# Final Exam

CSYE7200 37076 Big-Data Sys Engr Using Scala SEC 01 —Spring 2016

## Introduction—important

This exam is quite long. But it isn't really all that hard. I suggest you try to answer all the questions to the best of your ability and *only then* go back and check your results, assuming you have time. I don't want you to get stuck writing long essays or getting every little detail of code right before you go on to the next question. Attempting a question and getting in the "ball park" will get you many more points than leaving a complete blank (which earns a big fat zero).

In all your answers, I urge you to be brief! *Do not write essays*. Do not even write complete sentences unless that's the only way to be clear.

If you really need it, you have up until 9:00 pm (i.e. three hours). But I do hope you'll be finished before then!

## Prelude

One of the tricky things with Scala that most of us Java programmers find problematic, is how to rewrite algorithms that involve mutable state into functional programming (immutable) form. Take for example this simple program:

```
object Count extends App {
  val counter = Counter(10)
  var i = 0
  while ({i = counter.next; i>=0}) {println(i)}
}
case class Counter(var x: Int) {
  def next = {x = x-1; x}
}
```

There are two mutable objects here: *counter* and *i*. The two require slightly different techniques to refactor into a FP form.

---

### *Question 1*

1. Given a (generator) class A[T] with *mutable* state and method *next: T*, show, in general terms, how to refactor it into an *immutable* class *B[U]* that also has a method which you can use to get its next *U* value. Your solution must be referentially transparent. Hint: think back to Week 2 of the class.
2. In general terms, show how to refactor an iterative process using a *var x: Int* into a referentially transparent (FP-style) process. Hint: think back to Week 4 (quick review, part 2)
3. Using both (preferably) of the techniques you describe in 1 and 2, rewrite *Counter* and the *Count* object in a referentially transparent way.

# Assignment 1: Document

You have been asked to work on a data structure that was close to being completed by a colleague who suffered an unfortunate accident. The code that you have been given looks like this:

```
package edu.neu.coe.scala
trait Document[K, +V] extends (Seq[K]=>V) {
  def get(x: Seq[K]): Option[V]
}
case class Leaf[V](value: V) extends Document[Any,V] {
  def get(x: Seq[Any]): Option[V] = x match {
    case Nil => Some(value)
    case _ => None
  }
}
case class Clade[K,V](branches: Map[K,Document[K,V]]) extends
Document[K,V] {
  def get(x: Seq[K]): Option[V] = x match {
    case h::t => branches.get(h) flatMap {_.get(t)}
    case Nil => None
  }
}
```

When you try to compile it you see essentially the same error for both *Leaf* and *Clade*:

> class Leaf needs to be abstract, since method apply in trait Function1 of type (v1: Seq[Any])V is not defined.

---

## *Question 2*

1. Explain succinctly why you are seeing this error.
2. Fix the error in the most simple, obvious and elegant manner which is also consistent with the *scala.collection.immutable.Map* trait.

==========

Part of the Scalatest specification looks like this:

```
class DocumentSpec extends FlatSpec with Matchers {
  "Leaf(1).get" should "yield Some(1) for Nil, None for i" in {
    val doc = Leaf[Int](1)
    doc.get(Nil) should matchPattern { case Some(1) => }
    doc.get(Seq("i")) should matchPattern { case None => }
  }
  "Clade.get" should "work appropriately for one level" in {
    val one = Leaf[Int](1)
    val doc = Clade(Map("one" -> one))
    doc.get(Nil) should matchPattern { case None => }
```

```
    doc.get(Seq("one")) should matchPattern { case Some(1) => }
  }
  it should "work appropriately for two levels" in {
    val one = Leaf[Int](1)
    val doc1 = Clade(Map("one" -> one))
    val doc2 = Clade(Map("a" -> doc1))
    doc2.get(Nil) should matchPattern { case None => }
    doc2.get(Seq("a","one")) should matchPattern { case Some(1) => }
  }
}
```

