INFO0054 Programmation Fonctionnelle – Exercises

Christophe Debruyne

Exercises 5: ADTs and Exception Handling

Preamble

Given the following ADT for trees:

```
enum Tree[+A]:
    case Leaf(label: A)
    case Branch(left: Tree[A], right: Tree[A])

object Tree:
    def size[A](t: Tree[A]): Int = t match
        case Leaf(_) => 1
        case Branch(l, r) => size(l) + size(r) + 1
```

And the following trees which you can use to test your code:

```
import Tree._
val ti = Branch(Branch(Leaf(1),Leaf(2)),Branch(Leaf(3),Leaf(4)))
val ts = Branch(Branch(Leaf("01"),Leaf("02")),Branch(Leaf("03"),Leaf("04")))
```

Exercise 1:

Write, inside the companion object, a function `biggestInt` that takes as input a `Tree[Int]` and returns the largest integer inside that tree. Is this function total? Explain your answer. In case it is not total, how can you manage exceptions?

Solution 1:

By definition, an object of type Tree[Int] has at least one Int and the function is therefore total.

```
object Tree:
    def biggestInt(t: Tree[Int]): Int = t match
        case Leaf(i) => i
        case Branch(l, r) => biggestInt(l).max(biggestInt(r))
```

```
scala> biggestInt(ti)
val res0: Int = 4
```

Exercise 2:

Write, inside the companion object, a function `firstPositive` that takes as input a `Tree[Int]` and returns the first strict positive integer inside that tree. Is this function total? Explain your answer. In case it is not total, how can you manage exceptions?

Solution 2:

Trees of type `Tree[Int]` contain at least on Int, but those integers may all be lower than or equal to zero. This function is therefore not total and we could manage exceptions with `Option` or `Either`.

```
object Tree:
    def firstPositive(t: Tree[Int]): Option[Int] = t match
        case Leaf(i) => if i > 0 then Some(i) else None
        //case Branch(l, r) => firstPositive(l).orElse(firstPositive(r))
        case Branch(l, r) => firstPositive(l) orElse firstPositive(r)
```

Notice how you can write `<exp1>.fun(<exp2>)` as `<exp1> fun <exp2>` in Scala.

```
scala> firstPositive(ti)
val res0: Option[Int] = Some(1)

scala> firstPositive(Leaf(-5))
val res1: Option[Int] = None
```

Exercise 3:

> Exploring Scala features: extension methods
>
> Last week, we've seen that some ADTs have functions whose availability depend on their contents. For instance, objects of type `List[Int]` have utility functions such as `sum`. In this exercise, you will create your own functions that depend on the type of objects stored!

So far, we have declared ADTs with invariant or covariant type parameters. All functions that are declared on these ADTs (either in a companion object, our case classes, or our enum) operate on ADTs using As. I.e., the `size` function operates on every tree, regardless of the type of objects.

Some functions only make sense for certain types of trees; e.g., `biggestInt` for returning the biggest integer is a tree of integers. Defining this as a function of `Tree[A]` will not be appreciated by Scala. Instead, we should declare it as a function of `Tree[Int]`.

Is it possible to "extend" our Tree? Yes, with **extension methods**. "An extension method allows defining new methods to third-party types, without making any changes to the original type."[1].

Write, inside the companion object, extension methods for `Tree[Int]` and `Tree[String]` for the following functions:

- A function `biggestInt` for objects of the type `Tree[Int]`. This function returns the biggest Int inside a tree. Is this function total? Explain your answer. In case it is not total, how can you manage exceptions?
- A function `firstPositive` for objects of the type `Tree[Int]`. This function returns the first strict positive integer it finds in a tree. Is this function total? Explain your answer. In case it is not total, how can you manage exceptions?
- A function `esrever` for objects of the type `Tree[String]`. This function reverses all strings in a tree. Is this function total? Explain your answer. In case it is not total, how can you manage exceptions?

If you write those definitions in the same file as the previous exercises, you will have to either rename or comment out those functions.

Solution 3:

```
object Tree:
    extension (t: Tree[Int])
        // We extend objects of type Tree[Int] with a method biggestInt.
        // By definition, an object of type Tree[Int] has at least one Int.
```

[1] https://www.baeldung.com/scala/extension-methods

```
        def biggestInt: Int = t match
            case Leaf(i) => i
            case Branch(l, r) => l.biggestInt.max(r.biggestInt)

        // We extend objects of type Tree[Int] with a method firstPositive.
        // Trees of type Tree[Int] could contain no positive integers.
        // We therefore need to manage those exceptions!
        def firstPositive: Option[Int] = t match
            case Leaf(i) => if i > 0 then Some(i) else None
            case Branch(l, r) => l.firstPositive.orElse(r.firstPositive)

    extension (t: Tree[String])
        // We extend objects of type Tree[String] with a method esrever.
        // By definition, an object of type Tree[String] has at least one String.
        def esrever: Tree[String] = t match
            case Leaf(i) => Leaf(i.reverse)
            case Branch(l, r) => Branch(l.esrever, r.esrever)
```

