Markdown 2.10.0: LaTeX themes & snippets, two flavors of comments, and LuaMetaTeX

Vít Novotný

Abstract

Celebrating its fifth birthday, the Markdown package has received five new features: user-defined LaTeX themes, LaTeX setup snippets, semantic HTML comments, lexical TeX comments, and support for the LuaMetaTeX engine. In this article, we introduce each of these features and show how they can be used in practice. We also discuss five ideas for the future of Markdown and show how you can help turn them into a reality.

1 LaTeX themes & snippets

The goal of the Markdown package is simply to bring fire to the users of TeX, so that they can playfully incinerate each and every element of their markdown documents. The Markdown package does not aim to provide comprehensive defaults that would satisfy every kind of a document. Instead, all attention is directed towards making it easy for TeX programmers to style markdown. Although this goal fulfills the Unix philosophy of *doing one thing well*, the Markdown package would be well-served by encouraging users to lecture it on ever-new *things* and release their lecture notes to the whole world.

It is a paradox that one of the greatest successes of the LaTeX project may be its initial lack of features. Unlike in the cathedral of ConTeXt, where packages are few and most development is centralized, an extraordinary bazaar of action, ferment, and innovation has sprung up in the wake of the LaTeX 2ε kernel. To make it easier for Markdown users to share their lecture notes and combine them into thick tomes of tutelage, version 2.10.0 of the Markdown package has introduced *LaTeX themes & setup snippets*.

In Section 1.1, we introduce three example LaTeX themes that are included with the Markdown package. In Section 1.2, we show how users can create and use their own LaTeX themes. In Section 1.3, we introduce LaTeX setup snippets and show how users can create and use their own setup snippets.

1.1 Built-in themes

LaTeX themes are user-defined *building blocks* that

1. specify what markdown elements *do*,
2. can be shared and applied as a single unit, and
3. can be combined with one another to achieve high-level goals without low-level programming.

Since LaTeX is a Turing-complete language, what markdown elements *do* is not restricted to presentation, but includes computation and logic: We may typeset a table in various ways, but we may also store it as a matrix and compute its determinant, inverse, and eigenvalues, or we may apply it to an image as a linear transformation and display the result.

The Markdown package comes with three example LaTeX themes, listed by increasing complexity. These examples should help you develop an intuition for themes and what you can accomplish with them.

1.1.1 The `witiko/tilde` theme

The `witiko/tilde` theme redefines the tilde (`~`), so that it produces a non-breaking space:

```
\documentclass{article}
\usepackage[theme=witiko/tilde]{markdown}
\begin{document}
\begin{markdown}
Bartel~Leendert van~der~Waerden
\end{markdown}
\end{document}
```

The above code will produce the text “Bartel·Leendert van·der·Waerden”, where the middle dot (·) represents a non-breaking space.

1.1.2 The `witiko/dot` theme

The `witiko/dot` theme renders fenced code blocks with the `dot` infostring using Graphviz tools:

````
\documentclass{article}
\usepackage[theme=witiko/dot]{markdown}
\begin{document}
\begin{markdown}
```dot A parse tree of “Let's eat grandma!”
digraph tree {
    graph [margin = 0]; node [shape = none]
    edge  [arrowhead = none]
    {rank=same; S}
    {rank=same; VP1[label = VP]}
    {rank=same; Let
                NP1[label = NP]
                VP2[label = VP]}
    {rank=same; us; eat; NP2[label = NP]}
    {rank=same; grandma}
    S -> VP1; VP1 -> Let; VP1 -> NP1
    VP1 -> VP2; NP1 -> us; VP2 -> eat
    VP2 -> NP2; NP2 -> grandma }
\end{markdown}
\end{document}
````

The above code will produce Figure 1. The size of the graphics as well as other attributes can be controlled with the `\setkeys{Gin}{...}` command of the `graphicx` package. The placement of the figure can be controlled by redefining the `\fps@figure` LaTeX command.

doi.org/10.47397/tb/42-2/tb131novotny-markdown
````

