INFO0054 Programmation Fonctionnelle – Exercises

Christophe Debruyne

Exercises 02: Higher-order programming

Exercise 1:

What will the evaluation of the expression on the right-hand side of each variable assignment return?

```
scala> val a = (x: Int) => x + x
scala> val b = ((x: Int) => x + x)(5)
scala> val c = a(10)
scala> val d = List(1,2,3,4).map(_ * 2)
scala> val e = List((1,2),(3,4),(5,6)).unzip
scala> val (f, g) = List((1,2),(3,4),(5,6)).unzip
scala> val h = List.range(1,10,2)
scala> val i = List.range(2,11,2)
scala> val j = h.zip(i)
scala> val k = j.filter(_ + _ < 10)
```

Exercise 2:

Can you rewrite the following expressions so that they use function literals with variables?

```
scala> val d = List(1,2,3,4).map(_ * 2)
scala> val k = j.filter(_ + _ < 10)
```

Exercise 3:

Implement `findIndexOfLast`, a higher-order function with the following signature:

```
def findIndexOfLast[A](arr: Array[A], cond: A => Boolean): Int = ???
```

Exercise 4:

(Based on exercise 2.2 in [1]) Implement `isSorted`, which checks whether an `Array[A]` is sorted according to a given comparison function:

```
def isSorted[A](as: Array[A], comp: (A,A) => Boolean): Boolean = ???
```

The function `isSorted` is used as follows:

```
scala> isSorted(Array(1,2,2,3,4), (x, y) => x < y)
val res1: Boolean = false

scala> isSorted(Array(1,2,2,3,4), (x, y) => x <= y)
val res2: Boolean = true
```

Exercise 5:

Define `countIf`, which, given an array, returns the number of elements that satisfy a condition.

Exercise 6:

Given the code from our introductory example.

```
class PokemonGO:
    def buyPouch(a: Account): (Pouch, Charge) =
        val pouch = new Pouch()
        (pouch, Charge(a, pouch.price))

    def buyPouches(a: Account, n: Int): (List[Pouch], Charge) =
        val purchases: List[(Pouch, Charge)] = List.fill(n)(buyPouch(a))
        val (pouches, charges) = purchases.unzip
        (pouches, charges.reduce((p1, p2) => p1.combine(p2)))

    def coalesce(charges: List[Charge]): List[Charge] =
        charges.groupBy(_.acc).values.map(_.reduce(_ combine _)).toList

case class Charge(acc: Account, amount: Double):
    def combine(other: Charge): Charge =
        if (acc == other.acc)
            Charge(acc, amount + other.amount)
        else
            throw new Exception("Cannot combine charges of different accounts.")

/* Empty classes and methods for this example */
case class Pouch():
    val price = 0.99
case class Account(name: String)
```

We are going to instantiate some of these classes to create an example. The variable `t` contains a list of transactions using tuples. We then use that list of transactions to create a new list of charges. Can you explain what is happening on line 5?

```
val p = new PokemonGO()
val g = Account("gaston")
val v = Account("victor")
val t = List((g, 10), (g, 5), (v, 2), (g, 10), (v, 2))
val charges = t.map(x => Charge(x(0), x(1)))
```

What would be the result of:

```
val res = p.coalesce(charges)
```

Can you rewrite the `coalesce` function using anonymous functions with explicit parameters? You may use variables to store intermediate results.

Exercise 7:

(Challenging!) If you have solved the previous exercise, you will have noticed that we have "chopped up" one line of code into multiple lines by storing intermediate results in immutable (!) variables. This might seem to go against the idea that a purely function program is just a composition of function. In this exercise, you will be asked to look at the SICP book [2] and formulate an argument as to why this is not the case. Hint: look at the section "Using let to create local variables" starting on page 85.

References

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

[2] Abelson, Harold, and Gerald Jay Sussman. Structure and Interpretation of Computer Programs. The MIT Press, 1996. `https://web.mit.edu/6.001/6.037/sicp.pdf`