The `transconv` Package

Sievert von Stülpnagel

September 16, 2019

Contents

List of Tables

1 Introduction

1.1 About `transconv`

1.2 License

2 Package Options

`scheme`=⟨*lang*⟩.⟨*scheme*⟩

This option enables support for the transliteration scheme ⟨*scheme*⟩ for the language ⟨*lang*⟩ by setting up the matching converter in Lua and defines the commands `\`⟨*scheme*⟩`convert`, `\`⟨*scheme*⟩`font` and `\to`⟨*scheme*⟩. In order to do this, Lua has to be able to find a file ⟨*lang*⟩/⟨*scheme*⟩.lua which returns the converter. Can be used more than once to import multiple schemes.

`scheme`={⟨*lang1*⟩.⟨*scheme1*⟩, ⟨*lang2*⟩.⟨*scheme2*⟩,...}

Imports multiple schemes at once. Equivalent to a repeated use of `scheme`.

`defaultscheme`=⟨*lang*⟩.⟨*scheme*⟩

Defines the default scheme for the language ⟨*lang*⟩ in case more than one scheme is added for the same language. Can be used more than once to define defaults for multiple languages. In case of repeated use for the same language, later schemes override earlier ones.

TD

`defaultscheme`={⟨*lang1*⟩.⟨*scheme1*⟩, ⟨*lang2*⟩.⟨*scheme2*⟩,...}

Defines default schemes for multiple languages at once. Equivalent to a repeated use of `defaultscheme`. If the same language appears more than once, later schemes override earlier ones.

TD

3 Basic Interface

All commands in this section are robust and can therefore be used in expansion-only contexts (e. g. headings or footnotes). However at the moment you cannot use any commands within the argument of a conversion command. For example:

```
\topinyin{\emph{Zhe3} bu4 xing2.}
```

would result in an error.

3.1 Formatted Conversion Commands

`\tolang`{⟨*text*⟩}

where `lang` is a language declared via `\TransconvUseScheme` or the option `scheme`. This command will convert ⟨*text*⟩ to the default scheme for `lang` (either the one defined by `defaultscheme` or else the last scheme defined for `lang`), using the formatting defined in `\langfont`.

In most cases, this should be the go-to conversion command to use in your text as it abstracts both the scheme itself as well as its formatting away from the content.

TD

`\toscheme{⟨text⟩}`

where `scheme` is a scheme declared via `\TransconvUseScheme` or the option `scheme`. This command will convert ⟨*text*⟩ to the `scheme` using the formatting defined in `\schemefont`.

Use this command to access specific schemes if you need to import multiple ones for a single language (for example if the schemes themselves are a topic of discussion). Note that `\toscheme` is provided for all imported languages, including the default one. If you only intend to use a single theme for a given language, use `\tolang` instead to avoid hardcoding the scheme into your text.

3.2 Unformatted Conversion Commands

`\langconvert{⟨text⟩}`

where `lang` is a language declared via `\TransconvUseScheme` or the option `scheme`. This command will convert ⟨*text*⟩ to the default scheme for `lang` (either the one defined by `defaultscheme` or else the last scheme defined for `lang`), but without any formatting. In other words, `\langconvert` abstracts away the transcription scheme but not the formatting.

It is recommended that you use this command only if the formatting hooks provided by `transconv` are insufficient for your purposes, and only to define a custom macro rather than in the actual text in order to maintain the separation of content and presentation.

TD

`\schemeconvert{⟨text⟩}`

where `scheme` is again a scheme declared via `\TransconvUseScheme` or the option `scheme`.

This command merely constitutes a wrapper around the Lua converter without any TeX formatting. Therefore, using this command means you hard-code both the transcription scheme and the way it is formatted. For this reason, it is highly recommended that you use this command only if the formatting hooks provided by `transconv` are insufficient for your purposes, and even then only to define a custom macro rather than in the actual text in order to maintain the separation of content and presentation.

3.3 Output Formatting

`\langfont`

where `lang` is a language declared via `\TransconvUseScheme` or the option `scheme`.

This command sets the font formatting for the output of `\tolang`. By default it expands to `\itshape`.

`\schemefont`

where `scheme` is a transcription scheme declared via `\TransconvUseScheme` or the option `scheme`.

