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

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

November 5, 2015

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 v1.4 (last revised 2015/04/02)

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 2.7).

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

qualifiedimports Section ??) 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.

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 [Koh15a] 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.

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

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 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 $\{a \,|\, a > 0\}$

Example 4: A Semantic Macro Definition with Test

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

2.4 Axiomatic Assumptions

In many ways, axioms and assumptions in definitions behave a lot like symbols (see [**RabKoh:WSMSML10**] for discussion). Therefore we provide the macro `\assdef` `\assdef` that can be used to mark up assumptions. Given a phrase ⟨*phrase*⟩ in a
EdN:2 definition[2], we can use `\assdef{`⟨*name*⟩`}{`⟨*phrase*⟩`}` to give this the symbol name
EdN:3 ⟨*name*⟩.[3]

2.5 Semantic Macros for Variables

Up to now, the semantic macros generated OpenMath and MathML markup where the heads of the semantic macros become constants (the `OMS` and `csymbol` elements in Figure 1). But sometimes we want to have semantic macros for variables, e.g. to associate special notation conventions. For instance, if we want to define mathematical structures from components as in Figure 5, where the semigroup operation $\circ$ is a variable epistemologically, but is a n-ary associative operator – we are in a semigroup after all. Let us call such variables **semantic variables** to contrast them from **semantic constants** generated by `\symdef` and `\symvariant`.

Semantic variables differ from semantic constants in two ways: *i*) they do not participate in the imports mechanism and *ii*) they generate markup with variables. In the case of Figure 5 we (want to) have the Xml markup in Figure 6. To associate the notation to the variables, we define semantic macros for them, here the macro

[2] EdNote: only definitions?
[3] EdNote: continue

Definition 3.17 Let $\langle G, \circ\rangle$ be a semigroup, then we call $e \in G$ a **unit**, iff $e \circ x = x \circ e = x$. A semigroup with unit $\langle G, \circ, e\rangle$ is called a **monoid**.

Example 5: A Definition of a Structure with “semantic variables”.

`\vardef` \op for the (semigroup) operation via the `\vardef` macro. `\vardef` works exactly like, except *i*) semantic variables are local to the current TEX group and *ii*) they generate variable markup in the XML

sTEX	`\vardef{op}[1]{\assoc\circ{#1}}`
OMDoc	% <notation> % <prototype> % <OMA> % <OMV name="op"/> % <expr name="a1"/> % <expr name="a2"/> % </OMA> % </prototype> % <rendering> % <mrow> % <render name="a1"/> % <mo>⎄</mo> % <render name="a2"/> % </mrow> % </rendering> % </notation>
LATEX	`\op{x,e}`
PDF/DVI	$x \circ e$
OPENMATH	% <OMA><OMV name="op"/><OMV name="x"/><OMV name="e"/></OMA>
MATHML	% <apply><ci>op</ci><ci>x</ci><ci>e</ci></apply>

Example 6: Semantic Variables in OPENMATH and MATHML

2.6 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 analogue to `\symdef` that supports this: use `\termdef[`⟨*keys*⟩`]{`⟨*cseq*⟩`}{`⟨*concept*⟩`}` to declare the macro `\`⟨*cseq*⟩ that expands to ⟨*concept*⟩. See Figure 7 for an example, where we use the `\capitalize` macro to adapt ⟨*concept*⟩ to the sentence beginning.[4]. 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 [Koh15b]. 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

`\termdef` `\capitalize` EdN:4 `\termref` `\symref`

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

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 7: Extending Example 3 with Term References

2.7 Modules and Inheritance

`module` Themodule 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 8 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 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[`load=⟨*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.[5] EdN:5

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

[5] EdNote: MK: document the other keys of `module`

2.8 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 8: Self-contained Modules via `importmodule`

In Example 8, 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*⟩`}`,

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

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 necessary to circumvent the recursive load pattern providing (logically spurious) `\importmodule` commands. Consider for instance module `bar` in Example 8, 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.

`\requiremodules`

`\inputref`

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

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

`\defpath`

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

instead of the second line in Example 8. 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].

2.9 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 is hindered by LaTeX groups). In many cases, we only want to *use* the semantic

[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

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 \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 9, where we open the module ring (see Figure ??) 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.

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

Example 9: Using Semantic Macros in Narrative Structures

\adoptmodule Still another import-like relation is the adoption of a module (see [Koh13] for details). We use the \adoptmodule macro for that.[6]

EdN:6

2.10 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.

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:online**].

[6]EdNote: elaborate this, so that the documentation becomes (more) self-contained.

