Harshil Shah – SEC01 (NUID 002780887)

Big Data System Engineering with Scala
Spring 2023
Assignment No. 5 (Functional Composition)

- GitHub Repo URL - https://github.com/harshilshahneu/CSYE7200-Harshil-Shah

---

- List of Tasks Implemented

All the work you are going to do is in the asstfc package. Ingest hasn't changed, but Movie (and MovieSpec -- also inasstfc) have.

There are 13 TODOs in Function.scala and 2 TODOs in Movie.scala. You can get these from the class repo (see Course Material/Resources/Class Repository), the module name for this assignment is assignment-functional-composition.

Did you find it at all bothersome (in Assignments 1 and 2) that we didn't use Try (or Option) to deal with some of the String => Int conversions? Some of those invocations of toInt might easily have thrown exceptions. Part of the problem was that it was too difficult to do, given your knowledge of Scala at the time.

But now, you know all about functional composition, for comprehensions and monads.

The basic problem that I had to solve is that some of the apply functions (for example in Format) have a mixture of strings, as well as Int and Double which require conversion from String. One of the elements of a Movie (viz. Reviews) was based on seven Int conversions from seven strings. That's easy: just use map7 (!). Of course we didn't actually implement map7 in class, but you're going to implement it in this assignment. But what about mixtures like in Format? That required a rather creative solution (if I may say so). Take a look at the code in the object Format and also in the object Production. First, we need to curry the apply method. Functions in curried form are much more tractable than functions in tupled (normal) form. Now, the rest of the code transforms that curried function into a function that can be used in a for comprehension. Don't worry too much if you don't understand what's going on here. But, I believe that if you take the time to figure it out, you can see what's happening.

In any case, this code uses some utilities that I placed in the Function object.

Now, here's an important hint for the one TODO that is in the Movie.scala file: Back in assignment 2, we had a for comprehension which utilized a filter (you can see that by taking a look at Movie. scala in assignment-movie-database which is unchanged). At the time, we hadn't talked about for comprehensions but hopefully you figured it out (it is kind of obvious).

But do you remember in a recent lecture that I said that the left hand side of a generator (in a for comprehension) is actually a pattern? Yes, it's a pattern just like in the case statement of a match expression. Neat, huh? So, you are going to implement the filter on the country value by using a pattern in the for comprehension instead of a filter (thus you will not be using the method isKiwi or anything like it). Do take a look at how it was implemented (using a filter) in our previous version.

But there's one tricky bit of syntax which we haven't covered yet. In Scala there can arise a certain ambiguity when you have identifiers that are the same as keywords (not allowed in Java) or, as here, you have a variable mentioned in a pattern matching context where you don't want just to match on anything (you would be "shadowing" the variable in such a case). You actually want to match on the value of the variable. For this (and the similar case where you want to use a keyword as a variable), just enclose the variable name inside back-ticks the "`" character.

