2. Aufgabenblatt zu Funktionale Programmierung vom Mi, 19.10.2011. Fällig: Mi, 26.10.2011 / Mi, 02.11.2011 (jeweils 15:00 Uhr)

Themen: *Funktionen über ganzen Zahlen, Wahrheitswerten, Listen und Tupeln*

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.

Wir betrachten die folgende Teilmenge P der natürlichen Zahlen:

$$P =_{df} \{1 + 4n \mid n \in I\!N\} = \{1, 5, 9, 13, 17, 21, \ldots\}$$

Die Menge P ist bezüglich der Multiplikation abgeschlossen, d.h. das Produkt zweier Zahlen aus P ist selbst auch wieder aus P.

Einige Elemente aus P kann man in Faktorpaare aus P zerlegen. Z.B. gilt: $1 + 4 * 38 = 153 = 9 * 17$. Im allgemeinen ist diese Zerlegung nicht eindeutig. Z.B. gilt: $1 + 4 * 173 = 693 = 21 * 33 = 9 * 77$.

Andere Elemente aus P lassen sich nicht in dieser Weise in Faktorpaare aus P zerlegen. Z.B. 5, 9 und 13.

Elemente aus P, die sich nicht in Faktorpaare aus P zerlegen lassen, nennen wir *Primalzahlen.*

1. Schreiben Sie eine Haskell-Rechenvorschrift `istPrimal :: Integer -> Bool`, die überprüft, ob das Argument primal ist. Für Argumente nicht aus P liefert die Funktionsanwendung den Wert `False` liefern.

2. Schreiben Sie eine Haskell-Rechenvorschrift `faktorisiere :: Integer -> [(Integer,Integer)]`, die für ihr Argument die Liste aller Paarfaktorisierungen angibt. Die Anordnung der Elemente in der Liste ist gleichgültig. Ist das Argument primal, ist die Ergebnisliste leer. Ist das Argument nicht in P, wird die Auswertung mit dem Aufruf `error "Unzulaessig"` abgebrochen.

Ein einfacher Editor kann in Haskell als Zeichenliste realisiert werden:

```
type Editor = String
type Suchzeichenreihe = String
type Index = Integer
type Vorkommen = Integer
type Alt = String
type Neu = String
```

3. Schreiben Sie eine Haskell-Rechenvorschrift `suche` mit der Signatur `suche :: Editor -> Suchzeichenreihe -> Index`. Angewendet auf einen Editor `e` und eine Zeichenreihe `s`, liefert `suche` den kleinsten Index, an dem `s` in `e` als Teilzeichenreihe beginnt. Kommt `s` in `e` nicht vor, liefert `suche` den Wert (-1).

   Folgende Beispiele illustrieren das Aufrufverhalten:

   ```
   suche "Sein oder Nichtsein, das ist hier die Frage." "ei" -> 1
   suche "Sein oder Nichtsein, das ist hier die Frage." "sein" -> 15
   suche "Sein oder Nichtsein, das ist hier die Frage." "seit" -> (-1)
   suche "Sein oder Nichtsein, das ist hier die Frage." "Ei" -> (-1)
   ```

4. Schreiben Sie eine Haskell-Rechenvorschrift `sucheAlle` mit der Signatur `suche :: Editor -> Suchzeichenreihe -> [Index]`. Angewendet auf einen Editor `e` und eine Zeichenreihe `s`, liefert `sucheAlle` die aufsteigend geordnete Liste der Indizes, an denen `s` in `e` als Teilzeichenreihe beginnt. Kommt `s` in `e` nicht vor, liefert `sucheAlle` die leere Liste als Resultat.

   Folgende Beispiele illustrieren das Aufrufverhalten:

   ```
   sucheAlle "Sein oder Nichtsein, das ist hier die Frage." "ei" -> [1,16]
   sucheAlle "Sein oder Nichtsein, das ist hier die Frage." "sein" -> [15]
   ```

```
sucheAlle "Sein oder Nichtsein, das ist hier die Frage." "seit" -> []
sucheAlle "Mississippi." "issi" -> [1]
sucheAlle "aaaaaa" "aa" -> [0,2,4]
```

5. Schreiben Sie eine Haskell-Rechenvorschrift `ersetze` mit der Signatur `ersetze :: Editor -> Vorkommen -> Alt -> Neu -> Editor`. Angesetzt auf einen Editor `e`, eine ganze Zahl `i`, eine Zeichenreihe `s` und eine Zeichenreihe `t` ersetzt die Funktion das `i`-te Vorkommen von `s` in `e` durch `t` ersetzt. Gibt es weniger als `i` Vorkommen von `s` in `e`, gibt die Funktion `ersetze` den Editor `e` unverändert zurück, ebenso falls `i` nicht positiv ist.

   Die folgenden Beispiele illustrieren das Aufrufverhalten:

   ```
   ersetze "Sein oder Nichtsein." 2 "ein" "" -> "Sein oder Nichts."
   ersetze "Sein oder Nichtsein." 2 "sein" "mein" -> "Sein oder Nichtsein."
   ersetze "Sein oder Nichtsein." 1 "sein" "mein" -> "Sein oder Nichtmein."
   ersetze "Sein oder Nichtsein." (-3) "ei" "mein" -> "Sein oder Nichtsein."
   ersetze "Mississippi." 2 "issi" "abc"-> "Mississippi."
   ersetze "aaaaaa" 2 "aa" "bb" -> "aabbaa"
   ```

Wichtiger Hinweis: Wenn Sie einzelne Rechenvorschriften aus früheren Lösungen wieder verwenden möchten, so kopieren Sie diese in die neue Abgabedatei ein. Ein `import` schlägt für die Auswertung durch das Abgabeskript fehl, weil Ihre alte Lösung, aus der importiert wird, nicht mit abgesammelt wird. Dehalb: Kopieren statt importieren zur Wiederwendung!

Haskell Live

Am Freitag, den 21.10.2011, 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 "blanker 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"]
```