lua-list-hyphen — Per-language listing of hyphenated words for LuaLaTeX*

Alan J. Cain†

Released 2026-05-05

Abstract

This LuaLaTeX package writes each word that has been hyphenated across lines to a file, using a different file for each language, for subsequent external checking.

Contents

*This document describes v0.3.15, last revised 2026-05-05.

†a.j.cain (AT) gmail.com

1 Introduction

TEX's algorithm for finding points where a word can be hyphenated is good, but not perfect.[1] The present author writes in British English, where the valid division points can depend on both the pronunciation of a word and its internal structure (and hence its etymology). Currently, TEX's pattern-based approach produces *bio-lo-gic*, *bio-logy*, *bio-lo-gist*, rather than the standard *bio-logic*, *biol-ogy*, *biolo-gist*.[2] To deal with such cases, at least a substantially larger number of patterns would be required than are available at present. There are also various words where the valid division points in British English cannot be deduced from their spelling alone: for instance, the verbs *at-trib-ute*, *pre-sent*, *pro-duce*, *re-cord* have different division points from the orthographically identical nouns *at-tri-bute*, *pres-ent*, *prod-uce*, *rec-ord*. For another example, compare *cur-ric-ulum vitae* and *school cur-ricu-lum*.

Easy checking of the chosen hyphenations is desirable. With LuaTEX, it is possible to extract the hyphenated words. The LuaLATEX package lua-check-hyphen offers this facility. It checks hyphenated words against a whitelist, visually flags unknown hyphenations, and writes unknown hyphenations to a file. But it was first written in 2012, when LuaTEX was at an earlier stage of development, and so it has certain problems, such as with words containing ligatures. It also lacks multi-language support.

This LuaLATEX package, lua-list-hyphen, uses some ideas from lua-check-hyphen but was written from scratch to work with a modern LuaTEX. It simply writes hyphenated

[1] For a description of the algorithm and its limitations, see Knuth's account in Appendix H of *The TEXbook* (Addison-Wesley, 2021. ISBN: 978-0-201-13447-6)

[2] See the *New Oxford Spelling Dictionary*, which is the authority for word divisions in British English (Oxford University Press, 2005. ISBN: 978-0-19-860881-3).

words from each language to a separate file, so that they can be checked (manually or by an external program).

[The author has written a simple Python application `hyphenassist`[3] that checks the listed hyphenations against a dictionary of valid divisions and allows the user to quickly choose to add entries to the division dictionary, add hyphenation exceptions, or ignore particular hyphenations. He has used this program in conjunction with code incorporated into this package to check hyphenations in his own books.[4]]

Acknowledgements. The author thanks Keno Wehr for corrections and comments on the documentation.

Feature requests and bug reports The development code and issue tracker are hosted at Codeberg.[6]

2 Requirements

lua-list-hyphen requires

(1) LuaLaTeX,

(2) a recent LaTeX kernel with expl3 support (any kernel version since 2020-02-02 should suffice).

It does not depend on any other packages, but will interface with babel or polyglossia (if one of them is loaded) to determine language names.

3 Installation

To install lua-list-hyphen manually, run `luatex lua-list-hyphen.ins` and copy `lua-list-hyphen.sty` and `lua-list-hyphen.lua` to somewhere LuaLaTeX can find them.

4 Getting started

Simply load the package; the hyphenated words are by default written to the file `\jobname-`⟨*lang-id*⟩`.hyph`, without being sorted or having duplicates removed. The ⟨*lang-id*⟩ is either a LuaTeX numerical language ID, or a babel or polyglossia name of the language, if one of these packages is in use. The prefix `\jobname-` and the extension `.hyph` can be customized; see Section 5.

[3] URL: https://codeberg.org/ajcain/hyphenassist.

[4] In particular, *Form & Number: A History of Mathematical Beauty*. URL: https://archive.org/details/cain_formandnumber_ebook_large.

[5] URL: https://www.latex-project.org/lppl.txt

[6] URL: https://codeberg.org/ajcain/lua-list-hyphen

5 Package options

`verbose` The boolean option `verbose` controls how much information is written to the file about each hyphenated word. When `true`, for each hyphenated word, both the undivided original and the divided word are written out, as well as the page number on which the hyphenated word appears (or, more precisely, begins) and the undivided word in context (as specified by the `context` keys; see below). When `false`, only the hyphenated word is written. (*Default:* `false`)

`context` `context-before` `context-after` Integer options controlling how many words before (`context-before`) and after (`context-after`) the hyphenated word are written as context when `verbose=true`. The key `context` is simply a shortcut for setting `control-before` and `control-after` to the same value. (*Default:* `2`)

`unique` The option `unique` controls removal of duplicates from the list of hyphenated words written out. It can be be set to one of the following three values:

- **`none`:** Duplicate hyphenations are not removed.
- **`case`:** Hyphenations that are duplicate (case-sensitively) are removed. In this case, the hyphenations `geo-metry` and `Geo-metry` are considered to be distinct.
- **`nocase`:** Hyphenations that are duplicate (case-insensitively) are removed. In this case, the hyphenations `geo-metry` and `Geo-metry` are considered to be duplicates. The case of each listed hyphenation will be that of the first appearance of that hyphenation.

Note that removal of duplicates is unaffected by the page number or context that is written out when `verbose=true`. (*Default:* `none`)

`sort` The option `sort` controls sorting of the list of hyphenated words. It can be be set to one of the following three values:

- **`none`:** Hyphenations appear in the same order as the occur in the document, or, if duplicates are removed, in the order of first appearance in the document.
- **`case`:** Hyphenations are sorted case-sensitively. In this case, `Geo-metry` precedes `geo-meter`.
- **`nocase`:** Hyphenations are sorted case-insensitively. In this case, `geo-meter` precedes `Geo-metry`.

(*Default:* `none`)

`include-non-output` Boolean option determining whether hyphenated words that are never written to the page are listed. (For instance, a hyphenated word might occur in text that a package temporarily typesets into a box, measures, and then discards.) (*Default:* `false`)

The two options `prefix` and `extension` specify the files to which hyphenations are written. Between the prefix and the extension is either a LuaTeX numerical language ID, or a babel or polyglossia name of the language, if one of these packages is in use.

`prefix` The `prefix` is the part of the file name to which the list of hyphenated words is written, before the language ID. (*Default:* `\jobname-` (note the hyphen).)

`extension` The extension of the file (including the `.`) to which the list of hyphenated words for each language is written. (*Default:* `.hyph`)

`debug` The boolean option `debug` controls whether debugging information is written to the terminal. (*Default:* `false`)

6 Output format

Each output file begins with a header (each line of which begins with a 'comment' symbol `%`) that includes information about the language and the package options that were used. Each line of the remainder of the file describes one hyphenation.

