8. Aufgabenblatt zu Funktionale Programmierung vom 05.12.2012.
Fällig: 12.12.2012 / 09.01.2013 (jeweils 15:00 Uhr)

Themen: *Funktionen auf Zahlen, Zeichenreihen und Graphen*

Für dieses Aufgabenblatt sollen Sie Haskell-Rechenvorschriften für die Lösung der unten angegebenen Aufgabenstellungen entwickeln und für die Abgabe in einer Datei namens `Aufgabe8.hs` ablegen. Sie sollen für die Lösung dieses Aufgabenblatts also 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.

1. Wir betrachten folgende Zahlenspielerei. Wir addieren zu einer positiven ganzen Zahl n mit Ziffernfolge $z_1 z_2 \ldots z_k$, $z_i \in \{0, 1, \ldots, 9\}$, $z_1 \neq 0$, die Zahl m mit umgekehrter Ziffernfolge $z_k z_{k-1} \ldots z_1$. Ist die Ziffernfolge der Zahl $m + n$ ein Palindrom, d.h. stimmt die Ziffernfolge von $m + n$ von links nach rechts gelesen mit derjenigen von rechts nach links gelesen überein, so endet die Spielerei. Ansonsten fahren wir mit der Zahl $m + n$ in der gleichen Weise fort.

   Für viele positive ganze Zahlen als Startwert endet diese Spielerei nach wenigen Wiederholungen in einem Palindrom; für einige, etwa 196, möglicherweise nie, auch wenn ein Beweis dafür auszustehen scheint.

   Folgendes Beispiel zeigt die Spielerei für 195 als Startwert; nach 4 Additionen ist ein Palindrom erreicht:

```
 195     786     1473    5214
+591    +687    +3741   +4125
----    ----    -----    ----
 786    1473     5214    9339
```

   Wir wollen eine Funktion schreiben, die diese Zahlenspielerei nachbildet. Dazu betrachten wir folgende Typen:

```
type InitialValue   = Integer
type NumberOfRounds = Integer
type MaxRounds      = Integer
type FinalValue     = Integer
type TextRep        = String
data Solution       = Failure | Success (NumberOfRounds,FinalValue,TextRep)
                                                          deriving (Eq, Show)
```

   Schreiben Sie eine Haskell-Rechenvorschrift `addRev :: InitialValue -> MaxRounds -> Solution`, die angewendet auf zwei ganze Zahlen m und n überprüft, ob die Ziffernfolge von $|n|$ in $|m|$ oder weniger Spielrunden in eine Zahl übergeht, deren Ziffernfolge ein Palindrom ist. Falls ja, liefert `addRev` den Wert `Success (r,v,s)`, wobei `r` die kleinste Zahl an Runden angibt, nach der ein Palindrom erreicht ist, `v` den Wert dieser Palindromzahl als ganze Zahl repräsentiert und `s` die zugehörige Zeichenreihendarstellung dieses Wertes ist. Wird innerhalb der ersten $|m|$ Runden kein Palindrom erreicht, so liefert `addRev` den Wert `Failure`.

   Folgende Beispiele zeigen das gewünschte Ein-/Ausgabeverhalten:

```
addRev 195 100 ->> Success (4,9339,"9339")
addRev (-195) (-10) ->> Success (4,9339,"9339")
addRev 195 3 ->> Failure
addRev 888 0 ->> Success (0,888,"888")
addRev 989 10 ->> Success (0,989,"989")
addRev 750 (-5) ->> Success (3,6666,"6666")
addRev 196 1000 ->> Failure
```

2. Weihnachten steht vor der Tür. Der Weihnachtsmann plant seine Reise, die ihn durch eine Vielzahl von Ländern führen soll. Leider fühlt sich Rudolf, sein Rentier, in diesem Jahr unwohl. Der Weihnachtsmann kann deshalb nicht mit seinem Schlitten wie sonst immer fahren, sondern muss auf Flugzeug, Schiff, Bahn und Bus umsteigen. Leider wird ihm auf Schiffen selbst schnell unwohl, in Flugzeugen ebenso. Deshalb möchte er pro Reise meist nur eine begrenzte Zeit auf See und in der Luft zubringen. Helfen Sie dem Weihnachtsmann bei der Reiseplanung mit einigen Haskell-Rechenvorschriften, die ihm Antworten auf wichtige Fragen geben. Wir benötigen dazu folgende Datentypen:

```
type Country    = String
type Countries  = [Country]
type TravelTime = Integer  -- Travel time in minutes
data Connection = Air Country Country TravelTime
                | Sea Country Country TravelTime
                | Rail Country Country TravelTime
                | Road Country Country TravelTime deriving (Eq,Ord,Show)
type Connections = [Connection]
data Itinerary   = NoRoute | Route (Connections,TravelTime) deriving (Eq,Ord,Show)
```

