Babel, a multilingual package for use with LaTeX's standard document classes*

Johannes Braams
Kersengaarde 33
2723 BP Zoetermeer
The Netherlands
For version 3.9, Javier Bezos
www.tex-tipografia.com

Typeset May 16, 2013

Abstract

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 report 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. However, no attempt has been done to take full advantage of the features provided by the latter, which would require a completely new core (as for example polyglossia or as part of LaTeX3).

Contents

*During the development ideas from Nico Poppelier, Piet van Oostrum and many others have been used. Bernd Raichle has provided many helpful suggestions.

1 The user interface

The user interface of this package is quite simple. It consists of a set of commands that switch from one language to another, and a set of commands that deal with shorthands. It is also possible to find out what the current language is.
In LaTeX2e 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}
```

Another approach is making dutch and english global options in order to let other packages detect and use them:

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

In this last example, the package `varioref` will also see the options and will be able to use them.
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]{babel}
\usepackage[ngerman,main=italian]{babel}
```

New 3.9c The basic behaviour of some languages can be modified when loading babel. Modifiers 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 it can be written closed, too):

```
\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 modifiers list. However, modifiers is a more general mechanism. Currently babel provides no standard interface for scripts. Languages sharing the same non-Latin script may define macros to switch them (eg, `\textcyrillic`), but be aware they may also set the language to a certain default. (Future versions might add such an interface.)
Because of the way babel has evolved, "language" can refer to (1) a set of hyphenation patters 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.
Loading directly `sty` files in LaTeX (ie, `\usepackage{`⟨*language*⟩`}`) is deprected and you will get the error "You have used an old interface to call babel".
In Plain, load languages styles with `\input` and then use `\begindocument` (the latter is defined by babel):

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

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

1.1 Selecting languages

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.
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, like braces `{}`.
This command can be used as environment, too.

`\begin{otherlanguage}` `{`⟨*language*⟩`}` ... `\end{otherlanguage}`

The environment otherlanguage does basically the same as `\selectlanguage`, except the language change is (mostly) local to the environment. This environment is required for intermixing left-to-right typesetting with right-to-left typesetting. The language to switch to is specified as an argument to `\begin{otherlanguage}`.
Actually, there might be some non-local changes, as this environment is roughly equivalent to:

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

If you want a change which is really local, you must enclose this environment with and additional grouping, like braces `{}`.
Spaces after the environment are ignored.

`\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 argument. 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).

\begin{otherlanguage*} {⟨*language*⟩} ... \end{otherlanguage*}

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

\languagename The control sequence \languagename contains the name of the current language. However, due to some internal inconsistencies in catcodes it should *not* be used to test its value (use iflang, by Heiko Oberdiek).

\iflanguage {⟨*language*⟩}{⟨*true*⟩}{⟨*false*⟩}

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

\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 or characters like, say, ' done by some languages (eg, italian, frenchb, ukraineb). To set hyphenation exceptions, use \babelhyphenation (see below).

1.2 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 discretionaries and breaks can be inserted easily with "-, "=, etc.
The package inputenc as well as xetex an luatex have alleviated entering non-ASCII characters, but minority languages and some texts can still require characters not directly available in the keyboards (and sometimes not even as separated or precomposed Unicode characters). As to the point 2, now pdfTeX provides \knbccode. Tools of point 3 can be still very useful in general.
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.

Please, 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.

`\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 3.9 Note 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.

`\useshorthands` `*{`⟨*char*⟩`}`

The command `\useshorthands` initiates the definition of user-defined shorthand sequences. It has one argument, the character that starts these personal shorthands.
New 3.9 However, 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 `\useshorthands*{`⟨*char*⟩`}` is provided, which makes sure shorthands are always activated.
Currently, if the option `shorthands` is used, you must include any character to be activated with `\useshorthands`. This restriction will be lifted in a future release.

`\defineshorthand` `[`⟨*language*⟩`,`⟨*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 3.9 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*⟩). 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.
As an example of their applications, 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:

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

However, behaviour of hyphens is language dependent. For example, in languages like Polish and Portugese, a hard hyphen inside compound words are repeated at the beginning of the next line. You could 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 overriden 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{"}{/}`. *Please note* the substitute character must *not* have been declared before as shorthand (in such case, `\aliashorthands` is ignored).
The following example shows how to replace a shorthand by another

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

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

`\languageshorthands` `{`⟨*language*⟩`}`

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).[1] 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}}
```

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` `{`⟨*shorthand*⟩`}`

With this command you can use shorthands 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 `\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 ovelap.)
For your records, here is a list of shorthands, but you must check them, as they may change:[2]

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 `" . < > '`

[1] Actually, 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.
[2] Thanks to Enrico Gregorio

Turkish `: ! =`

In addition, the babel core declares `~` as a one-char shorthand which is let, like the standard `~`, to a non breaking space.[3]

1.3 Package options

New 3.9 These package options are processed before language options, so that they are taken into account irrespective of its order.

`shorthands=` ⟨*char*⟩⟨*char*⟩... | `off`

The only language shorthands activated are those given, like, eg:

```
\usepackage[esperanto,frenchb,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 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. Of course, in such a case you cannot use shorthands in these macros.

`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=` ⟨*file*⟩

Instead of loading `bblopts.cfg`, the file ⟨*file*⟩`.cfg` is loaded.

`main=` ⟨*language*⟩

Sets the main language, as explained above, ie, this language is always the last loaded. The language may be also given as package or global option or not.

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. The key `config` still works.

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.

strings= `generic` | `unicode` | `encoded` | ⟨*label*⟩ | ⟨*font encoding*⟩

Selects the encoding of strings in languages supporting this feature. Predefined labels are `generic` (for traditional TEX), `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.

(The following three options have been available in previous versions of babel.)

KeepShorthandsActive Tells babel not to deactivate shorthands after loading a language file, so that they are also availabe 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 `.

1.4 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.

\AfterBabelLanguage `{`⟨*option-name*⟩`}{`⟨*code*⟩`}`

This command is currently the only provided by `base`. Executes ⟨*code*⟩ when the file loaded by the corresponding package option is finished (at `\ldf@finish`). The setting is global. So

```
\AfterBabelLanguage{frenchb}{...}
```

[3] This declaration serves to nothing, but it is preserved for backward compatibility.

does ... at the end of `frenchb.ldf`. It can be used in `ldf` files, too, but in such a case the code is executed only if ⟨*option-name*⟩ is the same as `\CurrentOption` (which could not be the same as the option name as set in `\usepackage`!).
For 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}
```

1.5 Hooks

New 3.9 A hook is 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*⟩`}`
Current events are: `adddialect`, `write`, `beforeextras`, `afterextras`, `patterns`, `hyphenation`, `defaultcommands`, `encodedcommands`, `stopcommands` and `stringprocess`. Four events are used in `hyphen.cfg`, which are handled in a quite different way for efficiency reasons – these are `everylanguage`, `loadkernel`, `loadpatterns` and `loadexceptions`, which, unlike the precedent ones, only have a single hook and replace a default definition. (Names containing the string `babel` are reserved; they are used, for example, by `\useshortands*` to add a hook for the event `afterextras`.)
In some of them you can use one or two TEX parameters (`#1`, `#2`), with the meaning given below:

`everylanguage` (language) Executed before 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`.
`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` Intead 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}}
```

1.6 Hyphen tools

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

New 3.9 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 oportunity or, in TEX terms, a "discretionary"; a *hard hyphen* is a hyphen with a breaking oportunity after it. A further type is a *non-breaking hyphen*, a hyphen without a breaking oportunity. In TEX, `-` and `\-` forbid further breaking oportunities 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 oportunities in the rest of the word.
Therefore, some macros are provide 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 oportunity 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 `\babelnullhyphen`; (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` `[`⟨*language*⟩`,`⟨*language*⟩`,...]{`⟨*exceptions*⟩`}`

New 3.9 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 script 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 encondings are the same, like in Unicode based engines.

1.7 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.
Several language definition files use their own methods to set options. For example, frenchb uses `\frenchbsetup`, magyar (1.5) uses `\magyarOptions` and spanish a set of package options (eg, `es-nolayout`). Macros settting options are also used (eg, `\ProsodicMarksOn` in latin).

1.8 Languages supported by babel

In the following table most of the languages supported by babel are listed, together with the names of the options which you can load babel with for each language. Note

this list is open and the current options may be different.

Afrikaans afrikaans
Bahasa bahasa, indonesian, indon, bahasai, bahasam, malay, meyalu
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
Interlingua interlingua
Irish Gaelic irish
Italian italian
Latin latin
Lower Sorbian lowersorbian
North Sami samin
Norwegian norsk, nynorsk
Polish polish
Portuguese portuges, 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 your have 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.

1.9 Tips, workarounds and know issues

- 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 hhline 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 hhline (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 TeX only takes into account the values when the paragraph is hyphenated, i.e., when it has been finished.[4] So, if you write a chunk of French text with `\foreignlanguage`,

[4] This explains why LaTeX assumes the lowercase mapping of T1 and does not provide a tool for multiple mappings. Unfortunately, `\savinghyphcodes` is not a solution either, because lccodes for hyphenation are frozen in the format and cannot be changed.

the apostrophes might not be taken into account. This is a limitation of TEX, not of babel. Alternatively, you may use `\useshorthands` to activate `'` and `\defineshorthand`, 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 TEX enter in an infinite loop. (Another issue in the 'to do' list, although there is a partial solution.)
- Also in the 'to do' list is a common interface to switch scripts, to avoid the current problem of languages trying to define `\text`⟨*script*⟩ in different ways.

2 Preloading 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 loaded when a format is created (eg, LATEX, XeLATEX, pdfLATEX). babel provides a tool which has become standand 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).
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 stored[5]. 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:

```
% 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
```

[5]This is because different operating systems sometimes use *very* different file-naming conventions.

You may also set the font encoding the patterns are intended for by following the language name by a colon and the encoding code.[6] For example:

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

With the previous settings, if the enconding 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 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.

[6] This in not a new feature, but in former versions it didn't work correctly.

- 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 unknow language is selected, babel will attempt setting it after lowercasing its name.

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).

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 (and the corresponding PDF, if you like), as well as other files you think can be useful (eg, samples, readme).

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 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.

`\<lang>hyphenmins` 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 (currenty, 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 thing 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@ttribute` 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.

```
\ProvidesLanguage{<language>}
     [0000/00/00 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@@}

\def\captions<language>{}
\let\captions<dialect>\captions<language>

\def\date<language>{}
\def\date<dialect>{}

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

`\bbl@deactivate` `\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 TEXbook 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." [1, 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 `\dospecial`. 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 *re*define macros that already exist. Therefor a mechanism for saving (and restoring) the original definition of those macros is provided. We provide two macros for this[7].

`\babel@save` To save the current meaning of any control sequence, the macro `\babel@save` is provided. It takes one argument, ⟨*csname*⟩, 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 seems 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

[7] This mechanism was introduced by Bernd Raichle.

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

3.7 Encoding-dependent strings

New 3.9 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 with `\UseStrings`, see 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 german, just redefine `\germanchaptername`.

`\StartBabelCommands` `{`⟨*language-list*⟩`}{`⟨*category*⟩`}[`⟨*selector*⟩`]`

The ⟨*language-list*⟩ specifies which languages the block is intended for. A block is taken into account only if the `\CurrentOption` is listed here.
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).
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, `?`). If there is no key `strings`, string definitions are ignored, but `\SetCase`s are still honoured (an a `encoded` way).
The ⟨*category*⟩ is either `captions`, `date` or `extras`. You must stick to these three categories, even if no error is raised when using other name.[8] It may be empty, too, but in such a case using `\SetString` is an error (but not `\SetCase`).

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

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

\EndBabelCommands
```

A real example is:

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

\StartBabelCommands{german,austrian}{date}
  [unicode, fontenc=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\monthiiname{Februar}
  \SetString\monthiiiname{M\"{a}rz}
  \SetString\monthivname{April}
  \SetString\monthvname{Mai}
  \SetString\monthviname{Juni}
  \SetString\monthviiname{Juli}
  \SetString\monthviiiname{August}
  \SetString\monthixname{September}
```

[8] In future releases further categories may be added.

```
  \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 \⟨*category*⟩⟨*language*⟩ are overriden, 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 `\date`⟨*language*⟩ exists).

`\EndBabelCommands` Marks the end of the series of blocks.

`\UseStrings` You may also want to omit the old way of defining define strings altogether. Just add `\UseStrings` after the first `\StartBabelCommands` and the generic branchs will be taken into account even if there is no `strings` key. This directive applies to all subsequent blocks, until `\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 intented for minor readjustments only.
For example, as `T1` is the default case mapping in LaTeX, we could set for Turkish:

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

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

\StartBabelCommands{turkish}{}
\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.)

4 Compatibility and changes

4.1 Compatibility with `german.sty`

The file `german.sty` has been one of the sources of inspiration for the babel system. Because of this I wanted to include `german.sty` in the babel system. To be able to do that I had to allow for one incompatibility: in the definition of the macro `\selectlanguage` in `german.sty` the argument is used as the ⟨*number*⟩ for an `\ifcase`. So in this case a call to `\selectlanguage` might look like `\selectlanguage{\german}`.
In the definition of the macro `\selectlanguage` in `babel.def` the argument is used as a part of other macronames, so a call to `\selectlanguage` now looks like `\selectlanguage{german}`. Notice the absence of the escape character. As of version 3.1a of babel both syntaxes are allowed.
All other features of the original `german.sty` have been copied into a new file, called `germanb.sty`[9].

[9] The ‘b’ is added to the name to distinguish the file from Partls’ file.

Although the babel system was developed to be used with LaTeX, some of the features implemented in the language definition files might be needed by plain TeX users. Care has been taken that all files in the system can be processed by plain TeX.

4.2 Compatibility with `ngerman.sty`

When used with the options ngerman or naustrian, babel will provide all features of the package `ngerman`. There is however one exception: The commands for special hyphenation of double consonants (`"ff` etc.) and ck (`"ck`), which are no longer required with the new German orthography, are undefined. With the `ngerman` package, however, these commands will generate appropriate warning messages only.

4.3 Compatibility with the `french` package

It has been reported to me that the package `french` by Bernard Gaulle (`gaulle@idris.fr`) works together with babel. On the other hand, it seems *not* to work well together with a lot of other packages. Therefore I have decided to no longer load `french.ldf` by default. Instead, when you want to use the package by Bernard Gaulle, you will have to request it specifically, by passing either frenchle or frenchpro as an option to babel.

4.4 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 correspondin place.

4.5 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 Greek has been enhanced. Code from the `kdgreek` package (suggested by the author) was added and `\greeknumeral` has been added.
- Support for typesetting Basque is now available thanks to Juan Aguirregabiria.

- Support for typesetting Serbian with Latin script is now available thanks to Dejan Muhamedagić and Jankovic Slobodan.
- 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.
- Support for typesetting Bulgarian is now available thanks to Georgi Boshnakov.
- Support for typesetting Latin is now available, thanks to Claudio Beccari and Krzysztof Konrad Żelechowski.
- Support for typesetting North Sami is now available, thanks to Regnor Jernsletten.
- The options canadian, canadien and acadien have been added for Canadian English and French use.
- 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* ("Polutoniko" or multi-accented) Greek way of typesetting texts. These attributes will possibly find wider use in future releases.
- 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.

4.6 Changes in babel version 3.6

In babel version 3.6 a number of bugs that were found in version 3.5 are fixed. Also a number of changes and additions have occurred:

- A new environment otherlanguage* is introduced. it only switches the 'specials', but leaves the 'captions' untouched.

- The shorthands are no longer fully expandable. Some problems could only be solved by peeking at the token following an active character. The advantage is that `'{}a` works as expected for languages that have the `'` active.
- Support for typesetting french texts is much enhanced; the file `francais.ldf` is now replaced by `frenchb.ldf` which is maintained by Daniel Flipo.
- Support for typesetting the russian language is again available. The language definition file was originally developed by Olga Lapko from CyrTUG. The fonts needed to typeset the russian language are now part of the babel distribution. The support is not yet up to the level which is needed according to Olga, but this is a start.
- Support for typesetting greek texts is now also available. What is offered in this release is a first attempt; it will be enhanced later on by Yannis Haralambous.
- in babel 3.6j some hooks have been added for the development of support for Hebrew typesetting.
- Support for typesetting texts in Afrikaans (a variant of Dutch, spoken in South Africa) has been added to `dutch.ldf`.
- Support for typesetting Welsh texts is now available.
- A new command `\aliasshorthand` is introduced. It seems that in Poland various conventions are used to type the necessary Polish letters. It is now possible to use the character `/` as a shorthand character instead of the character `"`, by issuing the command `\aliasshorthand{"}{/}`.
- The shorthand mechanism now deals correctly with characters that are already active.
- Shorthand characters are made active at `\begin{document}`, not earlier. This is to prevent problems with other packages.
- A *preambleonly* command `\substitutefontfamily` has been added to create `.fd` files on the fly when the font families of the Latin text differ from the families used for the Cyrillic or Greek parts of the text.
- Three new commands `\LdfInit`, `\ldf@quit` and `\ldf@finish` are introduced that perform a number of standard tasks.
- In babel 3.6k the language Ukrainian has been added and the support for Russian typesetting has been adapted to the package 'cyrillic' to be released with the December 1998 release of LaTeX 2_ε.

4.7 Changes in babel version 3.5

In babel version 3.5 a lot of changes have been made when compared with the previous release. Here is a list of the most important ones:

- the selection of the language is delayed until `\begin{document}`, which means you must add appropriate `\selectlanguage` commands if you include `\hyphenation` lists in the preamble of your document.
- babel now has a language environment and a new command `\foreignlanguage`;
- the way active characters are dealt with is completely changed. They are called 'shorthands'; one can have three levels of shorthands: on the user level, the language level, and on 'system level'. A consequence of the new way of handling active characters is that they are now written to auxiliary files 'verbatim';
- A language change now also writes information in the `.aux` file, as the change might also affect typesetting the table of contents. The consequence is that an `.aux` file generated by a LaTeX format with babel preloaded gives errors when read with a LaTeX format without babel; but I think this probably doesn't occur;
- babel is now compatible with the `inputenc` and `fontenc` packages;
- the language definition files now have a new extension, `ldf`;
- the syntax of the file `language.dat` is extended to be compatible with the `french` package by Bernard Gaulle;
- each language definition file looks for a configuration file which has the same name, but the extension `.cfg`. It can contain any valid LaTeX code.

5 Identification and loading of required files

Code documentation is still under revision.
The file `babel.sty`[10] is meant for LaTeX 2_ε, therefor we make sure that the format file used is the right one.

`\ProvidesLanguage` The identification code for each file is something that was introduced in LaTeX 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.

```
1 ⟨*kernel⟩
2 \def\bbl@version{3.9f}
3 \def\bbl@date{2013/05/16}
4 ⟨/kernel⟩
```

[10] The file described in this section is called `babel.dtx`, has version number v3.9f and was last revised on 2013/05/16.

```
⟨*patterns⟩
\xdef\bbl@format{\jobname}
⟨/patterns⟩
⟨*core | kernel | patterns⟩
\ifx\ProvidesFile\@undefined
  \def\ProvidesFile#1[#2 #3 #4]{%
    \wlog{File: #1 #4 #3 <#2>}%
    \let\ProvidesFile\@undefined
    }
⟨/core | kernel | patterns⟩
```

As an alternative for `\ProvidesFile` we define `\ProvidesLanguage` here to be used in the language definition files.

