Lab 1: Installing and interacting with Lean

Functional Programming (ITI0212)

Task 1 (install Lean 4)
If you haven't already done so, install Lean 4 on your computer. To be able to follow the course, you must have a working Lean installation. You can also use the lab computers, but you should boot the computer into Ubuntu.

The recommended method for all platforms is to first install `vscode` (already installed on Lab computers) and then follow the instructions at `https://docs.lean-lang.org/lean4/doc/quickstart.html`.

When prompted to create a new project, select `Create a new standalone project`, and choose a directory called `labs` (you should use this project for all the labs, as some labs may depend on earlier ones), then open the project when prompted.

Alternatively, you can install Lean 4 manually and also a package for your editor such as `lean.nvim` or the Lean 4 mode for `emacs`.

The default project structure contains three `.lean` files:

- `Main.lean`
- `[project-name].lean`
- `[project-name]/Basic.lean`

`.lean` files define *modules*: the name of a module is the name of the file (without the extension). By convention, modules are named using `CamelCase`, e.g. `MyModule.lean`. In general, code should live in modules in the `[project-name]/` directory (for example, you should make a `labs/Lab_n.lean` file for each lab). For simple programs you can put code directly into `Main.lean`.

Task 2 (Building binaries)
We will mostly interact with Lean through the editor, but at some point we will want to build executable binaries. To do this in `vscode`, use the Lean menu (∀ symbol near the top-right), select `Project Actions... -> Project: Build Project`.

In a console, navigate to your project directory and locate the build directory: on unix systems this is inside the `.lake` hidden directory. Execute `bin/[project-name]`. You should see `Hello, world!`.

Task 3
Let's try interacting with Lean through the editor. In `Main.lean`, try hovering over terms such as `Unit`, `IO.println`, `hello` etc.

You should see the type of the term, e.g. `hello : String`, any provided documentation, and the module from which the term is imported. Make use of this feature to remind yourself what things mean and where they come from!

Task 4
Since Lean is implemented in Lean, it is possible to view the definition of any expression.

Try right-clicking `IO.println` for example, and select `Go to Definition`. The editor will jump to the module in Lean's core where this function is defined. For some expressions, such as `def`, you will need to first import the module `Lean` by adding `import Lean` at the start of your file.

This feature is helpful not only for quickly locating our own definitions, but also for looking under the hood of Lean's standard library or indeed any other library that we import.

Task 5
Lean also provides a number of commands, prefixed by `#`, which we can use to get information interactively but which do not themselves form part of the program (they are ignored by the compiler).

The `#check` command checks the type of an expression.

In a new module `labs/Lab1.lean`, add the line `#check 1`. In the `Lean infoview` panel, you should see

```
1 : Nat
```

telling us that `1` is a `Nat` (a natural number).

If you cannot see the `Lean infoview` panel, use the command palette (`ctrl+shift+p`) to open it using the `Lean 4:Infoview:Display Goal` command.

Task 6
The `#eval` command evaluates an expression.

Try something like `#eval 1 + 2` – you should see the result `3`.

These commands will be useful for quickly getting information about and testing programs, or even just fragments of code.

Task 7
Now let's write a simple program, to see some more interactive editing features.

Enter the following:

```
def average (x y : Nat) : Nat := ?goal
```

The expression `?goal` is a *hole*, a placeholder for a term. We can write any name after `?`, or simply `?_`. You can also use the expression `sorry`.

With your cursor on this line, in the `Lean infoview` panel, you should see

```
case goal
x y : Nat
⊢ Nat
```

This gives us information about the hole. The part `x y : Nat` before the turnstile (⊢) is called the *context*, it tells us what we have available to use at that point in the program, in this case two variables `x` and `y` of type `Nat`. The part after the turnstile, in this case `Nat`, is what we are trying to produce: we need to replace `?goal` with a term of type `Nat`, this is sometimes called the *goal*.

More generally, by placing your cursor at a point in a Lean program, the `Lean infoview` panel should display the context and goal at that point.

Task 8
Since we have `x y : Nat` ⊢ `Nat`, we can simply replace `?goal` with `x` or `y`, and we will have a well-typed program.

Try replacing `?goal` with `x`, for instance. Lean will underline `y` and the info view will note that `y` is not used. Unused variables may be replaced by an underscore.

Task 9
Obviously this function does not calculate the average of two natural numbers, so now write the expression that does this.

Test your function using `#eval`. For example,

`#eval average 2 4` should give `3` in the info panel.