The sTeX3 Manual *

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

2022-09-12

Abstract

sTeX is a collection of LaTeX packages that allow to markup documents semantically without leaving the document format.

Running 'pdflatex' over sTeX-annotated documents formats them into normal-looking PDF. But sTeX also comes with a conversion pipeline into semantically annotated HTML5, which can host semantic added-value services that make the documents active (i.e. interactive and user-adaptive) and 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, and

- 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 3.2 (last revised 2022-09-12)

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 easily be skipped on a first read.

Boxes like this one explain how some sTEX concept relates to the Mmt/OMDoc system, philosophy or language; see [MMT; Koh06] for introductions.

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 collection to use semantic annotations in LaTeX documents,

- RusTeX [RT] to convert `tex` sources to (semantically enriched) `xhtml`,

- The Mmt system [MMT], that extracts semantic information from the thus generated `xhtml` and provides semantically informed added value services. Notably, Mmt integrates the RusTeX system already.

Chapter 2

Setup

There are two ways of using sTeX: as a

1. way of writing LaTeX more modularly (object-oriented Math) for creating PDF documents or

2. foundation for authoring active documents in HTML5 instrumented with knowledge management services.

Both are legitimate and useful. The first requires a significantly smaller tool-chain, so we describe it first. The second requires a much more substantial toolchain of knowledge management systems.

Luckily, the sTeX-IDE will take care of much of the setup required for the full toolchain, if you are willing to use it.

2.1 Setting up the sTeX Package

2.1.1 Minimal Setup for the sTeX Package

In the best of all worlds, there is no setup, as you already have a new version of TeXLive on your system as a LaTeX enthusiast. If not now is the time to install it; see [TL]. You can usually update TeXLive via a package manager or the TeXLive manager **tlmgr**. sTeX requires a TeX kernel newer than February 2022.

Alternatively, you can install sTeX from CTAN, the Comprehensive TeX Archive Network; see [ST] for details. We assume you have the sTeX package in at least version 3.2 (September 2022).

2.1.2 GIT-based Setup for the sTeX Development Version

If you want use the latest and greatest sTeX packages that have not even been released to CTAN, then you can directly clone them from the sTeX development repository [sTeX] by the following command-line instructions:

```
cd <stexdir>
git clone https://github.com/slatex/sTeX.git
```

and keep it updated by pulling updates via `git pull` in the cloned sTeX directory. Make sure to either clone the sTeX repository into a local texmf-tree or to update your `TEXINPUTS` environment variable, e.g. by placing the following line in your `.bashrc`:

```
        export TEXINPUTS="$(TEXINPUTS):<sTeXDIR>//:"
```

2.1.3   Setting your MathHub Directory

One of sTeX's features is a proper *module system* of interconnected document snippets for mathematical content. Analogously to *object-oriented programming*, it allows for "object-oriented mathematics" via individual combinable and, importantly, *reusable* modules, developed collaboratively.

To make use of such modules, the sTeX system needs to be told where to find them. There are several ways to do so (see subsection 5.2.1), but the most convenient way to do so is via a system variable.

To do so, create a directory `MathHub` somewhere on your local file system and set the environment variable `MATHHUB` to the file path to that directory.

In linux, you can do so by writing

```
        export MATHHUB="/path/to/your/MathHub"
```

in your `~/.profile` (for all shells) or `~/.bashrc` (for the bash terminal only) file.

2.2   Setting up the sTeX IDE

The sTeX IDE consists of two components using the *Language Server Protocol (LSP)*: A *client* in the form of a VSCode extension, and a *server* included in the Mmt system. Installing the extension will open up a setup routine that will guide you through the rest.

2.2.1   The sTeX VSCode Extension

If you have not already, you should first install the VSCode editor available at https://code.visualstudio.com/.

Next, open VSCode and install the sTeX extension by clicking on the *extensions* menu on the very left of the VSCode window and searching for "sTeX" in the *"Search Extensions in Marketplace"* field, as in Figure 1, and clicking the *Install*-button of the sTeX extension by KWARC.

2.2.2   Setting up Mmt

Next, open any directory (`File → Open Folder...`) that contains a `.tex`-file, and a setup window as in Figure 2 will pop up. Clik on the highlighted link '*here*' and download the latest version of the `MMT.jar` file (at least version 23.0.0) anywhere you like. Then click the *"Browse..."*-button and select your freshly downloaded `MMT.jar`.

If you have already set a system variable for your MathHub-directory, you are now done and can click *"Finish"*. If you have not, you can now also enter a directory path in the lower text field, and the VSCode extension will attempt to globally set one up for you, depending on your operating system.

Once you click *"Finish"*, the client will connect to https://stexmmt.mathhub.info/:sTeX, query for available archives, download the core libraries required for all (or most) semantic services (`MMT/urtheories` and `sTeX/meta-inf`) and set up RusTeX for you automatically.

Figure 1: Installing the sTeX extension for VSCode

Figure 2: sTeX Setup Routine

2.3 Manual Setup

In lieu of using the SᴛEX IDE, we can do the following:

2.3.1 SᴛEX Archives (Manual Setup)

Writing semantically annotated SᴛEX becomes much easier, if we can use well-designed libraries of already annotated content. SᴛEX provides such libraries as SᴛEX archives – i.e. GIT repositories at https://gl.mathhub.info – most prominently the SMGLoM libraries at https://gl.mathhub.info/smglom.

To do so, we set up a **local MathHub** by creating a MathHub directory `<mhdir>`. Every SᴛEX archive as an **archive path `<apath>`** and a name `<archive>`. We can clone the SᴛEX archive by the following command-line instructions:

```
cd <mhdir>/<apath>
git clone https://gl.mathhub.info/smglom/<archive>.git
```

Note that SᴛEX archives often depend on other archives, thus you should be prepared to clone these as well – e.g. if `pdflatex` reports missing files. To make sure that SᴛEX 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 5.2).

```
export MATHHUB="<mhdir>"
```

2.3.2 Manual Setup for Active Documents and Knowledge Management Services

Foregoing on the SᴛEX IDE, we will need several additional (on top of the minimal setup above) pieces of software; namely:

- **The Mmt System** available here. 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 SᴛEX/Mmt content archives.

- **SᴛEX 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) SᴛEX 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.

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

Chapter 3

The sTeX IDE

Chapter 4

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*:

```
 1 \documentclass{article}
 2 \usepackage{stex,xcolor,stexthm}
 3
 4 \begin{document}
 5 \begin{smodule}{GeometricSeries}
 6     \importmodule[smglom/calculus]{series}
 7     \importmodule[smglom/arithmetics]{realarith}
 8
 9     \symdef{geometricSeries}[name=geometric-series]{\comp{S}}
10
11     \begin{sdefinition}[for=geometricSeries]
12         The \definame{geometricSeries} is the \symname{series}
13         \[\defeq{\geometricSeries}{\definiens{
14             \infinitesum{\svar{n}}{1}{
15                 \realdivide[frac]{1}{
16                     \realpower{2}{\svar{n}}
17             }}
18         }}.\]
19     \end{sdefinition}
20
21     \begin{sassertion}[name=geometricSeriesConverges,type=theorem]
22     The \symname{geometricSeries} \symname{converges} towards $1$.
23     \end{sassertion}
24 \end{smodule}
25 \end{document}
```

Compiling this document with **pdflatex** should yield the output

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

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

8

> **Theorem 0.2.** The geometric series converges towards 1.

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 4.0.1:

> 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, see section 7.3.

Let's investigate this document in detail to understand the respective parts of the SᴛEX markup infrastructure:

smodule (*env.*)
```
\begin{smodule}{GeometricSeries}
...
\end{smodule}
```
First, we open a new *module* called `GeometricSeries`. The main purpose of the `smodule` environment is to group the contents and associate it with a *globally unique* identifier (URI), which is computed from the name `GeometricSeries` and the document context.

(Depending on your pdf viewer), the URI should pop up in a tooltip if you hover over the word **geometric series**.

\importmodule
```
\importmodule[smglom/calculus]{series}
\importmodule[smglom/arithmetics]{realarith}
```
Next, we *import* two modules – `series` from the SᴛEX archive `smglom/calculus`, and `realarith` from the SᴛEX archive `smglom/arithmetics`. 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 SᴛEX symbols and associated semantic macros (e.g. *\infinitesum*, *\realdivide*, *\realpower*) in the imported module available to the current module `GeometricSeries`. The module `GeometricSeries` "exports" all of these symbols to all modules imports it via an `\importmodule{GeometricSeries}` instruction. Additionally it exports the local symbol *\geometricSeries*.

\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
```
\symdef{GeometricSeries}[name=geometric-series]{\comp{S}}
```
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 the `amsthm` package.

\symname
```
... is the \symname{?series}
```
The `\symname`-command prints the name of a symbol, highlights it (based on customizable settings) and associates the text printed with the corresponding symbol.

Note that the argument of `\symref` can be an imported symbol (here the `series` symbol is imported from the `series` module). sTeX tries to determine the full symbol URI from the argument. If there are name clashes in or with the imported symbols, the name of the exporting module can be prepended to the symbol name before the `?` character.

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. `\symref` takes two arguments: the first ist the symbol name (or macro name), and the second a variant verbalization of the symbol, e.g. an inflection variant, a different language or a synonym. In our example `\symname{?series}` abbreviates `\symref{?series}{series}`.

\definame
\definiendum
```
The \definame{geometricSeries} ...
```
The `sdefinition`-environment provides two additional macros, `\definame` and `\definiendum` which behave similarly 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.

4.1   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` while preserving all the sTEX markup in the result.

TODO VSCode Plugin

Using RusTEX [RT], 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 additional 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>
    <mrow resource="1" property="stex:arg">
     <mi resource="var://n" property="stex:OMV">n</mi>
    </mrow>
    <mo resource="...?series?infinitesum" property="stex:comp">=</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">
     <msup resource="...realarith?exponentiation" property="stex:OMA">
```

---

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

