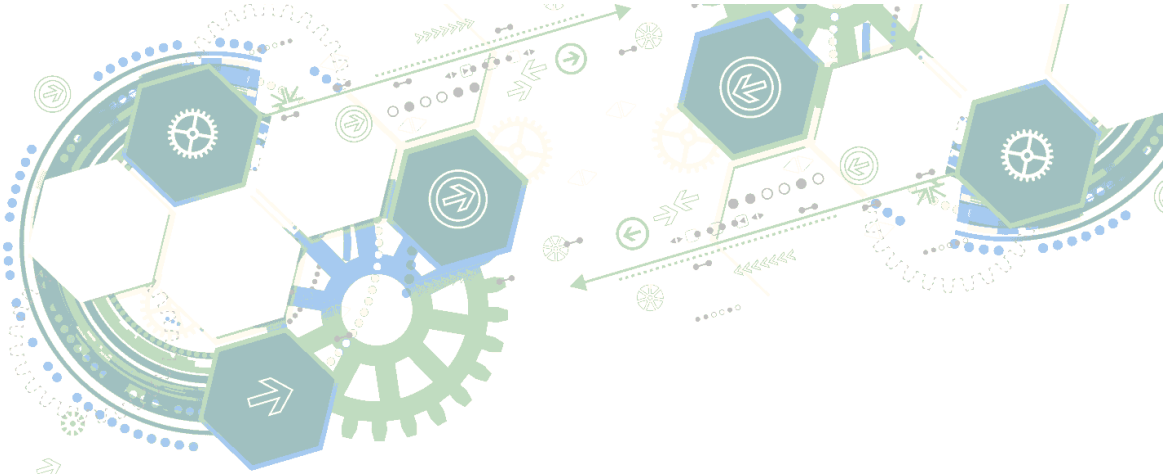


Part 1: Templates and Default Processing Rules.

A gentle introduction to stylesheets as adapted from the Women Writers Project and Further Adventures in XML

INTRODUCTION TO XSLT FOR TEI

LAT 0030/0130: Medieval Latin

April, 6 2016

Contents

Review of basic XML concepts

1.1 What is XML?

The eXtensible Markup Language (XML) is a markup language that defines a set of rules for encoding documents that is both human and machine-readable. That is, XML is a standard for constructing your tags that describes the *structure* of a document, rather than just its appearance. It is a simple, flexible and customizable.

In your career as a Digital Humanist most of the XML documents you will encounter will be accompanied by a set of rules, or *schema*, which generally describe how the tags are supposed to be used. In your class you are creating documents using epiDoc, which is really a subset of the larger Text Encoding Initiative (TEI) XML.

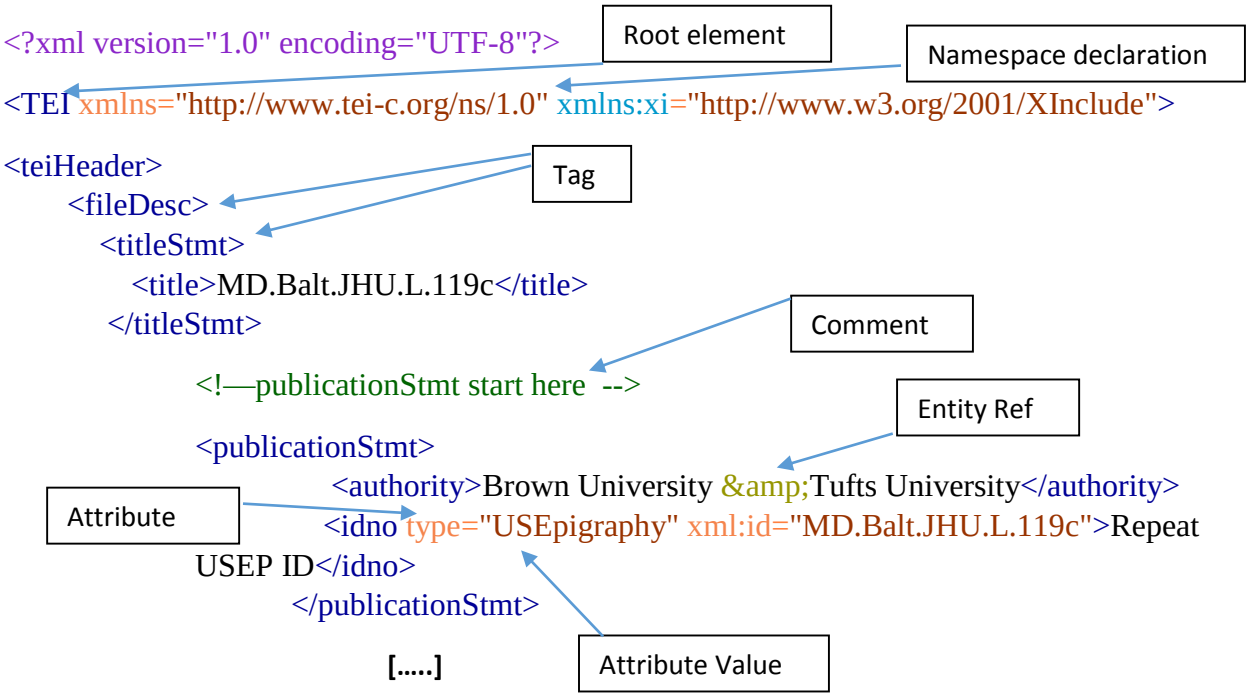
A markup language is a system for annotating a document that is syntactically distinguishable from the text.

