Managing a math exercise database with LaTeX

PÉTER SZABÓ
Budapest University of Technology and Economics
Dept. of Computer Science and Information Theory
H-1117 Hungary, Budapest, Magyar tudósok körútja 2.
`pts (at) fazekas dot hu`

ANDRÁS HRASKÓ
Fazekas Mihály Fővárosi Gyakorló Ált. Isk. és Gimnázium
Horváth Mihály tér 8.
H-1082 Budapest, Hungary
`hraskoa (at) fazekas dot hu`

Abstract

TeX is a good tool for creating beautiful books, especially when the book contains a lot of math formulas. It is not rare that TeX is used to typeset a view of a database, by generating TeX source from the database text, possibly using XML as an intermediate format. Some TeX packages and formats support reading XML data directly.

In the matbook *project we have created a database of math exercises for special class secondary school students, as well as solutions and instructions for teachers. The data is organized in a tree structure of custom LaTeX environments in* `.tex` *source files. LaTeX reads these data files several times for generating the books. CVS is used for data replication and concurrent co-authoring. We are planning to switch to using a LaTeX-to-HTML translator to publish the database on the web.*

This paper presents the simple software architecture of the matbook *project and the design decisions we made concerning software and workflow, and it also compares* matbook *with other approaches such as big content management systems and TeX-enabled wikis.*

Non-standard use of TeX

The original purpose of TeX (and LaTeX) was typesetting beautiful books, journals and other printed material.

Novel uses include preparing slides for talks, developing software and its documentation together (e.g. web and ltxdoc), typesetting math formulas (e.g. Texvc [11]), typesetting printed and on-line HTML documentation together, rearranging PDF pages (pdfTeX with pdfpages.sty) and typesetting text generated from databases or other markup formats.

In the matbook project we use LaTeX to read a database of math exercises (in several passes), and typeset the material to books for students and teachers. This paper presents the software architecture and some implementation details of the matbook project, and it is also a case study of integrating excellent free software tools for low-budget publishing.

Project goals and products

The *Fazekas Mihály Secondary Grammar School* of Budapest [1] has been launching special mathematics classes for several decades, and is proud of its students winning national and international student competitions, and later becoming appreciated mathematicians. E.g. László Lovász, the well-known Hungarian mathematician, graduated from Fazekas in 1966.

Good mathematicians have good problem solving skills, and this skill can be best developed by solving problems and exercises. It is the responsibility of the teacher to choose the exercises for the students which best fit their learning curve. Talented students in a special math class need special attention. A lot of exercises and didactic experience have accumulated in Fazekas over the last few decades, and we have decided to publish this in printed form in Hungary; we are also planning to provide a web interface where all material is available. Thus matbook was born.

We are compiling a comprehensive exercise database (which also includes solutions, didactic advice, exercise lists for lessons and metadata for more accurate searching). Students and teachers in Fazekas are both working on extending this database, and we are developing software that would present this database to its audience. We are planning to publish exercise books (for students) and teachers' guides. If students buy the exercise books, teachers can give homework assignments from those books. (Of course, teachers will assign exercises whose solutions cannot be found in the exercise book.)

We are also planning to provide a web interface on which visitors can browse and view exercises, solutions etc., they can do a full text search, and they can also search for exercises in a given topic (specified using a set of predefined keywords). We already have a web interface for a comprehensive database of Hungarian secondary school math contest problems (which stores text in LaTeX format, and converts it to HTML using TTH [2]), and we'd like unify this with the matbook database.

Database structure

The database consists of

- *exercises* for the students;
- *hints* and *solutions* corresponding to the exercises, for the students;
- solutions for the teachers only;
- *remarks* and *didactic advice* corresponding to the exercises, for the teachers;
- a hierarchic taxonomy of *keywords* covering the fields of mathematics (e.g. prime numbers, trigonometry);
- association between exercises and keywords;
- organization of exercises into *chapters* and *volumes*;
- chapter and volume introduction text;
- ordered *exercise lists* prepared for obligatory and facultative lessons;
- *figures* referred to in the text, in EPS format.

Software components

- *volume typesetter:* a set of LaTeX macros to read the database in multiple passes, and typeset the book volumes;
- *indexer:* generates the keyword index at the end of the volumes (similar to *makeindex*);
- *web user interface:* with browse, view and search functionality;
- *consistence validator:* checks whether database files conform to the specifications.

