

Babel

Version 3.15

2017/11/03

Original author

Johannes L. Braams

Current maintainer

Javier Bezos

The standard distribution of \LaTeX contains a number of document classes that are meant to be used, but also serve as examples for other users to create their own document classes. These document classes have become very popular among \LaTeX users. But it should be kept in mind that they were designed for American tastes and typography. At one time they even contained a number of hard-wired texts.

This manual describes babel, a package that makes use of the capabilities of \TeX version 3 and, to some extent, xetex and luatex, to provide an environment in which documents can be typeset in a language other than US English, or in more than one language or script.

Current development is focused on Unicode engines (Xe \TeX and Lua \TeX) and the so-called *complex scripts*. New features related to font selection, bidi writing and the like will be added incrementally.

Babel provides support (total or partial) for about 200 languages, either as a “classical” package option or as an ini file. Furthermore, new languages can be created from scratch easily.

Contents

I	User guide	4
1	The user interface	4
1.1	Monolingual documents	4
1.2	Multilingual documents	5
1.3	Modifiers	6
1.4	xelatex and lualatex	6
1.5	Troubleshooting	7
1.6	Plain	7
1.7	Basic language selectors	8
1.8	Auxiliary language selectors	8
1.9	More on selection	9
1.10	Shorthands	10
1.11	Package options	14
1.12	The base option	16
1.13	ini files	16
1.14	Selecting fonts	22
1.15	Modifying a language	24
1.16	Creating a language	24
1.17	Getting the current language name	26
1.18	Hooks	26
1.19	Hyphenation tools	28
1.20	Selecting scripts	29
1.21	Language attributes	30
1.22	Languages supported by babel	30
1.23	Tips, workarounds, known issues and notes	31
1.24	Current and future work	32
1.25	Tentative and experimental code	33
2	Loading languages with language.dat	34
2.1	Format	34
3	The interface between the core of babel and the language definition files	35
3.1	Basic macros	36
3.2	Skeleton	38
3.3	Support for active characters	38
3.4	Support for saving macro definitions	39
3.5	Support for extending macros	39
3.6	Macros common to a number of languages	39
3.7	Encoding-dependent strings	40
4	Changes	43
4.1	Changes in babel version 3.9	43
4.2	Changes in babel version 3.7	44
II	The code	44
5	Identification and loading of required files	45
6	Tools	45
6.1	Multiple languages	48

7	The Package File (\LaTeX, babel.sty)	49
7.1	base	50
7.2	key=value options and other general option	51
7.3	Conditional loading of shorthands	52
7.4	Language options	53
8	The kernel of Babel (babel.def, common)	56
8.1	Tools	56
8.2	Hooks	58
8.3	Setting up language files	60
8.4	Shorthands	62
8.5	Language attributes	71
8.6	Support for saving macro definitions	73
8.7	Short tags	74
8.8	Hyphens	74
8.9	Multiencoding strings	76
8.10	Macros common to a number of languages	82
8.11	Making glyphs available	82
8.11.1	Quotation marks	82
8.11.2	Letters	83
8.11.3	Shorthands for quotation marks	84
8.11.4	Umlauts and tremas	85
9	The kernel of Babel (babel.def, only \LaTeX)	87
9.1	The redefinition of the style commands	87
9.2	Creating languages	87
9.3	Cross referencing macros	93
9.4	Marks	96
9.5	Preventing clashes with other packages	97
9.5.1	ifthen	97
9.5.2	varioref	98
9.5.3	hhline	98
9.5.4	hyperref	98
9.5.5	fancyhdr	99
9.6	Encoding and fonts	99
9.7	Basic bidi support	101
9.8	Local Language Configuration	103
10	Multiple languages (switch.def)	104
10.1	Selecting the language	105
10.2	Errors	112
11	Loading hyphenation patterns	113
12	Font handling with fontspec	118
13	Hooks for XeTeX and LuaTeX	121
13.1	XeTeX	121
13.2	LuaTeX	122
14	Bidi support in luatex	127
15	The ‘nil’ language	131

16	Support for Plain T_EX (plain.def)	132
16.1	Not renaming hyphen.tex	132
16.2	Emulating some L ^A T _E X features	133
16.3	General tools	134
16.4	Encoding related macros	137
16.5	Babel options	140
17	Acknowledgements	141

Part I

User guide

This user guide focuses on \LaTeX . There are also some notes on its use with Plain \TeX . If you are interested in the \TeX multilingual support, please join the kadingira list on <http://tug.org/mailman/listinfo/kadingira>.

1 The user interface

1.1 Monolingual documents

In most cases, a single language is required, and then all you need in \LaTeX is to load the package using its standard mechanism for this purpose, namely, passing that language as an optional argument. In addition, you may want to set the font and input encodings.

EXAMPLE Here is a simple full example for “traditional” \TeX engines (see below for xetex and luatex). The packages fontenc and inputenc do not belong to babel, but they are included in the example because typically you will need them:

```
\documentclass{article}

\usepackage[T1]{fontenc}
\usepackage[utf8]{inputenc}

\usepackage[french]{babel}

\begin{document}

Plus ça change, plus c'est la même chose!

\end{document}
```

WARNING A common source of trouble is a wrong setting of the input encoding. Make sure you set the encoding actually used by your editor.

Another approach is making the language (french in the example) a global option in order to let other packages detect and use it:

```
\documentclass[french]{article}
\usepackage{babel}
\usepackage{varioref}
```

In this last example, the package `varioref` will also see the option and will be able to use it.

NOTE Because of the way babel has evolved, “language” can refer to (1) a set of hyphenation patterns as preloaded into the format, (2) a package option, (3) an `ldf` file, and (4) a name used in the document to select a language or dialect. So, a package option refers to a language in a generic way – sometimes it is the actual language name used to select it, sometimes it is a file name loading a language with a different name, sometimes it is a file name loading several languages. Please, read the documentation for specific languages for further info.

1.2 Multilingual documents

In multilingual documents, just use several options. The last one is considered the main language, activated by default. Sometimes, the main language changes the document layout (eg, spanish and french).

EXAMPLE In \LaTeX , the preamble of the document:

```
\documentclass{article}
\usepackage[dutch,english]{babel}
```

would tell \LaTeX that the document would be written in two languages, Dutch and English, and that English would be the first language in use, and the main one.

You can also set the main language explicitly:

```
\documentclass{article}
\usepackage[main=english,dutch]{babel}
```

WARNING Languages may be set as global and as package option at the same time, but in such a case you should set explicitly the main language with the package option main:

```
\documentclass[italian]{book}
\usepackage[ngerman,main=italian]{babel}
```

WARNING In the preamble the main language has *not* been selected, except hyphenation patterns and the name assigned to `\language` (in particular, shorthands, captions and date are not activated). If you need to define boxes and the like in the preamble, you might want to use some of the language selectors described below.

To switch the language there are two basic macros, described below in detail:

`\selectlanguage` is used for blocks of text, while `\foreignlanguage` is for chunks of text inside paragraphs.

EXAMPLE A full bilingual document follows. The main language is french, which is activated when the document begins.

```
\documentclass{article}

\usepackage[T1]{fontenc}
\usepackage[utf8]{inputenc}

\usepackage[english,french]{babel}

\begin{document}

Plus ça change, plus c'est la même chose!

\selectlanguage{english}

And an English paragraph, with a short text in
\foreignlanguage{french}{français}.

\end{document}
```

1.3 Modifiers

New 3.9c The basic behaviour of some languages can be modified when loading babel by means of *modifiers*. They are set after the language name, and are prefixed with a dot (only when the language is set as package option – neither global options nor the main key accept them). An example is (spaces are not significant and they can be added or removed):¹

```
\usepackage[latin.medieval, spanish.notilde.lcroman, danish]{babel}
```

Attributes (described below) are considered modifiers, ie, you can set an attribute by including it in the list of modifiers. However, modifiers is a more general mechanism.

1.4 xelatex and lualatex

Many languages are compatible with xetex and luatex. With them you can use babel to localize the documents.

The Latin script is covered by default in current L^AT_EX (provided the document encoding is UTF-8), because the font loader is preloaded and the font is switched to `lmroman`. Other scripts require loading `fontspec`.

EXAMPLE The following bilingual, single script document in UTF-8 encoding just prints a couple of ‘captions’ and `\today` in Danish and Vietnamese. No additional packages are required.

```
\documentclass{article}

\usepackage[vietnamese,danish]{babel}

\begin{document}

\prefacename{} -- \alsoname{} -- \today

\selectlanguage{vietnamese}

\prefacename{} -- \alsoname{} -- \today

\end{document}
```

EXAMPLE Here is a simple monolingual document in Russian (text from the Wikipedia). Note neither `fontenc` nor `inputenc` are necessary, but the document should be encoded in UTF-8 and a so-called Unicode font must be loaded (in this example `\babelfont` is used, described below).

```
\documentclass{article}

\usepackage[russian]{babel}

\babelfont{rm}{DejaVu Serif}

\begin{document}

Россия, находящаяся на пересечении множества культур, а также
```

¹No predefined “axis” for modifiers are provided because languages and their scripts have quite different needs.

с учётом многонационального характера её населения, — отличается высокой степенью этнокультурного многообразия и способностью к межкультурному диалогу.

```
\end{document}
```

1.5 Troubleshooting

- Loading directly sty files in L^AT_EX (ie, `\usepackage{⟨language⟩}`) is deprecated and you will get the error:²

```
! Package babel Error: You are loading directly a language style.
(babel)                This syntax is deprecated and you must use
(babel)                \usepackage[language]{babel}.
```

- Another typical error when using babel is the following:³

```
! Package babel Error: Unknown language `LANG'. Either you have misspelled
(babel)                its name, it has not been installed, or you requested
(babel)                it in a previous run. Fix its name, install it or just
(babel)                rerun the file, respectively
```

The most frequent reason is, by far, the latest (for example, you included spanish, but you realized this language is not used after all, and therefore you removed it from the option list). In most cases, the error vanishes when the document is typeset again, but in more severe ones you will need to remove the aux file.

- The following warning is about hyphenation patterns, which are not under the direct control of babel:

```
Package babel Warning: No hyphenation patterns were preloaded for
(babel)                the language `LANG' into the format.
(babel)                Please, configure your TeX system to add them and
(babel)                rebuild the format. Now I will use the patterns
(babel)                preloaded for \language=0 instead on input line 57.
```

The document will be typeset, but very likely the text will not be correctly hyphenated. Some languages may be raising this warning wrongly (because they are not hyphenated); it is a bug to be fixed – just ignore it. See the manual of your distribution (MacT_EX, MikT_EX, T_EXLive, etc.) for further info about how to configure it.

1.6 Plain

In Plain, load languages styles with `\input` and then use `\begindocument` (the latter is defined by babel):

```
\input estonian.sty
\begindocument
```

²In old versions the error read “You have used an old interface to call babel”, not very helpful.

³In old versions the error read “You haven’t loaded the language LANG yet”.

WARNING Not all languages provide a `sty` file and some of them are not compatible with Plain.⁴

1.7 Basic language selectors

This section describes the commands to be used in the document to switch the language in multilingual documents. In most cases, only the two basic macros `\selectlanguage` and `\foreignlanguage` are necessary. The environments `otherlanguage`, `otherlanguage*` and `hyphenrules` are auxiliary, and described in the next section.

The main language is selected automatically when the document environment begins.

`\selectlanguage` `{\language}`

When a user wants to switch from one language to another he can do so using the macro `\selectlanguage`. This macro takes the language, defined previously by a language definition file, as its argument. It calls several macros that should be defined in the language definition files to activate the special definitions for the language chosen:

```
\selectlanguage{german}
```

This command can be used as environment, too.

NOTE For “historical reasons”, a macro name is converted to a language name without the leading `\`; in other words, `\selectlanguage{\german}` is equivalent to `\selectlanguage{german}`. Using a macro instead of a “real” name is deprecated.

WARNING If used inside braces there might be some non-local changes, as this would be roughly equivalent to:

```
{\selectlanguage{<inner-language>} ...}\selectlanguage{<outer-language>}
```

If you want a change which is really local, you must enclose this code with an additional grouping level.

`\foreignlanguage` `{\language}{\text}`

The command `\foreignlanguage` takes two arguments; the second argument is a phrase to be typeset according to the rules of the language named in its first one. This command (1) only switches the extra definitions and the hyphenation rules for the language, *not* the names and dates, (2) does not send information about the language to auxiliary files (i.e., the surrounding language is still in force), and (3) it works even if the language has not been set as package option (but in such a case it only sets the hyphenation patterns and a warning is shown).

1.8 Auxiliary language selectors

`\begin{otherlanguage}` `{\language} ... \end{otherlanguage}`

The environment `otherlanguage` does basically the same as `\selectlanguage`, except the language change is (mostly) local to the environment.

Actually, there might be some non-local changes, as this environment is roughly equivalent to:

⁴Even in the babel kernel there were some macros not compatible with plain. Hopefully these issues will be fixed soon.

```

\begingroup
\selectlanguage{<inner-language>}
...
\endgroup
\selectlanguage{<outer-language>}

```

If you want a change which is really local, you must enclose this environment with an additional grouping, like braces {}.

Spaces after the environment are ignored.

`\begin{otherlanguage*}` {<language>} ... `\end{otherlanguage*}`

Same as `\foreignlanguage` but as environment. Spaces after the environment are *not* ignored.

This environment was originally intended for intermixing left-to-right typesetting with right-to-left typesetting in engines not supporting a change in the writing direction inside a line. However, by default it never complied with the documented behaviour and it is just a version as environment of `\foreignlanguage`.

`\begin{hyphenrules}` {<language>} ... `\end{hyphenrules}`

The environment `hyphenrules` can be used to select *only* the hyphenation rules to be used (it can be used as command, too). This can for instance be used to select ‘nohyphenation’, provided that in `language.dat` the ‘language’ `nohyphenation` is defined by loading `zerohyph.tex`. It deactivates language shorthands, too (but not user shorthands). Except for these simple uses, `hyphenrules` is discouraged and `otherlanguage*` (the starred version) is preferred, as the former does not take into account possible changes in encodings of characters like, say, ‘ done by some languages (eg, italian, french, ukraineb). To set hyphenation exceptions, use `\babelhyphenation` (see below).

1.9 More on selection

`\babeltags` {<tag1> = <language1>, <tag2> = <language2>, ...}

New 3.9i In multilingual documents with many language switches the commands above can be cumbersome. With this tool shorter names can be defined. It adds nothing really new – it is just syntactical sugar.

It defines `\text{<tag1>}{<text>}` to be `\foreignlanguage{<language1>}{<text>}`, and `\begin{<tag1>}` to be `\begin{otherlanguage*}{<language1>}`, and so on. Note `\{<tag1>` is also allowed, but remember to set it locally inside a group.

EXAMPLE With

```
\babeltags{de = german}
```

you can write

```
text \textde{German text} text
```

and

```

text
\begin{de}
  German text
\end{de}
text

```

NOTE Something like `\babeltags{finnish = finnish}` is legitimate – it defines `\textfinnish` and `\finnish` (and, of course, `\begin{finnish}`).

NOTE Actually, there may be another advantage in the ‘short’ syntax `\text{<tag>}`, namely, it is not affected by `\MakeUppercase` (while `\foreignlanguage` is).

\babelensure [`include=<commands>`], [`exclude=<commands>`], [`fontenc=<encoding>`]]{<language>}

New 3.9i Except in a few languages, like russian, captions and dates are just strings, and do not switch the language. That means you should set it explicitly if you want to use them, or hyphenation (and in some cases the text itself) will be wrong. For example:

```
\foreignlanguage{russian}{text \foreignlanguage{polish}{\seename} text}
```

Of course, \TeX can do it for you. To avoid switching the language all the while, `\babelensure` redefines the captions for a given language to wrap them with a selector:

```
\babelensure{polish}
```

By default only the basic captions and `\today` are redefined, but you can add further macros with the key `include` in the optional argument (without commas). Macros not to be modified are listed in `exclude`. You can also enforce a font encoding with `fontenc`.⁵ A couple of examples:

```
\babelensure[include=\Today]{spanish}
\babelensure[fontenc=T5]{vietnamese}
```

They are activated when the language is selected (at the `afterextras` event), and it makes some assumptions which could not be fulfilled in some languages. Note also you should include only macros defined by the language, not global macros (eg, `\TeX` of `\dag`). With `ini` files (see below), captions are ensured by default.

1.10 Shorthands

A *shorthand* is a sequence of one or two characters that expands to arbitrary \TeX code. Shorthands can be used for different kinds of things, as for example: (1) in some languages shorthands such as "a are defined to be able to hyphenate the word if the encoding is OT1; (2) in some languages shorthands such as ! are used to insert the right amount of white space; (3) several kinds of discretionary and breaks can be inserted easily with "-", "=", etc. The package `inputenc` as well as `xetex` and `luatex` have alleviated entering non-ASCII characters, but minority languages and some kinds of text can still require characters not directly available on the keyboards (and sometimes not even as separated or precomposed Unicode characters). As to the point 2, now `pdfTeX` provides `\knbccode`, and `luatex` can manipulate the glyph list. Tools for point 3 can be still very useful in general.

⁵With it encoded string may not work as expected.

There are three levels of shorthands: *user*, *language*, and *system* (by order of precedence). Version 3.9 introduces the *language user* level on top of the user level, as described below. In most cases, you will use only shorthands provided by languages.

NOTE Note the following:

1. Activated chars used for two-char shorthands cannot be followed by a closing brace } and the spaces following are gobbled. With one-char shorthands (eg, :), they are preserved.
2. If on a certain level (system, language, user) there is a one-char shorthand, two-char ones starting with that char and on the same level are ignored.
3. Since they are active, a shorthand cannot contain the same character in its definition (except if it is deactivated with, eg, string).

A typical error when using shorthands is the following:

```
! Argument of \language@active@arg" has an extra }.
```

It means there is a closing brace just after a shorthand, which is not allowed (eg, "}). Just add {} after (eg, "{}}).

`\shorthandon` `{\shorthands-list}`
`\shorthandoff` `*{\shorthands-list}`

It is sometimes necessary to switch a shorthand character off temporarily, because it must be used in an entirely different way. For this purpose, the user commands `\shorthandoff` and `\shorthandon` are provided. They each take a list of characters as their arguments. The command `\shorthandoff` sets the `\catcode` for each of the characters in its argument to other (12); the command `\shorthandon` sets the `\catcode` to active (13). Both commands only work on ‘known’ shorthand characters. If a character is not known to be a shorthand character its category code will be left unchanged.

New 3.9a However, `\shorthandoff` does not behave as you would expect with characters like ~ or ^, because they usually are not “other”. For them `\shorthandoff*` is provided, so that with

```
\shorthandoff*{~^}
```

~ is still active, very likely with the meaning of a non-breaking space, and ^ is the superscript character. The catcodes used are those when the shorthands are defined, usually when language files are loaded.

`\usesshorthands` `*{\char}`

The command `\usesshorthands` initiates the definition of user-defined shorthand sequences. It has one argument, the character that starts these personal shorthands.

New 3.9a User shorthands are not always alive, as they may be deactivated by languages (for example, if you use " for your user shorthands and switch from german to french, they stop working). Therefore, a starred version `\usesshorthands*{\char}` is provided, which makes sure shorthands are always activated.

Currently, if the package option `shorthands` is used, you must include any character to be activated with `\usesshorthands`. This restriction will be lifted in a future release.

\defineshorthand [*<language>* , ...] { *<shorthand>* } { *<code>* }

The command `\defineshorthand` takes two arguments: the first is a one- or two-character shorthand sequence, and the second is the code the shorthand should expand to.

New 3.9a An optional argument allows to (re)define language and system shorthands (some languages do not activate shorthands, so you may want to add `\languageshorthands{<lang>}` to the corresponding `\extras{<lang>}`, as explained below). By default, user shorthands are (re)defined.

User shorthands override language ones, which in turn override system shorthands. Language-dependent user shorthands (new in 3.9) take precedence over “normal” user shorthands.

EXAMPLE Let's assume you want a unified set of shorthand for discretionaries (languages do not define shorthands consistently, and “-”, “\”, “=” have different meanings). You could start with, say:

```
\usesshorthands*{}
\defineshorthand{"*"}{\babelhyphen{soft}}
\defineshorthand{"-"}{\babelhyphen{hard}}
```

However, behaviour of hyphens is language dependent. For example, in languages like Polish and Portuguese, a hard hyphen inside compound words are repeated at the beginning of the next line. You could then set:

```
\defineshorthand[*polish,*portugese]{"-"}{\babelhyphen{repeat}}
```

Here, options with `*` set a language-dependent user shorthand, which means the generic one above only applies for the rest of languages; without `*` they would (re)define the language shorthands instead, which are overridden by user ones.

Now, you have a single unified shorthand (“-”), with a content-based meaning (‘compound word hyphen’) whose visual behavior is that expected in each context.

\aliasshorthand { *<original>* } { *<alias>* }

The command `\aliasshorthand` can be used to let another character perform the same functions as the default shorthand character. If one prefers for example to use the character `/` over `"` in typing Polish texts, this can be achieved by entering `\aliasshorthand{""}{/}`.

NOTE The substitute character must *not* have been declared before as shorthand (in such a case, `\aliasshorthands` is ignored).

EXAMPLE The following example shows how to replace a shorthand by another

```
\aliasshorthand{~}{^}
\AtBeginDocument{\shorthandoff*{~}}
```

WARNING Shorthands remember somehow the original character, and the fallback value is that of the latter. So, in this example, if no shorthand is found, `^` expands to a non-breaking space, because this is the value of `~` (internally, `^` still calls `\active@char~` or `\normal@char~`). Furthermore, if you change the system value of `^` with `\defineshorthand` nothing happens.

`\languageshorthands` $\{\langle language \rangle\}$

The command `\languageshorthands` can be used to switch the shorthands on the language level. It takes one argument, the name of a language or none (the latter does what its name suggests).⁶ Note that for this to work the language should have been specified as an option when loading the babel package. For example, you can use in english the shorthands defined by `ngerman` with

```
\addto\extrasenglish{\languageshorthands{ngerman}}
```

(You may also need to activate them with, for example, `\usesshorthands`.)

Very often, this is a more convenient way to deactivate shorthands than `\shorthandoff`, as for example if you want to define a macro to easy typing phonetic characters with `tipa`:

```
\newcommand{\myipa}[1]{\{\languageshorthands{none}\tipaencoding#1}}
```

`\babelshorthand` $\{\langle shorthand \rangle\}$

With this command you can use a shorthand even if (1) not activated in shorthands (in this case only shorthands for the current language are taken into account, ie, not user shorthands), (2) turned off with `\shorthandoff` or (3) deactivated with the internal `\bbl@deactivate`; for example, `\babelshorthand{"u}` or `\babelshorthand{:}`. (You can conveniently define your own macros, or even you own user shorthands provided they do not overlap.)

For your records, here is a list of shorthands, but you must double check them, as they may change:⁷

Languages with no shorthands Croatian, English (any variety), Indonesian, Hebrew, Interlingua, Irish, Lower Sorbian, Malaysian, North Sami, Romanian, Scottish, Welsh
Languages with only " as defined shorthand character Albanian, Bulgarian, Danish, Dutch, Finnish, German (old and new orthography, also Austrian), Icelandic, Italian, Norwegian, Polish, Portuguese (also Brazilian), Russian, Serbian (with Latin script), Slovene, Swedish, Ukrainian, Upper Sorbian

Basque " ' ~
Breton : ; ? !
Catalan " ' ` ^
Czech " -
Esperanto ^
Estonian " ~
French (all varieties) : ; ? !
Galician " . ' ~ < >
Greek ~
Hungarian ` ^
Kurmanji ^
Latin " ^ =
Slovak " ^ ' -
Spanish " . < > ' ^
Turkish : ! =

In addition, the babel core declares ~ as a one-char shorthand which is let, like the standard ~, to a non breaking space.⁸

⁶Actually, any name not corresponding to a language group does the same as none. However, follow this convention because it might be enforced in future releases of babel to catch possible errors.

⁷Thanks to Enrico Gregorio

⁸This declaration serves to nothing, but it is preserved for backward compatibility.

1.11 Package options

New 3.9a These package options are processed before language options, so that they are taken into account irrespective of its order. The first three options have been available in previous versions.

KeepShorthandsActive Tells babel not to deactivate shorthands after loading a language file, so that they are also available in the preamble.

activeacute For some languages babel supports this options to set ' as a shorthand in case it is not done by default.

activegrave Same for `.

shorthands= $\langle char \rangle \langle char \rangle \dots$ | off
The only language shorthands activated are those given, like, eg:

```
\usepackage[esperanto,french,shorthands=:;!]{babel}
```

If ' is included, activeacute is set; if ` is included, activegrave is set. Active characters (like ~) should be preceded by \string (otherwise they will be expanded by \LaTeX before they are passed to the package and therefore they will not be recognized); however, t is provided for the common case of ~ (as well as c for not so common case of the comma). With shorthands=off no language shorthands are defined, As some languages use this mechanism for tools not available otherwise, a macro \babelshorthand is defined, which allows using them; see above.

safe= none | ref | bib
Some \LaTeX macros are redefined so that using shorthands is safe. With safe=bib only \nocite, \bibcite and \bibitem are redefined. With safe=ref only \newlabel, \ref and \pageref are redefined (as well as a few macros from varioref and ifthen). With safe=none no macro is redefined. This option is strongly recommended, because a good deal of incompatibilities and errors are related to these redefinitions – of course, in such a case you cannot use shorthands in these macros, but this is not a real problem (just use “allowed” characters).

math= active | normal
Shorthands are mainly intended for text, not for math. By setting this option with the value normal they are deactivated in math mode (default is active) and things like $\{a'\}$ (a closing brace after a shorthand) are not a source of trouble any more.

config= $\langle file \rangle$
Load $\langle file \rangle$.cfg instead of the default config file bblopts.cfg (the file is loaded even with noconfigs).

main= $\langle language \rangle$
Sets the main language, as explained above, ie, this language is always loaded last. If it is not given as package or global option, it is added to the list of requested languages.

- headfoot=** `<language>`
- By default, headlines and footlines are not touched (only marks), and if they contain language dependent macros (which is not usual) there may be unexpected results. With this option you may set the language in heads and foots.
- noconfigs** Global and language default config files are not loaded, so you can make sure your document is not spoilt by an unexpected `.cfg` file. However, if the key config is set, this file is loaded.
- showlanguages** Prints to the log the list of languages loaded when the format was created: number (remember dialects can share it), name, hyphenation file and exceptions file.
- nocase** New 3.9l Language settings for uppercase and lowercase mapping (as set by `\SetCase`) are ignored. Use only if there are incompatibilities with other packages.
- silent** New 3.9l No warnings and no *infos* are written to the log file.⁹
- strings=** `generic` | `unicode` | `encoded` | `<label>` | ``
- Selects the encoding of strings in languages supporting this feature. Predefined labels are `generic` (for traditional \TeX , LICR and ASCII strings), `unicode` (for engines like xetex and luatex) and `encoded` (for special cases requiring mixed encodings). Other allowed values are font encoding codes (T1, T2A, LGR, L7X...), but only in languages supporting them. Be aware with encoded captions are protected, but they work in `\MakeUppercase` and the like (this feature misuses some internal \LaTeX tools, so use it only as a last resort).
- hyphenmap=** `off` | `main` | `select` | `other` | `other*`
- New 3.9g Sets the behaviour of case mapping for hyphenation, provided the language defines it.¹⁰ It can take the following values:
- off** deactivates this feature and no case mapping is applied;
- first** sets it at the first switching commands in the current or parent scope (typically, when the aux file is first read and at `\begin{document}`}, but also the first `\selectlanguage` in the preamble), and it's the default if a single language option has been stated;¹¹
- select** sets it only at `\selectlanguage`;
- other** also sets it at `otherlanguage`;
- other*** also sets it at `otherlanguage*` as well as in heads and foots (if the option `headfoot` is used) and in auxiliary files (ie, at `\select@language`), and it's the default if several language options have been stated. The option `first` can be regarded as an optimized version of `other*` for monolingual documents.¹²
- bidi=** `default` | `basic-r`
- New 3.14 Selects the bidi algorithm to be used in luatex and xetex. With `default` the bidi mechanism is just activated (by default it is not), but every change must be marked up. In xetex this is the only option. In luatex, `basic-r`, provides a simple and fast method for R text, which handles numbers and unmarked L text within an R context.

⁹You can use alternatively the package `silence`.

¹⁰Turned off in plain.

¹¹Duplicated options count as several ones.

¹²Providing `foreign` is pointless, because the case mapping applied is that at the end of paragraph, but if either xetex or luatex change this behaviour it might be added. On the other hand, `other` is provided even if I [JBL] think it isn't really useful, but who knows.

EXAMPLE The following text comes from the Arabic Wikipedia (article about Arabia). Copy-pasting some text from the Wikipedia is a good way to test this feature, which will be improved in the future. Remember `basic-r` is available in `luatex` only.

```
\documentclass{article}

\usepackage[nil, bidi=basic-r]{babel}

\babelprovide[import=ar, main]{arabic}

\babelfont{rm}{FreeSerif}

\begin{document}

    وقد عرفت شبه جزيرة العرب طيلة العصر الهليني (الاريفي) بـ
    Arabia أو Aravia (بالاريفية Αραβία), استخدم الرومان ثلاث
    بادئات بـ "Arabia" على ثلاث مناطق من شبه الجزيرة العربية, إلا أنها
    حقيقةً كانت أكبر مما تعرف عليه اليوم.

\end{document}
```

1.12 The base option

With this package option `babel` just loads some basic macros (those in `switch.def`), defines `\AfterBabelLanguage` and exits. It also selects the hyphenations patterns for the last language passed as option (by its name in `language.dat`). There are two main uses: classes and packages, and as a last resort in case there are, for some reason, incompatible languages. It can be used if you just want to select the hyphenations patterns of a single language, too.

`\AfterBabelLanguage` $\langle\textit{option-name}\rangle\{\langle\textit{code}\rangle\}$

This command is currently the only provided by `base`. Executes $\langle\textit{code}\rangle$ when the file loaded by the corresponding package option is finished (at `\ldf@finish`). The setting is global. So

```
\AfterBabelLanguage{french}\{...\}
```

does ... at the end of `french.ldf`. It can be used in `ldf` files, too, but in such a case the code is executed only if $\langle\textit{option-name}\rangle$ is the same as `\CurrentOption` (which could not be the same as the option name as set in `\usepackage!`).

EXAMPLE Consider two languages `foo` and `bar` defining the same `\macro` with `\newcommand`. An error is raised if you attempt to load both. Here is a way to overcome this problem:

```
\usepackage[base]{babel}
\AfterBabelLanguage{foo}{%
    \let\macroFoo\macro
    \let\macro\relax}
\usepackage[foo,bar]{babel}
```

1.13 ini files

An alternative approach to define a language is by means of an `ini` file. Currently `babel` provides about 200 of these files containing the basic data required for a language.

Most of them set the date, and many also the captions (Unicode and LICR). They will be evolving with the time to add more features (something to keep in mind if backward compatibility is important). The following section shows how to make use of them currently (by means of `\babelprovide`), but a higher interface, based on package options, is under development.

EXAMPLE Although Georgian has its own `ldf` file, here is how to declare this language with an `ini` file in Unicode engines. The `nil` language is required, because currently `babel` raises an error if there is no language.

```
\documentclass{book}

\usepackage[nil]{babel}
\babelprovide[import=ka, main]{georgian}

\babelfont{rm}{DejaVu Sans}

\begin{document}

\tableofcontents

\chapter{სამზარეულო და სუფრის ტრადიციები}

ქართული ტრადიციული სამზარეულო ერთ-ერთი უმდიდრესია მთელ მსოფლიოში.

\end{document}
```

Here is the list (u means Unicode captions, and l means LICR captions):

af	Afrikaans ^u	ce	Chechen
agq	Aghem	cgg	Chiga
ak	Akan	chr	Cherokee
am	Amharic ^u	ckb	Central Kurdish
ar	Arabic ^u	cs	Czech ^u
as	Assamese	cy	Welsh ^u
asa	Asu	da	Danish ^u
ast	Asturian ^u	dav	Taita
az-Cyrl	Azerbaijani	de-AT	German ^u
az-Latn	Azerbaijani	de-CH	German ^u
az	Azerbaijani ^u	de	German ^u
bas	Basaa	dje	Zarma
be	Belarusian ^u	dsb	Lower Sorbian ^u
bem	Bemba	dua	Duala
bez	Bena	dyo	Jola-Fonyi
bg	Bulgarian ^u	dz	Dzongkha
bm	Bambara	ebu	Embu
bn	Bangla ^u	ee	Ewe
bo	Tibetan ^u	el	Greek ^u
brx	Bodo	en-AU	English ^u
bs-Cyrl	Bosnian	en-CA	English ^u
bs-Latn	Bosnian ^u	en-GB	English ^u
bs	Bosnian ^u	en-NZ	English ^u
ca	Catalan ^u	en-US	English ^u

en	English ^{ul}	kok	Konkani
eo	Esperanto ^{ul}	ks	Kashmiri
es-MX	Spanish ^{ul}	ksb	Shambala
es	Spanish ^{ul}	ksf	Bafia
et	Estonian ^{ul}	ksh	Colognian
eu	Basque ^{ul}	kw	Cornish
ewo	Ewondo	ky	Kyrgyz
fa	Persian ^{ul}	lag	Langi
ff	Fulah	lb	Luxembourgish
fi	Finnish ^{ul}	lg	Ganda
fil	Filipino	lkt	Lakota
fo	Faroese	ln	Lingala
fur	Friulian ^{ul}	lo	Lao ^{ul}
fy	Western Frisian	lrc	Northern Luri
ga	Irish ^{ul}	lt	Lithuanian ^{ul}
gd	Scottish Gaelic ^{ul}	lu	Luba-Katanga
gl	Galician ^{ul}	luo	Luo
gsw	Swiss German	luy	Luyia
gu	Gujarati	lv	Latvian ^{ul}
guz	Gusii	mas	Masai
gv	Manx	mer	Meru
ha-GH	Hausa	mfe	Morisyen
ha-NE	Hausa ¹	mg	Malagasy
ha	Hausa	mgf	Makhuwa-Meetto
haw	Hawaiian	mgo	Meta'
he	Hebrew ^{ul}	mk	Macedonian ^{ul}
hi	Hindi ^u	ml	Malayalam ^{ul}
hr	Croatian ^{ul}	mn	Mongolian
hsb	Upper Sorbian ^{ul}	mr	Marathi ^{ul}
hu	Hungarian ^{ul}	ms-BN	Malay ¹
hy	Armenian	ms-SG	Malay ¹
ia	Interlingua ^{ul}	ms	Malay ^{ul}
id	Indonesian ^{ul}	mt	Maltese
ig	Igbo	mua	Mundang
ii	Sichuan Yi	my	Burmese
is	Icelandic ^{ul}	mzn	Mazanderani
it	Italian ^{ul}	naq	Nama
ja	Japanese	nb	Norwegian Bokmål ^{ul}
jgo	Ngomba	nd	North Ndebele
jmc	Machame	ne	Nepali
ka	Georgian ^{ul}	nl	Dutch ^{ul}
kab	Kabyle	nmg	Kwasio
kam	Kamba	nn	Norwegian Nynorsk ^{ul}
kde	Makonde	nnh	Ngiemboon
kea	Kabuverdianu	nus	Nuer
khq	Koyra Chiini	nyn	Nyankole
ki	Kikuyu	om	Oromo
kk	Kazakh	or	Odia
kkj	Kako	os	Ossetic
kl	Kalaallisut	pa-Arab	Punjabi
klj	Kalenjin	pa-Guru	Punjabi
km	Khmer	pa	Punjabi
kn	Kannada ^{ul}	pl	Polish ^{ul}
ko	Korean	pms	Piedmontese ^{ul}

ps	Pashto	ta	Tamil ^u
pt-BR	Portuguese ^{ul}	te	Telugu ^{ul}
pt-PT	Portuguese ^{ul}	teo	Teso
pt	Portuguese ^{ul}	th	Thai ^{ul}
qu	Quechua	ti	Tigrinya
rm	Romansh ^{ul}	tk	Turkmen ^{ul}
rn	Rundi	to	Tongan
ro	Romanian ^{ul}	tr	Turkish ^{ul}
rof	Rombo	twq	Tasawaq
ru	Russian ^{ul}	tzm	Central Atlas Tamazight
rw	Kinyarwanda	ug	Uyghur
rwk	Rwa	uk	Ukrainian ^{ul}
sah	Sakha	ur	Urdu ^{ul}
saq	Samburu	uz-Arab	Uzbek
sbp	Sangu	uz-Cyrl	Uzbek
se	Northern Sami ^{ul}	uz-Latn	Uzbek
seh	Sena	uz	Uzbek
ses	Koyraboro Senni	vai-Latn	Vai
sg	Sango	vai-Vaii	Vai
shi-Latn	Tachelhit	vai	Vai
shi-Tfng	Tachelhit	vi	Vietnamese ^{ul}
shi	Tachelhit	vun	Vunjo
si	Sinhala	wae	Walser
sk	Slovak ^{ul}	xog	Soga
sl	Slovenian ^{ul}	yav	Yangben
smn	Inari Sami	yi	Yiddish
sn	Shona	yo	Yoruba
so	Somali	yue	Cantonese
sq	Albanian ^{ul}	zgh	Standard Moroccan Tamazight
sr-Cyrl-BA	Serbian ^{ul}	zh-Hans-HK	Chinese
sr-Cyrl-ME	Serbian ^{ul}	zh-Hans-MO	Chinese
sr-Cyrl-XK	Serbian ^{ul}	zh-Hans-SG	Chinese
sr-Cyrl	Serbian ^{ul}	zh-Hans	Chinese
sr-Latn-BA	Serbian ^{ul}	zh-Hant-HK	Chinese
sr-Latn-ME	Serbian ^{ul}	zh-Hant-MO	Chinese
sr-Latn-XK	Serbian ^{ul}	zh-Hant	Chinese
sr-Latn	Serbian ^{ul}	zh	Chinese
sr	Serbian ^{ul}	zu	Zulu
sv	Swedish ^{ul}		
sw	Swahili		

In some context (currently `\babel font`) an ini file may be loaded by its name. Here is the list of the names currently supported. With these languages, `\babel font` loads (if not done before) the language and script names (even if the language is defined as a package option with an ldf file).

aghem	armenian
akan	assamese
albanian	asturian
american	asu
amharic	australian
arabic	austrian

azerbaijani-cyrillic	dzongkha
azerbaijani-cyrl	embu
azerbaijani-latin	english-au
azerbaijani-latn	english-australia
azerbaijani	english-ca
bafia	english-canada
bambara	english-gb
basaa	english-newzealand
basque	english-nz
belarusian	english-unitedkingdom
bemba	english-unitedstates
benä	english-us
bengali	english
bodo	esperanto
bosnian-cyrillic	estonian
bosnian-cyrl	ewe
bosnian-latin	ewondo
bosnian-latn	faroesee
bosnian	filipino
brazilian	finnish
breton	french-be
british	french-belgium
bulgarian	french-ca
burmese	french-canada
canadian	french-ch
cantonese	french-lu
catalan	french-luxembourg
centralatlastamazight	french-switzerland
centralkurdish	french
chechen	friulian
cherokee	fulah
chiga	galician
chinese-hans-hk	ganda
chinese-hans-mo	georgian
chinese-hans-sg	german-at
chinese-hans	german-austria
chinese-hant-hk	german-ch
chinese-hant-mo	german-switzerland
chinese-hant	german
chinese-simplified-hongkongsarchina	greek
chinese-simplified-macausarchina	gujarati
chinese-simplified-singapore	gusii
chinese-simplified	hausa-gh
chinese-traditional-hongkongsarchina	hausa-ghana
chinese-traditional-macausarchina	hausa-ne
chinese-traditional	hausa-niger
chinese	hausa
colognian	hawaiian
cornish	hebrew
croatian	hindi
czech	hungarian
danish	icelandic
duala	igbo
dutch	inarisami

indonesian
interlingua
irish
italian
japanese
jolafonyi
kabuverdianu
kabyle
kako
kalaallisut
kalenjin
kamba
kannada
kashmiri
kazakh
khmer
kikuyu
kinyarwanda
konkani
korean
koyraborosenni
koyrachiini
kwasio
kyrgyz
lakota
langi
lao
latvian
lingala
lithuanian
lowersorbian
lsorbian
lubakatanga
luo
luxembourgish
luyia
macedonian
machame
makhuwameetto
makonde
malagasy
malay-bn
malay-brunei
malay-sg
malay-singapore
malay
malayalam
maltese
manx
marathi
masai
mazanderani
meru
meta

mexican
mongolian
morisyen
mundang
nama
nepali
newzealand
ngiemboon
ngomba
norsk
northernluri
northernsami
northndebele
norwegianbokmal
norwegiannynorsk
nswissgerman
nuer
nyankole
nynorsk
occitan
oriya
oromo
ossetic
pashto
persian
piedmontese
polish
portuguese-br
portuguese-brazil
portuguese-portugal
portuguese-pt
portuguese
punjabi-arab
punjabi-arabic
punjabi-gurmukhi
punjabi-guru
punjabi
quechua
romanian
romansh
rombo
rundi
russian
rwa
sakha
samburu
samin
sango
sangu
scottishgaelic
sena
serbian-cyrillic-bosniaherzegovina
serbian-cyrillic-kosovo
serbian-cyrillic-montenegro

serbian-cyrillic	telugu
serbian-cyrl-ba	teso
serbian-cyrl-me	thai
serbian-cyrl-xk	tibetan
serbian-cyrl	tigrinya
serbian-latin-bosniaherzegovina	tongan
serbian-latin-kosovo	turkish
serbian-latin-montenegro	turkmen
serbian-latin	ukenglish
serbian-latn-ba	ukrainian
serbian-latn-me	upporsorbian
serbian-latn-xk	urdu
serbian-latn	usenglish
serbian	usorbian
shambala	uyghur
shona	uzbek-arab
sichuanyi	uzbek-arabic
sinhala	uzbek-cyrillic
slovak	uzbek-cyrl
slovene	uzbek-latin
slovenian	uzbek-latn
soga	uzbek
somali	vai-latin
spanish-mexico	vai-latn
spanish-mx	vai-vai
spanish	vai-vaii
standardmoroccantamazight	vai
swahili	vietnam
swedish	vietnamese
swissgerman	vunjo
tachelhit-latin	walser
tachelhit-latn	welsh
tachelhit-tfng	westernfrisian
tachelhit-tifinagh	yangben
tachelhit	yiddish
taita	yoruba
tamil	zarma
tasawaq	zulu afrikaans

1.14 Selecting fonts

New 3.15 Babel provides a high level interface on top of fontspec to select fonts. There is no need to load fontspec explicitly – babel does it for you with the first `\babelfont`.

`\babelfont` [*<language-list>*] {*<font-family>*} [*<font-options>*] {*<font-name>*}

Here *font-family* is `rm`, `sf` or `tt` (or newly defined ones, as explained below), and *font-name* is the same as in fontspec and the like.

If no language is given, then it is considered the default font for the family, activated when a language is selected. On the other hand, if there is one or more languages in the optional argument, the font will be assigned to them, overriding the default. Alternatively, you may set a font for a script – just precede its name (lowercase) with a star (eg, `*devanagari`).

Babel takes care the font language and the font script when languages are selected (as well as the writing direction); see the recognized languages above. In most cases, you will not

need *font-options*, which is the same as in `fontspec`, but you may add further key/value pairs if necessary.

EXAMPLE Usage in most cases is very simple. Let us assume you are setting up a document in Swedish, with some words in Hebrew, with a font suited for both languages.

```
\documentclass{article}

\usepackage[swedish, bidi=default]{babel}

\babelprovide[import=he]{hebrew}

\babelfont{rm}{FreeSerif}

\begin{document}

Svenska \foreignlanguage{hebrew}{עברית} svenska.

\end{document}
```

If on the other hand you have to resort to different fonts, you could replace the red line above with, say:

```
\babelfont{rm}{Iwona}
\babelfont[hebrew]{rm}{FreeSerif}
```

`\babelfont` can be used to implicitly define a new font family. Just write its name instead of `rm`, `sf` or `tt`. This is the preferred way to select fonts in addition to the three basic ones.

EXAMPLE Here is how to do it:

```
\babelfont{kai}{FandolKai}
```

Now, `\kaifamily` and `\kaidefault` are at your disposal.

NOTE Directionality is a property affecting margins, indentation, column order, etc., not just text. Therefore, it is under the direct control of the language, which applies both the script and the direction to the text. As a consequence, there is no need to set `Script` when declaring a font (nor `Language`). In fact, it is even discouraged.

NOTE `\fontspec` is not touched at all, only the preset font families (`rm`, `sf`, `tt`, and the like). If a language is switched when an *ad hoc* font is active, or you select the font it with this command, neither the script nor the language are passed. You must add them by hand. This is by design, for several reasons (for example, each font has its own set of features and a generic setting for several of them could be problematic, and also a “lower level” font selection is useful).

NOTE The keys `Language` and `Script` just pass these values to the *font*, and do *not* set the script for the *language* (and therefore the writing direction). In other words, the `ini` file or `\babelprovide` provides default values for `\babelfont` if omitted, but the opposite is not true. See the note above for the reasons of this behaviour.

WARNING Do not use `\setxxxxfont` and `\babelfont` at the same time. `\babelfont` follows the standard \TeX conventions to set the basic families – define `\xxdefault`, and activate it with `\xxfamily`. On the other hand, `\setxxxxfont` in `fontspec` takes a different approach, because `\xxfamily` is redefined with the family name hardcoded (so that `\xxdefault` becomes no-op). Of course, both methods are incompatible, and if you use `\setxxxxfont`, font switching with `\babelfont` just does *not* work (nor the standard `\xxdefault`, for that matter).

