# IT5001-L2M Assignment 9

Functional Programming in Scala

The goal of this assignment is to write code in a typed functional programming environment using functional programming principles and patterns.

This assignment is compulsory for MComp General Track students and is a replacement for Assignment 2. Submit your solutions to Coursemology by **9 Apr 2359hrs**. Late submissions will be penalized 33% per late day or part thereof.

## Preamble

We know how to write simple functions like factorial. However, we know that the factorial function is not safe!

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)
factorial(-1) # oh no!
```

We *could* throw exceptions in Scala as well. However, that is not transparent in the type signature of the function.

```
def factorial(n: Int): Int =
    if n < 0 then
        throw Exception()
    if n == 0 then 1
    else n * factorial(n - 1)

factorial(-1) // I would expect an Int, but I end up with an Exception!
```

Instead, we should allow the factorial function to be transparent in its effects by producing algebraic effects. To help us with that, we are going to create the `Either` algebraic data type!

## Problem 1: Either

Let us create the `Either[A, B]` Algebraic Data Type. This type represents a disjoint union of two types: a `Left[A]` of some value of type `A`, or a `Right[B]` of some value of type `B`. Simple, right?

Note the following:

- `Left[A]` <: `Either[A, B]` for all `A` and `B`.
- `Right[B]` <: `Either[A, B]` for all `A` and `B`.

- The values stored in `Left` and `Right` are immutable, because we are cool functional programmers.
- `Left` and `Right` should be easy to destructure, thus supporting convenient pattern matching.

Example uses of `Either` follow.

```
val e1: Either[String, Int] = Right(1) // Legal
val e2: Either[String, Int] = Left("Hello") // Legal
val b = e1 match // true
  case Left(_) => false
  case Right(_) => true
```

## Problem 2: Applicatives

We know that a Functor (1) `F[A]` can take (2) a function going from `A => B` and produce an `F[B]` by calling (2) and passing in the value stored in (1) producing a `B`, then putting back inside `F` to obtain an `F[B]`. This is a `map` operation. An Applicative (3) `F[A]` can take a function inside `F`, (4) `F[A => B]` and produce an `F[B]` by calling the function stored in (4) and passing in the value stored in (3) producing a `B`, then putting back inside `F` to obtain an `F[B]`. This is an `ap` (apply) operation.

To help us with later sections, we want the `Either` type to be both a Functor and an Applicative. This is so that `Either` instances support `map` and `ap`.

Importantly, both `map` and `ap` for `Either` are right-biased operations: they only operate on `Rights`! Therefore, when either of these operations are given an instance of `Left`, nothing is done.

The `Functor` typeclass and supporting extension methods and given instances have already been created for you. Therefore, you should be able to use `map` on `Either` instances right away!

```
val e1: Either[String, Int] = Right(1)
e1.map(_ + 2) // Right(3): Either[String, Int]
val e2: Either[String, Int] = Left("Hello")
e2.map(x => x.toString + "!") // Left("Hello"): Either[String, String]
```

Your job now is to define the `Applicative` typeclass and the supporting `given` instance for `Applicative[Either]`. Defining the `Applicative` typeclass is easy. All you need to do is to define the signature for the abstract method `ap`. Remember that `ap` takes in an `F[A]` and an `F[A => B]` and produces an `F[B]`.

Then, when writing your `given` instance, take note of the following:

- `Either` is a type constructor of two arguments. However, Applicatives are higher-kinds that take in a type constructor of only one argument. Therefore, to convert `Either` into a type constructor with one argument, we will have to fix the left type to some `A`, then the argument to the Applicative higher-kind needs to be a type lambda taking in some `B` and producing an `Either[A, B]`. See the `given` instance for the `Either` Functor as an example.

- Because `ap` will take in two `Either` instances, do nothing when either of them are instances of `Left`.

Once you have done both, you should be able to use `ap` right away using the extension method provided. Note that the extension method has the order of arguments reversed: `ap` in `Applicative` should first take in the `F[A]`, then the `F[A => B]`. The `ap` extension method operates on `F[A => B]` and accepts an `F[A]`! The reason for this swap is so that it is more convenient to build an `Either` from multiple `Either` instances. Suppose we have two `Either` instances where we do not know if they are lefts or rights:

```
// Pretend we didn't know these were rights (like they came from function calls)
val e1: Either[String, Int] = Right(1)
val e2: Either[String, String] = Right("Hello")
```

Then let's say we wanted to use these two `Either`s to build a pair:

```
case class Pair[+A, +B](left: A, right: B)
val f = (x: Int) => (y: String) => Pair(x, y)
```

Now we can easily build a pair of e1 and e2 like so:

```
e1.map(f).ap(e2) // Right(Pair(1, "Hello"))
```

And of course, the great thing about algebraic effects is that if either happen to be lefts (some error may have occurred), then we are still safe!

```
val e1: Either[String, Int] = Right(1)
val e2: Either[String, String] = Left("Error!")
e1.map(f).ap(e2) // Left("Error!")
```

When using `map` and `ap`, use the type signatures to help you. If you wanted to build an `Either[String, Pair[Int, String]]` from an `Either[String, Int]` and an `Either[String, String]`, you can

1. `map` the `Either[String, Int]` using a `Int => String => Pair[Int, String]` to produce a `Either[String, String => Pair[Int, String]]`
2. `ap` the `Either[String, String => Pair[Int, String]]` using the `Either[String, String]` to finally get the `Either[String, Pair[Int, String]]`!

Good luck!

## Problem 3: Factorial

Now that we have defined our algebraic effect types and methods, we can go ahead and define a safe factorial method. The factorial either returns an error message `"Error!"` or the factorial value if everything went well. Thus, your factorial method should return an `Either`.

Your factorial implementation should be recursive, because we are cool functional programmers.

Hint: use tail recursion and the `BigInt` type to handle large numbers.

## Problem 4: n choose k

Let us test our skills even further. We are going to define a method `nChooseK` as such:

$$nChooseK(n, k) = \frac{n!}{k!(n-k)!}$$

This method definition uses factorials which is unsafe. Fret not, since we already have our safe factorial method! Thus, define the `nChooseK` method using our safe `factorial` method. In addition, use `map` and `ap` since they make our lives easier.

## Instructions and Submission

You are given two files:

1. `template.scala`. This template file contains incomplete implementations of the above. Edit this file.
2. `test.scala`. This file contains the testing suite. Do not edit this file.

Work on the problems above in `template.scala`. Whenever you're ready, you may test your program by compiling and running it.

**Commands for compiling and running your code**:

```
scala3-compiler *.scala
scala3 test
```

For submission, please rename `template.scala` into `<your name>_<date and time>.scala`. Upload this file to Coursemology.

## Grading

The assignment is graded out of 100. 75 marks are for correctness, 25 marks are for program design. There are 25 test cases. As long as your code compiles, you should be able to pass many of the test

cases. Each test case is worth 3 marks. Generally, if you use functional programming principles and patterns, you will be able to obtain the full 25 marks for design.

**Partial credit will be given** even if you fail the test cases.