Processing “Computed” Texts

Abstract
This article is a comparison of methods that may be used to derive texts to be typeset by a word processor. By ‘derive’, we mean that such texts are extracted from a larger structure, which can be viewed as a database. The present standard for such a structure uses an XML-like format, and we give an overview of the available tools for this derivation task.

Keywords
Typesetting computed texts, \TeX, \LaTeX, \ConTeXt, X\TeX, Lua\TeX, XML, XSLT, character maps, XQuery, XSL-FO.

Introduction
Formats based on the \TeX typesetting engine—e.g., Plain \TeX [27], or \LaTeX [30], or Con\TeXt [8], or Lua\TeX [9]—are known as wonderful tools to get high-quality print outputs. Of course, they have been initially designed to typeset texts directly written by end-users. But other texts may be generated dynamically, in the sense that they result from some computation applied to more data, in particular, items belonging to databases. A very simple example is given by a bill computed by means of a spreadsheet program like Microsoft Excel: the master file is an .xls or .xlsx file—that is, all information about data is centralised into this file—but we may wish such a bill to be typeset nicely using a word processor comparable with \LaTeX\(^1\).

We personally experienced a more significant example: at the University of Franche-Comté, we manage the projects proposed to Computer Science students in several degrees, this curriculum being located at Besançon, in the East of France. That is, we collect projects’ proposals, control the assignment of student groups to projects. Then, at the semester’s end, we organise the projects’ oral defences, and rate students from information transmitted by jurors. During projects, information is transmitted to students and projects’ supervisors either on the Web, or by means of printed documents. Managing only a list of project specifications and enriching it progressively is insufficient: it is better for oral defences’ announcement to be shown with respect to defences’ chronological order, and this order is unknown when projects are proposed. Likewise, we may wish to present the grades received by students according to the decreasing order of these grades, or according to students’ alphabetical order, other criteria being possible, too. In these cases, we have to perform a sort operation before typesetting the result. These examples are not limiting: other operations related to ‘classical’ programming may be needed if we are only interested in a subset of the information concerning projects, for example, extracting projects proposed by companies, not by people working at our university.

It is well-known that \TeX’s language is not very suitable for tasks directly related to programming\(^2\). A better idea is to use a format suitable for information management, with interface tools serving several purposes. Database formats could be used, but presently, the indisputable standard to model such formats is XML\(^3\), providing a rich toolbox for this kind of task. In particular, this toolbox provides the XPath language [38], this language’s expressions allow parts of an XML document to be addressed precisely. But these tools related to XML have advantages and drawbacks: we are going thoroughly into these points in this article, which is a revised, updated, and extended version of [19]. Reading this article requires only basic knowledge about (\LaTeX and the tools related to XML.

A simple example
The examples given in the introduction are ‘real’ applications and belong to our framework: Microsoft Excel can generate XML files\(^4\), and we personally manage students’ projects by means of a master file using XML-like syntax. However, for sake of simplicity, we consider an easier example for the present article, pictured in Figure 1. This XML text—a file ds.xml—describes some items of a series of stories—Doc Savage—first published as pulps in the 1930’s, then republished as pocket books in the 1960’s\(^5\). As it can be noticed in the given example, the original publication order—for pulps—was not followed by the series of pocket books\(^6\). In addition, some stories were unpublished as pulps (e.g., The Red Spider) or retitled when published as pocket books (e.g., The Deadly Dwarf, previously entitled Repel), in which case, the pulp’s title is given as the pulp element’s contents\(^7\). More precisely, if a pulp element’s contents is empty, this means that the pulp’s title was the same as

\(^{1}\)\LaTeX\ is not a word processor, but only a typesetting engine.

\(^{2}\)\LaTeX is a language specifically designed for typesetting, which is why it is not suitable for programming.

\(^{3}\)XML is a standard for structuring data, and it is particularly well-suited for the Web.

\(^{4}\)Microsoft Excel can generate XML files, but these files are not very well-suited for information management.

\(^{5}\)Doc Savage was a series of pulps published in the 1930’s, then republished as pocket books in the 1960’s.

\(^{6}\)Some stories were unpublished as pulps, or retitled when published as pocket books.

