A Simple Utility for Literate Lean

George Zakhour

October 29, 2025

1 Introduction

The `literatelean.py` Python 3 program reads a literate lean file and produces three files:

1. `out.tex`: a LaTeX file meant to be compiled with `xelatex` and the accompanying `report.tex` files,
2. `out.lean`: a Lean file with all the literate text stripped away,
3. `out.bib`: a bibliography file containing the citations of the literate lean file.

This very document is compiled from the literate lean file `Example.lean`.

2 Simple Literate Lean

A literate lean file must be first and foremost a legal Lean file [2] that the lean compiler and checker approve of.

The text and its accompanying citations in the LaTeX file are always inside comments.

2.1 Human Readable Text

The human readable part of the document must be valid LaTeX, but it may not be a complete document. To the user's convenience we supplied a minimal `report.tex` LaTeX document and `Makefile` that may help you in getting started.

All text must appear between the comment delimeters: `/-@tex` and `@-/`, or more convienently between `/-@` and `@-/`. Each delimeter must appear on its own line with no text before or after it (whitespace is allowed).

Multiple human readable blocks are allowed in a document. The final LaTeX document will concatenate all these blocks in the same order that they appear in.

2.2 Bibliography

Bibliography data must be valid bibtex [3]. Like human readable text, these must appear inside comments and their delimeters are `/-@bib` and `@-/`. Only whitespace may appear around these delimeters on the line they occupy.

Just like human readable text, many bibliography blocks are allowed and the resulting bib file will be the concatenation of all the blocks.

Human-readable text and bibliography commenst may be arbitrarily nested.

2.3 Lean Text

As a literate lean program is a Lean program then Lean code does not appear inside comments.

By default every line of Lean code makes it in the LaTeX document inside a formated `minted` block. It is possible to hide parts of Lean code by wrapping them inside `-- {{{` and `-- }}}`. All the lean lines in between two comments will not be displayed and instead a `-- ...` will be displayed instead exactly where the opening delimiter is.

Lean comments are equally moved into the LaTeX output. That choice was made to encourage moving all comments into a literate format. It may be interesting to look into single line comments annotating some Lean code to be added as footnotes.

The `minted` block will always have a git logo in its top left that, when clicked, redirects to the relevant lines in the resulting stripped `out.lean` file.

For example, these three lines of Lean code start by evaluating the identity function to 1, then they check the type of the result of applying the identity function to 2, and finally checks the type of the identity function.

```
#eval  (λ x ↦ x) 1
-- 1
#check (λ x ↦ x) 2
-- (fun x => x) 2 : Nat
#check (λ x ↦ x)
-- fun x => x : ?m.1 → ?m.1
```

Sadly, this utility is too simple to use the Lean LSP to show the result of the `#eval` and `#check` calls.

Or even to show errors in a nicely formatted way.

But it does have a cool feature that allows you to reference definitions, theorems, inductive types, and type classes from the LaTeX code. This can be done using the `\lean` macro which the utility expands before passing it to the LaTeX compiler.

For example, `nat` is the definition of the Peano [1] numbers.

```
-- Peano Numbers
inductive nat where
  | zero
  | succ (n: nat)
```

The definition of addition, in `add`, is the following:

```
def add (n m: nat) : nat := match n with
  | .zero => m
  | .succ n' => nat.succ (add n' m)
```

And the proof that add is commutative, add_comm, is the following. We omit the proof at the base case as it's obvious. If you wish to see it you may do so by clicking on the git icon

```
-- This is the add_comm from the standard library, not ours!
#check Nat.add_comm
-- Nat.add_comm (n m : Nat) : n + m = m + n

theorem add_comm: ∀ (n m : nat), add n m = add m n := by▶
  intros▶ n; induction▶ n
  case▶ zero =>
    -- 3 lines hidden
  case▶ succ n ih =>
  -- n : nat
  -- ih : ∀ (m : nat), add n m = add m n
  -- ⊢ ∀ (m : nat), add n.succ m = add m n.succ
    intros▶ m; induction▶ m
    case▶ zero => simp only [add, ih]
    case▶ succ m' ih' =>
    -- ih : ∀ (m : nat), add n m = add m n
    -- ih' : add n.succ m' = add m' n.succ
    -- ⊢ add n.succ m'.succ = add m'.succ n.succ
      unfold▶ add
      rw [ih▶, ←ih'▶]
      unfold▶ add
      rw [ih▶]
```

The tool is again simple, it creates a label to definitions, theorems, inductive types, and type classes if and only if the declaration starts on a new line with the keyword `def`, `theorem`, `inductive`, and `class` keyword (respectively) being the first token of the line and the identifier to be bound is the second token.

