

A Markdown Interpreter for T_EX

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

Version 1.0.1
June 6, 2016

Contents

| | | | | |
|----------|----------------------------------|----------|----------|---------------------------------------|
| 1 | Introduction | 1 | | |
| 1.1 | About Markdown | 1 | 2.3 | ℳT _E X Interface |
| 1.2 | Feedback | 2 | 2.4 | ConT _E Xt Interface |
| 1.3 | Acknowledgements | 2 | | |
| 1.4 | Prerequisites | 2 | 3 | Technical Documentation |
| 2 | User Guide | 4 | 3.1 | Lua Implementation |
| 2.1 | Lua Interface | 4 | 3.2 | Plain T _E X Implementation |
| 2.2 | Plain T _E X Interface | 8 | 3.3 | ℳT _E X Implementation |
| | | | 3.4 | ConT _E Xt Implementation |
| | | | | |

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 T_EX. These are provided both in the form of a Lua module and in the form of plain T_EX, ℳT_EX, and ConT_EXt macro packages that enable the direct inclusion of markdown documents inside T_EX documents.

Architecturally, the package consists of the Lunamark 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 T_EX, ℳT_EX, and ConT_EXt formats by Vít Novotný.

```
1 if not modules then modules = { } end modules ['markdown'] = {
2   version   = "1.0.1",
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 GitHub¹ 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 T_EX-~~E~~_TX Stack Exchange².

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 T_EX part of the package draws inspiration from several sources including the source code of ~~E~~T_X2_ε, the minted package by Geoffrey M. Poore – which likewise tackles the issue of interfacing with an external interpreter from T_EX, 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 Lua \TeX 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 Lua \TeX 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 Lua \TeX 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 Lua \TeX `\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 $\LaTeX 2_{\epsilon}$ 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 $\LaTeX 2_{\epsilon}$ 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 ConT_EXt prerequisites

The ConT_EXt part of the package requires that either the Mark II or the Mark IV format is loaded and also, since it uses the plain T_EX implementation, all the plain T_EX 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 T_EX 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 T_EX. This interface is used by the plain T_EX 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 T_EX

The Lua interface exposes the `new(options)` method. This method creates converter functions that perform the conversion from markdown to plain T_EX 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.

```
13 M.new = {}
```

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

```
14 local defaultOptions = {}
```

`blankBeforeBlockquote`=true, false default: false

| | |
|--------------------|---|
| <code>true</code> | Require a blank line between a paragraph and the following blockquote. |
| <code>false</code> | Do not require a blank line between a paragraph and the following blockquote. |

```
15 defaultOptions.blankBeforeBlockquote = false
```

`blankBeforeHeading`=true, false default: false

| | |
|--------------------|---|
| <code>true</code> | Require a blank line between a paragraph and the following header. |
| <code>false</code> | Do not require a blank line between a paragraph and the following header. |

```
16 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.

```
17 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.

```
18 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.

```
19 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 interpreted by \TeX . This is encouraged when typesetting automatically generated content or markdown documents that were not prepared with this package in mind.

```
20 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.

```
21 defaultOptions.footnotes = false
```

`preserveTabs=true, false` default: false

`true` Preserve all tabs in the input.

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

```
22 defaultOptions.preserveTabs = false
```

`smartEllipses=true, false` default: false

`true` Convert any ellipses in the input to the `\markdownRenderEllipsis` TeX macro.

`false` Preserve all ellipses in the input.

```
23 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 `\markdownRender01ItemWithNumber` TeX macro.

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

```
24 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.

```
25 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.

```
26 \def\markdownVersion{2016/06/03}%
```

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 `\inputting` 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 corollary, 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 T_EX (see [2, p. 46]). As a corollary, 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 T_EX 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 T_EX 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 T_EX.

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

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

The following example plain T_EX 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.

```
34 \let\markdownOptionBlankBeforeBlockquote\undefined
35 \let\markdownOptionBlankBeforeHeading\undefined
36 \let\markdownOptionDefinitionLists\undefined
37 \let\markdownOptionHashEnumerator\undefined
38 \let\markdownOptionHybrid\undefined
39 \let\markdownOptionFootnotes\undefined
40 \let\markdownOptionPreserveTabs\undefined
41 \let\markdownOptionSmartEllipses\undefined
42 \let\markdownOptionStartNumber\undefined
43 \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.

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

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

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

2.2.3.3 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.

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

2.2.3.4 Link Renderer The `\markdownRendererLink` macro represents a hyper-link. It receives three arguments: the label, the URI, and the title of the link.

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

2.2.3.5 Image Renderer The `\markdownRendererImage` macro represents an image. It receives three arguments: the label, the URL, and the title of the image.

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

2.2.3.6 Bullet List Renderers The `\markdownRendererUllBegin` 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.

```
54 \def\markdownRendererUllBegin{%  
55   \markdownRendererUllBeginPrototype}%
```

The `\markdownRendererUllBeginTight` 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.

```
56 \def\markdownRendererUllBeginTight{%  
57   \markdownRendererUllBeginTightPrototype}%
```

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

```
58 \def\markdownRendererUllItem{%  
59   \markdownRendererUllItemPrototype}%
```

The `\markdownRendererUllEnd` 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.

```
60 \def\markdownRendererUllEnd{%  
61   \markdownRendererUllEndPrototype}%
```

The `\markdownRendererUllEndTight` 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.

```
62 \def\markdownRendererUllEndTight{%  
63   \markdownRendererUllEndTightPrototype}%
```

2.2.3.7 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.

```
64 \def\markdownRendererOlBegin{%
65   \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.

```
66 \def\markdownRendererOlBeginTight{%
67   \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.

```
68 \def\markdownRendererOlItem{%
69   \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.

```
70 \def\markdownRendererOlItemWithNumber{%
71   \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.

```
72 \def\markdownRendererOlEnd{%
73   \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.

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

2.2.3.8 Definition List Renderers The following macros are only produced, 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.

```
76 \def\markdownRendererDlBegin{%
77   \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.

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

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

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

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.

```
82 \def\markdownRendererDlEnd{%  
83   \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.

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

2.2.3.9 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.

```
86 \def\markdownRendererEmphasis{%  
87   \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.

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

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

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

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

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

2.2.3.11 Code Block Renderer The `\markdownRendererInputVerbatim` macro represents a code block. The macro receives a single argument that corresponds to the filename of a file containing the code block to input.

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

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

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

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

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

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

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

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

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

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

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

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

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

2.2.3.13 Horizontal Rule Renderer The `\markdownRendererHorizontalRule` macro represents a horizontal rule. The macro receives no arguments.

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

2.2.3.14 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.

```

110 \def\markdownRendererFootnote{%
111   \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).

```

112 \def\markdownRendererLineBreakPrototype{%
113 \def\markdownRendererEllipsisPrototype{%
114 \long\def\markdownRendererCodeSpanPrototype#1{%
115 \long\def\markdownRendererLinkPrototype#1#2#3{%
116 \long\def\markdownRendererImagePrototype#1#2#3{%
117 \def\markdownRendererUlBeginPrototype{%
118 \def\markdownRendererUlBeginTightPrototype{%
119 \def\markdownRendererUlItemPrototype{%
120 \def\markdownRendererUlEndPrototype{%
121 \def\markdownRendererUlEndTightPrototype{%
122 \def\markdownRendererOlBeginPrototype{%
123 \def\markdownRendererOlBeginTightPrototype{%
124 \def\markdownRendererOlItemPrototype{%
125 \long\def\markdownRendererOlItemWithNumberPrototype#1{%
126 \def\markdownRendererOlEndPrototype{%
127 \def\markdownRendererOlEndTightPrototype{%
128 \def\markdownRendererDlBeginPrototype{%
129 \def\markdownRendererDlBeginTightPrototype{%
130 \long\def\markdownRendererDlItemPrototype#1{%
131 \def\markdownRendererDlEndPrototype{%
132 \def\markdownRendererDlEndTightPrototype{%
133 \long\def\markdownRendererEmphasisPrototype#1{%
134 \long\def\markdownRendererStrongEmphasisPrototype#1{%
135 \def\markdownRendererBlockQuoteBeginPrototype{%
136 \def\markdownRendererBlockQuoteEndPrototype{%
137 \long\def\markdownRendererInputVerbatimPrototype#1{%
138 \long\def\markdownRendererHeadingOnePrototype#1{%
139 \long\def\markdownRendererHeadingTwoPrototype#1{%
140 \long\def\markdownRendererHeadingThreePrototype#1{%
141 \long\def\markdownRendererHeadingFourPrototype#1{%
142 \long\def\markdownRendererHeadingFivePrototype#1{%
143 \long\def\markdownRendererHeadingSixPrototype#1{%
144 \def\markdownRendererHorizontalRulePrototype{%
145 \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.

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

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

```
148 \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.

```
149 \def\markdownLuaRegisterIBCallback#1{%  
150   local old_callback = callback.find("process_input_buffer")  
151   callback.register("process_input_buffer", #1)}%  
152 \def\markdownLuaUnregisterIBCallback{%  
153   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*.

```
154 \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.

```

155 \let\markdownReadAndConvert\relax
156 \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*.

```

157 \catcode'\|=0\catcode'\|=12%
158 \gdef\markdownBegin{%
159     \markdownReadAndConvert{\markdownEnd}%
160     {\markdownEnd}}%
161 \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 (`]`)).

2.3 L^AT_EX Interface

The L^AT_EX interface provides L^AT_EX environments for the typesetting of markdown input from within L^AT_EX, facilities for setting Lua interface options (see Section 2.1.2) used during the conversion from markdown to plain T_EX, and facilities for changing the way markdown tokens are rendered. The rest of the interface is inherited from the plain T_EX interface (see Section 2.2).

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

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

where `⟨options⟩` are the L^AT_EX interface options (see Section 2.3.2).

2.3.1 Typesetting Markdown

The interface exposes the `markdown` and `markdown*` L^AT_EX environments, and redefines the `\markdownInput` command.

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

```

162 \newenvironment{markdown}\relax\relax
163 \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` L^AT_EX 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:

| | |
|--------------------------------------|---|
| <code>\documentclass{article}</code> | <code>\documentclass{article}</code> |
| <code>\usepackage{markdown}</code> | <code>\usepackage{markdown}</code> |
| <code>\begin{document}</code> | <code>\begin{document}</code> |
| <code>% ...</code> | <code>% ...</code> |
| <code>\begin{markdown}</code> | <code>\begin{markdown*}{smartEllipses}</code> |
| <code>_Hello_ **world** ...</code> | <code>_Hello_ **world** ...</code> |
| <code>\end{markdown}</code> | <code>\end{markdown*}</code> |
| <code>% ...</code> | <code>% ...</code> |
| <code>\end{document}</code> | <code>\end{document}</code> |

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 influence this markdown document.