\(^{7}\)Another important aspect of XML is that it allows for hierarchical data structures, where elements can be nested within other elements.
Figure 1. Master file using XML-like syntax.

the pocket book's. Fig. 2 gives the schema modelling our taxonomy\textsuperscript{8}, written using XML Schema [42]. Let us recall that this language provides a datatype library: for example, 'xsd:string' for strings, the prefix 'xsl:' allows us to get access to XML Schema's constructs\textsuperscript{9}.

Now we propose to search the information given in Fig. 1, extract the items published as pulps, sort them according to the publication order\textsuperscript{10}. The title given is the pulp's; if the corresponding pocket book has been retitled, a footnote must give the 'new' title. Of course, we wish the result to be typeset nicely, as \LaTeX\ or Con\TeX\ is able to do. To be more precise, a good solution processable by Plain \TeX\ could look like the source text given in Fig. 3. As mentioned above, a \TeX\-based solution:

\begin{verbatim}
\story-list
  \story{\title{...}\pulp{...}...}
\end{verbatim}

could use \TeX\ commands for dealing with the elements \story-list, \story, \title, \pulp, and \pocket-book, but would lead to complicated programming.

\begin{verbatim}
<xs:schema
 xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xs:element name="story-list">
   <xs:complexType>
     <xs:sequence>
       <xs:element name="story" maxOccurs="unbounded">
         <xs:complexType>
           <xs:sequence>
             <xs:element name="title" type="xsd:string"/>
             <xs:element name="pulp" minOccurs="0">
               <xs:complexType>
                 <xs:simpleContent>
                   <xs:extension base="xsd:string">
                     <xs:attribute ref="nb" use="required"/>
                   </xs:extension>
                 </xs:simpleContent>
               </xs:complexType>
             </xs:element>
             <xs:element name="pocket-book">
               <xs:complexType>
                 <xs:attribute ref="nb" use="required"/>
               </xs:complexType>
             </xs:element>
           </xs:sequence>
         </xs:complexType>
       </xs:element>
       <xs:attribute name="nb" type="xsd:positiveInteger"/>
     </xs:sequence>
   </xs:complexType>
 </xs:element>
</xs:schema>
\end{verbatim}

Figure 2. Our organisation expressed in XML Schema.

Using tools related to XML

XSLT producing \TeX\ sources

XSLT\textsuperscript{11} [41] is the language designed for transformations of XML texts. By 'transformations', we mean that we can build printed documents as well as online documents to be put on the Web from the source file. 'Simple' texts are possible, too. We also can perform some computation from data stored in the original XML file. Let us come to our example, the text of a stylesheet that may be used to get Fig. 3 from Fig. 1 is given in extenso in Fig. 4. This stylesheet takes as much advantage as possible of the features introduced by XPath's and XSLT's new version\textsuperscript{12} (2.0): for example, we have made precise the types of the used variables, by means of as attributes,
these types being provided by XML Schema’s library\textsuperscript{13}. Likewise, we have made precise the types of templates’ results. As it can be noticed in Fig. 4, these types belong to XML Schema’s namespace, whereas XSLT constructs are prefixed by the namespace associated with ‘xsl1:’.

Since we are interested in deriving texts processable by (La)\TeX, an important new feature introduced by XSLT 2.0 is the possible use of character maps \textsuperscript{41, § 20.1}. In particular, they allow \TeX’s special characters to be replaced by commands producing them:

\begin{verbatim}
  \#  \rightarrow  \#  \quad  \\rightarrow  $\backslash$s
\end{verbatim}

whenever they appear within a text node to be put by XSLT. More precisely, a single character can be replaced by a string, as shown by the character map \texttt{some-special-characters} given in Fig. 4. To distinguish an ‘actual’ backslash character, belonging to a string, and a command’s beginning, a solution is to use a character belonging to a private area of Unicode \textsuperscript{34} for the latter. For sake of readability, we define this fictive character by means of an entity\textsuperscript{15}—\texttt{start-command}—[32, p. 48–49]. Introducing this entity leads us to put a dummy DOCTYPE tag, since XSLT stylesheets do not refer to a DTD. In other words, specifying these additional characters is a ‘trick’, but that allows us to process strings extracted from the original XML file systematically. As shown in Fig. 4, the same technique can be used for opening and closing a group: we use fictive characters the character map transforms into braces. The same for a delimited fragment in (La)\TeX’s math mode. Another solution could be the direct generation of Unicode texts and the use of a Unicode-compliant \TeX-like engine\textsuperscript{16}, e.g. Xe\TeX \textsuperscript{26} or Lua\TeX \textsuperscript{9}.