Unfortunately, although the *Document* module itself is now compiling, the spec file does not. In particular, the expression *Map("one" -> one)* shows the following error:

Multiple markers at this line:

- type mismatch; found : scala.collection.immutable.Map[String,edu.neu.coe.scala.Leaf[Int]] required: Map[Any,edu.neu.coe.scala.Document[Any,Int]] Note: String <: Any, but trait Map is invariant in type A. You may wish to investigate a wildcard type such as `_ <: Any`. (SLS 3.2.10)
- etc……

---

## Question 3

1. Explain succinctly why you are seeing this error.
2. Fix the error in the most simple, obvious and elegant manner (incidentally, this will not be consistent with the *scala.collection.immutable.Map* trait).

==========

---

## Question 4

1. Explain the purpose of this data structure.
2. Describe exactly what is going on in the right-hand-side of the case (in *Clade*) which matches the pattern *h::t*. For what purpose is *flatMap* used here?
3. Suppose you were asked to implement a method in the *Document* trait which has the following signature:

```
def get(x: String): Option[V]
```

such that *x* is split into Strings delimited by ".". For example, *get("a.b")* would be the equivalent of *get(List("a","b"))*.

Describe the components you will need to make this work (I do *not* need actual code). Don't forget that *Document* is defined for keys of type *Seq[K]*—not *Seq[String]*.

==========

# Intermezzo: Spark

---

## *Question 5*

1. Name at least two of the most significant characteristics of *RDD*s.
2. Explain the differences between the two types of operations on an *RDD*: actions and transformations.
3. Give one example of an action and one example of a transformation.
4. Given that, in general, *RDD*s have dependencies on other *RDD*s, how do you think transformations on *RDD*s are implemented. Put your thoughts concisely in words (not code), using a simple method as an example.

==========

---

## *Question 6*

Write a short Spark program to count the number of words in a text file, excluding the following "stop" words: *the, a, an, in, of, by, and, at, by, with*. Assume that you are given a *SparkContext* (as you would be in Zeppelin, for instance). Pseudo-code is acceptable.

==========

# Assignment 2: Bridge Movement

[Remember, you only need to read this to the extent that you need clarification]

Contract bridge is a card game where two pairs of players compete, sitting around a table. Each player is dealt 13 cards and he and his partner try to win “tricks” (each made up of four cards). Bridge can be played for money or for fun. Duplicate bridge is a highly competitive card game in which players play the exact same deals as other players. So that this can be achieved we place the cards after play into a holder with four pockets—one for each player’s cards—these holders are called “boards”. If no boards or players ever moved, it wouldn’t be duplicate, so at least the boards always move. The plan for how boards or players move is called the “movement.” The two most common forms are “pairs” and “teams” but in the normal team form, only the boards move so there is no complexity.

You have been asked to write a Scala program to show how the movement works in a *pairs* game. The pairs are divided (somewhat arbitrarily) into two groups according to which direction they sit: “N/S” (north/south) and “E/W” (east/west). So, a bridge “encounter” involves four “things”: the N/S and E/W pairs, the set of board(s), i.e. deals, which they play and the table at which the play occurs.

You therefore have four corresponding quantities:

- *t*: the table number (0..*T*-1)
- *n*: the N/S pair number (0..*N*-1)
- *e*: the E/W pair number (0..*E*-1)
- *b*: the number of the set of board(s) to be played. (0..*B*-1)

A game is divided (in the time dimension) into “rounds”. For any given round, there will be an encounter at each of the *T* tables. After each encounter, three of these things move by a pre-determined step. The tables (the least movable objects) stay where they are. So, we have the following three quantities:

- *dN*: the number of tables (in a positive direction) which N/S will move
- *dE*: the number of tables (in a positive direction) which E/W will move
- *dB*: the number of tables (in a positive direction) which the boards will move

By convention, *dB* is always -1 in duplicate bridge, but we won’t assume that because we want to write something very general. And in most movements, *dN=0* (the N/S pairs are stationary). The “perfect” bridge session has the following properties: *T = N = E = B = 13; dN = 0, dE = 1, dB = -1*. In this setup, there are two boards in each set, so players play 26 hands of bridge during about 200 minutes of play (this allows for 15 minute rounds, with a short break about halfway).