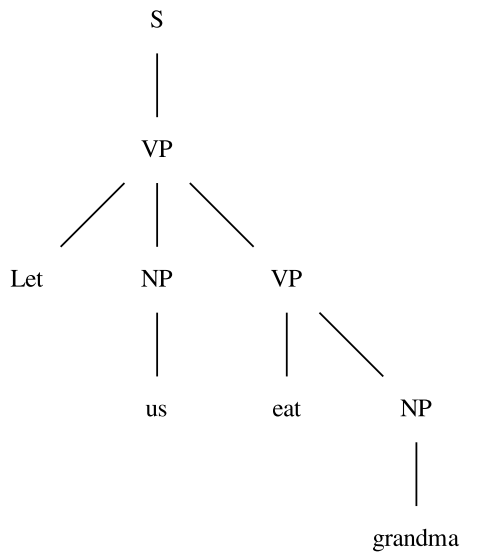

Figure 1: A parse tree of "Let's eat grandma!"

```
\documentclass{book}
\usepackage{markdown}
\markdownSetup{pipeTables,tableCaptions}
\begin{document}
\begin{markdown}
Introduction
============
## Section
### Subsection
Hello *Markdown*!

| Right | Left | Default | Center |
|------:|:-----|---------|:------:|
|   12  |  12  |    12   |    12  |
|  123  |  123 |   123   |   123  |
|    1  |    1 |     1   |     1  |

: Table
\end{markdown}
\end{document}
```

Chapter 1

Introduction

1.1 Section

1.1.1 Subsection

Hello *Markdown*!

Right	Left	Default	Center
12	12	12	12
123	123	123	123
1	1	1	1

Table 1.1: Table

Figure 2: The banner of the Markdown package

1.1.3 The `witiko/graphicx/http` theme

The `witiko/graphicx/http` theme downloads online images using either GNU Wget or cURL, whichever is available on your system, and displays them:

```
\documentclass{article}
\usepackage{markdown}
\markdownSetup{texComments, contentBlocks,
               theme=witiko/graphicx/http}
\begin{document}
\begin{markdown}
https://github.com/witiko/markdown/raw%
                  /main/banner.png
      (The banner of the Markdown package)
\end{markdown}
\end{document}
```

The above code will produce Figure 2 (grayscaled for *TUGboat*). As before, the size and placement of the figure can be controlled using the `\setkeys{Gin}{...}` and `\fps@figure` LaTeX commands.

1.2 Creating your own theme

To create your own LaTeX theme, you should decide on a name in the form ⟨*theme author*⟩`/`⟨*target package*⟩`/`⟨*private naming*⟩, where ⟨*theme author*⟩ specifies the provenance of the theme, ⟨*target package*⟩ specifies a LaTeX or software package that the theme relates to, and ⟨*private naming*⟩ specifies additional slash-delimited name segments. The ⟨*target package*⟩ and ⟨*private naming*⟩ name segments are optional, but at least one of them must be present.

Let us suppose that Jane Doe wishes to create a simple theme for the Beamer LaTeX package. Beamer creates presentation slides and Jane's theme will redefine markdown's first- and second-level headings to typeset the titles and subtitles of presentation slides. Therefore, Jane has decided to name her theme `jdoe/beamer/headings`.

Next, Jane will *munge* the name of the theme by substituting slashes (`/`) with underscores (`_`), and she will attach the prefix `markdowntheme` and the suffix `.sty` to arrive at the following filename:

```
markdownthemejdoe_beamer_headings.sty
```

Jane will create a text file with the above filename and the following content:

```
\ProvidesPackage{markdownthemejdoe_beamer_%
                 headings}[2021/06/04]
\markdownSetup{
    rendererPrototypes = {
        headingOne = {\frametitle{#1}},
        headingTwo = {\framesubtitle{#1}}
    }
}
```

Finally, Jane will use her new theme in her presentation slides, together with the `witiko/dot` theme, which she uses to typeset dietary assessments:

```
\documentclass[
    aspectratio=169
]{beamer}
\usepackage[
    theme = witiko/dot,
    theme = jdoe/beamer/headings
]{markdown}
\setkeys{Gin}{
    width=\columnwidth,
    keepaspectratio
}
\title{Dietary Assessment of Big Bad Wolf}
\author{Jane Doe}
\date{June 4, 2021}
\begin{document}
\maketitle
\begin{frame}[fragile]
\begin{markdown}
```

