Project proposal: expl3 static analyzer

Vít Starý Novotný

2024-09-06

Contents

1 Introduction

In 2021, I used the expl3 programming language for the first time in my life. I had already been eyeing expl3 for some time and, when it came to defining a LaTeX-specific interface for processing YAML metadata in version 2.11.0 of the Markdown package for TeX, I took the plunge.

After two and a half years, approximately 3.5k out of the 5k lines of TeX code in version 3.5.0 of the Markdown package are written in expl3. I also developed several consumer products with it, and I have written three journal articles for my local TeX users group about it. Needless to say, expl3 has been a blast for me!

In the Markdown package, each change is reviewed by a number of automated static analysis tools (so-called *linters*), which look for programming errors in the code. While these tools don't catch all programming errors, they have proven extremely useful in catching the typos that inevitably start trickling in after 2AM.

Since the Markdown package contains code in different programming languages, we use many different linters such as `shellcheck` for shell scripts, `luacheck` for Lua, and `flake8` and `pytype` for Python. However, since no linters for expl3 exist, typos are often only caught by regression tests, human reviewers, and sometimes even by our users after a release. Nobody is happy about this.

Earlier this year, I realized that, unlike TeX, expl3 has the following two properties that seem to make it well-suited to static analysis:

1. Simple uniform syntax: (Almost) all operations are expressed as function calls. This Lisp-like quality makes is easy to convert well-behaved expl3 programs that only use high-level interfaces into abstract syntax trees. This is a prerequisite for accurate static analysis.
2. Explicit type and scope: Variables and constants are separate from functions. Each variable is either local or global. Variables and constants are explicitly typed. This information makes it easy to detect common programming errors related to the incorrect use of variables.

For the longest time, I wanted to try my hand at building a linter from the ground up. Therefore, I decided to kill two birds with one stone and improve the tooling for expl3 while learning something new along the way by building a linter for expl3.

2 Requirements

In this section, I outline the requirements for the linter. These will form the basis of the design and the implementation.

2.1 Functional requirements

The linter should accept a list of input expl3 files. Then, the linter should process each input file and print out issues it has identified with the file.

Initially, the linter should recognize at least the following types of issues:

- Style:
  - Overly long lines
  - Missing stylistic white-spaces
  - Malformed names of functions, variables, constants, quarks, and scan marks
- Functions:
  - Multiply defined functions and function variants
  - Calling undefined functions and function variants
  - Calling deprecated and removed functions
  - Unknown argument specifiers
  - Unexpected function call arguments
  - Unused private functions and function variants
- Variables:
  - Multiply declared variables and constants
  - Using undefined variables and constants
  - Using variables of incompatible types
  - Using deprecated and removed variables and constants
  - Setting constants and undeclared variables
  - Unused variables and constants
  - Locally setting global variables and vice versa

2.2 Non-functional requirements

2.2.1 Issues

The linter should make distinction between two types of issues: warnings and errors. As a rule of thumb, whereas warnings are suggestions about best practices, errors will likely result in runtime errors.

Here are three examples of warnings:

- Missing stylistic white-spaces around curly braces
- Using deprecated functions and variables
- Unused variable or constant

Here are three examples of errors:

- Using an undefined message
- Calling a function with a `V`-type argument with a variable or constant that does not support `V`-type expansion
- Multiply declared variable or constant

The overriding design goal for the initial releases of the linter should be the simplicity of implementation and robustness to unexpected input. For all issues, the linter should prefer precision over recall and only print them out when it is reasonably certain that it has understood the code, even at the expense of potentially missing some issues.

Each issue should be assigned a unique identifier. Using these identifiers, issues can be disabled globally using a config file, for individual input files from the command-line, and for sections of code or individual lines of code using TEX comments.

2.2.2 Architecture

To make the linter easy to use in continuous integration pipelines, it should be written in Lua 5.3 using just the standard Lua library. One possible exception is checking whether functions, variables, and other symbols from the input files are expl3 build-ins. This may require using the `texlua` interpreter and a minimal TEX distribution that includes the LATEX3 kernel, at least initially.

