

The `multilang` package*

Richard Grewe
`r-g+tex@posteo.net`

November 2, 2018

Abstract

Maintaining a L^AT_EX document with translations for multiple languages can be cumbersome and error-prone. The `multilang` package provides a set of macros for defining macros and environments as wrappers around existing macros and environments. These wrappers allow one to clearly specify multiple translations for the arguments to the wrapped macros and environments while only the translation of the document's language is actually shown. Choosing a translation then is as simple as choosing the document's language via `babel` or `polyglossia`.

1 Introduction

The main goal of the `multilang` package is to facilitate the definition of macros and environments with which documents can be provisioned in multiple languages. To be more concrete, `multilang` facilitates

1. the specification of content translations:
 - Arguments to macros are specified by their name such that multiple translations do not clutter up the L^AT_EX code that easily.
 - If you forget to specify a mandatory argument in a language, `multilang` shows an error when you compile for that language.

2. the maintenance of translations:

Translations of content that is passed as an argument to a macro can be kept closely together such that you can keep track of which units belong together and can keep consistency among translations across content changes.

3. the selection of a document language:

You simply specify the language with `babel` or `polyglossia` and `multilang` selects the translation you provided for the language.

*This document corresponds to `multilang` v0.9b, dated 2018/11/02. The package is available online at <http://www.ctan.org/pkg/multilang> and <https://github.com/Ri-Ga/multilang>.

The motivating example that lead to the development of this package are CVs (curricula vitae). Suppose you want to maintain a CV document that contains the union of all your relevant personal data, education, achievements, etc. You update the CV from time to time with new achievements. Since you want to prepare for the possibility that you might apply nationally as well as internationally, you maintain the CV document in multiple languages (e.g., your mother language and English). When you use the CV in a concrete job application, you (1) filter out all details that are irrelevant for the position that you apply for and (2) select a suitable language for the employer.

This package provides a set of basic macros for the definition of multilingual macros and multilingual environments. The following example illustrates such a macro and how it could be used.

```
% ...
\usepackage[german,english]{babel}
\usepackage[languages={english,german}]
{multilang}
% ...
\begin{document}
\Sec{
  title/english={Foobar},
  title/german ={Dingsda}
}
...
\end{document}
```

1 Foobar

...

[Section 2](#) describes how a macro such as `\Sec` in the above example can be defined with `multilang`. The section also describes how analogous environments can be defined. [Section 3](#) describes extensible argument types. [Section 4](#) describes two extension packages, `multilang-sect` and `multilang-tags` for multilingual sectioning and, respectively, for filtering multilingual macros and environments.

2 Usage

2.1 Package Options

`languages` The `multilang` package has a single option: `languages`. This option expects a comma-separated list of language names. The ordering of the list does not make a difference. Through this option, one sets the languages for which translations can be provided. By default, the list is empty such that no translations can be specified. The example in [Section 1](#) demonstrates how this option can be used.

2.2 Multilingual Macros

```
\NewMultilangCmd{\langle command \rangle}{\langle options \rangle}
```

A multilingual macro can be defined via the `\NewMultilangCmd` macro. The macro defines `\langle command \rangle` as a macro that accepts a single argument. That is,

the signature of $\langle command \rangle$ is $\langle command \rangle\{\langle kvarg \rangle\}$. The argument, $\langle kvarg \rangle$, is a comma-separated key-value list. The $\langle options \rangle$ argument takes the form of a comma-separated key-value list and specifies what $\langle command \rangle$ does, including how $\langle kvarg \rangle$ is interpreted. The following keys are defined for $\langle options \rangle$:

- command:** This key is to be used in the form “ $\text{command}=\langle c \rangle$ ”, where $\langle c \rangle$ is an already existing command. The key specifies that $\langle command \rangle$ is defined to invoke $\langle c \rangle$ with some arguments. Example: see [Section 2.2.1](#).
- margs:** This key is to be used in the form “ $\text{margs}=\{\langle csvlist \rangle\}$ ”, where $\langle csvlist \rangle$ is a comma-separated list of names. The key specifies names for the mandatory arguments of $\langle c \rangle$, in the order specified. If $\langle c \rangle$ is a macro that takes n mandatory arguments, then $\langle csvlist \rangle$ should be a list of n names. If $\langle arg-i \rangle$ is the i -th entry in $\langle csvlist \rangle$, then the i -th mandatory argument to $\langle c \rangle$ can be specified as $\langle v \rangle$ in the invocation of $\langle command \rangle$ as follows: “ $\langle command \rangle\{\dots,\langle arg-i \rangle=\langle v \rangle,\dots\}$ ”. Example: see [Section 2.2.1](#).
- oargs:** This key is to be used in the form “ $\text{oargs}=\{\langle csvlist \rangle\}$ ”, where $\langle csvlist \rangle$ is a comma-separated list of names. This key is analogous to **margs**, but for optional arguments to $\langle c \rangle$. Example: see [Section 2.2.2](#).
- starred:** This key is to be used simply as “ $\text{starred}=\langle bool \rangle$ ”, where $\langle bool \rangle$ must be **true** or **false**, or just as “**starred**”, which is equivalent to “**starred=true**”. If $\langle bool \rangle$ is **true**, then “ $\langle command \rangle*\{\dots\}$ ” can be used to invoke “ $\langle c \rangle*\dots$ ”. Example: see [Section 2.2.3](#).
- disablable:** This key is to be used simply as “ $\text{disablable}=\langle bool \rangle$ ”, where $\langle bool \rangle$ must be **true** or **false**, or just as “**disablable**”, which is equivalent to “**disablable=true**”. If $\langle bool \rangle$ is **true**, then the invocation of $\langle c \rangle$ in $\langle command \rangle$ can be suppressed via “ $\langle command \rangle\{\dots,\text{disabled},\dots\}$ ”. Example: see [Section 2.2.4](#).
- defaults:** This key is to be used as “ $\text{defaults}=\{\dots,\langle arg \rangle=\langle value \rangle,\dots\}$ ”. Each key $\langle arg \rangle$ should be the name of an optional or mandatory argument and $\langle value \rangle$ should be the intended default value for this argument. Example: see [Section 2.2.5](#).
- alias/...:** This key is to be used as “ $\text{alias}/\langle name \rangle=\{\langle arglist \rangle\}$ ”, where $\langle name \rangle$ specifies the name of the alias and $\langle arglist \rangle$ is a possibly empty comma-separated list of argument names (optional or mandatory arguments allowed). Example: see [Section 2.2.6](#).

The remainder of [Section 2.2](#) provides examples based on the `\section` macro.

2.2.1 Basic Usage

The following example shows how the `\Section` macro used in the example of [Section 1](#) can be defined. We leave out the `babel` code in this section for more concise example code.

```
\NewMultilangCmd{\Sec}{  
    command=\section, margs=title}  
\Sec{  
    title/english={Foobar},  
    title/german ={Dingsda}  
}
```

1 Foobar

Language-independent arguments. Sometimes, arguments to macros need no translation. For instance, if an argument is a technical term or a name, it can remain in the original language. To avoid redundancy, in such instances the language part of a mandatory or optional argument can be omitted to specify the argument for all languages.

```
\Sec{title=multilang}
```

1 multilang

Forced foreign-language arguments. A particular instance of a language-independent argument is the case in which the argument shall explicitly be typeset in a particular language. For instance, the argument might use macros that internally determine the display based on the selected language. The following example shows the case of forcing German display of the \enquote macro.

```
\usepackage[autostyle=true]{csquotes}  
%...  
\Sec{title/german!=\enquote{multilang}}
```

1 „multilang“

2.2.2 Optional Arguments

Macros like \section don't just have a mandatory argument (the section title) but also an optional argument (a short title for the table of contents). We can make this optional argument accessible via the oargs option as follows. The displayed result does not show any difference as we don't have the table of contents here.

```
\NewMultilangCmd{\Sec}{  
    command=\section, margs=title, oargs=short}  
\Sec{  
    title/english={Foobar},  
    title/german ={Dingsda},  
    short/english={F},  
    short/german ={D},  
}
```

