

lua-list-hyphen — Per-language checking and listing of hyphenated words for Lua^AT^EX^{*}

Alan J. Cain[†]

Released 2026-08-01

Abstract

This Lua^AT^EX 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,	change of masters. Thus
the vacancy of the throne	Augustus, after all his
is a moment big with	fairer prospects had been
danger and mischief. The	snatched from him by un-
Roman emperors, desirous	timely deaths, rested his
to spare the legions that	last hopes on Tiberius, ob-
interval of suspense, and	tained for his adopted son
the temptation of an irreg-	the censorial and tribuni-
ular choice, invested their	tian powers, and dictated
designed successor with	a law, by which the fu-
so large a share of pres-	ture prince was invested
ent power, as should en-	with an authority equal to
able him, after their de-	his own over the provinces
cease, to assume the re-	and the armies. Thus Ves-
mainder without suffering	pasian subdued the gener-
the empire to perceive the	ous 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 v0.43.1, last revised 2026-08-01.

[†]a.j.cain (AT) gmail.com

Contents

1	Introduction	3
2	Requirements	5
3	Installation	5
4	Getting started	5
5	Classifications of hyphenations	6
6	Output format	7
7	Package options	7
7.1	Output files	8
7.2	Division dictionary files	8
7.3	Hyphenation checking	8
7.4	Flags	9
7.5	Output filtering	9
7.6	Output format	10
7.7	Output sorting and deduplication	10
8	Usage notes	11
8.1	Languages	11
8.2	Limitations	11
8.2.1	Unicode	11
8.2.2	Association of language names and numerical LuaTeX IDs	11
9	Appendix: demonstration source	12
9.1	LuaLaTeX source	12
9.2	Division dictionary	13
10	Implementation (LaTeX package)	14
10.1	Initial set-up	14
10.2	Options	14
10.2.1	Global-only options	14
10.2.2	Per-language and global options	14
10.2.3	Per-language options	16
10.2.4	Warnings for per-language-only and global-only options	17
10.3	Deprecated options	17
10.3.1	Developer option	18
10.3.2	Processing per-language options	18
10.4	Lua backend	18
10.5	Processing package options	19
10.5.1	Interface for passing options to the Lua backend	19
10.5.2	Passing global options to the Lua backend	20
10.5.3	Passing per-language options to the Lua backend	20
10.6	Initialization	21
10.7	Processing and writing hyphenation lists	22

11	Implementation (Lua backend)	23
11.1	Initial set-up	23
11.2	Message functions	23
11.3	Node ID and subtype constants	24
11.4	List utility functions	24
11.5	String manipulation functions	25
11.6	Hyphen utility functions	27
11.7	Language name tables and utility functions	28
11.8	Language options	32
11.9	Division dictionaries	34
11.10	Validation preprocessors	35
11.11	Getting the division dictionary and validation preprocessor for a language	36
11.12	Validating a hyphenation	37
11.13	Getting text from nodes	38
11.14	Getting language IDs from nodes	41
11.15	Pre-linebreak processing	42
11.16	Post-linebreak processing	45
	11.16.1 Node processing	45
	11.16.2 Flags	47
	11.16.3 Main list of hyphenations	50
	11.16.4 Check single line hyphenation	50
	11.16.5 Main post-linebreak function	52
11.17	Add callbacks	53
11.18	Processing hyphenation lists	53
	11.18.1 Comparisons and equality checks for stored hyphenations	53
	11.18.2 Sorting	54
	11.18.3 Deduplication	55
	11.18.4 Combined processing	56
11.19	Writing	56
11.20	Export public functions	61
	Index	63

1 Introduction

$\text{\TeX}$'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, $\text{\TeX}$'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

¹For a description of the algorithm and its limitations, see Knuth's account in Appendix H of *The $\text{\TeX}$ book* (Addison-Wesley, 2021. ISBN: 978-0-201-13447-6)

²See 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 Lua $\text{\TeX}$, it is possible to extract the hyphenated words. Patrick Gundlach’s Lua $\text{\TeX}$ 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 Lua $\text{\TeX}$ 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 Lua $\text{\TeX}$ package, `lua-list-hyphen`, uses some ideas from `lua-check-hyphen` but was written from scratch to work with a modern Lua $\text{\TeX}$. 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 $\text{\LaTeX}$ 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.

³URL: <https://ctan.org/pkg/lua-check-hyphen>

⁴URL: <https://www.latex-project.org/lppl.txt>

⁵URL: <https://codeberg.org/ajcain/lua-list-hyphen>

⁶URL: <https://codeberg.org/ajcain/hyphenassist>.

⁷In particular, *Form & Number: A History of Mathematical Beauty*. URL: https://archive.org/details/cain_formandnumber_ebook_large.

⁸URL: <https://codeberg.org/ajcain/division-dict>

2 Requirements

lua-list-hyphen requires

- (1) Lua $\text{\LaTeX}$,
- (2) a recent $\text{\LaTeX}$ 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 Lua $\text{\LaTeX}$ 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 Lua $\text{\LaTeX}$ 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 Lua $\text{\LaTeX}$ 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 $\text{\TeX}$ `\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 file 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 overridden 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 (*Global-only options.*) These two keys are used for the fallback output file
output-extension 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 simply 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-sensitive` 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 affected 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 Integer options controlling how many words before (**context-before**) and after (**context-after**) the hyphenation are written as context when **output-verbose=true**.
context-before 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 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 set to one of the following three values:
none: Hyphenations appear in the same order as they 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 Lua_{TeX}’s built-in Unicode functions for pattern matching and converting between upper and lower case. These functions are based on the `slunICODE` 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 Lua_{TeX} 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 Lua_{TeX} 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 Lua_{TeX} 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 Lua^AT_EX 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 Lua^LA^TE^X 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 (L^AT_EX package)

```

1 <*package>
2 <@@=lualisthyphen>

```

10.1 Initial set-up

Package identification/version information.

```

3 \NeedsTeXFormat{LaTeX2e}[2020-02-02]
4 \ProvidesExplPackage{lua-list-hyphen}{2026-08-01}{0.43.1}
5 {Listing hyphenated words for LuaLaTeX}

```

Check that Lua_TE_X is in use.

```

6 \sys_if_engine luatex:F
7 {
8   \msg_new:nnn{ lua-list-hyphen }{ luatex_required }
9   { LuaLaTeX~required.~Package~loading~will~abort. }
10  \msg_critical:nn{ lua-list-hyphen }{ luatex_required }
11 }

```

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.

```

12 \keys_define:nn { lua-list-hyphen }{
13   output-prefix .str_set:N = \l__lualisthyphen_output_prefix_str,
14   output-prefix .initial:e = { \c_sys_jobname_str- },
15 }

```

(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.

```

16 \keys_define:nn { lua-list-hyphen }{
17   output-extension .str_set:N = \l__lualisthyphen_output_extension_str,
18   output-extension .initial:n = { .hyph },
19 }

```

(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.

```

20 \keys_define:nn { lua-list-hyphen/global-per-lang }{
21   division-dict-path .str_set:N = \l__lualisthyphen_division_dict_path_str,
22 }
23 \keys_define:nn { lua-list-hyphen/global-per-lang }{
24   allowlist-path .meta:n = { division-dict-path=#1 },
25 }

```

(End of definition for \l__lualisthyphen_division_dict_path_str.)

`\l__lualisthyphen_validation_preprocessor_str` String option to specify validation preprocessor.

```

26 \keys_define:nn { lua-list-hyphen/global-per-lang }{
27   validation-preprocessor .str_set:N = \l__lualisthyphen_validation_preprocessor_str,
28 }

```

	(End of definition for <code>\l__lualisthyphen_validation_preprocessor_str</code> .)
<code>\l__lualisthyphen_validation_case_sensitive_bool</code>	Boolean option to specify whether validation of divisions is case-sensitive. <pre> 29 \keys_define:nn { lua-list-hyphen/global-per-lang }{ 30 validation-case-sensitive .bool_set:N 31 = \l__lualisthyphen_validation_case_sensitive_bool 32 } </pre> (End of definition for <code>\l__lualisthyphen_validation_case_sensitive_bool</code>.)
<code>\l__lualisthyphen_flag_mode_int</code>	Choice option to specify whether flags are added to the output PDF file to indicate the classification of hyphenations. <pre> 33 \int_new:N\l__lualisthyphen_flag_mode_int 34 \keys_define:nn { lua-list-hyphen/global-per-lang }{ 35 flag-mode .choices:nn = { none, all, non-valid }{ 36 \int_set:Nn\l__lualisthyphen_flag_mode_int{ \l_keys_choice_int - 1 } 37 }, 38 } </pre> (End of definition for <code>\l__lualisthyphen_flag_mode_int</code>.)
<code>\l__lualisthyphen_output_mode_int</code>	Choice option to indicate which hyphenations should be listed in the output files. <pre> 39 \int_new:N\l__lualisthyphen_output_mode_int 40 \keys_define:nn { lua-list-hyphen/global-per-lang }{ 41 output-mode .choices:nn = { all, non-valid }{ 42 \int_set:Nn\l__lualisthyphen_output_mode_int{ \l_keys_choice_int - 1 } 43 }, 44 } </pre> (End of definition for <code>\l__lualisthyphen_output_mode_int</code>.)
<code>\l__lualisthyphen_output_non_typeset_bool</code>	Boolean option to indicate whether output lists of hyphenations should include those that are never typeset to the page. <pre> 45 \keys_define:nn { lua-list-hyphen/global-per-lang }{ 46 output-non-typeset .bool_set:N = \l__lualisthyphen_output_non_typeset_bool, 47 } </pre> (End of definition for <code>\l__lualisthyphen_output_non_typeset_bool</code>.)
<code>\l__lualisthyphen_output_header_bool</code>	Boolean option to indicate whether a header should be written in the output. <pre> 48 \keys_define:nn { lua-list-hyphen/global-per-lang }{ 49 output-header .bool_set:N = \l__lualisthyphen_output_header_bool, 50 output-header .initial:n = {true}, 51 } </pre> (End of definition for <code>\l__lualisthyphen_output_header_bool</code>.)
<code>\l__lualisthyphen_output_verbose_bool</code>	Boolean option to indicate whether output lists of hyphenations should be written verbosely. <pre> 52 \keys_define:nn { lua-list-hyphen/global-per-lang }{ 53 output-verbose .bool_set:N = \l__lualisthyphen_output_verbose_bool, 54 } </pre> (End of definition for <code>\l__lualisthyphen_output_verbose_bool</code>.)

