The sTeX Manual

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

2023-03-21

If you have questions or problems with sTeX, you can talk to us directly at *https://matrix.to/#/#stex:fau.de*.

The dynamic HTML version of this document can be found at *https://stexmmt.mathhub.info/:sTeX/fullhtml?archive=sTeX/Documentation&filepath=manual.xhtml*

Contents

Chapter 1

Basics

1.1 Package and Class Options

- `debug=`⟨*prefixes*⟩: (see Developer Manual)
- `lang=`⟨*languages*⟩: If set, sTeX will load the `babel` package with the provided languages. Supported languages (currently) are:

`ar`	arabic
`bg`	bulgarian
`de`	german (with option `ngerman`)
`en`	english
`fi`	finnish
`fr`	french
`ro`	romanian
`ru`	russian
`tr`	turkish (with option `shorthands=:!`)

- `mathhub=`⟨*path*⟩: Uses the provided file path as `MathHub` directory (see section 1.2).
- `usesms`/`writesms`: If `writesms` is set, content loaded from external math archives (i.e. modules) is persisted in the file `\jobname.sms`.

  If `usesms` is set, the content of the `.sms`-file is loaded, obviating the need to reprocess the original files.

  The options are not mutually exclusive, but care should be taken if dependencies have changed between builds.

  This offers two advantages:

  1. If a document has many (transitive) dependencies, `usesms` should significantly speed up the build process, and
  2. setting `usesms` allows for distributing the `.sms`-file to make the document *standalone*, allowing for compilation without needing imported/used modules to be present.

The options debug, mathhub, usesms and writesms can also be set by the environment variables STEX_DEBUG, MATHHUB, STEX_USESMS and STEX_WRITESMS. In fact, the MATHHUB environment variable is the recommended way to set the MathHub directory. This is the only option where the *package option* overrides the environment variable.

The environment variables for USE/WRITESMS are particularly useful, in that they allow for convenient compilation workflows. For example, the Build PDF/XHTML/OMDoc-button in the IDE does the following:

```
STEX_WRITESMS=true pdflatex <job>.tex
[bibtex|biber] <job>
STEX_USESMS=true pdflatex <job>.tex
STEX_USESMS=true pdflatex <job>.tex
```

Guaranteeing (in the first run) that all dependencies are loaded from their respective sources and persisted, and in the two subsequent runs read from the generated .sms file, (likely) speeding ub the subsequent runs significantly.

1.2 Math Archives and the MathHub Directory

sTeX uses math archives to organize document content modularly, without a user having to specify absolute paths, which would differ between users and machines.

All sTeX archives need to exist in the local MathHub-directory. sTeX knows where this folder is via one of five 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.

4. If that too fails, sTeX will look for a file ~/.stex/mathhub.path. If this file exists, sTeX will assume that it contains the path to the local MathHub-directory. This method is recommended on systems where it is difficult to set environment variables, and is used by the IDE setup.

5. Finally, if all else fails, sTeX considers ~/MathHub to be the MathHub directory.

The sTeX IDE allows you to directly download math archives from gl.mathhub.info – currently available archives there are:

- sTeX/* – a group of semi-experimental documents showcasing sTeX3.3 features,

- smglom/* – a vast collection of multilingual modules of concepts in mathematics and computer science. The SMGloM predates sTeX3 and is thus largely "underannotated" with respect to (formal) semantics,

- MiKoMH/* – a vast collection of lecture slides and notes in computer science for courses held by Michael Kohlhase. They largely make use of SMGloM modules.

1.2.1 The Structure of Math Archives

An archive `group/name` is stored in the directory `<MathHub>/group/name`; e.g. assuming your local `MathHub`-directory is set as `/user/foo/MathHub`, then in order for the `sTeX/Documentation` archive to be found by the sTeX system, it needs to be in `/user/foo/MathHub/sTeX/Documentation`.

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 discussed in section 1.3.

1.2.2 MANIFEST.MF-Files

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

```
id: sTeX/Documentation
ns: http://mathhub.info/sTeX/Documentation
narration-base: http://mathhub.info/sTeX/Documentation
format: stex
title: The sTeX Documentation
teaser: The full Documentation for the sTeX system
url-base: https://stexmmt.mathhub.info/:sTeX
dependencies:sTeX/ComputerScience/Software,sTeX/MathTutorial
ignore: */code/*|*/tikz/*|*/tutorial/solution/*
```

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

`id`: The name of the archive, including its group (e.g. `sTeX/Document`). This is used by the MMT system in favor of the directory, but TeX's limited access to the file system enforces the directory structure.

`source-base` or

`ns`: The namespace from which all symbol and module MMT-URIs in this archive are formed.

`narration-base`: The namespace from which all document MMT-URI in this repository are formed. It can safely match the `ns`-field.

`url-base`: A URL that is formed as a basis for *external references.* and hyperlinks. An MMT (or comparable system) instance should run there and host (sTeX-generated) HTML.

`dependencies`: All archives that this archive depends on. sTeX ignores this field, but MMT can pick up on them to resolve dependencies, e.g. when downloading archives in the IDE, which will also download all dependencies.

`ignore`: A regular expression of `.tex` files in the `source` directory that should be ignored; e.g. they will not be compiled when building a whole directory or archive in the IDE.

1.3 The lib-Directory

A math archive group/archive may have a lib directory primarily intended for preamble code, packages, .bib files, etc., the files in which can be referenced in various ways.

Additionally, a *group* of archives group may have an additional archive group/meta-inf. If this meta-inf archive has a /lib-subdirectory, they too will be considered by the following.

\libinput

\libinput {some/file} searches for a file some/file[.tex] 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 \inputs all found files in reverse order; e.g. \libinput{preamble} in a .tex-file in sTeX/Documentation will *first* input .../sTeX/meta-inf/lib/preamble.tex and then ../sTeX/Documentation/lib/preamble.tex.

\libinput will throw an error if *no* candidate for some/file is found.

\libinput[some/archive]{some/file} will do the same, but starting in the lib directory of the math archive some/archive.

\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.

\libusepackage throws an error if not *exactly one* candidate for some/file.sty is found.

\addmhbibresource

\addmhbibresource [some/archive]{some/file} searches for a file like some/file.bib in some/archive's lib directory and calls \addbibresource to the result.

