Erster Haskell-Workshop
im OpenLab Augsburg
github.com/curry-club-aux/haskell-workshop

Übungsblatt zu Haskell

Learn You a Haskell for Great Good!

1 Aufwärmübungen in GHCi

Aufgabe 1. *Verschachtelte Tupel*

Benutze die vordefinierten Funktionen `fst :: (a, b) -> a` und `snd :: (a, b) -> b`, um das Textzeichen aus `(1, ('a', "foo"))` zu extrahieren.

Aufgabe 2. *Medianbestimmung*

Sei `xs` eine unsortierte Liste von Zahlen, z. B. `let xs = [3, 7, -10, 277, 89, 13, 22, -100, 1]`. Schreibe einen Ausdruck, der den Median (das mittlere Element in einer Sortierung der Liste) von `xs` berechnet. Verwende dazu `length`, `div` und `!!`.

Aufgabe 3. *Der Smiley-Operator erster Ordnung*

Was könnte der Ausdruck `(.) . (.)` bewirken? Finde es heraus mit Hilfe von GHCi!

2 Spiel und Spaß mit Listenfunktionen

3 Funktionsdefinitionen

4 Eigene Datentypen

Aufgabe 4. *Binäre Bäume*

Im Folgenden verwenden wir folgende Definition für binäre Bäume, deren Verzweigungsknoten mit Werten vom Typ `Int` dekoriert sind.

```
data Tree = Nil | Fork Int Tree Tree
    deriving (Show)
```

a) Schreibe eine Funktion, die die Gesamtzahl Blätter eines Baums berechnet: `numberOfLeaves :: Tree -> Int`
b) Schreibe eine Funktion, die die Höchsttiefe eines Baums berechnet.
c) Schreibe eine Funktion, die die Int-Werte der Verzweigungsknoten in einer Reihenfolge deiner Wahl als Liste zurückgibt.

Aufgabe 5. *Binäre Bäume bilden einen Funktor*

a) Verallgemeinere die vorherige Aufgaben auf Bäume, die Werte von einem beliebigen Typ `a` statt `Int` tragen. Vervollständige dazu zunächst folgende Definition:

```haskell
data Tree a = Nil | ...
    deriving (Show)
```

b) Implementiere eine Funktion `tmap :: (a -> b) -> Tree a -> Tree b`.

Aufgabe 6. *Unendliche Bäume*

a) Schreibe eine Funktion `cutOff :: Int -> Tree a -> Tree a`, die eine Maximaltiefe und einen Baum als Argumente nimmt und einen neuen Baum zurückgibt, der sich aus dem gegebenen durch Abschneidung bei der gegebenen Maximaltiefe ergibt.

b) Definiere eine Funktion, die eine unendliche Liste von Werten nimmt und einen Baum zurückgibt, auf dessen Verzweigungsknoten die Elemente der Liste sitzen. Suche dir selbst aus, in welcher Reihenfolge die Listenelemente auf dem Baum platziert werden sollen.

Aufgabe 7. *Der Stern–Brocot-Baum (für Fans von Kettenbrüchen)*

Informiere dich auf Wikipedia über den Stern–Brocot-Baum und implementiere ein Haskell-Programm, dass diesen unendlichen Baum berechnet. Hole dir gegebenenfalls einen (stark spoilernden) Tipp ab.

Aufgabe 8. *Termbäume*

a) Implementiere einen Datentyp für Funktionsterme. Zum Beispiel soll

$$(x \cdot x + 3) - x$$

so repräsentiert werden: `Sub (Add (Mul Var Var) (Lit 3)) Var`.

b) Schreibe eine Funktion `eval :: Exp -> Double -> Double`, die in einem gegebenen Term für die Variable x einen konkreten Wert einsetzt.

c) Schreibe eine Funktion `diff :: Exp -> Exp`, die einen gegebenen Funktionsterm ableitet. Zum Beispiel soll `diff (Mul Var Var)` im Wesentlichen äquivalent sein zu `Mul (Lit 2) Var`.

