The `transconv` Package

Sievert von Stülpnagel

March 21, 2020

Contents

List of Tables

1 Introduction

As a linguist who primarily works on East Asian languages, I frequently have to render languages in the Latin script which do not usually use it. While for some contexts, the IPA is certainly the best way to go, there are many situations where it is either unnecessarily unwieldy or omits certain information (e.g. historical one) which may be crucial to the discussion. Therefore, it is often more practical to use a transcription scheme to transcribe the pronunciation (and sometimes certain aspects of the orthography) into (usually) the Latin script. For example, we may use Hanyu Pinyin to transcribe Standard Chinese.

However, these schemes almost always pose at least one of these two problems on a LaTeX user:

1. They use non-ASCII characters which may be annoying to input, e.g. diacritics, super- or subscripts etc. Obviously can do this manually with LaTeX commands but it can be extremely obnoxious if you have to use those macros very frequently.

2. There are often multiple competing transcription schemes for each language and the author may not always be free in their choice of scheme. So I might write one article using scheme x, but I end up publishing it in a different paper than anticipated and that paper requires me to use scheme y. So I would essentially have to manually find every single instance of me using x and change it to y. Obviously this is both annoying and highly error-prone.

This package hopes to solve these problems by a) letting the user write in a transcription scheme which is easier to input and have the package handle the conversion to the actual output scheme with diacritics etc., and b) abstracting the actual scheme itself away from the text, so the user can switch schemes by simply changing an option, not every instance of them using the scheme.

For instance, if I had to transcribe the Southern Min word for "fifteen" in the Tâi-lô scheme (tsa̍p-gōo), I would normally have to write:

```
ts\textvbaraccent{a}p-g\={o}o
```

But `transconv` allows me to simply use numbers instead of the tone diacritics and write:

```
\tonan{tsap8-goo7}
```

What's more, if I suddenly find myself having to use the POJ transcription scheme instead, all I have to do is change a package option and recompile, and `transconv` will output the correct POJ version instead: cha̍p-gō. Or maybe I'm required to use Bbánlám pìngyīm? No problem: zápggoô.

Also as you can see, `transconv` has no problem with you switching back and forth between multiple different schemes, either, if that is what you need.

1.1 About `transconv`

1.2 License

2 Installation

In order to be able to use `transconv`, you need to copy the `transconv.sty` file as well as the `transconv` directory (found inside the `lua` folder) to a place where LuaTEX can find them. The suggested location for the `sty` file is within `tex/latex/local/` in your local `texmf/` directory (probably found within your `home` directory).

The `transconv` folder can be placed in any directory in your kpathsea path. You can check that path with the console command `kpsewhich --show-path=lua`. The suggested location is within `scripts/` inside your local `texmf/` directory. If for some reason, you cannot get LuaTEX to find the module, you can consider using the luapackageloader package to manually modify the path.

3 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*⟩. It will also ensure that the `\to`⟨*lang*⟩ and `\`⟨*lang*⟩`font` commands exist by defining them the first time you define a scheme for a new language. Loading more schemes for the same language after that will not change that default; if you want to change it, use the `defaultscheme` option or the `\TransconvMakeDefaultScheme` command.

In order to successfully load a scheme, Transconv has to be able to find a file ⟨*lang*⟩/⟨*scheme*⟩.lua in its lua folder which returns the converter. The option 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 default scheme assignments override earlier ones.

`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 default scheme assignments override earlier ones.

4 Basic Interface

All commands in this section are robust and can therefore be used in expansion-only contexts (e. g. headings or footnotes) without danger.

However, the list of commands you can use within the conversion argument is unfortunately very limited. This is because such commands would be expanded before the string is sent to the converter, which can easily result in an error.

