Semantic Macros and Module Scoping in STEX*

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

May 7, 2008

Abstract

The `modules` packge 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 depency 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 v0.9a (last revised 2006/01/13)

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 takes up the name of the mathematical object. This can range from simple definitions like `\def\Reals{{\mathbb R}}` for indivicual mathematical objects to more complex (functional) ones object constructors like `\def\SmoothFunctionsOn#1{{\cal{C}}^\infty(#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.

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 Modules

`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).

`\importmodule` 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. Thus `\importmodule` induces the semantic inheritance relation and

EdNote(1) `usesqualified`[1] for macros imported with a prefix (this is used whenever we have conflicting names for macros inherited from different modules).

2.2 Semantic Macros

`\symdef` 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.

A key `local` can be added to ⟨*keys*⟩ to specify that the symbol is local to the module and is invisible outside. The key-value pair `aliases=`⟨*symname*⟩

[1] EdNote: do an importqualified as well

specifies that the defined symbol ⟨*cseq*⟩ is a presentational variant of the symbol ⟨*symname*⟩.

Finallly, the keys `cmml`, `cattrs`, and `definitionURL` can be used to specify the Content-MathML encoding of the symbols. They key-value pair `cmml=`⟨*elt*⟩ specifies that the semantic macro corresponds to the Content-MathML element with the name ⟨*elt*⟩, `cattrs=`⟨*attrtring*⟩ its argument string and `definitionURL` allows to specify the `definitionURL` attribute on that element. The most common case will be a symbol definition of the following form:

`\symdef[cmml=csymbol,definitionURL=`⟨*URI*⟩`]{`⟨*cseq*⟩`}[`⟨*args*⟩`]{`⟨*definiens*⟩`}`

where ⟨*URI*⟩ is the URI pointing to the location of the XML file generated from the current LaTeX file.

`\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 1 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}
```

is formatted by STeX to

What is the sum of the first n odd numbers, i.e. $\sum_{i=1}^{n} 2i - 1$?

Example 1: Semantic Markup in a `module` context

2.3 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 enthropy is aggravated by the fact that modules are very self-contiained 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 4) 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 corresponing module files.

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

Example 2: Self-contained Modules via `importmodule`

In Example 2, 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.

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 formatte (it is usually set to something like 15). Therefore, it may be necessary to circumvent the recursive load pattern providing (logically spurious) `\importmodule` commands. Consider for instance module `bar` in Example 2, say that `bar` already has load deph 15, then we cannot naivedly import it in this way. If module `bar` depended say on a module `base` on the critical load path, then `\requiremodules` we could add a statement `\requiremodules{../base}` in the second line. This would load the modules from `../base.sms` in advance (uncritical, since it has load depth 10), so that it would not have to be re-loaded in the critical path of the module `foo`. Solving the load depth problem.

2.4 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 correponds 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 corresponing 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 The Implementation

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

```
⟨*package⟩
\newif\ifmod@env\mod@envfalse
\newif\ifmod@id\mod@idfalse
\newif\ifmod@display\mod@displayfalse
\newif\ifmod@uses\mod@usesfalse
\newif\ifmod@usesqualified\mod@usesqualifiedfalse
\DeclareOption{env}{\mod@envtrue}
\DeclareOption{id}{\mod@idtrue}
\DeclareOption{uses}{\mod@usestrue}
\DeclareOption{display}{\mod@displaytrue}
\DeclareOption{usesqualified}{\mod@usesqualifiedtrue}
```

Now, we define two collective options, which are equivalent to turning on all the other options.

```
\def\modtrue{\mod@idtrue\mod@usestrue\mod@displaytrue\mod@usesqualifiedtrue}
\DeclareOption{draft}{\modtrue}
\DeclareOption{all}{\modtrue}
```

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 KeyVal package is loaded (in the right version). for LaTeXML, we also initialize the package inclusions.