Aufgabe 9. *Isomorphe Typen*

Manche Typen lassen sich verlustfrei ineinander umwandeln, zum Beispiel die folgenden beiden:

```haskell
data Bool     = False  | True   -- schon vordefiniert
data Aussage = Falsch | Wahr
```

Man spricht in einem solchen Fall von *zueinander isomorphen Typen*. Die Umwandlungsfunktionen heißen *Isomorphismen* und können in diesem Fall wie folgt definiert werden:

```haskell
iso :: Bool -> Aussage
iso False = Falsch
iso True  = Wahr

osi :: Aussage -> Bool
osi Falsch = False
osi Wahr   = True
```

Das charakteristische an diesen beiden Funktionen ist, dass `osi . iso == id` und `iso . osi == id`.

Folgende Typen sind jeweils zueinander isomorph. Implementiere auf analoge Weise Funktionen `iso` und `osi`, die das bezeugen!

a) `(a, b)` versus `(b, a)`

b) `((a, b), c)` versus `(a, (b, c))`

c) `(a, Either b c)` versus `Either (a, b) (a, c)`

d) `a -> (b, c)` versus `(a -> b, a -> c)`

e) `(a, b) -> c` versus `(a -> b -> c)`

5 Typklassen

Aufgabe 10. *Eigene Show-Instanzen*

Für Debugging-Zwecke oder auch zum Datenaustausch ist die Show-Klasse nützlich, deren Definition in etwa die folgende ist:

```
class Show a where
    show :: a -> String
```

Bei der Deklaration eines neuen Datentyps hat man die Möglichkeit, mit einer `deriving`-Klausel den Compiler anzuweisen, automatisch eine geeignete Show-Instanz zu generieren:

```
data Tree a = Nil | Fork a (Tree a) (Tree a)
    deriving (Show)
```

In dieser Aufgabe aber sollst du den dafür nötigen Boilerplate-Code von Hand schreiben. Such dir einen Datentyp deiner Wahl aus und schreibe eine individuelle Show-Instanz für ihn.

Aufgabe 11. *Die Monoid-Typklasse*

Das Modul `Data.Monoid` definiert die Monoid-Typklasse:

```
class Monoid a where
    mempty  :: a
    mappend :: a -> a -> a
    mconcat :: [a] -> a
```

Ihr gehören solche Typen an, die eine sog. *Monoidstruktur* tragen. (Wenn du diesen Begriff nicht kennst, dann frag kurz nach!) Das neutrale Element soll durch `mempty` angegeben werden, die Monoidoperation durch `mappend`. Die Funktion `mconcat` soll gleich mehrere Elemente miteinander verknüpfen.[1]

a) Gebe einer Nachimplementierung des Listen-Datentyps, etwa `data List a = Nil | Cons a (List a)` eine Monoid-Instanz. Vergiss nicht, zu Beginn deines Programmtexts mit `import Data.Monoid` die Definition der Monoid-Klasse zu laden.

b) Implementiere folgende Funktion:

```
cata :: (Monoid m) => (a -> m) -> ([a] -> m)
```

[1] Die Funktionen `mappend` und `mconcat` lassen sich gegenseitig ausdrücken. Fällt dir ein Grund ein, wieso trotzdem beide Funktionen Teil der Klasse sind? Hätte man nicht auch einfach `mconcat` außerhalb der Klasse definieren können?

Aufgabe 12. *Sortierung nach mehreren Kriterien*

Oft steht man vor folgendem Problem: Eine Liste von Dingen soll nach mehreren Kriterien sortiert werden. Etwa zunächst nach Nachname, unter gleichen Nachnamen aber nach Vorname und unter gleichem Namen nach Geburtsdatum. Die in Haskell idiomatische Herangehensweise an dieses Problem verwendet … Monoide!

a) Schlage den `Ordering`-Typ nach.

