Technische Universität München
Institut für Informatik
Prof. Tobias Nipkow, Ph.D.
Lars Noschinski, Lars Hupel, Dr. Jasmin Blanchette

WS 2013/14
29.10.2013
Abgabe: 05.11.2013, 15:30

Einführung in die Informatik 2
3. Übung

Aufgabe G3.1 Einfache Listenfunktionen

Implementieren Sie per Rekursion über Listen die folgenden Funktionen:

1. Die Funktion `snoc :: [a] -> a -> [a]` nimmt eine Liste $[x_1, \ldots, x_n]$ und ein Element y und gibt die Liste $[x_1, \ldots, x_n, y]$ zurück. Implementieren Sie diese Funktion ohne ++ zu verwenden.
2. Die Funktion `member :: Eq a => a -> [a] -> Bool` nimmt ein Element t und eine Liste $[x_1, \ldots, x_n]$ und liefert genau dann `True` zurück, wenn es ein $1 \leq i \leq n$ gibt mit $x_i = t$.

   (Diese Funktion ist übrigens Teil der Bibliothek `Data.List` unter dem Namen `elem`.)
3. Die Funktion `butlast :: [a] -> [a]` nimmt eine Liste $[x_1, \ldots, x_{n-1}, x_n]$ und gibt die Liste $[x_1, \ldots, x_{n-1}]$ zurück. Ist $n = 0$, so wird die leere Liste erwartet.

   Eine ähnliche Funktion ist übrigens Teil der Bibliothek `Data.List` unter dem Namen `init`.

Aufgabe G3.2 Entfernen wiederholter Elemente

1. Schreiben Sie eine Funktion `uniq :: Eq a => [a] -> [a]`, die aufeinanderfolgende gleiche Elemente entfernt. Zum Beispiel sollen die folgenden Gleichheiten gelten:

```
uniq []       == []           uniq [1,2,2,1]   == [1,2,1]
uniq [1,2,3]  == [1,2,3]      uniq [1,1,4,1,1] == [1,4,1]
```

   Eine Hilfsfunktion vom Typ `Eq a => a -> [a] -> [a]` kann hilfreich sein.
2. Schreiben Sie eine Funktion `uniqCount :: Eq a => [a] -> [(a, Integer)]`, die zusätzlich zählt, wie viele gleiche Elemente in der Eingabe aufeinander folgen. Zum Beispiel sollen die folgenden Gleichheiten gelten:

```
uniqCount []           == []
uniqCount [1,2,3]      == [(1,1),(2,1),(3,1)]
uniqCount [1,2,2,1]    == [(1,1),(2,2),(1,1)]
uniqCount [1,1,4,1,1]  == [(1,2),(4,1),(1,2)]
```

   Eine Hilfsfunktion vom Typ `Eq a => (a, Integer) -> [a] -> [(a, Integer)]` kann hilfreich sein.

Aufgabe G3.3 Trenner

Implementieren Sie die folgenden Funktionen.

1. Die Funktion `intersep :: a -> [a] -> [a]` nimmt ein Element t (den *Trenner*) und eine Liste $[x_1, x_2, \ldots, x_n]$ und liefert die Liste $[x_1, t, x_2, t, \ldots, t, x_n]$ zurück, mit $n-1$ Trennern. Für $n = 0$ wird die leere Liste erwartet. Zum Beispiel:

   ```
   intersep ',' "" == ""
   intersep ',' "a" == "a"
   intersep ',' "ab" == "a,b"
   intersep ',' "abcdef" == "a,b,c,d,e,f"
   intersep 0 [3, 2, 1] == [3, 0, 2, 0, 1]
   ```

   (Diese Funktion ist übrigens Teil der `List`-Bibliothek unter dem Namen `intersperse`.)