\libusetikzlibrary

\libusetikzlibrary behaves like \libusepackage but looks specifically for tikz libraries and calls \usetikzlibrary on the results.

throws an error if not *exactly one* candidate for for the library is found.

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

```
\documentclass{stex}
\libinput{preamble}
\setsectionlevel{<your preference>}
\begin{document}
    \IfInputref{}{
      ...
      \maketitle
      \ifstexhtml \else \tableofcontents \fi
    }
    ...
    \IfInputref{}{\libinput{postamble}}
\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 a `postamble.tex` can e.g. print the bibliography, index etc.

`\libusepackage` is particularly useful in such a `preamble.tex` when we want to use custom packages that are not part of a TEX distribution, or on CTAN. In this case we commit the respective packages in one of the `lib` folders and use `\libusepackage` to load them.

1.4 Basic Macros

`\sTeX`
`\stex`

The `\sTeX` macro produces this sTEX logo. It is provided by the stex-logo package, included with the stex package.

`\ifstexhtml`

The TEX conditional `\ifstexhtml` is *true* if the current compilation generates HTML, and *false* otherwise (i.e. generates PDF).

`\STEXinvisible`

`\STEXinvisible{⟨code⟩}`

Processes ⟨*code*⟩, but does not generate any output. In the HTML, ⟨*code*⟩ is exported with `display:none`.

Can be used to declare formal content and preserve its semantics in HTML without generating output.

Chapter 2

Document Features

2.1 Document Fragments

sfragment (*env.*) To make reusability of document fragments more feasible, sTeX provides the **sfragment** environment. `\begin{sfragment}[id=⟨id⟩,short=⟨short title⟩]{section title}` calls `\part`, `\chapter`, `\section`, `\subsection`, `\subsubsection`, `\paragraph` or `\subparagraph` with argument `{section title}` depending on the current *section level* and availability, and increases the level accordingly.

The ⟨*id*⟩ can be used for cross-document references (see section 2.3).

blindfragment (*env.*) In the case where we want to increase the section level *without* producing a corresponding section header, the **blindfragment** environment can be used. This allows e.g. typesetting `\section`s before the first `\chapter`.

`\skipfragment` The `\skipfragment` macro "skips an `sfragment`", i.e. it just steps the respective sectioning counter. This macro is useful, when we want to keep two documents in sync structurally, so that section numbers match up: Any section that is left out in one becomes a `\skipfragment`.

`\setsectionlevel` The `\setsectionlevel` macro sets the current section level to that provided as argument. This is particularly useful in the preamble of a document, as to be ignored in e.g. `\inputref` and make sure that sectioning proceeds as desired; e.g. `\setsectionlevel{section}` make sure that the first `sfragment` will be typeset as a `\section` (rather than e.g. a `\part`).

`\currentsectionlevel`
`\Currentsectionlevel` The `\currentsectionlevel` macro produces the literal string corresponding to the current section level – e.g. within a chapter (but outside of a section), `\currentsectionlevel` produces "chapter".

The `\Currentsectionlevel` macro does the same, but capitalizes the first letter; e.g. in the above situation, `\Currentsectionlevel` produces "Chapter".

2.2 Using and Referencing Document Fragments

\inputref

\inputref [⟨*archive*⟩]{⟨*file*⟩}

Inputs the file ⟨*file*⟩ in ⟨*archive*⟩'s `source` directory. If [⟨*archive*⟩] is empty, the current archive's `source` directory is used. If there is no current archive, ⟨*file*⟩ is resolved relative to the current file.

The file's content is processed within a TEX group when using `pdflatex`. When converting to HTML however, the file is not processed *at all*, and instead, a reference to the file is inserted, that can be replaced by the HTML generated by the referenced file by e.g. the MMT system.

This is the recommended method to assemble documents from individual `.tex` files.

\mhinput

Like `\inputref`, but actually calls `\input` in both PDF and HTML mode. Useful for small fragments or those without modules, but generally `\inputref` should be preferred.

\ifinputref
\IfInputref

`\ifinputref` is a TEX conditional for whether the current file is currently processed via `\inputref`.

`\IfInputref` {⟨*true code*⟩}{⟨*false code*⟩} behaves like `\ifinputref`⟨*true code*⟩`\else`⟨*false code*⟩`\fi` when using `pdflatex`; in HTML mode however, *both* arguments are processed and marked-up accordingly, so a hosting server (like MMT) can dynamically decide which parts to show or omit.

\mhgraphics
\cmhgraphics

If the graphicx package is loaded, `\mhgraphics` takes the same arguments as `\includegraphics`, with the additional optional key `archive`. It then resolves the file path in `\mhgraphics[archive=some/archive]{some/image}` relative to the `source`-folder of the `some/archive` archive. If no `archive` is provided, the file path `some/image` is resolved relative to the current archive (if existent).

`\cmhgraphics` additional wraps the image in a `center`-environment.

\lstinputmhlisting
\clstinputmhlisting

Like `\mhgraphics`, but for `\lstinputlisting` instead of `\includegraphics`. Only defined if the listings package is loaded.

2.3 Cross-Document References

If we take features like `\inputref` and `\mhinput` (and the `sfragment` environment) seriously and build large documents modularly from individually compiling documents for sections, chapters and so on, cross-referencing becomes an interesting problem.

Say, we have a document `main.tex`, which `\inputref`s a section `section1.tex`, which references a definition with label `some_definition` in `section2.tex` (subsequently also `\inputref`ed in `main.tex`). Then the numbering of the definition will depend on the *document context* in which the document fragment `section2.tex` occurs - in `section2.tex` itself (as a standalone document), it might be *Definition 1*, in `main.tex` it might be *Definition 3.1*, and in `section1.tex`, the definition *does not even occur*, so it needs to be referenced by some other text.

What we would want in that instance is an equivalent of `\autoref`, that takes the document context into account to yield something like *Definition 1*, *Definition 3.1* or "*Definition 1 in the section on Foo*" respectively.

For that to work, we need to supply (up to) three pieces of information:

- The *label* of the reference target (e.g. `some_definition`),
- (optionally) the *file*/document containing the reference target (e.g. `section2`). This is not strictly necessary if the reference target occurs in the *same* document, but if not, we need to know where to find the label,
