Processing XML and TEI into What? A Free-for-all Series of Workshops

DHSI 2022, Course #32: Elisa Beshero-Bondar

Coursepack contents (coauthored by Elisa Beshero-Bondar and David J. Birnbaum)

XPath for processing XML and managing projects

DHSI 2019 Course 49 (Week 2, 10–14 June, 2019)

Course description Syllabus Resources and references Course Pack View on GitHub

Instructors: Elisa Beshero-Bondar and David J. Birnbaum

Description:

Learn XPath intensively and gain superpowers with XML processing! Whether you've recently learned XML and want to build something with it, or whether you've worked with XPath before but are rusty, new and experienced coders alike will benefit from our course. XPath is usually not the center of a DHSI class, and people often gain hasty "ad hoc" experience with it when learning it only along the way to doing something else. Concentrating intensively for a week on XPath will "power up" what you can do with XML, and will help you refine the way you code your documents. Our course will assist XML coders (whether beginners or experienced) with complex processing of information from markup and from plain text. Our goals are 1) to increase our participants' confidence and fluency in reading and extracting information coded in XML archives and databases, and 2) to share strategies for systematically reviewing, designing, and building those archives and databases.

Because we can "dig" latent information out of the document "strata" of texts, we think of working with XPath as something like planning an archaeology project, turning an XML project into a carefully managed digital dig site for cultural data! In our course you'll gain experience with writing precise and powerful XPath to illuminate information that isn't obvious on a human reading. For example, we'll write XPath to calculate how frequently you have marked a certain phenomenon, or locate which names of persons are mentioned together in the same chapter, paragraph, sentence, stanza, footnote, or other structural unit. We'll apply XPath to check for accuracy of text encoding—to write schema rules to manage your coding (or your project team's coding). You will learn how XPath can help you to pull data from your documents into lists, tables, and graphic visualizations.

XPath is the center of the course, but we will explore how it applies in multiple XML processing contexts so that you learn how these work similarly and how these are used, respectively, to validate documents and to transform them for publication and other reuse. Thus we devote serious, sustained attention to writing *and* applying XPath by surveying how it is expressed in a variety of frameworks (including XSLT, XQuery, and Schematron), with a variety of materials (including XML and plain-text documents), and involving a variety of task types (such as date arithmetic to calculate how much time elapsed between dates and string surgery to look for and manipulate patterns inside your coded elements). You'll gain fluency with XPath expressions and

patterns, including predicates, operators, functions (from the core library and user-defined), regular expressions, and other features, and we'll practice these in different XML-related contexts, starting with XQuery, and moving to XSLT and Schematron). Whether you are an XML beginner or a more experienced coder, you'll find that XPath will help you with systematic encoding, document processing, and project management.

This is a hands-on course. Consider this offering in complement with, and / or to be built on by: Text Encoding Fundamentals and their Application, Out-of-the-Box Text Analysis for the Digital Humanities, Text Processing - Techniques & Traditions, XML Applications for Historical and Literary Research. No advanced knowledge of XML processing is necessary but those with interests in document processing who have taken Digital Documentation and Imaging for Humanists; Advanced TEI Concepts / TEI Customization; A Collaborative Approach to XSLT; or Geographical Information Systems in the Digital Humanities will certainly benefit.

Link to register for DHSI courses

UpTransformation is maintained by ebeshero.

XPath for processing XML and managing projects

DHSI 2019 Course 49 (Week 2, 10–14 June, 2019)

Schedule

Expand all | Collapse all

Monday, June 10: XPath

Introduction to XPath in eXist-db and <oXygen/> (10:15 a.m.–12:00 p.m.)

A. Getting started with XPath and eXide (15 minutes; 10:15 a.m.–10:30 a.m.)
Expand | Collapse
B. Simple XPath expressions (25 minutes; 10:30 a.m.–10:55 a.m.) Expand | Collapse
C. XPath in <oXygen/> (20 minutes; 10:55 a.m.–11:15 a.m.) Expand | Collapse
D. XPath path expressions (20 minutes; 11:15 a.m.–11:35 a.m.) Expand | Collapse
E. XPath path steps (25 minutes; 11:35 a.m.–12:00 p.m.) Expand | Collapse

Exploring document structures and data with XPath (1:30 p.m.–4:00 p.m.)

A. XPath functions for strings (25 minutes; 1:30 p.m.–1:55 p.m.) Expand | Collapse
B. XPath functions for numbers (20 minutes; 1:55 p.m.–2:15 p.m.) Expand | Collapse
C. XPath functions for sequences (15 minutes; 2:15 p.m.–2:30 p.m.) Expand | Collapse
D. Looking Stuff Up: XPath function signatures and cardinality (10 minutes; 2:30 p.m.–2:40 p.m.)
Expand | Collapse
E. Break (10 minutes; 2:40 p.m.–2:50 p.m.)
F. XPath predicates (20 minutes; 2:50 p.m.–3:10 p.m.) Expand | Collapse
G. Odds and ends (15 minutes; 3:10 p.m.–3:25 p.m.) Expand | Collapse
H. Read and evaluate XML projects with XPath (35 minutes; 3:25 p.m.–4:00 p.m.)
Expand | Collapse

Tuesday, June 11: XPath and XQuery

XPath and XQuery in eXist-db (9:00 a.m.–12:00 p.m.)

A. Housekeeping: documents, collections, and namespaces (10 minutes; 9:00 a.m.–9:10 a.m.)
Expand | Collapse
B. The seven types of nodes (30 minutes; 9:10 a.m.–9:40 a.m.) Expand | Collapse
C. Neglected XPath axes (25 minutes; 9:40 a.m.–10:05 a.m.) Expand | Collapse
D. Scavenger hunt 1 (40 minutes; 10:05 a.m.–10:45 a.m.) Expand | Collapse
E. Break (10 minutes; 10:45 a.m.–10:55 a.m.)
F. Regex in XPath (35 minutes; 10:55 a.m.–11:30 a.m.) Expand | Collapse
G. Introducing variables (10 minutes; 11:30 a.m.–11:40 a.m.) Expand | Collapse
H. Introducing FLWOR (20 minutes; 11:40 a.m.–12:00 p.m.) Expand | Collapse

XQuery flow control (1:30 p.m.–4:00 p.m.)

A. Scavenger Hunt 2: in XQuery this time. (30 minutes; 1:30 p.m.–2:00 p.m.)
Expand | Collapse
B. Review XPath `for` loops; sequence and range variables (<oXygen/>) (20 minutes; 2:00 p.m.–2:20 p.m.) Expand | Collapse
C. `FLWOR` statements in XQuery: how `for` works: Part 1 (20 minutes; 2:20 p.m.–2:40 p.m.)
Expand | Collapse
D. Break (10 minutes; 2:40 p.m.–2:50 p.m.)
E. `FLWOR` statements in XQuery: how `for` works: Part 2 (30 minutes; 2:50 p.m.–3:20 p.m.)
Expand | Collapse
F.  Putting it all together: writing FLWORs to make new files (40 minutes; 3:20 p.m.–4:00 p.m.)
Expand | Collapse

Wednesday, June 12: XPath and XSLT

Introduction to XPath in XSLT (9:00 a.m.–12:00 p.m.)

A. Preparation for writing XSLT in <oXygen> (20 minutes; 9:00 a.m.–9:20 a.m.)
Expand | Collapse
B. XSLT overview in <oXygen/> (60 minutes; 9:20 a.m.–10:20 a.m.) Expand | Collapse
C. Break (10 minutes; 10:20 a.m.–10:30 a.m.)
D. Identity transformation for making changes to an XML file (50 minutes; 10:30 a.m.–11:20 a.m.)
Expand | Collapse
E. Comparing XSLT and XQuery (15 minutes; 11:20 a.m.–11:35 a.m.) Expand | Collapse
F. Preparing XSLT to output HTML from TEI XML (25 minutes; 11:35 a.m.–12:00 p.m.)
Expand | Collapse

XSLT Activity (1:30 p.m.–4:00 p.m.)

A. TEI XML to HTML transformation (70 minutes; 1:30 p.m.–2:40 p.m.) Expand | Collapse
B. Break (10 minutes; 2:40 p.m.–2:50 p.m.)
C. XSLT activity: Making a linked table of contents (70 minutes; 2:50 p.m.–4:00 p.m.)
Expand | Collapse

Thursday, June 13: XPath and Schematron

Using Schematron to constrain your markup (9:00 a.m.–12:00 p.m.)

A. Schematron overview (15 minutes; 9:00 a.m.–9:15 a.m.) Expand | Collapse
B. Looking at Schematron (30 minutes; 9:15 a.m.–9:45 a.m.) Expand | Collapse
C. Schematron error reporting (20 minutes; 9:45 a.m.–10:05 a.m.) Expand | Collapse
D. XPath functions practice: Leipzig glossing rules, part 1 (20 minutes; 10:05 a.m.–10:25 a.m.) Expand | Collapse
E. Break (10 minutes; 10:25 a.m.–10:35 a.m.)
F. XPath functions practice: Leipzig glossing rules, part 2 (40 minutes; 10:35 a.m.–11:15 a.m.) Expand | Collapse
G. The Three Stooges go to Schematron Summer Camp (25 minutes; 11:15 a.m.–11:40 a.m.) Expand | Collapse
H. One more way of counting spaces and hyphens (20 minutes; 11:40 a.m.–12:00 p.m.) Expand | Collapse

Schematron and external files (1:30 p.m.–4:00 p.m.)

A. ID/IDREF validation (25 minutes; 1:30 p.m.–1:55 p.m.) Expand | Collapse
B. General comparison and value comparison (20 minutes; 1:55 p.m.–2:15 p.m.) Expand | Collapse
C. Schematron validation (25 minutes; 2:15 p.m.–2:40 p.m.) Expand | Collapse
D. Break (10 minutes; 2:40 p.m.–2:50 p.m.)
E. Exploring Digital Mitford (15 minutes; 2:50 p.m.–3:05 p.m.) Expand | Collapse
F. Hamilton 1823-04-09 letter (25 minutes; 3:05 p.m.–3:30 p.m.) Expand | Collapse
G. Webb 1819-05-16 letter (30 minutes; 3:30 p.m.–4:00 p.m.) Expand | Collapse

Friday, June 14: Taking stock

Putting it all to work (9:00 a.m.–12:00 p.m.)

A. XPath in up-conversion: Syriaca taxonomy (60 minutes; 9:00 a.m.–10:00 a.m.) Expand | Collapse
B. Resources and references (25 minutes; 10:00 a.m.–10:25 a.m.) Expand | Collapse
C. Break (10 minutes; 10:25 a.m.–10:35 a.m.)
D. Building our syllabus (60 minutes; 10:35 a.m.–11:35 a.m.) Expand | Collapse
E. Retrospective (25 minutes; 11:35 a.m.–12:00 p.m.)

UpTransformation is maintained by ebeshero.

XPath for Document Archaeology and Project Management

a repository for materials related to teaching and writing on technologies of up-conversion, featuring regex, XPath, XSLT, Schematron

View on GitHub

XPath for Document Archaeology and Project Management: References

Specifications

- XPath: XML Path Language (XPath) 3.1. W3C Recommendation 21 March 2017
- XQuery: XQuery 3.1: An XML Query Language. W3C Recommendation 21 March 2017
- XPath and XQuery functions and operators: XPath and XQuery Functions and Operators 3.1. W3C Recommendation 21 March 2017
- XSLT: XSL Transformations (XSLT) Version 3.0. W3C Recommendation 8 June 2017
- Schematron: ISO Schematron, 2016 edition

Books and links

Our own teaching materials are available at Obdurodon and Newtfire.

XPath

There are no XPath-specific reference books, but XPath is discussed in the books about XQuery and XSLT listed below. XPath functions through version 3.1 are documented on line in the Function library section of the documentation for Saxon. The Mulberry Technologies XPath 2.0 Quick Reference and XQuery 1.0 and XPath 2.0 Functions and Operators Quick Reference by Sam Wilmott are excellent, but they do not include more recent features.

XQuery

The only XQuery book you need is Priscilla Walmsley, *XQuery*, 2nd edition, 2015, O'Reilly Media, Inc. It contains documentation of both XPath functions used in XQuery and XQuery itself. The Mulberry Technologies XQuery 1.0 Quick Reference and XQuery 1.0 and XPath 2.0 Functions and Operators Quick Reference by Sam Wilmott are excellent, but they do not include more recent features.

XSLT

The best XSLT reference book is Michael Kay, *XSLT 2.0 and XPath 2.0 Programmer's Reference*, 4th edition, 2008, Wrox, but it has not been updated for XPath 3.1 or XSLT 3.0. There are no XSLT 3.0 reference books, but XSLT 3.0 elements are documented on line in the XSLT elements section of the documentation for Saxon. The Mulberry Technologies XSLT 2.0 Quick Reference by Sam Wilmott is excellent, but it does not include more recent features.

Schematron

There are no Schematron books, but for a good Schematron tutorial on line see Mulberry Technologies' Introduction to Schematron, by Wendell Piez and Debbie Lapeyre. See also the Mulberry Technologies ISO Schematron Quick Reference, by Sam Wilmott.

UpTransformation is maintained by ebeshero.
This page was generated by GitHub Pages.

XPath for Document Archaeology and Project Management

a repository for materials related to teaching and writing on technologies of up-conversion, featuring regex, XPath, XSLT, Schematron

View on GitHub

Exercises and Tutorials

Here is a partial list of exercises and tutorials (in addition to those included in full in this course pack) that we may use and adapt in DHSI week. This list will grow. If you are viewing this from the DHSI coursepak, visit this page on the class GitHub Repository at https://ebeshero.github.io/UpTransformation/Exercises.html.

- XQuery and eXist-db Tutorial for Newtfire: http://dh.newtfire.org/explainXQuery.html
- XSLT to HTML for a play: http://dh.obdurodon.org/xslt-test_instructions.xhtml

Full Complement of Exercises on Obdurodon and Newtfire:

- Obdurodon: http://dh.obdurodon.org/
- Newtfire: http://dh.newtfire.org/

Class GitHub Repository for Up to Date Course Materials and Exercises: https://ebeshero.github.io/UpTransformation/

UpTransformation is maintained by ebeshero.
This page was generated by GitHub Pages.

<oo>→<dh> Digital humanities

Maintained by: David J. Birnbaum (djbpitt@gmail.com)
CC BY NC SA
Last modified: 2018-02-17T18:26:04+0000

What can XPath do for me?

Contents

Introduction

As we discussed in our general introduction to XML (What is XML and why should humanities scholars care?), there are two principal sets of reasons why digital humanists use XML to model their texts:

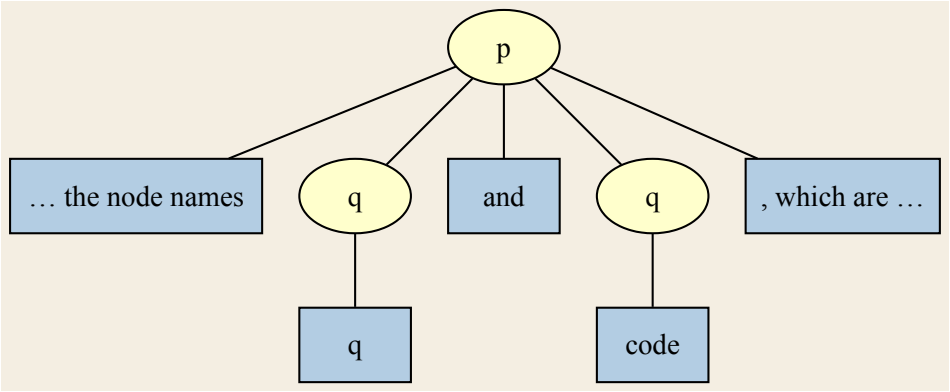
1. XML is a formal model designed to represent an ordered hierarchy, and to the extent that human documents are logically ordered and hierarchical, they can be formalized and represented easily as XML documents.
2. Computers can operate very quickly and efficiently on trees (ordered hierarchies), much more quickly and efficiently than they can on non-hierarchical text. This means that if we can model the documents we need to study as trees, we can manage and manipulate large amounts of data in a shorter time, and using fewer computer resources.

XPath is a language for selecting parts of an XML document for subsequent processing. As such, the main thing an *XPath expression* does is allow the user to describe, in a formal way that a computer can process easily, certain parts of a document (e.g, "all of the paragraphs", "all of the first paragraphs of a section, unless the section is part of an appendix", etc.). In addition to defining specific parts of a document, XPath can also manipulate the data it finds (see the discussion of functions below), but the main thing it does is serve as a helper language, or *ancillary technology*, to identify parts of a document that will then be manipulated by another language. The principal XML-related languages that employ XPath to find information in XML documents are *XSLT* (eXtensible Stylesheet Language Transformations) and *XQuery* (XML Query language). We'll learn about using XSLT and XQuery to manipulate XML documents later, but before you can do something with information in an XML document you have to be able to find it, and that's what XPath does.

This document provides some basic information about how to use XPath to describe, find, and navigate to information inside an XML document. For the most part you won't yet be doing anything with the information you find, but once you've learned how to find it, you'll employ that knowledge in subsequent lessons about XSLT and XQuery to interrogate and manipulate your source documents. The introduction you're reading now is not a complete description of XPath, but it will get you started, and you can then find information about additional XPath resources in Michael Kay's book.

Node

The discussion of XPath components (path expressions, axes, predicates, functions) below depends on two key concepts, *nodes* and *sequences*. A *node* is a piece of information in the XML tree, such as an element, an attribute, or a string of text. The XHTML paragraph that you are reading now is a single `<p>` element node that contains a mixture of text nodes (strings of plain text), `<code>` element nodes (used for snippets of XML, such as the element names "<q>" and "<code>", which are highlighted typographically by my style sheet), `<q>` element nodes (identifying quoted text, which has quotation marks inserted automatically during rendering), etc. In this particular example, each of the element nodes (`<code>`, `<q>`, etc.) within the `<p>` node happens to contain, in turn, just a single text node, but elements can also contain just other element nodes, just text, a mixture of elements and text, or nothing at all. The `<p>` node, in turn, is contained within a section (`<div>`) node, etc. This can be illustrated with the following partial tree diagram (element nodes are depicted as yellow ovals and text nodes as blue rectangles):



This partial tree corresponds to the following text as it appears in the paragraph above (without the ellipsis points, which I've added):

```
… the node names "q" and "code", which are …
```

This, in turn, appears in the markup serialization as:

```
<p> … the node names <q>q</q> and <q>code</q>, which are … </p>
```

These three perspectives (tree, rendering, XML serialization) all represent the same XML document. What XML (and, therefore XPath) cares about is the tree. In XPath terms (look at the tree):

- The `<p>` node contains just five *child* nodes. In order, these are a text node ("the node names"), the first `<q>` node, another text node ("and"), the second `<q>` node, and a third text node (", which are").
- Each `<q>` node contains a single text node, "q" for the first and "code" for the second.
- *The tree is hierarchical.* A node that contains another node is called its *parent*, so that, for example, the one `<p>` node is the parent of its five *child* nodes (text, the first `<q>`, more text, the second `<q>`, and still more text) and each `<q>` node is the parent of one text node. Nodes that have the same parent, such as the five children of the `<p>` node, are called *siblings* of one another. The tree diagram has been formatted to display siblings on the same level as one another, and the three text nodes directly under the `<p>` node plus the two `<q>` nodes are siblings. The two text nodes contained by the two `<q>` nodes are *descendants* of the `<p>` node, but not children. Although they are on the same level as each other, they aren't siblings because they don't share a parent. What we read on the web page as continuous text is actually a mixture of text nodes and element nodes, and the elements nodes, in turn, contain text nodes. The text may all appear continuous during rendering, but, as the tree shows, it lives at different levels of the hierarchy.
- *The nodes of the tree are ordered.* The child nodes of the `<p>` node, which are siblings of one another, occur in a particular order. This is why XML can be described as representing an *ordered* hierarchy of content objects.

Sequence

The group of nodes that an XPath expression returns is a *sequence*, which is a technical term for an *ordered* collection of items that *permits duplicates*. A sequence is not the same thing as a set because, according to the formal definition, the members of a set are unordered and cannot contain duplicates. The students enrolled in this course constitute a set insofar as there are no duplicates (nobody can be enrolled more than once) and they have no inherent order (one can organize them by height or alphabetically or in many other ways, but doing that doesn't change the identity of the set).

XPath components

XPath has four principal interrelated components, as follows (the examples are things one can do with each component, but they don't illustrate how to do those things, about which see below):

Component	Purpose	Examples
path expression	Describe the location of some nodes in a tree.	1. Find all `<paragraph>` elements in the tree. 2. Given a `<chapter>` element in the tree, find all `<footnote>` elements inside it. 3. Given a `<chapter>` element in the tree, find all `<footnote>` elements inside it that contain a `<definition>` element. This last expression requires a predicate, about which see below.
axis	Describe the direction in which one looks in the tree.	1. From a particular location in the tree, find all *preceding* `<footnote>` elements.

	An axis is part of a path expression.	Because we're looking for preceding footnotes, the direction searched is *backwards* or *left*. 2. From a particular `<paragraph>` element in the tree, find the `<title>` element of the `<chapter>` element that contains it. The direction searched is first *upward* in the tree (to the containing `<chapter>` element) and then *downward* (to the `<title>` element contained by that `<chapter>` element).
predicate	Filter the results of a path expression.	1. Find the *first* `<paragraph>` element in each `<chapter>` element. XPath does this by finding all `<paragraph>` elements in each `<chapter>` element and then filtering out the ones that are not the first in their cohort. 2. Find all of the `<paragraph>` elements that contain `<illustration>` elements, ignoring the ones that don't. You aren't trying to retrieve the `<illustration>` elements themselves; you're using them to filter the set of all `<paragraph>` elements according to whether or not they contain `<illustration>` elements.
function	Do something with the information retrieved from the document instead of just returning it as received.	1. Retrieve all of the `<paragraph>` elements in a `<chapter>` element (so far this is just a path expression) but instead of returning the actual elements, return just a count of how many there are. This uses the `count()` function. 2. Retrieve a bunch of nodes that contain textual items (such as from a list) and concatenate their contents into a single string, inserting a comma and space after each one except the last. This uses the `string-join()` function.

Each of these components is described in more detail below. (There are a few other types of XPath expressions that we don't discuss here. For example, `1 + 2` is an XPath expression that describes a sequence of one item, the integer "3".)

Paths

Path expressions are used to navigate from a current location (called the *context node*) to other nodes in the tree. By default, specifying the name of a node type in a path expression says to look for it *among the children of the current context node*. New steps in a path expression are indicated with slash characters (note: *not back-slashes*), and the context node changes with each step. This will be clearer if we walk step-by-step through some examples. Let's assume below that we're dealing with a prose document that consists of chapters, marked up as `<chapter>` elements, each of which contains one or more paragraphs, marked up as `<paragraph>` elements. The paragraphs, in turn, contain a mixture of plain text and quotations, marked up as `<quote>` elements.

- The path expression `quote` means "collect all the `<quote>` child elements of the current context node." If one launches this path expression from within a `<paragraph>` element, it retrieves all of the `<quote>` elements immediately inside that `<paragraph>` element, ignoring any others in the document. This means that it ignores `<quote>` elements outside the `<paragraph>` element context node, and it also ignores `<quote>` elements that are inside other `<quote>` elements in the `<paragraph>` element, since those more deeply-nested `<quote>` elements are not immediate children of the `<paragraph>` (they are children of children).
- The path expression `chapter/paragraph/quote` means "starting from the current context node, find all of the `<chapter>` elements that are its immediate children, then all of the `<paragraph>` elements that are children of those `<chapter>` elements, and then all of the `<quote>` elements that are children of those `<paragraph>` elements."

Only the items returned by the last step in the path are added to the sequence to be returned by the path expression. In the preceding example, the system traverses `<chapter>` and `<paragraph>` elements on its way to find `<quote>` elements, but only the `<quote>` elements themselves are part of the *value* of the path expression, that is, of the sequence that the expression returns. The expression visits the other elements in passing, but it does not collect them.

Slashes indicate stages in the path and the context node changes at each stage. Initially the context is wherever one starts (I'll explain how that's determined when we talk about how XSLT and XQuery use XPath), so in this example we begin by finding all of the `<chapter>` elements that are children of whatever element we're in. Once we reach the first slash, the context node changes to the sequence of `<chapter>` elements that we just retrieved at the first step, so we're now looking for `<paragraph>` elements that are children of those `<chapter>` elements. Another slash changes the context node yet again, this time to the sequence of all `<paragraph>` elements retrieved earlier, and we are now looking for `<quote>` elements that are children of those `<paragraph>` elements. Each step in the path is really defining a *sequence* of context nodes for the next step, and it then sets each one in turn as the new context node as it moves along the path.

By default, the steps in a path expression are the names of element nodes. It is also possible to address other types of nodes directly, such as attributes and text nodes. This means that, for example, if all paragraphs are tagged with an attribute value describing their language (e.g., `<paragraph language="english"> ... </paragraph>`), one could find all of the language information on paragraphs by navigating to the paragraphs and then not to any *element* within them, but to the value of the `@language` *attribute* instead. Assuming paragraphs are inside chapters, which are inside a root <novel> element, that path expression might look like /novel/chapter/paragraph/@language.

See below for an explanation of the leading slash. As is also explained below, *in XPath* a leading at-sign (`@`) identifies an attribute, and we'll use one from now on when we talk about attributes in XPath, but the attribute name *in the actual XML* is written without the at-sign.

As stated, this path would not retrieve the paragraphs in a particular language; it would retrieve the `@language` attributes, the values of which are the names of the languages. It is, of course, possible to retrieve all paragraphs (`<p>` elements) only if they are in English (for example) instead, but that isn't what this particular path expression does; this path expression retrieves the `@language` attribute nodes themselves.

Axes

By default a step in an XPath looks for an element that is a child of the current context node. As was noted above, it is possible to specify other types of nodes than elements, and it is also possible to look for nodes that are not just children, but also, for example, parents or siblings. XPath is capable of navigating from any context to any other location in the tree.

