

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

Alan J. Cain[†]

Released 2026-07-23

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 v0.42.1, last revised 2026-07-23.

[†]a.j.cain (AT) gmail.com

Contents

1	Introduction	3
2	Requirements	4
3	Installation	5
4	Getting started	5
5	Classifications of hyphenations	5
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	11
9.1	LuaLaTeX source	11
9.2	Division dictionary	12
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

$\text{\TeX}$'s algorithm for finding points where a word can be hyphenated is good, but not perfect.¹ The present author writes in British English, where the valid division points can depend on both the pronunciation of a word and its internal structure (and hence its etymology). Currently, $\text{\TeX}$'s pattern-based approach produces *bio-lo-gic*, *bio-logy*, *bio-lo-gist*, rather than the standard *bio-logic*, *biol-ogy*, *biolo-gist*.² To deal with such cases, 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

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

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

at-trib-ute, *pre-sent*, *pro-duce*, *re-cord* have different division points from the orthographically identical nouns *at-tri-bute*, *pres-ent*, *prod-uce*, *rec-ord*. For another example, compare *cur-ric-ulum vitae* and *school cur-ricu-lum*.

Easy checking of the chosen hyphenations is desirable. With 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

2 Requirements

`lua-list-hyphen` requires

- (1) Lua $\text{\TeX}$,
- (2) a recent $\text{\LaTeX}$ kernel with `expl3` support (any kernel version since 2020-02-02 should suffice).

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.

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_{TEX} 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_{TEX} can find them.

4 Getting started

Simply load the package with `\usepackage{lua-list-hyphen}`. By default, `lua-list-hyphen` 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_{TEX} 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⟩`, where `⟨lang-name⟩` can be any of the synonyms for a language listed in the `hyph-utf8` documentation.

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

See [Section 5](#) for a full account of classification of hyphenations, [Subsection 7.2](#) for further details of `division-dict-path` description, and [Subsection 7.5](#) for `output-mode`.

- To add visual flags for hyphenations to the output PDF, add the package option `flags=all` or `flags=non-valid`. See [Subsection 7.4](#) for further details.

See [Section 7](#) for all the options.

5 Classifications of hyphenations

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

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

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

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

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

invalid A hyphenation is *invalid* if there is an entry in the division dictionary for that word and the chosen hyphenation matches *no* entry for that word. Using the example lines from the division dictionary above, *attrib-ution* is invalid because it does not match the entry `at-tri-bu-tion`.

ambiguous A hyphenation is *ambiguous* if there 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,
  },
]{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.

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 for a language when no path has been specified in **output-paths**. 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 the `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>` 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-paths The key **division-dict-paths** 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. It takes a comma-separated list of key-value pairs of the form `<lang-name>=<path>`. Here, the `<lang-name>` can be any of the synonyms for the language specified in the `hyph-utf8` documentation. If more than one `<path>` is specified for the same language (for instance, using different synonyms as `<lang-name>`), later ones override earlier ones. (*Default: None*)

Paths specified using this key are 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`).

7.3 Hyphenation checking

validation-preprocessor The key **validation-preprocessor** can be used to specify, for each language, a Lua function that is applied to each hyphenated word before it is checked against the division dictionary. It takes a comma-separated list of key-value pairs of the form `<lang-name>=<function>`. Here, the `<lang-name>` can be any of the synonyms for the language specified in the `hyph-utf8` documentation and `<function>` should be 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* (in British English) 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

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

9.1 LuaLaTeX source

```

\documentclass[twocolumn,oneside]{book}

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

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

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

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

\begin{document}

```

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

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

10 Implementation (L^AT_EX package)

```

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

```

10.1 Initial set-up

Package identification/version information.

```

3 \NeedsTeXFormat{LaTeX2e}[2020-02-02]
4 \ProvidesExplPackage{lua-list-hyphen}{2026-07-23}{0.42.1}
5   {Listing hyphenated words for LuaLaTeX}

```

Check that Lua_TE_X is in use.

```

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

```

10.2 Options

10.2.1 Global-only options

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

```

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

```

(End of definition for \l__lualisthyphen_output_prefix_str.)

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

```

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

```

(End of definition for \l__lualisthyphen_output_extension_str.)

10.2.2 Per-language and global options

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

```

20 \keys_define:nn { lua-list-hyphen/global-per-lang }{
21   division-dict-path .str_set:N = \l__lualisthyphen_division_dict_path_str,
22 }

```

(End of definition for \l__lualisthyphen_division_dict_path_str.)

`\l__lualisthyphen_validation_preprocessor_str` String option to specify validation preprocessor.

```

23 \keys_define:nn { lua-list-hyphen/global-per-lang }{
24   validation-preprocessor .str_set:N = \l__lualisthyphen_validation_preprocessor_str,
25 }

```

(End of definition for \l__lualisthyphen_validation_preprocessor_str.)

<code>\lualisthyphen_validation_case_sensitive_bool</code>	Boolean option to specify whether validation of divisions is case-sensitive. <pre> 26 \keys_define:nn { lua-list-hyphen/global-per-lang }{ 27 validation-case-sensitive .bool_set:N 28 = \l__lualisthyphen_validation_case_sensitive_bool 29 } </pre> <i>(End of definition for \l__lualisthyphen_validation_case_sensitive_bool.)</i>
<code>\l__lualisthyphen_flag_mode_int</code>	Choice option to specify whether flags are added to the output PDF file to indicate the classification of hyphenations. <pre> 30 \int_new:N\l__lualisthyphen_flag_mode_int 31 \keys_define:nn { lua-list-hyphen/global-per-lang }{ 32 flag-mode .choices:nn = { none, all, non-valid }{ 33 \int_set:Nn\l__lualisthyphen_flag_mode_int{ \l_keys_choice_int - 1 } 34 }, 35 } </pre> <i>(End of definition for \l__lualisthyphen_flag_mode_int.)</i>
<code>\l__lualisthyphen_output_mode_int</code>	Choice option to indicate which hyphenations should be listed in the output files. <pre> 36 \int_new:N\l__lualisthyphen_output_mode_int 37 \keys_define:nn { lua-list-hyphen/global-per-lang }{ 38 output-mode .choices:nn = { all, non-valid }{ 39 \int_set:Nn\l__lualisthyphen_output_mode_int{ \l_keys_choice_int - 1 } 40 }, 41 } </pre> <i>(End of definition for \l__lualisthyphen_output_mode_int.)</i>
<code>\l__lualisthyphen_output_non_typeset_bool</code>	Boolean option to indicate whether output lists of hyphenations should include those that are never typeset to the page. <pre> 42 \keys_define:nn { lua-list-hyphen/global-per-lang }{ 43 output-non-typeset .bool_set:N = \l__lualisthyphen_output_non_typeset_bool, 44 } </pre> <i>(End of definition for \l__lualisthyphen_output_non_typeset_bool.)</i>
<code>\l__lualisthyphen_output_header_bool</code>	Boolean option to indicate whether a header should be written in the output. <pre> 45 \keys_define:nn { lua-list-hyphen/global-per-lang }{ 46 output-header .bool_set:N = \l__lualisthyphen_output_header_bool, 47 output-header .initial:n = {true}, 48 } </pre> <i>(End of definition for \l__lualisthyphen_output_header_bool.)</i>
<code>\l__lualisthyphen_output_verbose_bool</code>	Boolean option to indicate whether output lists of hyphenations should be written verbosely. <pre> 49 \keys_define:nn { lua-list-hyphen/global-per-lang }{ 50 output-verbose .bool_set:N = \l__lualisthyphen_output_verbose_bool, 51 } </pre> <i>(End of definition for \l__lualisthyphen_output_verbose_bool.)</i>

