spelling*

Stephan Hennig†

23rd May 2013

Abstract

This package supports spell-checking of TEX documents compiled with the LuaTEX engine. It can give visual feedback in PDF output similar to WYSIWYG word processors. The package relies on an external spell-checker application that can check a plain text file and output a list of bad spellings. The package should work with most spell-checkers, even dumb, TEX-unaware ones.

Warning! This package is in a very early state. Everything may change!

Contents

1 Introduction

Ther[1] are three main approaches to spell-checking TEX documents:

1. checking spelling in the `.tex` source file,

*This document describes the spelling package v0.4.

†sh2d@arcor.de

[1] A footnote containing mispellings.

2. converting a `.tex` file to another format, for which a proved spell-checking solution exists,

3. checking spelling after a `.tex` file has been processed by TEX.

All of these approaches have their strengths and weaknesses. This package follows the third approach, providing some unique features:

- In traditional solutions, text is extracted from typeset DVI, PS or PDF files, including hyphenated words. To avoid (lots of) false positives being reported by the spell-checker, hyphenation needs to be switched off during the TEX run. That is, one doesn't work on the original document any more.

  In contrast to that, the spelling package works transparently on the original `.tex` source file. Text is extracted *during* typesetting, after LuaTEX has applied its catcode and macro machinery, but before hyphenation takes place.

- The spelling package can highlight words with known incorrect spelling in PDF output, giving visual feedback similar to WYSIWYG word processors.[2]

2 Usage

The spelling package requires the LuaTEX engine. All functionality of the package is implemented in Lua. The LATEX interface, which is described below, is effectively a wrapper around the Lua interface.

Implementing such wrappers for other formats shouldn't be too difficult. The author is a LATEX-only user, though, and therefore grateful for contributions. By the way, the LATEX package needs some polishing, too, e. g.*, a key-value interface is desirable. Patches welcome!*

2.1 Work-flow

Here's a short outline of how using the spelling package fits into the general process of compiling a document with LuaTEX:

1. After loading the package in the preamble of a `.tex` source file, a list of bad spellings is read from a file (if that file exists).

[2] Currently, only colouring words is implemented.

2. During the LuaTEX run, text is extracted from pages and all words are checked against the list of bad spellings. Words with a known incorrect spelling are highlighted in PDF output.

3. At the end of the LuaTEX run, in addition to the PDF file, a text file is written, containing most of the text of the typeset document.

4. The text file is then checked by your favourite external spell-checker application, *e. g.*, Aspell or Hunspell. The spell-checker should be able to write a list of bad spellings to a file. Otherwise, visual feedback in PDF output won't work.

5. Visually minded people may now compile their document a second time. This time, the new list of bad spellings is read-in and words with incorrect spelling found by the spell-checker should now be highlighted in PDF output. Users can then apply the necessary corrections to the `.tex` source file.

Whatever way spell-checker output is employed, users not interested in visual feedback (because their spell-checker has an interactive mode only or because they prefer grabbing bad spellings from a file directly) can also benefit from this package. Using it, LuaTEX writes a pure text file that is particularly well suited as spell-checker input, because it contains no hyphenated words (and neither macros, nor active characters). That way, any spell-checker application, even TEX-unaware ones, can be used to check spelling of TEX documents.

2.2 Word lists

As described above, after loading the spelling package, a list of bad spellings is read from a file ⟨*jobname*⟩`.spell.bad`, if that file exists. Words found in this file are stored in an internal list of bad spellings and are later used for highlighting spelling mistakes in PDF output. Additionally, a list of good spellings is read from a file ⟨*jobname*⟩`.spell.good`, if that file exists. Words found in the latter file are stored in an internal list of good spellings. File format for both files is one word per line. Files must be in the UTF-8 encoding. Letter case is significant.

A word in the document is highlighted, if it occurs in the internal list of bad spellings, but not in the internal list of good spellings. That is, known good spellings take precedence over known bad spellings.

Users can load additional files containing lists of bad or good spellings with macros `\spellingreadbad` and `\spellingreadgood`. Argument to

`\spellingreadbad`
`\spellingreadgood`

both macros is a file name. If a file cannot be found, a warning is written to the console and `log` file and compilation continues. As an example, the command

```
\spellingreadgood{myproject.whitelist}
```

reads words from a file `myproject.whitelist` and adds them to the list of good spellings.

Known good spellings can be used to deal with words wrongly reported as bad spellings by the spell-checker (false positives). But note, most spell-checkers also provide means to deal with unknown words via additional dictionaries. It is recommended to configure your spell-checker to report as few false positives as possible.

2.3 Match rules

This section describes an advanced feature. You may safely skip this section upon first reading.