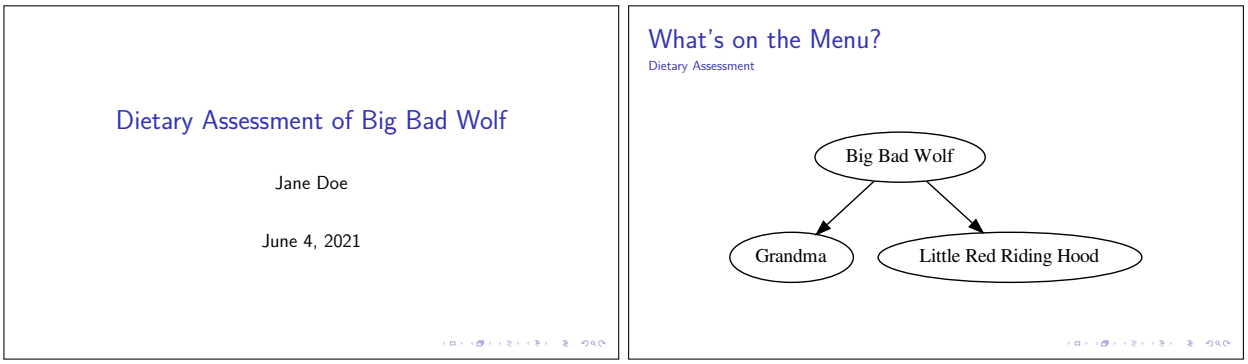

Figure 3: Presentation slides produced by Jane Doe using her `jdoe/beamer/headings` LaTeX theme

```
# What's on the Menu?
## Dietary Assessment
``` dot
digraph tree {
    Wolf -> Grandma
    Wolf -> Hood
    Wolf [label = "Big Bad Wolf"]
    Hood [label = "Little Red Riding Hood"]
}
\end{markdown}
\end{frame}
\end{document}
```

The above code will produce the two presentation slides shown in Figure 3. After adding a couple more features, Jane publishes her theme on CTAN, so that other authors can benefit from it.

1.3 Setup snippets

Let us suppose that Jane Doe has decided to create another theme named `jdoe/lists/roman`, which will make her ordered lists use Roman numerals:

```
\ProvidesPackage{markdownthemejdoe_lists_%
                 roman}[2021/06/04]
\markdownSetup{
    rendererPrototypes = {
        olItemWithNumber = {%
            \item[\romannumeral#1\relax.]%
        }
    }
}
```

Jane attempts to apply her theme in a local scope, displaying one of her ordered lists in Arabic numerals and another ordered list in Roman numerals:

```
\documentclass{article}
\usepackage{markdown}
\begin{document}
\begin{markdown}
1. wahid
2. aithnayn
\end{markdown}   % This won't work!
\begin{markdown*}{theme=jdoe/lists/roman}
3. tres
4. quattuor
\end{markdown*}
\end{document}
```

However, the above code will fail and produce the following LaTeX error: "Can be used only in preamble". LaTeX themes are full-featured LaTeX packages, which make permanent changes to a document, can only be loaded in the preamble of a document, and thus can't be applied in a local scope.

LaTeX setup snippets make it possible to *separate the cause from the effect*: We will load a LaTeX theme once in the preamble, the theme will define setup snippets, and we can apply the snippets in local scopes. Jane will first shorten her theme to `jdoe/lists`, making the `roman` segment into a snippet:

```
\ProvidesPackage{markdownthemejdoe_lists}%
                 [2021/06/04]
\markdownSetupSnippet{roman}{
    rendererPrototypes = {
        olItemWithNumber = {%
            \item[\romannumeral#1\relax.]%
        }
    }
}
```

Next, Jane will separate the loading of her `jdoe/lists` theme from using her `roman` setup snippet:

```
\documentclass{article}
\usepackage[theme=jdoe/lists]{markdown}
\begin{document}
\begin{markdown}
1. wahid
2. aithnayn
\end{markdown}
\begin{markdown*}{snippet=jdoe/lists/roman}
3. tres
4. quattuor
```
```

```
\end{markdown*}
\end{document}
```

The above code will produce the following list:

1. wahid
2. aithnayn

iii. tres

iv. quattuor

Notice how the setup snippet `roman` has been automatically *namespaced* to `jdoe/lists/roman`. This makes it less likely that different themes will define setup snippets with the same name. Snippets can also be defined outside of themes, in which case namespacing is not applied and `roman` stays `roman`.

