Lab 4: Functional literals and higher-order functions

Functional Programming (ITI0212)

This week we learned about function literals and higher-order functions. Function literals are free-standing expressions that represent values for the function types. We can refer to the function with formal parameter `x` and body `t` using the `fun` keyword `fun x => t` or the λ notation `λx => t`.

A *higher-order function* is a function that can take other functions as arguments and/or return them as results. We saw how the `map` and `filter` functions for `List` types allow us to perform tasks that would typically be done in imperative programming languages using loops, and how the `fold` function for an inductive type allows us to define other functions without using pattern matching or recursion.

Task 1
We have seen that functions in Lean are *curried*: technically, instead of functions with multiple arguments, we have functions of single arguments that return functions. This is useful because then we can *partially apply* functions.

A true "function of multiple arguments" can be represented as a function from an (iterated) `Prod` type, e.g.

```
def add' : Nat × Nat → Nat := fun (x,y) => x + y
```

In order to evaluate `add'`, we must provide both arguments (as a pair).

Write a (higher-order) function

```
curry (f : α × β → γ) : α → β → γ
```

which transforms a function of two arguments into its curried form, and a function

`uncurry` $(f : \alpha \to \beta \to \gamma) : \alpha \times \beta \to \gamma$

which transforms a curried function into a function of two arguments.

Task 2
Using `List.foldr`, write a function

`andl : List Bool → Bool`

which takes a list and returns the logical conjunction (`Bool.and`) of its members. The conjunction of an empty list is `true`.

For example,

```
#eval andl [true, true, false]
=> false
#eval andl [true, true, true]
=> true
```

Task 3
Using `List.foldr`, write a function

```
mull : List Nat → Nat
```

which takes a list of natural numbers and returns their product. You may assume that `mull []` evaluates to `1`.

Task 4
Write a function that returns numbers in a list that are multiples of 10.

Hint: Use `Nat.mod` aka `%` to compute remainders upon division, and `==` to compare numbers for equality.

Task 5
Write a function which behaves also follows: it takes a string as an input and returns the string in which the lowercase vowels were removed. For example:

```
#eval ignore_lowerCaseVowels "the cat who saw the moon."
"th ct wh sw th mn."
#eval ignore_lowerCaseVowels "the cat who sAw the moon."
"th ct wh sAw th mn."
```

Hint: the functions `List.asString` and `String.toList` from the standard library will be helpful, to convert strings to and from lists of characters.

Task 6
Write the `fold` function for the `Bool` type, `fold_bool`.

- First determine the type of this function using the algorithm described in the lecture.
- Then write the function definition using the algorithm for that.

Up to argument order, you should recognize this function as a construct present in nearly every programming language, what is it?