b) Reimplementiere die Funktion

```
comparing :: (Ord a) => (b -> a) -> b -> b -> Ordering
```

aus dem Modul `Data.Ord`. Sie kann zum Beispiel so verwendet werden:

```
import Data.List

data Person = MkPerson
    { lastName  :: String
    , givenName :: String
    , birthday  :: String
    }
    deriving (Show)

sortPersons :: [Person] -> [Person]
sortPersons = sortBy (comparing lastName)

-- sortBy hat den Typ (a -> a -> Ordering) -> [a] -> [a].
```

c) Trägt ein Typ `a` eine Monoidstruktur, so auch der Typ `e -> a` der Funktionen von `e` nach `a`. Bestätige das, indem du folgenden Code vervollständigst:

```
instance (Monoid a) => Monoid (e -> a) where
    -- ...
```

Da diese Instanz schon in `Data.Monoid` vordefiniert ist, musst du für diese Teilaufgabe den Import von `Data.Monoid` entfernen und die Monoid-Typklasse selbst definieren.

d) Was macht folgender Code? Wieso tut er das? Informiere dich dazu über die Monoid-Instanz von `Ordering` und erinnere dich an die Monoid-Instanz von Funktionstypen.

```
sortBy $ mconcat
    [ comparing lastName
    , comparing firstName
    , comparing birthday
    ]
```

Aufgabe 13. *Endliche Typen*

Manche Typen fassen nur endlich viele Werte, zum Beispiel `Bool` und `Either Bool Bool`. Für solche Typen ist es gelegentlich praktisch, eine vollständige Liste ihrer Werte zu kennen. Aus diesem Grund führen wir folgende Klasse ein:

```
class Finite a where
    elems :: [a]
```

a) Implementiere eine Finite-Instanz für `Bool`.

b) Implementiere folgende allgemeinen Instanzen:

```haskell
instance (Finite a, Finite b) => Finite (a,b)       where ...
instance (Finite a, Finite b) => Finite (Maybe a)   where ...
instance (Finite a, Finite b) => Finite (Either a b) where ...
```

c) Wenn du Lust auf eine Herausforderung hast, dann implementiere auch folgende Instanz. Sie ist für die weiteren Teilaufgaben aber nicht nötig.

```haskell
instance (Eq a, Finite a, Finite b) => Finite (a -> b) where ...
```

d) Implementiere eine Funktion `exhaustiveTest :: (Finite a) => (a -> Bool) -> Bool`.

e) Die Gleichheit zweier Funktionen (vom selben Typ) ist im Allgemeinen nicht entscheidbar, denn zwei Funktionen sind genau dann gleich, wenn sie auf allen Eingabewerten übereinstimmen. Um das zu überprüfen, muss man im Allgemeinen unendlich viele Fälle in Augenschein nehmen. Wenn der Quelltyp aber endlich ist, geht es doch. Implementiere also:

```haskell
instance (Finite a, Eq b) => Eq (a -> b) where ...
```

Aufgabe 14. *Abzählbare Typen*

Manche Typen sind zwar nicht endlich, aber immer noch *abzählbar*: Das heißt, dass es eine unendliche Liste gibt, in der alle Werte des Typs vorkommen. Zum Beispiel ist der Typ `Integer` abzählbar, denn in der Liste `[0, 1, -1, 2, -2, ...]` kommen alle ganzen Zahlen vor.

a) Definiere nach dem Vorbild der Finite-Typklasse aus der vorherigen Aufgabe eine Countable-Typklasse.

b) Implementiere eine Countable-Instanz von `Integer`.

c) Vervollständige folgenden Code:

```haskell
instance (Countable a, Countable b) => Countable (a,b) where ...
```

d) Vervollständige folgenden Code (schwierig!):

```haskell
instance (Countable a) => Countable [a] where ...
```

Dabei soll `[a]` für den Typ der endlichen Listen mit Werten in `a` stehen – obwohl der Typ `[a]` ja auch unendliche Listen enthält. Solche *sozialen Verträge* sind in Haskell leider gelegentlich nötig – man benötigt abhängige Typen und andere Entwicklungen, um sie vollständig zu vermeiden. Sauberer wäre an dieser Stelle, einen neuen Datentyp `FiniteList a` zu definieren, der isomorph zum gewöhnlichen Listentyp ist, aber den sozialen Vertrag an zentraler Stelle kundtut.

