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

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

January 28, 2012

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 v1.1 (last revised 2012/01/28)

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 [Mil] 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 2.4).

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

`showviews`
`qualifiedimports`

The `modules` package takes two options: If we set `showviews`, then the views (see Section 2.7) 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 3.2.

`showmeta` If the `showmeta` is set, then the metadata keys are shown (see [Koh10a] 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[`⟨*keys*⟩`]{`⟨*cseq*⟩`}[`⟨*args*⟩`]{`⟨*definiens*⟩`}`

where ⟨*cseq*⟩ is a control sequence (the name of the semantic macro) ⟨*args*⟩ is a number between 0 and 9 for the number of arguments ⟨*definiens*⟩ is the token sequence used in macro expansion for ⟨*cseq*⟩. Finally ⟨*keys*⟩ 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 ⟨*cseq*⟩. The key-value pair

`name` `name=`⟨*symname*⟩ 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 ⟨*keys*⟩ 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.

`\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. $\sum_{i=1}^{n} 2i - 1$?

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: $p \oplus q = q \oplus p$
Some authors also write exclusive or with the $\underline{\vee}$ operator, then the formula above is $p\underline{\vee}q = q\underline{\vee}p$

Example 3: Presentation Variants of a Semantic Macro

`\resymdef` 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 Symbol and Concept Names

`\termdef` `\capitalize` EdNote:1 `\termref` `\symref`

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 analogue to `\symdef` that supports this: use `\termdef[`⟨*keys*⟩`]{`⟨*cseq*⟩`}{`⟨*concept*⟩`}` to declare the macro `\`⟨*cseq*⟩ that expands to ⟨*concept*⟩. See Figure 4 for an example, where we use the `\captitalize` macro to adapt ⟨*concept*⟩ to the sentence beginning.[1]. The main use of the `\termdef`-defined concepts lies in automatic cross-referencing facilities via the `\termref` and `\symref` macros provided by the `statements` package [Koh10b]. Together with the `hyperref` package [RO], this provide cross-referencing to the definitions of the symbols and concepts. As discussed in section 3.4, 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}
\captitalize\xdisjunction is commutative: $\xor{p}q=\xor{q}p$
```

Example 4: Extending Example 3 with Term References

2.4 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 relation. Figure 6 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

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

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

— already came across the module ⟨*mod*⟩. In many situations, this is not obtainable; e.g. for "semantic forward references", where symbols or concepts are previewed or motivated to knowledgeable readers before they are formally introduced or for modularizations of documents into multiple files. To enable situations like these, the `module` package uses auxiliary files called **sTeX module signatures**. 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 instead its corresponding sTeX document, if we are only interested in the semantic macros. So `\importmodule[`⟨*filepath*⟩`]{`⟨*mod*⟩`}` will load the sTeX module signature ⟨*filepath*⟩`.sms` (if it exists and has not been loaded before) and activate the semantic macros from module ⟨*mod*⟩ (which was supposedly defined in ⟨*filepath*⟩`.tex`). Note that since ⟨*filepath*⟩`.sms` contains all `\importmodule` statements that ⟨*filepath*⟩`.tex` does, an `\importmodule` recursively loads all necessary files to supply the semantic macros inherited by the current module.

`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[\meta{source-cd}]{`⟨*tname*⟩`}{`⟨*source-tname*⟩`}`, which defines the term with name ⟨*tname*⟩ in the current via 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.

`\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 [RK11] for discussion.

2.5 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 reused, 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 nota-

```
\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 5: A Module for Rings with inheritance from monoids and commutative groups tion) 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 **??**) 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[../other/bar]{bar}
\importmodule[../mycolleaguesmodules]{baz}
\importmodule[../other/bar]{foobar}
  ...
\end{module}
```

Example 6: Self-contained Modules via `importmodule`

In Example 6, 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 OMDoc 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[`⟨*spath*⟩`]{`⟨*OURI*⟩`}{`⟨*name*⟩`}`, where ⟨*spath*⟩ is the file system path to $\mathcal{T}$ (as in `\importmodule`, this argument must not contain the file extension), ⟨*OURI*⟩ is the URI to the OMDoc module (this time with extension), and ⟨*name*⟩ 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 ⟨*spath*⟩ 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 the TeX formatter (it is usually set to something like 15[3]). Therefore, it may be

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

[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

\requiremodules

necessary to circumvent the recursive load pattern providing (logically spurious) `\importmodule` commands. Consider for instance module `bar` in Example 6, 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 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

\sinputref

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

\defpath

Finally, the separation of documents into multiple modules often profits from a symbolic management of file paths. To simplify this, the `modules` package supplies the `\defpath` macro: `\defpath{`⟨*cname*⟩`}{`⟨*path*⟩`}` defines a command, so that `\`⟨*csname*⟩`{`⟨*name*⟩`}` expands to ⟨*path*⟩`/`⟨*name*⟩. So we could have used

```
% \defpath{OPaths}{../other}
% \importmodule[\OPhats{bar}]{bar}
%
```

instead of the second line in Example 6. 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.

2.6 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

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

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.

2.7 Views

A view is a mapping between modules, such that all model assumptions (axioms) of the source module are satisfied in the target module. [2]

EdNote:2

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 [Ste].

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 [Ste, issue #1522] for a discussion.

3.2 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 [Ste, issue #1505]. Until this is implemented the infrastructure is turned off by default, but we have already introduced the `qualifiedimports` option for the future.

`qualifiedimports`

3.3 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.

[2] EdNote: Document and make Examples

3.4 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 [Koh10b]. A way around this limitation would be to import the current module from the sTeX module signature (see Section 2.4) via the `\importmodule` declaration.

3.5 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[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).

```
⟨*package⟩
\DeclareOption{showmeta}{\PassOptionsToPackage{\CurrentOption}{metakeys}}
\newif\ifmod@show\mod@showfalse
\DeclareOption{showmods}{\mod@showtrue}
\newif\ifmod@qualified\mod@qualifiedfalse
\DeclareOption{qualifiedimports}{\mod@qualifiedtrue}
```

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

```
\ProcessOptions
⟨/package⟩
```

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.

```
⟨*package⟩
\RequirePackage{sref}
\RequirePackage{xspace}
\RequirePackage{xcomment}
⟨/package⟩
⟨*ltxml⟩
# -*- CPERL -*-
package LaTeXML::Package::Pool;
use strict;
use LaTeXML::Global;
use LaTeXML::Package;
⟨/ltxml⟩
```

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.

```
⟨*package⟩
```

```
\addmetakey{module}{cd}
\addmetakey{module}{title}
⟨/package⟩
```

`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.

