Fall 2018 Midterm Solutions for Q8 & Q9 (asst-cache module)

Q.8 Cache.scala

```
case class MyCache[K, V](fulfill: K=>Future[V]) extends Cache[K, V] {

  private def put(k: K, v: V): Unit = cache.+=((k,v))

  override def apply(k: K): Future[V] = if (cache.contains(k)) Future(cache(k)) else for (v <- fulfill(k); _ = put(k, v)) yield v

  def expire(k: K): Unit = cache.-=(k)

  val cache: mutable.Map[K, V]  = mutable.Map.empty

  def empty: Unit = cache.empty
}
```

The apply method of an immutable List gives you the element at an index. Similarly, the apply method of MyCache should give the value (Future[V]) for the corresponding key k. MyCache internally uses a mutable Map to store key-value pairs. So, the apply method will return the value for key k from the mutable Map. If the key-value pair is not present in the Map, it will generate the value using the 'fulfill' constructor parameter.

Q.9 Portfolio.scala

Position case class is designed holds a stock symbol and its quantity. For eg, Position('MSFT', 100) would mean 100 stocks of MSFT (Microsoft).

Portfolio case class is designed to hold a sequence of Position, which is to say it holds several stock symbols and their corresponding quantities.

Also, here an instance of MyCache is used to store the stock symbol as key and its price as value.

```
case class Portfolio(positions: Seq[Position]) {

  def value(cache: Cache[String, Double]): Future[Double] = {
    val xfs = for (p <- positions) yield for (v <- p.value(cache)) yield v
    for (xs <- Future.sequence(xfs)) yield xs.sum
  }

}
```

Having said the above, value method of Portfolio (shown above) should give the sum of values of all the Position instances that it holds.

```
case class Position(symbol: String, quantity: Double) {
  def value(cache: Cache[String, Double]): Future[Double] = for (v <- cache(symbol)) yield v * quantity
}
```

The value method of Position case class (shown above) should give the value of the stock that it holds. The price of the stock multiplied by quantity would give the value. Note, here a Cache[String, Double] is used to get the price (value) for a stock symbol (key).

```
object Position {
  val positionR = """(\w+)\s+(\d+(\.\d+))""".r
  def parse(w: String): Try[Position] = w match {
    case positionR(a, b, _) => Try(Position(a,b.toDouble))
    case _ => Failure(new Exception(s"cannot parse $w as a Position"))
  }

  def value(cache: Cache[String, Double])(w: String): Future[Double] = flatten(for (p <- parse(w)) yield p.value(cache))

  private def flatten[X](xfy: Try[Future[X]]): Future[X] =
    xfy match {
      case Success(xf) => xf
      case Failure(e) => Future.failed(e)
    }

}
```

The value method of object Position (shown above) takes a parameter w of type String. If you look at the test case for this, you will notice w is something like “MSFT 100.0”. Now, we have been given a method parse that takes a String and returns a Try[Position]. Therefore, we do *parse(w)* to get this Try[Position], which will hold symbol = “MSFT” and quantity = 100.0

The value method should return the value of this Position instance. Therefore, we do *p.value(cache)* to get the Position’s value as Future[Double]. The *flatten* is used to bring this value into appropriate shape.

Q1.

The following are all good reasons to learn Scala -- except for one.

A. Spark is written in Scala.
B. Functional composition adds important capabilities to object-oriented languages, especially when it comes to running programs in parallel on possibly remote machines.
*C .Scala is the only functional language available on the Java Virtual Machine
D. Scala is strictly typed which allows the compiler to deduct more problems with your code before you even start debugging.

Q2.

```
line1: val h = 1
line2: val t = List(2,3)
line3: val ht = h :: t
line4: ht.reverse match {
line5:       case h :: _ => println(h + 1)
line6: }
```

A.What is the type and value of val ht?
ht is a List[Int], value is List(1,2,3)
B.What is the result of println in line 5?
4
C.There are two "::" in the code, are they the same thing?
No.
D.If they are the same, explain the meaning of it. If not what is the difference between "::" in line 3 and in line 5?
The "::" in line 3 is a method (of *List*), in line 5 is a case class which extends *List*.
E.What compiler-generated method is called to yield an actual value for h in line 5? [A]
unapply
F.The h in line 1 is said to be [B] by the h in line 5.
shadowed
G.Desugar the "::" in both lines.
line 3: t.::(h); line 5: ::(h,t)

Note that all of these answers are readily available in your IDE (not so much the REPL though). If, for example, you hover over the second *h*, you will see a message about shadowing.

Q3.

Your company is developing a service to check the balance of BitCoin accounts by address using Spark.

The original code is:

```
val input = sc.parallelize(inputData).filter(isValid)
```

```
val rdd = input.map(getBalance)
val result = rdd.collect()
```
During the Beta Test, you discover that some input files don't contain any valid addresses.

The rules for a valid BitCoin Address are:

1. A BitCoin address is between 25 and 34 characters long;

2. The address always starts with a 1;

3. An address can contain all alphanumeric characters, with the exceptions of 0, O, I, and l.

Here is an example of a valid BitCoin Address: 1AGNa15ZQXAZUgFiqJ2i7Z2DPU2J6hW62i

One of your coworkers who is an expert in Java but is new to Scala believes it can be"sped up" by saving the work of the *map* method thus:

```
val input = sc.parallelize(inputData).filter(isValid)
val rdd = if (!input.isEmpty) input.map(getBalance) else input
val result = rdd.collect()
```

The following methods are provided to you:

```
def isValid(x:String):Boolean = !(!x.startsWith("1") | x.contains("0") | x.contains("O") | x.contains("I") | x.contains("l") | x.length>34 | x.length<25)

def getBalance(x:String) = {
   //invoke database query to get balance
}
```

A.Is this a good idea?
No.
B. Explain why or why not?
RDD is lazy; isEmpty is an Action
C. How can you support your opinion with hard evidence? What is your test data, how to test it, how to compare the result.
Using System.nanoTime() or similar to record the time and compare the execution time; Compare it with a non-lazy collection: e.g. if input is a list than it is faster. Test data must include different conditions like empty input, all invalid address, a mix of valid and invalid address.

Q4.
Which of the following statements are true with regard to the following method?

```
def condition: Parser[RuleLike] = regex("""\w+""".r) ~ predicate ^^ { case s ~ p => Condition(s, p) }
```

*It will match exactly the concatenation of two strings: an identifier made up of at least one "word" characters (\w) and a string that must be parsed exactly by the parser yielded by method predicate.

The ^^ symbol is a logic operator which takes the "exclusive or" of two parsers.

*The result of this method is a parser which yields an instance of RuleLike.

*In the partially defined function which appears on the right hand side, the tokens s ~ p match a case class called "~" with two fields.

Q5.

Please ensure that you have the latest code from Majabigwaduce (by doing a pull). Open the project in your IDE, although you should be able to answers these questions without using your IDE, using it might be easier. If you have any difficulties, you can simply look at the code using your browser on the repository.

Majabigwaduce is a relatively simple framework for doing remote parallel computations using Akka actors. It follows the Map-Reduce paradigm of other systems, providing a low-level applications programming interface based on Mappers and Reducers similar to Hadoop's Java API. The CountWords example uses that API.

There is, in addition, a higher-level API which is similar to the way Spark works, in that the programmer creates a lazy, partitionable dataset and manipulates it using transformations and actions, similar to Spark's RDD. The in this case is called DataDefinition.

It has three important methods:

```
def apply(): Future[Map[K, V]]
def map[L, W: Monoid](f: ((K,V)) => (L,W)): DataDefinition[L, W]
def aggregate[W: Zero](f: (W, V) => W): Future[W]
```

The map method simply returns another DataDefinition. The other two return zero- or one-dimensional collections.

Please ensure that you have looked at all parts of this question. Make sure you label each of your answers with the correct part letter. As always, you are to provide brief answers that match what I'm looking for. No extra credit for writing essays!!

A. Which of the three methods can be considered a "transformation" (as opposed to an "action")?
map
B. The signatures of map and aggregate each include something of the form "[W: X]." What is W: X ?
It is a Context Bound
C. The signatures of map and aggregate each include something of the form "[W: X]." What is X?
It represents a Typeclass

D. Can you explain--briefly--what this "[W: X]" actually means in practical terms? or why the author uses it here?
We need an implicit value of type X[W] passed as evidence into the method.
E. Why do you think the author chose to use this mechanism rather than having W simply extend another type? This part is somewhat harder than the others.
It allows introducing new functionality without having to be able to extend some other class.
F. There is a method with a "negative-one-dimensional" result missing: that would be count. It should have a signature like *def count: Future[Int]*. Implement this method for *LazyDD* in a manner similar to the implementation of *aggregate*.
def count:Future[Int] = for (kWm <- apply()) yield kWm.size
G. In the definition of case class *LazyDD*, there is a third parameter set marked implicit. This value (context) is not passed in explicitly by any invocation of *LazyDD's apply* method. So, how does it get satisfied?
There is an implicit val context: DDContext = DDContext.apply in Object DataDefinition
H. Also, in the definition of *LazyDD*, there is a curious extra set of parentheses around (K,V). Can you explain why those are necessary (hard).
Without the extra set of parentheses, the compiler will think of *f* as a Function2 (with two input K and V) but we want it to be a Function1 with one input of type Tuple(K,V)
I. Can you think of another important type of transformation that the author has omitted? Note that, as a transformation, it must yield a *DataDefinition*, but it is not something that could be implemented by invoking map with an appropriate value for f. One more clue: it is not *flatMap*.
def filter(f: ((K,V)) => Boolean): DataDefinition[K,V]
J. Implement your transformation method for the previous question.
def filter(g: ((L, W)) => Boolean):DataDefinition[L,W] = LazyDD[K,V,L,W](kVm.filter(f andThen g), f)(partitions)

Coding Q1.

```
import java.io.PrintWriter
object Solution {

  //Stream from 1
  val a = Stream.from(1)
  //Stream from 2
  val b = Stream.from(2)
  //Tuple of two stream
  val tab = (a,b)

  //Declare a Stream of 'a' to 'z', repeatedly
  val c = Stream.continually('a' to 'z').flatten //or Stream.from(0).map(x => (x % 26 +'a').toChar)
//TODO 1, replace it with your own implementation, if you don't know how to implement it just leave it as Stream.Empty

  //Convert Tuple of two stream to Stream of tuple2
  def zip2[U,V](t:(Stream[U],Stream[V])):Stream[(U,V)] = t._1 zip t._2 //TODO 2, replace it with your
```

own implementation, if you don't know how to implement it just leave it as Stream.Empty

```
  //Define a method zip3, similar to zip2, Convert Tuple of three stream to Stream of tuple3
  def zip3[U,V,W](t:(Stream[U],Stream[V],Stream[W])):Stream[(U,V,W)] = (t._1 zip t._2 zip t._3).map(x => (x._1._1,x._1._2,x._2)) //TODO 3, replace it with your own implementation, if you don't know how to implement it just leave it as Stream.Empty

  //Tuple of three stream
  val tabc = (a,b,c)
  //Tuple of three stream
  val taba = (a,b,a)

    def main(args: Array[String]) {
        val scan = scala.io.StdIn
        val fw = new PrintWriter(sys.env("OUTPUT_PATH"))
        val l = scala.io.StdIn.readInt
        l match {
            case 0 => fw.println("Compiled")
            case 1 => fw.println(c.take(3).mkString(","))
            case 2 => fw.println(c.slice(25,28).mkString(","))
            case 3 => fw.println(zip2(tab).take(3).mkString(","))
            case 4 => fw.println(zip2(tab).slice(1000,1002).mkString(","))
            case 5 => fw.println(zip3(taba).take(3).mkString(","))
            case 6 => fw.println(zip3(taba).slice(25,28).mkString(","))
            case 7 => fw.println(zip3(tabc).slice(25,28).mkString(","))
        }
        fw.close()
    }

}
```

Coding Q2.

