

The knowledge package

[v1.34 — 2026/06/02]

Thomas Colcombet
thomas.colcombet@irif.fr

June 2, 2026

Abstract

The `knowledge` package offers commands and notations for handling semantical notions in a document. This allows to easily link the use of a notion to its definition, to add it to the index automatically, etc.

Status of this version

contact: thomas.colcombet@irif.fr
version: v1.34
date: 2026/06/02 (documentation produced June 2, 2026)
license: LaTeX Project Public License 1.2
web: <https://ctan.org/pkg/knowledge>
CTAN: <https://www.ctan.org/pkg/knowledge>

Contents

1	History	4
2	Quick start	8
2.1	Linking to outer documents/urls, and to labels	8
2.2	Linking inside a document	10
2.3	Mathematics	13
3	Usage of the <code>knowledge</code> package	14
3.1	Options and configuration	14
3.1.1	Options at package loading	14
3.1.2	Writing mode	14
3.1.3	Automatic loading of other packages	15
3.1.4	Configuring and <code>\knowledgeconfigure</code>	16
3.1.5	Other configuration option	17
3.2	What is a <code>knowledge</code> ?	17
3.3	The <code>\knowledge</code> command and variations	17
3.3.1	General description of the <code>\knowledge</code> command	17
3.3.2	Targeting and the corresponding directives	18
3.3.3	General directives	19
3.3.4	Knowledge styles and the <code>\knowledgestyle</code> command	21
3.3.5	New directives: the <code>\knowledgedirective</code> command	21
3.3.6	<code>\knowledgestyle</code> versus <code>\knowledgedirective</code>	22
3.3.7	Default directives: the <code>\knowledgedefault</code> command	23
3.4	The <code>\kl</code> command and variants	23
3.4.1	The standard syntax	23
3.4.2	The quotation notation	23
3.4.3	Variants	25
3.4.4	Defining your own variants	25
3.4.5	Examples of variants	26
3.5	Scoping	28
3.5.1	Principles of scoping	28
3.5.2	Scoping by examples	29
3.5.3	What is the structure of scopes in a document	30
3.5.4	How is chosen the scope of a <code>knowledge</code> ?	31
3.5.5	Naming scopes: the <code>\knowledgeimport</code> , <code>\knowledgescope</code> and <code>label</code> commands	32
3.5.6	Managing scoping environments	33
3.6	Error handling	34
3.7	The <code>diagnose</code> file	34
3.8	Other packages	35
3.8.1	The <code>xcolor</code> option	35
3.8.2	The <code>hyperref</code> option	36
3.8.3	The <code>makeidx</code> and <code>imakeidx</code> options	40
3.8.4	The <code>cleveref</code> option	41
3.9	Dealing with math	41
3.9.1	Defining knowledge-aware macros	42
3.9.2	The combination <code>\withkl</code> / <code>\cmdkl</code>	44
3.9.3	Defining macros for math: the <code>mathcommand</code> package	44

3.9.4	Mathematical objects that are singly introduced	45
3.9.5	Context dependent variables	46
3.10	Predefined configuration	46
3.10.1	The <code>notion</code> directive	46
3.11	Fixes	47
4	Some questions and some answers	48
4.1	How to compile?	48
4.2	Problem with <code>\item</code> parameters	48
4.3	Knowledges and moving arguments (table of contents, ...).	48
4.4	Problems with <code>tikzcd</code> and other issues with the quotation notation	49
4.5	Problems with <code>amsmath</code>	50
4.6	Hyperref complains	50
4.7	Name clash (eg with the <code>complexity</code> package)	50
4.8	Incorrect display	51
4.8.1	Incorrect breaking at the end of lines (in <code>arXiv</code> for instance)	51
4.9	Display errors in pdf output, in particular <code>arXiv</code>	51
4.9.1	Red boxes around links	51
4.9.2	Incorrect color for links in <code>paper mode</code> (e.g. red in with <code>acmart</code>)	51
4.9.3	Unexpected color in margin paragraph	51
4.10	Problems with scope	52
4.10.1	Problems in combination with <code>\bibitem</code> and the environment <code>thebibliography</code>	52
4.11	Editors	52
4.11.1	<code>Emacs</code> editor and quotes	52
4.12	Others	53
5	Resources	54
5.1	List of commands	54
5.2	List of environments	54
5.3	List of directives (to use with <code>\knowledge</code>)	54
5.4	List of configuration directives	55

1 History

- 2016-06-07** `\knowledgemacro` is now renamed to `\knowledgedirective` .
- 2017-01-13** `\AP` has been recoded, and is now more properly aligned in the margin. The `visible anchor points` option has also been made usable without the `xcolor` package.
- 2017-01-13** The package `scope option` can now be omitted. This in particular avoid clashes with the over-restriction on the structure of the document it entails. It should be improved to stop overloading the `\begin` command.
- 2017-01-14** The overloading of `\begin` and `\end` was done as protected commands, which should not be the case to be consistent with the behaviour of LaTeX (for instance, this was giving an extra line in the title in the conference mode of the class `IEEEtran`). Corrected: these commands are not protected anymore.
- 2017-01-15** A workaround for an incompatibility between the `hyperref` and the two-column mode as been added in the macro `\knowledgeFixHyperrefTwocolumn` (thanks to Daniela Petrişan).
- 2017-01-15** Added the `directive synonym`.
- 2017-01-15** Added the `noknowledge` package for minimizing the effects of not having `knowledge` activated.
- 2017-01-17** Changed the way options are handled, decoupling the package options (options of `\usepackage`) from the configuration options (see `\knowledgeconfigure`).
- 2017-01-17** Proper treatment of ‘final’ option and `composition` options.
- 2017-01-17** Added `\IfKnowledgeFinalMode [TF]` commands for the user.
- 2017-01-17** Added the option `fix hyperref twocolumn` as a shorthand for calling `\knowledgeFixHyperrefTwocolumn` (thanks to Daniela Petrişan and Luca Reggio).
- 2017-01-18** Added the configuration option `notion` that offers a basic configuration compatible with `xcolor` or not, and `final` and `composition` modes.
- 2017-01-19** Added `\phantomintro` and an explanation on how to deal with `align*`.
- 2017-02-20** Removed the warnings of latex for unknown labels in `autoref`.
- 2017-02-20** Removed nasty error making `\AP` not operative when anchor points were not visible.
- 2017-02-21** Added the `protect link` directive.
- 2017-02-21** Added the `hyperlinks=` configuration.
- 2017-02-27** `visible anchor points` is active by default now.
- 2017-02-27** A simple example is now included.
- 2017-02-28** Minor changes on the documentation.
- 2017-02-28** Added the `scope` environment.
- 2017-02-28** Added the `protect link` and `unprotect link` configuration directives.
- 2017-02-28** Added the `\knowledgeconfigureenvironment` command.
- 2017-03-03** Added the `breaklinks` faq (thanks to Luca Reggio for the request).
- 2017-03-10** Added the `"..."` and `"..."` notations and the `quotation` mode (requested by Gabriele Puppis and Andreas Krebs).
- 2017-03-11** Added the `"...@..."` and `"...@..."` notations.
- 2017-03-13** Corrected for being compatible with version of `expl3` posterior to Mars 2015 (`\c_sys_jobname_str` does not exist anymore). (Thanks to Jean-Éric Pin).
- 2017-03-14** Corrected that the `@` letter was left a letter after `\knowledgeFixHyperrefTwocolumn` .
- 2017-04-09** Internal change of code, for `scope` handling and for the `quotation notation`: slowly going toward an extended `quotation notation` that can make the `scope` of search explicit.
- 2017-04-09** Added the `protect quotation` configure option, that is given a list of environments, and deactivates automatically the `quotation notation` when in there

- environments. This is a simple code for the moment. Typically, one can use `\knowledgeconfigure {protect quotation=tikzcd}`. For the moment, it is not explained in the document.
- 2017-04-19** Changed the display code such that nested knowledges behave properly: before, the introduction would be performed for the object and the subobjects.
 - 2017-04-20** The `electronic mode` has been added, and the ‘final mode’ is now renamed into `paper mode`. The `\knowledgepackagemode` configuration variable is also available for easier scripting.
 - 2017-06-06** FAQ on deactivating the quote in Emacs (thanks to Sylvain Perifel).
 - 2017-06-08** Removed the `noknowledge` package and all references to it.
 - 2017-06-08** Removed the `knowledgeutils.sty` and `scopearticle.sty` which are now integrated in the main file.
 - 2017-06-08** The file `knowledge-example.tex` has been improved.
 - 2017-06-09** First release of version 1.0 on CTAN.
 - 2017-06-10** Corrected the `quotation notation` to make it expandable for avoiding problems in table of contents (the @ was not working).
 - 2017-06-11** Corrected a bug linked to changes of expl3 on recent distributions (pointed by Murray Eisenberg). Release of v1.01 on CTAN.
 - 2017-06-27** Overloaded labels now perform an expansion of the argument (this was causing problems with biblatex).
 - 2017-06-28** Options `log-declarations` of `xparse` package removed (causing clash with other packages, as pointed by Juliusz Chroboczek). Release of v1.02 on CTAN.
 - 2017-06-30** added the field `labelizable_bool` to `areas`. Coded missing features of scoping. Now the `scope=` directive works with as parameter an enclosing `area`, or a label.
 - 2017-06-30** Added in the source a Regression subdirectory containing files to be tested (so far only one: `regression-scope.tex`)
 - 2017-07-01** Corrected a conflict between the `scope` and `makeidx` option.
 - 2017-07-03** Scoping becomes operational.
 - 2017-07-04** The documentation for `notion` and `intro notion` are added (thanks to Fabian Reiter).
 - 2017-07-09** Added boolean `environment_bool` field to `areas`, in order to resolve an incompatibility with the package `standalone` noticed by Fabian Reiter.
 - 2017-07-20** `Scoping` becomes fully operational, with the parenthesis notation of `\kl` and `\intro`. The use of scope has been recoded. Now scope links reuse implicitly the key as a link. Documentation updated.
 - 2017-07-26** File and line numbers added in the `kaux file`. Added the option `diagnose line=` to deactivate it.
 - 2017-07-26** Corrections to the documentation. Version 1.03 on CTAN.
 - 2017-07-28** Corrected a bug of scoping in the context of synonyms. Added ctan for producing the ctan zip file.
 - 2017-08-06** Now passes the compliance test `check-declarations` of `expl3` (thanks to Marc Zeitoun)
 - 2017-09-12** The `hidelinks` option of `hyperref` is now always activated.
 - 2017-09-25** Ancient version of `xparse` does not have `\NewExpandableDocumentCommand`. Corrected. Version 1.05 on CTAN.
 - 2017-10-10** Bug in the implementation of `\knowledgevariant` (that was invisible for older versions of `expl3`). Found and corrected (thanks to Marc Zeitoun). Version 1.06 on CTAN.
 - 2017-10-15** Diagnose extended (suggested by Fabian Reiter). Minor corrections. Version 1.07 on CTAN.
 - 2017-10-17** Added `cyclic color` and `cyclic colors=`. Reorganization of the structure of the code for producing a better CTAN archive. Version 1.08 on CTAN.
 - 2018-01-31** Added the `strict` configuration option.

2018-02-05 Added the `smallcaps` formatting directive.
2018-02-17 Corrected incompatibility with latest version of `expl3`. Version 1.10 on CTAN.
2018-02-21 Bug correction concerning the activation of scopes.
2018-02-21 Documentation improvement for `Emacs` (thanks to Michaël Cadilhac).
2018-02-24 Documentation improvement for the environment `thebibliography`.
2018-05-17 Correction to be compatible with the latest version of `expl3` (thanks to Leo Stefanescu).
2018-07-26 Compatibility with utf8 symbols in labels (thanks to Yves Guiraud).
2018-11-22 Corrected bug for `makeidx` (thanks to Sylvain Schmitz). V1.14 on CTAN.
2019-01-27 Minor improvement of the doc, and hiding links in it. V1.15.
2019-02-15 Correction of a placement problem with `\AP`. V1.16.
2019-05-23 Adding of the `'|'`-notation for the `\knowledge` command. `Explicit scopes` are introduced. Updating of the documentation. `up` directive in math mode now silently does nothing, and `\knowledgedirective` now forbids redefinitions by default (thanks to Léo Stefanescu).
2019-07-02 Removing the `'kl'` and `'intro'` styles that prevented a proper configuration of `intro notion` (thanks to Léo Stefanescu).
2019-10-03 Update of the documentation, and V1.17.
2019-10-27 Bug correction and added the `'patch label'` configuration directive (thanks to Rebecca Turner). V1.18.
2019-11-19 Now the labels are evaluated before being written to the `kaux file` in a `\KAuxNewLinkScopetagInstance` command (bug fix). V1.19.
2019-11-29 Help added in the `diagnose file`. `bar suggestion` (still working) renamed to `diagnose bar`, and activated by default. `patch label` is renamed into `label scope`.
2019-12-02 The `kaux file` is now checked for completeness before being used. This should avoid errors when the previous compilation failed.
2019-12-03 Corrected bug in the scope access. V1.20.
2020-01-25 Corrected bug when `knowledge` is used without `hyperref` (thanks to Rémi Nollet).
2020-01-25 Corrected bug that made the `kaux file` not stabilize (thanks to Rémi Nollet). V1.21 on CTAN.
2020-03-05 Now `hidelinks` and `breaklinks` are automatically activated unless the new option `no patch` is activated. Doc update. V1.22 on CTAN.
2020-04-25 Made the package compatible with 2016 versions of LaTeX. Useful when `knowledge.sty` is included with and compile in arXiv. V1.23 on CTAN.
2020-05-02 Doc update (`acmart` colors, thanks to Daniela Petrișan).
2020-09-22 Adding the `silent` option as suggested by Patrick Lambein-Monette. V1.24.
2021-03-31 Correcting a bug caused by a change of semantics in `expl3`. V1.25.
2021-05-25 Added the `\kref`, `\kpageref`, `\kceref`, and `\kCref` commands as suggested by Ziv Scully.
2021-05-26 Corrected wrongly used default parameter in `\knowledgedirective`.
2021-05-27 The explicit text given in a `\kl` -command now overrides the `text=` directive.
2021-07-18 Adding the `no index` directive.
2021-12-20 Adding options `anchor point shape`, `anchor point color`, `anchor point shift` (request of Rémi Morvan).
2021-12-27 Bug correction for corner shape, and doc update (thanks to Rémi Morvan).
2022-01-12 Minor changes. Removing dead link for the webpage. V1.27
2022-02-12 Adding support of `imakeidx` and the directive `index name=` (request and code change by Maximilian Keßler). V1.28
2024-03-10 Correcting typographic bug of `\withkl` and `\cmdkl` when in combination

with e.g. `\mathrel` (suggested by Rémi Morvan). V1.29

2025-02-20 Correcting macro bug with `\KnowledgeNewDocumentCommand` (thanks to Sarah Winter). V1.30

2025-05-09 Correcting display bug with `makeidx` (thanks to Rémi Morvan). V1.31

2026-02-24 Adding in the documentation a patch for `smartparens` in `emacs` (thanks to Quentin Aristote).