1.15 Modifying a language

Modifying the behaviour of a language (say, the chapter “caption”), is sometimes necessary, but not always trivial.

- The old way, still valid for many languages, to redefine a caption is the following:

```
\addto\captionsenglish{%
  \renewcommand\contentsname{Foo}%
}
```

As of 3.15, there is no need to hide spaces with % (babel removes them), but it is advisable to do it.

- The new way, which is found in bulgarian, azerbaijani, spanish, french, turkish, icelandic, vietnamese and a few more, as well as in languages created with `\babelprovide` and its key import, is:

```
\renewcommand\spanishchaptername{Foo}
```

- Macros to be run when a language is selected can be add to `\extras<lang>`:

```
\addto\extrarussian{\mymacro}
```

There is a counterpart for code to be run when a language is unselected: `\noextras<lang>`.

NOTE These macros (`\captions<lang>`, `\extras<lang>`) may be redefined, but must not be used as such – they just pass information to babel, which executes them in the proper context.

1.16 Creating a language

New 3.10 And what if there is no style for your language or none fits your needs? You may then define quickly a language with the help of the following macro in the preamble.

`\babelprovide` [*<options>*] {*<language-name>*}

Defines the internal structure of the language with some defaults: the hyphen rules, if not available, are set to the current ones, left and right hyphen mins are set to 2 and 3, but captions and date are not defined. Conveniently, babel warns you about what to do. Very likely you will find alerts like that in the log file:

```
Package babel Warning: \mylangchaptername not set. Please, define
(babel)                it in the preamble with something like:
(babel)                \renewcommand\mylangchaptername{..}
(babel)                Reported on input line 18.
```

In most cases, you will only need to define a few macros.

EXAMPLE If you need a language named arhinish:

```
\usepackage[danish]{babel}
\babelprovide{arhinish}
\renewcommand\arhinishchaptername{Chapitula}
\renewcommand\arhinishrefname{Refirenke}
\renewcommand\arhinishhyphenmins{22}
```

The main language is not changed (danish in this example). So, you must add `\selectlanguage{arhinish}` or other selectors where necessary. If the language has been loaded as an argument in `\documentclass` or `\usepackage`, then `\babelprovide` redefines the requested data.

import= *<language-tag>*

New 3.13 Imports data from an ini file, including captions, date, and hyphenmins. For example:

```
\babelprovide[import=hu]{hungarian}
```

Unicode engines load the UTF-8 variants, while 8-bit engines load the LICR (ie, with macros like `\'` or `\ss`) ones.

There are about 200 ini files, with data taken from the ldf files and the CLDR provided by Unicode. Not all languages in the latter are complete, and therefore neither are the ini files. A few languages will show a warning about the current lack of suitability of the date format (hindi, french, breton, and occitan).

Besides `\today`, there is a `\<language>date` macro with three arguments: year, month and day numbers. In fact, `\today` calls `\<language>today` which in turn calls `\<language>date{\year}{\month}{\day}`.

captions= *<language-tag>*

Loads only the strings. For example:

```
\babelprovide[captions=hu]{hungarian}
```

hyphenrules= *<language-list>*

With this option, with a space-separated list of hyphenation rules, babel assigns to the language the first valid hyphenation rules in the list. For example:

```
\babelprovide[hyphenrules=chavacano spanish italian]{chavacano}
```

If none of the listed hyphenrules exist, the default behaviour applies. Note in this example we set chavacano as first option – without it, it would select spanish even if chavacano exists.

A special value is +, which allocates a new language (in the T_EX sense). It only makes sense as the last value (or the only one; the subsequent ones are silently ignored). It is mostly useful with luatex, because you can add some patterns with \babelpatterns, as for example:

```
\babelprovide[hyphenrules=+]{neo}
\babelpatterns[neo]{a1 e1 i1 o1 u1}
```

In other engines it just suppresses hyphenation (because the pattern list is empty).

main This valueless option makes the language the main one. Only in newly defined languages.

script= $\langle\textit{script-name}\rangle$

New 3.15 Sets the script name to be used by fontspec (eg, Devanagari). Overrides the value in the ini file. This value is particularly important because it sets the writing direction.

language= $\langle\textit{language-name}\rangle$

New 3.15 Sets the language name to be used by fontspec (eg, Hindi). Overrides the value in the ini file. Not so important, but sometimes still relevant.

NOTE (1) If you need shorthands, you can use \usesshorthands and \defineshorthand as described above. (2) Captions and \today are “ensured” with \babelensure (this is the default in ini-based languages).

1.17 Getting the current language name

\language The control sequence \language contains the name of the current language.

WARNING Due to some internal inconsistencies in catcodes, it should *not* be used to test its value. Use iflang, by Heiko Oberdiek.

\iflanguage $\{\langle\textit{language}\rangle\}\{\langle\textit{true}\rangle\}\{\langle\textit{false}\rangle\}$

If more than one language is used, it might be necessary to know which language is active at a specific time. This can be checked by a call to \iflanguage, but note here “language” is used in the T_EX sense, as a set of hyphenation patterns, and *not* as its babel name. This macro takes three arguments. The first argument is the name of a language; the second and third arguments are the actions to take if the result of the test is true or false respectively.

WARNING The advice about \language also applies here – use iflang instead of \iflanguage if possible.

1.18 Hooks

New 3.9a A hook is a piece of code to be executed at certain events. Some hooks are predefined when luatex and xetex are used.

\AddBabelHook {<name>}{<event>}{<code>}

The same name can be applied to several events. Hooks may be enabled and disabled for all defined events with `\EnableBabelHook{<name>}`, `\DisableBabelHook{<name>}`.

Names containing the string `babel` are reserved (they are used, for example, by `\useshortands*` to add a hook for the event `afterextras`).

Current events are the following; in some of them you can use one to three \TeX parameters (`#1`, `#2`, `#3`), with the meaning given:

adddialect (language name, dialect name) Used by `luababel.def` to load the patterns if not preloaded.

patterns (language name, language with encoding) Executed just after the `\language` has been set. The second argument has the patterns name actually selected (in the form of either `lang:ENC` or `lang`).

hyphenation (language name, language with encoding) Executed locally just before exceptions given in `\babelhyphenation` are actually set.

defaultcommands Used (locally) in `\StartBabelCommands`.

encodedcommands (input, font encodings) Used (locally) in `\StartBabelCommands`. Both `xetex` and `luatex` make sure the encoded text is read correctly.

stopcommands Used to reset the the above, if necessary.

write This event comes just after the switching commands are written to the aux file.

beforeextras Just before executing `\extras<language>`. This event and the next one should not contain language-dependent code (for that, add it to `\extras<language>`).

afterextras Just after executing `\extras<language>`. For example, the following deactivates shorthands in all languages:

```
\AddBabelHook{noshort}{afterextras}{\languageshorthands{none}}
```

stringprocess Instead of a parameter, you can manipulate the macro `\BabelString` containing the string to be defined with `\SetString`. For example, to use an expanded version of the string in the definition, write:

```
\AddBabelHook{myhook}{stringprocess}{%  
  \protected@edef\BabelString{\BabelString}}
```

initiateactive (char as active, char as other, original char) **New 3.9i** Executed just after a shorthand has been ‘initiated’. The three parameters are the same character with different catcodes: active, other (`\string’ed`) and the original one.

afterreset **New 3.9i** Executed when selecting a language just after `\originalTeX` is run and reset to its base value, before executing `\captions<language>` and `\date<language>`.

Four events are used in `hyphen.cfg`, which are handled in a quite different way for efficiency reasons – unlike the precedent ones, they only have a single hook and replace a default definition.

everylanguage (language) Executed before every language patterns are loaded.

loadkernel (file) By default loads `switch.def`. It can be used to load a different version of this files or to load nothing.

loadpatterns (patterns file) Loads the patterns file. Used by `luababel.def`.

loadexceptions (exceptions file) Loads the exceptions file. Used by `luababel.def`.

\BabelContentsFiles **New 3.9a** This macro contains a list of “toc” types which require a command to switch the language. Its default value is `toc,lof,lot`, but you may redefine it with `\renewcommand` (it’s up to you to make sure no toc type is duplicated).

1.19 Hyphenation tools

`\babelhyphen` `*{\type}`
`\babelhyphen` `*{\text}`

New 3.9a It is customary to classify hyphens in two types: (1) *explicit* or *hard hyphens*, which in \TeX are entered as `-`, and (2) *optional* or *soft hyphens*, which are entered as `\-`. Strictly, a *soft hyphen* is not a hyphen, but just a breaking opportunity or, in \TeX terms, a “discretionary”; a *hard hyphen* is a hyphen with a breaking opportunity after it. A further type is a *non-breaking hyphen*, a hyphen without a breaking opportunity.

In \TeX , `-` and `\-` forbid further breaking opportunities in the word. This is the desired behaviour very often, but not always, and therefore many languages provide shorthands for these cases. Unfortunately, this has not been done consistently: for example, in Dutch, Portugese, Catalan or Danish, `-` is a hard hyphen, while in German, Spanish, Norwegian, Slovak or Russian, it is a soft hyphen. Furthermore, some of them even redefine `\-`, so that you cannot insert a soft hyphen without breaking opportunities in the rest of the word. Therefore, some macros are provided with a set of basic “hyphens” which can be used by themselves, to define a user shorthand, or even in language files.

- `\babelhyphen{soft}` and `\babelhyphen{hard}` are self explanatory.
- `\babelhyphen{repeat}` inserts a hard hyphen which is repeated at the beginning of the next line, as done in languages like Polish, Portugese and Spanish.
- `\babelhyphen{nobreak}` inserts a hard hyphen without a break after it (even if a space follows).
- `\babelhyphen{empty}` inserts a break opportunity without a hyphen at all.
- `\babelhyphen{\text}` is a hard “hyphen” using `\text` instead. A typical case is `\babelhyphen{/}`.

With all of them hyphenation in the rest of the word is enabled. If you don’t want enabling it, there is a starred counterpart: `\babelhyphen*{soft}` (which in most cases is equivalent to the original `\-`), `\babelhyphen*{hard}`, etc.

Note `hard` is also good for isolated prefixes (eg, *anti-*) and `nobreak` for isolated suffixes (eg, *-ism*), but in both cases `\babelhyphen*{nobreak}` is usually better.

There are also some differences with \LaTeX : (1) the character used is that set for the current font, while in \LaTeX it is hardwired to `-` (a typical value); (2) the hyphen to be used in fonts with a negative `\hyphenchar` is `-`, like in \LaTeX , but it can be changed to another value by redefining `\babelexhyphen`; (3) a break after the hyphen is forbidden if preceded by a glue >0 pt (at the beginning of a word, provided it is not immediately preceded by, say, a parenthesis).

`\babelhyphenation` `[\langle language \rangle, \langle language \rangle, \dots]{\langle exceptions \rangle}`

New 3.9a Sets hyphenation exceptions for the languages given or, without the optional argument, for *all* languages (eg, proper nouns or common loan words, and of course monolingual documents). Language exceptions take precedence over global ones.

It can be used only in the preamble, and exceptions are set when the language is first selected, thus taking into account changes of `\lccodes`’s done in `\extras{lang}` as well as the language specific encoding (not set in the preamble by default). Multiple `\babelhyphenation`’s are allowed. For example:

```
\babelhyphenation{Wal-hal-la Dar-bhan-ga}
```

Listed words are saved expanded and therefore it relies on the LICR. Of course, it also works without the LICR if the input and the font encodings are the same, like in Unicode based engines.

\babelpatterns [*<language>* , *<language>* , ...] { *<patterns>* }

New 3.9m *In luatex only*,¹³ adds or replaces patterns for the languages given or, without the optional argument, for *all* languages. If a pattern for a certain combination already exists, it gets replaced by the new one.

It can be used only in the preamble, and patterns are added when the language is first selected, thus taking into account changes of \lccodes's done in \extras<lang> as well as the language specific encoding (not set in the preamble by default). Multiple \babelpatterns's are allowed.

Listed patterns are saved expanded and therefore it relies on the LICR. Of course, it also works without the LICR if the input and the font encodings are the same, like in Unicode based engines.

1.20 Selecting scripts

Currently babel provides no standard interface to select scripts, because they are best selected with either \fontencoding (low level) or a language name (high level). Even the Latin script may require different encodings (ie, sets of glyphs) depending on the language, and therefore such a switch would be in a sense incomplete.¹⁴

Some languages sharing the same script define macros to switch it (eg, \textcyrillic), but be aware they may also set the language to a certain default. Even the babel core defined \textlatin, but it was somewhat buggy because in some cases it messed up encodings and fonts (for example, if the main latin encoding was LY1), and therefore it has been deprecated.¹⁵

No macros to select the writing direction are provided, either – writing direction is intrinsic to each script and therefore it is best set by the language (which could be a dummy one). Furthermore, there are in fact two right-to-left modes, depending on the language, which differ in the way ‘weak’ numeric characters are ordered (eg, Arabic %123 vs Hebrew 123%).

\ensureascii { *<text>* }

New 3.9i This macro makes sure *<text>* is typeset with a LICR-savvy encoding in the ASCII range. It is used to redefine \TeX and \LaTeX so that they are correctly typeset even with LGR or X2 (the complete list is stored in \BabelNonASCII, which by default is LGR, X2, OT2, OT3, OT6, LHE, LWN, LMA, LMC, LMS, LMU, but you can modify it). So, in some sense it fixes the bug described in the previous paragraph.

If non-ASCII encodings are not loaded (or no encoding at all), it is no-op (also \TeX and \LaTeX are not redefined); otherwise, \ensureascii switches to the encoding at the beginning of the document if ASCII-savvy, or else the last ASCII-savvy encoding loaded. For example, if you load LY1, LGR, then it is set to LY1, but if you load LY1, T2A it is set to T2A. The symbol encodings TS1, T3, and TS3 are not taken into account, since they are not used for “ordinary” text.

¹³With luatex exceptions and patterns can be modified almost freely. However, this is very likely a task for a separate package and babel only provides the most basic tools.

¹⁴The so-called Unicode fonts do not improve the situation either. So, a font suited for Vietnamese is not necessarily suited for, say, the romanization of Indic languages, and the fact it contains glyphs for Modern Greek does not mean it includes them for Classic Greek. As to directionality, it poses special challenges because it also affects individual characters and layout elements.

¹⁵But still defined for backwards compatibility.

The foregoing rules (which are applied “at begin document”) cover most of cases. No assumption is made on characters above 127, which may not follow the LICR conventions – the goal is just to ensure most of the ASCII letters and symbols are the right ones.

1.21 Language attributes

\languageattribute This is a user-level command, to be used in the preamble of a document (after `\usepackage[...]{babel}`), that declares which attributes are to be used for a given language. It takes two arguments: the first is the name of the language; the second, a (list of) attribute(s) to be used. Attributes must be set in the preamble and only once – they cannot be turned on and off. The command checks whether the language is known in this document and whether the attribute(s) are known for this language. Very often, using a *modifier* in a package option is better. Several language definition files use their own methods to set options. For example, french uses `\frenchsetup`, magyar (1.5) uses `\magyarOptions`; modifiers provided by spanish have no attribute counterparts. Macros setting options are also used (eg, `\ProsodicMarksOn` in latin).

1.22 Languages supported by babel

In the following table most of the languages supported by babel are listed, together with the names of the option which you can load babel with for each language. Note this list is open and the current options may be different. It does not include ini files.

Afrikaans	afrikaans
Azerbaijani	azerbaijani
Basque	basque
Breton	breton
Bulgarian	bulgarian
Catalan	catalan
Croatian	croatian
Czech	czech
Danish	danish
Dutch	dutch
English	english, USenglish, american, UKenglish, british, canadian, australian, newzealand
Esperanto	esperanto
Estonian	estonian
Finnish	finnish
French	french, francais, canadien, acadian
Galician	galician
German	austrian, german, germanb, ngerman, naustrian
Greek	greek, polutonikogreek
Hebrew	hebrew
Icelandic	icelandic
Indonesian	bahasa, indonesian, indon, bahasai
Interlingua	interlingua
Irish Gaelic	irish
Italian	italian
Latin	latin
Lower Sorbian	lowersorbian
Malay	bahasam, malay, melayu
North Sami	samin
Norwegian	norsk, nynorsk
Polish	polish

Portuguese portugese, portuguese, brazilian, brazil
Romanian romanian
Russian russian
Scottish Gaelic scottish
Spanish spanish
Slovakian slovak
Slovenian slovene
Swedish swedish
Serbian serbian
Turkish turkish
Ukrainian ukrainian
Upper Sorbian uppersorbian
Welsh welsh

There are more languages not listed above, including hindi, thai, thaicjk, latvian, turkmen, magyar, mongolian, romansh, lithuanian, spanglish, vietnamese, japanese, pinyin, arabic, farsi, ibygreek, bgreek, serbianc, frenchle, ethiop and friulan.

Most of them work out of the box, but some may require extra fonts, encoding files, a preprocessor or even a complete framework (like CJK). For example, if you have got the `velthuis/devnag` package, you can create a file with extension `.dn`:

```

\documentclass{article}
\usepackage[hindi]{babel}
\begin{document}
{\dn devaanaa.m priya.h}
\end{document}

```

Then you preprocess it with `devnag <file>`, which creates `<file>.tex`; you can then typeset the latter with \LaTeX .

1.23 Tips, workarounds, know issues and notes

- If you use the document class *book* *and* you use `\ref` inside the argument of `\chapter` (or just use `\ref` inside `\MakeUppercase`), \LaTeX will keep complaining about an undefined label. To prevent such problems, you could revert to using uppercase labels, you can use `\lowercase{\ref{foo}}` inside the argument of `\chapter`, or, if you will not use shorthands in labels, set the `safe` option to `none` or `bib`.
- Both `ltxdoc` and `babel` use `\AtBeginDocument` to change some catcodes, and `babel` reloads `hline` to make sure `:` has the right one, so if you want to change the catcode of `|` it has to be done using the same method at the proper place, with

```
\AtBeginDocument{\DeleteShortVerb{\|}}
```

before loading `babel`. This way, when the document begins the sequence is (1) make `|` active (`ltxdoc`); (2) make it unactive (your settings); (3) make `babel` shorthands active (`babel`); (4) reload `hline` (`babel`, now with the correct catcodes for `|` and `:`).

- Documents with several input encodings are not frequent, but sometimes are useful. You can set different encodings for different languages as the following example shows:

```

\addto\extrasfrench{\inputencoding{latin1}}
\addto\extrasrussian{\inputencoding{koi8-r}}

```


(A recent version of inputenc is required.)

- For the hyphenation to work correctly, lccodes cannot change, because T_EX only takes into account the values when the paragraph is hyphenated, i.e., when it has been finished.¹⁶ So, if you write a chunk of French text with `\foreignlanguage`, the apostrophes might not be taken into account. This is a limitation of T_EX, not of babel. Alternatively, you may use `\usesshorthands` to activate ' and `\definesshorthand`, or redefine `\textquoteright` (the latter is called by the non-ASCII right quote).
- `\bibitem` is out of sync with `\selectlanguage` in the .aux file. The reason is `\bibitem` uses `\immediate` (and others, in fact), while `\selectlanguage` doesn't. There is no known workaround.
- Babel does not take into account `\normalsfcodes` and (non-)French spacing is not always properly (un)set by languages. However, problems are unlikely to happen and therefore this part remains untouched in version 3.9 (but it is in the 'to do' list).
- Using a character mathematically active (ie, with math code "8000) as a shorthand can make T_EX enter in an infinite loop in some rare cases. (Another issue in the 'to do' list, although there is a partial solution.)

The following packages can be useful, too (the list is still far from complete):

csquotes Logical markup for quotes.

iflang Tests correctly the current language.

hyphsubst Selects a different set of patterns for a language.

translator An open platform for packages that need to be localized.

siunitx Typesetting of numbers and physical quantities.

biblatex Programmable bibliographies and citations.

bicaption Bilingual captions.

babelbib Multilingual bibliographies.

microtype Adjusts the typesetting according to some languages (kerning and spacing).
Ligatures can be disabled.

substitutefont Combines fonts in several encodings.

mkpattern Generates hyphenation patterns.

tracklang Tracks which languages have been requested.

1.24 Current and future work

Current work is focused on the so-called complex scripts in luatex. In 8-bit engines, babel provided a basic support for bidi text as part of the style for Hebrew, but it is somewhat unsatisfactory and internally replaces some hardwired commands by other hardwired commands (generic changes would be much better).

It is possible now to typeset Arabic or Hebrew with numbers and L text. Next on the roadmap are line breaking in Thai and the like, as well as "non-European" digits. Also on the roadmap are R layouts (lists, footnotes, tables, column order), page and section numbering, and maybe kashida justification.

As to Thai line breaking, here is the basic idea of what luatex can do for us, with the Thai patterns and a little script (the final version will not be so little, of course). It replaces each discretionary by the equivalent to ZWJ.

¹⁶This explains why L^AT_EX assumes the lowercase mapping of T1 and does not provide a tool for multiple mappings. Unfortunately, `\savingshyphcodes` is not a solution either, because lccodes for hyphenation are frozen in the format and cannot be changed.

```

\documentclass{article}

\usepackage[nil]{babel}

\babelprovide[import=th, main]{thai}

\babelfont{rm}{FreeSerif}

\directlua{
local GLYPH = node.id'glyph'
function insertsp (head)
  local size = 0
  for item in node.traverse(head) do
    local i = item.id
    if i == GLYPH then
      f = font.getfont(item.font)
      size = f.size
    elseif i == 7 then
      local n = node.new(12, 0)
      node.setglue(n, 0, size * 1) % 1 is a factor
      node.insert_before(head, item, n)
      node.remove(head, item)
    end
  end
end
end

luatexbase.add_to_callback('hyphenate',
  function (head, tail)
    lang.hyphenate(head)
    insertsp(head)
  end, 'insertsp')
}

\begin{document}

(Thai text.)

\end{document}

```

Useful additions would be, for example, time, currency, addresses and personal names.¹⁷ But that is the easy part, because they don't require modifying the \LaTeX internals. Also interesting are differences in the sentence structure or related to it. For example, in Basque the number precedes the name (including chapters), in Hungarian “from (1)” is “(1)-ből”, but “from (3)” is “(3)-ból”, in Spanish an item labelled “3.^o” may be referred to as either “ítem 3.^o” or “3.^{er} ítem”, and so on.

1.25 Tentative and experimental code

Handling of “**Unicode**” fonts is problematic. There is fontspec, but special macros are required (not only the NFSS ones) and it doesn't provide “orthogonal axis” for features, including those related to the language (mainly language and script). A couple of tentative macros, were provided by babel (≥ 3.9 g) with a partial solution. These macros are now deprecated – use `\babelfont`.

¹⁷See for example POSIX, ISO 14652 and the Unicode Common Locale Data Repository (CLDR). Those system, however, have limited application to \TeX because their aim is just to display information and not fine typesetting.

- `\babelFSstore{\babel-language}` sets the current three basic families (rm, sf, tt) as the default for the language given.
- `\babelFSdefault{\babel-language}{\fontspec-features}` patches `\fontspec` so that the given features are always passed as the optional argument or added to it (not an ideal solution).

So, for example:

```
\setmainfont[Language=Turkish]{Minion Pro}
\babelFSstore{turkish}
\setmainfont{Minion Pro}
\babelFSfeatures{turkish}{Language=Turkish}
```

Bidi writing is taking its *first steps*. *First steps* means exactly that. For example, in `luatex` any Arabic text must be marked up explicitly in L mode. On the other hand, `xetex` poses quite different challenges. Document layout (lists, footnotes, etc.) is not touched at all. See the code section for `\foreignlanguage*` (a new starred version of `\foreignlanguage`). `xetex` relies on the font to properly handle these unmarked changes, so it is not under the control of `TEX`.

2 Loading languages with `language.dat`

`TEX` and most engines based on it (`pdfTEX`, `xetex`, `ε-TEX`, the main exception being `luatex`) require hyphenation patterns to be preloaded when a format is created (eg, `LATEX`, `XeLATEX`, `pdfLATEX`). `babel` provides a tool which has become standard in many distributions and based on a “configuration file” named `language.dat`. The exact way this file is used depends on the distribution, so please, read the documentation for the latter (note also some distributions generate the file with some tool).

New 3.9q With `luatex`, however, patterns are loaded on the fly when requested by the language (except the “0th” language, typically english, which is preloaded always).¹⁸ Until 3.9n, this task was delegated to the package `luatex-hyphen`, by Khaled Hosny, Élie Roux, and Manuel Pégourié-Gonnard, and required an extra file named `language.dat.lua`, but now a new mechanism has been devised based solely on `language.dat`. **You must rebuild the formats** if upgrading from a previous version. You may want to have a local `language.dat` for a particular project (for example, a book on Chemistry).¹⁹

2.1 Format

In that file the person who maintains a `TEX` environment has to record for which languages he has hyphenation patterns *and* in which files these are stored²⁰. When hyphenation exceptions are stored in a separate file this can be indicated by naming that file *after* the file with the hyphenation patterns.

The file can contain empty lines and comments, as well as lines which start with an equals (=) sign. Such a line will instruct `LATEX` that the hyphenation patterns just processed have to be known under an alternative name. Here is an example:

¹⁸This feature was added to 3.9o, but it was buggy. Both 3.9o and 3.9p are deprecated.

¹⁹The loader for `lua(e)tex` is slightly different as it's not based on `babel` but on `etex.src`. Until 3.9p it just didn't work, but thanks to the new code it works by reloading the data in the `babel` way, i.e., with `language.dat`.

²⁰This is because different operating systems sometimes use very different file-naming conventions.

```
% File      : language.dat
% Purpose   : tell iniTeX what files with patterns to load.
english    english.hyphenations
=british

dutch      hyphen.dutch exceptions.dutch % Nederlands
german     hyphen.ger
```

You may also set the font encoding the patterns are intended for by following the language name by a colon and the encoding code.²¹ For example:

```
german:T1 hyphenT1.ger
german hyphen.ger
```

With the previous settings, if the encoding when the language is selected is T1 then the patterns in `hyphenT1.ger` are used, but otherwise use those in `hyphen.ger` (note the encoding could be set in `\extras<lang>`).

A typical error when using `babel` is the following:

```
No hyphenation patterns were preloaded for
the language '<lang>' into the format.
Please, configure your TeX system to add them and
rebuild the format. Now I will use the patterns
preloaded for english instead}}
```

It simply means you must reconfigure `language.dat`, either by hand or with the tools provided by your distribution.

3 The interface between the core of babel and the language definition files

The *language definition files* (`ldf`) must conform to a number of conventions, because these files have to fill in the gaps left by the common code in `babel.def`, i. e., the definitions of the macros that produce texts. Also the language-switching possibility which has been built into the `babel` system has its implications.

The following assumptions are made:

- Some of the language-specific definitions might be used by plain \TeX users, so the files have to be coded so that they can be read by both \LaTeX and plain \TeX . The current format can be checked by looking at the value of the macro `\fmtname`.
- The common part of the `babel` system redefines a number of macros and environments (defined previously in the document style) to put in the names of macros that replace the previously hard-wired texts. These macros have to be defined in the language definition files.
- The language definition files must define five macros, used to activate and deactivate the language-specific definitions. These macros are `\<lang>hyphenmins`, `\captions<lang>`, `\date<lang>`, `\extras<lang>` and `\noextras<lang>` (the last two may be left empty); where `<lang>` is either the name of the language definition file or the name of the \LaTeX option that is to be used. These macros and their functions are

²¹This is not a new feature, but in former versions it didn't work correctly.

discussed below. You must define all or none for a language (or a dialect); defining, say, `\date⟨lang⟩` but not `\captions⟨lang⟩` does not raise an error but can lead to unexpected results.

- When a language definition file is loaded, it can define `\l@⟨lang⟩` to be a dialect of `\language0` when `\l@⟨lang⟩` is undefined.
- Language names must be all lowercase. If an unknown language is selected, babel will attempt setting it after lowercasing its name.
- The semantics of modifiers is not defined (on purpose). In most cases, they will just be simple separated options (eg, spanish), but a language might require, say, a set of options organized as a tree with suboptions (in such a case, the recommended separator is `/`).

Some recommendations:

- The preferred shorthand is `"`, which is not used in \LaTeX (quotes are entered as ``` and `'`). Other good choices are characters which are not used in a certain context (eg, `=` in an ancient language). Note however `=`, `<`, `>`, `:` and the like can be dangerous, because they may be used as part of the syntax of some elements (numeric expressions, key/value pairs, etc.).
- Captions should not contain shorthands or encoding dependent commands (the latter is not always possible, but should be clearly documented). They should be defined using the LICR. You may also use the new tools for encoded strings, described below.
- Avoid adding things to `\noextras⟨lang⟩` except for `umlauthigh` and friends, `\bbl@deactivate`, `\bbl@(non)frenchspacing`, and language specific macros. Use always, if possible, `\bbl@save` and `\bbl@savevariable` (except if you still want to have access to the previous value). Do not reset a macro or a setting to a hardcoded value. Never. Instead save its value in `\extras⟨lang⟩`.
- Do not switch scripts. If you want to make sure a set of glyphs is used, switch either the font encoding (low level) or the language (high level, which in turn may switch the font encoding). Usage of things like `\latintext` is deprecated.²²
- Please, for “private” internal macros do not use the `\bbl@` prefix. It is used by babel and it can lead to incompatibilities.

There are no special requirements for documenting your language files. Now they are not included in the base babel manual, so provide a standalone document suited for your needs, as well as other files you think can be useful. A PDF and a “readme” are strongly recommended.

3.1 Basic macros

In the core of the babel system, several macros are defined for use in language definition files. Their purpose is to make a new language known. The first two are related to hyphenation patterns.

- `\addlanguage` The macro `\addlanguage` is a non-outer version of the macro `\newlanguage`, defined in `plain.tex` version 3.x. For older versions of `plain.tex` and `lplain.tex` a substitute definition is used. Here “language” is used in the \TeX sense of set of hyphenation patterns.
- `\adddialect` The macro `\adddialect` can be used when two languages can (or must) use the same hyphenation patterns. This can also be useful for languages for which no patterns are preloaded in the format. In such cases the default behaviour of the babel system is to

²²But not removed, for backward compatibility.

`\<lang>hyphenmins`

define this language as a ‘dialect’ of the language for which the patterns were loaded as `\language0`. Here “language” is used in the \TeX sense of set of hyphenation patterns. The macro `\<lang>hyphenmins` is used to store the values of the `\lefthyphenmin` and `\righthyphenmin`. Redefine this macro to set your own values, with two numbers corresponding to these two parameters. For example:

```
\renewcommand\spanishhyphenmins{34}
```

(Assigning `\lefthyphenmin` and `\righthyphenmin` directly in `\extras<lang>` has no effect.)

`\providehyphenmins`

The macro `\providehyphenmins` should be used in the language definition files to set `\lefthyphenmin` and `\righthyphenmin`. This macro will check whether these parameters were provided by the hyphenation file before it takes any action. If these values have been already set, this command is ignored (currently, default pattern files do *not* set them).

`\captions<lang>`

The macro `\captions<lang>` defines the macros that hold the texts to replace the original hard-wired texts.

`\date<lang>`

The macro `\date<lang>` defines `\today`.

`\extras<lang>`

The macro `\extras<lang>` contains all the extra definitions needed for a specific language. This macro, like the following, is a hook – you can add things to it, but it must not be used directly.

`\noextras<lang>`

Because we want to let the user switch between languages, but we do not know what state \TeX might be in after the execution of `\extras<lang>`, a macro that brings \TeX into a predefined state is needed. It will be no surprise that the name of this macro is `\noextras<lang>`.

`\bbl@declare@tribute`

This is a command to be used in the language definition files for declaring a language attribute. It takes three arguments: the name of the language, the attribute to be defined, and the code to be executed when the attribute is to be used.

`\main@language`

To postpone the activation of the definitions needed for a language until the beginning of a document, all language definition files should use `\main@language` instead of `\selectlanguage`. This will just store the name of the language, and the proper language will be activated at the start of the document.

`\ProvidesLanguage`

The macro `\ProvidesLanguage` should be used to identify the language definition files. Its syntax is similar to the syntax of the \LaTeX command `\ProvidesPackage`.

`\LdfInit`

The macro `\LdfInit` performs a couple of standard checks that must be made at the beginning of a language definition file, such as checking the category code of the `@`-sign, preventing the `.ldf` file from being processed twice, etc.

`\ldf@quit`

The macro `\ldf@quit` does work needed if a `.ldf` file was processed earlier. This includes resetting the category code of the `@`-sign, preparing the language to be activated at `\begin{document}` time, and ending the input stream.

`\ldf@finish`

The macro `\ldf@finish` does work needed at the end of each `.ldf` file. This includes resetting the category code of the `@`-sign, loading a local configuration file, and preparing the language to be activated at `\begin{document}` time.

`\loadlocalcfg`

After processing a language definition file, \LaTeX can be instructed to load a local configuration file. This file can, for instance, be used to add strings to `\captions<lang>` to support local document classes. The user will be informed that this configuration file has been loaded. This macro is called by `\ldf@finish`.

`\substitutefontfamily`

(Deprecated.) This command takes three arguments, a font encoding and two font family names. It creates a font description file for the first font in the given encoding. This `.fd` file will instruct \LaTeX to use a font from the second family when a font from the first family in the given encoding seems to be needed.

3.2 Skeleton

Here is the basic structure of an ldf file, with a language, a dialect and an attribute. Strings are best defined using the method explained in in sec. 3.7 (babel 3.9 and later).

```
\ProvidesLanguage{<language>}
    [2016/04/23 v0.0 <Language> support from the babel system]
\LdfInit{<language>}{captions<language>}

\ifx\undefined\l@<language>
  \@nopatterns{<Language>}
  \adddialect\l@<language>0
\fi

\adddialect\l@<dialect>\l@<language>

\bbL@declare@ttribute{<language>}{<attrib>}{%
  \expandafter\addto\expandafter\extras<language>
  \expandafter{\extras<attrib><language>}%
  \let\captions<language>\captions<attrib><language>}

\providehyphenmins{<language>}{\tw@\thr@@}

\StartBabelCommands*{<language>}{captions}
\SetString\chaptername{<chapter name>}
% More strings

\StartBabelCommands*{<language>}{date}
\SetString\monthname{<name of first month>}
% More strings

\StartBabelCommands*{<dialect>}{captions}
\SetString\chaptername{<chapter name>}
% More strings

\StartBabelCommands*{<dialect>}{date}
\SetString\monthname{<name of first month>}
% More strings

\EndBabelCommands

\addto\extras<language>{}
\addto\noextras<language>{}
\let\extras<dialect>\extras<language>
\let\noextras<dialect>\noextras<language>

\ldf@finish{<language>}
```

3.3 Support for active characters

In quite a number of language definition files, active characters are introduced. To facilitate this, some support macros are provided.

`\initiate@active@char`

The internal macro `\initiate@active@char` is used in language definition files to instruct \LaTeX to give a character the category code ‘active’. When a character has been made active it will remain that way until the end of the document. Its definition may vary.

`\bbl@activate`
`\bbl@deactivate`

The command `\bbl@activate` is used to change the way an active character expands.

`\bbl@activate` ‘switches on’ the active behaviour of the character. `\bbl@deactivate` lets the active character expand to its former (mostly) non-active self.

`\declare@shorthand` The macro `\declare@shorthand` is used to define the various shorthands. It takes three arguments: the name for the collection of shorthands this definition belongs to; the character (sequence) that makes up the shorthand, i.e. `~` or `"a`; and the code to be executed when the shorthand is encountered. (It does *not* raise an error if the shorthand character has not been “initiated”.)

`\bbl@add@special`
`\bbl@remove@special` The \TeX book states: “Plain \TeX includes a macro called `\dospecials` that is essentially a set macro, representing the set of all characters that have a special category code.” [2, p. 380] It is used to set text ‘verbatim’. To make this work if more characters get a special category code, you have to add this character to the macro `\dospecials`. \LaTeX adds another macro called `\@sanitize` representing the same character set, but without the curly braces. The macros `\bbl@add@special⟨char⟩` and `\bbl@remove@special⟨char⟩` add and remove the character `⟨char⟩` to these two sets.

3.4 Support for saving macro definitions

Language definition files may want to *redefine* macros that already exist. Therefore a mechanism for saving (and restoring) the original definition of those macros is provided. We provide two macros for this²³.

`\babel@save` To save the current meaning of any control sequence, the macro `\babel@save` is provided. It takes one argument, `⟨cname⟩`, the control sequence for which the meaning has to be saved.

`\babel@savevariable` A second macro is provided to save the current value of a variable. In this context, anything that is allowed after the `\the` primitive is considered to be a variable. The macro takes one argument, the `⟨variable⟩`.

The effect of the preceding macros is to append a piece of code to the current definition of `\originalTeX`. When `\originalTeX` is expanded, this code restores the previous definition of the control sequence or the previous value of the variable.

3.5 Support for extending macros

`\addto` The macro `\addto{⟨control sequence⟩}{⟨ \TeX code⟩}` can be used to extend the definition of a macro. The macro need not be defined (ie, it can be undefined or `\relax`). This macro can, for instance, be used in adding instructions to a macro like `\extrasenglish`. Be careful when using this macro, because depending on the case the assignment could be either global (usually) or local (sometimes). That does not seem very consistent, but this behaviour is preserved for backward compatibility. If you are using `etoolbox`, by Philipp Lehman, consider using the tools provided by this package instead of `\addto`.

3.6 Macros common to a number of languages

`\bbl@allowhyphens` In several languages compound words are used. This means that when \TeX has to hyphenate such a compound word, it only does so at the ‘-’ that is used in such words. To allow hyphenation in the rest of such a compound word, the macro `\bbl@allowhyphens` can be used.

`\allowhyphens` Same as `\bbl@allowhyphens`, but does nothing if the encoding is T1. It is intended mainly for characters provided as real glyphs by this encoding but constructed with `\accent` in OT1.

Note the previous command (`\bbl@allowhyphens`) has different applications (hyphens and discretionaries) than this one (composite chars). Note also prior to version 3.7, `\allowhyphens` had the behaviour of `\bbl@allowhyphens`.

`\set@low@box` For some languages, quotes need to be lowered to the baseline. For this purpose the macro

²³This mechanism was introduced by Bernd Raichle.

`\set@low@box` is available. It takes one argument and puts that argument in an `\hbox`, at the baseline. The result is available in `\box0` for further processing.

`\save@sf@q` Sometimes it is necessary to preserve the `\spacefactor`. For this purpose the macro `\save@sf@q` is available. It takes one argument, saves the current `spacefactor`, executes the argument, and restores the `spacefactor`.

`\bbl@frenchspacing` The commands `\bbl@frenchspacing` and `\bbl@nonfrenchspacing` can be used to properly switch French spacing on and off.

`\bbl@nonfrenchspacing`

3.7 Encoding-dependent strings

New 3.9a Babel 3.9 provides a way of defining strings in several encodings, intended mainly for `luatex` and `xetex`. This is the only new feature requiring changes in language files if you want to make use of it.

Furthermore, it must be activated explicitly, with the package option `strings`. If there is no `strings`, these blocks are ignored, except `\SetCases` (and except if forced as described below). In other words, the old way of defining/switching strings still works and it's used by default.

It consist is a series of blocks started with `\StartBabelCommands`. The last block is closed with `\EndBabelCommands`. Each block is a single group (ie, local declarations apply until the next `\StartBabelCommands` or `\EndBabelCommands`). An `ldf` may contain several series of this kind.

Thanks to this new feature, string values and string language switching are not mixed any more. No need of `\addto`. If the language is `french`, just redefine `\frenchchaptername`.

`\StartBabelCommands` $\{ \langle \textit{language-list} \rangle \} \{ \langle \textit{category} \rangle \} [\langle \textit{selector} \rangle]$

The $\langle \textit{language-list} \rangle$ specifies which languages the block is intended for. A block is taken into account only if the `\CurrentOption` is listed here. Alternatively, you can define `\BabelLanguages` to a comma-separated list of languages to be defined (if undefined, `\StartBabelCommands` sets it to `\CurrentOption`). You may write `\CurrentOption` as the language, but this is discouraged – a explicit name (or names) is much better and clearer. A “selector” is a name to be used as value in package option strings, optionally followed by extra info about the encodings to be used. The name `unicode` must be used for `xetex` and `luatex` (the key `strings` has also other two special values: `generic` and `encoded`). If a string is set several times (because several blocks are read), the first one take precedence (ie, it works much like `\providecommand`).

Encoding info is `charset=` followed by a `charset`, which if given sets how the strings should be traslated to the internal representation used by the engine, typically `utf8`, which is the only value supported currently (default is no traslations). Note `charset` is applied by `luatex` and `xetex` when reading the file, not when the macro or string is used in the document. A list of font encodings which the strings are expected to work with can be given after `fontenc=` (separated with spaces, if two or more) – recommended, but not mandatory, although blocks without this key are not taken into account if you have requested `strings=encoded`.

Blocks without a selector are read always if the key `strings` has been used. They provide fallback values, and therefore must be the last blocks; they should be provided always if possible and all strings should be defined somehow inside it; they can be the only blocks (mainly LGC scripts using the LICR). Blocks without a selector can be activated explicitly with `strings=generic` (no block is taken into account except those). With `strings=encoded`, strings in those blocks are set as default (internally, `?`). With `strings=encoded` strings are protected, but they are correctly expanded in `\MakeUppercase` and the like. If there is no key `strings`, string definitions are ignored, but `\SetCases` are still honoured (in a encoded way).

The $\langle category \rangle$ is either captions, date or extras. You must stick to these three categories, even if no error is raised when using other name.²⁴ It may be empty, too, but in such a case using $\backslash SetString$ is an error (but not $\backslash SetCase$).

```
\StartBabelCommands{language}{captions}
  [unicode, fontenc=TU EU1 EU2, charset=utf8]
\SetString{\chaptername}{utf8-string}

\StartBabelCommands{language}{captions}
\SetString{\chaptername}{ascii-maybe-LICR-string}

\EndBabelCommands
```

A real example is:

```
\StartBabelCommands{austrian}{date}
  [unicode, fontenc=TU EU1 EU2, charset=utf8]
\SetString\monthiname{Jänner}

\StartBabelCommands{german,austrian}{date}
  [unicode, fontenc=TU EU1 EU2, charset=utf8]
\SetString\monthiiiname{März}

\StartBabelCommands{austrian}{date}
\SetString\monthiname{J\"{a}nner}

\StartBabelCommands{german}{date}
\SetString\monthiname{Januar}

\StartBabelCommands{german,austrian}{date}
\SetString\monthiiiname{Februar}
\SetString\monthiiiname{M\"{a}rz}
\SetString\monthivname{April}
\SetString\monthvname{Mai}
\SetString\monthviname{Juni}
\SetString\monthviiname{Juli}
\SetString\monthviiiname{August}
\SetString\monthixname{September}
\SetString\monthxname{Oktober}
\SetString\monthxiname{November}
\SetString\monthxiiname{Dezenber}
\SetString\today{\number\day.\~%
  \csname month\romannumeral\month name\endcsname\space
  \number\year}

\StartBabelCommands{german,austrian}{captions}
\SetString\prefacename{Vorwort}
[etc.]

\EndBabelCommands
```

When used in ldf files, previous values of $\backslash \langle category \rangle \langle language \rangle$ are overridden, which means the old way to define strings still works and used by default (to be precise, is first set to undefined and then strings are added). However, when used in the preamble or in a package, new settings are added to the previous ones, if the language exists (in the babel sense, ie, if $\backslash date \langle language \rangle$ exists).

²⁴In future releases further categories may be added.

\StartBabelCommands *{\<language-list>}{\<category>}[\<selector>]

The starred version just forces strings to take a value – if not set as package option, then the default for the engine is used. This is not done by default to prevent backward incompatibilities, but if you are creating a new language this version is better. It's up to the maintainers of the current languages to decide if using it is appropriate.²⁵

\EndBabelCommands Marks the end of the series of blocks.

\AfterBabelCommands {\<code>}

The code is delayed and executed at the global scope just after \EndBabelCommands.

\SetString {\<macro-name>}{\<string>}

Adds \<macro-name> to the current category, and defines globally \<lang-macro-name> to \<code> (after applying the transformation corresponding to the current charset or defined with the hook stringprocess).

Use this command to define strings, without including any “logic” if possible, which should be a separated macro. See the example above for the date.

\SetStringLoop {\<macro-name>}{\<string-list>}

A convenient way to define several ordered names at once. For example, to define \abmoniname, \abmoniiname, etc. (and similarly with abday):

```
\SetStringLoop{abmon#1name}{en,fb,mr,ab,my,jn,jl,ag,sp,oc,nv,dc}
\SetStringLoop{abday#1name}{lu,ma,mi,ju,vi,sa,do}
```

#1 is replaced by the roman numeral.

\SetCase [\<map-list>]{\<toupper-code>}{\<tolower-code>}

Sets globally code to be executed at \MakeUppercase and \MakeLowercase. The code would be typically things like \let\BB\bb and \uccode or \lccode (although for the reasons explained above, changes in lc/uc codes may not work). A \<map-list> is a series of macros using the internal format of \@uclclist (eg, \bb\BB\cc\CC). The mandatory arguments take precedence over the optional one. This command, unlike \SetString, is executed always (even without strings), and it is intended for minor readjustments only. For example, as T1 is the default case mapping in L^AT_EX, we could set for Turkish:

```
\StartBabelCommands{turkish}{}[ot1enc, fontenc=OT1]
\SetCase
{\uccode"10=`I\relax}
{\lccode`I="10\relax}

\StartBabelCommands{turkish}{}[unicode, fontenc=TU EU1 EU2, charset=utf8]
\SetCase
{\uccode`i=`I\relax
 \uccode`ı=`I\relax}
{\lccode`İ=`i\relax
 \lccode`I=`ı\relax}

\StartBabelCommands{turkish}{}[
```

²⁵This replaces in 3.9g a short-lived \UseStrings which has been removed because it did not work.

```

\SetCase
{\uccode`i="9D\relax
 \uccode"19=`I\relax}
{\lccode"9D=`i\relax
 \lccode`I="19\relax}

\EndBabelCommands

```

(Note the mapping for OT1 is not complete.)