The direction in which XPath looks at each step in a path is determined by an *axis*, and by default we look for element nodes on the *child* axis. The most important directional axes in XPath are:

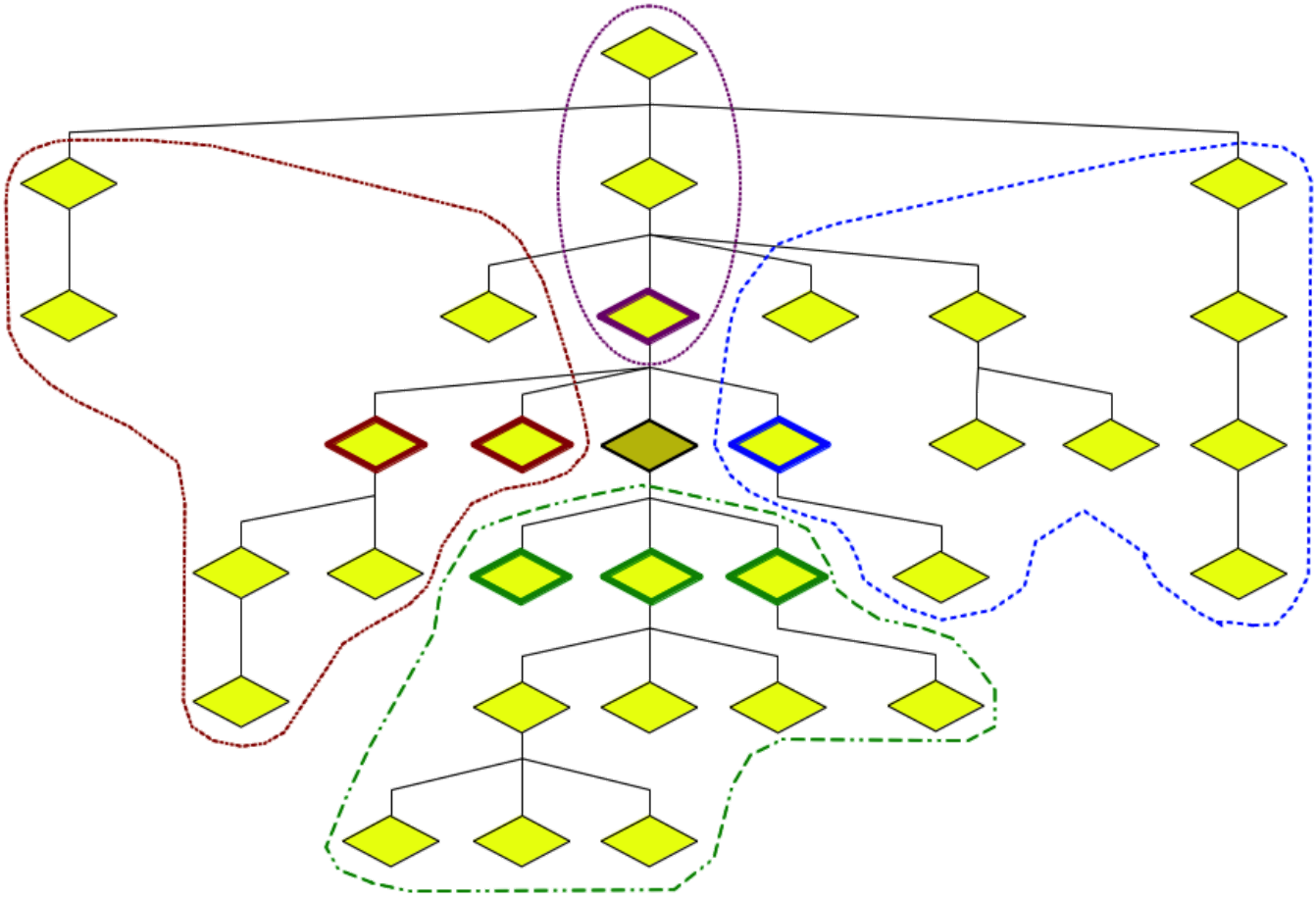
- `child`: All nodes contained directly by the current context node.
- `descendant`: All nodes contained directly by the current context node, recursively, that is, all the way down the tree. In other words, the descendants of a node are its children, its children's children, etc.
- `parent`: The node that contains the current context node. Within the social metaphor of the XML family, children have only one parent. The only node that does not have a parent is the node at the very top of the tree (above the root element), called the *document* node.
- `ancestor`: The parent of the current context node, its parent node, etc., all the way up to the document node.
- `preceding-sibling`: All nodes that share a parent with the context node and precede it in document order. In the list you're reading now, the preceding siblings of the current list item element are the other elements that precede it and have the same parents, which means the other list items that precede it in this list, but not those that follow it and not those that may precede it elsewhere in the document (since they have different parents).
- `preceding`: All nodes that precede the current context node in document order. This includes both preceding siblings and preceding nodes that are not siblings. Note that *preceding* must be understood in terms of nodes in a tree, rather than tags in a serialization. For this reason, ancestors are not preceding; although they *begin* before the current context (their start tag precedes it), the node itself doesn't precede the current context because it is still open. That is, the start tag precedes the current context, but the element *contains* it, rather than preceding it, and XPath cares about elements, not tags.
- `following-sibling`: All nodes that share a parent with the context node and follow it in document order. The mirror image of the `preceding-sibling` axis.
- `following`: All nodes that follow the current context node in document order, including both following siblings and following nodes that are not siblings. The mirror image of the `following` axis.

These eight axes fully describe looking in any direction from the current context node (there is also a `self` axis, which stays at the current context node, and a few others that also aren't used much). There is no `sibling` axis; if you want all siblings, regardless of direction, there are a couple of ways to express that, but there is no way to do so with just a single axis.

The axes can be categorized by direction (up, down, left, right) and distance (short, long), as follows:

Axis	Direction	Distance
child	down	short
descendant	down	long
parent	up	short
ancestor	up	long
preceding-sibling	left	short
preceding	left	long
following-sibling	right	short
following	right	long

The division of the tree into these eight directional axes is illustrated by the following example:



[Image courtesy of Syd Bauman, Northeastern University]

In the preceding image, intended to reflect the tree view of an XML document, the shaded diamond in the middle represents the current location, that is, the context node. The axes used to reach the other nodes are as follows:

Axes	Depiction	Nodes
child	Dark green edges	The three nodes immediately below the current location
descendant	Dashed green line	The three child nodes mentioned above, plus the seven nodes below them, all the way down (their children and their children's children)
parent	Magenta edges	The node immediately above the current location
ancestor	Magenta dashed line	The parent plus its parent, and its parent's parent
preceding-sibling	Dark red edges	The two nodes to the left of the current location that have the same parent
preceding	Dark red dashed line	The preceding-sibling nodes plus the six other nodes that are entirely to the left of the current location
following-sibling	Blue edges	The node to the right of the current location that has the same parent
following	Blue dashed line	The following-sibling node plus the nine other nodes that entirely to the right of the current location

A step in a path expression actually contains not just the name of an element type (or other node specifier; one can specify things other than elements), but also an axis. We often don't think about the axis because when no axis is specified explicitly, a default `child` axis is assumed, but the `child` axis is present, even if only implicitly, when no explicit axis is specified.

An axis is specified by taking its name followed by a double colon and prepending it to the element name (or other path step). For example, a path `paragraph` looks for `<paragraph>` elements on the child axis, while `preceding-sibling::paragraph` looks instead for `<paragraph>` elements that are preceding siblings. This means that `paragraph` as a step in a path by itself is short-hand for `child::paragraph`. Usually nobody specifies the child axis, since it's implicit when it isn't stated.

In addition to specifying the name of a specific element, one can look for any and all elements on an axis by using an asterisk (`*`). For example, the path `paragraph/*` means "find all the child `<paragraph>` elements of the current context and then find all of the child elements of those `<paragraph>` elements, regardless of element type." The asterisk can be used on other axes, as well, so that `preceding-sibling::*` means "starting at the current context node, find all preceding sibling elements, regardless of element type."

The notation single dot (`.`) refers to the current context, and is equivalent to `self::*`, that is, all of the nodes on the `self` axis, which is the one current context element, whatever it is. The notation double dot (`..`) refers to the one parent node, whatever it is, and is equivalent to `parent::*`.

A slash (`/`) normally indicates a step in a path expression, telling the system to look for whatever follows with reference to the current context. This means that, for example, `paragraph/quote` means "find all of the `<paragraph>` elements that are children of the current context and then (slash = new step in the path) all of the `<quote>` elements that are children of each of those `<paragraph>` elements." A slash at the very beginning of a path expression, though, has a special meaning: it means "start at the document node, at the top of the tree." Thus, `/paragraph` means "find all of the `<paragraph>` elements that are immediate children of the document node," a query that will succeed only if the root element of the document (the one that contains all other elements) happens to be a `<paragraph>` (and therefore immediately under the document node).

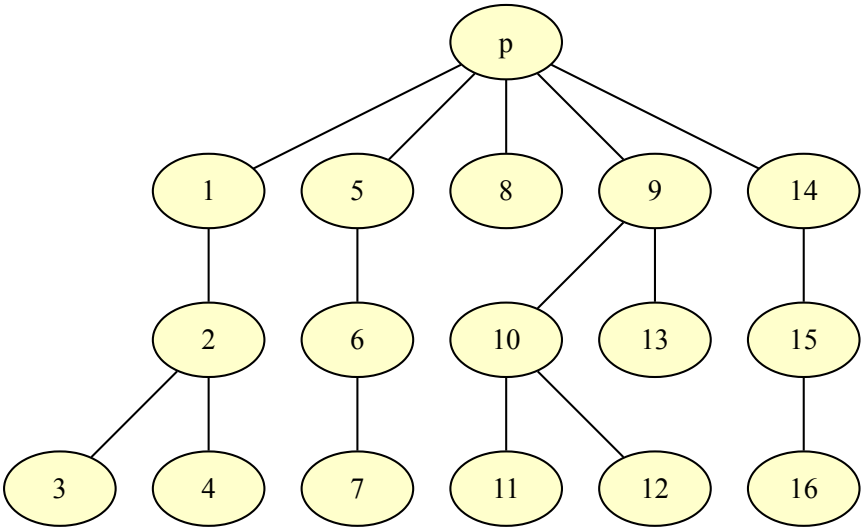
A double slash (`//`) is shorthand for the `descendant` axis, so that `chapter//quote` would first find all of the `<chapter>` elements that are children of the current context and then find all of the `<quote>` elements anywhere within them, at any depth (children, children's children, etc.). When used at the beginning of a path expression, e.g., `//paragraph//quote`, the double slash means that the path starts from the document node, at the top of the tree, and looks on the descendant axis. The preceding XPath expression therefore means "starting from the document node, find all descendant `<paragraph>` elements (= all `<paragraph>` elements anywhere in the document), and then find all `<quote>` elements anywhere inside those `<paragraph>` elements." This is one way to find all `<quote>` elements anywhere inside `<paragraph>` elements at any depth, while ignoring `<quote>` elements that are not inside `<paragraph>` elements.

Attributes are not children and are not located on the `child` axis. Instead, they are located on their own `attribute` axis. The attribute axis can be specified as `attribute::`, but it is usually abbreviated as an at sign (`@`). For example, the path expression `paragraph/@language`, which is short for `child::paragraph/attribute::language`, starts at the current context, finds all of the `<paragraph>` elements on the child axis, and then finds the `@language` attribute on each `<paragraph>` element. If a `<paragraph>` element doesn't happen to contain a `@language` attribute, nothing is added to the sequence for that particular `<paragraph>`. Curiously, although attributes are not children (they are not located on the `child` axis), they do have parents, which are the elements to which they're attached. This means that in the preceding example, although the `@language` attribute is not a child of the `<paragraph>` element (because attributes by definition are not children, they are located on the `attribute` axis, rather than the `child` axis), the `<paragraph>` element is nonetheless a parent of the attribute, and is found on the `parent` axis when the current context node is the attribute node itself. One can specify all of the attributes of the particular context node (which must be an element for this to make sense, since only elements can have attributes) with `@*` (short for `attribute::*`), so that `p/@*` navigates to all of the `<paragraph>` elements that are children of the current context node and then to all of the attributes of any type that are associated with each of them.

In addition to specifying elements and attributes by name, one can specify text nodes as `text()`, so that, for example, `paragraph/text()` navigates first to the `<paragraph>` elements that are children of the current context node and then to all of the text nodes that are its immediate children. Similarly, one can use the shorthand notation `node()` to refer to all types of nodes together. For example, `paragraph/node()` first finds all of the `<paragraph>` elements that are children of the current context and then all of the nodes of any type that are children of those `<paragraph>` nodes. Remember, though, that since no axis is specified explicitly before `node()`, the `child` axis is implied. This means that `node()` refers to elements and text nodes, but not attribute nodes, because attribute nodes are not found on the `child` axis.

Predicates

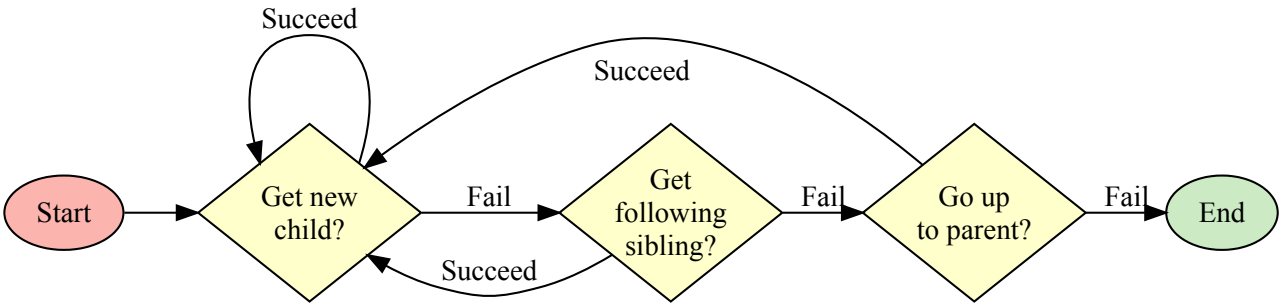
Predicates are used to filter the results of path expressions. The sequences that are returned by path expressions have an inherent and stable order, which is called *document order*. In XPath, document order is defined as *depth first*, which means that when the system has to return nodes in order, it looks down before it looks right, it never looks up (except to resume where it left off), and it never looks left. Here's an example:



Suppose we use the path expression `p//*` to find all of the elements of any type (thus the asterisk) anywhere (thus the double slash, which means `descendant` axis) inside a `<p>` element (that is a child of the current context). The preceding example shows one such `<p>` element with all of its descendant elements numbered in document order. Their type is not specified because this particular path expression is looking at all descendant elements, without checking their type.

Because XPath document order is depth first, the processor looks down and to the left and finds the first element to add to the sequence to be returned, which is #1. But what should the second element in the sequence be? In a depth first system, like XPath, before the processor looks for the siblings of #1, it looks to see whether #1 has any children, and if so, it goes there first, so the next element it retrieves in #2. Since #2, in turn, has children, the system then gets #3. Because #3 doesn't have children, the system then looks to the right, where it finds #4. At that point it has hit a dead end, with no children and no following siblings. It therefore backs up to the most recent place where it turned down, which is #2. Since the system has already visited the children of #2 (#3 and #4) and #2 doesn't have any following siblings, it backs up again, this time to #1. It has already visited its children (it has only one child, #2), so it looks to its following siblings and finds #5. Before it continues scanning other siblings, though, it notices that #5 has a child, #6, so it heads there next, etc., traversing the tree according to the numbering above.

The procedure for a depth-first traversal of the tree can be illustrated by the following flow chart:



If you're not familiar with flow charts, the conventions used here are:

- The chart is a formal representation of how to do something. In this case, that something is traversing a tree in depth-first order.
- Ellipses represent termini (start and end points).
- Arrows represent steps in the process. They can either be absolute (what you always do, e.g., you always go from the start point to looking for a child node) or conditional (depending on whether a test succeeds or fails, about which see below). Absolute steps are unlabeled. Conditional steps are labeled to indicate the condition under which you follow them.
- Diamonds represent tests, which in this chart can either succeed or fail. Each diamond has two labeled arrows emerging from it, telling you where to go depending on the outcome of the test. For example, if you're trying to get a new child node and you succeed in doing that, you then try to get its first child node, represented by the looping "Succeed" arrow. If you fail because there are no children (either none at all or none that you haven't already visited), you follow the "Fail" arrow instead and try to get a sibling.
- This chart has one starting point and one stopping point, and you always wind up at the stopping point.

If you follow the full sequence (in either the flow chart above or the numbered node diagram above that), you'll see that the algorithm is that you collect nodes as you visit them and add them to the sequence you're collecting, but you don't add any node more than once. The charts show the order for visiting nodes in a depth-first traversal:

1. Try to visit any children of the current context node that you haven't visited yet, starting with the leftmost. On the flow chart, this is the decision step labeled "Get a new child?" If that attempted visit succeeds, you add the node to the sequence you're building and it becomes the new context node, so now try to visit *its* children. This is shown in the flow chart as looping on success (that is, if you find a child, you then look at *its* children).
2. If your attempt to find an unvisited child fails, look right to see whether there are any following siblings. If there are, visit the closest one, which becomes the new current context node, and then start looking at *its* children, following the steps of this procedure.
3. If there are no unvisited children (step #1 in the flow chart fails) and no following siblings (step #2 fails), try to back up the tree to the *parent* to see whether *it* has any unvisited children. If so, visit them, following this procedure. If not (that is, if step #1 fails after you've backed up), check for siblings of the parent(step #2). Whenever checking for siblings fails, keep backing up. If you back all the way up to the document node (which doesn't have a parent) and there's nobody left to visit (that is, when step #3 fails), you're done.

In general it's best to think of XML in terms of node on a tree (elements, attribute, and text nodes), and not as a stream of characters with tags thrown in, but some users find the tag perspective helpful when considering document order, especially in the case of the `preceding` and `following` axes. From the perspective of tags:

- Starting from the current context element, the elements on the `preceding` axis (`preceding::*`) are those with *end* tags that *precede* the *start* tag of the current context element. If the *end* tag of an element *precedes* the *start* tag of the current context element, it means that the entire other element must precede the current context.
- Conversely, the elements on the `following` axis (`following::*`) are those with *start* tags that *follow* the *end* tag of the current context element. If the *start* tag of an element *follows* the *end* tag of the current context element, it means that the entire other element must follow the current context.

I've spent a lot of time discussing depth-first order because it can be used to filter a sequence by postion. Suppose you want to format the first paragraph of each chapter specially, perhaps with a drop cap, or by suppressing the indentation that you apply to all other paragraphs. One way to do this is to mark up that paragraph differently from the others, but that's a fragile solution, since if you decide to rearrange the text and the paragraph is no longer first, you have to change the markup in addition to moving it. On the web you can apply special formatting to the first paragraph using Cascading Style Sheets (CSS), but not all publication is on the web. In XPath, though, you can specify the first paragraph of each chapter by using a path expression like `//chapter/paragraph[1]`. This says: "First start at the document node, at the very top of the document, and find all descendant `<chapter>` elements, that is, all `<chapter>` elements anywhere in the document. Then, for each of them, find all of its child `<paragraph>` elements and select only the first one." There's nothing magic about "first," although it's typically the most useful in real projects. If what you care about is the third paragraph of each chapter, `//chapter/paragraph[3]` will retrieve that. Two other important details about numerical predicates are that:

- Because the last item in a sequence is often particularly useful, but its numerical value may vary, XPath provides a special pseudo-numerical predicate: `//chapter/paragraph[last()]` will retrieve the last paragraph of each chapter, without your having to tell it how many paragraphs there are.
- Nodes are counted away from the current context node, which means that with axes that travel up (`ancestor`) or left (`preceding-sibling`, `preceding`), the first node is the one closest to the current context node, etc., as if one were traversing a depth-first sequence backwards. You can find illustrations of numbered traversal on different axes on pp. 609–12 of Michael Kay's book.

Predicates are expressed by putting them in square brackets after the step in the path expression to which they apply, and it doesn't have to be the last step. For example, `//chapter[1]/paragraph[2]` finds all of the `<chapter>` elements anywhere in the document and keeps just the first of them, and it then gets all of the `<paragraph>` elements that are children of that particular `<chapter>` and keeps just the second of them.

Any expression in square brackets that filters a step in a path expression is a predicate. Numerical predicates are the easiest to understand, since they test simply for the location of an element in a sequence returned by a depth-first traversal of the tree. More complex predicates use *functions*, described in the next section.

Functions

Functions operate on the information returned by a path expression or another function. For example, the path expression `chapter/paragraph` finds all of the `<chapter>` children of the current context and uses them to find all of their `<paragraph>` children. If you don't need the actual paragraphs, and you just want to count them, you can use the `count()` function, so that `chapter/count(paragraph)` means that once you've made it to the chapter, you should return not the paragraphs themselves, but just a count of them. This XPath expression will return a sequence of number values, giving the count of the number of paragraphs in each chapter (that is, a count of the number of `<paragraph>` elements inside each `<chapter>` element). Note that this expression is different from `count(chapter/paragraph)`. The latter expression returns only one number because it defers counting until it has retrieved all of the paragraphs inside all of the chapter elements that are children of the current context. The first expression, on the other hand, counts separately inside each chapter. The difference is that the two use the `count()` function at different steps in the path expression. There are two steps (find the chapters, and then for each chapter find the paragraphs), and one can count at either point.

XPath has a little more than one hundred functions, but in practical projects you'll rarely needs more than a couple of dozen, which you'll learn quickly as you start using them. Don't try to memorize them all, but do read over the full list periodically, without trying to

memorize it, just to remind yourself of what's available, so that you can look it up as needed. There are organized lists of all of the XPath functions at https://www.w3schools.com/xml/xsl_functions.asp and detailed discussion with examples in Michael Kay's book.

Functions can be nested. For example, there is a string-manipulation function to convert all text to lower case and a different function to normalize the white space (spaces, tabs, new lines, etc.) in text (the rule converts all white space to plain space characters, reduces all sequences of white-space characters to single spaces, and removes all leading and trailing white space). If you want to retrieve a set of values and perform both of these functions, you can nest them: `normalize-space(lower-case(.))`. This means "take the current context node (represented by the dot), convert any text in it to lower case, and then take the output of the `lower-case()` function and normalize the white space in it."

You can use functions in predicates to filter expressions. For example, if you want to retrieve all of the chapters that consist of just a single paragraph (perhaps as part of proof-reading; if they consist of a single paragraph, perhaps they shouldn't have been independent chapters in the first place), you can do that with `//chapter[count(paragraph) eq 1]`. This says "first find all of the `<chapter>` elements and then filter them by saving only the ones where the number of `<paragraph>` elements they contain is equal to 1." Note that the `<paragraph>` elements in question are on the child axis because that's what's implied whenever no axis is specified.

You can also apply sequential predicates. Suppose you want to find all first paragraphs of chapters that contain more than a hundred characters. XPath provides a `string-length()` function that returns the length of text by counting characters. When it does this, it operates on the *string value* of the element, which is the total count of all textual characters anywhere inside it, no matter how deeply they may be nested. In other words, if a paragraph contains a mixture of plain text and, say, `<quote>` elements, the `string-length()` function, when applied to that paragraph, will count equally the textual characters directly inside the `<paragraph>` element and those inside the `<quote>` elements that may be inside the `<paragraph>` element. The XPath to specify all first paragraphs of chapters only if they contain more than a hundred characters is `//chapter/paragraph[1][string-length(.) gt 100]`. This says "find all of the `<chapter>` elements anywhere in the document, then find their child `<paragraph>` elements and select only the first ones. Then filter those by selecting only the ones whose string length is greater than 100, that is, that contain more than 100 characters." The dot in the `string-length()` function here refers to the current context node, which became a `<paragraph>` element at the step of the path expression that specified `paragraph`.

Note that retrieving the first paragraphs of all chapters only if they contain more than 100 characters is not the same as retrieving the first paragraphs of all chapters that contain more than 100 characters. The first of these tasks will return nothing for chapters where the first paragraph fails to contain more than 100 characters. The second will return nothing for a chapter only if none of its paragraphs contains more than 100 characters, and you could write it as `//chapter/paragraph[string-length() gt 100][1]`. The way this expression operates is that it finds all chapters in the document, and then, for each chapter, it finds all of its paragraph children. It filters those paragraph children by keeping only the ones longer than 100 characters, and it then keeps only the first of the paragraphs that survive that filtering. The two expressions, `//chapter/paragraph[1][string-length(.) gt 100]` and `//chapter/paragraph[string-length() gt 100][1]`, return different things because the predicates are applied in order, from left to right.

Review of terms and symbols

The preceding survey of XPath has introduced a lot of new terms. For review purposes, the ones you should remember (or, at least, recognize when you see them again) are:

Term	Definition
axis	Path direction and scope, e.g., `ancestor`, `preceding-sibling`.
depth-first order	See *document order*, below.
document node	The node that serves as the parent of the top-level, or root, element. The document node is the only node of any type on an XML tree that does not have a parent node.
document order	XPath traverses the tree in depth-first order, which means that it visits nodes in order and looks at a node's children before it looks at its following siblings.
function	Operation that can be performed on the result of a path expression, e.g., counting the number of nodes and returning just the count instead of the nodes themselves.
node	Part of an XML document. The most important types of nodes are element, attribute, and `text()`.
path expression	The way to reach the nodes you care about. Path expression may have multiple steps, separated by slash characters.
predicate	A filter applied to the results of a path expression, specified in square brackets.
root element	The element that contains the entire document. The root element is actually the child of the document node.
sequence	An ordered collection of pieces of information. One example of a sequence is the nodes singled out from the tree in document order by an XPath expression.

See also the table of axes, above. The shorthand axis notation is:

Symbol	Meaning	Expanded version
`.`	current context node	`self::*` (for elements)
`..`	parent element	`parent::*`
`//`	descendant axis	`descendant::`. At the beginning of a path expression, it means that the path starts at the document node.
`@`	attribute axis	`attribute::`

Slash (`/`) indicates a step in a path expression. At the beginning of a path expression, it represents the document node.

<oo>→<dh> Digital humanities

Maintained by: David J. Birnbaum (djbpitt@gmail.com)
Last modified: 2018-02-21T14:50:49+0000

The XPath functions we use most

There's a more complete list, with examples, at http://www.w3schools.com/xml/xsl_functions.asp.

1. A few useful XPath features

Variables

Variable names begin with a dollar sign, and you can create them as needed. See the discussion of the `for` construction, below.

The dot: `.`

In XPath the dot represents the current node, whatever it is. For example, `//age[. eq 10]` finds all of the `<age>` elements in the document and then filters them according to the predicate. The dot within the predicate means to take each `<age>` element in turn (make it the current node) and test whether it is equal to the value 10.

`for $i in (sequence) return …`

The `for` construction can be used for iteration. `for $i in (1, 3, 5) return (//sp)[$i]` will return the first, third, and fifth `<sp>` elements in the document. Variable names begin with a dollar sign, so this XPath expression creates a variable `$i`, sets it to each of the values in the parenthesized sequence in turn, and then uses that value as a numerical predicate to retrieve the corresponding `<sp>` element. The name of the variable is arbitrary, except that it must begin with the dollar sign.

2. General-purpose functions

`distinct-values(arg+)`

Removes duplicates from a set of values.

`reverse((arg*))`

Reverses the order of the items in a sequence. Handy for counting backwards; the XPath expression `(1 to 10)` yields ten numbers in order but `(10 to 1)` yields an empty sequence because XPath can't count backwards. You can overcome this limitation with `reverse(1 to 10)`. (Alternatively you could use `for $i in (1 to 10) return 11 - $i`.)

`name(arg?)`

Returns the name (GI) of the node. `//*/name()` will find all elements in the document and instead of returning them (tags, contents, and all), it will return just their names.

3. Casting

`number(arg)`, `string(arg)`

Convert the argument to the specified value. If you can't be sure in advance that the conversion will succeed (what does it mean to convert a string of letters to a number?), look up the details in Kay.

`xs:string(arg)`, `xs:integer(arg)`, `xs:double(arg)`, `xs:decimal(arg)`, `xs:float(arg)`,

Cast to specific datatypes. May generate an error if the input value isn't castable as the target datatype. There are other datatypes that can also be used here.

4. Strings

`concat(string+)`, `string-join((string+),string)`

`concat()` joins the strings as is. `string-join()` lets the user specify a sequence of strings to join (the first argument) and a separator string to insert between the items.