2026-04-02 Adding optional parameter to `\label` hook for compatibility with `cleveref`; adding directory name to the origin of lines (asked by Dorian Lesbre).

2026-06-02 Bug correction in `quotation` mode (double quotes), thanks to Cyprien de Saint Guilhem.

2 Quick start

The `knowledge` package offers several capabilities for handling colors, changing the display style, defining internal and external hyperlinks, producing an index, etc... All these possibilities arise from defining explicitly or implicitly `knowledges` associated to terms in plain english (or other languages).

We start by describing a certain number of problems/scenarii that a user may be confronted to, and show how to solve them. In the subsequent sections, a more detailed account of how the `package` works and can be parameterized is given.

There is also a file `knowledge-example.tex` that can be used as a starting point.

2.1 Linking to outer documents/urls, and to labels

The problem 1 *I have a lot of external url's that I would like to [very] often] have a link to, but I do not want to always type the full url. I do not want to remember weird labels/internal references/macro names either.*

A solution is as follows. One first loads the `knowledge` package with option `hyperref` using either:

```
\usepackage [hyperref,quotation]{knowledge}
```

or equivalently:

```
\usepackage {hyperref}  
\usepackage [quotationa]{knowledge}
```

^aIf you want to use the "... " notation.

Then, in the preamble (or in an external file), one uses commands of the form either:

```
\knowledge {url={https://en.wikipedia.org/wiki/LaTeX}}  
| latex
```

This configures the text `'latex'` to be associated with the sole directive `url=`, which means an hyperreference to this address.

Finally in the body of the paper, the sole extra command `\kl` (or the `"`-symbol if the `quotation` option is activated) is used, with as parameter a text. This text is searched for, and the directives attached to it (here `url=`), are used for formatting its printing¹. Hence:

This package has been written for use in `\kl {latex}`.

or, if the `quotation` option is activated,

This package has been written for use in `"latex"`.

yields

This package has been written for use in latex.

¹This resembles a lot a macro so far. It nevertheless differs in that: (a) if not defined, it does not make the compilation fail as a macro would, and thus does not interfere with the writing process, (b) any text can be used and not only alphabetic letters as in default T_EX, (c) you do not have to care about the space after, and (d) in fact the machinery for resolving the meaning of a `knowledge` is much more powerful than simple macro expansion.

Hint. You may use other options like `xcolor` for allowing debugging with colors (for undefined knowledges).

Hint. If the `knowledge` is not defined, this does not make the compilation fail. In fact, it is good practice to use many `\kl` commands or `"..."` notations while writing a text, and only resolve these questions at the end (see also the [diagnose file](#)).

Variation. But in fact, I would like ‘`latex`’ to also be properly typeset L^AT_EX, and in gray. This requires to load the package with the `xcolor` option (for being able to use colors, obviously), or by loading the package `xcolor` before, and then modify the `\knowledge` command using extra directives:

```
\knowledge {url={https://en.wikipedia.org/wiki/LaTeX},
           text=LaTeX , color=gray}
| latex
```

yields with the same text

This package has been written for use in L^AT_EX.

The directives `text=` and `color=` have quite obvious meaning. Directives can also control the style using `emphasize`, `boldface`, `italic`, `typewriter` and so on. See Section 5.3 for a complete list of directives.

Variation (synonyms). It happens very often that there are several ways to name a notion, because of capitalized letters, conjugacy, grammar, or simply because it is not explicitly named in the text. There are two ways to resolve this issue. The first is to use the syntax

```
\kl [knowledge]{text}      or      "text@knowledge"
```

the result is that the text ‘*text*’ is displayed, but urls, colors, etc from ‘*knowledge*’ are used.

Another more systematic way to do it is to declare synonyms. This obtained by adding the corresponding lines:

```
\knowledge {url=...}
| Donald Ervin Knuth
| Donald Knuth
| D. Knuth
| Knuth
```

would allow

```
\kl {Knuth}      as well as   \kl {Donald Knuth} ,
or simply "Knuth" as well as   "Donald Knuth" and so on
```

to all point to the same web address. It is even more convenient to use it for nouns that are sometimes in plural form or at the beginning of a sentence. There is also a scoping feature that helps distinguishing the same names in a different context. For instance:

```
\knowledge {url=https://en.wikipedia.org/wiki/Group_(mathematics)}
| group
| groups
| Groups
| group morphism
| group morphisms
| Group morphisms
| morphism@GROUP
| morphisms@GROUP
| Morphisms@GROUP
```

makes it possible to use the notions in many contexts:

```
"Groups" form a category when equipped with "group morphisms".
Consider now some "morphism@GROUP"  $\alpha$  ...
```

In the above code all quoted parts send the reader to the same wikipedia page about groups. Note in particular that "morphism@GROUP" simply displays 'morphism' at compilation. The GROUP part is a [scope](#). In another part of the document, "morphism@RING" can be safely used whereas using an generic "morphism" would become quickly unmanageable.

2.2 Linking inside a document

The problem 2 *I am writing a scientific document with many different definitions, typically a journal article, a PhD thesis², or a book.*

I would like all the notions to be linked inside the document for being able in one click, whenever something is used, to jump to its definition. I also want to easily write an index. However, I do not want it to be a hassle when writing.

A solution is as follows. First load the [knowledge](#) package in the preamble:

```
\usepackage [xcolor,hyperref,notion,quotation]{knowledge}
```

with suitable options: [hyperref](#) for links, [xcolor](#) for colors (if required, but always advised), [quotation](#) for using the [quotation notation](#) and [notion](#) for automatic configuration of the [notion directive](#).

Then write the document using [\intro](#) (or "["...](#)" if [quotation](#) is activated) when a notion is defined/introduced, and [\kl](#) (or "["...](#)" if [quotation](#) is activated) when it is used. For instance:

```
\AP A \intro {semigroup} is an ordered pair  $(S,\cdot)$  in
which  $S$  is a set and  $\cdot$  is an associative binary operator
over  $S$ . A \intro {semigroup morphism} is [...]
[...]
Consider a \kl {semigroup morphism} between \kl {semigroups}  $S$ 
and  $T$ . [...]
```

or when the [quotation notation](#) is activated:

```
\AP A "semigroup" is an ordered pair  $(S,\cdot)$  in which  $S$ 
is a set and  $\cdot$  is an associative binary operator over  $S$ .
\AP A "semigroup morphism" is [...]
[...]
Consider a "semigroup morphism" between "semigroups"  $S$  and  $T$ .
[...]
```

This yields

A [semigroup](#) is an ordered pair (S,\cdot) in which S is a set and \cdot is an associative binary operator over S . A [semigroup morphism](#) is [...]
[...]
Consider a [semigroup morphism](#) between [semigroups](#) S and T . [...]

Hint. Using an [\AP](#) command is strongly advised, and allows to control more precisely where the target of hyperreferences is: at the beginning of a paragraph is better than the beginning of the section several pages before...

Note that the [\AP](#) command is made visible thanks to a red corner. This red corner is shifted left in order to fall in the margin. [Undefined knowledges](#) also yield warnings during the compilation. These can be avoided using the [silent](#) option.

Undefined knowledges are displayed in orange the first time, and in brown the subsequent ones (it is an important feature that the compilation does not fail: undefined knowledges should not interfere with the main activity of the writer which is writing). One can now see the list of such problems in the file ‘filename.diagnose’, in particular in the ‘Undefined knowledges’ section:

```
\knowledge{ignore}
| semigroup
| semigroups
| semigroup morphism
```

which means that the strings ‘semigroup’, ‘semigroups’ and ‘semigroup morphism’ are used but unknown knowledges.

To solve this, let us copy these four lines in the preamble³, replacing ignore by the `\notion` directive and separating it into two blocks. We obtain:

```
\knowledge {notion}
| semigroup
| semigroups
\knowledge {notion}
| semigroup morphism
```

This informally means that ‘semigroup’ and ‘semigroups’ are two strings that represent the same notion, and that ‘semigroup morphism’ is a different one.

Note that the `\AP` command is made visible thanks to a red corner.

The result is then (after two compilations):

⌞ A *semigroup* is an ordered pair (S, \cdot) in which S is a set and \cdot is an associative binary operator over S . A *semigroup morphism* is [...] [...]

⌞ Consider a *semigroup morphism* between *semigroups* S and T . [...]

Clicking on ‘semigroups’ now jumps to the place where it was introduced, and very precisely at the location of the red corner depicting the presence of the `\AP`-command before. The same goes for ‘semigroup morphism’. If now one adds the option `electronic` while loading the package, then the red corners disappear as well as the undefined knowledges which become black. When using the option `paper`, the links are still there, but all texts are in black.

A more complete example would look as follows:

```
\knowledge {notion}
| semigroup
| semigroups
| Semigroups
\knowledge {notion}
| semigroup morphism
| semigroup morphisms
| morphism@SG
| morphisms@SG
```

The most interesting part consists of the two last lines. These define ‘morphism’ and ‘morphisms’ to represent the same notion as ‘semigroup morphism’ when

²Reviewers should appreciate...

³It is good practice to use a separate file, something like ‘paper-knowledge.tex’.

in the scope ‘SG’ (a convenient abbreviation chosen by the writer for meaning ‘semigroup’). This is particularly useful for separating morphisms of semigroups from, say, ring morphisms. Now, using `Take a \kl {morphism} ...` would yield an unknown knowledge. However, `Take a \kl (SG){morphism} ...` or `Take a "morphism@SG" ...` would yield

Take a [morphism](#) ...

which is properly linked to a semigroup morphism. One can in particular imagine that `"morphism@RG"` would instead represent ring morphisms.

Sometimes, one does not want to expand the knowledge base because the word is not specific. For instance `These \kl [semigroup]{algebraic objects} [...]`, or `These "algebraic objects@semigroup" [...]`, yields

These [algebraic objects](#) [...]

The term is properly linked to the definition of semigroups.

Another possibility is to refer to the location of the definition, using commands `\kref`, `\kpageref`, and even better their improvements `\kcref`, `\kCref`, etc (when using the `cleveref` package). For instance,

"Semigroups" are introduced in `\kCref {semigroup}`,
Page `\kpageref {semigroup}`.

yields:

[Semigroups](#) are introduced in Section [2.2](#), Page [11](#).

Finally, in particular for large documents, it is good to have an [index](#). For this, one should load the package `makeidx` before `knowledge`. Then use it normally: putting `\makeindex` in the preamble and `\printindex` at the end of the document. Finally, the `knowledge` commands are adapted in a straightforward manner:

```
\knowledge {notion,index=semigroup}
| semigroup
| semigroups
| Semigroups
\knowledge {notion,index=semigroup morphism}
| semigroup morphism
| semigroup morphisms
| morphism@SG
| morphisms@SG
```

Now, the index (after running `makeindex`) contains all entries and references to the use of semigroups and semigroup morphisms.

See Section [3.8.3](#) for more details on making an index.

2.3 Mathematics

The examples above show various techniques for using `knowledges` for enhancing the information associated to terms. In fact, these techniques are not incompatible with mathematics. Some special syntax is even offered:

```
\knowledge{newrobustcmd \closure [1]{\cmdkl {\langl }#1\cmdkl
{\rangle }}}
[...]
```

The closure of X under product is denoted $\langle X \rangle$.

```
[...]
```

Consider now $\langle \{a,b,c\} \rangle$.

would yield:

The closure of X under product is denoted $\langle X \rangle$.

[...]

Consider now $\langle \{a,b,c\} \rangle$.

The problem 3 *But I want more. I want to be able to introduce variables. Even better, I would like to be able to have variables hyperlinking to the place of their introduction, knowing that the same variable name may mean different things depending on the lemma or proof we are in. Hence, I want to properly control the scope of knowledges.*

To be done, this requires to use `scoping`. The principle of `scoping` is that a knowledge can be attached to a particular context. This is particularly true when typesetting mathematics: a variable is meaningful inside a statement, and inside the proof of the statement. Furthermore, the same variable name may reappear elsewhere with a different meaning.

The following code gives an idea of what is possible using `scoping`:

```
\knowledgeconfigureenvironment {theorem,lemma,proof}{}
[...]
```

`\begin {lemma}\label {theorem:main}`

```
  \knowledge {n}{notion}
  For all number  $\langle n \rangle$ , [...]
```

`\end {lemma}`

```
[...]
```

Here $\langle n \rangle$ is an undefined knowledge.

```
[...]
```

`\begin {proof}[Proof of theorem~{theorem:main}]`

```
  \knowledgeimport {theorem:main}
  Inside the proof,  $\langle n \rangle$  is hyperlinked to the theorem...
\end {proof}
```

More on `scoping` can be found in Section 3.5.

The use of `variants of \kl` is also useful for typesetting mathematics. It allows for instance, to implicitly execute the `\knowledge` command at the same time of the introduction. See 3.4.4 for more detail.

3 Usage of the `knowledge` package

3.1 Options and configuration

Options are used to activate some capabilities. Some options have to be used when loading the `knowledge` package, while some others can also be used inside the document thanks to the use of `\knowledgeconfigure`. In this section, we review these *package options*.

3.1.1 Options at package loading

The options that can be used in the optional parameter of `\usepackage` when loading the `knowledge` package belong to the following classes:

Writing mode The `paper`, `electronic` or `composition` modes are possible (`composition` is by default) (see Section 3.1.2 for more details). These modes change several rendering settings.

Other packages some of the options concern the loading and the use of other packages (`hyperref`, `xcolor`, `makeidx`, `imakeidx`, `cleveref`, ...). Note that these package can also be loaded before `knowledge`. This is explained in Section 3.1.3.

