Objectives

- (Practice) To develop function definitions in Scala and use these functions
- (Theory) To explain the basic principles of FP

Resources

You should refer to the following resources accessible via Blackboard:

- Topic 3 Lecture videos 3A, 3B, …
- **Topic 3 folder** (zipped) – includes the notes and slides for this topic
- External website for Scala doc and other **Learning Resources** (see the folder on Blackboard)

Introduction

Functional programming is not a new idea. Its roots go back to the lambda calculus work of Alonzo Church (https://en.wikipedia.org/wiki/Alonzo_Church) in the 1930s which led to the design of the LISP programming language, the first functional programming language, in the late 1950s. It is based on a model of computation which is different from the standard Von-Neumann (VN) model (1945). The latter describes the architecture of most digital computers in the past 80 years: a processing unit, RAM, registers, and a stored program with a program counter to step through the instructions. Programs work by specifying a sequence of updates to the state – the evolving state of the memory reflects the stages of the program as it runs.

The functional (FP) model, however, is very different and does not have the update of state at its core. For many years, the translation of the FP model onto standard VN hardware was difficult to achieve efficiently. This held back the widespread adoption of FP as a serious challenger to imperative program design and, later (1980s-), object-oriented design. Although taught at many universities across the world over the past 50 years due to its mathematical elegance and the insight it offers to algorithm design, commercial take-up of FP was very niche.

However, in the 1990s significant breakthroughs in compiler design were published for a language called Haskell. The latter, a purely functional, non-strict, programming language was used as the research bed for many ideas in this field, and it had a profound influence on the development of many other programming languages (including, e.g., Java, Scala, LINQ). Functional ideas that had been known to be useful for decades, such as closures, started to appear in mainstream languages (C#3.0, for example, introduced lambdas in 2007; C++11 introduced lambdas in 2011; Java 8 added support for lambda expressions in 2014). In fact, a list of languages that now support lambdas in one form or another can be seen at (https://en.wikipedia.org/wiki/Anonymous_function) which makes for entertaining reading.

The real game-changer in recent years was the widespread move to multi-core computer architectures. It is easier for the computer to distribute workloads across cores at runtime if the program does not make use of shared mutable state (which requires locks to manage – and locks do not scale well). Programs that use FP and immutable data structures can be distributed much more easily. Conveniently, the FP style is also a lot more concise than its imperative (and object-oriented) counterparts.

Activities

3.1 Watch the videos

Watch the videos 3A, 3B, etc. The accompanying slides can be found in the topic-3-folder. These videos give you the background to FP and explain some of the fundamental concepts.

For another view on the history and relevance of FP I recommend that you watch John Hughes' lecture on Why Functional Programming Matters (go to the **Library Resources** tab for the link). You can find Hughes' original paper there, too. It is a classic.

There is also a great video by Graham Hutton (Nottingham University) on the lambda calculus. You may like to play this too. The link can be found within the **Library Resources** tab.

3.2 Install the FunDemo 1 and 2 programs

Look at the file navigator hierarchy image on the next page to see how the various packages and Scala objects are organised. FunDemo1 has been highlighted in the picture as this is the file you will open to find the first set of practical exercises.

So, first of all you need to set up a package within **lib** called **sugar**. This will contain Scala definitions that provide some module-specific *syntactic sugar*. This term "syntactic sugar" is used in programming to refer to syntactic short-cuts that make code writing and understanding easier but which are not required technically. All modern programming languages contain redundant syntax which simplifies the task of writing code.