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

```

55 \keys_define:nn { lua-list-hyphen/global-per-lang }{
56   context-before .int_set:N = \l__lualisthyphen_context_before_int,
57   context-before .initial:n = { 2 },
58   context-after .int_set:N = \l__lualisthyphen_context_after_int,
59   context-after .initial:n = { 2 },
60   context .meta:n = {
61     context-before=#1,
62     context-after=#1,
63   },
64 }

```

(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.

```

65 \int_new:N\l__lualisthyphen_unique_int
66 \keys_define:nn { lua-list-hyphen/global-per-lang }{
67   unique .choices:nn = { none, case, nocase }{
68     \int_set:Nn\l__lualisthyphen_unique_int{ \l_keys_choice_int - 1 }
69   },
70 }

```

(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.

```

71 \int_new:N\l__lualisthyphen_sort_int
72 \keys_define:nn { lua-list-hyphen/global-per-lang }{
73   sort .choices:nn = { none, case, nocase }{
74     \int_set:Nn\l__lualisthyphen_sort_int{ \l_keys_choice_int - 1 }
75   },
76 }

```

(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.

```

77 \keys_define:nn { }{
78   lua-list-hyphen .inherit:n = lua-list-hyphen/global-per-lang,
79 }

```

10.2.3 Per-language options

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

```

80 \keys_define:nn { lua-list-hyphen }{
81   per-lang .inherit:n = lua-list-hyphen/global-per-lang,
82 }

```

There is one option that is strictly per-language:


```

\l__lualisthyphen_output_path_str Option to specify the path to write hyphenations.
83 \keys_define:nn { lua-list-hyphen/per-lang }{
84   output-path .str_set:N = \l__lualisthyphen_output_path_str,
85 }

```

(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.

```

86 \msg_new:nnnn{ lua-list-hyphen }{ per-language-only }
87 { The~option~"#1"~only~has~effect~per~language. }
88 { The~option~"#1"~has~been~specified~globally~and~will~be~ignored. }
89 \keys_define:nn { lua-list-hyphen }{
90   output-path .code:n = {
91     \msg_warning:nne{ lua-list-hyphen }{ per-language-only }
92     { \str_use:N\l_keys_key_str }
93   },
94 }

```

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

```

95 \msg_new:nnnn{ lua-list-hyphen }{ global-only }
96 { The~option~"#1"~only~has~effect~globally. }
97 { The~option~"#1"~has~been~specified~for~a~particular~language~and~will~be~ignored. }
98 \keys_define:nn { lua-list-hyphen/per-lang }{
99   output-prefix .code:n = {
100     \msg_warning:nne{ lua-list-hyphen }{ global-only }
101     { \str_use:N\l_keys_key_str }
102   },
103   output-extension .code:n = {
104     \msg_warning:nne{ lua-list-hyphen }{ global-only }
105     { \str_use:N\l_keys_key_str }
106   },
107 }

```

10.3 Deprecated options

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

```

108 \msg_new:nnnn{ lua-list-hyphen }{ deprecated-option }
109 { The~option~"#1"~works~but~is~deprecated.~Please~use~"#2"~instead. }
110 { The~option~"#1"~may~be~removed~in~a~future~version. }
111 \keys_define:nn { lua-list-hyphen }{
112   verbose .code:n = {
113     \msg_warning:nnee{ lua-list-hyphen }{ deprecated-option }
114     { \str_use:N\l_keys_key_str }{ output-verbose }
115     \keys_set:nn{ lua-list-hyphen }{ output-verbose=#1 }
116   },
117   include-non-output .code:n = {
118     \msg_warning:nnee{ lua-list-hyphen }{ deprecated-option }
119     { \str_use:N\l_keys_key_str }{ include-non-typeset }
120     \keys_set:nn{ lua-list-hyphen }{ include-non-typeset=#1 }
121   },
122   prefix .code:n = {

```

```

123     \msg_warning:nnee{ lua-list-hyphen }{ deprecated-option }
124     { \str_use:N\l_keys_key_str }{ output-prefix }
125     \keys_set:nn{ lua-list-hyphen }{ output-prefix=#1 }
126 },
127 extension .code:n = {
128     \msg_warning:nnee{ lua-list-hyphen }{ deprecated-option }
129     { \str_use:N\l_keys_key_str }{ output-extension }
130     \keys_set:nn{ lua-list-hyphen }{ output-extension=#1 }
131 }
132 }

```

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.

```

133 \keys_define:nn { lua-list-hyphen }{
134     debug .bool_set:N = \l__lualisthyphen_debug_bool,
135 }

```

(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.

```

136 \keys_define:nn { lua-list-hyphen }{
137     unknown .code:n = {
138         \__lualisthyphen_store_lang_options:en{\str_use:N\l_keys_key_str}{#1}
139     }
140 }

```

`\__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`.

```

141 \clist_new:N\l__lualisthyphen_lang_options_clist
142
143 \cs_new:Npn \__lualisthyphen_store_lang_options:nn #1#2
144 {
145     \clist_put_right:Nn\l__lualisthyphen_lang_options_clist{#1={#2}}
146 }
147 \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.

```

148 \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`.

```
149 \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.

```
150 \cs_new:Npn \__lualisthyphen_call_lua_boolean:nN #1#2
151 {
152   \lua_now:ef lualisthyphen.#1(\bool_to_str:N #2) }
153 }
```

(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.

```
154 \cs_new:Npn \__lualisthyphen_luastr_or_nil:N #1
155 {
156   \str_if_empty:NTF #1
157     { nil }
158     { '\str_use:N #1' }
159 }
```

(End of definition for __lualisthyphen_luastr_or_nil:N.)

`\__lualisthyphen_lua_set_global_option:nn` 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.

```
160 \cs_new:Npn \__lualisthyphen_lua_set_global_option:nn #1#2
161 {
162   \lua_now:n{ lualisthyphen.set_global_option('#1',#2) }
163 }
164 \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` 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.

```
165 \cs_new:Npn \__lualisthyphen_lua_set_lang_option:nnn #1#2#3
166 {
167   \lua_now:n{ lualisthyphen.set_lang_option(#1,'#2',#3) }
168 }
169 \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.

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

At this point the variables contain ‘global’ options.

```
172 \__lualisthyphen_lua_set_global_option:ne{division-dict-path}
173   {\__lualisthyphen_luastr_or_nil:N\l__lualisthyphen_division_dict_path_str}
174 \__lualisthyphen_lua_set_global_option:ne{output-prefix}
175   {\__lualisthyphen_luastr_or_nil:N\l__lualisthyphen_output_prefix_str}
176 \__lualisthyphen_lua_set_global_option:ne{output-extension}
177   {\__lualisthyphen_luastr_or_nil:N\l__lualisthyphen_output_extension_str}
178 \__lualisthyphen_lua_set_global_option:ne{validation-preprocessor}
179   {\__lualisthyphen_luastr_or_nil:N\l__lualisthyphen_validation_preprocessor_str}
180 \__lualisthyphen_lua_set_global_option:ne{validation-case-sensitive}
181   {\bool_to_str:N\l__lualisthyphen_validation_case_sensitive_bool}
182 \__lualisthyphen_lua_set_global_option:ne{flag-mode}
183   {\int_use:N\l__lualisthyphen_flag_mode_int}
184 \__lualisthyphen_lua_set_global_option:ne{output-mode}
185   {\int_use:N\l__lualisthyphen_output_mode_int}
186 \__lualisthyphen_lua_set_global_option:ne{output-non-typeset}
187   {\bool_to_str:N\l__lualisthyphen_output_non_typeset_bool}
188 \__lualisthyphen_lua_set_global_option:ne{output-header}
189   {\bool_to_str:N\l__lualisthyphen_output_header_bool}
190 \__lualisthyphen_lua_set_global_option:ne{output-verbose}
191   {\bool_to_str:N\l__lualisthyphen_output_verbose_bool}
192 \__lualisthyphen_lua_set_global_option:ne{context-before}
193   {\int_use:N\l__lualisthyphen_context_before_int}
194 \__lualisthyphen_lua_set_global_option:ne{context-after}
195   {\int_use:N\l__lualisthyphen_context_after_int}
196 \__lualisthyphen_lua_set_global_option:ne{unique}
197   {\int_use:N\l__lualisthyphen_unique_int}
198 \__lualisthyphen_lua_set_global_option:ne{sort}
199   {\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.

```
200 \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.

