A Markdown Interpreter for TeX

Vít Novotný (based on the work of John MacFarlane and Hans Hagen)
witiko@mail.muni.cz

Version 1.0.2
August 15, 2016

Contents

1 Introduction

This document is a reference manual for the Markdown package. It is split into three sections. This section explains the purpose and the background of the package and outlines its prerequisites. Section 2 describes the interfaces exposed by the package along with usage notes and examples. It is aimed at the user of the package. Section 3 describes the implementation of the package. It is aimed at the developer of the package and the curious user.

1.1 About Markdown

The Markdown package provides facilities for the conversion of markdown markup to plain TeX. These are provided both in the form of a Lua module and in the form of plain TeX, LaTeX, and ConTeXt macro packages that enable the direct inclusion of markdown documents inside TeX documents.

Architecturally, the package consists of the Lunamark v1.4.0 Lua module by John MacFarlane, which was slimmed down and rewritten for the needs of the package. On top of Lunamark sits code for the plain TeX, LaTeX, and ConTeXt formats by Vít Novotný.

```
1 if not modules then modules = { } end modules ['markdown'] = {
2     version   = "1.0.2",
3     comment   = "A module for the conversion from markdown to plain TeX",
4     author    = "John MacFarlane, Hans Hagen, Vít Novotný",
```

```
5     copyright = "2009-2016 John MacFarlane, Hans Hagen; 2016 Vít Novotný",
6     license   = "LPPL 1.3"
7 }
```

1.2 Feedback

Please use the markdown project page on GitHub[1] to report bugs and submit feature requests. Before making a feature request, please ensure that you have thoroughly studied this manual. If you do not want to report a bug or request a feature but are simply in need of assistance, you might want to consider posting your question on the TEX-LATEX Stack Exchange[2].

1.3 Acknowledgements

I would like to thank the Faculty of Informatics at the Masaryk University in Brno for providing me with the opportunity to work on this package alongside my studies. I would also like to thank the creator of the Lunamark Lua module, John Macfarlane, for releasing Lunamark under a permissive license that enabled its inclusion into the package.

The TEX part of the package draws inspiration from several sources including the source code of LATEX 2$_\varepsilon$, the minted package by Geoffrey M. Poore – which likewise tackles the issue of interfacing with an external interpreter from TEX, the filecontents package by Scott Pakin, and others.

1.4 Prerequisites

This section gives an overview of all resources required by the package.

1.4.1 Lua Prerequisites

The Lua part of the package requires the following Lua modules:

LPeg A pattern-matching library for the writing of recursive descent parsers via the Parsing Expression Grammars (PEGs). It is used by the Lunamark library to parse the markdown input.

```
8     local lpeg = require("lpeg")
```

Selene Unicode A library that provides support for the processing of wide strings. It is used by the Lunamark library to cast image, link, and footnote tags to the lower case.

```
9     local unicode = require("unicode")
```

[1] https://github.com/witiko/markdown/issues
[2] https://tex.stackexchange.com

MD5 A library that provides MD5 crypto functions. It is used by the Lunamark library to compute the digest of the input for caching purposes.

```
10      local md5 = require("md5")
```

All the abovelisted modules are statically linked into the LuaTEX engine (see [1, Section 3.3]).

1.4.2 Plain TEX Prerequisites

The plain TEX part of the package requires the following Lua module:

Lua File System A library that provides access to the filesystem via OS-specific syscalls. It is used by the plain TEX code to create the cache directory specified by the `\markdownOptionCacheDir` macro before interfacing with the Lunamark library.

The plain TEX code makes use of the `isdir` method that was added to the module by the LuaTEX engine developers (see [1, Section 3.2]). This method is not present in the base library.

The Lua File System module is statically linked into the LuaTEX engine (see [1, Section 3.3]).

The plain TEX part of the package also requires that the plain TEX format (or its superset) is loaded and that either the LuaTEX `\directlua` primitive or the shell access file stream 18 is available.

1.4.3 LATEX Prerequisites

The LATEX part of the package requires that the LATEX 2_ε format is loaded and also, since it uses the plain TEX implementation, all the plain TEX prerequisites (see Section 1.4.2).

```
11 \NeedsTeXFormat{LaTeX2e}%
```

The following LATEX 2_ε packages are also required:

keyval A package that enables the creation of parameter sets. This package is used to provide the `\markdownSetup` macro, the package options processing, as well as the parameters of the `markdown*` LATEX environment.

url A package that provides the `\url` macro for the typesetting of URLs. It is used to provide the default token renderer prototype (see Section 2.2.4) for links.

graphicx A package that provides the `\includegraphics` macro for the typesetting of images. It is used to provide the corresponding default token renderer prototype (see Section 2.2.4).

paralist A package that provides the `compactitem`, `compactenum`, and `compactdesc` macros for the typesetting of tight bulleted lists, ordered lists, and definition lists. It is used to provide the corresponding default token renderer prototypes (see Section 2.2.4).

ifthen A package that provides a concise syntax for the inspection of macro values. It is used to determine whether or not the paralist package should be loaded based on the user options.

fancyvrb A package that provides the `\VerbatimInput` macros for the verbatim inclusion of files containing code. It is used to provide the corresponding default token renderer prototype (see Section 2.2.4).

1.4.4 ConTEXt prerequisites

The ConTEXt part of the package requires that either the Mark II or the Mark IV format is loaded and also, since it uses the plain TEX implementation, all the plain TEX prerequisites (see Section 1.4.2).

2 User Guide

This part of the manual describes the interfaces exposed by the package along with usage notes and examples. It is aimed at the user of the package.

Since neither TEX nor Lua provide interfaces as a language construct, the separation to interfaces and implementations is purely abstract. It serves as a means of structuring this manual and as a promise to the user that if they only access the package through the interfaces, the future versions of the package should remain backwards compatible.

2.1 Lua Interface

The Lua interface provides the conversion from markdown to plain TEX. This interface is used by the plain TEX implementation (see Section 3.2) and will be of interest to the developers of other packages and Lua modules.

The Lua interface is implemented by the `markdown` Lua module.

```
12 local M = {}
```

2.1.1 Conversion from Markdown to Plain TEX

The Lua interface exposes the `new(options)` method. This method creates converter functions that perform the conversion from markdown to plain TEX according to the table `options` that contains options recognized by the Lua interface. (see Section

2.1.2). The `options` parameter is optional; when unspecified, the behaviour will be the same as if `options` were an empty table.

The following example Lua code converts the markdown string `_Hello world!_` to a TEX output using the default options and prints the TEX output:

```
local md = require("markdown")
local convert = md.new()
print(convert("_Hello world!_"))
```

2.1.2 Options

The Lua interface recognizes the following options. When unspecified, the value of a key is taken from the `defaultOptions` table.

```
13 local defaultOptions = {}
```

`blankBeforeBlockquote`=`true`, `false` default: `false`

`true` Require a blank line between a paragraph and the following blockquote.

`false` Do not require a blank line between a paragraph and the following blockquote.

```
14 defaultOptions.blankBeforeBlockquote = false
```

`blankBeforeHeading`=`true`, `false` default: `false`

`true` Require a blank line between a paragraph and the following header.

`false` Do not require a blank line between a paragraph and the following header.

```
15 defaultOptions.blankBeforeHeading = false
```

`cacheDir`=⟨*directory*⟩ default: .

The path to the directory containing auxiliary cache files.

When iteratively writing and typesetting a markdown document, the cache files are going to accumulate over time. You are advised to clean the cache directory every now and then, or to set it to a temporary filesystem (such as `/tmp` on UN*X systems), which gets periodically emptied.

```
16 defaultOptions.cacheDir = "."
```

`definitionLists`=`true`, `false` default: `false`

`true` Enable the pandoc definition list syntax extension:

```
Term 1

:   Definition 1

Term 2 with *inline markup*

:   Definition 2

        { some code, part of Definition 2 }

    Third paragraph of definition 2.
```

`false` Disable the pandoc definition list syntax extension.

```
17 defaultOptions.definitionLists = false
```

`hashEnumerators`=`true`, `false` default: `false`

`true` Enable the use of hash symbols (`#`) as ordered item list markers.

`false` Disable the use of hash symbols (`#`) as ordered item list markers.

```
18 defaultOptions.hashEnumerators = false
```

`hybrid`=`true`, `false` default: `false`

`true` Disable the escaping of special plain TEX characters, which makes it possible to intersperse your markdown markup with TEX code. The intended usage is in documents prepared manually by a human author. In such documents, it can often be desirable to mix TEX and markdown markup freely.

`false` Enable the escaping of special plain TEX characters outside verbatim environments, so that they are not interpretted by TEX. This is encouraged when typesetting automatically generated content or markdown documents that were not prepared with this package in mind.

```
19 defaultOptions.hybrid = false
```

`footnotes`=`true`, `false` default: `false`

`true` Enable the pandoc footnote syntax extension:

```
Here is a footnote reference,[^1] and another.[^longnote]

[^1]: Here is the footnote.

[^longnote]: Here's one with multiple blocks.

    Subsequent paragraphs are indented to show that they
belong to the previous footnote.

        { some.code }

    The whole paragraph can be indented, or just the
    first line.  In this way, multi-paragraph footnotes
    work like multi-paragraph list items.

This paragraph won't be part of the note, because it
isn't indented.
```

`false` Disable the pandoc footnote syntax extension.

```
20 defaultOptions.footnotes = false
```

`preserveTabs`=`true`, `false` default: `false`

`true` Preserve all tabs in the input.

`false` Convert any tabs in the input to spaces.

```
21 defaultOptions.preserveTabs = false
```

`smartEllipses`=`true`, `false` default: `false`

`true` Convert any ellipses in the input to the `\markdownRendererEllipsis` TEX macro.

`false` Preserve all ellipses in the input.

```
22 defaultOptions.smartEllipses = false
```

`startNumber`=`true`, `false` default: `true`

`true` Make the number in the first item in ordered lists significant. The item numbers will be passed to the `\markdownRendererOlItemWithNumber` TEX macro.

`false` Ignore the number in the items of ordered lists. Each item will only produce a `\markdownRendererOlItem` TEX macro.

```
23 defaultOptions.startNumber = true
```

`tightLists`=`true`, `false` default: `true`

`true` Lists whose bullets do not consist of multiple paragraphs will be detected and passed to the `\markdownRendererOlBeginTight`, `\markdownRendererOlEndTight`, `\markdownRendererUlBeginTight`, `\markdownRendererUlEndTight`, `\markdownRendererDlBeginTight`, and `\markdownRendererDlEndTight` macros.

`false` Lists whose bullets do not consist of multiple paragraphs will be treated the same way as lists that do.

```
24 defaultOptions.tightLists = true
```

2.2 Plain TEX Interface

The plain TEX interface provides macros for the typesetting of markdown input from within plain TEX, for setting the Lua interface options (see Section 2.1.2) used during the conversion from markdown to plain TEX, and for changing the way markdown the tokens are rendered.

```
25 \def\markdownLastModified{2016/08/14}%
26 \def\markdownVersion{1.0.2}%
```

The plain TEX interface is implemented by the `markdown.tex` file that can be loaded as follows:

```
\input markdown
```

It is expected that the special plain TEX characters have the expected category codes, when `\input`ting the file.

2.2.1 Typesetting Markdown

The interface exposes the `\markdownBegin`, `\markdownEnd`, and `\markdownInput` macros.

The `\markdownBegin` macro marks the beginning of a markdown document fragment and the `\markdownEnd` macro marks its end.

```
27 \let\markdownBegin\relax
28 \let\markdownEnd\relax
```

You may prepend your own code to the \markdownBegin macro and redefine the \markdownEnd macro to produce special effects before and after the markdown block.

There are several limitations to the macros you need to be aware of. The first limitation concerns the \markdownEnd macro, which must be visible directly from the input line buffer (it may not be produced as a result of input expansion). Otherwise, it will not be recognized as the end of the markdown string otherwise. As a corrolary, the \markdownEnd string may not appear anywhere inside the markdown input.

Another limitation concerns spaces at the right end of an input line. In markdown, these are used to produce a forced line break. However, any such spaces are removed before the lines enter the input buffer of TEX (see [2, p. 46]). As a corrolary, the \markdownBegin macro also ignores them.

The \markdownBegin and \markdownEnd macros will also consume the rest of the lines at which they appear. In the following example plain TEX code, the characters c, e, and f will not appear in the output.

```
\input markdown
a
b \markdownBegin c
d
e \markdownEnd   f
g
\bye
```

Note that you may also not nest the \markdownBegin and \markdownEnd macros.

The following example plain TEX code showcases the usage of the \markdownBegin and \markdownEnd macros:

```
\input markdown
\markdownBegin
_Hello_ **world** ...
\markdownEnd
\bye
```

The \markdownInput macro accepts a single parameter containing the filename of a markdown document and expands to the result of the conversion of the input markdown document to plain TEX.

```
29 \let\markdownInput\relax
```

This macro is not subject to the abovelisted limitations of the \markdownBegin and \markdownEnd macros.

The following example plain TEX code showcases the usage of the \markdownInput macro:

```
\input markdown
\markdownInput{hello.md}
\bye
```

2.2.2 Options

The plain TEX options are represented by TEX macros. Some of them map directly to the options recognized by the Lua interface (see Section 2.1.2), while some of them are specific to the plain TEX interface.

2.2.2.1 File and directory names

The `\markdownOptionHelperScriptFileName` macro sets the filename of the helper Lua script file that is created during the conversion from markdown to plain TEX in TEX engines without the `\directlua` primitive. It defaults to `\jobname.markdown.lua`, where `\jobname` is the base name of the document being typeset.

The expansion of this macro must not contain quotation marks (`"`) or backslash symbols (`\`). Mind that TEX engines tend to put quotation marks around `\jobname`, when it contains spaces.

```
30 \def\markdownOptionHelperScriptFileName{\jobname.markdown.lua}%
```

The `\markdownOptionInputTempFileName` macro sets the filename of the temporary input file that is created during the conversion from markdown to plain TEX in TEX engines without the `\directlua` primitive. It defaults to `\jobname.markdown.out`. The same limitations as in the case of the `\markdownOptionHelperScriptFileName` macro apply here.

```
31 \def\markdownOptionInputTempFileName{\jobname.markdown.in}%
```

The `\markdownOptionOutputTempFileName` macro sets the filename of the temporary output file that is created during the conversion from markdown to plain TEX in TEX engines without the `\directlua` primitive. It defaults to `\jobname.markdown.out`. The same limitations apply here as in the case of the `\markdownOptionHelperScriptFileName` macro.

```
32 \def\markdownOptionOutputTempFileName{\jobname.markdown.out}%
```

The `\markdownOptionCacheDir` macro corresponds to the Lua interface `cacheDir` option that sets the name of the directory that will contain the produced cache files. The option defaults to `_markdown-\jobname`, which is a similar naming scheme to the one used by the minted LATEX package. The same limitations apply here as in the case of the `\markdownOptionHelperScriptFileName` macro.

```
33 \def\markdownOptionCacheDir{_markdown-\jobname}%
```

2.2.2.2 Lua Interface Options

The following macros map directly to the options recognized by the Lua interface (see Section 2.1.2) and are not processed by the plain TEX implementation, only passed along to Lua. They are undefined, which makes them fall back to the default values provided by the Lua interface.

```
\let\markdownOptionBlankBeforeBlockquote\undefined
\let\markdownOptionBlankBeforeHeading\undefined
\let\markdownOptionDefinitionLists\undefined
\let\markdownOptionHashEnumerators\undefined
\let\markdownOptionHybrid\undefined
\let\markdownOptionFootnotes\undefined
\let\markdownOptionPreserveTabs\undefined
\let\markdownOptionSmartEllipses\undefined
\let\markdownOptionStartNumber\undefined
\let\markdownOptionTightLists\undefined
```

2.2.3 Token Renderers

The following TEX macros may occur inside the output of the converter functions exposed by the Lua interface (see Section 2.1.1) and represent the parsed markdown tokens. These macros are intended to be redefined by the user who is typesetting a document. By default, they point to the corresponding prototypes (see Section 2.2.4).

