The sTeX3 Manual *

Michael Kohlhase, Dennis Müller
FAU Erlangen-Nürnberg
http://kwarc.info/

2022-03-06

Abstract

sTeX is a collection of LaTeX package that allow to markup documents semantically without leaving the document format, essentially turning LaTeX into a document format for mathematical knowledge management (MKM).
sTeX augments LaTeX with

- *Semantic macros* that denote and distinguish between mathematical concepts, operators, etc. independent of their notational presentation,
- A powerful *module system* that allows for authoring and importing individual fragments containing document text and/or semantic macros, independent of – and without hard coding – directory paths relative to the current document,
- A mechanism for exporting sTeX documents to (modular) XHTML, preserving all the semantic information for semantically informed knowledge management services.

This is the user manual for the sTeX package and associated software. It is primarily directed at end-users who want to use sTeX to author semantically enriched documents. For the full documentation, see the sTeX documentation

*Version 3.0 (last revised 2022-03-06)

Contents

Boxes like this one contain implementation details that are mostly relevant for more advanced use cases, might be useful to know when debugging, or might be good to know to better understand how something works. They can easiyl be skipped on a first read.

Boxes like this one explain how some sTeX concept relates to the Mmt/OMDoc system, philosophy or language.

Chapter 1

What is sTeX?

Formal systems for mathematics (such as interactive theorem provers) have the potential to significantly increase both the accessibility of published knowledge, as well as the confidence in its veracity, by rendering the precise semantics of statements machine actionable. This allows for a plurality of added-value services, from semantic search up to verification and automated theorem proving. Unfortunately, their usefulness is hidden behind severe barriers to accessibility; primarily related to their surface languages reminiscent of programming languages and very unlike informal standards of presentation.

sTeX minimizes this gap between informal and formal mathematics by integrating formal methods into established and widespread authoring workflows, primarily LaTeX, via non-intrusive semantic annotations of arbitrary informal document fragments. That way formal knowledge management services become available for informal documents, accessible via an IDE for authors and via generated *active* documents for readers, while remaining fully compatible with existing authoring workflows and publishing systems.

Additionally, an extensible library of reusable document fragments is being developed, that serve as reference targets for global disambiguation, intermediaries for content exchange between systems and other services.

Every component of the system is designed modularly and extensibly, and thus lay the groundwork for a potential full integration of interactive theorem proving systems into established informal document authoring workflows.

The general sTeX workflow combines functionalities provided by several pieces of software:

- The sTeX package to use semantic annotations in LaTeX documents,
- RustTeX to convert `tex` sources to (semantically enriched) `xhtml`,
- The Mmt software, that extracts semantic information from the thus generated `xhtml` and provides semantically informed added value services.

Chapter 2

Quickstart

2.1 Setup

2.1.1 The sTeX IDE

TODO: VSCode Plugin

2.1.2 Manual Setup

Foregoing on the sTeX IDE, we will need several pieces of software; namely:

- **The sTeX-Package** available here.

  sTeX is also available on CTAN and in TeXLive.

- To make sure that sTeX too knows where to find its archives, we need to set a global system variable `MATHHUB`, that points to your local `MathHub`-directory (see section 3.2).

EdN:1

- **The Mmt System** available here[1]. We recommend following the setup routine documented here.

  Following the setup routine (Step 3) will entail designating a `MathHub`-directory on your local file system, where the Mmt system will look for sTeX/Mmt content archives.

- **sTeX Archives** If we only care about LaTeX and generating `pdf`s, we do not technically need Mmt at all; however, we still need the `MATHHUB` system variable to be set. Furthermore, Mmt can make downloading content archives we might want to use significantly easier, since it makes sure that all dependencies of (often highly interrelated) sTeX archives are cloned as well.

  Once set up, we can run `mmt` in a shell and download an archive along with all of its dependencies like this: `lmh install <name-of-repository>`, or a whole *group* of archives; for example, `lmh install smglom` will download all smglom archives.

- **RusTeX** The Mmt system will also set up RusTeX for you, which is used to generate (semantically annotated) `xhtml` from tex sources. In lieu of using Mmt, you can also download and use RusTeX directly here.

[1] EdNote: For now, we require the `sTeX`-branch, requiring manually compiling the MMT sources

2.2 A First sTeX Document

Having set everything up, we can write a first sTeX document. As an example, we will use the `smglom/calculus` and `smglom/arithmetics` archives, which should be present in the designated `MathHub`-folder, and write a small fragment defining the *geometric series*:

TODO: use some sTeX-archive instead of smglom, use a convergence-notion that includes the limit, mark-up the theorem properly

```
\documentclass{article}
\usepackage{stex,xcolor,stexthm}

\begin{document}
\begin{smodule}{GeometricSeries}
    \importmodule[smglom/calculus]{series}
    \importmodule[smglom/arithmetics]{realarith}

    \symdef{geometricSeries}[name=geometric-series]{\comp{S}}

    \begin{sdefinition}[for=geometricSeries]
        The \definame{geometricSeries} is the \symname{?series}
        \[\defeq{\geometricSeries}{\definiens{
            \infinitesum{\svar{n}}{1}{
                \realdivide[frac]{1}{
                    \realpower{2}{\svar{n}}
            }}
        }}.\]
    \end{sdefinition}

    \begin{sassertion}[name=geometricSeriesConverges,type=theorem]
    The \symname{geometricSeries} \symname{converges} towards $1$.
    \end{sassertion}
\end{smodule}
\end{document}
```

Compiling this document with `pdflatex` should yield the output

Definition 0.1. The **geometric series** is the series

$$S := \sum_{n=1}^{\infty} \frac{1}{2^n}.$$

Theorem 0.2. The geometric series converges towards 1.

Feel free to move your cursor over the various highlighted parts of the document – depending on your pdf viewer, this should yield some interesting (but possibly for now cryptic) information.

Remark 2.2.1:

Note that all of the highlighting, tooltips, coloring and the environment headers come from stexthm – by default, the amount of additional packages loaded is kept to a minimum and all the presentations can be customized.

Let's investigate this document in detail now:

```
\begin{smodule}{GeometricSeries}
...
\end{smodule}
```

smodule

First, we open a new *module* called `GeometricSeries`. This module is assigned a *globally unique* identifier (URI), which (depending on your pdf viewer) should pop up in a tooltip if you hover over the word **geometric series**.

```
\importmodule[smglom/calculus]{series}
\importmodule[smglom/arithmetics]{realarith}
```

\importmodule

Next, we *import* two modules – `series` in the `smglom/calculus`-archive, and `realarith` in the `smglom/arithmetics`-archive. If we investigate these archives, we find the files `series.en.tex` and `realarith.en.tex` (respectively) in their respective `source`-folders, which contain the statements `\begin{smodule}{series}` and `\begin{smodule}{realarith}` (respectively).

The `\importmodule`-statements make all STEX symbols and associated semantic macros (e.g. `\infinitesum`, `\realdivide`, `\realpower`) in the desired module available. Additionally, they "export" these symbols to all further modules which include the *current* module – i.e. if in some future module we would put `\importmodule{GeometricSeries}`, we would also have `\infinitesum` etc. at our disposal.

\usemodule

If we only want to *use* the content of some module `Foo`, e.g. in remarks or examples, but none of the symbols in our current module actually *depend* on the content of `Foo`, we can use `\usemodule` instead – like `\importmodule`, this will make the module content available, but will *not* export it to other modules.

```
\symdef{GeometricSeries}[name=geometric-series]{\comp{S}}
```

\symdef

Next, we introduce a new *symbol* with name `geometric-series` and assign it the semantic macro `\geometricSeries`. `\symdef` also immediately assigns this symbol a *notation*, namely S.

\comp

The macro `\comp` marks the S in the notation as a *notational component*, as opposed to e.g. arguments to `\geometricSeries`. It is the notational components that get highlighted and associated with the corresponding symbol (i.e. in this case `geometricSeries`). Since `\geometricSeries` takes no arguments, we can wrap the whole notation in a `\comp`.

```
\begin{sdefinition}[for=geometricSeries]
...
\end{sdefinition}
\begin{sassertion}[name=geometricSeriesConverges,type=theorem]
...
\end{sassertion}
```

What follows are two sTeX-*statements* (e.g. definitions, theorems, examples, proofs, ...). These are semantically marked-up variants of the usual environments, which take additional optional arguments (e.g. `for=`, `type=`, `name=`). Since many LaTeX templates predefine environments like `definition` or `theorem` with different syntax, we use `sdefinition`, `sassertion`, `sexample` etc. instead. You can customize these environments to e.g. simply wrap around some predefined `theorem`-environment. That way, we can still use `sassertion` to provide semantic information, while being fully compatible with (and using the document presentation of) predefined environments.

