`modules.sty`: Semantic Macros and Module Scoping in sTeX[*]

Michael Kohlhase & Deyan Ginev & Rares Ambrus
Jacobs University, Bremen
http://kwarc.info/kohlhase

October 23, 2014

Abstract

The `modules` package is a central part of the sTeX collection, a version of TeX/LaTeX that allows to markup TeX/LaTeX documents semantically without leaving the document format, essentially turning TeX/LaTeX into a document format for mathematical knowledge management (MKM).

This package supplies a definition mechanism for semantic macros and a non-standard scoping construct for them, which is oriented at the semantic dependency relation rather than the document structure. This structure can be used by MKM systems for added-value services, either directly from the sTeX sources, or after translation.

---

[*]Version v1.3 (last revised 2013/12/12)

Contents

1 Introduction

Following general practice in the TeX/LaTeX community, we use the term "semantic macro" for a macro whose expansion stands for a mathematical object, and whose name (the command sequence) is inspired by the name of the mathematical object. This can range from simple definitions like `\def\Reals{\mathbb{R}}` for individual mathematical objects to more complex (functional) ones object constructors like `\def\SmoothFunctionsOn#1{\mathcal{C}^\infty(#1,#1)}`. Semantic macros are traditionally used to make TeX/LaTeX code more portable. However, the TeX/LaTeX scoping model (macro definitions are scoped either in the local group or until the rest of the document), does not mirror mathematical practice, where notations are scoped by mathematical environments like statements, theories, or such. For an in-depth discussion of semantic macros and scoping we refer the reader [Koh08].

The `modules` package provides a LaTeX-based markup infrastructure for defining module-scoped semantic macros and LaTeXML bindings [LTX] to create OMDoc [Koh06] from sTeX documents. In the sTeX world semantic macros have a special status, since they allow the transformation of TeX/LaTeX formulae into a content-oriented markup format like OpenMath [Bus+04] and (strict) content MathML [Aus+10]; see Figure 1 for an example, where the semantic macros above have been defined by the `\symdef` macros (see Section 2.2) in the scope of a `\begin{module}[id=calculus]` (see Section 2.5).

LaTeX	`\SmoothFunctionsOn\Reals`
PDF/DVI	$\mathcal{C}^\infty(\mathbb{R}, \mathbb{R})$
OpenMath	% `<OMA>` % `  <OMS cd="calculus" name="SmoothFunctionsOn"/>` % `  <OMS cd="calculus" name="Reals"/>` % `</OMA>`
MathML	% `<apply>` % `  <csymbol cd="calculus">SmoothFunctionsOn</csymbol>` % `  <csymbol cd="calculus">Reals</csymbol>` % `</apply>`

Example 1: OpenMath and MathML generated from Semantic Macros

2 The User Interface

The main contributions of the `modules` package are the `module` environment, which allows for lexical scoping of semantic macros with inheritance and the `\symdef` macro for declaration of semantic macros that underly the `module` scoping.

2.1 Package Options

showmods The `modules` package takes two options: If we set `showmods`[1], then the views (see

EdN:1

---

[1]EdNote: This mechanism does not work yet, since we cannot disable it when importing modules and that leads to unwanted boxes. What we need to do instead is to tweak the sms utility to use an

Section 2.9) are shown. If we set the `qualifiedimports` option, then qualified imports are enabled. Qualified imports give more flexibility in module inheritance, but consume more internal memory. As qualified imports are not fully implemented at the moment, they are turned off by default see Limitation 3.4.

noauxreq The option `noauxreq` prohibits the registration of `\@requiremodules` commands in the `aux` file. They are necessary for preloading the module signatures so that entries in the table of contents can have semantic macros; but as they sometimes cause trouble the option allows to turn off preloading.

showmeta If the `showmeta` is set, then the metadata keys are shown (see [Koh14a] for details and customization options).

2.2 Semantic Macros

\symdef The is the main constructor for semantic macros in sTeX. A call to the `\symdef` macro has the general form

$$\symdef[\langle keys\rangle]\{\langle cseq\rangle\}[\langle args\rangle]\{\langle definiens\rangle\}$$

where $\langle cseq\rangle$ is a control sequence (the name of the semantic macro) $\langle args\rangle$ is a number between 0 and 9 for the number of arguments $\langle definiens\rangle$ is the token sequence used in macro expansion for $\langle cseq\rangle$. Finally $\langle keys\rangle$ is a keyword list that further specifies the semantic status of the defined macro.

The two semantic macros in Figure 1 would have been declared by invocations of the `\symdef` macro of the form:

```
\symdef{Reals}{\mathbb{R}}
\symdef{SmoothFunctionsOn}[1]{\mathcal{C}^\infty(#1,#1)}
```

Note that both semantic macros correspond to OpenMath or MathML "symbols", i.e. named representations of mathematical concepts (the real numbers and the constructor for the space of smooth functions over a set); we call these names the **symbol name** of a semantic macro. Normally, the symbol name of a semantic macro declared by a `\symdef` directive is just $\langle cseq\rangle$. The key-value pair

name **name**=$\langle symname\rangle$ can be used to override this behavior and specify a differing name. There are two main use cases for this.

The first one is shown in Example 3, where we define semantic macros for the "exclusive or" operator. Note that we define two semantic macros: `\xorOp` and `\xor` for the applied form and the operator. As both relate to the same mathematical concept, their symbol names should be the same, so we specify `name=xor` on the definition of `\xorOp`.

local A key `local` can be added to $\langle keys\rangle$ to specify that the symbol is local to the module and is invisible outside. Note that even though `\symdef` has no advantage over `\def` for defining local semantic macros, it is still considered good style to use `\symdef` and `\abbrdef`, if only to make switching between local and exported semantic macros easier.

primary Finally, the key `primary` (no value) can be given for primary symbols.

---

internal version that never shows anything during sms reading.

\abbrdef　　　　The `\abbrdef` macro is a variant of `\symdef` that is only different in semantics, not in presentation. An abbreviative macro is like a semantic macro, and underlies the same scoping and inheritance rules, but it is just an abbreviation that is meant to be expanded, it does not stand for an atomic mathematical object.

We will use a simple module for natural number arithmetics as a running example. It defines exponentiation and summation as new concepts while drawing on the basic operations like $+$ and $-$ from LaTeX. In our example, we will define a semantic macro for summation `\Sumfromto`, which will allow us to express an expression like $\sum i = 1^n x^i$ as `\Sumfromto{i}1n{2i-1}` (see Example 2 for an example). In this example we have also made use of a local semantic symbol for n, which is treated as an arbitrary (but fixed) symbol.

```
\begin{module}[id=arith]
  \symdef{Sumfromto}[4]{\sum_{#1=#2}^{#3}{#4}}
  \symdef[local]{arbitraryn}{n}
  What is the sum of the first $\arbitraryn$ odd numbers, i.e.
  $\Sumfromto{i}1\arbitraryn{2i-1}?$
\end{module}
```
What is the sum of the first n odd numbers, i.e. $\sum_{i1n}2i-1?$

Example 2: Semantic Markup in a `module` Context

\symvariant　　　　The `\symvariant` macro can be used to define presentation variants for semantic macros previously defined via the `\symdef` directive. In an invocation

　　　　`\symdef[`⟨*keys*⟩`]{`⟨*cseq*⟩`}[`⟨*args*⟩`]{`⟨*pres*⟩`}`
　　　　`\symvariant{`⟨*cseq*⟩`}[`⟨*args*⟩`]{`⟨*var*⟩`}{`⟨*varpres*⟩`}`

the first line defines the semantic macro \⟨*cseq*⟩ that when applied to ⟨*args*⟩ arguments is presented as ⟨*pres*⟩. The second line allows the semantic macro to be called with an optional argument ⟨*var*⟩: \⟨*cseq*⟩`[var]` (applied to ⟨*args*⟩ arguments) is then presented as ⟨*varpres*⟩. We can define a variant presentation for `\xor`; see Figure 3 for an example.

```
\begin{module}[id=xbool]
  \symdef[name=xor]{xorOp}{\oplus}
  \symvariant{xorOp}{uvee}{\underline{\vee}}
  \symdef{xor}[2]{#1\xorOp #2}
  \symvariant{xor}[2]{uvee}{#1\xorOp[uvee] #2}
  Exclusive disjunction is commutative: $\xor{p}q=\xor{q}p$\\
  Some authors also write exclusive or with the $\xorOp[uvee]$ operator,
  then the formula above is $\xor[uvee]{p}q=\xor[uvee]{q}p$
\end{module}
```
Exclusive disjunction is commutative: $pq = qp$
Some authors also write exclusive or with the $\underline{\vee}$ operator, then the formula above is $pq = qp$

Example 3: Presentation Variants of a Semantic Macro

Version 1.0 of the `modules` package had the `\resymdef` macro that allowed to locally redefine the presentation of a macro. But this did not interact well with the `beamer` package and was less useful than the `\symvariant` functionality. Therefore it is deprecated now and leads to an according error message.

2.3 Testing Semantic Macros

One of the problems in managing large module graphs with many semantic macros, so the `module` package gives an infrastructure for unit testing. The first macro is \symtest `\symtest`, which allows the author of a semantic macro to generate test output (if the `symtest` option is set) see figure 4 for a "tested semantic macro definition". Note that the language in this purely generated, so that it can be adapted (tbd).

```
\symdef[name=setst]{SetSt}[2]{\{#1\,\vert\,#2\}}
\symtest[name=setst]{SetSt}{\SetSt{a}{a>0}}
```
generates the output
Symbol setst with semantic macro \SetSt: used e.g. in $\{aa > 0$

Example 4: A Semantic Macro Definition with Test

\abbrtest The `\abbrtest` macro gives the analogous functionality for `\abbrdef`.

