XSLT, continued

Sebastian Rahtz
February 2004



# Recap

We have met the following XSL basic controls:

```
<xsl:stylesheet>
<xsl:template match="...">
<xsl:apply-templates select="...">
<xsl:value-of select="...">
<xsl:text>
<xsl:choose>
```

... with the following 'extras'

```
<xsl:sort>
<xsl:number>
count()
sum()
XPath axes
qualified matches with []
```



# Modes

You can process the same elements in different ways using modes:

```
<xsl:template match="/">
 <xsl:apply-templates select=".//div" mode="toc"/>
 <xsl:apply-templates/>
</xsl:template>

<xsl:template match="div" mode="toc">
 Heading <xsl:value-of select="head"/>
</xsl:template>
```

This is a very useful technique when the same information is processed in different ways in different places.



# More functions

- `document`: (*string*)
- `generate-id`: (*string*)
- `name`: (*node*)
- `concat`: (*string*, *string*)
- `contains`: (*string*, *string*)
- `substring-before`: (*string*, *string*)
- `substring-after`: (*string*, *string*)
- `string-length`: (*string*)
- `translate`: (*node*,*string*, *string*)
- `normalize-space`: (*string*)

# String functions

```
<xsl:template match="hi[@rend='upper']">
 <xsl:value-of
   select="translate(.,'abcdefghijklmnopqrstuvwxzy',
   'ABCDEFGHIJKLMNOPQRSTUVWXZY')"/>
</xsl:template>

<xsl:template match="xref">
 <span style="color:red">
  (<xsl:value-of
   select="substring-before(@url,'://')"/> protocol)
 </span>
 <span style="color:green">
   <xsl:value-of
    select="substring-after(@url,'://')"/>
 </span>
</xsl:template>
```

(test4.xsl)



# Using `generate-id()`:

```
<xsl:template match="/">
    <p> <xsl:for-each select=".//p">
      <a href="#para-{generate-id()}">para
        <xsl:number level="any"/>, </a>
      </xsl:for-each>
    </p>
    <xsl:apply-templates/>
  </xsl:template>

  <xsl:template match="p">
    <p> <a name="para-{generate-id()}"/><xsl:apply-templates/></p>

  </xsl:template>
```

(test5.xsl)



# The `document` function

Using `document` to pull in another file and process it:

```
<xsl:template match="*">
    <xsl:copy><xsl:apply-templates/></xsl:copy>
</xsl:template>

<xsl:template match="xptr[@type='include']">
  <xsl:for-each select="document(@url)/*">
    <xsl:copy><xsl:apply-templates/></xsl:copy>
  </xsl:for-each>
</xsl:template>
```



# Named templates, parameters and variables

`<xsl:template name="...">`: define a named template
`<xsl:call-template>`: call a named template
`<xsl:param>`: specify a parameter in a template definition
`<xsl:with-param>`: specify a parameter when calling a template
`<xsl:variable name="...">`: define a variable

# Variables

```
<xsl:template match="p">
    <xsl:variable name="n">
      <xsl:number/>
    </xsl:variable>
    Paragraph <xsl:value-of select="$n"/>
    <a name="P{$n}"/>
       <xsl:apply-templates/>
  </xsl:template>
```

(test11.xsl)



# Named templates

```
  <xsl:template match="/div">
    <html>
     <xsl:call-template name="header">
      <xsl:with-param name="title" select="head"/>
     </xsl:call-template>
     <xsl:apply-templates/>
   </html>
  </xsl:template>

  <xsl:template name="header">
    <xsl:param name="title"/>
    <head>
      <title><xsl:value-of select="$title"/></title>
    </head>
  </xsl:template>
```

(test12.xsl)



# Top-level commands

`<xsl:import href="...">`: include a file of XSLT templates, overriding them as needed
`<xsl:include href="...">`: include a file of XSLT templates, but do not override them
`<xsl:output>`: specify output characteristics of this job



# Some useful `xsl:output` attributes

`method="`xml | html | text"
`encoding="`*string"*
`omit-xml-declation="`yes | no"
`doctype-public="`*string"*
`doctype-system="`*string"*
`indent="`yes | no"

# An identity transform

```
<xsl:output
    method="xml"
    indent="yes"
    encoding="iso-8859-1"
    doctype-system="teixlite.dtd"/>

 <xsl:template match="/">
    <xsl:copy-of select="."/>
 </xsl:template>
```

(test9.xsl)



# A near-identity transform

```
<xsl:template match="*|@*|processing-instruction()">
 <xsl:copy>
 <xsl:apply-templates
    select="*|@*|processing-instruction()|comment()|text()"/>

 </xsl:copy>
</xsl:template>

<xsl:template match="text()">
    <xsl:value-of select="."/>
</xsl:template>

<xsl:template match="p">
  <para><xsl:apply-templates/></para>
</xsl:template>
```

(test10.xsl)



# TEI Stylesheets

A library of stylesheets for transforming TEI documents to either HTML or XSL Formatting Objects, at http://www.tei-c.org/Stylesheets , with a form-filling interface for the HTML at http://www.tei-c.org/tei-bin/stylebear



# TEI Stylesheets (example)

An example of an importing stylesheet:

```
<xsl:stylesheet
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 version="1.0">
<xsl:import
 href="http://www.tei-c.org/Stylesheets/teihtml.xsl"/>
<xsl:param
   name="splitLevel">1</xsl:param>
<xsl:param
   name="numberHeadings"></xsl:param>
<xsl:param
   name="topNavigationPanel">true</xsl:param>
<xsl:param
   name="bottomNavigationPanel">true</xsl:param>
</xsl:stylesheet>
```

# What is XSLT good for?

Yes, it is useful for making web pages from TEI texts, but it is also a good tool for

- Selecting subsets of our texts for further processing
- Summarizing aspects of our texts
- Checking our text in ways that DTDs cannot
- Converting text into formats other than HTML or XSL FO

we can solve many of the problems with a small range of techniques.