The spelling package provides an additional way to deal with bad and good spellings, match rules. Match rules can be used to employ regular patterns within certain 'words'. A typical example are bibliographic references like *Lin86*, which are often flagged by spell-checkers, but need not be highlighted as they are generated by TEX.

There are two kinds of rules, bad and good rules. A rule is a Lua function whose boolean return value indicates whether a word *matches* the rule. A bad rule should return a true value for all strings identified as bad spellings, otherwise a false value. A good rule should return a true value for all strings identified as good spellings, otherwise a false value. A word in the document is highlighted if it matches any bad rule, but no good rule.

Function arguments are a *raw* string and a *stripped* string. The raw string is a string representing a word as it is found in the document possibly surrounded by punctuation characters. The stripped string is the same string with surrounding punctuation already stripped.

As an example, the rule in Listing 1 matches all words consisting of exactly three letters. The function matches the stripped string against the Lua string pattern `^%a%a%a$` via function `unicode.utf8.find` from the Selene Unicode library. The latter function is a UTF-8 capable version of Lua's built-in function `string.find`. It returns `nil` (a false value) if there has been no match and a number (a true value) if there has been a match. The string `%a` represents a string class matching a single letter. Characters `^` and `$` are anchors for the beginning and the end of the string in question.

Listing 1: Matching three-letter words.

```
function three_letter_words(raw, stripped)
  return unicode.utf8.find(stripped, '^%a%a%a$')
end
```

Listing 2: Matching double punctuation.

```
function double_punctuation(raw, stripped)
  return unicode.utf8.find(raw, '%p%p')
end
```

Note, the pattern `%a%a%a` without anchors would match any string which contains *at least* three letters in a row. More information about Lua string patterns can be found in the Lua reference manual[3], the Selene Unicode library documentation[4] and in the Unicode standard[5].

Listing 2 shows a rule matching all 'words' containing double punctuation.

The rule in Listing 3 combines the results of three string searches to match bibliographic references as generated by the BibTEX *alpha* style.

Match rules have to be provided by means of a Lua module. Such modules can be loaded with the `\spellingmatchrules` command. Argument is a module name. To tell bad rules from good rules, the table returned by the module must follow this convention: Function identifiers representing bad and good match rules are prefixed `bad_rule_` and `good_rule_`, resp. The rest of an identifier is actually irrelevant. Other and non-function identifiers are ignored. `\spellingmatchrules`

Listing 4 shows an example module declaring the rules from Listing 1 and Listing 2 as *bad* match rules and the rule from Listing 3 as a *good* match rule. Note, how function identifiers are made local and how exported function identifiers are prefixed `bad_rule_` and `good_rule_`, while local function identifiers have no prefixes. When the module resides in a file named `myproject.rules.lua`, it can be loaded in the preamble of a document via

```
\spellingmatchrules{myproject.rules}
```

[3] http://www.lua.org/manual/5.2/manual.html#6.4

[4] https://github.com/LuaDist/slnunicode/blob/master/unitest

[5] http://www.unicode.org/Public/4.0-Update1/UCD-4.0.1.html#General_Category_Values

Listing 3: Matching references generated by the BibTEX alpha style.

```
function bibtex_alpha(raw, stripped)
  return unicode.utf8.find(stripped, '^%u%l%l%d%d$')
    or unicode.utf8.find(stripped, '^%u%u%u?%u?%d%d$')
    or unicode.utf8.find(stripped, '^%u%u%u%+%d%d$')
end
```

Listing 4: A module containing two bad and one good match rule.

```
-- Module table.
local M = {}

-- Import Selene Unicode library.
local unicode = require('unicode')
-- Add short-cut.
local Ufind = unicode.utf8.find

local function three_letter_words(raw, stripped)
  return Ufind(stripped, '^%a%a%a$')
end
M.bad_rule_three_letter_words = three_letter_words

local function double_punctuation(raw, stripped)
  return Ufind(raw, '%p%p')
end
M.bad_rule_double_punctuation = double_punctuation

local function bibtex_alpha(raw, stripped)
  return Ufind(stripped, '^%u%l%l%d%d$')
    or Ufind(stripped, '^%u%u%u?%u?%d%d$')
    or Ufind(stripped, '^%u%u%u%+%d%d$')
end
M.good_rule_bibtex_alpha = bibtex_alpha

return M
```

How are match rules and lists of bad and good spellings related? Internally, the lists of bad and good spellings are referred to by two special default match rules, that look-up raw and stripped strings and return a true value if either argument has been found in the corresponding list. Since good rules take precedence over bad rules, an entry in the list of good spellings takes precedence over any user-supplied bad rule. Likewise, any user-supplied good rule takes precedence over an entry in the list of bad spellings.