Implement a depth-first-search method using tail recursion. You are given a trait *BinaryTree* with two case classes: *Node* and *Empty*. A *Node* has a value and two child nodes, marked *left* and *right*. In a binary tree, a node's left sub-tree contains values which all compare less-than the node's value, while the right sub-tree contains values which all compare greater-than-or-equal to the node's value. Depth-first search, as opposed to breadth-first-search, is a method of tree (or acyclic graph) traversal in which we recurse through the sub-trees before dealing with the node itself. Note that, in the case of a binary tree, we could traverse it in value-order by recursing into the left sub-tree, then dealing with the value, then recursing into the right sub-tree. But that's not what is being asked for here, and so not how the unit tests are specified.

The *depthFirst* method takes a function *f: T=>U* and will apply that function to every value in the order determined by the DFS traversal.

Recall that when we create a tail-recursive inner method, we usually have two parameters: one represents the work yet to be done, the other represents the work that has been done.

```
import java.io.PrintWriter
import scala.annotation.tailrec
object Solution {

    sealed trait BinaryTree[+T] {
      def depthFirst[U](f: T=>U): Seq[U]
    }
    case class Node[+T](value: T, left: BinaryTree[T], right: BinaryTree[T]) extends BinaryTree[T] {
      override def toString: String = "T(" + value.toString + " " + left.toString + " " + right.toString +
")"
      def depthFirst[U](f: T => U): Seq[U] = {

    // TODO implement me, using a tail-recursive inner method, you may refer to the method
depthFirst above to find what parameter you may use and what is the return type expected
        @tailrec def inner(todo: Seq[BinaryTree[T]], done: List[T]): Seq[U] = todo match {
          case Nil => done map f
          case h::t => h match {
            case Node(v,l,r) => inner(r::l::t, v::done)
            case Empty => inner(t, done)
          }
        }

    // TODO call inner method with appropriate parameter
        inner(Seq(this),Nil)
        }
}

    case object Empty extends BinaryTree[Nothing] {
      override def toString = "."

      def depthFirst[U](f: Nothing => U): Seq[U] = Nil
    }

    object Node {
      def apply[T](value: T): Node[T] = Node(value, Empty, Empty)
    }

    def main(args: Array[String]) {
        val scan = scala.io.StdIn
        val fw = new PrintWriter(sys.env("OUTPUT_PATH"))
```

```
        val l = scala.io.StdIn.readInt
        l match {
            case 0 => fw.println("Compiled")
            case 1 => fw.println(Node("a",Node("b", Node("d"), Node("e")),Node("c", Empty,
Node("f", Node("g"), Empty))).toString)
            case 2 => fw.println(Node("a").depthFirst(identity).toString())
            case 3 => fw.println(Node("a",Node("b"),Empty).depthFirst(identity).toString())
            case 4 => fw.println(Node("a",Node("b"),Node("c")).depthFirst(identity).toString())
            case 5 => fw.println(Node("a",Node("b", Node("d"), Node("e")),
      Node("c", Empty, Node("f", Node("g"), Empty))).depthFirst(identity).toString())

            case 6 => {
                val sb = new StringBuilder
                Node("a").depthFirst(s => sb.append(s))
                fw.println(sb.toString())
            }
            case 7 => {
                val sb = new StringBuilder
                Node("a",Node("b"),Node("c")).depthFirst(s => sb.append(s))
                fw.println(sb.toString())
            }
            case 8 => {
                val sb = new StringBuilder
                Node('a',
                Node('b', Node('d'), Node('e')),
                Node('c', Empty, Node('f', Node('g'), Empty))).depthFirst(s => sb.append(s))
                fw.println(sb.toString())
            }

        }
        fw.close()
    }
}
```

Final Exam CSYE 7200 Spring 2018

1. **Multiple Choice: Why should someone learn Scala?: The following are all good reaso...** Points: 4

Question

The following are all good reasons to learn Scala -- except for one.

Answer

A. Spark is written in Scala.

B. Functional composition adds important capabilities to object-oriented languages, especially when it comes to running programs in parallel on possibly remote machines.

✓ C. Scala is the only functional language available on the Java Virtual Machine

D. Scala is strictly typed which allows the compiler to deduct more problems with your code before you even start debugging.

2. **Essay: Code Comprehension: match: line1: val h = 1line2: val t = List(2...** Points: 5

Question

```
line1: val h = 1
line2: val t = List(2,3)
line3: val ht = h :: t
line4: ht.reverse match {
line5:     case h :: _ => println(h + 1)
line6: }
```

Questions:

A. [3] What is the type and value of val *ht*?
B. [2] What is the result of *println* in line 5?

Answer

A. Type List[Int], value List(1,2,3)

7. **Essay: BitCoin Question Continued: Continued from the previous Ques...** Points: 13

Question

Continued from the previous Question

A. [5] Explain why or why not?
B. [8] How can you support your opinion with hard evidence? What is your test data, how to test it, how to compare the result.

Answer

A. It's not a good idea because size/length are actions which require completely evaluating the RDD. An empty RDD will take care of itself and not require many resources to evaluate it when the time comes.

B. You will perform a benchmark running both versions of the software many times and timing the results using the *nanos* method in System.

B. the first **::** refers to the case class **::** which extends *List*; the second is syntactic sugar for the *unapply* method of *List*'s companion object.

8. **Multiple Answer: Parsing: Which of the following statements are...** Points: 6

Question

Which of the following statements are true with regard to the following method?

```
def condition: Parser[RuleLike] = regex("""\w+""".r) ~ predicate ^^ { case s ~ p => Condition(s, p) }
```

Answer

✓ It will match
exactly
the concatenation of two strings: an identifier made up of at least one "word" characters (
\w
) and a string that must be parsed
exactly
by the parser yielded by method
predicate
.

The ^^ symbol is a logic operator which is takes the "exclusive or" of two parsers.

✓ The result of this method is a parser which yields an instance of
RuleLike
.

✓ In the partially defined function which appears on the right hand side, the tokens
s ~ p
match a case class called "~" with two fields.

9. **Multiple Answer: Scala code comprehension and enhancement: Please ensure that you have the lates...**

Points: 4

Question

Please ensure that you have the latest code from Majabigwaduce (by doing a pull). Open the project in your IDE, although you should be able to answers these questions without using your IDE, using it might be easier. If you have any difficulties, you can simply look at the code using your browser on the repository.

Majabigwaduce is a relatively simple framework for doing remote parallel computations using Akka actors. It follows the Map-Reduce paradigm of other systems, providing a low-level applications programming interface based on Mappers and Reducers similar to Hadoop's Java API. The *CountWords* example uses that API.

There is, in addition, a higher-level API which is similar to the way Spark works, in that the programmer creates a lazy, partitionable dataset and manipulates it using transformations and actions, similar to Spark's *RDD*. The in this case is called *DataDefinition.*

It has three important methods:

- `def apply(): Future[Map[K, V]]`
- `def map[L, W: Monoid](f: ((K,V)) => (L,W)): DataDefinition[L, W]`
- `def aggregate[W: Zero](f: (W, V) => W): Future[W]`

The *map* method simply returns another *DataDefinition*. The other two return zero- or one-dimensional collections.

Please ensure that you have looked at all parts of this question. Make sure you label each of your answers with the correct part letter. As always, you are to provide brief answers that match what I'm looking for. No extra credit for writing essays!!

Question:

Which of the three methods can be considered a "transformation" (as opposed to an "action")?

Answer

A. apply

B. map

C. aggregate

10. **Multiple Choice: Scala code comprehension and enhancement continued: Continued from the previous Question ...** Points: 3

Question	Continued from the previous Question The signatures of map and aggregate each include something of the form "[W: X]." What is ***W: X*** ?
Answer	✓ A. Context Bound B. Upper Bound C. Lower Bound D. View Bound

10. **Multiple Choice: Scala code comprehension and enhancement continued: Continued from the previous Question ...** Points: 3

Question	Continued from the previous Question The signatures of map and aggregate each include something of the form "[W: X]." What is ***W: X*** ?
Answer	✓ A. Context Bound B. Upper Bound C. Lower Bound D. View Bound

11. **Multiple Choice: Scala code comprehension and enhancement continued: Continued from the previous Question ...** Points: 3

Question	Continued from the previous Question The signatures of map and aggregate each include something of the form "[W: X]." What is ***X***?
Answer	✓ A. Typeclass B. Covariance C. Contravariance D. Invariance

12. **Short Answer: Scala code comprehension and enhancement continued: Continued from the previous Question ...** Points: 8

Question	Continued from the previous Question A. [3] Can you explain--briefly--what this "[W: X]" actually means in practical terms? or why the author uses it here? B. [5] Why do you think the author chose to use this mechanism rather than having *W* simply extend another type? This part is somewhat harder than the others.
Answer	A. It means that there is an implicit variable available of type *X[W]* that can be used to implement the methods of *X[W]*. A less elegant-looking form of this would be simply to pass in a parameter of type *X[W]*. B. It's used here because it is not convenient to redefine *W* to support the required methods possibly because *W* is a (third-party) library and/or final class.

