

L^AT_EX XML *The Manual*

A L^AT_EX to XML Converter;
0.6.0

Bruce R. Miller

April 9, 2008

Contents

1	Introduction	1
2	Using L^AT_EX_μ	3
2.1	Conversion	4
2.2	Postprocessing	5
2.3	Splitting	8
2.4	Sites	8
3	Architecture	9
3.1	Digestion	9
3.2	Construction	11
3.3	Rewriting	11
4	Customization	13
5	Mathematics	15
5.1	Math Details	16
5.1.1	Internal Math Representation	16
5.1.2	Grammatical Roles	18
6	ToDo	21
A	Commands	23
B	Modules	33
C	Utility Modules	89
D	Postprocessing Modules	93
E	Schema	95

Chapter 1

Introduction

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

The choice of \LaTeX for authoring, and XML for delivery were natural and uncontroversial choices for the Digital Library of Mathematical Functions¹. Faced with the need to perform this conversion and the lack of suitable tools to perform it, the DLMF project proceeded to develop their own tool, $\text{\LaTeX}XML$, for this purpose. This document describes a *preview* release of $\text{\LaTeX}XML$.

Design Goals The idealistic goals of $\text{\LaTeX}XML$ 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

¹<http://dlmf.nist.gov>

\TeX 's behaviour would seem to require the sneakiest modifications to \TeX , itself. '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 \LaTeX , 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 A, B give detailed documentation on the commands and modules comprising the system.

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

Chapter 2

Using L^AT_EX_{XML}

The main commands provided by the L^AT_EX_{XML} system are

`latexml` for converting T_EX 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 HTML, or as complicated as

```
latexml --dest=A.xml doca
latexml --dest=B.xml docb
...
latexmlpost --prescan --dbfile=my.db --dest=A.xhtml A
latexmlpost --prescan --dbfile=my.db --dest=B.xhtml B
...
latexmlpost --noscan --dbfile=my.db --dest=A.xhtml A
latexmlpost --noscan --dbfile=my.db --dest=B.xhtml B
...
```

to convert a whole set of documents into a complete 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 will be sufficient if the XML representation is what you want, or if you intend to carry out any further processing with your own XML-tools. The following sections consider a sequence of successively more complicated postprocessing situations, using `latexmlpost`, in which one or more T_EX sources can be converted into one or more web documents or a complete site.

2.1 Basic xml Conversion

The command

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

loads any required definition modules (see below), reads, tokenizes, expands and digests the T_EX document *doc.tex* (or from standard input, if - is given for the filename), converts it to XML, performs some document rewriting, parses the mathematical content and writes the result in *doc.xml*. For details on the processing, see Chapter 3, and Chapter 5 for more information about math parsing.

Module Loading A first consideration is what definitions for control sequences and environments are active and used for the processing. Definitions and customization modules, if present, are loaded in the following order:

TeX.pool.ltxml the core module is always loaded.

`--preload=module` causes loading of *module.ltxml*. For example, if L^AT_EX_{ML} fails to recognize a L^AT_EX document `--preload=LaTeX.pool` can be useful to force L^AT_EX-mode. Or if you want This option can be repeated, and the modules will be loaded in the given order.

doc.latexml a document-specific customization module is loaded if present.

As processing proceeds, additional modules may be loaded as follows.

LaTeX.pool.ltxml the core latex module, is loaded upon encountering certain recognizably L^AT_EX-specific commands, such as `\documentclass`.

`\documentclass{class}` loads *class.cls.ltxml*. (legacy `\documentstyle` behaves similarly, along with any required packages).

`\usepackage{package}` (or related) loads *package.sty.ltxml*. L^AT_EX_{ML} will not attempt to read the *package.sty* file, as these often involve L^AT_EX internals meaningless to the generation of XML, unless forced to with the option

```
--includestyles
```

A selective, per-file, option may be developed in the future — please provide use cases.

`\input{file}` loads an appropriate version of *file*, specifically the first found of: *file.tex.ltxml*, *file.tex*, *file.ltxml* or *file*.

Some of these modules (esp. TeX and LaTeX), are parts of the L^AT_EX_{ML} distribution; others are supplied by the user, or can be overridden by the user. See Chapter 4 for details about what can go in these modules.

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

`--path=directory`

This option can be repeated.

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

`--verbose` and `--quiet`

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

An option most useful in constructing complicated sites is

`--documentid=id`

which provides an ID for the document root element which is inherited as a prefix for id's of the child-elements in the document. Using this option can assure unique identifiers across a set of source documents.

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

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 D.

Scanning The scanning step collects information about all labels, ids, indexing commands, cross-references and so on, to be used in the following post-processing 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 Bibliographic data from BibTeX can be provided with the option

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

However, the tools to convert a BibTeX file to XML are not yet provided with the distribution.

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 right.

`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 L^AT_EX_{ML} document type is necessary; See Appendix E.

2.3 Splitting the Output

For larger documents, it is often desirable to break the result into several inter-linked 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. 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 forms create shorter names in subdirectories for each level of splitting. The `--urlstyle` option may also be useful here, as well as the `latexml` option `--documentid`.

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 T_EX sources into a single inter-linked site consisting of multiple pages and a composite index and bibliography. The games one must play with L^AT_EX's aux files to satisfy cross-references between these documents are not covered here, but the situation is handled by L^AT_EX_{ML} in the following fashion.

Conversion First, all T_EX 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.

Chapter 3

Architecture

Like $\text{T}_{\text{E}}\text{X}$, $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}\text{M}\text{L}$ is data-driven: the text and executable control sequences (ie. macros and primitives) in the source file (and any packages loaded) direct the processing. The user exerts control over the conversion, and customizes it, by providing alternative $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}\text{M}\text{L}$ -specific implementations 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 $\text{T}_{\text{E}}\text{X}$ document or string into an XML document, with varying degrees of postprocessing and optionally writing the document to file. A `LaTeXML::State` object maintains the current state of processing, current definitions for control sequences and emulates the $\text{T}_{\text{E}}\text{X}$'s scoping rules. The processing is broken into the following stages

Digestion the $\text{T}_{\text{E}}\text{X}$ -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.

See Figure 3.1 for illustration. The first three stages are discussed in the following sections; the parsing of mathematics is covered in detail in Chapter 5.

The `LaTeXML` object binds `STATE`, `GULLET`, `STOMACH`, and `MODEL` to corresponding active objects during processing.