2.4 Symbol and Concept Names

Just as the `\symdef` declarations define semantic macros for mathematical symbols, the `modules` package provides an infrastructure for *mathematical concepts* that are expressed in mathematical vernacular. The key observation here is that concept names like "finite symplectic group" follow the same scoping rules as mathematical symbols, i.e. they are module-scoped. The `\termdef` macro is an ana-\termdef logue to `\symdef` that supports this: use `\termdef[⟨keys⟩]{⟨cseq⟩}{⟨concept⟩}` to declare the macro \⟨cseq⟩ that expands to ⟨concept⟩. See Figure 5 for an ex-\capitalize ample, where we use the `\capitalize` macro to adapt ⟨concept⟩ to the sentence EdN:2 beginning.[2] The main use of the `\termdef`-defined concepts lies in automatic \termref cross-referencing facilities via the `\termref` and `\symref` macros provided by the \symref `statements` package [Koh14c]. Together with the `hyperref` package [RO], this provide cross-referencing to the definitions of the symbols and concepts. As discussed in section 3.6, the `\symdef` and `\termdef` declarations must be on top-level in a module, so the infrastructure provided in the `modules` package alone cannot be used to locate the definitions, so we use the infrastructure for mathematical statements for that.

```
\termdef[name=xor]{xdisjunction}{exclusive disjunction}
\capitalize\xdisjunction is commutative: $\xor{p}q=\xor{q}p$
```

Example 5: Extending Example 3 with Term References

---

[2]EDNOTE: continue, describe ⟨*keys*⟩, they will have to to with plurals,. . . once implemented

2.5   Modules and Inheritance

module   The module environment takes an optional KeyVal argument. Currently, only the
id key is supported for specifying the identifier of a module (also called the module
name). A module introduced by \begin{module}[id=foo] restricts the scope the
semantic macros defined by the \symdef form to the end of this module given by
the corresponding \end{module}, and to any other module environments that
import them by a \importmodule{foo} directive. If the module foo contains
\importmodule directives of its own, these are also exported to the importing
module.

\importmodule   Thus the \importmodule declarations induce the semantic inheritance rela-
tion. Figure 7 shows a module that imports the semantic macros from three
others. In the simplest form, \importmodule{⟨mod⟩} will activate the semantic
macros and concepts declared by \symdef and \termdef in module ⟨mod⟩ in the
current module[1]. To understand the mechanics of this, we need to understand a
bit of the internals. The module environment sets up an internal macro pool, to
which all the macros defined by the \symdef and \termdef declarations are added;
\importmodule only activates this macro pool. Therefore \importmodule{⟨mod⟩}
can only work, if the TeX parser — which linearly goes through the sTeX sources
— already came across the module ⟨mod⟩. In many situations, this is not ob-
tainable; e.g. for "semantic forward references", where symbols or concepts are
previewed or motivated to knowledgeable readers before they are formally intro-
duced or for modularizations of documents into multiple files. To enable situations
like these, the module package uses auxiliary files called **sTeX module signa-
tures**. For any file, ⟨file⟩.tex, we generate a corresponding sTeX module signature
⟨file⟩.sms with the sms utility (see also Limitation 3.1), which contains (copies of)
all \begin/\end{module}, \importmodule, \symdef, and \termdef invocations
in ⟨file⟩.tex. The value of an sTeX module signature is that it can be loaded in-
stead its corresponding sTeX document, if we are only interested in the semantic
macros. So \importmodule[load=⟨filepath⟩]{⟨mod⟩} will load the sTeX module
signature ⟨filepath⟩.sms (if it exists and has not been loaded before) and acti-
vate the semantic macros from module ⟨mod⟩ (which was supposedly defined in
⟨filepath⟩.tex). Note that since ⟨filepath⟩.sms contains all \importmodule state-
ments that ⟨filepath⟩.tex does, an \importmodule recursively loads all necessary
EdN:3   files to supply the semantic macros inherited by the current module.[3]

importmodulevia   The \importmodule macro has a variant \importmodulevia that allows the
specification of a theory morphism to be applied. \importmodulevia{⟨thyid⟩}{⟨assignments⟩}
specifies the "source theory" via its identifier ⟨thyid⟩ and the morphism by
⟨assignments⟩. There are three kinds:

\vassign   **symbol assignments** via \vassign{⟨sym⟩}{⟨exp⟩}, which defines the symbol
⟨sym⟩ introduced in the current theory by an expression ⟨exp⟩ in the source
theory.

\tassign   **term assignments** via \tassign[⟨source-cd⟩]{⟨tname⟩}{⟨source-tname⟩}, which

---

[1]Actually, in the current TeX group, therefore \importmodule should be placed directly after
the \begin{module}.

[3]EDNOTE: MK: document the other keys of module

7

assigns to the term with name ⟨*tname*⟩ in the current theory a term with name ⟨*source-tname*⟩ in the theory ⟨*source-cd*⟩ whose default value is the source theory.

\ttassign **term text assignments** via \tassign{⟨*tname*⟩}{⟨*text*⟩}, which defines a term with name ⟨*tname*⟩ in the current theory via a definitional text.

```
\begin{module}[id=ring]
\begin{importmodulevia}{monoid}
  \vassign{rbase}\magbase
  \vassign{rtimesOp}\magmaop
  \vassign{rone}\monunit
\end{importmodulevia}
\symdef{rbase}{G}
\symdef[name=rtimes]{rtimesOp}{\cdot}
\symdef{rtimes}[2]{\infix\rtimesOp{#1}{#2}}
\symdef{rone}{1}
\begin{importmodulevia}{cgroup}
  \vassign{rplus}\magmaop
  \vassign{rzero}\monunit
  \vassign{rinvOp}\cginvOp
\end{importmodulevia}
\symdef[name=rplus]{rplusOp}{+}
\symdef{rplus}[2]{\infix\rplusOp{#1}{#2}}
\symdef[name=rminus]{rminusOp}{-}
\symdef{rminus}[1]{\infix\rminusOp{#1}{#2}}
...
\end{module}
```

Example 6: A Module for Rings with inheritance from monoids and commutative groups

\metalanguage The \metalanguage macro is a variant of importmodule that imports the meta language, i.e. the language in which the meaning of the new symbols is expressed. For mathematics this is often first-order logic with some set theory; see [**RabKoh:WSMSML10**] for discussion.

2.6 Dealing with multiple Files

The infrastructure presented above works well if we are dealing with small files or small collections of modules. In reality, collections of modules tend to grow, get re-used, etc, making it much more difficult to keep everything in one file. This general trend towards increasing entropy is aggravated by the fact that modules are very self-contained objects that are ideal for re-used. Therefore in the absence of a content management system for LaTeX document (fragments), module collections tend to develop towards the "one module one file" rule, which leads to situations with lots and lots of little files.

Moreover, most mathematical documents are not self-contained, i.e. they do not build up the theory from scratch, but pre-suppose the knowledge (and notation) from other documents. In this case we want to make use of the semantic macros from these prerequisite documents without including their text into the current document. One way to do this would be to have LaTeX read the prerequisite documents without producing output. For efficiency reasons, STeX chooses a different route. It comes with a utility `sms` (see Section 3.1) that exports the modules and macros defined inside them from a particular document and stores them inside `.sms` files. This way we can avoid overloading LaTeX with useless information, while retaining the important information which can then be imported in a more efficient way.

`\importmodule`   For such situations, the `\importmodule` macro can be given an optional first argument that is a path to a file that contains a path to the module file, whose module definition (the `.sms` file) is read. Note that the `\importmodule` macro can be used to make module files truly self-contained. To arrive at a file-based content management system, it is good practice to reuse the module identifiers as module names and to prefix module files with corresponding `\importmodule` statements that pre-load the corresponding module files.

```
\begin{module}[id=foo]
\importmodule[load=../other/bar]{bar}
\importmodule[load=../mycolleaguesmodules]{baz}
\importmodule[load=../other/bar]{foobar}
  ...
\end{module}
```

Example 7: Self-contained Modules via `importmodule`

In Example 7, we have shown the typical setup of a module file. The `\importmodule` macro takes great care that files are only read once, as STeX allows multiple inheritance and this setup would lead to an exponential (in the module inheritance depth) number of file loads.

Sometimes we want to import an existing OMDoc theory[2] $\widehat{\mathcal{T}}$ into (the OM-Doc document $\widehat{\mathcal{D}}$ generated from) a STeX document $\mathcal{D}$. Naturally, we have to provide an STeX stub module $\mathcal{T}$ that provides `\symdef` declarations for all symbols we
`\importOMDocmodule`   use in $\mathcal{D}$. In this situation, we use `\importOMDocmodule[`$\langle spath \rangle$`]{`$\langle OURI \rangle$`}{`$\langle name \rangle$`}`, where $\langle spath \rangle$ is the file system path to $\mathcal{T}$ (as in `\importmodule`, this argument must not contain the file extension), $\langle OURI \rangle$ is the URI to the OMDoc module (this time with extension), and $\langle name \rangle$ is the name of the theory $\widehat{\mathcal{T}}$ and the module in $\mathcal{T}$ (they have to be identical for this to work). Note that since the $\langle spath \rangle$ argument is optional, we can make "local imports", where the stub $\mathcal{T}$ is in $\mathcal{D}$ and only contains the `\symdef`s needed there.

Note that the recursive (depth-first) nature of the file loads induced by this setup is very natural, but can lead to problems with the depth of the file stack in

---

[2]OMDoc theories are the counterpart of STeX modules.

the TeX formatter (it is usually set to something like 15[3]). Therefore, it may be necessary to circumvent the recursive load pattern providing (logically spurious) `\importmodule` commands. Consider for instance module `bar` in Example 7, say that `bar` already has load depth 15, then we cannot naively import it in this way. If module `bar` depended say on a module `base` on the critical load path, then

`\requiremodules`   we could add a statement `\requiremodules{../base}` in the second line. This would load the modules from `../base.sms` in advance (uncritical, since it has load depth 10) without activating them, so that it would not have to be re-loaded in the critical path of the module `foo`. Solving the load depth problem.