```
⟨*package⟩
\define@key{module}{id}{%
\edef\this@module{\expandafter\noexpand\csname module@defs@#1\endcsname}%
\global\@namedef{module@defs@#1}{}%
\ifmod@qualified
\edef\this@qualified@module{\expandafter\noexpand\csname module@defs@qualified@#1\endcsname}%
\global\@namedef{module@defs@qualified@#1}{}%
\fi
\def\mod@id{#1}%
\expandafter\edef\csname #1@cd@file@base\endcsname{\mod@path}%
\expandafter\g@addto@macro\csname module@defs@#1\expandafter\endcsname\expandafter%
{\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.

```
\newcounter{module}[section]
\newcommand\module@heading{\stepcounter{module}%
\noindent{\textbf{Module} \thesection.\themodule [\mod@id]}%
\sref@label@id{Module \thesection.\themodule [\mod@id]}%
\ifx\module@title\@empty :\quad\else\quad(\module@title)\hfill\\\fi}
```

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

```
\newcommand\module@footer{\noindent{\textbf{EndModule} \thesection.\themodule}}
```

`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.

```
\newenvironment{module}[1][]%
{\metasetkeys{module}{#1}\ifmod@show\module@heading\fi}
{\ifmod@show\module@footer\fi}
⟨/package⟩
```

for the LaTeXML bindings, we have to do the work all at once.

```
⟨*ltxml⟩
DefKeyVal('Module','id','Semiverbatim');
DefKeyVal('Module','cd','Semiverbatim');
DefEnvironment('{module} OptionalKeyVals:Module',
       "?#excluded()(<omdoc:theory "
             . "?&defined(&KeyVal(#1,'id'))(xml:id='&KeyVal(#1,'id')')(xml:id='#id')>#body</omd
#     beforeDigest=>\&useTheoryItemizations,
       afterDigestBegin=>sub {
       my($stomach, $whatsit)=@_;
       $whatsit->setProperty(excluded=>LookupValue('excluding_modules'));

       my $keys = $whatsit->getArg(1);
       my($id, $cd)=$keys
  && map(ToString($keys->getValue($_)),qw(id cd));
        #make sure we have an id or give a stub one otherwise:
if (not $id) {
#do magic to get a unique id for this theory
#$whatsit->setProperties(beginItemize('theory'));
#$id = ToString($whatsit->getProperty('id'));
                # changed: beginItemize returns the hash returned by RefStepCounter.
                # RefStepCounter deactivates any scopes for the current value of the
                # counter which causes the stored prop. of the env. not to be
                # visible anymore.
                $id = LookupValue('stex:theory:id') || 0;
                AssignValue('stex:theory:id', $id+1);
                $id = "I$id";
}
       $cd = $id unless $cd;
       # update the catalog with paths for modules
       my $module_paths = LookupValue('module_paths') || {};
       $module_paths->{$id} = LookupValue('last_module_path');
       AssignValue('module_paths', $module_paths, 'global');

       #Update the current module position
       AssignValue(current_module => $id);
       AssignValue(module_cd => $cd) if $cd;

       #activate the module in our current scope
       $STATE->activateScope("module:".$id);

       #Activate parent scope, if present
       my $parentmod = LookupValue('parent_module');
       use_module($parentmod) if $parentmod;
       #Update the current parent module
       AssignValue("parent_of_$id"=>$parentmod,'global');
       AssignValue("parent_module" => $id);
       return; },
     afterDigest => sub {
       #Move a step up on the module ancestry
       AssignValue("parent_module" => LookupValue("parent_of_".LookupValue("parent_module")));
       return;
     });
```

```
⟨/ltxml⟩
```

`usemodule` The `use_module` subroutine performs depth-first load of definitions of the used modules

```
⟨*ltxml⟩
sub use_module {
  my($module,%ancestors)=@_;
  $module = ToString($module);
  if (defined $ancestors{$module})  {
    Fatal(":module \"$module\" leads to import cycle!");
  }
  $ancestors{$module}=1;
  # Depth-first load definitions from used modules, disregarding cycles
  foreach my $used_module (@{ LookupValue("module_${module}_uses") || []}){
    use_module($used_module,%ancestors);
  }
  # then load definitions for this module
  $STATE->activateScope("module:$module"); }#$
⟨/ltxml⟩
```

`\activate@defs` To activate the `\symdefs` from a given module ⟨*mod*⟩, we call the macro `\module@defs@`⟨*mod*⟩.

```
⟨*package⟩
\def\activate@defs#1{\csname module@defs@#1\endcsname}
⟨/package⟩
```

`\export@defs` To export a the `\symdefs` from the current module, we all the macros `\module@defs@`⟨*mod*⟩ to `\module@defs@`⟨*mod*⟩ (if the current module has a name and it is ⟨*mod*⟩)

```
⟨*package⟩
\def\export@defs#1{\@ifundefined{mod@id}{}%
{\expandafter\expandafter\expandafter\g@addto@macro\expandafter%
\this@module\expandafter{\csname module@defs@#1\endcsname}}}
⟨/package⟩
```

EdNote:3 `\coolurion/off` [3]

```
⟨*package⟩
\def\coolurion{}
\def\coolurioff{}
⟨/package⟩
⟨*ltxml⟩
DefMacro('\coolurion',sub {AssignValue('cooluri'=>1);});
DefMacro('\coolurioff',sub {AssignValue('cooluri'=>0);});
⟨/ltxml⟩
```

`\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`.

[3] EdNote: @DG: this needs to be documented somewhere in section 1

```
⟨*package⟩
\newcommand{\importmodule}[2][]{{\def\mod@path{#1}%
\ifx\mod@path\@empty\else\requiremodules{#1}\fi}%
\activate@defs{#2}\export@defs{#2}}
⟨/package⟩
⟨*ltxml⟩
sub omext {
  my ($mod)=@_; my $dest=’’;
  $mod = ToString($mod);
  if ($mod) {
    #We need a constellation of abs_path invocations
    # to make sure that all symbolic links get resolved
    if ($mod=~/^(\w)+:\/\//) { $dest=$mod; } else {
      my ($d,$f,$t) = pathname_split(abs_path($mod));
      $d = pathname_relative(abs_path($d),abs_path(cwd()));
      $dest=$d."/".$f;
    }
  }
  $dest.=".omdoc" if (ToString($mod) && !LookupValue(’cooluri’));
  return Tokenize($dest);}
sub importmoduleI {
   my($stomach,$whatsit)=@_;
   my $file = ToString($whatsit->getArg(1));
   my $omdocmod = $file.".omdoc" if $file;
   my $module = ToString($whatsit->getArg(2));
   my $containing_module = LookupValue(’current_module’);
   AssignValue(’last_import_module’,$module);
   #set the relation between the current module and the one to be imported
   PushValue("module_".$containing_module."_uses"=>$module) if $containing_module;
  #check if we’ve already loaded this module file or no file path given
  if((!$file) || (LookupValue(’file_’.$module.’_loaded’))) {use_module($module);} #if so activa
  else {
    #if not:
    my $gullet = $stomach->getGullet;
    #1) mark as loaded
    AssignValue(’file_’.$module.’_loaded’ => 1, ’global’);
    #open a group for its definitions so that they are localized
    $stomach->bgroup;
    #update the last module path
    AssignValue(’last_module_path’, $file);
    #queue the closing tag for this module in the gullet where it will be executed
    #after all other definitions of the imported module have been taken care of
    $gullet->unread(Invocation(T_CS(’\end@requiredmodule’), Tokens(Explode($module)))->unlist);
    #we only need to load the sms definitions without generating any xml output, so we set the
    AssignValue(’excluding_modules’ => 1);
    #queue this module’s sms file in the gullet so that its definitions are imported
    $gullet->input($file,[’sms’]);
   }
   return;}
DefConstructor(’\importmodule OptionalSemiverbatim {}’,
   "<omdoc:imports from='?#1(&omext(#1))\##2'/>",
   afterDigest=>sub{ importmoduleI(@_)});
⟨/ltxml⟩
```

\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 groupling introduced by importmodulevia. For LaTeXML, we have to[4]