1 Foobar

2.2.3 Starred Macros

Some macros can be altered in their behavior with a star (“*”) after the macro. The \section command is an example of such a macro: \section* suppresses the display of the section number. The star can be transferred to the command defined via \NewMultilangCmd as the following example illustrates. Note the exclamation mark after “german”.

```
\NewMultilangCmd{\Sec}{starred,  
    command=\section, margs=title, oargs=short}  
\Sec{title=Foo}  
\Sec*{title=Bar}
```

1 Foo
Bar

2.2.4 Disabling Display

When a macro is defined with the `disablable` option, it can be passed the `disabled` argument which disables the display of the macro. One might consider this a nicer way to disable content than commenting out the macro.

```
\NewMultilangCmd{\Sec}{disablable,  
    command=\section, margs=title, oargs=short}  
\Sec{title=Foo}  
\Sec{title=Bar,disabled}
```

1 Foo

2.2.5 Default Arguments

When a mandatory or optional argument shall by default assume a particular value if no value is specified for the argument in the argument to `(command)`, this can be specified via the `defaults` key as the following example demonstrates.

```
\NewMultilangCmd{\Sec}{  
    command=\section, margs=title,  
    defaults={title={??}}}  
\Sec{}
```

1 ???

Defaults can be particularly useful for mandatory arguments of commands that take multiple arguments and that can often be left empty. Essentially, argument defaults turn mandatory arguments of macros to optional arguments.

2.2.6 Argument Aliases

Aliases allow one to specify argument names of three different kinds:

direct aliases: A direct alias declares an argument name that can then be used as a substitute for some mandatory or optional argument. The `heading` argument in the following example shows how direct aliases can be declared.

combiner aliases: A combiner alias declares an argument name that acts as a substitute for a sequence of mandatory or optional arguments. Such aliases particularly help maintaining concise L^AT_EX source code when argument values are rather short, as `both` in the following example shows.

comment aliases: A comment alias declares an argument name that represents no mandatory or optional argument. That is, a value specified for a comment alias is not used as an argument to the `command` and can, hence, be used for capturing comments or values for future use.

```
\NewMultilangCmd{\Sec}{  
    command=\section, margs=title, oargs=short,  
    alias/heading=title,  
    alias/both={title,short},  
    alias/remark}  
\Sec{  
    both/english={Foobar}{Foo},  
    both/german ={Dingsda}{Dings}}  
\Sec{  
    heading=Baz,  
    remark={select heading+translate}}
```

1 Foobar

2 Baz

2.3 Multilingual Environments

```
\NewMultilangEnv{\langle environment \rangle}{\langle options \rangle}
```

The usage of the `\NewMultilangEnv` macro is analogous to the usage of the macro `\NewMultilangCmd`, except for the following differences:

- The first argument, `\langle environment \rangle`, expects the name of an environment that shall be defined.
- In the `\langle options \rangle`, the `environment` key substitutes the `command` key and expects an environment name.
- The `starred` key is not available. Following standard L^AT_EX practices, if you want to define a starred environment, simply use the starred name for `\langle environment \rangle`.

Due to the similarity to `\NewMultilangCmd`, we don't provide separate examples for all the individual features. Continuing the line of examples started before, we again use an example about sections – just now with an environment for section boxes, as provided by the `sectionbox` package.

```
\usepackage{sectionbox}
\NewMultilangEnv{SecBox}{
    environment=sectionbox,
    disableable, margs=title, oargs=width}

\begin{SecBox}{title=Foo}
    content
\end{SecBox}

\begin{SecBox}{disabled,title=Bar}
    disabled content
\end{SecBox}
```



1 Foo
content

3 Extensible Argument Types

By default, an argument `arg` in a multilingual macro can be specified in three ways: just “`arg=...`”, “`arg/⟨language⟩=...`”, or “`arg/⟨language⟩!=...`”. The `multilang` package enables one to declare further so called ‘types’. These types can be used in place of `⟨language⟩` in the argument syntax. They can be used for uniformly enabling a special formatting if an argument is of a particular type (rather than just being text).

```
\NewMultilangType[⟨argcount⟩]{⟨typename⟩}{⟨format⟩}
```

The `\NewMultilangType` macro declares the `\langle typename \rangle` type. Values passed to arguments of this type must consist of `\langle argcount \rangle` arguments (default: 1). The value is then formatted with `\langle format \rangle` when displayed. The `\langle format \rangle` can (and should) contain positional parameters such as “`#1`” for the first argument. The remainder of this section demonstrates the use of the macro by examples.

3.1 Dates via `datetime2`

The `datetime2` package supports regional (language-specific) formatting of dates. We can build on this feature such that we don't have to provide translations of dates. To keep the display smaller, we here use `\textbf{t}` rather than `\section`.

```
\usepackage[useregional]{datetime2}
\NewMultilangType{date}{\DTMdate{\#1}}
\NewMultilangType[2]{daterange}
  {\DTMdate{\#1}--\DTMdate{\#2}}
\NewMultilangCmd{\Bold}
  {margs=title, command=\textbf{t}}
Date: \Bold{title/date={2017-08-01}}\\
Range: \Bold{title/daterange=
  {2017-01-01}{2017-08-01}}
```

Date: August 1, 2017
Range: January 1, 2017–August 1, 2017

3.2 Nesting Multilingual Macros

Multilingual macros can be used in arguments to multilingual macros. Coming back to the original motivation for developing `multilang-` CVs – you might want a translated two-column layout in which the right column might be filled with translated list items. The following example shows how this can be realized.

```
\usepackage{enumitem}% for "nosep"
\newcommand\entry[2]{%
  \begin{tabular}{p{1cm}p{4cm}}#1&#2
  \end{tabular}}
\NewMultilangType{list}{%
  \begin{minipage}[t]{4cm}
  \begin{itemize}[nosep]#1
  \end{itemize}\end{minipage}}
\NewMultilangCmd{\Entry}
  {margs={head,text}, command=\entry}
\NewMultilangCmd{\Item}
  {margs=name, command=\item}

\Entry{head=2017,
      text/list={
        \Item{name/english=foobar,
              name/german =Dingsda}
        \Item{name=multilang}}}
```

2017 • foobar
• multilang

4 Extension Packages

The `multilang` package comes bundled with a few generic packages that build on `multilang`. These packages are described below.

4.1 Sectioning Environments

Sectioning environments are provided by the `multilang-sect` package. That package defines, for each of L^AT_EX's sectioning macros (`\section`, ..., `\subparagraph`) an environment and a starred environment.

```
\begin{Section*}{<data>}
```

`\end{Section*}` This environment shows a section. It has a single, mandatory argument, named `title`. It is a disableable environment, i.e., the argument `disabled` can be used in `<data>` to disable the display of the whole section. This environment acts as a proxy for the `\section` macro as it is used by `multilang` (i.e., without optional argument and without the star).

```
\begin{SubSection*}{<data>}
```

`\end{SubSection*}` This environment is analogous to the `Section` environment, just for sub-sections.

```
\begin{SubSubSection*}{<data>}
```

`\end{SubSubSection*}` This environment is analogous to the `Section` environment, just for sub-sub-sections.

```
\begin{Paragraph*}{<data>}
```

`\end{Paragraph*}` This environment is analogous to the `Section` environment, just for paragraphs.

```
\begin{SubParagraph*}{<data>}
```

`\end{SubParagraph*}` This environment is analogous to the `Section` environment, just for sub-paragraphs. Examples:

```
\usepackage{multilang-sect}

\begin{Section}{
    title/english = Usage,
    title/german  = Benutzung,
}
(section content)
\begin{SubSection*}{
    title/english = Package Options,
    title/german  = Paketoptionen,
}
(subsection content)
\end{SubSection*}
\end{Section}
```

1 Usage

(section content)

Package Options

(subsection content)

4.2 Tags

Tags are an alternative to individually disabling macros or environments. They are provided by the `multilang-tags` package. A tag is just a word and sets of tags can be assigned to individual usages of multilingual macros and environments. As long as no tag filter policy is setup, specifying tags does not influence what is displayed.

```
\SetTagFilter[<default>]{<policy>}
```