```
⟨package⟩\RequirePackage{keyval}[1997/11/10]
⟨*ltxml⟩
# -*- CPERL -*-
package LaTeXML::Package::Pool;
use strict;
use LaTeXML::Global;
use LaTeXML::Package;
RequirePackage('keyval');
⟨/ltxml⟩
```

3.1 Modules

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⟩\define@key{module}{cd}{}
⟨ltxml⟩DefKeyVal('Module','cd','Semiverbatim');
```

module:id For a module with [id=name], we create a macro \module@defs@name and initialize it. Furthermore, we save the name in \mod@id.

```
⟨*package⟩
\define@key{module}{id}{%
  \edef\this@module{\expandafter\noexpand\csname module@defs@#1\endcsname}%
  \edef\this@qualified@module{\expandafter\noexpand\csname module@defs@qualified@#1\endcsname}%
  \global\@namedef{module@defs@#1}{}
  \global\@namedef{module@defs@qualified@#1}{}
  \def\mod@id{#1}}
⟨/package⟩
⟨ltxml⟩DefKeyVal('Module','id','Semiverbatim');
```

activate@defs To activate the symdefs from a given module xxx, we call the macro \module@defs@xxx.

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

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

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

module:uses For each the module name xxx specified in the uses key, we activate their symdefs and we export the local symdefs.[2]

EdNote(2)

[2] EdNote: this isue is deprecated, it will be removed before 1.0.

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

On the LaTeXML side, the corresponding shippet ist this: the `use_module` subroutine performs depth-first load of definitions of the used modules

```
⟨*ltxml⟩
sub use_module {
  my($module)=@_;
  $module = ToString($module);
  # Depth-first load definitions from used modules
  foreach my $used_module (@{ LookupValue("module_${module}_uses") || []}){
    use_module($used_module); }
  # then load definitions for this module
  $STATE->activateScope("module:$module"); }
⟨/ltxml⟩
```

`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⟩
```

`\show@mod@keys` The `\show@mod@keys` macro is used for the draft mode, they allow to annotate the document with reminders of the key values in the modules.

```
⟨*package⟩
\def\show@mod@keys@aux{%
\@ifundefined{mod@id}{}{\ifmod@id{id=\mod@id},\fi}%
\@ifundefined{mod@display}{}{\ifmod@display{display=\mod@display}\fi}
\@ifundefined{mod@uses}{\relax}{\ifmod@uses{uses=\mod@uses},\fi}
\@ifundefined{mod@usesqualified}\relax\else
            \ifmod@usesqualified{usesqualified=\mod@usesqualified},\fi\fi}
\def\clear@mod@keys{\let\mod@usesqualified=\relax\mod@uses=\relax
            \let\st@id=\relax\let\st@display=\relax}
\def\show@mod@keys#1{\footnote{#1[\show@mod@keys@aux]}\clear@mod@keys}
⟨/package⟩
```

`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.

```
⟨package⟩\newenvironment{module}[1][]{\setkeys{module}{#1}\ifmod@env\show@mod@keys{module}\fi}{}
```

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

```
⟨*ltxml⟩
DefEnvironment('{module} OptionalKeyVals:Module', sub {
 my ($doc, $keyvals, %props) = @_;
 unless ($props{excluded}) {
   my $theory = $keyvals->getValue('id');
```

```
  $theory = ref $theory ? $theory->toString : 'UNDEFINED';
  AssignValue(current_theory => $theory);
  $doc->openElement('omdoc:theory', 'xml:id' => $theory);
  my $uses = $keyvals->getValue('uses');
  $uses = ref $uses ? $uses->toString || '' : '';
  $uses =~ s/\s+//g; $uses=~ s/^\{//; $uses=~ s/\}$//;
  my $module_paths = LookupValue('module_paths') || {};
  foreach my $used(split(',',$uses)) {
    my $file = $module_paths->{$used}; $file .= '.omdoc#' if $file;
    $doc->openElement('omdoc:imports', 'from' => $file.$used);
    $doc->closeElement('omdoc:imports'); }
  $doc->absorb($props{body}) if $props{body};
  $doc->closeElement('omdoc:theory'); }
return; },
      beforeDigest=>\&useTheoryItemizations,
      afterDigestBegin=>sub { my($stomach, $whatsit)=@_;

      $whatsit->setProperty(excluded=>LookupValue('excluding_modules'));

      my $keys = $whatsit->getArg(1);
      my($id, $cd, $uses)=$keys
&& map(ToString($keys->getValue($_)),qw(id cd uses));
      $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');

