Digital Texts with XML and the TEI 4: Using TEI XML

Lou Burnard
February 2004

Today's topics

- Now that it's all in TEI XML, what next?
- More ways of using XSLT
- Two flavours of XML delivery system

What is XML for?

- exchanging information
  1. between people
  2. between people and machines
  3. between machines
- preserving information
  1. without usage-dependency
  2. without medium-dependency
  3. independent of time, space, and language

Delivering information

XML is an excellent way of representing and preserving information. But how about

- delivering XML content on the web
- ... and on paper
- storing and managing XML documents
- ... and virtual documents

Can we get the best of both worlds?

What tools do we need?

- Appropriately expressive vocabularies (eg TEI XML)
- Syntax-checking document creation tools (aka Editors)
- Document transformation tools
- Document delivery tools
- Document storage and management tools
- Programming interfaces
- Specialized applications

A choice of generic XML vocabularies

- XML Schema: describes structures and data types;
- XPath: describes how to address any part of an XML document
- XSLT: describes how to transform an XML document;
- XQuery: an XML database query language.

Some example specialised XML vocabularies

- SVG: scalable vector graphics;
- MathML: Mathematical Markup Language;
- RDF: Resource Description Framework;
- SMIL: Synchronised Multimedia Integration Language

... etc etc etc
The TEI provides an extensible framework in which these may be integrated.

Document creation and editing

There's an ever expanding choice of XML editing tools:

- Customised plain text editors, with built in tagging (e.g. Notetab)
- Customised programming editors (notably GNU Emacs)
- Word processors (e.g. Word2000, Open Office)
- Data-oriented XML editors (eg XML Spy)
- Document-oriented XML editors (eg XMetal)

And there's also the XML that gets generated without anyone noticing...

Typical transformation jobs

1. Render `<foo>` elements in italics
2. Render `<foo>` elements within `<bar>` elements in italics
3. Insert `Foo number` and the value of its `number` attribute in front of every `<foo>` element
4. Indent every `<p>` element by 1 em, except for the first one in a `<div>`
5. Take the first `<head>` element inside each `<div>` and add it to a table of contents

Less obvious transformation jobs

1. Count `<foo>` elements occurring within `<bar>` elements
2. Sort all `<foo>` elements by the value of their `which` attribute, suppressing duplicates
3. Display only `<foo>` elements whose `which` attribute has the same value as a `<bar>` element elsewhere
4. Display every `<p>` element containing some string
5. Display the parent element of every `<foo>` element, sorting them by the value of the `which` attribute on the last `<bar>` element they contain

Typical XSLT techniques (1)

Start at the root and process everything along the `descendant` axis:

```
<xsl:template match="/">
 <html>
   <body>
    <xsl:value-of select="count(descendant::pb)"/>
   </body>
 </html>
</xsl:template>
```

Re-order the nodes as you find them:

```
<xsl:template match="/">
  <html>   <body>    <table>
      <xsl:apply-templates select="descendant::div">
        <xsl:sort select="head"/>
      </xsl:apply-templates>
  </table>  </body> </html>
</xsl:template>
<xsl:template match="div">
  <tr>
    <td><xsl:number level="any"/></td>
    <td><xsl:value-of select="head"></td>
  </tr>
</xsl:template>
```

Some `<xsl:sort>` examples

`<sort select="head">`: sort on child `<head>` element
`<sort select=".">`: sort on text of current element
`<sort select="@n">`: sort on value of `n` attribute
`<sort select="string-length(.)" data-type="number">`: numeric sort on length of text of element
`<sort select="surname"> <sort select="forename">`: sort on first surname, then forename
`<sort select="@id" order="descending">`: reverse order sort by @id elements

Typical XSLT techniques (2)

Stash the contents of an element away for re-processing using `<xsl:variable>`:

```
<xsl:variable name="terminator">
<xsl:text>****</xsl:text>
</xsl:variable>

<xsl:template match="p">
 <xsl:variable name="paraContents">
    <xsl:apply-templates select=".//text()"/>
 </xsl:variable>
 <xsl:number level="any"/>:
 <xsl:value-of
   select="string-length($paraContents)"/>
 <xsl:value-of
   select="substring-before($paraContents,
           $terminator)"/>
 </xsl:template>
```

Typical XSLT techniques (3)

Recursion is *the* way to handle many tasks. A typical pattern is to use a named template which calls itself, like this:

```
<xsl:template name="do_it">
   <!-- do something -->
   <xsl:if test="not_finished_yet">
     <xsl:call-template name="do_it"/>
   </xsl:if>
</xsl:template>
```

Parameters can be passed in to the template in the usual way:

```
<xsl:template name="do_it">
 <xsl:param name="param"/>
   <!-- do something -->
   <xsl:if test="not_finished_yet">
     <xsl:call-template name="do_it"/>
       <xsl:with-param name="param">
          <!-- some value -->
       </xsl:with-param>
     </xsl:call-template>
   </xsl:if>
</xsl:template>
```

Example: print the `<foreign>` words, word by word

Call the template :

```
<xsl:template match="foreign">
   <xsl:call-template name="do_words">
     <xsl:with-param name="text">
      <xsl:value-of select="."/>
     </xsl:with-param>
   </xsl:call-template>
</xsl:template>
```

Example (continued)

