An introduction to linters

Damiano Testa

University of Warwick

2025 Lean Together

January 14, 2025

What are linters

Wikipedia has a Lint (software) entry:

> *Lint is the computer science term for a static code analysis tool used to flag programming errors, bugs, stylistic errors and suspicious constructs.*

The word was used by Stephen C. Johnson, while debugging the yacc.

If you used Mathlib, you may be familiar with two kinds of linters:

- environment linters, and
- syntax linters.

Environment and syntax linters

There are currently 17 environment and 29 syntax linters in Mathlib and dependencies (including the ones in Lean itself).

Environment linters

- perform essentially arbitrary code on each declaration;
- have natural access to the Environment, not the Syntax;
- warn you after the fact;
- good for *global* validation (e.g. `simp` normal form checks).

Syntax linters

- perform essentially arbitrary code on each Syntax tree;
- have natural access to the Environment;
- warn you right away;
- good for *local* validation (e.g. the `refine'` "deprecation").

What happens when you type `example : True := trivial`?

- Lean converts this command into a syntax tree

```
node Lean.Parser.Command.declaration, none
|-node Lean.Parser.Command.declModifiers, none
|   |-node null, none
|   |-node null, none
|   |-node null, none
|   |-node null, none
|   |-node null, none
|   |-node null, none
|-node Lean.Parser.Command.example, none
|   |-atom original: ⟨⟩⟨ ⟩-- 'example'
|   |-node Lean.Parser.Command.optDeclSig, none
|   |   |-node null, none
|   |   |-node null, none
|   |   |   |-node Lean.Parser.Term.typeSpec, none
|   |   |   |   |-atom original: ⟨⟩⟨ ⟩-- ':'
|   |   |   |   |-ident original: ⟨⟩⟨ ⟩-- (True,True)
|   |-node Lean.Parser.Command.declValSimple, none
|   |   |-atom original: ⟨⟩⟨ ⟩-- ':='
|   |   |-ident original: ⟨⟩⟨␣␣␣⟩-- (trivial,trivial)
|   |   |-node Lean.Parser.Termination.suffix, none
|   |   |   |-node null, none
|   |   |   |-node null, none
|   |   |-node null, none
```

- Elaborates it to the type `True` and the value `trivial : True`.
- (Lots of other computations.)
- Eventually, discards everything, since this was an `example`.

In the previous summary,

- environment linters see the final state;
- syntax linters see the whole process.

An environment linter would have a hard time detecting

- `set_option pp.all true`, see setOptionLinter;
- `lemma` vs `theorem`, see `lemmaThmLinter`;
- non-terminal `simp`s, see flexibleLinter.

Environment linters are *on-demand*: when you want to run their code, you should *do something*.

Syntax linters are *live*: their code runs *after every command*.

Typically, you build your whole project and then you use the environment linters to see what you've done.

This is great for large scale, far-away interactions between declarations.

A good example of this sort of check is the `simp`-normal-form linter, that performs some checks to ensure that the `simp` attribute is used

- consistently,
- confluently and
- minimally

in Mathlib.

Before this, every single command that you typed while developing the project would have been inspected by all the syntax linters.

Example: grep and linters

Our task is to figure out how many examples there are in Mathlib.

```
$ git grep 'example' Mathlib/* | wc -l
    1502
-- rule out, e.g., 'examples'
$ git grep 'example ' Mathlib/* | wc -l
    930
-- typically, declarations that are 'example's begin a line
$ git grep '^example ' Mathlib/* | wc -l
    572
```

What about noncomputable example?

grep/regular expressions are great for a quick estimate: there are somewhere between 500 and 1000 examples in Mathlib[1].

For the “linter way”, let’s look at syntax trees.