      AssignValue(current_module => $id);
      AssignValue(module_cd => $cd) if $cd;
      my @uses = ();
      if($uses){
$uses =~ s/\s//g; $uses=~ s/^\{//; $uses=~ s/\}$//;
@uses = split(',',$uses); }
      AssignValue("module_${id}_uses" => [@uses], 'global');
      use_module($id);
      return; });
⟨/ltxml⟩
```

`\importmodule` The `\importmodule[`⟨*file*⟩`]{`⟨*mod*⟩`}` macro is an interface macro that loads ⟨*file*⟩ and activates and re-exports the symdefs from module ⟨*mod*⟩.[3]

EdNote(3)

```
⟨*package⟩
\newcommand{\importmodule}[2][]{\def\@test{#1}%
\ifx\@test\@empty\else\requiremodules{#1}\fi
\activate@defs{#2}\export@defs{#2}}
⟨/package⟩
⟨*ltxml⟩
```

[3] EdNote: document it above, and implement it in LaTeXML

```
DefPrimitive('\importmodule[]{}', sub {
  my($stomach,$path,$module)=@_;
  my $GULLET = $stomach->getGullet;
  $module = Digest($path)->toString;
  if(LookupValue('module_'.$module.'_loaded')) {}
  else {
    AssignValue('module_'.$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;});
⟨/ltxml⟩
```

3.2 Semantic Macros

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

`symdef:aliases` This optional key aliases for the symdef function allows us to provide additional arguments representing other functions that are aliased by the one currently being defined.

```
⟨package⟩\define@key{symdef}{aliases}{}
⟨ltxml⟩DefKeyVal('symdef','aliases','Semiverbatim');
```

`symdef:local` 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}`

```
⟨package⟩\define@key{symdef}{local}[true]{\@symdeflocaltrue}
⟨ltxml⟩DefKeyVal('symdef','local','Semiverbatim','true');
```

`symdef:` The keys `cmmml`, `cattrs`, and `definitionURL` are not used in TEX/LATEX bindings[4].

EdNote(4)

```
⟨*package⟩
\define@key{symdef}{cmml}{}
\define@key{symdef}{cattrs}{}
\define@key{symdef}{definitionURL}{}
⟨/package⟩
⟨*ltxml⟩
DefKeyVal('symdef','cmml','Semiverbatim','true');
DefKeyVal('symdef','cattrs','Semiverbatim','true');
```

[4] EdNote: decide what we want to do about them in the future.

```
DefKeyVal('symdef','definitionURL','Semiverbatim','true');
⟨/ltxml⟩
```

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

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

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 check for the local keyword. Then we set all the keys that have been provided as arguments: aliases, local. First, using \providecommand we initialize the intermediate function, the one that can be changed internally with \redefine and then we link the actual function to it, again with \providecommand. We check if the switch for the local scope is set: if it is we are done, since this function has a local scope. Otherwise, we add these two functions to the module's pool of defined macros using \g@addto@macro. We add both functions so that we can keep the link between the real and the intermediate function whenever we inherit the module. Finally, using \g@addto@macro we add the two functions to the qualified version of the module.

```
\@symdeflocalfalse\setkeys{symdef}{#1}
   \expandafter\providecommand\csname modules@#2@pres\endcsname[#3]{#4}
   \expandafter\def\csname#2\endcsname{\csname modules@#2@pres\endcsname}
   \if@symdeflocal\else
   \@ifundefined{mod@id}{}{
   \expandafter\g@addto@macro\this@module{\expandafter\providecommand\csname modules@#2@pres\en
   \expandafter\g@addto@macro\this@module{\expandafter\def\csname#2\endcsname{\csname modules@#
   \expandafter\g@addto@macro\this@qualified@module{\expandafter\providecommand\csname modules@
   \expandafter\g@addto@macro\this@qualified@module{\expandafter\def\csname#2atqualified\endcsn
   }\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 3.5).

```
⟨*ltxml⟩
DefMacro('\symdef OptionalKeyVals:symdef {}[][]{}',
      sub {
my($self,@args)=@_;
# print STDERR "excluding" if LookupValue('excluding_modules');
(Invocation(T_CS('\@symdef'),@args)->unlist,
 (LookupValue('excluding_modules') ? ()
  : (Invocation(T_CS('\@symdef@pres'), @args)->unlist))); });
```

In the LaTeXML bindings we also use a `\@symdef` macro for doing the main work The implementation is similar to `\LXMathDef` (see `latexml.ltxml`), but defaults the CD to the module's cd, and stashes the definition in the module.[5][6]

