Lecture 19: The Many Hats of Scala: Functional

10:00 AM, Mar 9, 2020

Contents

1 Onto Scala! .......................... 1
   1.1 Thinking Back to Functional Programming ........................................... 1

2 Using Higher-Order Functions: The Rainfall Problem ......................... 2
   2.1 Additional higher-order functions ....................................................... 3

3 Putting Functions/Methods in Classes ................................................. 3

4 Testing Programs .................................................................................. 4

5 Running Programs ................................................................................. 4

6 Beyond Built-In Functions: The MaxTriples Problem ......................... 5
   6.1 Creating Classes ................................................................. 5
   6.2 Instantiating Objects from Classes .................................................. 5
   6.3 Pattern Matching ......................................................................... 6

7 Resources Pointers ................................................................................. 7

Objectives

By the end of these notes, you will know:

- How to write and test functional programs in Scala

By the end of these notes, you will be able to:

- Write and test basic functional programs in Scala

1 Onto Scala!

One of the neat things about Scala is that it was designed to embrace each of functional, imperative, and object-oriented programming. The result is a language that lets you choose whichever style best suits each part of your project, and to combine computations done in different styles. Over the
next three lectures, we’ll introduce you to the core Scala features from each of these styles. Then, over the rest of the course, we’ll be talking about how to effectively use them together.

Today, we’ll work with the functional perspective.

### 1.1 Thinking Back to Functional Programming ...

When you think about “functional programming” as you learned it in CS17, what comes to mind? There are many possible answers, but some might include (depending whether you remember Racket, Pyret, or Reason):

- Using map and filter
- Pattern matching
- Creating new datatypes in a lightweight way
- Writing programs recursively
- Taking functions as arguments and returning them as values
- Writing programs without assignment statements or variables

All of these features are part of Scala, but this time, they can be used with and within classes and objects. To see how this works, we will write two programs in Scala. Specifically, we’ll write code for the two programs for which you saw multiple solutions in lab this week.

### 2 Using Higher-Order Functions: The Rainfall Problem

Here again is the description of the rainfall problem, which you encountered in lab:

Write a program called rainfall that consumes a LinkedList<Double> representing daily rainfall readings. The list may contain the number -999 indicating the end of the data of interest. Produce the average of the non-negative values in the list up to the first -999 (if it shows up). There may be negative numbers other than -999 in the list (representing faulty readings). If you cannot compute the average for whatever reason, return -1.

How might you have written rainfall if you were back in CS17, using higher-order functions? As with the second lab solution, you could have written one function to extract the data you want to average, then you could use fold and length to concisely compute the average. To get started, let’s ignore the part of rainfall that talks about the sentinel (-999) value, and just think about (a) filtering out negative numbers, and (b) computing the average. Here’s a Scala function that does these two parts:

```scala
/**
 * Compute average of non-negative numbers in a list
 */
```
There’s a lot to unpack here – let’s do that piece by piece (you may find it useful to watch the lecture capture here, where we will have drawn all over this code on the board to label the parts).

- The comment notation is similar to Java’s.
- The keyword `def` defines a function or method – the difference between the two is based on what code you put in the body: a function has a single expression in the body, whereas a method may have multiple commands in the body. With a method, the value of the last line is returned. No explicit `return` statement is needed for either functions or methods.
- Functions and methods have their input and output types listed as part of the header (first line), but the return type goes after the parameter specification, rather than before the function/method name as in Java.
- The first line of the body uses `filter` to extract a list containing only the non-negative numbers from the input list, and stores the filtered list under the name `cleaned`.

  - `filter` gets called using method notation: through a dot after the list to be filtered
  - the one argument to `filter` is a function that checks whether a number is non-negative. The argument is written as an anonymous function (a.k.a., lambda): it takes one argument (`n`), and returns (=>) whether `n` is at least 0.
  - the anonymous function passed to `filter` is missing the type annotations that we wrote on the `rainfall` function header. Scala supports type inference, so types are optional in Scala. We will continue to put them on function and method headers (they are useful for helping us understand what functions are doing), but we will leave them off anonymous or otherwise very simple functions in which the types are already clear.