As a rule of thumb, if your command is a simple macro which expand to nothing but text like in the following example, it should be fine:

```
\newcommand\mystring{Zhe4 mei2you3 wen4ti2.}
\topinyin{\mystring}
```

But commands which change some settings (e. g. the font) will likely cause an error. For example both of the following will not compile:

```
\topinyin{\emph{Zhe3} wu2fa3 bian1yi4.}
\topinyin{{\itshape Zhe3} ye3 bu4 xing2.}
```

4.1 Formatted Conversion Commands

`\tolang{`⟨*text*⟩`}`

where `lang` is a language declared via `\TransconvUseScheme` or the package option `scheme`. This command will convert ⟨*text*⟩ to the default scheme for `lang` (either the one defined by `defaultscheme` or else the first 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.

`\toscheme{`⟨*text*⟩`}`

where `scheme` is a scheme declared via `\TransconvUseScheme` or the package 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 you are discussing differences between different schemes). 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.

4.2 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 first 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.

\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.

4.3 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.

5 Configuration

\TransconvUseScheme{⟨*schemes*⟩}

Sets one or more new scheme(s) up for later usage. Equivalent to using the `scheme` package option. As with the option, each scheme should be specified as ⟨*lang*⟩ . ⟨*scheme*⟩ and multiple schemes can be set up at once by stringing them together with commas.

\TransconvMakeDefaultScheme{⟨*schemes*⟩}

Sets up an existing scheme as the default scheme for a language used in `\tolang` or `\langconvert`. Equivalent to using the `defaultscheme` package option. As with the option, each scheme shoult be specified as ⟨*lang*⟩ . ⟨*scheme*⟩ and multiple default schemes can be set up at once by stringing them together with commas. The scheme(s) has (have) to already be initialised with either the `scheme` option or the `\TransconvUseScheme` command, otherwise this command will have no effect.

6 Custom Schemes and Languages

6.1 Adding a New Transliteration Scheme for an Existing Language

Adding a new transliteration scheme involves a little bit of Lua programming, though I tried to make it as painless as possible.

As a first step, navigate to the directory where the Transconv lua files are found (the exact location depends on where you installed it, but it should be called transconv and contain a file named init.lua.

Once in this directory 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
}
```

At the end of the file, return the scheme:

```
return MyScheme
```

You can 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.

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.

6.1.1 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"},
},
```

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.

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

Table 1: Lua string escape sequences

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]

```
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`.

[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).

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.

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

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:
   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.

6.2 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]

[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.

- 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:

6.3 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.

6.4 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`.

[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".

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.

7 Package Code

7.1 The `transconv.sty` file

```
\NeedsTeXFormat{LaTeX2e}

%%%%%%%%%%%%%%%%%%%%%%%
% required packages %
%%%%%%%%%%%%%%%%%%%%%%%
\RequirePackage{l3keys2e,xparse}
\ProvidesExplPackage{transconv}{2019/02/21}{1.0}{Transcription conversion
package}

\ExplSyntaxOn

%%%%%%%%%%%%%%%%%%%%%%%%
% set up environment %
%%%%%%%%%%%%%%%%%%%%%%%%
% Requires the package transconv to be in LuaTeX's search path for packages.
\directlua{transconv = require "transconv"}

%%%%%%%%%%%%%%
% functions %
%%%%%%%%%%%%%%

% Document command definitions depending on lang name
\cs_new:Npn \__transconv_define_lang_convert_command:n #1
{
  % e.g. "\cmnconvert"
  % use Npx version (expanding the argument as soon as the function is used)
  % because otherwise multiple langs would overwrite each other
  % Use protected to ensure it behaves correctly in fragile environments
  \expandafter\cs_set_protected:Npx\cs:w #1convert\cs_end: ##1
  {%
```

```
    % use the first entry in default schemes list for the specified language
    % (lua indexes start at 1)
    \noexpand\directlua{tex.sprint(transconv.default_schemes["#1"][1]:convert([[##1]]))}%
  }
}

% Document command definitions depending on scheme name
\cs_new:Npn \__transconv_define_scheme_convert_command:n #1
{
  % e.g. "\pinyinconvert"
  \expandafter\cs_set_protected:Npx\cs:w #1convert\cs_end: ##1
  {%
    \noexpand\directlua{tex.sprint(transconv.schemes["#1"]:convert([[##1]]))}%
  }
}