`\sinput`   In all of the above, we do not want to load an `sms` file, if the corresponding file has already been loaded, since the semantic macros are already in memory. Therefore the `modules` package supplies a semantic variant of the `\input` macro, which records in an internal register that the modules in the file have already been loaded. Thus if we consistently use `\sinput` instead of `\input` or `\include` for files that contain modules[4], we can prevent double loading of files and therefore

`\sinputref`   gain efficiency. The `\sinputref` macro behaves just like `\sinput` in the LaTeX workflow, but in the LaTeXML conversion process creates a reference to the

`\inputref`
`\includeref`   transformed version of the input file instead. `\inputref` and `\sincluderef` also only create references in the LaTeXML workflow but do not register loaded files or avoid duplicate loading.

Finally, the separation of documents into multiple modules often profits from a symbolic management of file paths. To simplify this, the `modules` package

`\defpath`   supplies the `\defpath` macro: `\defpath[⟨baseURI⟩]{⟨cname⟩}{⟨path⟩}` defines a command, so that `\⟨csname⟩{⟨name⟩}` expands to ⟨path⟩/⟨name⟩. So we could have used

```
\defpath{OPaths}{../other}
\importmodule[load=\OPahts{bar}]{bar}
```

instead of the second line in Example 7. The variant `\OPaths` has the big advantage that we can get around the fact that TeX/LaTeX does not set the current directory in `\input`, so that we can use systematically deployed `\defpath`-defined path macros to make modules relocatable by defining the path macros locally. The optional parameter ⟨baseURI⟩ is for the LaTeXML transformation, which (if ⟨baseURI⟩ is specified) resolves ⟨path⟩ to an absolute URI according to [BFM05, section 5.2].

2.7   Using Semantic Macros in Narrative Structures

The `\importmodule` macro establishes the inheritance relation, a transitive relation among modules that governs visibility of semantic macros. In particular, it can only be used in modules (and has to be used at the top-level, otherwise it

---

[3]If you have sufficient rights to change your TeX installation, you can also increase the variable `max_in_open` in the relevant `texmf.cnf` file. Setting it to 50 usually suffices

[4]files without modules should be treated by the regular LaTeX input mechanism, since they do not need to be registered.

is hindered by LATEX groups). In many cases, we only want to *use* the semantic macros in an environment (and not re-export them). Indeed, this is the normal situation for most parts of mathematical documents. For that sTEX provides the \usemodule macro, which takes the same arguments as \importmodule, but is treated differently in the sTEX module signatures. A typical situation is shown in Figure 8, where we open the module `ring` (see Figure 6) and use its semantic macros (in the `omtext` environment). In earlier versions of sTEX, we would have to wrap the `omtext` environment in an anonymous `module` environment to prevent re-export.

\usemodule

```
\begin{omtext}
  \usemodule[../algebra/rings.tex]{ring}
  We $R$ be a ring $(\rbase,\rplus,\rzero,\rminusOp,\rtimes,\rone)$, ...
\end{omtext}
```

Example 8: Using Semantic Macros in Narrative Structures

Still another import-like relation is the adoption of a module (see [Koh13] for details). We use the \adoptmodule macro for that.[4]

\adoptmodule

EdN:4

2.8 Including Externally Defined Semantic Macros

In some cases, we use an existing LATEX macro package for typesetting objects that have a conventionalized mathematical meaning. In this case, the macros are "semantic" even though they have not been defined by a \symdef. This is no problem, if we are only interested in the LATEX workflow. But if we want to e.g. transform them to OMDOC via LATEXML, the LATEXML bindings will need to contain references to an OMDOC theory that semantically corresponds to the LATEX package. In particular, this theory will have to be imported in the generated OMDOC file to make it OMDOC-valid.

\requirepackage To deal with this situation, the `modules` package provides the \requirepackage macro. It takes two arguments: a package name, and a URI of the corresponding OMDOC theory. In the LATEX workflow this macro behaves like a \usepackage on the first argument, except that it can — and should — be used outside the LATEX preamble. In the LATEXML workflow, this loads the LATEXML bindings of the package specified in the first argument and generates an appropriate `imports` element using the URI in the second argument.

2.9 Views

A view is a mapping between modules, such that all model assumptions (axioms) of the source module are satisfied in the target module. [5]

EdN:5

---

[4]EDNOTE: elaborate this, so that the documentation becomes (more) self-contained.
[5]EDNOTE: Document and make Examples

is hindered by LATEX groups). In many cases, we only want to *use* the semantic macros in an environment (and not re-export them). Indeed, this is the normal situation for most parts of mathematical documents. For that sTEX provides the \usemodule macro, which takes the same arguments as \importmodule, but is treated differently in the sTEX module signatures. A typical situation is shown in Figure 8, where we open the module `ring` (see Figure 6) and use its semantic macros (in the `omtext` environment). In earlier versions of sTEX, we would have to wrap the `omtext` environment in an anonymous `module` environment to prevent re-export.

\usemodule

```
\begin{omtext}
  \usemodule[../algebra/rings.tex]{ring}
  We $R$ be a ring $(\rbase,\rplus,\rzero,\rminusOp,\rtimes,\rone)$, ...
\end{omtext}
```

Example 8: Using Semantic Macros in Narrative Structures

Still another import-like relation is the adoption of a module (see [Koh13] for details). We use the \adoptmodule macro for that.[4]

\adoptmodule

EdN:4

2.8 Including Externally Defined Semantic Macros

In some cases, we use an existing LATEX macro package for typesetting objects that have a conventionalized mathematical meaning. In this case, the macros are "semantic" even though they have not been defined by a \symdef. This is no problem, if we are only interested in the LATEX workflow. But if we want to e.g. transform them to OMDOC via LATEXML, the LATEXML bindings will need to contain references to an OMDOC theory that semantically corresponds to the LATEX package. In particular, this theory will have to be imported in the generated OMDOC file to make it OMDOC-valid.

\requirepackage To deal with this situation, the `modules` package provides the \requirepackage macro. It takes two arguments: a package name, and a URI of the corresponding OMDOC theory. In the LATEX workflow this macro behaves like a \usepackage on the first argument, except that it can — and should — be used outside the LATEX preamble. In the LATEXML workflow, this loads the LATEXML bindings of the package specified in the first argument and generates an appropriate `imports` element using the URI in the second argument.

2.9 Views

A view is a mapping between modules, such that all model assumptions (axioms) of the source module are satisfied in the target module. [5]

EdN:5

---

[4]EDNOTE: elaborate this, so that the documentation becomes (more) self-contained.
[5]EDNOTE: Document and make Examples

2.10 Support for **MathHub**

Much of the STeX content is hosted on MathHub (`http://MathHub.info`), a portal and archive for flexiformal mathematics. MathHub offers GIT repositories (public and private escrow) for mathematical documentation projects, online and offline authoring and document development infrastructure, and a rich, interactive reading interface. The `modules` package supports repository-sensitive operations on MathHub.

Note that MathHub has two-level repository names of the form ⟨*group*⟩/⟨*repo*⟩, where ⟨*group*⟩ is a MathHub-unique repository group and ⟨*repo*⟩ a repository name that is ⟨*group*⟩-unique. The file and directory structure of a repository is arbitrary – except that it starts with the directory `source` because they are Math Archives in the sense of [Hor+11]. But this structure can be hidden from the STeX author with MathHub-enabled versions of the `modules` macros.

\importmhmodule — The `importmhmodule` macro is a variant of `\importmodule` with repository support. Instead of writing

```
\defpath{MathHub}{/user/foo/lmh/MathHub}
\importmodule[load=\MathHub{fooMH/bar/source/baz/foobar}]{foobar}
```

we can simply write (assuming that `\MathHub` is defined as above)

```
\importmhmodule[repos=fooMH/bar,path=baz/foobar]{foobar}
```

Note that the `\importmhmodule` form is more semantic, which allows more advanced document management features in MathHub.

If `baz/foobar` is the "current module", i.e. if we are on the MathHub path `...MathHub/fooMH/bar...`, then stating the repository in the first optional argument is redundant, so we can just use

```
\importmhmodule[path=baz/foobar]{foobar}
```

if no file needs to loaded, `\importmhmodule` is the same as `\importmodule`.

\mhcurrentrepos — Of course, neither LaTeX nor LaTeXMLknow about the repositories when they are called from a file system, so we can use the `\mhcurrentrepos` macro to tell them. But this is only needed to initialize the infrastructure in the driver file. In particular, we do not need to set it in in each module, since the `\importmhmodule` macro sets the current repository automatically.

\usemhmodule — The `\usemhmodule` and `\adoptmhmodule` macros are the analogs to `\usemodule`
\adoptmhmodule — and `\adoptmodule`.

Caveat if you want to use the MathHub support macros (let's call them mh-variants), then every time a module is imported or a document fragment is included from another repos, the mh-variant `\importmhmodule` must be used, so that the "current repository" is set accordingly. To be exact, we only need to use mh-variants, if the imported module or included document fragment use mh-variants.

For this, the `modules` package supplies the mh-variants `\mhinputref` and `\mhinput` of the `\inputref` macro introduced above and normal LaTeX `\input` macro.

3 Limitations & Extensions

In this section we will discuss limitations and possible extensions of the `modules` package. Any contributions and extension ideas are welcome; please discuss ideas, requests, fixes, etc on the sTeX TRAC [sTeX].

3.1 Perl Utility `sms`

Currently we have to use an external perl utility `sms` to extract sTeX module signatures from sTeX files. This considerably adds to the complexity of the sTeX installation and workflow. If we can solve security setting problems that allows us to write to sTeX module signatures outside the current directory, writing them from sTeX may be an avenue of future development see [sTeX, issue #1522] for a discussion.

3.2 Module Signatures loaded even if Modules are

Currently, the module signature ⟨*filepath*⟩`.sms` is loaded even if ⟨*filepath*⟩`.tex` has already been loaded. The Problem is that the `\input` `\inputref` macros do not register the files. I guess, for `\input` we may not want that, but for `\inputref` we should; and maybe we should also have a registering variant `\rinput` that does, then we can leave the choice to the user.

3.3 Only one level of path simplification

It seems that module simplification only covers one level, in particular `foo/bar/../..` is only simplified to `foo/..`, which is not enough in practice. As a consequence, module signatures are loaded unnecessary in the presence of relative paths.