```
      <mi resource="1" property="stex:arg">2</mi>
      <mrow resource="2" property="stex:arg">
       <mi resource="var://n" property="stex:OMV">n</mi>
      </mrow>
     </msup>
    </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>
```

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

Remark 4.1.1:

> 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.

4.2 Mmt/OMDOC Conversion

Another way to convert our document to *actual* MMT/OMDOC is to put it in an STEX **archive** (see section 5.2) and have MMT take care of everything.

Assuming the above file is `source/demo.tex` in an STEX archive `MyTest`, you can run MMT and do `build MyTest stex-omdoc demo.tex` to convert the document to both `xhtml` (which you will find in `xhtml/demo.xhtml` in the archive) and formal MMT/OMDOC, which you can subsequently view in the MMT browser (see https://

uniformal.github.io//doc/applications/server.html#the-mmt-web-site for details).

Chapter 5

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.

writesms (⟨*boolean*⟩) with this package option, sTeX will write the contents of all external modules imported via `\importmodule` or `\usemodule` into a file `\jobname.sms` (analogously to the table of contents `.toc`-file).

usems (⟨*boolean*⟩) subsequently tells sTeX to read the generated sms-file at the beginning of the document. This allows for e.g. collaborating on documents without all authors having to have all used archives and modules available – one author can load the modules with `writesms`, and the rest can use the the modules with `usesms`. Furthermore, the sms file can be submitted alongside a `tex`-file, effectively making it "standalone".

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.

5.1 How Knowledge is Organized in sTeX

sTeX content is organized on multiple levels:

1. sTeX **archives** (see section 5.2) contain individual `.tex`-files.

2. These may contain sTeX **modules**, introduced via `\begin{smodule}{ModuleName}`.

3. 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.

4. sTeX **expressions** finally are built up from usages of semantic macros.

> ↪M↣
> ─M→
> ⤳T↝
>
> - 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 **include**s and other *theory morphisms*, thus giving rise to a *theory graph* in the OMDoc sense [RK13].
> - 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 abstract syntax (and XML encoding) of OPENMATH [Bus+04].

5.2   sTeX Archives

5.2.1   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 four 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. Finally, if all else 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.

5.2.2 The Structure of ᴽᴛᴇX Archives

An ᴽᴛᴇX archive `group/name` is 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 ᴽᴛᴇX 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 ᴽᴛᴇX 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 the following additional directory structure in the `source`-folder of an ᴽᴛᴇX archive:

- `/source/mod/` – individual ᴽᴛᴇX modules, containing symbol declarations, notations, and `\begin{sparagraph}[type=symdoc,for=...]` environments for "encyclopaedic" 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.

5.2.3 MANIFEST.MF-Files

The `MANIFEST.MF` in the `META-INF`-directory consists of key-value-pairs, informing ᴽᴛᴇX (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`.

5.2.4 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, e.g. for lazy loading.

In the majority of practical cases `\inputref` is likely to be preferred over `\mhinput` because it leads to less duplication in the generated `xhtml`.

`\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. Typical invocations are

- `\addmhbibresource{lib/refs.bib}`, which specifies a bibliography in the `lib` folder in the local archive or

- `\addmhbibresource[HW/meta-inf]{lib/refs.bib}` in another.

\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`.

    \libinput 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.

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

> **Remark 5.2.1:**
>
> A good practice is to have individual SТEX fragments follow basically this document frame:
>
> ```
> 1 \documentclass{stex}
> 2 \libinput{preamble}
> 3 \begin{document}
> 4    ...
> 5    \ifinputref \else \libinput{postamble} \fi
> 6 \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.
>
>   \libusepackage is particularly useful in `preamble.tex` when we want to use custom packages that are not part of TEXLive. In this case we commit the respective packages in one of the `lib` folders and use \libusepackage to load them.

5.3   Module, Symbol and Notation Declarations

5.3.1   The `smodule`-Environment

smodule (*env.*)   A new module is declared using the basic syntax

$$\text{\textbackslash begin\{smodule\}[options]\{ModuleName\}...\textbackslash end\{smodule\}.}$$

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.

type ($\langle$*string*$\rangle*$) for use in customizations.

deprecate ($\langle$*module*$\rangle$) if set, will throw a warning when loaded, urging to use $\langle$*module*$\rangle$ instead.

id ($\langle$*string*$\rangle$) for cross-referencing.

ns ($\langle$*URI*$\rangle$) 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 ($\langle$*language*$\rangle$) if not set, computed from the current file name (e.g. `foo.en.tex`).

sig ($\langle$*language*$\rangle$) 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 ($\langle$*string*$\rangle*$) names of the creators.

contributors ($\langle$*string*$\rangle*$) names of contributors.

srccite ($\langle$*string*$\rangle$) a source citation for the content of this module.

> ↪M→ An SₜEX module corresponds to an Mᴍᴛ/OMDᴏᴄ *theory*. As such it
> —M→ gets assigned a module URI (*universal resource identifier*) of the form
> ⤳T⤳ `<namespace>?<module-name>`.

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

Example 1
Input:

```
1 \begin{smodule}[title={This is Some Module}]{SomeModule}
2     Hello World
3 \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:

```
1 \stexpatchmodule[display]
2   {\textbf{Module (\smoduletitle)}\par}
3   {\par\noindent\textbf{End of Module (\smoduletitle)}}}
4
5 \begin{smodule}[type=display,title={Some New Module}]{SomeModule2}
6     Hello World
7 \end{smodule}
```

Output:

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

.

5.3.2 Declaring New Symbols and Notations

Inside an `smodule` environment, we can declare new SᴛᴇX 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.

> ↪M→ `\symdecl` introduces a new OMDᴏᴄ/Mᴍᴛ constant in the current mod-
> —M→ ule (=OMDᴏᴄ/Mᴍᴛ theory). Correspondingly, they get assigned the URI
> ↝T↝ `<module-URI>?<constant-name>`.

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

Example 3
Input:

```
1 \symdecl*{foo}
2 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:

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

Output:

> this is a symbol taking two arguments.

.

So far we have gained exactly . . . nothing by adding the arity information: we cannot do anything with the arguments in the text.

We will now see what we can gain with more machinery.

`\notation` 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` For many semantic services e.g. semantic highlighting or **wikification** (linking user-visible notation components to the definition of the respective symbol they come from), we need to specify the notation components. Unfortunately, there is currently no way the sTeX engine can infer this by itself, so we have to specify it manually in the notation specification. 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 meaningful 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:
>
> ```
> 1 \symdef{newbinarysymbol}[hl,args=2]
> 2     {\comp{\text{1.: }}#1\comp{\text{; 2.: }}#2}
> 3 $\newbinarysymbol{a}{b}$
> ```
>
> 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.

But one man's meat is another man's poison: it is very subjective what the "default notation" of an operator should be. Different communities have different practices. For instance, the complex unit is written as i in Mathematics and as j in electrical engineering. So to allow modular specification and facilitate re-use of document fragments sTeX allows to re-set notation defaults.

\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}`.

\textsymdecl In the less mathematical settings where we want a symbol and semantic macro for some concept with a notation *beyond* its mere name, but which should also be available in TeX's text mode, the command `\textsymdecl` is useful. For example, we can declare a symbol `openmath` with the notation `\textsc{OpenMath}` using `\textsymdecl{openmath}[name=OpenMath]{\textsc{OpenMath}}`. The `\openmath` yields OPENMATH both in text and math mode.

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:

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

Output:

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

.

> ↪M→
> —M→ `\symbolname!` is translated to OMDoc/Mmt as `<OMS name="...?symbolname"/>`
> ↝T↝ directly.

5.3.3 Argument Modes

The notations so far used *simple* arguments which we call *mode-`i`* 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 mode-`i` arguments. However, there are three more argument modes which we will investigate now, namely mode-`b`, mode-`a` and mode-`B` arguments.

Mode-`b` Arguments

A mode-`b` 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.

> ↪M→ Mode-`b` arguments behave exactly like mode-`i` arguments within TeX, but appli-
> —M→ cations of binding operators, i.e. symbols with mode-`b` arguments, are translated
> ↝T↝ to `OMBIND`-terms in OMDoc/Mmt, rather than `OMA`.

For 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.

Mode-a Arguments

Mode-`a` 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. Mode-`a` 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 mode-`a` 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 mode-`a` 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:

```
1 \symdef{ascendingchain}[args=iai]
2   {\comp{\forall} #2\comp{.\,}#3}
3   {##1 \comp{<}_{#1} ##2}
4
5 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:
>
> ```
> 1   \symdef{addition}[args=a]{#1}{##1 \comp{+} ##2}
> 2
> 3 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, associative 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 $a \neq b \wedge a \neq c \wedge a \neq d \wedge b \neq c \wedge b \neq d \wedge c \neq d$

As before, at the PDF level, this annotation is invisible (and without effect), but at the level of the generated OMDoc/MMT this leads to more semantical expressions.

Mode-B Arguments

Finally, mode-B 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:

```
1 \symdef{quantforall}[args=Bi]
2   {\comp{\forall}#1\comp{.}#2}
3   {##1\comp,##2}
4
5 $\quantforall{\svar{x},\svar{y},\svar{z}}{P}$
```

Output:

$\forall x,y,z.P$

.

5.3.4 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!)

> ↪M→ The `type` and `def` keys correspond to the `type` and `definiens` components of OMDoc/Mmt constants.
> —M→ Correspondingly, the name "type" should be taken with a grain of salt, since
> ～T↝ 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 SLTeX symbols), e.g. for addition on natural numbers:

Example 13

Input:

```
1 \symdef{Nat}[type=\set]{\comp{\mathbb N}}
2 \symdef{addition}[
3     type=\funtype{\Nat,\Nat}{\Nat},
4     op=+,
5     args=a
6 ]{#1}{##1 \comp+ ##2}
7
8 \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:

```
1 \symdef{successor}[
2     type=\funtype{\Nat}{\Nat},
3     def=\fun{\svar{x}}{\addition{\svar{x},1}},
4     op=\mathtt{succ},
5     args=1
6 ]{\comp{\mathtt{succ(}#1\comp{)}}}}
7
8 The \symname{successor} operation $\funtype{\Nat}{\Nat}$
9 is defined as $\fun{\svar{x}}{\addition{\svar{x},1}}$
```

Output:

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

.

5.3.5 Precedences and Automated Bracketing

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

Example 15

Input:

```
1 \symdef{multiplication}[
2     type=\funtype{\Nat,\Nat}{\Nat},
3     op=\cdot,
4     args=a
5 ]{#1}{##1 \comp\cdot ##2}
6
7 \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:

```
1 $\addition{a,\multiplication{b,\addition{c,\multiplication{d,e}}}}$
```

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:

```
1 $\addition{a,\multiplication{b,(\addition{c,\multiplication{d,e}})}}$
```

Output:

```
a+b·(c+d·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 perfectly 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:

```
 1 \notation{multiplication}[
 2     op=\cdot,
 3     prec=50
 4 ]{#1}{##1 \comp\cdot ##2}
 5 \notation{addition}[
 6     op=+,
 7     prec=100
 8 ]{#1}{##1 \comp+ ##2}
 9
10 $\addition{a,\multiplication{b,\addition{c,\multiplication{d,e}}}}$
```

Output:

```
a+b·(c+d·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, e.g. for bracket-like notations such as intervals – for those purposes, `\infprec` and `\neginfprec` exist (which are implemented as the maximal and minimal integer values accordingly).g

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.

5.3.6 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.

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:

```
1 \vardef{varf}[
2    name=f,
3    type=\funtype{\Nat}{\Nat},
4    op=f,
5    args=1,
6    prec=0;\neginfprec
7 ]{\comp{f}#1}
8 \vardef{varn}[name=n,type=\Nat]{\comp{n}}
9 \vardef{varx}[name=x,type=\Nat]{\comp{x}}
10
11 Given a function $\varf!:\funtype{\Nat}{\Nat}$,
12 by $\addition{\varf!,\varn}$ we mean the function\rustexBREAK
13 $\fun{\varx}{\varf{\addition{\varx,\varn}}}$
```

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

5.3.7 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.

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. Note that both the starting as well as the ending index may be variables.

This is best shown by example:

Example 20

Input:

```
1 \vardef{varn}[name=n,type=\Nat]{\comp{n}}
2 \varseq{seqa}[name=a,type=\Nat]{1}{\varn}{\comp{a}_{#1}}
3
4 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:

```
1 $\addition{\seqa}$
```

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:

```
1 \vardef{varm}[name=m,type=\Nat]{\comp{m}}
2 \varseq{seqa}[
3     name=a,
4     args=2,
5     type=\Nat,
6 ]{1,1}{\varn,\varm}{\comp{a}_{#1}^{#2}}
7
8 $\seqa!$ 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:

```
1 \varseq{seqa}[
2     name=a,
3     type=\Nat,
4     args=2,
5     mid={\comp{a}_{\varn}^1,\comp{a}_1^2,\ellipses,\comp{a}_{1}^{\varm}}
6 ]{1,1}{\varn,\varm}{\comp{a}_{#1}^{#2}}
7
8 $\seqa!$ and $\addition{\seqa}$
```

Output:

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

.

5.4   Module Inheritance and Structures

The sTeX features for modular document management are inherited from the OM-Doc/MMT model that organizes knowledge into a graph, where the nodes are theories (called modules in sTeX) and the edges are truth-preserving mappings (called theory morphismes in MMT). We have already seen modules/theories above.

Before we get into theory morphisms in sTeX we will see a very simple application of modules: managing multilinguality modularly.

5.4.1   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 $\mathtt{lcm}(a, b)$ in english, but is called *kleinstes gemeinsames Vielfaches* in german and consequently denoted as $\mathtt{kgV}(a, b)$ 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`.