\cs_new:Npn \__transconv_define_document_command:n #1
{
  % e.g. "\topinyin"
  \expandafter\cs_set_protected:Npx\cs:w to#1\cs_end: ##1
  {%
    \cs:w #1font\cs_end:\cs:w #1convert\cs_end:{##1}
  }
}

% font switch definition function
\cs_new:Npn \__transconv_define_font_switch:nn #1#2
{
  % e.g. "\pinyinfont"
  \expandafter\DeclareDocumentCommand\cs:w #1font\cs_end:{}{#2}
}

% TODO: define environment?

% defining a new scheme for a language
\cs_new:Npn \__transconv_usescheme:n #1
{
    % retrieve language and scheme names
    \regex_extract_all:nnN {[\w-]+} {#1} \l__transconv_langscheme_seq
    \seq_pop_left:NN \l__transconv_langscheme_seq \l__transconv_lang_tl
    \seq_pop_left:NN \l__transconv_langscheme_seq \l__transconv_scheme_tl

    % set up the converter.
    % For example importing "cmn.pinyin" would result in the following Lua command:
    % transconv.schemes["pinyin"] = transconv.new_converter("cmn", "pinyin")
    \directlua{%
      transconv:new_converter(%
```

```
      "\tl_use:N \l__transconv_lang_tl",%
      "\tl_use:N \l__transconv_scheme_tl"%
    )%
  }

  % define font switches, but only if if they don't exist already in case the
  % user is redefining an existing scheme but wants to keep font settings
  \if_cs_exist:w \tl_use:N \l__transconv_scheme_tl font\cs_end:
  \else:
    \__transconv_define_font_switch:nn {\tl_use:N \l__transconv_scheme_tl}{}
  \fi:
  \if_cs_exist:w \tl_use:N \l__transconv_lang_tl font\cs_end:
  \else:
    \__transconv_define_font_switch:nn {\tl_use:N \l__transconv_lang_tl} {\itshape}
  \fi:

  % (re)define \langconvert and \schemeconvert commands
  \__transconv_define_scheme_convert_command:n {\tl_use:N \l__transconv_scheme_tl}
  \__transconv_define_lang_convert_command:n {\tl_use:N \l__transconv_lang_tl}

  % (re)define \tolang and \toscheme commands
  \__transconv_define_document_command:n {\tl_use:N \l__transconv_scheme_tl}
  \__transconv_define_document_command:n {\tl_use:N \l__transconv_lang_tl}

  % TODO: (re)define environment?
}

\cs_new:Npn \__transconv_make_default:n #1
{
  % retrieve language and scheme names
  \regex_extract_all:nnN {[\w-]+} {#1} \l__transconv_langscheme_seq
  \seq_pop_left:NN \l__transconv_langscheme_seq \l__transconv_lang_tl
  \seq_pop_left:NN \l__transconv_langscheme_seq \l__transconv_scheme_tl

  \directlua{%
    transconv:make_default_scheme(%
      "\tl_use:N \l__transconv_lang_tl",
      "\tl_use:N \l__transconv_scheme_tl"%
    )%
  }
}

%%%%%%%%%%%%%%%%%%%%%%%
% Settings commands %
%%%%%%%%%%%%%%%%%%%%%%%

\NewDocumentCommand\TransconvUseScheme{m}
```

```
{%
  % call usescheme for every item
  \clist_map_function:nN {#1} \__transconv_usescheme:n
}

\NewDocumentCommand\TransconvMakeDefaultScheme{m}
{%
  % call make_default for every item
  \clist_map_function:nN {#1} \__transconv_make_default:n
}

%%%%%%%%%%%%%%%%%%%%
% Package options %
%%%%%%%%%%%%%%%%%%%%
% defines legal options and how to process them
\keys_define:nn { transconv }
{
  scheme .code:n         = \TransconvUseScheme{#1},
  defaultscheme .code:n  = \TransconvMakeDefaultScheme{#1}
}