`normalize-space(string)`

Converts all white space to space characters, compresses sequences of spaces into a single space, strips leading and trailing spaces.

`upper-case(string)`, `lower-case(string)`

Changes case of string. Useful for case-insensitive searching, sorting, comparing, etc. We never use `upper-case()` ourselves.

`string-length(string)`

Returns the length of the string in characters. Often used as a path step, e.g., `//sp/string-length(.)`. The preceding XPath finds all of the `<sp>` elements and then returns the length of each in turn (the dot refers to the current context node, that is, to each individual `<sp>` as you loop through them). You can't use `string-length(//sp)` because the `string-length()` function can only take a single argument, and `//sp` is likely to return multiple nodes.

`contains(string1, string2)`, `starts-with(string1, string2)`, `ends-with(string1, string2)`

Tests whether the first string has the property specified by the second, that is, whether the first contains, starts with, or ends with the second. Useful for filtering; `//sentence[ends-with(., '?')]` finds all `<sentence>` elements and keeps only the ones that end with a question mark. The question mark in quotation marks is the second string. The dot (not in quotation marks) is an XPath way of representing the current node, whatever it is. For each `<sentence>` retrieved by `//sentence`, then, within the predicate the dot is treated as the value of that particular (current) `<sentence>` node.

`translate(string1, string2, string3)`

Takes `string1` and replaces every instance of a character in it from `string2` with the corresponding character from `string3`. `translate('string','ti','pa')` will change `string` into `sprang`. Can only do one-to-one replacements; see also `replace()`. Can be used for deletion by making `string3` shorter than `string2`; `//p/translate(., 'aeiou', '')` will strip all the vowels from each `<p>` by replacing them with nothing.

`substring-before(string1, string2)`, `substring-after(string1, string2)`

Returns the part of `string1` before (or after) the first occurrence of `string2`. Useful for breaking apart certain structures, e.g., the area code for a ten-digit US telephone number in normal `123-456-7890` format is `telephone/substring-before(.,'-')`.

`matches(string, regex)`

Tests whether the regex (regular expression pattern) occurs in the string. We cover regex later; for now, one type of regex is a plain string, so (with oversimplification) `matches(string1, string2)` is equivalent to `contains(string1, string2)`. The real power of `matches()` will become clearer once we get to regex.

`replace(string, regex, regex-replace)`

The `translate()` function, above, can only replace single characters with single characters. The `replace()` function can match regex patterns and perform more complicated replacements. Stay tuned.

`tokenize(string, regex)`

Breaks a string into parts by dividing at the regex. Handy for processing IDREFS attributes; ask for details.

5. Numbers

`count(arg*)`, `avg(arg*)`, `max(arg*)`, `min(arg*)`, `sum(arg*)`

Count, average (mean), largest value, smallest value, and total of all values. The arguments have to make sense; trying to run `sum()` over letters will generate an error. Note the double parentheses; these functions take a single value that is a sequence, not a set of values. `sum(1, 2, 3)` will generate an error because it lists three values. `sum((1, 2, 3))` yields `6` because there is just a single argument, a sequence of three values.

`ceiling(num)`, `floor(num)`, `round(num)`

These take a single argument and round up, down, or closest.

6. Boolean

`not(arg)`

Inverts the truth value of the argument. Usefully wrapped around other functions, e.g., `//p[not(q)]` returns all `<p>` elements that do not contain a `<q>` child element.

7. Context

`position()`

Returns the position of the node. Useful for filtering, e.g., `(//sp)[position() < 6]` retrieves the first five `<sp>` elements in the document. Note that nothing goes inside the parentheses.

`last()`

Used as a positional predicate. `(//p)[1]` returns the first `<p>` element in the document. `(//p)[last()]` returns the last. Note that nothing goes inside the parentheses.

8. Comparison

XPath supports two types of comparison: *value comparison* and *general comparison*.

Value comparison

The value comparison operators are:

- **eq** equal to
- **ne** not equal to
- **gt** greater than
- **ge** greater than or equal to (not less than)
- **lt** less than
- **le** less than or equal to (not greater than)

Value comparison can be used only to compare exactly one item to exactly one other item. For example, to create a predicate that will filter `<sp>` elements to keep only those where the value of the associated `@who` attribute is equal to the string "hamlet", we can write:

```
//sp[@who eq 'hamlet']
```

Since each `<sp>` has exactly one `@who` attribute and since we are comparing it to a single string, the test will return True or False for each `<sp>` in the document. Because the "exactly one item" can be an empty sequence (technically no items), the test will also work (and return False) when an `<sp>` element has no `@who` attribute. It is, however, an error if either side of the comparison contains a sequence of more than one item.

Value comparison is often used for numerical values. To keep all of the speeches (`<sp>` elements) with more than 8 line (`<l>`) descendants, we can write:

```
//sp[count(descendant::l) gt 8]
```

In the preceding example, the output of the `count()` function is a single item, an integer, and it is being compared to another single item, the integer value 8.

General comparison

The general comparison operators are:

- **=** equal to
- **!=** not equal to
- **>** greater than (may also be written **>**)
- **>=** greater than or equal to (not less than; may also be written **>=**)
- **<** less than (may also be written **<**)
- **<=** less than or equal to (not greater than; may also be written **<=**)

While value comparison operators can compare only one thing on the left to one thing on the right, general comparison operators can have one or more items on either side of the comparison (also zero items, since the empty sequence is also allowed). For example:

```
//sp[@who = ('hamlet', 'ophelia')]
```

will retain all `<sp>` elements where the `@who` attribute is equal to *either* "hamlet" or "ophelia". This makes general comparison a convenient alternative to a complex predicate like:

```
//sp[@who eq 'hamlet' or @who eq 'ophelia']
```

In comparisons with exactly one item on either side of the comparison operator, value comparison and general comparison are equivalent.

One possibly surprising feature of general comparison is the way it behaves with negation. Consider:

```
//sp[@who != ('hamlet', 'ophelia')]
```

This does not find all speeches by anyone other than Hamlet or Ophelia! It finds all speeches where the `@who` attribute is not equal to *any one* of the individual items in the sequence on the right. This means that it finds all speeches without exception, since the ones by Hamlet are not by Ophelia (the test succeeds because `@who` is not equal to "ophelia" in situations where it is equal to "hamlet") and vice versa.

So how *do* you find all speeches by anyone other than Hamlet or Ophelia? Try:

```
//sp[not(@who = ('hamlet', 'ophelia'))]
```

The preceding predicate says that we want to keep all speeches where it is not the case that the `@who` attribute is equal to either "hamlet" or "ophelia".

Summary of comparison operators

Description	Value	General
Equal to	eq	=
Not equal to	ne	!=
Greater than	gt	> (>)
Greater than or equal to (not less than)	ge	>= (>=)
Less than	lt	< (<)
Less than or equal to (not greater than)	le	<= (<=)

newtFire {dhlds}
Maintained by: Elisa E. Beshero-Bondar (ebb8 at pitt.edu) (cc) BY-NC-SA Last modified: Sunday, 06-Aug-2017 18:15:25 EDT. Powered by firebellies.

Autotagging with Regular Expressions (Regex)

Regular Expression Matching (Regex)

When we need to convert plain text or other digital text files into XML, we look for strategies to convert patterns into markup. For example, there may be clear signals in the text to show us divisions between sections (as in chapter breaks in a book, or act and scene divisions in a play), and we might be able to tell from patterns of line breaks where paragraph divisions fall. To help us identify, match, and locate all of these in a file at once (instead of one at a time), we use **regular expressions**, which are basically *patterns to match strings of text*. There are many slightly different varieties of regular expressions used in different coding and programming environments, and we will be using one of these that is standard for our XML editing work and the <oXygen/> editor we are using.

We use regular expression matching in what we call *up-conversion* from text to XML, and we also use it sometimes when we write XSLT to transform XML-to-XML, when we need to add markup based on particular patterns we can locate in the text. (For example, we might find that all the dates in a document are written in the same format and wrapped in square brackets, and we can quickly use regular expression matching to distinguish dates from other kinds of square-bracketed material by identifying the brackets and a pattern of numbers and hyphens. We locate and alter those dates with regular expressions either while coding an XML file or in up-converting a plain text file.)

In <oXygen/>, look at the Find/Replace window, select the checkbox next to “Regular Expressions” in the Options menus, and try typing a backslash character (\ ) into the Find window to bring up a short scrollable list of regular expression patterns. There are many others we can use, and we tend to look these up and deploy them as needed (rather than memorizing a long list). We use this handy Regular Expressions Info Quick Start Guide very frequently, and it’s a great place for you to start learning and looking up regular expression patterns. The regex expressions we are listing on this page are those we use frequently in our projects. There are other convenient listings online, such as The Regular Expression Library at RegExLib.com , or Wikipedia’s Regular Expression page which may also be helpful. In the next section, we’ll discuss some basic starting points and procedures we commonly use in our *up-conversion* work.

Autotagging: Up-conversion from Plain Text

When we begin converting text files to XML, we start in the <oXygen/> window, and we try to show all the special formatting characters in the document. In <oXygen/>, go to **Options -> Preferences -> Editor: Whitespaces:** and mark to **Show TAB and SPACE marks**.

We then go to the Find/Replace window (CTRL+F on a PC computer, or on the “Find” dropdown menu), and **do the following:**

- Select Case sensitive
- Select Wrap around

- Select Regular expression
- Important: At least at first, we suggest you **deselect** "Dot matches all." The “dot” represents any character, and it can be very powerful or a little unwieldy! When “Dot matches all” is selected, it includes newline characters, and so if you wrote .+ to match more than one character, it could match an entire document, what we call a *greedy match*. When we deselect “dot matches all,” it matches any character within a line, and is typically easier to maneuver! That said, there will be times that "Dot matches all" is useful, in combination with other expressions.

We typically do the following in the Find/Replace window, first working on changing special characters not permitted in XML content, **working with ampersands first**. (The order is important here, because you don’t want to change ampersands twice when you’re working on the angle bracket characters (if you have them). If you do the angle brackets first, you then leave those new ampersand characters designating the left and right brackets open for conversion when you only want the real ampersands by themselves. Make sense?)

1. Change & to &
2. Then change < to < and > to >
3. Look for ways to condense multiple blank lines, but only after analyzing your document and determining which ones should be kept as markers of, say, section breaks: We typically look for something like this, hunting for “newline” characters, \n:
   \n{3,} or \n\n\n+ in the Find window, and replace with \n\n, or whatever makes sense to you!
4. While it may make the most sense to save this for last, you will need to (manually) **add a root element** to surround everything and make an XML file.

Useful Regex Pattern Symbols:

- \n =new line character (in RegEx) Example: replace \n with </item>\n<item>
- \t = select tab
- \s = selects any white-space character (including tabs and new lines). In the Replace window, use the space-bar to insert spaces.
- \d = select digit
- \D = select non-digit (note upper-case)
- \w = select word (or alphanumeric) character, either a letter or a number
- \W = select non-word character (note upper-case)
- ^ = beginning of line.
- $ =end of a line
- . = the dot: Matches any character except new line. Selects any character within a line as long (**as long as you do NOT check “dot matches all”** in Find & Replace. If “dot matches all,” this will select line breaks too.)

Indicating How Many, Either | Or, and Character Sets [ ]:

- ? = used after a character, picks up zero or 1 of it: so colou?r matches both “color” and “colour”
- * =used after a character, picks up zero or more of it: (the character may or may not be there, and maybe there’s more than one of it). So \w\s\d* picks up a letter followed by a space, as well as a letter followed by a space and a number.
- + =used after a character, picks up 1 or more of it: For example, \d+ picks up either one or more digits, 2 and 25 and 65746, etc.
- | = (the pipe): selects one OR the other: grey|gray or gr(e|a)y are each patterns that will match either grey or gray.

- [ ] matches any ONE character enclosed. Example: [0-9] will select the first single digit from 0-9 that it finds. [IVXLC]+ is handy for picking up one or more Roman Numerals, but be careful because this will also pick up "I" when it's not a Roman Numeral but the first-person pronoun: (I, as in myself). [^IVXLC] will select anything but these characters.

Escaping Regex's Special Characters (When You Need To Find a Square Bracket, Period, Asterisk, Question Mark, Etc.)

Because characters like square brackets, asterisks, and question marks have special meaning in regular expressions, in order to search for a literal square bracket, asterisk, or question mark, you need to *escape* the regex character by using a backslash (\ ). The following characters need to be escaped with a backslash if you need to find the literal character in your text:

- the backslash itself: (\ )
- the caret (^)
- the dollar sign ($)
- the pipe (|)
- the dot (.)
- the question mark (?)
- the asterisk (*)
- the opening and closing parentheses ((and))
- the opening square bracket ([ ), and the opening curly brace ({)

So, for example, in order to search for a string of alphanumeric characters followed by a literal period, we would write the following expression:

\w+\.

The "backslash w plus" looks up any one or more alphaumeric characters, and the backslash dot looks for the literal period. This might look a little confusing at first, since we use the backslash to introduce specific kinds of regular expression characters (\d, \w, etc.). It might help to think of using the backslash as an escape character whenever you need to locate a character that means something special on its own in regular expressions.

How to Use Parenthetical Grouping in the Find Window and Select Groups with Backreferences in the Replace Window:

When we group patterns in the Find window with parentheses, we can use **backreferences** to select parenthetical groupings by number in the Replace window. We apply a set of **capturing parentheses** to isolate some parts of a pattern we find, if we want to exclude the rest when we go to replace.

- ( ) matches and captures all text enclosed. Groups a collection of characters together in the "Find" window so you can select it in the "Replace" window. We presume here that you set these parenthetical groups side by side, rather than nest them inside each other, so that the groupings read from left to right.
- \1 =under "Replace with", this represents the first instance of text captured using ( ), above, under "Text to find".
- \2 =second ( ) instance captured, as above
- \3 =third ( ) instance captured, as above, etc...
- \0 =capture the entire match regardless of parentheses.

Note that you can use backreferences in any order, and repeat them as needed when you are making replacements, so you can thoroughly remix the regex patterns you've grouped! For examples of backreferencing, see the Regular-expressions.info page on the subject.

For example, I've just gone hunting through our Georg Forster voyage file to see if I can find all the references to days that take this verbal form: the 23rd of April (or the 15th, the 2nd, or the 3rd of whatever month and/or year). Let's say I wanted to isolate only the numbers and not the letters (as in, simply, 23, 15, 2, 3), and wrap those in an element I'll call <day>, and then I also want to keep the rest of that text to immediately follow? What I want to do is change this form: **23rd**, into this: <day>23</day>**rd** . That's a perfect opportunity to use parenthetical grouping in Find and Replace, like this:

- **Find window:** (\d\d*)([a-z]+)

  Notice how we're applying parentheses here to isolate the numerical portion, and then a second set to surround the lower-case character set.

- **Replace window:** <day>\1</day>\2

  Here, I indicate that the "day" element is to sit around the first parenthetical grouping I've isolated: just the numbers. Then I give the second parenthetical grouping that's going to sit right outside. This works in my markup to help me hold only the numerical portion of the date inside a handy XML element.

Note that you might want to use parentheses for reasons other than capturing and backreferencing. For example, you might group a series of options marked with vertical pipes (|) inside a parenthetical group in order to set this group of options apart from the rest of your non-optional regex pattern. In this case, you're using **non-capturing parentheses**, but you can hold capturing parentheses inside, and when you refer to them, you still refer to them working from left to right, from inside the non-capturing parentheses. This can get a little complicated, and we refer you to the Regular-Expressions.info page on "Branch Reset Groups" for details and examples.

Thinking Your Way Through an Autotagging Challenge:

There's no single *one* way to do autotagging on a file: There are always options! Here are some hints:

1. When you begin, one of the things you do is analyze the structure of the document (do a "document analysis") to notice what regular patterns you can find. You don't want to be working on this line by line from the top to the bottom, because the point of autotagging is to collect all the **related** kinds of things across the whole document. Instead, the big decision you need to make is whether to work from the **outside in**, or the **inside out**.

   In other words, do you try to capture all the big outer elements first (the ones that hold most of the other elements inside), and then work your way in? Or go the other way, and start from the inside elements (all the items inside the lists, for example)? Either approach can work, and much depends on the patterns you spot as you analyze your text file.

2. Sometimes you "munge" your file accidentally and need to take steps backward, or start over with a fresh copy of the file--that has happened to us! It can be frustrating--take a break and try it again. (It's also very rewarding when you get it just right!)
3. Try a **close-open** strategy: Quite often, the place where you open a new element is ALSO the place where an old element closes. Can you do two things at once? Look for opportunities to close a tag when you open a new one (or vice versa).

4. When you work on autotagging, you usually do some work at the top and/or bottom of your file to change or eliminate a few things at the start or toward the end of your process. For example, if you try the **close-open** strategy to indicate at the start of a <list> element where the previous <list> ended, you'd write the code like this: </list><list>[regex pattern here]. When you've made your replacements, you'll always have an extra closing </list> tag ahead of your first <list> element, but you can easily just manually delete this one rogue tag when you're cleaning up your file.
5. When up-converting to XML, think about whether you really need or want to preserve things in your text files that function as **pseudo-markup**, that is, things that functioned like structural markup to indicate things like quotations, section divisions, separators between paragraphs. XML tags can be used to mark all these things, and you can apply HTML and CSS later to add dividers as you wish when you publish this in electronic form. But keep in mind as you analyze and convert your documents that you don't really need to preserve formatting for the sake of preserving it. Remember that you want your XML markup (your tags themselves) to hold meaningful information about the structure and content of your document, so you do not really need to include the pseudo-markup in the original text. Systematically removing that pseudo-markup is part of your up-conversion process.

Some useful patterns:

- (a|b) a or b
- x{2,} two or more x's
- p{3} Exactly 3 p's
- q{3,} 3 or more q's
- B{3,5} 3, 4 or 5 B's
- ^(.+)$ Since a caret (^) indicates the start of a line, and the dollar sign ($) indicates the end of a line, and the .+ indicates the presence of some characters inside, this pattern selects lines that contain text (and ignores any lines that are empty). You could run a Replace to work with the capturing parentheses and wrap that content inside an element that makes sense (like <item>). In the Replace window, we'd write <item>\1</item> to tag the text inside the line.
- ^[IVX]+\. .+$ =beginning of a line, any roman numeral less than 50, exactly one literal period, exactly one literal space character, then all characters up to the end of the line
- \s\s Find any sequence of two white-space characters (space, tab, new-line). If you're running a Find and Replace, you might replace these multiple white-space characters with a single \s, or use the spacebar.
- **Replacing line breaks:**: Match the \n (or newline character) in order to "consume" and replace a linebreak. It won't work to try to replace ^ and $, which indicate the start and end of lines, because these are not characters that can be replaced; they are merely *anchors* or indicators.
- Read about how to write a **Lookahead** and **Lookbehind** regex, to look for a pattern of something ahead or behind of a character, or something that is NOT ahead or behind a character. Read about it and look at examples on the Regular-Expressions.info guide to "Lookaround."

Regular Expressions in XPath and XSLT

There are XPath functions dedicated to matching and converting regular expressions: These include the following:

- matches(): This takes two arguments: you designate a first string, and then a second that indicates a particular pattern you're trying to find inside it. For example, if you were looking in all the paragraphs of a document coded with <p> to find any paragraphs that contain at least a single digit):
  //p[matches(., "\d")]

  Remember, the dot in the XPath indicates that you're looking at the string of text inside each paragraph in turn, and that is the first string. Then the second string is the regular expression pattern \d, which indicates a pattern to search for any numerical digit inside the string of text in the paragraph.

  Note: There are three other related XPath functions that work like matches(), only these work on literal strings, not regex patterns. We include them here because you may find them useful to think about in connection with matches():

  - contains(): Tests whether the first string contains a particular **literal string**. To adapt our example above, say we are looking for all the paragraphs that contain a mention of the specific year 1995. We'd use contains() much like we'd write matches(), but this time using the literal characters.
    //p[contains(., "1995")]

    (Note: You can actually write matches() to look for a literal string as well as a regex pattern, since one kind of regex actually is a literal string. So, of these two, matches() is the more adaptable XPath function, and contains() can only match on literal strings.)

  - starts-with(): Tests whether the first string *starts with* a particular literal string.
  - ends-with(): Tests whether the first string *ends with* a particular literal string.
- replace(): This function has three parts in its parenthetical expression: replace(string, regex, replacement-string), and works like this, for example, if we wanted to go look in any <author> element for capital letters, and replace them all with literal asterisk characters:
  //author/replace(., "[A-Z]", "*")

  Here, the regex pattern is described in the middle expression to define the pattern we're looking for, and it's a defined *character set*: This says, look for any single character from the set [A-Z] and replace it with a "splat" or an asterisk. When I ran this XPath expression on our ForsterGeorgComplete.xml file, I converted **Forster, Georg** in an author tag to ***orster, *eorg**. (Fortunately this was just a tester XPath, and it didn't change the string of text in my file, just in the XPath results window.)

- tokenize(): This one is extremely handy for fine-tuning XML markup: We use the tokenize() function for a sort of surgical precision in our documents, to break patterns into parts (or "tokens"), by dividing on a particular regex pattern: tokenize(string, regex-pattern), and the output breaks my string into parts that I can grab and work with. For example, I'll go looking for <author> elements again to grab their text, and **tokenize** it on white space, defined as a regex pattern by \s+:
  //author/tokenize(., "\s+")

  When I run this in the XPath window, I return (among other things), a list that separates "George" from "Forster.". (When we tokenize on white space, it's a good idea to work in the option for *one or more* spaces, in case we have a line break as well as a space character separating two parts of a thing.)

In XSLT, there is an element, xsl:analyze-string that we use for manipulating regular expressions, and you can read more about it in the Michael Kay book if you have it, or on the Obdurodon site's tutorial on using analyze-string.

<oo>→<dh> Digital humanities

Author: Janis Chinn (janis.chinn@gmail.com)
Maintained by: David J. Birnbaum (djbpitt@gmail.com)
Last modified: 2018-02-26T03:28:40+0000

Introduction to XSLT

The basics

You already know how to mark up (XML), constrain (Relax NG), and navigate (XPath) your documents; XSLT (*eXtensible Stylesheet Language Transformations*) is one way to transform your document, manipulate the tree, and output the results as XML, HTML, SVG, or plain text. You might use XSLT to generate project pages for display on your site, to generate intermediary pages for analysis and development, or to feed pieces of your data into another format for analysis with another tool, one that requires data in a particular format that is different from your main XML structure. Since XSLT is XML-aware, it uses XPath to navigate and manipulate your document, which means that when you use XSLT to implement a transformation (see below), you automatically use XPath within XSLT to find the pieces you want to transform (XPath expressions and XPath patterns) and to manipulate the data (XPath expressions).

An XSLT stylesheet is an XML document that must be valid against the XSLT schema. The root element in this schema is `<xsl:stylesheet>` and the children of the root are primarily `<xsl:template>` elements. These *template elements* typically have a `@match` attribute that matches an *XPath pattern* and instructs the computer to use that template to process all matching nodes. For example, a template node that matches `<p>` elements will be used to process `<p>` elements in the input document.

XSLT is a *declarative* programming language (unlike most programming languages with which you are likely to be familiar), which means that part of the way it works is that the templates don't get applied from the top of the file to the bottom. What happens instead is that program execution passes from template to template because an `<xsl:apply-templates>` element inside a template rule tells the system what to process next. One consequence of this model is that *the order of template rules inside the stylesheet doesn't matter* because they don't get applied in that order. Rather, they get applied whenever an `<xsl:apply-templates>` element or the equivalent specifies that a particular type of node must be processed. When that happens, for every element or other object in your input document, if there is a template anywhere in the stylesheet that matches it, the stylesheet will find it and the template will fire.

XSLT builds in default rules to handle nodes for which there is no explicit template rule, which means that you have to write your own template rules only where you want something other than the default behavior. The default behavior is that if you try to apply templates to an element for which you haven't created an explicit template, the system will pass silently into that element and apply templates to its children, until eventually the only thing left is to output the text. For that reason, if your stylesheet contains no templates at all, applying the stylesheet to the document will output all the plain text in your XML, without any markup; the default behavior will navigate from the document node at the top of the tree all the way down, outputting text whenever it encounters it. (This is rarely what you want!)

A typical stylesheet has the following exoskeleton, which <oXygen/> will generate for you when you create a new XSLT document:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="3.0">

</xsl:stylesheet>
```

For most purposes, you'll want to be sure the `@version` attribute is set to 3.0, which should be the default behavior in <oXygen/> (and you can make it the default if it isn't already).

Remember that running an XSLT stylesheet that contains no template rules on your XML will essentially strip out all your markup and output plain text. This is rarely what you want. Typically you'll need to add at least one template rule to generate useful output.

Namespaces

The input namespace

If your XML document is in a namespace, you'll need to tell your stylesheet about the namespace in order to process it with XSLT. To do this, add an `@xpath-default-namespace` attribute to the root `<xsl:stylesheet>` element and set its value to the value of the namespace declaration from the input XML file. For example, if you are transforming a TEI XML document with the following namespace declaration:

```
<TEI xmlns="http://www.tei-c.org/ns/1.0">
```

the root `<TEI>` element states that all elements within the document are in the TEI namespace (unless you explicitly say otherwise). If you were to write a template rule in your XSLT matching just "TEI", it wouldn't be applied, because the system would be looking for `<TEI>` elements *in no namespace*, whereas the XML declares that the `<TEI>` element is in the `http://www.tei-c.org/ns/1.0` namespace. (Generally, if you run a transformation where you have template rules, none of them gets applied, and you just get plain text in the output [as if you had had no template rules], it's because of mismatched namespaces. In that situation, no template rules are being applied because they only match elements in no namespace and all of the elements in your input XML are in a namespace, which means that the transformation falls back on the default behavior described above.) To tell your stylesheet always to look for elements in the TEI namespace, our `<xsl:stylesheet>` element should look something like this:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="3.0" xpath-default-namespace="http://www.tei-c.org/ns/1.0">
```

Should you have input in mixed namespaces (perhaps a TEI document in the TEI namespace that contains embedded SVG in the SVG namespace), see your instructors for guidance about how to deal with it.