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

| |
|--|
| <code>\documentclass{article}</code> |
| <code>\usepackage{markdown}</code> |
| <code>\begin{document}</code> |
| <code>% ...</code> |
| <code>\markdownInput[smartEllipses]{hello.md}</code> |
| <code>% ...</code> |
| <code>\end{document}</code> |

2.3.2 Options

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

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.

```
164 \newcommand\markdownSetup[1]{%
165   \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).

```
166 \RequirePackage{keyval}
167 \define@key{markdownOptions}{helperScriptFileName}{%
168   \def\markdownOptionHelperScriptFileName{#1}}%
169 \define@key{markdownOptions}{inputTempFileName}{%
170   \def\markdownOptionInputTempFileName{#1}}%
171 \define@key{markdownOptions}{outputTempFileName}{%
172   \def\markdownOptionOutputTempFileName{#1}}%
173 \define@key{markdownOptions}{blankBeforeBlockquote}[true]{%
174   \def\markdownOptionBlankBeforeBlockquote{#1}}%
175 \define@key{markdownOptions}{blankBeforeHeading}[true]{%
176   \def\markdownOptionBlankBeforeHeading{#1}}%
177 \define@key{markdownOptions}{cacheDir}{%
178   \def\markdownOptionCacheDir{#1}}%
179 \define@key{markdownOptions}{definitionLists}[true]{%
180   \def\markdownOptionDefinitionLists{#1}}%
181 \define@key{markdownOptions}{hashEnumerators}[true]{%
182   \def\markdownOptionHashEnumerators{#1}}%
183 \define@key{markdownOptions}{hybrid}[true]{%
184   \def\markdownOptionHybrid{#1}}%
185 \define@key{markdownOptions}{footnotes}[true]{%
186   \def\markdownOptionFootnotes{#1}}%
187 \define@key{markdownOptions}{preserveTabs}[true]{%
188   \def\markdownOptionPreserveTabs{#1}}%
189 \define@key{markdownOptions}{smartEllipses}[true]{%
190   \def\markdownOptionSmartEllipses{#1}}%
191 \define@key{markdownOptions}{startNumber}[true]{%
192   \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.

```
193 \define@key{markdownOptions}{tightLists}[true]{%
194   \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 T_EX Markdown Token Renderers The W_TE_X 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 T_EX interface (see Section 2.2.3).

```

195 \define@key{markdownOptions}{renderers}{%
196   \setkeys{markdownRenderers}{#1}}%
197 \define@key{markdownRenderers}{lineBreak}{%
198   \renewcommand\markdownRendererLineBreak{#1}}%
199 \define@key{markdownRenderers}{ellipsis}{%
200   \renewcommand\markdownRendererEllipsis{#1}}%
201 \define@key{markdownRenderers}{codeSpan}{%
202   \renewcommand\markdownRendererCodeSpan[1]{#1}}%
203 \define@key{markdownRenderers}{link}{%
204   \renewcommand\markdownRendererLink[3]{#1}}%
205 \define@key{markdownRenderers}{image}{%
206   \renewcommand\markdownRendererImage[3]{#1}}%
207 \define@key{markdownRenderers}{ulBegin}{%
208   \renewcommand\markdownRendererUlBegin{#1}}%
209 \define@key{markdownRenderers}{ulBeginTight}{%
210   \renewcommand\markdownRendererUlBeginTight{#1}}%
211 \define@key{markdownRenderers}{ulItem}{%
212   \renewcommand\markdownRendererUlItem{#1}}%
213 \define@key{markdownRenderers}{ulEnd}{%
214   \renewcommand\markdownRendererUlEnd{#1}}%
215 \define@key{markdownRenderers}{ulEndTight}{%
216   \renewcommand\markdownRendererUlEndTight{#1}}%
217 \define@key{markdownRenderers}{olBegin}{%
218   \renewcommand\markdownRendererOlBegin{#1}}%
219 \define@key{markdownRenderers}{olBeginTight}{%
220   \renewcommand\markdownRendererOlBeginTight{#1}}%
221 \define@key{markdownRenderers}{olItem}{%
222   \renewcommand\markdownRendererOlItem{#1}}%
223 \define@key{markdownRenderers}{olItemWithNumber}{%
224   \renewcommand\markdownRendererOlItemWithNumber[1]{#1}}%
225 \define@key{markdownRenderers}{olEnd}{%
226   \renewcommand\markdownRendererOlEnd{#1}}%
227 \define@key{markdownRenderers}{olEndTight}{%
228   \renewcommand\markdownRendererOlEndTight{#1}}%

```

```

229 \define@key{markdownRenderers}{dlBegin}{%
230   \renewcommand\markdownRendererDlBegin{#1}}%
231 \define@key{markdownRenderers}{dlBeginTight}{%
232   \renewcommand\markdownRendererDlBeginTight{#1}}%
233 \define@key{markdownRenderers}{dlItem}{%
234   \renewcommand\markdownRendererDlItem[1]{#1}}%
235 \define@key{markdownRenderers}{dlEnd}{%
236   \renewcommand\markdownRendererDlEnd{#1}}%
237 \define@key{markdownRenderers}{dlEndTight}{%
238   \renewcommand\markdownRendererDlEndTight{#1}}%
239 \define@key{markdownRenderers}{emphasis}{%
240   \renewcommand\markdownRendererEmphasis[1]{#1}}%
241 \define@key{markdownRenderers}{strongEmphasis}{%
242   \renewcommand\markdownRendererStrongEmphasis[1]{#1}}%
243 \define@key{markdownRenderers}{blockquoteBegin}{%
244   \renewcommand\markdownRendererBlockQuoteBegin{#1}}%
245 \define@key{markdownRenderers}{blockquoteEnd}{%
246   \renewcommand\markdownRendererBlockQuoteEnd{#1}}%
247 \define@key{markdownRenderers}{inputVerbatim}{%
248   \renewcommand\markdownRendererInputVerbatim[1]{#1}}%
249 \define@key{markdownRenderers}{headingOne}{%
250   \renewcommand\markdownRendererHeadingOne[1]{#1}}%
251 \define@key{markdownRenderers}{headingTwo}{%
252   \renewcommand\markdownRendererHeadingTwo[1]{#1}}%
253 \define@key{markdownRenderers}{headingThree}{%
254   \renewcommand\markdownRendererHeadingThree[1]{#1}}%
255 \define@key{markdownRenderers}{headingFour}{%
256   \renewcommand\markdownRendererHeadingFour[1]{#1}}%
257 \define@key{markdownRenderers}{headingFive}{%
258   \renewcommand\markdownRendererHeadingFive[1]{#1}}%
259 \define@key{markdownRenderers}{headingSix}{%
260   \renewcommand\markdownRendererHeadingSix[1]{#1}}%
261 \define@key{markdownRenderers}{horizontalRule}{%
262   \renewcommand\markdownRendererHorizontalRule{#1}}%
263 \define@key{markdownRenderers}{footnote}{%
264   \renewcommand\markdownRendererFootnote[1]{#1}}%

```

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

```

\markdownSetup{
  renderers = {
    link = {#3},                % Render links as the link title.
    emphasis = {\emph{#1}},    % Render emphasized text via ‘\emph’.
  }
}

```

2.3.2.3 Plain T_EX Markdown Token Renderer Prototypes The \TeX 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 T_EX interface (see Section 2.2.4).

```

265 \define@key{markdownOptions}{rendererPrototypes}{%
266   \setkeys{markdownRendererPrototypes}{#1}}%
267 \define@key{markdownRendererPrototypes}{lineBreak}{%
268   \renewcommand\markdownRendererLineBreakPrototype{#1}}%
269 \define@key{markdownRendererPrototypes}{ellipsis}{%
270   \renewcommand\markdownRendererEllipsisPrototype{#1}}%
271 \define@key{markdownRendererPrototypes}{codeSpan}{%
272   \renewcommand\markdownRendererCodeSpanPrototype[1]{#1}}%
273 \define@key{markdownRendererPrototypes}{link}{%
274   \renewcommand\markdownRendererLink[3]{#1}}%
275 \define@key{markdownRendererPrototypes}{image}{%
276   \renewcommand\markdownRendererImage[3]{#1}}%
277 \define@key{markdownRendererPrototypes}{ulBegin}{%
278   \renewcommand\markdownRendererUlBeginPrototype{#1}}%
279 \define@key{markdownRendererPrototypes}{ulBeginTight}{%
280   \renewcommand\markdownRendererUlBeginTightPrototype{#1}}%
281 \define@key{markdownRendererPrototypes}{ulItem}{%
282   \renewcommand\markdownRendererUlItemPrototype{#1}}%
283 \define@key{markdownRendererPrototypes}{ulEnd}{%
284   \renewcommand\markdownRendererUlEndPrototype{#1}}%
285 \define@key{markdownRendererPrototypes}{ulEndTight}{%
286   \renewcommand\markdownRendererUlEndTightPrototype{#1}}%
287 \define@key{markdownRendererPrototypes}{olBegin}{%
288   \renewcommand\markdownRendererOlBeginPrototype{#1}}%
289 \define@key{markdownRendererPrototypes}{olBeginTight}{%
290   \renewcommand\markdownRendererOlBeginTightPrototype{#1}}%
291 \define@key{markdownRendererPrototypes}{olItem}{%
292   \renewcommand\markdownRendererOlItemPrototype{#1}}%
293 \define@key{markdownRendererPrototypes}{olItemWithNumber}{%
294   \renewcommand\markdownRendererOlItemWithNumberPrototype[1]{#1}}%
295 \define@key{markdownRendererPrototypes}{olEnd}{%
296   \renewcommand\markdownRendererOlEndPrototype{#1}}%
297 \define@key{markdownRendererPrototypes}{olEndTight}{%
298   \renewcommand\markdownRendererOlEndTightPrototype{#1}}%
299 \define@key{markdownRendererPrototypes}{dlBegin}{%
300   \renewcommand\markdownRendererDlBeginPrototype{#1}}%
301 \define@key{markdownRendererPrototypes}{dlBeginTight}{%
302   \renewcommand\markdownRendererDlBeginTightPrototype{#1}}%
303 \define@key{markdownRendererPrototypes}{dlItem}{%
304   \renewcommand\markdownRendererDlItemPrototype[1]{#1}}%
305 \define@key{markdownRendererPrototypes}{dlEnd}{%
306   \renewcommand\markdownRendererDlEndPrototype{#1}}%

```



