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

Alan J. Cain†

Released 2026-08-23

Abstract

This LuaLaTeX package can check each word that has been hyphenated across lines against language-specific dictionaries of valid divisions, optionally adding visual flags to the generated document, and write files listing hyphenations (and their contexts), for external checking.

Demonstration

The (small) division dictionary for British English used for classifying hyphenations in this demonstration:

```
de-cease
en-able
fu-ture
gen-er-ous
ir-regu-lar
ob-tained
pres-ent (adj./noun)
pre-sent (verb)
re-main-der
trib-un-itian
un-time-ly
```

From Gibbon's *Decline and Fall of the Roman Empire*, ch.III
(set in narrow columns to increase frequency of hyphenation)

In elective monarchies, the vacancy of the throne is a moment big with danger and mischief. The Roman emperors, desirous to spare the legions that interval of suspense, and the temptation of an irregular choice, invested their designed successor with so large a share of present power, as should enable him, after their decease, to assume the remainder without suffering the empire to perceive the change of masters. Thus Augustus, after all his fairer prospects had been snatched from him by untimely deaths, rested his last hopes on Tiberius, obtained for his adopted son the censorial and tribunitian powers, and dictated a law, by which the future prince was invested with an authority equal to his own over the provinces and the armies. Thus Vespasian subdued the generous mind of his eldest son.

Using the supplied division dictionary, lua-list-hyphen has classified the hyphenations, flagged the results visually, and written to a file those that are either (1) invalid (not matching the dictionary, like *irreg-ular*), (2) ambiguous (matching one but not all entries for the word in the dictionary, like *pres-ent*), or (3) unknown (not in the dictionary, like the name *Ves-pasian*):

```
irregular   -> irreg-ular    (Inva.)     p.1 "of an irregular choice, invested"
present     -> pres-ent      (Ambi.)     p.1 "share of present power, as"
tribunitian -> tribuni-tian  (Inva.)     p.1 "censorial and tribunitian powers, and"
Vespasian   -> Ves-pasian    (Unkn.)     p.1 "armies. Thus Vespasian subdued the"
```

*This document describes v0.43.2, last revised 2026-08-23.

†a.j.cain (AT) gmail.com

Contents

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 generally, 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

[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).

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. Patrick Gundlach's LuaLaTeX package lua-check-hyphen[3] offers this facility. It checks hyphenated words against a list of accepted hyphenations, writes unknown hyphenations to a file, and optionally flags them in the generated PDF. 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 writes hyphenated words from each language to a separate file, so that they can be checked (manually or by an external program). The user can optionally supply per-language dictionaries of valid divisions, which are used to classify each hyphenation as either valid, invalid, ambiguous (like *re-cord*/*rec-ord*), or unknown (not in the relevant division dictionary). The user can choose to filter and not write out those hyphenations that are known to be valid with respect to the relevant division dictionary. lua-list-hyphen can also add visual flags either to every hyphenation in the generated PDF, or only those that are not classified as valid.

Licence. lua-list-hyphen is released under the LaTeX Project Public Licence v1.3c or later.[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.[5]

Other resources. The author has written a simple console Python application `hyphenassist`[6] that checks the output lists of 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 this package to check hyphenations in his own books.[7]

The author's very limited and partial division dictionary for British English is available at Codeberg.[8] He has assembled it through gradual checking of hyphenations in his own books, and so it is heavily skewed towards mathematical, philosophical, and historical terms and names. It includes divisions for some rarer technical terms and more obscure names, not found in references, which have been obtained by analysis, deduction, and analogy.

[3] URL: https://ctan.org/pkg/lua-check-hyphen
[4] URL: https://www.latex-project.org/lppl.txt
[5] URL: https://codeberg.org/ajcain/lua-list-hyphen
[6] URL: https://codeberg.org/ajcain/hyphenassist.
[7] In particular, *Form & Number: A History of Mathematical Beauty*. URL: https://archive.org/details/cain_formandnumber_ebook_large.
[8] URL: https://codeberg.org/ajcain/division-dict

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 the association of LuaTeX numerical language IDs with 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 with `\usepackage{lua-list-hyphen}`. By default, lua-list-package writes hyphenations to files `\jobname-`⟨*lang*⟩`.hyph`, without classification, sorting, or removal of duplicates. If either babel or polyglossia is loaded, ⟨*lang*⟩ will usually be the name of the language as defined in hyph-utf8 (which does not always coincide with the babel or polyglossia name or variant). If neither of these packages is in use, or if the language has been defined otherwise, ⟨*lang*⟩ is the LuaTeX numerical language ID.

To modify this basic functionality, add package options, which are a comma-separated list of key–value pairs. Here are a few to get started. Note that these options can be applied either globally or per-language; see the introduction to Section 7 for an explanation.

- To sort the output hyphenations case-sensitively or case-insensitively, add the package option `sort=case` or `sort=nocase`. See Subsection 7.7 for further details.
- To remove duplicates from the output hyphenations case-sensitively or case-insensitively, add the package option `unique=case` or `unique=nocase`. See Subsection 7.7 for further details.
- To classify hyphenations as valid/invalid/ambiguous/unknown, specify a file containing a dictionary of valid divisions using the package option `division-dict-path=`⟨*path*⟩ (or the synonymous `allowlist-path=`⟨*path*⟩).

  In particular, to use a division dictionary as a basic allowlist of hyphenations that should not be output, add `output-mode=non-valid`.

  See Section 5 for a full account of classification of hyphenations, Subsection 7.2 for further details of `division-dict-path` description, and Subsection 7.5 for `output-mode`.
- To add visual flags for hyphenations to the output PDF, add the package option `flags=all` or `flags=non-valid`. See Subsection 7.4 for further details.

See Section 7 for all the options.

5 Classifications of hyphenations

lua-list-hyphen can read use a dictionary of valid divisions to check each hyphenation that occurs during typesetting. The user can specify the file containing the dictionary (either globally or per-language) using the option `division-dict-path` (see Subsection 7.2).

These files should list accepted divisions of words, one per line. Multiple division points can be marked in each word (as in the TEX `\hyphenation` command and its babel and polyglossia equivalents). A division point can be marked with a hyphen `-`, a vertical bar `|`, or a broken vertical bar `¦`. (The latter two symbols are allowed to match the notation used in *New Oxford Spelling Dictionary* for preferred and acceptable division points.) Whitespace at the start of the line is ignored. Any text following whitespace after the accepted division is ignored and can be used for a note, perhaps to clarify different divisions of orthographically identical words. Thus lines in a division dictionary for British English might be:

```
at-tri-bute     (noun)
at-trib-ute     (verb)
at-trib-uted
at-tri-butes    (noun)
at-trib-utes    (verb)
at-tri-bu-tion
```

Each hyphenation is compared to the division dictionary (if specified) for the appropriate language and is classified as one of: *valid*, *invalid*, *ambiguous*, or *unknown*:

valid A hyphenation is *valid* if there is an entry in the division dictionary for that word and the chosen hyphenation matches *every* entry for that word. Using the example lines from the division dictionary above, *at-tributed* is valid because it matches the entry `at-trib-uted`. The hyphenation *at-tribute* is also valid because it matches *both* the entries `at-trib-ute` and `at-tri-bute`.

invalid A hyphenation is *invalid* if there is an entry in the division dictionary for that word and the chosen hyphenation matches *no* entry for that word. Using the example lines from the division dictionary above, *attrib-ution* is invalid because it does no match the entry `at-tri-bu-tion`.

ambiguous A hyphenation is *ambiguous* if there is are at least two entries in the division dictionary for that word and the chosen hyphenation matches at least one entry for that word *and* does not match at least one entry for that word. Using the example lines from the division dictionary above, *attrib-ute* is ambiguous because it matches the entry `at-trib-ute` and does not match the entry `at-tri-bute`.

unknown A hyphenation is *unknown* if there is no entry for the word in the division dictionary. (If no division dictionary is supplied for a language, all hyphenations in that language are classified as unknown.)

If `output-mode=all`, all hyphenations, regardless of classification, are written out. If `output-mode=non-valid`, only hyphenations that have been classified as invalid, ambiguous, or unknown are written out. Thus a division dictionary could be used as a simple allowlist: it could simply contain hyphenations that have been checked, with `output-mode=non-valid` meaning that any hyphenations not appearing in the allowlist would be written out.

When `output-verbose=true`, the classification is written to the output file along with the other data about the hyphenation.

If `flags=all`, visual flags are added next to every hyphenation according to its classification: valid ✓, invalid ✗, ambiguous ?, unknown ∅. If `flags=non-valid`, the flags on valid hyphenations are suppressed and only ✗, ?, and ∅ appear.

6 Output format

Each output files begins with a header (which can be disabled) and then lists the hyphenations for the corresponding language, one per line.

If `output-header=true`, 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, and the path to the division dictionary that was used to classify the words (if specified). `output-header=false`, this header does not appear.

Each line of the remainder of the file describes one hyphenation.

- When `output-verbose=false`, the line contains only the hyphenated word.
- When `output-verbose=true`, the line contains the original undivided word, the hyphenated word, the classification of the hyphenation (see Section 5), the page number where the hyphenated word appears (or, to be precise, begins), and the context in which the hyphenated word appears. Each part of the output is padded so that the 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 `" "`. Thus a line of output might be:

  ```
  thinking  -> think-ing  (Valid)  p.4 "and visual thinking, is its"
  ```

  If no division dictionary was specified for the language, the classification of the hyphenation (`(Valid)` in this example) does not appear (since all hyphenations are unknown).

7 Package options

Package options are given as a comma-separated list of ⟨*key*⟩=⟨*value*⟩ pairs. Most package options can be set either *globally*, applying to all languages, or *per-language*, applying only to a specific language. Global options apply to all languages unless overrriden for that language.

Two options (`output-prefix`, `output-extension`) can only be set globally and one (`output-path`) can only be set per-language. All other options can be set both globally and per-language.

