SXSLT: Manipulation Language for XML

Oleg Kiselyov
Fleet Numerical Meteorology and
Oceanography Center
Monterey, CA

Shriram Krishnamurthi
Brown University

Fifth International Symposium on
Practical Aspects of Declarative Languages:
PADL 2003. January 14, 2003. New Orleans,
LA, U.S.A.

`http://ssax.sourceforge.net/`

This talk {– SXSLT: Manipulation Language for XML –} is a joint work with Shriram Krishnamurthi of Brown University.

The title of the talk invites the question: aren't there already enough XML manipulation languages? For one, there is XSLT, which is a Web Consortium Recommendation. There are several other tools. There are a number of books about XSLT and these tools.

The trouble with XSLT

"The XSLT designers created a language that, while rich in domain-specific functionality, lacked much of the basic functionality necessary to make it genuinely suited for its intended purpose. While the designers probably left 'generic' functionality out of the spec on the grounds that XSLT was never intended to be a general-purpose programming language, they failed to realize that even simple document transformations often require a little nuts-and-bolts programming. Leaving out the nuts and bolts made XSLT a half-broken language."

Moertel, T.: XSLT, Perl, Haskell, & a word on language design. Posted on kuro5hin.org on January 15, 2002.

The trouble with XSLT is that it is half-baked (or half-broken). In any case it is only half-usable. XSLT is never meant to be a general-purpose language. Moreover, as the author of this article wrote: ”The really bad thing is that the designers of XSLT made the language free of side effects but then failed to include fundamental support for basic functional programming idioms. Without such support, many trivial tasks become hell.”

Languages for handling XML

Adam Bosworth:

"I noticed two new categories of books: books about programming with Java and XML and books about XSLT. In both cases, whole forests are dying to compensate for the XML community's one great failure – the lack of a decent programming model for manipulating XML."

...

"We need a language that can natively support XML as a data type and yet can gracefully integrate with the world of objects and can take advantage of the self-describing nature of XML by supporting querying of its own variables. This language as used by humans will look like a programming language, not an XML grammar. This is the language we will use to convert from one XML format to another."

Speaking about books. Adam Bosworth, a vice president for engineering of the leading Web Services vendor BEA Systems Inc., took a stroll through his local bookstore, saw those books about XSLT, and wrote: "whole forests are dying to compensate for the XML community's one great failure – the lack of a decent programming model for manipulating XML. We need a language that can natively support XML as a data type and yet can gracefully integrate with the world of objects and can take advantage of the self-describing nature of XML..." Adam Bosworth then described general requirements for such a language, and pointed out the most glaring drawback of XSLT – its XML syntax. It turns out, he wrote, that there are things that XML isn't well suited for, and expression languages that humans can read and write easily is one of them.

Languages for handling XML...

"I contend that until we have a language that natively understands the data structures inherent in XML and enables optimized algorithms for processing it, we will not have real XML programming."

Adam Bosworth: Programming Paradox. XML Magazine, February 2002.

His article finishes with the phrase: "I contend that until we have a language that natively understands the data structures inherent in XML and enables optimized algorithms for processing it, we will not have real XML programming."

I contend that we do have such a language that natively supports XML data structures and their efficient manipulations and transformations – and also reflection. SXSLT fits all Adam Bosworth's requirements. SXSLT is a library extension of Scheme, which is a dialect of Lisp. Lisp, incidentally, was specifically designed for processing of semi-structured data. The 40+-year history of the language proved that it and its syntax are quite suitable for describing both data and code.

A simple XSLT example

Problem: `"& text $"` $\Longrightarrow$ `"\& text \$"`

```
<xsl:stylesheet>
  <xsl:template name="doSubstitutions">
    <xsl:param name="input" select="."/>
    <xsl:param name="output">
      <xsl:call-template name="substitute">
        <xsl:with-param name="input" select="$input"/>
        <xsl:with-param name="target">&amp;</xsl:with-para
        <xsl:with-param name="replacement">\&amp;</xsl:wi
      </xsl:call-template>
    </xsl:param>
    <xsl:call-template name="doSubstitutions2">
      <xsl:with-param name="input" select="$output"/>
    </xsl:call-template>
  </xsl:template>

  <xsl:template name="doSubstitutions2">
    <xsl:param name="input" select="."/>
    <xsl:param name="output">
      <xsl:call-template name="substitute">
        <xsl:with-param name="input" select="$input"/>
        <xsl:with-param name="target">$</xsl:with-param>
        <xsl:with-param name="replacement">\$</xsl:with-p
      </xsl:call-template>
    </xsl:param>
    <xsl:value-of select="$output"/>
  </xsl:template>
</xsl:stylesheet>
```

Before we proceed further let's take a simple example of SXSLT.