The linter should process input files in a series of discrete steps, which should be represented as Lua modules. Users should be able to import the modules into their Lua code and use them independently on the rest of the linter.

Each step should process the input received from the previous step, identify any issues with the input, and transform the input to an output format appropriate for the next step. The default command-line script for the linter should execute all steps and print out issues from all steps. Users should be able to easily adapt the default script in the following ways:

1. Change how the linter discovers input files.
2. Change or replace processing steps or insert additional steps.
3. Change how the linter reacts to issues with the input files.

The linter should integrate easily with text editors. Therefore, the linter should either directly support the language server protocol (LSP) or be designed in a way that makes it easy to write an LSP wrapper for it.

2.2.3 Validation

As a part of the test-driven development paradigm, all issues identified by a processing step should have at least one associated test in the code repository of the linter. All tests should be executed periodically during the development of the linter.

As a part of the dogfooding paradigm, the linter should be used in the continuous integration pipeline of the Markdown Package for TEX since the initial releases of the linter in order to collect early user feedback. Other early adopters are also welcome to try the initial releases of the linter and report issues to its code repository.

At some point, a larger-scale validation should be conducted as an experimental part of a TUGboat article that will introduce the linter to the wider TEX

community. In this validation, all expl3 packages from current and historical TeX Live distributions should be processed with the linter. The results should be evaluated both quantitatively and qualitatively. While the quantitative evaluation should focus mainly on trends in how expl3 is used in packages, the qualitative evaluation should explore the shortcomings of the linter and ideas for future improvements.

2.2.4 License terms

The linter should be free software and dual-licensed under the GNU General Public License (GNU GPL) 2.0 or later and the LaTeX Project Public License (LPPL) 1.3c or later.

The option to use GNU GPL 2.0 or later is motivated by the fact that GNU GPL 2.0 and 3.0 are mutually incompatible. Supporting both GNU GPL 2.0 and 3.0 extends the number of free open-source projects that will be able to alter and redistribute the linter.

The option to use LPPL 1.3c is motivated by the fact that it imposes very few licensing restrictions on TeX users. Furthermore, it also preserves the integrity of TeX distributions by enforcing its naming and maintenance clauses, which ensure ongoing project stewardship and prevent confusion between modified and official versions.

Admittedly, GNU GPL and LPPL may seem like an unusual combination, since GNU GPL is a copyleft license whereas LPPL is a permissive license. However, there are strategic benefits to offering both.

We would offer LPPL as the primary license for derivative works within the TeX ecosystem. One downside of using LPPL is that it could potentially allow bad actors to create proprietary derivative works without contributing back to the original project. However, this trade-off helps maintain the TeX ecosystem's consistency and reliability. Incidentally, there is an element of trust in the TeX user community to voluntarily contribute improvements back, even though the license itself does not mandate it.

We would offer GNU GPL as an alternative license for derivative works outside the TeX ecosystem. The key benefit of including GNU GPL is that it enables the code to be integrated into free open-source projects, especially those with licenses that are incompatible with LPPL's naming requirements. This opens the door for broader collaboration with the free software community.

Notably, GNU GPL creates a one-way licensing situation: Once a derivative work is licensed under GNU GPL, it cannot be legally re-licensed under a less restrictive license like LPPL. As a result, we wouldn't be able to incorporate changes made to GNU GPL-licensed works back into the original project under LPPL without also creating two forks of the project licensed under GNU GPL 2.0 and GNU GPL 3.0, respectively. While this might seem like a downside, I view it as an important counterbalance to the potential for proprietary derivative works under LPPL.

In summary, this dual-licensing approach allows us to maintain the integrity of the TeX ecosystem while making the project more accessible to the broader

free open-source community. It provides flexibility for different use cases, though we will need to carefully manage contributions to ensure compliance with all licenses.

3 Related work

In this section, I review the related work in the analysis of TEX programs and documents. This related work should be considered in the design of the linter and reused whenever it is appropriate and compatible with the license of the linter.