In our case, the stexthm-package patches e.g. `\begin{sassertion}[type=theorem]` to use a `theorem`-environment defined (as usual) using amsthm.

```
The \definame{geometricSeries} is the \symname{?series}
```

`\symname` The `\symname`-command prints the name of a symbol, highlights it (based on customizable settings) and associates the text printed with the corresponding symbol. If you hover over the word series in the pdf output, you should see a tooltip showing the full URI of the symbol used.

`\symref` The `\symname`-command is a special case of the more general `\symref`-command, which allows customizing the precise text associated with a symbol.

`\definame` `\definiendum` The `sdefinition`-environment provides two additional macros, `\definame` and `\definiendum` which behave similar to `\symname` and `\symref`, but explicitly mark the symbols as *being defined* in this environment, to allow for special highlighting.

```
\[\defeq{\geometricSeries}{\definiens{
    \infinitesum{\svar{n}}{1}{
        \realdivide[frac]{1}{
            \realpower{2}{\svar{n}}
    }}
}}.\]
```

The next snippet – set in a math environment – uses several semantic macros imported from (or recursively via) `series` and `realarithmetics`, such as `\defeq`, `\infinitesum`, etc. In math mode, using a semantic macro inserts its (default) definition. A semantic macro can have several notations – in that case, we can explicitly choose a specific notation by providing its identifier as an optional argument; e.g. `\realdivide[frac]{a}{b}` will use the explicit notation named `frac` of the semantic macro `\realdivide`, which yields $\frac{a}{b}$ instead of a/b.

`\svar` The `\svar{n}` command marks up the `n` as a variable with name `n` and notation `n`.

`\definiens` The `sdefinition`-environment additionally provides the `\definiens`-command, which allows for explicitly marking up its argument as the *definiens* of the symbol currently being defined.

2.2.1 OMDoc/xhtml Conversion

So, if we run `pdflatex` on our document, then sTeX yields pretty colors and tooltips[1]. But sTeX becomes a lot more powerful if we additionally convert our document to `xhtml`.

TODO VSCode Plugin

Using RUSTEX, we can convert the document to `xhtml` using the command `rustex -i /path/to/file.tex -o /path/to/outfile.xhtml`. Investigating the resulting file, we notice additionaly semantic information resulting from our usage of semantic macros, `\symref` etc. Below is the (abbreviated) snippet inside our `\definiens` block:

```
<mrow resource="" property="stex:definiens">
 <mrow resource="...?series?infinitesum##" property="stex:OMBIND">
  <munderover displaystyle="true">
   <mo resource="...?series?infinitesum" property="stex:comp">Σ</mo>
   <mrow>
    <mi resource="1" property="stex:arg">n</mi>
    <mo class="rel">=</mo>
    <mi resource="2" property="stex:arg">1</mi>
   </mrow>
   <mi resource="...?series?infinitesum" property="stex:comp">∞</mi>
  </munderover>
   <mrow resource="3" property="stex:arg">
    <mfrac resource="...?realarith?division#frac#" property="stex:OMA">
     <mi resource="1" property="stex:arg">1</mi>
     <mrow resource="2" property="stex:arg">
     <mo class="opening">(</mo>
     <msup resource="...realarith?exponentiation##" property="stex:OMA">
      <mi resource="1" property="stex:arg">2</mi>
      <mi resource="2" property="stex:arg">n</mi>
     </msup>
     <mo class="closing">)</mo>
    </mrow>
   </mfrac>
  </mrow>
 </mrow>
</mrow>
```

...containing all the semantic information. The Mmt system can extract from this the following OpenMath snippet:

```
<OMBIND>
  <OMID name="...?series?infinitesum"/>
  <OMV name="n"/>
  <OMLIT name="1"/>
  <OMA>
    <OMS name="...?realarith?division"/>
    <OMLIT name="1"/>
    <OMA>
      <OMS name="...realarith?exponentiation"/>
      <OMLIT name="2"/>
      <OMV name="n"/>
    </OMA>
  </OMA>
</OMBIND>
```

[1] ...and hyperlinks for symbols, and indices, and allows reusing document fragments modularly, and...

...giving us the full semantics of the snippet, allowing for a plurality of knowledge management services – in particular when serving the `xhtml`.

Remark 2.2.2:

Note that the `html` when opened in a browser will look slightly different than the `pdf` when it comes to highlighting semantic content – that is because naturally `html` allows for much more powerful features than `pdf` does. Consequently, the `html` is intended to be served by a system like MMT, which can pick up on the semantic information and offer much more powerful highlighting, linking and similar features, and being customizable by *readers* rather than being prescribed by an author.

Additionally, not all browsers (most notably Chrome) support MathML natively, and might require additional external JavaScript libraries such as MathJax to render mathematical formulas properly.

Chapter 3

Creating sTeX Content

We can use sTeX by simply including the package with `\usepackage{stex}`, or – primarily for individual fragments to be included in other documents – by using the sTeX document class with `\documentclass{stex}` which combines the standalone document class with the stex package.

Both the stex package and document class offer the following options:

`lang` (⟨*language*⟩∗) Languages to load with the babel package.

`mathhub` (⟨*directory*⟩) MathHub folder to search for repositories – this is not necessary if the `MATHHUB` system variable is set.

`sms` (⟨*boolean*⟩) use *persisted* mode (not yet implemented).

`image` (⟨*boolean*⟩) passed on to tikzinput.

`debug` (⟨*log-prefix*⟩∗) Logs debugging information with the given prefixes to the terminal, or all if `all` is given. Largely irrelevant for the majority of users.

3.1 How Knowledge is Organized in sTeX

sTeX content is organized on multiple levels:

- sTeX **archives** (see section 3.2) contain individual `.tex`-files.
- These may contain sTeX **modules**, introduced via `\begin{smodule}{ModuleName}`.
- Modules contain sTeX **symbol declarations**, introduced via `\symdecl{symbolname}`, `\symdef{symbolname}` and some other constructions. Most symbols have a *notation* that can be used via a *semantic macro* `\symbolname` generated by symbol declarations.
- sTeX **expressions** finally are built up from usages of semantic macros.

- sTeX archives are simultaneously Mmt archives, and the same directory structure is consequently used.
- sTeX modules correspond to OMDoc/Mmt *theories*. `\importmodule`s (and

similar constructions) induce MMT `includes` and other *theory morphisms*, thus giving rise to a *theory graph* in the OMDoc sense.

- Symbol declarations induce OMDoc/MMT *constants*, with optional (formal) *type* and *definiens* components.
- Finally, STEX expressions are converted to OMDoc/MMT terms, which use the syntax of OpenMath.

3.2 STEX Archives

3.2.1 The Local MathHub-Directory

`\usemodule`, `\importmodule`, `\inputref` etc. allow for including content modularly without having to specify absolute paths, which would differ between users and machines. Instead, STEX uses *archives* that determine the global namespaces for symbols and statements and make it possible for STEX to find content referenced via such URIs.

All STEX archives need to exist in the local `MathHub`-directory. STEX knows where this folder is via one of three means:

1. If the STEX package is loaded with the option `mathhub=/path/to/mathhub`, then STEX will consider `/path/to/mathhub` as the local `MathHub`-directory.
2. If the `mathhub` package option is *not* set, but the macro `\mathhub` exists when the STEX-package is loaded, then this macro is assumed to point to the local `MathHub`-directory; i.e. `\def\mathhub{/path/to/mathhub}\usepackage{stex}` will set the `MathHub`-directory as `path/to/mathhub`.
3. Otherwise, STEX will attempt to retrieve the system variable `MATHHUB`, assuming it will point to the local `MathHub`-directory. Since this variant needs setting up only *once* and is machine-specific (rather than defined in tex code), it is compatible with collaborating and sharing tex content, and hence recommended.

3.2.2 The Structure of STEX Archives

An STEX archive `group/name` needs to be stored in the directory `/path/to/mathhub/group/name`; e.g. assuming your local `MathHub`-directory is set as `/user/foo/MathHub`, then in order for the `smglom/calculus`-archive to be found by the STEX system, it needs to be in `/user/foo/MathHub/smglom/calculus`.

Each such archive needs two subdirectories:

- `/source` – this is where all your tex files go.
- `/META-INF` – a directory containing a single file `MANIFEST.MF`, the content of which we will consider shortly

An additional `lib`-directory is optional, and is where STEX will look for files included via `\libinput`.