2.2.3.1 Line Break Renderer

The `\markdownRendererLineBreak` macro represents a forced line break. The macro receives no arguments.

```
\def\markdownRendererLineBreak{%
  \markdownRendererLineBreakPrototype}%
```

2.2.3.2 Ellipsis Renderer

The `\markdownRendererEllipsis` macro replaces any occurance of ASCII ellipses in the input text. This macro will only be produced, when the `smartEllipses` option is `true`. The macro receives no arguments.

```
\def\markdownRendererEllipsis{%
  \markdownRendererEllipsisPrototype}%
```

2.2.3.3 Special Character Renderers

The following macros replace any special plain TEX characters (including the active pipe character (`|`) of ConTEXt) in the input text. These macros will only be produced, when the `hybrid` option is `false`.

```
\def\markdownRendererLeftBrace{%
  \markdownRendererLeftBracePrototype}%
\def\markdownRendererRightBrace{%
  \markdownRendererRightBracePrototype}%
\def\markdownRendererDollarSign{%
  \markdownRendererDollarSignPrototype}%
```

```
\def\markdownRendererPercentSign{%
  \markdownRendererPercentSignPrototype}%
\def\markdownRendererAmpersand{%
  \markdownRendererAmpersandPrototype}%
\def\markdownRendererUnderscore{%
  \markdownRendererUnderscorePrototype}%
\def\markdownRendererHash{%
  \markdownRendererHashPrototype}%
\def\markdownRendererCircumflex{%
  \markdownRendererCircumflexPrototype}%
\def\markdownRendererBackslash{%
  \markdownRendererBackslashPrototype}%
\def\markdownRendererTilde{%
  \markdownRendererTildePrototype}%
\def\markdownRendererPipe{%
  \markdownRendererPipePrototype}%
```

2.2.3.4 Code Span Renderer

The `\markdownRendererCodeSpan` macro represents inlined code span in the input text. It receives a single argument that corresponds to the inlined code span.

```
\def\markdownRendererCodeSpan{%
  \markdownRendererCodeSpanPrototype}%
```

2.2.3.5 Link Renderer

The `\markdownRendererLink` macro represents a hyperlink. It receives four arguments: the label, the fully escaped URI that can be directly typeset, the raw URI that can be used outside typesetting, and the title of the link.

```
\def\markdownRendererLink{%
  \markdownRendererLinkPrototype}%
```

2.2.3.6 Image Renderer

The `\markdownRendererImage` macro represents an image. It receives four four arguments: the label, the fully escaped URI that can be directly typeset, the raw URI that can be used outside typesetting, and the title of the link.

```
\def\markdownRendererImage{%
  \markdownRendererImagePrototype}%
```

2.2.3.7 Bullet List Renderers

The `\markdownRendererUlBegin` macro represents the beginning of a bulleted list that contains an item with several paragraphs of text (the list is not tight). The macro receives no arguments.

```
\def\markdownRendererUlBegin{%
  \markdownRendererUlBeginPrototype}%
```

The \markdownRendererUlBeginTight macro represents the beginning of a bulleted list that contains no item with several paragraphs of text (the list is tight). This macro will only be produced, when the tightLists option is false. The macro receives no arguments.

```
\def\markdownRendererUlBeginTight{%
  \markdownRendererUlBeginTightPrototype}%
```

The \markdownRendererUlItem macro represents an item in a bulleted list. The macro receives no arguments.

```
\def\markdownRendererUlItem{%
  \markdownRendererUlItemPrototype}%
```

The \markdownRendererUlEnd macro represents the end of a bulleted list that contains an item with several paragraphs of text (the list is not tight). The macro receives no arguments.

```
\def\markdownRendererUlEnd{%
  \markdownRendererUlEndPrototype}%
```

The \markdownRendererUlEndTight macro represents the end of a bulleted list that contains no item with several paragraphs of text (the list is tight). This macro will only be produced, when the tightLists option is false. The macro receives no arguments.

```
\def\markdownRendererUlEndTight{%
  \markdownRendererUlEndTightPrototype}%
```

2.2.3.8 Ordered List Renderers

The \markdownRendererOlBegin macro represents the beginning of an ordered list that contains an item with several paragraphs of text (the list is not tight). The macro receives no arguments.

```
\def\markdownRendererOlBegin{%
  \markdownRendererOlBeginPrototype}%
```

The \markdownRendererOlBeginTight macro represents the beginning of an ordered list that contains no item with several paragraphs of text (the list is tight). This macro will only be produced, when the tightLists option is false. The macro receives no arguments.

```
\def\markdownRendererOlBeginTight{%
  \markdownRendererOlBeginTightPrototype}%
```

The \markdownRendererOlItem macro represents an item in an ordered list. This macro will only be produced, when the startNumber option is false. The macro receives no arguments.

```
\def\markdownRendererOlItem{%
  \markdownRendererOlItemPrototype}%
```

The \markdownRendererOlItemWithNumber macro represents an item in an ordered list. This macro will only be produced, when the startNumber option is true. The macro receives no arguments.

```
\def\markdownRendererOlItemWithNumber{%
  \markdownRendererOlItemWithNumberPrototype}%
```

The `\markdownRendererOlEnd` macro represents the end of an ordered list that contains an item with several paragraphs of text (the list is not tight). The macro receives no arguments.

```
\def\markdownRendererOlEnd{%
  \markdownRendererOlEndPrototype}%
```

The `\markdownRendererOlEndTight` macro represents the end of an ordered list that contains no item with several paragraphs of text (the list is tight). This macro will only be produced, when the `tightLists` option is `false`. The macro receives no arguments.

```
\def\markdownRendererOlEndTight{%
  \markdownRendererOlEndTightPrototype}%
```

2.2.3.9 Definition List Renderers

The following macros are only produces, when the `definitionLists` option is `true`.

The `\markdownRendererDlBegin` macro represents the beginning of a definition list that contains an item with several paragraphs of text (the list is not tight). The macro receives no arguments.

```
\def\markdownRendererDlBegin{%
  \markdownRendererDlBeginPrototype}%
```

The `\markdownRendererDlBeginTight` macro represents the beginning of a definition list that contains an item with several paragraphs of text (the list is not tight). This macro will only be produced, when the `tightLists` option is `false`. The macro receives no arguments.

```
\def\markdownRendererDlBeginTight{%
  \markdownRendererDlBeginTightPrototype}%
```

The `\markdownRendererDlItem` macro represents a term in a definition list. The macro receives a single argument that corresponds to the term being defined.

```
\def\markdownRendererDlItem{%
  \markdownRendererDlItemPrototype}%
```

The `\markdownRendererDlDefinitionBegin` macro represents the beginning of a definition in a definition list. There can be several definitions for a single term.

```
\def\markdownRendererDlDefinitionBegin{%
  \markdownRendererDlDefinitionBeginPrototype}%
```

The `\markdownRendererDlDefinitionEnd` macro represents the end of a definition in a definition list. There can be several definitions for a single term.

```
\def\markdownRendererDlDefinitionEnd{%
  \markdownRendererDlDefinitionEndPrototype}%
```

The `\markdownRendererDlEnd` macro represents the end of a definition list that contains an item with several paragraphs of text (the list is not tight). The macro receives no arguments.

```
\def\markdownRendererDlEnd{%
  \markdownRendererDlEndPrototype}%
```

The `\markdownRendererDlEndTight` macro represents the end of a definition list that contains no item with several paragraphs of text (the list is tight). This macro will only be produced, when the `tightLists` option is `false`. The macro receives no arguments.

```
\def\markdownRendererDlEndTight{%
  \markdownRendererDlEndTightPrototype}%
```

2.2.3.10 Emphasis Renderers

The `\markdownRendererEmphasis` macro represents an emphasized span of text. The macro receives a single argument that corresponds to the emphasized span of text.

```
\def\markdownRendererEmphasis{%
  \markdownRendererEmphasisPrototype}%
```

The `\markdownRendererStrongEmphasis` macro represents a strongly emphasized span of text. The macro receives a single argument that corresponds to the emphasized span of text.

```
\def\markdownRendererStrongEmphasis{%
  \markdownRendererStrongEmphasisPrototype}%
```

2.2.3.11 Block Quote Renderers

The `\markdownRendererBlockQuoteBegin` macro represents the beginning of a block quote. The macro receives no arguments.

```
\def\markdownRendererBlockQuoteBegin{%
  \markdownRendererBlockQuoteBeginPrototype}%
```

The `\markdownRendererBlockQuoteEnd` macro represents the end of a block quote. The macro receives no arguments.

```
\def\markdownRendererBlockQuoteEnd{%
  \markdownRendererBlockQuoteEndPrototype}%
```

2.2.3.12 Code Block Renderer

The `\markdownRendererInputVerbatim` macro represents a code block. The macro receives a single argument that corresponds to the filename of a file contaning the code block to input.

```
\def\markdownRendererInputVerbatim{%
  \markdownRendererInputVerbatimPrototype}%
```

2.2.3.13 Heading Renderers

The `\markdownRendererHeadingOne` macro represents a first level heading. The macro receives a single argument that corresponds to the heading text.

```
\def\markdownRendererHeadingOne{%
  \markdownRendererHeadingOnePrototype}%
```

The `\markdownRendererHeadingTwo` macro represents a second level heading. The macro receives a single argument that corresponds to the heading text.

```
\def\markdownRendererHeadingTwo{%
  \markdownRendererHeadingTwoPrototype}%
```

The `\markdownRendererHeadingThree` macro represents a third level heading. The macro receives a single argument that corresponds to the heading text.

```
\def\markdownRendererHeadingThree{%
  \markdownRendererHeadingThreePrototype}%
```

The `\markdownRendererHeadingFour` macro represents a fourth level heading. The macro receives a single argument that corresponds to the heading text.

```
\def\markdownRendererHeadingFour{%
  \markdownRendererHeadingFourPrototype}%
```

The `\markdownRendererHeadingFive` macro represents a fifth level heading. The macro receives a single argument that corresponds to the heading text.

```
\def\markdownRendererHeadingFive{%
  \markdownRendererHeadingFivePrototype}%
```

The `\markdownRendererHeadingSix` macro represents a sixth level heading. The macro receives a single argument that corresponds to the heading text.

```
\def\markdownRendererHeadingSix{%
  \markdownRendererHeadingSixPrototype}%
```

2.2.3.14 Horizontal Rule Renderer

The `\markdownRendererHorizontalRule` macro represents a horizontal rule. The macro receives no arguments.

```
\def\markdownRendererHorizontalRule{%
  \markdownRendererHorizontalRulePrototype}%
```

2.2.3.15 Footnote Renderer

The `\markdownRendererFootnote` macro represents a footnote. This macro will only be produced, when the `footnotes` option is `true`. The macro receives a single argument that corresponds to the footnote text.

```
\def\markdownRendererFootnote{%
  \markdownRendererFootnotePrototype}%
```

2.2.4 Token Renderer Prototypes

The following TEX macros provide definitions for the token renderers (see Section 2.2.3) that have not been redefined by the user. These macros are intended to be

redefined by macro package authors who wish to provide sensible default token renderers. They are also redefined by the LaTeX and ConTeXt implementations (see sections 3.3 and 3.4).

```
\def\markdownRendererLineBreakPrototype{}%
\def\markdownRendererEllipsisPrototype{}%
\def\markdownRendererLeftBracePrototype{}%
\def\markdownRendererRightBracePrototype{}%
\def\markdownRendererDollarSignPrototype{}%
\def\markdownRendererPercentSignPrototype{}%
\def\markdownRendererAmpersandPrototype{}%
\def\markdownRendererUnderscorePrototype{}%
\def\markdownRendererHashPrototype{}%
\def\markdownRendererCircumflexPrototype{}%
\def\markdownRendererBackslashPrototype{}%
\def\markdownRendererTildePrototype{}%
\def\markdownRendererPipePrototype{}%
\long\def\markdownRendererCodeSpanPrototype#1{}%
\long\def\markdownRendererLinkPrototype#1#2#3#4{}%
\long\def\markdownRendererImagePrototype#1#2#3#4{}%
\def\markdownRendererUlBeginPrototype{}%
\def\markdownRendererUlBeginTightPrototype{}%
\def\markdownRendererUlItemPrototype{}%
\def\markdownRendererUlEndPrototype{}%
\def\markdownRendererUlEndTightPrototype{}%
\def\markdownRendererOlBeginPrototype{}%
\def\markdownRendererOlBeginTightPrototype{}%
\def\markdownRendererOlItemPrototype{}%
\long\def\markdownRendererOlItemWithNumberPrototype#1{}%
\def\markdownRendererOlEndPrototype{}%
\def\markdownRendererOlEndTightPrototype{}%
\def\markdownRendererDlBeginPrototype{}%
\def\markdownRendererDlBeginTightPrototype{}%
\long\def\markdownRendererDlItemPrototype#1{}%
\def\markdownRendererDlDefinitionBeginPrototype{}%
\def\markdownRendererDlDefinitionEndPrototype{}%
\def\markdownRendererDlEndPrototype{}%
\def\markdownRendererDlEndTightPrototype{}%
\long\def\markdownRendererEmphasisPrototype#1{}%
\long\def\markdownRendererStrongEmphasisPrototype#1{}%
\def\markdownRendererBlockQuoteBeginPrototype{}%
\def\markdownRendererBlockQuoteEndPrototype{}%
\long\def\markdownRendererInputVerbatimPrototype#1{}%
\long\def\markdownRendererHeadingOnePrototype#1{}%
\long\def\markdownRendererHeadingTwoPrototype#1{}%
\long\def\markdownRendererHeadingThreePrototype#1{}%
\long\def\markdownRendererHeadingFourPrototype#1{}%
\long\def\markdownRendererHeadingFivePrototype#1{}%
```

```
\long\def\markdownRendererHeadingSixPrototype#1{}%
\def\markdownRendererHorizontalRulePrototype{}%
\long\def\markdownRendererFootnotePrototype#1{}%
```

2.2.5 Logging Facilities

The `\markdownInfo`, `\markdownWarning`, and `\markdownError` macros provide access to logging to the rest of the macros. Their first argument specifies the text of the info, warning, or error message.

```
\def\markdownInfo#1{}%
\def\markdownWarning#1{}%
```

The `\markdownError` macro receives a second argument that provides a help text suggesting a remedy to the error.

```
\def\markdownError#1{}%
```

You may redefine these macros to redirect and process the info, warning, and error messages.

2.2.6 Miscellanea

The `\markdownLuaRegisterIBCallback` and `\markdownLuaUnregisterIBCallback` macros specify the Lua code for registering and unregistering a callback for changing the contents of the line input buffer before a TEX engine that supports direct Lua access via the `\directlua` macro starts looking at it. The first argument of the `\markdownLuaRegisterIBCallback` macro corresponds to the callback function being registered.

Local members defined within `\markdownLuaRegisterIBCallback` are guaranteed to be visible from `\markdownLuaUnregisterIBCallback` and the execution of the two macros alternates, so it is not necessary to consider the case, when one of the macros is called twice in a row.

```
\def\markdownLuaRegisterIBCallback#1{%
  local old_callback = callback.find("process_input_buffer")
  callback.register("process_input_buffer", #1)}%
\def\markdownLuaUnregisterIBCallback{%
  callback.register("process_input_buffer", old_callback)}%
```

The `\markdownMakeOther` macro is used by the package, when a TEX engine that does not support direct Lua access is starting to buffer a text. The plain TEX implementation changes the category code of plain TEX special characters to other, but there may be other active characters that may break the output. This macro should temporarily change the category of these to *other*.

```
\let\markdownMakeOther\relax
```

The `\markdownReadAndConvert` macro implements the `\markdownBegin` macro. The first argument specifies the token sequence that will terminate the markdown

input (`\markdownEnd` in the instance of the `\markdownBegin` macro) when the plain TEX special characters have had their category changed to *other*. The second argument specifies the token sequence that will actually be inserted into the document, when the ending token sequence has been found.

```
\let\markdownReadAndConvert\relax
\begingroup
```

Locally swap the category code of the backslash symbol (`\`) with the pipe symbol (`|`). This is required in order that all the special symbols in the first argument of the `markdownReadAndConvert` macro have the category code *other*.

```
  \catcode`\|=0\catcode`\\=12%
  |gdef|markdownBegin{%
    |markdownReadAndConvert{\markdownEnd}%
                           {|markdownEnd}}%
|endgroup
```

The macro is exposed in the interface, so that the user can create their own markdown environments. Due to the way the arguments are passed to Lua (see Section 3.2.5), the first argument may not contain the string `]]` (regardless of the category code of the bracket symbol (`]`)).