5.4.2 Simple Inheritance and Namespaces

`\importmodule`
`\usemodule`

`\importmodule[Some/Archive]{path?ModuleName}` is only allowed within an `smodule`-environment and makes the symbols declared in `ModuleName` available therein. Additionally the symbols 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.

`\STEXexport` `\importmodule` and `\usemodule` import all symbols, notations, semantic macros and (recursively) `\importmodule`s. If you want to additionally export e.g. convenience macros and other (ꟷTEX) code from a module, you can use the command `\STEXexport`{<code>} in your module. Then `<code>` is executed (both immediately and) every time the current module is opened via `\importmodule` or `\usemodule`.

For persistency reasons, everything in an `\STEXexport` is digested by TEXin the LʼTEX3-category code scheme. This means that the characters `_` and `:` are considered *letters* and valid parts of control sequence names, and space characters are

ignored entirely. For spaces, use the character ~ instead, and keep in mind, that if you want to use subscripts, you should use `\c_math_subscript_token` instead of `_`!

Also note, that **\newcommand** defines macros *globally* and throws an error if the macro already exists, potentially leading to low-level LaTeX errors if we put a **\newcommand** in an **\STEXexport** and the `<code>` is executed more than once in a document – which can happen easily.

A safer alternative is to use macro definition principles, that are safe to use even if the macro being defined already exists, and ideally are local to the current TeX group, such as **\def** or **\let**.

5.4.3   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, rather, structure *signatures*) in order to talk about (concrete or variable) *particular* monoids, topological spaces, partial orders etc.

`mathstructure` (*env.*) 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:

```
1 \begin{mathstructure}{monoid}
2    \symdef{universe}[type=\set]{\comp{U}}
3    \symdef{op}[
4        args=2,
5        type=\funtype{\universe,\universe}{\universe},
6        op=\circ
7    ]{#1 \comp{\circ} #2}
8    \symdef{unit}[type=\universe]{\comp{e}}
9 \end{mathstructure}
10
11 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:

```
1 \symdef{Int}[type=\set]{\comp{\mathbb Z}}
2 \symdef{addition}[
3    type=\funtype{\Int,\Int}{\Int},
4    args=2,
5    op=+
6 ]{##1 \comp{+} ##2}
7 \symdef{zero}[type=\Int]{\comp{0}}
8
9 $\mathstruct{\Int,\addition!,\zero}$ 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:

```
1 \instantiate{intmonoid}{monoid}{\mathbb{Z}_{+,0}}[
2    universe = Int ,
3    op = addition ,
4    unit = zero
5 ]
6
7 $\intmonoid{universe}$, $\intmonoid{unit}$ and $\intmonoid{op}{a}{b}$.
8
9 Also: $\intmonoid!$
```

Output:

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

.

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 (see [MRK18]):
>
> `mathstructure{<name>}` simply creates 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` generates a constant whose definiens is a record term of type `Mod(<name>-structure)`, with the fields assigned based on the respective 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.

The answer is that we can also instantiate a `mathstructure` with a *variable*. The syntax of `\varinstantiate` 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:

```
1 \varinstantiate{varM}{monoid}{M}
2
3 A \symname{monoid} is a structure
4 $\varM!:=\mathstruct{\varM{universe},\varM{op}!,\varM{unit}}$
5 such that
6 $\varM{op}!:\funtype{\varM{universe},\varM{universe}}{\varM{universe}}$ ...
```

Output:

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

.

and

Example 28

Input:

```
1  \varinstantiate{varMb}{monoid}{M_2}[universe = Int]
2
3  Let $\varMb!:=\mathstruct{\varMb{universe},\varMb{op}!,\varMb{unit}}$
4  be a \symname{monoid} on $\Int$ ...
```

Output:

Let $M_2 := \langle \mathbb{Z}, \circ, e \rangle$ be a monoid on $\mathbb{Z}$ ...

.

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

usestructure (*env.*) The usestructure{<struct>} environment is used in multilingual settings as a parallel to the mathstructure. It opens a group and then issues a \usemodule{.../<struct>-structure} that gives the body access to all the semantic macros in the referenced structure.

5.4.4 The `copymodule` Environment

TODO: explain

Given modules:

Example 29

Input:

```
1  \begin{smodule}{magma}
2      \symdef{universe}{\comp{\mathcal U}}
3      \symdef{operation}[args=2,op=\circ]{#1 \comp\circ #2}
4  \end{smodule}
5  \begin{smodule}{monoid}
6      \importmodule{magma}
7      \symdef{unit}{\comp e}
8  \end{smodule}
9  \begin{smodule}{group}
10     \importmodule{monoid}
11     \symdef{inverse}[args=1]{{#1}^{\comp{-1}}}
12 \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 30

Input:

```
1  \begin{smodule}{ring}
2      \begin{copymodule}{group}{addition}
3          \renamedecl[name=universe]{universe}{runiverse}
4          \renamedecl[name=plus]{operation}{rplus}
5          \renamedecl[name=zero]{unit}{rzero}
6          \renamedecl[name=uminus]{inverse}{ruminus}
7      \end{copymodule}
8      \notation*{rplus}[plus,op=+,prec=60]{#1 \comp+ #2}
9      \notation*{rzero}[zero]{\comp0}
10     \notation*{ruminus}[uminus,op=-]{\comp- #1}
11     \begin{copymodule}{monoid}{multiplication}
12         \assign{universe}{\runiverse}
13         \renamedecl[name=times]{operation}{rtimes}
14         \renamedecl[name=one]{unit}{rone}
15     \end{copymodule}
16     \notation*{rtimes}[cdot,op=\cdot,prec=50]{#1 \comp\cdot #2}
17     \notation*{rone}[one]{\comp1}
18     Test: $\rtimes a{\rplus c{\rtimes de}}$
19 \end{smodule}
```

Output:

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

.

TODO: explain donotclone

5.4.5 The `interpretmodule` Environment

TODO: explain

Example 31

Input:

```
1  \begin{smodule}{int}
2      \symdef{Integers}{\comp{\mathbb Z}}
3      \symdef{plus}[args=2,op=+]{#1 \comp+ #2}
4      \symdef{zero}{\comp0}
5      \symdef{uminus}[args=1,op=-]{\comp-#1}
6
7      \begin{interpretmodule}{group}{intisgroup}
8          \assign{universe}{\Integers}
9          \assign{operation}{\plus!}
10         \assign{unit}{\zero}
11         \assign{inverse}{\uminus!}
12     \end{interpretmodule}
13 \end{smodule}
```

Output:

.

5.5 Primitive Symbols (The sTeX Metatheory)

The stex-metatheory package contains sTeX symbols so ubiquitous, that it is virtually impossible to describe any flexiformal content without them, or that are required to annotate even the most primitive symbols with meaningful (foundation-independent) "type"-annotations, or required for basic structuring principles (theorems, definitions). As such, it serves as the default meta theory for any sTeX module.

We can also see the stex-metatheory as a foundation of mathematics in the sense of [Rab15], albeit an informal one (the ones discussed there are all formal foundations). The state of the stex-metatheory is necessarily incomplete, and will stay so for a long while: It arises as a collection of empirically useful symbols that are collected as more and more mathematics are encoded in sTeX and are classified as foundational.

Formal foundations should ideally instantiate these symbols with their formal counterparts, e.g. `isa` corresponds to a typing operation in typed setting, or the $\in$-operator in set-theoretic contexts; `bind` corresponds to a universal quantifier in (nth-order) logic, or a Π in dependent type theories.

We make this theory part of the sTeX collection due to the obiquity of the symbols involved. Note however, that the metatheory is for all practical purposes a "normal" sTeX module, and the symbols contained "normal" sTeX symbols.

Chapter 6

Using sTeX Symbols

Given a symbol declaration `\symdecl{symbolname}`, we obtain a semantic macro `\symbolname`. We can use this semantic macro in math mode to use its notation(s), and we can use `\symbolname!` in math mode to use its operator notation(s). What else can we do?

6.1 \symref and its variants

We have already seen `\symname` and `\symref`, the latter being the more general.

`\symref{<symbolname>}{<code>}` marks-up `<code>` as referencing `<symbolname>`. Since quite often, the `<code>` should be (a variant of) the name of the symbol anyway, we also have `\symname{<symbolname>}`.

Note that `\symname` uses the *name* of a symbol, not its macroname. More precisely, `\symname` will insert the name of the symbol with "`-`" replaced by spaces. If a symbol does not have an explicit `name=` given, the two are equal – but for `\symname` it often makes sense to make the two explicitly distinct. For example:

Example 32

Input:

```
1 \symdef{Nat}[
2    name=natural-number,
3    type=\set
4 ]{\comp{\mathbb{N}}}
5
6 A \symname{Nat} is...
```

Output:

> A natural number is...

.

`\symname` takes two additional optional arguments, `pre=` and `post=` that get prepended or appended respectively to the symbol name.

Additionally, `\Symname` behaves exactly like `\symname`, but will capitalize the first letter of the name:

Example 33
Input:

```
1 \Symname[post=s]{Nat} are...
```

Output:

> Natural numbers are...

.

> This is as good a place as any other to explain how ᴤTₑX resolves a string `symbolname` to an actual symbol.
> If `\symbolname` is a semantic macro, then ᴤTₑX has no trouble resolving `symbolname` to the full 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...`. ᴤTₑX attempts to handle this case thusly:
> If `string` does *not* correspond to a semantic macro `\string` and does *not* contain a `?`, then ᴤTₑX checks all symbols currently in scope until it finds one, whose name is `string`. If `string` is of the form `pre?name`, ᴤTₑX first looks through all modules currently in scope, whose full 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.

6.2 Marking Up Text and On-the-Fly Notations

We can also use semantic macros outside of text mode though, which allows us to annotate arbitrary text fragments.

Let us assume again, that we have `\symdef{addition}[args=2]{#1 \comp+ #2}`. Then we can do

Example 34
Input:

```
1 \addition{\comp{The sum of} \arg{$\svar{n}$} \comp{ and }\arg{$\svar{m}$}}
2 is...
```

Output:

> The sum of n and m is...

˙…which marks up the text fragment as representing an *application* of the `addition`-symbol to two argument n and m.

←M→ As expected, the above example is translated to OMDoc/MMT as an
—M→ `OMA` with `<OMS name="...?addition"/>` as head and `<OMV name="n"/>` and
⤳T⤳ `<OMV name="m"/>` as arguments.

Note the difference in treating "arguments" between math mode and text mode. In math mode the (in this case two) tokens/groups following the `\addition` macro are treated as arguments to the addition function, whereas in text mode the group following `\addition` is taken to be the ad-hoc presentation. We drill in on this now.

`\arg` In text mode, every semantic macro takes exactly one argument, namely the text-fragment to be annotated. The `\arg` command is only valid within the argument to a semantic macro and marks up the *individual arguments* for the symbol.

We can also use semantic macros in text mode to invoke an operator itself instead of its application, with the usual syntax using `!`:

Example 35
Input:

```
1 \addition!{Addition} is...
```

Output:

```
  Addition is...
```

.

Indeed, `\symbolname!{<code>}` is exactly equivalent to `\symref{symbolname}{<code>}` (the latter is in fact implemented in terms of the former).

`\arg` also allows us to switch the order of arguments around and "hide" arguments: For example, `\arg[3]{<code>}` signifies that `<code>` represents the *third* argument to the current operator, and `\arg*[i]{<code>}` signifies that `<code>` represents the ith argument, but it should not produce any output (it is exported in the `xhtml` however, so that MMT and other systems can pick up on it).[1]

EdN:1

Example 36
Input:

```
1 \addition{\comp{adding}
2     \arg[2]{$\svar{k}$}
3     \arg*{$\addition{\svar{n}}{\svar{m}}$}} yields...
```

---

[1]EDNOTE: MK: I do not understand why we have to/want to give the second arg*; I think this must be elaborated on.

Output:

> adding k yields...

˙Note that since the second `\arg` has no explicit argument number, it automatically represents the first not-yet-given argument – i.e. in this case the first one.[2]

The same syntax can be used in math mod as well. This allows us to spontaneously introduce new notations on the fly. We can activate it using the starred variants of semantic macros:

Example 37
Input:

```
1 Given $\addition{\svar{n}}{\svar{m}}$, then
2 $\addition*{
3     \arg*{\addition{\svar{n}}{\svar{m}}}
4     \comp{+}
5     \arg{\svar{k}}
6 }$ yields...
```

Output:

> Given $n+m$, then $+k$ yields...

.

If we take features like `\inputref` and `\mhinput` (and the `sfragment`-environment, see subsection 9.2.1) 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 inputted 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.

The `\sref` command attempts to do precisely that. Unlike plain `\ref`, `\autoref` etc., `\sref` refers to not just a *label*, but instead a pair consisting of a *label* and the *document* in whose context we want to refer to it. Conversely, every *document* (i.e. standalone compilable `.tex`-file) keeps track of the "names" (*Definition 3.1* etc.) for every label as determined in the context of the document, and stores them in a dedicated file `\jobname.sref`. Additionally, every document has a "*reference name*" (e.g. "*the section on Foo*"). This allows us to refer to "label x in document D" to yield "*Definition 1 in the section on Foo*". And of course, sTEX can decide based on the current document

---

[2]EDNOTE: MK: I do not understand this at all.

to either refer to the label by its "full name" or directly as e.g. *Definition 3.1* depending on whether the label occurs in the current document anyway (and link to it accordingly).

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, but allows for additional disambiguation between possibly duplicate labels across files, and

- (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=`⟨*file*⟩`]`

`{`⟨*label*⟩`}[archive=`⟨*archive2*⟩`,in=`⟨*document-context*⟩`,title=`⟨*title*⟩`]`

This command references ⟨*label*⟩ (declared in ⟨*file*⟩ in ⟨*archive1*⟩). If the object (section, figure, etc.) with that label occurs ultimately 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 ⟨*document-context*⟩ in ⟨*archive2*⟩.

For example, the reference to the sfragment-environment above will appear as "subsection 7.2.1 (Introduction) in the SₜₑX3 manual" if you are reading this in the package documentation for `stex-references` directly, but as a linked "subsection 7.2.1" in the full documentation or manual. This is achieved using

`\sref[file=stex-document-structure]{sec:ds:intro}[in=../stex-manual,title={the \sT`

For a further example, the following:

Part III (Extensions) in the full SₜₑX3 documentation

will say "Part III" (and link accordingly) in the full documentation, and "Part III (Extensions) in the full SₜₑX3 documentation" everywhere else. This is achieved using

`\sref[file=../stex-doc]{part:extends}[in=../stex-doc,title={the full \sTeX{}3 docu`

`\extref` `\sref[archive=`⟨*archive1*⟩`,file=`⟨*file*⟩`]`

`{`⟨*label*⟩`}{archive=`⟨*archive2*⟩`,in=`⟨*document-context*⟩`,title=`⟨*title*⟩`}`

The `\extref`-command behaves exactly like `\sref`, but takes *required* the document context argument and will always use it for generating the document text, regardless of whether the label occurs in the current document.

Chapter 7

sTEX Statements

7.1 Definitions, Theorems, Examples, Paragraphs

As mentioned earlier, we can semantically mark-up *statements* such as definitions, theorems, lemmata, examples, etc.

The corresponding environments for that are:

- `sdefinition` for definitions,

- `sassertion` for assertions, i.e. propositions that are declared to be *true*, such as theorems, lemmata, axioms,

- `sexample` for examples and counterexamples, and

- `sparagraph` for "other" semantic paragraphs, such as comments, remarks, conjectures, etc.

The *presentation* of these environments can be customized to use e.g. predefined `theorem`-environments, see section 7.3 for details.

All of these environments take optional arguments in the form of `key=value`-pairs. Common to all of them are the keys `id=` (for cross-referencing, see chapter 8), `type=` for customization (see section 7.3) and additional information (e.g. definition principles, "difficulty" etc), as well as `title=` (for giving the paragraph a title), and finally `for=`.

The `for=` key expects a comma-separated list of existing symbols, allowing for e.g. things like

Example 38

Input:

```
1 \begin{sexample}[
2     id=additionandmultiplication.ex,
3     for={addition,multiplication},
4     type={trivial,boring},
5     title={An Example}
6 ]
7     $\addition{2,3}$ is $5$, $\multiplication{2,3}$ is $6$.
8 \end{sexample}
```

Output:

> **Example 7.1.1** (An Example). 2+3 is 5, 2·3 is 6.

.

sdefinition (and sparagraph with type=symdoc) introduce three new macros: definiendum
behaves like symref (and definame/Definame like symname/Symname, respectively), but
highlights the referenced symbol as *being defined* in the current definition.

> ↪M→ The special type=symdoc for sparagraph is intended to be used for "informal
> ─M→ definitions", or encyclopedia-style descriptions for symbols.
> ⤳T⤳ The MMT system can use those (in lieu of an actual sdefinition in scope) to
> present to users, e.g. when hovering over symbols.

Additionally, sdefinition (and sparagraph with type=symdoc) introduces \definiens[<optional sym
which marks up <code> as being the explicit *definiens* of <optional symbolname> (in
case for= has multiple symbols).

All four statement environments – i.e. sdefinition, sassertion, sexample, and
sparagraph – also take an optional parameter name= – if this one is given a value, the
environment will generate a *symbol* by that name (but with no semantic macro). Not
only does this allow for \symref et al, it allows us to resume our earlier example for
monoids much more nicely:[3]

EdN:3

> **Example 39**
> Input:

---

[3]EDNOTE: MK: we should reference the example explicitly here.

```
1 \begin{mathstructure}{monoid}
2     \symdef{universe}[type=\set]{\comp{U}}
3     \symdef{op}[
4         args=2,
5         type=\funtype{\universe,\universe}{\universe},
6         op=\circ
7     ]{#1 \comp{\circ} #2}
8     \symdef{unit}[type=\universe]{\comp{e}}
9
10     \begin{sparagraph}[type=symdoc,for=monoid]
11         A \definame{monoid} is a structure
12         $\mathstruct{\universe,\op!,\unit}$
13         where $\op!:\funtype{\universe}{\universe}$ and
14         $\inset{\unit}{\universe}$ such that
15
16         \begin{sassertion}[name=associative,
17             type=axiom,
18             title=Associativity]
19             $\op!$ is associative
20         \end{sassertion}
21         \begin{sassertion}[name=isunit,
22             type=axiom,
23             title=Unit]
24             $\equal{\op{\svar{x}}{\unit}}{\svar{x}}$
25             for all $\inset{\svar{x}}{\universe}$
26         \end{sassertion}
27     \end{sparagraph}
28 \end{mathstructure}
29
30 An example for a \symname{monoid} is...
```

Output:

A **monoid** is a structure $\langle U, \circ, e \rangle$ where $\circ : U \to U$ and $e \in U$ such that

Axiom 7.1.2 (Associativity). $\circ$ *is associative*

Axiom 7.1.3 (Unit). $x \circ e = x$ *for all* $x \in U$

An example for a monoid is...

.

The main difference to before[4] is that the two `sassertion`s now have `name=` attributes. Thus the `mathstructure` monoid now contains two additional symbols, namely the axioms for associativity and that e is a unit. Note that both symbols do not represent the mere *propositions* that e.g. $\circ$ is associative, but *the assertion that it is actually true* that $\circ$ is associative.

If we now want to instantiate `monoid` (unless with a variable, of course), we also need to assign `associative` and `neutral` to analogous assertions. So the earlier example

```
1 \instantiate{intmonoid}{monoid}{\mathbb{Z}_{+,0}}[
2     universe = Int ,
3     op = addition ,
4     unit = zero
5 ]
```

---

[4]EdNote: MK: reference

49

...will not work anymore. We now need to give assertions that `addition` is associative and that `zero` is a unit with respect to addition.[2]

7.2 Proofs

The stex-proof package supplies macros and environment that allow to annotate the structure of mathematical proofs in SₜᴇX document. This structure can be used by MKM systems for added-value services, either directly from the SₜᴇX sources, or after translation.

Its central component is the `sproof`-environment, whose body consists of:

- *subproofs* via the `subproof`-environment,

- *proof steps* via the `\spfstep`, `\eqstep` `\assumption`, and `\conclude` macros, and

- *comments*, via normal text without special markup.

`sproof`, `subproof` and the various proof step macros take the following optional arguments:

   id (⟨*string*⟩) for referencing,

method (⟨*string*⟩) the proof method (e.g. contradiction, induction,...)

 term (⟨*token list*⟩) the (ideally semantically-marked up) proposition that is derived/proven by this proof/subproof/proof step.

Additionally, they take one mandatory argument for the document text to be annotated, or (in the case of the environments) as an introductory description of the proof itself. Since the latter often contains the `term` to be derived as text, alternatively to providing it as an optional argument, the mandatory argument can use the `\yield`-macro to mark it up in the text.

The `sproof` and `subproof` environments additionally take two optional arguments:

 for the symbol identifier/name corresponding to the `sassertion` to be proven. This too subsumes `\yield` and the `term`-argument.

hide In the pdf, this only shows the mandatory argument text and hides the body of the environment. In the HTML (as served by Mᴍᴛ), the bodies of all `proof` and `subproof` environments are *collapsible*, and `hide` collapses the body by default.

```
1    \begin{sassertion}[type=theorem,name=sqrt2irr]
2      \conclusion{\irrational{$\arg{\realroot{2}}$ is \comp{irrational}}}.
3  \end{sassertion}
4
5  \begin{sproof}[for=sqrt2irr,method=contradiction]{By contradiction}
6      \assumption{Assume \yield{\rational{$\arg{\realroot{2}}$ is
7        \comp{rational}}}}
8      \begin{subproof}[method=straightforward]{Then
9          \yield{$\eq{\ratfrac{\intpow{\vara}{2}}{\intpow{\varb}2}}{2}$
10          for some $\inset{\vara,\varb}\PosInt$ with
11          \coprime{$\arg{\vara},\arg{\varb}$ \comp{coprime}}}}
```

---

[2]Of course, SₜᴇX can not check that the assertions are the "correct" ones – but if the assertions (both in `monoid` as well as those for addition and zero) are properly marked up, Mᴍᴛ can. TODO: should

50

