Seminar on Computer-assisted mathematics

Judith Ludwig and Florent Schaffhauser

Heidelberg University - Summer semester 2025

Session 1 - April 17, 2025

In this seminar, we will learn how to **formalize** mathematics in the proof assistant **Lean**. Let us explain what that means.

Formalizing mathematics

Formalizing mathematics means expressing mathematical concepts, theorems and proofs in a formal language; according to the syntactic rules and grammar of the language. This can help understand the structure and steps of a proof.
The idea is not new, e.g.

- Leibniz had the idea of a universal formal language, the ***characteristica universalis***,
- Introduction of **ZFC** aiming to avoid Russel's paradoxes in set theory.

Today, **formalizing mathematics** means representing mathematical concepts, theorems, and proofs in a programming language, in a way that can be **verified by computers**. This is done using so called proof assistants.

Proof assistants

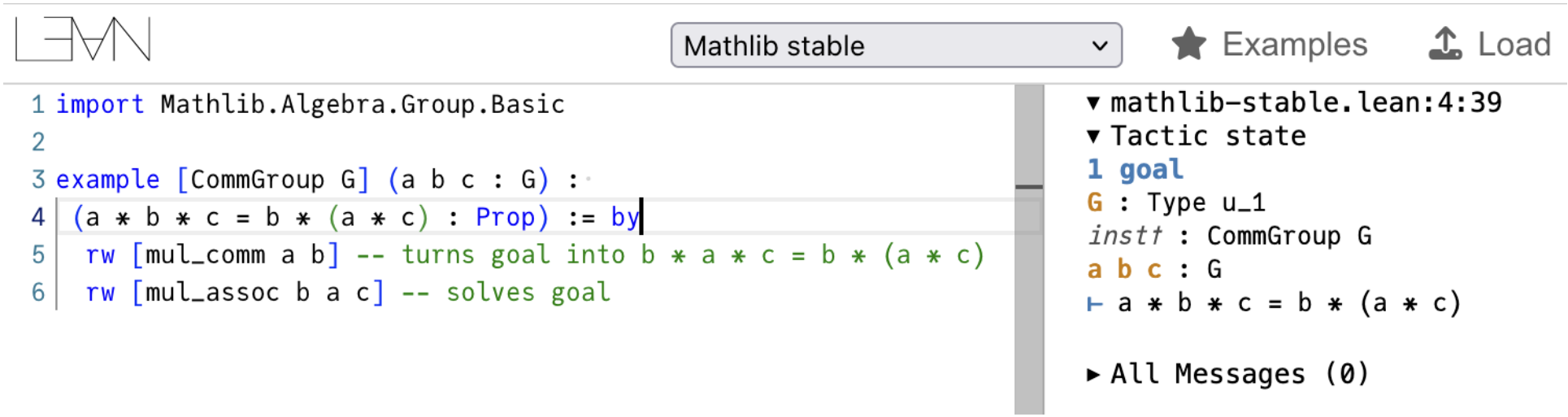
A **proof assistant** is a piece of software that assists with writing formal mathematics by human-machine collaboration.

- It **provides a computer language** for defining objects, specifying properties of these objects, and proving that these properties hold. This computer language is sometimes specifically developed.
- When the user enters a proof in the formal language, **the system checks two things:**
  1. That the **syntax** of the proof is correct with respect to the grammatical rules of the language.
  2. That the proof is indeed *a proof of the statement that it claims to prove*. This process is called **type-checking** and it is the key to formal verification: if the proof type-checks, the theorem is proved!
- The system also gives **feedback** to the user, in particular in the form of error messages. This is why proof assistants are also called **interactive theorem provers**.

Examples of proof assistants:

- Lean
- Coq, Agda
- HOL Light, Isabelle
- Mizar
- ...

Example of a proof assistant environment:



Examples of proof assistants:
Lean, Coq, Agda, HOL Light, Isabelle, Mizar, ...