EdNote:4

```
⟨*package⟩
\newenvironment{importmodulevia}[2][]{\gdef\@@doit{\importmodule[#1]{#2}}%
\ifmod@show\par\noindent importing module #2 via \@@doit\fi}
{\aftergroup\@@doit\ifmod@show end import\fi}
⟨/package⟩
⟨*ltxml⟩
DefMacro('\importmodulevia OptionalSemiverbatim {}','\endgroup\importmoduleI[#1]{#2}\begin{impo
DefMacroI('\end{importmodulevia}',undef,'\end{importmoduleenv}');
DefEnvironment('{importmoduleenv} OptionalSemiverbatim {}',
   "<omdoc:imports from='?#1(&omext(#1))\##2'>"
  .  "<omdoc:morphism>#body</omdoc:morphism>"
  ."</omdoc:imports>");
DefConstructor('\importmoduleI OptionalSemiverbatim {}', '',
   afterDigest=>sub{ importmoduleI(@_)});
⟨/ltxml⟩
```

vassign

```
⟨*package⟩
\newcommand\vassign[2]{\ifmod@show\ensuremath{#1\mapsto #2}, \fi}
⟨/package⟩
⟨*ltxml⟩
DefConstructor('\vassign{}{}',
    "<omdoc:requation>"
   .  "<ltx:Math><ltx:XMath>#1</ltx:XMath></ltx:Math>"
   .  "<ltx:Math><ltx:XMath>#2</ltx:XMath></ltx:Math>"
   ."</omdoc:requation>");
⟨/ltxml⟩
```

tassign

```
⟨*package⟩
\newcommand\tassign[3][]{\ifmod@show #2\ensuremath{\mapsto} #3, \fi}
⟨/package⟩
⟨*ltxml⟩
DefConstructor('\tassign[]{}{}',
    "<omdoc:requation>"
   .  "<om:OMOBJ><om:OMS cd='?#1(#1)(#lastImportModule)' name='#2'/></om:OMOBJ>"
   .  "<om:OMOBJ><om:OMS cd='#currentModule' name='#3'/></om:OMOBJ>"
   ."</omdoc:requation>",
   afterDigest=> sub {
```

[4] EdNote: MK@DG: needs implementation

```
     my ($stomach,$whatsit) = @_;
     $whatsit->setProperty('currentModule',LookupValue("current_module"));
     $whatsit->setProperty('lastImportModule',LookupValue("last_import_module"));
   });
⟨/ltxml⟩
```

ttassign

```
⟨*package⟩
\newcommand\ttassign[3][]{\ifmod@show #1\ensuremath{\mapsto} ``#2'', \fi}
⟨/package⟩
⟨*ltxml⟩
DefConstructor('\ttassign{}{}',
    "<omdoc:requation>"
   .  "<ltx:Math><ltx:XMath>#1</ltx:XMath></ltx:Math>"
   .  "<ltx:Math><ltx:XMath>#2</ltx:XMath></ltx:Math>"
   ."</omdoc:requation>");
⟨/ltxml⟩
```

\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.

```
⟨*package⟩
\newcommand{\importOMDocmodule}[3][]{\importmodule[#1]{#3}}
⟨/package⟩
⟨*ltxml⟩
DefConstructor('\importOMDocmodule OptionalSemiverbatim {}{}',"<omdoc:imports from='#3\##2'/>",
afterDigest=>sub{
  #Same as \importmodule, just switch second and third argument.
  my ($stomach,$whatsit) = @_;
  my $path = $whatsit->getArg(1);
  my $ouri = $whatsit->getArg(2);
  my $module = $whatsit->getArg(3);
  $whatsit->setArgs(($path, $module,$ouri));
  importmoduleI($stomach,$whatsit);
  return;
});
⟨/ltxml⟩
```

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

```
⟨*package⟩
\let\metalanguage=\importmodule
⟨/package⟩
⟨*ltxml⟩
DefConstructor('\metalanguage OptionalSemiverbatim {}',
   "<omdoc:imports type='metalanguage' from='?#1(&omext(#1))\##2'/>",
   afterDigest=>sub{ importmoduleI(@_)});
⟨/ltxml⟩
```

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. This is just a copy of the code from `latex.ltx` where I have removed the `\@ifdefinable` check.[5]

```
⟨*package⟩
\def\mod@newcommand{\@star@or@long\mod@new@command}
\def\mod@new@command#1{\@testopt{\@mod@newcommand#1}0}
\def\@mod@newcommand#1[#2]{\kernel@ifnextchar [{\mod@xargdef#1[#2]}{\mod@argdef#1[#2]}}
\long\def\mod@argdef#1[#2]#3{\@yargdef#1\@ne{#2}{#3}}
\long\def\mod@xargdef#1[#2][#3]#4{\expandafter\def\expandafter#1\expandafter{%
\expandafter\@protected@testopt\expandafter #1\csname\string#1\endcsname{#3}}%
\expandafter\@yargdef\csname\string#1\endcsname\tw@{#2}{#4}}
⟨/package⟩
```

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

`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.

```
⟨*package⟩
\newif\if@symdeflocal
\define@key{symdef}{local}[true]{\@symdeflocaltrue}
\define@key{symdef}{name}{}
\define@key{symdef}{assocarg}{}
\define@key{symdef}{bvars}{}
\define@key{symdef}{bvar}{}
\define@key{symdef}{bindargs}{}
⟨/package⟩
```

EdNote:5 [5]

`\symdef` The the `\symdef`, and `\@symdef` macros just handle optional arguments.

```
⟨*package⟩
\def\symdef{\@ifnextchar[{\@symdef}{\@symdef[]}}
\def\@symdef[#1]#2{\@ifnextchar[{\@@symdef[#1]{#2}}{\@@symdef[#1]{#2}[0]}}
```

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

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

[5] Someone must have done this before, I would be very happy to hear about a package that provides this.

[5] EdNote: MK@MK: we need to document the binder keys above.

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.

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

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.

```
\@symdeflocalfalse\setkeys{symdef}{#1}%
```

First, using \mod@newcommand we initialize the intermediate macro \module@⟨*sym*⟩@pres@, the one that can be extended with \symvariant

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

and then we define the actual semantic macro. Note that this can take an optional argument, for which we provide with \@ifnextchar and an internal macro \@⟨*sym*⟩, which when invoked with an optional argument ⟨*opt*⟩ calls \modules@⟨*sym*⟩@pres@⟨*opt*⟩.

```
\expandafter\def\csname #2\endcsname%
{\@ifnextchar[{\csname modules@#2\endcsname}{\csname modules@#2\endcsname[]}}%
\expandafter\def\csname modules@#2\endcsname[##1]%
{\csname modules@#2@pres@##1\endcsname}%
```

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

```
\expandafter\@mod@nc\csname mod@symref@#2\expandafter\endcsname\expandafter%
{\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.

```
\if@symdeflocal\else%
\@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*⟩@pres@.

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

Then we add add the definition of \⟨*sym*⟩ in terms of the function \@⟨*sym*⟩ to handle the optional argument.

```
\expandafter\g@addto@macro\this@module%
{\expandafter\def\csname#2\endcsname%
{\@ifnextchar[{\csname modules@#2\endcsname}{\csname modules@#2\endcsname[]}}}%
```

Finally, we add add the definition of \@⟨*sym*⟩, which calls the intermediate function.

```
\expandafter\g@addto@macro\this@module%
{\expandafter\def\csname modules@#2\endcsname[##1]%
{\csname modules@#2@pres@##1\endcsname}}%
```

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

```
\expandafter\g@addto@macro\csname  module@defs@\mod@id\expandafter\endcsname\expandafter%
{\expandafter\@mod@nc\csname mod@symref@#2\expandafter\endcsname\expandafter%
{\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.

```
\ifmod@qualified%
\expandafter\g@addto@macro\this@qualified@module%
{\expandafter\mod@newcommand\csname modules@#2@pres@qualified\endcsname[#3]{#4}}%
\expandafter\g@addto@macro\this@qualified@module%
{\expandafter\def\csname#2atqualified\endcsname{\csname modules@#2@pres@qualified\endcsname}}%
\fi%
```

So now we only need to close all brackets and the macro is done.

