The LaTeXML Manual

A LaTeX to XML Converter;
Version 0.6.0

Bruce R. Miller

October 9, 2007

Contents

Chapter 1

Introduction

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

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

Design Goals The idealistic goals of LaTeXML are:

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

As these goals are not entirely practical, or even somewhat contradictory, they are implicitly modified by "as much as possible." Completely mimicing

[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 LaTeXML, along with common use cases and techniques. Chapter 3 describes the system architecture in some detail. Strategies for customization and implementation of new packages is described in Chapter 4. The special considerations for mathematics, including details of representation and how to improve the conversion, are covered in Chapter 5. An overview of outstanding issues and planned future improvements are given in Chapter 6. Finally, the Appendices 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 LaTeXml

The main commands provided by the LaTeXML system are

`latexml` for converting TeX 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 --destination=doca.xml doca
latexml --destination=docb.xml docb
              ...
latexmlpost --prescan --dbfile=my.db --destination=doca.xhtml
doca
latexmlpost --prescan --dbfile=my.db --destination=docb.xhtml
docb
              ...
latexmlpost --noscan --dbfile=my.db --destination=doca.xhtml
doca
latexmlpost --noscan --dbfile=my.db --destination=docb.xhtml
docb
              ...
```

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 TeX 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 TeX 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, `--preload=LaTeX.pool` can be useful to force LaTeX-mode if LaTeXML fails to recognize it. 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 LaTeX-specific commands, such as `\documentclass`.

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

`\usepackage{`*`package`*`}` (or related commands) loads *`package`*`.sty.ltxml`. Normally, LaTeXML will not attempt to read the *`package`*`.sty` file, as these often involve LaTeX internals meaningless to the generation of XML. This behavior can be overridden with the option

```
--includestyles
```

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

`\input{`*`file`*`}` loads the first *`file`*`.tex.ltxml`, *`file`*`.tex`, *`file`*`.ltxml` or *`file`* that is found.

Some of these modules (esp. `TeX` and `LaTeX`), are parts of the LaTeXML 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 inheritted 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 math and graphics are converted to png images; the `LaTeXML-html.xslt` stylesheet is used.

xhtml math is converted to Presentation MathML, other graphics to images; the `LaTeXML-xhtml.xslt` stylesheet 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 postprocessing stages.

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

```
--noindex
```

The index entries can be permuted with the option

```
--permutedindex
```

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

Bibliography Bibilographic data from BibTeX can be provided with the option

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

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`) converts to Presentation MathML
> `--contentmathml` (or `--cmml`) converts to Content MathML
> `--openmath` (or `--om`) converts to 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. included using `\includegraphics` from the `graphics` or `graphicx` packages) 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 stylesheet, or a different CSS stylesheet, the options

> `--stylesheet=`*`stylesheet`*`.xsl`
> `--css=`*`stylesheet`*`.css`

can be used. The `--css` option can be repeated to accumulate several stylesheets; for example, the distribution provides several `navbar-left.css`, `navbar-right.css`, `theme-blue.css` and `amsart.css`, in addition to the `core.css` stylesheet which is included by default.

To develop such stylesheets, a knowledge of the LaTeXML 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 interlinked pages. This split, carried out before scanning, is requested by

```
--splitat=level
```

where *level* is one of `chapter`, `section`, `subsection`, or `subsubsection`. For example, `section` would split the document into chapters (if any) and sections, along with separate bibliography, index and any appendices. The removed document nodes are replaced by a Table of Contents.

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

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

The relative foms create shorter names in subdirectories for each level of splitting. 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 TEX sources into a single interlinked site consisting of multiple pages and a composite index and bibliography. The games one must play with LATEX's aux files to satisfy cross-references between these documents are not covered here, but the situation is handled by LATEXML in the following fashion.

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

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

```
latexmlpost --prescan --dbfile=DB --dest=doci.xhtml doci
```

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=doci.xhtml doci
```

which skips the unnecessary scanning step.

Chapter 3

Architecture

Like TeX, LaTeXML is data-driven: the text and executable control sequences (ie. macros and primitives) in the source file (and any packages loaded) direct the processing. The user exerts control over the conversion, and customizes it, by providing alternative LaTeXML-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 TeX 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 TeX's scoping rules. The processing is broken into the following stages

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

Construction converts the resulting boxes into an XML DOM.

Rewriting applies rewrite rules to modify the DOM.

Math Parsing parses the tokenized mathematics.

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::Token`s from the `LaTeXML::Gullet`, which itself pulls to-

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

kens from the `LaTeXML::Mouth`. The `LaTeXML::Mouth` converts characters from the plain text input into tokens according to the current category codes 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::Constructor`s into `LaTeXML::Whatsit`s, and converting the remaining tokens into a recursive structure consisting of `LaTeXML::Box`es and `LaTeXML::List`s and of `LaTeXML::Box`es.

3.2 Construction

The main (intentional) deviation of LaTeXML's digestion from that of TeX is in the extension of control sequences to include `LaTeXML::Constructor`s responsible for constructing XML document fragments, and `LaTeXML::Whatsit`s 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 througout 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 processsing 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 LaTeXML 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, LaTeXML 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, LaTeXML 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.

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

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

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

Additional exceptions that need fuller explanation are:

- `LaTeXML::Constructor`s may wish to create a dual object (`XMDual`) whose children are the semantic and presentational forms.
- Spacing and similar markup generates `XMHint` elements, which are currently ignored during parsing, but probably shouldn't.

5.1 Math Details

LaTeXML processes mathematical material by proceeding through several stages:

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

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

5.1.1 Internal Math Representation

The `XMath` element is the container for the internal representation

The following attributes can appear on all `XM*` elements:

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

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

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

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

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

- `name` the name that represents the 'meaning' of the token; this overrides the content for identifying the token.
- `omcd` the OpenMath content dictionary that the name belongs to.
- `font` the font to be used for presenting the content.
- `style` ?
- `size` ?
- `stackscripts` whether scripts should be stacked above/below the item, instead of the usual script position.

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

- `name` the name that represents the meaning of the construct as a whole.
- `stackscripts` ?

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

`XMHint` represents spacing or other apparent purely presentation material.

- `name` names the effect that the hint was intended to achieve.
- `style` ?

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

- `name` ?
- `style` ?

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

`rule` the grammar rule that this subexpression should match.

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

`idref` the identifier of the referenced math subexpression.

5.1.2 Grammatical Roles

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

The following grammatical roles are recognized by the math parser. These values can be specified in the `role` attribute during the initial document construction or by rewrite rules. Although the precedence of operators is loosely described in the following, since the grammar contains various special case productions, no rigidly ordered precedence is given.

`ATOM` a general atomic subexpression.

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

`PUNCT` punctuation.

`APPLYOP` an explicit infix application operator (high precedence).

`RELOP` a relational operator, loosely binding.

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

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

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

`MULOP` a multiplicative operator, high precedence.

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

`OPEN` an open delimiter.

`CLOSE` a close delimiter.

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

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

`SUMOP` a summation/union operator.

`INTOP` an integral operator.

`LIMITOP` a limiting operator.

`DIFFOP` a differential operator.

`BIGOP` a general operator, but lower precedence, such as a P preceding an integral to denote the principal value. Note that `SUMOP`, `INTOP`, `LIMITOP`, `DIFFOP` and `BIGOP` are treated equivalently by the grammar, but are distinguished to facilitate (*eventually!*) analyzing the argument structure (eg bound variables and differentials within an integral). **Note** are `SUMOP` and `LIMITOP` significantly different in this sense?

`VERTBAR`

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

`NUMBER` a number.

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

`POSTSUBSCRIPT` Similar to `POSTSUPERSCRIPT` for subscripts.