We will use the example from the kuroshin article cited earlier. Its goal was a function (or an XSLT template) to escape bad characters during an XML-to-LaTeX conversion. For example, given character data like this `"& text $"` the function should return this `"\& text \$"` with ampersand and dollar signs properly quoted. The article gives this XSLT solution. I had to use a smaller font to fit the code on one slide. It's not the complete code: we need an external template named `substitute` to perform the individual string-for-string substitutions. {First, the template named doSubstitutions escapes ampersands in the input text and then calls doSubstitutions2 on the result. Second, doSubstitutions2 escapes dollar signs in its input text and returns its results. The article discusses this and related solution in detail.}

A simple SXSLT example

```
(define doc '(p "& text $"))

(define main-ss
  `((p . ,(lambda (tag . elems)
       (list elems nl nl)))
    (*text* . ,(lambda (tag str)
       (if (string? str)
           ((make-char-quotator
              '((#\& . "\\&") (#\$ . "\\$")))
              str)
           str)))))

(SRV:send-reply (pre-post-order doc main-ss))
```

Result: `\& text \$`

Here is the same example in SXSLT. Here {define doc...} is a sample document, in a parsed form. Both XSLT and SXSLT operate on an abstract syntax tree of an XML document. We run SXSLT by invoking a function pre-post-order and passing it a parsed XML tree and a stylesheet. The function send-reply writes out the transformed tree.

The function make-char-quotator makes a specialized substitutor. We can easily and concisely specify the list of characters to quote, and their replacements. The XSLT code shown before achieves the similar effect, at the cost of great verbosity and obscurity.

This paper and these slides were authored in SXML and converted by a similar process into $\LaTeX$ and then into PDF. The fact you can see ampersand, dollar, *and* backslash characters here at all shows that our quotation works.

I'd like to point out now that the SXSLT stylesheet is an ordinary Scheme data structure, an associative list of tags and the corresponding handlers. The handlers are ordinary Scheme functions, which can invoke arbitrary Scheme procedures.

Outline

- Motivation and comparisons
  - Authoritative quotes
  - A simple SXSLT example
  - XSLT drawbacks and XSLT/SXSLT comparisons
- SXML: an XML AST
- An elaborate example
- Ease of use

So, SXSLT is a practical, higher-order, concise, expressive and readable declarative XML transformation language. {The language is a head-first rewriting system over abstract XML syntax trees, implemented as a library extension of Scheme.}

Now that we have seen a simple SXSLT example, let us talk about what XSLT cannot do but we can – and also what XSLT can do, and we can do too, often better.

Drawbacks of XSLT

- Poor syntax
- Missing nuts-and-bolts
- Low order
- Closed system

The articles by Moertel and Bosworth have identified the following significant drawbacks of XSLT, which are overcome in SXSLT.

Poor syntax First, large XSLT programs are so verbose that they become unreadable. We have seen that. Fewer and fewer people consider XML to be an acceptable notation for a programming language.

Missing nuts-and-bolts We have seen that too.

Low order XSLT templates are not first class. A transformed document or its part cannot easily be retransformed.

Closed system Finally, XSLT is designed as a closed system.

Common features of XSLT and SXSLT

- Infoset transformation
- Node dispatch
- Multiple effective rulesets
- Re-traversals
- Tree fold model

Let us point out what XSLT *can* do, and so can we, often better.

XSLT and SXSLT both assume that the source XML document has been parsed into a tree form, which represents the XML Infoset. They both transform the tree as instructed by a stylesheet. The result should be pretty-printed afterwards.

Both XSLT and SXSLT let the user specify transformers for particular tree nodes. Both languages permit wildcard rules and a context-sensitive dispatch. But our dispatch is far simpler: SXSLT is a head-first re-writing system. We do not burden the user with computing priorities of node handlers – and yet we achieve the same power as XSLT.

