LaTeXML *The Manual*

A LaTeX to XML/HTML/MathML Converter;
Version *0.8.7*

Bruce R. Miller

September 26, 2023

Contents

List of Figures

Chapter 1

Introduction

For many, LaTeX is the preferred format for document authoring, particularly those involving significant mathematical content and where quality typesetting is desired. On the other hand, content-oriented XML is an extremely useful representation for documents, allowing them to be used, and reused, for a variety of purposes, not least, presentation on the Web. Yet, the style and intent of LaTeX markup, as compared to XML markup, not to mention its programmability, presents difficulties in converting documents from the former format to the latter. Perhaps ironically, these difficulties can be particularly large for mathematical material, where there is a tendency for the markup to focus on appearance rather than meaning.

The choice of LaTeX for authoring, and XML for delivery were natural and uncontroversial choices for the Digital Library of Mathematical Functions[1]. Faced with the need to perform this conversion and the lack of suitable tools to perform it, the DLMF project proceeded to develop their own tool, LaTeXML, for this purpose.

Design Goals The idealistic goals are:

- Faithful emulation of TeX's behaviour;
- Easily extensible;
- Lossless, preserving both semantic and presentation cues;
- Use an abstract LaTeX-like, extensible, document type;
- Infer the semantics of mathematical content
  (*Good* Presentation MathML, eventually Content MathML and OpenMath).

As these goals are not entirely practical, even somewhat contradictory, they are implicitly modified by *as much as possible*. Completely mimicing TeX's, and LaTeX's, behaviour would seem to require the sneakiest modifications to TeX, itself; redefining LaTeX's internals does not really guarantee compatibility. "Ease of use" is, of course, in

[1] `https://dlmf.nist.gov`

the eye of the beholder; this manual is an attempt to make it easier! More significantly, few documents are likely to have completely unambiguous mathematics markup; human understanding of both the topic and the surrounding text is needed to properly interpret any particular fragment. Thus, while we'll try to provide a "turn-key" solution that does the 'Right Thing' automatically, we expect that applications requiring high semantic content will require document-specific declarations and tuning to achieve the desired result. Towards this end, we provide a variety of means to customize the processing and declare the author's intent. At the same time, especially for new documents, we encourage a more logical, content-oriented markup style, over a purely presentation-oriented style.

Overview of this Manual Chapter 2 describes the usage of LaTeXML, along with common use cases and techniques. Chapter 3 describes the system architecture in some detail. Strategies for customization and implementation of new packages is described in Chapter 4. The special considerations for mathematics, including details of representation and how to improve the conversion, are covered in Chapter 5. Several specialized topics are covered in the remaining chapters. An overview of outstanding issues and planned future improvements are given in Chapter 9.

Finally, the Appendices give detailed documentation the system components: Appendix A describes the command-line programs provided by the system; Appendix B lists the LaTeX style packages for which we've provided LaTeXML-specific bindings. Appendix C describes the various Perl modules, in groups, that comprise the system. Appendix D describes the XML schema used by LaTeXML. Appendix E gives an overview of the warning and error messages that LaTeXML may generate. Appendix F describes the strategy and naming conventions used for CSS styling of the resulting HTML.

Using LaTeXML, and programming for it, can be somewhat confusing as one is dealing with several languages not normally combined, often within the same file, — Perl, TeX and XML (along with XSLT, HTML, CSS), plus the occasional shell programmming. To help visually distinguish different contexts in this manual we will put 'programming' oriented material (Perl, TeX) in a typewriter font, `like this`; XML material will be put in a sans-serif face like this.

If you encounter difficulties, join the mailing list at `latexml-project`[2]. Bugs and enhancement requests can be reported at Github[3]. If all else fails, please consult the source code, or the author.

Danger! When you see this sign, be warned that the material presented is somewhat advanced and may not make much sense until you have dabbled quite a bit in LaTeXML's internals. Such advanced or 'dangerous' material will be presented like this paragraph to make it easier to skip over.

[2] `https://lists.informatik.uni-erlangen.de/mailman/listinfo/latexml`
[3] `https://github.com/brucemiller/LaTeXML`

Chapter 2

Using LaTeXML

The main commands provided by the LaTeXML system are

latexml for converting TeX and BibTeX sources to XML.

latexmlpost for various postprocessing tasks including conversion to HTML, processing images, conversion to MathML and so on.

latexmlc combines both `latexml` and `latexmlpost` into a single command, with some extra functionality.

The usage of these commands can be as simple as

```
latexmlc doc.tex --dest=doc.html
```

to convert a single document into HTML5 document, Or if you want to examine the XML, for some reason (I usually do)

```
latexml --dest=doc.xml doc ; latexmlpost doc --dest=doc.html
```

Or can be as complicated as

```
# Conversion
latexml --dest=main.xml main.tex
latexml --dest=A.xml    A
              ⋮
# Scan
latexmlpost --prescan --db=my.db --dest=/site/main.html main
latexmlpost --prescan --db=my.db --dest=/site/A.html    A
              ⋮
# Pagination
latexmlpost --noscan --db=my.db --dest=/site/main.html main
latexmlpost --noscan --db=my.db --dest=/site/A.html    A
              ⋮
```

to convert a whole set of documents, including a bibliography, into a complete interconnected site. See 2.4 for details.

How best to use the commands depends, of course, on what you are trying to achieve. In the next section, we'll describe the use of `latexml`, which performs the conversion to XML. The following sections consider a sequence of successively more complicated postprocessing situations, using `latexmlpost`, by which one or more TEX sources can be converted into one or more web documents or a complete site.

The command `latexmlc` can also create ePub documents:

```
latexmlc --dest=doc.epub doc
```

Additionally, there is a convenience command `latexmlmath` for converting individual formula into various formats.

2.1 Basic XML Conversion

The command

```
latexml {options} --destination=doc.xml doc
```

converts the TEX document `doc.tex`, or standard input if – is used in place of the filename, to XML. It loads any required definition bindings (see below), reads, tokenizes, expands and digests the document creating an XML structure. It then performs some document rewriting, parses the mathematical content and writes the result, in this case, to `doc.xml`; if no `--destination` is suppplied, it writes the result to standard output. For details on the processing, see Chapter 3, and Chapter 5 for more information about math parsing.

BIBTEX processing If the source file has an explicit extension of `.bib`, or if the `--bibtex` option is used, the source will be treated as a BIBTEX database. See 2.2 for how BIBTEX files are included in the final output.

Note that the timing is different than with BIBTEX and LATEX. Normally, BIBTEX simply selects and formats a subset of the bibliographic entries according to the `.aux` file; all TEX expansion and processing is carried out only when the result is included in the main LATEX document. In contrast, `latexml` processes and expands the entire bibliography, including any TEX markup within it, when it is converted to XML; the selection of entries is done during postprocessing. One implication is that latexml does not know about packages included in the main document; if the bibliography uses macros defined in such packages, the packages must be explicitly specified using the `--preload` option.

Useful Options The number and detail of progress and debugging messages printed during processing can be controlled using

```
--verbose or --quiet
```

They can be repeated to get even more or fewer details.

Directories to search (in addition to the working directory) for various files can be specified using

`--path={directory}`

This option can be repeated.

Whenever multiple sources are being used (including multiple bibliographies), the option

`--documentid=`*`id`*

should be used to provide a unique ID for the document root element. This ID is used as the base for id's of the child-elements within the document, so that they are unique, as well.

See the documentation for the command `latexml` for less common options.

Loading Bindings Although LaTeXML is reasonably adept at processing TeX macros, it generally benefits from having its own implementation of the macros, primitives, environments and other control sequences appearing in a document because these are what define the mapping into XML. The LaTeXML-analogue of a style or class file we call a LaTeXML-binding file, or *binding* for short; these files have an additional extension `.ltxml`.

In fact, since style files often bypass structurally or semantically meaningful macros by directly invoking macros internal to LaTeX, LaTeXML actually avoids processing style files when a binding is unavailable. The option

`--includestyles`

can be used to override this behaviour and allow LaTeXML to (attempt to) process raw style files. [A more selective, per-file, option may be developed in the future, if there is sufficient demand — please provide use cases.]

LaTeXML always starts with the `TeX.pool` binding loaded, and if LaTeX-specific commands are recognized, `LaTeX.pool` as well. Any input directives within the source loads the appropriate binding. For example, `\documentclass{article}` or `\usepackage{graphicx}` will load the bindings `article.cls.ltxml` or `graphicx.sty.ltxml`, respectively; the obsolete directive `\documentstyle` is also recognized. An `\input` directive will search for files with both `.tex` and `.sty` extensions; it will prefer a binding file if one is found, but will load and digest a `.tex` if no binding is found. An `\include` directive (and related ones) search only for a `.tex` file, which is processed and digested as usual.

There are two mechanisms for customization: a document-specific binding file *`doc`*`.latexml` will be loaded, if present; the option

`--preload=`*`binding`*

will load the binding file *`binding`*`.ltxml`. The `--preload` option can be repeated; both kinds of preload are loaded before document processing, and are processed in order.

See Chapter 4 for details about what can go in these bindings; and Appendix B for a list of bindings currently included in the distribution.

2.2 Basic Postprocessing

In the simplest situation, you have a single TEX source document from which you want to generate a single output document. The command

latexmlpost *options* --destination=*doc*.html *doc*

or similarly with --destination=*doc*.html4, --destination=*doc*.xhtml, will carry out a set of appropriate transformations in sequence:

- scanning of labels and ids;
- filling in the index and bibliography (if needed);
- cross-referencing;
- conversion of math;
- conversion of graphics and picture environments to web format (png);
- applying an XSLT stylesheet.

The output format affects the defaults for each step, and particularly, the XSLT stylesheet that is used, and is determined by the file extension of --destination, or by the option

--format=(html|html5|html4|xhtml|xml)

which overrides the extension used in the destination. The recognized formats are:

html or html5 math is converted to Presentation MATHML, some 'vector' style graphics are converted to SVG, other graphics are converted to images; LaTeXML-html5.xslt is used. The file extension html generates html5

html4 both math and graphics are converted to images; LaTeXML-html4.xslt is used.

xhtml math is converted to Presentation MATHML, other graphics are converted to images; LaTeXML-xhtml.xslt is used.

xml no math, graphics or XSLT conversion is carried out.

Of course, all of these conversions can be controlled or overridden by explicit options described below. For more details about less common options, see the command documentation latexmlpost, as well as Appendix C.5.

Scanning The scanning step collects information about all labels, ids, indexing commands, cross-references and so on, to be used in the following postprocessing stages.

Indexing An index is built from `\index` markup, if `makeidx`'s `\printindex` command has been used, but this can be disabled by

`--noindex`

The index entries can be permuted with the option

`--permutedindex`

Thus `\index{term a!term b}` also shows up as `\index{term b!term a}`. This leads to a more complete, but possibly rather silly, index, depending on how the terms have been written.

Bibliography When a document contains a request for bibliographies, typically due to the `\bibliography{..}` command, the postprocessor will look for the named bibliographies. It first looks for preconverted bibliographies with the extention `.bib.xml`, otherwise it will look for `.bib` and convert it internally (the latter is a somewhat experimental feature).

If you want to override that search, for example using a bibliography with a different name, you can supply that filename using the option

`--bibliography=`*`bibfile`*`.bib.xml`

Note that the internal bibliography list will then be ignored. The bibliography would have typically been produced by running

`latexml` `--dest=bibfile.bib.xml bibfile.bib`

Note that the XML file, bibfile, is not used to directly produce an HTML-formatted bibliography, rather it is used to fill in the `\bibliography{..}` within a TEX document.

Cross-Referencing In this stage, the scanned information is used to fill in the text and links of cross-references within the document. The option

`--urlstyle=(server|negotiated|file)`

can control the format of urls with the document.

server formats urls appropriate for use from a web server. In particular, trailing `index.html` are omitted. (default)

negotiated formats urls appropriate for use by a server that implements content negotiation. File extensions for `html` and `xhtml` are omitted. This enables you to set up a server that serves the appropriate format depending on the browser being used.

file formats urls explicitly, with full filename and extension. This allows the files to be browsed from the local filesystem.

Math Conversion Specific conversions of the mathematics can be requested using the options

```
--mathimages                    # converts math to png images,
--presentationmathml or --pmml  # creates Presentation MathML
--contentmathml or --cmml       # creates Content MathML
--openmath or --om              # creates OpenMath
--keepXMath                     # preserves LaTeXML's XMath
```

(Each of these options can also be negated if needed, eg. `--nomathimages`) It must be pointed out that the Content MATHML and OpenMath conversions are currently rather experimental.

If more than one of these conversions are requested, parallel math markup will be generated with the first format being the primary one, and the additional ones added as secondary formats. The secondary format is incorporated using whatever means the primary format uses; eg. MATHML combines formats using `m:semantics` and `m:annotation-xml`.

Given the state of current browsers, you may wish to use a polyfill such as MathJax[1] to support MathML on more platforms. See the example in 2.2 for one way to do it.

Graphics processing Conversion of graphics (eg. from the `graphic(s|x)` packages' `\includegraphics`) can be enabled or disabled using

```
--graphicsimages or --nographicsimages
```

Similarly, the conversion of `picture` environments can be controlled with

```
--pictureimages or --nopictureimages
```

An experimental capability for converting the latter to SVG can be controlled by

```
--svg or --nosvg
```

Stylesheets and Javascript If you wish to restyle the generated HTML either by adding CSS or by customizing the XSLT, change its functionality by adding javascript, or even generate an alternative output format with XSLT, some combination of the following options will be useful.

```
--nodefaultresources          # Omits the default resources (css..)
--css=stylesheet.css          # Adds a new CSS stylesheet
--javascript=program.js       # Adds a Javascript
--stylesheet=stylesheet.xsl # Uses an alternative XSLT stylesheet
--xsltparameter=name:value    # Sets an XSLT parameter
```

All but `--stylesheet` can be repeated to include multiple files or set multiple parameters. When a local CSS or javascript file is included, it will be copied to the destination directory, but otherwise urls are accepted.

The core CSS stylesheet, `LaTeXML.css`, along with certain styles or classes (`article`, `report`, `book`, `amsart`) which add stylesheets automatically, helps

[1] `https://mathjax.org/`

match the styling of LaTeX to HTML. You can also request the inclusion of your own stylesheets from the commandline using `--css` option. Some sample CSS enhancements are included with the distribution:

`LaTeXML-navbar-left.css` Places a navigation bar on the left.

`LaTeXML-navbar-right.css` Places a navigation bar on the left.

`LaTeXML-blue.css` Colors various features in a soft blue.

In cases where you wish to completely manage the CSS the option `--nodefaultcss` causes only explicitly requested (command-line) css files to be included.

Javascript files are included in the generated HTML by using the `--javascript` option. The distribution includes a sample `LaTeXML-maybeMathjax.js` which is useful for supporting MathML: it invokes MathJax[2] to render the mathematics in browsers without native support for MathML.

```
--javascript=LaTeXML-maybeMathjax.js
```

The option can also reference a remote script; for example to invoke MathJax unconditionally from the 'cloud':

```
latexmlpost --format=html5 \
   --javascript='https://cdnjs.cloudflare.com/ajax/libs/mathjax/2.7.2/MathJax.js?config=MML_CH
   --destination=somewhere/doc.html doc
```

See 4.2.2 for more information on developing your own stylesheets. To develop CSS and XSLT stylesheets, a knowledge of the LaTeXML document type is also necessary; see Appendix D.

Individual XSLT stylesheets may have parameters that can customize the conversion from LaTeXML's XML to the target format. An obscure example is

```
--xsltparameter=SIMPLIFY_HTML:true
```

which causes a 'simpler' HTML to be generated. Generally, LaTeXML's HTML relies on CSS to recreate the appearance of many features of LaTeX, but this sometimes results in somewhat convoluted HTML that may not be ideal in situations where CSS is not available. This parameter 'dumbs down' itemizations and enumerations by ignoring any custom item labels or numbers.

2.3 Splitting the Output

For larger documents, it is often desirable to break the result into several interlinked pages. This split, carried out before scanning, is requested by

```
--splitat=level
```

where *level* is one of `chapter`, `section`, `subsection`, or `subsubsection`. For example, `section` would split the document into chapters (if any) and sections,

[2] https://mathjax.org

along with separate bibliography, index and any appendices. (See also `--splitpath` in `latexml`.) The removed document nodes are replaced by a Table of Contents.

The extra files are named using either the id or label of the root node of each new page document according to

`--splitnaming=(id|idrelative|label|labelrelative)`

The relative foms create shorter names in subdirectories for each level of splitting. (See also `--urlstyle` and `--documentid` in `latexml`.)

Additionally, the index and bibliography can be split into separate pages according to the initial letter of entries by using the options

`--splitindex` *and* `--splitbibliography`

2.4 Site processing

A more complicated situation combines several TEX sources into a single interlinked site consisting of multiple pages and a composite index and bibliography.

Conversion First, all TEX sources must be converted to XML, using `latexml`. Since every target-able element in all files to be combined must have a unique identifier, it is useful to prefix each identifier with a unique value for each file. The `latexml` option `--documentid=`*id* provides this.

Scanning Secondly, all XML files must be split and scanned using the command

`latexmlpost` `--prescan --dbfile=`*DB* `--dest=`*i*`.html` *i*

where *DB* names a file in which to store the scanned data. Other conversions, including writing the output file, are skipped in this prescanning step.

Pagination Finally, all XML files are cross-referenced and converted into the final format using the command

`latexmlpost` `--noscan --dbfile=`*DB* `--dest=`*i*`.html` *i*

which skips the unnecessary scanning step.

For example, consider a set of nominally stand-alone LATEX documents: `main` (with title page, `\tableofcontents`, etc), `A` (with a chapter), `Aa` (with a section), `B` (with a chapter), ... and `bib` (with a `\bibliography`). Assume that the documents use `\lxDocumentID` from `\usepackage{latexml}` to declare ids `main`, `main.A`, `\main.A.a`, `main.B`, ...`bib`, respectively. And, of course, you'll have to arrange for appropriate counters to be initialized appropriately, if needed.

Now, processing the documents with the following commands

```
# Conversion
latexml --dest=main.xml main.tex
latexml --dest=A.xml    A
latexml --dest=Aa.xml   Aa
```

```
latexml --dest=B.xml    B
              ⋮
latexml --dest=bib.xml  bib
# Scan
latexmlpost --prescan --db=my.db --dest=/site/main.html main
latexmlpost --prescan --db=my.db --dest=/site/A.html    A
latexmlpost --prescan --db=my.db --dest=/site/Aa.html   Aa
latexmlpost --prescan --db=my.db --dest=/site/B.html    B
              ⋮
latexmlpost --prescan --db=my.db --dest=bib.html        bib
# Pagination
latexmlpost --noscan --db=my.db --dest=/site/main.html main
latexmlpost --noscan --db=my.db --dest=/site/A.html    A
latexmlpost --noscan --db=my.db --dest=/site/Aa.html   Aa
latexmlpost --noscan --db=my.db --dest=/site/B.html    B
              ⋮
latexmlpost --noscan --db=my.db --dest=bib.html        bib
```

This will result in a site built at `/site/`, with the following implied structure:

```
main.html
  A.html
    Aa.html
  B.html
    ...
  bib.html
```

2.5 Individual Formula

For cases where you'd just like to convert a single formula to, say, MATHML, and don't mind the overhead, we've combined the pre- and post-processing into a single, handy, command `latexmlmath`. For example,

```
latexmlmath --pmml=- \\frac{b\\pm\\sqrt{b^2-4ac}}{2a}
```

will print the MATHML to standard output. To convert the formula to a `png` image, say `quad.png`, use the option `--mathimage=quad.png`.

Note that this involves putting TEX code on the command line. You've got to 'slashify' your code in whatever way is necessary so that after your shell is finished with it, the string that is passed to `latexmlmath` sees is normal TEX. In the example above, in most unix-like shells, we only needed to double-up the backslashes.

Chapter 3

Architecture

As has been said, LaTeXML consists of two main programs: `latexml` responsible for converting the TeX source into XML; and `latexmlpost` responsible for converting to target formats. See Figure 3.1 for illustration.

The casual user needs only a superficial understanding of the architecture. The programmer who wants to extend or customize LaTeXML will, however, need a fairly good understanding of the process and the distinctions between text, Tokens, Boxes, Whatsits and XML, on the one hand, and Macros, Primitives and Constructors, on the other. In a way, the implementer of a LaTeXML binding for a LaTeX package may need a *better* understanding than when implementing for LaTeX since they have to understand not only the TeX-view, primarily just the macros and the intended appearance, but also the LaTeXML-view, with XML and representation questions, aw well.

The intention is that all semantics of the original document is preserved by `latexml`, or even inferred by parsing; `latexmlpost` is for formatting and conversion. Depending on your needs, the LaTeXML document resulting from `latexml` may be sufficient. Alternatively, you may want to enhance the document by applying third party programs before postprocessing.

3.1 latexml architecture

Like TeX, `latexml` is data-driven: the text and executable control sequences (ie. macros and primitives) in the source file (and any packages loaded) direct the processing. For LaTeXML, the user exerts control over the conversion, and customizes it, by providing alternative bindings of the control sequences and packages, by declaring properties of the desired document structure, and by defining rewrite rules to be applied to the constructed document tree.

The top-level class, `LaTeXML`, manages the processing, providing several methods for converting a TeX document or string into an XML document, with varying degrees of postprocessing and writing the document to file. It binds a `(LaTeXML::Core::)State` object (to `$STATE`)to maintain the current state of bindings for control sequence definitions and emulates TeX's scoping rules. The processing is broken

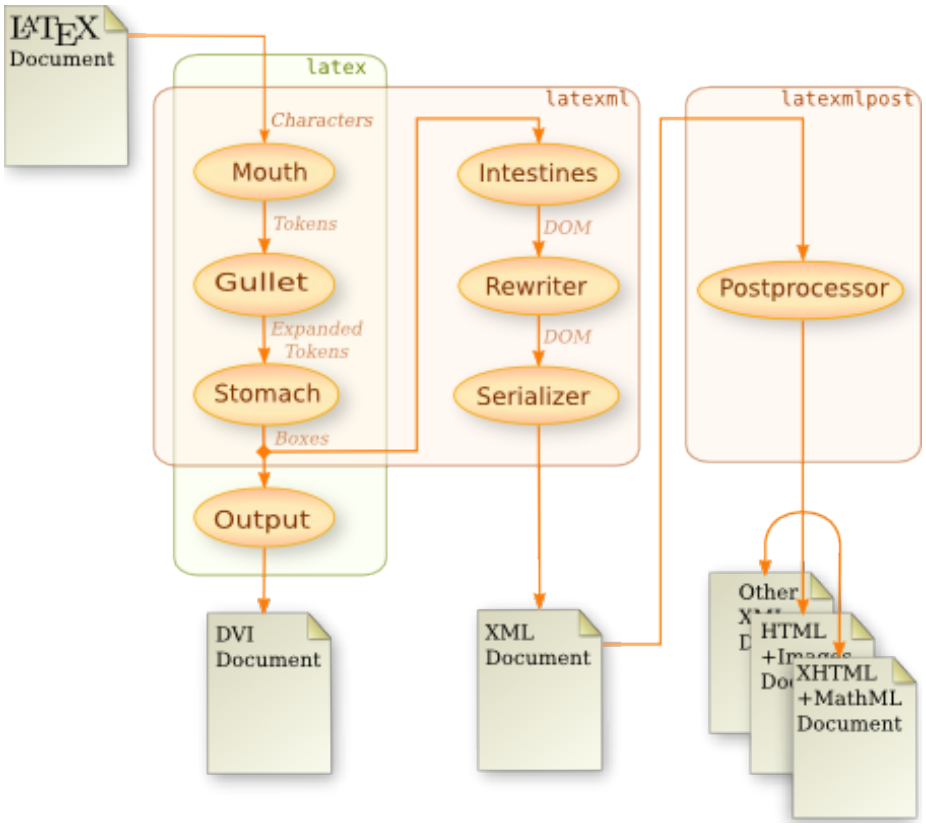

Figure 3.1: Flow of data through LaTeXML's digestive tract.

into the following stages

Digestion the TeX-like digestion phase which converts the input into boxes.

Construction converts the resulting boxes into an XML DOM.

Rewriting applies rewrite rules to modify the DOM.

Math Parsing parses the tokenized mathematics.

Serialization converts the XML DOM to a string, or writes to file.

Digestion Digestion is carried out primarily in a *pull* mode: The `(LaTeXML::Core::)Stomach` pulls expanded `(LaTeXML::Core::)Token`s from the `(LaTeXML::Core::)Gullet`, which itself pulls `Token`s from the `(LaTeXML::Core::)Mouth`. The `Mouth` converts characters from the plain text input into `Token`s according to the current *catcodes* (category codes) assigned to them (as

bound in the State). The Gullet is responsible for expanding Macros, that is, control sequences currently bound to (LaTeXML::Core::Definition::)Expandables and for parsing sequences of tokens into common core datatypes ((LaTeXML::Common::)Number, (LaTeXML::Common::)Dimension, etc.). See 4.1.1 for how to define macros and affect expansion.

The Stomach then digests these tokens by executing (LaTeXML::Core::Definition::)Primitive control sequences, usually for side effect, but often for converting material into (LaTeXML::Core::)Lists of (LaTeXML::Core::)Boxes and (LaTeXML::Core::)Whatsits (A Macro should never digest). Normally, textual tokens are converted to Boxes in the current font. The main (intentional) deviation of LaTeXML's digestion from that of TeX is the introduction of a new type of definition, a (LaTeXML::Core::Definition::)Constructor, responsible for constructing XML fragments. A control sequence bound to Constructor is digested by reading and processing its arguments and wrapping these up in a Whatsit. Before- and after-daemons, essentially anonymous primitives, associated with the Constructor are executed before and after digesting the Constructor arguments' markup, which can affect the context of that digestion, as well as augmenting the Whatsit with additional properties. See 4.1.2 for how to define primitives and affect digestion.

Construction Given the List of Boxes and Whatsits, we proceed to constructing an XML document. This consists of creating an (LaTeXML::Core::)Document object, containing a libxml2 document, XML::LibXML::Document, and having it absorb the digested material. Absorbing a Box converts it to text content, with provision made to track and set the current font. A Whatsit is absorbed by invoking the associated Constructor to insert an appropriate XML fragment, including elements and attributes, and recursively processing their arguments as necessary See 4.1.3 for how to define constructors.

A (LaTeXML::Common::)Model is maintained througout the digestion phase which accumulates any document model declarations, in particular the document type (RelaxNG is preferred, but DTD is also supported). As LaTeX markup is more like SGML than XML, additional declarations may be used (see Tag in (LaTeXML::)Package) to indicate which elements may be automatically opened or closed when needed to build a document tree that matches the document type. As an example, a <subsection> will automaticall be closed when a <section> is begun. Additionally, extra bits of code can be executed whenever particularly elements are openned or closed (also specified by Tag). See 4.1.4 for how to affect the schema.

Rewriting Once the basic document is constructed, (LaTeXML::Core::)Rewrite rules are applied which can perform various functions. Ligatures and combining mathematics digits and letters (in certain fonts) into composite math tokens are handled this way. Additionally, declarations of the type or grammatical role of math tokens can be applied here See 4.1.5 for how to define rewrite rules.

MathParsing After rewriting, a grammar based parser is applied to the mathematical nodes in order to infer, at least, the structure of the expressions, if not the meaning. Mathematics parsing, and how to control it, is covered in detail in Chapter 5.

Serialization Here, we simple convert the DOM into string form, and output it.

3.2 latexmlpost architecture

LaTeXML's postprocessor is primarily for format conversion. It operates by applying a sequence of filters responsible for transforming or splitting documents, or their parts, from one format to another.

Exactly which postprocessing filter modules are applied depends on the command-line options to `latexmlpost`. Postprocessing filter modules are generally applied in the following order:

Split splits the document into several 'page' documents, according to `--split` or `--splitpath` options.

Scan scans the document for all ID's, labels and cross-references. This data may be stored in an external database, depending on the `--db` option.

MakeIndex fills in the index element (due to a `\printindex`) with material generated by `index`.

MakeBibliography fills in the bibliography element (from `\bibliography`) with material extracted from the file specified by the `--bibilography` option, for all `\cite`'d items.

CrossRef establishes all cross-references between documents and parts thereof, filling in the references with appropriate text for the hyperlink.

MathImages, MathML, OpenMath performs various conversions of the internal Math representation.

PictureImages, Graphics, SVG performs various graphics conversions.

XSLT applies an XSLT transformation to each document.

Writer writes the document to a file in the appropriate location.

See 4.2 for how to customize the postprocessing.

Chapter 4

Customization

The processsing of the LaTeX document, its conversion into XML and ultimately to XHTML or other formats can be customized in various ways, at different stages of processing and in different levels of complexity. Depending on what you are trying to achieve, some approaches may be easier than others: Recall Larry Wall's adage "There's more than one way to do it."

By far, the easiest way to customize the style of the output is by modifying the CSS, see 4.2.2, so that is the recommended way when it applies.

The basic conversion from TeX markup to XML is done by `latexml`, and is obviously affected by the mapping between the TeX markup and the XML markup. This mapping is defined by macros, primitives and, of course, constructors; The mapping that is in force at any time is determined by the LaTeXML-specific implementations of the TeX packages involved, what we call 'bindings'. Consequently, you can customize the conversion by modifying the bindings used by `latexml`.

Likewise, you extend `latexml` by creating bindings for TeX styles that hadn't been covered.

Or by defining your own TeX style file along with it's LaTeXML binding.

In all these cases, you'll need the same skills: understanding and using text, tokens, boxes and whatsits, as well as macros and macro expansion, primitives and digestion, and finally whatsits and constructors. Understanding TeX helps; reading the LaTeXML bindings in the distribution will give an idea of how we use it. To teach LaTeXML about new macros, to implement bindings for a package not yet covered, or to modify the way TeX control sequences are converted to XML, you will want to look at 4.1. To modify the way that XML is converted to other formats such as HTML, see 4.2.

A particularly powerful strategy when you have control over the source documents is to develop a semantically oriented LaTeX style file, say `smacros.sty`, and then provide a LaTeXML binding as `smacros.sty.ltxml`. In the LaTeX version, you may style the terms as you like; in the LaTeXML version, you could control the conversion so as to preserve the semantics in the XML. If LaTeXML's schema is insufficient, then you would need to extend it with your own representation; although that is beyond the scope of the current manual, see the discussion below in 4.1.4. In such a case, you would also need to extend the XSLT stylesheets, as discussed in 4.2.1.

4.1 LaTeXML Customization

This layer of customization deals with modifying the way a LaTeX document is transformed into LaTeXML's XML, primarily through defining the way that control sequences are handled. In 2.1 the loading of various bindings was described. The facilities described in the following subsections apply in all such cases, whether used to customize the processing of a particular document or to implement a new LaTeX package. We make no attempt to be comprehensive here; please consult the documentation for `(LaTeXML::)Global` and `Package`, as well as the binding files included with the system for more guidance.

A LaTeXML binding is actually a Perl module, and as such, a familiarity with Perl is helpful. A binding file will look something like:

```
use LaTeXML::Package;
use strict;
use warnings;
# Your code here!

1;
```

The final '1' is required; it tells Perl that the module has loaded successfully. In between, comes any Perl code you wish, along with the definitions and declarations as described here.

Actually, familiarity with Perl is more than merely helpful, as is familiarity with TeX and XML! When writing a binding, you will be programming with all three languages. Of course, you need to know the TeX corresponding to the macros that you intend to implement, but sometimes it is most convenient to implement them completely, or in part, in TeX, itself (eg. using **`DefMacro`**), rather then in Perl. At the other end, constructors (eg. using **`DefConstructor`**) are usually defined by patterns of XML.

4.1.1 Expansion & Macros

`DefMacro(`*`$prototype`*`,`*`$replacement`*`,`*`%options`*`)` Macros are defined using `DefMacro`, such as the pointless:

```
DefMacro('\mybold{}','\textbf{#1}');
```

The two arguments to `DefMacro` we call the *prototype* and the *replacement*. In the prototype, the `{}` specifies a single normal TeX parameter. The replacement is here a string which will be tokenized and the `#1` will be replaced by the tokens of the argument. Presumably the entire result will eventually be further expanded and or processed.

Whereas, TeX normally uses `#1`, and LaTeX has developed a complex scheme where it is often necessary to peek ahead token by token to recognize optional arguments, we have attempted to develop a suggestive, and easier to use, notation for parameters. Thus a prototype `\foo{}` specifies a single normal argument, wheere `\foo[]{}` would take an optional argument followed by a required one. More complex argument

prototypes can be found in Package. As in TEX, the macro's arguments are neither expanded nor digested until the expansion itself is further expanded or digested.

The macro's replacement can also be Perl code, typically an anonymous sub, which gets the current Gullet followed by the macro's arguments as its arguments. It must return a list of Token's which will be used as the expansion of the macro. The following two examples show alternative ways of writing the above macro:

```
DefMacro('\mybold{}', sub {
  my($gullet,$arg)=@_;
  (T_CS('\textbf'),T_BEGIN,$arg,T_END); });
```

or alternatively

```
DefMacro('\mybold{}', sub {
  Invocation(T_CS('\textbf'),$_[1]); });
```

Generally, the body of the macro should *not* involve side-effects, assignments or other changes to state other than reading Token's from the Gullet; of course, the macro may expand into control sequences which do have side-effects.

Tokens, Catcodes and friends Functions that are useful for dealing with Tokens and writing macros include the following:

- Constants for the corresponding TEX catcodes:

  ```
  CC_ESCAPE, CC_BEGIN,  CC_END,     CC_MATH,
  CC_ALIGN,  CC_EOL,    CC_PARAM,   CC_SUPER,
  CC_SUB,    CC_IGNORE, CC_SPACE,   CC_LETTER,
  CC_OTHER,  CC_ACTIVE, CC_COMMENT, CC_INVALID
  ```

- Constants for tokens with the appropriate content and catcode:

  ```
  T_BEGIN, T_END,   T_MATH,  T_ALIGN, T_PARAM,
  T_SUB,   T_SUPER, T_SPACE, T_CR
  ```

- **T_LETTER**($char), **T_OTHER**($char), **T_ACTIVE**($char), create tokens of the appropriate catcode with the given text content.

- **T_CS**($cs) creates a control sequence token; the string $cs should typically begin with the slash.

- **Token**($string,$catcode) creates a token with the given content and catcode.

- **Tokens**($token,...) creates a (LaTeXML::Core::)Tokens object containing the list of Tokens.

- **Tokenize**($string) converts the string to a Tokens, using TEX's standard catcode assignments.

- **TokenizeInternal**($string) like Tokenize, but treating @ as a letter.

- **Explode**(`$string`) converts the string to a `Tokens` where letter character are given catcode **CC_OTHER**.
- **Expand**(`$tokens` expands `$tokens` (a `Tokens`), returning a `Tokens`; there should be no expandable tokens in the result.
- **Invocation**(`$cstoken,$arg,...`) Returns a `Tokens` representing the sequence needed to invoke `$cstoken` on the given arguments (each are `Tokens`, or undef for an unsupplied optional argument).

4.1.2 Digestion & Primitives

Primitives are processed during the digestion phase in the `Stomach`, after macro expansion (in the `Gullet`), and before document construction (in the `Document`). Our primitives generalize TEX's notion of primitive; they are used to implement TEX's primitives, invoke other side effects and to convert Tokens into Boxes, in particular, Unicode strings in a particular font.

Here are a few primitives from `TeX.pool`:

```
DefPrimitive('\begingroup',sub {
  $_[0]->begingroup; });
DefPrimitive('\endgroup',  sub {
  $_[0]->endgroup; });
DefPrimitiveI('\batchmode',      undef,undef);
DefPrimitiveI('\OE', undef, "\x{0152}");
DefPrimitiveI('\tiny',         undef, undef,
  font=>{size=>5});
```

Other than for implementing TEX's own primitives, `DefPrimitive` is needed less often than `DefMacro` or `DefConstructor`. The main thing to keep in mind is that primitives are processed after macro expansion, by the `Stomach`. They are most useful for side-effects, changing the `State`.

DefPrimitive(*$prototype*,*$replacement*,%options) The replacement is either a string which will be used to create a Box in the current font, or can be code taking the `Stomach` and the control sequence arguments as argument; like macros, these arguments are not expanded or digested by default, they must be explicitly digested if necessary. The replacement code must either return nothing (eg. ending with **return;**) or should return a list (ie. a Perl list `(...)`) of digested `Box`es or `Whatsit`s.

Options to DefPrimitive are:

- `mode=>('math'|'text')` switches to math or text mode, if needed;
- `requireMath=>1`, `forbidMath=>1` requires, or forbids, this primitive to appear in math mode;
- `bounded=>1` specifies that all digestion (of arguments and daemons) will take place within an implicit TEX group, so that any side-effects are localized, rather than affecting the global state;

- font=>{*hash*} switches the font used for any created text; recognized font keys are family, series, shape, size, color;

  Note that if the font change should only affect the material digested within this command itself, then bounded=>1 should be used; otherwise, the font change will remain in effect after the command is processed.

- beforeDigest=>CODE($stomach), afterDigest=>CODE($stomach) provides code to be digested before and after processing the main part of the primitive.

DefRegister(...) Needs descrition!

Other Utilities for Digestion Other functions useful for dealing with digestion and state are important for writing before & after daemons in constructors, as well as in Primitives; we give an overview here:

- **Digest**($tokens) digests $tokens (a (LaTeXML::Core::)Tokens), returning a list of Boxes and Whatsits.

- **Let**($token1,$token2) gives $token1 the same meaning as $token2, like \let.

Bindings The following functions are useful for accessing and storing information in the current State. It maintains a stack-like structure that mimics TEX's approach to binding; braces { and } open and close stack frames. (The Stomach methods bgroup and egroup can be used when explicitly needed.)

- **LookupValue**($symbol),**AssignValue**($string,$value,$scope) maintain arbitrary values in the current State, looking up or assigning the current value bound to $symbol (a string). For assignments, the $scope can be 'local' (the default, if $scope is omitted), which changes the binding in the current stack frame. If $scope is 'global', it assigns the value globally by undoing all bindings. The $scope can also be another string, which indicates a named scope — but that is a more advanced topic.

- **PushValue**($symbol,$value,...),**PopValue**($symbol), **UnshiftValue**($symbol,$value,...), **ShiftValue**($symbol) These maintain the value of $symbol as a list, with the operatations having the same sense as in Perl; modifications are always global.

- **LookupCatcode**($char), **AssignCatcode**($char,$catcode,$scope) maintain the catcodes associated with characters.

- **LookupMeaning**($token), **LookupDefinition**($token) looks up the current meaning of the token, being any executable definition bound for it. If there is no such defniition LookupMeaning returns the token itself, LookupDefinition returns undef.

Counters The following functions maintain LaTeX-like counters, and generally also associate an `ID` with them. A counter's print form (ie. `\theequation` for equations) often ends up on the refnum attribute of elements; the associated `ID` is used for the xml:id attribute.

- **`NewCounter`**`($name,$within,`*`options`*`)`, creates a LaTeX-style counters. When `$within` is used, the given counter will be reset whenever the counter `$within` is incremented. This also causes the associated `ID` to be prefixed with `$within`'s `ID`. The option `idprefix=>$string` causes the `ID` to be prefixed with that string. For example,

  ```
  NewCounter('section', 'document', idprefix=>'S');
  NewCounter('equation','document', idprefix=>'E',
    idwithin=>'section');
  ```

  would cause the third equation in the second section to have ID='S2.E3'.

- **`CounterValue`**`($name)` returns the `Number` representing the current value.

- **`ResetCounter`**`($name)` resets the counter to 0.

- **`StepCounter`**`($name)` steps the counter (and resets any others 'within' it), and returns the expansion of `\the$name`.

- **`RefStepCounter`**`($name)` steps the counter and any ID's associated with it. It returns a hash containing `refnum` (expansion of `\the$name`) and `id` (expansion of `\the$name@ID`)

- **`RefStepID`**`($name)` steps the ID associated with the counter, without actually stepping the counter; this is useful for unnumbered units that normally would have both a refnum and ID.

4.1.3 Construction & Constructors

Constructors are where things get interesting, but also complex; they are responsible for defining how the XML is built. There are basic constructors corresponding to normal control sequences, as well as environments. Mathematics generally comes down to constructors, as well, but is covered in Chapter 5.

Here are a couple of trivial examples of constructors:

```
DefConstructor('\emph{}',
 "<ltx:emph>#1</ltx:emph>", mode=>'text');
DefConstructor('\item[]',
  "<ltx:item>?#1(<ltx:tag>#1</ltx:tag>)");
DefEnvironment('{quote}',
  '<ltx:quote>#body</ltx:quote>',
  beforeDigest=>sub{ Let('\\\\','\@block@cr');});
DefConstructor('\footnote[]{}',
  "<ltx:note␣class='footnote'␣mark='#refnum'>#2</ltx:note>",
  mode=>'text',
```

```
properties=> sub {
  ($_[1] ? (refnum=>$_[1]) : RefStepCounter('footnote')) });
```

DefConstructor(*$prototype*,*$replacement*,*%options*) Creates a constructor control sequence, where `$replacement` describes the XML to be generated during the construction phase. It can either be a string representing the *XML pattern* (described below), or a subroutine `CODE($document,$arg1,...props)` receiving the arguments and properties from the `Whatsit`; it would invoke the methods of `Document` to construct the desired XML.

At its simplest, the XML pattern is a just serialization of the desired XML. For more expressivity, XML trees, text content, attributes and attribute values can be effectively 'interpolated' into the XML being constructed by use of the following expressions:

- *#1*,*#2*,... *#%name%* returns the construction of the numbered argument or named property of the Whatsit;
- `&`*function*`(`*arg1*`,`*arg2*`,...)` invokes the Perl *function* on the given arguments, *arg1*,..., returning the result. The arguments should be expressions for values, rather than XML subtrees.
- `?`*test*`(`*if pattern*`)` or `?`*test*`(`*if pattern*`)(`*else pattern*`)` returns the result of either the *if* or *else* pattern depending on whether the result of *test* (typically also an expression) is non-empty;
- `%`*expression* returns a hash (or rather assumes the result is a hash or KeyVals object); this is only allowed within an opening XML tag, where all the key-value pairs are inserted as attributes;
- `^` if this appears at the beginning of the pattern, the replacement is allowed to *float* up the current tree to whereever it might be allowed;

In each case, the result of an expression is expected to be either an XML tree, a string or a hash, depending on the context it was used in. In particular, values of attributes are typically given by quoted strings, but expressions within those strings are interpolated into the computed attribute value. The special characters `@ # ? %` which introduce these expressions can be escaped by preceding with a backslash, when the literal character is desired.

