xml2tex: An easy way to define XML-to-LaTeX converters

Keiichiro Shikano

Abstract

xml2tex is a framework to give XML a presentation layer using LaTeX. In other words, xml2tex lets you use an XML-based format as a source of LaTeX. It may sound awful at first, but an XML-based format has some advantages, especially for creating books. This paper describes why XML does matter, and introduces xml2tex's intuitive way of relating XML to LaTeX, based on a Scheme dialect and SXML.

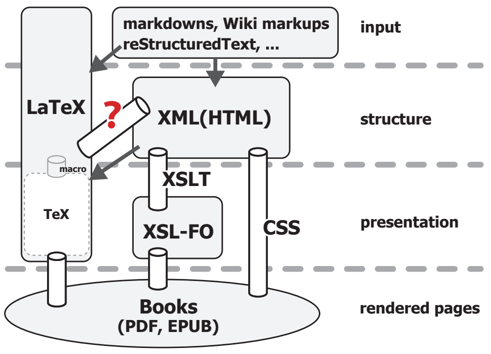

Figure 1: Ladder of abstraction in creating books

1 LaTeX as presentation for XML

Creating documents can be seen from two opposite aspects: structure versus presentation. In some document systems, they are divided into completely separate layers. For example, XSL [10] is the way to define a presentation of XML, which corresponds to the tree structure of a document. Håkon Wium Lie, the father of CSS [11], explains this separation in terms of a ladder of abstraction [6]. The structural tree of the document is at the highest abstraction level. Moving downwards on the ladder, the document becomes less and less abstract towards the rendered data. It's hard to climb the ladder without any manual aid. That means that reusing the document in other media requires much manual work.

Oddly enough, this separation is rather loose in LaTeX, despite the fact that LaTeX originally is the structured layer over the lower-level typesetting mechanism provided by the TeX engine. This weak separation can sometimes make reusability of LaTeX documents problematic. E.g., if you want to distribute a LaTeX document through the Web, chances are that it will be as a PDF.

Figure 1 shows the abstraction ladder for purposes of creating books, our main concern. Arrows in Figure 1 indicate that the translation or mapping between the formats is achieved by following some restrictions. In other words, the expressiveness of your document would be limited in accordance with the formats in higher abstraction levels. The left-down arrow from XML to LaTeX, for example, refers to a system that can transform XML files written in some given DTDs or XML Schemas, such as DB2LaTeX [2] (a converter from DocBook to LaTeX) or TeXML [7] (a feasible XML syntax for TeX). In those systems, you can hardly create a book requiring more structural elements than the specifications offer. Similarly, the easy-to-read-and-write input formats like markdown and Wiki syntaxes narrow the possible expression of documents down to their intended use. This can be very good for writing and editing the texts, but not for supporting a variety of page layouts.

On the other hand, the plumbing pipes connecting different formats indicate that there's practically no restriction to a downward direction. Needless to say, LaTeX is able to produce almost any possible page designs; as we'll see, this is one of the most important reasons we'd like to use LaTeX in creating books. The same is true for XSL (including XSLT and XSL-FO), regarded as the best path to render a variety of page layouts from a tree structure of XML. XML also has CSS as the mechanism to apply an arbitrary style to the tree.

What is missing here is a feasible mechanism for producing LaTeX from non-restrictive XML. The most common approach for now is to use XSLT. However, XSLT is meant to convert an XML into another XML, so it lacks support for writing XML-to-LaTeX converters. Another approach to providing a mechanism suitable for LaTeX is XMLTeX [1]: an XML parser written in TeX. This is a great accomplishment in terms of TeX macro programming, but we did not find it easy to write our required converter using XMLTeX. A more practical approach is to use ConTeXt's XML support [3]; this supports a declarative interface to select an XML element and define the corresponding ConTeXt syntax. When we are able to use ConTeXt in typesetting Japanese books, it will be a good alternative to our own attempt, called xml2tex [9], described in the following.