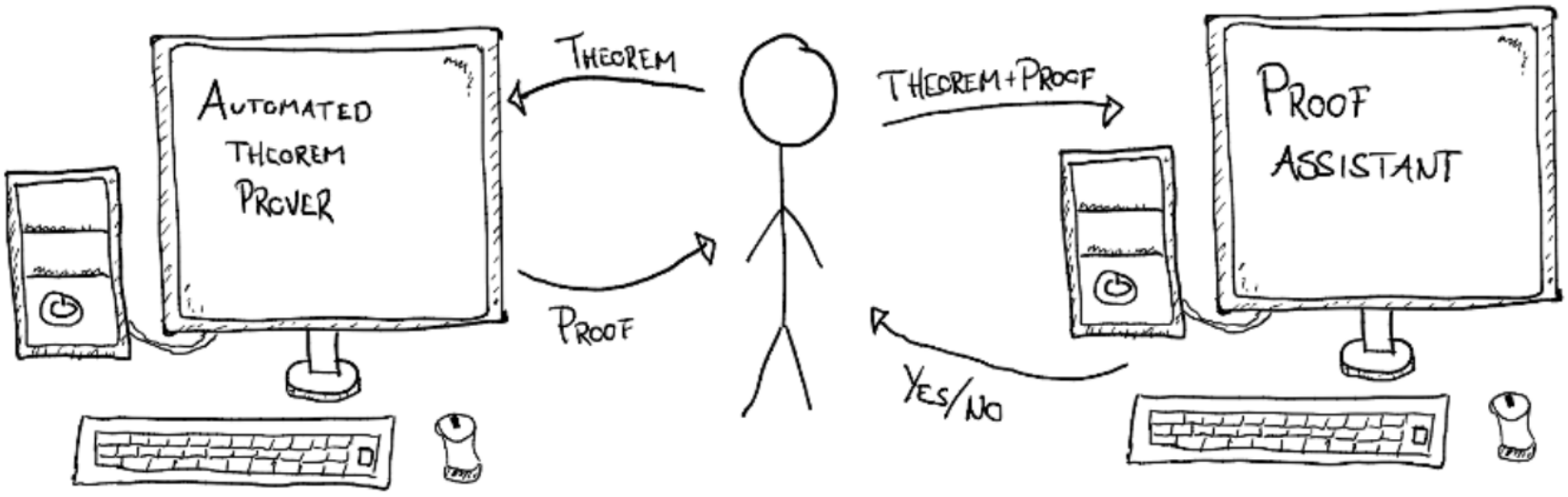
In this seminar, we will work with Lean.

Each proof assistant implements a particular **logic**.

For example:

- Mizar implements **set theory**.
- The logic that Lean implements is called **dependent type theory**. Same for Coq and Agda.
- HOL Light and Isabelle use what is called **simple type theory**.

What **proof assistants are not**:
They are not **calculators**. Asking proof assistants to *compute* $\sqrt{3}$ is not a sensible thing to do. The real number $\sqrt{3}$ is the positive real number that solves $x^2 - 3$, and it is precisely that number and any approximation of it is not $\sqrt{3}$.
They are not **automated theorem provers**.



But they might come with some automation tools.

Some motivation for formalizing mathematics.

