namen aller Nadelöhre bestimmt.

Der Einfachheit halber gehen wir davon aus, dass anstelle von Stadtteilnamen von 1 beginnende fortlaufende Bezirksnummern verwendet werden. Der Stadt- und Routenplan wird dabei in Form eines Tupels zur Verfügung gestellt, das die Anzahl der Bezirke angibt und die möglichen jeweils in beiden Richtungen befahrbaren direkten Routen von Bezirk zu Bezirk innerhalb der Stadt. In Haskell könnte dies durch einen Wert des Datentyps `CityMap` realisiert werden:

```
type Bezirk     = Integer
type AnzBezirke = Integer
type Route      = (Bezirk,Bezirk)
newtype CityMap = CM (AnzBezirke,[Route])
```

Gültige Stadt- und Routenpläne müssen offenbar bestimmten Wohlgeformtheitsanforderungen genügen. Überlegen Sie sich, welche das sind und wie sie überprüft werden können, so dass Ihr Nadelöhrsuchprogramm nur auf wohlgeformte Stadt- und Routenpläne angewendet wird.