\ProcessKeysOptions{ transconv }

\ExplSyntaxOff
```

7.2 The Lua code

7.2.1 The Main File (`transconv/init.lua`)

```lua
main_dir = ...
if main_dir ~= "init" then
    main_dir = main_dir.."."
else
    main_dir = ""
end

Raw = require(main_dir.."raw")

schemes = {}
default_schemes = {}

local function path_of(path)
    --[[
        Returns the parent directory of a (dot-separated) path. E.g. "lib.foo"
        for the input "lib.foo.dir"
    --]]
    return path:match("^(.-)%.[^%.]+$")
```

```
end

local function make_default_scheme(self, lang, scheme)
    --[[
        Moves scheme to the front of lang's list in default_languages. Returns
        integer code to reflect the result:
            1: scheme found and made default for lang
            0: scheme already default for lang (no changes done)
           -1: scheme currently does not exist for lang (error)
           -2: there is no scheme set up for lang at all (error)
    --]]

    local function move_element_to_front(list, old_i)
        local new_first = list[old_i]

        -- take all elements from the beginning (1) up to the one before old_i
        -- and move them to the range starting at index 2 (and thus ending on
        -- old_i)
        table.move(list, 1, old_i-1, 2)

        list[1] = new_first
        return list
    end

    -- return error code if no scheme is set up for lang
    if self.default_schemes[lang] == nil then
        return -2
    end

    -- find scheme in default_schemes list for lang and get its index
    local index = 0
    for i,v in ipairs(self.default_schemes[lang]) do
        if v.name == lang.."."..scheme then
            index = i
            break
        end
    end

    -- handle depending on if the scheme was found and at what index
    if index == 0 then
        -- index 0 means scheme was not found because Lua starts indexing at 1
        return -1
    elseif index == 1 then
        return 0
    else
        self.default_schemes[lang] = move_element_to_front(self.default_schemes[lang], ind
    end
```

```
end

local function new_converter(self, first, second, third)
    Converter = require(main_dir.."converter")
    Raw = require(main_dir.."raw")

    -- process options table if given (either in second or third argument)
    if type(second) == "table" then
        options = second
        second = nil
    else
        options = third or {}
    end

    -- if first input contains a dot surrounded by other characters, interpret
    -- it as a directory separator between language and scheme (second argument
    -- is discarded)
    local lang, scheme = first:match("^([^%.]-)%.(.-)$")

    -- if match failed, use first as directory and second as scheme names
    if not scheme then
        lang = first
        if type(second) == "string" then scheme = second else scheme = "" end
        options = third or {}
    end

    -- add main_directory information if invoked from outside
    if main_dir ~= "init" then
        lang_module = main_dir..lang
    else
        lang_module = lang
    end

    local c = require(string.format("%s.%s", lang_module, scheme)):new(options)

    self.schemes[scheme] = c

    -- ensure that default_schemes has an entry for lang
    self.default_schemes[lang] = self.default_schemes[lang] or {}
    -- check if scheme is already in default schemes and replace entry if it is
    local found = false
    for i,v in ipairs(self.default_schemes[lang]) do
        if v.name == scheme then
            self.default_schemes[lang][i] = c
            found = true
            break
        end
```

```
    end
    -- make new entry if it isn't
    if not found then table.insert(self.default_schemes[lang], c) end

    return c
end

transconv = {
    main_dir = main_dir,
    -- functions
    path_of = path_of,
    make_default_scheme = make_default_scheme,
    new_converter = new_converter,
    -- uselangs = uselangs,
    schemes = schemes,
    default_schemes = default_schemes,
}

return transconv
```

7.2.2 The Converter Prototype (`transconv/converter.lua`)