2 Two flavors of comments

In TEX, comments fulfill several distinct roles:

1. We can use comments to prevent the processing of some parts of our code without deleting them:

   ```
   %\author{Authors anonymized for review}
   \author{John Doe \and Jane Roe}
   ```

2. We can use comments to write two parallel documents, a technique frequently used to produce documentation in literate programming [4]:

   ```
   % The \cs{foo} command prints ``bar'':
   %  \begin{macrocode}
   \def\foo{bar}
   %    \end{macrocode}
   ```

3. We can use comments to insert little side notes:

   ```
   % Aren't we missing a comma here?
   Let's eat grandma!
   ```

4. We can use comments to prevent TEX's input processor from inserting spaces or starting a new paragraph when word-wrapping newline characters are encountered:

   ```
   My parents have first met in Llanfairp%
   wllgwyngyllgogerychwyrndrobwllllantysi%
                                  liogogogoch.
   ```

The language of markdown started out as a preprocessor for the HTML language. As a consequence, markdown does not provide its own syntax for comments and authors are expected to use HTML comments instead:

```
<!-- Aren't we missing a comma here? -->
Let's eat grandma!
```

Unlike TEX's comments, which consume the rest of a line (including the occasional grandma), HTML comments consume just a part of a line, which increases their expressiveness at the expense of verbosity:

```
Let's <!-- eat --> visit grandma!
```

However, the markdown language only allows HTML comments in text, not in the middle of other elements, such as hyperlinks. This makes HTML comments unsuitable for general word-wrapping (point 4):

```
<!-- This won't work! -->
[1]: http://a.very.long.url/that/should<!--
   -->/enjoy%20some%20serious#word-wrapping
```

Although HTML comments can be extracted from an HTML document to create a parallel document for literate programming (point 2), no such option exists in the TEX Markdown package. As a consequence, users of the Markdown package will find HTML comments useful but often lacking compared to TEX comments.

In Section 2.1, we first show how version 2.10.0 of the Markdown package improves the support for HTML comments. In Section 2.2, we introduce a new flavor of markdown comments, which can be combined with HTML comments to cover all use cases of TEX comments and more.

2.1 Semantic HTML comments

Since version 2.3.0, the Markdown package has recognized HTML elements, entities, processing instructions, and comments when the `html` option is enabled. HTML entities are resolved, and HTML elements, processing instructions, and comments are omitted from the output:

```
\documentclass{article}
\usepackage[html]{markdown}
\begin{document}
\begin{markdown}

<!-- Aren't we missing *a comma* here? -->
Let's eat <emph>grandma</emph>&excl;

\end{markdown}
\end{document}
```

The above code will produce the text "Let's eat grandma!"

HTML comments are *semantic* in the sense that they are not stripped away by an input processor, but recognized as an element of the markdown language. Since version 2.10.0, the Markdown package includes a renderer that makes the text of the comments actionable:

```
\documentclass{article}
\usepackage{marginnote}
\usepackage[html]{markdown}
\markdownSetup{
  renderers = {
    inlineHtmlComment = {\marginnote{#1}},
  },
}
```

```
\begin{document}
\begin{markdown}
<!-- Aren't we missing *a comma* here? -->
Let's eat grandma!
\end{markdown}
\end{document}
```

The above code will produce the text "Let's eat grandma!" with the comment displayed as a margin note as we see here. This makes HTML comments useful for inserting notes (point 3).

Aren't we missing *a comma* here?

Notice that the inline markdown markup for emphasis is recognized as well. This support for nested formatting makes HTML comments useful for writing parallel documents (point 2).