3.4 Qualified Imports

In an earlier version of the `modules` package we used the `usesqualified` for importing macros with a disambiguating prefix (this is used whenever we have conflicting names for macros inherited from different modules). This is not accessible from the current interface. We need something like a `\importqualified` macro for this; see [sTeX, issue #1505]. Until this is implemented the infrastructure
is turned off by default, but we have already introduced the `qualifiedimports` option for the future.

3.5   Error Messages

The error messages generated by the `modules` package are still quite bad. For instance if `thyA` does note exists we get the cryptic error message

```
! Undefined control sequence.
\module@defs@thyA ...hy
                        \expandafter \mod@newcomma...
l.490 ...ortmodule{thyA}
```

This should definitely be improved.

3.6   Crossreferencing

Note that the macros defined by `\symdef` are still subject to the normal TEX scoping rules. Thus they have to be at the top level of a module to be visible throughout the module as intended. As a consequence, the location of the `\symdef` elements cannot be used as targets for crossreferencing, which is currently supplied by the `statement` package [Koh14c]. A way around this limitation would be to import the current module from the STEX module signature (see Section 2.5) via the `\importmodule` declaration.

3.7   No Forward Imports

STEX allows imports in the same file via `\importmodule{⟨mod⟩}`, but due to the single-pass linear processing model of TEX, ⟨mod⟩ must be the name of a module declared *before* the current point. So we cannot have forward imports as in

```
\begin{module}[id=foo]
  \importmodule{mod}
  ...
\end{module}
...
\begin{module}[id=mod]
  ...
\end{module}
```

a workaround, we can extract the module ⟨mod⟩ into a file mod.tex and replace it with `\sinput{mod}`, as in

```
\begin{module}[id=foo]
  \importmodule[load=mod]{mod}
  ...
\end{module}
...
\sinput{mod}
```

then the `\importmodule` command can read `mod.sms` (created via the `sms` utility) without having to wait for the module ⟨mod⟩ to be defined.

4 The Implementation

The `modules` package generates two files: the LaTeX package (all the code between ⟨*package⟩ and ⟨/package⟩) and the LaTeXML bindings (between ⟨*ltxml⟩ and ⟨/ltxml⟩). We keep the corresponding code fragments together, since the documentation applies to both of them and to prevent them from getting out of sync.

4.1 Package Options

We declare some switches which will modify the behavior according to the package options. Generally, an option `xxx` will just set the appropriate switches to true (otherwise they stay false).

```
1 ⟨∗package⟩
2 \DeclareOption{showmeta}{\PassOptionsToPackage{\CurrentOption}{metakeys}}
3 \newif\ifmod@show\mod@showfalse
4 \DeclareOption{showmods}{\mod@showtrue}
5 \newif\ifaux@req\aux@reqtrue
6 \DeclareOption{noauxreq}{\aux@reqfalse}
7 \newif\ifmod@qualified\mod@qualifiedfalse
8 \DeclareOption{qualifiedimports}{\mod@qualifiedtrue}
```

Finally, we need to declare the end of the option declaration section to LaTeX.

```
9 \ProcessOptions
```

LaTeXML does not support module options yet, so we do not have to do anything here for the LaTeXML bindings. We only set up the PERL packages (and tell `emacs` about the appropriate mode for convenience

The next measure is to ensure that the `sref` and `xcomment` packages are loaded (in the right version). For LaTeXML, we also initialize the package inclusions.

```
10 \RequirePackage{sref}
11 \RequirePackage{xspace}
12 \RequirePackage{mdframed}
```

4.2 Modules and Inheritance

We define the keys for the `module` environment and the actions that are undertaken, when the keys are encountered.

module:cd This `KeyVal` key is only needed for LaTeXML at the moment; use this to specify a content dictionary name that is different from the module name.

```
13 \addmetakey{module}{cd}% no longer used
14 \addmetakey{module}{load}% ignored
15 \addmetakey*{module}{title}
16 \addmetakey*{module}{creators}
17 \addmetakey*{module}{contributors}
```

module:id For a module with [id=⟨*name*⟩], we have a macro `\module@defs@`⟨*name*⟩ that acts as a repository for semantic macros of the current module. I will be called by `\importmodule` to activate them. We will add the internal forms of the semantic

macros whenever \symdef is invoked. To do this, we will need an unexpended form \this@module that expands to \module@defs@⟨name⟩; we define it first and then initialize \module@defs@⟨name⟩ as empty. Then we do the same for qualified imports as well (if the qualifiedimports option was specified). Furthermore, we save the module name in \mod@id and the module path in \⟨name⟩@cd@file@base which we add to \module@defs@⟨name⟩, so that we can use it in the importing module.

```
18 \define@key{module}{id}{%
19 % \@ifundefined{module@defs@#1}{}{\PackageError{modules}%
20 %    {The module name #1 has already been used in this scope; please rename it!}}%
21 \edef\this@module{\expandafter\noexpand\csname module@defs@#1\endcsname}%
22 \global\@namedef{module@defs@#1}{}%
23 \ifmod@qualified
24 \edef\this@qualified@module{\expandafter\noexpand\csname module@defs@qualified@#1\endcsname}%
25 \global\@namedef{module@defs@qualified@#1}{}%
26 \fi
27 \def\mod@id{#1}%
28 \expandafter\edef\csname #1@cd@file@base\endcsname{\mod@path}%
29 \expandafter\g@addto@macro\csname module@defs@#1\expandafter\endcsname\expandafter%
30 {\expandafter\def\csname #1@cd@file@base\expandafter\endcsname\expandafter{\mod@path}}}
```

module@heading Then we make a convenience macro for the module heading. This can be customized.

```
31 \newcounter{module}[section]
32 \newcommand\module@heading{\stepcounter{module}%
33 \ifmod@show%
34 \noindent{\textbf{Module} \thesection.\themodule [\mod@id]}%
35 \sref@label@id{Module \thesection.\themodule [\mod@id]}%
36 \ifx\module@title\@empty :\quad\else\quad(\module@title)\hfill\\\fi%
37 \fi}%mod@show
```

module Finally, we define the begin module command for the module environment. All the work has already been done in the keyval bindings, so this is very simple.

```
38 \newenvironment{module}[1][]{\begin{@module}[#1]\module@heading}{\end{@module}}
39 \ifmod@show\surroundwithmdframed{module}\fi
```

@module A variant of the module environment that does not create printed representations (in particular no frames)

```
40 \newenvironment{@module}[1][]{\metasetkeys{module}{#1}}{}
```

\activate@defs To activate the \symdefs from a given module ⟨mod⟩, we call the macro \module@defs@⟨mod⟩. But to make sure that every module is activated only once, we only activate if the macro \module@defs@⟨mod⟩ is undefined, and define it directly afterwards to prohibit further activations.

```
41 \def\activate@defs#1{%
42 \@ifundefined{module@#1@activated}{\csname module@defs@#1\endcsname}{}%
43 \@namedef{module@#1@activated}{true}}
```

\export@defs \export@defs{⟨mod⟩} exports all the \symdefs from module ⟨mod⟩ to the current module (if it has the name ⟨currmod⟩), by adding a call to \module@defs@⟨mod⟩ to the registry \module@defs@⟨currmod⟩.[6]

```
44 \def\export@defs#1{\@ifundefined{mod@id}{}%
45 {\expandafter\expandafter\expandafter\g@addto@macro\expandafter%
46 \this@module\expandafter{\activate@defs{#1}}}}
```

Now we come to the implementation of \importmodule, but before we do, we define two conditionals:

\if@export \if@export distinguishes the \importmodule and \usemodule macros defined below, they differ in whether they re-export macros they import.

```
47 \newif\if@export
```

\if@importing \if@importing can be used to shut up macros in an import situation.

```
48 \newif\if@importing\@importingfalse
```

\importmodule The \importmodule[⟨file⟩]{⟨mod⟩} macro is an interface macro that loads ⟨file⟩ and activates and re-exports the \symdefs from module ⟨mod⟩. It also remembers the file name in \mod@path. As we will need to keep a record of the currently imported modules (top-level only), we divide the functionality into a user-visible macro that records modules in the \imported@modules register and an internal one (\@importmodule) that does the actual work.

```
49 \gdef\imported@modules{}
50 \srefaddidkey{importmodule}
51 \addmetakey{importmodule}{load}
52 \addmetakey[sms]{importmodule}{ext}
53 \newcommand\importmodule[2][]{\metasetkeys{importmodule}{#1}%
54 \ifx\imported@modules\@empty\edef\imported@modules{#2}%
55 \else\edef\imported@modules{#2,\imported@modules}\fi%
56 \@exporttrue\@importmodule[\importmodule@load]{#2}{\importmodule@ext}\ignorespaces}
```

\@importmodule \@importmodule[⟨file⟩]{⟨mod⟩}{⟨ext⟩} loads ⟨file⟩.⟨ext⟩ (if it is given) and activates the module ⟨mod⟩. There is a slight inefficiency in that modules can be recorded multiply, but since \activate@defs is efficient about not activating twice, this is OK. Another thing \@importmodule{⟨mod⟩} does is that it writes a \requiremodules{⟨mod⟩} into the aux file, so that the table of contents can use semantic macros (the table of contents uses \importmodules{\imported@modules} in the titles, see [Koh14b] for details).

```
57 \newcommand\@importmodule[3][]{%
58 {\@importingtrue% to shut up macros
59 \edef\@load{#1}\ifx\@load\@empty\else{\mod@showfalse\requiremodules\@load{#3}}\fi%
60 \activate@defs{#2}\if@export\export@defs{#2}\fi}
```

---

[6]EDNOTE: MK: I have the feeling that we may be exporting modules multiple times here, is that a problem?