EdNote(5)
EdNote(6)

```
DefPrimitive('\@symdef OptionalKeyVals:symdef {}[][]{}', sub {
  my($stomach,$keys,$cs,$nargs,$opt,$presentation)=@_;
  my($name,$cd,$role,$cmml,$cattrs,$definitionURL)=$keys
    && map($_ && $_->toString,map($keys->getValue($_), qw(name cd role cmml cattrs definitionUR
  $cd = LookupValue('module_cd') unless $cd;
  $nargs = (ref $nargs ? $nargs->toString : $nargs || 0);
  my $module = LookupValue('current_module');
#  print STDERR "Define ".Stringify($cs)." in $module\n";
  my $scope = (($keys && ($keys->getValue('local') || '' eq 'true')) ? 'module_local' : 'module
  my $attr="name='#name' meaning='#meaning' omcd='#omcd'";
  DefConstructorI("\\".$cs->toString,convertLaTeXArgs($nargs,$opt),
 ($nargs == 0
  ? "<ltx:XMTok $attr scriptpos='#scriptpos'/>"
  : "<ltx:XMApp scriptpos='#scriptpos'>"
  .   "<ltx:XMTok $attr scriptpos='#operator_scriptpos'/>"
  .   join('',map("<ltx:XMArg>#$_</ltx:XMArg>", (1..$nargs)))
  ."</ltx:XMApp>"),
 properties => {name=>$name, meaning=>$cs->toString,omcd=>$cd,role => $role},
 scope=>$scope);
  return; });
⟨/ltxml⟩
```

`\redefine` We can use this function to redefine our intermediate presentational function inside the modules[7]

EdNote(7)

```
⟨package⟩\def\redefine#1[#2]#3{\expandafter\renewcommand\csname modules@#1@pres\endcsname[#2]{#3
```

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

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

3.3 Loading Module Signatures

EdNote(8)

[8] Before we can come to the functionality we want to offer, we need some auxiliary

[5] EdNote: @IOAN: [local] does not seem to work yet.
[6] EdNote: we still need to wrap the arguments in `ltx:XMArg` elements!
[7] EdNote: does not seem to have a LaTeXML counterpart yet!
[8] EdNote: talk about module signatures above (SMS files)

functions that deal with path names.

`\mod@simplify` The `\mod@simplify` macro removes `xxx/..` from a string. eg: `aaa/bbb/../ddd` goes to `aaa/ddd`. This is used to normalize relative path names below.

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

It is based on the `\mod@simpl` macro

```
211 \def\mod@simpl#1/#2\relax{%\message{ 1 = #1, 2 = #2 }%
212 \def\mod@test{}\ifx\mod@blaaaa\mod@test\edef\mod@savedprefix{}\def\mod@blaaaa{aaa}\else\fi
213 \def\mod@comp{#2}\ifx\mod@test\mod@comp\edef\mod@savedprefix{\mod@savedprefix#1}%
214 \else\mod@simplhelp#1/#2\relax\fi}
```

which in turn is based on a helper macro

```
215 \def\mod@simplhelp#1/#2/#3\relax{%
216 \def\mod@test{}\def\mod@tust{#2}\def\mod@tist{#3}\def\mod@tost{..}\ifx\mod@test\mod@tist%
217 \ifx\mod@tost\mod@tust\edef\mod@savedprefix{}\else\edef\mod@savedprefix
218 {\mod@savedprefix#1/#2}\fi \else\ifx\mod@tost\mod@tust\mod@simpl#3\relax%
219 \else\edef\mod@savedprefix{\mod@savedprefix#1/}\mod@simplhelp#2/#3\relax\fi\fi}%
```

EdNote(9) We will need a switch[9]

```
220 \newif\ifmodules
```

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

`\reg` We initialize the `\reg` macro with the empty string.

```
221 \gdef\reg{}
```

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

```
222 \def\mod@update#1{\def\mod@empty{}
223 \ifx\reg\mod@empty\xdef\reg{#1}\else\xdef\reg{#1@\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 modulestrue, otherwise make modulesfalse. The macro `\mod@search` will look at ifinclude and update the registry for modulestrue or do nothing for modulesfalse.

