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.