3.1 Digestion

Digestion is carried out primarily in a *pull* mode: The `LaTeXML::Stomach` pulls expanded `LaTeXML::Tokens` from the `LaTeXML::Gullet`, which itself pulls tokens from the `LaTeXML::Mouth`. The `LaTeXML::Mouth` converts characters from the plain text input into tokens according to the current category codes

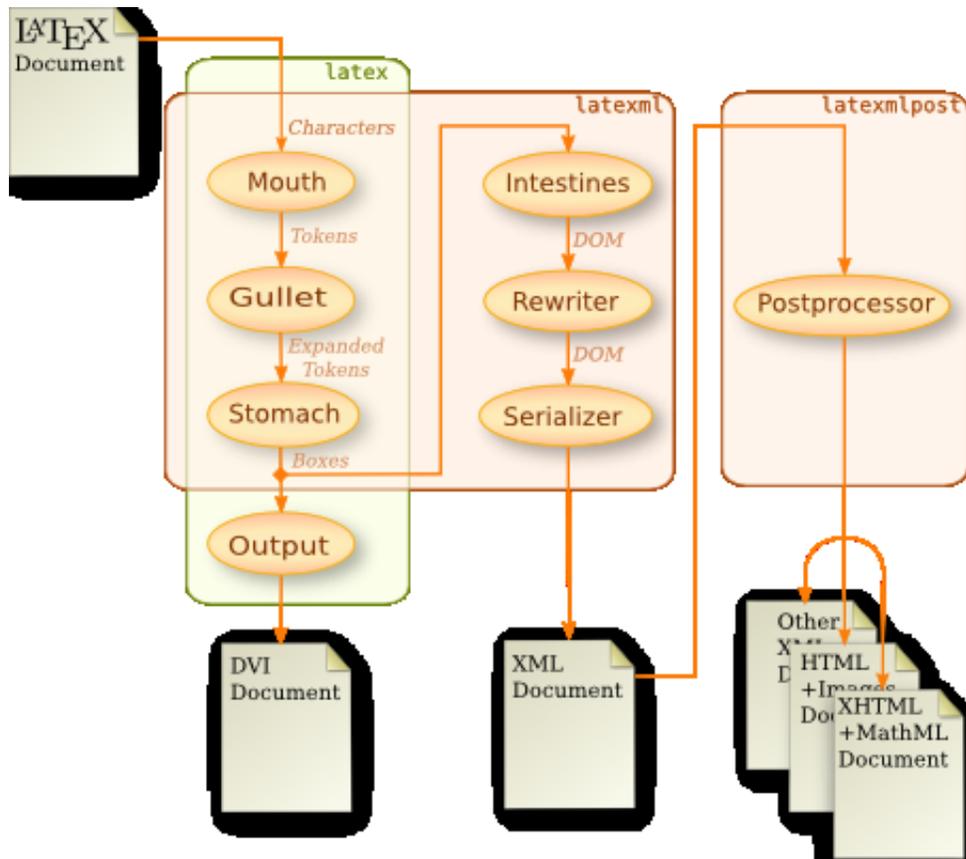


Figure 3.1: Flow of data through LaTeXML's digestive tract.

assigned to them (in the `LaTeXML::State`). The `LaTeXML::Gullet` is responsible for expanding any macro or expandible tokens (when the current binding of the token in the `LaTeXML::State` is an `LaTeXML::Expandable` definition), and for parsing sequences of tokens into common core datatypes (numbers, dimensions, etc.). The `LaTeXML::Stomach` digests these tokens by executing `LaTeXML::Primitive` control sequences (generally for side effect), converting control sequences bound to `LaTeXML::Constructors` into `LaTeXML::Whatsits`, and converting the remaining tokens into a recursive structure consisting of `LaTeXML::Boxes` and `LaTeXML::Lists` and of `LaTeXML::Boxes`.

3.2 Construction

The main (intentional) deviation of \LaTeX XML's digestion from that of \TeX is in the extension of control sequences to include `LaTeXML::Constructors` responsible for constructing XML document fragments, and `LaTeXML::Whatsits` to represent thier digested form including whatever arguments were passed to the control sequence.

Construction thus consists of creating an `LaTeXML::Document`, containing an `XML::LibXML::Document` structure, and having it absorb the digested lists, boxes and whatsits. Generally, boxes represent text which is converted to text nodes within the document. Whatsits generally create a document fragment involving elements, attributes and text.

A `LaTeXML::Model` is maintained throughtout the digestion phase which accumulates any document model declarations in particular the document type (currently only the DTD, but eventually may be RelaxNG based). As \LaTeX markup is more like SGML than XML, declarations may be used 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.

3.3 Rewriting

Once the basic document is constructed, `LaTeXML::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.

Chapter 4

Customization

The processing of the \LaTeX document and its conversion into XML is affected by the definitions of control sequences, either as macros, primitives or constructors, and other declarations specifying the document type, properties of XML tags, ligatures, These definitions and declarations are typically contained in ‘packages’ which provide the implementation of \LaTeX classes and packages. For example, the \LaTeX directive `\usepackage{foo}` would cause \LaTeX XML to load the file `foo.sty.ltxml`. This file would be sought in any of the directories in perl’s `@INC` list (typically including the current directory), or in a `LaTeXML/Package` subdirectory of any of those directories. If no such file is found, \LaTeX XML would look for `foo.sty` and attempt to process it.

When processing a typical file, say `jobname.tex`, the following packages are loaded:

1. the core `TeX` package
2. any packages named with the `--preload` option,
3. a file `jobname.latexml`, if present; this provides for document-specific declarations.

Document processing then commences; by default, \LaTeX XML assumes that the document is plain `TeX`. However, if a `\documentclass` directive is encountered, the `LaTeX` package, as well as a package for the named document class are loaded.

\LaTeX XML implementations are provided for a number of the standard \LaTeX packages, although many implement only part of the functionality. Contributed implementations are, of course, welcome. These files, as well as the document specific `jobname.latexml`, are essentially Perl modules, but use the facilities described in [LaTeXML::Package](#).

Much more needs to be explained here, but for the time being, please consult the documentation for the module [LaTeXML::Package](#), and the various implementations of packages included with the distribution.

Chapter 5

Mathematics

There are several issues that have to be dealt with in treating the mathematics. On the one hand, the $\text{T}_{\text{E}}\text{X}$ 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 $\text{T}_{\text{E}}\text{X}$ -like processing serves as the lexer, tokenizing the input which $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}^{\text{M}}\text{L}$ 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 $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$'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 **XMAp**, 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:

- `LaTeXML::Constructors` 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 ambiguous 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** parentheses 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 **XMAp**) 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.

XMAp represents the generalized application of some function or operator to arguments. The first child element is the operator, the remaining 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 interpreted 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. $\{x\}^{\{y\}}$. 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...!

