

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

Alan J. Cain[†]

Released 2026-07-25

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 v0.42.7, last revised 2026-07-25.

[†]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)	22
11.1	Initial set-up	22
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	30
11.9	Division dictionaries	31
11.10	Validation preprocessors	33
11.11	Getting the division dictionary and validation preprocessor for a language	33
11.12	Validating a hyphenation	35
11.13	Getting text from nodes	36
11.14	Getting language IDs from nodes	39
11.15	Pre-linebreak processing	40
11.16	Post-linebreak processing	42
	11.16.1 Node processing	42
	11.16.2 Flags	45
	11.16.3 Main list of hyphenations	47
	11.16.4 Check single line hyphenation	47
	11.16.5 Main post-linebreak function	49
11.17	Add callbacks	50
11.18	Processing hyphenation lists	50
	11.18.1 Comparisons and equality checks for stored hyphenations	51
	11.18.2 Sorting	52
	11.18.3 Deduplication	53
	11.18.4 Combined processing	53
11.19	Writing	54
11.20	Export public functions	59

Index	60
--------------	-----------

1 Introduction

TeX’s algorithm for finding points where a word can be hyphenated is good, but not perfect.¹ The present author writes in British English, where the valid division points can depend on both the pronunciation of a word and its internal structure (and hence its etymology). Currently, TeX’s pattern-based approach produces *bio-lo-gic*, *bio-logy*, *bio-lo-gist*, rather than the standard *bio-logic*, *biol-ogy*, *biolo-gist*.² To deal with such cases generally, at least a substantially larger number of patterns would be required than are available at present. There are also various words where the valid division points in British English cannot be deduced from their spelling alone: for instance, the verbs *at-trib-ute*, *pre-sent*, *pro-duce*, *re-cord* have different division points from the

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

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.

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 LuaTeX’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 LuaTeX IDs

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

9 Appendix: demonstration source

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

9.1 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  $\langle$ *package $\rangle$ 
2  $\langle$ @@=lualisthyphen $\rangle$ 
```

10.1 Initial set-up

Package identification/version information.

```
3 \NeedsTeXFormat{LaTeX2e}[2020-02-02]
4 \ProvidesExplPackage{lua-list-hyphen}{2026-07-25}{0.42.7}
5 {Listing hyphenated words for LuaLaTeX}
```

Check that LuaT_EX 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 \l__lualisthyphen_validation_preprocessor_str.)

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

```

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 }

```

(End of definition for \l__lualisthyphen_validation_case_sensitive_bool.)

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

```

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 }

```

(End of definition for \l__lualisthyphen_flag_mode_int.)

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

```

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 }

```

(End of definition for \l__lualisthyphen_output_mode_int.)

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

```

45 \keys_define:nn { lua-list-hyphen/global-per-lang }{
46   output-non-typeset .bool_set:N = \l__lualisthyphen_output_non_typeset_bool,
47 }

```

(End of definition for \l__lualisthyphen_output_non_typeset_bool.)

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

```

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 }

```

(End of definition for \l__lualisthyphen_output_header_bool.)

\l__lualisthyphen_output_verbose_bool Boolean option to indicate whether output lists of hyphenations should be written ver-
bously.

```

52 \keys_define:nn { lua-list-hyphen/global-per-lang }{
53   output-verbose .bool_set:N = \l__lualisthyphen_output_verbose_bool,
54 }

```

(End of definition for \l__lualisthyphen_output_verbose_bool.)

`\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

`\__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.

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

```

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

```

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

```

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

```

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

```

10.6 Initialization

At `begindocument`, run initialization code.

```

246 \hook_gput_code:nnn{ begindocument }{ lua-list-hyphen }{
247   \__lualisthyphen_initialize:
248 }

```

`\__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.)

```

249 \cs_new:Npn \__lualisthyphen_initialize:
250   {

```

```

251   \__lualisthyphen_babel_store_lang_id_names:
252   }

```

(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.

```

253 \cs_new:Npn \__lualisthyphen_babel_store_lang_id_names:
254 {
255   \cs_if_exist:NT\bbl@languages
256   {
257     \group_begin:
258     \cs_set_eq:NN
259       \bbl@elt
260       \__lualisthyphen_babel_store_lang_id_names_elt:nmmn
261       \bbl@languages
262     \group_end:
263   }%
264 }

```

(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.

```

265 \cs_new:Npn \__lualisthyphen_babel_store_lang_id_names_elt:nmmn #1#2#3#4
266 {
267   \lua_now:n{
268     lualisthyphen.store_lang_id_name(#2,'#1')
269   }
270 }

```

(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.

```

271 \hook_gput_code:nmm{ enddocument/info }{ lua-list-hyphen } {
272   \lua_now:n{
273     lualisthyphen.process_write_hyphenation_lists()
274   }
275 }
276 </package>

```

11 Implementation (Lua backend)

```

277 <lua>

```

11.1 Initial set-up

Module identification/version information.

```

278 local module_name = 'lualisthyphen'
279 local lualisthyphen_module = {

```

```

280     name      = module_name,
281     version   = '0.42.7',
282     date      = '2026-07-25',
283     description = 'lua-list-hyphen',
284     author    = 'Alan J. Cain',
285     copyright  = 'Alan J. Cain',
286     license   = 'LPPL v1.3c'
287 }
288 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.

```

289 local function info(s,...)
290     luatexbase.module_info(module_name,s:format(...))
291 end
292 local function warning(s,...)
293     luatexbase.module_warning(module_name,s:format(...))
294 end
295 local function error_(s,...)
296     luatexbase.module_error(module_name,s:format(...))
297 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**.

```

298 local function debug_dummy(s,...)
299     end
300
301 local function debug_real(s,...)
302     print(module_name .. ' DEBUG: ' .. s:format(...))
303 end
304
305 local debug = debug_dummy
306 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**.

```

307 local function set_debug(a)
308     if a then
309         debug = debug_real
310     else
311         debug = debug_dummy
312     end
313     debug_bool = a
314 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.

```
315 local NODE_ID_HLIST = node.id('hlist')
316 local NODE_ID_DISC = node.id('disc')
317 local NODE_ID_GLUE = node.id('glue')
318 local NODE_ID_KERN = node.id('kern')
319 local NODE_ID_MARGIN_KERN = node.id('margin_kern')
320 local NODE_ID_GLYPH = node.id('glyph')
321 local NODE_ID_MATH = node.id('math')
```

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

```
322 local NODE_KERN_SUBTYPE_FONTKERN
323 local NODE_KERN_SUBTYPE_USERKERN
324 for k,v in pairs(node.subtypes('kern')) do
325   if v == 'fontkern' then
326     NODE_KERN_SUBTYPE_FONTKERN = k
327   elseif v == 'userkern' then
328     NODE_KERN_SUBTYPE_USERKERN = k
329   end
330 end
```

Define constants for the math node subtypes.

```
331 local NODE_MATH_SUBTYPE_BEGIN
332 local NODE_MATH_SUBTYPE_END
333 for k,v in pairs(node.subtypes('math')) do
334   if v == 'beginmath' then
335     NODE_MATH_SUBTYPE_BEGIN = k
336   elseif v == 'endmath' then
337     NODE_MATH_SUBTYPE_END = k
338   end
339 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

```
340 local function list_filter(t, f)
341   local j = 1
342   local n = #t
343
344   for i=1,n do
345     if (f(t[i])) then
346       if (i ~= j) then
347         t[j] = t[i]
348         t[i] = nil
349       end
350       j = j + 1
351     else
352       t[i] = nil
353     end
354   end
355 end
```

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

```

353     end
354   end
355
356 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.

```

357 local function list_uniq(t, f)
358   local j = 1
359   local n = #t
360
361   for i=2,n do
362     if (f(t[i],t[j])) then
363       t[i] = nil
364     else
365       j = i
366     end
367   end
368
369   list_filter(
370     t,
371     function(a) return a end
372   )
373 end

```

(End of definition for list_uniq.)