- (optionally) the document context, in which we want to refer to the reference target (e.g. `main`).

Additionally, the document in which we want to reference a label needs a title for external references.

`\sref`

`\sref [archive=⟨archive1⟩,file=⟨file1⟩]`
`{⟨label⟩}[archive=⟨archive2⟩,file=⟨file2⟩,title=⟨title⟩]`

This macro references ⟨*label*⟩ (declared in ⟨*file1*⟩ in math archive ⟨*archive1*⟩). If the object (section, figure, etc.) with that label occurs (eventually) in the same document, `\sref` will ignore the second set of optional arguments and simply defer to `\autoref` if that command exists, or `\ref` if the hyperref package is not included.

If the referenced object does *not* occur in the current document however, `\sref` will refer to it by the object's name as it occurs in the file ⟨*file2*⟩ in archive ⟨*archive2*⟩, followed by the title.

In HTML mode, the reference additionally links to the HTML of the `file1`.[1]

This works by storing labels during compilation in a file ⟨*jobname*⟩`.sref`, analogous to e.g. the `.toc`. Note that this consequently requires both `file1.tex` and `file2.tex` to have been compiled previously, to generate the `.sref` file.

For example, doing
`\sref[file=tutorial/full.en]{sec:basics}[file=tutorial.en,title=the \stex Tutorial]`
in this very document fragment (`[sTeX/Documentation]macros/sref.en.tex`) will yield Part I (The Basics) in the sTeX Tutorial if compiled itself, or if compiled as part of the sTeX manual, and will yield the `\autoref` link chapter 2 in the documentation (which includes the tutorial).

`\srefsetin`

`\srefsetin [⟨archive2⟩]{⟨file2⟩}{⟨title⟩}`

Sets a default value for the optional arguments ⟨*archive2*⟩, ⟨*file2*⟩ and ⟨*title*⟩ of `\sref`. If the second set of optional arguments in `\sref` are omitted, these default values are used. Particularly useful to set in a preamble.

`\sreflabel`

`\sreflabel {⟨label⟩}` sets a label analogous to `\label{⟨label⟩}`, but for use in `\sref`.

Note that for every sTeX macro or environment that takes an optional `id=⟨id⟩` argument, the ⟨*id*⟩ (if non-empty) generates an `\sreflabel` automatically.

For example, `\begin{sfragment}[id=foo]{Foo}` is equivalent to
`\begin{sfragment}{Foo}\sreflabel{foo}`.

\extref

\extref [archive=⟨*archive1*⟩,file=⟨*file1*⟩]
{⟨*label*⟩}{archive=⟨*archive2*⟩,file=⟨*file2*⟩,title=⟨*title*⟩}

Like \sref, but with the third argument mandatory, \extref will *always* produce the output as if ⟨*label*⟩ would *not* occur in the current document.

Chapter 3

Modules and Symbols

3.1 Modules

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

> ↪M→ ⸺M→ ~T⇝ An sTEX module corresponds to an MMT/OMDoc *theory*. As such it gets assigned an MMT-URI (*universal resource identifier*) of the form `<namespace>?<module-name>`.

`smodule` (*env.*) 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 keyword arguments, all of which are optional:

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

`style` (⟨*string*⟩∗) for use in customizations, see chapter 5

`id` (⟨*string*⟩) for cross-referencing, see `\sreflabel`.

`ns` (⟨*URI*⟩) the namespace to use. *Should not be used, unless you know precisely what you're doing.* If not explicitly set, is computed from the containing file and archive's namespace.

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

`sig` (⟨*language*⟩) see below.

`\STEXexport` `\STEXexport{⟨code⟩}` executes ⟨*code*⟩ immediately and every time the current module is being used.

For technical reasons, \STEXexport processes its content in the expl3 category code scheme – what this means is that all spaces are ignored entirely, and the characters _ and : are valid characters in macro names.
In practice, this means you will have to use the ~ character for spaces, and if you want to use a subscript _, you should use the macro \c_math_subscript_token instead.
Also, note that no *global* macro definitions should happen in \STEXexport; this can lead to unexpected behaviour if the containing module has been used previously in the current document.

3.1.1 Signature Modules, Languages, and Multilinguality

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.

For example, we can have a file Foo.en.tex, that declares and documents a module Foo (using \begin{smodule}{Foo}). If we put a file Foo.de.tex next to it, we can do \begin{smodule}[sig=en]{Foo} to have all the content in the module Foo (as declared in Foo.en.tex) available and translate its document content to german.

The MMT backend, when serving sTeX content as HTML, will always attempt to find documentation in the language corresponding to the context; e.g. a user's preference.

3.2 Symbol Declarations

\symdecl

\symdecl {⟨*mname*⟩}[⟨*options*⟩]

The \symdecl macro is the simplest way to introduce a new symbol. If ⟨*options*⟩ contains name=⟨*name*⟩, then ⟨*name*⟩ is the *name* of the symbol; otherwise, ⟨*mname*⟩ is used for the *name*. Additionally, a semantic macro \mname is generated.

The starred variant \symdecl* does not generate a semantic macro, in which case the name-option is superfluous.

\symdecl introduces a new MMT/OMDoc constant in the current module (i.e. MMT/OMDoc theory). Correspondingly, they get assigned the MMT-URI <module-URI>?<constant-name>.

\symdecl takes the following optional arguments:

name see above,

args the arity of the symbol and its semantic macro; may be a number 0...9 or a string consisting of the characters i, a, b and B of length ≤ 9,

type the symbol's type,

def the symbol's definiens,

return the symbol's *return code* (see below), most commonly the semantic macro of a mathematical structure,

assoc how to resolve arguments of mode `a` or `B`; may be `pre`, `bin`, `binl`, `binr` or `conj`,

reorder how to reorder the arguments in OMDoc (*advanced*),

role symbols with certain roles are treated in particular ways in MMT/OMDoc (*advanced*),

argtypes TODO[2].

`\textsymdecl`

`\textsymdecl{⟨mname⟩}[⟨options⟩]{⟨code⟩}`

Like `\symdecl`, but requires that the symbol has arity 0 (hence `\textsymdecl` does not take the `args`-option), and generates a semantic macro that takes no arguments in either text or math mode, and produces marked-up ⟨*code*⟩ as output.

Additionally, a macro `\`⟨*mname*⟩`name` is generated that produces ⟨*code*⟩ without any semantic markup.

`\symdef`

`\symdef{⟨mname⟩}[⟨options⟩]{⟨notation⟩}`

Combines the functionalities and optional arguments of `\symdecl` and `\notation` in one.

3.2.1 Returns

Assume we have a symbol `foo` with semantic macro `\foo`, (examplary) taking two arguments, and `return=`⟨*code*⟩. If we do `\foo{a}{b}!`, the return code is simply ignored. If we do `\foo{a}{b}` *without* the `!`, here is what happens:

1. sTeX will replace `#1` and `#2` in ⟨*code*⟩ by `a` and `b`, yielding ⟨*retcode*⟩.
2. sTeX will insert ⟨*retcode*⟩`{\foo{a}{b}!}` in the input token stream.

This means that ⟨*code*⟩ should contain at most ⟨*arity of foo*⟩ argument markers, and eat precisely one argument appended to ⟨*code*⟩.

When in doubt, we recommend only using semantic macros for mathematical structures (with only optional arguments) and `\apply` (with only optional arguments) in `return`.

3.3 Referencing Symbols

`\symref`
`\sr`

`\symref{⟨symbol⟩}{⟨text⟩}`

The `\symref` macro (and its short version `\sr`) is the most general variant to mark-up arbitrary LaTeX code ⟨*text*⟩ with the symbol.

[2] TODO: experimental

This is as good a place as any other to explain how sTeX resolves a string `symbol` to an actual symbol.
If `\symbol` is a semantic macro, then sTeX has no trouble resolving `symbolname` to the full MMT-URI of the symbol that is being invoked.
However, especially in `\symname` (or if a symbol was introduced using `\symdecl*` without generating a semantic macro), we might prefer to use the *name* of a symbol directly for readability – e.g. we would want to write `A \symname{natural number} is...` rather than `A \symname{Nat} is...`. sTeX attempts to handle this case thusly:
If `symbol` does *not* correspond to a semantic macro `\symbol` and does *not* contain a `?`, then sTeX checks all symbols currently in scope until it finds one, whose name is `symbol`. If `symbol` is of the form `pre?name`, sTeX first looks through all modules currently in scope, whose full MMT-URI ends with `pre`, and then looks for a symbol with name `name` in those. This allows for disambiguating more precisely, e.g. by saying `\symname{Integers?addition}` or `\symname{RealNumbers?addition}` in the case where several `addition`s are in scope.

`\symref{⟨symbol⟩}{⟨text⟩}` in MMT/OMDoc generates the term `<OMS name="{⟨symbol URI⟩}"/>`.

`\symname` `\sn` `\Symname` `\Sn` `\sns` `\Sns`

`\symname[pre=⟨pre⟩,post=⟨post⟩]{⟨symbol⟩}`
If the symbol referenced by ⟨*symbol*⟩ has name `name`, this is a shortcut for `\symref{⟨symbol⟩}{⟨pre⟩name⟨post⟩}`.

For example, given a symbol `agroup` with name `abelian group`, we can do `\symname[pre=Non-,post=s]{agroup}` to produce Non-abelian groups.

`\sn` is a shorter variant for `\symname`; `\Symname` and `\Sn` additionally capitalize the first letter. `\sns` and `\Sns` are short for `\sn [post=s]` and `\Sn [post=s]`, respectively.

`\srefsym` `\srefsymuri`

`\srefsym{⟨symbol⟩}{⟨text⟩}`
turns ⟨*text*⟩ into a link to

- The documentation of ⟨*symbol*⟩, if it occurs in the same document, or
- the symbol's documentation online, based on the containing math archive's `url-base`.

`\srefsymuri` does the same, but expects a symbol's full MMT-URI as first argument. This is particularly useful for e.g. customizing highlighting (see chapter 5).

`\symuse`

`\symuse{⟨symbol⟩}` behaves exactly like a semantic macro for ⟨*symbol*⟩.

3.4 Notations and Semantic Macros

`\notation`

`\notation{⟨symbol⟩}[⟨options⟩]{⟨code⟩}`

introduces a new notation for the referenced symbol.

The starred variant `\notation*` sets this notation as the (new) default notation.

The optional arguments are:

- `prec=`⟨*opprec*⟩`;`⟨*argprec 1*⟩`x...x`⟨*argprec n*⟩: An operator precedence and one argument precedence for each argument of the semantic macro. If no argument precedences are given, all argument precedences are equal to the operator precedence. By default, all precedences are 0, unless the symbol takes no argument, in which case the operator precedence is `\neginfprec` (negative infinity).

  `prec=nobrackets` is an abbreviation for `prec=\neginfprec;\infprec x...x\infprec`.

- `op=`⟨*code*⟩: An operator notation. If none is given, the notation component marked with `\maincomp` is used. If no `\maincomp` occurs in the notation, the default operator notation is `\symname{`⟨*symbol*⟩`}`.

- `variant=`⟨*id*⟩: An id for this notation. The key `variant=` can be omitted; i.e. `\notation[foo]` is equivalent to `\notation[variant=foo]`.

`\comp`
`\maincomp`

`\comp` is used to mark notation components in a `\notation` to be highlighted. Additionally, each notation can use `\maincomp` at most once to mark the *primary* notation component.

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 macros 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 application `\plus{a}{b}` 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 `\plus`.

Similarly, a semantic macro can not conceptually be part of the notation of `\plus`,

since a symbol represents a *distinct (mathematical) concept* with *its own semantics*, and 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, like us), and might want to give those another thought - quite likely, the concept you aim to implement does not actually represent a semantically meaningful (mathematical) concept, and you will want to use `\def` and similar native LaTeX macro definitions rather than semantic macros.

`\setnotation` The first notation provided will stay the default notation unless explicitly changed:

`\setnotation{`⟨*symbol*⟩`}{`⟨*id*⟩`}` sets the default notation of ⟨*symbol*⟩ to that with id ⟨*id*⟩.

3.4.1 Precedences and Bracketing

`\infprec` `\neginfprec` `\infprec` and `\neginfprec` represent *infinitely large* and *infinitely small* precedences, respectively.

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 inserts parentheses.
When sTeX steps into an argument of a semantic macro, it sets p_d to the respective argument precedence of the notation used.

Example 1

Consider semantic macros `\plus` and `\mult` taking two arguments, with notations $a + b$ and $a \cdot b$ respectively, and precedences 100 for `\plus` and 50 for `\mult`.

Consider `$\plus{a,\mult{b,\plus{c,d}}}$` (i.e. $a + b \cdot (c + d)$). Then:

1. sTeX starts out with $p_d =$`\infprec`.
2. sTeX encounters `\plus` with $p_{op} = 100$. Since $100 \not>$`\infprec`, it inserts no parentheses.
3. Next, sTeX encounters the two arguments for `\plus`. Both have no specifically provided argument precedence, so sTeX uses $p_d = p_{op} = 100$ for both and recurses.
4. Next, sTeX encounters `\mult{b,...}`, whose notation has $p_{op} = 50$.
5. We compare to the current downward precedence p_d set by `\plus`, arriving at $p_{op} = 50 \not> 100 = p_d$, so sTeX again inserts no parentheses.

6. Since the notation of \mult has no explicitly set argument precedences, sTeX again uses the operator precedence for the arguments of \mult, hence sets $p_d = p_{op} = 50$ and recurses.