2 Defining maps from XML element tags to LaTeX syntax, the xml2tex way

Let's start with a silly HTML example.

```
<html>
Lorem % ipsum \ ... $10,000
</html>
```

Leaving aside the escaping of special characters (‘%’, ‘\’ and ‘$’), we have to decide how to express this HTML in LaTeX. One feasible representation is achieved by mapping its only element (`<html>`) to `\begin{document} ... \end{document}`. Here is the xml2tex way to do this:

```
(define-tag html (make-latex-env 'document))
```

That's it![1] Put this line down and save it as the file `silly.rule`, then run xml2tex like this:

```
$ xml2tex --rule="silly.rule" sample.html
\documentclass{book}
\usepackage[T1]{fontenc}
\begin{document}
Lorem {\symbol{37}} ipsum {\symbol{92}} ...
{\symbol{36}}10,000
\end{document}
```

Special characters are automatically escaped using the `\symbol` command under the T1 encoding. The argument to `\documentclass` defaults to `book`; of course this can be easily modified. Before that, however, let's give a slightly more practical example:

```
<html>
  <head>
    <title>a quite nice document</title>
  </head>
  <body>
    <p>Lorem % ipsum \ ... <em>$10,000</em></p>
    <p>dolor % sit \ amet ... <em>$42</em></p>
  </body>
</html>
```

In this HTML data, the main part of the document is wrapped with a `<body>` tag. That is, this time the `<body>` is the appropriate source for the LaTeX's `document` environment, instead of the `<html>` as in the previous example. So we change the previous rule like this:

```
(define-tag body (make-latex-env 'document))
```

We also need to handle the other tags in the `<body>`, namely `<p>` and `<em>`. Each `<p>` should be a paragraph in LaTeX. On the other hand, the LaTeX counterpart of `<em>` is `\emph{}`. These two types of mappings seem to be quite different. Nevertheless, when viewed as a recursive tree conversion, both mappings, and what is more almost all such mappings, can be regarded as a common routine:

1. Start a LaTeX piece with `\begin{foo}`, `\foo{`, or other strings.
2. Recursively process the node's children. If the only child is a simple string, then output the string with any necessary conversions.
3. End the LaTeX with the required `\end{foo}`, `}`, etc.

[1] The single quotation mark in `'document` is not a typo. It tells xml2tex that this is not a variable name, but a data item; specifically, a symbol in the Scheme programming language.

In fact, the second argument to `define-tag` is a rule which encodes this routine, and `make-latex-env` is the function that yields a common rule for generating a LaTeX environment. The rule is: “Put `\begin{...}` at the head; convert the children recursively with necessary escaping; put `\end{...}` at the tail.”

To explicitly define such a rule, we can use the `define-rule` declarative. `define-rule` takes three arguments, each corresponding to the above actions, in order: what to do at the beginning, what to do with the text nodes of the content, and what to do at the end.

For example, here is a possible rule for `<p>`:

```
(define-tag p   ; if the node is this name ...
  (define-rule
    "\n"        ; put this at the beginning ...
    trim        ; its text nodes should be ...
    "\\par\n")) ; put this at the ending.
```

where `trim` is a utility for trimming a string for use in LaTeX. Although we put the rule line by line in the above example, line breaks and other white space are generally immaterial. Text after a semicolon (`;`) is a comment.

A rule for `<em>` could be defined like

```
(define-tag em (define-rule "\\emph{" trim "}"))
```

or, equivalently,

```
(define-tag em (make-latex-cmd 'emph))
```

where `make-latex-cmd` is a utility to define a rule outputting the given LaTeX command.

The last rules we have to define are for `<head>` and `<title>`. Although we could use this meta-information to generate LaTeX content, here we will just ignore them. To make the converter discard an XML element, we can use a predefined rule `ignore`.

```
(define-tag head (ignore))
```

Consequently, we don't need to define a rule for the `<title>` tag, because the converter already knows that its parent tag is going to be discarded.