If you haven't already, you may wish to acquaint yourself with the official schema. A good place to start is the guidelines, which can be found at: http://www.stoa.org/epidoc/gl/latest/index.html

EpiDoc XML can be processed and manipulated by any software that can interpret those rules. In this workshop we will be using oXygen to both look at XML, and transform it, with XSLT. XSLT is part of a family of standards for XML that also includes XPath and XQuery.

1.2 Basic structure & rules

The XSLT stylesheets we will create in this workshop are in fact XML documents, and need to follow the rules of "well-formedness" that apply to all XML documents. It is important to have a firm understanding of XML before moving onto XSLT. Let's review some of the basic structures of a fictional TEI xml:



The building blocks of XML are the *elements*. In its simplest form an element consists simply of an opening tag and a corresponding closing tag. An element can take one or more *attributes*.

```
<tag attribute-name= “attribute-value”>Some text</tag>

<supplied reason="undefined" evidence="previouseditor" cert="high">Ecce</supplied>
```

Elements and attribute names must follow these rules:

- They may contain letters, numbers, ideographs
- They may not contain white space, or punctuation other than hyphen, period, and underscore
- They must begin with a letter or underscore.

Finally it is important to remember that all XML, including your XSLT, must obey the basic rules of “well-formedness.” That is, every opening tag must have a closing tag. If you have empty tags you may use the tag along with a forward slash, i.e.:

```
<tag/>

<xsl:template match="date"/> ←Same concept in XSLT
```

Some other rules to remember:

- Tags must also nest cleanly—in XML and XSLT, poorly nested tags are forbidden.

  Correct:

  ```
  <book>
     <title>Collected fictions</title>
  </book>
  ```

  Incorrect

  ```
  <book>
     <title>Collected fictions<book>
   </title>
  ```

- Attribute values must appear within quotation marks. You can use either single, or double quotes, but once you choose, always be consistent.
- Tags are case sensitive, and must match.
- Every XML document must have a single root element (the outer wrapper). XML is frequently expressed as a tree like structure, and the single root element is crucial to this tree structure.
- The left angle bracket and ampersand are special characters: We use them to construct element tags and entity references respectively, and as a result we can’t use them in our XML content (PCDATA). Use their entity references instead: < and &

1.3 Processing XML

An XML processor is any program that can read and process XML documents. Examples include parsers, validity checkers, web browsers, etc… A parser is piece of software that reads the XML document on behalf of an application. Every piece of XML software has a parser. The parser reads the file, tokenizes it (breaks it into chunks) and interprets it. A validating parser checks for both “well-formedness,” but also conformance to a content model in the form of a schema and will indicate an error if the content model is violated. Needless to say, the program we will be using, oXygen, has a validating parser.

A valid XML document is "Well-Formed", and also conforms to the rules of a Document Type Definition (DTD) , XML Schema, or other type of schema, which defines the structure of an XML document, all of which define the structure of an XML document

oXygen always integrates the latest version of the Xerces-J XML parser to validate documents against XML Schemas.

1.4 Namespaces

An XML namespace is a collection of element and attribute names uniquely identified by an explicit or implicit association to a Universal Resource Identifier (usually in the form of a url). Namespaces are invoked to differentiate elements and attributes that have the same name, but different meanings and different origins.

Lets say we have the two documents, and they share two elements: date & id, which are used differently in each context. What if we want to integrate the information? In other words, how would we be able to distinguish between our elements?

books.xm

```
<book>
    <title>Collected fictions</title>
    <author>Jorge Luis Borges</author>
    <publisher>Penguin Books</publisher>
    <date>1999</date>
    <id>PQ7797.B635 A24 1998</id>
</book>
```

Namespaces allow us to disambiguate like things

authors.xml

```
<author>
    <title>Collected fictions</title>
    <first_name>Jorge</first_name>
    <last_name>Borges</last_name>
    <date>1899-1986</date>
    <id>79007035</id>
</authors>
```

As humans, we can obviously distinguish the difference based on context. A computer, and the processor that reads the stylesheets we write, cannot.