7. Next, sTeX encounters the inner \plus{c,...} whose notation has $p_{op} = 100$. We compare to the current downward precedence p_d set by \mult, arriving at $p_{op} = 100 > 50 = p_d$ – which finally prompts sTeX to insert parentheses, and we proceed as before.

\dobracket

\dobrackets{⟨*code*⟩} wraps parentheses around {⟨*code*⟩}.

\withbrackets

\withbrackets{⟨*left*⟩}{⟨*right*⟩}{⟨*code*⟩} uses the opening and closing parentheses ⟨*left*⟩ and ⟨*right*⟩ for the next pair of parentheses automatically inserted in {⟨*code*⟩}.

3.4.2 Notations for Argument Sequences

The following macros can be used in notations that take mode a or B arguments:

\argsep

\argsep{⟨*parameter token*⟩}{⟨*separator*⟩}

takes the elements of the argument sequence in position ⟨*parameter token*⟩ and separates them by ⟨*separator*⟩.

Note that the first argument *must* be a parameter token of the form #k, and the argument at position k of the notation has to have argument mode a or B.

\argmap

\argmap{⟨*parameter token*⟩}{⟨*code*⟩}{⟨*separator*⟩}

takes the elements of the argument sequence in position ⟨*parameter token*⟩, applies the code {⟨*code*⟩} to each of them (which therefore should use ##1) and separates them by ⟨*separator*⟩.

For example, the notation {\argmap{#1}{X^{##1}}{++}} applied to the argument {a,b,c} produces $X^a + +X^b + +X^c$.

\argarraymap

TODO[3]

3.4.3 Semantic Macros

Assume we have a semantic macro \smacro taking (exemplary) two arguments. The precise behaviour of \smacro depends on whether we are in text or math mode.

Math Mode \smacro! produces the default operator notation of its symbol. Without !, \smacro expects at least two arguments, and \smacro{a}{b}! produces the default notation of its symbol.

If the symbol has a return code, then \smacro{a}{b} continues with executing the return code. Otherwise, \smacro{a}{b} also simply produces the default operator notation.

The starred variants \smacro* and \smacro!* behave as in *text mode*.

Text Mode \smacro!{⟨*arg*⟩} marks up ⟨*arg*⟩ similarly to how \symref{smacro}{⟨*arg*⟩} would.

Without the !, \smacro still only takes a single argument, but it is expected, that within ⟨*arg*⟩, the arguments for the symbol are explicitly marked up. The \comp macro is allowed in ⟨*arg*⟩ to determine the components of ⟨*arg*⟩ to be highlighted.

\arg

The \arg macro can be used to explicitly mark the arguments of a semantic macro in text mode.

By default, they are numbered consecutively; e.g. \smacro{...\arg{a}...\arg{b}} determines a and b to be the (first and second) arguments.

The starred variant \arg* allows for marking up the arguments, but does not produce any output. This can be used to provide arguments that are not mentioned in the text we want to mark up, because they are implicitly obvious or mentioned elsewhere.

If we want to change the order of the arguments, we can provide the precise argument number as an optional argument; e.g. \smacro{...\arg[2]{a}...\arg[1]{b}} determines b to be the first and a to be the second argument.

An argument number may be used repeatedly, if the corresponding argument mode is a or B.

> Applications of semantic macros with arguments are translated to MMT/OMDoc as OMA-terms with head <OMS name="⟨*symbol*⟩"/>, or <OMBIND name="⟨*symbol*⟩"/>, depending on the absence or presence of argument mode b or B arguments. Semantic macros with no arguments or invoked with ! correspond to OMS directly.

3.5 Simple Inheritance

There are three macros that allow for opening a module, making its contents available for use:

\usemodule

\usemodule{⟨*module*⟩} is allowed anywhere and makes the module's contents available up to the current TeX group. This is the right macro to use outside of modules, or when none of its contents use any of the used module's symbols directly (e.g. in types or definientia).

\requiremodule

\requiremodule{⟨*module*⟩} is only allowed in modules and makes the required module's contents available within the current module. The imported symbols can be safely used in types and definientia, but not in the return code of symbols, and the imported content is not exported further – i.e. using the current module does not also open the required module.

\importmodule

\importmodule{⟨*module*⟩} is only allowed in modules and makes the required module's contents available within the current module. The imported symbols can be safely used anywhere, and the imported content exported to any modules subsequently importing the current one.

↪M⇀ —M⇀ ⇝T⇝ In MMT, every *document* and every module induces an MMT theory. \usemodule induces and MMT include in the document theory, \importmodule and \requiremodule both induce an include in the module's theory.

It is worth going into some detail how exactly \usemodule, \importmodule and \requiremodule 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 MMT-URI.

Ideally, sTeX would allow for arbitrary MMT-URIs for modules, with no forced relationships between the *logical* namespace of a module and the *physical* location of the file declaring it – like MMT in fact allows for.
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 math 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 math 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` directory 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 file 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 file, 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` directory.
- 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` directory, 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.

3.6 Variables and Sequences

`\vardef`

`\vardef{⟨mname⟩}[⟨options⟩]{⟨notation⟩}`

Takes the same arguments as `\symdef`, but produces a variable rather than a symbol. Variables definitions are always local to the current TEX group and are allowed anywhere (i.e. outside of modules).

⟨*options*⟩ may include the additional keyword `bind`, in which case the variable will be appropriately abstracted away in statements (see also `\varbind`).

Unlike `\symdef`, there is no starred variant `\vardef*` – variables always generate a semantic macro.

The semantic macro for a variable behaves analogously to that of a symbol.

Variables induces the same MMT/OMDoc terms as symbols do, except for the head of the term being `<OMV name="..."/>` instead of `<OMS/>`.

`\varnotation`

`\varnotation{⟨variable⟩}[⟨options⟩]{⟨notation⟩}`

Takes the exact same argumens as `\notation`, but attaches an additional notation to the *variable* ⟨*variable*⟩ rather than a symbol.

`\svar`

`\svar[⟨name⟩]{⟨text⟩}`

Semantically marks up ⟨*text*⟩ as representing a variable ⟨*name*⟩. The variable does not need to have been defined prior. If no ⟨*name*⟩ is givenm ⟨*text*⟩ will be used as the name.

This is useful in situations like "throwaway expressions" or remarks; e.g.