This command sets the font formatting for the output of `\tolang`. By default it expands to nothing.

4 Configuration

`\TransconvUseScheme`{⟨*schemes*⟩}

Blabla

5 Custom Schemes and Languages

5.1 Adding a New Transliteration Scheme for an Existing Language

To add a new transliteration scheme involves a little bit of Lua programming, though `transconv` is designed with the goal of making it as painless as possible.

As a first step, add a new file ⟨*scheme*⟩.lua in the folder ⟨*lang*⟩. The name of the file will be the scheme name which you have to use to load the scheme later. Open the file in the text editor of your choice and define a new scheme:

```
local MyScheme = Converter:new{
  -- load raw scheme settings for your language
  raw = require(transconv.path_of(...)..".raw"),

  -- settings variables are going to go here
}
```

You use any name you want instead of MyScheme. This name is only used to refer to the scheme within the file itself; it has no consequence outside.

At the end of the file, return the scheme:

```
return MyScheme
```

Technically speaking you are now set; you have successfully defined a new scheme and should be able to load it by passing ⟨*lang*⟩.⟨*scheme*⟩ to `transconv`'s `schemes` option or the `\TransconvUseScheme` macro (make sure to use the folder and file name, *not* the name you used inside the Lua file).

However since you didn't specify any settings, `transconv` will use the default ones – which result in no changes to the input at all. To get your scheme to do something useful, you have to override these settings. This can be done by adding member variables and/or functions to your scheme.

5.1.1 Default Member Variables of Schemes

Member variable settings don't change the conversion process itself but merely provide resources which the scheme uses during this process. For minor changes, you usually only need to set a member variable and can leave the algorithm itself alone. The following variables are available to you by default (do not forget to add a comma after each member or Lua will get confused!):

`raw` This variable tells your scheme which raw (input) scheme to use. This should pretty much always be set to the following:

```
raw = require(transconv.path_of(...)..".raw"),
```

This will cause Lua to load the scheme from the `raw.lua` file in the same folder.

`rep_strings` This variable is probably going to be your best friend because it is what tells your scheme to replace certain letter sequences with others. It contains a comma-separated list of string pairs surrounded by curly braces. Each string (letter sequence) should be surrounded with (double or single) quotes and the two items of each pair should be separated with a comma also like so:

```
rep_strings = {
  {"c", "k"}, {"ts", "ch"},
},
```

During conversion, your scheme will look at each of the pairs, find all instances of the first item in your input and replace it with the second item. For example, the above settings will cause it to replace every “c” with “k” and every “”ts”” with “”ch”” (note that by default, the search is case-insensitive, so “C”, “Ts”, “TS” and “tS” will also be replaced).

Be aware that the replacements are executed in the order in which you defined them, which means earlier rules can feed into later ones if you're not careful. For example, if we swap the pairs above around:

```
rep_strings = {
 {"ts", "ch"}, {"c", "k"},
},
```

literal character	escape sequence
\	\\
.	%.
-	%-
*	%*
%	%%
(	%(
)	%)
+	%+
?	%?
^	%^
[	%[
$	%$

Table 1: Lua string escape sequences

Then the first rule will first replace every “ts” with “ch”, but then the second rule will replace the “c” with “k” and you end up with the possibly unexpected “kh”. So if you get surprising replacements, have a look at the order of rules and check if any might be feeding into later ones.

The second thing you have to pay attention to is that certain characters have a special meaning in Lua, so in order for your scheme to use their literal values, you have to escape them. You can find the full list in Table 1.

So if, for example, you want to replace all instances of “aa” with “â”, the correct rule would be:

```
{"aa", "\\%^{a}"}
```

sb_sep Use this variable to define a separator if your output scheme requires one to be inserted between different syllables (for example the Wade-Giles scheme for Mandarin Chinese has syllables separated with a hyphen). The separator is not inserted before a space or special characters. The default setting is an empty string.

tone_markers For tonal languages which mark the tones with diacritics, list all tones as integer keys with the macro name for the marker as a value (without the leading backslash). Tones which do not have such a marker should be marked as `false`. For example the correct setting for Hanyu Pinyin would be:[1]

[1] The elements 0 and 5 are both set false so the user can use either integer to mark the neutral tone. The single quote for the second tone macro name has to be escaped because Lua would otherwise take it as a special character (cf. Table 1).

```
tone_markers = {
  [0] = false, [1] = "=", [2] = "\'", [3] = "v", [4] = "`", [5] = false,
},
```

The converter will then end up replacing the input `a1` with `\={a}`, `a2` with `\'{a}` etc.

