INFO0054 Programmation Fonctionnelle – Exercises

Christophe Debruyne

Exercises 4: ADTs and Recursion

Exercise 1:

> **Attention!**
>
> We have seen this example in class. This is a warm up exercise. Try not to look at the slides (especially for the first part of the exercise).

The first two Fibonacci numbers are 0 and 1. The nth Fibonacci number is always the sum of the previous two Fibonacci numbers. The sequence begins with 0, 1, 1, 2, 3, 5,... You may assume that n=0 corresponds with the first Fibonacci number.

Define `fib1`, a recursive function to get the nth Fibonacci number. Is your definition tail recursive? Justify your answer. If `fib1` is tail-recursive, define a version `fib2` that is recursive, but not tail recursive. If `fib1` is recursive, `fib2` should use a local tail-recursive function.

```
scala> val x = List.range(0,15).map(fib1)
val x: List[Int] = List(0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377)
```

Solution 1:

```
def fib1(n: Int): Int = {
    if (n <= 1) n
    else fib1(n - 1) + fib1(n - 2)
}
```

`fib1` is not tail recursive as the result of the recursive call is used in a subsequent computation. In other words, the result of the recursive call is not immediately returned.

```
def fib2(n: Int): Int = {
    @annotation.tailrec
    def loop(n: Int, a: Int, b: Int): Int = {
      if (n == 0) then a
      else loop(n - 1, b, a + b)
    }
    loop(n, 0, 1)
}
```

Exercise 2:

Define `sumInt1`, a recursive function that takes as argument a natural number (positive integer in Scala) n and returns the sum of all natural numbers lower than or equal to n. Is your definition tail recursive? Justify your answer. If `sumInt1` is tail-recursive, define a version `sumInt2` that is recursive, but not tail recursive. If `sumInt1` is recursive, `sumInt2` should use a local tail-recursive function.

Solution 2:

```
// for a natural number n, sumInt1(n) = n + n-1 + ... + 0
def sumInt1(n: Int): Int =
    if n == 0 then 0
    else n + sumInt1(n - 1)

// sumInt1 is not tail recursive as the result of the recursive call
// is used in an expression. In other words, the result of the recursive
// call is not returned immediately.

// for a natural number n, sumIn21(n) = n + n-1 + ... + 0
def sumInt2(n: Int): Int =
    // for a natural number n and an integer acc, iter(n, acc) = n + n-1 + ... + acc
    @annotation.tailrec
    def iter(n: Int, acc: Int): Int =
        if n == 0 then acc
        else iter(n - 1, acc + n)
    iter(n, 0)
```

Exercise 3:

Define `power1`, a recursive function taking as arguments a number x and a natural number n, and returning x^n. Is your definition tail recursive? Justify your answer. If `power1` is tail-recursive, define a version `power2` that is recursive, but not tail recursive. If `power1` is recursive, `power2` should use a local tail-recursive function.

Hard(er): We know that if n is even, then $x^n = (x * x)^{\frac{n}{2}}$. Modify the solutions above to render it more efficient.

Solution 3:

```
// For an integer x and an natural number n, power1(x,n) = x^n
def power1(x: Int, n: Int): Int =
    if n == 0 then 1
    // else if n % 2 == 0 then power1(x * x, n / 2)
    else x * power1(x, n - 1)

// power1 is not tail recursive as the result of the recursive call
// is used in an expression. In other words, the result of the
// recursive call is not returned immediately.

// For an integer x and an natural number n, power1(x,n) = x^n
def power2(x: Int, n: Int): Int =
    // For an integer x, a natural number n, and an integer acc,
    // iter(x, n, acc) = x^n * acc
    @annotation.tailrec
    def iter(x: Int, n: Int, acc: Int): Int =
        if n == 0 then acc
        // else if n % 2 == 0 then iter(x * x, n / 2, acc)
        else iter(x, n - 1, acc * x)
    iter(x, n, 1)

// The addition of the extra recursive step on indirect components '
// of n turns this function that relies on structural recursion
// into a function that relies on complete structural recursion.
```

Exercise 4:

Define `taken1`, a recursive function taking as arguments a list `l` and a natural number `n`. The function returns a new list containing the first `n` elements of that list (or fewer if the list does not contain enough elements).

What kind of recursion is this?

Is your definition tail recursive? Justify your answer. If `taken1` is tail-recursive, define a version `taken2` that is recursive, but not tail recursive. If `taken1` is recursive, `taken2` should use a local tail-recursive function.

Can you come up with two strategies for implementing this function using tail recursion? What are those two strategies and which one is more efficient?