`\SetHyphenMap` {*<to-lower-macros>*}

New 3.9g Case mapping serves in T_EX for two unrelated purposes: case transforms (upper/lower) and hyphenation. `\SetCase` handles the former, while hyphenation is handled by `\SetHyphenMap` and controlled with the package option `hyphenmap`. So, even if internally they are based on the same T_EX primitive (`\lccode`), babel sets them separately. There are three helper macros to be used inside `\SetHyphenMap`:

- `\BabelLower{<uccode>}{<lccode>}` is similar to `\lccode` but it's ignored if the char has been set and saves the original lccode to restore it when switching the language (except with `hyphenmap=first`).
- `\BabelLowerMM{<uccode-from>}{<uccode-to>}{<step>}{<lccode-from>}` loops through the given uppercase codes, using the step, and assigns them the lccode, which is also increased (MM stands for *many-to-many*).
- `\BabelLowerMO{<uccode-from>}{<uccode-to>}{<step>}{<lccode>}` loops through the given uppercase codes, using the step, and assigns them the lccode, which is fixed (MO stands for *many-to-one*).

An example is (which is redundant, because these assignments are done by both `luatex` and `xetex`):

```

\SetHyphenMap{\BabelLowerMM{"100}{ "11F}{2}{ "101}}

```

This macro is not intended to fix wrong mappings done by Unicode (which are the default in both `xetex` and `luatex`) – if an assignment is wrong, fix it directly.

4 Changes

4.1 Changes in babel version 3.9

Most of changes in version 3.9 are related to bugs, either to fix them (there were lots), or to provide some alternatives. Even new features like `\babelhyphen` are intended to solve a certain problem (in this case, the lacking of a uniform syntax and behaviour for shorthands across languages). These changes are described in this manual in the corresponding place. A selective list follows:

- `\select@language` did not set `\language`name. This meant the language in force when auxiliary files were loaded was the one used in, for example, shorthands – if the language was `german`, a `\select@language{spanish}` had no effect.
- `\foreignlanguage` and `otherlanguage*` messed up `\extras<language>`. Scripts, encodings and many other things were not switched correctly.

- The :ENC mechanism for hyphenation patterns used the encoding of the *previous* language, not that of the language being selected.
- ' (with activeacute) had the original value when writing to an auxiliary file, and things like an infinite loop could happen. It worked incorrectly with ^ (if activated) and also if deactivated.
- Active chars were not reset at the end of language options, and that led to incompatibilities between languages.
- \textormath raised an error with a conditional.
- \aliasshorthand didn't work (or only in a few and very specific cases).
- \l@english was defined incorrectly (using \let instead of \chardef).
- ldf files not bundled with babel were not recognized when called as global options.

4.2 Changes in babel version 3.7

In babel version 3.7 a number of bugs that were found in version 3.6 are fixed. Also a number of changes and additions have occurred:

- Shorthands are expandable again. The disadvantage is that one has to type '{a when the acute accent is used as a shorthand character. The advantage is that a number of other problems (such as the breaking of ligatures, etc.) have vanished.
- Two new commands, \shorthandon and \shorthandoff have been introduced to enable to temporarily switch off one or more shorthands.
- Support for typesetting Hebrew (and potential support for typesetting other right-to-left written languages) is now available thanks to Rama Porrat and Boris Lavva.
- A language attribute has been added to the \mark... commands in order to make sure that a Greek header line comes out right on the last page before a language switch.
- Hyphenation pattern files are now read *inside a group*; therefore any changes a pattern file needs to make to lowercase codes, uppercase codes, and category codes are kept local to that group. If they are needed for the language, these changes will need to be repeated and stored in \extras...
- The concept of language attributes is introduced. It is intended to give the user some control over the features a language-definition file provides. Its first use is for the Greek language, where the user can choose the πολυτονικό ("polytonikó" or multi-accented) Greek way of typesetting texts.
- The environment hyphenrules is introduced.
- The syntax of the file language.dat has been extended to allow (optionally) specifying the font encoding to be used while processing the patterns file.
- The command \providehyphenmins should now be used in language definition files in order to be able to keep any settings provided by the pattern file.

Part II

The code

babel is being developed incrementally, which means parts of the code are under development and therefore incomplete. Only documented features are considered

complete. In other words, use babel only as documented (except, of course, if you want to explore and test them – you can post suggestions about multilingual issues to kadingira@tug.org on <http://tug.org/mailman/listinfo/kadingira>).

5 Identification and loading of required files

Code documentation is still under revision.

The babel package after unpacking consists of the following files:

switch.def defines macros to set and switch languages.

babel.def defines the rest of macros. It has two parts: a generic one and a second one only for LaTeX.

babel.sty is the \LaTeX package, which sets options and loads language styles.

plain.def defines some \LaTeX macros required by babel.def and provides a few tools for Plain.

hyphen.cfg is the file to be used when generating the formats to load hyphenation patterns. By default it also loads switch.def.

The babel installer extends docstrip with a few “pseudo-guards” to set “variables” used at installation time. They are used with `<@name@>` at the appropriated places in the source code and shown below with `<<name>>`. That brings a little bit of literate programming.

```
1 <<version=3.15>>
2 <<date=2017/11/03>>
```

6 Tools

Do not use the following macros in ldf files. They may change in the future. This applies mainly to those recently added for replacing, trimming and looping. The older ones, like `\bbl@afterfi`, will not change.

We define some basic macros which just make the code cleaner. `\bbl@add` is now used internally instead of `\addto` because of the unpredictable behaviour of the latter. Used in `babel.def` and in `babel.sty`, which means in \LaTeX is executed twice, but we need them when defining options and `babel.def` cannot be loaded until options have been defined. This does not hurt, but should be fixed somehow.

```
3 <<*Basic macros>> ≡
4 \def\bbl@stripslash{\expandafter\@gobble\string}
5 \def\bbl@add#1#2{%
6   \bbl@ifunset{\bbl@stripslash#1}%
7   {\def#1{#2}}%
8   {\expandafter\def\expandafter#1\expandafter{#1#2}}
9 \def\bbl@xin@{\@expandtwoargs\in@}
10 \def\bbl@csarg#1#2{\expandafter#1\csname bbl@#2\endcsname}%
11 \def\bbl@cs#1{\csname bbl@#1\endcsname}
12 \def\bbl@loop#1#2#3{\bbl@loop#1{#3}#2,\@nnil,}
13 \def\bbl@loopx#1#2{\expandafter\bbl@loop\expandafter#1\expandafter{#2}}
14 \def\bbl@loop#1#2#3,{%
15   \ifx\@nnil#3\relax\else
16     \def#1{#3}#2\bbl@afterfi\bbl@loop#1{#2}%
17   \fi}
18 \def\bbl@for#1#2#3{\bbl@loopx#1{#2}{\ifx#1\@empty\else#3\fi}}
```

`\bbl@add@list` This internal macro adds its second argument to a comma separated list in its first argument. When the list is not defined yet (or empty), it will be initiated. It presumes expandable character strings.

```

19 \def\bbl@add@list#1#2{%
20   \edef#1{%
21     \bbl@ifunset{\bbl@stripslash#1}%
22     {}%
23     {\ifx#1\@empty\else#1,\fi}%
24     #2}}

```

`\bbl@afterelse` `\bbl@afterfi` Because the code that is used in the handling of active characters may need to look ahead, we take extra care to ‘throw’ it over the `\else` and `\fi` parts of an `\if`-statement²⁶. These macros will break if another `\if... \fi` statement appears in one of the arguments and it is not enclosed in braces.

```

25 \long\def\bbl@afterelse#1\else#2\fi{\fi#1}
26 \long\def\bbl@afterfi#1\fi{\fi#1}

```

`\bbl@trim` The following piece of code is stolen (with some changes) from `keyval`, by David Carlisle. It defines two macros: `\bbl@trim` and `\bbl@trim@def`. The first one strips the leading and trailing spaces from the second argument and then applies the first argument (a macro, `\toks@` and the like). The second one, as its name suggests, defines the first argument as the stripped second argument.

```

27 \def\bbl@tempa#1{%
28   \long\def\bbl@trim##1##2{%
29     \futurelet\bbl@trim@a\bbl@trim@c##2\@nil\@nil#1\@nil\relax{##1}}%
30   \def\bbl@trim@c{%
31     \ifx\bbl@trim@a\sptoken
32       \expandafter\bbl@trim@b
33     \else
34       \expandafter\bbl@trim@b\expandafter#1%
35     \fi}%
36   \long\def\bbl@trim@b#1##1 \@nil{\bbl@trim@i##1}}
37 \bbl@tempa{ }
38 \long\def\bbl@trim@i#1\@nil#2\relax#3{#3{#1}}
39 \long\def\bbl@trim@def#1{\bbl@trim{\def#1}}

```

`\bbl@ifunset` To check if a macro is defined, we create a new macro, which does the same as `\@ifundefined`. However, in an ϵ -tex engine, it is based on `\ifcsname`, which is more efficient, and do not waste memory.

```

40 \def\bbl@ifunset#1{%
41   \expandafter\ifx\csname#1\endcsname\relax
42     \expandafter\@firstoftwo
43   \else
44     \expandafter\@secondoftwo
45   \fi}
46 \bbl@ifunset{ifcsname}%
47 {}%
48 {\def\bbl@ifunset#1{%
49   \ifcsname#1\endcsname
50     \expandafter\ifx\csname#1\endcsname\relax
51       \bbl@afterelse\expandafter\@firstoftwo
52     \else
53       \bbl@afterfi\expandafter\@secondoftwo
54     \fi
55   \else
56     \expandafter\@firstoftwo
57   \fi}}

```

²⁶This code is based on code presented in TUGboat vol. 12, no2, June 1991 in “An expansion Power Lemma” by Sonja Maus.

`\bbl@ifblank` A tool from url, by Donald Arseneau, which tests if a string is empty or space.

```
58 \def\bbl@ifblank#1{%
59   \bbl@ifblank@i#1\@nil\@nil\@secondoftwo\@firstoftwo\@nil}
60 \long\def\bbl@ifblank@i#1#2\@nil#3#4#5\@nil{#4}
```

For each element in the comma separated <key>=<value> list, execute <code> with #1 and #2 as the key and the value of current item (trimmed). In addition, the item is passed verbatim as #3. With the <key> alone, it passes \@empty (ie, the macro thus named, not an empty argument, which is what you get with <key>= and no value).

```
61 \def\bbl@forkv#1#2{%
62   \def\bbl@kvcmd##1##2##3{#2}%
63   \bbl@kvnext#1,\@nil,}
64 \def\bbl@kvnext#1,{%
65   \ifx\@nil#1\relax\else
66     \bbl@ifblank{#1}{\bbl@forkv@eq#1=\@empty=\@nil{#1}}%
67     \expandafter\bbl@kvnext
68   \fi}
69 \def\bbl@forkv@eq#1=#2=#3\@nil#4{%
70   \bbl@trim@def\bbl@forkv@a{#1}%
71   \bbl@trim{\expandafter\bbl@kvcmd\expandafter{\bbl@forkv@a}}{#2}{#4}}
```

A *for* loop. Each item (trimmed), is #1. It cannot be nested (it's doable, but we don't need it).

```
72 \def\bbl@vforeach#1#2{%
73   \def\bbl@forcmd##1{#2}%
74   \bbl@fornext#1,\@nil,}
75 \def\bbl@fornext#1,{%
76   \ifx\@nil#1\relax\else
77     \bbl@ifblank{#1}{\bbl@trim\bbl@forcmd{#1}}%
78     \expandafter\bbl@fornext
79   \fi}
80 \def\bbl@foreach#1{\expandafter\bbl@vforeach\expandafter{#1}}
```

`\bbl@replace`

```
81 \def\bbl@replace#1#2#3{% in #1 -> repl #2 by #3
82   \toks@{}}%
83 \def\bbl@replace@aux##1#2##2#2{%
84   \ifx\bbl@nil##2%
85     \toks@\expandafter{\the\toks@##1}%
86   \else
87     \toks@\expandafter{\the\toks@##1#3}%
88     \bbl@afterfi
89     \bbl@replace@aux##2#2%
90   \fi}%
91 \expandafter\bbl@replace@aux#1#2\bbl@nil#2%
92 \edef#1{\the\toks@}}
```

`\bbl@exp` Now, just syntactical sugar, but it makes partial expansion of some code a lot more simple and readable. Here `\` stands for `\noexpand` and `\<. .>` for `\noexpand` applied to a built macro name (the latter does not define the macro if undefined to `\relax`, because it is created locally). The result may be followed by extra arguments, if necessary.

```
93 \def\bbl@exp#1{%
94   \begingroup
95   \let\\noexpand
96   \def\<##1>{\expandafter\noexpand\csname##1\endcsname}%
97   \edef\bbl@exp@aux{\endgroup#1}%
98   \bbl@exp@aux}
```


Two further tools. `\bbl@samestring` first expand its arguments and then compare their expansion (sanitized, so that the catcodes do not matter). `\bbl@engine` takes the following values: 0 is pdf \TeX , 1 is luatex, and 2 is xetex. You may use the latter in your language style if you want.

```

99 \def\bbl@ifsamestring#1#2{%
100 \begingroup
101   \protected@edef\bbl@tempb{#1}%
102   \edef\bbl@tempb{\expandafter\strip@prefix\meaning\bbl@tempb}%
103   \protected@edef\bbl@tempc{#2}%
104   \edef\bbl@tempc{\expandafter\strip@prefix\meaning\bbl@tempc}%
105   \ifx\bbl@tempb\bbl@tempc
106     \aftergroup\@firstoftwo
107   \else
108     \aftergroup\@secondoftwo
109   \fi
110 \endgroup}
111 \chardef\bbl@engine=%
112 \ifx\directlua\@undefined
113   \ifx\XeTeXinputencoding\@undefined
114     \z@
115   \else
116     \tw@
117   \fi
118 \else
119   \@ne
120 \fi
121 <</Basic macros>>

```

Some files identify themselves with a \LaTeX macro. The following code is placed before them to define (and then undefine) if not in \LaTeX .

```

122 <<*Make sure ProvidesFile is defined>> ≡
123 \ifx\ProvidesFile\@undefined
124   \def\ProvidesFile#1[#2 #3 #4]{%
125     \wlog{File: #1 #4 #3 <#2>}%
126     \let\ProvidesFile\@undefined}
127 \fi
128 <</Make sure ProvidesFile is defined>>

```

The following code is used in `babel.sty` and `babel.def`, and loads (only once) the data in `language.dat`.

```

129 <<*Load patterns in luatex>> ≡
130 \ifx\directlua\@undefined\else
131   \ifx\bbl@luapatterns\@undefined
132     \input luababel.def
133   \fi
134 \fi
135 <</Load patterns in luatex>>

```

The following code is used in `babel.def` and `switch.def`.

```

136 <<*Load macros for plain if not LaTeX>> ≡
137 \ifx\AtBeginDocument\@undefined
138   \input plain.def\relax
139 \fi
140 <</Load macros for plain if not LaTeX>>

```

6.1 Multiple languages

`\language` Plain \TeX version 3.0 provides the primitive `\language` that is used to store the current language. When used with a pre-3.0 version this function has to be implemented by

allocating a counter. The following block is used in `switch.def` and `hyphen.cfg`; the latter may seem redundant, but remember `babel` doesn't require loading `switch.def` in the format.

```
141 <<*Define core switching macros>> ≡
142 \ifx\language\undefined
143   \csname newcount\endcsname\language
144 \fi
145 <</Define core switching macros>>
```

`\last@language` Another counter is used to store the last language defined. For pre-3.0 formats an extra counter has to be allocated.

`\addlanguage` To add languages to T_EX's memory plain T_EX version 3.0 supplies `\newlanguage`, in a pre-3.0 environment a similar macro has to be provided. For both cases a new macro is defined here, because the original `\newlanguage` was defined to be `\outer`. For a format based on plain version 2.x, the definition of `\newlanguage` can not be copied because `\count 19` is used for other purposes in these formats. Therefore `\addlanguage` is defined using a definition based on the macros used to define `\newlanguage` in plain T_EX version 3.0.

For formats based on plain version 3.0 the definition of `\newlanguage` can be simply copied, removing `\outer`. Plain T_EX version 3.0 uses `\count 19` for this purpose.

```
146 <<*Define core switching macros>> ≡
147 \ifx\newlanguage\undefined
148   \csname newcount\endcsname\last@language
149   \def\addlanguage#1{%
150     \global\advance\last@language\@ne
151     \ifnum\last@language<\@ccclvi
152       \else
153         \errmessage{No room for a new \string\language!}%
154       \fi
155     \global\chardef#1\last@language
156     \wlog{\string#1 = \string\language\the\last@language}}
157   \else
158     \countdef\last@language=19
159     \def\addlanguage{\alloc@9\language\chardef\@ccclvi}
160   \fi
161 <</Define core switching macros>>
```

Now we make sure all required files are loaded. When the command `\AtBeginDocument` doesn't exist we assume that we are dealing with a plain-based format or L^AT_EX 2.09. In that case the file `plain.def` is needed (which also defines `\AtBeginDocument`, and therefore it is not loaded twice). We need the first part when the format is created, and `\orig@dump` is used as a flag. Otherwise, we need to use the second part, so `\orig@dump` is not defined (`plain.def` undefines it).

Check if the current version of `switch.def` has been previously loaded (mainly, `hyphen.cfg`). If not, load it now. We cannot load `babel.def` here because we first need to declare and process the package options.

7 The Package File (L^AT_EX, `babel.sty`)

In order to make use of the features of L^AT_EX 2_ε, the `babel` system contains a package file, `babel.sty`. This file is loaded by the `\usepackage` command and defines all the language options whose name is different from that of the `.ldf` file (like variant spellings). It also takes care of a number of compatibility issues with other packages and defines a few additional package options.

Apart from all the language options below we also have a few options that influence the behaviour of language definition files.

Many of the following options don't do anything themselves, they are just defined in order to make it possible for babel and language definition files to check if one of them was specified by the user.

7.1 base

The first option to be processed is base, which set the hyphenation patterns then resets `ver@babel.sty` so that \LaTeX forgets about the first loading. After `switch.def` has been loaded (above) and `\AfterBabelLanguage` defined, exits.

```

162 (*package)
163 \NeedsTeXFormat{LaTeX2e}[2005/12/01]
164 \ProvidesPackage{babel}[<<date>> <<version>> The Babel package]
165 \@ifpackagewith{babel}{debug}
166   {\let\bbl@debug\@firstofone}
167   {\let\bbl@debug\gobble}
168 \input switch.def\relax
169 <<Load patterns in luatex>>
170 <<Basic macros>>
171 \def\AfterBabelLanguage#1{%
172   \global\expandafter\bbl@add\csname#1.ldf-h@@k\endcsname}%

```

If the format created a list of loaded languages (in `\bbl@languages`), get the name of the 0-th to show the actual language used.

```

173 \ifx\bbl@languages\undefined\else
174   \begingroup
175     \catcode`\^^I=12
176     \@ifpackagewith{babel}{showlanguages}{%
177       \begingroup
178         \def\bbl@elt#1#2#3#4{\wlog{#2^^I#1^^I#3^^I#4}}%
179         \wlog{<*languages>}%
180         \bbl@languages
181         \wlog{</languages>}%
182       \endgroup}{%
183     \endgroup
184     \def\bbl@elt#1#2#3#4{%
185       \ifnum#2=\z@
186         \gdef\bbl@nulllanguage{#1}%
187         \def\bbl@elt##1##2##3##4{}%
188       \fi}%
189     \bbl@languages
190 \fi
191 \@ifpackagewith{babel}{bidi=basic-r}{% must go before any \DeclareOption
192   \let\bbl@beforeforeign\leavevmode
193   \AtEndOfPackage{\EnableBabelHook{babel-bidi}}%
194   \RequirePackage{luatexbase}%
195   \directlua{
196     require('babel-bidi.lua')
197     require('babel-bidi-basic-r.lua')
198     luatexbase.add_to_callback('pre_linebreak_filter',
199       Babel.pre_otfload,
200       'Babel.pre_otfload',
201       luatexbase.priority_in_callback('pre_linebreak_filter',
202         'luaotfload.node_processor') or nil)
203     luatexbase.add_to_callback('hpack_filter',
204       Babel.pre_otfload,

```

```

205     'Babel.pre_otfload',
206     luatexbase.priority_in_callback('hpack_filter',
207     'luaotfload.node_processor') or nil)}}{}

```

Now the base option. With it we can define (and load, with `luatex`) hyphenation patterns, even if we are not interested in the rest of `babel`. Useful for old versions of `polyglossia`, too.

```

208 \@ifpackagewith{babel}{base}{%
209   \ifx\directlua\@undefined
210     \DeclareOption*{\bbl@patterns{\CurrentOption}}%
211   \else
212     \DeclareOption*{\bbl@patterns@lua{\CurrentOption}}%
213   \fi
214   \DeclareOption{base}{}%
215   \DeclareOption{showlanguages}{}%
216   \ProcessOptions
217   \global\expandafter\let\csname opt@babel.sty\endcsname\relax
218   \global\expandafter\let\csname ver@babel.sty\endcsname\relax
219   \global\let@ifl@ter@@\ifl@ter
220   \def\@ifl@ter#1#2#3#4#5{\global\let@ifl@ter\@ifl@ter@@}%
221   \endinput}}{}%

```

7.2 key=value options and other general option

The following macros extract language modifiers, and only real package options are kept in the option list. Modifiers are saved and assigned to `\BabelModifiers` at `\bbl@load@language`; when no modifiers have been given, the former is `\relax`. How modifiers are handled are left to language styles; they can use `\in@`, loop them with `\@for` or `load keyval`, for example).

```

222 \bbl@csarg\let{tempa\expandafter}\csname opt@babel.sty\endcsname
223 \def\bbl@tempb#1.#2{%
224   #1\ifx\@empty#2\else,\bbl@afterfi\bbl@tempb#2\fi}%
225 \def\bbl@tempd#1.#2\@nnil{%
226   \ifx\@empty#2%
227     \edef\bbl@tempc{\ifx\bbl@tempc\@empty\else\bbl@tempc,\fi#1}%
228   \else
229     \in@{=}{#1}\ifin@
230     \edef\bbl@tempc{\ifx\bbl@tempc\@empty\else\bbl@tempc,\fi#1.#2}%
231   \else
232     \edef\bbl@tempc{\ifx\bbl@tempc\@empty\else\bbl@tempc,\fi#1}%
233     \bbl@csarg\edef{mod@#1}{\bbl@tempb#2}%
234   \fi
235 \fi}
236 \let\bbl@tempc\@empty
237 \bbl@foreach\bbl@tempa{\bbl@tempd#1.\@empty\@nnil}
238 \expandafter\let\csname opt@babel.sty\endcsname\bbl@tempc

```

The next option tells `babel` to leave shorthand characters active at the end of processing the package. This is *not* the default as it can cause problems with other packages, but for those who want to use the shorthand characters in the preamble of their documents this can help.

```

239 \DeclareOption{KeepShorthandsActive}{}
240 \DeclareOption{activeacute}{}
241 \DeclareOption{activegrave}{}
242 \DeclareOption{debug}{}
243 \DeclareOption{noconfigs}{}
244 \DeclareOption{showlanguages}{}
245 \DeclareOption{silent}{}

```

```

246 \DeclareOption{shorthands=off}{\bbl@tempa shorthands=\bbl@tempa}
247 <<More package options>>

```

Handling of package options is done in three passes. (I [JBL] am not very happy with the idea, anyway.) The first one processes options which has been declared above or follow the syntax `<key>=<value>`, the second one loads the requested languages, except the main one if set with the key `main`, and the third one loads the latter. First, we “flag” valid keys with a `nil` value.

```

248 \let\bbl@opt@shorthands\@nnil
249 \let\bbl@opt@config\@nnil
250 \let\bbl@opt@main\@nnil
251 \let\bbl@opt@headfoot\@nnil

```

The following tool is defined temporarily to store the values of options.

```

252 \def\bbl@tempa#1=#2\bbl@tempa{%
253   \bbl@csarg\ifx{opt@#1}\@nnil
254     \bbl@csarg\edef{opt@#1}{#2}%
255   \else
256     \bbl@error{%
257       Bad option `#1=#2'. Either you have misspelled the\\%
258       key or there is a previous setting of `#1'}{%
259       Valid keys are `shorthands', `config', `strings', `main',\\%
260       `headfoot', `safe', `math'}
261   \fi}

```

Now the option list is processed, taking into account only currently declared options (including those declared with a `=`), and `<key>=<value>` options (the former take precedence). Unrecognized options are saved in `\bbl@language@opts`, because they are language options.

```

262 \let\bbl@language@opts\@empty
263 \DeclareOption*{%
264   \bbl@xin@{\string=}{\CurrentOption}%
265   \ifin@
266     \expandafter\bbl@tempa\CurrentOption\bbl@tempa
267   \else
268     \bbl@add@list\bbl@language@opts{\CurrentOption}%
269   \fi}

```

Now we finish the first pass (and start over).

```

270 \ProcessOptions*

```

7.3 Conditional loading of shorthands

If there is no `shorthands=<chars>`, the original babel macros are left untouched, but if there is, these macros are wrapped (in `babel.def`) to define only those given. A bit of optimization: if there is no `shorthands=`, then `\bbl@ifshorthands` is always true, and it is always false if `shorthands` is empty. Also, some code makes sense only with `shorthands=...`

```

271 \def\bbl@sh@string#1{%
272   \ifx#1\@empty\else
273     \ifx#1t\string~%
274     \else\ifx#1c\string,%
275     \else\string#1%
276   \fi\fi
277   \expandafter\bbl@sh@string
278   \fi}
279 \ifx\bbl@opt@shorthands\@nnil

```

```

280 \def\bbl@ifshorthand#1#2#3{#2}%
281 \else\ifx\bbl@opt@shorthands\@empty
282 \def\bbl@ifshorthand#1#2#3{#3}%
283 \else

```

The following macro tests if a shorthand is one of the allowed ones.

```

284 \def\bbl@ifshorthand#1{%
285   \bbl@xin@{\string#1}{\bbl@opt@shorthands}%
286   \ifin@
287     \expandafter\@firstoftwo
288   \else
289     \expandafter\@secondoftwo
290   \fi}

```

We make sure all chars in the string are ‘other’, with the help of an auxiliary macro defined above (which also zaps spaces).

```

291 \edef\bbl@opt@shorthands{%
292   \expandafter\bbl@sh@string\bbl@opt@shorthands\@empty}%

```

The following is ignored with shorthands=off, since it is intended to take some additional actions for certain chars.

```

293 \bbl@ifshorthand{'}%
294   {\PassOptionsToPackage{activeacute}{babel}}{}
295 \bbl@ifshorthand{`}%
296   {\PassOptionsToPackage{activegrave}{babel}}{}
297 \fi\fi

```

With headfoot=lang we can set the language used in heads/foots. For example, in babel/3796 just adds headfoot=english. It misuses \@resetactivechars but seems to work.

```

298 \ifx\bbl@opt@headfoot\@nnil\else
299   \g@addto@macro\@resetactivechars{%
300     \set@typeset@protect
301     \expandafter\select@language@x\expandafter{\bbl@opt@headfoot}%
302     \let\protect\noexpand}
303 \fi

```

For the option safe we use a different approach – \bbl@opt@safe says which macros are redefined (B for bibs and R for refs). By default, both are set.

```

304 \ifx\bbl@opt@safe\@undefined
305   \def\bbl@opt@safe{BR}
306 \fi
307 \ifx\bbl@opt@main\@nnil\else
308   \edef\bbl@language@opts{%
309     \ifx\bbl@language@opts\@empty\else\bbl@language@opts,\fi
310     \bbl@opt@main}
311 \fi

```

7.4 Language options

Languages are loaded when processing the corresponding option *except* if a main language has been set. In such a case, it is not loaded until all options has been processed. The following macro inputs the ldf file and does some additional checks (\input works, too, but possible errors are not caught).

```

312 \let\bbl@afterlang\relax
313 \let\BabelModifiers\relax
314 \let\bbl@loaded\@empty

```

```

315 \def\bbl@load@language#1{%
316   \InputIfFileExists{#1.ldf}%
317   {\edef\bbl@loaded{\CurrentOption
318     \ifx\bbl@loaded\@empty\else,\bbl@loaded\fi}%
319     \expandafter\let\expandafter\bbl@afterlang
320     \csname\CurrentOption.ldf-h@@k\endcsname
321     \expandafter\let\expandafter\BabelModifiers
322     \csname bbl@mod@\CurrentOption\endcsname}%
323   {\bbl@error{%
324     Unknown option '\CurrentOption'. Either you misspelled it\\%
325     or the language definition file \CurrentOption.ldf was not found}}{%
326     Valid options are: shorthands=, KeepShorthandsActive,\\%
327     activeacute, activegrave, noconfigs, safe=, main=, math=\\%
328     headfoot=, strings=, config=, hyphenmap=, or a language name.}}}

```

Now, we set language options whose names are different from ldf files.

```

329 \def\bbl@try@load@lang#1#2#3{%
330   \IfFileExists{\CurrentOption.ldf}%
331   {\bbl@load@language{\CurrentOption}}%
332   {#1\bbl@load@language{#2}#3}}
333 \DeclareOption{afrikaans}{\bbl@try@load@lang{}{dutch}{}}
334 \DeclareOption{brazil}{\bbl@try@load@lang{}{portuges}{}}
335 \DeclareOption{brazilian}{\bbl@try@load@lang{}{portuges}{}}
336 \DeclareOption{hebrew}{%
337   \input{rlbabel.def}%
338   \bbl@load@language{hebrew}}
339 \DeclareOption{hungarian}{\bbl@try@load@lang{}{magyar}{}}
340 \DeclareOption{lowersorbian}{\bbl@try@load@lang{}{lsorbian}{}}
341 \DeclareOption{nynorsk}{\bbl@try@load@lang{}{norsk}{}}
342 \DeclareOption{polutonikogreek}{%
343   \bbl@try@load@lang{}{greek}{\languageattribute{greek}{polutoniko}}}
344 \DeclareOption{portuguese}{\bbl@try@load@lang{}{portuges}{}}
345 \DeclareOption{russian}{\bbl@try@load@lang{}{russianb}{}}
346 \DeclareOption{ukrainian}{\bbl@try@load@lang{}{ukraineb}{}}
347 \DeclareOption{uppersorbian}{\bbl@try@load@lang{}{usorbian}{}}

```

Another way to extend the list of ‘known’ options for babel was to create the file `bblopts.cfg` in which one can add option declarations. However, this mechanism is deprecated – if you want an alternative name for a language, just create a new `.ldf` file loading the actual one. You can also set the name of the file with the package option `config=<name>`, which will load `<name>.cfg` instead.

```

348 \ifx\bbl@opt@config\@nnil
349   \@ifpackagewith{babel}{noconfigs}{}%
350   {\InputIfFileExists{bblopts.cfg}%
351     {\typeout{*****^^J%
352       * Local config file bblopts.cfg used^^J%
353       *}}%
354     {}}%
355 \else
356   \InputIfFileExists{\bbl@opt@config.cfg}%
357   {\typeout{*****^^J%
358     * Local config file \bbl@opt@config.cfg used^^J%
359     *}}%
360   {\bbl@error{%
361     Local config file '\bbl@opt@config.cfg' not found}}{%
362     Perhaps you misspelled it.}}%
363 \fi

```

Recognizing global options in packages not having a closed set of them is not trivial, as for them to be processed they must be defined explicitly. So, package options not yet taken

into account and stored in `bbl@language@opts` are assumed to be languages (note this list also contains the language given with `main`). If not declared above, the name of the option and the file are the same.

```

364 \bbl@for\bbl@tempa\bbl@language@opts{%
365   \bbl@ifunset{ds@\bbl@tempa}%
366   {\edef\bbl@tempb{%
367     \noexpand\DeclareOption
368     {\bbl@tempa}%
369     {\noexpand\bbl@load@language{\bbl@tempa}}}%
370   \bbl@tempb}%
371   \@empty}

```

Now, we make sure an option is explicitly declared for any language set as global option, by checking if an `ldf` exists. The previous step was, in fact, somewhat redundant, but that way we minimize accessing the file system just to see if the option could be a language.

```

372 \bbl@foreach\@classoptionslist{%
373   \bbl@ifunset{ds@#1}%
374   {\IfFileExists{#1.ldf}%
375    {\DeclareOption{#1}{\bbl@load@language{#1}}}%
376    {}}%
377   {}}

```

If a main language has been set, store it for the third pass.

```

378 \ifx\bbl@opt@main\@nnil\else
379   \expandafter
380   \let\expandafter\bbl@loadmain\csname ds@\bbl@opt@main\endcsname
381   \DeclareOption{\bbl@opt@main}{}
382 \fi

```

And we are done, because all options for this pass has been declared. Those already processed in the first pass are just ignored.

The options have to be processed in the order in which the user specified them (except, of course, global options, which \LaTeX processes before):

```

383 \def\AfterBabelLanguage#1{%
384   \bbl@ifsamestring\CurrentOption{#1}{\global\bbl@add\bbl@afterlang}{}}
385 \DeclareOption*{}
386 \ProcessOptions*

```

This finished the second pass. Now the third one begins, which loads the main language set with the key `main`. A warning is raised if the main language is not the same as the last named one, or if the value of the key `main` is not a language. Then execute directly the option (because it could be used only in `main`). After loading all languages, we deactivate `\AfterBabelLanguage`.

```

387 \ifx\bbl@opt@main\@nnil
388   \edef\bbl@tempa{\@classoptionslist,\bbl@language@opts}
389   \let\bbl@tempc\@empty
390   \bbl@for\bbl@tempb\bbl@tempa{%
391     \bbl@xin@{,\bbl@tempb,}{,\bbl@loaded,}%
392     \ifin@\edef\bbl@tempc{\bbl@tempb}\fi}
393   \def\bbl@tempa#1,#2\@nnil{\def\bbl@tempb{#1}}
394   \expandafter\bbl@tempa\bbl@loaded,\@nnil
395   \ifx\bbl@tempb\bbl@tempc\else
396     \bbl@warning{%
397       Last declared language option is '\bbl@tempc',\%
398       but the last processed one was '\bbl@tempb'.\%
399       The main language cannot be set as both a global\%
400       and a package option. Use 'main=\bbl@tempc' as\%
401       option. Reported}%

```



```

402 \fi
403 \else
404 \DeclareOption{\bbl@opt@main}{\bbl@loadmain}
405 \ExecuteOptions{\bbl@opt@main}
406 \DeclareOption*{}
407 \ProcessOptions*
408 \fi
409 \def\AfterBabelLanguage{%
410 \bbl@error
411 {Too late for \string\AfterBabelLanguage}%
412 {Languages have been loaded, so I can do nothing}}

```

In order to catch the case where the user forgot to specify a language we check whether `\bbl@main@language`, has become defined. If not, no language has been loaded and an error message is displayed.

```

413 \ifx\bbl@main@language\undefined
414 \bbl@error{%
415 You haven't specified a language option}{%
416 You need to specify a language, either as a global option\\%
417 or as an optional argument to the \string\usepackage\space
418 command;\\%
419 You shouldn't try to proceed from here, type x to quit.}
420 \fi
421 \end{package}

```

8 The kernel of Babel (`babel.def`, `common`)

The kernel of the babel system is stored in either `hyphen.cfg` or `switch.def` and `babel.def`. The file `babel.def` contains most of the code, while `switch.def` defines the language switching commands; both can be read at run time. The file `hyphen.cfg` is a file that can be loaded into the format, which is necessary when you want to be able to switch hyphenation patterns (by default, it also inputs `switch.def`, for “historical reasons”, but it is not necessary). When `babel.def` is loaded it checks if the current version of `switch.def` is in the format; if not it is loaded. A further file, `babel.sty`, contains \LaTeX -specific stuff. Because plain \TeX users might want to use some of the features of the babel system too, care has to be taken that plain \TeX can process the files. For this reason the current format will have to be checked in a number of places. Some of the code below is common to plain \TeX and \LaTeX , some of it is for the \LaTeX case only.

Plain formats based on `etex` (`etex`, `xetex`, `luatex`) don’t load `hyphen.cfg` but `etex.src`, which follows a different naming convention, so we need to define the babel names. It presumes `language.def` exists and it is the same file used when formats were created.

8.1 Tools

```

422 \core
423 \ifx\ldf@quit\undefined
424 \else
425 \expandafter\endinput
426 \fi
427 \langle\langle Make sure ProvidesFile is defined \rangle\rangle
428 \ProvidesFile{babel.def}[\langle\langle date \rangle\rangle] \langle\langle version \rangle\rangle Babel common definitions]
429 \langle\langle Load macros for plain if not LaTeX \rangle\rangle
430 \ifx\bbl@ifshorthand\undefined
431 \def\bbl@ifshorthand#1#2#3{#2}%
432 \def\bbl@opt@safe{BR}
433 \def\AfterBabelLanguage#1#2{}

```

```

434 \let\bbl@afterlang\relax
435 \let\bbl@language@opts\@empty
436 \fi
437 \input switch.def\relax
438 \ifx\bbl@languages\@undefined
439 \ifx\directlua\@undefined
440 \openin1 = language.def
441 \ifeof1
442 \closein1
443 \message{I couldn't find the file language.def}
444 \else
445 \closein1
446 \beginingroup
447 \def\addlanguage#1#2#3#4#5{%
448 \expandafter\ifx\csname lang@#1\endcsname\relax\else
449 \global\expandafter\let\csname l@#1\expandafter\endcsname
450 \csname lang@#1\endcsname
451 \fi}%
452 \def\uselanguage#1{}}%
453 \input language.def
454 \endgroup
455 \fi
456 \fi
457 \chardef\l@english\z@
458 \fi
459 <<Load patterns in luatex>>
460 <<Basic macros>>

```

\addto For each language four control sequences have to be defined that control the language-specific definitions. To be able to add something to these macro once they have been defined the macro \addto is introduced. It takes two arguments, a *<control sequence>* and T_EX-code to be added to the *<control sequence>*.

If the *<control sequence>* has not been defined before it is defined now. The control sequence could also expand to \relax, in which case a circular definition results. The net result is a stack overflow. Otherwise the replacement text for the *<control sequence>* is expanded and stored in a token register, together with the T_EX-code to be added. Finally the *<control sequence>* is redefined, using the contents of the token register.

```

461 \def\addto#1#2{%
462 \ifx#1\@undefined
463 \def#1{#2}%
464 \else
465 \ifx#1\relax
466 \def#1{#2}%
467 \else
468 {\toks@\expandafter{#1#2}%
469 \xdef#1{\the\toks@}}%
470 \fi
471 \fi}

```

The macro \initiate@active@char takes all the necessary actions to make its argument a shorthand character. The real work is performed once for each character.

```

472 \def\bbl@withactive#1#2{%
473 \beginingroup
474 \lccode`~=`#2\relax
475 \lowercase{\endgroup#1~}}

```

\bbl@redefine To redefine a command, we save the old meaning of the macro. Then we redefine it to call the original macro with the ‘sanitized’ argument. The reason why we do it this way is that

we don't want to redefine the \LaTeX macros completely in case their definitions change (they have changed in the past).

Because we need to redefine a number of commands we define the command `\bbl@redefine` which takes care of this. It creates a new control sequence, `\org@...`

```
476 \def\bbl@redefine#1{%
477   \edef\bbl@tempa{\bbl@stripslash#1}%
478   \expandafter\let\csname org@\bbl@tempa\endcsname#1%
479   \expandafter\def\csname\bbl@tempa\endcsname}
```

This command should only be used in the preamble of the document.

```
480 \@onlypreamble\bbl@redefine
```

`\bbl@redefine@long` This version of `\babel@redefine` can be used to redefine `\long` commands such as `\ifthenelse`.

```
481 \def\bbl@redefine@long#1{%
482   \edef\bbl@tempa{\bbl@stripslash#1}%
483   \expandafter\let\csname org@\bbl@tempa\endcsname#1%
484   \expandafter\long\expandafter\def\csname\bbl@tempa\endcsname}
485 \@onlypreamble\bbl@redefine@long
```

`\bbl@redefineroobust` For commands that are redefined, but which *might* be robust we need a slightly more intelligent macro. A robust command `foo` is defined to expand to `\protect\foo_`. So it is necessary to check whether `\foo_` exists. The result is that the command that is being redefined is always robust afterwards. Therefore all we need to do now is define `\foo_`.

```
486 \def\bbl@redefineroobust#1{%
487   \edef\bbl@tempa{\bbl@stripslash#1}%
488   \bbl@ifunset{\bbl@tempa\space}%
489   {\expandafter\let\csname org@\bbl@tempa\endcsname#1%
490    \bbl@exp{\def\#1{\protect\<\bbl@tempa\space>}}}%
491   {\bbl@exp{\let\<org@\bbl@tempa>\<\bbl@tempa\space>}}}%
492   \@namedef{\bbl@tempa\space}}
```

This command should only be used in the preamble of the document.

```
493 \@onlypreamble\bbl@redefineroobust
```

8.2 Hooks

Note they are loaded in `babel.def`. `switch.def` only provides a “hook” for hooks (with a default value which is a no-op, below). Admittedly, the current implementation is a somewhat simplistic and does very little to catch errors, but it is intended for developers, after all. `\bbl@usehooks` is the commands used by `babel` to execute hooks defined for an event.

```
494 \def\AddBabelHook#1#2{%
495   \bbl@ifunset{\bbl@hk@#1}{\EnableBabelHook{#1}}}%
496   \def\bbl@tempa##1,#2=##2,##3\@empty{\def\bbl@tempb{##2}}%
497   \expandafter\bbl@tempa\bbl@evargs,#2=,\@empty
498   \bbl@ifunset{\bbl@ev@#1@#2}%
499   {\bbl@csarg\bbl@add{ev@#2}{\bbl@elt{#1}}}%
500   \bbl@csarg\newcommand}%
501   {\bbl@csarg\let{ev@#1@#2}\relax
502   \bbl@csarg\newcommand}%
503   {ev@#1@#2}[\bbl@tempb]}
504 \def\EnableBabelHook#1{\bbl@csarg\let{hk@#1}\@firstofone}
505 \def\DisableBabelHook#1{\bbl@csarg\let{hk@#1}\@gobble}
506 \def\bbl@usehooks#1#2{%
507   \def\bbl@elt##1{%
```

```

508 \@nameuse{bbl@hk@##1}\@nameuse{bbl@ev@##1@#1}#2}}%
509 \@nameuse{bbl@ev@#1}}

```

To ensure forward compatibility, arguments in hooks are set implicitly. So, if a further argument is added in the future, there is no need to change the existing code. Note events intended for hyphen.cfg are also loaded (just in case you need them for some reason).

```

510 \def\bbl@evargs{% don't delete the comma
511   everylanguage=1,loadkernel=1,loadpatterns=1,loadexceptions=1,%
512   adddialect=2,patterns=2,defaultcommands=0,encodedcommands=2,write=0,%
513   beforeextras=0,afterextras=0,stopcommands=0,stringprocess=0,%
514   hyphenation=2,initiateactive=3,afterreset=0,foreign=0,foreign*=0}

```

`\babelensure` The user command just parses the optional argument and creates a new macro named `\bbl@e@<language>`. We register a hook at the `afterextras` event which just executes this macro in a “complete” selection (which, if undefined, is `\relax` and does nothing). This part is somewhat involved because we have to make sure things are expanded the correct number of times.