```
}\fi}
⟨/package⟩
```

In the LaTeXML bindings, we have a top-level macro that delegates the work to two internal macros: \@symdef, which defines the content macro and \@symdef@pres, which generates the OMDoc symbol and presentation elements (see Section 4.6.2).

```
⟨*package⟩
\define@key{DefMathOp}{name}{\def\defmathop@name{#1}}
\newcommand\DefMathOp[2][]{%
\setkeys{DefMathOp}{#1}%
\symdef[#1]{\defmathop@name}{#2}}
⟨/package⟩
⟨*ltxml⟩
DefMacro('\DefMathOp OptionalKeyVals:symdef {}',
 sub {
   my($self,$keyval,$pres)=@_;
   my $name = KeyVal($keyval,'name') if $keyval;
   #Rewrite this token
   my $scopes = $STATE->getActiveScopes;
   DefMathRewrite(xpath=>'descendant-or-self::ltx:XMath',match=>ToString($pres),
        replace=>sub{
          map {$STATE->activateScope($_);} @$scopes;
          $_[0]->absorb(Digest("\\".ToString($name)));
        });
   #Invoke symdef
   (Invocation(T_CS('\symdef'),$keyval,$name,undef,$pres)->unlist);
 });
DefMacro('\symdef OptionalKeyVals:symdef {}[]{}',
      sub {
  my($self,@args)=@_;
  ((Invocation(T_CS('\@symdef'),@args)->unlist),
   (LookupValue('excluding_modules') ? ()
     : (Invocation(T_CS('\@symdef@pres'), @args)->unlist))); });
```

```

#Current list of recognized formatter command sequences:
our @PresFormatters = qw (infix prefix postfix assoc mixfixi mixfixa mixfixii mixfixia mixfixai
DefPrimitive('\@symdef OptionalKeyVals:symdef {}[]{}', sub {
  my($stomach,$keys,$cs,$nargs,$presentation)=@_;
  my($name,$cd,$role,$bvars,$bvar)=$keys
    && map($_ && $_->toString,map($keys->getValue($_), qw(name cd role
    bvars bvar)));
  $cd = LookupValue('module_cd') unless $cd;
  $name = $cs unless $name;
  #Store for later lookup
  AssignValue("symdef.".ToString($cs).".cd"=>ToString($cd),'global');
  AssignValue("symdef.".ToString($cs).".name"=>ToString($name),'global');
  $nargs = (ref $nargs ? $nargs->toString : $nargs || 0);
  my $module = LookupValue('current_module');
  my $scope = (($keys && ($keys->getValue('local') || '' eq 'true')) ? 'module_local' : 'module
  #The DefConstructorI Factory is responsible for creating the \symbol command sequences as dic
  DefConstructorI("\\".$cs->toString,convertLaTeXArgs($nargs+1,'default'), sub {
     my ($document,@args) = @_;
     my $icvariant = shift @args;
     my @props = @args;
     #Lookup the presentation from the State, if a variant:
     @args = splice(@props,0,$nargs);
     my %prs = @props;
     my $localpres = $prs{presentation};
     $prs{isbound} = "BINDER" if ($bvars || $bvar);
     my $wrapped;
     my $parent=$document->getNode;
     if(! defined $parent->lookupNamespacePrefix("http://omdoc.org/ns")){ # namespace not alrea
       $document->getDocument->documentElement->setNamespace("http://omdoc.org/ns","omdoc",0);
     my $symdef_scope=$parent->exists('ancestor::omdoc:rendering'); #Are we in a \symdef render
     if (($localpres =~/^LaTeXML::Token/) && $symdef_scope) {
       #Note: We should probably ask Bruce whether this maneuver makes sense
       # We jump back to digestion, at a processing stage where it has been already completed
       # Hence need to reinitialize all scopes and make a new group. This is probably expensive

       my @toks = $localpres->unlist;
       while(@toks && $toks[0]->equals(T_SPACE)){ shift(@toks); }  # Remove leading space
       my $formatters = join("|",@PresFormatters);
       $formatters = qr/$formatters/;
       $wrapped = (@toks && ($toks[0]->toString =~ /^\\($formatters)$/));
       $localpres = Invocation(T_CS('\@use'),$localpres) unless $wrapped;
       # Plug in the provided arguments, doing a nasty reversion:
       my @sargs = map (Tokens($_->revert),  @args);
       $localpres = Tokens(LaTeXML::Expandable::substituteTokens($localpres,@sargs)) if $nargs>
       #Digest:
       my $stomach = $STATE->getStomach;
       $stomach->beginMode('inline-math');
       $STATE->activateScope($scope);
       use_module($module);
```

```
      use_module(LookupValue("parent_of_".$module)) if LookupValue("parent_of_".$module);
      $localpres=$stomach->digest($localpres);
      $stomach->endMode('inline-math');
    }
    else { #Some are already digested to Whatsit, usually when dropped from a wrapping constru
    }
    if ($nargs == 0) {
      if (!$symdef_scope) { #Simple case - discourse flow, only a single XMTok
        #Referencing XMTok when not in \symdefs:
        $document->insertElement('ltx:XMTok',undef,(name=>$cs->toString, meaning=>$name,omcd=>
      }
      else {
        if ($symdef_scope && ($localpres =~/^LaTeXML::Whatsit/) && (!$wrapped)) {#1. Simple ca
          $localpres->setProperties((name=>$cs->toString, meaning=>$name,omcd=>$cd,role => $ro
        }
        else {
          #Experimental treatment - COMPLEXTOKEN
          #$role=$role||'COMPLEXTOKEN';
          #$document->openElement('ltx:XMApp',role=>'COMPLEXTOKEN');
          #$document->insertElement('ltx:XMTok',undef,(name=>$cs->toString, meaning=>$name, om
          #$document->openElement('ltx:XMWrap');
          #$document->absorb($localpres);
          #$document->closeElement('ltx:XMWrap');
          #$document->closeElement('ltx:XMApp');
        }
        #We need expanded presentation when invoked in \symdef scope:

        #Suppress errors from rendering attributes when absorbing.
        #This is bad style, but we have no way around it due to the digestion acrobatics.
        my $verbosity = $LaTeXML::Global::STATE->lookupValue('VERBOSITY');
        my $errors = $LaTeXML::Global::STATE->getStatus('error');
        $LaTeXML::Global::STATE->assignValue('VERBOSITY',-5);

        #Absorb presentation:
        $document->absorb($localpres);

        #Return to original verbosity and error state:
        $LaTeXML::Global::STATE->assignValue('VERBOSITY',$verbosity);
        $LaTeXML::Global::STATE->setStatus('error',$errors);

        #Strip all/any <rendering><Math><XMath> wrappers:
        #TODO: Ugly LibXML work, possibly do something smarter
        my $parent = $document->getNode;
        my @renderings=$parent->findnodes(".//omdoc:rendering");
        foreach my $render(@renderings) {
          my $content=$render;
          while ($content && $content->localname =~/^rendering|[X]?Math/) {
            $content = $content->firstChild;
          }
          my $sibling = $content->parentNode->lastChild;
          my $localp = $render->parentNode;
          while ((defined $sibling) && (!$sibling->isSameNode($content))) {
            my $clone = $sibling->cloneNode(1);
            $localp->insertAfter($clone,$render);
            $sibling = $sibling->previousSibling;
          }
          $render->replaceNode($content);
        }
      }
    }
    else {#2. Constructors with arguments
      if (!$symdef_scope) { #2.1 Simple case, outside of \symdef declarations:
        #Referencing XMTok when not in \symdefs:
        my %ic = ($icvariant ne 'default') ? (ic=>'variant:'.$icvariant) : ();
        $document->openElement('ltx:XMApp',%ic,scriptpos=>$prs{'scriptpos'},role=>$prs{'isboun
        $document->insertElement('ltx:XMTok',undef,(name=>$cs->toString, meaning=>$name, omcd=
        foreach my $carg (@args) {
          if ($carg =~/^LaTeXML::Token/) {
            my $stomach = $STATE->getStomach;
            $stomach->beginMode('inline-math');
            $carg=$stomach->digest($carg);
            $stomach->endMode('inline-math');
          }
          $document->openElement('ltx:XMArg');
          $document->absorb($carg);
          $document->closeElement('ltx:XMArg');
        }
        $document->closeElement('ltx:XMApp');
      }
      else { #2.2 Complex case, inside a \symdef declaration
        #We need expanded presentation when invoked in \symdef scope:

        #Suppress errors from rendering attributes when absorbing.
        #This is bad style, but we have no way around it due to the digestion acrobatics.
        my $verbosity = $LaTeXML::Global::STATE->lookupValue('VERBOSITY');
        my $errors = $LaTeXML::Global::STATE->getStatus('error');
        $LaTeXML::Global::STATE->assignValue('VERBOSITY',-5);

        #Absorb presentation:
        $document->absorb($localpres);

        #Return to original verbosity and error state:
        $LaTeXML::Global::STATE->assignValue('VERBOSITY',$verbosity);
        $LaTeXML::Global::STATE->setStatus('error',$errors);

        #Strip all/any <rendering><Math><XMath> wrappers:
        #TODO: Ugly LibXML work, possibly do something smarter?
        my $parent = $document->getNode;
        if(! defined $parent->lookupNamespacePrefix("http://omdoc.org/ns")){ # namespace not a
          $document->getDocument->documentElement->setNamespace("http://omdoc.org/ns","omdoc",
        my @renderings=$parent->findnodes(".//omdoc:rendering");
        foreach my $render(@renderings) {
          my $content=$render;
          while ($content && $content->localname =~/^rendering|[X]?Math/) {
            $content = $content->firstChild;
          }
          my $sibling = $content->parentNode->lastChild;
          my $localp = $render->parentNode;
          while ((defined $sibling) && (!$sibling->isSameNode($content))) {
            my $clone = $sibling->cloneNode(1);
            $localp->insertAfter($clone,$render);
            $sibling = $sibling->previousSibling;
          }
          $render->replaceNode($content);
        }
      }
    }},
  properties => {name=>$cs->toString, meaning=>$name,omcd=>$cd,role => $role},
  scope=>$scope,
  beforeDigest => sub{
    my ($gullet, $variant) = @_;
    my $icvariant = ToString($variant);
    my $localpres = $presentation;
    if ($icvariant && $icvariant ne 'default') {
      $localpres = LookupValue($cs->toString."$icvariant:pres");
      if (!$localpres) {
        Error("No variant named '$icvariant' found! Falling back to ".
        "default.\n Please consider introducing \\symvariant{".
        $cs->toString."}[$nargs]{$icvariant}{... your presentation ...}");
        $localpres = $presentation;
      }
    }
    my $count = LookupValue(ToString($cs).'_counter') || 0;
    AssignValue(ToString($cs).":pres:$count",$localpres);
    AssignValue(ToString($cs).'_counter',$count+1);
    return;
  },
  afterDigest => sub{
    my ($stomach,$whatsit) = @_;
    my $count = LookupValue(ToString($cs).'_aftercounter') || 0;
    $whatsit->setProperty('presentation',LookupValue(ToString($cs).":pres:$count"));
    AssignValue(ToString($cs).'_aftercounter',$count+1);
  });
  return; });
⟨/ltxml⟩%$
```

\symvariant \symvariant{⟨*sym*⟩}[⟨*args*⟩]{⟨*var*⟩}{⟨*cseq*⟩} just extends the internal macro \modules@⟨*sym*⟩@pres@ defined by \symdef{⟨*sym*⟩}[⟨*args*⟩]{...} with a variant \modules@⟨*sym*⟩@pres@⟨*var*⟩ which expands to ⟨*cseq*⟩. Recall that this is called

EdNote:6 by the macro \⟨*sym*⟩[⟨*var*⟩] induced by the `\symdef`.[6]

```
⟨*package⟩
\def\symvariant#1{\@ifnextchar[{\@symvariant{#1}}{\@symvariant{#1}[0]}}
\def\@symvariant#1[#2]#3#4{%
\expandafter\mod@newcommand\csname modules@#1@pres@#3\endcsname[#2]{#4}%
```

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

```
\@ifundefined{mod@id}{}{%
\expandafter\g@addto@macro\this@module%
{\expandafter\mod@newcommand\csname modules@#1@pres@#3\endcsname[#2]{#4}}}}%
⟨/package⟩
⟨*ltxml⟩
 DefMacro('\symvariant{}[]{}{}', sub {
  my($self,@args)=@_;
  my $prestok = Invocation(T_CS('\@symvariant@pres'), @args);
  pop @args; push @args, $prestok;
  Invocation(T_CS('\@symvariant@construct'),@args)->unlist;
});
 DefMacro('\@symvariant@pres{}[]{}{}', sub {
   my($self,$cs,$nargs,$ic,$presentation)=@_;
   symdef_presentation_pmml($cs,ToString($nargs)||0,$presentation);
 });
 DefConstructor('\@symvariant@construct{}[]{}{}', sub {
  my($document,$cs,$nargs,$icvariant,$presentation)=@_;
  $cs = ToString($cs);
  $nargs = ToString($nargs);
  $icvariant = ToString($icvariant);
  # Save presentation for future reference:
  #Notation created by \symdef
  #Create the rendering at the right place:
  my $cnode = $document->getNode;
  my $root = $document->getDocument->documentElement;
  my $name = LookupValue("symdef.".ToString($cs).".name") || $cs;
  # Fix namespace (the LibXML XPath problems...)
  $root->setNamespace("http://omdoc.org/ns","omdoc",0);
  my ($notation) = $root->findnodes(".//omdoc:notation[\@name='$name' and ".
                                    "preceding-sibling::omdoc:symbol[1]/\@name
                                    = '$name']");
  if (!$notation) {
    #No symdef found, raise error:
    Error("No \\symdef found for \\$cs! Please define symbol prior to introducing variants!");
    return;
  }
  $document->setNode($notation);
  $document->absorb($presentation);
  $notation->lastChild->setAttribute("ic","variant:$icvariant");
  $document->setNode($cnode);
```

[6] EdNote: MK@DG: this needs to be implemented in LaTeXML

```
  return;
 },
 beforeDigest => sub {
    my($gullet,$cs,$nargs,$icvariant,$presentation)=@_;
    $cs = ToString($cs);
    $icvariant = ToString($icvariant);
    AssignValue("$cs:$icvariant:pres",Digest($presentation),'module:'.LookupValue('current_modu
 });
 #mode=>'math'
⟨/ltxml⟩
```

\resymdef This is now deprecated.

```
⟨*package⟩
\def\resymdef{\@ifnextchar[{\@resymdef}{\@resymdef[]}}
\def\@resymdef[#1]#2{\@ifnextchar[{\@@resymdef[#1]{#2}}{\@@resymdef[#1]{#2}[0]}}
\def\@@resymdef[#1]#2[#3]#4{\PackageError{modules}
  {The \protect\resymdef macro is deprecated,\MessageBreak
    use the \protect\symvariant instead!}}
⟨/package⟩
```

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

```
⟨*package⟩
\let\abbrdef\symdef
⟨/package⟩
⟨*ltxml⟩
DefPrimitive('\abbrdef OptionalKeyVals:symdef {}[]{}', sub {
  my($stomach,$keys,$cs,$nargs,$presentation)=@_;
  my $module = LookupValue('current_module');
  my $scope = (($keys && ($keys->getValue('local') || '' eq 'true')) ? 'module_local' : 'module
  DefMacroI("\\".$cs->toString,convertLaTeXArgs($nargs,''),$presentation,
   scope=>$scope);
  return; });
⟨/ltxml⟩
```

4.4 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.

