2. Aufgabenblatt zu Funktionale Programmierung vom Mi, 25.10.2017.

Fällig: Do, 02.11.2017 (15:00 Uhr) (später wg. Allerheiligen)

Themen: *Funktionen auf Werten elementarer Datentypen, Tupeln und Listen*

Zur Frist der Zweitabgabe: Siehe „Hinweise zu Organisation und Ablauf der Übung“ auf der Homepage der LVA (mit Do für Mi wg. Allerheiligen).

Aufgaben

Für dieses Aufgabenblatt sollen Sie Haskell-Rechenvorschriften für die Lösung der unten angegebenen Aufgabenstellungen entwickeln und für die Abgabe in einer Datei namens `Aufgabe2.hs` ablegen. Sie sollen für die Lösung dieses Aufgabenblatts also ein “gewöhnliches” Haskell-Skript schreiben. Versehen Sie wieder alle Funktionen, die Sie zur Lösung brauchen, mit ihren Typdeklarationen und kommentieren Sie Ihre Programme aussagekräftig. Benutzen Sie, wo sinnvoll, Hilfsfunktionen und Konstanten.

1. Gegeben sei die Funkion $p2p : IN_1 \times IN_1 \to IN_1 \times IN_1$ (“Paar zu Paar”) mit $p2p\ ((m,n)) = (p,q)$ für alle Paare $(m,n) \in IN_1 \times IN_1$. Dabei sind p und q die eindeutig bestimmten Zahlen aus IN_1 mit:

   - $m * p = n * q$
   - $\forall\, p', q' \in IN_1.\ m * p' = n * q' \Rightarrow p \leq p' \wedge q \leq q'$

   Schreiben Sie eine Haskell-Rechenvorschrift `p2p`

   ```
   type N1 = Int
   p2p :: (N1,N1) -> (N1,N1)
   ```

   so dass innerhalb des endlichen Ausschnitts der durch `N1` darstellbaren natürlichen Zahlen gilt:

   $$\forall\, (m,n) \in IN_1 \times IN_1.\ \texttt{p2p (m,n)} = p2p\ ((m,n))$$

   wobei `m`, `n` die Repräsentationen der Zahlen m, n in `N1` sind.

2. Wir betrachten noch einmal unser Lotterieunternehmen von Aufgabenblatt 1.

   Im Zuge einer Innovationsoffensive bietet dieses Unternehmen nicht mehr nur ein Glücksspiel nach dem Muster des `EuroMillionen`-Spiels an, sondern eine ganze Reihe. Für alle diese Spiele ist der Wetteinsatz pro Vorhersage und der mögliche Höchstgewinn unabhängig von der Zahl insgesamt eingebrachter Wetten gleich.

   Für einen Spieler ist deshalb ein Spiel um so attraktiver, je geringer die Zahl möglicher Wettkombinationen ist. Besitzen zwei Spiele dieselbe Zahl an Wettkombinationen, so ist das spannendere Spiel attraktiver. Ein Spiel ist spannender, je mehr Kugeln gezogen werden. So gibt es beim Spiel „1 aus 10“ genauso viele Wettkombinationen wie beim Spiel „9 aus 10“. Während beim ersten Spiel die Spannung bereits nach der ersten Kugel draußen ist, ist dies beim zweiten Spiel erst nach 9 Kugeln der Fall.

   Schreiben Sie eine Haskell-Rechenvorschrift `attraktiveSpieleVorne` mit der Signatur `attraktiveSpieleVorne :: AngeboteneSpiele -> [Glueckspiel]`, die die vom Unternehmen angebotenen Glücksspiele nach Attraktivität und Spannung anordnet. Je weniger Wettkombinationen ein Spiel bietet, desto weiter vorne in der

Ergebnisliste soll es stehen. Von Spielen mit gleicher Zahl an Wettkombinationen, soll jeweils das spannendere weiter vorne stehen. Sind Spiele bei gleicher Zahl von Wettkombinationen gleich spannend, d.h. wird die gleiche Gesamtzahl an Kugeln gezogen, ist egal, welches von diesen Spielen früher oder später als das andere angeordnet wird.