The macro `\bbl@e@<language>` contains `\bbl@ensure{<include>}{<exclude>}{<fontenc>}`, which in turn loops over the macros names in `\bbl@captionslist`, excluding (with the help of `\in@`) those in the exclude list. If the fontenc is given (and not `\relax`), the `\fontencoding` is also added. Then we loop over the include list, but if the macro already contains `\foreignlanguage`, nothing is done. Note this macro (1) is not restricted to the preamble, and (2) changes are local.

```

515 \newcommand\babelensure[2][{}% TODO - revise test files
516 \AddBabelHook{babel-ensure}{afterextras}{%
517   \ifcase\bbl@select@type
518     \@nameuse{bbl@e@\languagename}%
519   \fi}%
520 \begingroup
521 \let\bbl@ens@include\empty
522 \let\bbl@ens@exclude\empty
523 \def\bbl@ens@fontenc{\relax}%
524 \def\bbl@tempb##1{%
525   \ifx\@empty##1\else\noexpand##1\expandafter\bbl@tempb\fi}%
526 \edef\bbl@tempa{\bbl@tempb#1\@empty}%
527 \def\bbl@tempb##1=##2\@{\@namedef{bbl@ens@##1}{##2}}%
528 \bbl@foreach\bbl@tempa{\bbl@tempb##1\@}%
529 \def\bbl@tempc{\bbl@ensure}%
530 \expandafter\bbl@add\expandafter\bbl@tempc\expandafter{%
531   \expandafter{\bbl@ens@include}}%
532 \expandafter\bbl@add\expandafter\bbl@tempc\expandafter{%
533   \expandafter{\bbl@ens@exclude}}%
534 \toks@\expandafter{\bbl@tempc}%
535 \bbl@exp{%
536 \endgroup
537 \def\<bbl@e@#2>{\the\toks@{\bbl@ens@fontenc}}}%
538 \def\bbl@ensure#1#2#3{% 1: include 2: exclude 3: fontenc
539 \def\bbl@tempb##1{% elt for (excluding) \bbl@captionslist list
540   \ifx##1\@empty\else
541     \in@{##1}{#2}%
542     \ifin@
543       \bbl@ifunset{bbl@ensure@\languagename}%
544       {\bbl@exp{%
545         \\DeclareRobustCommand\<bbl@ensure@\languagename>[1]{%
546           \\foreignlanguage{\languagename}%
547           {\ifx\relax#3\else
548             \\fontencoding{#3}\\selectfont
549           \fi

```

```

550          #####1}}}%
551      {}%
552      \toks@\expandafter{##1}%
553      \edef##1{%
554          \bbl@csarg\noexpand{ensure@\language}%
555          {\the\toks@}}%
556      \fi
557      \expandafter\bbl@tempb
558      \fi}%
559      \expandafter\bbl@tempb\bbl@captionslist\today\@empty
560      \def\bbl@tempa##1{% elt for include list
561          \ifx##1\@empty\else
562              \bbl@csarg\in@{ensure@\language\expandafter}\expandafter{##1}%
563              \ifin\else
564                  \bbl@tempb##1\@empty
565              \fi
566              \expandafter\bbl@tempa
567          \fi}%
568      \bbl@tempa#1\@empty}
569      \def\bbl@captionslist{%
570          \prefacename\refname\abstractname\bibname\chaptername\appendixname
571          \contentsname\listfigurename\listtablename\indexname\figurename
572          \tablename\partname\enclname\ccname\headtoname\pagename\seename
573          \alsoname\proofname\glossaryname}

```

8.3 Setting up language files

`\LdfInit` The second version of `\LdfInit` macro takes two arguments. The first argument is the name of the language that will be defined in the language definition file; the second argument is either a control sequence or a string from which a control sequence should be constructed. The existence of the control sequence indicates that the file has been processed before.

At the start of processing a language definition file we always check the category code of the `@`-sign. We make sure that it is a ‘letter’ during the processing of the file. We also save its name as the last called option, even if not loaded.

Another character that needs to have the correct category code during processing of language definition files is the equals sign, ‘=’, because it is sometimes used in constructions with the `\let` primitive. Therefore we store its current catcode and restore it later on.

Now we check whether we should perhaps stop the processing of this file. To do this we first need to check whether the second argument that is passed to `\LdfInit` is a control sequence. We do that by looking at the first token after passing #2 through string. When it is equal to `\@backslashchar` we are dealing with a control sequence which we can compare with `\@undefined`.

If so, we call `\ldf@quit` to set the main language, restore the category code of the `@`-sign and call `\endinput`

When #2 was *not* a control sequence we construct one and compare it with `\relax`.

Finally we check `\originalTeX`.

```

574 \def\bbl@ldfinit{%
575     \let\bbl@screset\@empty
576     \let\BabelStrings\bbl@opt@string
577     \let\BabelOptions\@empty
578     \let\BabelLanguages\relax
579     \ifx\originalTeX\@undefined
580         \let\originalTeX\@empty
581     \else
582         \originalTeX

```

```

583 \fi}
584 \def\LdfInit#1#2{%
585 \chardef\atcatcode=\catcode`\@
586 \catcode`\@=11\relax
587 \chardef\eqcatcode=\catcode`\=
588 \catcode`\==12\relax
589 \expandafter\if\expandafter\@backslashchar
590 \expandafter\@car\string#2\@nil
591 \ifx#2\undefined\else
592 \ldf@quit{#1}%
593 \fi
594 \else
595 \expandafter\ifx\csname#2\endcsname\relax\else
596 \ldf@quit{#1}%
597 \fi
598 \fi
599 \bbl@ldfinit}

```

`\ldf@quit` This macro interrupts the processing of a language definition file.

```

600 \def\ldf@quit#1{%
601 \expandafter\main@language\expandafter{#1}%
602 \catcode`\@=\atcatcode \let\atcatcode\relax
603 \catcode`\==\eqcatcode \let\eqcatcode\relax
604 \endinput}

```

`\ldf@finish` This macro takes one argument. It is the name of the language that was defined in the language definition file.

We load the local configuration file if one is present, we set the main language (taking into account that the argument might be a control sequence that needs to be expanded) and reset the category code of the @-sign.

```

605 \def\bbl@afterldf#1{%
606 \bbl@afterlang
607 \let\bbl@afterlang\relax
608 \let\BabelModifiers\relax
609 \let\bbl@screset\relax}%
610 \def\ldf@finish#1{%
611 \loadlocalcfg{#1}%
612 \bbl@afterldf{#1}%
613 \expandafter\main@language\expandafter{#1}%
614 \catcode`\@=\atcatcode \let\atcatcode\relax
615 \catcode`\==\eqcatcode \let\eqcatcode\relax}

```

After the preamble of the document the commands `\LdfInit`, `\ldf@quit` and `\ldf@finish` are no longer needed. Therefore they are turned into warning messages in \LaTeX .

```

616 \@onlypreamble\LdfInit
617 \@onlypreamble\ldf@quit
618 \@onlypreamble\ldf@finish

```

`\main@language` This command should be used in the various language definition files. It stores its argument in `\bbl@main@language`; to be used to switch to the correct language at the beginning of the document.

```

619 \def\main@language#1{%
620 \def\bbl@main@language{#1}%
621 \let\languagename\bbl@main@language
622 \bbl@patterns{\languagename}}

```

We also have to make sure that some code gets executed at the beginning of the document.

```
623 \AtBeginDocument{%
624   \expandafter\selectlanguage\expandafter{\bbl@main@language}}
```

A bit of optimization. Select in heads/foots the language only if necessary.

```
625 \def\select@language@x#1{%
626   \ifcase\bbl@select@type
627     \bbl@ifsamestring\language\language{#1}{\select@language{#1}}%
628   \else
629     \select@language{#1}%
630   \fi}
```

8.4 Shorthands

`\bbl@add@special` The macro `\bbl@add@special` is used to add a new character (or single character control sequence) to the macro `\dospecials` (and `\@sanitize` if \LaTeX is used). It is used only at one place, namely when `\initiate@active@char` is called (which is ignored if the char has been made active before). Because `\@sanitize` can be undefined, we put the definition inside a conditional.

Items are added to the lists without checking its existence or the original catcode. It does not hurt, but should be fixed. It's already done with `\nfss@catcodes`, added in 3.10.

```
631 \def\bbl@add@special#1{% 1:a macro like \", \?, etc.
632   \bbl@add\dospecials{\do#1}% test @sanitize = \relax, for back. compat.
633   \bbl@ifunset{@sanitize}{\bbl@add@sanitize{\@makeother#1}}%
634   \ifx\nfss@catcodes\@undefined\else % TODO - same for above
635     \begingroup
636       \catcode`#1\active
637       \nfss@catcodes
638       \ifnum\catcode`#1=\active
639         \endgroup
640         \bbl@add\nfss@catcodes{\@makeother#1}%
641       \else
642         \endgroup
643       \fi
644   \fi}
```

`\bbl@remove@special` The companion of the former macro is `\bbl@remove@special`. It removes a character from the set macros `\dospecials` and `\@sanitize`, but it is not used at all in the babel core.

```
645 \def\bbl@remove@special#1{%
646   \begingroup
647     \def\x##1##2{\ifnum`#1=`##2\noexpand\@empty
648       \else\noexpand##1\noexpand##2\fi}%
649     \def\do{\x\do}%
650     \def\@makeother{\x\@makeother}%
651   \edef\x{\endgroup
652     \def\noexpand\dospecials{\dospecials}%
653     \expandafter\ifx\curname @sanitize\endcurname\relax\else
654       \def\noexpand\@sanitize{\@sanitize}%
655     \fi}%
656   \x}
```

`\initiate@active@char` A language definition file can call this macro to make a character active. This macro takes one argument, the character that is to be made active. When the character was already active this macro does nothing. Otherwise, this macro defines the control sequence `\normal@char⟨char⟩` to expand to the character in its ‘normal state’ and it defines the active character to expand to `\normal@char⟨char⟩` by default (`⟨char⟩` being the character

to be made active). Later its definition can be changed to expand to `\active@char⟨char⟩` by calling `\bbl@activate{⟨char⟩}`.

For example, to make the double quote character active one could have `\initiate@active@char{"}` in a language definition file. This defines " as `\active@prefix "\active@char"` (where the first " is the character with its original catcode, when the shorthand is created, and `\active@char` is a single token). In protected contexts, it expands to `\protect "` or `\noexpand "` (ie, with the original "); otherwise `\active@char` is executed. This macro in turn expands to `\normal@char` in "safe" contexts (eg, `\label`), but `\user@active` in normal "unsafe" ones. The latter search a definition in the user, language and system levels, in this order, but if none is found, `\normal@char` is used. However, a deactivated shorthand (with `\bbl@deactivate` is defined as `\active@prefix "\normal@char`".

The following macro is used to define shorthands in the three levels. It takes 4 arguments: the (string'ed) character, `\<level>@group`, `<level>@active` and `<next-level>@active` (except in system).

```
657 \def\bbl@active@def#1#2#3#4{%
658   \@namedef{#3#1}{%
659     \expandafter\ifx\csname#2@sh@#1@endcsname\relax
660       \bbl@afterelse\bbl@sh@select#2#1{#3@arg#1}{#4#1}%
661     \else
662       \bbl@afterfi\csname#2@sh@#1@endcsname
663     \fi}%
```

When there is also no current-level shorthand with an argument we will check whether there is a next-level defined shorthand for this active character.

```
664   \long\@namedef{#3@arg#1}##1{%
665     \expandafter\ifx\csname#2@sh@#1\string##1@endcsname\relax
666       \bbl@afterelse\csname#4#1@endcsname##1%
667     \else
668       \bbl@afterfi\csname#2@sh@#1\string##1@endcsname
669     \fi}}%
```

`\initiate@active@char` calls `\@initiate@active@char` with 3 arguments. All of them are the same character with different catcodes: active, other (`\string'ed`) and the original one. This trick simplifies the code a lot.

```
670 \def\@initiate@active@char#1#2#3{%
671   \bbl@ifunset{active@char\string#1}%
672   {\bbl@withactive
673     {\expandafter\@initiate@active@char\expandafter}#1\string#1#1}%
674   {}}
```

The very first thing to do is saving the original catcode and the original definition, even if not active, which is possible (undefined characters require a special treatment to avoid making them `\relax`).

```
675 \def\@initiate@active@char#1#2#3{%
676   \bbl@csarg\edef{oricat@#2}{\catcode`#2=\the\catcode`#2\relax}%
677   \ifx#1\@undefined
678     \bbl@csarg\edef{oridef@#2}{\let\noexpand#1\noexpand\@undefined}%
679   \else
680     \bbl@csarg\let{oridef@#2}#1%
681     \bbl@csarg\edef{oridef@#2}{%
682       \let\noexpand#1%
683       \expandafter\noexpand\csname bbl@oridef@@#2@endcsname}%
684   \fi
```

If the character is already active we provide the default expansion under this shorthand mechanism. Otherwise we write a message in the transcript file, and define

`\normal@char⟨char⟩` to expand to the character in its default state. If the character is mathematically active when babel is loaded (for example ') the normal expansion is somewhat different to avoid an infinite loop (but it does not prevent the loop if the `mathcode` is set to "8000 *a posteriori*").

```

685 \ifx#1#3\relax
686   \expandafter\let\csname normal@char#2\endcsname#3%
687 \else
688   \bbl@info{Making #2 an active character}%
689   \ifnum\mathcode`#2="8000
690     \@namedef{normal@char#2}{%
691       \textormath{#3}{\csname bbl@oridef@#2\endcsname}}%
692   \else
693     \@namedef{normal@char#2}{#3}%
694   \fi

```

To prevent problems with the loading of other packages after babel we reset the catcode of the character to the original one at the end of the package and of each language file (except with `KeepShorthandsActive`). It is re-activate again at `\begin{document}`. We also need to make sure that the shorthands are active during the processing of the .aux file. Otherwise some citations may give unexpected results in the printout when a shorthand was used in the optional argument of `\bibitem` for example. Then we make it active (not strictly necessary, but done for backward compatibility).

```

695   \bbl@restoreactive{#2}%
696   \AtBeginDocument{%
697     \catcode`#2\active
698     \if@filesw
699       \immediate\write\@mainaux{\catcode`\string#2\active}%
700     \fi}%
701   \expandafter\bbl@add@special\csname#2\endcsname
702   \catcode`#2\active
703 \fi

```

Now we have set `\normal@char⟨char⟩`, we must define `\active@char⟨char⟩`, to be executed when the character is activated. We define the first level expansion of `\active@char⟨char⟩` to check the status of the `@safe@actives` flag. If it is set to true we expand to the 'normal' version of this character, otherwise we call `\user@active⟨char⟩` to start the search of a definition in the user, language and system levels (or eventually `normal@char⟨char⟩`).

```

704 \let\bbl@tempa\@firstoftwo
705 \if\string^#2%
706   \def\bbl@tempa{\noexpand\textormath}%
707 \else
708   \ifx\bbl@mathnormal\@undefined\else
709     \let\bbl@tempa\bbl@mathnormal
710   \fi
711 \fi
712 \expandafter\edef\csname active@char#2\endcsname{%
713   \bbl@tempa
714     {\noexpand\if@safe@actives
715       \noexpand\expandafter
716       \expandafter\noexpand\csname normal@char#2\endcsname
717     \noexpand\else
718       \noexpand\expandafter
719       \expandafter\noexpand\csname bbl@doactive#2\endcsname
720     \noexpand\fi}%
721   {\expandafter\noexpand\csname normal@char#2\endcsname}}%
722 \bbl@csarg\edef{doactive#2}{%
723   \expandafter\noexpand\csname user@active#2\endcsname}%

```

We now define the default values which the shorthand is set to when activated or deactivated. It is set to the deactivated form (globally), so that the character expands to

`\active@prefix <char> \normal@char <char>`

(where `\active@char <char>` is *one* control sequence!).

```
724 \bbl@csarg\edef{active@#2}{%
725   \noexpand\active@prefix\noexpand#1%
726   \expandafter\noexpand\csname active@char#2\endcsname}%
727 \bbl@csarg\edef{normal@#2}{%
728   \noexpand\active@prefix\noexpand#1%
729   \expandafter\noexpand\csname normal@char#2\endcsname}%
730 \expandafter\let\expandafter#1\csname bbl@normal@#2\endcsname
```

The next level of the code checks whether a user has defined a shorthand for himself with this character. First we check for a single character shorthand. If that doesn't exist we check for a shorthand with an argument.

```
731 \bbl@active@def#2\user@group{user@active}{language@active}%
732 \bbl@active@def#2\language@group{language@active}{system@active}%
733 \bbl@active@def#2\system@group{system@active}{normal@char}%
```

In order to do the right thing when a shorthand with an argument is used by itself at the end of the line we provide a definition for the case of an empty argument. For that case we let the shorthand character expand to its non-active self. Also, When a shorthand combination such as `'` ends up in a heading \TeX would see `\protect'\protect'`. To prevent this from happening a couple of shorthand needs to be defined at user level.

```
734 \expandafter\edef\csname\user@group @sh@#2@@\endcsname
735   {\expandafter\noexpand\csname normal@char#2\endcsname}%
736 \expandafter\edef\csname\user@group @sh@#2@\string\protect@\endcsname
737   {\expandafter\noexpand\csname user@active#2\endcsname}%
```

Finally, a couple of special cases are taken care of. (1) If we are making the right quote (`'`) active we need to change `\pr@m@s` as well. Also, make sure that a single `'` in math mode 'does the right thing'. (2) If we are using the caret (`^`) as a shorthand character special care should be taken to make sure math still works. Therefore an extra level of expansion is introduced with a check for math mode on the upper level.

```
738 \if\string'#2%
739   \let\prim@s\bbl@prim@s
740   \let\active@math@prime#1%
741 \fi
742 \bbl@usehooks{initiateactive}{{#1}{#2}{#3}}}
```

The following package options control the behaviour of shorthands in math mode.

```
743 <<(*More package options)>> ≡
744 \DeclareOption{math=active}{}
745 \DeclareOption{math=normal}{{\def\bbl@mathnormal{\noexpand\textormath}}}
746 <</More package options>>
```

Initiating a shorthand makes active the char. That is not strictly necessary but it is still done for backward compatibility. So we need to restore the original catcode at the end of package *and* and the end of the *ldf*.

```
747 \@ifpackagewith{babel}{KeepShorthandsActive}%
748   {\let\bbl@restoreactive\gobble}%
749   {\def\bbl@restoreactive#1{%
750     \bbl@exp{%
751       \\\AfterBabelLanguage\\CurrentOption
752       {\catcode`#1=\the\catcode`#1\relax}%
753     \\\AtEndOfPackage
```

```

754      {\catcode`#1=\the\catcode`#1\relax}}}%
755      \AtEndOfPackage{\let\bbl@restoreactive\@gobble}}

```

`\bbl@sh@select` This command helps the shorthand supporting macros to select how to proceed. Note that this macro needs to be expandable as do all the shorthand macros in order for them to work in expansion-only environments such as the argument of `\hyphenation`. This macro expects the name of a group of shorthands in its first argument and a shorthand character in its second argument. It will expand to either `\bbl@firstcs` or `\bbl@scndcs`. Hence two more arguments need to follow it.

```

756 \def\bbl@sh@select#1#2{%
757   \expandafter\ifx\csname#1@sh@#2@sel\endcsname\relax
758     \bbl@afterelse\bbl@scndcs
759   \else
760     \bbl@afterfi\csname#1@sh@#2@sel\endcsname
761   \fi}

```

`\active@prefix` The command `\active@prefix` which is used in the expansion of active characters has a function similar to `\OT1-cmd` in that it `\protects` the active character whenever `\protect` is *not* `\@typeset@protect`.

```

762 \def\active@prefix#1{%
763   \ifx\protect\@typeset@protect
764     \else

```

When `\protect` is set to `\@unexpandable@protect` we make sure that the active character is also *not* expanded by inserting `\noexpand` in front of it. The `\@gobble` is needed to remove a token such as `\activechar:` (when the double colon was the active character to be dealt with).

```

765     \ifx\protect\@unexpandable@protect
766       \noexpand#1%
767     \else
768       \protect#1%
769     \fi
770   \expandafter\@gobble
771   \fi}

```

`\if@safe@actives` In some circumstances it is necessary to be able to change the expansion of an active character on the fly. For this purpose the switch `@safe@actives` is available. The setting of this switch should be checked in the first level expansion of `\active@char⟨char⟩`.

```

772 \newif\if@safe@actives
773 \@safe@activesfalse

```

`\bbl@restore@actives` When the output routine kicks in while the active characters were made “safe” this must be undone in the headers to prevent unexpected typeset results. For this situation we define a command to make them “unsafe” again.

```

774 \def\bbl@restore@actives{\if@safe@actives\@safe@activesfalse\fi}

```

`\bbl@activate` Both macros take one argument, like `\initiate@active@char`. The macro is used to change the definition of an active character to expand to `\active@char⟨char⟩` in the case of `\bbl@activate`, or `\normal@char⟨char⟩` in the case of `\bbl@deactivate`.

```

775 \def\bbl@activate#1{%
776   \bbl@withactive{\expandafter\let\expandafter}#1%
777   \csname bbl@active@\string#1\endcsname}
778 \def\bbl@deactivate#1{%
779   \bbl@withactive{\expandafter\let\expandafter}#1%
780   \csname bbl@normal@\string#1\endcsname}

```

<code>\bbl@firstcs</code> <code>\bbl@scndcs</code>	<p>These macros have two arguments. They use one of their arguments to build a control sequence from.</p> <pre> 781 \def\bbl@firstcs#1#2{\csname#1\endcsname} 782 \def\bbl@scndcs#1#2{\csname#2\endcsname} </pre>
<code>\declare@shorthand</code>	<p>The command <code>\declare@shorthand</code> is used to declare a shorthand on a certain level. It takes three arguments:</p> <ol style="list-style-type: none"> 1. a name for the collection of shorthands, i.e. ‘system’, or ‘dutch’; 2. the character (sequence) that makes up the shorthand, i.e. <code>~</code> or <code>"a</code>; 3. the code to be executed when the shorthand is encountered. <pre> 783 \def\declare@shorthand#1#2{\@decl@short{#1}#2\@nil} 784 \def\@decl@short#1#2#3\@nil#4{% 785 \def\bbl@tempa{#3}% 786 \ifx\bbl@tempa\@empty 787 \expandafter\let\csname #1@sh@\string#2@sel\endcsname\bbl@scndcs 788 \bbl@ifunset{#1@sh@\string#2@}\{}% 789 {\def\bbl@tempa{#4}% 790 \expandafter\ifx\csname#1@sh@\string#2@\endcsname\bbl@tempa 791 \else 792 \bbl@info 793 {Redefining #1 shorthand \string#2\\ 794 in language \CurrentOption}% 795 \fi}% 796 \@namedef{#1@sh@\string#2@}{#4}% 797 \else 798 \expandafter\let\csname #1@sh@\string#2@sel\endcsname\bbl@firstcs 799 \bbl@ifunset{#1@sh@\string#2@\string#3@}\{}% 800 {\def\bbl@tempa{#4}% 801 \expandafter\ifx\csname#1@sh@\string#2@\string#3@\endcsname\bbl@tempa 802 \else 803 \bbl@info 804 {Redefining #1 shorthand \string#2\string#3\\ 805 in language \CurrentOption}% 806 \fi}% 807 \@namedef{#1@sh@\string#2@\string#3@}{#4}% 808 \fi} </pre>
<code>\textormath</code>	<p>Some of the shorthands that will be declared by the language definition files have to be usable in both text and mathmode. To achieve this the helper macro <code>\textormath</code> is provided.</p> <pre> 809 \def\textormath{% 810 \ifmmode 811 \expandafter\@secondoftwo 812 \else 813 \expandafter\@firstoftwo 814 \fi} </pre>
<code>\user@group</code> <code>\language@group</code> <code>\system@group</code>	<p>The current concept of ‘shorthands’ supports three levels or groups of shorthands. For each level the name of the level or group is stored in a macro. The default is to have a user group; use language group ‘english’ and have a system group called ‘system’.</p> <pre> 815 \def\user@group{user} 816 \def\language@group{english} 817 \def\system@group{system} </pre>

`\useshortands` This is the user level command to tell \LaTeX that user level shorthands will be used in the document. It takes one argument, the character that starts a shorthand. First note that this is user level, and then initialize and activate the character for use as a shorthand character (ie, it's active in the preamble). Languages can deactivate shorthands, so a starred version is also provided which activates them always after the language has been switched.

```

818 \def\useshortands{%
819   \@ifstar\bb1@usesesh@s{\bb1@usesesh@x{}}
820 \def\bb1@usesesh@s#1{%
821   \bb1@usesesh@x
822   {\AddBabelHook{babel-sh-\string#1}{afterextras}{\bb1@activate{#1}}}%
823   {#1}}
824 \def\bb1@usesesh@x#1#2{%
825   \bb1@ifshorthand{#2}%
826   {\def\user@group{user}%
827     \initiate@active@char{#2}%
828     #1%
829     \bb1@activate{#2}}%
830   {\bb1@error
831     {Cannot declare a shorthand turned off (\string#2)}
832     {Sorry, but you cannot use shorthands which have been\\%
833       turned off in the package options}}}

```

`\defineshorthand` Currently we only support two groups of user level shorthands, named internally `user` and `user@<lang>` (language-dependent user shorthands). By default, only the first one is taken into account, but if the former is also used (in the optional argument of `\defineshorthand`) a new level is inserted for it (`user@generic`, done by `\bb1@set@user@generic`); we make also sure `{}` and `\protect` are taken into account in this new top level.

```

834 \def\user@language@group{user@\language@group}
835 \def\bb1@set@user@generic#1#2{%
836   \bb1@ifunset{user@generic@active#1}%
837   {\bb1@active@def#1\user@language@group{user@active}{user@generic@active}%
838     \bb1@active@def#1\user@group{user@generic@active}{language@active}%
839     \expandafter\edef\csname#2@sh@#1@@\endcsname{%
840       \expandafter\noexpand\csname normal@char#1\endcsname}%
841     \expandafter\edef\csname#2@sh@#1@\string\protect\endcsname{%
842       \expandafter\noexpand\csname user@active#1\endcsname}}%
843   \@empty}
844 \newcommand\defineshorthand[3][user]{%
845   \edef\bb1@tempa{\zap@space#1 \@empty}%
846   \bb1@for\bb1@tempb\bb1@tempa{%
847     \if*\expandafter\@car\bb1@tempb\@nil
848       \edef\bb1@tempb{user\expandafter\@gobble\bb1@tempb}%
849       \@expandtwoargs
850       \bb1@set@user@generic{\expandafter\string\@car#2\@nil}\bb1@tempb
851     \fi
852     \declare@shorthand{\bb1@tempb}{#2}{#3}}}

```

`\languageshortands` A user level command to change the language from which shorthands are used. Unfortunately, babel currently does not keep track of defined groups, and therefore there is no way to catch a possible change in casing.

```

853 \def\languageshortands#1{\def\language@group{#1}}

```

`\aliasshorthand` First the new shorthand needs to be initialized,

```

854 \def\aliasshorthand#1#2{%
855   \bb1@ifshorthand{#2}%
856   {\expandafter\ifx\csname active@char\string#2\endcsname\relax
857     \ifx\document\@notprerr

```

```

858      \@notshorthand{#2}%
859      \else
860      \initiate@active@char{#2}%

Then, we define the new shorthand in terms of the original one, but note with
\aliasshorthands{"}{/} is \active@prefix / \active@char/, so we still need to let the
latest to \active@char".

861      \expandafter\let\csname active@char\string#2\expandafter\endcsname
862      \csname active@char\string#1\endcsname
863      \expandafter\let\csname normal@char\string#2\expandafter\endcsname
864      \csname normal@char\string#1\endcsname
865      \bbl@activate{#2}%
866      \fi
867      \fi}%
868      {\bbl@error
869      {Cannot declare a shorthand turned off (\string#2)}
870      {Sorry, but you cannot use shorthands which have been\\%
871      turned off in the package options}}}

```

\@notshorthand

```

872 \def\@notshorthand#1{%
873   \bbl@error{%
874     The character '\string #1' should be made a shorthand character;\\%
875     add the command \string\usesshorthands\string{#1\string} to
876     the preamble.\\%
877     I will ignore your instruction}%
878   {You may proceed, but expect unexpected results}}

```

\shorthandon The first level definition of these macros just passes the argument on to \bbl@switch@sh, adding \@nil at the end to denote the end of the list of characters.

```

879 \newcommand*\shorthandon[1]{\bbl@switch@sh\@ne#1\@nnil}
880 \DeclareRobustCommand*\shorthandoff{%
881   \@ifstar{\bbl@shorthandoff\tw@}{\bbl@shorthandoff\z@}}
882 \def\bbl@shorthandoff#1#2{\bbl@switch@sh#1#2\@nnil}

```

\bbl@switch@sh The macro \bbl@switch@sh takes the list of characters apart one by one and subsequently switches the category code of the shorthand character according to the first argument of \bbl@switch@sh.

But before any of this switching takes place we make sure that the character we are dealing with is known as a shorthand character. If it is, a macro such as \active@char" should exist.

Switching off and on is easy – we just set the category code to ‘other’ (12) and \active. With the starred version, the original catcode and the original definition, saved in @initiate@active@char, are restored.

```

883 \def\bbl@switch@sh#1#2{%
884   \ifx#2\@nnil\else
885     \bbl@ifunset{\bbl@active@\string#2}%
886     {\bbl@error
887       {I cannot switch '\string#2' on or off--not a shorthand}%
888       {This character is not a shorthand. Maybe you made\\%
889       a typing mistake? I will ignore your instruction}}%
890     {\ifcase#1%
891       \catcode`#2\relax
892       \or
893       \catcode`#2\active
894       \or
895       \csname bbl@oricat@\string#2\endcsname

```

```

896         \csname bbl@oridef@\string#2\endcsname
897     \fi}%
898     \bbl@afterfi\bbl@switch@sh#1%
899 \fi}

```

Note the value is that at the expansion time, eg, in the preamble shorthands are usually deactivated.

```

900 \def\babelshorthand{\active@prefix\babelshorthand\bbl@putsh}
901 \def\bbl@putsh#1{%
902     \bbl@ifunset{\bbl@active@\string#1}%
903     {\bbl@putsh@i#1\@empty\@nnil}%
904     {\csname bbl@active@\string#1\endcsname}}
905 \def\bbl@putsh@i#1#2\@nnil{%
906     \csname\language\sh@\string#1@%
907     \ifx\@empty#2\else\string#2@\fi\endcsname}
908 \ifx\bbl@opt@shorthands\@nnil\else
909     \let\bbl@s@initiate@active@char\initiate@active@char
910     \def\initiate@active@char#1{%
911         \bbl@ifshorthand{#1}{\bbl@s@initiate@active@char{#1}}{}}
912     \let\bbl@s@switch@sh\bbl@switch@sh
913     \def\bbl@switch@sh#1#2{%
914         \ifx#2\@nnil\else
915             \bbl@afterfi
916             \bbl@ifshorthand{#2}{\bbl@s@switch@sh#1{#2}}{\bbl@switch@sh#1}%
917         \fi}
918     \let\bbl@s@activate\bbl@activate
919     \def\bbl@activate#1{%
920         \bbl@ifshorthand{#1}{\bbl@s@activate{#1}}{}}
921     \let\bbl@s@deactivate\bbl@deactivate
922     \def\bbl@deactivate#1{%
923         \bbl@ifshorthand{#1}{\bbl@s@deactivate{#1}}{}}
924 \fi

```

`\bbl@prim@s` One of the internal macros that are involved in substituting `\prime` for each right quote in
`\bbl@pr@m@s` mathmode is `\prim@s`. This checks if the next character is a right quote. When the right
quote is active, the definition of this macro needs to be adapted to look also for an active
right quote; the hat could be active, too.

```

925 \def\bbl@prim@s{%
926     \prime\futurelet\@let@token\bbl@pr@m@s}
927 \def\bbl@if@primes#1#2{%
928     \ifx#1\@let@token
929         \expandafter\@firstoftwo
930     \else\ifx#2\@let@token
931         \bbl@afterelse\expandafter\@firstoftwo
932     \else
933         \bbl@afterfi\expandafter\@secondoftwo
934     \fi\fi}
935 \begingroup
936     \catcode`\^=7 \catcode`\*= \active \lccode`\*=`^
937     \catcode`\'=12 \catcode`\`= \active \lccode`\`= ` '
938     \lowercase{%
939         \gdef\bbl@pr@m@s{%
940             \bbl@if@primes" "%
941             \pr@@@s
942             {\bbl@if@primes*^\pr@@@t\egroup}}
943 \endgroup

```

Usually the `~` is active and expands to `\penalty\@M__`. When it is written to the `.aux` file it is written expanded. To prevent that and to be able to use the character `~` as a start

character for a shorthand, it is redefined here as a one character shorthand on system level. The system declaration is in most cases redundant (when ~ is still a non-break space), and in some cases is inconvenient (if ~ has been redefined); however, for backward compatibility it is maintained (some existing documents may rely on the babel value).

```
944 \initiate@active@char{~}
945 \declare@shorthand{system}{~}{\leavevmode\nobreak\ }
946 \bbl@activate{~}
```

\OT1dqpos The position of the double quote character is different for the OT1 and T1 encodings. It will later be selected using the \f@encoding macro. Therefore we define two macros here to store the position of the character in these encodings.

```
947 \expandafter\def\csname OT1dqpos\endcsname{127}
948 \expandafter\def\csname T1dqpos\endcsname{4}
```

When the macro \f@encoding is undefined (as it is in plain T_EX) we define it here to expand to OT1

```
949 \ifx\f@encoding\undefined
950   \def\f@encoding{OT1}
951 \fi
```

8.5 Language attributes

Language attributes provide a means to give the user control over which features of the language definition files he wants to enable.

\languageattribute The macro \languageattribute checks whether its arguments are valid and then activates the selected language attribute. First check whether the language is known, and then process each attribute in the list.

```
952 \newcommand\languageattribute[2]{%
953   \def\bbl@tempc{#1}%
954   \bbl@fixname\bbl@tempc
955   \bbl@iflanguage\bbl@tempc{%
956     \bbl@vforeach{#2}{%
```

We want to make sure that each attribute is selected only once; therefore we store the already selected attributes in \bbl@known@attribs. When that control sequence is not yet defined this attribute is certainly not selected before.

```
957     \ifx\bbl@known@attribs\undefined
958       \in@false
959     \else
```

Now we need to see if the attribute occurs in the list of already selected attributes.

```
960       \bbl@xin@{,\bbl@tempc-##1,}{,\bbl@known@attribs,}%
961     \fi
```

When the attribute was in the list we issue a warning; this might not be the users intention.

```
962     \ifin@
963       \bbl@warning{%
964         You have more than once selected the attribute '##1'\%
965         for language #1}%
966     \else
```

When we end up here the attribute is not selected before. So, we add it to the list of selected attributes and execute the associated T_EX-code.

```
967       \bbl@exp{%
968         \\bbl@add@list\\bbl@known@attribs{\bbl@tempc-##1}}%
969       \edef\bbl@tempa{\bbl@tempc-##1}%

```



```

970     \expandafter\bb1@ifknown@ttrib\expandafter{\bb1@tempa}\bb1@attributes%
971     {\csname\bb1@tempc @attr##1\endcsname}%
972     {\@attrerr{\bb1@tempc}{##1}}%
973     \fi}}

```

This command should only be used in the preamble of a document.

```

974 \@onlypreamble\languageattribute

```

The error text to be issued when an unknown attribute is selected.

```

975 \newcommand*{\@attrerr}[2]{%
976   \bb1@error
977   {The attribute #2 is unknown for language #1.}%
978   {Your command will be ignored, type <return> to proceed}}

```

`\bb1@declare@ttribute` This command adds the new language/attribute combination to the list of known attributes.
Then it defines a control sequence to be executed when the attribute is used in a document. The result of this should be that the macro `\extras...` for the current language is extended, otherwise the attribute will not work as its code is removed from memory at `\begin{document}`.

```

979 \def\bb1@declare@ttribute#1#2#3{%
980   \bb1@xin@{, #2, }{\BabelModifiers,}%
981   \ifin@
982     \AfterBabelLanguage{#1}{\languageattribute{#1}{#2}}%
983   \fi
984   \bb1@add@list\bb1@attributes{#1-#2}%
985   \expandafter\def\csname#1@attr#2\endcsname{#3}}

```

`\bb1@ifattributeset` This internal macro has 4 arguments. It can be used to interpret T_EX code based on whether a certain attribute was set. This command should appear inside the argument to `\AtBeginDocument` because the attributes are set in the document preamble, *after* babel is loaded.

The first argument is the language, the second argument the attribute being checked, and the third and fourth arguments are the true and false clauses.

```

986 \def\bb1@ifattributeset#1#2#3#4{%

```

First we need to find out if any attributes were set; if not we're done.

```

987   \ifx\bb1@known@attribs\@undefined
988     \in@false
989   \else

```

The we need to check the list of known attributes.

```

990     \bb1@xin@{, #1-#2, }{\bb1@known@attribs,}%
991   \fi

```

When we're this far `\ifin@` has a value indicating if the attribute in question was set or not. Just to be safe the code to be executed is 'thrown over the `\fi`'.

```

992   \ifin@
993     \bb1@afterelse#3%
994   \else
995     \bb1@afterfi#4%
996   \fi
997 }

```

`\bb1@ifknown@ttrib` An internal macro to check whether a given language/attribute is known. The macro takes 4 arguments, the language/attribute, the attribute list, the T_EX-code to be executed when the attribute is known and the T_EX-code to be executed otherwise.

```

998 \def\bb1@ifknown@ttrib#1#2{%

```

We first assume the attribute is unknown.

```
999 \let\bbl@tempa\@secondoftwo
```

Then we loop over the list of known attributes, trying to find a match.

```
1000 \bbl@loopx\bbl@tempb{#2}{%
1001   \expandafter\in@\expandafter{\expandafter,\bbl@tempb,}{, #1,}%
1002   \ifin@
```

When a match is found the definition of `\bbl@tempa` is changed.

```
1003   \let\bbl@tempa\@firstoftwo
1004   \else
1005   \fi}%
```

Finally we execute `\bbl@tempa`.

```
1006 \bbl@tempa
1007 }
```

`\bbl@clear@ttribs` This macro removes all the attribute code from \LaTeX 's memory at `\begin{document}` time (if any is present).

```
1008 \def\bbl@clear@ttribs{%
1009   \ifx\bbl@attributes\@undefined\else
1010     \bbl@loopx\bbl@tempa{\bbl@attributes}{%
1011       \expandafter\bbl@clear@ttrib\bbl@tempa.
1012     }%
1013     \let\bbl@attributes\@undefined
1014   \fi}
1015 \def\bbl@clear@ttrib#1-#2.{%
1016   \expandafter\let\csname#1@attr@#2\endcsname\@undefined}
1017 \AtBeginDocument{\bbl@clear@ttribs}
```

8.6 Support for saving macro definitions

To save the meaning of control sequences using `\babel@save`, we use temporary control sequences. To save hash table entries for these control sequences, we don't use the name of the control sequence to be saved to construct the temporary name. Instead we simply use the value of a counter, which is reset to zero each time we begin to save new values. This works well because we release the saved meanings before we begin to save a new set of control sequence meanings (see `\selectlanguage` and `\originalTeX`). Note undefined macros are not undefined any more when saved – they are `\relax`'ed.

`\babel@savecnt` The initialization of a new save cycle: reset the counter to zero.

```
\babel@beginsave 1018 \def\babel@beginsave{\babel@savecnt\z@}
```

Before it's forgotten, allocate the counter and initialize all.

```
1019 \newcount\babel@savecnt
1020 \babel@beginsave
```

`\babel@save` The macro `\babel@save⟨csname⟩` saves the current meaning of the control sequence `⟨csname⟩` to `\originalTeX`²⁷. To do this, we let the current meaning to a temporary control sequence, the restore commands are appended to `\originalTeX` and the counter is incremented.

```
1021 \def\babel@save#1{%
1022   \expandafter\let\csname babel@number\babel@savecnt\endcsname#1\relax
1023   \toks@\expandafter{\originalTeX\let#1=}%
1024   \bbl@exp{%
1025     \def\\originalTeX{\the\toks@<babel@number\babel@savecnt>\relax}}%
1026   \advance\babel@savecnt@ne}
```

²⁷`\originalTeX` has to be expandable, i. e. you shouldn't let it to `\relax`.

`\babel@savevariable` The macro `\babel@savevariable⟨variable⟩` saves the value of the variable. `⟨variable⟩` can be anything allowed after the `\the` primitive.

```
1027 \def\babel@savevariable#1{%
1028   \toks@\expandafter{\originalTeX #1}%
1029   \bbl@exp{\def\originalTeX{\the\toks@\the#1\relax}}}
```

`\bbl@frenchspacing` Some languages need to have `\frenchspacing` in effect. Others don't want that. The command `\bbl@frenchspacing` switches it on when it isn't already in effect and `\bbl@nonfrenchspacing` switches it off if necessary.

```
1030 \def\bbl@frenchspacing{%
1031   \ifnum\the\sfcodes\.=\@m
1032     \let\bbl@nonfrenchspacing\relax
1033   \else
1034     \frenchspacing
1035     \let\bbl@nonfrenchspacing\nonfrenchspacing
1036   \fi}
1037 \let\bbl@nonfrenchspacing\nonfrenchspacing
```

8.7 Short tags

`\babetags` This macro is straightforward. After zapping spaces, we loop over the list and define the macros `\text⟨tag⟩` and `\⟨tag⟩`. Definitions are first expanded so that they don't contain `\csname` but the actual macro.

```
1038 \def\babetags#1{%
1039   \edef\bbl@tempa{\zap@space#1 \@empty}%
1040   \def\bbl@tempb##1=##2\@{#}%
1041   \edef\bbl@tempc{%
1042     \noexpand\newcommand
1043     \expandafter\noexpand\csname ##1\endcsname{%
1044       \noexpand\protect
1045       \expandafter\noexpand\csname otherlanguage*\endcsname{##2}}
1046     \noexpand\newcommand
1047     \expandafter\noexpand\csname text##1\endcsname{%
1048       \noexpand\foreignlanguage{##2}}
1049   \bbl@tempc}%
1050   \bbl@for\bbl@tempa\bbl@tempa{%
1051     \expandafter\bbl@tempb\bbl@tempa\@{}}
```

8.8 Hyphens

`\babelhyphenation` This macro saves hyphenation exceptions. Two macros are used to store them: `\bbl@hyphenation@` for the global ones and `\bbl@hyphenation<lang>` for language ones. See `\bbl@patterns` above for further details. We make sure there is a space between words when multiple commands are used.

```
1052 \@onlypreamble\babelhyphenation
1053 \AtEndOfPackage{%
1054   \newcommand\babelhyphenation[2][\@empty]{%
1055     \ifx\bbl@hyphenation@\relax
1056       \let\bbl@hyphenation@\@empty
1057     \fi
1058     \ifx\bbl@hyphlist\@empty\else
1059       \bbl@warning{%
1060         You must not intermingle \string\selectlanguage\space and\%
1061         \string\babelhyphenation\space or some exceptions will not\%
1062         be taken into account. Reported}%
1063     \fi}
```

```

1064 \ifx\@empty#1%
1065 \protected@edef\bbl@hyphenation@{\bbl@hyphenation@space#2}%
1066 \else
1067 \bbl@vforeach{#1}{%
1068 \def\bbl@tempa{##1}%
1069 \bbl@fixname\bbl@tempa
1070 \bbl@iflanguage\bbl@tempa{%
1071 \bbl@csarg\protected@edef{hyphenation@\bbl@tempa}{%
1072 \bbl@ifunset{bbl@hyphenation@\bbl@tempa}%
1073 \@empty
1074 {\csname bbl@hyphenation@\bbl@tempa\endcsname\space}%
1075 #2}}}%
1076 \fi}}

```

`\bbl@allowhyphens` This macro makes hyphenation possible. Basically its definition is nothing more than `\nobreak \hskip Opt plus Opt`²⁸.

```

1077 \def\bbl@allowhyphens{\ifvmode\else\nobreak\hskip\zskip\fi}
1078 \def\bbl@t@one{T1}
1079 \def\allowhyphens{\ifx\cf@encoding\bbl@t@one\else\bbl@allowhyphens\fi}

```

`\babelhyphen` Macros to insert common hyphens. Note the space before @ in `\babelhyphen`. Instead of protecting it with `\DeclareRobustCommand`, which could insert a `\relax`, we use the same procedure as shorthands, with `\active@prefix`.

```

1080 \newcommand\babellnullhyphen{\char\hyphenchar\font}
1081 \def\babelhyphen{\active@prefix\babelhyphen\bbl@hyphen}
1082 \def\bbl@hyphen{%
1083 \ifstar{\bbl@hyphen@i @}{\bbl@hyphen@i\@empty}}
1084 \def\bbl@hyphen@i#1#2{%
1085 \bbl@ifunset{bbl@hy#1#2\@empty}%
1086 {\csname bbl@#1usehyphen\endcsname{\discretionary{#2}{}{#2}}}%
1087 {\csname bbl@hy#1#2\@empty\endcsname}}

```

The following two commands are used to wrap the “hyphen” and set the behaviour of the rest of the word – the version with a single @ is used when further hyphenation is allowed, while that with @@ if no more hyphen are allowed. In both cases, if the hyphen is preceded by a positive space, breaking after the hyphen is disallowed.

There should not be a discretionary after a hyphen at the beginning of a word, so it is prevented if preceded by a skip. Unfortunately, this does handle cases like “(-suffix)”. `\nobreak` is always preceded by `\leavevmode`, in case the shorthand starts a paragraph.

```

1088 \def\bbl@usehyphen#1{%
1089 \leavevmode
1090 \ifdim\lastskip>\z@\mbox{#1}\else\nobreak#1\fi
1091 \nobreak\hskip\zskip}
1092 \def\bbl@@usehyphen#1{%
1093 \leavevmode\ifdim\lastskip>\z@\mbox{#1}\else#1\fi}

```

The following macro inserts the hyphen char.

```

1094 \def\bbl@hyphenchar{%
1095 \ifnum\hyphenchar\font=\m@ne
1096 \babellnullhyphen
1097 \else
1098 \char\hyphenchar\font
1099 \fi}

```

Finally, we define the hyphen “types”. Their names will not change, so you may use them in ldf’s. After a space, the `\mbox` in `\bbl@hy@nobreak` is redundant.