- Lots of useful L^AT_EX packages have not been implemented, and those that are aren't necessarily complete.
- T_EX boxes aren't really complete, and in particular things like `\ht0` don't work.
- Possibly useful to override (pre-override?) a macro defined in the source file; that is, define it and silently ignore the definition given in the source.
- ...um, ... *documentation!*

Appendix A

Command Documentation

latexml

Transforms a TeX/LaTeX file into XML.

Synopsis

latexml [options] texfile

Options:

<code>--destination=file</code>	specifies destination file (default stdout).
<code>--output=file</code>	[obsolete synonym for <code>--destination</code>]
<code>--preload=module</code>	requests loading of an optional module; can be repeated
<code>--includestyles</code>	allows latexml to load raw *.sty file; by default it avoids this.
<code>--path=dir</code>	adds dir to the paths searched for files, modules, etc;
<code>--documentid=id</code>	assign an id to the document root.
<code>--quiet</code>	suppress messages (can repeat)
<code>--verbose</code>	more informative output (can repeat)
<code>--strict</code>	makes latexml less forgiving of errors
<code>--xml</code>	requests xml output (default).
<code>--tex</code>	requests TeX output after expansion.
<code>--box</code>	requests box output after expansion and digestion.
<code>--noparse</code>	suppresses parsing math
<code>--nocomments</code>	omit comments from the output
<code>--inputencoding=enc</code>	specify the input encoding.
<code>--VERSION</code>	show version number.
<code>--debug=package</code>	enables debugging output for the named package

`--help` shows this help message.

If texfile is '-', latexml reads the TeX source from standard input.

Options & Arguments

`--destination=file`

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

`--preload=module`

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

`--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.

--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](#), [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:

<code>--verbose</code>	shows progress during processing.
<code>--VERSION</code>	show version number.
<code>--help</code>	shows help message.
<code>--sourcedirectory=sourcedir</code>	specifies directory of source TeX file.
<code>--validate, --novalidate</code>	Enables (the default) or disables validation of the source xml.
<code>--format=html xhtml+xml</code>	requests the output format.
<code>--destination=file</code>	specifies output file (and directory).
<code>--omitdoctype</code>	omits the Doctype declaration,
<code>--noomitdoctype</code>	disables the omission (the default)
<code>--numbersections</code>	enables (the default) the inclusion of section numbers in titles and crossrefs.
<code>--nonumbersections</code>	disables the above
<code>--stylesheet=xslfile</code>	requests the XSL transform using the given xslfile as stylesheet.
<code>--css=cssfile</code>	adds a css stylesheet to html/xhtml (can be repeated)
<code>--nodefaultcss</code>	disables inclusion of default css stylesheet
<code>--split</code>	requests splitting each document
<code>--nosplit</code>	disables the above (default)
<code>--splitat</code>	specifies level to split the document
<code>--splitpath=xpath</code>	specifies xpath expression for splitting (default is section-like, if splitting)
<code>--splitnaming=(id idrelative label labelrelative)</code>	specifies how to name split files (def. idrelative).
<code>--scan</code>	scans documents to extract ids, labels, section titles, etc. (default)
<code>--noscan</code>	disables the above
<code>--crossref</code>	fills in crossreferences (default)
<code>--nocrossref</code>	disables the above
<code>--urlstyle=(server negotiated file)</code>	format to use for urls (default server).
<code>--index</code>	requests filling in the index (default)
<code>--noindex</code>	disables the above
<code>--splitindex</code>	Splits the index into pages per initial.
<code>--nosplitindex</code>	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
--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
--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 LaTeXXML 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 LaTeXXML'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 right. `-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.

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 its 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 its 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 LaTeXXML 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 LaTeXXML doctype.

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

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

`--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`).

`--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.

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’.

--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).

--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](#), [LaTeXML](#)

Appendix B

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`.

```
$string = $latexml->DOMtoString($doc);
```

Converts the XML document to a string (of utf8 bytes).

```
$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` representation) 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`, `LaTeXML::Font`

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 usable string from tokens or boxes that might representing a filename or such.

`$boole = $object->>equals($other);`

Returns whether `$object` and `$other` are equal. Should perform a deep comparison, but the default implementation just compares for object identity.

`$boole = $object->isaToken;`

Returns whether `$object` is an `LaTeXML::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 primarily carried out for side effect during digestion in the `LaTeXML::Stomach` and for 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 specialized 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::Constructor` records the arguments used in the invocation 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 ctable, returning a `LaTeXML::Tokens`.

```
$tokens = TokenizeInternal($string);
```

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

```
@tokens = Explode($string);
```

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

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 a 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 short 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; most useful for Tokens or Boxes where the string content is desired. 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.

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.

```
$line = LaTeXML::Error::line_in_file($file);
```

This returns the line number in `$file` that is currently being executed, assuming that some stackframe is invoking code defined in that file.

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{#1}', "<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{#1}', '\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 imitation of TeX's digestive tract.

The first argument to each of these defining forms (`DefMacro`, `DefPrimitive`, 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.

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*

Reads tokens until a match to the tokens *match* is found, returning the tokens preceding the match. This corresponds to TeX delimited arguments.

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.

Variable

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

SkipSpaces

Skips any space tokens, 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 `$prototype`. If a `$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($prototype,CODE($stomach,@args),%options);
```

Define a primitive control sequence. These are usually done for side effect and so `CODE` should end with `return;`, but can also return a list of digested items.

The only option is for the special case: `isPrefix=>1` is used for 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 LaTeXXML 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 absorption 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 constructor *begins* with `^`, the XML fragment is allowed to *float up* to a parent node that is allowed to contain it, according to the Document Type.

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

DefConstructor options are

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](#) If the font change is to only apply to this construct, you would also use `<bounded=1>>`.

reversion=>\$textstring 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, occurrences 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

if true, arbitrary following material will be accumulated into a 'body' until the current grouping level is reverted. 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 inferred directly from the \$prototype (eg. when more args are explicitly read by Daemons).

scope=>\$scope

See [/scope](#).

```
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](#).

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 `<nogroup=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 when using this counter
           to generate ID's.
nested    Not sure that this is even sane.
```

`$num = CounterValue($ctr);`

Fetches the value associated with the counter `$ctr`.

`$tokens = StepCounter($ctr);`

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

`$keys = RefStepCounter($ctr);`

Like `\refstepcounter`, it steps the counter and returns the keys `refnum=$refnum, id=>$id>`, making 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);`

Analogous 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`).

`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 simplified. 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 sequence of key/value pairs with the initial specs successively narrowing the selection of document nodes, and the remaining specs indicating how to modify or replace the selected nodes.

