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 4 Lecture videos 4A, 4B
- **Topic 4 folder** (zipped) – includes the notes and slides for this topic
- External website for Scala doc and other **Learning Resources** (see the folder on Blackboard). This contains a link to a nice lecture on recursion which will give you another explanation as well as those provided here.

Introduction

A very useful technique in mathematics and computer science is *recursion*. This is the ability for a function to be defined in terms of itself! Although this self-reference sounds odd, it does lead to very concise and readable solutions. Essentially, a function can be evaluated by delegation to (another instance of) itself. Consider the following function:

```
def countdown(n: Int): Unit = {
  print(s"$n ")
  if (n==0)
    println("Stopped")
  else
    countdown(n-1)
}
```

A call to countdown(5) will generate the following output:

5 4 3 2 1 0 Stopped

How does this work?

- countdown(5) calls the function with n=5
- 5 is printed and then, because 5 is not equal to zero, countdown(5-1) (i.e. countdown(4)) is called. At this point countdown(5) has not yet finished - it will only finish when the recursive call to countdown(4) has finished.
- countdown(4) has its own variable n which equals 4. This is printed. Then, because 4 is not equal to zero, countdown(4-1) (i.e. countdown(3)) is called.
- etc.
- and finally countdown(0) is called. The 0 is printed and then, because 0 is equal to zero, the function prints "Stopped" and then terminates. Control is passed back to the (suspended) countdown(1) which terminates and passes back control to countdown(2), which stops and

passes back control to countdown(3), which stops and passes back control to countdown(4), which stops and passes back control to countdown(5), which stops.

Classic example

```
def factorial(n: Long): Long = {
   if (n == 0)
      1
   else
     n * factorial(n - 1)
}
```

The function makes a call to itself. However, and this is crucial, note that the function's formal parameter, is n, but the recursive call is made with n-1. Thus the recursive call *makes progress towards zero*. Why is this important? Zero is the terminating condition: the condition that does not make another recursive call.

This pattern is common in many recursively defined functions. There will be some overall if-statement (or similar) in which:

- One branch does NOT have a recursive call: this is referred to as the *base case*. Typically, if a function has a numeric parameter, n (say), then the base case could be when n=0. If a function has a list parameter, xs (say), then the base case could be when the list is empty.
- One branch DOES have a recursive call. Importantly, the recursive call passes a parameter that gets closer to the base case. For a numeric parameter, n, this might be n-1, for example. For a list, xs, this might be the xs.tail, for example.

Finally, let us evaluate factorial(4) "on paper" (below) to show how the computation proceeds. Notice how factorial(1) cannot complete until factorial(0) has completed, and so on. The indentation reflects this:

factorial(5)

= 5 * factorial(4)

= 5 * (4 * factorial(3))

= 5 * (4 * (3 * factorial(2)))

= 5 * (4 * (3 * (2 * factorial(1))))

= 5 * (4 * (3 * (2 * (1 * factorial(0)))))

= 5 * (4 * (3 * (2 * (1 * 1))))

= 5 * (4 * (3 * (2 * 1)))

= 5 * (4 * (3 * 2))

= 5 * (4 * 6)

= 5 * 24

= 120

Data structures can also be recursive

The classic example of a recursive data structure is a singly-linked list. This will be the subject of a future topic. However, in this week's exercises we introduce you to Peano's numbers. Here are a couple of references:

https://mathworld.wolfram.com/PeanosAxioms.html

https://en.wikipedia.org/wiki/Peano_axioms

It is possible to represent the natural numbers like this:

Zero // the first natural number

Succ(Zero) // the successor of Zero – i.e. One

Succ(Succ(Zero)) // the successor of One – i.e. Two

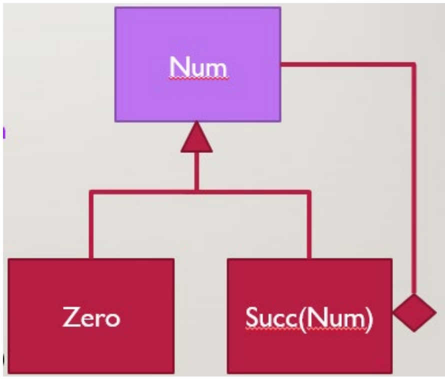
etc.

These numbers can be represented in a computer using a data structure like this:

```
sealed abstract class Num
case object Zero extends Num
case class Succ(n: Num) extends Num
```

The data structure is recursive (class **Succ** contains a value of type **Num**). It also resembles a singly-linked list – albeit one whose nodes do not store data items. In this case, the information we require is encoded by the *length* of the data structure. Let us assume that by *length* we mean the number of **Succ** constructors in the expression; then length 0 means Zero; length 1 means One; etc.

A class diagram shows the relationships.



The use of the Scala notation **case object** and **case class** for the concrete subclasses enables instances of these classes to be used in match expressions like this:

```
def toInt(n: Num): Int = n match {
        case Zero => 0
        case Succ(m) => 1 + toInt(m)
}
```

Activities

3.1 Watch the videos

Watch the videos 4A, 4B. The accompanying slides can be found in the topic-4-folder. These videos give you the background to FP and explain some of the fundamental concepts.

3.2 Install and run the FunctionDemo programs

Open your Scala IDE and within the **src** folder move to the **demo.funcion** package.

Copy the Scala files **FunDemo3.scala** and **FunDemo4.scala** into the **function** package.