In short, to get a decent result from the above HTML data all we need are these four lines:

```
(define-tag head (ignore))
(define-tag body (make-latex-env 'document))
(define-tag p (define-rule "\n" trim "\\par\n")
(define-tag em (make-latex-cmd 'emph))
```

If we run xml2tex with those rules, we get valid LaTeX source for a book, because the default `\documentclass` is `book`. This is defined in a file `default.rule` in a way similar to the other rules below, and can be overridden with your own definition.

3 Details and tricks

As we have seen through the examples, xml2tex works as a domain-specific language (DSL) for defining maps between each XML tag and corresponding LaTeX syntax. When it comes to DSL, a programming language in the Lisp family fits well. xml2tex is written in Gauche [4], a dialect of the Scheme programming language. In addition to being a member of the Lisp family, Gauche has many text processing features and libraries, useful in defining more complex conversion rules.

To take advantage of a profound feature of the general programming language: the first and third arguments of `define-rule` need not be literal strings but can be Lisp functions without arguments. For example, a rule for the `<title>` tag could be defined like this:

```
(define-tag title (define-rule
    (lambda ()
     (cond
      ((eq? ($parent) 'chapter) "\\chapter{")
      ((eq? ($parent) 'sect1)   "\\section{")
      ((eq? ($parent) 'sect2)   "\\subsection{")
      (else (error "title" $parent)))))
    trim
    "}"))
```

In this definition, the first argument to `define-rule` is a function to select the appropriate LaTeX representation for the `<title>` tag based on its parent node. If the `<title>` tag belongs to `<chapter>`, the function returns the Scheme string `"\\chapter{"`. If `<sect2>`, then `"\\subsection{"`.

The other new feature used here is the keyword `$parent`. It expands to the name of the direct parent of that node. This is one of many "shortcuts" provided by xml2tex that can be used to retrieve the information from the XML tree. Table 1 lists these predefined convenience keywords.

Below is a naive example of using `$@`, which works as a function to retrieve the value of the specified attribute. We will also use the Gauche syntax `#‘"..."` for string interpolation. For example, we want `#‘"[width=,($@ 'width)]"` to expand to `[width=100mm]` if the `<img>` tag has the attribute `width="100mm"`.

```
(define-tag img (define-rule
    (list "\\begin{figure}\n"
          "\\includegraphics"
          #‘"[width=,($@ 'width)]"
          #‘"{,($@ 'src)}"))
    trim
    "\\end{figure}"))
```

Using these `$` keywords, we are able to define most rules declaratively and intuitively. In this regard, we can think of xml2tex more like a DOM (tree model) rather than SAX (event model). Indeed, xml2tex parses the entire XML tree in advance. This parsed tree has a form of SXML [5], a representation of the XML Infoset in the form of S-expressions. Even this bare SXML tree is available when defining a rule. It is sometimes necessary for elements which require transforming the original structure to define a reasonable mapping to LaTeX syntax. Tables are one such formidable challenge, as we'll see next.

Table 1: List of keywords defined by xml2tex

keyword	description
`$body`	Body of the element.
`$root`	Whole document tree.
`$parent`	Direct parent of the element.
`$parent? [name]`	If the element has parent with `[name]`.
`$childs`	List of all children of the element.
`$child [name]`	List of children with `[name]`.
`$following-siblings`	List of following-siblings.
`$siblings`	List of both following- and preceding-siblings.
`$@ [name]`	String value of the attribute `[name]`.
`$under? [list]`	If the element is a descendant of one of `[list]`.
`$ancestor-of? [list]`	If the element has any descendant in `[list]`?

4 Tables

To convert XML's logical structure of tables into LaTeX is a substantial problem. We think the root cause is probably the lack of a general model for tables in LaTeX.[2]

Let us consider the conversion rule for typical HTML tables. If we use `tabular` environment as the basic LaTeX construct for HTML tables, then the first attempt might be:

```
;; this doesn't work!
(define-tag table (define-rule
    "\\begin{tabular}\n"
    trim
    "\\end{tabular}"))

; put "\\" after each lines of table.
(define-tag tr (define-rule "" trim " \\\\"))

; put "&" after each cells in line.
(define-tag td (define-rule "" trim " &"))
```

[2] In contrast, ConTeXt has a standard model for tables and thus it's easier to define mappings from XML tables [8].

