CENTRE FOR COMPUTING IN THE HUMANITIES
King's College London

# An XSLT Tutorial

**Introducing XSLT in a Humanities Context**

**Spring Semester**
**2007-2008**

**Tutors: Paul Spence and Elena Pierazzo**
**Course notes written by John Bradley and edited by Paul Spence and Elena Pierazzo**

# Table of Content

Chapter 1

# Basics of XSLT: Transformation, Templates and HTML

## 1. Introduction

It should be pretty clear that a document entitled "An XSLT Tutorial" has the primary purpose of introducing you to (the basics of) XSLT! There are several excellent resources in print already that introduce people to XSLT. Why do we need another one? One possible reason is that we are using these notes in the context of courses in Humanities Computing. If you are one of our students it is likely that your background is different from that assumed for most of the other XSLT tutorials you will find. We try to accommodate this difference in these notes. Furthermore, in several ways the materials that we use to illustrate points about XSLT here are almost entirely drawn from humanities-like sources, and many are actual examples that we have come up against in our daily work. Materials from humanities-like sources are the kind of things those interested in Humanities Computing are likely to come up against, and it is useful to see how XSLT works with these kinds of materials.

Before we really begin, however, it is important for you to understand that, although introducing you to XSLT with examples drawn from humanities sources *is* the primary purpose of this tutorial, the writer has other agendas as well -- and, furthermore, does not want them to be hidden from you:

- The writer's second agenda is tied up with the fact that XSLT is a technology that is designed to work with XML markup. We believe that the way XSLT makes you think about XML will also cause you to develop a better understanding of the significance of *XML markup itself*, and will help you begin to see how this markup might be more fully exploited. Hence, during this tutorial you might expect to see features of XML that you have perhaps not seen described, or only briefly touched on before. Also, you might well be shown uses for XML that are significantly different from what you have thought about up to now.
- There is a third agenda as well. The CCH applies its technologies to research projects in the Humanities, and thus you will also see along the way how XML and XSLT are used in several humanities-oriented research projects in which the CCH is a partner. We believe that technologies such as XML and XSLT have a great deal to offer to the humanities, and it is in the full understanding of these technologies, and their implication to scholarship, that humanists can gain the most benefit.
- Finally, there is a fourth agenda. You will also be briefly exposed to the kind of "formal" approach to defining XSLT that is commonly used in computer science to describe things called *formal languages*. There could be a great deal more said formally about formal languages, so you should consider that you are getting only a taste of the significance of this here.

Finally, before we really begin, I must give you a different sort of warning. Almost all of our material (indeed, one of the principal uses of XSLT world-wide) makes use of XSLT to generate HTML. It turns out that XSLT will require you to understand HTML tagging in ways that high-level tools that "do" HTML for you such as Dreamweaver or Word don't show. Indeed, all of the exercises we have provided in this tutorial will expect you to know HTML well enough that you could first create at least basic HTML pages with text, lists, headings, links and tables with a non-HTML-aware editor such as Notepad. If your only experience of HTML has been to generate it using these high-level tools then you will find that your understanding of HTML will not be not enough to allow you to deal with the material covered here. Before proceeding further with XSLT be sure that you develop a real hands-on, low level, understanding of HTML.

## 2. What is XSLT?

The name *XSLT* stands for "Extensible Stylesheet Language: Transformations" (compare with *XML*: "Extensible Markup Language"). It is part one of a pair of standards that, together, are named *XSL* (Extensible Stylesheet Language). The second part of XSL, *XSLFO* -- "Extensible Stylesheet Language: Formatting Objects -- may be introduced very briefly at some point in your course, but is not described in this tutorial. XSLFO provides a way to describe in XML markup how to lay out text for printing on paper.

### 2.1 XSLT as a "W3C Standard"

XSLT is one of the many *standards* developed by the World Wide Web Consortium (W3C) (http://www.w3c.org), a group of individuals and corporations who maintain and develop old and new standards for the World Wide Web. When The World Wide Web Consortium was introduced the WWW was already well established on the Internet, but the various components that made it up (HTTP, HTML, etc.) had begun to fragment into many different dialects -- the browser developed by one company "X" would support a different set of HTML tags than that developed by company "Y". The fear then arose that by breaking the WWW user community into groups by browser, that the company with the dominant browser would, in the end, dominate the WWW as well. Since evidence from the past was that the internet, and computing in general, developed best in the context of openness, it seemed best to try to encourage different developers to develop WWW tools such as browsers and servers that all "inter-operated" -- that allowed any user to view a webpage with a browser of their choice rather than having to use a particular product.

The W3C cannot act like a World Wide Web legislature -- able to establish "WWW laws" that compel software vendors to produce compatible products. Hence, the challenge has not been entirely met, and problems of incompatibility between different browsers continue to haunt the WWW. However, the W3C has had some *significant* successes in promoting interoperability, and the W3C has achieved most of this by developing documents called *standards documents* that describe, in a formal manner, what WWW-based things such as HTML or HTTP look like and how their constituent parts are supposed to behave. A W3C *standard* is developed in a relatively open atmosphere by the very individuals and corporations who are already active in developing software for the WWW. A standard is, in the end, described to the world in *standards documents* that fully describe how the standard operates, and are published freely on the W3C's website. The hope is that major developers for software on the WWW will then create tools that will implement this standard. By having many different developers all creating tools built around the same tool definition, the tools will be largely compatible.

It was an W3C initiative that resulted in the development of XML in the first place. Having first defined XML, the organisation has subsequently gone on to develop a large number of XML-related standards, of which XSLT is one.

A W3C standard goes through a development process that results in a definition that is, at least for any particular version, fixed. Generally, however, new features need to be added at a later date. Thus, the W3C allows for each standard to be given a *version number*, and allows later versions to be developed that add new features to the older version, but remain compatible with it. We will be working here with *XSLT Version 1.0*, but the W3C is currently working on a version 2.0 that is almost done. 2.0 has some new features (including things that we would find useful in this course if they were available), but it is expected that pretty well everything that *is* in Version 1.0 will remain the same in Version 2.0.

The development of XSLT's Version 1.0 standard was a long complex process, and before it got there there were earlier versions (sometimes given Version numbers below 1.0) that were made available for comment and testing. These preliminary versions deviated to some extent from the final version. XSLT grew originally out of a set of proposals from Microsoft, and Microsoft built prototype XSLT transformation software into their Internet Explorer browser that was, first, based on their original proposal, and then on an interim proposal of the XSLT standard. Neither of these versions correspond exactly to the standard as it eventually was defined, although Microsoft has introduced a properly compliant XSLT processor in recent browser software. The reader should be aware, however, that many of the older tutorials available on the WWW that are intended to introduce you to XSLT -- particularly those that focus XSLT within Microsoft's browser -- describe it in terms of one of these earlier, now outdated, versions.

We have mentioned that W3C publishes its standards as freely available documents on the WWW. We also mentioned earlier that the original XSL standard was split in time into two pieces; one of which was XSLT. Well, it turns out that some time ago XSLT was actually *further* split into two related standards, called *XSLT* and *XPath*. You can see the standards documents from W3C that describe them at:

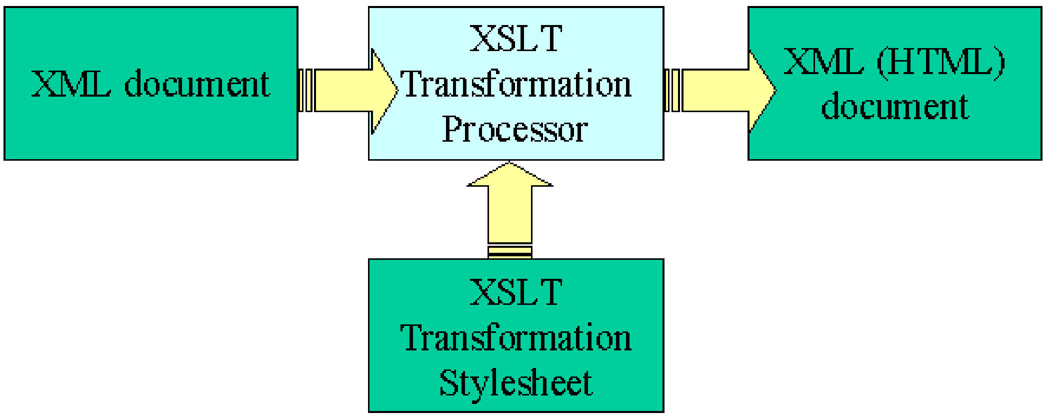
- XSLT Version 1.0: `http://www.w3.org/TR/1999/REC-xslt-19991116`
- XPath Version 1.0: `http://www.w3.org/TR/1999/REC-xpath-19991116`

Although these documents fully describe both how XSLT and XPath components "look" (their *syntax*) and what these components "mean" (their *semantics*), they are often described in a language that is difficult to understand at first. In this course we will not have time to introduce you fully to XSLT, so we hope that by the end you will be in a better position to do your own research into more advanced uses of XSLT for processing XML documents.

One option is to to use the XSLT specification itself as one of the sources of information about how to do other things in XSLT that we haven't introduced. It is by no means essential to use this somewhat technical document as a reference, but we would nevertheless encourage you to at least take a look at such standards documents, and to this end we have provided you with references to them in this document. Although at first you may find the reading of the standard hard going, as you work more with XSLT you will find that it becomes more understandable.

## 2.2 *Transformations* in XSLT

We have already mentioned that the "T" in XSLT represents the word "transformation". XSLT, then, provides a language to describe a transformation that can be applied to a set of (input) XML documents to generate a new set of transformed outputted documents. When one uses XSLT one has two XML DTD's in mind -- the DTD that describes the *source* document, and the DTD that describes the *output* document. One writes an XSLT document to describe how an XSLT processor should transform the text and markup in the source document into the desired output format. The process is shown diagrammatically:



Here one can see three documents: the original XML document on the left, an XSLT Transformation specification (given in what is called an *XSLT Transformation Stylesheet* document) at the bottom, and the resultant XML document being generated by the XSLT Transformation processor shown on the right.

Note that HTML is shown above as a possible kind of output from an XSLT processor. Now you probably already know that traditional HTML, (by this I mean the pre-XHTML version commonly known as HTML 4.0), is not an XML markup language, since it doesn't require one to follow the various rules that apply to XML markup. Under these circumstances, lumping HTML in with other XML documents may seem a little strange. We will discuss later in this chapter the implications of making HTML one of the languages that XSLT can generate. For now, however, it is useful to realise that one of XSLT's major uses is to transform XML documents into HTML. This is because XML is becoming a scheme in which more and more documents are being created and many if not most of these documents are meant to be viewed ultimately on the WWW. Since many WWW users continue to use browsers which do not support XML markup but only understand HTML, it usually necessary to be able to take these XML documents and automatically produce HTML from them. When we solve the problem of expressing HTML as an XML markup scheme then we can still use XSLT to perform this important task.

It turns out that almost all of the examples we will be showing in this tutorial will show XSLT transformations that map XML documents into HTML. It becomes natural, then to see *only* this use for XSLT. We will, however, examine a few cases late in the tutorial where the product of an XSLT transformation is *not* HTML, and will discuss why this other transformation would be useful.

## 2.3 A sample Transformation

Here is an example of a simple transformation (into HTML!) of the kind that XSLT handles easily:

```
<story>
<title>In which Eeyore loses a
Tail and Pooh Finds One</title>
<para>Owl lived at the Chestnuts,
an old-world residence of great
charm, which was grander than
anybody else's, or seemed so to
Bear, because it had both a
knocker <emp>and</emp> a bell-
pull.  Underneath the knocker
there was a notice which said:
<quote>
PLES RING IF AN RNSWER IS REQIRD.
</quote>
</para>
</story>
```

```
<html>
<head>
<title>Winnie-the-Pooh</title>
</head>
<body>
<h3 align="center">In which
Eeyore loses a Tail and Pooh
Finds One</h3>
<hr>
<p>Owl lived at the Chestnuts, an
old-world residence of great
charm, which was grander than
anybody else's, or seemed so to
Bear, because it had both a
knocker <i>and</i> a bell-pull.
Underneath the knocker there was
a notice which said:
<blockquote>
PLES RING IF AN RNSWER IS REQIRD.
</blockquote>
</p>
</body>
</html>
```

The XML document is shown on the left, the transformed HTML document on the right. You might already have noticed that the XML document is shown without specifying a DTD, but that it follows all the basic rules of XML (that all opening tags must be matched with closing tags, that tags must be properly nested, etc.). XML documents that follow these rules (and a few others we don't describe in detail here), even if they do not have a DTD, are called *well-formed*. All that an XSLT processor requires of an XML document is that it be well-formed. A DTD is not needed, and for most of our examples in this course you will find that a DTD has not been provided.

Now let us take a closer look at the nature of the transformation that might be made between the XML document, and an HTML rendition of it. We can see certain characteristics.

- Some of the elements can be transformed simply by substituting a single HTML tag for the XML one -- for example, the XML `<para>` tag (shown in the online version of this document in blue, above) is rendered in HTML using the `<p>` element. Similarly, the XML `<emp>` element (shown in pink) is transformed into an HTML `<i>` element.
- Some of the transformations, however, are more complex. The XML `<title>` element, for example (shown in green), is transformed so that its context is placed within an HTML `<h3>` element, and then an HTML `<hr>` element is placed after it. The transformation for the XML `<story>` element is more complex still. Since the `story` element surrounds the entire document, it must be transformed into the various pieces that make up a full HTML document, including the `<html>` element itself, and its continuing `<head>` and `<body>` elements.

Note, that although sometimes the transformation between the XML and HTML is simple (a one-for-one substitution of tags) and in other situations it is more complex, nonetheless, the XML to HTML transformation shown here can be characterised as the task of inserting before each XML element's content some HTML (and/or text), and inserting after the content some (different) HTML (and/or text). The description of what this insertion is supposed to be is the information that goes into an XSLT transformation stylesheet.

The task of describing this transformation from the XML into the HTML involves specifying a series of rules that specify what to do when a particular kind of XML element is found. In each case the rule has to specify what to do when the element starts, when it ends, and to specify what should be done with the element's contents.

## 3. Basics of an XSLT stylesheet

Here is a rudimentary *XSLT transformation stylesheet* document that describes the transformation on the little eeyore XML into HTML.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
<xsl:output method="html"/>
<xsl:template match="story">
<html>
<head>
<title>Winnie-the-Pooh</title>
</head>
<body>
<xsl:apply-templates/>
</body>
</html>
</xsl:template>
<xsl:template match="title">
<h3 align="center"><xsl:apply-templates/></h3>
<hr/>
</xsl:template>
<xsl:template match="para">
<p><xsl:apply-templates/></p>
</xsl:template>
<xsl:template match="emp">
<i><xsl:apply-templates/></i>
</xsl:template>
<xsl:template match="quote">
<blockquote><xsl:apply-templates/></blockquote>
</xsl:template>
</xsl:stylesheet>
```

Several things should be noted about this XSLT transformation document:

- It is, itself, a well-formed XML document -- using elements, attributes and element contents to describe the transformation. It turns out that the fact that an XSLT stylesheet is, itself, an XML document has several implications for how transformations are expressed that we will discuss shortly.
- It contains elements from two different tag-sets (or, if you like, "DTDs") mingled together. Some of the elements (such as `<html>` or <h3>) are elements defined in HTML itself. whereas some of them are defined in the XSLT specification. All the XSLT-defined elements shown here begin `xsl:`. To help you see them, we have shown the XSLT tags in bold, although, of course, they would not appear as bold in a real stylesheet.

  Mixing of tagsets together like this is facilitated by means of an XML convention called *namespaces*. The prefix notation `xsl:` is a part of this namespace convention. We will discuss it further below.
- The basic structure of this simple stylesheet, after the initial startup tag, and the XSLT `xsl:output` element, is a series of rules written formally which essentially say what we informally said were needed to do the transformation above. The rules are called *templates* in XSLT parlance. Each template element defines the form of some part of the output document that is to be generated whenever the input document's structure matches the given pattern. For example, the first template element begins

"`<xsl:template match="title">`", and this element start tag, with its match attribute, should be read as saying "I contain a definition of what is to be generated in the output document whenever the XSLT processor finds a 'title' element."

- There are a number of different ways to specify what should appear in the output document when a particular template matches the material in an input XML document. The transformations shown here are of the most straightforward type. Here, the contents of each template element is mainly a set of HTML elements.
- Inside each template element in our example is one occurrence of another XSLT element -- the empty `<xsl:apply-templates/>` element. The apply-templates element marks the place where the contents of the template's input element (after it is suitably transformed by other template element specifications, if any) should be inserted. In effect, then, in this stylesheet, the `xsl:apply-templates/` element marks the division between the material that is to appear in the output document *before* the input element's content, and *after* it.

  For example, look at the template for the "title" element:

  ```
  <xsl:template match="title">
  <h3 align="center"><xsl:apply-templates/></h3>
  <hr/>
  </xsl:template>
  ```

  and compare this with the XML input and HTML output shown earlier. The HTML tags for the `h3` element surround the `xsl:apply-templates` element in the template, since the contents of the XML title tag is meant to be surrounded by the H3 when the HTML is generated. The HTML `hr` tag follows the closing `h3` tag since that is where we want it in the HTML output.
- Note that in the `xsl:template` shown above, the HTML tag, usually written in a conventional HTML document as `<hr>`, is shown here as an XML empty tag: `<hr/>`. This is because it is appearing in the middle of an XSLT stylesheet, which is, itself, an XML document. Since it appears here, in the stylesheet, as a part of an XML document it must follow XML rules: and in this case the rule that empty elements in XML must be marked with an ending slash is the rule involved. We will discuss this further below.
- Note as well that the ordering of the template elements in the XSLT is not significant -- an XSLT processor would produce exactly the same result if the stylesheet contained the template element for *quote* elements first rather than last. Each rule is described independently of any other rule. With this XSLT transformation stylesheet, the ordering of the output HTML document is determined not by the ordering of the templates in the XSLT file, but instead *entirely* by the ordering of the elements in the input XML document.

## 4. Using an XSLT Processor

### 4.1 Which to use?

There are a number of freely available XSLT processors available. We will give you information in class about which to use and how to use them, but two of the most commonly used in our community are *Saxon* and *XALAN*. Be aware that setting up an XSLT processor to work on your own machine is a far from trivial process, and that it may be easier to use PAWS machines in the manner we will demonstrate in class when carrying out exercises or doing the assessed exercise.

### 4.2 Error messages

Error messages in XSLT processors can be tricky to resolve. If you find it difficult to decode the error message, look at the line number and positional information given by the processor. If there are many error messages, concentrate on the first one, as dealing with this may solve a lot of other problems.

Common problems:

Missing closing tags Mistyped tags Forgetting to mark an empty element Forgetting to surround values of attributes with double quotes

### 4.3 XML/XSLT editors

Remember that all XML files (including XSLT files) are just plain text files with a special extension. This means that you can edit them in any text editor in theory. It is not advisable to edit XML files in programs like Microsoft Word, but it is quite common to do so in tools like *Notepad*. The author of these course materials has used this method many times in the past.

While there is nothing wrong with editing XSLT using a plain text editor, it is much harder work, and it is easy to mis-type code in such a manner that the transformation process fails. For this reason, we introduce you to a simple but very powerful XML/XSLT editor called *Oxygen* that has many features (like automatic tag completion and code-checking) that will help you produce well-formed and valid code.

## 5. A more Formal Introduction to XSLT

We have, so far, introduced you to XSLT informally by showing you an example. However, XSLT is what computer scientists call a *formal language*, by which they mean in part that it has a formally defined syntax (or set of rules that must be applied when the language is written down) and a set of formally defined semantics (by which they mean that the meaning associated with each element of syntax can be describe in such a way that it suitable to be implemented on a computer). Here, then, we will examine the format of an XSLT stylesheet somewhat more formally, explaining a bit about how the formal language behind it works.

### 5.1 Namespaces

Before we begin looking at a stylesheet from the beginning there is an important issue of *Namespaces* to discuss. We have already mentioned that an XSLT stylesheet will contain tags from at least two different 'structures' (we can think of these as two DTDs) -- one is the XSLT tagset, and the other is for the tagset used in the output document -- say, HTML. It is necessary to distinguish between the origins of the two tagsets, so that for every tag the XSLT processor knows: whether it is to interpret it as *an XSLT instruction* or to interpret it as a *tag to be put in the output* document.

This is done by using the XML concept called "*Namespaces*". Note that Namespaces are not just a part of XSLT stylesheet documents -- but can be used in any XML context where tags from more than one origin are to be combined together. There is more information about Namespaces at **[XSLT §2.1]**

W3C describes a Namespace as a collection of tags from a single *schema* (where by the word *schema* they mean something very like a DTD). When elements from several namespaces are being used in a single XML document it is necessary to:

1. Identify the namespaces that are being used in the document
2. For each element (or attribute), make it possible to establish from which namespace it comes.

If all the tags from a document comes from a single DTD then there is no need to explicitly specify the namespace. If, however, more than one namespace is used in a single document, then at all but one of them must be specifically identified. In an XSLT stylesheet, for example, the opening tag will look something like this:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
```

The "xmlns:xsl" attribute here asserts that tags beginning "`xsl`" all belong to the namespace that has the URI id of "`http://www.w3.org/1999/XSL/Transform`". The URI given here is always used to identify the *XSLT namespace*. In effect, then, this tag announces that any tag beginning "xsl:" in the rest of this XSLT document comes from the XSLT tagset. In an XSLT document the namespace for the XSLT tagset itself must always be *explicitly specified* using this URI in this way in the first tag.

In the rest of the stylesheet document the part of an XML tag that identifies its namespace is called the namespace prefix. It is conventional to use "`xsl`" as the prefix for XSLT tags -- but this is not essential. The following XSLT transformation uses "`trans`" as the prefix for the XSLT namespace instead and will work just as well:

```
<trans:stylesheet xmlns:trans="http://www.w3.org/1999/XSL/Transform"
version="1.0">
<trans:output method="html"/>
<trans:template match="story">
<html>
<head>
<title>Winnie-the-Pooh</title>
</head>
<body>
<trans:apply-templates/>
</body>
</html>
</trans:template>
[...]
```

Here, the initial tag contains an `xmlns:trans` attribute (rather than `xmlns:xsl`), indicating that tags from one of the namespaces that this document contains will be identified with the "trans" prefix. The `xmlns:trans` attribute points to the same XSLT URL identifier (shown here in italics, for visual emphasis only) that we used above. Since it is the URI for the XSLT specification, it has the effect of asserting that the "trans" prefix will identify XSLT tags.

Although we have just shown that it is possible to identify XSLT elements using a namespace prefix other than `xsl:`, this is rarely done, and in the rest of these teaching materials, we will use `xsl:` to mark all XSLT elements.

## 5.2 The XML Declaration

Now that we have explained namespaces, let us begin to look at an XSLT stylesheet in detail, beginning at the beginning. We'll touch briefly on namespaces again in a moment.

Since an XSLT stylesheet is an XML document it may have an XML declaration at its beginning. Often, the following declaration is useful:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

The declaration asserts that the XSLT stylesheet corresponds to Version 1.0 of the XML specification. The *encoding* attribute is useful, since it asserts that the character representation used in the document is the *ISO-8859-1* standard -- the ISO standard that is most like the Windows character set for accented characters. If the encoding declaration is missing, then the character encoding is assumed to be something called *Unicode (and specifically the Unicode format called UTF-8)* -- and UTF-8 uses a way to encode accented characters that is quite different from that normally used by Windows programs.

If your XML and/or XSLT document contains accented characters, and they look OK when viewed on your windows computer, but if an XSLT processor (like *XALAN*) complains about them it is likely that the XSLT processor is assuming that the accented characters are encoded in *UTF-8*, and they are actually encoded using *ISO-8859-1*. In this situation, add the encoding attribute as shown above to the front of your XML document, and you will probably see the problem go away.