```

307 \define@key{markdownRendererPrototypes}{d1EndTight}{%
308   \renewcommand\markdownRendererD1EndTightPrototype{#1}}%
309 \define@key{markdownRendererPrototypes}{emphasis}{%
310   \renewcommand\markdownRendererEmphasisPrototype[1]{#1}}%
311 \define@key{markdownRendererPrototypes}{strongEmphasis}{%
312   \renewcommand\markdownRendererStrongEmphasisPrototype[1]{#1}}%
313 \define@key{markdownRendererPrototypes}{blockQuoteBegin}{%
314   \renewcommand\markdownRendererBlockQuoteBeginPrototype{#1}}%
315 \define@key{markdownRendererPrototypes}{blockQuoteEnd}{%
316   \renewcommand\markdownRendererBlockQuoteEndPrototype{#1}}%
317 \define@key{markdownRendererPrototypes}{inputVerbatim}{%
318   \renewcommand\markdownRendererInputVerbatimPrototype[1]{#1}}%
319 \define@key{markdownRendererPrototypes}{headingOne}{%
320   \renewcommand\markdownRendererHeadingOnePrototype[1]{#1}}%
321 \define@key{markdownRendererPrototypes}{headingTwo}{%
322   \renewcommand\markdownRendererHeadingTwoPrototype[1]{#1}}%
323 \define@key{markdownRendererPrototypes}{headingThree}{%
324   \renewcommand\markdownRendererHeadingThreePrototype[1]{#1}}%
325 \define@key{markdownRendererPrototypes}{headingFour}{%
326   \renewcommand\markdownRendererHeadingFourPrototype[1]{#1}}%
327 \define@key{markdownRendererPrototypes}{headingFive}{%
328   \renewcommand\markdownRendererHeadingFivePrototype[1]{#1}}%
329 \define@key{markdownRendererPrototypes}{headingSix}{%
330   \renewcommand\markdownRendererHeadingSixPrototype[1]{#1}}%
331 \define@key{markdownRendererPrototypes}{horizontalRule}{%
332   \renewcommand\markdownRendererHorizontalRulePrototype{#1}}%
333 \define@key{markdownRendererPrototypes}{footnote}{%
334   \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 Con \TeX t Interface

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


```

335 \writestatus{loading}{ConTeXt User Module / markdown}%
336 \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.

```

337 \let\startmarkdown\relax
338 \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 T_EX writer were hidden behind the converter functions exposed by the Lua interface (see Section 2.1).

```

339 local upper, gsub, format, length =
340   string.upper, string.gsub, string.format, string.len
341 local concat = table.concat
342 local P, R, S, V, C, Cg, Cb, Cmt, Cc, Ct, B, Cs, any =
343   lpeg.P, lpeg.R, lpeg.S, lpeg.V, lpeg.C, lpeg.Cg, lpeg.Cb,
344   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.

```

345 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.

```

346 function util.err(msg, exit_code)
347   io.stderr:write("markdown.lua: " .. msg .. "\n")
348   os.exit(exit_code or 1)
349 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.

```

350 function util.cache(dir, string, salt, transform, suffix)
351   local digest = md5.sumhexa(string .. (salt or ""))
352   local name = util.pathname(dir, digest .. suffix)
353   local file = io.open(name, "r")
354   if file == nil then -- If no cache entry exists, then create a new one.
355     local file = assert(io.open(name, "w"))
356     local result = string
357     if transform ~= nil then
358       result = transform(result)
359     end
360     assert(file:write(result))
361     assert(file:close())
362   end
363   return name
364 end

```

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

```

365 function util.table_copy(t)
366   local u = { }
367   for k, v in pairs(t) do u[k] = v end
368   return setmetatable(u, getmetatable(t))
369 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].

```

370 function util.expand_tabs_in_line(s, tabstop)
371   local tab = tabstop or 4
372   local corr = 0
373   return (s:gsub("\t", function(p)
374     local sp = tab - (p - 1 + corr) % tab
375     corr = corr - 1 + sp
376     return string.rep(" ", sp)
377   end))
378 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.

```

379 function util.walk(t, f)
380   local typ = type(t)
381   if typ == "string" then
382     f(t)
383   elseif typ == "table" then
384     local i = 1
385     local n
386     n = t[i]
387     while n do
388       util.walk(n, f)
389       i = i + 1
390       n = t[i]
391     end
392   elseif typ == "function" then
393     local ok, val = pcall(t)
394     if ok then
395       util.walk(val, f)
396     end
397   else
398     f(tostring(t))
399   end
400 end

```

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

```

401 function util.flatten(ary)
402   local new = {}
403   for _,v in ipairs(ary) do
404     if type(v) == "table" then
405       for _,w in ipairs(util.flatten(v)) do
406         new[#new + 1] = w
407       end
408     else
409       new[#new + 1] = v
410     end
411   end
412   return new
413 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.

```

414 function util.rope_to_string(rope)
415   local buffer = {}
416   util.walk(rope, function(x) buffer[#buffer + 1] = x end)
417   return table.concat(buffer)
418 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.

```

419 function util.rope_last(rope)
420   if #rope == 0 then
421     return nil
422   else
423     local l = rope[#rope]
424     if type(l) == "table" then
425       return util.rope_last(l)
426     else
427       return l
428     end
429   end
430 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 $1 \leq i \leq \#ary$.

```

431 function util.intersperse(ary, x)
432   local new = {}
433   local l = #ary
434   for i,v in ipairs(ary) do
435     local n = #new
436     new[n + 1] = v
437     if i ~= l then
438       new[n + 2] = x

```

```

439     end
440   end
441   return new
442 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 $1 \leq i \leq \#ary$.

```

443 function util.map(ary, f)
444   local new = {}
445   for i,v in ipairs(ary) do
446     new[i] = f(v)
447   end
448   return new
449 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 occurrences of escapable strings and characters (in this order).

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

```

450 function util.escaper(char_escapes, string_escapes)

```

Build a string of escapable characters.

```

451   local char_escapes_list = ""
452   for i,_ in pairs(char_escapes) do
453     char_escapes_list = char_escapes_list .. i
454   end

```

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

```

455   local escapable = S(char_escapes_list) / char_escapes

```

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

$$\sum_{(k,v) \in \text{string_escapes}} P(k) / v + \text{escapable}$$

capture that replaces any occurrence of the string `k` with the string `v` for each $(k, v) \in \text{string_escapes}$. 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.

```

456   if string_escapes then
457     for k,v in pairs(string_escapes) do
458       escapable = P(k) / v + escapable
459     end
460   end

```

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

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

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

```
462 return function(s)
463     return lpeg.match(escape_string, s)
464 end
465 end
```

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

```
466 function util.pathname(dir, file)
467     if #dir == 0 then
468         return file
469     else
470         return dir .. "/" .. file
471     end
472 end
```

3.1.2 Plain T_EX Writer

This section documents the `writer` object, which implements the routines for producing the T_EX output. The object is an amalgamate of the generic, T_EX, L^AT_EX writer objects that were located in the `lunamark/writer/generic.lua`, `lunamark/writer/tex.lua`, and `lunamark/writer/latex.lua` files in the Luna-mark 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.

```
473 M.writer = {}
```

The `writer.new` method creates and returns a new T_EX 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`.

```
474 function M.writer.new(options)
475     local self = {}
476     options = options or {}
```

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

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

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

```
479 self.suffix = ".tex"
```

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

```
480 self.space = " "
```

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

```
481 function self.plain(s)
482     return s
483 end
```

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

```
484 function self.paragraph(s)
485     return s
486 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.

```
487 function self.pack(name)
488     return [[\input]] .. name .. [[\relax]]
489 end
```

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

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

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

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

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

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

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

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

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

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

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

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

Define a table `escaped` containing the mapping from special plain `TEX` characters to their escaped variants.

```
496 local escaped = {
497     [{""] = "\\{",
498     ["}"] = "\\}",
499     ["$"] = "\\$",
500     ["%"] = "\\%",
501     ["&"] = "\\&",
502     ["_"] = "\\_",
```

```

503     ["#"] = "\\#",
504     ["^"] = "\\^{}",
505     ["\\"] = "\\char92{}",
506     ["~"] = "\\char126{}",
507     ["|"] = "\\char124{}", }

```

Use the `escaped` table to create an escaper function `escape`.

```

508     local escape = util.escaper(escaped)

```

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

```

509     if options.hybrid then
510         self.string = function(s) return s end
511     else
512         self.string = escape
513     end

```

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

```

514     function self.code(s)
515         return {"\\markdownRendererCodeSpan{",escape(s),"}"}
516     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.

```

517     function self.link(lab,src,tit)
518         return {"\\markdownRendererLink{",lab,"}",
519             {"",self.string(src),""},
520             {"",self.string(tit),""}}
521     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.

```

522     function self.image(lab,src,tit)
523         return {"\\markdownRendererImage{",lab,"}",
524             {"",self.string(src),""},
525             {"",self.string(tit),""}}
526     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.

```

527     local function ulitem(s)
528         return {"\\markdownRendererULItem ",s}
529     end

```



```

530
531 function self.bulletlist(items,tight)
532     local buffer = {}
533     for _,item in ipairs(items) do
534         buffer[#buffer + 1] = ulitem(item)
535     end
536     local contents = util.intersperse(buffer,"\n")
537     if tight and options.tightLists then
538         return {"\\markdownRendererUlBeginTight\n",contents,
539             "\n\\markdownRendererUlEndTight "}
540     else
541         return {"\\markdownRendererUlBegin\n",contents,
542             "\n\\markdownRendererUlEnd "}
543     end
544 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.