`FLOATINGSUPERSCRIPT` A special case for a superscript on an empty base, ie. `{}^{x}`. This 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 LaTeX packages have not been implemented, and those that are are aren't necessarily complete.
- TeX 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:
--destination=file specifies destination file; default to stdout.
--output=file      [obsolete synonym for --destination]
--preload=module   requests loading of an optional module;
                   can be repeated
--includestyles    allows latexml to load raw *.sty file;
                   by default it avoids this.
--path=dir         adds dir to the paths searched for files,
                   modules, etc;
--documentid=id    assign an id to the document root.
--quiet            suppress messages (can repeat)
--verbose          more informative output (can repeat)
--strict           makes latexml less forgiving of errors
--xml              requests xml output (default).
--tex              requests TeX output after expansion.
--box              requests box output after expansion
                   and digestion.
--noparse          suppresses parsing math
--nocomments       omit comments from the output
--inputencoding=enc specify the input encoding.
--VERSION          show version number.
--debug=package    enables debugging output for the named
                   package
--help             shows this help message.
```

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

Options & Arguments

–destination=*file*

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

–preload=*module*

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

–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:
--destination=file      specifies output file (and directory).
--source=sourcedir      specifies directory of source TeX file.
--format=html|xhtml|xml requests the output format.
--stylesheet=xslfile    requests the XSL transform using the
                        given xslfile as stylesheet.
--css=cssfile           adds a css stylesheet to html/xhtml
                        (can be repeated)
--nodefaultcss          disables use of the default css stylesheet
--split                 requests splitting each document
--nosplit               disables the above (default)
--splitat               specifies what level to split the document
--splitpath=xpath       specifies xpath expression for splitting
                        (default is section-like, if splitting)
--splitnaming=(id|idrelative|label|labelrelative) specifies how
                        to name split files (def. idrelative).
--index                 requests filling in the index (default)
--noindex               disables the above
--permutedindex         permutes index phrases in the index
--nopermutedindex       disables the above (default)
--splitindex            Splits the index into pages per initial.
--nosplitindex          disables the above (default)
--bibliography=file     specifies a bibliography file
--splitbibliography     splits the bibliography into pages per
                        initial.
--nosplitbibliography   disables the above (default)
--scan                  scans documents to extract ids, labels,
                        section titles, etc. (default)
--noscan                disables the above
--crossref              fills in crossreferences (default)
--nocrossref            disables the above
--urlstyle=(server|negotiated|file) format to use for urls
                        (default server).
--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 image 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
--graphicsimages        converts graphics to images (default)
--nographicsimages      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)
--keepXMath             preserves the intermediate XMath
                        representation (default is to remove)
--verbose               shows progress during processing.
--VERSION               show version number.
--help                  shows help message.
```

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

Options & Arguments

General Options

–verbose

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

–VERSION

Shows the version number of the LaTeXML package..

–help

Shows this help message.

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, the output is left in LaTeXML's xml, but the math is parsed and converted to presentation MathML. For html and xhtml, a default stylesheet is provided, but see the **–stylesheet** option.

–source=*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.

–destination=*destination*

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

–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`, unless that is inhibited (see **–nodefaultcss**). Some stylesheets included in the distribution are –css=navbar-left Puts a navigation bar on the left (default omits the navbar) –css=navbar-right Puts a navigation bar on the left –css=theme-blue A blue coloring theme for headings –css=amsart A style suitable for journal articles

–nodefaultcss

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

Site & Crossreferencing Options

–split, –nosplit

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

–splitat=*unit*

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

–splitpath=*xpath*

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

Specifying –splitpath="//ltx:section |//ltx:subsection |//ltx:bibliography |//ltx:appendix |//ltx:index"

would split the document at subsections as well as sections.

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

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

–scan, –noscan

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

–crossref, –nocrossref

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

–urlstyle=`(server|negotiated|file)`

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

–index, –noindex

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

–splitindex, –nosplitindex

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

–bibliography=*pathname*

Specifies a bibliography file generated from a BibTeX file. This is used to fill in a bibliography element. Explicit bibliographies generated by 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) Applies line-breaking to the generated Presentation MathML such 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

–graphicsimages, –nographicsimages

Enables (default) or disables the conversion of graphics inclusion 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 `XML::LibXML::Document`.

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

Digests `$string`, which presumably contains TeX markup, and 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` reprentation) into XML, returning the `XML::LibXML::Document`.

Customization

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

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

See also

See `latexml` for a simple command line script.

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

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

`LaTeXML::State`

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

`LaTeXML::Token`, `LaTeXML::Mouth` **and** `LaTeXML::Gullet`

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

`LaTeXML::Box` **and** `LaTeXML::Stomach`

deal with digestion of tokens into boxes.

`LaTeXML::Document`, `LaTeXML::Model`, `LaTeXML::Rewrite`
: dealing with conversion of the digested boxes into XML.

`LaTeXML::Definition` **and** `LaTeXML::Parameters`
: representation of LaTeX macros, primitives, registers and constructors.

`LaTeXML::MathParser`
: the math parser.

`LaTeXML::Global`, `LaTeXML::Error`, `LaTeXML::Object`, `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 comparision, but the default implementation just compares for object identity.

`$boole = $object->isaToken;`

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

`$boole = $object->isaBox;`

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

`$boole = $object->isaDefinition;`

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

`$digested = $object->beDigested;`

> Does whatever is needed to digest the object, and return the digested representation. Tokens would be digested into boxes; Some objects, such as numbers can just return themselves.

`$object->beAbsorbed($document);`

> Do whatever is needed (typically by invoking appropriate methods on the `$document`) to absorb the `$object` into the `$document`.

LaTeXML::Definition

Control sequence definitions, including specializations `LaTeXML::Expandable`, `LaTeXML::Primitive`, `LaTeXML::Register`, `LaTeXML::Constructor`

Description

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

`LaTeXML::Expandable`

> represents macros and other expandable control sequences like `\if`, etc that are carried out in the Gullet during expansion. The results of invoking an `LaTeXML::Expandable` should result in 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 speciallized primitive with a getter and setter to access and store values in the Stomach.

`LaTeXML::Constructor`

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

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

Methods in general

`$token = $defn->getCS;`

> Returns the (main) token that is bound to this definition.

`$string = $defn->getCSName;`

> Returns the string form of the token bound to this definition, taking into account any alias for this definition.

`$defn->readArguments($gullet);`

> Reads the arguments for this `$defn` from the `$gullet`, returning a list of `LaTeXML::Tokens`.

`$parameters = $defn->getParameters;`

> Return the `LaTeXML::Parameters` object representing the formal parameters of the definition.

`@tokens = $defn->invocation(@args);`

> Return the tokens that would invoke the given definition with the provided arguments. This is used to recreate the TeX code (or it's equivalent).

`$defn->invoke;`

> Invoke the action of the `$defn`. For expandable definitions, this is done in the Gullet, and returns a list of `LaTeXML::Token`s. For primitives, it is carried out in the Stomach, and returns a list of `LaTeXML::Box`es. For a constructor, it is also carried out by the Stomach, and returns a `LaTeXML::Whatsit`. That whatsit will be responsible for constructing the XML document fragment, when the `LaTeXML::Document` invokes `$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 subroutines, called `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` or similar) 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` either a subroutine, or a string pattern representing the XML fragment it should generate. In the case of a string pattern, the pattern is compiled into a subroutine on first usage by the internal class `LaTeXML::ConstructorCompiler`. Like primitives, constructors may have `beforeDigest` and `afterDigest`.

LaTeXML::Global

Global exports used within LaTeXML, and in Packages.