`example : True := trivial`

```
node Lean.Parser.Command.declaration, none
|-node Lean.Parser.Command.declModifiers, none
|   |-node null, none
|   |-node null, none
|   |-node null, none
|   |-node null, none
|   |-node null, none
|   |-node null, none
|-node Lean.Parser.Command.example, none
|   |-atom original: ⟨⟩⟨ ⟩-- 'example'
|   |-node Lean.Parser.Command.optDeclSig, none
|   |   |-node null, none
|   |   |-node null, none
|   |   |   |-node Lean.Parser.Term.typeSpec, none
|   |   |   |   |-atom original: ⟨⟩⟨ ⟩-- ':'
|   |   |   |   |-ident original: ⟨⟩⟨ ⟩-- (True,True)
|   |-node Lean.Parser.Command.declValSimple, none
|   |   |-atom original: ⟨⟩⟨ ⟩-- ':='
|   |   |-ident original: ⟨⟩⟨␣␣⟩-- (trivial,trivial)
|   |   |-node Lean.Parser.Termination.suffix, none
|   |   |   |-node null, none
|   |   |   |-node null, none
|   |   |-node null, none
```

`theorem X : True := trivial`

```
node Lean.Parser.Command.declaration, none
|-node Lean.Parser.Command.declModifiers, none
|   |-node null, none
|   |-node null, none
|   |-node null, none
|   |-node null, none
|   |-node null, none
|   |-node null, none
|-node Lean.Parser.Command.theorem, none
|   |-atom original: ⟨⟩⟨ ⟩-- 'theorem'
|   |-node Lean.Parser.Command.declId, none
|   |   |-ident original: ⟨⟩⟨ ⟩-- (X,X)
|   |   |-node null, none
|   |-node Lean.Parser.Command.declSig, none
|   |   |-node null, none
|   |   |-node Lean.Parser.Term.typeSpec, none
|   |   |   |-atom original: ⟨⟩⟨ ⟩-- ':'
|   |   |   |-ident original: ⟨⟩⟨ ⟩-- (True,True)
|   |-node Lean.Parser.Command.declValSimple, none
|   |   |-atom original: ⟨⟩⟨ ⟩-- ':='
|   |   |-ident original: ⟨⟩⟨␣␣⟩-- (trivial,trivial)
|   |   |-node Lean.Parser.Termination.suffix, none
|   |   |   |-node null, none
|   |   |   |-node null, none
|   |   |-node null, none
```

[1] Answer: 387! See branch#adomani/count_examples.

What else can we do with (syntax) linters?

- Check for duplicated namespace;
- flag unused tactics;
- deprecate `refine'` vs `refine`, `admit` vs `sorry`, `$` vs `<|`, `.` vs `·`;
- highlight variables that have been named, but not used;
- long lines, long files, copyright validation, non-terminal `simp`s, unfocused goals, large imports, `pp` options, ...

and so on!

Besides Syntax trees and the Environment, linters can (and often do) also inspect InfoTrees, gaining access to more information.

I may say something about this, if time permits, later on.

`MinImports` linter

The minImports linter flags each command in each file that requires more imports than what the imports so far are.

The data produced by this linter is then posted weekly on the Zulip Late importers report channel.

Limitation. Linters can access the environment, but all[2] modifications get reverted when their execution ends.

This linter has a mechanism for persisting the information of what imports have been used so far that assumes that the file is parsed linearly.

[2] All except for emitted messages, of course!

What may future linters do?

Linters in-progress/prototypes

- Papercut – warns about subtraction in $\mathbb{N}$ and more
- MetaTesting – expands test suites for tactics
- Refactors – extract "connected" declarations for minimization and debugging
- Identify unused code, such as unnecessary `variable`s, `set_option`s, `open`s, `nolint`s,...
- Repeated typeclass assumptions – e.g. warning on `variable [Add R] [Ring R]`

InfoTrees

I talked mostly about Syntax and Environment.

Some of the linters that I mentioned earlier would not really be able to perform their tasks without another crucial source of information: access to InfoTrees.

An InfoTree is another tree-like structure, like Syntax and Expressions.

It contains information about *everything*: Syntax, typing information for declarations, metavariables, local contexts, tactics, goal states...

Costs

`Nuisance vs helpfulness`: it is tricky to strike a balance

- formalizing new results,
- preparing a PR to Mathlib,
- teaching a module,
- giving a presentation,
- working on a Mathlib dependency,

all have different kinds of expectations from each linter (header, setOption, multiGoal, haveLet, docPrime).

`Performance concerns`

- Well, they exist!
- Hopefully, I will learn about making linters more performant soon!

Thank you!

Questions?