Funktionale Programmierung

`http://proglang.informatik.uni-freiburg.de/teaching/functional-programming/2013/`

Übungsblatt 5 (IO, Testdaten-Generierung)

Mi, 2013-11-20

Hinweise

- Lösungen sollen als Haskell Quellcode in das persönliche Subversion (svn) Repository hochgeladen werden. Die Adresse des Repositories wird per Email mitgeteilt.
- **Alle** Aufgaben müssen bearbeitet und pünktlich abgegeben werden. Falls das sinnvolle Bearbeiten einer Aufgaben nicht möglich ist, kann eine stattdessen eine Begründung abgegeben werden.
- Wenn die Abgabe korrigiert ist, wird das Feedback in das Repository hochgeladen. Die Feedback-Dateinamen haben die Form `Feedback-<user>-ex<XX>.txt`.
- Allgemeinen Fragen zum Übungsblatt können im Forum (`http://proglang.informatik.uni-freiburg.de/forum/viewforum.php?f=38`) geklärt werden.

Abgabe: Mi, 2013-11-27

Aufgabe 1 (Zahlenraten)

Wir implementieren das Spiel „Zahlenraten". Bei diesem Spiel versucht der Computer eine vom Benutzer gedachte Zahl zwischen 1 und 100 zu erraten. Hier ist ein Beispiellauf:

```
Main> game
Wähle eine Zahl zwischen 1 und 100!
Ist es 50? größer
Ist es 75? kleiner
Ist es 62? kleiner
Ist es 56? genau
Juhu, ich habe gewonnen!
```

(Der Text nach den Fragezeichen ist die Benutzereingabe.)

Implementieren Sie „Zahlenraten" als Konsolen-Spiel, dass ähnlich wie oben gezeigt funktioniert.

Aufgabe 2 (Listenpaare)

1. Implementieren Sie einen Generator

   ```
   1 listOfLength :: Integer -> Gen a -> Gen [a]
   ```

   so dass `listOfLength n g` eine `n`-elementige Liste mit Werten aus dem Generator `g` generiert. Verwenden Sie *nicht* den vordefinierten Generator `vectorOf`.

2. Implementieren Sie weiterhin einen Generator, der Paare gleich- (aber zufällig-) langer Listen erzeugt.

3. Definieren Sie zwei Properties für die Funkionen **zip** und **unzip**:

   a) **zip** ist die inverse Funktion für **unzip**

   b) **unzip** ist die inverse Funktion für **zip**

   Die zweite Property gilt nicht immer. Testen Sie die Properties mit geeigneten Generatoren, die den benötigten Vorbedingungen genügen.

Aufgabe 3 (Permutierte Listen)

Implementieren Sie einen Generator, der zufällige Permutation einer gegebenen Liste erzeugt. Um Speicherplatz zu sparen, verwenden Sie dabei *nicht* die Funktion Data.**List**.permutations.

Aufgabe 4 (TicTacToe zum Zweiten)
Implementieren Sie einen Generator für ihre TicTacToe Repräsentation aus Übungsblatt 2. Alternativ können Sie auch die Repräsentation aus der Musterlösung verwenden.

Ist das Verhältnis zwischen gültigen und ungültigen Spielen akzeptabel? Implementieren Sie auch einen Generator, der mindestens zu 60% gültige Spiele generiert. Außerdem sollte er so effizient sein, dass problemlos 1000 Spiele generiert werden können.

Ein Spiel-Wert ist ungültig, wenn

- es nicht durch wechselseitiges Ziehen der Spieler entstehen kann,
- mehr als ein Spieler gewonnen hat
- oder der Wert nicht sinnvoll als TicTacToe Spiel interpretiert werden kann.