11.5 String manipulation functions

String constants (which will ultimately be output).

```

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

```

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

```

386 local function trim_whitespace_both(s)
387
388   return unicode.utf8.match(s, '^%s*(.-)%s*$')
389
390 end

```

(End of definition for trim_whitespace_both.)

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

```
391 local function trim_whitespace_left(s)
392
393   return unicode.utf8.match(s, '^%s*(.*)$')
394
395 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.

```
396 local function trim_nonlettershyphens_both(s)
397
398   return unicode.utf8.match(s, '^[^%a-]*(-)[^%a-]*$')
399
400 end
```

(End of definition for trim_nonlettershyphens_both.)

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

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

(End of definition for trim_nonlettershyphens_start.)

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

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

(End of definition for trim_nonlettershyphens_end.)

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

```
411 local function rpadd(s,n)
412
413   return s .. unicode.utf8.rep(STR_SPACE,n - unicode.utf8.len(s))
414
415 end
```

(End of definition for rpadd.)

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

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

(End of definition for lpad.)

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

```
421 local function parenthesize(s)
422
423   return '(' .. s .. ')'
424
425 end
```

(End of definition for parenthesize.)

11.6 Hyphen utility functions

delete_hyphens Delete all hyphens.

```
426 local function delete_hyphens(s)
427
428   return unicode.utf8.gsub(s, '-', '')
429
430 end
```

(End of definition for delete_hyphens.)

uniformize_hyphens Make all appearances of | and ¦ into hyphens.

```
431 local function uniformize_hyphens(s)
432
433   return unicode.utf8.gsub(unicode.utf8.gsub(s, '|', '-'), '¦', '-')
434
435 end
```

(End of definition for uniformize_hyphens.)

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

```
436 local function hyphenation_matches_pattern(hyphenation, pattern)
437
438   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.

```
439   local i = 1
440   for j=1,#pattern do
441
442     local s
443     local t
444
445     s = string.sub(hyphenation, i, i)
446     t = string.sub(pattern, j, j)
447
448     if s == t then
449       i = i + 1
450     elseif s == '-' or t ~= '-' then
```

```

451         debug('      Fails at i=%d,s=%s,j=%d,t=%s',i,s,j,t)
452         return false
453     end
454
455 end
456
457 debug('      Matches')
458 return true
459
460 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 list of names.

```

461 local lang_name_cname_table = {}
462 local lang_cname_name_list_table = {}

```

Populate these tables from `language.dat.lua` (which means that they will contain languages not in use).

```

463 for k,v in pairs(require('language.dat.lua')) do
464     local t = { k }
465     lang_cname_name_list_table[k] = t
466     lang_name_cname_table[k] = k
467     for _,vv in pairs(v.synonyms) do
468         lang_name_cname_table[vv] = k
469         table.insert(t,vv)
470     end
471 end
472
473 if debug_bool then
474     debug('Language name -> cname:')
475     for k,v in pairs(lang_name_cname_table) do
476         debug('  %s -> %s',k,v)
477     end
478     debug('Language cname -> name list:')
479     for k,v in pairs(lang_cname_name_list_table) do
480         local s = ''
481         for _,vv in ipairs(v) do
482             s = s .. vv .. ', '
483         end
484         debug('  %s -> %s ',k,s)
485     end
486 end

```

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

```

487 local lang_id_cname_table = {}
488 local lang_cname_id_table = {}

```

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

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

```

489 local function store_lang_id_name(lang_id,name)
490
491     local cname = lang_name_cname_table[name]

```

The supplied name might not be in lang_name_cname_table (since things like dumylang are not in language.dat.lua).

```

492     if cname == nil then
493         cname = name
494         lang_name_cname_table[name] = cname
495         lang_cname_name_list_table[cname] = { name }
496     end
497
498     local lang_id_cname = lang_id_cname_table[lang_id]
499     if lang_id_cname == cname then
500         debug(
501             'Language name "%s" is a synonym for cname "%s" (lang. ID %s)',
502             name,cname,lang_id
503         )
504         return
505     elseif lang_id_cname ~= nil then
506         error(
507             'Languages with canonical names "%s" and "%s" both have ID %s',
508             cname,lang_id_cname,lang_id
509         )
510         return
511     end
512
513     lang_id_cname_table[lang_id] = cname
514     lang_cname_id_table[cname] = lang_id
515
516     debug('Language ID %s has cname "%s"',lang_id,cname)
517
518 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.

```

519 local function polyglossia_store_lang_id_name(lang_id)
520
521     if not polyglossia then
522         return
523     end
524
525     for name,language in pairs(polyglossia.newloader_loaded_languages) do
526         local this_lang_id = lang.id(language)
527         if this_lang_id == lang_id then
528             store_lang_id_name(lang_id,name)
529         end
530     end
531
532 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.

```
533 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.

```
534 local lang_id_options_table = {}
```

```
535 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.

```
536 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`.

```
537   local options = lang_id_options_table[lang_id]
```

```
538
```

```
539   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.

```
540     local cname = lang_id_cname_table[lang_id]
```

```
541     if cname then
```

```
542       options = lang_cname_options_table[cname]
```

```
543     end
```

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

```
544     if not options then
```

```
545       options = {}
```

```
546       for k,v in pairs(lang_id_options_table[LANG_DEFAULT]) do
```

```
547         options[k] = v
```

```
548       end
```

```
549     end
```

```
550     lang_id_options_table[lang_id] = options
```

```
551   end
```

```
552
```

```
553   return options[key]
```

```
554
```

```
555 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`.

```

556 local function set_global_option(key,value)
557
558   local options = lang_id_options_table[LANG_DEFAULT]
559   if not options then
560     options = {}
561     lang_id_options_table[LANG_DEFAULT] = options
562   end
563
564   options[key] = value
565   debug('Set global option %s="%s"',key,value)
566
567 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.

```

568 local function set_lang_option(name,key,value)
569
570   local cname = lang_name_cname_table[name]
571   if not cname then
572     cname = name
573   end
574
575   local options = lang_cname_options_table[cname]
576   if not options then
577     options = {}
578     lang_cname_options_table[cname] = options
579   end
580
581   options[key] = value
582   debug('Set lang. "%s" option %s="%s"',cname,key,value)
583
584 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 `!`.

```

585 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.

```

586 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.

```

587 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.