The `\SetTagFilter[⟨default⟩]{⟨policy⟩}` sets up a tag filter policy. The `⟨policy⟩` is a comma-separated list of `accept/deny` rules. The `⟨default⟩` argument is either `accept` (the default) or `deny` and specifies the default policy. The following toy example demonstrates tag filtering:

```
\usepackage{multilang-tags}
\NewMultilangCmd{\Item}{\disablable,
  command=\item,oargs=dd,margs=dt,
  alias/both={dd,dt}}
\SetTagFilter{accept={A}, deny={D}}
\begin{description}
\Item{tags=A, both={1}}{tagged A}
\Item{tags={A,D}, both={2}}{tagged A,D}
\Item{tags=D, both={3}}{tagged D}
\Item{tags={D,X}, both={4}}{tagged D,X}
\Item{tags!=D, both={5}}{tagged !D}
\Item{tags=X, both={6}}{tagged X}
\end{description}
```

1	tagged A
2	tagged A,D
5	tagged !D
6	tagged X

When a multilingual macro, such as `\BasicEntry` in the above example, is used, whether the macro content is displayed is determined as follows:

- If no `tags` are specified or no tag policy is setup, then the macro content is displayed.
- Otherwise, the rules of the tag policy are processed in sequential order until the specified `tags` match a rule. A `tags` list matches a rule if at least one tag in the `tags` occurs in the rule (with or without “!” prefix). Let t be the last tag in `tags` that occurs in the rule.
 - The macro content is displayed if the rule is an `accept` rule and t is not prefixed with “!”, or if the rule is a `deny` rule and t is prefixed with “!”.
 - Otherwise the display of the macro content is disabled.
- If the specified `tags` match no rule in the `⟨policy⟩`, then the macro content is displayed if and only if the `⟨default⟩` is `accept`.

Rather than directly setting up a filter policy, one can also use the following macros to first define filter policies and then select one for use.

`\DefineTagFilter{⟨name⟩}{⟨default⟩}{⟨policy⟩}`

This macro defines a tag filter policy with name `⟨name⟩` to represent the given `⟨policy⟩` and `⟨default⟩`.

`\UseTagFilter{⟨name⟩}`

This macro uses the tag filter policy with name `⟨name⟩`.

Further examples:

```

\DefineTagFilter{Show}{accept}{}
\DefineTagFilter{Hide}{deny}{}
\DefineTagFilter{OnlyA}{accept}{accept=A,
                           deny={D,X}}
\begin{description}
\UseTagFilter{Hide}
\Item{both={1}}{no tag}
\Item{tags=D, both={2}}{tagged D}
\UseTagFilter{OnlyA}
\Item{tags={D,!X}, both={3}}{tagged D,!X}
\Item{tags={!X,D}, both={4}}{tagged !X,D}
\end{description}

```

1 no tag
3 tagged D,!X

5 Questions and Answers

Can't I achieve the same thing simpler? To some extent, you can. A variety of ad-hoc solutions to managing translations in L^AT_EX documents exist.¹ An obvious approach is the following:

```

\newcommand\inEnglish[1]{#1}
\newcommand\inGerman[1]{}

\inEnglish{\section{Foobar}}
\inGerman{\section{Dingsda}}
% or
\section{\inEnglish{Foobar}}
\inGerman{Dingsda}

```

1 **Foobar**
2 **Foobar**

That is, for each translation language you define a macro with one argument; the macro for the “selected” language expands to the argument while all other macros have an empty expansion. An obvious advantage of this approach over `multilang` is that it does not require learning how to use `multilang`. However, in my opinion, the approach has the following disadvantages:

- Both the first variant and the second variant in the example make the code more difficult to read due to the nesting of macros and the curly braces.
- The first variant even requires the `\section` macro to be repeated for each language.
- The second variant easily gets chaotic if instead of `\section` a macro with several arguments is used.

Solutions with conditionals (e.g., `\ifEnglish ... \else ... \fi`) share the disadvantages of above approach (except the curly braces) and additionally have an inherent asymmetry that becomes particularly apparent if more than two languages are involved. Similar arguments apply to other ad-hoc solutions I have seen. That is, I find documents based on such approaches cumbersome to maintain and, hence, requiring more careful checks for ensuring consistency.

¹see, e.g., <https://tex.stackexchange.com/q/5076>

Can I use `multilang` with `polyglossia` instead of `babel`? Yes, you can use either one for selecting the language of your document.

Can I switch the language mid-document? You can switch the language mid-document (e.g., using Babel’s `selectlanguage` environment), but this does not effect what the multilingual macros or environments defined via `multilang`. The language that is displayed by a macro or environment is determined at the time of loading `multilang`. Future versions of `multilang` might add support for switching languages mid-document, though.

Are language dialects supported? No, currently they are not supported.

Can I store the “result” of a multilingual macro in another macro? No. You can store the macro itself (e.g., via `\newcommand`), but storing the result of the multilingual macro in a macro (e.g., via `\edef`) is not possible, as the multilingual macros are not expandable.

6 Related Packages

I’m not aware of any L^AT_EX packages that pursue similar goals or provide similar functionality. CTAN provides a list of many packages for supporting more than one language.² In the following, we compare against some of these packages.

babel, polyglossia: These package provide support for selecting a document language and switching the document language within a document. The selected language is then used for hyphenation and other layouting aspects. Providing multiple translations of pieces of content is not particularly facilitated by the two packages.

The `multilang` package builds on `babel` or `polyglossia` for determining the selected language and for forced foreign-language formatting (as illustrated in [Section 2.2.1](#)).

translations: This package aims primarily at package authors, providing them an easy interface for providing translations of package-specific terms. Essentially, one declares translations for terms up-front and then later can use these translations.

The separation of translations and the use of terms is beneficial for the maintenance of packages and documents in which individual terms occur multiple times. However, I assume that this separation would make it harder to maintain documents in which most translated units occur only once and at a particular location in the document.

One could combine the virtues of `translations` and `multilang` as follows:

²<https://www.ctan.org/topic/multilingual>

```
% in preamble
\usepackage{translations}
\DeclareTranslation{english}{foo}{Foobar}
\DeclareTranslation{german}{foo}{Dingsda}
\NewMultilangType{translate}
    {\GetTranslation{\#1}}
\NewMultilangCmd{\Sec}{%
    margs=title, command=\section}

% in document
\Sec{title/translate=foo}
```

1 Foobar

xt_capt: This package is similar to the `translations` package, even though the package's documentation does not explicitly refer to package authors as the target users. That is, it provides commands for declaring and using translations of terms. The main difference to `translations`, as far as I understand, is that the user interface of `translations` is larger and supports language dialects. Comparing with `multilang`, the same remarks as for `translations` apply.

7 Implementation

7.1 Dependencies

We use `pgfkeys` for the options parsing, both for the macros defined by `multilang` and for the macro defined via the macros in `multilang`. We use `etoolbox` to simplify the internal code.

```
1 \RequirePackage{pgfkeys,pgfopts}
2 \RequirePackage{etoolbox}
```

We use the `environ` package for scanning and later forgetting the body of disabled multilingual environments.

```
3 \RequirePackage{environ}
```

7.2 Package Options

The “languages” option selects the languages that `multilang` knows about.

```
4 \newcommand{\multilang@@langs}{}
5 \pgfqkeys{/multilang/pkg}{%
6   languages/.code={\forcsvlist{\listadd{\multilang@@langs}{#1}},%
7 }
8 \ProcessPgfOptions{/multilang/pkg}}
```

7.3 Main Macros

`\NewMultilangCmd` The `\NewMultilangCmd{\langle command \rangle}{\langle options \rangle}` macro defines `\langle command \rangle` to be a single-argument macro. The argument to `\langle command \rangle` specifies, in a key-value list style, the mandatory and optional arguments that are passed to a command specified in `\langle options \rangle`.

```
9 \newcommand{\NewMultilangCmd}[2]{%
10   \bgroup
```

