Lecture 1, Sept 8, 2026

Lean is a proof checker, not a theorem prover.

Theorem prover: you state the theorem, and the software tries to prove it.

Since the problem is undecidable, no software (not even AI) can do this reliably.

Proof checker: you state the theorem and the proof, and the software checks if your proof is correct.

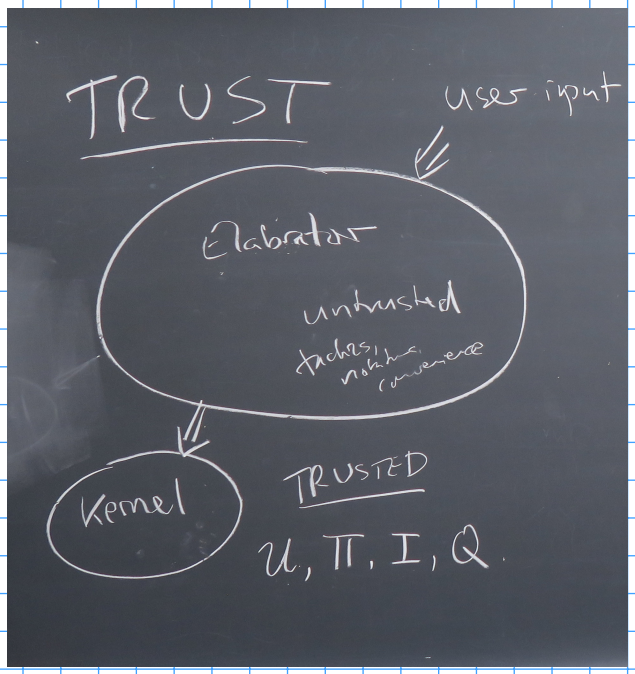
This is a decidable problem (in principle).

Lean is also a proof assistant, in the sense that it can not only check your proof, but also tell you what is missing. Therefore, it can be used to construct proofs interactively.

Lean has many features, and the user manual can read like an infinite list of "you can do this, you can also do that".

It is reassuring to know that Lean is organized as a very small "kernel", which only understands a handful of basic concepts, and a very large "elaborator", which provides many convenient notations but ultimately translates everything into the kernel language. We can think of kernel objects as the "actual" mathematical objects, and the eloborator language as "various notations for talking about them".

A major advantage of this design is that only the kernel needs to be trusted. The elaborator's job is to "figure out what the user meant", but no theorem of definition is accepted until it has been checked by the kernel.

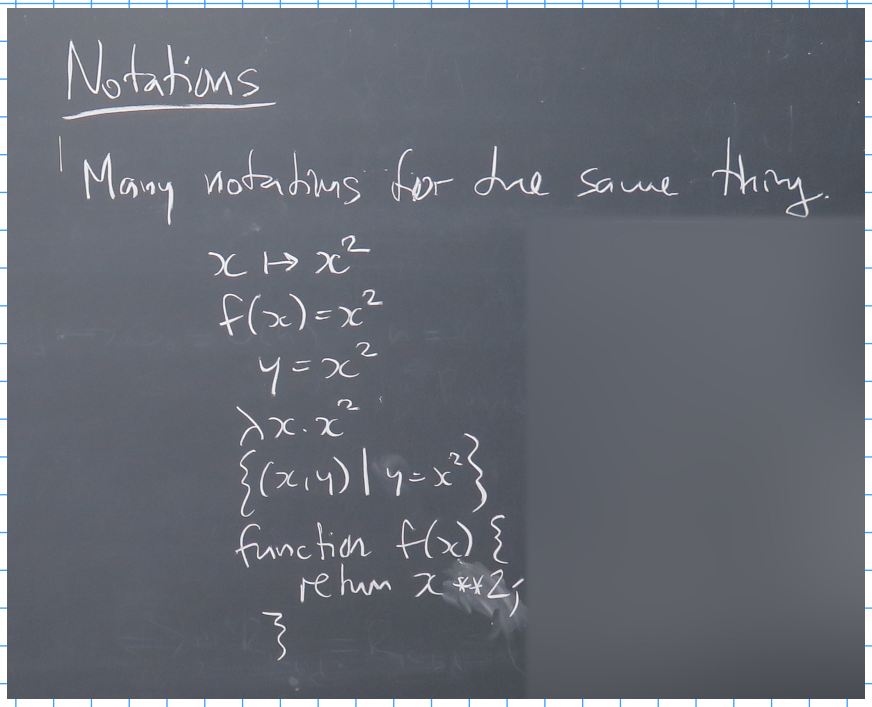


Let's talk about some things that mathematicians usually take for granted, but which might be difficult for computers.