```
$\plus{\svar{n},\svar{m}}$ means...
```

`\varseq`

`\varseq{⟨mname⟩}[⟨options⟩]{⟨range⟩}{⟨notation⟩}`

Declares a new variable sequence. The ⟨*options*⟩ are the same as for `\vardef`. If not provided, `args=1` by default (a 0-ary sequence would just be a normal variable).

A type (given as `type=`) is interpreted to be the type of every element a_i of the sequence $a_1, ..., a_n$ (not of the sequence itself). If the type is itself a sequence $A_1, ..., A_n$, the assumption is that its range is the same as the one of the new sequence, and the type of every a_i in the sequence is A_i.

⟨*range*⟩ needs to be a comma-separated sequence of either `args` many arguments, or `\ellipses`.

The resulting semantic macro is allowed anywhere sTEX expects an argument mode `a` or `B` argument.

\ellipses Represents ellipses in a range; produces \ellipses in math mode.

\seqmap

\seqmap{⟨code⟩}{⟨sequence⟩}

Maps the function ⟨code⟩ (containing #1) over every element of the ⟨sequence⟩.

Is allowed anywhere sTeX expects an argument mode a or B argument.

3.7 Structures

Mathematical structure bundle interdependent symbols together.

mathstructure (env.) \begin{mathstructure}{⟨mname⟩}[⟨name⟩,this=⟨code⟩] opens a new mathematical structure with name ⟨mname⟩ (if provided) or ⟨mname⟩ (otherwise), and semantic macro \mname. It subsequently behaves like the smodule environment.

\this The optional this=⟨code⟩ option allows for setting the typesetting of the \this macro within a mathstructure. In particular, \this can be used in notations for symbols declared in the structure. \this can be though of as representing "*the*" (current) instance of this structure.

extstructure (env.) \begin{extstructure}{⟨mname⟩}[⟨name⟩,this=⟨code⟩]{⟨structs⟩} opens a new mathematical structure extending the structures given in ⟨structs⟩ (a comma-separated list of names).

extstructure* (env.) \begin{extstructure*}{⟨struct⟩} opens a new mathematical structure *conservatively* extending the (single) structure ⟨struct⟩. *Conservative* meaning: Every symbol newly introduced in this structure needs to have a *definiens*. The new symbols are attached as fields directly to ⟨struct⟩.

\usestructure The \usestructure macro behaves like \usemodule for mathematical structures, making the symbols available to use directly.

↪M→ —M→ ~T⇝ mathstructure make use of the *Theories-as-Types* paradigm (see [?]): \begin{mathstructure}{⟨name⟩} creates a nested theory with name ⟨name⟩-module. The constant ⟨name⟩ is defined as a *dependent record type with manifest fields*, the fields of which are generated from (and correspond to) the constants in ⟨name⟩-module.

3.7.1 Semantic Macros for Structures

Assume we have a mathematical structure with semantic macro *\struct*:

Example 2