The output namespace

The `@xpath-default-namespace` attribute specifies the namespace of the *input* XML. If your output is going to be in a namespace (for example, if you are outputting HTML, which must be in the HTML namespace), you also need to specify the output namespace. When outputting HTML, the namespace declaration is "http://www.w3.org/1999/xhtml", so if you are transforming TEI to HTML, your root `<xsl:stylesheet>` element must read:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="3.0" xmlns="http://www.w3.org/1999/xhtml"
    xpath-default-namespace="http://www.tei-c.org/ns/1.0">
```

The green text says that the default namespace for all *elements that you are creating in your output document* is the HTML namespace. The blue text says that the namespace for all *elements in your input document* is the TEI namespace. If your input or output are in no namespace you should omit these declarations, and if they are in other namespaces, you'll need to use the appropriate namespace values.

Controlling your output with `<xsl:output>`

You should always have an `<xsl:output>` element to control the type and formatting of your output. `<xsl:output>` is a *top-level element*, which means it must be a child of the root `<xsl:stylesheet>` element (making it a sibling to all your template rules, which are also top-level elements). `<xsl:output>` is usually placed at the top of the document, as a first child of the root `<xsl:stylesheet>` element, because that makes it easier for humans to find, but as long as it is a child (not a grandchild or other descendant) of the root element, your document will be valid. Officially, `<xsl:output>` is an optional element, which means that if it's omitted you won't get an error message, and the system will try to guess the kind of output you want, which can lead to errors if it guesses wrong. At minimum, `<xsl:output>` should have a `@method`

attribute. You may also need to set a value for the optional `@indent` and `@doctype-system` attributes. Here are some guidelines:

- `@method` specifies the type of output, and the accepted values are "xml", "html", "xhtml", and "text". The correct output method for all XML documents (including HTML5 documents) is "xml" (that's right; we use "xml", rather than "html" or "xhtml", for HTML documents). The only other value we use in our own work is "text" (for plain text output).
- The `@indent` attribute specifies whether or not the output should be pretty-printed, that is, indented in a way that wraps long lines and makes it easy to see the hierarchical structure. We normally set this to "yes" because it makes the output easier for humans to read. Because this type of indentation works by inserting spaces and new-line characters, there are some situations where automatic indentation can mess up your content, and in those situations you can use this attribute to turn off the indentation. XML (including HTML5) doesn't normally care about the indentation, so whether you turn it on or off is just for the convenience of the human who may need to look at the angle-bracketed output of the transformation. HTML5 output will be wrapped properly in the browser even if you turn off indentation and the angle-bracketed view looks like one long line.
- If you are creating HTML5 output, you will also need to include the `@doctype-system` attribute, the value of which must be "about:legacy-compat".

For HTML5, then, putting it all together, you should use:

```
<xsl:output method="xml" indent="yes" doctype-system="about:legacy-compat"/>
```

Telling templates when to fire by using the `@match` attribute with an XPath *pattern*

Except in situations you are unlikely to encounter in our course, `<xsl:template>` requires the attribute `@match`, which matches an XPath pattern. *An XPath pattern is not the same as a full XPath expression*; it is just a piece of one, the minimum XPath needed to describe what you want to match. For example, to match all `<p>` elements in the document, write `match="p"` instead of `match="//p"`. In other words, templates don't specify where to look for the elements they match because they sit around waiting for the elements to come to them (courtesy of `<xsl:apply-templates>` or built-in processing rules), and for that reason they only have to describe what it is that they match, and not how or where to find it.

With that said, by varying the completeness of the pattern, you can get more or less specific about how to handle, say, `<p>` elements in different parts of the XML tree. If you want to treat `<p>` elements inside a `<chapter>` differently from `<p>` elements inside an `<introduction>`, you can create separate templates that match "chapter/p" and "introduction/p", with as little context as you can get away with to specify the difference. But you don't need (= shouldn't have) a full path; *your XPath pattern must be the simplest pattern that will match what you want to match.* Most of your stylesheets will consist of `<xsl:template>` elements for each type of element that might arise in your input document (unless the built-in behavior, described above, which applies if there is no template, already does what you want, in which case you should not create an explicit template just to mimic that behavior).

Most (if not all) stylesheets you'll write in this course will begin functionally with a template matching the *document node*, which is both the (generally invisible) parent of the root element and the uppermost node in the hierarchy of every XML document. When an XSLT stylesheet is applied to an XML document, the system always starts at the document node when looking for templates to apply. To match the document node, use the XPath pattern "/". Any instructions that should fire only once to create the superstructure for your output will typically be created inside this template, and you'll need at least one `<xsl:apply-templates>` element in order to interact with the lower branches of your tree. If you're planning on outputting HTML, the template that matches the document node is the place to create your HTML superstructure, and within this superstructure you'll want to include, typically, an `<xsl:apply-templates>` element that tells the processor how to build the HTML output inside that superstructure. For example, a typical XML-to-HTML transformation might start with code like:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    exclude-result-prefixes="xs"
    version="3.0" xmlns="http://www.w3.org/1999/xhtml">
    <xsl:output method="xml" indent="yes" doctype-system="about:legacy-compat"/>
    <xsl:template match="/">
        <html>
            <head>
```

```
                <title>Title goes here</title>
            </head>
            <body>
                <xsl:apply-templates/>
            </body>
        </html>
    </xsl:template>
</xsl:stylesheet>
```

This matches the document node, creates the HTML superstructure that will go into the output, and then, inside the HTML `<body>` element, applies templates to the children of the document node (by default, `<xsl:apply-templates>` means "apply templates to the children of the node currently being processed"). The only child element of the document node is always the root element of your input XML. Your stylesheet will also include other templates that specify what to do with the various elements of your input XML (see below).

Think of your `<xsl:apply-templates>` elements as place-holders that mark where to output the results of applying the templates they call. For example, any content you want to appear immediately inside the HTML `<body>` element that you're creating can be placed correctly by putting the `<xsl:apply-templates>` element between the `<body>` start and end tags.

XPath *expressions* vs XPath *patterns*

One common source of confusion for new XSLT coders involves the difference between `<xsl:template match="XPath pattern">` and `<xsl:apply-templates select="XPath expression"/>`. The terms are unfortunately similar, but here is how they work:

- Template rules start with `<xsl:template match="XPath pattern">` and describe what the system will do when something that matches the pattern happens to wander by. The pattern is "XPath-like", which is say that it uses a subset of XPath to describe what it will match. Don't begin a pattern with `//`; templates don't have to look for elements, so they don't need full paths. A template rule that begins `<xsl:template match="div">` will match any `<div>` element that needs to be processed, no matter where it's located in the document.
- Inside a template you specify what the system should do when it encounters the item (usually an element) matched by the `@match` attribute. A template can create new elements in the output, and it can also instruct the processor about which elements to process next. The principal way it does the latter is with `<xsl:apply-templates>`. By itself, an `<xsl:apply-templates>` element means "process (all) the children (elements and `text()` nodes) of the current context node." While this is the most common way to use `<xsl:apply-templates>`, you can tell it to process anything at all in the input XML document (and even in other documents) by including a `@select` attribute, as in `<xsl:apply-templates select="XPath expression">`. The value of the `@select` attribute is a full XPath expression, and can point to any nodes on any axes.

Processing something other than immediate child nodes

By default, `<xsl:apply-templates>` means "apply templates to all child nodes (elements and text) of the current context, that is, the node currently being processed". You are not restricted to processing only child nodes, though; `<xsl:apply-templates>` optionally takes a `@select` attribute, which tells the system what nodes to apply templates to. The value of `@select` is a full XPath expression and will start from the *current context*, that is, from whatever node is being processed at the time. For example, if you are transforming TEI to HTML and the only XML you want to process is in the `<teiHeader>`, you can replace the general `<xsl:apply-templates>` with `<xsl:apply templates select="//teiHeader">`. If `@select` is omitted, the system will default to applying templates to all descendants of the current node. This behavior means that you don't need to (= shouldn't) specify `@select` if what you want to select is all of the children of the current context.

`<xsl:apply-templates>` is usually an empty element, but you may include `<xsl:sort>` between separate start and end tags to sort the nodes you're applying templates to. By adding `@select` and `@order` to `<xsl:sort>` (see Michael Kay for details), you can specify what to sort by (the default is the textual value of the element, but you can override that) and whether to sort in ascending or descending order (the default is ascending).

Any elements you want to handle specially (that is, for which the built-in behavior is not what you want) will need their own template rules. Remember, though, that templates fire every time the system encounters a matching node in the XML, so if you want an element to be created once (for instance, the `<html>` element), it should go within a template that matches a node that only appears once (for instance, `/`). If you're generating HTML `<p>` elements, on the other hand, you'll need those

to be inside a template that will fire many times because you want to generate many <p> elements, not one giant <p> element which contains the text of all the paragraphs. Similarly, if you are creating an HTML table with a lot of rows, you typically want only one table, so you should create that directly inside the <body> element and then create the <tr> elements for the rows in a template that fires once for each row you want to create. If, say, you want to create one table row for each <character> element in your input, your XSLT will probably look something like:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xs="http://www.w3.org/2001/XMLSchema" exclude-result-prefixes="xs" version="3.0">
    <xsl:output method="xml" indent="yes" doctype-system="about:legacy-compat"/>
    <xsl:template match="/">
        <html>
            <head>
                <title>Title goes here</title>
            </head>
            <body>
                <table>
                    <tr>
                        <!-- header row with <th> elements to label the columns -->
                    </tr>
                    <xsl:apply-templates select="//character"/>
                </table>
            </body>
        </html>
    </xsl:template>
    <xsl:template match="character">
        <tr>
            <!-- apply other templates to create the cells in the table row
                 for a particular character-->
        </tr>
    </xsl:template>
</xsl:stylesheet>
```

Note that you create only one <table>, so you do that inside a template that fires just once, the template that matches the document node. You create one row for each character, though, so you create your <tr> elements inside a template that matches <character> elements, and therefore fires once for each <character> element. The only <tr> that gets created inside the template rule for the document node is the one that labels the columns, since you want just one of those.

<xsl:apply-templates> vs. <xsl:value-of>

Sometimes you get the content of an element or attribute by applying templates to it and sometimes you use <xsl:value-of>. The difference between <xsl:apply-templates> and <xsl:value-of> is that <xsl:value-of> can return only plain text, that is, the textual content of a node (throwing away any markup), as well as the results of many functions and other *atomic* values (an atomic value is essentially any value that isn't a node, such as a string or a number). The result of <xsl:value-of> is always an atomic value, and it represents a dead end in the XML tree insofar as it cannot contain markup, which means that you cannot apply templates to any part of it. If, for example, you are processing a paragraph node tagged as <p>, <xsl:value-of select="."/> will return the textual value of the paragraph, throwing away any internal markup. If you want to process that internal markup (for example, if the paragraph contains titles or foreign words or emphasis or anything that should be processed separately), <xsl:value-of> will make it impossible to process those elements, and all you'll get is their textual content, as if they weren't marked up in the first place. If, on the other hand, the paragraph has no internal markup, there is no difference in behavior between <xsl:apply-templates> and <xsl:value-of>. As a rule of thumb:

- Use <xsl:apply-templates> when processing a node (element, attribute) unless there is good reason to do otherwise. If there's no difference (because you're processing an element that just contains plain text), this will do what you want. Where there is a difference, though, using <xsl:value-of> will throw away internal markup without processing it, which is rarely what you want.
- Use <xsl:value-of> when outputting an atomic value, such as the value of an XPath function that retuns a string or a number.

Both <xsl:apply-templates> and <xsl:value-of> can take a @select attribute to specify what should be processed. That attribute is optional with <xsl:apply-templates> (if you don't use @select, you will apply templates to all of the child nodes of the current context, whatever they may be), but the @select attribute is obligatory with

`<xsl:value-of>`. (The `<xsl:value-of>` element optionally also accepts a `@separator` attribute, which allows you to specify a separating string to use when `<xsl:value-of>` outputs a sequence of values. The result is similar to the specification of a separator in the `string-join()` function.) If you're curious, you can read more about the differences between `<xsl:apply-templates>` and `<xsl:value-of>` in our guide to advanced XSLT features.

White space

As you know, white space is generally normalized automatically when processing XML documents. But what if you need to preserve the white space from your original document in your transformation? How do you distinguish that situation from one where there's extra white space in your XML document because it was pretty-printed (lines wrapped and extra spaces used for indentation), and the white-space isn't meaningful and shouldn't be retained? Although these cases aren't common, when they do come up they are critical to outputting your document correctly. To resolve them you'll want to use some combination of `<xsl:preserve-space>` or `<xsl:strip-space>`. These are both top-level elements (children of the root `<xsl:stylesheet>` element) that take the attribute `@elements`, the value of which is a space-delimited list of elements whose white space you want to preserve or strip out. If you want to affect all the elements in the document, you can set the value of the `@elements` attribute to `*`. Typically, XSLT will do what you expect and you won't need to use these elements at all. If a problem arises, though, you can use `<xsl:preserve-space>` or `<xsl:strip-space>` to override the default behavior and control the processing manually.

Outputting mixed content

XSLT usually does The Right Thing when it is outputting just elements or just plain text, but mixed-content output (that is, a mixture of elements and plain text) can lead to awkward white-space handling. You can avoid having to worry about the intricacies of XSLT white-space handling by applying the following rule of thumb: *when you are outputting mixed content, wrap all plain text in `<xsl:text>` tags.* For example, instead of writing:

```
<xsl:template match="book">
    <item>
        <cite>
            <xsl:apply-templates select="title"/>
        </cite>
        by
        <xsl:apply-templates select="author"/>
    </item>
</xsl:template>
```

you should use:

```
<xsl:template match="book">
    <item>
        <cite>
            <xsl:apply-templates select="title"/>
        </cite>
        <xsl:text> by </xsl:text>
        <xsl:apply-templates select="author"/>
    </item>
</xsl:template>
```

Putting it all together

By way of illustrating a complete transformation here are a sample XML doucment (whose content you may recognize from the first week of class) and a sample XSLT stylesheet to transform the XML into HTML for publication on the web.

```
1 <letter>
2     <head>
3         <context>The following letter was written shortly after Wilde's
4         release from prison:</context>
5     </head>
6     <content>
7         <dateline>
              <location>Rouen</location>
```

```
            <location>Rouen</location>,
            <date>
                <month>August</month>
                <year>1897</year>
            </date>
        </dateline>
        <salutation><person type="recipient">My own Darling Boy</person>,</salutation>
        <body>
            <p>I got your telegram half an hour ago, and just send a line to say that
                I feel that my only hope of again doing beautiful work in art is being
                with you. It was not so in the old days, but now it is different, and
you
                can really  recreate in me that energy and sense of joyous power on
which
                art depends.</p>
            <p>Everyone is furious with me for going back to you, but they don't
                understand us. I feel that it is only with you that I can do anything
at
                all. Do remake my ruined life for me, and then our friendship and love
                will have a different meaning to the world.</p>
            <p>I wish that when we met at <location>Rouen</location> we had not parted
at
                all. There are such wide abysses now of space and land between us. But
we
                love each other.</p>
        </body>
        <valediction>Goodnight, dear. Ever yours, <person type="sender">Oscar</person>
        </valediction>
    </content>
</letter>
```

The XML is pretty straightforward. The root element is `<letter>`, has two children, a `<head>` and a `<content>` element, and the latter contains the body of the letter and the rest of the element. Locations within the text are tagged, but for the sake of simplicity and brevity, the sender and recipient are tagged only in the salutation and valediction, as `<person>` elements. (That is, the personal pronouns that refer to them in the body of the letter are not tagged.)

Our sample output will be an HTML document that does not include any information from the `<head>` element; it outputs our paragraphs as HTML paragraphs and italicizes all persons and locations. The result of the transformation can be seen below:

***Rouen*, August 1897**

My own Darling Boy,

I got your telegram half an hour ago, and just send a line to say that I feel that my only hope of again doing beautiful work in art is being with you. It was not so in the old days, but now it is different, and you can really recreate in me that energy and sense of joyous power on which art depends.

Everyone is furious with me for going back to you, but they don't understand us. I feel that it is only with you that I can do anything at all. Do remake my ruined life for me, and then our friendship and love will have a different meaning to the world.

I wish that when we met at *Rouen* we had not parted at all. There are such wide abysses now of space and land between us. But we love each other.

Goodnight, dear. Ever yours, *Oscar*

This is relatively simple to accomplish. The stylesheet is included below, followed by a discussion of how it works:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns="http://www.w3.org/1999/xhtml" xmlns:xs="http://www.w3.org/2001/XMLSchema"
    exclude-result-prefixes="xs" version="3.0">
```

```
4     <xsl:output method="xml" indent="yes" doctype-system="about:legacy-compat"/>
5     <xsl:template match="/">
6         <html>
7             <head>
8                 <title>Oscar Wilde Letter 2</title>
9             </head>
10            <body>
11                <xsl:apply-templates select="//content"/>
12            </body>
13        </html>
14    </xsl:template>
15    <xsl:template match="dateline">
16        <h4>
17            <xsl:apply-templates/>
18        </h4>
19    </xsl:template>
20    <xsl:template match="location|person">
21        <em>
22            <xsl:apply-templates/>
23        </em>
24    </xsl:template>
25    <xsl:template match="p|salutation|valediction">
26        <p>
27            <xsl:apply-templates/>
28        </p>
29    </xsl:template>
30</xsl:stylesheet>
```

Lines 1–3 are created by <oXygen/> when you tell it to create a new XSLT stylesheet. The only part that we've added is the HTML namespace declaration on line 2, so that all output will be in the HTML namespace:

```
2     xmlns="http://www.w3.org/1999/xhtml" xmlns:xs="http://www.w3.org/2001/XMLSchema"
```

Line 4 tells the system what type of document we're outputting: an HTML5 document with indenting. Lines 6-13 set up our HTML superstructure (we've added a `<title>`, which will show up in the browser tab, but not in the browser window), populating our `<body>` element with the results of applying templates to all `<content>` elements wherever they appear. There's only one `<content>` element, and no template for `<content>`, so the system falls back on the default behavior and applies templates to all of its children. (Note that we never apply templates to `<head>` or `<context>`, so they will not be output at all in our result document.)

The children of `<content>` are `<dateline>`, `<salutation>`, `<body>`, and `<valediction>`, and we have templates for all of those except `<body>`. That means that we're relying on the default behavior for `<body>`, which is, again, to apply templates to its children. The `<dateline>` element, whose template is on lines 15-19, will process the contents of the element inside an HTML `<h4>` element. There's no `@select` attribute on the `<xsl:apply-templates>` here, so the system will apply templates to all children of the element (there are three: the `<location>` element, the `text()` node after it that contains a comma and some white space, and the `<date>` element). We don't have template rules for the second and third of these, so the built-in rules will take care of them; the `<location>` element is processed by the template on lines 20–24, which outputs the content wrapped in an `<em>` element (typically rendered as italics in the browser).

The template on lines 25–29 actually covers three different elements: `<p>` elements, `<salutation>` elements, and `<valediction>` elements. For all three, it outputs the contents inside an HTML `<p>` element. This way any `<p>`, `<salutation>`, and `<valediction>` element in the input XML will become an HTML paragraph in our output. Since we again applied templates without a select attribute, we again revert to the default behavior of applying templates to all children elements of any `<p>`, `<salutation>`, or `<valediction>` element. Finally, the template on lines 20–24, which we mentioned earlier, will tag the contents of any `<location>` or `<person>` element as an HTML `<em>` element, normally causing it to be italicized in the browser.

<oo>→<dh> Digital humanities

Maintained by: David J. Birnbaum (djbpitt@gmail.com)
CC BY NC SA
Last modified: 2018-03-02T00:21:30+0000

Attribute value templates (AVT)

When you might want to use attribute value templates

Attribute value templates (AVTs) are a strategy for inserting the value of XPath expressions into the attribute value of created content. Two typical situations where AVTs are useful are:

- You are creating a table of contents (TOC) and you want to create links (using the HTML `<a href="#fragment_identifier">` notation) in the TOC that will point to the various sections in the body of the document. Those links might make use of values taken from text in the input XML; for example, if each chapter has a unique one-word identifier in an attribute value, like `<chapter id="introduction">`, you might want your links to read something like `<a href="#introduction">`. You can use an AVT to insert the value of the `@id` attribute from the input XML into the value of the `@href` attribute of the output HTML.
- You are creating a research paper with footnotes. In your input XML, you've written the footnotes in the body of the text, to keep them where they belong logically, e.g.,

  ```
  <paragraph>Here is a sentence. Here is another sentence.<fn>Here
  is a footnote, the number for which will go here.</fn> And here is
  one more sentence, after the footnote number.</paragraph>
  ```

  You want this to be rendered like the following, with the footnote number inserted automatically and the footnote text rendered at the bottom of the page:

  ```
  Here is a sentence. Here is another sentence.[1] And
  here is one more sentence, after the footnote number.
  ```

  The footnote number should be generated automatically, so that when you add or delete footnotes from the XML, the XSLT will still generate correct, consecutive numbers in the output HTML. Furthermore, you want the footnote numbers to be links, so that the user can click on them to jump to the footnote text (and click on the footnote text to jump back to the previous location, although the back button would work in this case, as well). In this situation you can generate the numbers automatically by using XPath to figure out which footnote is which, and you can use an AVT to incorporate those numbers into the values of the `@href` attributes that will take the user to the notes themselves. For example, you might create HTML like:

  ```
  <p>Here is a sentence. Here is another sentence.<sup><a href="#note1">1</a></sup>
  And here is one more sentence, after the footnote number.</p>
  ```

  This output uses the HTML `<sup>` element to create a superscript number, and it puts the `<a>` element inside that to make the number a clickable link. The numerical value itself, inside the `<sup>` element and again at the end of the value of the `@href` attribute, is generated by the XSLT. For example, the next footnote would have "2" as its number and "#note2" as the value of its `@href` attribute.

How AVTs work

A bit of background: When we create output HTML elements during an XSLT transformation by just typing the raw HTML tags into the stylesheet, we are creating what are called *literal result elements* (LREs). As the name implies, when you want to have the *element* markup (the tags) inserted *literally* into the output *result*, you just type it in the appropriate place in your stylesheet and it gets created in the output tree. For example, in the following template rule:

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

the <p> that is being constructed is an LRE because it is being specified literally inside the template rule. An LRE may contains attributes, so if you want to give your paragraph a particular attribute value, you can create that value literally, as well, using code like the following (the attribute and its value are highlighted):

```
<xsl:template match="paragraph">
    <p class="interesting">
        <xsl:apply-templates/>
    </p>
</xsl:template>
```

The preceding template rule will take any <paragraph> element in the input document, create a corresponding <p> element in the output, associate a @class attribute with the value "interesting" with the <p> element, and process the children of the original paragraph, inserting the content they generate inside the newly created <p> element.

In the immediately preceding example, where we are labeling all paragraphs as "interesting", specifying that label as part of the LRE is all we need. In the two situations described earlier, however, TOC and footnotes, we don't want every element to have the same attribute value, so can't just write the attribute value directly into the LRE, as we did with the uniform class="interesting" example. In the TOC, we want each chapter title to point to a different part of the document with a unique identifier, and in the footnote example we want each footnote to have a unique number, and the associated link to have a unique value. An AVT lets us construct these attribute values on the fly, adapting the value to the situation for each affected node.

Using an AVT to link from a TOC into the body (and back)

An AVT is created by wrapping some XPath inside curly braces. For example, assuming chapters in our input XML document are <chapter> elements with unique identifiers in an associated @id attribute (e.g., <chapter id="introduction">), we can create TOC links with a template rule like (don't worry about the @mode attribute for the moment; we'll cover that shortly):

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

This will output something like:

```
<li>
    <a href="#introduction">introduction</a>
</li>
```

The LRE is output literally except for the part of the attribute value that is inside the curly braces (highlighted in the example above). That part is an AVT, and instead of being output literally, it is interpreted as an XPath expression (relative to the current context node, the <chapter> being processed), and the value of the expression is inserted into the attribute value. In this case, that means that the value of the @id attribute on the <chapter> element in the input XML document is copied into the output HTML as part of the value of the created @href attribute. The curly braces are not output themselves; they serve only to delimit the AVT.

To make the linking work, in the body of the document, where the chapter text itself is printed, you would have to create a target for the link by using the <a> element with a @name or @id attribute. That is, the @href attribute specifies where the user will go upon clicking the link, and the @name and @id identify parts of the document as potential targets to which @href attributes might point. The XHTML specification prefers @id, rather than @name, for specifying the target of a link (see http://www.w3.org/TR/xhtml1/#h-4.10), and in my work I often use both, set to the same value. To make the links bidirectional, add both @href and @name/@id attributes to the <a> elements on both ends of the link. In that case:

- The entry in the TOC might read:

```
<li>
    <a href="#introduction" name="introduction_toc" id="introduction_toc">introduction</a>
</li>
```

- The chapter title inside the body of the document, above the chapter text, might read:

```
<h2>
    <a href="#introduction_toc" name="introduction" id="introduction">introduction</a>
</h2>
```

Note, though, that the value in the `@href` attribute begins with a hash mark (“”#”) and the value in the `@name` and `@id` attributes doesn’t. The `<a>` inside the `<li>` in the TOC is called “introduction_toc” and the `<a>` inside the `<h2>` in the main body is called “introduction”. Each points to the other by setting the value of the `@href` attribute to the value of the `@name` or `@id` of the target, preceded by a hash mark (“#”).

Using an AVT to number footnotes and create links

If footnotes are encoded as `<fn>` elements inside the text, the logical number of each footnote is equal to the number of preceding footnotes in the document plus one (without the “plus one”, the first footnote would erroneously be considered number zero, since it has no preceding `<fn>` elements). The following template rule will create the footnote number in place of the footnote text:

```
<xsl:template match="fn">
    <sup><xsl:value-of select="count(preceding::fn) + 1"/></sup>
</xsl:template>
```

This uses the XPath `count()` function to count the number of `<fn>` elements on the `preceding` axis (that is, the number that precede the `<fn>` being processed at the moment) and adds one to that number. To add a link to the footnote itself, which you'll render in a set of notes at the end of the page, change the template to:

```
<xsl:template match="fn">
    <sup>
        <a href="note{count(preceding::fn) + 1}">
            <xsl:value-of select="count(preceding::fn) + 1"/>
        </a>
    </sup>
</xsl:template>
```

This sets the value of the `@href` attribute as the concatenation of the string “note” plus the value of the (highlighted) AVT inside the curly braces (`count(preceding::fn) + 1`). For the first footnote, this will produce:

```
<sup><a href="note1">1</a></sup>
```

The preceding code inserts the footnote numbers into the main text and makes the links clickable, but you also need to output the footnotes all together at the end of the page. You can do that by using `<xsl:apply templates select="//fn" mode="fn"/>` after you output the main text and then writing a modal template rule to process footnotes differently from the way they are processed in the main body of the document. As with the TOC, you’ll need to create `@name` and `@id` attributes on the targets of the links, and you can make the links bidirectional.

An alternative to attribute value templates

Attribute value templates are concise and legible, and they are the strategy we use most in our own work. There is an alternative, though, the *attribute constructor.* Instead of specifying the attribute and its value as part of the LRE, you can specify just the element name as a LRE and then specify the attribute name and value with an attribute constructor. The following template rules are exactly equivalent, and in both cases it is the highlighted parts that create the `@href` attribute and set its value:

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

```
<xsl:template match="chapter" mode="toc">
    <li>
        <a>
            <xsl:attribute name="href">
                <xsl:text>#</xsl:text>
                <xsl:value-of select="@id"/>
            </xsl:attribute>
            <xsl:apply-templates select="@id"/>
        </a>
    </li>
</xsl:template>
```

There is, by the way, also an *element constructor* (`<xsl:element>`), which can be used as an alternative to an LRE. See Kay for details. In most cases LREs with calculated attributes encoded through AVTs will do the job, and there is no need for element or attribute constructors. They are available, though, as alternatives, and for certain complex types of calculated content they may provide the only workable strategy. We'd recommend that you:

- Learn to use AVTs in your own work.
- Remember that element and attribute constructors (`<xsl:element>`, `<xsl:attribute>`) exist, but don't worry about them unless you run into a computed markup situation that looks like it can't be resolved any other way.