Unfortunately, this does not work. First, the tabular environment requires an argument explicitly specifying the appearance of each column. To determine this information from the given HTML table, we have to look through the entire table contents in advance. Second, we don't want the following '&' for the last cell of each line. Third, we should be able to change the width and color of each cell, as well as to span columns or rows. This information could be found in the attributes of <td>.

What we need is a way to transform the raw SXML tree before applying the corresponding conversion rules. To do that, we pass a procedure to define-rule using the :pre keyword. Below is a (relatively) simple example to decide the column specs for the tabular without any additional information except the table itself.

```
(use xmltex.latextable)
(define-tag table (define-rule
    #`"\\begin{tabular}{|,($@ 'colspec)|}\n"
    trim
    "\\end{tabular}\n"
    :pre calc-colspec))

(define (calc-colspec body root)
  (sxml:set-attr
   body
   (list
    'colspec
    (make-colspec
     (map
      (node-closure
       (ntype-names?? '(td th)))
      ((node-closure
        (ntype-names?? '(tr)))
       body))))))
```

The make-colspec function used in calc-colspec is one of the helper functions provided through an xml2tex package called xmltex.latextable, loaded at the beginning. With these helper functions and some understanding of Scheme and SXML, we have defined a conversion rule for a reasonable subset of HTML tables with colspan and rowspan. You can find the complete code in xml2tex's repository [9].

5 Conclusion

Like it or not, more and more documents are created and stored in XML. Books which one can buy are no exception. Considering the changing circumstances regarding e-books and the Web, nearly any book may well be created in one of the XML-based formats. If you were to use a desktop publishing applications, you could go with some very nice WYSIWYG environments and not be bothered with the ill-reputed syntax of XML. However, of course, we must prefer using TEX to such GUI software. This means, ultimately, writing a converter from XML-based formats to a TEX-flavored document.

We hope that xml2tex can help in this scenario. It works as a framework for using XML as a source for LATEX. All that's required is giving xml2tex a set of declarative mappings from each XML tag to an appropriate LATEX style. Aided by Scheme and SXML, you can even write a converter for a fairly complex XML document as needed.

To date, we have created dozens of commercial books using xml2tex while maintaining the manuscripts in a variety of XML-based formats.

References

[1] David Carlisle, "xmltex: A non-validating (and not 100% conforming) namespace aware XML parser implemented in TEX". http://tug.org/TUGboat/tb21-3/tb68carl.pdf

[2] Ramon Casellas and James Devenish, "Welcome to the DB2LATEX XSL Stylesheets". http://db2latex.sourceforge.net/

[3] ConTEXt Garden, "XML — ConTEXt wiki". http://wiki.contextgarden.net/XML

[4] Shiro Kawai, "Gauche — A Scheme Implementation". http://practical-scheme.net/gauche/

[5] Oleg Kiselyov, "SXML". http://okmij.org/ftp/Scheme/SXML.html

[6] Håkon Wium Lie, "PhD Thesis: Cascading Style Sheets". http://people.opera.com/howcome/2006/phd/

[7] Douglas Lovell, "TEXML: Typesetting XML with TEX". http://tug.org/TUGboat/tb20-3/tb64love.pdf

[8] Thomas A. Schmitz, "Getting Web Content and pdf-Output from One Source", http://dl.contextgarden.net/myway/tas/xhtml.pdf

[9] Keiichiro Shikano, "k16shikano/xml2tex". https://github.com/k16shikano/xml2tex

[10] World Wide Web Consortium, "Extensible Stylesheet Language (XSL) Version 1.1". http://www.w3.org/TR/xsl11/

[11] World Wide Web Consortium, "Cascading Style Sheets (CSS) Snapshot 2010". http://www.w3.org/TR/css-2010/

◇ Keiichiro Shikano
Tokyo, Japan
k16.shikano (at) gmail dot com
https://github.com/k16shikano/xml2tex