The following line processes the `\langle options \rangle` given and, as its result, defines the macros `\multilang@@actuals`, `\multilang@@checks`, and `\multilang@@keys`.

```
11 \multilang@processargs{\#1}{/multilang/newcommand}{defaults={},#2}%
```

To handle starred macros, we store the actual macro code into an auxiliary macro and define `\langle command \rangle` to be an interface to the auxiliary macro. The next line stores the name of the internal macro.

```
12 \expandafter\def\expandafter\multilang@@intcmd\expandafter{%
13   \csname multilang@intcmd@\expandafter\@gobble\string#\endcsname}%

```

Finally, we create the `\langle command \rangle`. The `\edef` starting with the `\egroup` shall expand the three macros constructed above but nothing else such that none of the `\pgfqkeys` result for #1 spills outside the `\NewMultilangCmd`.

```
14 \edef\do{\egroup
15   \expandonce{\multilang@@keys}%
16   \ifbool{multilang@@starred}{%
```

The following handles the case of a **starred** macro. We define the macro that scans for the star (*command*) as well as the internal macro that does the actual work.

```

17      \unexpanded{\newcommand#1}{%
18          \noexpand\@ifstar
19              {\expandonce{\multilang@@intcmd}{*}}%
20              {\expandonce{\multilang@@intcmd}{}{}}%
21      }{%

```

The following handles the case of a non-**starred** macro. Here we make *command* directly resort to the internal macro.

```

22      \unexpanded{\newcommand#1}{\expandonce{\multilang@@intcmd}{}{}}%
23      }{%

```

The remainder of the macro code defines the internal macro, with signature $\langle \multilang@@keys \rangle \{ \langle decoration \rangle \} \{ \langle kvar \rangle \}$. The *decoration* can assume any symbols that shall directly be put after the command encapsulated by `command`. A particular use for the *decoration* argument is the “*” symbol for a **starred** macro.

```

24      \noexpand\newcommand{\expandonce{\multilang@@intcmd}}[2]{%

```

First, *command* parses its argument using `pgfkeys`.

```

25      \bgroup
26      \noexpand\boolfalse{multilang@cmd@@disabled}%
27      \noexpand\pgfqkeys{\multilang@keyof{#1}}{%
28          \expandonce{\multilang@@defaults},####2}%
29      \noexpand\ifbool{multilang@cmd@@disabled}%

```

If the macro is disabled, simply use an empty invocation `\multilang@@invok`.

```

30      {\unexpanded{\def\multilang@@invok{}}}{%

```

Otherwise, first check the arguments and afterwards define `\multilang@@invok` to contain *command*, *decoration* (in `####1`), and the actual arguments (in `\multilang@@actuals`).

```

31      \expandonce{\multilang@@checks}%
32      \unexpanded{\edef\multilang@@invok}{%
33          \noexpand\unexpanded{\expandonce{\multilang@@cmd}}####1%
34          \expandonce{\multilang@@actuals}}{%

```

Finally, *command* invokes the command specified via the `command` key. The invocation happens outside the local group, just for the case that makes a difference with the `command`.

```

35      \unexpanded{\expandafter\egroup\multilang@@invok}%
36      }{%
37  }\do{%

```

`\NewMultilangEnv` The `\NewMultilangEnv{\langle environment \rangle}{\langle options \rangle}` macro defines *environment* to be a single-argument environment. The argument to *environment* specifies, in a key-value list style, the mandatory and optional arguments that are passed to an environment specified in *options*.

```

38 \newcommand{\NewMultilangEnv}[2]{%
39   \bgroup
```

The following line processes the *<options>* given and, as its result, defines the macros `\multilang@@actuals`, `\multilang@@checks`, and `\multilang@@keys`.

```

40   \multilang@processargs{#1}{\multilang/newenvir}{defaults={},#2}%

```

Finally, we create the *<environment>*. The `\edef` starting with the `\egroup` shall expand the three macros constructed above but nothing else such that none of the `\pgfqkeys` result for #1 spills outside the `\NewMultilangEnv`. We also pay attention that as few as possible internal macros spill into the environment or even the code that begins the environment.

```

41   \edef\do{\egroup
42     \expandonce{\multilang@@keys}%
43     \unexpanded{\newenvironment{#1}}[1]{%
```

First, *<command>* parses its argument using `pgfkeys`.

```

44   \bgroup
45     \noexpand\boolfalse{\multilang@cmd@disabled}%
46     \noexpand\pgfqkeys{\multilang@keyof{#1}}{####1}%
47     \noexpand\ifbool{\multilang@cmd@disabled}{}%
```

If the body shall be disabled, then we don't perform checks on the arguments, don't open `environment` but rather collect the body of the environment and finally also ignore the code for closing `environment` (via `\multilang@noend`). Through this trick, we avoid defining an additional macro for switching in the end-block of the environment.

```

48   \unexpanded{%
49     \def\multilang@invok{\Collect@Body{\multilang@noend}}}%
```

First check the arguments and afterwards define `\multilang@invok` to contain the opening code for the environment, including the actual arguments (in `\multilang@@actuals`).

```

50   \expandonce{\multilang@@checks}%
51   \unexpanded{\edef\multilang@invok{%
52     \noexpand\noexpand\noexpand\begin{\multilang@env}%
53     \expandonce{\multilang@@actuals}}}%
```

Finally, *<environment>* begins the environment specified via the `environment` key. This happens outside the local group, such that the internal macros set via `\pgfqkeys` are not visible anymore when the environment is started.

```

54   \unexpanded{\expandafter\egroup\multilang@invok}%
55 }{%
```

The following implements the closing of the environment.

```

56   \noexpand\end{\multilang@env}%
57 }%
58 }\do{
```

7.3.1 Option Keys

We first setup the shared keys for the $\langle options \rangle$ argument of `\NewMultilangCmd` and `\NewMultilangEnv`.

```

59 \pgfqkeys{/multilang/cmd-or-env}{
60   margs/.store in={\multilang@@margs},
61   oargs/.store in={\multilang@@oargs},
62   alias/.is family,
63   alias/.unknown/.code={%
64     \listead{\multilang@aliases}{\pgfkeyscurrentname}%
65     \csdef{multilang@@alias@\pgfkeyscurrentname}{#1}%
66   defaults/.store in={\multilang@@defaults},
67   disablable/.is if={multilang@@disablable},
68 }
69 \newbool{multilang@@disablable}
70 \newbool{multilang@cmd@@disabled}
```

Next, we setup the specific keys for `\NewMultilangCmd`.

```

71 \pgfqkeys{/multilang/newcommand}{
72   .search also={/multilang/cmd-or-env},
73   command/.store in={\multilang@@cmd},
74   starred/.is if={multilang@@starred},
75   alias/.search also={/multilang/cmd-or-env},
76 }
77 \newbool{multilang@@starred}
```

Finally, the specific keys for `\NewMultilangEnv`.

```

78 \pgfqkeys{/multilang/newenvir}{
79   .search also={/multilang/cmd-or-env},
80   environment/.store in={\multilang@@env},
81   alias/.search also={/multilang/cmd-or-env},
82 }
```

7.4 Registration of Datatypes

- `\NewMultilangType` The `\NewMultilangType[$\langle argcount \rangle$]{ $\langle typename \rangle$ }{}{ $\langle format \rangle$ }` macro registers the name $\langle typename \rangle$ as a type that can be used for specifying multilang arguments. The type has $\langle argcount \rangle$ arguments (default: 1) and is formatted according to code $\langle format \rangle$. Note that the definition of types is group-local. That is, if `\NewMultilangType` is used within a group, $\langle typename \rangle$ is only available inside that group.

```
83 \newcommand{\NewMultilangType}[3][1]{%
```

We first record the new type's name (in `\multilang@@types`) and store both $\langle argcount \rangle$ and $\langle format \rangle$ in macros.

```

84 \listadd{\multilang@@types}{#2}%
85 \expandafter\newcommand\cscname multilang@typecmd##2\endcsname[#1]{#3}%
86 \csdef{multilang@typeargc##2}{#1}%
```

