2. Aufgabenblatt zu Funktionale Programmierung vom Mi, 24.10.2018.
Fällig: Mi, 31.10.2018 (15:00 Uhr)

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.

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 (Wertvereinbarungen für) Konstanten.

1. Seien echte und unechte Querzahlen wie auf Aufgabenblatt 1 definiert. Schreiben Sie zwei Haskell-Rechenvorschriften `echt_quer_max` und `echt_quer` mit folgenden Signaturen:

   ```
   type Nat1 = Integer
   echt_quer_max :: Nat1 -> Nat1
   quer_alle :: Nat1 -> Nat1 -> [Nat1]
   ```

   - Angewendet auf eine natürliche Zahl n, $n \geq 1$, liefert `echt_quer_max` die größte echte Querzahl kleiner oder gleich n, wenn es eine solche gibt, anderenfalls den Wert 1.
   - Angewendet auf zwei natürliche Zahlen m, n liefert `quer_alle` die absteigend angeordnete Liste aller echten und unechten Querzahlen q mit $m \leq q \leq n$, wenn gilt: $1 \leq m \leq n$; ansonsten liefert `quer_alle` die leere Liste als Resultat.

2. Um Speicherplatz zu sparen, ist es oft ratsam, Zeichenfolgen in komprimierter Form abzulegen, aus der bei Bedarf wieder die originale Zeichenfolge gewonnen werden kann.

   Eine einfache Komprimierungsmethode macht sich Folgen wiederholender Zeichen in einer Zeichenfolge zunutze, sog. *Läufe*. Das folgende Beispiel verdeutlicht die Komprimierungsidee. Die Zeichenfolge

   ```
   "aaaaBBBaaBBBBBBccccccccccccAbCbDDDDDDDDDDDDDDDDDDD"
   ```

   wird kodiert als Folge von Paaren aus einer Ziffernfolge und einem von einer Ziffer verschiedenen Zeichen, wobei der Dezimalzahlwert der Ziffernfolgen jeweils die Länge des Laufs des auf die Ziffernfolge folgenden Zeichens angibt. Die obige Zeichenfolge hat also die Komprimierung:

   ```
   "4a3B2a6B12c1A1b1C1b18D"
   ```

   Damit dieses einfache Komprimierungs-/Expansionsverfahren funktioniert, muss verlangt werden, dass die zu komprimierende Zeichenreihe selbst keine Ziffern enthält. Umgekehrt können nur Zeichenreihen erfolgreich expandiert werden, die mit einer Ziffernfolge beginnen und in der auf jede Ziffernfolge genau ein von einer Ziffer verschiedenes Zeichen folgt.

   Schreiben Sie zwei Haskell-Rechenvorschriften `komp` und `exp` mit folgenden Eigenschaften:

   ```
   type Expandierte_Zeichenfolge  = String
   type Komprimierte_Zeichenfolge = String

   komp :: Expandierte_Zeichenfolge -> Komprimierte_Zeichenfolge
   exp  :: Komprimierte_Zeichenfolge -> Expandierte_Zeichenfolge
   ```

Angewendet auf eine nichtleere Zeichenfolge z, die frei von Ziffern ist, liefert die Funktion `komp` die entsprechend komprimierte Zeichenfolge (frei von führenden Nullen in Ziffernfolgen), ansonsten die Argumentfolge z. Umgekehrt liefert die Funktion `exp` die expandierte Zeichenfolge ihres Arguments, wenn dieses Argument als Komprimierung einer Zeichenfolge aufgefasst werden kann, ansonsten das Argument selbst.

Beachte: Von `komp` erzeugte Ziffernfolgen in komprimierten Zeichenfolgen sind frei von führenden Nullen. Umgekehrt können aber Zeichenfolgen mit führenden Nullen in Ziffernfolgen als Komprimierungen aufgefasst und erfolgreich expandiert werden. So expandieren `"0A"`, `"00A"`, `"000A"` zur leeren Zeichenfolge `""`; `"03A"`, `"003A"`, `"0003A"` zur Zeichenfolge `"AAA"`.

3. Schreiben Sie eine Haskell-Rechenvorschrift

```
aufteilen2 :: String -> [(String,String)]
```

die angewendet auf eine Zeichenreihe z eine Liste von Paaren von Zeichenreihen liefert, die alle Paaraufspaltungen von z darstellt. Folgendes Beispiel veranschaulicht die Bedeutung der Funktion:

```
aufteilen2 "Fun" ->> [("","Fun"),("F","un"),("Fu","n"),("Fun","")]
aufteilen2 "F" ->> [("","F"),("F","")]
aufteilen2 "" ->> [("","")]
```

4. Nicht alles, was glänzt, ist Gold. Manchmal ist es Katzengold. Eine natürliche Zahl ist ein *Goldnugget*, wenn sie eine Quadratzahl ist, die sich als Summe zweier Quadratzahlen darstellen lässt (z.B. $25 = 9 + 16 = 3^2 + 4^2$). Ist eine natürliche Zahl als Summe zweier Quadratzahlen darstellbar, ohne selbst eine Quadratzahl zu sein, so ist sie ein *Katzengoldnugget* (z.B. $13 = 4 + 9 = 2^2 + 3^2$). Ein Gold- und Katzengoldnugget ist um so wertvoller, je größer sein Zahlwert ist.

   Schreiben Sie eine Haskell-Rechenvorschrift `schuerfe`, die die Gold- und Katzengoldnuggets aus einer Mine ans Tageslicht befördert.

```
type Nat1             = Integer
type Mine             = [Nat1]
type Goldnugget       = Nat1
type Katzengoldnugget = Nat1
type Ausbeute         = ([Goldnugget],[Katzengoldnugget])

schuerfe :: Mine -> Ausbeute
```

   Angewendet auf eine Mine m, liefert die Funktion `schuerfe` ein Paar von Listen, die die Gold- und Katzengoldnuggets der Argumentmine m absteigend nach Wert geordnet enthalten.

   Aufrufbeispiele:

```
schuerfe [] ->> ([],[])
schuerfe [2,25,3,13,9,4,25] ->> ([25,25],[13,2])
```

Haskell Live (Achtung: Abweichender Termin - Do, 25.10.2018!)

Am Donnerstag, den 25.10.2018, 09:00 Uhr - 10:00 Uhr (Informatik-Hörsaal), oder an einem der späteren Termine, werden wir uns in *Haskell Live* u.a. mit der Aufgabe *"Krypto Kracker!"* beschäftigen.

Krypto Kracker!

Eine gleichermaßen 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 einer 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"]
```