2. Aufgabenblatt zu Funktionale Programmierung vom Mi, 23.10.2013. Fällig: Mi, 30.10.2013 / Mi, 06.11.2013 (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.

1. Schreiben Sie eine Haskell-Rechenvorschrift `histo :: [Integer] -> Histogramm`, die angewendet auf eine Liste l ganzer Zahlen ein Histogramm erstellt, in dem die Elemente aus l absteigend nach der Zahl ihrer Vorkommen in l sortiert sind und die Anzahl der Vorkommen durch eine entsprechend lange Folge des Zeichens `'x'` gegeben ist. Haben zwei Einträge der Argumentliste gleich viele Vorkommen, so wird der kleinere vor dem größeren Eintrag im Histogramm aufgeführt. `Histogramm` ist dabei durch folgenden Typalias gegeben:

   ```
   type Histogramm = [(Integer,[Char])]
   ```

   Folgende Beispiele illustrieren das Aufrufverhalten:

   ```
   histo [4,2,0,4,3,2,1,2,4,(-2),0,4]
        ->> [(4,"xxxx"),(2,"xxx"),(0,"xx"),(-2,"x"),(1,"x"),(3,"x")]
   histo [] ->> []
   ```

2. Schreiben Sie eine Haskell-Rechenvorschrift `normalize :: String -> Integer -> String`, die die Argumentzeichenreiche s in Abhängigkeit des Zahlarguments n in eine normalisierte Zeichenreihe t überführt. Ist n kleiner oder gleich Null, so hat die Funktion `normalize` die leere Liste als Resultat. Ist n größer als Null, so enthält t von jedem Zeichen, das ein oder mehrere Vorkommen in s besitzt genau n Vorkommen. Hat ein Zeichen c in s mehr als n Vorkommen, so werden die überzähligen letzten Vorkommen von c in s gestrichen; hat c weniger als n Vorkommen in s, so werden die fehlenden Vorkommen von c unmittelbar aufeinanderfolgend am Ende von s angefügt. Ist für mehr als ein Zeichen ein Anfügen am Ende von s nötig, so gehen die Füllzeichen desjenigen Zeichen denen des anderen voraus, das das frühere Vorkommen in s hat. Hat c genau n Vorkommen in s, so sind dies auch die Vorkommen von c in t.

   Folgende Beispiele illustrieren das Aufrufverhalten:

   ```
   normalize "Ottokar" 3 ->> "OttokarOOtookkaarr"
   normalize "Mississippi" 2 ->> "MissippM"
   normalize "Mississippi 4 ->> "MississippiMMMpp"
   normalize "Mississippi 5 ->> "MississippiMMMMisppp"
   normalize "Mississippi (-5) ->> ""
   ```

3. Ein Palindrom ist eine Zeichenreihe, die in gleicher Weise von links nach rechts und von rechts nach links gelesen werden kann. Zum Beispiel sind die Zeichenreihen `"otto"` und `"AberrebA"` Palindrome. Auch Zahlen wie 123321 und 787, deren Ziffernfolgen ein Palindrom bilden, wollen wir als Palindrom auffassen. Wir sprechen dann von einem Zahlpalindrom.

   Schreiben Sie zwei Haskell-Rechenvorschriften `istPalindrom :: String -> Bool` und `istZahlPalindrom :: Integer -> Bool`, die angewendet auf eine Zeichenreihe bzw. eine nicht negative ganze Zahl entscheiden, ob das vorgelegte Argument ein Zeichenreihen- bzw. Zahlpalindrom ist. Wird `istZahlPalindrom` auf ein negatives Argument angewendet, so endet die Auswertung mit dem Aufruf `error "Keine negativen Argumente"`.

   Folgende Beispiele illustrieren das Aufrufverhalten:

   ```
   istPalindrom "otto" ->> True
   istPalindrom "Otto" ->> False
   istPalindrom "OhThO ->> True
   istZahlPalindrom 123454321 -> True
   istZahlPalindrom 123123 ->> False
   istZahlPalindrom (-123321) ->> error "Keine negativen Argumente"
   ```

4. Schreiben Sie eine Haskell-Rechenvorschrift `anzZahlPalindrome :: Integer -> Integer -> Integer`. Angewendet auf zwei nicht negative Argumente m und n bestimmt die Funktion `anzZahlPalindrome` die Anzahl aller Zahlpalindrome zp zwischen m und n, d.h. die Anzahl aller Zahlpalindrome mit $m \leq zp \leq n$, falls $m \leq n$, bzw. mit $n \leq zp \leq m$, falls $n \leq m$ ist. Ist m oder n negativ, hat die Funktion `anzZahlPalindrome` den Wert -1.

5. Schreiben Sie eine Haskell-Rechenvorschrift `anzPalindrome :: String -> String -> Integer`. Angewendet auf zwei gleich lange ausschließlich aus den Kleinbuchstaben `'a'`, `'b'`, `'c'` und `'d'` gebildeten Argumenten s und t der Länge l bestimmt die Funktion `anzPalindrome` die Anzahl aller Palindrome p der Länge l, die lexikographisch zwischen s und t (einschließlich s und t selbst) liegen (d.h. Anordnung der Zeichenreihen wie in Wörterbuch oder Lexikon und somit alle Zeichenreihen der Länge l, die zwischen s und t zu finden wären). Haben s und t nicht dieselbe Länge oder enthalten Zeichen verschieden von `'a'`, `'b'`, `'c'` und `'d'`, so hat die Funktion `anzPalindrome` den Wert -1.

Wichtig: Wenn Sie einzelne Rechenvorschriften aus früheren Lösungen für dieses oder spätere Aufgabenblätter 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. Deshalb: Kopieren statt importieren zur Wiederwendung!

Haskell Live

Am Freitag, den 25.10.2013, 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"]
```