Both XSLT and SXSLT permit repeated traversals of the source document with the same or a different effective stylesheet. Our transformations are again simpler because we have no modes. In addition, we can re-traverse the result tree itself – something XSLT in version 1.0 cannot do. {Our pre-post-order combinator is akin to apply-templates. But apply-templates is a statement, whereas pre-post-order is a regular function that returns the transformed tree. The result can be easily manipulated –or transformed again.} In SXSLT, transformation templates are first class, which expands the expressiveness to a large degree, without any need for additional syntactic sugar.

Both XSLT and SXSLT implement the tree fold model. Only SXSLT however implements the genuine *higher-order*{, also known as CPS,} tree fold, which is the most general traversal combinator. {Our preorder traversal mode is an _optimization_ for partial traversals}.

The paper gives a more detailed comparison. The paper also discusses similarities and differences between SXSLT and Scheme macros.

Outline

- Motivation and comparisons
- SXML: an XML AST
- Elaborate Example
- Ease of use

Omitted from the talk: parsing, SXPath

This is the outline for the rest of the talk. First we will introduce SXML, which is an abstract syntax of an XML document. SXML underlies all our query and transformation tools.

We then expound SXML transformations on a simple and yet representative and practical example. We will give third-party evidence that SXSLT is indeed practical, is easy to use, and has been used.

In this talk we will not mention XML parsing and SXML queries. The paper gives the necessary references.

XML and SXML

```
<RESERVATION
 xmlns:HTML='http://www.w3.org/TR/REC-html40'>
<NAME HTML:CLASS="largeSansSerif">
    Layman, A</NAME>
<SEAT CLASS='Y'
  HTML:CLASS="largeMonotype">33B</SEAT>
<HTML:A HREF='/cgi-bin/ResStatus'>
    Check Status</HTML:A>
<DEPARTURE>1997-05-24T07:55:00+1
</DEPARTURE></RESERVATION>
```

```
(*TOP* (*NAMESPACES*
        (H "http://www.w3.org/TR/REC-html40"))
 (RESERVATION
  (NAME (@ (H:CLASS "largeSansSerif"))
      "Layman, A")
  (SEAT (@ (H:CLASS "largeMonotype")
           (CLASS "Y")) "33B")
  (H:A (@ (HREF "/cgi-bin/ResStatus"))
       "Check Status")
  (DEPARTURE "1997-05-24T07:55:00+1")))
```

As we said, at the core of our tools is SXML, which is an abstract syntax tree of an XML document. SXML is also a concrete representation of the XML Information set in the form of S-expressions. {SXML fully supports XML Namespaces, processing instructions, parsed and unparsed entities.}

This example eliminates the need for any further explanations. We see an XML document and the corresponding SXML representation. Please note the difference in the font size. The XML document on the slide is actually taken from the XML Namespaces Recommendation. You can see how SXML represents attributes {(@ ...)} and namespaces. {In SXML, all names are fully resolved. Because namespace (Universal Resource Identifiers) URIs are often too long, it's possible to tell the parser to use shorter namespace shortcuts, in this example, `H`. Note that these shortcuts have nothing to do with the XML prefixes in the original document {`HTML:CLASS`, etc.}. The XML prefixes are controlled by the author of the document; the shortcuts {`H:CLASS`} are controlled by the developer of an XML application. Furthermore, shortcuts and namespace URIs are in a 1-to-1 correspondence. This is not the case for XML prefixes.}

Our XML parser and pretty-printing tools convert between SXML {this} and the angular-bracket-format of XML documents {that}.

{The paper defines SXML formally.}

Patterns of SXML transformations

XML - SXML -*- SXML - xML, DB, File

In particular:

XML $\rightarrow$ SXML $\rightarrow$ SXML $\rightarrow$ xML, LaTeX, DB

SXML $\rightarrow$ SXML $\rightarrow$ xML, DTD, LaTeX

The general pattern of SXML transformations can be written like this {first phrase}. We can apply it in both directions.

We can start with XML, parse it in SXML and do various transformations and queries. We store the result as a new XML or HTML or LaTeX document, or put the data into a database. This approach resembles the ordinary XSLT processing.

On the other hand, we can start with SXML. That SXML can be generated from a database query, retrieved from a file, or entered by hand in Emacs. We can transform it several times – sometimes in a rather intricate ways – and generate a markup or a TeX document. Incidentally this approach lets us author web pages, XML documents, or papers – in SXML. The SXML specification, for example, was authored in SXML, and later converted into a LaTeX document and a web page, given different stylesheets. The present paper in the proceedings and the slides you are seeing were also authored in SXML.