Synopsis

use LaTeXML::Global;

Description

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

Global state

`$STATE;`

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

Tokens

`$catcode = CC_ESCAPE;`

Constants for the category codes:

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

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

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

Creates a `LaTeXML::Token` with the given content and catcode. The following shorthand versions are also exported for convenience:

```
T_BEGIN, T_END, T_MATH, T_ALIGN, T_PARAM,
T_SUB, T_SUPER, T_SPACE, T_LETTER($letter),
T_OTHER($char), T_ACTIVE($char),
T_COMMENT($comment), T_CS($cs)
```

`$tokens = Tokens(@token);`

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

`$tokens = Tokenize($string);`

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

`$tokens = TokenizeInternal($string);`

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

`@tokens = Explode($string);`

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

Numbers, etc.

`$number = Number($num);`

Creates a Number object representing `$num`.

`$number = Float($num);`

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

`$dimension = Dimension($dim);`

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

`$mudimension = MuDimension($dim);`

Creates a MuDimension object; similar to Dimension.

`$glue = Glue($gluespec);`

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

Creates a Glue object. `$gluespec` can be a string in the form that TeX recognizes (number units optional plus and minus parts). Alternatively, the dimension, plus and minus parts can be given separately: `$pfill` and `$mfill` are 0 (when the `$plus` or `$minus` part is in sp) or 1,2,3 for fil, fill or filll.

`$glue = MuGlue($gluespec);`

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

Creates a MuGlue object, similar to Glue.

`$pair = Pair($num1,$num2);`

Creates an object representing a pair of numbers; Not a part of TeX, but useful for graphical objects. The two components can be any numerical object.

`$pair = PairList(@pairs);`

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

Error Reporting

`Fatal($message);`

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

`Error($message);`

> Signals an error, printing `$message` along with some context. If in strict mode, this is the same as Fatal(). Otherwise, it attempts to continue processing..

`Warn($message);`

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

`NoteProgress($message);`

> Prints `$message` unless the verbosity level below 0.

Generic functions

`Stringify($object);`

> Returns a 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,$msg,$long,@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 `$long` is true, will generate a more verbose message; this also uses the VERBOSITY set in the `$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{}', "<ltx:thanks>#1</ltx:thanks>");

# And a simple environment ...
DefEnvironment('{abstract}','<abstract>#body</abstract>');

# A math  symbol \Real to stand for the Reals:
DefMath('\Real', "\x{211D}", role=>'ID');

 # Or a semantic floor:
DefMath('\floor{}','\left\lfloor#1\right\rfloor');

# More esoteric ...

# Use a special DocType, if not LaTeXML.dtd
DocType("rootelement","-//Your Site//Your DocType",'your.dtd',
        prefix=>"http://whatever/");

# Allow sometag elements to be automatically closed if needed
Tag('pre: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`, (so that `\usepackage{mypackage}` works), you create the file `mypackage.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). Likewise, to provide document-specific customization for, say, `mydoc.tex`, you would create the file `mydoc.latexml` (typically in the same directory). 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 Sequence Definitions

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

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

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

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

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

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

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

The shorthands {} and [] default the type to `Plain` and reads a normal TeX argument or LaTeX default argument.

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.

The defining forms

`DefExpandable($prototype,CODE($gullet,@args),%options);`

Defines an expandable control sequence. The CODE should return a list of `LaTeXML::Token`'s that replace the macro and its arguments. 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).

`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)`.

`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.

`DefPrimitive($prototype,CODE($stomach,@args),%options);`

Define a primitive control sequence. The CODE should return a list of digested items, but usually should return nothing (eg. end with return;).

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.

`DefConstructor($prototype,$xmlpattern |$code,%options);`

The Constructor is where LaTeXML really starts getting interesting; invoking the control sequence will generate an arbitrary XML fragment in the document tree. More specifically: during digestion, the arguments will be read and digested, creating a `LaTeXML::Whatsit` to represent the object. During absorbtion by the `LaTeXML::Document`, the `Whatsit` will generate the XML fragment according to the replacement `$xmlpattern`, or by executing `CODE`.

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

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

#1, #2 ... #name

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

`&function(@args)`

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

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

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

`^`

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

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

DefConstructor options are

mode=>(text|display_math| inline_math)

Changes to this mode during digestion.

bounded=>boolean

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

requireMath=>boolean

forbidMath=>boolean

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

font=>{fontspec...}

Specifies the font to be set by this invocation. See `/MergeFont` If the font change is to only apply to this construct, you would also use <`bounded=1`>>.

reversion=>$texstring or CODE($whatsit,#1,#2,...)

Specifies the reversion of the invocation back into TeX tokens (if the default reversion is not appropriate). The $textstring string can include #1,#2... The CODE is called with the $whatsit and digested arguments.

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

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

beforeDigest=>CODE($stomach)
: This option supplies a Daemon to be executed during digestion just before the Whatsit is created. The CODE should either return nothing (return;) or a list of digested items (Box's,List,Whatsit). It can thus change the State and/or add to the digested output.

afterDigest=>CODE($stomach,$whatsit)
: This option supplies a Daemon to be executed during digestion just after the Whatsit is created. it should either return nothing (return;) or digested items. It can thus change the State, modify the Whatsit, and/or add to the digested output.

beforeConstruct=>CODE($document,$whatsit)
: Supplies CODE to execute before constructing the XML (generated by $replacement).

afterConstruct=>CODE($document,$whatsit)
: Supplies CODE to execute after constructing the XML.

captureBody=>boolean
: 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=>$nargs
: This gives a number of args for cases where it can't be infered directly from the `$prototype` (eg. when more args are explictly 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 <`noggroup=`1>>inhibits this.

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

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

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

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

It shares options with `DefConstructor`:

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

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

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

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

`Let($token1,$token2);`

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

Document Declarations

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.

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.

`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.

`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).

Ligatures

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

`DefLigature($regexp,%options);`

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

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

`DefMathLigature(CODE($document,@nodes));`

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

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

Document Rewriting

`DefRewrite(%specification);`

`DefMathRewrite(%specification);`

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

The following select portions of the document:

label =>$label
: Selects the part of the document with label=$label

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

xpath =>$xpath
: Select those nodes matching an explicit xpath expression.

match =>$TeX
: Selects nodes that look like what the processing of $TeX would produce.

regexp=>$regexp
: Selects text nodes that match the regular expression.

The following act upon the selected node:

attributes =>$hash
: Adds the attributes given in the hash reference to the node.

replace =>$replacement
: Interprets the $replacement as TeX code to generate nodes that will replace the selected nodes.

Other useful operations

`RequirePackage($package);`

Finds an implementation (`*.sty` or `*.ltxml`) for the required `$package`, loading it as appropriate.

`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.

Convenience Functions

The following are exported as a convenience when writing definitions.

`$value = LookupValue($name);`

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

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

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

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

```
VERBOSITY         : the level of verbosity for debugging
                    output, with 0 being default.
STRICT            : whether errors (eg. undefined macros)
                    are fatal.
INCLUDE_COMMENTS  : whether to preserve comments in the
                    source, and to add occasional line
                    number comments. (Default true).
PRESERVE_NEWLINES : whether newlines in the source should
                    be preserved (not 100% TeX-like).
                    By default this is true.
SEARCHPATHS       : a list of directories to search for
                    sources, implementations, etc.
```

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

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

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

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

`$value = LookupCatcode($char);`

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

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

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

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

`$meaning = LookupMeaning($token);`

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

`$defn = LookupDefinition($token);`

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

`InstallDefinition($defn);`

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

`$boxes = Digest($tokens);`

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

`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.

`$tokens = Expand($tokens);`

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

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

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

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

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

`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.

LaTeXML::Parameters

Formal parameters, including `LaTeXML::Parameter`.

Description

Provides a representation for the formal parameters of `LaTeXML::Definition`s: `LaTeXML::Parameter` for an individual parameter, `LaTeXML::Parameters` for the complete parameter list.

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 `Token`s 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 this `$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

Additionally, `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::Token`s according to the current category codes (catcodes) stored in the `LaTeXML::State`.

`LaTeXML::FileMouth` specializes `LaTeXML::Mouth` to tokenize from a file. `LaTeXML::StyleMouth` further specializes `LaTeXML::FileMouth` for processing style files, setting the catcode for `@` and ignoring comments.