When `verbose=false`, the line contains only the hyphenated word.

When `verbose=true`, the line contains the original undivided word, the hyphenated word, the page number where the hyphenated appears (or, to be precise, begins), and the context in which the hyphenated word appears. Each part of the output is padded so that they various lines align. The original and undivided words are separated by the ASCII 'arrow' `->`; the page number is prefixed by `p.`; and the context is surrounded by (straight) quotation marks `" "`. If the hyphenation was never written to the page, `p.`⟨*page*⟩ is replaced by `<none>`. (This can only happen with `include-non-output=true`.)

7 Usage notes

7.1 Languages

To determine the language of a word, lua-list-hyphen looks at what language is applied at the first possible hyphenation point, first considering the part of the word before it, then the part after it. In the (presumably rare) case of a 'mixed-language' word like 'near-Zugzwang' being specified (using, for example, babel) with `near-\foreignlanguage{german}{Zugzwang}`, it would be assigned to the language in which 'near-' is set.

Duplicates are removed within each language. If the same hyphenation occurs in two different languages, it will appear in both files, regardless of the value of `unique`.

7.2 Limitations

lua-list-hyphen uses LuaTEX's built-in Unicode functions for pattern matching and converting between upper and lower case, which are based on the `slnunicode` library. This library has not been updated for some time and is based on an out-of-date version of the Unicode standard. Thus there may be problems with languages added to Unicode more recently. Hyphenated words from such languages should still be listed, but may contain extraneous characters (such as adjacent punctuation) and may not be sorted correctly. Users may prefer to leave sorting and removal of duplicates to an external program that adheres to the current Unicode standard.

8 Implementation (LaTeX package)

```
⟨*package⟩
⟨@@=lualisthyphen⟩
```

8.1 Initial set-up

Package identification/version information.

```
\NeedsTeXFormat{LaTeX2e}[2020-02-02]
\ProvidesExplPackage{lua-list-hyphen}{2026-05-05}{0.3.15}
  {Listing hyphenated words for LuaLaTeX}
```

Check that LuaTeX is in use.

```
\sys_if_engine_luatex:F
  {
    \msg_new:nnn{ lua-list-hyphen }{ lualatex_required }
      { LuaLaTeX~required.~Package~loading~will~abort. }
    \msg_critical:nn{ lua-list-hyphen }{ lualatex_required }
  }
```

8.2 Options

\l__lualisthyphen_verbose_bool

Boolean option to indicate whether lists of hyphenations should be written verbosely.

```
\keys_define:nn { lua-list-hyphen }{
  verbose .bool_set:N = \l__lualisthyphen_verbose_bool,
}
```

(*End of definition for* `\l__lualisthyphen_verbose_bool`.)

\l__lualisthyphen_context_before_int
\l__lualisthyphen_context_after_int

Integer options to determine the number of words before and after a hyphenation shown as context in verbose output.

```
\keys_define:nn { lua-list-hyphen }{
  context-before .int_set:N = \l__lualisthyphen_context_before_int,
  context-before .initial:n = { 2 },
  context-after .int_set:N = \l__lualisthyphen_context_after_int,
  context-after .initial:n = { 2 },
  context .code:n = {
    \keys_set:nn{ lua-list-hyphen }{
      context-before=#1,
      context-after=#1,
    }
  },

}
```

(*End of definition for* `\l__lualisthyphen_context_before_int` *and* `\l__lualisthyphen_context_after_int`.)

\l__lualisthyphen_unique_int

Choice option to indicate whether lists of hyphenations should have duplicates removed, case-sensitively or case-insensitively.

```
\int_new:N\l__lualisthyphen_unique_int
\keys_define:nn { lua-list-hyphen }{
  unique .choices:nn = { none, case, nocase }{
    \int_set:Nn\l__lualisthyphen_unique_int{ \l_keys_choice_int - 1 }
  },
}
```

(*End of definition for* `\l__lualisthyphen_unique_int`.)

`\l__lualisthyphen_sort_int` Choice option to indicate whether lists of hyphenations should be sorted, case-sensitively or case-insensitively.

```
\int_new:N\l__lualisthyphen_sort_int
\keys_define:nn { lua-list-hyphen }{
  sort .choices:nn = { none, case, nocase }{
    \int_set:Nn\l__lualisthyphen_sort_int{ \l_keys_choice_int - 1 }
  },
}
```

(*End of definition for* `\l__lualisthyphen_sort_int`.)

`\l__lualisthyphen_include_non_output_bool` Boolean option to indicate whether lists of hyphenations should include those that are never output to the page.

```
\keys_define:nn { lua-list-hyphen }{
  include-non-output .bool_set:N = \l__lualisthyphen_include_non_output_bool,
}
```

(*End of definition for* `\l__lualisthyphen_include_non_output_bool`.)

`\l__lualisthyphen_file_prefix_str` String option for the prefix of files to which hyphenations are writtten.

```
\keys_define:nn { lua-list-hyphen }{
  prefix .str_set:N = \l__lualisthyphen_file_prefix_str,
  prefix .initial:e = { \c_sys_jobname_str- },
}
```

(*End of definition for* `\l__lualisthyphen_file_prefix_str`.)

`\l__lualisthyphen_file_extension_str` String option for the extension of files to which hyphenations are writtten.

```
\keys_define:nn { lua-list-hyphen }{
  extension .str_set:N = \l__lualisthyphen_file_extension_str,
  extension .initial:n = { .hyph },
}
```

(*End of definition for* `\l__lualisthyphen_file_extension_str`.)

`\l__lualisthyphen_debug_int` Option to specify whether debug information is written to the terminal. Not intended for end users.

```
\int_new:N\l__lualisthyphen_debug_int
\keys_define:nn { lua-list-hyphen }{
  debug .code:n = {\int_set_eq:NN\l__lualisthyphen_debug_int\c_one_int}
}
```

(*End of definition for* `\l__lualisthyphen_debug_int`.)