```
201 \cs_new:Npn \__lualisthyphen_lua_set_per_lang_options:n #1
202   {
203     \__lualisthyphen_lua_set_lang_option:nne{#1}{division-dict-path}
204       {\__lualisthyphen_luastr_or_nil:N\l__lualisthyphen_division_dict_path_str}
205     \__lualisthyphen_lua_set_lang_option:nne{#1}{output-path}
206       {\__lualisthyphen_luastr_or_nil:N\l__lualisthyphen_output_path_str}
207     \__lualisthyphen_lua_set_lang_option:nne{#1}{validation-preprocessor}
208       {\__lualisthyphen_luastr_or_nil:N\l__lualisthyphen_validation_preprocessor_str}
209     \__lualisthyphen_lua_set_lang_option:nne{#1}{validation-case-sensitive}
210       {\bool_to_str:N\l__lualisthyphen_validation_case_sensitive_bool}
211     \__lualisthyphen_lua_set_lang_option:nne{#1}{flag-mode}
```

```

212     {\int_use:N\l__lualisthyphen_flag_mode_int}
213 \__lualisthyphen_lua_set_lang_option:nne{#1}{output-mode}
214     {\int_use:N\l__lualisthyphen_output_mode_int}
215 \__lualisthyphen_lua_set_lang_option:nne{#1}{output-non-typeset}
216     {\bool_to_str:N\l__lualisthyphen_output_non_typeset_bool}
217 \__lualisthyphen_lua_set_lang_option:nne{#1}{output-header}
218     {\bool_to_str:N\l__lualisthyphen_output_header_bool}
219 \__lualisthyphen_lua_set_lang_option:nne{#1}{output-verbose}
220     {\bool_to_str:N\l__lualisthyphen_output_verbose_bool}
221 \__lualisthyphen_lua_set_lang_option:nne{#1}{context-before}
222     {\int_use:N\l__lualisthyphen_context_before_int}
223 \__lualisthyphen_lua_set_lang_option:nne{#1}{context-after}
224     {\int_use:N\l__lualisthyphen_context_after_int}
225 \__lualisthyphen_lua_set_lang_option:nne{#1}{unique}
226     {\int_use:N\l__lualisthyphen_unique_int}
227 \__lualisthyphen_lua_set_lang_option:nne{#1}{sort}
228     {\int_use:N\l__lualisthyphen_sort_int}
229 }
230 \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.

```

231 \str_new:N\l__lualisthyphen_lang_name_str
232 \keys_define:nn { lua-list-hyphen/per-lang-processing }{
233   unknown .code:n = {
234     \group_begin:
235
236     \lua_now:e{ lualisthyphen.lang_options_check_name_known('\str_use:N\l_keys_key_str') }
237
238     \str_set_eq:NN\l__lualisthyphen_lang_name_str\l_keys_key_str
239     \keys_set:nn{ lua-list-hyphen/per-lang }{ #1 }
240
241     \__lualisthyphen_lua_set_per_lang_options:e
242     {'\str_use:N\l__lualisthyphen_lang_name_str'}
243
244     \group_end:
245   }
246 }

```

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

```

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

```

10.6 Initialization

At `begindocument`, run initialization code.

```

249 \hook_gput_code:nnn{ begindocument }{ lua-list-hyphen }{
250   \__lualisthyphen_initialize:
251 }

```

`\__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.)

```

252 \cs_new:Npn \__lualisthyphen_initialize:
253 {
254   \__lualisthyphen_babel_store_lang_id_names:
255 }

```

(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.

```

256 \cs_new:Npn \__lualisthyphen_babel_store_lang_id_names:
257 {
258   \cs_if_exist:NT\bbl@languages
259   {
260     \group_begin:
261     \cs_set_eq:NN
262       \bbl@elt
263       \__lualisthyphen_babel_store_lang_id_names_elt:nmmn
264     \bbl@languages
265     \group_end:
266   }%
267 }

```

(End of definition for __lualisthyphen_babel_store_lang_id_names:.)

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

```

268 \cs_new:Npn \__lualisthyphen_babel_store_lang_id_names_elt:nmmn #1#2#3#4
269 {
270   \lua_now:n{
271     lualisthyphen.store_lang_id_name(#2,'#1')
272   }
273 }

```

(End of definition for __lualisthyphen_babel_store_lang_id_names_elt:nmmn.)

10.7 Processing and writing hyphenation lists

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

```

274 \hook_gput_code:nnn{ enddocument/info }{ lua-list-hyphen } {
275   \lua_now:n{
276     lualisthyphen.process_write_hyphenation_lists()
277   }
278 }
279 </package>

```

11 Implementation (Lua backend)

280 $\langle *lua \rangle$

11.1 Initial set-up

Module identification/version information.

```
281 local module_name = 'lua-list-hyphen'
282 local lualisthyphen_module = {
283     name      = module_name,
284     version   = '0.43.1',
285     date      = '2026-08-01',
286     description = 'lua-list-hyphen',
287     author    = 'Alan J. Cain',
288     copyright  = 'Alan J. Cain',
289     license    = 'LPPL v1.3c'
290 }
291 luatexbase.provides_module(lualisthyphen_module)
```

11.2 Message functions

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

```
292 local function info(s,...)
293     luatexbase.module_info(module_name,s:format(...))
294 end
295 local function warning(s,...)
296     luatexbase.module_warning(module_name,s:format(...))
297 end
298 local function error_(s,...)
299     luatexbase.module_error(module_name,s:format(...))
300 end
```

(End of definition for info and warning.)

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

```
301 local function debug_dummy(s,...)
302     end
303
304 local function debug_real(s,...)
305     print(module_name .. ' DEBUG: ' .. s:format(...))
306 end
307
308 local debug = debug_dummy
309 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**.

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

```

313     else
314         debug = debug_dummy
315     end
316     debug_bool = a
317 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.

```

318 local NODE_ID_HLIST = node.id('hlist')
319 local NODE_ID_DISC = node.id('disc')
320 local NODE_ID_GLUE = node.id('glue')
321 local NODE_ID_KERN = node.id('kern')
322 local NODE_ID_MARGIN_KERN = node.id('margin_kern')
323 local NODE_ID_GLYPH = node.id('glyph')
324 local NODE_ID_MATH = node.id('math')
325 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.)

```

326 local NODE_KERN_SUBTYPE_FONTKERN
327 local NODE_KERN_SUBTYPE_USERKERN
328 for k,v in pairs(node.subtypes('kern')) do
329     if v == 'fontkern' then
330         NODE_KERN_SUBTYPE_FONTKERN = k
331     elseif v == 'userkern' then
332         NODE_KERN_SUBTYPE_USERKERN = k
333     end
334 end

```

Define constants for the math node subtypes.

```

335 local NODE_MATH_SUBTYPE_BEGIN
336 local NODE_MATH_SUBTYPE_END
337 for k,v in pairs(node.subtypes('math')) do
338     if v == 'beginmath' then
339         NODE_MATH_SUBTYPE_BEGIN = k
340     elseif v == 'endmath' then
341         NODE_MATH_SUBTYPE_END = k
342     end
343 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

```

344 local function list_filter(t, f)
345     local j = 1
346     local n = #t
347

```

⁹Code adapted from <https://stackoverflow.com/a/53038524>.


```

348   for i=1,n do
349       if (f(t[i])) then
350           if (i ~= j) then
351               t[j] = t[i]
352               t[i] = nil
353           end
354           j = j + 1
355       else
356           t[i] = nil
357       end
358   end
359
360 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.

```

361 local function list_uniq(t, f)
362     local j = 1
363     local n = #t
364
365     for i=2,n do
366         if (f(t[i],t[j])) then
367             t[i] = nil
368         else
369             j = i
370         end
371     end
372
373     list_filter(
374         t,
375         function(a) return a end
376     )
377 end

```

(End of definition for list_uniq.)

11.5 String manipulation functions

String constants (which will ultimately be output).

```

378 local STR_MATH = '[MATH]'
379 local STR_SPACE = ' '
380 local STR_SPACE_TWO = '  '
381 local STR_ARROW = ' -> '
382 local STR_PAGE_PREFIX = 'p.'
383 local STR_PAGE_NONE = '<none>'
384 local STR_QUOTE_OPEN = '"'
385 local STR_QUOTE_CLOSE = '"'
386 local STR_VALID = 'Valid'
387 local STR_INVALID = 'Inva.'
388 local STR_AMBIGUOUS = 'Ambi.'
389 local STR_UNKNOWN = 'Unkn.'

```

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

```
390 local function trim_whitespace_both(s)
391
392   return unicode.utf8.match(s, '^%s*(.-%s*$')
393
394 end
```

(End of definition for trim_whitespace_both.)

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

```
395 local function trim_whitespace_left(s)
396
397   return unicode.utf8.match(s, '^%s*(.*)$')
398
399 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.

```
400 local function trim_nonlettershyphens_both(s)
401
402   return unicode.utf8.match(s, '^[%a-]*(.-%[%a-]*$')
403
404 end
```

(End of definition for trim_nonlettershyphens_both.)

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

```
405 local function trim_nonlettershyphens_start(s)
406
407   return unicode.utf8.match(s, '^[%a-]*(.-%$')
408
409 end
```

(End of definition for trim_nonlettershyphens_start.)

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

```
410 local function trim_nonlettershyphens_end(s)
411
412   return unicode.utf8.match(s, '^(.-%[%a-]*$')
413
414 end
```

(End of definition for trim_nonlettershyphens_end.)

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

```
415 local function rpadd(s,n)
416
417   return s .. unicode.utf8.rep(STR_SPACE,n - unicode.utf8.len(s))
418
419 end
```

(End of definition for rpadd.)

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

```
420 local function lpad(s,n)
421
422     return unicode.utf8.rep(STR_SPACE,n - unicode.utf8.len(s)) .. s
423
424 end
```

(End of definition for lpad.)

`parenthesize` Return string `s` enclosed in parentheses.

```
425 local function parenthesize(s)
426
427     return '(' .. s .. ')'
428
429 end
```

(End of definition for parenthesize.)

11.6 Hyphen utility functions

`delete_hyphens` Delete all hyphens.

```
430 local function delete_hyphens(s)
431
432     return unicode.utf8.gsub(s,'-','')
433
434 end
```

(End of definition for delete_hyphens.)