```
⟨*package⟩
\def\mod@path{}
⟨/package⟩
```

\termdef

```
⟨*package⟩
\def\mod@true{true}
\addmetakey[false]{termdef}{local}
\addmetakey{termdef}{name}
```

```
\newcommand{\termdef}[3][]{\metasetkeys{termdef}{#1}%
\expandafter\mod@newcommand\csname#2\endcsname[0]{#3\xspace}%
\ifx\termdef@local\mod@true\else%
\@ifundefined{mod@id}{}{\expandafter\g@addto@macro\this@module%
{\expandafter\mod@newcommand\csname#2\endcsname[0]{#3\xspace}}}%
\fi}
⟨/package⟩
```

\capitalize

```
⟨*package⟩
\def\@captitalize#1{\uppercase{#1}}
\newcommand\capitalize[1]{\expandafter\@captitalize #1}
⟨/package⟩
```

\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.

```
⟨*package⟩
\def\mod@termref#1#2#3{\def\@test{#3}
\@ifundefined{#1@cd@file@base}
    {\protect\G@refundefinedtrue
      \@latex@warning{\protect\termref with unidentified cd "#1": the cd key must
        reference an active module}
      \def\@label{sref@#2 @target}}
  {\def\@label{sref@#2@#1@target}}%
\expandafter\ifx\csname #1@cd@file@base\endcsname\@empty% local reference
\sref@hlink@ifh{\@label}{\ifx\@test\@empty #2\else #3\fi}\else%
\def\@uri{\csname #1@cd@file@base\endcsname.pdf\#\@label}%
\sref@href@ifh{\@uri}{\ifx\@test\@empty #2\else #3\fi}\fi}
⟨/package⟩
```

4.5 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.

4.5.1 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.

```
⟨*package⟩
\def\mod@simplify#1{\expandafter\mod@simpl#1/\relax}
```

EdNote:7 It is based on the `\mod@simpl` macro[7]

[7] EdNote: what does the mod@blaaa do?

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

which in turn is based on a helper macro

```
\def\mod@updir{..}
\def\mod@simplhelp#1/#2/#3\relax{\def\@first{#1}\def\@second{#2}\def\@third{#3}%
%\message{mod@simplhelp: first=\@first, second=\@second, third=\@third, result=\mod@savedprefix
\ifx\@third\@empty% base case
\ifx\@second\mod@updir\else%

\ifx\mod@second\@empty\edef\mod@savedprefix{\mod@savedprefix#1}%
\else\edef\mod@savedprefix{\mod@savedprefix#1/#2}%
\fi%
\fi%
\else%
\ifx\@first\mod@updir%
\edef\mod@savedprefix{\mod@savedprefix#1/}\mod@simplhelp#2/#3\relax%
\else%
\ifx\@second\mod@updir\mod@simpl#3\relax%
\else\edef\mod@savedprefix{\mod@savedprefix#1/}\mod@simplhelp#2/#3\relax%
\fi%
\fi%
\fi}%
⟨/package⟩
```

We directly test the simplification:

source	result	should be
../../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

\defpath

```
⟨*package⟩
\newcommand{\defpath}[2]{\expandafter\newcommand\csname #1\endcsname[1]{#2/##1}}
⟨/package⟩
⟨*ltxml⟩
DefMacro('\defpath{}{}', sub {
  my ($gullet,$arg1,$arg2)=@_;
  $arg1 = ToString($arg1);
  $arg2 = ToString($arg2);
  my $paths = LookupValue('defpath')||{};
  $$paths{"$arg1"}=$arg2;
  AssignValue('defpath'=>$paths,'global');
  DefMacro('\\'.$arg1.' Semiverbatim',$arg2."/#1");
 });#$
⟨/ltxml⟩
```

4.6 Loading Module Signatures

EdNote:8 We will need a switch[8]

```
⟨*package⟩
\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.

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

`\mod@update` This macro provides special append functionality. It takes a string and appends it to the expansion of the `\mod@reg` macro in the following way: `string@\mod@reg`.

```
\def\mod@update#1{\ifx\mod@reg\@empty\xdef\mod@reg{#1}\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`.

```
\def\mod@check#1@#2///#3\relax{%
\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

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

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

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

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

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

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

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

[8] EdNote: explain why?

\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

```
\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.

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

\mod@updatedpre

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

```
\def\mod@updatedpre#1{%
\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

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

It needs two helper macros:

```
\def\mod@pathprefix@help#1/#2\relax{%
\ifx\\#2\\% end of recursion
\else\mod@ReturnAfterFi{#1/\mod@pathprefix@help#2\relax}%
\fi}
\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

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

Helper function for the \pathpostfix@check macro defined above

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

\mod@updatedpost

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

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

\mod@updatedsms

Finally: A macro that will add a .sms extension to a path. Will be used when adding a .sms file

```
\def\mod@updatesms{\edef\mod@savedprefix{\mod@savedprefix.sms}}
⟨/package⟩
```

4.6.1 Selective Inclusion

\requiremodules

```
⟨*package⟩
\newcommand\requiremodules[1]{%
{\mod@showfalse% save state and ensure silence while reading sms
\mod@updatedpre{#1}% add the new file to the already existing path
\let\mod@savedprefix\mod@prefix% add the path to the new file to the prefix
\mod@updatedpost{#1}%
\def\mod@blaaaa{}% macro used in the simplify function (remove .. from the prefix)
\mod@simplify{\mod@savedprefix}% remove |xxx/..| from the path (in case it exists)
\mod@reguse{\mod@savedprefix}%
\ifmodules\else%
\mod@updatesms% update the file to contain the .sms extension
\let\newreg\mod@reg% use to compare, in case the .sms file was loaded before
\mod@search{\mod@savedprefix}% update registry
\ifx\newreg\mod@reg\else\input{\mod@savedprefix}\fi% check if the registry was updated and load
\fi}}
⟨/package⟩
⟨*ltxml⟩
DefPrimitive('\requiremodules{}', sub {
  my($stomach,$module)=@_;
  my $GULLET = $stomach->getGullet;
  $module = Digest($module)->toString;
  if(LookupValue('file_'.$module.'_loaded')) {}
  else {
    AssignValue('file_'.$module.'_loaded' => 1, 'global');
    $stomach->bgroup;
    AssignValue('last_module_path', $module);
    $GULLET->unread(T_CS('\end@requiredmodule'));
    AssignValue('excluding_modules' => 1);
    $GULLET->input($module,['sms']);
       }
  return;});

DefPrimitive('\end@requiredmodule{}',sub {
 #close the group
 $_[0]->egroup;
 #print STDERR "END: ".ToString(Digest($_[1])->toString);
 #Take care of any imported elements in this current module by activating it and all its depend
 #print STDERR "Important: ".ToString(Digest($_[1])->toString)."\n";
 use_module(ToString(Digest($_[1])->toString));
 return; });#$
⟨/ltxml⟩
```

\sinput