2. Die Funktion `andList :: [[Char]] -> [Char]` nimmt eine Liste von Wörtern $[w_1, w_2, \ldots, w_{n-1}, w_n]$ und liefert im Allgemeinen den Text „w_1, w_2, …, w_{n-1}, and w_n“ zurück, mit einem Komma vor dem „and“.[1] Es gibt aber Ausnahmen für $n < 3$:

   ```
   andList [] == ""
   andList ["Ayize"] == "Ayize"
   andList ["Bhekizizwe", "Gugu"] == "Bhekizizwe and Gugu"
   andList ["Jabu", "Lwazi", "Nolwazi"] ==
     "Jabu, Lwazi, and Nolwazi"
   andList ["Phila", "Sihle", "Sizwe", "Zama"] ==
     "Phila, Sihle, Sizwe, and Zama"
   ```

Aufgabe G3.4 Dreieck (optional)

Gesucht ist eine polymorphe Funktion `triangle :: [a] -> [(a, a)]`, die eine Liste von Werten $[a_1, \ldots, a_n]$ nimmt und die folgende $n(n-1)/2$-elementige Liste von Paaren zurückgibt:

$$\big[(a_1,a_2),\ (a_1,a_3),\ (a_1,a_4),\ \ldots\ (a_1,a_{n-1}),\ (a_1,a_n),$$
$$(a_2,a_3),\ (a_2,a_4),\ \ldots\ (a_2,a_{n-1}),\ (a_2,a_n),$$
$$(a_3,a_4),\ \ldots\ (a_3,a_{n-1}),\ (a_3,a_n),$$
$$\ddots\ \vdots\ \vdots$$
$$(a_{n-2},a_{n-1}),\ (a_{n-2},a_n),$$
$$(a_{n-1},a_n)\big]$$

Zum Beispiel:

```
triangle [] == []
triangle [1111] == []
triangle [1, 2] == [(1, 2)]
triangle [222, 11] == [(222, 11)]
triangle ["AA", "AA"] == [("AA", "AA")]
triangle [1, 2, 3] == [(1, 2), (1, 3), (2, 3)]
triangle [3, 5, 7, 11] == [(3, 5), (3, 7), (3, 11),
                           (5, 7), (5, 11), (7, 11)]
```

[1] Dieses Komma heißt „serial comma“ oder „Oxford comma“ und ist im Prinzip wahlfrei, wobei es bei amerikanischen Autoren sehr beliebt ist.

Die Funktion lässt sich sowohl rekursiv als auch mit Listenkomprehensionen schreiben.

Schreiben Sie QuickCheck-Tests für Ihre Implementierung.

> **Hinweis:** Sofern nicht anders angegeben, dürfen Sie alle Funktionen der Standardbibliothek (und QuickCheck) für die Hausaufgaben benutzen. Im Zweifel ist genau das erlaubt, was auf `fp.in.tum.de` korrekt kompiliert wird.

Aufgabe H3.1 Leersymbole normalisieren (4 Punkte)

Wir definieren den Begriff eines Leersymbols.

Leersymbol Ein Zeichen, für das die Funktion `isSpace` (aus der `Data.Char`-Bibliothek) den Wert `True` zurück gibt. Unter anderem Leerzeichen (`' '`) und Zeilenumbruch (`'\n'`).

Schreiben Sie eine Funktion `simplifySpaces :: [Char] -> [Char]`, die in der Eingabe aufeinanderfolgende Leersymbole durch ein einziges Leerzeichen ersetzt, sowie führende und folgende Leersymbole entfernt. Alle verbleibenden Leersymbole sollen durch Leerzeichen ersetzt werden.

Beispiel (␣ ist ein Leerzeichen, `\n` ein Zeilenumbruch und `\t` ein Tabulator):

```
simplifySpaces
  "␣␣\nA\t␣quick␣␣␣brown␣\n\tfox\njumps␣over␣the\rlazy␣dog\r\n" ==
  "A␣quick␣brown␣fox␣jumps␣over␣the␣lazy␣dog"
```

Aufgabe H3.2 Goldene Mitte (5 Punkte)

Gegeben sei eine Funktion `centre :: [Char] -> [Char]`, die eine Zeichenkette nimmt und den Text je enthaltener Zeile zentriert. Die Ausgabe ist die umformatierte Zeichenkette. Wir definieren einige Begriffe. Für die Definition von Leersymbolen vergleiche H3.1.