As abovementioned, XSLT is not limited to texts’ generation, the \texttt{xsl:output} element’s \texttt{method} attribute may also be set to \texttt{html} or \texttt{xhtml}\textsuperscript{17} \textsuperscript{41, § 20}, in which case it allows Web pages to be generated. Likewise, this \texttt{method} attribute set to \texttt{xm1} means that XML texts are to be generated. Using these output methods provides a great advantage: since any XSLT stylesheets is an XML text, an XSLT processor checks that the final document is well-formed, in particular, opening and closing tags must be balanced. The generation of (La)\TeX texts lacks an analogous check: an XSLT processor cannot ensure that opening and closing braces are balanced, as in \TeX; likewise, when \LaTeX texts are generated, it is impossible to ensure that environments like:

\begin{verbatim}
\begin{document} . . . \end{document}
\end{verbatim}

are balanced. Since such errors are not detected statically, they just appear when generated texts are processed. Let us notice that such a check would be more difficult to apply to the texts generated for the \LaTeX format, because the opening and closing commands for \LaTeX’s environments are ‘\begin{…}’ and ‘\stop…’, e.g., the equivalent formulation for (1) in \LaTeX is:

\begin{verbatim}
\starttext \stop\text
\end{verbatim}

in \LaTeX. Concerning the delimiters of a command’s arguments, we can ensure that opening and closing braces—more precisely, the two character entities \texttt{start-group} and \texttt{end-group}, before their replacement by braces—are balanced by using only the XSLT function \texttt{ntg:make-group} to build a \TeX group from a sequence of strings. Let us remark that XSLT functions have been introduced in XSLT 2.0, so there is an additional reason to use this version. Another solution could be the use of an XML dialect whose architecture would reflect \TeX’s markup. From our point of view, that would complicate the process since an additional step—a translation from this dialect into ‘actual’ \TeX-like syntax—would be performed. In addition, some versions of such dialects have already been proposed—e.g., in [7, § A.1] for \LaTeX—but it seems to us that none is actually used.

\textbf{XSLT producing XSL-FO texts}

Since deriving XML texts by means of XSLT provides a better check level about syntax, an alternative idea is to get XSL-FO\textsuperscript{37} texts. Such texts use an XML-like syntax that aims to describe high-quality print outputs. As shown in [16], there is some similarity between \LaTeX and XSL-FO, the latter providing, of course, more systematic markup\textsuperscript{19}. This language is verbose, but it is not devoted to direct use: XSL-FO texts usually result from applying an XSLT stylesheet. The use of several namespaces—usually denoted by the prefixes ‘xsl1:’ and ‘xfo:’—clearly distinguish elements belonging to XSLT and XSL-FO.

This approach’s advantage is clear: generated texts are well-formed. However, XSL-FO lacks \textit{document classes}, as in \LaTeX. Some elements allow the description of \textit{page models}, but end-users are entirely responsible for this definition. XSL-FO provides much expressive power about placement of \textit{blocks}\textsuperscript{20}, but is very basic on other points. For example, let us consider footnotes, end-users are responsible of choosing each footnote.
<!DOCTYPE stylesheet [<!ENTITY start-command "&#xE000;"> <!ENTITY start-group "&#xE001;"> 
<!ENTITY end-group "&#xE002;">]>

<xsl:stylesheet version="2.0" id="pulps-plus" xmlns:maps="http://www.ntg.nl"
xmns:xsd="http://www.w3.org/2001/XMLSchema"
xmns:xsl="http://www.w3.org/1999/XSL/Transform"
extension-element-prefixes="xsd">
  <xsl:output method="text" encoding="ISO-8859-1" use-character-maps="some-special-characters"/>
  <xsl:strip-space elements="*"/>
  <xsl:character-map name="some-special-characters">
    <xsl:output-character character="#" string="\#"/>
    <xsl:output-character character="%" string="\%"/>
    <xsl:output-character character="\$" string="\$"/>
    <xsl:output-character character="&amp;" string="\&"/> <!-- & Plain \TeX’s commands -->
    <xsl:output-character character="\" string="\$\{"/> <!-- \{ and \} are only -->
    <xsl:output-character character="\" string="\}$"/> <!-- usable in math mode -->
    <xsl:output-character character="\" string="\{(char'7E)\}"/> <!-- Using hexadecimal code -->
    <xsl:output-character character="&end-group;" string="\}="/>
    <xsl:output-character character="&end-group;" string="{"/>
  </xsl:character-map>
  