The following select portions of the document:

label =>\$label

Selects the part of the document with label=\$label

scope =>\$scope

The \$scope could be "label:foo" or "section:1.2.3" or something similar. These select a subtree labelled 'foo', or a section with reference number "1.2.3"

xpath =>\$xpath

Select those nodes matching an explicit xpath expression.

match =>\$TeX

Selects nodes that look like what the processing of \$TeX would produce.

regexp=>\$regexp

Selects text nodes that match the regular expression.

The following act upon the selected node:

attributes =>\$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 digests 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.

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

Reads an argument delimited by braces, while disabling most TeX catcodes.

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:

<code>VERBOSITY</code>	: the level of verbosity for debugging output, with 0 being default.
<code>STRICT</code>	: whether errors (eg. undefined macros) are fatal.
<code>INCLUDE_COMMENTS</code>	: whether to preserve comments in the source, and to add occasional line number comments. (Default true).
<code>PRESERVE_NEWLINES</code>	: whether newlines in the source should be preserved (not 100% TeX-like). By default this is true.
<code>SEARCHPATHS</code>	: 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 `$label` of disallowed characters, and prepends `$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.

```
$defn = $token->getDefinition;
```

Return the current definition associated with \$token in \$STATE, or undef if none.

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 appropriate.

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 the `$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 extensions `.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 its 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 resulting 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 from the 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::Boxes` 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 `$font`, performing any required automatic opening and closing of intermediate nodes (including those needed for font changes) and inserting the string `$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 intermediate 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 intermediate 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), 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);
```

Return the prefix to use for the given `$namespace`.

```
$model->getNamespace($prefix);
```

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.

Possible 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 `XMap` node; 0 is the operator node.

```
Annotate($node,%attributes);
```

Add attributes to `$node`.

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

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

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

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

Given a set of expressions, construct a `Collection` 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.

Appendix C

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 utilities for LaTeXML. A pathname is represented by a simple string.

Pathname Manipulations

```
$path = pathname_make(%pieces);
```

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 path-name.

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

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

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

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

File System Operations

```
$mtime = pathname_timestamp($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);
```

This performs the same operation as `pathname_find`, but returns all matching paths that exist.

Appendix D

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 E

L^AT_EX_{XML} Schema

The document type used by L^AT_EX_{XML} 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

`LaTeXML-common` included.

`LaTeXML-inline` included.

`LaTeXML-block` included.

`LaTeXML-para` included.

`LaTeXML-math` included.

`LaTeXML-tabular` included.

`LaTeXML-picture` included.

`LaTeXML-structure` included.

`LaTeXML-bib` included.

`Inline.model` Combined model for inline content.

`== (#PCDATA | Inline.class | Misc.class | Meta.class)*`

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

`== (Block.class | Misc.class | Meta.class)*`

Flow.model Combined model for general flow containing both inline and block level content.

```
== (#PCDATA | Inline.class | Block.class | Misc.class
| Meta.class)*
```

Para.model Combined model for logical block-level context.

```
== (Para.class | Meta.class)*
```

Start == **document**

Module LaTeXML-common

Inline.class All strictly inline elements.

```
==()
```

Block.class All ‘physical’ block elements. A physical block is typically displayed as a block, but may not constitute a complete logical unit.

```
==()
```

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

```
==()
```

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.

```
==()
```

Meta.class All metadata elements, typically representing hidden data.

```
==()
```

Length.type The type for attributes specifying a length. Should be a number followed by a length, typically px. NOTE: To be narrowed later.

```
==()
```

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

```
==()
```

Common.attributes Attributes shared by ALL elements.