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

```

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

```

(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.

```

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

```

(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.

```

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

```

(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.

```

74 \keys_define:nn { }{
75   lua-list-hyphen .inherit:n = lua-list-hyphen/global-per-lang,
76 }

```

10.2.3 Per-language options

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

```

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

```

There is one option that is strictly per-language:

```

\l__lualisthyphen_output_path_str Option to specify the path to write hyphenations.
80 \keys_define:nn { lua-list-hyphen/per-lang }{
81   output-path .str_set:N = \l__lualisthyphen_output_path_str,
82 }

```

(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.

```

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

```

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

```

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

```

10.3 Deprecated options

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

```

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

```

```

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

```

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.

```

130 \keys_define:nn { lua-list-hyphen }{
131     debug .bool_set:N = \l__lualisthyphen_debug_bool,
132 }

```

(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.

```

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

```

`\__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`.

```

138 \clist_new:N\l__lualisthyphen_lang_options_clist
139
140 \cs_new:Npn \__lualisthyphen_store_lang_options:nn #1#2
141 {
142     \clist_put_right:Nn\l__lualisthyphen_lang_options_clist{#1={#2}}
143 }
144 \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.

```

145 \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`.

```
146 \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.

```
147 \cs_new:Npn \__lualisthyphen_call_lua_boolean:nN #1#2
148 {
149   \lua_now:ef lualisthyphen.#1(\bool_to_str:N #2) }
150 }
```

(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.

```
151 \cs_new:Npn \__lualisthyphen_luastr_or_nil:N #1
152 {
153   \str_if_empty:NTF #1
154     { nil }
155     { '\str_use:N #1' }
156 }
```

(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.

```
157 \cs_new:Npn \__lualisthyphen_lua_set_global_option:nn #1#2
158 {
159   \lua_now:n{ lualisthyphen.set_global_option('#1',#2) }
160 }
161 \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.

```
162 \cs_new:Npn \__lualisthyphen_lua_set_lang_option:nnn #1#2#3
163 {
164   \lua_now:n{ lualisthyphen.set_lang_option(#1,'#2',#3) }
165 }
166 \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.

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

At this point the variables contain ‘global’ options.

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

```
197 \cs_new:Npn \__lualisthyphen_lua_set_per_lang_options:n #1
198   {
199     \__lualisthyphen_lua_set_lang_option:nne{#1}{division-dict-path}
200     {\__lualisthyphen_luastr_or_nil:N\l__lualisthyphen_division_dict_path_str}
201     \__lualisthyphen_lua_set_lang_option:nne{#1}{output-path}
202     {\__lualisthyphen_luastr_or_nil:N\l__lualisthyphen_output_path_str}
203     \__lualisthyphen_lua_set_lang_option:nne{#1}{validation-preprocessor}
204     {\__lualisthyphen_luastr_or_nil:N\l__lualisthyphen_validation_preprocessor_str}
205     \__lualisthyphen_lua_set_lang_option:nne{#1}{validation-case-sensitive}
206     {\bool_to_str:N\l__lualisthyphen_validation_case_sensitive_bool}
207     \__lualisthyphen_lua_set_lang_option:nne{#1}{flag-mode}
208     {\int_use:N\l__lualisthyphen_flag_mode_int}
209     \__lualisthyphen_lua_set_lang_option:nne{#1}{output-mode}
210     {\int_use:N\l__lualisthyphen_output_mode_int}
```

```

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

```

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

```

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

```

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

```

10.6 Initialization

At `begindocument`, run initialization code.

```

243 \hook_gput_code:nnn{ begindocument }{ lua-list-hyphen }{
244   \__lualisthyphen_initialize:
245 }

```

`\__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.)

```

246 \cs_new:Npn \__lualisthyphen_initialize:
247 {

```

```

248     \__lualisthyphen_babel_store_lang_id_names:
249 }

```

(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.

```

250 \cs_new:Npn \__lualisthyphen_babel_store_lang_id_names:
251 {
252     \cs_if_exist:NT\bbl@languages
253     {
254         \group_begin:
255         \cs_set_eq:NN
256             \bbl@elt
257             \__lualisthyphen_babel_store_lang_id_names_elt:nmmn
258             \bbl@languages
259         \group_end:
260     }%
261 }

```

(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.

```

262 \cs_new:Npn \__lualisthyphen_babel_store_lang_id_names_elt:nmmn #1#2#3#4
263 {
264     \lua_now:n{
265         lualisthyphen.store_lang_id_name(#2,'#1')
266     }
267 }

```

(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.

```

268 \hook_gput_code:nnn{ enddocument/info }{ lua-list-hyphen } {
269     \lua_now:n{
270         lualisthyphen.process_write_hyphenation_lists()
271     }
272 }
273 </package>

```

11 Implementation (Lua backend)

```

274 <lua>

```

11.1 Initial set-up

Module identification/version information.

```

275 local module_name = 'lualisthyphen'
276 local lualisthyphen_module = {

```

```

277     name      = module_name,
278     version   = '0.42.1',
279     date      = '2026/07/23',
280     description = 'lua-list-hyphen',
281     author    = 'Alan J. Cain',
282     copyright  = 'Alan J. Cain',
283     license    = 'LPPL v1.3c'
284 }
285 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.

```

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

```

295 local function debug_dummy(s,...)
296 end
297
298 local function debug_real(s,...)
299     print(module_name .. ' DEBUG: ' .. s:format(...))
300 end
301
302 local debug = debug_dummy
303 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**.

```

304 local function set_debug(a)
305     if a then
306         debug = debug_real
307     else
308         debug = debug_dummy
309     end
310     debug_bool = a
311 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.

```
312 local NODE_ID_HLIST = node.id('hlist')
313 local NODE_ID_DISC = node.id('disc')
314 local NODE_ID_GLUE = node.id('glue')
315 local NODE_ID_KERN = node.id('kern')
316 local NODE_ID_MARGIN_KERN = node.id('margin_kern')
317 local NODE_ID_GLYPH = node.id('glyph')
318 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.)

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

Define constants for the math node subtypes.

```
328 local NODE_MATH_SUBTYPE_BEGIN
329 local NODE_MATH_SUBTYPE_END
330 for k,v in pairs(node.subtypes('math')) do
331   if v == 'beginmath' then
332     NODE_MATH_SUBTYPE_BEGIN = k
333   elseif v == 'endmath' then
334     NODE_MATH_SUBTYPE_END = k
335   end
336 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.)⁸

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

⁸Code adapted from <https://stackoverflow.com/a/53038524>.

```

350     end
351 end
352
353 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.

```

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

```

(End of definition for list_uniq.)

11.5 String manipulation functions

String constants (which will ultimately be output).

```

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

```

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

```

383 local function trim_whitespace_both(s)
384
385   return unicode.utf8.match(s, '^%s*(.-)%s*$')
386
387 end

```

(End of definition for trim_whitespace_both.)

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

```
388 local function trim_whitespace_left(s)
389
390   return unicode.utf8.match(s, '^%s*(.*)$')
391
392 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.

```
393 local function trim_nonlettershyphens_both(s)
394
395   return unicode.utf8.match(s, '^[^%a-]*(.-)[^%a-]*$')
396
397 end
```

(End of definition for trim_nonlettershyphens_both.)