13. **Short Answer: Scala code comprehension and enhancement continued: Continued from the previous Question ...** Points: 8

Question	Continued from the previous Question There is a method with a "negative-one-dimensional" result missing: that would be *count*. It should have a signature like `def count: Future[Int]`. Implement this method for *LazyDD* in a manner similar to the implementation of *aggregate*.
Answer	`for (kVm <- apply()) yield kVm.size`

14. **Short Answer: Scala code comprehension and enhancement continued: Continued from the previous Question ...** Points: 9

Question	Continued from the previous Question A. [4] In the definition of *case class LazyDD*, there is a third parameter set marked *implicit*. This value (*context*) is not passed in explicitly by any invocation of *LazyDD*'s apply method. So, how does it get satisfied? B. [5] Also, in the definition of *LazyDD*, there is a curious extra set of parentheses around *(K,V)*. Can you explain why those are necessary (hard).
Answer	A. When the compiler has to look for an implicit, it first looks in the current scope, then in the object of one of a relevant type. Here, inside object *DataDefinition*, an implicit context : *DDContext* is defined. B. Because *(K, V)* is of a *Tuple*.

15. **Short Answer: Scala code comprehension and enhancement continued: Continued from the previous Question ...** Points: 5

Question	Continued from the previous Question Can you think of another important type of transformation that the author has omitted? Note that, as a transformation, it must yield a *DataDefinition*, but it is not something that could be implemented by invoking *map* with an appropriate value for *f*. One more clue: it is not *flatMap*.
Answer	*filter*

16. **Short Answer: Scala code comprehension and enhancement continued: Continued from the previous Question ...** Points: 5 (Extra Credit)

Question	Continued from the previous Question [5 bonus points] Implement your transformation method for the previous question.
Answer	`def filter(p: ((L, W)) => Boolean): DataDefinition[L, W] = LazyDD[K, V, L, W](kVm.filter(f andThen p), f)(partitions)`

Q4:

```
  def hashCode(p: Product): Int = p.productIterator.foldLeft(1)
(_*31+hashIt(_))
  def equals(p: Product, q: Product): Boolean = q.canEqual(p) &&
p.productArity==q.productArity && (p.productIterator zip
q.productIterator).forall(x => x._1==x._2)
  private def hashIt(x: Any): Int = x match {
    case p: Product => hashCode(p)
    case null => 0
    case q => q.hashCode()
  }
```

Please note you can also implement equals as follows:

```
   def equals(p: Product, q: Product): Boolean = q.canEqual(p) &&
 p.productIterator.sameElements(q.productIterator)
```

This is a very elegant solution, although it doesn't use a higher-order function as requested.

Q8:

make the method parameter a Super type of original type, like following:

```
trait Orderable[+X] { def order[Y >: X](x1: Y, x2: Y): Y }
```

Spring 2017: Final Exam

1. **Multiple Choice: Question 1: Scala and Java are interoperable ...**

Points: 6

Question	Scala and Java are interoperable to some extent. Choose from the following the best match (there will be partial credit for appropriate answers)
Answer	✓ A. Java classes can be referenced from Scala and Scala classes can be referenced from Java. There are a few incompatibilities to be aware of such as collections, abstract classes, interfaces/traits.
	B. Like Groovy, Scala programs essentially *are* Java programs so there are no incompatibilities.
	C. Both Scala and Java programs are hosted on the Java Virtual Machine but that's as far as interoperability goes. You cannot reference Java classes from a Scala classes.
	D. Scala and Java classes are interoperable but there are so many incompatibilities that programmers normally avoid calling Java from Scala or the other way around. The biggest problem is that Java uses "primitives" like *int*, *boolean* whereas in Scala everything is an object like *Int*, *Boolean*.
Correct Feedback	Scala and Java are interoperable with the few relatively minor exceptions noted. Scala provides a set of implicit converters to/from Java/Scala collections which make using collections really easy.
Incorrect Feedback	Scala and Java are interoperable with few relatively minor exceptions. Scala provides a set of implicit converters to/from Java/Scala collections which make using collections really easy. It's true that Scala uses objects (*Int*, *Boolean*) whereas Java uses primitives (*int*, *boolean*) but these are taken care of by the Scala compiler through the Java mechanism called "boxing."

2. **Multiple Choice: Question 2: Recursion is a mathematical technique...**

Points: 6

Question	Recursion is a mathematical technique that has some issues when programming that you need to be aware of. Choose the best description.
Answer	✓ 1. By explicitly using tail recursion, a Scala programmer can guarantee that a recursive function will not exhaust the stack.
	2. Scala implements recursion more efficiently than does Java, so you are less likely to run out of memory with Scala.
	3. Recursion is a nice way to describe things mathematically, but it should never be used in a program because it is too easy to fill the stack and throw an exception.
	4. Recursion can never be implemented as efficiently as iteration so you should always code recursive functions as iterations, whatever language you are using.

4. Short Answer: HashCode and Equals: As you know, it's essential to ensure...

Points: 18

Question

As you know, it's essential to ensure that the *hashCode* and *equals* methods for any class are compatible. Otherwise, when you try to create a *Set* or *HashMap* based on such a class (i.e. with incompatible methods), the collections logic will go wrong and results will be unpredictable.

Here is an implementation of *hashCode* for a class with two fields (x and y) in Java (31 is a somewhat arbitrarily chosen prime number):

```
int prime = 31;
int result = 1;
result = prime * result + ((x == null) ? 0 : x.hashCode());
result = prime * result + ((y == null) ? 0 : y.hashCode());
return result;
```

The implementation of *equals* in Java is even easier: you simply have to loop through all of the fields, call equals on the two values and quit if you get false.

You are asked to implement *hashCode* and *equals* for any *Product* (for example a case class) using the same basic ideas. You can download the specification file from blackboard. You can get the field values by using *productIterator*. Credit will be given for SOE (simple, obvious, and elegant)--that's to say using just the right higher-order functions on *Iterator* (or one of its supertypes). Do not invent unnecessary methods. Your solution must compile and run the spec correctly.

Answer

I will post my solution on blackboard.

5. **Multiple Answer: Spark: Do you understand the difference betw...**

Points: 12

Question

Do you understand the difference between transformations and actions in Spark? Mark the following statements which are true.

Answer

- A. Transformations simply "decorate" an *RDD* by attaching a function to it and returning the result immediately as a new *RDD*.
- B. Actions initiate a new task in a Spark execution plan and therefore take noticeable time to execute.
- C. The *collect* method is a transformation which turns an *RDD* into a *Array* without having to wait for a Spark task to execute.
- D. *reduceByKey* is an action.
- E. *reduce* is a transformation
- F. When you only want the first element of an RDD you use the method *first*, which is a transformation and therefore returns immediately.
- G. The method *foreach* is an action which applies the given (side-effect) function to each element of an *RDD*.
- H. The function which is applied to an *RDD* in a transformation or action must be *serializable*. This in turn implies that it must be
  1. an anonymous function; or
  2. from a method defined in an object; or
  3. from a method defined in a serializable class.

Correct Feedback

Good understanding of Spark!

6. **Matching: Functions and Methods: Match up the following statements**

Points: 14

Question	Match up the following statements

Answer

Match	Question Items	Answer Items
A. -	A. Together, functions and methods make an ideal marriage of ...	A. ... functional programming and object-oriented programming concepts in the Scala language.
B. -	B. A function is ...	B. ... a "pure" object which can be passed as a parameter into other functions, can be serialized/deserialized, etc.
C. -	C. A method has ...	C. ... an identifier, parameter set(s), a result type, and a body (if concrete).
D. -	D. You can "convert" a method into a function by adding a "_" ...	D. ... which gives you the corresponding function (sometimes called a *partially-applied function)* [This transformation is formally known as the *η* ("eta") expansion..]
E. -	E. The definition of a method is...	E. ... bound to a trait, class or "object".
F. -	F. The concept of a function is...	F. ... a functional programming concept.
G. -	G. The concept of a method is...	G. ... an object-oriented programming concept.

Correct Feedback Well done, you have a good understanding of this tricky concept.

7. **Matching: Variance (part one): Match up the following statements:**

Points: 4

Question	Match up the following statements:

Answer

Match	Question Items	Answer Items
A. -	A. If a (parametric) type, for example, *A* in trait *List[A]*, appears in a method parameter, it is said to be in...	A. ...*contravariant* position, where only contravariant (or invariant) types can appear.
B. -	B. If a (parametric) type, for example, *A* in trait *List[A]*, appears in a method result, it is said to be in...	B. ...*covariant* position, where only covariant (or invariant) types can appear.

8. **Short Answer: Variance (part two): It is frequently the case that you ne...**

Points: 8

Question	It is frequently the case that you need the underlying (parametric) type of a data structure to appear in a method parameter. For example, you have an immutable data structure for which you want to provide a "+" method to add a value to your data structure. Or, you have a monadic container and want to implement *map*, *filter* and *flatMap*. If the type is declared to be covariant, you will run into a problem. Briefly describe the standard "workaround" for this.
Answer	

Big Data Systems Engineering with Scala

Mid-term exam with answers

2016/10/21

You must answer these questions individually (not in pairs). I expect you to complete your work in 60 minutes but you can have up to 15 extra minutes (with a small deduction).

Important: Close all computers, smart phones, and books for the duration of the exam.

Please write clearly and succinctly. I will be looking for you to make certain points. I don't need essays! Don't worry about grammar/spelling. As long as you mention the required points in a clear context, you will get credit. If you have to write code, try your best to make the meaning clear but don't worry too much if you can't think of the exact syntax.

You may find that, in order to implement something, you need to assume the existence of method. If you're not sure it exists or what its name is, just give me a signature (not a body) for your assumed method.

If you have any trouble understanding the intent of the question (or any other difficulties) please ask me and, if appropriate, I will give a hint or explanation on the blackboard for all to see.

Although this test is primarily designed to test what you have learned from the class so far, I have tried to make it as instructive as possible. I hope you will come away with a better understanding of Scala.

1) For each of the following code expressions/fragments please specify (a) the context in which you would find them; and (b) what does it signify, i.e. its meaning expressed in words or as code that would be substituted by de-sugaring.
   1) `"x" -> "Hello"`
   2) `a <- aa`
   3) `Int => String`
   4) `x: => Int`
   5) `case a => a`
   6) `a => a`

   (26) … (2)+(2); (2)+(2)+(1*); (3)+(2); (2)+(2);(3)+(2); (2)+(2)

   (1) anywhere an expression is valid, esp. in *Map* initialization; *Tuple2("x","Hello")* or simply *("x","Hello")*.

   (2) a for-comprehension; a generator of *a* from container *aa* (* bonus point for mentioning that *a* is a pattern).

   (3) in a type alias, a parameter list, or simply as a type for a val, etc.; a function type which maps *Int* to *String.*

(4) in a parameter list/set; a call-by-name parameter of type *Int*, which is to say a *Function0* which returns an *Int*.

(5) in *match* (or other pattern-matching context, including anonymous function—partial function); if the context matches the pattern *a* then this expression evaluates to *a*.