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

Creating Mouths

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

Creates a new Mouth reading from `$string`.

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

Creates a new FileMouth to read from the given file.

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

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

Methods

`$token = $mouth->readToken;`

Returns the next `LaTeXML::Token` from the source.

`$boole = $mouth->hasMoreInput;`

Returns whether there is more data to read.

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

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

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

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

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

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

`$endline` is returned as the last line, and any characters after `$endline` remains in the mouth to be tokenized.

`LaTeXML::Gullet`

Expands expandable tokens and parses common token sequences.

Description

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

Managing Input

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

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

`$gullet->openMouth($mouth, $noautoclose);`

Is this public? Prepares to read tokens from `$mouth`. If $noautoclose is true, the Mouth will not be automatically closed when it is exhausted.

`$gullet->closeMouth;`

Is this public? Finishes reading from the current mouth, and reverts to the one in effect before the last openMouth.

`$gullet->flush;`

Is this public? Clears all inputs.

`$gullet->getLocator;`

Returns a string describing the current location in the input stream.

Low-level methods

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

Return the `LaTeXML::Tokens` resulting from expanding all the tokens in `$tokens`. This is actually only used in a few circumstances where the arguments to an expandable need explicit expansion; usually expansion happens at the right time.

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

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

`$token = $gullet->readToken;`

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

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

Return the next unexpandable token from the input source, or undef if there is no more input. If the next token is expandable, it is expanded, and its expansion is reinserted into the input.

`$gullet->unread(@tokens);`

Push the `@tokens` back into the input stream to be re-read.

Mid-level methods

`$token = $gullet->readNonSpace;`

Read and return the next non-space token from the input after discarding any spaces.

`$gullet->skipSpaces;`

Skip the next spaces from the input.

`$gullet->skip1Space;`

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

`$tokens = $gullet->readBalanced;`

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

`$boole = $gullet->ifNext($token);`

Returns true if the next token in the input matches `$token`; the possibly matching token remains in the input.

`$tokens = $gullet->readMatch(@choices);`

Read and return whichever of `@choices` (each are `LaTeXML::Tokens`) matches the input, or undef if none do.

`$keyword = $gullet->readKeyword(@keywords);`

Read and return whichever of `@keywords` (each a string) matches the input, or undef if none do. This is similar to readMatch, but case and catcodes are ignored. Also, leading spaces are skipped.

`$tokens = $gullet->readUntil(@delims);`

Read and return a (balanced) sequence of `LaTeXML::Tokens` until matching one of the tokens in `@delims`. In a list context, it also returns which of the delimiters ended the sequence.

High-level methods

`$tokens = $gullet->readArg;`

Read and return a TeX argument; the next Token or Tokens (if surrounded by braces).

`$tokens = $gullet->readOptional($default);`

Read and return a LaTeX optional argument; returns `$default` if there is no '[', otherwise the contents of the [].

`$thing = $gullet->readValue($type);`

Reads an argument of a given type: one of 'Number', 'Dimension', 'Glue', 'MuGlue' or 'any'.

`$value = $gullet->readRegisterValue($type);`

Read a control sequence token (and possibly it's arguments) that names a register, and return the value. Returns undef if the next token isn't such a register.

`$number = $gullet->readNumber;`

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

`$dimension = $gullet->readDimension;`

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

`$mudimension = $gullet->readMuDimension;`

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

`$glue = $gullet->readGlue;`

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

`$muglue = $gullet->readMuGlue;`

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

`LaTeXML::Stomach`

Digests tokens into boxes, lists, etc.

Description

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

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

A plain character

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

A primitive

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

Grouping (or environment bodies)

are collected into a `LaTeXML::List`.

Constructors

A special class of control sequence, called a `LaTeXML::Constructor` produces a `LaTeXML::Whatsit` which remembers the control sequence and arguments that created it, and defines its own translation into `XML` elements, attributes and data. Arguments to a constructor are read from the gullet and also digested.

Digestion

`$list = $stomach->digestNextBody;`

Return the digested `LaTeXML::List` after reading and digesting a 'body' from the its Gullet. The body extends until the current level of boxing or environment is closed.

`$list = $stomach->digest($tokens);`

Return the `LaTeXML::List` resuting from digesting the given tokens. This is typically used to digest arguments to primitives or constructors.

`@boxes = $stomach->invokeToken($token);`

Invoke the given (expanded) token. If it corresponds to a Primitive or Constructor, the definition will be invoked, reading any needed arguments fromt he current input source. Otherwise, the token will be digested. A List of Box's, Lists, Whatsit's is returned.

`@boxes = $stomach->regurgitate;`

> Removes and returns a list of the boxes already digested at the current level. This peculiar beast is used by things like \choose (which is a Primitive in TeX, but a Constructor in LaTeXML).

Grouping

`$stomach->bgroup;`

> Begin a new level of binding by pushing a new stack frame, and a new level of boxing the digested output.

`$stomach->egroup;`

> End a level of binding by popping the last stack frame, undoing whatever bindings appeared there, and also decrementing the level of boxing.

`$stomach->begingroup;`

> Begin a new level of binding by pushing a new stack frame.

`$stomach->endgroup;`

> End a level of binding by popping the last stack frame, undoing whatever bindings appeared there.

Modes

`$stomach->beginMode($mode);`

> Begin processing in `$mode`; one of 'text', 'display-math' or 'inline-math'. This also begins a new level of grouping and switches to a font appropriate for the mode.

`$stomach->endMode($mode);`

> End processing in `$mode`; an error is signalled if `$stomach` is not currently in `$mode`. This also ends a level of grouping.

`LaTeXML::Document`

Represents an XML document under construction.

Description

A `LaTeXML::Document` constructs an XML document by absorbing the digested `LaTeXML::List` (created by `LaTeXML::Stomach`), Generally, the `LaTeXML::Box`s and `LaTeXML::List`s create text nodes, whereas the `LaTeXML::Whatsit`s create `XML` document fragments, elements and attributes according to the defining `LaTeXML::Constructor`.

The `LaTeXML::Document` maintains a current insertion point for where material will be added. The `LaTeXML::Model`, derived from various declarations and document type, is consulted to determine whether an insertion is allowed and when elements may need to be automatically opened or closed in order to carry out a given insertion. For example, a `subsection` element will typically be closed automatically when it is attempted to open a `section` element.

In the methods described here, the term `$qname` is used for XML qualified names. These are tag names with a namespace prefix. The prefix should be one registered with the current Model, for use within the code. This prefix is not necessarily the same as the one used in any DTD, but should be mapped to the a Namespace URI that was registered for the DTD.

The arguments named `$node` are an XML::LibXML node.

Accessors

`$doc = $document->getDocument;`

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

`$node = $document->getNode;`

Returns the node at the current insertion point during construction. This node is considered still to be ‘open’; any insertions will go into it (if possible). The node will be an `XML::LibXML::Element`, `XML::LibXML::Text` or, initially, `XML::LibXML::Document`.

`$node = $document->getElement;`

Returns the closest ancestor to the current insertion point that is an Element.

`$document->setNode($node);`

Sets the current insertion point to be `$node`. This should be rarely used, if at all; The construction methods of document generally maintain the notion of insertion point automatically. This may be useful to allow insertion into a different part of the document, but you probably want to set the insertion point back to the previous node, afterwards.

Construction Methods

`$document->absorb($digested);`

> Absorb the `$digested` object into the document at the current insertion point according to its type. Various of the the other methods are invoked as needed, and document nodes may be automatically opened or closed according to the document model.

`$xmldoc = $document->finalize;`

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

`$document->openText($text,$font);`

> Open a text node in font `$font`, performing any required automatic opening and closing of intermedate nodes (including those needed for font changes) and inserting the string `$text` into it.

