# Lab 7

Functional Programming (ITI0212)

2022-03-11

This week we learned how to do monadic I/O in typed *purely functional* programming languages. Unlike *imperative* programming languages these do not have a syntactic class of *statements*. Instead, some *expressions* represent *computations*, which are instructions to the run-time system to perform various *actions*. These are distinguished in the type system by being elements of `IO` types.

We can build up compound computations from simpler ones using two *monadic combinators*, `pure : a -> IO a`, which produces a trivial computation, and `(>>=): IO a -> (a -> IO b)-> IO b`, which sequences computations by running the first and passing the resulting value to the next.

There is syntactic sugar called `do`-*notation*, in which a sequence of computations can be written to resemble a block of statements in an imperative programming language. This can be convenient, but it is important to understand that it is merely a syntactic transformation: purely functional programming languages do not have statements.

**Task 1**
Write a computation that doesn't give up until it gets a number from the user.

```
get_number  :  IO Integer
```

For example:

```
Lab7> :exec get_number >>= printLn
Please enter a number: forty two
I'm sorry, I didn't understand that.
Please enter a number: You know, the answer to life, the
   universe and everything.
I'm sorry, I didn't understand that.
Please enter a number: 42
42
```

**Task 2**
Desugar the following computation to use the sequencing operator `(>>=)` rather than `do`-notation:

```
add_pair  : IO Integer
add_pair  =  do
  putStr "Please enter the first number: "
  x <- get_number
  putStr "Please enter the second number: "
  y <- get_number
  pure (x + y)
```

**Task 3**
Write a computation that gets a list of numbers from the user, like `get_numbers` from the lecture, and returns their sum, or zero if the list is empty.

```
add_numbers  :  IO Integer
```

*Hint:* you can write this as a point-free one-liner using `get_numbers` together with computation sequencing, function composition, and a fold.

**Task 4**
Write a checked version of `get_numbers`, which prompts the user to re-enter their input if it is unable to parse an integer from it.

```
get_numbers_checked  :  IO (List Integer)
```

For example:

```
Lab7> :exec get_numbers_checked >>= printLn
Please enter a number or 'done': 1
Please enter a number or 'done': two
I'm sorry, I didn't understand that.
Please enter a number or 'done': 2
Please enter a number or 'done': 3
Please enter a number or 'done': done
[1, 2, 3]
```

**Task 5**
Write a function that takes a string transformer function and paths to a source and target file, which reads the contents of the source file, transforms it by the function, and writes the result to the target file.

```
transform_file  :  (transform : String -> String) ->
  (src_path : String) -> (tgt_path : String) ->
  IO (Either FileError Unit)
```

*Note:* You can write text to a file using the `writeFile` and `appendFile` functions. To avoid accidentally deleting important data you should probably open the target file in `Append` mode. You may also want to browse the module `System.File.ReadWrite`, which is re-exported by `System.File`. You can use your `titlecase` string transformer from homework 1 to test your function.