Funktionale Programmierung

Blatt 3

Prof. Dr. Oliver Braun

Fakultät für Informatik und Mathematik
Hochschule München

Letzte Änderung: 09.10.2018 06:54

Wir werden auf diesem Blatt eine kleine Verwaltung für eine Videothek beginnen.

Das (leere) Repository für dieses und das nächste Blatt bekommen Sie unter https://classroom.github.com/a/2yu_VJdb. Erzeugen Sie sich wieder ein Stack-Projekt inkl. Travis CI-Anbindung und GitHub-Pages.

Implementieren Sie immer auch gleich Tests und schreiben Sie Dokumentation.

Filme als Tupel und Listen

Wir wollen unsere Videothek folgendermaßen repräsentieren:

```
type Id         = Int
type Name       = String
type FSK        = Int
type Movie      = (Id, Name, FSK)
type Moviestore = ( [Movie] {- verfügbare Filme-}
                  , [Movie] {- ausgeliehene Filme -})
```

Mit `type` können wir sogenannte Typsynonyme definieren, die den Code lesbarer machen. Wir sagen dem Compiler: Immer wenn Du den Typ `Name` siehst, ersetze das einfach durch den Typ `String`. Wir können das später noch besser mit selbst definierten Typen.

Ein Film ist also ein Tripel bestehend aus Id, Name und FSK und eine Videothek ist ein Tupel bestehend aus zwei Listen von Filmen. Die erste Liste sind die verfügbaren, also nicht ausgeliehenen, Filme, die zweite Liste die ausgeliehenen.

Eine Beispielvideothek könnte also folgendermaßen aussehen:

```haskell
myMovieStore :: Moviestore
myMovieStore = ( [ (1, "Matrix"                          , 16)
                 , (2, "Alpen - unsere Berge von oben",  0)
                 ]
               , [ (3, "The Breakfast Club"              , 12)
                 ]
               )
```

a) Implementieren Sie eine Funktion

```haskell
showMovie :: Movie -> String
```

die einen Film in eine Zeichenkette umwandelt. Z.B.

```haskell
showMovie (2,"Alpen - unsere Berge von oben",0)
    == "Alpen - unsere Berge von oben (Id: 2, FSK 0)"
```

Also zuerst der Titel und dann in Klammern die ID und die FSK-Freigabe.

Achtung: Nutzen Sie die Funktion `show` zum Umwandeln von `Int`s in `String`s.

b) Implementieren Sie eine Funktion

```haskell
showMovieList :: [Movie] -> String
```

die eine Liste von Filmen in einen String umwandelt der die Filme zeilenweise enthält.

c) Implementieren Sie eine Funktion

```haskell
showMoviestore :: Moviestore -> String
```

Die aus einem Moviestore eine Zeichenkette macht und zwar in der Form, dass zuerst alle verfügbaren, dann alle ausgeliehenen Filme angezeigt werden. Im GHCi (`stack repl`) soll das dann beispielsweise so aussehen:

```
*Main> putStrLn $ showMoviestore myMovieStore
Verfügbare Filme
================
Matrix (Id: 1, FSK 16)
Alpen - unsere Berge von oben (Id: 2, FSK 0)

Ausgeliehene Filme
==================
The Breakfast Club (Id: 3, FSK 12)
```

d) Erweitern Sie die Funktion `showMoviestore` so, dass hinter der jeweiligen Überschrift in Klammern die Anzahl der verfügbaren bzw. ausgeliehenen Filme steht. Der Doppelstrich unter der Überschrift soll dann entsprechend verlängert werden.

Die neue Ausgabe soll also so aussehen:

```
Verfügbare Filme (2)
====================
Matrix (Id: 1, FSK 16)
Alpen - unsere Berge von oben (Id: 2, FSK 0)

Ausgeliehene Filme (1)
======================
The Breakfast Club (Id: 3, FSK 12)
```

Probieren Sie insbesondere auch aus, ob bei 10 oder mehr Filmen in einer Liste ein ausreichend langer Doppelstrich erzeugt wird.

e) Implementieren Sie eine Funktion

```
extract :: Id -> [Movie] -> (Maybe Movie, [Movie])
```

die zu einer gegebenen `Id` den entsprechenden Film aus einer Filmliste entfernt. Wenn der Film nicht in der Liste enthalten ist, soll ein Tupel bestehend aus `Nothing` und der unveränderten Liste zurück gegeben werden. Sonst `Just movie` und die Liste ohne den Film.

Also z.B.

```
*Main> extract 0 [(1,"Matrix",16),(2,"Alpen - unsere Berge von oben",0)]
(Nothing,[(1,"Matrix",16),(2,"Alpen - unsere Berge von oben",0)])
*Main> extract 1 [(1,"Matrix",16),(2,"Alpen - unsere Berge von oben",0)]
(Just (1,"Matrix",16),[(2,"Alpen - unsere Berge von oben",0)])
```

Der Datentyp `Maybe a` besitzt die beiden Typkonstruktoren `Nothing` und `Just a`. Wenn eine Funktion erfolgreich ist, wird das Ergebnis in ein Just verpackt, ansonsten wird ein Nothing zurück gegeben, z.B.

```
biggerOne :: Ord a => a -> a -> Maybe a
biggerOne x y | x == y    = Nothing -- none is bigger
              | x < y     = Just y
              | otherwise = Just x
```

Wenn Sie `Optional` von Java 8, `std::optional` von C++17 oder `Option` von Scala kennen, wissen Sie was `Maybe` sein soll.

f) Implementieren Sie unter Verwendung von `extract` die beiden Funktionen

```
return :: Id -> Moviestore -> Moviestore
rent :: Age -> Id -> Moviestore -> (Bool, Moviestore)
```

`Age` ist ein Typsynonym für `Int`.

Mit der Funktion `return` können Sie einen Film zurück geben, mit `rent` ausleihen. Natürlich muss der Film dazu in der entsprechenden Liste vorhanden sein und beim Ausleihen muss die FSK-Freigabe zum Alter passen.