  <xsl:variable name="eol" select="'&#xA;'" as="xsd:string"/>
  <xsl:template match="story-list" as="xsd:string">
    <xsl:variable name="pulps" as="xsd:string+">
      <xsl:apply-templates select="story[pulp]">
        <xsl:sort select="xsd:integer(pulp/@nb)"/> <!-- Numerical sort -->
      </xsl:apply-templates>
    </xsl:variable>
    <xsl:value-of select="$pulps,$eol,'&start-command;end',$eol" separator=""/>
  </xsl:template>
  
  <xsl:template match="story" as="xsd:string">
    <xsl:variable name="pulp-0" select="pulp" as="element(pulp)"/>>
    <xsl:value-of select="$pulp-0/@nb" as="xsd:string"/>
    <xsl:variable name="pulp-title-0" select="data(pulp-0)" as="xsd:string"/>
    <xsl:variable name="title-processed" as="xsd:string">
      <xsl:apply-templates select="title"/>
    </xsl:variable>
    <xsl:value-of select=""&start-command;item",maps:mk-group($pulp-nb-0-string)," ",
      if ($pulp-title-0) then
        $pulp-title-0,&start-command;footnote*", maps:mk-group("Book\&apos;s title of pulp #",$pulp-nb-0-string,"",
        $title-processed)) else
        $title-processed,
      $eol" separator=""/>
  </xsl:template>
  
  <xsl:template match="title" as="xsd:string">\xsl:apply-templates</xsl:template>
  <xsl:function name="maps:mk-group" as="xsd:string">\xsl:param name="string-seq" as="xsd:string"></xsl:sequence select="&start-group;","string-seq,&end-group;"/>
</xsl:function>
</xsl:stylesheet>

Figure 4. Getting a source text for Plain \TeX by means of an XSLT stylesheet.
Figure 5. How to put a footnote in XSL-FO (see the equivalent Plain \TeX source text in Fig. 3).

mark. Fig. 5 provides the result of applying an XSLT stylesheet providing an XSL-FO text for our example. The first child of the fo:footnote element gives the footnote reference, the second child is the actual footnote's contents [37, § 6.12.3]. This fo:footnote element seems to be low-level in comparison with \LaTeX's \texttt{\footnote} command\textsuperscript{21}. Of course, if you want footnotes to be numbered automatically, XSLT addresses this problem. However, another point seems to us to be more debatable: end-users are responsible for putting a leader separating footnotes from the text body\textsuperscript{22} (we show how to proceed in Fig. 5). So some footnotes may be preceded by a leader, some may not. This point may seem anecdotal, but for \LaTeX users, some features of XSL-FO can be viewed as low-level and be difficult to handle since they are already programmed in \LaTeX classes.

Last but not least, most current XSL-FO processors do not implement the whole of this language, even if they can successfully process most of XSL-FO texts used in practice\textsuperscript{23}. So some features may be unusable, whereas an equivalent construct will work in (\LaTeX). XSL-FO has been designed to deal with the whole of Unicode, so it shows how the Unicode bidirectional algorithm [35] is put into action [18], but this point may also be observed with \LaTeX.\textsuperscript{24}

**XQuery producing \TeX sources**

XQuery [40] is a query language for data stored in XML form, as SQL\textsuperscript{24} does for relational data bases\textsuperscript{25}. XQuery can be used to search documents and arrange the result, as an XML structure or a simple text (possibly suitable for a \TeX-like engine). An XQuery program processing our example in order to get Fig. 3's text is given in Fig. 6. Like XSLT 2.0, XQuery uses XPath 2.0 expressions and the datatype library provided by XML Schema. As we did in XSLT, we systematically put type declarations using the as keyword, for sake of clarity and for taking as much advantage as possible of XQuery's type-checker. Such programs, using FLWOR\textsuperscript{26} expressions, are more compact than equivalent ones in XSLT.