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:online**] 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 [**sTeX:online**]. Until this is implemented the infrastructure is turned
`qualifiedimports` off by default, but we have already introduced the `qualifiedimports` option for the future.

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.

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 [Koh15b]. A way around this limitation would be to import the current module from the sTeX module signature (see Section 2.7) 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
EdN:7 declared *before* the current point. So we cannot have forward imports as in [7]

[7] EdNote: usemodule should work here; revise

```
\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). The options we are not using, we pass on to the `sref` package we require next.

```
⟨*package⟩
\newif\ifmod@show\mod@showfalse
\DeclareOption{showmods}{\mod@showtrue}
\newif\ifaux@req\aux@reqtrue
\DeclareOption{noauxreq}{\aux@reqfalse}
\newif\ifmod@qualified\mod@qualifiedfalse
\DeclareOption{qualifiedimports}{\mod@qualifiedtrue}
\DeclareOption*{\PassOptionsToPackage{\CurrentOption}{sref}}
\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.

```
\RequirePackage{sref}
\RequirePackage{presentation}
\RequirePackage{xspace}
\RequirePackage{mdframed}
\RequirePackage{xstring}
```

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.

```
\addmetakey{module}{cd}% no longer used
\addmetakey{module}{load}% ignored
\addmetakey*{module}{title}
\addmetakey*{module}{creators}
\addmetakey*{module}{contributors}
\addmetakey*{module}{srccite}
```

`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 the token register `\mod@id`.

```
\define@key{module}{id}{%
  \edef\this@module{%
    \expandafter\noexpand\csname module@defs@#1\endcsname%
  }%
  \csgdef{module@defs@#1}{}%
  \ifmod@qualified%
    \edef\this@qualified@module{%
      \expandafter\noexpand\csname module@defs@#1\endcsname%
    }%
    \csgdef{module@defs@qualified@#1}{}%
  \fi%
  \def\mod@id{#1}%
}%
```

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

```
\newcounter{module}[section]%
\newrobustcmd\module@heading{%
  \stepcounter{module}%
  \ifmod@show%
    \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%
  \fi%
}% mod@show
```

`module` Finally, we define the begin module command for the module environment. Much of the work has already been done in the keyval bindings, so this is quite simple. We store the file name (without extension) and extension of the module file in the global macros `\module@`⟨*name*⟩`@path` and `\module@`⟨*name*⟩`@ext`, so that we can use them later. The source of these two macros, `\mod@path` and `\mod@ext`, are defined in `\requiremodules`.

```
\newenvironment{module}[1][]{%
  \begin{@module}[#1]%
  \ifcsundef{mod@id}{}{% only define if components are!
    \ifcsundef{mod@path}{}{\csxdef{module@\mod@id @path}{\mod@path}}%
    \ifcsundef{mod@ext}{}{\csxdef{module@\mod@id @ext}{\mod@ext}}%
  }%
  \module@heading% make the headings
}{%
```

```
  \end{@module}%
}%
\ifmod@show\surroundwithmdframed{module}\fi%
```

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

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

`\activate@defs` To activate the `\symdef`s 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.

```
\def\activate@defs#1{%
  \ifcsundef{module@#1@activated}{\csname module@defs@#1\endcsname}{}%
  \@namedef{module@#1@activated}{true}%
}%
```

`\export@defs` `\export@defs{`⟨*mod*⟩`}` exports all the `\symdef`s 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*⟩.[8][9]

EdN:8
EdN:9

Naive understanding of this code: `#1` be will be expanded first, then `\this@module`, then `\active@defs`, then `\g@addto@macro`.

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

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.

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

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

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

`\importmodule` The `\importmodule[`⟨*file*⟩`]{`⟨*mod*⟩`}` macro is an interface macro that loads ⟨*file*⟩ and activates and re-exports the `\symdef`s from module ⟨*mod*⟩. As we will (probably) 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.

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

[9]EdNote: Jinbo: This part of code is extremely easy to generate bugs, cautiously edit this part of code.

```
\gdef\imported@modules{}
\srefaddidkey{importmodule}
\addmetakey{importmodule}{load}
\addmetakey[sms]{importmodule}{ext}
\addmetakey[false]{importmodule}{conservative}[true]
\newcommand\importmodule[2][]{%
  \metasetkeys{importmodule}{#1}%
  \ifx\imported@modules\@empty%
    \edef\imported@modules{#2}%
  \else%
    \edef\imported@modules{\imported@modules,#2}%
  \fi%
  \@exporttrue%
  \@importmodule[\importmodule@load]{#2}{\importmodule@ext}%
  \ignorespaces%
}%
```

