# Exercises Sheet 2 (Lectures 4, 5 and 6)

## Programming Scalable Systems

Universidad Politécnica de Madrid

2022-2023

### Voluntary Submissions

We expect you to practice the exercises and ask questions during the lectures. The submission is voluntary and we will do our best to review and give feedback during the following lectures.

Submission will be available in `https://deliverit.fi.upm.es`. You will have to submit all your code in one module *Sheet2* in a file `sheet2.ex`:

```
defmodule Sheet2 do
  # Your code goes here
  ...
end
```

**Recommendation:** create a Mix project (`mix new sheet2`), add your functions to module *Sheet2* (file `lib/sheet2.ex`), and change the tests `test/sheet2_test.exs` with the tests we are sharing with you.

### Recursion and list processing (do after Lecture 4)

**Exercise 1** (Fibonacci)**.** Define a recursive function `fib/1`[1] such that `fib(n)` returns the $n$th element of the Fibonacci sequence. You may consider alternative idioms: pattern matching, guards, conditionals, etc.

**Exercise 2** (Euclid's algorithm)**.** Define a function `gcd/2` such that `gcd(n,m)` returns the *greatest common divisor* of $n$ and $m$ using Euclid's algorithm.

**Exercise 3** (Fibonacci revisited)**.** Without calling the function defined in Ex. 1, define a recursive function `fibs/1` such that `fibs(n)` returns a list containing the elements of the Fibonacci sequence upto the $n$th element. Use it to define an alternative solution to Ex. 1 that runs in linear time.

**Exercise 4** (List reversal)**.** Define a recursive function `reverse/1` to reverse a list – do not call *Enum*`.reverse/1`.

**Exercise 5** (Reversal, revisited)**.** Without calling the function defined in the previous exercise nor *Enum*`.reverse`, define a recursive function `revonto/2` such that `revonto(xs,ys)== reverse(xs)++ys`. Use it to provide an alternative implementation for list reversal.

**Exercise 6** (Combinatory numbers)**.** Define a function `pascal/1` such that `pascal(n)` returns the $n$th row of Pascal's triangle. Use it to define a function `comb/2` such that `comb(n,m)` $= \binom{n}{m}$.

[1] We use $f/n$ to indicate that function $f$ has $n$ arguments.

**Exercise 7** (Merge Sort)**.** Define a function `mergesort/1` that sorts a list using the *MergeSort* algorithm.

**Exercise 8** (Permutations)**.** Define a function `permuts/1` that returns a list containing all the permutations of a given list. Different recursive approaches are possible.

**Exercise 9** (Vectors as lists)**.** Let us represent vectors in $R^n$ as lists of floating point numbers. Define functions `vadd/2`, `scale/2` and `dotprod/2` to implement *vector addition*, *multiplication by an scalar* (scalar being first parameter) and *scalar product*, respectively.

**Exercise 10** (Matrices)**.** Let us represent $n \times m$ matrices as lists made of $m$ lists (rows) each of length $n$. Define a function `dim/1` that, given a list of lists, returns its dimensions as a pair of integers when it actually represents a valid matrix.

**Exercise 11** (Matrix sum)**.** Define a function `matrixsum/2` that adds two matrices of the same dimensions.

**Exercise 12** (Transposition)**.** Define a function `transpose/1` that, given an $n \times m$ matrix returns its transpose – which will have dimensions $m \times n$.

**Exercise 13** (Matrix product)**.** Define a function `matrixprod/2` that multiplies two matrices when they have the right dimensions.

**Exercise 14** (Erathostenes)**.** Define a function `primes_upto/1` such that `primes_upto(n)` returns all the prime numbers upto $n$. Use the *Erathosthenes' sieve* method.

**Exercise 15** (Factorization)**.** Using the previous exercise, define a function `factorize/1` such that `factorize(n)` returns a list with all the prime factors of $n$. The list may include repetitions.

## Recursive data structures (do after Lecture 5)

**Exercise 16** (Binary insertion trees)**.** A binary insertion tree may be either empty (a *leaf*, a *tip*) or nonempty, and in this case it contains a (central) *value* and two subtrees, usually referred to as the *left* and *right* subtrees.

A binary insertion tree is ordered when the values stored in the left subtree are no greater than the central value and the values stored in the right subtree are no smaller than the central value.

If empty trees are represented by the atom

```
:tip
```

and nonempty trees by tuples

```
{:node, left, value, right},
```

implement a function `tree_insert/2` (tree as first parameter) that inserts an element into an ordered binary tree preserving its order.

**Exercise 17** (Tree sort)**.** Define a function `inorder/1` that returns a list with all the values stored in a binary insertion tree, in order. Combine it with the previous exercise to define a function `treesort/1` that sorts a list by inserting its elements in a tree.

### Review and Études (do after Lecture 5)

**Exercise 18** (Review)**.** Review material of lectures 3 and 4, and have a look at [Laurent and Eisenberg, 2014, chapters 2 to 8 and 13]

**Exercise 19** (Études)**.** Practice *Études* in [Eisenberg, 2013]: *Études* 4, 5, 6, 7, 8, and 13.

### Higher-order list operations and their relatives (do after Lecture 6)

**Exercise 20** (∀)**.** Give your own implementation of the `all/2` function defined in the *Enum* module. Use it to simplify your solution to exercise 10.

**Exercise 21** (4')**.** Revisit your solution to exercise 4 using *List*`.foldr`.
Name the function `reverse_fold/1`.

**Exercise 22** (5')**.** Revisit your solution to exercise 5 using *List*`.foldr`.
Name the function `revonto_fold/2`.

**Exercise 23** (zip_with)**.** Using *Enum*`.map` and *Enum*`.zip` define a function `zip_with/3` that, given a binary function and two lists combines them by applying the function to corresponding elements of both lists: `zip_with(f,[x1..xn],[y1..yn])== [f(x1,y1)..f(xn,yn)]`.

**Exercise 24** (6')**.** Revisit your solution to exercise 6 using `zip_with`.
Name the function `pascal_zip/1`.

**Exercise 25** (9')**.** Revisit your solution to exercise 9 using higher-order list operations.
Name the functions `vadd_ho/2`, `scale_ho/2`, and `dotprod_ho/2`.

**Exercise 26** (11')**.** Revisit your solution to exercise 11 using higher-order list operations.
Name the function `matrixsum_ho/2`

**Exercise 27** (12')**.** Revisit your solution to exercise 12 using higher-order list operations.
Name the function `transpose_ho/1`

**Exercise 28** (13')**.** Revisit your solution to exercise 13 using higher-order list operations.
Name the function `matrixsum_ho/2`

**Exercise 29** (Generic sorting)**.** Modify your solution to exercise 7 so that the sorting criterion comes as a binary predicate, i.e. `mergesort/2` will receive a list and a boolean function on pairs.
Name the function `mergsort/2`

**Exercise 30** (Maps on trees)**.** Define a function `map_tree/2` that applies the same function to all the values in a binary insertion tree (being the tree the first argument).

**Exercise 31** (Filter on trees)**.** Define a function `filter_tree/2` that removes the values that do *not* have a given property from a binary insertion tree (being the tree the first argument).

## References


[Eisenberg, 2013] Eisenberg, J. D. (2013). *Études for Elixir*. O'Reilly Media. **exercises book**.

[Laurent and Eisenberg, 2014] Laurent, S. S. and Eisenberg, J. D. (2014). *Introducing Elixir*. O'Reilly. **class book**.