8.3 Processing package options

Process package options.

```
\ProcessKeyOptions [ lua-list-hyphen ]
```

Convert boolean options to integers (which can be accessed from Lua).

```
\int_new:N\l__lualisthyphen_verbose_int
\bool_if:NT\l__lualisthyphen_verbose_bool
  { \int_set_eq:NN\l__lualisthyphen_verbose_int\c_one_int }
\int_new:N\l__lualisthyphen_include_non_output_int
\bool_if:NT\l__lualisthyphen_include_non_output_bool
  { \int_set_eq:NN\l__lualisthyphen_include_non_output_int\c_one_int }
```

8.4 Lua backend

Load the Lua backend.

```
\lua_now:n{
  lualisthyphen = require('lua-list-hyphen')
}
```

8.5 Saving babel language names

At `enddocument/afterlastpage`, if possible save babel's language names. (polyglossia's names can be found directly from Lua.)

```
\hook_gput_code:nnn{ enddocument/afterlastpage }{ lua-list-hyphen } {
  \__lualisthyphen_babel_save_language_names:
}
```

__lualisthyphen_babel_save_language_names: If babel is in use, get language names from `\bbl@languages`.

```
\cs_new:Npn \__lualisthyphen_babel_save_language_names:
  {
    \cs_if_exist:NT\bbl@languages
      {
```

Iterate through `\bbl@languages` to get language names. Items stored in this macro are quadruples prefixed with `\bbl@elt`, so locally redefine this latter macro to an auxiliary function that passes language ID/name pairs to the Lua backend.

```
        \group_begin:
        \cs_set_eq:NN
          \bbl@elt
          \__lualisthyphen_babel_save_language_names_elt:nnnn
        \bbl@languages
        \group_end:
      }
  }
```

(*End of definition for* `\__lualisthyphen_babel_save_language_names:`.)

sthyphen_babel_save_language_names_elt:nnnn Auxiliary function that takes a quadruple stored in `\bbl@languages` and passes language ID/name pairs to the Lua backend.

```
\cs_new:Npn \__lualisthyphen_babel_save_language_names_elt:nnnn #1#2#3#4
  {
    \lua_now:n{
      lualisthyphen.babel_save_language_name(#2,'#1')
    }
  }
```

(*End of definition for* `\__lualisthyphen_babel_save_language_names_elt:nnnn`.)

8.6 Processing and writing hyphenation lists

At `enddocument/info`, process and output the hyphenations that have been found.

```
\hook_gput_code:nnn{ enddocument/info }{ lua-list-hyphen } {
  \__lualisthyphen_process_write_hyphenation_lists:ee
    {\str_use:N\l__lualisthyphen_file_prefix_str}
    {\str_use:N\l__lualisthyphen_file_extension_str}
}
```

sthyphen_process_write_hyphenation_lists:nn Sort the list of hyphenations into separate lists for each language, sort and deduplicate them as required, and write them to files with prefix given in the first parameter and suffix in the second.

```
\cs_new:Npn \__lualisthyphen_process_write_hyphenation_lists:nn #1#2
  {
    \lua_now:e{
      lualisthyphen.process_write_hyphenation_lists(
        '\luaescapestring{#1}',
        '\luaescapestring{#2}'
      )
    }
  }
\cs_generate_variant:Nn
  \__lualisthyphen_process_write_hyphenation_lists:nn
  { ee }
```

(*End of definition for* `\__lualisthyphen_process_write_hyphenation_lists:nn`.)

⟨/package⟩

9 Implementation (Lua backend)

⟨*lua⟩

9.1 Debugging function

debug Debugging function. Defined according to the package option `debug` to either do nothing or write debugging information.

```
local debug

if tex.count['l__lualisthyphen_debug_int'] == 0 then
  debug = function(s)
  end
else
  debug = function(s)
    print('lua-list-hyphen DEBUG: ' .. s)
  end
end
```

(*End of definition for* `debug`.)

9.2 Table key constants

Keys for tables containing hyphenatable/hyphenated word data.

```
local KEY_TYPE = 'type'
local KEY_WORD = 'word'
local KEY_LANG = 'lang'
local KEY_DIVISION = 'division'
local KEY_INDEX = 'index'
local KEY_CONTEXT = 'context'
local KEY_PAGE = 'page'
```

9.3 Segment type

Constants for types of segments found while scanning hlist before linebreaking.

```
local SEGMENT_WORD = 0
local SEGMENT_SPACE = 1
local SEGMENT_MATH = 2
```

9.4 Node ID and subtype constants

Define constants for the node IDs that need to be recognized.

```
local NODE_ID_HLIST = node.id('hlist')
local NODE_ID_DISC = node.id('disc')
local NODE_ID_GLUE = node.id('glue')
local NODE_ID_KERN = node.id('kern')
local NODE_ID_MARGIN_KERN = node.id('margin_kern')
local NODE_ID_GLYPH = node.id('glyph')
local NODE_ID_MATH = node.id('math')
```

Define constants for the kern node subtypes that have to be recognized. (There seems to be no automatic way to get the numerical value from the subtype text other than searching the `node.subtype(`⟨*node type*⟩`)` tables.)

```
local NODE_KERN_SUBTYPE_FONTKERN
local NODE_KERN_SUBTYPE_USERKERN
for k,v in pairs(node.subtypes('kern')) do
  if v == 'fontkern' then
    NODE_KERN_SUBTYPE_FONTKERN = k
  elseif v == 'userkern' then
    NODE_KERN_SUBTYPE_USERKERN = k
  end
end
```

Define constants for the math node subtypes.

```
local NODE_MATH_SUBTYPE_BEGIN
local NODE_MATH_SUBTYPE_END
for k,v in pairs(node.subtypes('math')) do
  if v == 'beginmath' then
    NODE_MATH_SUBTYPE_BEGIN = k
  elseif v == 'endmath' then
    NODE_MATH_SUBTYPE_END = k
  end
end
```

9.5 Output constants

Constants for output.

```
local STR_MATH = '[MATH]'
local STR_SPACE = ' '
local STR_SPACE_TWO = '  '
local STR_ARROW = ' -> '
local STR_PAGE_PREFIX = 'p.'
local STR_PAGE_NONE = '<none>'
local STR_QUOTE_OPEN = '"'
local STR_QUOTE_CLOSE = '"'
```

9.6 Utility functions

list_filter Take a list `t` and remove from it any elements for which the function `f` does not return true. (The index `j` is always the destination index to which a 'keep' element is moved.)[7]

```
158 local function list_filter(t, f)
159   local j = 1
160   local n = #t
161
162   for i=1,n do
163     if (f(t[i])) then
164       if (i ~= j) then
165         t[j] = t[i]
166         t[i] = nil
167       end
168       j = j + 1
169     else
170       t[i] = nil
171     end
172   end
173
174 end
```

(*End of definition for* `list_filter`.)

list_uniq Take a list `t` and remove from it adjacent elements for which the function `f` returns true. (The index `j` is always the last 'kept' element.)

```
175 local function list_uniq(t, f)
176   local j = 1
177   local n = #t
178
179   for i=2,n do
180     if (f(t[i],t[j])) then
181       t[i] = nil
182     else
183       j = i
184     end
185   end
186
187   list_filter(
188     t,
189     function(a) return a end
190   )
191 end
```

(*End of definition for* `list_uniq`.)