However, XQuery is suitable only for generating simple texts: advanced features like character maps in XSLT are provided by some XQuery processors, but are not portable. You have to use the \texttt{replace} function [39, § 7.6.3] to deal with \TeX's special characters:

\begin{verbatim}
replace($s, "(#|%)", "\$1")
\end{verbatim}

substitutes each occurrence of ‘#’ (resp. ‘%’) by ‘\#’


(\% or `\%`) within the string \$s. Let us come back to implementation-dependent features, a simple example is given in Fig. 6: we declare that the result is not an XML text by a non-portable option, \texttt{saxon:output}. Of course, if XQuery is used to generate XML texts, they are well-formed, but no analogous check can be done about texts generated for a \TeX-like engine. In other words, XQuery has the same drawback as XSLT.

\section*{A curiosity: DSSSL}
DSSSL\textsuperscript{28} [21] was initially designed as the stylesheet language for displaying SGML\textsuperscript{29} texts. DSSSL includes a core expression language that is a side-effect free subset of the Scheme programming language [25]. XML being a subset of SGML, stylesheets written using DSSSL can be applied to XML texts. DSSSL is rarely used now, but the example we cite illustrates how a functional programming language can be suitable for our requirements. Fig. 7 gives a stylesheet that produces a result equivalent to Fig. 3. In fact, end-users do not write \LaTeX\ commands when they develop a stylesheet, the \texttt{jade}\textsuperscript{30} program can generate \TeX-like texts:\textsuperscript{31}:

\begin{verbatim}
jade -d pulps.dsl -t tex ds.sgml
\end{verbatim}

-we have to specify a predefined \texttt{backend}, here '\texttt{tex}'-

the typesetting engine usable to process \texttt{jade}'s results being \texttt{JadeTeX} [7, § 7.5.2]. Deriving texts directly processable by \LaTeX or \texttt{ConTeXt} is impossible.

As shown in Fig. 7, processing a \texttt{name} element uses pattern-matching:

\begin{verbatim}
(element name E)
\end{verbatim}

the \texttt{E} expression consists of assembling \texttt{literals} by means of the make \texttt{form}, using types predefined in DSSSL: \texttt{paragraph}, \texttt{sequence}, ... The generic type of such results is called \texttt{sosofo} w.r.t. DSSSL's terminology.

\section*{Enriched \TeX engines}
If we go back to programs based on \TeX-like typesetting engines, there are two other possible methods, based on \TeX-engines 'enriched' by using a 'more classical' programming language. In both cases, XML texts are pre-processed by procedures belonging to a programming language, and the result is sent to a \TeX-engine. Of course, such a \textit{modus operandi} is suitable only if we want to generate (\LaTeX) texts, it would be of little interest to get XML texts or pages written using (X)HTML\textsuperscript{33}.

\texttt{PyTeX} [6] is written using Python [28] and uses \TeX as a daemon. Getting the components of a 'computed'
The image contains a code snippet of a DSSSL stylesheet. This stylesheet is designed to generate a text to be printed. The code is structured using DSSSL (Document Style SheeT Language), which is a language for describing document styles. The stylesheet includes specifications for formatting elements such as paragraphs, titles, and footnotes. It also includes functions like `string->number`, `list-stable-sort`, and operations for handling collections of elements. The stylesheet appears to be well-documented with comments explaining various parts of the code. The primary language used in the code is XML. The figure caption reads: "Figure 7. DSSSL stylesheet generating a text to be printed."
text is left to the Python functions dealing with XML texts and successive results are sent to \TeX, in turn.

Lua\TeX{} [9] is able to call functions written using Lua [20]. On another point, this \TeX-engine can process texts using XML-like syntax, as shown in [10]. Fig. 8 gives an implementation of our example in Con\TeXt MkIV: it uses Lua to define an interface with sorting functions, the other functionalities being put into action using \TeX-like commands. As in Con\TeXt, the layout is controlled by set-up commands:

\begin{verbatim}
\enablemode[ds:pulp]
\startxmlsetups xml:ds:base
  \xmlsetsetup{#1}{%
    story-list|story|title|pulp|pocket-book}{%
      xml:ds:*}
\stopxmlsetups
\xmlregisterdocumentsetup{ds}{xml:ds:base}
\startxmlsetups xml:story-list
  \xmlresetsorter{story}
  \xmlfilter{#1}{%
    /story/command(xml:story-list:getkeys)}
  \subject{sortkeys}
  \xmlsortentries{story}
  \xmlflushsorter{story}{xml:story-list:flush}
\stopxmlsetups
\startxmlsetups xml:story-list:getkeys
  \xmladdsortentry{story}{#1}{%}
  \xmlattribute{#1}{/pulp}{nb}}
\stopxmlsetups
\startxmlsetups xml:story-list:flush
  \startitemize\xmlfirst{#1}{.}\stopitemize
\stopxmlsetups
\startxmlsetups xml:ds:story
  \sym{\xmlfirst{#1}{pulp}}
  \doifmodeelse{ds:pulp}{\xmlatt{#1}{nb}}{%}
  \xmlflush{#1}}
\stopxmlsetups
\startxmlsetups xml:ds:title
  \xmlflush{#1}
\stopxmlsetups
\startxmlsetups xml:ds:pulp
  \doifmodeelse{ds:pulp}{\xmlatt{#1}{nb}}{%}
  \xmlflush{#1}}
\stopxmlsetups
\starttext
  \xmlprocessfile{ds}{ds.xml}{}
\stoptext
\end{verbatim}

—for example, title elements' contents are just displayed, processing pulp elements displays the number or the title, depending on a mode—then our XML file is processed ‘atomically’, by means of the \xmlprocessfile command. This approach is promising, but let us recall that Lua\TeX{} and Con\TeXt MkIV have not yet reached stable state: that is planned for the year 2012. Another important drawback: as shown by the examples using the \xmlfilter command in Fig. 8, this command uses path expressions, very close to XPath expressions, but not identical. For example, command—used to connect a selected item to the set-up command that processes it—obviously does not belong to XPath. On the contrary, some XPath expressions are not recognised, even if ‘simple’ paths are processed. Some tricks may be used as workarounds, but we personally think that complete compatibility with XPath should be attained.

**Conclusion**

If we sum up the approaches shown throughout this article, those that seem suitable are XQuery for simple examples, XSLT for more ambitious ones, provided that Version 2.0 is used. The use of Lua\TeX{} could be interesting in a near future, too.

However, we think that there are two directions that should be explored. The first would be a modern implementation of XSL-FO using a \TeX-like typesetting engine. Such an implementation has begun: Passive \TeX{} [4], but this project is presently stalled. We think that processing XSL-FO could re-use the experience accumulated by \TeX{} developers, even if syntaxes are very different. The second direction would be the definition and implementation of an output mode of XSLT suitable for \TeX; some additional services could be performed: for example, checking that braces and environments are balanced. Such an output mode already exists in nbst, the language of bibliography styles close to XSLT and used by MfBn\TeX{} [11], but it concerns only the way to process \LaTeX{}'s special characters. If the method attribute of the nbst:output element is set to text, the result of:

\begin{verbatim}
<nbst:text>#60 The Maji</nbst:text>
\end{verbatim}

is ‘#60 The Maji’. If this attribute is set to \LaTeX, the result is ‘#60 The Maji’.

**Acknowledgements**

I wish to thank Hans Hagen who greatly helped me debug and improve Lua\TeX{} source texts. Thanks also to Karl Berry, who clarified some terminology notions.
Last but not least, thanks to Taco Hoekwater for his patience when he was waiting for this final version.

References


1. Let us mention that [2] shows—in French—how to use LaTeX to put spreadsheets’ functionalities into action.

2. As an example of using \TeX’s language for programming purposes, readers interested in putting a sort procedure into action can refer to [33]: this modus operandi may be viewed as a worthwhile exercise, but is unusable in practice, especially when it is not trivial to obtain sort keys from items to be sorted.

3. eXtensible Markup Language. Readers interested in a general introductory book to this formalism can refer to [32].

4. The XML format used by Microsoft Excel is OOXML (Office Open XML).

5. All the source texts mentioned throughout [19] and this article—including new versions realised for this present article, in which case the corresponding file names are suffixed with ‘-plus’—can be downloaded in extenso from the Web page http://lifc.univ-fcomte.fr/home/~jmhufflen/texts/guit-2009/.

6. If you are interested in the story of Doc Savage series and its successive editions, you can find more information in [43].

7. When several titles have been used, such a story is more commonly known under the pocket book’s title, because pocket books are easier to get than pulps, which are very rare. That is why our title elements always refer to pocket books', the contents of pocket-book elements being always empty.