Additionally a *group* of archives `group/name` may have an additional archive `group/meta-inf`. If this `meta-inf`-archive has a `/lib`-subdirectory, it too will be searched by `\libinput` from all tex files in any archive in the `group/*`-group.

We recommend this additional directory structure in the `source`-folder of an sTeX archive:

- `/source/mod/` – individual sTeX modules, containing symbol declarations, notations, and `\begin{sparagraph}[type=symdoc,for=...]` environments for "encyclopedic" symbol documentations
- `/source/def/` – definitions
- `/source/ex/` – examples
- `/source/thm/` – theorems, lemmata and proofs; preferably proofs in separate files to allow for multiple proofs for the same statement
- `/source/snip/` – individual text snippets such as remarks, explanations etc.
- `/source/frag/` – individual document fragments, ideally only `\inputref`ing snippets, definitions, examples etc. in some desirable order
- `/source/tikz/` – tikz images, as individual `.tex`-files
- `/source/pic/` – image files.

3.2.3 MANIFEST.MF-Files

The `MANIFEST.MF` in the `META-INF`-directory consists of key-value-pairs, instructing sTeX (and associated software) of various properties of an archive. For example, the `MANIFEST.MF` of the `smglom/calculus`-archive looks like this:

```
id: smglom/calculus
source-base: http://mathhub.info/smglom/calculus
narration-base: http://mathhub.info/smglom/calculus
dependencies: smglom/arithmetics,smglom/sets,smglom/topology,
            smglom/mv,smglom/linear-algebra,smglom/algebra
responsible: Michael.Kohlhase@FAU.de
title: Elementary Calculus
teaser: Terminology for the mathematical study of change.
description: desc.html
```

Many of these are in fact ignored by sTeX, but some are important:

`id`: The name of the archive, including its group (e.g. `smglom/calculus`),

`source-base` or

`ns`: The namespace from which all symbol and module URIs in this repository are formed, see (TODO),

`narration-base`: The namespace from which all document URIs in this repository are formed, see (TODO),

`url-base`: The URL that is formed as a basis for *external references*, see (TODO),

`dependencies`: All archives that this archive depends on. sTeX ignores this field, but MMT can pick up on them to resolve dependencies, e.g. for `lmh install`.

3.2.4 Using Files in sTeX Archives Directly

Several macros provided by sTeX allow for directly including files in repositories. These are:

`\mhinput` `\mhinput[Some/Archive]{some/file}` directly inputs the file `some/file` in the `source`-folder of `Some/Archive`.

`\inputref` `\inputref[Some/Archive]{some/file}` behaves like `\mhinput`, but wraps the input in a `\begingroup ... \endgroup`. When converting to `xhtml`, the file is not input at all, and instead an `html`-annotation is inserted that references the file.

In the majority of cases `\inputref` is likely to be preferred over `\mhinput`.

`\ifinput` Both `\mhinput` and `\inputref` set `\ifinput` to "true" during input. This allows for selectively including e.g. bibliographies only if the current file is not being currently included in a larger document.

`\addmhbibresource` `\addmhbibresource[Some/Archive]{some/file}` searches for a file like `\mhinput` does, but calls `\addbibresource` to the result and looks for the file in the archive root directory directly, rather than the `source` directory.

`\libinput` `\libinput{some/file}` searches for a file `some/file` in

- the `lib`-directory of the current archive, and
- the `lib`-directory of a `meta-inf`-archive in (any of) the archive groups containing the current archive

and include all found files in reverse order; e.g. `\libinput{preamble}` in a `.tex`-file in `smglom/calculus` will *first* input `.../smglom/meta-inf/lib/preamble.tex` and then `../smglom/calculus/lib/preamble.tex`.

Will throw an error if *no* candidate for `some/file` is found.

`\libusepackage` `\libusepackage[package-options]{some/file}` searches for a file `some/file.sty` in the same way that `\libinput` does, but will call `\usepackage[package-options]{path/to/some/file}` instead of `\input`.

Will throw an error if not *exactly one* candidate for `some/file` is found.

Remark 3.2.1:

A good practice is to have individual sTEX fragments follow basically this document frame:

```
\documentclass{stex}
\libinput{preamble}
\begin{document}
    ...
    \ifinputref \else \libinput{postamble} \fi
\end{document}
```

Then the `preamble.tex` files can take care of loading the generally required packages, setting presentation customizations etc. (per archive or archive group or both), and `postamble.tex` can e.g. print the bibliography, index etc.

3.3 Module, Symbol and Notation Declarations

3.3.1 The `smodule`-Environment

`smodule` A new module is declared using the basic syntax

`\begin{smodule}[options]{ModuleName}...\end{smodule}`.

A module is required to declare any new formal content such as symbols or notations (but not variables, which may be introduced anywhere).

The `smodule`-environment takes several optional arguments, all of which are optional:

`title` (⟨*token list*⟩) to display in customizations.

`type` (⟨*string*⟩∗) for use in customizations.

`deprecate` (⟨*module*⟩) if set, will throw a warning when loaded, urging to use ⟨*module*⟩ instead.

`id` (⟨*string*⟩) for cross-referencing.

`ns` (⟨*URI*⟩) the namespace to use. *Should not be used, unless you know precisely what you're doing.* If not explicitly set, is computed using `\stex_modules_current_namespace:`.

`lang` (⟨*language*⟩) if not set, computed from the current file name (e.g. `foo.en.tex`).

`sig` (⟨*language*⟩) if the current file is a translation of a file with the same base name but a different language suffix, setting `sig=<lang>` will preload the module from that language file. This helps ensuring that the (formal) content of both modules is (almost) identical across languages and avoids duplication.

`creators` (⟨*string*⟩∗) names of the creators.

`contributors` (⟨*string*⟩∗) names of contributors.

`srccite` (⟨*string*⟩) a source citation for the content of this module.

An sTeX module corresponds to an MMT/OMDoc *theory*. As such it gets assigned a module URI (*universal resource identifier*) of the form `<namespace>?<module-name>`.

By default, opening a module will produce no output whatsoever, e.g.:

Example 1

Input:

```
\begin{smodule}[title={This is Some Module}]{SomeModule}
   Hello World
\end{smodule}
```

Output:

Hello World

.

`\stexpatchmodule`

We can customize this behavior either for all modules or only for modules with a specific `type` using the command `\stexpatchmodule[optional-type]{begin-code}{end-code}`. Some optional parameters are then available in `\smodule*`-macros, specifically `\smoduletitle`, `\smoduletype` and `\smoduleid`. For example:

Example 2

Input:

```
\stexpatchmodule[display]
 {\textbf{Module (\smoduletitle)}\par}
 {\par\noindent\textbf{End of Module (\smoduletitle)}}

\begin{smodule}[type=display,title={Some New Module}]{SomeModule2}
   Hello World
\end{smodule}
```

Output:

Module (Some New Module)
Hello World
End of Module (Some New Module)

.

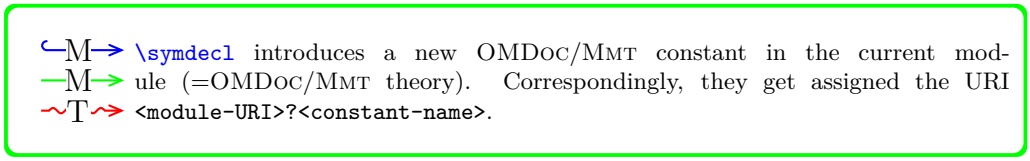
3.3.2 Declaring New Symbols and Notations

Inside an `smodule` environment, we can declare new sTeX symbols.

\symdecl

The most basic command for doing so is using \symdecl{symbolname}. This introduces a new symbol with name symbolname, arity 0 and semantic macro \symbolname.

The starred variant \symdecl*{symbolname} will declare a symbol, but not introduce a semantic macro. If we don't want to supply a notation (for example to introduce concepts like "abelian", which is not something that has a notation), the starred variant is likely to be what we want.



Without a semantic macro or a notation, the only meaningful way to reference a symbol is via \symref,\symname etc.

Example 3

Input:

```
\symdecl*{foo}
Given a \symname{foo}, we can...
```

Output:

Given a foo, we can...

.

Obviously, most semantic macros should take actual *arguments*, implying that the symbol we introduce is an *operator* or *function*. We can let \symdecl know the *arity* (i.e. number of arguments) of a symbol like this:

Example 4

Input:

```
\symdecl{binarysymbol}[args=2]
\symref{binarysymbol}{this} is a symbol taking two arguments.
```

Output:

this is a symbol taking two arguments.

.