A subroutine used as the `$replacement`, allows programmatic insertion of XML into, or modification of, the document being constructed. Although one could use LibXML's DOM API to manipulate the document tree, it is *strongly* recommended to use `Document`'s API whereever possible as it maintains consistency and manages namespace prefixes. This is particularly true for insertion of new content, setting attributes and finding existing nodes in the tree using XPath.

Options:

- `mode=>('math'|'text')` switches to math or text mode, if needed;
- `requireMath=>1`, `forbidMath=>1` requires, or forbids, this constructor to appear in math mode;

- `bounded=>1` specifies that all digestion (of arguments and daemons) will take place within an implicit TEX group, so that any side-effects are localized, rather than affecting the global state;
- `font=>{hash}` switches the font used for any created text; recognized font keys are `family`, `series`, `shape`, `size`, `color`;
- `properties=> {hash} | CODE($stomach,$arg1,..)`. provides a set of properties to store in the `Whatsit` for eventual use in the constructor `$replacement`. If a subroutine is used, it also should return a hash of properties;
- `beforeDigest=>CODE($stomach)`, `afterDigest=>CODE($stomach,$whatsit)` provides code to be digested before and after digesting the arguments of the constructor, typically to alter the context of the digestion (before), or to augment the properties of the `Whatsit` (after);
- `beforeConstruct=>CODE($document,$whatsit)`, `afterConstruct=>CODE($document,$whatsit)` provides code to be run before and after the main `$replacement` is effected; occassionaly it is convenient to use the pattern form for the main `$replacement`, but one still wants to execute a bit of Perl code, as well;
- `captureBody=>(1 | $token)` specifies that an additional argument (like an environment body) wiil be read until the current TEX grouping ends, or until the specified `$token` is encountered. This argument is available to `$replacement` as `$body`;
- `scope=>('global'|'local'|$name)` specifies whether this definition is made globally, or in the current stack frame (default), (or in a named scope);
- `reversion=>$string|CODE(...)`, `alias=>$cs` can be used when the `Whatsit` needs to be reverted into TEX code, and the default of simply reassembling based on the prototype is not desired. See the code for examples.

Some additional functions useful when writing constructors:

- **`ToString`**`($stuff)` converts `$stuff` to a string, hopefully without TEX markup, suitable for use as document content and attribute values. Note that if `$stuff` contains Whatsits generated by Constructors, it may not be possible to avoid TEX code. Constrast **`ToString`** to the following two functions.
- **`UnTeX`**`($stuff)` returns a string containing the TEX code that would generate `$stuff` (this might not be the original TEX). The function `Revert($stuff)` returns the same information as a Tokens list.
- **`Stringify`**`($stuff)` returns a string more intended for debugging purposes; it reveals more of the structure and type information of the object and its parts.

- **CleanLabel**(`$arg`), **CleanIndexKey**(`$arg`), **CleanBibKey**(`$arg`), **CleanURL**(`$arg`) cleans up arguments (converting to string, handling invalid characters, etc) to make the argument appropriate for use as an attribute representing a label, index ID, etc.

- **UTF**(`$hex`) returns the Unicode character for the given codepoint; this is useful for characters below `0x100` where Perl becomes confused about the encoding.

DefEnvironment($prototype,$replacement,%options) Environments are largely a special case of constructors, but the prototype starts with `{envname}`, rather than `\cmd`, the replacement will also typically involve `#body` representing the contents of the environment.

`DefEnvironment` takes the same options as `DefConstructor`, with the addition of

- `afterDigestBegin=>CODE($stomach,$whatsit)` provides code to digest after the `\begin{env}` is digested;

- `beforeDigestEnd=>CODE($stomach)` provides code to digest before the `\end{env}` is digested.

For those cases where you do not want an environment to correspond to a constructor, you may still (as in LaTeX), define the two control sequences `\envname` and `\endenvname` as you like.

4.1.4 Document Model

The following declarations are typically only needed when customizing the schema used by LaTeXML.

- **RelaxNGSchema**(`$schema,`*`namespaces)`* declares the created XML document should be fit to the RelaxNG schema in `$schema`; A file `$schema.rng` should be findable in the current search paths. (Note that currently, LaTeXML is unable to directly parse compact notation).

- **RegisterNamespace**(`$prefix,$url`) associates the prefix with the given namespace url. This allows you to use `$prefix` as a namespace prefix when writing `Constructor` patterns or XPath expressions.

- **Tag**(`$tag,`*`properties)`* specifies properties for the given XML `$tag`. Recognized properties include: `autoOpen=>1` indicates that the tag can automatically be opened if needed to create a valid document; `autoClose=>1` indicates that the tag can automatically be closed if needed to create a valid document; `afterOpen=>$code` specifies code to be executed before opening the tag; the code is passed the `Document` being constructed as well as the `Box` (or `Whatsit`) responsible for its creation; `afterClose=>code` similar to `afterOpen`, but executed after closing the element.

4.1.5 Rewriting

The following functions are a bit tricky to use (and describe), but can be quite useful in some circumstances.

DefLigature(*$regexp*, *%options*) applies a regular expression to substitute textnodes after they are closed; the only option is `fontTest=>$code` which restricts the ligature to text nodes where the current font passes `&$code($font)`.

DefMathLigature(*$code*, *%options*) allows replacement of sequences of math nodes. It applies `$code` to the current `Document` and each sequence of math nodes encountered in the document; if a replacement should occur, `$code` should return a list of the form `($n,$string,`*`attributes`*`)` in which case, the text content of the first node is replaced by `$string`, the given attributes are added, and the following `$n-1` nodes are removed.

DefRewrite(%spec) defines document rewrite rules. These specifications describe what document nodes match:

- `label=>$label` restricts to nodes contained within an element whose labels includes `$label`;
- `scope=>$scope` generalizes `label`; the most useful form a string like `'section:1.3.2'` where it matches the section element whose refnum is `1.3.2`;
- `xpath=>$xpath` selects nodes matching the given XPath;
- `match=>$tex` selects nodes that look like what processing the TEX string `$tex` would produce;
- `regexp=>$regexp` selects text nodes that match the given regular expression.

The following specifications describe what to do with the matched nodes:

- `attributes=>{`*`attr`*`}` adds the given attributes to the matching nodes;
- `replace=>$tex` replaces the matching nodes with the result of processing the TEX string `$tex`.

4.1.6 Packages and Options

The following declarations are useful for defining LATEXML bindings, including option handling. As when defining LATEX packages, the following, if needed at all, need to appear in the order shown.

- **DeclareOption**`($option,$handler)` specifies the handler for `$option` when it is passed to the current package or class. If `$option` is `undef`, it defines the default handler, for options that are otherwise unrecognized. `$handler` can be either a string to be expanded, or a sub which is executed like a primitive.

- **PassOptions**(`$name,$type,@options`) specifies that the given options should be passed to the package (if `$type` is `sty`) or class (if `$type` is `cls`) `$name`, if it is ever loaded.
- **ProcessOptions**(*keys*) processes any options that have been passed to the current package or class. If `inorder=>1` is specified, the options will be processed in the order passed to the package (`\ProcessOptions*`); otherwise they will be processed in the declared order (`\ProcessOptions`).
- **ExecuteOptions**(`@options`) executes the handlers for the specific set of options `@options`.
- **RequirePackage**(`$pkgname,`*keys*) loads the specified package. The keyword options have the following effect: `options=>$options` can provide an explicit array of string specifying the options to pass to the package; `withoptions=>1` means that the options passed to the currently loading class or package should be passed to the requested package; `type=>$ext` specifies the type of the package file (default is `sty`); `raw=>1` specifies that reading the raw style file (eg. `pkg.sty`) is permissible if there is no specific LaTeXML binding (eg. `pkg.sty.ltxml`) `after=>$after` specifies a string or `(LaTeXML::Core::)Tokens` to be expanded after the package has finished loading.
- **LoadClass**(`$classname,`*keys*) Similar to `RequirePackage`, but loads a class file (`type=>'cls'`).
- **AddToMacro**(`$cstoken,$tokens`) a little used utilty to add material to the expansion of `$cstoken`, like an `\edef`; typically used to add code to a class or package hook.

4.1.7 Miscellaneous

Other useful stuff:

- **RawTeX**(`$texstring`) expands and processes the `$texstring`; This is typically useful to include definitions copied from a TeX stylefile, when they are approriate for LaTeXML, as is. Single-quoting the `$texstring` is useful, since it isn't interpolated by Perl, and avoids having to double all the slashes!

4.2 latexmlpost Customization

The current postprocessing framework works by passing the document through a sequence of postprocessing filter modules. Each module is responsible for carrying out a specific transformation, augmentation or conversion on the document. In principle, this architecture has the flexibility to employ new filters to perform new or customized conversions. However, the driver, `latexmlpost`, currently provides no convenient means to instanciate and incorporate outside filters, short of developing your own specialized version.

Consequently, we will consider custom postprocessing filters outside the scope of this manual (but of course, you are welcome to explore the code, or contact us with suggestions).

The two areas where customization is most practical is in altering the XSLT transforms used and extending the CSS stylesheets.

4.2.1 XSLT

LaTeXML provides stylesheets for transforming its XML format to XHTML and HTML. These stylesheets are modular with components corresponding to the schema modules. Probably the best strategy for customizing the transform involves making a copy of the standard base stylesheets, `LaTeXML-xhtml.xsl`, `LaTeXML-html.xsl` and `LaTeXML-html5.xsl`, found at *installationdir*`/LaTeXML/resources/XSLT/` — they're short, consisting mainly of an `xsl:include` and setting appropriate parameters and output method; thus modifying the parameters and and adding your own rules, or including your own modules should be relatively easy.

Naturally, this requires a familiarity with LaTeXML's schema (see D), as well as XSLT and XHTML. See the other stylesheet modules in the same directory as the base stylesheet for guidance. Generally the strategy is to use various parameters to switch between common behaviors and to use templates with `modes` that can be overridden in the less common cases.

Conversion to formats other than XHTML are, of course, possible, as well, but are neither supplied nor covered here. How complex the transformation will be depends on the extent that the LaTeXML schema can be mapped to the desired one, and to what extent LaTeXML has lost or hidden information represented in the original document. Again, familiarity with the schema is needed, and the provided XHTML stylesheets may suggest an approach.

NOTE: I'm trying to make stylesheets *easily* customizable. However, this is getting tricky.

- You can import stylesheets which allows the templates to be overridden.
- You can call the overridden stylesheet using `apply-imports`
- You can *not* call `apply-imports` to call an overridden *named* template! (although you seemingly can override them?)
- You can refer to xslt modules using URN's, provided you have loaded the `LaTeXML.catalog`:

```
<xsl:import href="urn:x-LaTeXML:XSLT:LaTeXML-all-xhtml.xsl"/>
```

4.2.2 CSS

CSS stylesheets can be supplied to `latexmlpost` to be included in the generated documents in addition to, or as a replacement for, the standard stylesheet `LaTeXML.css`. See the directory *installationdir*`/LaTeXML/resources/CSS/` for samples.

To best take advantage of this capability so as to design CSS rules with the correct specificity, the following points are helpful:

- LaTeXML converts the TeX to its own schema, with structural elements (like equation) getting their own tag; others are transformed to something more generic, such as note. In the latter case, a class attribute is often used to distinguish. For example, a `\footnote` generates

  ```
  <note class='footnote'>...
  ```

  whereas an `\endnote` generates

  ```
  <note class='endnote'>...
  ```

- The provided XSLT stylesheets transform LaTeXML's schema to XHTML, generating a combined class attribute consisting of any class attributes already present as well as the LaTeXML tag name. However, there are some variations on the theme. For example, LaTeX's `\section` yeilds a LaTeXML element section, with a title element underneath. When transformed to XHTML, the former becomes a , while the latter becomes <h2 class='section-title'> (for example, the h-level may vary with the document structure),

Mode `begin` and `end` For most elements, once the main html element has been opened and the primary attributes have been added but before any content has been added, a template with mode `begin` is called; thus it can add either attributes or content. Just before closing the main html element, a template with mode `end` is called.

Computing class and style Templates with mode `classes` and `styling`.

Chapter 5

Mathematics

There are several issues that have to be dealt with in treating the mathematics. On the one hand, the TEX markup gives a pretty good indication of what the author wants the math to look like, and so we would seem to have a good handle on the conversion to presentation forms. On the other hand, content formats are desirable as well; there are a few, but too few, clues about what the intent of the mathematics is. And in fact, the generation of even Presentation MathML of high quality requires recognizing the mathematical structure, if not the actual semantics. The mathematics processing must therefore preserve the presentational information provided by the author, while inferring, likely with some help, the mathematical content.

From a parsing point of view, the TEX-like processing serves as the lexer, tokenizing the input which LATEXML will then parse [perhaps eventually a type-analysis phase will be added]. Of course, there are a few twists. For one, the tokens, represented by XMTok, can carry extra attributes such as font and style, but also the name, meaning and grammatical role, with defaults that can be overridden by the author — more on those, in a moment. Another twist is that, although LATEX's math markup is not nearly as semantic as we might like, there is considerable semantics and structure in the markup that we can exploit. For example, given a `\frac`, we've already established the numerator and denominator which can be parsed individually, but the fraction as a whole can be directly represented as an application, using XMApp, of a fraction operator; the resulting structure can be treated as atomic within its containing expression.This *structure preserving* character greatly simplifies the parsing task and helps reduce misinterpretation.

The parser, invoked by the postprocessor, works only with the top-level lists of lexical tokens, or with those sublists contained in an XMArg. The grammar works primarily through the name and grammatical role. The name is given by an attribute, or the content if it is the same. The role (things like ID, FUNCTION, OPERATOR, OPEN, ...) is also given by an attribute, or, if not present, the name is looked up in a document-specific dictionary (*jobname*`.dict`), or in a default dictionary.

Additional exceptions that need fuller explanation are:

- `Constructor`s may wish to create a dual object (XMDual) whose children are

the semantic and presentational forms.

- Spacing and similar markup generates XMHint elements, which are currently ignored during parsing, but probably shouldn't.

5.1 Math Details

LaTeXML processes mathematical material by proceeding through several stages:

- Basic processing of macros, primitives and constructors resulting in an XML document; the math is primarily represented by a sequence of tokens (XMTok) or structured items (XMApp, XMDual) and hints (XMHint, which are ignored).
- Document tree rewriting, where rules are applied to modify the document tree. User supplied rules can be used here to clarify the intent of markup used in the document.
- Math Parsing; a grammar based parser is applied, depth first, to each level of the math. In particular, at the top level of each math expression, as well as each subexpression within structured items (these will have been contained in an XMArg or XMWrap element). This results in an expression tree that will hopefully be an accurate representation of the expression's structure, but may be ambigous in specifics (eg. what the meaning of a superscript is). The parsing is driven almost entirely by the grammatical role assigned to each item.
- *Not yet implemented* a following stage must be developed to resolve the semantic ambiguities by analyzing and augmenting the expression tree.
- Target conversion: from the internal XM* representation to MathML or OpenMath.

The Math element is a top-level container for any math mode material, serving as the container for various representations of the math including images (through attributes mathimage, width and height), textual (through attributes tex, content-tex and text), MathML and the internal representation itself. The mode attribute specifies whether the math should be in display or inline mode.

5.1.1 Internal Math Representation

The XMath element is the container for the internal representation

The following attributes can appear on all XM* elements:

role the grammatical role that this element plays

open, close parenthese or delimiters that were used to wrap the expression represented by this element.

argopen, argclose, separators delimiters on an function or operator (the first element of an XMApp) that were used to delimit the arguments of the function. The separators is a string of the punctuation characters used to separate arguments.

xml:id a unique identifier to allow reference (XMRef) to this element.

Math Tags The following tags are used for the intermediate math representation:

XMTok represents a math token. It may contain text for presentation. Additional attributes are:

name the name that represents the *meaning* of the token; this overrides the content for identifying the token.

omcd the OpenMath content dictionary that the name belongs to.

font the font to be used for presenting the content.

style ?

size ?

stackscripts whether scripts should be stacked above/below the item, instead of the usual script position.

XMApp represents the generalized application of some function or operator to arguments. The first child element is the operator, the remainig elements are the arguments. Additional attributes:

name the name that represents the meaning of the construct as a whole.

stackscripts ?

XMDual combines representations of the content (the first child) and presentation (the second child), useful when the two structures are not easily related.

XMHint represents spacing or other apparent purely presentation material.

name names the effect that the hint was intended to achieve.

style ?

XMWrap serves to assert the expected type or role of a subexpression that may otherwise be difficult to interpret — the parser is more forgiving about these.

name ?

style ?

XMArg serves to wrap individual arguments or subexpressions, created by structured markup, such as `\frac`. These subexpressions can be parsed individually.

rule the grammar rule that this subexpression should match.

XMRef refers to another subexpression,. This is used to avoid duplicating arguments when constructing an XMDual to represent a function application, for example. The arguments will be placed in the content branch (wrapped in an XMArg) while XMRef's will be placed in the presentation branch.

idref the identifier of the referenced math subexpression.

5.1.2 Grammatical Roles

As mentioned above, the grammar take advantage of the structure (however minimal) of the markup. Thus, the grammer is applied in layers, to sequences of tokens or *atomic* subexpressions (like a fractions or arrays). It is the role attribute that indicates the syntactic and/or presentational nature of each item. On the one hand, this drives the parsing: the grammar rules are keyed on the role (say, ADDOP), rather than content (say + or -), of the nodes [In some cases, the content is used to distinguish special synthesized roles]. The role is also used to drive the conversion to presentation markup, (say, as an infix operator), especially Presentation MATHML. Some values of role are used only in the grammar, some are only used in presentation; most are used both ways.

The following grammatical roles are recognized by the math parser. These values can be specified in the role attribute during the initial document construction or by rewrite rules. Although the precedence of operators is loosely described in the following, since the grammar contains various special case productions, no rigidly ordered precedence is given. Also note that in the current design, an expresssion has only a single role, although that role may be involved in grammatical rules with distinct syntax and semantics (some roles directly reflect this ambiguity).

ATOM a general atomic subexpression (atomic at the level of the expression; it may have internal structure);

ID a variable-like token, whether scalar or otherwise, but not a function;

NUMBER a number;

ARRAY a structure with internal components and alignments; typically has a particular syntactic relationship to OPEN and CLOSE tokens.

UNKNOWN an unknown expression. This is the default for token elements. Such tokens are treated essential as ID, but generate a warning if it seems to be used as a function.

OPEN,CLOSE opening and closing delimiters, group expressions or enclose arguments among other structures;

MIDDLE a middle operator used to group items between an OPEN, CLOSE pair;

PUNCT,PERIOD punctuation; a period 'ends' formula (note that numbers, including floating point, are recognized earlier in processing);

VERTBAR a vertical bar (single or doubled) which serves a confusing variety of notations: absolute values, "at", divides;

RELOP a relational operator, loosely binding;

ARROW an arrow operator (with little semantic significance), but generally treated equivalently to RELOP;

METARELOP an operator used for relations between relations, with lower precedence;

MODIFIER an atomic expression following an object that 'modifies' it in some way, such as a restriction (< 0) or modulus expression;

MODIFIEROP an operator (such as mod) between two expressions such that the latter modifies the former;

ADDOP an addition operator, between `RELOP` and `MULOP` operators in precedence;

MULOP a multiplicative operator, high precedence than `ADDOOP`;

BINOP a generic infix operator, can act as either an `ADDOP` or `MULOP`, typically used for cases wrapped in `\mathbin`;

SUPOP An operator appearing *in* a superscript, such as a collection of primes, or perhaps a T for transpose. This is distinct from an expression in a superscript with an implied power or index operator;

PREFIX for a prefix operator;

POSTFIX for a postfix operator;

FUNCTION a function which (may) apply to following arguments with higher precedence than addition and multiplication, or to parenthesized arguments (enclosed between `OPEN`,`CLOSE`);

OPFUNCTION a variant of `FUNCTION` which doesn't require fenced arguments;

TRIGFUNCTION a variant of `OPFUNCTION` with special rules for recognizing which following tokens are arguments and which are not;

APPLYOP an explicit infix application operator (high precedence);

COMPOSEOP an infix operator that composes two `FUNCTION`'s (resulting in another `FUNCTION`);

OPERATOR a general operator; higher precedence than function application. For example, for an operator A, and function F, AFx would be interpretted as $(A(F))(x)$;

SUMOP,INTOP, LIMITOP,DIFFOP,BIGOP a summation/union, integral, limiting, differential or general purpose operator. These are treated equivalently by the grammar, but are distinguished to facilitate (*eventually*) analyzing the argument structure (eg bound variables and differentials within an integral). **Note** are `SUMOP` and `LIMITOP` significantly different in this sense?

POSTSUBSCRIPT,POSTSUPERSCRIPT intermediate form of sub- and superscript, roughly as TEX processes them. The script is (essentially) treated as an argument but the base will be determined by parsing.

FLOATINGSUBSCRIPT,FLOATINGSUPERSCRIPT A special case for a sub- and superscript on an empty base, ie. `{}^{x}`. It is often used to place a pre-superscript or for non-math uses (eg. `10${}^{th}`);

The following roles are not used in the grammar, but are used to capture the presentation style; they are typically used directly in macros that construct structured objects, or used in representing the results of parsing an expression.

STACKED corresponds to stacked structures, such as `\atop`, and the presentation of binomial coefficients.

SUPERSCRIPTOP,SUBSCRIPTOP after parsing, the operator involved in various sub/superscript constructs above will be comverted to these;

OVERACCENT,UNDERACCENT these are special cases of the above that indicate the 2nd operand acts as an accent (typically smaller), expressions using these roles are usually directly constructed for accenting macros;

FENCED this operator is used to represent containers enclosed by `OPEN` and `CLOSE`, possibly with punctuation, particularly when no semantic is known for the construct, such as an arbitrary list.

The content of a token is actually used in a few special cases to distinguish distinct syntactic constructs, but these roles are *not* assigned to the role attribute of expressions:

LANGLE,RANGLE recognizes use of $<$ and $>$ in the bra-ket notation used in quantum mechanics;

LBRACE,RBRACE recognizes use of $\{$ and $\}$ on either side of stacked or array constructions representing various kinds of cases or choices;

SCRIPTOPEN recognizes the use of $\{$ in opening specialized set notations.

Chapter 6

Localization

In this chapter, a few issues relating to various national or cultural styles, languages or text encodings, which we'll refer to collectively as 'localization', are breifly discussed.

6.1 Numbering

Generally when titles and captions are formatted or when equations are numbered and when they are referred to in a cross reference or table of contents, text consisting of some combination of the raw title or caption text, a reference number and a type name (eg. 'Chapter') or symbol (eg. §) is composed and used. The exact compositions that is used at each level can depend on language, culture, the subject matter as well as both journal and individual style preferences. LaTeX has evolved to accommodate many of these styles and LaTeXML attempts to follow that lead, while preserve its options (the demands of extensively hyper-linked online material sometimes seems to demand more options and flexibility than traditional print formatting).

For example, the various macros `\chaptername`, `\partname`, `\refname`, etc. are respected and used. Likewise, the various counters and formatters such as `\theequation` are supported.

LaTeX's mechanism for formatting caption tags (`\fnum@figure` and `\fnum@table`) is extended to cover more cases. If you define `\fnum@`*type*, (where *type* is `chapter`, `section`, `subsection`, etc.) it will be used to format the reference number and/or type name for instances of that *type*. The macro `\fnum@toc@`*type* is used when formatting numbers for tables of contents.

Alternatively, you can define a macro `\format@title@`*type* that will be used format the whole title including reference number and type as desired; it takes a single argument, the title text. The macro `\format@toctitle@`*type* is used for the formatting a (typically) short form use in tables of contents.

6.2 Input Encodings

LaTeXML supports the standard LaTeX mechanism for handling non-ASCII encodings of the input TeX sources: using the `inputenc` package. The LaTeXML binding of `inputenc` loads the encoding definition (generally with extension `def`) directly from the LaTeX distribution (which are generally well-enough behaved to be easily processed). These encoding definitions make the upper 128 code points (of 8 bit) active and define TeX macros to handle them.

Using the commandline option `--inputencoding=utf8` to `latexml` allows processing of sources encoded as utf8, without any special packages loaded. [future work will make LaTeXML compatible with xetex]

6.3 Output Encodings

At some level, as far as TeX is concerned, what you type ends up pointing into a font that causes some blob of ink to be printed. This mechanism is used to print a unique mathematical operator, say 'subset of and not equals'. It is also used to print greek when you seemed to have been typing ASCII!

So, we must accomodate that mechanism, as well. At the stage when character tokens are digested to create boxes in the current font, a font encoding table (a FontMap) is consulted to map the token's text (viewed as an index into the table) to Unicode. The declaration `DeclareFontMap` is used to associate a FontMap with an encoding name, or font.

Note that this mapping is only used for text originating from the source document; The text within Constructor's XML pattern is used *without* any such font conversion.

6.4 Babel

The `babel` package for supporting multiple languages by redefining various internal bits of text to replace, eg. "Chapter" by "Kapital" and by defining various shorthand mechanisms to make it easy to type the extra non-latin characters and glyphs used by those languages. Each supported language or dialect has a module which is loaded to provide the needed definitions.

To the extent: that LaTeXML's input and output encoding handling is sufficient; that its processing of raw TeX is good enough; and that it proceeds through the appropriate LaTeX internals, LaTeXML should be able to support `babel` and arbitrary languages by reading in the raw TeX implementation of the language module from the TeX distribution itself.

At least, that is the strategy that we use.

Chapter 7

Alignments

There are several situations where TEX stacks or aligns a number of objects into a one or two dimensional grids. In most cases, these are built upon low-level primitives, like `\halign`, and so share characteristics: using & to separate alignment columns; either `\\` or `\cr` to separate rows. Yet, there are many different markup patterns and environments used for quite different purposes from tabular text to math arrays to composing symbols and so it is worth recognizing the intended semantics in each case, while still processing them as TEX would.

In this chapter, we will describe some of the special complications presented by alignments and the strategies used to infer and represent the appropriate semantic structures, particularly for math.

7.1 TEX Alignments

NOTE This section needs to be written.

Many utilities for setting up and processing alignments are defined in `TeX.pool` with support from the module `(LaTeXML::Core::)Alignment`. Typically, one binds a set of control sequences specially for the alignment environment or structure encountered, particularly for & and `\\`. An alignment object is created which records information about each row and cell that was processed, such as width, alignment, span, etc. Then the alignment is converted to XML by specifying what tag wraps the entire alignment, each row and each cell.

The content of aligments is being expanded before the column and row markers are recognized; this allows more flexibility in defining markup since row and column markers can be hidden in macros, but it also means that simple means, such as delimited parameter lists, to parse the structure won't work.

7.2 Tabular Header Heuristics

To be written

7.3 Math Forks

There are several constructs for aligning mathematics in LaTeX, and common packages. Here we are concerned with the large scale alignments where one or more equations are displayed in a grid, such as `eqnarray`, in standard LaTeX, and a suite of constructs of the amsmath packages. The arrangements are worth preserving as they often convey important information to the reader by the grouping, or by drawing attention to similarities or differences in the formula. At the same time, the individual fragments within the grid cells often have little 'meaning' on their own: it is subsequences of these fragments that represent the logical mathematical objects or formula. Thus, we would also like to recognize those sequences and synthesize complete formula for use in content-oriented services. We therefore have to devise an XML structure to represent this duality, as well as developing strategies for inferring and rearranging the mathematics as it was authored into the desired form.

The needed structure shares some characteristics with XMDual, *which needs to be described*, but needs to resided at the document level, containing several, possibly numbered, equations each of which provide two views. Additional objects, such as textual insertions (such as amsmath's `\intertext`), must also be accomodated.

The following XML is used to represent these structures:

```
<ltx:equationgroup>
  <ltx:equation>
    <ltx:MathFork>
      <ltx:Math>logical math here</ltx:Math>
      <ltx:MathBranch>
        <ltx:td><ltx:Math>cell math</ltx:Math></ltx:td>...
        or
        <ltx:tr><ltx:td><ltx:Math>...
      </ltx:MathBranch>
    </ltx:MathFork>
  </ltx:equation>
  <ltx:text>inter-text</ltx:text>
  ...more text or equations
</ltx:equationgroup>
```

Typically, the contents of the MathBranch will be a sequence of td, each containing an Math, or of tr, each containing sequence of such td. This structure can thus represent both `eqnarray` where a logical equation consists of one or more complete rows, as well as AMS' `aligned` where equations consist of pairs of columns. The XSLT transformation that converts to end formats recognizes which case and lays out appropriately.

In most cases, the material that will yield a MathFork is given as a set of partial math expressions representing rows and/or columnns; these must be concatenated (and parsed) to form the composite logical expression.

Any ID's within the expressions (and references to them) must be modified to avoid duplicate ids. Moreover, a useful application associates the displayed tokens from the aligned presentation of the MathBranch with the presumably semantic tokens in the logcal content of the main branch of the MathFork. Thus, we desire that the IDs in the

two branches to have a known relationship; in particular, those in the branch should have `.fork1` appended.

7.4 eqnarray

The `eqnarray` environment seems intended to represent one or more equations, but each equation can be continued with additional right-hand-sides (by omitting the 1st column), or the RHS itself can be continued on multiple lines by omitting the 1st two columns on a row. With our goal of constructing well-structured mathematics, this gives us a fun little puzzle to sort out. However, being essentially the only structure for aligning mathematical stuff in standard LaTeX, `eqnarray` tended to be stretched into various other use cases; aligning numbered equations with bits of text on the side, for example. We therefore have some work to do to guess what the intent is.

The strategy used for `eqnarray` is process the material as an alignment in math mode and convert initially to the following XML structure:

```
<ltx:equationgroup>
  <ltx:equation>
    <ltx:_Capture_>
      <ltx:Math><ltx:XMath>column math here</ltx:XMath></ltx:Math>
    </ltx:_Capture_>
    ...
  </ltx:equation>
  ...
</ltx:equationgroup>
```

The results are then studied to recognize the patterns of empty columns so that the rows can be regrouped into logical equations. MathFork structures are used to contain those logical equations while preserving the layout in the MathBranch.

NOTE We need to deal better with the cases that have more rows numbered that we would like.

7.5 AMS Alignments

The AMS math packages define a number of useful math alignment structures. These have been well thought out and designed with particular logical structures in mind, as well as the layout. Thus these environments are less often abused than is `eqnarray`. In this section, we list the environments, their expected use case and describe the strategy used for converting them.

To be done Describe alternates for `equation` and things inside equations; Describe single vs multiple logical equations. (and started variants)

This list outlines the *intended* use of the AMS alignment environments The following constructs are intended as top-level environments, used like `equation`.

Several of the constructs are used in place of a top-level `equation` and represent one or more logical equations. The following describes the intended usage, as a guide to understanding the implementation code (or its limitations!)

- `align`,`flalign`,`alignat`,`xalignat`: Each row may be numbered; has even number of columns; Each pair of columns, aligned right then left, represents a logical equation; Note that the documentation suggests that annotative text can be added by putting `\text{}` in a column followed by an empty column.
- `gather`: Each row is a single centered column representing an equation.
- `multline`: This environment represents a single equation broken to multiple lines; the lines are aligned left, center (repeated) and finally, right. *alignment not yet implemented*

The following environments are used *within* an equation (or similar) environment and thus do not generate MathFork structures. Moreover, except for `aligned`, their semantic intent is less clear. The preservation of the alignment have not yet been implemented; they; presumably would yeiled an XMDual.

- `split`
- `gathered`
- `aligned`,`alignedat`

Note that the case of a single equation containing a single `aligned` is transformed into and treated equivalently to a top-level `align`.

Chapter 8

Metadata

8.1 RDFa

LaTeXML has support for representing and generating RDFa metadata in LaTeXML documents. The core attributes property, rel, rev, about resource, typeof and content are included. Provision is also made for about and resource to be specified using LaTeX-style labels, or plain XML id's.

The default set of vocabularies is specified in HTML Role Vocabulary[1], and the associated set of prefixes are predefined.

It is intended that the support will be extended to automatically generate RDFa data from the implied semantics of LaTeX markup; the idea would be not to inadvertently override any explicitly provided metadata supplied by one of the following packages.

The hyperref package The hyperref and hyperxmp packages provide a means to specify metadata which will be embedded in the generated pdf file; LaTeXML converts that data to RDFa in its generated XML.

The lxRDFa package There is also a LaTeXML-specific package, lxRDFa, which provides several commands for annotating the generated XML. The most powerful of which is `\lxRDFa` which allows you to specify any set or subset of RDFa attributes on the current XML element and thus take advantage of the arbitrary shorthands, chaining and partial triples that RDFa allows. Correspondingly, you are must beware of clashes or unintended changes to the set of triples generated by explicit and hidden RDFa data.

[1] `https://www.w3.org/1999/xhtml/vocab/#XHTMLRoleVocabulary`

Chapter 9

ToDo

Lots...!

- Many useful LaTeX packages have not been implemented, and those that are aren't necessarily complete.

  Contributed bindings are, of course, welcome!

- Low-level TeX capabilities, such as text modes (eg. vertical, horizonatal), box details like width and depth, as well as fonts, aren't mimicked faithfully, although it isn't clear how much can be done at the 'semantic' level.

- a richer math grammar, or more flexible parsing engine, better inferencing of math structure, better inferencing of math *meaning*...and thus better Content MathML and OpenMath support!

- Could be faster.

- Easier customization of the document schema, XSLT stylesheets.

- ...um, ...*documentation*!

Acknowledgements

Thanks to the DLMF project and it's Editors — Frank Olver, Dan Lozier, Ron Boisvert, and Charles Clark — for providing the motivation and opportunity to pursue this.

Thanks to the arXMLiv project, in particular Michael Kohlhase and Heinrich Stamerjohanns, for providing a rich testbed and testing framework to exercise the system. Additionally, thanks to Ioan Sucan, Catalin David and Silviu Oprea for testing help and for implementing additional packages.

Particular thanks go to Deyan Ginev as an enthusiastic supporter and developer.

Appendix A

Command Documentation

`latexml`

Transforms a TeX/LaTeX file into XML.

Synopsis

latexml [options] *texfile*

```
Options:
--destination=file sets destination file (default stdout).
--output=file      [obsolete synonym for --destination]
--preload=module   requests loading of an optional module;
                   can be repeated
--preamble=file    sets a preamble file which will
                   effectively be prepended to the main file.
--postamble=file   sets a postamble file which will
                   effectively be appended to the main file.
--includestyles    allows latexml to load raw *.sty file;
                   by default it avoids this.
--path=dir         adds to the paths searched for files,
                   modules, etc;
--log=file         specifies log file (default is file named after job name)
--documentid=id    assign an id to the document root.
--quiet            suppress messages (can repeat)
--verbose          more informative output (can repeat)
--strict           makes latexml less forgiving of errors
--bibtex           processes as a BibTeX bibliography.
--xml              requests xml output (default).
--tex              requests TeX output after expansion.
--box              requests box output after expansion
                   and digestion.
--noparse          suppresses parsing math
--nocomments       omit comments from the output
--inputencoding=enc specify the input encoding.
```

```
--VERSION          show version number.
--debug=package    enables debugging output for the named
                   package
--help             shows this help message.
```

If *texfile* is '-', latexml reads the TeX source from standard input. If *texfile* has an explicit extension of `.bib`, it is processed as a BibTeX bibliography.

Options & Arguments

`--destination=`*file*

Specifies the destination file; by default the XML is written to stdout.

`--preload=`*module*

Requests the loading of an optional module or package. This may be useful if the TeX code does not specifically require the module (eg. through input or usepackage). For example, use `--preload=LaTeX.pool` to force LaTeX mode.

`--preamble=`*file*, `--postamble=`*file*

Specifies a file whose contents will effectively be prepended or appended to the main document file's content. This can be useful when processing TeX fragments, in which case the preamble would contain documentclass and begindocument control sequences. This option is not used when processing BibTeX files.

`--includestyles`

This optional allows processing of style files (files with extensions `sty`, `cls`, `clo`, `cnf`). By default, these files are ignored unless a latexml implementation of them is found (with an extension of `ltxml`).

These style files generally fall into two classes: Those that merely affect document style are ignorable in the XML. Others define new markup and document structure, often using deeper LaTeX macros to achieve their ends. Although the omission will lead to other errors (missing macro definitions), it is unlikely that processing the TeX code in the style file will lead to a correct document.

`--path=`*dir*

Add *dir* to the search paths used when searching for files, modules, style files, etc; somewhat like TEXINPUTS. This option can be repeated.

`--documentid=`*id*

Assigns an ID to the root element of the XML document. This ID is generally inherited as the prefix of ID's on all other elements within the document. This is useful when constructing a site of multiple documents so that all nodes have unique IDs.

`--quiet`

Reduces the verbosity of output during processing, used twice is pretty silent.

--verbose

Increases the verbosity of output during processing, used twice is pretty chatty. Can be useful for getting more details when errors occur.

--strict

Specifies a strict processing mode. By default, undefined control sequences and invalid document constructs (that violate the DTD) give warning messages, but attempt to continue processing. Using --strict makes them generate fatal errors.

--bibtex

Forces latexml to treat the file as a BibTeX bibliography. Note that the timing is slightly different than the usual case with BibTeX and LaTeX. In the latter case, BibTeX simply selects and formats a subset of the bibliographic entries; the actual TeX expansion is carried out when the result is included in a LaTeX document. In contrast, latexml processes and expands the entire bibliography; the selection of entries is done during postprocessing. This also means that any packages that define macros used in the bibliography must be specified using the `--preload` option.

--xml

Requests XML output; this is the default.

--tex

Requests TeX output for debugging purposes; processing is only carried out through expansion and digestion. This may not be quite valid TeX, since Unicode may be introduced.

--box

Requests Box output for debugging purposes; processing is carried out through expansion and digestions, and the result is printed.

--nocomments

Normally latexml preserves comments from the source file, and adds a comment every 25 lines as an aid in tracking the source. The option --nocomments discards such comments.

--inputencoding=*encoding*

Specify the input encoding, eg. `--inputencoding=iso-8859-1`. The encoding must be one known to Perl's Encode package. Note that this only enables the translation of the input bytes to UTF-8 used internally by LaTeXML, but does not affect catcodes. It is usually better to use LaTeX's inputenc package. Note that this does not affect the output encoding, which is always UTF-8.

--VERSION

Shows the version number of the LaTeXML package..

--debug=*package*

Enables debugging output for the named package. The package is given without the leading LaTeXML::.

--help

Shows this help message.

See also

latexmlpost, latexmlmath, LaTeXML

latexmlpost

Postprocesses an xml file generated by `latexml` to perform common tasks, such as convert math to images and processing graphics inclusions for the web.

Synopsis

latexmlpost [options] *xmlfile*

```
Options:
--verbose               shows progress during processing.
--VERSION               show version number.
--help                  shows help message.
--sourcedirectory=sourcedir sets directory of the original
                        source TeX file.
--validate, --novalidate Enables (the default) or disables
                        validation of the source xml.
--format=html|html5|html4|xhtml|xml requests the output format.
                        (html defaults to html5)
--destination=file      sets output file (and directory).
--omitdoctype           omits the Doctype declaration,
--noomitdoctype         disables the omission (the default)
--numbersections        enables (the default) the inclusion of
                        section numbers in titles, crossrefs.
--nonumbersections      disables the above
--stylesheet=xslfile    requests the XSL transform using the
                        given xslfile as stylesheet.
--css=cssfile           adds css stylesheet to (x)html(5)
                        (can be repeated)
--nodefaultresources    disables processing built-in resources
--javascript=jsfile     adds a link to a javascript file into
                        html4/html5/xhtml (can be repeated)
--xsltparameter=name:value passes parameters to the XSLT.
--split                 requests splitting each document
--nosplit               disables the above (default)
--splitat               sets level to split the document
--splitpath=xpath       sets xpath expression to use for
```

```
                        splitting (default splits at
                        sections, if splitting is enabled)
--splitnaming=(id|idrelative|label|labelrelative) specifies
                        how to name split files (idrelative).
--scan                  scans documents to extract ids,
                        labels, etc.
                        section titles, etc. (default)
--noscan                disables the above
--crossref              fills in crossreferences (default)
--nocrossref            disables the above
--urlstyle=(server|negotiated|file) format to use for urls
                        (default server).
--navigationtoc=(context|none) generates a table of contents
                        in navigation bar
--index                 requests creating an index (default)
--noindex               disables the above
--splitindex            Splits index into pages per initial.
--nosplitindex          disables the above (default)
--permutedindex         permutes index phrases in the index
--nopermutedindex       disables the above (default)
--bibliography=file     sets a bibliography file
--splitbibliography     splits the bibliography into pages per
                        initial.
--nosplitbibliography   disables the above (default)
--prescan               carries out only the split (if
                        enabled) and scan, storing
                        cross-referencing data in dbfile
                        (default is complete processing)
--dbfile=dbfile         sets file to store crossreferences
--sitedirectory=dir     sets the base directory of the site
--mathimages            converts math to images
                        (default for html4 format)
--nomathimages          disables the above
--mathsvg               converts math to svg images
--nomathsvg             disables the above
--mathimagemagnification=mag sets magnification factor
--presentationmathml    converts math to Presentation MathML
                        (default for xhtml & html5 formats)
--pmml                  alias for --presentationmathml
--nopresentationmathml  disables the above
--linelength=n          formats presentation mathml to a
                        linelength max of n characters
--contentmathml         converts math to Content MathML
--nocontentmathml       disables the above (default)
--cmml                  alias for --contentmathml
--openmath              converts math to OpenMath
--noopenmath            disables the above (default)
--om                    alias for --openmath
--keepXMath             preserves the intermediate XMath
                        representation (default is to remove)
```

```
--mathtex               adds TeX annotation to parallel markup
--nomathtex             disables the above (default)
--unicodemath           adds TeX annotation to parallel markup
--nounicodemath         disables the above (default)
--plane1                use plane-1 unicode for symbols
                        (default, if needed)
--noplane1              do not use plane-1 unicode
--graphicimages         converts graphics to images (default)
--nographicimages       disables the above
--graphicsmap=type.type specifies a graphics file mapping
--pictureimages         converts picture environments to
                        images (default)
--nopictureimages       disables the above
--svg                   converts picture environments to SVG
--nosvg                 disables the above (default)
```

If *xmlfile* is '-', latexmlpost reads the XML from standard input.