2.2 Lexical TEX comments

The Markdown package has always supported the `hybrid` option, which allows users to use LATEX commands such as `\label` and `\ref` inside markdown:

```
\documentclass{article}
\usepackage[hybrid]{markdown}
\begin{document}
\begin{markdown}
I conclude in Section~\ref{sec:conclusion}.

Conclusion
==========
\label{sec:conclusion}
In this paper, we have discovered that most
grandmas would rather eat dinner with their
grandchildren than get eaten. Begone, wolf!
\end{markdown}
\end{document}
```

However, since the conversion of markdown to TEX does not preserve newlines, using TEX comments in the `hybrid` mode will lead to unexpected results. For example, typesetting the markdown document from point 4 on page 189 in the `hybrid` mode will produce the text "My parents have first met in Llanfairp".

Since version 2.6.0, the Markdown package has supported the `stripPercentSigns` option, which makes it possible to use TEX comments to produce documentation in literate programming (point 2). [5]

Since version 2.10.0, the Markdown package sets the category code of the percent sign to *other* when typesetting markdown documents, so that TEX comments can't produce malformed documents in the `hybrid` mode. Additionally, a *lexical* input processor that recognizes the regular language of TEX comments (for technical details, see Figure 4) has been added to the Markdown package and can be enabled with the `texComments` option. For example, typesetting the markdown document from point 4 on page 189 with the `texComments` option enabled will produce the expected text "My parents have first met in Llanfairpwllgwyngyllgogerychwyrndrobwllllantysiliogogogoch."

TEX comments are *lexical* in the sense that they are unaware of markdown. Therefore, we can use them for general word-wrapping (point 4):

```
[1]: http://a.very.long.url/that/should/%
     enjoy\%20some\%20serious#word-wrapping
```

In conclusion, both the semantic HTML comments and the lexical TEX comments are well-suited for preventing the processing of some parts of our documents (point 1) and for writing parallel documents (point 2). Whereas HTML comments are better-suited for writing and optionally typesetting little side notes (point 3), TEX comments can be used for word-wrapping anywhere in the text (point 4).

3 LuaMetaTEX

The LuaMetaTEX engine is a minimalist development version of LuaTEX, which removes many features of LuaTEX and alters some interfaces. LuaMetaTEX is used in the ConTEXt LMTX format [1], which may soon make its way into TEX Live and other TEX distributions. Since the Markdown package supports ConTEXt, the time is ripe to make Markdown play nice with LuaMetaTEX as well.

LuaTEX contains the Selene Unicode library, which the Markdown package uses to transform Unicode text. LuaMetaTEX removes Selene Unicode, but uses Lua 5.4, which contains a built-in `utf8` library. The `utf8` library has a similar interface and functionality to Selene Unicode. Since version 2.10.0, the Markdown package will use either Selene Unicode or `utf8`, whichever is available in Lua.

LuaTEX contains the `kpathsea` library, which is responsible for finding files in the TEX directory structure. The Markdown package uses `kpathsea` to find JSON files that map filename extensions to names of programming languages for the `contentBlocks` syntax extension. LuaMetaTEX removes the `kpathsea` library, but provides an optional library interface, which allows the use of an external `kpathsea` library when it is available. Since version 2.10.0, the Markdown package will search for files only in the current working directory when `kpathsea` is unavailable. This should only affect ConTEXt Standalone: in full TEX distributions, where an external `kpathsea` library is available, there should be no change of behavior.

4 What's next and how do I contribute?

There are many intriguing ideas for the future of Markdown. Some of these ideas are already under