2.4 Index

An index is automatically collected of all the definitions, theorems, inductive types, and type classes. If you wish to have them rendered in the final document you must have the two commands `\makeindex` and `\printindex`.

For example, in the following code, without even referring to the Printable type class in the text through the `\lean` macro, the symbol still makes it to the index.

```
class Printable (α: Type) where
  print: α → String

instance : Printable nat where
  print n := toString n
    where toString (n: nat) : String :=
      match n with
      | nat.zero => "0"
      | nat.succ n => "S" ++ toString n

instance : Printable String where
  print x := x

#eval Printable.print (nat.succ (nat.succ nat.zero))
-- "SS0"
#eval Printable.print "Hello, World!"
-- "Hello, World!"
```

References

[1] KENNEDY, H. *Peano: life and works of Giuseppe Peano*, vol. 4. Springer Science & Business Media, 2012.

[2] MOURA, L. D., AND ULLRICH, S. The lean 4 theorem prover and programming language. In *International Conference on Automated Deduction* (2021), Springer, pp. 625–635.

[3] PATASHNIK, O. Bibtex 101. *TUGboat 15* (1984), 269–273.

Index

A Proof Goals

◀ Line 21: by

```
⊢ ∀ (n m : nat), add n m = add m n
```

◀ Line 22: intros

```
n : nat
⊢ ∀ (m : nat), add n m = add m n
```

◀ Line 22: induction

```
case zero
⊢ ∀ (m : nat), add nat.zero m = add m nat.zero

case succ
n† : nat
n_ih† : ∀ (m : nat), add n† m = add m n†
⊢ ∀ (m : nat), add n†.succ m = add m n†.succ
```

◀ Line 23: case

```
⊢ ∀ (m : nat), add nat.zero m = add m nat.zero
```

◀ Line 27: case

```
n : nat
ih : ∀ (m : nat), add n m = add m n
⊢ ∀ (m : nat), add n.succ m = add m n.succ
```

◀ Line 28: intros

```
n : nat
ih : ∀ (m : nat), add n m = add m n
m : nat
⊢ add n.succ m = add m n.succ
```

◀ Line 28: induction

```
case zero
n : nat
ih : ∀ (m : nat), add n m = add m n
⊢ add n.succ nat.zero = add nat.zero n.succ

case succ
n : nat
ih : ∀ (m : nat), add n m = add m n
n† : nat
n_ih† : add n.succ n† = add n† n.succ
⊢ add n.succ n†.succ = add n†.succ n.succ
```

◀ Line 29: case

```
n : nat
ih : ∀ (m : nat), add n m = add m n
⊢ add n.succ nat.zero = add nat.zero n.succ
```

◀ Line 30: case

```
n : nat
ih : ∀ (m : nat), add n m = add m n
m' : nat
ih' : add n.succ m' = add m' n.succ
⊢ add n.succ m'.succ = add m'.succ n.succ
```

◀ Line 31: unfold

```
n : nat
ih : ∀ (m : nat), add n m = add m n
m' : nat
ih' : add n.succ m' = add m' n.succ
⊢ (add n m'.succ).succ = (add m' n.succ).succ
```

◀ Line 32: ih

```
n : nat
ih : ∀ (m : nat), add n m = add m n
m' : nat
ih' : add n.succ m' = add m' n.succ
⊢ (add m'.succ n).succ = (add m' n.succ).succ
```

◀ Line 32: ←ih'

```
n : nat
ih : ∀ (m : nat), add n m = add m n
m' : nat
ih' : add n.succ m' = add m' n.succ
⊢ (add m'.succ n).succ = (add n.succ m').succ
```

◀ Line 33: unfold

```
n : nat
ih : ∀ (m : nat), add n m = add m n
m' : nat
ih' : add n.succ m' = add m' n.succ
⊢ (add m' n).succ.succ = (add n m').succ.succ
```

◀ Line 34: ih

```
n : nat
ih : ∀ (m : nat), add n m = add m n
m' : nat
ih' : add n.succ m' = add m' n.succ
⊢ (add m' n).succ.succ = (add m' n).succ.succ
```