```

588 local function load_division_dict(path,case_sensitive)
589
590     local count = 0
591
592     local f = io.open(path,'r')
593     % \en{macrocode}
594     % If the file was not found, return immediately.
595     % \begin{macrocode}
596     if f == nil then
597         return nil
598     end
599
600     debug(' Loading divisions from "%s"',path)
601
602     local division_dict = {}
603
604     io.input(f)
605
606     for line in io.lines() do
607
608         line = trim_whitespace_left(line)
609         % \en{macrocode}
610         % Lines in which the first non-whitespace character is \texttt{\%} are comments.
611         % \begin{macrocode}
612         if #line == 0 or string.sub(line,1,1) == '%' then
613             goto continue
614         end
615
616         count = count + 1
617
618         local pattern = unicode.utf8.match(line,'^(.)%s') or line
619
620         pattern = uniformize_hyphens(pattern)
621
622         if not case_sensitive then
623             pattern = unicode.utf8.lower(pattern)
624         end
625
626         local word = delete_hyphens(pattern)
627
628         local divisions = division_dict[word]
629         if divisions == nil then
630             divisions = {}
631             division_dict[word] = divisions
632         end
633
634         table.insert(divisions,pattern)

```

```

635     ::continue::
636 end
637
638 io.close(f)
639
640 debug('Read %s divisions from "%s"',count,path)
641
642 return division_dict
643
644 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.

```

645 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.

```

646 local preprocessors = {}

```

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

```

647 local function identity(s)
648     return s
649 end
650 preprocessors.identity = identity

```

(End of definition for identity.)

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

```

651 local function english_strip_possessives(s)
652     return string.gsub(s, 's$', '')
653 end
654 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.

```

655 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'.

```

656 local division_dict = lang_id_division_dict_table[lang_id]
657 if division_dict ~= nil then
658     return division_dict, lang_id_validation_preprocessor_table[lang_id]
659 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 `inlang_id_cname_list_table` for this language ID.

```

660 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`.

```

661 local cname = lang_id_cname_table[lang_id]
662 local preprocessor_name = get_lang_option(lang_id, 'validation-preprocessor')
663 local preprocessor
664 if not preprocessor_name then
665     debug(
666         'No preprocessor specified for "%s"', cname
667     )
668     preprocessor = identity
669 else
670     preprocessor = preprocessors[preprocessor_name]
671     if not preprocessor then
672         preprocessor = _G[preprocessor_name]
673     end
674     if not preprocessor then
675         debug(
676             'Preprocessor specified for "%s" was not found', cname
677         )
678         preprocessor = identity
679     else
680         debug(
681             'Using preprocessor specified for "%s"', cname
682         )
683     end
684 end
685 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.

```

686 debug(
687     'Attempting to load division dictionary for "%s" (lang. ID %s)',
688     cname, lang_id
689 )
690 local file_name = get_lang_option(lang_id, 'division-dict-path')
691 if file_name then
692     debug('  Filename: ' .. file_name)
693     local p = kpse.find_file(file_name)
694     if p then
695         debug('  Path (after kpathsea lookup): %s', p)

```

```

696     lang_id_division_dict_path_table[lang_id] = p
697
698     local case_sensitive = get_lang_option(lang_id,'validation-case-sensitive')
699
700     division_dict = load_division_dict(p,case_sensitive)
701     if division_dict ~= nil then
702         info(
703             '   Loaded division dict for "%s" (lang. ID %s) from "%s"',
704             cname,lang_id,path
705         )
706         lang_id_division_dict_read_table[lang_id] = true
707         lang_id_division_dict_table[lang_id] = division_dict
708         return division_dict,preprocessor
709     end
710
711     else
712         warning(
713             '   Specified division dict "%s" not found for "%s" (lang. ID %s)',
714             file_name,cname,lang_id
715         )
716     end
717 else
718     debug('   No division dict path specified')
719 end
720
721 lang_id_division_dict_read_table[lang_id] = false
722 division_dict = {}
723 lang_id_division_dict_table[lang_id] = division_dict
724
725 return division_dict,preprocessor
726
727 end

```

(End of definition for get_lang_division_dict_validation_preprocessor.)

11.12 Validating a hyphenation

Constants for the status of a hyphenation

```

728 local STATUS_VALID = 1
729 local STATUS_INVALID = 2
730 local STATUS_AMBIGUOUS = 3
731 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.

```

732 local function validate_division(lang_id,word,divided)
733
734     debug(
735         '   Validating "%s" for language ID %s',divided,lang_id
736     )
737     local division_dict,preprocessor
738     = get_lang_division_dict_validation_preprocessor(lang_id)

```

Use the preprocessor on the original and divided word.

```
739 word = preprocessor(word)
740 divided = preprocessor(divided)
741
742 if not get_lang_option(lang_id, 'validation-case-sensitive') then
743     word = unicode.utf8.lower(word)
744     divided = unicode.utf8.lower(divided)
745 end
746
747 local divisions = division_dict[word]
748 if not divisions then
749     return STATUS_UNKNOWN
750 end
751
752 local count = 0
753
754 for _, division in ipairs(divisions) do
755     if hyphenation_matches_pattern(divided, division) then
756         count = count + 1
757     end
758 end
759
760 if count == 0 then
761     return STATUS_INVALID
762 elseif count == #divisions then
763     return STATUS_VALID
764 else
765     return STATUS_AMBIGUOUS
766 end
767
768 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.

```
769 local LIGATURE_TEXT = {
770     [0xfb00] = 'ff',
771     [0xfb01] = 'fi',
772     [0xfb02] = 'fl',
773     [0xfb03] = 'ffi',
774     [0xfb04] = 'ffl',
775     [0xfb05] = 'ft',
776     [0xfb06] = 'st',
777 }
```

Cache to save table lookups when extracting text.

```
778 local font_characters = {}
```

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

```
779 local get_node_text
780 local get_nodelist_text
```

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

```

781 get_node_text = function(n)
782
783   if n.id == NODE_ID_GLYPH then
784
785     local ligature_text = LIGATURE_TEXT[n.char]
786     if ligature_text ~= nil then
787       return ligature_text
788     elseif n.components then
789       return get_nodelist_text(n.components)
790     else
791       -- See [https://tug.org/pipermail/luatex/2018-March/006786.html]
792       local characters = font_characters[n.font]
793       if not characters then
794         characters = fonts.hashes.identifiers[n.font].characters
795         font_characters[n.font] = characters
796       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.

```

797     local ok,retval = pcall(
798       function ()
799         return utf8.char(tonumber(characters[n.char].tounicode,16))
800       end
801     )
802     if ok then
803       return retval
804     else
805       return ''
806     end
807   end
808
809   elseif n.id == NODE_ID_DISC then
810
811     if n.replace then
812       return get_nodelist_text(n.replace)
813     else
814       return ''
815     end
816
817   else
818     return ''
819   end
820
821 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.

```

822 get_nodelist_text = function (head,last)
823

```

```

824 local text = ''
825
826 for item in node.traverse(head) do
827
828     text = text .. get_node_text(item)
829
830     if item == last then
831         break
832     end
833 end
834
835 return text
836
837 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**.

```

838 local function is_possible_word_node(n)
839
840     return (
841         n.id == NODE_ID_GLYPH
842         or
843         n.id == NODE_ID_DISC
844         or
845         (n.id == NODE_ID_KERN and n.subtype == NODE_KERN_SUBTYPE_FONTKERN)
846         or
847         n.id == NODE_ID_MARGIN_KERN
848     )
849
850 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**.

```

851 local function is_possible_space_node(n)
852
853     return (
854         n.id == NODE_ID_GLUE
855         or
856         (n.id == NODE_ID_KERN and n.subtype == NODE_KERN_SUBTYPE_USERKERN)
857     )
858
859 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`.)