`$document->insertMathToken($string,%attributes);`

> Insert a math token (XMTok) containing the string `$string` with the given attributes. Useful attributes would be name, role, font. Returns the newly inserted node.

`$document->openElement($qname,%attributes);`

> Open an element, named `$qname` and with the given attributes. This will be inserted into the current node while performing any required automatic opening and closing of intermedate nodes. The new element is returned, and also becomes the current insertion point. An error (fatal if in `Strict` mode) is signalled if there is no allowed way to insert such an element into the current node.

`$document->closeElement($qname);`

> Close the closest open element named `$qname` including any intermedate nodes that may be automatically closed. If that is not possible, signal an error. The closed node's parent becomes the current node. This method returns the closed node.

`$node = $document->isOpenable($qname);`

> Check whether it is possible to open a `$qname` element at the current insertion point.

`$node = $document->isCloseable($qname);`

> Check whether it is possible to close a `$qname` element, returning the node that would be closed if possible, otherwise undef.

`$document->maybeCloseElement($qname);`

> Close a `$qname` element, if it is possible to do so, returns the closed node if it was found, else undef.

`$document->insertElement($qname,$content,%attributes);`

This is a shorthand for creating an element `$qname` (with given attributes), absorbing `$content` from within that new node, and then closing it. The `$content` must be digested material, either a single box, or an array of boxes. This method returns the newly created node, although it will no longer be the current insertion point.

`$document->insertComment($text);`

Insert, and return, a comment with the given `$text` into the current node.

`$document->insertPI($op,%attributes);`

Insert, and return, a ProcessingInstruction into the current node.

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

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

`LaTeXML::Model`

Represents the Document Model

Description

`LaTeXML::Model` encapsulates information about the document model to be used in converting a digested document into XML by the `LaTeXML::Document`. This information is based on the DTD, but may also be modified by modules implementing various macro packages; thus the model may not be complete until digestion is completed.

The kinds of information that is relevant is not only the content model (what each element can contain contain), but also SGML-like information such as whether an element can be implicitly opened or closed, if needed to insert a new element into the document.

Currently, only a DTD is understood (no schema yet), and even there, the stored model is only approximate. 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

`$name = $model->getRootName;`

Return the name of the expected root element.

`$publicid = $model->getPublicID;`

Return the public identifier for the document type.

`$systemid = $model->getSystemID;`

Return the system identifier for the document type (typically a filename for the DTD).

`$model->setDocType($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

`$model->registerNamespace($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.

Possibile Customizations

Needs description.

Convenience functions

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

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

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

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

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

`Annotate($node,%attributes);`

> Add attributes to `$node`.

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

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

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

> Create a new `XMApp` node representing the application of the node `$op` to the arguments found in `@stuff`. `@stuff` are delimited arguments in the sense that the leading and trailing nodes should represent open and close delimiters and the arguments are seperated by punctuation nodes. The text of these delimiters and punctuation are used to annotate the operator node with `argopen`, `argclose` and `separator` attributes.

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

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

`$node = InvisibleTimes;`

Creates an invisible times operator.

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

Checks whether `$open` and `$close` form a ‘normal’ pair of delimiters, or if either is ”.”.

`$node=>Fence(@stuff);`

Given a delimited sequence of nodes, starting and ending with open/close delimiters, and with intermediate nodes separated by punctuation or such, attempt to guess what type of thing is represented such as a set, absolute value, interval, and so on. If nothing specific is recognized, creates the application of `FENCED` to the arguments.

This would be a good candidate for customization!

`$node = NewFormulae(@stuff);`

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

`$node = 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

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

LaTeXml DocType

The document type used by LaTeXML is modular in the sense that it is composed of several modules that define different sets of elements related to, eg., inline content, block content, math and high-level document structure. This allows the possibility of mixing models or extension by predefining certain parameter entities. However, in order to present a more readable summary of the model, most lower level parameter entities have been expanded in the following. Customizers are recommended to study the actual dtd modules for detailed guidelines.

Module `core`

This module defines the parameter entities and core attribute sets used by most other modules.

`LaTeXML.Common.attrib` ≡ `xmlns`

Attributes shared by ALL elements.

`xmlns` provides for namespace declaration.

`class` can be used to add differentiate different instances of elements without introducing new element declarations; it generally shouldn't be used for deep semantic distinctions, however. This attribute is carried over to HTML and can be used for CSS selection.

`LaTeXML.ID.attrib` ≡ `id` [ID]

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

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

`LaTeXML.IDREF.attrib` ≡ `idref` [IDREF]

Attributes for elements that can cross-reference other elements.

`idref` the identifier of the referred-to element.

`LaTeXML.Labelled.attrib` ≡ `id` [ID], `label`, `refnum`

Attributes for elements that can be labelled from within LaTeX.

`label` the LaTeX label of the element, supplied by the acro.

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

`LaTeXML.Positionable.attrib` ≡ `width`, `height`, `depth`, `pad-width`, `pad-height`, `xoffset`, `yoffset`, `align` (left | center | right | justified), `vattach` (top | middle | bottom)

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

`width`, `height`, `depth` the size of the box.

`pad-width`, `pad-height` extra size beyond its natural size.

`xoffset`, `yoffset` shifts the position of the box.

`align` alignment of material within the box.

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

`LaTeXML.Imageable.attrib` ≡ `imagesrc`, `imagewidth`, `imageheight`

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

`imagesrc` the file, possibly generated from other data.

`imagewidth` the width in pixels of `imagesrc`.

`imageheight` the height in pixels of `imagesrc`.

Module `classes`

This module combines the contributions from the various included modules and assembles entities representing the several basic classes of content, such as inline, block and so on.
Basic element classes:

`LaTeXML.Inline.class` ≡ `text` | `emph` | `rule` | `Math` | `anchor` | `ref` | `cite` | `bibref` | `acronym`

All strictly inline elements.

`LaTeXML.Block.class` ≡ `p` | `equation` | `equationgroup` | `quote` | `centering` | `block` | `acronyms` | `itemize` | `enumerate` | `description`

All 'physical' block elements. A physical block is typically displayed as a block, but may not constitute a complete logical unit.

`LaTeXML.Misc.class` ≡ | `inline-block` | `verbatim` | `tabular` | `graphics` | `picture`

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

`LaTeXML.Para.class` ≡ `para` | `figure` | `table` | `theorem` | `proof`

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.

`LaTeXML.Meta.class` ≡ | `note` | `ERROR` | `indexmark`

All metadata elements, typically representing hidden data.

Core mixes of element classes:

`LaTeXML.Inline.mix` ≡ `%LaTeXML.Inline.class;` `%LaTeXML.Misc.class;` `%LaTeXML.Meta.class;`

Mix of all elements that can appear in an inline context.

`LaTeXML.Block.mix` ≡ `%LaTeXML.Block.class;` `%LaTeXML.Misc.class;` `%LaTeXML.Meta.class;`

Mix of all elements that can appear in a physical block-level context.

`LaTeXML.Flow.mix` ≡ `%LaTeXML.Inline.class;` | `%LaTeXML.Block.class;` `%LaTeXML.Misc.class;` `%LaTeXML.Meta.class;`

Mix of all 'flow'-level elements (ie. both inline and physical block).

`LaTeXML.Para.mix` ≡ `%LaTeXML.Para.class;` `%LaTeXML.Meta.class;`

Mix of all elements that can appearin a logical block-level context.

Models based on element classes:

`LaTeXML.Inline.model` ≡ (#PCDATA | `%LaTeXML.Inline.mix;`)*

Combined model for inline content.

`LaTeXML.Flow.model` ≡ (#PCDATA | `%LaTeXML.Flow.mix;`)*

Combined model for flow content.

Classes and Attributes for Equations, Math, Figures, Tables and such.

`LaTeXML.Math.class` ≡ `XMath`

The content of the Math element including the internal representation of math (`XMath`) and any additional representations.