Let us see a detailed example of one such transformation.

Sample SXML document

```
(define doc
  '(html (head (title "Document"))
    (body
      (section "First Section"
        (p "This is the intro section.")
        (p "Paragraph &c"))
      (section "Second Section"
        (section "A sub-section"
          (p "This is section 2.1."
            (br)
            (a (@ (href "another doc.html"))
              "link")))
        (section "Another sub-section"
          (p "This is section 2.2.")))
      (section "Last major section"
        (p "This is the third section")))))
```

The paper shows several advanced transformations including translation of SXML into LaTeX, context-sensitive and recursive reorganizations of the document. In this talk, we will use the most familiar topic of Web authoring.

Suppose we are given a markup for an HTML document, which contains sections and subsections. {I'd like to draw your attention to the ampersand character {here} and to the space in the name of the document here. Obviously these characters must be encoded in the output HTML document.}

Desired HTML document

- 1. First Section
- 2. Second Section
  - 2.1. A sub-section
  - 2.2. Another sub-section
- 3. Last major section

1. First Section

This is the intro section.

Paragraph &c

2. Second Section

2.1. A sub-section

This is section 2.1.
link

2.2. Another sub-section

This is section 2.2.

3. Last major section

This is the third section

We need to hierarchically number the sections and subsections. We should output the HTML document like this, with numbered sections. We should use the appropriate HTML tags (H2, H3, ...) to set off the title of the sections and subsections.

In addition, we want to generate a hierarchical table of contents and place it at the beginning.

This example epitomizes common XML processing tasks. The example is due to Jim Bender, who used a similar transformation for compiling an XML Schema into ASN.1 specifications.

The main stylesheet

```
(SRV:send-reply (pre-post-order doc main-ss))

(define main-ss
  `((body *preorder*
       . ,(lambda (tag . elems)
            (let ((numbered-sections
                     (number-sections '() elems)))
               ...
               )))
      ...
      ,@universal-conversion-rules)))
```

{You can get the full code of this example from the SSAX SourceForge project.}

The following expression executes the transformation of the source document into a target SXML tree: a tree of HTML fragments. The function {SRV:send-reply} then writes out that tree on the standard output. The tree transformation is performed by a traversal combinator pre-post-order with the help of the following stylesheet.

{The approach taken in this example is not meant to be the most efficient. Rather, it is aimed to be illustrative of various SXSLT facilities and idioms. In particular, we demonstrate: higher-order tags, pre-order and post-order transformations, re-writing of SXML elements in regular and special ways, context-sensitive applications of re-writing rules. Finally, we illustrate SXSLT reflection: an ability of a rule to query its own stylesheet.}

The general SXML-to-HTML conversion is taken care by the `*default*` and `*text*` rules in 'universal-conversion-rules'. These rules are written once and for all and do not need to be changed at all. The universal rules check text strings for dangerous characters such as angular brackets and the ampersand and encode them. The `*default*` rule turns an SXML element into the appropriate HTML element. These transformations will be uniformly applied to all nodes of the source SXML tree. We `only` need to add rules for the SXML elements that have to be treated in a special way.

The `body` of the document, a collection of sections, has to be processed specially. First we recursively number the sections...

Recursive numbering of sections

```
((section title (section title el...) ...)
 (section title (section title el...) ...) ...)
 =>
((*section (1) title (*section (1 1) title...) e
 (*section (2) title (*section (1 2) title...) e
```

```
(define (number-sections ancestors sections)
 (map
  (lambda (el i)
   (pre-post-order el
    `((section *preorder* .
        ,(lambda (tag title . elems)
           (let ((my-number (cons i ancestors)))
             (cons* '*section my-number title
               (number-sections my-number elems))
       (*default* *preorder* . ,(lambda x x))
       (*text* . ,(lambda (tag str) str)))))
  sections (list-numbering sections)))
```

By recursive numbering of sections we mean: given a list of `sections`, return the list of `*sections` as shown on the slide here.

The numbering (the first element of a `*section`) is a list of section numbers in reverse order: the numeric label of the current section followed by the labels of its ancestors.

A `section` of the source document may contain either (sub)sections, or other SXML nodes such as strings or paragraphs {... here}. The numbering transformation should not affect the latter nodes.