²⁸ $\mathrm{T}_{\mathrm{E}}\mathrm{X}$ begins and ends a word for hyphenation at a glue node. The penalty prevents a linebreak at this glue node.

```

1100 \def\bbl@hy@soft{\bbl@usehyphen\discretionary{\bbl@hyphenchar}{}}{}
1101 \def\bbl@hy@soft{\bbl@usehyphen\discretionary{\bbl@hyphenchar}{}}{}
1102 \def\bbl@hy@hard{\bbl@usehyphen\bbl@hyphenchar}
1103 \def\bbl@hy@hard{\bbl@usehyphen\bbl@hyphenchar}
1104 \def\bbl@hy@nobreak{\bbl@usehyphen\mbox{\bbl@hyphenchar}}
1105 \def\bbl@hy@nobreak{\mbox{\bbl@hyphenchar}}
1106 \def\bbl@hy@repeat{%
1107   \bbl@usehyphen{%
1108     \discretionary{\bbl@hyphenchar}{\bbl@hyphenchar}{\bbl@hyphenchar}}
1109 \def\bbl@hy@repeat{%
1110   \bbl@usehyphen{%
1111     \discretionary{\bbl@hyphenchar}{\bbl@hyphenchar}{\bbl@hyphenchar}}
1112 \def\bbl@hy@empty{\hskip\z@skip}
1113 \def\bbl@hy@empty{\discretionary{}{}{}}

```

`\bbl@disc` For some languages the macro `\bbl@disc` is used to ease the insertion of discretionary for letters that behave ‘abnormally’ at a breakpoint.

```

1114 \def\bbl@disc#1#2{\nobreak\discretionary{#2-}{}{#1}\bbl@allowhyphens}

```

8.9 Multiencoding strings

The aim following commands is to provide a common interface for strings in several encodings. They also contains several hooks which can be used by `luatex` and `xetex`. The code is organized here with pseudo-guards, so we start with the basic commands.

Tools But first, a couple of tools. The first one makes global a local variable. This is not the best solution, but it works.

```

1115 \def\bbl@tglobal#1{\global\let#1#1}
1116 \def\bbl@recatcode#1{%
1117   \@tempcnta="7F
1118   \def\bbl@tempa{%
1119     \ifnum\@tempcnta>"FF\else
1120       \catcode\@tempcnta=#1\relax
1121       \advance\@tempcnta\@ne
1122       \expandafter\bbl@tempa
1123     \fi}%
1124   \bbl@tempa}

```

The second one. We need to patch `\@uclclist`, but it is done once and only if `\SetCase` is used or if strings are encoded. The code is far from satisfactory for several reasons, including the fact `\@uclclist` is not a list any more. Therefore a package option is added to ignore it. Instead of gobbling the macro getting the next two elements (usually `\reserved@a`), we pass it as argument to `\bbl@uclc`. The parser is restarted inside `\(lang)\bbl@uclc` because we do not know how many expansions are necessary (depends on whether strings are encoded). The last part is tricky – when uppercasing, we have:

```

\let\bbl@tolower\@empty\bbl@toupper\@empty

```

and starts over (and similarly when lowercasing).

```

1125 \@ifpackagewith{babel}{nocase}%
1126   {\let\bbl@patchuclc\relax}%
1127   {\def\bbl@patchuclc{%
1128     \global\let\bbl@patchuclc\relax
1129     \g@addto@macro\@uclclist{\reserved@b{\reserved@b\bbl@uclc}}%
1130     \gdef\bbl@uclc##1{%

```

```

1131 \let\bbl@encoded\bbl@encoded@uclc
1132 \bbl@ifunset{\language @bbl@uclc}% and resumes it
1133 {##1}%
1134 {\let\bbl@tempa##1\relax % Used by LANG@bbl@uclc
1135 \csname\language @bbl@uclc\endcsname}%
1136 {\bbl@tolower\@empty}{\bbl@toupper\@empty}}%
1137 \gdef\bbl@tolower{\csname\language @bbl@lc\endcsname}%
1138 \gdef\bbl@toupper{\csname\language @bbl@uc\endcsname}}
1139 << *More package options >> ≡
1140 \DeclareOption{nocase}{}
1141 << /More package options >>

```

The following package options control the behaviour of `\SetString`.

```

1142 << *More package options >> ≡
1143 \let\bbl@opt@strings\@nnil % accept strings=value
1144 \DeclareOption{strings}{\def\bbl@opt@strings{\BabelStringsDefault}}
1145 \DeclareOption{strings=encoded}{\let\bbl@opt@strings\relax}
1146 \def\BabelStringsDefault{generic}
1147 << /More package options >>

```

Main command This is the main command. With the first use it is redefined to omit the basic setup in subsequent blocks. We make sure strings contain actual letters in the range 128-255, not active characters.

```

1148 \@onlypreamble\StartBabelCommands
1149 \def\StartBabelCommands{%
1150 \begingroup
1151 \bbl@recatcode{11}%
1152 <<Macros local to BabelCommands>>
1153 \def\bbl@provstring##1##2{%
1154 \providecommand##1{##2}%
1155 \bbl@toglobal##1}%
1156 \global\let\bbl@scafter\@empty
1157 \let\StartBabelCommands\bbl@startcmds
1158 \ifx\BabelLanguages\relax
1159 \let\BabelLanguages\CurrentOption
1160 \fi
1161 \begingroup
1162 \let\bbl@screset\@nnil % local flag - disable 1st stopcommands
1163 \StartBabelCommands}
1164 \def\bbl@startcmds{%
1165 \ifx\bbl@screset\@nnil\else
1166 \bbl@usehooks{stopcommands}{}%
1167 \fi
1168 \endgroup
1169 \begingroup
1170 \@ifstar
1171 {\ifx\bbl@opt@strings\@nnil
1172 \let\bbl@opt@strings\BabelStringsDefault
1173 \fi
1174 \bbl@startcmds@i}%
1175 \bbl@startcmds@i}
1176 \def\bbl@startcmds@i#1#2{%
1177 \edef\bbl@L{\zap@space#1 \@empty}%
1178 \edef\bbl@G{\zap@space#2 \@empty}%
1179 \bbl@startcmds@ii}

```

Parse the encoding info to get the label, input, and font parts.

Select the behaviour of \SetString. There are two main cases, depending of if there is an optional argument: without it and strings=encoded, strings are defined always; otherwise, they are set only if they are still undefined (ie, fallback values). With labelled blocks and strings=encoded, define the strings, but with another value, define strings only if the current label or font encoding is the value of strings; otherwise (ie, no strings or a block whose label is not in strings=) do nothing.

We presume the current block is not loaded, and therefore set (above) a couple of default values to gobble the arguments. Then, these macros are redefined if necessary according to several parameters.

```

1180 \newcommand\bbbl@startcmds@ii[1][\@empty]{%
1181   \let\SetString\@gobbletwo
1182   \let\bbbl@stringdef\@gobbletwo
1183   \let\AfterBabelCommands\@gobble
1184   \ifx\@empty#1%
1185     \def\bbbl@sc@label{generic}%
1186     \def\bbbl@encstring##1##2{%
1187       \ProvideTextCommandDefault##1{##2}%
1188       \bbbl@tglobal##1%
1189       \expandafter\bbbl@tglobal\csname\string?\string##1\endcsname}%
1190       \let\bbbl@sctest\in@true
1191   \else
1192     \let\bbbl@sc@charset\space % <- zapped below
1193     \let\bbbl@sc@fontenc\space % <- " "
1194     \def\bbbl@tempa##1=##2\@nil{%
1195       \bbbl@csarg\edef{sc@\zap@space##1 \@empty}{##2 }}%
1196       \bbbl@vforeach{label=#1}{\bbbl@tempa##1\@nil}%
1197       \def\bbbl@tempa##1 ##2{% space -> comma
1198         ##1%
1199         \ifx\@empty##2\else\ifx,##1,\else,\fi\bbbl@afterfi\bbbl@tempa##2\fi}%
1200       \edef\bbbl@sc@fontenc{\expandafter\bbbl@tempa\bbbl@sc@fontenc\@empty}%
1201       \edef\bbbl@sc@label{\expandafter\zap@space\bbbl@sc@label\@empty}%
1202       \edef\bbbl@sc@charset{\expandafter\zap@space\bbbl@sc@charset\@empty}%
1203       \def\bbbl@encstring##1##2{%
1204         \bbbl@foreach\bbbl@sc@fontenc{%
1205           \bbbl@ifunset{T#####1}%
1206           {}%
1207           {\ProvideTextCommand##1{#####1}{##2}%
1208             \bbbl@tglobal##1%
1209             \expandafter
1210             \bbbl@tglobal\csname#####1\string##1\endcsname}}}%
1211       \def\bbbl@sctest{%
1212         \bbbl@xin@{\bbbl@opt@strings,}{,\bbbl@sc@label,\bbbl@sc@fontenc,}}%
1213   \fi
1214   \ifx\bbbl@opt@strings\@nnil % ie, no strings key -> defaults
1215   \else\ifx\bbbl@opt@strings\relax % ie, strings=encoded
1216     \let\AfterBabelCommands\bbbl@aftercmds
1217     \let\SetString\bbbl@setstring
1218     \let\bbbl@stringdef\bbbl@encstring
1219   \else % ie, strings=value
1220     \bbbl@sctest
1221   \fi
1222   \let\AfterBabelCommands\bbbl@aftercmds
1223   \let\SetString\bbbl@setstring
1224   \let\bbbl@stringdef\bbbl@provstring
1225 \fi\fi\fi
1226 \bbbl@scswitch
1227 \ifx\bbbl@G\@empty
1228   \def\SetString##1##2{%

```

```

1229 \bbl@error{Missing group for string \string##1}%
1230 {You must assign strings to some category, typically\\%
1231 captions or extras, but you set none}}%
1232 \fi
1233 \ifx\@empty#1%
1234 \bbl@usehooks{defaultcommands}{}%
1235 \else
1236 \@expandtwoargs
1237 \bbl@usehooks{encodedcommands}{\bbl@sc@charset}{\bbl@sc@fontenc}}%
1238 \fi}

```

There are two versions of `\bbl@scswitch`. The first version is used when `ldfs` are read, and it makes sure `\group`*(language)* is reset, but only once (`\bbl@screset` is used to keep track of this). The second version is used in the preamble and packages loaded after `babel` and does nothing. The macro `\bbl@forlang` loops `\bbl@L` but its body is executed only if the value is in `\BabelLanguages` (inside `babel`) or `\date`*(language)* is defined (after `babel` has been loaded). There are also two version of `\bbl@forlang`. The first one skips the current iteration if the language is not in `\BabelLanguages` (used in `ldfs`), and the second one skips undefined languages (after `babel` has been loaded).

```

1239 \def\bbl@forlang#1#2{%
1240 \bbl@for#1\bbl@L{%
1241 \bbl@xin@{, #1, }{, \BabelLanguages,}%
1242 \ifin@#2\relax\fi}}
1243 \def\bbl@scswitch{%
1244 \bbl@forlang\bbl@tempa{%
1245 \ifx\bbl@G\@empty\else
1246 \ifx\SetString\@gobbletwo\else
1247 \edef\bbl@GL{\bbl@G\bbl@tempa}%
1248 \bbl@xin@{, \bbl@GL, }{, \bbl@screset,}%
1249 \ifin@\else
1250 \global\expandafter\let\csname\bbl@GL\endcsname\@undefined
1251 \xdef\bbl@screset{\bbl@screset, \bbl@GL}%
1252 \fi
1253 \fi
1254 \fi}}
1255 \AtEndOfPackage{%
1256 \def\bbl@forlang#1#2{\bbl@for#1\bbl@L{\bbl@ifunset{date#1}{}{#2}}}%
1257 \let\bbl@scswitch\relax}
1258 \onlypreamble\EndBabelCommands
1259 \def\EndBabelCommands{%
1260 \bbl@usehooks{stopcommands}{}%
1261 \endgroup
1262 \endgroup
1263 \bbl@scafter}

```

Now we define commands to be used inside `\StartBabelCommands`.

Strings The following macro is the actual definition of `\SetString` when it is “active”. First save the “switcher”. Create it if undefined. Strings are defined only if undefined (ie, like `\providescommand`). With the event `stringprocess` you can preprocess the string by manipulating the value of `\BabelString`. If there are several hooks assigned to this event, preprocessing is done in the same order as defined. Finally, the string is set.

```

1264 \def\bbl@setstring#1#2{%
1265 \bbl@forlang\bbl@tempa{%
1266 \edef\bbl@LC{\bbl@tempa\bbl@stripslash#1}%
1267 \bbl@ifunset{\bbl@LC}% eg, \germanchaptername
1268 {\global\expandafter % TODO - con \bbl@exp ?

```



```

1269      \bbl@add\csname\bbl@G\bbl@tempa\expandafter\endcsname\expandafter
1270      {\expandafter\bbl@scset\expandafter#1\csname\bbl@LC\endcsname}}%
1271      {}%
1272      \def\BabelString{#2}%
1273      \bbl@usehooks{stringprocess}{}%
1274      \expandafter\bbl@stringdef
1275      \csname\bbl@LC\expandafter\endcsname\expandafter{\BabelString}}

```

Now, some additional stuff to be used when encoded strings are used. Captions then include `\bbl@encoded` for string to be expanded in case transformations. It is `\relax` by default, but in `\MakeUppercase` and `\MakeLowercase` its value is a modified expandable `\@changed@cmd`.

```

1276 \ifx\bbl@opt@strings\relax
1277   \def\bbl@scset#1#2{\def#1{\bbl@encoded#2}}
1278   \bbl@patchuclc
1279   \let\bbl@encoded\relax
1280   \def\bbl@encoded@uclc#1{%
1281     \@inmathwarn#1%
1282     \expandafter\ifx\csname\cf@encoding\string#1\endcsname\relax
1283       \expandafter\ifx\csname ?\string#1\endcsname\relax
1284         \TextSymbolUnavailable#1%
1285       \else
1286         \csname ?\string#1\endcsname
1287       \fi
1288     \else
1289       \csname\cf@encoding\string#1\endcsname
1290     \fi}
1291 \else
1292   \def\bbl@scset#1#2{\def#1{#2}}
1293 \fi

```

Define `\SetStringLoop`, which is actually set inside `\StartBabelCommands`. The current definition is somewhat complicated because we need a count, but `\count@` is not under our control (remember `\SetString` may call hooks). Instead of defining a dedicated count, we just “pre-expand” its value.

```

1294 <<(*Macros local to BabelCommands)>> ≡
1295 \def\SetStringLoop##1##2{%
1296   \def\bbl@templ####1{\expandafter\noexpand\csname##1\endcsname}%
1297   \count@\z@
1298   \bbl@loop\bbl@tempa{##2}{% empty items and spaces are ok
1299     \advance\count@\@ne
1300     \toks@\expandafter{\bbl@tempa}%
1301     \bbl@exp{%
1302       \\SetString\bbl@templ{\romannumeral\count@}{\the\toks@}%
1303       \count@=\the\count@\relax}}}%
1304 <</Macros local to BabelCommands>>

```

Delaying code Now the definition of `\AfterBabelCommands` when it is activated.

```

1305 \def\bbl@aftercmds#1{%
1306   \toks@\expandafter{\bbl@scafter#1}%
1307   \xdef\bbl@scafter{\the\toks@}}

```

Case mapping The command `\SetCase` provides a way to change the behaviour of `\MakeUppercase` and `\MakeLowercase`. `\bbl@tempa` is set by the patched `\@uclclist` to the parsing command.

```

1308 <<(*Macros local to BabelCommands)>> ≡

```

```

1309 \newcommand\SetCase[3][]{%
1310   \bbl@patchuclc
1311   \bbl@forlang\bbl@tempa{%
1312     \expandafter\bbl@encstring
1313     \csname\bbl@tempa @bbl@uclc\endcsname{\bbl@tempa##1}%
1314     \expandafter\bbl@encstring
1315     \csname\bbl@tempa @bbl@uc\endcsname{##2}%
1316     \expandafter\bbl@encstring
1317     \csname\bbl@tempa @bbl@lc\endcsname{##3}}}%
1318 <</Macros local to BabelCommands>>

```

Macros to deal with case mapping for hyphenation. To decide if the document is monolingual or multilingual, we make a rough guess – just see if there is a comma in the languages list, built in the first pass of the package options.

```

1319 <<(*Macros local to BabelCommands)>> ≡
1320 \newcommand\SetHyphenMap[1]{%
1321   \bbl@forlang\bbl@tempa{%
1322     \expandafter\bbl@stringdef
1323     \csname\bbl@tempa @bbl@hyphenmap\endcsname{##1}}}%
1324 <</Macros local to BabelCommands>>

```

There are 3 helper macros which do most of the work for you.

```

1325 \newcommand\BabelLower[2]{% one to one.
1326   \ifnum\lccode#1=#2\else
1327     \babel@savevariable{\lccode#1}%
1328     \lccode#1=#2\relax
1329   \fi}
1330 \newcommand\BabelLowerMM[4]{% many-to-many
1331   \@tempcnta=#1\relax
1332   \@tempcntb=#4\relax
1333   \def\bbl@tempa{%
1334     \ifnum\@tempcnta>#2\else
1335       \@expandtwoargs\BabelLower{\the\@tempcnta}{\the\@tempcntb}%
1336       \advance\@tempcnta#3\relax
1337       \advance\@tempcntb#3\relax
1338       \expandafter\bbl@tempa
1339     \fi}%
1340   \bbl@tempa}
1341 \newcommand\BabelLowerMO[4]{% many-to-one
1342   \@tempcnta=#1\relax
1343   \def\bbl@tempa{%
1344     \ifnum\@tempcnta>#2\else
1345       \@expandtwoargs\BabelLower{\the\@tempcnta}{#4}%
1346       \advance\@tempcnta#3
1347       \expandafter\bbl@tempa
1348     \fi}%
1349   \bbl@tempa}

```

The following package options control the behaviour of hyphenation mapping.

```

1350 <<(*More package options)>> ≡
1351 \DeclareOption{hyphenmap=off}{\chardef\bbl@opt@hyphenmap\z@}
1352 \DeclareOption{hyphenmap=first}{\chardef\bbl@opt@hyphenmap\@ne}
1353 \DeclareOption{hyphenmap=select}{\chardef\bbl@opt@hyphenmap\tw@}
1354 \DeclareOption{hyphenmap=other}{\chardef\bbl@opt@hyphenmap\thr@@}
1355 \DeclareOption{hyphenmap=other*}{\chardef\bbl@opt@hyphenmap4\relax}
1356 <</More package options>>

```

Initial setup to provide a default behaviour if hyphenmap is not set.

```

1357 \AtEndOfPackage{%

```

```

1358 \ifx\bbl@opt@hyphenmap\undefined
1359 \bbl@xin@{,}\bbl@language@opts}%
1360 \chardef\bbl@opt@hyphenmap\ifin@4\else\ne\fi
1361 \fi}

```

8.10 Macros common to a number of languages

`\set@low@box` The following macro is used to lower quotes to the same level as the comma. It prepares its argument in box register 0.

```

1362 \def\set@low@box#1{\setbox\tw@\hbox{,}\setbox\z@\hbox{#1}%
1363 \dimen\z@\ht\z@ \advance\dimen\z@ -\ht\tw@%
1364 \setbox\z@\hbox{\lower\dimen\z@ \box\z@}\ht\z@\ht\tw@ \dp\z@\dp\tw@}

```

`\save@sf@q` The macro `\save@sf@q` is used to save and reset the current space factor.

```

1365 \def\save@sf@q#1{\leavevmode
1366 \begingroup
1367 \edef\SF{\spacefactor\the\spacefactor}#1\SF
1368 \endgroup}

```

8.11 Making glyphs available

This section makes a number of glyphs available that either do not exist in the OT1 encoding and have to be ‘faked’, or that are not accessible through `T1enc.def`.

8.11.1 Quotation marks

`\quotedblbase` In the T1 encoding the opening double quote at the baseline is available as a separate character, accessible via `\quotedblbase`. In the OT1 encoding it is not available, therefore we make it available by lowering the normal open quote character to the baseline.

```

1369 \ProvideTextCommand{\quotedblbase}{OT1}{%
1370 \save@sf@q{\set@low@box{\textquotedblright\}%
1371 \box\z@\kern-.04em\bbl@allowhyphens}}

```

Make sure that when an encoding other than OT1 or T1 is used this glyph can still be typeset.

```

1372 \ProvideTextCommandDefault{\quotedblbase}{%
1373 \UseTextSymbol{OT1}{\quotedblbase}}

```

`\quotesinglbase` We also need the single quote character at the baseline.

```

1374 \ProvideTextCommand{\quotesinglbase}{OT1}{%
1375 \save@sf@q{\set@low@box{\textquoteright\}%
1376 \box\z@\kern-.04em\bbl@allowhyphens}}

```

Make sure that when an encoding other than OT1 or T1 is used this glyph can still be typeset.

```

1377 \ProvideTextCommandDefault{\quotesinglbase}{%
1378 \UseTextSymbol{OT1}{\quotesinglbase}}

```

`\guillemotleft` The guillemet characters are not available in OT1 encoding. They are faked.

```

\guillemotright 1379 \ProvideTextCommand{\guillemotleft}{OT1}{%
1380 \ifmmode
1381 \ll
1382 \else
1383 \save@sf@q{\nobreak
1384 \raise.2ex\hbox{\scriptscriptstyle\ll}\bbl@allowhyphens}%
1385 \fi}

```

```

1386 \ProvideTextCommand{\guillemotright}{OT1}{%
1387   \ifmmode
1388     \gg
1389   \else
1390     \save@sf@q{\nobreak
1391       \raise.2ex\hbox{$\scriptscriptstyle\gg$}\bbl@allowhyphens}%
1392   \fi}

```

Make sure that when an encoding other than OT1 or T1 is used these glyphs can still be typeset.

```

1393 \ProvideTextCommandDefault{\guillemotleft}{%
1394   \UseTextSymbol{OT1}{\guillemotleft}}
1395 \ProvideTextCommandDefault{\guillemotright}{%
1396   \UseTextSymbol{OT1}{\guillemotright}}

```

`\guilsinglleft` The single guillemets are not available in OT1 encoding. They are faked.
`\guilsinglright`

```

1397 \ProvideTextCommand{\guilsinglleft}{OT1}{%
1398   \ifmmode
1399     <%
1400   \else
1401     \save@sf@q{\nobreak
1402       \raise.2ex\hbox{$\scriptscriptstyle<$}\bbl@allowhyphens}%
1403   \fi}
1404 \ProvideTextCommand{\guilsinglright}{OT1}{%
1405   \ifmmode
1406     >%
1407   \else
1408     \save@sf@q{\nobreak
1409       \raise.2ex\hbox{$\scriptscriptstyle>$}\bbl@allowhyphens}%
1410   \fi}

```

Make sure that when an encoding other than OT1 or T1 is used these glyphs can still be typeset.

```

1411 \ProvideTextCommandDefault{\guilsinglleft}{%
1412   \UseTextSymbol{OT1}{\guilsinglleft}}
1413 \ProvideTextCommandDefault{\guilsinglright}{%
1414   \UseTextSymbol{OT1}{\guilsinglright}}

```

8.11.2 Letters

`\ij` The dutch language uses the letter ‘ij’. It is available in T1 encoded fonts, but not in the OT1 encoded fonts. Therefore we fake it for the OT1 encoding.

```

1415 \DeclareTextCommand{\ij}{OT1}{%
1416   i\kern-0.02em\bbl@allowhyphens j}
1417 \DeclareTextCommand{\IJ}{OT1}{%
1418   I\kern-0.02em\bbl@allowhyphens J}
1419 \DeclareTextCommand{\ij}{T1}{\char188}
1420 \DeclareTextCommand{\IJ}{T1}{\char156}

```

Make sure that when an encoding other than OT1 or T1 is used these glyphs can still be typeset.

```

1421 \ProvideTextCommandDefault{\ij}{%
1422   \UseTextSymbol{OT1}{\ij}}
1423 \ProvideTextCommandDefault{\IJ}{%
1424   \UseTextSymbol{OT1}{\IJ}}

```

`\dj` The croatian language needs the letters `\dj` and `\DJ`; they are available in the T1 encoding, `\DJ` but not in the OT1 encoding by default.

Some code to construct these glyphs for the OT1 encoding was made available to me by Stipcevic Mario, (stipcevic@olimp.irb.hr).

```

1425 \def\crrtic@{\hrule height0.1ex width0.3em}
1426 \def\crttic@{\hrule height0.1ex width0.33em}
1427 \def\ddj@{%
1428   \setbox0\hbox{d}\dimen@=\ht0
1429   \advance\dimen@1ex
1430   \dimen@.45\dimen@
1431   \dimen@ii\expandafter\rem@pt\the\fontdimen\@ne\font\dimen@
1432   \advance\dimen@ii.5ex
1433   \leavevmode\rlap{\raise\dimen@\hbox{\kern\dimen@ii\vbox{\crrtic@}}}}
1434 \def\DDJ@{%
1435   \setbox0\hbox{D}\dimen@=.55\ht0
1436   \dimen@ii\expandafter\rem@pt\the\fontdimen\@ne\font\dimen@
1437   \advance\dimen@ii.15ex % correction for the dash position
1438   \advance\dimen@ii-.15\fontdimen7\font % correction for cmtt font
1439   \dimen\thr@@\expandafter\rem@pt\the\fontdimen7\font\dimen@
1440   \leavevmode\rlap{\raise\dimen@\hbox{\kern\dimen@ii\vbox{\crttic@}}}}
1441 %
1442 \DeclareTextCommand{\dj}{OT1}{\ddj@ d}
1443 \DeclareTextCommand{\DJ}{OT1}{\DDJ@ D}

```

Make sure that when an encoding other than OT1 or T1 is used these glyphs can still be typeset.

```

1444 \ProvideTextCommandDefault{\dj}{%
1445   \UseTextSymbol{OT1}{\dj}}
1446 \ProvideTextCommandDefault{\DJ}{%
1447   \UseTextSymbol{OT1}{\DJ}}

```

\SS For the T1 encoding \SS is defined and selects a specific glyph from the font, but for other encodings it is not available. Therefore we make it available here.

```

1448 \DeclareTextCommand{\SS}{OT1}{\SS}
1449 \ProvideTextCommandDefault{\SS}{\UseTextSymbol{OT1}{\SS}}

```

8.11.3 Shorthands for quotation marks

Shorthands are provided for a number of different quotation marks, which make them usable both outside and inside mathmode. They are defined with \ProvideTextCommandDefault, but this is very likely not required because their definitions are based on encoding dependent macros.

\glq The ‘german’ single quotes.

```

\grq 1450 \ProvideTextCommandDefault{\glq}{%
1451   \textormath{\quotesinglbase}{\mbox{\quotesinglbase}}}

```

The definition of \grq depends on the fontencoding. With T1 encoding no extra kerning is needed.

```

1452 \ProvideTextCommand{\grq}{T1}{%
1453   \textormath{\textquoteleft}{\mbox{\textquoteleft}}}
1454 \ProvideTextCommand{\grq}{TU}{%
1455   \textormath{\textquoteleft}{\mbox{\textquoteleft}}}
1456 \ProvideTextCommand{\grq}{OT1}{%
1457   \save@sf@q{\kern-.0125em
1458     \textormath{\textquoteleft}{\mbox{\textquoteleft}}}%
1459     \kern.07em\relax}}
1460 \ProvideTextCommandDefault{\grq}{\UseTextSymbol{OT1}{\grq}}

```

`\glqq` The ‘german’ double quotes.

```
\grqq 1461 \ProvideTextCommandDefault{\glqq}{%
1462 \textormath{\quotedblbase}{\mbox{\quotedblbase}}}

The definition of \grqq depends on the fontencoding. With T1 encoding no extra kerning is
needed.

1463 \ProvideTextCommand{\grqq}{T1}{%
1464 \textormath{\textquotedblleft}{\mbox{\textquotedblleft}}}
1465 \ProvideTextCommand{\grqq}{TU}{%
1466 \textormath{\textquotedblleft}{\mbox{\textquotedblleft}}}
1467 \ProvideTextCommand{\grqq}{OT1}{%
1468 \save@sf@q{\kern-.07em
1469 \textormath{\textquotedblleft}{\mbox{\textquotedblleft}}}%
1470 \kern.07em\relax}}
1471 \ProvideTextCommandDefault{\grqq}{\UseTextSymbol{OT1}\grqq}
```

`\flq` The ‘french’ single guillemets.

```
\frq 1472 \ProvideTextCommandDefault{\flq}{%
1473 \textormath{\guilsinglleft}{\mbox{\guilsinglleft}}}
1474 \ProvideTextCommandDefault{\frq}{%
1475 \textormath{\guilsinglright}{\mbox{\guilsinglright}}}
```

`\flqq` The ‘french’ double guillemets.

```
\frqq 1476 \ProvideTextCommandDefault{\flqq}{%
1477 \textormath{\guillemotleft}{\mbox{\guillemotleft}}}
1478 \ProvideTextCommandDefault{\frqq}{%
1479 \textormath{\guillemotright}{\mbox{\guillemotright}}}
```

8.11.4 Umlauts and tremas

The command `\` needs to have a different effect for different languages. For German for instance, the ‘umlaut’ should be positioned lower than the default position for placing it over the letters a, o, u, A, O and U. When placed over an e, i, E or I it can retain its normal position. For Dutch the same glyph is always placed in the lower position.

`\umlauthigh` To be able to provide both positions of `\` we provide two commands to switch the
`\umlautlow` positioning, the default will be `\umlauthigh` (the normal positioning).

```
1480 \def\umlauthigh{%
1481 \def\bbl@umlauta##1{\leavevmode\bggroup%
1482 \expandafter\accent\csname\fontencoding dqpos\endcsname
1483 ##1\bbl@allowhyphens\egroup}%
1484 \let\bbl@umlaute\bbl@umlauta}
1485 \def\umlautlow{%
1486 \def\bbl@umlauta{\protect\lower@umlaut}}
1487 \def\umlautelow{%
1488 \def\bbl@umlaute{\protect\lower@umlaut}}
1489 \umlauthigh
```

`\lower@umlaut` The command `\lower@umlaut` is used to position the `\` closer to the letter.

We want the umlaut character lowered, nearer to the letter. To do this we need an extra *⟨dimen⟩* register.

```
1490 \expandafter\ifx\csname U@D\endcsname\relax
1491 \csname newdimen\endcsname\U@D
1492 \fi
```

The following code fools \TeX 's `make_accent` procedure about the current x-height of the font to force another placement of the umlaut character. First we have to save the current x-height of the font, because we'll change this font dimension and this is always done globally.

Then we compute the new x-height in such a way that the umlaut character is lowered to the base character. The value of `.45ex` depends on the METAFONT parameters with which the fonts were built. (Just try out, which value will look best.) If the new x-height is too low, it is not changed. Finally we call the `\accent` primitive, reset the old x-height and insert the base character in the argument.

```

1493 \def\lower@umlaut#1{%
1494   \leavevmode\bggroup
1495     \U@D 1ex%
1496     {\setbox\z@\hbox{%
1497       \expandafter\char\csname\fontencoding dqpos\endcsname}%
1498       \dimen@ -.45ex\advance\dimen@\ht\z@
1499       \ifdim 1ex<\dimen@ \fontdimen5\font\dimen@ \fi}%
1500     \expandafter\accent\csname\fontencoding dqpos\endcsname
1501     \fontdimen5\font\U@D #1%
1502   \egroup}

```

For all vowels we declare `\` to be a composite command which uses `\bbl@umlauta` or `\bbl@umlaute` to position the umlaut character. We need to be sure that these definitions override the ones that are provided when the package `fontenc` with option `OT1` is used. Therefore these declarations are postponed until the beginning of the document. Note these definitions only apply to some languages, but `babel` sets them for *all* languages – you may want to redefine `\bbl@umlauta` and/or `\bbl@umlaute` for a language in the corresponding `ldf` (using the `babel` switching mechanism, of course).

```

1503 \AtBeginDocument{%
1504   \DeclareTextCompositeCommand{\}{OT1}{a}{\bbl@umlauta{a}}%
1505   \DeclareTextCompositeCommand{\}{OT1}{e}{\bbl@umlaute{e}}%
1506   \DeclareTextCompositeCommand{\}{OT1}{i}{\bbl@umlaute{i}}%
1507   \DeclareTextCompositeCommand{\}{OT1}{\i}{\bbl@umlaute{i}}%
1508   \DeclareTextCompositeCommand{\}{OT1}{o}{\bbl@umlauta{o}}%
1509   \DeclareTextCompositeCommand{\}{OT1}{u}{\bbl@umlauta{u}}%
1510   \DeclareTextCompositeCommand{\}{OT1}{A}{\bbl@umlauta{A}}%
1511   \DeclareTextCompositeCommand{\}{OT1}{E}{\bbl@umlaute{E}}%
1512   \DeclareTextCompositeCommand{\}{OT1}{I}{\bbl@umlaute{I}}%
1513   \DeclareTextCompositeCommand{\}{OT1}{O}{\bbl@umlauta{O}}%
1514   \DeclareTextCompositeCommand{\}{OT1}{U}{\bbl@umlauta{U}}%
1515 }

```

Finally, the default is to use English as the main language.

```

1516 \ifx\l@english\@undefined
1517   \chardef\l@english\z@
1518 \fi
1519 \main@language{english}

```

Now we load definition files for engines.

```

1520 \ifcase\bbl@engine\or
1521   \input luababel.def
1522 \or
1523   \input xebabel.def
1524 \fi

```

9 The kernel of Babel (babel.def, only L^AT_EX)

9.1 The redefinition of the style commands

The rest of the code in this file can only be processed by L^AT_EX, so we check the current format. If it is plain T_EX, processing should stop here. But, because of the need to limit the scope of the definition of `\format`, a macro that is used locally in the following `\if` statement, this comparison is done inside a group. To prevent T_EX from complaining about an unclosed group, the processing of the command `\endinput` is deferred until after the group is closed. This is accomplished by the command `\aftergroup`.

```
1525 {\def\format{lpplain}
1526 \ifx\fmtname\format
1527 \else
1528   \def\format{LaTeX2e}
1529   \ifx\fmtname\format
1530   \else
1531     \aftergroup\endinput
1532   \fi
1533 \fi}
```

9.2 Creating languages

`\babelprovide` is a general purpose tool for creating languages. Currently it just creates the language infrastructure, but in the future it will be able to read data from ini files, as well as to create variants. Unlike the nil pseudo-language, captions are defined, but with a warning to invite the user to provide the real string.

```
1534 \newcommand\babelprovide[2][]{%
1535   \let\bbl@savelangname\language
1536   \def\language{#2}%
1537   \let\bbl@KVP@captions\@nil
1538   \let\bbl@KVP@import\@nil
1539   \let\bbl@KVP@main\@nil
1540   \let\bbl@KVP@script\@nil
1541   \let\bbl@KVP@language\@nil
1542   \let\bbl@KVP@dir\@nil
1543   \let\bbl@KVP@hyphenrules\@nil
1544   \bbl@forkv{#1}{\bbl@csarg\def{KVP@##1}{##2}}% TODO - error handling
1545   \ifx\bbl@KVP@captions\@nil
1546     \let\bbl@KVP@captions\bbl@KVP@import
1547   \fi
1548   \bbl@ifunset{date#2}%
1549     {\bbl@provide@new{#2}}%
1550     {\bbl@ifblank{#1}%
1551       {\bbl@error
1552         {If you want to modify `#2' you must tell how in\\%
1553         the optional argument. Currently there are three\\%
1554         options: captions=lang-tag, hyphenrules=lang-list\\%
1555         import=lang-tag}%
1556       {Use this macro as documented}}%
1557     {\bbl@provide@renew{#2}}%
1558   \bbl@exp{\bbl@babelensure[exclude=\\today]{#2}}%
1559   \ifx\bbl@KVP@script\@nil\else
1560     \bbl@csarg\edef{sname@#2}{\bbl@KVP@script}%
1561   \fi
1562   \ifx\bbl@KVP@language\@nil\else
1563     \bbl@csarg\edef{lname@#2}{\bbl@KVP@language}%
1564   \fi}
```



```
1565 \let\language\bb1@savelangname}
```

Depending on whether or not the language exists, we define two macros.

```
1566 \def\bb1@provide@new#1{%
1567   \@namedef{date#1}{}% marks lang exists - required by \StartBabelCommands
1568   \@namedef{extras#1}{}%
1569   \@namedef{noextras#1}{}%
1570   \StartBabelCommands*{#1}{captions}%
1571   \ifx\bb1@KVP@captions\@nil %      and also if import, implicit
1572     \def\bb1@tempb##1{%             elt for \bb1@captionslist
1573       \ifx##1\@empty\else
1574         \bb1@exp{%
1575           \\SetString\\##1{%
1576             \\bb1@nocaption{\bb1@stripslash##1}{\<#1\bb1@stripslash##1>}}}%
1577         \expandafter\bb1@tempb
1578       \fi}%
1579   \expandafter\bb1@tempb\bb1@captionslist\@empty
1580   \else
1581     \bb1@read@ini{\bb1@KVP@captions}% Here all letters cat = 11
1582     \bb1@after@ini
1583     \bb1@savestrings
1584     \fi
1585   \StartBabelCommands*{#1}{date}%
1586   \ifx\bb1@KVP@import\@nil
1587     \bb1@exp{%
1588       \\SetString\\today{\bb1@nocaption{today}{\<#1today>}}}%
1589   \else
1590     \bb1@savetoday
1591     \bb1@savedate
1592     \fi
1593   \EndBabelCommands
1594   \bb1@exp{%
1595     \def\<#1hyphenmins>{%
1596       {\bb1@ifunset{\bb1@lfthm@#1}{2}{\@nameuse{\bb1@lfthm@#1}}}%
1597       {\bb1@ifunset{\bb1@rgthm@#1}{3}{\@nameuse{\bb1@rgthm@#1}}}}}%
1598   \bb1@provide@hyphens{#1}%
1599   \ifx\bb1@KVP@main\@nil\else
1600     \expandafter\main@language\expandafter{#1}%
1601   \fi}
1602 \def\bb1@provide@renew#1{%
1603   \ifx\bb1@KVP@captions\@nil\else
1604     \StartBabelCommands*{#1}{captions}%
1605     \bb1@read@ini{\bb1@KVP@captions}% Here all letters cat = 11
1606     \bb1@after@ini
1607     \bb1@savestrings
1608     \EndBabelCommands
1609   \fi
1610   \ifx\bb1@KVP@import\@nil\else
1611     \StartBabelCommands*{#1}{date}%
1612     \bb1@savetoday
1613     \bb1@savedate
1614     \EndBabelCommands
1615   \fi
1616   \bb1@provide@hyphens{#1}}
```

The hyphenrules option is handled with an auxiliary macro.

```
1617 \def\bb1@provide@hyphens#1{%
1618   \let\bb1@tempa\relax
1619   \ifx\bb1@KVP@hyphenrules\@nil\else
```

```

1620 \bbl@replace\bbl@KVP@hyphenrules{ }{,}%
1621 \bbl@foreach\bbl@KVP@hyphenrules{%
1622 \ifx\bbl@tempa\relax % if not yet found
1623 \bbl@ifsamestring{##1}{+}%
1624 {{\bbl@exp{\addlanguage\<l@##1>}}}%
1625 }%
1626 \bbl@ifunset{l@##1}%
1627 }%
1628 {\bbl@exp{\let\bbl@tempa\<l@##1>}}%
1629 \fi}%
1630 \fi
1631 \ifx\bbl@tempa\relax % if no opt or no language in opt found
1632 \ifx\bbl@KVP@import\@nil\else % if importing
1633 \bbl@exp{%
1634 \bbl@ifblank{\@nameuse\bbl@hyphr@#1}}%
1635 }%
1636 {\let\bbl@tempa\<l@\@nameuse\bbl@hyphr@\language\>}}%
1637 \fi
1638 \fi
1639 \bbl@ifunset\bbl@tempa{% ie, relax or undefined
1640 {\bbl@ifunset{l@##1}% no hyphenrules found - fallback
1641 {\bbl@exp{\adddialect\<l@##1>\language}}%
1642 }% so, l@<lang> is ok - nothing to do
1643 {\bbl@exp{\adddialect\<l@##1>\bbl@tempa}}% found in opt list or ini

```

The reader of ini files. There are 3 possible cases: a section name (in the form [...]), a comment (starting with ;) and a key/value pair. *TODO - Work in progress.*

```

1644 \def\bbl@read@ini#1{%
1645 \openin1=babel-#1.ini
1646 \ifeof1
1647 \bbl@error
1648 {There is no ini file for the requested language\%
1649 (#1). Perhaps you misspelled it or your installation\%
1650 is not complete.}%
1651 {Fix the name or reinstall babel.}%
1652 \else
1653 \let\bbl@section\@empty
1654 \let\bbl@savestrings\@empty
1655 \let\bbl@savetoday\@empty
1656 \let\bbl@savestate\@empty
1657 \let\bbl@inireader\bbl@iniskip
1658 \bbl@info{Importing data from babel-#1.ini for \language}%
1659 \loop
1660 \endlinechar@m@ne
1661 \read1 to \bbl@line
1662 \endlinechar\^^M
1663 \if \ifeof1\fi \if \relax % Trick, because inside \loop
1664 \ifx\bbl@line\@empty\else
1665 \expandafter\bbl@iniline\bbl@line\bbl@iniline
1666 \fi
1667 \repeat
1668 \fi}
1669 \def\bbl@iniline#1\bbl@iniline{%
1670 \@ifnextchar[\bbl@inisec{\@ifnextchar\bbl@iniskip\bbl@inireader}#1\@}% ]

```

The special cases for comment lines and sections are handled by the two following commands. In sections, we provide the possibility to take extra actions at the end or at the start (TODO - but note the last section is not ended). By default, key=val pairs are ignored.

```

1671 \def\bbl@iniskip#1\@{ } % if starts with ;

```

```

1672 \def\bb@inisec[#1]#2\@@{%    if starts with opening bracket
1673 \nameuse{bb@secpost\bb@section}% ends previous section
1674 \def\bb@section{#1}%
1675 \nameuse{bb@secpre\bb@section}% starts current section
1676 \bb@ifunset{bb@secline@#1}%
1677 {\let\bb@inireader\bb@iniskip}%
1678 {\bb@exp{\let\\bb@inireader\<bb@secline@#1>}}}

```

Reads a key=val line and stores the trimmed val in \bb@@kv@<section>.<key>.

```

1679 \def\bb@inikv#1=#2\@@{%    key=value
1680 \bb@trim\def\bb@tempa{#1}%
1681 \bb@trim\toks@{#2}%
1682 \bb@csarg\edef{@kv@\bb@section.\bb@tempa}{\the\toks@}}

```

The previous assignments are local, so we need to export them. If the value is empty, we can provide a default value.

```

1683 \def\bb@exportkey#1#2#3{%
1684 \bb@ifunset{bb@@kv@#2}%
1685 {\bb@csarg\gdef{#1\language}{#3}}%
1686 {\expandafter\ifx\csname bb@@kv@#2\endcsname\empty
1687 \bb@csarg\gdef{#1\language}{#3}}%
1688 \else
1689 \bb@exp{\global\let\<bb@#1\language>\<bb@@kv@#2>}%
1690 \fi}

```

Key-value pairs are treated differently depending on the section in the ini file. The following macros are the readers for identification and typography.

```

1691 \let\bb@secline@identification\bb@inikv
1692 \def\bb@secpost@identification{%
1693 \bb@exportkey{lname}{identification.name.english}{}}%
1694 \bb@exportkey{lbcpr}{identification.tag.bcp47}{}}%
1695 \bb@exportkey{lotf}{identification.tag.opentype}{dflt}%
1696 \bb@exportkey{sname}{identification.script.name}{}}%
1697 \bb@exportkey{sbcpr}{identification.script.tag.bcp47}{}}%
1698 \bb@exportkey{sotf}{identification.script.tag.opentype}{DFLT}}
1699 \let\bb@secline@typography\bb@inikv
1700 \def\bb@after@ini{%
1701 \bb@exportkey{lftm}{typography.lefthyphenmin}{2}}%
1702 \bb@exportkey{rgtm}{typography.righthyphenmin}{3}}%
1703 \bb@exportkey{hyphr}{typography.hyphenrules}{}}%
1704 \def\bb@tempa{0.9}%
1705 \bb@csarg\ifx{@kv@identification.version}\bb@tempa
1706 \bb@warning{%
1707 The '\language' date format may not be suitable\\%
1708 for proper typesetting, and therefore it very likely will\\%
1709 change in a future release. Reported}%
1710 \fi
1711 \bb@toglobal\bb@savetoday
1712 \bb@toglobal\bb@savestate}

```

Now captions and captions.licr, depending on the engine. And also for dates. They rely on a few auxiliary macros.

```

1713 \ifcase\bb@engine
1714 \bb@csarg\def{secline@captions.licr}#1=#2\@@{%
1715 \bb@ini@captions@aux{#1}{#2}}
1716 \bb@csarg\def{secline@date.gregorian}#1=#2\@@{% for defaults
1717 \bb@ini@dategreg#1...\relax{#2}}
1718 \bb@csarg\def{secline@date.gregorian.licr}#1=#2\@@{% override
1719 \bb@ini@dategreg#1...\relax{#2}}

```

```

1720 \else
1721   \def\bbl@secline@captions#1=#2\@{%
1722     \bbl@ini@captions@aux{#1}{#2}}
1723   \bbl@csarg\def{secline@date.gregorian}#1=#2\@{%
1724     \bbl@ini@dategreg#1...\relax{#2}}
1725 \fi

```

The auxiliary macro for captions define \<caption>name.

```

1726 \def\bbl@ini@captions@aux#1#2{%
1727   \bbl@trim@def\bbl@tempa{#1}%
1728   \bbl@ifblank{#2}%
1729   {\bbl@exp{%
1730     \toks@{\bbl@nocaption{\bbl@tempa}\<\language name\bbl@tempa name>}}}%
1731   {\bbl@trim\toks@{#2}}}%
1732   \bbl@exp{%
1733     \bbl@add\bbl@savestrings{%
1734       \SetString\<\bbl@tempa name>{\the\toks@}}}%

```

But dates are more complex. The full date format is stores in date.gregorian, so we must read it in non-Unicode engines, too.

```

1735 \bbl@csarg\def{secpre@date.gregorian.licr}{%
1736   \ifcase\bbl@engine\let\bbl@savestate\empty\fi}
1737 \def\bbl@ini@dategreg#1.#2.#3.#4\relax#5{% TODO - ignore with 'captions'
1738   \bbl@trim@def\bbl@tempa{#1.#2}%
1739   \bbl@ifsamestring{\bbl@tempa}{months.wide}%
1740   {\bbl@trim@def\bbl@tempa{#3}%
1741     \bbl@trim\toks@{#5}%
1742     \bbl@exp{%
1743       \bbl@add\bbl@savestate{%
1744         \SetString\<month\romannumeral\bbl@tempa name>{\the\toks@}}}%
1745     {\bbl@ifsamestring{\bbl@tempa}{date.long}%
1746       {\bbl@trim@def\bbl@toreplace{#5}%
1747         \bbl@TG@date
1748         \global\bbl@csarg\let{date@\language name}\bbl@toreplace
1749         \bbl@exp{%
1750           \gdef\<\language name date>####1####2####3{%
1751             \<\bbl@ensure@\language name>{%
1752               \<\bbl@date@\language name>{####1}{####2}{####3}}}%
1753             \bbl@add\bbl@savetoday{%
1754               \SetString\<\today{%
1755                 \<\language name date>{\year}{\month}{\day}}}}}%
1756           {}%

```

Dates will require some macros for the basic formatting. They may be redefined by language, so “semi-public” names (camel case) are used. Oddly enough, the CLDR places particles like “de” inconsistently in either in the date or in the month name.

```

1757 \newcommand\BabelDateSpace{\nobreakspace}
1758 \newcommand\BabelDateDot{.\@}
1759 \newcommand\BabelDated[1]{\number#1}
1760 \newcommand\BabelDatedd[1]{\ifnum#1<10 0\fi\number#1}
1761 \newcommand\BabelDateM[1]{\number#1}
1762 \newcommand\BabelDateMM[1]{\ifnum#1<10 0\fi\number#1}
1763 \newcommand\BabelDateMMMM[1]{%
1764   \csname month\romannumeral\month name\endcsname}%
1765 \newcommand\BabelDatey[1]{\number#1}%
1766 \newcommand\BabelDateyy[1]{%
1767   \ifnum#1<10 0\number#1 %
1768   \else\ifnum#1<100 \number#1 %
1769   \else\ifnum#1<1000 \expandafter\@gobble\number#1 %

```

```

1770 \else\ifnum#1<10000 \expandafter\@gobbletwo\number#1 %
1771 \else
1772   \bbl@error
1773   {Currently two-digit years are restricted to the\
1774    range 0-9999.}%
1775   {There is little you can do. Sorry.}%
1776 \fi\fi\fi\fi}}
1777 \newcommand\BabelDateyyyy[1]{\number#1}%
1778 \def\bbl@replace@finish@iii#1{%
1779   \bbl@exp{\def\#1####1####2####3{\the\toks@}}%
1780 \def\bbl@TG@date{%
1781   \bbl@replace\bbl@toreplace{[ ]}{\BabelDateSpace{}}%
1782   \bbl@replace\bbl@toreplace{[.]}{\BabelDateDot{}}%
1783   \bbl@replace\bbl@toreplace{[d]}{\BabelDateD{####3}}%
1784   \bbl@replace\bbl@toreplace{[dd]}{\BabelDateDd{####3}}%
1785   \bbl@replace\bbl@toreplace{[M]}{\BabelDateM{####2}}%
1786   \bbl@replace\bbl@toreplace{[MM]}{\BabelDateMM{####2}}%
1787   \bbl@replace\bbl@toreplace{[MMM]}{\BabelDateMMM{####2}}%
1788   \bbl@replace\bbl@toreplace{[y]}{\BabelDatey{####1}}%
1789   \bbl@replace\bbl@toreplace{[yy]}{\BabelDateyy{####1}}%
1790   \bbl@replace\bbl@toreplace{[yyyy]}{\BabelDateyyyy{####1}}%
1791 % Note after \bbl@replace \toks@ contains the resulting string.
1792 % TODO - Using this implicit behavior doesn't seem a good idea.
1793   \bbl@replace@finish@iii\bbl@toreplace}

```

Language and Script values to be used when defining a font or setting the direction are set with the following macros.

```

1794 \def\bbl@provide@lsys#1{%
1795   \bbl@ifunset{bbl@lname@#1}%
1796   {\bbl@ini@ids{#1}}%
1797   }%
1798   \bbl@csarg\let{lsys@#1}\@empty
1799   \bbl@ifunset{bbl@sname@#1}{\bbl@csarg\gdef{sname@#1}{Default}}{}%
1800   \bbl@ifunset{bbl@sotf@#1}{\bbl@csarg\gdef{sotf@#1}{DFLT}}{}%
1801   \bbl@csarg\bbl@add@list{lsys@#1}{Script=\bbl@cs{sname@#1}}%
1802   \bbl@ifunset{bbl@lname@#1}{}%
1803   {\bbl@csarg\bbl@add@list{lsys@#1}{Language=\bbl@cs{lname@#1}}}%
1804   \bbl@csarg\bbl@toGLOBAL{lsys@#1}%
1805 % \bbl@exp{% TODO - should be global
1806 %   \<keys_if_exist:nnF>{fontspec-opentype/Script}{\bbl@cs{sname@#1}}%
1807 %   {\newfontscript{\bbl@cs{sname@#1}}{\bbl@cs{sotf@#1}}}%
1808 %   \<keys_if_exist:nnF>{fontspec-opentype/Language}{\bbl@cs{lname@#1}}%
1809 %   {\newfontlanguage{\bbl@cs{lname@#1}}{\bbl@cs{lotf@#1}}}%

```

The following ini reader ignores everything but the identification section. It is called when a font is defined (ie, when the language is first selected) to know which script/language must be enabled. This means we must make sure a few characters are not active. The ini is not read directly, but with a proxy tex file named as the language.

```

1810 \def\bbl@ini@ids#1{%
1811   \def\BabelBeforeIni###1##2{%
1812     \begingroup
1813     \bbl@add\bbl@secpost@identification{%
1814       \def\bbl@inline#####1\bbl@inline{}}%
1815       \catcode`\[=12 \catcode`\]=12 \catcode`\=12
1816       \bbl@read@ini{##1}%
1817     \endgroup}
1818   \InputIfFileExists{babel-#1.tex}{}{}

```

9.3 Cross referencing macros

The \LaTeX book states:

The *key* argument is any sequence of letters, digits, and punctuation symbols; upper- and lowercase letters are regarded as different.