We create namespaces to explicitly differentiate between the two tag sets.

Our combined file might look like this:

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <authors xmlns="http://www.alex.org/authors/"
3.  xmlns:book="http://www.may.edu/books">
4.    <first_name>Jorge</first_name>
5.  <last_name>Borges</last_name>
6. <date>1899-1986</date>
7. <id>79007035</id>
8. <book:book>
9.    <book:title>Collected fictions</book:title>
10.       <book:date>1999</book:date>
11.      <book:id>PQ7797.B635 A24 1998</book:id>
12.    </book:book>
13. </authors>
```

Here are two namespace declarations. One is the default (the one without the prefix) and the other is explicitly prefixed

Although a namespace usually looks like a URL, that doesn't mean that one must be connected to the Internet to actually declare and use namespaces. Rather, the namespace is intended to serve as a virtual "container" for vocabulary and un-displayed content that can be shared in the Internet space. Typing the namespace URL in a browser doesn't mean it would show all the elements and attributes in that namespace; it's just a concept. So why is the URI usually in the form of a URL? The idea is that URLs are typically used because in webspace URLs are unique. But always remember that they are names, not locations! Namespaces are crucial to XSLT because they are what will allow us to distinguish XSLT instructions for the other elements we want to appear in our results document.

So what exactly is going on here?

1. "xmlns" is a reserved term that is used to bind elements (either implicitly, without a prefix, or explicitly, with a prefix) to a particular namepace
2. The xmlns declaration without the prefix (line 2) is what we call the *default* namespace, this means that all elements declared in the xml instance without a prefix belong implicitly to this namespace.
3. Line 3 is a regularly declared namespace in which the xmlns binds the prefix book to its namespace. Because this is explicitly defined, any element that is in that namespace must be qualified by that prefix, as in <book:book>,<book:titile>,<book:date> and so on.

XSLT also uses namespaces to distinguish between XSLT instructions (usually indicated by the prefix xsl:) and literal result elements, e.g., the <b> tags in the following example:

```
<xsl:template match="last_name">
<b><xsl:apply-templates/></b>
</xsl:template>
```

Getting started with XSLT

2.1 What is XSLT?

XSLT is a language for transforming the structure of an XML document. In the digital humanities world we usually want to restructure our XML for one of two reasons:

> **For presentation**: Remember that XML describes the structure of your document but not how it should be formatted. That is, while XML describes the content of document, for instance, how you should tag something if there is a missing letter from the document you are working on, it doesn't specify how you should display that piece of information. With XSLT, we can select the elements we want to present, how we want to present them, and then output them as a new XML, or xHTML, .KML, etc.. For instance, in the sample below we are outputting as xHTML.
>
> **For interchange**: As we develop distributed, cross-institutional digital projects, we need to be able to transfer our data from one system to another. XSLT can transform our data from one standard to another.

XSLT version 2.0 became a W3C specification in January 2007. When writing stylesheets it is important that you declare which version you are using. For this workshop, all of our stylesheets will be in XSLT 2.0.

```
<!-- Stylesheet version and namespaces: In terms of namespaces think of this as a way to disambiguate terms. The one you really need is the xpath-default-namespace -->

<xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xmlns:tei="http://www.tei-c.org/ns/1.0" xmlns:xhtml="http://www.w3.org/1999/xhtml"
   xmlns="http://www.w3.org/1999/xhtml" xpath-default-namespace="http://www.tei-
c.org/ns/1.0">