```
-- factory function
local new = function(self, conv)
    -- TODO: ensure proper encapsulation
    conv = conv or {} -- create converter object if not specified
    setmetatable(conv, self)
    conv.cache = {} -- necessary to prevent converters from trying to share their cache
    self.__index = self -- make this the prototype for new converters
    return conv
end

local add_tone_marker = function(self, instring)
    --[[
        Receives a string and returns it with tone digits replaced with the
        correct diacritics.
    --]]
    local t = 0 -- use while because Lua for loops starts at index 1 and we want to inclu
    while true do
        local marker = self.tone_markers[t]
        -- break on reaching the first index error (ignore 0 because people
        -- might not use it for a given scheme)
        if t > 0 and marker == nil then break end

        -- try to match the tone digit after a letter in the input string
        local needle = string.format('[%%w]%d', t)
        local match = string.match(instring, needle)
```

```
        -- ü has to be matched separately because it being 2 bytes long
        -- confuses the matching function if used together with the others
        match = match or string.match(instring, "ü"..t)
        -- if matched and tone is not unmarked
        if match and marker then
            -- get the letter which will carry the tone marker
            local carrier = match:sub(1, -2)

            -- for i and j, use dotless version instead to make way for the
            -- diacritic
            if carrier == "i" then
                carrier = "\\i"
            elseif carrier == "j" then
                carrier = "\\j"
            end

            -- wrap the letter in the appropriate LaTeX macro
            local rep = string.format("\\%s{%s}", marker, carrier)

            return instring:gsub(match, rep)
        -- if matched and unmarked
        elseif match and not marker then
            -- just delete the tone number
            return instring:gsub(tostring(t), "")
        end

        -- increase control variable for next iteration
        t = t + 1
    end

    return instring
end

local do_str_rep = function(self, instring, rep_dict)
    --[[
        Do the appropriate string replacements according to the passed
        replacement dictionary. E.g. Tâi-lô "ts" becoming "ch" in POJ.

        TODO: Can this be optimised so it doesn't have to loop over the
        whole thing?
    --]]

    for _, rep_pair in pairs(rep_dict) do
        local lower_input = instring:lower()

        local orig = rep_pair[1]:lower()
        local rep = rep_pair[2]:lower()
```

```
        -- find starting index of match if there is one
        -- also capture groups (look behind and look ahead) if returned
        local st, en, groupi, groupii = lower_input:find(orig)
        -- put empty strings if nothing was captured
        local groupi = groupi or ""
        local groupii = groupii or ""

        -- match cases
        if st then
            -- update start and end indexes according to lengths of the groups
            st = st + groupi:len()
            en = en - groupii:len()

            -- use indexes to check original input string for the case of
            -- the first two letters
            local match_first = instring:sub(st,st)
            local match_second = instring:sub(st+1,st+1)

            -- if first letter is lower, assume it's all lower
            if match_first == match_first:lower() then
                rep = rep:lower()
            -- if it's upper and the second is lower, assume title case
            elseif match_second == match_second:lower() then
                rep = rep:sub(1,1):upper()..rep:sub(2)
            -- if both are upper, assume all upper case
            else
                rep = rep:upper()
            end

            -- escape special characters in match string before substitution
            local match = instring:sub(st, en):gsub("([^%w])", "%%%1")
            instring = instring:gsub(match, rep)
        end
    end

    return instring
end

local join_sbs = function(self, sbs)
    --[[
        Receives a list of syllables in target scheme and joins them together
        to a valid output string.
        TODO
    --]]

    --[[
```

```
        for each syllable do the following tests:
        a) Is it the first one?
        b) Does it start with a non-alphanumeric character ≠ the separator?
        If either of these is true, leave sb as it is. Otherwise add the
        separator to the front.
        Then add sb to the output table.
    --]]
    for i, sb in ipairs(sbs) do
        if not (i == 1
            or sb == "" -- LuaTeX for some reason sometimes adds "" at the end
            or sb:match("^%W")
            and sb:match("^%W") ~= self.sb_sep
            and sb:match("^%W") ~= "\\") then
                sbs[i] = self.sb_sep..sb
        end
    end

    return table.concat(sbs)
end

local place_tone_digit = function(self, sb, tone)
    --[[
        Receive a syllable with the tone number at the end and return it with
        the number moved behind the letter that is going to carry the
        diacritic.
    --]]
    return sb..tostring(tone)
end

local to_target_scheme = function(self, sb)
    --[[
        Takes a SINGLE SYLLABLE in raw scheme and converts it to target
        scheme of the converter.
    --]]