```

545     local function olitem(s,num)
546         if num ~= nil then
547             return {"\\markdownRendererOlItemWithNumber{" ,num,"} ",s}
548         else
549             return {"\\markdownRendererOlItem ",s}
550         end
551     end
552
553 function self.orderedlist(items,tight,startnum)
554     local buffer = {}
555     local num = startnum
556     for _,item in ipairs(items) do
557         buffer[#buffer + 1] = olitem(item,num)
558         if num ~= nil then
559             num = num + 1
560         end
561     end
562     local contents = util.intersperse(buffer,"\n")
563     if tight and options.tightLists then
564         return {"\\markdownRendererOlBeginTight\n",contents,
565             "\n\\markdownRendererOlEndTight "}
566     else
567         return {"\\markdownRendererOlBegin\n",contents,
568             "\n\\markdownRendererOlEnd "}
569     end
570 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.

```

571 local function dlitem(term,defs)
572     return {"\\markdownRendererDlItem{",term,"}\\n",defs}
573 end
574
575 function self.definitionlist(items,tight)
576     local buffer = {}
577     for _,item in ipairs(items) do
578         buffer[#buffer + 1] = dlitem(item.term,
579             util.intersperse(item.definitions, self.interblocksep))
580     end
581     local contents = util.intersperse(buffer, self.containersep)
582     if tight and options.tightLists then
583         return {"\\markdownRendererDlBeginTight\\n\\n", contents,
584             "\\n\\n\\markdownRendererDlEndTight\\n"}
585     else
586         return {"\\markdownRendererDlBegin\\n\\n", contents,
587             "\\n\\n\\markdownRendererDlEnd\\n"}
588     end
589 end

```

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

```

590 function self.emphasis(s)
591     return {"\\markdownRendererEmphasis{",s,""}
592 end

```

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

```

593 function self.strong(s)
594     return {"\\markdownRendererStrongEmphasis{",s,""}
595 end

```

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

```

596 function self.blockquote(s)
597     return {"\\markdownRendererBlockQuoteBegin\\n",s,
598         "\\n\\markdownRendererBlockQuoteEnd "}
599 end

```

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

```

600 function self.verbatim(s)
601     local name = util.cache(options.cacheDir, s, nil, nil, ".verbatim")
602     return {"\\markdownRendererInputVerbatim{",name,""}

```

```
603 end
```

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

```
604 function self.heading(s,level)
605     local cmd
606     if level == 1 then
607         cmd = "\\markdownRendererHeadingOne"
608     elseif level == 2 then
609         cmd = "\\markdownRendererHeadingTwo"
610     elseif level == 3 then
611         cmd = "\\markdownRendererHeadingThree"
612     elseif level == 4 then
613         cmd = "\\markdownRendererHeadingFour"
614     elseif level == 5 then
615         cmd = "\\markdownRendererHeadingFive"
616     elseif level == 6 then
617         cmd = "\\markdownRendererHeadingSix"
618     else
619         cmd = ""
620     end
621     return {cmd,"{" ,s,""} }
622 end
```

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

```
623 function self.note(s)
624     return {"\\markdownRendererFootnote{" ,s,""} }
625 end
626
627 return self
628 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`.

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

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

```
633     setmetatable(options, { __index = function (_, key)
634         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.

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

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

```
639     local expandtabs
640     if options.preserveTabs then
641         expandtabs = function(s) return s end
642     else
643         expandtabs = function(s)
644             if s:find("\t") then
645                 return s:gsub("[^\n]*", util.expand_tabs_in_line)
646             else
647                 return s
648             end
649         end
650     end
```

3.1.3.2 Top Level Parsing Functions

```
651     local syntax
652     local blocks
653     local inlines
654
655     local parse_blocks =
656         function(str)
657             local res = lpeg.match(blocks, str)
658             if res == nil then
659                 error(format("parse_blocks failed on:\n%s", str:sub(1,20)))
```

```

660         else
661             return res
662         end
663     end
664
665     local parse_inlines =
666     function(str)
667         local res = lpeg.match(inlines, str)
668         if res == nil then
669             error(format("parse_inlines failed on:\n%s",
670                 str:sub(1,20)))
671         else
672             return res
673         end
674     end
675
676     local parse_inlines_no_link =
677     function(str)
678         local res = lpeg.match(inlines_no_link, str)
679         if res == nil then
680             error(format("parse_inlines_no_link failed on:\n%s",
681                 str:sub(1,20)))
682         else
683             return res
684         end
685     end

```

3.1.3.3 Generic PEG Patterns

```

686     local percent      = P("%")
687     local asterisk      = P("*")
688     local dash          = P("-")
689     local plus          = P("+")
690     local underscore    = P("_")
691     local period        = P(".")
692     local hash          = P("#")
693     local ampersand     = P("&")
694     local backtick      = P("`")
695     local less          = P("<")
696     local more          = P(">")
697     local space         = P(" ")
698     local squote        = P("'")
699     local dquote        = P('"')
700     local lparent       = P("(")
701     local rparent       = P(")")
702     local lbracket      = P("[")
703     local rbracket      = P("]")

```

```

704 local circumflex      = P("^")
705 local slash            = P("/")
706 local equal            = P("=")
707 local colon            = P(":")
708 local semicolon        = P(";")
709 local exclamation      = P("!")
710
711 local digit            = R("09")
712 local hexdigit         = R("09","af","AF")
713 local letter           = R("AZ","az")
714 local alphanumeric     = R("AZ","az","09")
715 local keyword          = letter * alphanumeric^0
716
717 local doubleasterisks   = P("**")
718 local doubleunderscores = P("__")
719 local fourspaces       = P("    ")
720
721 local any              = P(1)
722 local fail             = any - 1
723 local always           = P("")
724
725 local escapable        = S("\\'*_{}[]()+.!#-~:~")
726 local anyescaped       = P("\\") / "" * escapable
727 + any
728
729 local tab              = P("\t")
730 local spacechar        = S("\t ")
731 local spacing          = S(" \n\r\t")
732 local newline          = P("\n")
733 local nonspacechar     = any - spacing
734 local tightblocksep    = P("\001")
735
736 local specialchar
737 if options.smartEllipses then
738     specialchar = S("*_ '&[]!\\.")
739 else
740     specialchar = S("*_ '&[]!\\")
741 end
742
743 local normalchar        = any -
744     (specialchar + spacing + tightblocksep)
745 local optionalspace     = spacechar^0
746 local spaces            = spacechar^1
747 local eof               = - any
748 local nonindentSPACE    = space^-3 * - spacechar
749 local indent            = space^-3 * tab
750 + fourspaces / ""

```

```

751 local linechar          = P(1 - newline)
752
753 local blankline          = optionalspace * newline / "\n"
754 local blanklines         = blankline^0
755 local skipblanklines     = (optionalspace * newline)^0
756 local indentedline       = indent /"" * C(linechar^1 * newline^-1)
757 local optionallyindentedline = indent^-1 /"" * C(linechar^1 * newline^-1)
758 local sp                 = spacing^0
759 local spnl               = optionalspace * (newline * optionalspace)^-1
760 local line               = linechar^0 * newline
761                          + linechar^1 * eof
762 local nonemptyline       = line - blankline
763
764 local chunk = line * (optionallyindentedline - blankline)^0
765
766 -- block followed by 0 or more optionally
767 -- indented blocks with first line indented.
768 local function indented_blocks(bl)
769   return Cs( bl
770             * (blankline^1 * indent * -blankline * bl)^0
771             * blankline^1 )
772 end

```

3.1.3.4 List PEG Patterns

```

773 local bulletchar = C(plus + asterisk + dash)
774
775 local bullet      = ( bulletchar * #spacing * (tab + space^-3)
776                     + space * bulletchar * #spacing * (tab + space^-2)
777                     + space * space * bulletchar * #spacing * (tab + space^-1)
778                     + space * space * space * bulletchar * #spacing
779                     ) * -bulletchar
780
781 if options.hashEnumerators then
782   dig = digit + hash
783 else
784   dig = digit
785 end
786
787 local enumerator = C(dig^3 * period) * #spacing
788                  + C(dig^2 * period) * #spacing * (tab + space^1)
789                  + C(dig * period) * #spacing * (tab + space^-2)
790                  + space * C(dig^2 * period) * #spacing
791                  + space * C(dig * period) * #spacing * (tab + space^-1)
792                  + space * space * C(dig^1 * period) * #spacing

```

3.1.3.5 Code Span PEG Patterns

```

793 local openticks = Cg(backtick^1, "ticks")
794
795 local function captures_equal_length(s,i,a,b)
796     return #a == #b and i
797 end
798
799 local closeticks = space^-1 *
800     Cmt(C(backtick^1) * Cb("ticks"), captures_equal_length)
801
802 local intickschar = (any - S(" \n\r'"))
803     + (newline * -blankline)
804     + (space - closeticks)
805     + (backtick^1 - closeticks)
806
807 local inticks = openticks * space^-1 * C(intickschar^1) * closeticks

```

3.1.3.6 Tag PEG Patterns

```

808 local leader = space^-3
809
810 -- in balanced brackets, parentheses, quotes:
811 local bracketed = P{ lbracket
812     * ((anyescaped - (lbracket + rbracket
813         + blankline^2)) + V(1))^0
814     * rbracket }
815
816 local inparens = P{ lparent
817     * ((anyescaped - (lparent + rparent
818         + blankline^2)) + V(1))^0
819     * rparent }
820
821 local squoted = P{ squote * alphanumeric
822     * ((anyescaped - (squote + blankline^2))
823         + V(1))^0
824     * squote }
825
826 local dquoted = P{ dquote * alphanumeric
827     * ((anyescaped - (dquote + blankline^2))
828         + V(1))^0
829     * dquote }
830
831 -- bracketed 'tag' for markdown links, allowing nested brackets:
832 local tag = lbracket
833     * Cs((alphanumeric^1
834         + bracketed
835         + inticks
836         + (anyescaped - (rbracket + blankline^2)))^0)

```



```

837             * rbracket
838
839 -- url for markdown links, allowing balanced parentheses:
840 local url      = less * Cs((anyescaped-more)^0) * more
841               + Cs((inparens + (anyescaped-spacing-rparent))^1)
842
843 -- quoted text possibly with nested quotes:
844 local title_s  = squote * Cs(((anyescaped-squote) + squoted)^0) *
845               squote
846
847 local title_d  = dquote * Cs(((anyescaped-dquote) + dquoted)^0) *
848               dquote
849
850 local title_p  = lparent
851               * Cs((inparens + (anyescaped-rparent))^0)
852               * rparent
853
854 local title    = title_d + title_s + title_p
855
856 local optionaltitle = spnl * title * spacechar^0
857               + Cc("")