```
⟨*kernel⟩
  \def\ProvidesLanguage#1[#2 #3 #4]{%
    \wlog{Language: #1 #4 #3 <#2>}%
    }
\else
```

When `\ProvidesFile` is defined we give `\ProvidesLanguage` a similar definition.

```
  \def\ProvidesLanguage#1{%
    \begingroup
      \catcode`\ 10 %
      \@makeother\/%
      \@ifnextchar[%]
        {\@provideslanguage{#1}}{\@provideslanguage{#1}[]}}
  \def\@provideslanguage#1[#2]{%
    \wlog{Language: #1 #2}%
    \expandafter\xdef\csname ver@#1.ldf\endcsname{#2}%
    \endgroup}
⟨/kernel⟩
⟨*core | kernel | patterns⟩
\fi
⟨/core | kernel | patterns⟩
```

Identify each file that is produced from this source file.

```
⟨package⟩\ProvidesPackage{babel}
⟨core⟩\ProvidesFile{babel.def}
⟨patterns⟩\ProvidesFile{hyphen.cfg}
⟨kernel⟩\ProvidesFile{switch.def}
⟨nil⟩\ProvidesLanguage{nil}
⟨driver&!user⟩\ProvidesFile{babel.drv}
⟨driver & user⟩\ProvidesFile{user.drv}
    [2013/05/16 v3.9f %
⟨package⟩     The Babel package]
⟨core⟩     Babel common definitions]
⟨kernel | patterns⟩     Babel language switching mechanism]
⟨nil⟩    Nil language]
⟨driver⟩]
```

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 LaTeX2.09. 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).

```
⟨*core | kernel | patterns⟩
\ifx\AtBeginDocument\@undefined
⟨/core | kernel | patterns⟩
⟨*patterns⟩
  \let\orig@dump\dump
⟨/patterns⟩
⟨*core | kernel | patterns⟩
  \input plain.def\relax
\fi
⟨/core | kernel | patterns⟩
```

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.

```
⟨*package⟩
\@ifpackagewith{babel}{debug}
  {\let\bbl@tempa\relax}
  {\def\bbl@tempa{3.9f}}%
⟨/package⟩
⟨*core⟩
\def\bbl@tempa{3.9f}
⟨/core⟩
⟨*core | package⟩
\ifx\bbl@version\bbl@tempa\else
  \input switch.def\relax
\fi
```

The following macros just make the code cleaner. `\bbl@add` is now used internally instead of `\addto` because of the unpredictable behaviour of the latter. (There are some duplications, but, you know, I [JBL] never rememer where the code ends up when intermingled with docstrip.)

```
\def\bbl@for#1#2#3{\@for#1:=#2\do{\ifx#1\@empty\else#3\fi}}
\def\bbl@add#1#2{%
  \@ifundefined{\expandafter\@gobble\string#1}%
    {\def#1{#2}}%
    {\expandafter\def\expandafter#1\expandafter{#1#2}}}
\def\bbl@csarg#1#2{\expandafter#1\csname bbl@#2\endcsname}%
\long\def\bbl@afterelse#1\else#2\fi{\fi#1}
\long\def\bbl@afterfi#1\fi{\fi#1}
⟨/core | package⟩
```

6 The Package File

In order to make use of the features of LaTeX 2$_\varepsilon$, 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 an defines a few aditional 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.

6.1 `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.

```
⟨*package⟩
\def\AfterBabelLanguage#1{%
  \global\expandafter\bbl@add\csname#1.ldf-h@@k\endcsname}%
\@ifpackagewith{babel}{base}{%
  \DeclareOption*{\bbl@patterns{\CurrentOption}}%
  \DeclareOption{base}{}%
  \ProcessOptions
  \global\expandafter\let\csname opt@babel.sty\endcsname\relax
  \global\expandafter\let\csname ver@babel.sty\endcsname\relax
  \global\let\@ifl@ter@@\@ifl@ter
  \def\@ifl@ter#1#2#3#4#5{\global\let\@ifl@ter\@ifl@ter@@}%
  \endinput}{}%
```

6.2 `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` o load `keyval`).

```
\bbl@csarg\let{tempa\expandafter}\csname opt@babel.sty\endcsname
\def\bbl@tempb#1.#2{%
   #1\ifx\@empty#2\else,\bbl@afterfi\bbl@tempb#2\fi}%
\def\bbl@tempd#1.#2\@nnil{%
  \ifx\@empty#2%
    \edef\bbl@tempc{\ifx\bbl@tempc\@empty\else\bbl@tempc,\fi#1}%
  \else
    \in@{=}{#1}\ifin@
      \edef\bbl@tempc{\ifx\bbl@tempc\@empty\else\bbl@tempc,\fi#1.#2}%
```

```
    \else
      \edef\bbl@tempc{\ifx\bbl@tempc\@empty\else\bbl@tempc,\fi#1}%
      \bbl@csarg\edef{mod@#1}{\bbl@tempb#2}%
    \fi
  \fi}
\let\bbl@tempc\@empty
\bbl@for\bbl@tempa\bbl@tempa{%
  \expandafter\bbl@tempd\bbl@tempa.\@empty\@nnil}
\expandafter\let\csname opt@babel.sty\endcsname\bbl@tempc
\DeclareOption{activeacute}{}
\DeclareOption{activegrave}{}
```

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.

```
\DeclareOption{KeepShorthandsActive}{}
\DeclareOption{debug}{}
\DeclareOption{noconfigs}{}
\DeclareOption{showlanguages}{}
\DeclareOption{shorthands=off}{\bbl@tempa shorthands=\bbl@tempa}
\DeclareOption{strings}{\def\bbl@opt@strings{\BabelStringsDefault}}
\DeclareOption{strings=encoded}{\let\bbl@opt@strings\relax}
\DeclareOption{safe=none}{\let\bbl@opt@safe\@empty}
\DeclareOption{safe=bib}{\def\bbl@opt@safe{B}}
\DeclareOption{safe=ref}{\def\bbl@opt@safe{R}}
\DeclareOption{math=active}{}
\DeclareOption{math=normal}{\def\bbl@mathnormal{\noexpand\textormath}}
\def\BabelStringsDefault{generic}
```

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.

```
\let\bbl@opt@shorthands\@nnil
\let\bbl@opt@config\@nnil
\let\bbl@opt@main\@nnil
\let\bbl@opt@strings\@nnil
\let\bbl@opt@headfoot\@nnil
```

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

```
\def\bbl@tempa#1=#2\bbl@tempa{%
  \expandafter\ifx\csname bbl@opt@#1\endcsname\@nnil
    \expandafter\edef\csname bbl@opt@#1\endcsname{#2}%
  \else
    \bbl@error{%
      Bad option `#1=#2'. Either you have misspelled the\\%
      key or there is a previous setting of `#1'}{%
```

```
      Valid keys are ‘shorthands’, ‘config’, ‘strings’, ‘main’,\\%
      ‘headfoot’, ‘safe’, ‘math’}
  \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.

```
\let\bbl@language@opts\@empty
\DeclareOption*{%
  \@expandtwoargs\in@{\string=}{\CurrentOption}%
  \ifin@
    \expandafter\bbl@tempa\CurrentOption\bbl@tempa
  \else
    \edef\bbl@language@opts{%
      \ifx\bbl@language@opts\@empty\else\bbl@language@opts,\fi
      \CurrentOption}%
  \fi}
```

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

```
\ProcessOptions*
```

6.3 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=...`.

```
\def\bbl@sh@string#1{%
  \ifx#1\@empty\else
    \ifx#1t\string~%
    \else\ifx#1c\string,%
    \else\string#1%
    \fi\fi
    \expandafter\bbl@sh@string
  \fi}
\ifx\bbl@opt@shorthands\@nnil
  \def\bbl@ifshorthand#1#2#3{#2}%
\else\ifx\bbl@opt@shorthands\@empty
  \def\bbl@ifshorthand#1#2#3{#3}%
\else
```

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

```
  \def\bbl@ifshorthand#1{%
    \@expandtwoargs\in@{\string#1}{\bbl@opt@shorthands}%
    \ifin@
      \expandafter\@firstoftwo
    \else
      \expandafter\@secondoftwo
```

```
    \fi}
```

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

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

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

```
  \bbl@ifshorthand{'}%
    {\PassOptionsToPackage{activeacute}{babel}}{}
  \bbl@ifshorthand{`}%
    {\PassOptionsToPackage{activegrave}{babel}}{}
\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.

```
\ifx\bbl@opt@headfoot\@nnil\else
  \g@addto@macro\@resetactivechars{%
    \set@typeset@protect
    \expandafter\select@language@x\expandafter{\bbl@opt@headfoot}%
    \let\protect\noexpand}
\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.

```
\@ifundefined{bbl@opt@safe}{\def\bbl@opt@safe{BR}}{}
\ifx\bbl@opt@main\@nnil\else
  \edef\bbl@language@opts{%
    \ifx\bbl@language@opts\@empty\else\bbl@language@opts,\fi
      \bbl@opt@main}
\fi
```

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.

```
\ifx\bbl@languages\@undefined\else
  \begingroup
    \catcode`\^^I=12
    \@ifpackagewith{babel}{showlanguages}{%
      \begingroup
        \def\bbl@elt#1#2#3#4{\wlog{#2^^I#1^^I#3^^I#4}}%
        \wlog{<*languages>}%
        \bbl@languages
        \wlog{</languages>}%
      \endgroup}{}
  \endgroup
  \def\bbl@elt#1#2#3#4{%
    \ifnum#2=\z@
      \gdef\bbl@nulllanguage{#1}%
      \def\bbl@elt##1##2##3##4{}%
```

```
    \fi}%
  \bbl@languages
\fi
```

6.4 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 catched).

```
\let\bbl@afterlang\relax
\let\BabelModifiers\relax
\let\bbl@loaded\@empty
\def\bbl@load@language#1{%
  \InputIfFileExists{#1.ldf}%
    {\edef\bbl@loaded{\CurrentOption
       \ifx\bbl@loaded\@empty\else,\bbl@loaded\fi}%
     \expandafter\let\expandafter\bbl@afterlang
        \csname\CurrentOption.ldf-h@@k\endcsname
     \expandafter\let\expandafter\BabelModifiers
        \csname bbl@mod@\CurrentOption\endcsname}%
    {\bbl@error{%
       Unknow option `\CurrentOption'. Either you misspelled it\\%
       or the language definition file \CurrentOption.ldf was not found}{%
       Valid options are: shorthands=..., KeepShorthandsActive,\\%
       activeacute, activegrave, noconfigs, safe=, main=, math=\\%
       headfoot=, strings=, config=, or a language name.}}}
```

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

```
\DeclareOption{acadian}{\bbl@load@language{frenchb}}
\DeclareOption{afrikaans}{\bbl@load@language{dutch}}
\DeclareOption{american}{\bbl@load@language{english}}
\DeclareOption{australian}{\bbl@load@language{english}}
\DeclareOption{austrian}{\bbl@load@language{germanb}}
\DeclareOption{bahasa}{\bbl@load@language{bahasai}}
\DeclareOption{bahasai}{\bbl@load@language{bahasai}}
\DeclareOption{bahasam}{\bbl@load@language{bahasam}}
\DeclareOption{brazil}{\bbl@load@language{portuges}}
\DeclareOption{brazilian}{\bbl@load@language{portuges}}
\DeclareOption{british}{\bbl@load@language{english}}
\DeclareOption{canadian}{\bbl@load@language{english}}
\DeclareOption{canadien}{\bbl@load@language{frenchb}}
\DeclareOption{francais}{\bbl@load@language{frenchb}}
\DeclareOption{french}{\bbl@load@language{frenchb}}%
\DeclareOption{german}{\bbl@load@language{germanb}}
\DeclareOption{hebrew}{%
  \input{rlbabel.def}%
  \bbl@load@language{hebrew}}
\DeclareOption{hungarian}{\bbl@load@language{magyar}}
```

```
\DeclareOption{indon}{\bbl@load@language{bahasai}}
\DeclareOption{indonesian}{\bbl@load@language{bahasai}}
\DeclareOption{lowersorbian}{\bbl@load@language{lsorbian}}
\DeclareOption{malay}{\bbl@load@language{bahasam}}
\DeclareOption{meyalu}{\bbl@load@language{bahasam}}
\DeclareOption{naustrian}{\bbl@load@language{ngermanb}}
\DeclareOption{newzealand}{\bbl@load@language{english}}
\DeclareOption{ngerman}{\bbl@load@language{ngermanb}}
\DeclareOption{nynorsk}{\bbl@load@language{norsk}}
\DeclareOption{polutonikogreek}{%
  \bbl@load@language{greek}%
  \languageattribute{greek}{polutoniko}}
\DeclareOption{portuguese}{\bbl@load@language{portuges}}
\DeclareOption{russian}{\bbl@load@language{russianb}}
\DeclareOption{UKenglish}{\bbl@load@language{english}}
\DeclareOption{ukrainian}{\bbl@load@language{ukraineb}}
\DeclareOption{uppersorbian}{\bbl@load@language{usorbian}}
\DeclareOption{USenglish}{\bbl@load@language{english}}
```

Another way to extend the list of 'known' options for babel is 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.

```
\ifx\bbl@opt@config\@nnil
  \@ifpackagewith{babel}{noconfigs}{}%
    {\InputIfFileExists{bblopts.cfg}%
      {\typeout{**************************************^^J%
               * Local config file bblopts.cfg used^^J%
               *}}%
      {}}%
\else
  \InputIfFileExists{\bbl@opt@config.cfg}%
    {\typeout{**************************************^^J%
             * Local config file \bbl@opt@config.cfg used^^J%
             *}}%
    {\bbl@error{%
       Local config file `\bbl@opt@config.cfg' not found}{%
       Perhaps you misspelled it.}}%
\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.

```
\bbl@for\bbl@tempa\bbl@language@opts{%
  \@ifundefined{ds@\bbl@tempa}%
    {\edef\bbl@tempb{%
       \noexpand\DeclareOption
```

```
          {\bbl@tempa}%
          {\noexpand\bbl@load@language{\bbl@tempa}}}%
      \bbl@tempb}%
      \@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 accesing the file system just to see if the option could be a language.

```
\bbl@for\bbl@tempa\@classoptionslist{%
  \@ifundefined{ds@\bbl@tempa}%
    {\IfFileExists{\bbl@tempa.ldf}%
      {\edef\bbl@tempb{%
         \noexpand\DeclareOption
           {\bbl@tempa}%
           {\noexpand\bbl@load@language{\bbl@tempa}}}%
       \bbl@tempb}%
      \@empty}%
    \@empty}
```

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

```
\ifx\bbl@opt@main\@nnil\else
  \expandafter
  \let\expandafter\bbl@loadmain\csname ds@\bbl@opt@main\endcsname
  \DeclareOption{\bbl@opt@main}{}
\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):

```
\def\AfterBabelLanguage#1{%
  \bbl@ifsamestring\CurrentOption{#1}{\global\bbl@add\bbl@afterlang}{}}
\DeclareOption*{}
\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`.

```
\ifx\bbl@opt@main\@nnil
  \edef\bbl@tempa{\@classoptionslist,\bbl@language@opts}
  \let\bbl@tempc\@empty
  \bbl@for\bbl@tempb\bbl@tempa{%
    \@expandtwoargs\in@{,\bbl@tempb,}{,\bbl@loaded,}%
    \ifin@\edef\bbl@tempc{\bbl@tempb}\fi}
  \def\bbl@tempa#1,#2\@nnil{\def\bbl@tempb{#1}}
  \expandafter\bbl@tempa\bbl@loaded,\@nnil
  \ifx\bbl@tempb\bbl@tempc\else
```

```
    \bbl@warning{%
      Last declared language option is `\bbl@tempc,\\%
      but the last processed one was `\bbl@tempb'.\\%
      The main language cannot be set as both a global\\%
      and a package option. Use `main=\bbl@tempc' as\\%
      option. Reported}%
  \fi
\else
  \DeclareOption{\bbl@opt@main}{\bbl@loadmain}
  \ExecuteOptions{\bbl@opt@main}
  \DeclareOption*{}
  \ProcessOptions*
\fi
\def\AfterBabelLanguage{%
  \bbl@error
    {Too late for \string\AfterBabelLanguage}%
    {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.

```
\ifx\bbl@main@language\@undefined
  \bbl@error{%
    You haven't specified a language option}{%
    You need to specify a language, either as a global option\\%
    or as an optional argument to the \string\usepackage\space
    command;\\%
    You shouldn't try to proceed from here, type x to quit.}
\fi
⟨/package⟩
```

7 The Kernel of Babel

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.

7.1 Tools

```
⟨*core⟩
\ifx\bbl@opt@strings\@undefined
  \def\bbl@opt@safe{BR}
  \let\bbl@opt@strings\@nnil
  \let\bbl@opt@shorthands\@nnil
\fi
\ifx\BabelModifiers\@undefined\let\BabelModifiers\relax\fi
\ifx\bbl@afterlang\@undefined\let\bbl@afterlang\relax\fi
\providecommand\AfterBabelLanguage[2]{}
\def\bbl@csarg#1#2{\expandafter#1\csname bbl@#2\endcsname}%
```

`\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`-statement[11]. These macros will break if another `\if...\fi` statement appears in one of the arguments and it is not enclosed in braces.

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

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.

```
\def\bbl@withactive#1#2{%
  \begingroup
    \lccode`~=`#2\relax
    \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@...`

```
\def\bbl@redefine#1{%
  \edef\bbl@tempa{\expandafter\@gobble\string#1}%
  \expandafter\let\csname org@\bbl@tempa\endcsname#1%
  \expandafter\def\csname\bbl@tempa\endcsname}
```

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

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

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

```
\def\bbl@redefine@long#1{%
  \edef\bbl@tempa{\expandafter\@gobble\string#1}%
  \expandafter\let\csname org@\bbl@tempa\endcsname#1%
  \expandafter\long\expandafter\def\csname\bbl@tempa\endcsname}
\@onlypreamble\bbl@redefine@long
```

[11] This code is based on code presented in TUGboat vol. 12, no2, June 1991 in "An expansion Power Lemma" by Sonja Maus.

`\bbl@redefinerobust` 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. Therefor all we need to do now is define `\foo␣`.

```
\def\bbl@redefinerobust#1{%
  \edef\bbl@tempa{\expandafter\@gobble\string#1}%
  \expandafter\ifx\csname\bbl@tempa\space\endcsname\relax
    \expandafter\let\csname org@\bbl@tempa\endcsname#1%
    \expandafter\edef\csname\bbl@tempa\endcsname{\noexpand\protect
      \expandafter\noexpand\csname\bbl@tempa\space\endcsname}%
  \else
    \expandafter\let\csname org@\bbl@tempa\expandafter\endcsname
                     \csname\bbl@tempa\space\endcsname
  \fi
  \expandafter\def\csname\bbl@tempa\space\endcsname}
```

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

```
\@onlypreamble\bbl@redefinerobust
```

7.2 Encoding issues

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.

```
\AtEndOfPackage{\edef\latinencoding{\cf@encoding}}
```

But this might be overruled with a later loading of the package `fontenc`. Therefor 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.

```
\AtBeginDocument{%
  \gdef\latinencoding{OT1}%
  \ifx\cf@encoding\bbl@t@one
    \xdef\latinencoding{\bbl@t@one}%
  \else
    \@ifl@aded{def}{t1enc}{\xdef\latinencoding{\bbl@t@one}}{}%
  \fi
  }
```

`\latintext` Then we can define the command `\latintext` which is a declarative switch to a latin font-encoding.

```
\DeclareRobustCommand{\latintext}{%
  \fontencoding{\latinencoding}\selectfont
  \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.