When the above quote should still be true when a document is typeset in a language that has active characters, special care has to be taken of the category codes of these characters when they appear in an argument of the cross referencing macros.

When a cross referencing command processes its argument, all tokens in this argument should be character tokens with category ‘letter’ or ‘other’.

The only way to accomplish this in most cases is to use the trick described in the \TeX book [2] (Appendix D, page 382). The primitive `\meaning` applied to a token expands to the current meaning of this token. For example, ‘`\meaning\A`’ with `\A` defined as ‘`\def\A#1{\B}`’ expands to the characters ‘`macro:#1->\B`’ with all category codes set to ‘other’ or ‘space’.

`\newlabel` The macro `\label` writes a line with a `\newlabel` command into the `.aux` file to define labels.

```
1819 %\bbl@redefine\newlabel#1#2{%
1820 % \@safe@activestrue\org@newlabel{#1}{#2}\@safe@activesfalse}
```

`\@newl@bel` We need to change the definition of the \LaTeX -internal macro `\@newl@bel`. This is needed because we need to make sure that shorthand characters expand to their non-active version.

The following package options control which macros are to be redefined.

```
1821 <<*More package options>> ≡
1822 \DeclareOption{safe=none}{\let\bbl@opt@safe\@empty}
1823 \DeclareOption{safe=bib}{\def\bbl@opt@safe{B}}
1824 \DeclareOption{safe=ref}{\def\bbl@opt@safe{R}}
1825 <</More package options>>
```

First we open a new group to keep the changed setting of `\protect` local and then we set the `@safe@actives` switch to true to make sure that any shorthand that appears in any of the arguments immediately expands to its non-active self.

```
1826 \ifx\bbl@opt@safe\@empty\else
1827 \def\@newl@bel#1#2#3{%
1828 {\@safe@activestrue
1829 \bbl@ifunset{#1#2}%
1830 \relax
1831 {\gdef\@multiplelabels{%
1832 \@latex@warning@no@line{There were multiply-defined labels}}%
1833 \@latex@warning@no@line{Label `#2' multiply defined}}%
1834 \global\@namedef{#1#2}{#3}}}
```

`\@testdef` An internal \LaTeX macro used to test if the labels that have been written on the `.aux` file have changed. It is called by the `\enddocument` macro. This macro needs to be completely rewritten, using `\meaning`. The reason for this is that in some cases the expansion of `\#1@#2` contains the same characters as the `#3`; but the character codes differ. Therefore \LaTeX keeps reporting that the labels may have changed.

```
1835 \CheckCommand*\@testdef[3]{%
1836 \def\reserved@a{#3}%
1837 \expandafter\ifx\csname#1#2\endcsname\reserved@a
1838 \else
1839 \@tempswatrue
1840 \fi}
```

Now that we made sure that `\@testdef` still has the same definition we can rewrite it. First we make the shorthands ‘safe’.

```
1841 \def\@testdef#1#2#3{%
1842   \@safe@activestru
```

Then we use `\bbl@tempa` as an ‘alias’ for the macro that contains the label which is being checked.

```
1843   \expandafter\let\expandafter\bbl@tempa\csname #1@#2\endcsname
```

Then we define `\bbl@tempb` just as `\@newl@bel` does it.

```
1844 \def\bbl@tempb{#3}%
1845 \@safe@activestru
```

When the label is defined we replace the definition of `\bbl@tempa` by its meaning.

```
1846 \ifx\bbl@tempa\relax
1847 \else
1848   \edef\bbl@tempa{\expandafter\strip@prefix\meaning\bbl@tempa}%
1849 \fi
```

We do the same for `\bbl@tempb`.

```
1850   \edef\bbl@tempb{\expandafter\strip@prefix\meaning\bbl@tempb}%
```

If the label didn’t change, `\bbl@tempa` and `\bbl@tempb` should be identical macros.

```
1851 \ifx\bbl@tempa\bbl@tempb
1852 \else
1853   \@tempwatru
1854 \fi}
1855 \fi
```

`\ref` The same holds for the macro `\ref` that references a label and `\pageref` to reference a page. So we redefine `\ref` and `\pageref`. While we change these macros, we make them robust as well (if they weren’t already) to prevent problems if they should become expanded at the wrong moment.

```
1856 \bbl@xin@{R}\bbl@opt@safe
1857 \ifin@
1858   \bbl@redefinero\ref#1{%
1859     \@safe@activestru\org@ref{#1}\@safe@activestru}
1860   \bbl@redefinero\pageref#1{%
1861     \@safe@activestru\org@pageref{#1}\@safe@activestru}
1862 \else
1863   \let\org@ref\ref
1864   \let\org@pageref\pageref
1865 \fi
```

`\@citex` The macro used to cite from a bibliography, `\cite`, uses an internal macro, `\@citex`. It is this internal macro that picks up the argument(s), so we redefine this internal macro and leave `\cite` alone. The first argument is used for typesetting, so the shorthands need only be deactivated in the second argument.

```
1866 \bbl@xin@{B}\bbl@opt@safe
1867 \ifin@
1868   \bbl@redefine\@citex[#1]#2{%
1869     \@safe@activestru\edef\@tempa{#2}\@safe@activestru}
1870     \org@citex[#1]{\@tempa}}
```

Unfortunately, the packages `natbib` and `cite` need a different definition of `\@citex`... To begin with, `natbib` has a definition for `\@citex` with *three* arguments... We only know that a package is loaded when `\begin{document}` is executed, so we need to postpone the different redefinition.

```

1871 \AtBeginDocument{%
1872   \ifpackageloaded{natbib}{%

```

Notice that we use `\def` here instead of `\bbl@redefine` because `\org@citex` is already defined and we don't want to overwrite that definition (it would result in parameter stack overflow because of a circular definition).

(Recent versions of `natbib` change dynamically `\@citex`, so PR4087 doesn't seem fixable in a simple way. Just load `natbib` before.)

```

1873   \def\@citex[#1][#2]#3{%
1874     \@safe@activetrue\edef\@tempa{#3}\@safe@activetruefalse
1875     \org@citex[#1][#2]{\@tempa}}%
1876   }{}

```

The package `cite` has a definition of `\@citex` where the shorthands need to be turned off in both arguments.

```

1877 \AtBeginDocument{%
1878   \ifpackageloaded{cite}{%
1879     \def\@citex[#1]#2{%
1880       \@safe@activetrue\org@citex[#1][#2]\@safe@activetruefalse}%
1881     }{}

```

`\nocite` The macro `\nocite` which is used to instruct BiB_T_EX to extract uncited references from the database.

```

1882   \bbl@redefine\nocite#1{%
1883     \@safe@activetrue\org@nocite{#1}\@safe@activetruefalse}

```

`\bibcite` The macro that is used in the `.aux` file to define citation labels. When packages such as `natbib` or `cite` are not loaded its second argument is used to typeset the citation label. In that case, this second argument can contain active characters but is used in an environment where `\@safe@activetrue` is in effect. This switch needs to be reset inside the `\hbox` which contains the citation label. In order to determine during `.aux` file processing which definition of `\bibcite` is needed we define `\bibcite` in such a way that it redefines itself with the proper definition.

```

1884   \bbl@redefine\bibcite{%

```

We call `\bbl@cite@choice` to select the proper definition for `\bibcite`. This new definition is then activated.

```

1885     \bbl@cite@choice
1886     \bibcite}

```

`\bbl@bibcite` The macro `\bbl@bibcite` holds the definition of `\bibcite` needed when neither `natbib` nor `cite` is loaded.

```

1887   \def\bbl@bibcite#1#2{%
1888     \org@bibcite{#1}{\@safe@activetruefalse#2}}

```

`\bbl@cite@choice` The macro `\bbl@cite@choice` determines which definition of `\bibcite` is needed.

```

1889   \def\bbl@cite@choice{%

```

First we give `\bibcite` its default definition.

```

1890     \global\let\bibcite\bbl@bibcite

```

Then, when `natbib` is loaded we restore the original definition of `\bibcite`.

```

1891     \ifpackageloaded{natbib}{\global\let\bibcite\org@bibcite}}%

```

For `cite` we do the same.

```

1892     \@ifpackageloaded{cite}{\global\let\bibcite\org@bibcite}}%

```


Make sure this only happens once.

```
1893 \global\let\bbl@cite@choice\relax}
```

When a document is run for the first time, no .aux file is available, and \bbl@cite will not yet be properly defined. In this case, this has to happen before the document starts.

```
1894 \AtBeginDocument{\bbl@cite@choice}
```

\@bibitem One of the two internal L^AT_EX macros called by \bibitem that write the citation label on the .aux file.

```
1895 \bbl@redefine\@bibitem#1{%
1896 \@safe@activetrue\org@@bibitem{#1}\@safe@activesfalse}
1897 \else
1898 \let\org@nocite\nocite
1899 \let\org@@citex\citex
1900 \let\org@bbl@cite\bbl@cite
1901 \let\org@@bibitem\@bibitem
1902 \fi
```

9.4 Marks

\markright Because the output routine is asynchronous, we must pass the current language attribute to the head lines, together with the text that is put into them. To achieve this we need to adapt the definition of \markright and \markboth somewhat. We check whether the argument is empty; if it is, we just make sure the scratch token register is empty. Next, we store the argument to \markright in the scratch token register. This way these commands will not be expanded later, and we make sure that the text is typeset using the correct language settings. While doing so, we make sure that active characters that may end up in the mark are not disabled by the output routine kicking in while \@safe@activetrue is in effect.

```
1903 \bbl@redefine\markright#1{%
1904 \bbl@ifblank{#1}%
1905 {\org@markright{}}%
1906 {\toks@{#1}%
1907 \bbl@exp{%
1908 \org@markright{\protect\foreignlanguage{\language}\language}%
1909 {\protect\bbl@restore@actives\the\toks@}}}%
```

\markboth The definition of \markboth is equivalent to that of \markright, except that we need two token registers. The documentclasses report and book define and set the headings for the page. While doing so they also store a copy of \markboth in \@mkboth. Therefore we need to check whether \@mkboth has already been set. If so we need to do that again with the new definition of \markboth.

```
1910 \ifx\@mkboth\markboth
1911 \def\bbl@tempc{\let\@mkboth\markboth}
1912 \else
1913 \def\bbl@tempc{}
1914 \fi
```

Now we can start the new definition of \markboth

```
1915 \bbl@redefine\markboth#1#2{%
1916 \protected@edef\bbl@tempb##1{%
1917 \protect\foreignlanguage{\language}\language}{\protect\bbl@restore@actives##1}}%
1918 \bbl@ifblank{#1}%
1919 {\toks@{}}%
1920 {\toks@\expandafter{\bbl@tempb{#1}}}%
1921 \bbl@ifblank{#2}%
```

```

1922   {\temptokena{}}}%
1923   {\temptokena\expandafter{\bbl@tempb{#2}}}%
1924   \bbl@exp{\org@markboth{\the\toks@}{\the\temptokena}}

```

and copy it to \mkboth if necessary.

```
1925 \bbl@tempc
```

9.5 Preventing clashes with other packages

9.5.1 ifthen

\ifthenelse Sometimes a document writer wants to create a special effect depending on the page a certain fragment of text appears on. This can be achieved by the following piece of code:

```

\ifthenelse{\isodd{\pageref{some:label}}}{
  {code for odd pages}
  {code for even pages}
}

```

In order for this to work the argument of \isodd needs to be fully expandable. With the above redefinition of \pageref it is not in the case of this example. To overcome that, we add some code to the definition of \ifthenelse to make things work.

The first thing we need to do is check if the package ifthen is loaded. This should be done at \begin{document} time.

```

1926 \bbl@xin@{R}\bbl@opt@safe
1927 \ifin@
1928   \AtBeginDocument{%
1929     \ifpackageloaded{ifthen}{%

```

Then we can redefine \ifthenelse:

```
1930   \bbl@redefine@long\ifthenelse#1#2#3{%
```

We want to revert the definition of \pageref and \ref to their original definition for the first argument of \ifthenelse, so we first need to store their current meanings.

```

1931     \let\bbl@temp@pref\pageref
1932     \let\pageref\org@pageref
1933     \let\bbl@temp@ref\ref
1934     \let\ref\org@ref

```

Then we can set the \@safe@actives switch and call the original \ifthenelse. In order to be able to use shorthands in the second and third arguments of \ifthenelse the resetting of the switch *and* the definition of \pageref happens inside those arguments. When the package wasn't loaded we do nothing.

```

1935     \@safe@activetrue
1936     \org@ifthenelse{#1}%
1937     {\let\pageref\bbl@temp@pref
1938      \let\ref\bbl@temp@ref
1939      \@safe@activesfalse
1940      #2}%
1941     {\let\pageref\bbl@temp@pref
1942      \let\ref\bbl@temp@ref
1943      \@safe@activesfalse
1944      #3}%
1945   }%
1946 }{}%
1947 }

```

9.5.2 varioref

`\@@vpageref` When the package `varioref` is in use we need to modify its internal command `\@@vpageref` in order to prevent problems when an active character ends up in the argument of `\vref`.

```
\vrefpagemum
\Ref
1948 \AtBeginDocument{%
1949   \ifpackageloaded{varioref}{%
1950     \bbl@redefine\@@vpageref#1[#2]#3{%
1951       \@safe@activetrue
1952       \org@@vpageref{#1}[#2]#3}%
1953     \@safe@activesfalse}%
```

The same needs to happen for `\vrefpagemum`.

```
1954   \bbl@redefine\vrefpagemum#1#2{%
1955     \@safe@activetrue
1956     \org@vrefpagemum{#1}#2}%
1957   \@safe@activesfalse}%
```

The package `varioref` defines `\Ref` to be a robust command which uppercases the first character of the reference text. In order to be able to do that it needs to access the expandable form of `\ref`. So we employ a little trick here. We redefine the (internal) command `\Ref` to call `\org@ref` instead of `\ref`. The disadvantage of this solution is that whenever the definition of `\Ref` changes, this definition needs to be updated as well.

```
1958   \expandafter\def\csname Ref\endcsname#1{%
1959     \protected@edef\@tempa{\org@ref{#1}}\expandafter\MakeUppercase\@tempa}
1960   }{}%
1961 }
1962 \fi
```

9.5.3 hhlne

`\hhlne` Delaying the activation of the shorthand characters has introduced a problem with the `hhlne` package. The reason is that it uses the ‘:’ character which is made active by the french support in `babel`. Therefore we need to *reload* the package when the ‘:’ is an active character.

So at `\begin{document}` we check whether `hhlne` is loaded.

```
1963 \AtEndOfPackage{%
1964   \AtBeginDocument{%
1965     \ifpackageloaded{hhlne}%
```

Then we check whether the expansion of `\normal@char:` is not equal to `\relax`.

```
1966     {\expandafter\ifx\csname normal@char:string\endcsname\relax
1967       \else
```

In that case we simply reload the package. Note that this happens *after* the category code of the `@`-sign has been changed to other, so we need to temporarily change it to letter again.

```
1968       \makeatletter
1969       \def\@currname{hhlne}\input{hhlne.sty}\makeatother
1970     \fi}%
1971   {}}}
```

9.5.4 hyperref

`\pdfstringdefDisableCommands` A number of interworking problems between `babel` and `hyperref` are tackled by `hyperref` itself. The following code was introduced to prevent some annoying warnings but it broke bookmarks. This was quickly fixed in `hyperref`, which essentially made it no-op. However, it will not be removed for the moment because `hyperref` is expecting it.

```

1972 \AtBeginDocument{%
1973   \ifx\pdfstringdefDisableCommands\@undefined\else
1974     \pdfstringdefDisableCommands{\languageshorthands{system}}%
1975   \fi}

```

9.5.5 fancyhdr

`\FOREIGNLANGUAGE` The package `fancyhdr` treats the running head and foot lines somewhat differently as the standard classes. A symptom of this is that the command `\foreignlanguage` which `babel` adds to the marks can end up inside the argument of `\MakeUppercase`. To prevent unexpected results we need to define `\FOREIGNLANGUAGE` here.

```

1976 \DeclareRobustCommand{\FOREIGNLANGUAGE}[1]{%
1977   \lowercase{\foreignlanguage{#1}}}

```

`\substitutefontfamily` The command `\substitutefontfamily` creates an `.fd` file on the fly. The first argument is an encoding mnemonic, the second and third arguments are font family names.

```

1978 \def\substitutefontfamily#1#2#3{%
1979   \lowercase{\immediate\openout15=#1#2.fd\relax}%
1980   \immediate\write15{%
1981     \string\ProvidesFile{#1#2.fd}%
1982     [\the\year/\two@digits{\the\month}/\two@digits{\the\day}
1983     \space generated font description file]^{}
1984     \string\DeclareFontFamily{#1}{#2}{^{}
1985     \string\DeclareFontShape{#1}{#2}{m}{n}{<->ssub * #3/m/n}{^{}
1986     \string\DeclareFontShape{#1}{#2}{m}{it}{<->ssub * #3/m/it}{^{}
1987     \string\DeclareFontShape{#1}{#2}{m}{sl}{<->ssub * #3/m/sl}{^{}
1988     \string\DeclareFontShape{#1}{#2}{m}{sc}{<->ssub * #3/m/sc}{^{}
1989     \string\DeclareFontShape{#1}{#2}{b}{n}{<->ssub * #3/bx/n}{^{}
1990     \string\DeclareFontShape{#1}{#2}{b}{it}{<->ssub * #3/bx/it}{^{}
1991     \string\DeclareFontShape{#1}{#2}{b}{sl}{<->ssub * #3/bx/sl}{^{}
1992     \string\DeclareFontShape{#1}{#2}{b}{sc}{<->ssub * #3/bx/sc}{^{}
1993   }%
1994   \closeout15
1995 }

```

This command should only be used in the preamble of a document.

```

1996 \@onlypreamble\substitutefontfamily

```

9.6 Encoding and fonts

Because documents may use non-ASCII font encodings, we make sure that the logos of \TeX and \LaTeX always come out in the right encoding. There is a list of non-ASCII encodings. Unfortunately, `fontenc` deletes its package options, so we must guess which encodings has been loaded by traversing `\@filelist` to search for `\enc enc.def`. If a non-ASCII has been loaded, we define versions of `\TeX` and `\LaTeX` for them using `\ensureascii`. The default ASCII encoding is `set`, too (in reverse order): the “main” encoding (when the document begins), the last loaded, or `OT1`.

```

\ensureascii

```

```

1997 \newcommand\BabelNonASCII{LGR,X2,OT2,OT3,OT6,LHE,LWN,LMA,LMC,LMS,LMU,}
1998 \let\org@TeX\TeX
1999 \let\org@LaTeX\LaTeX
2000 \let\ensureascii\@firstofone
2001 \AtBeginDocument{%
2002   \in@false
2003   \bbl@foreach\BabelNonASCII{% is there a non-ascii enc?

```

```

2004 \ifin@ \else
2005 \lowercase{\bbl@xin@{, #1 enc.def, }{, \@filelist, }}%
2006 \fi}%
2007 \ifin@ % if a non-ascii has been loaded
2008 \def\ensureasciic#1{{\fontencoding{OT1}\selectfont#1}}%
2009 \DeclareTextCommandDefault{\TeX}{\org@TeX}%
2010 \DeclareTextCommandDefault{\LaTeX}{\org@LaTeX}%
2011 \def\bbl@tempb#1\@@{\uppercase{\bbl@tempc#1}ENC.DEF\@empty\@@}%
2012 \def\bbl@tempc#1ENC.DEF#2\@@{%
2013 \ifx\@empty#2\else
2014 \bbl@ifunset{T#1}%
2015 {}%
2016 {\bbl@xin@{, #1, }{, \BabelNonASCII, }}%
2017 \ifin@
2018 \DeclareTextCommand{\TeX}{#1}{\ensureasciic{\org@TeX}}%
2019 \DeclareTextCommand{\LaTeX}{#1}{\ensureasciic{\org@LaTeX}}%
2020 \else
2021 \def\ensureasciic##1{{\fontencoding{#1}\selectfont##1}}%
2022 \fi}%
2023 \fi}%
2024 \bbl@foreach\@filelist{\bbl@tempb#1\@@}% TODO - \@@ de mas??
2025 \bbl@xin@{, \cf@encoding, }{, \BabelNonASCII, }}%
2026 \ifin@ \else
2027 \edef\ensureasciic#1{%
2028 \noexpand\fontencoding{\cf@encoding}\noexpand\selectfont#1}}%
2029 \fi
2030 \fi}

```

Now comes the old deprecated stuff (with a little change in 3.9l, for fontspec). The first thing we need to do is to determine, at `\begin{document}`, which latin fontencoding to use.

`\latinencoding` When text is being typeset in an encoding other than ‘latin’ (OT1 or T1), it would be nice to still have Roman numerals come out in the Latin encoding. So we first assume that the current encoding at the end of processing the package is the Latin encoding.

```

2031 \AtEndOfPackage{\edef\latinencoding{\cf@encoding}}

```

But this might be overruled with a later loading of the package fontenc. Therefore we check at the execution of `\begin{document}` whether it was loaded with the T1 option. The normal way to do this (using `\@ifpackageloaded`) is disabled for this package. Now we have to revert to parsing the internal macro `\@filelist` which contains all the filenames loaded.

```

2032 \AtBeginDocument{%
2033 \ifpackageloaded{fontspec}%
2034 {\xdef\latinencoding{%
2035 \ifx\UTFencname\@undefined
2036 EU\ifcase\bbl@engine\or2\or1\fi
2037 \else
2038 \UTFencname
2039 \fi}}%
2040 {\gdef\latinencoding{OT1}%
2041 \ifx\cf@encoding\bbl@t@one
2042 \xdef\latinencoding{\bbl@t@one}%
2043 \else
2044 \@ifl@aded{def}{t1enc}{\xdef\latinencoding{\bbl@t@one}}}%
2045 \fi}}

```

`\latintext` Then we can define the command `\latintext` which is a declarative switch to a latin font-encoding. Usage of this macro is deprecated.

```

2046 \DeclareRobustCommand{\latintext}{%
2047   \fontencoding{\latinencoding}\selectfont
2048   \def\encodingdefault{\latinencoding}}

```

`\textlatin` This command takes an argument which is then typeset using the requested font encoding. In order to avoid many encoding switches it operates in a local scope.

```

2049 \ifx\@undefined\DeclareTextFontCommand
2050   \DeclareRobustCommand{\textlatin}[1]{\leavevmode{\latintext #1}}
2051 \else
2052   \DeclareTextFontCommand{\textlatin}{\latintext}
2053 \fi

```

9.7 Basic bidi support

Work in progress. This code is currently placed here for practical reasons.

- pdftex provides a minimal support for bidi text, and it must be done by hand. Vertical typesetting is not possible.
- xetex is somewhat better, thanks to its font engine (even if not always reliable) and a few additional tools. However, very little is done at the paragraph level. Another challenging problem is text direction does not honour T_EX grouping.
- luatex can provide the most complete solution, as we can manipulate almost freely the node list, the generated lines, and so on, but bidi text does not work out of the box and some development is necessary. It also provides tools to properly set left-to-right and right-to-left page layouts. As LuaT_EX-ja shows, vertical typesetting is possible, too. Its main drawback is font handling is often considered to be less mature than xetex.²⁹

```

2054 \def\bbl@alscripts{,Arabic,Syriac,Thaana,}
2055 \def\bbl@rscripts{%
2056   ,Imperial Aramaic,Avestan,Cypriot,Hatran,Hebrew,%
2057   Old Hungarian,Old Hungarian,Lydian,Mandaean,Manichaeen,%
2058   Manichaeen,Meroitic Cursive,Meroitic,Old North Arabian,%
2059   Nabataean,N'Ko,Orkhon,Palmyrene,Inscriptional Pahlavi,%
2060   Psalter Pahlavi,Phoenician,Inscriptional Parthian,Samaritan,%
2061   Old South Arabian,}%
2062 \def\bbl@provide@dirs#1{%
2063   \bbl@xin{\csname bbl@sname@#1\endcsname}{\bbl@alscripts\bbl@rscripts}%
2064   \ifin@
2065     \global\bbl@csarg\chardef{wdir@#1}\@ne
2066     \bbl@xin{\csname bbl@sname@#1\endcsname}{\bbl@alscripts}%
2067     \ifin@
2068       \global\bbl@csarg\chardef{wdir@#1}\tw@ % useless in xetex
2069     \fi
2070   \else
2071     \global\bbl@csarg\chardef{wdir@#1}\z@
2072   \fi}
2073 \def\bbl@switchdir{%
2074   \bbl@ifunset{bbl@lsys@\languagename}{\bbl@provide@lsys{\languagename}}{}}%
2075   \bbl@ifunset{bbl@wdir@\languagename}{\bbl@provide@dirs{\languagename}}{}}%
2076   \bbl@exp{\bbl@setdirs\bbl@cs{wdir@\languagename}}}%
2077 \def\bbl@setdirs#1{% TODO - math
2078   \ifcase\bbl@select@type % TODO - strictly, not the right test
2079     \bbl@pagedir{#1}%
2080     \bbl@bodydir{#1}%

```

²⁹Although in my [JBL] experience problems are in fact minimal.

```

2081 \bbl@pardir{#1}%
2082 \fi
2083 \bbl@textdir{#1}}
2084 \ifodd\bbl@engine
2085 \AddBabelHook{babel-bidi}{afterextras}{\bbl@switchdir}
2086 \DisableBabelHook{babel-bidi}
2087 \def\bbl@getluadir#1{%
2088 \directlua{
2089 if tex.#1dir == 'TLT' then
2090 tex.sprint('0')
2091 elseif tex.#1dir == 'TRT' then
2092 tex.sprint('1')
2093 end}}
2094 \def\bbl@setdir#1#2#3{% 1=text/par.. 2=\textdir.. 3=0 lr/1 rl
2095 \ifcase#3\relax
2096 \ifcase\bbl@getluadir{#1}\relax\else
2097 #2 TLT\relax
2098 \fi
2099 \else
2100 \ifcase\bbl@getluadir{#1}\relax
2101 #2 TRT\relax
2102 \fi
2103 \fi}
2104 \def\bbl@textdir#1{%
2105 \bbl@setdir{text}\textdir{#1}% TODO - ?\linedir
2106 \setattribute\bbl@attr@dir{#1}}
2107 \def\bbl@pardir{\bbl@setdir{par}\pardir}
2108 \def\bbl@bodydir{\bbl@setdir{body}\bodydir}
2109 \def\bbl@pagedir{\bbl@setdir{page}\pagedir}
2110 \def\bbl@dirparastext{\pardir\the\textdir\relax}% %%%
2111 \else
2112 \AddBabelHook{babel-bidi}{afterextras}{\bbl@switchdir}
2113 \DisableBabelHook{babel-bidi}
2114 \newcount\bbl@dirlevel
2115 \chardef\bbl@thetextdir\z@
2116 \chardef\bbl@thepardir\z@
2117 \def\bbl@textdir#1{%
2118 \ifcase#1\relax
2119 \chardef\bbl@thetextdir\z@
2120 \bbl@textdir@i\beginL\endL
2121 \else
2122 \chardef\bbl@thetextdir\@ne
2123 \bbl@textdir@i\beginR\endR
2124 \fi}
2125 \def\bbl@textdir@i#1#2{%
2126 \ifhmode
2127 \ifnum\currentgrouplevel>\z@
2128 \ifnum\currentgrouplevel=\bbl@dirlevel
2129 \bbl@error{Multiple bidi settings inside a group}%
2130 {I'll insert a new group, but expect wrong results.}%
2131 \bgroup\aftergroup#2\aftergroup\egroup
2132 \else
2133 \ifcase\currentgroup\or % 0 bottom
2134 \aftergroup#2% 1 simple {}
2135 \or
2136 \bgroup\aftergroup#2\aftergroup\egroup % 2 hbox
2137 \or
2138 \bgroup\aftergroup#2\aftergroup\egroup % 3 adj hbox
2139 \or\or\or % vbox vtop align

```

```

2140      \or
2141      \bgroup\aftergroup#2\aftergroup\egroup % 7 noalign
2142      \or\or\or\or\or\or\or % output math disc insert vcent mathchoice
2143      \or
2144      \aftergroup#2% 14 \begingroup
2145      \else
2146      \bgroup\aftergroup#2\aftergroup\egroup % 15 adj
2147      \fi
2148      \fi
2149      \bbl@dirlevel\currentgrouplevel
2150      \fi
2151      #1%
2152      \fi}
2153      \def\bbl@pardir#1{\chardef\bbl@thepardir#1\relax}
2154      \let\bbl@bodydir@gobble
2155      \let\bbl@pagedir@gobble
2156      \def\bbl@dirparastext{\chardef\bbl@thepardir\bbl@thetextdir}

```

The following command is executed only if there is a right-to-left script (once). It activates the `\everypar` hack for xetex, to properly handle the `par` direction. Note `text` and `par` dirs are decoupled.

```

2157      \def\bbl@xebidipar{%
2158      \let\bbl@xebidipar\relax
2159      \TeXeTstate\@ne
2160      \def\bbl@xeverypar{%
2161      \ifcase\bbl@thepardir\else
2162      {\setbox\z@\lastbox\beginR\box\z@}%
2163      \fi
2164      \ifcase\bbl@thetextdir\else\beginR\fi}%
2165      \let\bbl@severypar\everypar
2166      \newtoks\everypar
2167      \everypar=\bbl@severypar
2168      \bbl@severypar{\bbl@xeverypar\the\everypar}}
2169      \fi

```

9.8 Local Language Configuration

`\loadlocalcfg` At some sites it may be necessary to add site-specific actions to a language definition file. This can be done by creating a file with the same name as the language definition file, but with the extension `.cfg`. For instance the file `nor.sk.cfg` will be loaded when the language definition file `nor.sk.ldf` is loaded.

For plain-based formats we don't want to override the definition of `\loadlocalcfg` from `plain.def`.

```

2170 \ifx\loadlocalcfg\@undefined
2171 \@ifpackagewith{babel}{noconfigs}%
2172 {\let\loadlocalcfg@gobble}%
2173 {\def\loadlocalcfg#1{%
2174 \InputIfFileExists{#1.cfg}%
2175 {\typeout{*****^J%
2176 * Local config file #1.cfg used^^J%
2177 *}}}%
2178 \@empty}}
2179 \fi

```

Just to be compatible with L^AT_EX 2.09 we add a few more lines of code:

```

2180 \ifx\@unexpandable@protect\@undefined
2181 \def\@unexpandable@protect{\noexpand\protect\noexpand}

```



```

2182 \long\def\protected@write#1#2#3{%
2183   \begingroup
2184     \let\thepage\relax
2185     #2%
2186     \let\protect\@unexpandable@protect
2187     \edef\reserved@a{\write#1{#3}}%
2188     \reserved@a
2189   \endgroup
2190   \if@nobreak\ifvmode\nobreak\fi\fi}
2191 \fi
2192 </core>

```

10 Multiple languages (switch.def)

Plain T_EX version 3.0 provides the primitive `\language` that is used to store the current language. When used with a pre-3.0 version this function has to be implemented by allocating a counter.

```

2193 (*kernel)
2194 <<Make sure ProvidesFile is defined>>
2195 \ProvidesFile{switch.def}[\<date>] [\<version>] Babel switching mechanism]
2196 <<Load macros for plain if not LaTeX>>
2197 <<Define core switching macros>>

```

`\adddialect` The macro `\adddialect` can be used to add the name of a dialect or variant language, for which an already defined hyphenation table can be used.

```

2198 \def\bbl@version{\<version>}
2199 \def\bbl@date{\<date>}
2200 \def\adddialect#1#2{%
2201   \global\chardef#1#2\relax
2202   \bbl@usehooks{adddialect}{\#1}{\#2}}%
2203   \wlog{\string#1 = a dialect from \string\language#2}}

```

`\bbl@iflanguage` executes code only if the language `l@` exists. Otherwise raises an error. The argument of `\bbl@fixname` has to be a macro name, as it may get “fixed” if casing (lc/uc) is wrong. It’s intended to fix a long-standing bug when `\foreignlanguage` and the like appear in a `\MakeXXXcase`. However, a lowercase form is not imposed to improve backward compatibility (perhaps you defined a language named MYLANG, but unfortunately mixed case names cannot be trapped). Note `l@` is encapsulated, so that its case does not change.

```

2204 \def\bbl@fixname#1{%
2205   \begingroup
2206     \def\bbl@tempe{l@}%
2207     \edef\bbl@tempd{\noexpand\@ifundefined{\noexpand\bbl@tempe#1}}%
2208     \bbl@tempd
2209     {\lowercase\expandafter{\bbl@tempd}%
2210      {\uppercase\expandafter{\bbl@tempd}%
2211       \@empty
2212       {\edef\bbl@tempd{\def\noexpand#1{#1}}}%
2213       {\uppercase\expandafter{\bbl@tempd}}}%
2214      {\edef\bbl@tempd{\def\noexpand#1{#1}}}%
2215      {\lowercase\expandafter{\bbl@tempd}}}%
2216     \@empty
2217     \edef\bbl@tempd{\endgroup\def\noexpand#1{#1}}%
2218   \bbl@tempd}
2219 \def\bbl@iflanguage#1{%
2220   \@ifundefined{l@#1}{\@nolanerr{#1}\@gobble}\@firstofone}

```

`\iflanguage` Users might want to test (in a private package for instance) which language is currently active. For this we provide a test macro, `\iflanguage`, that has three arguments. It checks whether the first argument is a known language. If so, it compares the first argument with the value of `\language`. Then, depending on the result of the comparison, it executes either the second or the third argument.

```
2221 \def\iflanguage#1{%
2222   \bbl@iflanguage{#1}{%
2223     \ifnum\csname l@#1\endcsname=\language
2224       \expandafter\@firstoftwo
2225     \else
2226       \expandafter\@secondoftwo
2227     \fi}}
```

10.1 Selecting the language

`\selectlanguage` The macro `\selectlanguage` checks whether the language is already defined before it performs its actual task, which is to update `\language` and activate language-specific definitions.

To allow the call of `\selectlanguage` either with a control sequence name or with a simple string as argument, we have to use a trick to delete the optional escape character. To convert a control sequence to a string, we use the `\string` primitive. Next we have to look at the first character of this string and compare it with the escape character. Because this escape character can be changed by setting the internal integer `\escapechar` to a character number, we have to compare this number with the character of the string. To do this we have to use \TeX 's backquote notation to specify the character as a number. If the first character of the `\string`'ed argument is the current escape character, the comparison has stripped this character and the rest in the 'then' part consists of the rest of the control sequence name. Otherwise we know that either the argument is not a control sequence or `\escapechar` is set to a value outside of the character range 0–255. If the user gives an empty argument, we provide a default argument for `\string`. This argument should expand to nothing.

```
2228 \let\bbl@select@type\z@
2229 \edef\selectlanguage{%
2230   \noexpand\protect
2231   \expandafter\noexpand\csname selectlanguage \endcsname}
```

Because the command `\selectlanguage` could be used in a moving argument it expands to `\protect\selectlanguage_`. Therefore, we have to make sure that a macro `\protect` exists. If it doesn't it is `\let` to `\relax`.

```
2232 \ifx\@undefined\protect\let\protect\relax\fi
```

As \LaTeX 2.09 writes to files *expanded* whereas \LaTeX 2_ε takes care *not* to expand the arguments of `\write` statements we need to be a bit clever about the way we add information to .aux files. Therefore we introduce the macro `\xstring` which should expand to the right amount of `\string`'s.

```
2233 \ifx\documentclass\@undefined
2234   \def\xstring{\string\string\string}
2235 \else
2236   \let\xstring\string
2237 \fi
```

Since version 3.5 babel writes entries to the auxiliary files in order to typeset table of contents etc. in the correct language environment.

`\bbl@pop@language` But when the language change happens *inside* a group the end of the group doesn't write anything to the auxiliary files. Therefore we need \TeX 's aftergroup mechanism to help us.

The command `\aftergroup` stores the token immediately following it to be executed when the current group is closed. So we define a temporary control sequence `\bbl@pop@language` to be executed at the end of the group. It calls `\bbl@set@language` with the name of the current language as its argument.

`\bbl@language@stack` The previous solution works for one level of nesting groups, but as soon as more levels are used it is no longer adequate. For that case we need to keep track of the nested languages using a stack mechanism. This stack is called `\bbl@language@stack` and initially empty.

```
2238 \def\bbl@language@stack{}
```

When using a stack we need a mechanism to push an element on the stack and to retrieve the information afterwards.

`\bbl@push@language` The stack is simply a list of languagenames, separated with a '+' sign; the push function can be simple:

```
2239 \def\bbl@push@language{%
```

```
2240 \xdef\bbl@language@stack{\language+\bbl@language@stack}}
```

Retrieving information from the stack is a little bit less simple, as we need to remove the element from the stack while storing it in the macro `\language`. For this we first define a helper function.

`\bbl@pop@lang` This macro stores its first element (which is delimited by the '+'-sign) in `\language` and stores the rest of the string (delimited by '-') in its third argument.

```
2241 \def\bbl@pop@lang#1+#2-#3{%
```

```
2242 \edef\language{#1}\xdef#3{#2}}
```

The reason for the somewhat weird arrangement of arguments to the helper function is the fact it is called in the following way. This means that before `\bbl@pop@lang` is executed \TeX first *expands* the stack, stored in `\bbl@language@stack`. The result of that is that the argument string of `\bbl@pop@lang` contains one or more language names, each followed by a '+'-sign (zero language names won't occur as this macro will only be called after something has been pushed on the stack) followed by the '-'-sign and finally the reference to the stack.

```
2243 \let\bbl@ifrestoring\@secondoftwo
```

```
2244 \def\bbl@pop@language{%
```

```
2245 \expandafter\bbl@pop@lang\bbl@language@stack-\bbl@language@stack
```

```
2246 \let\bbl@ifrestoring\@firstoftwo
```

```
2247 \expandafter\bbl@set@language\expandafter{\language}%
```

```
2248 \let\bbl@ifrestoring\@secondoftwo}
```

Once the name of the previous language is retrieved from the stack, it is fed to `\bbl@set@language` to do the actual work of switching everything that needs switching.

```
2249 \expandafter\def\csname selectlanguage \endcsname#1{%
```

```
2250 \ifnum\bbl@hymapsel=\@cclv\let\bbl@hymapsel\tw@\fi
```

```
2251 \bbl@push@language
```

```
2252 \aftergroup\bbl@pop@language
```

```
2253 \bbl@set@language{#1}}
```

`\bbl@set@language` The macro `\bbl@set@language` takes care of switching the language environment *and* of writing entries on the auxiliary files. For historical reasons, language names can be either language of `\language`. To catch either form a trick is used, but unfortunately as a side effect the catcodes of letters in `\language` are not well defined. The list of auxiliary files can be extended by redefining `\BabelContentsFiles`, but make sure they are loaded inside a group (as `aux`, `toc`, `lof`, and `lot` do) or the last language of the document will remain active afterwards.