Zeile Eine Zeichenkette wird durch Zeilenumbrüche (`'\n'`) in Zeilen getrennt. Der Zeilenumbruch ist nicht Teil der Zeile. Die Zeilen einer Zeichenkette können mit der Haskell-Funktion `lines :: [Char] -> [[Char]]` aus (`Prelude`) berechnet werden.

Führende Leersymbole Die führenden Leersymbole einer Zeichenkette sind alle Leersymbole vom Anfang der Zeichenkette bis zum ersten Zeichen, das kein Leersymbol ist. Zum Beispiel sind `"␣\t␣"` die führenden Leersymbole von `"␣\t␣hello␣world\t␣␣"`.

Folgende Leersymbole Die folgenden Leersymbole einer Zeichenkette sind das alle Leersymbole vom letzten Nicht-Leersymbol bis zum Ende der Zeichenkette. Zum Beispiel sind `"\t␣␣"` die folgenden Leersymbole von `"␣\t␣hello␣world\t␣␣"`.

Ihre Aufgabe ist es jetzt, QuickCheck-Tests zu schreiben, die kontrollieren, ob die Ausgabe der `centre`-Funktion die folgenden Regeln erfüllt:

1. Die Anzahl der Zeilen in Ein- und Ausgabe sind identisch.
2. Jede Zeile der Ausgabe ist gleich lang.

3. Es gibt eine Zeile in der Ausgabe ohne führende und folgende Leersymbole.
4. Abgesehen von führenden und folgenden Leersymbolen sind die Zeilen von Ein- und Ausgabe gleich.
5. Als führende und folgende Leersymbole einer Zeile ist in der Ausgabe nur das Leerzeichen `'␣'` erlaubt.
6. Die Anzahlen der führenden und der folgenden Leersymbole einer Zeile dürfen sich höchstens um 1 unterscheiden.

Ihre Tests sollen das gewünschte Verhalten der Funktion `centre` *vollständig* spezifizieren; das heißt, wenn `centre` nicht korrekt implementiert wurde, muss es eine Eingabe für einen Ihrer Tests geben, so dass der Test fehlschlägt. (Ob diese Eingabe von QuickChecks Zufallsstrategie tatsächlich gefunden wird, sei dahingestellt.)

Sie können bis zu zehn Tests `prop_centre1`, ..., `prop_centre10` schreiben, es reichen aber deutlich weniger aus. Jede der Testfunktionen muss die folgende Form haben:

```
prop_centreN :: CentreFun -> [Char] -> Bool
prop_centreN centre ... = ...
```

Zum Beispiel:

```
prop_centre3 :: CentreFun -> [Char] -> Bool
prop_centre3 centre xs = centre xs == centre (xs + "  ")
```

Sie sehen hier, dass die Funktion `centre` sowohl links als auch rechts vom Gleichheitszeichen steht. Ignorieren Sie das vorerst. Der Typ `CentreFun` ist in der offiziellen Übungsschablone[2] durch

```
type CentreFun = [Char] -> [Char]
```

definiert. Sie können Ihre `prop_centre`-Funktionen testen, indem Sie eine Funktion mit der richtigen Typsignatur als erstes Argument angeben. Zum Beispiel:

```
prop_centre3 simplifySpaces "Hallo\nWelt!\n"
```

(Hier ist `simplifySpaces` nur ein Beispiel, für die mindestens einer Ihrer Tests fehlschlagen sollte.)

