Implementing a variant of XSLT in Scheme

Jean-Michel HUFFLEN

LIFC (FRE CNRS 2661), University of Franche-Comté
16, route de Gray. 25030 BESANÇON CEDEX. FRANCE.

Abstract. We describe a variant of the XSLT language, usable for designing bibliography styles according to a multilingual approach. Then we show how we implement it efficiently using Scheme. In particular, that allows us to give an operational semantics of this variant of XSLT.
Keywords XSLT, Scheme, bibliography styles, compiling templates into functions.

0 Introduction

XML[1] has succeeded as a format for data interchange. XSLT[2], the language of transformations used for XML texts, is now widely used. Indeed it is viewed as very high-level language, accessible to non-specialists in Computer Science. That is why we have chosen this kind of language within our implementation of a *bibliography processor*. Let us recall that such a program searches bibliographical database files for some *citation keys*, arranges the references in a 'Bibliography' section put at the end of a printed document. That is, this section is an output of the bibliography processor and becomes an input for the word processor. So users do not have to build themselves this section when they are writing a document, they just use citation keys throughout the document's body. A well-known example of such a bibliography processor is BIBTEX [12], used mainly with the LATEX word processor [8].

Our bibliography processor—MlBIBTEX, for '**M**ulti**L**ingual BIBTEX'—is a reimplementation of BIBTEX with particular focus on multilingual features. It is implemented using the Scheme programming language [4] and a general overview of this implementation is given in [2], where we also explain why we chose Scheme for this task. In this article, we focus on the language used for bibliography styles. Section 1 describes it as a variant of XSLT. Section 2 explains how the kernel of this language is implemented in Scheme. In this section, we attempt to be formal in order to show that we follow a precise approach about the types we use. Section 3 discusses some future directions for our program. Reading this article requires basic knowledge of XML[3], good practice of XSLT [16] and Scheme.

---

1 e**X**tensible **M**arkup **L**anguage.
2 e**X**tensible **S**tylesheet **L**anguage **T**ransformations.
3 Readers interested in an introductory book can refer to [13].

```
<nbst:template name="format.date" language="magyar">
  <nbst:apply-templates select="year"/>
  <nbst:if test="month">
    <nbst:text> </nbst:text>      <!--  Putting a space character.   -->
    <nbst:apply-templates select="month"/>
  <nbst:if>
</nbst:template>

<nbst:template name="format.date">
  <nbst:if test="month">
    <nbst:apply-templates select="month"/>
    <nbst:text> </nbst:text>
  <nbst:if>
  <nbst:apply-templates select="year"/>
</nbst:template>
```

Fig. 1. Templates in the nbst language.

1 XSLT vs nbst

Bibliography styles rule the layout of a 'Bibliography' section. For example, some titles of works are to be written using italicised characters, others are to be enclosed between quotation marks (' "..." ' in American English). Bibliography styles are diverse: the first name of an author may be written *in extenso* or abbreviated to initials, it may be put before or after its last name... They are influenced by cultural background, and depend on languages: for example, the quotation marks look like ` « ... » ' in French.

The nbst[4] language is described in [1]. In the following, we just emphasise the difference with XSLT. An nbst template may be given a `language` attribute, as shown in the examples of Figure 1. A template with such an attribute has higher priority than a template without it. If we consider the two templates of Figure 1, we can see intuitively that the first template puts down a date within a document in Hungarian, that is, the month comes after the year, whereas the second template is a *default* template, usable for languages such as English, French, German, ... where the month comes at first.