9.7 Getting text from nodes

Getting the components of the ligatures that have Unicode code points can be problematic, at least for some fonts, so define a lookup table for these cases.

```
192 local LIGATURE_TEXT = {
193   [0xfb00] = 'ff',
```

[7] Code adapted from https://stackoverflow.com/a/53038524.

```
  [0xfb01] = 'fi',
  [0xfb02] = 'fl',
  [0xfb03] = 'ffi',
  [0xfb04] = 'ffl',
}
```

Cache to save table lookups when extracting text.

```
local font_characters = {}
```

Extracting text from nodes uses two functions that call each other, so the names have to be defined ahead of time.

```
local get_node_text
local get_nodelist_text
```

get_node_text Return the text content of a glyph node (which might be a normal glyph, a ligature, etc.).

```
get_node_text = function(n)

  if n.id == NODE_ID_GLYPH then

    local ligature_text = LIGATURE_TEXT[n.char]
    if ligature_text ~= nil then
      return ligature_text
    elseif n.components then
      return get_nodelist_text(n.components)
    else
      -- See [https://tug.org/pipermail/luatex/2018-March/006786.html]
      local characters = font_characters[n.font]
      if not characters then
        characters = fonts.hashes.identifiers[n.font].characters
        font_characters[n.font] = characters
      end
      local c = characters[n.char]
      if c then
        return utf8.char(tonumber(c.tounicode,16))
      else
        return ''
      end
    end

  elseif n.id == NODE_ID_DISC then

    if n.replace then
      return get_nodelist_text(n.replace)
    else
      return ''
    end

  else
    return ''
  end

end
```

(*End of definition for* get_node_text.)

`get_nodelist_text` Return the text content of the glyph nodes in the list starting at `head` up to and including the node `last`, or up to the end of the list if `last` is not specified.

```
get_nodelist_text = function (head,last)

  local text = ''

  for item in node.traverse(head) do

    text = text .. get_node_text(item)

    if item == last then
      break
    end
  end

  return text

end
```

(*End of definition for* `get_nodelist_text`.)

`is_possible_word_node` Return boolean indicating if node `n` could be part of a word. Assume that `glyph`, `disc`, and `margin_kern` nodes could be part of a word, as could a `kern` node with subtype `fontkern`.

```
local function is_possible_word_node(n)

  return (
    n.id == NODE_ID_GLYPH
    or
    n.id == NODE_ID_DISC
    or
    (n.id == NODE_ID_KERN and n.subtype == NODE_KERN_SUBTYPE_FONTKERN)
    or
    n.id == NODE_ID_MARGIN_KERN
  )

end
```

(*End of definition for* `is_possible_word_node`.)

`is_possible_space_node` Return boolean indicating if node `n` could be part of a space. Assume that `glue` nodes could be part of a space, as could a `kern` node with subtype `userkern`.

```
local function is_possible_space_node(n)

  return (
    n.id == NODE_ID_GLUE
    or
    (n.id == NODE_ID_KERN and n.subtype == NODE_KERN_SUBTYPE_USERKERN)
  )

end
```

(*End of definition for* `is_possible_space_node`.)

9.8 String manipulation

trim_nonlettershyphens_both Remove characters other than letters and hyphens from both the start and end of a string.

```
local function trim_nonlettershyphens_both(s)

  return unicode.utf8.match(s,'^[^%a-]*(.-)[^%a-]*$')

end
```

(*End of definition for* `trim_nonlettershyphens_both`.)

trim_nonlettershyphens_start Remove characters other than letters and hyphens from the start of a string.

```
local function trim_nonlettershyphens_start(s)

  return unicode.utf8.match(s,'^[^%a-]*(.-)$')

end
```

(*End of definition for* `trim_nonlettershyphens_start`.)

trim_nonlettershyphens_end Remove characters other than letters and hyphens from the end of a string.

```
local function trim_nonlettershyphens_end(s)

  return unicode.utf8.match(s,'^(.-)[^%a-]*$')

end
```

(*End of definition for* `trim_nonlettershyphens_end`.)

rpad Return string `s` padded on the right with spaces to length `n`.

```
local function rpad(s,n)

   return s .. unicode.utf8.rep(STR_SPACE,n - unicode.utf8.len(s))

end
```

(*End of definition for* `rpad`.)

lpad Return string `s` padded on the left with spaces to length `n`.

```
local function lpad(s,n)

   return unicode.utf8.rep(STR_SPACE,n - unicode.utf8.len(s)) .. s

end
```

(*End of definition for* `lpad`.)

9.9 Pre-linebreak processing

Before each line has been broken, find all potential division points and store the words in which they occur, linking each potential break point to the corresponding word.

Declare a new attribute, which will be used to store in each disc node the index of the corresponding word in the table `hlist_segment_list`.

```
local hyphen_attr = luatexbase.new_attribute('hyphen_attr')
```

Table to hold segments (word/space/math) in the hlist that will be broken. This table will be cleared after the post-linebreak processing.

```
local hlist_segment_list = {}
```

get_first_glyph_lang Return the lang attribute of the first glyph in the the part of the list starting n that could be part of a word. (Currently unused; see the documentation of `get_disc_lang`.)

```
-- local function get_first_glyph_lang(n)

--   local item = n
--   while item and is_possible_word_node(item) do
--     if item.id == NODE_ID_GLYPH then
--       return item.lang
--     end
--     item = item.next
--   end

--   return nil

-- end
```

(*End of definition for* `get_first_glyph_lang`.)

get_disc_lang Try to find the language ID in force at a given disc node by looking at (1) the last glyph in the word before the disc node; (2) the first glyph in the word after the disc node. Default to language ID `0`.

(Looking at `replace`, `pre`, `post` is possible, but is unreliable and so disabled for the present. The author has encountered the situation where an explicit hyphen results in the hyphen characters in `replace` and `pre` having different language IDs. He has not had time to investigate how this arises from the interaction of babel/polyglossia and LuaLaTeX.)

```
local function get_disc_lang(n)

  -- lang = get_first_glyph_lang(n.replace)
  -- if lang then
  --   print(lang)
  --   return lang
  -- end

  -- lang = get_first_glyph_lang(n.pre)
  -- if lang then
  --   print(lang)
  --   return lang
  -- end

  -- lang = get_first_glyph_lang(n.post)
```

```
  -- if lang then
  --   return lang
  -- end

  local item
```

Before the disc node.

```
  item = n
  while item and is_possible_word_node(item) do
    if item.id == NODE_ID_GLYPH then
      return item.lang
    end
    item = item.prev
  end
```

After the disc node.

```
  item = n
  while item and is_possible_word_node(item) do
    if item.id == NODE_ID_GLYPH then
      return item.lang
    end
    item = item.next
  end

  return 0

end
```

(*End of definition for* `get_disc_lang`.)