```

3.1.3.7 Footnote PEG Patterns

```

858 local rawnotes = {}
859
860 local function strip_first_char(s)
861     return s:sub(2)
862 end
863
864 -- like indirect_link
865 local function lookup_note(ref)
866     return function()
867         local found = rawnotes[normalize_tag(ref)]
868         if found then
869             return writer.note(parse_blocks(found))
870         else
871             return {"[^", ref, "]" }
872         end
873     end
874 end
875
876 local function register_note(ref,rawnote)
877     rawnotes[normalize_tag(ref)] = rawnote
878     return ""
879 end
880

```

```

881 local RawNoteRef = #(lbracket * circumflex) * tag / strip_first_char
882
883 local NoteRef      = RawNoteRef / lookup_note
884
885 local NoteBlock
886
887 if options.footnotes then
888     NoteBlock = leader * RawNoteRef * colon * spnl *
889               indented_blocks(chunk) / register_note
890 else
891     NoteBlock = fail
892 end

```

3.1.3.8 Link and Image PEG Patterns

```

893 -- List of references defined in the document
894 local references
895
896 -- add a reference to the list
897 local function register_link(tag,url,title)
898     references[normalize_tag(tag)] = { url = url, title = title }
899     return ""
900 end
901
902 -- parse a reference definition:  [foo]: /bar "title"
903 local define_reference_parser =
904     leader * tag * colon * spacechar^0 * url * optionaltitle * blankline^1
905
906 -- lookup link reference and return either
907 -- the link or nil and fallback text.
908 local function lookup_reference(label,sps,tag)
909     local tagpart
910     if not tag then
911         tag = label
912         tagpart = ""
913     elseif tag == "" then
914         tag = label
915         tagpart = "[]"
916     else
917         tagpart = {"[", parse_inlines(tag), "]" }
918     end
919     if sps then
920         tagpart = {sps, tagpart}
921     end
922     local r = references[normalize_tag(tag)]
923     if r then
924         return r

```

```

925         else
926             return nil, {"[", parse_inlines(label), "]", tagpart}
927         end
928     end
929
930     -- lookup link reference and return a link, if the reference is found,
931     -- or a bracketed label otherwise.
932     local function indirect_link(label,sps,tag)
933         return function()
934             local r,fallback = lookup_reference(label,sps,tag)
935             if r then
936                 return writer.link(parse_inlines_no_link(label), r.url, r.title)
937             else
938                 return fallback
939             end
940         end
941     end
942
943     -- lookup image reference and return an image, if the reference is found,
944     -- or a bracketed label otherwise.
945     local function indirect_image(label,sps,tag)
946         return function()
947             local r,fallback = lookup_reference(label,sps,tag)
948             if r then
949                 return writer.image(writer.string(label), r.url, r.title)
950             else
951                 return {"!", fallback}
952             end
953         end
954     end

```

3.1.3.9 Inline Element PEG Patterns

```

955     local Inline      = V("Inline")
956
957     local Str          = normalchar^1 / writer.string
958
959     local Ellipsis     = P("...") / writer.ellipsis
960
961     local Smart        = Ellipsis
962
963     local Symbol       = (specialchar - tightblocksep) / writer.string
964
965     local Code         = inticks / writer.code
966
967     local bqstart      = more
968     local headerstart  = hash

```

```

969             + (line * (equal^1 + dash^1) * optionalspace * newline)
970
971 if options.blankBeforeBlockquote then
972     bqstart = fail
973 end
974
975 if options.blankBeforeHeading then
976     headerstart = fail
977 end
978
979 local Endline    = newline * -( -- newline, but not before...
980                     blankline -- paragraph break
981                     + tightblocksep -- nested list
982                     + eof          -- end of document
983                     + bqstart
984                     + headerstart
985                 ) * spacechar^0 / writer.space
986

```

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.

```

987 local Space      = spacechar^2 * Endline / writer.linebreak
988                 + spacechar^1 * Endline^-1 * eof / ""
989                 + spacechar^1 * Endline^-1 * optionalspace / writer.space
990
991 -- parse many p between starter and ender
992 local function between(p, starter, ender)
993     local ender2 = B(nonspacechar) * ender
994     return (starter * #nonspacechar * Ct(p * (p - ender2)^0) * ender2)
995 end
996
997 local Strong = ( between(Inline, doubleasterisks, doubleasterisks)
998                 + between(Inline, doubleunderscores, doubleunderscores)
999                 ) / writer.strong
1000
1001 local Emph    = ( between(Inline, asterisk, asterisk)
1002                 + between(Inline, underscore, underscore)
1003                 ) / writer.emphasis
1004
1005 local urlchar = anyescaped - newline - more
1006
1007 local AutoLinkUrl = less
1008                   * C(alphanumeric^1 * P(":/") * urlchar^1)
1009                   * more
1010                   / function(url)
1011                   return writer.link(writer.string(url), url)

```

```

1012             end
1013
1014     local AutoLinkEmail = less
1015         * C((alphanumeric + S("-._+"))~1 * P("@") * urlchar~1)
1016         * more
1017         / function(email)
1018             return writer.link(writer.string(email),
1019                               "mailto: "..email)
1020         end
1021
1022     local DirectLink      = (tag / parse_inlines_no_link) -- no links inside links
1023         * spnl
1024         * lparent
1025         * (url + Cc("")) -- link can be empty [foo]()
1026         * optionaltitle
1027         * rparent
1028         / writer.link
1029
1030     local IndirectLink = tag * (C(spnl) * tag)~1 / indirect_link
1031
1032     -- parse a link or image (direct or indirect)
1033     local Link          = DirectLink + IndirectLink
1034
1035     local DirectImage    = exclamation
1036         * (tag / parse_inlines)
1037         * spnl
1038         * lparent
1039         * (url + Cc("")) -- link can be empty [foo]()
1040         * optionaltitle
1041         * rparent
1042         / writer.image
1043
1044     local IndirectImage  = exclamation * tag * (C(spnl) * tag)~1 /
1045                               indirect_image
1046
1047     local Image          = DirectImage + IndirectImage
1048
1049     -- avoid parsing long strings of * or _ as emph/strong
1050     local U1OrStarLine   = asterisk^4 + underscore^4 / writer.string
1051
1052     local EscapedChar     = S("\\") * C(escapable) / writer.string

```

3.1.3.10 Block Element PEG Patterns

```

1053     local Block          = V("Block")
1054
1055     local Verbatim        = Cs( (blanklines

```

```

1056             * ((indentedline - blankline))^1)^1
1057             ) / expandtabs / writer.verbatim
1058
1059 -- strip off leading > and indents, and run through blocks
1060 local Blockquote = Cs((
1061     ((leader * more * space^-1)/" " * linechar^0 * newline)^1
1062     * (-blankline * linechar^1 * newline)^0
1063     * blankline^0
1064     )^1) / parse_blocks / writer.blockquote
1065
1066 local function lineof(c)
1067     return (leader * (P(c) * optionalspace)^3 * newline * blankline^1)
1068 end
1069
1070 local HorizontalRule = ( lineof(asterisk)
1071     + lineof(dash)
1072     + lineof(underscore)
1073     ) / writer.hrule
1074
1075 local Reference = define_reference_parser / register_link
1076
1077 local Paragraph = nonindentspace * Ct(Inline^1) * newline
1078     * ( blankline^1
1079     + #hash
1080     + #(leader * more * space^-1)
1081     )
1082     / writer.paragraph
1083
1084 local Plain = nonindentspace * Ct(Inline^1) / writer.plain

```

3.1.3.11 List PEG Patterns

```

1085 local starter = bullet + enumerator
1086
1087 -- we use \001 as a separator between a tight list item and a
1088 -- nested list under it.
1089 local NestedList = Cs((optionallyindentedline - starter)^1)
1090     / function(a) return "\001"..a end
1091
1092 local ListBlockLine = optionallyindentedline
1093     - blankline - (indent^-1 * starter)
1094
1095 local ListBlock = line * ListBlockLine^0
1096
1097 local ListContinuationBlock = blanklines * (indent / " ") * ListBlock
1098
1099 local function TightListItem(starter)

```

```

1100     return -HorizontalRule
1101         * (Cs(starter / "" * ListBlock * NestedList^-1) /
1102             parse_blocks)
1103         * -(blanklines * indent)
1104 end
1105
1106 local function LooseListItem(starter)
1107     return -HorizontalRule
1108         * Cs( starter / "" * ListBlock * Cc("\n")
1109             * (NestedList + ListContinuationBlock^0)
1110             * (blanklines / "\n\n")
1111             ) / parse_blocks
1112 end
1113
1114 local BulletList = ( Ct(TightListItem(bullet)^1)
1115                     * Cc(true) * skipblanklines * -bullet
1116                     + Ct(LooseListItem(bullet)^1)
1117                     * Cc(false) * skipblanklines ) /
1118     writer.bulletlist
1119
1120 local function orderedlist(items,tight,startNumber)
1121     if options.startNumber then
1122         startNumber = tonumber(startNumber) or 1 -- fallback for '#'
1123     else
1124         startNumber = nil
1125     end
1126     return writer.orderedlist(items,tight,startNumber)
1127 end
1128
1129 local OrderedList = Cg(enumerator, "listtype") *
1130     ( Ct(TightListItem(Cb("listtype")) *
1131         TightListItem(enumerator)^0)
1132       * Cc(true) * skipblanklines * -enumerator
1133       + Ct(LooseListItem(Cb("listtype")) *
1134         LooseListItem(enumerator)^0)
1135       * Cc(false) * skipblanklines
1136       ) * Cb("listtype") / orderedlist
1137
1138 local defstartchar = S("~:")
1139 local defstart      = ( defstartchar * #spacing * (tab + space^-3)
1140                       + space * defstartchar * #spacing * (tab + space^-2)
1141                       + space * space * defstartchar * #spacing *
1142                           (tab + space^-1)
1143                       + space * space * space * defstartchar * #spacing
1144                       )
1145
1146 local dlchunk = Cs(line * (indentedline - blankline)^0)

