Funktionale Programmierung

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

Übungsblatt 4 (Funktionen höherer Ordnung, IO)

Di, 2013-11-12

Hinweise

- Lösungen sollen 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: Di, 2013-11-19

Erratum 2013-11-14 Die Definition von **foldl** in Aufgabe 1 wurde korrigiert.

Erratum 2014-01-07 Der zweite Teil des optionalen Teils von Aufgabe 3 ist nicht sinnvoll und wurde gestrichen.

Aufgabe 1 (foldr)

Implementieren Sie die folgenden Funktionen mittels **foldr**:

1. **or**, liefert **True** falls mindestens ein Element einer **Bool**-Liste **True** ist
2. **filter**
3. **map**
4. **foldl**, die links-assoziative Variante von **foldr**:

```
1 foldl :: (b -> a -> b) -> b -> [a] -> b
2 foldl _ acc [] = acc
3 foldl f acc (x:xs) = foldl f (f acc xs) xs
```

5. remdups, entfernt aufeinanderfolgende Duplikate aus einer Liste

Aufgabe 2 (unfoldr)

Es gibt auch eine duale Funktion zu **foldr**:

```
1 unfoldr :: (b -> Maybe (a, b)) -> b -> [a]
```

Anstatt eine Liste zu einem Endergebnis zu reduzieren, baut **unfoldr** f seed eine neue Liste auf: Die Elemente der Liste werden durch wiederholtes Anwenden der Funktion f auf den Akkumulator b erzeugt. Gibt f b den Wert **Nothing** zurück, ist die Liste zu Ende. Gibt f b den Wert **Just** (a, b') zurück, dann wird a als vorderstes Element hinzugefügt. Der Wert b' wird dann zum Berechnen des nächsten Elements an f übergeben.

1. Definieren Sie **unfoldr**. (Natürlich, ohne die vordefinierte Version Data.**List**.**unfoldr** zu benutzen)
2. Definieren Sie **map** mittels **unfoldr**

3. Eine weitere Standardfunktion der funktionalen Programmierung ist

```
-- Definiert in Data.List, wird auch von Prelude exportiert
iterate :: (a -> a) -> a -> [a]
```

Was könnte diese Funktion tun? Implementieren Sie **iterate** mittels **unfoldr**.

Aufgabe 3 (Wahrscheinlichkeiten)
Wenn Sie mit 10% Wahrscheinlichkeit in der Bahn kontrolliert werden, wie oft dürfen Sie schwarz fahren, damit Sie mindestens mit einer 75%igen Wahrscheinlichkeit damit durchkommen?

Bei einem Durchgang Korrekturlesen finden Sie einen Fehler mit einer Wahrscheinlichkeit von 80%. Wie oft müssen Sie ihre Abschlussarbeit durchlesen, damit Sie mit einer Wahrscheinlichkeit von größer 99% korrekt ist?

Wenn Sie versuchen diese Fragen mit Hilfe von Haskell zu beantworten, könnten Sie wie folgt beginnen:

```
gesparteBahnfarten = triesWithAcceptableRisc 75 10
notwendigeKorrekturen = iterationsForAcceptableYield 99 80
triesWithAcceptableRisc = undefined
iterationsForAcceptableYield = undefined
```

Definieren Sie die Funktionen triesWithAcceptableRisc und iterationsForAcceptableYield. Versuchen Sie so wenig Code wie möglich zu duplizieren.

Optional: Versuchen Sie die Aufgabe mit der Standardfunktion **scanl** zu lösen. ~~Versuchen Sie auch einmal **scanl** mittels **foldr** selbst zu definieren.~~

Aufgabe 4 (Binärkodierung)
Definieren Sie Funktionen zur Binärkodierung und -dekodierung von **Integer**n.

```
type Bit = Bool
type Word = [Bit]
encode :: Int -> Integer -> [Word]
decode :: Int -> [Word] -> Maybe Integer
```

Die Serialisierung soll **Integer** beliebiger Größe unterstützen und parametrisierbar in der Wortgröße sein. Das erste Argument von encode und decode gibt die Wortgröße an. Die Byte-Reihenfolge soll little-endian sein und es soll 2er-Komplement-Darstellung verwendet werden. Innerhalb eines Wortes soll das Least-Significant-Bit an erster Stelle stehen (also lsb word = word !! 0)

Aufgabe 5 (readLine)
Implementieren Sie die IO-Aktion readLine :: **IO String**. Sie soll wie **getLine** funktionieren, allerdings zusätzlich auch erlauben, dass man dem Benutzer die Eingabe mit Backspace korrigieren kann.

Hinweise: Das Backspace-Zeichen wird in Haskell mit der Escape-Sequenz `\DEL` in String- und Zeichenliteralen dargestellt. (Das Newline-Zeichen ist wie immer `\n`). Desweiteren ist das Kontrollzeichen „Carriage-Return", `\CR`, interessant, das den Terminal-Cursor an den Beginn der Zeile setzt.

Auf manchen Systemen ist es nötig, die Ein-/Ausgabe Pufferung abzustellen. Wenn Sie Probleme mit dem Einlesen und Ausgeben haben, verwenden Sie die folgende IO-Aktion um die Pufferung abzustellen:

```
hSetBuffering stdin NoBuffering >>
  hSetBuffering stdout NoBuffering >>
  hSetEcho stdin False
```

Dazu ist es nötig das Modul **System.IO** zu importieren.

Aufgabe 6 (Stack Calculator Interface)
Wir schreiben unser erstes ausführbares Haskell-Programm! Implementieren Sie ein Kommandozeilen-Interface für den Stack-Rechner aus Blatt 2. Das Interface soll den aktuellen Stack darstellen und einzelne Stack Befehle vom Benutzer einlesen und ausführen, bis der String „exit" eingegeben wird.

Für die Rechen-Logik können Sie ihre eigene Implementierung oder die aus der Musterlösung verwenden bzw. anpassen.

Versuchen Sie den Stack-Rechner als eigenständig ausführbares Programm zu compilieren. Dazu muss eine Aktion `main :: IO ()` definiert sein und Sie müssen die Quelldatei mit dem Shell-Befehl `ghc -make <Name>.hs` kompilieren.