```
860 -- local function get_first_glyph_lang(n)
861
862 --   local item = n
863 --   while item and is_possible_word_node(item) do
864 --     if item.id == NODE_ID_GLYPH then
865 --       return item.lang
866 --     end
867 --     item = item.next
868 --   end
869
870 --   return nil
871
872 -- 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`.)

```
873 local function get_disc_lang(n)
874
875 -- lang = get_first_glyph_lang(n.replace)
876 -- if lang then
877 --   return lang
878 -- end
879
880 -- lang = get_first_glyph_lang(n.pre)
881 -- if lang then
882 --   return lang
883 -- end
884
885 -- lang = get_first_glyph_lang(n.post)
886 -- if lang then
887 --   return lang
888 -- end
889
890 local item
```

Before the disc node.

```
891 item = n
892 while item and is_possible_word_node(item) do
893   if item.id == NODE_ID_GLYPH then
894     return item.lang
895   end
896   item = item.prev
897 end
```

After the disc node.

```

898   item = n
899   while item and is_possible_word_node(item) do
900       if item.id == NODE_ID_GLYPH then
901           return item.lang
902       end
903       item = item.next
904   end
905
906   return 0
907
908 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`.

```

909 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.

```

910 local hlist_segment_list = {}

```

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

```

911 local SEGMENT_WORD = 0
912 local SEGMENT_SPACE = 1
913 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).

```

914 local function pre_linebreak(hlist_head,groupcode)
915
916     local word_start_node = nil
917     local segment_count = 0
918     local lang = nil
919
920     debug('Pre-linebreak processing start')
921

```

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

```

922     node.slide(hlist_head)
923
924     local item = hlist_head
925     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`.

```

926     if item.id == NODE_ID_MATH then
927         assert(item.subtype == NODE_MATH_SUBTYPE_BEGIN)
928         while not (
929             item.id == NODE_ID_MATH and item.subtype == NODE_MATH_SUBTYPE_END
930         ) do
931             item = item.next
932         end
933         item = item.next
934
935         segment_count = segment_count + 1
936         hlist_segment_list[segment_count] = {
937             type = SEGMENT_MATH
938         }
939
940         goto continue
941     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`.

```

942     if is_possible_word_node(item) then
943         word_start_node = item
944         segment_count = segment_count + 1
945         while item and is_possible_word_node(item) do
946

```

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

```

947             if item.id == NODE_ID_DISC then
948                 if not lang then
949                     lang = get_disc_lang(item)
950                 end
951                 node.set_attribute(item,hyphen_attr,segment_count)
952             end
953
954             item = item.next
955         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.

```

956         local word_end_node
957         if item then
958             word_end_node = item.prev
959         else
960             word_end_node = node.tail(word_start_node)
961         end
962
963         local word = get_nodelist_text(word_start_node,word_end_node)
964         hlist_segment_list[segment_count] = {
965             type = SEGMENT_WORD,
966             word = word,
967             lang = lang,
968         }

```

```

969
970     word_start_node = nil
971     lang = nil
972
973     goto continue
974 end

```

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

```

975     if is_possible_space_node(item) then
976         segment_count = segment_count + 1
977
978         while item and is_possible_space_node(item) do
979             item = item.next
980         end
981
982         hlist_segment_list[segment_count] = {
983             type = SEGMENT_SPACE
984         }
985
986         goto continue
987     end

```

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

```

988     item = item.next
989
990     ::continue::
991 end
992
993 debug('Pre-linebreak processing finish')
994
995 return true
996 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.

```

997 local USED_DISC_SEARCH_LIMIT = 20
998
999 local function get_used_disc(hlist_head)
1000
1001     debug(' Looking for disc node at end of line')
1002
1003     local item = node.tail(hlist_head)

```

```

1004
1005     local count = 0
1006     while item and item.id ~= NODE_ID_GLYPH and count < USED_DISC_SEARCH_LIMIT do
1007         if item.id == NODE_ID_DISC then
1008             return item
1009         end
1010         item = item.prev
1011         count = count + 1
1012     end
1013
1014     return nil
1015
1016 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.

```

1017 local function get_disc_word_start(hlist_head,n)
1018
1019     local item = n
1020
1021     while item do
1022         local prev = item.prev
1023
1024         if not (prev and is_possible_word_node(prev)) then
1025             return item
1026         end
1027
1028         item = prev
1029     end
1030
1031     return nil
1032 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.

```

1033 local function get_next_hlist(n)
1034
1035     local item = n.next
1036
1037     while item do
1038         if item.id == NODE_ID_HLIST then
1039             return item
1040         end
1041         item = item.next
1042     end
1043
1044     return nil
1045
1046 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.

```
1047 local function get_line_first_word(hlist_head)
word_start_node is either nil or the (glyph) node that starts the word.
1048   local word_start_node = nil
1049
1050   for item in node.traverse(hlist_head) do
1051
1052     if item.id == NODE_ID_GLYPH then
1053       if not word_start_node then
1054         word_start_node = item
1055       end
1056     end
1057
1058     if not is_possible_word_node(item) then
1059       if word_start_node then
1060         return get_nodelist_text(word_start_node,item.prev)
1061       end
1062     end
1063
1064   end
```

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

```
1065   if word_start_node then
1066     return get_nodelist_text(word_start_node,node.tail(hlist_head))
1067   else
1068     return nil
1069   end
1070 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.

```
1071 local function get_context(index,incr,target_word_count)
1072
1073   local result = ''
1074   local word_count = 0
1075
1076   local i = index + incr
1077   while (
1078     i > 0 and i <= #hlist_segment_list and word_count < target_word_count
1079   ) do
1080     local t = hlist_segment_list[i]
1081
1082     local item
1083
1084     if t.type == SEGMENT_WORD then
1085       item = t.word
1086       word_count = word_count + 1
1087     elseif t.type == SEGMENT_SPACE then
1088       item = STR_SPACE
1089     elseif t.type == SEGMENT_MATH then
1090       item = STR_MATH
```

```

1091     end
1092
1093     if incr > 0 then
1094         result = result .. item
1095     else
1096         result = item .. result
1097     end
1098
1099     i = i + incr
1100 end
1101
1102 return result
1103
1104 end

```

(End of definition for get_context.)

11.16.2 Flags

Define constant PDF literal nodes for flags.

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

```

1105 local function make_flag_node(code,context)
1106     if context then
1107         code = context .. ' ' .. code
1108     end
1109
1110     local n = node.new('whatsit','pdf_literal')
1111     n.data = 'q ' .. code .. ' Q'
1112
1113     return n
1114
1115 end

```

(End of definition for make_flag_node.)

Transformation to shift flags into the margin.

```

1116 local FLAG_MARGIN_SHIFT = '1 0 0 1 4 -1 cm'

```

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

```

1117 local FLAG_WIDTH = tex.sp('6pt')

```

Code for flags.