Existing free software used: standard tools in a TeX distribution, the lmodern font family [3], GNU Ghostscript, sam2p [4], ImageMagick, CVS, Perl, the new magyar.ldf (part of [5]), husort.pl (Hungarian index processor, part of [5]), stuki.sty (structogram figure generator [6]).

We work in a Linux–Windows mixed environment, so it was our aim that all components except for the server part of the web user interface should run on both Unix and Win32. TeX tools we need are available on both systems. We decided to implement the indexer and the consistence validator as command-line Perl applications so it would be easy to port them across systems.

Database layout

We chose to store our data in structured text files rather than using a relational database, because it is easier to change the schema later, and we don't have to develop a custom user interface for data editing. XML is a good and widely supported structured text data model and syntax, but we prefer a format which is quick to type and easy to review for humans. YAML [7] is such a format. We finally chose the XML data model (for interoperability with other software), but a LaTeX-compatible syntax (for easy typing), which can be converted to XML without loss when needed.

As a master text markup format, we quickly rejected XHTML + CSS + MathML, mostly because it is tiresome to type a document in this format. Also, it is not possible to archive a rendered version of an XHTML text in a scalable way; it is not possible to specify typesetting hints (such as penalties); and MathML is not powerful enough: it is not possible to type the right, textual side of `\cases` in MathML; MathML still lacks some symbols. Moreover, with current browsers it is not possible to ensure acceptable visual quality: browsers render the same document differently, MathML support usually doesn't come out of the box, browser MathML fonts lack important symbols, browsers cannot hyphenate long words automatically, the visual output depends on the installed fonts and the browser window size (which the author of the text cannot control), browsers cannot break the line in the middle of a MathML formula etc.

We could have adopted a safe and easy to type markup format, similar to MediaWiki's WikiText format [8] or ŞäferTeX [9]. The MediaWiki software implements the text rendering engine of Wikipedia [10], and it lets authors insert math formulas in a subset of (AMS)LaTeX syntax. When the page is rendered, these formulas are interpreted and converted to images or MathML formulas by Texvc [11]. We have rejected MediaWiki because — as with XHTML — it doesn't give the author enough power to ensure perfect visual output quality. We did not use ŞäferTeX because its source code was not available, and it was not mature enough.

We could have invented our own markup format. Doing this would have required us not only to invent an excellent format, but to write a renderer (to both PDF and HTML), and document the format thoroughly, including tutorials and examples. This option was not feasible in our project.

Thus we have chosen a restricted subset of LaTeX as a markup format. Its advantages are: it has been available for a long time, the basic commmand set is well-documented, it gives the text author sufficient control over the visual quality of the output, and there are lot of fonts and packages we can use. We had to impose restrictions in order to keep our format convertible (primarily to XHTML + CSS + MathML). The most important restrictions on the document text are: it is forbidden to load packages or other files, define or change macros, use conditionals or other programming features, change catcodes, use the character `"` in the input (the "proper quotes" must be used), use conditionals, or insert figures with `\includegraphics` (we provide a more restricted command instead).

Once we settled on LaTeX as a text markup format, it was straightforward to use the same syntax for structuring the data, so that our database text files won't contain two alternative formats, and they can be syntax-highlighted or otherwise processed in text editors easily. However, plain LaTeX is not suitable for structuring. For example, it is not obvious to deduce where chapter "First" ends in this LaTeX source, without knowing the meaning and depth of `\section`:

```
\chapter{First}    \emph{First}  content.
\section{Inside}   \emph{Inside} content.
\chapter{Second}\label{2nd} % dummy
                   \emph{Second} content.
```

The XHTML representation (using `<H1>` for chapter titles and `<H2>` for section titles) suffers from the same limitation.

Our data format solves the problem by specifying structure using custom LaTeX environments. The example above looks like this:

```
\begin{mchapter}{title={First}}
  \emph{First} content.
  \begin{msection}{title={Inside}}
    \emph{Inside} content.
  \end{msection}
\end{mchapter}
\begin{mchapter}{title={Second},id={2nd}}
  % dummy
  \emph{Second} content.
\end{mchapter}
```

When converted to XML, it becomes:

```
<mchapter title="First">
  \emph{First} content.
  <msection title="Inside">
    \emph{Inside} content.
  </msection>
</mchapter>
<mchapter title="Second" id="2nd">
  <!-- dummy -->
  \emph{Second} content.
</mchapter>
```

Please note that `\emph` is not converted, because it is part of the text markup, and not part of the structure.