Finally, we also store the invocation of the `\format` code macro. This is a bit cumbersome, as for all possible argument counts we provide the respective number of arguments. I did not yet find a more elegant way to achieve that the `style n args` code in `\multilang@regfieldtype` invokes `\format` properly with `\argcount` arguments. `\argcount` and `\format` in macros.

```
87   \ifcase#1\relax
88     \csdef{multilang@runcmd@#2}{\csuse{multilang@@typecmd@#2}}%
```

For one argument, we check a special case: `\format` is the identity function. In this case, we directly expand to the argument itself, i.e., we unfold `\format`. We do this such that emptiness of optional arguments can be checked by a command or environment without having to expand the `\format` (which might not work out if `\format` is not expandable). For instance, KOMA's `\section` macro hides the TOC entry for `\section[]{...}` but not for `\section[\X]{...}` even if `\X` expands to an empty result.

```
89   \or\ifcsequal{multilang@@typecmd@#2}{@firstofone}%
90     {\csdef{multilang@runcmd@#2}{####1}}%
91     {\csdef{multilang@runcmd@#2}{\csuse{multilang@@typecmd@#2}}%
92      {####1}}%
93   \or\csdef{multilang@runcmd@#2}{\csuse{multilang@@typecmd@#2}}%
94     {####1}{####2}}%
95   \or\csdef{multilang@runcmd@#2}{\csuse{multilang@@typecmd@#2}}%
96     {####1}{####2}{####3}}%
97   \or\csdef{multilang@runcmd@#2}{\csuse{multilang@@typecmd@#2}}%
98     {####1}{####2}{####3}{####4}}%
99   \or\csdef{multilang@runcmd@#2}{\csuse{multilang@@typecmd@#2}}%
100    {####1}{####2}{####3}{####4}{####5}}%
101  \or\csdef{multilang@runcmd@#2}{\csuse{multilang@@typecmd@#2}}%
102    {####1}{####2}{####3}{####4}{####5}{####6}}%
103  \or\csdef{multilang@runcmd@#2}{\csuse{multilang@@typecmd@#2}}%
104    {####1}{####2}{####3}{####4}{####5}{####6}{####7}}%
105  \or\csdef{multilang@runcmd@#2}{\csuse{multilang@@typecmd@#2}}%
106    {####1}{####2}{####3}{####4}{####5}{####6}{####7}{####8}}%
107  \or\csdef{multilang@runcmd@#2}{\csuse{multilang@@typecmd@#2}}%
108    {####1}{####2}{####3}{####4}{####5}{####6}{####7}{####8}{####9}}%
109 \else\multilang@error{Argument count expected to be between 0 and 9, %
110                           but is '#1'}\fi}
```

`\multilang@types` The `\multilang@types` macro collects and holds an etoolbox list of datatypes defined via `\NewMultilangType`.

```
111 \newcommand\multilang@types{}
```

`\NewMultilangType@code` The `\NewMultilangType@code[<argcount>]{<typename>}{<format>}` macro is an internal counterpart to `\NewMultilangType` with which not not the `.style` but the `.code` property of the `<typename>` key is defined. This is indicated by defining the `\multilang@codetype@<typename>` macro here and checking whether this macro is defined in `\multilang@regfieldtype`.

```
112 \newcommand\NewMultilangType@code[3][1]{%
```

```

113 \csdef{multilang@@codetype@#2}{true}%
114 \NewMultilangType[#1]{#2}{#3}

\multilang@regfield The \multilang@regfield{\langle cmd-or-env\rangle}{\langle fieldname\rangle} macro registers the respective pgfkeys keys for \langle fieldname\rangle for all registered datatypes.
115 \newcommand\multilang@regfield[2]{%
116   \pgfqkeys{\multilang@keyof{#1}}{%
117     #2/.code={\csdef{multilang@@val@#2}{##1}}%
118   }%
119   \forlistloop{\multilang@regfieldtype{#1}{#2}}{\multilang@@types}

```

\multilang@regfieldtype The \multilang@regfieldtype{\langle cmd-or-env\rangle}{\langle fieldname\rangle}{\langle typename\rangle} macro registers the pgfkeys key for \langle typename\rangle of \langle fieldname\rangle.

```

120 \bgroup
121 \newcommand\multilang@regfieldtype[3]{%
122 }
```

In the following, we check whether the number of arguments for the \langle typename\rangle macro is 1, because for some reason style n args seems not to work as we want it to work if $n = 1$.

```

123 \ifnumequal{\csuse{multilang@@typeargc@#3}}{1}{%
124   \ifcsdef{multilang@@codetype@#3}{%
125     \edef\do{\egroup\noexpand\pgfqkeys{\multilang@keyof{#1}}{%
126       #2/#3/.code={\csexpandonce{multilang@@runcmd@#3}}%
127     }}%
128   }{%
129     \edef\do{\egroup\noexpand\pgfqkeys{\multilang@keyof{#1}}{%
130       #2/#3/.style={#2=\csexpandonce{multilang@@runcmd@#3}}%
131     }}%
132   }{%
133     \ifcsdef{multilang@@codetype@#3}{%
134       \edef\do{\egroup\noexpand\pgfqkeys{\multilang@keyof{#1}}{%
135         #2/#3/.code n args={\csuse{multilang@@typeargc@#3}}{%
136           \csexpandonce{multilang@@runcmd@#3}}}}%
137       \edef\do{\egroup\noexpand\pgfqkeys{\multilang@keyof{#1}}{%
138         #2/#3/.style n args={\csuse{multilang@@typeargc@#3}}{%
139           {#2=\csexpandonce{multilang@@runcmd@#3}}}}%
140     }%
141   }\do}
```

7.4.1 Argument Aliases

\multilang@regcomb The \multilang@regcomb{\langle cmd-or-env\rangle}{\langle alias\rangle}{\langle fields\rangle} macro registers an \langle alias\rangle argument for \langle fields\rangle.

```

142 \newcommand\multilang@regcomb[3]{%
143   \multilang@regcombtpe{#1}{#2}{#3}{%
144     \forlistloop{\multilang@regcomb@i{#1}{#2}{#3}}{\multilang@@types}}
```

\multilang@regcomb@i The \multilang@regcomb@i{\cmd-or-env}{\alias}{\fields}{\type} macro is an auxiliary front-end to \multilang@regcombtype that transforms \type to a key \suffix (by prepending a "/").

```
145 \newcommand{\multilang@regcomb@i}[4]{%
146   \multilang@regcombtype{\#1}{\#2}{\#3}{/\#4}}
```

\multilang@regcombtype The \multilang@regcombtype{\cmd-or-env}{\alias}{\fields}{\suffix} registers the \alias with the given type-\suffix.

```
147 \newcommand{\multilang@regcombtype}[4]{%
148   \bgroup
```

We count the number of field names in \fields (in \tempcpta) and, in the same loop, gather the individual field assignments (in \toks@).

```
149   \toks@{}\@tempcpta=0\relax
150   \forcsvlist{%
151     \advance\@tempcpta by1\relax
152     \expandafter\multilang@regcomb@set\expandafter{\the\@tempcpta}{\#4}%
153   }{\#3}%
```

Finally, we set the style for the \alias\suffix key. Again we separately handle the case of a single field. Additionally, we treat also the case of *no* field special: It gets a **style** (with 1 argument), but the argument is essentially ignored (as \toks@ is empty). This makes \alias a “comment” field that is not passed to the command or environment.

```
154   \ifnumgreater{\the\@tempcpta}{1}{%
155     \edef\do{\egroup\noexpand\pgfqkeys{\multilang@keyof{\#1}}{%
156       #2#4/.style n args={\the\@tempcpta}{\the\toks@}}%
157   }{%
158     \edef\do{\egroup\noexpand\pgfqkeys{\multilang@keyof{\#1}}{%
159       #2#4/.style={\the\toks@}}%
160   }%
161   \do}
```

\multilang@regcomb@set The \multilang@regcomb@set{\index}{\suffix}{\field} macro appends the sequence “\field\suffix=\#\index” to the \toks@ register. When used in a .style n args key, this sets the “\field\suffix” key to the \index-th positional parameter.

```
162 \newcommand{\multilang@regcomb@set}[3]{%
163   \toks@\expandafter{\the\toks@,\#3#2={##1}}}
```