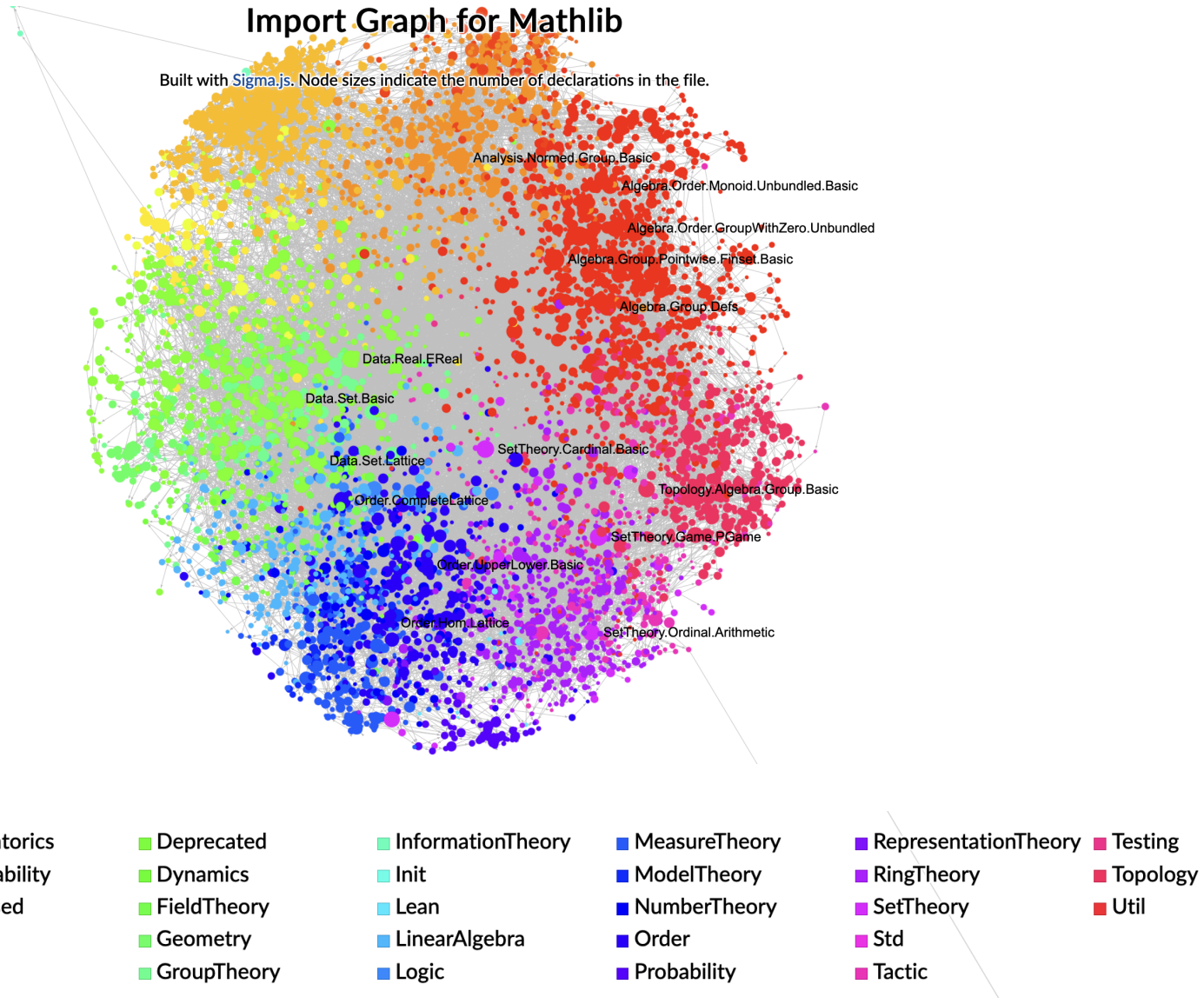
- **Verification of** mathematics, in particular of **modern research** results/ articles.
- **Documenting mathematics** in a gap and error free way.
- Build **searchable libraries** of mathematics.
- Educational tools / Teaching tools. Interactive textbooks.
- Mechanization / Automation of mathematics.
- AI.

Proof assistants are being used for work on all of these.

Lean

- Lean is a **functional programming language** and a **proof assistant**. Functional programming language means that functions are the primary building blocks. Programs are made by applying and composing functions.
- Lean was created by **Leonardo de Moura** at Microsoft Research in 2013.
- Lean is an open-source development, and since 2023 is also being developed at the **Lean Focused Research Organization**.

The Lean community is building a library. It is called **mathlib4**, and it currently looks like this:



Mathlib4

Mathlib is "a community-driven effort to build a unified library of mathematics formalized in the Lean proof assistant."
As of last week, Mathlib

- contains **103041 definitions** and **205699 theorems**,
- from **556 contributors**,
- is **maintained by 28** mathematicians and computer scientists.