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

```
435 local function uniformize_hyphens(s)
436
437     return unicode.utf8.gsub(unicode.utf8.gsub(s,'|','-'),'!','-')
438
439 end
```

(End of definition for uniformize_hyphens.)

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

```
440 local function hyphenation_matches_pattern(hyphenation,pattern)
441
442     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 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.

```
443     local i = 1
444     for j=1,#pattern do
445
446         local s
447         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.

```

```

489 % \begin{macrocode}
490 if p == nil then
491     warning('Could not find "%s"',LANGUAGE_DAT)
492     return
493 end
494
495 local f = io.open(p,'r')
496 % \end{macrocode}
497 % If the file was not found, return immediately.
498 % \begin{macrocode}
499 if f == nil then
500     warning('Could not read "%s"',p)
501     return nil
502 end
503
504 io.input(f)
505
506 local cname
507 local cname_name_set
508 % \end{macrocode}
509 % Iterate through lines read from \file{language.dat}.
510 % \begin{macrocode}
511 for line in io.lines() do
512 % \end{macrocode}
513 % Blank lines can be ignored and lines in which the first character is \texttt{\%} are com
514 % \begin{macrocode}
515 if #line == 0 or string.sub(line,1,1) == '%' then
516     goto continue
517 end
518 % \end{macrocode}
519 % Lines in which the first character is \texttt{=} are synonyms.
520 % \begin{macrocode}
521 local name = string.match(line,'^([~%s]+)')
522 if name then
523     if cname then
524         if cname_name_set[name] then
525             debug('    Known synonym "%s" found for "%s"',name,cname)
526         else
527             debug('    New synonym "%s" found for "%s"',name,cname)
528             cname_name_set[name] = true
529         end
530     else
531         warning('Problem in parsing "%s": the name "%s" is a synonym but no canonical name w
532     end
533
534     goto continue
535 end
536 % \end{macrocode}
537 % Otherwise the line is the name of a language, a space, and the file containing hyphenati
538 % Store the cname and corresponding set of synonyms and
539 % \begin{macrocode}
540 cname = string.match(line,'^([~%s]+)%s')
541 cname_name_set = lang_cname_name_set_table[cname]
542 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

```

```

585     return cname
586
587 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.

```

588 local function lang_options_check_name_known(name)
589
590     if lang_name_cname_table[name] == nil then
591         warning('Option(s) set for unknown language "%s"',name)
592     end
593
594 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.

```

595 local lang_id_cname_table = {}
596 local lang_cname_id_table = {}

```

Populating these tables is done differently for babel and polyglossia. If babel is in use, the L^AT_EX 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).

```

597 local function store_lang_id_name(lang_id,name)
598
599     local cname = get_cname(name)
600
601     local lang_id_cname = lang_id_cname_table[lang_id]
602     if lang_id_cname == cname then
603         debug(
604             'Language name "%s" is a synonym for cname "%s" (lang. ID %s)',
605             name,cname,lang_id
606         )
607         return
608     elseif lang_id_cname ~= nil then
609         error_(
610             'Languages with canonical names "%s" and "%s" both have ID %s',
611             cname,lang_id_cname,lang_id
612         )
613         return
614     end
615
616     lang_id_cname_table[lang_id] = cname
617     lang_cname_id_table[cname] = lang_id
618
619     debug('Language ID %s has cname "%s"',lang_id,cname)
620
621 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.

```

622 local function polyglossia_store_lang_id_name(lang_id)
623
624   if not polyglossia then
625     return
626   end
627
628   for name,language in pairs(polyglossia.newloader_loaded_languages) do
629     local this_lang_id = lang.id(language)
630     if this_lang_id == lang_id then
631       store_lang_id_name(lang_id,name)
632     end
633   end
634
635 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.

```

636 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 known.

```

637 local lang_id_options_table = {}
638 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.

```

639 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`.

```

640   local options = lang_id_options_table[lang_id]
641
642   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 L^AT_EX frontend will be found.

```

643     local cname = lang_id_cname_table[lang_id]
644     if cname then
645       options = lang_cname_options_table[cname]
646     end

```


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

```
647     if not options then
648         options = {}
649         for k,v in pairs(lang_id_options_table[LANG_DEFAULT]) do
650             options[k] = v
651         end
652     end
653     lang_id_options_table[lang_id] = options
654 end
655
656 return options[key]
657
658 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.

```
659 local function set_global_option(key,value)
660
661     local options = lang_id_options_table[LANG_DEFAULT]
662     if not options then
663         options = {}
664         lang_id_options_table[LANG_DEFAULT] = options
665     end
666
667     options[key] = value
668     debug('Set global option %s="%s"',key,value)
669
670 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.

```
671 local function set_lang_option(name,key,value)
672
673     local cname = get_cname(name)
674
675     local options = lang_cname_options_table[cname]
676     if not options then
677         options = {}
678         lang_cname_options_table[cname] = options
679     end
680
681     options[key] = value
682     debug('Set lang. "%s" option %s="%s"',cname,key,value)
683
684 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 !.

```
685 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.

```
686 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.

```
687 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.

```
688 local function load_division_dict(path,case_sensitive)
689
690     local count = 0
691
692     local f = io.open(path,'r')
693     % \en{macrocode}
694     % If the file was not found, return immediately.
695     % \begin{macrocode}
696     if f == nil then
697         return nil
698     end
699
700     debug(' Loading divisions from "%s"',path)
701
702     local division_dict = {}
703
704     io.input(f)
705
706     for line in io.lines() do
707
708         line = trim_whitespace_left(line)
709         % \en{macrocode}
710         % Lines in which the first non-whitespace character is \texttt{\%} are comments.
711         % \begin{macrocode}
712         if #line == 0 or string.sub(line,1,1) == '%' then
713             goto continue
714         end
715
716         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.)

```
717     local pattern = unicode.utf8.match(line,'^(.)%s') or line
718
719     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.

```
755 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'.

```
756   local division_dict = lang_id_division_dict_table[lang_id]
757   if division_dict ~= nil then
758     return division_dict, lang_id_validation_preprocessor_table[lang_id]
759   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.

```
760   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 in `lang_id_validation_preprocessor_table`.

```
761   local cname = lang_id_cname_table[lang_id]
762   local preprocessor_name = get_lang_option(lang_id, 'validation-preprocessor')
763   local preprocessor
764   if not preprocessor_name then
765     debug(
766       'No preprocessor specified for "%s"', cname
767     )
768     preprocessor = identity
769   else
770     preprocessor = preprocessors[preprocessor_name]
771     if not preprocessor then
772       preprocessor = _G[preprocessor_name]
773     end
774     if not preprocessor then
775       debug(
776         'Preprocessor specified for "%s" was not found', cname
777       )
778       preprocessor = identity
779     else
780       debug(
781         'Using preprocessor specified for "%s"', cname
782       )
783     end
784   end
785   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.

```

786 debug(
787     'Attempting to load division dictionary for "%s" (lang. ID %s)',
788     cname,lang_id
789 )
790 local file_name = get_lang_option(lang_id,'division-dict-path')
791 if file_name then
792     debug('  Filename: ' .. file_name)
793     local p = kpse.find_file(file_name)
794     if p then
795         debug('  Path (after kpathsea lookup): %s',p)
796         lang_id_division_dict_path_table[lang_id] = p
797
798         local case_sensitive = get_lang_option(lang_id,'validation-case-sensitive')
799
800         division_dict = load_division_dict(p,case_sensitive)
801         if division_dict ~= nil then
802             info(
803                 '  Loaded division dict for "%s" (lang. ID %s) from "%s"',
804                 cname,lang_id,path
805             )
806             lang_id_division_dict_read_table[lang_id] = true
807             lang_id_division_dict_table[lang_id] = division_dict
808             return division_dict,preprocessor
809         end
810
811     else
812         warning(
813             '  Specified division dict "%s" not found for "%s" (lang. ID %s)',
814             file_name,cname,lang_id
815         )
816     end
817 else
818     debug('  No division dict path specified')
819 end
820
821 lang_id_division_dict_read_table[lang_id] = false
822 division_dict = {}
823 lang_id_division_dict_table[lang_id] = division_dict
824
825 return division_dict,preprocessor
826
827 end

```

(End of definition for get_lang_division_dict_validation_preprocessor.)

11.12 Validating a hyphenation

Constants for the status of a hyphenation

```

828 local STATUS_VALID = 1
829 local STATUS_INVALID = 2
830 local STATUS_AMBIGUOUS = 3
831 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.

```

832 local function validate_division(lang_id,word,divided)
833
834     debug(
835         '    Validating "%s" for language ID %s',divided,lang_id
836     )
837     local division_dict,preprocessor
838         = get_lang_division_dict_validation_preprocessor(lang_id)
839
840     Use the preprocessor on the original and divided word.
841
842     word = preprocessor(word)
843     divided = preprocessor(divided)
844
845     word = delete_hyphens(word)
846
847     if not get_lang_option(lang_id,'validation-case-sensitive') then
848         word = unicode.utf8.lower(word)
849         divided = unicode.utf8.lower(divided)
850     end
851
852     local divisions = division_dict[word]
853     if not divisions then
854         return STATUS_UNKNOWN
855     end
856
857     local count = 0
858
859     for _,division in ipairs(divisions) do
860         if hyphenation_matches_pattern(divided,division) then
861             count = count + 1
862         end
863     end
864
865     if count == 0 then
866         return STATUS_INVALID
867     elseif count == #divisions then
868         return STATUS_VALID
869     else
870         return STATUS_AMBIGUOUS
871     end
872 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.