\notation

In that case, we probably want to supply a notation as well, in which case we can finally actually use the semantic macro in math mode. We can do so using the \notation command, like this:

Example 5

Input:

```
1 \notation{binarysymbol}{\text{First: }#1\text{; Second: }#2}
2 $\binarysymbol{a}{b}$
```

Output:

First: a; Second: b

.

↪M→ Applications of semantic macros, such as \binarysymbol{a}{b} are translated to
—M→ MMT/OMDoc as OMA-terms with head <OMS name="...?binarysymbol"/>.
⇝T↝ Semantic macros with no arguments correspond to OMS directly.

\comp

Unfortunately, we have no highlighting whatsoever now. That is because we need to tell STEX explicitly which parts of the notation are *notation components* which *should* be highlighted. We can do so with the \comp command.

We can introduce a new notation highlight for \binarysymbol that fixes this flaw, which we can subsequently use with \binarysymbol[highlight]:

Example 6

Input:

```
1 \notation{binarysymbol}[highlight]
2    {\comp{\text{First: }}#1\comp{\text{; Second: }}#2}
3 $\binarysymbol[highlight]{a}{b}$
```

Output:

First: a; Second: b

.

Ideally, \comp would not be necessary: Everything in a notation that is *not* an argument should be a notation component. Unfortunately, it is computationally expensive to determine where an argument begins and ends, and the argument markers #n may themselves be nested in other macro applications or TEX groups, making it ultimately almost impossible to determine them automatically while also remaining compatible with arbitrary highlighting customizations (such as tooltips, hyperlinks, colors) that users might employ, and that are ultimately invoked by \comp.

Note that it is required that

1. the argument markers `#n` never occur inside a `\comp`, and
2. no semantic arguments may ever occur inside a notation.

Both criteria are not just required for technical reasons, but conceptionally meaningful:
The underlying principle is that the arguments to a semantic macro represent *arguments to the mathematical operation* represented by a symbol. For example, a semantic macro `\addition{a}{b}` taking two arguments would represent *the actual addition of (mathematical objects) a and b*. It should therefore be impossible for a or b to be part of a notation component of `\addition`.
Similarly, a semantic macro can not conceptually be part of the notation of `\addition`, since a semantic macro represents a *distinct mathematical concept* with *its own semantics*, whereas notations are syntactic representations of the very symbol to which the notation belongs.
If you want an argument to a semantic macro to be a purely syntactic parameter, then you are likely somewhat confused with respect to the distinction between the precise *syntax* and *semantics* of the symbol you are trying to declare (which happens quite often even to experienced sTEX users), and might want to give those another thought - quite likely, the macro you aim to implement does not actually represent a semantically maningful mathematical concept, and you will want to use `\def` and similar native LaTeX macro definitions rather than semantic macros.

`\symdef`

In the vast majority of cases where a symbol declaration should come with a semantic macro, we will want to supply a notation immediately. For that reason, the `\symdef` command combines the functionality of both `\symdecl` and `\notation` with the optional arguments of both:

Example 7

Input:

```
\symdef{newbinarysymbol}[hl,args=2]
   {\comp{\text{1.: }}#1\comp{\text{; 2.: }}#2}
$\newbinarysymbol{a}{b}$
```

Output:

1.: a; 2.: b

.

We just declared a new symbol `newbinarysymbol` with `args=2` and immediately provided it with a notation with identifier `hl`. Since `hl` is the *first* (and so far, only) notation supplied for `newbinarysymbol`, using `\newbinarysymbol` without optional argument defaults to this notation.

\setnotation

The first notation provided will stay the default notation unless explicitly changed – this is enabled by the \setnotation command: \setnotation{symbolname}{notation-id} sets the default notation of \symbolname to notation-id, i.e. henceforth, \symbolname behaves like \symbolname[notation-id] from now on.

Often, a default notation is set right after the corresponding notation is introduced – the starred version \notation* for that reason introduces a new notation and immediately sets it to be the new default notation. So expressed differently, the *first* \notation for a symbol behaves exactly like \notation*, and \notation*{foo}[bar]{...} behaves exactly like \notation{foo}[bar]{...}\setnotation{foo}{bar}.

Operator Notations

Once we have a semantic macro with arguments, such as \newbinarysymbol, the semantic macro represents the *application* of the symbol to a list of arguments. What if we want to refer to the operator *itself*, though?

We can do so by supplying the \notation (or \symdef) with an *operator notation*, indicated with the optional argument op=. We can then invoke the operator notation using \symbolname![notation-identifier]. Since operator notations never take arguments, we do not need to use \comp in it, the whole notation is wrapped in a \comp automatically:

Example 8

Input:

```
\notation{newbinarysymbol}[ab,
 op={\text{a:}\cdot\text{; b:}\cdot}]
 {\comp{\text{a:}}#1\comp{\text{; b:}}#2}
\symname{newbinarysymbol} is also occasionally written
$\newbinarysymbol![ab]$
```

Output:

newbinarysymbol is also occasionally written a: · ; b:·

.

\symbolname! is translated to OMDoc/Mmt as <OMS name="...?symbolname"/> directly.

3.3.3 Argument Types

The notations so far used *simple* arguments which we call i-*type* arguments. Declaring a new symbol with \symdecl{foo}[args=3] is equivalent to writing \symdecl{foo}[args=iii], indicating that the semantic macro takes three i-type arguments. However, there are three more argument types which we will investigate now, namely b-type, a-type and B-type arguments.

b-Type Arguments

A b-type argument represents a *variable* that is *bound* by the symbol in its application, making the symbol a *binding operator*. Typical examples of binding operators are e.g. sums $\sum$, products $\prod$, integrals $\int$, quantifiers like $\forall$ and $\exists$, that λ-operator, etc.

b-type arguments behave exactly like i-type arguments within TEX, but applications of binding operators, i.e. symbols with b-type arguments, are translated to OMBIND-terms in OMDoc/Mmt, rather than OMA.

Fo example, we can implement a summation operator binding an index variable and taking lower and upper index bounds and the expression to sum over like this:

Example 9

Input:

```
1 \symdef{summation}[args=biii]
2 {\mathop{\comp{\sum}}_{#1\comp{=}#2}^{#3}#4}
3 $\summation{\svar{x}}{1}{\svar{n}}{\svar{x}}^2$
```

Output:

$$\sum_{x=1}^{n} x^2$$

where the variable x is now *bound* by the \summation-symbol in the expression.

a-Type Arguments

a-type arguments represent a *flexary argument sequence*, i.e. a sequence of arguments of arbitrary length. Formally, operators that take arbitrarily many arguments don't "exist", but in informal mathematics, they are ubiquitous. a-type arguments allow us to write e.g. \addition{a,b,c,d,e} rather than having to write something like \addition{a}{\addition{b}{\addition{c}{\addition{d}{e}}}}!

\notation (and consequently \symdef, too) take one additional argument for each a-type argument that indicates how to "accumulate" a comma-separated sequence of arguments. This is best demonstrated on an example.

Let's say we want an operator representing quantification over an ascending chain of elements in some set, i.e. \ascendingchain{S}{a,b,c,d,e}{t} should yield $\forall a <_S b <_S c <_S d <_S e.\, t$. The "base"-notation for this operator is simply {\comp{\forall} #2\comp{.\,}#3}, where #2 represents the full notation fragment *accumulated* from {a,b,c,d,e}.

The *additional* argument to \notation (or \symdef) takes the same arguments as the base notation and two *additional* arguments ##1 and ##2 representing successive pairs in the a-type argument, and accumulates them into #2, i.e. to produce $a <_S b <_S c <_S d <_S e$, we do {##1 \comp{<}_{#1} ##2}:

Example 10

Input:

```
\symdef{ascendingchain}[args=iai]
 {\comp{\forall} #2\comp{.\,}#3}
 {##1 \comp{<}_{#1} ##2}

Tadaa: $\ascendingchain{S}{a,b,c,d,e}{t}$
```

Output:

Tadaa: $\forall a <_S b <_S c <_S d <_S e.\, t$

.

If this seems overkill, keep in mind that you will rarely need the single-hash arguments `#1`,`#2` etc. in the `a`-notation-argument. For a much more representative and simpler example, we can introduce flexary addition via:

Example 11

Input:

```
\symdef{addition}[args=a]{#1}{##1 \comp{+} ##2}

Tadaa: $\addition{a,b,c,d,e}$
```

Output:

Tadaa: $a+b+c+d+e$

.

The `assoc`-key We mentioned earlier that "formally", flexary arguments don't really "exist". Indeed, formally, addition is usually defined as a binary operation, quantifiers bind a single variable etc.

