LaTeXML *The Manual*

A LaTeX to XML Converter;
0.7.0

Bruce R. Miller

November 7, 2010

Contents

Chapter 1

Introduction

For many, LaTeX is the prefered 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`http://dlmf.nist.gov`. Faced with the need to perform this conversion and the lack of suitable tools to perform it, the DLMF project proceeded to develop thier own tool, LaTeXML, for this purpose.

Design Goals The idealistic goals of LaTeXML are:

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

As these goals are not entirely practical, or 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 the eye of the beholder. 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, rather than pretend to provide a "turn-key" solution, we expect that document-specific declarations or tuning to be necessary to faithfully convert documents. 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. An overview of outstanding issues and planned future improvements are given in Chapter 6.

Finally, the Appendices give detailed documentation the system components: Appendix A describes the command-line programs provided by the system; Appendices C and D describes the core and utility Perl modules comprising the system, while Appendix E describes the postprocessing modules; Appendix F describes the XML schema used by LaTeXML; finally, Appendix G gives an overview of the warning and error messages that LaTeXML may generate.

If all else fails, you can consult the source code, or the author.

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.

The usage of these commands can be as simple as

```
latexml doc.tex | latexmpost --dest=doc.xhtml
```

to convert a single document into XHTML, or as complicated as

```
latexml --dest=1.xml ch1
latexml --dest=2.xml ch2
            ...
latexml --dest=b.xml b
latexml --dest=B.xml B.bib
latexmlpost --prescan --db=my.db --bib=B.xml --dest=1.xhtml 1
latexmlpost --prescan --db=my.db --bib=B.xml --dest=2.xhtml 2
            ...
latexmlpost --prescan --db=my.db --bib=B.xml --dest=b.xhtml b
latexmlpost --noscan --db=my.db --bib=B.xml --dest=1.xhtml 1
latexmlpost --noscan --db=my.db --bib=B.xml --dest=2.xhtml 2
            ...
latexmlpost --noscan --db=my.db --bib=B.xml --dest=b.xhtml b
```

to convert a whole set of documents, including a bibliography, into a complete interconnected site.

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 XMLThe 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.

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.

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 when it is converted to XML; the selection of entries is done during post-processing. 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: `\documentclass{article}` or `\usepackage{graphicx}` will load the bindings `article.cls.ltxml` or `graphicx.sty.ltxml`, respectively; the obsolete `\documentstyle{article}` directive 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*`.xhtml` *doc*

or similarly with `--destination=`*doc*`.html`, 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 is determined by the file extension of `--destination`, or by the option

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

html both math and graphics are converted to png images; the stylesheet `LaTeXML-html.xslt` is used.

xhtml math is converted to Presentation MathML, other graphics are converted to images; the stylesheet `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 E.

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 Bibilographic data from BibTeX can be provided with the option

```
--bibliography=bibfile.xml
```

The bibliography would have typically been produced by running

```
latexml --dest=bibfile.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
```

(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.

More than one of these conversions can be requested, and each will be included in the output document. However, the option

```
--parallelmath
```

can be used to generate parallel MathML markup, provided the first conversion is either `--pmml` or `--cmml`.

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
```

Stylesheet If you wish to provide your own XSLT or CSS stylesheets, the options

```
--stylesheet=stylesheet.xsl
--css=stylesheet.css
```

can be used. The `--css` option can be repeated to include multiple stylesheets; for example, the distribution provides several in addition to the `core.css` stylesheet which is included by default.

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

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

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

`amsart.css` A style appropriate for many journal articles.

To develop such stylesheets, a knowledge of the LATEXML document type is necessary; See Appendix F.

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, along with separate bibliography, index and any appendices. (See also `--splitxpath` 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.xhtml 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.xhtml i
```

which skips the unnecessary scanning step.

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`.

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 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 `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 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.

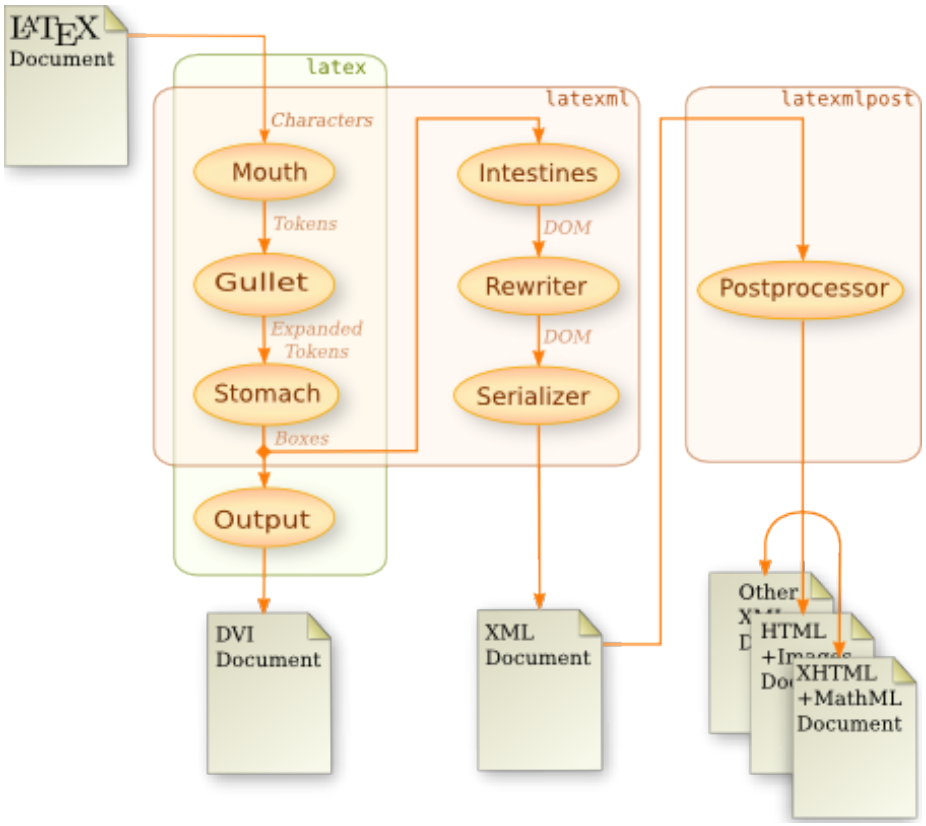

Figure 3.1: Flow of data through LaTeXML's digestive tract.

3.1.1 Digestion

Digestion is carried out primarily in a *pull* mode: The `Stomach` pulls expanded `Token`s from the `Gullet`, which itself pulls `Token`s from the `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 `Expandable`s and for parsing sequences of tokens into common core datatypes (`Number`, `Dimension`, etc.). See 4.1.1 for how to define macros and affect expansion.

The `Stomach` then digests these tokens by executing `Primitive` control sequences for side effect or converting material into `List`s of `Box`es and `Whatsit`s Normally, textual tokens are converted to `Box`es. The main (intentional) deviation of LaTeXML's digestion from that of TeX is the introduction of a new type of definition, a `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.

3.1.2 Construction

Given the List of Boxes and Whatsits, we proceed to constructing an XML document. This consists of creating an 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 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 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.

3.1.3 Rewriting

Once the basic document is constructed, 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.

3.1.4 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.

3.1.5 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 `--splitxpath` 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."

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. 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 `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;

# 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

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]); });
```

Functions that are useful for dealing with `Token`s 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 `Tokens` object which represents a list of `Token`s.
- **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=>'tiny'});
```

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.

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 `Tokens`), returning a list of `Box`es and `Whatsit`s.
- **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`)

The `$replacement` for a constructor describes the XML to be generated during the construction phase. It can either be a string representing the XML as a pattern (described below), or a subroutine `CODE($document,$arg1,...%props)` receiving the arguments and properties from the `Whatsit`; it would invoke the methods of `Document` to construct the desired XML. The pattern as illustrated above, simply represents a serialization of the desired XML. In addition to literal replacement, the following may appear:

- *`#1,#2,...#name`* inserts the construction of the argument or property in the XML;

- `&function($a,$b,...)` invokes the named function on the given arguments and inserts its value in place;

- `?COND(pattern)` or `?COND(ifpattern)(elsepattern)` conditionally inserts the patterns depending on the result of the conditional. `COND` would typically be testing the presence of an argument, *`#1`*, or property *`#name`* or invoking a function;

- ^ 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.

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($document,$whatit)` 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(`$prototypte,$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) 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 $n-1 nodes are removed.
- **DefRewrite**(%spec), **DefMathRewrite**(%spec) defines document rewrite rules. These specifications describe what document nodes match:
  - label=>$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 `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` and `LaTeXML-html.xsl`, found at *installationdir*`/LaTeXML/style/` — they're short, consisting mainly of sequence of `xsl:include` — and adding your own rules, or including your own modules. The two stylesheets differ primarily in their use of namespaces and handling of math.

Naturally, this requires a familiarity with LaTeXML's schema (see F), as well as XSLT and XHTML. See the other stylesheet modules in the same directory as the base stylesheet for guidance.

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.

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 `core.css`. See the directory *installationdir*`/LaTeXML/style/` 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 `<div class='section'>`, while the latter becomes `<h2 class='section-title'>` (for example, the h-level may vary with the document structure),

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

The role attempts to capture the syntactic nature of each item. This is used primarily to drive the parsing; the grammar rules are keyed on the role, rather than content, of the nodes. The role is also used to drive the conversion to presentation markup, especially Presentation MathML, and in fact some values of role are only used that way, never appearing explicitly in the grammar.

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.

ATOM a general atomic subexpression.

ID a variable-like token, whether scalar or otherwise.

PUNCT punctuation.

APPLYOP an explicit infix application operator (high precedence).

RELOP a relational operator, loosely binding.

ARROW an arrow operator (with little semantic significance). treated equivalently to RELOP.

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

ADDOP an addition operator, precedence between relational and multiplicative operators.

MULOP a multiplicative operator, high precedence.

SUPOP An operator appearing in a superscript, such as a collection of primes.

OPEN an open delimiter.

CLOSE a close delimiter.

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

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 a summation/union operator.

INTOP an integral operator.

LIMITOP a limiting operator.

DIFFOP a differential operator.