```

871 local LIGATURE_TEXT = {
872     [0xfb00] = 'ff',
873     [0xfb01] = 'fi',
874     [0xfb02] = 'fl',

```

```

875 [0xfb03] = 'ffi',
876 [0xfb04] = 'ffl',
877 [0xfb05] = 'ft',
878 [0xfb06] = 'st',
879 }

```

Cache to save table lookups when extracting text.

```

880 local font_characters = {}

```

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

```

881 local get_node_text
882 local get_nodelist_text

```

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

```

883 get_node_text = function(n)
884
885   if n.id == NODE_ID_GLYPH then
886
887     local ligature_text = LIGATURE_TEXT[n.char]
888     if ligature_text ~= nil then
889       return ligature_text
890     elseif n.components then
891       return get_nodelist_text(n.components)
892     else
893       -- See [https://tug.org/pipermail/luatex/2018-March/006786.html]
894       local characters = font_characters[n.font]
895       if not characters then
896         characters = fonts.hashes.identifiers[n.font].characters
897         font_characters[n.font] = characters
898       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.

```

899     local ok,retval = pcall(
900       function ()
901         return utf8.char(tonumber(characters[n.char].tounicode,16))
902       end
903     )
904     if ok then
905       return retval
906     else
907       return ''
908     end
909   end
910
911   elseif n.id == NODE_ID_DISC then
912
913     if n.replace then
914       return get_nodelist_text(n.replace)
915     else
916       return ''
917     end
918

```

```

919     else
920         return ''
921     end
922
923 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.

```

924 get_nodelist_text = function (head,last)
925
926     local text = ''
927
928     for item in node.traverse(head) do
929
930         text = text .. get_node_text(item)
931
932         if item == last then
933             break
934         end
935     end
936
937     return text
938
939 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.

```

940 local function is_possible_word_node(n)
941
942     return (
943         n.id == NODE_ID_GLYPH
944         or
945         n.id == NODE_ID_DISC
946         or
947         (n.id == NODE_ID_KERN and n.subtype == NODE_KERN_SUBTYPE_FONTKERN)
948         or
949         n.id == NODE_ID_MARGIN_KERN
950         or
951         n.id == NODE_ID_PENALTY
952         or
953         (n.id == NODE_ID_GLUE and n.width == 0)
954     )
955
956 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**.


```

957 local function is_possible_space_node(n)
958
959     return (
960         (n.id == NODE_ID_GLUE and n.width ~= 0)
961         or
962         (n.id == NODE_ID_KERN and n.subtype == NODE_KERN_SUBTYPE_USERKERN and n.kern ~= 0)
963     )
964
965 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`.)

```

966 -- local function get_first_glyph_lang(n)
967
968 --     local item = n
969 --     while item and is_possible_word_node(item) do
970 --         if item.id == NODE_ID_GLYPH then
971 --             return item.lang
972 --         end
973 --         item = item.next
974 --     end
975
976 --     return nil
977
978 -- 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 `LuaATEX`.)

```

979 local function get_disc_lang(n)
980
981 --     lang = get_first_glyph_lang(n.replace)
982 --     if lang then
983 --         return lang
984 --     end
985
986 --     lang = get_first_glyph_lang(n.pre)
987 --     if lang then
988 --         return lang
989 --     end
990
991 --     lang = get_first_glyph_lang(n.post)

```

```

992 -- if lang then
993 --   return lang
994 -- end
995
996 local item

```

Before the disc node.

```

997 item = n
998 while item and is_possible_word_node(item) do
999   if item.id == NODE_ID_GLYPH then
1000     return item.lang
1001   end
1002   item = item.prev
1003 end

```

After the disc node.

```

1004 item = n
1005 while item and is_possible_word_node(item) do
1006   if item.id == NODE_ID_GLYPH then
1007     return item.lang
1008   end
1009   item = item.next
1010 end
1011
1012 return 0
1013
1014 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`.

```

1015 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.

```

1016 local hlist_segment_list = {}

```

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

```

1017 local SEGMENT_WORD = 0
1018 local SEGMENT_SPACE = 1
1019 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).

```

1020 local function pre_linebreak(hlist_head,groupcode)
1021
1022   local word_start_node = nil

```

```

1023 local segment_count = 0
1024 local lang = nil
1025
1026 debug('Pre-linebreak processing start')
1027

```

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

```

1028 node.slide(hlist_head)
1029
1030 local item = hlist_head
1031 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`.

```

1032 if item.id == NODE_ID_MATH then
1033     assert(item.subtype == NODE_MATH_SUBTYPE_BEGIN)
1034     debug(' Math')
1035     while not (
1036         item.id == NODE_ID_MATH and item.subtype == NODE_MATH_SUBTYPE_END
1037     ) do
1038         item = item.next
1039     end
1040     item = item.next
1041
1042     segment_count = segment_count + 1
1043     hlist_segment_list[segment_count] = {
1044         type = SEGMENT_MATH
1045     }
1046
1047     goto continue
1048 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`.

```

1049 if is_possible_word_node(item) then
1050     word_start_node = item
1051     segment_count = segment_count + 1
1052     while item and is_possible_word_node(item) do
1053

```

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

```

1054         if item.id == NODE_ID_DISC then
1055             if not lang then
1056                 lang = get_disc_lang(item)
1057             end
1058             node.set_attribute(item, hyphen_attr, segment_count)
1059         end
1060
1061         item = item.next
1062     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.

```

1063     local word_end_node
1064     if item then
1065         word_end_node = item.prev
1066     else
1067         word_end_node = node.tail(word_start_node)
1068     end
1069
1070     local word = get_nodelist_text(word_start_node,word_end_node)
1071     hlist_segment_list[segment_count] = {
1072         type = SEGMENT_WORD,
1073         word = word,
1074         lang = lang,
1075     }
1076
1077     debug('  Word "%s"',word)
1078
1079     word_start_node = nil
1080     lang = nil
1081
1082     goto continue
1083 end

```

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

```

1084     if is_possible_space_node(item) then
1085         debug('  Space')
1086         segment_count = segment_count + 1
1087
1088         while item and is_possible_space_node(item) do
1089             item = item.next
1090         end
1091
1092         hlist_segment_list[segment_count] = {
1093             type = SEGMENT_SPACE
1094         }
1095
1096         goto continue
1097     end

```

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

```

1098     item = item.next
1099
1100     ::continue::
1101 end
1102
1103 debug('Pre-linebreak processing finish')
1104
1105 return true
1106 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.

1107 local USED_DISC_SEARCH_LIMIT = 20
1108
1109 local function get_used_disc(hlist_head)
1110
1111     debug(' Looking for disc node at end of line')
1112
1113     local item = node.tail(hlist_head)
1114
1115     local count = 0
1116     while item and item.id ~= NODE_ID_GLYPH and count < USED_DISC_SEARCH_LIMIT do
1117         if item.id == NODE_ID_DISC then
1118             return item
1119         end
1120         item = item.prev
1121         count = count + 1
1122     end
1123
1124     return nil
1125
1126 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.

```
1127 local function get_disc_word_start(hlist_head,n)
1128
1129     local item = n
1130
1131     while item do
1132         local prev = item.prev
1133
1134         if not (prev and is_possible_word_node(prev)) then
1135             return item
1136         end
1137
1138         item = prev
1139     end
1140
1141     return nil
1142 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.

```
1143 local function get_next_hlist(n)
1144
1145     local item = n.next
1146
1147     while item do
1148         if item.id == NODE_ID_HLIST then
1149             return item
1150         end
1151         item = item.next
1152     end
1153
1154     return nil
1155
1156 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.

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

```
1157 local function get_line_first_word(hlist_head)
1158
1159     local word_start_node = nil
1160
1161     for item in node.traverse(hlist_head) do
1162         if item.id == NODE_ID_GLYPH then
1163             if not word_start_node then
1164                 word_start_node = item
1165             end
1166         end
1167
1168         if not is_possible_word_node(item) then
1169             if word_start_node then
1170                 return get_nodelist_text(word_start_node, item.prev)
1171             end
1172         end
1173
1174     end
```

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

```
1175     if word_start_node then
1176         return get_nodelist_text(word_start_node, node.tail(hlist_head))
1177     else
1178         return nil
1179     end
1180 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.

```

1181 local function get_context(index,incr,target_word_count)
1182
1183   local result = ''
1184   local word_count = 0
1185
1186   local i = index + incr
1187   while (
1188     i > 0 and i <= #hlist_segment_list and word_count < target_word_count
1189   ) do
1190     local t = hlist_segment_list[i]
1191
1192     local item
1193
1194     if t.type == SEGMENT_WORD then
1195       item = t.word
1196       word_count = word_count + 1
1197     elseif t.type == SEGMENT_SPACE then
1198       item = STR_SPACE
1199     elseif t.type == SEGMENT_MATH then
1200       item = STR_MATH
1201     end
1202
1203     if incr > 0 then
1204       result = result .. item
1205     else
1206       result = item .. result
1207     end
1208
1209     i = i + incr
1210   end
1211
1212   return result
1213
1214 end

```

(End of definition for get_context.)

11.16.2 Flags

Define constant PDF literal nodes for flags.

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

```

1215 local function make_flag_node(code,context)
1216   if context then
1217     code = context .. ' ' .. code
1218   end
1219
1220   local n = node.new('whatsit','pdf_literal')
1221   n.data = 'q ' .. code .. ' Q'
1222
1223   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 1 5 8 1 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 1 1 7 m 5 3 1 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 1 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 1 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 ' -- Line width .8pt, colour white
1249 .. '-1.41421 -1.41421 m 1.41421 1.41421 1 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))

```



```

1267     end
1268
1269 end

```

(End of definition for insert_flag.)

write_flag Generic function to write a flag to the current T_EX list.