```
== Attributes:
```

class 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.]

= *NMTOKENS*

ID.attributes Attributes for elements that can be cross-referenced from inside or outside the document.

== Attributes:

xml:id the unique identifier of the element, usually generated automatically by the latexml.

= *ID*

IDREF.attributes Attributes for elements that can cross-reference other elements.

== Attributes:

idref the identifier of the referred-to element.

= *IDREF*

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

== Attributes:

ID.attributes included

labels 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!)

= *text*

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

= *text*

Positionable.attributes Attributes shared by low-level, generic inline and block elements that can be sized or shifted.

== Attributes:

width the desired width of the box
 = `Length.type`
height the desired height of the box
 = `Length.type`
depth the desired depth of the box
 = `Length.type`
pad-width extra width beyond the boxes natural size.
 = `Length.type`
pad-height extra height beyond the boxes natural size.
 = `Length.type`
xoffset horizontal shift the position of the box.
 = `Length.type`
yoffset vertical shift the position of the box.
 = `Length.type`
align alignment of material within the box.
 = ('left' | 'center' | 'right' | 'justified')
vattach specifies which line of the box is aligned to the baseline of
 the containing object.
 = ('top' | 'middle' | 'bottom')

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

== Attributes:

imagesrc the file, possibly generated from other data.
 = *anyURI*
imagewidth the width in pixels of **imagesrc**.
 = *nonNegativeInteger*
imageheight the height in pixels of **imagesrc**.
 = *nonNegativeInteger*
description a description of the image
 = *text*

Module LaTeXML-inline

Inline.class The inline module defines basic inline elements used
 throughout

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

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

`|= (note | indexmark | ERROR)`

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

attributes:

`Common.attributes, Positionable.attributes` included

font the font to use (describe!)

`= text`

size the text size to use (describe!)

`= text`

color the color to use; any CSS compatible color specification.

`= text`

framed the kind of frame or outline for the text.

`= ('rectangle' | 'underline')`

content: `Inline.model`

emph Emphasized text.

attributes: `Common.attributes`

content: `Inline.model`

acronym Represents an acronym.

attributes:

`Common.attributes` included

name should be used to indicate the expansion of the acronym.

`= text`

content: `Inline.model`

rule A Rule.

attributes: `Common.attributes, Positionable.attributes`

content: `empty`

ref A hyperlink reference to some other object. When converted to HTML, the content would be the content of the anchor. The destination can be specified by one of the attributes `labelref`, `idref` or `href`; Missing fields will usually be filled in during postprocessing, based on data extracted from the document(s).

attributes:

Common.attributes, IDREF.attributes included

labelref reference to a LaTeX labelled object.

= *text*

href reference to an arbitrary url.

= *text*

show 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.

= *text*

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

= *text*

content: **Inline.model**

anchor Inline anchor.

attributes: **Common.attributes, ID.attributes**

content: **Inline.model**

cite A container for a bibliographic citation. The model is inline to allow arbitrary comments before and after the expected **bibref**(s) which are the specific citation.

attributes: **Common.attributes**

content: **Inline.model**

bibref A bibliographic citation referring to a specific bibliographic item.

attributes:

Common.attributes, IDREF.attributes included

bibrefs a comma separated list of bibliographic keys.

= *text*

show a pattern encoding how the text content (of an empty bibref) will be filled in. Consists of strings **author**, **year** and **title** (to be replaced by data from the bibliographic item) mixed with arbitrary characters.

= *text*

content: **Inline.model**

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

attributes:

`Common.attributes` included

mark indicates the desired visible marker to be linked to the note.
= *text*

content: `Flow.model`

ERROR error object for undefined control sequences, or whatever

attributes: `Common.attributes`

content: `#PCDATA*`

indexmark Metadata to record an indexing position. The content is a sequence of `indexphrase`, each representing a level in a multilevel indexing entry.

attributes:

`Common.attributes` included

see_also a flattened form (like `key`) of another `indexmark`, used to crossreference.
= *text*

style NOTE: describe this.
= *text*

content: `indexphrase*`

indexphrase A phrase within an `indexmark`

attributes:

`Common.attributes` included

key a flattened form of the phrase for generating an ID.
= *text*

content: `Inline.model`

Module LaTeXML-block

Block.class The block module defines the following ‘physical’ block elements.

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

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

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

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

`== constraint`

`p` A physical paragraph.

attributes: `Common.attributes`

content: `Inline.model`

`centering` A physical block that centers its content. NOTE: Reconsider this; perhaps should be a property on other blocks?

attributes: `Common.attributes`

content: `(caption | tocaption | Block.model)*`

`constraint` A constraint upon an equation.

attributes:

`hidden`

`= boolean`

content: `Inline.model`

`equation` An Equation. The model is just `Inline` which includes `Math`, the main expected ingredient. However, other things can end up in `display math`, too, so we use `Inline`. Note that `tabular` is here only because it's a common, if misguided, idiom; the processor will lift such elements out of `math`, when possible

attributes: `Common.attributes, Labelled.attributes`

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

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

attributes: `Common.attributes, Labelled.attributes`

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

`MathFork` A wrapper for `Math` that provides alternative, but typically less semantically meaningful, formatted representations. The first child is the meaningful form, the extra children provide formatted forms, for example being table rows or cells arising from an `eqnarray`.

attributes: `Common.attributes`

content: `Math MathBranch*`

MathBranch A container for an alternatively formatted math representation.

attributes:

`Common.attributes` included

`format`

= *text*

content: (`Math` | `tr` | `td`)*

quote A quotation.

attributes: `Common.attributes`

content: `Inline.model`

block A generic block (fallback).

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

content: `Inline.model`

break A forced line break.

attributes: `Common.attributes`

content: *empty*

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

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

content: `Inline.model`

verbatim Verbatim content

attributes:

`Common.attributes` included

`font` the font to use; generally typewriter.

= *text*

content: `Inline.model`

itemize An itemized list.

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

content: `item`*

enumerate An enumerated list.

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

content: `item`*

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

attributes: **Common.attributes**, **ID.attributes**

content: **item***

item An item within a list.

attributes: **Common.attributes**, **Labelled.attributes**

content: **tag?** **Block.model**

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

attributes:

Common.attributes included

open specifies an open delimiters used to display the tag.
= *text*

close specifies an close delimiters used to display the tag.
= *text*

content: **Inline.model**

graphics A graphical insertion of an external file.

attributes:

Common.attributes, **Imageable.attributes** included

graphic the path to the graphics file
= *text*

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

content: *empty*

Module LaTeXML-para

Para.class This module ‘logical’ block elements.

|= (**para** | **theorem** | **proof** | **figure** | **table**)

para A Logical paragraph. It has an **id**, but not a **label**.