```
⟨*package⟩
\def\sinput#1{
{\mod@updatedpre{#1}% add the new file to the already existing path
\let\mod@savedprefix\mod@prefix% add the path to the new file to the prefix
\mod@updatedpost{#1}%
\def\mod@blaaaa{}% macro used in the simplify function (remove .. from the prefix)
\mod@simplify{\mod@savedprefix}% remove |xxx/..| from the path (in case it exists)
\mod@reguse{\mod@savedprefix}%
\let\newreg\mod@reg% use to compare, in case the .sms file was loaded before
\mod@search{\mod@savedprefix}% update registry
\ifx\newreg\mod@reg%\message{This file has been previously introduced}
\else\input{\mod@savedprefix}%
\fi}}
⟨/package⟩
⟨*ltxml⟩
DefPrimitive(’\sinput Semiverbatim’, sub {
  my($stomach,$module)=@_;
  my $GULLET = $stomach->getGullet;
  $module = Digest($module)->toString;
  AssignValue(’file_’.$module.’_loaded’ => 1, ’global’);
  $stomach->bgroup;
  AssignValue(’last_module_path’, $module);
  $GULLET->unread(Invocation(T_CS(’\end@requiredmodule’),Tokens(Explode($module)))->unlist);
  $GULLET->input($module,[’tex’]);
  return;});#$
⟨/ltxml⟩
```

EdNote:9 [9]

```
⟨*package⟩
\let\sinputref=\sinput
\let\inputref=\input
⟨/package⟩
⟨*ltxml⟩
DefConstructor(’\sinputref{}’,"<omdoc:oref href=’#1.omdoc’ class=’expandable’/>");
DefConstructor(’\inputref{}’,"<omdoc:oref href=’#1.omdoc’ class=’expandable’/>");
⟨/ltxml⟩
```

4.6.2 Generating OMDoc Presentation Elements

Additional bundle of code to generate presentation encodings. Redefined to an expandable (macro) so that we can add conversions.

```
⟨*ltxml⟩
DefMacro(’\@symdef@pres  OptionalKeyVals:symdef {}[]{}’, sub {
  my($self,$keys, $cs,$nargs,$presentation)=@_;

  my($name,$cd,$role)=$keys
   && map($_ && $_->toString,map($keys->getValue($_), qw(name cd role)));
  $cd = LookupValue(’module_cd’) unless $cd;
  $name = $cs unless $name;
  AssignValue(’module_name’=>$name) if $name;
```

[9] EdNote: the sinput macro is just faked, it should be more like requiremodules, except that the tex file is inputted; I wonder if this can be simplified.

```
  $nargs = 0 unless ($nargs);
  my $nargkey = ToString($name).’_args’;
  AssignValue($nargkey=>ToString($nargs)) if $nargs;
  $name=ToString($name);

  Invocation(T_CS(’\@symdef@pres@aux’),
     $cs,
     ($nargs || Tokens(T_OTHER(0))),
     symdef_presentation_pmml($cs,ToString($nargs)||0,$presentation),
     (Tokens(Explode($name))),
     (Tokens(Explode($cd))),
     $keys)->unlist; });#$
```

Generate the expansion of a symdef’s macro using special arguments.

Note that the `symdef_presentation_pmml` subroutine is responsible for preserving the rendering structure of the original definition. Hence, we keep a collection of all known formatters in the `@PresFormatters` array, which should be updated whenever the list of allowed formatters has been altered.

```
sub symdef_presentation_pmml {
  my($cs,$nargs,$presentation)=@_;
  my @toks = $presentation->unlist;
  while(@toks && $toks[0]->equals(T_SPACE)){ shift(@toks); }  # Remove leading space
  $presentation = Tokens(@toks);
  # Wrap with \@use, unless already has a recognized formatter.
  my $formatters = join("|",@PresFormatters);
  $formatters = qr/$formatters/;
  $presentation = Invocation(T_CS(’\@use’),$presentation)
    unless (@toks && ($toks[0]->toString =~ /^\\($formatters)$/));
  # Low level substitution.
  my @args =
  map(Invocation(T_CS(’\@SYMBOL’),T_OTHER("arg:".($_))),1..$nargs);
  $presentation =  Tokens(LaTeXML::Expandable::substituteTokens($presentation,@args));
  $presentation; }#$
```

The `\@use` macro just generates the contents of the notation element

```
sub  getSymmdefProperties {
  my $cd = LookupValue(’module_cd’);
  my $name = LookupValue(’module_name’);
  my $nargkey = ToString($name).’_args’;
  my $nargs = LookupValue($nargkey);
  $nargs = 0 unless ($nargs);
  my %props = (’cd’=>$cd,’name’=>$name,’nargs’=>$nargs);
  return %props;}
DefConstructor(’\@use{}’, sub{
  my ($document,$args,%properties) = @_;
  #Notation created at \@symdef@pres@aux
  #Create the rendering:
  $document->openElement(’omdoc:rendering’);
  $document->openElement(’ltx:Math’);
  $document->openElement(’ltx:XMath’);
  if ($args->isMath) {$document->absorb($args);}
  else {  $document->insertElement('ltx:XMText',$args);}
  $document->closeElement('ltx:XMath');
  $document->closeElement('ltx:Math');
  $document->closeElement('omdoc:rendering');
},
properties=>sub { getSymmdefProperties($_[1]);},
               mode=>'inline_math');
```

The get_cd procedure reads of the cd from our list of keys.

```
sub get_cd {
   my($name,$cd,$role)=@_;
   return $cd;}
```

The \@symdef@pres@aux creates the symbol element and the outer layer of the of the notation element. The content of the latter is generated by applying the LaTeXML to the definiens of the \symdef form.

```
DefConstructor('\@symdef@pres@aux{}{}{}{}{} OptionalKeyVals:symdef', sub {
  my ($document,$cs,$nargs,$pmml,$name,$cd,$keys)=@_;
  my $assocarg = ToString($keys->getValue('assocarg')) if $keys;
  $assocarg = $assocarg||"0";
  my $bvars = ToString($keys->getValue('bvars')) if $keys;
  $bvars = $bvars||"0";
  my $bvar = ToString($keys->getValue('bvar')) if $keys;
  $bvar = $bvar||"0";
  my $appElement = 'om:OMA'; $appElement = 'om:OMBIND' if ($bvars || $bvar);
  my $root = $document->getDocument->documentElement;
  my $name_str = ToString($name);
  my ($notation) = $root->findnodes(".//omdoc:notation[\@name='$name_str' and ".
                                    "preceding-sibling::omdoc:symbol[1]/\@name
                                    = '$name_str']");
  if (!$notation) {
    $document->insertElement("omdoc:symbol",undef,(name=>$name,"xml:id"=>$name_str.".sym"));
  }
   $document->openElement("omdoc:notation",(name=>$name,cd=>$cd));
   #First, generate prototype:
   $nargs = ToString($nargs)||0;
   $document->openElement('omdoc:prototype');
   $document->openElement($appElement) if $nargs;
   my $cr="fun" if $nargs;
   $document->insertElement('om:OMS',undef,
    (cd=>$cd,
     name=>$name,
     "cr"=>$cr));
   if ($bvar || $bvars) {
     $document->openElement('om:OMBVAR');
     if ($bvar) {
       $document->insertElement('omdoc:expr',undef,(name=>"arg$bvar"));
     } else {
       $document->openElement('omdoc:exprlist',(name=>"args"));
       $document->insertElement(’omdoc:expr’,undef,(name=>"arg"));
       $document->closeElement(’omdoc:exprlist’);
     }
     $document->closeElement(’om:OMBVAR’);
   }
   for my $id(1..$nargs) {
     next if ($id==$bvars || $id==$bvar);
     if ($id!=$assocarg) {
       my $argname="arg$id";
       $document->insertElement(’omdoc:expr’,undef,(name=>"$argname"));
     }
     else {
       $document->openElement(’omdoc:exprlist’,(name=>"args"));
       $document->insertElement(’omdoc:expr’,undef,(name=>"arg"));
       $document->closeElement(’omdoc:exprlist’);
     }
   }
   $document->closeElement($appElement) if $nargs;
   $document->closeElement(’omdoc:prototype’);
   #Next, absorb rendering:
   $document->absorb($pmml);
   $document->closeElement("omdoc:notation");
 }, afterDigest=>sub { my ($stomach, $whatsit) = @_;
  my $keys = $whatsit->getArg(6);
  my $module = LookupValue(’current_module’);
  $whatsit->setProperties(for=>ToString($whatsit->getArg(1)));
  $whatsit->setProperty(role=>($keys ? $keys->getValue(’role’)
       : (ToString($whatsit->getArg(2)) ? ’applied’
  : undef))); });
```

Convert a macro body (tokens with parameters #1,..) into a Presentation `style=TeX` form. walk through the tokens, breaking into chunks of neutralized (`T_OTHER`) tokens and parameter specs.