In case your tone numbers are consecutive integers starting with 1, you can also simply list the marker strings without explicitly stating the index. So if we disallow using `0` for the neutral tone, the above could also be simplified to:

```
tone_markers = { "=", "\'", "v", "`", false},
```

second_rep_strings For tonal languages it may occasionally be necessary to do a second round of string replacement after it is already decided where the tone marker should go. Use this variable for this purpose. It works the same way as `rep_strings`.

no_tones For tonal languages, this variable can be set to either `true` or `false`. If `true`, your scheme will simply delete all tones from your output. This allows you to first write your document with tones but then turn them off if your publisher wants tones to be omitted.

5.1.2 Adding and Overriding Methods

If the variable settings are not sufficient to produce the intended result, you can override the default functions of your scheme or add your own ones to supplement them. You may also choose to remove unneeded default functions to get a slight boost in performance if you find conversion is too slow. However, this requires at least a basic understanding of Lua to write the new functions. I will therefore assume for this section that you know how to define a function and add it to a table.

Overriding Default Methods

Any new transcription scheme will provide you with the following default methods which you can override by simply defining your own custom version and adding it to your scheme table.

convert(self, input) This is the central function of your scheme. It must always be present because this is the function which `transconv` will call when you use `\tolang` or a similar command in your document.

By default it will:

1. split the input into syllables by calling the associated `raw` scheme's `split_sbs` method,

2. check each syllable if it is a valid syllable in the associated `raw` scheme by calling its `is_valid_sb` method. If not, the syllable is funneled directly into the output without any further processing.[2]

3. For valid syllables, it checks if there is already a cached conversion result. If not, it will call the `to_target_scheme` function which handles the actual conversion, store the result in the cache and then funnel it to the output.

4. After all syllables have been processed, they are joined back together using the `join_sbs` method and the result is returned.

to_target_scheme(self, syllable) The most basic conversion function. It will:

1. call the associated `raw` scheme's `get_sb_and_tone` method to separate potential tone digits at the end and store them separately,

2. call the `do_str_rep` method using the `rep_strings` member variable to execute string replacements,

3. call the `place_tone_digit` method to identify the correct letter which should carry the marker and insert the tone digit after it,

4. call the `do_str_rep` method again, but this time using the `second_rep_strings` member variable for secondary replacements,

5. check if the `no_tones` variable is set to `true`. If so, it simply deletes any digits from the string. Otherwise, it calls `add_tone_markers` to replace the digits with the correct tone markers,

6. return the end result.

do_str_rep(self, syllable, list_of_replacements) This function will perform the actual string replacements according to the provided list. It assumes that the list is of the same form as `rep_strings` above, i. e. a table of tables, where each of the inner tables contains exactly two strings (original and replacement). It will then loop over the outer table and for each member table:

1. convert both the input syllable and the search string to lower so case is ignored,

2. search the lower-case syllable for instances of the search string. If it finds any, it then:

[2] This allows you to use foreign words in the input.

a) checks the case of the first letter in the match for case. If the former is lowercase, it will assume all lower case. If it is upper and the following one is lower, it will assume title case. If both are upper, it will assume all upper case.

b) performs the string replacement in the appropriate casing,

After it has finished the loop it returns the end result.

place_tone_digit(self, syllable, tone) For tonal languages, this method is responsible for placing the tone digit back into the syllable at the correct position. If some other letter will carry the tonal information (typically using a diacritic, but possibly with other means, e. g. reduplication, replacement with another letter etc), this method should identify that letter and place the raw digit behind it. Note that this function only handles placement; the conversion into the correct output form will be handled later by the `add_tone_marker` function.