### 5.3 The XSLT stylesheet element

The first tag in the XSLT stylesheet announces the document element that will contain all the other elements in the stylesheet. The form we will use for most of our stylesheets is:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
```

Both the attributes are required:

- **xmlns:xsl="http://www.w3.org/1999/XSL/Transform"**: as mentioned earlier, this attribute is a "namespace" declaration, and asserts that elements in this document which are tagged with the `xsl:` prefix belong to XSLT.
- **version="1.0"**: asserts that the stylesheet follows version 1.0 of the XSLT stylesheet specification.

Unless there is good reason, code the opening tag exactly this way.**[XSLT §2.2]**

### 5.4 The *xsl:output* element

In almost all the stylesheets we will be looking at you will see the `xsl:output` tag immediately following the opening `xsl:stylesheet` tag. The `xsl:output` tag allows one to assert what kind of document is to be generated. To ask your XSLT processor to generate HTML as a result of a transformation, code:

```
<xsl:output method="html"/>
```

Other possible values for the method attribute that are widely supported are *xml* or *text*. There are a number of other attributes that can be attached to the `xsl:output` element, and we will not introduce them here.

Note that the `xsl:output` element is itself always an empty element. Although it never has any content, the significance of the attributes set in it will apply to the entire generated document.

For more information about `xsl:output` see **[XSLT §16]**

### 5.5 The *xsl:template* element

XSLT stylesheets almost always consist mostly of `xsl:template` elements. The `xsl:template` has a number of different uses, but it is generally used to specify how a particular object (usually element) in the input document (and that element's content) is to be transformed into the form suitable for presentation in the output document.

In our first XSLT example you have seen the most common way to use the `xsl:template` element: to provide a "rule-based" transformation, and until chapter 4 in this tutorial all our templates will be of this type. The simplest general form of `xsl:template` element in rule-based transformations is

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

```
[materials to include before node's content]
<xsl:apply-templates/>
[materials to include after node's content]
</xsl:template>
```

Where *node* can mean element, attribute, comment, text and some other XML components. The concept of node will be more formally explained in Chapter 2 § 2.

where the square-bracketted materials are replaced by a combination of tags and text that represents the material to include in the output file before and after the transformed content of the *element*.

An XSLT stylesheet, then, contains *templates*, each of which provides a rule to guide the generation of output. Each template contains a matching pattern in the `match` attribute that indicates when the particular template is to be invoked. When the XSL processor finds XML data that matches a template pattern, it applies that template's styling rules as shown in *xsl:template*'s contents to the data extracted from the input XML data, filtering out unwanted sections, and manipulating the data into some presentable layout.

The `xsl:template` element is fully described in the standard at **[XSLT §5.3]**

### 5.6 The *xsl:apply-templates* element

The `xsl:apply-templates` element indicates where other material (usually the current element's content) should be placed inside the template rule. It is usually coded simply as "`<xsl:apply-templates/>`": as an empty element with no attributes; although it can sometimes take optional attributes that somewhat change its behaviour. We will introduce some of these later in the course module. For more information about it see the full description in the standard at **[XSLT §5.4]**

Note that if you use an XSLT editor like *Oxygen* you may produce the following: `<xsl:apply-templates><xsl:apply-templates/>`. There is nothing wrong with this as code, but it is quite verbose, and we usually use the shorcut `<xsl:apply-templates/>` instead. It is possible to include content inside an *xsl:apply-templates* element, although you are **very** unlikely to ever need to do so.

## 6. HTML in XSLT stylesheets

We have already stated that one of the uses of XSLT is to transform XML into HTML documents so that they can be viewed on the web. We know that HTML is somewhat similar to XML sytactically, but traditionally it has not been used in an XML conformant manner. We also know that an XSLT stylesheet must be a fully-formed XML document, and that one that generates HTML must contain HTML tags inside itself. How, then, must the HTML be coded if it is to be successfully included in the stylesheet? In this section of the notes we talk about several practices that are common in HTML generation that, in fact, will cause problems if they are followed when coding HTML in XSLT templates. We also show how these problem should be fixed.

*Be aware that the type of HTML mentioned in the following discussion is the pre-XHTML variety. If you have some sense of the difference between XHTML and older HTML syntax, then the following should be familiar.*

- **Empty tags**: There are a few tags in HTML that never contain any content. Examples include `hr` (display a horizontal rule), `br` (insert a line break in the output) and `img` (insert an image). When these elements are included in the XSLT stylesheet they must be marked explicitly as empty tags by ending them with the sequence "`/>`". It is recommended that you add a space before the forward slash ` /`.

| **HTML allows:** | **XSLT requires:** |
|---|---|
| <img src="myimg.jpg"> | <img src="myimg.jpg"/> |

- **Quoting of Attributes:** HTML allows values of attributes to be specified in certain contexts without quote marks. Quote marks are always needed in XML documents.

| **HTML allows:** | **XSLT requires:** |
|---|---|
| <img src="myimg.jpg" align=left> | <img src="myimg.jpg" align="left"/> |

- **Attributes without values:** Certain attributes can be specified in HTML tags without attribute values. Values are always needed in XML document. To get around this problem, the common practice is to either use the attribute name as its value as well, or to use the value "1" as its value.

| **HTML allows:** | **XSLT requires:** |
|---|---|
| <table border> | <table border="border"> |

- **XML element names are case sensitive:** Case of the letters in an element name in HTML has traditionally been unimportant (although in recent versions lower case has become obligatory). Since XML element names are case sensitive, the case of the name in the closing element must match the case used in the opening:

| **HTML allows:** | **XSLT requires:** |
|---|---|
| <H1>Alice in Wonderland</h1> | <h1>Alice in Wonderland</h1> |

  Note here that the XSLT processor itself will not care whether the HTML header tag is coded as "`H1`" or "`h1`" -- it only is concerned that the *same* capitalisation is used for both opening and closing tags.

- **Closing tags are always required:** Some HTML tags (<p> or `<li>`, for example) do not require a closing tag. In XML all tags must be closed.

| **HTML allows:** | **XSLT requires:** |
|---|---|
| `<ol>` | `<ol>` |
| `<li>Apples` | `<li>Apples</li>` |
| `<li>Oranges` | `<li>Oranges</` |
| `</ol>` | `li>` |
| | `</ol>` |

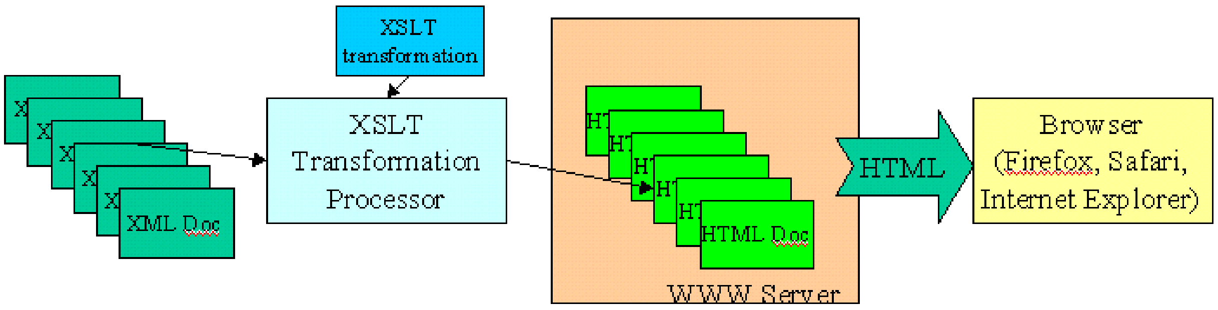
- **Tags must be properly nested:** Usually HTML browsers do not care if tags are properly nested or not, and some older programs (such as older versions of Microsoft Word, when generating HTML from Word documents) don't bother to properly nest elements that they generate. These improperly nested tags must be fixed when used in an XSLT document.

| **HTML allows:** | **XSLT requires:** |
|---|---|
| <p>Here is some <font color="red">red text.</p></font> | <p>Here is some <font color="red">red text.</font></p> |

## 7. XML, XSLT and the WWW

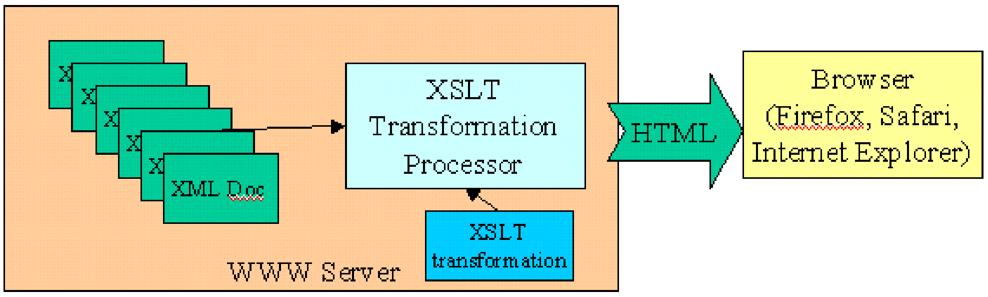
There are three ways, involving (let us say) different "configurations" of the documents and software, in which XSLT is used to transform XML into HTML.

1. The first of the three configurations matches most closely to what we have already done and will continue to do for this tutorial.


XSLT transformation
XSLT Transformation Processor
XML Doc
HTML Doc
WWW Server
HTML
Browser (Firefox, Safari, Internet Explorer)

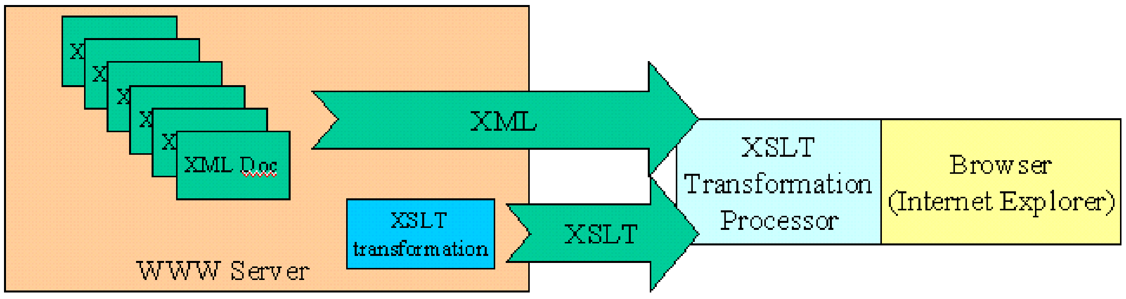
Here, the XML documents (shown on the left) are transformed by an XSLT processor into a set of HTML documents before they are placed on the WWW using an *XSLT processor* (a piece of software such as *Saxon* or *XALAN* that can perform an XSLT-type transformation on an input XML document) that is not necessarily sitting on a Web server. The resulting HTML documents are then shipped to a WWW server (shown in the middle), and are thereby made usable by any HTML aware browser. The requesting URL refers to the HTML version of the document, and the web server simply delivers the requested document in exactly the same way as it would for any other HTML object.

2. In the second of the three configurations the transformation is done by the Web server at the time the browser requests the document.


XML Doc
XSLT Transformation Processor
XSLT transformation
WWW Server
HTML
Browser (Firefox, Safari, Internet Explorer)

Here the server holds the XML documents, rather than HTML versions. When the browser requests the XML document the server, just before it delivers it to the browser, applies the transformation into HTML described in the XSLT transformation document. The browser only sees a document containing HTML which it then has to display. Here, the XSLT transformation processor must available on the server to be invoked by it whenever an XML document is requested by a browser.

3. The third of the three configurations requires the browser (such as Internet Explorer) to contain an XSLT processor:



Here, the browser delivers both the XML document itself, and the XSLT stylesheet (which describes how to translate the XML document into HTML) to the browser. The browser uses its own XSLT processor to first convert the XML into HTML, and then uses the resulting HTML instructions to tell it how to display the document on the screen.

Of these three configurations, (1) and (2) allow the viewer to use any browser, whereas only users with new versions of browsers can see the documents properly with scheme (3); notice that schema (3) has some other limitations as, for instance, it doesn't allow to process multiple documents (see Chapters 3 and 4). In the WWW context, where users from around the world can potentially visit your website, it is unwise to assume (as in configuration (3)) that the user is using a browser with an XSLT processor built into it.

## Exercises

**Ex. 1.** Take the *eeyore* xml and XSLT files and use the XSLT processor to create an HTML file:

- Open the `D:\user` directory and empty it of all files.
- Use the browser to fetch the bundle of exercise materials provided for you to use, and double-click on it to unpack it. It will create a folder containing exercise materials for this tutorial, and that folder, in turn, will contains a copy of the XML document `chap 1 > exercises > eeyore.xml`. Open it in Oxygen so you can see its contents.
- Now examine a copy of XSLT document `chap 1 > exercises > eeyore.xsl` in Oxygen.
- Invoke the XSLT Processor, asking it to use `eeyore.xml` as its input xml file, using `eeyore.xsl` as its XSLT stylesheet, and creating the resulting file in `eeyore.html`:
- Examine eeyore.html both in a text editor (e.g. Notepad) and using your browser.

**Ex. 2.** Using the same "eeyore" materials as you used in exercise 1, modify the XSLT stylesheet so that the background colour displayed by the browser from the generated HTML is a cream colour (done by setting the bgcolor attribute of the body tag to "#FFFFE0":

```
<body bgcolor="#FFFFE0">
```

After editing and saving the XSLT stylesheet, run the XSLT processor as before so that it can regenerate the HTML document, and check it to ensure that it is displaying correctly.

Then, further modify the XSLT stylesheet so that the XML *quote* element displays as centered rather than as a `blockquote`.

**Ex. 3.** It would be nice to have the *title* element in the eeyore.xml document appear in both the `body` of the HTML document as it does now, but also in the `head/title` of the HTML document. Think about the aspects of XSLT that you've been introduced to so far and either show how this could be done, or explain why it cannot be done.

**Ex. 4.** For this and the following exercise we will work a on new document `chap 1 > exercises > books.xml` which is provided in the bundle of files made available to you for this chapter's materials. View it with *Oxygen*. It is a simple list of books with their prices, shown as a XML document. Note the relationship between *books*, *book*, *author*, *surname*, *firstnames*, *title* and *price* elements.

Now take a look at an HTML rendering of this material as an HTML unordered list: `chap 1 > exercises > books1.html` and establish for yourself how the tags that are contained in the HTML document could be related to the tags found in the XML document. Based on this model, create an XSLT stylesheet that will create an HTML document like this for you. If you wish, start with the model XSLT document contained in `chap 1 > exercises > template.xsl`. Test your stylesheet with the XSLT processor.

You might note that the output HTML document contains two characters that are not in the input XML document -- the comma that separates the author's surname from his/her first names, and the "£" sign that appears prefixing the price. To get them to appear simply place them in the appropriate place in your XSLT stylesheet.

You might note as well that some information that might be useful is encoded as attributes to the book element. You will learn in the next chapter how you might take advantage of this information in an XSLT stylesheet.

**Ex. 5.** Now, create a new stylesheet that will render the same booklist as a table as in `chap 1 > exercises > books2.html`. This time it might be easiest to copy the stylesheet you created for the previous exercise, name the copy `book2.xsl`, and then modify it to generate the table form. Be sure to test your stylesheet using the XSLT processor.

**Ex. 6.** This exercise asks you to work with a short excerpt from *Hamlet* as marked up by Jon Bosak, a well-known writer on SGML and XML markup. You can see the XML text in: `chap 1 > exercises > hamlet-excerpt.xml`. You will be aiming to produce output that looks like this: `chap 1 > exercises > hamlet-excerpt.html`. Compare the XML and the resultant output, and, as usual determine the mapping between the XML elements and the HTML output. Note as well that certain material in the XML document does not appear in the HTML -- how can your characterise what is missing?

1. Begin by creating a stylesheet that deals with the play's Title, with the scene description, with the act, scene designation, and with the stage directions and speech text, including the "lines". Run your resultant stylesheet through the XSLT processor. Note what happens to text contained in elements for which you have not provided a template -- can you characterise what occurs?
2. One of the tasks that needs to be done is to suppress the list of *Dramatis Personae*. The easiest way to do this is to put in a template that does not include the `xslt:apply-templates` element, e.g.:

   ```
   <xsl:template match="PERSONAE">
   </xsl:template>
   ```

   Since there is no `xslt:apply-templates` element, the XSLT processor is not instructed to process any contents of the `PERSONAE` element. In effect, then, the entire contents is suppressed in the resultant output. Modify your stylesheet to include this empty template, and insert a similar stylesheet template to suppress the `fm` element as well.
3. Note that the TITLE tag appears in a number of different contexts within the document -- once near the beginning for the entire play, once at the beginning of the *Dramatis Personae*, and again to announce the Act and Scene. Although the element name is the same in each case, the title is, of course, for a different thing in each of these contexts -- one is the title of the play, one is the title of the Act, etc.

   XSLT allows one to make this distinction in the "match" element of the *xsl:template* tag -- allowing, in effect, a different template to be used for each kind of title. You will learn more about the exact notation and its meaning in the next chapter -- but for now, use the notation `<xsl:template match="ACT/TITLE">` to begin a template that is to be used for any TITLE element which is directly nested inside an ACT element. Similarly, separate templates can be written for `SCENE/TITLE` and `PLAY/TITLE`. Use this technique to ask XSLT to put the play's title in an HTML h2 element, the Act's title in an *h3* element, and the scene title in an h4 element.

# Chapter 2

# XML-Trees and XPATH; References and IDs

## 1. Introduction

As mentioned in the previous chapter, there is an important element in XSLT which is, strictly speaking, not a part of XSLT at all -- the component that allows one to select elements so that one can specify what is to be generated from the contents of these elements. *XPath* started out as a part of XSLT, but when it became evident that XPath was useful for other new XML based developments in addition to XSLT, it was split off from XSLT. It now exists as a separate standard which is developed and maintained by the W3C -- there is a detailed technical description at `http://www.w3.org/TR/xpath`. Although there is a version 2.0 from W3C in the works, we will be basing our material on Version 1.0 of XPath which is used with Version 1.0 of XSLT.

We have already mentioned that XPATH is the language that is used to describe how an XSLT processor is supposed to select parts of an XML document for processing. In the last chapter's basic XSLT stylesheets, we wrote some XPATH when we provide a value for the `match` attribute of the `xsl:template` element. Although, of course, the XPATH we used then was very rudimentary, today we will look at more of the XPATH notation, and thereby expand the kind of things we can specify with XPATH. We will also find new places to use it as we see more XSLT constructs.

First, however, XPATH is based on a particular view of an XML document, called a "tree", that we need to introduce now.

## 2. The *Tree* view of XML documents

In your work with XML so far you have seen only a "file view" of XML documents. This way of representing XML is used to store XML documents in a conventional computer file, and, as you know, emphasizes a view that presents XML as what at first appears to be a simple sequence of characters, some of which (announced with "<" and ">" characters) represent tags that, in turn represent elements. When you view an XML document in a non-XML-aware viewer such as Notepad it may not be immediately evident that the elements shown there form a hierarchy -- but as you know by now this hierarchical organisation of elements is characteristic of all XML documents -- indeed is one of an XML document's fundamental organisational principles. For example, in this very small XML document:

```
<p>Here is some <hi type="ital">highlighted</hi> text.</p>
```

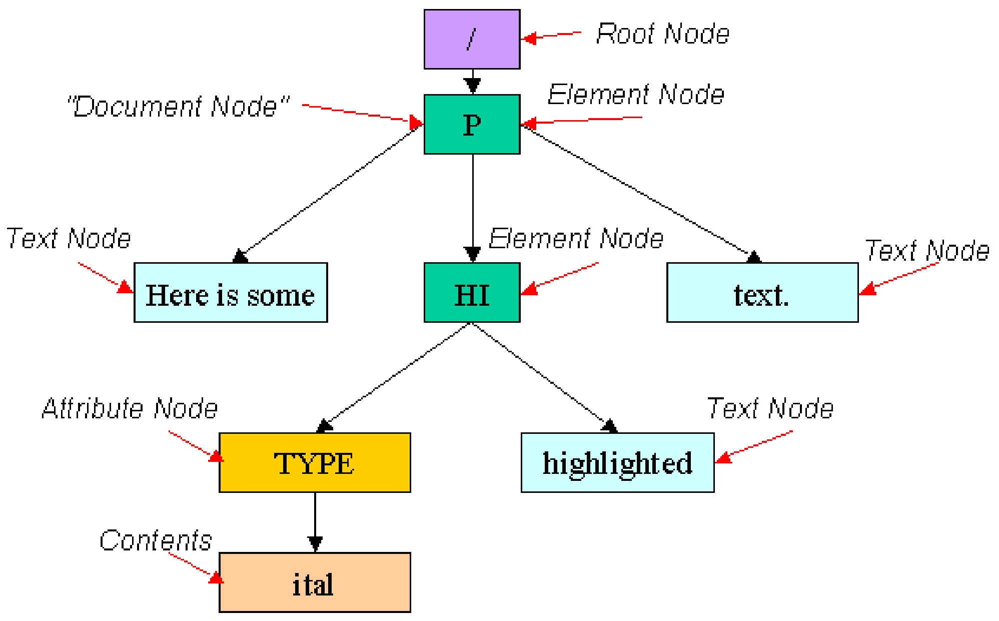
we can see the following things:

- The contents of the file can be interpreted as having text, elements, element attributes (and various other kinds of object (e.g. "Processing Instructions") that we will for the moment ignore).
- Elements are announced by an opening tag, and continue until a corresponding end tag is found. The material found between the opening and closing tag is the element's *content*.
- Elements can be included inside the content of other elements. We can see, for example, that the `hi` element is entirely enclosed (or owned) by the P element.

- The content of any element can be a block of text, a set of elements, or a mixture of both text and elements (exactly what is allowed is, of course, determined by the DTD if there is one).
- Element attributes (such as `type` in the example above) clearly belong to the element, but are also clearly different from an element's content.
- The content of any element is "owned" (and thereby subordinate) to that containing element -- in this way a hierarchical structure is developed.

The structure and contents of an XML document can be presented in quite a different, although equivalent, way: as a *tree view* or *tree graph* ("tree" for short). This view is not one that one sees in an XML computer file, but is often the form in which XML documents are held when they are being manipulated by computer programs such as XML editors or an XSLT processor such as *XALAN*. Such a program, when it reads in an XML file to work on it, first transforms the data it contains into a tree representation. In XML editors, when it is time to "save" the edited document back to disk, the program takes the tree representation it then has and creates a corresponding file representation to be stored on disk. The tree view, although *equivalent* to the file representation, emphasises the relationships between the parts (elements, attributes and text) that make up an XML document in a way that allows the computer to locate and manipulate them most effectively. The following figure illustrates this tree representation by showing the same short bit of XML we showed earlier in both forms: in the file-format at the top, and as a tree underneath:

<P>Here is some <HI TYPE="ital">highlighted</HI> text.</P>



- This kind of XML representation is called a *tree* by mathematicians, and if you turn the page (or monitor) upside down you will see why. All the boxes in the figure are connected to each other by directed lines (shown by black arrows), and they are connected in such a way that you can see that they all originate from a single box -- in the same way that the leaves on trees in nature are connected to little branches, which are connected to bigger branches which eventually connect to the trunk which connects to the tree's root.
- A tree is made up of *nodes* and *links*. The *nodes* are shown in the figure above as boxes, and the *links* as black arrows. When an XML document is shown as a tree, it turns out to be important to be able to distinguish different kinds of nodes, and if you are viewing this text in a colour representation, you will see different kinds shown with

different colours. The node kinds are also labelled in the diagram by text that is shown here as connected to the node by red arrows. (The red arrows and the labels attached to them are not considered a part of a proper tree representation.)

- The one node from which the other nodes originate is called the *root* node. For some reason, tree views of XML documents always show the root node at the top (!) rather than at the bottom. In this figure the root node is shown in purple when viewed online. It is often useful to think of the tree representation as a hierarchy -- with objects nearer to the top of the figure being higher up in the hierarchy than those lower down. Notice that "sub-tree" of nodes directly under a node "A" and connected to it would be "A"'s *content* in a file view. Thus, we say even in the tree representation that these nodes are *contained* within "A".
- In this particular tree (representing the very simple bit of XML, after all) there are two nodes that represent the *elements* in the XML document. Nodes that represent elements are called *element nodes*, and in this figure they are labelled "Element Node" and are shown (when viewed online) in green. You can see the names of the element they represent in the boxes: `p` and `hi`. Since the entire XML document is contained within a single p element, the p element is shown as directly connected to the root node, and just below it in the hierarchy. All the other nodes are either directly or indirectly part of `P`'s content. Since the `P` element is the "top" element in the tree, and contains all of the other nodes as its content, it has a special name. In addition to being called an Element Node: it is also referred to as the *Document Node*. Note that the document node is *not* the same as the root node.
- Directly connected to the p element are the three nodes that represent the 3 items that are directly attached to the p element. The outer two are pieces of text. They are shown as *Text Nodes* (light blue) in the figure. Although Text nodes have textual content, they can never have further nodes below them in the hierarchy (Can you think why?).
- Between the two text nodes is another element node, representing the `hi` element that is contained within the p element. The `hi` element has an attribute attached to it. The attribute `type` "belongs" to the `hi` element, and is shown here as an *Attribute Node* which has its value shown as the attribute's "content" (both shown as orange). The text node containing the word "highlighted" is also considered to belong to the `hi` node. In the XPATH tree model, however, attribute nodes are *not* considered formally as a part of their element's *content*, although the enclosed text node(s) (and element nodes, if any) would be.

XPATH makes much use of genealogical terminology, because a tree graph bears an obvious similarity to a genealogical chart. In this terminology, then, the relationship between two nodes can be given a genealogical interpretation. In the above figure, for example:

- the HI element is said to have the P element as its *parent*. Note that unlike in human genealogy, all nodes except one (which has no parent -- do you see which one?) have exactly one parent (not two).
- The P element in the example has three *children* nodes -- the two text nodes and the contained HI node. Any element node can have any number of children nodes -- although text nodes never have any for obvious reasons.
- When the children of a particular node consist of both other element nodes and text nodes, that node is said to have "mixed content".
- Nodes that share the same parent are said to be *sibling* nodes. Thus, the two text nodes and the HI element node are all children of the P node, and are therefore considered to be *sibling* nodes.
- The relationship between *attribute* nodes and their owning element node are, perhaps strangely, not described using this geneological terminology -- the TYPE attribute node shown in the above figure is *not* considered to be a child of the HI element, even though

it belongs to it. Thus, there is a distinction made in XPATH's geneological terminology between attribute nodes and element/text nodes

- The chain of nodes from any particular node to the root is called that node's *ancestors*. For example, the ancestor nodes for the text node "highlighted" in the above figure are the HI, P and root node. The list of ancestor nodes for any particular nodes corresponds to the elements in the file view in which the given node is contained -- the word "highlighted" is contained in the HI element which is, in turn, contained in the "P" element which is, in turn, contained in the root node.
- Text and element nodes that are contained within a particular node are called that node's *descendants*. Any element node, with its descendants, can be thought of as a *sub-tree* of the complete document tree, in that it will also of necessity possess a tree-like structure.

Of course, the material in any XML document has a particular order associated with it. Thus, the ordering of nodes in an XPATH trees is important. This ordering is called the *document order*. In a 2-D picture representation of graphs one tries, as much as possible, to order the nodes from left to right so that the document order of these corresponding nodes is made visible. The vertical direction is generally used to represent the hierarchy of nodes -- what belongs to what.

We will see this genealogical-like terminology much used when we talk about XPATH in more detail, so it is important that it be clearly understood. There is a rather technical introduction to the relationship between XML trees and XPATH in the XPATH specification **[XPATH §5]**.

## 3. XPATH Notation

XPATH is a notational scheme for expressing relationships between nodes in an XML tree. XPATH is used in three ways in XSLT:

1. In one use, an XPATH notation is provided to select things. You have already seen this kind of use, since XPATH is used in this way in the `xsl:template`'s `match` attribute. We will see other places where XPATH is used to select nodes from time to time in this module. An XPATH notation used in this way is called a *pattern* or *path*.
2. XPATH can also be used to provide a value (a string of characters) which can be used in various ways inside an XSLT stylesheet. We will see examples of this kind of use later today. An XPATH used this way is called an *expression*.
3. Finally, we will see examples later in the module when XPATH notation is used to provide a test to be applied against a part of a particular document, where it might be *true* or *false*. Again, we will see examples of this later in the course. An XPATH used this way is considered a kind of expression as well.

There are some differences in how one interprets an XPATH notation when it is a path, and when it is an expression. Please note them, because it will make it easier to make XPATH expressions do what you expect in XSLT stylesheets.

### 3.1 XPATH *Paths* and *Steps*

XPATH notation can be written in two separate but related ways: using what is called in the XPATH specification the *unabbreviated syntax* and *abbreviated syntax*. The *unabbreviated syntax* allows more things to be expressed, but is more complex to create and understand. The *abbreviated syntax* does not allow the full range of options that the full notation allows, but is easier to write, and you will find that it allows you to say enough things in XPATH to meet most of your needs. Thus, the *abbreviated syntax* we will introduce to you now will usually be what you will need, and you will need to use the *unabbreviated syntax* only in places where you need to say things that cannot be said in the basic style. We will introduce the *unabbreviated syntax* briefly in chapter 4, but for now we will focus on the more basic notation of the *abbreviated syntax*.

We will begin by examining XPATH *patterns*, as they are used in <xsl:template match="...">. As we mentioned above, when XPATH is used as a selection *pattern*, the XPATH expression replacing the "..."s provides a pattern that the XSLT processor can use to locate nodes in the input document to which the template should apply.

- You will recall from the previous chapter, that the construction:

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

  tells the processor to look for *para* elements in the text, and apply the template which follows to those elements. Here, the path consists of a single name -- the name of an element in the input document called *para*. A path of this kind "matches" all elements with this name in the input document.

  It is important to remember that the match operator always could match more than one element in a particular document. When used as a template match attribute, it will produce a *set of nodes* that it matched, and the template that follows will be applied to *all* these nodes that the match attribute selected from the document.
- XPATH allows you to connect two element names together using what it calls the *step operator*: "/". In the abbreviated syntax notation the element after the step operator is assumed to be a *child* of the element before the step operator. Thus, the construction:

  ```
  <xsl:template match="section/title">
  ```

  begins a template that matches title elements only when their immediate *parent* is the section element (note that the selected nodes here are "*title*" nodes, not "*section*" nodes). It is useful to be able to provide this type of specification when you have a document where a `title` element can be used in different kinds of elements and you want to have different XSLT processing for these different situations. For example, the path `subsection/title` will match different "title" elements from those matched with `section/title` -- the first matches titles within subsections, the second titles within sections.

  More than two elements can be chained together using the step operator. The path `body/section/title` selects *titles* that appear in *sections* that, in turn, appear in the *body* element, and not other titles.
- The notation symbol "*" is used to mean "any node". Thus, the path `forward/*/p` selects by first finding the *forward* element node (or nodes), selecting any immediate child of the forward node, and then from any of those immediate children, selecting *p* nodes. Here again, the end of the expression indicates which notes are selected -- here "p" nodes that meet the test, not "forward" nodes.
- When two step operators are put together with nothing between them ("//") the path refers to any *descendant* of the first named node who matches the second. Thus, the construct `<xsl:template match="section//p">` means any *p* element in a *section* whether it is a child (direct descendant) or some other descendant node. This is a little like the "*"-notation example given in the previous point, but matches even if there is more than one level of elements between the `section` and the p element.
- When the step operator "/" begins a pattern it changes it's meaning and becomes a reference to the root node. Thus, the pattern: `<xsl:template match="/doc/section/p">` matches only p elements who's parents are `section` nodes whose parent is the doc node who is, of necessity since it is the child of the root node, the document node for the entire input document.

Paths with the beginning "/" are called *absolute*, paths without it are called *relative*. Note the similarity in the role of the "/" in XPATH paths, and in the directory portion of WWW URLs.

- When XPATH has selected the set of nodes that match the given criteria for the *xsl:template match* element, the processor will apply the template to *each node in turn* in this set. As the template is applied to each node, that node is called the *current node*, and we will soon see places where this idea of a *current node* is important. The full-stop character (".") as an item in a path refers to what is called this *current-node*. One use of this construct (and there are others, see below for examples) is to allow you to create a relative path that begins with the a search for descendants of the current node: "`.//para`" matches all paragraphs that are contained in the current node ("`//para`" matches all para elements in the entire document since it begins with the "/", and is, therefore, relative to the document's root). Examples later in the tutorial will make the meaning of "." and that of the *current node* clearer.

An important thing to remember: XPATH notation results in selecting the last (right-most) path item. Thus `body/section/title` selects element nodes with name **title**, *not* element nodes with name **body**.

Examples of paths specified with XPATH notation (including constructs we will introduce later in the module) can be found in the XSLT and XPATH standard documents at **[XSLT §5.2]** and **[XPATH §2.5]**

## 4. XPATH in *xsl:apply-templates* select attribute

So far we have seen only templates in which the the `xsl:apply-templates` is used without any attributes. When it appears in a template it causes an XSLT processor to take the contents of the selected node and apply templates to each of the nodes found there in turn, and in their original order. However, the `xsl:apply-templates` element can be provided with a `select` attribute, which, if used, causes the processor to selectively process content nodes. Selective inclusion of content, achieved in this way, is particularly useful when more than one `<xsl:apply-templates/>` element is provided in a template.

Suppose, for example, that you had an XML document containing entries like the following:

```
<AUTHOR TYPE="main">
  <NAME><FIRST>Professor Christopher</FIRST><LAST>Gantley</LAST></
NAME>
  <AFFIL>University of Northumbria</AFFIL>
  <EMAIL>c.gantley@unnx.ac.uk</EMAIL>
</AUTHOR>
<AUTHOR>
  <NAME><FIRST>Megan</FIRST><LAST>Graham</LAST></NAME>
  <AFFIL>Institute of Image Research, University of Northumbria</
aFFIL>
  <EMAIL>megan.graham@unnx.ac.uk</EMAIL>
</AUTHOR>
```

Here, the document structure uses elements FIRST and LAST to explicitly label the first and last names of the authors, and gives them in their normal speaking order.

Suppose, however, that you need to generate them with the names reversed and italised:

*Gantley, Professor Christopher*,
University of Northumbria, ...

The following template will achieve this goal:

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

```
<Ii><xsl:apply-templates select="LAST"/>, <xsl:apply-templates
select="FIRST"/></i>
</xsl:template>
```

This template is invoked when the XSLT processor is working on `NAME` elements. As you can see, there are two `xsl:apply-template` references in the template, and each has a `select` attribute that specifies elements that are contained within the `NAME` element. To understand how the select works here it is important to understand that an XPATH expression in an `apply-templates select` attribute is, in fact, interpreted somewhat *differently* from that in a `template match` attribute. The most important difference is this: the `select` attribute works *only in the context of the nodes matched by the template*. What happens is that before the select attributes get to choose elements the template's `match` attribute has already generated a list of nodes to which the template rule will apply. In this case the list the `match` attribute will have generated will be NAME elements (two in our small example). The `select` attributes within this template, then, are applied only against each of the `NAME` elements in this list in turn. Thus, when the first NAME element is being processed (describing Professor Gantley), the specification of LAST in the select element refers only to the LAST element within Gantley's NAME element. When the second NAME element is being processed (describing Megan Graham) the LAST element referred to is the one within Graham's NAME element.

There is a further subtle but important difference between the meaning of the path in the `xsl:apply-templates select` attribute and `xsl:template match` attribute. In the `apply-templates select` attribute, an XPATH pattern consisting of a single name will *only* match immediate *children* of the current node -- not *any descendant* of the current node. If the `FIRST` and `LAST` elements had not been direct children of the `NAME` element, the specification of `LAST` and `FIRST` in the apply-templates select would have selected nothing at all. This will appear to be quite different behavour from what would happen with a specification of, say, `FIRST` if it had appeared as a `xsl:template match` attribute value. To match any `FIRST` node with a `select` attribute even if it had been not only the immediate child, but a more removed descendant of the current node, the `xsl:apply-template select` attribute would have to appear as "`.//FIRST`".

In the end, then, this template causes the XSLT processor, for each `NAME` element it finds, to first output an opening `<i>` tag, followed by the contents of any child element node of the `NAME` element called `LAST`, followed by a comma and a space, and then followed by the contents of any child element node named `FIRST`, followed by the closing `</i>` tag. Note that, by this technique, we have reversed the ordering of the input text in the output.

The difference in interpretation of paths between `xsl:template` match elements and select elements (both in `xsl:apply-templates` and elsewhere) is an important one that, if not understood, will trip you up. Be sure that you have it clear.

Information about *xsl:apply-templates* is available in the XSLT standard at **[XSLT §5.4]**.

## 5. *xsl:value-of* element

The `xsl:value-of` element is used inside an *xsl:template*, and takes the following form:

```
<xsl:value-of select="..."/>
```

where the "..." is replaced by a path that selects material in the input document. The `xsl:value-of` element takes the material selected by the given path, converts it to string of characters (removing any element tags that might be in the selected material), and inserts the resulting value in the output document at the place it is found. (Note that what is selected by the `xsl:value-of select` is relative to the *current node* -- in exactly the same way as `xsl:apply-template`'s `select` attribute is.)

When the select path in `xsl:value-of` results in an element, the operation often produces results that appear to be the same as the `<xsl:apply-templates select="..."/>` construct, *except* that the `xsl:apply-templates` approach might result in material inserted into the output document which contains tags, and the `xsl:value-of` construct will never contain any tags.

The value-of construct would be useful when, for example, one wishes to take a part of an XML document and use it not only in the body of the HTML resulting document, but also, a second time, as either all of or part of the *TITLE* element in the *HTML HEAD* section. As you will recall, HTML's TITLE element is not allowed to contain any tags, and the fact that the `xsl:value-of` element takes only the text nodes that are the content of the element neatly solves this problem.

You may recall, for example, that we thought it would be good to be able to use the <*title*> element in the `eeyore.xml` document we looked at in the first chapter as a part of the HTML HEAD as well as having it appear (as an *H2*) in the body of the document. Here is an ideal place to use the xsl:value-of element:

```
<xsl:template match="story">
<html>
<head>
<title>Winnie-the-Pooh: <xsl:value-of select="/story/title"/></title>
</head>
<body>
<xsl:apply-templates/>
</body>
</html>
</xsl:template>
```

Here the template matches the top "story" element, and therefore contains all the top-level HTML tags that appear in a normal HTML document. The `<xsl:apply-templates/>` tag will cause the XSLT processor to use the XML document to produce all the body material -- exactly as before.

The only thing here that is new here is the use of the `xsl:value-of` element in the contents of the HTML title tag. The path given in its `select` attribute is an "absolute" path (shown by the beginning "/"), indicating that the processor should select its material by beginning the specified element match at the root node. The *story* element must be the document node in the tree, and within the story element the processor is asked to locate the *title* element, and insert the text it finds within it at this spot in the output HTML document it is generated. The XML document begins:

```
<story>
<title>In which Eeyore loses a Tail and Pooh Finds One</title>
[...]
```

Thus, the absolute path "story/title" will, indeed, select the title for the story. The processor, then, when using the template shown above, produces:

```
<html>
<head>
<title>Winnie-the-Pooh: In which Eeyore loses a Tail and Pooh Finds
One<title>
[...]
```

Information about `xsl:value-of` is available in the XSLT standard at **[XSLT §7.6.1]**

## 6. Handling XML attributes

So far we have looked at examples of XSLT processing that focuses on elements and their textual content. You may have noted that attributes in the input text are not considered part of the contents of their respective elements, and their values will not automatically appear in any generated output element. Furthermore, although we have shown you some techniques to generate content of an output element from material in an input document, we have not seen how to generate a value for an output *attribute* based on materials in the input document. We discuss these two items below.

## 6.1 Getting Attribute values from elements in input documents

In XPATH one can refer to an element's attribute by preceding the name of the attribute by an "@". Thus, in XPATH, the construct `img/@src` will select the value of the *src* attribute of *img* elements. Here is an example of how one might use the XPATH attribute selector. Suppose that one was working with the list of authors shown earlier:

```
<AUTHOR TYPE="main">
  <NAME><FIRST>Professor Christopher</FIRST><LAST>Gantley</LAST></
NAME>
  <AFFIL>University of Northumbria</aFFIL>
  <EMAIL>c.gantley@unnx.ac.uk</EMAIL>
</aUTHOR>
<AUTHOR>
  <NAME><FIRST>Megan</FIRST><LAST>Graham</LAST></NAME>
  <AFFIL>Institute of Image Research, University of Northumbria</
aFFIL>
  <EMAIL>megan.graham@unnx.ac.uk</EMAIL>
</aUTHOR>
```

and one wanted to take the *TYPE* attribute, if present, and display it in the output -- perhaps in its own column in a table of author data. A template that would do this could be:

```
<xsl:template match="AUTHOR">
<tr>
<xsl:apply-templates/>
<td><xsl:value-of select="@TYPE"/></td></tr>
</xsl:template>
```

Note the use of the `xsl:value-of` element in the last line of the template's content. The select path shows @TYPE, which in the context of selected *AUTHOR* elements, means their *TYPE* attribute. The *xsl:value-of* element will get the value of the *TYPE* attribute if there is one and insert it in its own outputted HTML table <*TD*> element. If the AUTHOR tag has no TYPE attribute given, then the exact action will depend upon whether, for example, the DTD specifies a default attribute values (and whether the XSLT processor reads the DTD well enough to know this). If the DTD gives no default value, we can expect the `select="@TYPE"` to result in an empty result.

The XSLT standard provides an example of using *xsl:value-of* to get the value of an attribute at **[XSLT §7.6.1]**

## 6.2 Generating Attributes in output documents

It is not unusual to find that the value of attributes in an XML document should be used as the value for an attribute in the output document as well. Suppose, for example, that the input XML document refers to images using its own `IMGREF` tag:

```
<IMGREF PICT="mypict.gif"/>
```

and that this needs to be transformed into the standard HTML `img` tag:

```
<img src="mypict.gif">
```

We now know how to use XPATH to get at the `PICT` attribute in the `IMGREF` element: `IMGREF/@PICT` (or, if the current node is an IMGREF element, just `@PICT`). How do we ask our processor to include the value of this attribute in the outputted tag's attribute? Remember that we cannot use `xsl:value-of` here (e.g. ...`<img src="<xsl:value-of select="@PICT">"/>`) because *elements* (such as `xsl:value-of`) cannot appear as part of the values of *attributes* in XML.

XSLT solves this problem by giving brace brackets ("{" and "}") special meanings when they appear inside an attribute in an element in a template. It treats the material between the brace brackets as an XPATH, selects the material the XPATH indicates, converts the result into a string, and inserts the string into the attribute, replacing the brackets and the path. In the XSLT standard these brace-bracketted materials are called *Attribute Value Templates*, and we can use them to transform the input IMGREF element into an output HTML `img` tag in the following way:

```
<xsl:template match="IMGREF">
<IMG SRC="{@PICT}"/>
</xsl:template>
```

Although this example shows an XPATH in the brackets which is a reference to an attribute, attribute value templates are not restricted to selecting attributes. If the input document had been designed to show the URL as content instead of an attribute, then the brackets can still be used to include what was a content value in the input document as an attribute in the output. Suppose, for example, that the input document contains a URL tag which is meant to have its content displayed in the document, but also be used as the HREF in a HTML A-type link (like this: www.kcl.ac.uk). If the input document contained:

```
<URL>http://www.kcl.ac.uk</URL>
```

then the following template:

```
<xsl:template match="URL">
<a href="{.}"><xsl:apply-templates/></a>
</xsl:template>
```

would result in an HTML output like:

```
<a href="http://www.kcl.ac.uk">http://www.kcl.ac.uk</a>
```

Here is a place where it is useful to specify an XPATH selector which refers to the current node -- a use for "." which we promised earlier.

If the input document DTD provided, instead, for the referenced URL to be provided as an attribute:

```
<URL HREF="http://www.kcl.as.uk"/>
```

then the following template would do the trick:

```
<xsl:template match="URL">
<a href="{@HREF}"><xsl:value-of select="@HREF"></a>
</xsl:template>
```

and would again generate HTML output like:

```
<a href="http://www.kcl.ac.uk">http://www.kcl.ac.uk</a>
```

There is more information about *Attribute Value Templates* in the XSLT standard at **[XSLT §7.6.2]**.

## 7. References, IDs and IDRefs

In textual documents it is not unusual to need to be able to point to one spot in a document from another spot. Perhaps in the text of the document in some spot in the text (spot "A") you are referring to something written elsewhere in the document at spot "B". You would then say that the text at spot A *refers* to spot B, and the thing that makes this connection explicit is called a *reference*.

## 7.1 References within texts

Suppose, for example, that you are writing a text on the history of the Church of England, and you have an element in your text that talks about a particular issue -- say, provides a list of bishops in the diocese of Rochester in the 18th century. Later in your document you wish to make reference to this particular list. "We have provided a list (above) of bishops of Rochester in the 18th century ...". This bit of text is an example of a "reference" we described in the previous paragraph because it refers to some other spot in the document -- although it does not make explicit exactly where the reference points to.

In a written text the use of the word "above" might be sufficient to allow a human reader to locate where the reference refers, particularly if the spot referred to is close by. However, merely saying "above" would likely be insufficient if the reference was to something, say, 100 pages earlier. In printed documents, of course, a more explicit reference is possible:

```
We have provided a list (see section 3.5) of bishops of Rochester...
```

or

```
We have provided a list (see page 96) of bishops of Rochester...
```

In both these cases the link is to a much more specific spot in the text. If the text is maintained dynamically by the computer, both these references would have to be "maintained" as the document changed. If a new section was added before 3.5, then the "3.5" reference might need to become "3.6". If any kind of new text as added or removed, a page number reference to "96" might well have to change to "97" or "95" - or the change might be more radical if the text was substantially reordered.

If we are working with a digital document than it is reasonable to expect the computer to keep track of such things: to know that the table of bishops turns out to be printed on page 96, or appears in section 3.5. However, it can only do this if the table of bishops is made explicitly visible to it. In the reference one needs to be able to tell the computer exactly what bit of text is being referenced if you expect it to keep track of where it appears for you. The "reference" you would then give to the computer would have to take on the task of specifically identifying the textual object that it refers to.

In textual situations like this it might be possible to identify, say, a paragraph of text by describing its location within the text as a whole: e.g. "*the third paragraph in section 3.5*". This will work as long as the text is not changed in such a way that it stops being the 3rd paragraph --but this can go wrong in many different ways. If, for example, a new paragraph was inserted as the first paragraph in section 3.5, then what was the "third paragraph" becomes the fourth. Similarly, if section 3.4 is moved to, say, 8.6 -- then what was section 3.5 becomes 3.4, and again the reference to "the third paragraph in section 3.5" will point to the wrong thing. This kind of reference, then, is very dependant on the *context* of the referred thing, and is likely therefore to be unsatisfactory if the text in which it is used is sufficiently dynamic that the context of the thing referred could change.

Another, and preferred, way to deal with specifying a reference to the computer is to assign an identifier to the object being referenced that stays with it no matter what happens to its context. This identifier should *only* have the significance of allowing one to uniquely identify an object -- and should not be used other purposes as well. It thus must be unique, and not used by more than one object in the document. By attaching a unique identifier to a spot in our text we can use that identifier as a name to point at that spot from elsewhere and be sure that the system will always point to the correct object in the text no matter where in the text the object eventually ends up.

In HTML a spot in a document can be identified in this way by using the `name` attribute of an `A` tag: To identify the table containing our list of bishops in an HTML document so that it can be referred to elsewhere one could wrap the table with a name-type A tag:

```
<a name="BishRochester"><table>
<tr><td ...>Bishops of Rochester</td></tr>
[...]
</table></a>
```

Here the HTML table containing the list of bishops of Rochester is "named" (given an name identifier) of "BishRochester".

Then, elsewhere in the document one can request from your browser a blue-and-underlined link which refers to this identified Bishops of Rochester table by making use of the name assigned to it in a a href element:

```
We have provided <a href="#BishRochester">a list</a> of bishops of
Rochester ...
```

When a browser sees this HTML it will look for a bit of HTML in the current document that has been named "`BishRochester`". If it finds it, it will turn the text "a list" into a link to the identified spot.

In summary, then, referencing within a document involves two separate mechanisms:

1. First, one provides a set of unique identifiers for spots in the document you wish to refer to. Note how important it is that the identifiers you provide be *unique*. If there are two spots named "`BishRochester`" a browser will not know which one it should use. In HTML you attach an identifier to a spot in your document with the `<a name="">` tag. The "`BishRochester`" is the *ID* for this element in the text.
2. Then, one uses a mechanism to refer to the spots you identified in 1. In HTML one can use named spots in the text as the destination for a link by providing a name-reference in the `a href` attribute: `<a href="#BishRochester">`. This URL is a *reference* to an ID found elsewhere in the text.

References between paragraphs are not the only kind of references that might appear in an electronic document. Perhaps you are generating a table of contents for a printed text and need to say on what page a section begins, or perhaps you are doing a book index and need to be able to say that a particular topic appears on a set of pages in the book. These three kinds of inter-document pointers are considered called "references", and soon we will be describing how they can be handled in XML and XSLT. Furthermore, although one might use references in an electronic text to produce hyperlinks between two spots in the text, this is not the only possible use for such references. If the text is to be printed on paper rather than simply displayed on the WWW, then a reference might better appear as a reference to the page containing the destination of the link. It would be the job of the software processing the reference to make this happen.

## 7.2 XML's IDs and IDREFs

We have described the setting up of intra-document references in HTML using its "a name=" attribute. In HTML the attribute names used for referencing are fixed by the HTML specification. One always uses the A element's `name` attribute to attach an identifying name to a spot in the text, and one always uses a *name reference URL* in the `a href` attribute to refer to it. In XML the process is similar to the HTML: one just described in that one uses an element attribute to contain an identifier for the element, and a different attribute to refer to the elements thus named. However, in XML it is not tied down to particular element and attribute names. Instead, in your DTD you are able to specify an attribute with whatever name you wish to use for the attribute to identify a bit of text and to refer to text so identified. Often, however, in spite of this one finds that the document designer for an XML document has used the attribute name "ID" for the identifier, and "IDREF" for the reference. Remember that although this is often done, in fact it needn't be so.

Instead of forcing you to use a particular attribute name to contain identifiers and references, in XML asserts in the DTD for a particular document type that a particular attribute is used to identify an element by making its type "ID". One then refers to identified elements in an XML by using attributes that you have declared in the DTD to be of type "IDREF".

Let's say, for example, that in your particular DTD you have chosen to allow the user to attach an ID to your paragraph element *para* by coding an `ident` attribute to hold the paragraph's ID:

```
<para ident="X35">
```

In your DTD for this document you would tell your XML processor that that is what is going on by declaring that the attribute is type "ID":

```
<!ATTLIST para ...
ident ID #REQUIRED
```

This bit of DTD asserts that the `para` element takes an `ident` attribute, that it is an ID-type of attribute, and that it is required (it must be coded *each time* the para element is coded in your document). If you wanted, instead to allow someone to code an ident attribute as an ID only when needed -- not for every para element, one changes the "#REQUIRED" to "#IMPLIED":

```
<!ATTLIST para ...
ident ID #IMPLIED
```

The "#IMPLIED" information asserts that the "ident" attribute was optional - that it was alright if some (or all!) para's didn't specify it.

If you identify an attribute as an ID in your DTD, then you are asserting that the values assigned to it in any document using that DTD will only occur once in that document. If a particular ID, say "X35" as attached to the para element above, is found, then *no other* ID attribute within the same document may have the same ID assigned to it. This uniqueness must be true even if the attribute is in another type of element. If you have `<para ident="X35">` in your document, you may not also have `<list-item id="X35">`, if the "id" attribute for list-item is asserted to be an ID type attribute in the DTD as well. If you do assert that an attribute is an ID in your DTD, then an XML processor will check to be sure that this uniqueness is in fact correct -- that indeed any particular ID string like "X35" does not occur more than once among all the attribute values of type ID. It cannot declare that your XML document is "valid" if you break this rule, and usually will then stop processing as soon as it detects that this rule is broken.

The values assigned to ID elements are restricted to certain characters. The only characters that are permitted are letters, digits, full-stops, dashes and underscore. Furthermore, the given ID *must* begin with a letter. Thus, if ident is declared to be an ID attribute, then the following examples are all **illegal** XML:

```
<para ident="456">
<para ident="D 973">
<para ident="D/5678">
```

In the first example, the ident attribute does not begin with a letter. For the second it contains a space -- a character that is not a letter, digit, full-stop, dash or underscore. In the third illegal example it contains a slash character -- again a character not from this restricted list.

Thus, with the attributes of type ID we have met the first of our two requirements to support references within a text: in an XML document we can use an attribute of type "ID" to attach identities to elements that we can refer to elsewhere. If the attribute is of type "ID" then the XML processor will guarantee for us that the IDs we code must be unique -- and will notify us if by accident we use the same ID name in two places in the same document.

Having now provided a mechanism to uniquely identify elements in XML, we need a mechanism to refer to them. We can make an attribute refer to an ID by asserting it to be an "IDREF" type attribute. Suppose, for example, that in your DTD you can refer to an element with an ID by using the "refer" tag:

```
See <refer ident="X35">here</refer> for the list of bishops of
Rochester...
```

Note the use of a `refer` element around the textual work "here". The `refer` tag has an attribute `ident` which points to another element in the document -- in this case the element it refers to must have the ID of "X35".

In the DTD we would find the ident attribute to be declared thus:

```
<!ATTLIST refer ...
ident IDREF #REQUIRED
```

Here, the DTD has asserted that the ident attribute is required whenever the refer element is used, and that its value must be an ID that appears somewhere in the current document. Again, your XML processor will verify that the IDREF attribute is being used correctly -- it will check all the IDs it finds in your document for you and complain if the IDREF you specify here is not among them.

In summary then, the ID and IDREF attributes in XML provide a way to identify and refer to elements in XML documents. Your XML processor will help you ensure that attributes declared of type ID and IDREF correctly by ensuring that:

- ID names are legal within XML rules (they are made up of letters, digits, full-stops, dashes and underscore *only*,
- any particular ID name occurs only once in a particular document, and so can only possibly refer to the element to which it is attached, and
- IDs provided within IDREF attributes do, in fact, occur within a document. The IDREF test your XML processor provides automatically ensures that the reference is to something in the document that is, in fact, guaranteed to be there.

## 7.3 IDs in XSLT

One use of ID and IDREF attributes in an XML document is to transform them into <a `name...`> and <a `href...`> elements in HTML. If, for example, one had text marked up using the `para` element with the `ident` element describe above, and the `ident` element was required, then the following template would handle the translation into HTML, using the specified ID field `ident` to provide an HTML name:

```
<xsl:template match="para">
<a name="{@ident}"><p><xsl:apply-templates/></p></a>
</xsl:template>
```

The contents of the paragraph tag would be inserted inside an HTML p element. The p element would be surrounded by HTML's a name attribute, and the value of the `name` attribute would be taken from para's `ident` attribute.

This template would take XML code <`para ident="X46">...` and transform it into <a `name="X46"><p>....` Furthermore, if the XML ident attribute was an ID attribute and also required the XSLT template writer could be assured that every paragraph have attached to it a unique HTML name that would allow it to be referenced from elsewhere in the document.

The reference tag `<refer ident="X46">` would be processed using the XML template:

```
<xsl:template match="refer">
<a href="#{@ident}"><xsl:apply-templates/></a>
</xsl:template>
```

This template would transform the XML code shown above into the HTML: `<a href="#X46">...`. Here again, the XSLT designer could be assured that the XML test for IDs and IDREFs would guarantee that the link generated by this code would have to link to somewhere -- that the link could not be missing.

Exercises 2, below, make use of ID and IDREF-like attributes to make references between different positions in a document. Exercise 4 makes use of ID-like attributes in a document but does not need IDREFs from within the XML document itself -- instead asking you to create an XSLT template that generates an table of contents HTML document.

## Exercises

**Ex. 1.** Load up an excerpt from the *Prosopography of the Byzantine Empire*'s (PBE's) "Persons File" stored in `chap 2 > exercises > pbe-excerpt.xml`. This file was produced as a part of the work involved in producing the PBE's first published CD-ROM. You can read more about this project at `http://www.pbw.kcl.ac.uk`. As a part of the extraction of material from a database, and the presentation of it for the CD, we first put all the material into "Persons File" and then generated all the material to go on the CD from this file. Thus, the PBE I Persons file was an XML document that contained all the information about all the people that went on this CD-ROM. We have provided only a small excerpt from the full Persons file that was produced. In this excerpt, you will find pieces of information about only 4 Leos -- all short entries compared to many for the CD-ROM.

The data about each person is structured the same way:

- Information about each person is included in a `person` element.
- The first child of the person element is always the nn element which contains the person's "name" (actually a name and a number -- used to separate people with the same name). "nn" is an abbreviation for "name-number".
- material extracted from the relational database maintained for PBE appears after the nn element in the `details` element.

  The `details` element is structured as a set of elements that contain different kinds of information about the person. In most cases the name of the element containing the information makes it clear what the item means. Of the two that may not be immediately clear, "`flor`" stands for *floruit* (where a person's *floruit* is the period the individual is thought to have lived), and "pmbz": a reference to the ID of person or persons from another project (called "PmbZ") that correspond to this person. Note that in several of these elements, there may be more than one value. Note as well that some of the elements have attribute "type" which contains further useful information to display.
- Following the details element is an `article` element that contains a brief article written by the researcher who researched and prepared the PBE I materials - John Martindale. The article text is made up or one or more paragraphs.

The meaning of tagging in the article is generally obvious except for three of elements:

- **<GL>**: is an element that provides a reference to a glossary entry. The ID attribute links to the glossary entry. The contents of the element is the displayed text. For this exercise we are not providing you with access to the full PBE glossary, so simply here display the glossary entry in some way that highlights it.
- **<X>**: is an element that provides a reference to another person. We will deal with this later in the exercise.
- **<grk>**: is an element that surrounds bits of Greek -- shown here in a coding convention for Greek called "BetaCode". In the PBE I CD-ROM these bits of greek were transformed so that they appeared in a proper Greek font. We will not ask you to take the BetaCode Greek and display it using a greek font: this is outside the needs of this exercise. Here, simply highlight the Greek in some way.

Create an XSLT stylesheet that will transform this material into HTML so that it is displayed in some suitable fashion. Use the `xsl:value-of` element to get the value of the office type into the display. You can, if you like, start by copying the empty template XSLT document to get you started: `chap 2 > examples > template.xsl`.

**Ex. 2.** In the article for Leo 128 you will find a reference to Leo 220, and *vice-versa*. For example (in Leo 128's article):

```
(see <X d="5035">Leo 220</X>)
```

refers to Leo 220 by providing, as X's d attribute, a number that corresponds to the number in the "id" attribute of Leo 220's person element. Using the attribute techniques we talked about today, transform the `person` element found here so that it includes an "`<a name=...`" tag, so that each person can be reference from elsewhere. Next, take the "X" element, and define an XSLT template for them that changes them into links that link to the corresponding person element: "`<a href=...`".

**Ex. 3.** Use the techniques we learned today to reverse the *details* and *article* display order -- so that the article appears before the material in the details element. The easiest way to do this is to use the `xsl:apply-templates select` attribute twice inside the person template -- one to specify the position of the article material, the second to specify the position of the details.

**Ex. 4.** In the real PBE materials the list of persons numbers over 8000. Clearly, a single large document containing the material about all 8000 people would be cumbersome to use -- it would be much better if the material for each person was presented on the WWW in its own HTML document. We will find out next chapter how to use XSLT processors to automatically generate many separate documents from one large source, and this is, indeed, part of the way to deal with such a large collection of materials. For now, however, we will deal with a different issue that arises when one has a large collection of materials to present -- generating an index or table of contents.

Write a *second* stylesheet that is meant to be applied against *the same* text that generates a list. This second stylesheet will generate HTML showing only the *name-number* of all people who appear in the document. For this sample document, the resulting list will be the 3 Leo's. Define the list so that each item in the list in this second document links to the appropriate Leo in the main document. Hint: to do this stylesheet you will need to remember how to suppress material from elements. We introduced this in the last chapter's exercise materials.

Note the role of ID-like attributes (why cannot we not call them, in fact, XML IDs?) in this and exercise 2. The ID attribute uniquely identifies elements in the file, and can be used something like a database key to refer to them.

# Chapter 3

# *Predicates*, *Modes* and processing Multiple Documents

## 1. Predicates

A *predicate* in an XPATH statement acts like a "filter". Think of a water filter. It takes dirty water in, filters out substances, and lets cleaner water through. The filter has, let us say, a kind of criteria against which it measures the things it is given -- the materials it is given which are "OK" it lets through, the materials it is given which are "not OK" it stops from going through.

An XPATH *predicate* selects from a list of things (not dirty water this time, of course, but parts of an XML document) it is given in the same sort of way, rejecting some that don't pass its test, and letting though others that do. It takes as its source nodes that have been selected with the kind of XPATH expressions we have dealt with up to now (say, all "P" elements, or all "P" elements whose parent is a "DIV2" element). The predicate then takes this list and applies a specified test to each node (generally, element or attribute) in turn. Those nodes that pass the predicate test are included in the final result. Predicates are written in square brackets ('[' and ']') and are coded so that they follow the set of items that they need to filter. We will see some examples shortly.

As you do more XSLT coding you will find predicates useful in many situations. However, two of the most commonly used applications of predicates are:

- when one needs to select elements based on particular attribute values; and
- when one needs to select elements based on their position in their list.

We will give you some examples of using predicates in both of these ways. There are examples of predicates later in this tutorial that makes selections based on other kinds of criteria.

### 1.1 Using Predicates to select based on attribute values

Suppose your text has certain paragraphs in it which are meant to be restricted to a particular audience. The XML design for your text uses an attribute `AUDIENCE` for this purpose. In the rendering you are now building you wish to show the paragraphs that are reserved for the AUDIENCE of "Instructors" in red. For this you would use predicates in the following way:

```
<xsl:template match="p">
  <p><xsl:apply-templates/></p>
</xsl:template>
<xsl:template match="p[@audience='Instructor']">
  <p><font color="red"><xsl:apply-templates/></font></p>
</xsl:template>
```

Look at the path specifications provided in the two `match` attributes. The first one, which does not contain a predicate, is used to match all paragraphs. It simply transforms the paragraph into an output HTML `P` element.

The second one has a more complex match criteria. The "p" at the front selects all paragraphs and puts them into a set -- the same set of nodes as before. The material in the square brackets is the *predicate* and provides a test to be applied to each of the nodes in this set of "p" elements. For each `p` element node selected by the XPATH "p" path, it tests to see if its `audience` attribute is assigned to be 'Instructor'. The `p` elements that do have the `audience` attribute set in this way are included in the result of the template's matching criteria, and will be processed by this template, resulting in HTML that displays the text in red.

When both these two templates are present in a single XSLT stylesheet what happens is that those nodes that match the second (more complex) match test are transformed by it, and only those nodes that match the first test, but *not* the second are transformed by the first template. This in spite of the fact that, of course, for those nodes where the `audience` attribute *is* set to "`Instructor`", the test for *both* templates (both the one with and without the predicate) applies -- not just the one that uses the predicate. How does the processor know to select the right one (the one involving the predicate test) to apply? Well, this is a question of what is called in XSLT *priority*, and there are complex rules provided in the XSLT specification to assign priorities to templates. If more than one template might apply to a particular element in the input text, the processor needs to calculate the priority associated with each possible template and select the one with the highest priority score in order to choose what to do. In general, the priority rules provided about how to calculate these priorities automatically make XSLT processors "do the right thing" in situations like the one we have shown here. However, there are ways to set the priority for your templates yourself, and we will not be talking about how to do this here. There is more information about XSLT priorities in the standard at **[XSLT §5.5]**

The predicate notation in the square brackets looks somewhat like how one would code the `audience` attribute in an element's opening tag, but it is interpreted here quite differently. It is important to notice the syntax differences:

- In the predicate formula, the attribute name is announced with the XSLT "@" operator. As you might recall, the "@" operator tells XSLT that it should match an attribute rather than an element.
- The value to test for is surrounded by single quotes, not double.

Furthermore, although the equal sign is used in the sample predicate shown above, other symbols can be used for other tests. We use the equal sign here to specify that we want a test for equality. However, a predicate test can also test for a value of an attribute based on something other than equality:

- `p[@audience != 'Instructor']`: would select p elements who have a value for their `audience` attribute which is not instructor: note the use of "`!=`" to specify *not equal to*.
- `p[@WIDTH &gt; 40]`: would select p elements who have a value for their WIDTH attribute which was greater than 40. Note the use of the entity `&gt;` to represent ">" (which is interpreted here as the mathematical operator "is greater than"), since the character ">" itself should not be coded in an XML document as part of a value in an attribute, and such a predicate in an XPATH expression would be in an XSLT attribute such as the template "match" element.
- `p[@WIDTH &lt; 40]`: would select p elements who have a value for their WIDTH attribute which was less than 40. Note the use of the entity `&lt;` to represent "<", which is not allowed in XML as a part of a value in an attribute.
- `p[@WIDTH &gt;= 40]`: would select p elements who have a value for their WIDTH attribute which was greater than or equal to 40.
- `p[@WIDTH &lt;= 40]`: would select p elements who have a value for their WIDTH attribute which was less than or equal to 40.

In the first example we've seen predicate tests used to check for a value of an element's attribute. However, predicate tests are really XPATH expressions in their own right and we can use other XPATH constructs to construct test for many other criteria:

- `p[@justify]`: This test is applied against p elements. It is true (and the particular p elements for which it is true would be included in the result) when the p element has a `justify` attribute of whatever value.
- `p[i]`: This test is also applied against p elements. It is true for p elements which contain an `i` element as a child.

  To understand how this works it is useful to realise that the specification of `i` as the predicate is interpreted as an XPATH path in the same way as it is in an `xsl:apply-templates` select statement -- as applying to a current node. The XPATH processor will take each p element it finds in turn as the current element and then apply the predicate test to each one. The test here is an XPATH path statement that selects `i` elements that are direct children of the current p element. If there are any, the predicate test will be *true* and that p element that contains them will be included in the result. If there are not any, the predicate test will be *false* and the current p element will not be included.

  It is important to note the difference between this test and the XSLT pattern "`p/i`". In the predicate test p elements are selected. In the `p/i` test, `i` elements are selected.
- `p[.//i]`: This test is true for p elements which contain an `i` element as a descendent.

## 1.2 Using predicates to select based on position

A second common use for predicates is to select elements based on some sort of position within the list of selected items. Here is an example. Recall the PBE XML example you worked on in the exercises for chapter 2 (), and the list of PMBZ numbers that appear in all the person elements it contains:

```
<person id="4943">
[...]
<pmbzs>
<pmbz>4384</pmbz>
<pmbz>4490</pmbz>
</pmbzs>
[...]
</person>
```

The PmbZ numbers were, as you recall, IDs used to identify the same person in PBE's sister project in Berlin. In this example, the PBE researchers thought that material they had read referred to one person, whereas the PmbZ project thought it referred to two people. Thus, PBE's person with ID 4943 corresponds to two PmbZ's persons: 4384 and 4490.

Since the amount of data is quite short -- numbers only -- it would make sense to display this in HTML as a simple comma-separated list of numbers rather than, as we did in chapter 2, items of bulleted list:

4384, 4490

However, this turns out to be slightly more difficult to do than one might first expect. The difficulty has to do with the comma separator -- there are no commas in the input text so they have to be generated by XSLT. However, whereas there are 2 numbers, only 1 comma needed. You need to be able to tell your XSLT processor to generate the comma after each number *except* the last one (if there is only 1 number, then the first one will also be the last one and no comma is needed at all).

XPATH and XSLT provide a number of things called *functions* that allow you to perform special operations on data, and in some cases to get at certain kinds of information about the data you are processing. Two of these *functions* are useful when this kind of task needs to be done:

- `position()`: This function reports on the position of the current node in the current selected list.
- `last()`: This function tells you the number of the last node in the current selected list.

We can use these two functions to formulate a test for the `pmbz` items in the `pbmzs` element. If the position of the node we are currently processing is the last node in the list (`[position()=last()]`), we don't want a comma following it. Otherwise, we do want a comma to follow the value in the element. This leads us to write two templates:

```
<xsl:template match="pmbz[position()=last()]">
<xsl:apply-templates/>
</xsl:template>
<xsl:template match="pmbz">
<xsl:apply-templates/>,
</xsl:template>
```

The first of these two templates is the more specialised one: selecting only those `pmbz` items which are the last ones in their respective sets. Here the content of the template is only an `xsl:apply-templates` element, meaning that the content of the `pmbz` element will simply be copied in to the output exactly as it is. The second of these two templates is used for other items -- all of them except the last one. For them we *do* want the comma to follow the number, and here the content of the template shows a comma following the `xsl:apply-templates` element.

*(Note that here again, for some selected pmbz elements both templates match -- here again XSLT priorities will ensure that the correct one (the one with the predicate where it applies, and the one without the predicate otherwise) will be applied.)*

Now all that remains is to provide a template which puts this generated list of items into a proper context -- to display them in an HTML paragraph, say. At first glance, it would appear that this is very simple to achieve, in addition to the two templates that process the contained *pmbz* elements we provide a template for the *pmbzs* element:

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

Since the `pmbzs` element in the input document always apparently only contains `pmbz` elements, it would appear that the `xsl:apply-templates` element in the template will, when used in this template, only find *pmbz* elements, and so this will result in sets of pmbz elements only, which is, of course, exactly what we would want. However, if this is tried with the excerpt PBE text we have, one will find that it does not work correctly, we see "4384, 4490," generated -- with the comma appearing at the end. The template for the last item does not seem to affect the last pmbz element at all. Why?

This problem is an example of a more general issue in XML documents called *white space*. Take a closer look at the little excerpt from the PBE file shown above. Each pmbz node appears on a separate line, and that means that between each pmbz node is the newline character. This new-line character is recognised by XSLT as a text node, even though (because it contains only non-printable characters) it is invisible to the eye. XML documents often contain bits of text that contain only white-space characters (spaces, tabs, new lines), and it is easy to forget that these white-space nodes are there. When an XSLT processor is processing it the `xsl:apply-templates` element in the `pmbzs` template does generate a list of child nodes -- but they include some items that we don't want to include -- the "white-space" nodes that contain the newline characters that are, in turn, present because the pmbz items are each on a separate line. When we ask our processor to simply select *all* the nodes that are in the contents of the `pmbzs` element -- as the `pmbzs` template's `<xsl:apply-templates/>` does -- these whitespace/new-line character nodes are included in the list to be processed by the two `pmbz` templates, as follows:

```
[whitespace node: newline character]
<pmbz>4384</pmbz>
[whitespace node: newline character]
<pmbz>4490</pmbz>
[whitespace node: newline character]
```

The last whitespace node arises because there is a newline character between the last *pmbz* element and the closing *pmbzs* element. Because it is there, our test `[position()=last()]` never works properly because the last *pmbz* node is never, in fact, the last node in the list of nodes the `xsl:apply-templates` element selected -- it is the second last instead.

The problem is easily fixed however by making the `xsl:apply-templates` element in the `pmbzs` template more selective:

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

Now, the processor is asked to select only *pmbz* children of the *pmbzs* node and to ignore any other children. The selected list now becomes:

```
<pmbz>4384</pmbz>
<pmbz>4490</pmbz>
```

and we get the results we expect.

Tests based on position are surprisingly common, and so an abbreviated notation is available for them. If the predicate is only a number, or the `last()` function, the assumption is that the test is to match the position of the current node in the selected list to the given number. Thus

- `[1]` is equivalent to `[position()=1]` (to check to see if the current node is the first node in the selected set. The effect is to select the first node in the set.
- `[last()]` is equivalent to `[position()=last()]`. This will select the last node in the set.

### 1.3 Predicates and functions in XSLT and XPATH Standards

In this section we have introduced both *predicates* and *functions*. There is a great deal more than could be said on both subjects -- for *functions* in particular. Because we have just scraped the surface on the topic of *functions*, you will find that the information about functions in the XSLT and XPATH standards may be particularly difficult to understand -- *functions* provide a large range of kinds of capabilities in addition to those we have touched on here. However, here are some references to the XSLT and XPATH standards documents that provide more information:

- Examples of predicates, including some more sophisticated ones than what we have described here, are provided in the section of XPATH paths in both documents: **[XSLT §5.2]** and **[XPATH §2,5]**
- A complete list of XPATH's functions are given in **[XPATH §4]**, but you will find it hard going to understand many of the functions provided until you have more experience with the building of XSLT. The section of functions that deal with the position of items in a list (including `last()` and `position()`)is given in **[XPATH §4.1]**

## 2. Modes

XSLT provides a facility called *modes* which allows you to process a node in more than one place, producing different results in each place in which it is processed. In effect, it allows you to combine two stylesheets -- one doing one sort of transformation on a document, the other doing something quite different with the same input material -- into one stylesheet and generating one output document. Although information about modes does appear in the XSLT standards document (**[XSLT §5.7]**), there is very little information there, and from the standards document alone it would seem very difficult for even experienced developers to see how modes are meant to be used.

Here are two examples of places where XSLT modes are helpful:

1. These handout documents are created in XML and converted into HTML documents using XSLT. You will note that the web version of each handout document begins with a summary containing the titles of the major divisions in the document, and, of course, these same titles are used elsewhere in the document to head the sections to which they belong. So, there are *two* uses for each of these headers in a single output document -- one in the "table of contents" section at the front of the document, the other in the body -- and the formatting of the two uses is different.
2. In the one of the exercises for chapter 2 you were asked to create two separate stylesheets for the same PBE XML materials -- one to generate a single document that contains all the information about the persons, and one to generate a table of contents. Here, although more than one output document is generated, it is possible to use modes, combined with "multiple output" tools included in XSLT processors (introduced in the next section), to generate from a single input document and single stylesheet more than one document as a result, and indeed to use the same input material differently in different output document. We will revisit this example in this context soon.

You might have noticed that both of our examples involve the generation of "table of contents" material, and, indeed, the perhaps most frequent use of modes in XSLT is to allow one to generate a table of contents. However, once you master modes you will find that they are useful in other situations as well.

Here is a worked example which will make modes clear.

Suppose you wanted to process the PBE document we used for our chapter 2 exercises and include, at its top, the list of names that were found in this document as a table of contents. You would want to, in effect, take the stylesheet that did the main body of the display and the separate stylesheet that generated a separate table of contents document and put them together. What makes the combining particularly difficult without modes would be that you want to do two *different* things to the *same* elements in the two *different* situations:

- in the main part of the output document the nn element should generate a heading that announces the beginning of information about a new person -- exactly as we have been doing in earlier processing with this document;
- in the table of contents, however, the same nn element needs to contribute an item in a list in the table of contents.

As you may recall, as part of the chapter 2 exercises you did create two different stylesheets -- one to produce the main result, and the other to produce a table of contents. Here are the fragments in these two stylesheets which deal with the `person` nn element in the two stylesheets -- almost exactly as you would have written them then, except the TOC stylesheet would not need to name the second, main, file, since it is going to be included at the top of the main file:

**Main Stylesheet** **TOC Stylesheet**

```
<xsl:template
match="person">
<a name="pr{@id}"></a>
<xsl:apply-templates/>
</xsl:template>
<xsl:template match="nn">
<h3><xsl:apply-templates/
></h3>
</xsl:template>
```

```
<xsl:template
match="person">
<a href="#pr{@id}">
<xsl:apply-templates
select="nn"/></a>
</xsl:template>
<xsl:template match="nn">
<li><xsl:apply-templates/
></li>
</xsl:template>
```

The *Main Stylesheet* inserts an HTML <a name= tag into the text at the beginning of each person so that other tags elsewhere may refer to this particular spot. Then the nn template inserts the person header -- an HTML H3 element. The *table of contents stylesheet*, on the other hand, does something quite different with the same two elements. The person template takes the id stored with the person and turns it into a <a href= element, putting the "#" in front so that it will refer to the a name assigned by the other stylesheet. It then selects only the imbedded nn element to ensure that for the table of contents display the other person data is ignored, and turns the one it finds in the current person item into a list item in a list. In short, although the two templates produce quite different kinds of output, they work on exactly the same elements. For this reason they cannot, as they are shown here, be put in the same stylesheet since they provide two different specifications for exactly the same template match. We really want to process them both -- using the main stylesheet processing in the main body of the output document, and the TOC stylesheet at the top of the output document only.

The first step to achieving this is to identify one of these two specifications as a special "mode". We do this by giving the mode a name, and then adding the mode attribute specifying this name to those templates that belong to that mode. The Table of contents stylesheet has fewer templates in it, so we'll apply the name "TOC" to the templates that generate the table of contents items. You can see the places where it appears in the following example easily because we have emboldened them here so you can readily spot them:

```
<xsl:template match="person" mode="TOC">
<a href="#nn{@id}"><xsl:apply-templates select="nn" mode="TOC"/></a>
</xsl:template>
<xsl:template match="nn" mode="TOC">
<li><xsl:apply-templates/></li>
</xsl:template>
```

Note that the mode name "TOC" appears not only in the opening xsl:template tag, but also in the xsl:apply-templates tag in the person tag. It is there to ensure that the processing of the contained "nn" item is done only with the correct *TOC* template.

Now that the mode specification differentiates them we can put all four templates in a single stylesheet with the templates that process other materials in the document. What remains is to make sure that both the standard and the TOC stylesheets are invoked appropriately. The table of contents must appear near the top, before all the other items are generated. We want the HTML in the output document to look something like this:

```
<html>
[...]
<body>
<h3>Table of Contents</h3>
<ul>
<a href="#pr4937"><li>Leo 122</li></a>
<a href="#pr4938"><li>Leo 123</li></a>
```

```
<a href="#pr4943"><li>Leo 128</li></a>
<a href="#pr5035"><li>Leo 220</li></a>
</ul>
<hr>
[... main display of data about PBE persons follows here]
```

Clearly, the list items are those generated by the templates that belong to the *TOC* mode. To make this happen then, we need to insert in the template that generates the overall structure of the document an `xsl:apply-templates` element that invokes the TOC mode explicitly at the exact place where we want these TOC items to appear:

```
<xsl:template match="persons">
<html>
[...]
<body>
<h3>Table of Contents</h3>
<ul>
<xsl:apply-templates select="person" mode="TOC"/>
</ul>
<hr/>
<xsl:apply-templates/>
</body>
</html>
</xsl:template>
```

*persons* is the document node in the input XML document, so this template will be generating the beginning and ending of the output HTML document. Note the *two* `xsl:apply-templates` elements it contains. The first one, placed after the *Table of Contents* heading and inside an HTML `ul` element, selects the child person elements of persons, and uses the *TOC* mode templates to process them. The second one, placed after the HTML `hr` element, will invoke the templates that have no explicit mode assigned to them, and will generate the main body of the results.

An XSLT processor, when it sees this persons template will copy to the output document the fixed HTML that is provided, and will insert the results of template transformations in the places marked with the `xsl:apply-templates` element. For the first one, the processor will note that it is to use *only* the templates marked as being a part of the TOC node to generate materials to insert; and these will, as we know, generate the table of contents entries. When the processor comes to the second `xsl:apply-templates` it will note that no mode was specified, so the rest of the templates (those with no `mode` attribute) will be used to generate the body of the document.

## 3. Multiple Documents

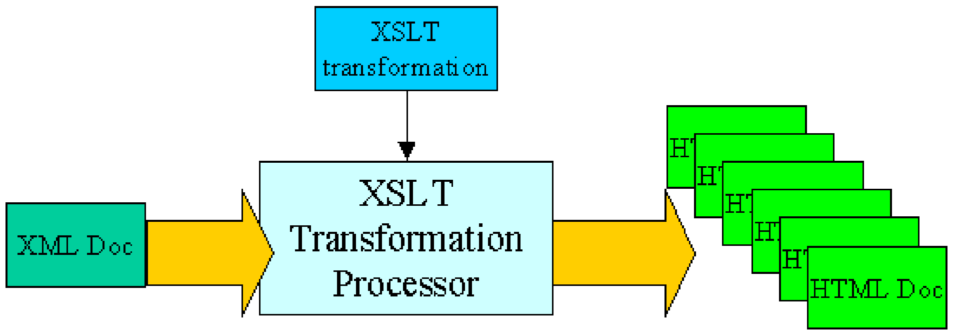
XSLT is often used in situations where *a large amount* of XML material will need to be processed. When you are converting a large amount of XML material, two symmetric situations are likely to arise:

1. The large amount of material you are processing must be split into a number of separate HTML documents, so that none of them are too big to be loaded quickly over the internet into a browser; and/or
2. the input XML material you need to process is, in fact, divided as well into a large number of smaller, separate, documents that must be processed as a group.

In this section, under the heading of "generating multiple documents" we will address issue 1 and under the heading "processing multiple documents" we will address issue 2.

## 3.1 Generating Multiple Documents

A schematic showing what we will achieve here is shown below:



Here, a single XML document is given to an XSLT transformation processor. The transformation stylesheet causes the processor to break the content of the single XML document into multiple HTML documents.

Notice how useful this kind of thing is. Recall the PBE XML material we worked with in chapter 2. We provide an excerpt in containing 4 individuals. For the real PBE file, however, there are 8000 people in this single XML document. We know how, indeed we already have, written an XSLT stylesheet that would put all the information about all 8000 people in a single HTML file. Unfortunately, it is not appropriate to put all the HTML for all these 8000 people in one file -- the resulting file would be huge and would be too slow to serve over the Internet. The sensible way to proceed would be to instead put information about each person in a separate HTML file -- about 8000 documents in all! By having an XSLT transformation that divides the output into separate HTML documents by person it meant that the transformation stylesheet user need only invoke the XSLT processor *once* and generate 8000 documents in, as it were, one stroke. To generate the 8000 documents separately, by invoking the processor 8000 times would have taken many hours of tedious work. To generate the 8000 this way takes only a few seconds.

### 3.1.1 Generation of multiple output with XSLT 1.0

We have described the kind of process that might be used for a typical project, but in practical terms how do we go about this? You may recall us telling you that XSLT 1.0 lacks certain key features that are commonly needed when processing XML documents. Unfortunately, producing multiple output is one of those key features!

The alternatives are listed below (in reality all function in similar ways, although some are easier to implement than others):

*Don't use XSLT 1.0!* In other words, use XSLT 2.0. This is without doubt the solution in the long term, but at the time of writing this XSLT 2.0 was still not an official recommendation (although there will probably not be long to wait). *Use the `xsl:document` features introduced with an interim 'Working Draft' version of XSLT called XSLT `1.1`.* Although it never became an official recommendation, this option became quite popular due to its inclusion in Michael Kay's seminal *Programmer's Reference* for XSLT 1.0. We merely mention it in passing here, and do not recommend you go this route. *Use an extension element defined by the EXSLT initiative*. The extension mechanism is almost identical to the `xsl:document` option we have just mentioned, but is more standards compliant. For technical reasons we have chosen not to take this approach, although the EXSLT is a very interesting initiative. *Use a processor-specific alternative.* In other words, use an XSLT instruction that only works with a specific processor. Normally, we would discourage you strongly from using anything that relies on one particular piece of software, as this is both dangerous and goes against the principles of technical standards and interoperability. However, in this case, it is the most practical alternative until

XSLT 2.0 has 'become law' (and it would be easy to convert this approach to the XSLT 2.0 approach).

Within the final option, which is the one we will use on this course, there are various alternatives, depending on the XSLT processor used. Each processor provides its own syntax for producing multiple output, but we will describe how to do it for one processor (*Saxon*) in some detail.

### 3.1.2 How to produce multiple output for this course

The approach we are going to describe uses the processor *Saxon*, but could easily be edited to work with a different processor by simply replacing the XSLT instruction to work with another XSLT instruction using a slightly different syntax. Now we will describe the general principles, before applying them to a real example. We should start with a general example first, showing a snippet from an XSLT file:

```
<xsl:template match="myElement">
<saxon:output href="d:\user\xxxx{@yyy}.html">
<html>
<head>
<title>title text here: <xsl:value-of select="zzz"/></title>
</head>
<body>
<xsl:apply-templates/>
</body>
</html>
</saxon:output>
</xsl:template>
```

In this example, you will note the following:

There is a standard XSLT instruction in the form `xsl:template`, which is set up to process the element called *myElement*. There is a new element, called `saxon:output` that appears inside the `xsl:template` element but which functions as a wrapper around the HTML code. In turn, this contains the basic HTML code required to produce an HTML page Within the HTML code, there are two instructions: one of these affects output within the HTML <title> tag, while the other produces output within the HTML <body> tag.

Now let us examine what is happening here in more detail. The first thing to say, is that the `saxon:output` instruction does not begin with the customary `xsl:` prefix, but rather `saxon:`. We will explain why, and what else we need to change in the document to make this work, in a moment.

If we decode the XSLT snippet above further, we can see that:

We first select an element in the source XML in the customary way using `xsl:template`. We then ask that for each instance of the element selected, a new HTML document is output using the `saxon:output` instruction. The name of each file is dictated by what appears in the `href` attribute. In this case, we have chosen to create each name as follows: Firstly, by adding the text string 'd:\user\xxx'. In other words, all the output files will appear in the path *d:\user\* and the filenames will start 'xxx...' It is not essential to specify the path, but we **strongly** recommend that you do so, so that you can control where files get output. (If you were working on your own machine, you would probably replace *d:\user\* with another path such as *c:\user\* that used the 'c' drive, but whenever you work on PAWS machines you will usually be working on the 'd' drive.) *Note the direction of the slash: you need to use a backward slash- a forward slash will not work here.* Then by adding information extracted from the source XML file. Using the curly brackets syntax, we effectively ask that the *yyy* attribute of each *myElement* element be used. If we didn't use some mechanism to distinguish between the naming of each output file then only one file would actually be output! The simplest way of doing this is to use information in the source XML file, preferably something like an 'id' attribute that is likely to be unique and likely also to help us to conform to basic filenaming good practice, e.g. no spaces in filesnames. The text string '.html'. Note that the processor does not automatically add the file extension. We recommend you use the extension '.html' as opposed to the extension 'htm', although both work. We then ask the processor to output the basic HTML code needed for any web page. Within the <title> tag of the HTML element we ask that some generic text be included, followed by information held inside the 'zzz' tag. Within the <body> tag of the HTML element there is an `xsl:apply-templates` instruction. In other words, any XSLT *template* instructions that apply to elements lower down the hierarchical chain will be processed and the output placed here.

There is one more problem to solve. At the beginning of this course, we briefly discussed the concept of *namespaces*. This is a complicated subject, and a comprehensive examination of namespaces goes far beyond the needs of this course, but it is helpful to revisit it briefly. You will remember that we introduced namespaces to explain how an XSLT processor could cope with tags from two (or more) different structures. The example given then was the distinction that needs to be made between HTML tags and XSLT tags, and you learnt that in order to use XSLT tags we had to do two things:

*'Declare' the namespace.* We did this by including an instruction within the outer `xsl:stylesheet` element. *Use the relevant namespace prefix (in this case `xsl:`) on any XSLT elements we use.*

We need to do something very similar when we use the `saxon:output` element, as this uses a new namespace, *saxon*. Let us see the example below, which shows a basic but complete XSLT stylesheet that we can use as a template when producing multiple output:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0"
xmlns:saxon="http://icl.com/saxon"
extension-element-prefixes="saxon">
<xsl:output method="html"/>
<xsl:template match="outerElement">
<xsl:apply-templates/>
</xsl:template>
<xsl:template match="myElement">
<saxon:output href="d:\user\xxxx{@yyy}.html">
<html>
<head>
<title>title text here: <xsl:value-of select="zzz"/></title>
</head>
<body>
<xsl:apply-templates/>
</body>
</html>
</saxon:output>
</xsl:template>
</xsl:stylesheet>
```

You will see here that another line has been added to declare the *saxon namespace*. You will also notice that a line has been added below that which describes something called *extension element prefixes.* Do not worry too much about what this actually means- but it is essential that you include this. Once we have added these two instructions, the XSLT processor will understand and know what to do with the occurence of `saxon:output` within the stylesheet body. Because the two instructions have been added to the content of the outer `xsl:stylesheet` element, they will apply to the stylesheet as a whole.

One final word of warning: this is an extension written for Saxon when it is processing XSLT 1.0, and is not likely to work well with versions of Saxon built to process XSLT 2.0. For this reason you should ensure that you use a version of Saxon in the *6.5.x* range. At the time of writing, the only version of Saxon that could be used in Oxygen (as installed on PAWS machines) for this purpose was **6.5.3**.

### 3.1.3 A concrete example: *PBE people*

So now that we've set the stage, we're ready to return to the challenge described earlier- how to begin to deal with the need to generate 8000 HTML person files for the PBE project. First, there is a new, albeit minor, issue to sort out -- since each person is to be in a separate file, we need a file name for each of them. Probably the best thing to use as a base for a filename is the person id. Every person in the full file has a uniquely assigned number attached as the *ID* in the *person* element. For example, in the excerpt we have to work with, the first person in the file (Leo 122) has person ID of "p_4937". The filename that would contain his details could be called "pbep_4937.html". Since each person is assigned a different number, we can be sure that each person will get a unique filename into which his/her data can be put.

The *person* element in the input XML document groups all information about one person into one element. It is also the place where the person ID is provided. Thus, the template that processes the *person* element is the natural place to indicate that a change of document is needed -- it will be invoked once for each person in the input file, and each time it is invoked we want a new HTML file as an output. We will use `saxon:output` to delineate the places where each file is needed. Here is a person template that will achieve this:

```
<xsl:template match="person">
<saxon:output href="d:\user\pbe{@id}.html">
<html>
<head>
<title>PBE Person: <xsl:value-of select="nn"/></title>
</head>
<body>
<xsl:apply-templates/>
</body>
</html>
</saxon:output>
</xsl:template>
```

Each time the XSLT processor finds a *person* element it will generate the output directed by this template. The `saxon:output` instruction creates a new file for each *person* element, and constructs the filename by taking the literal string *pbe*, adding something which comes from the source XML file (hence the curly brackets) and then appending the literal string *.html*. Note how XSLT is told to use the person's ID to generate the file name: it is told to take the XML person element's *id* attribute and insert it between "pbe" and "`.html`".

Now look at the content of the `saxon:output` element. This is the material that will make up the separate HTML document. Since the contents of each `saxon:output` element is going to soon be an HTML file in its own right, an `html` tag appears immediately inside the `saxon:output` element, and inside that is a `head` and `body` element for this person-oriented HTML file. Note as well the use of `xsl:value-of` to get the person element's name-number inserted into the `title`. The contents of the XML `person` element will generate the `body` of this output file, so the `xsl:apply-templates` tag is found as the only contents of the `body` tag in the `person` XSLT template.

If we now use this modified stylesheet against the PBE document we get a series of documents out that looks like this:

```
<html>
<html>
<head>
<title>PBE Person: Leo 122</title>
</head>
<body>
<h3>Leo 122</h3>
<table>
[...]
</body>
</html>
```

This is, of course, only an excerpt from the entire output file (material has been removed where one sees *[...]* ).

If we look at the result we can see that the cross references that have been generated (from the table of contents, now in a separate file; and between different persons) are broken. They need to be changed from being #-references that worked inside a single HTML document, to be references to separate HTML documents. We leave that as "an exercise to the reader".

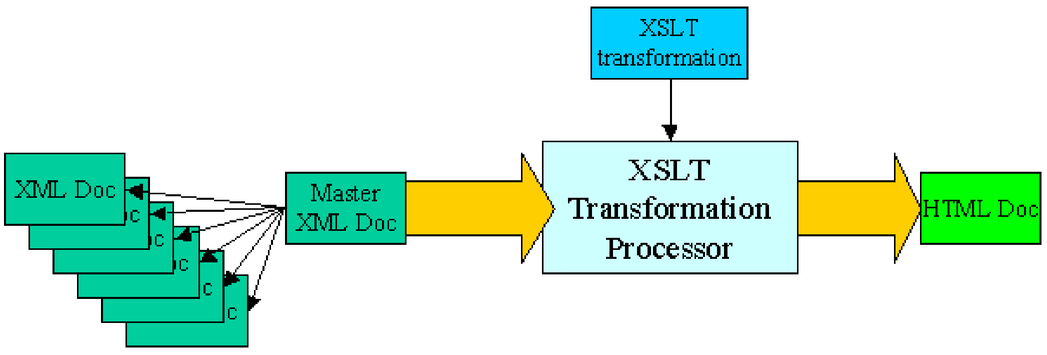
Be aware also that files are output in two different ways using this method:

using the `saxon:output` method to produce multiple documents a single file is produced from the 'main' transformation in the usual manner

When producing multiple documents, the latter will often simply be a kind of 'residue' that can be discarded.

## 3.2 Processing Multiple Documents

Now we turn to look into the opposite problem: taking a number of separate XML documents as input, and treating them as if they were one large single document for processing in a single XSLT transformation. There is more than one way to achieve this, but we will talk about using of XML *external entities* to achieve this end, since it turns out to be, in some ways at least, the most straightforward. XML external entities are not a feature of XSLT at all, but are instead part of the definition of XML itself. We will create a "master XML" document that will make reference to each of the other XML documents we wish to process. When we run the XSLT processor we will point it at the master document we have created, and the XML part of the XSLT processor will, while processing the master document pull in as well the other documents that were referred to in it. XSLT sees the master document, and separate XML documents that it refers to as if they were a single large document. Schematically, this process looks like this:



Here the advantage of doing things this way should be evident. If there were several hundred separate XML documents to be processed separately, one would need to invoke the XSLT processor several hundred times. Furthermore, you would need to take the hundreds of separate documents and manually cut-and-paste them into the single large output document. By allowing your processor to handle all the XML documents in one "go", you only need to invoke the XSLT processor once, and you generate the one, combined, output document in one step.

Suppose, for example, that we have a collection of abstracts for papers to be given at a conference that need to be assembled into a single larger program. Each paper has been given a number, and the name of the file containing the paper's abstract contains that number. Thus, the abstract for paper 10 has the name *abstr10.xml*. The master XML document that will link them together might look something like this:

```
<!DOCTYPE DRH-PGM [
   <!ENTITY ab10 SYSTEM "abst10.xml">
   <!ENTITY ab95 SYSTEM "abst95.xml">
[... include other ENTITY definitions here.]
]>
<DRH-PGM>
&ab95;
&ab10;
[... include other ENTITY references here.]
</DRH-PGM>
```

To use entities to refer to documents outside the master document, one needs to first *define* an entity for each file to be included in the local part of the DOCTYPE declaration. The excerpt from the master XML document shows two definitions (shown in bold), but there would be one definition for each abstract that needed to included. Entity definitions that define an entity by referring to an external file are called *external entities*, and take the form:

```
<!ENTITY entity-name SYSTEM "file-name">
```

where *entity-name* is the name that will be used in the document body to refer to this entity and to mark the spot where it is to be inserted. and where *file-name* is the name of the file that contains the material to be inserted at the marked spot.

Once the entities are defined, they can be used in the body of the document. One refers to an entity by writing its name between "&" and ";", e.g. `&ab10;` to refer to entity *ab10*. This notation may remind you of the entities in HTML that represent letters-plus-accents, e.g. `&eacute;`. The principle is quite similar: `&eacute;` refers to an entity named *eacute*, although in HTML at least the browser has the job of knowing how to handle such an entity so no definition of what *eacute* means is necessary. Here, when the XML processor sees a reference to *ab95*, it can look in the DOCTYPE definition and find the entity defined there as a reference to the file *abst95.xml*. The XML processor's job then is to read in file *abst95.xml* and insert its contents at this designated spot in the model of the XML document it is preparing for the XSLT processor to work on.