Within this package you should add the Scala *object* **QuestionSetting**. *Be careful to create a Scala **object** and not a Scala class*. Its contents are printed below:

```scala
package lib.sugar

object QuestionSetting {

  /* The itShould method is just a syntactic device we invented to make it easier
   * to describe the functions below. By using it we provide a default definition
   * for each of the uncompleted functions which type-checks and compiles. However,
   * at run-time an exception is thrown if you try to use an unimplemented
   * function.
   */
  def itShould[A](s: String): A = throw new Exception("Not implemented: " + s)

}
```

This allows us to write exercise questions for you to try and to specify them, e.g., like this:

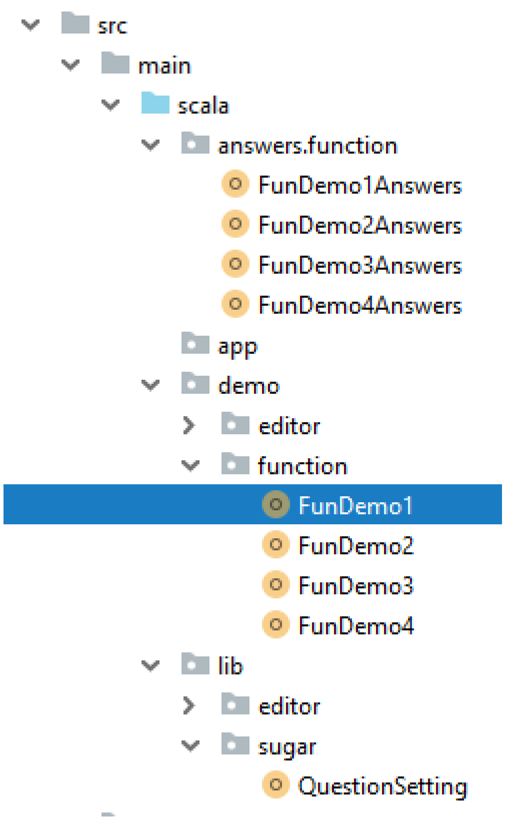
```scala
val lcm: Int => Int => Int =
 itShould("return the lowest common multiple of its arguments")
```

The result is a program that compiles but, when run, the exception is thrown to remind you that the particular value or definition is unimplemented. Your task is to replace the **itShould** call with your own code to implement the value or definition.

You should also set up a sub-package within **demo** called **function**. This is where you can store the **FunDemo1/2** objects which contain explanation, demonstration, and exercises. (This week you will only install Fundemo1 and FunDemo2 – next week you will add FunDemo3 and FunDemo4.) *Once again, be mindful that these demo programs are Scala **objects** not Scala classes*.

Finally, you might want to mirror the **demo.functions** package with one called **answers.functions**. This will give you a handy place to store the answers to the exercises separate from your original attempts. The answers will be published on Blackboard at a later date!

The easiest way to show the package structure that you should be constructing is to take a picture of the hierarchy from an existing set-up. (*The image shows the hierarchy within **IntelliJ**. In Eclipse you will most likely not have the intermediate **main->scala** folders*.)



```
src
  main
    scala
      answers.function
        FunDemo1Answers
        FunDemo2Answers
        FunDemo3Answers
        FunDemo4Answers
      app
      demo
        editor
        function
          FunDemo1
          FunDemo2
          FunDemo3
          FunDemo4
      lib
        editor
        sugar
          QuestionSetting
```

3.2 Work through the FunDemo 1 and 2 programs

Once you have set up your package structures as shown above you can begin the exercises. All of the tasks for this topic are contained within the Scala objects **FunDemo1** and **FunDemo2**. These files contain explanations and examples within comment blocks and also fragments of Scala code. You should read through each file carefully and attempt the exercises. We suggest you read them and attempt the exercises in sequence.

NB: The explanations within the Scala text files are not quite as easy to read as they would be if presented in *MS-Word*, for example, due to the lack of fonts and colours etc. for highlight. It is a trade-off because we wanted you to have the code and its explanation embedded in the same document. To mitigate the presentational restrictions this poses, we have used upper case to signpost the start of each exercise, and we have kept each **FunDemo** file relatively short. The alternative would be to provide the descriptions of each function/method in a separate document and ask you to cross-reference continually between the two. It was a moot choice. We welcome your feedback on this means of presentation and whether or not you feel that the compromise to keep the descriptions and explanations close to the exercises was worth it

The answers to the exercises will be published separately on Blackboard in due course once you have had an opportunity to try them.