<oo>→<dh> Digital humanities

Maintained by: David J. Birnbaum (djbpitt@gmail.com)
Last modified: 2018-03-16T22:34:07+0000

XSLT identity transformation

The XSLT *identity transformation* is used to transform an XML document to itself, that is, to generate XML output that is identical to the XML input. By itself this is not very useful, since there are more computationally efficient ways to produce an identical copy of a document. Where it pays off, though, is if you want to make an *almost* identical copy, except that you want to introduce a small but systematic change or two. You can do this by using the identity template to transform everything in the document except the parts that you want to modify. The result is that you produce a copy of the document that is identical to the original except that it includes your modifications.

For example, you might have a document filled with sonnets with a structure like:

```
<sonnet number="I">
    <line>From fairest creatures we desire increase,</line>
    <line>That thereby beauty's rose might never die,</line>
    <line>But as the riper should by time decease,</line>
    <line>His tender heir might bear his memory:</line>
    <line>But thou contracted to thine own bright eyes,</line>
    <line>Feed'st thy light's flame with self-substantial fuel,</line>
    <line>Making a famine where abundance lies,</line>
    <line>Thy self thy foe, to thy sweet self too cruel:</line>
    <line>Thou that art now the world's fresh ornament,</line>
    <line>And only herald to the gaudy spring,</line>
    <line>Within thine own bud buriest thy content,</line>
    <line>And tender churl mak'st waste in niggarding:</line>
    <line>Pity the world, or else this glutton be,</line>
    <line>To eat the world's due, by the grave and thee.</line>
</sonnet>
```

that you'd like to convert to:

```
<sonnet>
    <number>I</number>
    <line>From fairest creatures we desire increase,</line>
    <line>That thereby beauty's rose might never die,</line>
    <line>But as the riper should by time decease,</line>
    <line>His tender heir might bear his memory:</line>
    <line>But thou contracted to thine own bright eyes,</line>
    <line>Feed'st thy light's flame with self-substantial fuel,</line>
    <line>Making a famine where abundance lies,</line>
    <line>Thy self thy foe, to thy sweet self too cruel:</line>
    <line>Thou that art now the world's fresh ornament,</line>
    <line>And only herald to the gaudy spring,</line>
    <line>Within thine own bud buriest thy content,</line>
    <line>And tender churl mak'st waste in niggarding:</line>
    <line>Pity the world, or else this glutton be,</line>
```

```
    <line>To eat the world's due, by the grave and thee.</line>
</sonnet>
```

That is, your input has a `@number` attribute that you'd like to replace with a `<number>` child of the `<sonnet>` element, but you'd like to keep the wrapper `<sonnet>` element and the internal `<line>` elements. To make that change you can start with an identity transformation, which applies templates to every node in the document and tells it to reproduce itself unchanged in the output, except that you also write a template that handles the numbering specially.

The identity template looks like:

```
<xsl:stylesheet version="3.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="@*|node()">
        <xsl:copy>
            <xsl:apply-templates select="@*|node()"/>
        </xsl:copy>
    </xsl:template>
</xsl:stylesheet>
```

The preceding is an entire XSLT stylesheet that transforms a document to itself unchanged. The way it works is that it matches all attributes (`@*`) and all other nodes (`node()`; this includes the document node and all elements, `text()` nodes, comments, etc.), makes a copy of those, and then applies templates to any attributes or other nodes associated with whatever it's processing at the moment. Since an XSLT transformation starts automatically at the document node (and because the document node is a node, this template will match it), you wind up processing everything in the document, moving down level by level through the hierarchy. Because `text()` nodes are nodes, too, this also winds up copying all of the text.

There are two subtle details that enable such a simple template to do so much work:

1. The `<xsl:copy>` element makes a *shallow copy*. This means that it creates a copy of the node that it is processing, but it doesn't do anything with any attributes or child elements or textual content. For example, if you apply `<xsl:copy>` to a `<sonnet>` element in the example above, it will create a `<sonnet>` element in the output, but it won't copy the `@number` attribute or the `<line>` child elements. This is why we need to apply templates explicitly inside the `<xsl:copy>` tags.
2. We have to apply templates to both nodes and attributes. We don't usually think of it this way, but because XPath looks by default at the child axis, when we apply templates to `node()`, we're applying templates to all nodes *on the child axis* of the current context (that is, of the element we matched in the template that is doing the work at the moment). In other words, `<xsl:apply-templates select="node()"/>` is synonymous with `<xsl:apply-templates select="child::node()"/>`. Since *attributes are on the attribute axis, and not on the child axis*, this would not apply templates to attributes, which means that attributes on elements in the input document would be lost during the transformation. To avoid losing them, we have to apply templates to the union (using the union operator `|`) of anything on the attribute axis (`@*`) and any node on the child axis (`node()`).

One more bit of magic is that XSLT templates have *precedence* rules, which specifies what happens when more than one template matches a node that is being processed, and we exploit those rules to override the identity template when we want to change something during the transformation. The most important precedence rule is that *the more specific `@match` value wins*. Since the `@match` value of the identity template, above, is very general (it matches any attribute and any other node), just about any other `@match` value would be more specific. What we'll do for our present task, then, is let the identity template take care of everything in our document except the bits that we want to change. The identity template will match

absolutely everything in the input document, but our more specific template for what we want to change will override it where we need it to.

This simplified example contains just two element types: `<sonnet>` and `<line>`, but in Real Life you would have other elements: a root element (perhaps `<sonnets>`), some metadata, headers or titles, and perhaps more. We want to leave all of those other element types unchanged, and we also want to leave `<line>` unchanged, but we want to make two changes to `<sonnet>`:

1. We want to remove the `@number` attribute, and
2. We want to add a `<number>` child element.

A full XSLT stylesheet to do that would be:

```
<xsl:stylesheet version="3.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:output method="xml" indent="yes"/>
    <xsl:template match="@* | node()">
        <xsl:copy>
            <xsl:apply-templates select="@* | node()"/>
        </xsl:copy>
    </xsl:template>
    <xsl:template match="sonnet">
        <xsl:copy>
            <number>
                <xsl:value-of select="@number"/>
            </number>
            <xsl:apply-templates/>
        </xsl:copy>
    </xsl:template>
</xsl:stylesheet>
```

The identity template matches everything in the input document: the document node, all elements, all attributes, and all `text()` nodes. Both templates match `<sonnet>` elements; the second one matches them specifically and the identity template matches them because it matches all nodes. This means that the identity template will be used to make no changes in anything else, but the second template, which has the more specific `@match` attribute value and therefore outranks the identity template in case of a tie, will be the one that gets to handle the `<sonnet>` element. What the sonnet-specific template does is copy the element it just matched (that is, create a shallow copy of the `<sonnet>` element in the output) and then, inside the element it has just created in the output, create a new `<number>` child element, the content of which is the value of the `@number` attribute that was on the original `<sonnet>` we're processing. Below that new `<number>` child element of the `<sonnet>` we apply templates without a `@select` attribute, which means that we apply templates to all of the child nodes of the `<sonnet>` we're processing at the moment. Those child nodes are the `line` elements of the sonnet, and when we apply templates to them, the identity template does the processing and just copies them unchanged to the output.

So what happened to the original `@number` attribute? Attributes are not children because they aren't on the child axis; they're on the attribute axis. This means that in our template that matches `<sonnet>` elements, our `<xsl:apply-templates/>` without a `@select` attribute applies templates to all of the *children* of the current context (in this case, the `<line>` elements), but not to its attributes. This means that the original `@number` attribute disappears because we simply don't apply templates to it.

The XSLT 3.0 way

The identity template described above does the job well, and you'll encounter it widely, including in tutorials and homework answers in this course. But since the advent of XSLT 3.0, there is an alternative way of

describing a default identity operation in a single line:

```
<xsl:mode on-no-match="shallow-copy"/>
```

The new XSLT 3.0 **<xsl:mode>** element is a *top-level* element, which means that it's a child of the root **<xsl:stylesheet>** element and a sibling of **<xsl:output>** and **<xsl:template>** elements. What this statement does is overwrite the built-in rules, which would otherwise be that 1) if there's no template for an element, throw away the tags and process its children, and 2) if there's no template that matches **text()** nodes, output the text. This new rule says that if there's no template for any node (element, **text()**, anything else), make a shallow copy of it. It will apply first to the document node, because that's where XSLT starts its work, and it will then work its way down the tree. As with the older strategy described above, you can let this XSLT 3.0 identity rule take care of most of your document, and write explicit rules only for the stuff you want to change. Our solution above could thus be rewritten as:

```
<xsl:stylesheet version="3.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:output method="xml" indent="yes"/>
    <xsl:mode on-no-match="shallow-copy"/>
    <xsl:template match="sonnet">
        <xsl:copy>
            <number>
                <xsl:value-of select="@number"/>
            </number>
            <xsl:apply-templates/>
        </xsl:copy>
    </xsl:template>
</xsl:stylesheet>
```

It's better to use this new XSLT 3.0 version in new stylesheets, but because XSLT 3.0 is fairly young, you'll still run into lots of examples of the older strategy. Don't forget, though, that if you use the XSLT 3.0 method, you should set the value of the **@version** attribute on the **<xsl:stylesheet>** root element to **3.0** and you should select Saxon-PE or Saxon-EE as your transformation engine. If you don't do that, you won't raise an error in <oXygen/>, but it's still a mistake; if you're using XSLT 3.0 features, set the value of the **@version** attribute accordingly and use one of the XSLT 3.0 transformation engines.

newtFire {dhlds}
Maintained by: Elisa E. Beshero-Bondar (ebb8 at pitt.edu) (cc) BY-NC-SA Last modified: Tuesday, 17-Oct-2017 15:26:37 EDT. Powered by firebellies.

XSLT Exercise 1

Our very first XSLT assignment is an Identity Transformation, a kind transformation we have to do frequently in our projects when we need to make specific changes to our encoding. We want to make some small changes in our Georg Forster file to make better choices of TEI elements for some of our tags.

To begin, download the Georg Forster file from here: ForsterGeorgComplete.xml and open it in <oXygen>. We don't want to change much about this file, but we do want to alter its tagging just a little, and that is a good occasion to write an Identity Transformation XSLT, converting our XML to XML that is meant to be (for the most part) *identical* to the original.

Here are two changes we want to make to our XML file:

- Looking through the file in the Outline view, we notice that our <head> elements inside each are holding <l> elements, which we originally applied to preserve line breaks in the original document. But we really should not be using the <l> element, because in TEI that element is reserved for a line of poetry! We should change our tagging, and we think we should instead *end* each line with the self-closing <lb/> element used to record a (non-poetry) line-break in TEI.
- Scrolling through the document, we notice we have used <emph> elements in TEI when we wanted to indicate a rendering in italics in the original. Just like the problem with the use of <l>, that was a mistaken application of the TEI (even though it looks perfectly valid), because the <emph> element is only supposed to be used when a writer is placing *strong emphasis* on a word or phrase. In this document, the <emph> elements are being applied to designate non-English words and book titles, so this tagging is not really for emphasis. We really should be using the TEI <hi rend="italic"> tagging for these instead, since this element is designated for highlighting of any kind.

You may already be calculating how to do these tasks with a regular expression Find and Replace, and while we know you could do that, our purpose with this exercise is to make the changes using an XSLT transformation, and we hope you will learn some things about how XSLT works through this exercise!

To begin, open a new XSLT stylesheet in <oXygen> and switch to the XSLT view. We will have some housekeeping to do as we get started.

Namespaces matter! Setting up an XSLT stylesheet to Read TEI

Georg Forster's *A Voyage Round the World* is coded in the TEI namespace, which means that your XSLT stylesheet must include an instruction at the top to specify that when it tries to match elements, it needs to match them in that TEI namespace. When you create a new XSLT document in <oXygen/> it won't contain that instruction by default, so *whenever you are working with TEI* you need to add it (See the text in blue below). We also need to make sure that our XSLT parser understands it is outputting results to the TEI namespace, so we change one more line (See the text in red below).Our modified stylesheet template looks like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xpath-default-namespace="http://www.tei-c.org/ns/1.0"
    xmlns="http://www.tei-c.org/ns/1.0"
    version="3.0">

</xsl:stylesheet>
```

Writing the Identity Transformation!

1. We will give you your first template rule, to set this as an *identity transformation*. We're going to use a new form for this in version XSLT 3.0, so that is why we have set `version="3.0"` in our stylesheet template above. On future assignments we are setting the default version 2.0 for tranforming to HTML mostly because the old version is better tested for processing HTML output, but for an identity transformation of XML to XML, we like the efficient new code we can write in version 3.0. (You can see an old form here in the first template rule of our Identity transformation of Shakespeare's sonnets, which you can download, save and open from here. That old first rule matches on all nodes, elements and attributes throughout the document and simply copies them. It's perfectly fine to use that older template rule in place of the one we show you below, but we like the simplicity of this new form even better!).
   <xsl:mode on-no-match="shallow-copy"/>

   This XSLT statement is the *opposite* of the xsl:template match we have been showing you in our XSLT tutorial. You basically say, if I do not write a template rule to *match* an element, attribute, or comment node, really of any part of the document that I do not mention in a template match rule, XSLT should simply make a copy of that element and output it. Try running this and look at your output: it will look exactly *identical* to the current XML document. Obviously we do not need to do this *unless* we want to make changes with template match rules! There is another way to copy, called "deep copy" in XSLT, but we do not want use it here. When you use "deep copy" in XSLT, you reproduce the full directory tree underneath a given element, so the understanding is that we would match on the root element *only*, and reproduce all the descendents of that one node just as they are. We like the "on-no-match-shallow-copy" approach because we do not necessarily want to copy every node just as it is in the original. We only want to copy if it we do not want to write a new template rule that will change it.

2. Next, we will simply write our template rules to match on the particular elements we wish to change. You may wish to start with the simpler of the two, to convert all the <emph> elements into <hi rend="italics"> in the output XML. Review our Introduction to XSLT to see how to write a template match on any particular element, and how to output as a different element in its place using <xsl:apply-templates/>.
3. Now, write the template rule that will match *only* on <l> elements that are children of <head> elements. And see if you can figure out how to replace these by positioning the self-closing line-break element <lb/>, positioned in the correct spot in relation to <xsl:apply-templates/> so that the <lb/> sits at the end of a line.

4. When we write Identity Transformation XSLTs, we often work with **Attribute Value Templates** (or **AVT**), a handy special format in XSLT that helps us to add attributes to elements like <p> or <l>, and work with values we calculate. This is the tool we use when we want to tell the computer to count and calculate line or paragraph numbers to output in an attribute (like `@n` or `@number`). An AVT offers a special way to extract or calculate information from our input XML to output in an attribute value (for example, this lets us come up with a `count()` of where the particular line we are processing sits in relation to all the `preceding::` line elements ahead of it). You need to look at some examples of AVTs in order to write one yourself, so for this last task, go and look at the examples in Obdurodon's Attribute Value Templates (AVT) tutorial. After reading the AVT tutorial, write **two more template rules** to add @n attributes that automatically number the elements for Books, and the elements for Chapters. (We would ask you to number the paragraphs, too, but we already did that!) **Hint:** For help with teaching the computer how to count these properly, look at my example ID-transform stylesheet that adds line numbers to a series of sonnets, downloadable from here if you didn't download it earlier from the Introduction to XSLT tutorial.) We will return to this later, since you will be working with AVTs in later XSLT exercises and almost certainly in your projects.

When you are finished, save your XSLT file and your XML output of the Georg Forster file, following our usual homework file naming conventions, and upload these to the appropriate place in Courseweb.

<oo>→<dh> Digital humanities

Maintained by: David J. Birnbaum (djbpitt@pitt.edu)
CC BY NC SA
Last modified: 2017-03-01T03:20:17+0000

Modal XSLT

One advantage that XSLT provides over CSS is that while CSS can "decorate the tree," XSLT can rearrange it, fetching nodes from one place in the input tree and writing them somewhere else in the output tree. A subtle side-benefit is that not only can XSLT move a node to a new location, but it can output the same node in multiple locations and treat it differently each time. For example, if your input document has chapters with titles, your XSLT can create output that formats the titles and the chapters as they might appear in a book, but it can also use the chapter titles again, differently, to create a table of contents at the beginning of the output document. It does this by using `<xsl:apply-templates/>` more than once; in the table of contents it applies templates just to the titles, while when it is formatting the body it might apply templates to each chapter, and then, within each chapter, apply templates first to the chapter title and then to the chapter body contents.

Reusing the same input nodes more than once in the output raises the question of how you might treat a node in different ways when you reuse it. For example, when you output the actual chapters you might want to render the chapter titles as HTML `<h3>` elements before the chapter contents, so that they look like large, bold chapter titles. In the table of contents, though, you might want to create a bulleted list with an HTML `<ul>` element, where each chapter title gets created inside the list as an HTML `<li>` element. How do you write template rules that can create an `<h3>` element when needed for a chapter title in the body, and that can also create a `<li>` element when needed for the same chapter title in a table of contents at the front of the output document?

There are several possible solutions to the question of how to reuse parts of the input tree to output them differently in different locations in the output tree, but the one that is often most convenient is the XSLT `@mode` attribute.

XSLT modes

XSLT has a `@mode` attribute that can appear on `<xsl:template>` and `<xsl:apply-templates>` elements. The `@mode` attribute can be used to create a separate set of rules for processing titles in the table of contents that can operate alongside the regular template rules that you might use to output the main text. For example, suppose your input document is something like:

```
<report>
    <chapter>
        <title>This is the title of the first chapter</title>
        <paragraph>This paragraph is the content of the first chapter. In
        real life a chapter would probably have a lot of paragraphs.</paragraph>
    </chapter>
    <chapter>
        <title>This is the title of the second chapter</title>
        <paragraph>This paragraph is the content of the second chapter.</paragraph>
    </chapter>
    <chapter>
        <title>This is the title of the third chapter</title>
        <paragraph>This paragraph is the content of the third chapter.</paragraph>
    </chapter>
    <chapter>
        <title>This is the title of the fourth chapter</title>
```

```
        <paragraph>This paragraph is the content of the fourth chapter.</paragraph>
    </chapter>
</report>
```

You already know how to generate HTML output from this input:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="2.0">
    <xsl:template match="/">
        <html>
            <head>
                <title>My output</title>
            </head>
            <body>
                <xsl:apply-templates select="//chapter"/>
            </body>
        </html>
    </xsl:template>
    <xsl:template match="chapter">
        <xsl:apply-templates/>
    </xsl:template>
    <xsl:template match="title">
        <h3>
            <xsl:apply-templates/>
        </h3>
    </xsl:template>
    <xsl:template match="paragraph">
        <p>
            <xsl:apply-templates/>
        </p>
    </xsl:template>
</xsl:stylesheet>
```

The **<xsl:template>** and **<xsl:apply-templates>** elements in this stylesheet have no **@mode** attribute. You can now augment the stylesheet by adding additional rules that specify a **@mode** attribute; you can name the mode anything you want, and I normally use “toc” for tables of contents.

The rules you add to your stylesheet are highlighted below:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="2.0">
    <xsl:template match="/">
        <html>
            <head>
                <title>My output</title>
            </head>
            <body>
                <h2>Contents</h2>
                <ul>
                    <xsl:apply-templates select="//title" mode="toc"/>
                </ul>
                <hr/>
                <xsl:apply-templates select="//chapter"/>
            </body>
        </html>
    </xsl:template>
    <xsl:template match="chapter">
        <xsl:apply-templates/>
    </xsl:template>
    <xsl:template match="title">
        <h2>
```

```
            <xsl:apply-templates/>
        </h2>
    </xsl:template>
    <xsl:template match="paragraph">
        <p>
            <xsl:apply-templates/>
        </p>
    </xsl:template>
    <xsl:template match="title" mode="toc">
        <li>
            <xsl:apply-templates/>
        </li>
    </xsl:template>
</xsl:stylesheet>
```

Where we want to generate the table of contents (in this case, before the actual contents), we create the `<ul>` container that will hold the list of chapter titles. Inside it we apply templates to all of the titles as a way of rounding them up for processing, but when we apply templates, we specify `@mode="toc"`. The presence of the `@mode` attribute on the `<xsl:apply-templates>` element tells the system to ignore any template rule for `<title>` nodes that does not specify the same value for the `@mode` attribute. The system therefore ignores the regular (modeless) rule that we wrote for titles, and chooses instead the new one (with `mode="toc"`), which specifies that titles should be wrapped `<li>` tags.

Meanwhile, when we later process titles a second time, to render each title as an HTML `<h2>` before the chapter contents in the body of the output, the old rule applies. The new, modal rules don't affect the logic of the original stylesheet; our old, modeless rules continue to work as before, but they are now augmented by modal rules that let us also generate a table of contents that treats titles differently from the way they're treated in the main output.

You can have as many modes as you need in a stylesheet and you can call them anything you want. Note also that you can call a template that is in one mode from inside a template that is in a different mode. In the example above, the template rule for titles that is in the "toc" mode applies templates to the contents of each title, that is, to the `text()` nodes that contain the actual title text, and it doesn't specify a mode. We can use the regular (modeless and built-in) template rule in this case because we don't need to process the text any differently than we do in the body of the output document. What's important is that there is no prohibition against calling a modeless template from inside a modal one or vice versa; whether you specify a mode depends only on the processing you need at that point in the transformation.

The most common mistake people make with modal XSLT is forgetting to specify the mode value when needed. Think of XSLT as involving throwing and catching, where `<xsl:apply-templates>` throws nodes out into the ether and template rules sit around waiting to catch them as they come by. If you have a rule like `<xsl:apply-templates select="//title" mode="toc"/>`, it throws the `<title>` elements out, but it specifies that they can be caught only by a template rule that not only matches the correct gi ("gi" is the standard abbreviation for "generic identifier," which is XML-speak for what humans would call an "element name") with `match="title"`, but also invokes the correct mode with `mode="toc"`.

<oo>→<dh> Digital humanities

Maintained by: David J. Birnbaum (djbpitt@gmail.com)
Last modified: 2018-03-01T22:49:03+0000

XSLT, part 2: Advanced features

This supplementary XSLT tutorial concentrates on four topics: variables, keys, conditionals (`<xsl:if>` and `<xsl:choose>`), and the difference between push processing (`<xsl:apply-templates>` and `<xsl:template>`) and pull processing (`<xsl:for-each>` and `<xsl:value-of>`).

The `<xsl:variable>` element

If you've used variables in other programming languages before, be aware that variables in XSLT do not work like variables in most other languages. *The value of a variable in XSLT cannot be updated once the variable has been declared.*

The `<xsl:variable>` element requires a `@name` attribute, which names the variable for future use. The value of the variable is typically assigned through a `@select` attribute (in which case `<xsl:variable>` is an empty element), but it can also be specified as the content of the `<xsl:variable>` (in which case the element cannot have a `@select` attribute). It's usually easier to use `@select`, since this typically produces less complicated code, but if you need to do anything particularly involved, you may not be able to use the `@select` method of assigning your value. To reference a variable later on, just use `$variableName`, where `variableName` is the value of the `@name` attribute you wrote when creating the variable. That is, when you declare a variable to, say, count the paragraphs (`<p>` elements) in your input, you might give it the name "paragraphCount" with something like:

```
<xsl:variable name="pargraphCount" select="count(//p)"/>
```

Note that there is no leading dollar sign associated with the name when you declare and define a variable. But should you later refer to the variable, you need the leading dollar sign. For example, to get the value of the variable, you might use:

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

The `<xsl:variable>` element may be defined in different locations in the stylesheet, and the location makes a difference. When you define a variable (that is, use `<xsl:variable>`) as an immediate child of the root `<xsl:stylesheet>` element, the variable can be used anywhere in the stylesheet. When, on the other hand, you define a variable inside a template rule, it's available only within that template rule.

We often use a top-level `<xsl:variable>` element to avoid having to recalculate a value that is used often, as well as to access the tree from an atomized context (such as when you've used `<xsl:for-each/>`, which we'll explain when the situation arises). You might also use variables to avoid typing a long XPath expression within some other complicated instruction. Variables that are not strictly necessary and that are created for the convenience of the developer are called, not surprisingly, *convenience variables*.

For more information about variables, see Michael Kay, 500 ff.

The `<xsl:key>` element

`<xsl:key>` may be overlooked in situations where comparable functionality is available through other means, but it is often simpler (and almost always faster) to use `<xsl:key>` than the alternatives (we once reduced the run time for a transformation from twenty minutes to just a few seconds by switching to an implementation that used `<xsl:key>`!). The `<xsl:key>` element requires three attributes. Consider, for example, XML structured like:

```
<book>
    <title>XSLT 2.0 and XPath 2.0 Programmer's Reference</title>
    <author>Michael Kay</author>
    <publisher>Wrox</publisher>
    <edition>4</edition>
```

```
    <year>2008</year>
</book>
```

where you want to be able to find books by their authors. You could define a key as:

```
<xsl:key name="bookByAuthor" match="book" use="author">
```

The three required attributes in this case are:

- The `@name` attribute is the name you'll use when referring to the key when you need to use it to look up and retrieve a value. You can make up your own value here, but the syntax we've used is a common strategy for reminding the human operator of what the key retrieves and how it finds what it's looking for.
- The `@match` attribute is an XPath pattern—like the `@match` on `<xsl:template>`. Here we say that we want to find `<book>` elements. Note that because this is an XPath pattern, and not a full XPath expressions, all we need is the name of the element. This is similar to specifying an XSLT template rule that is supposed to process `<book>` elements no matter where they appear, which would have a `@match` attribute value of just "book". In both cases, the key and the template don't need (= should not be given) a full path to the elements to which they apply; they just need enough information to know what to match, and in this case the bare element name will suffice.
- The `@use` attribute is an XPath expression starting at the value of the `@match` attribute. In this example, we are saying that we want to find `<book>` elements by using their child `<author>` elements. That is, the XPath expression that constitutes the value of the `@use` attribute takes the value of the `@match` attribute as its starting context, so if the value of `@match` is a `<book>` element, a `@use` value of "author" means to look at any `<author>` elements that are on the child axis of `<book>` elements.

The `@match` attribute value of the key is the object (typically an element) that the processor will return when the key is referenced (see below), while the `@use` attribute value tells the processor what to use to look up those values. In the example above, you would be able to use the key to retrieve `<book>` elements according to their `<author>` child elements. To retrieve information with the help of a key, you use the `key()` XPath function, which takes two or three arguments. The first argument is the name of the key (matching the `@name` value from the `<xsl:key>` element), and it must be in quotation marks (single or double). The second argument is the value to look up; for example, in the sample above, if you were to specify "Michael Kay" as the second argument to the `key()` function (`key("bookByAuthor","Michael Kay")`), you would retrieve all `<book>` elements with `<author>` children that have the value "Michael Kay". The (optional) third argument is the document root of the document in which to look. When the third argument is omitted, the function searches in the current document. For further discussion of `<xsl:key>`, consult Michael Kay, page 376.

Conditionals

`<xsl:if>`

`<xsl:if>` is useful when you have one particular feature whose value may sometimes require special treatment. For example, you might use `<xsl:if>` to color all `<speaker>` elements with the `@who` value "Hamlet" differently from all other `<speaker>` elements. `<xsl:if>` takes a required attribute `@test`, which takes a *Boolean* argument (that is, the attribute value has to describe a test that evaluates to either "True" or "False") just like a predicate expression in XPath. The contents of the `<xsl:if>` element, then, describe what the system is to do if the result of `@test` is True: for example, you might want to apply templates or use `<xsl:value-of>` to display the results of a particular function, or you might want to create a special `@class` attribute value (if you are generating HTML) using `<xsl:attribute>` that can be styled with CSS (see our Using `<span>` and `@class` to style your HTML to refresh your memory about the `@class` attribute). Consider:

```
<xsl:template match="sp">
    <p>
        <xsl:if test="speaker='Hamlet'">
            <xsl:attribute name="class">mainCharacter</xsl:attribute>
        </xsl:if>
        ...
    </p>
</xsl:template>
```

In this example we are checking each `<sp>` (because we're doing this inside the template rule for `<sp>` elements) to see whether its child `<speaker>` (remember that we default to the child axis) is equal to the string "Hamlet". If the result of this test is True, we'll go on to perform whatever is inside `<xsl:if>`. If it isn't, we'll throw it away and won't do anything special with it. In this case, everywhere this test is True we'll create an attribute using `<xsl:attribute>`, and we use the `@name` attribute to specify

what name this attribute should have: in this case we're creating the attribute `@class`. This attribute gets attached to the parent element: in this case, `<p>`. The contents of `<xsl:attribute>` indicate the value to be assigned to this new attribute: in this case the value of `@style` will be "mainCharacter". This means that anywhere there's a speech by Hamlet, we're mapping it to something like:

```
<p class="mainCharacter"> . . . </p>
```

If we then have a rule in our CSS like:

```
.mainCharacter { color: red; }
```

any `<p>` element that contains a speech by Hamlet will have this attribute and will now be colored red.

`<xsl:choose>`

Although `<xsl:if>` can be useful, sometimes we need to code for multiple possible environments, or we care about what should happen when the results of our conditional are False, and this is where `<xsl:choose>` comes in. `<xsl:if>` can run only one test and can have only two results: True or False. On the other hand, you can use `<xsl:choose>` to specify a number of different conditional environments, as well as a fallback action if none of the conditions is true. `<xsl:choose>` takes at least one child `<xsl:when>` element (and up to as many as you want) and one optional `<xsl:otherwise>` element. `<xsl:when>` requires the same `@test` attribute that we discussed above. Since `<xsl:otherwise>` is the fallback condition, it doesn't take this `@test` attribute; it only applies when all `<xsl:when>` tests return False.