\usemodule	\usemodule acts like \importmodule for the LaTeX side, except that the sms utility does not transfer it to the module signatures and it does not export the symdefs.

```
61 \newcommand\usemodule[2][]{\@exportfalse\importmodule[#1]{#2}}
```

\importmodules	This variant just imports all the modules in a comma-separated list (usually \imported@modules)

```
62 \newcommand\importmodules[1]{\@for\@I:=#1\do{\importmodule\@I}}%
```

\importmodulevia	The importmodulevia environment just calls \importmodule, but to get around the group, we first define a local macro \@@doit, which does that and can be called with an \aftergroup to escape the environment grouping introduced by importmodulevia.

```
63 \newenvironment{importmodulevia}[2][]{\gdef\@@doit{\importmodule[#1]{#2}}%
64 \ifmod@show\par\noindent importing module #2 via \@@doit\fi}
65 {\aftergroup\@@doit\ifmod@show end import\fi}
```

vassign	

```
66 \newcommand\vassign[3][]{\ifmod@show\ensuremath{#2\mapsto #3}, \fi}
```

tassign	

```
67 \newcommand\tassign[3][]{\ifmod@show #2\ensuremath{\mapsto} #3, \fi}
```

ttassign	

```
68 \newcommand\ttassign[3][]{\ifmod@show #2\ensuremath{\mapsto} ``#3'', \fi}
```

\importOMDocmodule	for the LaTeX side we can just re-use \importmodule, for the LaTeXML side we have a full URI anyways. So things are easy.[7]

EdN:7

```
69 \newcommand\importOMDocmodule[3][]{\importmodule[#1]{#3}}
```

\metalanguage	\metalanguage behaves exactly like \importmodule for formatting. For LaTeXML, we only add the type attribute.

```
70 \let\metalanguage=\importmodule
```

\adoptmodule	\adoptmodule macro behaves exactly like \importmodule for formatting. For LaTeXML.

```
71 \let\adoptmodule=\importmodule
```

4.3 Semantic Macros

\mod@newcommand	We first hack the LaTeX kernel macros to obtain a version of the \newcommand macro that does not check for definedness.

```
72 \let\mod@newcommand=\providerobustcmd
```

Now we define the optional KeyVal arguments for the \symdef form and the actions that are taken when they are encountered.

---

[7]EdNote: MK@DG: this macro is seldom used, maybe I should just switch arguments.

```
73 \srefaddidkey{conceptdef}
74 \addmetakey*{conceptdef}{title}
75 \addmetakey{conceptdef}{subject}
76 \addmetakey*{conceptdef}{display}
77 \def\conceptdef@type{Symbol}
78 \newcommand\conceptdef[2][]{\metasetkeys{conceptdef}{#1}%
79 \ifx\conceptdef@display\st@flow\else{\stDMemph{\conceptdef@type} #2:}\fi%
80 \ifx\conceptdef@title\@empty~\else~(\stDMemph{\conceptdef@title})\par\fi}
```

EdN:8

8

symdef:keys   The optional argument local specifies the scope of the function to be defined. If
local is not present as an optional argument then \symdef assumes the scope of
the function is global and it will include it in the pool of macros of the current
module. Otherwise, if local is present then the function will be defined only
locally and it will not be added to the current module (i.e. we cannot inherit
a local function). Note, the optional key local does not need a value: we write
\symdef[local]{somefunction}[0]{some expansion}. The other keys are not
used in the LaTeX part.

```
81 \newif\if@symdeflocal
82 \srefaddidkey{symdef}
83 \define@key{symdef}{local}[true]{\@symdeflocaltrue}
84 \define@key{symdef}{primary}[true]{}
85 \define@key{symdef}{assocarg}{}
86 \define@key{symdef}{bvars}{}
87 \define@key{symdef}{bargs}{}
88 \addmetakey{symdef}{name}
89 \addmetakey*{symdef}{title}
90 \addmetakey*{symdef}{description}
91 \addmetakey{symdef}{subject}
92 \addmetakey*{symdef}{display}
```

EdN:9

9

\symdef   The the \symdef, and \@symdef macros just handle optional arguments and hand
off processing for in math mode to \@@symdef.

```
93 \def\symdef{\@ifnextchar[{\@symdef}{\@symdef[]}}
94 \def\@symdef[#1]#2{\@ifnextchar[{\@@symdef[#1]{#2}{math}}{\@@symdef[#1]{#2}{math}[0]}}
```

next we locally abbreviate \mod@newcommand to simplify argument passing.

```
95 \def\@mod@nc#1{\mod@newcommand{#1}[1]}
```

\@@symdef   now comes the real meat: the \@@symdef macro does two things, it adds the macro
definition to the macro definition pool of the current module and also provides it.

```
96 \def\@@symdef[#1]#2#3[#4]#5{%
```

---

[8]EDNOTE: MK@DG: maybe we need to add DefKeyVals here?
[9]EDNOTE: MK@MK: we need to document the binder keys above.

We use a switch to keep track of the local optional argument. We initialize the switch to false and set all the keys that have been provided as arguments: `name`, `local`.

97 `\@symdeflocalfalse\metasetkeys{symdef}{#1}%`

First, using `\mod@newcommand` we initialize the intermediate macro `\modules@⟨sym⟩@⟨mode⟩@`, the one that can be extended with `\symvariant`

98 `\expandafter\mod@newcommand\csname modules@#2@#3@\endcsname[#4]{#5}%`

and then we define the actual semantic macro `\⟨sym⟩`, which when invoked with an optional argument `⟨opt⟩` calls `\modules@⟨sym⟩@⟨mode⟩@⟨opt⟩` provided by the `\symvariant` macro.

99 `\expandafter\mod@newcommand\csname #2\endcsname[1][]%`
100 `{\ifmmode\csname modules@#2@math@##1\endcsname%`
101 `\else\csname modules@#2@text@##1\endcsname\fi}%`

Finally, we prepare the internal macro to be used in the `\symref` call.

102 `\expandafter\@mod@nc\csname mod@symref@#2\expandafter\endcsname\expandafter%`
103 `{\expandafter\mod@termref\expandafter{\mod@id}{#2}{##1}}%`

We check if the switch for the local scope is set: if it is we are done, since this function has a local scope. Similarly, if we are not inside a module, which we could export from.

104 `\if@symdeflocal\else%`
105 `\@ifundefined{mod@id}{}{%`

Otherwise, we add three functions to the module's pool of defined macros using `\g@addto@macro`. We first add the definition of the intermediate function `\modules@⟨sym⟩@⟨mode⟩@`.

106 `\expandafter\g@addto@macro\this@module%`
107 `{\expandafter\mod@newcommand\csname modules@#2@#3@\endcsname[#4]{#5}}%`

Then we add add the definition of `\⟨sym⟩` which calls the intermediate function and handles the optional argument.

108 `\expandafter\g@addto@macro\this@module%`
109 `{\expandafter\mod@newcommand\csname #2\endcsname[1][]%`
110 `{\ifmmode\csname modules@#2@math@##1\endcsname%`
111 `\else\csname modules@#2@text@##1\endcsname\fi}}%`

We also add `\mod@symref@⟨sym⟩` macro to the macro pool so that the `\symref` macro can pick it up.

112 `\expandafter\g@addto@macro\csname  module@defs@\mod@id\expandafter\endcsname\expandafter%`
113 `{\expandafter\@mod@nc\csname mod@symref@#2\expandafter\endcsname\expandafter%`
114 `{\expandafter\mod@termref\expandafter{\mod@id}{#2}{##1}}}%`

Finally, using `\g@addto@macro` we add the two functions to the qualified version of the module if the `qualifiedimports` option was set.

115 `\ifmod@qualified%`
116 `\expandafter\g@addto@macro\this@qualified@module%`
117 `{\expandafter\mod@newcommand\csname modules@#2@#3@qualified\endcsname[#4]{#5}}%`
118 `\expandafter\g@addto@macro\this@qualified@module%`
119 `{\expandafter\def\csname#2atqualified\endcsname{\csname modules@#2@#3@qualified\endcsname}}%`

20

```
120 \fi}% mod@qualified
121 \fi% symdeflocal
```

So now we only need to show the data in the symdef, if the options allow.

```
122 \ifmod@show
123 \ifx\symdef@display\st@flow\else{\noindent\stDMemph{\symdef@type} #2:}\fi%
124 \ifx\symdef@title\@empty~\else~(\stDMemph{\symdef@title})\par\fi
125 \fi}% mod@show
126 \def\symdef@type{Symbol}
```

\symvariant   \symvariant{⟨sym⟩}[⟨args⟩]{⟨var⟩}{⟨cseq⟩} just defines a variant \modules@⟨sym⟩@⟨mode⟩@⟨var⟩
which expands to ⟨cseq⟩. Recall that this is called by the macro \⟨sym⟩[⟨var⟩]

induced by the \symdef. [10]

```
127 \def\symvariant#1{\@ifnextchar[{\@symvariant{#1}{math}}{\@symvariant{#1}{math}[0]}}
128 \def\@symvariant#1#2[#3]#4#5{%
129 \expandafter\mod@newcommand\csname modules@#1@#2@#4\endcsname[#3]{#5}%
```

and if we are in a named module, then we need to export the function
\modules@⟨sym⟩@⟨mode⟩@⟨opt⟩ just as we have done that in \symdef.

```
130 \@ifundefined{mod@id}{}{%
131 \expandafter\g@addto@macro\this@module%
132 {\expandafter\mod@newcommand\csname modules@#1@#2@#4\endcsname[#3]{#5}}}}%
```

\resymdef   This is now deprecated.

```
133 \def\resymdef{\@ifnextchar[{\@resymdef}{\@resymdef[]}}
134 \def\@resymdef[#1]#2{\@ifnextchar[{\@@resymdef[#1]{#2}}{\@@resymdef[#1]{#2}[0]}}
135 \def\@@resymdef[#1]#2[#3]#4{\PackageError{modules}
136   {The \protect\resymdef macro is deprecated,\MessageBreak
137    use the \protect\symvariant instead!}}
```

\abbrdef   The \abbrdef macro is a variant of \symdef that does the same on the LATEX
level.

```
138 \let\abbrdef\symdef
```