`pre_linebreak` Extract segments (word/space/math) from the hlist at `hlist_head` and store appropriate data in `hlist_segment_list`. For spaces and math, this is just the existence of a segment. For a word, store its text and its language ID (as determined by `get_disc_lang`). Also, for each disc node, assign the index of the word in `hlist_segment_list` to its `hyphen_attr` attribute (declared above).

```
local function pre_linebreak(hlist_head,groupcode)

  local word_start_node = nil
  local segment_count = 0
  local lang = nil

  debug('Pre-linebreak processing start')

  local item = hlist_head
  while item do
```

If `item` is a math node (which must have subtype beginmath, unless something has changed the node list), skip the math and add `[MATH]` to `hlist_segment_list`.

```
    if item.id == NODE_ID_MATH then
      assert(item.subtype == NODE_MATH_SUBTYPE_BEGIN)
      while not (
        item.id == NODE_ID_MATH and item.subtype == NODE_MATH_SUBTYPE_END
      ) do
        item = item.next
      end
```

```
      item = item.next

      segment_count = segment_count + 1
      hlist_segment_list[segment_count] = {
        [KEY_TYPE] = SEGMENT_MATH
      }

      goto continue
    end
```

If `item` is a possible word node, read the whole word, setting the `hyphen_attr` of any disc nodes to `segment_count`, and adding the word to `hlist_segment_list`.

```
    if is_possible_word_node(item) then
      word_start_node = item
      segment_count = segment_count + 1
      while item and is_possible_word_node(item) do

```

When the first disc node is found, find the language of the word.

```
        if item.id == NODE_ID_DISC then
          if not lang then
            lang = get_disc_lang(item)
          end
          node.set_attribute(item,hyphen_attr,segment_count)
        end

        item = item.next
      end
```

`item` should be a node, because even after the last word node, the hlist will contain something. But just in case, check and find the last node using `node.tail` if necessary. This latter case should be very rare, so it is more efficient to recalculate here if necessary rather than having an extra assignment to store the previous node in the while loop.

```
      local word_end_node
      if item then
        word_end_node = item.prev
      else
        word_end_node = node.tail(word_start_node)
      end

      local word = get_nodelist_text(word_start_node,word_end_node)
      hlist_segment_list[segment_count] = {
        [KEY_TYPE] = SEGMENT_WORD,
        [KEY_WORD] = word,
        [KEY_LANG] = lang,
      }

      word_start_node = nil
      lang = nil

      goto continue
    end
```

If `item` is a node that could be part of a space, add a space to the segment list.

```
    if is_possible_space_node(item) then
```

```
      segment_count = segment_count + 1

      while item and is_possible_space_node(item) do
        item = item.next
      end

      hlist_segment_list[segment_count] = {
        [KEY_TYPE] = SEGMENT_SPACE
      }

      goto continue
    end
```

If `item` is anything else, just move on.

```
    item = item.next

    ::continue::
  end

  debug('Pre-linebreak processing finish')

  return true
end
```

(*End of definition for* `pre_linebreak`.)

9.10 Post-linebeak processing

After linebreaking, look for a discretionary node at the end of each line, which indicates that a word has been divided between the end of that line and the start of the next. Extract the two word-pieces from the lines and store them, together with the undivided word and its context in the appropriate language table. Also insert a whatsit to that will set the page number when the hyphenation is written out.

`get_used_disc` If at the tail of the hlist at `hlist_head` (which will be a line) there is a disc node not followed by a glyph node, return that disc node. Otherwise return `nil`. Only check up to `USED_DISC_SEARCH_LIMIT` nodes from the tail, because malformed node lists might have loops.

```
local USED_DISC_SEARCH_LIMIT = 20

local function get_used_disc(hlist_head)

  debug('  Looking for disc node at end of line')

  local item = node.tail(hlist_head)

  local count = 0
  while item and item.id ~= NODE_ID_GLYPH and count < USED_DISC_SEARCH_LIMIT do
    if item.id == NODE_ID_DISC then
      return item
    end
    item = item.prev
    count = count + 1
  end
```

```
452
453   return nil
454
455 end
```

(*End of definition for* `get_used_disc`.)

`get_disc_word_start` Return the node starting the word that includes a given disc node `n`, or `nil` if there is no such node.

```
456 local function get_disc_word_start(hlist_head,n)
457
458   local item = n
459
460   while item do
461     local prev = item.prev
462
463     if not (prev and is_possible_word_node(prev)) then
464       return item
465     end
466
467     item = prev
468   end
469
470   return nil
471 end
```

(*End of definition for* `get_disc_word_start`.)

`get_next_hlist` Return the next hlist in the list containing the given node `n`, or `nil` if there is no such hlist node.

```
472 local function get_next_hlist(n)
473
474   local item = n.next
475
476   while item do
477     if item.id == NODE_ID_HLIST then
478       return item
479     end
480     item = item.next
481   end
482
483   return nil
484
485 end
```

(*End of definition for* `get_next_hlist`.)

`get_line_first_word` Return the first word in the hlist at `hlist_head`, or `nil` if there is no such word.

```
486 local function get_line_first_word(hlist_head)
```

`word_start_node` is either `nil` or the (glyph) node that starts the word.

```
487   local word_start_node = nil
488
489   for item in node.traverse(hlist_head) do
490
```

```
    if item.id == NODE_ID_GLYPH then
      if not word_start_node then
        word_start_node = item
      end
    end

    if not is_possible_word_node(item) then
      if word_start_node then
        return get_nodelist_text(word_start_node,item.prev)
      end
    end

  end
```

It is possible that the word ends at the end of the hlist, so check if a word has been started.

```
  if word_start_node then
    return get_nodelist_text(word_start_node,node.tail(hlist_head))
  else
    return nil
  end
end
```

(*End of definition for* `get_line_first_word`.)

`get_context` Return a string assembled from the part of `hlist_segment_list` before or after `index` according to `incr` (which must be ± 1) up a maximum of `target_word_count` words.

```
local function get_context(index,incr,target_word_count)

  local result = ''
  local word_count = 0

  local i = index + incr
  while (
    i > 0 and i <= #hlist_segment_list and word_count < target_word_count
  ) do
    local t = hlist_segment_list[i]

    local item

    if t[KEY_TYPE] == SEGMENT_WORD then
      item = t[KEY_WORD]
      word_count = word_count + 1
    elseif t[KEY_TYPE] == SEGMENT_SPACE then
      item = STR_SPACE
    elseif t[KEY_TYPE] == SEGMENT_MATH then
      item = STR_MATH
    end

    if incr > 0 then
      result = result .. item
    else
      result = item .. result
    end

```

```
    i = i + incr
  end

  return result

end
```

(*End of definition for* `get_context`.)