```
\ifx\@undefined\DeclareTextFontCommand
  \DeclareRobustCommand{\textlatin}[1]{\leavevmode{\latintext #1}}
\else
  \DeclareTextFontCommand{\textlatin}{\latintext}
\fi
```

The second version of this 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 at-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. Therefor 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`.

```
\def\LdfInit#1#2{%
  \chardef\atcatcode=\catcode`\@
  \catcode`\@=11\relax
  \chardef\eqcatcode=\catcode`\=
  \catcode`\==12\relax
  \expandafter\if\expandafter\@backslashchar
                  \expandafter\@car\string#2\@nil
    \ifx#2\@undefined\else
      \ldf@quit{#1}%
    \fi
  \else
    \expandafter\ifx\csname#2\endcsname\relax\else
      \ldf@quit{#1}%
    \fi
  \fi
  \let\bbl@screset\@empty
  \ifx\originalTeX\@undefined
    \let\originalTeX\@empty
  \else
```

```
    \originalTeX
  \fi}
```

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

```
\def\ldf@quit#1{%
  \expandafter\main@language\expandafter{#1}%
  \catcode`\@=\atcatcode \let\atcatcode\relax
  \catcode`\==\eqcatcode \let\eqcatcode\relax
  \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.

```
\def\ldf@finish#1{%
  \loadlocalcfg{#1}%
  \bbl@afterlang
  \let\bbl@afterlang\relax
  \let\BabelModifiers\relax
  \let\bbl@screset\relax
  \expandafter\main@language\expandafter{#1}%
  \catcode`\@=\atcatcode \let\atcatcode\relax
  \catcode`\==\eqcatcode \let\eqcatcode\relax}
```

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

```
\@onlypreamble\LdfInit
\@onlypreamble\ldf@quit
\@onlypreamble\ldf@finish
```

`\main@language` `\bbl@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.

```
\def\main@language#1{%
  \def\bbl@main@language{#1}%
  \let\languagename\bbl@main@language
  \bbl@patterns{\languagename}}
```

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

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

7.3 Support for active characters

`\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).

To keep all changes local, we begin a new group. Then we redefine the macros `\do` and `\@makeother` to add themselves and the given character without expansion.
To add the character to the macros, we expand the original macros with the additional character inside the redefinition of the macros. Because `\@sanitize` can be undefined, we put the definition inside a conditional.

```
\def\bbl@add@special#1{%
  \begingroup
    \def\do{\noexpand\do\noexpand}%
    \def\@makeother{\noexpand\@makeother\noexpand}%
  \edef\x{\endgroup
    \def\noexpand\dospecials{\dospecials\do#1}%
    \expandafter\ifx\csname @sanitize\endcsname\relax \else
      \def\noexpand\@sanitize{\@sanitize\@makeother#1}%
    \fi}%
  \x}
```

The macro `\x` contains at this moment the following:
`\endgroup\def\dospecials{`*old contents* `\do`⟨*char*⟩`}`.
If `\@sanitize` is defined, it contains an additional definition of this macro. The last thing we have to do, is the expansion of `\x`. Then `\endgroup` is executed, which restores the old meaning of `\x`, `\do` and `\@makeother`. After the group is closed, the new definition of `\dospecials` (and `\@sanitize`) is assigned.

`\bbl@remove@special` The companion of the former macro is `\bbl@remove@special`. It is used to remove a character from the set macros `\dospecials` and `\@sanitize`.
To keep all changes local, we begin a new group. Then we define a help macro `\x`, which expands to empty if the characters match, otherwise it expands to its nonexpandable input. Because TEX inserts a `\relax`, if the corresponding `\else` or `\fi` is scanned before the comparison is evaluated, we provide a 'stop sign' which should expand to nothing.
With the help of this macro we define `\do` and `\make@other`.
The rest of the work is similar to `\bbl@add@special`.

```
\def\bbl@remove@special#1{%
  \begingroup
    \def\x##1##2{\ifnum`#1=`##2\noexpand\@empty
                 \else\noexpand##1\noexpand##2\fi}%
    \def\do{\x\do}%
    \def\@makeother{\x\@makeother}%
  \edef\x{\endgroup
    \def\noexpand\dospecials{\dospecials}%
    \expandafter\ifx\csname @sanitize\endcsname\relax \else
      \def\noexpand\@sanitize{\@sanitize}%
    \fi}%
  \x}
```

7.4 Shorthands

`\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`).

```
\def\bbl@active@def#1#2#3#4{%
  \@namedef{#3#1}{%
    \expandafter\ifx\csname#2@sh@#1@\endcsname\relax
      \bbl@afterelse\bbl@sh@select#2#1{#3@arg#1}{#4#1}%
    \else
      \bbl@afterfi\csname#2@sh@#1@\endcsname
    \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.

```
  \long\@namedef{#3@arg#1}##1{%
    \expandafter\ifx\csname#2@sh@#1@\string##1@\endcsname\relax
      \bbl@afterelse\csname#4#1\endcsname##1%
    \else
      \bbl@afterfi\csname#2@sh@#1@\string##1@\endcsname
    \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.

```
\def\initiate@active@char#1{%
  \expandafter\ifx\csname active@char\string#1\endcsname\relax
    \bbl@withactive
      {\expandafter\@initiate@active@char\expandafter}#1\string#1#1%
  \fi}
```

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 treatement to avoid making them `\relax`).

```
\def\@initiate@active@char#1#2#3{%
  \expandafter\edef\csname bbl@oricat@#2\endcsname{%
```

```
    \catcode`#2=\the\catcode`#2\relax}%
  \ifx#1\@undefined
    \expandafter\edef\csname bbl@oridef@#2\endcsname{%
      \let\noexpand#1\noexpand\@undefined}%
  \else
    \expandafter\let\csname bbl@oridef@@#2\endcsname#1%
    \expandafter\edef\csname bbl@oridef@#2\endcsname{%
      \let\noexpand#1%
      \expandafter\noexpand\csname bbl@oridef@@#2\endcsname}%
  \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*).

```
  \ifx#1#3\relax
    \expandafter\let\csname normal@char#2\endcsname#3%
  \else
    \bbl@info{Making #2 an active character}%
    \ifnum\mathcode`#2="8000
      \@namedef{normal@char#2}{%
        \textormath{#3}{\csname bbl@oridef@@#2\endcsname}}%
    \else
      \@namedef{normal@char#2}{#3}%
    \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).

```
    \bbl@restoreactive{#2}%
    \AtBeginDocument{%
      \catcode`#2\active
      \if@filesw
        \immediate\write\@mainaux{\catcode`\string#2\active}%
      \fi}%
    \expandafter\bbl@add@special\csname#2\endcsname
    \catcode`#2\active
  \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*⟩).

```
\let\bbl@tempa\@firstoftwo
\if\string^#2%
  \def\bbl@tempa{\noexpand\textormath}%
\else
  \ifx\bbl@mathnormal\@undefined\else
    \let\bbl@tempa\bbl@mathnormal
  \fi
\fi
\expandafter\edef\csname active@char#2\endcsname{%
  \bbl@tempa
    {\noexpand\if@safe@actives
       \noexpand\expandafter
       \expandafter\noexpand\csname normal@char#2\endcsname
     \noexpand\else
       \noexpand\expandafter
       \expandafter\noexpand\csname user@active#2\endcsname
     \noexpand\fi}%
   {\expandafter\noexpand\csname normal@char#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!).

```
\bbl@csarg\edef{active@#2}{%
  \noexpand\active@prefix\noexpand#1%
  \expandafter\noexpand\csname active@char#2\endcsname}%
\bbl@csarg\edef{normal@#2}{%
  \noexpand\active@prefix\noexpand#1%
  \expandafter\noexpand\csname normal@char#2\endcsname}%
\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.

```
\bbl@active@def#2\user@group{user@active}{language@active}%
\bbl@active@def#2\language@group{language@active}{system@active}%
\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.

```
\expandafter\edef\csname\user@group @sh@#2@@\endcsname
  {\expandafter\noexpand\csname normal@char#2\endcsname}%
```

```
\expandafter\edef\csname\user@group @sh@#2@\string\protect@\endcsname
  {\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. Therefor an extra level of expansion is introduced with a check for math mode on the upper level.

```
\if\string'#2%
  \let\prim@s\bbl@prim@s
  \let\active@math@prime#1%
\fi}
```

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`.

```
\@ifpackagewith{babel}{KeepShorthandsActive}%
  {\let\bbl@restoreactive\@gobble}%
  {\def\bbl@restoreactive#1{%
     \edef\bbl@tempa{%
       \noexpand\AfterBabelLanguage\noexpand\CurrentOption
         {\catcode`#1=\the\catcode`#1\relax}%
       \noexpand\AtEndOfPackage{\catcode`#1=\the\catcode`#1\relax}}%
     \bbl@tempa}%
   \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.

```
\def\bbl@sh@select#1#2{%
  \expandafter\ifx\csname#1@sh@#2@sel\endcsname\relax
    \bbl@afterelse\bbl@scndcs
  \else
    \bbl@afterfi\csname#1@sh@#2@sel\endcsname
  \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 `\protect`s the active character whenever `\protect` is *not* `\@typeset@protect`.

```
\def\active@prefix#1{%
  \ifx\protect\@typeset@protect
  \else
```

When `\protect` is set to `\@unexpandable@protect` we make sure that the active character is als *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).

```
    \ifx\protect\@unexpandable@protect
      \noexpand#1%
    \else
      \protect#1%
    \fi
    \expandafter\@gobble
  \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*⟩.

```
\newif\if@safe@actives
\@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.

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

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

```
\def\bbl@activate#1{%
  \bbl@withactive{\expandafter\let\expandafter}#1%
    \csname bbl@active@\string#1\endcsname}
\def\bbl@deactivate#1{%
  \bbl@withactive{\expandafter\let\expandafter}#1%
    \csname bbl@normal@\string#1\endcsname}
```

`\bbl@firstcs` `\bbl@scndcs` These macros have two arguments. They use one of their arguments to build a control sequence from.

```
\def\bbl@firstcs#1#2{\csname#1\endcsname}
\def\bbl@scndcs#1#2{\csname#2\endcsname}
```

`\declare@shorthand` The command `\declare@shorthand` is used to declare a shorthand on a certain level. It takes three arguments:

1. a name for the collection of shorthands, i.e. 'system', or 'dutch';
2. the character (sequence) that makes up the shorthand, i.e. `~` or `"a`;
3. the code to be executed when the shorthand is encountered.

```
\def\declare@shorthand#1#2{\@decl@short{#1}#2\@nil}
\def\@decl@short#1#2#3\@nil#4{%
  \def\bbl@tempa{#3}%
```

```
\ifx\bbl@tempa\@empty
  \expandafter\let\csname #1@sh@\string#2@sel\endcsname\bbl@scndcs
  \@ifundefined{#1@sh@\string#2@}{}%
    {\def\bbl@tempa{#4}%
     \expandafter\ifx\csname#1@sh@\string#2@\endcsname\bbl@tempa
     \else
       \bbl@info
         {Redefining #1 shorthand \string#2\\%
          in language \CurrentOption}%
     \fi}%
  \@namedef{#1@sh@\string#2@}{#4}%
\else
  \expandafter\let\csname #1@sh@\string#2@sel\endcsname\bbl@firstcs
  \@ifundefined{#1@sh@\string#2@\string#3@}{}%
    {\def\bbl@tempa{#4}%
     \expandafter\ifx\csname#1@sh@\string#2@\string#3@\endcsname\bbl@tempa
     \else
       \bbl@info
         {Redefining #1 shorthand \string#2\string#3\\%
          in language \CurrentOption}%
     \fi}%
  \@namedef{#1@sh@\string#2@\string#3@}{#4}%
\fi}
```

`\textormath` 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 `\textormath` is provided.

```
\def\textormath{%
  \ifmmode
    \expandafter\@secondoftwo
  \else
    \expandafter\@firstoftwo
  \fi}
```

`\user@group` `\language@group` `\system@group` 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'.

```
\def\user@group{user}
\def\language@group{english}
\def\system@group{system}
```

`\useshorthands` 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.

```
\def\useshorthands{%
```

```
  \@ifstar\bbl@usesh@s{\bbl@usesh@x{}}}
\def\bbl@usesh@s#1{%
  \bbl@usesh@x
    {\AddBabelHook{babel-sh-\string#1}{afterextras}{\bbl@activate{#1}}}%
    {#1}}
\def\bbl@usesh@x#1#2{%
  \bbl@ifshorthand{#2}%
    {\def\user@group{user}%
     \initiate@active@char{#2}%
     #1%
     \bbl@activate{#2}}%
    {\bbl@error
       {Cannot declare a shorthand turned off (\string#2)}
       {Sorry, but you cannot use shorthands which have been\\%
        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 `\bbl@set@user@generic`); we make also sure `{}` and `\protect` are taken into account in this new top level.

```
\def\user@language@group{user@\language@group}
\def\bbl@set@user@generic#1#2{%
  \@ifundefined{user@generic@active#1}%
    {\bbl@active@def#1\user@language@group{user@active}{user@generic@active}%
     \bbl@active@def#1\user@group{user@generic@active}{language@active}%
     \expandafter\edef\csname#2@sh@#1@@\endcsname{%
       \expandafter\noexpand\csname normal@char#1\endcsname}%
     \expandafter\edef\csname#2@sh@#1@\string\protect@\endcsname{%
       \expandafter\noexpand\csname user@active#1\endcsname}}%
  \@empty}
\newcommand\defineshorthand[3][user]{%
  \edef\bbl@tempa{\zap@space#1 \@empty}%
  \bbl@for\bbl@tempb\bbl@tempa{%
    \if*\expandafter\@car\bbl@tempb\@nil
      \edef\bbl@tempb{user@\expandafter\@gobble\bbl@tempb}%
      \@expandtwoargs
        \bbl@set@user@generic{\expandafter\string\@car#2\@nil}\bbl@tempb
    \fi
    \declare@shorthand{\bbl@tempb}{#2}{#3}}}
```

`\languageshorthands` 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.

```
\def\languageshorthands#1{\def\language@group{#1}}
```

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

```
\def\aliasshorthand#1#2{%
```

```
\bbl@ifshorthand{#2}%
  {\expandafter\ifx\csname active@char\string#2\endcsname\relax
     \ifx\document\@notprerr
       \@notshorthand{#2}%
     \else
       \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 lattest to `\active@char"`.

```
       \expandafter\let\csname active@char\string#2\expandafter\endcsname
         \csname active@char\string#1\endcsname
       \expandafter\let\csname normal@char\string#2\expandafter\endcsname
         \csname normal@char\string#1\endcsname
       \bbl@activate{#2}%
     \fi
   \fi}%
  {\bbl@error
     {Cannot declare a shorthand turned off (\string#2)}
     {Sorry, but you cannot use shorthands which have been\\%
      turned off in the package options}}}
```

`\@notshorthand`

```
\def\@notshorthand#1{%
  \bbl@error{%
    The character `\string #1' should be made a shorthand character;\\%
    add the command \string\useshorthands\string{#1\string} to
    the preamble.\\%
    I will ignore your instruction}{}}
```

`\shorthandon` `\shorthandoff` 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.

```
\newcommand*\shorthandon[1]{\bbl@switch@sh\@ne#1\@nnil}
\DeclareRobustCommand*\shorthandoff{%
  \@ifstar{\bbl@shorthandoff\tw@}{\bbl@shorthandoff\z@}}
\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.

```
\def\bbl@switch@sh#1#2{%
  \ifx#2\@nnil\else
    \@ifundefined{bbl@active@\string#2}%
```

```
      {\bbl@error
         {I cannot switch `\string#2' on or off--not a shorthand}%
         {This character is not a shorthand. Maybe you made\\%
          a typing mistake? I will ignore your instruction}}%
      {\ifcase#1%
         \catcode`#212\relax
       \or
         \catcode`#2\active
       \or
         \csname bbl@oricat@\string#2\endcsname
         \csname bbl@oridef@\string#2\endcsname
       \fi}%
    \bbl@afterfi\bbl@switch@sh#1%
  \fi}
```

7.5 Conditional loading of shorthands

Note the value is that at the expansion time, eg, in the preample shorhands are usually deactivated.

```
\def\babelshorthand{\active@prefix\babelshorthand\bbl@putsh}
\def\bbl@putsh#1{%
   \@ifundefined{bbl@active@\string#1}%
      {\bbl@putsh@i#1\@empty\@nnil}%
      {\csname bbl@active@\string#1\endcsname}}
\def\bbl@putsh@i#1#2\@nnil{%
  \csname\languagename @sh@\string#1@%
    \ifx\@empty#2\else\string#2@\fi\endcsname}
\ifx\bbl@opt@shorthands\@nnil\else
  \let\bbl@s@initiate@active@char\initiate@active@char
  \def\initiate@active@char#1{%
    \bbl@ifshorthand{#1}{\bbl@s@initiate@active@char{#1}}{}}
  \let\bbl@s@switch@sh\bbl@switch@sh
  \def\bbl@switch@sh#1#2{%
    \ifx#2\@nnil\else
      \bbl@afterfi
      \bbl@ifshorthand{#2}{\bbl@s@switch@sh#1{#2}}{\bbl@switch@sh#1}%
    \fi}
  \let\bbl@s@activate\bbl@activate
  \def\bbl@activate#1{%
    \bbl@ifshorthand{#1}{\bbl@s@activate{#1}}{}}
  \let\bbl@s@deactivate\bbl@deactivate
  \def\bbl@deactivate#1{%
    \bbl@ifshorthand{#1}{\bbl@s@deactivate{#1}}{}}
\fi
```

`\bbl@prim@s` `\bbl@pr@m@s` One of the internal macros that are involved in substituting `\prime` for each right quote in 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.

```
\def\bbl@prim@s{%
  \prime\futurelet\@let@token\bbl@pr@m@s}
\def\bbl@if@primes#1#2{%
  \ifx#1\@let@token
    \expandafter\@firstoftwo
  \else\ifx#2\@let@token
    \bbl@afterelse\expandafter\@firstoftwo
  \else
    \bbl@afterfi\expandafter\@secondoftwo
  \fi\fi}
\begingroup
  \catcode`\^=7  \catcode`\*=\active  \lccode`\*=`\^
  \catcode`\'=12 \catcode`\"=\active  \lccode`\"=`\'
  \lowercase{%
    \gdef\bbl@pr@m@s{%
      \bbl@if@primes"'%
        \pr@@@s
        {\bbl@if@primes*^\pr@@@t\egroup}}}
\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).

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

`\OT1dqpos` `\T1dqpos` 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. Therefor we define two macros here to store the position of the character in these encodings.

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

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

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

7.6 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.

```
\newcommand\languageattribute[2]{%
  \def\bbl@tempc{#1}%
  \bbl@fixname\bbl@tempc
  \bbl@iflanguage\bbl@tempc{%
    \@for\bbl@attr:=#2\do{%
```

We want to make sure that each attribute is selected only once; therefor 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.