4.4   Defining Math Operators

\DefMathOp   [11]

```
139 \define@key{DefMathOp}{name}{\def\defmathop@name{#1}}
140 \newcommand\DefMathOp[2][]{%
141 \setkeys{DefMathOp}{#1}%
142 \symdef[#1]{\defmathop@name}{#2}}
```

4.5   Testing Semantic Macros

\symtest   Allows to test a \symdef in place, this shuts up when being imported.

```
143 \addmetakey{symtest}{name}
```

---

[10]EdNote: MK@DG: this needs to be implemented in LaTeXML
[11]EdNote: MK@MK,DG: DefMathOp needs to be documented above, what can we do with it?

```
144 \addmetakey{symtest}{variant}
145 \newcommand\symtest[3][]{\if@importing\else%
146 \metasetkeys{symtest}{#1}%
147 \par\noindent \textbf{Symbol}~%
148 \ifx\symtest@name\@empty\texttt{#2}\else\texttt{\symtest@name}\fi%
149 \ifx\symtest@variant\@empty\else\ (variant \texttt{\symtest@variant})\fi%
150 \ with semantic macro %
151 \texttt{\textbackslash #2\ifx\symtest@variant\@empty\else[\symtest@variant]\fi}%
152 : used e.g. in \ensuremath{#3}%
153 \fi}
```

\abbrtest

```
154 \addmetakey{abbrtest}{name}
155 \newcommand\abbrtest[3][]{\if@importing\else%
156 \metasetkeys{abbrtest}{#1}%
157 \par\noindent \textbf{Abbreviation}~%
158 \ifx\abbrtest@name\@empty\texttt{#2}\else\texttt{\abbrtest@name}\fi%
159 : used e.g. in \ensuremath{#3}%
160 \fi}
```

4.6   Symbol and Concept Names

\mod@path   the \mod@path macro is used to remember the local path, so that the `module` environment can set it for later cross-referencing of the modules. If \mod@path is empty, then it signifies the local file.

```
161 \def\mod@path{}
```

\termdef

```
162 \def\mod@true{true}
163 \addmetakey[false]{termdef}{local}
164 \addmetakey{termdef}{name}
165 \newcommand\termdef[3][]{\metasetkeys{termdef}{#1}%
166 \expandafter\mod@newcommand\csname#2\endcsname[0]{#3\xspace}%
167 \ifx\termdef@local\mod@true\else%
168 \@ifundefined{mod@id}{}{\expandafter\g@addto@macro\this@module%
169 {\expandafter\mod@newcommand\csname#2\endcsname[0]{#3\xspace}}}%
170 \fi}
```

\capitalize

```
171 \def\@capitalize#1{\uppercase{#1}}
172 \newcommand\capitalize[1]{\expandafter\@capitalize #1}
```

\mod@termref   \mod@termref{⟨module⟩}{⟨name⟩}{⟨nl⟩} determines whether the macro \⟨module⟩@cd@file@base is defined. If it is, we make it the prefix of a URI reference in the local macro \@uri, which we compose to the hyper-reference, otherwise we give a warning.

```
173 \def\mod@termref#1#2#3{\def\@test{#3}%
174 \@ifundefined{#1cd@file@base}%
175 {\protect\G@refundefinedtrue%
```

```
176 \@latex@warning{\protect\termref with unidentified cd "#1": the cd key must reference an active
177 \def\@label{sref@#2 @target}}%
178 {\def\@label{sref@#2@#1@target}}%
179 \ifx\csname #1@cd@file@base\endcsname\@empty% local reference
180 \sref@hlink@ifh{\@label}{\ifx\@test\@empty #2\else #3\fi}\else%
181 \def\@uri{\csname #1@cd@file@base\endcsname.pdf\#\@label}%
182 \sref@href@ifh{\@uri}{\ifx\@test\@empty #2\else #3\fi}\fi}
```

4.7 Dealing with Multiple Files

Before we can come to the functionality we want to offer, we need some auxiliary functions that deal with path names.

4.7.1 Simplifying Path Names

The `\mod@simplify` macro is used for simplifying path names by removing ⟨*xxx*⟩/.. from a string. eg: ⟨*aaa*⟩/⟨*bbb*⟩/../⟨*ddd*⟩ goes to ⟨*aaa*⟩/⟨*ddd*⟩ unless ⟨*bbb*⟩ is ... This is used to normalize relative path names below.

`\mod@simplify` The macro `\mod@simplify` recursively runs over the path collecting the result in the internal `\mod@savedprefix` macro.

```
183 \def\mod@simplify#1{\expandafter\mod@simpl#1/\relax}
```

EdN:12 It is based on the `\mod@simpl` macro[12]

```
184 \def\mod@simpl#1/#2\relax{\def\@second{#2}%
185 \ifx\mod@blaaaa\@empty\edef\mod@savedprefix{}\def\mod@blaaaa{aaa}\fi%
186 \ifx\@second\@empty\edef\mod@savedprefix{\mod@savedprefix#1}%
187 \else\mod@simplhelp#1/#2\relax\fi}
```

which in turn is based on a helper macro

```
188 \def\mod@updir{..}
189 \def\mod@simplhelp#1/#2/#3\relax{\def\@first{#1}\def\@second{#2}\def\@third{#3}%
190 %\message{mod@simplhelp: first=\@first, second=\@second, third=\@third, result=\mod@savedprefix
191 \ifx\@third\@empty% base case
192 \ifx\@second\mod@updir\else%
193 \ifx\@second\@empty\edef\mod@savedprefix{\mod@savedprefix#1}%
194 \else\edef\mod@savedprefix{\mod@savedprefix#1/#2}%
195 \fi% @second empty
196 \fi% @second = ..
197 \else% non-base case: @third non-empty
198 \ifx\@first\mod@updir%
199 \edef\mod@savedprefix{\mod@savedprefix#1/}\mod@simplhelp#2/#3\relax%
200 \else% @first != ..
201 \ifx\@second\mod@updir\mod@simpl#3\relax%
202 \else\edef\mod@savedprefix{\mod@savedprefix#1/}\mod@simplhelp#2/#3\relax%
203 \fi% @first
204 \fi% @first = ..
205 \fi}% non-base case
```

---

[12]EDNOTE: what does the mod@blaaa do?; it seems to be a switch that checks if mod@savedprefix has been saved.

We directly test the simplification:

source	result	should be
aaa	aaa	aaa
../../aaa	../../aaa	../../aaa
aaa/bbb	aaa/bbb	aaa/bbb
aaa/..		
../../aaa/bbb	../../aaa/bbb	../../aaa/bbb
../aaa/../bbb	../bbb	../bbb
../aaa/bbb	../aaa/bbb	../aaa/bbb
aaa/bbb/../ddd	aaa/ddd	aaa/ddd
aaa/bbb/../..	aaa/..	

\defpath

```
206 \newcommand\defpath[3][]{\expandafter\newcommand\csname #2\endcsname[1]{#3/##1}}
```

4.8 Loading Module Signatures

EdN:13 We will need a switch[13]

```
207 \newif\ifmodules
```

and a "registry" macro whose expansion represents the list of added macros (or files)

\mod@reg We initialize the \mod@reg macro with the empty string.

```
208 \gdef\mod@reg{}
```

\mod@update This macro provides special append functionality. \mod@update{⟨string⟩} takes appends ⟨string⟩ to the expansion of the \mod@reg macro in the following way: ⟨string⟩@\mod@reg.

```
209 \def\mod@update#1{\ifx\mod@reg\@empty\xdef\mod@reg{#1}%
210 \else\xdef\mod@reg{#1@\mod@reg}\fi}
```

\mod@check The \mod@check takes as input a file path (arg 3), and searches the registry. If the file path is not in the registry it means it means it has not been already added, so we make \ifmodules true, otherwise make \ifmodules false. The macro \mod@search will look at \ifmodules and update the registry for \modulestrue or do nothing for \modulesfalse.

```
211 \def\mod@check#1@#2///#3\relax{%
212 \def\mod@one{#1}\def\mod@two{#2}\def\mod@three{#3}%
```

Define a few intermediate macros so that we can split the registry into separate file paths and compare to the new one

```
213 \expandafter%
214 \ifx\mod@three\mod@one\modulestrue%
215 \else%
216 \ifx\mod@two\@empty\modulesfalse\else\mod@check#2///#3\relax\fi%
217 \fi}
```

---

[13]EDNOTE: explain why?

\mod@search Macro for updating the registry after the execution of \mod@check

```
218 \def\mod@search#1{%
```

We put the registry as the first argument for \mod@check and the other argument is the new file path.

```
219 \modulesfalse\expandafter\mod@check\mod@reg @///#1\relax%
```

We run \mod@check with these arguments and the check \ifmodules for the result

```
220 \ifmodules\else\mod@update{#1}\fi}
```

\mod@reguse The macro operates almost as the mod@search function, but it does not update the registry. Its purpose is to check whether some file is or not inside the registry but without updating it. Will be used before deciding on a new sms file

```
221 \def\mod@reguse#1{\modulesfalse\expandafter\mod@check\mod@reg @///#1\relax}
```

\mod@prefix This is a local macro for storing the path prefix, we initialize it as the empty string.

```
222 \def\mod@prefix{}
```

\mod@updatedpre This macro updates the path prefix \mod@prefix with the last word in the path given in its argument.

```
223 \def\mod@updatedpre#1{%
224 \edef\mod@prefix{\mod@prefix\mod@pathprefix@check#1/\relax}}
```

\mod@pathprefix@check \mod@pathprefix@check returns the last word in a string composed of words separated by slashes

```
225 \def\mod@pathprefix@check#1/#2\relax{%
226 \ifx\\#2\\% no slash in string
227 \else\mod@ReturnAfterFi{#1/\mod@pathprefix@help#2\relax}%
228 \fi}
```

It needs two helper macros:

```
229 \def\mod@pathprefix@help#1/#2\relax{%
230 \ifx\\#2\\% end of recursion
231 \else\mod@ReturnAfterFi{#1/\mod@pathprefix@help#2\relax}%
232 \fi}
233 \long\def\mod@ReturnAfterFi#1\fi{\fi#1}
```