However, your task is to list the encounters that occur in each round of a more general game. In particular, there is a kind of movement which involves “phantom” tables where the encounters do not count. This is called a “Howell” movement after the Massachusetts native who developed it—without the help of computers—about 100 years ago.

In this movement, dN = -3, dE = -2, dB = -1. If these numbers are co-prime with T then all proceeds smoothly. However, in the case where T=9, the N/S pairs would return to their starting table after three rounds. This requires something different in the movement such as setting *dN = -2 or -4* after three rounds. Depending on how we do this, we might now reencounter an E/W pair from earlier so we might need to change the E/W movement after some number of rounds.

Since we are typically operating on things in threes (pertaining to N/S, E/W, and board respectively) we have chosen to create a type called a *Triple* which extends *Product3:*

```
case class Triple[T](_1: T, _2: T, _3: T) extends Product3[T,T,T] {
  def map[U](f: T=>U): Triple[U] = ???
  override def toString = s"n:${_1} e:${_2} b:${_3}"
}
```

---

## *Question 7*

1. Explain the key differences between a *TupleN* and a *ProductN* (you will most likely want to look them up in the Scala API to answer this question). Hint: think about why we didn't define *Triple* to extend *Tuple3*.
2. Why are the names of the three identifiers in *Triple* chosen as "_1", etc.? What would happen if we simply call them "ns", "ew", "bd"?
3. Implement *map* on *Triple*:

   ```
   def map[U](f: T=>U): Triple[U] = ???
   ```

4. Is *Triple* a monad? Justify your answer.
5. Given, the definition of *Triple* above, would you expect the following code to compile and work?:

   ```
   def toStrings = for (p <- this) yield p.toString
   ```

6. What about this code? Explain why it does or does not compile:

   ```
   def toStrings = for (p <- this; n <- this) yield p.toString
   ```

==========

---

## *Question 8*

1. Why do we need the "type" keyword in Scala?
2. What use is it if we can only define it within a package or class? i.e. what do we need to do to bring it into scope?

==========

We model the moves using a *Stream[Trio]* where *Trio* is defined thus:

```
type Trio = Triple[Int]
```

The source of these moves is a set of three *Seq[Int]* objects, one for each of the three things to be moved. For our situation, we have:

```
val nsMoves = Seq(-3, -3, -2)
val ewMoves = Seq(-2)
val bdMoves = Seq(-1)
```

There are two additional types defined:

```
type MovePlan = Triple[Seq[Int]]
type Moves = Triple[Stream[Int]]
```

---

## *Question 9*

In order to make use of these, we need to write some methods for the *Triple* object:

1. implement the following method to convert a *Triple* of *Stream*s into a *Stream* of *Triple*s:

```
def zip[U](ust: Triple[Stream[U]]): Stream[Triple[U]] = ???
```

2. Implement the following method (presumably invoking your new *map* method) to use the elements each *Seq* provided to populate the corresponding *Stream*:

```
def toStreams[U](ust: Triple[Seq[U]]): Triple[Stream[U]] = ???
```

==========

Another class we will need is *Encounter* defined thus:

```
case class Encounter(table: Int, n: Int, e: Int, b: Int)(implicit tables:
Int) {
  def move(moves: Trio, current: Position): Encounter = {
    val x = moves map {i => current.encounters(modulo(table-i)-1)}
    Encounter.fromPrevious(table,x)
  }
 override def toString = s"T$table: $n-$e@#$b"
}
```

---

## *Question 10*

1. In order to figure out the moves, we need to add a method *modulo* which transforms a number *n* in the range *1-T…infinity* into the range *1..T.* Implement the body of this method (note that we start counting all of our objects that move from *one*, not *zero)*.

```
def modulo(n: Int): Int = ???
```

2. You may not have seen an implicit parameter as part of a case class before but recall that the case class definition essentially just specifies how the constructor looks. Everything else is inferred from that. Why do you think the designer chose to make *tables* implicit while the other four parameters are explicit?
3. How does this kind of implicit value get satisfied?