Aufgabe H3.3 Rotation gegen den Uhrzeigersinn (5 Punkte)

Wie aus der Vorlesung bekannt werden Bilder (Typ `Picture`) durch Listen von Strings dargestellt. In der Übungsschablone finden Sie die Definition eines Beispielbildes

```
pic = [".##.", ".#.#", ".###", "####"]
```

sowie der Funktion `printPicture :: Picture -> IO ()`, die Sie als Black-Box für die zweidimensionale Ausgabe von Bildern **im Interpreter** nutzen können. Zum Beispiel:

[2] `http://www21.in.tum.de/teaching/info2/WS1314/blaetter/Exercise_3.hs`

```
Exercise_3> printPicture pic
.##.
.#.#
.###
####
```

Sie sollen eine Funktion `rotateCCW :: Picture -> Picture` definieren, die ein Bild um 90° gegen den Uhrzeigersinn dreht. Zum Beispiel:

```
Exercise_3> printPicture (rotateCCW pic)
.###
#.##
####
...#
```

Beachten Sie dass die Eingabebilder nicht unbedingt „rechteckig“ sein müssen (die Längen der inneren Listen bzw. Zeilen dürfen unterschiedlich sein). Dabei kann es passieren, dass Sie nach der Rotation manche Zeilen mit dem Leerzeichen `'.'` auffüllen müssen, wie die Beispiele a_1, b_1, c_1 und a_2, b_2, c_2 zeigen. a_i ist dabei jeweils das Eingabebild angezeigt mittels `printPicture` (die Eingabe im Beispiel a_1 ist `["##","."]` und im Beispiel a_2 `[".##",".#","#"]`). b_i ist das erwartete Ergebnis nach der Rotation, welches jedoch nicht mit unserer Repräsentation der Bilder kompatibel ist. Die Funktion `rotateCCW` soll stattdessen die in c_i dargestellten Bilder zurückgeben. Beachten Sie außerdem Leerzeilen in der Eingabe, die natürlich auch aufgefüllt werden müssen.

a_1)
```
##
.
```

b_1)
```
#
#.
```

c_1)
```
#.
#.
```

a_2)
```
.##
.#
#
```

b_2)
```
#
##
..#
```

c_2)
```
#..
##.
..#
```

Aufgabe H3.4 Ver- und Entschlüsselung (6 Punkte, Wettbewerbsaufgabe)

Quelle: `www.focus.de`

Der Master of Competition wurde vom Bundeskanzleramt beauftragt, die SMS-App der Kanzlerin auf ihre Sicherheit zu prüfen. Dabei musste der MC feststellen, dass die in C++ geschriebene App nicht nur unverständlich war, sondern auch gravierende Sicherheitslücken aufwies. Daraufhin beschloss der MC eine SafeKanzlerSMS-App auszuschreiben: Der Kern muss in Haskell programmiert werden.

Ein Modul des Kerns soll zwei Funktionen realisieren:

```
encode :: String -> String -> String
decode :: String -> String -> String
```

Der Aufruf `encode` *key cleartext* verschlüsselt den Klartext *cleartext* mit dem Schlüssel *key*. Der Schlüssel ist eine (duplikatenfreie) Permutation der ersten n kleinen Buchstaben des englischen Alphabets ($n \leq 26$). Zum Beispiel ist `dcab` eine Permutation der ersten vier Buchstaben des Alphabets und deshalb ein gültiger Schlüssel. Der Klartext ist ein beliebiger Text, der ausschließlich aus den Buchstaben des Schlüssels besteht.

Ein Schlüssel $[k_0, \ldots, k_{n-1}]$ dient als Lookuptabelle für den Kodierungsalgorithmus. Der Klartextbuchstabe `a` wird mit dem ersten Schlüsseleintrag (d.h. `a` $\mapsto k_0$), `b` mit dem zweiten Schlüsseleintrag (d.h. `b` $\mapsto k_1$), ..., `z` mit dem sechsundzwanzigsten Schlüsseleintrag (d.h. `z` $\mapsto k_{25}$) kodiert. Auf diese Weise wird der Klartext `avecaesar` mit dem Schlüssel `bcd...yza` (d.h. {`a` $\mapsto$ `b`, `b` $\mapsto$ `c`, `c` $\mapsto$ `d`, ..., `x` $\mapsto$ `y`, `y` $\mapsto$ `z`, `z` $\mapsto$ `a`}) zu `bwfdbftbs` kodiert.