Top-level ⟨*key*⟩=⟨*value*⟩ options apply globally. Options for a specific language are given in the form ⟨*lang-name*⟩={⟨*options*⟩}, where ⟨*lang-name*⟩ can be any of the synonyms for the language specified in the hyph-utf8 documentation, and ⟨*options*⟩ is a comma-separated list of ⟨*key*⟩=⟨*value*⟩ pairs. A warning will be issued if ⟨*lang-name*⟩ is unrecognized.

For example, suppose package options are given as follows:

```
\usepackage[
  output-verbose=true,
  flag-mode=all,
  ukenglish={
    output-verbose=false,
    division-dict-path=division-dict-en-GB.txt,
```

```
  },
]{lua-list-hyphen}
```

Then the option `flag-mode=all` applies to all languages, including `ukenglish`. The option `output-verbose=true` applies to all language except `ukenglish`, for which `output-verbose=false` applies. The option `division-dict-path=division-dict-en-GB.txt` applies only to `ukenglish`.

7.1 Output files

The following keys specify the paths to the files where hyphenations are written. If an output directory has been specified (using the command-line option `--output-directory`), paths are relative to this directory.

`output-path` (*Per-language only option.*) `output-path` can be used to specify, for a particular language, the file to which hyphenations for that language are written. If no `output-path` is specified for a language, the fallback output file is determined by `output-prefix` and `output-extension`. (*Default:* `None`)

This option cannot be set globally, since the intention is to write each language is written to a different output file. If the same `output-path` is given for two different languages, and there are hyphenations in both languages, one list of hyphenations will overwrite the other.

`output-prefix` `output-extension` (*Global-only options.*) These two keys are used for the fallback output file for a language when no path has been specified using `output-path`. The fallback is ⟨*prefix*⟩⟨*lang*⟩⟨*extension*⟩, where ⟨*prefix*⟩ is the value of `output-prefix`, ⟨*extension*⟩ is the value of `output-extension`, and ⟨*lang*⟩ will usually be the name of the language as defined in hyph-utf8 (which does not always coincide with the babel or polyglossia name). If neither of these packages is in use, or if the language has been defined otherwise, ⟨*lang*⟩ will be the LuaTEX numerical language ID. (*Default:* `output-prefix=\jobname-` (note the hyphen); `output-extension=.hyph` (note the dot).)

These options cannot be set per-language, only globally. To alter the output file per-language, use the `output-path` option.

7.2 Division dictionary files

`division-dict-path` The key `division-dict-path` can be used to specify the file where lua-list-hyphen looks for a dictionary of valid divisions against which to check each hyphenation found during typesetting. (*Default:* `None`)

Any path specified using this key is passed to the kpathsea library, so a specified division dictionary can be anywhere in the directories it searches (such as the user or system `texmf/` hierarchies or paths specified using `TEXINPUTS`).

`allowlist-path` The key `allowlist-path` is a synonym for `division-dict-path`.

7.3 Hyphenation checking

`validation-preprocessor` The key `validation-preprocessor` can be used to specify a Lua function that is applied to each hyphenated word before it is checked against the division dictionary. Its value should be the name of a Lua function that takes a word — perhaps including a hyphen — and returns a possibly modified version. The result is used in place of the original when checking against the division dictionary, and may thus affect the classification of hyphenations, but does not alter the words written out. (*Default:* `None`)

For example, the following function deletes `'s` from the end of a word:

```
function english_strip_possessives(s)
  return string.gsub(s','s$','')
end
```

Users may wish to use a function like this for English, so that the division dictionary does not have to contain separate entries for possessives of nouns and names. If `validation-preprocessor=english_strip_possessives` is specified then the hyphenation *math-ematician's* would be changed to *math-ematician* for checking against the division dictionary, which could contain *math-em-at-ician* without having to contain *math-em-at-ician's*.

lua-list-hyphen has two built-in Lua functions for this purpose:

`lualisthyphen.preprocessors.identity` This function simplye returns the word unaltered.

`lualisthyphen.preprocessors.english_strip_possessives` The same function as shown above, which removes `'s` from the end of a word.

lua-list-hyphen first looks in `lualisthyphen.preprocessors` for ⟨*`function`*⟩, and, if a function with the given name is not found there, in the global context. Thus `validation-preprocessor=english_strip_possessives` specifies the built-in function listed above. `lualisthyphen.preprocessors` can be used a location for further such functions written by the user, but this is not required.

`validation-case-sensitve` This boolean key controls whether checking hyphenations against dictionaries of valid divisions is case-sensitive. If it is set to `true`, then (for example) the hyphenation *Pol-ish* (of Poland) would not match the division *pol-ish* (shine). (*Default:* `false`)

7.4 Flags

`flag-mode` The option `flag-mode` controls whether hyphenations are 'flagged' with an adjacent mark, indicating whether they are valid ✅, invalid ❌, ambiguous ❓ (like *record*, which has different valid divisions as a noun and as a verb), or unknown 🚫, with respect to the dictionary of valid divisions for the corresponding language. (If no dictionary is found for the language, all hyphenations in that language will be treated as unknown.) Its possible values are:

`none`: No flags are added.

`all`: Flags are shown for all hyphenations.

`non-valid`: Flags are shown only for non-valid hyphenations (that is, those that classified as invalid, ambiguous, or unknown).

(*Default:* `none`)

7.5 Output filtering

`output-mode` The option `output-mode` controls which hyphenations are listed in the output files. Its possible values are:

`all`: All hyphenations are listed

`non-valid`: Only non-valid hyphenations (with respect to the dictionary of valid divisions) are listed (that is, those that are invalid, ambiguous, or unknown).

(*Default:* `none`)

`output-non-typeset` Boolean option determining whether hyphenated words that are never typeset on the page are listed in the output. (For instance, a hyphenated word might occur in text that a package temporarily typesets into a box, measures, and then discards.) If `output-verbose=true`, then `p.`⟨*page*⟩ is replaced by `<none>` for such. hyphenations. Whether these hyphenations are listed is afftected by `output-mode`. (*Default:* `false`)

7.6 Output format

`output-header` The boolean key `output-header` controls whether a header is written to the output file, specifying the language, the relevant package options used, and (if relevant) the dictionary of valid divisions. (*Default:* `true`)

`output-verbose` The boolean option `output-verbose` controls how much information is written to the file about each hyphenation. When `true`, for each hyphenation, both the undivided original and the divided word are written out, the classification of the hyphenation as valid/invalid/ambiguous/unknown, 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 hyphenation are written as context when `output-verbose=true`. The key `context` is simply a shortcut for setting `context-before` and `context-after` to the same value. Only words from the same paragraph are written as context. (*Default:* `2`)

7.7 Output sorting and deduplication

`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 `output-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`)

8 Usage notes

8.1 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 the `unique` package option.

8.2 Limitations

8.2.1 Unicode

lua-list-hyphen uses LuaTEX's built-in Unicode functions for pattern matching and converting between upper and lower case. These function 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 classified, sorted, or deduplicated correctly. Users may prefer to leave these tasks to an external program that adheres to the current Unicode standard. (The author's `hyphenassist` program, mentioned on page 1 is as up-to-date as the Python installation on which it runs.)

8.2.2 Association of language names and numerical LuaTEX IDs

The lua-list-hyphen only interfaces with babel or polyglossia to determine the hyph-utf8 name for the language (in the sense of a set of hyphenation patterns) that has been assigned to each LuaTEX numerical language ID. The hyph-utf8 name does not necessarily match the babel or polyglossia name or variant: for instance, for British English, polyglossia uses the language `english` and the variant `british`, while hyph-utf8 uses the name `ukenglish` with synonyms `UKenglish` and `british`. (There seems to be no straightforward way of mapping between LuaTEX IDs and specified babel/polyglossia names and variants.)

9 Appendix: demonstration source

This following subsections contain the source code and example division dictionary used to produce the demonstration on the first page of this document.

9.1 LuaLaTeX source

```
\documentclass[twocolumn,oneside]{book}

\usepackage[
paperwidth=100mm,
paperheight=77.48mm,
inner=5mm,
width=87mm,
columnsep=7mm,
top=5mm,
height=192pt,
nohead,
nofoot,
]{geometry}

\usepackage{polyglossia}
\setdefaultlanguage[variant=british]{english}

\usepackage[
output-verbose=true,
output-header=false,
sort=nocase,
unique=nocase,
context-before=2,
context-after=2,
flag-mode=all,
output-mode=non-valid,
division-dict-path=division-dict-ukenglish.txt
]{lua-list-hyphen}

\pagestyle{empty}
\setlength{\parfillskip}{0pt}

\begin{document}

In elective monarchies, the vacancy of the throne is a moment
big with danger and mischief. The Roman emperors, desirous to
spare the legions that interval of suspense, and the temptation of an
irregular choice, invested their designed successor with so large a share
of pres\-ent power, as should enable him, after their decease, to assume
the remainder without suffering the empire to perceive the change of
masters. Thus Augustus, after all his fairer prospects had been
snatched from him by untimely deaths, rested his last hopes on
Tiberius, obtained for his adopted son the censorial and tribunitian
powers, and dictated a law, by which the future prince was invested
with an authority equal to his own over the provinces and the armies.
```

```
Thus Vespasian subdued the generous mind of his eldest son.

\end{document}
```

9.2 Division dictionary

(Saved as `division-dict-ukenglish.txt`, to match `division-dict-paths` in the package options shown in the LuaLaTeX source code above.)

```
de-cease
en-able
fu-ture
gen-er-ous
ir-regu-lar
ob-tained
pres-ent (adj./noun)
pre-sent (verb)
re-main-der
trib-un-itian
un-time-ly
```

10 Implementation (LaTeX package)

```
⟨*package⟩
⟨@@=lualisthyphen⟩
```

10.1 Initial set-up

Package identification/version information.

```
\NeedsTeXFormat{LaTeX2e}[2020-02-02]
\ProvidesExplPackage{lua-list-hyphen}{2026-08-23}{0.43.2}
  {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 }
  }
```

10.2 Options

10.2.1 Global-only options

\l__lualisthyphen_output_prefix_str String option for the prefix of files to which hyphenations are written.