Options & Arguments

General Options

`--verbose`

Requests informative output as processing proceeds. Can be repeated to increase the amount of information.

`--VERSION`

Shows the version number of the LaTeXML package..

`--help`

Shows this help message.

Source Options

`--sourcedirectory=`*source*

Specifies the directory where the original latex source is located. Unless latexmlpost is run from that directory, or it can be determined from the xml filename, it may be necessary to specify this option in order to find graphics and style files.

`--validate, --novalidate`

Enables (or disables) the validation of the source XML document (the default).

Format Options

`--format=(html|html5|html4|xhtml|xml)`

Specifies the output format for post processing. By default, it will be guessed from the file extension of the destination (if given), with html implying `html5`,

xhtml implying `xhtml` and the default being `xml`, which you probably don't want.

The `html5` format converts the material to html5 form with mathematics as MathML; `html5` supports SVG. `html4` format converts the material to the earlier html form, version 4, and the mathematics to png images. `xhtml` format converts to xhtml and uses presentation MathML (after attempting to parse the mathematics) for representing the math. `html5` similarly converts math to presentation MathML. In these cases, any graphics will be converted to web-friendly formats and/or copied to the destination directory. If you simply specify `html`, it will treat that as `html5`.

For the default, `xml`, the output is left in LaTeXML's internal xml, although the math can be converted by enabling one of the math postprocessors, such as --pmml to obtain presentation MathML. For html, html5 and xhtml, a default stylesheet is provided, but see the `--stylesheet` option.

`--destination=`*destination*

Specifies the destination file and directory. The directory is needed for mathimages, mathsvg and graphics processing.

`--omitdoctype, --noomitdoctype`

Omits (or includes) the document type declaration. The default is to include it if the document model was based on a DTD.

`--numbersections, --nonumbersections`

Includes (default), or disables the inclusion of section, equation, etc, numbers in the formatted document and crossreference links.

`--stylesheet=`*xslfile*

Requests the XSL transformation of the document using the given xslfile as stylesheet. If the stylesheet is omitted, a 'standard' one appropriate for the format (html4, html5 or xhtml) will be used.

`--css=`*cssfile*

Adds *cssfile* as a css stylesheet to be used in the transformed html/html5/xhtml. Multiple stylesheets can be used; they are included in the html in the order given, following the default `ltx-LaTeXML.css`. The stylesheet is copied to the destination directory, unless it is an absolute url.

Some stylesheets included in the distribution are --css=navbar-left Puts a navigation bar on the left. (default omits navbar) --css=navbar-right Puts a navigation bar on the left. --css=theme-blue A blue coloring theme for headings. --css=amsart A style suitable for journal articles.

`--javascript=`*jsfile*

Includes a link to the javascript file *jsfile*, to be used in the transformed html/html5/xhtml. Multiple javascript files can be included; they are linked in the

html in the order given. The javascript file is copied to the destination directory, unless it is an absolute url.

--icon=*iconfile*

Copies *iconfile* to the destination directory and sets up the linkage in the transformed html/html5/xhtml to use that as the "favicon".

--nodefaultresources

Disables the copying and inclusion of resources added by the binding files; This includes CSS, javascript or other files. This does not affect resources explicitly requested by the `--css` or `--javascript` options.

--timestamp=*timestamp*

Provides a timestamp (typically a time and date) to be embedded in the comments by the stock XSLT stylesheets. If you don't supply a timestamp, the current time and date will be used. (You can use `--timestamp=0` to omit the timestamp).

--xsltparameter=*name*:*value*

Passes parameters to the XSLT stylesheet. See the manual or the stylesheet itself for available parameters.

Site & Crossreferencing Options

--split, --nosplit

Enables or disables (default) the splitting of documents into multiple 'pages'. If enabled, the the document will be split into sections, bibliography, index and appendices (if any) by default, unless `--splitpath` is specified.

--splitat=*unit*

Specifies what level of the document to split at. Should be one of `chapter`, `section` (the default), `subsection` or `subsubsection`. For more control, see `--splitpath`.

--splitpath=*xpath*

Specifies an XPath expression to select nodes that will generate separate pages. The default splitpath is //ltx:section | //ltx:bibliography | //ltx:appendix | //ltx:index

Specifying

```
--splitpath="//ltx:section | //ltx:subsection
       | //ltx:bibliography | //ltx:appendix | //ltx:index"
```

would split the document at subsections as well as sections.

--splitnaming=(id|idrelative|label|labelrelative)

Specifies how to name the files for subdocuments created by splitting. The values `id` and `label` simply use the id or label of the subdocument's root node for it's filename. `idrelative` and `labelrelative` use the portion of the id or label that follows the parent document's id or label. Furthermore, to impose structure and uniqueness, if a split document has children that are also split, that document (and it's children) will be in a separate subdirectory with the name index.

--scan, --noscan

Enables (default) or disables the scanning of documents for ids, labels, references, indexmarks, etc, for use in filling in refs, cites, index and so on. It may be useful to disable when generating documents not based on the LaTeXML doctype.

--crossref, --nocrossref

Enables (default) or disables the filling in of references, hrefs, etc based on a previous scan (either from `--scan`, or `--dbfile`) It may be useful to disable when generating documents not based on the LaTeXML doctype.

--urlstyle=(server|negotiated|file)

This option determines the way that URLs within the documents are formatted, depending on the way they are intended to be served. The default, `server`, eliminates unnecessary trailing `index.html`. With `negotiated`, the trailing file extension (typically `html` or `xhtml`) are eliminated. The scheme `file` preserves complete (but relative) urls so that the site can be browsed as files without any server.

--navigationtoc=(context|none)

Generates a table of contents in the navigation bar; default is `none`. The 'context' style of TOC, is somewhat verbose and reveals more detail near the current page; it is most suitable for navigation bars placed on the left or right. Other styles of TOC should be developed and added here, such as a short form.

--index, --noindex

Enables (default) or disables the generation of an index from indexmarks embedded within the document. Enabling this has no effect unless there is an index element in the document (generated by \printindex).

--splitindex, --nosplitindex

Enables or disables (default) the splitting of generated indexes into separate pages per initial letter.

--bibliography=*pathname*

Specifies a bibliography generated from a BibTeX file to be used to fill in a bibliography element. Hand-written bibliographies placed in a `thebibliography`

environment do not need this. The option has no effect unless there is an bibliography element in the document (generated by \bibliography).

Note that this option provides the bibliography to be used to fill in the bibliography element (generated by `\bibliography`); latexmlpost does not (currently) directly process and format such a bibliography.

`--splitbibliography`, `--nosplitbibliography`

Enables or disables (default) the splitting of generated bibliographies into separate pages per initial letter.

`--prescan`

By default `latexmlpost` processes a single document into one (or more; see `--split`) destination files in a single pass. When generating a complicated site consisting of several documents it may be advantageous to first scan through the documents to extract and store (in `dbfile`) cross-referencing data (such as ids, titles, urls, and so on). A later pass then has complete information allowing all documents to reference each other, and also constructs an index and bibliography that reflects the entire document set. The same effect (though less efficient) can be achieved by running `latexmlpost` twice, provided a `dbfile` is specified.

`--dbfile=`*file*

Specifies a filename to use for the crossreferencing data when using two-pass processing. This file may reside in the intermediate destination directory.

`--sitedirectory=`*dir*

Specifies the base directory of the overall web site. Pathnames in the database are stored in a form relative to this directory to make it more portable.

Math Options These options specify how math should be converted into other formats. Multiple formats can be requested; how they will be combined depends on the format and other options.

`--mathimages`, `--nomathimages`

Requests or disables the conversion of math to images (png by default). Conversion is the default for html4 format.

`--mathsvg`, `--nomathsvg`

Requests or disables the conversion of math to svg images.

`--mathimagemagnification=`*factor*

Specifies the magnification used for math images (both png and svg), if they are made. Default is 1.75.

`--presentationmathml`, `--nopresentationmathml`

Requests or disables conversion of math to Presentation MathML. Conversion is the default for xhtml and html5 formats.

--linelength=*number*

(Experimental) Line-breaks the generated Presentation MathML so that it is no longer than *number* 'characters'.

--plane1

Converts the content of Presentation MathML token elements to the appropriate Unicode Plane-1 codepoints according to the selected font, when applicable (the default).

--hackplane1

Converts the content of Presentation MathML token elements to the appropriate Unicode Plane-1 codepoints according to the selected font, but only for the mathvariants double-struck, fraktur and script. This gives support for current (as of August 2009) versions of Firefox and MathPlayer, provided a sufficient set of fonts is available (eg. STIX).

--contentmathml, --nocontentmathml

Requests or disables conversion of math to Content MathML. Conversion is disabled by default. **Note** that this conversion is only partially implemented.

--openmath

Requests or disables conversion of math to OpenMath. Conversion is disabled by default. **Note** that this conversion is only partially implemented.

--keepXMath

By default, when any of the MathML or OpenMath conversions are used, the intermediate math representation will be removed; this option preserves it; it will be used as secondary parallel markup, when it follows the options for other math representations.

Graphics Options

--graphicimages, --nographicimages

Enables (default) or disables the conversion of graphics to web-appropriate format (png).

--graphicsmap=*sourcetype.desttype*

Specifies a mapping of graphics file types. Typically, graphics elements specify a graphics file that will be converted to a more appropriate file target format; for example, postscript files used for graphics with LaTeX will be converted to png format for use on the web. As with LaTeX, when a graphics file is specified without a file type, the system will search for the most appropriate target type file.

When this option is used, it overrides *and replaces* the defaults and provides a mapping of *sourcetype* to *desttype*. The option can be repeated to provide

several mappings, with the earlier formats preferred. If the *desttype* is omitted, it specifies copying files of type *sourcetype*, unchanged.

The default settings is equivalent to having supplied the options:

```
--graphicsmap=svg
--graphicsmap=png
--graphicsmap=gif
--graphicsmap=jpg
--graphicsmap=jpeg
--graphicsmap=eps.png
--graphicsmap=ps.png
--graphicsmap=ai.png
--graphicsmap=pdf.png
```

The first formats are preferred and used unchanged, while the latter ones are converted to png.

--pictureimages, --nopictureimages

Enables (default) or disables the conversion of picture environments and pstricks material into images.

--svg, --nosvg

Enables or disables (default) the conversion of picture environments and pstricks material to SVG.

See also

latexml, latexmlmath, LaTeXML

latexmlc

An omni-executable for LaTeXML, capable of stand-alone, socket-server and web service conversion. Supports both core processing and post-processing.

SYNOPSYS

See the OPTIONS section in LaTeXML::Common::Config for usage information.

Description

latexmlc provides a client which automatically sets up a LaTeXML local server if necessary (via latexmls).

```
If such server already exists, the client proceeds to communicate normally.

A stand-alone conversion (the default) can also be requested via --expire=-1
```

See also

LaTeXML::Common::Config

latexmlmath

Transforms a TeX/LaTeX math expression into various formats.

Synopsis

latexmlmath [options] *texmath*

```
Options:
--mathimage=file           converts to image in file
--mathsvg=file             converts to svg image in file
--magnification=mag        specifies magnification factor
--presentationmathml=file  converts to Presentation MathML
--pmml=file                alias for --presentationmathml
--linelength=n             do linewrapping of pMML
--contentmathml=file       convert to Content MathML
--cmml=file                alias for --contentmathml
--openmath=file            convert to OpenMath
--om=file                  alias for --openmath
--unicodemath=file         convert to UnicodeMath
--XMath=file               output LaTeXML's internal format
--noparse                  disables parsing of math
                           (not useful for cMML or openmath)
--preload=file             loads a style file.
--includestyles            allows processing raw *.sty files
                           (normally it avoids this)
--path=dir                 adds a search path for style files.
--quiet                    reduces verbosity (can repeat)
--verbose                  increases verbosity (can repeat)
--strict                   be more strict about errors.
--documentid=id            assign an id to the document root.
--debug=package            enables debugging output for the
                           named package
--inputencoding=enc specify the input encoding.
--VERSION                  show version number and exit.
--help                     shows this help message.
--                         ends options
```

If *texmath* is '-', `latexmlmath` reads the TeX from standard input. If any of the output files are '-', the result is printed on standard output.

Input notes Note that, unless you are reading *texmath* from standard input, the *texmath* string will be processed by whatever shell you are using before `latexmlmath` even sees it. This means that many so-called meta characters, such as backslash and star, may confuse the shell or be changed. Consequently, you will need to quote and/or

slashify the input appropriately. Most particularly, \ will need to be doubled to \\ for `latexmlmath` to see it as a control sequence.

Using `--` to explicitly end the option list is useful for cases when the math starts with a minus (and would otherwise be interpreted as an option, probably an unrecognized one). Alternatively, wrapping the *texmath* with {} will hide the minus.

Simple examples:

```
latexmlmath \\frac{-b\\pm\\sqrt{b^2-4ac}}{2a}
echo "\\sqrt{b^2-4ac}" | latexmlmath --pmml=quad.mml -
```

Options & Arguments

Conversion Options These options specify what formats the math should be converted to. In each case, the destination file is given. Except for mathimage, the file can be given as '-', in which case the result is printed to standard output.

If no conversion option is specified, the default is to output presentation MathML to standard output.

`--mathimage=`*file*

Requests conversion to png images.

`--mathsvg=`*file*

Requests conversion to svg images.

`--magnification=`*factor*

Specifies the magnification used for math image. Default is 1.75.

`--presentationmathml=`*file*

Requests conversion to Presentation MathML.

`--linelength=`*number*

(Experimental) Line-breaks the generated Presentation MathML so that it is no longer than *number* 'characters'.

`--plane1`

Converts the content of Presentation MathML token elements to the appropriate Unicode Plane-1 codepoints according to the selected font, when applicable.

`--hackplane1`

Converts the content of Presentation MathML token elements to the appropriate Unicode Plane-1 codepoints according to the selected font, but only for the mathvariants double-struck, fraktur and script. This gives support for current (as of August 2009) versions of Firefox and MathPlayer, provided a sufficient set of fonts is available (eg. STIX).

--contentmathml=*file*

Requests conversion to Content MathML. **Note** that this conversion is only partially implemented.

--openmath=*file*

Requests conversion to OpenMath. **Note** that this conversion is only partially implemented.

--unicodemath=*file*

Requests conversion to UnicodeMath (an unstructured Unicode string).

--XMath=*file*

Requests conversion to LaTeXML's internal format.

Other Options

--preload=*module*

Requests the loading of an optional module or package. This may be useful if the TeX code does not specifically require the module (eg. through input or usepackage). For example, use `--preload=LaTeX.pool` to force LaTeX mode.

--includestyles

This optional allows processing of style files (files with extensions `sty`, `cls`, `clo`, `cnf`). By default, these files are ignored unless a latexml implementation of them is found (with an extension of `ltxml`).

These style files generally fall into two classes: Those that merely affect document style are ignorable in the XML. Others define new markup and document structure, often using deeper LaTeX macros to achieve their ends. Although the omission will lead to other errors (missing macro definitions), it is unlikely that processing the TeX code in the style file will lead to a correct document.

--path=*dir*

Add *dir* to the search paths used when searching for files, modules, style files, etc; somewhat like TEXINPUTS. This option can be repeated.

--documentid=*id*

Assigns an ID to the root element of the XML document. This ID is generally inherited as the prefix of ID's on all other elements within the document. This is useful when constructing a site of multiple documents so that all nodes have unique IDs.

--quiet

Reduces the verbosity of output during processing, used twice is pretty silent.

--verbose

Increases the verbosity of output during processing, used twice is pretty chatty. Can be useful for getting more details when errors occur.

--strict

Specifies a strict processing mode. By default, undefined control sequences and invalid document constructs (that violate the DTD) give warning messages, but attempt to continue processing. Using --strict makes them generate fatal errors.

--inputencoding=*encoding*

Specify the input encoding, eg. `--inputencoding=iso-8859-1`. The encoding must be one known to Perl's Encode package. Note that this only enables the translation of the input bytes to UTF-8 used internally by LaTeXML, but does not affect catcodes. It is usually better to use LaTeX's inputenc package. Note that this does not affect the output encoding, which is always UTF-8.

--VERSION

Shows the version number of the LaTeXML package..

--debug=*package*

Enables debugging output for the named package. The package is given without the leading LaTeXML::.

--help

Shows this help message.

BUGS

This program runs much slower than would seem justified. This is a result of the relatively slow initialization including loading TeX and LaTeX macros and the schema. Normally, this cost would be ammortized over large documents, whereas, in this case, we're processing a single math expression.

See also

`latexml`, `latexmlpost`, `LaTeXML`

Appendix B

Implemented Bindings

Bindings for the following classes and packages are supplied with the distribution:

classes: IEEEtran, JHEP, JHEP2, JHEP3, OmniBus, PoS, a0poster, aa, aastex, acmart, aipproc, ams_core, amsart, amsbook, amsproc, article, beamer, book, elsart, elsarticle, emulateapj, gen-j-l, gen-m-l, gen-p-l, ieeeconf, iopart, llncs, minimal, mn, mn2e, mnras, moderncv, quantumarticle, report, revtex, revtex4, revtex4-1, slides, standalone, subfiles, svjour, svjour3, svmult

packages: SIunits, a0size, a4, a4wide, aa_support, aas_macros, aas_support, aasms, aaspp, aastex, accents, acronym, actuarialangle, adjustbox, ae, aecompl, afterpage, aipproc, algc, algcompatible, algmatlab, algorithm, algorithm2e, algorithmic, algorithmicx, algpascal, algpseudocode, alltt, ams_support, amsaddr, amsbsy, amscd, amsfonts, amsgen, amsmath, amsopn, amsppt, amsrefs, amssymb, amstex, amstext, amsthm, amsxtra, apjfonts, appendix, array, attachfile, atveryend, authblk, auxhook, avant, babel, babel_support, balance, bbding, bbm, bbold, beton, bezier, bibunits, bigintcalc, bitset, blindtext, bm, bookman, bookmark, booktabs, boxedminipage, braket, breakurl, calc, calrsfs, cancel, caption, cases, ccfonts, chancery, chapterbib, charter, chngcntr, circuitikz, cite, citesort, cleveref, cmap, cmbright, color, colordvi, colortbl, comment, concmath, courier, crop, cropmark, csquotes, dcolumn, deluxetable, diagbox, doi, doublespace, dsfont, ed, ellipsis, elsart, elsart_support, elsart_support_core, empheq, emulateapj, emulateapj5, endnotes, english, enumerate, enumitem, epigraph, epsf, epsfig, epstopdf, esint, espcrc, etex, etoolbox, eucal, eufrak, euler, eulervm, eurosym, euscript, expl3, exscale, fancybox, fancyhdr, fancyheadings, fancyvrb, feynmf, filehook, fix-cm, fixltx2e, fixme, flafter, fleqn, float, floatfig, floatflt, floatpag, flowchart, flushend, fontenc, fontspec, footmisc, footnote, fourier, framed, fullpage, gensymb, geometry, german, gettitlestring, glossaries, graphics, graphicx, grffile, helvet, hepunits, here, hhline, html, hypcap, hyperref, hyperxmp, icml, icml_support, ifdraft, ifetex, ifluatex, ifpdf, ifplatform, ifsym, iftex, ifthen, ifvtex, ifxetex, ijcai, import, indentfirst, infwarerr, inputenc, inst_support, intcalc, iopams, iopart_support, jheppub, keyval, kvdefinekeys, kvoptions, kvsetkeys, l3keys2e, lastpage, latexml, latexsym,

lineno, lipsum, listings, listingsutf8, lmodern, longtable, lscape, ltxcmds, luximono, lxRDFa, makecell, makeidx, marginnote, marvosym, mathabx, mathbbol, mathdots, mathpazo, mathpple, mathptm, mathptmx, mathrsfs, mathtools, microtype, mleftright, mn2e_support, multicol, multido, multirow, nameref, natbib, neurips, newcent, newfloat, newlfont, newtxmath, newtxtext, ngerman, nicefrac, nomencl, nopageno, ntheorem, numprint, overpic, palatino, paralist, parskip, pdflscape, pdfpages, pdfsync, pdftexcmds, pgf, pgfplots, pgfplotstable, physics, pifont, placeins, prettyref, preview, proof, proofwiki, psfig, psfrag, pslatex, pspicture, pst-grad, pst-node, pstricks, pxfonts, ragged2e, refcount, relsize, remreset, revsymb, revtex, revtex3_support, revtex4, revtex4_support, rotate, rotating, rsfs, scalefnt, sectsty, setspace, showkeys, sidecap, siunitx, slashbox, slashed, soul, srcltx, standalone, stfloats, stmaryrd, subcaption, subeqn, subeqnarray, subfig, subfigure, subfiles, subfloat, supertabular, sv_support, svg, t1enc, tablefootnote, tabularx, tabulary, tcolorbox, textcase, textcomp, texvc, theorem, thm-restate, thmtools, threeparttable, tikz, tikz-3dplot, tikzbricks, times, titlesec, titling, tocbibind, todonotes, tracefnt, transparent, turing, twoopt, txfonts, type1cm, ulem, underscore, undertilde, units, upgreek, upquote, upref, url, utopia, varioref, varwidth, verbatim, vmargin, wasysym, wiki, wrapfig, xargs, xcolor, xfrac, xkeyval, xkvview, xparse, xspace, xunicode, xurl, xy, xypic, yfonts

Appendix C

Perl Modules Documentation

LaTeXML

A converter that transforms TeX and LaTeX into XML/HTML/MathML

Synopsis

```
use LaTeXML;
my $converter = LaTeXML->get_converter($config);
my $converter = LaTeXML->new($config);
$converter->prepare_session($opts);
$converter->initialize_session; # SHOULD BE INTERNAL
$hashref = $converter->convert($tex);
my ($result,$log,$status)
     = map {$hashref->{$_}} qw(result log status);
```

Description

LaTeXML is a converter that transforms TeX and LaTeX into XML/HTML/MathML and other formats.

A LaTeXML object represents a converter instance and can convert files on demand, until dismissed.

Methods

`my $converter = LaTeXML->new($config);`

Creates a new converter object for a given LaTeXML::Common::Config object, $config.

`my $converter = LaTeXML->get_converter($config);`

Either creates, or looks up a cached converter for the $config configuration object.

$converter->prepare_session($opts);

Top-level preparation routine that prepares both a correct options object and an initialized LaTeXML object, using the ”initialize_options” and ”initialize_session” routines, when needed.

Contains optimization checks that skip initializations unless necessary.

Also adds support for partial option specifications during daemon runtime, falling back on the option defaults given when converter object was created.

my ($result,$status,$log) = $converter->convert($tex);

Converts a TeX input string $tex into the LaTeXML::Core::Document object $result.

Supplies detailed information of the conversion log ($log), as well as a brief conversion status summary ($status).

INTERNAL ROUTINES

$converter->initialize_session($opts);

Given an options hash reference $opts, initializes a session by creating a new LaTeXML object with initialized state and loading a daemonized preamble (if any).

Sets the ”ready” flag to true, making a subsequent ”convert” call immediately possible.

my $latexml = new_latexml($opts);

Creates a new LaTeXML object and initializes its state.

my $postdoc = $converter->convert_post($dom);

Post-processes a LaTeXML::Core::Document object $dom into a final format, based on the preferences specified in $$self{opts}.

Typically used only internally by `convert`.

$converter->bind_log;

Binds STDERR to a ”log” field in the $converter object

my $log = $converter->flush_log;

Flushes out the accumulated conversion log into $log, resetting STDERR to its usual stream.

LaTeXML::Global

Global exports used within LaTeXML, and in Packages.

Synopsis

use LaTeXML::Global;

Description

This module exports the various constants and constructors that are useful throughout LaTeXML, and in Package implementations.

Global state

$STATE;

This is bound to the currently active LaTeXML::Core::State by an instance of LaTeXML during processing.

LaTeXML::Package

Support for package implementations and document customization.

Synopsis

This package defines and exports most of the procedures users will need to customize or extend LaTeXML. The LaTeXML implementation of some package might look something like the following, but see the installed LaTeXML/Package directory for realistic examples.

```
package LaTeXML::Package::pool;  # to put new subs & variables in common pool
use LaTeXML::Package;            # to load these definitions
use strict;                      # good style
use warnings;
#
# Load "anotherpackage"
RequirePackage('anotherpackage');
#
# A simple macro, just like in TeX
DefMacro('\thesection', '\thechapter.\roman{section}');
#
# A constructor defines how a control sequence generates XML:
DefConstructor('\thanks{}', "<ltx:thanks>#1</ltx:thanks>");
#
# And a simple environment ...
DefEnvironment('{abstract}','<abstract>#body</abstract>');
#
# A math  symbol \Real to stand for the Reals:
DefMath('\Real', "\x{211D}", role=>'ID');
#
# Or a semantic floor:
DefMath('\floor{}','\left\lfloor#1\right\rfloor');
#
# More esoteric ...
# Use a RelaxNG schema
RelaxNGSchema("MySchema");
# Or use a special DocType if you have to:
```

```
# DocType("rootelement",
#         "-//Your Site//Your DocType",'your.dtd',
#          prefix=>"http://whatever/");
#
# Allow sometag elements to be automatically closed if needed
Tag('prefix:sometag', autoClose=>1);
#
# Don't forget this, so perl knows the package loaded.
1;
```

Description

This module provides a large set of utilities and declarations that are useful for writing 'bindings': LaTeXML-specific implementations of a set of control sequences such as would be defined in a LaTeX style or class file. They are also useful for controlling and customization of LaTeXML's processing. See the See also section, below, for additional lower-level modules imported & re-exported.

To a limited extent (and currently only when explicitly enabled), LaTeXML can process the raw TeX code found in style files. However, to preserve document structure and semantics, as well as for efficiency, it is usually necessary to supply a LaTeXML-specific 'binding' for style and class files. For example, a binding `mypackage.sty.ltxml` would encode LaTeXML-specific implementations of all the control sequences in `mypackage.sty` so that `\usepackage{mypackage}` would work. Similarly for `myclass.cls.ltxml`. Additionally, document-specific bindings can be supplied: before processing a TeX source file, eg `mydoc.tex`, LaTeXML will automatically include the definitions and settings in `mydoc.latexml`. These `.ltxml` and `.latexml` files should be placed LaTeXML's searchpaths, where will find them: either in the current directory or in a directory given to the --path option, or possibly added to the variable SEARCHPATHS).

Since LaTeXML mimics TeX, a familiarity with TeX's processing model is critical. LaTeXML models: catcodes and tokens (See LaTeXML::Core::Token, LaTeXML::Core::Tokens) which are extracted from the plain source text characters by the LaTeXML::Core::Mouth; Macros, which are expanded within the LaTeXML::Core::Gullet; and Primitives, which are digested within the LaTeXML::Core::Stomach to produce LaTeXML::Core::Box, LaTeXML::Core::List. A key additional feature is the Constructors: when digested they generate a LaTeXML::Core::Whatsit which, upon absorption by LaTeXML::Core::Document, inserts text or XML fragments in the final document tree.

Notation: Many of the following forms take code references as arguments or options. That is, either a reference to a defined sub, eg. `\&somesub`, or an anonymous function `sub { ... }`. To document these cases, and the arguments that are passed in each case, we'll use a notation like *code*`($stomach,...)`.

Control Sequences Many of the following forms define the behaviour of control sequences. While in TeX you'll typically only define macros, LaTeXML is effectively re-

defining TeX itself, so we define `Macros` as well as `Primitives`, `Registers`, `Constructors` and `Environments`. These define the behaviour of these control sequences when processed during the various phases of LaTeX's imitation of TeX's digestive tract.

Prototypes LaTeXML uses a more convenient method of specifying parameter patterns for control sequences. The first argument to each of these defining forms (`DefMacro`, `DefPrimive`, etc) is a *prototype* consisting of the control sequence being defined along with the specification of parameters required by the control sequence. Each parameter describes how to parse tokens following the control sequence into arguments or how to delimit them. To simplify coding and capture common idioms in TeX/LaTeX programming, latexml's parameter specifications are more expressive than TeX's `\def` or LaTeX's `\newcommand`. Examples of the prototypes for familiar TeX or LaTeX control sequences are:

```
DefConstructor('\usepackage[]{}',...
DefPrimitive('\multiply Variable SkipKeyword:by Number',..
DefPrimitive('\newcommand OptionalMatch:* DefToken[]{}', ...
```

The general syntax for parameter specification is

{*spec*}

reads a regular TeX argument. *spec* can be omitted (ie. {}). Otherwise *spec* is itself a parameter specification and the argument is reparsed to accordingly. ({} is a shorthand for `Plain`.)

[*spec*]

reads an LaTeX-style optional argument. *spec* can be omitted (ie. {}). Otherwise, if *spec* is of the form Default:stuff, then stuff would be the default value. Otherwise *spec* is itself a parameter specification and the argument, if supplied, is reparsed according to that specification. (`[]` is a shorthand for `Optional`.)

Type

Reads an argument of the given type, where either Type has been declared, or there exists a ReadType function accessible from LaTeXML::Package::Pool. See the available types, below.

***Type*:*value* | *Type*:*value1*:*value2*...**

These forms invoke the parser for *Type* but pass additional Tokens to the reader function. Typically this would supply defaults or parameters to a match.

Optional*Type*

Similar to *Type*, but it is not considered an error if the reader returns undef.

Skip*Type*

Similar to `Optional`*Type*, but the value returned from the reader is ignored, and does not occupy a position in the arguments list.

The predefined argument *Type*s are as follows.

Plain, Semiverbatim

Reads a standard TeX argument being either the next token, or if the next token is an {, the balanced token list. In the case of `Semiverbatim`, many catcodes are disabled, which is handy for URL's, labels and similar.

Token, XToken

Read a single TeX Token. For `XToken`, if the next token is expandable, it is repeatedly expanded until an unexpandable token remains, which is returned.

Number, Dimension, Glue | MuGlue

Read an Object corresponding to Number, Dimension, Glue or MuGlue, using TeX's rules for parsing these objects.

Until:*match* | XUntil:*match*>

Reads tokens until a match to the tokens *match* is found, returning the tokens preceding the match. This corresponds to TeX delimited arguments. For `XUntil`, tokens are expanded as they are matched and accumulated (but a brace reads and accumulates till a matching close brace, without expanding).

UntilBrace

Reads tokens until the next open brace {. This corresponds to the peculiar TeX construct `\def\foo#{...`

Match:*match*(|*match*)* | Keyword:*match*(|*match*)*>

Reads tokens expecting a match to one of the token lists *match*, returning the one that matches, or undef. For `Keyword`, case and catcode of the *matches* are ignored. Additionally, any leading spaces are skipped.

Balanced

Read tokens until a closing }, but respecting nested {} pairs.

BalancedParen

Read a parenthesis delimited tokens, but does *not* balance any nested parentheses.

Undigested, Digested, DigestUntil:*match*

These types alter the usual sequence of tokenization and digestion in separate stages (like TeX). A `Undigested` parameter inhibits digestion completely and remains in token form. A `Digested` parameter gets digested until the (required) opening { is balanced; this is useful when the content would usually need to have been protected in order to correctly deal with catcodes. `DigestUntil` digests tokens until a token matching *match* is found.

Variable

Reads a token, expanding if necessary, and expects a control sequence naming a writable register. If such is found, it returns an array of the corresponding definition object, and any arguments required by that definition.

SkipSpaces, Skip1Space

Skips one, or any number of, space tokens, if present, but contributes nothing to the argument list.

Common Options

scope=>'local' | 'global' | *scope*

Most defining commands accept an option to control how the definition is stored, for global or local definitions, or using a named *scope* A named scope saves a set of definitions and values that can be activated at a later time.

Particularly interesting forms of scope are those that get automatically activated upon changes of counter and label. For example, definitions that have `scope=>'section:1.1'` will be activated when the section number is "1.1", and will be deactivated when that section ends.

locked=>*boolean*

This option controls whether this definition is locked from further changes in the TeX sources; this keeps local 'customizations' by an author from overriding important LaTeXML definitions and breaking the conversion.

protected=>*boolean*

Makes a definition "protected", in the sense of eTeX's `\protected` directive. This inhibits expansion under certain circumstances.

robust=>*boolean*

Makes a definition "robust", in the sense of LaTeX's `\DeclareRobustCommand`. This essentially creates an indirect macro definition which is preceded by `\protect`. This inhibits expansion (and argument processing!) under certain circumstances. It usually only makes sense for macros, but may be useful for Primitives, Constructors and DefMath in cases where LaTeX would normally have created a macro that needs protection.

Macros

DefMacro(*prototype, expansion, %options*);

Defines the macro expansion for *prototype*; a macro control sequence that is expanded during macro expansion time in the `LaTeXML::Core::Gullet`. The *expansion* should be one of *tokens* | *string* | *code*($gullet,@args)>: a *string* will be tokenized upon first usage. Any macro arguments will be substituted for parameter indicators (eg #1) in the *tokens* or tokenized *string* and the result is

used as the expansion of the control sequence. If *code* is used, it is called at expansion time and should return a list of tokens as its result.

DefMacro options are

scope=>*scope*,

locked=>*boolean*

See Common Options.

mathactive=>*boolean*

specifies a definition that will only be expanded in math mode; the control sequence must be a single character.

Examples:

```
DefMacro('\thefootnote','\arabic{footnote}');
DefMacro('\today',sub { ExplodeText(today()); });
```

DefMacroI(*cs*, *paramlist*, *expansion*, *%options*);

Internal form of DefMacro where the control sequence and parameter list have already been separated; useful for definitions from within code. Also, slightly more efficient for macros with no arguments (use undef for *paramlist*), and useful for obscure cases like defining \begin{something*} as a Macro.

Conditionals

DefConditional(*prototype*, *test*, *%options*);

Defines a conditional for *prototype*; a control sequence that is processed during macro expansion time (in the LaTeXML::Core::Gullet). A conditional corresponds to a TeX \if. If the *test* is undef, a \newif type of conditional is defined, which is controlled with control sequences like \footrue and \foofalse. Otherwise the *test* should be *code*($gullet,@args) (with the control sequence's arguments) that is called at expand time to determine the condition. Depending on whether the result of that evaluation returns a true or false value (in the usual Perl sense), the result of the expansion is either the first or else code following, in the usual TeX sense.

DefConditional options are

scope=>*scope*,

locked=>*boolean*

See Common Options.

skipper=>*code*($gullet)

This option is *only* used to define \ifcase.

Example:

```
DefConditional('\ifmmode',sub {
   LookupValue('IN_MATH'); });
```

DefConditionalI(*cs*, *paramlist*, *test*, *%options*);

Internal form of DefConditional where the control sequence and parameter list have already been parsed; useful for definitions from within code. Also, slightly more efficient for conditinal with no arguments (use undef for paramlist).

IfCondition(*$ifcs*,*@args*)

IfCondition allows you to test a conditional from within perl. Thus something like if(IfCondition('\ifmmode')){ domath } else { dotext } might be equivalent to TeX's \ifmmode domath \else dotext \fi.

Primitives

DefPrimitive(*prototype*, *replacement*, *%options*);

Defines a primitive control sequence; a primitive is processed during digestion (in the LaTeXML::Core::Stomach), after macro expansion but before Construction time. Primitive control sequences generate Boxes or Lists, generally containing basic Unicode content, rather than structured XML. Primitive control sequences are also executed for side effect during digestion, effecting changes to the LaTeXML::Core::State.

The *replacement* can be a string used as the text content of a Box to be created (using the current font). Alternatively *replacement* can be *code*($stomach,@args) (with the control sequence's arguments) which is invoked at digestion time, probably for side-effect, but returning Boxes or Lists or nothing. *replacement* may also be undef, which contributes nothing to the document, but does record the TeX code that created it.

DefPrimitive options are

scope=>*scope*,

locked=>*boolean*

See Common Options.

mode=> ('text' | 'display_math' | 'inline_math')

Changes to this mode during digestion.

font=>{*%fontspec*}

Specifies the font to use (see Fonts). If the font change is to only apply to material generated within this command, you would also use <bounded=1>>; otherwise, the font will remain in effect afterwards as for a font switching command.

bounded=>*boolean*

If true, TeX grouping (ie. {}) is enforced around this invocation.

requireMath=>*boolean*,

forbidMath=>*boolean*

specifies whether the given constructor can *only* appear, or *cannot* appear, in math mode.

beforeDigest=>*code*($stomach)

supplies a hook to execute during digestion just before the main part of the primitive is executed (and before any arguments have been read). The *code* should either return nothing (return;) or a list of digested items (Box's,List,Whatsit). It can thus change the State and/or add to the digested output.

afterDigest=>*code*($stomach)

supplies a hook to execute during digestion just after the main part of the primitive ie executed. it should either return nothing (return;) or digested items. It can thus change the State and/or add to the digested output.

isPrefix=>*boolean*

indicates whether this is a prefix type of command; This is only used for the special TeX assignment prefixes, like \global.

Example:

```
DefPrimitive('\begingroup',sub { $_[0]->begingroup; });
```

DefPrimitiveI(*cs*, *paramlist*, *code*($stomach,@args), *%options*);

Internal form of DefPrimitive where the control sequence and parameter list have already been separated; useful for definitions from within code.

Registers

DefRegister(*prototype*, *value*, *%options*);

Defines a register with *value* as the initial value (a Number, Dimension, Glue, MuGlue or Tokens --- I haven't handled Box's yet). Usually, the *prototype* is just the control sequence, but registers are also handled by prototypes like \count{Number}. DefRegister arranges that the register value can be accessed when a numeric, dimension, ... value is being read, and also defines the control sequence for assignment.

Options are

readonly=>*boolean*

specifies if it is not allowed to change this value.

getter=>*code*(@args),

setter=>*code*($value,$scope,@args)

By default *value* is stored in the State's Value table under a name concatenating the control sequence and argument values. These options allow other means of fetching and storing the value.

Example:

```
DefRegister('\pretolerance',Number(100));
```

DefRegisterI(*cs*, *paramlist*, *value*, *%options*);

Internal form of `DefRegister` where the control sequence and parameter list have already been parsed; useful for definitions from within code.

Constructors

DefConstructor(*prototype*, *$replacement*, *%options*);

The Constructor is where LaTeXML really starts getting interesting; invoking the control sequence will generate an arbitrary XML fragment in the document tree. More specifically: during digestion, the arguments will be read and digested, creating a `LaTeXML::Core::Whatsit` to represent the object. During absorption by the `LaTeXML::Core::Document`, the `Whatsit` will generate the XML fragment according to *replacement*. The *replacement* can be *code*`($document,@args,%properties)` which is called during document absorption to create the appropriate XML (See the methods of `LaTeXML::Core::Document`).

More conveniently, *replacement* can be an pattern: simply a bit of XML as a string with certain substitutions to be made. The substitutions are of the following forms:

#1, #2 ... #name

These are replaced by the corresponding argument (for #1) or property (for #name) stored with the Whatsit. Each are turned into a string when it appears as in an attribute position, or recursively processed when it appears as content.

&*function*(@args)

Another form of substituted value is prefixed with `&` which invokes a function. For example, `&func(#1)` would invoke the function `func` on the first argument to the control sequence; what it returns will be inserted into the document.

?*test*(*pattern*) or ?*test*(*ifpattern*)(*elsepattern*)

Patterns can be conditionallized using this form. The *test* is any of the above expressions (eg. `#1`), considered true if the result is non-empty. Thus `?#1(<foo/>)` would add the empty element `foo` if the first argument were given.

^

If the constructor *begins* with ^, the XML fragment is allowed to *float up* to a parent node that is allowed to contain it, according to the Document Type.

The Whatsit property `font` is defined by default. Additional properties `body` and `trailer` are defined when `captureBody` is true, or for environments. By using `$whatsit->setProperty(key=>$value);` within `afterDigest`, or by using the `properties` option, other properties can be added.

DefConstructor options are

scope=>*scope*,

locked=>*boolean*

See Common Options.

mode=>*mode*,

font=>{*%fontspec*},

bounded=>*boolean*,

requireMath=>*boolean*,

forbidMath=>*boolean*

These options are the same as for Primitives

reversion=>*texstring* | *code*($whatsit,#1,#2,...)

specifies the reversion of the invocation back into TeX tokens (if the default reversion is not appropriate). The *textstring* string can include `#1`, `#2`... The *code* is called with the `$whatsit` and digested arguments and must return a list of Token's.

alias=>*control_sequence*

provides a control sequence to be used in the `reversion` instead of the one defined in the `prototype`. This is a convenient alternative for reversion when a 'public' command conditionally expands into an internal one, but the reversion should be for the public command.

sizer=>*string* | *code*($whatsit)

specifies how to compute (approximate) the displayed size of the object, if that size is ever needed (typically needed for graphics generation). If a string is given, it should contain only a sequence of `#1` or `#name` to access arguments and properties of the Whatsit: the size is computed from these items layed out side-by-side. If *code* is given, it should return the three Dimensions (width, height and depth). If neither is given, and the `reversion` specification is of suitible format, it will be used for the sizer.

properties=>{*%properties*} | *code*($stomach,#1,#2...)

supplies additional properties to be set on the generated Whatsit. In the first form, the values can be of any type, but if a value is a code references, it

takes the same args ($stomach,#1,#2,...) and should return the value; it is executed before creating the Whatsit. In the second form, the code should return a hash of properties.

beforeDigest=>*code*($stomach)

supplies a hook to execute during digestion just before the Whatsit is created. The *code* should either return nothing (return;) or a list of digested items (Box's,List,Whatsit). It can thus change the State and/or add to the digested output.

afterDigest=>*code*($stomach,$whatsit)

supplies a hook to execute during digestion just after the Whatsit is created (and so the Whatsit already has its arguments and properties). It should either return nothing (return;) or digested items. It can thus change the State, modify the Whatsit, and/or add to the digested output.

beforeConstruct=>*code*($document,$whatsit)

supplies a hook to execute before constructing the XML (generated by *replacement*).

afterConstruct=>*code*($document,$whatsit)

Supplies *code* to execute after constructing the XML.

captureBody=>*boolean* | *Token*

if true, arbitrary following material will be accumulated into a 'body' until the current grouping level is reverted, or till the `Token` is encountered if the option is a `Token`. This body is available as the `body` property of the Whatsit. This is used by environments and math.

nargs=>*nargs*

This gives a number of args for cases where it can't be inferred directly from the *prototype* (eg. when more args are explicitly read by hooks).