```

1118 local FLAG_VALID_CODE
1119     = '0 0 6 10 re .1 .7 .1 rg f ' -- Green background
1120     .. '1 w 1 G ' -- Line width 1pt, colour white
1121     .. '1 5.5 m 2.33333 3 1 5 8 1 S' -- "Tick"
1122 local FLAG_INVALID_CODE
1123     = '0 0 6 10 re .9 0 0 rg f ' -- Red background
1124     .. '1 w 1 G ' -- Line width 1pt, colour white
1125     .. '1 3 m 5 7 1 1 7 m 5 3 1 S'
1126 local FLAG_AMBIGUOUS_CODE = (
1127     '0 0 6 10 re .1 .7 .7 rg f ' -- Cyan background
1128     .. '1 0 0 1 2.85 6.25 cm ' -- Shift
1129     .. '1 w 1 G ' -- Line width 1pt, colour white

```

```

1130 .. '-1.55885 .9 m -1 1.4 -.65 2 0 2 c 1.3 2 1.8 1.3 1.8 .5 c '
1131 .. '1.8 -.75 0 -1.25 0 -2.5 c 0 -2.75 1 S ' -- Question mark body
1132 .. '1.4 w 1 J ' -- Line width 1.4pt, line cap round
1133 .. '0 -4.25 m 0 -4.25 1 S' -- Question mark dot
1134 )
1135 local FLAG_UNKNOWN_CODE = (
1136 '0 0 6 10 re .9 .9 0 rg f ' -- Yellow background
1137 .. '1 0 0 1 3 5 cm ' -- Shift
1138 .. '.8 w 1 G ' -- Linec with .8pt, colour white
1139 .. '-1.41421 -1.41421 m 1.41421 1.41421 1 S ' -- Stroke of 0
1140 .. '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 '
1141 .. '1.2 -3 2 -1.2 2 0 c h S' -- Outside of 0
1142 )

```

Nodes made from the preceding code.

```

1143 local FLAG_VALID_NODE = make_flag_node(FLAG_VALID_CODE,FLAG_MARGIN_SHIFT)
1144 local FLAG_INVALID_NODE = make_flag_node(FLAG_INVALID_CODE,FLAG_MARGIN_SHIFT)
1145 local FLAG_AMBIGUOUS_NODE = make_flag_node(FLAG_AMBIGUOUS_CODE,FLAG_MARGIN_SHIFT)
1146 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.

```

1147 local function insert_flag(head,current,status)
1148
1149   if status == STATUS_VALID then
1150     node.insert_after(head,current,node.copy(FLAG_VALID_NODE))
1151   elseif status == STATUS_UNKNOWN then
1152     node.insert_after(head,current,node.copy(FLAG_UNKNOWN_NODE))
1153   elseif status == STATUS_INVALID then
1154     node.insert_after(head,current,node.copy(FLAG_INVALID_NODE))
1155   elseif status == STATUS_AMBIGUOUS then
1156     node.insert_after(head,current,node.copy(FLAG_AMBIGUOUS_NODE))
1157   end
1158
1159 end

```

(End of definition for insert_flag.)

`write_flag` Generic function to write a flag to the current $\TeX$ list.

```

1160 local function write_flag(code)
1161
1162   node.write(make_flag_node(code,'1 0 0 1 0 -1 cm'))
1163
1164   local kern_n = node.new('kern','userkern')
1165   kern_n.kern=FLAG_WIDTH
1166   node.write(kern_n)
1167
1168 end

```

(End of definition for write_flag.)

`write_flag_valid` Write the valid flag to the current $\TeX$ list.

```

1169 local function write_flag_valid()
1170
1171   write_flag(FLAG_VALID_CODE)

```

```

1172
1173 end

```

(End of definition for write_flag_valid.)

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

```

1174 local function write_flag_invalid()
1175
1176   write_flag(FLAG_INVALID_CODE)
1177
1178 end

```

(End of definition for write_flag_invalid.)

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

```

1179 local function write_flag_ambiguous()
1180
1181   write_flag(FLAG_AMBIGUOUS_CODE)
1182
1183 end

```

(End of definition for write_flag_ambiguous.)

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

```

1184 local function write_flag_unknown()
1185
1186   write_flag(FLAG_UNKNOWN_CODE)
1187
1188 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.

```

1189 local hyphenation_list = {}
1190 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`.

```

1191 local function check_line_hyphenation(hlist)

```

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

```

1192   local last_disc = get_used_disc(hlist.head)
1193   if not last_disc then
1194     debug(' No disc node found at end of line')
1195     return
1196   end
1197   debug(' Disc node found at end of line')

```

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

```
1198 local hyphenation_index = node.has_attribute(last_disc,hyphen_attr)
1199 local t = hlist_segment_list[hyphenation_index]
1200 assert(t)
1201 assert(t.type == SEGMENT_WORD)
1202 local word = t.word
1203 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.

```
1204 if not unicode.utf8.match(word,'%a') then
1205     debug(' Divided "word" contains no letters')
1206     return
1207 end
```

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

```
1208 local next_line = get_next_hlist(hlist)
1209
1210 if not next_line then
1211     debug(' No following line found (which should not happen)')
1212     return
1213 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.

```
1214 local pre = get_nodelist_text(get_disc_word_start(hlist.head,last_disc))
1215 local pre_temp = trim_nonlettershyphens_start(pre)
1216 if pre_temp ~= '' then
1217     pre = pre_temp
1218 end
```

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

```
1219 local post = trim_nonlettershyphens_end(get_line_first_word(next_line.head))
```

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

```
1220 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.)

```
1221 word = trim_nonlettershyphens_both(word)
1222
1223 local divided = pre .. post
1224 debug(' Hyphenated word found: %s -> %s (%s|%s)',word,divided,pre,post)
1225 local status = validate_division(lang,word,divided)
```

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

```
1226 local flag_mode = get_lang_option(lang,'flag-mode')
1227 if flag_mode == 1 or (flag_mode == 2 and status ~= STATUS_VALID) then
1228     insert_flag(hlist.head,last_disc,status)
1229 end
```

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

```

1230     local context =
1231       get_context(
1232         hyphenation_index,-1,get_lang_option(lang,'context-before')
1233       )
1234     .. word_orig ..
1235     get_context(
1236       hyphenation_index,1,get_lang_option(lang,'context-after')
1237     )

```

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

```

1238     hyphenation_count = hyphenation_count + 1
1239     hyphenation_list[hyphenation_count] = {
1240       lang = lang,
1241       word = word,
1242       divided = divided,
1243       index = hyphenation_count,
1244       context = context,
1245       status = status,
1246     }

```

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.

```

1247     late_lua_n = node.new('whatsit','late_lua')
1248     late_lua_n.data =
1249       'lualisthyphen.set_hyphenation_page('
1250       .. hyphenation_count .. ',tex.count["c@page"])'
1251
1252     node.insert_after(hlist.head,last_disc,late_lua_n)
1253
1254   end

```

(End of definition for check_line_hyphenation.)

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

```

1255     local function set_hyphenation_page(index,page)
1256
1257       hyphenation_list[index].page = page
1258
1259     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.

```

1260     local function post_linebreak(vlist_head,groupcode)
1261
1262       debug('Post-linebreak processing start')

```

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

```

1263   for item in node.traverse(vlist_head) do
1264       if item.id == NODE_ID_HLIST then
1265           node.slide(item.head)
1266       end
1267   end

```

Now actually look for hyphenations.

```

1268   local line_no = 0
1269
1270   for item in node.traverse(vlist_head) do
1271
1272       if item.id == NODE_ID_HLIST then
1273           line_no = line_no + 1
1274           debug(' Line no. %d',line_no)
1275           check_line_hyphenation(item)
1276       end
1277
1278   end
1279
1280   hlist_segment_list = {}
1281
1282   debug('Post-linebreak processing end')
1283
1284   return true
1285
1286 end

```

(End of definition for `post_linebreak`.)