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

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

\l__lualisthyphen_output_extension_str String option for the extension of files to which hyphenations are written.

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

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

10.2.2 Per-language and global options

\l__lualisthyphen_division_dict_path_str String option to specify the path from which to read the division dictionary.

```
\keys_define:nn { lua-list-hyphen/global-per-lang }{
  division-dict-path .str_set:N = \l__lualisthyphen_division_dict_path_str,
}
\keys_define:nn { lua-list-hyphen/global-per-lang }{
  allowlist-path .meta:n = { division-dict-path=#1 },
}
```

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

__lualisthyphen_validation_preprocessor_str String option to specify validation preprocessor.

```
\keys_define:nn { lua-list-hyphen/global-per-lang }{
  validation-preprocessor .str_set:N = \l__lualisthyphen_validation_preprocessor_str,
}
```

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

ualisthyphen_validation_case_sensitive_bool Boolean option to specify whether validation of divisions is case-sensitive.

```
\keys_define:nn { lua-list-hyphen/global-per-lang }{
  validation-case-sensitve .bool_set:N
    = \l__lualisthyphen_validation_case_sensitive_bool
}
```

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

\l__lualisthyphen_flag_mode_int Choice option to specify whether flags are added to the output PDF file to indicate the classification of hyphenations.

```
\int_new:N\l__lualisthyphen_flag_mode_int
\keys_define:nn { lua-list-hyphen/global-per-lang }{
  flag-mode .choices:nn = { none, all, non-valid }{
    \int_set:Nn\l__lualisthyphen_flag_mode_int{ \l_keys_choice_int - 1 }
  },
}
```

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

\l__lualisthyphen_output_mode_int Choice option to indicate which hyphenations should be listed in the output files.

```
\int_new:N\l__lualisthyphen_output_mode_int
\keys_define:nn { lua-list-hyphen/global-per-lang }{
  output-mode .choices:nn = { all, non-valid }{
    \int_set:Nn\l__lualisthyphen_output_mode_int{ \l_keys_choice_int - 1 }
  },
}
```

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

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

```
\keys_define:nn { lua-list-hyphen/global-per-lang }{
  output-non-typeset .bool_set:N = \l__lualisthyphen_output_non_typeset_bool,
}
```

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

\l__lualisthyphen_output_header_bool Boolean option to indicate whether a header should be written in the output.

```
\keys_define:nn { lua-list-hyphen/global-per-lang }{
  output-header .bool_set:N = \l__lualisthyphen_output_header_bool,
  output-header .initial:n = {true},
}
```

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

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

```
\keys_define:nn { lua-list-hyphen/global-per-lang }{
  output-verbose .bool_set:N = \l__lualisthyphen_output_verbose_bool,
}
```

(*End of definition for* `\l__lualisthyphen_output_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/global-per-lang }{
  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 .meta:n = {
    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/global-per-lang }{
  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/global-per-lang }{
  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`.)

Make options in the `global-per-lang` submodule options available at the top level in the `lua-list-hyphen` module.

```
\keys_define:nn {  }{
  lua-list-hyphen .inherit:n = lua-list-hyphen/global-per-lang,
}
```

10.2.3 Per-language options

Make options in the `global-per-lang` submodule options available in the `per-lang` submodule.

```
\keys_define:nn { lua-list-hyphen }{
   per-lang .inherit:n = lua-list-hyphen/global-per-lang,
}
```

There is one option that is is strictly per-language:

\l__lualisthyphen_output_path_str Option to specify the path to write hyphenations.

```
\keys_define:nn { lua-list-hyphen/per-lang }{
  output-path .str_set:N = \l__lualisthyphen_output_path_str,
}
```

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

10.2.4 Warnings for per-language-only and global-only options

Emit a warning when the per-language-only option `output-path` is set globally.

```
\msg_new:nnnn{ lua-list-hyphen }{ per-language-only }
  { The~option~"#1"~only~has~effect~per-language. }
  { The~option~"#1"~has~been~specified~globally~and~will~be~ignored. }
\keys_define:nn { lua-list-hyphen }{
  output-path .code:n = {
    \msg_warning:nne{ lua-list-hyphen }{ per-language-only }
      { \str_use:N\l_keys_key_str }
  },
}
```

Emit a warning when either of the global-only options `output-prefix` or `output-extension` is set per-language.

```
\msg_new:nnnn{ lua-list-hyphen }{ global-only }
  { The~option~"#1"~only~has~effect~globally. }
  { The~option~"#1"~has~been~specified~for~a~particular~language~and~will~be~ignored. }
\keys_define:nn { lua-list-hyphen/per-lang }{
  output-prefix .code:n = {
    \msg_warning:nne{ lua-list-hyphen }{ global-only }
      { \str_use:N\l_keys_key_str }
  },
  output-extension .code:n = {
    \msg_warning:nne{ lua-list-hyphen }{ global-only }
      { \str_use:N\l_keys_key_str }
  },
}
```

10.3 Deprecated options

For deprecated options, emit a warning and forward the setting to the replacement option.

```
\msg_new:nnnn{ lua-list-hyphen }{ deprecated-option }
  { The~option~"#1"~works~but~is~deprecated.~Please~use~"#2"~instead. }
  { The~option~"#1"~may~be~removed~in~a~future~version. }
\keys_define:nn { lua-list-hyphen }{
  verbose .code:n = {
    \msg_warning:nnee{ lua-list-hyphen }{ deprecated-option }
      { \str_use:N\l_keys_key_str }{ output-verbose }
    \keys_set:nn{ lua-list-hyphen }{ output-verbose=#1 }
  },
  include-non-output .code:n = {
    \msg_warning:nnee{ lua-list-hyphen }{ deprecated-option }
      { \str_use:N\l_keys_key_str }{ include-non-typeset }
    \keys_set:nn{ lua-list-hyphen }{ include-non-typeset=#1 }
  },
  prefix .code:n = {
```

```
    \msg_warning:nnee{ lua-list-hyphen }{ deprecated-option }
      { \str_use:N\l_keys_key_str }{ output-prefix }
    \keys_set:nn{ lua-list-hyphen }{ output-prefix=#1 }
  },
  extension .code:n = {
    \msg_warning:nnee{ lua-list-hyphen }{ deprecated-option }
      { \str_use:N\l_keys_key_str }{ output-extension }
    \keys_set:nn{ lua-list-hyphen }{ output-extension=#1 }
  }
}
```

10.3.1 Developer option

\l__lualisthyphen_debug_bool

Option to specify whether debug information is written to the terminal. Undocumented and not intended for end-users.

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

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

10.3.2 Processing per-language options

Language-specific options should be processed *after* all other options, so that they inherit the 'general' options. `\ProcessKeyOptions` does not seem to allow for selective processing similar to `\keys_set_known:nn`, so store all unknown options in a clist for later processing.

```
\keys_define:nn { lua-list-hyphen }{
  unknown .code:n = {
    \__lualisthyphen_store_lang_options:en{\str_use:N\l_keys_key_str}{#1}
  }
}
```

__lualisthyphen_store_lang_options:nn

Store the key–value option whose key is `#1` and whose value is `#2` in `\l__lualisthyphen_lang_options_clist`.

```
\clist_new:N\l__lualisthyphen_lang_options_clist

\cs_new:Npn \__lualisthyphen_store_lang_options:nn #1#2
  {
    \clist_put_right:Nn\l__lualisthyphen_lang_options_clist{#1={#2}}
  }
\cs_generate_variant:Nn\__lualisthyphen_store_lang_options:nn{ en }
```

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

10.4 Lua backend

Load the Lua backend.

```
\lua_load_module:n{lua-list-hyphen}
```

10.5 Processing package options

Process the package options. All the unknown options — in principle, language-specific options — are stored in `\l__lualisthyphen_lang_options_clist`.

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

10.5.1 Interface for passing options to the Lua backend

`\__lualisthyphen_call_lua_boolean:nN` Call the Lua function given in `#1` with the value of the boolean given in `#2`.

```
\cs_new:Npn \__lualisthyphen_call_lua_boolean:nN #1#2
  {
    \lua_now:e{ lualisthyphen.#1(\bool_to_str:N #2) }
  }
```

(*End of definition for* `\__lualisthyphen_call_lua_boolean:nN`.)

`\__lualisthyphen_luastr_or_nil:N` Leave in the input stream the single-quoted content of a string variable, if it is non-empty, otherwise `nil`.

```
\cs_new:Npn \__lualisthyphen_luastr_or_nil:N #1
  {
    \str_if_empty:NTF #1
      { nil }
      { '\str_use:N #1' }
  }
```

(*End of definition for* `\__lualisthyphen_luastr_or_nil:N`.)

`\__lualisthyphen_lua_set_global_option:nn` `\__lualisthyphen_lua_set_global_option:ne` pass a global option to the Lua backend. `#1` is the name for the option in the Lua table in the backend. `#2` is the value as a literal string, boolean, integer, or `nil`. In particular, strings literals passed as `#2` must be quoted.

```
\cs_new:Npn \__lualisthyphen_lua_set_global_option:nn #1#2
  {
    \lua_now:n{ lualisthyphen.set_global_option('#1',#2) }
  }
\cs_generate_variant:Nn\__lualisthyphen_lua_set_global_option:nn{ ne }
```

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

`\__lualisthyphen_lua_set_lang_option:nnn` `\__lualisthyphen_lua_set_lang_option:nne` Pass a language-specific option to the Lua backend. `#1` should be a literal Lua string with the language name. `#2` is the name for the option in the Lua table in the backend. `#3` is the value as a literal string, boolean, or integer. In particular, strings literals passed as `#1` or `#3` must be quoted.

```
\cs_new:Npn \__lualisthyphen_lua_set_lang_option:nnn #1#2#3
  {
    \lua_now:n{ lualisthyphen.set_lang_option(#1,'#2',#3) }
  }
\cs_generate_variant:Nn\__lualisthyphen_lua_set_lang_option:nnn{ nne }
```

(*End of definition for* `\__lualisthyphen_lua_set_lang_option:nnn`.)