The `\markdownMode` macro specifies how the plain TEX implementation interfaces with the Lua interface. The valid values and their meaning are as follows:

- `0` – Shell escape via the 18 output file stream
- `1` – Shell escape via the Lua `os.execute` method
- `2` – Direct Lua access

By defining the macro, the user can coerce the package to use a specific mode. If the user does not define the macro prior to loading the plain TEX implementation, the correct value will be automatically detected. The outcome of changing the value of `\markdownMode` after the implementation has been loaded is undefined.

```
\ifx\markdownMode\undefined
  \ifx\directlua\undefined
    \def\markdownMode{0}%
  \else
    \def\markdownMode{2}%
  \fi
\fi
```

2.3 LATEX Interface

The LATEX interface provides LATEX environments for the typesetting of markdown input from within LATEX, facilities for setting Lua interface options (see Section 2.1.2) used during the conversion from markdown to plain TEX, and facilities for changing the

way markdown tokens are rendered. The rest of the interface is inherited from the plain TEX interface (see Section 2.2).

The LATEX interface is implemented by the `markdown.sty` file, which can be loaded from the LATEX document preamble as follows:

```
\usepackage[⟨options⟩]{markdown}
```

where ⟨*options*⟩ are the LATEX interface options (see Section 2.3.2).

2.3.1 Typesetting Markdown

The interface exposes the `markdown` and `markdown*` LATEX environments, and redefines the `\markdownInput` command.

The `markdown` and `markdown*` LATEX environments are used to typeset markdown document fragments. The starred version of the `markdown` environment accepts LATEX interface options (see Section 2.3.2) as its only argument. These options will only influnce this markdown document fragment.

```
\newenvironment{markdown}\relax\relax
\newenvironment{markdown*}[1]\relax\relax
```

You may prepend your own code to the `\markdown` macro and append your own code to the `\endmarkdown` macro to produce special effects before and after the `markdown` LATEX environment (and likewise for the starred version).

Note that the `markdown` and `markdown*` LATEX environments are subject to the same limitations as the `\markdownBegin` and `\markdownEnd` macros exposed by the plain TEX interface.

The following example LATEX code showcases the usage of the `markdown` and `markdown*` environments:

```
\documentclass{article}
\usepackage{markdown}
\begin{document}
% ...
\begin{markdown}
_Hello_ **world** ...
\end{markdown}
% ...
\end{document}
```

```
\documentclass{article}
\usepackage{markdown}
\begin{document}
% ...
\begin{markdown*}{smartEllipses}
_Hello_ **world** ...
\end{markdown*}
% ...
\end{document}
```

The `\markdownInput` macro accepts a single mandatory parameter containing the filename of a markdown document and expands to the result of the conversion of the input markdown document to plain TEX. Unlike the `\markdownInput` macro provided by the plain TEX interface, this macro also accepts LATEX interface options (see Section

2.3.2) as its optional argument. These options will only influnce this markdown document.

The following example LaTeX code showcases the usage of the `\markdownInput` macro:

```
\documentclass{article}
\usepackage{markdown}
\begin{document}
% ...
\markdownInput[smartEllipses]{hello.md}
% ...
\end{document}
```

2.3.2 Options

The LaTeX options are represented by a comma-delimited list of ⟨⟨*key*⟩=⟨*value*⟩⟩ pairs. For boolean options, the ⟨=⟨*value*⟩⟩ part is optional, and ⟨⟨*key*⟩⟩ will be interpreted as ⟨⟨*key*⟩=*true*⟩.

The LaTeX options map directly to the options recognized by the plain TeX interface (see Section 2.2.2) and to the markdown token renderers and their prototypes recognized by the plain TeX interface (see Sections 2.2.3 and 2.2.4).

The LaTeX options may be specified when loading the LaTeX package (see Section 2.3), when using the `markdown*` LaTeX environment, or via the `\markdownSetup` macro. The `\markdownSetup` macro receives the options to set up as its only argument.

```
\newcommand\markdownSetup[1]{%
  \setkeys{markdownOptions}{#1}}%
```

2.3.2.1 Plain TeX Interface Options

The following options map directly to the option macros exposed by the plain TeX interface (see Section 2.2.2).

```
\RequirePackage{keyval}
\define@key{markdownOptions}{helperScriptFileName}{%
  \def\markdownOptionHelperScriptFileName{#1}}%
\define@key{markdownOptions}{inputTempFileName}{%
  \def\markdownOptionInputTempFileName{#1}}%
\define@key{markdownOptions}{outputTempFileName}{%
  \def\markdownOptionOutputTempFileName{#1}}%
\define@key{markdownOptions}{blankBeforeBlockquote}[true]{%
  \def\markdownOptionBlankBeforeBlockquote{#1}}%
\define@key{markdownOptions}{blankBeforeHeading}[true]{%
  \def\markdownOptionBlankBeforeHeading{#1}}%
\define@key{markdownOptions}{cacheDir}{%
  \def\markdownOptionCacheDir{#1}}%
\define@key{markdownOptions}{definitionLists}[true]{%
```

```
  \def\markdownOptionDefinitionLists{#1}%
\define@key{markdownOptions}{hashEnumerators}[true]{%
  \def\markdownOptionHashEnumerators{#1}%
\define@key{markdownOptions}{hybrid}[true]{%
  \def\markdownOptionHybrid{#1}%
\define@key{markdownOptions}{footnotes}[true]{%
  \def\markdownOptionFootnotes{#1}%
\define@key{markdownOptions}{preserveTabs}[true]{%
  \def\markdownOptionPreserveTabs{#1}%
\define@key{markdownOptions}{smartEllipses}[true]{%
  \def\markdownOptionSmartEllipses{#1}%
\define@key{markdownOptions}{startNumber}[true]{%
  \def\markdownOptionStartNumber{#1}%
```

If the `tightLists=false` option is specified, when loading the package, then the paralist package for typesetting tight lists will not be automatically loaded. This precaution is meant to minimize the footprint of this package, since some document-classes (beamer) do not play nice with the paralist package.

```
\define@key{markdownOptions}{tightLists}[true]{%
  \def\markdownOptionTightLists{#1}%
```

The following example LaTeX code showcases a possible configuration of plain TeX interface options `\markdownOptionHybrid`, `\markdownOptionSmartEllipses`, and `\markdownOptionCacheDir`.

```
\markdownSetup{
  hybrid,
  smartEllipses,
  cacheDir = /tmp,
}
```

2.3.2.2 Plain TeX Markdown Token Renderers

The LaTeX interface recognizes an option with the `renderers` key, whose value must be a list of options that map directly to the markdown token renderer macros exposed by the plain TeX interface (see Section 2.2.3).

```
\define@key{markdownOptions}{renderers}{%
  \setkeys{markdownRenderers}{#1}%
\define@key{markdownRenderers}{lineBreak}{%
  \renewcommand\markdownRendererLineBreak{#1}%
\define@key{markdownRenderers}{ellipsis}{%
  \renewcommand\markdownRendererEllipsis{#1}%
\define@key{markdownRenderers}{leftBrace}{%
  \renewcommand\markdownRendererLeftBrace{#1}%
\define@key{markdownRenderers}{rightBrace}{%
  \renewcommand\markdownRendererRightBrace{#1}%
```

```
\define@key{markdownRenderers}{dollarSign}{%
  \renewcommand\markdownRendererDollarSign{#1}}%
\define@key{markdownRenderers}{percentSign}{%
  \renewcommand\markdownRendererPercentSign{#1}}%
\define@key{markdownRenderers}{ampersand}{%
  \renewcommand\markdownRendererAmpersand{#1}}%
\define@key{markdownRenderers}{underscore}{%
  \renewcommand\markdownRendererUnderscore{#1}}%
\define@key{markdownRenderers}{hash}{%
  \renewcommand\markdownRendererHash{#1}}%
\define@key{markdownRenderers}{circumflex}{%
  \renewcommand\markdownRendererCircumflex{#1}}%
\define@key{markdownRenderers}{backslash}{%
  \renewcommand\markdownRendererBackslash{#1}}%
\define@key{markdownRenderers}{tilde}{%
  \renewcommand\markdownRendererTilde{#1}}%
\define@key{markdownRenderers}{pipe}{%
  \renewcommand\markdownRendererPipe{#1}}%
\define@key{markdownRenderers}{codeSpan}{%
  \renewcommand\markdownRendererCodeSpan[1]{#1}}%
\define@key{markdownRenderers}{link}{%
  \renewcommand\markdownRendererLink[4]{#1}}%
\define@key{markdownRenderers}{image}{%
  \renewcommand\markdownRendererImage[4]{#1}}%
\define@key{markdownRenderers}{ulBegin}{%
  \renewcommand\markdownRendererUlBegin{#1}}%
\define@key{markdownRenderers}{ulBeginTight}{%
  \renewcommand\markdownRendererUlBeginTight{#1}}%
\define@key{markdownRenderers}{ulItem}{%
  \renewcommand\markdownRendererUlItem{#1}}%
\define@key{markdownRenderers}{ulEnd}{%
  \renewcommand\markdownRendererUlEnd{#1}}%
\define@key{markdownRenderers}{ulEndTight}{%
  \renewcommand\markdownRendererUlEndTight{#1}}%
\define@key{markdownRenderers}{olBegin}{%
  \renewcommand\markdownRendererOlBegin{#1}}%
\define@key{markdownRenderers}{olBeginTight}{%
  \renewcommand\markdownRendererOlBeginTight{#1}}%
\define@key{markdownRenderers}{olItem}{%
  \renewcommand\markdownRendererOlItem{#1}}%
\define@key{markdownRenderers}{olItemWithNumber}{%
  \renewcommand\markdownRendererOlItemWithNumber[1]{#1}}%
\define@key{markdownRenderers}{olEnd}{%
  \renewcommand\markdownRendererOlEnd{#1}}%
\define@key{markdownRenderers}{olEndTight}{%
  \renewcommand\markdownRendererOlEndTight{#1}}%
\define@key{markdownRenderers}{dlBegin}{%
```

```
  \renewcommand\markdownRendererDlBegin{#1}}%
\define@key{markdownRenderers}{dlBeginTight}{%
  \renewcommand\markdownRendererDlBeginTight{#1}}%
\define@key{markdownRenderers}{dlItem}{%
  \renewcommand\markdownRendererDlItem[1]{#1}}%
\define@key{markdownRenderers}{dlDefinitionBegin}{%
  \renewcommand\markdownRendererDlDefinitionBegin{#1}}%
\define@key{markdownRenderers}{dlDefinitionEnd}{%
  \renewcommand\markdownRendererDlDefinitionEnd{#1}}%
\define@key{markdownRenderers}{dlEnd}{%
  \renewcommand\markdownRendererDlEnd{#1}}%
\define@key{markdownRenderers}{dlEndTight}{%
  \renewcommand\markdownRendererDlEndTight{#1}}%
\define@key{markdownRenderers}{emphasis}{%
  \renewcommand\markdownRendererEmphasis[1]{#1}}%
\define@key{markdownRenderers}{strongEmphasis}{%
  \renewcommand\markdownRendererStrongEmphasis[1]{#1}}%
\define@key{markdownRenderers}{blockQuoteBegin}{%
  \renewcommand\markdownRendererBlockQuoteBegin{#1}}%
\define@key{markdownRenderers}{blockQuoteEnd}{%
  \renewcommand\markdownRendererBlockQuoteEnd{#1}}%
\define@key{markdownRenderers}{inputVerbatim}{%
  \renewcommand\markdownRendererInputVerbatim[1]{#1}}%
\define@key{markdownRenderers}{headingOne}{%
  \renewcommand\markdownRendererHeadingOne[1]{#1}}%
\define@key{markdownRenderers}{headingTwo}{%
  \renewcommand\markdownRendererHeadingTwo[1]{#1}}%
\define@key{markdownRenderers}{headingThree}{%
  \renewcommand\markdownRendererHeadingThree[1]{#1}}%
\define@key{markdownRenderers}{headingFour}{%
  \renewcommand\markdownRendererHeadingFour[1]{#1}}%
\define@key{markdownRenderers}{headingFive}{%
  \renewcommand\markdownRendererHeadingFive[1]{#1}}%
\define@key{markdownRenderers}{headingSix}{%
  \renewcommand\markdownRendererHeadingSix[1]{#1}}%
\define@key{markdownRenderers}{horizontalRule}{%
  \renewcommand\markdownRendererHorizontalRule{#1}}%
\define@key{markdownRenderers}{footnote}{%
  \renewcommand\markdownRendererFootnote[1]{#1}}%
```

The following example LaTeX code showcases a possible configuration of the `\markdownRendererLink` and `\markdownRendererEmphasis` markdown token renderers.

```
\markdownSetup{
  renderers = {
    link = {#4},                    % Render links as the link title.
    emphasis = {\emph{#1}},    % Render emphasized text via `\emph`.
```

```
  }
}
```