Configuration options as used by the command `\knowledgeconfigure` can be used when loading the package.

Scoping The `scope` option makes the package aware at a fine level of the structure of the document (see Section 3.5 for explanations). This provides, for instance, the possibility to define pieces of `knowledge` that are attached to a sections of the document.

Other The `no patch` option prevents the `knowledge` to apply some patches that are convenient by default.

3.1.2 Writing mode

There are three *writing modes* usable when loading the package `knowledge`:

- In `paper` mode, the paper is rendered as for printing: in particular, no informative colors are visible. Hyperlinks are nevertheless present.
- In `electronic` mode, the document has some colors witnessing the existence of the links for the reader to know that clicking is available.
- In `composition` mode (the default), the document has colors helping the writing: undefined knowledges appear explicitly, anchor points are displayed, and so on.

Activating the modes is obtained either at load time using one of:

```
\usepackage [paper]{knowledge}
```

or

```
\usepackage [electronic]{knowledge}
```

or

```
\usepackage [composition]{knowledge}
```

or by setting before loading the variable `\knowledgepackagemode` as in:

```
\def \knowledgepackagemode {paper}
```

The idea is that this can be used in automatic compilation scripts. For instance, using in a terminal:

```
pdflatex "\def \knowledgepackage mode {electronic}\input {file.tex}"
```

would result in compiling ‘file.tex’ using **knowledge** in **electronic mode**.

The following primitives are available to the user for **writing mode**-sensitive configuration:

```
\IfKnowledgePaperModeTF{true code}{false code}
\ifKnowledgePaperMode true code [\ else false code] \ fi
\IfKnowledgeElectronicModeTF{true code}{false code}
\ifKnowledgeElectronicMode true code [\ else false code] \ fi
\IfKnowledgeCompositionModeTF{true code}{false code}
\ifKnowledgeCompositionMode true code [\ else false code] \
fi
```

3.1.3 Automatic loading of other packages

A certain number of **package options** coincide with the loading of other packages. For the moment, the packages that are concerned are **hyperref**, **xcolor**, **makeidx**, and **imakeidx**.

For activating these functionalities, it is sufficient, either to load the package *before* the **knowledge** package, or to name it explicitly as an **option** for **knowledge**. Loading separately the package is convenient for setting options for it. For instance, a typical preamble may look like:

```
\documentclass {article}
\usepackage [svgnames]{xcolor}
\usepackage [draft]{hyperref}
\usepackage [makeidx]{knowledge}
```

Such a sequence will activate the **knowledge** package using the features related to **xcolor** configured with **svgnames** option, to **hyperref** configured with **draft** option, and to **makeidx** with its standard configuration.

In fact, the syntax when a package is loaded as an option of **knowledge** is of the form ‘package=choice’ in which choice can take the following values:

active The package will be loaded, and all the capabilities that it triggers are activated. This is the implicit meaning when nothing more is specified.

inactive The package is not loaded, and no capabilities are activated (even if it had been loaded previously by another **\usepackage** command).

compatibility The package is not loaded. The directives it uses do not cause any error, but have no effect.

auto If the package was loaded before, then the associated capabilities are activated. This is the default behavior when the package is not named while loading.

Currently, the packages that can be loaded are:

hyperref which activates all the (auto)referencing capabilities.

xcolor which activates coloring commands.

makeidx and **imakeidx** for handling the index automatically.

3.1.4 Configuring and `\knowledgeconfigure`

Some part of the configuration can be done outside of the `\usepackage` command that loads the `knowledge` package. This is done using the `\knowledgeconfigure` command:

```
\knowledgeconfigure{configuration directives}
```

Note that by default, the `configuration directives` used by `\knowledgeconfigure` can be used in the optional parameter of `\usepackage` when loading the `knowledge` package, but the converse is not true. *Configuration directives* consists of a comma separated list of elements that can take the following values:

`diagnose bar=` (de)activates the ‘|’-notation in the `diagnose file`. True by default.

`diagnose help=` can be set to true or false. It activates or deactivates the help in the `diagnose file`. True by default.

`diagnose line=` can be set to true or false. It activates or deactivates the line numbering in the `diagnose file`. False by default.

`fix hyperref twocolumn` triggers a hack that solves a known problem that may occur when `hyperref` is used in two-columns mode.

`label scope` enables or disables the redefined `\label` command, which helps automatically define scopes (default is true).

`notion` configures the `notion directive` which is a refined version of `autoref`.

`protect quotation=` is followed by a comma separated list of environments in which the `quotation notation` will be automatically deactivated (surrounded by braces if more than one item in the list).

`protect link and unprotect link` starts and ends respectively a zone in which the `knowledge` package do not create hyperlinks. These can be nested. This is typically useful around, e.g. the table of contents.

`quotation` activates the `quotation notation`, which allows to use "...", "...@..." and "...@...@..." instead of `\kl` commands and "...", "...@..." and "...@...@..." instead of the `\intro` command.

`silent` is a Boolean option which, when activated, switches off all warnings during the compilation, but the recap ones at the end.

`strict` is a Boolean option which, when activated, turns some warnings (for instance when a knowledge is redefined) into errors.

`visible anchor points` is an option that makes visible or invisible the `anchor points` of the `\AP` and `\itemAP` commands. Usually, this is automatically set to true when the `composition mode` is used (the default), and to false when the `paper mode` or the `electronic mode` are used.

3.1.5 Other configuration option

`no patch` deactivates some patches which otherwise are applied automatically.

Currently, the option `hidelinks` and `breaklinks` of the package `hyperref` are automatically applied, unless `no patch` is used while loading the package. Without `hidelinks` the links in the document are surrounded by red or light blue boxes (it depends also on the pdf viewer): while this may be acceptable when links are seldom used, this becomes problematic in combination with the `knowledge` package. Without `breaklinks`, links are not broken as normal text: this may corrupt the appearance of paragraphs, in particular in a multi column context.

3.2 What is a `knowledge`?

A *knowledge* is often informally used in this document. Essentially, it captures what is an elementary concept in the document. Internally, a `knowledge` is identified by three components:

The *knowledge name* is a `TeX` string that has almost no limitation (but being well balanced, and containing no `§`). It is the text entered by the user for defining and using the `knowledge`.

The *scope* which is a simple string identifying where the `knowledge` is usable. Scopes can be created by the user, or automatically generated by the system. For instance, internally, each section will be uniquely named ‘`section-1`’, ‘`section-2`’, and so on (this is invisible for the user). Each `knowledge` is primarily valid in exactly one such *scope*. `Knowledges` defined in the preamble are given the *scope* ‘`document`’.

The *namespace* is a simple string that is used for avoiding clashes. It is most of the time simply ‘`default`’. It is ‘`style`’ for *styles* (that are internally as `knowledges`). It is a possibility available to a developer to, when creating a new set of functionalities, use a different *namespace* for avoiding clashes of names (for instance if one wants a french and an english set of `knowledges` that should not conflict, and would use separate sets of macros). Usually, a normal user does not see *namespaces*.

3.3 The `\knowledge` command and variations

In this section, we describe the main commands that create `knowledges`. The main one is `\knowledge`. It can also be used in combination with `\knowledgedirective`, `\knowledgedestyle` and `\knowledgedefault`.

3.3.1 General description of the `\knowledge` command

The key command for introducing `knowledges` is `\knowledge`. There are two syntaxes. The standard one is:

```
\knowledge{knowledge name}[synonym 1|synonym 2|...]{directives}
```

The second one is the ‘*’-notation*⁴:

⁴This is a non-standard `LaTeX` syntax. The rule is that each `knowledge` appears in a distinguished line that starts with some spaces and a ‘`’`’, and ends at the end of the line. Detecting the end of the line requires to change the catcode of the end of line character; this is not robust for being used in an argument or a macro.

```

\knowledge{directives}
| knowledge name@optional scope
| synonym 1@optional scope
| synonym 2@optional scope
...

```

The *knowledge name* as well as the *synonyms* are plain text strings describing the knowledge. It may contain any combination of symbols, including accents or special characters as long as it well bracketted. This string will be used to fetch the *knowledge*. Note (and this is a standard T_EX behavior) that several consecutive spaces is the same as one or a line feed. In the normal syntax, *synonyms* are given in a ‘|’ separated list, while in the ‘|’-notation each of them has to be in a distinct line. In the ‘|’-notation, an *optional scope* can be given after each *knowledge name/synonym*.

The *directives* consists of ‘key=value’ statements in a comma separated list. There are many *directives*. A list of them can be found in Section 5.3. New ones can be defined using the `\knowledgedirective` command.

The principle of the `\knowledge` command is to introduce a new *knowledge*, ready for being used. However, what it does exactly depends a lot on the situations. First, the *directives* (a comma separated list of ‘key=value’ commands) are parsed, and from it, the namespace and *scope* of the knowledge are determined, and it is decided if it will be defined immediately or postponed to the next compilation phase (using the *kaux file*).

3.3.2 Targeting and the corresponding directives

The `\knowledge` has to decide what to do when defining something. The basic behaviour is as follows.

- If the `\knowledge` command is used in the preamble, then the *knowledge* given as argument is defined immediately (the same effect can be obtained using the *now directive*), and is accessible in the first compilation phase everywhere in the document (one extra phase is nevertheless required if *autoref* or *ref=* directives are used, for the *hyperref* to do its job, or if *scope=* is used). This is the simplest way to use `\knowledge` .
- Otherwise, the *knowledge* is written in an external file (the *jobname.kaux file*), and the *knowledge* will be really usable in the next compilation phase. This is particularly useful in conjunction with the *scope option*: the *knowledge* will have a scope depending on where it is introduced (for instance the document, or a theorem, or a lemma). The same *knowledge name* can then point to different *knowledges* depending on where it is used.
- Exporting (*not implemented*) furthermore writes a document containing a list of `\knowledge` commands giving access to its content. This is to be imported by another document.

The *targeting directives* refine the above defined behaviour:

scope= or ‘@’ in the ‘|’-notation When using a *directive* ‘*scope=name*’ or ‘@name’ in the ‘|’-notation, the scope of the definition can be modified. `\knowledge` will first check if there is an outer *area* of this name (*theorem*,

`section, ...`), that accepts knowledge (only `scope` environments are subject to this unless `\knowledgeconfigureenvironment` is used, or the `scope package option` is used when loading the package). If this is the case, the knowledge will be associated to the corresponding `instance`. For instance, inside a theorem, by default, the scope is the theorem, but adding the directive '`scope=section`', the `knowledge` becomes available in the whole section.

If no scope is found using the above search, an `explicit scope` of the given name is used.

`export=` (not implemented) When using this directive, the knowledge will be (furthermore) written to another file, ready for being used in another document. In particular, the knowledge (in the other document) will point to the present one. The details on how this is supposed to work is to be specified.

`namespace=` Allows to change the `namespace`. In itself, this is useless. It has to be used in conjunction with new forms of `\kl` -like commands.

`now` requires the `knowledge` to be defined immediately. This may save one compilation phase. The drawback is that the `knowledge` cannot be accessed before the `\knowledge` command that has been introduced. It may help for modularity considerations. (for instance a `knowledge` is used inside a proof, it makes no sense to make it available elsewhere, and it is better style to locally define it). This is implicit if the `\knowledge` command happens in the preamble.

`also now` requires the `knowledge` to be defined immediately as well as delayed to the next compilation phase. This is in particular how `auto references` should be handled. See the use of `\knowledgevariant` for more examples.

3.3.3 General directives

We give here the list of *display directives* that are available without loading any sub packages. A certain number of Boolean directives are available without any options. These most of the time are used for typesetting the output. Each of these can be used as '`bool=true`' (or shortly just '`bool`'), '`bool=false`' or '`bool=default`' (that leaves it in the default state, or the one determined by surrounding knowledges). The general boolean *directives* are the following:

`autoref` activates the ability to introduce once, use several times an instance. This is often used for defining links inside a document (see the `hyperref` in Section 3.8.2). This can also be used for constructing the index (see the `makeidx` and in `imakeidx` Section 3.8.3). It can simply be used for referencing the number of the environment or the proper page of introduction (see `\kref` and `\kpageref` commands and their more powerful `cleveref` in Section 3.8.4).

`autorefherr` puts immediately a label at the location of the definition, and makes all `\kl` occurrences of this `knowledge` hyperlink to this location.

`emphasize` forces the text to be emphasized using '`\emph` ',

`italic/up` forces/unforces italic (`up` does nothing in math mode),

`boldface/md` forces/unforces boldface (be it in math or text mode),
`smallcaps` forces small capitals,
`underline` forces the text to be emphasized using ‘`\underline`’ ,
`fbbox` puts a box around the text,
`typewriter` puts in typewriter font (be it in math or text mode),
`ensuretext` guarantees that text mode is used (using the ‘`\text`’ macro, thus in a way consistent with the surrounding style),
`ensuremath` guarantees that math mode is used,
`mathord`, `mathop`, `mathbin`, `mathrel`, `mathopen`, `mathclose`, `mathpunct` yield the corresponding standard T_EX spacing features in math mode,
`mathord` for an ordinary mathematical object,
`mathop` for a large operator (such as \sum , \prod , ...),
`mathbin` for a binary operation (such as $+$, $-$, or \otimes , ...),
`mathrel` for a binary relation (such as $=$, $<$, \leq , ...),
`mathopen` for an opening bracket, parenthesis, ...
`mathclose` for an closing bracket, parenthesis, ...
`mathpunct` for a punctuation symbol.
`lowercase` puts the content in lowercase,
`uppercase` puts the content in uppercase,
`detokenize` detokenizes the content, i.e., instead of executing it provides a string that displays it (this is useful for commands),
`remove space` removes the spaces from the text
`invisible` prevents the rendering of the knowledge.

The non-boolean general `directives` are the following:

`text={text}` will execute the L^AT_EX code ‘text’ instead of the key used for calling `\kl` . For instance, `\knowledge {latex}{text=\LaTeX }` will typeset ‘L^AT_EX’ properly when used. Surrounding braces can be omitted if there are no commas. Be careful when linking to such knowledges, since the substitution of meaning will happen for all the knowledges linking to it, and this may not be the expected behaviour.

`link={knowledge}` will continue searching the for linked `knowledge`. Surrounding braces can be omitted if there are no commas. This directive is often bypassed by the use of the `optional argument` of `\knowledge` defining synonyms or the `synonym` directive.

