Review of Mid-Term Exam

Overall stats for mid-term*

	Poss.	Max	Mean	Std. Dev
Overall	100	91	63	15.1
Q1	22.5	22.5	15.7	3.0
Q2	12	12	9.5	2.8
Q3	8	8	5.0	2.3
Q4	15	15	10	3.75
Q5	10	10	7.4	2.5
Q6	5	5	4	1.6
Q7	4	4	2.2	1.1
Q8	8.5	8.5	4	3.6
Q9	5	5	2.7	2
Q10	10	9**	2.5	3.0

* Before late deductions

** Only Q w/o perfect answer

Mid-term Review: Q1

Scala is a **functional** programming language which is object-oriented, statically **typed** and runs on the **JVM**. It is, therefore, ideally suited for solving Big Data problems. One of the most elegant features is the clear distinction between eager and **lazy** evaluation allowing, for example, the definition of **Streams**, which are essentially Lists with a lazily evaluated **tail**. Functions are first-class **objects** and can be easily composed into new functions. So it's quite possible to "decorate" datasets with composite functions rather than applying each **function** one-by-one. This makes it ideal for **parallel** processing because you can delay the **evaluation** (by remote parallel execution) of such a transformed dataset, until the user really needs to see one or more elements, thus reducing unnecessary remote-job initiation overhead. This idea is the basis of Spark, the cluster-computing framework. Another important feature of Scala is the use of *implicits*. The ability to define local variables or methods--which modify the **behavior** of library functions--is a key enabler for Big Data-required functionality like serialization/deserialization, especially that of **JSON**. One of the biggest enemies of a successful cluster-computing framework (like Spark) is the throwing of **exceptions**. Scala's use of ***Try*** and *Option* are instrumental in avoiding these kinds of interruption. And, of course, you couldn't have a cluster-computing framework without the ability to invoke asynchronous calls. Scala's **Future** type is the key to such calls.

Mid-term Review: Q2

A. `val x = 1`: Within the context of a block, define an identifier (i.e. variable) *x* such that, for the remainder of the current scope, *x* will be identical, semantically, with 1.

B. `x <- List(1, 2, 3)`: Within the context of a for-comprehension, match the pattern *x* with successive values from the list, i.e. 1, 2, and 3; and, for the remainder of the scope of the for-comprehension, make *x* available as a variable.

C. `x: => Int`: Within the context of a method's parameter set, define a call-by-name parameter *x* of type *Int*.

D. `x: Int => x+1`: Within the context of an anonymous 1-parameter function or case clause of a match (i.e. a *lambda*), define a variable *x* of type *Int*. which will be bound to the input parameter of the function or will be matched to the target of the match, providing it is of type *Int*.

E. `x = 1`: Within the context of a for-comprehension, define the variable *x* such that for the remainder of the scope of the for-comprehension, *x* will be identical, semantically, to 1.

F. `x -> 1`: A 2-tuple whose components are the variable *x* and the constant 1.

Mid-term Review: Q3

A. `val x = x`: Valid syntax but will cause compiler error.

B. `val x: Int = x`: Valid syntax but semantically meaningless (compiler warning).

C. `val x: Stream[BigInt] = 0 #:: 1 #:: x.zip(x.tail).map(p => p._1 + p._2)`: Valid, syntactically and semantically.

D. `def x: Int = x + 0` Valid, semantically and syntactically, but will cause a stack overflow on any actual computer system if you actually reference *x*.

Mid-term Review: Q4

- Explain the following code: `val x: Stream[BigInt] = 0 #:: 1 #:: x.zip(x.tail).map(p => p._1 + p._2)`
- Describe in no more than six short sentences the meaning (semantics) of this code. In particular, mention the following syntactic elements:
  - `#::`
  - `zip`
  - `tail`
  - `map`
  - `p =>`
  - `._1`
- If you are unsure of some of the meanings, you may use the REPL to help.

Q4: Analysis

Lots of variation in the answers to this question. Many students were very confused about this. It's hard for me to understand why because you had the opportunity to use the REPL to help with things. There was a lot of credit given for relatively bad answers. For a perfect score you needed to make all of the points here (or the equivalent).

Did you notice by the way that *x* is a *val* (and not a *def*)? It can still be declared recursively.

- Overall, x is a Stream of BigInt representing the Fibonacci series (bonus point). It is a recursive definition (bonus point)
  - #:: — "Cons" or concatenation operator for Streams;
  - zip — Takes two Streams and yields a Stream of Tuple2s;
  - tail — yields *this* Stream, but without the head;
  - map — higher order method which takes a function as parameter and applies that function to each element of *this* in turn, yielding a new Stream;
  - p => — definition of the bound variable (formal parameter) which takes the value of the input in the resulting lambda (function literal);
  - ._1 — method to yield first field (element) of a Tuple.