7.4.2 Language “Types”

\multilang@addlanguage The \multilang@addlanguage{\language} registers \language, essentially registering “\language” and “\language!” as argument datatypes.

```
164 \newcommand{\multilang@addlanguage}[1]{%
```

The following checks the current language (\languagename) against \language. In the following, ##1 is the argument to the key when the key is used.

```
165 \ifdefstring{\languagename}{#1}{%
```

```

166      {\NewMultilangType{#1}{##1}}%
167      {\NewMultilangType@code{#1}{}}%
The following defines “⟨language⟩!” key for forcing an argument to be formatted in language ⟨language⟩.
168  \NewMultilangType{#1!}{\foreignlanguage{#1}{##1}}}

Register all languages passed as argument to the package.
169 \forlistloop{\multilang@addlanguage}{\multilang@@langs}

```

7.5 Auxiliary Macros

\multilang@keyof	The \multilang@keyof{⟨cmd-or-env⟩}, when fully expanded such as in the first argument of the \pgfqkeys macro, represents the key under which the parameter keys of the command or environment ⟨cmd-or-env⟩ are stored. Note that the branching in the code below checks whether ⟨cmd-or-env⟩ is a command sequence (true case) or not (false case).
	<pre> 170 \newcommand{\multilang@keyof}[1]{% 171 \ifcat\relax\noexpand#1% 172 /multilang/cmd/\expandafter\gobble\string#1% 173 \else 174 /multilang/env/#1% 175 \fi} </pre>
\multilang@error	The \multilang@error{⟨message⟩} macro shows ⟨message⟩ as an error message of the multilang package.
	<pre> 176 \newcommand{\multilang@error}[1]{\PackageError{multilang}{#1}{}} </pre>
\multilang@processargs	The \multilang@processargs{⟨cmd-or-env⟩}{⟨opt-key⟩}{⟨options⟩} macro processes the ⟨options⟩ in key ⟨opt-key⟩. Afterwards, it post-processes the arguments <code>margs</code> , <code>oargs</code> , <code>alias/...</code> , and <code>disabable</code> . It stores the result of the post-processing in the macros \multilang@@actuals, \multilang@@checks, and \multilang@@keys.
	<pre> 177 \newcommand{\multilang@processargs}[3]{% 178 \let\multilang@@aliases=\empty 179 \pgfqkeys{#2}{#3}} </pre>
	In the following, we iteratively construct three macros: \multilang@@actuals, \multilang@@checks, and \multilang@@keys. <ul style="list-style-type: none"> • In \multilang@@actuals, we step by step construct the arguments that shall be passed to ⟨cmd-or-env⟩. • In \multilang@@checks, we construct a list of preliminary checks that ⟨cmd-or-env⟩ shall perform on its arguments. • In \multilang@@keys, we construct a list of \pgfqkeys commands that set up the keys for ⟨cmd-or-env⟩’s one argument.
	The following first initializes the three macros.
	<pre> 180 \edef\multilang@@actuals{}% 181 \def\multilang@@checks{}% 182 \def\multilang@@keys{}% </pre>

We first process the optional arguments. We process them before the mandatory arguments because they must come first in `\multilang@@actuals`. In the following `\do` macro, `##1` iterates over all optional argument names in the `margs` list.

```

183  \ifdefvoid{\multilang@@oargs}{%
184    \def\do##1{%
185      \appto{\multilang@@actuals}{%
186        \ifcsmacro{\multilang@@val@##1}{%
187          {[ \csexpandonce{\multilang@@val@##1} ]}%
188        {}%
189      }%
190      \appto{\multilang@@keys}{\multilang@regfield{#1}{##1}}%
191    }%
192    \expandafter\docslist\expandafter{\multilang@@oargs}}%

```

Next, we append the mandatory arguments, specified by the `margs` list. In the following `\do` macro, `##1` iterates over all mandatory argument names in the `margs` list.

```

193  \ifdefvoid{\multilang@@margs}{%
194    \def\do##1{%
195      \appto{\multilang@@actuals}{%
196        {\csexpandonce{\multilang@@val@##1}}%
197      }%
198      \appto{\multilang@@checks}{%
199        \ifcsmacro{\multilang@@val@##1}{%
200          {}%
201          {\multilang@error{mandatory argument ##1 missing}}%
202        }%
203        \appto{\multilang@@keys}{\multilang@regfield{#1}{##1}}%
204      }%
205      \expandafter\docslist\expandafter{\multilang@@margs}}%

```

Afterwards, we handle argument aliases. The list of aliases' names is in `\multilang@@aliases` and the list of arguments that a `\alias{alias}` combines is in `\multilang@@alias@{alias}`. Note that aliases only modify `\multilang@@keys` – i.e., not `\multilang@@actuals` or `\multilang@@checks`.

```

206  \def\do##1{%
207    \eappto{\multilang@@keys}{%
208      \unexpanded{\multilang@regcomb{#1}{##1}}%
209      {\csuse{\multilang@@alias@##1}}}}%
210  \expandafter\dolistloop\expandafter{\multilang@@aliases}}%

```

To handle `disablable` macros, we simply add the `disabled` key. This key is a Boolean key that just sets a conditional.

```

211  \ifbool{\multilang@@disablable}{%
212    \eappto{\multilang@@keys}{%
213      \noexpand\pgfqkeys{\multilang@keyof{#1}}{%
214        disabled/.is if={\multilang@cmd@@disabled}}}}%
215  {}%

```

Invoke the hook. To avoid macro arguments to the hook, we store the command or environment name in `\multilang@@cmdorenv` and store the `pgfkeys` key for the command or environment in `\multilang@cekey`.

```
216 \def\multilang@@cmdorenv{\#1}%
217 \edef\multilang@@cekey{\multilang@keyof{\#1}}%
218 \multilang@hook@processargs
219 }
```

`\multilang@hook@processargs` The `\multilang@hook@processargs` macro is a hook that enables extensions to the `\multilang@processargs`.

```
220 \newcommand\multilang@hook@processargs{}
```

`\multilang@noend` The `\multilang@noend{\langle body\rangle}` macro is intended to be used as the argument to `\Collect@Body` (or `\collect@body`) of the `environ` package. In this context, it performs the following: Firstly, it ignores the collected `\langle body\rangle` of the environment. Secondly, it temporarily disables the `\end`-code of the environment in which the `\Collect@Body` is expanded.

```
221 \newcommand\multilang@noend[1]{\cslet{\end}{\currenvir}\relax}
```

8 Implementation of Tags

`\SetTagFilter` The `\SetTagFilter[\langle default\rangle]{\langle policy\rangle}` sets the tag filter policy based on accept/deny rules in `\langle policy\rangle`.

```
222 \newcommand\SetTagFilter[2][accept]{%
223   \kcvml@parsepolicy{\kcvml@tagfilter}{\#1}{\#2}}
224 \newcommand\kcvml@tagfilter{}
```

`\DefineTagFilter` The `\DefineTagFilter{\langle name\rangle}{\langle default\rangle}{\langle policy\rangle}` macro defines a tag filter policy under name `\langle name\rangle`.

```
225 \newcommand\DefineTagFilter[3]{%
226   \kcvml@parsepolicy{\kcvml@filter@@\#1}{\#2}{\#3}}
```

`\UseFilter` The `\UseTagFilter{\langle name\rangle}` macro uses the previously defined tag filter policy with name `\langle name\rangle`.

```
227 \newcommand\UseTagFilter[1]{%
228   \letcs{\kcvml@tagfilter}{\kcvml@filter@@\#1}}
```

`\kcvml@parsepolicy` The `\kcvml@parsepolicy{\langle csname\rangle}{\langle default\rangle}{\langle policy\rangle}` parses a tag filter policy, `\langle policy\rangle` with default `\langle default\rangle`, and stores the resulting filter in the control sequence `\langle csname\rangle`.

```
229 \newcommand\kcvml@parsepolicy[3]{%
```