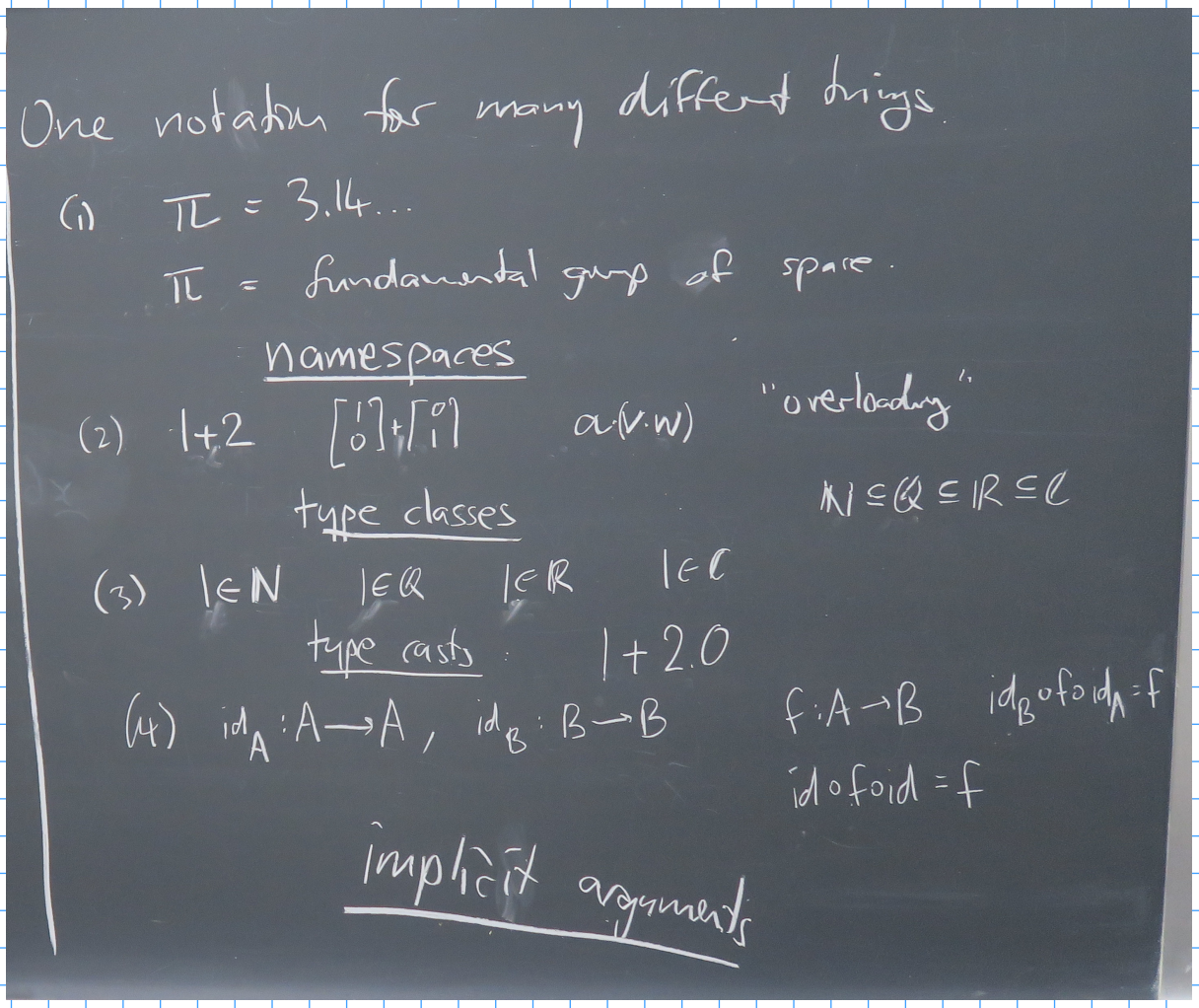
Notation

Notation can be ambiguous in two different ways.

(1) We can have several notations for the same thing.
This is perfectly fine and unproblematic.



(2) We can use the same notation for several different things.
This is not normally OK. We cannot have $x=2$ and also $x=3$.
But we do it anyway, all the time. The usual explanation is that it is "clear from the context" what was meant.

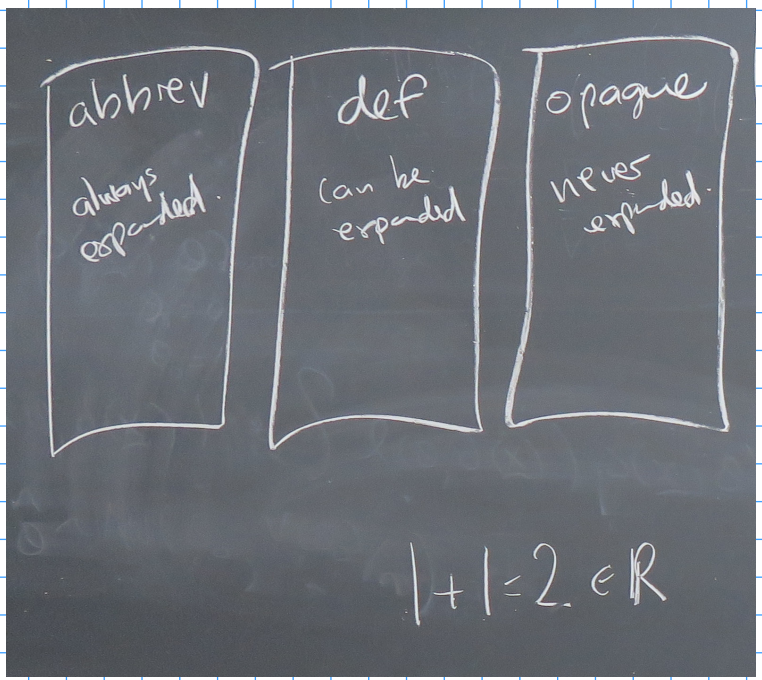


Lean (or more precisely, the Lean elaborator) has (at least) 4 different mechanisms for disambiguating such notations:
(1) namespaces, (2) type classes, (3) type casts, (4) implicit arguments.
You will learn about each of these later.