```

1270 local function write_flag(code)
1271
1272     node.write(make_flag_node(code,'1 0 0 1 0 -1 cm'))
1273
1274     local kern_n = node.new('kern','userkern')
1275     kern_n.kern=FLAG_WIDTH
1276     node.write(kern_n)
1277
1278 end

```

(End of definition for write_flag.)

write_flag_valid Write the valid flag to the current T_EX list.

```

1279 local function write_flag_valid()
1280
1281     write_flag(FLAG_VALID_CODE)
1282
1283 end

```

(End of definition for write_flag_valid.)

write_flag_invalid Write the invalid flag to the current T_EX list.

```

1284 local function write_flag_invalid()
1285
1286     write_flag(FLAG_INVALID_CODE)
1287
1288 end

```

(End of definition for write_flag_invalid.)

write_flag_ambiguous Write the ambiguous flag to the current T_EX list.

```

1289 local function write_flag_ambiguous()
1290
1291     write_flag(FLAG_AMBIGUOUS_CODE)
1292
1293 end

```

(End of definition for write_flag_ambiguous.)

write_flag_unknown Write the unknown flag to the current T_EX list.

```

1294 local function write_flag_unknown()
1295
1296     write_flag(FLAG_UNKNOWN_CODE)
1297
1298 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.

```
1299 local hyphenation_list = {}
1300 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`.

```
1301 local function check_line_hyphenation(hlist)
```

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

```
1302 local last_disc = get_used_disc(hlist.head)
1303 if not last_disc then
1304     debug(' No disc node found at end of line')
1305     return
1306 end
1307 debug(' Disc node found at end of line')
```

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

```
1308 local hyphenation_index = node.has_attribute(last_disc,hyphen_attr)
1309 local t = hlist_segment_list[hyphenation_index]
1310 assert(t)
1311 assert(t.type == SEGMENT_WORD)
1312 local word = t.word
1313 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.

```
1314 if not unicode.utf8.match(word,'%a') then
1315     debug(' Divided "word" contains no letters')
1316     return
1317 end
```

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

```
1318 local next_line = get_next_hlist(hlist)
1319
1320 if not next_line then
1321     debug(' No following line found (which should not happen)')
1322     return
1323 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.

```
1324 local pre = get_nodelist_text(get_disc_word_start(hlist.head,last_disc))
1325 local pre_temp = trim_nonlettershyphens_start(pre)
1326 if pre_temp ~= '' then
1327     pre = pre_temp
1328 end
```

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

```
1329 local post = trim_nonlettershyphens_end(get_line_first_word(next_line.head))
```

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

```
1330 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.)

```
1331 word = trim_nonlettershyphens_both(word)
```

```
1332
```

```
1333 local divided = pre .. post
```

```
1334 if string.sub(post,1,1) == '-' and string.sub(pre,-1,-1) == '-' then
```

```
1335     divided = pre .. string.sub(post,2,-1)
```

```
1336     debug(' Hyphenated word with split hyphen found: %s -> %s (%s/%s)',word,divided,pre,post)
```

```
1337 else
```

```
1338     debug(' Hyphenated word found: %s -> %s (%s/%s)',word,divided,pre,post)
```

```
1339 end
```

```
1340
```

```
1341 local status = validate_division(lang,word,divided)
```

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

```
1342 local flag_mode = get_lang_option(lang,'flag-mode')
```

```
1343 if flag_mode == 1 or (flag_mode == 2 and status ~= STATUS_VALID) then
```

```
1344     insert_flag(hlist.head,last_disc,status)
```

```
1345 end
```

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

```
1346 local context =
```

```
1347     get_context(
```

```
1348         hyphenation_index,-1,get_lang_option(lang,'context-before')
```

```
1349     )
```

```
1350     .. word_orig ..
```

```
1351     get_context(
```

```
1352         hyphenation_index,1,get_lang_option(lang,'context-after')
```

```
1353     )
```

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

```
1354 hyphenation_count = hyphenation_count + 1
```

```
1355 hyphenation_list[hyphenation_count] = {
```

```
1356     lang = lang,
```

```
1357     word = word,
```

```
1358     divided = divided,
```

```
1359     index = hyphenation_count,
```

```
1360     context = context,
```

```
1361     status = status,
```

```
1362 }
```

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.

```
1363 late_lua_n = node.new('whatsit','late_lua')
```

```

1364 late_lua_n.data =
1365     'lualisthyphen.set_hyphenation_page('
1366     .. hyphenation_count .. ',tex.count["c@page"])'
1367
1368 node.insert_after(hlist.head,last_disc,late_lua_n)
1369
1370 end

```

(End of definition for check_line_hyphenation.)

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

```

1371 local function set_hyphenation_page(index,page)
1372
1373     hyphenation_list[index].page = page
1374
1375 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.

```

1376 local function post_linebreak(vlist_head,groupcode)
1377
1378     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.

```

1379     for item in node.traverse(vlist_head) do
1380         if item.id == NODE_ID_HLIST then
1381             node.slide(item.head)
1382         end
1383     end

```

Now actually look for hyphenations.

```

1384     local line_no = 0
1385
1386     for item in node.traverse(vlist_head) do
1387
1388         if item.id == NODE_ID_HLIST then
1389             line_no = line_no + 1
1390             debug(' Line no. %d',line_no)
1391             check_line_hyphenation(item)
1392         end
1393
1394     end
1395
1396     hlist_segment_list = {}
1397
1398     debug('Post-linebreak processing end')
1399
1400     return true
1401
1402 end

```

(End of definition for post_linebreak.)

11.17 Add callbacks

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

```
1403 local LUA_LIST_HYPHEN_PRE_LINEBREAK = 'LUA_LIST_HYPHEN_PRE_LINEBREAK'
1404 luatexbase.add_to_callback(
1405     'pre_linebreak_filter',
1406     pre_linebreak,
1407     LUA_LIST_HYPHEN_PRE_LINEBREAK
1408 )
1409
1410 local LUA_LIST_HYPHEN_POST_LINEBREAK = 'LUA_LIST_HYPHEN_POST_LINEBREAK'
1411 luatexbase.add_to_callback(
1412     'post_linebreak_filter',
1413     post_linebreak,
1414     LUA_LIST_HYPHEN_POST_LINEBREAK
1415 )
```

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.

```
1416 local function equal_hyphenation_case_sensitive(a,b)
1417     return (
1418         a.word == b.word
1419         and
1420         a.divided == b.divided
1421     )
1422 end
```

(End of definition for equal_hyphenation_case_sensitive.)

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

```
1423 local function equal_hyphenation_case_insensitive(a,b)
1424     return (
1425         unicode.utf8.lower(a.word) == unicode.utf8.lower(b.word)
1426         and
1427         unicode.utf8.lower(a.divided) == unicode.utf8.lower(b.divided)
1428     )
1429 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.

```
1430 local function lessthan_hyphenation_case_sensitive(a,b)
1431     return (
1432         a.word < b.word
1433         or
1434         (
1435             a.word == b.word
```

```

1436         and
1437         a.divided < b.divided
1438     )
1439     or
1440     (
1441         a.word == b.word
1442         and
1443         a.divided == b.divided
1444         and
1445         a.index < b.index
1446     )
1447 )
1448 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.

```

1449 local function lessthan_hyphenation_case_insensitive(a,b)
1450     return (
1451         unicode.utf8.lower(a.word) < unicode.utf8.lower(b.word)
1452     or
1453     (
1454         unicode.utf8.lower(a.word) == unicode.utf8.lower(b.word)
1455         and
1456         unicode.utf8.lower(a.divided) < unicode.utf8.lower(b.divided)
1457     )
1458     or
1459     (
1460         unicode.utf8.lower(a.word) == unicode.utf8.lower(b.word)
1461         and
1462         unicode.utf8.lower(a.divided) < unicode.utf8.lower(b.divided)
1463         and
1464         a.index < b.index
1465     )
1466 )
1467 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.

```

1468 local function sort_hyphenation_list_none(hyphenation_list)
1469     table.sort(
1470         hyphenation_list,
1471         function(a,b)
1472             return a.index < b.index
1473         end
1474     )
1475 end

```

(End of definition for sort_hyphenation_list_none.)

sort_hyphenation_list_case Sort hyphenation_list case-sensitively.

```
1476 local function sort_hyphenation_list_case(hyphenation_list)
1477     table.sort(
1478         hyphenation_list,
1479         lessthan_hyphenation_case_sensitive
1480     )
1481 end
```

(End of definition for sort_hyphenation_list_case.)

sort_hyphenation_list_nocase Sort hyphenation_list case-insensitively.

```
1482 local function sort_hyphenation_list_nocase(hyphenation_list)
1483     table.sort(
1484         hyphenation_list,
1485         lessthan_hyphenation_case_insensitive
1486     )
1487 end
```

(End of definition for sort_hyphenation_list_nocase.)

11.18.3 Deduplication

deduplicate_hyphenation_list_case Remove duplicates from hyphenation_list case-sensitively.

```
1488 local function deduplicate_hyphenation_list_case(hyphenation_list)
1489     table.sort(
1490         hyphenation_list,
1491         lessthan_hyphenation_case_sensitive
1492     )
1493     list_uniq(
1494         hyphenation_list,
1495         equal_hyphenation_case_sensitive
1496     )
1497 end
```

(End of definition for deduplicate_hyphenation_list_case.)

deduplicate_hyphenation_list_nocase Remove duplicates from hyphenation_list case-insensitively.