We first reset the temporary filter variable `\kcvml@tmptagfilter` then populate it via `\pgfqkeys`, first with the `\langle policy\rangle` list and subsequently with the `\langle default\rangle` policy.

```
230 \bgroup
231 \def\kcvml@tmptagfilter{}%
232 \pgfqkeys{\kcvml/tagfilter}{\#3,default/\#2}%

```

Now we export the temporary `\kcvml@@tmptagfilter` to outside the local group and into the control sequence `\csname`.

```
233 \edef\do{\egroup
234   \unexpanded{\csdef{\#1}{\expandonce{\kcvml@@tmptagfilter}}}}
235 \do
```

The following lines specify how `\kcvml@@tmptagfilter` is modified when `accept` or `deny` filter rules are specified.

```
236 \pgfqkeys{kcvml/tagfilter}{%
237   accept/.code={\kcvml@appendrule{\#1}{\boolfalse}{\booltrue}},
238   deny/.code ={\kcvml@appendrule{\#1}{\booltrue}{\boolfalse}},
239   default/accept/.code n args={0}{\appto\kcvml@@tmptagfilter{%
240     \kcvml@applydefault{\boolfalse}}},
241   default/deny/.code n args={0}{\appto\kcvml@@tmptagfilter{%
242     \kcvml@applydefault{\booltrue}}},
243 }
```

`\kcvml@appendrule` The `\kcvml@appendrule{\ruletags}{\flagmacro}{\invmacro}` macro appends a filter rule to the overall tag filter. The rule filters for the given comma-separated list `\ruletags`. The `\flagmacro` specifies whether a match shall be disabled (if `\booltrue`) or enabled (if `\boolfalse`). The `\invmacro` must be the inverse of `\flagmacro`.

```
244 \newcommand\kcvml@appendrule[3]{%
245   \bgroup
```

For simplified later processing, we turn `\ruletags` into an etoolbox list (in `\kcvml@ruletags`) first.

```
246 \def\kcvml@ruletags{}%
247 \forcsvlist{\listadd{\kcvml@ruletags}}{\#1}%
```

Now we append to the overall filter (in `\kcvml@@tmptagfilter`) outside the local group and use the cascade of `\expandafters` to get `\cvmkl@@ruletags` out of the group without polluting the outer scope.

```
248 \expandafter\egroup
249 \expandafter\listadd\expandafter\kcvml@@tmptagfilter\expandafter{%
250   \expandafter\kcvml@applyrule\expandafter{\kcvml@ruletags}{\#2}{\#3}}}
```

We add the additional `tags` key to every disableable multilingual macro and environment. For this, we use multilang's `\multilang@hook@processargs` hook.

```
251 \appto\multilang@hook@processargs{%
252   \ifbool{\multilang@disnableable}{%
253     {\eappto{\multilang@keys}{%
```

Note that in `\multilang@cekey`, the parent key of the command or environment is stored. Whenever the `tags` argument is used, we make it invoke `\kcvml@applyfilter` with the given `\tags` (`\#1`).

```
254   \noexpand\pgfqkeys{\multilang@cekey}{%
255     tags/.code={\noexpand\kcvml@applyfilter{\#1}}}}}
256 {}}
```

\kcvml@applyfilter The \kcvml@applyfilter{\langle tags\rangle} macro applies the current filter, which is in the \kcvml@@tagfilter etoolbox list, to the comma-separated list \langle tags\rangle of tags to check whether the entity with the \langle tags\rangle should be disabled or not. The result of the check is stored in the Boolean flag multilang@cmd@@disabled for further use in with multilang code.

```
257 \newcommand\kcvml@applyfilter[1]{%
258   \ifbool{multilang@cmd@@disabled}{\relax}{%
```

We check the filter only if the entry has not already explicitly been marked as disabled. I.e., explicit disabling takes precedence, no matter whether it is specified before or after a tags element. In the Boolean flag kcvml@@match, we store whether a tag in \langle tags\rangle matched one of the accept/deny filters already. After initializing this flag, we iterate through \kcvml@@tagfilter with \do{\langle rule\rangle}, and we stop iterating after a match has been found.

```
259   \boolearn{multilang@match}%
260   \def\do##1{%
```

Note that \langle rule\rangle is a curried macro whose missing last argument, \langle tags\rangle, is added here.

```
261   ##1{#1}%
262   \ifbool{multilang@match}{\listbreak}{%
263   \dolistloop{\kcvml@@tagfilter}}%
264 \newbool{multilang@match}
```

\kcvml@applyrule The \kcvml@applyrule{\langle ruletags\rangle}{\langle flagmacro\rangle}{\langle invmacro\rangle}{\langle tags\rangle} macro applies a single filter rule to the given \langle tags\rangle. The \langle flagmacro\rangle must be either \booltrue or \boolearn. If it is \booltrue, this specifies that a match of \langle tags\rangle against \langle ruletags\rangle disables the display of the respective entity; If it is \boolearn, this specifies that a match enables the display. The \langle invmacro\rangle must be the inverse of \langle flagmacro\rangle. We just check each tag in \langle tags\rangle individually.

```
265 \newcommand\kcvml@applyrule[4]{%
266   \forcsvlist{\kcvml@applyrule@i{#1}{#2}{#3}{#4}}%
```

\kcvml@applyrule@i The \kcvml@applyrule@i{\langle ruletags\rangle}{\langle flagmacro\rangle}{\langle invmacro\rangle}{\langle tag\rangle} macro applies a single filter rule to the given \langle tag\rangle. It checks whether the \langle tag\rangle is inverted (i.e., starting with “!”) and hands over the actual check to \kcvml@applyrule@ii.

```
267 \newcommand\kcvml@applyrule@i[4]{%
268   \if !\@car\@nil
269     \expandafter\kcvml@applyrule@ii\expandafter{\@cdr\@nil}{#1}{#3}%
270   \else
271     \kcvml@applyrule@ii{#4}{#1}{#2}\fi}
```

\kcvml@applyrule@ii The \kcvml@applyrule@ii{\langle tag\rangle}{\langle ruletags\rangle}{\langle flagmacro\rangle} macro applies a single filter rule to the given \langle tag\rangle. We check whether \langle tag\rangle is in the etoolbox list \langle ruletags\rangle. Since \ifinlist expects a list macro for its second argument (which it then expands once), we just give it \empty to eat (expand) and then use \langle ruletags\rangle as it is.

```
272 \newcommand\kcvml@applyrule@ii[3]{%
273   \ifinlist{#1}{\empty}{#2}%
274 }
```

If we have a match, we apply `\flagmacro` to `multilang@cmd@@disabled` and then record, in `kcvml@@match`, that we found a match.

```
274     {#3{multilang@cmd@@disabled}\booltrue{kcvml@@match}}{}}
```

- `\kcvml@applydefault` The `\kcvml@applydefault{\flagmacro}{\tags}` macro applies a default filter to `\tags`. A `\flagmacro` of `\booltrue` corresponds to a “default deny”; `\boolfalse` corresponds to “default accept”.

```
275 \newcommand\kcvml@applydefault[2]{%
276   #1{multilang@cmd@@disabled}}
```

9 Implementation of Sectioning Environments

The sectioning environments are proxies for the corresponding sectioning macros. They are defined as environments such that the whole environments rather than just the headings can be disabled. Each of the environments has one optional argument, `short` (for a short title), and one mandatory argument, `title` (for the actual title).

