The GLOSS system for transformations from plain text to XML

Richard Kaye, School of Mathematics, University of Birmingham

2006-06-05

Glosing is a full glorious thing certain,
For letter slayeth, as we clerkes sayn.

from The Summonner's Tale,
By Geoffrey Chaucer

1 Inputing mathematics as XML

Mathematical texts are complicated, and XML representations of mathematics rightly insist on accuracy and attention-to-detail. This asks a lot of any human being required to enter mathematical data. But the potential of the web and XML-based applications for mathematics cannot be realised until professional mathematicians in particular are persuaded of the advantages and presented with suitable tools to author their texts that are as straightforward as the ones they are used to. Most working mathematicians see the main product of their work as being the 'mathematical' content, rather than 'metamathematical' details such as the precise details of the system or systems being worked in, and naturally think of common mathematical operators in a textual form such as 'f-of-x' or $f\,(x)$ instead of a more conceptually rigorous form such as 'apply-f-to-x' or $\operatorname{apply}(f, x)$. They are not naturally sympathetic to systems that require them to enter 'unnecessary' or 'obvious' extra details.

Professional mathematicians have, however, shown willingness to work with text-based systems rather than the WYSIWYG point-and-click offerings in the commercial and home-base markets. In particular LaTeX is used as the main format for text and mathematical input. LaTeX and its text-based input system provides greater fine-control over mathematical input. Also, LaTeX has a number of important practical advantages including the availability of many extensions that add new macros or other functionality. LaTeX source files can be authored in most text editors. However, complex LaTeX code (for long equations and displays for example) is still difficult to write, difficult to set out in a readable form in a text file, and hence can be difficult to maintain. The macro facility in LaTeX is powerful, but makes parsing general LaTeX sources problematic, and LaTeX syntax lies uncomfortably under the fingers on a normal keyboard and is idiosyncratic in many ways, especially in its use of non-standard macro combinations for standard letters represented by Unicode characters. Nevertheless LaTeX should be the yardstick against which any document-preparation system for mathematical texts should be compared.

We are left in a Catch-22 situation: the typical mathematician seems unaware of progress in mathematical XML or sees little advantage in it due to lack of examples from the mathematical community, whereas the promise of mathematical XML will not be realised without input from mathematicians.

One of the goals of the work being described here is to explore ideas towards alternative systems which are as easy (or easier) to use as LaTeX for a professional mathematician. The new system should have all of the advantages, such as extendibility, of LaTeX, be at least as easy to use, and hopefully have many other advantages such as applicability of standard tools such as Unicode tools, XML tools, and a wider range of semantic defaults. I make no apology that the main test-case 'professional mathematician' that I have used here is myself. If the reader prefers, this document could be read as a list of personal desiderata for a working system for XML and mathematics. However, a useful and very general system for text-to-XML conversions, called GLOSS, has emerged from this work and this will be described in its bare-bones form too. It is suggested that a specific GLOSS-application might be developed that would be suitable as a LaTeX-substitute, providing a general, extensible, system for input of complex XML documents, including mathematical documents. GLOSS should have other applications too, including use as a general parser for legacy text documents, so may well be of interest in a wider field.

I should stress that text-based systems are for technically minded users who require full control over a document. For other users, point-and-click XML editors will always have a part to play, probably even the major part. These work well for small quantities of text, especially when the author is not familiar with the format in question and the editor can direct the author's choices; but for trained, specialist authors, or for work which involves producing and processing a large number of similar documents, they are rarely the most productive solution. Specialists may well have their own preferred text editing environment, and a specific tailored XML editing environment may not be compatible with this. In addition, point-and-click XML editors often turn out to be prescriptive and restrictive, or difficult to use by an author with an unusual or very specific task in mind.

2 Transforming plain text to XML

From the previous section, we have seen a need for a plain-text format (or set of formats) for mathematical XML. Because of the wide variety of mathematical texts, and because of the ever-changing nature of XML applications for mathematics, it would seem sensible to consider text-processors that handle arbitrary XML as output. Therefore we must investigate text-to-XML converter systems in general, in particular ones with the following key features.