```
sub symdef_presentation_TeX {
  my($presentation)=@_;
  my @tokens = $presentation->unlist;
  my(@frag,@frags) = ();
  while(my $tok = shift(@tokens)){
    if($tok->equals(T_PARAM)){
      push(@frags,Invocation(T_CS(’\@symdef@pres@text’),Tokens(@frag))) if @frag;
      @frag=();
      my $n = shift(@tokens)->getString;
      push(@frags,Invocation(T_CS(’\@symdef@pres@arg’),T_OTHER($n+1))); }
    else {
      push(@frag,T_OTHER($tok->getString)); }} # IMPORTANT! Neutralize the tokens!
  push(@frags,Invocation(T_CS(’\@symdef@pres@text’),Tokens(@frag))) if @frag;
  Tokens(map($_->unlist,@frags)); }
DefConstructor(’\@symdef@pres@arg{}’, "<omdoc:recurse select=’#select’/>",
       afterDigest=>sub { my ($stomach, $whatsit) = @_;
  my $select = $whatsit->getArg(1);
  $select = ref $select ? $select->toString : '';
  $whatsit->setProperty(select=>"*[".$select."]"); });
DefConstructor('\@symdef@pres@text{}', "<omdoc:text>#1</omdoc:text>");
⟨/ltxml⟩#$
```

4.7 Including Externally Defined Semantic Macros

\requirepackage

```
⟨*package⟩
\def\requirepackage#1#2{\makeatletter\input{#1.sty}\makeatother}
⟨/package⟩
⟨*ltxml⟩
DefConstructor('\requirepackage{} Semiverbatim',"<omdoc:imports from='#2'/>",
       afterDigest=>sub { my ($stomach, $whatsit) = @_;
  my $select = $whatsit->getArg(1);
  RequirePackage($select->toString); });#$
⟨/ltxml⟩
```

4.8 Views

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

```
⟨*package⟩
\srefaddidkey{view}
\addmetakey*{view}{title}
\define@key{view}{load}{\requiremodules{#1}}
```

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

```
\newcounter{view}[section]
\newcommand\view@heading[2]{\stepcounter{view}%
{\textbf{View} \thesection.\theview: from #1 to #2}%
\sref@label@id{View \thesection.\theview}%
\ifx\view@title\@empty :\quad\else\quad(\view@title)\hfill\\\fi}
```

view The view environment only has an effect if the showmods option is set.

```
\ifmod@show\newsavebox{\viewbox}
\newenvironment{view}[3][]{\metasetkeys{view}{#1}\sref@target\stepcounter{view}
\begin{lrbox}{\viewbox}\begin{minipage}{.9\textwidth}
\importmodule{#1}\importmodule{#2}\gdef\view@@heading{\view@heading{#2}{#3}}}
{\end{minipage}\end{lrbox}
\setbox0=\hbox{\begin{minipage}{.9\textwidth}%
\noindent\view@@heading\rm%
\end{minipage}}
\smallskip\noindent\fbox{\vbox{\box0\vspace*{.2em}\usebox\viewbox}}\smallskip}
\else\newxcomment[]{view}\fi%ifmod@show
⟨/package⟩
⟨*ltxml⟩
DefKeyVal('view','id','Semiverbatim');
DefEnvironment('{view} OptionalKeyVals:view {}{}',
   "<omdoc:theory-inclusion from='#2' to='#3'>"
```

```
 .  "<omdoc:morphism>#body</omdoc:morphism>"
 ."</omdoc:theory-inclusion>");
⟨/ltxml⟩
```

4.9 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.[10] EdNote:10

```
⟨*package⟩
\define@key{module}{uses}{%
\@for\module@tmp:=#1\do{\activate@defs\module@tmp\export@defs\module@tmp}}
⟨/package⟩
```

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.

```
⟨*package⟩
\define@key{module}{usesqualified}{%
\@for\module@tmp:=#1\do{\activate@defs{qualified@\module@tmp}\export@defs\module@tmp}}
⟨/package⟩
```

4.10 Providing IDs for OMDoc Elements

To provide default identifiers, we tag all OMDoc elements that allow xml:id attributes by executing the numberIt procedure below.

```
⟨*ltxml⟩
Tag('omdoc:recurse',afterOpen=>\&numberIt,afterClose=>\&locateIt);
Tag('omdoc:imports',afterOpen=>\&numberIt,afterClose=>\&locateIt);
Tag('omdoc:theory',afterOpen=>\&numberIt,afterClose=>\&locateIt);
⟨/ltxml⟩
```

4.11 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.

```
⟨*package⟩
\def\csymdef{\@ifnextchar[{\@csymdef}{\@csymdef[]}}
\def\@csymdef[#1]#2{\@ifnextchar[{\@@csymdef[#1]{#2}}{\@@csymdef[#1]{#2}[0]}}
\def\@@csymdef[#1]#2[#3]#4#5{\@@symdef[#1]{#2}[#3]{#4}}
```

[10] EdNote: this issue is deprecated, it will be removed before 1.0.

```
⟨/package⟩
⟨*ltxml⟩
⟨/ltxml⟩
```

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

```
⟨*package⟩
\def\notationdef[#1]#2#3{}
⟨/package⟩
⟨*ltxml⟩
⟨/ltxml⟩
```

4.12 Finale

Finally, we need to terminate the file with a success mark for perl.

```
⟨ltxml⟩1;
```

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*. W3C Recommendation. World Wide Web Consortium (W3C), 2010. URL: `http://www.w3.org/TR/MathML3`.

[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`.

[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`.

[Koh10a] Michael Kohlhase. *metakeys.sty: A generic framework for extensible Metadata in LaTeX*. Self-documenting LaTeX package. Comprehensive TeX Archive Network (CTAN), 2010. URL: `http://www.ctan.org/tex-archive/macros/latex/contrib/stex/metakeys/metakeys.pdf`.

[Koh10b] Michael Kohlhase. *statements.sty: Structural Markup for Mathematical Statements*. Self-documenting LaTeX package. Comprehensive TeX Archive Network (CTAN), 2010. URL: `http://www.ctan.org/tex-archive/macros/latex/contrib/stex/statements/statements.pdf`.

[Mil] Bruce Miller. *LaTeXML: A LaTeX to XML Converter*. URL: `http://dlmf.nist.gov/LaTeXML/` (visited on 09/08/2011).

[RK11] Florian Rabe and Michael Kohlhase. "A Scalable Module System". Manuscript, submitted to Information & Computation. 2011. URL: `http://kwarc.info/frabe/Research/mmt.pdf`.

[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).

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