10.5.2 Passing global options to the Lua backend

First of all, pass the `debug` setting to the Lua backend.

```
\__lualisthyphen_call_lua_boolean:nN
  {set_debug}\l__lualisthyphen_debug_bool
```

At this point the variables contain 'global' options.

```
\__lualisthyphen_lua_set_global_option:ne{division-dict-path}
  {\__lualisthyphen_luastr_or_nil:N\l__lualisthyphen_division_dict_path_str}
\__lualisthyphen_lua_set_global_option:ne{output-prefix}
  {\__lualisthyphen_luastr_or_nil:N\l__lualisthyphen_output_prefix_str}
\__lualisthyphen_lua_set_global_option:ne{output-extension}
  {\__lualisthyphen_luastr_or_nil:N\l__lualisthyphen_output_extension_str}
\__lualisthyphen_lua_set_global_option:ne{validation-preprocessor}
  {\__lualisthyphen_luastr_or_nil:N\l__lualisthyphen_validation_preprocessor_str}
\__lualisthyphen_lua_set_global_option:ne{validation-case-sensitive}
  {\bool_to_str:N\l__lualisthyphen_validation_case_sensitive_bool}
\__lualisthyphen_lua_set_global_option:ne{flag-mode}
  {\int_use:N\l__lualisthyphen_flag_mode_int}
\__lualisthyphen_lua_set_global_option:ne{output-mode}
  {\int_use:N\l__lualisthyphen_output_mode_int}
\__lualisthyphen_lua_set_global_option:ne{output-non-typeset}
  {\bool_to_str:N\l__lualisthyphen_output_non_typeset_bool}
\__lualisthyphen_lua_set_global_option:ne{output-header}
  {\bool_to_str:N\l__lualisthyphen_output_header_bool}
\__lualisthyphen_lua_set_global_option:ne{output-verbose}
  {\bool_to_str:N\l__lualisthyphen_output_verbose_bool}
\__lualisthyphen_lua_set_global_option:ne{context-before}
  {\int_use:N\l__lualisthyphen_context_before_int}
\__lualisthyphen_lua_set_global_option:ne{context-after}
  {\int_use:N\l__lualisthyphen_context_after_int}
\__lualisthyphen_lua_set_global_option:ne{unique}
  {\int_use:N\l__lualisthyphen_unique_int}
\__lualisthyphen_lua_set_global_option:ne{sort}
  {\int_use:N\l__lualisthyphen_sort_int}
```

10.5.3 Passing per-language options to the Lua backend

Load language names and synonyms before processing per-language options.

```
\lua_now:n{ lualisthyphen.lang_load_names() }
```

__lualisthyphen_lua_set_per_lang_options:n

Pass all options for a given language to the Lua backend. `#1` should be any name of the language as a (quoted) Lua string literal.

```
\cs_new:Npn \__lualisthyphen_lua_set_per_lang_options:n #1
  {
    \__lualisthyphen_lua_set_lang_option:nne{#1}{division-dict-path}
      {\__lualisthyphen_luastr_or_nil:N\l__lualisthyphen_division_dict_path_str}
    \__lualisthyphen_lua_set_lang_option:nne{#1}{output-path}
      {\__lualisthyphen_luastr_or_nil:N\l__lualisthyphen_output_path_str}
    \__lualisthyphen_lua_set_lang_option:nne{#1}{validation-preprocessor}
      {\__lualisthyphen_luastr_or_nil:N\l__lualisthyphen_validation_preprocessor_str}
    \__lualisthyphen_lua_set_lang_option:nne{#1}{validation-case-sensitive}
      {\bool_to_str:N\l__lualisthyphen_validation_case_sensitive_bool}
    \__lualisthyphen_lua_set_lang_option:nne{#1}{flag-mode}
```

```
      {\int_use:N\l__lualisthyphen_flag_mode_int}
    \__lualisthyphen_lua_set_lang_option:nne{#1}{output-mode}
      {\int_use:N\l__lualisthyphen_output_mode_int}
    \__lualisthyphen_lua_set_lang_option:nne{#1}{output-non-typeset}
      {\bool_to_str:N\l__lualisthyphen_output_non_typeset_bool}
    \__lualisthyphen_lua_set_lang_option:nne{#1}{output-header}
      {\bool_to_str:N\l__lualisthyphen_output_header_bool}
    \__lualisthyphen_lua_set_lang_option:nne{#1}{output-verbose}
      {\bool_to_str:N\l__lualisthyphen_output_verbose_bool}
    \__lualisthyphen_lua_set_lang_option:nne{#1}{context-before}
      {\int_use:N\l__lualisthyphen_context_before_int}
    \__lualisthyphen_lua_set_lang_option:nne{#1}{context-after}
      {\int_use:N\l__lualisthyphen_context_after_int}
    \__lualisthyphen_lua_set_lang_option:nne{#1}{unique}
      {\int_use:N\l__lualisthyphen_unique_int}
    \__lualisthyphen_lua_set_lang_option:nne{#1}{sort}
      {\int_use:N\l__lualisthyphen_sort_int}
  }
\cs_generate_variant:Nn\__lualisthyphen_lua_set_per_lang_options:n{ e }
```

(*End of definition for* `\__lualisthyphen_lua_set_per_lang_options:n`.)

Define a new module that only handles unknown keys, in order to process the stored per-language options. Each unknown key is saved as `\l__lualisthyphen_lang_name_str` and its value is used to set `lua-list-hyphen/per-lang`. The handling of this value takes place inside a group, so the variables defined for the options are set locally and passed to the Lua backend.

```
\str_new:N\l__lualisthyphen_lang_name_str
\keys_define:nn { lua-list-hyphen/per-lang-processing }{
  unknown .code:n = {
    \group_begin:

    \lua_now:e{ lualisthyphen.lang_options_check_name_known('\str_use:N\l_keys_key_str') }

    \str_set_eq:NN\l__lualisthyphen_lang_name_str\l_keys_key_str
    \keys_set:nn{ lua-list-hyphen/per-lang }{ #1 }

    \__lualisthyphen_lua_set_per_lang_options:e
      {'\str_use:N\l__lualisthyphen_lang_name_str'}

    \group_end:
  }
}
```

Use this new module to process the stored language-specific options.

```
\keys_set:ne{ lua-list-hyphen/per-lang-processing }
  { \clist_use:N\l__lualisthyphen_lang_options_clist }
```

10.6 Initialization

At `begindocument`, run initialization code.

```
\hook_gput_code:nnn{ begindocument }{ lua-list-hyphen } {
  \__lualisthyphen_initialize:
}
```

__lualisthyphen_initialize: Store babel's language names if it is in use. (When polyglossia is in use, the association of names and language IDs will be set on the fly.)

```
\cs_new:Npn \__lualisthyphen_initialize:
  {
    \__lualisthyphen_babel_store_lang_id_names:
  }
```

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

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

```
\cs_new:Npn \__lualisthyphen_babel_store_lang_id_names:
  {
    \cs_if_exist:NT\bbl@languages
      {
        \group_begin:
        \cs_set_eq:NN
          \bbl@elt
          \__lualisthyphen_babel_store_lang_id_names_elt:nnnn
        \bbl@languages
        \group_end:
      }%
  }
```

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

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

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

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

10.7 Processing and writing hyphenation lists

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

```
\hook_gput_code:nnn{ enddocument/info }{ lua-list-hyphen } {
  \lua_now:n{
    lualisthyphen.process_write_hyphenation_lists()
  }
}
```

⟨/package⟩

11 Implementation (Lua backend)

⟨*lua⟩

11.1 Initial set-up

Module identification/version information.

```
local module_name = 'lua-list-hyphen'
local lualisthyphen_module = {
  name        = module_name,
  version     = '0.43.2',
  date        = '2026-08-23',
  description = 'lua-list-hyphen',
  author      = 'Alan J. Cain',
  copyright   = 'Alan J. Cain',
  license     = 'LPPL v1.3c'
}
luatexbase.provides_module(lualisthyphen_module)
```

11.2 Message functions

info
warning

Functions for writing info or warning messages. First argument is used as a format string for later arguments.

```
local function info(s,...)
  luatexbase.module_info(module_name,s:format(...))
end
local function warning(s,...)
  luatexbase.module_warning(module_name,s:format(...))
end
local function error_(s,...)
  luatexbase.module_error(module_name,s:format(...))
end
```

(*End of definition for* `info` *and* `warning`.)

debug
debug_dummy
debug_real

Debugging function. `debug_dummy` and `debug_real` respectively do nothing and write debugging information. `debug_real`'s uses its parameters in the same way as `info` and `warning`. `debug` is set equal to the former and can be changed using `set_debug`.

```
local function debug_dummy(s,...)
end

local function debug_real(s,...)
  print(module_name .. ' DEBUG: ' .. s:format(...))
end

local debug = debug_dummy
local debug_bool = false
```

(*End of definition for* `debug`, `debug_dummy`, *and* `debug_real`.)

set_debug

If the parameter `a` is `true`, set `debug` to be `debug_real`, otherwise set it to `debug_dummy`.

```
local function set_debug(a)
  if a then
    debug = debug_real
```

```
  else
    debug = debug_dummy
  end
  debug_bool = a
end
```

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

11.3 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')
local NODE_ID_PENALTY = node.id('penalty')
```

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
```

11.4 List 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.)[9]

```
local function list_filter(t, f)
  local j = 1
  local n = #t

```

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

```
  for i=1,n do
    if (f(t[i])) then
      if (i ~= j) then
        t[j] = t[i]
        t[i] = nil
      end
      j = j + 1
    else
      t[i] = nil
    end
  end

end
```

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

`list_uniq` Take a list `t` and remove from it any element for which the function `f` returns `true` when called with the last 'kept' element (which always has index `j`) and the element being considered.

```
local function list_uniq(t, f)
  local j = 1
  local n = #t

  for i=2,n do
    if (f(t[i],t[j])) then
      t[i] = false
    else
      j = i
    end
  end

  list_filter(
    t,
    function(a) return a end
  )
end
```

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

11.5 String manipulation functions

String constants (which will ultimately be 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 = '"'
local STR_VALID = 'Valid'
local STR_INVALID = 'Inva.'
local STR_AMBIGUOUS = 'Ambi.'
local STR_UNKNOWN = 'Unkn.'
```

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

```
local function trim_whitespace_both(s)

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

end
```

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

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

```
local function trim_whitespace_left(s)

  return unicode.utf8.match(s,'^%s*(.*)$')

end
```

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

`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`.)

parenthesize Return string `s` enclosed in parentheses.