By default, this function simply adds the tone digit back to the end of the string, so if your scheme requires a different behaviour, you will have to override this method.

add_tone_marker(self, syllable) This function looks for digits in the input syllable. If it finds one, it looks up in the `tone_markers` variable to find the name of the replacement macro. It then wraps the preceding letter in that macro and deletes the digit. For example, if the `tone_markers` variable contains the value `"="` at index `1` and your input string is `a1ng`, then this function will return `\={a}ng`.

Note that this function should only be used for tones marked with diacritics. If the tone is marked in another way (e. g. reduplication of a letter), add corresponding replacement rules in the `second_rep_strings` instead.

Adding Your Own Methods

To add your own method to the conversion process, you have to take two steps:

1. Implement your Method and add it to your scheme table. If it needs access to any other member variables or methods, make sure to pass a reference to your table as the first argument (either by using Lua's colon syntax or as an explicit argument).

2. Override one of the default methods and have it call your custom method at the appropriate step with the appropriate arguments.

5.2 Adding a New Language

To add a new language for which transcription schemes can be implemented, first add a new folder where `transconv` can find it. The folder name will be the language name you will have to use when loading schemes from your document later, so make sure you choose something unique and easy to memorise. The default languages use the ISO 693-3 abbreviations and it is strongly suggested you stick to the same convention. If you need to specify a certain subgrouping within a bigger language variety for which no ISO abbreviation has been coined, specify the location after a dash, for example `jpn-kyoto` for Kyōtō Japanese.

As a second step, you will have to choose a raw input scheme. This scheme should fulfil the following criteria:

- It must contain enough information to convert to any intended target scheme. More specifically, if any feature is reflected even in just one possible target scheme, it must be reflected in the raw scheme also, otherwise conversion to that target scheme will be incorrect. For example, the Japanese Hiragana characters じ and ぢ are pronounced exactly the same (ji) and most transcription schemes spell them the same as well. However a select few – most notable the Nihon-shiki scheme – do reflect the difference in Kana spelling. Therefore, the raw scheme has to make the distinction as well to allow for accurate conversion to those schemes.[3]

- It is strongly suggested that the raw scheme should not make use of non-ascii characters. The reason for this is that Lua makes use of your computer's locale for certain aspects of string handling. As a result, if you use non-ascii characters, the code may or may not work as you expect it on your machine. But even if it does work for you, it might not do so on a machine in a different locale.

Once you have decided on your raw scheme, add a file called `raw.lua` to your language folder.[4] Document your raw scheme in a comment section at the top.[5]

Below that, add a table for the raw scheme and return it:

```
local raw = Raw:new{
}

return raw
```

Then populate your table with settings and methods. The following are provided by default:

[3] In this case, I decided to follow Nihon-shiki and spell them according to their original phonetic series: `zi` for じ and `di` for ぢ.

[4] You don't have to use this name but I suggest you follow the convention. If you don't, any authors of new target schemes will have to use your different name when importing the raw scheme to their file, which could confuse especially unexperienced Lua users.

[5] If you are using an existing scheme or a slightly modified version of it, you can make this very brief by referencing it, e. g.: "Pinyin, just with tone numbers at the end of each syllable instead of diacritics".

5.3 Default Raw Scheme Member Variables

cutting_markers A list of strings which can be used to define borders between syllables when splitting up an input strings using the `split_sbs` method. If not empty, your raw scheme will scan the input string and make a cut whenever it finds one of the strings in this list. By default, the list is empty.

5.4 Default Raw Scheme Methods

get_sb_and_tone(self, input) Expects a single syllable with information on the tone in it. Returns the raw syllable (without tone information) and an integer representing the tone. By default it will always simply return the input string and 0. If your language is non-tonal, simply ignore this method.

is_valid_sb(self, input) Use this method to test if the passed string is a valid syllable in your input string. If it returns `false` for a given string, `transconv` will simply not make any changes to it. By default it always returns `true`.

split_sbs(self, input) Used to split the input into syllables. If your language does not require this (like Japanese for instance), you can override this method to simply return a list with the input string as its only member.

By default it will:

1. Check if the `cutting_markers` variable is empty. If it is, it will simply make a cut before each non-word character (special characters or whitespace). Otherwise it will search the input for any strings in the list and make a cut before any matches.

2. Either way, it puts all syllables in a list and returns it.

6 Package Code

6.1 The `transconv.sty` file

6.2 The Lua code