Consequently, we can tell STEX (or, rather, MMT/OMDOC) how to "resolve" flexary arguments by providing `\symdecl` or `\symdef` with an optional `assoc`-argument, as in `\symdecl{addition}[args=a,assoc=bin]`. The possible values for the `assoc`-key are:

`bin`: A binary, assoiative argument, e.g. as in `\addition`

`binl`: A binary, left-associative argument, e.g. $a^{b^{c^d}}$, which stands for $((a^b)^c)^d$

`binr`: A binary, right-associative argument, e.g. as in $A \to B \to C \to D$, which stands for $A \to (B \to (C \to D))$

`pre`: Successively prefixed, e.g. as in $\forall x, y, z.\, P$, which stands for $\forall x.\, \forall y.\, \forall z.\, P$

`conj`: Conjunctive, e.g. as in $a = b = c = d$ or $a, b, c, d \in A$, which stand for $a = d \wedge b = d \wedge c = d$ and $a \in A \wedge b \in A \wedge c \in A \wedge d \in A$, respectively

`pwconj`: Pairwise conjunctive, e.g. as in $a \neq b \neq c \neq d$, which stands for $a \neq b \wedge a \neq c \wedge a \neq d \wedge b \neq c \wedge b \neq d \wedge c \neq d$

B-Type Arguments

Finally, B-type arguments simply combine the functionality of both `a` and `b` - i.e. they represent an arbitrarily long sequence of variables to be bound, e.g. for implementing quantifiers:

Example 12

Input:

```
\symdef{quantforall}[args=Bi]
 {\comp{\forall}#1\comp{.}#2}
 {##1\comp,##2}

$\quantforall{\svar{x},\svar{y},\svar{z}}{P}$
```

Output:

$\forall x,y,z.P$

.

3.3.4 Type and Definiens Components

`\symdecl` and `\symdef` take two more optional arguments. TEX largely ignores them (except for special situations we will talk about later), but MMT can pick up on them for additional services. These are the `type` and `def` keys, which expect expressions in math-mode (ideally using semantic macros, of course!)

The `type` and `def` keys correspond to the `type` and `definiens` components of OMDoc/MMT constants.
Correspondingly, the name "type" should be taken with a grain of salt, since OMDoc/MMT– being foundation-independent – does not a priori implement a fixed typing system.

The `type`-key allows us to provide additional information (given the necessary STEX symbols), e.g. for addition on natural numbers:

Example 13

Input:

```
 \symdef{Nat}[type=\set]{\comp{\mathbb N}}
\symdef{addition}[
    type=\funtype{\Nat,\Nat}{\Nat},
    op=+,
    args=a
]{#1}{##1 \comp+ ##2}

\symname{addition} is an operation $\funtype{\Nat,\Nat}{\Nat}$
```

Output:

addition is an operation $\mathbb{N}\times\mathbb{N}\to\mathbb{N}$

.

The `def`-key allows for declaring symbols as abbreviations:

Example 14

Input:

```
\symdef{successor}[
   type=\funtype{\Nat}{\Nat},
   def=\fun{\svar{x}}{\addition{\svar{x},1}},
   op=\mathtt{succ},
   args=1
]{\comp{\mathtt{succ(}#1\comp{)}}}

The \symname{successor} operation $\funtype{\Nat}{\Nat}$
is defined as $\fun{\svar{x}}{\addition{\svar{x},1}}$
```

Output:

The successor operation $\mathbb{N}\to\mathbb{N}$ is defined as $x\mapsto x+1$

.

3.3.5 Precedences and Automated Bracketing

Having done `\addition`, the obvious next thing to implement is `\multiplication`. This is in theory straight-forward:

Example 15

Input:

```
\symdef{multiplication}[
   type=\funtype{\Nat,\Nat}{\Nat},
   op=\cdot,
   args=a
]{#1}{##1 \comp\cdot ##2}

\symname{multiplication} is an operation $\funtype{\Nat,\Nat}{\Nat}$
```

Output:

multiplication is an operation $\mathbb{N}\times\mathbb{N}\to\mathbb{N}$

.

However, if we *combine* `\addition` and `\multiplication`, we notice a problem:

Example 16

Input:

```
$\addition{a,\multiplication{b,\addition{c,\multiplication{d,e}}}}$
```

Output:

$a+b\cdot c+d\cdot e$

We all know that $\cdot$ binds stronger than $+$, so the output $a+b\cdot c+d\cdot e$ does not actually reflect the term we wrote. We can of course insert parentheses manually

Example 17

Input:

```
$\addition{a,\multiplication{b,(\addition{c,\multiplication{d,e}})}}$
```

Output:

$a+b\cdot(c+d\cdot e)$

but we can also do better by supplying *precedences* and have STEX insert parentheses automatically.

For that purpose, `\notation` (and hence `\symdef`) take an optional argument `prec=<opprec>;<argprec1>x...x<argprec n>`.

We will investigate the precise meaning of `<opprec>` and the `<argprec>`s shortly – in the vast majority of cases, it is prefectly sufficient to think of `prec=` taking a single number and having that be *the* precedence of the notation, where lower precedences (somewhat counterintuitively) bind stronger than higher precedences. So fixing our notations for `\addition` and `\multiplication`, we get:

Example 18

Input:

```
\notation{multiplication}[
   op=\cdot,
   prec=50
]{#1}{##1 \comp\cdot ##2}
\notation{addition}[
   op=+,
   prec=100
]{#1}{##1 \comp+ ##2}

$\addition{a,\multiplication{b,\addition{c,\multiplication{d,e}}}}$
```

Output:

$a+b\cdot(c+d\cdot e)$

Note that the precise numbers used for precedences are pretty arbitrary - what matters is which precedences are higher than which other precedences when used in conjunction.

`\infprec`
`\neginfprec`

It is occasionally useful to have "infinitely" high or low precedences to enforce or forbid automated bracketing entirely – for those purposes, `\infprec` and `\neginfprec` exist (which are implemented as the maximal and minimal integer values accordingly).

More precisely, each notation takes

1. One *operator precedence* and

2. one *argument precedence* for each argument.

By default, all precedences are 0, unless the symbol takes no argument, in which case the operator precedence is \neginfprec (negative infinity). If we only provide a single number, this is taken as both the operator precedence and all argument precedences.
STEX decides whether to insert parentheses by comparing operator precedences to a *downward precedence* p_d with initial value \infprec. When encountering a semantic macro, STEX takes the operator precedence p_{op} of the notation used and checks whether $p_{op} > p_d$. If so, STEX insert parentheses.
When STEX steps into an argument of a semantic macro, it sets p_d to the respective argument precedence of the notation used.
In the example above:

1. STEX starts out with $p_d =$\infprec.
2. STEX encounters \addition with $p_{op} = 100$. Since $100 \not> $\infprec, it inserts no parentheses.
3. Next, STEX encounters the two arguments for \addition. Both have no specifically provided argument precedence, so STEX uses $p_d = p_{op} = 100$ for both and recurses.
4. Next, STEX encounters \multiplication{b,...}, whose notation has $p_{op} = 50$.
5. We compare to the current downward precedence p_d set by \addition, arriving at $p_{op} = 50 \not> 100 = p_d$, so STEX again inserts no parentheses.
6. Since the notation of \multiplication has no explicitly set argument precedences, STEX uses the operator precedence for all arguments of \multiplication, hence sets $p_d = p_{op} = 50$ and recurses.
7. Next, STEX encounters the inner \addition{c,...} whose notation has $p_{op} = 100$.
8. We compare to the current downward precedence p_d set by \multiplication, arriving at $p_{op} = 100 > 50 = p_d$ – which finally prompts STEX to insert parentheses, and we proceed as before.

3.3.6 Variables

All symbol and notation declarations require a module with which they are associated, hence the commands \symdecl, \notation, \symdef etc. are disabled outside of smodule-environments.

Variables are different – variables are allowed everywhere, are not exported when the current module (if one exists) is imported (via \importmodule or \usemodule) and (also unlike symbol declarations) "disappear" at the end of the current TEX group.

\svar

So far, we have always used variables using \svar{n}, which marks-up n as a variable with name n. More generally, \svar[foo]{<texcode>} marks-up the arbitrary <texcode> as representing a variable with name foo.

Of course, this makes it difficult to reuse variables, or introduce "functional" variables with arities > 0, or provide them with a type or definiens.

\vardef

For that, we can use the \vardef command. Its syntax is largely the same as that of \symdef, but unlike symbols, variables have only one notation (TODO: so far?), hence there is only \vardef and no \vardecl.