DefConstructorI(*cs*, *paramlist*, *replacement*, *%options*);

Internal form of `DefConstructor` where the control sequence and parameter list have already been separated; useful for definitions from within code.

DefMath(*prototype*, *tex*, *%options*);

A common shorthand constructor; it defines a control sequence that creates a mathematical object, such as a symbol, function or operator application. The options given can effectively create semantic macros that contribute to the eventual parsing of mathematical content. In particular, it generates an XMDual using the replacement *tex* for the presentation. The content information is drawn from the name and options

`DefMath` accepts the options:

scope=>*scope*,

locked=>*boolean*

See `Common Options`.

font=>{%*fontspec*},

reversion=>*reversion*,

alias=>*cs*,

sizer=>*sizer*,

properties=>*properties*,

beforeDigest=>*code*($stomach),

afterDigest=>*code*($stomach,$whatsit),

These options are the same as for Constructors

name=>*name*

gives a name attribute for the object

omcd=>*cdname*

gives the OpenMath content dictionary that name is from.

role=>*grammatical_role*

adds a grammatical role attribute to the object; this specifies the grammatical role that the object plays in surrounding expressions. This direly needs documentation!

mathstyle=>('display' | 'text' | 'script' | 'scriptscript')

Controls whether the this object will be presented in a specific mathstyle, or according to the current setting of mathstyle.

scriptpos=>('mid' | 'post')

Controls the positioning of any sub and super-scripts relative to this object; whether they be stacked over or under it, or whether they will appear in the usual position. TeX.pool defines a function doScriptpos() which is useful for operators like \sum in that it sets to mid position when in displaystyle, otherwise post.

stretchy=>*boolean*

Whether or not the object is stretchy when displayed.

operator_role=>*grammatical_role*,

operator_scriptpos=>*boolean*,

operator_stretchy=>*boolean*

These three are similar to role, scriptpos and stretchy, but are used in unusual cases. These apply to the given attributes to the operator token in the content branch.

nogroup=>*boolean*

Normally, these commands are digested with an implicit grouping around them, localizing changes to fonts, etc; noggroup=>1 inhibits this.

Example:

```
DefMath('\infty',"\x{221E}",
   role=>'ID', meaning=>'infinity');
```

DefMathI(*cs*, *paramlist*, *tex*, *%options*);

Internal form of DefMath where the control sequence and parameter list have already been separated; useful for definitions from within code.

Environments

DefEnvironment(*prototype*, *replacement*, *%options*);

Defines an Environment that generates a specific XML fragment. replacement is of the same form as for DefConstructor, but will generally include reference to the #body property. Upon encountering a \begin{env}: the mode is switched, if needed, else a new group is opened; then the environment name is noted; the beforeDigest hook is run. Then the Whatsit representing the begin command (but ultimately the whole environment) is created and the afterDigestBegin hook is run. Next, the body will be digested and collected until the balancing \end{env}. Then, any afterDigest hook is run, the environment is ended, finally the mode is ended or the group is closed. The body and \end{env} whatsit are added to the \begin{env}'s whatsit as body and trailer, respectively.

DefEnvironment takes the following options:

scope=>*scope*,

locked=>*boolean*

See Common Options.

mode=>*mode*,

font=>{*%fontspec*}

requireMath=>*boolean*,

forbidMath=>*boolean*,

These options are the same as for Primitives

reversion=>*reversion*,

alias=>*cs*,

sizer=>*sizer*,

properties=>*properties*,

nargs=>*nargs*

These options are the same as for Constructors

beforeDigest=>*code*($stomach)

This hook is similar to that for DefConstructor, but it applies to the \begin{environment} control sequence.

afterDigestBegin=>*code*($stomach,$whatsit)

This hook is similar to DefConstructor's afterDigest but it applies to the \begin{environment} control sequence. The Whatsit is the one for the beginning control sequence, but represents the environment as a whole. Note that although the arguments and properties are present in the Whatsit, the body of the environment is *not* yet available!

beforeDigestEnd=>*code*($stomach)

This hook is similar to DefConstructor's beforeDigest but it applies to the \end{environment} control sequence.

afterDigest=>*code*($stomach,$whatsit)

This hook is similar to DefConstructor's afterDigest but it applies to the \end{environment} control sequence. Note, however that the Whatsit is only for the ending control sequence, *not* the Whatsit for the environment as a whole.

afterDigestBody=>*code*($stomach,$whatsit)

This option supplies a hook to be executed during digestion after the ending control sequence has been digested (and all the 4 other digestion hook have executed) and after the body of the environment has been obtained. The Whatsit is the (useful) one representing the whole environment, and it now does have the body and trailer available, stored as a properties.

Example:

```
DefConstructor('\emph{}',
   "<ltx:emph>#1</ltx:emph", mode=>'text');
```

DefEnvironmentI(*name*, *paramlist*, *replacement*, *%options*);

Internal form of DefEnvironment where the control sequence and parameter list have already been separated; useful for definitions from within code.

Inputing Content and Definitions

FindFile(*name*, *%options*);

Find an appropriate file with the given *name* in the current directories in SEARCHPATHS. If a file ending with .ltxml is found, it will be preferred.

Note that if the name starts with a recognized *protocol* (currently one of (literal|http|https|ftp)) followed by a colon, the name is returned, as is, and no search for files is carried out.

The options are:

type=>*type*

specifies the file type. If not set, it will search for both *name*.tex and *name*.

noltxml=>1

inhibits searching for a LaTeXML binding (*name*.*type*.ltxml) to use instead of the file itself.

notex=>1

inhibits searching for raw tex version of the file. That is, it will *only* search for the LaTeXML binding.

InputContent(*request*, *%options*);

InputContent is used for cases when the file (or data) is plain TeX material that is expected to contribute content to the document (as opposed to pure definitions). A Mouth is opened onto the file, and subsequent reading and/or digestion will pull Tokens from that Mouth until it is exhausted, or closed.

In some circumstances it may be useful to provide a string containing the TeX material explicitly, rather than referencing a file. In this case, the literal pseudo-protocal may be used:

```
InputContent('literal:\textit{Hey}');
```

If a file named $request.latexml exists, it will be read in as if it were a latexml binding file, before processing. This can be used for adhoc customization of the conversion of specific files, without modifying the source, or creating more elaborate bindings.

The only option to InputContent is:

noerror=>*boolean*

Inhibits signalling an error if no appropriate file is found.

Input(*request*);

Input is analogous to LaTeX's \input, and is used in cases where it isn't completely clear whether content or definitions is expected. Once a file is found, the approach specified by InputContent or InputDefinitions is used, depending on which type of file is found.

InputDefinitions(*request*, *%options*);

InputDefinitions is used for loading *definitions*, ie. various macros, settings, etc, rather than document content; it can be used to load LaTeXML's binding files, or for reading in raw TeX definitions or style files. It reads and processes the material completely before returning, even in the case of TeX definitions. This procedure optionally supports the conventions used for standard LaTeX packages and classes (see RequirePackage and LoadClass).

Options for InputDefinitions are:

type=>*type*

the file type to search for.

noltxml=>*boolean*
: inhibits searching for a LaTeXML binding; only raw TeX files will be sought and loaded.

notex=>*boolean*
: inhibits searching for raw TeX files, only a LaTeXML binding will be sought and loaded.

noerror=>*boolean*
: inhibits reporting an error if no appropriate file is found.

The following options are primarily useful when `InputDefinitions` is supporting standard LaTeX package and class loading.

withoptions=>*boolean*
: indicates whether to pass in any options from the calling class or package.

handleoptions=>*boolean*
: indicates whether options processing should be handled.

options=>[...]
: specifies a list of options (in the 'package options' sense) to be passed (possibly in addition to any provided by the calling class or package).

after=>*tokens* | *code*($gullet)
: provides *tokens* or *code* to be processed by a *name*.*type*-h@@k macro.

as_class=>*boolean*
: fishy option that indicates that this definitions file should be treated as if it were defining a class; typically shows up in latex compatibility mode, or AMSTeX.

A handy method to use most of the TeX distribution's raw TeX definitions for a package, but override only a few with LaTeXML bindings is by defining a binding file, say `tikz.sty.ltxml`, to contain

```
InputDefinitions('tikz', type => 'sty', noltxml => 1);
```

which would find and read in `tizk.sty`, and then follow it by a couple of strategic LaTeXML definitions, `DefMacro`, etc.

Class and Packages

RequirePackage(*package*, %*options*);

Finds and loads a package implementation (usually *package*`.sty.ltxml`, unless `noltxml` is specified)for the requested *package*. It returns the pathname of the loaded package. The options are:

type=>*type*
: specifies the file type (default `sty`.

options=>[...]

specifies a list of package options.

noltxml=>*boolean*

inhibits searching for the LaTeXML binding for the file (ie. *name*.*type*.ltxml

notex=>1

inhibits searching for raw tex version of the file. That is, it will *only* search for the LaTeXML binding.

LoadClass(*class*, *%options*);

Finds and loads a class definition (usually *class*.cls.ltxml). It returns the pathname of the loaded class. The only option is

options=>[...]

specifies a list of class options.

LoadPool(*pool*, *%options*);

Loads a *pool* file (usually *pool*.pool.ltxml), one of the top-level definition files, such as TeX, LaTeX or AMSTeX. It returns the pathname of the loaded file.

DeclareOption(*option*, *tokens* | *string* | *code*($stomach));

Declares an option for the current package or class. The 2nd argument can be a *string* (which will be tokenized and expanded) or *tokens* (which will be macro expanded), to provide the value for the option, or it can be a code reference which is treated as a primitive for side-effect.

If a package or class wants to accommodate options, it should start with one or more DeclareOptions, followed by ProcessOptions().

PassOptions(*name*, *ext*, *@options*);

Causes the given @*options* (strings) to be passed to the package (if *ext* is sty) or class (if *ext* is cls) named by *name*.

ProcessOptions(*%options*);

Processes the options that have been passed to the current package or class in a fashion similar to LaTeX. The only option (to ProcessOptions is inorder=>*boolean* indicating whehter the (package) options are processed in the order they were used, like ProcessOptions*.

This will also process a limited form of keyval class and package options, if option keysets provides a list of keyval set names, and option inorder is true.

ExecuteOptions(*@options*);

Process the options given explicitly in @*options*.

AtBeginDocument(*@stuff*);

Arranges for *@stuff* to be carried out after the preamble, at the beginning of the document. *@stuff* should typically be macro-level stuff, but carried out for side effect; it should be tokens, tokens lists, strings (which will be tokenized), or *code*($gullet) which would yield tokens to be expanded.

This operation is useful for style files loaded with --preload or document specific customization files (ie. ending with .latexml); normally the contents would be executed before LaTeX and other style files are loaded and thus can be overridden by them. By deferring the evaluation to begin-document time, these contents can override those style files. This is likely to only be meaningful for LaTeX documents.

AtEndDocument(*@stuff*)

Arranges for *@stuff* to be carried out just before \\end{document}. These tokens can be used for side effect, or any content they generate will appear as the last children of the document.

Counters and IDs

NewCounter(*ctr*, *within*, *%options*);

Defines a new counter, like LaTeX's \newcounter, but extended. It defines a counter that can be used to generate reference numbers, and defines \the*ctr*, etc. It also defines an "uncounter" which can be used to generate ID's (xml:id) for unnumbered objects. *ctr* is the name of the counter. If defined, *within* is the name of another counter which, when incremented, will cause this counter to be reset. The options are

idprefix=>*string*

Specifies a prefix to be used to generate ID's when using this counter

nested

Not sure that this is even sane.

$num = CounterValue($ctr);

Fetches the value associated with the counter $ctr.

$tokens = StepCounter($ctr);

Analog of \stepcounter, steps the counter and returns the expansion of \the$ctr. Usually you should use RefStepCounter($ctr) instead.

$keys = RefStepCounter($ctr);

Analog of \refstepcounter, steps the counter and returns a hash containing the keys refnum=$refnum, id=>$id>. This makes it suitable for use in a properties option to constructors. The id is generated in parallel with the reference number to assist debugging.

$keys = RefStepID($ctr);

Like to RefStepCounter, but only steps the "uncounter", and returns only the id; This is useful for unnumbered cases of objects that normally get both a refnum and id.

ResetCounter($ctr);

Resets the counter $ctr to zero.

GenerateID($document,$node,$whatsit,$prefix);

Generates an ID for nodes during the construction phase, useful for cases where the counter based scheme is inappropriate. The calling pattern makes it appropriate for use in Tag, as in

```
Tag('ltx:para',afterClose=>sub { GenerateID(@_,'p'); })
```

If $node doesn't already have an xml:id set, it computes an appropriate id by concatenating the xml:id of the closest ancestor with an id (if any), the prefix (if any) and a unique counter.

Document Model Constructors define how TeX markup will generate XML fragments, but the Document Model is used to control exactly how those fragments are assembled.

Tag(*tag, %properties*);

Declares properties of elements with the name *tag*. Note that Tag can set or add properties to any element from any binding file, unlike the properties set on control by DefPrimtive, DefConstructor, etc.. And, since the properties are recorded in the current Model, they are not subject to TeX grouping; once set, they remain in effect until changed or the end of the document.

The *tag* can be specified in one of three forms:

```
prefix:name matches specific name in specific namespace
prefix:*    matches any tag in the specific namespace;
*           matches any tag in any namespace.
```

There are two kinds of properties:

Scalar properties

For scalar properties, only a single value is returned for a given element. When the property is looked up, each of the above forms is considered (the specific element name, the namespace, and all elements); the first defined value is returned.

The recognized scalar properties are:

autoOpen=>*boolean*

Specifies whether *tag* can be automatically opened if needed to insert an element that can only be contained by *tag*. This property can help match the more SGML-like LaTeX to XML.

autoClose=>*boolean*

Specifies whether this *tag* can be automatically closed if needed to close an ancestor node, or insert an element into an ancestor. This property can help match the more SGML-like LaTeX to XML.

Code properties

These properties provide a bit of code to be run at the times of certain events associated with an element. *All* the code bits that match a given element will be run, and since they can be added by any binding file, and be specified in a random orders, a little bit of extra control is desirable.

Firstly, any *early* codes are run (eg `afterOpen:early`), then any normal codes (without modifier) are run, and finally any *late* codes are run (eg. `afterOpen:late`).

Within *each* of those groups, the codes assigned for an element's specific name are run first, then those assigned for its package and finally the generic one (`*`); that is, the most specific codes are run first.

When code properties are accumulated by `Tag` for normal or late events, the code is appended to the end of the current list (if there were any previous codes added); for early event, the code is prepended.

The recognized code properties are:

afterOpen=>*code*($document,$box)

Provides *code* to be run whenever a node with this *tag* is opened. It is called with the document being constructed, and the initiating digested object as arguments. It is called after the node has been created, and after any initial attributes due to the constructor (passed to openElement) are added.

`afterOpen:early` or `afterOpen:late` can be used in place of `afterOpen`; these will be run as a group before, or after (respectively) the unmodified blocks.

afterClose=>*code*($document,$box)

Provides *code* to be run whenever a node with this *tag* is closed. It is called with the document being constructed, and the initiating digested object as arguments.

`afterClose:early` or `afterClose:late` can be used in place of `afterClose`; these will be run as a group bfore, or after (respectively) the unmodified blocks.

RelaxNGSchema(*schemaname*);

Specifies the schema to use for determining document model. You can leave off the extension; it will look for *schemaname*`.rng` (and maybe eventually, `.rnc` if that is ever implemented).

RegisterNamespace(*prefix, URL*);

Declares the *prefix* to be associated with the given *URL*. These prefixes may be used in ltxml files, particularly for constructors, xpath expressions, etc. They are not necessarily the same as the prefixes that will be used in the generated document Use the prefix `#default` for the default, non-prefixed, namespace. (See RegisterDocumentNamespace, as well as DocType or RelaxNGSchema).

RegisterDocumentNamespace(*prefix, URL*);

Declares the *prefix* to be associated with the given *URL* used within the generated XML. They are not necessarily the same as the prefixes used in code (RegisterNamespace). This function is less rarely needed, as the namespace declarations are generally obtained from the DTD or Schema themselves Use the prefix `#default` for the default, non-prefixed, namespace. (See DocType or RelaxNGSchema).

DocType(*rootelement, publicid, systemid, %namespaces*);

Declares the expected *rootelement*, the public and system ID's of the document type to be used in the final document. The hash *%namespaces* specifies the namespaces prefixes that are expected to be found in the DTD, along with each associated namespace URI. Use the prefix `#default` for the default namespace (ie. the namespace of non-prefixed elements in the DTD).

The prefixes defined for the DTD may be different from the prefixes used in implementation CODE (eg. in ltxml files; see RegisterNamespace). The generated document will use the namespaces and prefixes defined for the DTD.

Document Rewriting During document construction, as each node gets closed, the text content gets simplfied. We'll call it *applying ligatures*, for lack of a better name.

DefLigature(*regexp, %options*);

Apply the regular expression (given as a string: "/fa/fa/" since it will be converted internally to a true regexp), to the text content. The only option is `fontTest=>`*`code`*`($font)`; if given, then the substitution is applied only when `fontTest` returns true.

Predefined Ligatures combine sequences of "." or single-quotes into appropriate Unicode characters.

DefMathLigature(*$string=$replacment,%options*);>

A Math Ligature typically combines a sequence of math tokens (XMTok) into a single one. A simple example is

```
DefMathLigature(":=" => ":=", role => 'RELOP', meaning => 'assign');
```

replaces the two tokens for colon and equals by a token representing assignment. The options are those characterising an XMTok, namely: `role`, `meaning` and `name`.

For more complex cases (recognizing numbers, for example), you may supply a function `matcher`=CODE($document,$node)>, which is passed the current document and the last math node in the sequence. It should examine `$node` and any preceding nodes (using `previousSibling`) and return a list of `($n,$string,%attributes)` to replace the `$n` nodes by a new one with text content being `$string` content and the given attributes. If no replacement is called for, CODE should return undef.

After document construction, various rewriting and augmenting of the document can take place.

DefRewrite(%*specification*);

DefMathRewrite(%*specification*);

These two declarations define document rewrite rules that are applied to the document tree after it has been constructed, but before math parsing, or any other postprocessing, is done. The *%specification* consists of a sequence of key/value pairs with the initial specs successively narrowing the selection of document nodes, and the remaining specs indicating how to modify or replace the selected nodes.

The following select portions of the document:

label=>*label*

Selects the part of the document with label=$label

scope=>*scope*

The *scope* could be ”label:foo” or ”section:1.2.3” or something similar. These select a subtree labelled ’foo’, or a section with reference number ”1.2.3”

xpath=>*xpath*

Select those nodes matching an explicit xpath expression.

match=>*tex*

Selects nodes that look like what the processing of *tex* would produce.

regexp=>*regexp*

Selects text nodes that match the regular expression.

The following act upon the selected node:

attributes=>*hashref*

Adds the attributes given in the hash reference to the node.

replace=>*replacement*

Interprets *replacement* as TeX code to generate nodes that will replace the selected nodes.

Mid-Level support

`$tokens = Expand($tokens);`

Expands the given `$tokens` according to current definitions.

`$boxes = Digest($tokens);`

Processes and digestes the `$tokens`. Any arguments needed by control sequences in `$tokens` must be contained within the `$tokens` itself.

`@tokens = Invocation($cs,@args);`

Constructs a sequence of tokens that would invoke the token `$cs` on the arguments.

`RawTeX('... tex code ...');`

RawTeX is a convenience function for including chunks of raw TeX (or LaTeX) code in a Package implementation. It is useful for copying portions of the normal implementation that can be handled simply using macros and primitives.

`Let($token1,$token2);`

Gives `$token1` the same 'meaning' (definition) as `$token2`; like TeX's \let.

`StartSemiVerbatim(); ... ; EndSemiVerbatim();`

Disable disable most TeX catcodes.

`$tokens = Tokenize($string);`

Tokenizes the `$string` using the standard catcodes, returning a `LaTeXML::Core::Tokens`.

`$tokens = TokenizeInternal($string);`

Tokenizes the `$string` according to the internal cattable (where @ is a letter), returning a `LaTeXML::Core::Tokens`.

Argument Readers

`ReadParameters($gullet,$spec);`

Reads from `$gullet` the tokens corresponding to `$spec` (a Parameters object).

`DefParameterType(`*`type`*`, `*`code`*`($gullet,@values), %`*`options`*`);`

Defines a new Parameter type, *type*, with *code* for its reader.

Options are:

`reversion=>`*`code`*`($arg,@values);`

This *code* is responsible for converting a previously parsed argument back into a sequence of Token's.

optional=>*boolean*

whether it is an error if no matching input is found.

novalue=>*boolean*

whether the value returned should contribute to argument lists, or simply be passed over.

semiverbatim=>*boolean*

whether the catcode table should be modified before reading tokens.

<DefColumnType(*proto*, *expansion*);

Defines a new column type for tabular and arrays. *proto* is the prototype for the pattern, analogous to the pattern used for other definitions, except that macro being defined is a single character. The *expansion* is a string specifying what it should expand into, typically more verbose column specification.

Access to State

$value = LookupValue($name);

Lookup the current value associated with the the string `$name`.

AssignValue($name,$value,$scope);

Assign $value to be associated with the the string `$name`, according to the given scoping rule.

Values are also used to specify most configuration parameters (which can therefore also be scoped). The recognized configuration parameters are:

```
STRICT            : whether errors (eg. undefined macros)
                    are fatal.
INCLUDE_COMMENTS  : whether to preserve comments in the
                    source, and to add occasional line
                    number comments. (Default true).
PRESERVE_NEWLINES : whether newlines in the source should
                    be preserved (not 100% TeX-like).
                    By default this is true.
SEARCHPATHS       : a list of directories to search for
                    sources, implementations, etc.
```

PushValue($name,@values);

This function, along with the next three are like `AssignValue`, but maintain a global list of values. `PushValue` pushes the provided values onto the end of a list. The data stored for `$name` is global and must be a LIST reference; it is created if needed.

UnshiftValue($name,@values);

Similar to `PushValue`, but pushes a value onto the front of the list. The data stored for `$name` is global and must be a LIST reference; it is created if needed.

PopValue($name);

Removes and returns the value on the end of the list named by $name. The data stored for $name is global and must be a LIST reference. Returns undef if there is no data in the list.

ShiftValue($name);

Removes and returns the first value in the list named by $name. The data stored for $name is global and must be a LIST reference. Returns undef if there is no data in the list.

LookupMapping($name,$key);

This function maintains a hash association named by $name. It returns the value associated with $key within that mapping. The data stored for $name is global and must be a HASH reference. Returns undef if there is no data associated with $key in the mapping, or the mapping is not (yet) defined.

AssignMapping($name,$key,$value);

This function associates $value with $key within the mapping named by $name. The data stored for $name is global and must be a HASH reference; it is created if needed.

$value = LookupCatcode($char);

Lookup the current catcode associated with the the character $char.

AssignCatcode($char,$catcode,$scope);

Set $char to have the given $catcode, with the assignment made according to the given scoping rule.

This method is also used to specify whether a given character is active in math mode, by using math:$char for the character, and using a value of 1 to specify that it is active.

$meaning = LookupMeaning($token);

Looks up the current meaning of the given $token which may be a Definition, another token, or the token itself if it has not otherwise been defined.

$defn = LookupDefinition($token);

Looks up the current definition, if any, of the $token.

InstallDefinition($defn);

Install the Definition $defn into $STATE under its control sequence.

XEquals($token1,$token2)

Tests whether the two tokens are equal in the sense that they are either equal tokens, or if defined, have the same definition.

Fonts

MergeFont(%*fontspec*);

Set the current font by merging the font style attributes with the current font. The *%fontspec* specifies the properties of the desired font. Likely values include (the values aren't required to be in this set):

```
family : serif, sansserif, typewriter, caligraphic,
         fraktur, script
series : medium, bold
shape  : upright, italic, slanted, smallcaps
size   : tiny, footnote, small, normal, large,
         Large, LARGE, huge, Huge
color  : any named color, default is black
```

Some families will only be used in math. This function returns nothing so it can be easily used in beforeDigest, afterDigest.

DeclareFontMap($name,$map,%options);

Declares a font map for the encoding `$name`. The map `$map` is an array of 128 or 256 entries, each element is either a unicode string for the representation of that codepoint, or undef if that codepoint is not supported by this encoding. The only option currently is `family` used because some fonts (notably cmr!) have different glyphs in some font families, such as `family`='typewriter'>.

FontDecode($code,$encoding,$implicit);

Returns the unicode string representing the given codepoint `$code` (an integer) in the given font encoding `$encoding`. If `$encoding` is undefined, the usual case, the current font encoding and font family is used for the lookup. Explicit decoding is used when `\\char` or similar are invoked (`$implicit` is false), and the codepoint must be represented in the fontmap, otherwise undef is returned. Implicit decoding (ie. `$implicit` is true) occurs within the Stomach when a Token's content is being digested and converted to a Box; in that case only the lower 128 codepoints are converted; all codepoints above 128 are assumed to already be Unicode.

The font map for `$encoding` is automatically loaded if it has not already been loaded.

FontDecodeString($string,$encoding,$implicit);

Returns the unicode string resulting from decoding the individual characters in `$string` according to *FontDecode*, above.

LoadFontMap($encoding);

Finds and loads the font map for the encoding named `$encoding`, if it hasn't been loaded before. It looks for `encoding.fontmap.ltxml`, which would typically define the font map using `DeclareFontMap`, possibly including extra maps for families like `typewriter`.

Color

`$color=LookupColor($name);`

Lookup the color object associated with `$name`.

`DefColor($name,$color,$scope);`

Associates the `$name` with the given `$color` (a color object), with the given scoping.

`DefColorModel($model,$coremodel,$tocore,$fromcore);`

Defines a color model `$model` that is derived from the core color model `$coremodel`. The two functions `$tocore` and `$fromcore` convert a color object in that model to the core model, or from the core model to the derived model. Core models are rgb, cmy, cmyk, hsb and gray.

Low-level Functions

`CleanID($id);`

Cleans an `$id` of disallowed characters, trimming space.

`CleanLabel($label,$prefix);`

Cleans a `$label` of disallowed characters, trimming space. The prefix `$prefix` is prepended (or `LABEL`, if none given).

`CleanIndexKey($key);`

Cleans an index key, so it can be used as an ID.

`CleanBibKey($key);`

Cleans a bibliographic citation key, so it can be used as an ID.

`CleanURL($url);`

Cleans a url.

`UTF($code);`

Generates a UTF character, handy for the the 8 bit characters. For example, `UTF(0xA0)` generates the non-breaking space.

`@tokens = roman($number);`

Formats the `$number` in (lowercase) roman numerals, returning a list of the tokens.

`@tokens = Roman($number);`

Formats the `$number` in (uppercase) roman numerals, returning a list of the tokens.

See also

See also LaTeXML::Global, LaTeXML::Common::Object, LaTeXML::Common::Error, LaTeXML::Core::Token, LaTeXML::Core::Tokens, LaTeXML::Core::Box, LaTeXML::Core::List, LaTeXML::Common::Number, LaTeXML::Common::Float, LaTeXML::Common::Dimension, LaTeXML::Common::Glue, LaTeXML::Core::MuDimension LaTeXML::Core::MuGlue, LaTeXML::Core::Pair, LaTeXML::Core::PairList, LaTeXML::Common::Color, LaTeXML::Core::Alignment, LaTeXML::Common::XML, LaTeXML::Util::Radix.

LaTeXML::MathParser

Parses mathematics content

Description

LaTeXML::MathParser parses the mathematical content of a document. It uses *Parse::RecDescent* and a grammar MathGrammar.

Math Representation Needs description.

Possibile Customizations Needs description.

Convenience functions The following functions are exported for convenience in writing the grammar productions.

$node = New($name,$content,%attributes);

Creates a new XMTok node with given $name (a string or undef), and $content (a string or undef) (but at least one of name or content should be provided), and attributes.

$node = Arg($node,$n);

Returns the $n-th argument of an XMApp node; 0 is the operator node.

Annotate($node,%attributes);

Add attributes to $node.

$node = Apply($op,@args);

Create a new XMApp node representing the application of the node $op to the nodes @args.

$node = ApplyDelimited($op,@stuff);

Create a new XMApp node representing the application of the node $op to the arguments found in @stuff. @stuff are delimited arguments in the sense that the leading and trailing nodes should represent open and close delimiters and the arguments are separated by punctuation nodes.

$node = InterpretDelimited($op,@stuff);

Similar to `ApplyDelimited`, this interprets sequence of delimited, punctuated items as being the application of `$op` to those items.

$node = recApply(@ops,$arg);

Given a sequence of operators and an argument, forms the nested application `op(op(...(arg)))`>.

$node = InvisibleTimes;

Creates an invisible times operator.

$boole = isMatchingClose($open,$close);

Checks whether `$open` and `$close` form a 'normal' pair of delimiters, or if either is ".".

$node = Fence(@stuff);

Given a delimited sequence of nodes, starting and ending with open/close delimiters, and with intermediate nodes separated by punctuation or such, attempt to guess what type of thing is represented such as a set, absolute value, interval, and so on.

This would be a good candidate for customization!

$node = NewFormulae(@stuff);

Given a set of formulas, construct a `Formulae` application, if there are more than one, else just return the first.

$node = NewList(@stuff);

Given a set of expressions, construct a `list` application, if there are more than one, else just return the first.

$node = LeftRec($arg1,@more);

Given an expr followed by repeated (op expr), compose the left recursive tree. For example `a + b + c - d` would give `(- (+ a b c) d)`>

MaybeFunction($token);

Note the possible use of `$token` as a function, which may cause incorrect parsing. This is used to generate warning messages.

C.1 Common Modules Documentation

LaTeXML::Common::Config

Configuration logic for LaTeXML

SYNPOSIS

```
use LaTeXML::Common::Config;
my $config = LaTeXML::Common::Config->new(
          profile=>'name',
          timeout=>60,
          ... );
$config->read(\@ARGV);
$config->check;

my $value = $config->get($name);
$config->set($name,$value);
$config->delete($name);
my $bool = $config->exists($name);
my @keys = $config->keys;
my $options_hashref = $config->options;
my $config_clone = $config->clone;
```

Description

Configuration management class for LaTeXML options. * Responsible for defining the options interface and parsing the usual Perl command-line options syntax * Provides the intuitive getters, setters, as well as hash methods for manipulating the option values. * Also supports cloning into new configuration objects.

Methods

`my $config = LaTeXML::Common::Config->new(%options);`

Creates a new configuration object. Note that you should try not to provide your own %options hash but rather create an empty configuration and use $config->read to read in the options.

`$config->read(\@ARGV);`

This is the main method for parsing in LaTeXML options. The input array should either be @ARGV, e.g. when the options were provided from the command line using the classic Getopt::Long syntax, or any other array reference that conforms to that setup.

`$config->check;`

Ensures that the configuration obeys the given profile and performs a set of assignments of meaningful defaults (when needed) and normalizations (for relative paths, etc).

`my $value = $config->get($name);`

Classic getter for the $value of an option $name.

`$config->set($name,$value);`

Classic setter for the $value of an option $name.

$config->delete($name);

Deletes option $name from the configuration.

my $bool = $config->exists($name);

Checks whether the key $name exists in the options hash of the configuration. Similarly to Perl's "exist" for hashes, it returns true even when the option's value is undefined.

my @keys = $config->keys;

Similar to "keys %hash" in Perl. Returns an array of all option names.

my $options_hashref = $config->options;

Returns the actual hash reference that holds all options within the configuration object.

my $config_clone = $config->clone;

Clones $config into a new LaTeXML::Common::Config object, $config_clone.

OPTION SYNOPSIS

latexmlc [options]

```
Options:
--VERSION               show version number.
--help                  shows this help message.
--destination=file      specifies destination file.
--output=file           [obsolete synonym for --destination]
--preload=module        requests loading of an optional module;
                        can be repeated
--preamble=file         loads a tex file containing document
                        frontmatter. MUST include \begin{document}
                        or equivalent
--postamble=file        loads a tex file containing document
                        backmatter. MUST include \end{document}
                        or equivalent
--includestyles         allows latexml to load raw *.sty file;
                        by default it avoids this.
--base=dir              sets the current working directory
--path=dir              adds dir to the paths searched for files,
                        modules, etc;
--log=file              specifies log file (default: STDERR)
--autoflush=count       Automatically restart the daemon after
                        "count" inputs. Good practice for vast
                        batch jobs. (default: 100)
--timeout=secs          Timecap for conversions (default 600)
--expire=secs           Timecap for server inactivity (default 600)
--address=URL           Specify server address (default: localhost)
--port=number           Specify server port (default: 3354)
```

```
--documentid=id         assign an id to the document root.
--quiet                 suppress messages (can repeat)
--verbose               more informative output (can repeat)
--strict                makes latexml less forgiving of errors
--bibtex                processes a BibTeX bibliography.
--xml                   requests xml output (default).
--tex                   requests TeX output after expansion.
--box                   requests box output after expansion
                        and digestion.
--format=name           requests "name" as the output format.
                        Supported: tex,box,xml,html4,html5,xhtml
                        html implies html5
--noparse               suppresses parsing math (default: off)
--parse=name            enables parsing math (default: on)
                        and selects parser framework "name".
                        Supported: RecDescent, no
--profile=name          specify profile as defined in
                        LaTeXML::Common::Config
                        Supported: standard|math|fragment|...
                        (default: standard)
--mode=name             Alias for profile
--cache_key=name        Provides a name for the current option set,
                        to enable daemonized conversions without
                        needing re-initializing
--whatsin=chunk         Defines the provided input chunk,
                        choose from document (default), fragment
                        and formula
--whatsout=chunk        Defines the expected output chunk,
                        choose from document (default), fragment
                        and formula
--post                  requests a followup post-processing
--nopost                forbids followup post-processing
--validate, --novalidate Enables (the default) or disables
                        validation of the source xml.
--omitdoctype           omits the Doctype declaration,
--noomitdoctype         disables the omission (the default)
--numbersections        enables (the default) the inclusion of
                        section numbers in titles, crossrefs.
--nonumbersections      disables the above
--timestamp             provides a timestamp (typically a time and date)
                        to be embedded in the comments
--embed                 requests an embeddable XHTML snippet
                        (requires: --post,--profile=fragment)
                        DEPRECATED: Use --whatsout=fragment
                        TODO: Remove completely
--stylesheet            specifies a stylesheet,
                        to be used by the post-processor.
--css=cssfile           adds a css stylesheet to html/xhtml
                        (can be repeated)
--nodefaultresources    disables processing built-in resources
```

```
--javascript=jsfile     adds a link to a javascript file into
                        html/html5/xhtml (can be repeated)
--icon=iconfile         specify a file to use as a "favicon"
--xsltparameter=name:value passes parameters to the XSLT.
--split                 requests splitting each document
--nosplit               disables the above (default)
--splitat               sets level to split the document
--splitpath=xpath       sets xpath expression to use for
                        splitting (default splits at
                        sections, if splitting is enabled)
--splitnaming=(id|idrelative|label|labelrelative) specifies
                        how to name split files (idrelative).
--scan                  scans documents to extract ids,
                        labels, etc.
                        section titles, etc. (default)
--noscan                disables the above
--crossref              fills in crossreferences (default)
--nocrossref            disables the above
--urlstyle=(server|negotiated|file) format to use for urls
                        (default server).
--navigationtoc=(context|none) generates a table of contents
                        in navigation bar
--index                 requests creating an index (default)
--noindex               disables the above
--splitindex            Splits index into pages per initial.
--nosplitindex          disables the above (default)
--permutedindex         permutes index phrases in the index
--nopermutedindex       disables the above (default)
--bibliography=file     sets a bibliography file
--splitbibliography     splits the bibliography into pages per
                        initial.
--nosplitbibliography   disables the above (default)
--prescan               carries out only the split (if
                        enabled) and scan, storing
                        cross-referencing data in dbfile
                        (default is complete processing)
--dbfile=dbfile         sets file to store crossreferences
--sitedirectory=dir     sets the base directory of the site
--sourcedirectory=dir   sets the base directory of the
                        original TeX source
--source=input          as an alternative to passing the input as
                        the last argument, after the option set
                        you can also specify it as the value here.
                        useful for predictable API calls
--mathimages            converts math to images
                        (default for html4 format)
--nomathimages          disables the above
--mathimagemagnification=mag specifies magnification factor
--presentationmathml    converts math to Presentation MathML
                        (default for xhtml & html5 formats)
```

```
--pmml                  alias for --presentationmathml
--nopresentationmathml  disables the above
--linelength=n          formats presentation mathml to a
                        linelength max of n characters
--contentmathml         converts math to Content MathML
--nocontentmathml       disables the above (default)
--cmml                  alias for --contentmathml
--openmath              converts math to OpenMath
--noopenmath            disables the above (default)
--om                    alias for --openmath
--keepXMath             preserves the intermediate XMath
                        representation (default is to remove)
--mathtex               adds TeX annotation to parallel markup
--nomathtex             disables the above (default)
--unicodemath           adds UnicodeMath annotation to parallel markup
--nounicodemath         disables the above (default)
--parallelmath          use parallel math annotations (default)
--noparallelmath        disable parallel math annotations
--plane1                use plane-1 unicode for symbols
                        (default, if needed)
--noplane1              do not use plane-1 unicode
--graphicimages         converts graphics to images (default)
--nographicimages       disables the above
--graphicsmap=type.type specifies a graphics file mapping
--pictureimages         converts picture environments to
                        images (default)
--nopictureimages       disables the above
--svg                   converts picture environments to SVG
--nosvg                 disables the above (default)
--nocomments            omit comments from the output
--inputencoding=enc     specify the input encoding.
--debug=package         enables debugging output for the named
                        package
```

If you want to provide a TeX snippet directly on input, rather than supply a filename, use the `literal:` protocol to prefix your snippet.

Options & Arguments

General Options

`--verbose`

Increases the verbosity of output during processing, used twice is pretty chatty. Can be useful for getting more details when errors occur.

`--quiet`

Reduces the verbosity of output during processing, used twice is pretty silent.

`--VERSION`

Shows the version number of the LaTeXML package..

--debug=*package*

Enables debugging output for the named package. The package is given without the leading LaTeXML::.

--base=*dir*

Specifies the base working directory for the conversion server. Useful when converting sets of documents that use relative paths.

--log=*file*

Specifies the log file; be default any conversion messages are printed to STDERR.

--help

Shows this help message.

Source Options

--destination=*file*

Specifies the destination file; by default the XML is written to STDOUT.

--preload=*module*

Requests the loading of an optional module or package. This may be useful if the TeX code does not specifically require the module (eg. through input or usepackage). For example, use `--preload=LaTeX.pool` to force LaTeX mode.

--preamble=*file*

Requests the loading of a tex file with document frontmatter, to be read in before the converted document, but after all --preload entries.

Note that the given file MUST contain \begin{document} or an equivalent environment start, when processing LaTeX documents.

If the file does not contain content to appear in the final document, but only macro definitions and setting of internal counters, it is more appropriate to use --preload instead.

--postamble=*file*

Requests the loading of a tex file with document backmatter, to be read in after the converted document.

Note that the given file MUST contain \end{document} or an equivalent environment end, when processing LaTeX documents.

--sourcedirectory=*source*

Specifies the directory where the original latex source is located. Unless LaTeXML is run from that directory, or it can be determined from the xml filename, it may be necessary to specify this option in order to find graphics and style files.

--path=*dir*

Add *dir* to the search paths used when searching for files, modules, style files, etc; somewhat like TEXINPUTS. This option can be repeated.

--validate, --novalidate

Enables (or disables) the validation of the source XML document (the default).

--bibtex

Forces latexml to treat the file as a BibTeX bibliography. Note that the timing is slightly different than the usual case with BibTeX and LaTeX. In the latter case, BibTeX simply selects and formats a subset of the bibliographic entries; the actual TeX expansion is carried out when the result is included in a LaTeX document. In contrast, latexml processes and expands the entire bibliography; the selection of entries is done during post-processing. This also means that any packages that define macros used in the bibliography must be specified using the `--preload` option.

--inputencoding=*encoding*

Specify the input encoding, eg. `--inputencoding=iso-8859-1`. The encoding must be one known to Perl's Encode package. Note that this only enables the translation of the input bytes to UTF-8 used internally by LaTeXML, but does not affect catcodes. In such cases, you should be using the inputenc package. Note also that this does not affect the output encoding, which is always UTF-8.

TeX Conversion Options

--includestyles

This optional allows processing of style files (files with extensions `sty`, `cls`, `clo`, `cnf`). By default, these files are ignored unless a latexml implementation of them is found (with an extension of `ltxml`).

These style files generally fall into two classes: Those that merely affect document style are ignorable in the XML. Others define new markup and document structure, often using deeper LaTeX macros to achieve their ends. Although the omission will lead to other errors (missing macro definitions), it is unlikely that processing the TeX code in the style file will lead to a correct document.

--timeout=*secs*

Set time cap for conversion jobs, in seconds. Any job failing to convert in the time range would return with a Fatal error of timing out. Default value is 600, set to 0 to disable.

--nocomments

Normally latexml preserves comments from the source file, and adds a comment every 25 lines as an aid in tracking the source. The option --nocomments discards such comments.

--documentid=*id*

Assigns an ID to the root element of the XML document. This ID is generally inherited as the prefix of ID's on all other elements within the document. This is useful when constructing a site of multiple documents so that all nodes have unique IDs.

--strict

Specifies a strict processing mode. By default, undefined control sequences and invalid document constructs (that violate the DTD) give warning messages, but attempt to continue processing. Using `--strict` makes them generate fatal errors.

--post

Request post-processing, auto-enabled by any requested post-processor. Disabled by default. If post-processing is enabled, the graphics and cross-referencing processors are on by default.

Format Options

--format=(html|html5|html4|xhtml|xml|epub)

Specifies the output format for post processing. By default, it will be guessed from the file extension of the destination (if given), with html implying `html5`, xhtml implying `xhtml` and the default being `xml`, which you probably don't want.

The `html5` format converts the material to html5 form with mathematics as MathML; `html5` supports SVG. `html4` format converts the material to the earlier html form, version 4, and the mathematics to png images. `xhtml` format converts to xhtml and uses presentation MathML (after attempting to parse the mathematics) for representing the math. `html5` similarly converts math to presentation MathML. In these cases, any graphics will be converted to web-friendly formats and/or copied to the destination directory. If you simply specify `html`, it will treat that as `html5`.