`\@importmodule`

`\@importmodule[`⟨*file(with path, without extension)*⟩`]{`⟨*module*⟩`}{`⟨*extension*⟩`}` loads ⟨*file*⟩`.`⟨*extension*⟩ (if it is given) and activates the module ⟨*module*⟩. First `\@load` will store the base file name with full path, then check if `\module@`⟨*module*⟩`@path` is defined. If this macro is defined, that means the module has been already loaded, otherwise we load it. The actual loading will be executed by `\requiremodules`.

```
\newcommand\@importmodule[3][]{%
  {\@importingtrue% to shut up macros while in the group opened here
  \edef\@load{#1}%
  \ifx\@load\@empty%
    \relax%
  \else%
    \ifcsundef{module@#2@path}{%
      \requiremodules{#1}{#3}%
    }{%
      \edef\@path{\csname module@#2@path\endcsname}%
      \IfStrEq\@load\@path{% if the known path is the same as the requested one
        \relax% do nothing, it has already been loaded, else signal an error
  }{%
        \PackageError{modules}
        {{Module Name Clash\MessageBreak
        A module with name #2 was already loaded under the path "\@path"\MessageBreak
        The imported path "\@load" is probably a different module with the\MessageBreak
        same name; this is dangerous -- not importing}%
        {Check whether the Module name is correct}}%
      }%
    }%
  \fi}%
  \activate@defs{#2}% activate the module
  \if@export\export@defs{#2}\fi% export the module
}%
```

`\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.

```
\newrobustcmd\usemodule[2][]{\@exportfalse\importmodule[#1]{#2}}%
```

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

```
\newrobustcmd\importmodules[1]{\@for\@I:=#1\do{\activate@defs\@I}}%
```

`\importOMDocmodule` for the LaTeX side we can just re-use `\importmodule`, for the LaTeXML side we
EdN:10 have a full URI anyways. So things are easy.[10]

```
\newrobustcmd\importOMDocmodule[3][]{\importmodule[#1]{#3}}%
```

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

```
\let\metalanguage=\importmodule%
```

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

```
\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.

```
\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.

`conceptdef`

```
\srefaddidkey{conceptdef}%
\addmetakey*{conceptdef}{title}%
\addmetakey{conceptdef}{subject}%
\addmetakey*{conceptdef}{display}%
\def\conceptdef@type{Symbol}%
\newrobustcmd\conceptdef[2][]{%
  \metasetkeys{conceptdef}{#1}%
  \ifx\conceptdef@display\st@flow\else{\stDMemph{\conceptdef@type} #2:}\fi%
  \ifx\conceptdef@title\@empty~\else~(\stDMemph{\conceptdef@title})\par\fi%
}%
```

EdN:11 [11]

`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

[10]EdNote: MK@DG: this macro is seldom used, maybe I should just switch arguments.
[11]EdNote: MK@DG: maybe we need to add DefKeyVals here?

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.

```
\newif\if@symdeflocal%
\srefaddidkey{symdef}%
\define@key{symdef}{local}[true]{\@symdeflocaltrue}%
\define@key{symdef}{primary}[true]{}%
\define@key{symdef}{assocarg}{}%
\define@key{symdef}{bvars}{}%
\define@key{symdef}{bargs}{}%
\addmetakey{symdef}{name}%
\addmetakey*{symdef}{title}%
\addmetakey*{symdef}{description}%
\addmetakey{symdef}{subject}%
\addmetakey*{symdef}{display}%
```

EdN:12 [12]

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

```
\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]}%
```

`\@@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.

```
\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%
  \metasetkeys{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, which when invoked with an optional argument ⟨*opt*⟩ calls `\modules@`⟨*sym*⟩`@pres@`⟨*opt*⟩ provided by the `\symvariant` macro.

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

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

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%
  \ifcsundef{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*⟩ which calls the intermediate function and handles the optional argument.

```
    \expandafter\g@addto@macro\this@module%
    {\expandafter\mod@newcommand\csname #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#2@qualified\endcsname{\csname modules@#2@pres@qualified\endcsn
    \fi%
  }% mod@qualified
\fi% symdeflocal
```

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

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

`\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 by the macro \⟨*sym*⟩[⟨*var*⟩] induced by the \symdef.[13]

EdN:13

```
\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.

```
  \ifcsundef{mod@id}{}{%
    \expandafter\g@addto@macro\this@module%
    {\expandafter\mod@newcommand\csname modules@#1@pres@#3\endcsname[#2]{#4}}%
  }%
}%
```

\resymdef

This is now deprecated.

```
\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}{use the \protect\symvariant instead!}%
}%
```

\abbrdef

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

```
\let\abbrdef\symdef%
```