`link scope={label}` will continue searching in the `scope` identified by the label. Surrounding braces can be omitted if there are no commas. If no `directive link=` is given, then the same key is searched for.

This directive is often bypassed by the use of the `optional argument` of `\knowledge` defining synonyms or the `synonym` directive.

`synonym` defines the knowledge as a link to the previously defined knowledge (in fact, the most recently defined that was not using `synonym`). For instance

```
\knowledge {Leslie Lamport}
  {ref={https://fr.wikipedia.org/wiki/Leslie_Lamport}}
\knowledge {L. Lamport}{synonym}
\knowledge {Lamport}{synonym}
```

results in the two subsequent `knowledge` names to point to the first one.

`style={knowledge style}` will adopt the styling option of the `knowledge style`. Surrounding braces can be omitted if there are no commas.

`wrap=\token` will execute the macro `\token` with as argument the knowledge text before displaying it. For instance, `wrap=\robustdisplay`, (where `\robustdisplay` is a variant of `\tl _to_str:n` removing the trailing space) is used in this document for typesetting the commands.

3.3.4 Knowledge styles and the `\knowledgestyle` command

Styles are formatting pieces of information, as for `knowledges`, but that can be used by other `knowledges`. In some respect, this is very similar to `macro directives` (see below), but the difference lies in that *styles* are dynamically resolved, while `macro directives` are statically resolved. *Styles* in particular offer the access to some configuration features of the system. For instance, changing the `intro style` changes the way the `\intro` command is displayed. See below for some instances.

The central command is `\knowledgestyle`, that has the following syntax:

```
\knowledgestyle*{style name}{directives}
```

The optional star `*` permits to overload an existing style (otherwise, this results in an error). The *directives* follow the same structure as for a normal `\knowledge` command. When defined, a *style* can be used in a `\knowledge` command using the *directives* `'style=style name'` (it will be used when a `\kl` command calls for the `knowledge`) or `'intro style=style name'` (that will be used by `\intro` commands).

A certain number of *default styles* are also offered, that in particular includes *warning styles*. The list is as follows:

`kl` is the default style for macros using `\kl`. It can be modified dynamically using the `'style='` directive.

`kl unknown` and `kl unknown cont` are the default *styles* used when an undefined `knowledge` is met.

`intro` and `intro` is the default style for macros using `\intro`. It can be modified dynamically using the `'intro style='` directive.

`intro unknown` and `intro unknown cont` are the default *styles* used when an undefined `knowledge` is met.

3.3.5 New directives: the `\knowledgedirective` command

When defining `knowledges`, it is often the case that the same sequence of directives are used. *Macro directives* are here for simplifying this situation (see `\knowledgedefault` and `\knowledgestyle`). This is achieved using the `\knowledgedirective` macro:

```
\knowledgedirective*{name}[optional parameter]{directives}
```

Hint. This should not be confused with *styles* which offer another way to control the display.

After execution of the command, ‘name’ becomes a `directive` usable in `\knowledge` commands, that amounts to execute the comma separated list ‘directives’. The newly created `directive` may receive a value, that is accessible as #1 in ‘directives’. By default, it does not allow the redefinition of a directive. This can be forced using the optional *. The ‘optional parameter’ gives a default value. For instance:

```
\knowledgedirective {highlight}[brown]{color={#1},emphasize,md}
[...]
\knowledge {notion A}{highlight}
\knowledge {notion B}{highlight}
\knowledge {notion C}{highlight}
\knowledge {important notion D}{highlight=red}
[...]
```

We shall now see `\kl {notion A}`, `\kl {notion B}`, `\kl {notion C}`, as well as the `\kl {important notion D}`.

yields

We shall now see *notion A*, *notion B*, *notion C*, as well as the *important notion D*.

3.3.6 `\knowledgedestyle` versus `\knowledgedirective`

The two commands `\knowledgedestyle` and `\knowledgedirective` offer ways to systematize the writing of knowledges. These can seem redundant. This is not the case, and for understanding it, it is necessary to understand a bit the way the `\knowledge` command works.

In general when a `\knowledge` (or `\knowledgedestyle`) command is found, the `directives` are parsed and a new internal form of the `\knowledge` command is written in the `kaux` file, that will be executed during the next compilation of the document. In this phase, some first operations are performed. For instance, in an `autoref` directive, an internal label name is constructed.

The postponed command is then executed during the next compilation phase (or immediately if we are in the preamble, or if the `now` directive is used). The execution effectively stores the `knowledge` in the system. This is only at that moment that the `knowledge` becomes available to be used by `\kl` and similar commands.

When a `\kl` command (or similar) is met, it is ‘executed’, and display informations are considered, and in particular `styles` are called.

Somes consequences of this kind of this are as follows:

- `autoref` directives should not be used in the definition of a `style`, since this would mean that there would be one anchor point for all the `knowledges` that use this `style`. This is usually not the kind of behavior that we expect.
- configuring the default displays of the system (such as the `intro style=` in particular) has to be done through the `style` mechanism.
- `styles` use less memory than macros.

3.3.7 Default directives: the `\knowledgedefault` command

It may happen that a sequence of consecutive `\knowledge` commands have to share the same list of *directives*. The *macro directives* can help solving this issue. The *default directives* also go in this direction, using the `\knowledgedefault` command:

```
\knowledgedefault*{directives}
```

When such a command is applied, then from that point, all `\knowledge` commands will use the given *directives* as default. This will stop when another `\knowledgedefault` command is met or the current group is closed. The optional star does not reset the *default directives* but simply add new ones.

3.4 The `\kl` command and variants

The `\kl` command is used for displaying a knowledge in the text, while using the data associated to it (introducing a target link, linking to its definition, linking to an outer document, changing color, putting an entry in the index, etc).

3.4.1 The standard syntax

Hint. Note that the `\kl` command can often be replaced by the `"..."` notation, activated by the *quotation* option.

The `\kl` command has one of the following syntaxes:

```
\kl(optional scope)[optional knowledge name]{text}  
or \kl[optional knowledge name](optional scope){text} .
```

Its meaning is to search for the ‘optional knowledge name’ if present, or for ‘text’ otherwise. How this is exactly performed depends on the presence of the *optional scope*. The search process is as follows:

- if an *optional scope* is given, the *knowledge* is searched in the corresponding scope.
- otherwise, the *stack of visible scope instances* is processed through (starting from the inner most) until a *knowledge* of name ‘knowledge name’ (or ‘text’ if not optional *knowledge name* has been given), of namespace ‘default’, and this *scope* is found.

If the ‘knowledge name/text’ has not been found, the *style* ‘`kl unknown`’ (or similar *styles*, as defined by the `unknown style=` or `unknown style cont=`) is used, and the text displayed.

- Otherwise, the *knowledge* is executed. If it is a `link=` or `synonym` defined *knowledge*, the link is followed, and the process continues.

- Finally, all the definitions involved in the *knowledge* are processed, following a `style=` if defined, the *knowledge* is updated (essentially incrementing the counter of use), and the *knowledge* is displayed.

This general mechanism is used also by other commands that are *variants* of `\kl` such as `\intro` , `\reintro` , `\kref` , `\kpageref` , and so on.

3.4.2 The *quotation* notation

When activated, the *quotation* mode activates shorthand notations for the `\kl` and `\intro` macros. Possible syntaxes are as follows:

`"text"` uses the `knowledge` pointed to by ‘text’. Equivalent to `\kl {text}`.

`"text@knowledge"` uses the `knowledge` pointed to by ‘knowledge’ to display ‘text’. Equivalent to `\kl [knowledge]{text}`.

`"text@scope"` uses the `knowledge` pointed to by ‘text’ in `scope` ‘scope’ to display ‘text’. Equivalent to `\kl (scope){text}`.

`"text@knowledge@scope"` uses the `knowledge` pointed to by ‘knowledge in `scope`’ to display ‘text’. Equivalent to `\kl [knowledge](scope){text}`.

`"text"` introduces the `knowledge` pointed to by ‘text’. Equivalent to `\intro {text}`.

`"text@knowledge"` introduces the `knowledge` pointed to by ‘knowledge while displaying ‘text’. Equivalent to `\intro [knowledge]{text}`.

`"text@scope"` introduces the `knowledge` pointed by ‘text’ in `scope` ‘scope’. Equivalent to `\intro (scope){text}`.

`"text@knowledge@scope"` introduces the `knowledge` pointed to by ‘knowledge in `scope`’ while displaying ‘text’. Equivalent to `\intro [knowledge](scope){text}`.

Activating the `quotation notation` is obtained using:

```
\knowledgeconfigure {quotation} ,
```

and deactivating it is obtained using:

```
\knowledgeconfigure {quotation=false} .
```

It can also be activated while loading the package.

It is sometimes the case that some packages do use the quote symbol, usually in some environment (this is the case of the `tikzcd` environment). The `knowledge` package can be configured to deactivate always the `quotation notation` when entering the environment. This is obtained using the configuration option `protect quotation=` followed by a list of environments to be protected:

```
\knowledgeconfigure {protect quotation={env1,env2,...}}
```

Note that the braces surrounding the list of environments can be omitted if the list contains only one item.

There are nevertheless some situations in which one would prefer to use the original `\kl` notation:

- When nesting of `knowledges` is involved, or the `knowledge` includes the symbol "`"`,
- when `quotation` is deactivated (or not activated) because of a conflict
- in particular, this should be avoided in macros, in particular for the math mode, since these may be used one day or another in a `tikzcd` or similar environment for instance.

3.4.3 Variants

Several variants of the `\kl` -command are defined by default. The syntax for their use is the same (including `scopes`, ...). The following variants are linked to the `autoref` directive:

- `\intro` is used to introduce a knowledge.
- `\reintro` displays like for introducing the knowledge, but does not do it.
- `\rekl` behaves like `\kl` acts as a modifier and transforms `\intro` into `\reintro` .
- `\phantomintro` performs the introduction, but does display anything.
- `\kref` displays the number of the environment in which the introduction took place.
- `\kpageref` displays the number of the page in which the knowledge introduction took place.
- `\kcref`, `\kCref`, `\kcpageref`, `\kCpageref`, `\knamecref`, `\knameCref`, `\kn-amerefs`, and `\knameCrefs`, display informations about the place of introduction of a knowledge, following the semantics of the corresponding functions in the `cleveref` package (see Section 3.8.4).

3.4.4 Defining your own variants

It may happen for several reasons that we may want to define new variants of the `\kl` macros, that essentially perform the same task, but are configured differently. Typical examples may be:

- several sets of `knowledges` may intersect but should use different `namespace`,
- some `knowledges` involve macros and for this reason should be non-expanded even if the `\knowledge` command is not met,
- the `\knowledge` command should be called implicitly,
- activate or deactivate the warnings or messages in the `diagnose file`.

In fact, several macros in this document are instantiation of this mechanism. This is the case for for instance for `\intro` , `\phantomintro` , `\reintro` or `\mathkl` etc...

The macro for introducing a new *variant of* `\kl` is:

```
\knowledgenewvariant\variant {variant directives}
```

and is similar to the one for modifying the behavior of a *variant of* `\kl` :

```
\knowledgesetvariant\variant {variant directives} .
```

These command define/modify a/the macro `\variant` that uses the same syntax as `\kl` . The *variant directives* consist of a comma separated list of `directives` as follows:

`namespace=namespace` declares in which `namespace` (a string) the knowledges are to be searched. This means in particular that the `\knowledge` concerned should be defined using the the proper `namespace=` directive.

`default style=, unknown style=, unknown style cont={list of style names}` declares the style name to be used (1) by default when the `knowledge` is found, (2) when it is not found for the first time, and (3) the subsequent times.

`style directive={directive names list}` defines a list (comma separated) of directives that can be used in a `\knowledge` command to modify the aspect (for instance, the `\intro` behavior is modified by the `intro style=` directive, while the `\kl` command is configured using the `style=` directive). If the `directives` do not exist, these are created.

`auto knowledge={directives}` declares that the use of `\variant` should automatically execute a `\knowledge` command, and what should be the directives it uses. See examples below.

`unknown warning=true/false` activates or deactivates the warnings when a `knowledge` is not found (for instance, these are deactivated in `paper mode`). True by default.

`unknown diagnose=true/false` activates or deactivates the corresponding messages in the `diagnose file`. True by default.

`suggestion={directives}` configures the `directives` to be suggested in the `diagnose file` when the `knowledge` is unknown.

`PDF string={code}` gives a substitute text for `hyperref` to use for producing the bookmarks. This code has to be expandable. The code may use three parameters; #1 is the main text of the command, #2 is the optional parameter, and #3 is the scope. The macro `\IfNoValueTF` of the package `xparse` can be used to test if the second and third arguments are present. By default, the code is {#1}. Note that the star syntax cannot be used in this context. If the expected result cannot be achieved using this directive, the less convenient macro `\texorpdfstring` of the `hyperref` package should be used.

`display code=` can be used to shortcut the standard code for display. See the code for more details. Typical examples are the `variants \kref`, `\kpageref` and so on, which do not display the knowledge itself, but instead treat the underlying label associated.

The second feature is to use `modifiers`. These correspond to the stored version of the command. For instance, one expects ‘`\intro *\kl`’ to reduce to ‘`\intro`’. For this, one has to declare explicitly the reduction using:

```
\knowledgevariantmodifier{stored sequence}\variant ,
```

in which the *stored sequence* is of the form ‘`variant1*variant2*...*variantk`’.

This sequence is declared to reduce to `\variant`. For instance, `\knowledgevariantmodifier {\intro` declares ‘`\intro *\kl`’ to reduce to ‘`\intro`’.

3.4.5 Examples of `variants`

The best way for introducing new variants is to look at examples. We provide two of them now. the first one is the configuration of the `\kl` and `\intro` commands as defined in the `package`. The second one is the code used in this documentation for displaying macros, defining the macros `\cs` and `\csintro`.

The configuration of `\kl` and `\intro` It is also interesting to see this code since it gives more ideas on how to modify the standard behaviour of these commands correctly.

```
\knowledgestyle {autoref link}{autoref link}
\knowledgestyle {autoref target}{autoref target}
\knowledgestyle {invisible}{invisible}
\knowledgevariant \kl {
  namespace=default,
  default style={kl,autoref link},
  unknown style= kl unknown,
  unknown style cont= kl unknown cont,
  style directive= style
}
\knowledgevariant \intro {
  namespace= default,
  default style= {intro,autoref target},
  unknown style= intro unknown,
  unknown style cont= intro unknown cont,
  style directive= intro style
}
\knowledgevariantmodifier {\intro *\kl }\intro
```

Note that `\reintro` and `\phantomintro` are defined using similar code.