- The system should provide coding methods for arbitrary XML

- Input code should be easy and fast to type by hand.
- Input code should be easy to structure and read.
- The system should be configurable, extensible and scriptable
- The system should be editor-neutral, that is, it should be easy to type in any text editor.
- The system should provide coding methods for standard OpenMath and MathML data, in particular datatypes such as arbitrary precision integers and base64 data
- The system should provide sensible defaults for mathematical and other constructs, but it should always be possible to over-ride these defaults.

In addition, the following is highly desirable:

- It should be possible to configure the system to process 'legacy' text formats

I shall take a short look at some of the off-the-shelf solutions currently available.

The first and perhaps most obvious one is to author text in XML and process it to the required target with XSLT. Defaults and other features could be provided by an XSLT stylesheet. Where this scheme falls down is in the ease-of-authoring and editor-neutrality. Without special editor tools, XML is prone to parse errors, or validation errors, and in any case the < and > characters in tags names do not lie easily under the fingers. For a typical mathematical document, coding time is certainly not reduced (compared to LaTeX) and is most likely significantly increased. However, direct input of XML and conversion with XSLT remains a possible solution, provided one is willing to give up the prospect of converting legacy text documents.

The next idea is to use a standard LaTeX-to-MathML converter. Clearly no such converter will ever fully succeed, as LaTeX is not fully-Unicode compliant and LaTeX macros are too general to be interpreted by anything other than the full LaTeX macro engine itself. (If the basic TeX engine was re-written to accept Unicode and to output XML or even HTML+MathML rather than DVI or PDF, this might be a more interesting prospect.) The LaTeX syntax itself is not particularly elegant and does not easily provide for general XML names, for example. (TeX and LaTeX use a different character set for macro names, even disallowing the digits 0-9 so native macro names in the form `\name` cannot be used.) Also, LaTeX's syntax does not include provision for many necessary semantic notions for MathML and other target formats, though these could be added, possibly at some cost to usability. From a personal point of view, I regard the standard LaTeX-to-XML software as an interim measure for very simple LaTeX code only. I have experimented with standard LaTeX-to-MathML converters using my own legacy documents as test input and been disappointed every time. However, LaTeX-to-MathML converters may well work for documents that use a limited subset of LaTeX, in particular new documents specially ritten for them. I am not sufficiently familiar with such convertors to know how flexible or extensible they are.

These considerations lead one to consider at a minimum a general parser to construct XML from text files by scanning tokens corresponding to XML element-names, text data, arbitrary precision integers, base64 data, and the like. My first experiments with this idea included a simple parser based on a fixed Python-like syntax followed by a post-processing XSLT stage. This worked, but was not easily configurable nor, it turned out, as easy to enter data as I hoped. The eventual solution I was lead to was to use a configuration file (rather like a stylesheet) describing the various separate 'modes' for mathematics and text processing according to the context. This enables the system to be programmable to many example text formats and allows a target application to be exactly tailored to a concise and precise source-format which is much shorter and easier to type.

The resulting system is called GLOSS (for 'Gloss Linguistic Or Semantic Structure'), a system written to convert plain text files to XML with mark-up added automatically. The GLOSS processor is a general purpose tool that reads plain text files and 'glosses' them, i.e., extracts structural information and adds inferred meanings or mark-up in the form of XML tags, writing well-formed XML as output. It is not necessary to use it as anything other than a text-to-raw-XML converter for post-processing with XSLT, though it has an extensible built-in style-sheet language similar to XSLT. (GLOSS configuration files or stylesheets are called 'modular vocabularies' because they define a number of 'modes'.) Some aspects of the output XML are easier to control in GLOSS: this includes the Python-like syntax (which can be over-ridden in a number of ways) and control of the number of arguments of a variable-argument element. On the other hand GLOSS does not have the full input text or the full XML tree-structure in memory at any given time, so cannot perform some of the operations easy in XSLT.

For general use, GLOSS is intended for authors with detailed knowledge of XML and the target application they are interested in. However, GLOSS is provided with a number of modular vocabularies for standard applications (including authoring target-formats such as XHTML+MathML) and these more specialised GLOSS applications should require rather less background knowledge to use. The GLOSS application for XHTML+MathML is intended to be at least as user-friendly as easy to use and extend as LaTeX and is appropriate for shorter papers and longer collections of web pages and papers. Actual examples include this paper, and further examples will be cited below.