11.17 Add callbacks

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

```

1287 local LUA_LIST_HYPHEN_PRE_LINEBREAK = 'LUA_LIST_HYPHEN_PRE_LINEBREAK'
1288 luatexbase.add_to_callback(
1289     'pre_linebreak_filter',
1290     pre_linebreak,
1291     LUA_LIST_HYPHEN_PRE_LINEBREAK
1292 )
1293
1294 local LUA_LIST_HYPHEN_POST_LINEBREAK = 'LUA_LIST_HYPHEN_POST_LINEBREAK'
1295 luatexbase.add_to_callback(
1296     'post_linebreak_filter',
1297     post_linebreak,
1298     LUA_LIST_HYPHEN_POST_LINEBREAK
1299 )

```

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.

```
1300 local function equal_hyphenation_case_sensitive(a,b)
1301   return (
1302     a.word == b.word
1303     and
1304     a.divided == b.divided
1305   )
1306 end
```

(End of definition for equal_hyphenation_case_sensitive.)

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

```
1307 local function equal_hyphenation_case_insensitive(a,b)
1308   return (
1309     unicode.utf8.lower(a.word) == unicode.utf8.lower(b.word)
1310     and
1311     unicode.utf8.lower(a.divided) == unicode.utf8.lower(b.divided)
1312   )
1313 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.

```
1314 local function lessthan_hyphenation_case_sensitive(a,b)
1315   return (
1316     a.word < b.word
1317     or
1318     (
1319       a.word == b.word
1320       and
1321       a.divided < b.divided
1322     )
1323     or
1324     (
1325       a.word == b.word
1326       and
1327       a.divided == b.divided
1328       and
1329       a.index < b.index
1330     )
1331   )
1332 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.

```
1333 local function lessthan_hyphenation_case_insensitive(a,b)
1334   return (
1335     unicode.utf8.lower(a.word) < unicode.utf8.lower(b.word)
1336     or
1337     (
```

```

1338         unicode.utf8.lower(a.word) == unicode.utf8.lower(b.word)
1339         and
1340         unicode.utf8.lower(a.divided) < unicode.utf8.lower(b.divided)
1341     )
1342     or
1343     (
1344         unicode.utf8.lower(a.word) == unicode.utf8.lower(b.word)
1345         and
1346         unicode.utf8.lower(a.divided) < unicode.utf8.lower(b.divided)
1347         and
1348         a.index < b.index
1349     )
1350 )
1351 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.

```

1352 local function sort_hyphenation_list_none(hyphenation_list)
1353     table.sort(
1354         hyphenation_list,
1355         function(a,b)
1356             return a.index < b.index
1357         end
1358     )
1359 end

```

(End of definition for sort_hyphenation_list_none.)

`sort_hyphenation_list_case` Sort `hyphenation_list` case-sensitively.

```

1360 local function sort_hyphenation_list_case(hyphenation_list)
1361     table.sort(
1362         hyphenation_list,
1363         lessthan_hyphenation_case_sensitive
1364     )
1365 end

```

(End of definition for sort_hyphenation_list_case.)

`sort_hyphenation_list_nocase` Sort `hyphenation_list` case-insensitively.

```

1366 local function sort_hyphenation_list_nocase(hyphenation_list)
1367     table.sort(
1368         hyphenation_list,
1369         lessthan_hyphenation_case_insensitive
1370     )
1371 end

```

(End of definition for sort_hyphenation_list_nocase.)

11.18.3 Deduplication

`deduplicate_hyphenation_list_case` Remove duplicates from `hyphenation_list` case-sensitively.

```
1372 local function deduplicate_hyphenation_list_case(hyphenation_list)
1373   table.sort(
1374     hyphenation_list,
1375     lessthan_hyphenation_case_sensitive
1376   )
1377   list_uniq(
1378     hyphenation_list,
1379     equal_hyphenation_case_sensitive
1380   )
1381 end
```

(End of definition for deduplicate_hyphenation_list_case.)

`deduplicate_hyphenation_list_nocase` Remove duplicates from `hyphenation_list` case-insensitively.

```
1382 local function deduplicate_hyphenation_list_nocase(hyphenation_list)
1383   table.sort(
1384     hyphenation_list,
1385     lessthan_hyphenation_case_insensitive
1386   )
1387   list_uniq(
1388     hyphenation_list,
1389     equal_hyphenation_case_insensitive
1390   )
1391 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`.

```
1392 local function process_lang_hyphenation_list(lang_id,hyphenation_list)
1393
1394   local output_non_typeset = get_lang_option(lang_id,'output-non-typeset')
1395   local include_valid = (get_lang_option(lang_id,'output-mode') == 0)
1396
```

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

```
1397   list_filter(
1398     hyphenation_list,
1399     function (a)
1400       return (
1401         (a.status ~= STATUS_VALID or include_valid)
1402         and
1403         (a.page or output_non_typeset)
1404       )
1405     end
1406   )
```

Deduplicate in accordance with the specified options for this language.

```

1407 local unique = get_lang_option(lang_id,'unique')
1408 if unique == 1 then
1409     deduplicate_hyphenation_list_case(hyphenation_list)
1410 elseif unique == 2 then
1411     deduplicate_hyphenation_list_nocase(hyphenation_list)
1412 end

```

Sort in accordance with the specified options for this language.

```

1413 local sort_mode = get_lang_option(lang_id,'sort')
1414 if sort_mode == 1 then
1415     sort_hyphenation_list_case(hyphenation_list)
1416 elseif sort_mode == 2 then
1417     sort_hyphenation_list_nocase(hyphenation_list)
1418 else
1419     sort_hyphenation_list_none(hyphenation_list)
1420 end
1421
1422 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`.

```

1423 local function write_lang_hyphenation_list_standard(f,hyphenation_list,widths)
1424
1425     for i,v in ipairs(hyphenation_list) do
1426
1427         if v then
1428             f:write(v.divided .. '\n')
1429         end
1430
1431     end
1432
1433 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.

```

1434 local function write_lang_hyphenation_list_verbose(f,hyphenation_list,show_status)

```

Calculate the number of columns required for each output field.

```

1435 local cols_word = 0
1436 local cols_divided = 0
1437 local cols_page = 0
1438
1439 for _,h in pairs(hyphenation_list) do
1440
1441     cols_word = math.max(
1442         cols_word,
1443         unicode.utf8.len(h.word)

```

```

1444     )
1445     cols_divided = math.max(
1446         cols_divided,
1447         unicode.utf8.len(h.divided)
1448     )
1449     cols_page = math.max(
1450         cols_page,
1451         unicode.utf8.len(tostring(h.page))
1452     )
1453
1454 end

```

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

```

1455     cols_page = math.max(
1456         cols_page + unicode.utf8.len(STR_PAGE_PREFIX),
1457         unicode.utf8.len(STR_PAGE_NONE)
1458     )
1459
1460     local cols_status = math.max(
1461         unicode.utf8.len(STR_VALID),
1462         unicode.utf8.len(STR_INVALID),
1463         unicode.utf8.len(STR_AMBIGUOUS),
1464         unicode.utf8.len(STR_UNKNOWN)
1465     ) + 2 -- Add for parentheses
1466
1467     for i,v in ipairs(hyphenation_list) do
1468
1469         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.

```

1470         local page = v.page
1471         if page then
1472             page = STR_PAGE_PREFIX .. page
1473         else
1474             page = STR_PAGE_NONE
1475         end

```

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

```

1476         local status
1477         if show_status then
1478             status = v.status
1479             if status == STATUS_VALID then
1480                 status = STR_VALID
1481             elseif status == STATUS_AMBIGUOUS then
1482                 status = STR_AMBIGUOUS
1483             elseif status == STATUS_INVALID then
1484                 status = STR_INVALID
1485             else
1486                 status = STR_UNKNOWN
1487             end
1488             status = rpad(parenthesize(status), cols_status) .. STR_SPACE_TWO
1489         else
1490             status = ''

```

```
1491         end
```

Write everything out on a single line.