- the result of the call to `filter` is named `cleaned`. The `val` indicates that `cleaned` is immutable (a constant, rather than a variable)
- Next, we use `foldLeft` to aggregate the values from the `cleaned` list into a single value. `foldLeft` takes two arguments: the value when the list is empty, and the function (of two arguments) to use to combine each element with the running result so far.

If you don’t remember `fold`, or were never comfortable with it in the first place, you could also have written a `sum` function directly. We’ll see how to write straightup recursive list functions in a moment.

### 2.1 Additional higher-order functions

Now let’s augment our code to deal with the -999 value. Rather than keep all non-negative numbers, as we are doing now, we instead want to keep only those that occur before -999. Scala has a larger collection of higher-order functions than you saw in CS17/19/111. In particular, it has `takeWhile`,

```scala
def rainfall(lst : List[Int]): Double = {
  val cleaned = lst.filter(n => n >= 0)
  cleaned.foldLeft(0)((a, b) => a + b) / cleaned.length
}
```
which returns the prefix of the list before some condition is met. Here’s the code augmented with a call to `takeWhile`:

```scala
def rainfall(lst : List[Int]): Double = {
  val truncated = lst.takeWhile(n => n != -999)
  val cleaned = truncated.filter(n => n >= 0)
  cleaned.foldLeft(0)((a, b) => a + b) / cleaned.length
}
```

You can see the available higher-order functions in the Scala documentation:

http://www.scala-lang.org/api/2.7.7/scala/List.html

### 3 Putting Functions/Methods in Classes

All Scala functions and methods live in classes (as in Java) or objects (different from Java). In general, Scala embraces a principle of allowing lightweight ways to do simple tasks. Sometimes, you have a class that you would only instantiate once (meaning there would only ever be one object for the class). This is often true of classes that just hold functions for other classes to use (as with our `rainfall` example). Scala helpfully gives a notation for creating such singleton objects without first going through a class:

```scala
object FunctionExamples {
  // Compute average of non-negative numbers that occur before -999
  def rainfall(lst : List[Int]): Double = {
    val truncated = lst.takeWhile(n => n != -999)
    val cleaned = truncated.filter(n => n >= 0)
    cleaned.foldLeft(0)((a, b) => a + b) / cleaned.length
  }
}
```

The essential part here is the word `object` before the class name (`FunctionExamples`). This instructs Scala to create a class with the default constructor, then implicitly use `new` to create an object from that class. The object can be referenced elsewhere in the code through the name `FunctionExamples`.

### 4 Testing Programs

The `tester` library also works in Scala. Other than some minor changes in syntax, testing should look familiar to what you’ve done in Java. Here’s an object with a test case for `rainfall`. The file containing this object would also need to import the testing library.

```scala
import tester.Tester

object FunctionsTest {
  def testRainfall(t : Tester) {
  
```
5 Running Programs

As in Java, we create a `Main` object with a `main` method, and invoke the tester within it. Here’s such an object, which uses constructs we’ve already discussed.

```scala
object Main {
    def main(args: Array[String]) = {
        Tester.run(FunctionsTest)
    }
}
```

Again in the spirit of giving you lightweight ways to do simple tasks, Scala provides a simpler notation for writing the above code. You can have `Main` extend a built-in class called `App`. The entire body of the class is treated as the body of the `main` method. The following code is equivalent to the `Main` class above:

```scala
object Main extends App {
    Tester.run(FunctionsTest)
}
```

You can use either of these two forms in your own code.

6 Beyond Built-In Functions: The MaxTriples Problem

Now we turn to writing functional programs that involve explicit recursion, rather than just higher-order functions. For this, we’ll use the `maxTriples` problem from lab. Here’s the description again:

Write a program called `maxTripleLength` that consumes a `LinkedList<String>` and produces the length of the longest concatenation of three consecutive elements. Assume the input contains at least three strings. For example, given a list containing `"a", "bb", "c", "dd", "e"`, the program would return 5 (for `"bb", "c", "dd"`).