Exercise 4:

In Chapter 04, we have covered the ADTs `Option` and `Either`. We have provided implementations for some functions on `Option`. The only function we have covered for `Either` is `toList`. In this exercise, you are asked to complete the following ADT.

```
enum Either[+E, +A]:
    case Left(get: E)
    case Right(get: A)

    def toList: List[A] = this match
        case Right(a) => List(a)
        case Left(_) => List()

    def map[B](f: A => B): Either[E, B] = ???
    def flatMap[EE >: E, B](f: A => Either[EE, B]): Either[EE, B] = ???
    def orElse[EE >: E, AA >: A](b: => Either[EE, AA]): Either[EE, AA] = ???
    def getOrElse[B >: A](default: => B): B = ???
    def toOption: Option[A] = ???
```

Solution 4:

```
    def map[B](f: A => B): Either[E, B] = this match
        case Right(a) => Right(f(a))
        case Left(e) => Left(e)

    def flatMap[EE >: E, B](f: A => Either[EE, B]): Either[EE, B] = this match
        case Left(e) => Left(e)
        case Right(a) => f(a)

    def orElse[EE >: E, AA >: A](alt: => Either[EE, AA]): Either[EE, AA] = this match
        case Left(_) => alt
        case Right(a) => Right(a)

    def getOrElse[B >: A](default: => B): B = this match
        case Left(_) => default
        case Right(a) => a
```

```
    def toOption: Option[A] = this match
        case Left(_) => None
        case Right(a) => Some(a)
```

Exercise 5:

Now demonstrate your ADT. For the demonstration, consider implementing a safe division and implementing a save version of `fib` (i.e., one that manages the exception of negative arguments). Try lifting a function as well.

```
import Either._

def lift[A,B](f: A =>B): Either[_,A] => Either[_,B] = _.map(f)

def safeDiv(x: Int, y: Int): Either[Exception, Int] = ???
def safeFib(n: Int): Either[String, Int] = ???
```

Solution 5:

```
import Either._

def safeDiv(x: Int, y: Int): Either[Exception, Int] =
    try Right(x / y)
    catch case e: Exception => Left(e)

def safeFib(n: Int): Either[String, Int] =
    if n < 0 then Left("Argument is negative.")
    else if n < 2 then Right(n)
    else safeFib(n - 1).flatMap(a => safeFib(n - 2).map(b => a + b))
```

```
scala> safeDiv(10,2)
val res0: Either[Exception, Int] = Right(5)

scala> safeDiv(10,2).map(x => x + 2)
val res1: Either[Exception, Int] = Right(7)

scala> safeDiv(10,2).map(x => x + 2).getOrElse("Oops")
val res2: Matchable = 7

scala> safeDiv(10,2).map(x => x + 2).orElse(Left("Oops"))
val res3: Either[Object, Int] = Right(7)

scala> safeDiv(10,2).flatMap(safeDiv(_, 2))
val res4: Either[Exception, Int] = Right(2)

scala> safeDiv(10,0)
val res5: Either[Exception, Int] = Left(java.lang.ArithmeticException: / by zero)

scala> safeDiv(10,0).map(x => x + 2)
val res6: Either[Exception, Int] = Left(java.lang.ArithmeticException: / by zero)

scala> safeDiv(10,0).map(x => x + 2).getOrElse("Oops")
val res7: Matchable = Oops

scala> safeDiv(10,0).map(x => x + 2).orElse(Left("Oops"))
val res8: Either[Object, Int] = Left(Oops)
```

```
scala> safeDiv(10,0).flatMap(safeDiv(_, 2))
val res9: Either[Exception, Int] = Left(java.lang.ArithmeticException: / by zero)
```

```
def fib(n: Int): Int =
    if n < 2 then n
    else fib(n - 1) + fib(n - 2)

def fib0 = lift(fib)
```

```
scala> val x = List((2,3),(2,2),(2,1),(2,0)).map((x,y) => safeDiv(x, y)).map(fib0)
val x: List[Either[?, Int]] = List(
    Right(0),
    Right(1),
    Right(1),
    Left(java.lang.ArithmeticException: / by zero))
```

Here's an alternative with for-expressions:

```
def safeFib2(n: Int): Either[String, Int] =
    if n < 0 then Left("Argument is negative.")
    else if n < 2 then Right(n)
    //else safeFib(n - 1).flatMap(a => safeFib(n - 2).map(b => a + b))
    for
        a <- safeFib(n - 1) // represents the outer flatMap
        b <- safeFib(n - 2) // represents the inner map
    yield a + b // the expression to be computed
```

Here's an alternative that is tail-recursive:

```
def safeFib3(n: Int): Either[String, Int] =
    @annotation.tailrec
    def helper(n: Int, a: Int, b: Int): Either[String, Int] =
        if n < 0 then Left("Argument is negative.")
        if n == 0 then Right(a)
        else helper(n - 1, b, a + b)
    helper(n, 0, 1)
```

References

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