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

```
398 local function trim_nonlettershyphens_start(s)
399
400   return unicode.utf8.match(s, '^[^%a-]*(.-)$')
401
402 end
```

(End of definition for trim_nonlettershyphens_start.)

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

```
403 local function trim_nonlettershyphens_end(s)
404
405   return unicode.utf8.match(s, '^(.-)[^%a-]*$')
406
407 end
```

(End of definition for trim_nonlettershyphens_end.)

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

```
408 local function rpadd(s,n)
409
410   return s .. unicode.utf8.rep(STR_SPACE,n - unicode.utf8.len(s))
411
412 end
```

(End of definition for rpadd.)

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

```
413 local function lpadd(s,n)
414
415   return unicode.utf8.rep(STR_SPACE,n - unicode.utf8.len(s)) .. s
416
417 end
```

(End of definition for lpad.)

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

```
418 local function parenthesize(s)
419
420   return '(' .. s .. ')'
421
422 end
```

(End of definition for parenthesize.)

11.6 Hyphen utility functions

`delete_hyphens` Delete all hyphens.

```
423 local function delete_hyphens(s)
424
425   return unicode.utf8.gsub(s, '-', '')
426
427 end
```

(End of definition for delete_hyphens.)

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

```
428 local function uniformize_hyphens(s)
429
430   return unicode.utf8.gsub(unicode.utf8.gsub(s, '|', '-'), '!', '-')
431
432 end
```

(End of definition for uniformize_hyphens.)

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

```
433 local function hyphenation_matches_pattern(hyphenation, pattern)
434
435   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.

```
436   local i = 1
437   for j=1,#pattern do
438
439     local s
440     local t
441
442     s = string.sub(hyphenation, i, i)
443     t = string.sub(pattern, j, j)
444
445     if s == t then
446       i = i + 1
447     elseif s == '-' or t ~= '-' then
```

```

448         debug('      Fails at i=%d,s=%s,j=%d,t=%s',i,s,j,t)
449         return false
450     end
451
452 end
453
454 debug('      Matches')
455 return true
456
457 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.

```

458 local lang_name_cname_table = {}
459 local lang_cname_name_list_table = {}

```

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

```

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

```

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

```

484 local lang_id_cname_table = {}
485 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).

```

486 local function store_lang_id_name(lang_id,name)
487
488     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).
489     if cname == nil then
490         cname = name
491         lang_name_cname_table[name] = cname
492         lang_cname_name_list_table[cname] = { name }
493     end
494
495     local lang_id_cname = lang_id_cname_table[lang_id]
496     if lang_id_cname == cname then
497         debug(
498             'Language name "%s" is a synonym for cname "%s" (lang. ID %s)',
499             name,cname,lang_id
500         )
501         return
502     elseif lang_id_cname ~= nil then
503         error(
504             'Languages with canonical names "%s" and "%s" both have ID %s',
505             cname,lang_id_cname,lang_id
506         )
507         return
508     end
509
510     lang_id_cname_table[lang_id] = cname
511     lang_cname_id_table[cname] = lang_id
512
513     debug('Language ID %s has cname "%s"',lang_id,cname)
514
515 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.

```

516 local function polyglossia_store_lang_id_name(lang_id)
517
518     if not polyglossia then
519         return
520     end
521
522     for name,language in pairs(polyglossia.newloader_loaded_languages) do
523         local this_lang_id = lang.id(language)
524         if this_lang_id == lang_id then
525             store_lang_id_name(lang_id,name)
526         end
527     end
528
529 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.

```
530 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.

```
531 local lang_id_options_table = {}
```

```
532 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.

```
533 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`.

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

```
535
```

```
536   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.

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

```
538     if cname then
```

```
539       options = lang_cname_options_table[cname]
```

```
540     end
```

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

```
541     if not options then
```

```
542       options = {}
```

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

```
544         options[k] = v
```

```
545       end
```

```
546     end
```

```
547     lang_id_options_table[lang_id] = options
```

```
548   end
```

```
549
```

```
550   return options[key]
```

```
551
```

```
552 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`.

```

553 local function set_global_option(key,value)
554
555   local options = lang_id_options_table[LANG_DEFAULT]
556   if not options then
557     options = {}
558     lang_id_options_table[LANG_DEFAULT] = options
559   end
560
561   options[key] = value
562   debug('Set global option %s="%s"',key,value)
563
564 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.

```

565 local function set_lang_option(name,key,value)
566
567   local cname = lang_name_cname_table[name]
568   if not cname then
569     cname = name
570   end
571
572   local options = lang_cname_options_table[cname]
573   if not options then
574     options = {}
575     lang_cname_options_table[cname] = options
576   end
577
578   options[key] = value
579   debug('Set lang. "%s" option %s="%s"',cname,key,value)
580
581 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 `!`.

```

582 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.

```

583 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.

```

584 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.

```

585 local function load_division_dict(path,case_sensitive)
586
587     local count = 0
588
589     local f = io.open(path,'r')
590     % \en{macrocode}
591     % If the file was not found, return immediately.
592     % \begin{macrocode}
593     if f == nil then
594         return nil
595     end
596
597     debug(' Loading divisions from "%s"',path)
598
599     local division_dict = {}
600
601     io.input(f)
602
603     for line in io.lines() do
604
605         line = trim_whitespace_left(line)
606         % \en{macrocode}
607         % Lines in which the first non-whitespace character is \texttt{\%} are comments.
608         % \begin{macrocode}
609         if #line == 0 or string.sub(line,1,1) == '%' then
610             goto continue
611         end
612
613         count = count + 1
614
615         local pattern = unicode.utf8.match(line,'^(.)%s') or line
616
617         pattern = uniformize_hyphens(pattern)
618
619         if not case_sensitive then
620             pattern = unicode.utf8.lower(pattern)
621         end
622
623         local word = delete_hyphens(pattern)
624
625         local divisions = division_dict[word]
626         if divisions == nil then
627             divisions = {}
628             division_dict[word] = divisions
629         end
630
631         table.insert(divisions,pattern)
632

```

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

```

632     ::continue::
633 end
634
635 io.close(f)
636
637 debug('Read %s divisions from "%s"',count,path)
638
639 return division_dict
640
641 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.

```

642 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.

```

643 local preprocessors = {}

```

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

```

644 local function identity(s)
645     return s
646 end
647 preprocessors.identity = identity

```

(End of definition for identity.)

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

```

648 local function english_strip_possessives(s)
649     return string.gsub(s, 's$', '')
650 end
651 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.

```

652 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'.

```

653 local division_dict = lang_id_division_dict_table[lang_id]
654 if division_dict ~= nil then
655     return division_dict, lang_id_validation_preprocessor_table[lang_id]
656 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.

```

657 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`.

```

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

```

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

```

```

693     lang_id_division_dict_path_table[lang_id] = p
694
695     local case_sensitive = get_lang_option(lang_id,'validation-case-sensitive')
696
697     division_dict = load_division_dict(p,case_sensitive)
698     if division_dict ~= nil then
699         info(
700             '   Loaded division dict for "%s" (lang. ID %s) from "%s"',
701             cname,lang_id,path
702         )
703         lang_id_division_dict_read_table[lang_id] = true
704         lang_id_division_dict_table[lang_id] = division_dict
705         return division_dict,preprocessor
706     end
707
708     else
709         warning(
710             '   Specified division dict "%s" not found for "%s" (lang. ID %s)',
711             file_name,cname,lang_id
712         )
713     end
714     else
715         debug('   No division dict path specified')
716     end
717
718     lang_id_division_dict_read_table[lang_id] = false
719     division_dict = {}
720     lang_id_division_dict_table[lang_id] = division_dict
721
722     return division_dict,preprocessor
723
724 end

```

(End of definition for get_lang_division_dict_validation_preprocessor.)