```
local function parenthesize(s)

  return '(' .. s .. ')'

end
```

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

11.6 Hyphen utility functions

delete_hyphens Delete all hyphens.

```
local function delete_hyphens(s)

  return unicode.utf8.gsub(s,'-','')

end
```

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

uniformize_hyphens Make all appearances of `|` and `¦` into hyphens.

```
local function uniformize_hyphens(s)

  return unicode.utf8.gsub(unicode.utf8.gsub(s,'|','-'),'¦','-')

end
```

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

hyphenation_matches_pattern Test that each hyphen in `hyphenation` also occurs in `pattern`.

```
local function hyphenation_matches_pattern(hyphenation,pattern)

  debug('    Testing "%s" against "%s"',hyphenation,pattern)
```

`i` is the 'current' position in `hyphenation`, just as `j` is the 'current' position in `pattern`. The 'current' characters are respectively `s` and `t`. If they are equal, both `i` and `j` are incremented. If they are unequal and `s` is a hyphen, then the there is a mismatch in the hyphenation. If they are unequal and `t` is not a hyphen, then there is a mismatch as words (which should never happen). Otherwise they are unequal and `t` is a hyphen that does not appear in `hyphenation`, and so only `j` must be incremented.

```
  local i = 1
  for j=1,#pattern do

    local s
    local t
```

```
448
449     s = string.sub(hyphenation,i,i)
450     t = string.sub(pattern,j,j)
451
452     if s == t then
453       i = i + 1
454     elseif s == '-' or t ~= '-' then
455       debug('      Fails at i=%d,s=%s,j=%d,t=%s',i,s,j,t)
456       return false
457     end
458
459   end
460
461   debug('      Matches')
462   return true
463
464 end
```

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

11.7 Language name tables and utility functions

The following tables map (respectively) each name for a language to its canonical name (or *cname*), and each cname to a set of names (members of the set map to `true`).

```
465 local lang_name_cname_table = {}
466 local lang_cname_name_set_table = {}
```

`lang_load_language_dat` Load language names from `language.dat`.

```
467 local LANGUAGE_DAT_LUA = 'language.dat.lua'
468
469 local function lang_load_language_dat_lua()
470   debug('  Loading names from %s',LANGUAGE_DAT_LUA)
471   for k,v in pairs(require(LANGUAGE_DAT_LUA)) do
472     local t = { [k] = true }
473     lang_cname_name_set_table[k] = t
474     lang_name_cname_table[k] = k
475     for _,vv in pairs(v.synonyms) do
476       lang_name_cname_table[vv] = k
477       t[vv] = true
478     end
479   end
480 end
```

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

`lang_load_language_dat` Load language names from `language.dat`.

```
481 local LANGUAGE_DAT = 'language.dat'
482
483 local function lang_load_language_dat()
484   debug('  Loading names from %s',LANGUAGE_DAT)
485
486   local p = kpse.find_file(LANGUAGE_DAT)
487 %    \en{macrocode}
488 %   If \pkg{kpathsea} could not locate the file, issue a warning and return immediately.
```

```
%    \begin{macrocode}
  if p == nil then
    warning('Could not find "%s"',LANGUAGE_DAT)
    return
  end

  local f = io.open(p,'r')
%    \en{macrocode}
%   If the file was not found, return immediately.
%    \begin{macrocode}
  if f == nil then
    warning('Could not read "%s"',p)
    return nil
  end

  io.input(f)

  local cname
  local cname_name_set
%    \en{macrocode}
%   Iterate through lines read from \file{language.dat}.
%    \begin{macrocode}
  for line in io.lines() do
%    \en{macrocode}
%   Blank lines can be ignored and lines in which the first character is \texttt{\%} are com
%    \begin{macrocode}
    if #line == 0 or string.sub(line,1,1) == '%' then
      goto continue
    end
%    \en{macrocode}
%   Lines in which the first character is \texttt{=} are synonyms.
%    \begin{macrocode}
    local name = string.match(line,'^=([^%s]+)')
    if name then
      if cname then
        if cname_name_set[name] then
          debug('    Known synonym "%s" found for "%s"',name,cname)
        else
          debug('    New synonym "%s" found for "%s"',name,cname)
          cname_name_set[name] = true
        end
      else
        warning('Problem in parsing "%s": the name "%s" is a synonym but no canonical name w
      end

      goto continue
    end
%    \en{macrocode}
%   Otherwise the line is the name of a language, a space, and the file containing hyphenati
%   Store the cname and corresponding set of synonyms and
%    \begin{macrocode}
    cname = string.match(line,'^([^%s]+)%s')
    cname_name_set = lang_cname_name_set_table[cname]
    if cname_name_set == nil then
```

```
543       debug('    New cname "%s" found',cname)
544       cname_name_set = { [cname] = true }
545       lang_cname_name_set_table[cname] = cname_name_set
546       lang_name_cname_table[cname] = cname
547     else
548       debug('    Known cname "%s" found',cname)
549     end
550
551     ::continue::
552   end
553
554   io.close(f)
555
556 end
```

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

`lang_load_names` Populate these tables from `language.dat.lua` (which means that they will contain languages not in use), then try `language.dat` for anything not listed in `language.dat.lua` (such as languages dumped into the format).

```
557 local function lang_load_names()
558   debug('Loading language names')
559   lang_load_language_dat_lua()
560   lang_load_language_dat()
```

If debugging, print the results.

```
561   if true then
562     debug('Language name -> cname:')
563     for k,v in pairs(lang_name_cname_table) do
564       debug('  %s -> %s',k,v)
565     end
566     debug('Language cname -> name list:')
567     for k,v in pairs(lang_cname_name_set_table) do
568       local s = ''
569       for kk,_ in pairs(v) do
570         s = s .. kk .. ','
571       end
572       debug('  %s -> %s ',k,s)
573     end
574   end
575 end
```

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

`get_cname` Get the cname of a language given one of its names. If the name doesn't appear in `lang_name_cname_table`, add the mapping name <-> cname.

```
576 local function get_cname(name)
577
578   local cname = lang_name_cname_table[name]
579   if cname == nil then
580     cname = name
581     lang_name_cname_table[name] = cname
582     lang_cname_name_set_table[cname] = { [name] = true }
583   end
584
```

```
  return cname

end
```

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

`lang_options_check_name_known` If `name` is not the name of a known language, emit a warning.

```
local function lang_options_check_name_known(name)

  if lang_name_cname_table[name] == nil then
    warning('Option(s) set for unknown language "%s"',name)
  end

end
```

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

The following tables map (respectively) each language ID to its cname and each cname to the language ID.

```
local lang_id_cname_table = {}
local lang_cname_id_table = {}
```

Populating these tables is done differently for babel and polyglossia. If babel is in use, the LaTeX frontend iterates through `\bbl@languages` and calls `store_lang_id_name`. for each item If polyglossia is in use, `lang_id_name_list_table` is populated on the fly.

`store_lang_id_name` Store the association of a language ID to the cname of the language with the given name (which may be the cname itself or a synonym).

```
local function store_lang_id_name(lang_id,name)

  local cname = get_cname(name)

  local lang_id_cname = lang_id_cname_table[lang_id]
  if lang_id_cname == cname then
    debug(
      'Language name "%s" is a synonym for cname "%s" (lang. ID %s)',
      name,cname,lang_id
    )
    return
  elseif lang_id_cname ~= nil then
    error_(
      'Languages with canonical names "%s" and "%s" both have ID %s',
      cname,lang_id_cname,lang_id
    )
    return
  end

  lang_id_cname_table[lang_id] = cname
  lang_cname_id_table[cname] = lang_id

  debug('Language ID %s has cname "%s"',lang_id,cname)

end
```

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

polyglossia_store_lang_id_name If polyglossia has been loaded, use it to store the association of the given language ID to the cname of the language.

```
local function polyglossia_store_lang_id_name(lang_id)

  if not polyglossia then
    return
  end

  for name,language in pairs(polyglossia.newloader_loaded_languages) do
    local this_lang_id = lang.id(language)
    if this_lang_id == lang_id then
      store_lang_id_name(lang_id,name)
    end
  end

end
```

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

11.8 Language options

The following table maps cnames to tables of options for languages. It will be populated via `set_lang_options` when the package options are processed.

```
local lang_cname_options_table = {}
```

The following table maps language IDs to tables of options for languages, and `LANG_DEFAULT` to a set of default options. The table of default options will be created and populated when package options are processed. The options for other languages will be added from `lang_cname_options_table` the first time an option for each language is needed, at which time the association of language IDs to cnames will be knowns.

```
local lang_id_options_table = {}
local LANG_DEFAULT = -1
```

get_lang_option Return the value associated to `key` for the language with the given ID, or value for `LANG_DEFAULT` if no per-language options have been specified for the given. This macro should only be called when `lang_id_cname_table` is known to have entry for `lang_id`, which means *after* `get_lang_division_dict_validation_preprocessor` has been called during post-linebreak processing.