```

```

1147
1148 local function definition_list_item(term, defs, tight)
1149     return { term = parse_inlines(term), definitions = defs }
1150 end
1151
1152 local DefinitionListItemLoose = C(line) * skipblanklines
1153     * Ct((defstart *
1154         indented_blocks(dlchunk) /
1155         parse_blocks)^1)
1156     * Cc(false)
1157     / definition_list_item
1158
1159 local DefinitionListItemTight = C(line)
1160     * Ct((defstart * dlchunk /
1161         parse_blocks)^1)
1162     * Cc(true)
1163     / definition_list_item
1164
1165 local DefinitionList = ( Ct(DefinitionListItemLoose^1) * Cc(false)
1166     + Ct(DefinitionListItemTight^1)
1167     * (skipblanklines *
1168         -DefinitionListItemLoose * Cc(true))
1169     ) / writer.definitionlist

```

3.1.3.12 Blank PEG Patterns

```

1170 local Blank          = blankline / ""
1171                      + NoteBlock
1172                      + Reference
1173                      + (tightblocksep / "\n")

```

3.1.3.13 Heading PEG Patterns

```

1174 -- parse Atx heading start and return level
1175 local HeadingStart = #hash * C(hash^-6) * -hash / length
1176
1177 -- parse setext header ending and return level
1178 local HeadingLevel = equal^1 * Cc(1) + dash^1 * Cc(2)
1179
1180 local function strip_atx_end(s)
1181     return s:gsub("#%s]*\n$", "")
1182 end
1183
1184 -- parse atx header
1185 local AtxHeading = Cg(HeadingStart, "level")
1186     * optionalspace
1187     * (C(line) / strip_atx_end / parse_inlines)
1188     * Cb("level")

```



```

1189             / writer.heading
1190
1191 -- parse setext header
1192 local SettextHeading = #(line * S("--"))
1193             * Ct(line / parse_inlines)
1194             * HeadingLevel
1195             * optionalspace * newline
1196             / writer.heading

```

3.1.3.14 Top Level PEG Specification

```

1197 syntax =
1198   { "Blocks",
1199
1200     Blocks          = Blank^0 *
1201                     Block^-1 *
1202                     (Blank^0 / function()
1203                       return writer.interblocksep
1204                       end * Block)^0 *
1205                     Blank^0 *
1206                     eof,
1207
1208     Blank           = Blank,
1209
1210     Block           = V("Blockquote")
1211                     + V("Verbatim")
1212                     + V("HorizontalRule")
1213                     + V("BulletList")
1214                     + V("OrderedList")
1215                     + V("Heading")
1216                     + V("DefinitionList")
1217                     + V("Paragraph")
1218                     + V("Plain"),
1219
1220     Blockquote      = Blockquote,
1221     Verbatim        = Verbatim,
1222     HorizontalRule  = HorizontalRule,
1223     BulletList      = BulletList,
1224     OrderedList     = OrderedList,
1225     Heading         = AtxHeading + SettextHeading,
1226     DefinitionList  = DefinitionList,
1227     DisplayHtml     = DisplayHtml,
1228     Paragraph       = Paragraph,
1229     Plain           = Plain,
1230
1231     Inline          = V("Str")
1232                     + V("Space")

```

```

1233         + V("Endline")
1234         + V("U1OrStarLine")
1235         + V("Strong")
1236         + V("Emph")
1237         + V("NoteRef")
1238         + V("Link")
1239         + V("Image")
1240         + V("Code")
1241         + V("AutoLinkUrl")
1242         + V("AutoLinkEmail")
1243         + V("EscapedChar")
1244         + V("Smart")
1245         + V("Symbol"),
1246
1247     Str          = Str,
1248     Space       = Space,
1249     Endline     = Endline,
1250     U1OrStarLine = U1OrStarLine,
1251     Strong      = Strong,
1252     Emph       = Emph,
1253     NoteRef    = NoteRef,
1254     Link       = Link,
1255     Image      = Image,
1256     Code       = Code,
1257     AutoLinkUrl = AutoLinkUrl,
1258     AutoLinkEmail = AutoLinkEmail,
1259     InlineHtml  = InlineHtml,
1260     HtmlEntity  = HtmlEntity,
1261     EscapedChar = EscapedChar,
1262     Smart       = Smart,
1263     Symbol      = Symbol,
1264 }
1265
1266 if not options.definitionLists then
1267     syntax.DefinitionList = fail
1268 end
1269
1270 if not options.footnotes then
1271     syntax.NoteRef = fail
1272 end
1273
1274 if not options.smartEllipses then
1275     syntax.Smart = fail
1276 end
1277
1278 blocks = Ct(syntax)
1279

```

```

1280 local inlines_t = util.table_copy(syntax)
1281 inlines_t[1] = "Inlines"
1282 inlines_t.Inlines = Inline^0 * (spacing^0 * eof / "")
1283 inlines = Ct(inlines_t)
1284
1285 inlines_no_link_t = util.table_copy(inlines_t)
1286 inlines_no_link_t.Link = fail
1287 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 T_EX output and returns it. Note that the converter assumes that the input has UNIX line endings.

```

1288 function self.convert(input)
1289     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.

```

1290     local opt_string = {}
1291     for k,_ in pairs(defaultOptions) do
1292         local v = options[k]
1293         if k ~= "cacheDir" then
1294             opt_string[#opt_string+1] = k .. "=" .. tostring(v)
1295         end
1296     end
1297     table.sort(opt_string)
1298     local salt = table.concat(opt_string, ",")

```

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

```

1299     local name = util.cache(options.cacheDir, input, salt, function(input)
1300         return util.rope_to_string(parse_blocks(input)) .. writer.eof
1301     end, ".md" .. writer.suffix)
1302     return writer.pack(name)
1303 end
1304 return self
1305 end

```

3.1.4 Conversion from Markdown to Plain T_EX

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`.

```

1306 function M.new(options)
1307     local writer = M.writer.new(options)
1308     local reader = M.reader.new(writer, options)

```

```

1309 return reader.convert
1310 end
1311
1312 return M

```

3.2 Plain T_EX Implementation

The plain T_EX implementation provides macros for the interfacing between T_EX 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 T_EX exposed by the plain T_EX interface (see Section 2.2).

3.2.1 Logging Facilities

```

1313 \def\markdownInfo#1{%
1314   \message{(1.\the\inputlineno) markdown.tex info: #1.}}%
1315 \def\markdownWarning#1{%
1316   \message{(1.\the\inputlineno) markdown.tex warning: #1}}%
1317 \def\markdownError#1#2{%
1318   \errhelp{#2.}}%
1319 \errmessage{(1.\the\inputlineno) markdown.tex error: #1}}%

```

3.2.2 Options

The following definitions should be considered placeholder.

```

1320 \def\markdownRendererLineBreakPrototype{\hfil\break}%
1321 \let\markdownRendererEllipsisPrototype\dots
1322 \long\def\markdownRendererCodeSpanPrototype#1{{\tt#1}}%
1323 \long\def\markdownRendererLinkPrototype#1#2#3{#1}%
1324 \long\def\markdownRendererImagePrototype#1#2#3{#1}%
1325 \def\markdownRendererULBeginPrototype{%
1326 \def\markdownRendererULBeginTightPrototype{}%
1327 \def\markdownRendererULItemPrototype{}%
1328 \def\markdownRendererULEndPrototype{}%
1329 \def\markdownRendererULEndTightPrototype{}%
1330 \def\markdownRendererOLBeginPrototype{%
1331 \def\markdownRendererOLBeginTightPrototype{}%
1332 \def\markdownRendererOLItemPrototype{}%
1333 \long\def\markdownRendererOLItemWithNumberPrototype#1{}%
1334 \def\markdownRendererOLEndPrototype{}%
1335 \def\markdownRendererOLEndTightPrototype{}%
1336 \def\markdownRendererDLBeginPrototype{%
1337 \def\markdownRendererDLBeginTightPrototype{}%
1338 \long\def\markdownRendererDLItemPrototype#1{#1}%
1339 \def\markdownRendererDLEndPrototype{}%
1340 \def\markdownRendererDLEndTightPrototype{}%

```

```

1341 \long\def\markdownRendererEmphasisPrototype#1{\it#1}%
1342 \long\def\markdownRendererStrongEmphasisPrototype#1{\b\it#1}%
1343 \def\markdownRendererBlockQuoteBeginPrototype{\par\begin\group\it}%
1344 \def\markdownRendererBlockQuoteEndPrototype{\end\group\par}%
1345 \long\def\markdownRendererInputVerbatimPrototype#1{%
1346   \par{\tt\input"#1"\relax}\par}%
1347 \long\def\markdownRendererHeadingOnePrototype#1{#1}%
1348 \long\def\markdownRendererHeadingTwoPrototype#1{#1}%
1349 \long\def\markdownRendererHeadingThreePrototype#1{#1}%
1350 \long\def\markdownRendererHeadingFourPrototype#1{#1}%
1351 \long\def\markdownRendererHeadingFivePrototype#1{#1}%
1352 \long\def\markdownRendererHeadingSixPrototype#1{#1}%
1353 \def\markdownRendererHorizontalRulePrototype{}%
1354 \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`.

```

1355 \def\markdownLuaOptions{%
1356   \ifx\markdownOptionBlankBeforeBlockquote\undefined\else
1357     blankBeforeBlockquote = \markdownOptionBlankBeforeBlockquote,
1358   \fi
1359   \ifx\markdownOptionBlankBeforeHeading\undefined\else
1360     blankBeforeHeading = \markdownOptionBlankBeforeHeading,
1361   \fi
1362   \ifx\markdownOptionCacheDir\undefined\else
1363     cacheDir = "\markdownOptionCacheDir",
1364   \fi
1365   \ifx\markdownOptionDefinitionLists\undefined\else
1366     definitionLists = \markdownOptionDefinitionLists,
1367   \fi
1368   \ifx\markdownOptionHashEnumerators\undefined\else
1369     hashEnumerators = \markdownOptionHashEnumerators,
1370   \fi
1371   \ifx\markdownOptionHybrid\undefined\else
1372     hybrid = \markdownOptionHybrid,
1373   \fi
1374   \ifx\markdownOptionFootnotes\undefined\else
1375     footnotes = \markdownOptionFootnotes,
1376   \fi
1377   \ifx\markdownOptionPreserveTabs\undefined\else
1378     preserveTabs = \markdownOptionPreserveTabs,
1379   \fi

```

```

1380 \ifx\markdownOptionSmartEllipses\undefined\else
1381     smartEllipses = \markdownOptionSmartEllipses,
1382 \fi
1383 \ifx\markdownOptionStartNumber\undefined\else
1384     startNumber = \markdownOptionStartNumber,
1385 \fi
1386 \ifx\markdownOptionTightLists\undefined\else
1387     tightLists = \markdownOptionTightLists,
1388 \fi}
1389 }%

```

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.

