Algorithmen und Programmieren 1

Funktionale Programmierung

- Übungsklausur -

Punkte: **A1:** 30, **A2:** 20, **A3:** 20, **A4:** 20, **A5:** 10, **A6:** 20

`Punkte:  /120`

10.02.2012

Hinweis: Geben Sie bei allen verwendeten Funktionen die Signaturen an. Viel Erfolg!

1 Haskell-la-vista! *(10+10+10=30 Punkte)*

1.1 Funktionen höherer Ordnung *(10 Teilpunkte)*

Schreiben Sie unter Verwendung der `map` und `foldr` Funktionen eine Funktion `sumQuad`, die bei Eingabe eines ganzzahligen positiven Werts n, die Summe alle Quadratzahlen von 1 bis n berechnet.

1.2 Klassen und Instanzen *(10 Teilpunkte)*

Seien zwei verschiedene algebraische Datentypen, einer für die Darstellung der natürlichen Zahlen (N) und einer für die Darstellung der ganzen Zahlen (Z) gegeben:

```
data N = Zero | S (N) deriving (Show)
data Z = Z (N,N)      deriving (Show)
```

Erklärung:
Wenn `a` die Zahl x darstellt und `b` die Zahl y darstellt, dann stellt Z `(a,b)` die Zahl $x - y$ dar.

Beispiele:

```
Mit dem Datentyp N kann man Zahlen wie folgt darstellen:
 3 = S(S(S(Zero)))
 7 = S(S(S(S(S(S(S(Zero)))))))

Mit dem Datentyp Z kann man Zahlen wie folgt darstellen:
 3 = Z ( S(S(S(Zero))) , Zero )
 3 = Z ( S(S(S(S(Zero)))) , S(Zero) )
-2 = Z ( Zero , S(S(Zero)) )
-2 = Z ( S(S(Zero)) , S(S(S(S(Zero)))) )
```

Schreiben Sie die Typ-Klasse `Rechnen`, die Ihnen die Operation *Addition* bereitstellt und implementieren Sie jeweils eine Instanz für jeden algebraischen Datentypen.

1.3 Typ-Inferenz *(10 Teilpunkte)*

Es seien folgende Funktionssignaturen bekannt.

```
map    :: (a -> b) -> [a] -> [b]
sum    :: (Num a) => [a] -> a
length :: [a] -> Int
show   :: (Show a) => a -> String
```

Bestimmen Sie den Typ der folgenden Funktionen und begründen Sie Ihr Ergebnis.

```
f = sum . map sum
g = map (show . length)
```

2 Datenbank *(20 Punkte)*

Schreiben Sie ein *Modul* `SimpleDB`, das einen *algebraischen Datentyp* `Datenbank` verwendet um eine einfache Datenbank zu simulieren. Eine Datenbank ist eine Liste von 2-Tupeln. Jedes dieser 2-Tupel hat die Form: `(key, [value`$_1$`, value`$_2$`, ... , value`$_n$`])`, wobei $n \geq 1$ und jeder Schlüssel einzigartig sein muss (Primärschlüssel). Überlegen Sie sich geeignete Typklassen für die Typen der Schlüssel und der Werte. Es sollen folgende Funktionen implementiert werden:

- `create`
  Erzeugt eine leere Datenbank.
  Der Rückgabewert ist eine Datenbank.

- `select db key`
  Diese Funktion gibt alle Wert zurück, die einem Schlüssel in einer Datenbank zugeordnet sind.

  - Falls der Schlüssel nicht existiert wird eine leere Liste ausgegeben.
  - Falls der Schlüssel existiert wird eine Liste mit allen Werten, die diesem Schlüssel zugeordnet sind, zurückgegeben.

  Der Rückgabewert ist eine Liste von Werten.

- `update db key value`
  Diese Funktion fügt einer Datenbank einen Schlüsseleintrag mit zugeordnetem Wert hinzu.

  - Falls der Schlüssel existiert und der Wert noch nicht dem Schlüssel zugeordnet ist, wird der Wert dem Schlüssel zugeordnet.
  - Falls der Schlüssel existiert und der Wert bereits dem Schlüssel zugeordnet ist, wird nichts verändert.
  - Falls der Schlüssel nicht existiert wird dieser erzeugt und der Wert diesem zugeordnet.

  Der Rückgabewert ist eine Datenbank.

- `drop db key value`
  Diese Funktion löscht aus einer Datenbank einen Wert, der einem Schlüssel zugeordnet war.