```
      \ifx\bbl@known@attribs\@undefined
        \in@false
      \else
```

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

```
        \@expandtwoargs\in@{,\bbl@tempc-\bbl@attr,}{,\bbl@known@attribs,}%
      \fi
```

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

```
      \ifin@
        \bbl@warning{%
          You have more than once selected the attribute '\bbl@attr'\\%
          for language #1}%
      \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 TEX-code.

```
        \edef\bbl@tempa{%
          \noexpand\bbl@add@list
          \noexpand\bbl@known@attribs{\bbl@tempc-\bbl@attr}}%
        \bbl@tempa
        \edef\bbl@tempa{\bbl@tempc-\bbl@attr}%
        \expandafter\bbl@ifknown@ttrib\expandafter{\bbl@tempa}\bbl@attributes%
        {\csname\bbl@tempc @attr@\bbl@attr\endcsname}%
        {\@attrerr{\bbl@tempc}{\bbl@attr}}%
    \fi}}
```

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

```
\@onlypreamble\languageattribute
```

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

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

`\bbl@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}`.

```
\def\bbl@declare@ttribute#1#2#3{%
  \@expandtwoargs\in@{,#2,}{,\BabelModifiers,}%
  \ifin@
    \AfterBabelLanguage{#1}{\languageattribute{#1}{#2}}%
  \fi
  \bbl@add@list\bbl@attributes{#1-#2}%
  \expandafter\def\csname#1@attr@#2\endcsname{#3}}
```

`\bbl@ifattributeset` This internal macro has 4 arguments. It can be used to interpret TEX 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.

```
\def\bbl@ifattributeset#1#2#3#4{%
```

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

```
  \ifx\bbl@known@attribs\@undefined
    \in@false
  \else
```

The we need to check the list of known attributes.

```
    \@expandtwoargs\in@{,#1-#2,}{,\bbl@known@attribs,}%
  \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`'.

```
  \ifin@
    \bbl@afterelse#3%
  \else
    \bbl@afterfi#4%
  \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

```
\def\bbl@add@list#1#2{%
  \ifx#1\@undefined
    \def#1{#2}%
  \else
    \ifx#1\@empty
      \def#1{#2}%
    \else
      \edef#1{#1,#2}%
    \fi
```

```
  \fi
  }
```

`\bbl@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 TEX-code to be executed when the attribute is known and the TEX-code to be executed otherwise.

```
\def\bbl@ifknown@ttrib#1#2{%
```

We first assume the attribute is unknown.

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

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

```
  \@for\bbl@tempb:=#2\do{%
    \expandafter\in@\expandafter{\expandafter,\bbl@tempb,}{,#1,}%
    \ifin@
```

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

```
      \let\bbl@tempa\@firstoftwo
    \else
    \fi}%
```

Finally we execute `\bbl@tempa`.

```
  \bbl@tempa
}
```

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

```
\def\bbl@clear@ttribs{%
  \ifx\bbl@attributes\@undefined\else
    \@for\bbl@tempa:=\bbl@attributes\do{%
      \expandafter\bbl@clear@ttrib\bbl@tempa.
      }%
    \let\bbl@attributes\@undefined
  \fi
  }
\def\bbl@clear@ttrib#1-#2.{%
  \expandafter\let\csname#1@attr@#2\endcsname\@undefined}
\AtBeginDocument{\bbl@clear@ttribs}
```

7.7 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
```
\def\babel@beginsave{\babel@savecnt\z@}
```
Before it's forgotten, allocate the counter and initialize all.
```
\newcount\babel@savecnt
\babel@beginsave
```

\babel@save The macro \babel@save⟨*csname*⟩ saves the current meaning of the control sequence ⟨*csname*⟩ to \originalTeX[12]. 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.
```
\def\babel@save#1{%
  \expandafter\let\csname babel@\number\babel@savecnt\endcsname#1\relax
  \begingroup
    \toks@\expandafter{\originalTeX\let#1=}%
    \edef\x{\endgroup
      \def\noexpand\originalTeX{\the\toks@ \expandafter\noexpand
          \csname babel@\number\babel@savecnt\endcsname\relax}}%
  \x
  \advance\babel@savecnt\@ne}
```

\babel@savevariable The macro \babel@savevariable⟨*variable*⟩ saves the value of the variable. ⟨*variable*⟩ can be anything allowed after the \the primitive.
```
\def\babel@savevariable#1{\begingroup
    \toks@\expandafter{\originalTeX #1=}%
    \edef\x{\endgroup
      \def\noexpand\originalTeX{\the\toks@ \the#1\relax}}%
  \x}
```

\bbl@frenchspacing Some languages need to have \frenchspacing in effect. Others don't want that. The
\bbl@nonfrenchspacing command \bbl@frenchspacing switches it on when it isn't already in effect and \bbl@nonfrenchspacing switches it off if necessary.
```
\def\bbl@frenchspacing{%
  \ifnum\the\sfcode`\.=\@m
    \let\bbl@nonfrenchspacing\relax
  \else
    \frenchspacing
    \let\bbl@nonfrenchspacing\nonfrenchspacing
  \fi}
\let\bbl@nonfrenchspacing\nonfrenchspacing
```

7.8 Support for extending 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 TEX-code to be added to the ⟨*control sequence*⟩.

[12] \originalTeX has to be expandable, i. e. you shouldn't let it to \relax.

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 TEX-code to be added. Finally the ⟨*control sequence*⟩ is *re*defined, using the contents of the token register.

```
\def\addto#1#2{%
  \ifx#1\@undefined
    \def#1{#2}%
  \else
    \ifx#1\relax
      \def#1{#2}%
    \else
      {\toks@\expandafter{#1#2}%
       \xdef#1{\the\toks@}}%
    \fi
  \fi}
```

7.9 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.

```
\@onlypreamble\babelhyphenation
\AtEndOfPackage{%
  \newcommand\babelhyphenation[2][\@empty]{%
    \ifx\bbl@hyphenation@\relax
      \let\bbl@hyphenation@\@empty
    \fi
    \ifx\bbl@hyphlist\@empty\else
      \bbl@warning{%
        You must not intermingle \string\selectlanguage\space and\\%
        \string\babelhyphenation\space or some exception will not\\%
        be taken into account. Reported}%
    \fi
    \ifx\@empty#1%
      \protected@edef\bbl@hyphenation@{\bbl@hyphenation@\space#2}%
    \else
      \edef\bbl@tempb{\zap@space#1 \@empty}%
      \bbl@for\bbl@tempa\bbl@tempb{%
        \bbl@fixname\bbl@tempa
        \bbl@iflanguage\bbl@tempa{%
          \bbl@csarg\protected@edef{hyphenation@\bbl@tempa}{%
            \@ifundefined{bbl@hyphenation@\bbl@tempa}%
              \@empty
              {\csname bbl@hyphenation@\bbl@tempa\endcsname\space}%
            #2}}}%
    \fi}}
```

`\bbl@allowhyphens` This macro makes hyphenation possible. Basically its definition is nothing more than `\nobreak \hskip 0pt plus 0pt`[13].

```
\def\bbl@allowhyphens{\nobreak\hskip\z@skip}
\def\bbl@t@one{T1}
\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.

```
\newcommand\babelnullhyphen{\char\hyphenchar\font}
\def\babelhyphen{\active@prefix\babelhyphen\bbl@hyphen}
\def\bbl@hyphen{%
  \@ifstar{\bbl@hyphen@i @}{\bbl@hyphen@i\@empty}}
\def\bbl@hyphen@i#1#2{%
  \@ifundefined{bbl@hy@#1#2\@empty}%
    {\csname bbl@#1usehyphen\endcsname{\discretionary{#2}{}{#2}}}%
    {\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 discretionaty 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.

```
\def\bbl@usehyphen#1{%
  \leavevmode
  \ifdim\lastskip>\z@\mbox{#1}\nobreak\else\nobreak#1\fi
  \hskip\z@skip}
\def\bbl@@usehyphen#1{%
  \leavevmode\ifdim\lastskip>\z@\mbox{#1}\else#1\fi}
```

The following macro inserts the hyphen char.

```
\def\bbl@hyphenchar{%
  \ifnum\hyphenchar\font=\m@ne
    \babelnullhyphen
  \else
    \char\hyphenchar\font
  \fi}
```

Finally, we define the hyphen "types". Their names will not change, so you may use them in `ldf`'s.

```
\def\bbl@hy@soft{\bbl@usehyphen{\discretionary{\bbl@hyphenchar}{}{}}}
\def\bbl@hy@@soft{\bbl@@usehyphen{\discretionary{\bbl@hyphenchar}{}{}}}
\def\bbl@hy@hard{\bbl@usehyphen\bbl@hyphenchar}
\def\bbl@hy@@hard{\bbl@@usehyphen\bbl@hyphenchar}
```

[13] TEX begins and ends a word for hyphenation at a glue node. The penalty prevents a linebreak at this glue node.

```
\def\bbl@hy@nobreak{\bbl@usehyphen{\mbox{\bbl@hyphenchar}\nobreak}}
\def\bbl@hy@@nobreak{\mbox{\bbl@hyphenchar}}
\def\bbl@hy@repeat{%
  \bbl@usehyphen{%
    \discretionary{\bbl@hyphenchar}{\bbl@hyphenchar}{\bbl@hyphenchar}%
    \nobreak}}
\def\bbl@hy@@repeat{%
  \bbl@@usehyphen{%
    \discretionary{\bbl@hyphenchar}{\bbl@hyphenchar}{\bbl@hyphenchar}}}
\def\bbl@hy@empty{\hskip\z@skip}
\def\bbl@hy@@empty{\discretionary{}{}{}}
```

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

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

7.10 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.

```
\def\set@low@box#1{\setbox\tw@\hbox{,}\setbox\z@\hbox{#1}%
    \dimen\z@\ht\z@ \advance\dimen\z@ -\ht\tw@%
    \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.

```
\def\save@sf@q#1{\leavevmode
  \begingroup
    \edef\@SF{\spacefactor\the\spacefactor}#1\@SF
  \endgroup}
```

7.11 Making glyphs available

The file `babel.dtx`[14] 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`.

7.12 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, therefor we make it available by lowering the normal open quote character to the baseline.

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

[14] The file described in this section has version number v3.9f, and was last revised on 2013/05/16.

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

```
\ProvideTextCommandDefault{\quotedblbase}{%
  \UseTextSymbol{OT1}{\quotedblbase}}
```

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

```
\ProvideTextCommand{\quotesinglbase}{OT1}{%
  \save@sf@q{\set@low@box{\textquoteright\/}%
    \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.

```
\ProvideTextCommandDefault{\quotesinglbase}{%
  \UseTextSymbol{OT1}{\quotesinglbase}}
```

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

```
\ProvideTextCommand{\guillemotleft}{OT1}{%
  \ifmmode
    \ll
  \else
    \save@sf@q{\nobreak
      \raise.2ex\hbox{$\scriptscriptstyle\ll$}\bbl@allowhyphens}%
  \fi}
\ProvideTextCommand{\guillemotright}{OT1}{%
  \ifmmode
    \gg
  \else
    \save@sf@q{\nobreak
      \raise.2ex\hbox{$\scriptscriptstyle\gg$}\bbl@allowhyphens}%
  \fi}
```

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

```
\ProvideTextCommandDefault{\guillemotleft}{%
  \UseTextSymbol{OT1}{\guillemotleft}}
\ProvideTextCommandDefault{\guillemotright}{%
  \UseTextSymbol{OT1}{\guillemotright}}
```

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

```
\ProvideTextCommand{\guilsinglleft}{OT1}{%
  \ifmmode
    <%
  \else
    \save@sf@q{\nobreak
      \raise.2ex\hbox{$\scriptscriptstyle<$}\bbl@allowhyphens}%
  \fi}
\ProvideTextCommand{\guilsinglright}{OT1}{%
  \ifmmode
    >%
  \else
```

```
    \save@sf@q{\nobreak
      \raise.2ex\hbox{$\scriptscriptstyle>$}\bbl@allowhyphens}%
  \fi}
```

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

```
\ProvideTextCommandDefault{\guilsinglleft}{%
  \UseTextSymbol{OT1}{\guilsinglleft}}
\ProvideTextCommandDefault{\guilsinglright}{%
  \UseTextSymbol{OT1}{\guilsinglright}}
```

7.13 Letters

\ij \IJ The dutch language uses the letter 'ij'. It is available in T1 encoded fonts, but not in the OT1 encoded fonts. Therefor we fake it for the OT1 encoding.

```
\DeclareTextCommand{\ij}{OT1}{%
  i\kern-0.02em\bbl@allowhyphens j}
\DeclareTextCommand{\IJ}{OT1}{%
  I\kern-0.02em\bbl@allowhyphens J}
\DeclareTextCommand{\ij}{T1}{\char188}
\DeclareTextCommand{\IJ}{T1}{\char156}
```

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

```
\ProvideTextCommandDefault{\ij}{%
  \UseTextSymbol{OT1}{\ij}}
\ProvideTextCommandDefault{\IJ}{%
  \UseTextSymbol{OT1}{\IJ}}
```

\dj \DJ The croatian language needs the letters \dj and \DJ; they are available in the T1 encoding, 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).

```
\def\crrtic@{\hrule height0.1ex width0.3em}
\def\crttic@{\hrule height0.1ex width0.33em}
\def\ddj@{%
  \setbox0\hbox{d}\dimen@=\ht0
  \advance\dimen@1ex
  \dimen@.45\dimen@
  \dimen@ii\expandafter\rem@pt\the\fontdimen\@ne\font\dimen@
  \advance\dimen@ii.5ex
  \leavevmode\rlap{\raise\dimen@\hbox{\kern\dimen@ii\vbox{\crrtic@}}}}
\def\DDJ@{%
  \setbox0\hbox{D}\dimen@=.55\ht0
  \dimen@ii\expandafter\rem@pt\the\fontdimen\@ne\font\dimen@
  \advance\dimen@ii.15ex %           correction for the dash position
  \advance\dimen@ii-.15\fontdimen7\font %     correction for cmtt font
  \dimen\thr@@\expandafter\rem@pt\the\fontdimen7\font\dimen@
  \leavevmode\rlap{\raise\dimen@\hbox{\kern\dimen@ii\vbox{\crttic@}}}}
```

```
%
\DeclareTextCommand{\dj}{OT1}{\ddj@ d}
\DeclareTextCommand{\DJ}{OT1}{\DDJ@ D}
```

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

```
\ProvideTextCommandDefault{\dj}{%
  \UseTextSymbol{OT1}{\dj}}
\ProvideTextCommandDefault{\DJ}{%
  \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. Therefor we make it available here.

```
\DeclareTextCommand{\SS}{OT1}{SS}
\ProvideTextCommandDefault{\SS}{\UseTextSymbol{OT1}{\SS}}
```

7.14 Shorthands for quotation marks

Shorthands are provided for a number of different quotation marks, which make them usable both outside and inside mathmode.

`\glq` The 'german' single quotes.

`\grq`

```
\ProvideTextCommand{\glq}{OT1}{%
  \textormath{\quotesinglbase}{\mbox{\quotesinglbase}}}
\ProvideTextCommand{\glq}{T1}{%
  \textormath{\quotesinglbase}{\mbox{\quotesinglbase}}}
\ProvideTextCommandDefault{\glq}{\UseTextSymbol{OT1}\glq}
```

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

```
\ProvideTextCommand{\grq}{T1}{%
  \textormath{\textquoteleft}{\mbox{\textquoteleft}}}
\ProvideTextCommand{\grq}{OT1}{%
  \save@sf@q{\kern-.0125em%
  \textormath{\textquoteleft}{\mbox{\textquoteleft}}%
  \kern.07em\relax}}
\ProvideTextCommandDefault{\grq}{\UseTextSymbol{OT1}\grq}
```

`\glqq` The 'german' double quotes.

`\grqq`

```
\ProvideTextCommand{\glqq}{OT1}{%
  \textormath{\quotedblbase}{\mbox{\quotedblbase}}}
\ProvideTextCommand{\glqq}{T1}{%
  \textormath{\quotedblbase}{\mbox{\quotedblbase}}}
\ProvideTextCommandDefault{\glqq}{\UseTextSymbol{OT1}\glqq}
```

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

```
\ProvideTextCommand{\grqq}{T1}{%
  \textormath{\textquotedblleft}{\mbox{\textquotedblleft}}}
\ProvideTextCommand{\grqq}{OT1}{%
```

```
  \save@sf@q{\kern-.07em%
  \textormath{\textquotedblleft}{\mbox{\textquotedblleft}}%
  \kern.07em\relax}}
\ProvideTextCommandDefault{\grqq}{\UseTextSymbol{OT1}\grqq}
```

`\flq` `\frq` The 'french' single guillemets.

```
\ProvideTextCommand{\flq}{OT1}{%
  \textormath{\guilsinglleft}{\mbox{\guilsinglleft}}}
\ProvideTextCommand{\flq}{T1}{%
  \textormath{\guilsinglleft}{\mbox{\guilsinglleft}}}
\ProvideTextCommandDefault{\flq}{\UseTextSymbol{OT1}\flq}
\ProvideTextCommand{\frq}{OT1}{%
  \textormath{\guilsinglright}{\mbox{\guilsinglright}}}
\ProvideTextCommand{\frq}{T1}{%
  \textormath{\guilsinglright}{\mbox{\guilsinglright}}}
\ProvideTextCommandDefault{\frq}{\UseTextSymbol{OT1}\frq}
```

`\flqq` `\frqq` The 'french' double guillemets.

```
\ProvideTextCommand{\flqq}{OT1}{%
  \textormath{\guillemotleft}{\mbox{\guillemotleft}}}
\ProvideTextCommand{\flqq}{T1}{%
  \textormath{\guillemotleft}{\mbox{\guillemotleft}}}
\ProvideTextCommandDefault{\flqq}{\UseTextSymbol{OT1}\flqq}
\ProvideTextCommand{\frqq}{OT1}{%
  \textormath{\guillemotright}{\mbox{\guillemotright}}}
\ProvideTextCommand{\frqq}{T1}{%
  \textormath{\guillemotright}{\mbox{\guillemotright}}}
\ProvideTextCommandDefault{\frqq}{\UseTextSymbol{OT1}\frqq}
```

7.15 Umlauts and trema's

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` `\umlautlow` To be able to provide both positions of `\"` we provide two commands to switch the positioning, the default will be `\umlauthigh` (the normal positioning).

```
\def\umlauthigh{%
  \def\bbl@umlauta##1{\leavevmode\bgroup%
      \expandafter\accent\csname\f@encoding dqpos\endcsname
      ##1\bbl@allowhyphens\egroup}%
  \let\bbl@umlaute\bbl@umlauta}
\def\umlautlow{%
  \def\bbl@umlauta{\protect\lower@umlaut}}
\def\umlautelow{%
  \def\bbl@umlaute{\protect\lower@umlaut}}
\umlauthigh
```

`\lower@umlaut` The command `\lower@umlaut` is used to position the `\"` closer the the letter. We want the umlaut character lowered, nearer to the letter. To do this we need an extra ⟨*dimen*⟩ register.

```
\expandafter\ifx\csname U@D\endcsname\relax
  \csname newdimen\endcsname\U@D
\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.

```
\def\lower@umlaut#1{%
  \leavevmode\bgroup
    \U@D 1ex%
    {\setbox\z@\hbox{%
      \expandafter\char\csname\f@encoding dqpos\endcsname}%
      \dimen@ -.45ex\advance\dimen@\ht\z@
      \ifdim 1ex<\dimen@ \fontdimen5\font\dimen@ \fi}%
    \expandafter\accent\csname\f@encoding dqpos\endcsname
    \fontdimen5\font\U@D #1%
  \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. Therefor 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).