Count and list for hyphenated words. Each entry in the list will be a table containing the original word, the hyphenation, the language, the index of the table in the list (which is needed later for stable sorting and sorting into the original order), and the context.

```
local hyphenation_list = {}
local hyphenation_count = 0
```

`check_line_hyphenation` Check whether there is a hyphenated word at the end of the given hlist; if so, save the word to `hyphenation_list`.

```
local function check_line_hyphenation(hlist)
```

First, is there a disc node not followed by a glyph node at the end of the list?

```
  local last_disc = get_used_disc(hlist.head)
  if not last_disc then
    debug('  No disc node found at end of line')
    return
  end
  debug('  Disc node found at end of line')
```

Get the undivided word and its language from `hlist_segment_list`.

```
  local hyphenation_index = node.has_attribute(last_disc,hyphen_attr)
  local t = hlist_segment_list[hyphenation_index]
  assert(t)
  assert(t[KEY_TYPE] == SEGMENT_WORD)
  local word = t[KEY_WORD]
  local lang = t[KEY_LANG]
```

`word` might be something other than a genuine word, such as an ISBN (with hyphen separators). So only proceed if it contains at least one letter.

```
  if not unicode.utf8.match(word,'%a') then
    debug('  Divided "word" contains no letters')
    return
  end
```

There should always be a next line, since there is a disc node at the end of `hlist`, but check anyway.

```
  local next_line = get_next_hlist(hlist)

  if not next_line then
    debug('  No following line found (which should not happen)')
    return
  end
```

For the pre-linebreak part of the word, get the word that ends the line, and trim any leading non-letters. This could leave an empty word; for example, if n-dimensional is broken at the hyphen, the word ending the line is just the hyphen. If an empty word is left, just use the non-trimmed result.

```
  local pre = get_nodelist_text(get_disc_word_start(hlist.head,last_disc))
```

```
  local pre_temp = trim_nonlettershyphens_start(pre)
  if pre_temp ~= '' then
    pre = pre_temp
  end
```

For the post-linebreak part, just get the word at the start of the next line, and trim and trailing non-letters.

```
  local post = trim_nonlettershyphens_end(get_line_first_word(next_line.head))
```

Compute the context and then trim any unwanted symbols from the word itself.

```
  local context =
    get_context(
      hyphenation_index,-1,tex.count['l__lualisthyphen_context_before_int']
    )
    .. word ..
    get_context(
      hyphenation_index,1,tex.count['l__lualisthyphen_context_after_int']
    )

  word = trim_nonlettershyphens_both(word)

  debug(
    '  Hyphenated word found: "' .. word .. '" -> "' .. pre .. '<>' .. post .. '"'
  )
```

Store everything (except the page number on which the hyphenated word appears, which is not yet known) in the hyphenation list.

```
  hyphenation_count = hyphenation_count + 1
  hyphenation_list[hyphenation_count] = {
    [KEY_LANG] = lang,
    [KEY_WORD] = word,
    [KEY_DIVISION] = pre .. post,
    [KEY_INDEX] = hyphenation_count,
    [KEY_CONTEXT] = context,
  }
```

Add a whatsit to record the page number when the page with the hyphenation is shipped out. This information also serves to distinguish hyphenations that are written to the page from those that occur in (e.g.) boxes that are discarded without being written to the page.

```
  late_lua_n = node.new('whatsit','late_lua')
  late_lua_n.data =
    'lualisthyphen.set_hyphenation_page(' .. hyphenation_count .. ',tex.count["c@page"])'

  node.insert_after(hlist.head,last_disc,late_lua_n)

end
```

(*End of definition for* `check_line_hyphenation`.)

`set_hyphenation_page` Set the page on which the hyphenation with the given index appears.

```
local function set_hyphenation_page(index,page)

  hyphenation_list[index][KEY_PAGE] = page

end
```

(*End of definition for* `set_hyphenation_page`.)

`post_linebreak` For every line in the vlist at `vlist_head`, check whether there is a hyphenated word at the end.

```
local function post_linebreak(vlist_head,groupcode)

  debug('Post-linebreak processing start')

  local line_no = 0

  for item in node.traverse(vlist_head) do

    if item.id == NODE_ID_HLIST then
      line_no = line_no + 1
      debug('  Line no.' .. line_no)
      check_line_hyphenation(item)
    end

  end

  hlist_segment_list = {}

  debug('Post-linebreak processing end')

  return true

end
```

(*End of definition for* `post_linebreak`.)

9.11 Callbacks

Add `pre_linebreak` and `post_linebreak` to the relevant callbacks.

```
local LUA_LIST_HYPHEN_PRE_LINEBREAK = 'LUA_LIST_HYPHEN_PRE_LINEBREAK'
luatexbase.add_to_callback(
  'pre_linebreak_filter',
  pre_linebreak,
  LUA_LIST_HYPHEN_PRE_LINEBREAK
)

local LUA_LIST_HYPHEN_POST_LINEBREAK = 'LUA_LIST_HYPHEN_POST_LINEBREAK'
luatexbase.add_to_callback(
  'post_linebreak_filter',
  post_linebreak,
  LUA_LIST_HYPHEN_POST_LINEBREAK
)
```

9.12 Language settings

Table mapping language IDs to textual names.

```
local language_table = {}
```

Populating `language_table` is done differently for babel and polyglossia. If babel is in use, the LaTeX frontend iterates through `\bbl@languages` and calls `babel_save_language_name`. If polyglossia is in use, `language_table` is populated by `polyglossia_get_language_names`, which is called just before the hyphenation lists are written.

babel_save_language_name

Store the association of a language ID to babel's texual name, if no name has been assigned to that ID already.

```
local function babel_save_language_name(lang_id,name)

  if not language_table[lang_id] then
    language_table[lang_id] = name
  end

end
```

(*End of definition for* `babel_save_language_name`.)

polyglossia_get_language_names

If polyglossia has been loaded, use it to build the table mapping language IDs to textual names.

```
local function polyglossia_get_language_names()

  if not polyglossia then
    return
  end

  for name,language in pairs(polyglossia.newloader_loaded_languages) do
    language_table[lang.id(language)] = name
  end

end
```

(*End of definition for* `polyglossia_get_language_names`.)

9.13 Processing hyphenation lists

Before writing out hyphenation lists, remove duplicates and/or perform sorting, in accordance with the set options.