\mod@pathpostfix@check \mod@pathpostfix@check takes a string composed of words separated by slashes and returns the part of the string until the last slash

```
234 \def\mod@pathpostfix@check#1/#2\relax{% slash
235 \ifx\\#2\\%no slash in string
236 #1\else\mod@ReturnAfterFi{\mod@pathpostfix@help#2\relax}%
237 \fi}
```

Helper function for the \pathpostfix@check macro defined above

```
238 \def\mod@pathpostfix@help#1/#2\relax{%
239 \ifx\\#2\\%
240 #1\else\mod@ReturnAfterFi{\mod@pathpostfix@help#2\relax}%
241 \fi}
```

\mod@updatedpost   This macro updates `\mod@savedprefix` with leading path (all but the last word) in the path given in its argument.

```
242 \def\mod@updatedpost#1{%
243 \edef\mod@savedprefix{\mod@savedprefix\mod@pathpostfix@check#1/\relax}}
```

4.8.1   Selective Inclusion

\requiremodules   this macro loads the modules in a file. It does quite a lot of work to make sure that they are only loaded once (actually this does not quite work so they are loaded only twice). `\requiremodules{⟨file⟩}{⟨ext⟩}` also deposits a call to `\@requiremodules{⟨file⟩}{⟨ext⟩}` to the aux file to make sure that the semantic macros are already loaded for the table of contents.

```
244 \newcommand\requiremodules[2]{%
245 {\mod@showfalse% save state and ensure silence while reading sms
246 \mod@updatedpre{#1}% add the new file to the already existing path
247 \let\mod@savedprefix\mod@prefix% add the path of the new file to the prefix
248 \mod@updatedpost{#1}%
249 \def\mod@blaaaa{}% macro used in the simplify function (remove .. from the prefix)
250 \mod@simplify{\mod@savedprefix}% remove |xxx/..| from the path (in case it exists)
251 \mod@reguse{\mod@savedprefix}%
252 \ifmodules\else%
253 \let\newreg\mod@reg% use to compare, in case the .sms file was loaded before
254 \mod@search{\mod@savedprefix}% update registry
255 \ifx\newreg\mod@reg\else % load if not in registry
256 \ifaux@req\immediate\write\@auxout{\string\@requiremodules{\mod@savedprefix}{#2}}\fi%
257 \input{\mod@savedprefix.#2}%
258 \fi% load
259 \fi}}%  ifmodules
```

\@requiremodules   the internal macro that actually does the loading. We disable it at the end of the document, so that when the aux file is read again, nothing is loaded.

```
260 \newcommand\@requiremodules[2]{\if@tempswa\mod@showfalse\@importingtrue\input{#1.#2}\fi}
```

\sinput

```
261 \def\sinput#1{
262 {\mod@updatedpre{#1}% add the new file to the already existing path
263 \let\mod@savedprefix\mod@prefix% add the path to the new file to the prefix
264 \mod@updatedpost{#1}%
265 \def\mod@blaaaa{}% macro used in the simplify function (remove .. from the prefix)
266 \mod@simplify{\mod@savedprefix}% remove |xxx/..| from the path (in case it exists)
267 \mod@reguse{\mod@savedprefix}%
268 \let\newreg\mod@reg% use to compare, in case the .sms file was loaded before
269 \mod@search{\mod@savedprefix}% update registry
270 \ifx\newreg\mod@reg%\message{This file has been previously introduced}
271 \else\input{\mod@savedprefix}%
272 \fi}}
```

```
273 \let\sinputref=\sinput
274 \let\inputref=\input
275 \let\includeref=\include
```

4.9   Including Externally Defined Semantic Macros

\requirepackage

```
276 \def\requirepackage#1#2{\makeatletter\input{#1.sty}\makeatother}
```

4.10   Views

We first prepare the ground by defining the keys for the `view` environment.

```
277 \srefaddidkey{view}
278 \addmetakey*{view}{title}
279 \addmetakey{view}{display}
280 \addmetakey{view}{from}
281 \addmetakey{view}{to}
282 \addmetakey{view}{creators}
283 \addmetakey{view}{contributors}
284 \addmetakey[sms]{view}{ext}
```

\view@heading   Then we make a convenience macro for the view heading. This can be customized.

```
285 \newcounter{view}[section]
286 \newcommand\view@heading[4]{\if@importing\else%
287 \stepcounter{view}%
288 \edef\@display{#3}\edef\@title{#4}%
289 \noindent\ifx\@display\st@flow\else
290 {\textbf{View} {\thesection.\theview} from \textsf{#1} to \textsf{#2}}%
291 \sref@label@id{View \thesection.\theview}%
292 \ifx\@title\@empty\quad\else\quad(\@title)\fi%
293 \par\noindent%
294 \fi\ignorespaces\fi}%ifmod@show
```

view   The `view` environment relies on the `@view` environment (used also in the STEX module signatures) for module bookkeeping and adds presentation (a heading and a box) if the `showmods` option is set.

```
295 \newenvironment{view}[3][]%
296 {\metasetkeys{view}{#1}\sref@target%
297 \begin{@view}{#2}{#3}\view@heading{#2}{#3}{\view@display}{\view@title}}
298 {\end{@view}\ignorespaces}
299 \ifmod@show\surroundwithmdframed{view}\fi
```

@view   The `@view` does the actual bookkeeping at the module level.

```
300 \newenvironment{@view}[2]{%from, to
```

```
301 \@importmodule[\view@from]{#1}{\view@ext}%
302 \@importmodule[\view@to]{#2}{\view@ext}}
303 {}
```

viewsketch   The `viewsketch` environment behaves like `view`, but only has text contents.

```
304 \newenvironment{viewsketch}[3][]%
305 {\metasetkeys{view}{#1}\sref@target%
306 \begin{@view}{#2}{#3}\view@heading{#2}{#3}{\view@display}{\view@title}}
307 {\end{@view}}
308 \ifmod@show\surroundwithmdframed{viewsketch}\fi
```

4.11   Support for **MathHub**

\importmhmodule   The `\importmhmodule` saves the current value of `\mh@currentrepos` in a local macro `\mh@@repos`, resets `\mh@currentrepos` to the new value if one is given in the optional argument, and after importing resets `\mh@currentrepos` to the old value in `\mh@@repos`. We do all the `\ifx` comparison with an `\expandafter`, since the values may be passed on from other key bindings.

```
309 \srefaddidkey{importmhmodule}
310 \addmetakey{importmhmodule}{repos}
311 \addmetakey{importmhmodule}{path}
312 \addmetakey[sms]{importmhmodule}{ext}
313 \newcommand\importmhmodule[2][]{\metasetkeys{importmhmodule}{#1}%
314 \ifx\importmhmodule@path\@empty
315 \importmodule[ext=\importmhmodule@ext,id=\importmhmodule@id]{#2}\else
316 \edef\mh@@repos{\mh@currentrepos}%
317 \ifx\importmhmodule@repos\@empty\else\mhcurrentrepos{\importmhmodule@repos}\fi%
318 \importmodule[load=\MathHub{\mh@currentrepos/source/\importmhmodule@path},
319           ext=\importmhmodule@ext,id=\importmhmodule@id]{#2}%
320 \mhcurrentrepos\mh@@repos\fi\ignorespaces}
```

and now the analogs

\usemhmodule

```
321 \newcommand\usemhmodule[2][]{\metasetkeys{importmhmodule}{#1}%
322 \ifx\importmhmodule@path\@empty
323 \usemodule[ext=\importmhmodule@ext,id=\importmhmodule@id]{#2}\else
324 \edef\mh@@repos{\mh@currentrepos}%
325 \ifx\importmhmodule@repos\@empty\else\mhcurrentrepos{\importmhmodule@repos}\fi%
326 \usemodule[load=\MathHub{\mh@currentrepos/source/\importmhmodule@path},
327           ext=\importmhmodule@ext,id=\importmhmodule@id]{#2}%
328 \mhcurrentrepos\mh@@repos\fi\ignorespaces}
```

\adoptmhmodule

```
329 \newcommand\adoptmhmodule[2][]{\metasetkeys{importmhmodule}{#1}%
330 \ifx\importmhmodule@path\@empty
331 \adoptmodule[ext=\importmhmodule@ext,id=\importmhmodule@id]{#2}\else
332 \edef\mh@@repos{\mh@currentrepos}%
333 \ifx\importmhmodule@repos\@empty\else\mhcurrentrepos{\importmhmodule@repos}\fi%
```

```
334 \adoptmodule[load=\MathHub{\mh@currentrepos/source/\importmhmodule@path},
335            ext=\importmhmodule@ext,id=\importmhmodule@id]{#2}%
336 \mhcurrentrepos\mh@@repos\fi\ignorespaces}
```

\mhinputref

```
337 \newcommand\mhinputref[2][]{\def\@repos{#1}%
338 \edef\mh@@repos{\mh@currentrepos}%
339 \ifx\@repos\@empty\else\mhcurrentrepos{#1}\fi%
340 \inputref{\MathHub{\mh@currentrepos/source/#2}}%
341 \mhcurrentrepos\mh@@repos\ignorespaces}
```

\mhinput

```
342 \let\mhinput\mhinputref
```

\mhcurrentrepos \mhcurrentrepos is used to initialize the current repository. If the repos has
\@mhcurrentrepos changed, it writes a call to the internal macro \@mhcurrentrepos fo the aux file
and calls it. So that the \importmodule calls there work with the correct repos.

```
343 \newcommand\mhcurrentrepos[1]{\edef\@test{#1}%
344 \ifx\@test\mh@currentrepos\else%
345 \immediate\write\@auxout{\string\@mhcurrentrepos{#1}}\fi%
346 \@mhcurrentrepos{#1}}
347 \newcommand\@mhcurrentrepos[1]{\edef\mh@currentrepos{#1}}
```

importmhmodulevia

```
348 \newenvironment{importmhmodulevia}[3][]{\gdef\@@doit{\importmhmodule[#1]{#2}{#3}}%
349 \ifmod@show\par\noindent importing module #2 via \@@doit\fi}
350 {\aftergroup\@@doit\ifmod@show end import\fi}
```

```
351 \srefaddidkey{mhview}
352 \addmetakey{mhview}{display}
353 \addmetakey{mhview}{creators}
354 \addmetakey{mhview}{contributors}
355 \addmetakey*{mhview}{title}
356 \addmetakey{mhview}{fromrepos}
357 \addmetakey{mhview}{torepos}
358 \addmetakey{mhview}{frompath}
359 \addmetakey{mhview}{topath}
360 \addmetakey[sms]{mhview}{ext}
```

mhview   the MathHub version

```
361 \newenvironment{mhview}[3][]% keys, from, to
362 {\metasetkeys{mhview}{#1}\sref@target%
363 \begin{@mhview}{#2}{#3}\view@heading{#2}{#3}{\mhview@display}{\mhview@title}}
364 {\end{@mhview}\ignorespaces}
365 \ifmod@show\surroundwithmdframed{mhview}\fi
```