BIGOP a general operator, but lower precedence, such as a P preceding an integral to denote the principal value. Note that `SUMOP`, `INTOP`, `LIMITOP`, `DIFFOP` and `BIGOP` 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?

VERTBAR

FUNCTION a function which (may) apply to following arguments with higher precedence than addition and multiplication, or parenthesized arguments.

NUMBER a number.

POSTSUPERSCRIPT the usual superscript, where the script is treated as an argument, but the base will be determined by parsing. Note that this is not necessarily assumed to be a power. Very high precedence.

POSTSUBSCRIPT Similar to `POSTSUPERSCRIPT` for subscripts.

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

FLOATINGSUBSCRIPT Similar to `POSTSUPERSCRIPT` for subscripts.

POSTFIX for a postfix operator

UNKNOWN an unknown expression. This is the default for token elements, and generates a warning if the unknown seems to be used as a function.

The following roles are not used in the grammar, but are used to capture the presentation style:

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

Chapter 6

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, Deyan Ginev and Catalin David for testing help and for implementing additional packages.

Appendix A

Command Documentation

`latexml`

Transforms a TeX/LaTeX file into XML.

Synopsis

latexml [options] *texfile*

```
Options:
--destination=file specifies destination file (default stdout).
--output=file      [obsolete synonym for --destination]
--preload=module   requests loading of an optional module;
                   can be repeated
--includestyles    allows latexml to load raw *.sty file;
                   by default it avoids this.
--path=dir         adds dir to the paths searched for files,
                   modules, etc;
--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 the file 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 extention 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 specificly 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.

--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. 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.

--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 specifies directory of the original
                        source TeX file.
--validate, --novalidate Enables (the default) or disables
                        validation of the source xml.
--format=html|xhtml|xml requests the output format.
--destination=file      specifies 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 and crossrefs.
--nonumbersections      disables the above
--stylesheet=xslfile    requests the XSL transform using the
                        given xslfile as stylesheet.
--css=cssfile           adds a css stylesheet to html/xhtml
                        (can be repeated)
--nodefaultcss          disables the default css stylesheet
--split                 requests splitting each document
--nosplit               disables the above (default)
--splitat               specifies level to split the document
--splitpath=xpath       specifies xpath expression for splitting
                        (default is section-like, if splitting)
--splitnaming=(id|idrelative|label|labelrelative) specifies how
                        to name split files (def. idrelative).
--scan                  scans documents to extract ids, labels,
                        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).
--index                 requests filling in the index (default)
--noindex               disables the above
--splitindex            Splits the index into pages per initial.
```

```
--nosplitindex          disables the above (default)
--permutedindex         permutes index phrases in the index
--nopermutedindex       disables the above (default)
--bibliography=file     specifies 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         specifies file to store crossreferences
--sitedirectory=dir     specifies the base directory of the site
--mathimages            converts math to images
                        (default for html format)
--nomathimages          disables the above
--mathimagemagnification=mag specifies magnification factor
--presentationmathml    converts math to Presentation MathML
                        (default for xhtml format)
--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
--parallelmath          requests parallel math markup for MathML
                        (default when multiple math formats)
--noparallelmath        disables the above
--keepXMath             preserves the intermediate XMath
                        representation (default is to remove)
--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|xhtml|xml)

Specifies the output format for post processing. `html` format converts the material to html 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. In both cases, any graphics will be converted to web-friendly formats and/or copied to the destination directory. By default, `xml`, the output is left in LaTeXML's xml, but the math is parsed and converted to presentation MathML. For html 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 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 (html or xhtml) will be used.

--css=*cssfile*

Adds *cssfile* as a css stylesheet to be used in the transformed html/xhtml. Multiple stylesheets can be used; they are included in the html in the order given, following the default `core.css` (but see `--nodefaultcss`). 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.

--nodefaultcss

Disables the inclusion of the default `core.css` stylesheet.

--icon=*iconfile*

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

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 unneccessary 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.

--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 file generated from a BibTeX file and used to fill in a bibliography element. Hand-written bibliographies placed in a `thebibliography` environment do not need this processing. Enabling this 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. Conversion is the default for html format.

--mathimagemagnification=*factor*

Specifies the magnification used for math images, 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 format.

--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, --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.

--parallelmath, --noparallelmath

Requests or disables parallel math markup. Parallel markup is the default for xhtml formats when multiple math formats are requested.

This method uses the MathML `semantics` element with additional formats appearing as `annotation`'s. The first math format requested must be either Presentation or Content MathML; additional formats may be MathML or OpenMath.

If this option is disabled and multiple formats are requested, the representations are simply stored as separate children of the `Math` element.

--keepXMath

By default, when any of the MathML or OpenMath conversions are used, the intermediate math representation will be removed; this option preserves it.

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. The default settings of the Graphics module search for pdf, gif, jpg or jpeg files, which it copies unchanged, or ps, eps and pdf files, which are converted to png.

When this option is used, it overrides 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.

--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

latexmlmath

Transforms a TeX/LaTeX math expression into various formats.

Synopsis

latexmlmath [options] *texmath*

```
Options:
--mathimage=file           converts to 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
--XMath=file               convert to 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 be repeated)
--verbose                  increases verbosity (can be repeated)
--strict                   be more strict about errors.
--documentid=id            assign an id to the document root.
--debug=package            enables debugging output for the
                           named package
--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 "\\frac{-b\\pm\\sqrt{b^2-4ac}}{2a}" |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.

`--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.

--XMath=*file*

Requests convertion 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 specificly 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.

--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: aa, aastex, amsart, amsbook, amsproc, article, book, elsart, emulateapj, gen-j-l, gen-m-l, gen-p-l, iopart, llncs, mn, mn2e, report, revtex, revtex4, svjour, svmult

packages: a4, a4wide, aasms, aaspp, aastex, acronym, ae, alltt, amsbsy, amscd, amsfonts, amsmath, amsopn, amsppt, amsrefs, amssymb, amstex, amstext, amsthm, amsxtra, array, avant, bbm, beton, bm, bookman, caption, ccfonts, chancery, charter, cite, cmbright, color, comment, concmath, courier, dcolumn, elsart, emulateapj, enumerate, epsf, epsfig, eucal, eufrak, euler, eulervm, eurosym, euscript, exscale, fixltx2e, float, fontenc, fourier, graphics, graphicx, helvet, hhline, html, hyperref, ifpdf, inputenc, iopams, keyval, latexml, latexsym, listings, longtable, lscape, luximono, makeidx, mathpazo, mathpple, mathptm, mathptmx, multicol, multido, multirow, natbib, newcent, palatino, paralist, pifont, psfig, pspicture, pst-grad, pst-node, pstricks, pxfonts, revsymb, revtex, revtex4, rsfs, subfigure, supertabular, tabularx, textcomp, times, tocbibind, txfonts, upgreek, upref, url, utopia, wrapfig, xspace, yfonts

Appendix C

Core Module Documentation

LaTeXML

Transforms TeX into XML.

Synopsis

```
use LaTeXML;
my $latexml = LaTeXML->new();
$latexml->convertAndWrite("adocument");
```

But also see the convenient command line script `latexml` which suffices for most purposes.

Description

Methods

`my $latexml = LaTeXML->new(%options);`

Creates a new LaTeXML object for transforming TeX files into XML.

```
verbosity  : Controls verbosity; higher is more verbose,
             smaller is quieter. 0 is the default.
strict     : If true, undefined control sequences and
             invalid document constructs give fatal
             errors, instead of warnings.
includeComments : If false, comments will be excluded
             from the result document.
preload    : an array of modules to preload
searchpath : an array of paths to be searched for Packages
             and style files.
```

(these generally set config variables in the LaTeXML::State object)

$latexml->convertAndWriteFile($file);

Reads the TeX file $file.tex, digests and converts it to XML, and saves it in $file.xml.

$doc = $latexml->convertFile($file);

Reads the TeX file $file, digests and converts it to XML and returns the resulting XML::LibXML::Document.

$doc = $latexml->convertString($string);

Digests $string, presumably containing TeX markup, converts it to XML and returns the XML::LibXML::Document.

$latexml->writeDOM($doc,$name);

Writes the XML document to $name.xml.

$box = $latexml->digestFile($file);

Reads the TeX file $file, and digests it returning the LaTeXML::Box representation.

$box = $latexml->digestString($string);

Digests $string, which presumably contains TeX markup, returning the LaTeXML::Box representation.

$doc = $latexml->convertDocument($digested);

Converts $digested (the LaTeXML::Box reprentation) into XML, returning the XML::LibXML::Document.

Customization

In the simplest case, LaTeXML will understand your source file and convert it automatically. With more complicated (realistic) documents, you will likely need to make document specific declarations for it to understand local macros, your mathematical notations, and so forth. Before processing a file *doc.tex*, LaTeXML reads the file *doc.latexml*, if present. Likewise, the LaTeXML implementation of a TeX style file, say *style.sty* is provided by a file *style.ltxml*.

See LaTeXML::Package for documentation of these customization and implementation files.

See also

See latexml for a simple command line script.

See LaTeXML::Package for documentation of these customization and implementation files.

For cases when the high-level declarations described in LaTeXML::Package are not enough, or for understanding more of LaTeXML's internals, see

LaTeXML::State

maintains the current state of processing, bindings or variables, definitions, etc.

LaTeXML::Token, LaTeXML::Mouth and LaTeXML::Gullet

deal with tokens, tokenization of strings and files, and basic TeX sequences such as arguments, dimensions and so forth.

LaTeXML::Box and LaTeXML::Stomach

deal with digestion of tokens into boxes.

LaTeXML::Document, LaTeXML::Model, LaTeXML::Rewrite

dealing with conversion of the digested boxes into XML.

LaTeXML::Definition and LaTeXML::Parameters

representation of LaTeX macros, primitives, registers and constructors.

LaTeXML::MathParser

the math parser.

LaTeXML::Global, LaTeXML::Error, LaTeXML::Object

other random modules.

LaTeXML::Object

Abstract base class for most LaTeXML objects.

Description

`LaTeXML::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.

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 comparision, but the default implementation just compares for object identity.

`$boole = $object->isaToken;`

Returns whether `$object` is an `LaTeXML::Token`.

`$boole = $object->isaBox;`

Returns whether `$object` is an `LaTeXML::Box`.