4.4 Defining Math Operators

\DefMathOp

\DefMathOp[⟨*key pair*⟩]{definition} will take 2 arguments. ⟨*key pair*⟩ should be something like [name=...], for example, [name=equal]. Though \setkeys, \defmathop@name will be set. Further definition will be done by \symdef.

```
\define@key{DefMathOp}{name}{%
  \def\defmathop@name{#1}%
}%
\newrobustcmd\DefMathOp[2][]{%
  \setkeys{DefMathOp}{#1}%
  \symdef[#1]{\defmathop@name}{#2}%
}%
```

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

4.5 Axiomatic Assumptions

\assdef We fake it for now, not clear what we should do on the LaTeX side.

```
\newcommand\assdef[2][]{#2}
```

4.6 Semantic Macros for Variables

\vardef We do the argument parsing like in `\symdef` above, but add the `local` key. All the other changes are in the LaTeXML binding exclusively.

```
\def\vardef{%
  \@ifnextchar[{\@vardef}{\@vardef[]}%
}%
\def\@vardef[#1]#2{%
  \@ifnextchar[{\@@vardef[#1]{#2}}{\@@vardef[#1]{#2}[0]}%
}%
\def\@@vardef[#1]#2[#3]#4{%
  \def\@test{#1}%
  \ifx\@test\@empty%
    \@@symdef[local]{#2}[#3]{#4}%
  \else%
    \symdef[local,#1]{#2}[#3]{#4}%
  \fi%
}%
```

4.7 Testing Semantic Macros

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

```
\addmetakey{symtest}{name}%
\addmetakey{symtest}{variant}%
\newrobustcmd\symtest[3][]{%
  \if@importing%
  \else%
    \metasetkeys{symtest}{#1}%
    \par\noindent \textbf{Symbol}~%
    \ifx\symtest@name\@empty\texttt{#2}\else\texttt{\symtest@name}\fi%
    \ifx\symtest@variant\@empty\else\ (variant \texttt{\symtest@variant})\fi%
    \ with semantic macro %
    \texttt{\textbackslash #2\ifx\symtest@variant\@empty\else[\symtest@variant]\fi}%
    : used e.g. in \ensuremath{#3}%
  \fi%
}%
```

\abbrtest

```
\addmetakey{abbrtest}{name}%
\newrobustcmd\abbrtest[3][]{%
  \if@importing%
  \else%
    \metasetkeys{abbrtest}{#1}%
    \par\noindent \textbf{Abbreviation}~%
```

```
  \ifx\abbrtest@name\@empty\texttt{#2}\else\texttt{\abbrtest@name}\fi%
  : used e.g. in \ensuremath{#3}%
 \fi%
}%
```

4.8 Symbol and Concept Names

`\termdef`

```
\def\mod@true{true}%
\addmetakey[false]{termdef}{local}%
\addmetakey{termdef}{name}%
\newrobustcmd\termdef[3][]{%
 \metasetkeys{termdef}{#1}%
 \expandafter\mod@newcommand\csname#2\endcsname[0]{#3\xspace}%
 \ifx\termdef@local\mod@true%
 \else%
  \ifcsundef{mod@id}{}{%
   \expandafter\g@addto@macro\this@module%
   {\expandafter\mod@newcommand\csname#2\endcsname[0]{#3\xspace}}%
  }%
 \fi%
}%
```

`\capitalize`

```
\def\@capitalize#1{\uppercase{#1}}%
\newrobustcmd\capitalize[1]{\expandafter\@capitalize #1}%
```

`\mod@termref` `\mod@termref{`⟨*module*⟩`}{`⟨*name*⟩`}{`⟨*nl*⟩`}` determines whether the macro `\module@`⟨*module*⟩`@path` 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.[14]

EdN:14

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

[14] EdNote: MK: this should be rethought, in particular the local reference does not work!

4.9 Dealing with Multiple Files

`cpath` package deals with the canonicalization of paths. `\@cpath` will canonicalize a path and store the result into `\@CanPath`. To print a canoncalized path, simply use `\cpath{`⟨*path*⟩`}`.