(6) in a for comprehension or an anonymous function (function literal); introduce identifier *a* bound to a (single) free variable and evaluate to *a*.

Overall, this should have been a fairly easy question for you. The only tricky part was dealing with the last two situations. If you said something intelligent, then you got marks for these. I was frankly appalled that some of you thought that the <- and/or the -> (maybe even the =>) symbols indicated some sort of assignment!

2) Scala shares many similar concepts with Java, for example lists, arrays, and other collections. However, Scala includes some "containers" (also known as "effects") which are either not found in Java (or at least not before Java 1.8) or are significantly different. Explain briefly what is the primary purpose of each of the following "containers" and what aspect of Java it is designed to avoid:

1) Option

2) Try

(10) … (3)+(2); (3)+(2)

(1) allows for optional values (may or may not exist), i.e. two cases: *Some(x)* and *None*; avoids *null*.

(2) allows for *Success(x)* or *Failure(e)* (where *e* is an exception); avoids handling exceptions as a side-effect (*catch* clause).

Again, I expected this one to be easy. We've spent a lot of time talking about *Option* in detail, and a fair bit of time on *Try*. Many of you were hopelessly confused about what these "effects" are trying to avoid.

3) You are working on the following method which employs a for-comprehension as follows:

```
def timesTable(r: Int,c: Int) =
  for {
    rr <- 1 to r
    bb <- 1 to c
  } yield rr*bb
```

A. write this out in the equivalent form which uses *map* and *flatMap*

B. what type is returned by *timesTable*?

C. actually, the numbers of rows and columns are available to you in the form of *Option[Int]* and *Option[Int]*. Briefly, why won't the following reasonable looking solution compile? That is, why cannot these four generators co-exist?

```
def timesTable(ro: Option[Int],co: Option[Int]) =
  for {
  r <- ro
  c <- co
```

```
    rr <- 1 to r
    bb <- 1 to c
  } yield rr*bb
```

(16+3)… 8; 3; 5+3

A. `(1 to r) flatMap (rr => (1 to c) map (bb => rr*bb))`

B. some sort of sequence of *Int* (*Vector*, to be precise, but I don't expect anyone to know that)

C. because *Option* is not the same kind of container as the sequence. 3 bonus points if anything like the following is shown:

```
for (rr<-r;cc<-c) yield for (rrr<-1 to rr;ccc<-1 to cc) yield rrr*ccc
```

Given that I had mentioned at least twice that you would want to be able to de-sugar a for-comprehension, I was surprised that many of you had a problem with this part of the question. There was an enormous amount of variation in the answers. I tried to award points for each essential part of the expression. In particular, many of you forgot that *map* and *flatMap* operate on containers, not on *Int*s.

For the second part, most of you got something like *Seq[Int]* but a few thought the result would be an *Int*.

As expected, many of you struggled with the third part. Yes, this isn't easy. I don't think anyone got the three bonus points, but if you feel that you should have, let me know. Some of you said that the problem was that you couldn't base a for comprehension on an *Option*. That is nonsense.

4) For each of the following "rules" about "implicit"s, mark it as either **true** or **false**:
   A. an implicit value can substitute for a parameter that is marked "implicit"
   B. an implicit value can substitute for a parameter that is in the left-most parameter set
   C. an implicit value can substitute for a parameter provided that its definition comes before the method call;
   D. an implicit value can substitute for a parameter if it has been imported
   E. an implicit conversion function *X=>Y* must be defined in <u>both</u> the companion objects of the *X* and *Y* types.

(10)… 2x5

A. T

B. F

C. T

D. T

E. F (either companion)

Most of you had no trouble with this question.

5) A member of your project team has written the following method to calculate the factorial of a number:

```
def fact(n: Int): Int = if (n<=1) 1 else n*fact(n-1)
```

By substitution, prove that fact(3) = 6.

(5)

*fact(3) -> 3 * fact(2) -> 3 * 2 * fact(1) -> 3 * 2 * 1 -> 6 * 1 -> 6* (or similar).

None of you had trouble with this question (well, I think I marked one as 4.5 because it was not at all clear).

6) Write an expression which, given a *List[Double],* calculates the sum of the squares of the list's values (without using any mutable variables). Hint: use a well-known higher-order function.

(7) `def sumOfSquares(xs: List[Double]) = xs map (x => x*x) sum`

or

```
def sumOfSquares(xs: List[Double]) = xs.reduceLeft((r,x)=>r
+x*x)
```

or

```
def sumOfSquares(xs: List[Double]) = xs.foldLeft(0.0)
((r,x)=>r+x*x)
```

(CANNOT use "_" in any of these solutions)

This was a very simple question and I expected pretty much perfect scores. But I was disappointed. Any equivalent code scored fine. Again, I tried to give as much credit for incorrect answers as I could.

7) The following code represents a slightly simplified version of Option:

```
package edu.neu.coe.scala
object Option {
  import scala.language.implicitConversions
  implicit def option2Iterable[A](xo: Option[A]): Iterable[A] = xo.toList
  def apply[A](x: A): Option[A] = if (x == null) None else Some(x)
  def empty[A] : Option[A] = None
}

sealed trait Option[+A] extends Product with Serializable {
  self =>
  def isEmpty: Boolean
  def isDefined: Boolean = !isEmpty
  def get: A
  def getOrElse[B >: A](default: => B): B = if (isEmpty) default else this.get
  def orNull[A1 >: A](implicit ev: Null <:< A1): A1 = this getOrElse ev(null)
  def map[B](f: A => B): Option[B] = if (isEmpty) None else Some(f(this.get))
  def fold[B](ifEmpty: => B)(f: A => B): B = if (isEmpty) ifEmpty else f(this.get)
  def flatMap[B](f: A => Option[B]): Option[B] = if (isEmpty) None else f(this.get)
  def flatten[B](implicit ev: A <:< Option[B]): Option[B] = ???
  def filter(p: A => Boolean): Option[A] = if (isEmpty || p(this.get)) this else None
  def filterNot(p: A => Boolean): Option[A] = if (isEmpty || !p(this.get)) this else None
  final def nonEmpty = isDefined
  def withFilter(p: A => Boolean): WithFilter = new WithFilter(p)
  class WithFilter(p: A => Boolean) {
    def map[B](f: A => B): Option[B] = self filter p map f
    def flatMap[B](f: A => Option[B]): Option[B] = self filter p flatMap f
    def foreach[U](f: A => U): Unit = self filter p foreach f
    def withFilter(q: A => Boolean): WithFilter = new WithFilter(x => p(x) && q(x))
  }
  final def contains[A1 >: A](elem: A1): Boolean = !isEmpty && this.get == elem
  def exists(p: A => Boolean): Boolean = !isEmpty && p(this.get)
  def forall(p: A => Boolean): Boolean = isEmpty || p(this.get)
  def foreach[U](f: A => U) { if (!isEmpty) f(this.get) }
  def collect[B](pf: PartialFunction[A, B]): Option[B] = ???
  def orElse[B >: A](alt: => Option[B]): Option[B] = if (isEmpty) alt else this
  def iterator: Iterator[A] = ???
  def toList: List[A] = if (isEmpty) List() else new ::(this.get, Nil)
  def toRight[X](left: => X) = if (isEmpty) Left(left) else Right(this.get)
```

```
  def toLeft[X](right: => X) = if (isEmpty) Right(right) else Left(this.get)
}
final case class Some[+A](x: A) extends Option[A] {
  def isEmpty = false
  def get = x
}
case object None extends Option[Nothing] {
  def isEmpty = true
  def get = throw new NoSuchElementException("None.get")
}
```

1) When might you use the apply method defined in the object?
2) What does the keyword “sealed” mean and why is it used for *Option*?
3) What is the significance of the “+” in the declaration of *Option*?
4) What do you think the purpose of the “self =>” construct is?
5)
    1) What is the purpose of the inner class *WithFilter*?
    2) Semantically, how does the behavior of *withFilter* differ from that of *filter*?
6) Why is *Some* a case class while *None* is a case object?
7) In the method *getOrElse*, explain:
    1) and interpret “[B >: A]”
    2) and interpret “default: => B” (include the semantic implications of this construct)
    3) what warning would the compiler give if you tried to define this method as:

```
def getOrElse(default: => A): A = if (isEmpty) default else
this.get
```

(30+2) … (3); (2)+(3); (2)+(3); (2); (2)+(3);(2); (2)+(4)+(4)

(1) when you receive a result from a Java method which might return *null (apply* is a factory method).

(2) the class cannot be extended outside *Option.scala*; because we can only have two possible implementations of *Option: Some* and *None*.

(3) it declares type *A* to be covariant. Bonus points (2): This in turn implies that we can assign an Option[B] to a variable which is an Option[A] provided that B is a sub-type of A (including A itself). For example:

```
scala> type X=Option[CharSequence]
defined type alias X
scala> val x:X=Some("hello")
x: X = Some(hello)
scala> val x:X=Some("hello".asInstanceOf[CharSequence])
x: X = Some(hello)
```

(4) it provides an identifier with which to refer to this *Option* (useful inside an inner class such as *WithFilter*). Similar to *outer.this* in Java.

(5)

(1) It is a class, an instance of which can be used to implement the *withFilter* method;

(2) it is a lazy version of filter: it decorates an instance of *Option* as being filtered, without actually performing the filter.

(6) because a case class must have a parameter/field while a case object must not.

(7) x

(1) type B must be a super-type of A

(2) parameter *default* is called by name (it will never be evaluated if *this.isEmpty* is *false*).

(3) it would complain that *default* is in *contravariant* position.

This was quite a long question but I felt that each part was fairly simple, with the exception of part 5 (the *WithFilter* part). The point of this exercise (and of all situations where *withFilter* is implemented) is that the predicate itself is lazily evaluated. Nobody got that completely right.

In part 3, I hadn't really expected you to explain covariance, but I added bonus points for such explanations.

I don't think anyone got part 7-3. This was a very hard one, but within your knowledge base if you had been paying perfect attention.

Final Exam with answers…
…and comments

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!

The answers I have given are exactly as I wrote them up last evening during your exam with a few minor tweaks here and there to add clarity, formatting, or to give alternatives. Where the text is in bold, it implies that you should have **something equivalent to get full marks**. If it's not in bold but follows "or" then that's equivalent. Otherwise if you have something else which is correct, you might get points for extra credit.

Overall, I'm shocked by some of your answers. Apparently, quite a few of you really don't seem to understand Scala and/or functional programming at all.

I'm always surprised to find what's easy and what's hard. I thought that a lot of the questions would be really easy. For example, Questions 2, 7, 8, 10 and perhaps 6. The only one you all found easy was #6. #2 was, on the other hand, one of the least well answered yet the answer really is staring you in the face!