`$boole = $object->isaDefinition;`

Returns whether `$object` is an `LaTeXML::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::Definition

Control sequence definitions.

Description

These represent the various executables corresponding to control sequences. See LaTeXML::Package for the most convenient means to create them.

LaTeXML::Expandable

represents macros and other expandable control sequences that are carried out in the Gullet during expansion. The results of invoking an LaTeXML::Expandable should be a list of LaTeXML::Tokens.

LaTeXML::Primitive

represents 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::Stomach, changing the LaTeXML::State. The results of invoking a LaTeXML::Primitive, if any, should be a list of digested items (LaTeXML::Box, LaTeXML::List or LaTeXML::Whatsit).

LaTeXML::Register

is set up as a speciallized primitive with a getter and setter to access and store values in the Stomach.

LaTeXML::CharDef

represents a further specialized Register for chardef.

LaTeXML::Constructor

represents control sequences that contribute arbitrary XML fragments to the document tree. During digestion, a LaTeXML::Constuctor records the arguments used in the invokation to produce a LaTeXML::Whatsit. The resulting LaTeXML::Whatsit (usually) generates an XML document fragment when absorbed by an instance of LaTeXML::Document. Additionally, a LaTeXML::Constructor may have beforeDigest and afterDigest daemons defined which are executed for side effect, or for adding additional boxes to the output.

More documentation needed, but see LaTeXML::Package for the main user access to these.

Methods in general

$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::Tokens.

$parameters = $defn->getParameters;

Return the LaTeXML::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::Tokens. For primitives, it is carried out in the Stomach, and returns a list of LaTeXML::Boxes. For a constructor, it is also carried out by the Stomach, and returns a LaTeXML::Whatsit. That whatsit will be responsible for constructing the XML document fragment, when the LaTeXML::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.

More about Primitives

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::Box, etc) that will be contributed to the current list.

More about Registers

Registers generally store some value in the current LaTeXML::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:

$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,@args);

Assign a value to the register, by invoking it's `setter` function.

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::ConstructorCompiler`. Like primitives, constructors may have `beforeDigest` and `afterDigest`.

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::State by an instance of LaTeXML during processing.

Tokens

$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, CC_NOTEXPANDED.
```

[The last 2 are (apparent) extensions, with catcodes 16 and 17, respectively].

$token = Token($string,$cc);

Creates a LaTeXML::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 = Tokens(@token);

Creates a LaTeXML::Tokens from a list of LaTeXML::Token's

$tokens = Tokenize($string);

Tokenizes the $string according to the standard cattable, returning a LaTeXML::Tokens.

$tokens = TokenizeInternal($string);

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

@tokens = Explode($string);

Returns a list of the tokens corresponding to the characters in $string.

$tokens = Revert($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.

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

Desable disable most TeX catcodes.

Boxes, etc.

$box = Box($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 (See UnTeX). If $font or $locator are undef, they are obtained from the currently active LaTeXML::State. Note that $string can be undef which contributes nothing to the generated document, but does record the TeX code (in $tokens).

Numbers, etc.

$number = Number($num);

Creates a Number object representing $num.

$number = Float($num);

Creates a floating point object representing $num; This is not part of TeX, but useful.

$dimension = Dimension($dim);

Creates a Dimension object. $num can be a string with the number and units (with any of the usual TeX recognized units), or just a number standing for scaled points (sp).

$mudimension = MuDimension($dim);

Creates a MuDimension object; similar to Dimension.

$glue = Glue($gluespec);

$glue = Glue($sp,$plus,$pfill,$minus,$mfill);

Creates a Glue object. $gluespec 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: $pfill and $mfill are 0 (when the $plus or $minus part is in sp) or 1,2,3 for fil, fill or filll.

$glue = MuGlue($gluespec);

$glue = MuGlue($sp,$plus,$pfill,$minus,$mfill);

Creates a MuGlue object, similar to Glue.

$pair = Pair($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.

$pair = PairList(@pairs);

Creates an object representing a list of pairs of numbers; Not a part of TeX, but useful for graphical objects.

Error Reporting

Fatal($message);

Signals an fatal error, printing $message along with some context. In verbose mode a stack trace is printed.

Error($message);

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($message);

Prints a warning message along with a short indicator of the input context, unless verbosity is quiet.

NoteProgress($message);

Prints $message unless the verbosity level below 0.

Generic functions

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.

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.

Equals($a,$b);

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.

Revert($object);

Converts `$object` to a Tokens list containing the 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.

UnTeX($object);

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.

LaTeXML::Error

Internal Error reporting code.

Description

LaTeXML::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.

No user serviceable parts inside. No symbols are exported.

Functions

$string = LaTeXML::Error::generateMessage($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::Error::stacktrace;

Return a formatted string showing a trace of the stackframes up until this function was invoked.

@objects = LaTeXML::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.

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.

```
use LaTeXML::Package;
use strict;
#
# 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

To provide a LaTeXML-specific version of a LaTeX package `mypackage.sty` or class `myclass.cls` (so that eg. `\usepackage{mypackage}` works), you create the file `mypackage.sty.ltxml` or `myclass.cls.ltxml` and save it in the searchpath (current directory, or one of the directories given to the –path option, or possibly added to the variable SEARCHPATHS). Similarly, to provide document-specific customization for, say, `mydoc.tex`, you would create the file `mydoc.latexml` (typically in the same directory). However, in the first cases, `mypackage.sty.ltxml` are loaded *instead* of `mypackage.sty`, while a file like `mydoc.latexml` is loaded in *addition* to `mydoc.tex`. In either case, you'll `use LaTeXML::Package;` to import the various declarations and defining forms that allow you to specify what should be done with various control sequences, whether there is special treatment of certain document elements, and so forth. Using `LaTeXML::Package` also imports the functions and variables defined in `LaTeXML::Global`, so see that documentation as well.

Since LaTeXML attempts to mimic TeX, a familiarity with TeX's processing model is also helpful. Additionally, it is often useful, when implementing non-trivial behaviour, to think TeX-like.

Many of the following forms take code references as arguments or options. That is, either a reference to a defined sub, `\&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($token,..).

Control Sequences

Many of the following forms define the behaviour of control sequences. In TeX you'll typically only define macros. In LaTeXML, we're effectively redefining TeX itself, so we define macros as well as primitives, registers, constructors and environments. These define the behaviour of these commands when processed during the various phases of LaTeX's immitation of TeX's digestive tract.

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:* {Token}[][]{}', ...
```

Control Sequence Parameters The general syntax for parameter for a control sequence is something like

```
OpenDelim? Modifier? Type (: value (| value)* )? CloseDelim?
```

The enclosing delimiters, if any, are either {} or [], affect the way the argument is delimited. With {}, a regular TeX argument (token or sequence balanced by braces) is read before parsing according to the type (if needed). With [], a LaTeX optional argument is read, delimited by (non-nested) square brackets.

The modifier can be either `Optional` or `Skip`, allowing the argument to be optional. For `Skip`, no argument is contributed to the argument list.

The shorthands {} and [] default the type to `Plain` and reads a normal TeX argument or LaTeX default argument with no special parsing.

The predefined argument types 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` or `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.

`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.

Skip1Space, SkipSpaces

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

Control of Scoping Most defining commands accept an option to control how the definition is stored, `scope=>$scope`, where `$scope` can be c<'global'>for global definitions, `'local'`, to be stored in the current stack frame, or a string naming a *scope*. A 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 the section ends.

Macros

DefMacro($prototype,$string |$tokens |$code,%options);

Defines the macro expansion for `$prototype`; a macro control sequence that is expanded during macro expansion time (in the `LaTeXML::Gullet`). If a `$string` is supplied, it will be tokenized at definition time, and any macro arguments will be substituted for parameter indicators (eg #1) at expansion time; the result is used as the expansion of the control sequence. The only option, other than `scope`, is `isConditional` which should be true, for conditional control sequences (TeX uses these to keep track of conditional nesting when skipping to \else or \fi).

If defined by `$code`, the form is `CODE($gullet,@args)` and it must return a list of `LaTeXML::Token`'s.

DefMacroI($cs,$paramlist,$string |$tokens |$code,%options);

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

Primitives

`DefPrimitive($prototype,$replacement,%options);`

Define a primitive control sequence; a primitive is processed during digestion (in the `LaTeXML::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::State`.

The `$replacement` is either a string, used as the Boxes text content (the box gets the current font), or `CODE($stomach,@args)`, which is invoked at digestion time, probably for side-effect, but returning Boxes or Lists. `$replacement` may also be undef, which contributes nothing to the document, but does record the TeX code that created it.

DefPrimitive options are

mode=>(text|display_math| inline_math)

Changes to this mode during digestion.

bounded=>boolean

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

requireMath=>boolean,

forbidMath=>boolean

These specify whether the given constructor can only appear, or cannot appear, in math mode.

font=>{fontspec...}

Specifies the font to be set by this invocation. See `/"MergeFont(%style);"` 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.

beforeDigest=>CODE($stomach)

This option supplies a Daemon to be executed during digestion just before the main part of the primitive is executed. 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)

This option supplies a Daemon to be executed 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.

scope=>$scope

See `/"Control of Scoping"`.

isPrefix=>1

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

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

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

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

Defines a register with the given 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

specifies if it is not allowed to change this value.

getter=>CODE(@args) =item setter=>CODE($value,@args)

By default the 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.

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($prototype,$xmlpattern |$code,%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::Whatsit to represent the object. During absorbtion by the LaTeXML::Document, the Whatsit will generate the XML fragment according to the replacement $xmlpattern, or by executing CODE.

The $xmlpattern is simply a bit of XML as a string with certain substitutions to be made. The substitutions are of the following forms:

If code is supplied, the form is CODE($document,@args,%properties)

#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.

`?COND(pattern)` or `?COND(ifpattern)(elsepattern)`

Patterns can be conditionallized using this form. The `COND` is any of the above expressions, 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 constuctor *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

mode=>(text|display_math| inline_math)

Changes to this mode during digestion.

bounded=>boolean

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

requireMath=>boolean,

forbidMath=>boolean

These specify whether the given constructor can only appear, or cannot appear, in math mode.

font=>{fontspec...}

Specifies the font to be set by this invocation. See `/"MergeFont(%style);"` 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.

reversion=>$texstring or CODE($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.

properties=>{prop=>value,...} or CODE($stomach,#1,#2...)

This option supplies additional properties to be set on the generated Whatsit. In the first form, the values can be of any type, but (1) if it is a code references, it takes the same args ($stomach,#1,#2,...) and should return a value. and (2) if the value is a string, occurances of #1 (etc) are replaced by the corresponding argument. In the second form, the code should return a hash of properties.

beforeDigest=>CODE($stomach)

This option supplies a Daemon to be executed 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)

This option supplies a Daemon to be executed during digestion just after the Whatsit is created. 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 CODE to execute before constructing the XML (generated by $replacement).

afterConstruct=>CODE($document,$whatsit)

Supplies CODE to execute after constructing the XML.

captureBody=>boolean or 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.

alias=>$control_sequence

Provides a control sequence to be used when reverting Whatsit's back to Tokens, in cases where it isn't the command used in the `$prototype`.

nargs=>$nargs

This gives a number of args for cases where it can't be infered directly from the `$prototype` (eg. when more args are explictly read by Daemons).

scope=>$scope

See /"Control of Scoping".

DefConstructorI($cs,$paramlist,$xmlpattern |$code,%options);

Internal form of `DefConstructor` where the control sequence and parameter list have already been parsed; 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

These DefConstructor options also apply:

reversion, alias, beforeDigest, afterDigest, beforeConstruct, afterConstruct and scope.

Additionally, it accepts

style=>astyle

adds a style attribute to the object.

name=>aname

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!

font=>{fontspec}

Specifies the font to be used for when creating this object. See /"MergeFont(%style);".

scriptpos=>boolean

Controls whether any sub and super-scripts will be stacked over or under this object, or whether they will appear in the usual position.

WRONG: Redocument this!

operator_role=>grammatical_role

operator_scriptpos=>boolean

These two are similar to role and scriptpos, 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, so that changes to fonts, etc, are local. Providing <noggroup=1>>inhibits this.

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

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

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

Defines an Environment that generates a specific XML fragment. The $replacement is of the same form as that 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 daemon is run. Then the Whatsit representing the begin command (but ultimately the whole environment) is created and the afterDigestBegin daemon is run. Next, the body will be digested and collected until the balancing \end{env}. Then, any afterDigest daemon 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.

It shares options with DefConstructor:

```
mode, requireMath, forbidMath, properties, nargs,
font, beforeDigest, afterDigest, beforeConstruct,
afterConstruct and scope.
```

Additionally, afterDigestBegin is effectively an afterDigest for the \begin{env} control sequence.

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

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

Class and Packages

RequirePackage($package,%options);

Finds and loads a package implementation (usually *.sty.ltxml, unless raw is specified) for the required $package. The options are:

type=>type specifies the file type (default sty.

options=>[...] specifies a list of package options.

raw=>1 specifies that it is allowable to try to read a raw TeX style file.

LoadClass($class,%options);

Finds and loads a class definition (usually *.cls.ltxml). The only option is

options=>[...] specifies a list of class options.

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. The options are:

type=>type specifies the file type (default sty.

raw=>1 specifies that it is allowable to try to read a raw TeX style file.

DeclareOption($option,$code);

Declares an option for the current package or class. The $code can be a string or Tokens (which will be macro expanded), or can be a code reference which is treated as a primitive.

If a package or class wants to accomodate 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();

Processes the options that have been passed to the current package or class in a fashion similar to LaTeX. If the keyword inorder=>1 is given, the options are processed in the order they were used, like ProcessOptions*.

ExecuteOptions(@options);

Process the options given explicitly in @options.

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  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',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 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.

The recognized properties are:

autoOpen=>boolean

Specifies whether this $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.

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.

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.

`RelaxNGSchema($schemaname);`

Specifies the schema to use for determining document model. You can leave off the extension; it will look for `.rng`, and maybe eventually, `.rnc` once that is 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 (See 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 (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(CODE($document,@nodes));

CODE is called on each sequence of math nodes at a given level. If they should be replaced, return a list of `($n,$string,%attributes)` to replace the text content of the first node with `$string` content and add the given attributes. The next `$n-1` nodes are removed. If no replacement is called for, CODE should return undef.

Predefined Math Ligatures combine letter or digit Math Tokens (XMTok) into multicharacter symbols or numbers, depending on the font (non math italic).

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 seqeuence 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 =>$hash

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

replace =>$replacement

Interprets the $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.

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 therefor also be scoped). The recognized configuration parameters are:

```
VERBOSITY         : the level of verbosity for debugging
                    output, with 0 being default.
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($type,$name,@values);`

This is like `AssignValue`, but pushes values onto the end of the value, which should be a LIST reference. Scoping is not handled here (yet?), it simply pushes the value onto the last binding of `$name`.

`UnshiftValue($type,$name,@values);`

Similar to `PushValue`, but pushes a value onto the front of the values, which should be a LIST reference.

`$value = LookupCatcode($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.

Low-level Functions

CleanLabel($label,$prefix);

Cleans a $label of disallowed characters, prepending $prefix (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.

MergeFont(%style);

Set the current font by merging the font style attributes with the current font. The attributes and likely values (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.

`@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.

LaTeXML::Parameters

Formal parameters, including LaTeXML::Parameter.

Description

Provides a representation for the formal parameters of LaTeXML::Definitions:

LaTeXML::Parameter

represents an individual parameter.

Parameters Methods

$parameters = parseParameters($prototype,$for);

Parses a string for a sequence of parameter specifications. Each specification should be of the form

```
{}      reads a regular TeX argument, a sequence of
        tokens delimited by braces, or a single token.
{spec}  reads a regular TeX argument, then reparses it
        to match the given spec. The spec is parsed
        recursively, but usually should correspond to
        a single argument.
[spec]  reads an LaTeX-style optional argument. If the
        spec is of the form Default:stuff, then stuff
        would be the default value.
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.
Type:value, or Type:value1:value2...    These forms
        pass additional Tokens to the reader function.
OptionalType  Similar to Type, but it is not considered
        an error if the reader returns undef.
SkipType  Similar to OptionalType, but the value returned
        from the reader is ignored, and does not occupy a
        position in the arguments list.
```

@parameters = $parameters->getParameters;

Return the list of LaTeXML::Parameter contained in $parameters.

@tokens = $parameters->revertArguments(@args);

Return a list of LaTeXML::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.

LaTeXML::State

Stores the current state of processing.

Description

A LaTeXML::State object stores the current state of processing. It recording catcodes, variables values, definitions and so forth, as well as mimicing 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 thier 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::Stomach.

$STATE->popFrame;

Ends the current level of grouping. Note that this is lower level than \egroup; See LaTeXML::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 therefor also be scoped). The recognized configuration parameters are:

```
VERBOSITY          : the level of verbosity for debugging
                     output, with 0 being default.
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::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::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::Token

Representation of a token, and LaTeXML::Tokens, representing lists of tokens.

Description

This module defines Tokens (LaTeXML::Token, LaTeXML::Tokens) that get created during tokenization and expansion.

A LaTeXML::Token represents a TeX token which is a pair of a character or string and a category code. A LaTeXML::Tokens is a list of tokens (and also implements the API of a LaTeXML::Mouth so that tokens can be read from a list).

Common methods

The following methods apply to all objects.

@tokens = $object->unlist;

Return a list of the tokens making up this $object.

$string = $object->toString;

Return a string representing $object.

Token methods

The following methods are specific to LaTeXML::Token.

$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.

Tokens methods

The following methods are specific to LaTeXML::Tokens.

$tokenscopy = $tokens->clone;

Return a shallow copy of the $tokens. This is useful before reading from a LaTeXML::Tokens.

$token = $tokens->readToken;

Returns (and remove) the next token from $tokens. This is part of the public API of LaTeXML::Mouth so that a LaTeXML::Tokens can serve as a LaTeXML::Mouth.

LaTeXML::Box

Representations of digested objects.

Description

These represent various kinds of digested objects

LaTeXML::Box

represents text in a particular font;

LaTeXML::MathBox

represents a math token in a particular font;

LaTeXML::List

represents a sequence of digested things in text;

LaTeXML::MathList

represents a sequence of digested things in math;

LaTeXML::Whatsit

represents a digested object that can generate arbitrary elements in the XML Document.

Common Methods

All these classes extend `LaTeXML::Object` and so implement the `stringify` and `equals` operations.

$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 a string 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 appppropriate.

Box Methods

The following methods are specific to LaTeXML::Box and LaTeXML::MathBox.

$string = $box->getString;

Returns the string part of the $box.

Whatsit Methods

Note that the font is stored in the data properties under 'font'.

$defn = $whatsit->getDefinition;

Returns the LaTeXML::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::List or LaTeXML::MathList).

$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::Number

Representation of numbers, dimensions, skips and glue.

Description

This module defines various dimension and number-like data objects

LaTeXML::Number

represents numbers,

LaTeXML::Float

represents floating-point numbers,

LaTeXML::Dimension

represents dimensions,

LaTeXML::MuDimension

represents math dimensions,

LaTeXML::Glue

represents glue (skips),

LaTeXML::MuGlue

represents math glue,

LaTeXML::Pair

represents pairs of numbers

LaTeXML::Pairlist

represents list of pairs.

Common methods

The following methods apply to all objects.

@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.

Numerics methods

These methods apply to the various numeric objects

$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).

LaTeXML::Font

Representation of fonts, along with the specialization LaTeXML::MathFont.

Description

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.

LaTeXML::Font and LaTeXML::MathFont represent fonts (the latter, fonts in math-mode, obviously) in LaTeXML.

The attributes are

