Best-effort LaTeX parsing for interactive editing

Mathias Jakobsen

Abstract

Overleaf is a LaTeX editor with contextual assistance and interactive editing tools powered by a "best-effort" parsing engine. Rather than trying to fully understand all possible TeX documents, we focus on understanding conventional LaTeX structures where we can provide assistance to the user. This paper details the strategies behind this forgiving parser, and describes how we use the output of the parser to enable WYSIWYG editing features.

1 Introduction

Overleaf provides two different editing modes for LaTeX files: The Code Editor and the Visual Editor [1]. The Code Editor shows the LaTeX code with syntax highlighting, while the Visual Editor hides some of the LaTeX markup in favour of WYSIWYG elements, like replacing `\textbf{bold text}` with **bold text** directly in the editor, or indeed replacing entire `\begin{figure}...\end{figure}` environments with actual figures. Shared amongst both editors is the need to provide contextual assistance to users during editing. For example, autocompletion should provide *contextual suggestions*, e.g. when you are writing `\cite{...}` citation keys should be suggested, and when writing `\begin{...}` environment names should be suggested.

The Visual Editor comes with further requirements to the level of understanding required of LaTeX documents. While the Visual Editor does "replace" LaTeX code with WYSIWYG tools, the underlying LaTeX code is still the source of truth. Any changes performed in the WYSIWYG tool must be transformed into changes to the LaTeX code, which will in turn update the visual representation. To facilitate this, and any feature that relies on document contents, we require a deep and structural understanding of the commands, environments, and modes present in a document.

1.1 Challenges of TeX parsing

Parsing TeX is difficult due to its dynamic nature. The way that a TeX document is interpreted means that the rules of tokenisation can be changed on the fly, which complicates using traditional parsing techniques. A crucial component of parsing a TeX document is the use of *category codes* [2], a mapping from characters to categories, each of which are handled differently during interpretation. Consider for example the following TeX snippet:

Mathias Jakobsen

doi.org/10.47397/tb/46-2/tb143jakobsen-parsing-editing

`! \catcode`!=0 !TeX`. The first time the interpreter encounters the `!` character, it is assigned the "other" category code (meaning that it is not a letter, and not particularly special). Following this, the command `\catcode`!=0` is encountered and evaluated, leading the interpreter to internally assign category code 0 to subsequent encounters of the `!` character. Category 0 is for "escape characters", those characters that indicate a control sequence (normally the \ character). Resuming tokenisation, the interpreter will encounter the `!` character, a category 0 character, which means that the subsequent `TeX` should be treated as a control sequence. The resulting output of the snippet will be "!TEX", with the first `!` being typeset as a character, and the second `!` being interpreted as the start of a control sequence. What this example illustrates is that due to the dynamic nature of TEX interpretation, we cannot assign static tokenisation rules, and expect to parse *all* TEX documents.

Fortunately, user documents rarely change category codes (and certainly not of something so integral as escape characters!). So as long as we can accept that we will not be able to accurately parse every possible valid TEX document, we can use standard parsing techniques with static tokenisation rules, and still reasonably expect to parse most user documents.

2 The Overleaf LATEX parser

Overleaf's parser is developed using Lezer (`lezer.codemirror.net`), a parser generator and parsing library developed alongside CodeMirror 6, the text editor used by Overleaf. As Lezer is built for use in a code editor, it includes error recovery strategies. This is critical for a code editor where, during typing, a document will often be in an invalid state. For the Overleaf editor, it means that a missing $ is not going to highlight the entire document as math mode or, in the case of the Visual Editor, a typo is not going to incorrectly replace a large portion of your document with a WYSIWYG widget.

It is not feasible to present Overleaf's LATEX Lezer grammar in full,[1] so in the following section we will instead build a small grammar from scratch, that illustrates the core ideas of the grammar we use in practice.

2.1 A small Lezer LATEX grammar

Consider a simple LATEX document:

```
\documentclass{article}
\begin{document}
  Text
  \begin{fancytext}
    cool formatting
  \end{fancytext}
\end{document}
```

Identifying the smallest constituents of the document, we can produce a list like:

- Control sequences: `\documentclass`, `\begin`, `\end`
- Braces: `{`, `}`
- Words: `article`, `document`, `Text`, `fancytext`, `cool`, `formatting`
- Whitespace

Converting this into Lezer tokens looks like this:

```
@tokens {
  controlSequence { "\\" @asciiLetter+ }
  openBrace { "{" }
  closeBrace { "}" }
  word { @asciiLetter+ }
  space { @whitespace+ }
}
```

Building on top of this, we define *rules* to specify how multiple tokens interact. The parse starts from the rule marked `@top`, so this rule should describe an entire document. In our case, we say that a document contains *content*, and that content is made up from text, whitespace, or commands. Commands are further broken down in terms of tokens to be a control sequence followed by a number of arguments (which are themselves content, delimited by braces).

```
@top Document { Content }
Content { (Command | Text | space)* }
Command { controlSequence
          (openBrace Content closeBrace)* }
Text { word }
```

This small grammar is enough to accurately parse the small LATEX example, but results in a fairly flat structure as seen in the parse tree below:

```
Document {
  Content {
    Command { Content { Text } }
    Command { Content { Text } }
    Text
    Command { Content { Text } }
    Text
    Text
    Command { Content { Text } }
    Command { Content { Text } }
  }
}
```

[1] However, the interested reader can find the full grammar at `github.com/overleaf/overleaf/blob/c378f0961c92c6bcc2e7b13e442a1ccd22fb7211/services/web/frontend/js/features/source-editor/lezer-latex/latex.grammar`

So, let's introduce the concept of *specialisation*. This is how we introduce a special case into the grammar, in this example the concept of environments:

```
Content { (Command | Text
         | Environment | space)* }

Environment {
  @specialize<controlSequence, "\\begin">
    openBrace word closeBrace
      Content
  @specialize<controlSequence, "\\end">
    openBrace word closeBrace
}
```

We define an environment to start and end with specific control sequences (`\begin` and `\end`), contain a name enclosed by braces, and contain more content in its body. We also update the content rule to allow environments. Note the clear recursion here: `Content` can contain `Environment`s which in turn contains `Content`. This is what allows nested environments.