The function 'number-sections' illustrates a typical XSLT-like processing. We transform an SXML tree by invoking the traversal combinator pre-post-order and passing it the source tree and a stylesheet. The stylesheet {here} has only three rules. The `*text*` rule is the identity rule. {It passes the character data from the source SXML tree to the result SXML tree as they are.} The `*default*` rule is also an identity rule. A `*preorder*` label by the rule tells pre-post-order to return a non-`section` branch as it is, *without* recursing into it. A `section` rule tells what to do when we encounter a section: we make a (*section ...) element out of the title and the numbered children of the section in question.

We should point out that the traversal combinator pre-post-order is an ordinary Scheme function, and can be *mapped*. The stylesheet handlers are likewise ordinary Scheme functions, which can invoke other Scheme functions, including pre-post-order itself.

Building a hierarchical table of contents

```
((*section (1) title1 non-section-el ...)
 (*section (2) title2
    (*section (1 2) title21 el...) el...)...)
 =>
((li "1." title1)
 (li "2." title2 (ul (li "2.1." title21) ...)) .
```

```
(define (make-toc-entries nsections)
 (pre-post-order nsections
  `((*section .
      ,(lambda (tag numbering title . elems)
          (let ((elems (filter pair? elems)))
           `(li ,(numbers->string numbering)
               ". " ,title
               ,(and (pair? elems)
                  (list 'ul elems))))))
    (*default* . ,(lambda _ '()))
    (*text* . ,(lambda (tag str) str)))))
```

It is more lucid to build the TOC in a separate pass, by traversing the previously numbered sections. In this pass, we turn `*section` elements into TOC entries, and rewrite everything else to nothing. To be more precise, we do the following transformation {on the slide}.

Again, we execute pre-post-order with a three-rule stylesheet. As before, the character data are not affected: see the `*text*` rule. The `*default*` rule is different now: it transforms a non-`section` branch to nothing at all. {To be more precise, an SXML element other than `*section` is turned into an empty list, which will be disregarded later.}

The stylesheet almost literally looks like the above rewriting example. We should note that the stylesheet rules are applied to all elements of the tree, recursively. We indeed process the arbitrary nesting of `*sections` without much ado. We do not need to write something like `<apply-templates/>`. Unlike XSLT, but like tree fold, the pre-post-order combinator traverses the tree in post-order by default. That is, the handler for the `*section` rule `(lambda (tag numbering title . elems) ...)` receives, as `elems`, the list of the *transformed* children of the `*section` in question, which are the list of TOC elements for internal subsections – or the list of nothing.

{We use an auxiliary function `numbers->string` to convert the list of numerical labels such as `(1 2 3)` into a string label "3.2.1"}

Insertion of TOC and re-traversal

```
(define main-ss
 `((body *preorder* .
    ,(lambda (tag . elems)
       (let* ((nsections
                (number-sections '() elems))
               (toc
                (make-toc-entries nsections)))
                ; re-apply the main-ss
          (pre-post-order
           `(body (ul ,toc) ,nsections)
            (append `((body .
              ,(cdr (assq '*default* main-ss))))
             main-ss))))))
    ...
    ,@universal-conversion-rules))
```

Let us go back to the main stylesheet. First we recursively number the sections and replace them with *sections. We saw how to do that. We then pass the renumbered sections to make-toc-entries and get the list of TOC entries.

We insert the TOC elements before the numbered sections, and re-apply the stylesheet. Actually, we re-apply a slightly *modified* stylesheet: the element `body` no longer needs to be processed specially. We can indeed re-apply a stylesheet with some dynamic modifications. XSLT can accomplish something similar, with the help of modes. SXSLT gives far simpler tools to dynamically 'modify' the effective ruleset. There are, of course, no mutations. SXSLT is purely declarative. We merely re-invoke pre-post-order and pass it main-ss with the modified rules prepended.

The 'overridden' rule for the `body` element has the same handler as that of the **`*default*`** rule. To find the latter, we *query* the stylesheet {main-ss} itself! Indeed, the stylesheet is a simple data structure, an associative list, and can be queried as such. This example demonstrates reflexive abilities of SXSLT. A stylesheet can analyze itself.