Displaying control sequences The second code example is used in this document (the documentation of the package) and consists of two macros `\cs` and `\csintro` which have the following semantics:

- these correspond, and have the same syntax as, `\kl` and `\intro` respectively,
- these are used to display control sequences without executing them,
- if `\csintro {\command }` is used, then it displays `\command` in color blue, and every occurrences of `\cs {\command }` yields `\command` being displayed in dark blue and linking to the introduction point,
- if `\csintro {\command }` is never used, then `\cs {\command }` simply displays `\command` in black,
- no `\knowledge` command is necessary, and no warning is issued if a command is unknown.

```

\knowledgestyle {cs}
  {detokenize,remove space,typewriter,up,md,color=NavyBlue}
\knowledgestyle {cs unknown}
  {detokenize,remove space,typewriter,up,md,color=black}
\knowledgevariant \cs {
  namespace=cs,
  default style={autoref link,cs},
  unknown style=cs unknown,
  unknown style cont=cs unknown,
  unknown warning=false,
  unknown diagnose=false,
  suggestion=cs
}
\knowledgestyle {csintro}
  {detokenize,remove space,typewriter,up,md,color=blue}
\knowledgestyle {csintro unknown}
  {detokenize,remove space,typewriter,up,md,color=black}
\knowledgevariant \csintro {
  namespace=cs,
  auto knowledge={autoref,scope=document,also now},
  default style={autoref target,csintro},
  unknown style=csintro unknown,
  unknown style cont=csintro unknown,
}
\knowledgevariantmodifier {\intro *\cs }\csintro
\knowledgevariantmodifier {\csintro *\cs }\csintro

```

Several things can be noted about this code:

- the directives `detokenize` and `remove space` prevent the execution of the argument, and instead display its name, this is important since the argument is a control sequence,
- the directives `typewriter`, `up` and `md` give a uniform aspect (no italic, no boldface) to the result in all contexts,
- the `namespace` is set to be different from the default one, avoiding possible clashes with `\kl`,
- when a `\csintro` command is met, the corresponding `\knowledge` command is automatically issued, in particular with ‘`scope=document`’ for guaranteeing the visibility of each command everywhere in the document,
- the `also now` directive is necessary for the compilation to (possibly) stabilize in two iterations, since it uses the proper `\label` already at the first iteration (without `also now`, it would be performed on the second one only, and with just `now`, it would be visible only by the uses after the introduction).
- warnings and diagnose pieces of information are explicitly eliminated.

3.5 Scoping

3.5.1 Principles of scoping

When writing long documents, one often wants `knowledges` to be isolated in some subparts. For instance, one may want a temporary definition in a proof to not

leak elsewhere in the document where the same term could be used with a different meaning. Some definitions may be only meaningful in, say, the current section/part.

Two separate things have to be understood: how to define `knowledge` in a given `scope` (and create `scopes`), and how to access `knowledge` from a given `scope`.

Accessing knowledge attached to a given scope This can be done directly either using the parenthesis notations of `\kl` and the second `@` of the `quotation` notation:

```
\kl (scope name){knowledge}    or    \kl (scope name)[knowledge]{text}
      "knowledge@@scope"        or        "text@knowledge@scope"
```

It works also for `\intro` and with double quotes.

Another option is to import the scope locally, using:

```
\knowledgeimport {scope name 1,scope name 2,...}
```

After this command, the knowledges will be searched automatically in the imported scopes. The import stops at the end of the current scoping environment.

Attaching knowledge to a given scope This can be done directly using the `scope= directive`, for instance in:

```
\knowledge {knowledge}{scope=scope name,directives}
```

or, this is obtained using the `'|'-notation` using `'@'` :

```
\knowledge {directives}
| knowledge@scope name 1
| synonym@scope name 2
:
:
```

The other possibility is to define a knowledge inside a `scope` environment:

```
\begin {scope}\label {label}
  \knowledge {knowledge 1}{directives}
  :
\end {scope}
```

In such a code, the knowledge defined is automatically visible in the environment, and from outside, using the scope name `label`. Indeed, the `\label` is overloaded for doing automatically a `\knowledgescope` command.

In fact, it is possible to do even more: other environments can be defined to behave like `scope`.

3.5.2 Scoping by examples

Explicit scoping consists in attaching a precise scope name to a `knowledge` using the `scope= directive`:

```
\knowledge {thing}{scope=s1,color=red}
\knowledge {thing}{scope=s2,color=green}
```

Here, "thing" and `\kl {thing}` are unknown.
 But "thing@s1" and `\kl (s1){thing}` are in red,
 and "thing@s2" and `\kl (s2){thing}` are in green.

The `|`-notation can also be used for [explicit scoping](#). This is convenient, in particular for having synonyms in different scopes:

```
\knowledge {color=red}
| abelian group
| abelian groups
| Abelian groups
| group@abelian
| groups@abelian
| Groups@abelian
```

Here, general "groups" are not defined but "groups@@abelian" are, and correspond to "abelian groups".

```
\begin {scope}\knowledgeimport {abelian}
  All "groups" here are abelian.
\end {scope}
```

[Scopes](#) can also be attached to areas in the code. It is convenient to use the usual `\label` command to name them (though, in practice, two different naming spaces are used).

```
% We declare first in the preamble the environments that can have
% knowledges attached to them.
\knowledgeconfigureenvironment {theorem,lemma,proof}{}

% and now in the main body of the document.
\begin {theorem}\label {theorem:main}
  \knowledge {rabbit}[rabbits]{notion}
  In every hat, there is a \kl {rabbit},
  \AP in which a \intro {rabbit} is a small animal with long
  ears.
\end {theorem}
Here a "rabbit" is an unknown knowledge.
But "rabbits@@theorem:main" point to Theorem \ref {theorem:main}.
\begin {proof}\knowledgeimport {theorem:main}
  Now, "rabbit" is hyperlinked to Theorem \ref {theorem:main}.
\end {proof}
```

3.5.3 What is the structure of [scopes](#) in a document

To start with, one needs to understand what are the possible [scopes](#). Scopes are aggregation of zones in the document.

- By default, all the body of the document belongs to a [scope](#) called 'document'. The user can open new scopes using the [scope](#) environment:

```

\begin {scope}
  \knowledge {local notion}{color=green}
  Here is a \kl {local notion} that appears in green.
\end {scope}
But here the \kl {local notion} is undefined.

```

Note that scoping is independent from the grouping mechanism of L^AT_EX.

The user can also declare environments such as `lemma`, `theorem`, `remark` or `proof` to behave like `scope`. This is achieved using using `\knowledgeconfigureenvironment` command.

- The use of the `scope configuration option` goes one step further, and attaches `scopes` to sections, subsections, itemize, items, and so on. But be cautious, this feature, though working, may cause some compiling document to not compile anymore if some weird (and unnatural) nesting of scopes are used (this is the case for instance when using `\bibitem` and `\thebibliography`, and this has to be corrected by hand).

3.5.4 How is chosen the `scope` of a `knowledge`?

In general, when a `\knowledge` command is used, the system tries to figure out what should be its `scope`:

- If the command occurs in the preamble, then the default `scope` will be ‘document’.
- Otherwise, the information is searched for in the *stack of visible scope instances* which means that the `knowledge` will be defined at the level of the innermost surrounding scope that ‘attracts knowledges’. If the `scope option` is not activated (and the user did not perform its own configuration), this is the inner most `scope` environment (or similar environment if `\knowledgeconfigureenvironment` has been used), or ‘document’ if the declaration is not in the scope. If the `scope option` is used, this will be the innermost lemma, proof, or theorem in the context.
- This default behavior can be modified using the `scope=` directive. The `scope=` directive can be followed with a scope level, such as ‘section’, ‘subsection’, ‘chapter’ or ‘itemize’ (in particular in combination with the `scope option`), that will be looked for in the current context and will receive the `knowledge`. The `directive` can also be followed by a label name, and the active scope at the moment of this label will be used.

The following code (that requires the `scope option` for being functional) should be self explanatory:

```

\section {First section}
\label {section:first}
\knowledge {one}{scope=section,color=green}
\knowledge {two}{scope=some label,color=green}

\begin {scope}\label {some label}
  Here \kl {one} and \kl {two} are defined.
\end {scope}
Here \kl {one} is defined but \kl {two} isn't.

\section {Second section}
Here neither \kl {one} nor \kl {two} is defined. However,
I can still use them using \kl (section:first){one} and \kl
(some label){two} (or "one@@section:first" and "two@@some
label", or using the \knowledgeimport {section:first}).

```

3.5.5 Naming scopes: the `\knowledgeimport`, `\knowledgescope` and `\label` commands

It is often the case in a text, that one has to locally break the nesting structure of a document, and refer to a object local in an environment. For instance, a comment may refer to a variables/concept that has been locally used in the proof. The `\knowledge` provides suitable mechanisms for complex referencing of scopes. Let us explain this through an example:

```

% We declare first in the preamble the environments that use
knowledge.
\knowledgeconfigureenvironment {definition}{knowledge=attracts}
[...]
\begin {definition}\label {somewhere}
  \knowledge {something}{notion}
  Here, \intro {something} is a notion internal to the definition.
\end {definition}

```

Note here that what is important is the location of the `\knowledge` command, irrespective of the location of the `\intro` command.

The `\label` command is used to name the `scope`. In fact, the real command is

```
\knowledgescope{scope name}
```

which associates a scope name to the surrounding environment (providing it has been declared possible to do it using `\knowledgeconfigureenvironment`). The standard \LaTeX command `\label` is overloaded and performs implicitly a call to `\knowledgescope` (this behavior can be deactivated/reactivated using the `\label scope={true,false}` directive). The result is that the same string of characters can be used in order to name the scope, and at the same time is used as a standard \LaTeX label.

Something important is missing so far: one rapidly wants to access to `\knowledge`s that do not exist in the current `scope`. For instance, a notion is used in a section of a document, and one would like to refer to it in the introduction. Another case is that of a notion or a mathematic variable that is introduced in the statement

of a theorem, and should be accessible inside the proof. There are essentially two ways to access such distant **knowledges**: either use the `\kl (label){text}` command (or the equivalent `"...@...@..."` notation), or use the `\knowledgeimport` command. We describe the second possibility now. The syntax is:

```
\knowledgeimport{label}
```

The result is that the **knowledges** in the **scope** identified by the label are now accessible until the closure of the current **scope**.

For instance:

```
\knowledgeconfigureenvironment {theorem,proof}{  
[...]  
\begin {theorem}\label {theorem:1}  
  \knowledge \alpha{autoref,color=red}  
  Let $\intro \alpha$ be an integer [...]  
\end {theorem}  
[...]  
Here $\kl \alpha$ is unknown.  
[...]  
\begin {proof}  
  \knowledgeimport {theorem:1}  
  But now $\kl \alpha$ points to its definition.  
\end {proof}
```

3.5.6 Managing scoping environments

The user can also declare an environment to behave like **scope** using the command `\knowledgeconfigureenvironment`, as well as adapt some of its characteristics using **scope directives**.

```
\knowledgeconfigureenvironment{environments}{scope directives}
```

For instance:

```
\knowledgeconfigureenvironment {lemma,theorem,fact,proof}  
  {knowledge=attracts}
```

will induce the corresponding environments to have internal knowledges.

Most of the time, it is not necessary to use **scope directives**.

Remark 1 *Note that (in the current implementation) it is necessary to use the commands `\begin` and `\end`. Hence `\proof ...\endproof` would not trigger a scoping environment while `\begin {proof}...\end {proof}` would.*

The **scope directives** are low level and advanced features. These should not be used in general. The list is the following:

scope=true/false tells whether an environment should induce a scope. For the moment, this is not used (as soon as configured, it always behaves like a scope).

label=none/accepts tells whether a `\label` command can refer to an **instance** of this **area**,

`environment=true/false` should be set to true if the scope has to be opened whenever an environment of same name is opened using the `\begin` and `\end` commands of L^AT_EX.

`autoclose=true/false` means that the closure is triggered by another event (closure of another enclosing instance, or pushing of an area that requires its closure). It should be true for L^AT_EX environments, and false when configuring, e.g., `\section` opens a `scope` and its closure should be automatic when another section begins.

`parents={area1,area2,...}` takes a comma separated list of areas that are allowed as parent. For opening the area, some enclosing instances may be automatically closed for reaching such a parent (if their `autoclose=directive` is set to true). For instance, the parent of a subsection should be a section: Hence when a subsection begins, the currently opened subsection and subsubsection are closed.

`push code={code}` defines some code to be executed when the area is pushed (each time, these are added).

`pop code={code}` defines some code to be executed when the area is popped (added too).

`occurrences=once/multiple/recursive` can be one of ‘once’ if the area can only have one instance in the document, ‘multiple’ if there can be several instances, but not nested, and ‘recursive’, if there is no restriction.

`forces=area` requires a specific area as an ancestor of this area. This ancestor is implicitly pushed if necessary.

3.6 Error handling

By default, the `knowledge` package tries to not stop the compilation unless a serious problem has been found. In particular, it is possible to write an entire document using `\intro` and `\kl` commands or the `quotation notation` without ever introducing a `knowledge`, and only in the end provide this information. This is a feature: as opposed to normal macros, not defining a knowledge should not stop the real work, which is the writing of the document.

Very often there is some `undefined knowledge`. Such `knowledges` are displayed using the `kl unknown` and `kl unknown cont` styles when issued by `\kl` ; using `intro unknown` and `intro unknown cont` when issued by `\intro` . The detail of the problems are then gathered in the `diagnose file`.

3.7 The `diagnose file`

The *diagnose file* is a file that is created when the `knowledge` package is used (note that another file, *jobname.kaux* is also created by the `knowledge` package, for internal use). It enormously eases the use of the package, and it is important to look into it when finalizing a document. It gathers a certain number of informations, that can be warning, code to be used, or simply information. This file has the name of the tex document with the extension `.diagnose`. Its content is divided into clearly identified parts. Depending on the used options, some of these parts may appear or disappear.

