Day - 1

Touseef Haider

1.1. What is an Interactive Theorem Prover? And Lean's Role?

Imagine you're working on a complex math problem. You write down a step, and then pause to ask, "Is this step correct?" An **interactive theorem prover (ITP)** is like a meticulous assistant that checks your logic step-by-step.

- You state a mathematical claim (theorem or proposition).
- You prove it using logical rules and definitions.
- The ITP checks every step to ensure sound reasoning.
- It's called **interactive** because you are in constant dialogue with the software.

Lean is a modern interactive theorem prover used for:

- **Formalizing mathematics**: Creating verified mathematical definitions and proofs.
- **Software verification**: Ensuring programs meet specifications.
- **Education**: Teaching logic and precision in mathematical reasoning.

Think of it as a video game where the objective is to build a sound proof, and Lean is the game engine validating your moves!

1.2. The Basic Workflow: Editor, Lean, and the Infoview

You interact with Lean using an editor—most commonly **Visual Studio Code (VS Code)**—with the Lean 4 extension:

1. **Writing Code**: You type your Lean code into a file in VS Code (these files usually end in '.lean'). This code can be definitions, theorems you want to prove, or commands to Lean.

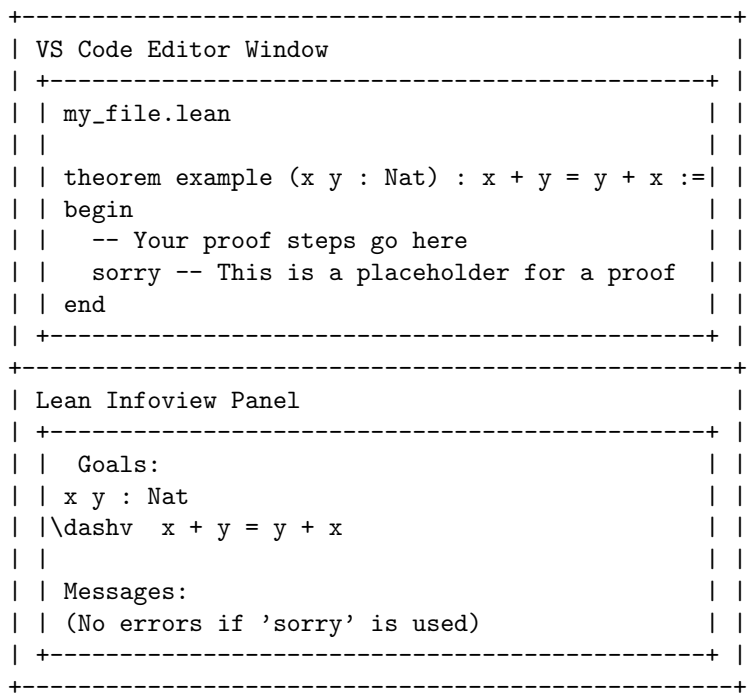
```
def hello := "Hello, Lean!"
#check hello

```

2. **Lean Processing**: As you type (or when you save the file, depending on the setup), Lean automatically processes your code in the background. It's constantly checking for errors, trying to understand your definitions, and figuring out the state of your proofs.
3. **Infoview Panel**: This is where the magic of the "feedback loop" really happens! The Lean extension in VS Code displays an "Infoview" panel. This panel is your main window into Lean's mind. It shows you:
    - **Goal State**: What is left to prove?
    - **Hypotheses**: What assumptions are available?
    - **Messages**: Lean will give you messages here. These could be:

- Information: Like the type of an expression if you use a command like #`check`.
- Warnings: If you're doing something that's not strictly wrong but might be problematic or unintentional.
- Errors: If you've made a syntax error, a logical mistake, or if a proof step doesn't work. The error messages are usually very specific and help you pinpoint the problem.

Conceptual Illustration:



In this illustration:

- The $\dashv x + y = y + x$ in the Infoview is the "goal" - it's what Lean is waiting for you to prove. The x y : Nat‘ above it are your available variables or hypotheses.
- If you made a mistake in a proof step, the "Messages" area would show an error.

So, you type, Lean thinks, and the Infoview updates. This cycle happens continuously, allowing you to experiment, make mistakes, and correct them quickly. It's a very dynamic way to work!

1.3. A Simple Example: `#check` and Lean's Response

Let's say you type the following into your Lean file in VS Code: Use `#check` to ask Lean the type of an expression:

```
#check Nat
#check 1 + 2
#check Nat \rightarrow Bool -- Function type from Nat to Bool
```

For `$check Nat`, the infoview would display tells you that 'Nat' (the type of natural numbers $0, 1, 2, \ldots$) is itself of type 'Type'. 'Type' is Lean's way of saying "this is a type". It's a bit meta, like saying "the category 'mammal' is a type of biological classification."

For `#check 1 + 2`, Lean evaluates '1 + 2‘ (or at least recognizes its structure) and tells you that the result is a 'Nat‘ (a natural number). It confirms that the expression '1 + 2‘ has the type 'Nat‘.

For `#check Nat → Bool`, This describes a function type. 'Nat → Bool' represents the type of functions that take a 'Nat' as input and return a 'Bool' (true or false) as output. Lean confirms that this function signature is a valid 'Type'.

Example Error:

```
#check 1 + "hello"
```

Lean's Infoview might show:

```
error: type mismatch
  "hello"
has type
  String
but is expected to have type
  Nat
```

This error message is Lean's feedback telling you that you can't add a 'String' to a 'Nat' directly because the '+' operator expects both sides to be numbers (or more precisely, to be of types that have an addition operation defined between them, and 'Nat' and 'String' usually don't). This immediate feedback is what makes Lean so powerful for learning and for ensuring correctness. You instantly know if what you've written makes sense to Lean according to its rules.