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)
```

Solution 1:

- The evaluation of a function literal.
- The application of a function literal with the argument 5; the value is 10.
- The application of the function assigned to the variable a with the argument 10; the result is 20.
- The list containing the values 2, 4, 6, and 6.
- The expression on the RHS of the assignment operator yields a tuple of arity two. The first element is a list containing the values 1, 3, and 5. The second element is a list containing the values 2, 4, and 6. In this case, the tuple is assigned to the variable e.
- The expression on the RHS of the assignment operator yields a tuple of arity two. In this case, a list containing the values 1, 3, and 5 will be assigned to the variable f. And a list containing the values 2, 4, and 6 will be assigned to the variable g.
- A list containing the values 1, 3, 5, 7, and 9.
- A list containing the values 2, 4, 6, 8, and 10.
- A list containing the tuples (1,2), (3,4), (5,6), (7,8), and (9,10).
- A list containing the tuples (1,2) and (3,4).

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)
```

Solution 2:

```
// Scala is able to infer the type of x
scala> val d = List(1,2,3,4).map((x) => x * 2)
// We can add the type, however
scala> val d = List(1,2,3,4).map((x: Int) => x * 2)
// We can also store the function in a separate variable
scala> val dfun = (x: Int) => x * 2
scala> val d = List(1,2,3,4).map(dfun)
```

```
scala> val k = j.filter((x, y) => x + y < 10)
```

Exercise 3:

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

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

Solution 3:

```
    def findIndexOfLast[A](arr: Array[A], cond: A => Boolean): Int =
        @annotation.tailrec
        def loop(n: Int): Int =
            if(n < 0) -1
            else if (cond(arr(n))) n
            else loop(n - 1)
        loop(arr.size - 1)
```

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
```

Solution 4:

```scala
def isSorted[A](as: Array[A], comp: (A,A) => Boolean): Boolean = {
    @annotation.tailrec
    def helper(n: Int): Boolean = {
      if (n >= as.length - 1) then true
      else if (!comp(as(n), as(n + 1))) then false
      else helper(n + 1)
    }
    helper(0)
}
```

Watch out: if the Array is empty, then Scala is unable to infer the types from the array and the function literal. In that case, the array should have a type parameter.

```scala
scala> isSorted(Array[Int](), (x,y) => x < y)
val res4: Boolean = true
```

Exercise 5:

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

Solution 5:

```scala
    def countIf[A](arr: Array[A], cond: A => Boolean): Int =
        @annotation.tailrec
        def loop(n: Int, acc: Int): Int =
            if(n >= arr.length) acc
            else loop(n + 1, if(cond(arr(n))) acc + 1 else acc + 0)
        loop(0, 0)
```

The following function definitions are examples of how Scala's built-in libraries support higher-order functions. These will be explore in more detail in the next few weeks.

```scala
    def countIf2[A](arr: Array[A], cond: A => Boolean): Int =
        arr.filter(cond).size

    def countIf3[A](arr: Array[A], cond: A => Boolean): Int =
        arr.count(cond)
```

Exercise 6:

Given the code from our introductory example.

```scala
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.

Solution 6:

The function `map` applies a function to each list element and returns a new list. In this case, each element is a tuple, and each tuple consists of an account and an integer. The account and integer, representing the price, are retrieved using `x(0)` and `x(1)` to create new instances of `Charge`. The resulting list contains five `Charge` objects.

```
val charges: List[Charge] = List(
   Charge(Account(gaston),10.0),
   Charge(Account(gaston),5.0),
   Charge(Account(victor),2.0),
   Charge(Account(gaston),10.0),
   Charge(Account(victor),2.0)
)
```

The application of the `coalesce` function returns a list containing two charge objects, one for each account and whose charge is the total of each account's charges.

```
val res: List[Charge] = List(Charge(Account(gaston),25.0), Charge(Account(victor),4.0))
```

```
    def coalesce(charges: List[Charge]): List[Charge] =
        // To lists of charges grouped by account
        val groups = charges.groupBy(e => e.acc).values
        // Map the list of lists and reduce them
        val reduced = groups.map(list => list.reduce((a, b) => a.combine(b)))
        // Transform the result into a List
        reduced.toList
```

In this case, the conversion to a list can be omitted as the `map` function applied to a list will return a list. Later in this course, however, we will see that we will not care about the underlying data structure for applying functions such as `map`, `fold`, `foldRight`, etc. However, we sometimes want to convert a particular data structure explicitly. In this example, the application of `toList` to a list returns itself.

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 functions. 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.

Solution 7:

```
val x = <expression1>
val y = <expression2>
/// ...
<body>
```

is "basically" syntactic sugar for

```
((x, y) => {
    /// ...
    <body>
})(<expression1>, <expression2>)
```

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`