Undefined knowledges in this section are listed all the knowledges that have been unsuccessfully searched for. These are given in lines either of the form

```

\knowledge {suggested directives}
|   undefined knowledge/@scope/
:

```

or of the form

```

\knowledge {undefined knowledge}{suggested directives}

```

Switching from one mode to another is obtained using the `configuration directive` `diagnose bar={true,false}` (default is true). The intent is that copying the content of this section to the document itself will solve all problems of `undefined knowledges`. It is an efficient way, when one has written a document without caring so much about knowledges to copy the content of this section, and then modify it/reorganize it, in order to suit ones purposes. By default, no suggestion is offered, or `notion` is suggested if the `notion` directive has been used. Suggestion can be automatically configured using the `suggestion= directive` of the macros `\knowledgevariant` and `\knowledgevariant` .

For instance, using:

```

\knowledgevariant \kl {notion}

```

the `directive notion` is suggested for more directly copying the content.

Autoref not introduced This section lists all `knowledges` that were declared using the `autoref` directive (this can be the case indirectly using, e.g. `notion`), but have not been introduced in the document . When a document reaches its final states, this section should be empty. Usually, one should add the corresponding `\intro` or `\phantomintro` command somewhere in the text.

Autoref introduced twice In this section, all `knowledges` that were declared using the `autoref` directive and introduced using `\intro` or `\nointro` more than once are listed. When a document reaches its final states, this section should be empty. Consider using `\kl` or `\reintro` for solving the problem. Note that this may be caused by an `\intro` used in some title (and repeated in the table of contents).

By default, the `diagnose` file does not give the file and the line of the messages. This can be activated using the `diagnose line` boolean option:

```

\knowledgeconfigure {diagnose line=true}

```

By default, the `diagnose` file gives suggestions to be used with the normal `\knowledge` syntax. A ‘|’-notation suggestion can be activated using:

```

\knowledgeconfigure {diagnose bar=true}

```

3.8 Other packages

3.8.1 The `xcolor` option

The `xcolor` option is used if one wants to change colors. It is good to always load it since it also triggers coloring for debugging. It triggers colors in the `warning` styles that can be useful in debugging. It also offers two new directives:

`color=` where in ‘`color=name`’, name is a color description following the syntax of the `xcolor` package.

`cyclic color` similar to ‘`color=`’, but selects the color automatically in a cyclic list of colors. This can be convenient for debugging. The cyclic list of colors can be chosen using the configuration directive `cyclic colors=`:

```
\knowledgeconfigure {cyclic colors={color1,color2,...}}
```

Note that no spaces are allowed between colors, and that at least two colors are required.

The default cyclic color list is:

```
{red,green,blue,cyan,magenta,yellow,gray,brown,lime,
olive,orange,pink,purple,teal,violet}
```

`colorbox=` surrounds the text with a colorbox of given color (following the syntax of the `xcolor` package).

Loading the package before is necessary for changing the options of the `xcolor` package (for instance for using `svgnames`).

3.8.2 The `hyperref` option

Hint. The `hyperref` surrounds by default links by boxes that are graphically heavy (this are visible in some viewers, and not in some others): this is automatically deactivated when loading the `knowledge` package. Such automatic behaviour can be avoided using the `no patch` option at loading.

Activating the `hyperref` option The `hyperref` option loads the `hyperref` and triggers a certain number of link-related features. This is done either by the command:

```
\usepackage [hyperref]{knowledge}
```

or by loading the `hyperref` before the `knowledge` package.

The directives activated by the package are:

`url=` for hyperlinking to an external document

`ref=` for hyperlinking inside document

`protect link` it a boolean for protecting from the creation of nested hyperlinks,

`autoref` for relating objects with their definition

`autorefhref` similar, and used implicitly for math

The package comes also with the configuration directive `hyperlinks=` which is a boolean deactivates or reactivates the links.

Functionnalities triggered by the `hyperref` option

Hint. You may have to use `\~` instead of `~` in url’s addresses.

`ref= {label}` puts an hyperlink pointing toward a label inside the document (the braces can be omitted when there is no comma).

`protect link` disables the inside hyperlinks,

Hint. It is usually easier to use the ‘`notion`’ directive than simply the `autoref` directive. Its use it already configured.

`url= {url address}` puts an hyperlink to an (external) url (the braces can be omitted when there is no comma).

The `\autoref` directive The `\autoref` directive is among the most useful offered by the `knowledge` package. It is very often used indirectly through directives like `\notion`. When set, the `knowledge` should be used with both `\intro` (exactly once) – or the `"..."` and `"...@..."` notations (and variants) if `\quotation` is active – and `\kl` (possibly several times) – or the `"..."` notation if `\quotation` is active. The use of `\kl` will hyperlink to the location of the `\intro`. The syntax of `\intro` is the same as for `\kl`:

```
\intro [optional knowledge name]{knowledge name}
```

See `\AP` below for improving the result.

A typical use looks as follows:

Hint. Though the `\intro` command can be used in the title of, e.g. sections, without any errors, this may cause a warning when a `table of contents` is used: the command is executed twice, once in the `table of contents`, and once in the document itself.

```
\knowledge {house}[Houses|houses]{\autoref}
[...]\begin {document}
[...]\AP
In this document, we will see the very important notion of
"houses".
[...]\AP
Let us define a "house" to be a building that functions as a
home.
[...]\end {document}
yields
```

```
[...]
In this document, we will see the very important notion
of houses.
[...]
Let us define a house to be a building that functions as
a home.
[...]
```

The variant `\intro*` makes the next `\kl` command behave like `\intro`. This is useful in particular in math mode:

```
\newcommand \monoid{\kl [\monoid]{\mathcal M}}
\knowledge \monoid{\autoref}
[...]\AP
Let now  $\intro * \monoid$  be a monoid.
[...]\AP
Remember now who is  $\monoid$ .
```

Hint. This does not work in `align*` and similar environments. Section 4.5 gives some solutions.

```
Let now  $\mathcal{M}$  be a monoid.
[...]
Remember now who is  $\mathcal{M}$ .
```

The `\phantomintro` version:

```
\phantomintro (optional label){knowledge}
```

takes a `knowledge`, and introduces it at the current location, without displaying anything. This behaves like an invisible intro, i.e., essentially an abbreviation for `\intro [knowledge]{}`. This can be used as a workaround in environment like `align*` that do not allow the use of labels (see Section 4.5).

The `\nointro` command:

```
\nointro {knowledge}
```

does not display anything and silently prevents the knowledge from issuing warnings because it is not introduced.

The `\reintro` command:

```
\reintro [optional knowledge]{knowledge}
```

is displayed as for `\intro`, but without being an anchor for hyperlinks, and without counting as a real `\intro`. It is used if there are for some reason several places that should look like an introduction (typically in the same paragraph), but count as a single target. There is a variant `\reintro*` that makes the next `\kl` command behave like a `\reintro` (similar to `\intro *` with respect to `\intro`).

`Knowledges` that use this directive can be parameterized by modifying the style `intro`.

For modifying the display of `knowledges` introduced by `\intro`, there are new directives:

`intro style=` that takes the name of a `style` as argument. This `style` will be used when the `knowledge` is used in a `\intro` or `\reintro` command.

`autoref target` declares the knowledge to be the target of the `autoref` (this is implicit when using `\intro`).

`autoref link` requires a link to the target of the `autoref` to be produced (this is implicit when using `\kl`).

See the use of `\knowledgesetvariant` for examples of configuration.

The `autorefh ere` directive The `autorefh ere` directive silently introduces an anchor point at the location of the `\knowledge` command invoking it. Uses of `\kl` commands will be hyperlinked to this location.

In some sense, an `autorefh ere` directive can be understood as the sequence of a `autoref` directive that would be immediately followed by the corresponding `\intro` command. This is a bit better since using `autoref` in the body of the document requires three phases of compilation (two only if in the preamble). However, the `autorefh ere` directive does only require two (as for normal labels).

In fact, this `autorefh ere` directive is what is used underneath when introducing mathematical variables, and should be used for implementing similar behaviors.

Using anchor points The directives `autoref` and `autorefhere` use underneath the `hyperref` package. This means that it puts a label at the place of the `\intro` command, and then points to it. However, the semantics in this case, is to jump to the beginning of the surrounding ‘region’. If the `\intro` happens in a ‘section’ (but not inside a theorem-like environment) then the `\kl` command will point at the beginning of the section, possibly 10 pages above the definition itself.

The standard solution in the `hyperref` package is to use the `\phantomsection`. This means defining *anchor points* in the document that will be the target of hyperlinks.

We offer here new commands for helping using this feature:

`\AP` declares an *anchor point* at the left of the current column, at the height of the current line. If the configuration option `visible anchor points` is set (and this is the case by default), a mark will show the precise location of the target. Be careful: it does not work in some situations, like for instance inside the optional argument of an `\item` command (but this is ok elsewhere in an itemize environment), or inside a some macros in mathmode (e.g. fractions).

In the particular case of `\item`, one should use instead:

`\itemAP` Similar to `\AP`, but to be used instead of an `\item`.

Usually putting an `\AP` (a standard command of the `hyperref`) at the beginning of every paragraph, and replacing `\item` by `\itemAP` in itemize-like environments is most of the time good and safe option.

For instance:

```
\AP
In order to describe what is a \kl {monoid}, let us us first define
a \intro {product} to be an associative binary operator, and a
\intro {unit} to be [...]

\begin {description}
\itemAP [A \intro {semigroup}] is a set equipped with a \kl
{product}.
\itemAP [A \intro {monoid}] is a \kl {semigroup} that has a \kl
{unit}.
\end {description}
```

yields

⌞ In order to describe what is a **monoid**, let us us first define a product to be an associative binary operator, and a unit to be [...]

⌞ **A semigroup** is a set equipped with a **product**.

⌞ **A monoid** is a **semigroup** that has a **unit**.

One can check that the different knowledges are properly hyperlinked, and that precise targets are the one described by `\AP` and `\itemAP`. For helping debugging the *anchor points*, these are by default made visible as (red) corners on output. When the `knowledge` package is loaded with the `paper` option this graphical help disappears. This can also be deactivated using:

```
\knowledgeconfigure {visible anchor points=false}.
```

The color of `anchor points` can be changed using (in which colors are defined as in the package `xcolor`):

```
\knowledgeconfigure {anchor point color=color} , or
\knowledgeconfigure {AP color=color}.
```

The shape of `anchor points` can be changed using (in which shape can be one of `tiny corner`, `small corner`, `corner`, `large corner`, `tiny cross`, `small cross`, `cross`, `large cross`, or some code to be used in the `picture` environment):

```
\knowledgeconfigure {anchor point shape=shape} , or
\knowledgeconfigure {AP shape=shape}.
```

Finally, the `anchor points` reference point can be shifted, using:

```
\knowledgeconfigure {anchor point shift={x,y}} , or
\knowledgeconfigure {AP shift={x,y}} ,
```

in which x and y are by default in unit `em` (note the use of curly braces and of no parenthesis, which is mandatory).

3.8.3 The `makeidx` and `imakeidx` options

Activating the `makeidx` and `imakeidx` options The `makeidx` option loads the `imakeidx` package and triggers a certain number of link-related features. This is done either by the command:

```
\usepackage [makeidx]{knowledge}
```

or by loading the `makeidx` before the `knowledge` package. The same goes for the `imakeidx` option.

Features When activated, it becomes possible to trigger the `\index` command when a `\kl` command is used. The following `directives` are to be used:

`index=` is the text typeset in the index. It uses the standard syntax of the `\index` command. By default, it is the knowledge name itself. You can use the full syntax of `\index` in it, i.e. using `‘!` and `‘@’`.

`index key=` takes as argument the `index key`: a text that is used for identifying the `index entry` (usually an accent free version of it). You can use `‘!` in it, as long as it does not clash with `index=` in order to avoid clashes.

`index parent key=` makes the `index entry` be a subentry of the given `main index entry` (a replacement of `‘!`). Once more, it should not clash with `index=` and `index key=`.

`index style=` is followed by a token (without the escape character) that will be used for displaying the number (e.g. `index style=textbf`). Usually, this is to be used in order to typeset in a particular manner the knowledges in the index when introduced. By default, when introducing `notion` (or when

using `\KnowledgeConfigureNotion` for instance) the macros `\notionKnowledgeIndexStyle` and `\notionKnowledgeIndexIntroStyle` are defined and used. Hence, you can use for instance:

```
\def \notionKnowledgeIndexIntroStyle #1{\fbox {#1}}
```

`index name=` requires to activate `imakeidx`, and allows to choose the index name in case of multiple indexes.

The directive `no index`, deactivates the index feature for a knowledge. This is meant to be used in a `style`. For instance

```
\knowledgeDirective {standard}
    {autoref,style=standard,intro style=intro standard}
\knowledgeStyle {standard}{no index}
\knowledgeStyle {intro standard}{emphasize}

\knowledge {test}{standard,index= test}
```

result in that the introductions of ‘test’ will be gathered in the index, but not the other uses.

3.8.4 The `cleveref` option

The `cleveref` option loads the `cleveref` package and gives access to new commands: `\kcref`, `\kCref`, `\kcpageref`, `\kCpageref`, `\knamecref`, `\knameCref`, `\knamerefs`, and `\knameCrefs` which are respectively equivalent to `\cref`, `\Cref`, `\cpageref`, `\Cpageref`, `\namecref`, and `\nameCref` of the `cleveref` package, but applied to the label corresponding to the `knowledge` (see the documentation of `cleveref` for more details). These commands are variants of `\kl` and hence are subject to the same syntax.

For instance,

```
"Anchor points" have been described in \kCref {anchor point},
Page \kpageref {anchor point}.
```

would typeset as

Anchor points have been introduced in Section 3.8.2, Page 39.

3.9 Dealing with math

There are essentially two ways in which one would like to use `knowledge` with mathematics:

Single introduction Some mathematical objects are introduced once and for all in the paper. In this case, all the use of the object should point to the same introduction location. This way of doing is covered intensively in the

Variables The more advanced case is when one wants to track variables. For instance, a macro variable `\x` could have a different meaning in two distinct theorems. This case is more similar to variables in a programming language, that have a scope. Here the situation is slightly more complicated since a variable could be introduced in a theorem, and then used many pages later in

a proof section for instance. This is doable with the package, using scoping, but is not described yet in the documentation.