In der Gegenrichtung entschlüsselt der Aufruf `decode` *key cryptotext* den Kryptotext *cryptotext* mit dem Schlüssel *key*. Für gültige Schlüssel und Klartexte gilt die folgende Eigenschaft, die übrigens einen ausgezeichneten QuickCheck-Test ausmacht:

`decode` *key* `(encode` *key cleartext*`) ==` *cleartext*,

Beispiele:

```
encode "" "" == ""
encode "bca" "" == ""
encode "bca" "aabbccaba" == "bbccaabcb"
encode "bdac" "dcbadbca" == "cadbcdab"
encode "bcdefghijklmnopqrstuvwxyza" "avecaesar" == "bwfdbftbs"

decode "" "" == ""
decode "bca" "" == ""
decode "bca" "bbccaabcb" == "aabbccaba"
decode "bdac" "cadbcdab" == "dcbadbca"
decode "bcdefghijklmnopqrstuvwxyza" "bwfdbftbs" == "avecaesar"
```

Sie dürfen für Ihre Implementierung annehmen, dass die beiden Funktionen nur mit gültigen Schlüsseln und Texten aufgerufen werden. Zudem dürfen Sie alle Funktionen aus der Standardbibliothek verwenden. Insbesondere könnten die folgenden Funktionen aus `Data.Char` und `Data.List` nützlich sein:

```
(!!) :: [a] -> Int -> a      -- "abcdef" !! 4 == 'e'
chr :: Int -> Char           -- chr 97 == 'a'
ord :: Char -> Int           -- ord 'a' == 97
elemIndices :: a -> [a] -> [Int]
                             -- elemIndices 'e' "abcdef" == [4]
head :: [a] -> a             -- head [4, 5, 6] == 4
```

Der Buchstabe `a` stellt einen beliebigen Typ (z.B. `Integer`, `Char` oder `String`) dar.

Für den Wettbewerb zählt die Tokenanzahl (je kleiner, desto besser[3]). Lösungen, die gleichwertig bezüglich der Tokenanzahl sind, werden im Hinblick auf ihre Lesbarkeit verglichen, wobei Formatierung und Namensgebung besonders gewichtet werden.

Die vollständige Lösung (außer Standardfunktionen und `import`-Direktiven) muss innerhalb der Kommentare {-WETT-} und {-TTEW-} abgegeben werden, um als Wettbewerbsbeitrag zu zählen. Zum Beispiel:

```
import Test.QuickCheck
import Data.Char
import Data.List

{-WETT-}
encode :: String -> String -> String
encode key cleartext = ...

decode :: String -> String -> String
decode key cryptotext = ...
{-TTEW-}
```

Der Kollege des Master of Competition hat für seine Lösung 25 Token (exklusive Typsignaturen) gebraucht. Unterbieten Sie ihn!

Wichtig: Wenn Sie diese Aufgabe als Wettbewerbsaufgabe abgeben, stimmen Sie zu, dass Ihr Name ggf. auf der Ergebnisliste auf unserer Internetseite veröffentlicht wird. Sie können diese Einwilligung jederzeit widerrufen, indem Sie eine Email an `fp@fp.in.tum.de` schicken. Wenn Sie nicht am Wettbewerb teilnehmen, sondern die Aufgabe allein im Rahmen der Hausaufgabe abgeben möchten, lassen Sie bitte die {-WETT-} ... {-TTEW-} Kommentare weg. Bei der Bewertung Ihrer Hausaufgabe entsteht Ihnen hierdurch kein Nachteil.

[3] `http://www21.in.tum.de/teaching/info2/WS1314/wettbewerb.html`