2.3.2.3 Plain TEX Markdown Token Renderer Prototypes

The LATEX interface recognizes an option with the `rendererPrototypes` key, whose value must be a list of options that map directly to the markdown token renderer prototype macros exposed by the plain TEX interface (see Section 2.2.4).

```
\define@key{markdownOptions}{rendererPrototypes}{%
  \setkeys{markdownRendererPrototypes}{#1}}%
\define@key{markdownRendererPrototypes}{lineBreak}{%
  \renewcommand\markdownRendererLineBreakPrototype{#1}}%
\define@key{markdownRendererPrototypes}{ellipsis}{%
  \renewcommand\markdownRendererEllipsisPrototype{#1}}%
\define@key{markdownRendererPrototypes}{leftBrace}{%
  \renewcommand\markdownRendererLeftBracePrototype{#1}}%
\define@key{markdownRendererPrototypes}{rightBrace}{%
  \renewcommand\markdownRendererRightBracePrototype{#1}}%
\define@key{markdownRendererPrototypes}{dollarSign}{%
  \renewcommand\markdownRendererDollarSignPrototype{#1}}%
\define@key{markdownRendererPrototypes}{percentSign}{%
  \renewcommand\markdownRendererPercentSignPrototype{#1}}%
\define@key{markdownRendererPrototypes}{ampersand}{%
  \renewcommand\markdownRendererAmpersandPrototype{#1}}%
\define@key{markdownRendererPrototypes}{underscore}{%
  \renewcommand\markdownRendererUnderscorePrototype{#1}}%
\define@key{markdownRendererPrototypes}{hash}{%
  \renewcommand\markdownRendererHashPrototype{#1}}%
\define@key{markdownRendererPrototypes}{circumflex}{%
  \renewcommand\markdownRendererCircumflexPrototype{#1}}%
\define@key{markdownRendererPrototypes}{backslash}{%
  \renewcommand\markdownRendererBackslashPrototype{#1}}%
\define@key{markdownRendererPrototypes}{tilde}{%
  \renewcommand\markdownRendererTildePrototype{#1}}%
\define@key{markdownRendererPrototypes}{pipe}{%
  \renewcommand\markdownRendererPipe{#1}}%
\define@key{markdownRendererPrototypes}{codeSpan}{%
  \renewcommand\markdownRendererCodeSpanPrototype[1]{#1}}%
\define@key{markdownRendererPrototypes}{link}{%
  \renewcommand\markdownRendererLink[4]{#1}}%
\define@key{markdownRendererPrototypes}{image}{%
  \renewcommand\markdownRendererImage[4]{#1}}%
\define@key{markdownRendererPrototypes}{ulBegin}{%
  \renewcommand\markdownRendererUlBeginPrototype{#1}}%
\define@key{markdownRendererPrototypes}{ulBeginTight}{%
  \renewcommand\markdownRendererUlBeginTightPrototype{#1}}%
```

```
\define@key{markdownRendererPrototypes}{ulItem}{%
  \renewcommand\markdownRendererUlItemPrototype{#1}}%
\define@key{markdownRendererPrototypes}{ulEnd}{%
  \renewcommand\markdownRendererUlEndPrototype{#1}}%
\define@key{markdownRendererPrototypes}{ulEndTight}{%
  \renewcommand\markdownRendererUlEndTightPrototype{#1}}%
\define@key{markdownRendererPrototypes}{olBegin}{%
  \renewcommand\markdownRendererOlBeginPrototype{#1}}%
\define@key{markdownRendererPrototypes}{olBeginTight}{%
  \renewcommand\markdownRendererOlBeginTightPrototype{#1}}%
\define@key{markdownRendererPrototypes}{olItem}{%
  \renewcommand\markdownRendererOlItemPrototype{#1}}%
\define@key{markdownRendererPrototypes}{olItemWithNumber}{%
  \renewcommand\markdownRendererOlItemWithNumberPrototype[1]{#1}}%
\define@key{markdownRendererPrototypes}{olEnd}{%
  \renewcommand\markdownRendererOlEndPrototype{#1}}%
\define@key{markdownRendererPrototypes}{olEndTight}{%
  \renewcommand\markdownRendererOlEndTightPrototype{#1}}%
\define@key{markdownRendererPrototypes}{dlBegin}{%
  \renewcommand\markdownRendererDlBeginPrototype{#1}}%
\define@key{markdownRendererPrototypes}{dlBeginTight}{%
  \renewcommand\markdownRendererDlBeginTightPrototype{#1}}%
\define@key{markdownRendererPrototypes}{dlItem}{%
  \renewcommand\markdownRendererDlItemPrototype[1]{#1}}%
\define@key{markdownRendererPrototypes}{dlDefinitionBegin}{%
  \renewcommand\markdownRendererDlDefinitionBeginPrototype{#1}}%
\define@key{markdownRendererPrototypes}{dlDefinitionEnd}{%
  \renewcommand\markdownRendererDlDefinitionEndPrototype{#1}}%
\define@key{markdownRendererPrototypes}{dlEnd}{%
  \renewcommand\markdownRendererDlEndPrototype{#1}}%
\define@key{markdownRendererPrototypes}{dlEndTight}{%
  \renewcommand\markdownRendererDlEndTightPrototype{#1}}%
\define@key{markdownRendererPrototypes}{emphasis}{%
  \renewcommand\markdownRendererEmphasisPrototype[1]{#1}}%
\define@key{markdownRendererPrototypes}{strongEmphasis}{%
  \renewcommand\markdownRendererStrongEmphasisPrototype[1]{#1}}%
\define@key{markdownRendererPrototypes}{blockQuoteBegin}{%
  \renewcommand\markdownRendererBlockQuoteBeginPrototype{#1}}%
\define@key{markdownRendererPrototypes}{blockQuoteEnd}{%
  \renewcommand\markdownRendererBlockQuoteEndPrototype{#1}}%
\define@key{markdownRendererPrototypes}{inputVerbatim}{%
  \renewcommand\markdownRendererInputVerbatimPrototype[1]{#1}}%
\define@key{markdownRendererPrototypes}{headingOne}{%
  \renewcommand\markdownRendererHeadingOnePrototype[1]{#1}}%
\define@key{markdownRendererPrototypes}{headingTwo}{%
  \renewcommand\markdownRendererHeadingTwoPrototype[1]{#1}}%
\define@key{markdownRendererPrototypes}{headingThree}{%
```

```
  \renewcommand\markdownRendererHeadingThreePrototype[1]{#1}%
\define@key{markdownRendererPrototypes}{headingFour}{%
  \renewcommand\markdownRendererHeadingFourPrototype[1]{#1}%
\define@key{markdownRendererPrototypes}{headingFive}{%
  \renewcommand\markdownRendererHeadingFivePrototype[1]{#1}%
\define@key{markdownRendererPrototypes}{headingSix}{%
  \renewcommand\markdownRendererHeadingSixPrototype[1]{#1}%
\define@key{markdownRendererPrototypes}{horizontalRule}{%
  \renewcommand\markdownRendererHorizontalRulePrototype{#1}%
\define@key{markdownRendererPrototypes}{footnote}{%
  \renewcommand\markdownRendererFootnotePrototype[1]{#1}%
```

The following example LaTeX code showcases a possible configuration of the `\markdownRendererImagePrototype` and `\markdownRendererCodeSpanPrototype` markdown token renderer prototypes.

```
\markdownSetup{
  renderers = {
    image = {\includegraphics{#2}},
    codeSpan = {\texttt{#1}},    % Render inline code via `\texttt`.
  }
}
```

2.4 ConTeXt Interface

The ConTeXt interface provides a start-stop macro pair for the typesetting of markdown input from within ConTeXt. The rest of the interface is inherited from the plain TeX interface (see Section 2.2).

```
\writestatus{loading}{ConTeXt User Module / markdown}%
\unprotect
```

The ConTeXt interface is implemented by the `t-markdown.tex` ConTeXt module file that can be loaded as follows:

```
\usemodule[t][markdown]
```

It is expected that the special plain TeX characters have the expected category codes, when `\input`ting the file.

2.4.1 Typesetting Markdown

The interface exposes the `\startmarkdown` and `\stopmarkdown` macro pair for the typesetting of a markdown document fragment.

```
\let\startmarkdown\relax
\let\stopmarkdown\relax
```

You may prepend your own code to the `\startmarkdown` macro and redefine the `\stopmarkdown` macro to produce special effects before and after the markdown block.

Note that the `\startmarkdown` and `\stopmarkdown` macros are subject to the same limitations as the `\markdownBegin` and `\markdownEnd` macros exposed by the plain TEX interface.

The following example ConTEXt code showcases the usage of the `\startmarkdown` and `\stopmarkdown` macros:

```
\usemodule[t][markdown]
\starttext
\startmarkdown
_Hello_ **world** ...
\stopmarkdown
\stoptext
```

3 Technical Documentation

This part of the manual describes the implementation of the interfaces exposed by the package (see Section 2) and is aimed at the developers of the package, as well as the curious users.

3.1 Lua Implementation

The Lua implementation implements `writer` and `reader` objects that provide the conversion from markdown to plain TEX.

The Lunamark Lua module implements writers for the conversion to various other formats, such as DocBook, Groff, or HTML. These were stripped from the module and the remaining markdown reader and plain TEX writer were hidden behind the converter functions exposed by the Lua interface (see Section 2.1).

```
local upper, gsub, format, length =
  string.upper, string.gsub, string.format, string.len
local concat = table.concat
local P, R, S, V, C, Cg, Cb, Cmt, Cc, Ct, B, Cs, any =
  lpeg.P, lpeg.R, lpeg.S, lpeg.V, lpeg.C, lpeg.Cg, lpeg.Cb,
  lpeg.Cmt, lpeg.Cc, lpeg.Ct, lpeg.B, lpeg.Cs, lpeg.P(1)
```

3.1.1 Utility Functions

This section documents the utility functions used by the Lua code. These functions are encapsulated in the `util` object. The functions were originally located in the `lunamark/util.lua` file in the Lunamark Lua module.

```
local util = {}
```

The `util.err` method prints an error message `msg` and exits. If `exit_code` is provided, it specifies the exit code. Otherwise, the exit code will be 1.

```
function util.err(msg, exit_code)
  io.stderr:write("markdown.lua: " .. msg .. "\n")
  os.exit(exit_code or 1)
end
```

The `util.cache` method computes the digest of `string` and `salt`, adds the `suffix` and looks into the directory `dir`, whether a file with such a name exists. If it does not, it gets created with `transform(string)` as its content. The filename is then returned.

```
function util.cache(dir, string, salt, transform, suffix)
  local digest = md5.sumhexa(string .. (salt or ""))
  local name = util.pathname(dir, digest .. suffix)
  local file = io.open(name, "r")
  if file == nil then -- If no cache entry exists, then create a new one.
    local file = assert(io.open(name, "w"))
    local result = string
    if transform ~= nil then
      result = transform(result)
    end
    assert(file:write(result))
    assert(file:close())
  end
  return name
end
```

The `util.table_copy` method creates a shallow copy of a table `t` and its metatable.

```
function util.table_copy(t)
  local u = { }
  for k, v in pairs(t) do u[k] = v end
  return setmetatable(u, getmetatable(t))
end
```

The `util.expand_tabs_in_line` expands tabs in string `s`. If `tabstop` is specified, it is used as the tab stop width. Otherwise, the tab stop width of 4 characters is used. The method is a copy of the tab expansion algorithm from [3, Chapter 21].

```
function util.expand_tabs_in_line(s, tabstop)
  local tab = tabstop or 4
  local corr = 0
  return (s:gsub("()\t", function(p)
            local sp = tab - (p - 1 + corr) % tab
            corr = corr - 1 + sp
            return string.rep(" ", sp)
          end))
end
```

The `util.walk` method walks a rope `t`, applying a function `f` to each leaf element in order. A rope is an array whose elements may be ropes, strings, numbers, or functions. If a leaf element is a function, call it and get the return value before proceeding.

```
function util.walk(t, f)
  local typ = type(t)
  if typ == "string" then
    f(t)
  elseif typ == "table" then
    local i = 1
    local n
    n = t[i]
    while n do
      util.walk(n, f)
      i = i + 1
      n = t[i]
    end
  elseif typ == "function" then
    local ok, val = pcall(t)
    if ok then
      util.walk(val,f)
    end
  else
    f(tostring(t))
  end
end
```

The `util.flatten` method flattens an array `ary` that does not contain cycles and returns the result.

```
function util.flatten(ary)
  local new = {}
  for _,v in ipairs(ary) do
    if type(v) == "table" then
      for _,w in ipairs(util.flatten(v)) do
        new[#new + 1] = w
      end
    else
      new[#new + 1] = v
    end
  end
  return new
end
```

The `util.rope_to_string` method converts a rope `rope` to a string and returns it. For the definition of a rope, see the definition of the `util.walk` method.

```
function util.rope_to_string(rope)
  local buffer = {}
```

```
514   util.walk(rope, function(x) buffer[#buffer + 1] = x end)
515   return table.concat(buffer)
516 end
```

The `util.rope_last` method retrieves the last item in a rope. For the definition of a rope, see the definition of the `util.walk` method.

```
517 function util.rope_last(rope)
518   if #rope == 0 then
519     return nil
520   else
521     local l = rope[#rope]
522     if type(l) == "table" then
523       return util.rope_last(l)
524     else
525       return l
526     end
527   end
528 end
```

Given an array `ary` and a string `x`, the `util.intersperse` method returns an array `new`, such that `ary[i] == new[2*(i-1)+1]` and `new[2*i] == x` for all $1 \leq$ `i` $\leq$ `#ary`.

```
529 function util.intersperse(ary, x)
530   local new = {}
531   local l = #ary
532   for i,v in ipairs(ary) do
533     local n = #new
534     new[n + 1] = v
535     if i ~= l then
536       new[n + 2] = x
537     end
538   end
539   return new
540 end
```

Given an array `ary` and a function `f`, the `util.map` method returns an array `new`, such that `new[i] == f(ary[i])` for all $1 \leq$ `i` $\leq$ `#ary`.

```
541 function util.map(ary, f)
542   local new = {}
543   for i,v in ipairs(ary) do
544     new[i] = f(v)
545   end
546   return new
547 end
```

Given a table `char_escapes` mapping escapable characters to escaped strings and optionally a table `string_escapes` mapping escapable strings to escaped strings, the

`util.escaper` method returns an escaper function that escapes all occurances of escapable strings and characters (in this order).

The method uses LPeg, which is faster than the Lua `string.gsub` built-in method.

```
548 function util.escaper(char_escapes, string_escapes)
```

Build a string of escapable characters.

```
549   local char_escapes_list = ""
550   for i,_ in pairs(char_escapes) do
551     char_escapes_list = char_escapes_list .. i
552   end
```

Create an LPeg capture `escapable` that produces the escaped string corresponding to the matched escapable character.

```
553   local escapable = S(char_escapes_list) / char_escapes
```

If `string_escapes` is provided, turn `escapable` into the

$$\sum_{\texttt{(k,v)}\in\texttt{string\_escapes}} \texttt{P(k) / v} + \texttt{escapable}$$

capture that replaces any occurance of the string `k` with the string `v` for each $(\texttt{k}, \texttt{v}) \in$ `string_escapes`. Note that the pattern summation is not commutative and the its operands are inspected in the summation order during the matching. As a corrolary, the strings always take precedence over the characters.

```
554   if string_escapes then
555     for k,v in pairs(string_escapes) do
556       escapable = P(k) / v + escapable
557     end
558   end
```

Create an LPeg capture `escape_string` that captures anything `escapable` does and matches any other unmatched characters.

```
559   local escape_string = Cs((escapable + any)^0)
```

Return a function that matches the input string `s` against the `escape_string` capture.

```
560   return function(s)
561     return lpeg.match(escape_string, s)
562   end
563 end
```

The `util.pathname` method produces a pathname out of a directory name `dir` and a filename `file` and returns it.

```
564 function util.pathname(dir, file)
565   if #dir == 0 then
566     return file
567   else
568     return dir .. "/" .. file
569   end
570 end
```

3.1.2 Plain TEX Writer

This section documents the `writer` object, which implements the routines for producing the TEX output. The object is an amalgamate of the generic, TEX, LATEX writer objects that were located in the `lunamark/writer/generic.lua`, `lunamark/writer/tex.lua`, and `lunamark/writer/latex.lua` files in the Lunamark Lua module.

Although not specified in the Lua interface (see Section 2.1), the `writer` object is exported, so that the curious user could easily tinker with the methods of the objects produced by the `writer.new` method described below. The user should be aware, however, that the implementation may change in a future revision.

```
571 M.writer = {}
```

The `writer.new` method creates and returns a new TEX writer object associated with the Lua interface options (see Section 2.1.2) `options`. When `options` are unspecified, it is assumed that an empty table was passed to the method.