Some final remarks on match rules:

- It must be stressed that the boolean return value of a match rule *does not* indicate whether a spelling is bad or good, but whether a word matches a certain rule or not. Whether it's a bad or a good spelling, depends on the name of the match rule in the module table.
- When written without care, match rules can easily produce false positives as well as false negatives. While false positives in bad rules and false negatives in good rules can easily be spotted due to the unexpected highlighting of words, the other cases are more problematic. To avoid all kinds of false results, match rules should be stated as specific as possible.
- Match rules are only called upon the first occurrence of a spelling in a document. The information, whether a spelling needs to be highlighted, is stored in a cache table. Subsequent occurrences of a spelling just need a table look-up to determine highlighting status. For that reason, it is safe to do relatively expensive operations within a match rule without affecting compilation time too much. Nevertheless, match rules should be stated as efficient as possible.[6]

2.4 Highlighting spellling mistakes

Enabling/disabling Highlighting spelling mistakes (words with known incorrect spelling) in PDF output can be toggled on and off with command `\spellinghighlight`. If the argument is `on`, highlighting is enabled. For other arguments, highlighting is disabled. Highlighting is enabled, by default.

`\spellinghighlight`

[6] Some Lua performance tips can be found in the book *Lua Programming Gems* by Figueiredo, Celes and Ierusalimschy *(eds.)*, 2008, ch. 2. That chapter is also available online at `http://www.lua.org/gems/`.

Colour The colour used for highlighting bad spellings can be determined by command `\spellinghighlightcolor`. Argument is a colour statement in the PDF language. As an example, the colour red in the RGB colour space is represented by the statement `1 0 0 rg`. In the CMYK colour space, a reddish colour is represented by `0 1 1 0 k`. Default colour used for highlighting is `1 0 0 rg`, *i. e.*, red in the RGB colour space.

2.5 Text output

Text file After loading the spelling package, at the end of the LuaTEX run, a text file is written that contains most of the document text. The text file is no close text rendering of the typeset document, but serves as input for your favourite spell-checker application. It contains the document text in a simple format: paragraphs separated by blank lines. A paragraph is anything that, during typesetting, starts with a `local_par` whatsit node in the node list representing a typeset page of the original document, *e. g.*, paragraphs in running text, footnotes, marginal notes, (in-lined) `\parbox` commands or cells from `p`-like table columns *etc.*

Paragraphs consist of words separated by spaces. A word is the textual representation of a chain of consecutive nodes of type `glyph`, `disc` or `kern`. Boxes are processed transparently. That is, the spelling package (highly imperfectly) tries to recognise as a single word what in typeset output looks like a single word. As an example, the LaTeX code

```
foo\mbox{'s bar}s
```

which is typeset as

foo's bars

is considered two words *foo's* and *bars*, instead of the four words *foo*, *'s*, *bar* and *s*.[7]

Enabling/disabling Text output can be toggled on and off with command `\spellingoutput`. If the argument is `on`, text output is enabled. For other arguments, text output is disabled. Text output is enabled, by default. `\spellingoutput`

File name Text output file name can be configured via command `\spellingoutputname`. Argument is the new file name. Default text output file name is ⟨*jobname*⟩`.spell.txt`. `\spellingoutputname`

[7] This document has been compiled with a custom list of bad spellings, which is why the word *foo's* should be highlighted.

Line length In text output, paragraphs can either be put on a single line or broken into lines of a fixed length. The behaviour can be controlled via command `\spellingoutputlinelength`. Argument is a number. If the number is less than 1, paragraphs are put on a single line. For larger arguments, the number specifies maximum line length. Note, lines are broken at spaces only. Words longer than maximum line length are put on a single line exceeding maximum line length. Default line length is 72.

`\spellingoutputlinele`

2.6 Text extraction

Enabling/disabling Text extraction can be enabled and disabled in the document via command `\spellingextract`. If the argument is `on`, text extraction is enabled. For other arguments, text extraction is disabled. The command should be used in vertical mode, *i. e.*, outside paragraphs. If text extraction is disabled in the document preamble, an empty text file is written at the end of the LuaTEX run. Text extraction is enabled, by default.

`\spellingextract`

Note, text extraction and visual feedback are orthogonal features. That is, if text extraction is disabled for part of a document, *e. g.*, a long table, words with a known incorrect spelling are still highlighted in that part.