```
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
```

They are usually merged against the current font, attempting to mimic the, sometimes counter-intuitive, way that TeX does it, particularly for math

LaTeXML::MathFont

LaTeXML::MathFont supports $font−specialize($string);>for 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::Mouth

Tokenize the input.

Description

A LaTeXML::Mouth (and subclasses) is responsible for *tokenizing*, ie. converting plain text and strings into LaTeXML::Tokens according to the current category codes (catcodes) stored in the LaTeXML::State.

LaTeXML::FileMouth

specializes LaTeXML::Mouth to tokenize from a file.

LaTeXML::StyleMouth

further specializes LaTeXML::FileMouth for processing style files, setting the catcode for @ and ignoring comments.

LaTeXML::PerlMouth

is not really a Mouth in the above sense, but is used to definitions from perl modules with exensions .ltxml and .latexml.

Creating Mouths

$mouth = LaTeXML::Mouth->new($string);

Creates a new Mouth reading from $string.

$mouth = LaTeXML::FileMouth->new($pathname);

Creates a new FileMouth to read from the given file.

$mouth = LaTeXML::StyleMouth->new($pathname);

Creates a new StyleMouth to read from the given style file.

Methods

$token = $mouth->readToken;

Returns the next LaTeXML::Token from the source.

$boole = $mouth->hasMoreInput;

Returns whether there is more data to read.

$string = $mouth->getLocator($long);

Return a description of current position in the source, for reporting errors.

$tokens = $mouth->readTokens($until);

Reads tokens until one matches $until (comparing the character, but not catcode). This is useful for the \verb command.

$lines = $mouth->readRawLines($endline,$exact);

Reads raw (untokenized) lines from $mouth until a line matching $endline is found. If $exact is true, $endline is matched exactly, with no leading or trailing data (like in the c<comment>package). Otherwise, the match is done like with the c<verbatim>environment; any text preceding $endline is returned as the last line, and any characters after $endline remains in the mouth to be tokenized.

LaTeXML::Gullet

Expands expandable tokens and parses common token sequences.

Description

A LaTeXML::Gullet reads tokens (LaTeXML::Token) from a LaTeXML::Mouth. It is responsible for expanding macros and expandable control sequences, if the current definition associated with the token in the LaTeXML::State is an LaTeXML::Expandable definition. The LaTeXML::Gullet also provides a variety of methods for reading various types of input such as arguments, optional arguments, as well as for parsing LaTeXML::Number, LaTeXML::Dimension, etc, according to TeX's rules.

Managing Input

$gullet->input($name,$types,%options);

Input the file named $name; Searches for matching files in the current searchpath with an extension being one of $types (an array of strings). If the found file has a perl extension (pm, ltxml, or latexml), it will be executed (loaded). If the found file has a TeX extension (tex, sty, cls) it will be opened and latexml will prepare to read from it.

$gullet->openMouth($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 a string describing the current location in the input stream.

Low-level methods

$tokens = $gullet->expandTokens($tokens);

Return the LaTeXML::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.

`@tokens = $gullet->neutralizeTokens(@tokens);`

Another unusual method: Used for things like \edef and token registers, to inhibit further expansion of control sequences and proper spawning of register tokens.

`$token = $gullet->readToken;`

Return the next token from the input source, or undef if there is no more input.

`$token = $gullet->readXToken($toplevel);`

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.

`$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;`

Skip the next token from the input if it is a space.

`$tokens = $gullet->readBalanced;`

Read a sequence of tokens from the input until the balancing '}' (assuming the '{' has already been read). Returns a `LaTeXML::Tokens`.

`$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` (each are `LaTeXML::Tokens`) matches the input, or undef if none do.

`$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::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::Number according to TeX's rules of the various things that can be used as a numerical value.

$dimension = $gullet->readDimension;

Read a LaTeXML::Dimension according to TeX's rules of the various things that can be used as a dimension value.

$mudimension = $gullet->readMuDimension;

Read a LaTeXML::MuDimension according to TeX's rules of the various things that can be used as a mudimension value.

$glue = $gullet->readGlue;

Read a LaTeXML::Glue according to TeX's rules of the various things that can be used as a glue value.

$muglue = $gullet->readMuGlue;

Read a LaTeXML::MuGlue according to TeX's rules of the various things that can be used as a muglue value.

LaTeXML::Stomach

Digests tokens into boxes, lists, etc.

Description

LaTeXML::Stomach digests tokens read from a LaTeXML::Gullet (they will have already been expanded).

There are basically four cases when digesting a LaTeXML::Token:

A plain character

is simply converted to a LaTeXML::Box (or LaTeXML::MathBox in math mode), recording the current LaTeXML::Font.

A primitive

If a control sequence represents LaTeXML::Primitive, the primitive is invoked, executing its stored subroutine. This is typically done for side effect (changing the state in the LaTeXML::State), although they may also contribute digested material. As with macros, any arguments to the primitive are read from the LaTeXML::Gullet.

Grouping (or environment bodies)

are collected into a LaTeXML::List.

Constructors

A special class of control sequence, called a LaTeXML::Constructor produces a LaTeXML::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::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::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::Document

Represents an XML document under construction.

Description

A LaTeXML::Document constructs an XML document by absorbing the digested LaTeXML::List (from a LaTeXML::Stomach), Generally, the LaTeXML::Boxs and LaTeXML::Lists create text nodes, whereas the LaTeXML::Whatsits create XML document fragments, elements and attributes according to the defining LaTeXML::Constructor.

The LaTeXML::Document maintains a current insertion point for where material will be added. The LaTeXML::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.

Accessors

$doc = $document->getDocument;

Returns the XML::LibXML::Document currently being constructed.

$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.

$document->setNode($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.

Construction Methods

`$document->absorb($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.

`$xmldoc = $document->finalize;`

This method finalizes the document by cleaning up various temporary attributes, and returns the `XML::LibXML::Document` that was constructed.

`$document->openText($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.

`$document->insertMathToken($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.

`$document->openElement($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.

$document->insertElement($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. This method returns the newly created node, although it will no longer be the current insertion point.

$document->insertComment($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.

$document->addAttribute($key=>$value);

Add the given attribute to the nearest node that is allowed to have it.

LaTeXML::Model

Represents the Document Model

Description

LaTeXML::Model encapsulates information about the document model to be used in converting a digested document into XML by the LaTeXML::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.

Model Creation

$model = LaTeXML::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.

$auto = $model->canContainIndirect($tag,$childtag);

Checks whether an element with qualified name $tag could contain an element with qualified name $childtag, provided an 'autoOpen'able element $auto were inserted in $tag.

$boole = $model->canContainSomehow($tag,$childtag);

Returns whether an element with qualified name $tag could contain an element with qualified name $childtag, either directly or indirectly.

$boole = $model->canAutoClose($tag);

Returns whether an element with qualified name $tag is allowed to be closed automatically, if needed.

$boole = $model->canHaveAttribute($tag,$attribute);

Returns whether an element with qualified name $tag is allowed to have an attribute with the given name.

Tag Properties

$value = $model->getTagProperty($tag,$property);

Gets the value of the $property associated with the qualified name $tag Known properties are:

```
autoOpen   : This asserts that the tag is allowed to
             be opened automatically if needed to
             insert some other element.  If not set,
             the tag can only be opened explicitly.
autoClose  : This asserts that the $tag is allowed to
             be closed automatically if needed to
             insert some other element.  If not set,
             the tag can only be closed explicitly.
afterOpen  : supplies code to be executed whenever
             an element of this type is opened. It
             is called with the created node and the
             responsible digested object as arguments.
afterClose : supplies code to be executed whenever
             an element of this type is closed.  It
             is called with the created node and the
             responsible digested object as arguments.
```

$model->setTagProperty($tag,$property,$value);

sets the value of the `$property` associated with the qualified name `$tag` to `$value`.

Rewrite Rules

$model->addRewriteRule($mode,@specs);

Install a new rewrite rule with the given `@specs` to be used in `$mode` (being either `math` or `text`). See `LaTeXML::Rewrite` for a description of the specifications.

$model->applyRewrites($document,$node,$until_rule);

Apply all matching rewrite rules to `$node` in the given document. If `$until_rule` is define, apply all those rules that were defined before it, otherwise, all rules

LaTeXML::Rewrite

Rewrite rules for modifying the XML document.

Description

LaTeXML::Rewrite implements rewrite rules for modifying the XML document.

Methods

$rule->rewrite($document,$node);

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 seperated by punctuation nodes. The text of these delimiters and punctuation are used to annotate the operator node with argopen, argclose and separator attributes.

$node = recApply(@ops,$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. If nothing specific is recognized, creates the application of FENCED to the arguments.

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)>

Problem($text);

Warn of a potential math parsing problem.

MaybeFunction($token);

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

LaTeXML::Bib

Implements a BibTeX parser for LaTeXML.

Description

LaTeXML::Bib serves as a low-level parser of BibTeX database files. It parses and stores a LaTeXML::Bib::BibEntry 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 Bib

my $bib = LaTeXML::Bib->newFromFile($bibname);

Creates a LaTeXML::Bib object representing a bibliography from a BibTeX database file.

my $bib = LaTeXML::Bib->newFromString($string);

Creates a LaTeXML::Bib 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.

Appendix D

Utility Module 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 utilties 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.

$path = pathname_relative($path,$base);

Returns the path to file $path relative to the directory $base.

$path = pathname_absolute($path,$base);

Returns the absolute pathname resulting from interpretting $path relative to the directory $base. If $path is already absolute, it is returned unchanged.

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.

@paths = pathname_findall($name,%options);

Like pathname_find, but returns all matching paths that exist.

LaTeXML::Util::KeyVal

Support for keyvals

Description

Provides a parser and representation of keyval pairs `LaTeXML::KeyVal` represents parameters handled by LaTeX's keyval package.

Declarations

`DefKeyVal($keyset,$key,$type);`

Defines the type of value expected for the key $key when parsed in part of a KeyVal using `$keyset`. `$type` would be something like 'any' or 'Number', but I'm still working on this.

Accessors

`KeyVal($arg,$key)`

This is useful within constructors to access the value associated with `$key` in the argument `$arg`.

`KeyVals($arg)`

This is useful within constructors to extract all keyvalue pairs to assign all attributes.

KeyVal Methods

`$value = $keyval->getValue($key);`

Return the value associated with `$key` in the `$keyval`.

`@keyvals = $keyval->getKeyVals;`

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.

`@keyvals = $keyval->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.

`$keyval->digestValues;`

Return a new `LaTeXML::KeyVals` object with all values digested as appropriate.

Appendix E

Postprocessing Module Documentation

LaTeXML::Post

LaTeXML::Post is the driver for various postprocessing operations. It has a complicated set of options that I'll document shortly.

Appendix F

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.

Module `LaTeXML`

Module **`LaTeXML-common`** included.

Module **`LaTeXML-inline`** included.

Module **`LaTeXML-block`** included.

Module **`LaTeXML-para`** included.

Module **`LaTeXML-math`** included.

Module **`LaTeXML-tabular`** included.

Module **`LaTeXML-picture`** included.

Module **`LaTeXML-structure`** included.

Module **`LaTeXML-bib`** included.

Pattern **`Inline.model`** Combined model for inline content.

> ***Content*:** (*text* | `Inline.class` | `Misc.class` | `Meta.class`)*
>
> ***Used by*:** `acknowledgements`, `acronym`, `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, bibref, bibtag, block, caption, cite, classification, constraint, contact, date, emph, givenname, indexphrase, indexrefs, keywords, lineage, p, personname, ref, subtitle, surname, tag, text, title, toccaption, toctitle, verbatim

***Pattern* `Block.model`** Combined model for physical block-level content.

***Content*:** (Block.class | Misc.class | Meta.class)*

***Used by*:** abstract, figure, inline-block, listing, listingblock, 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, 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, index.body.class, paragraph.body.class, part.body.class, section.body.class, subparagraph.body.class, subsection.body.class, subsubsection.body.class, inline-para, item, proof, theorem

***Start* = document**

Module `LaTeXML-common`

***Pattern* `Inline.class`** All strictly inline elements.

***Expansion*:** (text | emph | acronym | rule | anchor | ref | cite | bibref | Math)

***Used by*:** Flow.model, Inline.model, XMText, clippath, g, picture

***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 | listingblock | itemize | enumerate | description)

***Used by*:** Block.model, Flow.model, titlepage

Pattern **`Misc.class`** Additional miscellaneous elements that can appear in both inline and block contexts.

***Expansion*:** (`inline-block` | `verbatim` | `break` | `graphics` | `inline-para` | `tabular` | `picture`)

***Used by*:** `Block.model`, `Flow.model`, `Inline.model`, `XMText`, `clippath`, `creator`, `g`, `picture`

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` | `listing`)

***Used by*:** `BackMatter.class`, `Para.model`

Pattern **`Meta.class`** All metadata elements, typically representing hidden data.

***Expansion*:** (`note` | `indexmark` | `ERROR`)

***Used by*:** `BackMatter.class`, `Block.model`, `Flow.model`, `Inline.model`, `Para.model`, `clippath`, `document`, `equation`, `equationgroup`, `g`, `picture`

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*:** `Positionable.attributes`, `tabular`, `td`

Pattern **`Color.type`** The type for attributes specifying a color. NOTE: To be narrowed later.

***Content*:** *text*

Pattern **`Common.attributes`** Attributes shared by ALL elements.

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.]

***Used by*:** Sectional.attributes, ERROR, Math, MathBranch, MathFork, XMApp, XMArg, XMArray, XMCell, XMDual, XMHint, XMRef, XMRow, XMText, XMTok, XMWrap, XMath, abstract, acknowledgements, acronym, 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, block, break, caption, circle, cite, classification, clip, clippath, contact, creator, curve, date, description, dots, ellipse, emph, enumerate, equation, equationgroup, figure, g, graphics, grid, indexentry, indexlist, indexmark, indexphrase, indexrefs, inline-block, inline-para, item, itemize, keywords, line, listing, listingblock, note, p, para, parabola, path, personname, picture, polygon, proof, quote, rect, ref, rule, subtitle, table, tabular, tag, tbody, td, text, tfoot, thead, theorem, title, toccaption, 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.

***Used by*:** Labelled.attributes, Math, XMApp, XMArg, XMArray, XMDual, XMHint, XMRef, XMText, XMTok, XMWrap, anchor, bibentry, bibitem, block, description, enumerate, graphics, indexentry, indexlist, inline-block, itemize, p, para, picture, quote, 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*:** XMRef, bibref, ref

***Pattern* `Labelled.attributes`** Attributes for elements that can be labelled from within LaTeX.

***Includes*:** ID.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!)

Attribute **`refnum`** = *text*

the reference number (ie. section number, equation number, etc) of the object.

Used by: `Sectional.attributes`, `equation`, `equationgroup`, `figure`, `item`, `listing`, `listingblock`, `proof`, `table`, `theorem`

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 **`pad-width`** = `Length.type`

extra width beyond the boxes natural size.

Attribute **`pad-height`** = `Length.type`

extra height beyond the boxes natural size.

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`)

specifies which line of the box is aligned to the baseline of the containing object.

Used by: `block`, `inline-block`, `inline-para`, `p`, `rule`, `text`

Pattern **`Imageable.attributes`** Attributes for elements that may be converted to image form during postprocessing, such as math, graphics, pictures, etc.

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`.

Attribute **`description`** = *text*

a description of the image

Used by: `Math`, `graphics`, `picture`

Module `LaTeXML-inline`

Add to **`Inline.class`** The inline module defines basic inline elements used throughout

|= (`text` | `emph` | `acronym` | `rule` | `anchor` | `ref` | `cite` | `bibref`)

Add to **`Meta.class`** Additionally, it defines these meta elements. These are generally hidden, and can appear in inline and block contexts.

|= (`note` | `indexmark` | `ERROR`)

Element **`text`** General container for styled text. Attributes cover a variety of styling and position shifting properties.

Includes: `Common.attributes`, `Positionable.attributes`

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 **`size`** = (`Huge` | `huge` | `LARGE` | `Large` | `large` | `normal` | `small` | `footnote` | `tiny` | *text*)

Indicates the text size to use. The values are modeled after the more abstract LaTeX font size switches, rather than point-sizes. The values are open-ended for extensibility; In postprocessing, these values are carried to the `class` attribute, and can thus be effected by CSS.

Attribute **`color`** = *text*

the color to use; any CSS compatible color specification. In postprocessing, these values are carried to the `class` attribute, and can thus be effected by CSS.

Attribute **`framed`** = (`rectangle` | `underline` | *text*)

the kind of frame or outline for the text.

Content: `Inline.model`

Used by: `Inline.class`, `equation`

Element **`emph`** Emphasized text.

Includes: `Common.attributes`

Content: `Inline.model`

Used by: `Inline.class`

Element **`acronym`** Represents an acronym.

Includes: `Common.attributes`

Attribute **`name`** = *text*

should be used to indicate the expansion of the acronym.

Content: `Inline.model`

Used by: `Inline.class`

Element **`rule`** A Rule.

Includes: `Common.attributes`, `Positionable.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).

Includes: `Common.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.

***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` mixed with arbitrary characters. The It can also be `fulltitle`, which indicates the title with prefix and type if section numbering is enabled.

***Attribute* `title` =** *text*

gives a longer form description of the target, this would typically appear as a tooltip in HTML. Typically filled in by postprocessor.

***Content*:** `Inline.model`

***Used by*:** `Inline.class`

***Element* `anchor`** Inline anchor.

***Includes*:** `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.

***Includes*:** `Common.attributes`

***Content*:** `Inline.model`

***Used by*:** `Inline.class`

***Element* `bibref`** A bibliographic citation refering to a specific bibliographic item.

***Includes*:** `Common.attributes`, `IDREF.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*:** `Inline.model`

Used by: Inline.class

Element **note** Metadata that covers several 'out of band' annotations. It's content allows both inline and block-level content.

Includes: Common.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: Flow.model

Used by: Meta.class

Element **ERROR** error object for undefined control sequences, or whatever

Includes: Common.attributes

Content: *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.

Includes: Common.attributes

Attribute **see_also** = *text*

a flattened form (like key) of another indexmark, used to crossreference.

Attribute **style** = *text*

NOTE: describe this.

Content: indexphrase*

Used by: Meta.class

Element **indexphrase** A phrase within an indexmark

Includes: Common.attributes

Attribute **key** = *text*