Making Sections

```
(*section (1 2 3) "title321" elem ...)
 =>
((h4 "3.2.1. " "title321") elem ...)


(define main-ss
 `((body *preorder* ...)
   (*section *preorder* .
     ,(lambda (tag numbering title . elems)
       (let ((header-tag
               (list-ref '(h1 h2 h3 h4 h5 h6)
                 (length numbering))))
         (pre-post-order
           `((,header-tag
              ,(numbers->string numbering)
               ". " ,title)
             ,@ elems)
           main-ss))))
     ...
     ,@universal-conversion-rules))
```

On the re-application pass, the traversal combinator treats `body` as any other element. The combinator processes its children first, and now notices `*section` elements. We transform a `*section` as shown on the slide, and re-apply the main stylesheet. {Our auxiliary function `numbers->string` helps us to convert the list of labels to a string. We use the length of the list (that is, the depth of the section in question) to choose the HTML tag for the section: h2, h3, h4, etc.}

We should point out that `*section` elements are processed twice, with two *different* stylesheets. First we scan `*sections` and turn them into TOC entries. Later we turn the same `*sections` into HTML headers.

The elements `section` and `body` of the source document act as *higher-level* SXML elements. They are recursively re-written into more primitive SXML elements until they are finally turned into HTML text fragments. Essentially, we compute the fixpoint of the re-writing stylesheet. We do not iterate on the whole document however, only on the branches that need iterating. The whole approach is rather similar to that of Scheme macros. {Scheme macros do not have default rules however. R5RS macros cannot transform in post-order and cannot explicitly re-invoke the macro-expander.}

Context-sensitive transformations

```
(define doc
  '(...
     (a (@ (href "another doc.html"))
       "link"))) ...


(define main-ss
  `((body *preorder* ...)
    (*section *preorder* ...)
    (href
      ((*text* .
         ,(lambda (tag str)
           (if (string? str)
             ((make-char-quotator
                '((#\space . "%20"))) str) str)))
      . ,(lambda (attr-key . value)
           ((enattr attr-key) value)))
     )
    ,@universal-conversion-rules))
```

We have one more thing to take care of. The source document had an anchor element {quoted here, on the slide} with the name of a local file in the `href` attribute. In the output HTML document, that name will become a URL. File names may contain spaces – but URLs may not. Therefore, we need to encode the space character. We should URL-encode the space character only in the context of the `href` node, and nowhere else. The white space elsewhere in the document must remain the white space. Hence we need a special rule for `href`, with its own handler for character data. This `*text*` handler is *local*: it acts only in scope of the `href` node. The local text handler looks for the space character and URL-encodes it. We have just shown a *context-sensitive* application of re-writing rules. It still appears clear and intuitive.

Ease of Use

http://www.netfort.gr.jp/~kiyoka/sxmlcnv/index_ja.html

Sxmlcnvとは？

- Sxmlcnvはフリーソフトウェアです。
- GNU General Public License (GPL2)のもとで配布されています。
- XMLとSXMLを相互変換するソフトウェアです。

特徴

- Gaucheの全ての機能を使ってプログラマブルなドキュメントを作成できます。

SXMLとは？

- SXMLはXML Infosetの一種で、XMLの持つ情報量を損なわずにS式で記述できるようにしたフォーマットです。
- 本ソフトウェアではssaxというソフトウェアを使用して、XMLをパースしています。

何をするためのもの？

個人的にはSmartDoc という XMLドキュメントフォーマットをもっと書きやすくできないかと考えて、これを作るにいたりました。

サンプル

このページは Sxmlvncを使って作成しています。

このページがサンプルとして最適でしょう。

- scmソース(UTF-8)
- 生成したSmartDocソース(UTF-8)

SXSLT is easy to use. Independent developers can easily learn it and use it for their projects. I'd like to show a few pieces of evidence for such a claim. I'd like to emphasize that we had no participation and even no knowledge of the projects and their authors.

The web page with this URL is one example. The slide is a partial snapshot of that page. It is in Japanese (although it is now translated into English). The author wrote {here} that he wanted to *easily* write XML for a SmartDoc system. He came across SXML, apparently found it useful, and even wrote a tool to perform the desired conversion to SmartDoc.

This link, under sample, points to the source code for the page, which looks as follows...

Ease of Use ...

http://www.netfort.gr.jp/~kiyoka/sxmlcnv/index.scm

```
(define (M:link keyword url)
  `(a (@ (href ,url)) ,keyword))
(define (W:sxml)
  (M:link "SXML" "http://okmij.org/ftp/Scheme/SXML.html")
(define (W:xml)
  (M:link "XML"  "http://www.w3.org/XML/"))

(define (L:body)
  `(body
    (center
     (a (@ (href "mailto:kiyoka@netfort.gr.jp"))
        "email: Kiyoka  Nishiyama"))

    (section
     (@ (title  "Sxmlcnv ..."))
     ,(title-en "What is Sxmlcnv?")
     (p
      (@ (locale "ja"))
      (ul
       (li "Sxmlcnv ...")
       (li ,(W:GPL) " ...")
       (li ,(W:xml) "..." ,(W:sxml) "...")))
```

Given the examples we saw before, I believe this should look familiar... I don't have a good unicode font so I replaced non-ASCII characters with dots.