We will be using Mathlib for the projects in this seminar.

A tiny bit on type theory in Lean

Recall that in **set theory**:
Sets are the primary objects and everything else is built out of them. For instance, a function $f : X \to Y$ is defined via its graph Γ, a subset of $X \times Y$. You can form sets like $\{2, \cos\}$ and statements like $\mathbb{Z} \in \{\pi\}$ formally make sense.
In **Type theory**:
Types are the primary notion. Everything has a **type** and you have to specify it, e.g.

$$2 : \mathbb{N}$$

says, 2 is a natural number.
We say 2 is **a term of type** $\mathbb{N}$.

Similarly,

$$\texttt{cos} \ : \ \mathbb{R} \to \mathbb{R}$$

says the type of $\cos$ is a function from $\mathbb{R}$ to $\mathbb{R}$.
The type of 2 and of $\cos$ is different, so something like $\{2, \cos\}$ wouldn't make sense to Lean. Writing the statement $\mathbb{Z} \in \{\pi\}$ would result in a type error.
The type is **unique**.
This means for example that $\texttt{2} \ : \ \mathbb{N}$ and $\texttt{2} \ : \ \mathbb{R}$ are different things.
This is confusing, because in mathematics we always automatically identify them without even thinking about it. Lean has an automatized mechanism to convert natural numbers to real numbers. It is called a **coercion** and it is used all the time.

Functions

There are various ways to declare functions in a programming language such as Lean (declarative and statically-typed). For instance via a *formula*:

```
def square : Nat → Nat :=
fun (n : Nat) => n * n

#eval square 2  -- 4
```

- The above definition makes sense because all the terms in the formula have been previously defined. In particular, a product function `*` has already been defined for natural numbers.
- That product function `*` takes two natural numbers and returns a natural number, so the function `square` is *well-typed* (it indeed takes a natural number and returns a natural number, as demanded by the *type signature* `Nat → Nat` of that function).

Recursive definitions

Functions can also be introduced via *pattern-matching* (which can be viewed as a form of case analysis):

```
def fact : Nat → Nat :=
fun (n : Nat) => match n with
| 0     => 1
| k + 1 => (k + 1) * (fact k)

#eval fact 3  -- 6
```

Note that the function `fact` calls on itself:

```
fact 3 = 3 * (fact 2)
       = 3 * (2 * (fact 1))
       = 3 * (2 * (1 * (fact 0))
       = 3 * (2 * (1 * 1))
       = 6
```

Lean uses a feature called *structural recursion* to guarantee that, for all natural number `n`, this process terminates.

Curried functions

Functions of two variables can be introduced using *product types*, as we usually do in mathematics (uncurried notation):

```
def add_u : Nat × Nat → Nat :=
fun ((n, m) : Nat × Nat) =>  n + m

#eval add_u (1, 1)  -- 2
```

Or as a function that sends a term to a function (curried notation[1]):

```
def add : Nat → (Nat → Nat) :=
fun (n : Nat) => (fun (m : Nat) =>  n + m)

#eval (add 1) 1  -- 2
```

[1] Thus called in honor of Haskell Curry (1900-1982), an American logician, mathematician, and computer scientist.

Functional programming

Curried functions can also be declared as follows, removing unnecessary brackets in the type signature and listing all arguments at once:

```
def add : Nat → Nat → Nat :=
fun (n : Nat) (m : Nat) =>  n + m
```

Note that we can simply write `add 1 1` instead of `(add 1) 1`.

```
#eval add 1 1  -- 2
```

This notation becomes particularly useful for functions of three variables or more. Note that we do not need to specify the types of the arguments: these are usually *inferred* by Lean from the type signature of the function.

```
def f : Nat → Nat → Nat → Nat :=
fun a b c => 2 * a + b + c * c

#eval f 1 0 3  -- 11
```

Higher-order functions

Functions that take a function as argument are usually called *higher-order functions*. Here is an example:

```
def F : (Nat → Nat) → Nat → Nat :=
fun (g : Nat → Nat) (n : Nat) => 2 * (g n)
```

Note that the term `g` does *not* have to be previously defined: the first argument of the function `F` can be passed as an *anonymous function*[2].

```
#eval F (fun (n : Nat) => n + 1) 2  -- 6
```

Observe that the arrow type `Nat → Nat → Nat → Nat` is, by definition, `Nat → (Nat → (Nat → Nat))`, and that this is different from the arrow type `(Nat → Nat) → Nat → Nat`.