    -- separate tone and syllable proper
    sb, tone = self.raw:get_sb_and_tone(sb)

    sb = self.do_str_rep(self, sb, self.rep_strings)
    sb = self.place_tone_digit(self, sb, tone)
    -- secondary replacements that depend on the digit being in the
    -- right place already
    sb = self.do_str_rep(self, sb, self.second_rep_strings)

    -- convert numbers to diacritics if wanted, otherwise delete
    -- digits
    if not self.no_tones then
```

```
        sb = self.add_tone_marker(self, sb)
    end

    return sb
end

local convert = function(self, instring)
    --[[
        Use splitting function to split input strings into syllables. For
        each syllable, check cache if it has been converted before. If not,
        delegate computation to actual conversion function. Either way, join
        the outputs back together and return.
    --]]

    -- split input into sbs
    local sbs = self.raw:split_sbs(instring)

    local outsbs = {}

    for _, sb in ipairs(sbs) do
        -- Do replacements only on syllables that are valid in raw scheme
        if self.raw:is_valid_sb(sb) then
            if self.cache[sb] == nil then
                self.cache[sb] = self.to_target_scheme(self, sb)
            end
            table.insert(outsbs, self.cache[sb])
        else
            table.insert(outsbs, sb)

        end

    end

    return self.join_sbs(self, outsbs)
end

local __tostring = function(self)
    return self.name
end

local Converter = {
    -- converter prototype object
    name = "",
    raw = Raw, -- associate prototype raw scheme as default
    cache = {}, -- cache conversion results for better performance
    no_tones = false, -- set true to omit tone markers from output
    rep_strings = {},
```

```
    second_rep_strings = {}, -- for secondary replacement after number movement
    sb_sep = "",
    tone_markers = {
         -- list all tones as integer keys with their appropriate latex macro
         -- name (without the backslash). Unmarked tones should be set false.
        },

    -- functions
    new = new,
    add_tone_marker = add_tone_marker,
    convert = convert,
    do_str_rep = do_str_rep,
    join_sbs = join_sbs,
    place_tone_digit = place_tone_digit,
    to_target_scheme = to_target_scheme,
    __tostring = __tostring,
}

return Converter
```

7.2.3 The Raw Scheme Prototype (`transconv/raw.lua`)

```
local cutting_markers = {}

local function new(self, conv)
    -- TODO: ensure proper encapsulation
    conv = conv or {} -- create converter object if not specified
    setmetatable(conv, self)
    self.__index = self -- make this the prototype for new converters
    return conv
end

local function get_sb_and_tone(self, sb)
    --[[
        Determines the tone of the syllable passed into it. Returns the
        syllable without the tone digit and the tone as an int.
    --]]

    return sb, 0
end

local function is_valid_sb(self,sb)
    return true
end

local function split_sbs(self, instring)
    --[[
```

```
        Split input string into syllables.
    --]]
    local sbs = {}

    if next(self.cutting_markers) ~= nil then -- checks if table is empty
        -- TODO: make cut before each marker
    else
        -- TODO: how accurate is this pattern?
        for sb in instring:gmatch("%W*%w*") do

            -- Test if a) this raw scheme has a (sensible) syllable separator
            -- set, b) that separator is non-empty, and c) the current syllable
            -- starts with it. If so, remove it
            if type(self.sb_sep) == "string" and self.sb_sep:len() > 0 and
                sb:match("^"..self.sb_sep) then
                    sb = sb:sub(self.sb_sep:len() + 1)
            end

            table.insert(sbs, sb)
        end
    end

    return sbs
end

local Raw = {
    cutting_markers = {}, -- used for splitting

    --functions
    new = new,
    get_sb_and_tone = get_sb_and_tone,
    is_valid_sb = is_valid_sb,
    -- reorder = reorder,
    split_sbs = split_sbs,
}

return Raw
```

7.2.4 An Example Scheme: Hanyu Pinyin (`transconv/cmn/pinyin.lua`)

