Lab 2: Inductive types and recursive functions

Functional Programming (ITI0212)

This week we learned about inductive types and recursive functions. Inductive types are specified by (element) constructors. These constructors define the possible ways of creating elements of the given type, and each may take different numbers and types of arguments. In particular, the type itself may be an input to one of its constructors.

Recursive functions on inductive types use case analysis or pattern matching in order to specialize the function being defined for the possible element constructors. These functions may call themselves using recursive calls to compute the result for the current case using the results for other cases.

Task 1
An important function in digital circuit design is the `xor` function, which takes two `Bool` inputs and returns `true` just in case they differ. Implement this function in Lean by pattern matching on the input(s). Try implementing it with only three clauses.

Either write this function in a `namespace Lab2`, or call it `xor'` to avoid a name collision with Lean's standard library.

Task 2
The two-element type `Bool` is used to represent the truth or falsity of a proposition. But sometimes we are not so sure about things. Write a four-element type `Prob` with elements name `Definitely`, `Likely`, `Doubtful`, and `Impossible`.

Task 3
Define a conjunction function `and : Prob -> Prob -> Prob`

according to the following table

↓ and →	Definitely	Likely	Doubtful	Impossible
Definitely	Definitely	Likely	Doubtful	Impossible
Likely	Likely	Likely	Doubtful	Impossible
Doubtful	Doubtful	Doubtful	Doubtful	Impossible
Impossible	Impossible	Impossible	Impossible	Impossible

For example: `#eval and Definitely Likely` should evaluate to `Likely`.

Challenge: try to write this definition using as few clauses as possible (it should be possible with six).

Task 4
The *factorial* function $n!$ on the natural numbers can be characterized by the following recursive specification:

$$n! := \begin{cases} 1 & \text{if n} = 0 \\ n \times (n-1)! & \text{otherwise.} \end{cases}$$

Turn this recursive mathematical specification into a recursive function definition in Lean.

Task 5
Define a multiplication function for the natural numbers by recursion over the inductive structure of the type `Nat` (without using the `*` function from the standard library),

```
mul (n m :  Nat) :  Nat
```

Hint: try using recursion on the first argument.

Task 6
Let's consider a type of arithmetic expressions, that we might use to implement a calculator or simple programming language. An arithmetic expression is either a natural number literal, a sum of a left and right expression, or a multiplication of a left and a right expression.

This kind of definition is sometimes given in "Backus-Naur form" (BNF),

```
<AExp> ::= <nat> | <AExp> + <AExp> | <AExp> * <AExp>
```

a) Encode arithmetic expressions as inductive type `AExp`.

Elements of such types are sometimes called *abstract syntax* (representing in a structured way the *concrete syntax* specified by BNF, e.g. *"(1+2)*3"*).

b) Write a recursive function `eval_math : AExpr → Nat` that evaluates an `AExpr`.

For example, `eval_math (.Mul (.Sum (.Num 1)(.Num 2))(.Num 3))` should evaluate to `9` $= ((1 + 2) * 3)$.

You have just written an interpreter for this simple language of arithmetic expressions!

The syntax of `Lean` is given by an inductive type `Expr`. You can inspect this type by importing the Lean module

```
import Lean
```

at the top of your file, then write

```
#check Lean.Expr
```

then right click on `Lean.Expr` and choose `Go to definition...`

It's a bit more complicated than `AExpr`, and you won't be expected to understand everything here, but it's fun to take a look under the hood `:-)`

Task 7
Inductive types with one constructor are *structures*, and Lean provides special syntax for defining, instantiating and manipulating them. Here is a structure representing a point in the plane by its x and y coordinates,

```
structure Point where
  (x y : Float)
```

Recall that we can access the fields of a record using dot-syntax, e.g. `myPoint.x`, and this also works for records that are fields of other records.

Define a `structure` called `Segment` that represents a line segment by its endpoints, and then define a function `length : Segment → Float` that calculates the length of a `Segment`.

Hint: `Float.sqrt` takes square roots, and `^` provides exponentiation.