```
<xsl:template match="sp">
    <p>
        <xsl:choose>
            <xsl:when test="speaker='Hamlet'">
                <xsl:text>[Hi, Hamlet!] </xsl:text>
            </xsl:when>
            <xsl:when test="speaker='Ophelia'">
                <xsl:text>[Hi, Ophelia!] </xsl:text>
            </xsl:when>
            <xsl:otherwise>
                <xsl:text>[Neither Hamlet nor Ophelia] </xsl:text>
            </xsl:otherwise>
        </xsl:choose>
        <xsl:apply-templates/>
    </p>
</xsl:template>
```

In this example, we have two tests (`<xsl:when>`) and one fallback (`<xsl:otherwise>`), which is used if neither test returns True. The first test checks whether the child `<speaker>` element (remember that we're in the template rule for `<sp>`) is equal to "Hamlet". If it is, we use the `<xsl:text>` element to create a `text()` node with the content "[Hi, Hamlet!] ", which means that we return the plain text: "[Hi, Hamlet!] ". The second test works along the same lines, except that it checks whether the child `<speaker>` is equal to "Ophelia". If *this* test is True, then we return plain text reading "[Hi, Ophelia!] ". If neither of these tests returns True (that is, if the speaker is anyone other than Hamlet or Ophelia), then the `<xsl:otherwise>` condition kicks in. In this case, that means that we return the plain text "[Neither Hamlet nor Ophelia] ". Note that we've put in a space at the end of each of these strings of plain text, because we apply templates at the end of this block of conditionals in order to output the speech, and we want a space before it. If you run this code, your output should look like this (we've added bolding to the speaker names to make them easier to see here):

[Neither Hamlet nor Ophelia] **Osric:** It is indifferent cold, my lord, indeed.

[Hi, Hamlet!] **Hamlet:** But yet methinks it is very sultry and hot for my complexion.

The examples above of `<xsl:if>` and `<xsl:choose>` came from the following stylesheet, which is included in its entirety for your reference. It outputs all of the speeches in Bad Hamlet normally, but we have the system do some extra formatting depending

on whether the speaker is Hamlet, Ophelia, or anyone else. If you use it to transform the play, you'll see how the formatting works, and how new content is created before each speech.

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="3.0"
    xpath-default-namespace="http://www.tei-c.org/ns/1.0"
    xmlns="http://www.w3.org/1999/xhtml">
    <xsl:output method="xml" indent="yes" doctype-system="about:legacy-compat"/>
    <xsl:template match="/">
        <html>
            <head>
                <title>XSLT conditional practice</title>
            </head>
            <body>
                <h1>XSLT conditional practice</h1>
                <xsl:apply-templates select="//sp"/>
            </body>
        </html>
    </xsl:template>
    <xsl:template match="sp">
        <p>
            <xsl:if test="speaker='Hamlet'">
                <xsl:attribute name="class">mainCharacter</xsl:attribute>
            </xsl:if>
            <xsl:choose>
                <xsl:when test="speaker='Hamlet'">
                    <xsl:text>[Hi, Hamlet!] </xsl:text>
                </xsl:when>
                <xsl:when test="speaker='Ophelia'">
                    <xsl:text>[Hi, Ophelia!] </xsl:text>
                </xsl:when>
                <xsl:otherwise>
                    <xsl:text>[Neither Hamlet nor Ophelia] </xsl:text>
                </xsl:otherwise>
            </xsl:choose>
            <xsl:apply-templates/>
        </p>
    </xsl:template>
    <xsl:template match="speaker">
        <strong>
            <xsl:apply-templates/>
            <xsl:text>: </xsl:text>
        </strong>
    </xsl:template>
    <xsl:template match="l | ab">
        <xsl:apply-templates/>
        <xsl:if test="following-sibling::l or following-sibling::ab">
            <br/>
        </xsl:if>
    </xsl:template>
</xsl:stylesheet>
```

Push and pull design

The XSLT processing model supports both *push* and *pull* design. The push model, which is what we've been using exclusively so far, relies on `<xsl:apply-templates>` to identify what is supposed to get processed where, and on `<xsl:template>` to describe how it is supposed to be processed. This is called "push" because you *push* the elements and other components out into the stylesheet and rely on the templates to grab the individual pieces and process them. For example, you don't say "take all the paragraphs and paint them blue"; what you do instead is say in one place "here are some paragraphs; take care of them" and in another "whenever you happen to run into a paragraph, paint it blue." The great strength of push processing is that you don't have to know the structure of your input document—that is, you don't have to know which elements will be encountered where. The declarative template rules ensure that no matter where an element pops up, you'll have a template around that will know what to do with it. Since the structure of humanities documents involves a lot of variable mixed content, this declarative approach creates a flexibility that is difficult to achieve with the sort of procedural programming that requires you to know at each moment exactly what is supposed to happen next.

The pull model, on the other hand, is procedural in nature, and relies primarily on `<xsl:for-each>` and `<xsl:value-of>`. It is useful when you need to round up specific information, instead of dealing with it on the fly whenever it happens to come up. Pull design is useful for generating tables, for example, where you might want to create a row for each character in a play with columns for the name and the number of speeches (see the example below). In this case you don't want to process each speech where it

occurs; you want to go out and grab them all for one character, and then for the next, etc. The pull model would work poorly, on the other hand, for rendering each speech as it occurs, since it might contain an unpredictable variety of in-line elements, and you need to be able to deal with those as they arise, without having to know in advance which ones to call for explicitly.

About pull

Pull design is frequently overused by beginning XSLT programmers, especially if they have experience with procedural programming languages. In many cases the end result of using pull will be the same as the result of using push, but pull design is often harder to maintain because it is less consistent with the declarative nature of XSLT as a programming language. With that said, pull design does have its uses. As noted above, the two principal elements used in pull coding are `<xsl:for-each>` and `<xsl:value-of>`.

`<xsl:for-each>`

The `<xsl:for-each>` element is used to iterate over a sequence of items (most often elements, but other items, including string or numerical values, are also permissible). `<xsl:for-each>` requires one attribute, `@select`, the value of which can be a full XPath expression (just like the value of the `@select` attribute with `<xsl:apply-templates>`). Whatever `@select` identifies becomes the sequence of current context items, so any XPath expressions used in children of `<xsl:for-each>` begin at the current context node, not at the document node.

We often use `<xsl:for-each>` with scalable vector graphics (SVG), which we'll be introducing later in the semester. It is also useful for creating a sorted list when used in conjunction with `<xsl:sort>` (see Michael Kay for details).

`<xsl:value-of>`

Although the results of `<xsl:value-of>` and `<xsl:apply-templates>` are often the same, the real usefulness of `<xsl:value-of>` is that it allows you to output the results of functions and non-node values. For example, if you want to output a list of unique speakers in a play, the following code will generate an error message:

```
<xsl:for-each select="distinct-values(//speaker)">
    <xsl:apply-templates/>
</xsl:for-each>
```

The problem is that you can't apply templates to an "atomic value" (Michael Kay: "an item such as an integer, a string, a date, or a boolean", rather than a node [element, attribute, and a few others]). What you should do instead is:

```
<xsl:for-each select="distinct-values(//speaker)">
    <xsl:value-of select="."/>
</xsl:for-each>
```

If you want to do something to every *instance* of a `<speaker>` *element* in a play, though, repeats and all, you should prefer `<xsl:apply-templates>`. The difference is that each instance of a `<speaker>` element is a node in the tree, but the sequence produced by applying the `distinct-values()` function to all of the `<speaker>` nodes is a sequence of atomic values, and not of nodes.

The following example creates an HTML page that lists the number of speeches by each speaker in Bad Hamlet:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="3.0"
    xpath-default-namespace="http://www.tei-c.org/ns/1.0"
    xmlns="http://www.w3.org/1999/xhtml">
    <xsl:output method="xml" indent="yes" doctype-system="about:legacy-compat"/>
    <xsl:template match="/">
        <html>
            <head>
                <title>Bad Hamlet Speeches</title>
            </head>
            <body>
                <xsl:for-each select="//role">
                    <p>
                        <xsl:value-of select="."/>
                        <xsl:text>: </xsl:text>
                        <xsl:value-of select="count(//sp[contains(@who, current()/@xml:id)])"/>
```

```
                    </p>
                </xsl:for-each>
            </body>
        </html>
    </xsl:template>
</xsl:stylesheet>
```

What's happening here is that we loop through each `<role>` element in the whole of *Bad Hamlet* (at any depth, as specified by the `//`) and create a `<p>` element:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="2.0"
    xpath-default-namespace="http://www.tei-c.org/ns/1.0"
    xmlns="http://www.w3.org/1999/xhtml">
    <xsl:output method="xml" indent="yes" doctype-system="about:legacy-compat"/>
    <xsl:template match="/">
        <html>
            <head>
                <title>Bad Hamlet Speeches</title>
            </head>
            <body>
                <xsl:for-each select="//role">
                    <p>
                        <xsl:value-of select="."/>
                        <xsl:text>: </xsl:text>
                        <xsl:value-of select="count(//sp[contains(@who, current()/@xml:id)])"/>
                    </p>
                </xsl:for-each>
            </body>
        </html>
    </xsl:template>
</xsl:stylesheet>
```

Inside each `<p>`, return the *value of* the context node (specified by the `.`):

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="2.0"
    xpath-default-namespace="http://www.tei-c.org/ns/1.0"
    xmlns="http://www.w3.org/1999/xhtml">
    <xsl:output method="xml" indent="yes" doctype-system="about:legacy-compat"/>
    <xsl:template match="/">
        <html>
            <head>
                <title>Bad Hamlet Speeches</title>
            </head>
            <body>
                <xsl:for-each select="//role">
                    <p>
                        <xsl:value-of select="."/>
                        <xsl:text>: </xsl:text>
                        <xsl:value-of select="count(//sp[contains(@who, current()/@xml:id)])"/>
                    </p>
                </xsl:for-each>
            </body>
        </html>
    </xsl:template>
</xsl:stylesheet>
```

We then output a colon followed by a space, just as plain text:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="2.0"
    xpath-default-namespace="http://www.tei-c.org/ns/1.0"
    xmlns="http://www.w3.org/1999/xhtml">
    <xsl:output method="xml" indent="yes" doctype-system="about:legacy-compat"/>
    <xsl:template match="/">
        <html>
            <head>
                <title>Bad Hamlet Speeches</title>
            </head>
            <body>
```

```
                <xsl:for-each select="//role">
                    <p>
                        <xsl:value-of select="."/>
                        <xsl:text>: </xsl:text>
                        <xsl:value-of select="count(//sp[contains(@who, current()/@xml:id)])"/>
                    </p>
                </xsl:for-each>
            </body>
        </html>
    </xsl:template>
</xsl:stylesheet>
```

Finally we return a count of all the `<sp>` elements that meet a certain condition (the predicate):

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="2.0"
    xpath-default-namespace="http://www.tei-c.org/ns/1.0"
    xmlns="http://www.w3.org/1999/xhtml">
    <xsl:output method="xml" indent="yes" doctype-system="about:legacy-compat"/>
    <xsl:template match="/">
        <html>
            <head>
                <title>Bad Hamlet Speeches</title>
            </head>
            <body>
                <xsl:for-each select="//role">
                    <p>
                        <xsl:value-of select="."/>
                        <xsl:text>: </xsl:text>
                        <xsl:value-of select="count(//sp[contains(@who, current()/@xml:id)])"/>
                    </p>
                </xsl:for-each>
            </body>
        </html>
    </xsl:template>
</xsl:stylesheet>
```

The predicate here says "get all the `<sp>` elements in the entire play and check to see whether their `@who` attributes contain some substring." When we're executing an `<xsl:for-each>` loop, the value of each item in the loop can be represented by `current()`. This means that we're comparing the value of the `@who` attribute of each `<sp>` element to the `@xml:id` attribute of the `<role>` element that we're processing at the moment. For example, when we process `<role xml:id="Hamlet">Hamlet</role>`, we check the `@who` attribute of every `<sp>` in the play to see whether it contains, as a substring, the value of the `@xml:id` attribute of that `<role>`. If it does, that's a speech by Hamlet, so it gets included in our count. After we've gone through every `<sp>` and checked who the speaker is, we output the count of the speeches for the `<role>` we're looking at at the moment. Then we move on to the next `<role>`. When we run out of roles, the `<xsl:for-each>` terminates gracefully. (Our use of the XPath `contains()` function is brittle, since it could strike a false positive if, say, there were characters named both "Ham" and "Hamlet". In that case we would erroneously identify Hamlet's speeches as by both Ham and Hamlet. If false positive substring matches are a risk with your data, more robust methods are available.)

This is the first time you've seen the function `current()`, and you may be wondering why you can't write:

```
count(//sp[contains(@who, ./@xml:id)])
```

(with a dot instead of `current()`). The problem is that the dot refers to wherever you are at that moment in your current XPath expression. Since you're inside a predicate that is being applied to a preceding `<sp>`, a dot would check the `@xml:id` attribute of the `<sp>`, and not of the `<role>`. Since the `<sp>` doesn't have an `@xml:id` attribute (it's the `<role>` that does), this wouldn't find the matches we care about. That is:

- Inside a `<xsl:for-each>` loop, `current()` refers to the current item in that loop. In the case of `<xsl:for-each select="//role">`, `current()` will refer to each `<role>` in turn.
- The dot always refers to the current context in an XPath expression. Inside a predicate, the dot represents the item to which the predicate is being applied. For example, in

  ```
  //sp[contains(@who, ./@xml:id)]
  ```

we would be testing whether each `<sp>` has a `@who` attribute that contains the value of the `@xml:id` attribute of that same `<sp>`. As noted above, this is wrong; we want to look at the `@xml:id` attribute of the `<role>` elements, and not of the `<sp>` ones.

You may find the following distinction helpful: From a technical perspective, `current()` refers to the current context *at the XSLT level* and the dot refers to the current context in *an XPath path expression*. At the first step of an XPath path expression, the two mean the same thing. In the example above, when we output `<xsl:value-of select="."/>` we could instead have said `<xsl:value-of select="current()"/>`, since in this simple path the XSLT and XPath contexts are the same. We don't have that choice in `count(//sp[contains(@who, current()/@xml:id)])`, though; here the more complicated XPath includes a new step, `//sp`, which changes the XPath context. Here we need to use `current()` because the XSLT context was set at the `<xsl:for-each>` stage, and is unaffected by the comparison. For more discussion, with examples, see Michael Kay, p. 735.

If you try to run this code, you'll notice that it takes a bit longer than usual to finish. That's because it's looping through the entire play repeatedly, looking at every speech once for every role in the play. At 1137 `<sp>` elements and 37 `<role>` elements, that's 42069 comparisons. This is part of why we usually avoid using `<xsl:for-each>` unless the problem really calls for it, and in those cases there are ways to speed it up (such as by using a key, as described above).

There are situations that can be managed with either push or pull strategies. In most of those cases, your instinct, unless you are a veteran XSLT programmer, will draw you toward pull. It's much more common in humanities-oriented XSLT to use push programming, and where there's a choice, we'd encourage you to train yourselves to think of push first, and fall back on pull only where it is truly more appropriate.

<oo>→<dh> Digital humanities

Maintained by: David J. Birnbaum (djbpitt@gmail.com)
Last modified: 2016-08-28T20:35:34+0000

Using <xsl:analyze-string>

The `<xsl:analyze-string>` element uses regular expressions to parse a string of text and identify substrings that match a particular regex pattern. Kay writes: "It is useful where the source document contains text whose structure is not fully marked up using XML elements and attributes."

Consider the following XHTML document (adapted from a page that no longer exists, but that we found a few years ago at http://ies.sas.ac.uk/cmps/Projects/OUP/index.htm):

```
<html xmlns="http://www.w3.org/1999/xhtml">
   <head>
       <title>The History of Oxford University Press</title>
       <!-- from http://ies.sas.ac.uk/cmps/Projects/OUP/index.htm -->
   </head>
   <body>
       <p>The History of Oxford University Press</p>
       <p>This major national and international scholarly project, which will be inaugurated on 1
            January 2006, is a co-operative venture between Oxford University Press and the
            Institute of English Studies. Its General Editor is Professor Simon Eliot who holds the
            newly-created chair in the History of the Book in the Institute.</p>
       <p>The History will consist of four volumes which will cover the following periods:</p>
       <div>
           <ul>
               <li>Volume I 1478-1780s</li>
               <li>Volume II 1780s-1890s</li>
               <li>Volume III 1890s-1960s</li>
               <li>Volume IV 1960s-2000</li>
           </ul>
       </div>
       <p>Each volume will be edited by a distinguished scholar in the field and will consist of
            chapters written by that scholar and specialists in book history, social and economic
            history, history of scholarship and the history of science and technology.</p>
       <p>Oxford University Press will be funding the equivalent of six years of postdoctoral
            fellowships in order to provide the fundamental research on which the History will be
            based. These fellowships will most likely be divided up in the following way:</p>
       <div>
           <ol>
               <li>A three-year postdoctoral fellowship on the economic and business history of
                    the Press.</li>
               <li>A one-year postdoctoral fellowship on the impact of technological and
                    communications revolutions on the Press.</li>
               <li>A one-year postdoctoral fellowship on the origins and development of OUP's
                    branches in the USA and Canada.</li>
               <li>A one-year postdoctoral fellowship on the origins and development of OUP's
                    branches in South East Asia.</li>
           </ol>
       </div>
       <p>It is intended that appointments will be made to these fellowships in 2006 and 2007.</p>
       <p>In addition, a major Book History research seminar series focusing, though not
            exclusively so, on the History will be established. It is hoped that this will involve
            members of the History and English faculties at Oxford, and members of the Institute of
            Historical Research and the Institute of English Studies in the School of Advanced Study
            in the University of London. Monthly meetings will be held alternately in Oxford and
            London and will be open to all.</p>
       <p>Updates and progress reports on this ambitious and exciting project will be posted on the
            Institute web site from time to time.</p>
   </body>
</html>
```

This document contains years, which are four-digit numbers, but they haven't been tagged as years. If, for example, we want to make the years clickable links that will take us to a place where we can look up what happened in that year, we'll need to insert the markup. This is the sort of not-fully-marked-up text that Kay had in mind, and we can add the markup we want by using a modified identity transformation and `<xsl:analyze-string>`. For the purpose of this exercise, we're going to use a resource at http://www.historyorb.com/dates-by-year.php that allows us to look up whatever happened in a particular year by going to, for example, http://www.historyorb.com/events/date/1960 (replacing the "1960" in the example with whatever year we care about). What we want, then, is for each year in the input document to create a link in the output document that will let us click on the year and look it up at this site. To simplify our task, we'll cut a few corners: we'll treat

every year reference as a single year (for example, when the text says "1960s" we'll just look up 1960), we won't check for missing years (which means that we might get an error message should we happen to look up a year that isn't represented at http://www.historyorb.com because nothing of interest happened then), and we'll assume that all four-digit numbers are years and all years are later than the year 999, that is, that all years are four-digit years. In Real Life we'd have to evaluate whether those were sensible assumptions given our data, and if not, we'd have to decide how to cope.

Here's our stylesheet (discussion follows):

```
<xsl:stylesheet xmlns="http://www.w3.org/1999/xhtml"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xpath-default-namespace="http://www.w3.org/1999/xhtml" version="2.0">
    <xsl:template match="*">
        <xsl:copy>
            <xsl:apply-templates/>
        </xsl:copy>
    </xsl:template>
    <xsl:template match="text()">
        <xsl:analyze-string select="." regex="\d{{4}}">
            <xsl:matching-substring>
                <a href="http://www.historyorb.com/events/date/{.}">
                    <xsl:value-of select="."/>
                </a>
            </xsl:matching-substring>
            <xsl:non-matching-substring>
                <xsl:value-of select="."/>
            </xsl:non-matching-substring>
        </xsl:analyze-string>
    </xsl:template>
</xsl:stylesheet>
```

We begin with the identity transformation, and because our input document has no attributes, we're using a simplified template that doesn't need to match attributes (in the `@match` attribute of the `<xsl:template>` element) or process them (in the `@select` attribute of the `<xsl:apply-templates>` element). Because we're writing a separate rule for `text()` nodes (since we need to parse them to look for dates), our basic identity template has to match only elements, which we can do with an asterisk, which means "any element."

Our other template rule processes `text()` nodes. When we match a `text()` node, we invoke the `<xsl:analyze-string>` element, passing it something to parse (the `text()` node we just matched, represented by the dot, since it's the current context node) and a regular expression that will be used to parse it. The regular expression in this case, `regex="\d{{4}}"`, is designed to match any four-digit number. Let's look at the pieces:

- The `\d` matches any single digit.
- In addition to the general repetition indicators that we've been using since we learned about Relax NG (question mark, asterisk, plus sign), regex syntax lets us specify an exact number of repetitions or a range of repetitions. To specify an exact number of repetitions, we follow whatever we're counting with an integer in curly braces, so `\d{4}` would match exactly four digits. (There are also ways to specify just a minimum number of matches, just a maximum, or both.)
- Regex syntax requires that the number of matches appear in curly braces, but in the `<xsl:analyze-string>` context, the regex attribute is an attribute value template (AVT), which means that anything that appears in curly braces would be interpreted not as part of regex syntax, but as an XPath expression. We need, therefore, to *escape* the curly braces by doubling them; this tells the AVT analyzer that it should pass real curly braces to the regex analyzer, which then recognizes them as saying "match a digit exactly four times in sequence," that is, match a four-digit number.

The `<xsl:analyze-string>` element normally takes two child elements, `<xsl:matching-substring>` and `<xsl:non-matching-substring>`. As the element name implies, once the `<xsl:analyze-string>` parser breaks the string into parts that match the regex and parts that don't, one or the other of these subelements handles each of those parts. In our stylesheet, for a non-matching substring (any string of text inside the `text()` node that isn't a four-digit number), we just output the value of that substring using the `<xsl:value-of>` element. For any substring that matches, we create an HTML link (`<a>`) and insert the appropriate value for the `@href` attribute, using an AVT to plug in the four-digit number that we matched. Note that within `<xsl:matching-substring>` and `<xsl:non-matching-substring>`, a dot refers to the specific substring, and not to the entire `text()` node.

Here's a snippet of the output:

```
<p>This major national and international scholarly project, which will be inaugurated on 1 January
<a href="http://www.historyorb.com/events/date/2006">2006</a>, is a co-operative venture between
Oxford University Press and the Institute of English Studies. Its General Editor is Professor Simon Eliot
who holds the newly-created chair in the History of the Book in the Institute.</p>
```

Note that the four-digit year is marked up as a link to the http://www.historyorb.com site, but the one-digit part of "1 January 2006" is not. Similarly:

```
<li>Volume I <a href="http://www.historyorb.com/events/date/1478">1478</a>-
<a href="http://www.historyorb.com/events/date/1780">1780</a>s </li>
```

Here the four-digit years are tagged even when they are part of an expression like “1780s”.

newtFire {dhlds}
Authored by: Rob Spadafore (spadafour at gmail.com) Edited and maintained by: Elisa E. Beshero-Bondar (ebb8 at pitt.edu)
(cc) BY-NC-SA Last modified: Sunday, 06-Aug-2017 18:15:26 EDT. Powered by firebellies.

Guide to Schema Writing with Schematron

As we learned in the Relax NG tutorial, we write and associate schema to constrain the content of an XML document. This helps if you are working with many complex files or trying to coordinate a team of coders to maintain consistency across an entire project. Relax NG is a *grammar-based schema language*, which means that it defines the hierarchical relationship of elements and attributes in an entire document from its starting root to all its branches. It may seem like Relax NG ought to be able to govern everything we need, but there are certain kinds of constraints that it can't handle. For these we apply a *rule-based schema* to function alongside our grammar-based schema in order to fine-tune precise relationships among elements and attributes. We work with Schematron, a rule-based constraint language that uses XPath expressions to *assert* or *report* on the presence or absence of patterns. Rule-based schema languages like Schematron typically do not constrain every element and attribute like our Relax NG Schemas. Instead, when we write Schematron, we usually concentrate on just a few things that we need to control very precisely, as we will show you here.

Relax NG and **Schematron** are commonly used together. For example, let's say we are collecting data from 100 people and want to record their votes for their favorite ice cream flavor: vanilla, chocolate, or strawberry. Limiting our attributes to those three flavors and defining the responses as integers would not be difficult using Relax NG. But what if, instead of 31 votes for chocolate, I accidentally entered 131 votes? A basic Relax NG schema that defines the element vote this way `vote = element vote {type, xsd:integer}` and `type = attribute type {"chocolate" | "vanilla" | "strawberry" }` wouldn't catch any problems with the specific numbers I enter, because the data type for integer is not something we can set to specific numerical values in relation to a total. If we want to make sure that the numerical values of all `<vote>` elements add up to 100, Schematron is the tool we need. More generally, we use Schematron if we need to define rules that assert relationships in the content of our elements and attributes, such as (among other things) to make sure that the preceding-sibling::header does not contain the identical text of a following-sibling header, to check that elements holding page number values appear in the correct order, or to flag every time we are missing a punctuation mark that is supposed to appear inside a sentence element.

Superstructure and namespaces (the stuff at the top of the document)

When you open a new **Schematron** file in <oXygen/>, you will see the following **superstructure**:

```
<?xml version="1.0" encoding="UTF-8"?>
        <sch:schema xmlns:sch="http://purl.oclc.org/dsdl/schematron" queryBinding="xslt2"
        xmlns:sqf="http://www.schematron-quickfix.com/validator/process">

</sch:schema>
```

We will first add some namespace information that will dictate how we represent the elements in a) the Schematron document we are writing, and b) the XML document it will constrain **if** that XML document is in a special namespace. We typically set the **Schematron** namespace as a default. (Without this line, we would have to type sch:, a namespace prefix , in front of all of our Schematron elements, so we really prefer to use it.) Paste the line bolded in red below into your new Schematron:

```
<schema xmlns:sch="http://purl.oclc.org/dsdl/schematron" queryBinding="xslt2"
        xmlns:sqf="http://www.schematron-quickfix.com/validator/process"
        xmlns="http://purl.oclc.org/dsdl/schematron">

</schema>
```

If the XML document(s) you're trying to constrain are in a specific namespace, such as the TEI, you must identify that namespace with an empty element called `<ns/>`, and you will also have to use a namespace prefix when representing the XML elements in your schema rules. The next box shows how to define the TEI namespace and its special namespace prefix. If you are writing Schematron to govern TEI XML and you don't define your namespace, or if you forget to use a prefix to point out the elements that belong to that namespace, the Schematron's rules simply will not fire when you associate it with your TEI document(!)