There are two ways to handle the information about languages:

reference-dependent approach each reference of the 'Bibliography' section of a document is expressed using the language of the corresponding entry in a bibliographical data base file: for example, the month name of a reference to a book written in English (resp. French, German, ...) is given in English (resp. French, German, ...); so the language of a reference—what is put within a 'References' section—is the language of the corresponding entry—what is included in a bibliographical database—;

[4] **N**ew **B**ibliography **ST**yles.

```
<mlbiblio>
  ...
  <book id="harrison1984" language="english">
    <author>
      <name>
        <personname>
          <first>Harry</first>
          <last>Harrison</last>
        </personname>
      </name>
    </author>
    <title>West of Eden</title>
    <publisher>Grafton Books</publisher>
    <year>1984</year>
    <month><aug/></month>
    <address>London</address>
  </book>
  ...
</mlbiblio>
```

Fig. 2. Example of bibliographical entry.

document-dependent all the references are expressed using the document's language, as far as possible: for example, all the dates are expressed using the document's language. According to this approach, the language of a reference may be different from the language of the corresponding entry: for example, a book written in English belonging to the 'References' section of a document written in German.

Inside a bibliographical entry, some information may be expressed by another language than the entry's. For example, the following title for a book written in English uses French words:

`TITLE = {[Danse macabre] : french}`

what is expressed by the notation '`[...] : ...`' in MlBibTeX [1]. Such information allows a word processor to hyphenate these words correctly, if need be. This situation is more frequent with the document-dependent approach: for example, when we process the title of a book written in English within a 'Bibliography' section for a document written in German, as mentioned above.

As shown by the title given above as an example, MlBibTeX deals with an extended syntax of the format used by BibTeX [12], our parser building an XML tree. An example is given in Figure 2, the `language` attribute of such an XML tree for a bibliographical entry defaulting to `english`.

According to a document-dependent approach, building all the references from an XML tree rooted by the `mlbiblio` element can be done as follows—the `use-language` attribute being an extension of `nbst`—:

```
(*top*
 (mlbiblio
  ...
  (book (@ (id "harrison1984") (language "english"))
        (author (name (personname (first "Harry") (last "Harrison"))))
        (title "West of Eden")
        (publisher "Grafton Books")
        (year "1984")
        (month (aug))
        (address "London"))
 ...))
```

Fig. 3. Representation of the tree given in Figure 2 using the SXML format.

```
<nbst:apply-templates use-language="$document-language"/>
```

the document's language being deduced from the multilingual packages loaded when LaTeX processes the document.

According to a reference-dependent approach, each son of an `mlbiblio` element uses its own language information, what is specified by:

```
<nbst:apply-templates use-language="*self*"/>
```

2 Implementation of the nbst language

MlBibTeX uses the SXML[5], described in [7], for XML trees. SXML is a concrete representation of the XML infoset in the form of S-expressions. For example, Figure 3 gives the SXML representation of the XML tree of Figure 2. SXML is the basis of a toolboox including SSAX[6]—a SAX[7] parser[8] [6]—, SXPath[9] [9]—that allows users to address parts of an SXML document by means of paths defined within the XPath language [15]. SXML is used within an implementation of XSLT, STX[10], described in [10]. Now this implementation is not complete, but it allows fragments using XSLT and Scheme to be mixed.

2.1 Rough implementation

STX compiles each template of an XSLT stylesheet into a Scheme function being the form:

[5] Scheme implementation of XML.
[6] Scheme implementation of SAX.
[7] **S**imple API (Application Programming Interface) for XML.
[8] Used within MlBibTeX to parse nbst texts.
[9] Scheme implementation of XPath.
[10] Scheme-enabled **T**ransformation for XML data.

```
(lambda (cur-tree templates root envt) ...)
```

where `cur-tree` is the current SXML tree, that is, the SXML tree we are processing, `templates` groups all the templates of the stylesheet (in STX, they are grouped into a list), `root` the root of the SXML tree, and `envt` manages the bindings of the variables defined in STX styleshets. To extend this process to nbst, the compiled form of an nbst template is a function being the form:

```
(lambda (doc-lg cur-lg mode cur-tree templates root envt) ...)
```