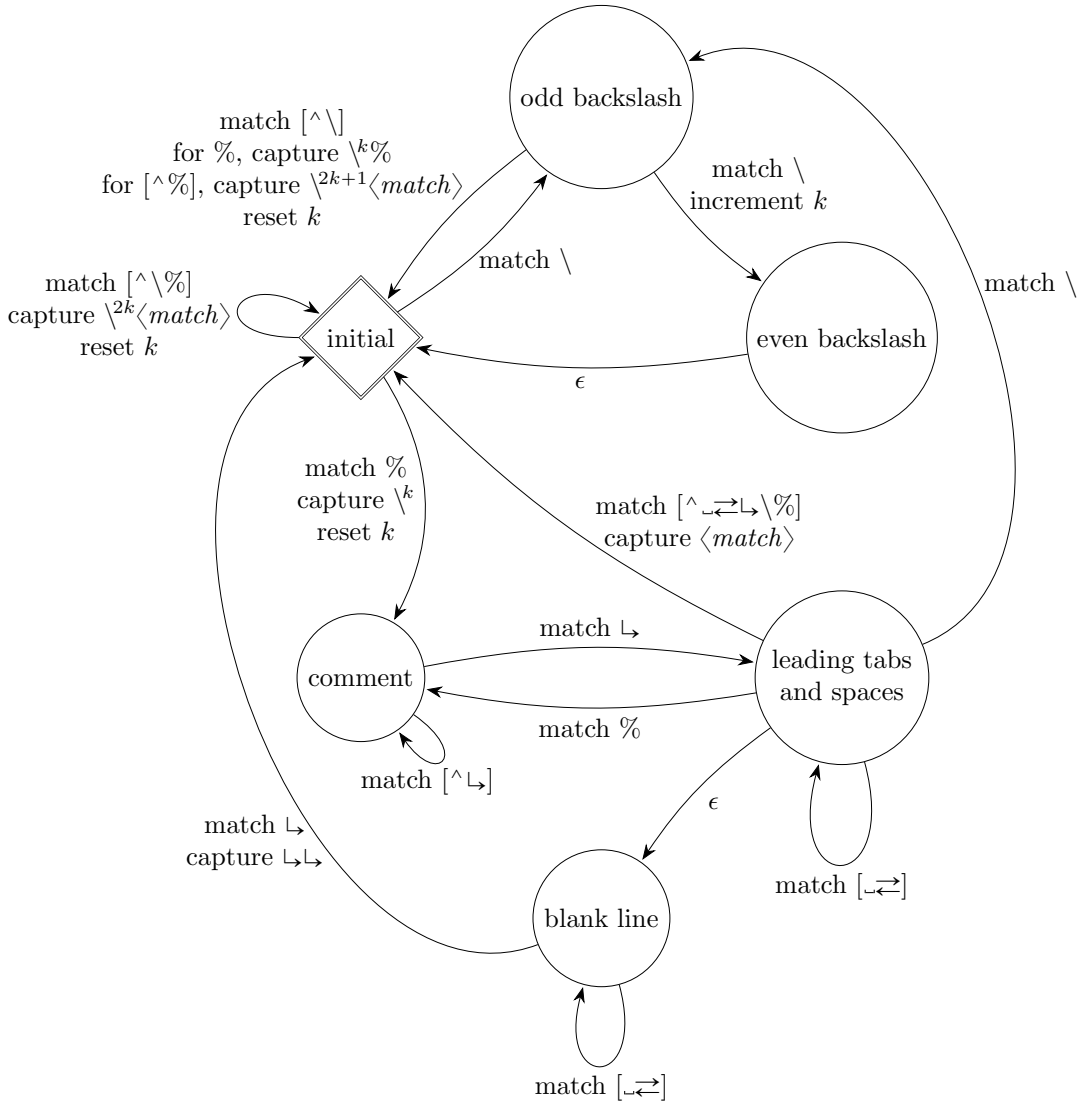

Figure 4: An automaton that strips TEX comments from markdown input with the `texComments` option.

The automaton contains a counter k, which is initially zero. The automaton reads and *matches* characters from the input to transition between *states*. During a state transition, the automaton may *capture* character strings by writing them to the output, *increment* counter k by one, or *reset* counter k back to zero.

Until the automaton encounters a backslash (\) or a percent sign (%), it stays in the **initial** state, which is similar to the state M of TEX's input processor [2, Chapter 8], capturing every character that it matches.

When the automaton encounters a sequence of backslashes, it counts the pairs of backslashes in counter k. If a percent sign follows an **odd backslash**, the percent sign has been *escaped* and the automaton captures k backslashes and the percent sign: This makes capture a surjective function, allowing us to write `\%` as `\\\%`. If a character other than a percent sign or a backslash follows an **odd** or **even backslash**, the automaton captures $2k + 1$ or $2k$ backslashes, respectively, and the matched character: This makes capture an identity function for inputs that do not contain a percent sign.