9.13.1 Comparisons and equality checks

equal_hyphenation_case_sensitive

Equality check for deduplicating the list of hyphenations case-sensitively.

```
local function equal_hyphenation_case_sensitive(a,b)
  return (
    a[KEY_WORD] == b[KEY_WORD]
    and
    a[KEY_DIVISION] == b[KEY_DIVISION]
  )
end
```

(*End of definition for* `equal_hyphenation_case_sensitive`.)

equal_hyphenation_case_insensitive Equality check for deduplicating the list of hyphenations case-insensitively.

```
local function equal_hyphenation_case_insensitive(a,b)
  return (
    unicode.utf8.lower(a[KEY_WORD]) == unicode.utf8.lower(b[KEY_WORD])
    and
    unicode.utf8.lower(a[KEY_DIVISION]) == unicode.utf8.lower(b[KEY_DIVISION])
  )
end
```

(*End of definition for* `equal_hyphenation_case_insensitive`.)

lessthan_hyphenation_case_sensitive Comparison for sorting the list of hyphenations case-sensitively.
The comparison of index keys ensures that the sorting is stable.

```
local function lessthan_hyphenation_case_sensitive(a,b)
  return (
    a[KEY_WORD] < b[KEY_WORD]
    or
    (
      a[KEY_WORD] == b[KEY_WORD]
      and
      a[KEY_DIVISION] < b[KEY_DIVISION]
    )
    or
    (
      a[KEY_WORD] == b[KEY_WORD]
      and
      a[KEY_DIVISION] == b[KEY_DIVISION]
      and
      a[KEY_INDEX] < b[KEY_INDEX]
    )
  )
end
```

(*End of definition for* `lessthan_hyphenation_case_sensitive`.)

lessthan_hyphenation_case_insensitive Comparison for sorting the list of hyphenations case-insensitively.
The comparison of index keys ensures that the sorting is stable.

```
local function lessthan_hyphenation_case_insensitive(a,b)
  return (
    unicode.utf8.lower(a[KEY_WORD]) < unicode.utf8.lower(b[KEY_WORD])
    or
    (
      unicode.utf8.lower(a[KEY_WORD]) == unicode.utf8.lower(b[KEY_WORD])
      and
      unicode.utf8.lower(a[KEY_DIVISION]) < unicode.utf8.lower(b[KEY_DIVISION])
    )
    or
    (
      unicode.utf8.lower(a[KEY_WORD]) == unicode.utf8.lower(b[KEY_WORD])
      and
      unicode.utf8.lower(a[KEY_DIVISION]) < unicode.utf8.lower(b[KEY_DIVISION])
      and
      a[KEY_INDEX] < b[KEY_INDEX]
    )
```

```
  )
end
```

(*End of definition for* `lessthan_hyphenation_case_insensitive`.)

9.13.2 Sorting

`sort_hyphenation_list_none` Sort `hyphenation_list` into its original order of appearance.

```
local function sort_hyphenation_list_none(hyphenation_list)
  table.sort(
    hyphenation_list,
    function(a,b)
      return a[KEY_INDEX] < b[KEY_INDEX]
    end
  )
end
```

(*End of definition for* `sort_hyphenation_list_none`.)

`sort_hyphenation_list_case` Sort `hyphenation_list` case-sensitively.

```
local function sort_hyphenation_list_case(hyphenation_list)
  table.sort(
    hyphenation_list,
    lessthan_hyphenation_case_sensitive
  )
end
```

(*End of definition for* `sort_hyphenation_list_case`.)

`sort_hyphenation_list_nocase` Sort `hyphenation_list` case-insensitively.

```
local function sort_hyphenation_list_nocase(hyphenation_list)
  table.sort(
    hyphenation_list,
    lessthan_hyphenation_case_insensitive
  )
end
```

(*End of definition for* `sort_hyphenation_list_nocase`.)

`process_lang_hyphenation_list_sort` Select the appropriate function for sorting.

```
local sort_hyphenation_list
if tex.count['l__lualisthyphen_sort_int'] == 1 then
  sort_hyphenation_list = sort_hyphenation_list_case
elseif tex.count['l__lualisthyphen_sort_int'] == 2 then
  sort_hyphenation_list = sort_hyphenation_list_nocase
else
  sort_hyphenation_list = sort_hyphenation_list_none
end
```

(*End of definition for* `process_lang_hyphenation_list_sort`.)

9.13.3 Deduplication

deduplicate_hyphenation_list_none Dummy function; does not deduplicate `hyphenation_list`.

```
local function deduplicate_hyphenation_list_none(hyphenation_list)
end
```

(*End of definition for* `deduplicate_hyphenation_list_none`.)

deduplicate_hyphenation_list_case Remove duplicates from `hyphenation_list` case-sensitively.

```
local function deduplicate_hyphenation_list_case(hyphenation_list)
  table.sort(
    hyphenation_list,
    lessthan_hyphenation_case_sensitive
  )
  list_uniq(
    hyphenation_list,
    equal_hyphenation_case_sensitive
  )
end
```

(*End of definition for* `deduplicate_hyphenation_list_case`.)

deduplicate_hyphenation_list_nocase Remove duplicates from `hyphenation_list` case-insensitively.

```
local function deduplicate_hyphenation_list_nocase(hyphenation_list)
  table.sort(
    hyphenation_list,
    lessthan_hyphenation_case_insensitive
  )
  list_uniq(
    hyphenation_list,
    equal_hyphenation_case_insensitive
  )
end
```

(*End of definition for* `deduplicate_hyphenation_list_nocase`.)

deduplicate_hyphenation_list Select the appropriate function for whether duplicates whould be removed.

```
local deduplicate_hyphenation_list
if tex.count['l__lualisthyphen_unique_int'] == 1 then
  deduplicate_hyphenation_list = deduplicate_hyphenation_list_case
elseif tex.count['l__lualisthyphen_unique_int'] == 2 then
  deduplicate_hyphenation_list = deduplicate_hyphenation_list_nocase
else
  deduplicate_hyphenation_list = deduplicate_hyphenation_list_none
end
```

(*End of definition for* `deduplicate_hyphenation_list`.)

9.13.4 Combined processing

process_lang_hyphenation_list Remove duplicates and sort `hyphenation_list`.

```
local function process_lang_hyphenation_list(hyphenation_list)
  deduplicate_hyphenation_list(hyphenation_list)
  sort_hyphenation_list(hyphenation_list)
end
```

(*End of definition for* `process_lang_hyphenation_list`.)

9.14 Writing

write_lang_hyphenation_list_standard

Write out just the hyphenated words in `hyphenation_list` to file handle `f`.

```
local function write_lang_hyphenation_list_standard(f,hyphenation_list,widths)

  for i,v in ipairs(hyphenation_list) do

    if v then
      f:write(v[KEY_DIVISION] .. '\n')
    end

  end

end
```

(*End of definition for* `write_lang_hyphenation_list_standard`.)

write_lang_hyphenation_list_verbose

Write out all hyphenation information in `hyphenation_list` to file handle `f`, in columns as specified in `widths`.

```
local function write_lang_hyphenation_list_verbose(f,hyphenation_list,widths)

  local cols_word = widths[KEY_WORD]
  local cols_division = widths[KEY_DIVISION]
  local cols_page = widths[KEY_PAGE]

  for i,v in ipairs(hyphenation_list) do

    if v then
```

It is possible for KEY_PAGE not to have been set, for instance if the hyphenation occured in a box that was never output.

```
      local page = v[KEY_PAGE]
      if page then
        page = STR_PAGE_PREFIX .. page
      else
        page = STR_PAGE_NONE
      end

      f:write(
        rpad(v[KEY_WORD],cols_word)
        .. STR_ARROW
        .. rpad(v[KEY_DIVISION],cols_division)
        .. STR_SPACE_TWO
        .. lpad(page,cols_page)
        .. STR_SPACE
        .. STR_QUOTE_OPEN
        .. v[KEY_CONTEXT]
        .. STR_QUOTE_CLOSE
        .. '\n'
      )
    end

  end

end
```

(*End of definition for* `write_lang_hyphenation_list_verbose`.)