11.12 Validating a hyphenation

Constants for the status of a hyphenation

```

725 local STATUS_VALID = 1
726 local STATUS_INVALID = 2
727 local STATUS_AMBIGUOUS = 3
728 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.

```

729 local function validate_division(lang_id,word,divided)
730
731     debug(
732         '   Validating "%s" for language ID %s',divided,lang_id
733     )
734     local division_dict,preprocessor
735         = get_lang_division_dict_validation_preprocessor(lang_id)

```

Use the preprocessor on the original and divided word.

```

736 word = preprocessor(word)
737 divided = preprocessor(divided)
738
739 if not get_lang_option(lang_id,'validation-case-sensitive') then
740     word = unicode.utf8.lower(word)
741     divided = unicode.utf8.lower(divided)
742 end
743
744 local divisions = division_dict[word]
745 if not divisions then
746     return STATUS_UNKNOWN
747 end
748
749 local count = 0
750
751 for _,division in ipairs(divisions) do
752     if hyphenation_matches_pattern(divided,division) then
753         count = count + 1
754     end
755 end
756
757 if count == 0 then
758     return STATUS_INVALID
759 elseif count == #divisions then
760     return STATUS_VALID
761 else
762     return STATUS_AMBIGUOUS
763 end
764
765 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.

```

766 local LIGATURE_TEXT = {
767     [0xfb00] = 'ff',
768     [0xfb01] = 'fi',
769     [0xfb02] = 'fl',
770     [0xfb03] = 'ffi',
771     [0xfb04] = 'ffl',
772     [0xfb05] = 'ft',
773     [0xfb06] = 'st',
774 }

```

Cache to save table lookups when extracting text.

```

775 local font_characters = {}

```

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

```

776 local get_node_text
777 local get_nodelist_text

```

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

```

778 get_node_text = function(n)
779
780   if n.id == NODE_ID_GLYPH then
781
782     local ligature_text = LIGATURE_TEXT[n.char]
783     if ligature_text ~= nil then
784       return ligature_text
785     elseif n.components then
786       return get_nodelist_text(n.components)
787     else
788       -- See [https://tug.org/pipermail/luatex/2018-March/006786.html]
789       local characters = font_characters[n.font]
790       if not characters then
791         characters = fonts.hashes.identifiers[n.font].characters
792         font_characters[n.font] = characters
793       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.

```

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

```

819 get_nodelist_text = function (head,last)
820

```

```

821  local text = ''
822
823  for item in node.traverse(head) do
824
825      text = text .. get_node_text(item)
826
827      if item == last then
828          break
829      end
830  end
831
832  return text
833
834 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**.

```

835 local function is_possible_word_node(n)
836
837  return (
838      n.id == NODE_ID_GLYPH
839      or
840      n.id == NODE_ID_DISC
841      or
842      (n.id == NODE_ID_KERN and n.subtype == NODE_KERN_SUBTYPE_FONTKERN)
843      or
844      n.id == NODE_ID_MARGIN_KERN
845  )
846
847 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**.

```

848 local function is_possible_space_node(n)
849
850  return (
851      n.id == NODE_ID_GLUE
852      or
853      (n.id == NODE_ID_KERN and n.subtype == NODE_KERN_SUBTYPE_USERKERN)
854  )
855
856 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`.)

```

857 -- local function get_first_glyph_lang(n)
858
859 --   local item = n
860 --   while item and is_possible_word_node(item) do
861 --     if item.id == NODE_ID_GLYPH then
862 --       return item.lang
863 --     end
864 --     item = item.next
865 --   end
866
867 --   return nil
868
869 -- 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`.)

```

870 local function get_disc_lang(n)
871
872 -- lang = get_first_glyph_lang(n.replace)
873 -- if lang then
874 --   return lang
875 -- end
876
877 -- lang = get_first_glyph_lang(n.pre)
878 -- if lang then
879 --   return lang
880 -- end
881
882 -- lang = get_first_glyph_lang(n.post)
883 -- if lang then
884 --   return lang
885 -- end
886
887 local item

```

Before the disc node.

```

888 item = n
889 while item and is_possible_word_node(item) do
890   if item.id == NODE_ID_GLYPH then
891     return item.lang
892   end
893   item = item.prev
894 end

```

After the disc node.

```

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

```

906 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.

```

907 local hlist_segment_list = {}

```

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

```

908 local SEGMENT_WORD = 0
909 local SEGMENT_SPACE = 1
910 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).

```

911 local function pre_linebreak(hlist_head,groupcode)
912
913   local word_start_node = nil
914   local segment_count = 0
915   local lang = nil
916
917   debug('Pre-linebreak processing start')
918

```

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

```

919   node.slide(hlist_head)
920
921   local item = hlist_head
922   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`.

```

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

```

939     if is_possible_word_node(item) then
940         word_start_node = item
941         segment_count = segment_count + 1
942         while item and is_possible_word_node(item) do
943

```

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

```

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

```

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

```

```

966         word_start_node = nil
967         lang = nil
968
969         goto continue
970     end
971
972     If item is a node that could be part of a space, add a space to the segment list.
973     if is_possible_space_node(item) then
974         segment_count = segment_count + 1
975
976         while item and is_possible_space_node(item) do
977             item = item.next
978         end
979
980         hlist_segment_list[segment_count] = {
981             type = SEGMENT_SPACE
982         }
983
984         goto continue
985     end
986
987     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.

```

994 local USED_DISC_SEARCH_LIMIT = 20
995
996 local function get_used_disc(hlist_head)
997
998     debug(' Looking for disc node at end of line')
999
1000    local item = node.tail(hlist_head)

```

```

1001
1002   local count = 0
1003   while item and item.id ~= NODE_ID_GLYPH and count < USED_DISC_SEARCH_LIMIT do
1004       if item.id == NODE_ID_DISC then
1005           return item
1006       end
1007       item = item.prev
1008       count = count + 1
1009   end
1010
1011   return nil
1012
1013 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.

```

1014 local function get_disc_word_start(hlist_head,n)
1015
1016   local item = n
1017
1018   while item do
1019       local prev = item.prev
1020
1021       if not (prev and is_possible_word_node(prev)) then
1022           return item
1023       end
1024
1025       item = prev
1026   end
1027
1028   return nil
1029 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.

```

1030 local function get_next_hlist(n)
1031
1032   local item = n.next
1033
1034   while item do
1035       if item.id == NODE_ID_HLIST then
1036           return item
1037       end
1038       item = item.next
1039   end
1040
1041   return nil
1042
1043 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.

```
1044 local function get_line_first_word(hlist_head)
word_start_node is either nil or the (glyph) node that starts the word.
1045   local word_start_node = nil
1046
1047   for item in node.traverse(hlist_head) do
1048
1049     if item.id == NODE_ID_GLYPH then
1050       if not word_start_node then
1051         word_start_node = item
1052       end
1053     end
1054
1055     if not is_possible_word_node(item) then
1056       if word_start_node then
1057         return get_nodelist_text(word_start_node,item.prev)
1058       end
1059     end
1060
1061   end
```

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

```
1062   if word_start_node then
1063     return get_nodelist_text(word_start_node,node.tail(hlist_head))
1064   else
1065     return nil
1066   end
1067 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.