where the additional arguments are `doc-lg` for the document's language, `cur-lg` for the current tree's language, and `mode` for the `mode` attribute used by the `nbst:apply-templates` and `nbst:template`, as in XSLT[11] [16, § 5.7]. If nbst works according to a reference-dependent approach, `doc-lg` is bound to `#f`. If there is no `mode` attribute, `mode` is bound to `#f`. If fact, the compiled form of an nbst template can be viewed as a function of type:

$$\begin{array}{l} \mathit{LANGUAGE}^{\#} \times \mathit{LANGUAGE} \times \mathit{MODE}^{\#} \times \mathit{TREE} \times \\ \mathit{TEMPLATES} \times \mathit{TREE} \times \mathit{ENVIRONMENT} \rightarrow \\ \qquad \mathit{STRING} \end{array} \tag{1}$$

where '$T^{\#}$' is for a disjoint union of the T type and the false value (`#f` in Scheme). It is well-known that this operation is easy to put into action within Lisp dialects, provided that all the values of the T type are viewed as *true*. The types used are:

– *LANGUAGE* (resp. *MODE*): an enumerated type for language information (resp. mode information),
– *TREE* for SXML trees,
– *TEMPLATES* for grouping nbst templates,
– *ENVIRONMENT* for the management of the variables introduced in nbst stylesheets.

As in SXLT, such a function results in a string (*STRING* type).

As in STX, the first implementation of nbst looked into a list of templates and found the first template matching the current tree. As in STX, priorities were not managed and all the templates are stored into a list. That is irrelevant for a prototype, but unacceptable for a program that aims to become public, as a successor of BibTeX. In addition, let us notice that in 'real' bibliography styles:

– there is a large number of templates in a 'actual' bibliography style, that is, a large number of potential values for the `match` attribute of an `nbst:template` element, so it is inefficient to search a list as many times as a template is invoked;

[11] This feature being not implemented yet in STX.

– cases may be handled by using priority among rules [16, § 5.5]: for example, if the title of a book has to be displayed differently in comparison with other titles, this can be done by specifying two templates with `book/title` and `title` as values for `match` attributes; in practice, this kind of situation is frequent and it would be very inefficient to reach the end of a template list in order to know if a 'better' template matches the current tree.

So MlBibTEX's public version will include the compilation of nbst described below.

2.2 Compilation of XPath expressions

When an nbst program is processed, the `match` attribute[12] of each template is split into the name of the subtree—element or attribute—matched and additional constraints. Let us recall that the values of this `match` attribute form a subset of XPath expressions [15]. More precisely, they are *steps*, separated by the '`/`' sign, each step exploring the childs or attribute of a node. The '`//`' separator is also allowed, in which case all the descendants of a node are explored. Here are some examples:

```
title       ⟹ title —
book/title  ⟹ title — [name(parent()) = "book"]
book/title[../year = "2006"] ⟹
        title — [name(parent()) = "book"] [../year = "2006"]
```

More formally, let `t` an (S)XML tree whose current node is an element named `element-name`. Let us assume that we are writing a function checking that such a subtree is matched by the XPath expression:

$$\texttt{/?step}_1\texttt{/}\ldots\texttt{/step}_n\texttt{/element-name[expr}_1\texttt{]}\ldots\texttt{[expr}_p\texttt{]} \qquad (2)$$

we explain the meaning of our symbols in Fig. 4. We split this expression into ts element's name and a sequence of boolean expressions that are the compiled form of the additional constraints given in (2). These boolean expressions can be grouped into a function whose formal argument is texttt. This split operation is performed as follows:

$\textit{split}(\texttt{/?step}_1\texttt{/}\ldots\texttt{/step}_n\texttt{/element-name[expr}_1\texttt{]}\ldots\texttt{[expr}_p\texttt{]})\ \textit{for}\ \texttt{t} \longrightarrow$
(`element-name`,
$\textit{compile}(\texttt{/?step}_1\texttt{/}\ldots\texttt{/step}_n)\ \textit{for}\ \texttt{parent(t)}$;
$\textit{compile-boolean-expr}(\texttt{expr}_1)\ \textit{for}\ \texttt{t}$;
...
$\textit{compile-boolean-expr}(\texttt{expr}_p)\ \textit{for}\ \texttt{t}$)

This operation can be put into action within the SXML representation of XML trees, but is not limited to it[13]. It just requires functions allowing us to move throughout such a tree, e.g., `parent`, that returns the parent of a tree. Figure 4 gives the broad outlines of compiling XPath expressions.