2.7 Code point mapping

As explained in subsection 2.5, the text file written at the end of the LuaTEX run is in the UTF-8 encoding. Unicode contains a wealth of code points with a special meaning, such as ligatures, alternative letters, symbols *etc.* Unfortunately, not all spell-checker applications are smart enough to correctly interpret all Unicode code points that may occur in a document. For that reason, a code point mapping feature has been implemented that allows for mapping certain Unicode code points that may appear in a node list to arbitrary strings in text output. A typical example is to map ligatures to the characters corresponding to their constituting letters. The default mappings applied can be found in Table 1.

Additional mappings can be declared by command `\spellingmapping`. This command takes two arguments, a number that refers to the Unicode code point, and a sequence of arbitrary characters that is the mapping target. The code point number must be in a format that can be parsed by Lua. The characters must be in the UTF-8 encoding.

`\spellingmapping`

New mappings only have effect on the following document text. The command should therefore be used in the document preamble. As an example, the character `A` can be mapped to `Z` and *vice versa* with the following

Unicode name	sample glyph[a]	code point	target characters
LATIN CAPITAL LIGATURE IJ	Ĳ	0x0132	IJ
LATIN SMALL LIGATURE IJ	ĳ	0x0133	ij
LATIN CAPITAL LIGATURE OE	Œ	0x0152	OE
LATIN SMALL LIGATURE OE	œ	0x0153	oe
LATIN SMALL LETTER LONG S	ſ	0x017f	s
LATIN SMALL LIGATURE FF	ff	0xfb00	ff
LATIN SMALL LIGATURE FI	fi	0xfb01	fi
LATIN SMALL LIGATURE FL	fl	0xfb02	fl
LATIN SMALL LIGATURE FFI	ffi	0xfb03	ffi
LATIN SMALL LIGATURE FFL	ffl	0xfb04	ffl
LATIN SMALL LIGATURE LONG S T	ſt	0xfb05	st
LATIN SMALL LIGATURE ST	st	0xfb06	st

Table 1: Default code point mappings.

[a] Sample glyphs are taken from font *Linux Libertine O*.

code:

```
\spellingmapping{65}{Z}% A => Z
\spellingmapping{90}{A}% Z => A
```

Another command `\spellingclearallmappings` can be used to remove all existing code point mappings. `\spellingclearallmapp`

2.8 Tables

How do tables fit into the simple text file format that has only paragraphs and blank lines as described in subsection 2.5? What is a paragraph with regards to tables? A whole table? A row? A single cell?

By default, only text from cells in `p`(aragraph)-like columns is put on their own paragraph, because the corresponding node list branches contain a `local_par` whatsit node (*cf.* subsection 2.5). The behaviour can be changed with the `\spellingtablepar` command. This command takes as argument a number. If the argument is 0, the behaviour is described as above. If the argument is 1, a blank line is inserted before and after every table row (but at most once between table rows). If the argument is 2, a blank line is inserted before and after every table cell. By default, no blank lines are inserted. `\spellingtablepar`

3 LanguageTool support

Installing spell-checkers and dictionaries can be a difficult task if there are no pre-built packages available for an architecture. That's one reason the spelling package is rather spell-checker agnostic and the manual doesn't recommend a particular spell-checker application. Another reason is, there is no best spell-checker. The only recommendation the author makes is not to trust in one spell-checker, but to use multiple spell-checkers at the same time, with different dictionaries or, better yet, different checking engines under the hood.

Among the set of options available, LanguageTool[8], a style and grammar checker that can also check spelling since version 1.8, deserves some notice for its portability, ease of installation and active development. For these reasons, the spelling package provides explicit LanguageTool support. LanguageTool uses Hunspell as the spell-checking engine, augmenting results with a rule based engine and a morphological analyser (depending on the language). The spelling package can parse LanguageTool's error reports in the XML format, pick those errors that are spelling related and use them to highlight bad spellings.[9]

3.1 Installation

Here are some brief installation instructions for the stand-alone version of LanguageTool (tested with LanguageTool 2.1). The stand-alone version contains a GUI as well as a command-line interface. For the spelling package, the latter is needed.

1. LanguageTool is primarily written in Java. Make sure a recent Java Runtime Environment (JRE) is installed.

2. Open a command-line and type

   ```
   java -version
   ```

   If you get an error message, find out the full path to the Java executable (called `java.exe` on Windows) for later reference.

3. Download the stand-alone version of LanguageTool (should be a ZIP archive).

[8] http://www.languagetool.org/

[9] Highlighting style and grammar errors found by LanguageTool should be possible, but requires major restructuring of the spelling package.

4. Uncompress the downloaded archive to a location of your choice.

5. Open a command-line in the directory containing file `languagetool-commandline.jar` and type

   ⟨*path to*⟩`/java -jar languagetool-commandline.jar --help`

   Prepending the path to the Java executable is optional, depending on the result in step 2. If you now see a list of LanguageTool's command-line options rush by, all is well.