```
local function get_lang_option(lang_id,key)
```

First look for the language's options table in `lang_id_options_table`. This will always be set after the first call to this function with this `lang_id`.

```
  local options = lang_id_options_table[lang_id]

  if not options then
```

If no options table was found, trying lookup a cname for the language and then looking in `lang_name_cname_table`, where options set from the LaTeX frontend will be found.

```
    local cname = lang_id_cname_table[lang_id]
    if cname then
      options = lang_cname_options_table[cname]
    end
```

If still no options table was found, make a copy of the default options table.

```
    if not options then
      options = {}
      for k,v in pairs(lang_id_options_table[LANG_DEFAULT]) do
        options[k] = v
      end
    end
    lang_id_options_table[lang_id] = options
  end

  return options[key]

end
```

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

`set_global_option` Set the global value of `key` to the given `value`. That is, set `lang_id_options_table[LANG_DEFAULT][key]` to `value`.

```
local function set_global_option(key,value)

  local options = lang_id_options_table[LANG_DEFAULT]
  if not options then
    options = {}
    lang_id_options_table[LANG_DEFAULT] = options
  end

  options[key] = value
  debug('Set global option %s="%s"',key,value)

end
```

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

`set_lang_option` Set the global value of `key` to the given `value`. That is, set `lang_cname_options_table[cname][key]` to `value`, where `cname` is the cname of the language with the given `name`.

```
local function set_lang_option(name,key,value)

  local cname = get_cname(name)

  local options = lang_cname_options_table[cname]
  if not options then
    options = {}
    lang_cname_options_table[cname] = options
  end

  options[key] = value
  debug('Set lang. "%s" option %s="%s"',cname,key,value)

end
```

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

11.9 Division dictionaries

The following table maps each language ID to a dictionary (possibly empty) of divisions for that language. Each such table maps a word to the same word with division points marked by `-`, `|`, or `¦`.

```
local lang_id_division_dict_table = {}
```

The following table maps each language ID to `nil`/`true`/`false` indicating, respectively, no attempt to read a division dictionary/read successfully/failed to read.

```
local lang_id_division_dict_read_table = {}
```

The following table maps language IDs to the full path (after kpathsea lookup) from which the division dictionary is read.

```
local lang_id_division_dict_path_table = {}
```

load_division_dict

Load the dictionary of valid divisions for a given language ID from a given path, if it points to a file. Return a table containing the division dictionary if the file exists and `nil` if it does not. Unless `case_sensitive` is `true`, convert the loaded divisions to lower case.

```
local function load_division_dict(path,case_sensitive)

  local count = 0

  local f = io.open(path,'r')
%    \en{macrocode}
%   If the file was not found, return immediately.
%    \begin{macrocode}
  if f == nil then
    return nil
  end

  debug('  Loading divisions from "%s"',path)

  local division_dict = {}

  io.input(f)

  for line in io.lines() do

    line = trim_whitespace_left(line)
%    \en{macrocode}
%   Lines in which the first non-whitespace character is \texttt{\%} are comments.
%    \begin{macrocode}
    if #line == 0 or string.sub(line,1,1) == '%' then
      goto continue
    end

    count = count + 1
```

The pattern is between the end of leading whitespace and the next piece of whitespace. (Anything after that can be a note etc.)

```
    local pattern = unicode.utf8.match(line,'^(.-)%s') or line

    pattern = uniformize_hyphens(pattern)
```

```
720
721     if not case_sensitive then
722       pattern = unicode.utf8.lower(pattern)
723     end
724
725     local word = delete_hyphens(pattern)
726
727     local divisions = division_dict[word]
728     if divisions == nil then
729       divisions = {}
730       division_dict[word] = divisions
731     end
732
733     table.insert(divisions,pattern)
734
735     ::continue::
736   end
737
738   io.close(f)
739
740   debug('Read %s divisions from "%s"',count,path)
741
742   return division_dict
743
744 end
```

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

11.10 Validation preprocessors

The following table maps a language ID to a functions that is applied to each hyphenated word found for that language, before it is validated. It will be populated when the first hyphenation in each language is encountered, at the same time as the division dictionary for that language is set.

```
745 local lang_id_validation_preprocessor_table = {}
```

Table to hold preprocessors. When options are processed, this table will be checked first for an the function, then the global environment.

```
746 local preprocessors = {}
```

`identity` A function just returning its parameter, used as a preprocessor when the user has not set one.

```
747 local function identity(s)
748   return s
749 end
750 preprocessors.identity = identity
```

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

`english_strip_possessives` A function removing a possessive 's from the end of a word.

```
751 local function english_strip_possessives(s)
752   return string.gsub(s,'’s$','')
753 end
754 preprocessors.english_strip_possessives = english_strip_possessives
```

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

11.11 Getting the division dictionary and validation preprocessor for a language

_lang_division_dict_validation_preprocessor Return the division dictionary and word preprocessor for a given language ID, loading the dictionary if necessary and caching the results. If no division dictionary has been specified for this language, this part of the result is empty dictionary. If no preprocessor has been specified for this languages, this part of the result is the identity function. In the case of polyglossia, also store the association of the language ID to the cname.

```
local function get_lang_division_dict_validation_preprocessor(lang_id)
```

If `lang_id_division_dict_table` already has an entry for the given language ID, just return it and the preprocessor. Both `lang_id_division_dict_table` and `lang_id_validation_preprocessor_table` are set in this function and will be 'in sync'.

```
  local division_dict = lang_id_division_dict_table[lang_id]
  if division_dict ~= nil then
    return division_dict,lang_id_validation_preprocessor_table[lang_id]
  end
```

Otherwise, it is necessary load a division dictionary. If polyglossia is in use, it is also first of all necessary to set the value in`lang_id_cname_list_table` for this language ID.

```
  polyglossia_store_lang_id_name(lang_id)
```

See if there is a user-specified word preprocessor. If not, use `identity`. In either case, store it it in `lang_id_validation_preprocessor_table`.

```
  local cname = lang_id_cname_table[lang_id]
  local preprocessor_name = get_lang_option(lang_id,'validation-preprocessor')
  local preprocessor
  if not preprocessor_name then
    debug(
      'No preprocessor specified for "%s"',cname
    )
    preprocessor = identity
  else
    preprocessor = preprocessors[preprocessor_name]
    if not preprocessor then
       preprocessor = _G[preprocessor_name]
    end
    if not preprocessor then
      debug(
        'Preprocessor specified for "%s" was not found',cname
      )
      preprocessor = identity
    else
      debug(
        'Using preprocessor specified for "%s"',cname
      )
    end
  end
  lang_id_validation_preprocessor_table[lang_id] = preprocessor
```

Similarly see if there is a user-specified path for the division dictionary. If so, look it up. If successful, load the division dictionary from it, save it in `lang_id_division_dict_table` and return it. If any of these conditions fail, save and return an empty dictionary.

```
  debug(
    'Attempting to load division dictionary for "%s" (lang. ID %s)',
    cname,lang_id
  )
  local file_name = get_lang_option(lang_id,'division-dict-path')
  if file_name then
    debug('  Filename: ' .. file_name)
    local p = kpse.find_file(file_name)
    if p then
      debug('  Path (after kpathsea lookup): %s',p)
      lang_id_division_dict_path_table[lang_id] = p

      local case_sensitive = get_lang_option(lang_id,'validation-case-sensitive')

      division_dict = load_division_dict(p,case_sensitive)
      if division_dict ~= nil then
        info(
          '  Loaded division dict for "%s" (lang. ID %s) from "%s"',
          cname,lang_id,path
        )
        lang_id_division_dict_read_table[lang_id] = true
        lang_id_division_dict_table[lang_id] = division_dict
        return division_dict,preprocessor
      end

    else
      warning(
        '  Specified division dict "%s" not found for "%s" (lang. ID %s)',
        file_name,cname,lang_id
      )
    end
  else
    debug('  No division dict path specified')
  end

  lang_id_division_dict_read_table[lang_id] = false
  division_dict = {}
  lang_id_division_dict_table[lang_id] = division_dict

  return division_dict,preprocessor

end
```

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

11.12 Validating a hyphenation

Constants for the status of a hyphenation

```
local STATUS_VALID = 1
local STATUS_INVALID = 2
local STATUS_AMBIGUOUS = 3
local STATUS_UNKNOWN = 4
```

validate_division Return the appropriate `STATUS_*` depending on whether `divided` matches the known pattern for `word` in the language with the given ID.

```
local function validate_division(lang_id,word,divided)

  debug(
    '    Validating "%s" for language ID %s',divided,lang_id
  )
  local division_dict,preprocessor
    = get_lang_division_dict_validation_preprocessor(lang_id)
```

Use the preprocessor on the original and divided word.

```
  word = preprocessor(word)
  divided = preprocessor(divided)

  word = delete_hyphens(word)

  if not get_lang_option(lang_id,'validation-case-sensitive') then
    word = unicode.utf8.lower(word)
    divided = unicode.utf8.lower(divided)
  end

  local divisions = division_dict[word]
  if not divisions then
    return STATUS_UNKNOWN
  end

  local count = 0

  for _,division in ipairs(divisions) do
    if hyphenation_matches_pattern(divided,division) then
      count = count + 1
    end
  end

  if count == 0 then
    return STATUS_INVALID
  elseif count == #divisions then
    return STATUS_VALID
  else
    return STATUS_AMBIGUOUS
  end

end
```

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

11.13 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.

```
local LIGATURE_TEXT = {
  [0xfb00] = 'ff',
  [0xfb01] = 'fi',
  [0xfb02] = 'fl',
```

```
  [0xfb03] = 'ffi',
  [0xfb04] = 'ffl',
  [0xfb05] = 'ft',
  [0xfb06] = 'st',
}
```

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
```

In looking up the text, things can go wrong (e.g. the glyph might not be part of the version of unicode than is supported by the underlying `slnunicode` library). So use `pcall` and return an empty string if there is an error.

```
      local ok,retval = pcall(
        function ()
          return utf8.char(tonumber(characters[n.char].tounicode,16))
        end
      )
      if ok then
        return retval
      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, as could a penalty, as could glue of width zero.