`LaTeXML.XMath.class` ≡ `XMApp` | `XMTok` | `XMRef` | `XMHint` | `XMArg` | `XMWrap` | `XMDual` | `XMText` | `XMArray`

The model for `XMath` (the internal representation of math).

`LaTeXML.XMath.attrib` ≡ `role`, `open`, `close`, `punctuation`, `argopen`, `argclose`, `separators`, `possibleFunction`

Combined attributes for `XMath` elements. `XMath` attributes include

`role` The role that this item plays in the Grammar.

`open, close` fences around the object;

`argopen, argclose, punctuation` fences and punctuation around and within the arguments when this object is applied to arguments;

`possibleFunction` the parser suspects this may be used as a function.

`LaTeXML.Caption.class` ≡ `caption` | `toccaption`

Additional caption-like content allowed in `table` and `figure`.

`LaTeXML.Picture.class` ≡ `g` | `rect` | `line` | `circle` | `path` | `arc` | `wedge` | `ellipse` | `polygon` | `bezier` | `%LaTeXML.Inline.mix;`

Content of a `picture` element.

`LaTeXML.Picture.attrib` ≡ `x`, `y`, `r`, `rx`, `ry`, `width`, `height`, `fill`, `stroke`, `stroke-width`, `stroke-dasharray`, `transform`, `terminators`, `arrowlength`, `points`, `showpoints`, `displayedpoints`, `arc`, `angle1`, `angle2`, `arcsepA`, `arcsepB`, `curvature`

Combined attributes of a `picture` element. These attributes correspond roughly to SVG, but need documentation.

`LaTeXML.PictureGroup.attrib` ≡ `pos`, `framed` (yes | no), `frametype` (rect | circle | oval), `fillframe` (yes | no), `boxsep`, `shadowbox` (yes | no), `doubleline` (yes | no)

Combined attributes for PictureGroup (`g`) element. These attributes correspond roughly to SVG, but need documentation.

`LaTeXML.Person.class` ≡ `personname` | `contact %LaTeXML.Misc.class;`

Content for elements representing a person (but conflicts with bibliographic!).

Document structure classes:

`LaTeXML.SectionalFrontMatter.class` ≡ `title` | `toctitle` | `creator`

Model for the FrontMatter of sections. This precedes the normal content of the section, such as logical block level content.

`LaTeXML.FrontMatter.class` ≡ `title` | `toctitle` | `creator` | `subtitle` | `date` | `abstract` | `acknowledgements` | `keywords` | `classification`

Model for the FrontMatter of documents. This precedes the normal content of the section, such as logical block level content.

`LaTeXML.BackMatter.class` ≡ `bibliography` | `appendix` | `index`

Model for the BackMatter of documents. This follows the normal content of the section, such as logical block level content.

`LaTeXML.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-issn` | `bib-doi` | `bib-isbn` | `bib-review` | `bib-mrnumber` | `bib-mrreviewer` | `bib-language` | `bib-url` | `bib-eprint` | `bib-preprint` | `bib-note`

The content model of a bibliographic entry (`bibentry`). These elements have a direct correspondence to BibTeX fields.

`LaTeXML.Bibname.model` ≡ (`surname`, (`givenname`)?, (`initials`)?, (`lineage`)?)

The content model of the bibliographic name fields (`bib-author`, `bib-editor`, `bib-translator`)

Module `text`

`text` : `%LaTeXML.Inline.model;`

General container for styled text.

attributes: `%LaTeXML.Positionable.attrib;` , `font`, `size`, `color`, `framed` (square | rectangle | circle | underline)

Attributes cover a variety of styling and position shifting properties.

`emph` : `%LaTeXML.Inline.model;`

Emphasized text.

`rule` :*empty*

A Rule.

attributes: `%LaTeXML.Positionable.attrib;`

`note` : `%LaTeXML.Flow.model;`

Metadata that covers several 'out of band' annotations.

attributes: `mark`

`mark` indicates the desired visible marker to be linked to the note.

`ERROR` :(#PCDATA)*

error object for undefined control sequences, or whatever

Module `block`

`p` :(#PCDATA | `%LaTeXML.Inline.mix;` | `break`)*

A physical paragraph.

`centering` :(`caption` | `toccaption` | `%LaTeXML.Block.mix;`)*

A physical block that centers its content.

`equation` :(#PCDATA | `%LaTeXML.Inline.mix;`)*

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.

attributes: `%LaTeXML.Labelled.attrib;`

`equationgroup` :(`%LaTeXML.Block.mix;`)*

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

attributes: `%LaTeXML.Labelled.attrib;`

`quote` :(#PCDATA | `%LaTeXML.Inline.mix;` | `break`)*

A quotation

`block` :(#PCDATA | `%LaTeXML.Inline.mix;` | `break`)*

A generic block (fallback).

attributes: `%LaTeXML.Positionable.attrib;`

`break` :*empty*

A forced line break.

`inline-block` : `%LaTeXML.Inline.model;`

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

attributes: `%LaTeXML.Positionable.attrib;`

`verbatim` :(#PCDATA | `%LaTeXML.Inline.mix;` | `break`)*

Verbatim content

attributes: `font`

`para` :(`%LaTeXML.Block.mix;`)*

A Logical paragraph. It has an `ID`, but not a `label`.

attributes: `%LaTeXML.ID.attrib;`

Module `math`

`Math` :(`%LaTeXML.Math.class;`)*

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

attributes: `%LaTeXML.Imageable.attrib;` , `mode` (display | inline), `tex`, `content-tex`, `text`

`mode` display or inline mode.

`tex` reconstruction of TeX that generated the math.

`content-tex` more semantic version of above.

`text` a textified representation of the math.

`XMath` :(`%LaTeXML.XMath.class;`)*

Internal representation of mathematics.

attributes: `status`

`XMTok` :(#PCDATA)*

General mathematical token.

attributes: `%LaTeXML.XMath.attrib;` , `%LaTeXML.ID.attrib;` , `name`, `meaning`, `omcd`, `style`, `font`, `size`, `color`, `scriptpos`, `thickness`

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

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

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

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

`font` The font, size a used for the symbol.

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

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

`thickness` ?

`XMApp` :(`%LaTeXML.XMath.class;`)*

Generalized application of a function, operator, whatever (the first child) to arguments (the remaining children).

attributes: `%LaTeXML.XMath.attrib;` , `%LaTeXML.ID.attrib;` , `name`, `meaning`, `scriptpos`

The attributes are a subset of those for `XMTok`.

`XMDual` :((`%LaTeXML.XMath.class;`), (`%LaTeXML.XMath.class;`))

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: `%LaTeXML.XMath.attrib;` , `%LaTeXML.ID.attrib;`

`XMHint` :*empty*

Various spacing items, generally ignored in parsing.

attributes: `%LaTeXML.XMath.attrib;` , `%LaTeXML.ID.attrib;` , `name`, `meaning`, `style`

The attributes are a subset of those for `XMTok`.

`XMText` :(#PCDATA | `%LaTeXML.Inline.class;` `%LaTeXML.Misc.class;`)*

Text appearing within math.

attributes: `%LaTeXML.XMath.attrib;` , `%LaTeXML.ID.attrib;`

`XMWrap` :(`%LaTeXML.XMath.class;`)*

Wrapper for a sequence of tokens used to assert the role of the contents in its parent. This element generally disappears after parsing.

attributes: `%LaTeXML.XMath.attrib;` , `%LaTeXML.ID.attrib;` , `name`, `meaning`, `style`

The attributes are a subset of those for `XMTok`.

