The GLOSS system for transformations from plain text to XML

Richard Kaye, School of Mathematics, University of Birmingham

2006-05-24

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

from The Summonner's Tale,
By Geoffrey Chaucer

1 Transforming plain text to XML

Mathematical texts are complicated, and XML representations of mathematics rightly insist on accuracy and attention-to-detail. This asks a lot of a mere mortal 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.

Point-and-click XML editors will always have a part to play, perhaps even the major part. But for trained, specialist authors they may not be the most productive solution. Most professional mathematicians are happy using LaTeX in a form without any of these facilities. Some form of encoding of mathematics as easy-to-type plain text is still required, at least in the medium term.

One of the advantages of LaTeX is that it has separate 'modes' for maths and text and is thus able to process input in different ways according to the context. This enables the source to be much shorter and easier to type. There are a number of LaTeX-to-XML converters and LaTeX-like text encodings for XML available. Unfortunately, I have never been very impressed by these since:

- due to the nature of LaTeX, and in particular macros, LaTeX-to-XML converters cannot always do a good job on all input documents; and
- LaTeX syntax is at best limited and not sufficient to express the rich mathematical semantics more modern processors require.

A useful text-to-XML processor, to my mind, would be one that is fully configurable and able to produce arbitrary XML. Its input should also be easy to type and it should be able to take advantage of the context and sub-contexts of the text at the point being processed, to reduce the work required on the part of the author. A secondary objective is that it should be possible to configure it to process legacy text documents as well as new documents.

These were the objectives of my system GLOSS (for 'Gloss Linguistic Or Semantic Structure'), a system written to convert plain text files to XML with mark-up added automatically. This is a general purpose tool to extract structural information from a text file and write well-formed XML as output. The glossing that is performed follows rules expressed in XML in a form somewhat similar to an XSL style-sheet. GLOSS is primarily intended for authors with background knowledge of XML and the XML application they are writing for, though GLOSS applications for some target formats (such as XHTML+MathML) are provided with the system and these may require less background knowledge to use.

2 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 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, and although this is still an important aspect, MVs are much more powerful than this, but the name has remained. (One important difference between modes and XSLT templates is that a computational instance of a mode may accept and process more than one token, whereas an instance of a template only processes the surrent 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, and 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 arbitary 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 `mo`, `mi` or `mn` tags 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.

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, and 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 a XML document with additional mark-up for the various elements of XML and how it should be printed in the fanal 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://web.mat.bham.ac.uk/R.W.Kaye/gloss/`, 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].

3 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. 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://web.mat.bham.ac.uk/R.W.Kaye/gloss/dtd/modularvocab/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://web.mat.bham.ac.uk/R.W.Kaye/gloss/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 uses braces to override indentation when necessary, as well as support for CDATA, PI, comment and other XML features. 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.

4 Further examples

I am using GLOSS for almost all of my papers and web pages (including this one) and will make the sources available. 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. 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 objective 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.

5 References

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

[GLOSS] `http://web.mat.bham.ac.uk/R.W.Kaye/gloss/`, 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/`