3.1 Unravel

The unravel package by Bruno Le Floch analyses of expl3 programs as well as TEX programs and documents in general. The package was suggested to me as related work by Joseph Wright in personal correspondence.

Unlike a linter, which performs *static* analysis by leafing through the code and makes suggestions, unravel is a *debugger* that is used for *dynamic* analysis. It allows the user to step through the execution of code while providing extra information about the state of TEX. Unravel is written in expl3 and emulates TEX primitives using expl3 functions. It has been released under the LATEX Project Public License (LPPL) 1.3c.

While both linters and debuggers are valuable in producing bug-free software, linters prevent bugs by proactively pointing out potential bugs without any user interaction, whereas debuggers are typically used interactively to determine the cause of a bug after it has already manifested.

3.2 Chktex, chklref, cmdtrack, lacheck, match_parens, nag, and tex2tok

The Comprehensive TEX Archive Network (CTAN) lists related software projects on the topics of debuging support and LATEX quality, some of which I list in this section.

The chktex package by Jens T. Berger Thielemann is a linter for the static analysis of LATEX documents. It has been written in ANSI C and released under the GNU GPL 2.0 license. The types of issues with the input files and how they are reported to the user can be configured to some extent from the command-line and using configuration files to a larger extent. Chktex is extensible and, in addition to the configuration of existing issues, it allows the definition of new types of issues using regular expressions.

The lacheck package by Kresten Krab Thorup is a linter for the static analysis of LATEX documents. Similarly to chktex, lacheck has been written in ANSI C and released under the GNU GPL 1.0 license. Unlike chktex, lacheck cannot be configured either from the command-line or using configuration files.

The chklref package by Jérôme Lelong is a linter for the static analysis of LaTeX documents. It has been written in Perl and released under the GNU GPL 3.0 license. Unlike chktex, chklref focuses just on the detection of unused labels, which often accumulate over the lifetime of a LaTeX document.

The match_parens package by Wybo Dekker is a linter for the static analysis of expl3 programs as well as TeX programs and documents in general. It has been written in Ruby and released under the GNU GPL 1.0 license. Unlike chktex, match_parens focuses just on the detection of mismatched paired punctuation, such as parentheses, braces, brackets, and quotation marks. As such, it can also be used for the static analysis of natural text as well as programs and documents in programming and markup languages that use paired punctuation in its syntax.

The cmdtrack package by Michael John Downes is a debugger for the dynamic analysis of LaTeX documents. It has been written in LaTeX and released under the LPPL 1.0 license. It detects unused user-defined commands, which also often accumulate over the lifetime of a LaTeX document, and mentions them in the `.log` file produced during the compilation of a LaTeX document.

The nag package by Ulrich Michael Schwarz is a debugger for the dynamic analysis of LaTeX documents. Similarly to cmdtrack, nag has also been written in LaTeX and released under the LPPL 1.0 license. It detects the use of obsolete LaTeX commands, document classes, and packages and mentions them in the `.log` file produced during the compilation of a LaTeX document.

The tex2tok package by Jonathan Fine is a debugger for the dynamic analysis of expl3 programs as well as TeX programs and documents in general. It has been written in TeX and released under the GNU GPL 2.0 license. It executes a TeX file and produces a new `.tok` file with a list of TeX tokens in the file. Compared to static analysis, the dynamic analysis ensures correct category codes. However, it requires the execution of the TeX file, which may take long or never complete in the presence of bugs in the code.

3.3 Luacheck and flake8

Luacheck by Peter Melnichenko and flake8 by Tarek Ziade are linters for the static analysis of Lua and Python programs, respectively. They have been written in Lua and Python, respectively, and released under the MIT license. Both tools are widely used and should inform the design of my linter in terms of architecture, configuration, and extensibility.

Similar to chktex, the types of issues with the input files and how they are reported to the user can be configured from the command-line and using configuration files. Additionally, the reporting can also be enabled or disabled in the code of the analyzed program using inline comments.