```
\AtBeginDocument{%
  \DeclareTextCompositeCommand{\"}{OT1}{a}{\bbl@umlauta{a}}%
  \DeclareTextCompositeCommand{\"}{OT1}{e}{\bbl@umlaute{e}}%
  \DeclareTextCompositeCommand{\"}{OT1}{i}{\bbl@umlaute{\i}}%
  \DeclareTextCompositeCommand{\"}{OT1}{\i}{\bbl@umlaute{\i}}%
  \DeclareTextCompositeCommand{\"}{OT1}{o}{\bbl@umlauta{o}}%
  \DeclareTextCompositeCommand{\"}{OT1}{u}{\bbl@umlauta{u}}%
  \DeclareTextCompositeCommand{\"}{OT1}{A}{\bbl@umlauta{A}}%
  \DeclareTextCompositeCommand{\"}{OT1}{E}{\bbl@umlaute{E}}%
  \DeclareTextCompositeCommand{\"}{OT1}{I}{\bbl@umlaute{I}}%
  \DeclareTextCompositeCommand{\"}{OT1}{O}{\bbl@umlauta{O}}%
  \DeclareTextCompositeCommand{\"}{OT1}{U}{\bbl@umlauta{U}}%
}
```

7.16 Multiencoding strings

The aim following commands is to provide a commom interface for strings in several encodings. They also contains several hooks which can be ued by luatex and xetex. 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.

```
\def\bbl@recatcode#1{%
  \@tempcnta="7F
  \def\bbl@tempa{%
    \ifnum\@tempcnta>"FF\else
      \catcode\@tempcnta=#1\relax
      \advance\@tempcnta\@ne
      \expandafter\bbl@tempa
    \fi}%
  \bbl@tempa}
\@onlypreamble\StartBabelCommands
\def\StartBabelCommands{%
  \begingroup
  \bbl@recatcode{11}%
  \def\bbl@scuse{%
    \ifx\bbl@opt@strings\@nnil\def\bbl@opt@strings{generic}\fi}%
  \def\UseStrings{\bbl@scuse\aftergroup\bbl@scuse}%
  \def\SetStringLoop{\afterassignment\bbl@sclp\def\bbl@templ####1}%
  \def\bbl@sclp##1{%
    \count@\z@ % dangerous if a hook is used
    \@for\bbl@tempm:=##1\do{%
      \advance\count@\@ne
      \toks@\expandafter{\bbl@tempm}%
      \edef\bbl@tempn{%
        \expandafter\noexpand
        \csname\bbl@templ{\romannumeral\count@}\endcsname%
        {\the\toks@}}%
      \expandafter\SetString\bbl@tempn}}%
  \def\SetCase{%
    \@ifundefined{bbl@tolower}{%
      \g@addto@macro\@uclclist{%
        \reserved@b{\reserved@b\@gobble}% stops processing the list
        \@ifundefined{\languagename @bbl@uclc}% and resumes it
          {\reserved@a}%
          {\csname\languagename @bbl@uclc\endcsname}%
        {\bbl@tolower\@empty}{\bbl@toupper\@empty}}%
      \gdef\bbl@tolower{\csname\languagename @bbl@lc\endcsname}%
      \gdef\bbl@toupper{\csname\languagename @bbl@uc\endcsname}}{}%
    \let\SetCase\bbl@setcase
    \SetCase}%
  \def\bbl@provstring##1{%
    \@ifundefined{\expandafter\@gobble\string##1}{\gdef##1}\@gobble}%
  \def\bbl@dftstring##1##2{%
```

```
    \@dec@text@cmd\gdef##1?{##2}%
    \global\let##1##1}%
  \def\bbl@encstring##1##2{%
    \bbl@for\bbl@tempc\bbl@sc@fontenc{%
      \@ifundefined{T@\bbl@tempc}%
        \@empty
        {\@dec@text@cmd\gdef##1\bbl@tempc{##2}%
         \global\let##1##1}}}%
  \let\StartBabelCommands\bbl@startcmds
  \begingroup
  \let\bbl@screset\@nnil % local flag - disable 1st stopcommands
  \StartBabelCommands}
\def\bbl@startcmds#1#2{%
  \ifx\bbl@screset\@nnil\else
    \bbl@usehooks{stopcommands}{}%
  \fi
  \endgroup
  \begingroup
  \edef\bbl@L{\zap@space#1 \@empty}%
  \edef\bbl@G{\zap@space#2 \@empty}%
  \let\bbl@sc@charset\space
  \let\bbl@sc@fontenc\space
  \let\SetString\@gobbletwo
  \let\bbl@stringdef\@gobbletwo
  \bbl@startcmds@i}
```

Parse the encoding info to get the label, input, and font parts.
Select the behaviour of `\SetString`. Thre are two main cases: `*` blocks, which are always taken into account, and labelled blocks, which are not always taken into account. With `*` and `strings` set to `encoded` or `generic`, 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.

```
\newcommand\bbl@startcmds@i[1][\@empty]{%
  \ifx\@empty#1%
    \def\bbl@sc@label{generic}%
    \bbl@scswitch{%
      \ifx\bbl@opt@strings\@nnil
        \let\bbl@stringdef\bbl@dftstring
      \else\ifx\bbl@opt@strings\relax
        \let\SetString\bbl@setstring
        \let\bbl@stringdef\bbl@dftstring
      \else
        \let\SetString\bbl@setstring
        \let\bbl@stringdef\bbl@provstring
```

```
      \fi\fi}%
    \@expandtwoargs
    \bbl@usehooks{defaultcommands}{}%
  \else
    \def\bbl@tempa##1=##2\@nil{%
      \bbl@csarg\edef{sc@\zap@space##1 \@empty}{##2 }}%
    \bbl@for\bbl@tempb{label=#1}{\expandafter\bbl@tempa\bbl@tempb\@nil}%
    \def\bbl@tempa##1 ##2{%
      ##1%
      \ifx\@empty##2\else\ifx,##1,\else,\fi\bbl@afterfi\bbl@tempa##2\fi}%
    \edef\bbl@sc@fontenc{\expandafter\bbl@tempa\bbl@sc@fontenc\@empty}%
    \edef\bbl@sc@label{\expandafter\zap@space\bbl@sc@label\@empty}%
    \edef\bbl@sc@charset{\expandafter\zap@space\bbl@sc@charset\@empty}%
    \bbl@scswitch{%
      \ifx\bbl@opt@strings\@nnil
        \let\bbl@stringdef\bbl@encstring
      \else\ifx\bbl@opt@strings\relax
        \let\SetString\bbl@setstring
        \let\bbl@stringdef\bbl@encstring
      \else
      \@expandtwoargs
      \in@{,\bbl@opt@strings,}{,\bbl@sc@label,\bbl@sc@fontenc,}%
      \ifin@
        \let\SetString\bbl@setstring
        \let\bbl@stringdef\bbl@provstring
      \fi\fi\fi}%
    \@expandtwoargs
    \bbl@usehooks{encodedcommands}{{\bbl@sc@charset}{\bbl@sc@fontenc}}%
  \fi}
```

There are two versions of `\bbl@scswitch`. The first version is used when `ldf`s are read, and with it blocks not intented for the current language (as set in `\CurrentOption`) are ignored; it also makes sure $\backslash\langle group\rangle\langle language\rangle$ 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 defines strings for all known languages.

```
\def\bbl@scswitch#1{%
  \@expandtwoargs\in@{,\CurrentOption,}{,\bbl@L,}%
  \ifin@
    \let\bbl@L\CurrentOption
    #1\relax
    \bbl@scswitch@i
    \ifx\bbl@G\@empty\else
      \ifx\SetString\@gobbletwo\else
        \edef\bbl@GL{\bbl@G\bbl@L}%
        \@expandtwoargs\in@{,\bbl@GL,}{,\bbl@screset,}%
        \ifin@\else
          \global\expandafter\let\csname\bbl@GL\endcsname\@undefined
          \xdef\bbl@screset{\bbl@screset,\bbl@GL}%
        \fi
      \fi
```

```
    \fi
  \fi}
\def\bbl@scswitch@i{%
  \ifx\bbl@G\@empty
    \def\SetString##1##2{%
      \bbl@error{Missing group for string \string##1}%
        {You must assign strings to some category, typically\\%
         captions or extras, but you set none}}%
  \fi}
\AtEndOfPackage{\def\bbl@scswitch#1{#1\relax\bbl@scswitch@i}}
\@onlypreamble\EndBabelCommands
\def\EndBabelCommands{%
  \bbl@usehooks{stopcommands}{}%
  \endgroup
  \endgroup}
```

First save the "switcher". Create it if undefined. Strings are defined only if undefined (ie, like `\providescommmand`). 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.

```
\def\bbl@scset#1#2{\def#1{#2}}
\def\bbl@setstring#1#2{%
  \bbl@for\bbl@tempa\bbl@L{%
    \edef\bbl@LC{\bbl@tempa\expandafter\@gobble\string#1}%
    \@ifundefined{\bbl@LC}% eg, \germanchaptername
      {\global\expandafter
       \bbl@add\csname\bbl@G\bbl@tempa\expandafter\endcsname\expandafter
         {\expandafter\bbl@scset\expandafter#1\csname\bbl@LC\endcsname}}%
      {}%
    \def\BabelString{#2}%
    \bbl@usehooks{stringprocess}{}%
    \expandafter\bbl@stringdef
      \csname\bbl@LC\expandafter\endcsname\expandafter{\BabelString}}}
\newcommand\bbl@setcase[3][]{%
  \bbl@for\bbl@tempa\bbl@L{%
    \expandafter\bbl@stringdef
      \csname\bbl@tempa @bbl@uclc\endcsname{\reserved@a#1}%
    \expandafter\bbl@stringdef
      \csname\bbl@tempa @bbl@uc\endcsname{#2}%
    \expandafter\bbl@stringdef
      \csname\bbl@tempa @bbl@lc\endcsname{#3}}}
```

7.17 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 vety little to catch errors, but it is intended for

developpers, after all. `\bbl@usehooks` is the commands used by babel to execute hooks defined for an event.

```
\def\AddBabelHook#1#2{%
  \@ifundefined{bbl@hk@#1}{\EnableBabelHook{#1}}{}%
  \def\bbl@tempa##1,#2=##2,##3\@empty{\def\bbl@tempb{##2}}%
  \expandafter\bbl@tempa\bbl@evargs,#2=,\@empty
  \@ifundefined{bbl@ev@#1@#2}%
    {\bbl@csarg\bbl@add{ev@#2}{\bbl@elt{#1}}%
     \bbl@csarg\newcommand}%
    {\bbl@csarg\renewcommand}%
  {ev@#1@#2}[\bbl@tempb]}
\def\EnableBabelHook#1{\bbl@csarg\let{hk@#1}\@firstofone}
\def\DisableBabelHook#1{\bbl@csarg\let{hk@#1}\@gobble}
\def\bbl@usehooks#1#2{%
  \def\bbl@elt##1{%
    \@nameuse{bbl@hk@##1}{\@nameuse{bbl@ev@##1@#1}#2}}%
  \@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).

```
\def\bbl@evargs{,% don't delete the comma
  everylanguage=1,loadkernel=1,loadpatterns=1,loadexceptions=1,%
  adddialect=2,patterns=2,defaultcommands=0,encodedcommands=2,write=0,%
  beforeextras=0,afterextras=0,stopcommands=0,stringprocess=0,%
  hyphenation=2}
\ifx\directlua\@undefined
  \ifx\XeTeXinputencoding\@undefined\else
    \input xebabel.def
  \fi
\else
  \input luababel.def
\fi
```

7.18 The redefinition of the style commands

The rest of the code in this file can only be processed by LaTeX, so we check the current format. If it is plain TeX, 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 TeX 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`.

```
{\def\format{lplain}
\ifx\fmtname\format
\else
  \def\format{LaTeX2e}
  \ifx\fmtname\format
```

```
  \else
    \aftergroup\endinput
  \fi
\fi}
```

7.19 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 TeXbook [1] (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.

```
%\bbl@redefine\newlabel#1#2{%
%  \@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.
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.

```
\ifx\bbl@opt@safe\@empty\else
  \def\@newl@bel#1#2#3{%
   {\@safe@activestrue
    \@ifundefined{#1@#2}%
       \relax
       {\gdef\@multiplelabels{%
          \@latex@warning@no@line{There were multiply-defined labels}}%
        \@latex@warning@no@line{Label `#2' multiply defined}}%
    \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. Therefor LaTeX keeps reporting that the labels may have changed.

```
\CheckCommand*\@testdef[3]{%
  \def\reserved@a{#3}%
  \expandafter\ifx\csname#1@#2\endcsname\reserved@a
  \else
    \@tempswatrue
  \fi}
```

Now that we made sure that `\@testdef` still has the same definition we can rewrite it. First we make the shorthands 'safe'.

```
\def\@testdef#1#2#3{%
  \@safe@activestrue
```

Then we use `\bbl@tempa` as an 'alias' for the macro that contains the label which is being checked.

```
  \expandafter\let\expandafter\bbl@tempa\csname #1@#2\endcsname
```

Then we define `\bbl@tempb` just as `\@newl@bel` does it.

```
  \def\bbl@tempb{#3}%
  \@safe@activesfalse
```

When the label is defined we replace the definition of `\bbl@tempa` by its meaning.

```
  \ifx\bbl@tempa\relax
  \else
    \edef\bbl@tempa{\expandafter\strip@prefix\meaning\bbl@tempa}%
  \fi
```

We do the same for `\bbl@tempb`.

```
  \edef\bbl@tempb{\expandafter\strip@prefix\meaning\bbl@tempb}%
```

If the label didn't change, `\bbl@tempa` and `\bbl@tempb` should be identical macros.

```
  \ifx\bbl@tempa\bbl@tempb
  \else
    \@tempswatrue
  \fi}
\fi
```

`\ref` `\pageref`

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.

```
\@expandtwoargs\in@{R}\bbl@opt@safe
\ifin@
  \bbl@redefinerobust\ref#1{%
    \@safe@activestrue\org@ref{#1}\@safe@activesfalse}
  \bbl@redefinerobust\pageref#1{%
    \@safe@activestrue\org@pageref{#1}\@safe@activesfalse}
\else
  \let\org@ref\ref
  \let\org@pageref\pageref
\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.

```
\@expandtwoargs\in@{B}\bbl@opt@safe
\ifin@
  \bbl@redefine\@citex[#1]#2{%
    \@safe@activestrue\edef\@tempa{#2}\@safe@activesfalse
    \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.

```
  \AtBeginDocument{%
    \@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.)

```
      \def\@citex[#1][#2]#3{%
        \@safe@activestrue\edef\@tempa{#3}\@safe@activesfalse
        \org@@citex[#1][#2]{\@tempa}}%
  }{}}
```

The package `cite` has a definition of `\@citex` where the shorthands need to be turned off in both arguments.

```
  \AtBeginDocument{%
    \@ifpackageloaded{cite}{%
      \def\@citex[#1]#2{%
        \@safe@activestrue\org@@citex[#1]{#2}\@safe@activesfalse}%
      }{}}
```

`\nocite` The macro `\nocite` which is used to instruct BiBTEX to extract uncited references from the database.

```
  \bbl@redefine\nocite#1{%
    \@safe@activestrue\org@nocite{#1}\@safe@activesfalse}
```

`\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@activestrue` 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.

```
  \bbl@redefine\bibcite{%
```

We call `\bbl@cite@choice` to select the proper definition for `\bibcite`. This new definition is then activated.

```
    \bbl@cite@choice
    \bibcite}
```

`\bbl@bibcite` The macro `\bbl@bibcite` holds the definition of `\bibcite` needed when neither `natbib` nor `cite` is loaded.

```
\def\bbl@bibcite#1#2{%
  \org@bibcite{#1}{\@safe@activesfalse#2}}
```

`\bbl@cite@choice` The macro `\bbl@cite@choice` determines which definition of `\bibcite` is needed.

```
\def\bbl@cite@choice{%
```

First we give `\bibcite` its default definition.

```
  \global\let\bibcite\bbl@bibcite
```

Then, when `natbib` is loaded we restore the original definition of `\bibcite` .

```
  \@ifpackageloaded{natbib}{\global\let\bibcite\org@bibcite}{}%
```

For `cite` we do the same.

```
  \@ifpackageloaded{cite}{\global\let\bibcite\org@bibcite}{}%
```

Make sure this only happens once.

```
  \global\let\bbl@cite@choice\relax}
```

When a document is run for the first time, no `.aux` file is available, and `\bibcite` will not yet be properly defined. In this case, this has to happen before the document starts.

```
\AtBeginDocument{\bbl@cite@choice}
```

`\@bibitem` One of the two internal LaTeX macros called by `\bibitem` that write the citation label on the `.aux` file.

```
\bbl@redefine\@bibitem#1{%
  \@safe@activestrue\org@@bibitem{#1}\@safe@activesfalse}
\else
  \let\org@nocite\nocite
  \let\org@@citex\@citex
  \let\org@bibcite\bibcite
  \let\org@@bibitem\@bibitem
\fi
```

7.20 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.

```
\bbl@redefine\markright#1{%
```

First of all we temporarily store the language switching command, using an expanded definition in order to get the current value of `\languagename`.

```
\edef\bbl@tempb{\noexpand\protect
  \noexpand\foreignlanguage{\languagename}}%
```

Then, we check whether the argument is empty; if it is, we just make sure the scratch token register is empty.

```
\def\bbl@arg{#1}%
\ifx\bbl@arg\@empty
  \toks@{}%
\else
```

Next, we store the argument to `\markright` in the scratch token register, together with the expansion of `\bbl@tempb` (containing the language switching command) as defined before. This way these commands will not be expanded by using `\edef` later on, 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@activestrue` is in effect.

```
  \expandafter\toks@\expandafter{%
            \bbl@tempb{\protect\bbl@restore@actives#1}}%
\fi
```

Then we define a temporary control sequence using `\edef`.

```
\edef\bbl@tempa{%
```

When `\bbl@tempa` is executed, only `\languagename` will be expanded, because of the way the token register was filled.

```
  \noexpand\org@markright{\the\toks@}}%
\bbl@tempa
}
```

`\markboth` `\@mkboth` 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`. Therefor we need to check whether `\@mkboth` has already been set. If so we neeed to do that again with the new definition of `\makrboth`.

```
\ifx\@mkboth\markboth
  \def\bbl@tempc{\let\@mkboth\markboth}
\else
  \def\bbl@tempc{}
\fi
```

Now we can start the new definition of `\markboth`

```
\bbl@redefine\markboth#1#2{%
  \edef\bbl@tempb{\noexpand\protect
    \noexpand\foreignlanguage{\languagename}}%
  \def\bbl@arg{#1}%
  \ifx\bbl@arg\@empty
    \toks@{}%
  \else
```

```
    \expandafter\toks@\expandafter{%
              \bbl@tempb{\protect\bbl@restore@actives#1}}%
  \fi
  \def\bbl@arg{#2}%
  \ifx\bbl@arg\@empty
    \toks8{}%
  \else
    \expandafter\toks8\expandafter{%
              \bbl@tempb{\protect\bbl@restore@actives#2}}%
  \fi
  \edef\bbl@tempa{%
    \noexpand\org@markboth{\the\toks@}{\the\toks8}}%
  \bbl@tempa
}
```

and copy it to `\@mkboth` if necesary.