3 GLOSS: overview

At its heart, GLOSS consists of a parser that transforms an input text file to an XML document. The transformation is controlled by a Modular Vocabulary (MV) document written in XML and analogous to an XSLT style-sheet [XSLT]. At any moment, GLOSS is working in one particular Mode (analogous to a template), reading tokens from the text via a tokeniser object, and inserting data in a particular point of the output XML tree.

The original idea for a Modular Vocabulary was a set of look-up tables of valid names usable in different contexts, the number of arguments each of these names takes, and what each one translate to. This remains one important application of Modular Vocabularies, and one which can simplify user-input considerably, but in GLOSS, MVs are much more powerful than this.

Conceptually, Modes and XSLT templates are similar, but one important difference is that a computational instance of a mode may accept and process an arbitrary number of tokens until some condition such as the required number of tokens being read, a recognised 'end' token, or even an end-of-stream marker, indicates it should stop. On the other hand, an instance of a template only processes the current node. This will be seen in operation in the examples below.

One of the strengths of GLOSS is that the tokeniser can be configured by the mode for the token-types required and even the number of tokens required. So there may be one mode for PCDATA, where the tokens are the individual Unicode characters, another for more complex mark-up, where the tokens might be XML element names, and so on. A number of standard token types are available, including ones for XML element names, attribute names, etc., and also token types for arbitrary precision integers, hexadecimal numbers, floating point numbers, strings, individual characters, URIs, and blocks of base-64 data.

In fact, one is not limited to a mode for each kind of token: different modes may process tokens in different ways, adding default XML structure in a context where this may be inferred. This can massively reduce the burden of typing the source. For example, one simple application of GLOSS I have developed converts source text files to XHTML with embedded presentation MathML [MathML] and in mathematics mode, automatically wraps characters in the most appropriate of the standard tags, `mo`, `mi` or `mn`—if they are not already so-wrapped. It also counts the number of arguments of elements such as `mfrac`, `msub`, `munderover`, etc., so that much of the tree-structure can be correctly inferred.

Where it is necessary to present XML tree-structure, the standard GLOSS applications use a Python-like syntax based on indentation and only over-ruled by braces where necessary. The tokeniser has a number of features that enable syntax to be defined using indentation, but it is the MV file itself that defines whether and precisely how this is done. In other words, indentation is a feature built into the tokeniser for the application MVs to choose whether or not they use it.

The use of a Python-like language for XML is not a new idea, admittedly, but I believe the stylesheet-like MV language for describing parsing is novel. As well as the obvious commands for creating elements, attribute nodes, text and CDATA nodes, this MV language has a number of other interesting features, but a full discussion of these would take us too far off-track here.

Information is passed from one mode to another via *parameters*, variables that hold Unicode strings as values, and these enable the MV files to be modularised. A typical MV file is a minimal 'driver' that includes a number of modules, and by a mechanism of hooks using parameters, a new module hooks into an existing module, so several different modules may be chosen by the user to select a particular format. For example, the main HTML module extends the standard XML module provided, and other modules are provided for providing extra convenience and functionality for authoring HTML files, such as section-numbering, recursive subsections and titles, an extended system for hyper-references, a system for presenting theorems and proofs, in HTML, the Dublin-Core metadata, and of course a number of MathML modules. (All of this is mapped by GLOSS in a uniform and consistent way to to standards-compliant XHTML.) There is also a GLOSS module for writing further modules making writing further modules much more straightforward.

GLOSS parses input text and produces an output XML document. This document initially exists in computer memory but is usually printed out to a file in the usual XML format for saving and later use. The document can then be transformed, using XSLT for example, either from the on-memory copy or on re-reading from disk file. A number of XML features, such as entity and character references, document type identifiers, DTDs and even the internal DTD subset are often important for subsequent applications but cannot be expressed in the usual XML infoset [XMLinfo]. Moreover for aesthetic considerations the user may wish to have further control over the final appearance of the XML document, and these are not normally expressible using the usual DOM tools [DOM]. GLOSS solves these problems in a flexible and standards-compliant way by using an internal representation of an XML document with additional markup for the various elements of XML and how it should be printed in the final version, written in XML itself. This XML representation document type and its DTD could be useful for other XML applications, such as an XML editor, in its own right.

In principle, it is also possible to parse a wide variety of other existing or legacy formats, though with a couple of exceptions the standard ones that are provided at present are all based on the standard XML module. The exceptions are modules to extract data from comments in XML files, and to extract data from comments in GLOSS input files. These are used in the system itself as self-documentation tools.