# Finding any occurrence of an element

Very often, we will sit on the root element and process all the occurrences of a specific element by using the `descendant` axis:

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

   </body>
 </html>
</xsl:template>
```

here we use the `count` function. We continue to generate an HTML document to provide a convenient reporting format.



# Sorting occurrences of an element

Instead of just finding the elements, get them out in sorted order:

```
<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>
```

Notice here the `<sort>` child of `<apply-templates>` and the `level="any"` attribute for `<number>` which provides a count from the start of document regardless of depth.



# 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

# Intermediate storage and text output

If we need to gather the contents of an element and examine it, we use <xsl:variable>:

```
<xsl:template match="/">
      <xsl:apply-templates select=".//p"/>
</xsl:template>

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

All the text inside each <p> is stored in a variable called `contents`, which we can access as `$contents`.



# Finding my context

Often, we want to put a report on an element into context. Three common techniques are:

- Use <xsl:number level="any"> to give an absolute position within the document
- Locate the nearest (e.g.)  element and report its ID:

```
<xsl:value-of select="ancestor::div[1]/@id"/>
```

  (note the [1] qualifier, to get the nearest ancestor. div[last()] would get the farthest away
- Find the most recent element of some kind, regardless of nesting. This example finds the nearest <pb> element in the document before where we are now:

```
<xsl:apply-templates select="preceding::pb[1]"/>
```



# Lookup tables (1)

Elements may have categorisation attributes. Instead of saying

```
<condition>good</condition>
```

it is much more sensible to say

```
<condition target="c_1"/>
```

and have a lookup table in the <teiHeader>:

```
<taxonomy id="Condition">
<category id="c_1"><catDesc>excellent</catDesc></category>

<category id="c_2"><catDesc>good</catDesc></category>
<category id="c_3"><catDesc>reasonable</catDesc></category>

<category id="c_4"><catDesc>bad</catDesc></category>
<category id="c_5"><catDesc>very poor</catDesc></category>

</taxonomy>
```

so how do we retrieve those expansions?



# Lookup tables (2)

From

```
<condition target="c_1"/>
```

we want to generate the phrase *excellent condition*. A simple template would be:

```
<xsl:template match="condition">
<xsl:value-of
  select="//teiHeader//taxonomy/category[@id=current()/@target]"

condition
</xsl:template>
```

but this is inefficient, as every lookup with result in a quite serious tree traversal. Particularly expensive are XPath queries with noticeable use of //, especially at the start of the patttern.

# Lookup tables (3)

A better solution is XSLT *keys*, which let the processor maintain the lookup table for us. First we declare a key table, at the top of the stylesheet outside any templates:

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

This says that the processor must 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('CATS',@target)"/> condition
</xsl:template>
```

which is much more efficient, and more readable.



# Lookup tables (4)

The XSLT key tables can index *multiple elements* under one key. So

```
<xsl:key name="PTRS" match="ptr" use="@target"/>
```

will maintain an index of all `<ptr>` elements, with one entry per unique value of the target attributes. We could retrieve all the occasions when a `<ptr>` referred to the target 'Cats' with

```
<xsl:for-each select="key('PTRS','Cats')">
 There is a pointer to 'Cats' in the ptr inside section
 <xsl:value-of  select="ancestor::div[1]/head"/>
</xsl:for-each>
```



# Recursion

The use of named templates which call themselves is common practice in XSLT programming. The skeleton is

```
<xsl:template name="go">
   <!-- do something -->
   <xsl:if test="some condition">
     <xsl:call-template name="go"/>
   </xsl:if>
</xsl:template>
```

in which the templates keeps on calling itself until some release condition is met. You may also be passing parameters from one call of the template to the next.



# Recursion example 1

Can we track where a particular element occurs in the hierarchy?

```
<xsl:template match="emph">
   <xsl:message> * route for a emph to the top </xsl:message>

   <xsl:call-template name="where"/>
</xsl:template>

<xsl:template name="where">
  <xsl:message><xsl:value-of select="name()"/></xsl:message>

  <xsl:if test="parent::*">
    <xsl:for-each select="parent::*">
        <xsl:call-template name="where"/>
    </xsl:for-each>
  </xsl:if>
</xsl:template>
```

Notice the pattern `parent::*`, which translates as 'any element above me'.

# Recursion example 2

What if we need to pass parameters through a recursive template? The first call has a default value:

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



# Recursion example 2 (cont.)

And now the guts:

```
<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>
```



# Contextual validation

DTDs can only check fairly simple syntactic rules, and they have limited datatyping. You can write an XSLT script which has no output, but uses `<xsl:message>` to give diagnostics:

```
<xsl:template match="div">
    <xsl:choose>
    <xsl:when test="not(head) or head=”">
      <xsl:message>div <xsl:number level="any"/> has
       no head, or an empty head</xsl:message>
    </xsl:when>
    <xsl:when test="contains(head,'.')">
      <xsl:message> div <xsl:number level="any"/> has
       a head with a . in: <xsl:value-of select="head"/>
      </xsl:message>
    </xsl:when>
  </xsl:choose>
  </xsl:template>
```



# Schematron

Schematron is a complete XML validator written in XSLT, which can be used to check the semantics of XML documents; it is developed by Rick Jelliffe and can be used with any XSLT implementation.
It can be found at http://www.ascc.net/xml/resource/schematron/schematron.html.

# Converting to other formats

In our our earlier template, we listed the size of paragraphs:

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

Adding

```
<xsl:output method="text">
```

will produce pure text output which could be loaded into a spreadsheet or database.