# Homework 2

Functional Programming (ITI0212)

due: 2021.03.26

Place your solutions in a module named `Homework2` in a file with path `homework/Homework2.idr` within your `iti0212-2021` repository on the TalTech GitLab server (`https://gitlab.cs.ttu.ee/`). Your solutions will be pulled automatically for marking. At the start of the file include a comment containing your name and the Idris version you are using. Precede each problem's solution with a comment specifying the problem number.

**Problem 1**
Write functions with each of the following types:

```
joinIO :  IO (IO a) -> IO a

mapIO  :  (a -> b) -> IO a -> IO b

(>=>)  :  (a -> IO b) -> (b -> IO c) -> a -> IO c
```

Do this without using standard library functions that we haven't yet discussed in this course. Specifically, your definitions should be written in terms of the primitive `IO` functions that we learned about, `pure : a -> IO a` and `(>>=) : IO a -> (a -> IO b) -> IO b`, or the `do`-notation syntactic sugar, if you prefer.

**Problem 2**
Write a function that takes either a computation that when run produces a result of type `a` or a computation that when run produces a result of type `b`, and returns a computation that when run, runs whichever computation was given and produces the corresponding result:

```
eitherIO  :  Either (IO a) (IO b) -> IO (Either a b)
```

Now write a function that takes both a computation that when run produces a result of type `a` and a computation that when run produces a result of type `b`, and returns a computation that when run, runs the two computations in order and produces the pair of their results:

```
bothIO  :  Pair (IO a) (IO b) -> IO (Pair a b)
```

**Problem 3**
Write `Num`, `Eq` and `Ord` instances for the type of coinductive natural numbers, `CoNat`, introduced in lecture 7. All of your methods of the `Num` interface (`(+)`, `(*)`, and `fromInteger`) should be total. Your `(==)` and `compare` methods should agree with their `Nat`-counterparts on all finite `CoNats`, and as a result will necessarily be partial. For example:

```
> 1 + 1 == the CoNat 2
True
> 2 * 3 == the CoNat 6
True
> 42 == infinity
False
> 42 <= infinity
True
```

**Problem 4**
Recall that `List`s are necessarily finite sequences of data, `Stream`s are necessarily infinite sequences of data, and `CoList`s, introduced in lecture 7, are sequences of data that may be either finite or infinite. It should always be safe to convert a `List` or `Stream` to a `CoList` that contains the same elements in the same order. Write `Cast` instances to convert both `List`s and `Stream`s to `CoList`s:

```
implementation  Cast (List a) (CoList a)  where

implementation  Cast (Stream a) (CoList a)  where
```

Both `cast` methods should be recognized by Idris as total.

**Problem 5**
A *queue* is a data structure with two "ends". An *empty queue* contains no data. Given any queue we can *push* an element onto its back end, and if it is not empty, we can also *pop* an element off of its front end in a first-in–first-out manner.

```
interface  Queue (queue : Type -> Type)  where
  empty  :  queue a
  push  :  a -> queue a -> queue a
  pop  :  queue a -> Maybe (Pair a (queue a))
```

A list can be used as a simple form of queue. Write a `Queue` implementation for `List`s in Idris:

```
implementation  Queue List  where
```

**Problem 6**
Unfortunately, implementing a queue as a `List` is unavoidably inefficient. In order to perform one of the two operations (push or pop), we must traverse the entire list. This causes that operation to take time proportional to the size of the queue. Fortunately, we can do much better. There is an old trick of implementing a queue using two `List`s rather than one. We use one list as the "back" of the queue, where we can push elements onto it in constant time using cons. We use the other list as the "front" of the queue, but oriented the other way. As long as the front list is not empty we can pop elements off of it in constant time by pattern matching against its head and tail. The only tricky part is what happens when the front list is empty but the back list is not. In this case we replace the empty front list with the reversal of the back list, and replace the back list with the empty list. It can be shown that using this technique we can both push and pop elements in amortized (that is, on average) constant time. Write a Queue implementation for the following data type of list pairs, using the strategy just described.

```
data  ListPair : Type -> Type  where
  LP  :  (back : List a) -> (front : List a) -> ListPair a
```