```
1068 local function get_context(index,incr,target_word_count)
1069
1070   local result = ''
1071   local word_count = 0
1072
1073   local i = index + incr
1074   while (
1075     i > 0 and i <= #hlist_segment_list and word_count < target_word_count
1076   ) do
1077     local t = hlist_segment_list[i]
1078
1079     local item
1080
1081     if t.type == SEGMENT_WORD then
1082       item = t.word
1083       word_count = word_count + 1
1084     elseif t.type == SEGMENT_SPACE then
1085       item = STR_SPACE
1086     elseif t.type == SEGMENT_MATH then
1087       item = STR_MATH
```

```

1088     end
1089
1090     if incr > 0 then
1091         result = result .. item
1092     else
1093         result = item .. result
1094     end
1095
1096     i = i + incr
1097 end
1098
1099 return result
1100
1101 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.

```

1102 local function make_flag_node(code,context)
1103     if context then
1104         code = context .. ' ' .. code
1105     end
1106
1107     local n = node.new('whatsit','pdf_literal')
1108     n.data = 'q ' .. code .. ' Q'
1109
1110     return n
1111
1112 end

```

(End of definition for make_flag_node.)

Transformation to shift flags into the margin.

```

1113 local FLAG_MARGIN_SHIFT = '1 0 0 1 4 -1 cm'

```

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

```

1114 local FLAG_WIDTH = tex.sp('6pt')

```

Code for flags.

```

1115 local FLAG_VALID_CODE
1116     = '0 0 6 10 re .1 .7 .1 rg f ' -- Green background
1117     .. '1 w 1 G ' -- Line width 1pt, colour white
1118     .. '1 5.5 m 2.33333 3 1 5 8 1 S' -- "Tick"
1119 local FLAG_INVALID_CODE
1120     = '0 0 6 10 re .9 0 0 rg f ' -- Red background
1121     .. '1 w 1 G ' -- Line width 1pt, colour white
1122     .. '1 3 m 5 7 1 1 7 m 5 3 1 S'
1123 local FLAG_AMBIGUOUS_CODE = (
1124     '0 0 6 10 re .1 .7 .7 rg f ' -- Cyan background
1125     .. '1 0 0 1 2.85 6.25 cm ' -- Shift
1126     .. '1 w 1 G ' -- Line width 1pt, colour white

```

```

1127 .. '-1.55885 .9 m -1 1.4 -.65 2 0 2 c 1.3 2 1.8 1.3 1.8 .5 c '
1128 .. '1.8 -.75 0 -1.25 0 -2.5 c 0 -2.75 1 S ' -- Question mark body
1129 .. '1.4 w 1 J ' -- Line width 1.4pt, line cap round
1130 .. '0 -4.25 m 0 -4.25 1 S' -- Question mark dot
1131 )
1132 local FLAG_UNKNOWN_CODE = (
1133 '0 0 6 10 re .9 .9 0 rg f ' -- Yellow background
1134 .. '1 0 0 1 3 5 cm ' -- Shift
1135 .. '.8 w 1 G ' -- Linec with .8pt, colour white
1136 .. '-1.41421 -1.41421 m 1.41421 1.41421 1 S ' -- Stroke of 0
1137 .. '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 '
1138 .. '1.2 -3 2 -1.2 2 0 c h S' -- Outside of 0
1139 )

```

Nodes made from the preceding code.

```

1140 local FLAG_VALID_NODE = make_flag_node(FLAG_VALID_CODE,FLAG_MARGIN_SHIFT)
1141 local FLAG_INVALID_NODE = make_flag_node(FLAG_INVALID_CODE,FLAG_MARGIN_SHIFT)
1142 local FLAG_AMBIGUOUS_NODE = make_flag_node(FLAG_AMBIGUOUS_CODE,FLAG_MARGIN_SHIFT)
1143 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.

```

1144 local function insert_flag(head,current,status)
1145
1146   if status == STATUS_VALID then
1147     node.insert_after(head,current,node.copy(FLAG_VALID_NODE))
1148   elseif status == STATUS_UNKNOWN then
1149     node.insert_after(head,current,node.copy(FLAG_UNKNOWN_NODE))
1150   elseif status == STATUS_INVALID then
1151     node.insert_after(head,current,node.copy(FLAG_INVALID_NODE))
1152   elseif status == STATUS_AMBIGUOUS then
1153     node.insert_after(head,current,node.copy(FLAG_AMBIGUOUS_NODE))
1154   end
1155
1156 end

```

(End of definition for insert_flag.)

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

```

1157 local function write_flag(code)
1158
1159   node.write(make_flag_node(code,'1 0 0 1 0 -1 cm'))
1160
1161   local kern_n = node.new('kern','userkern')
1162   kern_n.kern=FLAG_WIDTH
1163   node.write(kern_n)
1164
1165 end

```

(End of definition for write_flag.)

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

```

1166 local function write_flag_valid()
1167
1168   write_flag(FLAG_VALID_CODE)

```

```

1169
1170 end

```

(End of definition for write_flag_valid.)

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

```

1171 local function write_flag_invalid()
1172
1173   write_flag(FLAG_INVALID_CODE)
1174
1175 end

```

(End of definition for write_flag_invalid.)

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

```

1176 local function write_flag_ambiguous()
1177
1178   write_flag(FLAG_AMBIGUOUS_CODE)
1179
1180 end

```

(End of definition for write_flag_ambiguous.)

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

```

1181 local function write_flag_unknown()
1182
1183   write_flag(FLAG_UNKNOWN_CODE)
1184
1185 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.

```

1186 local hyphenation_list = {}
1187 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`.

```

1188 local function check_line_hyphenation(hlist)

```

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

```

1189   local last_disc = get_used_disc(hlist.head)
1190   if not last_disc then
1191     debug(' No disc node found at end of line')
1192     return
1193   end
1194   debug(' Disc node found at end of line')

```

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

```

1195 local hyphenation_index = node.has_attribute(last_disc,hyphen_attr)
1196 local t = hlist_segment_list[hyphenation_index]
1197 assert(t)
1198 assert(t.type == SEGMENT_WORD)
1199 local word = t.word
1200 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.

```

1201 if not unicode.utf8.match(word,'%a') then
1202     debug(' Divided "word" contains no letters')
1203     return
1204 end

```

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

```

1205 local next_line = get_next_hlist(hlist)
1206
1207 if not next_line then
1208     debug(' No following line found (which should not happen)')
1209     return
1210 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.

```

1211 local pre = get_nodelist_text(get_disc_word_start(hlist.head,last_disc))
1212 local pre_temp = trim_nonlettershyphens_start(pre)
1213 if pre_temp ~= '' then
1214     pre = pre_temp
1215 end

```

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

```

1216 local post = trim_nonlettershyphens_end(get_line_first_word(next_line.head))

```

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

```

1217 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.)

```

1218 word = trim_nonlettershyphens_both(word)
1219
1220 local divided = pre .. post
1221 debug(' Hyphenated word found: %s -> %s (%s|%s)',word,divided,pre,post)
1222 local status = validate_division(lang,word,divided)

```

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

```

1223 local flag_mode = get_lang_option(lang,'flag-mode')
1224 if flag_mode == 1 or (flag_mode == 2 and status ~= STATUS_VALID) then
1225     insert_flag(hlist.head,last_disc,status)
1226 end

```

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

```

1227     local context =
1228     get_context(
1229         hyphenation_index,-1,get_lang_option(lang,'context-before')
1230     )
1231     .. word_orig ..
1232     get_context(
1233         hyphenation_index,1,get_lang_option(lang,'context-after')
1234     )

```

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

```

1235     hyphenation_count = hyphenation_count + 1
1236     hyphenation_list[hyphenation_count] = {
1237         lang = lang,
1238         word = word,
1239         divided = divided,
1240         index = hyphenation_count,
1241         context = context,
1242         status = status,
1243     }

```

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.

```

1244     late_lua_n = node.new('whatsit','late_lua')
1245     late_lua_n.data =
1246         'lualisthyphen.set_hyphenation_page('
1247         .. hyphenation_count .. ',tex.count["c@page"])'
1248
1249     node.insert_after(hlist.head,last_disc,late_lua_n)
1250
1251 end

```

(End of definition for check_line_hyphenation.)