```
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
    or
    n.id == NODE_ID_PENALTY
    or
    (n.id == NODE_ID_GLUE and n.width == 0)
  )

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 and n.width ~= 0)
    or
    (n.id == NODE_ID_KERN and n.subtype == NODE_KERN_SUBTYPE_USERKERN and n.kern ~= 0)
  )

end
```

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

11.14 Getting language IDs from nodes

`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
  --   return lang
  -- end

  -- lang = get_first_glyph_lang(n.pre)
  -- if lang then
  --   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`.)

11.15 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 = {}
```

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

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

`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')

```

Per § 8.2 of the LuaTEX manual, there can be cases where `.prev` is invalid, so run `node.slide()` to set them correctly.

```
  node.slide(hlist_head)

  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)
      debug('  Math')
      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] = {
        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] = {
        type = SEGMENT_WORD,
        word = word,
        lang = lang,
      }

      debug('  Word "%s"',word)

      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
      debug('  Space')
      segment_count = segment_count + 1

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

      hlist_segment_list[segment_count] = {
        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`.)

11.16 Post-linebreak processing

11.16.1 Node 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 search up to at most `USED_DISC_SEARCH_LIMIT` nodes from the tail.

```
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

  return nil

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.

```
local function get_disc_word_start(hlist_head,n)

  local item = n

  while item do
    local prev = item.prev

    if not (prev and is_possible_word_node(prev)) then
      return item
    end

    item = prev
  end

  return nil
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.

```
local function get_next_hlist(n)

  local item = n.next

  while item do
    if item.id == NODE_ID_HLIST then
      return item
    end
    item = item.next
  end

  return nil

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.

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

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

```
  local word_start_node = nil

  for item in node.traverse(hlist_head) do

    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.type == SEGMENT_WORD then
      item = t.word
      word_count = word_count + 1
    elseif t.type == SEGMENT_SPACE then
      item = STR_SPACE
    elseif t.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`.)

11.16.2 Flags

Define constant PDF literal nodes for flags.

make_flag_node Return `pdf_literal` whatsit node containing `code` preceded by `context` (if non-`nil`) enclosed in a state save/restore.

```
local function make_flag_node(code,context)
  if context then
    code = context .. ' ' .. code
  end

  local n = node.new('whatsit','pdf_literal')
  n.data = 'q ' .. code .. ' Q'

  return n
```

```
1224
1225 end
```

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

Transformation to shift flags into the margin.

```
1226 local FLAG_MARGIN_SHIFT = '1 0 0 1 4 -1 cm'
```

Width of flags. (Must match the actual code for flags below.)

```
1227 local FLAG_WIDTH = tex.sp('6pt')
```

Code for flags.

```
1228 local FLAG_VALID_CODE
1229   = '0 0 6 10 re .1 .7 .1 rg f ' -- Green background
1230   .. '1 w 1 G ' -- Line width 1pt, colour white
1231   .. '1 5.5 m 2.33333 3 l 5 8 l S' -- "Tick"
1232 local FLAG_INVALID_CODE
1233   = '0 0 6 10 re .9 0 0 rg f ' -- Red background
1234   .. '1 w 1 G ' -- Line width 1pt, colour white
1235   .. '1 3 m 5 7 l 1 7 m 5 3 l S'
1236 local FLAG_AMBIGUOUS_CODE = (
1237   '0 0 6 10 re .1 .7 .7 rg f ' -- Cyan background
1238   .. '1 0 0 1 2.85 6.25 cm ' -- Shift
1239   .. '1 w 1 G ' -- Line width 1pt, colour white
1240   .. '-1.55885 .9 m -1 1.4 -.65 2 0 2 c 1.3 2 1.8 1.3 1.8 .5 c '
1241   .. '1.8 -.75 0 -1.25 0 -2.5 c 0 -2.75 l S ' -- Question mark body
1242   .. '1.4 w 1 J ' -- Line width 1.4pt, line cap round
1243   .. '0 -4.25 m 0 -4.25 l S' -- Question mark dot
1244 )
1245 local FLAG_UNKNOWN_CODE = (
1246   '0 0 6 10 re .9 .9 0 rg f ' -- Yellow background
1247   .. '1 0 0 1 3 5 cm ' -- Shift
1248   .. '.8 w 1 G ' -- Linec with .8pt, colour white
1249   .. '-1.41421 -1.41421 m 1.41421 1.41421 l S ' -- Stroke of 0
1250   .. '2 0 m 2 1.2 1.2 3 0 3 c -1.2 3 -2 1.2 -2 0 c -2 -1.2 -1.2 -3 0 -3 c '
1251   .. '1.2 -3 2 -1.2 2 0 c h S' -- Outside of 0
1252 )
```

Nodes made from the preceding code.

```
1253 local FLAG_VALID_NODE = make_flag_node(FLAG_VALID_CODE,FLAG_MARGIN_SHIFT)
1254 local FLAG_INVALID_NODE = make_flag_node(FLAG_INVALID_CODE,FLAG_MARGIN_SHIFT)
1255 local FLAG_AMBIGUOUS_NODE = make_flag_node(FLAG_AMBIGUOUS_CODE,FLAG_MARGIN_SHIFT)
1256 local FLAG_UNKNOWN_NODE = make_flag_node(FLAG_UNKNOWN_CODE,FLAG_MARGIN_SHIFT)
```

`insert_flag` Add a 'flag' after the node `current` in the list at `head`, with the type of the flag corresponding to `status`.

```
1257 local function insert_flag(head,current,status)
1258
1259   if status == STATUS_VALID then
1260     node.insert_after(head,current,node.copy(FLAG_VALID_NODE))
1261   elseif status == STATUS_UNKNOWN then
1262     node.insert_after(head,current,node.copy(FLAG_UNKNOWN_NODE))
1263   elseif  status == STATUS_INVALID then
1264     node.insert_after(head,current,node.copy(FLAG_INVALID_NODE))
1265   elseif  status == STATUS_AMBIGUOUS then
1266     node.insert_after(head,current,node.copy(FLAG_AMBIGUOUS_NODE))
```

```
  end

end
```

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

`write_flag` Generic function to write a flag to the current TEX list.

```
local function write_flag(code)

  node.write(make_flag_node(code,'1 0 0 1 0 -1 cm'))

  local kern_n = node.new('kern','userkern')
  kern_n.kern=FLAG_WIDTH
  node.write(kern_n)

end
```

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

`write_flag_valid` Write the valid flag to the current TEX list.

```
local function write_flag_valid()

  write_flag(FLAG_VALID_CODE)

end
```

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

`write_flag_invalid` Write the invalid flag to the current TEX list.

```
local function write_flag_invalid()

  write_flag(FLAG_INVALID_CODE)

end
```

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

`write_flag_ambiguous` Write the ambiguous flag to the current TEX list.

```
local function write_flag_ambiguous()

  write_flag(FLAG_AMBIGUOUS_CODE)

end
```

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

`write_flag_unknown` Write the unknown flag to the current TEX list.

```
local function write_flag_unknown()

  write_flag(FLAG_UNKNOWN_CODE)

end
```

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

11.16.3 Main list of hyphenations

Count and list for hyphenations. 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
```

11.16.4 Check single line hyphenation

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.type == SEGMENT_WORD)
  local word = t.word
  local lang = t.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))
```

Save the original word to be included in the context later.

```
local word_orig = word
```

Check if the division is valid/invalid/ambiguous/unknown. (No calls to `get_lang_option` should appear before this point in this function, since it is `validate_division` that ultimately sets the language–name association when polyglossia is in use.)

```
word = trim_nonlettershyphens_both(word)

local divided = pre .. post
if string.sub(post,1,1) == '-' and string.sub(pre,-1,-1) == '-' then
  divided = pre .. string.sub(post,2,-1)
  debug('  Hyphenated word with split hyphen found: %s -> %s (%s/%s)',word,divided,pre,pos
else
  debug('  Hyphenated word found: %s -> %s (%s/%s)',word,divided,pre,post)
end

local status = validate_division(lang,word,divided)
```

If specified, insert a flag depending on the flag mode for this language and the status.

```
local flag_mode = get_lang_option(lang,'flag-mode')
if flag_mode == 1 or (flag_mode == 2 and status ~= STATUS_VALID) then
  insert_flag(hlist.head,last_disc,status)
end
```

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

```
 local context =
  get_context(
    hyphenation_index,-1,get_lang_option(lang,'context-before')
  )
  .. word_orig ..
  get_context(
    hyphenation_index,1,get_lang_option(lang,'context-after')
  )
```

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] = {
  lang = lang,
  word = word,
  divided = divided,
  index = hyphenation_count,
  context = context,
  status = status,
}
```

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].page = page

end
```

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

11.16.5 Main post-linebreak function

`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')
```

Per § 8.2 of the LuaTEX manual, there can be cases where `.prev` is invalid, so run `node.slide()` on each line to set them correctly.

```
  for item in node.traverse(vlist_head) do
    if item.id == NODE_ID_HLIST then
      node.slide(item.head)
    end
  end
```

Now actually look for hyphenations.

```
  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. %d',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`.)