Ease of Use ...

http://homepage1.nifty.com/blankspace/scheme/nsx.html

neat sxml

S-expressed XML

今やどこもかしこも XML ばかりです.
この XML ってのはタグで囲まれてますけど, 長文な文章があるようなものでない限り, データよりタグの方が多いことって 結構ありますよね.
つまりデータがタグに埋もれているので非常に読みにくい, ということです.

んで, S 式な人から見ると, 閉じタグって邪魔じゃんって発想になるんですよね.
まあなぜ閉じタグがあるかというのは単純な検索でテキストを処理しやすいとか lisp み たくカッコだけだと対応をミスったときにエラー個所を特定するのが難しいとかいろいろ あるみたいです. まあそれはしょうがない.

それなら, S 式で同等の内容を記述して, それを必要に応じて XML に展開するようにし て普段は S 式の方をメンテナンスしようよ, という発想はとても自然だと思われます.
構文の定義の仕方はいろいろやり方がありそうですが, 以下では SXML というのを使わし てもらいます.
SXML はOleg Kiselyov さんの site で紹介されているものです.
この人の site にはコードサンプルが沢山あります.
XML の parser tool とか, 高度な話題で満載です.
ちなみにkawai さん の site では 上記 Oleg さんのツールを Gauche で使えるようにする方法があります.

SXML は, 百聞は一見に如かず, 例を見ればすぐわかるでしょう.
以下は XML というか XHTML ですけど,

sxml	**xml**
`(html`	`<html>`
` (head (title "sxml"))`	` <head><title>sxml</title></head>`
` (body`	` <body bgcolor="blue" text="white">`
`  (@ (bgcolor "blue")`	`  <center><b> Hello.</b></center>`
`     (text    "white"))`	` </body>`
`  (center (b "Hello."))))`	`</html>`

Here is another page. It says that in XML, tags bury the content. SXML comes to rescue. The web page goes to explain how to convert SXML into the format reminiscent of James Clark's Jade. The author then describes higher-order SXML tags and a tree fold.

All these examples {here} were done by the author of that page, without any assistance from me. In fact, I don't even know his name.

You may conclude that either SXML transformations are indeed so easy to use that they transcend linguistic boundaries. Or I am able to communicate ideas in Japanese. Whichever conclusion you reach, I'll take it.

Conclusions

SXSLT is a domain-specific declarative layer over a general-purpose functional language

Base-language features:

- mature syntax
- first-class and higher-order transformers and traversal combinators
- extensibility
- ”nuts-and-bolts.”

We have presented a practical XML transformation language that is free from the drawbacks of XSLT. {This language, SXSLT, is more expressive than XSLT and is also simpler and more coherent.} The language is implemented as a library in a pure functional subset of Scheme.

The base language, Scheme, gives us mature syntax, first-class and higher-order transformers and traversal combinators, extensibility, and "nuts-and-bolts."

Conclusions...

Domain-specific features:

- abstract XML syntax trees
- pre- and post-order traversals
- transformation environments (stylesheets)
- name-based dispatch to re-writing transformers
- wildcard transformation rules
- local scoping of re-writing rules

R5RS dictum: rather than add features, remove weaknesses and restrictions

`http://ssax.sourceforge.net/`

The domain-specific layer implements right abstractions and patterns of XML transformations: (i) abstract XML syntax trees, (ii) pre- and post-order traversals of abstract syntax trees, (iii) transformation environments (stylesheets), (iv) dispatch to re-writing transformers based on node names, (v) wildcard transformation rules, (vi) local scoping of re-writing rules.

We believe the combination of these features is unique and powerful. We must stress that the features are not heaped up indiscriminately, but are induced by a compact core of tree traversal combinators and first class transformers.

In the design of SXSLT, we strove to follow the R5RS dictum: Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary.

The software mentioned in this talk is in public domain. It is a SourceForge project. You are very welcome to use it. If you have any question, please post it on the project mailing list, or send it to me.