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

```

1252 local function set_hyphenation_page(index,page)
1253
1254     hyphenation_list[index].page = page
1255
1256 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.

```

1257 local function post_linebreak(vlist_head,groupcode)
1258
1259     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.

```

1260   for item in node.traverse(vlist_head) do
1261     if item.id == NODE_ID_HLIST then
1262       node.slide(item.head)
1263     end
1264   end

```

Now actually look for hyphenations.

```

1265   local line_no = 0
1266
1267   for item in node.traverse(vlist_head) do
1268
1269     if item.id == NODE_ID_HLIST then
1270       line_no = line_no + 1
1271       debug(' Line no. %d',line_no)
1272       check_line_hyphenation(item)
1273     end
1274
1275   end
1276
1277   hlist_segment_list = {}
1278
1279   debug('Post-linebreak processing end')
1280
1281   return true
1282
1283 end

```

(End of definition for `post_linebreak`.)

11.17 Add callbacks

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

```

1284 local LUA_LIST_HYPHEN_PRE_LINEBREAK = 'LUA_LIST_HYPHEN_PRE_LINEBREAK'
1285 luatexbase.add_to_callback(
1286   'pre_linebreak_filter',
1287   pre_linebreak,
1288   LUA_LIST_HYPHEN_PRE_LINEBREAK
1289 )
1290
1291 local LUA_LIST_HYPHEN_POST_LINEBREAK = 'LUA_LIST_HYPHEN_POST_LINEBREAK'
1292 luatexbase.add_to_callback(
1293   'post_linebreak_filter',
1294   post_linebreak,
1295   LUA_LIST_HYPHEN_POST_LINEBREAK
1296 )

```

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.

```
1297 local function equal_hyphenation_case_sensitive(a,b)
1298   return (
1299     a.word == b.word
1300     and
1301     a.divided == b.divided
1302   )
1303 end
```

(End of definition for equal_hyphenation_case_sensitive.)

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

```
1304 local function equal_hyphenation_case_insensitive(a,b)
1305   return (
1306     unicode.utf8.lower(a.word) == unicode.utf8.lower(b.word)
1307     and
1308     unicode.utf8.lower(a.divided) == unicode.utf8.lower(b.divided)
1309   )
1310 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.

```
1311 local function lessthan_hyphenation_case_sensitive(a,b)
1312   return (
1313     a.word < b.word
1314     or
1315     (
1316       a.word == b.word
1317       and
1318       a.divided < b.divided
1319     )
1320     or
1321     (
1322       a.word == b.word
1323       and
1324       a.divided == b.divided
1325       and
1326       a.index < b.index
1327     )
1328   )
1329 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.

```
1330 local function lessthan_hyphenation_case_insensitive(a,b)
1331   return (
1332     unicode.utf8.lower(a.word) < unicode.utf8.lower(b.word)
1333     or
1334     (
```

```

1335         unicode.utf8.lower(a.word) == unicode.utf8.lower(b.word)
1336         and
1337         unicode.utf8.lower(a.divided) < unicode.utf8.lower(b.divided)
1338     )
1339     or
1340     (
1341         unicode.utf8.lower(a.word) == unicode.utf8.lower(b.word)
1342         and
1343         unicode.utf8.lower(a.divided) < unicode.utf8.lower(b.divided)
1344         and
1345         a.index < b.index
1346     )
1347 )
1348 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.

```

1349 local function sort_hyphenation_list_none(hyphenation_list)
1350     table.sort(
1351         hyphenation_list,
1352         function(a,b)
1353             return a.index < b.index
1354         end
1355     )
1356 end

```

(End of definition for sort_hyphenation_list_none.)

`sort_hyphenation_list_case` Sort `hyphenation_list` case-sensitively.

```

1357 local function sort_hyphenation_list_case(hyphenation_list)
1358     table.sort(
1359         hyphenation_list,
1360         lessthan_hyphenation_case_sensitive
1361     )
1362 end

```

(End of definition for sort_hyphenation_list_case.)

`sort_hyphenation_list_nocase` Sort `hyphenation_list` case-insensitively.

```

1363 local function sort_hyphenation_list_nocase(hyphenation_list)
1364     table.sort(
1365         hyphenation_list,
1366         lessthan_hyphenation_case_insensitive
1367     )
1368 end

```

(End of definition for sort_hyphenation_list_nocase.)

11.18.3 Deduplication

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

```
1369 local function deduplicate_hyphenation_list_case(hyphenation_list)
1370   table.sort(
1371     hyphenation_list,
1372     lessthan_hyphenation_case_sensitive
1373   )
1374   list_uniq(
1375     hyphenation_list,
1376     equal_hyphenation_case_sensitive
1377   )
1378 end
```

(End of definition for deduplicate_hyphenation_list_case.)

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

```
1379 local function deduplicate_hyphenation_list_nocase(hyphenation_list)
1380   table.sort(
1381     hyphenation_list,
1382     lessthan_hyphenation_case_insensitive
1383   )
1384   list_uniq(
1385     hyphenation_list,
1386     equal_hyphenation_case_insensitive
1387   )
1388 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`.

```
1389 local function process_lang_hyphenation_list(lang_id,hyphenation_list)
1390
1391   local output_non_typeset = get_lang_option(lang_id,'output-non-typeset')
1392   local include_valid = (get_lang_option(lang_id,'output-mode') == 0)
1393
```

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

```
1394   list_filter(
1395     hyphenation_list,
1396     function (a)
1397       return (
1398         (a.status ~= STATUS_VALID or include_valid)
1399         and
1400         (a.page or output_non_typeset)
1401       )
1402     end
1403   )
```

Deduplicate in accordance with the specified options for this language.

```

1404 local unique = get_lang_option(lang_id,'unique')
1405 if unique == 1 then
1406     deduplicate_hyphenation_list_case(hyphenation_list)
1407 elseif unique == 2 then
1408     deduplicate_hyphenation_list_nocase(hyphenation_list)
1409 end

```

Sort in accordance with the specified options for this language.

```

1410 local sort_mode = get_lang_option(lang_id,'sort')
1411 if sort_mode == 1 then
1412     sort_hyphenation_list_case(hyphenation_list)
1413 elseif sort_mode == 2 then
1414     sort_hyphenation_list_nocase(hyphenation_list)
1415 else
1416     sort_hyphenation_list_none(hyphenation_list)
1417 end
1418
1419 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`.