Definitions

Definitions are fundamental in mathematics. But what is a definition, really? Essentially a definition is a kind of abbreviation. We have a symbol (like a word), and a thing (like a term or a mathematical object or a property), and we declare that the symbol denotes that thing from now on.

The Lean elaborator distinguishes several kinds of definitions, depending on how eagerly the definition should be expanded. This sometimes matters: if I ask: "What is 1 + 1 in the real numbers", I expect the answer to be "2", and not the actual definition of the real number 2 as an equivalence class of Cauchy sequences of rational numbers.



Type Theory

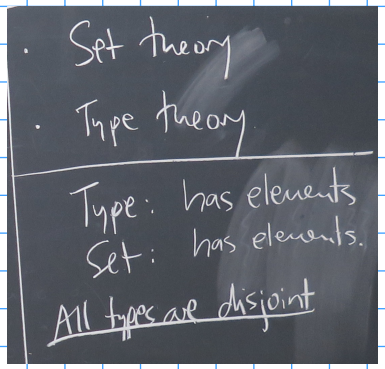
Mathematics is usually based on set theory – at least that is what we are taught to believe. In reality, almost nobody works in set theory directly. We use set theory for the foundations, and after that, we live in a world of notations.

Lean, and many other proof assistants, are based on type theory.

Types are in many ways like sets: they have elements, they have cardinality. There is one important difference: All types are disjoint. Each thing has one and only one type.

This is enormously helpful for formalization, and not so far from how we often use sets in practice. Mathematicians say surprisingly often "let's assume without loss of generality that the relevant sets are disjoint".

(Of course it is perfectly fine to talk about subsets of a type, such as sets of integers, like $\{1, 2, 3\}$. These are merely elements of the type "powerset of integers". However, $\{1, 2, 3\}$ is not itself a type.)



```
-- Lecture 1, 2026/09/08. Note that this lecture
-- also had a "prose" component, see the accompanying
-- PDF file.

-- Lean can be used as a calculator. The #eval command
-- asks Lean to evalate an expression:
#eval 2+1

-- Note that the answer is not displayed "inline": it
-- only shows up in the right-hand side panel (the "Lean
-- info" panel), and only when your cursor is on the
-- line in question.

-- In these commented lecture notes, I will sometimes
-- write "-->" to indicate what Lean's output will be.
-- This is for illustration purposes only. From
-- Lean's point of view, anything after "--" is a comment
-- and is ignored.
#eval 2+1         --> 3


-- We can use "def" to make a definition. Note that
-- ":=" means "is defined to be equal to".
def x := 1+2

-- You may have noticed that Lean did not print any
-- response to our line "def x := 1+2".
-- In fact, that is what we want. When Lean doesn't
-- say anything, it means everything is ok. Lean
-- only talks to us when there is an error, of if we
-- ask a specific question. The commands
-- that are "questions" all start with "#".

#eval x           --> 3

-- Every constant and expression in Lean has a type.
-- Use "#check" to ask for the type of an expression.
-- "Nat" is the type of natural numbers.
#check 1          --> 1 : Nat
#check 1+2        --> 3 : Nat
#check x          --> x : Nat
#check 19872634987163294876129387461928736491827364981727364981724 3

-- Lean is not afraid of large numbers. It knows that there
-- are infinitely many of them!

-- Not everything is a natural number. The next two
-- constants have type "Bool".
#check true       --> Bool.true : Bool
#check false      --> Bool.false : Bool

-- The following definition generates an error. Leans
-- does not like us to use the same name for more than
-- one thing.
def x := 1+3      --> ERROR: `x` has already been declared

-- But what if we want to use the variable x in one example
-- and then later in another example? We use namespaces.
-- A namespace is like a chapter in a book. All of the
-- things that are defined in a namespace stay in the
-- namespace. When the namespace ends, those things are
-- gone... kindof.

namespace Example1

  def x := 1+3
  #eval x         --> 4
  -- Note that x is now 4, and not 3.

end Example1

-- After the namespace is finished, we now see the old
-- value of x again:
#eval x           --> 3

-- But we can still access the "other" x too. We just
-- have to refer to it by its full name, which is
-- Example1.x.
#eval Example1.x  --> 4

-- We actually already saw namespaces (without knowing
-- it at the time) when we asked Lean about "true" and
-- "false". Lean likes to call them by their full names,
-- which are "Bool.true" and "Bool.false".
#check true       --> Bool.true : Bool
#check false      --> Bool.false : Bool

```