Aufgabe 15. *Überabzählbare Typen*

Diese Aufgabe richtet sich nur an Leute, die das sog. *Cantorsche Diagonalargument* und die *Russelsche Antinomie* kennen. Sorry! Bei Gelegenheit suchen wir eine einführende Referenz.

Wir definieren ein Typalias für Mengen:

```haskell
type Set a = a -> Bool

-- Ist `f :: Set a`, so soll `f x == True` bedeuten, dass `x` in
-- der Menge `f` liegt.
```

a) Setze in diesem Modell die leere Menge, die Universalmenge (welche alle Werte überhaupt enthält) und die Menge, die nur ein bestimmtes Element enthält, um. Welche Voraussetzung an den Typ `a` musst du im letzten Teil stellen?

b) Implementiere folgende Funktionen:

```
member       :: a     -> Set a -> Bool
union        :: Set a -> Set a -> Set a
intersection :: Set a -> Set a -> Set a
complement   :: Set a -> Set a
```

c) Setze die Russelsche Antinomie in Haskell um. Definiere also eine Menge all derjenigen Mengen, die sich nicht selbst enthalten. Wie äußert sich das paradoxe Verhalten in Haskell?

d) Setze das Cantorsche Diagonalargument in Haskell um. Definiere also eine Funktion

```
cantor :: (a -> Set a) -> Set a
```

die folgendes leistet: Für jede Funktion `f :: a -> Set a` soll `cantor f` eine Menge sein, die nicht im Bild (in der Wertemenge) von `f` enthalten ist.

e) Bonusfrage zum Grübeln: Die vorherige Teilaufgabe zeigt, dass es in Haskell überabzählbare Typen gibt. Andererseits ist die Menge der Haskell-Programme abzählbar. Wie passt das zusammen?

6 Ideen für größere Projekte

Aufgabe 16. *Ein Zahlenrätsel*

Wie muss man die Zahlen 1, 3, 4 und 6 mit Klammern und den Operatoren `+ * - /` ergänzen, damit das Ergebnis 24 ist? Die Zahlen dürfen und müssen dabei alle genau einmal verwendet werden. Sie können aber in einer beliebigen Reihenfolge auftreten. Denkbar wäre also etwa die Lösung `3+((1-4)/6)`, aber dieser Term hat den Wert 2,5.

Schreibe ein Haskell-Programm, das für dich die Lösung findet! Ein mögliches Vorgehen ist folgendes.

a) Definiere einen Datentyp `Exp` von Termen zu definieren. Das Beispiel könnte dabei durch `Add (Lit 3) (Div (Sub (Lit 1) (Lit 4)) (Lit 6))` ausgedrückt werden.

b) Schreibe eine Funktion `eval :: (Fractional a) => Exp a -> a`.

c) Schreibe eine Funktion `groups :: [a] -> [([a],[a])]`, die folgendes leistet: Gegeben eine Liste, berechnet alle Möglichkeiten, diese Liste in zwei Teile zu zerlegen: einen vorderen und einen hinteren. Zum Beispiel:

```
> groups "abc"
[("abc",""),("ab","c"),("a","bc"),("","abc")]
```

d) Schreibe eine Funktion `arb :: [a] -> [Exp a]`, die folgendes leistet: Gegeben eine Liste `xs` von (zum Beispiel) Zahlen, gibt eine Liste von allen Termbäumen zurück, an deren Blättern (in genau der gegebenen Reihenfolge) die Zahlen aus `xs` stehen. Alle Zahlen müssen verwendet werden, und zwar jeweils genau einmal.

e) Importiere oder reimplementiere die Funktion `permutations :: [a] -> [[a]]` aus `Data.List`.

f) Füge alle Puzzleteile zusammen.