```
12          \assumption{By assumption, \yield{there are
13          $\inset{\vara,\varb}\PosInt $ with
14          $\realroot{2}=\ratfrac{\vara}{\varb}$}}
15          \spfstep{wlog, we can assume \coprime{$\arg{\vara},\arg{\varb}$
16          to be \comp{coprime}}}
17              % a comment:
18              If not, reduce the fraction until numerator and denominator
19              are coprime, and let the resulting components be
20              $\vara $ and $\varb $
21          \spfstep{Then \yield{$\eq{\intpow{\ratfrac{\vara}{\varb}}2}2$}}
22          \eqstep{\ratfrac{\intpow{\vara}2}{\intpow{\varb}2}}
23      \end{subproof}
24      \begin{subproof}[term=\divides{2}{\vara},method=straightforward]{
25          Then $\vara $ is even}
26          \spfstep{Multiplying the equation by $\intpow{\varb}2$ yields
27          $\yield{\eq{\intpow{\vara}2}{\inttimes{2}{\intpow{\varb}2}}}$}
28          \spfstep[term=\divides{2}{\intpow{\vara}2}]{Hence
29          $\intpow{\vara}2$ is even}
30          \conclude[term=\divides{2}{\vara}]{Hence $\vara $ is even as well}
31          % another comment:
32          Hint: Think about the prime factorizations of $\vara $ and
33          $\intpow{\vara}2$
34      \end{subproof}
35      \begin{subproof}[term=\divides{2}{\varb},method=straightforward,]{
36          Then $\varb $ is also even}
37          \spfstep{Since $\vara $ is even, we have \yield{some $\varc $
38              such that $\eq{\inttimes{2}{\varc}}{\vara}$}}
39          \spfstep{Plugging into the above, we get
40              \yield{$\eq{\intpow{\inttimes{2}{\vara}}2}
41                  {\inttimes{2}{\intpow{\varb}2}}$}}
42          \eqstep{\inttimes{4}{\intpow{\vara}2}}
43          \spfstep{Dividing both sides by $2$ yields
44              \yield{$\eq{\intpow{\varb}2}{\inttimes{2}{\intpow{\vara}2}}$}}
45          \spfstep[term=\divides{2}{\intpow{\varb}2}]{Hence
46              $\intpow{\varb}2$ is even}
47          \conclude[term=\divides{2}{\varb}]{Hence $\varb $ is even}
48          % one more comment:
49          By the same argument as above
50      \end{subproof}
51      \conclude[term=\contradiction]{Contradiction to $\vara,\varb $ being
52      \symname{coprime}.}
53 \end{sproof}
```

which will produce:

Theorem 7.2.1. $\sqrt{2}$ *is irrational.*

Proof: By contradiction

1. Assume $\sqrt{2}$ is rational

2. Then $(\frac{a^2}{b^2})=2$ for some $a,b \in \mathbb{Z}^+$ with a,b coprime

2.1. By assumption, there are $a,b \in \mathbb{Z}^+$ with $\sqrt{2} = \frac{a}{b}$

2.2. wlog, we can assume a, b to be coprime

   If not, reduce the fraction until numerator and denominator are coprime, and let the re-

---

sulting components be a and b

2.3. Then $(\frac{a}{b})^2 {=} 2$

$= \frac{a^2}{b^2}$

3. Then a is even
3.1. Multiplying the equation by b^2 yields $a^2{=}2b^2$

3.2. Hence a^2 is even

$\Rightarrow$ Hence a is even as well

Hint: Think about the prime factorizations of a and a^2

4. Then b is also even
4.1. Since a is even, we have some c such that $2c{=}a$

4.2. Plugging into the above, we get $(2a)^2{=}2b^2$

$= 4a^2$

4.3. Dividing both sides by 2 yields $b^2{=}2a^2$

4.4. Hence b^2 is even

$\Rightarrow$ Hence b is even

By the same argument as above

$\Rightarrow$ Contradiction to a, b being coprime.

$\square$

If we mark all subproofs with `hide`, we will obtain the following instead:

Theorem 7.2.2. $\sqrt{2}$ *is irrational.*
Proof: By contradiction
1. Assume $\sqrt{2}$ is rational

2. Then $(\frac{a^2}{b^2}){=}2$ for some $a,b{\in}\mathbb{Z}^+$ with a, b coprime

3. Then a is even

4. Then b is also even

$\Rightarrow$ Contradiction to a, b being coprime.

$\square$

However, the hidden subproofs will still be shown in the HTML, only in an expandable section which is collapsed by default.

The above style of writing proofs is usually called *structured proofs*. They have a huge advantage over the traditional purely prosaic style, in that (as the name suggests) the actual *structure* of the proof is made explicit, which almost always makes it considerably more comprehensible. We, among many others, encourage the general use of structured proofs.

Alas, most proofs are not written in this style, and we would do users a disservice by insisting on this style. For that reason, the `spfblock` environment turns all subproofs and proof step macros into presentationally neutral *inline* annotations, as in the induction step of the following example:

```
1 \begin{sproof}[id=simple-proof,method=induction]
2   {We prove that $\sum_{i=1}^n{2i-1}=n^{2}$ by induction over $n$}
```

```
 3   For the induction we have to consider three cases: % <- a comment
 4    \begin{subproof}{$n=1$}
 5     \spfstep*{then we compute $1=1^2$}
 6    \end{subproof}
 7    \begin{subproof}{$n=2$}
 8         This case is not really necessary, but we do it for the
 9         fun of it (and to get more intuition).
10       \spfstep*{We compute $1+3=2^{2}=4$.}
11    \end{subproof}
12    \begin{subproof}{$n>1$}\begin{spfblock}
13       \assumption[id=ind-hyp]{
14         Now, we assume that the assertion is true for a certain $k\geq 1$,
15         i.e. \yield{$\sum_{i=1}^k{(2i-1)}=k^{2}$}.
16       }
17
18         We have to show that we can derive the assertion for $n=k+1$ from
19         this assumption, i.e. $\sum_{i=1}^{k+1}{(2i-1)}=(k+1)^{2}$.
20
21       \spfstep{
22         We obtain $\yield{\sum_{i=1}^{k+1}{2i-1}=
23           \sum_{i=1}^k{2i-1}+2(k+1)-1}$
24         \spfjust{by \splitsum{\comp{splitting the sum}
25         \arg*{$\sum_{i=1}^{k+1}{(2i-1)}=(k+1)^{2}$}}}.
26       }
27       \spfstep{
28         Thus we have $\yield{\sum_{i=1}^{k+1}{(2i-1)}=k^2+2k+1}$
29         \spfjust{by \symname{induction-hypothesis}}.
30       }
31       \conclude{
32         We can \spfjust{\simplification{\comp{simplify} the right-hand side
33         \arg*{k^2+2k+1}}} to
34         ${k+1}^2$, which proves the assertion.
35       }
36    \end{spfblock}\end{subproof}
37     \conclude{
38       We have considered all the cases, so we have proven the assertion.
39     }
40 \end{sproof}
```

This yields the following result:

Proof: We prove that $\sum_{i=1}^n 2i - 1 = n^2$ by induction over n

For the induction we have to consider three cases:

1. $n = 1$

   then we compute $1 = 1^2$

2. $n = 2$

   This case is not really necessary, but we do it for the fun of it (and to get more intuition).

   We compute $1 + 3 = 2^2 = 4$.

3. $n > 1$

   Now, we assume that the assertion is true for a certain $k \geq 1$, i.e. $\sum_{i=1}^k (2i - 1) = k^2$.

   We have to show that we can derive the assertion for $n = k+1$ from this assumption,

---

i.e. $\sum_{i=1}^{k+1}(2i-1) = (k+1)^2$.

We obtain $\sum_{i=1}^{k+1} 2i - 1 = \sum_{i=1}^{k} 2i - 1 + 2(k+1) - 1$ by splitting the sum. Thus we have $\sum_{i=1}^{k+1}(2i-1) = k^2 + 2k + 1$ by induction hypothesis. We can simplify the right-hand side to $k+1^2$, which proves the assertion.

$\Rightarrow$ We have considered all the cases, so we have proven the assertion.

$\square$

sproof (*env.*) The `sproof` environment is the main container for proofs. It takes an optional `KeyVal` argument that allows to specify the `id` (identifier) and `for` (for which assertion is this a proof) keys. The regular argument of the `proof` environment contains an introductory comment, that may be used to announce the proof style. The `proof` environment contains a sequence of `spfstep`, `spfcomment`, and `spfcases` environments that are used to markup the proof steps.

\spfidea The `\spfidea` macro allows to give a one-paragraph description of the proof idea.

\spfsketch For one-line proof sketches, we use the `\spfsketch` macro, which takes the same optional argument as `sproof` and another one: a natural language text that sketches the proof.

\spfstep Regular proof steps are marked up with the `\spfstep` macro, which takes an optional `KeyVal` argument for annotations. A proof step usually contains a local assertion (the text of the step) together with some kind of evidence that this can be derived from already established assertions.

\yield See above

\spfjust This evidence is marked up with the `\spfjust` macro in the stex-proofs package. This environment totally invisible to the formatted result; it wraps the text in the proof step that corresponds to the evidence (ideally, a semantically marked-up term).

\assumption The `\assumption` macro allows to mark up a (justified) assumption.

\justarg

subproof (*env.*) The `subproof` environment is used to mark up a subproof. This environment takes an optional `KeyVal` argument for semantic annotations and a second argument that allows to specify an introductory comment (just like in the `proof` environment). The `method` key can be used to give the name of the proof method executed to make this subproof.

\sproofend Traditionally, the end of a mathematical proof is marked with a little box at the end of the last line of the proof (if there is space and on the end of the next line if there isn't), like so:

The stex-proofs package provides the `\sproofend` macro for this.

\sProofEndSymbol If a different symbol for the proof end is to be used (e.g. *q.e.d*), then this can be obtained by specifying it using the `\sProofEndSymbol` configuration macro (e.g. by specifying `\sProofEndSymbol{q.e.d}`).

Some of the proof structuring macros above will insert proof end symbols for sub-proofs, in most cases, this is desirable to make the proof structure explicit, but sometimes this wastes space (especially, if a proof ends in a case analysis which will supply its own proof end marker). To suppress it locally, just set `proofend={}` in them or use use `\sProofEndSymbol{}`.

7.3 Highlighting and Presentation Customizations

The environments starting with `s` (i.e. `smodule`, `sassertion`, `sexample`, `sdefinition`, `sparagraph` and `sproof`) by default produce no additional output whatsoever (except for the environment content of course). Instead, the document that uses them (whether directly or e.g. via `\inputref`) can decide how these environments are supposed to look like.

The stexthm package defines some default customizations that can be used, but of course many existing LaTeX templates come with their own `definition`, `theorem` and similar environments that authors are supposed (or even required) to use. Their concrete syntax however is usually not compatible with all the additional arguments that sTeX allows for semantic information.

Therefore we introduced the separate environments `sdefinition` etc. instead of using `definition` directly. We allow authors to specify how these environments should be styled via the commands `stexpatch*`.

\stexpatchmodule
\stexpatchdefinition
\stexpatchassertion
\stexpatchexample
\stexpatchparagraph
\stexpatchproof

All of these commands take one optional and two proper arguments, i.e. `\stexpatch*[<type>]{<begin-code>}{<end-code>}`.

After sTeX reads and processes the optional arguments for these environments, (some of) their values are stored in the macros `\s*<field>` (i.e. `sexampleid`, `\sassertionname`, etc.). It then checks for all the values `<type>` in the `type=`-list, whether an `\stexpatch*[<type>]` for the current environment has been called. If it finds one, it uses the patches `<begin-code>` and `<end-code>` to mark up the current environment. If no patch for (any of) the type(s) is found, it checks whether and `\stexpatch*` was called without optional argument.

For example, if we want to use a predefined `theorem` environment for `sassertion`s with `type=theorem`, we can do

```
1 \stexpatchassertion[theorem]{\begin{theorem}}{\end{theorem}}
```

...or, rather, since e.g. `theorem`-like environments defined using amsthm take an optional title as argument, we can do:

```
1 \stexpatchassertion[theorem]
2   {\ifx\sassertiontitle\@empty
3       \begin{theorem}
4    \else
5       \begin{theorem}[\sassertiontitle]
6    \fi}
7 {\end{theorem}}
```

Or, if we want *all kinds of* sdefinitions to use a predefined definition-environment irrespective of their `type=`, then we can issue the following customization patch:

```
1 \stexpatchdefinition
2   {\ifx\sdefinitiontitle\@empty
3       \begin{definition}
4    \else
5       \begin{definition}[\sdefinitiontitle]
6    \fi}
7   {\end{definition}}
```

`\compemph`
`\varemph`
`\symrefemph`
`\defemph`

Apart from the environments, we can control how sTEX highlights variables, notation components, `\symref`s and `\definiendum`s, respectively.

To do so, we simply redefine these four macros. For example, to highlight notation components (i.e. everything in a `\comp`) in blue, as in this document, we can do `\def\compemph#1{\textcolor{blue}{#1}}`. By default, `\compemph` et al do nothing.

`\compemph@uri`
`\varemph@uri`
`\symrefemph@uri`
`\defemph@uri`

For each of the four macros, there exists an additional macro that takes the full URI of the relevant symbol currently being highlighted as a second argument. That allows us to e.g. use pdf tooltips and links. For example, this document uses[5]