The objects produced by the `writer.new` method expose instance methods and variables of their own. As a convention, I will refer to these ⟨*member*⟩s as `writer->`⟨*member*⟩.

```
572 function M.writer.new(options)
573   local self = {}
574   options = options or {}
```

Make the `options` table inherit from the `defaultOptions` table.

```
575   setmetatable(options, { __index = function (_, key)
576     return defaultOptions[key] end })
```

Define `writer->suffix` as the suffix of the produced cache files.

```
577   self.suffix = ".tex"
```

Define `writer->space` as the output format of a space character.

```
578   self.space = " "
```

Define `writer->plain` as a function that will transform an input plain text block `s` to the output format.

```
579   function self.plain(s)
580     return s
581   end
```

Define `writer->paragraph` as a function that will transform an input paragraph `s` to the output format.

```
582   function self.paragraph(s)
583     return s
584   end
```

Define `writer->pack` as a function that will take the filename `name` of the output file prepared by the reader and transform it to the output format.

```
585   function self.pack(name)
```

```
    return [[\input"]] .. name .. [["\relax]]
  end
```

Define `writer->interblocksep` as the output format of a block element separator.

```
  self.interblocksep = "\n\n"
```

Define `writer->containersep` as the output format of a container separator.

```
  self.containersep = "\n\n"
```

Define `writer->eof` as the end of file marker in the output format.

```
  self.eof = [[\relax]]
```

Define `writer->linebreak` as the output format of a forced line break.

```
  self.linebreak = "\\markdownRendererLineBreak "
```

Define `writer->ellipsis` as the output format of an ellipsis.

```
  self.ellipsis = "\\markdownRendererEllipsis{}"
```

Define `writer->hrule` as the output format of a horizontal rule.

```
  self.hrule = "\\markdownRendererHorizontalRule "
```

Define a table `escaped_chars` containing the mapping from special plain TEX characters (including the active pipe character (`|`) of ConTEXt) to their escaped variants. Define tables `escaped_minimal_chars` and `escaped_minimal_strings` containing the mapping from special plain characters and character strings that need to be escaped even in content that will not be typeset.

```
  local escaped_chars = {
     ["{"] = "\\markdownRendererLeftBrace{}",
     ["}"] = "\\markdownRendererRightBrace{}",
     ["$"] = "\\markdownRendererDollarSign{}",
     ["%"] = "\\markdownRendererPercentSign{}",
     ["&"] = "\\markdownRendererAmpersand{}",
     ["_"] = "\\markdownRendererUnderscore{}",
     ["#"] = "\\markdownRendererHash{}",
     ["^"] = "\\markdownRendererCircumflex{}",
     ["\\"] = "\\markdownRendererBackslash{}",
     ["~"] = "\\markdownRendererTilde{}",
     ["|"] = "\\markdownRendererPipe{}", }
   local escaped_minimal_chars = {
     ["{"] = "\\markdownRendererLeftBrace{}",
     ["}"] = "\\markdownRendererRightBrace{}",
     ["%"] = "\\markdownRendererPercentSign{}",
     ["\\"] = "\\markdownRendererBackslash{}", }
   local escaped_minimal_strings = {
     ["^^"] = "\\markdownRendererCircumflex\\markdownRendererCircumflex ", }
```

Use the `escaped_chars` table to create an escaper function `escape` and the `escaped_minimal_chars` and `escaped_minimal_strings` tables to create an escaper function `escape_minimal`.

```
  local escape = util.escaper(escaped_chars)
  local escape_minimal = util.escaper(escaped_minimal_chars,
    escaped_minimal_strings)
```

Define `writer->string` as a function that will transform an input plain text span `s` to the output format and `writer->uri` as a function that will transform an input URI `u` to the output format. If the `hybrid` option is `true`, use identity functions. Otherwise, use the `escape` and `escape_minimal` functions.

```
  if options.hybrid then
    self.string = function(s) return s end
    self.uri = function(u) return u end
  else
    self.string = escape
    self.uri = escape_minimal
  end
```

Define `writer->code` as a function that will transform an input inlined code span `s` to the output format.

```
  function self.code(s)
    return {"\\markdownRendererCodeSpan{",escape(s),"}"}
  end
```

Define `writer->link` as a function that will transform an input hyperlink to the output format, where `lab` corresponds to the label, `src` to URI, and `tit` to the title of the link.

```
  function self.link(lab,src,tit)
    return {"\\markdownRendererLink{",lab,"}",
                           "{",self.string(src),"}",
                           "{",self.uri(src),"}",
                           "{",self.string(tit or ""),"}"}
  end
```

Define `writer->image` as a function that will transform an input image to the output format, where `lab` corresponds to the label, `src` to the URL, and `tit` to the title of the image.

```
  function self.image(lab,src,tit)
    return {"\\markdownRendererImage{",lab,"}",
                            "{",self.string(src),"}",
                            "{",self.uri(src),"}",
                            "{",self.string(tit or ""),"}"}
  end
```

Define `writer->bulletlist` as a function that will transform an input bulleted list to the output format, where `items` is an array of the list items and `tight` specifies, whether the list is tight or not.

```
  local function ulitem(s)
    return {"\\markdownRendererUlItem ",s}
  end
```

```
641
642   function self.bulletlist(items,tight)
643     local buffer = {}
644     for _,item in ipairs(items) do
645       buffer[#buffer + 1] = ulitem(item)
646     end
647     local contents = util.intersperse(buffer,"\n")
648     if tight and options.tightLists then
649       return {"\\markdownRendererUlBeginTight\n",contents,
650         "\n\\markdownRendererUlEndTight "}
651     else
652       return {"\\markdownRendererUlBegin\n",contents,
653         "\n\\markdownRendererUlEnd "}
654     end
655   end
```

Define `writer->ollist` as a function that will transform an input ordered list to the output format, where `items` is an array of the list items and `tight` specifies, whether the list is tight or not. If the optional parameter `startnum` is present, it should be used as the number of the first list item.

```
656   local function olitem(s,num)
657     if num ~= nil then
658       return {"\\markdownRendererOlItemWithNumber{",num,"}",s}
659     else
660       return {"\\markdownRendererOlItem ",s}
661     end
662   end
663
664   function self.orderedlist(items,tight,startnum)
665     local buffer = {}
666     local num = startnum
667     for _,item in ipairs(items) do
668       buffer[#buffer + 1] = olitem(item,num)
669       if num ~= nil then
670         num = num + 1
671       end
672     end
673     local contents = util.intersperse(buffer,"\n")
674     if tight and options.tightLists then
675       return {"\\markdownRendererOlBeginTight\n",contents,
676         "\n\\markdownRendererOlEndTight "}
677     else
678       return {"\\markdownRendererOlBegin\n",contents,
679         "\n\\markdownRendererOlEnd "}
680     end
681   end
```

Define `writer->definitionlist` as a function that will transform an input definition list to the output format, where `items` is an array of tables, each of the form `{ term = t, definitions = defs }`, where `t` is a term and `defs` is an array of definitions. `tight` specifies, whether the list is tight or not.

```
682  local function dlitem(term, defs)
683    local retVal = {"\\markdownRendererDlItem{",term,"}"}
684    for _, def in ipairs(defs) do
685      retVal[#retVal+1] = {"\\markdownRendererDlDefinitionBegin ",def,
686                           "\\markdownRendererDlDefinitionEnd "}
687    end
688    return retVal
689  end
690
691  function self.definitionlist(items,tight)
692    local buffer = {}
693    for _,item in ipairs(items) do
694      buffer[#buffer + 1] = dlitem(item.term, item.definitions)
695    end
696    local contents = util.intersperse(buffer, self.containersep)
697    if tight and options.tightLists then
698      return {"\\markdownRendererDlBeginTight\n\n", contents,
699        "\n\n\\markdownRendererDlEndTight\n"}
700    else
701      return {"\\markdownRendererDlBegin\n\n", contents,
702        "\n\n\\markdownRendererDlEnd\n"}
703    end
704  end
```

Define `writer->emphasis` as a function that will transform an emphasized span `s` of input text to the output format.

```
705  function self.emphasis(s)
706    return {"\\markdownRendererEmphasis{",s,"}"}
707  end
```

Define `writer->strong` as a function that will transform a strongly emphasized span `s` of input text to the output format.

```
708  function self.strong(s)
709    return {"\\markdownRendererStrongEmphasis{",s,"}"}
710  end
```

Define `writer->blockquote` as a function that will transform an input block quote `s` to the output format.

```
711  function self.blockquote(s)
712    return {"\\markdownRendererBlockQuoteBegin\n",s,
713      "\n\\markdownRendererBlockQuoteEnd "}
714  end
```

Define `writer->verbatim` as a function that will transform an input code block `s` to the output format.

```
715   function self.verbatim(s)
716     local name = util.cache(options.cacheDir, s, nil, nil, ".verbatim")
717     return {"\\markdownRendererInputVerbatim{",name,"}"}
718   end
```

Define `writer->heading` as a function that will transform an input heading `s` at level `level` to the output format.

```
719   function self.heading(s,level)
720     local cmd
721     if level == 1 then
722       cmd = "\\markdownRendererHeadingOne"
723     elseif level == 2 then
724       cmd = "\\markdownRendererHeadingTwo"
725     elseif level == 3 then
726       cmd = "\\markdownRendererHeadingThree"
727     elseif level == 4 then
728       cmd = "\\markdownRendererHeadingFour"
729     elseif level == 5 then
730       cmd = "\\markdownRendererHeadingFive"
731     elseif level == 6 then
732       cmd = "\\markdownRendererHeadingSix"
733     else
734       cmd = ""
735     end
736     return {cmd,"{",s,"}"}
737   end
```

Define `writer->footnote` as a function that will transform an input footnote `s` to the output format.

```
738   function self.note(s)
739     return {"\\markdownRendererFootnote{",s,"}"}
740   end
741
742   return self
743 end
```

3.1.3 Markdown Reader

This section documents the `reader` object, which implements the routines for parsing the markdown input. The object corresponds to the markdown reader object that was located in the `lunamark/reader/markdown.lua` file in the Lunamark Lua module.

Although not specified in the Lua interface (see Section 2.1), the `reader` object is exported, so that the curious user could easily tinker with the methods of the objects

produced by the `reader.new` method described below. The user should be aware, however, that the implementation may change in a future revision.

The `reader.new` method creates and returns a new TEX reader object associated with the Lua interface options (see Section 2.1.2) `options` and with a writer object `writer`. When `options` are unspecified, it is assumed that an empty table was passed to the method.

The objects produced by the `reader.new` method expose instance methods and variables of their own. As a convention, I will refer to these ⟨*member*⟩s as `reader->`⟨*member*⟩.

```
M.reader = {}
function M.reader.new(writer, options)
  local self = {}
  options = options or {}
```

Make the `options` table inherit from the `defaultOptions` table.

```
  setmetatable(options, { __index = function (_, key)
    return defaultOptions[key] end })
```

3.1.3.1 Top Level Helper Functions

Define `normalize_tag` as a function that normalizes a markdown reference tag by lowercasing it, and by collapsing any adjacent whitespace characters.

```
  local function normalize_tag(tag)
    return unicode.utf8.lower(
      gsub(util.rope_to_string(tag), "[ \n\r\t]+", " "))
  end
```

Define `expandtabs` either as an identity function, when the `preserveTabs` Lua inrerface option is `true`, or to a function that expands tabs into spaces otherwise.

```
  local expandtabs
  if options.preserveTabs then
    expandtabs = function(s) return s end
  else
    expandtabs = function(s)
                   if s:find("\t") then
                     return s:gsub("[^\n]*", util.expand_tabs_in_line)
                   else
                     return s
                   end
                 end
  end
```

3.1.3.2 Top Level Parsing Functions

```
  local syntax
  local blocks
  local inlines
```

```
769
770 local parse_blocks =
771   function(str)
772     local res = lpeg.match(blocks, str)
773     if res == nil then
774       error(format("parse_blocks failed on:\n%s", str:sub(1,20)))
775     else
776       return res
777     end
778   end
779
780 local parse_inlines =
781   function(str)
782     local res = lpeg.match(inlines, str)
783     if res == nil then
784       error(format("parse_inlines failed on:\n%s",
785       str:sub(1,20)))
786     else
787       return res
788     end
789   end
790
791 local parse_inlines_no_link =
792   function(str)
793     local res = lpeg.match(inlines_no_link, str)
794     if res == nil then
795       error(format("parse_inlines_no_link failed on:\n%s",
796       str:sub(1,20)))
797     else
798       return res
799     end
800   end
```

3.1.3.3 Generic PEG Patterns

```
801 local percent               = P("%")
802 local asterisk              = P("*")
803 local dash                  = P("-")
804 local plus                  = P("+")
805 local underscore            = P("_")
806 local period                = P(".")
807 local hash                  = P("#")
808 local ampersand             = P("&")
809 local backtick              = P("`")
810 local less                  = P("<")
811 local more                  = P(">")
812 local space                 = P(" ")
```

```
local squote                = P("’")
local dquote                = P(’"’)
local lparent               = P("(")
local rparent               = P(")")
local lbracket              = P("[")
local rbracket              = P("]")
local circumflex            = P("^")
local slash                 = P("/")
local equal                 = P("=")
local colon                 = P(":")
local semicolon             = P(";")
local exclamation           = P("!")

local digit                 = R("09")
local hexdigit              = R("09","af","AF")
local letter                = R("AZ","az")
local alphanumeric          = R("AZ","az","09")
local keyword               = letter * alphanumeric^0

local doubleasterisks       = P("**")
local doubleunderscores     = P("__")
local fourspaces            = P("    ")

local any                   = P(1)
local fail                  = any - 1
local always                = P("")

local escapable             = S("\\‘*_{}[]()+_.!<>#-~:^")
local anyescaped            = P("\\") / "" * escapable
                            + any

local tab                   = P("\t")
local spacechar             = S("\t ")
local spacing               = S(" \n\r\t")
local newline               = P("\n")
local nonspacechar          = any - spacing
local tightblocksep         = P("\001")

local specialchar
if options.smartEllipses then
  specialchar               = S("*_‘&[]<!\\.")
else
  specialchar               = S("*_‘&[]<!\\")
end

local normalchar            = any -
                              (specialchar + spacing + tightblocksep)
```

```
local optionalspace          = spacechar^0
local spaces                 = spacechar^1
local eof                    = - any
local nonindentspace         = space^-3 * - spacechar
local indent                 = space^-3 * tab
                             + fourspaces / ""
local linechar               = P(1 - newline)

local blankline              = optionalspace * newline / "\n"
local blanklines             = blankline^0
local skipblanklines         = (optionalspace * newline)^0
local indentedline           = indent    /"" * C(linechar^1 * newline^-1)
local optionallyindentedline = indent^-1 /"" * C(linechar^1 * newline^-1)
local sp                     = spacing^0
local spnl                   = optionalspace * (newline * optionalspace)^-1
local line                   = linechar^0 * newline
                             + linechar^1 * eof
local nonemptyline           = line - blankline

local chunk = line * (optionallyindentedline - blankline)^0

-- block followed by 0 or more optionally
-- indented blocks with first line indented.
local function indented_blocks(bl)
  return Cs( bl
           * (blankline^1 * indent * -blankline * bl)^0
           * blankline^1 )
end
```

3.1.3.4 List PEG Patterns

```
local bulletchar = C(plus + asterisk + dash)

local bullet     = ( bulletchar * #spacing * (tab + space^-3)
                   + space * bulletchar * #spacing * (tab + space^-2)
                   + space * space * bulletchar * #spacing * (tab + space^-1)
                   + space * space * space * bulletchar * #spacing
                   ) * -bulletchar

if options.hashEnumerators then
  dig = digit + hash
else
  dig = digit
end

local enumerator = C(dig^3 * period) * #spacing
                 + C(dig^2 * period) * #spacing * (tab + space^1)
```

```
                  + C(dig * period) * #spacing * (tab + space^-2)
                  + space * C(dig^2 * period) * #spacing
                  + space * C(dig * period) * #spacing * (tab + space^-1)
                  + space * space * C(dig^1 * period) * #spacing
```

3.1.3.5 Code Span PEG Patterns

```
local openticks   = Cg(backtick^1, "ticks")

local function captures_equal_length(s,i,a,b)
  return #a == #b and i
end

local closeticks  = space^-1 *
                    Cmt(C(backtick^1) * Cb("ticks"), captures_equal_length)

local intickschar = (any - S(" \n\r`"))
                  + (newline * -blankline)
                  + (space - closeticks)
                  + (backtick^1 - closeticks)

local inticks     = openticks * space^-1 * C(intickschar^1) * closeticks
```

3.1.3.6 Tag PEG Patterns

```
local leader        = space^-3

-- in balanced brackets, parentheses, quotes:
local bracketed     = P{ lbracket
                         * ((anyescaped - (lbracket + rbracket
                             + blankline^2)) + V(1))^0
                         * rbracket }

local inparens      = P{ lparent
                         * ((anyescaped - (lparent + rparent
                             + blankline^2)) + V(1))^0
                         * rparent }

local squoted       = P{ squote * alphanumeric
                         * ((anyescaped - (squote + blankline^2))
                             + V(1))^0
                         * squote }

local dquoted       = P{ dquote * alphanumeric
                         * ((anyescaped - (dquote + blankline^2))
                             + V(1))^0
                         * dquote }

```

```
-- bracketed 'tag' for markdown links, allowing nested brackets:
local tag           = lbracket
                    * Cs((alphanumeric^1
                         + bracketed
                         + inticks
                         + (anyescaped - (rbracket + blankline^2)))^0)
                    * rbracket

-- url for markdown links, allowing balanced parentheses:
local url           = less * Cs((anyescaped-more)^0) * more
                    + Cs((inparens + (anyescaped-spacing-rparent))^1)

-- quoted text possibly with nested quotes:
local title_s       = squote * Cs(((anyescaped-squote) + squoted)^0) *
                      squote

local title_d       = dquote * Cs(((anyescaped-dquote) + dquoted)^0) *
                      dquote

local title_p       = lparent
                    * Cs((inparens + (anyescaped-rparent))^0)
                    * rparent

local title         = title_d + title_s + title_p

local optionaltitle = spnl * title * spacechar^0
                    + Cc("")
```

