# INFO0054 Programmation Fonctionnelle – Exercises

Christophe Debruyne

## Exercises 6: Non-Strict Evaluation

### Class discussion

- Explain in your own words "call by name."
- Explain in your own words "call by need."
- How does one use call by name and call by need in Scala?
- What's the main-motivation for call by name in functional programming?
- What's the main-motivation for call by need in functional programming?

**Exercise 1:**

Non-strict evaluation is a concept that many functional programming languages support. Not many imperative languages support this, though they incorporated this for Boolean operators. Using `println` statements, demonstrate the behavior of lazy evaluation in Scala's REPL environment. Can you explain the behavior?

**Exercise 2:**

Implement `myif`, a function that takes as input an argument `test` that contains a Boolean value, and two arguments `onIf` and `onElse` that contains expressions evaluating to objects of the same type `A`. Ensure that `myif` simulates the behavior of an if-statement. Demonstrate your function. Implement `myif2` that relies on currying.

Discussion: When and why would you use currying?

**Exercise 3:**

Implement `myifelseifelse`, a function that simulates an if-else if-else statement. Your `myifelseifelse` should rely on `myif`.

```
scala> myifelseifelse(
     |     { println("t1") ; false },
     |     { println("a") ; "a" },
     |     { println("t2") ; true },
     |     { println("b") ; "b" },
     |     { println("c") ; "c" })
t1
t2
b
val res0: String = b
```

**Exercise 4:**

Given the following function `foo`

```scala
def foo(x: => Int): Int =
    x + x
```

How many times will the expression we give the function `foo` be evaluated? Can you demonstrate this? How can we ensure that the expression is only evaluated once?

**Exercise 5:**

Given our ADT `Flux`, implement the following functions:

1. `takeWhile` returning, from the beginning of a `Flux`, all elements that satisfy a condition.
2. `exists` checks whether an element satisfying a condition exists.
3. `foldRight`, with which you should already be familiar. ;-)

```scala
import Flux.*

enum Flux[+A]:
    case Empty
    case Cons(h: () => A, t: () => Flux[A])

    def headOption: Option[A] = this match
        case Empty => None
        case Cons(h, t) => Some(h())

    def toList: List[A] = this match
        case Cons(h,t) => h() :: t().toList
        case Empty => Nil

    def take(n: Int): Flux[A] = this match
        case Cons(h, t) if n > 1 => cons(h(), t().take(n - 1))
        case Cons(h, _) if n == 1 => cons(h(), empty)
        case _ => empty

    def filter(f: A => Boolean): Flux[A] = this match
        case Cons(h, t) if f(h()) => cons(h(), t().filter(f))
        case Cons(_, t) => t().filter(f)
        case _ => empty

    def map[B](f: A => B): Flux[B] = this match
        case Cons(h, t) => cons(f(h()), t().map(f))
        case _ => empty

    def takeWhile(p: A => Boolean): Flux[A] = ???
    def exists(p: A => Boolean): Boolean = ???
    def foldRight[B](acc: => B)(f: (A, => B) => B): B = ???

object Flux:
    def cons[A](hd: => A, tl: => Flux[A]): Flux[A] =
        lazy val head = hd
        lazy val tail = tl
        Cons(() => head, () => tail)

    def empty[A]: Flux[A] = Empty

    def apply[A](as: A*): Flux[A] =
        if (as.isEmpty) empty
        else cons(as.head, apply(as.tail*))
```

# References

[1] Paul Chiusano and Rnar Bjarnason. 2015. Functional Programming in Scala (2nd. ed.). Manning Publications Co., USA.