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?

Solution 1:

```
scala> { println("1") ; true } && { println("2") ; true } && { println("3") ; true }
1
2
3
val res0: Boolean = true

scala> { println("1") ; true } && { println("2") ; false } && { println("3") ; true }
1
2
val res1: Boolean = false

scala> { println("1") ; true } || { println("2") ; true } || { println("3") ; true }
1
val res2: Boolean = true

scala> { println("1") ; false } || { println("2") ; true } || { println("3") ; true }
1
2
val res3: Boolean = true
```

The operands of Boolean operations are evaluated from left to right, bearing in mind that the logical AND has precedence over the logical OR.

A logical AND will yield true if all its operands yield true, which weans that all expressions need to be evaluated. A logical AND will stop the evaluation of its operands when it encounters an expression that evaluates to false.

A logical AND will yield true when it encounters its first expression that evaluates to true. It will return false when all of its operands evaluate to false.

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?

Solution 2:

```
def myif[A](test: Boolean, onIf: => A, onElse: => A): A =
    if test then onIf else onElse

def myif2[A](test: Boolean)(onIf: => A)(onElse: => A): A =
    if test then onIf else onElse
```

```
scala> myif(3 > 5, 2 / 0, 5)
val res0: Int = 5

scala> myif2(3 > 5)(2 / 0)(5)
val res1: Int = 5
```

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
```

Solution 3:

```
def myifelseifelse[A](t1: Boolean, i: => A, t2: => Boolean, ei: => A, e: => A): A =
    myif(t1, i, myif(t2, ei, e))
```

Exercise 4:

Given the following function `foo`

```
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?

Solution 4:

The expression will be evaluated twice.

```
scala> foo({ println("bar") ; 2 })
bar
bar
val res3: Int = 4
```

We can ensure that the expression passed to the function with call by name is evaluated with an additional variable: a "regular" variable in this case will suffice. We can also use a variable call by need.

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. ;-)

```
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*))
```

Solution 5:

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

    def exists(p: A => Boolean): Boolean = this match
        case Cons(h, t) => p(h()) || t().exists(p)
        case _ => false

    def foldRight[B](acc: => B)(f: (A, => B) => B): B = this match
        case Cons(h,t) => f(h(), t().foldRight(acc)(f))
        case _ => acc
```

References

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