```
\bbl@tempc
```

7.21 Preventing clashes with other packages

7.21.1 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.

```
\@expandtwoargs\in@{R}\bbl@opt@safe
\ifin@
  \AtBeginDocument{%
    \@ifpackageloaded{ifthen}{%
```

Then we can redefine `\ifthenelse`:

```
      \bbl@redefine@long\ifthenelse#1#2#3{%
```

We want to revert the definition of `\pageref` and `\ref` to their original definition for the duration of `\ifthenelse`, so we first need to store their current meanings.

```
        \let\bbl@tempa\pageref
        \let\pageref\org@pageref
        \let\bbl@tempb\ref
        \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.

```
      \@safe@activestrue
      \org@ifthenelse{#1}{%
        \let\pageref\bbl@tempa
        \let\ref\bbl@tempb
        \@safe@activesfalse
        #2}{%
        \let\pageref\bbl@tempa
        \let\ref\bbl@tempb
        \@safe@activesfalse
        #3}%
      }%
    }{}%
  }
```

7.21.2 varioref

`\@@vpageref` `\vrefpagenum` `\Ref`

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`.

```
\AtBeginDocument{%
  \@ifpackageloaded{varioref}{%
    \bbl@redefine\@@vpageref#1[#2]#3{%
      \@safe@activestrue
      \org@@@vpageref{#1}[#2]{#3}%
      \@safe@activesfalse}%
```

The same needs to happen for `\vrefpagenum`.

```
    \bbl@redefine\vrefpagenum#1#2{%
      \@safe@activestrue
      \org@vrefpagenum{#1}{#2}%
      \@safe@activesfalse}%
```

The package `varioref` defines `\Ref` to be a robust command wich uppercases the first character of the reference text. In order to be able to do that it needs to access the exandable form of `\ref`. So we employ a little trick here. We redefine the (internal) command `\Ref␣` to call `\org@ref` instead of `\ref`. The disadvantgage of this solution is that whenever the derfinition of `\Ref` changes, this definition needs to be updated as well.

```
    \expandafter\def\csname Ref \endcsname#1{%
      \protected@edef\@tempa{\org@ref{#1}}\expandafter\MakeUppercase\@tempa}
    }{}%
  }
\fi
```

7.21.3 hhline

\hhline Delaying the activation of the shorthand characters has introduced a problem with the `hhline` package. The reason is that it uses the ':' character which is made active by the french support in babel. Therefor we need to *reload* the package when the ':' is an active character.

So at `\begin{document}` we check whether `hhline` is loaded.

```
\AtEndOfPackage{%
  \AtBeginDocument{%
    \@ifpackageloaded{hhline}%
```

Then we check whether the expansion of `\normal@char:` is not equal to `\relax`.

```
      {\expandafter\ifx\csname normal@char\string:\endcsname\relax
       \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.

```
         \makeatletter
         \def\@currname{hhline}\input{hhline.sty}\makeatother
       \fi}%
      {}}}
```

7.21.4 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 removed for the moment because `hyperref` is expecting it, .

```
\AtBeginDocument{%
  \@ifundefined{pdfstringdefDisableCommands}%
    {}%
    {\pdfstringdefDisableCommands{%
       \languageshorthands{system}}%
    }%
}
```

7.21.5 fancyhdr

\FOREIGNLANGUAGE The package `fancyhdr` treats the running head and fout 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.

```
\DeclareRobustCommand{\FOREIGNLANGUAGE}[1]{%
  \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.

```
\def\substitutefontfamily#1#2#3{%
  \lowercase{\immediate\openout15=#1#2.fd\relax}%
  \immediate\write15{%
    \string\ProvidesFile{#1#2.fd}%
    [\the\year/\two@digits{\the\month}/\two@digits{\the\day}
     \space generated font description file]^^J
    \string\DeclareFontFamily{#1}{#2}{}^^J
    \string\DeclareFontShape{#1}{#2}{m}{n}{<->ssub * #3/m/n}{}^^J
    \string\DeclareFontShape{#1}{#2}{m}{it}{<->ssub * #3/m/it}{}^^J
    \string\DeclareFontShape{#1}{#2}{m}{sl}{<->ssub * #3/m/sl}{}^^J
    \string\DeclareFontShape{#1}{#2}{m}{sc}{<->ssub * #3/m/sc}{}^^J
    \string\DeclareFontShape{#1}{#2}{b}{n}{<->ssub * #3/bx/n}{}^^J
    \string\DeclareFontShape{#1}{#2}{b}{it}{<->ssub * #3/bx/it}{}^^J
    \string\DeclareFontShape{#1}{#2}{b}{sl}{<->ssub * #3/bx/sl}{}^^J
    \string\DeclareFontShape{#1}{#2}{b}{sc}{<->ssub * #3/bx/sc}{}^^J
    }%
  \closeout15
  }
```

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

```
\@onlypreamble\substitutefontfamily
```

7.22 Encoding issues (part 2)

\TeX \LaTeX Because documents may use font encodings other than one of the latin encodings, we make sure that the logos of TEX and LATEX always come out in the right encoding.

```
\bbl@redefine\TeX{\textlatin{\org@TeX}}
\bbl@redefine\LaTeX{\textlatin{\org@LaTeX}}
```

\nfss@catcodes LATEX's font selection scheme sometimes wants to read font definition files in the middle of processing the document. In order to guard against any characters having the wrong \catcodes it always calls \nfss@catcodes before loading a file. Unfortunately, the characters " and ' are not dealt with. Therefor we have to add them until LATEX does that itself.

```
\bbl@add\nfss@catcodes{%
  \@makeother\'%
  \@makeother\"}
```

7.23 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 norsk.cfg will be loaded when the language definition file norsk.ldf is loaded.

For plain-based formats we don't want to override the definition of `\loadlocalcfg` from `plain.def`.

```
\ifx\loadlocalcfg\@undefined
  \@ifpackagewith{babel}{noconfigs}%
    {\let\loadlocalcfg\@gobble}%
    {\def\loadlocalcfg#1{%
      \InputIfFileExists{#1.cfg}%
        {\typeout{**********************************^^J%
                        * Local config file #1.cfg used^^J%
                        *}}%
        \@empty}}
\fi
```

Just to be compatible with LaTeX 2.09 we add a few more lines of code:

```
\ifx\@unexpandable@protect\@undefined
  \def\@unexpandable@protect{\noexpand\protect\noexpand}
  \long\def\protected@write#1#2#3{%
    \begingroup
      \let\thepage\relax
      #2%
      \let\protect\@unexpandable@protect
      \edef\reserved@a{\write#1{#3}}%
      \reserved@a
    \endgroup
    \if@nobreak\ifvmode\nobreak\fi\fi}
\fi
```

Finally, the default is to use English as the main language.

```
\ifx\l@english\@undefined
  \chardef\l@english\z@
\fi
\main@language{english}
```

⟨/core⟩

Now that we're sure that the code is seen by LaTeX only, we have to find out what the main (primary) document style is because we want to redefine some macros. This is only necessary for releases of LaTeX dated before December 1991. Therefor this part of the code can optionally be included in `babel.def` by specifying the `docstrip` option `names`.

⟨*names⟩

The standard styles can be distinguished by checking whether some macros are defined. In table 1 an overview is given of the macros that can be used for this purpose. The macros that have to be redefined for the `report` and `book` document styles happen to be the same, so there is no need to distinguish between those two styles.

`\doc@style` First a parameter `\doc@style` is defined to identify the current document style. This parameter might have been defined by a document style that already uses macros instead of hard-wired texts, such as `artikel1.sty` [6], so the existence of `\doc@style`

article	:	both the `\chapter` and `\opening` macros are undefined
report and book	:	the `\chapter` macro is defined and the `\opening` is undefined
letter	:	the `\chapter` macro is undefined and the `\opening` is defined

Table 1: How to determine the main document style

is checked. If this macro is undefined, i. e., if the document style is unknown and could therefore contain hard-wired texts, `\doc@style` is defined to the default value '0'.

```
\ifx\@undefined\doc@style
  \def\doc@style{0}%
```

This parameter is defined in the following `if` construction (see table 1):

```
  \ifx\@undefined\opening
    \ifx\@undefined\chapter
      \def\doc@style{1}%
    \else
      \def\doc@style{2}%
    \fi
  \else
    \def\doc@style{3}%
  \fi%
\fi%
```

7.23.1 Redefinition of macros

Now here comes the real work: we start to redefine things and replace hard-wired texts by macros. These redefinitions should be carried out conditionally, in case it has already been done.

For the `figure` and `table` environments we have in all styles:

```
\@ifundefined{figurename}{\def\fnum@figure{\figurename{} \thefigure}}{}
\@ifundefined{tablename}{\def\fnum@table{\tablename{} \thetable}}{}
```

The rest of the macros have to be treated differently for each style. When `\doc@style` still has its default value nothing needs to be done.

```
\ifcase \doc@style\relax
\or
```

This means that `babel.def` is read after the `article` style, where no `\chapter` and `\opening` commands are defined[15].

First we have the `\tableofcontents`, `\listoffigures` and `\listoftables`:

```
\@ifundefined{contentsname}%
    {\def\tableofcontents{\section*{\contentsname\@mkboth
```

[15] A fact that was pointed out to me by Nico Poppelier and was already used in Piet van Oostrum's document style option `nl`.

```
          {\uppercase{\contentsname}}{\uppercase{\contentsname}}}%
      \@starttoc{toc}}}{}
\@ifundefined{listfigurename}%
    {\def\listoffigures{\section*{\listfigurename\@mkboth
          {\uppercase{\listfigurename}}{\uppercase{\listfigurename}}}%
     \@starttoc{lof}}}{}
\@ifundefined{listtablename}%
    {\def\listoftables{\section*{\listtablename\@mkboth
          {\uppercase{\listtablename}}{\uppercase{\listtablename}}}%
      \@starttoc{lot}}}{}
```

Then the `\thebibliography` and `\theindex` environments.

```
\@ifundefined{refname}%
    {\def\thebibliography#1{\section*{\refname
      \@mkboth{\uppercase{\refname}}{\uppercase{\refname}}}%
      \list{[\arabic{enumi}]}{\settowidth\labelwidth{[#1]}%
        \leftmargin\labelwidth
        \advance\leftmargin\labelsep
        \usecounter{enumi}}%
        \def\newblock{\hskip.11em plus.33em minus.07em}%
        \sloppy\clubpenalty4000\widowpenalty\clubpenalty
        \sfcode`\.=1000\relax}}{}
\@ifundefined{indexname}%
    {\def\theindex{\@restonecoltrue\if@twocolumn\@restonecolfalse\fi
     \columnseprule \z@
     \columnsep 35pt\twocolumn[\section*{\indexname}]%
       \@mkboth{\uppercase{\indexname}}{\uppercase{\indexname}}%
       \thispagestyle{plain}%
       \parskip\z@ plus.3pt\parindent\z@\let\item\@idxitem}}{}
```

The `abstract` environment:

```
\@ifundefined{abstractname}%
    {\def\abstract{\if@twocolumn
    \section*{\abstractname}%
    \else \small
    \begin{center}%
    {\bf \abstractname\vspace{-.5em}\vspace{\z@}}%
    \end{center}%
    \quotation
    \fi}}{}
```

And last but not least, the macro `\part`:

```
\@ifundefined{partname}%
{\def\@part[#1]#2{\ifnum \c@secnumdepth >\m@ne
        \refstepcounter{part}%
        \addcontentsline{toc}{part}{\thepart
        \hspace{1em}#1}\else
      \addcontentsline{toc}{part}{#1}\fi
   {\parindent\z@ \raggedright
    \ifnum \c@secnumdepth >\m@ne
```

```
      \Large \bf \partname{} \thepart
      \par \nobreak
    \fi
    \huge \bf
    #2\markboth{}{}\par}%
    \nobreak
    \vskip 3ex\@afterheading}%
}{}
```

This is all that needs to be done for the `article` style.

```
\or
```

The next case is formed by the two styles `book` and `report`. Basically we have to do the same as for the `article` style, except now we must also change the `\chapter` command.

The tables of contents, figures and tables:

```
\@ifundefined{contentsname}%
    {\def\tableofcontents{\@restonecolfalse
      \if@twocolumn\@restonecoltrue\onecolumn
      \fi\chapter*{\contentsname\@mkboth
          {\uppercase{\contentsname}}{\uppercase{\contentsname}}}%
      \@starttoc{toc}%
      \csname if@restonecol\endcsname\twocolumn
      \csname fi\endcsname}}{}
\@ifundefined{listfigurename}%
    {\def\listoffigures{\@restonecolfalse
      \if@twocolumn\@restonecoltrue\onecolumn
      \fi\chapter*{\listfigurename\@mkboth
          {\uppercase{\listfigurename}}{\uppercase{\listfigurename}}}%
      \@starttoc{lof}%
      \csname if@restonecol\endcsname\twocolumn
      \csname fi\endcsname}}{}
\@ifundefined{listtablename}%
    {\def\listoftables{\@restonecolfalse
      \if@twocolumn\@restonecoltrue\onecolumn
      \fi\chapter*{\listtablename\@mkboth
          {\uppercase{\listtablename}}{\uppercase{\listtablename}}}%
      \@starttoc{lot}%
      \csname if@restonecol\endcsname\twocolumn
      \csname fi\endcsname}}{}
```

Again, the `bibliography` and `index` environments; notice that in this case we use `\bibname` instead of `\refname` as in the definitions for the `article` style. The reason for this is that in the `article` document style the term 'References' is used in the definition of `\thebibliography`. In the `report` and `book` document styles the term 'Bibliography' is used.

```
\@ifundefined{bibname}%
    {\def\thebibliography#1{\chapter*{\bibname
     \@mkboth{\uppercase{\bibname}}{\uppercase{\bibname}}}%
     \list{[\arabic{enumi}]}{\settowidth\labelwidth{[#1]}%
```

```
     \leftmargin\labelwidth \advance\leftmargin\labelsep
     \usecounter{enumi}}%
     \def\newblock{\hskip.11em plus.33em minus.07em}%
     \sloppy\clubpenalty4000\widowpenalty\clubpenalty
     \sfcode`\.=1000\relax}}{}
\@ifundefined{indexname}%
    {\def\theindex{\@restonecoltrue\if@twocolumn\@restonecolfalse\fi
    \columnseprule \z@
    \columnsep 35pt\twocolumn[\@makeschapterhead{\indexname}]%
      \@mkboth{\uppercase{\indexname}}{\uppercase{\indexname}}%
    \thispagestyle{plain}%
    \parskip\z@ plus.3pt\parindent\z@ \let\item\@idxitem}}{}
```

Here is the `abstract` environment:

```
\@ifundefined{abstractname}%
    {\def\abstract{\titlepage
    \null\vfil
    \begin{center}%
    {\bf \abstractname}%
    \end{center}}}{}
```

And last but not least the `\chapter`, `\appendix` and `\part` macros.

```
\@ifundefined{chaptername}{\def\@chapapp{\chaptername}}{}
%
\@ifundefined{appendixname}%
    {\def\appendix{\par
      \setcounter{chapter}{0}%
      \setcounter{section}{0}%
      \def\@chapapp{\appendixname}%
      \def\thechapter{\Alph{chapter}}}}{}
%
\@ifundefined{partname}%
    {\def\@part[#1]#2{\ifnum \c@secnumdepth >-2\relax
            \refstepcounter{part}%
            \addcontentsline{toc}{part}{\thepart
            \hspace{1em}#1}\else
            \addcontentsline{toc}{part}{#1}\fi
       \markboth{}{}%
       {\centering
        \ifnum \c@secnumdepth >-2\relax
          \huge\bf \partname{} \thepart
        \par
        \vskip 20pt \fi
        \Huge \bf
        #1\par}\@endpart}}{}%
\or
```

Now we address the case where `babel.def` is read after the `letter` style. The `letter` document style defines the macro `\opening` and some other macros that are specific to `letter`. This means that we have to redefine other macros, compared to the previous two cases.

First two macros for the material at the end of a letter, the `\cc` and `\encl` macros.

```
\@ifundefined{ccname}%
    {\def\cc#1{\par\noindent
     \parbox[t]{\textwidth}%
     {\@hangfrom{\rm \ccname : }\ignorespaces #1\strut}\par}}{}
\@ifundefined{enclname}%
    {\def\encl#1{\par\noindent
     \parbox[t]{\textwidth}%
     {\@hangfrom{\rm \enclname : }\ignorespaces #1\strut}\par}}{}
```

The last thing we have to do here is to redefine the `headings` pagestyle:

```
\@ifundefined{headtoname}%
  {\def\ps@headings{%
     \def\@oddhead{\sl \headtoname{} \ignorespaces\toname \hfil
                       \@date \hfil \pagename{} \thepage}%
     \def\@oddfoot{}}}{}
```

This was the last of the four standard document styles, so if `\doc@style` has another value we do nothing and just close the `if` construction.

```
\fi
```

⟨/names⟩

Here ends the code that can be optionally included when a version of LaTeX is in use that is dated *before* December 1991.
We also need to redefine a number of commands to ensure that the right font encoding is used, but this can't be done before `babel.def` is loaded.

7.24 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.

⟨∗kernel | patterns⟩

```
\ifx\language\@undefined
  \csname newcount\endcsname\language
\fi
```

`\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 TeX's memory plain TeX 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. Therefor `\addlanguage` is defined using a definition based on the macros used to define `\newlanguage` in plain TeX version 3.0.
For formats based on plain version 3.0 the definition of `\newlanguage` can be simply copied, removing `\outer`.

```
\ifx\newlanguage\@undefined
  \csname newcount\endcsname\last@language
  \def\addlanguage#1{%
    \global\advance\last@language\@ne
    \ifnum\last@language<\@cclvi
    \else
        \errmessage{No room for a new \string\language!}%
    \fi
    \global\chardef#1\last@language
    \wlog{\string#1 = \string\language\the\last@language}}
```

plain TEX version 3.0 uses `\count 19` for this purpose.

```
\else
  \countdef\last@language=19
  \def\addlanguage{\alloc@9\language\chardef\@cclvi}
\fi
```

⟨/kernel | patterns⟩

`\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.

```
⟨*kernel⟩
\def\adddialect#1#2{%
  \global\chardef#1#2\relax
  \bbl@usehooks{adddialect}{{#1}{#2}}%
  \wlog{\string#1 = a dialect from \string\language#2}}
```

`\bbl@iflanguage` executes code only of the language exists. Otherwise raises and 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 intented 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).

```
\def\bbl@fixname#1{%
  \begingroup
    \def\bbl@tempe{l@}%
    \edef\bbl@tempd{\noexpand\@ifundefined{\noexpand\bbl@tempe#1}}%
    \bbl@tempd
      {\lowercase\expandafter{\bbl@tempd}%
         {\uppercase\expandafter{\bbl@tempd}%
           \@empty
           {\edef\bbl@tempd{\def\noexpand#1{#1}}%
            \uppercase\expandafter{\bbl@tempd}}}%
         {\edef\bbl@tempd{\def\noexpand#1{#1}}%
          \lowercase\expandafter{\bbl@tempd}}}%
      \@empty
    \edef\bbl@tempd{\endgroup\def\noexpand#1{#1}}%
  \bbl@tempd}
\def\bbl@iflanguage#1{%
  \@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.