Thus there is a simple mapping between XML and our data format:

- LaTeX environment with attributes (i.e. "key = value" pairs) $\longleftrightarrow$ XML tag with attributes, properly escaped
- TeX comment $\longleftrightarrow$ XML comment
- other LaTeX text $\longleftrightarrow$ XML text (PCDATA)

This direct mapping makes it possible to use XML tools on our database. For example, we can use XSLT to do structural transformations on the XML, and we can use DTD or XML Schema validators to validate our database.

Database folders and files

The database is spread into several small text files in several folders. The files read each other (using `\input`). The points where the data must be split and the naming conventions for the files and folders are strictly regulated.

The database is replicated on each co-worker's machine, using the CVS [12] revision control system. People can work offline, and commit their changes back to the repository on the server a few times a day. Server failures and network connection slowdowns don't affect working hours seriously. CVS is smart enough to merge concurrent but independent changes of text files, and it enforces human interaction when a conflict occurs, so there is no danger of accidentally overwriting somebody else's changes. CVS also keeps old versions, so accidentally deleted text can be recovered any time later. (CVS merges files line-by-line, which complicates concurrent editing of binary files — but this limitation doesn't affect our project since we mostly use text files.) Subversion (= SVN [13]) is a newer and more advanced revision control system, and it won't be hard to migrate from CVS when those advanced features are needed. Both CVS and Subversion have clients on multiple platforms, including Unix and Win32. We use the standard `cvs` client on Linux, and TortoiseCVS on Windows.

The reason why the database is split into multiple files is that it is easier to transfer changes of smaller files in CVS, and it is also easier for humans to edit a few small files concurrently than to edit one large file. Usually multiple people are modifying the database at the same time, but most of the time they work in their own files, so no conflict occurs. Using multiple folders makes it easier to select the correct file for opening.

The file and folder layout also follows the LaTeX compilation process. Compilation always starts in the root folder (of the CVS tree). For example, to compile the first volume of Algebra, one runs "`latex volume_a_i`", which starts processing the file `volume_a_i.tex`. All other files belonging to this volume reside in the folder `chs_a_i` and its subfolders. Files `\input` are thus specified relative to the root folder, thus adding `../` is not necessary when referring to local `.sty` files. It is also convenient that all temporary and output files go to the root folder, thus subfolders are not changed during the compilation process.

LaTeX tricks

In this section we present some problems we faced when typesetting with LaTeX; solutions included.

Unified labels

This is a feature that makes it possible to refer to a `\label` defined in another volume. It is accomplished by reading `\newlabel` commands from the other `.aux` files, and adding them with both the label text and page numbers prefixed with the other volume name.

Volume split

Since teachers' guides can be several hundred pages long, it might be necessary to split them into multiple volumes. If this is so, the editor promotes some chapter boundaries to volume boundaries by adding the appropriate command to the source of the main `.tex` file. We chose the most portable ways to typeset these subvolumes: all subvolumes are separate LaTeX documents, which `\input` the main `.tex` file in a mode in which the `\shipout` of the unnecessary pages is cancelled.

The advantage of this method is that it doesn't require external tools (such as psselect), it works for both PostScript and PDF, and it can be run from a regular LaTeX IDE. Its disadvantage is its slowness. An alternative approach for PostScript would be generating and running a psselect command line for each volume. And for PDF, an alternative approach is selecting the appropriate pages from the main volume using pdfpages.sty.

Splitting the volume into real subvolumes (so that compiling a subvolume doesn't read the other subvolumes' LaTeX source) would not work, because it would render page numbers, the bibliography and inter-subvolume references incorrectly, or need additional programming.

Fuzzy keyword names

When specifying the list of keywords associated with an exercise, it is a big burden to specify a long keyword precisely (with spaces, punctuation etc.). In order to solve this, a Perl script generates keyword aliases, e.g. the first word of a keyword will become an alias for the keyword if this is not ambiguous.

