Lab 7: More monads - Purely functional IO, State etc.

Functional Programming (ITI0212)

This week we looked more at monads. `Monad` is a typeclass whose instances allow us to write programs with "effects" such as IO, non-determinism, exceptions, etc.

Unlike imperative languages, Lean 4 does not have a syntactic class of statements, but do-notation allows us to write imperative style code.

Task 1
Write an expression `main : IO Unit`, which reads a line of user input and prints the number of words.

To test your function, navigate to the directory of your lab file and run `lean --run Lab7.lean`

```
Enter a sentence: Hello world!
Sentence contains 2 words.
```

Task 2
Let `t` be a `Monad`. Write a generic function which takes two functions $f : \alpha \to$ `t` β and $g : \beta \to$ `t` γ and returns a function $\alpha \to$ `t` γ. There is essentially one way to do this, using the `bind` (`>>=`) of `t`.

This operation is known as *Kleisli composition*: it is a canonical way of composing functions whose outputs are in some `Monad`. Lean provides the infix notation `>=>` for Kleisli composition.

Notice the resemblance between your answer and the definition of function composition, `Function.comp`.

Bonus: given a function `>=>`, can you define a function `>>=`?

Task 3
For any `Monad t`, there is an essentially unique way to define a function

`t (t` α`)` $\to$ `t` α.

Write a generic implementation of this function, parameterized by `[Monad t]`, and call it `join`. What does `join` do when `t` is the `List` monad? Recall that

```
instance : Monad List where
  pure x := [x]
  bind xs f := List.flatten (Functor.map f xs)
```

(this instance not provided by Lean).

Hint: you will need to make use of the generic identity function `id` : $\beta \to \beta$.

Fact: conversely, given a `Functor` equipped with a function `t (t` α`)` $\to$ `t` α, it is possible to define `>>=`.

Task 4
For this task, you will need to make sure you have copied the instance of `Monad List` from Task 3 into your file.

Write a function `binaryLists (n : Nat) : List (List Nat)` that for a given `n` produces a list of all the lists of length `n` whose entries are either `0` or `1`.

You should write this in monadic style, with do-notation, i.e. use the fact that `List` is a `Monad`. You will need to use recursion, so think about how `binaryLists` of length `n + 1` can be built from `binaryLists` of length `n`.

For example, `binaryLists 0` evaluates to `[[]]`.

`binaryLists 2` evaluates to `[[0,0],[1,0],[0,1],[1,1]]` (or in some other order).

Hint: while you build intuition for the `List` monad, you may wish to first write the function using `List.flatten` and `Functor.map`, and then notice how to replace this with monad structure.

Task 5
Write a function,

```
addIO : IO (Option Int)
```

which takes asks the user to enter a number, twice. If the inputs are valid integers, return their sum wrapped in `some`. Otherwise, return `none`.

Note that the `getLine` function will include the return character, thus you will probably need to use `String.trim`. You can use `String.toInt? : String → Option Int` to convert strings to integers.

To test your function, define the following

```
def main : IO Unit := do
    let stdout <- IO.getStdout
    let add <- addIO
    stdout.putStrLn s!"{add}"
```

In a terminal, navigate to the directory of your lab file, and run `lean --run Lab7.lean`, which executes the `main` function.

Example:

```
Please enter number 1: 7
Please enter number 2: 10
some 17
Please enter number 1: 10
Please enter number 2: no
none
```