6. For easier access to LanguageTool, create a small batch script and put that somewhere into the `PATH`.

   - For users of unixoide systems, the script might look like

     ```
     #!/bin/sh
     ⟨path to⟩/java -jar ⟨path to⟩/languagetool-commandline.jar $*
     ```

     where ⟨*path to*⟩ should point to the Java executable (optional) and file `languagetool-commandline.jar` (mandatory). If the script is named `lt.sh`, you should be able to run LanguageTool on the command shell by typing, *e. g.*,

     ```
     sh lt.sh --version
     ```

     Don't forget to put the script into the `PATH`! For other ways of making scripts executable, please consult the operating system documentation.

   - For Windows users, the script might look like

     ```
     @echo off
     ⟨path to⟩\java -jar ⟨path to⟩\languagetool-commandline.jar %*
     ```

     where ⟨*path to*⟩ should point to the Java executable (optional) and file `languagetool-commandline.jar` (mandatory). If the script is named `lt.bat`, you should be able to run LanguageTool on the command-line by typing, *e. g.*,

     ```
     lt --version
     ```

     Don't forget to put the script into the `PATH`!

3.2 Usage

The results of checking a text file with LanguageTool are written to an error report, either in a human readable format or in a machine friendly XML format. The spelling package can only parse the latter format. When it was said in subsection 2.2 that the spelling package reads files ⟨*jobname*⟩`.spell.bad` and ⟨*jobname*⟩`.spell.good`, if they exist, that was not the whole truth. Additionally, a file ⟨*jobname*⟩`.spell.xml` is read, if it exists. This file should contain a LanguageTool error report in the XML format. Additional LanguageTool XML error reports can be loaded via the `\spellingreadLT` command. `\spellingreadLT`
Argument is a file name. Macros `\spellingreadLT`, `\spellingreadbad` and `\spellingreadgood` can be used in combination in a TEX file.

To check a text file and create an error report in the XML format, LanguageTool can be called on the command-line like this

`lt ⟨options⟩ ⟨input file⟩ > ⟨error report⟩`

where `⟨options⟩` is a list of options described below, `⟨input file⟩` is the text file written by the spelling package in the first LuaTEX run and `⟨error report⟩` is the file containing the error report. Note, how standard output is redirected to a file via the `>` operator. By default, LanguageTool writes error reports to standard output, that is, the command-line. Redirection is a feature most operating systems provide.

- Option `-l` determines the language (variant) of the file to check. As an example, language variant US English can be selected via `-l en-US`. The full list of languages supported by LanguageTool can be requested via option `--list`.
- Option `-c` determines the encoding of the input file. Since the text file written by the spelling package is in the UTF-8 encoding, this part should be `-c utf-8`.
- By default, LanguageTool outputs error reports in a human readable format. The spelling package can only parse error reports in the XML format. If the `--api` option is present, LanguageTool outputs XML data.
- Users that don't want to highlight bad spellings, but prefer to study the list of bad spellings themselves, should refer to the `-u` option. But note, that with the latter option present, LanguageTool doesn't output pure XML any more, even if the `--api` option is present. Make sure such error reports aren't read by the spelling package.

- If the `--help` option is present, LanguageTool shows more information about command-line options.

As an example, to compile a LaTeX file `myletter.tex` written in French that uses the spelling package with standard settings to highlight bad spellings and to use LanguageTool as a spell-checker, the following commands should be typed on the command-line:

```
lualatex myletter
lt --api -c utf-8 -l fr myletter.spell.txt > myletter.spell.xml
lualatex myletter
```

4 Bugs

Note, this package is in a very early state. Expect bugs! Package development is hosted at **GitHub**. The full list of known bugs and feature requests can be found in the **issue tracker**. New bugs should be reported there.

The most user-visible issues are listed below:

- There's no support for the Plain TeX or ConTeXt formats other than the API of the package's Lua modules, yet (issue 1).
- Macros provided by the LaTeX package have very long names. A *key-value* package option interface would be much more user-friendly (issue 2).
- There are a couple of issues with text extraction and highlighting incorrect spellings:
  - Text in head and foot lines is neither extracted nor highlighted (issue 7).
  - The first word starting right after an `hlist`, *e. g.*, the first word within an `\mbox`, is never highlighted. It is extracted and written to the text file, though. This might affect acronyms, names *etc.* (issue 6).
  - Bad spellings that are hyphenated at a page break are not highlighted (issue 10).

Patches welcome!

Happy TeXing!