Example 19

Input:

```
\vardef{varf}[
   name=f,
   type=\funtype{\Nat}{\Nat},
   op=f,
   args=1,
   prec=0;\neginfprec
]{\comp{f}#1}
\vardef{varn}[name=n,type=\Nat]{\comp{n}}
\vardef{varx}[name=x,type=\Nat]{\comp{x}}

Given a function $\varf!:\funtype{\Nat}{\Nat}$,
by $\addition{\varf!,\varn}$ we mean the function
$\fun{\varx}{\varf{\addition{\varx,\varn}}}$
```

Output:

Given a function $f : \mathbb{N} \to \mathbb{N}$, by $f + n$ we mean the function $x \mapsto f(x + n)$

˙(of course, "lifting" addition in the way described in the previous example is an operation that deserves its own symbol rather than abusing \addition, but... well.)

TODO: bind=forall/exists

3.3.7 Variable Sequences

Variable *sequences* occur quite frequently in informal mathematics, hence they deserve special support. Variable sequences behave like variables in that they disappear at the end of the current TEX group and are not exported from modules, but their declaration is quite different.

\varseq

A variable sequence is introduced via the command \varseq, which takes the usual optional arguments name and type. It then takes a starting index, an end index and a *notation* for the individual elements of the sequence parametric in an index.

This is best shown by example:

Example 20

Input:

```
\vardef{varn}[name=n,type=\Nat]{\comp{n}}
\varseq{seqa}[name=a,type=\Nat]{1}{\varn}{\comp{a}_{#1}}

The $i$th index of $\seqa!$ is $\seqa{i}$.
```

Output:

The ith index of $a_1, \ldots, a_n$ is a_i.

.

Note that the syntax `\seqa!` now automatically generates a presentation based on the starting and ending index.

TODO: more notations for invoking sequences.

Notably, variable sequences are nicely compatible with `a`-type arguments, so we can do the following:

Example 21

Input:

```
$\addition{\seqa}$
```

Output:

$a_1+\ldots+a_n$

.

Sequences can be *multidimensional* using the `args`-key, in which case the notation's arity increases and starting and ending indices have to be provided as a comma-separated list:

Example 22

Input:

```
\vardef{varm}[name=m,type=\Nat]{\comp{m}}
\varseq{seqa}[
    name=a,
    args=2,
    type=\Nat,
]{1,1}{\varn,\varm}{\comp{a}_{#1}^{#2}}

$\seqa!$ and $\addition{\seqa}$
```

Output:

$a_1^1,\ldots,a_n^m$ and $a_1^1+\ldots+a_n^m$

˙We can also explicitly provide a "middle" segment to be used, like such:

Example 23

Input:

```
\varseq{seqa}[
    name=a,
    type=\Nat,
    args=2,
    mid={\comp{a}_{\varn}^1,\comp{a}_1^2,\ellipses,\comp{a}_{1}^{\varm}}
]{1,1}{\varn,\varm}{\comp{a}_{#1}^{#2}}

$\seqa!$ and $\addition{\seqa}$
```

Output:

$a_1^1,\ldots,a_n^1,a_1^2,\ldots,a_1^m,\ldots,a_n^m$ and $a_1^1+\ldots+a_n^1+a_1^2+\ldots+a_1^m+\ldots+a_n^m$

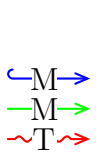
.

3.4 Module Inheritance and Structures

3.4.1 Multilinguality and Translations

If we load the sTeX document class or package with the option `lang=<lang>`, sTeX will load the appropriate babel language for you – e.g. `lang=de` will load the babel language `ngerman`. Additionally, it makes sTeX aware of the current document being set in (in this example) *german*. This matters for reasons other than mere babel-purposes, though:

Every *module* is assigned a language. If no sTeX package option is set that allows for inferring a language, sTeX will check whether the current file name ends in e.g. `.en.tex` (or `.de.tex` or `.fr.tex`, or...) and set the language accordingly. Alternatively, a language can be explicitly assigned via `\begin{smodule}[lang=<language>]{Foo}`.



Technically, each `smodule`-environment induces *two* OMDoc/Mmt theories: `\begin{smodule}[lang=<lang>]{Foo}` generates a theory `some/namespace?Foo` that only contains the "formal" part of the module – i.e. exactly the content that is exported when using `\importmodule`.
Additionally, Mmt generates a *language theory* `some/namespace/Foo?<lang>` that includes `some/namespace?Foo` and contains all the other document content – variable declarations, includes for each `\usemodule`, etc.

Notably, the language suffix in a filename is ignored for `\usemodule`, `\importmodule` and in generating/computing URIs for modules. This however allows for providing *translations* for modules between languages without needing to duplicate content:

If a module `Foo` exists in e.g. english in a file `Foo.en.tex`, we can provide a file `Foo.de.tex` right next to it, and write `\begin{smodule}[sig=en]{Foo}`. The `sig`-key then signifies, that the "signature" of the module is contained in the *english* version of the module, which is immediately imported from there, just like `\importmodule` would.

Additionally to translating the informal content of a module file to different languages, it also allows for customizing notations between languages. For example, the *least common multiple* of two numbers is often denoted as $\texttt{lcm}(a,b)$ in english, but is called *kleinstes gemeinsames Vielfaches* in german and consequently denoted as $\texttt{kgV}(a,b)$ there.

We can therefore imagine a german version of an lcm-module looking something like this:

```
1 \begin{smodule}[sig=en]{lcm}
2   \notation*{lcm}[de]{\comp{\mathtt{kgV}}(#1,#2)}
3
4   Das \symref{lcm}{kleinste gemeinsame Vielfache}
5   $\lcm{a,b}$ von zwei Zahlen $a,b$ ist...
6 \end{smodule}
```

If we now do `\importmodule{lcm}` (or `\usemodule{lcm}`) within a *german* document, it will also load the content of the german translation, including the `de`-notation for `\lcm`.

3.4.2 Simple Inheritance and Namespaces

\importmodule
\usemodule

\importmodule[Some/Archive]{path?ModuleName} is only allowed within an smodule-environment and makes the symbols declared therein available. Additionally the content of ModuleName will be exported if the current module is imported somewhere else via \importmodule.

\usemodule behaves the same way, but without exporting the content of the used module.

It is worth going into some detail how exactly \importmodule and \usemodule resolve their arguments to find the desired module – which is closely related to the *namespace* generated for a module, that is used to generate its URI.

Ideally, sTeX would use arbitrary URIs for modules, with no forced relationships between the *logical* namespace of a module and the *physical* location of the file declaring the module – like Mmt does things.
Unfortunately, TeX only provides very restricted access to the file system, so we are forced to generate namespaces systematically in such a way that they reflect the physical location of the associated files, so that sTeX can resolve them accordingly. Largely, users need not concern themselves with namespaces at all, but for completenesses sake, we describe how they are constructed:

- If \begin{smodule}{Foo} occurs in a file /path/to/file/Foo[.⟨*lang*⟩].tex which does not belong to an archive, the namespace is file://path/to/file.
- If the same statement occurs in a file /path/to/file/bar[.⟨*lang*⟩].tex, the namespace is file://path/to/file/bar.

In other words: outside of archives, the namespace corresponds to the file URI with the filename dropped iff it is equal to the module name, and ignoring the (optional) language suffix.
If the current file is in an archive, the procedure is the same except that the initial segment of the file path up to the archive's source-folder is replaced by the archive's namespace URI.

Conversely, here is how namespaces/URIs and file paths are computed in import statements, examplary \importmodule:

- \importmodule{Foo} outside of an archive refers to module Foo in the current namespace. Consequently, Foo must have been declared earlier in the same document or, if not, in a file Foo[.⟨*lang*⟩].tex in the same directory.
- The same statement *within* an archive refers to either the module Foo declared earlier in the same document, or otherwise to the module Foo in the archive's top-level namespace. In the latter case, is has to be declared in a file Foo[.⟨*lang*⟩].tex directly in the archive's source-folder.
- Similarly, in \importmodule{some/path?Foo} the path some/path refers to either the sub-directory and relative namespace path of the current directory and namespace outside of an archive, or relative to the current archive's top-level namespace and source-folder, respectively.
  The module Foo must either be declared in the

file ⟨*top-directory*⟩`/some/path/Foo[.`⟨*lang*⟩`].tex`, or in ⟨*top-directory*⟩`/some/path[.`⟨*lang*⟩`].tex` (which are checked in that order).

- Similarly, `\importmodule[Some/Archive]{some/path?Foo}` is resolved like the previous cases, but relative to the archive `Some/Archive` in the mathhub-directory.
- Finally, `\importmodule{full://uri?Foo}` naturally refers to the module `Foo` in the namespace `full://uri`. Since the file this module is declared in can not be determined directly from the URI, the module must be in memory already, e.g. by being referenced earlier in the same document.