11.17 Add 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
)
```

11.18 Processing hyphenation lists

Before hyphenation lists are written out, they are filtered, deduplicated and/or sorted in accordance with the set options.

11.18.1 Comparisons and equality checks for stored hyphenations

equal_hyphenation_case_sensitive

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

```
local function equal_hyphenation_case_sensitive(a,b)
  return (
    a.word == b.word
    and
    a.divided == b.divided
  )
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.word) == unicode.utf8.lower(b.word)
    and
    unicode.utf8.lower(a.divided) == unicode.utf8.lower(b.divided)
  )
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.word < b.word
    or
    (
      a.word == b.word
```

```
      and
      a.divided < b.divided
    )
    or
    (
      a.word == b.word
      and
      a.divided == b.divided
      and
      a.index < b.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.word) < unicode.utf8.lower(b.word)
    or
    (
      unicode.utf8.lower(a.word) == unicode.utf8.lower(b.word)
      and
      unicode.utf8.lower(a.divided) < unicode.utf8.lower(b.divided)
    )
    or
    (
      unicode.utf8.lower(a.word) == unicode.utf8.lower(b.word)
      and
      unicode.utf8.lower(a.divided) < unicode.utf8.lower(b.divided)
      and
      a.index < b.index
    )
  )
end
```

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

11.18.2 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.index < b.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`.)

11.18.3 Deduplication

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`.)

11.18.4 Combined processing

process_lang_hyphenation_list

As specified in the options for the given language, filter, deduplicates, and sort `hyphenation_list`.

```
local function process_lang_hyphenation_list(lang_id,hyphenation_list)

  local output_non_typeset = get_lang_option(lang_id,'output-non-typeset')
  local include_valid = (get_lang_option(lang_id,'output-mode') == 0)

```

Filter the list to remove valid hyphenations, if required, and non-typeset hyphenations, if required.

```
  list_filter(
    hyphenation_list,
    function (a)
      return (
        (a.status ~= STATUS_VALID or include_valid)
        and
        (a.page or output_non_typeset)
      )
    end
  )
```

Deduplicate in accordance with the specified options for this language.

```
  local unique = get_lang_option(lang_id,'unique')
  if unique == 1 then
    deduplicate_hyphenation_list_case(hyphenation_list)
  elseif unique == 2 then
    deduplicate_hyphenation_list_nocase(hyphenation_list)
  end
```

Sort in accordance with the specified options for this language.

```
  local sort_mode = get_lang_option(lang_id,'sort')
  if sort_mode == 1 then
    sort_hyphenation_list_case(hyphenation_list)
  elseif sort_mode == 2 then
    sort_hyphenation_list_nocase(hyphenation_list)
  else
    sort_hyphenation_list_none(hyphenation_list)
  end

end
```

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

11.19 Writing

write_lang_hyphenation_list_standard

Write out just the divided 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.divided .. '\n')
    end
```

```
1546
1547   end
1548
1549 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. The parameter `show_status` is boolean and controls whether the status is written.

```
1550 local function write_lang_hyphenation_list_verbose(f,hyphenation_list,show_status)
```

Calculate the number of columns required for each output field.

```
1551   local cols_word = 0
1552   local cols_divided = 0
1553   local cols_page = 0
1554
1555   for _,h in pairs(hyphenation_list) do
1556
1557     cols_word = math.max(
1558       cols_word,
1559       unicode.utf8.len(h.word)
1560     )
1561     cols_divided = math.max(
1562       cols_divided,
1563       unicode.utf8.len(h.divided)
1564     )
1565     cols_page = math.max(
1566       cols_page,
1567       unicode.utf8.len(tostring(h.page))
1568     )
1569
1570   end
```

Adjust the number of columns required the page output, since there is a prefix and a 'no page' indicator to consider.

```
1571   cols_page = math.max(
1572     cols_page + unicode.utf8.len(STR_PAGE_PREFIX),
1573     unicode.utf8.len(STR_PAGE_NONE)
1574   )
1575
1576   local cols_status = math.max(
1577     unicode.utf8.len(STR_VALID),
1578     unicode.utf8.len(STR_INVALID),
1579     unicode.utf8.len(STR_AMBIGUOUS),
1580     unicode.utf8.len(STR_UNKNOWN)
1581   ) + 2 -- Add for parentheses
1582
1583   for i,v in ipairs(hyphenation_list) do
1584
1585     if v then
```

It is possible for the `page` key not to have been set, for instance if the hyphenation occured in a box that was never output, so prepare the string containing the page accordingly.

```
1586       local page = v.page
```

```
      if page then
        page = STR_PAGE_PREFIX .. page
      else
        page = STR_PAGE_NONE
      end
```

Set the string containing the status depending on `show_status`.

```
      local status
      if show_status then
        status = v.status
        if status == STATUS_VALID then
          status = STR_VALID
        elseif status == STATUS_AMBIGUOUS then
          status = STR_AMBIGUOUS
        elseif status == STATUS_INVALID then
          status = STR_INVALID
        else
          status = STR_UNKNOWN
        end
        status = rpad(parenthesize(status),cols_status) .. STR_SPACE_TWO
      else
        status = ''
      end
```

Write everything out on a single line.

```
      f:write(
        rpad(v.word,cols_word)
        .. STR_ARROW
        .. rpad(v.divided,cols_divided)
        .. STR_SPACE_TWO
        .. status
        .. lpad(page,cols_page)
        .. STR_SPACE
        .. STR_QUOTE_OPEN
        .. v.context
        .. STR_QUOTE_CLOSE
        .. '\n'
      )
    end

  end

end
```

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

`get_settings_desc` Compute a settings description to insert into file headers.

```
local function get_settings_desc(lang_id)

  local settings_desc

  if get_lang_option(lang_id,'output-verbose') then
    settings_desc = string.format(
      'verbose=true,context-before=%d,context-after=%s',
      get_lang_option(lang_id,'context-before'),
```

```
      get_lang_option(lang_id,'context-after')
    )
  else
    settings_desc = 'verbose=false'
  end

  local ALL_NONVALID = {
    [0] = 'all',
    [1] = 'non-valid',
  }
  settings_desc = settings_desc .. string.format(
    ',output-mode=%s',
    ALL_NONVALID[get_lang_option(lang_id,'output-mode')]
  )
  settings_desc = settings_desc .. string.format(
    ',include-non-typeset=%s',
    get_lang_option(lang_id,'output-non-typeset')
  )
  local NONE_CASE_NOCASE = {
    [0] = 'none',
    [1] = 'case',
    [2] = 'nocase'
  }
  settings_desc = settings_desc .. string.format(
    ',sort=%s,unique=%s',
    NONE_CASE_NOCASE[get_lang_option(lang_id,'sort')],
    NONE_CASE_NOCASE[get_lang_option(lang_id,'unique')]
  )

  return settings_desc

end
```

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

`prefix_output_directory` Return the given `path` with the output directory prefixed, if defined.

```
local function prefix_output_directory(path)

  local output_directory = status.output_directory
  if not output_directory then
    return path
  end

  if string.sub(output_directory,-1,-1) == '/' then
    return output_directory .. path
  else
    return output_directory .. '/' .. path
  end

end
```

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

`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 `output_prefix` and `output_ext`, using `widths`

for the ‘columns’ in verbose mode.

```
local function process_write_lang_hyphenation_list(
    lang_id,hyphenation_list
)
  process_lang_hyphenation_list(lang_id,hyphenation_list)

  local output_path
  local lang_desc

  local cname = lang_id_cname_table[lang_id]

  debug('Processing and writing hyphenation lists for "%s" (lang. ID %d)',cname,lang_id)

  if cname then
    output_path = get_lang_option(lang_id,'output-path')
    if not output_path then
      output_path = get_lang_option(LANG_DEFAULT,'output-prefix')
        .. cname .. get_lang_option(LANG_DEFAULT,'output-extension')
    end
    lang_desc = string.format('language "%s" (ID %s)',cname,lang_id)
  else
    output_path = get_lang_option(LANG_DEFAULT,'output-prefix')
      .. lang_id .. get_lang_option(LANG_DEFAULT,'output-extension')
    lang_desc = 'language with ID ' .. lang_id
  end

  output_path = prefix_output_directory(output_path)
  debug('Output path: "%s"',output_path)

  local output_header = get_lang_option(lang_id,'output-header')
  local output_verbose = get_lang_option(lang_id,'output-verbose')

  local division_dict_read = lang_id_division_dict_read_table[lang_id]

  local f = io.open(output_path,'w')

  if output_header then
    f:write('% This file was generated by lua-list-hyphen\n')
    f:write('% Hyphenations for ' ..  lang_desc .. '\n')
    f:write('% General settings: ' .. get_settings_desc(lang_id) .. '\n')
    local division_dict_path = lang_id_division_dict_path_table[lang_id]
    if division_dict_path and division_dict_read then
      f:write(
        '% Division dictionary read from: "' .. division_dict_path .. '"\n'
      )
    end
  end

  if output_verbose then
    write_lang_hyphenation_list_verbose(f,hyphenation_list,division_dict_read)
  else
    write_lang_hyphenation_list_standard(f,hyphenation_list,division_dict_read)
  end

```

```
  f:close()

end
```

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

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

```
local function process_write_hyphenation_lists()

  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).

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

    local lang = h.lang

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

    table.insert(t,h)

  end
```

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

```
  for lang_id,hyphenations in pairs(lang_hyphenation_table) do
    process_write_lang_hyphenation_list(lang_id,hyphenations)
  end

end
```

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

11.20 Export public functions

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

```
lualisthyphen = lualisthyphen or {}

lualisthyphen.lang_load_names = lang_load_names
lualisthyphen.store_lang_id_name = store_lang_id_name

lualisthyphen.set_debug = set_debug
lualisthyphen.set_global_option = set_global_option
lualisthyphen.set_lang_option = set_lang_option
lualisthyphen.lang_options_check_name_known = lang_options_check_name_known

lualisthyphen.preprocessors = preprocessors

lualisthyphen.set_hyphenation_page = set_hyphenation_page

lualisthyphen.process_write_hyphenation_lists = process_write_hyphenation_lists
```

Also make available some functions that are only used for the documentation:

```
lualisthyphen.write_flag_valid = write_flag_valid
lualisthyphen.write_flag_invalid = write_flag_invalid
lualisthyphen.write_flag_ambiguous = write_flag_ambiguous
lualisthyphen.write_flag_unknown = write_flag_unknown
```

⟨/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.