CS-205 Declarative Programming — October 17th 2023
Cécilia Pradic, Monika Seisenberger

Lab 3: pattern-matching and recursion

Dear all,

- You can do your lab by editing a single file.
- If you want to sign off, please recall to load your files in `ghci` to check for errors before calling a demonstator over. Additionally, it is also a good idea to test the functions you defined in `ghci` on a variety of examples to double-check they have the behaviour you are expecting.
- This week the Haskell runtime on the lab machines work! But no nice VSCode support is set up on the lab machines (and it probably won't be done this term).

Happy hacking!

Task 3.1: **(conditionals)** For this task, you can use conditionals or guarded equations, whichever you prefer.

(a) Write a function

```
orderPair :: Ord a => (a, a) -> (a, a)
```

that takes as input a pair and outputs another pair containing the same elements, but ordered. For instance `orderPair (0,1)` should evaluate to `(0,1)` while `orderPair (15,7)` should compute `(7,15)`

(b) Get acquainted with the function

```
elem :: Eq a => a -> [a] -> Bool
```

from Haskell's `base` library; it allows to check if an element is in a list or not. Using this, implement a function

```
addToSet :: Eq a => a -> [a] -> [a]
```

such that `addToSet x xs` represents the list `xs` extended with `x` if `x` was not an element of `xs`, or simply `xs` otherwise[1].

Task 3.2: **(pattern-matching)** Please use pattern-matching and no conditionals for this task.

(a) Write a function

```
majority :: Bool -> Bool -> Bool -> Bool
```

such that `majority a b c` is `True` if and only if at least two out of three of the arguments `a`, `b` and `c` are `True`.

(b) Write a function

```
removeFstZero :: String -> String
```

[1] The intent here is that this operation ensures that if we start appying it to a list that has no duplicates, the resulting list has no duplicates as well; this would be a good helper function to encode a naive datastructure for unordered collections.

which removes the first character of its input if it happens to be a `'0'`. (Recall that `String` is the same as `[Char]` in Haskell)

Task 3.3 (recursion) All of the solutions of this task should involve a recursive function definition.

(a) Write a function

```
pow :: Double -> Int -> Double
```

corresponding to the usual power operator $(x, y) \mapsto x^y$ using recursion (and without using the built-in `(^)` operator).

(b) Define a function

```
deleteAll :: Eq a => a -> [a] -> [a]
```

such that `deleteAll x xs` is the list `xs` where all occurences of `x` have been deleted. For instance, we should have `deleteAll 5 [1,5,2,3,5,2] == [1,2,3,2]`.

(c) Write a function

```
sumPrefixesPlusConst :: Int -> [Int] -> [Int]
```

such that `sumPrefixesPlusConst k xs` contains the sums of all the prefixes[2] of `xs` with `k` added on top. To be more precise, the output list `sumPrefixesPlusConst k xs` should have the same length as `xs`, and if `xs` corresponds to the list $[x_1, x_2, ..., x_n, ...]$, the nth element element of the output should be $k + x_1 + x_2 + \ldots + x_n$. For instance, we will have `sumPrefixesPlusConst 2 [1,10,2] == [3,13,15]`.

Then use it to define the function

```
sumPrefixes :: [Int] -> [Int]
```

that maps `xs` to the sums of all its prefixes.

(Note that defining `sumPrefixes` directly by recursion would be challenging! This is one of the cases where defining an auxiliary function is useful.)

[2] A prefix of a list `xs` is sublist containing the same elements in order, and such that if an element of index i of `xs` is included, so are all the elements of lesser index $j < i$.

Task 3.4 Challenge task: In this exercise, we will see how to use recursion to code the mergesort algorithm, one of the most efficient way of sorting lists. You should try to do this exercise without using *any* of the functions of the `base` library except `(<=)` in your code.

(a) Write a function

```
split :: [a] -> ([a],[a])
```

which takes as input a list and splits it into two equal parts. Try to make it so your program needs to traverse the input list only once (applying `length` counts as traversing the list). Hint: if that is useful, you can use a sort of pattern-matching syntax to decompose a tuple you obtain through a recursive call as in the following bit of code (which does not do much that is interesting)

```
iteratedswap :: Int -> (a,a) -> (a,a)
iteratedswap 0 p             = p
iteratedswap n p | n < 0     = iteratedswap (-n) p
                 | otherwise = (y,x)
                     where (x,y) = iteratedswap (n-1) p
```

(b) Write a function

```
merge :: Ord a => [a] -> [a] -> [a]
```

which takes as input two sorted lists and outputs a merging of the two lists which remains sorted (in particular, you can check that your function is correct on examples by checking that `merge (sort xs) (sort ys) = sort (xs ++ ys)` for arbitrary `xs` and `ys`).

(c) Using the two functions above, write a function

```
mergeSort :: Ord a => [a] -> [a]
```

which takes as input a list and returns the same list sorted.