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.

1. (6) **The *next* method must return both the next *U* value and the next *B[U]* value** (which will be used to give us the following *U* value); Alternatively, you can return just the next *B* value provided that *B* as a method to access its *U* value. Some of you took a rather literal view of my hint: including stuff about random number generators (seeds, etc.): you must be able to adapt from solutions to other, similar problems, not clone them!
2. (8) **we need to create a (private) *tail-recursive* method that will include among its parameters** both ***x*** and whatever is used to terminate the loop. Any cumulative results will be collected in an immutable collection which will typically be a third parameter. Some of you came up with an interesting approach: make an immutable list of all the values that could arise from calling *B.next* and iterate through those. But that will only work if you happen to know in advance the sequence that *B* is going to yield! What if *B* is a random number generator?
3. (11)
```
case class B[U](u: U)(implicit nextU: U=> U) {
  def next = {val v = nextU(u); (v, B(v)(nextU))
}
def loop(b: B, f: (Int)=>Unit): B = b.next match {
 case (0,b2) => b2
 case (x2,b2) => f(x2); loop(b2,f)
 }
loop(B(10),println(_))
```

[Note that I put in a mechanism to get the next value of *u* in case *U* is not an *Int*. It's a generalization that I didn't expect you to put in — and in any case it could be done in several different ways.]

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.

1. (4) **Because *Document* extends *Function1[Seq[K],V]* we need a *def apply(ks: Seq[K]): V* method.**

2. (8)
```
def apply(ks: Seq[K]): V = get(ks).getOrElse(default())
  def default() = throw new NoSuchElementException
```

==========

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).

1. (6) **The *doc* variable is inferred to be a *Seq[String]=>V* (because of key: "one") but *one* is a *Seq[Any]=>Int*. *Any* is a super-type of *String*** (because this is a contra- variant position)**, but *Seq[Any]* can only be a super-type of *Seq[String]* if *K* is contravariant.** I reworked this answer to try to make it clearer. It only goes to show that this question was really too hard!

2. (3) **we add a "-" in front of *K* (after *Document*)**

==========

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]*.

1. (5) It provides us a hierarchical key-value store of unlimited depth;
2. (5) the result of *branches.get(h)* is an *Option[Document[K,V]]*. *FlatMap* allows us to (recursively) invoke *get(t)* on an actual document (i.e. a *Some*) otherwise simply return *None* if *h* doesn't match a key;
3. (9) we need:
   A. a parser to split the string up (suggest *StringOps.split("""\.""")*)
   B. a means of turning an array into a *Seq* (suggest *toList.toSeq*)
   C. an *implicit val String=>K* which will be (implicitly) **invoked on a *String* to provide a *K***

==========

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.

1. (5) **Lazy, parallelizable** (or partitioned/distributed), immutable, monad, etc. etc.;
2. (4) **actions turn the *RDD* into a strict value; transformations create a new *RDD* based on the original *RDD;*** My sense is that many of you don't really understand that there is a fundamental difference here. It's best illustrated in the Spark REPL: do anything with a bunch of transformations (or invoke parallelize) and the REPL will respond immediately. Do an action and the underlying map/reduce (Yarn, Mesos, whatever) will spring into action and you will have to wait several seconds depending on the complexity. Even *sc.textFile("non-existent file")* will appear to work until you invoke an action.
3. *(3)*
   A. ***collect*** or any of… *reduce, count, first, take, takeSample, takeOrdered, saveAsTextFile, saveAsSequenceFile, saveAsObjectFile, countByKey, foreach;*
   B. ***map*** or any of… *flatMap, filter, mapPartitions, mapPartitionsWithIndex, sample, union, intersection, distinct, groupByKey, reduceByKey, aggregateByKey, sortByKey, join, cogroup, cartesian, pipe,coalesce, repartition, repartionAndSort…* etc.;
4. (6) **for the *map* method on *RDD[T]* which takes as a parameter *f: T=>U*, we create a new *RDD* which simply references the original *RDD* and remembers that we want to apply *f* to the elements (without actually applying *f*).** If we apply another function *g* we will compose the functions together using *f andThen g* and remember that function instead (the same as we did with *LazyNumber* in assignment 5).

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.

(13)

```
val r = sc.textFile("/Users/scalaprof/flatland.txt")
```

```
val stopWords = List("the","a", "an", "in", "of", "by", "and", "at",
"by", "with")
val s = r flatMap {w => w.split("""\p{Punct}?\s+""")} filter (w => !
stopWords.contains(w.toLowerCase))
println(s.count)
```

I'm curious why so many did a key-based word count which is nice but not actually requested.

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
   ```

1. (4) ***TupleN* is a case class** (with all that implies, including no sub-classes) **which extends *ProductN*** and mixes in *Product* with *Serializable*. *TupleN* has explicit parameter/field names of _1, _2, ... **On the other hand, *ProductN* is a trait** with only a few concrete members such as *productArity,* and which is extended (synthetically) by all case classes— Many of you talked about how *Product* represents a Cartesian product—yes, this is the origin of the name but it isn’t really very relevant to the question asked. You further described these things as if they were completely different: yet you know (or should) that *TupleN* extends *ProductN*!
2. (2) **because they would be undefined otherwise (they are defined as abstract in *Product3*);**
3. (5) **`def map[U](f: T=>U): Triple[U] = Triple(f(_1), f(_2), f(_3))`**
4. (2) **no—it lacks *flatMap*** and *filter;*

5. (2) **yes—it implements *map;*** Many of you thought that this would not work. My answer to you is: **E pur si muove** (look it up).
6. (2) **no—it doesn't implement *flatMap.***

==========

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?

1. (5) **For three reasons: as an alias for a type which is a non-simple type;** or to indicate the type of a Scala (singleton) "object"; to define a type *member* of a trait (or abstract class)—an alternative way of making classes polymorphic, as opposed to parametric polymorphism (i.e. generics). [Note that I forgot the last of these when I first wrote up the answers—we haven't actually talked about this in class because it's not really necessary unless you are going to design Scala libraries.]
2. (3) **we can import the definition of the type into our own scope.** Alternatively, we can create a "package object" [which we didn't talk about much in class] and put the type definitions in there.

==========

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]] = ???
```

1. (10) **`def zip[U](ust: Triple[Stream[U]]): Stream[Triple[U]] = for (((x,y),z) <- (ust._1 zip ust._2 zip ust._3)) yield Triple(x,y,z)`** [or *map* alternative]. Another way very elegant way to write it is this (bonus point if you got this): **`def zip[U](ust: Triple[Stream[U]]): Stream[Triple[U]] = ust match {`**

```
  case Triple(a,b,c) => for  {
    aa <- a
    bb <- b
    cc <- c
  }yield (Triple(aa, bb, cc))
}
```

2. (10) **`def toStreams[U](ust: Triple[Seq[U]]): Triple[Stream[U]] = ust map { case us=>Stream.continually(us).flatten }`**

==========

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?

1. (6) **`def modulo(n: Int) = (n-1+tables) % tables + 1`**
2. (3) **because in an application, *tables* doesn't change so we can set up a starting set of *Encounter* objects economically (i.e. without having to specify *tables* for each one).**
3. (4) **it gets satisfied by a statement in scope of the form *implicit val tables = T***

==========

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.

1. (7) **~ is a case class with two parameters** of type *Parser[T]* (the left side and right side). It successfully parses if the two parsers match in sequence. **On the right hand side of ^^, ~ is used in a pattern-matching situation** (i.e. its *unapply* method is being invoked). Many of you ignored (or missed) the bit about appearing on both sides of "^^". The only way that it could, syntactically, appear in both places is if it's a case class (or any class with an extractor).
2. (4) **~> is an operator** (method) **on *Parser[T]* which, like ~, must satisfy both operands in sequence but results in just the right-hand parameter** (i.e. it only captures the right side).
3. (6) **^^ is an operator on *Parser[T]* which transforms it into a *Parser[U]*. The right hand side is a function of form *T=>U*. *wholeNumber* is a *Parser[String]*.** ^^ behaves very much like *map* does for a functor (or monad). In fact, it turns out that ^^ actually invokes *map* which is in fact defined for *Parser[T]* (as is *flatMap*). I

don't think anyone satisfactorily mentioned the transformation of *wholeNumber* as an example

==========

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.

1. (5) **Type of *isIs* is *(Seq[Int],Seq[Int])***

2. (8) **it is evaluated in two parts:**

A. a for-comprehension that generates *p*: *Play* from *ps*; checks that *p* doesn't equal *x*; invokes *p.compare(x.result)* which yields *io* (an *Option[Int]*); from this we generate *i*, an *Int* (if it is a *Some* else we ignore it); finally yielding a *Tuple2* of *i* and 2.

B. the result of the first part is a *Seq[(Int,Int)]* since *ps*, the first generator, is a *Seq[Play]* **and this is unzipped into a *(Seq[Int],Seq[Int])*.**

3. (4) **because *unzip* actually takes a parameter (it's implicit) so the compiler thinks the next line might be specifying that parameter.** It thus thinks the whole expression is self-recursive.
4. (3) **put a "." before *unzip*;** put an extra blank line after the *val isIs* line; provide an actual implicit value; *import postfixOps* (?)**.**

Review

for mid-term exam

Is this neat or what? Case classes

- "Case" classes: extensions of *Tuple* **with a name and named values**
  - `case class Date (year: Int, month: Int, day: Int)`
  - Just an ordinary class whose constructor takes three integer arguments plus…
    - each of those arguments is a public (val) field;
    - methods are generated (in an *automagical* companion object):
      - `def apply(year: Int, month: Int, day: Int) = new Date(year,month,day)`
      - `def unapply(x: Date) = Some(x.year,x.month,x.day)`
      - `def equals(x: Date) = year==x.year && etc.`
      - `def hashcode = etc.`
      - `def toString = s"Date($year,$month,$day)"`

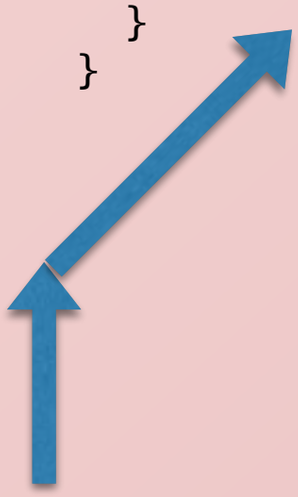
Can also be marked "var"

This "unapply" method is an *extractor*, a very powerful concept for pattern matching

- See also:
  - http://www.alessandrolacava.com/blog/scala-case-classes-in-depth/
  - http://www.codecommit.com/blog/scala/case-classes-are-cool
  - http://twitter.github.io/scala_school/basics2.html
  - http://docs.scala-lang.org/tutorials/tour/case-classes.html