```

1390 \def\markdownPrepare{%

```

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

```

1391     local lfs = require("lfs")
1392     local cacheDir = "\markdownOptionCacheDir"
1393     if lfs.isdir(cacheDir) == true then else
1394         assert(lfs.mkdir(cacheDir))
1395     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.

```

1396     local md = require("markdown")
1397     local convert = md.new(\markdownLuaOptions)
1398 }%

```

3.2.4 Lua `\write18` 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 through the output file stream 18 ($\text{\X}\text{\TeX}$, $\text{\pdf}\text{\TeX}$). 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 $\text{\Lua}\text{\TeX}$ engine, their \TeX distribution contains it, and uses shell access to produce and execute Lua scripts using the $\text{\TeX}\text{\Lua}$ interpreter (see [1, Section 3.1.1]).

```

1399 \ifx\directlua\undefined

```

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`.

```
1400 \csname newwrite\endcsname\markdownLuaExecuteFileStream
```

The `\markdownLuaExecuteShellEscape` macro contains the numeric value of either the `\pdfshellescape` (Lua_T_EX, Pdf_T_EX) or the `\shellescape` (X_T_EL_A) commands. This value indicates, whether the shell access is enabled (1), disabled (0), or restricted (2). If neither of these commands is defined, act as if the shell access were enabled.

```
1401 \csname newcount\endcsname\markdownLuaExecuteShellEscape
```

```
1402 \ifx\pdfshellescape\undefined
```

```
1403 \ifx\shellescape\undefined
```

```
1404 \markdownLuaExecuteShellEscape=1%
```

```
1405 \else
```

```
1406 \markdownLuaExecuteShellEscape=\shellescape
```

```
1407 \fi
```

```
1408 \else
```

```
1409 \markdownLuaExecuteShellEscape=\pdfshellescape
```

```
1410 \fi
```

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

```
1411 \def\markdownLuaExecute#1{%
```

If the shell is accessible, create the file `\markdownOptionHelperScriptFileName` and fill it with the input Lua code prepended with `kpathsea` initialization, so that Lua modules from the _T_EX distribution are available.

```
1412 \ifnum\markdownLuaExecuteShellEscape=1%
```

```
1413 \immediate\openout\markdownLuaExecuteFileStream=%
```

```
1414 \markdownOptionHelperScriptFileName
```

```
1415 \markdownInfo{Writing a helper Lua script to the file
```

```
1416 "\markdownOptionHelperScriptFileName"}%
```

```
1417 \immediate\write\markdownLuaExecuteFileStream{%
```

```
1418 local kpse = require('kpse')
```

```
1419 kpse.set_program_name('luatex') #1}%
```

```
1420 \immediate\closeout\markdownLuaExecuteFileStream
```

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

```
1421 \markdownInfo{Executing a helper Lua script from the file
```

```
1422 "\markdownOptionHelperScriptFileName" and storing the result in the
```

```
1423 file "\markdownOptionOutputTempFileName"}%
```

```
1424 \immediate\write18{texlua "\markdownOptionHelperScriptFileName" >
```

```
1425 "\markdownOptionOutputTempFileName"}%
```

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

```

1426     \input\markdownOptionOutputTempFileName\relax
1427     \else

If the shell is inaccessible, let the user know and suggest a remedy.

1428     \markdownError{I can not access the shell}{Either run the TeX
1429         compiler with the --shell-escape or the --enable-write18 flag,
1430         or set shell_escape=t in the texmf.cnf file}%
1431     \fi}%

```

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

```

1432 \begingroup
1433     \catcode'\^^I=12%
1434     \gdef\markdownReadAndConvertTab{^^I}%
1435 \endgroup

```

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

```

1436 \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.

1437     \catcode'\^^M=13%
1438     \catcode'\^^I=13%
1439     \catcode'|=0%
1440     \catcode'\=12%
1441     \gdef\markdownReadAndConvert#1#2{%
1442         \begingroup%

```

Open the `\markdownOptionInputTempFileName` file for writing.

```

1443         \immediate\openout\markdownLuaExecuteFileStream%
1444             \markdownOptionInputTempFileName%
1445         \markdownInfo{Buffering markdown input into the temporary %
1446             input file "\markdownOptionInputTempFileName" and scanning %
1447             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 and tab characters, so that we can retrieve them unaltered.

```

1448         \def\do##1{\catcode'##1=12}\dospecials%
1449         \catcode'|=12%
1450         \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.

```

1451         \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.

```
1452         |ifx|relax##3|relax%
1453         |immediate|write|markdownLuaExecuteFileStream{##1}%
1454         |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.

```
1455         |def^^M{%
1456         |markdownInfo{The ending token sequence was found}%
1457         |immediate|write|markdownLuaExecuteFileStream{%}%
1458         |immediate|closeout|markdownLuaExecuteFileStream%
1459         |endgroup%
1460         |markdownInput|markdownOptionInputTempFileName%
1461         #2}%
1462         |fi%
```

Repeat with the next line.

```
1463         ^^M}%
```

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

```
1464         |catcode'|^^I=13%
1465         |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.

```
1466         |catcode'|^^M=13%
1467         |def^^M##1^^M{%
1468         |def^^M####1^^M{%
1469         |markdownReadAndConvertProcessLine####1#1#1|relax}%
1470         ^^M}%
1471         ^^M}%
```

Reset the character categories back to the former state.

```
1472         |endgroup
```

3.2.5 Direct Lua Access

The following \TeX code is intended for \TeX engines that provide direct access to Lua (Lua \TeX). The `\markdownLuaExecute` and `\markdownReadAndConvert` defined here and in Section 3.2.4 are meant to be transparent to the remaining code.

```
1473 \else
```

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,

```
1474 \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 caret symbol (`^`), the dollar sign (`$`), the backslash symbol (`\`), the percent sign (`%`), and the braces (`{}`) as a part of the Lua syntax.

```
1475 \begin{group}
```

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

```
1476 \catcode'\_ =12%
```

```
1477 \catcode'\$ =12%
```

```
1478 \catcode'\^ =12%
```

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

```
1479 \catcode'\/=6%
```

```
1480 \catcode'\# =12%
```

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

```
1481 \catcode'\@ =14%
```

```
1482 \catcode'\% =12%
```

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

```
1483 \catcode'\| =0%
```

```
1484 \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.

```
1485 |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.

```
1486 |def|markdownReadAndConvertAfter{/2}@
```

```
1487 |markdownInfo{Buffering markdown input and scanning for the
```

```
1488 closing token sequence "/1"}@
```

```
1489 |directlua{@
```

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

```
1490 |markdownPrepare
```

```
1491 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.

```
1492 local ending_sequence = "^.-" .. ([[/1]]):gsub(
```

```
1493 "([%(%)%.%%+%-%*%?[%]%^%$])", "%1")
```

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

```
1494 |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 T_EX, and insert the result into the input line buffer of T_EX.

```
1495     if line:match(ending_sequence) then
1496         |markdownLuaUnregisterIBCallback
1497         local input = table.concat(buffer, "\n") .. "\n\n"
1498         local output = convert(input)
1499         return [[\markdownInfo{The ending token sequence was found}]] ..
1500             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 T_EX.

```
1501     else
1502         buffer[#buffer+1] = line
1503         return [[\]] .. (#buffer == 1 and "fi" or "relax")
1504     end
1505 end}}@
```

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

```
1506 |iffalse}@
```

Reset the character categories back to the former state.

```
1507 |endgroup
1508 \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 T_EX.

```
1509 \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.

```
1510 \catcode'\=0%
1511 \catcode'\=12%
1512 |gdef|markdownInput#1{%
1513     |markdownInfo{Including markdown document "#1"}%
1514     |markdownLuaExecute{%
1515         |markdownPrepare
1516         local input = assert(io.open("#1","r")):read("*a") .. "\n\n"
```

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

```
1517     print(convert(input:gsub("\r\n?", "\n")))}%
1518 |endgroup
```

3.3 \LaTeX Implementation

The \LaTeX implementation 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.

```
1519 \input markdown
1520 \ProvidesPackage{markdown}[\markdownVersion]%
```

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.

```
1521 \renewcommand\markdownInfo[1]{\PackageInfo{markdown}{#1}}%
1522 \renewcommand\markdownWarning[1]{\PackageWarning{markdown}{#1}}%
1523 \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).

```
1524 \let\markdownInputPlainTeX\markdownInput
1525 \renewcommand\markdownInput[2][{}]{%
1526   \begingroup
1527     \markdownSetup{#1}%
1528     \markdownInputPlainTeX{#2}%
1529   \endgroup}%

```

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

```
1530 \renewenvironment{markdown}{%
1531   \markdownReadAndConvert@markdown{}}\relax
1532 \renewenvironment{markdown*}[1]{%
1533   \markdownSetup{#1}%
1534   \markdownReadAndConvert@markdown*}\relax
1535 \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*.

```

1536 \catcode'\|=0\catcode'\<=1\catcode'\>=2%
1537 \catcode'\|=12\catcode'\{=12\catcode'\}=12%
1538 |gdef|markdownReadAndConvert@markdown#1<%
1539     |markdownReadAndConvert<\end{markdown#1}>%
1540                                     <|end<markdown#1>>>%
1541 |endgroup

```

3.3.3 Options

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

```

1542 \DeclareOption*{%
1543     \expandafter\markdownSetup\expandafter{\CurrentOption}}%
1544 \ProcessOptions\relax

```

The following configuration should be considered placeholder.

```

1545 \RequirePackage{url}
1546 \RequirePackage{graphicx}

```

If the `\markdownOptionTightLists` macro expands to `false`, do not load the `paralist` package.