```
1492         f:write(
1493             rpad(v.word,cols_word)
1494             .. STR_ARROW
1495             .. rpad(v.divided,cols_divided)
1496             .. STR_SPACE_TWO
1497             .. status
1498             .. lpad(page,cols_page)
1499             .. STR_SPACE
1500             .. STR_QUOTE_OPEN
1501             .. v.context
1502             .. STR_QUOTE_CLOSE
1503             .. '\n'
1504         )
1505     end
1506
1507 end
1508
1509 end
```

(End of definition for write_lang_hyphenation_list_verbose.)

get_settings_desc Compute a settings description to insert into file headers.

```
1510 local function get_settings_desc(lang_id)
1511
1512     local settings_desc
1513
1514     if get_lang_option(lang_id,'output-verbose') then
1515         settings_desc = string.format(
1516             'verbose=true,context-before=%d,context-after=%s',
1517             get_lang_option(lang_id,'context-before'),
1518             get_lang_option(lang_id,'context-after')
1519         )
1520     else
1521         settings_desc = 'verbose=false'
1522     end
1523
1524     local ALL_NONVALID = {
1525         [0] = 'all',
1526         [1] = 'non-valid',
1527     }
1528     settings_desc = settings_desc .. string.format(
1529         ',output-mode=%s',
1530         ALL_NONVALID[get_lang_option(lang_id,'output-mode')]
1531     )
1532     settings_desc = settings_desc .. string.format(
1533         ',include-non-typeset=%s',
1534         get_lang_option(lang_id,'output-non-typeset')
1535     )
1536     local NONE_CASE_NOCASE = {
1537         [0] = 'none',
1538         [1] = 'case',
1539         [2] = 'nocase'
```

```

1540 }
1541 settings_desc = settings_desc .. string.format(
1542     ',sort=%s,unique=%s',
1543     NONE_CASE_NOCASE[get_lang_option(lang_id,'sort')],
1544     NONE_CASE_NOCASE[get_lang_option(lang_id,'unique')]
1545 )
1546
1547 return settings_desc
1548
1549 end

```

(End of definition for get_settings_desc.)

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

```

1550 local function prefix_output_directory(path)
1551
1552     local output_directory = status.output_directory
1553     if not output_directory then
1554         return path
1555     end
1556
1557     if string.sub(output_directory,-1,-1) == '/' then
1558         return output_directory .. path
1559     else
1560         return output_directory .. '/' .. path
1561     end
1562
1563 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.

```

1564 local function process_write_lang_hyphenation_list(
1565     lang_id,hyphenation_list
1566 )
1567     process_lang_hyphenation_list(lang_id,hyphenation_list)
1568
1569     local output_path
1570     local lang_desc
1571
1572     local cname = lang_id_cname_table[lang_id]
1573
1574     debug('Processing and writing hyphenation lists for "%s" (lang. ID %d)',cname,lang_id)
1575
1576     if cname then
1577         output_path = get_lang_option(lang_id,'output-path')
1578         if not output_path then
1579             output_path = get_lang_option(LANG_DEFAULT,'output-prefix')
1580             .. cname .. get_lang_option(LANG_DEFAULT,'output-extension')
1581         end
1582         lang_desc = string.format('language "%s" (ID %s)',cname,lang_id)
1583     else

```

```

1584     output_path = get_lang_option(LANG_DEFAULT,'output-prefix')
1585     .. lang_id .. get_lang_option(LANG_DEFAULT,'output-extension')
1586     lang_desc = 'language with ID ' .. lang_id
1587 end
1588
1589 output_path = prefix_output_directory(output_path)
1590 debug('Output path: "%s"',output_path)
1591
1592 local output_header = get_lang_option(lang_id,'output-header')
1593 local output_verbose = get_lang_option(lang_id,'output-verbose')
1594
1595 local division_dict_read = lang_id_division_dict_read_table[lang_id]
1596
1597 local f = io.open(output_path,'w')
1598
1599 if output_header then
1600     f:write('% This file was generated by lua-list-hyphen\n')
1601     f:write('% Hyphenations for ' .. lang_desc .. '\n')
1602     f:write('% General settings: ' .. get_settings_desc(lang_id) .. '\n')
1603     local division_dict_path = lang_id_division_dict_path_table[lang_id]
1604     if division_dict_path and division_dict_read then
1605         f:write(
1606             '% Division dictionary read from: "' .. division_dict_path .. '"\n'
1607         )
1608     end
1609 end
1610
1611 if output_verbose then
1612     write_lang_hyphenation_list_verbose(f,hyphenation_list,division_dict_read)
1613 else
1614     write_lang_hyphenation_list_standard(f,hyphenation_list,division_dict_read)
1615 end
1616
1617 f:close()
1618
1619 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.

```

1620 local function process_write_hyphenation_lists()
1621
1622     local lang_hyphenation_table = {}
1623     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).

```

1624     for _,h in pairs(hyphenation_list) do
1625
1626         local lang = h.lang
1627
1628         local t = lang_hyphenation_table[lang]
1629         if not t then
1630             lang_hyphenation_table[lang] = {}
1631             t = lang_hyphenation_table[lang]

```

```

1632     end
1633
1634     table.insert(t,h)
1635
1636 end

```

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

```

1637 for lang_id,hyphenations in pairs(lang_hyphenation_table) do
1638     process_write_lang_hyphenation_list(lang_id,hyphenations)
1639 end
1640
1641 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`.