```
1 \protected\def\symrefemph@uri#1#2{
2   \pdftooltip{
3     \symrefemph{#1}
4   }{
5     URI:~\detokenize{#2}
6   }
7 }
```

By default, `\compemph@uri` is simply defined as `\compemph{#1}` (analogously for the other three commands).

56

Chapter 8

Cross References

If we take features like `\inputref` and `\mhinput` (and the `sfragment`-environment, see subsection 9.2.1) 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 inputted 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.

The `\sref` command attempts to do precisely that. Unlike plain `\ref`, `\autoref` etc., `\sref` refers to not just a *label*, but instead a pair consisting of a *label* and the *document* in whose context we want to refer to it. Conversely, every *document* (i.e. standalone compilable `.tex`-file) keeps track of the "names" (*Definition 3.1* etc.) for every label as determined in the context of the document, and stores them in a dedicated file `\jobname.sref`. Additionally, every document has a "*reference name*" (e.g. "*the section on Foo*"). This allows us to refer to "label x in document D" to yield "*Definition 1 in the section on Foo*". And of course, sTEX can decide based on the current document to either refer to the label by its "full name" or directly as e.g. *Definition 3.1* depending on whether the label occurs in the current document anyway (and link to it accordingly).

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, but allows for additional disambiguation between possibly duplicate labels across files, and

- (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=⟨*file*⟩]
{⟨*label*⟩}[archive=⟨*archive2*⟩,in=⟨*document-context*⟩,title=⟨*title*⟩]

This command references ⟨*label*⟩ (declared in ⟨*file*⟩ in ⟨*archive1*⟩). If the object (section, figure, etc.) with that label occurs ultimately 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 ⟨*document-context*⟩ in ⟨*archive2*⟩.

For example, the reference to the sfragment-environment above will appear as "subsection 7.2.1 (Introduction) in the sTeX3 manual" if you are reading this in the package documentation for `stex-references` directly, but as a linked "subsection 7.2.1" in the full documentation or manual. This is achieved using

    \sref[file=stex-document-structure]{sec:ds:intro}[in=../stex-manual,title={the \sTe

For a further example, the following:

<div align="center">Part III (Extensions) in the full sTeX3 documentation</div>

will say "Part III" (and link accordingly) in the full documentation, and "Part III (Extensions) in the full sTeX3 documentation" everywhere else. This is achieved using

    \sref[file=../stex-doc]{part:extends}[in=../stex-doc,title={the full \sTeX{}3 docur

---
\extref \sref[archive=⟨*archive1*⟩,file=⟨*file*⟩]
{⟨*label*⟩}{archive=⟨*archive2*⟩,in=⟨*document-context*⟩,title=⟨*title*⟩}

The \extref-command behaves exactly like \sref, but takes *required* the document context argument and will always use it for generating the document text, regardless of whether the label occurs in the current document.

Chapter 9

Additional Packages

9.1 Tikzinput: Treating TIKZ code as images

`image` The behavior of the `ikzinput` package is determined by whether the `image` option is given. If it is not, then the `tikz` package is loaded, all other options are passed on to it and `\tikzinput{⟨file⟩}` inputs the TIKZ file ⟨file⟩`.tex`; if not, only the `graphicx` package is loaded and `\tikzinput{⟨file⟩}` loads an image file ⟨file⟩`.`⟨ext⟩ generated from ⟨file⟩`.tex`.

The selective input functionality of the `tikzinput` package assumes that the TIKZ pictures are externalized into a standalone picture file, such as the following one

```
1 \documentclass{standalone}
2 \usepackage{tikz}
3 \usetikzpackage{...}
4 \begin{document}
5   \begin{tikzpicture}
6     ...
7   \end{tikzpicture}
8 \end{document}
```

The `standalone` class is a minimal LATEX class that when loaded in a document that uses the `standalone` package: the preamble and the `documenat` environment are disregarded during loading, so they do not pose any problems. In effect, an `\input` of the file above only sees the `tikzpicture` environment, but the file itself is standalone in the sense that we can run LATEX over it separately, e.g. for generating an image file from it.

`\tikzinput`
`\ctikzinput` This is exactly where the `tikzinput` package comes in: it supplies the `\tikzinput` macro, which – depending on the `image` option – either directly inputs the TIKZ picture (source) or tries to load an image file generated from it.

Concretely, if the `image` option is not set for the `tikzinput` package, then `\tikzinput[⟨opt⟩]{⟨file⟩}` disregards the optional argument ⟨opt⟩ and inputs ⟨file⟩`.tex` via `\input` and resizes it to as specified in the `width` and `height` keys. If it is, `\tikzinput[⟨opt⟩]{⟨file⟩}` expands to `\includegraphics[⟨opt⟩]{⟨file⟩}`.

`\ctizkinput` is a version of `\tikzinput` that is centered.

\mhtikzinput
\cmhtikzinput `\mhtizkinput` is a variant of `\tikzinput` that treats its file path argument as a relative path in a math archive in analogy to `\inputref`. To give the archive path, we use the `mhrepos=` key. Again, `\cmhtizkinput` is a version of `\mhtikzinput` that is centered.

\libusetikzlibrary Sometimes, we want to supply archive-specific TIKZ libraries in the `lib` folder of the archive or the `meta-inf/lib` of the archive group. Then we need an analogon to `\libinput` for `\usetikzlibrary`. The `stex-tikzinput` package provides the `libusetikzlibrary` for this purpose.

9.2 Modular Document Structuring

9.2.1 Introduction

The `document-structure` package supplies an infrastructure for writing OMDoc documents in LaTeX. This includes a simple structure sharing mechanism for STEX that allows to to move from a copy-and-paste document development model to a copy-and-reference model, which conserves space and simplifies document management. The augmented structure can be used by MKM systems for added-value services, either directly from the STEX sources, or after translation.

The `document-structure` package supplies macros and environments that allow to label document fragments and to reference them later in the same document or in other documents. In essence, this enhances the document-as-trees model to documents-as-directed-acyclic-graphs (DAG) model. This structure can be used by MKM systems for added-value services, either directly from the STEX sources, or after translation. Currently, trans-document referencing provided by this package can only be used in the STEX collection.

DAG models of documents allow to replace the "Copy and Paste" in the source document with a label-and-reference model where document are shared in the document source and the formatter does the copying during document formatting/presentation.

9.2.2 Package Options

The `document-structure` package accepts the following options:

class=⟨*name*⟩	load ⟨*name*⟩`.cls` instead of `article.cls`
topsect=⟨*sect*⟩	The top-level sectioning level; the default for ⟨*sect*⟩ is `section`

9.2.3 Document Fragments

sfragment (*env.*) The structure of the document is given by nested `sfragment` environments. In the LaTeX route, the `sfragment` environment is flexibly mapped to sectioning commands, inducing the proper sectioning level from the nesting of `sfragment` environments. Correspondingly, the `sfragment` environment takes an optional key/value argument for metadata followed by a regular argument for the (section) title of the sfragment. The optional metadata argument has the keys `id` for an identifier, `creators` and `contributors` for the Dublin Core metadata [DCM03]. The option `short` allows to give a short title for the generated section. If the title contains semantic macros, we need to give the `loadmodules` key (it needs no value). For instance we would have

```
1 \begin{smodule}{foo}
2   \symdef{bar}{B^a_r}
3   ...
4   \begin{sfragment}[id=sec.barderiv,loadmodules]
5     {Introducing $\protect\bar$ Derivations}
```

sTeX automatically computes the sectioning level, from the nesting of `sfragment` environments.

But sometimes, we want to skip levels (e.g. to use a `\subsection*` as an introduction for a chapter).

blindfragment (*env.*) Therefore the document-structure package provides a variant `blindfragment` that does not produce markup, but increments the sectioning level and logically groups document parts that belong together, but where traditional document markup relies on convention rather than explicit markup. The `blindfragment` environment is useful e.g. for creating frontmatter at the correct level. The example below shows a typical setup for the outer document structure of a book with parts and chapters.

```
1 \begin{document}
2 \begin{blindfragment}
3 \begin{blindfragment}
4 \begin{frontmatter}
5 \maketitle\newpage
6 \begin{sfragment}{Preface}
7 ... <<preface>> ...
8 \end{sfragment}
9 \clearpage\setcounter{tocdepth}{4}\tableofcontents\clearpage
10 \end{frontmatter}
11 \end{blindfragment}
12 ... <<introductory remarks>> ...
13 \end{blindfragment}
14 \begin{sfragment}{Introduction}
15 ... <<intro>> ...
16 \end{sfragment}
17 ... <<more chapters>> ...
18 \bibliographystyle{alpha}\bibliography{kwarc}
19 \end{document}
```

Here we use two levels of `blindfragment`:

- The outer one groups the introductory parts of the book (which we assume to have a sectioning hierarchy topping at the part level). This `blindfragment` makes sure that the introductory remarks become a "chapter" instead of a "part".

- The inner one groups the frontmatter[3] and makes the preface of the book a section-level construct. The `frontmatter` environment also suppresses numbering as is traditional for prefaces.

`\skipfragment` The `\skipfragment` "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`.

---

[3]We shied away from redefining the `frontmatter` to induce a blindfragment, but this may be the "right" way to go in the future.

`\currentsectionlevel` `\CurrentSectionLevel`	The `\currentsectionlevel` macro supplies the name of the current sectioning level, e.g. "chapter", or "subsection". `\CurrentSectionLevel` is the capitalized variant. They are useful to write something like "In this `\currentsectionlevel`, we will..." in an `sfragment` environment, where we do not know which sectioning level we will end up.

9.2.4 Ending Documents Prematurely

`\prematurestop` `\afterprematurestop`	For prematurely stopping the formatting of a document, sTeX provides the `\prematurestop` macro. It can be used everywhere in a document and ignores all input after that – backing out of the `sfragment` environments as needed. After that – and before the implicit `\end{document}` it calls the internal `\afterprematurestop`, which can be customized to do additional cleanup or e.g. print the bibliography.

`\prematurestop` is useful when one has a driver file, e.g. for a course taught multiple years and wants to generate course notes up to the current point in the lecture. Instead of commenting out the remaining parts, one can just move the `\prematurestop` macro. This is especially useful, if we need the rest of the file for processing, e.g. to generate a theory graph of the whole course with the already-covered parts marked up as an overview over the progress; see `import_graph.py` from the `lmhtools` utilities [LMH].

Text fragments and modules can be made more re-usable by the use of global variables. For instance, the admin section of a course can be made course-independent (and therefore re-usable) by using variables (actually token registers) `courseAcronym` and `courseTitle` instead of the text itself. The variables can then be set in the sTeX preamble of the course notes file.

9.2.5 Global Document Variables

To make document fragments more reusable, we sometimes want to make the content depend on the context. We use **document variables** for that.

`\setSGvar` `\useSGvar`	`\setSGvar{⟨vname⟩}{⟨text⟩}` to set the global variable ⟨vname⟩ to ⟨text⟩ and `\useSGvar{⟨vname⟩}` to reference it.

`\ifSGvar`	With `\ifSGvar` we can test for the contents of a global variable: the macro call `\ifSGvar{⟨vname⟩}{⟨val⟩}{⟨ctext⟩}` tests the content of the global variable ⟨vname⟩, only if (after expansion) it is equal to ⟨val⟩, the conditional text ⟨ctext⟩ is formatted.

9.3 Slides and Course Notes

9.3.1 Introduction

The `notesslides` document class is derived from `beamer.cls` [Tana], it adds a "notes version" for course notes that is more suited to printing than the one supplied by `beamer.cls`.

The `notesslides` class takes the notion of a slide frame from Till Tantau's excellent `beamer` class and adapts its notion of frames for use in the sTeX and OMDoc. To

support semantic course notes, it extends the notion of mixing frames and explanatory text, but rather than treating the frames as images (or integrating their contents into the flowing text), the notesslides package displays the slides as such in the course notes to give students a visual anchor into the slide presentation in the course (and to distinguish the different writing styles in slides and course notes).

In practice we want to generate two documents from the same source: the slides for presentation in the lecture and the course notes as a narrative document for home study. To achieve this, the notesslides class has two modes: *slides mode* and *notes mode* which are determined by the package option.

9.3.2 Package Options

The notesslides class takes a variety of class options:

slides
notes
The options `slides` and `notes` switch between slides mode and notes mode (see subsection 9.3.3).

sectocframes
If the option `sectocframes` is given, then for the `sfragment`s, special frames with the `sfragment` title (and number) are generated.

frameimages
fiboxed
If the option `frameimages` is set, then slide mode also shows the `\frameimage`-generated frames (see **??**). If also the `fiboxed` option is given, the slides are surrounded by a box.

9.3.3 Notes and Slides

frame (*env.*) Slides are represented with the `frame` environment just like in the beamer class, see [Tanb] for details.

note (*env.*) The notesslides class adds the `note` environment for encapsulating the course note fragments.

Note that it is essential to start and end the `notes` environment at the start of the line – in particular, there may not be leading blanks – else LaTeX becomes confused and throws error messages that are difficult to decipher.

By interleaving the `frame` and `note` environments, we can build course notes as shown here:

```
1 \ifnotes\maketitle\else
2 \frame[noframenumbering]\maketitle\fi
3
4 \begin{note}
5   We start this course with ...
6 \end{note}
7
8 \begin{frame}
9   \frametitle{The first slide}
10  ...
```

```
11 \end{frame}
12 \begin{note}
13   ... and more explanatory text
14 \end{note}
15
16 \begin{frame}
17   \frametitle{The second slide}
18   ...
19 \end{frame}
20 ...
```

\ifnotes Note the use of the \ifnotes conditional, which allows different treatment between notes and slides mode – manually setting \notestrue or \notesfalse is strongly discouraged however.

> ⚠ We need to give the title frame the **noframenumbering** option so that the frame numbering is kept in sync between the slides and the course notes.

> ⚠ The **beamer** class recommends not to use the **allowframebreaks** option on frames (even though it is very convenient). This holds even more in the **notesslides** case: At least in conjunction with \newpage, frame numbering behaves funnily (we have tried to fix this, but who knows).

\inputref* If we want to transclude a the contents of a file as a note, we can use a new variant \inputref* of the \inputref macro: \inputref*{foo} is equivalent to \begin{note}\inputref{foo}\end{note}.

nparagraph (*env.*) There are some environments that tend to occur at the top-level of **note** environments.
nparagraph (*env.*) We make convenience versions of these: e.g. the **nparagraph** environment is just an
ndefinition (*env.*) **sparagraph** inside a **note** environment (but looks nicer in the source, since it avoids one
nexample (*env.*) level of source indenting). Similarly, we have the **nfragment**, **ndefinition**, **nexample**,
nsproof (*env.*) **nsproof**, and **nassertion** environments.
nassertion (*env.*)

9.3.4 Customizing Header and Footer Lines

The notesslides package and class comes with a simple default theme named sTeX that provided by the beamterthemesTeX. It is assumed as the default theme for sTeX-based notes and slides. The result in **notes** mode (which is like the **slides** version except that the slide hight is variable) is
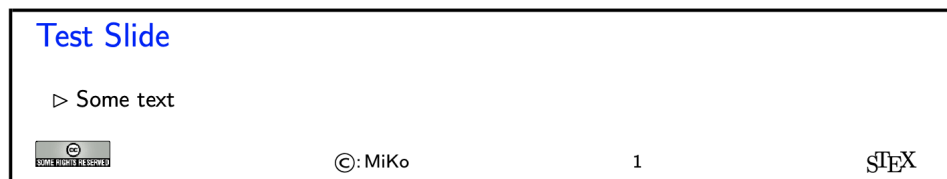
The footer line can be customized. In particular the logos.

\setslidelogo The default logo provided by the notesslides package is the sTEX logo it can be customized using \setslidelogo{⟨*logo name*⟩}.

\setsource The default footer line of the notesslides package mentions copyright and licensing. In notesslides \source stores the author's name as the copyright holder. By default it is the author's name as defined in the \author macro in the preamble. \setsource{⟨*name*⟩} can change the writer's name.

\setlicensing For licensing, we use the Creative Commons Attribuition-ShareAlike license by default to strengthen the public domain. If package hyperref is loaded, then we can attach a hyperlink to the license logo. \setlicensing[⟨*url*⟩]{⟨*logo name*⟩} is used for customization, where ⟨*url*⟩ is optional.

9.3.5   Frame Images

Sometimes, we want to integrate slides as images after all – e.g. because we already have a PowerPoint presentation, to which we want to add sTEX notes.

\frameimage
\mhframeimage
In this case we can use \frameimage[⟨*opt*⟩]{⟨*path*⟩}, where ⟨*opt*⟩ are the options of \includegraphics from the graphicx package [CR99] and ⟨*path*⟩ is the file path (extension can be left off like in \includegraphics). We have added the label key that allows to give a frame label that can be referenced like a regular beamer frame.

The \mhframeimage macro is a variant of \frameimage with repository support. Instead of writing

```
1 \frameimage{\MathHub{fooMH/bar/source/baz/foobar}}
```

we can simply write (assuming that \MathHub is defined as above)

```
1 \mhframeimage[fooMH/bar]{baz/foobar}
```

Note that the \mhframeimage form is more semantic, which allows more advanced document management features in MathHub.

If baz/foobar is the "current module", i.e. if we are on the MathHub path . . . MathHub/fooMH/bar. . . , then stating the repository in the first optional argument is redundant, so we can just use

```
1 \mhframeimage{baz/foobar}
```

\textwarning The \textwarning macro generates a warning sign: ⚠

9.3.6 Excursions

In course notes, we sometimes want to point to an "excursion" – material that is either presupposed or tangential to the course at the moment – e.g. in an appendix. The typical setup is the following:

```
1 \excursion{founif}{../fragments/founif.en}
2   {We will cover first-order unification in}
3 ...
4 \begin{appendix}\printexcursions\end{appendix}
```

It generates a paragraph that references the excursion whose source is in the file `../fragments/founif.en.tex` and automatically books the file for the `\printexcursions` command that is used here to put it into the appendix. We will look at the mechanics now.

`\excursion` The `\excursion{⟨ref⟩}{⟨path⟩}{⟨text⟩}` is syntactic sugar for

```
1 \begin{nparagraph}[title=Excursion]
2   \activateexcursion{founif}{../ex/founif}
3   We will cover first-order unification in \sref{founif}.
4 \end{nparagraph}
```

`\activateexcursion`
`\printexcursion`
`\excursionref`
Here `\activateexcursion{⟨path⟩}` augments the `\printexcursions` macro by a call `\inputref{⟨path⟩}`. In this way, the `\printexcursions` macro (usually in the appendix) will collect up all excursions that are specified in the main text.

Sometimes, we want to reference – in an excursion – part of another. We can use `\excursionref{⟨label⟩}` for that.

`\excursiongroup` Finally, we usually want to put the excursions into an `sfragment` environment and add an introduction, therefore we provide the a variant of the `\printexcursions` macro: `\excursiongroup[id=⟨id⟩,intro=⟨path⟩]` is equivalent to

```
1 \begin{note}
2 \begin{sfragment}[id=<id>]{Excursions}
3   \inputref{<path>}
4   \printexcursions
5 \end{sfragment}
6 \end{note}
```

When option `book` which uses `\pagestyle{headings}` is given and semantic macros are given in the `sfragment` titles, then they sometimes are not defined by the time the heading is formatted. Need to look into how the headings are made. This is a problem of the underlying `document-structure` package.

9.4 Representing Problems and Solutions

9.4.1 Introduction

The problem package supplies an infrastructure that allows specify problem. Problems are text fragments that come with auxiliary functions: hints, notes, and solutions[4]. Furthermore, we can specify how long the solution to a given problem is estimated to take and how many points will be awarded for a perfect solution.

Finally, the problem package facilitates the management of problems in small files, so that problems can be re-used in multiple environment.

9.4.2 Problems and Solutions

solutions
notes
hints
gnotes
pts
min
boxed
test

The problem package takes the options `solutions` (should solutions be output?), `notes` (should the problem notes be presented?), `hints` (do we give the hints?), `gnotes` (do we show grading notes?), `pts` (do we display the points awarded for solving the problem?), `min` (do we display the estimated minutes for problem soling). If theses are specified, then the corresponding auxiliary parts of the problems are output, otherwise, they remain invisible.

The `boxed` option specifies that problems should be formatted in framed boxes so that they are more visible in the text. Finally, the `test` option signifies that we are in a test situation, so this option does not show the solutions (of course), but leaves space for the students to solve them.

problem (*env.*) The main environment provided by the problempackage is (surprise surprise) the `problem` environment. It is used to mark up problems and exercises. The environment takes an optional KeyVal argument with the keys `id` as an identifier that can be reference later, `pts` for the points to be gained from this exercise in homework or quiz situations, `min` for the estimated minutes needed to solve the problem, and finally `title` for an informative title of the problem.

> **Example 40**
> Input:

---

[4]for the moment multiple choice problems are not supported, but may well be in a future version

```
 1 \documentclass{article}
 2 \usepackage[solutions,hints,pts,min]{problem}
 3 \begin{document}
 4   \begin{sproblem}[id=elefants,pts=10,min=2,title=Fitting Elefants]
 5     How many Elefants can you fit into a Volkswagen beetle?
 6     \begin{hint}
 7       Think positively, this is simple!
 8     \end{hint}
 9     \begin{exnote}
10       Justify your answer
11     \end{exnote}
12 \begin{solution}[for=elefants]
13   Four, two in the front seats, and two in the back.
14   \begin{gnote}
15     if they do not give the justification deduct 5 pts
16   \end{gnote}
17 \end{solution}
18 \end{sproblem}
19 \end{document}
```

Output:

Problem 9.4.1 (Fitting Elefants)
How many Elefants can you fit into a Volkswagen beetle?

Hint: Think positively, this is simple!

Note: Justify your answer

Solution: Four, two in the front seats, and two in the back.

Grading: if they do not give the justification deduct 5 pts

.

solution (*env.*) The `solution` environment can be to specify a solution to a problem. If the package option `solutions` is set or `\solutionstrue` is set in the text, then the solution will be presented in the output. The `solution` environment takes an optional KeyVal argument with the keys `id` for an identifier that can be reference `for` to specify which problem this is a solution for, and `height` that allows to specify the amount of space to be left in test situations (i.e. if the `test` option is set in the `\usepackage` statement).

hint (*env.*) The `hint` and `exnote` environments can be used in a `problem` environment to give hints
exnote (*env.*) and to make notes that elaborate certain aspects of the problem. The `gnote` (grading
gnote (*env.*) notes) environment can be used to document situations that may arise in grading.

\startsolutions Sometimes we would like to locally override the `solutions` option we have given to
\stopsolutions the package. To turn on solutions we use the `\startsolutions`, to turn them off, `\stopsolutions`. These two can be used at any point in the documents.

\ifsolutions Also, sometimes, we want content (e.g. in an exam with master solutions) conditional on whether solutions are shown. This can be done with the `\ifsolutions` conditional.

9.4.3 Markup for Added-Value Services

The `problem` package is all about specifying the meaning of the various moving parts of practice/exam problems. The motivation for the additional markup is that we can base added-value services from these, for instance auto-grading and immediate feedback.

The simplest example of this are multiple-choice problems, where the `problem` package allows to annotate answer options with the intended values and possibly feedback that can be delivered to the users in an interactive setting. In this section we will give some infrastructure for these, we expect that this will grow over time.

Multiple Choice Blocks

mcb (*env.*) Multiple choice blocks can be formatted using the `mcb` environment, in which single choices are marked up with `\mcc` macro.

`\mcc` `\mcc[⟨keyvals⟩]{⟨text⟩}` takes an optional key/value argument ⟨*keyvals*⟩ for choice metadata and a required argument ⟨*text*⟩ for the proposed answer text. The following keys are supported

- `T` for true answers, `F` for false ones,
- `Ttext` the verdict for true answers, `Ftext` for false ones, and
- `feedback` for a short feedback text given to the student.

What we see when this is formatted to PDF depends on the context. In solutions mode (we start the solutions in the code fragment below) we get

> **Example 41**
> Input:
>
> ```
>  1 \startsolutions
>  2 \begin{sproblem}[title=Functions,name=functions1]
>  3   What is the keyword to introduce a function definition in python?
>  4   \begin{mcb}
>  5     \mcc[T]{def}
>  6     \mcc[F,feedback=that is for C and C++]{function}
>  7     \mcc[F,feedback=that is for Standard ML]{fun}
>  8     \mcc[F,Ftext=Noooooooo,feedback=that is for Java]{public static void}
>  9   \end{mcb}
> 10 \end{sproblem}
> ```
>
> Output:

> **Problem 9.4.2 (Functions)**
> What is the keyword to introduce a function definition in python?
>
> - ☐ def
>   **Correct!**
> - ☐ function
>   **Wrong!** *that is for C and C++*
> - ☐ fun
>   **Wrong!** *that is for Standard ML*
> - ☐ public static void
>   **Wrong!** *that is for Java*

.

In "exam mode" where disable solutions (here via `\stopsolutions`)

Example 42

Input:

```
 1 \stopsolutions
 2 \begin{sproblem}[title=Functions,name=functions1]
 3   What is the keyword to introduce a function definition in python?
 4   \begin{mcb}
 5     \mcc[T]{def}
 6     \mcc[F,feedback=that is for C and C++]{function}
 7     \mcc[F,feedback=that is for Standard ML]{fun}
 8     \mcc[F,Ftext=Noooooooooo,feedback=that is for Java]{public static void}
 9   \end{mcb}
10 \end{sproblem}
```

Output:

> **Problem 9.4.3 (Functions)**
> What is the keyword to introduce a function definition in python?
>
> - ☐ def
> - ☐ function
> - ☐ fun
> - ☐ public static void

˙we get the questions without solutions (that is what the students see during the exam/quiz).

Filling-In Concrete Solutions

The next simplest situation, where we can implement auto-grading is the case where we have fill-in-the-blanks

The `\fillinsol` macro takes[6] an a single argument, which contains a concrete solution (i.e. a number, a string, . . .), which generates a fill-in-box in test mode:

Example 43

Input:

```
1    \stopsolutions
2    \begin{sproblem}[id=elefants.fillin,title=Fitting Elefants]
3      How many Elefants can you fit into a Volkswagen beetle? \fillinsol{4}
4    \end{sproblem}
```

Output:

> **Problem 9.4.4 (Fitting Elefants)**
>
> How many Elefants can you fit into a Volkswagen beetle?

and the actual solution in solutions mode:

Example 44

Input:

```
1    \begin{sproblem}[id=elefants.fillin,title=Fitting Elefants]
2      How many Elefants can you fit into a Volkswagen beetle? \fillinsol{4}
3    \end{sproblem}
```

Output:

> **Problem 9.4.5 (Fitting Elefants)**
> How many Elefants can you fit into a Volkswagen beetle?   !

If we do not want to leak information about the solution by the size of the blank we can also give `\fillinsol` an optional argument with a size: `\fillinsol[3cm]{12}` makes a box three cm wide.

Obviously, the required argument of `\fillinsol` can be used for auto-grading. For concrete data like numbers, this is immediate, for more complex data like strings "soft comparisons" might be in order. [7]

9.4.4 Including Problems

The `\includeproblem` macro can be used to include a problem from another file. It takes an optional KeyVal argument and a second argument which is a path to the file containing the problem (the macro assumes that there is only one problem in the include file). The keys `title`, `min`, and `pts` specify the problem title, the estimated minutes for solving the problem and the points to be gained, and their values (if given) overwrite the ones specified in the `problem` environment in the included file.

The sum of the points and estimated minutes (that we specified in the `pts` and `min` keys to the `problem` environment or the `\includeproblem` macro) to the log file and the

---

[7]EDNOTE: For the moment we only assume a single concrete value as correct. In the future we will almost certainly want to extend the functionality to multiple answer classes that allow different feedback like im MCQ. This still needs a bit of design. Also we want to make the formatting of the answer in solutions/test mode configurable.

screen after each run. This is useful in preparing exams, where we want to make sure that the students can indeed solve the problems in an allotted time period.

The `\min` and `\pts` macros allow to specify (i.e. to print to the margin) the distribution of time and reward to parts of a problem, if the `pts` and `pts` options are set. This allows to give students hints about the estimated time and the points to be awarded.

9.4.5 Testing and Spacing

The `problem` package is often used by the `hwexam` package, which is used to create homework assignments and exams. Both of these have a "test mode" (invoked by the package option `test`), where certain information –master solutions or feedback – is not shown in the presentation.

`\testspace` takes an argument that expands to a dimension, and leaves vertical space accordingly. Specific instances exist: `\testsmallspace`, `\testsmallspace`, `\testsmallspace` give small (1cm), medium (2cm), and big (3cm) vertical space.

`\testnewpage` makes a new page in `test` mode, and `\testemptypage` generates an empty page with the cautionary message that this page was intentionally left empty.

9.5 Homeworks, Quizzes and Exams

9.5.1 Introduction

The `hwexam` package and class supplies an infrastructure that allows to format nice-looking assignment sheets by simply including problems from problem files marked up with the `problem` package. It is designed to be compatible with `problems.sty`, and inherits some of the functionality.

9.5.2 Package Options

The `hwexam` package and class take the options `solutions`, `notes`, `hints`, `gnotes`, `pts`, `min`, and `boxed` that are just passed on to the `problems` package (cf. its documentation for a description of the intended behavior).

Furthermore, the `hwexam` package takes the option `multiple` that allows to combine multiple assignment sheets into a compound document (the assignment sheets are treated as section, there is a table of contents, etc.).

Finally, there is the option `test` that modifies the behavior to facilitate formatting tests. Only in `test` mode, the macros `\testspace`, `\testnewpage`, and `\testemptypage` have an effect: they generate space for the students to solve the given problems. Thus they can be left in the LaTeX source.

9.5.3   Assignments

This package supplies the `assignment` environment that groups problems into assignment sheets. It takes an optional KeyVal argument with the keys `number` (for the assignment number; if none is given, 1 is assumed as the default or — in multi-assignment documents — the ordinal of the `assignment` environment), `title` (for the assignment title; this is referenced in the title of the assignment sheet), `type` (for the assignment type; e.g. "quiz", or "homework"), `given` (for the date the assignment was given), and `due` (for the date the assignment is due).

9.5.4   Including Assignments

The `\inputassignment` macro can be used to input an assignment from another file. It takes an optional KeyVal argument and a second argument which is a path to the file containing the problem (the macro assumes that there is only one `assignment` environment in the included file). The keys `number`, `title`, `type`, `given`, and `due` are just as for the `assignment` environment and (if given) overwrite the ones specified in the `assignment` environment in the included file.

9.5.5   Typesetting Exams

The `\testheading` takes an optional keyword argument where the keys `duration` specifies a string that specifies the duration of the test, `min` specifies the equivalent in number of minutes, and `reqpts` the points that are required for a perfect grade.

```
1 \title{320101 General Computer Science (Fall 2010)}
2 \begin{testheading}[duration=one hour,min=60,reqpts=27]
3   Good luck to all students!
4 \end{testheading}
```

Will result in

Name: Matriculation Number:

320101 General Computer Science (Fall 2010)

2022-09-12

You have one hour (sharp) for the test;
Write the solutions to the sheet.
The estimated time for solving this exam is 60 minutes, leaving you
0 minutes for revising your exam.
You can reach 40 points if you solve all problems. You will only need
27 points for a perfect score, i.e. 13 points are bonus points.

You have ample time, so take it slow and avoid rushing to mistakes!

Different problems test different skills and knowledge, so do not get stuck on one problem.

prob.	9.4.1	9.4.2	9.4.3	9.4.4	9.4.5	1.1	2.1	2.2	2.3	3.1	3.2	3.3	Sum	grade
total	10					4	4	6	6	4	4	2	40	
reached														

To be used for grading, do not write here

good luck

References

[Bus+04]   Stephen Buswell et al. *The Open Math Standard, Version 2.0*. Tech. rep. The OpenMath Society, 2004. URL: http://www.openmath.org/standard/om20.

[CR99]   David Carlisle and Sebastian Rathz. *The* graphicxl *package*. Part of the TeX distribution. The Comprehensive TeX Archive Network. 1999. URL: https://www.tug.org/texlive/devsrc/Master/texmf-dist/doc/latex/graphics/graphicx.pdf.

[DCM03]   The DCMI Usage Board. *DCMI Metadata Terms*. DCMI Recommendation. Dublin Core Metadata Initiative, 2003. URL: http://dublincore.org/documents/dcmi-terms/.

[Koh06]   Michael Kohlhase. *OMDoc – An open markup format for mathematical documents [Version 1.2]*. LNAI 4180. Springer Verlag, Aug. 2006. URL: http://omdoc.org/pubs/omdoc1.2.pdf.

[LMH]   *LMH Scripts*. URL: https://github.com/sLaTeX/lmhtools.

[MMT]   *MMT – Language and System for the Uniform Representation of Knowledge*. Project web site. URL: https://uniformal.github.io/ (visited on 01/15/2019).

[MRK18]   Dennis Müller, Florian Rabe, and Michael Kohlhase. "Theories as Types". In: *9th International Joint Conference on Automated Reasoning*. Ed. by Didier Galmiche, Stephan Schulz, and Roberto Sebastiani. Springer Verlag, 2018. URL: https://kwarc.info/kohlhase/papers/ijcar18-records.pdf.

[Rab15]   Florian Rabe. "The Future of Logic: Foundation-Independence". In: *Logica Universalis* 10.1 (2015). 10.1007/s11787-015-0132-x; Winner of the Contest "The Future of Logic" at the World Congress on Universal Logic, pp. 1–20.

[RK13]   Florian Rabe and Michael Kohlhase. "A Scalable Module System". In: *Information & Computation* 0.230 (2013), pp. 1–54. URL: https://kwarc.info/frabe/Research/mmt.pdf.

[RT]   *sLaTeX/RusTeX*. URL: https://github.com/sLaTeX/RusTeX (visited on 04/22/2022).

[ST]   *sTeX – An Infrastructure for Semantic Preloading of LaTeX Documents*. URL: https://ctan.org/pkg/stex (visited on 04/22/2022).

[sTeX]   *sTeX: A semantic Extension of TeX/LaTeX*. URL: https://github.com/sLaTeX/sTeX (visited on 05/11/2020).

[Tana]   Till Tantau. *beamer – A LaTeX class for producing presentations and slides*. URL: http://ctan.org/pkg/beamer (visited on 01/07/2014).

[Tanb]   Till Tantau. *User Guide to the Beamer Class*. URL: http://ctan.org/macros/latex/contrib/beamer/doc/beameruserguide.pdf.

[TL]   *TeX Live*. URL: http://www.tug.org/texlive/ (visited on 12/11/2012).