a flattened form of the phrase for generating an ID.

Content: Inline.model

Used by: indexentry, indexmark

Module `LaTeXML-block`

Add to **`Block.class`** The block module defines the following 'physical' block elements.

|= (p | equation | equationgroup | quote | block
| listingblock | itemize | enumerate | description)

Add to **`Misc.class`** Additionally, it defines these miscellaneous elements that can appear in both inline and block contexts.

|= (inline-block | verbatim | break | graphics)

Pattern **`EquationMeta.class`** Additional Metadata that can be present in equations.

Content: constraint

Used by: equation, equationgroup

Element **`p`** A physical paragraph.

Includes: Common.attributes, ID.attributes, Positionable.attributes

Content: Inline.model

Used by: Block.class

Element **`constraint`** A constraint upon an equation.

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

Includes: Common.attributes, Labelled.attributes

Content: (Math | MathFork | text | tabular | Meta.class | EquationMeta.class)*

Used by: Block.class, equationgroup

Element **`equationgroup`** A group of equations, perhaps aligned (Though this is nowhere recorded).

Includes: Common.attributes, Labelled.attributes

Content: (equationgroup | equation | block | Meta.class | EquationMeta.class)*

Used by: Block.class, equationgroup

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.

Includes: Common.attributes

Content: Math, MathBranch*

Used by: equation

Element **MathBranch** A container for an alternatively formatted math representation.

Includes: Common.attributes

Attribute **format** = *text*

Content: (Math | tr | td)*

Used by: MathFork

Element **quote** A quotation.

Includes: Common.attributes, ID.attributes

Content: Block.model

Used by: Block.class

Element **block** A generic block (fallback).

Includes: Common.attributes, ID.attributes, Positionable.attributes

Content: Inline.model

Used by: Block.class, equationgroup

Element **listingblock** An in-block Listing, without caption

Includes: Common.attributes, Labelled.attributes

Content: Block.model*

Used by: Block.class

Element **break** A forced line break.

Includes: Common.attributes

Content: *empty*

Used by: Misc.class

Element **`inline-block`** An inline block. Actually, can appear in inline or block mode, but typesets its contents as a block.

Includes: `Common.attributes`, `ID.attributes`, `Positionable.attributes`

Content: `Block.model`

Used by: `Misc.class`

Element **`verbatim`** Verbatim content

Includes: `Common.attributes`, `ID.attributes`

Attribute **`font`** = *text*

the font to use; generally typewriter.

Content: `Inline.model`

Used by: `Misc.class`

Element **`itemize`** An itemized list.

Includes: `Common.attributes`, `ID.attributes`

Content: `item`*

Used by: `Block.class`

Element **`enumerate`** An enumerated list.

Includes: `Common.attributes`, `ID.attributes`

Content: `item`*

Used by: `Block.class`

Element **`description`** A description list. The `item`s within are expected to have a `tag` which represents the term being described in each `item`.

Includes: `Common.attributes`, `ID.attributes`

Content: `item`*

Used by: `Block.class`

Element **`item`** An item within a list.

Includes: `Common.attributes`, `Labelled.attributes`

Content: `tag`?, `Para.model`

Used by: `description`, `enumerate`, `itemize`

Element **`tag`** A tag within an item indicating the term or bullet for a given item.

Includes: `Common.attributes`

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*:** `item`

Element **`graphics`** A graphical insertion of an external file.

***Includes*:** `Common.attributes`, `ID.attributes`, `Imageable.attributes`

Attribute **`graphic`** = *text*

the path to the graphics file

Attribute **`options`** = *text*

an encoding of the scaling and positioning options to be used in processing the graphic.

***Content*:** *empty*

***Used by*:** `Misc.class`

Module `LaTeXML-para`

Add to **`Para.class`** This module defines the following 'logical' block elements.

|= (`para` | `theorem` | `proof` | `figure` | `table` | `listing`)

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`.

***Includes*:** `Common.attributes`, `ID.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.

***Includes*:** `Common.attributes`, `Positionable.attributes`

***Content*:** `Para.model`

***Used by*:** `Misc.class`

Element **`theorem`** A theorem or similar object. The `class` attribute can be used to distinguish different kinds of theorem.

***Includes*:** `Common.attributes`, `Labelled.attributes`

***Content*:** `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.

***Includes*:** `Common.attributes`, `Labelled.attributes`

***Content*:** `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`, `listing`, `table`

Element **`figure`** A figure, possibly captioned.

***Includes*:** `Common.attributes`, `Labelled.attributes`

Attribute **`placement`** = *text*

the floating placement parameter that determines where the object is displayed.

***Content*:** (`Block.model` | `Caption.class`)*

***Used by*:** `Para.class`

Element **`table`** A Table, possibly captioned. This is not necessarily a `tabular`.

***Includes*:** `Common.attributes`, `Labelled.attributes`

Attribute **`placement`** = *text*

the floating placement parameter that determines where the object is displayed.

***Content*:** (`Block.model` | `Caption.class`)*

***Used by*:** `Para.class`

Element **`listing`** A Listing, possibly captioned.

***Includes*:** `Common.attributes`, `Labelled.attributes`

Attribute **`placement`** = *text*

the floating placement parameter that determines where the object is displayed.

***Content*:** (`Block.model` | `Caption.class`)*

***Used by*:** `Para.class`

Element **`caption`** A caption for a `table` or `figure`.

Includes: Common.attributes

Content: Inline.model

Used by: Caption.class

Element **toccaption** A short form of table or figure caption, used for lists of figures or similar.

Includes: Common.attributes

Content: Inline.model

Used by: Caption.class

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)

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.

Includes: Common.attributes, Imageable.attributes, ID.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.

***Content*:** `Math.class`*

***Used by*:** `Inline.class`, `MathBranch`, `MathFork`, `equation`

Pattern **`XMath.attributes`**

Attribute **`role`** = *text*

The role that this item plays in the Grammar.

Attribute **`open`** = *text*

an open delimiter to enclose the object;

Attribute **`close`** = *text*

an close delimiter to enclose the object;

Attribute **`argopen`** = *text*

an open delimiter to enclose the argument list, when this token is applied to arguments with `XMApp`.

Attribute **`argclose`** = *text*

a close delimiter to enclose the argument list, when this token is applied to arguments with `XMApp`.

Attribute **`separators`** = *text*

characters to separate arguments, when this token is applied to arguments with `XMApp`. Can be multiple characters for different argument positions; the last character is repeated if there aren't enough.

Attribute **`punctuation`** = *text*

trailing (presumably non-semantic) punctuation.

Attribute **`possibleFunction`** = *text*

an annotation placed by the parser when it suspects this token may be used as a function.

***Used by*:** `XMApp`, `XMArg`, `XMArray`, `XMDual`, `XMHint`, `XMRef`, `XMText`, `XMTok`, `XMWrap`

Element **`XMath`** Internal representation of mathematics.

***Includes*:** `Common.attributes`

***Content*:** `XMath.class`*

***Used by*:** `Math.class`

Element **`XMTok`** General mathematical token.

***Includes*:** `Common.attributes`, `XMath.attributes`, `ID.attributes`

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 **`style`** = *text*

Various random styling information. NOTE This needs to be made consistent.

Attribute **`font`** = *text*

The font, size a used for the symbol.

Attribute **`size`** = *text*

The size for the symbol, not presumed to be meaningful(?)

Attribute **`color`** = *text*

The color (CSS format) for the symbol, not presumed to be meaningful(?)

Attribute **`scriptpos`** = *text*

An encoding of the position of this token as a sub/superscript, used to handle aligned and nested scripts, both pre and post. It is a concatenation of (pre—mid—post), which indicates the horizontal positioning of the script with relation to it's base, and a counter indicating the level. These are used to position the scripts, and to pair up aligned sub- and superscripts. NOTE: Clarify where this appears: token, base, script operator, apply?

Attribute **`thickness`** = *text*

NOTE: How is this used?

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`.

Includes**:** `Common.attributes`, `XMath.attributes`, `ID.attributes`

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 **`scriptpos`** = *text*

An encoding of the position of this token as a sub/superscript, used to handle aligned and nested scripts, both pre and post.

***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

***Includes*:** Common.attributes, XMath.attributes, ID.attributes

***Content*:** XMath.class, XMath.class

***Used by*:** XMath.class

***Element* XMHint** Various spacing items, generally ignored in parsing. The attributes are a subset of those for XMTok.

***Includes*:** Common.attributes, XMath.attributes, ID.attributes

***Attribute* name** = *text*

***Attribute* meaning** = *text*

***Attribute* style** = *text*

***Content*:** *empty*

***Used by*:** XMath.class

***Element* XMText** Text appearing within math.

***Includes*:** Common.attributes, XMath.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.

***Includes*:** Common.attributes, XMath.attributes, ID.attributes

***Attribute* name** = *text*

***Attribute* meaning** = *text*

A more semantic name corresponding to the intended meaning, such as the OpenMath name.

***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.

***Includes*:** `Common.attributes`, `XMath.attributes`, `ID.attributes`

***Attribute* `rule` =** *text*

***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.

***Includes*:** `Common.attributes`, `XMath.attributes`, `ID.attributes`, `IDREF.attributes`

***Content*:** *empty*

***Used by*:** `XMath.class`

***Element* `XMArray`** Math Array/Alignment structure.

***Includes*:** `Common.attributes`, `XMath.attributes`, `ID.attributes`

***Attribute* `name` =** *text*

***Attribute* `meaning` =** *text*

***Attribute* `style` =** *text*

***Attribute* `vattach` =** (`top` | `bottom`)

***Attribute* `width` =** *text*

***Content*:** `XMRow`*

***Used by*:** `XMath.class`

***Element* `XMRow`** A row in a math alignment.

***Includes*:** `Common.attributes`

***Content*:** `XMCell`*

***Used by*:** `XMArray`

***Element* `XMCell`** A cell in a row of a math alignment.

***Includes*:** `Common.attributes`

***Attribute* `colspan` =** *nonNegativeInteger*
indicates how many columns this cell spans or covers.

***Attribute* `rowpan` =** *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`** = *boolean*
whether this cell corresponds to a table head or foot.

Content: `XMath.class`*

Used by: `XMRow`

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.

Includes: `Common.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.

Content: (`thead` | `tfoot` | `tbody` | `tr`)*

Used by: `Misc.class`, `equation`

Element **`thead`** A container for a set of rows that correspond to the header of the tabular.

Includes: `Common.attributes`

Content: `tr`*

Used by: `tabular`

Element **`tfoot`** A container for a set of rows that correspond to the footer of the tabular.

Includes: `Common.attributes`

Content: `tr`*

***Used by*:** tabular

***Element* `tbody`** A container for a set of rows corresponding to the body of the tabular.

***Includes*:** Common.attributes

***Content*:** tr*

***Used by*:** tabular

***Element* `tr`** A row of a tabular.

***Includes*:** Common.attributes

***Content*:** td*

***Used by*:** MathBranch, tabular, tbody, tfoot, thead

***Element* `td`** A cell in a row of a tabular.

***Includes*:** Common.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` =** Length.type

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` =** *boolean*

whether this cell corresponds to a table head or foot.

***Content*:** Flow.model

***Used by*:** MathBranch, tr

Module `LaTeXML-picture`

***Add to* `Misc.class`** This module defines a picture environment, roughly a subset of SVG. NOTE: Consider whether it is sensible to drop this and incorporate SVG itself.

|= picture

***Pattern* `Picture.class`**

***Content*:** (g | rect | line | circle | path | arc | wedge | ellipse | polygon | bezier | parabola | curve | dots | grid | clip)

***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.

Includes: Common.attributes, ID.attributes, Picture.attributes, Imageable.attributes

Attribute **clip** = *boolean*

Attribute **baseline** = *text*

Attribute **unitlength** = *text*

Attribute **xunitlength** = *text*

Attribute **yunitlength** = *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.

Includes: Common.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.

Includes: Common.attributes, Picture.attributes

Content: *empty*

Used by: Picture.class

Element **line** A line within a picture.