```

1420 local function write_lang_hyphenation_list_standard(f,hyphenation_list,widths)
1421
1422     for i,v in ipairs(hyphenation_list) do
1423
1424         if v then
1425             f:write(v.divided .. '\n')
1426         end
1427
1428     end
1429
1430 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.

```

1431 local function write_lang_hyphenation_list_verbose(f,hyphenation_list,show_status)

```

Calculate the number of columns required for each output field.

```

1432 local cols_word = 0
1433 local cols_divided = 0
1434 local cols_page = 0
1435
1436 for _,h in pairs(hyphenation_list) do
1437
1438     cols_word = math.max(
1439         cols_word,
1440         unicode.utf8.len(h.word)

```

```

1441     )
1442     cols_divided = math.max(
1443         cols_divided,
1444         unicode.utf8.len(h.divided)
1445     )
1446     cols_page = math.max(
1447         cols_page,
1448         unicode.utf8.len(tostring(h.page))
1449     )
1450
1451 end

```

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

```

1452 cols_page = math.max(
1453     cols_page + unicode.utf8.len(STR_PAGE_PREFIX),
1454     unicode.utf8.len(STR_PAGE_NONE)
1455 )
1456
1457 local cols_status = math.max(
1458     unicode.utf8.len(STR_VALID),
1459     unicode.utf8.len(STR_INVALID),
1460     unicode.utf8.len(STR_AMBIGUOUS),
1461     unicode.utf8.len(STR_UNKNOWN)
1462 ) + 2 -- Add for parentheses
1463
1464 for i,v in ipairs(hyphenation_list) do
1465
1466     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.

```

1467         local page = v.page
1468         if page then
1469             page = STR_PAGE_PREFIX .. page
1470         else
1471             page = STR_PAGE_NONE
1472         end

```

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

```

1473         local status
1474         if show_status then
1475             status = v.status
1476             if status == STATUS_VALID then
1477                 status = STR_VALID
1478             elseif status == STATUS_AMBIGUOUS then
1479                 status = STR_AMBIGUOUS
1480             elseif status == STATUS_INVALID then
1481                 status = STR_INVALID
1482             else
1483                 status = STR_UNKNOWN
1484             end
1485             status = rpad(parenthesize(status), cols_status) .. STR_SPACE_TWO
1486         else
1487             status = ''

```

```
1488         end
```

Write everything out on a single line.

```
1489         f:write(
1490             rpad(v.word,cols_word)
1491             .. STR_ARROW
1492             .. rpad(v.divided,cols_divided)
1493             .. STR_SPACE_TWO
1494             .. status
1495             .. lpad(page,cols_page)
1496             .. STR_SPACE
1497             .. STR_QUOTE_OPEN
1498             .. v.context
1499             .. STR_QUOTE_CLOSE
1500             .. '\n'
1501         )
1502     end
1503
1504 end
1505
1506 end
```

(End of definition for write_lang_hyphenation_list_verbose.)

get_settings_desc Compute a settings description to insert into file headers.

```
1507 local function get_settings_desc(lang_id)
1508
1509     local settings_desc
1510
1511     if get_lang_option(lang_id,'output-verbose') then
1512         settings_desc = string.format(
1513             'verbose=true,context-before=%d,context-after=%s',
1514             get_lang_option(lang_id,'context-before'),
1515             get_lang_option(lang_id,'context-after')
1516         )
1517     else
1518         settings_desc = 'verbose=false'
1519     end
1520
1521     local ALL_NONVALID = {
1522         [0] = 'all',
1523         [1] = 'non-valid',
1524     }
1525     settings_desc = settings_desc .. string.format(
1526         ',output-mode=%s',
1527         ALL_NONVALID[get_lang_option(lang_id,'output-mode')]
1528     )
1529     settings_desc = settings_desc .. string.format(
1530         ',include-non-typeset=%s',
1531         get_lang_option(lang_id,'output-non-typeset')
1532     )
1533     local NONE_CASE_NOCASE = {
1534         [0] = 'none',
1535         [1] = 'case',
1536         [2] = 'nocase'
```

```

1537 }
1538 settings_desc = settings_desc .. string.format(
1539     ',sort=%s,unique=%s',
1540     NONE_CASE_NOCASE[get_lang_option(lang_id,'sort')],
1541     NONE_CASE_NOCASE[get_lang_option(lang_id,'unique')]
1542 )
1543
1544 return settings_desc
1545
1546 end

```

(End of definition for get_settings_desc.)

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

```

1547 local function prefix_output_directory(path)
1548
1549     local output_directory = status.output_directory
1550     if not output_directory then
1551         return path
1552     end
1553
1554     if string.sub(output_directory,-1,-1) == '/' then
1555         return output_directory .. path
1556     else
1557         return output_directory .. '/' .. path
1558     end
1559
1560 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.

```

1561 local function process_write_lang_hyphenation_list(
1562     lang_id,hyphenation_list
1563 )
1564     process_lang_hyphenation_list(lang_id,hyphenation_list)
1565
1566     local output_path
1567     local lang_desc
1568
1569     local cname = lang_id_cname_table[lang_id]
1570
1571     debug('Processing and writing hyphenation lists for "%s" (lang. ID %d)',cname,lang_id)
1572
1573     if cname then
1574         output_path = get_lang_option(lang_id,'output-path')
1575         if not output_path then
1576             output_path = get_lang_option(LANG_DEFAULT,'output-prefix')
1577             .. cname .. get_lang_option(LANG_DEFAULT,'output-extension')
1578         end
1579         lang_desc = string.format('language "%s" (ID %s)',cname,lang_id)
1580     else

```

```

1581     output_path = get_lang_option(LANG_DEFAULT,'output-prefix')
1582     .. lang_id .. get_lang_option(LANG_DEFAULT,'output-extension')
1583     lang_desc = 'language with ID ' .. lang_id
1584 end
1585
1586 output_path = prefix_output_directory(output_path)
1587 debug('Output path: "%s"',output_path)
1588
1589 local output_header = get_lang_option(lang_id,'output-header')
1590 local output_verbose = get_lang_option(lang_id,'output-verbose')
1591
1592 local division_dict_read = lang_id_division_dict_read_table[lang_id]
1593
1594 local f = io.open(output_path,'w')
1595
1596 if output_header then
1597     f:write('% This file was generated by lua-list-hyphen\n')
1598     f:write('% Hyphenations for ' .. lang_desc .. '\n')
1599     f:write('% General settings: ' .. get_settings_desc(lang_id) .. '\n')
1600     local division_dict_path = lang_id_division_dict_path_table[lang_id]
1601     if division_dict_path and division_dict_read then
1602         f:write(
1603             '% Division dictionary read from: "' .. division_dict_path .. '"\n'
1604         )
1605     end
1606 end
1607
1608 if output_verbose then
1609     write_lang_hyphenation_list_verbose(f,hyphenation_list,division_dict_read)
1610 else
1611     write_lang_hyphenation_list_standard(f,hyphenation_list,division_dict_read)
1612 end
1613
1614 f:close()
1615
1616 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.

```

1617 local function process_write_hyphenation_lists()
1618
1619     local lang_hyphenation_table = {}
1620     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).

```

1621     for _,h in pairs(hyphenation_list) do
1622
1623         local lang = h.lang
1624
1625         local t = lang_hyphenation_table[lang]
1626         if not t then
1627             lang_hyphenation_table[lang] = {}
1628             t = lang_hyphenation_table[lang]

```

```

1629     end
1630
1631     table.insert(t,h)
1632
1633 end

```

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

```

1634 for lang_id,hyphenations in pairs(lang_hyphenation_table) do
1635     process_write_lang_hyphenation_list(lang_id,hyphenations)
1636 end
1637
1638 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`.