==========

After the movement (and play) is complete, we need to score the event to see who won. Each board (traditionally) carries with it a “traveler”, a slip of paper that records the scores, pair numbers, etc. at each table. I say traditionally because we usually using an electronic scoring device nowadays. Imagine that we could scan the travelers at the end of the session and input them using a bridge-DSL. Here’s (part of) the code we need (“recap” is the term used for all of the travelers collected together):

```
class RecapParser extends JavaTokenParsers {
 // XXX traveler parser yields a Traveler object and must start with a "T"
and end with a period. In between is a list of Play objects
  def traveler: Parser[Traveler] = "T"~>wholeNumber~rep(play)<~"""."""  ^^
{case b~r => Traveler(Try(b.toInt),r)}
  // XXX play parser yields a Play object and must be two integer numbers
followed by a result
  def play: Parser[Play] = wholeNumber~wholeNumber~result ^^ { case n~e~r =>
Play(Try(n.toInt),Try(e.toInt),r) }
  // XXX result parser yields a PlayResult object and must be either a number
(a bridge score) or a string such as DNP or A+-
  def result: Parser[PlayResult] = (wholeNumber | "DNP" | regex("""A[\-\
+]?""".r) ) ^^ { case s => PlayResult(s) }
}
```

---

## *Question 11*

For these questions, you might find the following link helpful: http://www.scala-lang.org/files/archive/api/2.11.2/scala-parser-combinators/. In your answers, I want *key points*, not *essays*!

1. Explain briefly what "~" signifies. Your explanation should take into account the fact that we see it on both sides of the "^^".
2. Explain briefly what "~>" signifies.
3. Explain briefly what "^^" signifies. Include the type of *wholeNumber*, for example, as part of your explanation.

==========

The way duplicate is scored is that, for any one board, each pair gets two points for every pair they beat and one point for every pair that they tie. So, let's say a board is played 13 times. One pair achieves a better score than all of the other pairs sitting in their direction. They get 24 "matchpoints". This is called a "top." Let's say all of the twelve other pairs get the same result as each other. They each score 11 matchpoints. The total matchpoints for the board adds up to 24 + 12 * 11 which equals 156. This is correct (top * pairs).

The *Traveler* class is defined thus:

```
case class Traveler(board: Int, ps: Seq[Play]) {
  def isPlayed = ps.nonEmpty
   …
}
```

In the *Traveler* class, there is a method to matchpoint a play:

```
  def matchpoint(x: Play): Option[Rational] = if (isPlayed) {
      val isIs = (for (p <- ps; if p != x; io = p.compare(x.result); i <-
io) yield (i,2)) unzip;
      Some(Rational.normalize(isIs._1.sum,isIs._2.sum))
    }
    else None
```

where *Play* is defined thus:

```
case class Play(ns: Int, ew: Int, result: PlayResult) {
  def compare(x: PlayResult): Option[Int] = ??? // if Int, then 0, 1, or 2
  def matchpoints(t: Traveler): Option[Rational] = ???
}
```

---

## *Question 12*

1. What is the type of *isIs*?
2. Explain clearly how *isIs* is evaluated, describing what each component of the expression signifies.
3. Why is there a ";" at the end of the definition? What would happen if we removed it?
4. Can you think of *one* simple way to avoid having ";"? I can think of three, maybe four, ways, one of which is not simple.