3.1.3.7 Footnote PEG Patterns

```
local rawnotes = {}

local function strip_first_char(s)
  return s:sub(2)
end

-- like indirect_link
local function lookup_note(ref)
  return function()
    local found = rawnotes[normalize_tag(ref)]
    if found then
      return writer.note(parse_blocks(found))
    else
      return {"[", parse_inlines("^" .. ref), "]"}
    end
  end
end
```

```
990
991   local function register_note(ref,rawnote)
992     rawnotes[normalize_tag(ref)] = rawnote
993     return ""
994   end
995
996   local RawNoteRef = #(lbracket * circumflex) * tag / strip_first_char
997
998   local NoteRef    = RawNoteRef / lookup_note
999
1000  local NoteBlock
1001
1002  if options.footnotes then
1003    NoteBlock = leader * RawNoteRef * colon * spnl *
1004                indented_blocks(chunk) / register_note
1005  else
1006    NoteBlock = fail
1007  end
```

3.1.3.8 Link and Image PEG Patterns

```
1008  -- List of references defined in the document
1009  local references
1010
1011  -- add a reference to the list
1012  local function register_link(tag,url,title)
1013      references[normalize_tag(tag)] = { url = url, title = title }
1014      return ""
1015  end
1016
1017  -- parse a reference definition:  [foo]: /bar "title"
1018  local define_reference_parser =
1019    leader * tag * colon * spacechar^0 * url * optionaltitle * blankline^1
1020
1021  -- lookup link reference and return either
1022  -- the link or nil and fallback text.
1023  local function lookup_reference(label,sps,tag)
1024      local tagpart
1025      if not tag then
1026          tag = label
1027          tagpart = ""
1028      elseif tag == "" then
1029          tag = label
1030          tagpart = "[]"
1031      else
1032          tagpart = {"[", parse_inlines(tag), "]"}
1033      end
```

```
      if sps then
        tagpart = {sps, tagpart}
      end
      local r = references[normalize_tag(tag)]
      if r then
        return r
      else
        return nil, {"[", parse_inlines(label), "]", tagpart}
      end
  end

  -- lookup link reference and return a link, if the reference is found,
  -- or a bracketed label otherwise.
  local function indirect_link(label,sps,tag)
    return function()
      local r,fallback = lookup_reference(label,sps,tag)
      if r then
        return writer.link(parse_inlines_no_link(label), r.url, r.title)
      else
        return fallback
      end
    end
  end

  -- lookup image reference and return an image, if the reference is found,
  -- or a bracketed label otherwise.
  local function indirect_image(label,sps,tag)
    return function()
      local r,fallback = lookup_reference(label,sps,tag)
      if r then
        return writer.image(writer.string(label), r.url, r.title)
      else
        return {"!", fallback}
      end
    end
  end
```

3.1.3.9 Inline Element PEG Patterns

```
  local Inline    = V("Inline")

  local Str       = normalchar^1 / writer.string

  local Ellipsis  = P("...") / writer.ellipsis

  local Smart     = Ellipsis

```

```
local Symbol    = (specialchar - tightblocksep) / writer.string

local Code      = inticks / writer.code

local bqstart      = more
local headerstart  = hash
                   + (line * (equal^1 + dash^1) * optionalspace * newline)

if options.blankBeforeBlockquote then
  bqstart = fail
end

if options.blankBeforeHeading then
  headerstart = fail
end

local Endline   = newline * -( -- newline, but not before...
                      blankline -- paragraph break
                    + tightblocksep  -- nested list
                    + eof       -- end of document
                    + bqstart
                    + headerstart
                  ) * spacechar^0 / writer.space

```

Make two and more trailing spaces before a newline produce a forced line break, throw away one or more trailing spaces and an optional newline at the end of a file, and reduce one or more spaces and an optional newline into a single space.

```
local Space     = spacechar^2 * Endline / writer.linebreak
                + spacechar^1 * Endline^-1 * eof / ""
                + spacechar^1 * Endline^-1 * optionalspace / writer.space

-- parse many p between starter and ender
local function between(p, starter, ender)
    local ender2 = B(nonspacechar) * ender
    return (starter * #nonspacechar * Ct(p * (p - ender2)^0) * ender2)
end

local Strong = ( between(Inline, doubleasterisks, doubleasterisks)
               + between(Inline, doubleunderscores, doubleunderscores)
               ) / writer.strong

local Emph   = ( between(Inline, asterisk, asterisk)
               + between(Inline, underscore, underscore)
               ) / writer.emphasis

local urlchar = anyescaped - newline - more
```

```
1121
1122  local AutoLinkUrl   = less
1123                      * C(alphanumeric^1 * P("://") * urlchar^1)
1124                      * more
1125                      / function(url)
1126                        return writer.link(writer.string(url), url)
1127                      end
1128
1129  local AutoLinkEmail = less
1130                      * C((alphanumeric + S("-._+"))^1 * P("@") * urlchar^1)
1131                      * more
1132                      / function(email)
1133                        return writer.link(writer.string(email),
1134                                           "mailto:"..email)
1135                      end
1136
1137  local DirectLink    = (tag / parse_inlines_no_link)  -- no links inside links
1138                      * spnl
1139                      * lparent
1140                      * (url + Cc(""))  -- link can be empty [foo]()
1141                      * optionaltitle
1142                      * rparent
1143                      / writer.link
1144
1145  local IndirectLink = tag * (C(spnl) * tag)^-1 / indirect_link
1146
1147  -- parse a link or image (direct or indirect)
1148  local Link          = DirectLink + IndirectLink
1149
1150  local DirectImage   = exclamation
1151                      * (tag / parse_inlines)
1152                      * spnl
1153                      * lparent
1154                      * (url + Cc(""))  -- link can be empty [foo]()
1155                      * optionaltitle
1156                      * rparent
1157                      / writer.image
1158
1159  local IndirectImage  = exclamation * tag * (C(spnl) * tag)^-1 /
1160                         indirect_image
1161
1162  local Image         = DirectImage + IndirectImage
1163
1164  -- avoid parsing long strings of * or _ as emph/strong
1165  local UlOrStarLine  = asterisk^4 + underscore^4 / writer.string
1166
1167  local EscapedChar   = S("\\") * C(escapable) / writer.string
```

3.1.3.10 Block Element PEG Patterns

```
local Block          = V("Block")

local Verbatim       = Cs( (blanklines
                         * ((indentedline - blankline))^1)^1
                         ) / expandtabs / writer.verbatim

-- strip off leading > and indents, and run through blocks
local Blockquote     = Cs((
          ((leader * more * space^-1)/"" * linechar^0 * newline)^1
        * (-blankline * linechar^1 * newline)^0
        * blankline^0
        )^1) / parse_blocks / writer.blockquote

local function lineof(c)
    return (leader * (P(c) * optionalspace)^3 * newline * blankline^1)
end

local HorizontalRule = ( lineof(asterisk)
                       + lineof(dash)
                       + lineof(underscore)
                       ) / writer.hrule

local Reference      = define_reference_parser / register_link

local Paragraph      = nonindentspace * Ct(Inline^1) * newline
                     * ( blankline^1
                       + #hash
                       + #(leader * more * space^-1)
                       )
                     / writer.paragraph

local Plain          = nonindentspace * Ct(Inline^1) / writer.plain
```

3.1.3.11 List PEG Patterns

```
local starter = bullet + enumerator

-- we use \001 as a separator between a tight list item and a
-- nested list under it.
local NestedList            = Cs((optionallyindentedline - starter)^1)
                            / function(a) return "\001"..a end

local ListBlockLine         = optionallyindentedline
                              - blankline - (indent^-1 * starter)

local ListBlock             = line * ListBlockLine^0
```

```
1211
1212  local ListContinuationBlock = blanklines * (indent / "") * ListBlock
1213
1214  local function TightListItem(starter)
1215      return -HorizontalRule
1216             * (Cs(starter / "" * ListBlock * NestedList^-1) /
1217                parse_blocks)
1218             * -(blanklines * indent)
1219  end
1220
1221  local function LooseListItem(starter)
1222      return -HorizontalRule
1223             * Cs( starter / "" * ListBlock * Cc("\n")
1224               * (NestedList + ListContinuationBlock^0)
1225               * (blanklines / "\n\n")
1226               ) / parse_blocks
1227  end
1228
1229  local BulletList = ( Ct(TightListItem(bullet)^1)
1230                       * Cc(true) * skipblanklines * -bullet
1231                     + Ct(LooseListItem(bullet)^1)
1232                       * Cc(false) * skipblanklines ) /
1233                         writer.bulletlist
1234
1235  local function orderedlist(items,tight,startNumber)
1236    if options.startNumber then
1237      startNumber = tonumber(startNumber) or 1  -- fallback for '#'
1238    else
1239      startNumber = nil
1240    end
1241    return writer.orderedlist(items,tight,startNumber)
1242  end
1243
1244  local OrderedList = Cg(enumerator, "listtype") *
1245                      ( Ct(TightListItem(Cb("listtype")) *
1246                           TightListItem(enumerator)^0)
1247                        * Cc(true) * skipblanklines * -enumerator
1248                      + Ct(LooseListItem(Cb("listtype")) *
1249                           LooseListItem(enumerator)^0)
1250                        * Cc(false) * skipblanklines
1251                      ) * Cb("listtype") / orderedlist
1252
1253  local defstartchar = S("~:")
1254  local defstart     = ( defstartchar * #spacing * (tab + space^-3)
1255                       + space * defstartchar * #spacing * (tab + space^-2)
1256                       + space * space * defstartchar * #spacing *
1257                         (tab + space^-1)
```

```
                    + space * space * space * defstartchar * #spacing
                    )

local dlchunk = Cs(line * (indentedline - blankline)^0)

local function definition_list_item(term, defs, tight)
  return { term = parse_inlines(term), definitions = defs }
end

local DefinitionListItemLoose = C(line) * skipblanklines
                          * Ct((defstart *
                                indented_blocks(dlchunk) /
                                parse_blocks)^1)
                          * Cc(false)
                          / definition_list_item

local DefinitionListItemTight = C(line)
                          * Ct((defstart * dlchunk /
                                           parse_blocks)^1)
                          * Cc(true)
                          / definition_list_item

local DefinitionList =  ( Ct(DefinitionListItemLoose^1) * Cc(false)
                        +  Ct(DefinitionListItemTight^1)
                           * (skipblanklines *
                              -DefinitionListItemLoose * Cc(true))
                        ) / writer.definitionlist
```

3.1.3.12 Blank PEG Patterns

```
local Blank          = blankline / ""
                     + NoteBlock
                     + Reference
                     + (tightblocksep / "\n")
```

3.1.3.13 Heading PEG Patterns

```
-- parse Atx heading start and return level
local HeadingStart = #hash * C(hash^-6) * -hash / length

-- parse setext header ending and return level
local HeadingLevel = equal^1 * Cc(1) + dash^1 * Cc(2)

local function strip_atx_end(s)
  return s:gsub("[#%s]*\n$","")
end

-- parse atx header
```

```
local AtxHeading = Cg(HeadingStart,"level")
                   * optionalspace
                   * (C(line) / strip_atx_end / parse_inlines)
                   * Cb("level")
                   / writer.heading

-- parse setext header
local SetextHeading = #(line * S("=-"))
                   * Ct(line / parse_inlines)
                   * HeadingLevel
                   * optionalspace * newline
                   / writer.heading
```

3.1.3.14 Top Level PEG Specification

```
syntax =
  { "Blocks",

    Blocks                = Blank^0 *
                            Block^-1 *
                            (Blank^0 / function()
                              return writer.interblocksep
                             end * Block)^0 *
                            Blank^0 *
                            eof,

    Blank                 = Blank,

    Block                 = V("Blockquote")
                          + V("Verbatim")
                          + V("HorizontalRule")
                          + V("BulletList")
                          + V("OrderedList")
                          + V("Heading")
                          + V("DefinitionList")
                          + V("Paragraph")
                          + V("Plain"),

    Blockquote            = Blockquote,
    Verbatim              = Verbatim,
    HorizontalRule        = HorizontalRule,
    BulletList            = BulletList,
    OrderedList           = OrderedList,
    Heading               = AtxHeading + SetextHeading,
    DefinitionList        = DefinitionList,
    DisplayHtml           = DisplayHtml,
    Paragraph             = Paragraph,
```

```
      Plain                = Plain,

      Inline               = V("Str")
                           + V("Space")
                           + V("Endline")
                           + V("UlOrStarLine")
                           + V("Strong")
                           + V("Emph")
                           + V("NoteRef")
                           + V("Link")
                           + V("Image")
                           + V("Code")
                           + V("AutoLinkUrl")
                           + V("AutoLinkEmail")
                           + V("EscapedChar")
                           + V("Smart")
                           + V("Symbol"),

      Str                  = Str,
      Space                = Space,
      Endline              = Endline,
      UlOrStarLine         = UlOrStarLine,
      Strong               = Strong,
      Emph                 = Emph,
      NoteRef              = NoteRef,
      Link                 = Link,
      Image                = Image,
      Code                 = Code,
      AutoLinkUrl          = AutoLinkUrl,
      AutoLinkEmail        = AutoLinkEmail,
      InlineHtml           = InlineHtml,
      HtmlEntity           = HtmlEntity,
      EscapedChar          = EscapedChar,
      Smart                = Smart,
      Symbol               = Symbol,
    }

  if not options.definitionLists then
    syntax.DefinitionList = fail
  end

  if not options.footnotes then
    syntax.NoteRef = fail
  end

  if not options.smartEllipses then
    syntax.Smart = fail
```

```
  end

  blocks = Ct(syntax)

  local inlines_t = util.table_copy(syntax)
  inlines_t[1] = "Inlines"
  inlines_t.Inlines = Inline^0 * (spacing^0 * eof / "")
  inlines = Ct(inlines_t)

  inlines_no_link_t = util.table_copy(inlines_t)
  inlines_no_link_t.Link = fail
  inlines_no_link = Ct(inlines_no_link_t)
```

3.1.3.15 Exported Conversion Function

Define `reader->convert` as a function that converts markdown string `input` into a plain TEX output and returns it. Note that the converter assumes that the input has UNIX line endings.

```
  function self.convert(input)
    references = {}
```

When determining the name of the cache file, create salt for the hashing function out of the passed options recognized by the Lua interface (see Section 2.1.2). The `cacheDir` option is disregarded.

```
    local opt_string = {}
    for k,_ in pairs(defaultOptions) do
      local v = options[k]
      if k ~= "cacheDir" then
        opt_string[#opt_string+1] = k .. "=" .. tostring(v)
      end
    end
    table.sort(opt_string)
    local salt = table.concat(opt_string, ",")
```

Produce the cache file, transform its filename via the `writer->pack` method, and return the result.

```
    local name = util.cache(options.cacheDir, input, salt, function(input)
        return util.rope_to_string(parse_blocks(input)) .. writer.eof
      end, ".md" .. writer.suffix)
    return writer.pack(name)
  end
  return self
end
```

3.1.4 Conversion from Markdown to Plain TEX

The `new` method returns the `reader->convert` function of a reader object associated with the Lua interface options (see Section 2.1.2) `options` and with a writer object associated with `options`.

```
function M.new(options)
  local writer = M.writer.new(options)
  local reader = M.reader.new(writer, options)
  return reader.convert
end

return M
```

3.2 Plain TEX Implementation

The plain TEX implementation provides macros for the interfacing between TEX and Lua and for the buffering of input text. These macros are then used to implement the macros for the conversion from markdown to plain TEX exposed by the plain TEX interface (see Section 2.2).