- Code

- Function.scala

```
def map2[T1, T2, R](t1y: Try[T1], t2y: Try[T2])(f: (T1, T2) => R): Try[R] = t1y.flatMap(t1 => {
  t2y.map(t2 => {
    f(t1, t2)
  })
})
```

```scala
def map3[T1, T2, T3, R](t1y: Try[T1], t2y: Try[T2], t3y: Try[T3])(f: (T1, T2, T3) => R): Try[R] = for {
  t1 <- t1y
  t2 <- t2y
  t3 <- t3y
} yield f(t1, t2, t3) // TO BE IMPLEMENTED

/**
  * You get the idea...
  */
def map7[T1, T2, T3, T4, T5, T6, T7, R](t1y: Try[T1], t2y: Try[T2], t3y: Try[T3], t4y: Try[T4], t5y: Try[T5], t6y: Try[T6], t7y: Try[T7])
                                       (f: (T1, T2, T3, T4, T5, T6, T7) => R): Try[R] = for {
  t1 <- t1y
  t2 <- t2y
  t3 <- t3y
  t4 <- t4y
  t5 <- t5y
  t6 <- t6y
  t7 <- t7y
} yield f(t1, t2, t3, t4, t5, t6, t7) // TO BE IMPLEMENTED
```

```scala
// You know this one
def lift[T, R](f: T => R): Try[T] => Try[R] = _ map f // TO BE IMPLEMENTED

/**
  * Lift function to transform a function f of type (T1,T2)=>R into a function of type (Try[T1],Try[T2])=>Try[R]
  *
  * @param f the function we start with, of type (T1,T2)=>R
  * @tparam T1 the type of the first parameter to f
  * @tparam T2 the type of the second parameter to f
  * @tparam R  the type of the result of f
  * @return a function of type (Try[T1],Try[T2])=>Try[R]
  */
// Think Simple, Elegant, Obvious
def lift2[T1, T2, R](f: (T1, T2) => R): (Try[T1], Try[T2]) => Try[R] =  map2(_,_)(f) // TO BE IMPLEMENTED

/**
  * Lift function to transform a function f of type (T1,T2,T3)=>R into a function of type (Try[T1],Try[T2],Try[T3])=>Try[R]
  *
  * @param f the function we start with, of type (T1,T2,T3)=>R
  * @tparam T1 the type of the first parameter to f
  * @tparam T2 the type of the second parameter to f
  * @tparam T3 the type of the third parameter to f
  * @tparam R  the type of the result of f
  * @return a function of type (Try[T1],Try[T2],Try[T3])=>Try[R]
  */
// If you can do lift2, you can do lift3
def lift3[T1, T2, T3, R](f: (T1, T2, T3) => R): (Try[T1], Try[T2], Try[T3]) => Try[R] = map3(_,_,_)(f) // TO BE IMPLEMENTED
```

```scala
// If you can do lift3, you can do lift7
def lift7[T1, T2, T3, T4, T5, T6, T7, R](f: (T1, T2, T3, T4, T5, T6, T7) => R):
(Try[T1], Try[T2], Try[T3], Try[T4], Try[T5], Try[T6], Try[T7]) => Try[R] = map7(_,_,_,_,_,_,_)(f) // TO BE IMPLEMENTED

/**
  * This method inverts the order of the first two parameters of a two-(or more-)parameter curried function.
  *
  * @param f the function
  * @tparam T1 the type of the first parameter
  * @tparam T2 the type of the second parameter
  * @tparam R  the result type
  * @return a curried function which takes the second parameter first
  */
// Hint: think about writing an anonymous function that takes a t2, then a t1 and returns the appropriate result
// NOTE: you won't be able to use the "_" character here because the compiler infers an ordering that you don't want
def invert2[T1, T2, R](f: T1 => T2 => R): T2 => T1 => R = t2 => t1 => f(t1)(t2) // TO BE IMPLEMENTED

/**
  * This method inverts the order of the first three parameters of a three-(or more-)parameter curried function.
  *
  * @param f the function
  * @tparam T1 the type of the first parameter
  * @tparam T2 the type of the second parameter
  * @tparam T3 the type of the third parameter
  * @tparam R  the result type
  * @return a curried function which takes the third parameter first, then the second, etc.
  */
// If you can do invert2, you can do this one too
def invert3[T1, T2, T3, R](f: T1 => T2 => T3 => R): T3 => T2 => T1 => R = t3 => t2 => t1 => f(t1)(t2)(t3) // TO BE IMPLEMENTED
```

```scala
// If you can do invert3, you can do this one too
def invert4[T1, T2, T3, T4, R](f: T1 => T2 => T3 => T4 => R): T4 => T3 => T2 => T1 => R = t4 => t3 => t2 => t1 => f(t1)(t2)(t3)(t4)  // TO BE IMPLEM

/**
  * This method uncurries the first two parameters of a three- (or more-)
  * parameter curried function.
  * The result is a (curried) function whose first parameter is a tuple of the first two parameters of f;
  * whose second parameter is the third parameter, etc.
  *
  * @param f the function
  * @tparam T1 the type of the first parameter
  * @tparam T2 the type of the second parameter
  * @tparam T3 the type of the third parameter
  * @tparam R  the result type of function f
  * @return a (curried) function of type (T1,T2)=>T4=>R
  */
// This one is a bit harder. But again, think in terms of an anonymous function that is what you want to return
def uncurried2[T1, T2, T3, R](f: T1 => T2 => T3 => R): (T1, T2) => T3 => R = (t1: T1,t2: T2) => (t3: T3) => f(t1)(t2)(t3) // TO BE IMPLEMENTED

/**
  * This method uncurries the first three parameters of a four- (or more-)
  * parameter curried function.
  * The result is a (curried) function whose first parameter is a tuple of the first three parameters of f;
  * whose second parameter is the third parameter, etc.
  *
  * @param f the function
  * @tparam T1 the type of the first parameter
  * @tparam T2 the type of the second parameter
  * @tparam T3 the type of the third parameter
  * @tparam T4 the type of the fourth parameter
  * @tparam R  the result type of function f
  * @return a (curried) function of type (T1,T2,T3)=>T4=>R
  */
// If you can do uncurried2, then you can do this one
def uncurried3[T1, T2, T3, T4, R](f: T1 => T2 => T3 => T4 => R): (T1, T2, T3) => T4 => R = (t1: T1,t2: T2,t3: T3) => (t4: T4) => f(t1)(t2)(t3)(t4) /
```

```scala
  */
// If you can do uncurried3, then you can do this one
def uncurried7[T1, T2, T3, T4, T5, T6, T7, T8, R](f: T1 => T2 => T3 => T4 => T5 => T6 => T7 => T8 => R): (T1, T2, T3, T4, T5, T6, T7) => T8 => R =
  (t1: T1, t2: T2, t3: T3, t4: T4, t5: T5, t6: T6, t7: T7) => (t8: T8) => f(t1)(t2)(t3)(t4)(t5)(t6)(t7)(t8)


def sequence[X](xys: Seq[Try[X]]): Try[Seq[X]] = xys.foldLeft(Try(Seq[X]())) {
  (xsy, xy) => for (xs <- xsy; x <- xy) yield xs :+ x
}
```