`\@rinput` `\@rinput{`⟨*path to the current file without extension*⟩`}{`⟨*extension*⟩`}` allows loading modules with relative path. For example, `\@rinput{foo/bar/B}{tex}` will load `foo/bar/B.tex`. [15]

EdN:15

```
\RequirePackage{cpath}
\def\CurrentDir{}%
\newrobustcmd{\@rinput}[2]{%
  \@cpath{\CurrentDir#1}%
  \StrCut[\value{RealAddrNum}]{/\@CanPath}{/}\@TempPath\@Rubbish%
  \StrCut[1]{\@TempPath/}{/}\@Rubbish\@DirPath%
  \edef\CurrentDir{\@DirPath}%
%  \edef\mod@path{}% what should I put in here???
%  \edef\mod@ext{}%
  \input{\@CanPath.#2}%
  \def\CurrentDir{}%
}%
```

`\defpath` `\defpath[optional argument]{macro name}{base path}` defines a new macro which can take another path to formal one integrated path. For example, `\MathHub` in every `localpaths.tex` is defined as:

`\defpath{MathHub}{/path/to/localmh/MathHub}`

then we can use `\MathHub` to form other paths, for example,

`\MathHub{source/smglom/sets}`

will generate `/path/to/localmh/MathHub/source/smglom/sets`.

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

4.10 Loading Module Signatures

4.10.1 Selective Inclusion

`\requiremodules` this macro loads the modules in a file and makes sure that no text is deposited (we set the flags `\mod@showfalse` and `\@importingtrue` in the local group). It also remembers the file name and extension in `\mod@path` and `\mod@ext` so that `\begin{module}` can pick them up later.

```
\newrobustcmd\requiremodules[2]{%
  \mod@showfalse%
  \@importingtrue% save state and ensure silence while reading sms
```

[15] EdNote: Jinbo: How to handle `mod@path`?

```
  \edef\mod@path{#1}%
  \edef\mod@ext{#2}% set up path/ext
  \input{#1.#2}%
}%
```

`\@requiremodules` the internal version of `\requiremodules` for use in the `*.aux` file. We disable it at the end of the document, so that when the `aux` file is read again, nothing is loaded.

```
\newrobustcmd\@requiremodules[2]{%
  \if@tempswa\requiremodules{#1}{#2}\fi%
}%
```

`\inputref` `\inputref{`⟨*path to the current file without extension*⟩`}` supports both absolute path and relative path, meanwhile, records the path and the extension (not for relative path). [16]

EdN:16

```
\newrobustcmd\inputref[2][]{%
  \def\@Slash{/}
  \edef\@load{#2}%
  \StrChar{\@load}{1}[\@testchar]
  \ifx\@testchar\@Slash%
    \edef\mod@path{#2}%
    \edef\mod@ext{tex}%
    \input{#2}%
  \else%
    \@rinput{#2}{tex}%
  \fi%
}%
```

4.11 Including Externally Defined Semantic Macros

`\requirepackage`

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

4.12 Deprecated Functionality

`\sinput*`

```
\newrobustcmd\sinput[1]{%
  \PackageError{modules}%
  {The \protect\sinput macro is deprecated}{use the \protect\input instead!}%
}%
\newrobustcmd\sinputref[1]{%
  \PackageError{modules}%
  {The \protect\sinputref macro is deprecated}{use the \protect\inputref instead!}%
}%
```

[16] EdNote: MK: maybe do something with a non-standard (i.e. non-tex) extension with an optional argument?

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.[17]

EdN:17

```
\define@key{module}{uses}{%
  \@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.

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

`\coolurion/off`

```
\def\coolurion{\PackageWarning{modules}{coolurion is obsolete, please remove}}%
\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.

```
\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}%
}%
```

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

```
\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)

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

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.

```
\newrobustcmd\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

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

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

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

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

```
\def\file@path#1{%
  \setcounter{@pl}{0}%
  \forpathlist{\stepcounter{@pl}\listadd\@pathlist}{#1}%
  \def\do##1{%
    \ifnumequal{\value{@pl}}{1}{}{%
      \addtocounter{@pl}{-1}%
      \ifnumequal{\value{@pl}}{1}{##1}{##1/}%
    }%
  }%
  \dolistloop{\@pathlist}%
}%
⟨/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`.

```
\def\@NEWcurrentprefix{}
\def\NEWrequiremodules#1{%
```

```
  \def\@pref{\file@path{#1}}%
  \ifx\@pref\@empty%
  \else%
    \xdef\@NEWcurrentprefix{\@NEWcurrentprefix/\@pref}%
  \fi%
  \edef\@input@me{\@NEWcurrentprefix/\file@name{#1}}%
  \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.

[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.

[Koh15a] Michael Kohlhase. *metakeys.sty: A generic framework for extensible Metadata in LaTeX*. Tech. rep. Comprehensive TeX Archive Network (CTAN), 2015. URL: http://www.ctan.org/tex-archive/macros/latex/contrib/stex/metakeys/metakeys.pdf.

[Koh15b] Michael Kohlhase. *statements.sty: Structural Markup for Mathematical Statements*. Tech. rep. Comprehensive TeX Archive Network (CTAN), 2015. 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).