3.2.1 Logging Facilities

```
\def\markdownInfo#1{%
  \message{(l.\the\inputlineno) markdown.tex info: #1.}}%
\def\markdownWarning#1{%
  \message{(l.\the\inputlineno) markdown.tex warning: #1}}%
\def\markdownError#1#2{%
  \errhelp{#2.}%
  \errmessage{(l.\the\inputlineno) markdown.tex error: #1}}%
```

3.2.2 Token Renderers

The following definitions should be considered placeholder.

```
\def\markdownRendererLineBreakPrototype{\hfil\break}%
\let\markdownRendererEllipsisPrototype\dots
\def\markdownRendererLeftBracePrototype{\char`{}%
\def\markdownRendererRightBracePrototype{\char`}}%
\def\markdownRendererDollarSignPrototype{\char`$}%
\def\markdownRendererPercentSignPrototype{\char`\%}%
\def\markdownRendererAmpersandPrototype{\char`&}%
\def\markdownRendererUnderscorePrototype{\char`_}%
\def\markdownRendererHashPrototype{\char`\#}%
\def\markdownRendererCircumflexPrototype{\char`^}%
\def\markdownRendererBackslashPrototype{\char`\\}%
\def\markdownRendererTildePrototype{\char`~}%
\def\markdownRendererPipePrototype{|}%
```

```
\long\def\markdownRendererCodeSpanPrototype#1{{\tt#1}}%
\long\def\markdownRendererLinkPrototype#1#2#3#4{#2}%
\long\def\markdownRendererImagePrototype#1#2#3#4{#2}%
\def\markdownRendererUlBeginPrototype{}%
\def\markdownRendererUlBeginTightPrototype{}%
\def\markdownRendererUlItemPrototype{}%
\def\markdownRendererUlEndPrototype{}%
\def\markdownRendererUlEndTightPrototype{}%
\def\markdownRendererOlBeginPrototype{}%
\def\markdownRendererOlBeginTightPrototype{}%
\def\markdownRendererOlItemPrototype{}%
\long\def\markdownRendererOlItemWithNumberPrototype#1{}%
\def\markdownRendererOlEndPrototype{}%
\def\markdownRendererOlEndTightPrototype{}%
\def\markdownRendererDlBeginPrototype{}%
\def\markdownRendererDlBeginTightPrototype{}%
\long\def\markdownRendererDlItemPrototype#1{#1}%
\def\markdownRendererDlDefinitionBeginPrototype{}%
\def\markdownRendererDlDefinitionEndPrototype{\par}%
\def\markdownRendererDlEndPrototype{}%
\def\markdownRendererDlEndTightPrototype{}%
\long\def\markdownRendererEmphasisPrototype#1{{\it#1}}%
\long\def\markdownRendererStrongEmphasisPrototype#1{{\it#1}}%
\def\markdownRendererBlockQuoteBeginPrototype{\par\begingroup\it}%
\def\markdownRendererBlockQuoteEndPrototype{\endgroup\par}%
\long\def\markdownRendererInputVerbatimPrototype#1{%
\par{\tt\input"#1"\relax}\par}%
\long\def\markdownRendererHeadingOnePrototype#1{#1}%
\long\def\markdownRendererHeadingTwoPrototype#1{#1}%
\long\def\markdownRendererHeadingThreePrototype#1{#1}%
\long\def\markdownRendererHeadingFourPrototype#1{#1}%
\long\def\markdownRendererHeadingFivePrototype#1{#1}%
\long\def\markdownRendererHeadingSixPrototype#1{#1}%
\def\markdownRendererHorizontalRulePrototype{}%
\long\def\markdownRendererFootnotePrototype#1{#1}%
```

3.2.3 Lua Snippets

The `\markdownLuaOptions` macro expands to a Lua table that contains the plain TEX options (see Section 2.2.2) in a format recognized by Lua (see Section 2.1.2). Note that the boolean options are not sanitized and expect the plain TEX option macros to expand to either `true` or `false`.

```
\def\markdownLuaOptions{{%
\ifx\markdownOptionBlankBeforeBlockquote\undefined\else
  blankBeforeBlockquote = \markdownOptionBlankBeforeBlockquote,
\fi
```

```
\ifx\markdownOptionBlankBeforeHeading\undefined\else
  blankBeforeHeading = \markdownOptionBlankBeforeHeading,
\fi
\ifx\markdownOptionCacheDir\undefined\else
  cacheDir = "\markdownOptionCacheDir",
\fi
\ifx\markdownOptionDefinitionLists\undefined\else
  definitionLists = \markdownOptionDefinitionLists,
\fi
\ifx\markdownOptionHashEnumerators\undefined\else
  hashEnumerators = \markdownOptionHashEnumerators,
\fi
\ifx\markdownOptionHybrid\undefined\else
  hybrid = \markdownOptionHybrid,
\fi
\ifx\markdownOptionFootnotes\undefined\else
  footnotes = \markdownOptionFootnotes,
\fi
\ifx\markdownOptionPreserveTabs\undefined\else
  preserveTabs = \markdownOptionPreserveTabs,
\fi
\ifx\markdownOptionSmartEllipses\undefined\else
  smartEllipses = \markdownOptionSmartEllipses,
\fi
\ifx\markdownOptionStartNumber\undefined\else
  startNumber = \markdownOptionStartNumber,
\fi
\ifx\markdownOptionTightLists\undefined\else
  tightLists = \markdownOptionTightLists,
\fi}
}%
```

The `\markdownPrepare` macro contains the Lua code that is executed prior to any conversion from markdown to plain TEX. It exposes the `convert` function for the use by any further Lua code.

```
\def\markdownPrepare{%
```

First, ensure that the `\markdownOptionCacheDir` directory exists.

```
local lfs = require("lfs")
local cacheDir = "\markdownOptionCacheDir"
if lfs.isdir(cacheDir) == true then else
  assert(lfs.mkdir(cacheDir))
end
```

Next, load the `markdown` module and create a converter function using the plain TEX options, which were serialized to a Lua table via the `\markdownLuaOptions` macro.

```
local md = require("markdown")
local convert = md.new(\markdownLuaOptions)
```

```
}%
```

3.2.4 Lua Shell Escape Bridge

The following TEX code is intended for TEX engines that do not provide direct access to Lua, but expose the shell of the operating system. This corresponds to the `\markdownMode` values of `0` and `1`.

The `\markdownLuaExecute` and `\markdownReadAndConvert` macros defined here and in Section 3.2.5 are meant to be transparent to the remaining code.

The package assumes that although the user is not using the LuaTEX engine, their TeX distribution contains it, and uses shell access to produce and execute Lua scripts using the TEXLua interpreter (see [1, Section 3.1.1]).

```
\ifnum\markdownMode<2\relax
\ifnum\markdownMode=0\relax
  \markdownInfo{Using mode 0: Shell escape via write18}%
\else
  \markdownInfo{Using mode 1: Shell escape via os.execute}%
\fi
```

The macro `\markdownLuaExecuteFileStream` contains the number of the output file stream that will be used to store the helper Lua script in the file named `\markdownOptionHelperScriptFileName` during the expansion of the macro `\markdownLuaExecute`, and to store the markdown input in the file named `\markdownOptionInputTempFileName` during the expansion of the macro `\markdownReadAndConvert`.

```
\csname newwrite\endcsname\markdownLuaExecuteFileStream
```

The `\markdownExecuteShellEscape` macro contains the numeric value indicating whether the shell access is enabled (`1`), disabled (`0`), or restricted (`2`).

If Lua is unavailable, inherit the value of the the `\pdfshellescape` (LuaTEX, PdfTEX) or the `\shellescape` (XETEX) commands. If neither of these commands is defined, act as if the shell access were enabled.

```
\ifnum\markdownMode=0\relax
  \ifx\pdfshellescape\undefined
    \ifx\shellescape\undefined
      \def\markdownExecuteShellEscape{1}%
    \else
      \let\markdownExecuteShellEscape\shellescape
    \fi
  \else
    \let\markdownExecuteShellEscape\pdfshellescape
  \fi
```

If Lua is available, inherit the value of the `status.shell_escape` configuration item.

```
\else
```

```
  \def\markdownExecuteShellEscape{%
    \directlua{tex.sprint(status.shell_escape)}}%
\fi
```

The `\markdownExecuteDirect` macro executes the code it has received as its first argument by writing it to the output file stream 18, if Lua is unavailable, or by using the Lua `markdown.execute` method otherwise.

```
\ifnum\markdownMode=0\relax
  \def\markdownExecuteDirect#1{\immediate\write18{#1}}%
\else
  \def\markdownExecuteDirect#1{%
    \directlua{os.execute("\luaescapestring{#1}")}}%
\fi
```

The `\markdownExecute` macro is a wrapper on top of `\markdownExecuteDirect` that checks the value of `\markdownExecuteShellEscape` and prints an error message if the shell is inaccessible.

```
\def\markdownExecute#1{%
  \ifnum\markdownExecuteShellEscape=1\relax
    \markdownExecuteDirect{#1}%
  \else
    \markdownError{I can not access the shell}{Either run the TeX
      compiler with the --shell-escape or the --enable-write18 flag,
      or set shell_escape=t in the texmf.cnf file}%
  \fi}%
```

The `\markdownLuaExecute` macro executes the Lua code it has received as its first argument. The Lua code may not directly interact with the TEX engine, but it can use the `print` function in the same manner it would use the `tex.print` method.

```
\def\markdownLuaExecute#1{%
```

Create the file `\markdownOptionHelperScriptFileName` and fill it with the input Lua code prepended with kpathsea initialization, so that Lua modules from the TEX distribution are available.

```
  \immediate\openout\markdownLuaExecuteFileStream=%
    \markdownOptionHelperScriptFileName
  \markdownInfo{Writing a helper Lua script to the file
    "\markdownOptionHelperScriptFileName"}%
  \immediate\write\markdownLuaExecuteFileStream{%
    local kpse = require('kpse')
    kpse.set_program_name('luatex') #1}%
  \immediate\closeout\markdownLuaExecuteFileStream
```

Execute the generated `\markdownOptionHelperScriptFileName` Lua script using the TEXLua binary and store the output in the `\markdownOptionOutputTempFileName` file.

```
  \markdownInfo{Executing a helper Lua script from the file
    "\markdownOptionHelperScriptFileName" and storing the result in the
```

```
    file "\markdownOptionOutputTempFileName"}%
  \markdownExecute{texlua "\markdownOptionHelperScriptFileName" >
    "\markdownOptionOutputTempFileName"}%
```

`\input` the generated `\markdownOptionOutputTempFileName` file.

```
  \input\markdownOptionOutputTempFileName\relax}%
```

The `\markdownReadAndConvertTab` macro contains the tab character literal.

```
\begingroup
  \catcode`\^^I=12%
  \gdef\markdownReadAndConvertTab{^^I}%
\endgroup
```

The `\markdownReadAndConvert` macro is largely a rewrite of the LaTeX 2_ε `\filecontents` macro to plain TeX.

```
\begingroup
```

Make the newline and tab characters active and swap the character codes of the backslash symbol (`\`) and the pipe symbol (`|`), so that we can use the backslash as an ordinary character inside the macro definition.

```
  \catcode`\^^M=13%
  \catcode`\^^I=13%
  \catcode`|=0%
  \catcode`\\=12%
  |gdef|markdownReadAndConvert#1#2{%
    |begingroup%
```

Open the `\markdownOptionInputTempFileName` file for writing.

```
    |immediate|openout|markdownLuaExecuteFileStream%
      |markdownOptionInputTempFileName%
    |markdownInfo{Buffering markdown input into the temporary %
      input file "|markdownOptionInputTempFileName" and scanning %
      for the closing token sequence "#1"}%
```

Locally change the category of the special plain TeX characters to *other* in order to prevent unwanted interpretation of the input. Change also the category of the space character, so that we can retrieve it unaltered.

```
    |def|do##1{|catcode`##1=12}|dospecials%
    |catcode`| =12%
    |markdownMakeOther%
```

The `\markdownReadAndConvertProcessLine` macro will process the individual lines of output. Note the use of the comments to ensure that the entire macro is at a single line and therefore no (active) newline symbols are produced.

```
    |def|markdownReadAndConvertProcessLine##1#1##2#1##3|relax{%
```

When the ending token sequence does not appear in the line, store the line in the `\markdownOptionInputTempFileName` file.

```
      |ifx|relax##3|relax%
```

```
      |immediate|write|markdownLuaExecuteFileStream{##1}%
    |else%
```

When the ending token sequence appears in the line, make the next newline character close the `\markdownOptionInputTempFileName` file, return the character categories back to the former state, convert the `\markdownOptionInputTempFileName` file from markdown to plain TEX, `\input` the result of the conversion, and expand the ending control sequence.

```
      |def^^M{%
        |markdownInfo{The ending token sequence was found}%
        |immediate|write|markdownLuaExecuteFileStream{}%
        |immediate|closeout|markdownLuaExecuteFileStream%
        |endgroup%
        |markdownInput|markdownOptionInputTempFileName%
        #2}%
    |fi%
```

Repeat with the next line.

```
    ^^M}%
```

Make the tab character active at expansion time and make it expand to a literal tab character.

```
  |catcode`|^^I=13%
  |def^^I{|markdownReadAndConvertTab}%
```

Make the newline character active at expansion time and make it consume the rest of the line on expansion. Throw away the rest of the first line and pass the second line to the `\markdownReadAndConvertProcessLine` macro.

```
  |catcode`|^^M=13%
  |def^^M##1^^M{%
    |def^^M####1^^M{%
      |markdownReadAndConvertProcessLine####1#1#1|relax}%
    ^^M}%
  ^^M}%
```

Reset the character categories back to the former state.

```
|endgroup
```

3.2.5 Direct Lua Access

The following TEX code is intended for TEX engines that provide direct access to Lua (LuaTEX). The `\markdownLuaExecute` and `\markdownReadAndConvert` defined here and in Section 3.2.4 are meant to be transparent to the remaining code. This corresponds to the `\markdownMode` value of 2.

```
\else
\markdownInfo{Using mode 2: Direct Lua access}%
```

The direct Lua access version of the `\markdownLuaExecute` macro is defined in terms of the `\directlua` primitive. The `print` function is set as an alias to the `\tex.print` method in order to mimic the behaviour of the `\markdownLuaExecute` definition from Section 3.2.4,

```
\def\markdownLuaExecute#1{\directlua{local print = tex.print #1}}%
```

In the definition of the direct Lua access version of the `\markdownReadAndConvert` macro, we will be using the hash symbol (`#`), the underscore symbol (`_`), the circumflex symbol (`^`), the dollar sign (`$`), the backslash symbol (`\`), the percent sign (`%`), and the braces (`{}`) as a part of the Lua syntax.

```
\begingroup
```

To this end, we will make the underscore symbol, the dollar sign, and circumflex symbols ordinary characters,

```
    \catcode`\_=12%
    \catcode`\$=12%
    \catcode`\^=12%
```

swap the category code of the hash symbol with the slash symbol (`/`).

```
    \catcode`\/=6%
    \catcode`\#=12%
```

swap the category code of the percent sign with the at symbol (`@`).

```
    \catcode`\@=14%
    \catcode`\%=12%
```

swap the category code of the backslash symbol with the pipe symbol (`|`),

```
    \catcode`|=0@
    \catcode`\\=12@
```

Braces are a part of the plain TEX syntax, but they are not removed during expansion, so we do not need to bother with changing their category codes.

```
    |gdef|markdownReadAndConvert/1/2{@
```

Make the `\markdownReadAndConvertAfter` macro store the token sequence that will be inserted into the document after the ending token sequence has been found.

```
      |def|markdownReadAndConvertAfter{/2}@
      |markdownInfo{Buffering markdown input and scanning for the
        closing token sequence "/1"}@
      |directlua{@
```

Set up an empty Lua table that will serve as our buffer.

```
        |markdownPrepare
        local buffer = {}
```

Create a regex that will match the ending input sequence. Escape any special regex characters (like a star inside `\end{markdown*}`) inside the input.

```
        local ending_sequence = "^.-" .. ([[/1]]):gsub(
          "([%(%)%.%%%+%-%*%?%[%]%^%$])", "%%%1")
```

Register a callback that will notify you about new lines of input.

```
      |markdownLuaRegisterIBCallback{function(line)
```

When the ending token sequence appears on a line, unregister the callback, convert the contents of our buffer from markdown to plain TEX, and insert the result into the input line buffer of TEX.

```
        if line:match(ending_sequence) then
          |markdownLuaUnregisterIBCallback
          local input = table.concat(buffer, "\n") .. "\n\n"
          local output = convert(input)
          return [[\markdownInfo{The ending token sequence was found}]] ..
            output .. [[\markdownReadAndConvertAfter]]
```