For the default, `xml`, the output is left in LaTeXML's internal xml, although the math can be converted by enabling one of the math postprocessors, such as --pmml to obtain presentation MathML. For html, html5 and xhtml, a default stylesheet is provided, but see the `--stylesheet` option.

--xml

Requests XML output; this is the default. DEPRECATED: use --format=xml instead

--tex

Requests TeX output for debugging purposes; processing is only carried out through expansion and digestion. This may not be quite valid TeX, since Unicode may be introduced.

--box

Requests Box output for debugging purposes; processing is carried out through expansion and digestions, and the result is printed.

--profile

Variety of shorthand profiles. Note that the profiles come with a variety of preset options. You can examine any of them in their `resources/Profiles/name.opt` file.

Example: `latexmlc --profile=math 'literal:1+2=3'`

--omitdoctype, --noomitdoctype

Omits (or includes) the document type declaration. The default is to include it if the document model was based on a DTD.

--numbersections, --nonumbersections

Includes (default), or disables the inclusion of section, equation, etc, numbers in the formatted document and crossreference links.

--stylesheet=*xslfile*

Requests the XSL transformation of the document using the given xslfile as stylesheet. If the stylesheet is omitted, a 'standard' one appropriate for the format (html4, html5 or xhtml) will be used.

--css=*cssfile*

Adds *cssfile* as a css stylesheet to be used in the transformed html/html5/xhtml. Multiple stylesheets can be used; they are included in the html in the order given, following the default `ltx-LaTeXML.css` (unless `--nodefaultcss`). The stylesheet is copied to the destination directory, unless it is an absolute url.

Some stylesheets included in the distribution are --css=navbar-left Puts a navigation bar on the left. (default omits navbar) --css=navbar-right Puts a navigation bar on the left. --css=theme-blue A blue coloring theme for headings. --css=amsart A style suitable for journal articles.

--javascript=*jsfile*

Includes a link to the javascript file *jsfile*, to be used in the transformed html/html5/xhtml. Multiple javascript files can be included; they are linked in the html in the order given. The javascript file is copied to the destination directory, unless it is an absolute url.

--icon=*iconfile*

Copies *iconfile* to the destination directory and sets up the linkage in the transformed html/html5/xhtml to use that as the "favicon".

--nodefaultresources

Disables the copying and inclusion of resources added by the binding files; This includes CSS, javascript or other files. This does not affect resources explicitly requested by the `--css` or `--javascript` options.

--timestamp=*timestamp*

Provides a timestamp (typically a time and date) to be embedded in the comments by the stock XSLT stylesheets. If you don't supply a timestamp, the current time and date will be used. (You can use `--timestamp=0` to omit the timestamp).

--xsltparameter=*name*:*value*

Passes parameters to the XSLT stylesheet. See the manual or the stylesheet itself for available parameters.

Site & Crossreferencing Options

--split, --nosplit

Enables or disables (default) the splitting of documents into multiple 'pages'. If enabled, the the document will be split into sections, bibliography, index and appendices (if any) by default, unless `--splitpath` is specified.

--splitat=*unit*

Specifies what level of the document to split at. Should be one of `chapter`, `section` (the default), `subsection` or `subsubsection`. For more control, see `--splitpath`.

--splitpath=*xpath*

Specifies an XPath expression to select nodes that will generate separate pages. The default splitpath is //ltx:section | //ltx:bibliography | //ltx:appendix | //ltx:index

Specifying

```
--splitpath="//ltx:section | //ltx:subsection
       | //ltx:bibliography | //ltx:appendix | //ltx:index"
```

would split the document at subsections as well as sections.

--splitnaming=(id|idrelative|label|labelrelative)

Specifies how to name the files for subdocuments created by splitting. The values `id` and `label` simply use the id or label of the subdocument's root node for it's filename. `idrelative` and `labelrelative` use the portion of the id or label that follows the parent document's id or label. Furthermore, to impose structure and uniqueness, if a split document has children that are also split, that document (and it's children) will be in a separate subdirectory with the name index.

--scan, --noscan

Enables (default) or disables the scanning of documents for ids, labels, references, indexmarks, etc, for use in filling in refs, cites, index and so on. It may be useful to disable when generating documents not based on the LaTeXML doctype.

--crossref, --nocrossref

Enables (default) or disables the filling in of references, hrefs, etc based on a previous scan (either from `--scan`, or `--dbfile`) It may be useful to disable when generating documents not based on the LaTeXML doctype.

--urlstyle=(server|negotiated|file)

This option determines the way that URLs within the documents are formatted, depending on the way they are intended to be served. The default, `server`, eliminates unnecessary trailing `index.html`. With `negotiated`, the trailing file extension (typically `html` or `xhtml`) are eliminated. The scheme `file` preserves complete (but relative) urls so that the site can be browsed as files without any server.

--navigationtoc=(context|none)

Generates a table of contents in the navigation bar; default is `none`. The 'context' style of TOC, is somewhat verbose and reveals more detail near the current page; it is most suitable for navigation bars placed on the left or right. Other styles of TOC should be developed and added here, such as a short form.

--index, --noindex

Enables (default) or disables the generation of an index from indexmarks embedded within the document. Enabling this has no effect unless there is an index element in the document (generated by \printindex).

--splitindex, --nosplitindex

Enables or disables (default) the splitting of generated indexes into separate pages per initial letter.

--bibliography=*pathname*

Specifies a bibliography generated from a BibTeX file to be used to fill in a bibliography element. Hand-written bibliographies placed in a `thebibliography` environment do not need this. The option has no effect unless there is an bibliography element in the document (generated by \bibliography).

Note that this option provides the bibliography to be used to fill in the bibliography element (generated by `\bibliography`); latexmlpost does not (currently) directly process and format such a bibliography.

--splitbibliography, --nosplitbibliography

Enables or disables (default) the splitting of generated bibliographies into separate pages per initial letter.

--prescan

By default `latexmlpost` processes a single document into one (or more; see `--split`) destination files in a single pass. When generating a complicated site consisting of several documents it may be advantageous to first scan through the documents to extract and store (in `dbfile`) cross-referencing data (such as ids, titles, urls, and so on). A later pass then has complete information allowing all documents to reference each other, and also constructs an index and bibliography that reflects the entire document set. The same effect (though less efficient) can be achieved by running `latexmlpost` twice, provided a `dbfile` is specified.

--dbfile=*file*

Specifies a filename to use for the crossreferencing data when using two-pass processing. This file may reside in the intermediate destination directory.

--sitedirectory=*dir*

Specifies the base directory of the overall web site. Pathnames in the database are stored in a form relative to this directory to make it more portable.

--embed

TODO: Deprecated, use --whatsout=fragment Requests an embeddable XHTML div (requires: --post --format=xhtml), respectively the top division of the document's body. Caveat: This experimental mode is enabled only for fragment profile and post-processed documents (to XHTML).

Math Options These options specify how math should be converted into other formats. Multiple formats can be requested; how they will be combined depends on the format and other options.

--noparse

Suppresses parsing math (default: parsing is on)

--parse=name

Enables parsing math (default: parsing is on) and selects parser framework "name". Supported: RecDescent, no Tip: --parse=no is equivalent to --noparse

--mathimages, --nomathimages

Requests or disables the conversion of math to images (png by default). Conversion is the default for html4 format.

--mathsvg, --nomathsvg

Requests or disables the conversion of math to svg images.

--mathimagemagnification=*factor*

Specifies the magnification used for math images (both png and svg), if they are made. Default is 1.75.

--presentationmathml, --nopresentationmathml

Requests or disables conversion of math to Presentation MathML. Conversion is the default for xhtml and html5 formats.

--linelength=*number*

(Experimental) Line-breaks the generated Presentation MathML so that it is no longer than *number* 'characters'.

--plane1

Converts the content of Presentation MathML token elements to the appropriate Unicode Plane-1 codepoints according to the selected font, when applicable (the default).

--hackplane1

Converts the content of Presentation MathML token elements to the appropriate Unicode Plane-1 codepoints according to the selected font, but only for the mathvariants double-struck, fraktur and script. This gives support for current (as of August 2009) versions of Firefox and MathPlayer, provided a sufficient set of fonts is available (eg. STIX).

--contentmathml, --nocontentmathml

Requests or disables conversion of math to Content MathML. Conversion is disabled by default. **Note** that this conversion is only partially implemented.

--openmath

Requests or disables conversion of math to OpenMath. Conversion is disabled by default. **Note** that this conversion is only partially implemented.

--keepXMath, --xmath

By default, when any of the MathML or OpenMath conversions are used, the intermediate math representation will be removed; this option preserves it; it will be used as secondary parallel markup, when it follows the options for other math representations.

Graphics Options

--graphicimages, --nographicimages

Enables (default) or disables the conversion of graphics to web-appropriate format (png).

--graphicsmap=*sourcetype.desttype*

Specifies a mapping of graphics file types. Typically, graphics elements specify a graphics file that will be converted to a more appropriate file target format; for example, postscript files used for graphics with LaTeX will be converted to png format for use on the web. As with LaTeX, when a graphics file is specified

without a file type, the system will search for the most appropriate target type file.

When this option is used, it overrides *and replaces* the defaults and provides a mapping of *sourcetype* to *desttype*. The option can be repeated to provide several mappings, with the earlier formats preferred. If the *desttype* is omitted, it specifies copying files of type *sourcetype*, unchanged.

The default settings is equivalent to having supplied the options: svg png gif jpg jpeg eps.png ps.png ai.png pdf.png

The first formats are preferred and used unchanged, while the latter ones are converted to png.

`--pictureimages`, `--nopictureimages`

Enables (default) or disables the conversion of picture environments and pstricks material into images.

`--svg`, `--nosvg`

Enables or disables (default) the conversion of picture environments and pstricks material to SVG.

Daemon, Server and Client Options Options used only for daemonized conversions, e.g. talking to a remote server via latexmlc, or local processing via the `LaTeXML::Plugin::latexmls` plugin.

For reliable communication and a stable conversion experience, invoke latexmls only through the latexmlc client (you need to set --expire to a positive value, in order to request auto-spawning of a dedicated conversion server).

`--autoflush=`*`count`*

Automatically restart the daemon after converting "count" inputs. Good practice for vast batch jobs. (default: 100)

`--expire=`*`secs`*

Set an inactivity timeout value in seconds. If the server process is not given any input for the specified duration, it will automatically terminate. The default value is 600 seconds, set to 0 to never expire, -1 to entirely opt out of using an independent server.

`--address=`*`URL`*

Specify server address (default: localhost)

`--port=`*`number`*

Specify server port (default: 3334 for math, 3344 for fragment and 3354 for standard)

`LaTeXML::Common::Object`

Abstract base class for most LaTeXML objects.

Description

LaTeXML::Common::Object serves as an abstract base class for all other objects (both the data objects and control objects). It provides for common methods for stringification and comparison operations to simplify coding and to beautify error reporting.

Generic functions

$string = Stringify($object);

Returns a string identifying $object, for debugging. Works on any values and objects, but invokes the stringify method on blessed objects. More informative than the default perl conversion to a string.

$string = ToString($object);

Converts $object to string attempting, when possible, to generate straight text without TeX markup. This is most useful for converting Tokens or Boxes to document content or attribute values, or values to be used for pathnames, keywords, etc. Generally, however, it is not possible to convert Whatsits generated by Constructors into clean strings, without TeX markup. Works on any values and objects, but invokes the toString method on blessed objects.

$boolean = Equals($x,$y);

Compares the two objects for equality. Works on any values and objects, but invokes the equals method on blessed objects, which does a deep comparison of the two objects.

$tokens = Revert($object);

Returns a Tokens list containing the TeX that would create $object. Note that this is not necessarily the original TeX code; expansions or other substitutions may have taken place.

Methods

$string = $object->stringify;

Returns a readable representation of $object, useful for debugging.

$string = $object->toString;

Returns the string content of $object; most useful for extracting a clean, usable, Unicode string from tokens or boxes that might representing a filename or such. To the extent possible, this should provide a string that can be used as XML content, or attribute values, or for filenames or whatever. However, control sequences defined as Constructors may leave TeX code in the value.

$boole = $object->equals($other);

Returns whether $object and $other are equal. Should perform a deep comparison, but the default implementation just compares for object identity.

$boole = $object->isaToken;

Returns whether $object is an LaTeXML::Core::Token.

$boole = $object->isaBox;

Returns whether $object is an LaTeXML::Core::Box.

$boole = $object->isaDefinition;

Returns whether $object is an LaTeXML::Core::Definition.

$digested = $object->beDigested;

Does whatever is needed to digest the object, and return the digested representation. Tokens would be digested into boxes; Some objects, such as numbers can just return themselves.

$object->beAbsorbed($document);

Do whatever is needed to absorb the $object into the $document, typically by invoking appropriate methods on the $document.

LaTeXML::Common::Color

Abstract class representating colors using various color models; extends LaTeXML::Common::Object.

Exported functions

$color = Color($model,@components);

Creates a Color object using the given color model, and with the given components. The core color models are rgb, hsv, cmy, cmyk and gray. The components of colors using core color models are between 0 and 1 (inclusive)

Black, White

Constant color objects representing black and white, respectively.

Methods

$model = $color->model;

Return the name of the color model.

@components = $color->components;

Return the components of the color.

$other = $color->convert($tomodel);

Converts the color to another color model.

$string = $color->toString;

Returns a printed representation of the color.

`$hex = $color->toHex;`

Returns a string representing the color as RGB in hexadecimal (6 digits).

`$other = $color->toCore();`

Converts the color to one of the core colors.

`$complement = $color->complement();`

Returns the complement color (works for colors in `rgb`, `cmy` and `gray` color models).

`$new = $color->mix($other,$fraction);`

Returns a new color which results from mixing a `$fraction` of `$color` with `(1-$fraction)` of color `$other`.

`$new = $color->add($other);`

Returns a new color made by adding the components of the two colors.

`$new = $color->scale($m);`

Returns a new color made by mulitiplying the components by `$n`.

`$new = $color->multiply(@m);`

Returns a new color made by mulitiplying the components by the corresponding component from `@n`.

See also

Supported color models: `LaTeXML::Common::Color::rgb`, `LaTeXML::Common::Color::hsb`, `LaTeXML::Common::Color::cmy`, `LaTeXML::Common::Color::cmyk`, `LaTeXML::Common::Color::gray` and `LaTeXML::Common::Color::Derived`.

LaTeXML::Common::Color::rgb

Represents colors in the rgb color model: red, green and blue in [0..1]; extends `LaTeXML::Common::Color`.

LaTeXML::Common::Color::hsb

Represents colors in the hsb color model: hue, saturation, brightness in [0..1]; extends `LaTeXML::Common::Color`.

LaTeXML::Common::Color::cmy

Represents colors in the cmy color model: cyan, magenta and yellow [0..1]; extends `LaTeXML::Common::Color`.

LaTeXML::Common::Color::cmyk

Represents colors in the cmyk color model: cyan, magenta, yellow and black in [0..1]; extends LaTeXML::Common::Color.

LaTeXML::Common::Color::gray

Represents colors in the gray color model: gray value in [0..1]; extends LaTeXML::Common::Color.

LaTeXML::Common::Color::Derived

Represents colors in derived color models

Synopsis

LaTeXML::Common::Color::Derived represents colors in derived color models. These are used to support various color models defined and definable via the xcolor package, such as colors where the components are in different ranges. It extends LaTeXML::Common::Color.

LaTeXML::Common::Number

Representation of numbers; extends LaTeXML::Common::Object.

Exported functions

$number = Number($num);

Creates a Number object representing $num.

Methods

@tokens = $object->unlist;

Return a list of the tokens making up this $object.

$string = $object->toString;

Return a string representing $object.

$string = $object->ptValue;

Return a value representing $object without the measurement unit (pt) with limited decimal places.

$string = $object->pxValue;

Return an integer value representing $object in pixels. Uses the state variable DPI (dots per inch).

$n = $object->valueOf;

Return the value in scaled points (ignoring shrink and stretch, if any).

$n = $object->smaller($other);

Return $object or $other, whichever is smaller

$n = $object->larger($other);

Return $object or $other, whichever is larger

$n = $object->absolute;

Return an object representing the absolute value of the $object.

$n = $object->sign;

Return an integer: -1 for negatives, 0 for 0 and 1 for positives

$n = $object->negate;

Return an object representing the negative of the $object.

$n = $object->add($other);

Return an object representing the sum of $object and $other

$n = $object->subtract($other);

Return an object representing the difference between $object and $other

$n = $object->multiply($n);

Return an object representing the product of $object and $n (a regular number).

$n = $object->divide($n);

Return an object representing the (truncating) division of $object by $n (a regular number).

$n = $object->divideround($n);

Return an object representing the (rounding) division of $object by $n (a regular number).

LaTeXML::Common::Float

Representation of floating point numbers; extends LaTeXML::Common::Number.

Exported functions

$number = Float($num);

Creates a floating point object representing $num; This is not part of TeX, but useful.

LaTeXML::Common::Dimension

Representation of dimensions; extends LaTeXML::Common::Number.

Exported functions

`$dimension = Dimension($spec);`

Creates a Dimension object. `$spec` can be a string with a floating point number and units (with any of the usual TeX recognized units, except mu), or just a number standing for scaled points (sp).

LaTeXML::Common::Glue

Representation of glue, skips, stretchy dimensions; extends `LaTeXML::Common::Dimension`.

Exported functions

`$glue = Glue($spec);`

`$glue = Glue($sp,$plus,$pfill,$minus,$mfill);`

Creates a Glue object. `$spec` can be a string in the form that TeX recognizes (number units optional plus and minus parts). Alternatively, the dimension, plus and minus parts can be given separately as scaled points (fixpoint), while `$pfill` and `$mfill` are 0 (when the `$plus` or `$minus` part is in scaled-points) or 1,2,3 for fil, fill or filll, respectively.

LaTeXML::Common::Font

Representation of fonts

Description

`LaTeXML::Common::Font` represent fonts in LaTeXML. It extends `LaTeXML::Common::Object`.

This module defines Font objects. I'm not completely happy with the arrangement, or maybe just the use of it, so I'm not going to document extensively at this point.

The attributes are

```
family : serif, sansserif, typewriter, caligraphic,
         fraktur, script
series : medium, bold
shape  : upright, italic, slanted, smallcaps
size   : TINY, Tiny, tiny, SMALL, Small, small,
         normal, Normal, large, Large, LARGE,
         huge, Huge, HUGE, gigantic, Gigantic, GIGANTIC
color  : any named color, default is black
```

They are usually merged against the current font, attempting to mimic the, sometimes counter-intuitive, way that TeX does it, particularly for math

Methods

$font->specialize($string);

In math mode, LaTeXML::Common::Font supports computing a font reflecting how the specific $string would be printed when $font is active; This (attempts to) handle the curious ways that lower case greek often doesn't get a different font. In particular, it recognizes the following classes of strings: single latin letter, single uppercase greek character, single lowercase greek character, digits, and others.

LaTeXML::Common::Model

Represents the Document Model

Description

LaTeXML::Common::Model encapsulates information about the document model to be used in converting a digested document into XML by the LaTeXML::Core::Document. This information is based on the document schema (eg, DTD, RelaxNG), but is also modified by package modules; thus the model may not be complete until digestion is completed.

The kinds of information that is relevant is not only the content model (what each element can contain contain), but also SGML-like information such as whether an element can be implicitly opened or closed, if needed to insert a new element into the document.

Currently, only an approximation to the schema is understood and used. For example, we only record that certain elements can appear within another; we don't preserve any information about required order or number of instances.

It extends LaTeXML::Common::Object.

Model Creation

$model = LaTeXML::Common::Model->new(%options);

Creates a new model. The only useful option is permissive=>1 which ignores any DTD and allows the document to be built without following any particular content model.

Document Type

$model->setDocType($rootname,$publicid,$systemid,%namespaces);

Declares the expected rootelement, the public and system ID's of the document type to be used in the final document. The hash %namespaces specifies the namespace prefixes that are expected to be found in the DTD, along with the associated namespace URI. These prefixes may be different from the prefixes used in implementation code (eg. in ltxml files; see RegisterNamespace). The generated document will use the namespaces and prefixes defined here.

Namespaces Note that there are *two* namespace mappings between namespace URIs and prefixes that are relevant to LaTeXML. The 'code' mapping is the one used in code implementing packages, and in particular, constructors defined within those packages. The prefix `ltx` is used consistently to refer to LaTeXML's own namespace (`http://dlmf.nist.gov/LaTeXML`).

The other mapping, the 'document' mapping, is used in the created document; this may be different from the 'code' mapping in order to accommodate DTDs, for example, or for use by other applications that expect a rigid namespace mapping.

`$model->registerNamespace($prefix,$namespace_url);`

Register `$prefix` to stand for the namespace `$namespace_url`. This prefix can then be used to create nodes in constructors and Document methods. It will also be recognized in XPath expressions.

`$model->getNamespacePrefix($namespace,$forattribute,$probe);`

Return the prefix to use for the given `$namespace`. If `$forattribute` is nonzero, then it looks up the prefix as appropriate for attributes. If `$probe` is nonzero, it only probes for the prefix, without creating a missing entry.

`$model->getNamespace($prefix,$probe);`

Return the namespace url for the given `$prefix`.

Model queries

`$boole = $model->canContain($tag,$childtag);`

Returns whether an element with qualified name `$tag` can contain an element with qualified name `$childtag`. The tag names #PCDATA, #Document, #Comment and #ProcessingInstruction are specially recognized.

`$boole = $model->canHaveAttribute($tag,$attribute);`

Returns whether an element with qualified name `$tag` is allowed to have an attribute with the given name.

See also

LaTeXML::Common::Model::DTD, LaTeXML::Common::Model::RelaxNG.

LaTeXML::Common::Model::DTD

Represents DTD document models; extends LaTeXML::Common::Model.

LaTeXML::Common::Model::RelaxNG

Represents RelaxNG document models; extends LaTeXML::Common::Model.

LaTeXML::Common::XML

XML utilities

Description

This module provides utilities for accessing XML, along with some patches to XML::LibXML.

element_nodes($node)

Returns a list of the element children of `$node`.

text_in_node($node)

Returns the string combining the text nodes within `$node`.

isTeXtNode($node)

Checks whether `$node` is a text node.

isElementNode($node)

Checks whether `$node` is a element node.

isChild($child,$parent)

Checks whether `$child` is a child of `$parent`.

isDecscendant($child,$parent)

Checks whether `$child` is a descendant of `$parent`.

isDecscendantOrSelf($child,$parent)

Checks whether `$child` is a descendant of, or the same as, `$parent`.

new_node($nsURI,$tag,$children,%attributes)

Creates a new element node with tag `$tag` (in the namespace `$nsURI`), with the children in the array ref `$children` (if any) and assigning the given attributes.

append_nodes($node,@children)

Appends the given children to `$node`.

clear_node($node)

Removes all element and text children from `$node`.

maybe_clone($node)

Clones `$node` if it has a parent, otherwise returns it.

copy_attributes($to,$from)

Copy all attributes from `$from` to `$to`.

rename_attribute($node,$from,$to)

Rename the attribute `$from` to `$to` on the node `$node`.

remove_attr($node,@attr)

Remove the given attributes from `$node`.

get_attr($node,@attr)

Returns the list of values for the given attributes on `$node`

initialize_catalogs()

Initialize XML::LibXML to recognize the catalogs given in LaTeXML.catalogs.

set_RDFa_prefixes($document,$map)

This method scans the document's RDFa attributes, extracting the prefixes used. These prefixes are then filtered through a `$map` of known RDFa prefixes and the ones allowed are declared globally for the document via the `prefix` attribute of its root element.

LaTeXML::Common::Error

Error and Progress Reporting and Logging support.

Description

`LaTeXML::Common::Error` does some simple stack analysis to generate more informative, readable, error messages for LaTeXML. Its routines are used by the error reporting methods from `LaTeXML::Global`, namely `Warn`, `Error` and `Fatal`.

The general idea is that a minimal amount should be printed to STDERR (possibly with colors, spinners, etc if it is a terminal), and more complete information is printed to a log file. Neither of these are enabled, by default; see below.

SetVerbosity($verbosity);

Controls the verbosity of output to the terminal; default is 0, higher gives more information, lower gives less. A verbosity less than 0 inhibits all output to STDERR.

UseSTDERR(); ... UseSTDERR(undef);

`UseSTDERR();` Enables and initializes STDERR to accept messages. If this is not called, there will be no output to STDERR. `UseSTDERR(undef);` disables STDERR from further messages.

UseLog($path, $append); ... UseLog(undef);

`UseLog($path, $append);` opens a log file on the given path. If `$append` is true, this file will be appended to, otherwise, it will be created initially empty. If this is not called, there will be no log file. `UseLog(undef);` disables and closes the log file.

Error Reporting The Error reporting functions all take a similar set of arguments, the differences are in the implied severity of the situation, and in the amount of detail that will be reported.

The `$category` is a string naming a broad category of errors, such as "undefined". The set is open-ended, but see the manual for a list of recognized categories. `$object` is the object whose presence or lack caused the problem.

`$where` indicates where the problem occurred; passs in the `$gullet` or `$stomach` if the problem occurred during expansion or digestion; pass in a document node if it occurred there. A string will be used as is; if an undefined value is used, the error handler will try to guess.

The `$message` should be a somewhat concise, but readable, explanation of the problem, but ought to not refer to the document or any "incident specific" information, so as to support indexing in build systems. `@details` provides additional lines of information that may be indident specific.

`Fatal($category,$object,$where,$message,@details);`

Signals an fatal error, printing `$message` along with some context. In verbose mode a stack trace is printed.

`Error($category,$object,$where,$message,@details);`

Signals an error, printing `$message` along with some context. If in strict mode, this is the same as Fatal(). Otherwise, it attempts to continue processing..

`Warn($category,$object,$where,$message,@details);`

Prints a warning message along with a short indicator of the input context, unless verbosity is quiet.

`Info($category,$object,$where,$message,@details);`

Prints an informational message along with a short indicator of the input context, unless verbosity is quiet.

Progress Reporting

`Note($message);`

General status message, printed whenever verbosity at or above 0, to both STDERR and the Log file (when enabled).

`NoteLog($message);`

Prints a status message to the Log file (when enabled).

`NoteSTDERR($message);`

Prints a status message to the terminal (STDERR) (when enabled).

`ProgressSpinup($stage);`

Begin a processing stage, which will be ended with `ProgressSpindown($stage);` This prints a message to the log such as "(stage... runtime)", where runtime is

the time required. In conjunction with `ProgressStep()`, creates a progress spinner on STDERR.

`ProgressSpinup($stage);`

End a processing stage bugin with `ProgressSpindown($stage);`.

`ProgressStep();`

Steps a progress spinner on STDERR.

Debugging Debugging statements may be embedded throughout the program. These are associated with a feature keyword. A given feature is enabled using the command-line option `--debug=feature`.

`Debug($message) if $LaTeXML::DEBUG{$feature}`

Prints `$message` if debugging has been enabled for the given feature.

`DebuggableFeature($feature,$description)`

Declare that `$feature` is a known debuggable feature, and give a description of it.

`CheckDebuggable()`

A untility to check and report if all requested debugging features actually have debugging messages declared.

Internal Functions No user serviceable parts inside. These symbols are not exported.

`$string = LaTeXML::Common::Error::generateMessage($typ,$msg,$lng,@more);`

Constructs an error or warning message based on the current stack and the current location in the document. `$typ` is a short string characterizing the type of message, such as "Error". `$msg` is the error message itself. If `$lng` is true, will generate a more verbose message; this also uses the VERBOSITY set in the `$STATE`. Longer messages will show a trace of the objects invoked on the stack, `@more` are additional strings to include in the message.

`$string = LaTeXML::Common::Error::stacktrace;`

Return a formatted string showing a trace of the stackframes up until this function was invoked.

`@objects = LaTeXML::Common::Error::objectStack;`

Return a list of objects invoked on the stack. This procedure only considers those stackframes which involve methods, and the objects are those (unique) objects that the method was called on.

C.2 Core Modules Documentation

LaTeXML::Core::State

Stores the current state of processing.

Description

A LaTeXML::Core::State object stores the current state of processing. It recording catcodes, variables values, definitions and so forth, as well as mimicking TeX's scoping rules.

Access to State and Processing

$STATE->getStomach;

Returns the current Stomach used for digestion.

$STATE->getModel;

Returns the current Model representing the document model.

Scoping The assignment methods, described below, generally take a $scope argument, which determines how the assignment is made. The allowed values and their implications are:

```
global   : global assignment.
local    : local assignment, within the current grouping.
undef    : global if \global preceded, else local (default)
<name>   : stores the assignment in a 'scope' which
           can be loaded later.
```

If no scoping is specified, then the assignment will be global if a preceding \global has set the global flag, otherwise the value will be assigned within the current grouping.

$STATE->pushFrame;

Starts a new level of grouping. Note that this is lower level than \bgroup; See LaTeXML::Core::Stomach.

$STATE->popFrame;

Ends the current level of grouping. Note that this is lower level than \egroup; See LaTeXML::Core::Stomach.

$STATE->setPrefix($prefix);

Sets a prefix (eg. global for \global, etc) for the next operation, if applicable.

$STATE->clearPrefixes;

Clears any prefixes.

Values

$value = $STATE->lookupValue($name);

Lookup the current value associated with the the string $name.

$STATE->assignValue($name,$value,$scope);

Assign $value to be associated with the the string $name, according to the given scoping rule.

Values are also used to specify most configuration parameters (which can therefore also be scoped). The recognized configuration parameters are:

```
STRICT            : whether errors (eg. undefined macros)
                    are fatal.
INCLUDE_COMMENTS  : whether to preserve comments in the
                    source, and to add occasional line
                    number comments. (Default true).
PRESERVE_NEWLINES : whether newlines in the source should
                    be preserved (not 100% TeX-like).
                    By default this is true.
SEARCHPATHS       : a list of directories to search for
                    sources, implementations, etc.
```

$STATE->pushValue($name,$value);

This is like ->assign, but pushes a value onto the end of the stored value, which should be a LIST reference. Scoping is not handled here (yet?), it simply pushes the value onto the last binding of $name.

$boole = $STATE->isValuebound($type,$name,$frame);

Returns whether the value $name is bound. If $frame is given, check whether it is bound in the $frame-th frame, with 0 being the top frame.

Category Codes

$value = $STATE->lookupCatcode($char);

Lookup the current catcode associated with the the character $char.

$STATE->assignCatcode($char,$catcode,$scope);

Set $char to have the given $catcode, with the assignment made according to the given scoping rule.

This method is also used to specify whether a given character is active in math mode, by using math:$char for the character, and using a value of 1 to specify that it is active.

Definitions

`$defn = $STATE->lookupMeaning($token);`

Get the "meaning" currently associated with `$token`, either the definition (if it is a control sequence or active character) or the token itself if it shouldn't be executable. (See `LaTeXML::Core::Definition`)

`$STATE->assignMeaning($token,$defn,$scope);`

Set the definition associated with `$token` to `$defn`. If `$globally` is true, it makes this the global definition rather than bound within the current group. (See `LaTeXML::Core::Definition`, and `LaTeXML::Package`)

`$STATE->installDefinition($definition, $scope);`

Install the definition into the current stack frame under its normal control sequence.

Named Scopes Named scopes can be used to set variables or redefine control sequences within a scope other than the standard TeX grouping. For example, the LaTeX implementation will automatically activate any definitions that were defined with a named scope of, say "section:4", during the portion of the document that has the section counter equal to 4. Similarly, a scope named "label:foo" will be activated in portions of the document where `\label{foo}` is in effect.

`$STATE->activateScope($scope);`

Installs any definitions that were associated with the named `$scope`. Note that these are placed in the current grouping frame and will disappear when that grouping ends.

`$STATE->deactivateScope($scope);`

Removes any definitions that were associated with the named `$scope`. Normally not needed, since a scopes definitions are locally bound anyway.

`$sp = $STATE->convertUnit($unit);`

Converts a TeX unit of the form `'10em'` (or whatever TeX unit) into scaled points. (Defined here since in principle it could track the size of ems and so forth (but currently doesn't))

`LaTeXML::Core::Mouth`

Tokenize the input.

Description

A `LaTeXML::Core::Mouth` (and subclasses) is responsible for *tokenizing*, ie. converting plain text and strings into `LaTeXML::Core::Token`s according to the current category codes (catcodes) stored in the `LaTeXML::Core::State`.

It extends `LaTeXML::Common::Object`.

Creating Mouths

$mouth = LaTeXML::Core::Mouth->create($source, %options);

Creates a new Mouth of the appropriate class for reading from $source.

$mouth = LaTeXML::Core::Mouth->new($string, %options);

Creates a new Mouth reading from $string.

Methods

$token = $mouth->readToken;

Returns the next LaTeXML::Core::Token from the source.

$boole = $mouth->hasMoreInput;

Returns whether there is more data to read.

$string = $mouth->getLocator;

Return a description of current position in the source, for reporting errors.

$tokens = $mouth->readTokens;

Reads all remaining tokens in the mouth, removing any trailing space catcode tokens

$lines = $mouth->readRawLine;

Reads a raw (untokenized) line from $mouth, or undef if none is found.

LaTeXML::Core::Gullet

Expands expandable tokens and parses common token sequences.

Description

A LaTeXML::Core::Gullet reads tokens (LaTeXML::Core::Token) from a LaTeXML::Core::Mouth. It is responsible for expanding macros and expandable control sequences, if the current definition associated with the token in the LaTeXML::Core::State is an LaTeXML::Core::Definition::Expandable definition. The LaTeXML::Core::Gullet also provides a variety of methods for reading various types of input such as arguments, optional arguments, as well as for parsing LaTeXML::Common::Number, LaTeXML::Common::Dimension, etc, according to TeX's rules.

It extends LaTeXML::Common::Object.

Managing Input

`$gullet->openMouth($mouth, $noautoclose);`

Is this public? Prepares to read tokens from `$mouth`. If $noautoclose is true, the Mouth will not be automatically closed when it is exhausted.

`$gullet->closeMouth;`

Is this public? Finishes reading from the current mouth, and reverts to the one in effect before the last openMouth.

`$gullet->flush;`

Is this public? Clears all inputs.

`$gullet->getLocator;`

Returns an object describing the current location in the input stream.

Low-level methods

`$tokens = $gullet->expandTokens($tokens);`

Return the `LaTeXML::Core::Tokens` resulting from expanding all the tokens in `$tokens`. This is actually only used in a few circumstances where the arguments to an expandable need explicit expansion; usually expansion happens at the right time.

`$token = $gullet->readToken;`

Return the next token from the input source, or undef if there is no more input.

`$token = $gullet->readXToken($toplevel,$commentsok);`

Return the next unexpandable token from the input source, or undef if there is no more input. If the next token is expandable, it is expanded, and its expansion is reinserted into the input. If `$commentsok`, a comment read or pending will be returned.

`$gullet->unread(@tokens);`

Push the `@tokens` back into the input stream to be re-read.

Mid-level methods

`$token = $gullet->readNonSpace;`

Read and return the next non-space token from the input after discarding any spaces.

`$gullet->skipSpaces;`

Skip the next spaces from the input.

$gullet->skip1Space($expanded);

Skip the next token from the input if it is a space. If C($expanded> is true, expands (like <one optional space>).

$tokens = $gullet->readBalanced;

Read a sequence of tokens from the input until the balancing '}' (assuming the '{' has already been read). Returns a LaTeXML::Core::Tokens, except in an array context, returns the collected tokens and the closing token.

$boole = $gullet->ifNext($token);

Returns true if the next token in the input matches $token; the possibly matching token remains in the input.

$tokens = $gullet->readMatch(@choices);

Read and return whichever of @choices matches the input, or undef if none do. Each of the choices is an LaTeXML::Core::Tokens.

$keyword = $gullet->readKeyword(@keywords);

Read and return whichever of @keywords (each a string) matches the input, or undef if none do. This is similar to readMatch, but case and catcodes are ignored. Also, leading spaces are skipped.

$tokens = $gullet->readUntil(@delims);

Read and return a (balanced) sequence of LaTeXML::Core::Tokens until matching one of the tokens in @delims. In a list context, it also returns which of the delimiters ended the sequence.

High-level methods

$tokens = $gullet->readArg;

Read and return a TeX argument; the next Token or Tokens (if surrounded by braces).

$tokens = $gullet->readOptional($default);

Read and return a LaTeX optional argument; returns $default if there is no '[', otherwise the contents of the [].

$thing = $gullet->readValue($type);

Reads an argument of a given type: one of 'Number', 'Dimension', 'Glue', 'MuGlue' or 'any'.

$value = $gullet->readRegisterValue($type);

Read a control sequence token (and possibly it's arguments) that names a register, and return the value. Returns undef if the next token isn't such a register.

$number = $gullet->readNumber;

Read a LaTeXML::Common::Number according to TeX's rules of the various things that can be used as a numerical value.

$dimension = $gullet->readDimension;

Read a LaTeXML::Common::Dimension according to TeX's rules of the various things that can be used as a dimension value.

$mudimension = $gullet->readMuDimension;

Read a LaTeXML::Core::MuDimension according to TeX's rules of the various things that can be used as a mudimension value.

$glue = $gullet->readGlue;

Read a LaTeXML::Common::Glue according to TeX's rules of the various things that can be used as a glue value.

$muglue = $gullet->readMuGlue;

Read a LaTeXML::Core::MuGlue according to TeX's rules of the various things that can be used as a muglue value.

LaTeXML::Core::Stomach

Digests tokens into boxes, lists, etc.

Description

LaTeXML::Core::Stomach digests tokens read from a LaTeXML::Core::Gullet (they will have already been expanded).

It extends LaTeXML::Common::Object.

There are basically four cases when digesting a LaTeXML::Core::Token:

A plain character

is simply converted to a LaTeXML::Core::Box recording the current LaTeXML::Common::Font.

A primitive

If a control sequence represents LaTeXML::Core::Definition::Primitive, the primitive is invoked, executing its stored subroutine. This is typically done for side effect (changing the state in the LaTeXML::Core::State), although they may also contribute digested material. As with macros, any arguments to the primitive are read from the LaTeXML::Core::Gullet.

Grouping (or environment bodies)

are collected into a LaTeXML::Core::List.

Constructors

A special class of control sequence, called a LaTeXML::Core::Definition::Constructor produces a LaTeXML::Core::Whatsit which remembers the control sequence and arguments that created it, and defines its own translation into XML elements, attributes and data. Arguments to a constructor are read from the gullet and also digested.

Digestion

$list = $stomach->digestNextBody;

Return the digested LaTeXML::Core::List after reading and digesting a 'body' from the its Gullet. The body extends until the current level of boxing or environment is closed.

$list = $stomach->digest($tokens);

Return the LaTeXML::Core::List resuting from digesting the given tokens. This is typically used to digest arguments to primitives or constructors.

@boxes = $stomach->invokeToken($token);

Invoke the given (expanded) token. If it corresponds to a Primitive or Constructor, the definition will be invoked, reading any needed arguments fromt he current input source. Otherwise, the token will be digested. A List of Box's, Lists, Whatsit's is returned.

@boxes = $stomach->regurgitate;

Removes and returns a list of the boxes already digested at the current level. This peculiar beast is used by things like \choose (which is a Primitive in TeX, but a Constructor in LaTeXML).

Grouping

$stomach->bgroup;

Begin a new level of binding by pushing a new stack frame, and a new level of boxing the digested output.

$stomach->egroup;

End a level of binding by popping the last stack frame, undoing whatever bindings appeared there, and also decrementing the level of boxing.

$stomach->begingroup;

Begin a new level of binding by pushing a new stack frame.

$stomach->endgroup;

End a level of binding by popping the last stack frame, undoing whatever bindings appeared there.

Modes

$stomach->beginMode($mode);

Begin processing in $mode; one of 'text', 'display-math' or 'inline-math'. This also begins a new level of grouping and switches to a font appropriate for the mode.

$stomach->endMode($mode);

End processing in $mode; an error is signalled if $stomach is not currently in $mode. This also ends a level of grouping.

LaTeXML::Core::Document

Represents an XML document under construction.

Description

A LaTeXML::Core::Document represents an XML document being constructed by LaTeXML, and also provides the methods for constructing it. It extends LaTeXML::Common::Object.

LaTeXML will have digested the source material resulting in a LaTeXML::Core::List (from a LaTeXML::Core::Stomach) of LaTeXML::Core::Boxs, LaTeXML::Core::Whatsits and sublists. At this stage, a document is created and it is responsible for 'absorbing' the digested material. Generally, the LaTeXML::Core::Boxs and LaTeXML::Core::Lists create text nodes, whereas the LaTeXML::Core::Whatsits create XML document fragments, elements and attributes according to the defining LaTeXML::Core::Definition::Constructor.

Most document construction occurs at a *current insertion point* where material will be added, and which moves along with the inserted material. The LaTeXML::Common::Model, derived from various declarations and document type, is consulted to determine whether an insertion is allowed and when elements may need to be automatically opened or closed in order to carry out a given insertion. For example, a subsection element will typically be closed automatically when it is attempted to open a section element.

In the methods described here, the term $qname is used for XML qualified names. These are tag names with a namespace prefix. The prefix should be one registered with the current Model, for use within the code. This prefix is not necessarily the same as the one used in any DTD, but should be mapped to the a Namespace URI that was registered for the DTD.

The arguments named $node are an XML::LibXML node.

The methods here are grouped into three sections covering basic access to the document, insertion methods at the current insertion point, and less commonly used, lower-level, document manipulation methods.

Accessors

`$doc = $document->getDocument;`

Returns the `XML::LibXML::Document` currently being constructed.

`$doc = $document->getModel;`

Returns the `LaTeXML::Common::Model` that represents the document model used for this document.

`$node = $document->getNode;`

Returns the node at the *current insertion point* during construction. This node is considered still to be 'open'; any insertions will go into it (if possible). The node will be an `XML::LibXML::Element`, `XML::LibXML::Text` or, initially, `XML::LibXML::Document`.

`$node = $document->getElement;`

Returns the closest ancestor to the current insertion point that is an Element.

`$node = $document->getChildElement($node);`

Returns a list of the child elements, if any, of the `$node`.

`@nodes = $document->getLastChildElement($node);`

Returns the last child element of the `$node`, if it has one, else undef.

`$node = $document->getFirstChildElement($node);`

Returns the first child element of the `$node`, if it has one, else undef.