Unlike luacheck, which is not extensible at the time of writing and only allows the configuration of existing issues, flake8 supports Python extensions that can add support for new types of issues.

3.4 TeXLab and digestif

TeXLab by Eric and Patrick Förscher and digestif by Augusto Stoffel are language servers for the static analysis of TEX programs and documents. They have been written in Rust and Lua, respectively, and released under the GNU GPL 3.0 license. The language servers were suggested to me as related work by Michal Hoftich at TUG 2024.

Whereas TEXLab focuses on LATEX documents, digestif also supports other formats such as ConTEXt and GNU Texinfo. Neither TEXLab nor digestif support expl3 code at the time of writing.

In terms of the programming language, license, and scope, digestif seems like the most related work to my linter. However, its GNU GPL 3.0 license is incompatible with the dual license of the linter, which prohibits code reuse.

4 Design

In this section, I outline the design of the linter and I create a code repository for the linter.

4.1 Processing steps

As outlined in the requirements, the linter will process input files in a series of discrete steps, each represented by a single Lua module.

Here are the individual processing steps that should be supported by the linter:

1. Preprocessing: Determine which parts of the input files contain expl3 code.
2. Lexical analysis: Convert expl3 parts of the input files into TEX tokens.
3. Syntactic analysis: Convert TEX tokens into a tree of function calls.
4. Semantic analysis: Determine the meaning of the different function calls.
5. Flow analysis: Determine additional emergent properties of the code.

4.2 Warnings and errors

As also outlined in the requirements, each processing step should identify issues with the output and produce either a warning or an error. Furthermore, the requirements list 16 types of issues that should be recognized by the linter at a minimum. Lastly, the requirements require that, as a part of the test-driven development paradigm, all issues identified by a processing step should have at least one associated test in the code repository of the linter.

In a document titled "Warnings and errors for the expl3 analysis tool", I compiled a list of 66 warnings and errors that should be recognized by the initial version of the linter. For each issue, there is also an example of expl3 code with and without the issue. These examples can be directly converted to tests and used during the development of the corresponding processing steps.

4.3 Limitations

Due to the dynamic nature of TEX, initial versions of the linter will make some naïve assumption and simplification during the analysis, such as:

- Assume default expl3 catcodes everywhere.
- Ignore non-expl3 and third-party code.
- Do not analyze expansion and key–value calls.

As a result, the initial version of the linter may not have a sufficient understanding of expl3 code to support proper flow analysis. Instead, the initial version of the linter may need to use pseudo-flow-analysis that would check for simple cases of the warnings and errors from flow analysis. Future versions of the linter should improve their code understanding to the point where proper flow analysis can be performed.

The warnings and errors in this document do not cover the complete expl3 language. The limitations currently include the areas outlined in a section of the document with warnings and errors titled "Caveats". Future versions of the linter should improve the coverage.

4.4 Code repository

I created a repository `witiko/expltools` titled "Development tools for expl3 programmers" at GitHub. As outlined in the requirements, I dual-license the code under GNU GPL 2.0 or later and LPPL 1.3c or later.

Furthermore, I also registered the expl3 prefix `expltools`, so that it can be used in the documentation for the linter, in other supporting expl3 code used in the linter, and also possibly in development tools for expl3 programmers other than the linter.

5 Benefits of grant funding

Securing this grant will significantly enhance my ability to dedicate focused and uninterrupted time to this project, enabling me to allocate at least two full weeks of work over the next 12 months. This concentrated effort will be far more productive than the fragmented hours I currently manage to find after a long day's work, ensuring that I can make substantial progress.

Additionally, the grant will serve as a meaningful endorsement of the project's value, reflecting the community's interest and support. This recognition will not only reinforce the importance of the work but also help attract other contributors who share a commitment to advancing the project.

Finally, the visibility that comes with receiving this grant will elevate the project's profile, making it more prominent within the TeX community and beyond. This increased visibility is crucial for attracting further interest, feedback, and potential collaborations, all of which are vital for the project's long-term success.