`write_lang_hyphenation_list` Set `write_lang_hyphenation_list` to be either `write_lang_hyphenation_list_standard` or `write_lang_hyphenation_list_verbose`, depending on the package options.

```
local write_lang_hyphenation_list
if tex.count['l__lualisthyphen_verbose_int'] == 0 then
  write_lang_hyphenation_list = write_lang_hyphenation_list_standard
else
  write_lang_hyphenation_list = write_lang_hyphenation_list_verbose
end
```

(*End of definition for* `write_lang_hyphenation_list`.)

Compute a settings description to insert into file headers.

```
local settings_desc
if tex.count['l__lualisthyphen_verbose_int'] == 0 then
  settings_desc = 'verbose=false'
else
  settings_desc = 'verbose=true'
    .. ',context-before=' .. tex.count['l__lualisthyphen_context_before_int']
    .. ',context-after=' .. tex.count['l__lualisthyphen_context_after_int']
end
if tex.count['l__lualisthyphen_include_non_output_int'] == 0 then
  settings_desc = settings_desc .. ',include-non-output=false'
else
  settings_desc = settings_desc .. ',include-non-output=true'
end
local NONE_CASE_NOCASE = {
  [0] = 'none',
  [1] = 'case',
  [2] = 'nocase'
}
settings_desc = settings_desc
  .. ',sort=' .. NONE_CASE_NOCASE[tex.count['l__lualisthyphen_sort_int']]
  .. ',unique=' .. NONE_CASE_NOCASE[tex.count['l__lualisthyphen_unique_int']]
```

`get_hyphenation_file_path` Get the file to which the list of hyphenated words will be written, based on the given `prefix`, `extension`, `lang_name`, and taking into account any specified output directory for LuaTeX, and with a file header.

```
local function get_hyphenation_file_path(prefix,extension,lang_name)

  local hyphenation_file_path = prefix .. tostring(lang_name) .. extension

  if not status.output_directory then
    return hyphenation_file_path
  end

  if string.sub(status.output_directory,-1,-1) == '/' then
    hyphenation_file_path = status.output_directory
      .. hyphenation_file_path
  else
    hyphenation_file_path = status.output_directory
      .. '/' .. hyphenation_file_path
```

```
  end

  return hyphenation_file_path

end
```

(*End of definition for* `get_hyphenation_file_path`.)

`process_write_lang_hyphenation_list` Process and write out the `hyphenation_list` (which will be for the language with the numerical `lang_id`) to a file with the given `prefix` and `extension`, using `widths` for the 'columns' in verbose mode.

```
local function process_write_lang_hyphenation_list(
    prefix,extension,lang_id,hyphenation_list,widths
)

  process_lang_hyphenation_list(hyphenation_list)

  local lang_name = language_table[lang_id]
  local lang_desc
  if not lang_name then
    lang_name = lang_id
    lang_desc = 'language with ID ' .. lang_id
  else
    lang_desc = 'language "' .. lang_name .. '" (ID ' .. lang_id .. ')'
  end

  local f = io.open(get_hyphenation_file_path(prefix,extension,lang_name),'w')

  f:write('% Chosen hyphenations for ' ..  lang_desc .. '\n')
  f:write('% Generated by lua-list-hyphen (' .. settings_desc .. ')\n')

  write_lang_hyphenation_list(f,hyphenation_list,widths)
  f:close()

end
```

(*End of definition for* `process_write_lang_hyphenation_list`.)

`process_write_hyphenation_lists` Sort `hyphenation_list` into per-language lists and write them out to separate files.

```
local function process_write_hyphenation_lists(prefix,extension)

  local lang_hyphenation_table = {}
  local lang_widths_table = {}
```

Iterate through all the stored hyphenations. Sort them into per-language lists (creating the list the first time each language is encountered) and also storing the maximum width of values, for output alignment.

```
  for _,h in pairs(hyphenation_list) do

    if h[KEY_PAGE] or tex.count['l__lualisthyphen_include_non_output_int'] == 1 then

      local lang = h[KEY_LANG]

      local t = lang_hyphenation_table[lang]
```

```
      if not t then
        lang_hyphenation_table[lang] = {}
        t = lang_hyphenation_table[lang]
      end

      local widths = lang_widths_table[lang]
      if not widths then
        lang_widths_table[lang] = {
          [KEY_WORD] = 0,
          [KEY_DIVISION] = 0,
          [KEY_PAGE] = 0
        }
        widths = lang_widths_table[lang]
      end

      widths[KEY_WORD] = math.max(
        widths[KEY_WORD],
        unicode.utf8.len(h[KEY_WORD])
      )
      widths[KEY_DIVISION] = math.max(
        widths[KEY_DIVISION],
        unicode.utf8.len(h[KEY_DIVISION])
      )
      widths[KEY_PAGE] = math.max(
        widths[KEY_PAGE],
        unicode.utf8.len(tostring(h[KEY_PAGE]))
      )

      table.insert(t,h)

    end

  end
```

Adjust the maximum width for the page output, since there is a prefix and a 'no page' indicator to consider.

```
  for _,widths in pairs(lang_widths_table) do
    widths[KEY_PAGE] = math.max(
      widths[KEY_PAGE] + unicode.utf8.len(STR_PAGE_PREFIX),
      unicode.utf8.len(STR_PAGE_NONE)
    )
  end
```

If polyglossia is in use, populate `language_table`.

```
  polyglossia_get_language_names()
```

For each language, process and write out its hyphenations to a file.

```
  for k,v in pairs(lang_hyphenation_table) do
    process_write_lang_hyphenation_list(prefix,extension,k,v,lang_widths_table[k])
  end

end
```

(*End of definition for* `process_write_hyphenation_lists`.)

9.15 Export public functions

Finally, make available the functions that will be called from the LaTeX frontend using `\lua_now:n`.

```
return {
  process_write_hyphenation_lists = process_write_hyphenation_lists,
  set_hyphenation_page = set_hyphenation_page,
  babel_save_language_name = babel_save_language_name,
}
```

⟨/lua⟩

Index

The italic numbers denote the pages where the corresponding entry is described, numbers underlined point to the definition, all others indicate the places where it is used.

M

N

O

P

R

S

T

U

V

W