```
224 \def\mod@check#1@#2///#3\relax{%
225 \def\mod@empty{}\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

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

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

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

[9] EdNote: Rares, say why?

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

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

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

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

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

```
\def\mod@reguse#1{\modulesfalse\expandafter\mod@check\reg @///#1\relax}
%   \end{macrocode}
% \end{macro}
%
% \begin{macro}{\mod@prefix}
% This is a local macro for storing the path prefix, we initialize it as the empty
% string.
%    \begin{macrocode}
\def\mod@prefix{}
```

`\mod@updatedpre` This macro allows to update the path prefix with the last part of the new path

```
\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 function defined above

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

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

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⟩
```

3.4 Selective Inclusion

`\requiremodules`

```
⟨*package⟩
\newcommand{\requiremodules}[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}
\ifmodules\else
\mod@updatesms    % update the file to contain the .sms extension
\let\newreg\reg   % use to compare, in case the .sms file was loaded before
\mod@search{\mod@savedprefix} % update registry
\ifx\newreg\reg\else\input{\mod@savedprefix}\fi % check if the registry was updated and load if
\fi}}
⟨/package⟩
⟨*ltxml⟩
DefPrimitive('\requiremodules{}', sub {
  my($stomach,$module)=@_;
  my $GULLET = $stomach->getGullet;
  $module = Digest($module)->toString;
  if(LookupValue('module_'.$module.'_loaded')) {}
  else {
    AssignValue('module_'.$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 { $_[0]->egroup; 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\reg   % use to compare, in case the .sms file was loaded before
\mod@search{\mod@savedprefix} % update registry
\ifx\newreg\reg%\message{This file has been previously introduced}
\else\input{\mod@savedprefix}\fi}}
⟨/package⟩
⟨*ltxml⟩
DefPrimitive('\sinput{}', sub {
  my($stomach,$module)=@_;
  my $GULLET = $stomach->getGullet;
  $module = Digest($module)->toString;
  if(LookupValue('module_'.$module.'_loaded')) {}
  else {
    AssignValue('module_'.$module.'_loaded' => 1, 'global');
    $stomach->bgroup;
    AssignValue('last_module_path', $module);
    $GULLET->unread(T_CS('\end@requiredmodule'));
    $GULLET->input($module,['tex']);
       }
  return;});
⟨/ltxml⟩
```

EdNote(10)

[10]

3.5 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,$opt,$presentation)=@_;
  Invocation(T_CS('\@symdef@pres@aux'),
     $cs,
     ($nargs || Tokens(T_OTHER(0))),
     symdef_presentation_pmml($cs,ToString($nargs)||0,$presentation),
     symdef_presentation_TeX($presentation),
     $keys)->unlist; });
```

Generate the expansion of a symdef'd macro using special arguments

```
sub symdef_presentation_pmml {
  my($cs,$nargs,$presentation)=@_;
  my @toks = $presentation->unlist;
  # Remove leading space
  while(@toks && $toks[0]->equals(T_SPACE)){
```

[10] 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.

```
    pop(@toks); }
  $presentation = Tokens(@toks);
  # Wrap with \use, unless already has a recognized formatter.
  $presentation = Invocation(T_CS('\use'),$presentation)
    unless @toks && ($toks[0]->toString =~ /^\\(infix|prefix|postfix|assoc|use|mixfixi|mixfixa|
  # Low level substitution.
  my @args = map(Invocation(T_CS('\@SYMBOL'),T_OTHER("pres_arg:".($_+1))),1..$nargs);
  $presentation = Tokens(LaTeXML::Expandable::substituteTokens($presentation,@args));
  $presentation; }

DefConstructor('\@symdef@pres@aux{}{}{}{} OptionalKeyVals:symdef',
        "<omdoc:symbol name='#1'/>"
       ."<omdoc:presentation for='#for' role='#role'>"
         ."#3"
         ."<omdoc:style format='TeX'>#4</omdoc:style>"
       ."</omdoc:presentation>",
       afterDigest=>sub { my ($stomach, $whatsit) = @_;
  my $keys = $whatsit->getArg(5);
  $whatsit->setProperties(for=>"#".ToString($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⟩
```

3.6 Including Externally Defined Semantic Macros

\requirepackage

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

3.7 Finale

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

```
⟨ltxml⟩1;
```

4 Utility