Einen Sonderfall bilden Spiele, für die die Zahl möglicher Wettkombinationen 0 ist, wenn mehr Kugeln gezogen werden sollen, als anfänglich im Topf vorhanden sind. Diese Spiele sollen von der Funktion `attraktiveSpieleVorne` ausssortiert und nicht in die Ergebnisliste eingeordnet werden.

Für die verwendeten Typen gelten folgende Vereinbarungen:

```
type Nat0              = Integer
type Nat1              = Integer
type GesamtKugelZahl   = Nat1
type GezogeneKugelZahl = Nat1
type Spiel             = (GesamtKugelZahl,GezogeneKugelZahl)
type Glueckspiel       = (Spiel,Spiel)
type AngeboteneSpiele  = [Glueckspiel]

attraktiveSpieleVorne :: AngeboteneSpiele -> [Glueckspiel]
```

3. *Die guten ins Töpfchen, die schlechten ins Kröpfchen.* Eine ganze Zahl ist 'gut', wenn die Zahl der Einsen in ihrer Darstellung im 3er-System, also im b-adischen System zur Basis 3, ohne Rest durch 3 teilbar ist, sonst 'schlecht'.

   Schreiben Sie eine Haskell-Rechenvorschrift `aufteilen` mit der Typsignatur `aufteilen :: Zahlenliste -> (Toepfchen,Kroepfchen)`, die eine Liste ganzer Zahlen in gute und schlechte aufteilt. Dabei sollen folgende Typvereinbarungen gelten:

   ```
   type Toepfchen   = [Int]
   type Kroepfchen  = [Int]
   type Zahlenliste = [Int]

   aufteilen :: Zahlenliste -> (Toepfchen,Kroepfchen)
   ```

   Angewendet auf eine Liste `ns` ganzer Zahlen, teilt die Funktion `aufteilen` die Elemente von `ns` auf zwei Teillisten `gs` (die 'Guten') und `ss` (die 'Schlechten') der Typen `Toepfchen` und `Kroepfchen` auf. Dabei landet ein Element `n` von `ns` in `gs`, wenn `n` 'gut' ist, sonst in `ss`. Die Elemente in `gs` und `ss` sollen dabei in derselben Reihenfolge wie in `ns` angeordnet sein, d.h. ein Element `n` in `gs` bzw. `ss` steht genau dann vor einem anderen Element `m`, wenn `n` auch schon in `ns` vor `m` steht.

4. Betrachtet man die ganzen Zahlen 0, 1, 2,...,9 als Ziffern, so lassen sich natürliche Zahlen als Listen der Werte 0, 1, 2,...,9 modellieren. So repräsentiert die Liste `[0]` die Zahl 0, die Listen `[1,2,3]` und `[0,0,0,1,2,3]` die Zahl 123 usw. Listen ohne führende Nullen heißen dabei in *Normalform*. Eine Ausnahme bildet die Zahl 0, deren Normalform die Liste `[0]` ist. Listen wie `[17,42,1]`, `[4,-5,3,0,0]` oder `[1,2,3,0,4711]` sind keine gültigen Darstellungen einer natürlichen Zahl.

Zur Modellierung in Haskell betrachten wir folgende Typ- und Wertvereinbarungen:

```
type Nat = [Int]
ziffern = [0,1,2,3,4,5,6,7,8,9] :: [Int]
```

- Schreiben Sie eine Wahrheitswertfunktion `istGueltig` mit der Typsignatur `istGueltig :: Nat -> Bool`, die angewendet auf einen Wert `w` vom Typ `Nat` überprüft, ob `w` gültige Repräsentation einer natürlichen Zahl ist oder nicht und entsprechend `True` oder `False` als Resultat liefert.
- Schreiben Sie eine Transformationsfunktion `normalForm` mit der Typsignatur `normalForm :: Nat -> Nat`, die angewendet auf eine gültige Repräsentation einer natürlichen Zahl deren Normalform als Ergebnis liefert, ansonsten die leere Liste zurückgibt.
- Schreiben Sie zwei Haskell-Rechenvorschriften `addiere` und `subtrahiere` zur Addition und Subtraktion zweier gültiger Darstellungen natürlicher Zahlen. Das Ergebnis soll dabei stets in Normalform sein. Für die Subtraktion gilt, dass die Differenz von `m` und `n` das Maximum von 0 und der Differenz von `m` und `n` in $\mathbb{Z}$ ist.

  ```
  addiere     :: Nat -> Nat -> Nat
  subtrahiere :: Nat -> Nat -> Nat
  ```