When the automaton encounters a percent sign that has not been escaped, the automaton captures k backslashes and reads the rest of the line as a **comment** without capturing any characters. After reading the newline character (↳), the automaton enters a state similar to the state N of TEX's input processor, reading any **leading tabs and spaces** (⇄ and ␣) without capturing any characters. If the automaton encounters additional newline characters, there has been a **blank line** in the input and the automaton captures two newline characters (↳↳) before it transitions back to the **initial** state.

development by contributors and soon to become the present reality, whereas some other ideas are only now beginning to be discussed,[1] and others yet are waiting to be discovered by you.

For your inspiration, I list some existing ideas for improving Markdown by increasing complexity and suggest how you can contribute:

4.1 Actionable HTML attributes

In my previous article [5, Section 2.4], I introduced HTML attributes as a way of typesetting only small parts of markdown documents. However, the HTML attributes are currently not *actionable*, which means that users can't react to them from TEX.

If the HTML element identifiers were actionable, we could rewrite the code from Section 2.2 without the `hybrid` mode and all its security problems. Additionally, if HTML class names were actionable, we could apply setup snippets without switching between markdown and LATEX:

```
I conclude in Section <#sec:conclusion>.

Conclusion {#sec:conclusion .some-snippet}
==========
In this paper, we have discovered that most
grandmas would rather eat dinner with their
grandchildren than get eaten. Begone, wolf!
```

Future development should add syntax extensions such as Pandoc's `fenced_divs`, `bracketed_spans`, and `inline_code_attributes` for specifying HTML attributes on elements other than headings.

If you would like to contribute, you should have a look at issue 91[2] and the Contributing section of the `README.md` document.[3] The introductory article by Henri Menke [3] about writing parsing expression grammars (PEG) in the Lua LPeg library is recommended reading.

4.2 Jekyll front matter

Jekyll is a static site generator that takes Markdown documents and converts them to a website. In Jekyll, each Markdown document can start with *front matter*: a block in your amazing markup language (YAML) that can specify various metadata:

```
---
title: Of *Wolves* and _Grandmas_
author:
- name: Little Red Riding Hood
- name: Big Bad Wolf
---
```

If Jekyll's front matter were supported in Markdown, we could set up all metadata of a document from Markdown without ever switching to TEX.

If you would like to contribute, you should have a look at issue 22[4] and the implementation drafted by Marei Peischl in pull request 77.[5] The article by Henri Menke [3] about writing PEG in the Lua LPeg library is again recommended reading.

4.3 The `witiko/graphicx/http` theme in Lua

The `witiko/graphicx/http` LATEX theme from Section 1.1.3 requires either GNU Wget or cURL to download online images. We could remove both prerequisites by using the `socket.http` Lua library.

In issue 82,[6] I drafted an implementation and listed several issues that prevent its use:

1. The `http.request` method mishandles redirects.
2. LuaMetaTEX lacks the `socket.http` library.
3. The `\directlua` command needs to be replaced with a shell escape for non-Lua TEX engines.

Lua programmers familiar with the Luasocket library are encouraged to help tackle points 1 and 2.

4.4 Integration with Pandoc

Pandoc is a Haskell library for converting between dozens of document formats. Since it would be difficult to write conversion functions for every pair of formats, Pandoc uses an intermediate abstract syntax tree (AST), so that every document format only needs a conversion function from the document format to the AST and back. If the Markdown package understood the AST, we could typeset any of the document formats of Pandoc while maintaining full control over the formatting:

```
\documentclass{article}
\usepackage[theme=jdoe/lists]{markdown}
\begin{document}
\pandocInput[snippet=jdoe/lists/roman]
            {of-wolves-and-grandmas.docx}
\end{document}
```

If you would like to contribute, you should have a look at the corresponding issues[7] and the GitHub repository of Dominik Rehák,[8] who is extending the Lunamark Lua parser with an AST reader. Ideas on how to best integrate the AST reader into the interface of Markdown will be appreciated.