[12] Of course, the templates with a `name` attribute are processed differently.
[13] ... what we suggest by '(S)XML'.

compile() *for* `t` ⟶ ∅

compile(`/?step`$_1$`/.../step`$_n$`/L[expr`$_1$`]...[expr`$_p$`]`) *for* `t` ⟶
  compile-filter(`L`) *for* `t` ;
  compile(`/?step`$_1$`/.../step`$_n$) *for* `parent(t)` ;
  compile-boolean-expr(`expr`$_1$) *for* `t` ;
  ...
  compile-boolean-expr(`expr`$_p$) *for* `t`

compile(`/?step`$_1$`/.../step`$_n$`//L[expr`$_1$`]...[expr`$_p$`]`) *for* `t` ⟶
  compile-filter(`L`) *for* `t` ;
  compile(`/?step`$_1$`/.../step`$_n$) *for*
    `t`$_0$ ∈ `ascendant-or-self(t)` ;
  compile-boolean-expr(`expr`$_1$) *for* `t` ;
  ...
  compile-boolean-expr(`expr`$_p$) *for* `t`

compile(`step`) *for* `t` ⟶ *compile-step*(`step`) *for* `t`

compile(`/step`) *for* `t` ⟶ *compile-filter*(`step`) *for* `t` ;
  `at-root(t)`

compile-step(`L[expr`$_1$`]...[expr`$_p$`]`) *for* `t` ⟶
  compile-filter(`L`) *for* `t` ;
  compile-boolean-expr(`expr`$_1$) *for* `t` ;
  ...
  compile-boolean-expr(`expr`$_p$) *for* `t`

compile-filter(`element-name`) *for* `t` ⟶ `name(t) = element-name`

compile-filter(`f()`) *for* `t` ⟶ `boolify(`$\overline{\texttt{f}}$`(t))`

compile-boolean-expr(`expr`) *for* `t` ⟶ `boolify`(*compile-expr*(`expr`) *for* `t`

compile-expr(`op(expr`$_1$`,expr`$_2$`)`) *for* `t` ⟶
  $\overline{\texttt{op}}$(*compile-expr*(`expr`$_1$) *for* `t`,*compile-expr*(`expr`$_2$) *for* `t`)

where:

- `step`, `step`$_1$, ,..., `step`$_n$ ($n \in \mathbb{N}$ and $n > 0$) are steps of a path—the '`/?...`' notation means that the path may or may not start at the document's root—;
- `L` is an expression matching (S)XML subtrees;
- `expr`, `expr`$_1$, ... ,`expr`$_p$ ($p \in \mathbb{N}$) are expressions;
- `t`, `t`$_0$ are (S)XML trees;
- `element-name` is the name of an XML element;
- `f` a function belonging to XPath's library, it applies to the current node and its compiled form is $\overline{\texttt{f}}$;
- `op` is a binary operator (for example, a logical connector), and its compiled form is $\overline{\texttt{op}}$.

Fig. 4. Compiling XPath expressions used in `match` attributes.

2.3 Organising compiled forms

Each element name originating from this split operation gives access to the modes that can be used with it. For a particular mode, there are some possible languages. For a particular language, there may be several rules organised by priority. Each rule consists of a list of additional constraints (boolean expressions) and a function to be applied in case of success. The data structure we use can be defined as:

$$\begin{array}{l} \mathit{ELEMENT} \rightarrow \\ \quad \mathit{MODE}^{\#} \rightarrow \\ \quad\quad \mathit{LANGUAGE}^{\#} \rightarrow \mathit{INTEGER} \stackrel{\geq}{\rightarrow} \mathit{CONSTRAINTS} \rightarrow \mathit{FUNCTION} \end{array}$$

where:

- *MODE* and *LANGUAGE* have been introduced at § 2.1;
- *ELEMENT* is an enumerated type for elements' names;
- priorities are integers, of *INTEGER* type;
- *CONSTRAINTS* is the type for sequence of boolean expressions;
- *FUNCTION* is the type of functions implementing a template, given in (1).