```
local function join_sbs(self, sbs)
    -- list of vowels before which an apostrophe needs to be inserted if they
    -- are not the first syllable in a word
    local vowels = {["a"]=true, ["e"]=true, ["o"]=true, ["\\={a}"]=true,
        ["\\={e}"]=true, ["\\={o}"]=true, ["\\\'{a}"]=true,
        ["\\\'{e}"]=true, ["\\\'{o}"]=true, ["\\v{a}"]=true,
        ["\\v{e}"]=true, ["\\v{o}"]=true, ["\\`{a}"]=true, ["\\`{e}"]=true,
```

```
        ["\\`{o}"]=true,
    }
    for i, sb in ipairs(sbs) do
        if i ~= 1 and vowels[sb:match("^%w")]
            or vowels[sb:match("^\\[v=\'`]{%w}")] then
            sbs[i] = "\'"..sb
        end
    end

    return table.concat(sbs, "")
end

local function place_tone_digit(self, sb, tone)
    -- For the digraph iu place digit behind the u (ui is caught by the vowel
    -- list later)
    if string.match(sb, "iu") then
        return sb:gsub("iu", string.format("iu%d", tone), 1)
    end

    -- check for letters in the order "a e o i u ng m" and place the digit
    -- behind the first one that is found
    local vowels = {
        "A", "a", "E", "e", "O", "o", "i", "u", "ü", "Ng", "NG", "M", "m",
    }
    for _, v in ipairs(vowels) do
        if string.match(sb, v) then
            -- put the number behind the first letter in the match
            local vhead = string.match(v, '^[aeioumnAEOMNü].*')
            local vtail = string.match(v, '^[aeioumnAEOMNü](.*)')
            local rep = string.format("%s%d%s", vhead, tone, vtail)
            return sb:gsub(v, rep, 1)
        end
    end

    -- return the result
    return sb
end

local Pinyin = Converter:new{
    name = "cmn.pinyin",
    raw = require(transconv.path_of(...)..".raw"),
    sb_sep = "",

    tone_markers = {
        -- unmarked tones should be set false
        [0] = false, [1] = "=", [2] = "\'", [3] = "v", [4] = "`", [5] = false,
    },
```

```
    rep_strings = {},

    -- functions
    join_sbs = join_sbs,
    place_tone_digit = place_tone_digit,
}

return Pinyin
```

7.2.5 An Example Raw Scheme: Standard Chinese (`transconv/cmn/raw.lua`)

```
--[[
    Use Hanyu Pinyin but with the tone as a number after the syllable. Both 0
    and 5 are accepted as markers for the neutral tone
--]]

local function get_sb_and_tone(self, sb)
    local last = string.sub(sb,-1)

    -- get tone number (including omitted unmarked tones)
    if tonumber(last) then -- returns nil if not a digit
        tone = tonumber(last)
        sb = string.sub(sb,1,-2) -- save syllable without tone number
    else
        tone = 5
    end

    return sb, tone
end

local function is_valid_sb(self, sb)
     -- return invalid if it contains a digit in another position besides last
    local notail = string.sub(sb, 1,-2)
    if sb:gsub("%d", "") ~= sb and notail:gsub("%d", "") ~= notail then
        return false
    end

    -- if all checks were negative:
    return true
end

local function split_sbs(self, instring)
    local sbs = {}

    -- TODO: exact enough?
    for sb in instring:gmatch("%W*[a-zA-Zü]*%d?") do
        table.insert(sbs, sb)
```

```
    end

    return sbs
end

local cmnraw = Raw:new{
    get_sb_and_tone = get_sb_and_tone,
    is_valid_sb = is_valid_sb,
    split_sbs = split_sbs,
}

return cmnraw
```