- `Section` The `Section` and `Section*` environments are multilingual proxies to `\section` and, respectively, `\section*`.

```
277 \NewMultilangEnv{Section}{disablable,
278   environment=section, oargs=short, margs=title}
279 \NewMultilangEnv{Section*}{disablable,
280   environment=multilang@secstar, oargs=short, margs=title}
281 \newenvironment{multilang@secstar}{\section*}{}{}
```

- `SubSection` The `SubSection` and `SubSection*` environments are multilingual proxies to `\subsection` and, respectively, `\subsection*`.

```
282 \NewMultilangEnv{SubSection}{disablable,
283   environment=subsection, oargs=short, margs=title}
284 \NewMultilangEnv{SubSection*}{disablable,
285   environment=multilang@ssecstar, oargs=short, margs=title}
286 \newenvironment{multilang@ssecstar}{\subsection*}{}{}
```

- `SubSubSection` The `SubSubSection` and `SubSubSection*` environments are multilingual proxies to `\subsubsection` and, respectively, `\subsubsection*`.

```
287 \NewMultilangEnv{SubSubSection}{disablable,
288   environment=subsubsection, oargs=short, margs=title}
289 \NewMultilangEnv{SubSubSection*}{disablable,
290   environment=multilang@sssecstar, oargs=short, margs=title}
291 \newenvironment{multilang@sssecstar}{\subsubsection*}{}{}
```

- `Paragraph` The `Paragraph` and `Paragraph*` environments are multilingual proxies to `\paragraph` and, respectively, `\paragraph*`.

```
292 \NewMultilangEnv{Paragraph}{disablable,
293   environment=paragraph, oargs=short, margs=title}
294 \NewMultilangEnv{Paragraph*}{disablable,
```

```

295   environment=multilang@parstar, oargs=short, margs=title}
296 \newenvironment{multilang@parstar}{\paragraph*}{}{}
```

SubParagraph The SubParagraph and SubParagraph* environments are multilingual proxies to \subparagraph and, respectively, \subparagraph*.

```

297 \NewMultilangEnv{SubParagraph}{disablable,
298   environment=subparagraph, oargs=short, margs=title}
299 \NewMultilangEnv{SubParagraph*}{disablable,
300   environment=multilang@sparstar, oargs=short, margs=title}
301 \newenvironment{multilang@sparstar}{\subparagraph*}{}{}
```

Change History

v0.9

General: Initial version 1

v0.9b

General: Package author's name
change 1

Index

Symbols

\@car	268
\@cdr	269
\@currenvir	221
\@gobble	13, 172
\@ifstar	18
\@nil	268, 269
\@tempcnta	149, 151, 152, 154, 156

A

\advance	151
\appto	185, 190, 195, 198, 203, 239, 241, 251

B

\begin	52
\bgroup	10, 25, 39, 44, 120, 148, 230, 245
\boolfalse	26, 45, 237, 238, 240, 259
\booltrue	237, 238, 242, 274

C

\Collect@Body	49
-------------------------	----

\csdef	65, 86, 88, 90, 91, 93, 95, 97, 99, 101, 103, 105, 107, 113, 117, 234
\csexpandonce	124, 128, 135, 139, 187, 196
\cslet	221
\csname	13, 85
\csuse	88, 91, 93, 95, 97, 99, 101, 103, 105, 107, 121, 134, 138, 209

D

\DefineTagFilter	9, 225
\do	14, 37, 41, 58, 123, 127, 133, 137, 141, 155, 158, 161, 184, 194, 206, 233, 235, 260
\docslist	192, 205
\dolistloop	210, 263

E

\eappto	207, 212, 253
\egroup	14, 35, 41, 54, 123, 127, 133, 137, 155, 158, 233, 248
\else	109, 173, 270
\empty	178, 273

\end	56
\endcsname	13, 85
environments:	
Paragraph	8, 292
Paragraph*	8, 292
Section	8, 277
Section*	8, 277
SubParagraph	8, 297
SubParagraph*	8, 297
SubSection	8, 282
SubSection*	8, 282
SubSubSection	8, 287
SubSubSection*	8, 287
\expandafter ...	12, 13, 35, 54, 85, 152, 163, 172, 192, 205, 210, 248, 249, 250, 269
\expandoncde	15, 19, 20, 22, 24, 28, 31, 33, 34, 42, 50, 53, 234
F	
\fi	110, 175, 271
\forcsvlist ...	6, 150, 247, 266
\foreignlanguage	168
\forlistloop ...	118, 144, 169
I	
\if	268
\ifbool .	16, 29, 47, 211, 252, 258, 262
\ifcase	87
\ifcat	171
\ifcsdef	122, 132
\ifcsequal	89
\ifcsmacro	186, 199
\ifdefstring	165
\ifdefvoid	183, 193
\infinlist	273
\ifnumequal	121
\ifnumgreater	154
K	
\kcvml@ruletags .	246, 247, 250
\kcvml@tagfilter	224, 228, 263
\kcvml@tmptagfilter	231, 234, 239, 241, 249
\kcvml@appendrule	237, 238, 244
\kcvml@applydefault .	240, 242, <u>275</u>
\kcvml@applyfilter ...	255, <u>257</u>
\kcvml@applyrule	250, <u>265</u>
\kcvml@applyrule@i	266, <u>267</u>
\kcvml@applyrule@ii .	269, 271, <u>272</u>
\kcvml@parsepolicy	223, 226, <u>229</u>
L	
\languagename	165
\let	178
\letcs	228
\listadd	6, 84, 247, 249
\listbreak	262
\listeadd	64
M	
\multilang@@actualls	34, 53, 180, 185, 195
\multilang@@aliases	64, 178, 210
\multilang@@cekey	217, 254
\multilang@@checks	31, 50, 181, 198
\multilang@@cmd	33, 73
\multilang@@cmdorenv	216
\multilang@@defaults ..	28, 66
\multilang@@env	52, 56, 80
\multilang@@intcmd .	12, 19, 20, 22, 24
\multilang@@invok	30, 32, 35, 49, 51, 54
\multilang@@keys ..	15, 42, 182, 190, 203, 207, 212, 253
\multilang@@langs ...	4, 6, 169
\multilang@@margs .	60, 193, 205
\multilang@@oargs .	61, 183, 192
\multilang@@types	84, <u>111</u> , 118, 144
\multilang@addlanguage	164, 169
\multilang@error .	109, <u>176</u> , 201
\multilang@hook@processargs ..	
	... 218, <u>220</u> , 251
\multilang@keyof	27, 46, 116, 123, 127, 133, 137, 155, 158, <u>170</u> , 213, 217

\multilang@noend	49, 221
\multilang@processargs .	11, 40 ,
177	
\multilang@regcomb	142 , 208
\multilang@regcomb@i	144 , 145
\multilang@regcomb@set	152 , 162
\multilang@regcomptype	143 ,
146 , 147	
\multilang@regfield	115 , 190 ,
203	
\multilang@regfieldtype	118 ,
119	
N	
\newbool	69 , 70 , 77 , 264
\NewMultilangCmd	2 , 9
\NewMultilangEnv	6 ,
38 , 277 , 279 , 282 , 284 , 287 ,	
289 , 292 , 294 , 297 , 299	
\NewMultilangType	6 , 83 , 114 ,
166 , 168	
\NewMultilangType@code	112 , 167
\noexpand	18 , 24 , 26 , 27 , 29 , 33 ,
45 , 46 , 47 , 52 , 56 , 123 , 127 ,	
133 , 137 , 155 , 158 , 171 , 213 ,	
254 , 255	
O	
\or	89 , 93 , 95 , 97 , 99 , 101 , 103 ,
105 , 107	
P	
\PackageError	176
\Paragraph (environment)	8 , 292
\paragraph	296
\Paragraph* (environment)	8 , 292
\pgfkeyscurrentname	64 , 65
R	
\pgfqkeys	5 , 27 , 46 , 59 , 71 , 78 ,
116 , 123 , 127 , 133 , 137 , 155 ,	
158 , 179 , 213 , 232 , 236 , 254	
\ProcessPgfOptions	8
S	
Section (environment)	8 , 277
\section	281
Section* (environment)	8 , 277
\SetTagFilter	9 , 222
\string	13 , 172
\SubParagraph (environment)	8 , 297
\subparagraph	301
\SubParagraph* (environment)	8 , 297
\SubSection (environment)	8 , 282
\subsection	286
\SubSection* (environment)	8 , 282
\SubSubSection (environment)	8 , 287
\subsubsection	291
\SubSubSection* (environment)	8 , 287
T	
\the	152 , 154 , 156 , 159 , 163
\toks@	149 , 156 , 159 , 163
U	
\unexpanded	17 , 22 , 30 , 32 , 33 , 35 ,
43 , 48 , 51 , 54 , 208 , 234	
\UseFilter	227
\UseTagFilter	9 , 227