# Exercise sheet for lecture 02

In the Git repository (https://gitlab2.informatik.uni-wuerzburg.de/intro-to-fp/tasksheets) you can find templates for the exercise sheets. These include already discussed code from the lecture (e.g. the List type), as well as function signatures given in the sheet. The repo also contains a `build.sbt` file with preconfigured compile options. Using that file, you can import the repository as a project into your favorite Editor (with LSP support) or IDE.

In the lecture, we looked at the functional data structure `List`. You can find the implementation in the file `fp02/List.scala` in the template repo.

## 1 dropWhile

In the lecture we saw a function named `drop`, which removes the first $n$ elements of a list.

Implement the function `dropWhile`, which removes elements from the beginning of the list, as long as the given boolean function returns `true` for them. Use pattern matching and recursion.

```
def dropWhile(f: A => Boolean): List[A] = ???
```

## 2 Folds — step by step

In the lecture the execution of a call to `foldRight` was traced step by step.

*a*) What does this *trace* look like, if we use `foldLeft` instead?

```
Cons(1, Cons(2, Cons(3, Nil))).foldLeft(0)((x, y) => x + y)
```

*b*) The following call to `foldRight` increments every element in the list `l` by 1 and creates a new list from that. Trace the execution step by step.

```
val l = Cons(1, Cons(2, Nil))
l.foldRight(Nil: List[Int])((h, t) => Cons(h + 1, t))
```

## 3 reverse

Implement the function `reverse`, which returns the list in reverse order. Use `foldLeft`. Think about which initial *accumulator* has to be given to `foldLeft` to build a list from.

```
def reverse: List[A] = ???
```

# 4 append

Implement the function `append`, which concatenates this list with another one, i.e. appends the given list to this one, using `foldRight`.

```
def append[AA >: A](r: List[AA]): List[AA] = ???
```

You can find an Implementation using pattern matching and manual recursion in the template.

# 5 map, filter and flatMap

## map

Implement the function `map` using `foldRight`. This function modifies every element in the list while keeping the structure of the list. In the lecture, two example applications for this function were shown.

```
def map[B](f: A => B): List[B] = ???

//Example 1: double all values
List(1,2,3).map(_ * 2) == List(2,4,6)

//Example 2: result type can be different
List(1,2,3).map(_.toString) == List("1", "2", "3")
```

*Hint:* You've seen a special case of the implementation of `map` in an earlier exercise on this sheet.

## filter

Implement the function `filter` as seen in the lecture. This function returns a new list, which only contains those elements, for which the given boolean function returns true.

```
def filter(p: A => Boolean): List[A] = ???

//Example: only keep odd numbers
List(1,2,3,4,5,6).filter(_ % 2 == 1) == List(1,3,5)
```

## flatMap

The function `flatMap` is similar to `map`, but the function passed into it returns a list of elements. The returned lists are concatenated into a single list.

```
def flatMap[B](f: A => List[B]): List[B] = ???

//Example: for each number, add it multiplied by 10 to the list
List(1,2,3).flatMap(i => List(i, i * 10)) == List(1, 10, 2, 20, 3, 30)
```

Use `foldRight` and `append` to implement `flatMap`.

## 6 zip

The functions in the `zip` family combine multiple lists in various ways.

*a*) Implement a function `zipAdd`, which takes to lists of integers and creates a new list by adding corresponding elements in the lists. For example `zipAdd(List(1,2,3), List(4,5,6))` results in the new list `List(5,7,9)`

   Implement this function in the companion object!

*b*) Generalize the function `zipAdd` to `zipWith`, so that it isn't limited to integer lists and adding, but takes the operation as a parameter.

   Implement this function directly in the `List` enum. (Beware: not simple!)

Think about the required signatures for each function and write them yourself.

If the two lists aren't the same length, the zip functions should stop at the end of the shorter one, so that the resulting list also has the length of the shorter input list.

*Hint:* Use pattern matching and recursion. For pattern matching multiple elemns, you can combine them into a tuple `(a,b)` and match on that. For example, using integers instead of lists:

```
val a = 2
val b = 4
(a,b) match
  case (x,y) => /* x == 2, y == 4 */
```

## 7 Parametricity

*Implement the following functions in the file* `Parametricity.scala`

Implement the functions `curry uncurry`, which convert functions taking two parameters into nested functions taking one parameter (and the other way round)

```
def curry[A,B,C](f: (A,B) => C): A => (B => C) = ???

def uncurry[A,B,C](f: A => (B => C)): (A, B) => C = ???
```

*Hint:* Because of the generic definition and choice of type parameters, there is basically only one way to implement these.