String processing

Another idea in fuzzy keyword name matching is to match keywords after stripping spaces, punctuation, upper case and accents. This stripping had to be implemented in LaTeX, too. Since TeX doesn't have string processing primitives, we have to implement them using macros. Here is a mix of a macro definitions that shows the most important string processing tricks:

```
\def\stripit#1>{}\def\empty{}\def\space{ }
\def\rmonestar#1{\ifx#1\hfuzz\empty\else
  \if*\string#1\else#1\fi
  \expandafter\rmonestar\fi}
\begingroup\lccode`!` \lowercase{\endgroup
\def\oonespace#1 {\ifx\hfuzz#1\empty\else
  #1!\expandafter\oonespace\fi}}
\def\rmstars{%
  \afterassignment\rmstarsb\def\M}
\def\rmstarsb{%
  \edef\M{\expandafter\stripit\meaning\M
          \space\hfuzz\space}
  \edef\M{\expandafter\oonespace\M}
  \edef\M{\expandafter\rmonestar\M\hfuzz}}
```

The macro `\rmstars` above removes all stars (*) from a string. The string is given as an argument in braces, and the result — without the stars and all tokens having catcode 12 — is put into the macro `\M`. Example invocation: `\rmstars{a * B**cd} \show\M`

A rough outline of its operation: `\meaning` converts tokens to catcode 12, except for spaces, which are converted to catcode 10. Then `\oonespace` iterates over all spaces and converts them to catcode 12, too. Finally `\rmonestar` iterates over the tokens and removes all stars. Almost beautiful.

Read more about the primitives involved here in *The TeXbook* [14].

One or more solutions?

If there is one solution for an exercise, it should be prefixed with "Solution" (and not "Solution 1"). If there are more solutions, each of them should have a number, "Solution 1", "Solution 2" etc. By the time of emitting the 1st solution, we don't have the information whether there are more. How do we typeset it properly?

To solve problems like that, it is a common trick to use the `\label`–`\ref` mechanism. We emit `\label {exercise42-sol2}` at "Solution 2", and the next time the document is recompiled, at "Solution 1" we check for the presence of this label, e.g. with
`\@ifundefined{r@exercise42-sol2}{...}{...}`

Bibliography three times

When a `\cite` command with a new target is added to the document, it is necessary to run LaTeX three times: `latex doc; bibtex doc; latex doc; latex doc`. The 1st run of LaTeX records the `\citation` command to the `.aux` file. The BibTeX run generates the `.bbl` file. The 2nd run of LaTeX inserts the new `.bbl` file to the document, and it also records the `\bibcite` command to the `.aux` file indicating the new number to be displayed by the `\cite` command. The last, 3rd run of LaTeX makes `\cite` emit that number.

We speed this up by parsing the `.bbl` file at the beginning (before the first `\cite`), so the 3rd run of LaTeX is not necessary.

Conclusion and future work

The matbook project demonstrates not only the power, openness and flexibility of LaTeX, but is also an example of low-budget publishing using free software and a little scripting. matbook is also free software.

Our most important future goals are completing the exercise database and implementing the missing software components: a thorough consistence generator and the web user interface.

References

[1] Fazekas Mihály Secondary Grammar School of Budapest. `http://www.fazekas.hu/`

[2] TTH: the TeX to HTML translator. `http://hutchinson.belmont.ma.us/tth/`

[3] Bogusław Jackowski and Janusz M. Nowacki. *Latin Modern fonts: how less means more.* In proc. of EuroTeX 2005, pp. 172–178. `http://www.dante.de/dante/events/eurotex/papers/TUT09.pdf`, 2005.

[4] Péter Szabó. *Inserting figures into TeX documents.* In proceedings of EuroBachoTeX 2003.

[5] Péter Szabó. *Implementation tricks in the Hungarian Babel module.* In proc. of TUG 2004.

[6] Károly Lőrentey. *stuki.sty: Structograms in LaTeX.* `http://lorentey.hu/project/stuki.html.en`

[7] YAML: machine parsable data serialization format. `http://www.yaml.net/`

[8] WikiText: wiki markup language. `http://en.wikipedia.org/wiki/Wikitext`

[9] Frank Schäfer. *ŞäferTeX: Source Code Esthetics for Automated Typesetting.* In proc. of TUG 2004.

[10] Wikipedia: the free encyclopedia that anyone can edit. `http://en.wikipedia.org/`

[11] Texvc: TeX validator and converter. `http://en.wikipedia.org/wiki/Texvc`

[12] Karl Fogel and Moshe Bar. *Open Source Development with CVS.* 3rd Edition. O'Reilly, 2003.

[13] Ben Collins-Sussman et al. *Version Control with Subversion.* O'Reilly, 2004.

[14] Donald E. Knuth. *The TeXbook.* Addison-Wesley, 1984.