GLOSS is detailed at `http://gloss.bham.ac.uk`, where you can download a working version and all the documentation. GLOSS is a Java program of around 5000 lines, together with a number of DTDs, MVs, GLOSS source files and documentation. It should run on any platform. It is being made freely available under the Gnu Public Licence [GPL].

4 A minimal GLOSS MV for XML

This section presents a simple example of a modular vocabulary for XML. The standard GLOSS-xml vocabulary is based on the ideas here, but is much more sophisticated and flexible. Background knowledge of XML is assumed throughout this section; the examples taken are from presentation-MathML, though GLOSS modes can be written for any XML vocabulary.

The idea is to use indentation to represent tree-structure and `[` and `]` to delimit text. (The characters `[` and `]` were selected as they are much more conveniently located on most keyboards and are rarely used in text itself.)

The idea is that an input text file such as

```
math
  mrow
    mi[A]
    mo[=]
    mfenced @open[(] @close[)]
      mtable
        mtr
          mtd mi[x]
          mtd mi[y]
        mtr
          mtd mi[z]
          mtd mi[w]
```

should result in

```
<mrow>
  <mi>A</mi>
  <mo>=</mo>
  <mfenced open="(" close=")">
    <mtable>
      <mtr>
         <mtd><mi>x</mi></mtd>
         <mtd><mi>y</mi></mtd>
      </mtr>
      <mtr>
         <mtd><mi>z</mi></mtd>
         <mtd><mi>w</mi></mtd>
      </mtr>
    </mtable>
  </mfenced>
</mrow>
```

This (and many other transformations like it) is achieved with the MV

```
<?xml version="1.0"?>
<!DOCTYPE mv:modularvocab
    SYSTEM "http://gloss.bham.ac.uk/dtd/mv/modularvocab.dtd">
<!--

minimalxml.mv: Minimal MV to process XML with elements, text
and attributes, producing a document.

Richard Kaye. May 2006. Licence: GPL. Warranty: none.

-->

<mv:modularvocab xmlns:mv="http://gloss.bham.ac.uk/xmlns/modularvocab">

<mv:mode name="main" accept="elt" children="1">
  <mv:match type="elt">
    <mv:document>
      <mv:element>
        <mv:process-tokens mode="elt-content"/>
      </mv:element>
    </mv:document>
    <mv:return />
  </mv:match>
```

```
</mv:mode>

<mv:mode name="elt-content" accept="elt|attr|[">
  <mv:match type="elt">
    <mv:element>
      <mv:process-tokens mode="elt-content"/>
    </mv:element>
  </mv:match>
  <mv:match type="attr">
    <mv:attribute>
      <mv:process-tokens mode="text"/>
    </mv:attribute>
  </mv:match>
  <mv:include mode="text"/>
</mv:mode>

<mv:mode name="text" accept="[">
  <mv:match type="punc">
    <mv:text>
      <mv:process-tokens mode="text-content"/>
    </mv:text>
  </mv:match>
</mv:mode>

<mv:mode name="text-content" accept="]|uc" use-indentation="false">
  <mv:match type="punc">
    <mv:return/>
  </mv:match>
  <mv:match type="uc">$v</mv:match>
</mv:mode>

</mv:modularvocab>
```

Hopefully the reader will be able to guess how this code operates. In any case, there is much more information in the documentation section of the web site. As explained, modes get and process tokens until the rules about indentation, number of child-tokens or an explicit `return` command require them to stop. The `accept` attribute lists the token-types a mode accepts, separated by `|`. The modes above use the token-types for unicode-characters, elements and attributes, as well as the explicitly given punctuation tokens `[` and `]`. The token read is matched against the mode's child nodes. It is also possible to match against the token's data as well as its type, but this was not required here. The string `$v` represents the token's value. The content of the match is data to be inserted into the output or commands controlling GLOSS's behaviour. The `text`, `element` and `attribute` commands make output nodes of the obvious kinds, and they have convenient defaults for the name of the element or attribute to be inserted.

The real MV for XML provides the use of braces to override indentation when necessary, with the rule that an XML group cannot cross { or }. It also provides a feature that allows the user to 'push' back into element-mode from text mode, so that HTML code such as '`ordinary text, <i>italic</i> text, and <b>bold</b> text`' have the convenient and easy-to-type form '`ordinary text, [i[italic]] text, and [b[bold]] text`'.