Thus, we can see that the body of our master document is particularly straightforward. The references to all the external entities we defined in the document header are listed as part of an element called DRH-PGM. Note that the ordering of the entity *references* is different from the ordering of the corresponding entity *definitions*. This is perfectly acceptable. The ordering of the material as it is inserted into what the XSLT processor will see will be based on the ordering of the references to the entities in the body of the master document.

When the XSLT processor is invoked, it is pointed at the master XML document, rather than at any of the individual pieces that the master document refers to. The XSLT processor will see the master document and its external entities as if they were single large XML document that is, in fact, made up of many smaller pieces. We call such a view that presents what appears to be a single document from what is reality many smaller pieces a *virtual document*.

## 3.3 Multiple Documents in and out

It is not unusual to actually have multiple XML documents that need to be processed as a group, and that in turn generate multiple HTML documents. To achieve this, one simply combines the two processes together:

- One creates a master XML document that uses external entity references to collect all the separate XML documents into a single "virtual document".
- One takes advantage of the multi-document output facilities of your XSLT processor to split the resulting virtual document into separate pieces.

Suppose the PBE project had produced separate XML documents for each of their 8000 people. If one used the master document model to specify a single large virtual document that refers to all 8000 people, and then used the DOC feature to split them apart into separate HTML documents, then one could read the 8000 input documents and produce 8000 separate HTML documents in one invocation -- a considerable saving of time with relatively little additional complexity!

Furthermore, since you have, in the master document, a single virtual look at all the materials as if they were in a single large XML file, you can take advantage of XSLT's abilities to gather data together from across the entire large virtual document to create summary documents like table of contents. The exercise materials today will give you a chance to experiment with materials of this kind.

## Exercises

**Ex. 1.** Today's exercises will centre around texts prepared for the *Modern Poetry in Translation* (MPT) project. You can see an example of a typical MPT file in `chap 3 > exercises > Tr.Fr.Can1.xml`. Open this document in an XML editor so you can see how it is structured.

The project collects information about publications that feature modern poetry that has been translated from its original language into another language. Descriptions are grouped by the poetry's original language (and/or, sometimes, cultural group), and at least at the present time the project is focusing on translations *into* English, although the project expects to add information about other "translation destination languages" when list of translations into English are complete. The example you are looking at contains information about publications that contain poetry translated into English from Canadian French.

The most important elements in the XML document are:

- **MPTGRP**: This is the document element. The `LG` attribute contains a brief description of the Language Group represented in the file.
- **CLASS**: The CLASS element is the first child of the MPTGRP element and always contains a textual description of the Language Group that the file represents in the form of a title suitable for display.
- **INDIV/ANTHOLS**: Following the "CLASS" element are two main divisions. The INDIV element contains information about texts in which the original language poetry is all by one poet. The ANTHOLS element contains information about anthologies -- collections of poems by several authors published as a single volume.

- **AUTHOR**: Inside the INDIV class there are a collection of AUTHOR elements -- one for each author represented. Each AUTHOR element is assigned an ID that is unique within this particular MPTGRP file.
- **AHEAD**: The first child of the AUTHOR element is the AHEAD element which contains information about the author's name and birthdate/deathdate. The elements AS are for the Surname, AF for the Forename, BY for the birthyear, DA for birth/death dates, etc.
- **TITLE**: There is one TITLE element within the AUTHOR element for each title containing poetry by that author. The TITLE element is also used in the ANTHOLS section, although there, of course, it is not contained in an AUTHOR element, since anthologies are multi-author. Each TITLE element is given an ID that is unique within a particular MPT XML document. The content of a TITLE element is "mixed content", which means it contains a mixture of material identified by being inside an element tag, and text that is not further enclosed. The entire material in the TITLE tag is meant to be a description of the title in a form that is suitable for presentation.
- **TL**: The first child in a TITLE element is the title of the poetry collection.
- **TR, ED, EDTR (and others)**: Following the title is information about who edited and translated the poetry. This information is surrounded by tags like TR, ED, EDTR to show the kind of association ("TR"="translator", "ED"="editor", "EDTR"="translator and editor", etc.) the people named therein have to the text. Several other tags can occur when other kinds of associations (such as the person who wrote the forward, for example) are given.
- **PERS**: When a person is identified by on of the tags that identifies a *translator* (within the TR, EDTR or similar elements), the name or person reference is enclosed in a PERS element. The PERS element has two attributes, "f" for the first name, and "l" for the last name of the person. By having attributes where the names are stored, it is possible to have the person reference appear in various forms in the displayed text (e.g. "translated by poet"), and still explicitly identify (in the attributes) who the person is. Also, it is possible indicate that both "John Bradley" and "J Bradley" refer to the same person: `<PERS f="John" l="Bradley">J. Bradley</PERS>`.
- **LA**: The language(s) contained in the translated edition.
- **PB**: Information about the publisher. This information could, at some point, be further structured by identifying subcomponents in it, but this has not been done here.
- **TX and X**: These tags contain information for that the researcher has recorded for the information of the series editor, and are not meant to be seen, at least in this form, by users of the data.

We provide you with an XSLT stylesheet that will transform any of the MPTGRP documents into some presentable HTML: `chap 3 > exercises > mptgrp.xsl`. Take a brief look at it in an XML editor so that you can review what things it does.

You have been provided with a few more sample XML documents (there are actually about 35 different language groups being managed by the project):

- `chap 3 > exercises > Tr.Afrk1.xml`
- `chap 3 > exercises > Tr.Spa1.xml`

These two XML documents also use the markup scheme briefly outlined above, of course. Apply the given stylesheet against more or more of these to help you clarify in your mind what the structure is and the function of the various elements within it.

**Ex. 2.** Using the model we have given you earlier about using modes to generate a table of contents, copy the XSLT stylesheet you've been given and modify your copy so that it displays at its front of HTML that it generates a table-of-contents like section that lists the authors contained in the file. How can you include a reference within this table-of-contents to the anthology section?

**Ex. 3.** Some of the AUTHOR elements in the *Tr.Fr.Can1* text do not contain any TITLEs (see, for example, the entry for *Dany LaFerriere*). Use a predicate to indicate that AUTHOR elements with TITLEs should be processed, whereas those without should be skipped.

**Ex. 4.** Build a master XML document that allows an XSLT processor to view the three MPTGRP files you have been given as a single large text. Now, modify the stylesheet so that the entire text comes out in a single document. Add to this stylesheet another mode to generate a master table of contents that takes the user directly to the language group that interests them.

**Ex. 5.** Now, again modify the stylesheet so that the result is a collection of separate HTML files and a separate master table of contents that contains a list of language groups. Modify your stylesheet so that the master table of contents also contains, as a sublist under each language group heading, a link to the authors that are represented there.

# Chapter 4

# Generating Indices with XSLT

## 1. Introduction

We have looked at tools in XSLT that allow one to extract materials from a more complex document to form type of "summary document" called a *table of contents*. Table-of-contents-like documents always present their material in the same order as the original material. Often, however, it is useful to create a different kind of summary document that presents material sorted differently. We call these type of summary documents *indices*, and in this module will introduce some of the components in XSLT that will help us to create them. We will introduce a number of things in XSLT that allow it to express, in terms of what we have achieved so far, quite radical transformations of the input data. It turns out, however, that at least with XSLT Version 1.0, it is not possible to create a completely satisfactory index with the tools we will have the opportunity to introduce in this course -- although with more advanced features of XSLT version 1.0 it can be done. However, we can get quite well enough along in the time we have available to produce something useful even if it is not entirely ideal.

In this module we will work to generate an index of translators in the *Modern Poetry in Translation* materials we started to work with in chapter 3. It is natural, after all, while working with material that is by its very nature talking about translations, to want to access the material by the translator's name -- so we can both find the names of all the translators that are represented, look through them to find a particular person of interest, then see what s/he has translated. A classic index would do the trick:

```
Hyde, Lewis
  (Spanish)
    Aleixandre, Vicente A LONGING FOR THE LIGHT
    Aleixandre, Vicente TWENTY POEMS
```

Here we see what an entry for translator *Lewis Hyde* would look like. The translator's names would be sorted by surname so that they could most readily be searched by the user. The works a translator had produced would be shown under the translator's name, grouped by original language. Here, Lewis Hyde is shown as having translated two works by Vicente Aleixandre from Spanish. We would complete the index by turning the reference to each translated work into a link that would take us to the main entry for that work in the MPT web pages.

Since the text has references to all translators identified by the PERS tag, it becomes possible to write an XSLT process that locates these and, based on what it finds, generates suitable output. In the process, however, the ordering of the material we derive from these PERS nodes will be presented in a substantially different order and fashion than it is in the original input texts. In the original MPT XML documents, the records are sorted by the *author* of the poetry. If we look at the translators names (tagged with the PERS tag) in document order we can see that they are *not* in alphabetical order. In our translators index, however, the translators names need to be in alphabetical order.

## 2. Multiple input files

We begin by (re)creating an XML file that uses the external entity techniques we presented in chapter 3 to allow us to process a number of separate XML files together in one go. We have three input files to experiment with available to us here. For the real job, of course, there would be many more. To begin, find these document among the bundle of materials you have fetched for this chapter's work, and have stored in `D:\USER`:

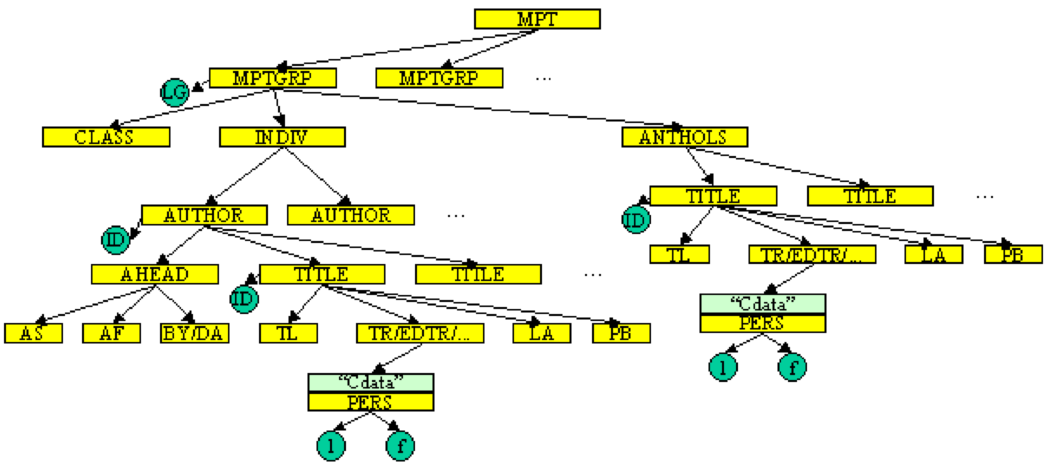
- Original language *Afrikans*: `chap 4 > examples > Tr.Afrk1.xml`
- Original language *French Canadian*: `chap 4 > examples > Tr.Fr.Can1.xml`
- Original language *Spanish*: `chap 4 > examples > Tr.Spa1.xml`

Next, we create another XML file called `mpt.xml` that makes reference to all three -- ordered by the name of their language groups.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE MPT [
<!ENTITY afrik SYSTEM "Tr.Afrk1.xml">
<!ENTITY fr-can SYSTEM "Tr.Fr.Can1.xml">
<!ENTITY spa1 SYSTEM "Tr.Spa1.xml">
]>
<MPT>
&afrik;
&fr-can;
&spa1;
</MPT>
```

Now, by directing our XSLT processor to process `mpt.xml`, we will cause it to see the contents of all three of the main source XML files in one "go".

Before we begin the task of creating an XSLT stylesheet to create an index of translators, it is useful to review the structure of the MPT documents that our stylesheet will be looking at. Here it is, shown as a tree diagram:



In this diagram the boxes represent elements. The three dots (ellipsis) indicates that the preceding element can occur multiple times. The circle represents attributes attached to certain elements. Take a moment now to study the diagram to improve your understanding of MPT's structure, since it will be necessary to understand it well to follow how the index stylesheet is created. You can, of course, refer back to the diagram later if necessary.

## 3. *xsl:for-each* and *xsl:sort*

Now that we have a way to process all the documents in one XSLT pass, we shall begin to create the XSLT stylesheet document by starting with a basic, but initially incomplete, stylesheet and then making it more complex and more complete as we go along. This iterative strategy -- breaking a complex task down into individual managable steps -- is a recommended way to tackle any complex XSLT template development

We will work with a new XSLT construct: the *xsl:for-each* element. It is described in detail in the XSLT specification document at **[XSLT §8]**. Here is an example of its use *in situ*:

```
<xsl:stylesheet
version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>
<xsl:template match="MPT">
<html>
<head>
<meta http-equiv="content-type" content="text/html;
charset=iso-8859-1"/>
<title>Twentieth Century Poetry in Translation: Index of
Translators</title>
</head>
<body bgcolor="#ffffff">
<h3>Twentieth Century Poetry in Translation: Index of Translators</
h3>
<hr/>
<ul>
<xsl:for-each select=".//PERS">
  <li><xsl:value-of select="@l"/>, <xsl:value-of select="@f"/></li>
</xsl:for-each>
</ul>
</body>
</html>
</xsl:template>
</xsl:stylesheet>
```

Note that this template has, as of yet, only one template in it, and inside it, instead of the `xsl:apply-templates` element we are used to seeing, we have an `xsl:for-each` element with some contents (the material is highlighted in boldface above to make it easier to locate this in the above display). The `xsl:for-each` element tells the XSLT processor that at this point in the template it is supposed to generate a list of nodes based on its `select` test (ignoring entirely the rest!) and then, *for each* selected node, it should generate output that matches the `xsl:for-each`'s content. In the above example, the select test is "`.//PERS`" which locates all *PERS* elements in the text (why wouldn't a select test '`select="PERS"`' work?). The *contents* of the `xsl:for-each` element specifies, then, what is to be reproduced in the output document *for each PERS* element that is found. Each PERS node that is selected by "`.//PERS`" becomes in turn the "current node", and hence the two `xsl:value-of` elements contained within it apply to each *PERS* node in turn, and generate, in turn, the last name and first name of the translator by referring to the PERS's `l` and `f` attributes.

*(An aside: we don't really need the `xsl:for-each` element here. How could we use the `xsl:apply-templates` element you already know to achieve the identical result? However, although the `xsl:for-each` element is not needed, one often sees it used in a context like this -- when the stylesheet designer has thought of the processing as something that iterates through a set of selected nodes, as we are doing here.)*

The first refinement we need to make to the above stylesheet fixes a problem in the input text file: sometimes *PERS* tags have been introduced into the text we are working with here, but the person introducing the PERS tag has neglected to provide `l` and `f` attribute values. To properly fix this one should, of course, revisit the text to fix these PERS tags. For the purposes of this example, however, we will change the `xsl:for-each` test to select only those PERS elements that contain an "`l`" attribute, and ignore those that miss this key data. This test can be done using a predicate:

```
<xsl:for-each select=".//PERS[@l]">
```

The `xsl:for-each` element now will select only PERS element that contain a `l` attribute. We hope that PERS elements that contain an `l` element also have an `f`!

When we use an XSLT processor to process this stylesheet we will see results that look like this:

```
<html>
[...]
    <h3>Twentieth Century Poetry in Translation: Index of
Translators</H3>
        <hr>
        <ul>
            <li>Coetzee, A.J.</li>
            <li>Brink, Andre P.</li>
            <li>Leigh-Loohuizen, Ria</li>
            <li>Hirson, Denis</li>
            <li>Hirson, Denis</li>
            <li>Breytenbach, Breyten</li>
[...]
```

In this excerpt we can see the beginning of the list of materials generated by the `xsl:for-each` element as we have it so far. This *is* a start, but of course by itself it cannot function usefully as an index -- we can see the names of the translators, as recorded with each *PERS* element, but not the names of the works they translated.

There is an even more fundamental problem as well -- in an index we would need to have the names presented in *alphabetical order* so that the index user can locate material about the person s/he is interested in. Here they are presented in *document order* -- in the order in which the XSLT processor finds them when it scans the document it has been given to process. It turns out that that this can be easily fixed using an XSLT element we have not yet introduced: it is possible to ask the XSLT processor to *sort* the nodes it selects in an `xsl:for-each` process by including, in the `xsl:for-each`'s content one or more `xsl:sort` elements which specify how material is to be ordered:

```
<xsl:for-each select=".//PERS[@l]">
  <xsl:sort select="@l"/>
  <xsl:sort select="@f"/>
  <li><xsl:value-of select="@l"/>, <xsl:value-of select="@f"/></li>
</xsl:for-each>
```

Here, two `xsl:sort` elements have been inserted into the `xsl:for-each` element's content. They cause the processor, after it has selected the nodes indicated in the `xsl:for-each`'s select attribute, to order those nodes according to the criteria given in the `xsl:sort` element's criteria. By giving two `xsl:sort` elements we are asking for a two-level sort to be performed. The first `xsl:sort` element causes a top-level sort to be based on the values of each *PERS*'s *l* attribute -- to sort by the translator's last name. The second xsl:sort elements causes a second-order sort to be done among those nodes with the same first order sort value -- here to sort secondarily on the value of the *f* attribute -- the translator's first name(s).

Note that the `xsl:sort` element's `select` attribute specifies what data is to be used from the selected node to control ordering. Since it is expressed in terms of an XSLT path, the data to be selected for sorting can be an attribute value (as shown here), element content, or any combination of these two.

Another useful attribute of the `xsl:sort` element that you may often find you need is the `data-type` attribute. If you specify it, it can take on one of two possible values: "*text*" or "*number*", and indicates whether that material selected by the select attribute is to be sorted *alphabetically* or *numerically*. The default action is "*text*" so it does not need to be specified in this particular example we are currently working on. However, if material was to be sorted based on the value of a *number* rather than, as it is here, a *name*, it would be necessary to include the attribute `data-type="number"` in the xsl:sort specification to ensure that the ordering you got was numerical

You can also code the `order="ascending"` or `order="descending"` attribute to sort in ascending or descending order.

The xsl:sort element is described in the XSLT standards document at **[XSLT §10]**.

If we now apply our stylesheet again with the `xsl:sort` elements in place, we will see:

```
<HTML>
[...]
        <ul>
            <li>Alfau, Felipe</li>
            <li>Amprimoz, Alexandre L.</li>
            <li>Amprimoz, Alexandre L.</li>
            <li>Baland, Timothy</li>
            <li>Barnstone, Willis</li>
            <li>Barnstone, Willis</li>
            <li>Barnstone, Willis</li>
[...]
```

*(Continuation of the aside begun above: We have shown you here how to use the `xsl:sort` element within `xsl:for-each` elements to control the ordering of the nodes. `xsl:sort` can also be used in `xsl:apply-templates` by supplying the xsl:sort elements as content within the xsl:apply-templates element:*

```
<xsl:apply-templates select=".//PERS[@l]">
<xsl:sort select="@l"/>
<xsl:sort select="@f"/>
</xsl:apply-templates>
```

*So, even though we need to sort the nodes in a different order, this need to sort does not, by itself, make the use of `xsl:for-each` a true requirement to deal with the generation of this index. If you have the time, you might consider rewriting the stylesheet to use `xsl:apply-templates` instead of `xsl:for-each` to generate the results you see below. Parenthetically, you might note that this is the first time we have suggested content to be included <u>inside</u> an `xsl:apply-templates` element.)*

## 4. Getting more information from *parent* nodes

The names are now in the right order, but without more information about each entry, a user of the resulting "index" would still not find it very useful. As our model showing what we were aiming for makes plain, the index becomes more useful if it also shows the author and title of the works the person has translated. We will move on to address this now, and then (since we are creating HTML documents in which <a `href=...>` hypertext links are possible and desirable) discuss how to generate links to the full entry from each line in the translator's index as the final step.

In trying to include author and title information, the first thing to notice is that each node that the xsl:for-each selects is a PERS node, and the information we now want to display is not from *inside* the PERS node, but from what *surrounds* it. Let us sort out the exact relationship by looking at an excerpt from the input XML text:

```
<INDIV>
  <AUTHOR ID="A1">
    <AHEAD>
      <AS>ALBERTI</AS><AF>Rafael</AF><BY>(b.1902)</BY>
    </AHEAD>
    <TITLE ID="T1">
      <TL>SELECTED POEMS</TL>
      <EDTR>ed. &amp; tr. <PERS f="Ben" l="Belitt">Ben Belitt</
PERS></EDTR>
      <IN>intro. Louis Monguio</IN><LA>Spanish &amp; English texts.</
LA>
      <PB>University of California Press (Berkeley CA) 219pp (intros.
1-44)
        1966 paper &amp; cloth.</PB>
    </TITLE>
```

When the current node is the PERS node referring to Ben Belitt, the *title* of the corresponding work is reached by:

- First, stepping to PERS's *parent* node which is EDTR; then
- then, from EDTR stepping to its *parent* node, which is `TITLE`; then
- finally, from `TITLE` stepping to its *child* node `TL`, and taking the contents of this.

*(note that if we look more widely in the MPT XML files, we will see that other PERS elements are not necessarily contained with an EDTR element as Ben Belitt's PERS element is here -- indeed several other elements such as "TR" might appear as immediate parent of PERS -- the important thing to note is that there is always an immediate parent that functions something like how EDTR works here.)*

Here, then, we need to do something rather different in XPATH from what we had had to be able to do up to now. We have always described steps in XPATH from *parent to child* by simply using "/" in the XPATH abbreviated syntax (or from *node to descendant* -- a series of successive child steps -- with "//"). Here, however, we need to express a step that goes the other way: from *child to parent*. The notation for a parent-step in XPATH's abbreviated syntax is "`..`". Thus, the entire path that takes us from each PERS to the corresponding title in TL is: "`../../TL`". (Note that the *child-to-parent* steps do not (and do not need to) explicitly name the parent element in XPATH notation).

Similarly, we locate *author* information for a title containing a PERS by stepping:

- from PERS to its *parent*: EDTR;
- from EDTR to its *parent*: `TITLE`;
- from `TITLE` to its *parent*: AUTHOR;
- and from AUTHOR to its AHEAD *child*.

The XPATH construct that expresses this is "`../../../AHEAD`"

So, now it is time to modify our rudimentary XSLT index-generating stylesheet to include the author and title for each reference. We will use the XPATH notations we just described to fetch this data, but do one with a `xsl:value-of` and the other with an `xsl:apply-templates`. Here is the relevant fragment of the entire stylesheet that show how it is done:

```
<xsl:template match="MPT">
[...]
<xsl:for-each select=".//PERS[@l]">
  <xsl:sort select="@l"/>
```

```
    <xsl:sort select="@f"/>
    <li><xsl:value-of select="@l"/>, <xsl:value-of select="@f"/>:
      <xsl:value-of select="../../../AHEAD/AS"/>,
      <xsl:value-of select="../../../AHEAD/AF"/>,
      <I><xsl:apply-templates select="../../TL"/></I>
    </li>
  </xsl:for-each>
  [...]
  </xsl:template>
```

The new material has been highlighted in bold. Immediately after the display of PERS's `l` and `f` attributes we have added a colon to separate the translator's name from the information about the work they translated. The work they translated is to be displayed by giving the original author's name, and then the work's title. Thus, two `xsl:value-of` elements are first provided to select the author's surname and forename -- using paths to the author's name derived from those shown above. Finally, the title of the work itself is included by providing an `xsl:apply-templates` element, and selecting the TL. Note that we use `xsl:apply-templates` here (rather than `xsl:value-of`) because it is possible that the contents of the corresponding TL could be mixed -- that it could contain both base texts and sub-elements to (for example) make some text in italics. By processing it using `xsl:apply-templates` we allow the XSLT template mechanism to deal with such elements contained in the TL element.

There are examples of the use of "`..`" to provide a parent step in the XPATH documentation at **[XPATH §2.5]**.

## 5. Conditional Processing: *xsl:if*

There remains a problem, and it is related to the fact that the XML input documents contain information not only about translations of works by a single author, but also information about anthologies. Anthologies, of course, have no single author, and as a result are not grouped by author. As a result of this, for references to anthologies the selection "../../../AHEAD/AS" and "../../../AHEAD/AF" both fail. When these paths select nothing, what results from the `xsl:value-of` items that contain these selection attributes is nothing -- the empty string -- resulting in HTML in the output that looks like this:

```
<li>Bauer, Carlos:
, ,
<i>CRIES FROM A WOUNDED MADRID</i>
</li>
```

The commas that are "hard-coded" in the XSLT stylesheet to separate the authors first and last name, and the name from the title as included in the output, but there is no author's name between them. Clearly, this is undesirable, and what *should* happen is that the `xsl:value-of`'s that get the author's first and surnames and the commas that separate them should be processed in the template *only if* there is an author. This kind of processing -- where something is selected based on the result of a test of some sort is called *conditional processing*. XSLT provides two ways to specify this kind of processing, and we only have time to look at one of them: the `xsl:if` element (described in the standards documentation at **[XSLT §9.1]**). Here is how we would use it to fix this particular problem:

```
<xsl:for-each select=".//PERS[@l]">
  <xsl:sort select="@l"/>
  <xsl:sort select="@f"/>
  <li><xsl:value-of select="@l"/>, <xsl:value-of select="@f"/>:
    <xsl:if test="../../../AHEAD">
```

```
        <xsl:value-of select="../../../AHEAD/AS"/>, <xsl:value-of
    select="../../../AHEAD/AF"/>,
        </xsl:if>
        <i><xsl:apply-templates select="../../TL"/></i></li>
    </xsl:for-each>
```

Note that the `xsl:if` element (highlighted in bold above) surrounds the part of the template that generates the surname and firstnames of the author. `xsl:if` always contains a `test` attribute that specifies a test to be applied. Here the test establishes if the particular title the processor is working on has an author or not. If the test fails, the material it contains is ignored and for this item the HTML document will contain no information about an author -- there is none to display. If the test succeeds, then there is an author to display, and the contents of the `xsl:if` element is processed, resulting in the display of the author's name.

The other kind of conditional processing is coded in XSLT using the `xsl:choose`, `xsl:when` and `xsl:otherwise` elements. These elements are described in the XSLT standards documentation at **[XSLT §9.2]**. Where the xsl:if element manages an "on/off" situation (it allows something to be included if some condition is true, and nothing appears if it is false), the xsl:choose elements allow the processor to choose between different alternatives.

## 6. Generating links: the XPATH *ancestor::* axis

The final task to complete is to make the stylesheet generate a link to the actual full title entry that is generated (presumably) by a different stylesheet against the same text.

We suppose here that this "main" stylesheet would cause each input XML document to generate its own HTML document that contains the information relevant to each language group. Each of these output HTML computer files must have a name, of course, and it turns out to be practical to take the value of the `LG` attribute of the head `MPTGRP` element to use as a part of the filename: The file `Tr.Spa1.xml`, for example, contains the `MPTGRP` with attribute `LG` set to `"Spanish"`. Thus, the corresponding HTML file could be called `Spanish.html`. Each link from our index HTML document to the particular title data will need, then, to similarly locate and use the LG attribute that it is in to establish which HTML output document to link to.

Our index HTML document will also need to be able to point to *particular titles* in the generated language-group HTML files. You might note that each TITLE element in the input XML has an ID attribute which is unique within that particular input document. Thus, the main stylesheet simply has to transform the input XML `<TITLE ID="T2">` into output HTML `<a name="T2">` to make each title it contains directly addressable.

Now, knowing these two things, we know how to formulate an href index reference from the generated index document to each main entry. The title with `ID` of `T2` in the "Spanish" document would be referred to as `<a href="Spanish.html#T2">`. All we have to do now is determine how to make our stylesheet generate such a thing.

Recall that the index stylesheet is using `xsl:for-each` to select `PERS` elements. Thus, for each of these `PERS` elements we need to locate the two pieces of information that go into the a `href` reference, (1) the filename that will contain the full material, and (2) the title ID that allows us to address the particular title within that document. It turns out that (2) can be accomplished using the tools we know already. From `PERS`:

- The *parent* element will be an element like `EDTR`,
- The *parent* of `EDTR` will be the `TITLE` element, and
- The `TITLE` element's `ID` *attribute* contains the value we will want.

Thus, the path from each PERS to its corresponding title `ID` is "`../../@ID`". *(Note again that the fact that `ID` is an attribute of the `TITLE` element is not made explicit in the path's `..` parent step, and neither does it matter.)*

At first it would appear that a similar technique could be used to establish the filename from the LG attribute of the MPTGRP that contains each PERS element: walk up through the various ancestor nodes from each `PERS` element until we get to the document element `MPTGRP`, and then take its `LG` attribute. The problem is that the *number* of child-to-parent steps is different for nodes that are found within anthologies from those found within single-author titles -- for `PERS` nodes within single-author titles, the path would be `../../../../../@LG`, whereas for `PERS` nodes in anthology titles it contains one less parent step: `../../../../@LG`

To solve this problem best we need, at last, to begin to make use of parts of XPATH's so-called *unabbreviated syntax*. You may recall that when we first mentioned XPATH we explained then that we would be focusing mainly on the XPATH *abbreviated syntax*, but that we would, at some point touch on the *unabbreviated syntax*. Here is the first place where it is useful to introduce a part of XPATH's unabbreviated syntax, and in this way take advantage of a concept in XPATH called the *axis*. The XPATH "axis" refers to the genealogical-like relationship (e.g. child, parent, sibling, ancestor, etc.) between two nodes in an XML tree. In the abbreviated syntax we have been using up to now the axis associated with each step is implicit, and has generally been the "child" relationship -- called in XPATH the "child" axis. For example, the XPATH notation "`A/B`" means, "find elements called B which are *children* of elements called A". In the unabbreviated syntax one specifies the axis to be used for a step explicitly rather than implicitly: the corresponding unabbreviated syntax for the same path would be "`A/child::B`" -- note that the axis name has been introduced directly after the "/" step operator, and the name of the axis is followed by two colons ("`::`"). Following an axis name by two colons is the standard way to show that you are introducing an axis into a path.

Although we do not need to explicitly specify the *child* axis because it is the default one, we must use the unabbreviated syntax and explicitly provide an axis for a step when one wants to define a step that is something other than parent-child. For our particular problem here, for example, we want to examine all the *ancestor*-nodes of the current `PERS` node until we find the one named `MPTGRP`. The axis name for the set of ancestor nodes from the current node is, not surprisingly, *ancestor*. Thus, this XPATH statement: "`ancestor::MPTGRP`" asks the processor to collect all the ancestor nodes of the current node and locate from within them the element MPTGRP.. The full specification, then, to locate MPTGRP's LG attribute from any PERS element would be "`ancestor::MPTGRP/@LG`". The "`ancester::MPTGRP`" part locates the MPTGRP element. The "`/@LG`" uses a bit of abbreviated syntax to locate this element's child LG attribute.

There are 13 different axes defined in XPATH and they provide different ways to select nodes for a step. The `descendant::` axis selects all descendants of the current node. The `preceding-sibling::` or `following-sibling::` axis selects all preceding or following siblings of the current node. More generally still, the `preceding::` or `following::` axes select nodes that appear before or after the current node (excluding ancestors). A path specification "`preceding-sibling::P`", for example, would find the set of P nodes that were siblings of the current node and preceded it. Although, as I hope you can imagine, the XPATH axes provide powerful ways to select data from anywhere in a document tree, unfortunately there is no time to introduce all 13 axes to you more formally here. You should, however, when time permits read more about them in the XPATH standard document in section **[XPATH §2.2]**

So, now we are ready to formulate the link by generating the relevant HTML filename using the "`ancestor::`" axis, and locating the ID that establishes the position within that file using the XPATH "`..`" notation. Recall that you can ask your XSLT processor to provide a part of a value for an attribute it is generating from the value of an XPATH statement by including the appropriate path with an *attribute value template* (in brace brackets). We need to do this twice to generate a single a `href`-type link here:

```
<a href="{ancestor::MPTGRP/@LG}.html#{../../@ID}">
```

The XSLT processor will be using a PERS element for its current node when it sees this material. Thus, the first brace-bracketed materials ask the XSLT processor to locate the LG attribute of the *ancestor* node for the current PERS element named MPTGRP and use its value for the first part of the filename. The second brace bracketed material locates the ID attribute of the TITLE node that contains the current PERS node and takes the value of its ID attribute.

Note that we could have used the `ancestor::` axis to select the TITLE element's ID attribute as well by specifying "`ancestor::TITLE/@ID`", and this might, in fact, be preferable since it still would have worked even if the number of parent steps was different from the 2 we coded with the "`..`" notation.

This link attribute appears in the stylesheet as follows:

```
<xsl:for-each select=".//PERS[@l]">
  <xsl:sort select="@l"/>
  <xsl:sort select="@f"/>
  <li><xsl:value-of select="@l"/>, <xsl:value-of select="@f"/>:
  <a href="{ancestor::MPTGRP/@LG}.html#{../../@ID}">
    <xsl:if test="../../../AHEAD">
      <xsl:value-of select="../../../AHEAD/AS"/>, <xsl:value-of
select="../../../AHEAD/AF"/>,
    </xsl:if>
    <i><xsl:apply-templates select="../../TL"/></i></li>
  </a>
</xsl:for-each>
```

Here is an excerpt of the HTML that is generated when one applies this stylesheet against the `mpt.xml` document:

```
<li>Alfau, Felipe:
  <a href="Spanish.html#T22">ALFAU, Felipe,
    <i>SENTIMENTAL SONGS/ (La Poes&iacute;a Cursi)</i></a>
</li>
<li>Amprimoz, Alexandre L.:
  <a href="French-Canadian.html#T18">CLOUTIER, Cecile,
    <i>SPRINGTIME OF SPOKEN WORDS</i></a>
</li>
<li>Amprimoz, Alexandre L.:
  <a href="French-Canadian.html#T50">MORENCY, Pierre,
    <i>A SEASON FOR BIRDS</i></a>
</li>
```

This is almost exactly what we want. One thing that should probably be included in the text displayed to the user -- perhaps before the information about the title -- is the information about what language group (also from the MPTGRP LG attribute) each title belongs to. We leave this refinement as an exercise to the reader.

## 7. What we haven't done: grouping

Although in this chapter we have come a long way towards generating a proper index, the task it not fully complete. The main thing that we have not achieved is the *grouping* of entries for each translator under a single translator title so that instead of generating output that looks like this:

```
Alfau, Felipe: ALFAU, Felip, SENTIMENTAL SONGS/ (La Poesía Cursi)
```

```
Amprimoz, Alexandre L.: CLOUTIER, Cecile, SPRINGTIME OF SPOKEN WORDS
Amprimoz, Alexandre L.: MORENCY, Pierre, A SEASON FOR BIRDS
```

(where the translator's name is repeated on each line) it appears like this:

```
Alfau, Felipe:
  ALFAU, Felip, SENTIMENTAL SONGS/ (La Poesía Cursi)
Amprimoz, Alexandre L.:
  CLOUTIER, Cecile, SPRINGTIME OF SPOKEN WORDS
  MORENCY, Pierre, A SEASON FOR BIRDS
```

Although this kind of thing (called, generically, *grouping*) can be done in XSLT Version 1.0, it turns out to be (perhaps surprisingly given how often one wishes to do it!) difficult to do with XSLT version 1.0 and involves several advanced aspects of XSLT that we have not had time to cover. The importance of this kind of transformation has been recognised, and it appears that features to make this easier to specify will be added in a later version of the standard.

In the next chapter we will address the issue of grouping, but describe a way of doing so that will move XML more in line with the way we do things in relational databases.

## Exercises

**Ex. 1.** For today's exercise you are asked to build an index similar to the index of translators. The source XML document is an excerpt from one of the files used in the production of the *Stellenbibliographie zum "Parzival" Wolframs von Eschenbach* which was created by David N. Yeandle and Carol Magner of King's College London's German Department with the help of various people at the CCH and which has now been published on CD-ROM. The work represents a full bibliography of the secondary writings on the famous Medieval German poem, Wolfram von Eschenbach's *Parzival*. The current publication covers the scholarship published about *Parzival* over the period 1984-1996 and contains information about almost 4000 works.

The XML file we are providing contains some of the information contained in the *Stellenbiblioigraphie* for about 100 of these works. This particular file does not appear among the materials on the CD-ROM, but represents one of the resources that was processed to generate the material in the form that does appear there. Fetch it from `chap 4 > exercises > wolfram.xml` and take a closer look at it now in an XML editor.

You can see that the file is a collection of information about the scholarly works covered by the *Stellenbibliographie*. The information about each work is contained in a `WORK` element, and each `WORK` is assigned a unique `ID`. Within the `WORK` element are other elements. Not all of them are needed for this exercise, but the ones that are most useful are:

- `FULLNAME`: the full formal bibliographic entry for the work.
- `SHORTNAME`: a short reference name that is unique to this particular work within this collection. It contains the author's name and the year of publication (and a letter if a particular author published more than one work in a single year).
- `THEMES`: each work is categorised into one or more thematic categories. The `THEMES` element contains information about these themes in `THEME` elements. The contents of the `THEME` element contains the theme's name in a form suitable for display. The `THEME` element's `S` attribute contains the same name in a form suitable for sorting -- a form that, if sorted by computer, will result in an alphabetical ordering of the themes that will correspond to more reader's expectations. The `THEME ID` attribute can be ignored for this exercise.

- `AUTHORS`: the names of the author or authors of a particular work are presented here in contained `AUTHOR` elements. The `AUTHOR ID` and `REL` elements can be ignored for the purposes of this exercise. The Author's full first name is in the contained `FNAME` element. The Author's last name is in the contained `LNAME` element.

The file `chap 4 > exercises > base.xsl` provides a rudimentary XSLT stylesheet that generates a base HTML document that your index document should refer to. In it, each work is given an `<a name="">` derived from the WORK's ID number -- the first work in the file has ID 4, so the HTML file contains a `<a name="W4">` for it. The present of these `a names` will allow your index document to address each individual work. The displayed data in this base document contains the *Shortname* and the *Fullname*. In the real CD the *work* display contains considerably more data, but we are not showing that to you here, and this base document will be sufficient for the purposes of this exercise.

The task to be undertaken in this exercise is to generate an index of themes based on the data in this XML file. We will break it down into steps similar to those used to generate the index of translators in the MPT we have worked through today.

As the first step create a stylesheet that uses an `xsl:for-each` element to select all the `THEME` elements in the document. Use the `xsl:sort` element to sort them alphabetically using the `THEME`'s `S` attribute.

**Ex. 2.** As the second step, amend the stylesheet you have just created to include in each line of output the `SHORTNAME` element that identifies which work contains the theme.

**Ex. 3.** As the final step, amend the stylesheet again so that you provide a link from the *shortname* display in your index to the fuller reference in the base HTML document. Recall that each work can be addressed using an `<a name` value based on the letter "W" with the work's ID number appended to it.

**Ex. 4.** If time permits, now create a similar index of authors.

# Chapter 5

# Grouping of Results: XML as Database

## 1. Introduction: The Grouping Problem

We ended the previous chapter without solving the "grouping" problem. We could ensure that all the works translated by a single translator were brought together by the *xsl:sort* element. However, if a single translator translated more than a single work we found that our generated display would display the same translator's name on more than one line:

```
Alfau, Felipe: ALFAU, Felip, SENTIMENTAL SONGS/ (La Poesía Cursi)
Amprimoz, Alexandre L.: CLOUTIER, Cecile, SPRINGTIME OF SPOKEN WORDS
Amprimoz, Alexandre L.: MORENCY, Pierre, A SEASON FOR BIRDS
```

rather than

```
Alfau, Felipe:
  ALFAU, Felip, SENTIMENTAL SONGS/ (La Poesía Cursi)
Amprimoz, Alexandre L.:
  CLOUTIER, Cecile, SPRINGTIME OF SPOKEN WORDS
  MORENCY, Pierre, A SEASON FOR BIRDS
```

Note that whereas the material we generated in chapter 4 was "flat",. in the more desirable result there is an element of hierarchy in the output -- at the top level are the translators; identified by their names. Below the translator list is the list of their works. The works have been "grouped" by translator. The design of MPT's XML makes it difficult to get XSLT to generate this kind of hierarchical grouping from a flat source such as our PERS elements, and it is worth taking a moment to understand the nature of the problem

XPATH expressions in *xsl:templates* and *xsl:for-each* elements generally match on elements, not on element contents, and these elements generate output for *each* element that they match. If a template matches, say, *para* elements, and there are 5 para elements in our document, then the template will generate material in the generated document 5 times. The translator's XSLT stylesheet we built in chapter 4 generated results based around one list item for each PERS element. Since XSLT templates and the for-each element produce one batch of output for each item that they match, there was bound to be the same number of LI items in the output as there were PERS items in the inputted XML. Although at the bottom (works) level in our two-level hierarchy this is correct -- there should be, indeed, one line of generated output for each PERS occurrence -- that model is not right for the top (translators) level. As we know, some translators will be referenced in several PERS records -- the short example above, for example, there are *3 works*, but only *2 translators*. Since *xsl:template* and *xsl:for-each* elements produce output on a one-to-one basis for each selected item, in order to generate material specific to each translator (here, the translator's name that appears as a heading in the output) and be sure that the translator's name appears only once, we need to be able to have a template or for-each select element that matches each translator's name only once.

There is a very tricky technique available in XPATH that can use our PERS elements exactly as they are to generate a selection expression that will result in one selected PERS item per translator in spite of the fact that the same translator's name may appear in several PERS elements. It is called the *Muench method* (named after XSLT guru Steve Muench) There is more information about it at `http://www.jenitennison.com/xslt/grouping/muenchian.html`, but you should be warned that the technique is quite subtle and difficult to understand at first! Furthermore, for reasons that should become evident shortly, it is not really the right technique for the problem we have here.

So, instead of doing something as complex as the Muench method we will take a more straightforward approach. We will use information in the existing MPT XML to generate some new xml that will have one element in it for each translator. Furthermore, we will do it in such a way that we can be sure that this newly created XML about each translator can be linked to all the PERS elements that share this name. Then, we will process both the existing MPT XML, and our newly created XML to get the result we want. While we examine this solution to the problem we will end up discovering a few other problems with the MPT XML design that might be addressed if the XML markup scheme was revised, and brought more in line with design as one does it for databases.

## 2. Identifying Translators with Elements

The technique to generate this new XML that is specifically about the translators will take three steps:

1. First, we will write an XSLT stylesheet that extracts all the data from the PERS elements and presents it in a new set of `trans` (for "translator") elements. Each PERS element will contribute one `trans` element, so there initially may well be more than one `trans` element per translator in the generated result.
2. Next, we will edit the resulting list of `trans` elements, removing duplicate trans elements until there is only one `trans` entry for each actual translator
3. Finally, we will write a stylesheet that will process both the list of translators we have just generated, and the PERS elements in the base document to produce the hierarchically structured index -- basing the display of each translator's name on the information in the newly created `trans` element, and the works they have translated on the PERS elements that are associated with that translator.

### 2.1 Generating a Preliminary List of Translators

Here is a first attempt at an XSLT stylesheet that performs the first step in our three step process -- extracting all the data from the PERS elements and storing them in a newly created set of `translator` elements. As you will see in a moment, its not *quite* right.

```
<xsl:stylesheet
   version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="xml"/>
<xsl:template match="MPT">
<translators>
<xsl:for-each select=".//PERS">
  <xsl:sort select="@l"/>
  <xsl:sort select="@f"/>
  <trans l="{@l}" f="{@f}"><xsl:apply-templates/></trans>
</xsl:for-each>
</translators>
</xsl:template>
</xsl:stylesheet>
```

Let us take a closer look at this stylesheet:

1. First, note that the xsl:output element specifies a method of "`xml`" rather than "`html`". The generated output from this particular stylesheet is to be more xml which will

need further processing before being used, itself, as input XML data for the final transformation which is described above as step "3".

2. There is only one template, and it matches the document element MPT. It contains an *xsl:for-each* element that selects all the PERS elements in the MPT text, and uses the *xsl:sort* element to order them by last name and first name.
3. For each PERS element selected, the output generated is a new trans element with the same two l and f attributes that were provided in the PERS element. Also, the contents of each PERS element is included in the generated `trans` output.
4. The `xsl:for-each` element in the template is "wrapped" in a `translators` element. This means that the entire set of generated `trans` elements will be enclosed in a `translators` element as well. All XML documents must have a *document element*, and here it will be this `translators` element.

Now we are ready to do the transformation. We are generating xml output rather than html output so we must be sure that we ask the XSLT processor to store the generated file in an xml file by specifying a filename with the ".xml" extension.

When we examine the resulting file we will see something like this. At first glance it doesn't look very promising:

```
<?xml version="1.0" encoding="UTF-8"?>
<translators><trans l="" f="">Josee Michaud</trans><trans l=""
f="">Susanne de
Lotbiniere-Harwood</trans><trans l="" f="">Bruce Whiteman</
trans><trans
l="" f="">Francis Farley-Chevrier</trans><trans l="" f="">Andre Roy</
trans>
<trans l="" f="">Andrea Moorehead</trans><trans l="" f="">Yann
Lovelock</trans>
<trans l="" f="">Antonio D'Alfonso</trans><trans l="" f="">Gilles
Marcotte
</trans><trans l="" f="">Douglas G. Jones</trans><trans l=""
f="">Jacques
[...]
</translators>
```

There appear to be two problems.

1. The first and most obvious one is not, from XML's perspective, a problem at all -- all the output appears to be run together with no separation between different occurrences of the trans element. This reflects the fact that, from XML's "perspective", generally it is safe to ignore "white-space" (spaces, newlines, tabs) between elements, and the meaning of the file is the same whether each `trans` element is on a line by itself or not. However, we are going to have to hand edit this output, and although this is actually correct XML, it will be difficult to hand-edit. It would be far better if each `trans` element was on a line by itself. We will address this problem in the next subsection.
2. The second problem is observed if we look more closely at the first few trans elements. There we can see that there are no `l` or `f` attribute values given in the example we show here -- although if we look further down in the file we will find `trans` elements with `l` and `f` attributes set to names. This problem is also not really a problem with the XSLT either. It reflects that fact that the MPT XML you have been given to work with is not finished by the researchers, and still has PERS tags for which the `l` and `f` attributes have not been coded. These PERS elements without `l` or `f` attributes provide the source for these generated `trans` elements without `l` or `f` attributes either. Since we asked the XSLT processor to sort the data by these `l` and `f` attributes, all the empty ones floated to the top. In this module we will soon solve the problem by removing these

useless `trans` elements manually. In the real MPT project, of course, it would be better to go back and fix the XML input.

### 2.1.1 Managing white-space in XSLT output: the *xsl:text* element

If we look at our stylesheet we can see that the `trans` element in it is on a line by itself:

```
[...]
  <trans l="{@l}" f="{@f}"><xsl:apply-templates/></trans>
[...]
```

-- it therefore has "newline" characters both preceding and following it. If these newline characters had been preserved in the template processing there would have been newline characters inserted both before and after the generated trans element in the output, rather than no newlines at all. What happened to them?

The XSLT standard talks a great deal about now to handle "white-space" nodes (text nodes that contain only spaces, tabs and/or newline characters). Much needs to be said because there is *no* simple, single, rule that can specify what *should* be done with white-space nodes that covers all situations: in certain contexts white-space nodes can be safely ignored, and in other cases they are significant. In general, however, an XSLT processor is *supposed* to ignore white-space text nodes *in the XSLT stylesheet*. That is what happened here, and for that reason, even though the XSLT stylesheet had the `trans` element appearing on a line by itself, the preceding and following newlines that surrounded it were not recognised as part of the materials to be generated, and did not appear in the generated output.

Although the XSLT standard specifies that white-space nodes in XSLT stylesheets should be simply ignored, it does provide a way to specify that they should, instead, be included in the generated output: the *xsl:text* element. The contents of an *xsl:text* element can only be character data. An XSLT processor, when presented with materials inside an *xsl:text* element is supposed to ensure that it appears exactly as provided there in the output that it generates -- even if the content of the xsl:text element is only white-space. We can use the *xsl:text* element in our stylesheet as shown below (the xsl:text element is highlighted in bold here) to specifically protect some of the whitespace in the stylesheet so that it is not ignored:

```
<xsl:stylesheet
   version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="xml"/>
<xsl:template match="MPT">
<translators>
<xsl:for-each select=".//PERS">
  <xsl:sort select="@l"/>
  <xsl:sort select="@f"/><xsl:text>
  </xsl:text><trans l="{@l}" f="{@f}"><xsl:apply-templates/></
trans></xsl:for-each>
</translators>
</xsl:template>
</xsl:stylesheet>
```

Note that the `xsl:text` element surrounds the newline and spaces that precede the `trans` element. By putting this element here we instruct the XSLT processor to preserve the newline and spaces before each trans element it generates.

Processing the stylesheet with the xsl:text element in place generates:

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

```
  <trans l="" f="">Josee Michaud</trans>
  <trans l="" f="">Susanne de Lotbiniere-Harwood</trans>
  <trans l="" f="">Bruce Whiteman</trans>
  <trans l="" f="">Francis
Farley-Chevrier</trans>
  <trans l="" f="">Andre Roy</trans>
  <trans l="" f="">Andrea Moorehead</trans>
  <trans l="" f="">Yann
Lovelock</trans>
  <trans l="" f="">Antonio D'Alfonso</trans>
  <trans l="" f="">Gilles Marcotte</trans>
  <trans l="" f="">Douglas G. Jones</trans>
[...]
</translators>
```

This is much more "editor friendly". There are still some spurious newlines in the output (the splitting of "Francis Farley-Chevrier" across two lines, for example), and these are caused by their appearance in the MPT XML source document, but these will not cause us any serious problem.

## 2.1.2 Editing the List of Translators

Now that we have a file that represents a starting point for our list of translators, we need to edit it to remove the spurious (for us) entries at the front, and remove duplicates. So, after we start editing the created XML file in an XML editor, our first task is to delete the block of items at the front that have no *l* and *f* attribute values assigned. When this has been done the file will start like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<translators>
  <trans l="Alfau" f="Felipe">poet</trans>
  <trans l="Amprimoz" f="Alexandre L.">Alexandre L.
Amprimoz</trans>
  <trans l="Amprimoz" f="Alexandre L.">Alexandre L.
Amprimoz</trans>
  <trans l="Baland" f="Timothy">Timothy Baland</trans>
  <trans l="Barnstone" f="Willis">Willis Barnstone</trans>
  <trans l="Barnstone" f="Willis">Willis Barnstone</trans>
  <trans l="Barnstone" f="Willis">Willis Barnstone</trans>
  <trans l="Barnstone" f="Willis">Willis Barnstone</trans>
  <trans l="Bartman" f="Jeoffrey">Jeoffrey Bartman</trans>
  <trans l="Bary" f="David">David Bary</trans>
  <trans l="Bauer" f="Carlos">Carlos Bauer</trans>
  <trans l="Bauer" f="Carlos">Carlos Bauer</trans>
  <trans l="Bauer" f="Carlos">Carlos Bauer</trans>
[...]
</translators>
```

This file contains a list of translators found in the set of XML documents we are working on. When we are done editing this file each translator will be represented by exactly one `trans` element. Thus our current task is to work through the file removing duplicates. There appear to be two `trans` elements for *Alexandre L. Amprimoz*, four `trans` elements for *Willis Barnstone* and three for *Carlos Bauer*. In each case, we can simply edit out the duplicate lines, so that there is only one line for each individual. The file would then begin like this:

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

```
<translators>
  <trans l="Alfau" f="Felipe">poet</trans>
  <trans l="Amprimoz" f="Alexandre L.">Alexandre L.
Amprimoz</trans>
  <trans l="Baland" f="Timothy">Timothy Baland</trans>
  <trans l="Barnstone" f="Willis">Willis Barnstone</trans>
  <trans l="Bartman" f="Jeoffrey">Jeoffrey Bartman</trans>
  <trans l="Bary" f="David">David Bary</trans>
  <trans l="Bauer" f="Carlos">Carlos Bauer</trans>
[...]
</translators>
```

We would then continue throughout the entire file, removing duplicate names until there is exactly one `trans` element for each translator left.

We might stop for a moment and consider whether this is always the right thing to do. Perhaps, for example, there are *two* different translators called "Willis Barnstone", and some of the works are translated by one, and some by the other. Of course, in our situation, our "rough and ready" elimination is all we can do since we don't have enough knowledge about *Willis Barnstone* to know whether the name refers to one person, or more than one. However, if we were experts in the subject matter represented by the database, and happened to know that there were *two* Willis Barnstones how would we handle it? We could leave two (identical) `trans` records for Willis Barnstone -- one for each -- but that would not really be a solution for two reasons:

1. First, we will be using the `l` and `f` attributes to "link" each translator to his/her PERS reference in the main XML file. If the `l` and `f` attributes are the same for both "Willis Barnstone" records, there would be no way to indicate which PERS elements belong to which. The XSLT processor would simply select the full set of PERS elements associated with the name "Willis Barnstone" both times.

   People who are experienced with relational database design might find this problem familiar. Indeed, it is useful to think of this as what in database terms is called the "key" problem -- in a database each record should have what is called a *key* identified that makes the record uniquely identifiable. In the MPT PERS and `trans` elements we are using the `l` and `f` attributes like database table keys, and in the `trans` set of elements, each translator should have unique values for the `l` and `f` attribute. We could think of making them unique (so that each "Willis Barnstone" has a different "key") by changing, say, the `f`attribute in one of the two `trans` elements (making the first Willis Barnstone have the `f` attribute of "Willis", and the second one as, say, "Willis (2)"). Then we would have to turn to the linked PERS elements in the main XML documents and change the `f` attribute in the PERS element that belongs to the second Willis so that they say "Willis (2)".
2. Second, unless we provide a way to in the `trans` element to provide some different text to display to our users for the generated HTML translator headers -- which will, in the end, be generated from these `trans` elements, after all -- our users will see the same name "Willis Barnstone" turn up twice as a header. We might think about how we would want to show the difference to our users. Perhaps we would want to specify birth dates to differentiate them? Perhaps location where the person lives? To handle this we would need to think about the design of the `trans` element so that it could accommodate this additional information about the translator. Perhaps the translator element structure could be expanded to contain further elements that provided information such as life-dates or locations.

   It would be useful, then, to turn this "problem" into an "opportunity"! In generating a table of translators in XML we have made it possible to record more information about the translators themselves than what could be usefully recorded in the PERS element -- since we have, in effect, elements that represent the translators themselves. The generation of this translation document provides a place to significantly expand what we can express about the translators in MPT if we wish to do so.

### 2.1.3 Recognising Accents in UTF-8

*Please note that much of the discussion below refers to an older programming editor called PFE used in previous years. Some of the discussion is obsolete if Oxygen is used, as Oyxgen is Unicode-aware, but it is included to show the character encoding problems one can encounter when using different programs.*

We should take a moment here to discuss very briefly a different issue that will become evident if you work through the entire "translator file". If we look further down in the text in an older text editor like PFE we will at some point see something like this:

```
<trans l="Elgorriaga" f="JosÃ©">JosÃ© Elgorriaga</trans>
```

What are the "funny" characters in the middle of this translator's first name? We should remember that we have asked XSLT to generate an XML file, and XML files are usually written in *Unicode*. It turns out that these "funny characters" are implicated as a way of storing Unicode in a computer file, using a convention called `UTF-8` (notice that the file's xml heading specifically says this). We don't have time to describe Unicode or UTF-8 further here, unfortunately. However, here you need to know that in UTF-8 these two "things" ("Ã©") in the text represent a single accented character. Since a text editor like PFE is not "Unicode aware", it doesn't know how to properly display these characters as the accented character they actually represent. As we will see, if we leave them as they are in PFE without trying to "fix" them, we will eventually be giving this text to our XSLT processor and it is (indeed, must be) "Unicode aware". It will recognise these properly as the accented characters that they are, and will handle them properly for us in the end. Since there are relatively few of them, we can simply not worry about them here.

Suppose, however, that we needed to see the generated XML in ISO-8859-1 rather than UTF-8. Fortunately, we can ask our XSLT processor to generate XML in ISO-8859-1 by adding an encoding attribute to the `xsl:output` element in our stylesheet:

```
<xsl:output method="xml" encoding="ISO-8859-1"/>
```

If this was done, and XSLT was used to generate a new XML document it would look like this:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<translators>
<trans l="Alfau" f="Felipe">poet</trans>
[...]
<trans l="Elgorriaga" f="José">José Elgorriaga</trans>
[...]
```

and we would see that the accented characters would then be represented in ISO-8859-1 encoding, a scheme that PFE -- our non-Unicode-aware editor -- would recognise.

## 3. Using the Translators File

After we have completed our editing of the translator file so that each translator is represented by one `trans` element we are in a position to properly generate an hierarchical display of translators and the works they have translated. We have, in the translators file, the translators uniquely identified, and will use this list in our stylesheet to generate the list of translators in our index. We have, in the main MPT files, the works they have translated, and we have links between the works in the MPT file and the translators by means of the `l` and `f` attributes which appear in both the `PERS` and `trans` elements.

The first step is to make it possible for our XSLT processor to work with both the MPT files and the translators simultaneously. We can slightly expand our "master" file we have been using (which brought together the separate MPT files so that they they could be treated as a single object), so that it includes the information about the translators as well. Here is a suitable version of the expanded master file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE MPT [
<!ENTITY afrik SYSTEM "Tr.Afrk1.xml">
<!ENTITY fr-can SYSTEM "Tr.Fr.Can1.xml">
<!ENTITY spa1 SYSTEM "Tr.Spa1.xml">
<!ENTITY trans-list SYSTEM "trans-edited.xml">
]>
<MPT>
<DOCS>
&afrik;
&fr-can;
&spa1;
</DOCS>
&trans-list;
</MPT>
```

The differences between this master file and the one we have previously used are:

1. We have added an entity definition called "trans-list" that points to the file containing our `trans` elements. Here, the filename is "`trans.edited.xml`".
2. In the body of the master document, and after the entity references for the three MPT files we have included a reference to this new entity *trans-list*. This makes the list of translators explicitly included (albeit "virtually") in the master file.
3. Finally, we have added in the master file a new element `DOCS` that surrounds the references to the pre-existing MPT files. From the point of view of our XSLT processor who uses this file, all the pre-existing MPT file information is contained in the element `DOCS`. The list of translators in `trans` elements is contained in the element `translators` (actually already contained in `trans-edited.xml` itself).

Let us now look at a stylesheet that will generate the result we want:

```
<xsl:stylesheet
   version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>
<xsl:template match="MPT">
<HTML>
<HEAD>
<META HTTP-EQUIV="Content-Type" CONTENT="text/html;
charset=iso-8859-1"/>
<TITLE>Twentieth Century Poetry in Translation: Index of
Translators</TITLE>
</HEAD>
<BODY BGCOLOR="#FFFFFF">
<H3>Twentieth Century Poetry in Translation: Index of Translators</
H3>
<HR/>
```

```
<xsl:for-each select="translators/trans">
<xsl:sort select="@l"/>
<xsl:sort select="@f"/>
<H4><xsl:value-of select="@l"/>, <xsl:value-of select="@f"/></H4>
<UL>
<xsl:for-each select="/DOCS//PERS[@l=current()/@l][@f=current()/@f]">
<LI><A HREF="{ancestor::MPTGRP/@LG}.html#{../../@ID}">
<xsl:apply-templates select="../../../AHEAD"/>
<I><xsl:value-of select="../../TL"/></I></A></LI>
</xsl:for-each>
</UL>
</xsl:for-each>
</BODY>
</HTML>
</xsl:template>
<xsl:template match="AS">
<xsl:apply-templates/>, </xsl:template>
<xsl:template match="BY"/>
<xsl:template match="DA"/>
</xsl:stylesheet>
```

The first thing to notice about this is that much of the "hard" bits of getting author and works names from the individual PERS elements is exactly the same as what we developed in the last chapter. However, the central idea to grasp first here is that we have two `xsl:for-each` elements, one nested inside the other. The first (outer one) causes the processor to select in turn each translator from the new translator list. The inside `xsl:for-each` element is used once for each translator, and has the job of locating PERS elements that match the translator that it is handed, one by one, by the outer `xsl:for-each`. Once you understand how the two nested `xsl:for-each` operates in a general way, you are ready to understand the details described below:

1. This stylesheet generates the names of each translator in HTML H4 elements, so that they are displayed prominently in the HTML document. If we read the stylesheet from the top we will find the first `xsl:for-each` element (there are two here):

   ```
   <xsl:for-each select="translators/trans">
   ```

   As just mentioned above, this selects all the trans lines one by one, and for each one processes the data it contains. From its content we can see that the first thing it generates will be the name of the translator, taken here from the `l` and `f` attributes of the current trans element itself.
2. The list of PERS elements that "belong" to that translator are to be displayed as bulleted list below the translators' names. Thus, following the H4 header for the translator is the opening HTML UL tag to begin the bulletted list.
3. Now we come to the second, and more complex, `xsl:for-each` statement:

   ```
   <xsl:for-each select="/DOCS//PERS[@l=current()/@l]
   [@f=current()/@f]">
   ```

   As mentioned above, this `xsl:for-each` is nested inside the other `xsl:for-each`, and is therefore processed for each `trans` element (and for each translator). The rather complex `select` attribute it contains has the job of selecting the PERS elements in the MPT materials that belong to the "current `trans`" element that have been selected by the containing `xsl:for-each`. We describe this `xsl:for-each` select criteria by pulling it apart into its constituent parts, and you will need to read this carefully to understand how it works:

- /DOCS//PERS: it begins by specifying that it selects PERS elements contained in the PERS element which, as we know from the structure of the master XML document described above, contains the pre-existing MPT documents.
- [@l=current()/@l]: Next there is a predicate filter. It's job is to select PERS elements whose l attribute matches the l attribute of the trans element currently being processed. Note the use of the *current()* function (not previously introduced in this course) to specify that the "@l" step at the end refers to the trans element, not the PERS element.

  The central idea behind XSLT's current() function is rather subtle -- that there are sometimes *two* elements that might claim the name to be "current" that an XPath expression needs to work with, and sometimes (such as here) we need a way to refer to both of them in a single predicate.

1. The "/DOC//PERS" XPath construct will create a list of PERS nodes, and present them one at a time to this predicate. This node is called the "context" node by some XSLT writers, and is the node to which the first "@l" refers.
2. In addition to the list of PERS nodes there is also a list of trans nodes that the surrounding xsl:for-each element is generating, and each of these are also being selected, one by one, and made available to the current xsl:for-each element. This node (selected by a surrounding *xsl:for-each* or *xsl:template* element), is called the "current" node by some XSLT writers to differentiate it from the context node, and is the one that the the "*current()*" function refers to. Hence, the complete construct for the right side of the predicate's equal sign ("current()/@l") tells XPath to take the "current trans node", and to take that trans node's l attribute as the basis for the comparison.

- [@f=current()/@f]: The final component is another predicate which selects, from PERS elements whose l attributes match the current trans element, those whose f attributes also match.

  In the end then, we have filtered out all the PERS elements so that the only ones left have both l and f attributes that match the current trans element. These are the PERS elements that refer to the current translator as identified in the trans element.

4. Within the second for-each element, then, we are dealing with a set of PERS elements that belong to a particular translator. The processing of them to glean the name of the work, and the name of the author, etc., is essentially the same as that shown in the stylesheet we developed in chapter 4.

The generated HTML begins like this:

```
<HTML>
<HEAD>
    <META HTTP-EQUIV="Content-Type" CONTENT="text/html;
charset=iso-8859-1">
    <TITLE>Twentieth Century Poetry in Translation: Index of
Translators</TITLE>
</HEAD>
<BODY BGCOLOR="#FFFFFF">
    <H3>Twentieth Century Poetry in Translation: Index of
Translators</H3>
    <HR>
    <H4>Alfau, Felipe</H4>
    <UL>
        <LI><A HREF="Spanish.html#T22">ALFAU, Felipe
            <I>SENTIMENTAL SONGS/ (La Poes&iacute;a Cursi)</I></A>
        </LI>
    </UL>
    <H4>Amprimoz, Alexandre L.</H4>
    <UL>
        <LI><A HREF="French-Canadian.html#T18">CLOUTIER, Cecile
            <I>SPRINGTIME OF SPOKEN WORDS</I></A>
        </LI>
        <LI>
            <A HREF="French-Canadian.html#T50">MORENCY, Pierre
            <I>A SEASON FOR BIRDS</I></A>
        </LI>
    </UL>
[...]
```

You can see here the H4 elements that are generated from the trans elements directly, and the UL/LI list items generated from the selected matching PERS elements.

## 4. The XSLT *key* element and function

The stylesheet described above does a good job, but operates quite slowly, even with the small amount of materials we have in the sample. If we had ten or a hundred times as much material the slowness would be quite pronounced. The reason is easy enough to understand -- for each `trans` selected by the outer `xsl:for-each` element the processor has to -- because of the inner `xsl:for-each` -- go through all the PERS elements; even though the presence of the two predicates in the inner `xsl:for-each`'s select means that each time almost all the PERS elements it reviews are rejected.

This kind of processing is not uncommon in XSLT, and a *key* element and function is provided to make it more efficient. One uses the *xsl:key* element to ask the processor to set up its own internal index, and the *key()* function to use this index to quickly select items based on the index. There is a great deal more writing about the key in the XSLT standard. It can be best explained by invoking it in a stylesheet that does the same job as the stylesheet we just wrote, and does it much more quickly.

Using the XSLT key facilities is a two-step process:

1. First, one has to tell your XSLT processor to generate an index for you. You do this by providing a `xsl:key` element that tells your processor the details about how the index should be created. Here is the one we would use to handle the trans/PERS problem we were just describing:

   ```
           <xsl:key name="persByTrans" match="PERS" use="concat(@l,'
           ',@f)"/>
   ```

   With this element we ask our processor to set up an index for us. We give it a name of `persByTrans` with *xsl:key*'s *name* attribute. We tell it that it is to index PERS elements for us with the *match* attribute, and we tell it to generate the index based on the specification provided in the *use* element. Here the *use* element contains the an XSLT function we have not seen before called `concat()` that here takes the value of PERS's `l` and `f` attribute and joins them with a space between.

   When this is done the index `persByTrans` can be given the sequence last-name, space, first-name and will immediately respond with a list of PERS elements that have their `l` and `f` attributes set to the given value.
2. Now that we have the index `persByTrans` setup we can use it. We use it to replace the complex (and slow) XPATH selection attribute used in the inner `xsl:for-each` element with this new construction:

```
<xsl:for-each select="key('persByTrans', concat(@l,'
',@f))">
```

Here we use the `key()` function (not the *xsl:key* element shown above) in the `xsl:for-each` element, asking the processor to use the index called persByTrans to look up PERS elements that it has indexed under the category formed by taking the current context `trans` element's `l` and `f` attributes and inserting a space between them.

The entire stylesheet which makes use of this key index is shown here:

```
<xsl:stylesheet
   version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>
<xsl:key name="persByTrans" match="PERS" use="concat(@l,' ',@f)"/>
<xsl:template match="/">
<HTML>
<HEAD>
<META HTTP-EQUIV="Content-Type" CONTENT="text/html;
charset=iso-8859-1"/>
<TITLE>Twentieth Century Poetry in Translation: Index of
Translators</TITLE>
</HEAD>
<BODY BGCOLOR="#FFFFFF">
<H3>Twentieth Century Poetry in Translation: Index of Translators</
H3>
<HR/>
<xsl:for-each select="MPT/translators/trans">
<xsl:sort select="@l"/>
<xsl:sort select="@f"/>
<H4><xsl:value-of select="@l"/>, <xsl:value-of select="@f"/></H4>
<UL>
<xsl:for-each select="key('persByTrans', concat(@l,' ',@f))">
<LI><A HREF="{ancestor::MPTGRP/@LG}.html#{../../@ID}">
<xsl:apply-templates select="../../../AHEAD"/>
<I><xsl:value-of select="../../TL"/></I></A></LI>
</xsl:for-each>
</UL>
</xsl:for-each>
</BODY>
</HTML>
</xsl:template>
<xsl:template match="AS">
<xsl:apply-templates/>, </xsl:template>
<xsl:template match="BY"/>
<xsl:template match="DA"/>
</xsl:stylesheet>
```

Note that the `xsl:key` element is specified as a direct child of the document `xsl:stylesheet` element. One must place all the `xsl:key` elements you want to provide here, before all the `xsl:template` elements. Note as well that the reference to the *key()* function is the only other change that needed to be made in the stylesheet from the example shown earlier, and that it is rather simpler than the select criteria we needed before.

The performance impact of using a key index is significant. On my (home) desktop machine the processor I used reported timings for the old stylesheet without the use of the key index as:

```
Transforming...
transform took 8863 milliseconds
Total time took 9283 milliseconds
```

When the key index was used the reported timings were:

```
Transforming...
transform took 461 milliseconds
Total time took 831 milliseconds
```

The entire process took less than 1/10th of the time to do. If there had been more data the speed up would have been even more dramatic.

## 5. Closing Remarks

We have come a long way with XSLT and I hope that you have gained a good sense of the broad range of transformation on an XML document that XSLT makes possible. It is possible, using XSLT alone, to generate a complete and sophisticated website, including both base material and various kinds of finding tools to locate useful information in them from a single XML document. Almost all our examples here have generated HTML as output, and we have barely looked at either how to use XSLT to transform material coded using one XML schema (DTD) into another scheme (DTD), or why this would also be useful -- although as the examples in this chapter show, this kind of transformation is also possible with XSLT, and indeed it is exactly XSLT"s ability to do this that makes it the subject of much interest in the Internet computing world at present.

I think it should be clear by now that although the XSLT and XPATH standards are, in the scheme of such things, relatively small documents, they describe a complex and powerful tool. Our 5 chapters have not covered all the basic elements of XSLT, let alone the more advanced features, and if you are interested in applying XSLT to real-world problems you will find that there is still more to learn about it. Learning to work with W3C's standard document can be helpful, and I suggest that you persist in learning how to read and understand it. There are also a number of good reference and tutorial books (described in the references section) that I suggest that you acquire if you plan to do more advanced XSLT than what we have covered here.

Finally, I hope that we have also broadened your understanding of how XML markup itself can be used to express useful information. We have, in our work with XSLT, looked at XML files that in some ways go beyond relatively conventional marked-up text to use markup to make explicit data-like information in texts. We have even seen XML documents that, in fact, are not really text documents at all, but instead represent collections of structured data such as what one might store in a relational databases. Thus, XML and XSLT are useful not only in the transformation of "documents" for "electronic publishing", but for more general purpose representation and transformation of "data" as well. Indeed, XML and XSLT, when coupled with relational databases, provide a powerful set of tools that can work well with a broad range of materials relevant to almost any humanities disciplines -- text based or not. We hope that you will find it fruitful to think of ways to apply these technologies to materials you work with in your discipline.

## Exercises

**Ex. 1.** Take the flat index of themes you generated from chapter 4's exercises *Stellenbibliographie zum "Parzival" Wolframs von Eschenbach* exercise materials and transform it into a hierarchical structure using the tools and procedures described today. Note that, unlike the index of translators for MPT where you had to first generate and edit a file of translators, you are provided here with a set of themes in the document `themes.xml`. The ID field provides a link between the themes in `themes.xml` and their reference in `wolfram.xml`.

**Ex. 2.** Create two stylesheets, one that uses a key and one that does not so that you can compare the performance with and without the use of an index key.

**Ex. 3.** Now, generate an hierarchical index for your index of authors. You will need to create an "authors" file yourself using techniques similar to those describe above. Set things up so that you can use the "ID" field as a link between the two files.