  Since this is less compatible with a modular development, using full URIs directly is strongly discouraged, unless the module is delared in the current file directly.

3.4.3 The `mathstructure` Environment

A common occurence in mathematics is bundling several interrelated "declarations" together into *structures*. For example:

- A *monoid* is a structure $\langle M, \circ, e\rangle$ with $\circ : M \times M \to M$ and $e \in M$ such that...
- A *topological space* is a structure $\langle X, \mathcal{T}\rangle$ where X is a set and $\mathcal{T}$ is a topology on X
- A *partial order* is a structure $\langle S, \leq \rangle$ where $\leq$ is a binary relation on S such that...

This phenomenon is important and common enough to warrant special support, in particular because it requires being able to *instantiate* such structures (or, ratherer, structure *signatures*) in order to talk about (concrete or variable) *particular* monoids, topological spaces, partial orders etc.

`mathstructure`

The `mathstructure` environment allows us to do exactly that. It behaves exactly like the `smodule` environment, but is itself only allowed inside an `smodule` environment, and allows for instantiation later on.

How this works is again best demonstrated by example:

Example 24

Input:

```
\begin{mathstructure}{monoid}
   \symdef{universe}[type=\set]{\comp{U}}
   \symdef{op}[
      args=2,
      type=\funtype{\universe,\universe}{\universe},
      op=\circ
   ]{#1 \comp{\circ} #2}
   \symdef{unit}[type=\universe]{\comp{e}}
\end{mathstructure}

A \symname{monoid} is...
```

Output:

A monoid is...

Note that the `\symname{monoid}` is appropriately highlighted and (depending on your pdf viewer) shows a URI on hovering – implying that the `mathstructure` environment has generated a *symbol* `monoid` for us. It has not generated a semantic macro though, since we can not use the `monoid`-symbol *directly*. Instead, we can instantiate it, for example for integers:

Example 25

Input:

```
\symdef{Int}[type=\set]{\comp{\mathbb Z}}
\symdef{addition}[
    type=\funtype{\Int,\Int}{\Int},
    args=2,
    op=+
]{##1 \comp{+} ##2}
\symdef{zero}[type=\Int]{\comp{0}}

$\mathstruct{\Int,\addition!,\zero}$ is a \symname{monoid}.
```

Output:

$\langle \mathbb{Z},+,0\rangle$ is a monoid.

.

So far, we have not actually instantiated `monoid`, but now that we have all the symbols to do so, we can:

Example 26

Input:

```
\instantiate{intmonoid}{
    universe = Int ,
    op = addition ,
    unit = zero
}{monoid}{\mathbb{Z}_{+,0}}

    $\intmonoid{universe}$, $\intmonoid{unit}$ and $\intmonoid{op}{a}{b}$.

    Also: $\intmonoid!$
```

Output:

$\mathbb{Z}$, 0 and $a+b$.
Also: $\mathbb{Z}_{+,0}$

.

\instantiate

So summarizing: \instantiate takes four arguments: The (macro-)name of the instance, a key-value pair assigning declarations in the corresponding mathstructure to symbols currently in scope, the name of the mathstructure to instantiate, and lastly a notation for the instance itself.

It then generates a semantic macro that takes as argument the name of a declaration in the instantiated mathstructure and resolves it to the corresponding instance of that particular declaration.

\instantiate and mathstructure make use of the *Theories-as-Types* paradigm: mathstructure{<name>} does in fact simply create a nested theory with name <name>-structure. The *constant* <name> is defined as Mod(<name>-structure) – a *dependent record type with manifest fields*, the fields of which are generated from (and correspond to) the constants in <name>-structure.
\instantiate appropriately generates a constant whose definiens is a record term of type Mod(<name>-structure), with the fields assigned appropriately based on the key-value-list.

Notably, \instantiate throws an error if not *every* declaration in the instantiated mathstructure is being assigned.

You might consequently ask what the usefulness of mathstructure even is.

\varinstantiate

The answer is that we can also instantiate a mathstructure with a *variable*. The syntax of \varianstantiate is equivalent to that of \instantiate, but all of the key-value-pairs are optional, and if not explicitly assigned (to a symbol *or* a variable declared with \vardef) inherit their notation from the one in the mathstructure environment.

This allows us to do things like:

Example 27

Input:

```
\varinstantiate{varM}{}{monoid}{M}

A \symname{monoid} is a structure
$\varM!:=\mathstruct{\varM{universe},\varM{op}!,\varM{unit}}$
such that
$\varM{op}!:\funtype{\varM{universe},\varM{universe}}{\varM{universe}}$
and...
```

Output:

A monoid is a structure $M := \langle U, \circ, e \rangle$ such that $\circ : U \times U \to U$ and...

.

We will return to this example later, when we also know how to handle the *axioms* of a monoid.

3.4.4 The copymodule Environment

TODO: explain

Given modules:

Example 28

Input:

```
\begin{smodule}{magma}
   \symdef{universe}{\comp{\mathcal U}}
   \symdef{operation}[args=2,op=\circ]{#1 \comp\circ #2}
\end{smodule}
\begin{smodule}{monoid}
   \importmodule{magma}
   \symdef{unit}{\comp e}
\end{smodule}
\begin{smodule}{group}
   \importmodule{monoid}
   \symdef{inverse}[args=1]{{#1}^{\comp{-1}}}
\end{smodule}
```

Output:

.

We can form a module for *rings* by "cloning" an instance of `group` (for addition) and `monoid` (for multiplication), respectively, and "glueing them together" to ensure they share the same universe:

Example 29

Input:

```
\begin{smodule}{ring}
   \begin{copymodule}{group}{addition}
      \renamedecl[name=universe]{universe}{runiverse}
      \renamedecl[name=plus]{operation}{rplus}
      \renamedecl[name=zero]{unit}{rzero}
      \renamedecl[name=uminus]{inverse}{ruminus}
   \end{copymodule}
   \notation*{rplus}[plus,op=+,prec=60]{#1 \comp+ #2}
         \notation*{rzero}[zero]{\comp0}
         \notation*{ruminus}[uminus,op=-]{\comp- #1}
         \begin{copymodule}{monoid}{multiplication}
      \assign{universe}{\runiverse}
      \renamedecl[name=times]{operation}{rtimes}
      \renamedecl[name=one]{unit}{rone}
   \end{copymodule}
   \notation*{rtimes}[cdot,op=\cdot,prec=50]{#1 \comp\cdot #2}
         \notation*{rone}[one]{\comp1}
         Test: $\rtimes a{\rplus c{\rtimes de}}$
\end{smodule}
```

Output:

Test: $a \cdot (c + d \cdot e)$

.

TODO: explain donotclone

3.4.5 The `interpretmodule` Environment

TODO: explain

Example 30

Input:

```
\begin{smodule}{int}
  \symdef{Integers}{\comp{\mathbb Z}}
  \symdef{plus}[args=2,op=+]{#1 \comp+ #2}
  \symdef{zero}{\comp0}
  \symdef{uminus}[args=1,op=-]{\comp-#1}

  \begin{interpretmodule}{group}{intisgroup}
    \assign{universe}{\Integers}
    \assign{operation}{\plus!}
    \assign{unit}{\zero}
    \assign{inverse}{\uminus!}
  \end{interpretmodule}
\end{smodule}
```

Output:

.

3.5 Primitive Symbols (The SᴛEX Metatheory)

TODO: metatheory documentation

Chapter 4

Using sTeX Symbols

4.1 Using sTeX Symbols in Text Mode

4.2 Customizing Highlighting

TODO: references documentation

Chapter 5

sTEX Statements (Definitions, Theorems, Examples, ...)

TODO: statements documentation
TODO: sproofs documentation

Chapter 6

Additional Packages

TODO: tikzinput documentation

6.1 Modular Document Structuring

TODO: document-structure documentation

6.2 Slides and Course Notes

TODO: notesslides documentation

6.3 Homework, Problems and Exams

TODO: problem documentation

TODO: hwexam documentation

Chapter 7

Stuff

`\sTeX` `\stex`

Both print this sTEX logo.

7.0.1 Semantic Macros and Notations

Semantic macros invoke a formally declared symbol.

To declare a symbol (in a module), we use `\symdecl`, which takes as argument the name of the corresponding semantic macro, e.g. `\symdecl{foo}` introduces the macro `\foo`. Additionally, `\symdecl` takes several options, the most important one being its arity. `foo` as declared above yields a *constant* symbol. To introduce an *operator* which takes arguments, we have to specify which arguments it takes.

