Lecture 19: The Many Hats of Scala: Functional

11:00 AM, Mar 5, 2021

Contents

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 previously, 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 a handfull of small illustrative programs in Scala.

2 Adding Two Numbers

Let's start very simply, writing a function that takes two integers and adds them. Here is that function in Scala:

```
def add1(x : Int, y : Int): Int = {
   x + y
}
```

Things to note:

- Scala has a single type `Int` for integers (as opposed to `int` vs `Integer` in Java)
- Type names go after parameter names, not before as in Java.
- There are no `return` statements: you indicate that the last expression gets returned by having a return type and = between the parameters and the open brace on the header line.

As in Java, all functions must live in classes. Had we been in Java, we might have set up a class such as the following with general-purpose helper functions:

```
class FunctionExamplesClass {
  FunctionExamples(){}
  Integer add1(...) ...
}

FunctionExamplesClass FunctionExamples = new FunctionExamplesClass()
```

One of Scala's goals is to reduce the amount of code needed for simple, common patterns. In practice, it is not uncommon to create a class either (a) just to hold some general-purpose functions, or (b) to create a class that you only ever intend to make one object from (the same Java code above covers both cases). Scala optimizes writing this pattern with its `object` keyword:

```
object FunctionExamples {
  def add1(x : Int, y : Int): Int = {
    x + y
  }
}
```

This Scala code does the same thing as the Java code. Under the hood, Scala creates a class for FunctionExamples and makes a single object from it. Note that this use of `object` in Scala is not the same as the `Object` type in Java (which covers all objects). In Scala, this "universal" type (with default `.equals`, `toString` and `.hashCode` methods) is called `Any`.

3 Running Programs

As in Java, Scala programs get run from a `main` method in some class. If you wanted a `Main` class like we have written in Java, you could write:

```
object Main {
  def main(args: Array[String]) = {
    System.out.println(FunctionExamples.add1(3, 4))
  }
}
```

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:

```
object Main extends App {
  System.out.println(FunctionExamples.add1(3, 4))
}
```

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

4 Writing Tests

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. For example:

```
import tester.Tester

object FunctionsTest {
  def testFuns(t : Tester) {
    t.checkExpect(FunctionExamples.add1(3, 4), 7)
  }
}

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

Unlike in Java, Scala lets you have multiple classes defined in the same file (and the filename does NOT need to match the name of a class). All of the code developed in this lecture (to this point and again to the end) can live in one Scala file.

5 Writing Recursive Functions: Summing a List

Here's a recursive `sum` function in Scala:

```
  object FunctionExamples {

  def sum(lst: List[Double]): Double = lst match {
    case Nil => 0
    case n :: tail => n + sum(tail)
  }
```

Scala supports pattern-matching with cases (as most of you saw in either Pyret or ReasonML). The key to setting up the pattern-match is the use of the `match` keyword before the open brace on the header line. Other things to note:

- `Nil` is the empty list
- `n :: tail` is the pattern for a non-empty list. `n` refers to the first element and `tail` refers to the rest. The specific variable names here are up to the programmer.
- Double colon (`::`) is `cons` (Racket) or `link` (Pyret).

6 Using Higher-Order Functions: Averaging Non-Negative Nums

Back when you last did functional programming, how would you write a function to compute the average of the non-negative numbers in a list?

One approach is to filter out the non-negative numbers, sum them, and divide by the list length. Let's write that in Scala:

```
def avgNonNeg(lst : List[Double]): Double = {
    val nonNegNums = lst.filter(n => n >= 0)
    sum(nonNegNums) / nonNegNums.length
}
```

Three new notes about Scala arise here:

- Semicolons are not needed at the ends of lines
- We've named an intermediate list of the non-negative numbers. In Java, we would have to write the type name before the variable name. Scala does not require us to write the type name (it will figure out the type from the value on the right side of the `=`. Instead, Scala asks us to indicate whether this name should be mutable. Here, we write `val`, indicating that our code will NOT mutate the `nonNegNums` value. If we wanted a mutable variable, we would write `var` instead of `val`.
- `n => n >= 0` shows Scala notation for lambda functions. This sample is equivalent to `lam(n): n >= 0`.

In general, Scala performs *type inference*, meaning you can omit types on parameters, fields, and return values (leaving Scala to compute them for you). That explains why our lambda expression here does not need a type on the parameter. Unlike Python, however, Scala will check the type it has determined before running your program, alerting you to any inconsistencies.

6.1 Creating Lists (for Testing)

Let's write a test case for `avgNonNeg` as an example of how to create list data in Scala:

```
import tester.Tester

object FunctionsTest {
  def testAvg(t : Tester) {
    val l1 = List(-5, 10, 0, -3, 5, -999, 20)
    t.checkExpect(FunctionExamples. avgNonNeg (l1), 5.0)
  }
}
```

The `List` operator creates an immutable list with the given elements in order. (We'll see in another couple of lectures how to create mutable lists).

6.2 Optional Side Note: summing with a higher-order function

We could have written the `sum` computation using a higher-order function called `foldLeft`. We're showing that here in case you are interested in options beyond `filter` and `map`.

```
def avgNonNeg(lst : List[Double]): Double = {
    val nonNegNums = lst.filter(n => n >= 0)
    nonNegNums.foldLeft(0)((a,b) => a + b) / nonNegNums.length
  }
```

- `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.

6.3 Additional higher-order functions

Imagine now that our list of numbers is actually streaming in from a weather monitor somewhere, and the monitor inserts a value -999 into the list to mark the end of the values over which to average. 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`, which returns the prefix of the list before some condition is met. Here's the code augmented with a call to `takeWhile`:

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

You can see the available higher-order functions in the Scala documentation:
`https://www.scala-lang.org/api/current/?search=list`

7 Defining Classes and Objects: The MaxTriples Problem

Let's use the following problem to illustrate classes, objects, and functional programming working together:

> 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”).

7.1 Creating Classes

We'll approach this by first breaking the list into triples, then computing the longest-length triple from that list. A triple is a new kind of data, so we will need to create a class. Furthermore, that class will need a method that computes the total length of the strings in that triple:

```
class Triple (s1 : String, s2 : String, s3 : String) {
  def totalLen: Int = {s1.length + s2.length + s3.length}
}
```

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 (appearing on the first line, after the class name). 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).
- Also not that a function that takes no arguments doesn't need to include empty parens (as in `()`) after the name of the function.

7.2 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:

```
/*
  * Create list of consecutive Triples from a list of Strings
  */
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
- **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)

7.3 Another Example of Pattern Matching

As a final example of pattern matching, let's write a function to find the largest number in a list. 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)
}
```

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 intput
    list
 */
def maxTripleLength(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.

8 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/`

- The course style guide for Scala (linked to the website)
- 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 Ed 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 Ed 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`.