The first mapping (*ELEMENT* → ...) is implemented by a hash table, so-called `t-table` in Fig. 5, the others by association lists. The association list whose keys are priorities is sorted decreasingly, what we mean by the '$\stackrel{\geq}{\rightarrow}$' sign. In other words, priorities are arranged w.r.t. a decreasing order.

If we do not consider the language information, handled by nbst, this structure is suitable for an operational semantics of XSLT programs: if an XML tree is matched by several templates having the same priority, the choice among them is left unspecified, that is, implementation-dependent.

The Scheme function putting the `nbst:aply-templates` element into action is given in Figure 5: it uses some macros and functions defined in SRFIs[14] [3,5,14]. If finding some information fails, the false value is returned. Otherwise, the successive mappings are explored until a function implementing the right template is found. Information is directly associated with keys, except for languages, where the list we get is to be searched, by decreasing order of priority for corresponding rules.

3 Going further

It is well-known that some operations are difficult to perform in XSLT. The nbst language could be extended by adding elements more related to programming, as XSieve [11] does for XSLT. (Presently, nbst allows Scheme functions to be called but only inside path expressions, by means of the `call` function [1, App. B].) Our implementation could be useful for XSLT itself, although some features are

[14] Scheme **R**equest for **I**mplementation.

```
(define (n-apply-templates doc-lg cur-lg mode cur-tree t-table root envt)
  ;; See § 2.1 about the meaning of the formal arguments. The root argument is
  ;; needed, because some apply-templates elements may give access to the
  ;; document's root at any point of the program.
  (and-let* (;; Finding the whole information associated with the element name
             ;; of the current tree:
             (a-list (hash-table-ref/default t-table (car cur-tree) #f))
             ;; Finding the information associated with the right mode or #f for
             ;; a template without mode:
             (mode-assoc (assoc mode a-list))
             (alist-0 (cdr mode-assoc))
             ;; Finding the template associated with the language information.
             ;; We consider the document's language in document-dependent
             ;; approach, the current language otherwise:
             (lg-0-assoc (assoc (or doc-lg cur-lg) alist-0)))
    (let (;; Finding the first template whose constraint is fulfilled by the current
          ;; tree:
          (c-assoc (find (lambda (association)
                           ((car association) current-tree))
                         lg-0-assoc)))
      (if c-assoc
          ((cdr c-assoc) doc-lg (car c-assoc) cur-tree t-table root envt)
          ;; Otherwise, backtracking to look for a default template:
          (and-let* ((lg-assoc (assoc #f alist-0))
                     (c-assoc-0 (find (lambda (association)
                                        ((car association) current-tree))
                                      lg-assoc)))
            ((cdr c-assoc-0) doc-lg
                             ;; If a default template is selected, the current
                             ;; language remains the same:
                             cur-lg cur-tree t-table root envt)))))
```

Fig. 5. Applying an nbst template to a current SXML tree.

to added: for example, there is no equivalent to the `xsl:fallback` element in XSLT [16, § 15].

Some techniques related to *partial evaluation* could be used to optimise multiple evaluations among boolean expressions belonging to additional constraints or remove unreachable templates. In the first case, this would lead to an improvement of the *CONSTRAINTS* type and new organisation, based on decision trees. However, we can remark that an nbst program is compiled on the fly, just before being applied, in order to build a 'References' section. Such techniques are probably more accurate for a language like C, where a source program is compiled into an executable file that may be run as many times as we want. Maybe the same approach would be suitable for nbst.