`XMArg` :(`%LaTeXML.XMath.class;`)*

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: `%LaTeXML.XMath.attrib;` , `%LaTeXML.ID.attrib;` , `rule`

`XMRef` :*empty*

Structure sharing element typically used in the presentation branch of an `XMDual` to refer to the arguments present in the content branch.

attributes: `%LaTeXML.XMath.attrib;` , `%LaTeXML.ID.attrib;` , `%LaTeXML.IDREF.attrib;`

`XMArray` :(`XMRow`)*

Math Array/Alignment structure.

attributes: `%LaTeXML.XMath.attrib;` , `%LaTeXML.ID.attrib;` , `name`, `meaning`, `style`, `vattach` (top | bottom), `width`

The attributes are a subset of those for `XMTok` or of `tabular`.

`XMRow` :(`XMCell`)*

A row in a math alignment.

`XMCell` :(`%LaTeXML.XMath.class;`)*

A cell in a row of a math alignment.

attributes: `rowpan`, `colspan`, `align`, `width`, `border`, `thead` (yes | no)

The attributes are the same as those for the `td` element.

Module `xref`

`ref` : `%LaTeXML.Inline.model;`

A hyperlink reference to some other object. When converted to HTML, the content would be the content of the anchor.

attributes: `%LaTeXML.IDREF.attrib;` , `labelref`, `show`, `href`, `title`

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).

`labelref` for a LaTeX labelled object,

`idref` for an internal identifier, or

`href` for an arbitrary url.

`title` attribute gives a longer form description of the target, this would typically appear as a tooltip in HTML.

`anchor` : `%LaTeXML.Inline.model;`

Inline anchor.

attributes: `%LaTeXML.ID.attrib;`

`cite` : `%LaTeXML.Inline.model;`

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.

`bibref` : `%LaTeXML.Inline.model;`

A bibliographic citation refering to a specific bibliographic item.

attributes: `%LaTeXML.IDREF.attrib;` , `bibrefs`, `show`

`bibrefs` a comma separated list of bibligraphic keys.

`show` encodes which of author(s), year, title, etc will be displayed. NOTE: Describe this.

Module `index`

`indexmark` :(`indexphrase`)*

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

attributes: `see_also`, `style`

`see` _also would be flattened form (`key`) of another `indexmark`, used to crossreference.

`indexphrase` : `%LaTeXML.Inline.model;`

A phrase within an `indexmark`

attributes: `key`

`key` is a flattened form of the phrase.

`indexlist` :(`indexentry`)*

An index generated from the collection of `indexmark` in a document (or document collection).

attributes: `%LaTeXML.ID.attrib;`

`indexentry` :((`indexphrase`), (`indexrefs`)?, (`indexlist`)?)

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: `%LaTeXML.ID.attrib;`

`indexrefs` : `%LaTeXML.Inline.model;`

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.

Module `tabular`

`tabular` :(((`col`)* | (`colgroup`)*), (`thead` | `tfoot` | `tbody` | `tr`)*)

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

attributes: `vattach` (top | middle | bottom), `width`

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

`width` the desired width of the tabular.

`colgroup` :(`col`)*

A container for descriptions of columns within the table.

attributes: `span`, `align`

`span` the number of columns spanned by this column

`align` the default alignment of column content.

`col` :*empty*

A description of a column, but not the column data itself.

attributes: `span`, `align`

`span` the number of columns spanned by this column `align` the default alignment of column content.

`thead` :(`tr`)*

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

`tfoot` :(`tr`)*

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

`tbody` :(`tr`)*

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

`tr` :(`td`)*

A row of a tabular.

`td` : `%LaTeXML.Flow.model;`

A cell in a row of a tabular.

attributes: `colspan`, `rowspan`, `align`, `width`, `border`, `thead` (yes | no)

`colspan, rowspan` indicate how many columns or rows this cell spans or covers.

`align` should be left, right, center or justify.

`width` specifies the desired width for the column.

`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.

`thead` is yes if the cell corresponds to a table head or foot.

Module `graphics`

`graphics` :*empty*

A graphical insertion of an external file.

attributes: `%LaTeXML.Imageable.attrib;` , `graphic`, `options`

`graphics` the path to the graphics file

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

Module `picture`

`picture` :(`%LaTeXML.Picture.class;`)*

A picture environment.

attributes: `%LaTeXML.Picture.attrib;` , `%LaTeXML.Imageable.attrib;` , `clip` (yes | no), `baseline`, `unitlength`, `xunitlength`, `yunitlength`, `tex`, `content-tex` `g` :(`%LaTeXML.Picture.class;`)*

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

attributes: `%LaTeXML.Picture.attrib;` , `%LaTeXML.PictureGroup.attrib;`

`rect` :*empty*

A rectangle within a `picture`.

attributes: `%LaTeXML.Picture.attrib;`

`line` :*empty*

A line within a `picture`.

attributes: `%LaTeXML.Picture.attrib;`

`polygon` :*empty*

A polygon within a `picture`.

attributes: `%LaTeXML.Picture.attrib;`

`wedge` :*empty*

A wedge within a `picture`.

attributes: `%LaTeXML.Picture.attrib;`

`arc` :*empty*

An arc within a `picture`.

attributes: `%LaTeXML.Picture.attrib;`

`circle` :*empty*

A circle within a `picture`.

attributes: `%LaTeXML.Picture.attrib;`

`ellipse` :*empty*

An ellipse within a `picture`.

attributes: `%LaTeXML.Picture.attrib;`

`path` :*empty*

A path within a `picture`.

attributes: `%LaTeXML.Picture.attrib;`

`bezier` :*empty*

A bezier curve within a `picture`.

attributes: `%LaTeXML.Picture.attrib;`

Module `float`

`figure` :(`%LaTeXML.Block.mix;` | `%LaTeXML.Caption.class;`)*

A figure, possibly captioned.

attributes: `%LaTeXML.Labelled.attrib;` , `placement`

`table` :(`%LaTeXML.Block.mix;` | `%LaTeXML.Caption.class;`)*

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

attributes: `%LaTeXML.Labelled.attrib;` , `placement`

`caption` : `%LaTeXML.Inline.model;`

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

`toccaption` : `%LaTeXML.Inline.model;`

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

Module `theorem`

`theorem` :((`title`)?, (`%LaTeXML.Block.mix;`)*)

A theorem or similar object. Attribute `class` can be used to distinguish.

attributes: `%LaTeXML.Labelled.attrib;`

`proof` :((`title`)?, (`%LaTeXML.Block.mix;`)*)

A proof or similar object. Attribute `class` can be used to distinguish.

Module `acro`

`acronym` : `%LaTeXML.Inline.model;`

Represents an acronym.

attributes: `name`

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

`acronyms` :(`item`)*

An acronyms list similar to a description. The `tag` within an `item` would typically be the acronym, with the text of the `item` providing a description of it.

Module `list`

`itemize` :(`item`)*

An itemized list.

attributes: `%LaTeXML.ID.attrib;`

`enumerate` :(`item`)*

An enumerated list.

attributes: `%LaTeXML.ID.attrib;`

`description` :(`item`)*

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

attributes: `%LaTeXML.ID.attrib;`

`item` :(`%LaTeXML.Block.mix;` | `tag`)*

An item within a list.

attributes: `%LaTeXML.Labelled.attrib;`

`tag` : `%LaTeXML.Inline.model;`

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

attributes: `open`, `close`

`open`, `close` opening and closing delimiters used to display the tag.

Module `bib`

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

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

`bibentry` :(`%LaTeXML.Bibentry.class;`)*

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

attributes: `%LaTeXML.ID.attrib;` , `key`, `type`

`bib-author` : `%LaTeXML.Bibname.model;`

Author of a bibliographic entry.

`bib-editor` : `%LaTeXML.Bibname.model;`

Editor of a bibliographic entry.

`bib-translator` : `%LaTeXML.Bibname.model;`

Translator of a bibliographic entry.

`surname` : `%LaTeXML.Inline.model;`

Surname of an author, editor or translator.

`givenname` : `%LaTeXML.Inline.model;`

Given name of an author, editor or translator.