Using an extractor

```
/**
  * The production: its country, year, and financials
  *
  * @param country   country of origin
  * @param budget    production budget in US dollars
  * @param gross     gross earnings (?)
  * @param titleYear the year the title was registered (?)
  */
case class Production(country: String, budget: Int, gross: Int, titleYear: Int) {
  def isKiwi() = this match {
    case Production("New Zealand", _, _, _) => true
    case _ => false
  }
}
```



"_" is essentially a wild-card which matches anything

Some things to review

- The general benefits of functional programming and Scala in particular;
- Basic syntax: val, var, def and lazy val; basics of types, classes and companion objects;
- Tuples, case classes, extractors;
- Tail-recursion vs. iteration; maintaining state; basics of actors;
- Proving programs via substitution;

Mid-term study guide

What to expect

- The final exam will last 160 minutes, online and in two parts. There will be a HackerRank part (for coding questions) which will be *closed-book* and take maybe 50 minutes. The other part will be on Blackboard and will be *open-book.* No communications devices/apps/smoke signals, etc. will be allowed—having one of these will result in a zero on the exam. [Sorry to have to impose all these rules!]
- Expect quite a few multiple choice questions, true-false, and other similar types of question.
- The coding questions on Blackboard will require *pseudo-code* (or real code if you prefer). On HackerRank, actual compiling/running code will be required.
- There will probably be some “Short Answer” questions. DO NOT write an essay! I just need to see that you have mentioned the salient points. Spelling and grammar are not important. Exact syntax (or names of methods) is nice but not absolutely required. If you don’t know the name of a method you need—invent one.
- There may be more questions than you have time for. Don’t worry, but try to get as many reasonable answers as possible: there will usually be some credit at least for attempting a question.

Topics to expect

- Major points from the first half of the semester are always likely. So, pretty much any of the following mid-term exam topics are possible:
  - Language fundamentals: what are the principal features of Scala;
  - Basic syntax: use of “_”, “=>”, “<-“, etc.;
  - Significance of val, def, lazy; difference between call-by-value and call-by-name; functions vs. methods;
  - For comprehensions, *map* and *flatMap;* syntactic sugar;
  - *Option, Try, Future, List, Stream, Tuple2*;
  - Basic rules of implicits;
  - Functional composition: *map2, lift, curried, compose,* etc.;
  - Recursion (especially tail-recursion).

Additional Topics: Final Exam

- Scala API (collections, etc.):
  - exact names of methods not required, but an understanding of the various kinds of signatures is.
- Implicits, including type-classes and context bounds.
- Simple parsers (from parser-combinator library).
- Spark
  - How RDDs work; transformations and actions; Datasets.
- Testing
- Actors

Final Exam Review

Update Fall 2022

Scala in general

- What platforms does Scala run on?
- What are the three “pillars” of Scala? And the fourth major feature
  - O-O, functional and statically typed
  - Implicits
- What are the Scalastic principles?
  - Types, Objects, Functions, with Implicits
- What are the Scalastic pragmatics?
  - Usability, Power, Simplicity
- What is Scala good for?
  - Microservices, reactive programming, parallel programming, parsing, general programming.

Structure of a program

- A Scala program/method looks like what?
  - Definitions
  - **Expression**

def sqr(x: Double): Double = x * x

println(*math.sqrt(sqr(3) + sqr(4))*) *[side effect of mutating the program environment but still an expression with no result].*

Functional Programming

- What is a *pure* function?
  - No side effects (no mutation) and
  - output is always same for given input
- What is a side effect?
  - Change of state within the program?
  - Change of state outside the program? (I/O)
  - Abnormal jump or termination (e.g. an exception)
- And why do we care about pure functions?
  - They allow us to “prove” programs to be true.
    - We can prove entire applications to be good.
    - In practice, we don’t
    - Combine testing with proof: that’s good enough.
  - We need pure functions when we use parallelism:
    - because side-effects on remote worker nodes will not help us on the driver node.

Functional Programming (2)

- “Symbolic programming”
  - The compiler is allowed to take short cuts (i.e. evaluate expressions at compile time, rather than leave them until runtime).
  - For example (the most obvious example), de-sugaring is a short cut.

- Lazy (deferred) evaluation
  - Can significantly reduce the amount of work we have to do;
  - Is more mathematical:
    - We can define infinite series
      - Val fibonacci: LazyList = …. fibonacci.scanLeft “terminating” at least for finite number of elements.
      - Val x = x. “non-terminating"

Functional Programming (3)

- Functional programs are expressions
- Higher-order functions and functional composition
  - Output of which is another function or:
  - Input of which is another function.
- Lambda (mathematical concept based on free and bound variables). aka as a “function literal” or “anonymous function”
- Recursion as the primary re-use mechanism (vs. iteration)
  - def factorial(x: Int): Int = if (x<=1) 1 else x * factorial(x-1)

Functional Programming (4)

- Pattern matching
- What did I say about pattern matching?
- What is it the opposite of?
- Expression
- Pattern matching is based on the concept of *un-*expression.

Functional Programming (5)

- Ability to construct types
- I don’t mean the ability to define classes.
- I mean *Option[Int]*.
- In Java Optional<Int>.
- Can you think of any way to construct a new type based on existing types that doesn’t use some name???
  - Val x = 1; val y = math.Pi; val z = “Hello World!”
  - Val q = (x, y, z)
  - Newton’s Approximation: pass in max tries; initial guess; function; first derivative. What we get back is: *either the solution*; or a *reason why it failed*.
  - (x, “reason”). Either[Double, String]. Try[Double]

Language types

- All the normal stuff that you have in, say, Java.
- plus what other kind of type is *built-in* to Scala but not Java?
  - Tuples
- Feature of functional programming:
  - Curried functions
    - f(x, y, z) “tupled” form
    - g(x)(y)(z) “curried” form (same result provided that *g = f.curried*)
    - val x = 1, y = 2
    - val h1 = *f(x, y, _)* and val h2 = *g(x)(y)* are the same thing!
    - h1(3) == h2(3)

Monads and all that jazz

- Do you think monads are a bit scary?
  - They’re really a pretty simple concept—just a wrapper around something.
  - That wrapper can have various cardinalities:
    - 0 or 1: *Option, Future** or *Try**
    - 1: *Either*
    - 0 to N: *List*, *LazyList*, etc.
- So, if these things are wrappers, how do we unwrap them?
  - Well, the idea is that we can do things to the content of the monad without actually losing the wrapper aspect.
  - The most flexible and important construct for doing this is the for-comprehension

* *actually,* Future *kind of has three possibilities but always 0 or 1*
* *With Try, we always have 1 value (but it might be an exception)*

Monads: for-comprehensions

- Suppose you have two instances of *Option[T]*, and you have a function *f: (T1, T2) => R*;
  - Then you can do something like:
    - *val t1o: Option[T1] = Some(???), t2o: Option[T2] = Some(???)*
    - *val r: Option[R] = for (t1 <- t1o; t2 <- t2o) yield f(t1, t2)*
  - Recall that this is the same as “map2,” a method we talked about but don’t actually need. It’s also our *map3*, *map4*, etc.
  - Realize that, as far as most of your code is concerned, the *t1o*, *t20* and the result of the for-comprehension are all still “wrapped.”
  - We only actually unwrap these monads in the context of the for-comprehension.

Monads: for-comprehensions (2)

- Monads can also be thought of as a means to sequencing, something which generally isn't important in functional programming.
  - *for (t1 <- t1o; t2 <- t2o) yield f(t1, t2)*
- In this example, we forced t1o to be unwrapped before t2o. Indeed, we might have a for-comprehension that looks like this:
  - *for (t1 <- t1o; t2 <- g(t1)) yield f(t1, t2)*
- Here, it's clear that we can't get a value for *t2* until we have a value for *t1*.

Spark

- What is a Spark Context for?
- What is a data frame? Is it the same thing as an RDD?
- What's an action? How is it different from a transformation?
- Is RDD lazy?
- Why is *count* an action and not a transformation?

Variance

- Liskov Substitution Principle (assignment)
  - *val x: X = ???*
  - *val y: Y = ???*
  - Assign the value of *y* to variable *x* (e.g. *val x = y*)
    - *val z = x.q*
    - Provided that *X* is a super-class of *Y*
    - So, *Y* is a sub-class of *X*, which implies that:
      - *Y* has all of the properties of *X*, and maybe others.

Variance (2)

- Liskov Substitution Principle (yield or return)

def f(z: Z): Y = z

val x: X = f(z)

val z: Z = ???

  - This all works OK provided that *X* is a super-class of *Y*
  - And *Y* is a super-class of *Z*.

Covariance

- X is a super-class of Y

```
def method(xs: List[X], f: X=>Z): List[Z] {
   for (x <- xs) yield f(x)
}
```

```
val q: List[Y] = List(….)
method(q)
```

```
class List[+X] { …. }
```

We need List[X] to be a sub-class of List[Y] which will be true if X is a sub-class of Y.

```
def get(i: Int): X = ???
```

Contravariance

- If a type is defined thus: *X[-T]*, i.e. where *T* is *contravariant*, then a value of *X[U]* can substitute for a *X[T]*, provided *U* is a *super*-type of *T*.

- The typical usage of contra variance is in a function type, such as:

```
trait Function1[-T, +R] {
   def apply(t: T): R
}
```

Invariance

- X can’t be both covariant and contravariant

```
class List[+X] {
    def ::[Y >: X](y: Y): List[Y] = ??? (but we include Y = X)
    def apply(i: Int): X = ???
}
```

Arrays and ArrayBuffer are invariant in their underlying type.