```

1639 lualisthyphen = lualisthyphen or {}
1640 lualisthyphen.process_write_hyphenation_lists = process_write_hyphenation_lists
1641 lualisthyphen.set_hyphenation_page = set_hyphenation_page
1642 lualisthyphen.store_lang_id_name = store_lang_id_name
1643
1644 lualisthyphen.write_flag_valid = write_flag_valid
1645 lualisthyphen.write_flag_invalid = write_flag_invalid
1646 lualisthyphen.write_flag_ambiguous = write_flag_ambiguous
1647 lualisthyphen.write_flag_unknown = write_flag_unknown
1648
1649 lualisthyphen.set_debug = set_debug
1650 lualisthyphen.set_global_option = set_global_option
1651 lualisthyphen.set_lang_option = set_lang_option
1652
1653 lualisthyphen.preprocessors = preprocessors
1654  $\langle$ /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>	607
B	
<code>\begin</code>	592, 608
bool commands:	
<code>\bool_to_str:N</code>	149,
178, 184, 186, 188, 206, 212, 214, 216	
C	
check commands:	
<code>check_line_hyphenation</code>	<u>1188</u>
clist commands:	
<code>\clist_new:N</code>	138
<code>\clist_put_right:Nn</code>	142
<code>\clist_use:N</code>	242
context (option)	10
context-after (option)	10
context-before (option)	10
cs commands:	
<code>\cs_generate_variant:Nn</code>	
144, 161, 166, 226	
<code>\cs_if_exist:NTF</code>	252
<code>\cs_new:Npn</code>	140,
147, 151, 157, 162, 197, 246, 250, 262	
<code>\cs_set_eq:NN</code>	255
D	
debug	<u>295</u>
debug commands:	
<code>debug_dummy</code>	<u>295</u>
<code>debug_real</code>	<u>295</u>
deduplicate commands:	
<code>deduplicate_hyphenation_list_-</code>	
case	<u>1369</u>
<code>deduplicate_hyphenation_list_-</code>	
nocase	<u>1379</u>
delete commands:	
<code>delete_hyphens</code>	<u>423</u>
division-dict-paths (option)	8
E	
<code>\en</code>	590, 606
english commands:	
<code>english_strip_possessives</code>	<u>648</u>
equal commands:	
<code>equal_hyphenation_case_insensitive</code>	
.....	<u>1304</u>
<code>equal_hyphenation_case_sensitive</code>	
.....	<u>1297</u>
F	
flag-mode (option)	9
<code>\foreignlanguage</code>	11
G	
get commands:	
<code>get_context</code>	<u>1068</u>
<code>get_disc_lang</code>	<u>870</u>
<code>get_disc_word_start</code>	<u>1014</u>
<code>get_first_glyph_lang</code>	<u>857</u>
<code>get_lang_division_dict_validation_-</code>	
preprocessor	<u>652</u>
<code>get_lang_option</code>	<u>533</u>
<code>get_line_first_word</code>	<u>1044</u>
<code>get_next_hlist</code>	<u>1030</u>
<code>get_node_text</code>	<u>778</u>
<code>get_nodelist_text</code>	<u>819</u>
<code>get_settings_desc</code>	<u>1507</u>
<code>get_used_disc</code>	<u>994</u>
group commands:	
<code>\group_begin:</code>	230, 254
<code>\group_end:</code>	238, 259
H	
hook commands:	
<code>\hook_gput_code:nmn</code>	243, 268
<code>\hyphenation</code>	6
hyphenation commands:	
<code>hyphenation_matches_pattern</code>	<u>433</u>
I	
identity	<u>644</u>
info	<u>286</u>
insert commands:	
<code>insert_flag</code>	<u>1144</u>
int commands:	
<code>\int_new:N</code>	30, 36, 62, 68
<code>\int_set:Nn</code>	33, 39, 65, 71
<code>\int_use:N</code>	180, 182, 190, 192,
194, 196, 208, 210, 218, 220, 222, 224	
is commands:	
<code>is_possible_space_node</code>	<u>848</u>
<code>is_possible_word_node</code>	<u>835</u>
J	
<code>\jobname</code>	5, 8

K

keys commands:

\l_keys_choice_int 33, 39, 65, 71
 \keys_define:nn 12, 16,
 20, 23, 26, 31, 37, 42, 45, 49, 52, 63,
 69, 74, 77, 80, 86, 95, 108, 130, 133, 228
 \l_keys_key_str 89,
 98, 102, 111, 116, 121, 126, 135, 232
 \keys_set:nn 112, 117, 122, 127, 233, 241
 \keys_set_known:nn 18

L

lessthan commands:

lessthan_hyphenation_case_-
 insensitive 1330
 lessthan_hyphenation_case_-
 sensitive 1311

list commands:

list_filter 337
 list_uniq 354

load commands:

load_division_dict 585

lpad 413

lua commands:

\lua_load_module:n 145
 \lua_now:n . 59, 149, 159, 164, 264, 269

lualisthyphen internal commands:

__lualisthyphen_babel_store_-
 lang_id_names: 248, 250, 250
 __lualisthyphen_babel_store_-
 lang_id_names_elt:nnnn
 257, 262, 262
 __lualisthyphen_call_lua_-
 boolean:nN 147, 147, 167
 \l__lualisthyphen_context_after_-
 int 52, 192, 220
 \l__lualisthyphen_context_-
 before_int 52, 190, 218
 \l__lualisthyphen_debug_bool 130, 168
 \l__lualisthyphen_division_dict_-
 path_str 20, 170, 200
 \l__lualisthyphen_flag_mode_int
 30, 180, 208
 __lualisthyphen_initialize:
 244, 246, 246
 \l__lualisthyphen_lang_name_str
 21, 227, 232, 236
 \l__lualisthyphen_lang_options_-
 clist 18, 19, 138, 142, 242
 __lualisthyphen_lua_set_global_-
 option:nn 157, 157,
 161, 169, 171, 173, 175, 177, 179,
 181, 183, 185, 187, 189, 191, 193, 195

__lualisthyphen_lua_set_lang_-
 option:nnn 162,
 162, 166, 199, 201, 203, 205, 207,
 209, 211, 213, 215, 217, 219, 221, 223
 __lualisthyphen_lua_set_per_-
 lang_options:n . . 197, 197, 226, 235
 __lualisthyphen_luastr_or_nil:N
 151,
 151, 170, 172, 174, 176, 200, 202, 204
 \l__lualisthyphen_output_-
 extension_str 16, 174
 \l__lualisthyphen_output_header_-
 bool 45, 186, 214
 \l__lualisthyphen_output_mode_-
 int 36, 182, 210
 \l__lualisthyphen_output_non_-
 typeset_bool 42, 184, 212
 \l__lualisthyphen_output_path_-
 str 80, 202
 \l__lualisthyphen_output_prefix_-
 str 12, 172
 \l__lualisthyphen_output_-
 verbose_bool 49, 188, 216
 \l__lualisthyphen_sort_int
 68, 196, 224
 __lualisthyphen_store_lang_-
 options:nn 135, 138, 140, 144
 \l__lualisthyphen_unique_int
 62, 194, 222
 \l__lualisthyphen_validation_-
 case_sensitive_bool . 26, 178, 206
 \l__lualisthyphen_validation_-
 preprocessor_str 23, 176, 204

M

make commands:

make_flag_node 1102

msg commands:

\msg_critical:nn 10
 \msg_new:nnn 8
 \msg_new:nnnn 83, 92, 105
 \msg_warning:nnn 88, 97, 101
 \msg_warning:nnnn . 110, 115, 120, 125

N

\n 1425, 1500, 1597, 1598, 1599, 1603
 \NeedsTeXFormat 3

O

options:

context 10
 context-after 10
 context-before 10
 division-dict-paths 8