  - Falls der Schlüssel oder der Wert nicht existiert wird ein Fehler ausgegeben.
  - Falls der Schlüssel existiert wird der Wert aus der Zuordnung zum Schlüssel entfernt.
  - Falls der Schlüssel existiert und nur noch der angegebene Wert als einziger Wert dem Schlüssel zugeordnet ist, wird der Schlüssel ebenfalls aus der Datenbank entfernt.

  Der Rückgabewert ist eine Datenbank.

Tipp: Hilfreiche Funktionen sind `takeWhile`, `dropWhile` und `filter`.

3 Sortieren + Laufzeit *(10+10=20 Punkte)*

3.1 Sortieren durch Einfügen *(10 Teilpunkte)*

Implementieren Sie den **Insertion-Sort-Algorithmus.**

3.2 Laufzeit *(10 Teilpunkte)*

Bestimmen Sie die Laufzeit Ihrer Implementierung.
Es reicht, wenn Sie die Laufzeit in O-Notation angeben und diese verbal begründen.

4 Strukturelle Induktion *(10+10=20 Punkte)*

4.1 Bäume *(10 Teilpunkte)*

Sei folgender algebraischer Datentyp und folgende Funktion gegeben:

```
data Baum = Blatt Int | Knoten Baum Baum
   deriving (Show)

-- wendet die Funktion f auf jedes Element des Baumes an
mapTree :: (Int -> Int) -> Baum -> Baum
mapTree f (Blatt x)      = Blatt (f x)                          -- mapT.1
mapTree f (Knoten lB rB) = Knoten (mapTree f lB) (mapTree f rB)  -- mapT.2
```

Beweisen Sie mit struktureller Induktion folgende Behauptung:

```
mapTree id baum = id baum
```

4.2 Listen *(10 Teilpunkte)*

Seien folgende Gleichungen gegeben:

```
++.1  :    []  ++ ys       = ys
++.2  : (x:xs) ++ ys       = x:(xs ++ ys)

rev.1 : reverse []         = []
rev.2 : reverse (x:xs)     = reverse xs ++ [x]
rev.3 : reverse (xs ++ ys) = (reverse ys) ++ (reverse xs)
```

Beweisen Sie mit struktureller Induktion folgende Behauptung:

```
reverse (reverse xs) = xs
```

5 Primitive Rekursion *(10 Punkte)*

Rekursionsschema:
$R(0, x_1, ..., x_m) = g(x_1, ..., x_m)$
$R(S(n), x_1, ..., x_m) = h(R(n, x_1, ..., x_m), n, x_1, ..., x_m)$

Zeigen Sie, dass die Funktion `isOdd` (Ein Prädikat, das prüft ob eine Zahl ungerade ist. Das Ergebnis ist 1 wenn die Zahl ungerade ist und 0 sonst.) primitiv-Rekursiv ist. Sie dürfen nur die Grundfunktionen S und Z voraussetzen, das heißt, alle Funktion die Sie ansonsten für Ihren Beweis verwenden müssen Sie ebenfalls beweisen.

6 λ - Kalkül *(5+5+10=20 Punkte)*

6.1 Freie und gebundene Ausdrücke *(5 Teilpunkte)*

Bestimmen Sie die freien und gebundenen Variablen im folgenden Ausdruck und geben Sie diese explizit an. Benennen Sie die gebundenen Variablen so um, dass in jedem Ausdruck alle λ-Abstraktionen verschiedene Variablen binden.

$$(\lambda ab.(\lambda ab.(\lambda c.(\lambda ab.abc))ab(cc))(\lambda ab.a)(\lambda ac.c(c(a))))$$

6.2 λ - Ausdruck reduzieren *(5 Teilpunkte)*

$$(\lambda a.a(\lambda ab.b)(\lambda a.a(\lambda ab.b)(\lambda ab.a))(\lambda ab.b))(\lambda ab.ab)$$

6.3 λ - Ausdruck erstellen *(10 Teilpunkte)*

Schreiben Sie einen λ - Ausdruck, der rekursiv die Quadrate aller natürlichen Zahlen von 0 bis n addiert. Sie können dabei die folgenden Funktionen als gegeben voraussetzen:

- **A** $\equiv$ Addition (Präfixnotation)
- **M** $\equiv$ Multiplikation (Präfixnotation)
- **P** $\equiv$ Vorgänger
- **Y** $\equiv$ Fixpunktoperator
- **Z** $\equiv$ Vergleich auf 0