<!-- Output as xhtml -->
   <xsl:output method="xhtml" doctype-public="-//W3C/DTD XHTML 1.0 STRICT//EN"
      doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
encoding="UTF-8"
      indent="yes"/>
```

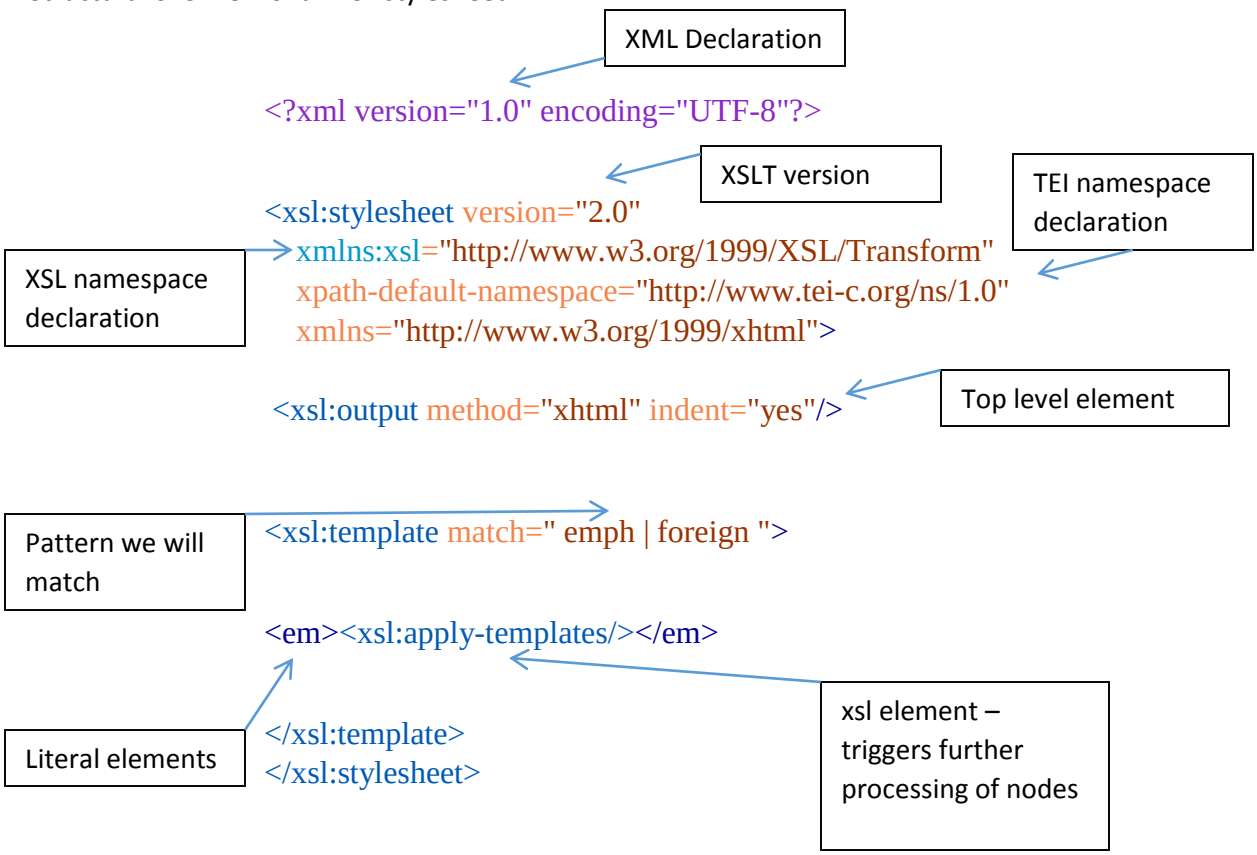
2.2 How XSLT works

XSLT is a declarative language in that you define the rules describing the desired transformation. The *stylesheet* is a document that contains these rules. The XSLT processor is a program that applies the stylesheet to the original XML document and performs the transformation. The original document is your input tree (or source tree) and the output is called the output tree (or results tree).

2.3 Structure of an XSLT Stylesheet

As already indicated, the XSLT stylesheet is itself a valid XML document and must follow the rules of "well-formedness." It contains special elements that are defined by XSLT, which provide the rules for the transformation, and may also contain other types of elements or formatting codes such as HTML. XSLT-defined elements belong to the XSLT *namespace* and are distinguished from other elements by the XSLT namespace prefix (usually **xsl:**).

Structural overview of an XSL stylesheet:



The processor reads the XML document as a *node tree* and matches these nodes against the rules in the stylesheet to produce the output document. These rules are called templates.

On the left is an abstract model of a template, and on the right is a very specific example of a template.

Abstract

```
<xsl:template match="pattern">

        [. . .Instructions about the
        transformation . . .]