The real MV for XML also provides support for CDATA, PI, comments and other XML features, including DTDs. Other standard MVs extend the basic one in other ways; the p-MathML module knows the usual MathML names for mathematical tokens such as `alpha`, `beth`, `c`, `=`, and so on and automatically wraps then with the most appropriate of `mi`, `mo`, `mn`, etc., when they are not already so-wrapped. Further extension modules can easily be written. For example if the matrix construction given here was a common one in a document an extension mode could easily be written to allow it to be input using the syntax

```
math
  mrow
    A =
    matrix
      x y
      z w
```

or

```
math mrow A = matrix {
  x y
  z w
}
```

for it.

Further GLOSS modules for processing XHTHML files include modules for presenting definitions, theorems, and proofs, sections and subsections with automatic numbering, support for Dublin Core metadata, and others. Each of these is selected for inclusion in the main driver MV and automatically 'hooks' onto the base module without further input required from the user.

Finally, as an experiment and quick demonstration of the programmability and configurability of GLOSS, I have written an alternative MV (available in the 'doc' directory of the main GLOSS distribution) which uses a TEX-like syntax, and in which the same example is encoded as

```
\math{
  \mrow{
    \mi{A}
    \mo{=}
    \mfenced[open="(" close=")"]{
      \mtable{
\mtr{\mtd{\mi{x}}\mtd{\mi{y}}}
\mtr{\mtd{\mi{y}}\mtd{\mi{z}}}
      }
    }
  }
}
```

This format does not use the Python-like indentation to present structure but instead uses braces. (Indentation is used in the above example for aesthetic considerations only.) Obviously shortcuts can be devised to make this input more palatable to the general user. The format is not TEX (for example `\testing123` would be a valid element name in this format), but is sufficiently familiar to (La)TEX to be of interest. However, my personal preference is for the Python-like syntax without \ and so many braces as is it easier to type and clearer to read.

5 Further examples

Like other mathematicians, the main focus of my mathematical work is necessarily towards written mathematical texts, though GLOSS is not exclusively aimed in this direction. I am using GLOSS for almost all of my papers and web pages (including this one) and will make the sources available via web pages. Whilst developing GLOSS I have been using it for two major mathematical projects and have been delighted with the extra flexibility that XML provides and improved productivity compared to LaTeX.

I have been working on a textbook on logic for mathematics students, which is now almost finished. The main objectives are to produce a paper-based text with supporting web pages. The final typesetting will be done by latex, and I used GLOSS to convert plain text sources to an intermediate XML form for conversion by XSLT to LaTeX or XHTML+p-MathML. The web page is at `http://web.mat.bham.ac.uk/R.W.Kaye/logic/` though much of the the material I have written is not available there for copyright reasons.

I have also produced a comprehensive set of web pages for an introductory real analysis course I gave at Birmingham University. The results are available at `http://web.mat.bham.ac.uk/R.W.Kaye/seqser/`, and full GLOSS sources are available there. These pages were transformed by GLOSS to standard p-MathML+XHTML directly, with no final transformation by XSLT and a very minimal and optional CSS style-sheet. The GLOSS modes also provide several extra facilities such as boiler-plate text, as well as the features already mentioned in the standard GLOSS-html modules. A XSLT style sheet was used to convert XHTML+p-MathML to standard HTML for browsers not equipped to display MathML, and content negotiation is used (with some notable problems on the client side) on the server to provide the right version of the document.

6 References

DOM `http://www.w3.org/DOM/`, Document Object Model. W3C.

GLOSS `http://gloss.bham.ac.uk`, Web page for GLOSS.

GPL `http://www.gnu.org/copyleft/gpl.html`, Gnu Public Licence.

MathML `http://www.w3.org/TR/MathML2/`, Mathematical Markup Language (MathML) Version 2.0. W3C, 21 October 2003.

XMLinfo `http://www.w3.org/TR/xml-infoset/`, XML Information Set. W3C, 4 February 2004.

XSLT `http://www.w3.org/Style/XSL/`, The Extensible Stylesheet Language. W3C.

Richard Kaye
School of Mathematics
University of Birmingham
`http://web.mat.bham.ac.uk/R.W.Kaye/`