attributes: **Common.attributes**, **ID.attributes**

content: **Block.model**

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

attributes: `Common.attributes, Labelled.attributes`

content: `title? Block.model`

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

attributes: `Common.attributes, Labelled.attributes`

content: `title? Block.model`

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?

== `(caption | tocaption)`

figure A figure, possibly captioned.

attributes:

`Common.attributes, Labelled.attributes` included

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

`= text`

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

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

attributes:

`Common.attributes, Labelled.attributes` included

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

`= text`

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

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

attributes: `Common.attributes`

content: `Inline.model`

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

attributes: `Common.attributes`

content: `Inline.model`

Module LaTeXML-math

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**

Math.class This class defines the content of the **Math** element. Additionally, it could contain MathML or OpenMath, after postprocessing.

== **XMath**

XMath.class These elements comprise the internal math representation, being the content of the **XMath** element.

== (**XMApp** | **XMTok** | **XMRef** | **XMHint** | **XMArg** | **XMWrap** | **XMDual** | **XMText** | **XMArray**)

Math Outer container for all math. This holds the internal **XMath** representation, as well as image data and other representations.

attributes:

Common.attributes, **Imageable.attributes** included

mode display or inline mode.

= ('display' | 'inline')

tex reconstruction of the \TeX that generated the math.

= *text*

content-tex more semantic version of **tex**.

= *text*

text a textified representation of the math.

= *text*

content: **Math.class***

XMath.attributes

== Attributes:

role The role that this item plays in the Grammar.

= *text*

open an open delimiter to enclose the object;

= *text*

close an close delimiter to enclose the object;

= *text*

argopen an open delimiter to enclose the argument list, when this token is applied to arguments with **XMApP**.

= *text*

argclose a close delimiter to enclose the argument list, when this token is applied to arguments with **XMApP**.

= *text*

separators 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.

= *text*

punctuation trailing (presumably non-semantic) punctuation.

= *text*

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

= *text*

XMath Internal representation of mathematics.

attributes: **Common.attributes**

content: **XMath.class***

XMTok General mathematical token.

attributes:

Common.attributes, **XMath.attributes**, **ID.attributes** included

name The name of the token, typically the control sequence that created it.

= *text*

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

= *text*

omcd The OpenMath CD for which **meaning** is a symbol.

= *text*

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

= *text*

font The font, size a used for the symbol.

= *text*

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

= *text*

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

= *text*

scriptpos 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 its 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?

= *text*

thickness NOTE: How is this used?

= *text*

content: #PCDATA*

XMApp Generalized application of a function, operator, whatever (the first child) to arguments (the remaining children). The attributes are a subset of those for **XMTok**.

attributes:

Common.attributes, **XMath.attributes**, **ID.attributes** included

name The name of the token, typically the control sequence that created it.

= *text*

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

= *text*

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

= *text*

content: **XMath.class***

XMDual Parallel markup of content (first child) and presentation (second child) of a mathematical object. Typically, the arguments are shared between the two branches: they appear in the content branch, with **id**'s, and **XMRef** is used in the presentation branch

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

content: **XMath.class XMath.class**

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

attributes:

- Common.attributes**, **XMath.attributes**, **ID.attributes** included
- name**
= *text*
- meaning**
= *text*
- style**
= *text*
- content:** *empty*
- XMText** Text appearing within math.
- attributes:** **Common.attributes**, **XMath.attributes**, **ID.attributes**
- content:** (**#PCDATA** | **Inline.class** | **Misc.class**)*
- XMWrap** Wrapper for a sequence of tokens used to assert the role of the contents in its parent. This element generally disappears after parsing. The attributes are a subset of those for **XMTok**.
- attributes:**
- Common.attributes**, **XMath.attributes**, **ID.attributes** included
- name**
= *text*
- meaning** A more semantic name corresponding to the intended meaning, such as the OpenMath name.
= *text*
- style**
= *text*
- content:** **XMath.class***
- XMArg** Wrapper for an argument to a structured macro. It implies that its content can be parsed independently of its parent, and thus generally disappears after parsing.
- attributes:**
- Common.attributes**, **XMath.attributes**, **ID.attributes** included
- rule**
= *text*
- content:** **XMath.class***
- XMRef** Structure sharing element typically used in the presentation branch of an **XMDual** to refer to the arguments present in the content branch.
- attributes:** **Common.attributes**, **XMath.attributes**, **ID.attributes**, **IDREF.attributes**
- content:** *empty*

XMArray Math Array/Alignment structure.

attributes:

Common.attributes, **XMath.attributes**, **ID.attributes** included

name

= *text*

meaning

= *text*

style

= *text*

vattach

= ('top' | 'bottom')

width

= *text*

content: **XMRow***

XMRow A row in a math alignment.

attributes: **Common.attributes**

content: **XMCell***

XMCell A cell in a row of a math alignment.

attributes:

Common.attributes included

colspan indicates how many columns this cell spans or covers.

= *nonNegativeInteger*

rowpan indicates how many rows this cell spans or covers.

= *nonNegativeInteger*

align specifies the alignment of the content.

= *text*

width specifies the desired width for the column.

= *text*

border 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.

= *text*

thead whether this cell corresponds to a table head or foot.

= *boolean*

content: **XMath.class***

Module LaTeXML-tabular

`Misc.class` This module defines the basic tabular, or alignment, structure. It is roughly parallel to the HTML model.

`|= tabular`

`tabular` An alignment structure corresponding to tabular or various similar forms. The model is basically a copy of HTML4's table.

attributes:

`Common.attributes` included

`vattach` which row's baseline aligns with the container's baseline.

= ('top' | 'middle' | 'bottom')

`width` the desired width of the tabular.

= `Length.type`

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

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

attributes: `Common.attributes`

content: `tr`*

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

attributes: `Common.attributes`

content: `tr`*

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

attributes: `Common.attributes`

content: `tr`*

`tr` A row of a tabular.

attributes: `Common.attributes`

content: `td`*

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

attributes:

`Common.attributes` included

`colspan` indicates how many columns this cell spans or covers.

= `nonNegativeInteger`

rowspan indicates how many rows this cell spans or covers.
 = *nonNegativeInteger*

align specifies the alignment of the content.
 = *text*

width specifies the desired width for the column.
 = **Length.type**

border 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.
 = *text*

thead whether this cell corresponds to a table head or foot.
 = *boolean*

content: **Flow.model**

Module LaTeXML-picture

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**

Picture.class

== (**g** | **rect** | **line** | **circle** | **path** | **arc** | **wedge** | **ellipse**
 | **polygon** | **bezier**)

Picture.attributes These attributes correspond roughly to SVG, but need documentation.

== Attributes:

x
 = *text*

y
 = *text*

r
 = *text*

rx
 = *text*

ry
 = *text*

width
 = *text*