But the :: method of List gets around the covariance of underlying type by the super-type trick so the following works:

val dogs = List[Dog]

val cat = Cat(“squeaky”)

val animals: List[Animal] = cat :: dogs

Dealing with this super type thing

```
scala> trait Animal
defined trait Animal
scala> case class Dog(name: String) extends Animal
defined class Dog
scala> val dogs = List(Dog("Bentley"))
dogs: List[Dog] = List(Dog(Bentley))
scala> case class Cat(name: String) extends Animal
defined class Cat
scala> val cat = Cat("squeaky")
cat: Cat = Cat(squeaky)
scala> val cat: Cat = Cat("squeaky")
cat: Cat = Cat(squeaky)
scala> cat :: dogs
res2: List[Product with Animal with java.io.Serializable] = List(Cat(squeaky),
Dog(Bentley))
scala> val animals: List[Animal] = cat :: dogs
animals: List[Animal] = List(Cat(squeaky), Dog(Bentley))
scala> val dog = Dog("Xena")
dog: Dog = Dog(Xena)
scala> val dogs2 = dog :: dogs
dogs2: List[Dog] = List(Dog(Xena), Dog(Bentley))
scala> animals(0) match { case dog: Dog => dog.name; case cat: Cat => cat.name; case _ =>
"no name" }
res11: String = squeaky
```

Note: when we did *cat::dogs*, the compiler found all the common super-types

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

- 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

```
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.

Final Spring 2025

- Due 22 Apr at 14:00
- Points 120
- Questions 10
- Available after 22 Apr at 11:45
- Time limit 120 Minutes

Instructions

You have two hours for this exam. Try to pace yourself: 1 point per minute.

This is an open-book exam. You **may not directly copy code** from any source (unless you explicitly cite that source). Results from AI will not be accepted. I *will* be checking your answers for AI generation.

However, your IDE is your friend and you can check the answers to any of the questions using the IDE or StackOverflow or whatever. There will be parts of this exam that you don't know and will have to look up (intelligently).

In each of the "essay" questions, you must be brief. Just concepts. No fluff. Spelling and grammar are not important!

Finally, please ask if there's anything at all that isn't clear in the questions!

Quiz results are protected for this quiz and are not visible to students.

ⓘ Correct answers are hidden.

Score for this quiz: 116.5 out of 120

Spring 2025 Midterm Exam

- Due 20 Feb at 16:30
- Points 115
- Questions 5
- Available 20 Feb at 15:00 - 20 Feb at 16:30 1 hour and 30 minutes
- Time limit 90 Minutes

Instructions

You have 90 minutes to finish this exam. You should at least attempt each question (except for the bonus question which is optional). Try not to spend more time (in minutes) on a question than its point total.

You will not all be answering the exact same questions.

Multiple answer questions deduct for incorrect answers, however, you cannot score less than zero for any question.

Some questions test your code *comprehension*, that's to say your ability to read and understand the code. In these comprehension questions, you do not have to understand the details of every line of code--only enough to answer the questions

I don't expect you to know all the answers. But I do expect you to try things in the REPL (or IDE) to confirm your answers.

You have a little less than one minute per point. Use your time wisely. Don't spend a long time working on a low-value question and then find you have no time left for a high-value question.

This quiz was locked 20 Feb at 16:30.

Attempt history

	Attempt	Time	Score
LATEST	Attempt 1	84 minutes	85.44 out of 115

Score for this quiz: 85.44 out of 115
Submitted 20 Feb at 16:30
This attempt took 84 minutes.

Question 1
12 / 12 pts

Please select all the true statements about Scala:

Correct!

Scala often has two ways of writing the same code. One is functional and easy for the compiler to understand.

- [x] The other is designed to be easier to understand by humans. This latter form is called "syntactic sugar."

- [ ] Scala runs only on the Java Virtual Machine.

Correct!

- [x] Scala is the language that Spark is implemented in.

- [ ] Scala is not an object-oriented language.

Correct!

- [x] Scala is a functional programming language.
- [ ] Scala is 100% compatible with Java such that, for example, *List* is the exact same type in both languages.

Correct!

- [x] Scala programs which obey referential transparency are capable of being proven to be correct.

Correct!

- [x] Scala has a well-developed pattern-matching capability.

Correct!

- [x]

Scala is designed to be *scalable* (ideal for processing huge datasets), hence the name Scala for scalable language

Question 2
11.11 / 25 pts

Please select all the statements below that you believe are true (if you're not sure, check using the compiler, REPL, etc.):

Correct!

- [x]

A **case class** generates, synthetically (auto-magically), various instance methods including: *toString, equals, hashCode;* and also the following "class" methods: *apply,* and *unapply*.

Correct!

- [x]

Members of a **case class** can be referenced from other classes unless marked **private**. The programmer doesn't have to create a specific "getter" method.

- [ ] A **case class** must extend a sealed trait.

Correct!

- [x]

A **case class** is essentially a named tuple; that is, it has members with names chosen by the programmer.

Both case classes and tuples extend the trait *Product* (which, incidentally, corresponds to a Cartesian product type).

Correct!

:: is a **case class** that extends List.

A **case class** can be extended by another case class, provided that it is in the same module.

Correct!

A **case class** extends Product, just as tuples do.

Correct!

A member of a **case class** can be kept hidden by prefixing its identifier with "private val"

You Answered

A **case class** can be extended (i.e. sub-classed).

Correct answer

If you must have a call-by-name parameter of a **case class**, it should be placed in a subsequent parameter set, and thus is not a true "member."

Correct answer

The synthetic methods of a **case class**, such as *toString*, *apply* and *unapply,* do not "see" a parameter in a second (or subsequent) parameter set.

You Answered

A **case class** must have at least one member (that's to say, at least one parameter in the first parameter set of its declaration).

All concrete classes in Scala must be **case classes**.

Correct answer

A **case object** is similar to a case class but has no members or parametric types.

A **case class** cannot extend an abstract class.

A member of a **case class** cannot be made mutable.

Question 3

8 / 12 pts

Which of the following statements regarding *LazyList* in Scala are true? Mark all that apply.

Remember that I don't expect you to know all the answers to these questions. But I do expect you to be able to experiment sufficiently that you can determine the answer.

You Answered

Any instance of *LazyList* can be successfully output to the console by invoking *foreach println* on it.

Correct!

The elements of a *LazyList* are memoized once they are evaluated.

The elements of an *Iterator* (another "non-strict" type) are *not* memoized, or even accessible, after being evaluated.

Correct!

An instance of *LazyList* defined by *LazyList.from* or *LazyList.continually* (or similar method) is:

- of unknown length;
- unevaluated.

Correct!

An instance of *LazyList* that is the result of applying the *take* method is:

- of known length
- unevaluated

Defining a *LazyList* using the method *LazyList.from* will always cause an *OutOfMemory* error unless it is immediately made finite with a method such as *take*.

For example:

```
val xs = LazyList from 0
```

Question 4
26 / 28 pts

Take your time to understand the code given, along with the "specification" or unit tests.

Your colleague, Joe, has been building a special *Collection* library where the collections have somewhat similar behavior to RDDs in Spark in that transformations are implemented lazily and actions do the full evaluation. But here, the actions are performed asynchronously, and the results are instances of *Future*.

Annoyingly, Joe has now gone on vacation to the Galapagos Islands, where mobile phone coverage is spotty. You're on your own trying to complete his work for a deadline later today. Fortunately, he writes excellent scaladoc (actually, all generated by AI), which should be of considerable help when working on the code.

Collection.scala (https://northeastern.instructure.com/courses/206870/files/33136728?wrap=1) **(https://northeastern.instructure.com/courses/206870/files/33136728/download?download_frd=1)** and **CollectionSpec.scala (https://northeastern.instructure.com/courses/206870/files/33136732?wrap=1)** **(https://northeastern.instructure.com/courses/206870/files/33136732/download?download_frd=1)**

Important: It is assumed that these two Scala source files are in the "csye7200" package. If not, you will have to adjust the declaration of *xs* in *LazyCollectionBase*.

Joe's also done an excellent job of figuring out all of the types needed. All you have to do is fill in the code where he's put ??? (each is marked with a TODO). The code shows you the marks available for each TODO.

Once you've got all the unit tests running green, you can move on to the next question.

Collection.scala (https://northeastern.instructure.com/files/33155813/download)

Question 5

28.33 / 38 pts

In this question, you should answer the various questions below. Each question refers to code marked by XXX #*n* where *n* is the corresponding question number (points available shown in parentheses). Be brief in all of your answers. Seriously!

1. (4) What method is conspicuous by its absence from the *Collection* trait? Probably, Joe had some trouble declaring that.
2. (5) Why do you think *LazyCollectionBase* is declared as an abstract class rather than a trait?
3. (re: *xs*)
   A. (7) Your colleague chose to make *xs* an *Iterator[S]*. What's the big disadvantage of that? What might you consider changing it to?
   B. (6) Why is *xs* marked *val*? Why is it marked *private*?
4. (re: unit)
   A. (5) Why do you think *unit* is an instance method of *LazyCollectionBase*? Think about the various parametric types involved.
   B. (5) Why do you think it is declared abstract?
5. (6) why are we doing pattern matching here? Why do we need *c*, an instance of *LazyCollectionBase*?

Your answer:

```
def iterator: Iterator[T] = xs map f
```

```
def map[U](g: T => U): Collection[U] = unit(this)(g compose f)
```

```
def aggregateAsync[U](u: => U)(g: (U, T) => U)(implicit ec: ExecutionContext): Future[U] = Future(iterato
r.foldLeft(u)(g))
```