```
\def\iflanguage#1{%
  \bbl@iflanguage{#1}{%
    \ifnum\csname l@#1\endcsname=\language
      \expandafter\@firstoftwo
    \else
      \expandafter\@secondoftwo
    \fi}}
```

\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.

```
\let\bbl@select@type\z@
\edef\selectlanguage{%
  \noexpand\protect
  \expandafter\noexpand\csname selectlanguage \endcsname}
```

Because the command \selectlanguage could be used in a moving argument it expands to \protect\selectlanguage␣. Therefor, we have to make sure that a macro \protect exists. If it doesn't it is \let to \relax.

```
\ifx\@undefined\protect\let\protect\relax\fi
```

As LaTeX 2.09 writes to files *expanded* whereas LaTeX 2ε 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. Therefor we introduce the macro \xstring which should expand to the right amount of \string's.

```
\ifx\documentclass\@undefined
```

```
\def\xstring{\string\string\string}
\else
  \let\xstring\string
\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. Therefor 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.

```
\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`
`\bbl@pop@language`

The stack is simply a list of languagenames, separated with a '+' sign; the push function can be simple:

```
\def\bbl@push@language{%
  \xdef\bbl@language@stack{\languagename+\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 `\languagename`. For this we first define a helper function.

`\bbl@pop@lang`

This macro stores its first element (which is delimited by the '+'-sign) in `\languagename` and stores the rest of the string (delimited by '-') in its third argument.

```
\def\bbl@pop@lang#1+#2-#3{%
  \edef\languagename{#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.

```
\def\bbl@pop@language{%
  \expandafter\bbl@pop@lang\bbl@language@stack-\bbl@language@stack
  \expandafter\bbl@set@language\expandafter{\languagename}}
```

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.

```
\expandafter\def\csname selectlanguage \endcsname#1{%
  \bbl@push@language
  \aftergroup\bbl@pop@language
  \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 historial 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 `\languagename` 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.

```
\def\BabelContentsFiles{toc,lof,lot}%
\def\bbl@set@language#1{%
  \edef\languagename{%
    \ifnum\escapechar=\expandafter`\string#1\@empty
    \else\string#1\@empty\fi}%
  \select@language{\languagename}%
  \expandafter\ifx\csname date\languagename\endcsname\relax\else
    \if@filesw
      \protected@write\@auxout{}{\string\select@language{\languagename}}%
      \bbl@for\bbl@tempa\BabelContentsFiles{%
        \addtocontents{\bbl@tempa}{\xstring\select@language{\languagename}}}%
      \bbl@usehooks{write}{}%
    \fi
  \fi}
\def\select@language#1{%
  \edef\languagename{#1}%
  \bbl@fixname\languagename
  \bbl@iflanguage\languagename{%
    \expandafter\ifx\csname date\languagename\endcsname\relax
      \bbl@error
        {You haven't loaded the language #1\space yet}%
        {You may proceed, but expect unexpected results}%
    \else
      \let\bbl@select@type\z@
      \expandafter\bbl@switch\expandafter{\languagename}%
    \fi}}
% A bit of optmization:
\def\select@language@x#1{%
  \ifcase\bbl@select@type
    \bbl@ifsamestring\languagename{#1}{}{\select@language{#1}}%
  \else
    \select@language{#1}%
```

```
\fi}
```

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 `\languagename`.
Then we have to *re*define `\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.

```
\def\bbl@switch#1{%
  \originalTeX
  \expandafter\def\expandafter\originalTeX\expandafter{%
    \csname noextras#1\endcsname
    \let\originalTeX\@empty
    \babel@beginsave}%
  \languageshorthands{none}%
  \ifcase\bbl@select@type
    \csname captions#1\endcsname
    \csname date#1\endcsname
  \fi
  \bbl@usehooks{beforeextras}{}%
  \csname extras#1\endcsname\relax
  \bbl@usehooks{afterextras}{}%
  \bbl@patterns{#1}%
  \babel@savevariable\lefthyphenmin
  \babel@savevariable\righthyphenmin
  \expandafter\ifx\csname #1hyphenmins\endcsname\relax
    \set@hyphenmins\tw@\thr@@\relax
  \else
    \expandafter\expandafter\expandafter\set@hyphenmins
      \csname #1hyphenmins\endcsname\relax
  \fi}
\def\bbl@ifsamestring#1#2{%
  \protected@edef\bbl@tempb{#1}%
  \edef\bbl@tempb{\expandafter\strip@prefix\meaning\bbl@tempb}%
  \protected@edef\bbl@tempc{#2}%
  \edef\bbl@tempc{\expandafter\strip@prefix\meaning\bbl@tempc}%
  \ifx\bbl@tempb\bbl@tempc
    \expandafter\@firstoftwo
  \else
    \expandafter\@secondoftwo
```

```
\fi}
```

`otherlanguage` The otherlanguage 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 first thing this environment does is store the name of the language in `\languagename`; it then calls `\selectlanguage`␣ to switch on everything that is needed for this language The `\ignorespaces` command is necessary to hide the environment when it is entered in horizontal mode.

```
\long\def\otherlanguage#1{%
  \csname selectlanguage \endcsname{#1}%
  \ignorespaces}
```

The `\endotherlanguage` part of the environment tries to hide itself when it is called in horizontal mode.

```
\long\def\endotherlanguage{%
  \global\@ignoretrue\ignorespaces}
```

`otherlanguage*` The otherlanguage 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`.

```
\expandafter\def\csname otherlanguage*\endcsname#1{%
  \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".

```
\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`.

```
\edef\foreignlanguage{%
  \noexpand\protect
  \expandafter\noexpand\csname foreignlanguage \endcsname}
\expandafter\def\csname foreignlanguage \endcsname#1#2{%
  \begingroup
    \foreign@language{#1}%
    #2%
  \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`.

```
\def\foreign@language#1{%
  \edef\languagename{#1}%
  \bbl@fixname\languagename
  \bbl@iflanguage\languagename{%
    \expandafter\ifx\csname date\languagename\endcsname\relax
      \bbl@warning
        {You haven't loaded the language #1\space yet\\%
         I'll proceed, but expect unexpected results.\\%
         Reported}%
    \fi
    \let\bbl@select@type\@ne
    \expandafter\bbl@switch\expandafter{\languagename}}}
```

`\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.

```
\let\bbl@hyphlist\@empty
\let\bbl@hyphenation@\relax
\def\bbl@patterns#1{%
  \language=\expandafter\ifx\csname l@#1:\f@encoding\endcsname\relax
      \csname l@#1\endcsname
      \edef\bbl@tempa{#1}%
    \else
      \csname l@#1:\f@encoding\endcsname
      \edef\bbl@tempa{#1:\f@encoding}%
    \fi\relax
  \@expandtwoargs\bbl@usehooks{patterns}{{#1}{\bbl@tempa}}%
  \@ifundefined{bbl@hyphenation@}{}{%
    \begingroup
      \@expandtwoargs\in@{,\number\language,}{,\bbl@hyphlist}%
      \ifin@\else
        \@expandtwoargs\bbl@usehooks{hyphenation}{{#1}{\bbl@tempa}}%
        \hyphenation{%
          \bbl@hyphenation@
          \@ifundefined{bbl@hyphenation@#1}%
            \@empty
            {\space\csname bbl@hyphenation@#1\endcsname}}%
        \xdef\bbl@hyphlist{\bbl@hyphlist\number\language,}%
      \fi
    \endgroup}}
```

`hyphenrules` The environment hyphenrules can be used to select *just* the hyphenation rules. This environment does *not* change `\languagename` 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 `otherlanguage*`.

```
\def\hyphenrules#1{%
  \edef\languagename{#1}%
  \bbl@fixname\languagename
  \bbl@iflanguage\languagename{%
    \expandafter\bbl@patterns\expandafter{\languagename}%
    \languageshorthands{none}%
    \expandafter\ifx\csname\languagename hyphenmins\endcsname\relax
      \set@hyphenmins\tw@\thr@@\relax
    \else
      \expandafter\expandafter\expandafter\set@hyphenmins
      \csname\languagename hyphenmins\endcsname\relax
    \fi}}
\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 `\`⟨*lang*⟩`hyphenmins` is already defined this command has no effect.

```
\def\providehyphenmins#1#2{%
  \expandafter\ifx\csname #1hyphenmins\endcsname\relax
    \@namedef{#1hyphenmins}{#2}%
  \fi}
```

`\set@hyphenmins` This macro sets the values of `\lefthyphenmin` and `\righthyphenmin`. It expects two values as its argument.

```
\def\set@hyphenmins#1#2{\lefthyphenmin#1\relax\righthyphenmin#2\relax}
```

`\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`

```
\def\LdfInit{%
  \chardef\atcatcode=\catcode`\@
  \catcode`\@=11\relax
  \input babel.def\relax
  \catcode`\@=\atcatcode \let\atcatcode\relax
  \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`.

```
\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.

```
\ifx\babel@beginsave\@undefined\let\babel@beginsave\relax\fi
```

`\@nolanerr` `\@nopatterns` 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_ε, so we can safely use its error handling interface. Otherwise we'll have to 'keep it simple'.

```
\edef\bbl@nulllanguage{\string\language=0}
\ifx\PackageError\@undefined
  \def\bbl@error#1#2{%
    \begingroup
      \newlinechar=`\^^J
      \def\\{^^J(babel) }%
      \errhelp{#2}\errmessage{\\#1}%
    \endgroup}
  \def\bbl@warning#1{%
    \begingroup
      \newlinechar=`\^^J
      \def\\{^^J(babel) }%
      \message{\\#1}%
    \endgroup}
  \def\bbl@info#1{%
    \begingroup
      \newlinechar=`\^^J
      \def\\{^^J}%
      \wlog{#1}%
    \endgroup}
\else
  \def\bbl@error#1#2{%
    \begingroup
      \def\\{\MessageBreak}%
      \PackageError{babel}{#1}{#2}%
    \endgroup}
  \def\bbl@warning#1{%
    \begingroup
      \def\\{\MessageBreak}%
      \PackageWarning{babel}{#1}%
    \endgroup}
  \def\bbl@info#1{%
    \begingroup
      \def\\{\MessageBreak}%
      \PackageInfo{babel}{#1}%
    \endgroup}
\fi
```

```
\def\@nolanerr#1{%
  \bbl@error
    {You haven't defined the language #1\space yet}%
    {Your command will be ignored, type <return> to proceed}}
\def\@nopatterns#1{%
  \bbl@warning
    {No hyphenation patterns were preloaded for\\%
     the language `#1' into the format.\\%
     Please, configure your TeX system to add them and\\%
     rebuild the format. Now I will use the patterns\\%
     preloaded for \bbl@nulllanguage\space instead}}
\let\bbl@usehooks\@gobbletwo
⟨/kernel⟩
```

The following code is meant to be read by iniTEX 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 at lower level than the rest. `toks8` stores info to be shown when the program is run.

```
⟨*patterns⟩
\toks8{Babel <3.9f> 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.

```
\def\process@line#1#2 #3 #4 {%
  \ifx=#1%
    \process@synonym{#2}%
  \else
    \process@language{#1#2}{#3}{#4}%
  \fi
  \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.

```
\toks@{}
\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.

```
\def\process@synonym#1{%
  \ifnum\last@language=\m@ne
    \toks@\expandafter{\the\toks@\relax\process@synonym{#1}}%
  \else
    \expandafter\chardef\csname l@#1\endcsname\last@language
```

```
    \wlog{\string\l@#1=\string\language\the\last@language}%
    \expandafter\let\csname #1hyphenmins\expandafter\endcsname
      \csname\languagename hyphenmins\endcsname
    \let\bbl@elt\relax
    \edef\bbl@languages{\bbl@languages\bbl@elt{#1}{\the\last@language}{}{}}%
  \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 `\bbl@get@enc` extracts the font encoding from the language name and stores it in `\bbl@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. Therefor we try to detect such assignments and store them in the `\`⟨*lang*⟩`hyphenmins` macro. When no assignments were made we provide a default setting.

Some pattern files contain changes to the `\lccode` en `\uccode` arrays. Such changes should remain local to the language; therefor 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.)

`\bbl@languages` saves a snapshot of the loaded languages in the form `\bbl@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.

```
\def\process@language#1#2#3{%
  \expandafter\addlanguage\csname l@#1\endcsname
  \expandafter\language\csname l@#1\endcsname
  \edef\languagename{#1}%
  \bbl@hook@everylanguage{#1}%
  \bbl@get@enc#1::\@@@
  \begingroup
    \lefthyphenmin\m@ne
```

```
    \bbl@hook@loadpatterns{#2}%
    \ifnum\lefthyphenmin=\m@ne
    \else
      \expandafter\xdef\csname #1hyphenmins\endcsname{%
        \the\lefthyphenmin\the\righthyphenmin}%
    \fi
  \endgroup
  \def\bbl@tempa{#3}%
  \ifx\bbl@tempa\@empty\else
    \bbl@hook@loadexceptions{#3}%
  \fi
  \let\bbl@elt\relax
  \edef\bbl@languages{%
    \bbl@languages\bbl@elt{#1}{\the\language}{#2}{\bbl@tempa}}%
  \ifnum\the\language=\z@
    \expandafter\ifx\csname #1hyphenmins\endcsname\relax
      \set@hyphenmins\tw@\thr@@\relax
    \else
      \expandafter\expandafter\expandafter\set@hyphenmins
        \csname #1hyphenmins\endcsname
    \fi
    \the\toks@
    \toks@{}%
  \fi}
```

`\bbl@get@enc` `\bbl@hyph@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.

```
\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.

```
\def\bbl@hook@everylanguage#1{}
\def\bbl@hook@loadpatterns#1{\input #1\relax}
\let\bbl@hook@loadexceptions\bbl@hook@loadpatterns
\let\bbl@hook@loadkernel\bbl@hook@loadpatterns
\begingroup
  \def\AddBabelHook#1#2{%
    \expandafter\ifx\csname bbl@hook@#2\endcsname\relax
      \def\next{\toks1}%
    \else
      \def\next{\expandafter\gdef\csname bbl@hook@#2\endcsname####1}%
    \fi
    \next}
  \ifx\directlua\@undefined
    \ifx\XeTeXinputencoding\@undefined\else
      \input xebabel.def
    \fi
  \else
    \input luababel.def
```

```
  \fi
  \openin1 = babel-\bbl@format.cfg
  \ifeof1
  \else
    \input babel-\bbl@format.cfg\relax
  \fi
  \closein1
\endgroup
\bbl@hook@loadkernel{switch.def}
```

\readconfigfile The configuration file can now be opened for reading.

```
\openin1 = language.dat
```

See if the file exists, if not, use the default hyphenation file `hyphen.tex`. The user will be informed about this.

```
\def\languagename{english}%
\ifeof1
  \message{I couldn't find the file language.dat,\space
           I will try the file hyphen.tex}
  \input hyphen.tex\relax
  \chardef\l@english\z@
\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.

```
  \last@language\m@ne
```

We now read lines from the file until the end is found

```
  \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.

```
    \endlinechar\m@ne
    \read1 to \bbl@line
    \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.

```
    \if T\ifeof1F\fi T\relax
      \ifx\bbl@line\@empty\else
        \edef\bbl@line{\bbl@line\space\space\space}%
        \expandafter\process@line\bbl@line\relax
      \fi
  \repeat
```

Check for the end of the file. We must reverse the test for `\ifeof` without `\else`. Then reactivate the default patterns,

```
\begingroup
  \def\bbl@elt#1#2#3#4{%
    \global\language=#2\relax
    \gdef\languagename{#1}%
    \def\bbl@elt##1##2##3##4{}}%
  \bbl@languages
\endgroup
\fi
```

and close the configuration file.

```
\closein1
```

We add a message about the fact that babel is loaded in the format and with which language patterns to the `\everyjob` register.

```
\if/\the\toks@/\else
  \errhelp{language.dat loads no language, only synonyms}
  \errmessage{Orphan language synonym}
\fi
\ifx\addto@hook\@undefined
\else
  \edef\bbl@tempa{%
    \noexpand\typeout{\the\toks8 \the\last@language\space languages
    loaded.}}%
  \expandafter\addto@hook\expandafter\everyjob\expandafter{\bbl@tempa}
\fi
```

Also remove some macros from memory and raise an error if `\toks@` is not empty. Finally load `switch.def`, but the letter is not required and the line inputting it may be commented out.

```
\let\bbl@line\@undefined
\let\process@line\@undefined
\let\process@synonym\@undefined
\let\process@language\@undefined
\let\bbl@get@enc\@undefined
\let\bbl@hyph@enc\@undefined
\let\bbl@tempa\@undefined
\let\bbl@hook@loadkernel\@undefined
\let\bbl@hook@everylanguage\@undefined
\let\bbl@hook@loadpatterns\@undefined
\let\bbl@hook@loadexceptions\@undefined
⟨/patterns⟩
```

Here the code for iniTEX ends.

8 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.

```
⟨*nil⟩
\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.

```
\ifx\l@nohyphenation\@undefined
    \@nopatterns{nil}
    \adddialect\l@nil0
\else
    \let\l@nil\l@nohyphenation
\fi
```

This macro is used to store the values of the hyphenation parameters `\lefthyphenmin` and `\righthyphenmin`.

```
\providehyphenmins{\CurrentOption}{\m@ne\m@ne}
```

The next step consists of defining commands to switch to (and from) the 'nil' language.

`\captionnil`
`\datenil`

```
\let\captionsnil\@empty
\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.

```
\ldf@finish{nil}
⟨/nil⟩
```

9 Support for Plain TEX

9.1 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 localhyphen.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 acheive the desired effect, based on the `babel` package.

If you load each of them with iniTEX, 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 iniTEX sees, we need to set some category codes just to be able to change the definition of `\input`

```
⟨*bplain | blplain⟩
\catcode`\{=1 % left brace is begin-group character
\catcode`\}=2 % right brace is end-group character
\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...

```
\openin 0 hyphen.cfg
```

If the file wasn't found the following test turns out true.

```
\ifeof0
\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).

```
  \let\a\input
```

Then `\input` is defined to forget about its argument and load `hyphen.cfg` instead.

```
  \def\input #1 {%
    \let\input\a
    \a hyphen.cfg
```

Once that's done the original meaning of `\input` can be restored and the definition of `\a` can be forgotten.

```
    \let\a\undefined
  }
\fi
⟨/bplain | blplain⟩
```

Now that we have made sure that `hyphen.cfg` will be loaded at the right moment it is time to load `plain.tex`.

```
⟨bplain⟩\a plain.tex
⟨blplain⟩\a 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.

```
⟨bplain⟩\def\fmtname{babel-plain}
⟨blplain⟩\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.

