Assignment 4 – Pattern matching & lists

Advanced programming paradigms

In this assignment, you will work with pattern matching in the context of expressions evaluations and lists. In the last two exercises, you will implement several handy functions to work with lists. Even though these functions are already present in the Scala `List` library, implementing them is a good practice to understand how they work.

Question 1 – *Working with expressions*

Here is the code of the expression interpreter shown in class:

```scala
sealed abstract class Expr
case class Number(n: Int) extends Expr
case class Sum(e1: Expr, e2: Expr) extends Expr

def eval(e: Expr): Int = e match {
  case Number(n) => n
  case Sum(e1, e2) => eval(e1) + eval(e2)
}
```

(a) Add to this code the `show` method in order to display an expression.

(b) Add to this expression interpreter the product `x*y` operation. *Hint:* pattern matching with constructor pattern also works recursively.

(c) In addition, change your show function so that it also deals with products.

⚠ Pay attention you get operator precedence right and that you use as few parentheses as possible when showing your expression.

(d) When you are done, check your code with the following snippet:

```scala
val expr0 = Sum(Product(Number(2), Number(3)), Number(4))
println("Expr0: " + show(expr0))
assert(eval(expr0) == 10)

val expr1 = Product(Number(4), Number(12))
println("Expr1: " + show(expr1))
assert(eval(expr1) == 48)

val expr2 = Product(Sum(Number(2), Number(3)), Number(4))
println("Expr2: " + show(expr2))
assert(eval(expr2) == 20)

val expr3 = Product(Number(2), Sum(Number(3), Number(4)))
println("Expr3: " + show(expr3))
assert(eval(expr3) == 14)
```

It should produce no assertion error and display the following result (pay attention to the parentheses for the product in `Expr2` and `Expr3`):

```
Expr0: 2*3+4
Expr1: 4*12
Expr2: (2+3)*4
Expr3: 2*(3+4)
```

Question 2 – *Defining trees*

We now propose a similar approach for modelling trees, as follows:

```scala
sealed abstract class BinaryTree
case class Leaf(value: Int) extends BinaryTree
case class Node(left: BinaryTree, right: BinaryTree) extends BinaryTree
```

Starting with this code,

(a) write a function to compute the sum of the **leaves** of the tree.

(b) write a function to find the smallest element of the tree. Please note that we are not using a *binary-sorted tree* here, which means you have to check every node of the tree.

Question 3 – *Lists functions*

(a) In order to apply pattern matching and exercise the way of thinking with lists, you have to implement several functions on lists. Please note that your functions should work on lists of arbitrary types, use only pattern matching and no builtin `List` functions.

- `last`, which returns the last element of a list;
- `init`, which returns a list of every element but the last;
- `reverse`, which returns the list with its elements in the reversed order;
- `concat`, which concatenates two lists together;
- `take(n)`, which returns the first *n* elements of the list;
- `drop(n)`, which returns the list without its first *n* elements. If $n > length(list)$, then `Nil` should be returned;
- `apply(n)`, a function that returns the *nth* element of a list. Note that the first element of a list is at position 0.

(b) What is the complexity of

1) `last`

1) ____________________

2) `concat`

2) ____________________

3) `reverse`

3) ____________________

Question 4 – *Predicates on lists*

(a) Define the function `any` which should have the following prototype[1]:

```scala
def any[T](p: T => Boolean)(l: List[T]): Boolean
```

This function should return `true` if any element of the list satisfies the predicate.

(b) Define then the function `every` which controls if *every* element of a list satisfy this predictate. Use pattern matching.

[1] This exercise is taken from the *Programmation 4* course given by M. Odersky at EPFL