`@nodes = $document->findnodes($xpath,$node);`

Returns a list of nodes matching the given `$xpath` expression. The *context node* for `$xpath` is `$node`, if given, otherwise it is the document element.

`$node = $document->findnode($xpath,$node);`

Returns the first node matching the given `$xpath` expression. The *context node* for `$xpath` is `$node`, if given, otherwise it is the document element.

`$node = $document->getNodeQName($node);`

Returns the qualified name (localname with namespace prefix) of the given `$node`. The namespace prefix mapping is the code mapping of the current document model.

`$boolean = $document->canContain($tag,$child);`

Returns whether an element `$tag` can contain a child `$child`. `$tag` and `$child` can be nodes, qualified names of nodes (prefix:localname), or one of a set of special symbols `#PCDATA`, `#Comment`, `#Document` or `#ProcessingInstruction`.

`$boolean = $document->canContainIndirect($tag,$child);`

Returns whether an element `$tag` can contain a child `$child` either directly, or after automatically opening one or more autoOpen-able elements.

`$boolean = $document->canContainSomehow($tag,$child);`

Returns whether an element `$tag` can contain a child `$child` either directly, or after automatically opening one or more autoOpen-able elements.

`$boolean = $document->canHaveAttribute($tag,$attrib);`

Returns whether an element `$tag` can have an attribute named `$attrib`.

`$boolean = $document->canAutoOpen($tag);`

Returns whether an element `$tag` is able to be automatically opened.

`$boolean = $document->canAutoClose($node);`

Returns whether the node `$node` can be automatically closed.

Construction Methods These methods are the most common ones used for construction of documents. They generally operate by creating new material at the *current insertion point*. That point initially is just the document itself, but it moves along to follow any new insertions. These methods also adapt to the document model so as to automatically open or close elements, when it is required for the pending insertion and allowed by the document model (See *Tag*).

`$xmldoc = $document->finalize;`

This method finalizes the document by cleaning up various temporary attributes, and returns the *XML::LibXML::Document* that was constructed.

`@nodes = $document->absorb($digested);`

Absorb the `$digested` object into the document at the current insertion point according to its type. Various of the the other methods are invoked as needed, and document nodes may be automatically opened or closed according to the document model.

This method returns the nodes that were constructed. Note that the nodes may include children of other nodes, and nodes that may already have been removed from the document (See filterChildren and filterDeleted). Also, text insertions are often merged with existing text nodes; in such cases, the whole text node is included in the result.

`$document->insertElement($qname,$content,%attributes);`

This is a shorthand for creating an element `$qname` (with given attributes), absorbing `$content` from within that new node, and then closing it. The `$content` must be digested material, either a single box, or an array of boxes, which will be absorbed into the element. This method returns the newly created node, although it will no longer be the current insertion point.

$document->insertMathToken($string,%attributes);

Insert a math token (XMTok) containing the string `$string` with the given attributes. Useful attributes would be name, role, font. Returns the newly inserted node.

$document->insertComment($text);

Insert, and return, a comment with the given `$text` into the current node.

$document->insertPI($op,%attributes);

Insert, and return, a ProcessingInstruction into the current node.

$document->openText($text,$font);

Open a text node in font `$font`, performing any required automatic opening and closing of intermedate nodes (including those needed for font changes) and inserting the string `$text` into it.

$document->openElement($qname,%attributes);

Open an element, named `$qname` and with the given attributes. This will be inserted into the current node while performing any required automatic opening and closing of intermedate nodes. The new element is returned, and also becomes the current insertion point. An error (fatal if in `Strict` mode) is signalled if there is no allowed way to insert such an element into the current node.

$document->closeElement($qname);

Close the closest open element named `$qname` including any intermedate nodes that may be automatically closed. If that is not possible, signal an error. The closed node's parent becomes the current node. This method returns the closed node.

$node = $document->isOpenable($qname);

Check whether it is possible to open a `$qname` element at the current insertion point.

$node = $document->isCloseable($qname);

Check whether it is possible to close a `$qname` element, returning the node that would be closed if possible, otherwise undef.

$document->maybeCloseElement($qname);

Close a `$qname` element, if it is possible to do so, returns the closed node if it was found, else undef.

$document->addAttribute($key=>$value);

Add the given attribute to the node nearest to the current insertion point that is allowed to have it. This does not change the current insertion point.

$document->closeToNode($node);

This method closes all children of `$node` until `$node` becomes the insertion point. Note that it closes any open nodes, not only autoCloseable ones.

Internal Insertion Methods These are described as an aide to understanding the code; they rarely, if ever, should be used outside this module.

`$document->setNode($node);`

Sets the *current insertion point* to be `$node`. This should be rarely used, if at all; The construction methods of document generally maintain the notion of insertion point automatically. This may be useful to allow insertion into a different part of the document, but you probably want to set the insertion point back to the previous node, afterwards.

`$string = $document->getInsertionContext($levels);`

For debugging, return a string showing the context of the current insertion point; that is, the string of the nodes leading up to it. if `$levels` is defined, show only that many nodes.

`$node = $document->find_insertion_point($qname);`

This internal method is used to find the appropriate point, relative to the current insertion point, that an element with the specified `$qname` can be inserted. That position may require automatic opening or closing of elements, according to what is allowed by the document model.

`@nodes = getInsertionCandidates($node);`

Returns a list of elements where an arbitrary insertion might take place. Roughly this is a list starting with `$node`, followed by its parent and the parents siblings (in reverse order), followed by the grandparent and siblings (in reverse order).

`$node = $document->floatToElement($qname);`

Finds the nearest element at or preceding the current insertion point (see `getInsertionCandidates`), that can accept an element `$qname`; it moves the insertion point to that point, and returns the previous insertion point. Generally, after doing whatever you need at the new insertion point, you should call `$document->setNode($node);` to restore the insertion point. If no such point is found, the insertion point is left unchanged, and undef is returned.

`$node = $document->floatToAttribute($key);`

This method works the same as `floatToElement`, but find the nearest element that can accept the attribute `$key`.

`$node = $document->openText_internal($text);`

This is an internal method, used by `openText`, that assumes the insertion point has been appropriately adjusted.)

`$node = $document->openMathText_internal($text);`

This internal method appends `$text` to the current insertion point, which is assumed to be a math node. It checks for math ligatures and carries out any combinations called for.

$node = $document->closeText_internal();

This internal method closes the current node, which should be a text node. It carries out any text ligatures on the content.

$node = $document->closeNode_internal($node);

This internal method closes any open text or element nodes starting at the current insertion point, up to and including `$node`. Afterwards, the parent of `$node` will be the current insertion point. It condenses the tree to avoid redundant font switching elements.

$document->afterOpen($node);

Carries out any afterOpen operations that have been recorded (using `Tag`) for the element name of `$node`.

$document->afterClose($node);

Carries out any afterClose operations that have been recorded (using `Tag`) for the element name of `$node`.

Document Modification The following methods are used to perform various sorts of modification and rearrangements of the document, after the normal flow of insertion has taken place. These may be needed after an environment (or perhaps the whole document) has been completed and one needs to analyze what it contains to decide on the appropriate representation.

$document->setAttribute($node,$key,$value);

Sets the attribute `$key` to `$value` on `$node`. This method is preferred over the direct LibXML one, since it takes care of decoding namespaces (if `$key` is a qname), and also manages recording of xml:id's.

$document->recordID($id,$node);

Records the association of the given `$node` with the `$id`, which should be the `xml:id` attribute of the `$node`. Usually this association will be maintained by the methods that create nodes or set attributes.

$document->unRecordID($id);

Removes the node associated with the given `$id`, if any. This might be needed if a node is deleted.

$document->modifyID($id);

Adjusts `$id`, if needed, so that it is unique. It does this by appending a letter and incrementing until it finds an id that is not yet associated with a node.

$node = $document->lookupID($id);

Returns the node, if any, that is associated with the given `$id`.

$document->setNodeBox($node,$box);

Records the $box (being a Box, Whatsit or List), that was (presumably) responsible for the creation of the element $node. This information is useful for determining source locations, original TeX strings, and so forth.

$box = $document->getNodeBox($node);

Returns the $box that was responsible for creating the element $node.

$document->setNodeFont($node,$font);

Records the font object that encodes the font that should be used to display any text within the element $node.

$font = $document->getNodeFont($node);

Returns the font object associated with the element $node.

$node = $document->openElementAt($point,$qname,%attributes);

Opens a new child element in $point with the qualified name $qname and with the given attributes. This method is not affected by, nor does it affect, the current insertion point. It does manage namespaces, xml:id's and associating a box, font and locator with the new element, as well as running any afterOpen operations.

$node = $document->closeElementAt($node);

Closes $node. This method is not affected by, nor does it affect, the current insertion point. However, it does run any afterClose operations, so any element that was created using the lower-level openElementAt should be closed using this method.

$node = $document->appendClone($node,@newchildren);

Appends clones of @newchildren to $node. This method modifies any ids found within @newchildren (using modifyID), and fixes up any references to those ids within the clones so that they refer to the modified id.

$node = $document->wrapNodes($qname,@nodes);

This method wraps the @nodes by a new element with qualified name $qname, that new node replaces the first of @node. The remaining nodes in @nodes must be following siblings of the first one.

NOTE: Does this need multiple nodes? If so, perhaps some kind of movenodes helper? Otherwise, what about attributes?

$node = $document->unwrapNodes($node);

Unwrap the children of $node, by replacing $node by its children.

$node = $document->replaceNode($node,@nodes);

Replace $node by @nodes; presumably they are some sort of descendant nodes.

$node = $document->renameNode($node,$newname);

Rename $node to the tagname $newname; equivalently replace $node by a new node with name $newname and copy the attributes and contents. It is assumed that $newname can contain those attributes and contents.

@nodes = $document->filterDeletions(@nodes);

This function is useful with $doc->absorb($box), when you want to filter out any nodes that have been deleted and no longer appear in the document.

@nodes = $document->filterChildren(@nodes);

This function is useful with $doc->absorb($box), when you want to filter out any nodes that are children of other nodes in @nodes.

LaTeXML::Core::Rewrite

Rewrite rules for modifying the XML document.

Description

LaTeXML::Core::Rewrite implements rewrite rules for modifying the XML document. See LaTeXML::Package for declarations which create the rewrite rules. Further documentation needed.

LaTeXML::Core::Token

Representation of a Token: a pair of character and category code (catcode); It extends LaTeXML::Common::Object.

Exported functions

$catcode = CC_ESCAPE;

Constants for the category codes:

```
CC_BEGIN, CC_END, CC_MATH, CC_ALIGN, CC_EOL,
CC_PARAM, CC_SUPER, CC_SUB, CC_IGNORE,
CC_SPACE, CC_LETTER, CC_OTHER, CC_ACTIVE,
CC_COMMENT, CC_INVALID, CC_CS.
```

[The last 2 are (apparent) extensions, with catcodes 16 and 17, respectively].

$token = Token($string,$cc);

Creates a LaTeXML::Core::Token with the given content and catcode. The following shorthand versions are also exported for convenience:

```
T_BEGIN, T_END, T_MATH, T_ALIGN, T_PARAM,
T_SUB, T_SUPER, T_SPACE, T_LETTER($letter),
T_OTHER($char), T_ACTIVE($char),
T_COMMENT($comment), T_CS($cs)
```

@tokens = Explode($string);

Returns a list of the tokens corresponding to the characters in `$string`. All tokens have catcode CC_OTHER, except for spaces which have catcode CC_SPACE.

@tokens = ExplodeText($string);

Returns a list of the tokens corresponding to the characters in `$string`. All (roman) letters have catcode CC_LETTER, all others have catcode CC_OTHER, except for spaces which have catcode CC_SPACE.

UnTeX($object, $suppress_linebreaks);

Converts `$object` to a string containing TeX that created it (or could have). Note that this is not necessarily the original TeX code; expansions or other substitutions may have taken place.

Line-breaking of the generated TeX can be explicitly requested or disabled by passing 0 or 1 as the second `$suppress_linebreaks` argument. The default behavior of line-breaking is controlled by the global State value `SUPPRESS_UNTEX_LINEBREAKS`.

Methods

@tokens = $object->unlist;

Return a list of the tokens making up this `$object`.

$string = $object->toString;

Return a string representing `$object`.

$string = $token->getCSName;

Return the string or character part of the `$token`; for the special category codes, returns the standard string (eg. `T_BEGIN->getCSName` returns "{").

$string = $token->getString;

Return the string or character part of the `$token`.

$code = $token->getCharcode;

Return the character code of the character part of the `$token`, or 256 if it is a control sequence.

$code = $token->getCatcode;

Return the catcode of the `$token`.

LaTeXML::Core::Tokens

Represents lists of LaTeXML::Core::Token's; extends LaTeXML::Common::Object.

Exported functions

$tokens = Tokens(@token);

Creates a LaTeXML::Core::Tokens from a list of LaTeXML::Core::Token's

Tokens methods The following method is specific to LaTeXML::Core::Tokens.

$tokenscopy = $tokens->clone;

Return a shallow copy of the $tokens. This is useful before reading from a LaTeXML::Core::Tokens.

LaTeXML::Core::Box

Representations of digested objects; extends LaTeXML::Common::Object.

Exported Functions

$box = Box($string,$font,$locator,$tokens);

Creates a Box representing the $string in the given $font. The $locator records the document source position. The $tokens is a Tokens list containing the TeX that created (or could have) the Box. If $font or $locator are undef, they are obtained from the currently active LaTeXML::Core::State. Note that $string can be undef which contributes nothing to the generated document, but does record the TeX code (in $tokens).

Methods

$font = $digested->getFont;

Returns the font used by $digested.

$boole = $digested->isMath;

Returns whether $digested was created in math mode.

@boxes = $digested->unlist;

Returns a list of the boxes contained in $digested. It is also defined for the Boxes and Whatsit (which just return themselves) so they can stand-in for a List.

$string = $digested->toString;

Returns a string representing this $digested.

$string = $digested->revert;

Reverts the box to the list of Tokens that created (or could have created) it.

$string = $digested->getLocator;

Get an object describing the location in the original source that gave rise to $digested.

$digested->beAbsorbed($document);

$digested should get itself absorbed into the $document in whatever way is apppropriate.

$string = $box->getString;

Returns the string part of the $box.

LaTeXML::Core::List

Represents lists of digested objects; extends LaTeXML::Core::Box.

LaTeXML::Core::Comment

Representations of digested objects.

Description

LaTeXML::Core::Comment is a representation of digested objects. It extends LaTeXML::Common::Object.

LaTeXML::Core::Whatsit

Representations of digested objects.

Description

represents a digested object that can generate arbitrary elements in the XML Document. It extends LaTeXML::Core::Box.

Methods Note that the font is stored in the data properties under 'font'.

$defn = $whatsit->getDefinition;

Returns the LaTeXML::Core::Definition responsible for creating $whatsit.

$value = $whatsit->getProperty($key);

Returns the value associated with $key in the $whatsit's property list.

$whatsit->setProperty($key,$value);

Sets the $value associated with the $key in the $whatsit's property list.

$props = $whatsit->getProperties();

Returns the hash of properties stored on this Whatsit. (Note that this hash is modifiable).

$props = $whatsit->setProperties(%keysvalues);

Sets several properties, like setProperty.

$list = $whatsit->getArg($n);

Returns the $n-th argument (starting from 1) for this $whatsit.

@args = $whatsit->getArgs;

Returns the list of arguments for this $whatsit.

$whatsit->setArgs(@args);

Sets the list of arguments for this $whatsit to @args (each arg should be a LaTeXML::Core::List).

$list = $whatsit->getBody;

Return the body for this $whatsit. This is only defined for environments or top-level math formula. The body is stored in the properties under 'body'.

$whatsit->setBody(@body);

Sets the body of the $whatsit to the boxes in @body. The last $box in @body is assumed to represent the 'trailer', that is the result of the invocation that closed the environment or math. It is stored separately in the properties under 'trailer'.

$list = $whatsit->getTrailer;

Return the trailer for this $whatsit. See setBody.

LaTeXML::Core::Alignment

Representation of aligned structures

Description

This module defines aligned structures. It needs more documentation. It extends LaTeXML::Common::Object.

LaTeXML::Core::KeyVals

Key-Value Pairs in LaTeXML

Description

Provides a parser and representation of keyval pairs LaTeXML::Core::KeyVals represents parameters handled by LaTeX's keyval package. It extends LaTeXML::Common::Object.

Accessors

GetKeyVal($arg,$key)

Access the value associated with a given key. This is useful within constructors to access the value associated with `$key` in the argument `$arg`. Example usage in a copnstructor:

<foo attrib='&GetKeyVal(#1,'key')'>

GetKeyVals($arg)

Access the entire hash. Can be used in a constructor like: Can use in constructor: <foo %&GetKeyVals(#1)/>

Constructors

<LaTeXML::Core::KeyVals->new(*prefix*, *keysets*, *options*)>>

Creates a new KeyVals object with the given parameters. All arguments are optional and the simples way of calling this method is `my $keyvals = LaTeXML::Core::KeyVals->new()`.

prefix is the given prefix all key-value pairs operate in and defaults to `'KV'`. If given, prefix should be a string.

keysets should be a list of keysets to find keys inside of. If given, it should either be reference to a list of strings or a comma-seperated string. This argument defaults to `'_anonymous_'`.

Furthermore, the KeyVals constructor accepts a variety of options that can be used to customize its behaviour. These are *setAll*, *setInternals*, *skip*, *skipMissing*, *hookMissing*, *open*, *close*.

setAll is a flag that, if set, ensures that keys will be set in all existing keysets, instad of only in the first one.

setInternals is a flag that, if set, ensures that certain 'xkeyval' package internals are set during key digestion.

skip should be a list of keys to be skipped when digesting the keys of this object.

skipMissing allows one way of handling keys during key digestion that have not been explictilty declared using `DefKey` or related functionality. If set to `undef` or `0`, an error is thrown upon trying to set such a key, if set to `1` they are ignored. Alternatively, this can be set to a key macro which is then extended to contain a comman-separated list of the undefined keys.

hookMissing allows to call a specific macro if a single key is unknown during key digestion.

KeyVals Accessors (intended for internal usage)

`$keyvals->setTuples(@tuples)`

Sets the *tuples* which should be a list of five-tuples (array references) representing the key-value pairs this KeyVals object is seeded with. See the *getTuples* function on details of the structure of this list. *rebuild* is called automatically to populate the other caches. Typically, the tuples is set by *readFrom*.

Resolution to KeySets

`my @keysets = $keyvals->resolveKeyValFor($key)`

Finds all keysets that should be used for interacting with the given *key*. May return `undef` if no matching keysets are found. Use the parameters *keysets*, *setAll* and *skipMissing* to customize the exact behaviour of this function.

`my $canResolveKeyVal = $keyvals->canResolveKeyValFor($key)`

Checks if this *KeyVals* object can resolve a KeyVal for *key*. Ignores *setAll* and *skipMissing* parameters.

`my $keyval = $keyvals->getPrimaryKeyValOf($key, @keysets)`

Gets the primary keyset to be used for interacting a a single *key*, given that it resolves to *keysets*. Defaults to first keyset in KeyVals, if none given.

Changing contained values

`$keyvals->addValue($key, $value, $useDefault, $noRebuild)`

Adds the given *value* for *key* at the end of the given list of values and rebuilds all internal caches. If the *useDefault* flag is set, the specific value is ignored, and the default is set instead.

If this function is called multiple times the *noRebuild* option should be given to prevent constant rebuilding and the *rebuild* function should be called manually called.

`$keyvals->setValue($key, $value, $useDefault)`

Sets the value of *key* to *value*, optionally using the default if *useDefault* is set. Note that if *value* is a reference to an array, the key is inserted multiple times. If *value* is `undef`, the values is deleted.

`$keyvals->rebuild($skip)`

Rebuilds the internal caches of key-value mapping and list of pairs from from main list of tuples. If *skip* is given, all values for the given key are omitted, and the given key is deleted.

Parsing values from a gullet

`$keyvals->readFrom($gullet, $until, %options)`

Reads a set of KeyVals from *gullet*, up until the *until* token, and updates the state of this *KeyVals* object accordingly.

Furthermore, this methods supports several options.

When the *silenceMissing* option is set, missing keys will be completely ignored when reading keys, that is they do not get recorded into the KeyVals object and no warnings or errors will be thrown.

`$keyvals->readKeyWordFrom($gullet, $until)`

Reads a single keyword from *gullet*. Intended for internal use only.

KeyVals Accessors

`my $value = $keyvals->getValue($key);`

Return a value associated with `$key`.

`@values = $keyvals->getValues($key);`

Return the list of all values associated with `$key`.

`%keyvals = $keyvals->getKeyVals;`

Return the hash reference containing the keys and values bound in the `$keyval`. Each value in the hash may be a single value or a list if the key is repeated.

`@keyvals = $keyvals->getPairs;`

Return the alternating keys and values bound in the `$keyval`. Note that this may contain multiple entries for a given key, if they were repeated.

`%hash = $keyvals->getHash;`

Return the hash reference containing the keys and values bound in the `$keyval`. Note that will only contain the last value for a given key, if they were repeated.

`$haskey = $keyvals->hasKey($key);`

Checks if the KeyVals object contains a value for $key.

Value Related Reversion

`$expansion = $keyvals->setKeysExpansion;`

Expand this KeyVals into a set of tokens for digesting keys.

`$keyvals = $keyvals->beDigested($stomach);`

Return a new `LaTeXML::Core::KeyVals` object with both keys and values digested.

$reversion = $keyvals->revert();

Revert this object into a set of tokens representing the original sequence of Tokens that was used to be read it from the gullet.

$str = $keyvals->toString();

Turns this object into a key=value comma seperated string.

LaTeXML::Core::MuDimension

Representation of math dimensions; extends LaTeXML::Common::Dimension.

Exported functions

$mudimension = MuDimension($spec);

Creates a MuDimension object, similar to Dimension. $spec can be a string with a floating point number and "mu" or just a number standing for scaled mu (ie. fixedpoint).

LaTeXML::Core::MuGlue

Representation of math glue; extends LaTeXML::Common::Glue.

Exported functions

$glue = MuGlue($spec);

$glue = MuGlue($spec,$plus,$pfill,$minus,$mfill);

Creates a MuGlue object (similar to Glue). $spec can be a string in the form that TeX recognizes (number "mu" optional plus and minus parts). Alternatively, the dimension, plus and minus parts can be given separately as scaled mu (fixpoint), while $pfill and $mfill are 0 (when the $plus or $minus part is in scaledmu) or 1,2,3 for fil, fill or filll, respectively.

LaTeXML::Core::Pair

Representation of pairs of numerical things

Description

represents pairs of numerical things, coordinates or such. Candidate for removal!

Exported functions

$pair = Pair($num1,$num2);

Creates an object representing a pair of numbers; Not a part of TeX, but useful for graphical objects. The two components can be any numerical object.

LaTeXML::Core::PairList

Representation of lists of pairs of numerical things

Description

represents lists of pairs of numerical things, coordinates or such. Candidate for removal!

Exported functions

$pair = PairList(@pairs);

Creates an object representing a list of pairs of numbers; Not a part of TeX, but useful for graphical objects.

LaTeXML::Core::Definition

Control sequence definitions.

Description

This abstract class represents the various executables corresponding to control sequences. See LaTeXML::Package for the most convenient means to create them.

It extends LaTeXML::Common::Object.

Methods

$token = $defn->getCS;

Returns the (main) token that is bound to this definition.

$string = $defn->getCSName;

Returns the string form of the token bound to this definition, taking into account any alias for this definition.

$defn->readArguments($gullet);

Reads the arguments for this $defn from the $gullet, returning a list of LaTeXML::Core::Tokens.

$parameters = $defn->getParameters;

Return the LaTeXML::Core::Parameters object representing the formal parameters of the definition.

@tokens = $defn->invocation(@args);

Return the tokens that would invoke the given definition with the provided arguments. This is used to recreate the TeX code (or it's equivalent).

$defn->invoke;

Invoke the action of the $defn. For expandable definitions, this is done in the Gullet, and returns a list of LaTeXML::Core::Tokens. For primitives, it is carried out in the Stomach, and returns a list of LaTeXML::Core::Boxes. For a constructor, it is also carried out by the Stomach, and returns a LaTeXML::Core::Whatsit. That whatsit will be responsible for constructing the XML document fragment, when the LaTeXML::Core::Document invokes $whatsit->beAbsorbed($document);>.

Primitives and Constructors also support before and after daemons, lists of subroutines that are executed before and after digestion. These can be useful for changing modes, etc.

See also

LaTeXML::Core::Definition::Expandable, LaTeXML::Core::Definition::Conditional, LaTeXML::Core::Definition::Primitive, LaTeXML::Core::Definition::Register, LaTeXML::Core::Definition::CharDef and LaTeXML::Core::Definition::Constructor.

LaTeXML::Core::Definition::CharDef

Control sequence definitions for chardefs.

Description

Representation as a further specialized Register for chardef. See LaTeXML::Package for the most convenient means to create them. It extends LaTeXML::Core::Definition::Register.

LaTeXML::Core::Definition::Conditional

Conditionals Control sequence definitions.

Description

These represent the control sequences for conditionals, as well as \else, \or and \fi. See LaTeXML::Package for the most convenient means to create them.

It extends LaTeXML::Core::Definition::Expandable.

LaTeXML::Core::Definition::Constructor

Control sequence definitions.

Description

This class represents control sequences that contribute arbitrary XML fragments to the document tree. During digestion, a LaTeXML::Core::Definition::Constuctor records the arguments used in the invocation to produce a LaTeXML::Core::Whatsit.

The resulting LaTeXML::Core::Whatsit (usually) generates an XML document fragment when absorbed by an instance of LaTeXML::Core::Document. Additionally, a LaTeXML::Core::Definition::Constructor may have beforeDigest and afterDigest daemons defined which are executed for side effect, or for adding additional boxes to the output.

It extends LaTeXML::Core::Definition.

More documentation needed, but see LaTeXML::Package for the main user access to these.

More about Constructors A constructor has as it's replacement a subroutine or a string pattern representing the XML fragment it should generate. In the case of a string pattern, the pattern is compiled into a subroutine on first usage by the internal class LaTeXML::Core::Definition::ConstructorCompiler. Like primitives, constructors may have beforeDigest and afterDigest.

LaTeXML::Core::Definition::Expandable

Expandable Control sequence definitions.

Description

These represent macros and other expandable control sequences that are carried out in the Gullet during expansion. The results of invoking an LaTeXML::Core::Definition::Expandable should be a list of LaTeXML::Core::Tokens. See LaTeXML::Package for the most convenient means to create Expandables.

It extends LaTeXML::Core::Definition.

LaTeXML::Core::Definition::Primitive

Primitive Control sequence definitions.

Description

These represent primitive control sequences that are converted directly to Boxes or Lists containing basic Unicode content, rather than structured XML, or those executed for side effect during digestion in the LaTeXML::Core::Stomach, changing the LaTeXML::Core::State. The results of invoking a LaTeXML::Core::Definition::Primitive, if any, should be a list of digested items (LaTeXML::Core::Box, LaTeXML::Core::List or LaTeXML::Core::Whatsit).

It extends LaTeXML::Core::Definition.

Primitive definitions may have lists of daemon subroutines, beforeDigest and afterDigest, that are executed before (and before the arguments are read) and after digestion. These should either end with return;, (), or return a list of digested objects (LaTeXML::Core::Box, etc) that will be contributed to the current list.

LaTeXML::Core::Definition::Register

Control sequence definitions for Registers.

Description

These are set up as a speciallized primitive with a getter and setter to access and store values in the Stomach. See LaTeXML::Package for the most convenient means to create them.

It extends LaTeXML::Core::Definition::Primitive.

Registers generally store some value in the current LaTeXML::Core::State, but are not required to. Like TeX's registers, when they are digested, they expect an optional =, and then a value of the appropriate type. Register definitions support these additional methods:

Methods

$value = $register->valueOf(@args);

Return the value associated with the register, by invoking it's getter function. The additional args are used by some registers to index into a set, such as the index to \count.

$register->setValue($value,$scope,@args);

Assign a value to the register, by invoking it's setter function.

LaTeXML::Core::Parameter

A formal parameter

Description

Provides a representation for a single formal parameter of LaTeXML::Core::Definitions:
It extends LaTeXML::Common::Object.

See also

LaTeXML::Core::Parameters.

LaTeXML::Core::Parameters

Formal parameters.

Description

Provides a representation for the formal parameters of LaTeXML::Core::Definitions:
It extends LaTeXML::Common::Object.

Methods

`@parameters = $parameters->getParameters;`

Return the list of `LaTeXML::Core::Parameter` contained in `$parameters`.

`@tokens = $parameters->revertArguments(@args);`

Return a list of `LaTeXML::Core::Token` that would represent the arguments such that they can be parsed by the Gullet.

`@args = $parameters->readArguments($gullet,$fordefn);`

Read the arguments according to this `$parameters` from the `$gullet`. This takes into account any special forms of arguments, such as optional, delimited, etc.

`@args = $parameters->readArgumentsAndDigest($stomach,$fordefn);`

Reads and digests the arguments according to this `$parameters`, in sequence. this method is used by Constructors.

See also

`LaTeXML::Core::Parameter`.

C.3 Utility Modules Documentation

`LaTeXML::Util::Pathname`

Portable pathname and file-system utilities

Description

This module combines the functionality *File::Spec* and *File::Basename* to give a consistent set of filename utilities for LaTeXML. A pathname is represented by a simple string.

Pathname Manipulations

`$path = pathname_make(%peices);`

Constructs a pathname from the keywords in pieces dir : directory name : the filename (possibly with extension) type : the filename extension

`($dir,$name,$type) = pathname_split($path);`

Splits the pathname `$path` into the components: directory, name and type.

`$path = pathname_canonical($path);`

Canonicallizes the pathname `$path` by simplifying repeated slashes, dots representing the current or parent directory, etc.

$dir = pathname_directory($path);

Returns the directory component of the pathname $path.

$name = pathname_name($path);

Returns the name component of the pathname $path.

$type = pathname_type($path);

Returns the type component of the pathname $path.

$path = pathname_concat($dir,$file);

Returns the pathname resulting from concatenating the directory $dir and filename $file.

$boole = pathname_is_absolute($path);

Returns whether the pathname $path appears to be an absolute pathname.

$boole = pathname_is_url($path);

Returns whether the pathname $path appears to be a url, rather than local file.

$boole = pathname_is_literaldata($path);

Returns whether the pathname $path is actually a blob of literal data, with a leading "literal:" protocol.

$boole = pathname_is_raw($path);

Check if pathname indicates a raw TeX source or definition file.

$rel = pathname_is_contained($path,$base);

Checks whether $path is underneath the directory $base; if so it returns the pathname $path relative to $base; otherwise returns undef.

$path = pathname_relative($path,$base);

If $path is an absolute, non-URL pathname, returns the pathname relative to the directory $base, otherwise simply returns the canonical form of $path.

$path = pathname_absolute($path,$base);

Returns the absolute pathname resulting from interpretting $path relative to the directory $base. If $path is already absolute, it is returned unchanged.

$relative_url = pathname_to_url($path);

Creates a local, relative URL for a given pathname, also ensuring proper path separators on non-Unix systems.

File System Operations

`$modtime = pathname_timestamp($path);`

Returns the modification time of the file named by `$path`, or undef if the file does not exist.

`$path = pathname_cwd();`

Returns the current working directory.

`$dir = pathname_mkdir($dir);`

Creates the directory `$dir` and all missing ancestors. It returns `$dir` if successful, else undef.

`$dest = pathname_copy($source,$dest);`

Copies the file `$source` to `$dest` if needed; ie. if `$dest` is missing or older than `$source`. It preserves the timestamp of `$source`.

`$path = pathname_find($name,%options);`

Finds the first file named `$name` that exists and that matches the specification in the keywords `%options`. An absolute pathname is returned.

If `$name` is not already an absolute pathname, then the option `paths` determines directories to recursively search. It should be a list of pathnames, any relative paths are interpreted relative to the current directory. If `paths` is omitted, then the current directory is searched.

If the option `installation_subdir` is given, it indicates, in addition to the above, a directory relative to the LaTeXML installation directory to search. This allows files included with the distribution to be found.

The `types` option specifies a list of filetypes to search for. If not supplied, then the filename must match exactly. The type `*` matches any extension.

`@paths = pathname_findall($name,%options);`

Like `pathname_find`, but returns *all* matching (absolute) paths that exist.

`$path = pathname_kpsewhich(@names);`

Attempt to find a candidate name via the external `kpsewhich` capability of the system's TeX toolchain. If `kpsewhich` is not available, or the file is not found, returns a Perl undefined value.

LaTeXML::Util::WWW

Auxiliaries for web-scalability of LaTeXML's IO

Synopsis

```
my $response = auth_get($url,$authlist);
```

Description

Utilities for enabling general interaction with the World Wide Web in LaTeXML's Input/Output.

Still in development, more functionality is expected at a later stage.

Methods

`my $response = auth_get($url,$authlist);`

Given an authentication list, attempts a get request on a given URL ($url) and returns the $response.

If no authentication is possible automatically, the routine prompts the user for credentials.

`LaTeXML::Util::Pack`

Smart packing and unpacking of TeX archives

Description

This module provides an API and convenience methods for: 1. Unpacking Zip archives which contain a TeX manuscript. 2. Packing the files of a LaTeXML manuscript into a single archive 3. Extracting embeddable fragments, as well as single formulas from LaTeXML documents

All user-level methods are unconditionally exported by default.

Methods

`$main_tex_source = unpack_source($archive,$extraction_directory);`

Unpacks a given $archive into the $extraction_directory. Next, perform a heuristic analysis to determine, and return, the main file of the TeX manuscript. If the main file cannot be determined, the $extraction_directory is removed and undef is returned.

In this regard, we implement a simplified form of the logic in TeX::AutoTeX and particularly arXiv::FileGuess

`@packed_documents = pack_collection(collection=>\@documents, whatsout=>'math|fragment`

Packs a collection of documents using the packing method specified via the 'whatsout' option. If 'fragment' or 'math' are chosen, each input document is transformed into an embeddable fragment or a single formula, respectively. If 'archive' is chose, all input documents are written into an archive in the specified 'siteDirectory'. The name of the archive is provided by the 'destination' property of the first provided $document object. Each document is expected to be a LaTeXML::Post::Document object.

LaTeXML::Util::Radix

Simple radix conversion utilities

Description

This module provides some simple utilities for radix conversion.

radix_alpha($n)

Converts the number into one or more lowercase latin letters.

radix_Alpha($n)

Converts the number into one or more uppercase latin letters.

radix_greek($n)

Converts the number into one or more lowercase greek letters.

radix_Greek($n)

Converts the number into one or more uppercase greek letters.

radix_roman($n)

Converts the number as a lowercase roman numeral

radix_Roman($n)

Converts the number as a uppercase roman numeral

radix_format($n,@symbols)

(Internal) Converts the number into one or symbols taken from `symbols`.

C.4 Preprocessing Modules Documentation

LaTeXML::Pre::BibTeX

Implements a BibTeX parser for LaTeXML.

Description

`LaTeXML::Pre::BibTeX` serves as a low-level parser of BibTeX database files. It parses and stores a `LaTeXML::Pre::BibTeX::Entry` for each entry into the current STATE. BibTeX `string` macros are substituted into the field values, but no other processing of the data is done. See `LaTeXML::Package::BibTeX.pool.ltxml` for how further processing is carried out, and can be customized.

Creating a BibTeX

my $bib = LaTeXML::Pre::BibTeX->newFromFile($bibname);

Creates a LaTeXML::Pre::BibTeX object representing a bibliography from a BibTeX database file.

my $bib = LaTeXML::Pre::BibTeX->newFromString($string);

Creates a LaTeXML::Pre::BibTeX object representing a bibliography from a string containing the BibTeX data.

Methods

$string = $bib->toTeX;

Returns a string containing the TeX code to be digested by a LaTeXML object to process the bibliography. The string contains all @PREAMBLE data and invocations of \\ProcessBibTeXEntry{$key} for each bibliographic entry. The $key can be used to lookup the data from $STATE as LookupValue('BIBITEM@'.$key). See BibTeX.pool for how the processing is carried out.

BibEntry objects The representation of a BibTeX entry.

$type = $bibentry->getType;

Returns a string naming the entry type of the entry (No aliasing is done here).

$key = $bibentry->getKey;

Returns the bibliographic key for the entry.

@fields = $bibentry->getFields;

Returns a list of pairs [$name,$value] representing all fields, in the order defined, for the entry. Both the $name and $value are strings. Field names may be repeated, if they are in the bibliography.

$value = $bibentry->getField($name);

Returns the value (or undef) associated with the the given field name. If the field was repeated in the bibliography, only the last one is returned.

C.5 Postprocessing Modules Documentation

LaTeXML::Post

Postprocessing driver.

Description

LaTeXML::Post is the driver for various postprocessing operations. It has a complicated set of options that I'll document shortly.

LaTeXML::Post::MathML

Post-Processing modules for converting math to MathML.

Synopsis

LaTeXML::Post::MathML is the abstract base class for the MathML Postprocessor; LaTeXML::Post::MathML::Presentation and LaTeXML::Post::MathML::Content convert XMath to either Presentation or Content MathML, or with that format as the principle branch for Parallel markup.

Description

The conversion is carried out primarly by a tree walk of the XMath expression; appropriate handlers are selected and called depending on the operators and forms encountered. Handlers can be defined on applications of operators, or on tokens; when a token is applied, it's application handler takes precedence over it's token handler

DefMathML($key,$presentation,$content); Defines presentation and content handlers for $key. $key is of the form TYPE:ROLE:MEANING, where

```
TYPE    : is one either C<Token> or C<Apply> (or C<Hint> ?)
ROLE    : is a grammatical role (on XMath tokens)
MEANING : is the meaning attribute (on XMath tokens)
```

Any of these can be ? to match any role or meaning; matches of both are preferred, then match of meaning or role, or neither.

The subroutine handlers for presentation and content are given by $presentation and $content, respectively. Either can be undef, in which case some other matching handler will be invoked.

For Token handlers, the arguments passed are the token node; for Apply handler, the arguments passed are the operator node and any arguments.

However, it looks like some TOKEN handlers are being defined to take $content,%attributes being the string content of the token, and the token's attributes!

Presentation Conversion Utilties

$mmlpost->pmml_top($node,$style);

This is the top-level converter applied to an XMath node. It establishes a local context for font, style, size, etc. It generally does the bulk of the work for a PresentationMathML's translateNode, although the latter wraps the actual m:math element around it. (style is display or text).

pmml($node),pmml_smaller($node), pmml_scriptsizsize($node)

Converts the XMath $node to Presentation MathML. The latter two are used when the context calls for smaller (eg. fraction parts) or scriptsize (eg sub or superscript) size or style, so that the size encoded within $node will be properly accounted for.

pmml_mi($node,%attributes), pmml_mn($node,%attributes),pmml_mo($node,%attributes)

These are Token handlers, to create m:mi, m:mn and m:mo elements, respectively. When called as a handler, they will be supplied only with an XMath node (typically an XMTok). For convenient reuse, these functions may also be called on a 'virtual' token: with $node being a string (that would have been the text content of the XMTok), and the %attributes that would have been the token's attributes.

pmml_infix($op,@args),pmml_script($op,@args), pmml_bigop($op,@args)

These are Apply handlers, for handling general infix, sub or superscript, or bigop (eg. summations) constructs. They are called with the operator token, followed by the arguments; all are XMath elements.

pmml_row(@items)

This wraps an m:mrow around the already converted @items if need; That is, if there is only a single item it is returned without the m:mrow.

pmml_unrow($pmml)

This perverse utility takes something that has already been converted to Presentation MathML. If the argument is an m:mrow, it returns a list of the mathml elements within that row, otherwise it returns a list containing the single element $pmml.

pmml_parenthesize($item,$open,$close)

This utility parenthesizes the (already converted MathML) $item with the string delimiters $open and $close. These are converted to an m:mrow with m:mo for the fences, unless the usemfenced switch is set, in which case m:mfenced is used.

pmml_punctuate($separators,@items)

This utility creates an m:mrow by interjecting the punctuation between suceessive items in the list of already converted @items. If there are more than one character in $separators the first is used between the first pair, the next between the next pair; if the separators is exhausted, the last is repeated between remaining pairs. $separators defaults to (repeated) comma.

Content Conversion Utilties

$mmlpost->cmml_top($node); >

This is the top-level converter applied to an `XMath` node. It establishes a local context for font, style, size, etc (were it needed). It generally does the bulk of the work for a ContentMathML's `translateNode`, although the latter wraps the actual `m:math` element around it.

cmml($node)

Converts the `XMath` `$node` to Content MathML.

cmml_leaf($token)

Converts the `XMath` token to an `m:ci`, `m:cn` or `m:csymbol`, under appropriate circumstances.

cmml_decoratedSymbol($item)

Similar to `cmml_leaf`, but used when an operator is itself, apparently, an application. This converts `$item` to Presentation MathML to use for the content of the `m:ci`.

cmml_not($arg)

Construct the not of the argument `$arg`.

cmml_synth_not($op,@args)

Synthesize an operator by applying `m:not` to another operator (`$op`) applied to its `@args` (`XMath` elements that will be converted to Content MathML). This is useful to define a handler for, eg., c<not-approximately-equals> in terms of c<m:approx>.

cmml_synth_complement($op,@args)

Synthesize an operator by applying a complementary operator (`$op`) to the reverse of its `@args` (`XMath` elements that will be converted to Content MathML). This is useful to define a handler for, eg. `superset-of-or-equals` using `m:subset`.

cmml_or_compose($operators,@args)

Synthesize an operator that stands for the `or` of several other operators (eg. c<less-than-or-similar-to-or-approximately-equals>) by composing it of the `m:or` of applying each of `m:less` and `m:approx` to the arguments. The first operator is applied to the converted arguments, while the rest are applied to `m:share` elements referring to the previous ones.

cmml_share($node)

Converts the `XMath` `$node` to Content MathML, after assuring that it has an id, so that it can be shared.

cmml_shared($node)

Generates a `m:share` element referting to `$node`, which should have an id (such as after calling `cmml_share`).

Math Processors, Generally.

We should probably formalize the idea of a Math Processor as an abstract class, but let this description provide a starting overview. A MathProcessor follows the API of `LaTeXML::Post` processors, by handling `process`, which invokes `processNode` on all `Math` nodes; That latter inserts the result of either `translateNode` or `translateParallel`, applied to the `XMath` representation, into the `Math` node.