When the ending token sequence does not appear on a line, store the line in our buffer, and insert either `\fi`, if this is the first line of input, or an empty token list to the input line buffer of TEX.

```
        else
          buffer[#buffer+1] = line
          return [[\]] .. (#buffer == 1 and "fi" or "relax")
        end
      end}}@
```

Insert `\iffalse` after the `\markdownReadAndConvert` macro in order to consume the rest of the first line of input.

```
    |iffalse}@
```

Reset the character categories back to the former state.

```
  |endgroup
\fi
```

3.2.6 Typesetting Markdown

The `\markdownInput` macro uses an implementation of the `\markdownLuaExecute` macro to convert the contents of the file whose filename it has received as its single argument from markdown to plain TEX.

```
\begingroup
```

Swap the category code of the backslash symbol and the pipe symbol, so that we may use the backslash symbol freely inside the Lua code.

```
  \catcode`|=0%
  \catcode`\\=12%
  |gdef|markdownInput#1{%
    |markdownInfo{Including markdown document "#1"}%
    |markdownLuaExecute{%
      |markdownPrepare
      local input = assert(io.open("#1","r")):read("*a") .. "\n\n"
```

Since the Lua converter expects UNIX line endings, normalize the input.

```
      print(convert(input:gsub("\r\n?", "\n")))}}%
|endgroup
```

3.3 LaTeX Implementation

The LaTeX implemenation makes use of the fact that, apart from some subtle differences, LaTeX implements the majority of the plain TeX format (see [4, Section 9]). As a consequence, we can directly reuse the existing plain TeX implementation.

```
\input markdown
\def\markdownVersionSpace{ }%
\ProvidesPackage{markdown}[\markdownLastModified\markdownVersionSpace v%
  \markdownVersion\markdownVersionSpace markdown renderer]%
```

3.3.1 Logging Facilities

The LaTeX implementation redefines the plain TeX logging macros (see Section 3.2.1) to use the LaTeX `\PackageInfo`, `\PackageWarning`, and `\PackageError` macros.

```
\renewcommand\markdownInfo[1]{\PackageInfo{markdown}{#1}}%
\renewcommand\markdownWarning[1]{\PackageWarning{markdown}{#1}}%
\renewcommand\markdownError[2]{\PackageError{markdown}{#1}{#2.}}%
```

3.3.2 Typesetting Markdown

The `\markdownInputPlainTeX` macro is used to store the original plain TeX implementation of the `\markdownInput` macro. The `\markdownInput` is then redefined to accept an optional argument with options recognized by the LaTeX interface (see Section 2.3.2).

```
\let\markdownInputPlainTeX\markdownInput
\renewcommand\markdownInput[2][]{%
  \begingroup
    \markdownSetup{#1}%
    \markdownInputPlainTeX{#2}%
  \endgroup}%
```

The `markdown`, and `markdown*` LaTeX environments are implemented using the `\markdownReadAndConvert` macro.

```
\renewenvironment{markdown}{%
  \markdownReadAndConvert@markdown{}}\relax
\renewenvironment{markdown*}[1]{%
  \markdownSetup{#1}%
  \markdownReadAndConvert@markdown*}\relax
\begingroup
```

Locally swap the category code of the backslash symbol with the pipe symbol, and of the left (`{`) and right brace (`}`) with the less-than (`<`) and greater-than (`>`) signs. This is required in order that all the special symbols that appear in the first argument of the `markdownReadAndConvert` macro have the category code *other*.

```
  \catcode`\|=0\catcode`\<=1\catcode`\>=2%
  \catcode`\\=12|catcode`|{=12|catcode`|}=12%
  |gdef|markdownReadAndConvert@markdown#1<%
    |markdownReadAndConvert<\end{markdown#1}>%
                           <|end<markdown#1>>>%
|endgroup
```

3.3.3 Options

The supplied package options are processed using the `\markdownSetup` macro.

```
\DeclareOption*{%
  \expandafter\markdownSetup\expandafter{\CurrentOption}}%
\ProcessOptions\relax
```

The following configuration should be considered placeholder.

```
\RequirePackage{url}
\RequirePackage{graphicx}
```

If the `\markdownOptionTightLists` macro expands to `false`, do not load the paralist package. This is necessary for LaTeX 2_ε document classes that do not play nice with paralist, such as beamer.

```
\RequirePackage{ifthen}
\ifx\markdownOptionTightLists\undefined
  \RequirePackage{paralist}
\else
  \ifthenelse{\equal{\markdownOptionTightLists}{false}}{}{
    \RequirePackage{paralist}}
\fi
\RequirePackage{fancyvrb}
\markdownSetup{rendererPrototypes={
  lineBreak = {\\},
  leftBrace = {\textbraceleft},
  rightBrace = {\textbraceright},
  dollarSign = {\textdollar},
  underscore = {\textunderscore},
  circumflex = {\textasciicircum},
  backslash = {\textbackslash},
  tilde = {\textasciitilde},
  pipe = {\textbar},
  codeSpan = {\texttt{#1}},
  link = {#1\footnote{\ifx\empty#4\empty\else#4:
    \fi\texttt<\url{#3}\texttt>}},
```

```
  image = {\begin{figure}
      \begin{center}%
        \includegraphics{#3}%
      \end{center}%
      \ifx\empty#4\empty\else
        \caption{#4}%
      \fi
      \label{fig:#1}%
    \end{figure}},
  ulBegin = {\begin{itemize}},
  ulBeginTight = {\begin{compactitem}},
  ulItem = {\item},
  ulEnd = {\end{itemize}},
  ulEndTight = {\end{compactitem}},
  olBegin = {\begin{enumerate}},
  olBeginTight = {\begin{compactenum}},
  olItem = {\item},
  olItemWithNumber = {\item[#1.]},
  olEnd = {\end{enumerate}},
  olEndTight = {\end{compactenum}},
  dlBegin = {\begin{description}},
  dlBeginTight = {\begin{compactdesc}},
  dlItem = {\item[#1]},
  dlEnd = {\end{description}},
  dlEndTight = {\end{compactdesc}},
  emphasis = {\emph{#1}},
  strongEmphasis = {%
    \ifx\alert\undefined
      \textbf{\emph{#1}}%
    \else % Beamer support
      \alert{\emph{#1}}%
    \fi},
  blockQuoteBegin = {\begin{quotation}},
  blockQuoteEnd = {\end{quotation}},
  inputVerbatim = {\VerbatimInput{#1}},
  horizontalRule = {\noindent\rule[0.5ex]{\linewidth}{1pt}},
  footnote = {\footnote{#1}}}}%

\ifx\chapter\undefined
  \markdownSetup{rendererPrototypes={
    headingOne = {\section{#1}},
    headingTwo = {\subsection{#1}},
    headingThree = {\subsubsection{#1}},
    headingFour = {\paragraph{#1}},
    headingFive = {\subparagraph{#1}}}}%
\else
  \markdownSetup{rendererPrototypes={
```

```
    headingOne = {\chapter{#1}},
    headingTwo = {\section{#1}},
    headingThree = {\subsection{#1}},
    headingFour = {\subsubsection{#1}},
    headingFive = {\paragraph{#1}},
    headingSix = {\subparagraph{#1}}}}%
\fi
```

3.3.4 Miscellanea

Unlike base LuaTEX, which only allows for a single registered function per a callback (see [1, Section 8.1]), the LATEX 2ε format disables the `callback.register` method and exposes the `luatexbase.add_to_callback` and `luatexbase.remove_from_callback` methods that enable the user code to hook several functions on a single callback (see [4, Section 73.4]).

To make our code function with the LATEX 2ε format, we need to redefine the `\markdownLuaRegisterIBCallback` and `\markdownLuaUnregisterIBCallback` macros accordingly.

```
\renewcommand\markdownLuaRegisterIBCallback[1]{%
  luatexbase.add_to_callback("process_input_buffer", #1, %
    "The markdown input processor")}
\renewcommand\markdownLuaUnregisterIBCallback{%
  luatexbase.remove_from_callback("process_input_buffer",%
    "The markdown input processor")}
```

When buffering user input, we should disable the bytes with the high bit set, since these are made active by the inputenc package. We will do this by redefining the `\markdownMakeOther` macro accordingly. The code is courtesy of Scott Pakin, the creator of the filecontents package.

```
\newcommand\markdownMakeOther{%
  \count0=128\relax
  \loop
    \catcode\count0=11\relax
    \advance\count0 by 1\relax
  \ifnum\count0<256\repeat}%
```

3.4 ConTEXt Implementation

The ConTEXt implementation makes use of the fact that, apart from some subtle differences, the Mark II and Mark IV ConTEXt formats *seem* to implement (the documentation is scarce) the majority of the plain TEX format required by the plain TEX implementation. As a consequence, we can directly reuse the existing plain TEX implementation after supplying the missing plain TEX macros.

```
\def\dospecials{\do\ \do\\\do\{\do\}\do\$\do\&%
  \do\#\do\^\do\_\do\%\do\~}%
```

When there is no Lua support, then just load the plain TEX implementation.

```
\ifx\directlua\undefined
  \input markdown
\else
```

When there is Lua support, check if we can set the `process_input_buffer` LuaTEX callback.

```
  \directlua{%
    local function unescape(str)
      return (str:gsub("|", string.char(92))) end
    local old_callback = callback.find("process_input_buffer")
    callback.register("process_input_buffer", function() end)
    local new_callback = callback.find("process_input_buffer")
```

If we can not, we are probably using ConTeXt Mark IV. In ConTeXt Mark IV, the `process_input_buffer` callback is currently frozen (inaccessible from the user code) and, due to the lack of available documentation, it is unclear to me how to emulate it. As a workaround, we will force the plain TEX implementation to use the Lua shell escape bridge (see Section 3.2.4) by setting the `\markdownMode` macro to the value of `1`.

```
    if new_callback == false then
      tex.print(unescape([[|def|markdownMode{1}|input markdown]]))
```

If we can set the `process_input_buffer` LuaTEX callback, then just load the plain TEX implementation.

```
    else
      callback.register("process_input_buffer", old_callback)
      tex.print(unescape("|input markdown"))
    end}%
\fi
```

If the shell escape bridge is being used, define the `\markdownMakeOther` macro, so that the pipe character (`|`) is inactive during the scanning. This is necessary, since the character is active in ConTeXt.

```
\ifnum\markdownMode<2
  \def\markdownMakeOther{%
    \catcode`|=12}%
\fi
```

3.4.1 Logging Facilities

The ConTeXt implementation redefines the plain TEX logging macros (see Section 3.2.1) to use the ConTeXt `\writestatus` macro.

```
\def\markdownInfo#1{\writestatus{markdown}{#1.}}%
\def\markdownWarning#1{\writestatus{markdown\space warn}{#1.}}%
```

3.4.2 Typesetting Markdown

The `\startmarkdown` and `\stopmarkdown` macros are implemented using the `\markdownReadAndConvert` macro.

```
\begingroup
```

Locally swap the category code of the backslash symbol with the pipe symbol. This is required in order that all the special symbols that appear in the first argument of the `markdownReadAndConvert` macro have the category code *other*.

```
  \catcode`\|=0%
  \catcode`\\=12%
  |gdef|startmarkdown{%
    |markdownReadAndConvert{\stopmarkdown}%
                           {|stopmarkdown}}%
|endgroup
```

3.4.3 Options

The following configuration should be considered placeholder.

```
\def\markdownRendererLineBreakPrototype{\blank}%
\def\markdownRendererLeftBracePrototype{\textbraceleft}%
\def\markdownRendererRightBracePrototype{\textbraceright}%
\def\markdownRendererDollarSignPrototype{\textdollar}%
\def\markdownRendererPercentSignPrototype{\percent}%
\def\markdownRendererUnderscorePrototype{\textunderscore}%
\def\markdownRendererCircumflexPrototype{\textcircumflex}%
\def\markdownRendererBackslashPrototype{\textbackslash}%
\def\markdownRendererTildePrototype{\textasciitilde}%
\def\markdownRendererPipePrototype{\char`|}%
\long\def\markdownRendererLinkPrototype#1#2#3#4{%
  \useURL[#1][#3][][#4]#1\footnote[#1]{\ifx\empty#4\empty\else#4:
  \fi\tt<\hyphenatedurl{#3}>}}%
\long\def\markdownRendererImagePrototype#1#2#3#4{%
  \placefigure[][fig:#1]{#4}{\externalfigure[#3]}}%
\def\markdownRendererUlBeginPrototype{\startitemize}%
\def\markdownRendererUlBeginTightPrototype{\startitemize[packed]}%
\def\markdownRendererUlItemPrototype{\item}%
\def\markdownRendererUlEndPrototype{\stopitemize}%
\def\markdownRendererUlEndTightPrototype{\stopitemize}%
\def\markdownRendererOlBeginPrototype{\startitemize[n]}%
\def\markdownRendererOlBeginTightPrototype{\startitemize[packed,n]}%
\def\markdownRendererOlItemPrototype{\item}%
\long\def\markdownRendererOlItemWithNumberPrototype#1{\sym{#1.}}%
\def\markdownRendererOlEndPrototype{\stopitemize}%
\def\markdownRendererOlEndTightPrototype{\stopitemize}%
\definedescription
```

```
  [markdownConTeXtDlItemPrototype]
  [location=hanging,
   margin=standard,
   headstyle=bold]%
\definestartstop
  [MarkdownConTeXtDlPrototype]
  [before=\blank,
   after=\blank]%
\definestartstop
  [MarkdownConTeXtDlTightPrototype]
  [before=\blank\startpacked,
   after=\stoppacked\blank]%
\def\markdownRendererDlBeginPrototype{%
  \startMarkdownConTeXtDlPrototype}%
\def\markdownRendererDlBeginTightPrototype{%
  \startMarkdownConTeXtDlTightPrototype}%
\long\def\markdownRendererDlItemPrototype#1{%
  \markdownConTeXtDlItemPrototype{#1}}%
\def\markdownRendererDlEndPrototype{%
  \stopMarkdownConTeXtDlPrototype}%
\def\markdownRendererDlEndTightPrototype{%
  \stopMarkdownConTeXtDlTightPrototype}%
\long\def\markdownRendererEmphasisPrototype#1{{\em#1}}%
\long\def\markdownRendererStrongEmphasisPrototype#1{{\bf\em#1}}%
\def\markdownRendererBlockQuoteBeginPrototype{\startquotation}%
\def\markdownRendererBlockQuoteEndPrototype{\stopquotation}%
\long\def\markdownRendererInputVerbatimPrototype#1{\typefile{#1}}%
\long\def\markdownRendererHeadingOnePrototype#1{\chapter{#1}}%
\long\def\markdownRendererHeadingTwoPrototype#1{\section{#1}}%
\long\def\markdownRendererHeadingThreePrototype#1{\subsection{#1}}%
\long\def\markdownRendererHeadingFourPrototype#1{\subsubsection{#1}}%
\long\def\markdownRendererHeadingFivePrototype#1{\subsubsubsection{#1}}%
\long\def\markdownRendererHeadingSixPrototype#1{\subsubsubsubsection{#1}}%
\def\markdownRendererHorizontalRulePrototype{%
  \blackrule[height=1pt, width=\hsize]}%
\long\def\markdownRendererFootnotePrototype#1{\footnote{#1}}%
\stopmodule\protect
```

References

1. LUATEX DEVELOPMENT TEAM. *LuaTEX reference manual (0.95.0)* [online] [visited on 2016-05-12]. Available from: http://www.luatex.org/svn/trunk/manual/luatex.pdf.
2. KNUTH, Donald Ervin. *The TEXbook*. 3rd ed. Addison-Westley, 1986. ISBN 0-201-13447-0.

3. IERUSALIMSCHY, Roberto. *Programming in Lua*. 3rd ed. Rio de Janeiro: PUC-Rio, 2013. ISBN 978-85-903798-5-0.

4. BRAAMS, Johannes; CARLISLE, David; JEFFREY, Alan; LAMPORT, Leslie; MITTELBACH, Frank; ROWLEY, Chris; SCHÖPF, Rainer. *The LaTeX2ε Sources* [online]. 2016 [visited on 2016-06-02]. Available from: http://mirrors.ctan.org/macros/latex/base/source2e.pdf.