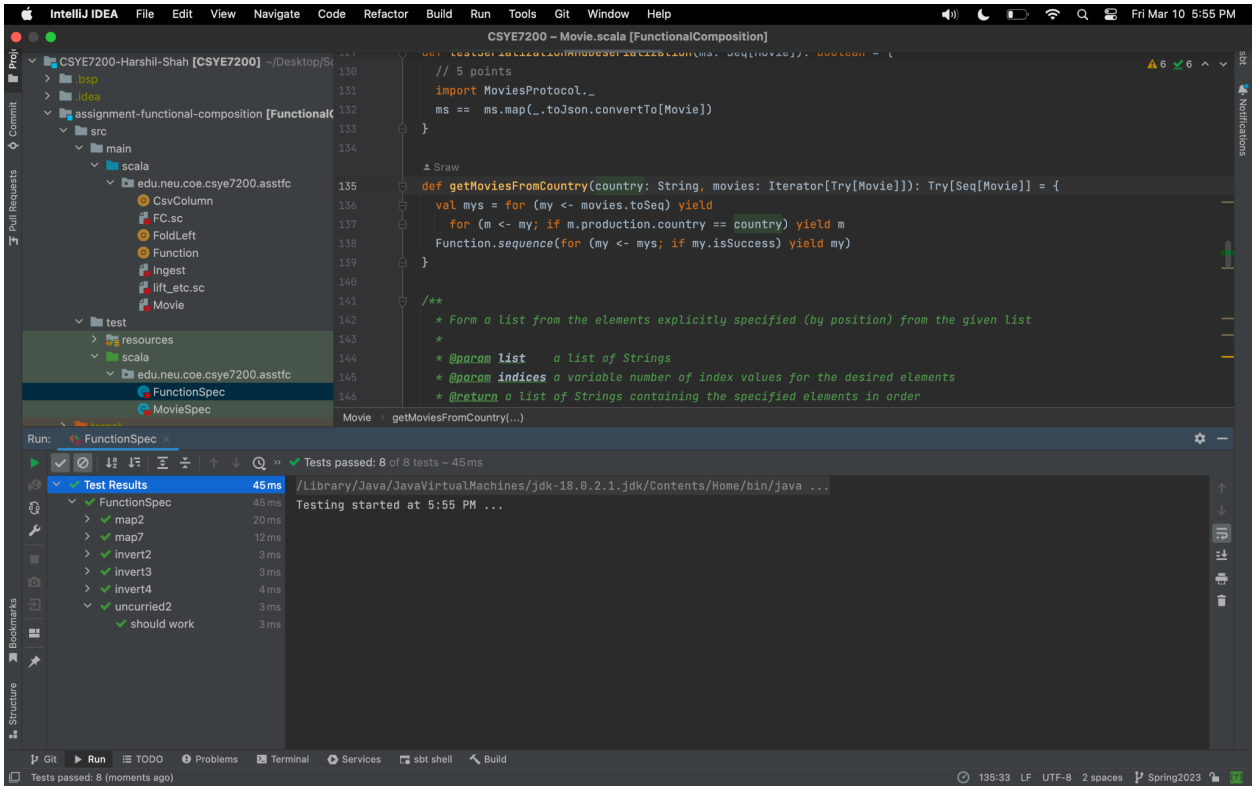
- Movie.scala

```scala
//Hint: You may refer to the slides discussed in class for how to serialize object to json
object MoviesProtocol extends DefaultJsonProtocol {
  // 20 points
  // TO BE IMPLEMENTED
  implicit val nameFormat: RootJsonFormat[Name] = jsonFormat4(Name.apply)
  implicit val ratingFormat: RootJsonFormat[Rating] = jsonFormat2(Rating.apply)
  implicit val formatFormat: RootJsonFormat[Format] = jsonFormat4(Format.apply)
  implicit val productionFormat: RootJsonFormat[Production] = jsonFormat4(Production.apply)
  implicit val principalFormat: RootJsonFormat[Principal] = jsonFormat2(Principal.apply)
  implicit val reviewsFormat: RootJsonFormat[Reviews] = jsonFormat7(Reviews.apply)
  implicit val movieFormat: RootJsonFormat[Movie] = jsonFormat11(Movie.apply)
}
```

```scala
//Hint: Serialize the input to Json format and deserialize back to Object, check the result is still equal to original input.
def testSerializationAndDeserialization(ms: Seq[Movie]): Boolean = {
  // 5 points
  import MoviesProtocol._
  ms ==  ms.map(_.toJson.convertTo[Movie])
}
```

- Unit tests

- FunctionSpec.scala



- MovieSpec.scala