```
\begin{mathstructure}{struct}
  \symdef{fielda}{a}
  \symdef{fieldb}[args=2]{#1 \maincomp{b} #2}
  \symdef{fieldc}[args=2,def=\sn{fieldb}]{#1 \maincomp{c} #2}
  \inlineass[name=axiom1]{\conclusion{some axiom}}
```

```
\end{mathstructure}
\notation{struct}{StRuCt}
```

- If `\struct` has no notations, then `$\struct!$` produces $\langle a, b, c\rangle$. Otherwise, it produces the notation, i.e. *StRuCt*. In both cases, `$\struct{}{}$` produces $\langle a, b, c\rangle$ (for reasons that will become clearer in a moment).
- `$\struct{}{}$` (or `$\struct!$` in the case where no notations are around) can be modified in the following ways:
  - `$\struct{}{}[`⟨*fieldname*⟩`,...]$` lets you pick, which precise fields to show, so e.g. `$\struct{}{}[fielda,fieldb]$` produces $\langle a, b\rangle$. By default, `$\struct{}{}$` shows exactly the fields that have semantic macros (which are also used to access the fields in `$\struct{...}{fieldname}`).
  - `$\struct{}{}[`⟨*id*⟩`]$` lets you pick the notation of the "mathematical structure" symbol to use to typeset the structure; e.g. `$\struct{}{}[parens]` yields (a, b, c), and (combining both) `$\struct{}{}[fielda,fieldb][angle]` yields $\langle a, b\rangle$.
- The two arguments in `$\struct{first}{second}$` represent 1. the term that is to be treated as an instance of `\struct`, and 2. the precise field to invoke. If the first is empty, then there is no instance. If the second is empty, sTEX will present all of them (that are not assertions). Hence, `$\struct{S}{}$` yields $\langle a_S, b_S, c_S\rangle$, `$\struct{S}{fielda}$` produces a_S, `$\struct{}{fielda}$` produces a, and `$\struct{S}{fieldb}{x}{y}$` produces xb_Sy.
- For the sake of completion, `$\struct{first}!$` simply produces the given argument; e.g. `$\struct{S}!$` simply produces S.

More precisely: `$\struct{`⟨*code*⟩`}$` acts like a "type coercion" of ⟨*code*⟩ to be an instance of `\struct`.

Of course, it is (occasionally, but) rarely useful to use the semantic macro `\struct` by giving it two arguments *manually*; but this is what sTEX does when using `\struct` in the `return` of a symbol (or variable).

Continuing:

- `$\struct[comp=`⟨*code*⟩`]{...}{...}$` applies ⟨*code*⟩ (using `#1`) to every occurence of `\maincomp` in the notations of the fields, as a replacement for the default modification `{#1}_{\this}`. For example, `$\struct[comp={#1}^{Foo}]{S}{}$` produces $\langle a^{Foo}, b^{Foo}, c^{Foo}\rangle$, and `$\struct[comp={#1}^{\this}]{S}{fieldb}{x}{y}$` yields xb^Sy.
- `$\struct[this=`⟨*code*⟩`]{...}{...}$` modifies the way `\this` is being typeset; i.e. the presentation of the first `{...}` argument. For example `$\struct[this=T]{S}{}$` produces $\langle a, b, c\rangle$ – since `\this` is not being used in the notations of the fields. Note that the `this=` and `comp=` variants are (as of yet) mutually incompatible.
- Finally, `$\struct[`⟨*fieldname*⟩`=`⟨*code*⟩`]{...}{...}$` *assigns* the field ⟨*fieldname*⟩ to the term ⟨*code*⟩. ⟨*code*⟩ is subsequently used when using `{...}{}`, but not in fields directly. For example, `$\struct[fielda=A]{S}{}$` produces $\langle A!, b_S, c_S\rangle$, but `$\struct[fielda=A]{S}{fielda}$` produces a_S.

Note the insertion of `!` behind the `A` – this is to make sure that assignments to semantic macros that takes arguments don't accidentally eat more than they should.

Also note that multiple assignments can be done in the same pair of `[]`, or chained – i.e. both `$\struct[fielda=A,fieldb=B]...$` and `$\struct[fielda=A][fieldb=B]...$` are valid and equivalent. `$\struct[fielda=A,fielda=B]...$` however is not – every field may be assigned at most once.

Chapter 4

Statements

sTeX provides four environments to semantically annotate various kinds of statements:

sdefinition (*env.*) The **sdefinition** environment represents (primarily mathematical) *definitions*; in particular for symbols. The contents of the environment will be used as *documentation* for any symbol that either orrurs as a \definiendum (or \definame) within the **sdefinition**, or that is listed in the optional for= argument of the environment.

If a \definiens occurs, this will be used by Mmt as the formal definiens for the respective symbol.

sassertion (*env.*) The **sassertion** environment represents *assertions*, i.e. propositions such as *theorems*, *lemmata*, *axioms*, etc. If a \conclusion occurs within the **sassertion**, its argument will be used as the formal *statement* of the assertion.

sexample (*env.*) The **sexample** environment represents examples (or counterexamples).

sparagraph (*env.*) The **sparagraph** environment represents all other kinds of (logical) paragraphs, such as remarks, comments, transitions bwetween topics, recaps, reminders, etc.

All of these take the same arguments:

- for=⟨*csl*⟩: a comma-separated list of symbols.
- The same optional arguments as \symdecl, with macro= replacing the name of the semantic macro. All of them are only relevant, if either name= or macro= are provided.

  As with \symdecl, if no name is given, but macro is, then the same name is used for both the semantic macro and the symbol itself.

  If name is given but macro isn't, no semantic macro is generated. Subsequently, the newly generated symbol is added the for-list.
- style: see chapter 5.
- title: a title to use in various styles (see chapter 5).
- id: a label to use for \sref.

`\inlineass` `\inlinedef` `\inlineex`

The macros `\inlineass`, `\inlinedef` and `\inlineex` behave like the `sassertion`, `sdefinition` and `sexample` environments respectively, but take the text to annotate as an argument, rather than as the body of an environment, and do not break paragraphs.

The same macros available in the environments are also available in the argument of the `\inline*` macros.

`\varbind`

`\varbind{⟨cls⟩}`

retroactively attaches the `bind` option to every variable provided (as a comma-separated list).

4.1 More on Definitions

In `sdefinition` (and `sparagraph` with `style=symdoc`), the following additional macros are available:

`\definiendum` `\definame` `\Definame` `\defnotation`

The `\definiendum` macro behaves largely like `\symref`, but it uses a dedicated highlighting for *definienda* and adds the referenced symbol to the `for=` list of the environment.

`\definame` is to `\definiendum` as `\symname` is to `\symref`. Analogously, `\Definame` behaves like `\Symname`.

`\defnotation` can be used in math mode to apply the `\definiendum` highlighting to notations.

`\definiens`

The `\definiens` macro can be used to semantically annotate the *definiens* in a `sdefinition`.

If the `sdefinition` environment has several elements in its `for` list, an optional argument `\definiens[⟨symbol⟩]{...}` can be used to tell sTeX which symbol's definiens this is. By default, the *first* symbol in the `for` list is used.

> Here is how MMT will treat the fragment marked up with `\definiens`:
> Firstly, it will attempt to translate its contents into an MMT/OMDoc term. This succeeds easily if `\definiens` is some semantic macro (applied to arguments).
> Secondly, it will check for all variables currently in scope, that were defined with the optional argument `bind`. It then will check, whether a symbol is in scope, that has `role=lambda`. If so, it will use that `lambda` symbol to bind all these variables (in the order in which they were defined) in the term. If no `lambda` symbol is found, it will use the bind symbol that ships with sTeX.
> The final term will be attached as definiens to the corresponding MMT constant, if it was declared in the same module as the `\definiens` occurrence.

4.2 More on Assertions

`\premise` `\conclusion`

The `\conclusion` macro can be used to mark up the actual statement within an `sassertion`. The `\premise` macro can be used to additionally mark up *premises*.

If the `sassertion` environment has several elements in its `for` list, an optional argument `\conclusion[⟨symbol⟩]{...}` can be used to tell sTeX which symbol's statement this is. By default, the *first* symbol in the `for` list is used.

Here is how MMT will treat the fragments marked up with `\conclusion` and `\premise`:
Firstly, it will attempt to translate the contents of `\conclusion` into an MMT/OMDoc term c. This succeeds easily if the `\conclusion` is some semantic macro (applied to arguments).
Secondly, it will collect all fragments marked up with `\premise` and do the same to them $(p_1, ..., p_n)$.
It will then check, whether a symbol is in scope, that has `role=implication`. If so, it will use that `implication` symbol to attach all the premises to the conclusion, resulting in $t := \texttt{imply}(p_1, ..., p_n, c)$.
Next, it will check for all variables currently in scope, that were defined with the optional argument `bind`. It then will check, whether a symbol is in scope, that has `role=forall`. If so, it will use that `forall` symbol to bind all these variables (in the order in which they were defined) in the term t.
Finally, it will check, whether a symbol is in scope, that has `role=judgment`. If so, it will set $t := \texttt{judgment}(t)$.
If no `forall` symbol is found, it will first apply the `judgment` symbol (if existent) and then use the bind symbol that ships with sTeX to bind the variables.
The final term will be attached as type to the corresponding MMT constant, if it was declared in the same module as the `\definiens` occurrence.

`sproof` (*env.*)

TODO[4]

[4]TODO: proofs

Chapter 5

Customizing Typesetting

There are two kinds of typesetting that can be customized in sTeX: symbol references (notation components, \symref, variables, etc.) are highlighted using a small set of macros that can be simply redefined by authors.

Other macros and environments usually have more complicated “typesetting rules” associated with them – often in the form of other already existing environments that should be used.

Lastly, in HTML we can provide custom CSS rules in math archives that determine the styling of certain environments, so that the actual presentation depends on the document in which the fragments are included (e.g. via \inputref), rather than the file the fragment is implemented in.

It is generally recommended to implement these customizations in a preamble in the `lib` directory (see section 1.3)

5.1 Highlighting Symbol References

\symrefemph
\symrefemph@uri

\symrefemph governs how references via \symref (or \symname, or theirt short variants) are highlighted;

Doing \symref{⟨*symbol*⟩}{⟨*text*⟩} ultimately expands to \symrefemph@uri{⟨*text*⟩}{⟨*symbol URI*⟩}, the default implementation of which is just \symrefemph{⟨*text*⟩}. The default implementation of \symrefemph, in turn, is just \emph{⟨*text*⟩}.

If you only want to change e.g. the color of \symrefs, you only need to redefine \symrefemph, e.g. using