4 Conclusion

We think that our nbst language is suitable for developing multilingual bibliography styles. After some experiment, it seems to us that simple cases are handled easily. But we confess that the whole architecture—handling modes, languages, priorities—can appear as complex. The management of modes is strict, in the sense that mode must coincide between the producer and consumer—the `nbst:apply-templates` and `nbst:template` elements—whereas the management of language information is a kind of inheritance. The present work has practical applications as a guideline for implementations. Even if Scheme is not a strongly typed language, it shows that a precise approach has been followed. It also establishes the behaviour of nbst programs from a mathematical point of view.

References

1. Jean-Michel Hufflen: "MlBibTEX's Version 1.3". TUG*boat*, Vol. 24, no. 2, pp. 249–262. July 2003.
2. Jean-Michel Hufflen: "Implementing a Bibliography Processor in Scheme". In: J. Michael Ashley and Michel Sperber, eds., *Proc. of the 6th Workshop on Scheme and Functional Programming*, Vol. 619 of *Indiana University Computer Science Department*, pp. 77–87. Tallinn. September 2005.
3. Panu Kalliokoski: *Basic Hash Tables*. September 2005. `http://srfi.schemers.org/srfi-69/`.
4. Richard Kelsey, William D. Clinger, Jonathan A. Rees, Harold Abelson, Norman I. Adams iv, David H. Bartley, Gary Brooks, R. Kent Dybvig, Daniel P. Friedman, Robert Halstead, Chris Hanson, Christopher T. Haynes, Eugene Edmund Kohlbecker, Jr, Donald Oxley, Kent M. Pitman, Guillermo J. Rozas, Guy Lewis Steele, Jr, Gerald Jay Sussman and Mitchell Wand: "Revised^5 Report on the Algorithmic Language Scheme". HOSC, Vol. 11, no. 1, pp. 7–105. August 1998.
5. Oleg B. Kiselyov: `and-let*`*: an* `and` *with local bindings, a guarded* `let*` *special form*. March 1999. `http://srfi.schemers.org/srfi-2/`.
6. Oleg E. Kiselyov: "A Better XML Parser through Functional Programming". In: *4th International Symposium on Practical Aspects of Declarative Languages*, Vol. 2257 of LNCS. Springer. 2002.
7. Oleg E. Kiselyov: XML *and Scheme*. September 2005. `http://okmij.org/ftp/Scheme/xml.html`.
8. Leslie Lamport: *LaTeX. A Document Preparation System. User's Guide and Reference Manual*. Addison-Wesley Publishing Company, Reading, Massachusetts. 1994.
9. Kirill Lisovsky and Dmitry Lizorkin: "XML Path Language (XPath) and its functional implementation SXPath". *Russian Digital Libraries Journal*, Vol. 6, no. 4. 2003.
10. Kirill Lisovsky and Dmitry Lizorkin: "XSLT and XLink and their implementation with functional techniques". *Russian Digital Libraries Journal*, Vol. 6, no. 5. 2003.
11. Oleg Parashchenko: *XSieve: extending* XSLT *with the roots of* XSLT. 2006. `http://xmlhack.ru/protva/xtech2006-paper.pdf`.

12. Oren PATASHNIK: *BibTeXing*. February 1988. Part of BibTeX's distribution.
13. Erik T. RAY: *Learning* XML. O'Reilly & Associates, Inc. January 2001.
14. Olin SHIVERS: *List Library*. October 1999. `http://srfi.schemers.org/srfi-1/`.
15. W3C: XML *Path Language (XPath). Version 1.0.* W3C Recommendation. Edited by James Clark and Steve DeRose. November 1999. `http://www.w3.org/TR/1999/REC-xpath-19991116`.
16. W3C: XSL *Transformations (* XSLT*). Version 1.0.* W3C Recommendation. Edited by James Clark. November 1999. `http://www.w3.org/TR/1999/REC-xslt-19991116`.