9.2 Emulating some LaTeX features

The following code duplicates or emulates parts of LaTeX 2_ε that are needed for babel. When `\adddialect` is still undefined we are making a format. In that case only the first part of this file is needed.

```
2104 ⟨*code⟩
2105 \def\@empty{}
2106 \ifx\orig@dump\@undefined\else
```

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 SLiTeX the above scheme won't work. The reason is that SLiTeX 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 `\org@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`.

```
2107  \def\dump{%
2108    \ifx\@ztryfc\@undefined
2109    \else
2110      \toks0=\expandafter{\@preamblecmds}
2111      \edef\@preamblecmds{\noexpand\@begindocumenthook\the\toks0}
2112      \def\@begindocumenthook{}
2113    \fi
```

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.

```
2114    \everyjob\expandafter{\the\everyjob%
2115      \immediate\write16{\the\toks8 loaded.}}%
```

Then everything is restored to the old situation and the format is dumped.

```
    \let\dump\orig@dump\let\orig@dump\@undefined\dump}
  \expandafter\endinput
\fi
```

The rest of this file is not processed by iniTEX but during the normal document run. We need to define `\loadlocalcfg` for plain users as the LaTeX definition uses `\InputIfFileExists`. We have to execute `\@endofldf` in this case.

```
\def\loadlocalcfg#1{%
  \openin0#1.cfg
  \ifeof0
    \closein0
  \else
    \closein0
    {\immediate\write16{**************************************}%
     \immediate\write16{* Local config file #1.cfg used}%
     \immediate\write16{*}%
     }
    \input #1.cfg\relax
  \fi
  \@endofldf}
```

A number of LaTeX macro's that are needed later on.

```
\long\def\@firstofone#1{#1}
\long\def\@firstoftwo#1#2{#1}
\long\def\@secondoftwo#1#2{#2}
\def\@ifstar#1{\@ifnextchar *{\@firstoftwo{#1}}}
\def\@star@or@long#1{%
  \@ifstar
  {\let\l@ngrel@x\relax#1}%
  {\let\l@ngrel@x\long#1}}
\let\l@ngrel@x\relax
\def\@car#1#2\@nil{#1}
\def\@cdr#1#2\@nil{#2}
\let\@typeset@protect\relax
\long\def\@gobble#1{}
\edef\@backslashchar{\expandafter\@gobble\string\\}
\def\strip@prefix#1>{}
\def\g@addto@macro#1#2{{%
    \toks@\expandafter{#1#2}%
    \xdef#1{\the\toks@}}}
\def\@namedef#1{\expandafter\def\csname #1\endcsname}
\def\@nameuse#1{\csname #1\endcsname}
\def\@ifundefined#1{%
  \expandafter\ifx\csname#1\endcsname\relax
    \expandafter\@firstoftwo
  \else
    \expandafter\@secondoftwo
  \fi}
\def\@expandtwoargs#1#2#3{%
```

```
  \edef\reserved@a{\noexpand#1{#2}{#3}}\reserved@a}
```

LaTeX 2$_\varepsilon$ has the command `\@onlypreamble` which adds commands to a list of commands that are no longer needed after `\begin{document}`.

```
\ifx\@preamblecmds\@undefined
  \def\@preamblecmds{}
\fi
\def\@onlypreamble#1{%
  \expandafter\gdef\expandafter\@preamblecmds\expandafter{%
    \@preamblecmds\do#1}}
\@onlypreamble\@onlypreamble
```

Mimick LaTeX's `\AtBeginDocument`; for this to work the user needs to add `\begindocument` to his file.

```
\def\begindocument{%
  \@begindocumenthook
  \global\let\@begindocumenthook\@undefined
  \def\do##1{\global\let ##1\@undefined}%
  \@preamblecmds
  \global\let\do\noexpand
  }
\ifx\@begindocumenthook\@undefined
  \def\@begindocumenthook{}
\fi
\@onlypreamble\@begindocumenthook
\def\AtBeginDocument{\g@addto@macro\@begindocumenthook}
```

We also have to mimick LaTeX's `\AtEndOfPackage`. Our replacement macro is much simpler; it stores its argument in `\@endofldf`.

```
\def\AtEndOfPackage#1{\g@addto@macro\@endofldf{#1}}
\@onlypreamble\AtEndOfPackage
\def\@endofldf{}
\@onlypreamble\@endofldf
\let\bbl@afterlang\@empty
```

LaTeX needs to be able to switch off writing to its auxiliary files; plain doesn't have them by default.

```
\ifx\if@filesw\@undefined
  \expandafter\let\csname if@filesw\expandafter\endcsname
    \csname iffalse\endcsname
\fi
```

Mimick LaTeX's commands to define control sequences.

```
\def\newcommand{\@star@or@long\new@command}
\def\new@command#1{%
  \@testopt{\@newcommand#1}0}
\def\@newcommand#1[#2]{%
  \@ifnextchar [{\@xargdef#1[#2]}%
                {\@argdef#1[#2]}}
\long\def\@argdef#1[#2]#3{%
  \@yargdef#1\@ne{#2}{#3}}
```

```
\long\def\@xargdef#1[#2][#3]#4{%
  \expandafter\def\expandafter#1\expandafter{%
    \expandafter\@protected@testopt\expandafter #1%
    \csname\string#1\expandafter\endcsname{#3}}%
  \expandafter\@yargdef \csname\string#1\endcsname
  \tw@{#2}{#4}}
\long\def\@yargdef#1#2#3{%
  \@tempcnta#3\relax
  \advance \@tempcnta \@ne
  \let\@hash@\relax
  \edef\reserved@a{\ifx#2\tw@ [\@hash@1]\fi}%
  \@tempcntb #2%
  \@whilenum\@tempcntb <\@tempcnta
  \do{%
    \edef\reserved@a{\reserved@a\@hash@\the\@tempcntb}%
    \advance\@tempcntb \@ne}%
  \let\@hash@##%
  \l@ngrel@x\expandafter\def\expandafter#1\reserved@a}
\let\providecommand\newcommand

\def\DeclareRobustCommand{\@star@or@long\declare@robustcommand}
\def\declare@robustcommand#1{%
   \edef\reserved@a{\string#1}%
   \def\reserved@b{#1}%
   \edef\reserved@b{\expandafter\strip@prefix\meaning\reserved@b}%
   \edef#1{%
      \ifx\reserved@a\reserved@b
         \noexpand\x@protect
         \noexpand#1%
      \fi
      \noexpand\protect
      \expandafter\noexpand\csname
         \expandafter\@gobble\string#1 \endcsname
   }%
   \expandafter\new@command\csname
      \expandafter\@gobble\string#1 \endcsname
}
\def\x@protect#1{%
   \ifx\protect\@typeset@protect\else
      \@x@protect#1%
   \fi
}
\def\@x@protect#1\fi#2#3{%
   \fi\protect#1%
}
```

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`.

```
\def\bbl@tempa{\csname newif\endcsname\ifin@}
\ifx\in@\@undefined
  \def\in@#1#2{%
    \def\in@@##1#1##2##3\in@@{%
      \ifx\in@##2\in@false\else\in@true\fi}%
    \in@@#2#1\in@\in@@}
\else
  \let\bbl@tempa\@empty
\fi
\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).

```
\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.

```
\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 LaTeX 2_ε versions; just enough to make things work in plain TeXenvironments.

```
\ifx\@tempcnta\@undefined
  \csname newcount\endcsname\@tempcnta\relax
\fi
\ifx\@tempcntb\@undefined
  \csname newcount\endcsname\@tempcntb\relax
\fi
```

To prevent wasting two counters in LaTeX 2.09 (because counters with the same name are allocated later by it) we reset the counter that holds the next free counter (`\count10`).

```
\ifx\bye\@undefined
 \advance\count10 by -2\relax
\fi
\ifx\@ifnextchar\@undefined
  \def\@ifnextchar#1#2#3{%
    \let\reserved@d=#1%
    \def\reserved@a{#2}\def\reserved@b{#3}%
    \futurelet\@let@token\@ifnch}
  \def\@ifnch{%
    \ifx\@let@token\@sptoken
      \let\reserved@c\@xifnch
    \else
      \ifx\@let@token\reserved@d
```

```
        \let\reserved@c\reserved@a
      \else
        \let\reserved@c\reserved@b
      \fi
    \fi
    \reserved@c}
  \def\:{\let\@sptoken= } \:  % this makes \@sptoken a space token
  \def\:{\@xifnch} \expandafter\def\: {\futurelet\@let@token\@ifnch}
\fi
\def\@testopt#1#2{%
  \@ifnextchar[{#1}{#1[#2]}}
\def\@protected@testopt#1{%
  \ifx\protect\@typeset@protect
    \expandafter\@testopt
  \else
    \@x@protect#1%
  \fi}
\long\def\@whilenum#1\do #2{\ifnum #1\relax #2\relax\@iwhilenum{#1\relax
    #2\relax}\fi}
\long\def\@iwhilenum#1{\ifnum #1\expandafter\@iwhilenum
        \else\expandafter\@gobble\fi{#1}}
```

Code from `ltoutenc.dtx`, adapted for use in the plain TEX environment.

```
\def\DeclareTextCommand{%
   \@dec@text@cmd\providecommand
}
\def\ProvideTextCommand{%
   \@dec@text@cmd\providecommand
}
\def\DeclareTextSymbol#1#2#3{%
   \@dec@text@cmd\chardef#1{#2}#3\relax
}
\def\@dec@text@cmd#1#2#3{%
   \expandafter\def\expandafter#2%
      \expandafter{%
         \csname#3-cmd\expandafter\endcsname
         \expandafter#2%
         \csname#3\string#2\endcsname
      }%
%   \let\@ifdefinable\@rc@ifdefinable
   \expandafter#1\csname#3\string#2\endcsname
}
\def\@current@cmd#1{%
  \ifx\protect\@typeset@protect\else
      \noexpand#1\expandafter\@gobble
  \fi
}
\def\@changed@cmd#1#2{%
   \ifx\protect\@typeset@protect
      \expandafter\ifx\csname\cf@encoding\string#1\endcsname\relax
```

```
          \expandafter\ifx\csname ?\string#1\endcsname\relax
             \expandafter\def\csname ?\string#1\endcsname{%
                \@changed@x@err{#1}%
             }%
          \fi
          \global\expandafter\let
            \csname\cf@encoding \string#1\expandafter\endcsname
            \csname ?\string#1\endcsname
       \fi
       \csname\cf@encoding\string#1%
         \expandafter\endcsname
   \else
      \noexpand#1%
   \fi
}
\def\@changed@x@err#1{%
    \errhelp{Your command will be ignored, type <return> to proceed}%
    \errmessage{Command \protect#1 undefined in encoding \cf@encoding}}
\def\DeclareTextCommandDefault#1{%
   \DeclareTextCommand#1?%
}
\def\ProvideTextCommandDefault#1{%
   \ProvideTextCommand#1?%
}
\expandafter\let\csname OT1-cmd\endcsname\@current@cmd
\expandafter\let\csname?-cmd\endcsname\@changed@cmd
\def\DeclareTextAccent#1#2#3{%
  \DeclareTextCommand#1{#2}[1]{\accent#3 ##1}
}
\def\DeclareTextCompositeCommand#1#2#3#4{%
   \expandafter\let\expandafter\reserved@a\csname#2\string#1\endcsname
   \edef\reserved@b{\string##1}%
   \edef\reserved@c{%
     \expandafter\@strip@args\meaning\reserved@a:-\@strip@args}%
   \ifx\reserved@b\reserved@c
      \expandafter\expandafter\expandafter\ifx
         \expandafter\@car\reserved@a\relax\relax\@nil
         \@text@composite
      \else
         \edef\reserved@b##1{%
            \def\expandafter\noexpand
               \csname#2\string#1\endcsname####1{%
               \noexpand\@text@composite
                  \expandafter\noexpand\csname#2\string#1\endcsname
                  ####1\noexpand\@empty\noexpand\@text@composite
                  {##1}%
            }%
         }%
         \expandafter\reserved@b\expandafter{\reserved@a{##1}}%
      \fi
```

```
      \expandafter\def\csname\expandafter\string\csname
         #2\endcsname\string#1-\string#3\endcsname{#4}
   \else
     \errhelp{Your command will be ignored, type <return> to proceed}%
     \errmessage{\string\DeclareTextCompositeCommand\space used on
         inappropriate command \protect#1}
   \fi
}
\def\@text@composite#1#2#3\@text@composite{%
   \expandafter\@text@composite@x
      \csname\string#1-\string#2\endcsname
}
\def\@text@composite@x#1#2{%
   \ifx#1\relax
      #2%
   \else
      #1%
   \fi
}
%
\def\@strip@args#1:#2-#3\@strip@args{#2}
\def\DeclareTextComposite#1#2#3#4{%
   \def\reserved@a{\DeclareTextCompositeCommand#1{#2}{#3}}%
   \bgroup
      \lccode`\@=#4%
      \lowercase{%
   \egroup
      \reserved@a @%
   }%
}
%
\def\UseTextSymbol#1#2{%
%   \let\@curr@enc\cf@encoding
%   \@use@text@encoding{#1}%
   #2%
%   \@use@text@encoding\@curr@enc
}
\def\UseTextAccent#1#2#3{%
%   \let\@curr@enc\cf@encoding
%   \@use@text@encoding{#1}%
%   #2{\@use@text@encoding\@curr@enc\selectfont#3}%
%   \@use@text@encoding\@curr@enc
}
\def\@use@text@encoding#1{%
%   \edef\f@encoding{#1}%
%   \xdef\font@name{%
%      \csname\curr@fontshape/\f@size\endcsname
%   }%
%   \pickup@font
%   \font@name
```

```
%   \@@enc@update
}
\def\DeclareTextSymbolDefault#1#2{%
   \DeclareTextCommandDefault#1{\UseTextSymbol{#2}#1}%
}
\def\DeclareTextAccentDefault#1#2{%
   \DeclareTextCommandDefault#1{\UseTextAccent{#2}#1}%
}
\def\cf@encoding{OT1}
```

Currently we only use the LaTeX 2_ε method for accents for those that are known to be made active in *some* language definition file.

```
\DeclareTextAccent{\"}{OT1}{127}
\DeclareTextAccent{\'}{OT1}{19}
\DeclareTextAccent{\^}{OT1}{94}
\DeclareTextAccent{\`}{OT1}{18}
\DeclareTextAccent{\~}{OT1}{126}
```

The following control sequences are used in `babel.def` but are not defined for PLAIN TeX.

```
\DeclareTextSymbol{\textquotedblleft}{OT1}{92}
\DeclareTextSymbol{\textquotedblright}{OT1}{`\"}
\DeclareTextSymbol{\textquoteleft}{OT1}{`\`}
\DeclareTextSymbol{\textquoteright}{OT1}{`\'}
\DeclareTextSymbol{\i}{OT1}{16}
\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 sofisticated font mechanism as LaTeX has, we just `\let` it to `\sevenrm`.

```
\ifx\scriptsize\@undefined
  \let\scriptsize\sevenrm
\fi
⟨/code⟩
```

10 Hooks for XeTeX and LuaTeX

10.1 XeTeX

Unfortunately, the current encoding cannot be retrieved and therefore it is reset always to `utf8`, which seems a sensible default.

```
⟨*xetex⟩
\def\BabelStringsDefault{unicode}
\let\xebbl@stop\relax
\AddBabelHook{xetex}{encodedcommands}{%
  \def\bbl@tempa{#1}%
  \ifx\bbl@tempa\@empty
    \XeTeXinputencoding"bytes"%
  \else
```

```
      \XeTeXinputencoding"#1"%
  \fi
  \def\xebbl@stop{\XeTeXinputencoding"utf8"}}
\AddBabelHook{xetex}{stopcommands}{%
  \xebbl@stop
  \let\xebbl@stop\relax}
```
⟨/xetex⟩

10.2 LuaTeX

This part relies on the `lua` stripts in `luatex-hyphen` by Khaled Hosny, Élie Roux, and Manuel Pégourié-Gonnard. Élie also improved the code below.

⟨*luatex⟩
```
\directlua{%
  require("luatex-hyphen")
  Babel = {}
  function Babel.bytes(line)
    return line:gsub("(.)",
      function (chr) return unicode.utf8.char(string.byte(chr)) end)
  end
  function Babel.begin_process_input()
    if luatexbase and luatexbase.add_to_callback then
      luatexbase.add_to_callback('process_input_buffer',Babel.bytes,'Babel.bytes')
    else
      Babel.callback = callback.find('process_input_buffer')
      callback.register('process_input_buffer',Babel.bytes)
    end
  end
  function Babel.end_process_input ()
    if luatexbase and luatexbase.remove_from_callback then
      luatexbase.remove_from_callback('process_input_buffer','Babel.bytes')
    else
      callback.register('process_input_buffer',Babel.callback)
    end
  end
}
\def\BabelStringsDefault{unicode}
\let\luabbl@stop\relax
\AddBabelHook{luatex}{encodedcommands}{%
  \def\bbl@tempa{utf8}\def\bbl@tempb{#1}%
  \ifx\bbl@tempa\bbl@tempb\else
    \directlua{Babel.begin_process_input()}%
    \def\luabbl@stop{%
      \directlua{Babel.end_process_input()}}%
  \fi}%
\AddBabelHook{luatex}{stopcommands}{%
  \luabbl@stop
  \let\luabbl@stop\relax}
\AddBabelHook{luatex}{patterns}{%
```

```
  \ifx\directlua\relax\else
    \ifcsname lu@texhyphen@loaded@\the\language\endcsname \else
      \global\@namedef{lu@texhyphen@loaded@\the\language}{}%
      \directlua{
        luatexhyphen.loadlanguage('\luatexluaescapestring{\string#1}',
          '\the\language')}%
    \fi
  \fi}
\AddBabelHook{luatex}{adddialect}{%
  \ifx\directlua\relax\else
    \directlua{
      luatexhyphen.adddialect('\luatexluaescapestring{\string#1}',
        '\luatexluaescapestring{\string#2}')
    }%
  \fi}
\AddBabelHook{luatex}{everylanguage}{%
  \directlua{
    processnow = (tex.language == 0) or
      (luatexhyphen.lookupname('\luatexluaescapestring{\string#1}') == nil)}%
  \ifnum0=\directlua{tex.sprint(processnow and "0" or "1")}\relax
    \global\@namedef{lu@texhyphen@loaded@\the\language}{}%
  \fi}
\AddBabelHook{luatex}{loadpatterns}{%
  \ifnum0=\directlua{tex.sprint(processnow and "0" or "1")}\relax
    \input #1\relax
  \fi}
\AddBabelHook{luatex}{loadexceptions}{%
  \ifnum0=\directlua{tex.sprint(processnow and "0" or "1")}\relax
    \input #1\relax
  \fi
  \directlua{processnow = nil}}
⟨/luatex⟩
```

11 Conclusion

A system of document options has been presented that enable the user of LaTeX to adapt the standard document classes of LaTeX to the language he or she prefers to use. These options offer the possibility of switching between languages in one document. The basic interface consists of using one option, which is the same for *all* standard document classes.

In some cases the language definition files provide macros that can be useful to plain TeX users as well as to LaTeX users. The babel system has been implemented so that it can be used by both groups of users.

12 Acknowledgements

I would like to thank all who volunteered as β-testers for their time. I would like to mention Julio Sanchez who supplied the option file for the Spanish language and Maurizio Codogno who supplied the option file for the Italian language. 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] Donald E. Knuth, *The TEXbook*, Addison-Wesley, 1986.

[2] Leslie Lamport, *LATEX, A document preparation System*, Addison-Wesley, 1986.

[3] K.F. Treebus. *Tekstwijzer, een gids voor het grafisch verwerken van tekst.* SDU Uitgeverij ('s-Gravenhage, 1988). A Dutch book on layout design and typography.

[4] Hubert Partl, *German TEX*, *TUGboat* 9 (1988) #1, p. 70–72.

[5] Leslie Lamport, in: TEXhax Digest, Volume 89, #13, 17 February 1989.

[6] Johannes Braams, Victor Eijkhout and Nico Poppelier, *The development of national LATEX styles*, *TUGboat* 10 (1989) #3, p. 401–406.

[7] Joachim Schrod, *International LATEX is ready to use*, *TUGboat* 11 (1990) #1, p. 87–90.