[2] Also called a *λ-abstraction*.

Inductive types

- A type is a *quantification domain*. Inductive types are types whose terms are introduced as values of certain special functions, called *constructors*.
- In *Martin-Löf's type theory* (MLTT), on which Lean is based, the type of natural numbers is an inductive type with two constructors, called `Nat.zero : Nat` and `Nat.succ : Nat → Nat`.

```
inductive Nat : Type where
| zero           : Nat
| succ (n : Nat) : Nat
```

  The constructor `Nat.zero` takes no arguments, while `Nat.succ` takes one (a term of type `Nat`).

- Product types are also inductive types in MLTT, with only one constructor `Prod.mk : X → Y → X × Y`, taking two arguments.

```
inductive Prod (X : Type) (Y : Type) : Type where
| mk (x : X) (y : Y) : Prod X Y
```

Pattern-matching

To construct a function *out of* an inductive type, we can use pattern-matching on the constructors of that inductive type (case analysis). This is how we declared the factorial function, using slightly different notation.

```
def fact : Nat → Nat :=
fun (n : Nat) => match n with
| Nat.zero   => 1
| Nat.succ k => (k + 1) * (fact k)
```

This is also how the function `add_u` should have been defined (but, with Lean's definition of a product, we can by-pass that particular pattern-matching)[3]:

```
def add_u : Nat × Nat → Nat :=
fun (t : Nat × Nat) =>  match t with
| Prod.mk m n => m + n
```

[3] Products in Lean are *records*, and terms of record type, in Lean, are definitionally equal to their η-expansion.

Propositions and proofs

- In formal mathematics, *a proposition is a special kind of type*. Namely, it is a type `P` in which any two terms can be identified. This means that, to any two terms `(p : P)` and `(p' : P)`, there is associated an identification `p = p'`. Intuitively, this means that `P` has either zero or one element.
- This property is called *proof irrelevance* and, in Lean, it is built into the definition of the type `Prop`, so the equality $p = p'$ holds *by reflexivity* for terms `p p' : P` where `P : Prop`.

```
theorem proof_irrel (P : Prop) (p : P) (p' : P) : p = p' :=
rfl
```

- If `(P : Prop)` is a proposition, a term `(p : P)` is called a *proof* of `P`. You can think of the proposition with no terms as `False : Prop` and of the proposition with one term as `True : Prop`. These two types are usually introduced as inductive types.

Implications

- If `P` and `Q` are propositions, then the arrow type `P → Q` is a proposition[4]. That proposition is usually denoted by $P \Rightarrow Q$.
- A term `f : P → Q` transforms a proof `(p : P)` into a proof `(f p : Q)`, so we can think of the term `(f : P → Q)` as a proof of the fact that `P` *implies* `Q`.
- For example, we can construct a proof of the following implication:

```
example (n m : Nat) : n ≤ m → n ≤ m + 1 :=
by exact?  -- this will start a library search for you
```

- It also follows from this interpretation of propositions (and the fact that `False` is an inductive type with no constructors) that, for all `Q : Prop`, we have a term of type `False → Q`. This in turn implies that, if you want to prove something, you can always prove `False` instead (*exfalso quod libet*).

[4] This is a consequence of *function extensionality* (if for all $p : P$, we have $f_1(p) = f_2(p)$ in Q, then $f_1 = f_2$ in $P \to Q$) and it holds as soon as `Q` is a proposition (without any assumption on the type `P`).