@mhview   The @mhview does the actual bookkeeping at the module level.

```
366 \newenvironment{@mhview}[2]{%from, to
367 \importmhmodule[repos=\mhview@fromrepos,path=\mhview@frompath,ext=\mhview@ext]{#1}%
```

```
368 \importmhmodule[repos=\mhview@torepos,path=\mhview@topath,ext=\mhview@ext]{#2}}
369 {}
```

mhviewsketch   The `mhviewsketch` environment behaves like `mhview`, but only has text contents.

```
370 \newenvironment{mhviewsketch}[3][]%
371 {\metasetkeys{mhview}{#1}\sref@target%
372 \begin{@mhview}{#2}{#3}\view@heading{#2}{#3}{\mhview@display}{\mhview@title}}
373 {\end{@mhview}\ignorespaces}
374 \ifmod@show\surroundwithmdframed{mhviewsketch}\fi
```

      `\obligation`   The `\obligation` element does not do anything yet on the latexml side.[15]

```
375 \newcommand\obligation[3][]{\if@importing\else Axiom #2 is proven by \sref{#3}\fi}
```

4.12 Deprecated Functionality

In this section we centralize old interfaces that are only partially supported any more.

module:uses   For each the module name `xxx` specified in the `uses` key, we activate their symdefs
    and we export the local symdefs.[16]

```
376 \define@key{module}{uses}{%
377 \@for\module@tmp:=#1\do{\activate@defs\module@tmp\export@defs\module@tmp}}
```

module:usesqualified   This option operates similarly to the module:uses option defined above. The only difference is that here we import modules with a prefix. This is useful when two modules provide a macro with the same name.

```
378 \define@key{module}{usesqualified}{%
379 \@for\module@tmp:=#1\do{\activate@defs{qualified@\module@tmp}\export@defs\module@tmp}}
```

\coolurion/off

```
380 \def\coolurion{\PackageWarning{modules}{coolurion is obsolete, please remove}}
381 \def\coolurioff{\PackageWarning{modules}{coolurioff is obsolete, please remove}}
```

4.13 Experiments

In this section we develop experimental functionality. Currently support for complex expressions, see `https://svn.kwarc.info/repos/stex/doc/blue/comlex_semmacros/note.pdf` for details.

\csymdef   For the LATEX we use `\symdef` and forget the last argument. The code here is just needed for parsing the (non-standard) argument structure.

```
382 \def\csymdef{\@ifnextchar[{\@csymdef}{\@csymdef[]}}
383 \def\@csymdef[#1]#2{\@ifnextchar[{\@@csymdef[#1]{#2}}{\@@csymdef[#1]{#2}[0]}}
384 \def\@@csymdef[#1]#2[#3]#4#5{\@@symdef[#1]{#2}[#3]{#4}}
```

---

[15]EDNOTE: document above
[16]EDNOTE: this issue is deprecated, it will be removed before 1.0.

\notationdef   For the LaTeX side, we just make `\notationdef` invisible.

```
385 \def\notationdef[#1]#2#3{}
```

The code for avoiding duplicate loading is very very complex and brittle (and does not quite work). Therefore I would like to replace it with something better. It has two parts:

- keeping a registry of file paths, and only loading when the file path has not been mentioned in that, and

- dealing with relative paths (for that we have to string together prefixes and pass them one)

For the first problem, there is a very nice and efficient solution using `etoolbox` which I document below. If I decide to do away with relative paths, this would be it.

\reqmodules   We keep a file path registry `\@register` and only load a module signature, if it is not in there.

```
386 \newcommand\reqmodules[2]{\ifinlist{#1}{\@register}{}{\listadd\@register{#1}\input{#1.#2}}}
```

for the relative paths, I have to find out the directory prefix and the file name. Here are two helper functions, which work well, but do not survive being called in an `\edef`, which is what we would need. First some preparation: we set up a path parser

```
387 \newcounter{@pl}
388 \DeclareListParser*{\forpathlist}{/}
```

\file@name   `\file@name` selects the filename of the file path: `\file@name{/foo/bar/baz.tex}` is `baz.tex`.

```
389 \def\file@name#1{\setcounter{@pl}{0}%
390 \forpathlist{\stepcounter{@pl}\listadd\@pathlist}{#1}
391 \def\do##1{\ifnumequal{\value{@pl}}{1}{##1}{\addtocounter{@pl}{-1}}}
392 \dolistloop{\@pathlist}}
```

\file@path   `\file@path` selects the path of the file path `\file@path{/foo/bar/baz.tex}` is `/foo/bar`

```
393 \def\file@path#1{\setcounter{@pl}{0}%
394 \forpathlist{\stepcounter{@pl}\listadd\@pathlist}{#1}
395 \def\do##1{\ifnumequal{\value{@pl}}{1}{}{\addtocounter{@pl}{-1}%
396 \ifnumequal{\value{@pl}}{1}{##1}{##1/}}}
397 \dolistloop{\@pathlist}}
398 ⟨/package⟩
```

what I would really like to do in this situation is

\NEWrequiremodules   but this does not work, since the `\file@name` and `\file@path` do not survive the `\edef`.

```
399 \def\@NEWcurrentprefix{}
```

```
400 \def\NEWrequiremodules#1{\def\@pref{\file@path{#1}}%
401 \ifx\@pref\@empty\else\xdef\@NEWcurrentprefix{\@NEWcurrentprefix/\@pref}\fi%
402 \edef\@input@me{\@NEWcurrentprefix/\file@name{#1}}
403 \message{requiring \@input@me}\reqmodule{\@input@me}}
```

Index

Numbers written in italic refer to the page where the corresponding entry is described; numbers underlined refer to the code line of the definition; numbers in roman refer to the code lines where the entry is used.

References

[Aus+10]   Ron Ausbrooks et al. *Mathematical Markup Language (MathML) Version 3.0*. Tech. rep. World Wide Web Consortium (W3C), 2010. URL: http://www.w3.org/TR/MathML3.

[BFM05]   Tim Berners-Lee, Roy T. Fielding, and Larry Masinter. *Uniform Resource Identifier (URI): Generic Syntax*. RFC 3986. Internet Engineering Task Force (IETF), 2005. URL: http://www.ietf.org/rfc/rfc3986.txt.

[Bus+04]   Stephen Buswell et al. *The Open Math Standard, Version 2.0*. Tech. rep. The OpenMath Society, 2004. URL: http://www.openmath.org/standard/om20.

[Hor+11]   Fulya Horozal et al. "Combining Source, Content, Presentation, Narration, and Relational Representation". In: *Intelligent Computer Mathematics*. Ed. by James Davenport et al. LNAI 6824. Springer Verlag, 2011, pp. 212–227. ISBN: 978-3-642-22672-4. URL: http://kwarc.info/frabe/Research/HIJKR_dimensions_11.pdf.

[Koh06]   Michael Kohlhase. OMDOC – *An open markup format for mathematical documents [Version 1.2]*. LNAI 4180. Springer Verlag, Aug. 2006. URL: http://omdoc.org/pubs/omdoc1.2.pdf.

[Koh08]   Michael Kohlhase. "Using LaTeX as a Semantic Markup Format". In: *Mathematics in Computer Science* 2.2 (2008), pp. 279–304. URL: https://svn.kwarc.info/repos/stex/doc/mcs08/stex.pdf.

[Koh13]   Michael Kohlhase. "Organizing Symbols between Slides, Lecture Notes, Encyclopedias, and Original Papers in Semantic Digital Libraries". KWARC Blue Note. 2013. URL: https://svn.omdoc.org/repos/omdoc/doc/blue/slides-notes/note.pdf.

[Koh14a]   Michael Kohlhase. *metakeys.sty: A generic framework for extensible Metadata in LaTeX*. Tech. rep. Comprehensive TeX Archive Network (CTAN), 2014. URL: http://www.ctan.org/tex-archive/macros/latex/contrib/stex/metakeys/metakeys.pdf.

[Koh14b]   Michael Kohlhase. *omdoc.sty/cls: Semantic Markup for Open Mathematical Documents in LaTeX*. Tech. rep. Comprehensive TeX Archive Network (CTAN), 2014. URL: http://www.ctan.org/tex-archive/macros/latex/contrib/stex/omdoc/omdoc.pdf.

[Koh14c]   Michael Kohlhase. *statements.sty: Structural Markup for Mathematical Statements*. Tech. rep. Comprehensive TeX Archive Network (CTAN), 2014. URL: http://www.ctan.org/tex-archive/macros/latex/contrib/stex/statements/statements.pdf.

[LTX]   Bruce Miller. *LaTeXML: A LaTeX to XML Converter*. URL: http://dlmf.nist.gov/LaTeXML/ (visited on 03/12/2013).

[RO]   Sebastian Rahtz and Heiko Oberdiek. *Hypertext marks in LATEX: a manual for hyperref*. URL: `http : / / tug . org / applications / hyperref/ftp/doc/manual.pdf` (visited on 01/28/2010).

[sTeX]   *Semantic Markup for LATEX*. Project Homepage. URL: `http://trac. kwarc.info/sTeX/` (visited on 02/22/2011).