Parsing the document with the updated grammar will generate a parse tree with clearly nested environments, making the structure of the document more obvious.

```
Document {
  Content {
    Command { Content { Text } }
    Environment {
      Content {
        Text
        Environment {
          Content { Text Text }
        }
      }
    }
  }
}
```

The grammar used on Overleaf follows the same principle of understanding generic commands, with specialisations being used to provide a deeper understanding of specific commands, indicated by special control sequences. A few examples of specialisation in the real grammar are for `\cite` (for autocompletion), `\includegraphics` (for displaying figures in the Visual Editor), and `\textbf` (for supporting a text formatting toolbar).

2.2 Limitations

As already discussed, using static tokenisation rules for an inherently dynamic language, along with the approach of supporting all control sequences and adding specialisation for a select few, imposes certain assumptions on the user document. Consider a custom command like `\typesetasmath{\pi r^2}`. With a definition like `\def\typesetasmath#1{$#1$}` this would be perfectly valid TeX, but the parser would incorrectly assume `\pi r^2` to be text rather than math, due to the lack of enclosing `$...$`. Generally, any non-standard way of changing internal interpreter state can cause the parser to incorrectly parse the document. For commands provided by popular packages we can add specialisations to overcome this. An example of that is the `\text{}` command provided by the `amsmath` package which will typeset the argument in text mode, if used inside math mode.

3 Using the parser output

Using the parser outlined in the previous section, we can produce structural representations of arbitrary LaTeX documents, also known as *syntax trees*.

3.1 Simple usage

A few features follow immediately from having the syntax tree available: Syntax highlighting is as simple as assigning colours to the node types defined in the grammar. Contextual autocompletion follows from always being able to look up in the syntax tree what editing context the user is currently in. If the user is editing the argument to a citation command we can provide a citation key search, and if the user is inside the argument to a `\begin` command we can suggest environment names, and so forth.

3.2 WYSIWYG editing

More complex usages of the syntax tree arise from the Visual Editor. The basis of the Visual Editor is to replace sections of the document, identified by the syntax tree, with WYSIWYG representations of the content. Returning to our previous example of replacing `\textbf{bold text}` with **bold text**, the syntax tree would look approximately like this:

```
TextBoldCommand
  textBoldCtrlSeq
  Argument
    openBrace
      Text
    closeBrace
```

We can remove the `textBoldCtrlSeq`, `openBrace`, and `closeBrace` nodes from the document and mark the `Text` node as bold. In this simple case, mapping a change inside the element to the original source code is trivial (and in fact handled automatically by CodeMirror 6), but in more complex WYSIWYG elements this requires careful handling.

The most complex WYSIWYG tool we support is a table generator, as shown in Figure 1. In this

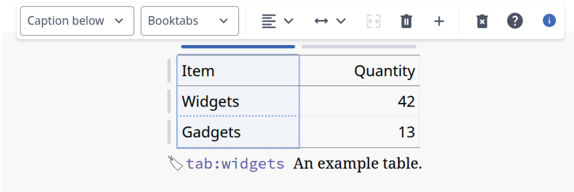

Figure 1: The table generator which can perform structural modifications, as well as changes to the content.

case we replace the entire `TabularEnvironment` node with a WYSIWYG editor that supports both content and structural changes like borders, columns, rows, etc. To map changes made in such WYSIWYG tools back to LaTeX code, we follow three guidelines: *Perform minimal edits*; *Write appropriate LaTeX*; and *If we don't understand it, we don't edit it.*

The first guideline ensures that we do not excessively modify user documents, and only do the minimally required modifications. An example of this is leaving whitespace exactly as-is when editing a table. We do not wish for the table generator to reformat hand-written code. This has the added benefit of only showing actual content changes when using the Tracked Changes feature.

The second guideline is about the kind of LaTeX we produce. We stick to only robust LaTeX code that we can confidently say will work. This means that we sometimes limit the functionality of our tools in order to only produce high-quality LaTeX.

Finally, the third guideline requires us to provide our tools conservatively. In the context of the table generator, if we cannot accurately understand the definition of a column we will fall back to displaying the source code rather than guess at any particular column type. This ensures that any operations we may do using the WYSIWYG tools are based on accurate understanding of the content.

4 Conclusion

In spite of the challenges with interpreting an inherently dynamic language like TeX, Overleaf uses the Lezer parser generator to produce a LaTeX parser that accurately parses most user documents, and drives many features across both available editors. The strategy of manually extending the grammar for common cases allows the parser to remain lightweight while still handling the vast majority of documents seen in practice. We have explored the foundations of building such a grammar, and outlined how it would be expanded to facilitate real-world applications. Finally we have detailed how the output of the parser, the syntax tree, can be used to produce WYSIWYG editing tools while remaining respectful of the already existing LaTeX code by performing the minimal changes necessary.

References

[1] B. Davies. Bumpy road towards a good LaTeX visual editor at Overleaf. *TUGboat* 44(2):255–255, 2023. `doi.org/10.47397/tb/44-2/tb137davies-visual`

[2] D.E. Knuth. *The TeXbook*. Computers & Typesetting. Addison Wesley, Boston, MA, Jan. 1984.

◇ Mathias Jakobsen
Overleaf
`mathias dot jakobsen (at) overleaf dot com`
ORCID 0000-0002-6128-7004