`initials` : `%LaTeXML.Inline.model;`

Initials of an author, editor or translator.

`lineage` : `%LaTeXML.Inline.model;`

Lineage of an author, editor or translator. (eg. von)

`bib-title` : `%LaTeXML.Inline.model;`

Title of a bibliographic entry.

`bib-subtitle` : `%LaTeXML.Inline.model;`

Subtitle of a bibliographic entry.

`bib-booktitle` : `%LaTeXML.Inline.model;`

Title of the book containing a bibliographic entry.

`bib-key` : `%LaTeXML.Inline.model;`

Unique key of a bibliographic entry.

`bib-journal` : `%LaTeXML.Inline.model;`

Journal of a bibliographic entry.

`bib-series` : `%LaTeXML.Inline.model;`

Series of a bibliographic entry.

`bib-conference` : `%LaTeXML.Inline.model;`

Conference of a bibliographic entry.

`bib-publisher` : `%LaTeXML.Inline.model;`

Publisher of a bibliographic entry.

`bib-organization` : `%LaTeXML.Inline.model;`

Organization responsible for a bibliographic entry.

`bib-institution` : `%LaTeXML.Inline.model;`

Institution responsible for a bibliographic entry.

`bib-address` : `%LaTeXML.Inline.model;`

Address of party responsible for a bibliographic entry.

`bib-volume` : `%LaTeXML.Inline.model;`

Volume of a bibliographic entry.

`bib-number` : `%LaTeXML.Inline.model;`

Number of a bibliographic entry.

`bib-pages` : `%LaTeXML.Inline.model;`

Pages of a bibliographic entry.

`bib-part` : `%LaTeXML.Inline.model;`

Part of a bibliographic entry.

`bib-date` : `%LaTeXML.Inline.model;`

Date of a bibliographic entry.

`bib-edition` : `%LaTeXML.Inline.model;`

Edition of a bibliographic entry.

`bib-status` : `%LaTeXML.Inline.model;`

Status of a bibliographic entry.

`bib-type` : `%LaTeXML.Inline.model;`

Type of a bibliographic entry.

`bib-issn` : `%LaTeXML.Inline.model;`

ISSN of a bibliographic entry.

`bib-isbn` : `%LaTeXML.Inline.model;`

ISBN of a bibliographic entry.

`bib-doi` : `%LaTeXML.Inline.model;`

Document Object Identifier of a bibliographic entry.

`bib-review` :(#PCDATA | `%LaTeXML.Inline.mix;` | `bib-mr`)*

Review of a bibliographic entry.

`bib-mr` : `%LaTeXML.Inline.model;`

Math Review number of a bibliographic entry.

`bib-mrnumber` : `%LaTeXML.Inline.model;`

Math Review number of a bibliographic entry.

`bib-mrreviewer` : `%LaTeXML.Inline.model;`

Math Review Reviewer of a bibliographic entry.

`bib-language` : `%LaTeXML.Inline.model;`

Language of a bibliographic entry.

`bib-url` : `%LaTeXML.Inline.model;`

A URL for a bibliographic entry.

`bib-eprint` : `%LaTeXML.Inline.model;`

Eprint (url?) for a bibliographic entry.

`bib-preprint` : `%LaTeXML.Inline.model;`

Preprint (url?) for a bibliographic entry.

`bib-note` : `%LaTeXML.Inline.model;`

Notes about a bibliographic entry.

`bibitem` :((`tag`)?, (`bibblock`)*)

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: `%LaTeXML.ID.attrib;` , `key`

`bibblock` : `%LaTeXML.Inline.model;`

A block of data appearing within a `bibitem`.

Module `structure`

`document` :((`%LaTeXML.FrontMatter.class;`)*, (`%LaTeXML.Para.mix;`)*, (`part`)*,(`chapter`)*,(`section`)*, (`%LaTeXML.BackMatter.class;`)*)

The document root.

attributes: `%LaTeXML.Labelled.attrib;`

`part` :((`%LaTeXML.SectionalFrontMatter.class;`)*, (`%LaTeXML.Para.mix;`)*, (`chapter`)*)

A part within a document.

attributes: `%LaTeXML.Labelled.attrib;`

`chapter` :((`%LaTeXML.SectionalFrontMatter.class;`)*, (`%LaTeXML.Para.mix;`)*, (`subparagraph`)*, (`paragraph`)*, (`subsection`)*, (`section`)*)

A Chapter within a document.

attributes: `%LaTeXML.Labelled.attrib;`

`section` :((`%LaTeXML.SectionalFrontMatter.class;`)*, (`%LaTeXML.Para.mix;`)*, (`subparagraph`)*, (`paragraph`)*, (`subsection`)*)

A Section within a document.

attributes: `%LaTeXML.Labelled.attrib;`

`appendix` :((`%LaTeXML.SectionalFrontMatter.class;`)*, (`%LaTeXML.Para.mix;`)*, (`subparagraph`)*,(`paragraph`)*, (`subsection`)*, (`section`)*)

An Appendix within a document.

attributes: `%LaTeXML.Labelled.attrib;`

`subsection` :((`%LaTeXML.SectionalFrontMatter.class;`)*, (`%LaTeXML.Para.mix;`)*, (`subparagraph`)*,(`paragraph`)*, (`subsubsection`)*)

A Subsection within a document.

attributes: `%LaTeXML.Labelled.attrib;`

`subsubsection` :((`%LaTeXML.SectionalFrontMatter.class;`)*, (`%LaTeXML.Para.mix;`)*, (`subparagraph`)*,(`paragraph`)*)

A Subsubsection within a document.

attributes: `%LaTeXML.Labelled.attrib;`

`paragraph` :((`%LaTeXML.SectionalFrontMatter.class;`)*, (`%LaTeXML.Para.mix;`)*,(`subparagraph`)*)

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

attributes: `%LaTeXML.Labelled.attrib;`

`subparagraph` :((`%LaTeXML.SectionalFrontMatter.class;`)*, (`%LaTeXML.Para.mix;`)*)

A Subparagraph within a document.

attributes: `%LaTeXML.Labelled.attrib;`

`bibliography` :((`%LaTeXML.SectionalFrontMatter.class;`)?, (`biblist`)*)

A Bibliography within a document.

attributes: `%LaTeXML.Labelled.attrib;` , `files`

`files` is the list of bib files used to create the bibliograph.

`index` :((`%LaTeXML.SectionalFrontMatter.class;`)?, (`indexlist`)*)

An Index within a document.

attributes: `%LaTeXML.Labelled.attrib;`

`title` : `%LaTeXML.Inline.model;`

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

`toctitle` : `%LaTeXML.Inline.model;`

The short form of a title, for use in tables of contents or similar.

`subtitle` : `%LaTeXML.Inline.model;`

A subtitle, or secondary title.

`personname` : `%LaTeXML.Inline.model;`

A person's name. NOTE: This should be aligned with Bibname.

`creator` :(`%LaTeXML.Person.class;`)*

Generalized document creator.

attributes: `role`

`role` indicates the role of the person in creating the docment. Values include author, editor and translator, but is open-ended to support extension.

`contact` : `%LaTeXML.Inline.model;`

Generalized contact information for a document creator.

attributes: `role`

`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.

`date` : `%LaTeXML.Inline.model;`

Generalized document date.

attributes: `role`

`role` indicates the relevance of the date to the document. Values include creation, but is open-ended to support extension.

`abstract` :(`%LaTeXML.Block.mix;`)*

A document abstract.

`acknowledgements` : `%LaTeXML.Inline.model;`

Acknowledgements for the document.

`keywords` : `%LaTeXML.Inline.model;`

Keywords for the document. The content is freeform.

`classification` : `%LaTeXML.Inline.model;`

A classification of the document.

attributes: `scheme`

`scheme` attribute can record what classification scheme was used.