6.1 Creating Classes

The second solution from lab created a new class for storing triples, and split the input list into triples as a first step. The `Triple` class had a single method for computing the combined length of the triple’s three string fields. Here’s how we do that in Scala:

```scala
val l1 = List(-5, 10, 0, -3, 5, -999, 20)
t.checkExpect(FunctionExamples.rainfall(l1), 5.0)
```
What’s most interesting here is what is missing relative to Java: there is no constructor! At least, not one that you write explicitly. The entire body of the class is the constructor in Scala. Any fields that you would have taken as part of the constructor in Java are just parameters to the class. You’ll see how to create additional fields across the remaining Scala lectures. (You can also create multiple constructors, as we did in Java, but that’s beyond what we are trying to do today).

### 6.2 Instantiating Objects from Classes

Now let’s write a function that creates a list of triples from a list of strings. The base case in this function occurs when the list has fewer than three elements. Here’s the code:

```scala
private def breakIntoTriples(l : List[String]): List[Triple] = {
  if (l.length < 3) {
    Nil
  } else {
    new Triple(l(0), l(1), l(2)) :: breakIntoTriples(l.tail)
  }
}
```

This code illustrates several Scala features:

- The if-statement syntax is the same as in Java
- `Nil` is the empty list
- `new` is used to create objects, just as in Java
- You can access specific list elements with a notation similar to array indexing from Java (just with parens instead of square brackets)
- Double colon (::) is `cons` (or `link` if you took cs19)
- As in OCaml, you have `head` and `tail` operations (methods) on lists (these were called `first` and `rest` in cs19)
- The access modifiers (private, etc), are the same as in Java.

### 6.3 Pattern Matching

Scala also supports pattern matching. We illustrate this feature through writing a function to take the max of a list of integers. This solution features a nested helper function that takes the current max element as an input:
/*
 * Compute the max of a list of integers.
 * Assumes list has at least one element
 */
def listMax(lst : List[Int]): Int = {
def maxHelp(xs : List[Int], currmax : Int): Int = xs match {
case Nil => currmax
case x :: tail => maxHelp(tail, math.max(x, currmax))
}
maxHelp(lst.tail, lst.head)
}

• Pattern matching uses the **match** and **case** keywords.

(for those from cs19, **match** and **case** are akin to **cases** from Pyret.)

This code relies on the input having at least one element. If we wanted to properly handle an empty list input, we would either (a) throw an exception (similar to Java), or (b) use an option type, which you saw in Java in one of the dynamic programming lectures.

From here, the overall **maxTripleLen** function is just a composition of the functions we wrote, plus a use of map:

/*/  
 * Computes the total max length of three consecutive strings from input list
 */
def maxTripleLen(l : List[String]): Int = {
  listMax(breakIntoTriples(l).map(t => t.totalLen()))
}

All of these functions would get added to the **FunctionExamples** object and tested in **FunctionsTest**. The posted source code shows how this looks all together.

## 7 Resources Pointers

For information on running programs in Scala within IntelliJ, see the IntelliJ setup guide on the course homepage.

As you start to program in Scala, you’ll likely find yourself with questions like “how do I do X in Scala”. Here are some resources you may find useful:

• The Scala “cheatsheet”, summarizing the core syntax:
  [https://docs.scala-lang.org/cheatsheets/](https://docs.scala-lang.org/cheatsheets/)

• The course style guide for Scala
  [http://cs.brown.edu/courses/csci0180/content/docs-spec/scala-style.pdf](http://cs.brown.edu/courses/csci0180/content/docs-spec/scala-style.pdf)
- The formal Scala reference (API) for all built-in classes (summarizes the methods available for each type, etc)
  
  [https://www.scala-lang.org/api/current/]

Feel free to post questions to Piazza as well. Sources like the StackOverflow programming Q&A site (easily found through web search) generally provides really good answers to programming questions, but the answers will often get into more advanced language features than you need for this course. If an answer you see on StackOverflow looks a bit beyond what we’ve been doing, you might want to clarify in office hours or on Piazza instead.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS18 document by filling out the anonymous feedback form: [https://cs.brown.edu/courses/cs018/feedback]