```

1642 lualisthyphen = lualisthyphen or {}
1643 lualisthyphen.process_write_hyphenation_lists = process_write_hyphenation_lists
1644 lualisthyphen.set_hyphenation_page = set_hyphenation_page
1645 lualisthyphen.store_lang_id_name = store_lang_id_name
1646
1647 lualisthyphen.write_flag_valid = write_flag_valid
1648 lualisthyphen.write_flag_invalid = write_flag_invalid
1649 lualisthyphen.write_flag_ambiguous = write_flag_ambiguous
1650 lualisthyphen.write_flag_unknown = write_flag_unknown
1651
1652 lualisthyphen.set_debug = set_debug
1653 lualisthyphen.set_global_option = set_global_option
1654 lualisthyphen.set_lang_option = set_lang_option
1655
1656 lualisthyphen.preprocessors = preprocessors
1657  $\langle$ /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>	<u>610</u>
A	
<code>allowlist-path</code> (option)	<u>8</u>
B	
<code>\begin</code>	<u>595</u> , <u>611</u>
bool commands:	
<code>\bool_to_str:N</code>	<u>152</u> , <u>181</u> , <u>187</u> , <u>189</u> , <u>191</u> , <u>209</u> , <u>215</u> , <u>217</u> , <u>219</u>
C	
check commands:	
<code>check_line_hyphenation</code>	<u>1191</u>
clist commands:	
<code>\clist_new:N</code>	<u>141</u>
<code>\clist_put_right:Nn</code>	<u>145</u>
<code>\clist_use:N</code>	<u>245</u>
<code>context</code> (option)	<u>10</u>
<code>context-after</code> (option)	<u>10</u>
<code>context-before</code> (option)	<u>10</u>
cs commands:	
<code>\cs_generate_variant:Nn</code>	<u>147</u> , <u>164</u> , <u>169</u> , <u>229</u>
<code>\cs_if_exist:NTF</code>	<u>255</u>
<code>\cs_new:Npn</code>	<u>143</u> , <u>150</u> , <u>154</u> , <u>160</u> , <u>165</u> , <u>200</u> , <u>249</u> , <u>253</u> , <u>265</u>
<code>\cs_set_eq:NN</code>	<u>258</u>
D	
<code>debug</code>	<u>298</u>
debug commands:	
<code>debug_dummy</code>	<u>298</u>
<code>debug_real</code>	<u>298</u>
deduplicate commands:	
<code>deduplicate_hyphenation_list_-case</code>	<u>1372</u>
<code>deduplicate_hyphenation_list_-nocase</code>	<u>1382</u>
delete commands:	
<code>delete_hyphens</code>	<u>426</u>
<code>division-dict-path</code> (option)	<u>8</u>
E	
<code>\en</code>	<u>593</u> , <u>609</u>
english commands:	
<code>english_strip_possessives</code>	<u>651</u>
equal commands:	
<code>equal_hyphenation_case_insensitive</code>	<u>1307</u>
<code>equal_hyphenation_case_sensitive</code>	<u>1300</u>
F	
<code>flag-mode</code> (option)	<u>9</u>
<code>\foreignlanguage</code>	<u>11</u>
G	
get commands:	
<code>get_context</code>	<u>1071</u>
<code>get_disc_lang</code>	<u>873</u>
<code>get_disc_word_start</code>	<u>1017</u>
<code>get_first_glyph_lang</code>	<u>860</u>
<code>get_lang_division_dict_validation_preprocessor</code>	<u>655</u>
<code>get_lang_option</code>	<u>536</u>
<code>get_line_first_word</code>	<u>1047</u>
<code>get_next_hlist</code>	<u>1033</u>
<code>get_node_text</code>	<u>781</u>
<code>get_nodelist_text</code>	<u>822</u>
<code>get_settings_desc</code>	<u>1510</u>
<code>get_used_disc</code>	<u>997</u>
group commands:	
<code>\group_begin:</code>	<u>233</u> , <u>257</u>
<code>\group_end:</code>	<u>241</u> , <u>262</u>
H	
hook commands:	
<code>\hook_gput_code:nnn</code>	<u>246</u> , <u>271</u>
<code>\hyphenation</code>	<u>6</u>
hyphenation commands:	
<code>hyphenation_matches_pattern</code>	<u>436</u>
I	
<code>identity</code>	<u>647</u>
<code>info</code>	<u>289</u>
insert commands:	
<code>insert_flag</code>	<u>1147</u>
int commands:	
<code>\int_new:N</code>	<u>33</u> , <u>39</u> , <u>65</u> , <u>71</u>
<code>\int_set:Nn</code>	<u>36</u> , <u>42</u> , <u>68</u> , <u>74</u>
<code>\int_use:N</code>	<u>183</u> , <u>185</u> , <u>193</u> , <u>195</u> , <u>197</u> , <u>199</u> , <u>211</u> , <u>213</u> , <u>221</u> , <u>223</u> , <u>225</u> , <u>227</u>
is commands:	
<code>is_possible_space_node</code>	<u>851</u>
<code>is_possible_word_node</code>	<u>838</u>

J	
<code>\jobname</code>	5, 8
K	
keys commands:	
<code>\l_keys_choice_int</code>	36, 42, 68, 74
<code>\keys_define:nn</code>	12, 16, 20, 23, 26, 29, 34, 40, 45, 48, 52, 55, 66, 72, 77, 80, 83, 89, 98, 111, 133, 136, 231
<code>\l_keys_key_str</code>	92, 101, 105, 114, 119, 124, 129, 138, 235
<code>\keys_set:nn</code> 115, 120, 125, 130, 236, 244	
<code>\keys_set_known:nn</code>	18
L	
lessthan commands:	
<code>lessthan_hyphenation_case_-insensitive</code>	1333
<code>lessthan_hyphenation_case_-sensitive</code>	1314
list commands:	
<code>list_filter</code>	340
<code>list_uniq</code>	357
load commands:	
<code>load_division_dict</code>	588
<code>lpad</code>	416
lua commands:	
<code>\lua_load_module:n</code>	148
<code>\lua_now:n</code> . 59, 152, 162, 167, 267, 272	
lualisthyphen internal commands:	
<code>__lualisthyphen_babel_store_-lang_id_names:</code>	251, 253, 253
<code>__lualisthyphen_babel_store_-lang_id_names_elt:nnnn</code>	260, 265, 265
<code>__lualisthyphen_call_lua_-boolean:nN</code>	150, 150, 170
<code>\l__lualisthyphen_context_after_-int</code>	55, 195, 223
<code>\l__lualisthyphen_context_-before_int</code>	55, 193, 221
<code>\l__lualisthyphen_debug_bool</code> 133, 171	
<code>\l__lualisthyphen_division_dict_-path_str</code>	20, 173, 203
<code>\l__lualisthyphen_flag_mode_int</code>	33, 183, 211
<code>__lualisthyphen_initialize:</code> ...	247, 249, 249
<code>\l__lualisthyphen_lang_name_str</code>	21, 230, 235, 239
<code>\l__lualisthyphen_lang_options_-clist</code>	18, 19, 141, 145, 245
<code>__lualisthyphen_lua_set_global_-option:nn</code>	160, 160, 164, 172, 174, 176, 178, 180, 182, 184, 186, 188, 190, 192, 194, 196, 198
<code>__lualisthyphen_lua_set_lang_-option:nnn</code>	165, 165, 169, 202, 204, 206, 208, 210, 212, 214, 216, 218, 220, 222, 224, 226
<code>__lualisthyphen_lua_set_per_-lang_options:n</code> ..	200, 200, 229, 238
<code>__lualisthyphen_luastr_or_nil:N</code>	154, 154, 173, 175, 177, 179, 203, 205, 207
<code>\l__lualisthyphen_output_-extension_str</code>	16, 177
<code>\l__lualisthyphen_output_header_-bool</code>	48, 189, 217
<code>\l__lualisthyphen_output_mode_-int</code>	39, 185, 213
<code>\l__lualisthyphen_output_non_-typeset_bool</code>	45, 187, 215
<code>\l__lualisthyphen_output_path_-str</code>	83, 205
<code>\l__lualisthyphen_output_prefix_-str</code>	12, 175
<code>\l__lualisthyphen_output_-verbose_bool</code>	52, 191, 219
<code>\l__lualisthyphen_sort_int</code>	71, 199, 227
<code>__lualisthyphen_store_lang_-options:nn</code>	138, 141, 143, 147
<code>\l__lualisthyphen_unique_int</code> ...	65, 197, 225
<code>\l__lualisthyphen_validation_-case_sensitive_bool</code> .	29, 181, 209
<code>\l__lualisthyphen_validation_-preprocessor_str</code>	26, 179, 207
M	
make commands:	
<code>make_flag_node</code>	1105
msg commands:	
<code>\msg_critical:nn</code>	10
<code>\msg_new:nnn</code>	8
<code>\msg_new:nnnn</code>	86, 95, 108
<code>\msg_warning:nnn</code>	91, 100, 104
<code>\msg_warning:nnnn</code> .	113, 118, 123, 128
N	
<code>\n</code>	1428, 1503, 1600, 1601, 1602, 1606
<code>\NeedsTeXFormat</code>	3
O	
options:	
<code>allowlist-path</code>	8
<code>context</code>	10