3.9.1 Defining knowledge-aware macros

It is possible to use the `\kl` command inside `\math`, and in math mode (however, the `\math` notation should really be avoided). Nevertheless, this may cause problematic results in various situations. For this reason, specialized tools are offered.

In this first section, we describe the simplest way to define macros that are linked to `\knowledge`. This covers almost all situations, as long as one is not interested in having the knowledge dependant of the parameters of the macro. In the subsequent sections, some more explanations are given covering the case of parameters.

The use through an example Let us define the `[x]` command representing the floor rounding of x , and use it in `[1.5]`. This is achieved by first defining a macro `\floor` (for instance in the preamble):

```
\knowledgenewrobustcmd \floor [1]{\cmdkl {\lfloor }#1\cmdkl
{\rfloor }}
```

and then use it in a natural way:

```
\AP Let us define the $\intro *\floor x$ command representing the
floor rounding of $x$, and use it in $\floor {1.5}$.
```

There, we used a special command, `\knowledgenewrobustcmd`, which is similar to the `\newrobustcmd`⁵ and at the same time introduces a `\knowledge` for the macro name (defined using the directive `automatic in command`). It sets the contexts such that `\cmdkl` implements the code for using this `\knowledge`. It is somehow similar to:

```
\newrobustcmd \floor [1]{\kl [\floor ]{\lfloor }#1\kl [\floor
]\{\rfloor }}
\knowledge {\floor }{automatic in command}
```

but is shorter to write and also corrects some bugs of this direct implementation⁶.

As explained above, the `\knowledge` command is implicit when defining such macros. The `\knowledge` name is the macro name itself, and the directive used is `automatic in command`. Hence, you can modify the behavior by redefining the directive:

```
\knowledgedirective *{automatic in command}{...}
```

Defining new commands We have used `\knowledgenewrobustcmd` in the above example. In fact, several commands are available for defining such knowledge-enhanced macros. Each of them corresponds to a normal command (from L^AT_EX, or `xparse`, or `mathcommand`), prefixed with ‘`\knowledge`’ or ‘`\Knowledge`’. The behavior of these commands is identical to the original ones,

⁵It has the same syntax as `\newcommand` and should always be used instead as far as a good reason for not to do it is given.

⁶For instance, with this wrong implementation, `\intro *\floor x` would display in `\intro` aspect the ‘`\lfloor`’ symbol but not the ‘`\rfloor`’ one.

and furthermore activates the ability to use `\cmdkl`. (Note that it is forbidden for your macro to absorb anything more than the parameters described during its definition.) The commands

```
\knowledgenewrobustcmd, [ \knowledgecommand, and
\knowledgecommand7],
```

correspond to `\newrobustcmd` (from `etoolbox`), `\newcommand`, `\renewcommand` respectively. The commands

```
\KnowledgeNewDocumentCommand,
\KnowledgeRenewDocumentCommand,
\KnowledgeProvideDocumentCommand, and
\KnowledgeDeclareDocumentCommand
```

extend the corresponding commands from the `xparse` package (it offers more complete possibilities to define the signature of macros, and do it in a cleaner and more systematic way).

Finally, the commands of the `mathcommand` package (see Section 3.9.3) are also extended (provided it has been loaded before the package `knowledge`), yielding:

```
\knowledgedeclarecommand,
\knowledgenewmathcommand,
\knowledgenewtextcommand,
\knowledgegerenewmathcommand,
\knowledgegerenewtextcommand,
\knowledgedeclaremathcommand,
\knowledgedeclaretextcommand,
\KnowledgeNewDocumentMathCommand,
\KnowledgeNewDocumentTextCommand,
\KnowledgeRenewDocumentMathCommand,
\KnowledgeRenewDocumentTextCommand,
\KnowledgeProvideDocumentMathCommand,
\KnowledgeProvideDocumentTextCommand,
\KnowledgeDeclareDocumentMathCommand,
\KnowledgeDeclareDocumentTextCommand,
\knowledgenewcommandPIE,
\knowledgegerenewcommandPIE,
\knowledgedeclarecommandPIE,
\knowledgegerenewmathcommandPIE,
\knowledgegerenewmathcommandPIE,
\knowledgedeclaremathcommandPIE,
\KnowledgeNewDocumentCommandPIE,
\KnowledgeRenewDocumentCommandPIE,
\KnowledgeDeclareDocumentCommandPIE,
\KnowledgeProvideDocumentCommandPIE,
\KnowledgeNewDocumentMathCommandPIE,
\KnowledgeRenewDocumentMathCommandPIE,
\KnowledgeDeclareDocumentMathCommandPIE, and
\KnowledgeProvideDocumentMathCommandPIE.
```

⁷These commands are here for completeness. You should always prefer `\knowledgenewrobustcmd` instead, in particular to avoid problems in sectioning commands.

3.9.2 The combination `\withkl` / `\cmdkl`

In fact, inside commands like `\knowledgegenewrobustcmd`, the command `\withkl` is used. It has to be used in combination with `\cmdkl`:

```
\withkl{\kl -like command}{...\cmdkl{A}...\cmdkl{B}...}
```

Inside the right argument of `\withkl`, `\cmdkl` stands for a shorthand for the `\kl`-like command, with the following specificities: the `modifiers` used before the `\withkl` are absorbed and reinjected for each use of `\cmdkl`. Furthermore, all the uses of `\cmdkl`, but the first one, get to be preceded of the `\re-kl*` modifier, which transforms in particular `\intro` into `\reintro`. Hence, `\intro *\withkl {\kl [example]}{...\cmdkl {A}...\cmdkl {B}...}` gets to be evaluated as `...\intro [example]{A}...\reintro [example]{B}...`

An example of use of this technique is if we want `knowledge` aware of the parameters of a macro. Imagine that you use the norms 1 and 2 of a function:

```
\newrobustcmd \norm [2]
  {\withkl {\kl [\norm {#1}]}{\cmdkl {\|}#2\cmdkl {\|}_\{\cmdkl
{#1}}}}
\knowledge {\norm {1}}{notion}
\knowledge {\norm {2}}{notion}
```

Now, the norm 1 and the norm 2 can be defined in different places since `knowledge` is specific to each case.

```
\AP The norm $1$ of a function $f$ is denoted $\intro *\norm 1f$.
[...]
\AP The norm $2$ of a function $f$ is denoted $\intro *\norm 2f$.
[...]
Here, $\norm 1g$ and $\norm 2h$ link to the proper place, while
$\norm 3g$ is an undefined knowledge.
```

3.9.3 Defining macros for math: the `mathcommand` package

Defining macros is standard in T_EX, and it should be used systematically when writing scientific documents. This is particularly true when using the `knowledge` package. The standard way in L^AT_EX for defining macros is to use `\newcommand`. However, the resulting macro, if it has no optional parameters, is then expandable, and it is better to avoid it using instead `\newrobustcmd` from the `etoolbox` package (it has the same syntax). Generally speaking, the `xparse` package, which gives easier syntax for describing commands, possibly with more complex parameter templates, should be preferred.

Another package, `mathcommand`, has been designed to be used in conjunction with `knowledge`⁸. We shall use it in the advanced examples below. It allows:

- to define/redefine commands to be used in math mode only. This allows for instance to use the macro `\c` for producing as usual a cedilla in text mode, and at the same time some variable `c` in math mode. This is achieved using for instance: `\renewmathcommand \c {\mathbf {c}}`.

⁸Note that all the commands for defining macros also exist in a `knowledge`-compatible form (see Section 3.9.1)

- When redefining a command, it automatically stores the original command `\macro` as `\LaTeXmacro`. So for instance, if one wants to introduce the constant π in a document and have it linked, one can use:

```
\renewmathcommand \pi {\k1 [\pi ]{\LaTeXpi }}
\knowledge \pi {notion}
[...]
\AP Let $\intro *\pi =3.1415$.
[...]
Now $\pi$ points to the above sentence.
```

This code works because a different name `\LaTeXpi` stores the original macro. Using `\pi` instead would yield an infinite loop. Even better is to use the knowledge-aware version:

```
\knowledge\renewmathcommand \pi {\cmdkl {\LaTeXpi }}
```

- The package also gives access to the exponents and indices as well as primes that follow a command (see the documentation).
- It also has some facilities for disabling \LaTeX commands and provide suggestions of replacement (useful for remembering the macros and working with colleagues).
- Finally, it offers some commodity for redefining many variables in one command. E.g. for defining `\calA`, `\calB`, ... to be shortcuts for `\cal {A}`, `\cal {B}`, ...:

```
\LoopCommands {ABCDEFGHJKLMNOPQRSTUVWXYZ}[cal#1]
{\newmathcommand #2{\cal {#1}}}
```

3.9.4 Mathematical objects that are singly introduced

In this case, this is essentially as in text mode. Five points have to be kept in mind:

- Do not use the [quotation notation](#) in math mode and in macros. Indeed, some packages, like `tikzcd` use the double-quote symbol in their (math) syntax. In this context, quotes have to be deactivated, and hence macros that would use the [quotation notation](#) would suddenly not have the expected result.
- Do not mix the [knowledge](#) concerning math commands and normal text. It is in practice difficult to maintain.
- Use the control sequence of the macro itself as the [knowledge](#) name. This does not cause problems and is easier to maintain. This is automatically done when commands such as `\knowledge\newrobustcmd` are used.
- Use the `\intro *` notation for introducing macros.

- The surrounding typography may be broken when using the command `\kl`, or `\cmdkl` (if someone knows how to solve that, please contact me). Commands such as `\mathrel` should be used to recover it.

These five points are illustrated in the following code:

```
\knowledgegnewrobustcmd \comp {\mathrel {\cmdkl {\circ }}}
[...]
\AP Composition is denoted $\intro *\comp $.
[...]
Now, each use of $\comp $ points to its introduction.
```

Two variation may be comfortable to use.

Disabling commands When writing a paper, in particular with coauthors, one may be tempted to not always use the macros designed for each case. For helping to remember the macro, one can use instead (using the `mathcommand` package):

```
\disablecommand\circ
\suggestcommand\circ {Use instead \comp for function composition.}
\knowledgegnewrobustcmd \comp {\cmdkl {\LaTeXcirc }}
```

The result is that if one uses `\circ` in the code, an error will be issued, and `\comp` will be suggested as a replacement. Note that more than one suggestions can be attached to the macro (if several macros use the symbol `\comp` with different meanings). Note also that in the definition of `\comp`, `\LaTeXcirc` is used instead of `\circ`. Indeed, `\circ` has been deactivated, but `\LaTeXcirc` gives access to its original meaning.

Redefining the original T_EX macro Another situation is that one would like to use the `\circ` control sequence for accessing our function instead of `\comp` (simply because this is more convenient and easy to remember, and we know in advance that no confusion may arise). In this case, the `mathcommand` package can also be of some help:

```
\knowledgegerenewmathcommand \circ {\mathrel {\cmdkl {\LaTeXcirc }}}
[...]
\AP Let $g\intro *\circ f$ denote the composition of functions.
[...]
Now, each use of $\circ $ points to its introduction.
```

The effect of `\knowledgegerenewmathcommand` is that it sets `\LaTeXcirc` to have the same effect as the original `\circ` command, and then redefines `\circ`.

3.9.5 Context dependent variables

This section is not yet written.

3.10 Predefined configuration

3.10.1 The `notion` directive

The configuration option `notion` is activated using:

```
\knowledgeconfigure {notion}
```

It automatically configures a directive `notion` which is an `autoref` configured to be displayed in a configurable way:

- In `paper mode`, the `\intro` commands (not in math mode) are emphasized, while the `\kl` commands are displayed as normal. It has the aspect of a normal paper.
- In `electronic mode` and `composition mode` (with the `xcolor` package), notions are furthermore typeset in blue when introduced, and in dark blue when used. Without the `xcolor` package, underlining draw the attention to the knowledges (not in math mode).

The behavior of the `notion` directive is to activate `autoref`, and to configure the following two styles:

- the style `notion` is used for normal use,
- the style `intro notion` is used for introduction.

A typical document using notion could start by the following commands:

```
\documentclass {article}
\usepackage {xcolor}
\usepackage {hyperref}
\usepackage [electronic]{knowledge}
\knowledgeconfigure {notion}
[...]
\knowledge {some text}{notion}
```

Then the paper is displayed in a colorful way.

3.11 Fixes

In this section, we present some fixes that have been added to help the user solve problems.

Hyperref and twocolumn It happens that the `hyperref` and two-column mode together may yield a fatal error. This happens when a link spans across the boundary between two pages. This is an issue which is not related to the `knowledge` package, but becomes severely more annoying when more links have to be used. A *workaround* can be tried by using using

```
\knowledgeconfigure {fix hyperref twocolumn}.
```

I do not know to which extend it is compatible with various classes...

4 Some questions and some answers

4.1 How to compile?

As usual with L^AT_EX, a certain number of compilation phases are necessary for reaching a document in final form. The problematic point is of course the use of labels, and in particular the `\intro` command. When it is used, and all the `\knowledge` commands are in the preamble, then two phases are necessary. When `\knowledge` commands are used in the body of the documents, then one extra phase is required, meaning three with `autoref` definitions. This is also the case when `scoping` is used.

4.2 Problem with `\item` parameters

The use of `\AP` inside the optional parameter of `\item` does not work. Do not use `\AP` inside the optional argument of `\item`, and rather use the command `\itemAP`.

Argument of `\kl` has an extra ‘’. This is a problem of using optional parameters inside optional parameters such as in `\item [\kl [test]{Test}]`. You can surround the content of the optional parameter by two level of curly braces as in `\item [{\kl [test]{Test}}]`. The notation `"..."` does not have this issue.