Solution 4:

This is an example of mixed recursion, as it recurses over two arguments: `n` and `l`. In mixed recursion, we process the two ADTs in a structural manner (i.e., using their direct or indirect components). Notice that this solution has two base cases, one for each ADT.

```
def taken1[A](l: List[A], n: Int): List[A] =
    if n == 0 || l.isEmpty then Nil
    else l.head :: taken1(l.tail, n - 1)

def taken2[A](l: List[A], n: Int): List[A] =
    @annotation.tailrec
    def iter(l: List[A], n: Int, acc: List[A]): List[A] =
        if n == 0 || l.isEmpty then acc
        else iter(l.tail, n - 1, acc ++ List(l.head))
    iter(l, n, Nil)

def taken3[A](l: List[A], n: Int): List[A] =
    @annotation.tailrec
    def iter(l: List[A], n: Int, acc: List[A]): List[A] =
        if n == 0 || l.isEmpty then acc
        else iter(l.tail, n - 1, l.head :: acc)
    iter(l, n, Nil).reverse
```

There are two strategies for defining this function using tail recursion: appending lists and reversing a list created with cons. While the definition with append is arguably more intuitive, it is inefficient. Each call to append requires O(n) steps where n corresponds with the size of the list in the accumulator. The cons operator returns a new list in constant time and reversing the list just requires 0(n).

Exercise 5:

Create an ADT for `LTrees`, which are labelled binary trees. The constructors are `LLeaf` and `LBranch`.

Solution 5:

```
scala> LBranch(3, LLeaf(1), LBranch(2, LLeaf(1), LLeaf(1)))
val res0: LTree[Int] = LBranch(3,LLeaf(1),LBranch(2,LLeaf(1),LLeaf(1)))
```

Now use your ADT to create a representation for the following mathematical expression on Doubles: $((3.0 + 5.0) + (3.0 - 4.0)) \times (3.0/2.0)$. You can use the String objects "ADD", "SUB", "DIV", and "MUL" to represent the arithmetic operations. By mixing String and Int objects, you will obtain a `LTree[Matchable]`

Then create a function `compute` in `LTree`'s companion object that, given an `LTree` containing an arithmetic expression, computes the result. You may assume that there the tree contains valid values.

Question: Given the lecture on exception handling, how could you solve this exercise using `Option` or `Either`?

Solution 6:

```
enum LTree[+A]:
    case LLeaf(label: A)
    case LBranch(label: A, left: LTree[A], right: LTree[A])

object LTree:
    def compute(t: LTree[_]): Double = t match
        case LLeaf(l: Double) => l
        case LBranch("ADD", left, right) => compute(left) + compute(right)
        case LBranch("SUB", left, right) => compute(left) - compute(right)
        case LBranch("DIV", left, right) => compute(left) / compute(right)
        case LBranch("MUL", left, right) => compute(left) * compute(right)


import LTree._

val test = LBranch( "MUL",
                    LBranch("ADD",
                            LBranch("ADD",
                                    LLeaf(3.0),
                                    LLeaf(5.0)),
                            LBranch("SUB",
                                    LLeaf(3.0),
                                    LLeaf(4.0))),
                    LBranch("DIV",
                            LLeaf(3.0),
                            LLeaf(2.0)))

val test2 = compute(test)
```

Exercise 6:

Using your ADT `LTree`, define a function `transform` that takes as input a `Tree[Int]` and a function `f: (Int,Int)=>Int`, and returns a `LTree[Int]` in which the values of each `LBranch` are computed using the function and the labels of each of its trees. You will need to rely on a function to retrieve the label. For reasons beyond this course, you will need to choose a function name that is different from the names of the labels in your constructors: e.g., `value`.

Solution 7:

```
enum LTree[+A]:
    case LLeaf(label: A)
    case LBranch(label: A, left: LTree[A], right: LTree[A])

    def value: A = this match
        case LLeaf(l) => l
        case LBranch(l, _, _) => l

object LTree:
    import Tree._
    def transform(t: Tree[Int], f: (Int,Int) => Int): LTree[Int] =
        t match
            case Leaf(a) => LLeaf(a)
            case Branch(left, right) =>
                val l = transform(left, f)
                val r = transform(right, f)
                LBranch(f(l.value, r.value), l, r)
```

```
scala>import LTree._
scala>import Tree._
scala>transform(Branch(Branch(Leaf(3),Leaf(5)),Leaf(4)), _ + _)
val res0: LTree[Int] = LBranch(12,LBranch(8,LLeaf(3),LLeaf(5)),LLeaf(4))
```

References

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