We also write a command to change the current language in the auxiliary files.

```

2254 \def\BabelContentsFiles{toc,lof,lot}
2255 \def\bb1@set@language#1{%
2256   \edef\language{%
2257     \ifnum\escapechar=\expandafter`\string#1\@empty
2258     \else\string#1\@empty\fi}%
2259   \select@language{\language}%
2260   \expandafter\ifx\csname date\language\endcsname\relax\else
2261     \if@filesw
2262       \protected@write\@auxout{}\string\select@language{\language}}%
2263       \bb1@foreach\BabelContentsFiles{%
2264         \addtocontents{##1}{\xstring\select@language{\language}}}%
2265       \bb1@usehooks{write}}}%
2266   \fi
2267 \fi}
2268 \def\select@language#1{%
2269   \ifnum\bb1@hymapset=\@cc1v\chardef\bb1@hymapset4\relax\fi
2270   \edef\language{#1}%
2271   \bb1@fixname\language
2272   \bb1@iflanguage\language{%
2273     \expandafter\ifx\csname date\language\endcsname\relax
2274       \bb1@error
2275       {Unknown language `#1'. Either you have\\%
2276        misspelled its name, it has not been installed,\\%
2277        or you requested it in a previous run. Fix its name,\\%
2278        install it or just rerun the file, respectively}%
2279       {You may proceed, but expect wrong results}%
2280     \else
2281       \let\bb1@select@type\z@
2282       \expandafter\bb1@switch\expandafter{\language}%
2283     \fi}}

```

A bit of optimization. Select in heads/foots the language only if necessary. The real thing is in `babel.def`.

```

2284 \let\select@language\x\select@language

```

First, check if the user asks for a known language. If so, update the value of `\language` and call `\originalTeX` to bring \TeX in a certain pre-defined state.

The name of the language is stored in the control sequence `\language`.

Then we have to *redefine* `\originalTeX` to compensate for the things that have been activated. To save memory space for the macro definition of `\originalTeX`, we construct the control sequence name for the `\noextras<lang>` command at definition time by expanding the `\csname` primitive.

Now activate the language-specific definitions. This is done by constructing the names of three macros by concatenating three words with the argument of `\selectlanguage`, and calling these macros.

The switching of the values of `\lefthyphenmin` and `\righthyphenmin` is somewhat different. First we save their current values, then we check if `\<lang>hyphenmins` is defined. If it is not, we set default values (2 and 3), otherwise the values in `\<lang>hyphenmins` will be used.

```

2285 \def\bb1@switch#1{%
2286   \originalTeX
2287   \expandafter\def\expandafter\originalTeX\expandafter{%
2288     \csname noextras#1\endcsname
2289     \let\originalTeX\@empty
2290     \babel@beginsave}%
2291   \bb1@usehooks{afterreset}}}%

```

```

2292 \languageshorthands{none}%
2293 \ifcase\bbbl@select@type
2294   \ifhmode
2295     \hskip\z@skip % trick to ignore spaces
2296     \csname captions#1\endcsname\relax
2297     \csname date#1\endcsname\relax
2298     \loop\ifdim\lastskip>\z@\unskip\repeat\unskip
2299   \else
2300     \csname captions#1\endcsname\relax
2301     \csname date#1\endcsname\relax
2302   \fi
2303 \fi
2304 \bbbl@usehooks{beforeextras}{}%
2305 \csname extras#1\endcsname\relax
2306 \bbbl@usehooks{afterextras}{}%
2307 \ifcase\bbbl@opt@hyphenmap\or
2308   \def\BabelLower##1##2{\lccode##1=##2\relax}%
2309   \ifnum\bbbl@hymapsel>4\else
2310     \csname\language\language @bbbl@hyphenmap\endcsname
2311   \fi
2312   \chardef\bbbl@opt@hyphenmap\z@
2313 \else
2314   \ifnum\bbbl@hymapsel>\bbbl@opt@hyphenmap\else
2315     \csname\language\language @bbbl@hyphenmap\endcsname
2316   \fi
2317 \fi
2318 \global\let\bbbl@hymapsel\@cclv
2319 \bbbl@patterns{#1}%
2320 \babel@savevariable\lefthyphenmin
2321 \babel@savevariable\righthyphenmin
2322 \expandafter\ifx\csname #1hyphenmins\endcsname\relax
2323   \set@hyphenmins\tw@\thr@\relax
2324 \else
2325   \expandafter\expandafter\expandafter\set@hyphenmins
2326   \csname #1hyphenmins\endcsname\relax
2327 \fi}

```

otherlanguage The other language environment can be used as an alternative to using the `\selectlanguage` declarative command. When you are typesetting a document which mixes left-to-right and right-to-left typesetting you have to use this environment in order to let things work as you expect them to. The `\ignorespaces` command is necessary to hide the environment when it is entered in horizontal mode.

```

2328 \long\def\otherlanguage#1{%
2329   \ifnum\bbbl@hymapsel=\@cclv\let\bbbl@hymapsel\thr@\fi
2330   \csname selectlanguage \endcsname{#1}%
2331   \ignorespaces}

```

The `\endotherlanguage` part of the environment tries to hide itself when it is called in horizontal mode.

```

2332 \long\def\endotherlanguage{%
2333   \global\@ignoretrue\ignorespaces}

```

otherlanguage* The other language environment is meant to be used when a large part of text from a different language needs to be typeset, but without changing the translation of words such as ‘figure’. This environment makes use of `\foreign@language`.

```

2334 \expandafter\def\csname otherlanguage*\endcsname#1{%
2335   \ifnum\bbbl@hymapsel=\@cclv\chardef\bbbl@hymapsel4\relax\fi

```

2336 \foreign@language{#1}}

At the end of the environment we need to switch off the extra definitions. The grouping mechanism of the environment will take care of resetting the correct hyphenation rules and “extras”.

2337 \expandafter\let\csname endotherlanguage*\endcsname\relax

\foreignlanguage The \foreignlanguage command is another substitute for the \selectlanguage command. This command takes two arguments, the first argument is the name of the language to use for typesetting the text specified in the second argument. Unlike \selectlanguage this command doesn’t switch *everything*, it only switches the hyphenation rules and the extra definitions for the language specified. It does this within a group and assumes the \extras⟨*lang*⟩ command doesn’t make any \global changes. The coding is very similar to part of \selectlanguage.

\bbl@beforeforeign is a trick to fix a bug in bidi texts. \foreignlanguage is supposed to be a ‘text’ command, and therefore it must emit a \leavevmode, but it does not, and therefore the indent is placed on the opposite margin. For backward compatibility, however, it is done only if a right-to-left script is requested; otherwise, it is no-op.

(3.11) \foreignlanguage* is a temporary, experimental macro for a few lines with a different script direction, while preserving the paragraph format (thank the braces around \par, things like \hangindent are not reset). Do not use it in production, because its semantics and its syntax may change (and very likely will, or even it could be removed altogether). Currently it enters in vmode and then selects the language (which in turn sets the paragraph direction).

(3.11) Also experimental are the hook foreign and foreign*. With them you can redefine \BabelText which by default does nothing. Its behaviour is not well defined yet. So, use it in horizontal mode only if you do not want surprises.

In other words, at the beginning of a paragraph \foreignlanguage enters into hmode with the surrounding lang, and with \foreignlanguage* with the new lang.

```

2338 \let\bbl@beforeforeign\@empty
2339 \edef\foreignlanguage{%
2340   \noexpand\protect
2341   \expandafter\noexpand\csname foreignlanguage \endcsname}
2342 \expandafter\def\csname foreignlanguage \endcsname{%
2343   \@ifstar\bbl@foreign@s\bbl@foreign@x}
2344 \def\bbl@foreign@x#1#2{%
2345   \begingroup
2346     \let\BabelText\@firstofone
2347     \bbl@beforeforeign
2348     \foreign@language{#1}%
2349     \bbl@usehooks{foreign}{}%
2350     \BabelText{#2}% Now in horizontal mode!
2351   \endgroup}
2352 \def\bbl@foreign@s#1#2{% TODO - \shapemode, \setpar, ?\@@par
2353   \begingroup
2354     {\par}%
2355     \let\BabelText\@firstofone
2356     \foreign@language{#1}%
2357     \bbl@usehooks{foreign*}{}%
2358     \bbl@dirparastext
2359     \BabelText{#2}% Still in vertical mode!
2360     {\par}%
2361   \endgroup}

```

\foreign@language This macro does the work for \foreignlanguage and the otherlanguage* environment. First we need to store the name of the language and check that it is a known language. Then it just calls bbl@switch.

```

2362 \def\foreign@language#1{%
2363   \edef\language{#1}%
2364   \bbl@fixname\language
2365   \bbl@iflanguage\language{%
2366     \expandafter\ifx\csname date\language\endcsname\relax
2367       \bbl@warning
2368       {Unknown language `#1'. Either you have\\%
2369         misspelled its name, it has not been installed,\\%
2370         or you requested it in a previous run. Fix its name,\\%
2371         install it or just rerun the file, respectively.\\%
2372         I'll proceed, but expect wrong results.\\%
2373         Reported}%
2374     \fi
2375     \let\bbl@select@type\@ne
2376     \expandafter\bbl@switch\expandafter{\language}}%

```

\bbl@patterns This macro selects the hyphenation patterns by changing the `\language` register. If special hyphenation patterns are available specifically for the current font encoding, use them instead of the default.

It also sets hyphenation exceptions, but only once, because they are global (here language `\lccode's` has been set, too). `\bbl@hyphenation@` is set to relax until the very first `\babelhyphenation`, so do nothing with this value. If the exceptions for a language (by its number, not its name, so that `:ENC` is taken into account) has been set, then use `\hyphenation` with both global and language exceptions and empty the latter to mark they must not be set again.

```

2377 \let\bbl@hyphlist\@empty
2378 \let\bbl@hyphenation@\relax
2379 \let\bbl@pttnlist\@empty
2380 \let\bbl@patterns@\relax
2381 \let\bbl@hymapsel=\@cclv
2382 \def\bbl@patterns#1{%
2383   \language=\expandafter\ifx\csname l@#1:\f@encoding\endcsname\relax
2384     \csname l@#1\endcsname
2385     \edef\bbl@tempa{#1}%
2386   \else
2387     \csname l@#1:\f@encoding\endcsname
2388     \edef\bbl@tempa{#1:\f@encoding}%
2389   \fi
2390   \@expandtwoargs\bbl@usehooks{patterns}{#1}{\bbl@tempa}%
2391   \@ifundefined{bbl@hyphenation@}{#1}{% Can be \relax!
2392     \begingroup
2393       \bbl@xin@{,\number\language,}{,\bbl@hyphlist}%
2394       \ifin@else
2395         \@expandtwoargs\bbl@usehooks{hyphenation}{#1}{\bbl@tempa}%
2396         \hyphenation{%
2397           \bbl@hyphenation@
2398           \@ifundefined{bbl@hyphenation@#1}%
2399             \@empty
2400             {\space\csname bbl@hyphenation@#1\endcsname}}%
2401         \xdef\bbl@hyphlist{\bbl@hyphlist\number\language,}%
2402       \fi
2403     \endgroup}}

```

hyphenrules The environment `hyphenrules` can be used to select *just* the hyphenation rules. This environment does *not* change `\language` and when the hyphenation rules specified were not loaded it has no effect. Note however, `\lccode's` and font encodings are not set at all, so in most cases you should use `other language*`.

```

2404 \def\hyphenrules#1{%
2405   \edef\bbl@tempf{#1}%
2406   \bbl@fixname\bbl@tempf
2407   \bbl@iflanguage\bbl@tempf{%
2408     \expandafter\bbl@patterns\expandafter{\bbl@tempf}%
2409     \languageshortands{none}%
2410     \bbl@ifunset{\bbl@tempf hyphenmins}%
2411       {\set@hyphenmins\tw@\thr@\relax}%
2412       {\bbl@exp{\set@hyphenmins\@nameuse{\bbl@tempf hyphenmins}}}}
2413 \let\endhyphenrules\empty

```

`\providehyphenmins` The macro `\providehyphenmins` should be used in the language definition files to provide a *default* setting for the hyphenation parameters `\lefthyphenmin` and `\righthyphenmin`. If the macro `\langhyphenmins` is already defined this command has no effect.

```

2414 \def\providehyphenmins#1#2{%
2415   \expandafter\ifx\csname #1hyphenmins\endcsname\relax
2416   \@namedef{#1hyphenmins}{#2}%
2417   \fi}

```

`\set@hyphenmins` This macro sets the values of `\lefthyphenmin` and `\righthyphenmin`. It expects two values as its argument.

```

2418 \def\set@hyphenmins#1#2{%
2419   \lefthyphenmin#1\relax
2420   \righthyphenmin#2\relax}

```

`\ProvidesLanguage` The identification code for each file is something that was introduced in L^AT_EX 2_ε. When the command `\ProvidesFile` does not exist, a dummy definition is provided temporarily. For use in the language definition file the command `\ProvidesLanguage` is defined by `babel`. Depending on the format, ie, on if the former is defined, we use a similar definition or not.

```

2421 \ifx\ProvidesFile\@undefined
2422   \def\ProvidesLanguage#1[#2 #3 #4]{%
2423     \wlog{Language: #1 #4 #3 <#2>}%
2424     }
2425 \else
2426   \def\ProvidesLanguage#1{%
2427     \begingroup
2428     \catcode`\ 10 %
2429     \@makeother\/%
2430     \@ifnextchar[%]
2431       {\@provideslanguage{#1}}{\@provideslanguage{#1}[]}
2432   \def\@provideslanguage#1[#2]{%
2433     \wlog{Language: #1 #2}%
2434     \expandafter\xdef\csname ver@#1.ldf\endcsname{#2}%
2435     \endgroup}
2436 \fi

```

`\LdfInit` This macro is defined in two versions. The first version is to be part of the ‘kernel’ of `babel`, ie. the part that is loaded in the format; the second version is defined in `babel.def`. The version in the format just checks the category code of the ampersand and then loads `babel.def`. The category code of the ampersand is restored and the macro calls itself again with the new definition from `babel.def`

```

2437 \def\LdfInit{%
2438   \chardef\atcatcode=\catcode`\@
2439   \catcode`\@=11\relax
2440   \input babel.def\relax
2441   \catcode`\@=\atcatcode \let\atcatcode\relax
2442   \LdfInit}

```


`\originalTeX` The macro `\originalTeX` should be known to \TeX at this moment. As it has to be expandable we `\let` it to `\@empty` instead of `\relax`.

```
2443 \ifx\originalTeX\undefined\let\originalTeX\@empty\fi
```

Because this part of the code can be included in a format, we make sure that the macro which initialises the save mechanism, `\babel@beginsave`, is not considered to be undefined.

```
2444 \ifx\babel@beginsave\undefined\let\babel@beginsave\relax\fi
```

A few macro names are reserved for future releases of babel, which will use the concept of ‘locale’:

```
2445 \providecommand\setlocale{%
2446   \bbl@error
2447   {Not yet available}%
2448   {Find an armchair, sit down and wait}}
2449 \let\uselocale\setlocale
2450 \let\locale\setlocale
2451 \let\selectlocale\setlocale
2452 \let\textlocale\setlocale
2453 \let\textlanguage\setlocale
2454 \let\languagetext\setlocale
```

10.2 Errors

`\@nolanerr` The babel package will signal an error when a documents tries to select a language that hasn’t been defined earlier. When a user selects a language for which no hyphenation patterns were loaded into the format he will be given a warning about that fact. We revert to the patterns for `\language=0` in that case. In most formats that will be (US)english, but it might also be empty.

`\@noopterr` When the package was loaded without options not everything will work as expected. An error message is issued in that case.
When the format knows about `\PackageError` it must be $\LaTeX 2_{\epsilon}$, so we can safely use its error handling interface. Otherwise we’ll have to ‘keep it simple’.

```
2455 \edef\bbl@nulllanguage{\string\language=0}
2456 \ifx\PackageError\@undefined
2457   \def\bbl@error#1#2{%
2458     \begingroup
2459     \newlinechar=`^^J
2460     \def\{^^J(babel) }%
2461     \errhelp{#2}\errmessage{\{#1}%
2462     \endgroup}
2463   \def\bbl@warning#1{%
2464     \begingroup
2465     \newlinechar=`^^J
2466     \def\{^^J(babel) }%
2467     \message{\{#1}%
2468     \endgroup}
2469   \def\bbl@info#1{%
2470     \begingroup
2471     \newlinechar=`^^J
2472     \def\{^^J}%
2473     \wlog{#1}%
2474     \endgroup}
2475 \else
2476   \def\bbl@error#1#2{%
2477     \begingroup
```

```

2478     \def\{\MessageBreak}%
2479     \PackageError{babel}{#1}{#2}%
2480   \endgroup}
2481 \def\bbl@warning#1{%
2482   \begingroup
2483     \def\{\MessageBreak}%
2484     \PackageWarning{babel}{#1}%
2485   \endgroup}
2486 \def\bbl@info#1{%
2487   \begingroup
2488     \def\{\MessageBreak}%
2489     \PackageInfo{babel}{#1}%
2490   \endgroup}
2491 \fi
2492 \@ifpackagewith{babel}{silent}
2493   {\let\bbl@info@gobble
2494    \let\bbl@warning@gobble}
2495   {}
2496 \def\bbl@nocaption#1#2{% 1: text to be printed 2: caption macro \langXname
2497   \gdef#2{\textbf{?#1?}}%
2498   #2%
2499   \bbl@warning{%
2500     \string#2 not set. Please, define\\%
2501     it in the preamble with something like:\\%
2502     \string\renewcommand\string#2{..}\\%
2503     Reported}}
2504 \def\@nolanerr#1{%
2505   \bbl@error
2506     {You haven't defined the language #1\space yet}%
2507     {Your command will be ignored, type <return> to proceed}}
2508 \def\@nopatterns#1{%
2509   \bbl@warning
2510     {No hyphenation patterns were preloaded for\\%
2511     the language `#1' into the format.\\%
2512     Please, configure your TeX system to add them and\\%
2513     rebuild the format. Now I will use the patterns\\%
2514     preloaded for \bbl@nulllanguage\space instead}}
2515 \let\bbl@usehooks\@gobbletwo
2516 </kernel>

```

11 Loading hyphenation patterns

The following code is meant to be read by \LaTeX because it should instruct \TeX to read hyphenation patterns. To this end the `docstrip` option `patterns` can be used to include this code in the file `hyphen.cfg`. Code is written with lower level macros. `toks8` stores info to be shown when the program is run.

We want to add a message to the message \LaTeX 2.09 puts in the `\everyjob` register. This could be done by the following code:

```

\let\orgeveryjob\everyjob
\def\everyjob#1{%
  \orgeveryjob{#1}%
  \orgeveryjob\expandafter{\the\orgeveryjob\immediate\write16{%
    hyphenation patterns for \the\loaded@patterns loaded.}}%
  \let\everyjob\orgeveryjob\let\orgeveryjob\@undefined}

```

The code above redefines the control sequence `\everyjob` in order to be able to add something to the current contents of the register. This is necessary because the processing of hyphenation patterns happens long before \LaTeX fills the register. There are some problems with this approach though.

- When someone wants to use several hyphenation patterns with \LaTeX the above scheme won't work. The reason is that \LaTeX overwrites the contents of the `\everyjob` register with its own message.
- Plain \TeX does not use the `\everyjob` register so the message would not be displayed.

To circumvent this a 'dirty trick' can be used. As this code is only processed when creating a new format file there is one command that is sure to be used, `\dump`. Therefore the original `\dump` is saved in `\orig@dump` and a new definition is supplied.

To make sure that \LaTeX 2.09 executes the `\@begindocumenthook` we would want to alter `\begin{document}`, but as this done too often already, we add the new code at the front of `\@preamblecmds`. But we can only do that after it has been defined, so we add this piece of code to `\dump`.

This new definition starts by adding an instruction to write a message on the terminal and in the transcript file to inform the user of the preloaded hyphenation patterns.

Then everything is restored to the old situation and the format is dumped.

```

2517 <*patterns>
2518 <<Make sure ProvidesFile is defined>>
2519 \ProvidesFile{hyphen.cfg}[<<date>> <<version>> Babel hyphens]
2520 \xdef\bbl@format{\jobname}
2521 \ifx\AtBeginDocument\@undefined
2522   \def\@empty{}
2523   \let\orig@dump\dump
2524   \def\dump{%
2525     \ifx\@ztryfc\@undefined
2526     \else
2527       \toks0=\expandafter{\@preamblecmds}%
2528       \edef\@preamblecmds{\noexpand\@begindocumenthook\the\toks0}%
2529       \def\@begindocumenthook{}%
2530     \fi
2531     \let\dump\orig@dump\let\orig@dump\@undefined\dump}
2532 \fi
2533 <<Define core switching macros>>
2534 \toks8{Babel <<@version@>> and hyphenation patterns for }%
```

`\process@line` Each line in the file language.dat is processed by `\process@line` after it is read. The first thing this macro does is to check whether the line starts with `=`. When the first token of a line is an `=`, the macro `\process@synonym` is called; otherwise the macro `\process@language` will continue.

```

2535 \def\process@line#1#2 #3 #4 {%
2536   \ifx=#1%
2537     \process@synonym{#2}%
2538   \else
2539     \process@language{#1#2}{#3}{#4}%
2540   \fi
2541   \ignorespaces}
```

`\process@synonym` This macro takes care of the lines which start with an `=`. It needs an empty token register to begin with. `\bbl@languages` is also set to empty.

```

2542 \toks@{}
2543 \def\bbl@languages{}
```

When no languages have been loaded yet, the name following the = will be a synonym for hyphenation register 0. So, it is stored in a token register and executed when the first pattern file has been processed. (The `\relax` just helps to the `\if` below catching synonyms without a language.)

Otherwise the name will be a synonym for the language loaded last.

We also need to copy the hyphenmin parameters for the synonym.

```

2544 \def\process@synonym#1{%
2545   \ifnum\last@language=\m@ne
2546     \toks@\expandafter{\the\toks@\relax\process@synonym{#1}}%
2547   \else
2548     \expandafter\chardef\csname l@#1\endcsname\last@language
2549     \wlog{\string\l@#1=\string\language\the\last@language}%
2550     \expandafter\let\csname #1hyphenmins\endcsname\expandafter\endcsname
2551     \csname\language\endcsname hyphenmins\endcsname
2552     \let\bb1@elt\relax
2553     \edef\bb1@languages{\bb1@languages\bb1@elt{#1}{\the\last@language}{}}%
2554   \fi}

```

`\process@language` The macro `\process@language` is used to process a non-empty line from the ‘configuration file’. It has three arguments, each delimited by white space. The first argument is the ‘name’ of a language; the second is the name of the file that contains the patterns. The optional third argument is the name of a file containing hyphenation exceptions. The first thing to do is call `\addlanguage` to allocate a pattern register and to make that register ‘active’. Then the ‘name’ of the language that will be loaded now is added to the token register `\toks8`. and finally the pattern file is read.

For some hyphenation patterns it is needed to load them with a specific font encoding selected. This can be specified in the file `language.dat` by adding for instance ‘:T1’ to the name of the language. The macro `\bb1@get@enc` extracts the font encoding from the language name and stores it in `\bb1@hyph@enc`. The latter can be used in hyphenation files if you need to set a behaviour depending on the given encoding (it is set to empty if no encoding is given).

Pattern files may contain assignments to `\lefthyphenmin` and `\righthyphenmin`. \TeX does not keep track of these assignments. Therefore we try to detect such assignments and store them in the `\langhyphenmins` macro. When no assignments were made we provide a default setting.

Some pattern files contain changes to the `\lccode` and `\uccode` arrays. Such changes should remain local to the language; therefore we process the pattern file in a group; the `\patterns` command acts globally so its effect will be remembered.

Then we globally store the settings of `\lefthyphenmin` and `\righthyphenmin` and close the group.

When the hyphenation patterns have been processed we need to see if a file with hyphenation exceptions needs to be read. This is the case when the third argument is not empty and when it does not contain a space token. (Note however there is no need to save hyphenation exceptions into the format.)

`\bb1@languages` saves a snapshot of the loaded languages in the form

`\bb1@elt{(language-name)}{(number)}{(patterns-file)}{(exceptions-file)}`. Note the last 2 arguments are empty in ‘dialects’ defined in `language.dat` with =. Note also the language name can have encoding info.

Finally, if the counter `\language` is equal to zero we execute the synonyms stored.

```

2555 \def\process@language#1#2#3{%
2556   \expandafter\addlanguage\csname l@#1\endcsname
2557   \expandafter\language\csname l@#1\endcsname
2558   \edef\language{#1}%
2559   \bb1@hook@everylanguage{#1}%
2560   \bb1@get@enc#1::\@@@

```

```

2561 \begingroup
2562 \lefthyphenmin\m@ne
2563 \bbl@hook@loadpatterns{#2}%
2564 \ifnum\lefthyphenmin=\m@ne
2565 \else
2566 \expandafter\xdef\csname #1hyphenmins\endcsname{%
2567 \the\lefthyphenmin\the\righthyphenmin}%
2568 \fi
2569 \endgroup
2570 \def\bbl@tempa{#3}%
2571 \ifx\bbl@tempa\@empty\else
2572 \bbl@hook@loadexceptions{#3}%
2573 \fi
2574 \let\bbl@elt\relax
2575 \edef\bbl@languages{%
2576 \bbl@languages\bbl@elt{#1}{\the\language}{#2}{\bbl@tempa}}%
2577 \ifnum\the\language=\z@
2578 \expandafter\ifx\csname #1hyphenmins\endcsname\relax
2579 \set@hyphenmins\tw@\thr@@\relax
2580 \else
2581 \expandafter\expandafter\expandafter\set@hyphenmins
2582 \csname #1hyphenmins\endcsname
2583 \fi
2584 \the\toks@
2585 \toks@{}%
2586 \fi}

```

\bbl@get@enc The macro \bbl@get@enc extracts the font encoding from the language name and stores it in \bbl@hyph@enc. It uses delimited arguments to achieve this.

```

2587 \def\bbl@get@enc#1:#2:#3\@@{\def\bbl@hyph@enc{#2}}

```

Now, hooks are defined. For efficiency reasons, they are dealt here in a special way. Besides luatex, format specific configuration files are taken into account.

```

2588 \def\bbl@hook@everylanguage#1{}
2589 \def\bbl@hook@loadpatterns#1{\input #1\relax}
2590 \let\bbl@hook@loadexceptions\bbl@hook@loadpatterns
2591 \let\bbl@hook@loadkernel\bbl@hook@loadpatterns
2592 \begingroup
2593 \def\AddBabelHook#1#2{%
2594 \expandafter\ifx\csname bbl@hook@#2\endcsname\relax
2595 \def\next{\toks1}%
2596 \else
2597 \def\next{\expandafter\gdef\csname bbl@hook@#2\endcsname####1}%
2598 \fi
2599 \next}
2600 \ifx\directlua\@undefined
2601 \ifx\XeTeXinputencoding\@undefined\else
2602 \input xebabel.def
2603 \fi
2604 \else
2605 \input luababel.def
2606 \fi
2607 \openin1 = babel-\bbl@format.cfg
2608 \ifeof1
2609 \else
2610 \input babel-\bbl@format.cfg\relax
2611 \fi
2612 \closein1

```

```

2613 \endgroup
2614 \bbl@hook@loadkernel{switch.def}

```

\readconfigfile The configuration file can now be opened for reading.

```

2615 \openin1 = language.dat

```

See if the file exists, if not, use the default hyphenation file hyphen.tex. The user will be informed about this.

```

2616 \def\languagename{english}%
2617 \ifEOF1
2618   \message{I couldn't find the file language.dat,\space
2619           I will try the file hyphen.tex}
2620   \input hyphen.tex\relax
2621   \chardef\l@english\z@
2622 \else

```

Pattern registers are allocated using count register \last@language. Its initial value is 0. The definition of the macro \newlanguage is such that it first increments the count register and then defines the language. In order to have the first patterns loaded in pattern register number 0 we initialize \last@language with the value -1.

```

2623   \last@language\m@ne

```

We now read lines from the file until the end is found

```

2624   \loop

```

While reading from the input, it is useful to switch off recognition of the end-of-line character. This saves us stripping off spaces from the contents of the control sequence.

```

2625     \endlinechar\m@ne
2626     \read1 to \bbl@line
2627     \endlinechar`^^^M

```

If the file has reached its end, exit from the loop here. If not, empty lines are skipped. Add 3 space characters to the end of \bbl@line. This is needed to be able to recognize the arguments of \process@line later on. The default language should be the very first one.

```

2628     \if T\ifEOF1F\fi T\relax
2629     \ifx\bbl@line\@empty\else
2630       \edef\bbl@line{\bbl@line\space\space\space}%
2631       \expandafter\process@line\bbl@line\relax
2632     \fi
2633   \repeat

```

Check for the end of the file. We must reverse the test for \ifEOF without \else. Then reactivate the default patterns.

```

2634 \begingroup
2635   \def\bbl@elt#1#2#3#4{%
2636     \global\language=#2\relax
2637     \gdef\languagename{#1}%
2638     \def\bbl@elt##1##2##3##4{}}%
2639   \bbl@languages
2640 \endgroup
2641 \fi

```

and close the configuration file.

```

2642 \closein1

```

We add a message about the fact that babel is loaded in the format and with which language patterns to the \everyjob register.

```

2643 \if/\the\toks@/\else

```

```

2644 \errhelp{language.dat loads no language, only synonyms}
2645 \errmessage{Orphan language synonym}
2646 \fi
2647 \advance\last@language\@ne
2648 \edef\bbl@tempa{%
2649   \everyjob{%
2650     \the\everyjob
2651     \ifx\typeout\@undefined
2652       \immediate\write16%
2653     \else
2654       \noexpand\typeout
2655     \fi
2656     {\the\toks8 \the\last@language\space language(s) loaded.}}}
2657 \advance\last@language\m@ne
2658 \bbl@tempa

```

Also remove some macros from memory and raise an error if `\toks@` is not empty. Finally load `switch.def`, but the latter is not required and the line inputting it may be commented out.

```

2659 \let\bbl@line\@undefined
2660 \let\process@line\@undefined
2661 \let\process@synonym\@undefined
2662 \let\process@language\@undefined
2663 \let\bbl@get@enc\@undefined
2664 \let\bbl@hyph@enc\@undefined
2665 \let\bbl@tempa\@undefined
2666 \let\bbl@hook@loadkernel\@undefined
2667 \let\bbl@hook@everylanguage\@undefined
2668 \let\bbl@hook@loadpatterns\@undefined
2669 \let\bbl@hook@loadexceptions\@undefined
2670 \</patterns>

```

Here the code for `iniTEX` ends.

12 Font handling with fontspec

Add the bidi handler just before `luaotfload`, which is loaded by default by LaTeX. Just in case, consider the possibility it has not been loaded. First, a couple of definitions related to bidi [misplaced].

```

2671 <<{*More package options}>> ≡
2672 \DeclareOption{bidi=basic-r}%
2673   {\newattribute\bbl@attr@dir
2674    \let\bbl@beforeforeign\leavevmode
2675    \AtEndOfPackage{\EnableBabelHook{babel-bidi}}}
2676 \DeclareOption{bidi=default}%
2677   {\let\bbl@beforeforeign\leavevmode
2678    \ifodd\bbl@engine
2679      \newattribute\bbl@attr@dir
2680    \fi
2681    \AtEndOfPackage{%
2682      \EnableBabelHook{babel-bidi}%
2683      \ifodd\bbl@engine\else
2684        \bbl@xebidipar
2685      \fi}}
2686 <</More package options>>

```

With explicit languages, we could define the font at once, but we don't. Just wait and see if the language is actually activated.

```

2687 <<{*Font selection}>> ≡
2688 \@onlypreamble\babelfont
2689 \newcommand\babelfont[2][\% 1=langs/scripts 2=fam
2690   \edef\bbbl@tempa{#1}%
2691   \def\bbbl@tempb{#2}%
2692   \ifx\fontspec\undefined
2693     \usepackage{fontspec}%
2694   \fi
2695   \EnableBabelHook{babel-fontspec}%
2696   \bbbl@babelfont}
2697 \newcommand\bbbl@babelfont[2][\% 1=features 2=fontname
2698   \bbbl@ifunset{\bbbl@tempb family}{\bbbl@providefam{\bbbl@tempb}}{}}%
2699   \bbbl@ifunset{\bbbl@lsys@\language}{\bbbl@provide@lsys{\language}}{}}%
2700   \expandafter\bbbl@ifblank\expandafter{\bbbl@tempa}%
2701   {\bbbl@csarg\edef{\bbbl@tempb dflt@}{<{#1}{#2}}% save \bbbl@rmdflt@
2702   \bbbl@exp{%
2703     \let\<\bbbl@tempb dflt@\language>\<\bbbl@tempb dflt@>%
2704     \\\bbbl@font@set\<\bbbl@tempb dflt@\language>%
2705     \<\bbbl@tempb default>\<\bbbl@tempb family>}}%
2706   {\bbbl@foreach\bbbl@tempa{% ie \bbbl@rmdflt@lang / *scrt
2707     \bbbl@csarg\def{\bbbl@tempb dflt@##1}{<{#1}{#2}}}}}%

```

If the family in the previous command does not exist, it must be defined. Here is how:

```

2708 \def\bbbl@providefam#1{%
2709   \bbbl@exp{%
2710     \\\newcommand\<#1default>{}% Just define it
2711     \\\bbbl@add@list\bbbl@font@fams{#1}%
2712     \\\DeclareRobustCommand\<#1family>{%
2713       \\\not@math@alphabet\<#1family>\relax
2714       \\\fontfamily\<#1default>\selectfont}%
2715     \\\DeclareTextFontCommand{\<text#1>}{\<#1family>}}%

```

The following macro is activated when the hook babel-fontspec is enabled.

```

2716 \def\bbbl@switchfont{%
2717   \bbbl@ifunset{\bbbl@lsys@\language}{\bbbl@provide@lsys{\language}}{}}%
2718   \bbbl@exp{% eg Arabic -> arabic
2719     \lowercase{\edef\bbbl@tempa{\bbbl@cs{sname@\language}}}}%
2720   \bbbl@foreach\bbbl@font@fams{%
2721     \bbbl@ifunset{\bbbl@##1dflt@\language}% (1) language?
2722     {\bbbl@ifunset{\bbbl@##1dflt@*\bbbl@tempa}% (2) from script?
2723       {\bbbl@ifunset{\bbbl@##1dflt@}% 2=F - (3) from generic?
2724         {}% 123=F - nothing!
2725         {\bbbl@exp{% 3=T - from generic
2726           \global\let\<\bbbl@##1dflt@\language>%
2727           \<\bbbl@##1dflt@>}}}%
2728         {\bbbl@exp{% 2=T - from script
2729           \global\let\<\bbbl@##1dflt@\language>%
2730           \<\bbbl@##1dflt@*\bbbl@tempa>}}}%
2731         {}% 1=T - language, already defined
2732     \def\bbbl@tempa{%
2733       \bbbl@warning{The current font is not a standard family.\\%
2734         Script and Language are not applied. Consider defining\\%
2735         a new family with \string\babelfont,}}%
2736     \bbbl@foreach\bbbl@font@fams{% don't gather with prev for
2737       \bbbl@ifunset{\bbbl@##1dflt@\language}%
2738       {\bbbl@cs{famrst@##1}%
2739         \global\bbbl@csarg\let{famrst@##1}\relax}%
2740       {\bbbl@exp{% order is relevant
2741         \\\bbbl@add\originalTeX{%

```



```

2742      \\bbl@font@rst{\bbl@cs{##1dflt@\languagename}}%
2743      \<##1default>\<##1family>{##1}}%
2744      \\bbl@font@set\<bbl@##1dflt@\languagename>% the main part!
2745      \<##1default>\<##1family>}}}%
2746      \bbl@ifrestoring{}{\bbl@tempa}}%

```

Now the macros defining the font with fontspec.

When there are repeated keys in fontspec, the last value wins. So, we just place the ini settings at the beginning, and user settings will take precedence.

```

2747 \def\bbl@font@set#1#2#3{%
2748   \bbl@xin@{<>}{#1}%
2749   \ifin@
2750     \bbl@exp{\bbl@fontspec@set\#1\expandafter\@gobbletwo#1}%
2751   \fi
2752   \bbl@exp{%
2753     \def\#2{#1}%          eg, \rmdefault{\bbl@rm1dflt@lang}
2754     \\bbl@ifsamestring{#2}{\f@family}{\#3\let\bbl@tempa\relax}}}%
2755 \def\bbl@fontspec@set#1#2#3{%
2756   \bbl@exp{\<fontspec_set_family:Nnn>\#1%
2757     {\bbl@cs{lsys@\languagename},#2}}{#3}%
2758   \bbl@tglobal#1}%

```

font@rst and famrst are only used when there is no global settings, to save and restore de previous families. Not really necessary, but done for optimization.

```

2759 \def\bbl@font@rst#1#2#3#4{%
2760   \bbl@csarg\def{famrst@#4}{\bbl@font@set{#1}#2#3}}

```

The default font families. They are eurocentric, but the list can be expanded easily with \babelfont.

```

2761 \def\bbl@font@fams{rm,sf,tt}

```

The old tentative way. Short and preverved for compatibility, but deprecated. Note there is no direct alternative for \babelFSfeatures. The reason in explained in the user guide, but essentially – that was not the way to go :-).

```

2762 \newcommand\babelFSstore[2][{%
2763   \bbl@ifblank{#1}%
2764   {\bbl@csarg\def{sname@#2}{Latin}}}%
2765   {\bbl@csarg\def{sname@#2}{#1}}}%
2766 \bbl@provide@dirs{#2}%
2767 \bbl@csarg\ifnum{wdir@#2}>\z@
2768   \let\bbl@beforeforeign\leavevmode
2769   \EnableBabelHook{babel-bidi}%
2770 \fi
2771 \bbl@foreach{#2}{%
2772   \bbl@FSstore{##1}{rm}\rmdefault\bbl@save@rmdefault
2773   \bbl@FSstore{##1}{sf}\sfdefault\bbl@save@sfdefault
2774   \bbl@FSstore{##1}{tt}\ttdefault\bbl@save@ttdefault}}
2775 \def\bbl@FSstore#1#2#3#4{%
2776   \bbl@csarg\edef{#2default#1}{#3}%
2777   \expandafter\addto\csname extras#1\endcsname{%
2778     \let#4#3%
2779     \ifx#3\f@family
2780       \edef#3{\csname bbl@#2default#1\endcsname}%
2781       \fontfamily{#3}\selectfont
2782     \else
2783       \edef#3{\csname bbl@#2default#1\endcsname}%
2784       \fi}%
2785   \expandafter\addto\csname noextras#1\endcsname{%

```

```

2786 \ifx#3\f@family
2787 \fontfamily{#4}\selectfont
2788 \fi
2789 \let#3#4}}
2790 \let\bbl@langfeatures\@empty
2791 \def\babelFSfeatures{% make sure \fontspec is redefined once
2792 \let\bbl@ori@fontspec\fontspec
2793 \renewcommand\fontspec[1][{}]{%
2794 \bbl@ori@fontspec[\bbl@langfeatures##1]}
2795 \let\babelFSfeatures\bbl@FSfeatures
2796 \babelFSfeatures}
2797 \def\bbl@FSfeatures#1#2{%
2798 \expandafter\addto\csname extras#1\endcsname{%
2799 \babel@save\bbl@langfeatures
2800 \edef\bbl@langfeatures{#2,}}}}
2801 <</Font selection>>

```

13 Hooks for XeTeX and LuaTeX

13.1 XeTeX

Unfortunately, the current encoding cannot be retrieved and therefore it is reset always to `utf8`, which seems a sensible default.

\LaTeX sets many “codes” just before loading `hyphen.cfg`. That is not a problem in `luatex`, but in `xetex` they must be reset to the proper value. Most of the work is done in `xe(la)tex.ini`, so here we just “undo” some of the changes done by \LaTeX . Anyway, for consistency `LuaTeX` also resets the catcodes.

```

2802 <<(*Restore Unicode catcodes before loading patterns)>> ≡
2803 \begingroup
2804 % Reset chars "80-"C0 to category "other", no case mapping:
2805 \catcode\@=11 \count@=128
2806 \loop\ifnum\count@<192
2807 \global\uccode\count@=0 \global\lccode\count@=0
2808 \global\catcode\count@=12 \global\sffcode\count@=1000
2809 \advance\count@ by 1 \repeat
2810 % Other:
2811 \def\O ##1 {%
2812 \global\uccode"##1=0 \global\lccode"##1=0
2813 \global\catcode"##1=12 \global\sffcode"##1=1000 }%
2814 % Letter:
2815 \def\L ##1 ##2 ##3 {\global\catcode"##1=11
2816 \global\uccode"##1="##2
2817 \global\lccode"##1="##3
2818 % Uppercase letters have sffcode=999:
2819 \ifnum"##1="##3 \else \global\sffcode"##1=999 \fi }%
2820 % Letter without case mappings:
2821 \def\l ##1 {\L ##1 ##1 ##1 }%
2822 \l 00AA
2823 \L 00B5 039C 00B5
2824 \l 00BA
2825 \O 00D7
2826 \l 00DF
2827 \O 00F7
2828 \L 00FF 0178 00FF
2829 \endgroup
2830 \input #1\relax
2831 <</Restore Unicode catcodes before loading patterns>>

```

Now, the code.

```
2832 (*xetex)
2833 \def\BabelStringsDefault{unicode}
2834 \let\xebbl@stop\relax
2835 \AddBabelHook{xetex}{encodedcommands}{%
2836   \def\bbl@tempa{#1}%
2837   \ifx\bbl@tempa@empty
2838     \XeTeXinputencoding"bytes"%
2839   \else
2840     \XeTeXinputencoding"#1"%
2841   \fi
2842   \def\xebbl@stop{\XeTeXinputencoding"utf8"}}}
2843 \AddBabelHook{xetex}{stopcommands}{%
2844   \xebbl@stop
2845   \let\xebbl@stop\relax}
2846 \AddBabelHook{xetex}{loadkernel}{%
2847   <<Restore Unicode catcodes before loading patterns>>}
2848 \ifx\DisableBabelHook\undefined\endinput\fi
2849 \AddBabelHook{babel-fontspec}{afterextras}{\bbl@switchfont}
2850 \DisableBabelHook{babel-fontspec}
2851 <<Font selection>>
2852 </xetex>
```

13.2 LuaTeX

The new loader for luatex is based solely on `language.dat`, which is read on the fly. The code shouldn't be executed when the format is build, so we check if `\AddBabelHook` is defined. Then comes a modified version of the loader in `hyphen.cfg` (without the `hyphenmins` stuff, which is under the direct control of `babel`).

The names `\l@<language>` are defined and take some value from the beginning because all `ldf` files assume this for the corresponding language to be considered valid, but patterns are not loaded (except the first one). This is done later, when the language is first selected (which usually means when the `ldf` finishes). If a language has been loaded, `\bbl@hyphendata@<num>` exists (with the names of the files read).

The default setup preloads the first language into the format. This is intended mainly for 'english', so that it's available without further intervention from the user. To avoid duplicating it, the following rule applies: if the "0th" language and the first language in `language.dat` have the same name then just ignore the latter. If there are new synonymous, they are added, but note if the language patterns have not been preloaded they won't at run time.

Other preloaded languages could be read twice, if they have been preloaded into the format. This is not optimal, but it shouldn't happen very often – with luatex patterns are best loaded when the document is typeset, and the "0th" language is preloaded just for backwards compatibility.

As of 1.1b, `lua(e)tex` is taken into account. Formerly, loading of patterns on the fly didn't work in this format, but with the new loader it does. Unfortunately, the format is not based on `babel`, and data could be duplicated, because languages are reassigned above those in the format (nothing serious, anyway). Note even with this format `language.dat` is used (under the principle of a single source), instead of `language.def`.

Of course, there is room for improvements, like tools to read and reassign languages, which would require modifying the language list, and better error handling.