For example, to introduce binary multiplication, we can do `\symdecl{mult}[args=2]`. We can then supply the semantic macro with arbitrarily many notations, such as `\notation{mult}{#1 #2}`.

Example 31

Input:

```
\symdecl{mult}[args=2]
\notation{mult}{#1 #2}
$\mult{a}{b}$
```

Output:

ab

.

Since usually, a freshly introduced symbol also comes with a notation from the start, the `\symdef` command combines `\symdecl` and `\notation`. So instead of the above, we could have also written

```
\symdef{mult}[args=2]{#1 #2}
```

Adding more notations like `\notation{mult}[cdot]{#1 \comp{\cdot} #2}` or `\notation{mult}[times]{#1 \comp{\times} #2}` allows us to write `$\mult[cdot]{a}{b}$` and `$\mult[times]{a}{b}$`:

Example 32

Input:

```
\notation{mult}[cdot]{#1 \comp{\cdot} #2}
\notation{mult}[times]{#1 \comp{\times} #2}
$\mult[cdot]{a}{b}$ and $\mult[times]{a}{b}$
```

Output:

$a \cdot b$ and $a \times b$

˙Not using an explicit option with a semantic macro yields the first declared notation, unless changed[2].

EdN:2

Outside of math mode, or by using the starred variant `\foo*`, allows to provide a custom notation, where notational (or textual) components can be given explicitly in square brackets.

Example 33

Input:

```
$\mult*{\arg{a}\comp{\ast}\arg{b}}$ is the
\mult{\comp{product of} \arg{$a$} \comp{and} \arg{$b$}}
```

Output:

$a * b$ is the product of a and b

.

In custom mode, prefixing an argument with a star will not print that argument, but still export it to OMDoc:

Example 34

Input:

```
\mult{\comp{Multiplying} \arg*{$\mult{a}{b}$} again by \arg{$b$}} yields...
```

Output:

Multiplying again by b yields...

˙The syntax `*[`⟨*int*⟩`]` allows switching the order of arguments. For example, given a 2-ary semantic macro `\forevery` with exemplary notation `\forall #1. #2`, we can write

[2] EdNote: TODO

Example 35

Input:

```
\symdecl{forevery}[args=2]
\forevery{\arg[2]{The proposition $P$} \comp{holds for every} \arg[1]{$x\in A$}}
```

Output:

The proposition P holds for every $x \in A$

.

When using `*[n]`, after reading the provided (nth) argument, the "argument counter" automatically continues where we left off, so the `*[1]` in the above example can be omitted.

For a macro with arity > 0, we can refer to the operator *itself* semantically by suffixing the semantic macro with an exclamation point `!` in either text or math mode. For that reason `\notation` (and thus `\symdef`) take an additional optional argument `op=`, which allows to assign a notation for the operator itself. e.g.

Example 36

Input:

```
\symdef{add}[args=2,op={+}]{#1 \comp+ #2}
The operator $\add!$ adds two elements, as in $\add ab$.
```

Output:

The operator + adds two elements, as in $a+b$.

.

`*` is composable with `!` for custom notations, as in:

Example 37

Input:

```
\mult!{\comp{Multiplication}} (denoted by $\mult!*{\comp\cdot}$) is defined by...
```

Output:

Multiplication (denoted by $\cdot$) is defined by...

.

The macro `\comp` as used everywhere above is responsible for highlighting, linking, and tooltips, and should be wrapped around the notation (or text) components that should be treated accordingly. While it is attractive to just wrap a whole notation, this would also wrap around e.g. the arguments themselves, so instead, the user is tasked with marking the notation components themself.

The precise behaviour of `\comp` is governed by the macro `\@comp`, which takes two arguments: The tex code of the text (unexpanded) to highlight, and the URI of the current symbol. `\@comp` can be safely redefined to customize the behaviour.

The starred variant `\symdecl*{foo}` does not introduce a semantic macro, but still declares a corresponding symbol. `foo` (like any other symbol, for that matter) can then be accessed via `\STEXsymbol{foo}` or (if `foo` was declared in a module `Foo`) via `\STEXModule{Foo}?{foo}`.

both `\STEXsymbol` and `\STEXModule` take any arbitrary ending segment of a full URI to determine which symbol or module is meant. e.g. `\STEXsymbol{Foo?foo}` is also valid, as are e.g. `\STEXModule{path?Foo}?{foo}` or `\STEXsymbol{path?Foo?foo}`

There's also a convient shortcut `\symref{?foo}{some text}` for `\STEXsymbol{?foo}![some text]`

Other Argument Types

So far, we have stated the arity of a semantic macro directly. This works if we only have "normal" (or more precisely: `i`-type) arguments. To make use of other argument types, instead of providing the arity numerically, we can provide it as a sequence of characters representing the argument types – e.g. instead of writing `args=2`, we can equivalently write `args=ii`, indicating that the macro takes two `i`-type arguments.

Besides `i`-type arguments, STEX has two other types, which we will discuss now.

The first are *binding* (`b`-type) arguments, representing variables that are *bound* by the operator. This is the case for example in the above `\forevery`-macro: The first argument is not actually an argument that the `forevery` "function" is "applied" to; rather, the first argument is a new variable (e.g. x) that is *bound* in the subsequent argument. More accurately, the macro should therefore have been implemented thusly:

```
\symdef{forevery}[args=bi]{\forall #1.\; #2}
```

`b`-type arguments are indistinguishable from `i`-type arguments within STEX, but are treated very differently in OMDoc and by Mmt. More interesting *within* STEX are `a`-type arguments, which represent (associative) arguments of flexible arity, which are provided as comma-separated lists. This allows e.g. better representing the `\mult`-macro above:

Example 38

Input:

```
1 \symdef{mult}[args=a]{#1}{##1 \comp\cdot ##2}
2 $\mult{a,b,c,{d^e},f}$
```

Output:

$a \cdot b \cdot c \cdot d^e \cdot f$

As the example above shows, notations get a little more complicated for associative arguments. For every `a`-type argument, the `\notation`-macro takes an additional argument that declares how individual entries in an `a`-type argument list are aggregated. The first notation argument then describes how the aggregated expression is combined into the full representation.

For a more interesting example, consider a flexary operator for ordered sequences in ordered set, that taking arguments `{a,b,c}` and `\mathbb{R}` prints $a \leq b \leq c \in \mathbb{R}$. This operator takes two arguments (an `a`-type argument and an `i`-type argument), aggregates the individuals of the associative argument using `\leq`, and combines the result with `\in` and the second argument thusly:

Example 39

Input:

```
\symdef{numseq}[args=ai]{#1 \comp\in #2}{##1 \comp\leq ##2}
$\numseq{a,b,c}{\mathbb R}$
```

Output:

$a \leq b \leq c \in \mathbb{R}$

.

Finally, `B`-type arguments combine the functionalities of `a` and `b`, i.e. they represent flexary binding operator arguments.

EdN:3 EdN:4 [3] [4]

Precedences

Every notation has an (upwards) *operator precedence* and for each argument a (downwards) *argument precedence* used for automated bracketing. For example, a notation for a binary operator `\foo` could be declared like this:

```
\notation{foo}[prec=200;500x600]{#1 \comp{+} #2}
```

assigning an operator precedence of 200, an argument precedence of 500 for the first argument, and an argument precedence of 600 for the second argument.

STEX insert brackets thusly: Upon encountering a semantic macro (such as `\foo`), its operator precedence (e.g. 200) is compared to the current downwards precedence (initially `\neginfprec`). If the operator precedence is *larger* than the current downwards precedence, parentheses are inserted around the semantic macro.

Notations for symbols of arity 0 have a default precedence of `\infprec`, i.e. by default, parentheses are never inserted around constants. Notations for symbols with arity > 0 have a default operator precedence of 0. If no argument precedences are explicitly provided, then by default they are equal to the operator precedence.

Consequently, if some operator A should bind stronger than some operator B, then As operator precedence should be smaller than Bs argument precedences.

For example:

Example 40

Input:

```
\notation{plus}[prec=100]{#1 \comp{+} #2}
\notation{times}[prec=50]{#1 \comp{\cdot} #2}
$\plus{a}{\times{b}{c}}$ and $\times{a}{\plus{b}{c}}$
```

Output:

$a+b \cdot c$ and $a \cdot (b+c)$

.

[3] EdNote: what about e.g. \int _x\int _y\int _z f dx dy dz?
[4] EdNote: "decompose" a-type arguments into fixed-arity operators?

7.0.2 Archives and Imports

Namespaces

Paths in Import-Statements