Includes: Common.attributes, Picture.attributes

Content: *empty*

Used by: Picture.class

Element **polygon** A polygon within a picture.

Includes: Common.attributes, Picture.attributes

Content: *empty*

Used by: Picture.class

Element **wedge** A wedge within a picture.

Includes: Common.attributes, Picture.attributes

Content: *empty*

Used by: Picture.class

Element **arc** An arc within a picture.

Includes: Common.attributes, Picture.attributes

Content: *empty*

Used by: Picture.class

Element **circle** A circle within a picture.

Includes: Common.attributes, Picture.attributes

Content: *empty*

Used by: Picture.class

Element **ellipse** An ellipse within a picture.

Includes: Common.attributes, Picture.attributes

Content: *empty*

Used by: Picture.class

Element **path** A path within a picture.

Includes: Common.attributes, Picture.attributes

Content: *empty*

Used by: Picture.class

Element **bezier** A bezier curve within a picture.

Includes: Common.attributes, Picture.attributes

Content: *empty*

Used by: Picture.class

Element **curve** A curve within a picture.

Includes: Common.attributes, Picture.attributes

Content: *empty*

Used by: Picture.class

Element **parabola** A parabola curve within a picture.

Includes: Common.attributes, Picture.attributes

Content: *empty*

Used by: Picture.class

Element **dots** A sequence of dots (?) within a picture.

Includes: Common.attributes, Picture.attributes

Content: *empty*

Used by: Picture.class

Element **grid** A grid within a picture.

Includes: Common.attributes, Picture.attributes

Content: *empty*

Used by: Picture.class

Element **clip** Establishes a clipping region within a picture.

Includes: Common.attributes, Picture.attributes

Content: clippath*

Used by: Picture.class

Element **clippath** Establishes a clipping region within a picture.

Includes: Common.attributes, Picture.attributes

Content: (Picture.class | Inline.class | Misc.class | Meta.class)*

Used by: clip

Module LaTeXML-structure

Element **document** The document root.

Includes: 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)

Used by: document

Element **part** A part within a document.

Includes: Sectional.attributes

Content: SectionalFrontMatter.class*, part.body.class*

Used by: document.body.class

Pattern **part.body.class** The content allowable as the main body of a part.

Content: (Para.model | chapter)

Used by: part

Element **chapter** A Chapter within a document.

Includes: Sectional.attributes

Content: SectionalFrontMatter.class*, chapter.body.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)

Used by: chapter

Element **section** A Section within a document.

Includes: Sectional.attributes

Content: SectionalFrontMatter.class*, section.body.class*

Used by: appendix.body.class, chapter.body.class, document.body.class

Pattern **section.body.class** The content allowable as the main body of a section.

Content: (Para.model | subparagraph | paragraph | subsubsection | subsection)

Used by: section

Element **appendix** An Appendix within a document.

Includes: 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)

Used by: appendix

Element **subsection** A Subsection within a document.

Includes: Sectional.attributes

Content: SectionalFrontMatter.class*, subsection.body.class*

Used by: appendix.body.class, chapter.body.class, document.body.class, section.body.class

Pattern **subsection.body.class** The content allowable as the main body of a chapter.

Content: (Para.model | subparagraph | paragraph | subsubsection)

Used by: subsection

Element **subsubsection** A Subsubsection within a document.

Includes: Sectional.attributes

Content: SectionalFrontMatter.class*, subsubsection.body.class*

Used by: appendix.body.class, chapter.body.class, document.body.class, section.body.class, subsection.body.class

Pattern **subsubsection.body.class** The content allowable as the main body of a chapter.

Content: (Para.model | subparagraph | paragraph)

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.

Includes: Sectional.attributes

Content: SectionalFrontMatter.class*, paragraph.body.class*

Used by: appendix.body.class, chapter.body.class, document.body.class, section.body.class, subsection.body.class, subsubsection.body.class

Pattern **paragraph.body.class** The content allowable as the main body of a chapter.

Content: (Para.model | subparagraph)

Used by: paragraph

Element **subparagraph** A Subparagraph within a document.

Includes: Sectional.attributes

Content: SectionalFrontMatter.class*, subparagraph.body.class*

Used by: appendix.body.class, chapter.body.class, paragraph.body.class, section.body.class, subsection.body.class, subsubsection.body.class

Pattern **subparagraph.body.class** The content allowable as the main body of a chapter.

Content: Para.model

Used by: subparagraph

Element **bibliography** A Bibliography within a document.

Includes: Sectional.attributes

Attribute **files** = *text*
the list of bib files used to create the bibliograph.

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.

Includes: Sectional.attributes

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 generated. Typically generated during postprocessing from the collection of `indexmark` in the document (or document collection).

Includes: `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.

Includes: `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.

Includes: `Common.attributes`

Content: `Inline.model`

Used by: `indexentry`

Element **`title`** The title of a document, section or similar document structure container.

Includes: `Common.attributes`

Content: `Inline.model`

Used by: `SectionalFrontMatter.class`, `proof`, `theorem`

Element **`toctitle`** The short form of a title, for use in tables of contents or similar.

Includes: `Common.attributes`

Content: `Inline.model`

Used by: `SectionalFrontMatter.class`

Element **`subtitle`** A subtitle, or secondary title.

Includes: `Common.attributes`

Content: `Inline.model`

Used by: `FrontMatter.class`

Element **`personname`** A person's name.

Includes: `Common.attributes`

Content: `Inline.model`

Used by: `Person.class`

Element **`creator`** Generalized document creator.

Includes: `Common.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.

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 **`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.

Includes: `Common.attributes`

Attribute **`role`** = (`affiliation` | `address` | `current_address` | `email` | `url` | `thanks` | `dedicatory` | *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.

Includes: `Common.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.

Includes: Common.attributes

Content: Block.model

Used by: FrontMatter.class

Element **acknowledgements** Acknowledgements for the document.

Includes: Common.attributes

Content: Inline.model

Used by: BackMatter.class, FrontMatter.class

Element **keywords** Keywords for the document. The content is freeform.

Includes: Common.attributes

Content: Inline.model

Used by: FrontMatter.class

Element **classification** A classification of the document.

Includes: Common.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

Includes: Sectional.attributes

Content: (FrontMatter.class | SectionalFrontMatter.class | Block.class)*

Used by: document

Pattern **Sectional.attributes** Attributes shared by all sectional elements

Includes: Common.attributes, Labelled.attributes

Used by: appendix, bibliography, chapter, document, index, paragraph, part, section, subparagraph, subsection, subsubsection, titlepage

Pattern **SectionalFrontMatter.class** The content allowed for the front matter of each sectional unit, and the document.

Content: (title | toctitle | creator)

Used by: appendix, bibliography, chapter, document, index, paragraph, part, section, 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, 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 | acknowledgements | Para.class | Meta.class)

Used by: document

Module LaTeXML-bib

Element **biblist** A list of bibliographic bibentry or bibitem.

Includes: 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.

Includes: Common.attributes, ID.attributes

Attribute **key** = *text*

The unique key for this object; this key is referenced by the bibrefs attribute of bibref.

Content: bibtag*, bibblock*

Used by: biblist

Element **bibtag** Various formatted tags for bibliographic items. Typically @role refnum is shown in the displayed bibliography, as the beginning of the item. Other roles (eg. number, authors, fullauthors, year, title) record formatted info to be used for filling in citations (bibref).

Attribute **role** = (number | authors | fullauthors | key | year | bibtype | title | *text*)

Attribute **`open`** = *text*

A delimiter for formatting the refnum in the bibliography

Attribute **`close`** = *text*

A delimiter for formatting the refnum in the bibliography

Content: `Inline.model`

Used by: `bibitem`

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

Includes: `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` | `masterthesis` | `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.

Includes: `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.

Includes: Common.attributes

Content: Inline.model

Used by: Bibentry.class

Element **bib-subtitle** Subtitle of a bibliographic entry.

Includes: Common.attributes

Content: Inline.model

Used by: Bibentry.class

Element **bib-key** Unique key of a bibliographic entry.

Includes: Common.attributes

Content: Inline.model

Used by: Bibentry.class

Element **bib-type** Type of a bibliographic entry.

Includes: Common.attributes

Content: Inline.model

Used by: Bibentry.class

Element **bib-date** Date of a bibliographic entry.

Includes: 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.

Includes: Common.attributes

Content: Inline.model

Used by: Bibentry.class

Element **bib-organization** Organization responsible for a bibliographic entry.

Includes: Common.attributes

Content: Inline.model

Used by: Bibentry.class

Element **bib-place** Location of publisher or event

Includes: 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.

Includes: 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.

Includes: 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.

Includes: `Common.attributes`

Content: `Inline.model`

Used by: `Bibentry.class`

Element **`bib-status`** Status of a bibliographic entry.

Includes: `Common.attributes`

Content: `Inline.model`

Used by: `Bibentry.class`

Element **`bib-identifier`** Some form of document identfier. The content is descriptive.

Includes: `Common.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.

Attribute **`href`** = *text*
a url to the document, if available

Content: `Inline.model`

Used by: `Bibentry.class`

Element **`bib-review`** Review of a bibliographic entry. The content is descriptive.

Includes: `Common.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.

Attribute **`href`** = *text*
a url to the review, if available

Content: `Inline.model`

Used by: `Bibentry.class`

Element **`bib-links`** Links to other things like preprints, source code, etc.

Includes: `Common.attributes`

Content: `Inline.model`

Used by: `Bibentry.class`

Element **`bib-language`** Language of a bibliographic entry.

Includes: `Common.attributes`

Content: `Inline.model`

Used by: `Bibentry.class`

Element **`bib-url`** A URL for a bibliographic entry. The content is descriptive

Includes: `Common.attributes`

Attribute **`href`** = *text*

Content: `Inline.model`

Used by: `Bibentry.class`

Element **`bib-extract`** An extract from the referenced object.

Includes: `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.

Includes: `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?).

Includes: `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 G

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, itself.

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:

```
(Warning|Error|Fatal)(:\S*)\s+(.*)
```

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 : *arg* indicates the undefined control sequence.

expected : *arg* was expected in the input but missing. The expected thing will likely either be a control sequence or something like `<variable>` to indicate that a variable was expected.

unexpected : *arg* was not expected to appear in the input.

missing_file : the file *arg* could not be found. Also used when the file is otherwise not readable or processable.

latex : An error or message generated from LATEX code.

parse : An issue parsing the mathematics.

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.

perl : A perl-level error or warning,not specifically recognized by LaTeXML, was encountered. The second keyword will typically `die`, `interrupt` or `warn`.

malformed : some sort of malformed XML problem.

model : some sort of problem with the document model or schema.

misdefined : Some sort of error in the definition of *arg*.

internal : Something unexpected happened; most likey an internal coding error within LATEXML.

too_many : Too many error were encountered.

Should there be an additional level that identifies the processing stage? Eg. mouth, gullet, stomach, intestine, ...? That might semi-automatically distinguish expected, unexpected, malformed? Or does it?

Index