We need catcode tables, but no format (targeted by `babel`) provide a command to allocate them (although there are packages like `ctablestack`). For the moment, a dangerous approach is used – just allocate a high random number and cross the fingers. To complicate things, `etex.sty` changes the way languages are allocated.

```

2853 (*luatex)
2854 \ifx\AddBabelHook\undefined
2855 \begingroup
2856 \toks@{}
2857 \count@ \z@ % 0=start, 1=0th, 2=normal
2858 \def\bbl@process@line#1#2 #3 #4 {%
2859   \ifx=#1%
2860     \bbl@process@synonym{#2}%
2861   \else
2862     \bbl@process@language{#1#2}{#3}{#4}%
2863   \fi
2864   \ignorespaces}
2865 \def\bbl@manylang{%
2866   \ifnum\bbl@last>\@ne
2867     \bbl@info{Non-standard hyphenation setup}%
2868   \fi
2869   \let\bbl@manylang\relax}
2870 \def\bbl@process@language#1#2#3{%
2871   \ifcase\count@
2872     \@ifundefined{zth#1}{\count@\tw@}{\count@\@ne}%
2873   \or
2874     \count@\tw@
2875   \fi
2876   \ifnum\count@=\tw@
2877     \expandafter\addlanguage\csname l@#1\endcsname
2878     \language\allocationnumber
2879     \chardef\bbl@last\allocationnumber
2880     \bbl@manylang
2881     \let\bbl@elt\relax
2882     \xdef\bbl@languages{%
2883       \bbl@languages\bbl@elt{#1}{\the\language}{#2}{#3}}%
2884   \fi
2885   \the\toks@
2886   \toks@{}}
2887 \def\bbl@process@synonym@aux#1#2{%
2888   \global\expandafter\chardef\csname l@#1\endcsname#2\relax
2889   \let\bbl@elt\relax
2890   \xdef\bbl@languages{%
2891     \bbl@languages\bbl@elt{#1}{#2}{}}}%
2892 \def\bbl@process@synonym#1{%
2893   \ifcase\count@
2894     \toks@\expandafter{\the\toks@\relax\bbl@process@synonym{#1}}%
2895   \or
2896     \@ifundefined{zth#1}{\bbl@process@synonym@aux{#1}{0}}{}%
2897   \else
2898     \bbl@process@synonym@aux{#1}{\the\bbl@last}%
2899   \fi}
2900 \ifx\bbl@languages\undefined % Just a (sensible?) guess
2901   \chardef\l@english\z@
2902   \chardef\l@USenglish\z@
2903   \chardef\bbl@last\z@
2904   \global\@namedef{bbl@hyphendata@0}{\hyphen.tex}{}
2905   \gdef\bbl@languages{%
2906     \bbl@elt{english}{0}{\hyphen.tex}}%
2907   \bbl@elt{USenglish}{0}{}
2908 \else
2909   \global\let\bbl@languages@format\bbl@languages
2910   \def\bbl@elt#1#2#3#4{% Remove all except language 0
2911     \ifnum#2>\z@\else

```

```

2912     \noexpand\bb1@elt{#1}{#2}{#3}{#4}%
2913     \fi}%
2914     \xdef\bb1@languages{\bb1@languages}%
2915 \fi
2916 \def\bb1@elt#1#2#3#4{\@namedef{zth#1}{}} % Define flags
2917 \bb1@languages
2918 \openin1=language.dat
2919 \ifeof1
2920     \bb1@warning{I couldn't find language.dat. No additional\\%
2921                 patterns loaded. Reported}%
2922 \else
2923     \loop
2924     \endlinechar\m@ne
2925     \read1 to \bb1@line
2926     \endlinechar\^^M
2927     \if T\ifeof1F\fi T\relax
2928     \ifx\bb1@line\@empty\else
2929         \edef\bb1@line{\bb1@line\space\space\space}%
2930         \expandafter\bb1@process@line\bb1@line\relax
2931     \fi
2932     \repeat
2933 \fi
2934 \endgroup
2935 \def\bb1@get@enc#1:#2:#3@@@{\def\bb1@hyph@enc{#2}}
2936 \ifx\babelcatcodetablenum\@undefined
2937     \def\babelcatcodetablenum{5211}
2938 \fi
2939 \def\bb1@luapatterns#1#2{%
2940     \bb1@get@enc#1::@@@
2941     \setbox\z@\hbox\bgroup
2942     \begingroup
2943     \ifx\catcodetable\@undefined
2944         \let\savecatcodetable\luatexsavecatcodetable
2945         \let\initcatcodetable\luatexinitcatcodetable
2946         \let\catcodetable\luatexcatcodetable
2947     \fi
2948     \savecatcodetable\babelcatcodetablenum\relax
2949     \initcatcodetable\numexpr\babelcatcodetablenum+1\relax
2950     \catcodetable\numexpr\babelcatcodetablenum+1\relax
2951     \catcode`\_#6 \catcode`\$=3 \catcode`\&=4 \catcode`\^=7
2952     \catcode`\_#8 \catcode`\{=1 \catcode`\}=2 \catcode`\~#13
2953     \catcode`\@#11 \catcode`\^^I=10 \catcode`\^^J=12
2954     \catcode`\<#12 \catcode`\>#12 \catcode`\*=12 \catcode`\.=12
2955     \catcode`\-#12 \catcode`\/=12 \catcode`\[=12 \catcode`\]=12
2956     \catcode`\`#12 \catcode`\'=12 \catcode`\"#12
2957     \input #1\relax
2958     \catcodetable\babelcatcodetablenum\relax
2959     \endgroup
2960     \def\bb1@tempa{#2}%
2961     \ifx\bb1@tempa\@empty\else
2962         \input #2\relax
2963     \fi
2964 \egroup}%
2965 \def\bb1@patterns@lua#1{%
2966     \language=\expandafter\ifx\csname l@#1:\f@encoding\endcsname\relax
2967         \csname l@#1\endcsname
2968         \edef\bb1@tempa{#1}%
2969     \else
2970         \csname l@#1:\f@encoding\endcsname

```

```

2971 \edef\bbl@tempa{#1:\f@encoding}%
2972 \fi\relax
2973 \@namedef{lu@texhyphen@loaded@the\language}{}% Temp
2974 \@ifundefined{bbl@hyphendata@the\language}%
2975 {\def\bbl@elt##1##2##3##4{%
2976 \ifnum##2=\csname l@bbl@tempa\endcsname % #2=spanish, dutch:OT1...
2977 \def\bbl@tempb{##3}%
2978 \ifx\bbl@tempb\@empty\else % if not a synonymous
2979 \def\bbl@tempc{##3}{##4}}%
2980 \fi
2981 \bbl@csarg\xdef{hyphendata@##2}{\bbl@tempc}%
2982 \fi}%
2983 \bbl@languages
2984 \@ifundefined{bbl@hyphendata@the\language}%
2985 {\bbl@info{No hyphenation patterns were set for\%
2986 language '\bbl@tempa'. Reported}}%
2987 {\expandafter\expandafter\expandafter\bbl@luapatterns
2988 \csname bbl@hyphendata@the\language\endcsname}}}%
2989 \endinput\fi
2990 \begingroup
2991 \catcode`\%=12
2992 \catcode`\'=12
2993 \catcode`\ "=12
2994 \catcode`\:=12
2995 \directlua{
2996 Babel = Babel or {}
2997 function Babel.bytes(line)
2998 return line:gsub(".",
2999 function (chr) return unicode.utf8.char(string.byte(chr)) end)
3000 end
3001 function Babel.begin_process_input()
3002 if luatexbase and luatexbase.add_to_callback then
3003 luatexbase.add_to_callback('process_input_buffer',
3004 Babel.bytes, 'Babel.bytes')
3005 else
3006 Babel.callback = callback.find('process_input_buffer')
3007 callback.register('process_input_buffer', Babel.bytes)
3008 end
3009 end
3010 function Babel.end_process_input ()
3011 if luatexbase and luatexbase.remove_from_callback then
3012 luatexbase.remove_from_callback('process_input_buffer', 'Babel.bytes')
3013 else
3014 callback.register('process_input_buffer', Babel.callback)
3015 end
3016 end
3017 function Babel.addpatterns(pp, lg)
3018 local lg = lang.new(lg)
3019 local pats = lang.patterns(lg) or ''
3020 lang.clear_patterns(lg)
3021 for p in pp:gmatch('[^%s]+') do
3022 ss = ''
3023 for i in string.utfcharacters(p:gsub('%d', '')) do
3024 ss = ss .. '%d?' .. i
3025 end
3026 ss = ss:gsub('^%%d%?%', '%%.') .. '%d?'
3027 ss = ss:gsub('%.%%d%?$', '%%.')
3028 pats, n = pats:gsub('%s' .. ss .. '%s', ' ' .. p .. ' ')
3029 if n == 0 then

```

```

3030         tex.sprint(
3031             [[\string\csname\space bbl@info\endcsname{New pattern: }]]
3032             .. p .. [[]])
3033         pats = pats .. ' ' .. p
3034     else
3035         tex.sprint(
3036             [[\string\csname\space bbl@info\endcsname{Renew pattern: }]]
3037             .. p .. [[]])
3038     end
3039 end
3040 lang.patterns(lg, pats)
3041 end
3042 }
3043 \endgroup
3044 \def\BabelStringsDefault{unicode}
3045 \let\luabbl@stop\relax
3046 \AddBabelHook{luatex}{encodedcommands}{%
3047     \def\bbl@tempa{utf8}\def\bbl@tempb{#1}%
3048     \ifx\bbl@tempa\bbl@tempb\else
3049         \directlua{Babel.begin_process_input()}%
3050         \def\luabbl@stop{%
3051             \directlua{Babel.end_process_input()}}%
3052     \fi}%
3053 \AddBabelHook{luatex}{stopcommands}{%
3054     \luabbl@stop
3055     \let\luabbl@stop\relax}
3056 \AddBabelHook{luatex}{patterns}{%
3057     \@ifundefined{bbl@hyphendata@the\language}%
3058     {\def\bbl@elt##1##2##3##4{%
3059         \ifnum##2=\csname l@##2\endcsname % #2=spanish, dutch:OT1...
3060         \def\bbl@tempb{##3}%
3061         \ifx\bbl@tempb\@empty\else % if not a synonymous
3062             \def\bbl@tempc{##3}{##4}}%
3063         \fi
3064         \bbl@csarg\xdef{hyphendata@##2}{\bbl@tempc}%
3065     \fi}%
3066     \bbl@languages
3067     \@ifundefined{bbl@hyphendata@the\language}%
3068     {\bbl@info{No hyphenation patterns were set for\%
3069         language ‘#2’. Reported}}%
3070     {\expandafter\expandafter\expandafter\bbl@luapatterns
3071         \csname bbl@hyphendata@the\language\endcsname}}}%
3072 \@ifundefined{bbl@patterns@}{}%
3073 \begingroup
3074     \bbl@xin@{, \number\language,}{, \bbl@pttnlist}%
3075     \ifin@ \else
3076         \ifx\bbl@patterns@\@empty\else
3077             \directlua{ Babel.addpatterns(
3078                 [[\bbl@patterns@]], \number\language) }%
3079             \fi
3080             \@ifundefined{bbl@patterns@#1}%
3081             \@empty
3082             {\directlua{ Babel.addpatterns(
3083                 [[\space\csname bbl@patterns@#1\endcsname]],
3084                 \number\language) }}%
3085             \xdef\bbl@pttnlist{\bbl@pttnlist\number\language,}%
3086         \fi
3087     \endgroup}}
3088 \AddBabelHook{luatex}{everylanguage}{%

```

```

3089 \def\process@language##1##2##3{%
3090   \def\process@line####1####2 ####3 ####4 {}}}
3091 \AddBabelHook{luatex}{loadpatterns}{%
3092   \input #1\relax
3093   \expandafter\gdef\csname bbl@hyphendata@the\language\endcsname
3094     {{#1}{}}}
3095 \AddBabelHook{luatex}{loadexceptions}{%
3096   \input #1\relax
3097   \def\bbl@tempb##1##2{{##1}{#1}}%
3098   \expandafter\xdef\csname bbl@hyphendata@the\language\endcsname
3099     {\expandafter\expandafter\expandafter\bbl@tempb
3100       \csname bbl@hyphendata@the\language\endcsname}}

```

`\babelpatterns` This macro adds patterns. Two macros are used to store them: `\bbl@patterns@` for the global ones and `\bbl@patterns@<lang>` for language ones. We make sure there is a space between words when multiple commands are used.

```

3101 \@onlypreamble\babelpatterns
3102 \AtEndOfPackage{%
3103   \newcommand\babelpatterns[2][\@empty]{%
3104     \ifx\bbl@patterns\relax
3105       \let\bbl@patterns@\@empty
3106     \fi
3107     \ifx\bbl@pttnlist\@empty\else
3108       \bbl@warning{%
3109         You must not intermingle \string\selectlanguage\space and\\%
3110         \string\babelpatterns\space or some patterns will not\\%
3111         be taken into account. Reported}%
3112     \fi
3113     \ifx\@empty#1%
3114       \protected@edef\bbl@patterns@{\bbl@patterns@\space#2}%
3115     \else
3116       \edef\bbl@tempb{\zap@space#1 \@empty}%
3117       \bbl@for\bbl@tempa\bbl@tempb{%
3118         \bbl@fixname\bbl@tempa
3119         \bbl@iflanguage\bbl@tempa{%
3120           \bbl@csarg\protected@edef{patterns@\bbl@tempa}{%
3121             \@ifundefined{bbl@patterns@\bbl@tempa}%
3122               \@empty
3123               {\csname bbl@patterns@\bbl@tempa\endcsname\space}%
3124             #2}}}%
3125     \fi}}

```

Common stuff.

```

3126 \AddBabelHook{luatex}{loadkernel}{%
3127   <<Restore Unicode catcodes before loading patterns>>}
3128 \ifx\DisableBabelHook\@undefined\endinput\fi
3129 \AddBabelHook{babel-fontspec}{afterextras}{\bbl@switchfont}
3130 \DisableBabelHook{babel-fontspec}
3131 <<Font selection>>
3132 </luatex>

```

14 Bidi support in luatex

Work in progress. The file `babel-bidi.lua` currently only contains data. It's a large and boring file and it's not shown here. See the generated file.

Now the basic-`r` bidi mode. One of the aims is to implement a fast and simple bidi algorithm, with a single loop. I managed to do it for R texts, with a second smaller loop for

a special case. The code is still somewhat chaotic, but its behavior is essentially correct. I cannot resist copying the following text from Emacs `bidi.c` (which also attempts to implement the bidi algorithm with a single loop):

Arrrgh!! The UAX#9 algorithm is too deeply entrenched in the assumption of batch-style processing [...]. May the fleas of a thousand camels infest the armpits of those who design supposedly general-purpose algorithms by looking at their own implementations, and fail to consider other possible implementations!

Well, it took me some time to guess what the batch rules in UAX#9 actually mean (in other word, *what* they do and *why*, and not only *how*), but I think (or I hope) I've managed to understand them.

In some sense, there are two bidi modes, one for numbers, and the other for text. Furthermore, setting just the direction in R text is not enough, because there are actually *two* R modes (set explicitly in Unicode with RLM and ALM). In babel the dir is set by a higher protocol based on the language/script, which in turn sets the correct dir (<l>, <r> or <al>).

From UAX#9: “Where available, markup should be used instead of the explicit formatting characters”. So, this simple version just ignores formatting characters. Actually, most of that annex is devoted to how to handle them.

BD14-BD16 are not implemented. Unicode (and the W3C) are making a great effort to deal with some special problematic cases in “streamed” plain text. I don't think this is the way to go – particular issues should be fixed by a high level interface taking into account the needs of the document. And here is where `luatex` excels, because everything related to bidi writing is under our control.

```

3133 (*basic-r)
3134 Babel = Babel or {}
3135
3136 require('babel-bidi.lua')
3137
3138 local characters = Babel.characters
3139 local ranges = Babel.ranges
3140
3141 local DIR = node.id("dir")
3142
3143 local function dir_mark(head, from, to, outer)
3144   dir = (outer == 'r') and 'TLT' or 'TRT' -- ie, reverse
3145   local d = node.new(DIR)
3146   d.dir = '+' .. dir
3147   node.insert_before(head, from, d)
3148   d = node.new(DIR)
3149   d.dir = '-' .. dir
3150   node.insert_after(head, to, d)
3151 end
3152
3153 function Babel.pre_otfload(head)
3154   local first_n, last_n          -- first and last char with nums
3155   local last_es                  -- an auxiliary 'last' used with nums
3156   local first_d, last_d          -- first and last char in L/R block
3157   local dir, dir_real

```

Next also depends on `script/lang` (<al>/<r>). To be set by babel. `tex.pardir` is dangerous, could be (re)set but it should be changed only in vmode. There are two strong's – strong = l/al/r and strong_lr = l/r (there must be a better way):

```

3158   local strong = ('TRT' == tex.pardir) and 'r' or 'l'
3159   local strong_lr = (strong == 'l') and 'l' or 'r'
3160   local outer = strong

```

```

3161
3162 local new_dir = false
3163 local first_dir = false
3164
3165 local last_lr
3166
3167 local type_n = ''
3168
3169 for item in node.traverse(head) do
3170
3171   -- three cases: glyph, dir, otherwise
3172   if item.id == node.id'glyph' then
3173
3174     local chardata = characters[item.char]
3175     dir = chardata and chardata.d or nil
3176     if not dir then
3177       for nn, et in ipairs(ranges) do
3178         if item.char < et[1] then
3179           break
3180         elseif item.char <= et[2] then
3181           dir = et[3]
3182           break
3183         end
3184       end
3185     end
3186     dir = dir or 'l'

```

Next is based on the assumption babel sets the language AND switches the script with its dir. We treat a language block as a separate Unicode sequence. The following piece of code is executed at the first glyph after a ‘dir’ node. We don’t know the current language until then.

```

3187   if new_dir then
3188     attr_dir = 0
3189     for at in node.traverse(item.attr) do
3190       if at.number == luatexbase.registernumber'bbl@attr@dir' then
3191         attr_dir = at.value
3192       end
3193     end
3194     texio.write_nl(attr_dir)
3195     if attr_dir == 1 then
3196       strong = 'r'
3197     elseif attr_dir == 2 then
3198       strong = 'al'
3199     else
3200       strong = 'l'
3201     end
3202     strong_lr = (strong == 'l') and 'l' or 'r'
3203     outer = strong_lr
3204     new_dir = false
3205   end
3206   if dir == 'nsm' then dir = strong end -- W1

```

Numbers. The dual <al>/<r> system for R is somewhat cumbersome.

```

3207   dir_real = dir -- We need dir_real to set strong below
3208   if dir == 'al' then dir = 'r' end -- W3

```

By W2, there are no <en> <et> <es> if strong == <al>, only <an>. Therefore, there are not <et en> nor <en et>, W5 can be ignored, and W6 applied:

```

3209   if strong == 'al' then

```

```

3210         if dir == 'en' then dir = 'an' end           -- W2
3211         if dir == 'et' or dir == 'es' then dir = 'on' end -- W6
3212         strong_lr = 'r'                               -- W3
3213     end

```

Once finished the basic setup for glyphs, consider the two other cases: dir node and the rest.

```

3214     elseif item.id == node.id'dir' then
3215         new_dir = true
3216         dir = nil
3217     else
3218         dir = nil           -- Not a char
3219     end

```

Numbers in R mode. A sequence of <en>, <et>, <an>, <es> and <cs> is typeset (with some rules) in L mode. We store the starting and ending points, and only when anything different is found (including nil, ie, a non-char), the textdir is set. This means you cannot insert, say, a whatsit, but this is what I would expect (with luacolor you may colorize some digits). Anyway, this behaviour could be changed with a switch in the future. Note in the first branch only <an> is relevant if <al>.

```

3220     if dir == 'en' or dir == 'an' or dir == 'et' then
3221         if dir ~= 'et' then
3222             type_n = dir
3223         end
3224         first_n = first_n or item
3225         last_n = last_es or item
3226         last_es = nil
3227     elseif dir == 'es' and last_n then -- W3+W6
3228         last_es = item
3229     elseif dir == 'cs' then           -- it's right - do nothing
3230     elseif first_n then -- & if dir = any but en, et, an, es, cs, inc nil
3231         if strong_lr == 'r' and type_n ~= '' then
3232             dir_mark(head, first_n, last_n, 'r')
3233         elseif strong_lr == 'l' and first_d and type_n == 'an' then
3234             dir_mark(head, first_n, last_n, 'r')
3235             dir_mark(head, first_d, last_d, outer)
3236             first_d, last_d = nil, nil
3237         elseif strong_lr == 'l' and type_n ~= '' then
3238             last_d = last_n
3239         end
3240         type_n = ''
3241         first_n, last_n = nil, nil
3242     end

```

R text in L, or L text in R. Order of dir_ mark's are relevant: d goes outside n, and therefore it's emitted after. See dir_mark to understand why (but is the nesting actually necessary or is a flat dir structure enough?). Only L, R (and AL) chars are taken into account – everything else, including spaces, whatsits, etc., are ignored:

```

3243     if dir == 'l' or dir == 'r' then
3244         if dir ~= outer then
3245             first_d = first_d or item
3246             last_d = item
3247         elseif first_d and dir ~= strong_lr then
3248             dir_mark(head, first_d, last_d, outer)
3249             first_d, last_d = nil, nil
3250         end
3251     end

```

Mirroring. Each chunk of text in a certain language is considered a “closed” sequence. If <r on r> and <l on l>, it’s clearly <r> and <l>, resp’tly, but with other combinations depends on outer. From all these, we select only those resolving <on> → <r>. At the beginning (when last_lr is nil) of an R text, they are mirrored directly.

TODO - numbers in R mode are processed. It doesn’t hurt, but should not be done.

```

3252   if dir and not last_lr and dir ~= 'l' and outer == 'r' then
3253       item.char = characters[item.char] and
3254           characters[item.char].m or item.char
3255   elseif (dir or new_dir) and last_lr ~= item then
3256       local mir = outer .. strong_lr .. (dir or outer)
3257       if mir == 'rrr' or mir == 'lrr' or mir == 'rrl' or mir == 'rlr' then
3258           for ch in node.traverse(node.next(last_lr)) do
3259               if ch == item then break end
3260               if ch.id == node.id'glyph' then
3261                   ch.char = characters[ch.char].m or ch.char
3262               end
3263           end
3264       end
3265   end

```

Save some values for the next iteration. If the current node is ‘dir’, open a new sequence.

Since dir could be changed, strong is set with its real value (dir_real).

```

3266   if dir == 'l' or dir == 'r' then
3267       last_lr = item
3268       strong = dir_real          -- Don't search back - best save now
3269       strong_lr = (strong == 'l') and 'l' or 'r'
3270   elseif new_dir then
3271       last_lr = nil
3272   end
3273 end

```

Mirror the last chars if they are no directed. And make sure any open block is closed, too.

```

3274   if last_lr and outer == 'r' then
3275       for ch in node.traverse_id(node.id'glyph', node.next(last_lr)) do
3276           ch.char = characters[ch.char].m or ch.char
3277       end
3278   end
3279   if first_n then
3280       dir_mark(head, first_n, last_n, outer)
3281   end
3282   if first_d then
3283       dir_mark(head, first_d, last_d, outer)
3284   end

```

In boxes, the dir node could be added before the original head, so the actual head is the previous node.

```

3285   return node.prev(head) or head
3286 end
3287 </basic-r>

```

15 The ‘nil’ language

This ‘language’ does nothing, except setting the hyphenation patterns to nohyphenation. For this language currently no special definitions are needed or available.

The macro \LdfInit takes care of preventing that this file is loaded more than once, checking the category code of the @ sign, etc.

```

3288 (*nil)
3289 \ProvidesLanguage{nil}[<<date>> <<version>> Nil language]
3290 \LdfInit{nil}{datenil}

```

When this file is read as an option, i.e. by the `\usepackage` command, `nil` could be an ‘unknown’ language in which case we have to make it known.

```

3291 \ifx\l@nohyphenation\@undefined
3292   \@nopatterns{nil}
3293   \adddialect\l@nil0
3294 \else
3295   \let\l@nil\l@nohyphenation
3296 \fi

```

This macro is used to store the values of the hyphenation parameters `\lefthyphenmin` and `\righthyphenmin`.

```

3297 \providehyphenmins{\CurrentOption}{\m@ne\m@ne}

```

The next step consists of defining commands to switch to (and from) the ‘nil’ language.

```

\captionnil
\datenil
3298 \let\captionnil\@empty
3299 \let\datenil\@empty

```

The macro `\ldf@finish` takes care of looking for a configuration file, setting the main language to be switched on at `\begin{document}` and resetting the category code of `@` to its original value.

```

3300 \ldf@finish{nil}
3301 </nil>

```

16 Support for Plain \TeX (plain.def)

16.1 Not renaming hyphen.tex

As Don Knuth has declared that the filename `hyphen.tex` may only be used to designate *his* version of the american English hyphenation patterns, a new solution has to be found in order to be able to load hyphenation patterns for other languages in a plain-based \TeX -format. When asked he responded:

That file name is “sacred”, and if anybody changes it they will cause severe upward/downward compatibility headaches.

People can have a file `locallyhyphen.tex` or whatever they like, but they mustn’t diddle with `hyphen.tex` (or `plain.tex` except to preload additional fonts).

The files `bplain.tex` and `blplain.tex` can be used as replacement wrappers around `plain.tex` and `lplain.tex` to achieve the desired effect, based on the `babel` package. If you load each of them with \TeX , you will get a file called either `bplain.fmt` or `blplain.fmt`, which you can use as replacements for `plain.fmt` and `lplain.fmt`. As these files are going to be read as the first thing \TeX sees, we need to set some category codes just to be able to change the definition of `\input`

```

3302 (*bplain | blplain)
3303 \catcode`\{=1 % left brace is begin-group character
3304 \catcode`\}=2 % right brace is end-group character
3305 \catcode`\#=6 % hash mark is macro parameter character

```

Now let’s see if a file called `hyphen.cfg` can be found somewhere on \TeX ’s input path by trying to open it for reading...

```

3306 \openin 0 hyphen.cfg

```

If the file wasn't found the following test turns out true.

```
3307 \ifeof0
3308 \else
```

When `hyphen.cfg` could be opened we make sure that *it* will be read instead of the file `hyphen.tex` which should (according to Don Knuth's ruling) contain the american English hyphenation patterns and nothing else.

We do this by first saving the original meaning of `\input` (and I use a one letter control sequence for that so as not to waste multi-letter control sequence on this in the format).

```
3309 \let\input
```

Then `\input` is defined to forget about its argument and load `hyphen.cfg` instead.

```
3310 \def\input #1 {%
3311     \let\input\input
3312     \input{hyphen.cfg}
```

Once that's done the original meaning of `\input` can be restored and the definition of `\input` can be forgotten.

```
3313 \let\input\undefined
3314 }
3315 \fi
3316 </bplain | bplain>
```

Now that we have made sure that `hyphen.cfg` will be loaded at the right moment it is time to load `plain.tex`.

```
3317 <bplain>\input{plain.tex}
3318 <bplain>\input{lplain.tex}
```

Finally we change the contents of `\fmtname` to indicate that this is *not* the plain format, but a format based on plain with the `babel` package preloaded.

```
3319 <bplain>\def\fmtname{babel-plain}
3320 <bplain>\def\fmtname{babel-lplain}
```

When you are using a different format, based on `plain.tex` you can make a copy of `blplain.tex`, rename it and replace `plain.tex` with the name of your format file.

16.2 Emulating some \LaTeX features

The following code duplicates or emulates parts of $\text{\LaTeX} 2_{\epsilon}$ that are needed for `babel`.

```
3321 <*plain>
3322 \def\@empty{}
3323 \def\loadlocalcfg#1{%
3324     \openin0#1.cfg
3325     \ifeof0
3326         \closein0
3327     \else
3328         \closein0
3329         {\immediate\write16{*****}%
3330             \immediate\write16{* Local config file #1.cfg used}%
3331             \immediate\write16{*}%
3332         }
3333     \input{#1.cfg}\relax
3334 \fi
3335 \endoflfd}
```

16.3 General tools

A number of \LaTeX macro's that are needed later on.

```
3336 \long\def\@firstofone#1{#1}
3337 \long\def\@firstoftwo#1#2{#1}
3338 \long\def\@secondoftwo#1#2{#2}
3339 \def\@nnil{\@nil}
3340 \def\@gobbletwo#1#2{}
3341 \def\@ifstar#1{\@ifnextchar *{\@firstoftwo{#1}}}
3342 \def\@star@or@long#1{%
3343   \@ifstar
3344   {\let\l@ngrel@x\relax#1}%
3345   {\let\l@ngrel@x\long#1}}
3346 \let\l@ngrel@x\relax
3347 \def\@car#1#2\@nil{#1}
3348 \def\@cdr#1#2\@nil{#2}
3349 \let\@typeset@protect\relax
3350 \let\protected@edef\edef
3351 \long\def\@gobble#1{}
3352 \edef\@backslashchar{\expandafter\@gobble\string\}
3353 \def\strip@prefix#1>{}
3354 \def\g@addto@macro#1#2{%
3355   \toks@\expandafter{#1#2}%
3356   \xdef#1{\the\toks@}}
3357 \def\@namedef#1{\expandafter\def\csname #1\endcsname}
3358 \def\@nameuse#1{\csname #1\endcsname}
3359 \def\@ifundefined#1{%
3360   \expandafter\ifx\csname#1\endcsname\relax
3361     \expandafter\@firstoftwo
3362   \else
3363     \expandafter\@secondoftwo
3364   \fi}
3365 \def\@expandtwoargs#1#2#3{%
3366   \edef\reserved@a{\noexpand#1{#2}{#3}}\reserved@a}
3367 \def\zap@space#1 #2{%
3368   #1%
3369   \ifx#2\@empty\else\expandafter\zap@space\fi
3370   #2}
```

$\LaTeX 2_{\epsilon}$ has the command `\@onlypreamble` which adds commands to a list of commands that are no longer needed after `\begin{document}`.

```
3371 \ifx\@preamblecmds\@undefined
3372   \def\@preamblecmds{}
3373 \fi
3374 \def\@onlypreamble#1{%
3375   \expandafter\gdef\expandafter\@preamblecmds\expandafter{%
3376     \@preamblecmds\do#1}}
3377 \@onlypreamble\@onlypreamble
```

Mimick \LaTeX 's `\AtBeginDocument`; for this to work the user needs to add `\begindocument` to his file.

```
3378 \def\begindocument{%
3379   \@begindocumenthook
3380   \global\let\@begindocumenthook\@undefined
3381   \def\do##1{\global\let##1\@undefined}%
3382   \@preamblecmds
3383   \global\let\do\noexpand}
3384 \ifx\@begindocumenthook\@undefined
```

```

3385 \def\@begindocumenthook{}
3386 \fi
3387 \@onlypreamble\@begindocumenthook
3388 \def\AtBeginDocument{\g@addto@macro\@begindocumenthook}

```

We also have to mimic L^AT_EX's \AtEndOfPackage. Our replacement macro is much simpler; it stores its argument in \@endofldf.

```

3389 \def\AtEndOfPackage#1{\g@addto@macro\@endofldf{#1}}
3390 \@onlypreamble\AtEndOfPackage
3391 \def\@endofldf{}
3392 \@onlypreamble\@endofldf
3393 \let\bbl@afterlang\@empty
3394 \chardef\bbl@opt@hyphenmap\z@

```

L^AT_EX needs to be able to switch off writing to its auxiliary files; plain doesn't have them by default.

```

3395 \ifx\if@files\@undefined
3396 \expandafter\let\csname if@files\expandafter\endcsname
3397 \csname iffalse\endcsname
3398 \fi

```

Mimick L^AT_EX's commands to define control sequences.

```

3399 \def\newcommand{\@star@or@long\new@command}
3400 \def\new@command#1{%
3401 \@testopt{\@newcommand#1}0}
3402 \def\@newcommand#1[#2]{%
3403 \@ifnextchar [{\@xargdef#1[#2]}%
3404 {\@argdef#1[#2]}}
3405 \long\def\@argdef#1[#2]#3{%
3406 \@yargdef#1\@ne{#2}{#3}}
3407 \long\def\@xargdef#1[#2][#3]#4{%
3408 \expandafter\def\expandafter#1\expandafter{%
3409 \expandafter\@protected@testopt\expandafter #1%
3410 \csname\string#1\expandafter\endcsname{#3}}%
3411 \expandafter\@yargdef \csname\string#1\endcsname
3412 \tw@{#2}{#4}}
3413 \long\def\@yargdef#1#2#3{%
3414 \@tempcnta#3\relax
3415 \advance \@tempcnta \@ne
3416 \let\@hash@\relax
3417 \edef\reserved@a{\ifx#2\tw@ [\@hash@1]\fi}%
3418 \@tempcntb #2%
3419 \@whilenum\@tempcntb <\@tempcnta
3420 \do{%
3421 \edef\reserved@a{\reserved@a\@hash@\the\@tempcntb}%
3422 \advance\@tempcntb \@ne}%
3423 \let\@hash@###
3424 \l@ngrelx\expandafter\def\expandafter#1\reserved@a}
3425 \def\providecommand{\@star@or@long\provide@command}
3426 \def\provide@command#1{%
3427 \begingroup
3428 \escapechar\m@ne\xdef\@gtempa{\string#1}%
3429 \endgroup
3430 \expandafter\ifundefined\@gtempa
3431 {\def\reserved@a{\newcommand#1}}%
3432 {\let\reserved@a\relax
3433 \def\reserved@a{\newcommand\reserved@a}}%
3434 \reserved@a}%

```



```

3435 \def\DeclareRobustCommand{\@star@or@long\declare@robustcommand}
3436 \def\declare@robustcommand#1{%
3437   \edef\reserved@a{\string#1}%
3438   \def\reserved@b{#1}%
3439   \edef\reserved@b{\expandafter\strip@prefix\meaning\reserved@b}%
3440   \edef#1{%
3441     \ifx\reserved@a\reserved@b
3442       \noexpand\x@protect
3443       \noexpand#1%
3444     \fi
3445     \noexpand\protect
3446     \expandafter\newcommand\csname\bbl@stripslash#1 \endcsname
3447   }%
3448   \expandafter\newcommand\csname\bbl@stripslash#1 \endcsname
3449 }
3450 \def\x@protect#1{%
3451   \ifx\protect\@typeset@protect\else
3452     \@x@protect#1%
3453   \fi
3454 }
3455 \def\@x@protect#1\fi#2#3{%
3456   \fi\protect#1%
3457 }

```

The following little macro `\in@` is taken from `latex.ltx`; it checks whether its first argument is part of its second argument. It uses the boolean `\in@`; allocating a new boolean inside conditionally executed code is not possible, hence the construct with the temporary definition of `\bbl@tempa`.

```

3458 \def\bbl@tempa{\csname newif\endcsname\ifin@}
3459 \ifx\in@\@undefined
3460   \def\in@#1#2{%
3461     \def\in@@##1##2##3\in@@{%
3462       \ifx\in@@##2\in@false\else\in@true\fi}%
3463     \in@@#2#1\in@\in@@}
3464 \else
3465   \let\bbl@tempa\@empty
3466 \fi
3467 \bbl@tempa

```

\LaTeX has a macro to check whether a certain package was loaded with specific options. The command has two extra arguments which are code to be executed in either the true or false case. This is used to detect whether the document needs one of the accents to be activated (`activegrave` and `activeacute`). For plain \TeX we assume that the user wants them to be active by default. Therefore the only thing we do is execute the third argument (the code for the true case).

```

3468 \def\ifpackagewith#1#2#3#4{#3}

```

The \LaTeX macro `\@ifl@aded` checks whether a file was loaded. This functionality is not needed for plain \TeX but we need the macro to be defined as a no-op.

```

3469 \def\@ifl@aded#1#2#3#4{}

```

For the following code we need to make sure that the commands `\newcommand` and `\providecommand` exist with some sensible definition. They are not fully equivalent to their $\text{\LaTeX} 2_{\epsilon}$ versions; just enough to make things work in plain \TeX environments.

```

3470 \ifx\@tempcnta\@undefined
3471   \csname newcount\endcsname\@tempcnta\relax
3472 \fi
3473 \ifx\@tempcntb\@undefined

```

```

3474 \csname newcount\endcsname\@tempcntb\relax
3475 \fi

```

To prevent wasting two counters in L^AT_EX 2.09 (because counters with the same name are allocated later by it) we reset the counter that holds the next free counter (\count10).

```

3476 \ifx\bye\@undefined
3477 \advance\count10 by -2\relax
3478 \fi
3479 \ifx\@ifnextchar\@undefined
3480 \def\@ifnextchar#1#2#3{%
3481   \let\reserved@d=#1%
3482   \def\reserved@a{#2}\def\reserved@b{#3}%
3483   \futurelet\@let@token\@ifnch}
3484 \def\@ifnch{%
3485   \ifx\@let@token\@sptoken
3486     \let\reserved@c\@xifnch
3487   \else
3488     \ifx\@let@token\reserved@d
3489       \let\reserved@c\reserved@a
3490     \else
3491       \let\reserved@c\reserved@b
3492     \fi
3493   \fi
3494   \reserved@c}
3495 \def\:{\let\@sptoken= } \: % this makes \@sptoken a space token
3496 \def\:{\@xifnch} \expandafter\def\:{\futurelet\@let@token\@ifnch}
3497 \fi
3498 \def\@testopt#1#2{%
3499   \@ifnextchar[#{#1}{#1[#2]}}
3500 \def\@protected@testopt#1{%
3501   \ifx\protect\@typeset@protect
3502     \expandafter\@testopt
3503   \else
3504     \@x@protect#1%
3505   \fi}
3506 \long\def\@whilenum#1\do #2{\ifnum #1\relax #2\relax\@iwhilenum{#1\relax
3507   #2\relax}\fi}
3508 \long\def\@iwhilenum#1{\ifnum #1\expandafter\@iwhilenum
3509   \else\expandafter\@gobble\fi{#1}}

```

16.4 Encoding related macros

Code from `ltoutenc.dtx`, adapted for use in the plain T_EX environment.

```

3510 \def\DeclareTextCommand{%
3511   \@dec@text@cmd\providecommand
3512 }
3513 \def\ProvideTextCommand{%
3514   \@dec@text@cmd\providecommand
3515 }
3516 \def\DeclareTextSymbol#1#2#3{%
3517   \@dec@text@cmd\chardef#1{#2}#3\relax
3518 }
3519 \def\@dec@text@cmd#1#2#3{%
3520   \expandafter\def\expandafter#2%
3521     \expandafter{%
3522       \csname#3-cmd\expandafter\endcsname
3523       \expandafter#2%
3524       \csname#3\string#2\endcsname

```

```

3525     }%
3526 % \let@ifdefinable@rc@ifdefinable
3527 \expandafter#1\csname#3\string#2\endcsname
3528 }
3529 \def\@current@cmd#1{%
3530 \ifx\protect\@typeset@protect\else
3531 \noexpand#1\expandafter\@gobble
3532 \fi
3533 }
3534 \def\@changed@cmd#1#2{%
3535 \ifx\protect\@typeset@protect
3536 \expandafter\ifx\csname\cf@encoding\string#1\endcsname\relax
3537 \expandafter\ifx\csname ?\string#1\endcsname\relax
3538 \expandafter\def\csname ?\string#1\endcsname{%
3539 \@changed@x@err{#1}%
3540 }%
3541 \fi
3542 \global\expandafter\let
3543 \csname\cf@encoding \string#1\expandafter\endcsname
3544 \csname ?\string#1\endcsname
3545 \fi
3546 \csname\cf@encoding\string#1%
3547 \expandafter\endcsname
3548 \else
3549 \noexpand#1%
3550 \fi
3551 }
3552 \def\@changed@x@err#1{%
3553 \errhelp{Your command will be ignored, type <return> to proceed}%
3554 \errmessage{Command \protect#1 undefined in encoding \cf@encoding}}
3555 \def\DeclareTextCommandDefault#1{%
3556 \DeclareTextCommand#1?%
3557 }
3558 \def\ProvideTextCommandDefault#1{%
3559 \ProvideTextCommand#1?%
3560 }
3561 \expandafter\let\csname OT1-cmd\endcsname\@current@cmd
3562 \expandafter\let\csname?-cmd\endcsname\@changed@cmd
3563 \def\DeclareTextAccent#1#2#3{%
3564 \DeclareTextCommand#1{#2}[1]{\accent#3 #1}
3565 }
3566 \def\DeclareTextCompositeCommand#1#2#3#4{%
3567 \expandafter\let\expandafter\reserved@a\csname#2\string#1\endcsname
3568 \edef\reserved@b{\string##1}%
3569 \edef\reserved@c{%
3570 \expandafter\@strip@args\meaning\reserved@a:-\@strip@args}%
3571 \ifx\reserved@b\reserved@c
3572 \expandafter\expandafter\expandafter\ifx
3573 \expandafter\@car\reserved@a\relax\relax\@nil
3574 \@text@composite
3575 \else
3576 \edef\reserved@b##1{%
3577 \def\expandafter\noexpand
3578 \csname#2\string#1\endcsname####1{%
3579 \noexpand\@text@composite
3580 \expandafter\noexpand\csname#2\string#1\endcsname
3581 ####1\noexpand\@empty\noexpand\@text@composite
3582 {##1}%
3583 }%

```

```

3584         }%
3585         \expandafter\reserved@b\expandafter{\reserved@a{##1}}%
3586     \fi
3587     \expandafter\def\csname\expandafter\string\csname
3588       #2\endcsname\string#1-\string#3\endcsname{#4}
3589   \else
3590     \errhelp{Your command will be ignored, type <return> to proceed}%
3591     \errmessage{\string\DeclareTextCompositeCommand\space used on
3592       inappropriate command \protect#1}
3593   \fi
3594 }
3595 \def\@text@composite#1#2#3\@text@composite{%
3596   \expandafter\@text@composite@x
3597     \csname\string#1-\string#2\endcsname
3598 }
3599 \def\@text@composite@x#1#2{%
3600   \ifx#1\relax
3601     #2%
3602   \else
3603     #1%
3604   \fi
3605 }
3606 %
3607 \def\@strip@args#1:#2-#3\@strip@args{#2}
3608 \def\DeclareTextComposite#1#2#3#4{%
3609   \def\reserved@a{\DeclareTextCompositeCommand#1{#2}{#3}}%
3610   \bgroup
3611     \lccode` \@=#4%
3612     \lowercase{%
3613   \egroup
3614     \reserved@a @%
3615   }%
3616 }
3617 %
3618 \def\UseTextSymbol#1#2{%
3619 %   \let\@curr@enc\cf@encoding
3620 %   \@use@text@encoding{#1}%
3621 %   #2%
3622 %   \@use@text@encoding\@curr@enc
3623 }
3624 \def\UseTextAccent#1#2#3{%
3625 %   \let\@curr@enc\cf@encoding
3626 %   \@use@text@encoding{#1}%
3627 %   #2{\@use@text@encoding\@curr@enc\selectfont#3}%
3628 %   \@use@text@encoding\@curr@enc
3629 }
3630 \def\@use@text@encoding#1{%
3631 %   \edef\font@name{#1}%
3632 %   \xdef\font@name{%
3633 %     \csname\curr@fontshape/\font@size\endcsname
3634 %   }%
3635 %   \pickup@font
3636 %   \font@name
3637 %   \@@enc@update
3638 }
3639 \def\DeclareTextSymbolDefault#1#2{%
3640   \DeclareTextCommandDefault#1{\UseTextSymbol{#2}#1}%
3641 }
3642 \def\DeclareTextAccentDefault#1#2{%

```

```

3643 \DeclareTextCommandDefault#1{\UseTextAccent{#2}#1}%
3644 }
3645 \def\cf@encoding{OT1}

```

Currently we only use the $\LaTeX 2_{\epsilon}$ method for accents for those that are known to be made active in *some* language definition file.

```

3646 \DeclareTextAccent{"}{OT1}{127}
3647 \DeclareTextAccent{'}{OT1}{19}
3648 \DeclareTextAccent{^}{OT1}{94}
3649 \DeclareTextAccent`{OT1}{18}
3650 \DeclareTextAccent~{OT1}{126}

```

The following control sequences are used in `babel.def` but are not defined for plain \TeX .

```

3651 \DeclareTextSymbol{\textquotedblleft}{OT1}{92}
3652 \DeclareTextSymbol{\textquotedblright}{OT1}{`\"}
3653 \DeclareTextSymbol{\textquoteleft}{OT1}{`\'}
3654 \DeclareTextSymbol{\textquoteright}{OT1}{`'}
3655 \DeclareTextSymbol{\i}{OT1}{16}
3656 \DeclareTextSymbol{\ss}{OT1}{25}

```

For a couple of languages we need the \LaTeX -control sequence `\scriptsize` to be available. Because plain \TeX doesn't have such a sophisticated font mechanism as \LaTeX has, we just \let it to `\sevenrm`.

```

3657 \ifx\scriptsize\@undefined
3658 \let\scriptsize\sevenrm
3659 \fi

```

16.5 Babel options

The file `babel.def` expects some definitions made in the \LaTeX style file. So we must provide them at least some predefined values as well some tools to set them (even if not all options are available). There in no package options, and therefore an alternative mechanism is provided. For the moment, only `\babeloptionstrings` and `\babeloptionmath` are provided, which can be defined before loading `babel`. `\BabelModifiers` can be set too (but not sure it works).

```

3660 \let\bbl@opt@shorthands\@nnil
3661 \def\bbl@ifshorthand#1#2#3{#2}%
3662 \ifx\babeloptionstrings\@undefined
3663 \let\bbl@opt@strings\@nnil
3664 \else
3665 \let\bbl@opt@strings\babeloptionstrings
3666 \fi
3667 \def\bbl@tempa{normal}
3668 \ifx\babeloptionmath\bbl@tempa
3669 \def\bbl@mathnormal{\noexpand\textormath}
3670 \fi
3671 \def\BabelStringsDefault{generic}
3672 \ifx\BabelModifiers\@undefined\let\BabelModifiers\relax\fi
3673 \let\bbl@afterlang\relax
3674 \let\bbl@language@opts\@empty
3675 \ifx\@uclclist\@undefined\let\@uclclist\@empty\fi
3676 \def\AfterBabelLanguage#1#2{}
3677 </plain>

```

17 Acknowledgements

I would like to thank all who volunteered as β -testers for their time. Michel Goossens supplied contributions for most of the other languages. Nico Poppelier helped polish the text of the documentation and supplied parts of the macros for the Dutch language. Paul Wackers and Werenfried Spit helped find and repair bugs. During the further development of the babel system I received much help from Bernd Raichle, for which I am grateful.

References

- [1] Huda Smitshuijzen Abifares, *Arabic Typography*, Saqi, 2001.
- [2] Donald E. Knuth, *The T_EXbook*, Addison-Wesley, 1986.
- [3] Leslie Lamport, *L^AT_EX, A document preparation System*, Addison-Wesley, 1986.
- [4] K.F. Treebus. *Tekstwijzer, een gids voor het grafisch verwerken van tekst*. SDU Uitgeverij ('s-Gravenhage, 1988).
- [5] Hubert Partl, *German T_EX*, *TUGboat* 9 (1988) #1, p. 70–72.
- [6] Leslie Lamport, in: T_EXhax Digest, Volume 89, #13, 17 February 1989.
- [7] Johannes Braams, Victor Eijkhout and Nico Poppelier, *The development of national L^AT_EX styles*, *TUGboat* 10 (1989) #3, p. 401–406.
- [8] Yannis Haralambous, *Fonts & Encodings*, O'Reilly, 2007.
- [9] Joachim Schrod, *International L^AT_EX is ready to use*, *TUGboat* 11 (1990) #1, p. 87–90.
- [10] Apostolos Syropoulos, Antonis Tsolomitis and Nick Sofroniu, *Digital typography using L^AT_EX*, Springer, 2002, p. 301–373.