8. Schemas allow specifiers to define document types, which can be viewed as some taxonomy common to a family of XML texts. There exist several schema languages, and the Web page abovementioned gives several versions, using a DTD (Document Type Definition) [32, Ch. 5], XML Schema [42], Relax NG (New Generation) [5], and Schematron [22]. A discussed comparison among these schema languages can be found in [36].

9. Readers interested in more details about XML namespaces can consult [32, pp. 41-45].

10. This example seems to us to be pertinent, because there is no order ‘better’ than others: the original order is based on pulps, but—as mentioned above—some stories are unpublished as pulps, whereas sorting stories according to pocket books’ order allows us to sort all the stories, but this is not really chronological.

11. eXtensible Stylesheet Language Transformations. Introductions to this language have been given in some Bacho\TeX conferences, held in Poland: [12, 13, 15].

12. A study of XPath 2.0’s and XSLT 2.0’s new features, in comparison with XPath 1.0 and XSLT 1.0, can be found in [15].

13. In addition, let us mention that when an XML text is processed by XSLT 2.0, the information put into a DTD or an XML Schema text can be exploited [23, p. 58]. That is not true about other schema languages.

14. In text mode, \LaTeX2e’s modern versions provide the \textbackslash{backslash} command [30, Table 7.33].

15. If we name this character by introducing a variable by means of an xml:variable element—as we did in Fig. 4 for the end-of-line character—we cannot use this variable’s value within the character attribute of the xml:output-character element.

16. Engines are not formats: a format is a set of pre-loaded definitions based on primitives of a \TeX-like engine, whereas an engine is \TeX or is derived from \TeX by adding or redefining...
some primitives. Plain \TeX and \LaTeX are formats, \XeLaTeX and LuaLaTeX are engines.

17. (eXtensible) HyperText Markup Language. XHTML is a reformulation of HTML—the original language of Web pages—using XML conventions. [31] is a good introduction to these languages.

18. eXtensible Stylesheet Language—Formatting Objects.

19. [16] is written in French. If you would like a similar text in English, [14] is an abridged version.

20. Roughly speaking, a block in XSL-FO is analogous to a \texttt{minipage} in \LaTeX [16, § 1.2].

21. XSL-FO’s footnotes can be compared with Plain \TeX’s \texttt{\footnote} command, as shown in Fig. 3.

22. That is done in standard \LaTeX class, but not universal: as an example related to \TeX’s community, the arstexnica class, used for articles of the \\texttt{Ars\ Te\ xnica} journal—published by \texttt{g\textunderscore it} (Gruppo Utilizzatori Italiani di \TeX), the Italian-speaking \TeX users group—does not put leaders just above footnotes.

23. We personally use Apache FOP (Formatting Objects Processor) [1]:

\begin{verbatim}
    fop pulps-result.fo pulps-result.pdf
\end{verbatim}

generates a PDF (Portable Document Format) file from a source text in XSL-FO.

24. Structured Query Language. A good introductory book about it is [29].

25. A short introduction to XQuery is given in [17].

26. ‘For, Let, Where, Order by, Return’, the keywords used throughout such expressions.

27. The XQuery processor we have used for this example is Saxon [24]. Fig. 6’s text can be processed by:

\begin{verbatim}
    java net.sf.saxon.Query pulps-plus.xq \
    -filename="ds.xml"
\end{verbatim}

We also use Saxon as an XSLT 2.0 processor and the stylesheet of Fig. 4 can be processed by:

\begin{verbatim}
    java net.sf.saxon.Transform -s:ds.xml \
    -xsl:puls-plus.xsl
\end{verbatim}


29. Standard Generalised Markup Language. Now it is only of a historical interest. Readers interested in this metalanguage can refer to [3].

30. James Clark’s Awesome DSSSL Engine.

31. We use a different file and a different name for the root element (\texttt{story-list-sgml}) because of syntactic reasons: empty tags’ syntax was different in SGML [3, p. 259].

32. Specification Of a Sequence Of Flow Objects.

33. Unless a converter to (X)HTML is used, of course.

34. Besides, it is well-known that \TeX recognises only its own formats, which complicates cooperation between \TeX and other programs.

35. New Bibliography STyles.

36. MultiLingual Bio\TeX.

Jean-Michel Hufflen
LIFC (EA CNRS 4157),
University of Franche-Comté, 16, route de Gray,
25030 Besançon Cedex, France