  Ist eines der Argumente der Funktionen `addiere` und `subtrahiere` keine gültige Darstellung einer natürlichen Zahl, so ist das Ergebnis von `addiere` und `subtrahiere` die leere Liste.

  Hinweis: Sind beide Argumente gültig, bringe die Listen mittels führender Nullen auf gleiche Länge, reversiere die enstandenen Listen, addiere und subtrahiere stellenweise mit Übertrag und reversiere schließlich die entstandene Liste, um die Resultatliste (vor Normalformbildung) zu erhalten.

Haskell Live

Am Freitag, den 20.10.2017, oder an einem der späteren Termine, werden wir uns in *Haskell Live* u.a. mit der Aufgabe *"Krypto Kracker!"* beschäftigen.

Krypto Kracker!

Eine ebenso populäre wie einfache und unsichere Methode zur Verschlüsselung von Texten besteht darin, eine Permutation des Alphabets zu verwenden. Bei dieser Methode wird jeder Buchstabe des Alphabets einheitlich durch einen anderen Buchstaben ersetzt, wobei keine zwei Buchstaben durch denselben Buchstaben ersetzt werden. Das stellt sicher, dass verschlüsselte Texte auch wieder eindeutig entschlüsselt werden können.

Eine Standardmethode zur Entschlüsselung nach obiger Methode verschlüsselter Texte ist als "reiner Textangriff" bekannt. Diese Angriffsmethode beruht darauf, dass der Angreifer den Klartext einer Textphrase kennt, von der er weiß, dass sie in verschlüsselter Form im Geheimtext vorkommt. Durch den Vergleich von Klartext- und verschlüsselter Phrase wird auf die Verschlüsselung geschlossen, d.h. auf die verwendete Permutation des Alphabets. In unserem Fall wissen wir, dass der Geheimtext die Verschlüsselung der Klartextphrase

```
the quick brown fox jumps over the lazy dog
```

enthält.

Ihre Aufgabe ist nun, eine Liste von Geheimtextphrasen, von denen eine die obige Klartextphrase codiert, zu entschlüsseln und die entsprechenden Klartextphrasen auszugeben. Kommt mehr als eine Geheimtextphrase als Verschlüsselung obiger Klartextphrase in Frage, geben Sie alle möglichen Entschlüsselungen der Geheimtextphrasen an. Im Geheimtext kommen dabei neben Leerzeichen ausschließlich Kleinbuchstaben vor, also weder Ziffern noch sonstige Sonderzeichen.

Schreiben Sie ein Programm in Haskell oder in irgendeiner anderen Programmiersprache ihrer Wahl, das diese Entschlüsselung für eine Liste von Geheimtextphrasen vornimmt.

Angewendet auf den aus drei Geheimtextphrasen bestehenden Geheimtext (der in Form einer Haskell-Liste von Zeichenreihen vorliegt)

```
["vtz ud xnm xugm itr pyy jttk gmv xt otgm xt xnm puk ti xnm fprxq",
 "xnm ceuob lrtzv ita hegfd tsmr xnm ypwq ktj",
 "frtjrpgguvj otvxmdxd prm iev prmvx xnmq"]
```

sollte Ihre Entschlüsselungsfunktion folgende Klartextphrasen liefern (ebenfalls wieder in Form einer Haskell-Liste von Zeichenreihen):

```
["now is the time for all good men to come to the aid of the party",
 "the quick brown fox jumps over the lazy dog",
 "programming contests are fun arent they"]
```