```

1547 \RequirePackage{ifthen}
1548 \ifx\markdownOptionTightLists\undefined
1549     \RequirePackage{paralist}
1550 \else
1551     \ifthenelse{\equal{\markdownOptionTightLists}{false}}{ }{
1552         \RequirePackage{paralist}}
1553 \fi
1554 \RequirePackage{fancyvrb}
1555 \markdownSetup{rendererPrototypes={
1556     lineBreak = {\},
1557     codeSpan = {\texttt{#1}},
1558     link = {#1\footnote{\ifx\empty#3\empty\else#3:
1559         \fi\texttt<\url{#2}\texttt>}},
1560     image = {\begin{figure}
1561         \begin{center}%
1562             \includegraphics{#2}%
1563         \end{center}%
1564         \ifx\empty#3\empty\else
1565             \caption{#3}%
1566         \fi
1567         \label{fig:#1}%
1568     \end{figure}},
1569     ulBegin = {\begin{itemize}},
1570     ulBeginTight = {\begin{compactitem}},
1571     ulItem = {\item},

```

```

1572 ulEnd = {\end{itemize}},
1573 ulEndTight = {\end{compactitem}},
1574 olBegin = {\begin{enumerate}},
1575 olBeginTight = {\begin{compactenum}},
1576 olItem = {\item},
1577 olItemWithNumber = {\item[#1.]},
1578 olEnd = {\end{enumerate}},
1579 olEndTight = {\end{compactenum}},
1580 dlBegin = {\begin{description}},
1581 dlBeginTight = {\begin{compactdesc}},
1582 dlItem = {\item[#1]},
1583 dlEnd = {\end{description}},
1584 dlEndTight = {\end{compactdesc}},
1585 emphasis = {\emph{#1}},
1586 strongEmphasis = {%
1587   \ifx\alert\undefined
1588     \textbf{\emph{#1}}%
1589   \else % Beamer support
1590     \alert{\emph{#1}}
1591   \fi},
1592 blockQuoteBegin = {\begin{quotation}},
1593 blockQuoteEnd = {\end{quotation}},
1594 inputVerbatim = {\VerbatimInput{#1}},
1595 horizontalRule = {\noindent\rule[0.5ex]{\linewidth}{1pt}},
1596 footnote = {\footnote{#1}}}%
1597
1598 \ifx\chapter\undefined
1599   \markdownSetup{rendererPrototypes={
1600     headingOne = {\section{#1}},
1601     headingTwo = {\subsection{#1}},
1602     headingThree = {\subsubsection{#1}},
1603     headingFour = {\paragraph{#1}},
1604     headingFive = {\subparagraph{#1}}}%
1605   \else
1606     \markdownSetup{rendererPrototypes={
1607       headingOne = {\chapter{#1}},
1608       headingTwo = {\section{#1}},
1609       headingThree = {\subsection{#1}},
1610       headingFour = {\subsubsection{#1}},
1611       headingFive = {\paragraph{#1}},
1612       headingSix = {\subparagraph{#1}}}%
1613     \fi

```

3.3.4 Miscellanea

Unlike base Lua \TeX , which only allows for a single registered function per a callback (see [1, Section 8.1]), the $\text{\LaTeX}2_{\epsilon}$ 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 $\text{\TeX}2_{\epsilon}$ format, we need to redefine the `\markdownLuaRegisterIBCallback` and `\markdownLuaUnregisterIBCallback` macros accordingly.

```
1614 \renewcommand\markdownLuaRegisterIBCallback[1]{%
1615   luatexbase.add_to_callback("process_input_buffer", #1, %
1616     "The markdown input processor")}
1617 \renewcommand\markdownLuaUnregisterIBCallback{%
1618   luatexbase.remove_from_callback("process_input_buffer",%
1619     "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.

```
1620 \newcommand\markdownMakeOther{%
1621   \count0=128\relax
1622   \loop
1623     \catcode\count0=11\relax
1624     \advance\count0 by 1\relax
1625   \ifnum\count0<256\repeat}%
```

3.4 Con \TeX t Implementation

The Con \TeX t implementation makes use of the fact that, apart from some subtle differences, the Mark II and Mark IV Con \TeX t 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.

```
1626 \def\dospecials{\do\ \do\\\do\{\do\}\do\$\do\&%
1627   \do\#\do\^\do\_do\%\do\~}%
```

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

```
1628 \ifx\directlua\undefined
1629   \input markdown
1630 \else
```

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

```
1631   \directlua{%
1632     local function unescape(str)
1633       return (str:gsub("|", string.char(92))) end
1634     local old_callback = callback.find("process_input_buffer")
1635     callback.register("process_input_buffer", function() end)
```

```
1636     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. Therefore, we will just force the plain T_EX implementation to use the `\write18` bridge (see Section 3.2.4) by locally undefining the `\directlua` primitive.

```
1637     if new_callback == false then
1638         tex.print(unescape([[|let|markdownDirectLua|directlua
1639                             |let|directlua|undefined
1640                             |input markdown
1641                             |let|directlua|markdownDirectLua
1642                             |let|markdownDirectLua|undefined]]))
```

If we can, then just load the plain T_EX implementation.

```
1643     else
1644         callback.register("process_input_buffer", old_callback)
1645         tex.print(unescape("|input markdown"))
1646     end}%
1647 \fi
```

3.4.1 Logging Facilities

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

```
1648 \def\markdownInfo#1{\writestatus{markdown}{#1.}}%
1649 \def\markdownWarning#1{\writestatus{markdown\space warn}{#1.}}%
```

3.4.2 Typesetting Markdown

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

```
1650 \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*.

```
1651     \catcode'\|=0%
1652     \catcode'\=12%
1653     |gdef|startmarkdown{%
1654         |markdownReadAndConvert{\stopmarkdown}%
1655         {|stopmarkdown}}%
1656 |endgroup
```


3.4.3 Options

The following configuration should be considered placeholder.

```
1657 \def\markdownRendererLineBreakPrototype{\blank}%
1658 \long\def\markdownRendererLinkPrototype#1#2#3{%
1659   \useURL[#1][#2][#3]#1\footnote[#1]{\ifx\empty#3\empty\else#3:
1660   \fi\tt<\hyphenatedurl{#2}>}}%
1661 \long\def\markdownRendererImagePrototype#1#2#3{%
1662   \placefigure[][fig:#1]{#3}{\externalfigure[#2]}}%
1663 \def\markdownRendererUlBeginPrototype{\startitemize}%
1664 \def\markdownRendererUlBeginTightPrototype{\startitemize[packed]}%
1665 \def\markdownRendererUlItemPrototype{\item}%
1666 \def\markdownRendererUlEndPrototype{\stopitemize}%
1667 \def\markdownRendererUlEndTightPrototype{\stopitemize}%
1668 \def\markdownRendererOlBeginPrototype{\startitemize[n]}%
1669 \def\markdownRendererOlBeginTightPrototype{\startitemize[packed,n]}%
1670 \def\markdownRendererOlItemPrototype{\item}%
1671 \long\def\markdownRendererOlItemWithNumberPrototype#1{\sym{#1.}}%
1672 \def\markdownRendererOlEndPrototype{\stopitemize}%
1673 \def\markdownRendererOlEndTightPrototype{\stopitemize}%
1674 \definedescription
1675   [markdownConTeXtDlItemPrototype]
1676   [location=hanging,
1677   margin=standard,
1678   headstyle=bold]%
1679 \definestartstop
1680   [MarkdownConTeXtDlPrototype]
1681   [before=\blank,
1682   after=\blank]%
1683 \definestartstop
1684   [MarkdownConTeXtDlTightPrototype]
1685   [before=\blank\startpacked,
1686   after=\stoppacked\blank]%
1687 \def\markdownRendererDlBeginPrototype{%
1688   \startMarkdownConTeXtDlPrototype}%
1689 \def\markdownRendererDlBeginTightPrototype{%
1690   \startMarkdownConTeXtDlTightPrototype}%
1691 \long\long\def\markdownRendererDlItemPrototype#1{%
1692   \markdownConTeXtDlItemPrototype{#1}}%
1693 \def\markdownRendererDlEndPrototype{%
1694   \stopMarkdownConTeXtDlPrototype}%
1695 \def\markdownRendererDlEndTightPrototype{%
1696   \stopMarkdownConTeXtDlTightPrototype}%
1697 \long\def\markdownRendererEmphasisPrototype#1{\em#1}%
1698 \long\def\markdownRendererStrongEmphasisPrototype#1{\bf\em#1}%
1699 \def\markdownRendererBlockQuoteBeginPrototype{\startquotation}%
1700 \def\markdownRendererBlockQuoteEndPrototype{\stopquotation}%
```

```

1701 \long\def\markdownRendererInputVerbatimPrototype#1{\typefile{#1}}%
1702 \long\def\markdownRendererHeadingOnePrototype#1{\chapter{#1}}%
1703 \long\def\markdownRendererHeadingTwoPrototype#1{\section{#1}}%
1704 \long\def\markdownRendererHeadingThreePrototype#1{\subsection{#1}}%
1705 \long\def\markdownRendererHeadingFourPrototype#1{\subsubsection{#1}}%
1706 \long\def\markdownRendererHeadingFivePrototype#1{\subsubsubsection{#1}}%
1707 \long\def\markdownRendererHeadingSixPrototype#1{\subsubsubsubsection{#1}}%
1708 \def\markdownRendererHorizontalRulePrototype{%
1709   \blackrule[height=1pt, width=\hsize]}%
1710 \long\def\markdownRendererFootnotePrototype#1{\footnote{#1}}%
1711 \stopmodule\protect

```

References

1. L^AT_EX DEVELOPMENT TEAM. *LuaT_EX 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 T_EXbook*. 3rd ed. Addison-Westley, 1986. ISBN 0201134470.
3. IERUSALIMSKY, Roberto. *Programming in Lua*. 3rd ed. Rio de Janeiro: PUC-Rio, 2013. ISBN 9788590379850.
4. BRAAMS, Johannes; CARLISLE, David; JEFFREY, Alan; LAMPORT, Leslie; MITTELBACH, Frank; ROWLEY, Chris; SCHÖPF, Rainer. *The L^AT_EX 2_ε Sources* [online]. 2016 [visited on 2016-06-02]. Available from: <http://mirrors.ctan.org/macros/latex/base/source2e.pdf>.