Conjunctions

- If `P` and `Q` are propositions, then the product type `P × Q` is a proposition[5]. This proposition is usually denoted by $P \wedge Q$.
- To prove $P \wedge Q$ means to introduce a term of type `P × Q`. For that, it suffices to supply a proof of `P` and a proof of `Q`, and to use the constructor `Prod.mk`.

```
example (P Q : Prop) (p : P) (q : Q) : P × Q :=
Prod.mk p q
```

- To prove an implication $P \wedge Q \Rightarrow R$ means to construct a function $f_u : P \times Q \to R$. By what we have seen about curried notation, this is equivalent to constructing a function `f : P → Q → R`.

[5] This is again an extensionality result: two terms `(t t' : P × Q)` are equal, as terms of type `P × Q`, if and only if their components are equal, meaning that `t.1 = t'.1` in `P` and `t.2 = t'.2` in `Q`.

Natural deduction

- The *modus ponens* rule of natural deduction is given as follows:

$$\big((P \Rightarrow Q) \wedge P\big) \Rightarrow Q$$

  This proposition corresponds to the type `((P → Q) × P) → Q`, or equivalently `(P → Q) → P → Q`, where `P` and `Q` are propositions.

- The key observation is that this rule is *provable* in our typing system, since it corresponds to the evaluation of a function `(f : P → Q)` on a term `(p : P)`.

```
theorem mp (P Q : Prop) : (P → Q) → P → Q :=
fun (f : P → Q) (p : P) => f p
```

  This *type-checks* because the term `f p` is indeed of type `Q`.

- The *modus ponens* rule is then proved *because* we have constructed a term `mp P Q : (P → Q) → P → Q`. We say that the type `(P → Q) → P → Q` is *inhabited*.

A proof that $0 \leq 1$ in $\mathbb{N}$

In Lean, the usual ordering of the natural numbers is the *relation* `Nat.le` defined inductively as follows (`Nat.le n m` means `n ≤ m` in `Nat`).

```
inductive Nat.le : Nat → Nat → Prop where
| refl (n : Nat)    : Nat.le n n
| step (n m : Nat) : Nat.le n m → Nat.le n (m + 1)
```

We can use this to construct a proof of the proposition `0 ≤ 1` via *modus ponens*.

- `f := Nat.le.step 0 0` is a proof of the proposition `(0 ≤ 0) → (0 ≤ 1)`.
- `p := Nat.le.refl 0` is a proof of the proposition `(0 ≤ 0)`.
- Thus, the term `zero_leq_one := f p` is a proof of the proposition `(0 ≤ 1)`.

```
theorem zero_leq_one : 0 ≤ 1 :=
(Nat.le.step 0 0) (Nat.le.refl 0)
```

Summary

- In a programming language such as Lean, mathematical objects like the natural numbers and mathematical propositions like the *modus ponens* rule are represented as *types*.
- To construct a proof of a proposition `P`, we must write a *program* whose output is a term of type `P`.
- To write such a program, which is always a function of some kind, we need to learn the *syntax* of the language.
- We suggest to start with the following notions: *curried functions*, *higher-order functions*, *inductive types*, and *pattern-matching on constructors*.

Let's get started!

Today, you should:

- Either work on the following file to practice Lean's syntax.
- Or play the Natural Number Game, to get a first feel about *tactics*, which we will study more in detail next week.

And between today and next week, you should give a try to whichever activity you did not do today! More precisely, before 24.04.2025, you should:

1. Go through the whole Lean syntax file.
2. Play the Natural Number Game, all the way until *Multiplication World* and *Implication World*.

Further resources:

- https://leanprover-community.github.io/
- Online Lean Server: https://live.lean-lang.org/
- The Hitchhiker's Guide to Logical Verification 2025.

On Müsli you will find the address of the seminar webpage, where these slides and all other resources are kept.

You should decide before Friday 25.04 if you want to stay in the seminar (let us know via Müsli).

Zulip invitation link

We also ask you to join the following Zulip channel, through which we will communicate during the semester (you can use your GitHub account to log in, no need to create a Zulip account).