Parallel translation is done whenever additional MathProcessors have been specified, via the `setParallel` method; these are simply other MathProcessors following the same API.

Appendix D

LaTeXML Schema

The document type used by LaTeXML is modular in the sense that it is composed of several modules that define different sets of elements related to, eg., inline content, block content, math and high-level document structure. This allows the possibility of mixing models or extension by predefining certain parameter entities.

D.1 Module `LaTeXML`

Included `LaTeXML-common`, `LaTeXML-inline`, `LaTeXML-block`, `LaTeXML-misc`, `LaTeXML-meta`, `LaTeXML-para`, `LaTeXML-math`, `LaTeXML-tabular`, `LaTeXML-picture`, `LaTeXML-structure`, `LaTeXML-bib`

Pattern ***Inline.model*** Combined model for inline content.

> ***Content*****:** (*text* | *Inline.class* | *Misc.class* | *Meta.class*)*
>
> ***Used by*****:** acknowledgements, anchor, bib-data, bib-date, bib-edition, bib-extract, bib-identifier, bib-key, bib-language, bib-links, bib-note, bib-organization, bib-part, bib-place, bib-publisher, bib-review, bib-status, bib-subtitle, bib-title, bib-type, bib-url, bibrefphrase, cite, classification, constraint, contact, date, del, emph, givenname, glossaryphrase, glossaryref, indexphrase, indexrefs, indexsee, keywords, lineage, p, personname, ref, sub, subtitle, sup, surname, tag, text, verbatim

Pattern ***Block.model*** Combined model for physical block-level content.

> ***Content*****:** (*Block.class* | *Misc.class* | *Meta.class*)*
>
> ***Used by*****:** abstract, block, figure, float, inline-block, para, quote, table

Pattern ***Flow.model*** Combined model for general flow containing both inline and block level content.

***Content*:** (*text* | *Inline.class* | *Block.class* | *Misc.class* | *Meta.class*)*

***Used by*:** bibblock, note, rdf, td

Pattern Para.model Combined model for logical block-level context.

***Content*:** (*Para.class* | *Meta.class*)*

***Used by*:** *appendix.body.class*, *bibliography.body.class*, *chapter.body.class*, *document.body.class*, *glossary.body.class*, *index.body.class*, *paragraph.body.class*, *part.body.class*, *section.body.class*, *sidebar.body.class*, *slide.body.class*, *subparagraph.body.class*, *subsection.body.class*, *subsubsection.body.class*, inline-para, item, proof, theorem

Start == document

D.2 Module LaTeXML-common

Pattern Inline.class All strictly inline elements.

***Expansion*:** (text | emph | del | sub | sup | glossaryref | rule | anchor | ref | cite | bibref | Math)

***Used by*:** *Flow.model*, *Inline.model*, XMText, caption, clippath, g, inline-item, listingline, picture, title, toccaption, toctitle

Pattern Block.class All 'physical' block elements. A physical block is typically displayed as a block, but may not constitute a complete logical unit.

***Expansion*:** (p | equation | equationgroup | quote | block | listing | itemize | enumerate | description | pagination)

***Used by*:** *Block.model*, *Flow.model*, titlepage

Pattern Misc.class Additional miscellaneous elements that can appear in both inline and block contexts.

***Expansion*:** (inline-itemize | inline-enumerate | inline-description | inline-block | verbatim | break | graphics | svg:svg | rawhtml | rawliteral | inline-para | tabular | picture)

***Used by*:** *Block.model*, *Flow.model*, *Inline.model*, XMText, caption, clippath, creator, equation, g, inline-item, listingline, picture, title, toccaption, toctitle

Pattern Para.class All logical block level elements. A logical block typically contains one or more physical block elements. For example, a common situation might be p,equation,p, where the entire sequence comprises a single sentence.

***Expansion*:** (para | theorem | proof | figure | table | float | pagination | TOC)

***Used by*:** *BackMatter.class*, *Para.model*

***Pattern* Meta.class** All metadata elements, typically representing hidden data.

***Expansion*:** (note | declare | indexmark | glossarydefinition | rdf | ERROR | resource | navigation)

***Used by*:** *BackMatter.class*, *Block.model*, *Flow.model*, *Inline.model*, *Para.model*, caption, clippath, document, equation, equationgroup, g, inline-item, listingline, picture, title, toccaption, toctitle

***Pattern* Length.type** The type for attributes specifying a length. Should be a number followed by a length, typically px. NOTE: To be narrowed later.

***Content*:** *text*

***Used by*:** *Fontable.attributes*, *Positionable.attributes*, *Transformable.attributes*, XMArray, equationgroup, item, tabular, td

***Pattern* Color.type** The type for attributes specifying a color. NOTE: To be narrowed later.

***Content*:** *text*

***Pattern* Foreign.attributes** Attributes in foreign namespaces

Attribute* *: = *text*

Excluding attribute `xml:*`

***Used by*:** *Common.attributes*

***Pattern* Common.attributes** Attributes shared by ALL elements.

***Attributes*:** *Foreign.attributes*, *RDF.attributes*

***Attribute* class** = *NMTOKENS*

a space separated list of tokens, as in CSS. The class can be used to add differentiate different instances of elements without introducing new element declarations. However, this generally shouldn't be used for deep semantic distinctions. This attribute is carried over to HTML and can be used for CSS selection. [Note that the default XSLT stylesheets for html and xhtml add the latexml element names to the class of html elements for more convenience in using CSS.]

***Attribute* cssstyle** = *text*

CSS styling rules. These will only be effective when the target system supports CSS.

***Attribute* xml:lang** = *text*

Language attribute

***Used by*:** *Sectional.attributes*, ERROR, MathBranch, MathFork, Math, TOC, XMApp, XMArg, XMArray, XMCell, XMDual, XMHint, XMRef, XMRow, XMText, XMTok, XMWrap, XMath, abstract, acknowledgements, anchor, arc, bezier, bib-data, bib-date, bib-edition, bib-extract, bib-identifier, bib-key, bib-language, bib-links, bib-name, bib-note, bib-organization, bib-part, bib-place, bib-publisher, bib-related, bib-review, bib-status, bib-subtitle, bib-title, bib-type, bib-url, bibentry, bibitem, biblist, bibref, bibrefphrase, block, break, caption, circle, cite, classification, clip, clippath, contact, creator, curve, date, del, description, dots, ellipse, emph, enumerate, equation, equationgroup, figure, float, g, glossarydefinition, glossaryentry, glossarylist, glossaryphrase, glossaryref, graphics, grid, indexentry, indexlist, indexmark, indexphrase, indexrefs, indexsee, inline-block, inline-description, inline-enumerate, inline-item, inline-itemize, inline-para, item, itemize, keywords, line, listing, listingline, navigation, note, p, pagination, para, parabola, path, personname, picture, polygon, proof, quote, rdf, rect, ref, resource, rule, sub, subtitle, sup, table, tabular, tag, tbody, td, text, tfoot, thead, theorem, title, toccaption, tocentry, toclist, toctitle, tr, verbatim, wedge

Pattern* *ID.attributes Attributes for elements that can be cross-referenced from inside or outside the document.

***Attribute* xml:id** = *ID*

the unique identifier of the element, usually generated automatically by the latexml.

***Attribute* fragid** = *text*

a ”fragment identifier” derived from the xml:id relative to a page split from the complete document. This is used internally and may go away some day.

***Used by*:** *Labelled.attributes*, ERROR, Math, XMApp, XMArg, XMArray, XMCell, XMDual, XMHint, XMRef, XMRow, XMText, XMTok, XMWrap, XMath, anchor, bibentry, bibitem, block, declare, del, description, emph, enumerate, glossaryentry, glossarylist, graphics, indexentry, indexlist, inline-block, inline-description, inline-enumerate, inline-itemize, inline-para, itemize, p, para, picture, quote, sub, sup, tabular, td, text, tr, verbatim

Pattern* *IDREF.attributes Attributes for elements that can cross-reference other elements.

***Attribute* idref** = *IDREF*

the identifier of the referred-to element.

***Used by*:** *Refable.attributes*, XMRef, bibref, glossaryphrase

***Pattern* Listable.attributes** Attributes for items that can be put into lists, like index, table of contents.

***Attribute* inlist** = *text*

Records which lists, such as toc=table of contents,..., this object could be listed in. Space separated set of toc, lof, lot, etc.

***Used by*:** *Labelled.attributes*, bibref, cite, glossarydefinition, glossaryref, indexmark

***Pattern* Listing.attributes** Attributes for items that create lists, like index, table of contents.

***Attribute* lists** = *text*

Records which lists, such as toc(=table of contents), this object should create a list of. Space separated set of toc, lof, lot, etc.

***Used by*:** bibliography, glossary, index

***Pattern* Labelled.attributes** Attributes for elements that can be labelled from within LaTeX. These attributes deal with assigning a label and generating cross references. The label migrates to an xml:id and href and the element can serve as a hypertext target.

***Attributes*:** *ID.attributes*, *Listable.attributes*

***Attribute* labels** = *text*

Records the various labels that LaTeX uses for crossreferencing. (note that `\label` can associate more than one label with an object!) It consists of space separated labels for the element. The `\label` macro provides the label prefixed by `LABEL:`; Spaces in a label are replaced by underscore. Other mechanisms (like acro?) might use other prefixes (but `ID:` is reserved!)

***Used by*:** *Sectional.attributes*, equation, equationgroup, figure, float, inline-item, item, listing, listingline, note, proof, table, theorem

***Pattern* Refable.attributes** Attributes for elements that can be referred to from within LaTeX. Such elements may serve as the starting point of a hypertext link. The reference can be made using label, xml:id or href; these attributes will be converted, as needed, from the former to the latter.

***Attributes*:** *IDREF.attributes*

***Attribute* labelref** = *text*

reference to a LaTeX labelled object; See the labels attribute of *Labelled.attributes*.

***Attribute* href** = *text*

reference to an arbitrary url.

***Used by*:** bib-identifier, bib-review, bib-url, contact, glossaryref, personname, ref

***Pattern* Fontable.attributes** Attributes for elements that contain (indirectly) text whose font can be specified.

***Attribute* font** = *text*

Indicates the font to use. It consists of a space separated sequence of values representing the family (`serif`, `sansserif`, `math`, `typewriter`, `caligraphic`, `fraktur`, `script`, ...), series (`medium`, `bold`, ...), and shape (`upright`, `italic`, `slanted`, `smallcaps`, ...). Only the values differing from the current context are given. Each component is open-ended, for extensibility; it is thus unclear whether unknown values specify family, series or shape. In postprocessing, these values are carried to the class attribute, and can thus be effected by CSS.

***Attribute* fontsize** = *Length.type*

Indicates the text size to use, as a length, as in CSS. Normally, this should be a percentage value relative to the containing element.

***Used by*:** XMTok, caption, del, emph, glossaryref, ref, text, title, verbatim

***Pattern* Colorable.attributes** Attributes for elements that draw something, text or otherwise, that can be colored.

***Attribute* color** = *text*

the color to use (for foreground material); any CSS compatible color specification. In postprocessing, these values are carried to the class attribute, and can thus be effected by CSS.

***Attribute* opacity** = *float*

the opacity of foreground material; a number between 0 and 1.

***Used by*:** XMApp, XMTok, caption, del, emph, glossaryref, ref, rule, text, title, verbatim

***Pattern* Backgroundable.attributes** Attributes for elements that take up space and make sense to have a background color. This is independent of the colors of any things that it may draw.

***Attribute* backgroundcolor** = *text*

the color to use for the background of the element; any CSS compatible color specification. In postprocessing, these values are carried to the class attribute, and can thus be effected by CSS; the background will presumably correspond to a bounding rectangle, but is determined by the CSS rendering engine.

***Attribute* framed** = (`rectangle` | `underline` | *text*)

the kind of frame or outline for the box.

***Attribute* framecolor** = *text*

the color of the frame or outlie for the box.

***Used by*:** *Sectional.attributes*, Math, XMApp, XMCell, XMRow, XMText, XMTok, XMWrap, block, caption, constraint, del, description, emph, enumerate, equation, equationgroup, figure, float, glossaryref, inline-block, inline-description, inline-enumerate, inline-item, inline-itemize, inline-para, item, itemize, listing, p, para, proof, quote, ref, rule, table, tabular, tag, tbody, td, text, tfoot, thead, theorem, title, tr, verbatim

***Pattern* Positionable.attributes** Attributes shared by low-level, generic inline and block elements that can be sized or shifted.

***Attribute* width** = *Length.type*

the desired width of the box

***Attribute* height** = *Length.type*

the desired height of the box

***Attribute* depth** = *Length.type*

the desired depth of the box

***Attribute* xoffset** = *Length.type*

horizontal shift the position of the box.

***Attribute* yoffset** = *Length.type*

vertical shift the position of the box.

***Attribute* align** = (`left` | `center` | `right` | `justified`)

alignment of material within the box.

***Attribute* vattach** = (`top` | `middle` | `bottom` | `baseline`)

specifies which line of the box is aligned to the baseline of the containing object. The default is baseline.

***Attribute* float** = (`right` | `left` | *text*)

the horizontal floating placement parameter that determines where the object is displayed.

***Used by*:** *XMath.attributes*, block, figure, float, inline-block, inline-para, listing, p, para, rule, table, text

***Pattern* Transformable.attributes** Attributes shared by (hopefully few) elements that can be transformed. Such elements should also have Positionable.attributes. Transformation order of an individual element is assumed to be translate, scale, rotate; wrap elements to achieve different orders. Attributes innerwidth, innerheight and innerdepth describe the size of the contents of the element before transformation; The result size would be encoded in Positional.attributes.

***Attribute* xtranslate** = *Length.type*

horizontal shift the position of the inner element.

Attribute **ytranslate** = *Length.type*
vertical shift the position of the inner element.

Attribute **xscale** = *text*
horizontal scaling of the inner element.

Attribute **yscale** = *text*
vertical scalign of the inner element.

Attribute **angle** = *text*
the rotation angle, counter-clockwise, in degrees.

Attribute **innerwidth** = *Length.type*
the expected width of the contents of the inner element

Attribute **innerheight** = *Length.type*
the expected height of the contents of the inner element

Attribute **innerdepth** = *Length.type*
the expected depth of the contents of the inner element

***Used by*:** figure, float, g, inline-block, table

Pattern ***Imageable.attributes*** Attributes for elements that may be converted to image form during postprocessing, such as math, graphics, pictures, etc. Note that these attributes are generally not filled in until postprocessing, but that they could be init

Attribute **imagesrc** = *anyURI*
the file, possibly generated from other data.

Attribute **imagewidth** = *nonNegativeInteger*
the width in pixels of imagesrc.

Attribute **imageheight** = *nonNegativeInteger*
the height in pixels of imagesrc. Note that, unlike TEX, this is the total height, including the depth (if any).

Attribute **imagedepth** = *integer*
the depth in pixels of imagesrc, being the location of the baseline of the content shown in the image. When displayed inilne, an image with a positive depth should be shifted down relative to the baseline of neighboring material.

Attribute **description** = *text*
a description of the image

***Used by*:** Math, graphics, picture

Pattern ***RDF.attributes*** Attributes for RDFa (Resource Description Framework), following RDFa Core 1.1 http://www.w3.org/TR/rdfa-syntax/. The following descriptions give a short overview of the usage of the attributes, but see the specification for the complete details, which are sometimes complex.

Attribute **vocab** = *text*

indicates the default vocabulary (generally should be managed by LaTeXML and only appear on root node)

Attribute **prefix** = *text*

specifies a mapping between CURIE prefixes and IRI (URI). (generally should be managed by LaTeXML and only appear on root node)

Attribute **about** = *text*

indicates the subject for predicates appearing on the same or descendant nodes.

Attribute **aboutlabelref** = *text*

gives the label for the document element that serves as the subject; it will be converted to aboutidref and about during post-processing.

Attribute **aboutidref** = *text*

gives the id for the document element that serves as the subject; it will be converted to about during post-processing.

Attribute **resource** = *text*

indicates the subject for predicates appearing on descendant nodes, and also indicates the object for predicates when property appears on the same node, or when rel or rev appears on an ancestor.

Attribute **resourcelabelref** = *text*

gives the label for the document element that serves as the resource object; it will be converted to resourceidref and resource during post-processing.

Attribute **resourceidref** = *text*

gives the id for the document element that serves as the resource object; it will be converted to resource during post-processing.

Attribute **property** = *text*

indicates the predicate and asserts that the subject is related by that predicate to the object. The subject is determined by one of about on same node, resource or typeof on an ancestor node, or by the document root. The object is determined by one of resource, href, content or typeof on the same node, or by the text content of the node.

Attribute **rel** = *text*

indicates the predicate exactly as property except that it can assert multiple RDF triples where the objects are the nearest descendent resources.

Attribute **rev** = *text*

indicates the predicate exactly as rel except that it indicates the reverse relationship (with subject and object swapped).

Attribute **typeof** = *text*

indicates the type of the resource and thus implicitly asserts a relation that the object has the given type. Additionally, if no resource was given on the same node, indicates an anonymous subject and or object exactly as resource

Attribute **datatype** = *text*

indicates the datatype of the target resource;

Attribute **content** = *text*

indicates the content of the property to be used as the object, to be used instead of the content of the element itself;

Used by: *Common.attributes*

Pattern ***Data.attributes*** Attributes for raw data storage

Attribute **data** = *text*

the data itself

Attribute **datamimetype** = *text*

the MIME type of the data

Attribute **dataencoding** = *text*

the encoding of the data (either empty, base64, or)

Attribute **dataname** = *text*

the suggested file name of the data

Used by: figure, float, listing, proof, table

D.3 Module `LaTeXML-inline`

Add to ***Inline.class*** The inline module defines basic inline elements used throughout.

|= (text | emph | del | sub | sup | glossaryref | rule | anchor | ref | cite | bibref)

Element **text** General container for styled text. Attributes cover a variety of styling and position shifting properties.

Attributes: *Common.attributes*, *ID.attributes*, *Positionable.attributes*, *Fontable.attributes*, *Colorable.attributes*, *Backgroundable.attributes*

Content: *Inline.model*

Used by: *Inline.class*, MathFork, declare, equation

Element **emph** Emphasized text.

Attributes: *Common.attributes*, *ID.attributes*, *Fontable.attributes*, *Colorable.attributes*, *Backgroundable.attributes*

***Content*:** *Inline.model*

***Used by*:** *Inline.class*

Element **del** Deleted text.

***Attributes*:** *Common.attributes*, *ID.attributes*, *Fontable.attributes*, *Colorable.attributes*, *Backgroundable.attributes*

***Content*:** *Inline.model*

***Used by*:** *Inline.class*

Element **sub** Textual subscript text.

***Attributes*:** *Common.attributes*, *ID.attributes*

***Content*:** *Inline.model*

***Used by*:** *Inline.class*

Element **sup** Textual superscript text.

***Attributes*:** *Common.attributes*, *ID.attributes*

***Content*:** *Inline.model*

***Used by*:** *Inline.class*

Element **glossaryref** Represents the usage of a term from a glossary.

***Attributes*:** *Common.attributes*, *Refable.attributes*, *Listable.attributes*, *Fontable.attributes*, *Colorable.attributes*, *Backgroundable.attributes*

Attribute **key** = *text*

should be used to identifier used for the glossaryref.

Attribute **title** = *text*

gives a expanded form of the glossaryref (unused?),

Attribute **show** = *text*

a pattern encoding how the text content should be filled in during postprocessing, if it is empty. It consists of the words `type` (standing for the object type, eg. Ch.), `refnum`, `typerefnum` and `title` or `toctitle` (for the shortform of the title) mixed with arbitrary characters.

***Content*:** *Inline.model*

***Used by*:** *Inline.class*

Element **rule** A Rule.

***Attributes*:** *Common.attributes*, *Positionable.attributes*, *Colorable.attributes*, *Backgroundable.attributes*

***Content*:** *empty*

***Used by*:** *Inline.class*

***Element* ref** A hyperlink reference to some other object. When converted to HTML, the content would be the content of the anchor. The destination can be specified by one of the attributes labelref, idref or href; Missing fields will usually be filled in during postprocessing, based on data extracted from the document(s).

***Attributes*:** *Common.attributes*, *Refable.attributes*, *Fontable.attributes*, *Colorable.attributes*, *Backgroundable.attributes*

***Attribute* show** = *text*

a pattern encoding how the text content should be filled in during postprocessing, if it is empty. It consists of the words `type` (standing for the object type, eg. Ch.), `refnum` and `title` (including type and refnum) or `toctitle` (for the shortform of the title) mixed with arbitrary characters.

***Attribute* title** = *text*

gives a description of the target, not repeating the content, used for accessibility or a tooltip in HTML. Typically filled in by postprocessor.

***Attribute* fulltitle** = *text*

gives a longer form description of the target, useful when the link appears outside its original context, eg in navigation. Typically filled in by postprocessor.

***Content*:** *Inline.model*

***Used by*:** *Inline.class*, navigation, tocentry

***Element* anchor** Inline anchor.

***Attributes*:** *Common.attributes*, *ID.attributes*

***Content*:** *Inline.model*

***Used by*:** *Inline.class*

***Element* cite** A container for a bibliographic citation. The model is inline to allow arbitrary comments before and after the expected bibref(s) which are the specific citation.

***Attributes*:** *Common.attributes*, *Listable.attributes*

***Content*:** *Inline.model*

***Used by*:** *Inline.class*

***Element* bibref** A bibliographic citation refering to a specific bibliographic item. Postprocessing will turn this into an ref for the actual link.

***Attributes*:** *Common.attributes*, *IDREF.attributes*, *Listable.attributes*

***Attribute* bibrefs** = *text*

a comma separated list of bibliographic keys. (See the key attribute of bibitem and bibentry)

Attribute **show** = *text*

a pattern encoding how the text content (of an empty bibref) will be filled in. Consists of strings `author`, `fullauthor`, `year`, `number` and `title` (to be replaced by data from the bibliographic item) mixed with arbitrary characters.

Attribute **separator** = *text*

separator between formatted references

Attribute **yyseparator** = *text*

separator between formatted years when duplicate authors are combined.

***Content*:** bibrefphrase*

***Used by*:** *Inline.class*

Element **bibrefphrase** A preceding or following phrase used in composing a bibliographic reference, such as listing pages or chapter.

***Attributes*:** *Common.attributes*

***Content*:** *Inline.model*

***Used by*:** bibref

D.4 Module LaTeXML-block

Add to ***Block.class*** The block module defines the following 'physical' block elements.

|= (p | equation | equationgroup | quote | block | listing | itemize | enumerate | description | pagination)

Add to ***Misc.class*** These are inline forms of logical lists (they are included in Misc since that has been the general strategy)

|= (inline-itemize | inline-enumerate | inline-description)

Pattern ***EquationMeta.class*** Additional Metadata that can be present in equations.

***Content*:** constraint

***Used by*:** equation, equationgroup

Element **p** A physical paragraph.

***Attributes*:** *Common.attributes*, *ID.attributes*, *Positionable.attributes*, *Backgroundable.attributes*

***Content*:** *Inline.model*

***Used by*:** *Block.class*, equationgroup

Element **constraint** A constraint upon an equation.

Attributes**:** *Backgroundable.attributes*

Attribute **hidden** = *boolean*

Content**:** *Inline.model*

Used by**:** *EquationMeta.class*

Element **equation** An Equation. The model is just Inline which includes Math, the main expected ingredient. However, other things can end up in display math, too, so we use Inline. Note that tabular is here only because it's a common, if misguided, idiom; the processor will lift such elements out of math, when possible

Attributes**:** *Common.attributes*, *Labelled.attributes*, *Backgroundable.attributes*

Content**:** (tags | Math | MathFork | text | *Misc.class* | *Meta.class* | *EquationMeta.class*)*

Used by**:** *Block.class*, equationgroup, listingline

Element **equationgroup** A group of equations, perhaps aligned (Though this is nowhere recorded).

Attributes**:** *Common.attributes*, *Labelled.attributes*, *Backgroundable.attributes*

Attribute **rowsep** = *Length.type*

the spacing between rows (equations, intertext,...)

Content**:** (tags | equationgroup | equation | p | *Meta.class* | *EquationMeta.class*)*

Used by**:** *Block.class*, equationgroup, listingline

Element **MathFork** A wrapper for Math that provides alternative, but typically less semantically meaningful, formatted representations. The first child is the meaningful form, the extra children provide formatted forms, for example being table rows or cells arising from an eqnarray.

Attributes**:** *Common.attributes*

Content**:** ((Math | text), MathBranch*)

Used by**:** equation

Element **MathBranch** A container for an alternatively formatted math representation.

Attributes**:** *Common.attributes*

Attribute **format** = *text*

Content**:** (Math | tr | td)*

Used by: MathFork

Element **quote** A quotation.

Attributes: *Common.attributes*, *ID.attributes*, *Backgroundable.attributes*

Attribute **role** = *text*

The kind of quotation; could be something like verse, or translation

Content: *Block.model*

Used by: *Block.class*

Element **block** A generic block (fallback).

Attributes: *Common.attributes*, *ID.attributes*, *Positionable.attributes*, *Backgroundable.attributes*

Content: *Block.model*

Used by: *Block.class*

Element **listing** An Listing, (without caption: see float)

Attributes: *Common.attributes*, *Labelled.attributes*, *Positionable.attributes*, *Backgroundable.attributes*, *Data.attributes*

Content: listingline*

Used by: *Block.class*

Element **listingline** a line in a listing

Attributes: *Common.attributes*, *Labelled.attributes*

Content: (tags?, (*text* | *Inline.class* | *Misc.class* | *Meta.class* | equation | equationgroup)*)

Used by: listing

Element **itemize** An itemized list.

Attributes: *Common.attributes*, *ID.attributes*, *Backgroundable.attributes*

Content: item*

Used by: *Block.class*

Element **enumerate** An enumerated list.

Attributes: *Common.attributes*, *ID.attributes*, *Backgroundable.attributes*

Content: item*

Used by: *Block.class*

Element **description** A description list. The items within are expected to have a tag which represents the term being described in each item.

Attributes: *Common.attributes*, *ID.attributes*, *Backgroundable.attributes*

Content: item*

Used by: *Block.class*

Element **item** An item within a list (itemize,enumerate or description).

Attributes: *Common.attributes*, *Labelled.attributes*, *Backgroundable.attributes*

Attribute **itemsep** = *Length.type*

the vertical spacing between items

Content: (tags?, *Para.model*)

Used by: description, enumerate, itemize

Element **inline-itemize** An inline form of itemized list.

Attributes: *Common.attributes*, *ID.attributes*, *Backgroundable.attributes*

Content: inline-item*

Used by: *Misc.class*

Element **inline-enumerate** An inline form of enumerated list.

Attributes: *Common.attributes*, *ID.attributes*, *Backgroundable.attributes*

Content: inline-item*

Used by: *Misc.class*

Element **inline-description** An inline form of description list. The inline-items within are expected to have a tags which represents the term being described in each inline-item.

Attributes: *Common.attributes*, *ID.attributes*, *Backgroundable.attributes*

Content: inline-item*

Used by: *Misc.class*

Element **inline-item** An item within an inline list (inline-itemize,inline-enumerate or inline-description).

Attributes: *Common.attributes*, *Labelled.attributes*, *Backgroundable.attributes*

Content: (tags?, (*Inline.class* | *Misc.class* | *Meta.class*)*)

Used by: inline-description, inline-enumerate, inline-itemize

Element **tags** A container for one or more tags. At most one will have no role, which would be the default display. Other tag will have the role attribute for use in special forms of referencing.

Content: tag*

Used by: *SectionalFrontMatter.class*, bibitem, declare, equation, equationgroup, figure, float, inline-item, item, listingline, note, proof, table, theorem

Element **tag** A tag within an item indicating the term or bullet for a given item.

Attributes: *Common.attributes*, *Backgroundable.attributes*

Attribute **role** = *text*

specifies the purpose this tag is used for: no value is default display

Attribute **open** = *text*

specifies an open delimiters used to display the tag.

Attribute **close** = *text*

specifies an close delimiters used to display the tag.

Content: *Inline.model*

Used by: caption, tags, title, toccaption, toctitle

Element **pagination** A page break or related pagination information.

Attributes: *Common.attributes*

Attribute **role** = *text*

what kind of pagination

Content: *empty*

Used by: *Block.class*, *Para.class*

D.5 Module LaTeXML-misc

Add to ***Misc.class*** Miscellaneous (Misc) elements are (typically) visible elements which don't have clear inline or block character; they can appear in both inline and block contexts.

|= (inline-block | verbatim | break | graphics | svg:svg | rawhtml | rawliteral)

Element **inline-block** An inline block. Actually, can appear in inline or block mode, but typesets its contents as a block.

Attributes: *Common.attributes*, *ID.attributes*, *Positionable.attributes*, *Transformable.attributes*, *Backgroundable.attributes*

Content: *Block.model*

Used by: *Misc.class*

Element **verbatim** Verbatim content

Attributes: *Common.attributes*, *ID.attributes*, *Fontable.attributes*, *Colorable.attributes*, *Backgroundable.attributes*

***Content*:** *Inline.model*

***Used by*:** *Misc.class*

Element **break** A forced line break.

***Attributes*:** *Common.attributes*

***Content*:** *empty*

***Used by*:** *Misc.class*

Element **graphics** A graphical insertion of an external file.

***Attributes*:** *Common.attributes*, *ID.attributes*, *Imageable.attributes*

Attribute **graphic** = *text*

the path to the graphics file. This is the (often minimally specified) path to a graphics file omitting the type extension. Once resolved to a specific image file, the imagesrc (from Imageable.attributes) is used.

Attribute **candidates** = *text*

a comma separated list of candidate graphics files that could be used to for graphic. A post-processor or application may choose from these, or may make its own selection or synthesis to implement the graphic for a given target.

Attribute **options** = *text*

an encoding of the scaling and positioning options to be used in processing the graphic.

***Content*:** *empty*

***Used by*:** *Misc.class*

Element **rawhtml** A container for arbitrary markup in the xhtml namespace (not currently validated against any particular html schema)

***Content*:** `xhtml:*`*

***Used by*:** *Misc.class*

Element **rawliteral** A container for even more arbitrary directives like jsp, php, etc Doesn't create an element, but an open angle bracket followed by open then the text content, followed by a close angle bracket followed by close.

Attribute **open** = *text*

Attribute **close** = *text*

***Content*:** *text*

***Used by*:** *Misc.class*

D.6 Module LaTeXML-meta

***Add to* Meta.class** Meta elements are generally hidden; they can appear in both inline and block contexts.

|= (note | declare | indexmark | glossarydefinition | rdf | ERROR | resource | navigation)

***Element* note** Metadata that covers several 'out of band' annotations. It's content allows both inline and block-level content.

***Attributes*:** *Common.attributes*, *Labelled.attributes*

***Attribute* mark** = *text*

indicates the desired visible marker to be linked to the note.

***Attribute* role** = (`footnote` | *text*)

indicates the kind of note

***Content*:** (tags?, *Flow.model*)

***Used by*:** *Meta.class*

***Element* declare** declare records declarative mathematical information.

***Attributes*:** *ID.attributes*

***Attribute* type** = *text*

the type of declaration

***Attribute* definiens** = *text*

the thing being defined (if global), else must have xml:id

***Attribute* sortkey** = *text*

the sort key for use creating notation indices

***Content*:** (tags?, text?)

***Used by*:** *Meta.class*

***Element* indexmark** Metadata to record an indexing position. The content is a sequence of indexphrase, each representing a level in a multilevel indexing entry.

***Attributes*:** *Common.attributes*, *Listable.attributes*

***Attribute* see_also** = *text*

a flattened form (like key) of another indexmark, used to crossreference.

***Attribute* style** = *text*

NOTE: describe this.

***Content*:** (indexphrase*, indexsee*)

***Used by*:** *Meta.class*

***Element* indexphrase** A phrase within an indexmark

***Attributes*:** *Common.attributes*

***Attribute* key** = *text*

a flattened form of the phrase for generating an ID.

***Content*:** *Inline.model*

***Used by*:** indexentry, indexmark

***Element* indexsee** A see-also phrase within an indexmark

***Attributes*:** *Common.attributes*

***Attribute* key** = *text*

a flattened form of the phrase for generating an ID.

***Attribute* name** = *text*

a name for the see phrase, such as "see also".

***Content*:** *Inline.model*

***Used by*:** indexmark

***Element* glossarydefinition** A definition within an glossaryentry

***Attributes*:** *Common.attributes*, *Listable.attributes*

***Attribute* key** = *text*

a flattened form of the definition for generating an ID.

***Content*:** glossaryphrase*

***Used by*:** *Meta.class*

***Element* glossaryphrase** A phrase being clarified within an glossaryentry

***Attributes*:** *Common.attributes*, *IDREF.attributes*

***Attribute* key** = *text*

a flattened form of the phrase for generating an ID.

***Attribute* role** = *text*

a keyword naming the format of this phrase (to match show in glossaryref).

***Content*:** *Inline.model*

***Used by*:** glossarydefinition, glossaryentry

***Element* rdf** A container for RDF annotations. (See document structure for rdf-prefixes attribute)

***Attributes*:** *Common.attributes*

***Content*:** *Flow.model*

***Used by*:** *Meta.class*

***Element* ERROR** error object for undefined control sequences, or whatever

Attributes: *Common.attributes*, *ID.attributes*

Content: *text**

Used by: *Meta.class*, *XMath.class*

Element **resource** a resource for use in further processing such as javascript or CSS

Attributes: *Common.attributes*

Attribute **src** = *text*

the source url to the resource

Attribute **type** = *text*

the mime type of the resource

Attribute **media** = *text*

the media for which this resource is applicable (in the sense of CSS).

Content: *text**

Used by: *Meta.class*

Element **navigation** Records navigation cross-referencing information, or serves as a container for page navigational blocks. An inline-para child should have attribute class being one of `ltx_page_navbar`, `ltx_page_header` or `ltx_page_footer` and its contents will be used to create those components of webpages. Lacking those, a TOC requests a table of contents in the navigation bar. Page headers and footers will be synthesized from Links from the current page or document to related ones; these are represented by ref elements with rel being up, down, previous, next, and so forth.

Attributes: *Common.attributes*

Content: (ref | TOC | inline-para)*

Used by: *Meta.class*

D.7 Module `LaTeXML-para`

Add to ***Para.class*** This module defines the following 'logical' block elements.

|= (para | theorem | proof | figure | table | float | pagination)

Add to ***Misc.class*** Additionally, it defines these miscellaneous elements that can appear in both inline and block contexts.

|= inline-para

Element **para** A Logical paragraph. It has an id, but not a label.

Attributes: *Common.attributes*, *ID.attributes*, *Positionable.attributes*, *Backgroundable.attributes*

Content: *Block.model*

Used by: *Para.class*

Element **inline-para** An inline para. Actually, can appear in inline or block mode, but typesets its contents as para.

Attributes: *Common.attributes*, *ID.attributes*, *Positionable.attributes*, *Backgroundable.attributes*

Content: *Para.model*

Used by: *Misc.class*, navigation

Element **theorem** A theorem or similar object. The class attribute can be used to distinguish different kinds of theorem.

Attributes: *Common.attributes*, *Labelled.attributes*, *Backgroundable.attributes*

Content: (tags$^{?}$, title$^{?}$, *Para.model*)

Used by: *Para.class*

Element **proof** A proof or similar object. The class attribute can be used to distinguish different kinds of proof.

Attributes: *Common.attributes*, *Labelled.attributes*, *Backgroundable.attributes*, *Data.attributes*

Content: (tags$^{?}$, title$^{?}$, *Para.model*)

Used by: *Para.class*

Pattern ***Caption.class*** These are the additional elements representing figure and table captions. NOTE: Could title sensibly be reused here, instead? Or, should caption be used for theorem and proof?

Content: (caption | toccaption)

Used by: figure, float, table

Element **figure** A figure, possibly captioned.

Attributes: *Common.attributes*, *Labelled.attributes*, *Positionable.attributes*, *Transformable.attributes*, *Backgroundable.attributes*, *Data.attributes*

Attribute **placement** = *text*

the vertical floating placement parameter that determines where the object is displayed.

Content: (tags$^{?}$ | figure | table | float | *Block.model* | *Caption.class*)*

Used by: *Para.class*, figure, float, table

Element **table** A Table, possibly captioned. This is not necessarily a tabular.

***Attributes*:** *Common.attributes*, *Labelled.attributes*, *Positionable.attributes*, *Transformable.attributes*, *Backgroundable.attributes*, *Data.attributes*

***Attribute* placement** = *text*

the vertical floating placement parameter that determines where the object is displayed.

***Content*:** (tags? | table | figure | float | *Block.model* | *Caption.class*)*

***Used by*:** *Para.class*, figure, float, table

***Element* float** A generic float, possibly captioned, something other than a table or figure

***Attributes*:** *Common.attributes*, *Labelled.attributes*, *Positionable.attributes*, *Transformable.attributes*, *Backgroundable.attributes*, *Data.attributes*

***Attribute* role** = *text*

The kind of float; could be something like a listing, or some other thing

***Attribute* placement** = *text*

the vertical floating placement parameter that determines where the object is displayed.

***Content*:** (tags? | float | figure | table | *Block.model* | *Caption.class*)*

***Used by*:** *Para.class*, figure, float, table

***Element* caption** A caption for a table or figure.

***Attributes*:** *Common.attributes*, *Fontable.attributes*, *Colorable.attributes*, *Backgroundable.attributes*

***Content*:** (tag | *text* | *Inline.class* | *Misc.class* | *Meta.class*)*

***Used by*:** *Caption.class*

***Element* toccaption** A short form of table or figure caption, used for lists of figures or similar.

***Attributes*:** *Common.attributes*

***Content*:** (tag | *text* | *Inline.class* | *Misc.class* | *Meta.class*)*

***Used by*:** *Caption.class*

D.8 Module LaTeXML-math

Add to* *Inline.class The math module defines LaTeXML's internal representation of mathematical content, including the basic math container Math. This element is considered inline, as it will be contained within some other block-level element, eg. equation for display-math.

|= Math

***Pattern* Math.class** This class defines the content of the Math element. Additionally, it could contain MathML or OpenMath, after postprocessing.

***Content*:** XMath

***Used by*:** Math

***Pattern* XMath.class** These elements comprise the internal math representation, being the content of the XMath element.

***Content*:** (XMApp | XMTok | XMRef | XMHint | XMArg | XMWrap | XMDual | XMText | XMArray | ERROR)

***Used by*:** XMApp, XMArg, XMCell, XMDual, XMWrap, XMath

***Element* Math** Outer container for all math. This holds the internal XMath representation, as well as image data and other representations.

***Attributes*:** *Common.attributes*, *Imageable.attributes*, *ID.attributes*, *Backgroundable.attributes*

***Attribute* mode** = (`display` | `inline`)

display or inline mode.

***Attribute* tex** = *text*

reconstruction of the TEX that generated the math.

***Attribute* content-tex** = *text*

more semantic version of tex.

***Attribute* text** = *text*

a textified representation of the math.

***Attribute* lexemes** = *text*

preserved grammar-near lexemes for export to external apps

***Content*:** *Math.class**

***Used by*:** *Inline.class*, MathBranch, MathFork, equation

***Pattern* XMath.attributes** Common attributes for the various XMath elements.

***Attributes*:** *Positionable.attributes*

***Attribute* role** = *text*

The role that this item plays in the Grammar.

***Attribute* enclose** = *text*

an enclose style to enclose the object with legitimate values being those of MathML's menclose notations;

***Attribute* lpadding** = *text*

left, or leading, (presumably non-semantic) padding space.

Attribute **rpadding** = *text*

right, or trailing, (presumably non-semantic) padding space.

Attribute **name** = *text*

The name of the token, typically the control sequence that created it.

Attribute **meaning** = *text*

A more semantic name corresponding to the intended meaning, such as the OpenMath name.

Attribute **omcd** = *text*

The OpenMath CD for which meaning is a symbol.

Attribute **scriptpos** = *text*

An encoding of the position of sub/superscripts Before parsing, it takes two forms. On a base token or element, it is one of (pre—mid—post), indicating where any script can be placed. On a script token, it is an integer level. After parsing, the concatenation is moved to the sub—super-script ”operator”.

Attribute **possibleFunction** = *text*

an annotation placed by the parser when it suspects this token may be used as a function.

Attribute **decl_id** = *text*

an id to where the declaration of this object is given, preferably the xml:id of an ltx:declare

Attribute **href** = *text*

reference to an arbitrary url.

Used by: XMApp, XMArg, XMArray, XMDual, XMHint, XMRef, XMText, XMTok, XMWrap

Element **XMath** Internal representation of mathematics.

Attributes: *Common.attributes*, *ID.attributes*

Content: *XMath.class**

Used by: *Math.class*

Element **XMTok** General mathematical token.

Attributes: *Common.attributes*, *XMath.attributes*, *ID.attributes*, *Fontable.attributes*, *Colorable.attributes*, *Backgroundable.attributes*

Attribute **thickness** = *text*

A thickness used for drawing any lines which are part of presenting the token, such as the fraction line for the fraction operator.

Attribute **stretchy** = *boolean*

Whether or not the symbol should be stretchy. This shares MathML’s ambiguity about horizontal versus vertical stretchiness. When not set, defaults to whatever MathML’s operator dictionary says.

Attribute **mathstyle** = (`display` | `text` | `script` | `scriptscript`)
The math style used for displaying the application of this token when it represents some sort of fraction, variable-sized operator or stack of expressions (note that this applies to binomials or other stacks of expressions as well as fractions). Values of `display` or `text` correspond to TEX's displaystyle or textstyle, while `inline` indicates the stack should be arranged horizontally (the layout may depend on the operator).

Content: *text**

Used by: *XMath.class*

Element **XMApp** Generalized application of a function, operator, whatever (the first child) to arguments (the remaining children). The attributes are a subset of those for XMTok.

Attributes: *Common.attributes*, *XMath.attributes*, *ID.attributes*, *Colorable.attributes*, *Backgroundable.attributes*

Content: *XMath.class**

Used by: *XMath.class*

Element **XMDual** Parallel markup of content (first child) and presentation (second child) of a mathematical object. Typically, the arguments are shared between the two branches: they appear in the content branch, with id's, and XMRef is used in the presentation branch

Attributes: *Common.attributes*, *XMath.attributes*, *ID.attributes*

Content: (*XMath.class*, *XMath.class*)

Used by: *XMath.class*

Element **XMHint** Various layout hints, usually spacing, generally ignored in parsing. The attributes are a subset of those for XMTok.