```

height
  = text
fill
  = text
stroke
  = text
stroke-width
  = text
stroke-dasharray
  = text
transform
  = text
terminators
  = text
arrowlength
  = text
points
  = text
showpoints
  = text
displayedpoints
  = text
arc
  = text
angle1
  = text
angle2
  = text
arcsepA
  = text
arcsepB
  = text
curvature
  = text

```

`PictureGroup.attributes` These attributes correspond roughly to SVG, but need documentation.

== Attributes:

```

pos
  = text
framed
  = boolean
frametype
  = ('rect' | 'circle' | 'oval')
fillframe
  = boolean
boxsep
  = text
shadowbox
  = boolean
doubleline
  = boolean

```

picture A picture environment.

attributes:

Common.attributes, Picture.attributes, Imageable.attributes
included

```
clip
  = boolean
```

```
baseline
  = text
```

```
unitlength
  = text
```

```
xunitlength
  = text
```

```
yunitlength
  = text
```

```
tex
  = text
```

```
content-tex
  = text
```

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

g A graphical grouping; the content inherits by the transformations, positioning and other properties.

attributes: `Common.attributes`, `Picture.attributes`,
`PictureGroup.attributes`

content: (`Picture.class` | `Inline.class` | `Misc.class`
| `Meta.class`)*

rect A rectangle within a `picture`.

attributes: `Common.attributes`, `Picture.attributes`

content: *empty*

line A line within a `picture`.

attributes: `Common.attributes`, `Picture.attributes`

content: *empty*

polygon A polygon within a `picture`.

attributes: `Common.attributes`, `Picture.attributes`

content: *empty*

wedge A wedge within a `picture`.

attributes: `Common.attributes`, `Picture.attributes`

content: *empty*

arc An arc within a `picture`.

attributes: `Common.attributes`, `Picture.attributes`

content: *empty*

circle A circle within a `picture`.

attributes: `Common.attributes`, `Picture.attributes`

content: *empty*

ellipse An ellipse within a `picture`.

attributes: `Common.attributes`, `Picture.attributes`

content: *empty*

path A path within a `picture`.

attributes: `Common.attributes`, `Picture.attributes`

content: *empty*

bezier A bezier curve within a `picture`.

attributes: `Common.attributes`, `Picture.attributes`

content: *empty*

Module LaTeXML-structure

document The document root.

attributes: `Common.attributes, Labelled.attributes`
content: `(FrontMatter.class | SectionalFrontMatter.class)*`
`Para.model paragraph* section* chapter* part* BackMatter.class*`

part A part within a document.

attributes: `Common.attributes, Labelled.attributes`
content: `SectionalFrontMatter.class* Para.model chapter*`

chapter A Chapter within a document.

attributes: `Common.attributes, Labelled.attributes`
content: `SectionalFrontMatter.class* Para.model subparagraph*`
`paragraph* subsection* section*`

section A Section within a document.

attributes: `Common.attributes, Labelled.attributes`
content: `SectionalFrontMatter.class* Para.model subparagraph*`
`paragraph* subsection*`

appendix An Appendix within a document.

attributes: `Common.attributes, Labelled.attributes`
content: `SectionalFrontMatter.class* Para.model subparagraph*`
`paragraph* subsection* section*`

subsection A Subsection within a document.

attributes: `Common.attributes, Labelled.attributes`
content: `SectionalFrontMatter.class* Para.model subparagraph*`
`paragraph* subsubsection*`

subsubsection A Subsubsection within a document.

attributes: `Common.attributes, Labelled.attributes`
content: `SectionalFrontMatter.class* Para.model subparagraph*`
`paragraph*`

paragraph A Paragraph within a document. This corresponds to a ‘formal’ marked, possibly labelled LaTeX Paragraph, in distinction from an unlabelled logical paragraph.

attributes: `Common.attributes, Labelled.attributes`
content: `SectionalFrontMatter.class* Para.model subparagraph*`

- subparagraph** A Subparagraph within a document.
attributes: `Common.attributes, Labelled.attributes`
content: `SectionalFrontMatter.class* Para.model`
- bibliography** A Bibliography within a document.
attributes:
`Common.attributes, Labelled.attributes` included
files the list of bib files used to create the bibliography.
= *text*
content: `FrontMatter.class* SectionalFrontMatter.class*`
`biblist*`
- index** An Index within a document.
attributes: `Common.attributes, Labelled.attributes`
content: `SectionalFrontMatter.class* indexlist*`
- indexlist** An index generated from the collection of `indexmark` in a document (or document collection).
attributes: `Common.attributes, ID.attributes`
content: `indexentry*`
- indexentry** An entry in an `indexlist` consisting of a phrase, references to points in the document where the phrase was found, and possibly a nested `indexlist` represented index levels below this one.
attributes: `Common.attributes, ID.attributes`
content: `indexphrase indexrefs? indexlist?`
- indexrefs** A container for the references (`ref`) to where an `indexphrase` was encountered in the document. The model is `Inline` to allow arbitrary text, in addition to the expected `ref`'s.
attributes: `Common.attributes`
content: `Inline.model`
- title** The title of a document, section or similar document structure container.
attributes: `Common.attributes`
content: `Inline.model`
- toctitle** The short form of a title, for use in tables of contents or similar.
attributes: `Common.attributes`
content: `Inline.model`

subtitle A subtitle, or secondary title.
 attributes: `Common.attributes`
 content: `Inline.model`

personname A person's name.
 attributes: `Common.attributes`
 content: `Inline.model`

creator Generalized document creator.
 attributes:
 `Common.attributes` included
 role indicates the role of the person in creating the document. Values include author, editor and translator, but is open-ended to support extension.
 = *text*
 content: `(Person.class | Misc.class)*`

contact Generalized contact information for a document creator.
 attributes:
 `Common.attributes` included
 role indicates the type of contact information contained. Values include address, current_address, affiliation, thanks, email, url, dedicatory to cover various common constructs, but is open-ended to support extension.
 = *text*
 content: `Inline.model`

date Generalized document date.
 attributes:
 `Common.attributes` included
 role indicates the relevance of the date to the document. Values include creation, but is open-ended to support extension.
 = *text*
 content: `Inline.model`

abstract A document abstract.
 attributes: `Common.attributes`
 content: `Block.model`

acknowledgements Acknowledgements for the document.

attributes: `Common.attributes`

content: `Inline.model`

keywords Keywords for the document. The content is freeform.

attributes: `Common.attributes`

content: `Inline.model`

classification A classification of the document.

attributes:

`Common.attributes` included

scheme indicates what classification scheme was used.

= *text*

content: `Inline.model`

Person.class

== (`personname` | `contact`)

SectionalFrontMatter.class

== (`title` | `toctitle` | `creator`)

FrontMatter.class

== (`subtitle` | `date` | `abstract` | `acknowledgements` | `keywords`
| `classification`)

BackMatter.class

== (`bibliography` | `appendix` | `index`)

Module LaTeXML-bib

biblist A list of bibliographic `bibentry` or `bibitem`.

attributes: `Common.attributes`

content: (`bibentry` | `bibitem`)*

bibentry Semantic representation of a bibliography entry, typically resulting from parsing BibTeX

attributes:

`Common.attributes`, `ID.attributes` included

key

= *text*

```
    type
      = text
    content: Bibentry.class*
bib-author    Author of a bibliographic entry.
    attributes: Common.attributes
    content: Bibname.model
bib-editor    Editor of a bibliographic entry.
    attributes: Common.attributes
    content: Bibname.model
bib-translator  Translator of a bibliographic entry.
    attributes: Common.attributes
    content: Bibname.model
surname      Surname of an author, editor or translator.
    content: Inline.model
givenname    Given name of an author, editor or translator.
    content: Inline.model
initials     Initials of an author, editor or translator.
    content: Inline.model
lineage      Lineage of an author, editor or translator. (eg. von)
    content: Inline.model
bib-title     Title of a bibliographic entry.
    attributes: Common.attributes
    content: Inline.model
bib-subtitle  Subtitle of a bibliographic entry.
    attributes: Common.attributes
    content: Inline.model
bib-booktitle  Title of the book containing a bibliographic entry.
    attributes: Common.attributes
    content: Inline.model
bib-key      Unique key of a bibliographic entry.
    attributes: Common.attributes
    content: Inline.model
```

- bib-journal Journal of a bibliographic entry.
 attributes: `Common.attributes`
 content: `Inline.model`
- bib-series Series of a bibliographic entry.
 attributes: `Common.attributes`
 content: `Inline.model`
- bib-conference Conference of a bibliographic entry.
 attributes: `Common.attributes`
 content: `Inline.model`
- bib-publisher Publisher of a bibliographic entry.
 attributes: `Common.attributes`
 content: `Inline.model`
- bib-organization Organization responsible for a bibliographic entry.
 attributes: `Common.attributes`
 content: `Inline.model`
- bib-institution Institution responsible for a bibliographic entry.
 attributes: `Common.attributes`
 content: `Inline.model`
- bib-address Address of party responsible for a bibliographic entry.
 attributes: `Common.attributes`
 content: `Inline.model`
- bib-volume Volume of a bibliographic entry.
 attributes: `Common.attributes`
 content: `Inline.model`
- bib-number Number of a bibliographic entry.
 attributes: `Common.attributes`
 content: `Inline.model`
- bib-pages Pages of a bibliographic entry.
 attributes: `Common.attributes`
 content: `Inline.model`
- bib-part Part of a bibliographic entry.
 attributes: `Common.attributes`
 content: `Inline.model`

- bib-date** Date of a bibliographic entry.
attributes: `Common.attributes`
content: `Inline.model`
- bib-edition** Edition of a bibliographic entry.
attributes: `Common.attributes`
content: `Inline.model`
- bib-status** Status of a bibliographic entry.
attributes: `Common.attributes`
content: `Inline.model`
- bib-type** Type of a bibliographic entry.
attributes: `Common.attributes`
content: `Inline.model`
- bib-identifier** Some form of document identifier. The content is descriptive.
attributes:
`Common.attributes` included
scheme indicates what sort of identifier it is: such as `doi`, `issn`, `isbn`, `mr`, or others.
 = *text*
id the identifier.
 = *text*
href a url to the document, if available
 = *text*
content: `Inline.model`
- bib-review** Review of a bibliographic entry. The content is descriptive.
attributes:
`Common.attributes` included
scheme indicates what sort of identifier it is: such as `doi`, `issn`, `isbn`, `mr`, or others.
 = *text*
id the identifier.
 = *text*
href a url to the review, if available
 = *text*

content: `Inline.model`

bib-links Links to other things like preprints, source code, etc.
attributes: `Common.attributes`
content: `Inline.model`

bib-language Language of a bibliographic entry.
attributes: `Common.attributes`
content: `Inline.model`

bib-url A URL for a bibliographic entry. The content is descriptive
attributes:
`Common.attributes` included
href
= *text*
content: `Inline.model`

bib-note Notes about a bibliographic entry.
attributes: `Common.attributes`
content: `Inline.model`

bibitem A formatted bibliographic item, typically as written explicit in a LaTeX article. This has generally lost most of the semantics present in the BibTeX data.
attributes:
`Common.attributes`, `ID.attributes` included
key
= *text*
content: `tag? bibblock*`

bibblock A block of data appearing within a `bibitem`.
content: `Inline.model`

Bibentry.class