```
\renewcommand\symrefemph[1]{\textcolor{red}{#1}}
```

the `@uri` variant is useful if you want to link somewhere, or show the URI in a tooltip. The stex-highlighting package does both, using:

```
\usepackage{pdfcomment}
\protected\def\symrefemph@uri#1#2{%
  \pdftooltip{%
    \srefsymuri{#2}{\symrefemph{#1}}%
  }{%
    URI:~\detokenize{#2}%
  }%
```

```
}
\def\symrefemph#1{%
  \ifcsname textcolor\endcsname
    \textcolor{teal}{#1}%
  \else#1\fi
}
```

`\compemph`
`\compemph@uri`

Like `\symrefemph` and `\symrefemph@uri`, but governs the highlighting of components (marked with `\comp` or `\maincomp`) in notations.

`\defemph`
`\defemph@uri`

Like `\symrefemph` and `\symrefemph@uri`, but governs the highlighting of definienda marked with `\definiendum` (or `\definame`).

`\varemph`
`\varemph@uri`

Like `\compemph` and `\compemph@uri`, but governs the highlighting of components (marked with `\comp` or `\maincomp`) in the notations of variables.

The second argument to `\varemph@uri` is the *name* of the variable.

5.2 Styling Environments and Macros

A variety of environments and macros provided by sTeX are *stylable* using the macros `\stexstyle`⟨*name*⟩`[`⟨*style*⟩`]{...}`. These stylable environments and macros bind various of their parameters to macros `\this`⟨*parameter*⟩, which can then be used in the styles.

For example, if we have a `definition` environment that we would want to use to style our `sdefinition`s, we can do (in the simplest case)

```
\stexstyledefinition{\begin{definition}}{\end{definition}}
```

This tells sTeX to insert `\begin{definition}` at the beginning of every `sdefinition` environment, and `\end{definition}` at the end.

If we have a environments `theorem` and `lemma`, we probably want the `sassertion` environment to use those for theorems and lemma. We can achieve that by doing

```
\stexstyleassertion[theorem]{\begin{theorem}}{\end{theorem}}
\stexstyleassertion[lemma]{\begin{lemma}}{\end{lemma}}
```

Now if we do `\begin{sassertion}[style=theorem]`, it will wrap the environment with `\begin{theorem}...\end{theorem}`.

Of course, many such statements might have a title, as e.g. in

Theorem 5.2.1 (Gödel's First Incompleteness Theorem)**.** ...

In `sasssertion`, we can provide that title as optional argument using `title=...`. Before calling the styling provided, `sassertion` will store that title in the macro `\thistitle`, which we can use in the styling. For example, we might prefer to pass it on to the `theorem` environment:

```
\stexstyleassertion[theorem]{\ifx\thistitle\@empty
  \begin{theorem}\else\begin{theorem}[\thistitle]\fi}
  {\end{theorem}}
```

\stexstylemodule
\stexstylecopymodule
\stexstyleinterpretmodule
\stexstylerealization
\stexstylemathstructure
\stexstyleextstructure
\stexstyledefinition
\stexstyleassertion
\stexstyleexample
\stexstyleparagraph
\stexstyleproof
\stexstylesubproof

TODO[5]

Additionally, we can style certain macros, if we want them to produce output. For example, we might (for debuggin or documentation purposes) \symdecl to give a short summary of the symbol.

We can achieve that by doing, for example:

```
\stexstylesymdecl[debug]{
  Symbol \thisdeclname~(with arity \thisargs) of type $\thistype$.
}
```

in which case

```
\symdecl{foo}[args=2,type=\mathbb{N},style=debug]
```

will yield

> Symbol foo (with arity ii) of type $\mathbb{N}$.

\stexstyleusemodule
\stexstyleimportmodule
\stexstylerequiremodule
\stexstyleassign
\stexstylerenamedecl
\stexstyleassignMorphism
\stexstylecopymod
\stexstyleinterpretmod
\stexstylerealize
\stexstylesymdecl
\stexstyletextsymdecl
\stexstylenotation
\stexstylevarnotation
\stexstylesymdef
\stexstylevardef
\stexstylevarseq
\stexstylespfsketch
\stexstyleMMTinclude

TODO[6]

5.3 Custom CSS for Environments