[1] `github.com/witiko/markdown/issues` & `/discussions`
[2] `github.com/witiko/markdown/issues/91`
[3] `github.com/witiko/markdown#contributing`
[4] `github.com/witiko/markdown/issues/22`
[5] `github.com/witiko/markdown/pull/77`
[6] `github.com/witiko/markdown/issues/82`
[7] `github.com/witiko/markdown/issues/25` & `/62`
[8] `github.com/drehak/lunamark`

4.5 Direct mapping of elements

In Section 1.2, Jane Doe has created a `jdoe/beamer/headings` LaTeX theme for producing presentation slides. However, we still needed to use the LaTeX `frame` environment for each presentation slide. Could we produce presentation slides without switching between markdown and LaTeX?

The Markdown package relies on TeX's *expansion processor*: For example, the Lua parser converts the markdown text "`# What's on the Menu?`" into the TeX code

```
\markdownRendererHeadingOne{What's on the Menu?}
```

which TeX then *expands* to

```
\frametitle{What's on the Menu?}
```

and typesets. However, we can't always rely on TeX's expansion processor. For example, the Beamer command `\begin{frame}` will read input until it has found a matching `\end{frame}` command. If `\end{frame}` is hidden behind expansion, it will never be found.

One solution would be to make the conversion of "`# What's on the Menu?`" configurable, so that instead of producing `\markdownRendererHeadingOne`, it can *map directly* to "`\begin{frame}{What's on the Menu?}`". If you would like to contribute, you should have a look at issue 92[9] and the Contributing section of the `README.md` document.[10]

On a more decentralized level: play with the Markdown package, cherish it, use it in your writing, and find ways to abuse its syntax in unexpected and unsettling ways. LaTeX themes make it easier than ever to share your discoveries and compose them into a beautiful cacophony of mayhem.

[9] `github.com/witiko/markdown/issues/92`
[10] `github.com/witiko/markdown#contributing`

Figure 5: Grandma Jane and the Big Bad Wolf celebrate the fifth birthday of the Markdown package. Illustration by `fiverr.com/quickcartoon`.

Book announcement

Lloyd R. Prentice and I are writing a book: *Publish Beautiful Books with Markdown: Fast Track to LaTeX* (`publishbeautifulbooks.com`) is about book design, typography, and technology that allows you to go from well-chosen words to a beautiful book with just a few keystrokes.

Lloyd is a novelist and indie book publisher. His novels include *Freein' Pancho* and *The Gospel of Ashes*. He wrote and produced the three-year running web manga *Aya Takeo* with illustrator Sonia Leong; print version published in three volumes by UK independent publisher and comic collaborative Sweatdrop Studios. Lloyd also co-wrote and published the technical programming book *Build It with Nitrogen: The Fast-Off-the-Block Erlang Web Framework*. In an earlier life, Lloyd designed and developed nearly 100 consumer and educational software products for major US publishers. Web experience includes design and development of a soup-to-nuts application to support marketing and management of world-class technical conferences.

I am a computer scientist, university teacher, and digital typography enthusiast. Before I developed the Markdown package, I had prepared a dozen TeX document templates for universities, scientific journals, and small businesses. I am the technical editor of the Czechoslovak TeX users group ($\mathcal{CS}$TUG) Bulletin, which publishes research works on electronic document preparation and digital typography.

References

[1] H. Hagen. ConTeXt LMTX. *TUGboat* 40(1):34–37, 2019. `https://tug.org/TUGboat/tb40-1/tb124hagen-lmtx.pdf`

[2] D.E. Knuth. *The TeXbook*. Addison-Wesley, 1986.

[3] H. Menke. Parsing complex data formats in LuaTeX with LPEG. *TUGboat* 40(2):129–135, 2019. `https://tug.org/TUGboat/tb40-2/tb125menke-lpeg.pdf`

[4] F. Mittelbach. Format LaTeX documentation, 2021. `https://ctan.org/pkg/doc`

[5] V. Novotný. Markdown 2.7.0: Towards lightweight markup in TeX. *TUGboat* 40(1):25–27, 2019. `https://tug.org/TUGboat/tb40-1/tb124novotny-markdown.pdf`

◇ Vít Novotný
Studená 453/15
Brno, 638 00
Czech Republic
`witiko (at) mail dot muni dot cz`
`github.com/witiko`