```
1498 local function deduplicate_hyphenation_list_nocase(hyphenation_list)
1499     table.sort(
1500         hyphenation_list,
1501         lessthan_hyphenation_case_insensitive
1502     )
1503     list_uniq(
1504         hyphenation_list,
1505         equal_hyphenation_case_insensitive
1506     )
1507 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`.

```
1508 local function process_lang_hyphenation_list(lang_id,hyphenation_list)
1509
1510     local output_non_typeset = get_lang_option(lang_id,'output-non-typeset')
1511     local include_valid = (get_lang_option(lang_id,'output-mode') == 0)
1512
```

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

```
1513     list_filter(
1514         hyphenation_list,
1515         function (a)
1516             return (
1517                 (a.status ~= STATUS_VALID or include_valid)
1518                 and
1519                 (a.page or output_non_typeset)
1520             )
1521         end
1522     )
```

Deduplicate in accordance with the specified options for this language.

```
1523     local unique = get_lang_option(lang_id,'unique')
1524     if unique == 1 then
1525         deduplicate_hyphenation_list_case(hyphenation_list)
1526     elseif unique == 2 then
1527         deduplicate_hyphenation_list_nocase(hyphenation_list)
1528     end
```

Sort in accordance with the specified options for this language.

```
1529     local sort_mode = get_lang_option(lang_id,'sort')
1530     if sort_mode == 1 then
1531         sort_hyphenation_list_case(hyphenation_list)
1532     elseif sort_mode == 2 then
1533         sort_hyphenation_list_nocase(hyphenation_list)
1534     else
1535         sort_hyphenation_list_none(hyphenation_list)
1536     end
1537
1538 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`.

```
1539 local function write_lang_hyphenation_list_standard(f,hyphenation_list,widths)
1540
1541     for i,v in ipairs(hyphenation_list) do
1542
1543         if v then
1544             f:write(v.divided .. '\n')
1545         end
1546     end
1547
```



```

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 occurred in a box that was never output, so prepare the string containing the page accordingly.

```

1586         local page = v.page

```

```

1587     if page then
1588         page = STR_PAGE_PREFIX .. page
1589     else
1590         page = STR_PAGE_NONE
1591     end

```

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

```

1592     local status
1593     if show_status then
1594         status = v.status
1595         if status == STATUS_VALID then
1596             status = STR_VALID
1597         elseif status == STATUS_AMBIGUOUS then
1598             status = STR_AMBIGUOUS
1599         elseif status == STATUS_INVALID then
1600             status = STR_INVALID
1601         else
1602             status = STR_UNKNOWN
1603         end
1604         status = rpad(parenthesize(status),cols_status) .. STR_SPACE_TWO
1605     else
1606         status = ''
1607     end

```

Write everything out on a single line.

```

1608     f:write(
1609         rpad(v.word,cols_word)
1610         .. STR_ARROW
1611         .. rpad(v.divided,cols_divided)
1612         .. STR_SPACE_TWO
1613         .. status
1614         .. lpad(page,cols_page)
1615         .. STR_SPACE
1616         .. STR_QUOTE_OPEN
1617         .. v.context
1618         .. STR_QUOTE_CLOSE
1619         .. '\n'
1620     )
1621 end
1622
1623 end
1624
1625 end

```

(End of definition for write_lang_hyphenation_list_verbose.)

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

```

1626 local function get_settings_desc(lang_id)
1627
1628     local settings_desc
1629
1630     if get_lang_option(lang_id,'output-verbose') then
1631         settings_desc = string.format(
1632             'verbose=true,context-before=%d,context-after=%s',
1633             get_lang_option(lang_id,'context-before'),

```

```

1634     get_lang_option(lang_id, 'context-after')
1635 )
1636 else
1637     settings_desc = 'verbose=false'
1638 end
1639
1640 local ALL_NONVALID = {
1641     [0] = 'all',
1642     [1] = 'non-valid',
1643 }
1644 settings_desc = settings_desc .. string.format(
1645     ',output-mode=%s',
1646     ALL_NONVALID[get_lang_option(lang_id, 'output-mode')]
1647 )
1648 settings_desc = settings_desc .. string.format(
1649     ',include-non-typeset=%s',
1650     get_lang_option(lang_id, 'output-non-typeset')
1651 )
1652 local NONE_CASE_NOCASE = {
1653     [0] = 'none',
1654     [1] = 'case',
1655     [2] = 'nocase'
1656 }
1657 settings_desc = settings_desc .. string.format(
1658     ',sort=%s,unique=%s',
1659     NONE_CASE_NOCASE[get_lang_option(lang_id, 'sort')],
1660     NONE_CASE_NOCASE[get_lang_option(lang_id, 'unique')]
1661 )
1662
1663 return settings_desc
1664
1665 end

```

(End of definition for get_settings_desc.)

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

```

1666 local function prefix_output_directory(path)
1667
1668     local output_directory = status.output_directory
1669     if not output_directory then
1670         return path
1671     end
1672
1673     if string.sub(output_directory, -1, -1) == '/' then
1674         return output_directory .. path
1675     else
1676         return output_directory .. '/' .. path
1677     end
1678
1679 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.

```
1680 local function process_write_lang_hyphenation_list(  
1681     lang_id,hyphenation_list  
1682 )  
1683     process_lang_hyphenation_list(lang_id,hyphenation_list)  
1684  
1685     local output_path  
1686     local lang_desc  
1687  
1688     local cname = lang_id_cname_table[lang_id]  
1689  
1690     debug('Processing and writing hyphenation lists for "%s" (lang. ID %d)',cname,lang_id)  
1691  
1692     if cname then  
1693         output_path = get_lang_option(lang_id,'output-path')  
1694         if not output_path then  
1695             output_path = get_lang_option(LANG_DEFAULT,'output-prefix')  
1696             .. cname .. get_lang_option(LANG_DEFAULT,'output-extension')  
1697         end  
1698         lang_desc = string.format('language "%s" (ID %s)',cname,lang_id)  
1699     else  
1700         output_path = get_lang_option(LANG_DEFAULT,'output-prefix')  
1701         .. lang_id .. get_lang_option(LANG_DEFAULT,'output-extension')  
1702         lang_desc = 'language with ID ' .. lang_id  
1703     end  
1704  
1705     output_path = prefix_output_directory(output_path)  
1706     debug('Output path: "%s"',output_path)  
1707  
1708     local output_header = get_lang_option(lang_id,'output-header')  
1709     local output_verbose = get_lang_option(lang_id,'output-verbose')  
1710  
1711     local division_dict_read = lang_id_division_dict_read_table[lang_id]  
1712  
1713     local f = io.open(output_path,'w')  
1714  
1715     if output_header then  
1716         f:write('% This file was generated by lua-list-hyphen\n')  
1717         f:write('% Hyphenations for ' .. lang_desc .. '\n')  
1718         f:write('% General settings: ' .. get_settings_desc(lang_id) .. '\n')  
1719         local division_dict_path = lang_id_division_dict_path_table[lang_id]  
1720         if division_dict_path and division_dict_read then  
1721             f:write(  
1722                 '% Division dictionary read from: "' .. division_dict_path .. '"\n'  
1723             )  
1724         end  
1725     end  
1726  
1727     if output_verbose then  
1728         write_lang_hyphenation_list_verbose(f,hyphenation_list,division_dict_read)  
1729     else  
1730         write_lang_hyphenation_list_standard(f,hyphenation_list,division_dict_read)  
1731     end  
1732
```

```

1733     f:close()
1734
1735 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.

```

1736 local function process_write_hyphenation_lists()
1737
1738     local lang_hyphenation_table = {}
1739     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).

```

1740     for _,h in pairs(hyphenation_list) do
1741
1742         local lang = h.lang
1743
1744         local t = lang_hyphenation_table[lang]
1745         if not t then
1746             lang_hyphenation_table[lang] = {}
1747             t = lang_hyphenation_table[lang]
1748         end
1749
1750         table.insert(t,h)
1751
1752     end

```

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

```

1753     for lang_id,hyphenations in pairs(lang_hyphenation_table) do
1754         process_write_lang_hyphenation_list(lang_id,hyphenations)
1755     end
1756
1757 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 L^AT_EX frontend using `\lua_now:n` or `\lua_now:e`.

```

1758 lualisthyphen = lualisthyphen or {}
1759
1760 lualisthyphen.lang_load_names = lang_load_names
1761 lualisthyphen.store_lang_id_name = store_lang_id_name
1762
1763 lualisthyphen.set_debug = set_debug
1764 lualisthyphen.set_global_option = set_global_option
1765 lualisthyphen.set_lang_option = set_lang_option
1766 lualisthyphen.lang_options_check_name_known = lang_options_check_name_known
1767
1768 lualisthyphen.preprocessors = preprocessors
1769
1770 lualisthyphen.set_hyphenation_page = set_hyphenation_page
1771
1772 lualisthyphen.process_write_hyphenation_lists = process_write_hyphenation_lists

```

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

```
1773 lualisthyphen.write_flag_valid = write_flag_valid
1774 lualisthyphen.write_flag_invalid = write_flag_invalid
1775 lualisthyphen.write_flag_ambiguous = write_flag_ambiguous
1776 lualisthyphen.write_flag_unknown = write_flag_unknown
1777  $\langle$ /lua $\rangle$ 
```

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.