Mid-term Review: Q5

1. Recursion is an important technique for the implementation of many mathematical concepts. T
2. Recursion can cause a stack overflow even when an algorithm is operating correctly. T
3. Tail-recursion is a technique which avoids stack the prevents a stack overflow. T
4. The compiler can recognize tail-recursive situations on its own but you can ensure that your code is tail-call optimized by adding the *@tailrec* annotation. T
5. Recursion is good for mathematical definitions but it's a bad idea to use it in a programming language because it is inefficient. F
6. A successful recursion must have a valid test for the terminating condition. T
7. You do not need to specify the type of a variable or method which is invoked recursively because the compiler is able to infer the type. F

Mid-term Review: Q6

NOTE: this and the following questions refer to the code in *lab-sorted.*

- Why is *Different* declared as a case *class* while *Same* is declared as a case *object*?
- Because only a case class can have fields (the use of the term *class* relates to the fact there can be any number of objects in the class: each with different values of the fields). However, a case object (without fields) must be a singleton for the simple reason that there can only be one instance.

Q6 continued

- This brings up a more general issue: Classes and Objects.
- In O-O, a “class”, as in mathematics in general, refers to a the set of all possible instances of something. The definition of the “class” defines what is constant among the (potential) instances and what can vary. So, a class with only one field (a byte) can occur in exactly 256 different instances. For most classes, the number of possible instances is huge.
- In Scala, an Object (with upper-case “O”) refers to a singleton instance of a class, which is also called the “companion” object. Some “classes” have no variability and so these are defined as objects.
- The companion object has all of the non-instance fields and methods. In Java, we would mark these as “static” and call them “class” fields and methods.

Q7

- In trait *Comparer*, you will notice that the first line is "self =>". We have not yet covered that in class. What do you think "self" means here and why do you think it is necessary? Regard particularly the *compare* method of the new *Comparer* definition within the *apply* method.
- *self* is used to allow expressions inside an inner class to refer to the "this" of an outer class. Java has a similar mechanism (involving the keyword "outer") which is not quite so clear as the use of *self*.

Q8

```scala
trait Comparison extends (() => Option[Boolean]) {
  def toInt: Int = ???
  def orElse(c: => Comparison): Comparison = Comparison(apply.orElse(c()))
  def flip: Comparison = ???
}
```

There are several reasonable ways to code this depending on what you want to use as a component. There is an art of object-oriented and functional programming where you should choose the most relevant code, provided that it does not obscure the meaning (SOE principle). Here, we know that we want to create an instance of *Comparison* which will be either *Different(x)* or *Same*. We already have code that distinguishes these: it's the *apply* method of the companion object and it takes as parameter an *Option[Boolean]*. To get the appropriate value, we know we'll need *this Comparison* and we have *c*. We can get *this* as an *Option[Boolean]* by calling *apply*. If that is *Same*, then we want to use *c*. Hence the use of the *orElse* method of *Option*.

Q9

Suppose you want to add a unit test for sorting *Char* as followed:

```
it should "sort List[Char]" in {
  val list = List('b', 'c', 'a')
  val sorted = Sorted(list)
  sorted() shouldBe List('a', 'b', 'c')
}
```

- Will it work? If it works explain how it works, if not please write the code which can fix this. Also include where (in which class/object) you put your code.
- Add `implicit val charComparer: Comparer[Char] = Ordering[Char]` in companion object *Comparer*.
- Or, you could simply add the same statement right here above the *val sorted…* line.

Q10 (Hard, bonus question)

There was no perfect answer for this tough question. In fact, to be honest, nobody really got close. But many (most) of you didn't answer the question at all. Instead you said things that were true generically *of Scala vs. Java but were not relevant to this particular situation. I did give one 9 for this question because the student made some very relevant points.*

In the description of this project, you were told that the new API for comparisons was more *functional* than the Java-style API. What do you think the author meant by *more functional*? Why can't you do the same thing with the Java-style? What will you write in the documentation about this aspect of the new API? Hint: pay special attention, again, to the *apply* method inside the *compose* method of *Comparer*.

This API is more functional because we can *compose Comparers*. That's because the result of *apply* is one of two cases (not almost infinitely many values as in the Java result). This allows for an *orElse* type of method to be employed. Either we were able to discriminate (*Different*) in which case that's our result, or else we try the other *Comparer*.
If you think about the way this has to be done in Java, you would have to write something like:

```
int x = compare(t1,t2); if (x!=0) return x else return o.compare(t1,t2)
```

Note the semi-colon (extra statement) in the Java version. This isn't so bad for just two comparisons, but it can get quite ugly for three or more.