4.3 Knowledges and moving arguments (table of contents, ...).

The use of `\kl` does not work in (e.g.) the table of content. When the `knowledge name` contains expandable macros, or accentuated letters, then these are not copied in the table of content as the exact same text, but are expanded/translated. Thus, when the table of content is displayed, the `\kl` command complains of not knowing the `knowledge`. For instance⁹:

```
\newcommand \Ltwo {\ensuremath {L^2}}
\knowledge {\Ltwo -space}[\Ltwo -spaces]{autoref}
\knowledge {  tale topology}[  tale topology]
    {url={https://en.wikipedia.org/wiki/  tale_topology}}
[...]
\begin {document}
\tableofcontents
\section {On \kl {\Ltwo -spaces}
[...]
\section {On the \kl {  tale topology}}
[...]
\end {document}
```

⁹with `\usepackage [utf8]{inputenc}` and, for instance `\usepackage [T1]{fontenc}` for the accents.

will result in that both `knowledge`s are considered unknown in the table of contents. For the first one, this is due to the expansion of `\Ltwo` . For the second, this is due to an implicit translation of the accentuated letter into an internal sequence of commands (for instance ‘é’ is translated into the internal sequence ‘\IeC {\’e}’). Some solutions are as follows:

- Make the macros non-expandable, for instance using `\newrobustcmd` (of the `etoolbox` package) or `\NewDocumentCommand` (of the `xparse` package, with a different handling of arguments) instead of `\newcommand` . Hence:

```
\newrobustcmd \Ltwo {\ensuremath {L^2}}
```

solves the first problem.

- Using an equivalent text that does not have the problem:

```
\knowledge {\’etale topology}{link=étale topology}
[...]
\section {On the \kl {\’etale topology}}
```

- Both problems can be solved using synonyms/links that have no problem. For instance:

```
\knowledge {Ltwo-space}{link=\Ltwo -space}
\knowledge {etale topology}{link=étale topology}
[...]
\section {On \kl [Ltwo-space]{\Ltwo -spaces}}
\section {On the \kl [etale topology]{étale topology}}
```

- Other solutions? None so far. I am trying to systematize the treatment of these problems.

Using `\intro` in a section title causes introducing the knowledge twice. Do not use `\intro` in titles, but rather `\reintro` . If you want the section to be the target of the `knowledge`, then put after the section a `\pantomintro` command.

```
\section {On \intro
{topology}}
```

Problematic code

```
\section {On \reintro
{topology}}
\pantomintro {topology}
```

A solution

4.4 Problems with `tikzcd` and other issues with the `quotation notation`

The package `tikzcd` uses (heavily) the quotes. Thus, it conflicts with the `quotation notation`. Some other packages may do the same. For solving this issue, the only things to do are:

- be sure to load these packages before `knowledge`, or at least be sure that the `quotation notation` is not active when you do so, and

- to temporarily deactivate the `\quotation notation` when in a context where the package may use the quotes.

This can be done either explicitly using before each figure:

```
\knowledgeconfigure {quotation=false}
```

and after the figure:

```
\knowledgeconfigure {quotation}
```

Another possibility is to force some environment to deactivate systematically the `\quotation notation` when used. For instance

```
\knowledgeconfigure {protect quotation={tikzcd}}
```

will deactivate the `\quotation notation` in all the `tikzcd` environments.

4.5 Problems with `amsmath`

The `\intro` command does not work in `align*` or similar environments. It happens that in `stared` environment (i.e., unnumbered), the package `amsmath` deactivates the labels. As a consequence the command `\intro`, which internally uses `\label` (at least so far), does not work. For the moment, there is no real solution, but a workaround which consists in introducing the `knowledge` before the incriminated environment using `\phantomsection`, and then use `\reintro` inside the environment. Imagine for instance a command `\SomeCommand`, that inside uses `\kl` [`\Somecommand`], then:

does not work

```
\begin {align*}
\intro *\SomeCommand
\end {align*}
```

works

```
\phantomintro \SomeCommand
\begin {align*}
\reintro *\SomeCommand
\end {align*}
```

4.6 Hyperref complains

A fatal error occurs in `twocolumn` mode. A workaround is to use `\knowledgeconfigure {fix hyperref twocolumn}`.

4.7 Name clash (eg with the `complexity` package)

It may happen that an already defined command is redefined by the `knowledge` package. This happens in particular when used in combination with the `complexity` package: both packages try to define the command `\AP`. For `complexity`, this denotes a complexity class, and for `knowledge`, it is an anchor point. The problem has to be resolved by hand. The two following solutions are quick hacks:

```
\usepackage {complexity}
\let \compAP \AP
\let \AP \undefined
\usepackage [...] {knowledge}
for using \AP from knowledge,
and \compAP for complexity,
```

```
\usepackage [...] {knowledge}
\let \kAP \AP
\let \AP \undefined
\usepackage {complexity}
for using \AP from complexity,
and \kAP for knowledge.
```

4.8 Incorrect display

4.8.1 Incorrect breaking at the end of lines (in `arXiv` for instance)

It may happen that some hyperlinks generated by `knowledge` are not broken properly at the end of lines. This is an issue with the `hyperref` package. This happens in particular for files compiled by the `arXiv` system while the file on the local computer was not having any problem. A workaround is to use the `breaklinks` option of `hyperref`. If you need this for `arXiv`, then you also have to force the use of `\pdflatex` (because the `breaklinks` option does not work if compiled via the ancestral sequence `TEX`→`DVI`→`PS`→`PDF`). This can be obtained by adding `\pdfoutput =1` within the five first lines of the preamble.

The preamble thus looks like:

```
\documentclass {[...]}
\pdfoutput =1
[...]
\usepackage [breaklinks]{hyperref}
[...]
\usepackage {knowledge}
[...]
```

4.9 Display errors in pdf output, in particular `arXiv`

Again, try in the preamble:

```
\pdfoutput =1
```

4.9.1 Red boxes around links

This is an annoying feature of the `hyperref` package to surround all links by red boxes (that may appear or not depending on the viewer). This is very heavy in document with many links. By default, this is deactivated when using the package `knowledge`, unless the `no patch` option is used when loading the package. When the `no patch` option is used, the same effect can nevertheless be obtained, eg, using `\hypersetup {hidelinks}`.

4.9.2 Incorrect color for links in `paper mode` (e.g. red in with `acmart`)

Some classes, like `acmart`, change the default color of the hyperlinks of `hyperref`. Since `knowledge` does not apply any color to the links in `paper mode`, this unexpected color appears. The black color of the `paper mode` can be forced by using:

```
\IfKnowledgePaperMode
\knowledgestyle {kl}{color=black}
\knowledgestyle {notion}{color=black}
\ fi
```

4.9.3 Unexpected color in margin paragraph

This is a problem of the combination of the package `xcolor` with `\marginpar` : when a colored text gets to be broken into separated lines and the `\marginpar`

be inserted at this place, the colors leaks in to the margin text. It is independent of `knowledge`, but is more likely to occur when colors are often used. A correction is to force the use of the color black whenever inside a `\marginpar`. For instance:

```
\let \LaTeXmarginpar \marginpar
\def \marginpar #1{\LaTeXmarginpar {\textcolor {black}{#1}}}
```

4.10 Problems with scope

4.10.1 Problems in combination with `\bibitem` and the environment `thebibliography`

The `scope option` of the package triggers some analysis of the code, and restrains the structure of the code (in particular, this is because `scopes` have to be nested, and thus some not so well nested parts of L^AT_EX yield errors). In particular, the `scope option` does not allow to have a `\section` command inside a list. However, this is what does the environment `thebibliography`, yielding a scoping error.

A simple hack to treat this situation:

```
\let \section \SUPERsection
\begin {thebibliography}
\bibitem ...
[...]
\end {thebibliography}
\let \section \NEWsection
```

The result is to revert to the original version of the macro `\section`, which does not make any structural test, and then reactivate the modified version of the command.

Another solution is to reconfigure the environment `thebibliography` using in the preamble:

```
\ScopeConfigure {thebibliography}
{push code=\let \section \SUPERsection ,
pop code=\let \section \NEWsection }
```

4.11 Editors

4.11.1 Emacs editor and `quotes`

The AucTex mode in Emacs binds the quote symbol to other characters, cycling through ‘, ", and ". This is not convenient when using the `knowledge` package.

This behavior can be deactivated temporarily using:

```
M-x local-unset-key RET " RET
```

or definitively using:

```
(defun my-hook () (local-unset-key "\""))
(add-hook 'LaTeX-mode-hook 'my-hook)
```

Alternatively, this can be changed so as to cycle through ", ‘, and ", which is slightly more convenient than the default. This is achieved by customizing `TeX-quote-after-quote`:

M-x customize-set-variable RET TeX-quote-after-quote RET y

If you also use `smartparens`, there are some known interaction problems with AucTex (see <https://github.com/Fuco1/smartparens/issues/1100>). A reported solution is to execute:

```
(setq TeX-quote-after-quote t)
(sp-with-modes '(tex-mode plain-tex-mode latex-mode LaTeX-mode)
  (sp-local-pair "\"" nil :actions nil)
  (sp-local-pair "\"\" nil :actions nil)
  (sp-local-pair "\"" nil :actions nil)
  (sp-local-pair "\"\" \"'\" :trigger nil :actions nil)
)
```

4.12 Others

If other kind of problems occur, report them to `thomas.colcombet@irif.fr`. You can also do it if you found a solution. It will be added.

5 Resources

5.1 List of commands

`\AP` introduces an `anchor point`.
`\intro` searches for a `knowledge` and put an anchor to it (to be used with the `autoref` directive).
`\kcref` , `\kCref` , `\kcpageref` , `\kCpageref` , `\knamecref` , `\knameCref` , `\knamecref` and `\knameCref` display some information concerning the place of introduction of a `knowledge` as the `\cref` , `\Cref` , `\cpageref` , `\Cpageref` , `\namecref` , `\nameCref` , `\namecref` and `\nameCref` commands of the `cleveref` package.
`\kl` searches for a `knowledge` and displays it accordingly.
`\knowledge` defines new `knowledges`.
`\knowledgeconfigure` configures the package.
`\knowledgedirective` defines a new `directive`.
`\knowledgedefault` declares the default `directives` to be automatically used in `\knowledge` commands.
`\knowledgeimport` gives access to `knowledges` existing in other scopes.
`\knowledgevariant` defines a new `variant` of `\kl` .
`\knowledgevariant` configures a `variant` of `\kl` .
`\knowledgestyle` defines a new `style`.
`\knowledgevariantmodifier` declares a meaning of `*` in `variants` of `\kl` .
`\kpageref` displays the page number of a reference,
`\kref` displays the number associated to a reference introduction.
`\nointro` declares that the knowledge will never be introduced (does not work properly yet).
`\phantomintro` performs an invisible `\intro` .
`\reintro` uses the `display style` of `\intro` without introducing an anchor.

5.2 List of environments

`export` (not implemented) requires exportation of the content.
`import` (not implemented) declares external resources.
`scope` Defines a `scope` in which `knowledges` are internal.

5.3 List of directives (to use with `\knowledge`)

`autoref` Activates the `\intro` feature.
`autoref link` activates an hyperlink to the target.
`autoref target` puts a target for a hyperlink.
`autorefhere` creates an `anchor point` that points to the `\knowledge` command (Requires the `hyperref option`).
`boldface` Displays the knowledge in boldface.
`color=` Displays the `knowledge` is the given color (resquires `xcolor`).
`colorbox=` Displays the `knowledge` in a box of the given color (requires `xcolor`).
`cyclic color` Displays in a color among a cyclic list (requires `xcolor`).
`detokenize` Avoids evaluation of the text.
`emphasize` Emphasizes the displayed output.
`ensuretext` Guarantees that the output will be displayed in text mode.

ensuremath Guarantees that the output will be displayed in math mode.
export= (not implemented)
invisible= no display
italic= displays in italic
fbox Surround the text with a box.
md Removes boldface typesetting.
index= Chooses the text to be displayed in the **index=**.
index key= the key used to choose the place in the **index**.
index name= the name of the **index** (requires **imakeidx**).
index style= the **style** to be used to display in the **index**.
index parent key= the parent key in the **index**.
intro style= Chooses the typesetting in case of an intro.
italic Typesets the output in italic.
link= Follow with the search the linked knowledge.
link scope= Follow the search in the corresponding scope, using the same key, or the one provided by **link=** if present.
lowercase Put all letters of the output in lowercase.
mathord, mathop, mathbin, mathrel, mathopen, mathclose, mathpunct Selects a spacing behaviour in math mode.
no index avoids the indexing of the term.
notion automatic configuration for displaying and autoreferencing.
protect link Disables the hyperlinks inside the link.
ref= Links to a label inside the document.
scope= Choose the **scope** of the definition.
smallcaps Forces the use of small capitals.
style= Links to a style.
synonym Is a synonym of the lastly defined **knowledge**.
text= Changes the output text.
remove space removes the spaces from the input
typewriter Typeset in as with `\texttt` .
underline Underlines the text.
up Removes italic typesetting.
uppercase Put all letters of the output in uppercase.
url= An url to point to (uses the **hyperref**).
wrap= A macro used to process the displayed text.

5.4 List of configuration directives

The following directives can be used as options when loading the package, or later using `\knowledgeconfigure` .

anchor point color=color changes the color of **anchor point**,
anchor point shape=shape changes the shape of **anchor points**, in which shape can be `tiny corner`, `small corner`, `corner`, `large corner`, `tiny cross`, `small cross`, `cross`, `large cross`, or some code to be execute in the `picture` environment,
anchor point shift={x,y} shifts the **anchor point** (in em unit),
diagnose bar={true,false} activates the ‘|’-notation in the **diagnose file** (default is false)
composition switches to **composition mode**,
cyclic colors= fixes the cyclic list of colors used by the directive **cyclic color**.

`diagnose line={true,false}` activates or deactivates the line numbering in the diagnose file.

`electronic` switches to `electronic mode`.

`fix hyperref twocolumn` fixes a known problem between `hyperref` and the two column mode.

`hyperlinks={true,false}` activates or deactivates the hyperlinks.

`no patch` deactivates the default patches made to other packages.

`notion` activates the `notion` directive.

`paper` switches to `paper mode`.

`label scope={true,false}` enables or disables the redefined `\label` command, which helps automatically define scopes (default is true).

`protect link and unprotect link` starts and ends respectively a zone in which the `knowledge` package does not create hyperlinks.

`protect quotation={environment list}` declares a list of environment in which the `quotation notation` should be deactivated.

`quotation={true,false}` activates or deactivates the `quotation notation`.

`silent` is a Boolean option which, when activated, switches off all warnings during the compilation, but the recap ones at the end.

`strict` is a Boolean which, when true, makes the compilation more restrictive by turning some of the warnings into errors (in particular in case of redefinition of knowledges).

`visible anchor points={true,false}` makes the `anchor points` either visible or invisible.

List of default `styles`

```
intro  
kl unknown, kl unknown cont  
intro unknown, intro unknown cont  
notion (if notion is activated)  
intro notion (if notion is activated)
```