   - Gibt es eine Reiseroute von Land A nach Land B?
     `isRoute :: Connections -> Country -> Country -> Bool`
   - Liefere eine Reiseroute von Land A nach Land B, falls es eine solche gibt, ansonsten die leere Liste. Gibt es mehrere solche Reiserouten, so ist es egal, welche dieser Routen als Ergebnis geliefert wird:
     `yieldRoute :: Connections -> Country -> Country -> Connections`
   - Liefere die schnellste Reiseroute von Land A nach Land B, falls es eine solche gibt, zusammen mit der Gesamtreisedauer; ansonsten den Wert `NoRoute`. Gibt es mehrere solcher Reiserouten, so ist es egal, welche dieser Routen als Ergebnis geliefert wird:
     `yieldFastestRoute :: Connections -> Country -> Country -> Itinerary`
   - Gibt es eine Reiseroute von Land A nach Land B, auf der die Summe aus Flug- und Seezeiten höchstens max Minuten beträgt?
     `isFeelGoodRoute :: Connections -> Country -> Country -> TravelTime -> Bool`
   - Liefere eine Reiseroute von Land A nach Land B, auf der die Summe aus Flug- und Seezeiten höchstens max Minuten beträgt, falls es eine solche gibt, zusammen mit dieser Reisezeit; ansonsten den Wert `NoRoute`. Gibt es mehrere solcher Reiserouten, so ist es egal, welche dieser Routen als Ergebnis geliefert wird:
     `yieldFeelGoodRoute :: Connections -> Country -> Country -> TravelTime -> Itinerary`

   Dabei gilt: Zwischen je zwei Ländern kann es keine, eine oder auch mehr als eine Verbindung geben. Es kann auch mehr als je eine Flug-, See-, Bahn- oder Busverbindung zwischen zwei Ländern geben. Mit jeder Verbindung von Land A nach Land B ist zugleich auch die Rückverbindung mit gleicher Reisezeit gegeben. Ausgangs- und Zielland in einer Verbindung dürfen übereinstimmen; solche Inlandsverbindungen bringen den Weihnachtsmann auf dem Weg zu einem davon abweichenden Zielland allerdings nicht weiter. Auch sonst wird dem Weihnachtsmann nichts geschenkt. Nicht einmal Zeit. Deshalb ist bei möglicherweise negativ angegebenen Reisezeiten in `Connection`-Werten oder Funktionsaufrufen mit dem Betrag des entsprechenden `TravelTime`-Werts zu rechnen. Weiters gilt, dass alle Länder implizit durch ihr Vorkommen in mindestens einer der Verbindungen vom Wert `Connections` gegeben sind.

Hinweis:

- Verwenden Sie *keine* Module. Wenn Sie Funktionen wiederverwenden möchten, kopieren Sie diese in die Abgabedatei `Aufgabe8.hs`. Andere Dateien als diese werden vom Abgabeskript ignoriert.

Haskell Live

An einem der kommenden *Haskell Live*-Termine, der nächste ist am Freitag, den 07.12.2012, werden wir uns u.a. mit der Aufgabe *Tortenwurf* beschäftigen.

Tortenwurf

Wir betrachten eine Reihe von $n + 2$ nebeneinanderstehenden Leuten, die von paarweise verschiedener Größe sind. Eine größere Person kann stets über eine kleinere Person hinwegblicken. Demnach kann eine Person in der Reihe so weit nach links bzw. nach rechts in der Reihe sehen bis dort jemand größeres steht und den weitergehenden Blick verdeckt.

In dieser Reihe ist etwas Ungeheuerliches geschehen. Die ganz links stehende 1-te Person hat die ganz rechts stehende $n + 2$-te Person mit einer Torte beworfen. Genau p der n Leute in der Mitte der Reihe hatten während des Wurfs freien Blick auf den Tortenwerfer ganz links; genau r der n Leute in der Mitte der Reihe hatten freien Blick auf das Opfer des Tortenwerfers ganz rechts.

Wieviele Permutationen der n in der Mitte der Reihe stehenden Leute gibt es, so dass gerade p von ihnen freie Sicht auf den Werfer und r von ihnen auf das Tortenwurfopfer hatten?

Schreiben Sie in Haskell oder einer anderen Programmiersprache ihrer Wahl eine Funktion, die zu einer vorgegebenen Zahl $n \leq 10$ von Leuten in der Mitte der Reihe, davon p mit $1 \leq p \leq n$ mit freier Sicht auf den Werfer und r mit $1 \leq r \leq n$ mit freier Sicht auf das Opfer, diese Anzahl von Permutationen berechnet.