</xsl:template>
```

Specific

```
<xsl:template match="lg">
  <html>
   <head>
    <title>a poem</title>
   </head>
   <body>
   <xsl:apply-templates/>
   </body>
  </html>
 </xsl:template>
```

Each template has two parts: a pattern that identifies the nodes in the XML document that the template can be applied to, and then the instructions about the actual transformation that should take place. When a node in the source tree matches a template's pattern, the content of that template is created in the results tree.

There are seven types of nodes in an XML document: **elements, attributes, text, comments, processing instructions, namespaces** and the **root node**.

Beginning at the top level, or ***root node*** of the XML document, the processor searches through the stylesheets for the appropriate template to apply. Templates can contain *literal elements* that can output verbatim into the results tree, and/or XSLT-defined elements that select or further process the nodes in the source document. Therefore, if an element starts with the XSLT namespace prefix (typically xsl:), it is not copied into the output, but instead provides instructions for selecting or modifying the XML data. Lets take a closer look at an example:

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

Where l identifies the node set in the input tree to which the template may be applied

Where and are literal element

Where <xsl:apply-templates/> is an xsl: element, which triggers further processing of nodes in the input tree

If the template contains XSLT elements that select additional nodes to process, the sequence of matching and template content outputting continues recursively until there are no nodes left to process.

 Do not confuse the ***root node*** with the *root element* (ie, the outermost element). The root node is an abstract point above the XML document that contains everything within it, including the root element!

Lets Try It Out!

In oXygen and from the Debugger view, open hello_world xml from the input tree/part 1 folder, and the hello_world.xslt from the stylesheets/part 1 folder, and then click the blue arrow. Go ahead and click the html button, followed by the blue arrow to see the same output as a webpage.

An abstracted view of your oXygen Debugger environment.

Input Tree

```
<?xml version="1.0"
encoding="UTF-8"?>

<paragraph>Hello,
world!</paragraph>
```

Stylesheet

```
<?xml version="1.0"
encoding="UTF-8"?>
<xsl:stylesheet
xmlns:xsl="http://www.w3.org/1
999/XSL/Transform"
                version="2.0">

  <xsl:template
match="paragraph">
    <html>
      <head>
        <title>A Short Test
Document</title>
      </head>
      <body>
        <div>
          <p>
            <xsl:apply-templates/>
          </p>
        </div>
      </body>
    </html>
  </xsl:template>

</xsl:stylesheet>
```

Output Tree

```
<html>
  <head>
    <meta http-
equiv="Content-Type"
content="text/html;
charset=UTF-8">
    <title>A Short Test
Document</title>
  </head>
  <body>
    <div>
      <p>Hello,
world!</p>
    </div>
  </body>
</html>
```

Go ahead and try playing with this template. Add a root element and call it text, then add another paragraph so your xml looks like this:

<text>

<paragraph>Hello world!</paragraph>

<paragraph>This is a silly test.</paragraph>

</text>

If you match on paragraph, and have selected html in your debugger view, your html output will only say: “Hello world!”, but if you switch to the text view in Debugger, it should look like this:

```
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>A Short Test Document</title>
  </head>
  <body>
    <div>
      <p>Hello, world!</p>
    </div>
  </body>
</html>
```

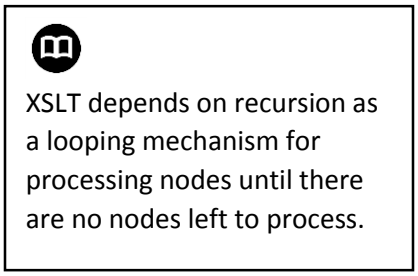
```
<html>
  <head>
    <title>A Short Test Document</title>
  </head>
  <body>
    <div>
      <p>This is a silly test.</p>
    </div>
  </body>
</html>
```

The processor did exactly what we wanted it to do when we wrote xsl:apply-templates It matched on the first paragraph, produced the literal output, found content and put it into the output tree, it went back, found another paragraph element, produced the literal output, found content and put it into the output tree. It is just that what we produced was XHTML, and our html view in oXygen’s Debugger suppressed the second output.

Lets try one more thing: instead of matching on paragraph, let’s match the root node, which is an abstract point above our XML document, and contains everything within it. “/ “matches the root node. See what happens. Note, this is how most XSLTs start, can you see why?

2.4 Default Rules

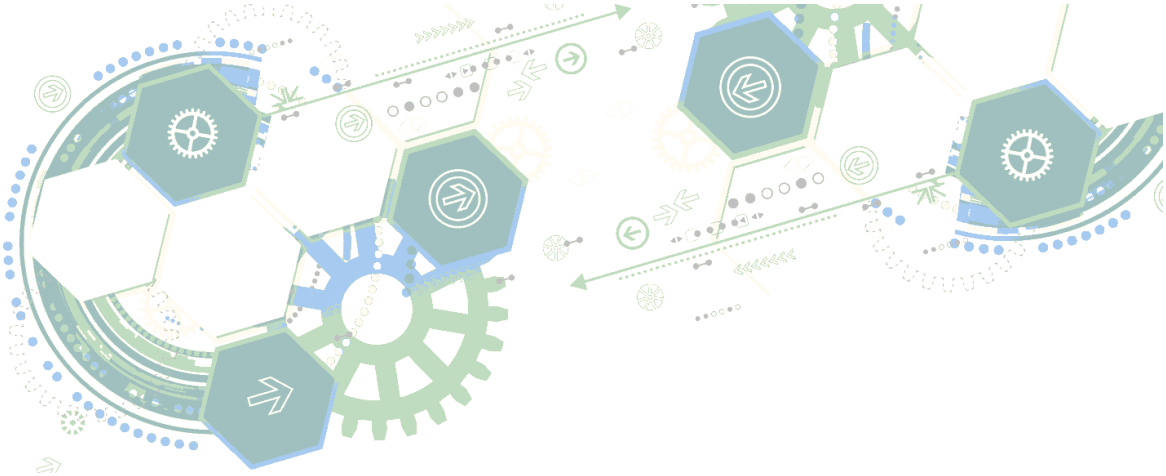
As indicated above, there are seven types of nodes in an XML document: elements, attributes, text, comments, processing instructions, namespaces and the root node. There are default rules to process each type of node in the absence of a more specific rule. Here is a quick summary:



- The default rule for the **root node** and all **element nodes** is to process all children (ensuring that the whole tree gets processed recursively, from the root, to the outermost branches.
- The default rule for attributes specifies that the *attribute value* is output into the results tree.
- The default rule for text nodes specifies that text is output into the results tree.
- The default rule for processing instructions and comments is that theses nodes are omitted from the results tree.

Essentially, this means that by default, all elements in the XML input tree, beginning with the root, are processed, and that text content is output into the resulting document, while instructions and comments are left out.

Any template rules you place in your stylesheet, will override these default behaviors. In XSLT, only the best matching rule will be applied to any given node, and more specific rules are better matches than general rules.



Part 2: X-Path and Working with XSLT.

A gentle introduction to stylesheets as adapted from the Women Writers Project and Further Adventures in XML

INTRODUCTION TO XSLT FOR TEI

LAT 0030/0130: Medieval Latin

April, 25 2016

Contents

X-Path and Working with XSLT

3.1 Common XSLT Elements

Although templates can be empty, most will contain one or more XSLT elements, also known as *XSLT instructions*, which describe how to select, sort, or further process nodes from our source tree. There are three XSLT instructions that do much of the heavy lifting:

xsl:apply-templates

> <xsl:apply-templates select = “node-set-expression”/>
>
> When this XSLT element appears in a template, it tells the processor to compare each node identified by the select attribute against the templates in the style sheet, if a match is found, output the template for the matched node.
>
> N.B. If the select is omitted, then all child elements, text, comments and processing instructions nodes of the context node should be processed. Attribute nodes will not be selected. (Remember your default processing rules!)

xsl:value-of

> <xsl:value-of select = “expression”/>
>
> When this XSLT element appears in a template, it tells the processor to compute the value of the selected node and copies into the output document. The item whose value is selected is relative to the current node (the item matched in the template).
>
> How the value of the node is determined depends on the type of node. The most common type of node is an element, and the value of an element node is all of the text between the element’s start tag and end tag, with markup and comments stripped out.

xsl:for-each

> <xsl:for-each select = “node-set-expresion>
>
> […]
>
> </xsl:for-each>
>
> This element appears inside a template and iterates through a set of nodes. The select attribute establishes a new context node set, which can be useful for numbering, formatting lists, and calling special functions.

In addition to the template elements, XSLT contains several top-level elements. As the name suggests, these appear at the top of the stylesheet, as children of the <xsl:stylesheet> element, rather than within templates.

xsl:output

> <xsl:output method = “xml” indent = “yes” encoding = “UTF-8”/>
>
> This element defines the characteristics of the output document.

method – specifies whether the output document is xml, html, or text.

indent – specifies whether the output is to be indented to highlight the hierarchical nesting of elements. This is a yes or no value.

output – declares an encoding standard for the output document. Mostly you will declare UTF-8.

xsl:strip-space and **xsl:preserve-space**

<xsl:strip-space elements = "element1 element2 …."/>

<xsl:preserve-space elements = "element1 element2 …."/>

<xsl:strip-space> defines elements for which non-significant whitespace (text nodes that contain nothing but spaces) should be removed. Conversely, <xsl:preserve-space> defines the elements for which white space should be preserved.

3.2 Navigating with X-Path