```

== (bib-author | bib-editor | bib-translator | bib-title
| bib-subtitle | bib-booktitle | bib-key | bib-journal
| bib-series | bib-conference | bib-publisher
| bib-organization | bib-institution | bib-address
| bib-volume | bib-number | bib-pages | bib-part | bib-date
| bib-edition | bib-status | bib-type | bib-language | bib-url
| bib-note | bib-identifier | bib-review | bib-links)

```

`Bibname.model` The content model of the bibliographic name fields
(`bib-author`, `bib-editor`, `bib-translator`)
== `surname` `givenname?` `initials?` `lineage?`

Index

- LaTeXML, [33](#)
 - Customization, [34](#)
 - Description, [33](#)
 - Methods, [33](#)
 - See also, [35](#)
 - Synopsis, [33](#)
- latexml, [23](#)
 - Options & Arguments, [24](#)
 - See also, [25](#)
 - Synopsis, [23](#)
 - usage, [4](#)
- LaTeXML::Box, [67](#)
 - Box Methods, [68](#)
 - Common Methods, [67](#)
 - Description, [67](#)
 - Whatsit Methods, [68](#)
- LaTeXML::Definition, [37](#)
 - Description, [37](#)
 - Methods in general, [37](#)
 - More about Constructors, [39](#)
 - More about Primitives, [38](#)
 - More about Registers, [38](#)
- LaTeXML::Document, [80](#)
 - Accessors, [80](#)
 - Construction Methods, [81](#)
 - Description, [80](#)
- LaTeXML::Error, [43](#)
 - Description, [43](#)
 - Functions, [43](#)
- LaTeXML::Font, [72](#)
 - Description, [72](#)
 - LaTeXML::MathFont, [72](#)
- LaTeXML::Global, [40](#)
 - Description, [40](#)
 - Error Reporting, [42](#)
 - Generic functions, [42](#)
 - Global state, [40](#)
- Numbers, etc., [41](#)
- Synopsis, [40](#)
- Tokens, [40](#)
- LaTeXML::Gullet, [75](#)
 - Description, [75](#)
 - High-level methods, [77](#)
 - Low-level methods, [75](#)
 - Managing Input, [75](#)
 - Mid-level methods, [76](#)
- LaTeXML::MathParser, [87](#)
 - Convenience functions, [87](#)
 - Description, [87](#)
 - Math Representation, [87](#)
 - Possible Customizations, [87](#)
- LaTeXML::Model, [83](#)
 - Description, [83](#)
 - Document Type, [83](#)
 - Model Creation, [83](#)
 - Model queries, [84](#)
 - Namespaces, [83](#)
 - Rewrite Rules, [85](#)
 - Tag Properties, [84](#)
- LaTeXML::Mouth, [73](#)
 - Creating Mouths, [73](#)
 - Description, [73](#)
 - Methods, [73](#)
- LaTeXML::Number, [70](#)
 - Common methods, [70](#)
 - Description, [70](#)
 - Numerics methods, [71](#)
- LaTeXML::Object, [36](#)
 - Description, [36](#)
 - Methods, [36](#)
- LaTeXML::Package, [44](#)
 - Access to State, [58](#)
 - Argument Readers, [57](#)
 - Class and Packages, [52](#)

- Constructors, 48
- Control of Scoping, 47
- Control Sequence Parameters, 45
- Control Sequences, 45
- Counters and IDs, 53
- Description, 45
- Document Model, 54
- Document Rewriting, 55
- Low-level Functions, 59
- Macros, 47
- Mid-Level support, 56
- Primitives, 47
- Synopsis, 44
- LaTeXML::Parameters, 60
 - Description, 60
 - Parameters Methods, 60
- LaTeXML::Post, 93
- LaTeXML::Rewrite, 86
 - Description, 86
 - Methods, 86
- LaTeXML::State, 62
 - Access to State and Processing, 62
 - Category Codes, 63
 - Definitions, 64
 - Description, 62
 - Named Scopes, 64
 - Scoping, 62
 - Values, 63
- LaTeXML::Stomach, 78
 - Description, 78
 - Digestion, 78
 - Grouping, 79
 - Modes, 79
- LaTeXML::Token, 65
 - Common methods, 65
 - Description, 65
 - Token methods, 65
 - Tokens methods, 66
- LaTeXML::Util::Pathname, 89
 - Description, 89
 - File System Operations, 90
 - Pathname Manipulations, 89
- latexmlpost, 26
 - Format Options, 28
 - General Options, 28
 - Graphics Options, 32
 - Math Options, 31
 - Options & Arguments, 28
 - See also, 32
 - Site & Crossreferencing Options, 29
 - Source Options, 28
 - Synopsis, 26
 - usage, 5
 - site, 8
 - split pages, 8