Attributes: *Common.attributes*, *XMath.attributes*, *ID.attributes*

Content: *empty*

Used by: *XMath.class*

Element **XMText** Text appearing within math.

Attributes: *Common.attributes*, *XMath.attributes*, *Backgroundable.attributes*, *ID.attributes*

Content: (*text* | *Inline.class* | *Misc.class*)*

Used by: *XMath.class*

Element **XMWrap** Wrapper for a sequence of tokens used to assert the role of the contents in its parent. This element generally disappears after parsing. The attributes are a subset of those for XMTok.

***Attributes*:** *Common.attributes*, *XMath.attributes*, *Backgroundable.attributes*, *ID.attributes*

***Attribute* rule** = *text*

The grammatical rule that should apply to the contained sequence

***Attribute* style** = *text*

***Content*:** *XMath.class**

***Used by*:** *XMath.class*

***Element* XMArg** Wrapper for an argument to a structured macro. It implies that its content can be parsed independently of its parent, and thus generally disappears after parsing.

***Attributes*:** *Common.attributes*, *XMath.attributes*, *ID.attributes*

***Attribute* rule** = *text*

The grammatical rule that should apply to the contained sequence

***Content*:** *XMath.class**

***Used by*:** *XMath.class*

***Element* XMRef** Structure sharing element typically used in the presentation branch of an XMDual to refer to the arguments present in the content branch.

***Attributes*:** *Common.attributes*, *XMath.attributes*, *ID.attributes*, *IDREF.attributes*

***Content*:** *empty*

***Used by*:** *XMath.class*

***Element* XMArray** Math Array/Alignment structure.

***Attributes*:** *Common.attributes*, *XMath.attributes*, *ID.attributes*

***Attribute* rowsep** = *Length.type*

the spacing between rows

***Attribute* colsep** = *Length.type*

the spacing between columns

***Content*:** XMRow*

***Used by*:** *XMath.class*

***Element* XMRow** A row in a math alignment.

***Attributes*:** *Common.attributes*, *Backgroundable.attributes*, *ID.attributes*

***Content*:** XMCell*

***Used by*:** XMArray

***Element* XMCell** A cell in a row of a math alignment.

***Attributes*:** *Common.attributes*, *Backgroundable.attributes*, *ID.attributes*

***Attribute* colspan** = *nonNegativeInteger*

indicates how many columns this cell spans or covers.

***Attribute* rowspan** = *nonNegativeInteger*

indicates how many rows this cell spans or covers.

***Attribute* align** = *text*

specifies the alignment of the content.

***Attribute* width** = *text*

specifies the desired width for the column.

***Attribute* border** = *text*

records a sequence of t or tt, r or rr, b or bb and l or ll for borders or doubled borders on any side of the cell.

***Attribute* thead** = (`column` | `column row` | `row`)

whether this cell corresponds to a table row or column heading or both

***Content*:** *XMath.class**

***Used by*:** XMRow

D.9 Module `LaTeXML-tabular`

Add to* *Misc.class This module defines the basic tabular, or alignment, structure. It is roughly parallel to the HTML model.

|= tabular

***Element* tabular** An alignment structure corresponding to tabular or various similar forms. The model is basically a copy of HTML4's table.

***Attributes*:** *Common.attributes*, *Backgroundable.attributes*, *ID.attributes*

***Attribute* vattach** = (`top` | `middle` | `bottom`)

which row's baseline aligns with the container's baseline.

***Attribute* width** = *Length.type*

the desired width of the tabular.

***Attribute* rowsep** = *Length.type*

the spacing between rows

***Attribute* colsep** = *Length.type*

the spacing between columns

***Content*:** (thead | tfoot | tbody | tr)*

***Used by*:** *Misc.class*

***Element* thead** A container for a set of rows that correspond to the header of the tabular.

Attributes: *Common.attributes*, *Backgroundable.attributes*
Content: tr*
Used by: tabular

Element **tfoot** A container for a set of rows that correspond to the footer of the tabular.

Attributes: *Common.attributes*, *Backgroundable.attributes*
Content: tr*
Used by: tabular

Element **tbody** A container for a set of rows corresponding to the body of the tabular.

Attributes: *Common.attributes*, *Backgroundable.attributes*
Content: tr*
Used by: tabular

Element **tr** A row of a tabular.

Attributes: *Common.attributes*, *Backgroundable.attributes*, *ID.attributes*
Content: td*
Used by: MathBranch, tabular, tbody, tfoot, thead

Element **td** A cell in a row of a tabular.

Attributes: *Common.attributes*, *Backgroundable.attributes*, *ID.attributes*
Attribute **colspan** = *nonNegativeInteger*
indicates how many columns this cell spans or covers.
Attribute **rowspan** = *nonNegativeInteger*
indicates how many rows this cell spans or covers.
Attribute **align** = (`left` | `right` | `center` | `justify` | *text*)
specifies the horizontal alignment of the content. The allowed values are open-ended to accomodate `char:.` type alignments.
Attribute **width** = *Length.type*
specifies the desired width for the column.
Attribute **vattach** = (`top` | `middle` | `bottom`)
how the cell contents aligns with the row's baseline.
Attribute **border** = *text*
records a sequence of t or tt, r or rr, b or bb and l or ll for borders or doubled borders on any side of the cell.
Attribute **thead** = (`column` | `column row` | `row`)
whether this cell corresponds to a table row or column heading or both (whether in head or foot).
Content: *Flow.model*
Used by: MathBranch, tr

D.10 Module `LaTeXML-picture`

***Add to* Misc.class** This module defines a picture environment, roughly a subset of SVG. NOTE: Eventually we will drop these subset elements and incorporate SVG itself.

|= picture

***Pattern* Picture.class *Content*:** (g | rect | line | circle | path | arc | wedge | ellipse | polygon | bezier | parabola | curve | dots | grid | clip | `svg:svg`)

***Used by*:** clippath, g, picture

***Pattern* Picture.attributes** These attributes correspond roughly to SVG, but need documentation.

***Attribute* x** = *text*

***Attribute* y** = *text*

***Attribute* r** = *text*

***Attribute* rx** = *text*

***Attribute* ry** = *text*

***Attribute* width** = *text*

***Attribute* height** = *text*

***Attribute* fill** = *text*

***Attribute* stroke** = *text*

***Attribute* stroke-width** = *text*

***Attribute* stroke-dasharray** = *text*

***Attribute* transform** = *text*

***Attribute* terminators** = *text*

***Attribute* arrowlength** = *text*

***Attribute* points** = *text*

***Attribute* showpoints** = *text*

***Attribute* displayedpoints** = *text*

***Attribute* arc** = *text*

***Attribute* angle1** = *text*

***Attribute* angle2** = *text*

***Attribute* arcsepA** = *text*

***Attribute* arcsepB** = *text*

***Attribute* curvature** = *text*

***Used by*:** arc, bezier, circle, clip, clippath, curve, dots, ellipse, g, grid, line, parabola, path, picture, polygon, rect, wedge

***Pattern* PictureGroup.attributes** These attributes correspond roughly to SVG, but need documentation.

***Attribute* pos** = *text*

***Attribute* framed** = *boolean*

***Attribute* frametype** = (`rect` | `circle` | `oval`)

***Attribute* fillframe** = *boolean*

***Attribute* boxsep** = *text*

***Attribute* shadowbox** = *boolean*

***Attribute* doubleline** = *boolean*

***Used by*:** g

***Element* picture** A picture environment.

***Attributes*:** *Common.attributes*, *ID.attributes*, *Picture.attributes*, *Imageable.attributes*

***Attribute* clip** = *boolean*

***Attribute* baseline** = *text*

***Attribute* unitlength** = *text*

***Attribute* xunitlength** = *text*

***Attribute* yunitlength** = *text*

***Attribute* origin-x** = *text*

***Attribute* origin-y** = *text*

***Attribute* tex** = *text*

***Attribute* content-tex** = *text*

***Content*:** (*Picture.class* | *Inline.class* | *Misc.class* | *Meta.class*)*

***Used by*:** *Misc.class*

***Element* g** A graphical grouping; the content is inherits by the transformations, positioning and other properties.

***Attributes*:** *Common.attributes*, *Transformable.attributes*, *Picture.attributes*, *PictureGroup.attributes*

***Content*:** (*Picture.class* | *Inline.class* | *Misc.class* | *Meta.class*)*

***Used by*:** *Picture.class*

***Element* rect** A rectangle within a picture.

***Attributes*:** *Common.attributes*, *Picture.attributes*

***Content*:** *empty*

***Used by*:** *Picture.class*

***Element* line** A line within a picture.

***Attributes*:** *Common.attributes*, *Picture.attributes*

***Content*:** *empty*

***Used by*:** *Picture.class*

***Element* polygon** A polygon within a picture.

***Attributes*:** *Common.attributes*, *Picture.attributes*

***Content*:** *empty*

***Used by*:** *Picture.class*

***Element* wedge** A wedge within a picture.

***Attributes*:** *Common.attributes*, *Picture.attributes*

***Content*:** *empty*

***Used by*:** *Picture.class*

***Element* arc** An arc within a picture.

***Attributes*:** *Common.attributes*, *Picture.attributes*

***Content*:** *empty*

***Used by*:** *Picture.class*

***Element* circle** A circle within a picture.

***Attributes*:** *Common.attributes*, *Picture.attributes*

***Content*:** *empty*

***Used by*:** *Picture.class*

***Element* ellipse** An ellipse within a picture.

***Attributes*:** *Common.attributes*, *Picture.attributes*

***Content*:** *empty*

***Used by*:** *Picture.class*

***Element* path** A path within a picture.

***Attributes*:** *Common.attributes*, *Picture.attributes*

***Content*:** *empty*

***Used by*:** *Picture.class*

***Element* bezier** A bezier curve within a picture.

Attributes: *Common.attributes*, *Picture.attributes*

Content: *empty*

Used by: *Picture.class*

Element **curve** A curve within a picture.

Attributes: *Common.attributes*, *Picture.attributes*

Content: *empty*

Used by: *Picture.class*

Element **parabola** A parabola curve within a picture.

Attributes: *Common.attributes*, *Picture.attributes*

Content: *empty*

Used by: *Picture.class*

Element **dots** A sequence of dots (?) within a picture.

Attributes: *Common.attributes*, *Picture.attributes*

Attribute **dotstyle** = *text*

Attribute **dotsize** = *text*

Attribute **dotscale** = *text*

Content: *empty*

Used by: *Picture.class*

Element **grid** A grid within a picture.

Attributes: *Common.attributes*, *Picture.attributes*

Content: *empty*

Used by: *Picture.class*

Element **clip** Establishes a clipping region within a picture.

Attributes: *Common.attributes*, *Picture.attributes*

Content: clippath*

Used by: *Picture.class*

Element **clippath** Establishes a clipping region within a picture.

Attributes: *Common.attributes*, *Picture.attributes*

Content: (*Picture.class* | *Inline.class* | *Misc.class* | *Meta.class*)*

Used by: clip

D.11 Module `LaTeXML-structure`

Element **document** The document root.

Attributes: *Sectional.attributes*

Content: ((*FrontMatter.class* | *SectionalFrontMatter.class* | *Meta.class* | titlepage)*, (*document.body.class* | *BackMatter.class*)*)

Pattern ***document.body.class*** The content allowable as the main body of the document.

Content: (*Para.model* | paragraph | subsubsection | subsection | section | chapter | part | slide | slidesequence | sidebar)

Used by: document

Element **part** A part within a document.

Attributes: *Sectional.attributes*

Content: (*SectionalFrontMatter.class*, (*part.body.class* | *BackMatter.class*)*)

Used by: *document.body.class*

Pattern ***part.body.class*** The content allowable as the main body of a part.

Content: (*Para.model* | subparagraph | paragraph | subsubsection | subsection | section | chapter | slide | slidesequence | sidebar)

Used by: part

Element **chapter** A Chapter within a document.

Attributes: *Sectional.attributes*

Content: (*SectionalFrontMatter.class*, (*chapter.body.class* | *BackMatter.class*)*)

Used by: *document.body.class*, *part.body.class*

Pattern ***chapter.body.class*** The content allowable as the main body of a chapter.

Content: (*Para.model* | subparagraph | paragraph | subsubsection | subsection | section | slide | slidesequence | sidebar)

Used by: chapter

Element **section** A Section within a document.

Attributes: *Sectional.attributes*

Content: (*SectionalFrontMatter.class*, (*section.body.class* | *BackMatter.class*)*)

Used by: *appendix.body.class*, *chapter.body.class*, *document.body.class*, *part.body.class*

Pattern ***section.body.class*** The content allowable as the main body of a section.

Content: (*Para.model* | subparagraph | paragraph | subsubsection | subsection | slide | slidesequence | sidebar)

Used by: section

Element **subsection** A Subsection within a document.

Attributes: *Sectional.attributes*

Content: (*SectionalFrontMatter.class**, (*subsection.body.class* | *BackMatter.class*)*)

Used by: *appendix.body.class*, *chapter.body.class*, *document.body.class*, *part.body.class*, *section.body.class*

Pattern ***subsection.body.class*** The content allowable as the main body of a subsection.

Content: (*Para.model* | subparagraph | paragraph | subsubsection | slide | slidesequence | sidebar)

Used by: subsection

Element **subsubsection** A Subsubsection within a document.

Attributes: *Sectional.attributes*

Content: (*SectionalFrontMatter.class**, (*subsubsection.body.class* | *BackMatter.class*)*)

Used by: *appendix.body.class*, *chapter.body.class*, *document.body.class*, *part.body.class*, *section.body.class*, *subsection.body.class*

Pattern ***subsubsection.body.class*** The content allowable as the main body of a subsubsection.

Content: (*Para.model* | subparagraph | paragraph | slide | slidesequence | sidebar)

Used by: subsubsection

Element **paragraph** A Paragraph within a document. This corresponds to a 'formal' marked, possibly labelled LaTeX Paragraph, in distinction from an unlabelled logical paragraph.

Attributes: *Sectional.attributes*

Content: (*SectionalFrontMatter.class**, (*paragraph.body.class* | *BackMatter.class*)*)

***Used by*:** *appendix.body.class*, *chapter.body.class*, *document.body.class*, *part.body.class*, *section.body.class*, *subsection.body.class*, *subsubsection.body.class*

***Pattern* paragraph.body.class** The content allowable as the main body of a paragraph.

***Content*:** (*Para.model* | subparagraph | slide | slidesequence | sidebar)

***Used by*:** paragraph

***Element* subparagraph** A Subparagraph within a document.

***Attributes*:** *Sectional.attributes*

***Content*:** (*SectionalFrontMatter.class**, (*subparagraph.body.class* | *BackMatter.class*)*)

***Used by*:** *appendix.body.class*, *chapter.body.class*, *paragraph.body.class*, *part.body.class*, *section.body.class*, *subsection.body.class*, *subsubsection.body.class*

***Pattern* subparagraph.body.class** The content allowable as the main body of a subparagraph.

***Content*:** (*Para.model* | slide | slidesequence | sidebar)

***Used by*:** subparagraph

***Element* slidesequence** A slidesequence within a slideshow. Each slide contains a set slides, typically those that are revealed constructively.

***Attributes*:** *Sectional.attributes*

***Content*:** slide*

***Used by*:** *chapter.body.class*, *document.body.class*, *paragraph.body.class*, *part.body.class*, *section.body.class*, *subparagraph.body.class*, *subsection.body.class*, *subsubsection.body.class*

***Element* slide** A Slide within a slideshow, that may or may not be contained within a slidesequence.

***Attributes*:** *Sectional.attributes*

***Attribute* overlay** = *text*

overlay is the number of the current overlay. This must be specified when part of a slidesequence, else it may be omitted. Should be unique and rising within a slidesequence.

***Content*:** ((*SectionalFrontMatter.class* | subtitle)*, (*slide.body.class* | *BackMatter.class*)*)

Used by: *appendix.body.class*, *chapter.body.class*, *document.body.class*, *paragraph.body.class*, *part.body.class*, *section.body.class*, *subparagraph.body.class*, *subsection.body.class*, *subsubsection.body.class*, slidesequence

Pattern ***slide.body.class*** The content allowable as the main body of a slide.

Content: *Para.model*

Used by: slide

Element **sidebar** A Sidebar; a short section-like object that floats outside the main flow.

Attributes: *Sectional.attributes*

Content: ((*FrontMatter.class* | *SectionalFrontMatter.class*)*, (*sidebar.body.class* | *BackMatter.class*)*)

Used by: *appendix.body.class*, *chapter.body.class*, *document.body.class*, *paragraph.body.class*, *part.body.class*, *section.body.class*, *subparagraph.body.class*, *subsection.body.class*, *subsubsection.body.class*

Pattern ***sidebar.body.class*** The content allowable as the main body of a sidebar.

Content: *Para.model*

Used by: sidebar

Element **appendix** An Appendix within a document.

Attributes: *Sectional.attributes*

Content: (*SectionalFrontMatter.class**, *appendix.body.class**)

Used by: *BackMatter.class*

Pattern ***appendix.body.class*** The content allowable as the main body of a chapter.

Content: (*Para.model* | subparagraph | paragraph | subsubsection | subsection | section | slide | sidebar)

Used by: appendix

Element **bibliography** A Bibliography within a document.

Attributes: *Sectional.attributes*, *Listing.attributes*

Attribute **files** = *text*

the list of bib files used to create the bibliography.

Attribute **bibstyle** = *text*

the bibliographic style to be used to format the bibliography (presumably a BibTeX bst file name)

Attribute **citestyle** = *text*

the citation style to be used when citing items from the bibliography

Attribute **sort** = *boolean*

whether the bibliographic items should be sorted or in order of citation.

Content: (*FrontMatter.class**, *SectionalFrontMatter.class**, *bibliography.body.class**)

Used by: *BackMatter.class*

Pattern ***bibliography.body.class*** The content allowable as the main body of a chapter.

Content: (*Para.model* | biblist)

Used by: bibliography

Element **index** An Index within a document.

Attributes: *Sectional.attributes*, *Listing.attributes*

Attribute **role** = *text*

The kind of index (obsolete?)

Content: (*SectionalFrontMatter.class**, *index.body.class**)

Used by: *BackMatter.class*

Pattern ***index.body.class*** The content allowable as the main body of a chapter.

Content: (*Para.model* | indexlist)

Used by: index

Element **indexlist** A heirarchical index structure typically generated during postprocessing from the collection of indexmark in the document (or document collection).

Attributes: *Common.attributes*, *ID.attributes*

Content: indexentry*

Used by: *index.body.class*, indexentry

Element **indexentry** An entry in an indexlist consisting of a phrase, references to points in the document where the phrase was found, and possibly a nested indexlist represented index levels below this one.

Attributes: *Common.attributes*, *ID.attributes*

Content: (indexphrase, indexrefs?, indexlist?)

Used by: indexlist

Element **indexrefs** A container for the references (ref) to where an indexphrase was encountered in the document. The model is Inline to allow arbitrary text, in addition to the expected ref's.

Attributes: *Common.attributes*

Content: *Inline.model*

Used by: glossaryentry, indexentry

Element **glossary** An Glossary within a document.

Attributes: *Sectional.attributes*, *Listing.attributes*

Attribute **role** = *text*

The kind of glossary

Content: (*SectionalFrontMatter.class**, *glossary.body.class**)

Used by: *BackMatter.class*

Pattern ***glossary.body.class*** The content allowable as the main body of a chapter.

Content: (*Para.model* | glossarylist)

Used by: glossary

Element **glossarylist** A glossary list typically generated during postprocessing from the collection of glossaryphrase's in the document (or document collection).

Attributes: *Common.attributes*, *ID.attributes*

Content: glossaryentry*

Used by: *glossary.body.class*

Element **glossaryentry** An entry in an glossarylist consisting of a phrase, (one or more, presumably in increasing detail?), possibly a definition, and references to points in the document where the phrase was found.

Attributes: *Common.attributes*, *ID.attributes*

Attribute **role** = *text*

The kind of glossary

Attribute **key** = *text*

a flattened form of the phrase for generating an ID.

Content: (glossaryphrase*, indexrefs?)

Used by: glossarylist

Element **title** The title of a document, section or similar document structure container.

Attributes: *Common.attributes*, *Fontable.attributes*, *Colorable.attributes*, *Backgroundable.attributes*

Content: (tag | *text* | *Inline.class* | *Misc.class* | *Meta.class*)*

Used by: *SectionalFrontMatter.class*, TOC, proof, theorem

Element **toctitle** The short form of a title, for use in tables of contents or similar.

Attributes: *Common.attributes*

Content: (tag | *text* | *Inline.class* | *Misc.class* | *Meta.class*)*

Used by: *SectionalFrontMatter.class*

Element **subtitle** A subtitle, or secondary title.

Attributes: *Common.attributes*

Content: *Inline.model*

Used by: *FrontMatter.class*, slide

Element **creator** Generalized document creator.

Attributes: *Common.attributes*, *FrontMatter.attributes*

Attribute **role** = (`author` | `editor` | `translator` | `contributor` | `translator` | *text*)

indicates the role of the person in creating the docment. Commonly useful values are specified, but is open-ended to support extension.

Attribute **before** = *text*

specifies opening text to display before this creator in a formatted titlepage. This would be typically appear outside the author information, like ”and”.

Attribute **after** = *text*

specifies closing text, punctuation or conjunction to display after this creator in a formatted titlepage.

Content: (*Person.class* | *Misc.class*)*

Used by: *SectionalFrontMatter.class*

Pattern ***Person.class*** The content allowed in authors, editors, etc.

Content: (personname | contact)

Used by: creator

Element **personname** A person’s name.

Attributes: *Common.attributes*, *Refable.attributes*

Content: *Inline.model*

Used by: *Person.class*

Element **contact** Generalized contact information for a document creator. Note that this element can be repeated to give different types of contact information (using role) for the same creator.

Attributes: *Common.attributes*, *FrontMatter.attributes*, *Refable.attributes*

Attribute **role** = (`affiliation` | `address` | `current_address` | `email` | `url` | `thanks` | `dedicatory` | `orcid` | *text*)
indicates the type of contact information contained. Commonly useful values are specified, but is open-ended to support extension.

***Content*:** *Inline.model*

***Used by*:** *Person.class*

Element **date** Generalized document date. Note that this element can be repeated to give the dates of different events (using role) for the same document.

***Attributes*:** *Common.attributes*, *FrontMatter.attributes*

Attribute **role** = (`creation` | `conversion` | `posted` | `received` | `revised` | `accepted` | *text*)
indicates the relevance of the date to the document. Commonly useful values are specified, but is open-ended to support extension.

***Content*:** *Inline.model*

***Used by*:** *FrontMatter.class*

Element **abstract** A document abstract.

***Attributes*:** *Common.attributes*, *FrontMatter.attributes*

***Content*:** *Block.model*

***Used by*:** *FrontMatter.class*

Element **acknowledgements** Acknowledgements for the document.

***Attributes*:** *Common.attributes*, *FrontMatter.attributes*

***Content*:** *Inline.model*

***Used by*:** *BackMatter.class*, *FrontMatter.class*

Element **keywords** Keywords for the document. The content is freeform.

***Attributes*:** *Common.attributes*, *FrontMatter.attributes*

***Content*:** *Inline.model*

***Used by*:** *FrontMatter.class*

Element **classification** A classification of the document.

***Attributes*:** *Common.attributes*, *FrontMatter.attributes*

Attribute **scheme** = *text*
indicates what classification scheme was used.

***Content*:** *Inline.model*

***Used by*:** *FrontMatter.class*

Element **titlepage** block of random stuff marked as a titlepage

Attributes: *Sectional.attributes*

Content: (*FrontMatter.class* | *SectionalFrontMatter.class* | *Block.class*)*

Used by: document

Element **TOC** (Generalized) Table Of Contents, represents table of contents as well as list of figures, tables, and other such things. This will generally be placed by a `\tableofcontents` command and filled in by postprocessing.

Attributes: *Common.attributes*, *FrontMatter.attributes*

Attribute **lists** = *text*

indicates the kind of lists; space separated names of lists like "toc","lof", etc.

Attribute **select** = *text*

indicates what kind of document elements to list, in the form of one or more tags such as `ltx:chapter` separated by `|` (suggestive of an xpath expression).

Attribute **scope** = (`current` | `global` | *text*)

indicates the scope set of elements to include: `current` (default) is all in current document; `global` indicates all in the document set; otherwise an xml:id

Attribute **show** = *text*

indicates what things to show in each entry

Attribute **format** = (`normal` | `short` | `veryshort` | *text*)

indicates how to format the listing

Content: (title?, toclist?)

Used by: *Para.class*, navigation

Element **toclist** The actual table of contents list, filled in.

Attributes: *Common.attributes*

Content: tocentry*

Used by: TOC, tocentry

Element **tocentry** An entry in a toclist.

Attributes: *Common.attributes*

Content: (ref | toclist)*

Used by: toclist

Pattern ***Sectional.attributes*** Attributes shared by all sectional elements

Attributes: *Common.attributes*, *Labelled.attributes*, *Backgroundable.attributes*

Attribute **rdf-prefixes** = *text*

Stores RDFa prefixes as space separated pairs, with the pairs being prefix and url separated by a colon; this should only appear at the root element.

***Used by*:** appendix, bibliography, chapter, document, glossary, index, paragraph, part, section, sidebar, slide, slidesequence, subparagraph, subsection, subsubsection, titlepage

Pattern ***FrontMatter.attributes*** Attributes for other elements that can be used in frontmatter.

Attribute **name** = *text*

Records the name of the type of object this is to be used when composing the presentation. The value of this attribute is often set by language localization packages.

***Used by*:** TOC, abstract, acknowledgements, classification, contact, creator, date, keywords

Pattern ***SectionalFrontMatter.class*** The content allowed for the front matter of each sectional unit, and the document.

***Content*:** (tags$^{?}$ | title | toctitle | creator)

***Used by*:** appendix, bibliography, chapter, document, glossary, index, paragraph, part, section, sidebar, slide, subparagraph, subsection, subsubsection, titlepage

Pattern ***FrontMatter.class*** The content allowed (in addition to *SectionalFrontMatter.class*) for the front matter of a document.

***Content*:** (subtitle | date | abstract | acknowledgements | keywords | classification)

***Used by*:** bibliography, document, sidebar, titlepage

Pattern ***BackMatter.class*** The content allowed a the end of a document. Note that this includes random trailing Block and Para material, to support articles with figures and similar data appearing ‘at end’.

***Content*:** (bibliography | appendix | index | glossary | acknowledgements | *Para.class* | *Meta.class*)

***Used by*:** chapter, document, paragraph, part, section, sidebar, slide, subparagraph, subsection, subsubsection

Add to ***Para.class***

|= TOC

D.12 Module `LaTeXML-bib`

Element **biblist** A list of bibliographic bibentry or bibitem.

***Attributes*:** *Common.attributes*

***Content*:** (bibentry | bibitem)*

***Used by*:** *bibliography.body.class*

Element **bibitem** A formatted bibliographic item, typically as written explicit in a LaTeX article. This has generally lost most of the semantics present in the BibTeX data.

***Attributes*:** *Common.attributes*, *ID.attributes*

Attribute **key** = *text*

The unique key for this object; this key is referenced by the bibrefs attribute of bibref.

***Content*:** (tags?, bibblock*)

***Used by*:** biblist

Element **bibblock** A block of data appearing within a bibitem.

***Content*:** *Flow.model*

***Used by*:** bibitem

Element **bibentry** Semantic representation of a bibliography entry, typically resulting from parsing BibTeX

***Attributes*:** *Common.attributes*, *ID.attributes*

Attribute **key** = *text*

The unique key for this object; this key is referenced by the bibrefs attribute of bibref.

Attribute **type** = *bibentry.type*

The type of the referenced object. The values are a superset of those types recognized by BibTeX, but is also open-ended for extensibility.

***Content*:** *Bibentry.class**

***Used by*:** biblist

Pattern ***bibentry.type*** ***Content*:** (`article` | `book` | `booklet` | `conference` | `inbook` | `incollection` | `inproceedings` | `manual` | `mastersthesis` | `misc` | `phdthesis` | `proceedings` | `techreport` | `unpublished` | `report` | `thesis` | `website` | `software` | `periodical` | `collection` | `collection.article` | `proceedings.article` | *text*)

***Used by*:** bib-related, bibentry

***Element* bib-name** Name of some participant in creating a bibliographic entry.

***Attributes*:** *Common.attributes*

***Attribute* role** = (`author` | `editor` | `translator` | *text*)
The role that this participant played in creating the entry.

***Content*:** *Bibname.model*

***Used by*:** *Bibentry.class*

***Pattern* Bibname.model** The content model of the bibliographic name fields (bib-name)

***Content*:** surname, givenname?, lineage?

***Expansion*:** (surname, givenname?, lineage?)

***Used by*:** bib-name

***Element* surname** Surname of a participant (bib-name).

***Content*:** *Inline.model*

***Used by*:** *Bibname.model*

***Element* givenname** Given name of a participant (bib-name).

***Content*:** *Inline.model*

***Used by*:** *Bibname.model*

***Element* lineage** Lineage of a participant (bib-name), eg. Jr. or similar.

***Content*:** *Inline.model*

***Used by*:** *Bibname.model*

***Element* bib-title** Title of a bibliographic entry.

***Attributes*:** *Common.attributes*

***Content*:** *Inline.model*

***Used by*:** *Bibentry.class*

***Element* bib-subtitle** Subtitle of a bibliographic entry.

***Attributes*:** *Common.attributes*

***Content*:** *Inline.model*

***Used by*:** *Bibentry.class*

***Element* bib-key** Unique key of a bibliographic entry.

***Attributes*:** *Common.attributes*

***Content*:** *Inline.model*

***Used by*:** *Bibentry.class*

Element **bib-type** Type of a bibliographic entry.

Attributes: *Common.attributes*

Content: *Inline.model*

Used by: *Bibentry.class*

Element **bib-date** Date of a bibliographic entry.

Attributes: *Common.attributes*

Attribute **role** = (`publication` | `copyright` | *text*)
characterizes what happened on the given date

Content: *Inline.model*

Used by: *Bibentry.class*

Element **bib-publisher** Publisher of a bibliographic entry.

Attributes: *Common.attributes*

Content: *Inline.model*

Used by: *Bibentry.class*

Element **bib-organization** Organization responsible for a bibliographic entry.

Attributes: *Common.attributes*

Content: *Inline.model*

Used by: *Bibentry.class*

Element **bib-place** Location of publisher or event

Attributes: *Common.attributes*

Content: *Inline.model*

Used by: *Bibentry.class*

Element **bib-related** A Related bibliographic object, such as the book or journal that the current item is related to.

Attributes: *Common.attributes*

Attribute **type** = *bibentry.type*
The type of this related entry.

Attribute **role** = (`host` | `event` | `original` | *text*)
How this object relates to the containing object. Particularly important is `host` which indicates that the outer object is a part of this object.

Attribute **bibrefs** = *text*
If the bibrefs attribute is given, it is the key of another object in the bibliography, and this element should be empty; otherwise the object should be described by the content of the element.

***Content*:** *Bibentry.class**

***Used by*:** *Bibentry.class*

Element **bib-part** Describes how the current object is related to a related (bib-related) object, in particular page, part, volume numbers and similar.

***Attributes*:** *Common.attributes*

Attribute **role** = (`pages` | `part` | `volume` | `issue` | `number` | `chapter` | `section` | `paragraph` | *text*)
indicates how the value partitions the containing object.

***Content*:** *Inline.model*

***Used by*:** *Bibentry.class*

Element **bib-edition** Edition of a bibliographic entry.

***Attributes*:** *Common.attributes*

***Content*:** *Inline.model*

***Used by*:** *Bibentry.class*

Element **bib-status** Status of a bibliographic entry.

***Attributes*:** *Common.attributes*

***Content*:** *Inline.model*

***Used by*:** *Bibentry.class*

Element **bib-identifier** Some form of document identfier. The content is descriptive.

***Attributes*:** *Common.attributes*, *Refable.attributes*

Attribute **scheme** = (`doi` | `issn` | `isbn` | `mr` | *text*)
indicates what sort of identifier it is; it is open-ended for extensibility.

Attribute **id** = *text*
the identifier.

***Content*:** *Inline.model*

***Used by*:** *Bibentry.class*

Element **bib-review** Review of a bibliographic entry. The content is descriptive.

***Attributes*:** *Common.attributes*, *Refable.attributes*

Attribute **scheme** = (`doi` | `issn` | `isbn` | `mr` | *text*)
indicates what sort of identifier it is; it is open-ended for extensibility.

Attribute **id** = *text*
the identifier.

***Content*:** *Inline.model*

Used by: *Bibentry.class*

Element **bib-links** Links to other things like preprints, source code, etc.

Attributes: *Common.attributes*

Content: *Inline.model*

Used by: *Bibentry.class*

Element **bib-language** Language of a bibliographic entry.

Attributes: *Common.attributes*

Content: *Inline.model*

Used by: *Bibentry.class*

Element **bib-url** A URL for a bibliographic entry. The content is descriptive

Attributes: *Common.attributes*, *Refable.attributes*

Content: *Inline.model*

Used by: *Bibentry.class*

Element **bib-extract** An extract from the referenced object.

Attributes: *Common.attributes*

Attribute **role** = (`keywords` | `abstract` | `contents` | *text*)
Classify what kind of extract

Content: *Inline.model*

Used by: *Bibentry.class*

Element **bib-note** Notes about a bibliographic entry.

Attributes: *Common.attributes*

Attribute **role** = (`annotation` | `publication` | *text*)
Classify the kind of note

Content: *Inline.model*

Used by: *Bibentry.class*

Element **bib-data** Random data, not necessarily even text. (future questions: should model be text or ANY? maybe should have encoding attribute?).

Attributes: *Common.attributes*

Attribute **role** = *text*
Classify the relationship of the data to the entry.

Attribute **type** = *text*
Classify the type of the data.

Content: *Inline.model*

***Used by*:** *Bibentry.class*

***Pattern* Bibentry.class *Content*:** (bib-name | bib-title | bib-subtitle | bib-key | bib-type | bib-date | bib-publisher | bib-organization | bib-place | bib-part | bib-related | bib-edition | bib-status | bib-language | bib-url | bib-note | bib-extract | bib-identifier | bib-review | bib-links | bib-data)

***Used by*:** bib-related, bibentry

Appendix E

Error Codes

Warning and Error messages are printed to STDERR during the execution of `latexml` and `latexmlpost`. As with TEX, it is not always possible to indicate where the real underying mistake originated; sometimes it is only realized later on that some problem has occurred, such as a missing brace. Moreover, whereas error messages from TEX may be safely assumed to indicate errors with the source document, with LATEXML they may also indicate LATEXML's inability to figure out what you wanted, or simply bugs in LATEXML or the librarys it uses.

Warnings are generally informative that the generated result may not be as good as it can be, but is most likely properly formed. A typical warning is that the math parser failed to recognize an expression.

Errors generally indicate a more serious problem that is likely to lead to a malformed result. A typical error would be an undefined control sequence. Generally, processing continues so that you can (hopefully) solve all errors at once.

Fatals are errors so serious as to make it unlikely that processing can continue; the system is likely to be out-of-sync, for example not knowing from which point in the input to continue reading. A fatal error is also generated when too many (typically 100 regular errors have been encountered.

Warning and Error messages are slightly structured to allow unattended processing of documents to classify the degree of success in processing. A typical message satisfies the following regular expression:

```
severity:category:object summary
    source locator
    description
    ...
    stack trace
```

the second and following lines are indented using a tab.

severity One of `Info`, `Warn`, `Error` or `Fatal`, indicating the severity of the problem;

category classifies the error or warning into an open-ended set of categories indicating whether something was `expected`, or `undefined`;

object indicates the offending object; what filename was missing, or which token was undefined;

summary gives a brief readable summary of the condition;

source locator indicates where in the source document the error occurred;

description gives one or more lines of more detailed information;

stack trace optionally gives a brief or long trace of the current execution stack.

The type is followed by one or more keywords separated by colons, then a space, and a human readable error message. Generally, this line is followed by one or more lines describing where in the source document the error occured (or was detected). For example:

```
Error:undefined:\foo The control sequence \foo is undefined.
```

Some of the more common keywords following the message type are listed below, where we assume that *arg* is the second keyword (if any).

The following errors are generally due to malformed TEX input, incomplete LATEXML bindings, or bindings that do not properly account for the way TEX, or the macros, are actually used.

`undefined` : The operation indicated by *arg*, typically a control sequence or other operation, is undefined.

`ignore` : Indicates that *arg* is being ignored; typically it is a duplicated definition, or a definition of something that cannot be redefined.

`expected` : A particular token, or other type of data object, indicated by *arg*, was expected in the input but was missing.

`unexpected` : *arg* was not expected to appear in the input.

`not_parsed` : A mathematical formula could not be successfully parsed.

`missing_file` : the file *arg* could not be found.

`latex` : An error or message generated from LATEX code. and the corresponding LATEXML code should be updated.

`too_many_errors` : Too many non-fatal errors were encountered, causing a Fatal error and program termination.

The following errors are more likely to be due to programming errors in the LATEXML core, or in binding files, or in the document model.

`misdefined` : The operation indicated by *arg*, typically a control sequence or other operation, has not been defined properly.

deprecated : Indicates that *arg* is a deprecated usage.

malformed : The document is malformed, or will be made so by insert *arg* into it.

I/O : some problem with input/output of the file *arg*, such as it not being readable. The exact error is reported in the additional details.

perl : A perl-level error or warning, not specifically recognized by LaTeXML, was encountered. *arg* will typically `die`, `interrupt` or `warn`.

internal : Something unexpected happened; most likey an internal coding error within LaTeXML.

Appendix F

CSS Classes

When the target format is in the HTML family (XHTML, HTML or HTML5), LaTeXML adds various classes to the generated html elements. This provides a trail back to the originating markup, and leverage to apply CSS styling to the results. Recall that the class attribute is a space-separated list of class names. This appendix describes the class names used.

The basic strategy is the following:

`ltx_`*element* with *element* being the LaTeXML element name that generated the html element. These elements reflect the original TeX/LaTeX markup, but are not identical. See Appendix D for details.

`ltx_font_`*font* where *font* can indicate any of the font characteristics:

family : `serif`, `sansserif`, `typewriter`, `caligraphic`, `fraktur`, `script`;

series : `bold`, `medium`;

shape : `upright`, `italic`, `slanted`, `smallcaps`;

These sets are open-ended.

`ltx_align_`*alignment* where *alignment* indicates the alignment of the contents within the element.

horizontally : `left`, `right`, `center`, `justify`;

vertically : `top`, `bottom`, `baseline`, `middle`.

`ltx_border_`*edges* indicates single or double borders on an element with *edges* being: `t`, `r`, `b`, `l`, `tt`, `rr`, `bb`, `ll`; these are typically used for table cells.

`ltx_role_`*role* reflects the distinct uses a particular LaTeXML elements serve which is indicated by the role attribute. Examples include creator, for 'document creators', where the role may be `author`, `editor`, `translator` or others. Thus, depending on your purposes and the expected markup, you might choose

to write CSS rules for `ltx_creator` or `ltx_role_author`. Similarly, quote is stretched to accomodate `translation` or `verse`.

`ltx_title_`*section* marks the titles of various sectional units. For example, a chapter's title will have two classes: `ltx_title` and `ltx_title_chapter`.

`ltx_theorem_`*type* marks various types of 'theorem-like' objects, where the *type* is whatever was used in `\newtheorem`.

`ltx_float_`*type* marks various types of floating objects, such as might be defined using the `float` package using `\newfloat`.

`ltx_lst_`*role* reflects the various roles of items within listings, such as those created using the `listings` package (whose containing element would have class `ltx_lstlisting`). Such classes include: `ltx_lst_language_`*lang*, `ltx_lst_`*keywordclass*, `ltx_lxt_line`, `ltx_lst_linenum`.

`ltx_bib_`*item* indicates various items in bibliographys, typically generated via BibTeX; the items include `key`, `number`, `type`, `author`, `editor`, `year`, `title`, `author-year`, `edition`, `series`, `part`, `journal`, `volume`, `number`, `status`, `pages`, `language`, `publisher`, `place`, `status`, `crossref`, `external`, `cited` and *others*.

`ltx_toclist_`*type*, `ltx_tocentry_`*type* reflects the levels of Table of Contents lists: they carry the `ltx_toclist` class, from the element used to represent them, and also `ltx_toclist_`*section* naming the sectional unit for which this list applies to assist in styling. A nested TOC for a chapter might thus have `ul`'s carrying `ltx_toclist_chapter` and `ltx_toclist_section`. Additionally, `ltx_toc_compact` and `ltx_toc_verycompact` can be added to style compact and very compact styles (eg single line). Note that the generated `li` items will have class `ltx_tocentry` and `ltx_tocentry_`*type*, for the type of the entry.

`ltx_ref_`*item* hypertext links, whether within or across documents, whether created from `\ref` or `\href`, will get `ltx_ref` and, sometimes, extra classes applied. For example, a reference that ends up pointing to the current page is marked with `ltx_ref_self`. Cross-referencing material used to fill-in the contents of the reference is marked: a reference number gets `ltx_ref_tag`; a title `ltx_ref_title`.

`ltx_note_`*part* reflects the separate parts of notes; Note that the kind of note is generally reflected in the role attribute, such as `footnote`, `endnote`, etc. The parts are separated to facilitate formatting, hover effects, etc: `outer` contains the whole; `mark` for the mark, if any; `content` the actual contents of the note. `type` is for an extra span indicating the type of note if it is unusual.

`ltx_page_`*item* reflects page layout components created during the XSLT; *item*s include: `main`, `content`, `header`, `footer`, `navbar logo`, `columns`, `column1`, `column2`.

`ltx_eqn_`*item* reflects different parts related to equation formatting: `pad` reflects padding to align equations on the page; `eqnarray` and `lefteqn` arise from LaTeX's `eqnarray` environment; `gather` and `align` arise from AMS environments; `intertext` arises from text injected between aligned equations.

Explicit use of the `addClass(class)` function or the `\lxAddClass{class}` macro from the `latexml` package will add the given class as is, without any additional `ltx_` prefix.

Two oddball items that may get refactored away are: `ltx_phantom` and `ltx_centering`. The latter seems slightly distinct from `ltx_align_center`.

Index