X-Path is a sophisticated language for marking locations and selecting sets of nodes with an XML document. The *match* and *select* patterns with XSL elements are X-Path expressions. They specify the location of the nodes we want to operate on, either with an *absolute path* that traces the location from the root node, or with a *relative path* that expresses the location in relation to the current or *context node.*

 An X-Path location is absolute if it begins with a slash ("/") and relative otherwise.

The following shows some simple X-Path syntax, and the most likely you will encounter or use in your career as a "DH'er":

Expression	Match
/	Matches the *root node*
.	Matches the *context node* (which could be an element, attribute, comment, etc.)
..	Matches the *parent* of the *context node*
*****	Matches any element node
@*	Matches any attribute node
@type	Matches the type attribute of the context node
text()	Matches any text node that's a child of the context node
comment()	Matches any comment node that's a child of the context node
paragraph	Matches <paragraph> child elements of the context node
emph | foreign	Matches <emph> or <foreign> child elements of the context node

Compound location paths

Expression	Match
/TEI/text/body/div/p	Our match starts with the root node, and then through the TEI element into the text, body, and div elements respectively, until it finally selects the <p> element children (this is an absolute path.)
div/p	Travels through the div element of the current node and selects any <p> element children of those nodes (this is a relative path.)
***/p**	Travels through all element children of the *context* node and selects any <p> elements of those notes
//p	Selects all <p> descendants of the root node

Predicates

You can use a predicate to filter the results from your node tests. Predicates are evaluated as either true or false, and only true predicates are selected for your result set. Predicates are expressed within square brackets [] and are appended to your node test.

 Sometimes it helps to think through your predicate with the statement: "Where it is true that…"

Expression	Match
//person[death]	Matches all <person> elements that have a <death> child element
//person[@role = "author"]	Matches all <person> elements that have a role attribute named "author"
//pubdate[.= "1911"]	Matches all <pubdate> elements whose value is 1911
person[2]	Matches the second <person> element in the *current context*
//person[not(death)]	Matches all person elements that do not have a <death> child element

For more complex expressions (selecting ancestors or siblings, for example) the unabbreviated X-Path syntax is required. An unabbreviated expression consists of an **axis**, a double colon **::** and the name/type of node[s] to be selected. Thus the following abbreviated expression:

<xsl:apply templates select= "div/p">

Would look like this in unabbreviated syntax:

<xsl:apply templates select= "child::div/child::p"/>

Where child is an axis that describes a position relative to the current node, and p is the name of an element node.

There are 13 X-Path axes in all, including child, parent, self, ancestor, descendent, preceding-sibling, and following-sibling. Here is a quick explanation of the most commonly used axes.

The child axis

This is the default axis when no axis is specified. It directs the XSLT processor down one-level in your document's hierarchy.

The parent axis

The parent axis is the exact opposite of the child axis. It directs the XSLT processor to look up one level in your document, and retrieve the node that the current context is contained within. N.B. The root node does not have a parent, so the parent axis does not apply. Every other node in the tree will have exactly one parent.

The descendant axis

The child axis only directs your processor down one level. If you want to find the child of a child (and so on), you can use the descendant axis to direct your processor down, as far as the source document tree will allow, from the current context. If you are at the root node, the descendant axis will return all element and text nodes in your source document.

The ancestor axis

This is the opposite of the descendant axis. The ancestor axis directs your processor up through your document tree from the current context as far as the source document structure will allow. It will retrieve all parental nodes along the way.

The self axis

The self axis permits the processor to see only the current context

The following-sibling axis and **The preceding-sibling axis**

All the children of any given node (the parent) are siblings. If your current context is one of these siblings, you can examine siblings that come before you in the document structure (the preceding-sibling axis) or siblings that come after you in the document structure (the following-sibling axis).

3.3 XSLT Functions

Functions serve a wide range of purposes in XSLT. They can allow you to manipulate strings, perform mathematical operations, to count and assign numbers, and generate unique IDs. In this section we will look at a few of the most used functions that will help you perform some simple analysis of your text.

A more comprehensive list of functions can be found at the W3C schools site: http://www.w3schools.com/xsl/xsl_functions.asp

Count(node-set) This function counts the number of nodes in a given node-set. You should note that the value of the count() *is specific to the axis you are traveling along from the current context*. For example, if I wanted to count the number of people listed in a TEI personography, I could use the following:

```
<xsl:value-of select="count( //person )"/>
```

<xsl:value-of select="count(*)"/> will return the number of immediate child nodes (the default axis) of the current node.
<xsl:value-of select="count(descendant::*)"/> will return the total number of descendant nodes.

last()

The last function is somewhat like the count() function, except it doesn't require an argument. last() returns the number of nodes in the current context, so its value is equal to the position of the last node. last() is often used as a predicate. For example, to retrieve the last person in our personography, we can do this:

```
<xsl:value-of select="//person[last()]"/>
```

position()

The position() function returns the numeric position of the context node within the current node-set. Combined with the last() function, position() will allow you to map your document's structure.

```
<xsl:for-each select="//person">
      <p><xsl:value-of select="persName/forename"/> <xsl:text> </xsl:text><xsl:value-of select="persName/surname"/> is number <xsl:value-of select="position()"/> of <xsl:value-of select="last()"/></p>
</xsl:for-each>
```

This example is also a good example of how to use <xsl:for-each> to step through a set of nodes and process each one in turn. Because <xsl:for-each> changes the current context, it is especially effective when used in conjunction with functions like postion() and last(). When you are testing the position of a node within a group of related nodes, scooping them all up in an <xsl:for-each> seems more logical, and makes it easier to maintain and read the stylesheet.

contains(string1,string2)

This Boolean function test to see if string1 contains string2. If so, it evaluates as true. To see the records in the personography that mention Stuart, we could do this:

```
<xsl:for-each select="//person[contains(persName/surname,'Stuart')]">
       <li><xsl:value-of select="."/></li>
</xsl:for-each>
```

Note, that strings are case sensitive. Also note that the first argument in this case lacks the quotation marks that would identify it as a string; it is therefore an X-Path expression referring to the element <surname>.

substring-after(string,string)
substring-before(string,string)

The substring-after and the substring before functions act in a similar manner but in opposite directions.

Two arguments are required: a string to act upon, and the character(s) that will trigger the substring function. If you are using a string literal as the second argument, make sure you enclose it within single quotes. To see the records in the personography that mention Stuart, we could do this, add the date, and only show the year:

```
<xsl:for-each select="//person[contains(persName/surname,'Stuart')]">
        <li><xsl:value-of select="."/><xsl:text>(</xsl:text><xsl:value-of select="substring-
before(birth/@when,'-')"/><xsl:text> - </xsl:text><xsl:value-of select="substring-
before(death/@when,'-')"/><xsl:text>)</xsl:text></li>
</xsl:for-each>
```

All we have done is extended the original contains function with a substring-before function that converts the attribute values for birth/@when and death/@when to a string, looks for a substring (the first hyphen '-' and returns everything that occurs before the substring. This effectively strips off the month and date from our attribute values, returning only the year.

normalize-space(string)

This function takes the string specified in the argument and removes insignificant whitespace. Specifically, all leading whitespaces are removed, all trailing whitespaces are removed and multiple whitespaces within the string are converted to a single whitespace.

<xsl:value-of select= “normalize-space(‘ Hello, World! ‘)”/> → “Hello, World!”

3.4 XSLT Conditionals

Sometimes you need to test some condition of a node—its value, its position, the existence of particular ancestors or children—before you can specify how it should be treated.

The <xsl:if> statement is a simple construct for testing certain conditions. It looks like this:

<xsl:if test= “some expression”>

    [further instructions]

</xsl:if>

The expression being tested is evaluated and then converted to a Boolean value. If it evaluates true, the instructions with the <xsl:if> elements will be processed; if the test is false, those instructions are ignored. To create more complex conditional statements, XSLT provides the <xsl:choose> element.

```
<xsl:choose>
        <xsl:when test=”expression”>          ← <xsl:when> is repeatable
                [further instructions]
        </xsl:when>
        <xsl:otherwise>                       ← <xsl:otherwise> is optional
                [other instructions]
        </xsl:otherwise>
</xsl:choose>
```

Each <xsl:choose> element contains at least one <xsl:when< statement, which follows the same syntax as the <xsl:if> statements. The processor evaluates each <xsl:when> statement in order, looking for one that evaluates to true. As soon as a true value is found, the instructions inside that <xsl:when> statement are executed and the <xsl:choose> is not processed any further. If none of the <xsl:when> statements are true, the <xsl:otherwise> statement will be executed if found; if not, nothing happens.

3.5 Variables

XSL variables are not actually variable in the sense that once a value is assigned, it is constant, and can be invoked later. Variables can be defined at the top of an XSL stylesheet, giving them global scope, or defined within a template, which limits it to that specific template.

Once you have declared a variable you can retrieve its value by prefixing its name with a dollar sign ($).

Below is a declared variable called supplied_lost in which we invoked the count() function on an X-Path that is looking for the supplied element with an attribute reason, whose value is lost.

```
<xsl:variable name="supplied_lost" select="count(//supplied[@reason='lost'])"/>
```

The variable can now be invoked with $supplied_lost anywhere in our document, like so:

```
<p>Of those, <xsl:value-of select="$supplied_lost"/> are lost elements;</p>
```