```
case c: LazyCollectionBase[S, T] => LazyCollectionOnceOnly(ts)(g)
```

1.

2. LazyCollectionBase is abstract since we can derive concrete implementations of methods such as unit, for example in LazyCollectionOnceOnly and we can have a reference to xs. It is also abstract since it contains a parametric value, which a trait cannot have.

3.

A. Iterators evaluates elements as needed but they cannot be revisited again.
We could change it to LazyList. Even LazyLists can evaluate elements when needed, but they can also revisit previous ones as well.

B. xs need not be modified when performing operations in a functional manner, that's why it is marked as val. It is marked as private (ex: Pushdown computing) since we are manipulating xs by specifying an action of function on it

4.

A. Abstract classes allow us to have implementations for unit and we can have parametric values in

abstract class

B. To allow the usage of parametric values , (traits may not provide it)

5. c helps us to perform unapply on collection , a concrete class of LazyCollectionBase

0 4.5 7 5.5 3 2 2 After 4:30 2.67 3.33 -- -- 2 3.33 2.33 Net increase 2.67 0 0 0 0 1.33 0.33 = 4.33

Quiz score: 85.44 out of 115

This Test has 28 attempts. For information on editing questions, click **More Help** below.

Test Canvas: Mid-term Fall 2019- Requires Respondus LockDown Browser

The Test Canvas lets you add, edit, and reorder questions, as well as review a test. More Help

Question Settings

You can edit, delete, or change the point values of test questions on this page. If necessary, test attempts will be regraded after you submit your changes.

Description	This is your mid-term exam.
Instructions	
Total Questions	7
Total Points	68
Number of Attempts	28

Select: All None | Select by Type: - Question Type -

Delete and Regrade | Points | Update and Regrade | Hide Question Details

1. **Multiple Answer: Fake news? Or real?: Which of the following represent real...** Points: 10

Question Which of the following represent real news stories about Scala?

Answer

- ✓ Scala can now be used in Browser code (it compiles to javascript), natively, and on the JVM.
- Scala implements most code more efficiently than Java so, if you have a performance-critical task to be implemented on the JVM, you should use Scala instead of Java.
- ✓ The rewrite of Scala for version 3.0 and beyond (currently being released experimentally as "Dotty") is scheduled for full release in early 2020, soon after the release of the 2.14 version.
- The concept of "lambdas" was invented by Martin Odersky, one cool dude, and was ultimately copied by Java in Java 8.
- ✓ Scala version 2.13 is out and one of the major differences is that *Stream* has been deprecated in favor of *LazyList*.
- Scala is the greatest language of all time and the ideal language for experimenting in Data Science research (as opposed to Data Engineering). This is because Scala is a compiled, strictly-typed language. For this reason, you should stop using Python for Data Science work. Trust me. I know.

Implicits have been found to be too confusing for programmers to use and will therefore be unavailable in Scala 3.0. Sad.

Crooked Martin's *Scala* is a dead language because Java 11 has caught up. Everything that you can do in Scala can be done in Java now so there's no point in Scala.

✓ Functional programming has been around since the 1950s but Scala is the first mainstream FP language which is also object-oriented.

✓ Scala actors were deprecated a few versions ago. Instead, we use Akka for all applications that need to be reactive, including streaming operations.

2. Multiple Answer: Parameters: The presence or absence of param...

Points: 6

Question	The presence or absence of parameters has great significance. Which of the following pairs of concepts (or constructs) differ *primarily* (or *at least partly)* according to their having/not having parameters?
Answer	✓ A. *class* vs. *object* ✓ B. *trait* vs. *abstract class* ✓ C. *lazy val* vs. *def* D. *function* vs. *method* E. *lazy list* vs. *list* F. *match expression* vs. *partial function* G. *call by name* vs. *call by value*

3. Multiple Answer: Call by name vs. function: Considering the declarations of metho...

Points: 6

Question	Considering the declarations of methods, which of the following are valid reasons why Scala has a special syntax for call-by-name parameters, for example: *x:* => *X*, as opposed to simply declaring the parameter as a function, for example x: () => X.
Answer	A. Because passing the function would force its evaluation. ✓ B. Because, when referencing the value x inside the method, you would need to apply the function by adding parentheses, for example *val y = x()* ✓ C. Because the caller of the method would typically have to pass a lambda of the form *() => x* as the argument to the parameter (instead of just passing *x*). D. Because passing a function would not be as efficient as the call-by-name mechanism.

4. Matching: Scala syntax: Match up the following left-hand-side...

Points: 9

Question Match up the following left-hand-sides of declarations with their right-hand-sides.

Answer

Match	Question Items	Answer Items
A. -	A. `val f: PartialFunction[Any, Int]`	A. `{ case x: Int => x; case x: String => x.toInt }`
B. -	B. `val a: AnyVal`	B. `if (true) 1`
C. -	C. `val (a, b)`	C. `"a" -> "b"`
D. -	D. `def method[T, R](f: T => T => R): T => T => R`	D. `a => b => f(b)(a)`
E. -	E. `def method[T, T, R](f: (T, T) => R): (Try[T], Try[T]) => Try[R]`	E. `map2(_,_)(f)`
F. -	F. `val a: Int`	F. `if (true) 1 else 0`

5. Multiple Answer: Variance: Which of the following statements reg...

Points: 10

Question Which of the following statements regarding variance are true?

Answer

✓ A.

Dealing with variance is necessary in a strictly-typed object-oriented language like Scala because of the implications of the Liskov Substitution Principle.

✓ B.

If container type *C* is covariant on *A*, i.e. *C* is defined something like **trait C[+A] {...}**, then *C[S]* is a sub-class of *C[T]* provided that *S* is a sub-class of *T*.

✓ C.

If container type *C* is contravariant on *A*, i.e. *C* is defined something like **trait C[-A] {...}**, then *C[T]* is a sub-class of *C[S]* provided that *S* is a sub-class of *T*.

D.

In Scala, an instance of *Array[S]* can be passed to a method that requires an *Array[T]* where *S* is a sub-class of *T*.

✓ E.

Covariant position refers to the result type of a method while *contravariant position* refers to a type referenced in a parameter of a method.

F. All mutable collections in Scala are declared to be covariant on the parametric (i.e. underlying) type.

6. Short Answer: Lazy Collection: You have been asked to

Points: 12

implement a La...

Question

You have been asked to implement a *LazyCollection*. Because the initial values of the collection are typically expensive to gather (requiring an internet lookup), we will store them as a call-by-name iterator. Whenever we invoke the *iterator* method on our *LazyCollection* it will force an evaluation of each element. Therefore we try to do that as few times as possible. Meanwhile, we can invoke *map* as often as we like because it is required to be very inexpensive. Here is the code you have been given--your task is to implement where you see the ???. Remember SOE.

```
case class LazyCollection[X, Y](private val f: X => Y)(xs: => Iterator[X]) {
  def map[Z](g: Y => Z): LazyCollection[X, Z] = ???
  def iterator: Iterator[Y] = xs map f
}

object LazyCollection extends App {
  val sequence = Seq(1,2,3)
  val lazyNumbers = LazyCollection[Int, Int](identity)(sequence.toIterator)
  val transformed = lazyNumbers map (_*2)
  transformed.iterator.foreach (println) // 2 4 6
}
```

Answer

```
LazyCollection(f andThen g)(xs)
```

7. Short Answer: Folding: A colleague has written a method to a...

Points: 15

Question

A colleague has written a method to apply an associative aggregation function to a sequence of Doubles. He insists that this cannot be accomplished without using a mutable variable. Please show him how it can be done without any mutable variables and without any danger of a stack overflow. [For full credit, you may not use any higher-order library functions--however, any answer that yields the correct result will get some credit].

His code:

```
object Aggregation {

  def aggregate[Double](xs: Seq[Double], f: (Double,Double)=>Double, initial: Double = 0): Double = {
    var result = initial
    xs foreach { x => result = f(result, x) }
    result
  }
}
```

Answer

```
object Aggregation {

  def aggregate(xs: Seq[Double], f: (Double, Double) => Double, initial: Double = 0): Double = {
    @scala.annotation.tailrec // not required for full marks
    def inner(r: Double, work: Seq[Double]): Double = work match {
      case Nil => r
      case h :: t => inner(f(r, h), t)
    }

    inner(initial, xs)
  }
}
```

Select: All None Select by Type: - Question Type -

Delete and Regrade | Points | Update and Regrade | Hide Question Details

← OK

Spring 2019 Mid-term answers

Q1:

True.

Q2:

False: No, you can also write Scala code for your browser with Scala.js (which compiles into Javascript) and you can also run Scala natively using Scala Native (with LLVM).

Q3:

val evaluates its expression exactly once (possibly zero times if marked *lazy*);
def defines an expression which is evaluated every time the identifier is referenced.

A *val* cannot be parameterized;
A *def* can be parameterized.

A *val* must be declared--and, therefore, evaluated (or "initialized")--before (above or to the left of) it is actually used--unless it is marked *lazy* (in which case it is treated more like a *def*).
A *def* may be declared anywhere in the appropriate scope where it is used.

Q4:

A method is tail-recursive (i.e. tail-call optimized) provided that the result of any recursive call is returned without further processing.

A tail-recursive (i.e. tail-call optimized) method will not throw a *StackOverflowError*, no matter how deep the "recursion."

The following code will likely cause a stack overflow:

```
def factorial(x: BigInt): BigInt = if (x<=1) 1 else x * factorial(x-1)
factorial(BigInt(100000))
```

In most simple situations, you can write a tail-recursive method by giving it two parameters:

- the parameter whose value will be returned when the terminating condition is reached;
- the parameter which represents the work remaining to be done by the method.

In programming, recursion is the corresponding technique to the mathematical technique of proof by induction. Indeed, it is trivially easy to "prove" recursive code by using the substitution principle and then using induction.

Q5:

mystery is a higher-order function (method).

The implementation of *mystery* (i.e. its right-hand-side) uses functional composition.

The implementation (right-hand side) could be replaced by:

```
t => for (x <- g(t)) yield f(x)
```

The implementation (right-hand side) could be replaced by:

```
g andThen (_ map f)
```

The underscore ("_") denotes the value of what will be the input parameter of the function that is result of invoking the *mystery* method.

The function *f* will never actually be called if the result of function *g* is a *Failure*.

Q6:

The purpose of the mystery method is to create a function which will return the value of *Try(f(t))* for some *t*, provided that the result of invoking *g(t)* is a *Success*. If *g(t)* yields a *Failure(x)*, then *Failure(x)* will be returned by the resulting function. It is useful when you have a function *f* (or method) which takes a *T* and returns an *R* (perhaps from some third-party library) but where that function *f* might throw an exception with some inputs. In particular, this is useful when *f* is a Java method.

Q7:

A type class is a mechanism that allows a programmer to impute additional behavior (functionality) to another type without using inheritance.

A "context bound" can be applied to a parametric type of a class or method such that there must be an implicit instance of the appropriate type class in scope. For example, declaring the following class requires that concrete instances of type *X* must have an appropriate *Ordering*:

```
case class Sortable[X: Ordering](xs: Seq[X]) {
  def sorted: Seq[X] = xs.sorted
}
```

Methods defined by a trait which is a type class do not normally refer to *this*, that is to say they are not ordinary instance methods. For example, in the trait *Zero* (see below), the only method (*zero*) does not refer to *this*.

```
trait Zero[X] {
 /**
 * Method to create a zero/empty/nothing value of X
 *
 * @return an X which is zero (empty, etc.)
 */
 def zero: X
}
trait IntZero extends Zero[Int] {
 def zero: Int = 0
}
```

Numeric is a type class which extends *Ordering*.

Q8:

In a pattern-matching context, the identifier "*x*" matches any value and this value can then be referred to by *x*in the evaluated expression.

A pattern-matching context is present following a *case* token and preceding the rocket symbol "=>" in a match expression.

A pattern-matching context is present preceding the "<-" in a generator of a for comprehension.

A pattern-matching context is present after *val* or *var* and before the "=" of a variable definition.

In a pattern-matching context, an object can be "deconstructed" using a compiler-generated invocation of *unapply*. An example of this might be:

```
case h :: t => inner(r ++ h, t)
```

Here, the *unapply* method of the companion object to "::" is used to yield values for *h* and *t*.

Q9:

When an implicit is defined in the same scope as its intended use, it must follow similar rules to an ordinary *val*: it must be declared before it is referenced.

An implicit thing (*val*, *object*, etc.) must be marked *implicit* if it is to be used implicitly.

The compiler will find an *implicit* according to the usual scope rules and, if no matching implicit is found, it will then look in the companion object of the required type. If seeking an implicit function, it will look in the companion object of each of the input and output types of the function.