Symbols	
<code>\%</code>	513 , 710
A	
<code>allowlist-path</code> (option)	8
B	
<code>\begin</code>	489 , 498 , 510 , 514 , 520 , 539 , 695 , 711
bool commands:	
<code>\bool_to_str:N</code>	152 , 181 , 187 , 189 , 191 , 210 , 216 , 218 , 220
C	
check commands:	
<code>check_line_hyphenation</code>	1301
clist commands:	
<code>\clist_new:N</code>	141
<code>\clist_put_right:Nn</code>	145
<code>\clist_use:N</code>	248
context (option)	10
context-after (option)	10
context-before (option)	10
cs commands:	
<code>\cs_generate_variant:Nn</code>	147 , 164 , 169 , 230
<code>\cs_if_exist:NTF</code>	258
<code>\cs_new:Npn</code>	143 , 150 , 154 , 160 , 165 , 201 , 252 , 256 , 268
<code>\cs_set_eq:NN</code>	261
D	
debug	301
debug commands:	
<code>debug_dummy</code>	301
<code>debug_real</code>	301
deduplicate commands:	
<code>deduplicate_hyphenation_list_-case</code>	1488
<code>deduplicate_hyphenation_list_-nocase</code>	1498
delete commands:	
<code>delete_hyphens</code>	430
division-dict-path (option)	8
E	
<code>\en</code> ..	487 , 496 , 508 , 512 , 518 , 536 , 693 , 709
english commands:	
<code>english_strip_possessives</code>	751
equal commands:	
<code>equal_hyphenation_case_insensitive</code>	1423
<code>equal_hyphenation_case_sensitive</code>	1416
F	
<code>\file</code>	509
flag-mode (option)	9
<code>\foreignlanguage</code>	11
G	
get commands:	
<code>get_cname</code>	576
<code>get_context</code>	1181
<code>get_disc_lang</code>	979
<code>get_disc_word_start</code>	1127
<code>get_first_glyph_lang</code>	966
<code>get_lang_division_dict_validation-preprocessor</code>	755
<code>get_lang_option</code>	639
<code>get_line_first_word</code>	1157
<code>get_next_hlist</code>	1143
<code>get_node_text</code>	883
<code>get_nodelist_text</code>	924
<code>get_settings_desc</code>	1626
<code>get_used_disc</code>	1107
group commands:	
<code>\group_begin:</code>	234 , 260
<code>\group_end:</code>	244 , 265
H	
hook commands:	
<code>\hook_gput_code:nnn</code>	249 , 274
<code>\hyphenation</code>	6
hyphenation commands:	
<code>hyphenation_matches_pattern</code>	440
I	
identity	747
info	292
insert commands:	
<code>insert_flag</code>	1257
int commands:	
<code>\int_new:N</code>	33 , 39 , 65 , 71
<code>\int_set:Nn</code>	36 , 42 , 68 , 74
<code>\int_use:N</code>	183 , 185 , 193 , 195 , 197 , 199 , 212 , 214 , 222 , 224 , 226 , 228

is commands:		
is_possible_space_node		957
is_possible_word_node		940
J		
\jobname		5 , 8
K		
keys commands:		
\l_keys_choice_int		36 , 42 , 68 , 74
\keys_define:nn		12 , 16 , 20 , 23 , 26 , 29 , 34 , 40 , 45 , 48 , 52 , 55 , 66 , 72 , 77 , 80 , 83 , 89 , 98 , 111 , 133 , 136 , 232
\l_keys_key_str		92 , 101 , 105 , 114 , 119 , 124 , 129 , 138 , 236 , 238
\keys_set:nn	115 , 120 , 125 , 130 , 239 , 247	
\keys_set_known:nn		18
L		
lang commands:		
lang_load_language_dat		467 , 481
lang_load_names		557
lang_options_check_name_known	..	588
lessthan commands:		
lessthan_hyphenation_case_-		
insensitive		1449
lessthan_hyphenation_case_-		
sensitive		1430
list commands:		
list_filter		344
list_uniq		361
load commands:		
load_division_dict		688
lpad		420
lua commands:		
\lua_load_module:n		148
\lua_now:n		61 , 152 , 162 , 167 , 200 , 236 , 270 , 275
lualisthyphen internal commands:		
__lualisthyphen_babel_store_-		
lang_id_names:		254 , 256 , 256
__lualisthyphen_babel_store_-		
lang_id_names_elt:nnnn		263 , 268 , 268
__lualisthyphen_call_lua_-		
boolean:nN		150 , 150 , 170
\l__lualisthyphen_context_after_-		
int		55 , 195 , 224
\l__lualisthyphen_context_-		
before_int		55 , 193 , 222
\l__lualisthyphen_debug_bool	133 , 171	
\l__lualisthyphen_division_dict_-		
path_str		20 , 173 , 204
\l__lualisthyphen_flag_mode_int		33 , 183 , 212
__lualisthyphen_initialize:	...	250 , 252 , 252
\l__lualisthyphen_lang_name_str		21 , 231 , 238 , 242
\l__lualisthyphen_lang_options_-		
clist		18 , 19 , 141 , 145 , 248
__lualisthyphen_lua_set_global_-		
option:nn		160 , 160 , 164 , 172 , 174 , 176 , 178 , 180 , 182 , 184 , 186 , 188 , 190 , 192 , 194 , 196 , 198
__lualisthyphen_lua_set_lang_-		
option:nnn		165 , 165 , 169 , 203 , 205 , 207 , 209 , 211 , 213 , 215 , 217 , 219 , 221 , 223 , 225 , 227
__lualisthyphen_lua_set_per_-		
lang_options:n	..	201 , 201 , 230 , 241
__lualisthyphen_luastr_or_nil:N		154 , 154 , 173 , 175 , 177 , 179 , 204 , 206 , 208
\l__lualisthyphen_output_-		
extension_str		16 , 177
\l__lualisthyphen_output_header_-		
bool		48 , 189 , 218
\l__lualisthyphen_output_mode_-		
int		39 , 185 , 214
\l__lualisthyphen_output_non_-		
typeset_bool		45 , 187 , 216
\l__lualisthyphen_output_path_-		
str		83 , 206
\l__lualisthyphen_output_prefix_-		
str		12 , 175
\l__lualisthyphen_output_-		
verbose_bool		52 , 191 , 220
\l__lualisthyphen_sort_int		71 , 199 , 228
__lualisthyphen_store_lang_-		
options:nn		138 , 141 , 143 , 147
\l__lualisthyphen_unique_int	...	65 , 197 , 226
\l__lualisthyphen_validation_-		
case_sensitive_bool	..	29 , 181 , 210
\l__lualisthyphen_validation_-		
preprocessor_str		26 , 179 , 208
M		
make commands:		
make_flag_node		1215
msg commands:		
\msg_critical:nn		10
\msg_new:nnn		8
\msg_new:nnnn		86 , 95 , 108
\msg_warning:nnn		91 , 100 , 104
\msg_warning:nnnn	..	113 , 118 , 123 , 128

N	
<code>\n</code>	1544, 1619, 1716, 1717, 1718, 1722
<code>\NeedsTeXFormat</code>	3
O	
options:	
<code>allowlist-path</code>	8
<code>context</code>	10
<code>context-after</code>	10
<code>context-before</code>	10
<code>division-dict-path</code>	8
<code>flag-mode</code>	9
<code>output-extension</code>	8
<code>output-header</code>	10
<code>output-mode</code>	9
<code>output-non-typeset</code>	10
<code>output-path</code>	8
<code>output-prefix</code>	8
<code>output-verbose</code>	10
<code>sort</code>	10
<code>unique</code>	10
<code>validation-case-sensitive</code>	9
<code>validation-preprocessor</code>	8
<code>output-extension (option)</code>	8
<code>output-header (option)</code>	10
<code>output-mode (option)</code>	9
<code>output-non-typeset (option)</code>	10
<code>output-path (option)</code>	8
<code>output-prefix (option)</code>	8
<code>output-verbose (option)</code>	10
P	
<code>parenthesize</code>	425
<code>\pkg</code>	488
polyglossia commands:	
<code>polyglossia_store_lang_id_name</code>	622
post commands:	
<code>post_linebreak</code>	1376
pre commands:	
<code>pre_linebreak</code>	1020
prefix commands:	
<code>prefix_output_directory</code>	1666
process commands:	
<code>process_lang_hyphenation_list</code>	1508
<code>process_write_hyphenation_lists</code>	1736
<code>process_write_lang_hyphenation_-list</code>	1680
<code>\ProcessKeyOptions</code>	18, 149
<code>\ProvidesExplPackage</code>	4
R	
<code>rpadd</code>	415
S	
set commands:	
<code>set_debug</code>	310
<code>set_global_option</code>	659
<code>set_hyphenation_page</code>	1371
<code>set_lang_option</code>	671
<code>sort (option)</code>	10
sort commands:	
<code>sort_hyphenation_list_case</code>	1476
<code>sort_hyphenation_list_nocase</code>	1482
<code>sort_hyphenation_list_none</code>	1468
store commands:	
<code>store_lang_id_name</code>	597
str commands:	
<code>\str_if_empty:N</code>	156
<code>\str_new:N</code>	231
<code>\str_set_eq:NN</code>	238
<code>\str_use:N</code>	92, 101, 105, 114, 119, 124, 129, 138, 158, 236, 242
sys commands:	
<code>\sys_if_engine luatex:TF</code>	6
<code>\c_sys_jobname_str</code>	14
T	
TeX and L ^A T _E X 2 _ε commands:	
<code>\bbl@elt</code>	22, 262
<code>\bbl@languages</code>	22, 31, 258, 264
<code>\texttt</code>	513, 519, 710
trim commands:	
<code>trim_nonlettershyphens_both</code>	400
<code>trim_nonlettershyphens_end</code>	410
<code>trim_nonlettershyphens_start</code>	405
<code>trim_whitespace_both</code>	390
<code>trim_whitespace_left</code>	395
U	
uniformize commands:	
<code>uniformize_hyphens</code>	435
<code>unique (option)</code>	10
<code>\usepackage</code>	5
V	
validate commands:	
<code>validate_division</code>	832
<code>validation-case-sensitive (option)</code>	9
<code>validation-preprocessor (option)</code>	8
W	
warning	292
write commands:	
<code>write_flag</code>	1270
<code>write_flag_ambiguous</code>	1289
<code>write_flag_invalid</code>	1284
<code>write_flag_unknown</code>	1294
<code>write_flag_valid</code>	1279
<code>write_lang_hyphenation_list_-standard</code>	1539
<code>write_lang_hyphenation_list_-verbose</code>	1550