Here's the template:

```
<xsl:template name="words">
  <xsl:param name="text"/>
<xsl:choose>
<xsl:when test="contains($text,' ')">
<xsl:message>
 <xsl:value-of select="substring-before($text,' ')"/>
</xsl:message>
<xsl:call-template name="words">
   <xsl:with-param name="text">
     <xsl:value-of select="substring-after($text,' ')"/>
   </xsl:with-param>
 </xsl:call-template>
</xsl:when>
<xsl:otherwise>
  <xsl:message>
    <xsl:value-of select="$text"/></xsl:message>
</xsl:otherwise>
</xsl:choose>
</xsl:template>
```

Typical XSLT techniques (4)

A lookup table is a convenient way of transforming a coded value (the *key*) into some other *value*. Suppose we have `<category>` elements like this:

```
<category id="C1">good</category>
<category id="C2">bad</category>
<category id="C3">indifferent</category>
```

which are referenced elsewhere in the document like this:

```
<pot id="123">...
<condition status="C2">....</condition>... </pot>
```

Lookup Tables contd.

The `<xsl:key>` element is used to create a lookup table in our stylesheet:

```
<xsl:key name="category_table" match="category" use="@id"/>
```

The XSLT processor will make an index of the location of all `<category>` elements, based on their `id` attributes. Now we can access the index in a template as follows:

```
<xsl:template match="condition">
<xsl:value-of  select="key('category_table',@status)"/> condition

</xsl:template>
```

```
<xsl:for-each select="key('category_table','C2')">
 The following pots are in a bad way:
 <xsl:value-of  select="ancestor::pot[1]/@id"/>
</xsl:for-each>
```

In summary: what is XSLT good for?

- ☛ Extracting and reorganizing texts for further processing
- ☛ Summarizing and analysing texts
- ☛ Validating textual content
- ☛ Converting texts into other formats

It is not *just* a way of turning stuff into web pages...

Storage strategies

Data has to be stored somewhere. How should XML data be managed? There are several possibilities:

1. as discrete XML documents
2. within any convenient DBMS
3. within an XML fragment repository

XML documents

In the traditional docucentric world...

- information is stored in XML documents, somewhere, and in some form
- entities give some degree of modularity
- but there has to be centralized naming and management for version control, integrity, etc.
- appropriate for static documents only

```
<!ENTITY doc1 SYSTEM "docs/frag1.xml">
<!ENTITY doc2 SYSTEM "docs/frag2.xml">
```

```
<?xml version="1.0" ?>
<!DOCTYPE theDoc SYSTEM "theDTD.dtd" [
  <!ENTITY % theDocList SYSTEM "theDocs.ent">
  %theDocList; ]>
<theDoc>
&doc1; &doc2;
</theDoc>
```

In the datacentric world

- information is stored as fragments
- documents are constructed dynamically
- XML software for this is (at last) beginning to appear

Virtual documents

Storage is a special kind of processing, like formatting, requiring a transformation in and out of some storage format. So we could

- store information in non-XML formats (optimized for specific functions, e.g. text retrieval or relational tables)
- recover all and only the information needed from the store in the form of a dynamically-generated XML document/fragment
- in an XML repository, access should be in XML terms

DBMS or XML?

Do you have to choose?

- The argument from history
  1. flatfiles gave way to network DBMS
  2. network DBMS gave way to relational
  3. will relational DBMS give way to XML databases?
- Getting the best of both worlds
  - DBMS are good at storing and managing *relations*
  - but equivalent XML technologies are rapidly maturing
  - and even traditional DBMS can be cajoled into presenting their contents in XML terms

Xquery

A new W3C recommendation (not yet stable) which combines

- ☛ a powerful expression language
- ☛ traditional SQL-like database facilities
- ☛ Xpath and XSLT

Recommended way of using the eXist XML DBMS

Xquery expressions

path expressions return a nodeset

element constructors return a new element

FLWOR expressions analogous to SQL Select statement

list expressions operations on lists or sets of values

conditional expressions traditional if then else construction

quantified expressions boolean operations over lists or sets of values

datatype expressions test datatypes of values

Path expression

The simplest kind of Xquery:

```
document("test.xml")//p
//p/foreign[@lang='lat']
//foreign[@lang='lat']/text()
```

Element constructor

May contain literal text or variables:

```
<latin>o tempora o mores</latin>
<latin>{$s}</latin>
```

FLWOR expressions

For - Let - Where - Order - Return

```
for  $t in //text
let $lats := $t//foreign[@lang='lat']
where count($lats) > 1
order by count($lats)
return
<latin>
{$lats}
<txt>{$t/@id}</txt>
</latin>
```

- `for` defines a *cursor* over an xpath
- `let` defines a name for the contents of an xpath
- `where` selects from the nodes as in SQL
- `order` sorts the results as in SQL
- `return` specifies the XML fragments to be constructed
- Curly braces are used for grouping, and define the scope of the for clause

Delivery strategies

- Our goal is fast and efficient access to any subtree of the docuverse, of any size
- Xpath has an adequately rich semantics
- XSLT has an adequately rich syntax
- XQuery offers all the programming features we need
- The rest is a Simple Matter of Programming...

Looking for words

eXist also has some useful text searching capabilities. For example,

```
//p &= 'fish dutch'
```

will find paragraphs containing both the words `fish` and `dutch` (in either order), and is rather easier to type than the equivalent xpath:

```
//p[contains(.,'fish') and contains(.,'dutch')]
```

You can also do a proximity search:

```
//p[near(.,'fish dutch',20)]
```

and stem matching:

```
//p &= 'fish*'
```