```
<schema xmlns:sch="http://purl.oclc.org/dsdl/schematron" queryBinding="xslt2"
                xmlns:sqf="http://www.schematron-quickfix.com/validator/process"
                xmlns="http://purl.oclc.org/dsdl/schematron">
                <ns uri="http://www.tei-c.org/ns/1.0" prefix="tei"/>

</schema>
```

About namespaces: Documents are in a namespace or in no namespace, as signaled in their root element. We can see in the code above that a Schematron document has a special `xmlns` (or XML namespace) attribute that seems to point to a web address. This is not really a website (though sometimes developers put up placeholder websites at namespace URIs): it's simply a unique uniform resource identifier (that is what URI stands for) and it is simply a unique string of characters used to identify the Schematron namespace. The TEI has its own namespace URI too, and so do other forms of XML (like XSLT) that we are presenting in this course. If your input document is in the TEI namespace (that is, the root element is `<TEI xmlns="http://www.tei-c.org/ns/1.0">`, you have to include the `<ns uri="http://www.tei-c.org/ns/1.0" prefix="tei"/>` element we illustrated above in your Schematron and you must use the `tei:` prefix before all references to elements **(but not attributes)** from the input TEI document in your Schematron file. That means you need to write `//tei:body/tei:div` and not just `//body/div`. Attributes are special because they exist in **no namespace**, so they do not take a prefix (and you will not be able to find them if you apply the prefix). So if we are looking for `@ref` attributes on TEI `<div>` elements, we would write: `//tei:body/tei:div/@ref`. You can think of this as a magic incantation that's needed for Schematron to match just the elements in the TEI document, but if you'd like more explanation of how namespaces work, see http://www.w3schools.com/xml/xml_namespaces.asp.

The skeleton of a Schematron rule

Pattern, Rule, and Context

Each new schema rule starts with a `<pattern>` element. Inside the `<pattern>` is a `<rule>` element with an `@context` attribute. It looks like this:

```
<pattern>
        <rule context=" ">

        </rule>
</pattern>
```

We can set as many rules as we wish inside a `pattern` element, which simply works as a convenient organizing structure for you to put related rules together. A `pattern` element may contain one or multiple `rule` elements as you wish. A `rule` element must have a `@context` attribute that is distinct from other `rule` elements in your Schematron file. The value of `@context` is the specific place in your XML document where the rule applies. (When you have associated your Schematron file with your XML and do validation checking in <oXygen/>, the XPath pattern defined by your `@context` is where <oXygen/> will mark a validation error triggered by a test of your Schematron rule.) The form this takes is called an ***XPath pattern*** and we also use it in XSLT: it is a pattern of elements and/or attributes set in relation to each other that might appear at any level of your document hierarchy: For example, if you write the XPath pattern `p/said` as the value of `@context`, rule context will apply to *any* `<said>` elements within a `<p>` element positioned at any level in the XML document hierarchy, whether the parent `p` element is sitting inside a TEI header in an outer level of the hierarchy or deeply nested inside a `note` element inside another body `p`. **XPath pattern expressions** let us locate particular patterned relationships *wherever they sit in the document hierarchy* so they can be a powerful tool to keep our Schematron and XSLT code tidy and efficient. Why is this more efficient? Because we do not have to write the same rule for `said` elements over and over again depending on the different XPath positions of `p`, and we may save computer parsing time by not starting our searches over and over again from the document node were we to begin with `//p/said`. Constructing an **XPath pattern**, `p/said` takes advantage of the relational patterns that rule-based schema languages are designed for. XPath patterns can also be set to use predicates, so that, for example, `said[@who]` matches on any `<said>` elements that have `@who` attributes anywhere they are sitting in our XML document.

Assert or Report

The `<assert>` or `<report>` element is the heart of each **Schematron** rule. Within each `<rule>` element we can set one or more `<assert>` or `<report>` elements, which contain an attribute called `@test`. With all of these pieces together, here is the basic skeleton of a **Schematron** rule using `<assert>`:

```
<pattern>
        <rule context=" ">
                <assert test=" "> </assert>
        </rule>
</pattern>
```

The value of `@test` is a literal XPath statement *defined in immediate relation to the current XPath location of `@context`, wherever this is*. The `@test` sets a condition for the True or False value of something you write here: For example, does particular string pattern exist here? Does the numerical value of this equal the preceding::sibling of the current context? Imagine the *current context* to be shifting with each discovery of the XPath pattern. As the validation checker lands on each new instance, it runs your `@test` and checks for some condition, true or false, that hinges on that pattern in some way. Basically, `@context` tells <oXygen/> where to look, and `@test` tells <oXygen/> what to test when it gets there. You then type a message, your very own customized validation error message, inside the `<assert>` or `<report>` element as its text content, and explain (to yourself and/or your project team) the reason the rule is firing. When a rule fires, it will generate an alert message in <oXygen/> just like a message from Relax NG, although in Schematron, it's your own custom-made message that fires.

Writing the rules

An assert rule

Okay, now that we understand the structure, let's construct some sample rules so we understand how and why they function. Let's say you're keeping track of points in a game where the goal is to get as many points as possible. The person in first place got 23 points, second place got 16, and third place got 12. Let's construct a basic XML document to store the results:

```
<gameResults>
        <first>23</first>
        <second>16</second>
        <third>12</third>
</gameResults>
```

In our very simple example, the first place score should always be more points than the second place score. Let's write a **Schematron** rule to make sure the values are entered correctly. First, let's start by writing the `<pattern>`, `<rule>`, and `@context`. We want the rule to fire (or alert the user) on the `<gameResults>` element.

```
<pattern>
        <rule context="gameResults">

        </rule>
</pattern>
```

Now, we want to write the rule. We want to **assert** (or say definitively) that the first-place score must always be greater than the second-place score. This means that the rule will fire when the defined assert test **fails**.

```
<pattern>
        <rule context="gameResults">
                <assert test="number(first) gt number(second)">The first-place score must be greater than the second-place scor
        </rule>
</pattern>
```

When we associate our schema, if we have entered 116 instead of 16 for the second place score, our schema will fire an error because what we typed fails to fulfill our Schematron assert test. Notice that we need to use an Xpath `number()` function for our rule to treat the contents of the `first` and `second` elements as a numerical value to be compared. Note: XPath functions that return numerical values are frequently used in Schematron for comparison tests. Some of these functions operate over text content that needs to be converted to numbers as we did here, and some of them calculate and measure things (like string-length() to return a numerical value. Here are the standard wyas to indicate comparisons in XPath and Schematron:

- equality: `eq` or `=`
- greater than: `gt` or `&gt;`
- greater than or equal to: `ge`

- less than: `lt` or `<`
- less than or equal to: `le`
- not equal to: `ne` or `!=`

A note on inconsistency between Relax NG and Schematron: Even if you write a Relax NG schema as we did for our gameResults.xml file to define and xsd:integer data type for the element contents of `first`, `second`, and `third`, we discover that our Schematron still reads the contents of those elements as a string of text until we convert them to a number in XPath. The Relax NG grammar constructs a numerical data format, then, that is nevertheless not read as a number by an XPath parser unless it is prompted to do so. (We provide links to our sample Relax NG and Schematron files so you can test this for yourself.)

A report rule

Now that we have a working schema rule to test the difference between the first- and second- place scores, let's make a rule that tests the second- and third-place scores. The rule is essentially the same (the second-place score is always greater than the third-place score), but we'll use the `report` element instead to demonstrate how it works. We must add a new test within our rule since it shares the same @context in the `gameResults` element. **Note:** If we attempt to define a new rule with the *same*:`@context` as the first, one of the two rules will be applied and the other ignored! So within a given rule @context, we need to define all our assert and report tests together.

When we write a `report` element, we are saying to tell us (flag or report) when a particular condition in an @test is met. The difference between assert and report then, is that an `assert` test fires and error when its assertion is violated, while a `report` test fires and error when its condition is met. In this case, we call for a report when the second-place score (or current context) *is less than or equal to* the third-place score. Using `report` in our second test in the example below, the rule will fire when these conditions **are** met.

```
<pattern>
            <rule context="gameResults">

            <assert test="number(first) gt number(second)">
                The first-place score must be greater than the second-place score.
            </assert>
            <report test="number(second) le number(third)">
                The second-place score must be greater than the  third-place score.
            </report>
        </rule>
</pattern>
```

Here is another way we might write that report statement, to illustrate how we might use the XPath function `not()` wrapped around a test value:

```
 <report test="not(number(second) gt number(third))">
            The second-place score must be greater than the  third-place score.
            </report>
```

Associating a Schematron schema with your XML and testing it

Associating a **Schematron** schema is a lot like associating a Relax NG schema. While viewing your XML document, in the taskbar, click on Document -> Schema -> Associate Schema. From there, locate your schema file (the file extension should be **.sch**). When you associate a .sch file, <oXygen/> should automatically set the schema type to Schematron. *A note on mindful file management:* Remember to save your Schematron in a directory where you can easily and consistently locate it. Finalize that, and <oXygen/> should insert a superscript that looks like this:

```
<?xml-model href="your_file_name.sch" type="application/xml" schematypens="http://purl.oclc.org/dsdl/schematron"?>
```

If you also have a Relax NG schema associated, you will have two different schema lines at the top of your XML document. The two different kinds of schema will function together so that as you code the red square in <oXygen/> will appear as validation errors. The bottom window will feature messages associated with these validation errors, and this will include the messages you write in the text content of your Schematron `assert` and `report` elements.

When you associate your schema, **always tinker with your XML** to create conditions that will cause your Schematron rules to fire! Testing your schema code should be a back-and-forth process to ensure that your assert and report tests are functioning as you want them to.

More information and examples

- Wendell Piez and Debbie Lapeyre's Introduction to Schematron: a very detailed and engaging tutorial with many examples.
- Obdurodon's Examples of Schematron from our projects: See if you can figure out Schematron rules that would constrain the sample cases described here.
- Obdurodon's Using Schematron in editing: See if you can work out a Schematron code that would constrain the XML as described here.
- Obdurodon's tutorial on Validating references with Schematron
- Digital Mitford project Schematron: examples of how to validate @ref attributes with a list of standard xml:ids
- Amadis in Translation project Schematron: a wide range of sample rules to study.
- The Schematron website

<oo>→<dh> Digital humanities

Maintained by: David J. Birnbaum (djbpitt@gmail.com)
Last modified: 2017-04-10T13:50:02+0000

Validating references with Schematron

The text

This activity uses Skyrim (http://dh.obdurodon.org/skyrim.xml), a small text originally prepared by a former participant in our course. The file has a `<cast>` element at the top that contains, as child element, a list of *characters* (`<character>` elements) and *factions* (`<faction>` elements) that are mentioned in the `<body>` section, below the `<cast>` element. For this activity we are going to ignore characters initially and concentrate only on factions.

An entry for a faction in the cast list looks like:

```
<faction id="MythicDawn" alignment="evil"/>
```

Meanwhile, a reference to a faction in the body looks like:

```
The <faction ref="MythicDawn">assassins</faction> first attacked …
```

Note that we use the `<faction>` element differently inside the `<cast>` element (where it has a unique `@id` attribute, plus other attributes we will ignore for now) and inside the `<body>` element (where it has a `@ref` attribute). Any mention of a `<faction>` element in the body must point to a matching `<faction>` element in the cast list. The way the pointing happens is that a `<faction>` element in the body always has a `@ref` attribute, and the value of that `@ref` attribute should point to (= match the value of) the `@id` attribute of some `<faction>` element in the cast list. In other words, 1) there should be no `<faction>` element in the body that does not point to a `<faction>` element in the cast list, and 2) there should be no `<faction>` element in the cast list that is not pointed to by at least one `<faction>` element in the body. The attribute that does the pointing is the `@ref` on the `<faction>` element in the `<body>`; the target of the pointing is an `@id` attribute on a `<faction>` element in the cast list.

You can assume that the developer has used a Relax NG schema and declared that the `@id` attribute is of type `xsd:ID`, that is, that it is an XML id. That means that it has certain properties, including constraints on the characters that it can contain and the fact that it is unique in the document. You can also assume that your Relax NG schema verifies that all `<faction>` elements in the cast list have an `@id` attribute and all `<faction>` elements in the `<body>` have a `@ref` attribute.

How a developer could screw up

There are at least two ways a developer could mangle these cross-references:

1. It's possible to encode a `<faction>` element in the body with a `@ref` attribute that doesn't point to (that is, correspond to) the `@id` attribute of a `<faction>` element in the cast list. For example, the `@id` attribute might be on a `<character>` element in the cast list, instead of on a `<faction>` element, or there might be no corresponding `@id` attribute at all.
2. It's possible to encode an unused `<faction>` element in the cast list, that is, one that is not pointed to by the `@ref` attribute of any `<faction>` element in the body. Since the inventory of factions in the cast list is supposed to summarize which factions occur in the body, such an error would bring the list out of sync with the reality of the body.

The task

We want to write a Schematron schema that will guard against the types of error described above by checking for consistency in two ways:

1. We want to write a rule for `<faction>` elements in the cast list to verify that all factions mentioned there also occur in the body. That is, there should be no faction listed in the cast list that is not also present in the body.
2. We want to write a rule for `<faction>` elements in the body that verifies that they have a `@ref` attribute that points to an `@id` attribute on a `<faction>` element in the cast list. Note that it isn't enough to check for the existence of a corresponding `@id` attribute, since there are `@id` attributes on `<character>` elements in the cast list, and not only on `<faction>` elements. Not only must the `@id` exist, but it must be associated specifically with a `<faction>` element in the cast list, and not with a `<character>` element.

Our solution

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns:sch="http://purl.oclc.org/dsdl/schematron" queryBinding="xslt2"
    xmlns:sqf="http://www.schematron-quickfix.com/validator/process"
    xmlns="http://purl.oclc.org/dsdl/schematron">
    <pattern>
        <let name="cast-factions" value="//cast/faction/@id"/>
        <let name="body-factions" value="//body//faction/@ref"/>
        <rule context="cast/faction">
            <assert test="@id = $body-factions">The @id "<value-of select="@id"/>" occurs in the cast
                list, but not in the body.</assert>
        </rule>
        <rule context="body//faction">
            <assert test="@ref = $cast-factions">The @ref "<value-of select="@ref"/>" occurs in the
                body, but not in the cast list.</assert>
        </rule>
    </pattern>
</schema>
```

We begin by setting some *convenience variables*. What we mean by convenience variable is that we don't have to use them (we could have put the XPath expressions directly in the `@test` attributes), but they make our code more legible. `$cast-factions` is a sequence of all `@id` values for all `<faction>` elements in the cast list. `$body-factions` is a sequence of all `@ref` attributes for all `<faction>` elements in the `<body>`. (We could have used `distinct-values()` to get rid of the duplicates in the list of `@ref` values, but they do no harm in the tests we're running, so we just left them in. There cannot be any duplicates in the list of `@id` values because we have declared them as type `xsd:ID` in our Relax NG schema.)

The first rule fires on each `<faction>` element in the cast list. It checks whether the `@id` attribute value for that element matches the value of any of the `@ref` attributes on `<faction>` elements in the `<body>`. If not, it reports an error. To make the report more informative, we use the Schematron element `<value-of>` to print the offending value.

We use *general equals* (`=`) for this test, rather than *value comparison* (`eq`). General equals takes two sequences of any length and tests whether any member of one is a member of the other. If so, the test succeeds—no matter how many members of the sequence don't match! We have a sequence of one item on the left (the `@id` of the `<faction>` in the cast list that we're testing at the moment, and it qualifies as a sequence in the XPath sense even if it's a sequence of one item) and a sequence of many items on the right (all `@ref` values on all `<faction>` elements in the `<body>`), this test will succeed whenever the `@id` we're examining has a matching `@ref`. This type of one-to-many general comparison is very common in digital humanities coding. (Value comparison with `eq` does only one-to-one comparison, so if you try `eq` here, you'll get an error message because there is a sequence of more than one item on the right side.)

The second rule does the reverse. It fires on every `<faction>` element in the `<body>` and checks whether the value of the `@ref` attribute matches the value of an `@id` attribute on a `<faction>`element in the cast list.

An alternative that works

We could have hung the rules on the `@id` and `@ref` attribute values instead of on the `<faction>` elements that are their parents (making the necessary modifications to the paths in the code), and there's no particular reason to favor one of these strategies over the other.

Alternatives that don't work

It's possible to write rules that fire on `<cast>` or even on `<skyrim>` and that run one check of the entire document. This is much harder to code, although it can be done, but it's also less informative, since it doesn't associate the error message with a specific offending element. Even if you manage to poke the offending value into the error report, the red squiggly line will show up on `<cast>` or `<skyrim>`, so you'll have to work a bit harder to find the element you need to fix.

Some students in past semesters tried to run a single, global test on `<cast>` or `<skyrim>` just to count the number of `@id` values on `<faction>` elements in the cast list and compare that number to the count of distinct values of `@ref` attributes on `<faction>` elements in the `<body>`. That's not a tenable strategy because if, say, the factions in the cast list are "A", "B", and "C" and the ones in the `<body>` are "X", "Y", and "Z", there are three of each, but they don't correspond, you would want that to be reported as an error, and you can't do that just by counting them.

So how about `<character>` elements?

One might think one could use the same type of validation to check for cross-references on `<character>` elements: is every character mentioned in the cast list also encountered in the `<body>` and does every character mentioned in the `<body>` have a `@ref` attribute that points to the `@id` attribute of a `<character>` element in the cast list? This turns out to be harder than with factions because there are elements in the body like:

```
… the <character ref="hero Jauffre MartinSeptim">three of them</character> made their way …
```

The problem here is that there is no `<character>` element in the cast list with an `@id` attribute whose value is "hero Jauffre MartinSeptim". Instead, this is a pointer to three separate characters in the header. The strategy for checking coreference therefore has to involve breaking apart the `@ref` attribute and checking each of the three pointers separately. This is the sort of task for which the XPath `tokenize()` function was created.

There is a hypothetical parallel problem concerning the other half of the assignment. Suppose there is a `<character id="Alex" loyalty="empire" alignment="neutral"/>` element in the head, but the only time Alex occurs in the body is in combination with another character, e.g., `<character ref="Alex Alathia">`. We can't just check whether there is a `@ref` attribute in the body that matches the string "Alex" because there isn't; here, too, we have to break apart the value of the `@ref` and check each part separately. This situation doesn't happen to occur in our text, but it is potentially possible and therefore something against which a well-designed development environment would protect the user.

Our extended solution

To avoid cluttering the screen, the Schematron below checks only `<character>` elements. In real life, you'd combine it with the one above.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns:sch="http://purl.oclc.org/dsdl/schematron" queryBinding="xslt2"
    xmlns:sqf="http://www.schematron-quickfix.com/validator/process"
    xmlns="http://purl.oclc.org/dsdl/schematron">
   <pattern>
        <let name="cast-characters" value="//cast/character/@id"/>
        <let name="body-characters" value="//body//character/@ref"/>
        <rule context="cast/character">
            <assert test="@id = $body-characters">The @id "<value-of select="@id"/>"
                occurs in the cast list, but not in the body.</assert>
        </rule>
        <rule context="body//character">
            <assert test="@ref">The character "<value-of select="."/>" is missing its @ref
                attribute.</assert>
            <assert
                test="every $i in tokenize(normalize-space(@ref),'\s+') satisfies $i = $cast-characters"
                >The @ref "<value-of select="@ref"/>" occurs in the body, but not in the cast
                list.</assert>
        </rule>
    </pattern>
</schema>
```

In our first rule, which fires on `<character>` elements that are children of `<cast>`, we check the `@id` attribute on every `<character>` element in the cast list to verify that it is pointed to by at least one `@ref` attribute on a `<character>` element in the `<body>`

Our second rule verifies that every `<character>` element in the `<body>` has a `@ref` attribute. The developer could have checked this in Relax NG by making the `@ref` attribute obligatory, but she didn't. To our surprise, although we had used this document for other exercises previously, until we wrote this Schematron rule we had never noticed that there are two `<character>` elements in the `<body>` that don't have any `@ref` attribute! This is real inadvertent error, and had we already learned Schematron when the original developer encoded this file, she would have been able to use it to catch this error and fix it.

Once we've confirmed that there is a `@ref` attribute on the `<character>` element in the `<body>` that we're looking at at the moment, we use the XPath `tokenize()` function to break it apart into pieces. We then use the `every X in Y satisfies` construction (Kay, p. 646 ff.) to check each one individually.

Why not use ID/IDREF validation?

The Relax NG `xsd:ID` datatype is guaranteed to be unique in the document, and it is also guaranteed to conform to the lexical specification (= spelling rules) for an *XML non-colonized name*, abbreviated NCName. NCNames must begin with an alphabetic character and can otherwise contain alphanumeric characters and selected punctuation. They cannot contain most punctuation characters and they cannot contain whitespace characters. If you declare an attribute as being of type `xsd:ID` in your schema and try to use an illegal character, <oXygen/> will notify you of the error. You can read a more precise, human-readable description of what is and is not permitted in an NCName (and therefore in an attribute value of type `xsd:ID`) at http://stackoverflow.com/questions/1631396/what-is-an-xsncname-type-and-when-should-it-be-used.

Attributes declared as the Relax NG datatype `xsd:IDREF` must have values that match an item of type `xsd:ID` in the same document. This means that they have the same requirements about legal and illegal characters, and they also have to match (that is, refer to) a real declared `xsd:ID` value in the same document. There is also an `xsd:IDREFS` datatype which refers to a white-space-delimited set of one or more `xsd:ID` values, which means, for example, that you can tag the word "they" in the body of your text and have it refer to the Three Stooges with:

```
… and then <character ref="curly larry moe">they</character> said …
```

In the preceding example, if the `@ref` attribute were declared as having type `xsd:IDREFS`, <oXygen/> would verify that there were values of type `xsd:ID` for “curly”, “larry”, and “moe” in the document, and it would report an error if it couldn’t find all three.

The limitation of ID/IDREF is that it can only compare an `xsd:IDREF` value to all `xsl:ID` values in the document. This imposes two more specific limitations on its utility:

- A parser can confirm that, say, a `@ref` attribute on a `<faction>` element in the `<body>` points to an `xsd:ID` somewhere, it it cannot determine whether the `xsd:ID` is specifically on a `<faction>` element in the `<cast>`.
- A parser cannot check an `xsd:IDREF` attribute value in one document against an `xsd:ID` attribute value in a different document. ID/IDREF valuation happens only within a single document.

The Schematron strategy illustrated above does not suffer from either of these limitations, and this example illustrates how it overcomes the first of them.

newtFire {dhlds}
Maintained by: Elisa E. Beshero-Bondar (ebb8 at pitt.edu) (cc) BY-NC-SA Last modified: Sunday, 15-Oct-2017 19:22:02 EDT.
Powered by firebellies.

Coding with Unique Identifiers and How to Test for them with Schematron

Why and how we use unique identifiers in XML

Most of our projects demand that we compile *prosopography* data, that is, a standard list of unique identifiers for people, places, books, and other named entities. We might compile a prosopography list in a `header` element holding metadata in a project file, or perhaps in a separate file altogether if our project contains multiple XML files and we just want one document to store all the detailed prosopography information relevant anywhere and everywhere throughout our project. Storing a list of prosopography entries involves creating something like a contacts list of everyone you know with their contact information and unique identification information. In XML prosopography lists, we apply the `@xml:id` attribute to hold a unique string of text, different from any other `@xml:id` in your project, designed to identify a particular person, place, or any named thing that you need to distinguish from other named things. In Relax NG, the `@xml:id` is coded to have an `xsd:ID` datatype, which requires a unique string of text for each distinct `@xml:id` and permits no duplicates. The `xsd:ID` datatype permits text only or a combination of text and numbers, but it must not be a string of numbers alone and it must not contain any white space.

A common standardized format for prosopography data is defined by TEI Guidelines, specifically Chapter 13: Names, Dates, People, and Places which provides helpful examples and useful things to think about when distinguishing among proper names. The Digital Mitford project's Site Index file offers several lists combined together in one file, lists of historical people, fictional characters, real and fictional places, books, journal publications, works of art, and more, and it serves as a kind of backbone for all of the project XML files. Each XML file representing a writing by or about the nineteenth-century author Mary Russell Mitford, say, a letter, a poem, a play, or some other kind of text, makes reference to named entities, often several of the kinds we have mentioned, and we are coding them in a way to help keep track of particular people, fictional characters, places, books, etc. whenever they appear (in whatever form they appear) in our files. Think about why we need to do this. One person might be called several different names in different places, so Mitford sometimes affectionately referred to her father as "Drum" or "Papa" while another text in our project might refer to him more formally as "Dr. Mitford" or "George Mitford". The same person might have several nicknames, and in the case of women, their surnames might change in marriage, but we still need a way to track these names and trace them to the same person when Mitford knew them before and after they were married. Here is an example of an entry for Mary Ann Harvey or Mary Ann Davenport from the Digital Mitford site index:

```
<person xml:id="Davenport_MA" sex="2">
                    <persName>
                        <surname type="maiden">Harvey</surname>
                        <surname type="married">Davenport</surname>
                        <forename>Mary</forename>
                        <forename>Ann</forename>
                    </persName>
                   <occupation>actor</occupation>
                   <note type="bio" resp="#lmw">English actor (1759-1843).</note>
</person>
```

We could add other `<persName>` elements inside this entry if we discover that Mary Ann Harvey used a pseudonym or fake name, or had a nickname. We add as much or as little information as we can find and as seems necessary to our project, and our entries are always in a state of development as we keep working on our project. In the body of a Mitford letter, if she mentions that her father has seen Mary Davenport perform in a play, we encode Mary's name like this:

```
<p><persName ref="#Mitford_Geo">Drum</persName> saw <persName ref="#Davenport_MA">Mary Davenport</persName>
         perform in <title ref="#TwelfthNight_Shkspr">Twelfth Night</title> last week.</p>
```

Because we are only permitted to use an `@xml:id` attribute value *once* in our entire project, we place a hashtag on all attributes (like `@ref`, `@wit`, `@corresp` that *refer* or *point to* that `@xml:id`. The use of the hashtag permits us to invoke the identifying string as often as we need to, and it permits us to pull up more information about a given name by pointing to information in our site index file.

How to write Schematron to check hashtagged attributes against `@xml:id` values

Since it is very easy to mistype an attribute value as we are coding our project files, we can either embed our `@xml:id` values in a Relax NG file, or write Schematron rules to help us check their values against our standard list, wherever we have placed it in relation to our project files. We can also write Schematron rules to make sure we remember to include a hashtag at the start of each value when it is in a pointer `@ref` or other attribute that points to an `@xml:id`. That way we can point to those identifiers as many times as we need to in the document. We show you how to do with an example from the Emily Dickinson Fascicle 16 project.

The Dickinson team prepared a list of published editions of Dickinson's poems with `@xml:id` attributes and detailed bibliography information about each, and they compiled each entry inside a TEI `<listWit>` element or a list of witnesses. Each entry on the list holds distinct identifying information about a different *witness* that produced a distinct published representation of Emily Dickinson's manuscript poems. The witnesses produced *variants* or different readings of the same document, and coding these variant readings in the lines of poetry helped the Dickinson team to study how published editions normalized or otherwise altered Dickinson's poems in the different published editions. To mark the different *variants* (or different versions of words and phrases and punctuation) within particular lines in each poem, and the Dickinson team referred to the source of each variant with a hashtagged `@wit` attribute pointing to the `@xml:id` in the `<listWit>` up in their `teiHeader` element. Here is an abbreviated view of their list of witnesses:

```
<listWit>
   [. . .]
      <witness xml:id="poems3">
        <bibl>
            <title>Poems, Third Series</title>
            <author>Emily Dickinson</author>
            <editor>Mabel Loomis Todd</editor>
            <pubPlace>Boston</pubPlace>
            <publisher>Little, Brown and Company</publisher>
            <date>1896</date>
            <ref target="http://catalog.hathitrust.org/Record/100654113">Hathi Trust Digital Library</ref>
        </bibl>
```

```
    </witness>
    <witness xml:id="ce">
        <bibl>
            <title>The Poems of Emily Dickinson: Centenary Edition</title>
            <author>Emily Dickinson</author>
            <editor>Martha Dickinson Bianchi and Alfred Leete Hampson</editor>
            <pubPlace>Boston</pubPlace>
            <publisher>Little, Brown and Company</publisher>
            <date>1930</date>
        </bibl>
    </witness>
    <witness xml:id="fh">
        <bibl>
            <title>Final Harvest: Emily Dickinson's Poems</title>
            <author>Emily Dickinson</author>
            <editor>Thomas H. Johnson</editor>
            <pubPlace>Boston</pubPlace>
            <publisher>Little, Brown and Company</publisher>
            <date>1961</date>
        </bibl>
        [. . .]
    </witness>
</listWit>
```

Schematron can be used to constrain the writing of `@xml:id` values in this list to meet project specifications. For example, the Dickinson team will want to make sure their `@xml:ids` do not begin with hashtags:

```
<pattern>
     <rule context="@xml:id">
            <report test="starts-with(., '#')">
                xml:id attributes must not begin with a hashtag!
            </report>
        </rule>
    </pattern>
```

The Dickinson team uses an `@wit` attribute with a hashtag in the body of the poem to point to *one or more* identified published editions. Often a particular variant is displayed in three or four published editions, and when that is the case, the project team separates each distinct hashtagged identifier with a white space, like this: `<rdg wit="#ce #poems2 #fh">`. These point to `@xml:id` values defined up in the TEI header and its `<listWit>` as: df16, fh, and poems. It is very easy to mistype these ids, so we need a good schema rule to ensure they are correct. Here is how we prepared our rule for the `@wit` attribute

```
<pattern>
                <rule context="@wit">
                    <let name="tokens" value="for $w in tokenize(., '\s+') return substring-after($w, '#')"/>
                    <assert test="every $token in $tokens satisfies $token = //tei:TEI//tei:listWit//@xml:id">
                        Every reading witness (@wit) after the hashtag must match an xml:id defined in the list of witnesses in
                    </assert>
                </rule>
            </pattern>
```

This complex rule permits us to use white space as a separator, so we can refer to multiple published editions that represent a particular variant in the text.

The rule accomplishes several things:

1. The `<let>` element: This defines a *variable* in Schematron, and gives it an `@name` (`"tokens"`) which we can quickly refer to with a dollar-sign in front of it, as `$token`. We can define a variable inside a rule to make it *local* (in which case the parser only "knows" about it and reads it within the context of a particular rule), or we can define it as a *global* variable by setting the `<let>` element above the `<pattern>` elements in the Schematron hierarchy so the variable can be invoked everywhere.

   Note: We make global variables when we need to write Schematron rules that point to other files, to see if a value of an attribute matches an `@xml:id` defined in a separate project file, for example. But variables can be used to hold any complex pattern that you want to invoke in a `rule @context` or in an `@test` on an `assert` or `report` element.

2. Dealing with multiple values: First, in our variable, we tokenized our `@wit` attribute on white space, and that created multiple values or token. So if we *do* use one or more white spaces in an `@wit` attribute, we use those white spaces as a dividing point: we separate the value into "**tokens**": so `<rdg wit="#ce #poems2 #fh">` would be tokenized into three pieces. Our language, `for $w in tokenize(., '\s+')`, defines a separate variable *for each one of these tokens*, since we need to look at them one by one. For each of these, we need to cut off the leading hashtag, so we do one more thing: return the `substring-after($w, '#')`. This creates three tokens in this format:
   - token 1: ce
   - token 2: poems2
   - token 3: fh

3. Now our assert test needs to do something more, so it can deal with a situation in which there's only one token **or** multiple tokens. We can't just test all the tokens at once against each `@xml:id` because Schematron needs to look at them one at a time: first ce, then poems2, then fh. For that one-at-a-time handling, we use this syntax: `<assert test="every $token in $tokens satisfies $token = //tei:TEI//tei:listWit//@xml:id">` The work of this is done by the structure: `every [singular] in [plural] satisfies [a test you design for the singular value]`.

   If the Dickinson team members move their `<listWit>` to separate file of prosopography lists, like the site index of the Mitford project, they would define a variable with another `let` statement pointing to a file in its location relative to the Dickinson project files. The file can be served on the web and pointed to with an absolute web address, or referenced by a relative address as we do here:

```
<let name="si" value="doc('Dickinson_listIds.xml')//@xml:id"/>
    <pattern>
                <rule context="@wit">
                    <let name="tokens" value="for $w in tokenize(., '\s+') return substring-after($w, '#')"/>
                    <assert test="every $token in $tokens satisfies $token = $si">
                        Every reading witness (@wit) after the hashtag must match an xml:id defined in the list of witness
                    </assert>
                </rule>
            </pattern>
```

   Defining the variable `$si` above a `<pattern>` element makes it a *global variable* in Schematron, which means it can be referenced in multiple `pattern` elements. We could also define it *inside* the `<rule>` if we only need to represent the variable within this particular rule. Defining variables as filepaths in Schematron is a convenient way to make your schema rules cross-check values between multiple files. Here it permits us to use one file as a way to validate the attribute values in use on any project file associated with this Schematron file.

<oo>→<dh> Digital humanities

Maintained by: David J. Birnbaum (djbpitt@gmail.com)
Last modified: 2017-10-27T18:54:39+0000

What's new in XSLT 3.0 and XPath 3.1?

1. Introduction

This page aims to provide a brief introduction to small but useful enhancements to XPath and XSLT that have emerged since the publication of Michael Kay's *XSLT 2.0 and XPath 2.0 programmer's reference, 4th edition*, which covers XPath 2.0 and XSLT 2.0. Two of the most significant additions to XSLT 3.0, streaming and packaging, are not covered here because, as important as they are for large files or complex transformations, we haven't found a need for them in the smaller scale on which we usually operate.

2. References

- XML Path Language (XPath) 3.1 W3C Recommendation 21 March 2017
- XPath and XQuery Functions and Operators 3.1. W3C Recommendation 21 March 2017
- XSL Transformations (XSLT) Version 3.0. W3C Recommendation 8 June 2017
- Roger Costello's Pearls of XSLT and XPATH 3.0 design

3. Configuring <oXygen/>

To tell <oXygen/> that new XSLT files should default to XSLT 3.0, click on File → New → XSLT → Customize and select "3.0".

4. XPath 3.0 and 3.1

4.1. Variable declaration

XPath in XSLT allows the use of the `let` construction, which was previously available only in XQuery. See immediately below, under Concatenation.

4.2. Concatenation with `||`

The string concatenation operator `||` can be used in situations that previously required the `concat()` function. For example, the following XPath expression:

```
let $a := 'hi', $b := 'bye' return $a || ' ' || $b
```

is equivalent to:

```
let $a := 'hi',  $b := 'bye' return concat($a,' ', $b)
```

4.3. Simple mapping with the bang operator (`!`)

The bang operator applies the operation to the right of the bang to each item in the sequence on the left. For example:

```
('curly', 'larry', 'moe') ! string-length(.)
```

returns a sequence of three integers: (5, 5, 3). The expression is equivalent to:

```
for $stooge in ('curly', 'larry', 'moe') return string-length($stooge)
```

The simple mapping operator is similar to `/`, except that 1) the sequence to the left of `/` must be a sequence of nodes, while the sequence to the left of `!` can be a sequence of any items, and 2) `/` sorts the sequence on the left into document order and eliminates duplicates, while `!`

performs no sorting or deduplication.

4.4. Function chaining with the arrow operator (=>)

The arrow operator pipes the output of the item on the left into the first argument of the function on the right. It thus provides an alternative to nested parentheses. For example (from the XPath 3.1 spec, §3.16):

```
tokenize((normalize-unicode(upper-case($string))),"\s+")
```

is equivalent to:

```
$string => upper-case() => normalize-unicode() => tokenize("\s+")
```

The functionality of the bang and arrow operators overlaps where the operation on the right is a function, but only then. For that reason:

```
$book ! (@author, @title)
```

return the values of the `@author` and `@title` attributes of some element that is the value of the variable `$book`, but because the operation on the right is not function, if you replace the bang with the arrow operator, you throw an error. The arrow operator does not use the dot to specify the first argument to the function because the operator supplies that argument instead.

Because the bang operator is a mapping and the arrow operator is a pipe, the following two expressions produce different results:

```
'curly larry moe' => tokenize('\s+') => count()
```

The preceding returns the integer value “3”. But

```
'curly larry moe' ! tokenize(.,'\s+') ! count(.)
```

returns a sequence of three instances of the integer value “1”. The difference is that after `tokenize()` returns a sequence of three items, the bang operator maps each item individually as the input to the `count()` function, while the arrow operator counts the items in the sequence.

4.5. `unparsed-text-lines()`

`unparsed-text-lines()` works like `unparsed-text()`, except that it tokenizes on newlines and streams the input line by line.

4.6. Maps

The following example creates a map and then serializes it as JSON on output:

```
<xsl:variable name="mymap" as="map(*)"
        select='map {
        "Su" : "Sunday",
        "Mo" : "Monday",
        "Tu" : "Tuesday",
        "We" : "Wednesday",
        "Th" : "Thursday",
        "Fr" : "Friday",
        "Sa" : "Saturday"
        }'/>
    <xsl:template match="/">
        <root>
            <text>Hi, Mom! Here’s some information:</text>
            <para>{
                serialize($mymap, map{"method":"json","indent":true()})
            }</para>
        </root>
    </xsl:template>
```

$stuff?row ...

In eXist-db, to create a map using a `for` loop use something like:

```
declare variable $map as map(*) :=
    map:merge(for $i in $realTitles return map:entry($i, count($items/tei:title[. eq $i])));
```

The `map:entry()` function creates anonymous separate one-item maps with the string values of `$realTitles` as the keys and the number of times each title appears in the corpus as the value. Wrapping the FLWOR in the `map:merge()` function merges the individual maps into a single map, which is assigned to the variable `$map`. Note the syntax of the value specified by the `as` operator, which is necessary (should we choose to specify a datatype) because maps are not traditional atomic types. To access the values of the map, use something like:

```
for $bg in map:keys($map)
let $en as element(en) := $titles[bg eq $bg]/en
order by $en
return <option value="{$bg}">{$en || ' (' || $map($bg) || ')'}</option>
```

This gets each key from the map, uses it to retrieve the English translation of a Bulgarian title from the `$titles` variable, and then also uses it as the single argument of the `$map()` function to retrieve the number of times the Bulgarian title appears in the corpus. Note that because maps are functions, instead of indexing into them with square brackets, we execute them with the key as the single argument to the function, and the argument is in parentheses, as is usual for functions.

4.7. Arrays

Add stuff here

5. XSLT 3.0

5.1. Boolean values

Boolean values can be expressed as any of “true”/“1”/“yes” or “false”/“0”/“no”. For example, to turn on pretty-printed output, set the value of the `@indent` attribute of `<xsl:output>` to any of “true”, “1”, or “yes”.

5.2. Starting from a named template

If you set the value of the `@name` attribute of an `<xsl:template>` element to “xsl:initial-template” and run a transformation from the command line with the “-it” (= ‘initial template’) switch, the template named “xsl:initial-template” is now the default. Previously you had to specify the name of your initial template on the command line.

5.3. Content Value Templates

Like Attribute Value Templates, Content Value Templates let you specify that certain text should be intepreted as XPath instead of being output literally. The syntax for CVTs is the same as for AVTs: surround the expression in curly braces (to use a literal curly brace, double them), and multiple values are output with a single space between them. CVTs work ony if you create an `@expand-text` attribute on the root `<xsl:stylesheet>` element and give it a positive Boolean value. CVTs are similar to the use of curly braces in XQuery to switch from XML mode into XQuery mode, and they can be used in situations where you may previously have had to use `<xsl:value-of>` or something that converts its arguments to strings, like `concat()` or `||`. Here’s an example:

```
<xsl:template name="xsl:initial-template">Hello, World! It’s {current-time()}</xsl:template>
```

The preceding is equivalent to:

```
<xsl:template name="xsl:initial-template">
    <xsl:text>Hello, World! It’s </xsl:text>
    <xsl:value-of select="current-time()"/>
</xsl:template>
```

or

```
<xsl:template name="xsl:initial-template">
    <xsl:value-of select="concat('Hello, World! It’s ', current-time())"/>
</xsl:template>
```

or

```
<xsl:template match="/">
    <xsl:value-of select="'Hello, World! It's ' || current-time()"/>
</xsl:template>
```

5.4. `@item-separator`

The `@item-separator` attribute on `<xsl:output>` can be used to change the item separator from the default space to something else. Must be combined with `@build-tree="no"`.

5.5. Shadow attributes

Shadow attributes mask regular attribute values, and have the same name as the regular attribute, but with a leading underscore.

5.6. Variables and functions

Functions can be assigned to a variable. To reference them, add parentheses after the variable name.

5.7. Creating HTML5

To create HTML5 output, use `<xsl:output method="html" version="5"/>`. This creates HTML5 using HTML (not XML) syntax, which means that it omits the XML declaration and it creates a `<meta>` element inside the `<head>`. If you serve your HTML5 as mime type "application/xhtml+xml" and want to validate it as XML, set `@method` to "xml" instead (and set `@indent` to a positive Boolean value unless that messes up your white space). Setting the `@method` to "html" also doesn't add the HTML namespace automatically (fair enough).

5.8. Identity transformation

The identity transformation can be expressed in a single top-level `<xsl:mode>` element:

```
<xsl:mode on-no-match="shallow-copy"/>
```

5.9. Iteration

Iteration may sometimes be easier to write than recursion. The following code returns a running total of the integers from 1 through 10:

```
<xsl:iterate select="1 to 10">
    <xsl:param name="total" as="xs:integer" select="0"/>
    <xsl:variable name="newTotal" as="xs:integer" select="$total + ."/>
    <xsl:value-of select="concat($total, ' + ', . , ' = ' , $newTotal, '&#x0a;')"/>
    <xsl:next-iteration>
        <xsl:with-param name="total" select="$newTotal"/>
    </xsl:next-iteration>
</xsl:iterate>
```

This outputs the results of each iteration. To output only the final total, remove the `<xsl:value-of>` statement and use `<xsl:on-completion>`:

```
<xsl:iterate select="1 to 10">
    <xsl:param name="total" as="xs:integer" select="0"/>
    <xsl:on-completion select="$total"/>
    <xsl:variable name="newTotal" as="xs:integer" select="$total + ."/>
    <xsl:next-iteration>
        <xsl:with-param name="total" select="$newTotal"/>
    </xsl:next-iteration>
</xsl:iterate>
```

although for this contrived problem it would, of course, be simpler to write `<xsl:value-of select="sum(1 to 10)"/>`.

A recursive template call might look like:

```
<xsl:template match="/">
    <xsl:variable name="result">
        <xsl:call-template name="accumulate">
            <xsl:with-param name="total" select="0"/>
            <xsl:with-param name="range" select="1 to 10"/>
        </xsl:call-template>
    </xsl:variable>
    <xsl:sequence select="$result"/>
</xsl:template>
<xsl:template name="accumulate">
    <xsl:param name="total" as="xs:integer"/>
    <xsl:param name="range" as="xs:integer*"/>
    <xsl:choose>
        <xsl:when test="empty($range)">
            <xsl:sequence select="'done'"/>
        </xsl:when>
        <xsl:otherwise>
            <xsl:variable name="currentValue" as="xs:integer" select="$range[1]"/>
            <xsl:variable name="newTotal" as="xs:integer" select="$total + $currentValue"/>
            <xsl:value-of select=" concat($total, ' + ', $currentValue, ' = ', $newTotal, '&#x0a;')"/>
            <xsl:call-template name="accumulate">
                <xsl:with-param name="total" as="xs:integer" select="$newTotal"/>
                <xsl:with-param name="range" as="xs:integer*" select="remove($range, 1)"/>
            </xsl:call-template>
        </xsl:otherwise>
    </xsl:choose>
</xsl:template>
```

This returns a report on each step plus the word “done” at the end. To see just the steps, make `<xsl:when>` an empty element. To return just the total, remove the `<xsl:value-of>` from the `<xsl:otherwise>` element and set the value of the sequence returned inside `<xsl:when>` to `$total`.

<oo>→<dh> Digital humanities

Maintained by: David J. Birnbaum (djbpitt@gmail.com)
Last modified: 2018-01-31T00:45:33+0000

Regex assignment #1

What to submit

We describe below how to use regex to transform plain text into XML, which is the task for this assignment. We don't need to see your XML, but we do need to see a step-by-step explanation of how you used regex to create it. *That explanation should be a plain-text document (not a word-processor document);* the reason is that a word processor might convert your straight quotation marks to curly ones or make other typographic changes that are desirable when you're writing a term paper, but that will corrupt your regex (curly quotation marks are not legal replacements for straight ones in regex).

You can create your plain text explanation of your regex conversion process in <oXygen/>. Create a new document in <oXygen/>, selecting "Text" as the document type. Write up your process there, hitting the "Enter" or "Return" key at the end of each line to keep the line length manageable (<oXygen/> does not wrap long lines automatically, and pretty-printing doesn't work with plain text documents). Save your document with a ".txt" filename extention, which is the conventional extension for plain text documents.

The input text

The text we'll be using as input for the first regex homework assignment is a plain-text version of Shakespeare's sonnets, which you should download from http://www.gutenberg.org/cache/epub/1041/pg1041.txt and open in <oXygen/>. (Note that this site sometimes shows you a pop-up welcome screen and a list of different versions of the file, instead of taking you to the plain text one directly. If that happens, click OK on the Welcome pop-up and then select the version labeled Plain Text UTF-8.) You should manually delete any of the Project Gutenberg information from the beginning and end of this file, so that what you're left with is just the sonnets in order, with roman numerals before each one.

If you find it easier to work with a small amount of text, you can make yourself a document that contains just a few sonnets and use that during development. Once you think you're getting the results you want, you can then try applying the same strategy to the entire file. The structure of this document is very regular, so whatever works for a handful of sonnets should work for all of them.

The task

Your goal is to produce an XML version of this file by using the search-and-replace techniques we discussed in class. Your output should look something like http://dh.obdurodon.org/shakespeare-sonnets.xml. That is, each sonnet should be its own element, each line should be tagged separately, and the roman numerals should be encoded in a useful way (we've used attributes, but you could also put them in a child element).

How to proceed

There are several ways to get to the target output, but here is how we might approach the task:

Reserved characters

The plain text file could, at least in principle, contain characters that have special meaning in XML: the ampersand and the angle brackets. You need to search for those and replace them with their corresponding XML entities; if you don't remember the entity strings, you can look them up in the "Entities and numerical character references" section of http://dh.obdurodon.org/what-is-xml.xhtml. Note that you need to process them in the correct order. What is that order, and why is it important?

Title and author

The title and author at the top are going to have to be tagged manually. You can either remove them now and then paste them back in later, after you've tagged the sonnets, or you can leave them in place and fix them up at the end. You'll use global find-and-replace to tag the sonnets, and if you leave the title and author in place while you do that, you'll wind up tagging them incorrectly. That isn't a problem as long as you remember to fix them manually at the end.

To perform regex searching, you need to check the box labeled "Regular expression" at the bottom of the <oXygen/> find-and-replace dialog box, which you open with Control-f (Windows) or Command-f (Mac). If you don't check this box, <oXygen/> will just search for what you type literally, and it won't recognize that some characters in regex have special meaning. You don't have to check anything else yet. Be sure that "Dot matches all" is unchecked, though; we'll explain why below.

Leading space characters

The non-blank lines all begin with space characters: there are two spaces before most lines (the Roman numerals and the first twelve lines of each sonnet) and four spaces before the last two lines of every sonnet. Those spaces are presentational formatting, and not part of the content of the text, and since we don't need them in order to tag the text, we'll start by deleting them. The regex to match a space character is just a space character, and you can match one or more space characters by using the plus sign repetition indicator. To match one or more instances of the letter "X", you would use a regex like `X+`. To match one or more instances of a space character, just replace the "X" with a space.

You don't want to remove all space characters, though; you just want to remove the ones at the beginning of a line. You can do that by using the caret metacharacter, which anchors a match so that it succeeds only at the beginning of a line. For example, if the regex `X+` matches one or more instances of "X", the regex `^X+` matches one or more instances of "X" *only at the beginning of line*. You can use this information to match one or more space characters at the beginning of a line and replace them with nothing, that is, delete them.

We aren't going to use the blank lines in this approach, so you can delete those if you'd like, or you can leave them in place to enhance the legibility. To delete them, you need to match a blank line, and the easiest way to do that is to match two new line characters in a row and replace them with a single new line character. The regex for a new line character is `\n`. Try it.

Inside out or outside in

We can create our markup either from the outside in (document, then sonnet, then divide the sonnet into Roman numeral and lines) or from the inside out (lines and Roman numeral, then wrap those in a sonnet, then wrap all of the sonnets in a document). Either strategy can be made to work, but we generally find it easier to work from the inside out because when we work from outside in, it's easy to wind up incorrectly wrapping `<line>` tags around the `<sonnet>` start and end tags, etc.

Lines

We'll start by tagging every line as a `<line>`. This will erroneously tag the Roman numerals as if they were lines of poetry, which they aren't, but it's easier to let the first find-and-replace overgeneralize and then go back and retag the Roman numerals than to try to write a more constrained regex that won't overgeneralize. We don't want to tag blank lines (if we left them in), though, so we need a regex that matches only lines that have characters in them. Remember where we told you above to make sure that "Dot matches all" was unchecked? Normally the dot (`.`) matches any character except a new line, which means that we can use the plus sign repetition indicator to match one or more instances of any character except a new line (that is, `.+`). By default regex selects the longest possible match, so even though just two characters on a line will match the pattern, when we run it it will always match the entire line. Since the dot matches any character except a new line, the regex will match each line individually, that is, it won't run over a new line and continue the same match. Try it and examine the results. Now check "Dot matches all", run Find all, and look at those results. Notice that the match no longer stops at the end of the line, and since you want to tag each line individually, you need to uncheck that box to revert to the normal, default behavior.

A human might think of our task as "wrap every line in `<line>` tags", but regex has a find-and-replace view of the world, so a regex way to think about it would be "match every line, delete it, and replace it with itself wrapped in `<line>` tags". That is, regex doesn't think about leaving the line in place and inserting something before and after it; it thinks about matching the line, deleting it, and then putting it back, but with the addition of the desired tags. The regex selects and matches each full line, but how do we write what we selected into the replacement string? The answer is that the sequence `\0` in the replacement pattern means "the entire regex match", and you can use that to write the matched line back into the replacement, but wrapped in `<line>` tags. Try it.

Roman numerals

The Roman numerals are now erroneously tagged as if they were lines of poetry, and in our sample output at http://dh.obdurodon.org/shakespeare-sonnets.xml we want them to be attribute values. To start that process we need to think about how to distinguish a Roman numeral line from a real line of poetry. Since there are 154 sonnets, a Roman numeral line is a line that contains one or more instances of "I", "V", "X", "L", and "C" in any order and nothing else, and no real line of poetry matches that pattern. That means that we can match that pattern by using a regex *character class*, which you can read about at http://www.regular-expressions.info/charclass.html. This approach will match sequences that aren't Roman numerals, like "XVX", but those don't occur, so we don't have to worry about them. This illustrates a useful strategy: a simple regex that overgeneralizes vacuously may be more useful than a complex one that avoids matching things that won't occur anyway. You can use the character class (wrapped in square brackets) followed by a plus sign (meaning one or more) to complete your regex so that it matches only `<line>` elements that contain a Roman numeral and nothing but a Roman numeral. Try it.

In this case you want to write the Roman numeral into the replacement string, but you want to get rid of the spurious `<line>` tags and replace them with other markup. `\0` will write the entire match into the replacement, but that would include the original `<line>` tags that you want to remove. To capture just part of a regex match for reuse in the replacement, you wrap it in parentheses; this doesn't match

parenthesis characters, but it does make the part of the regex that's between the parentheses available for reuse in the replacement string. For example, `a(b)c` would match the sequence "abc" and capture the "b" in the middle, so that it could be written into the replacement. Capturing a single literal character value isn't very useful because you could have just written the "b" into the replacement literally, but you can also capture wildcard matches. For example, `a(.)c` matches a sequence of a literal "a" character followed by any single character except a new line followed by a literal "c" character. You can use that type of approach to capture everything between the `<line>` tags in the matched string: write a regex that matches the entire line with the Roman numeral, including the `<line>` tags, but put parentheses around the stuff between the `<line>` tags.

Okay, you've captured the Roman numeral, but how do you write it into the replacement? To write a captured pattern into the replacement, use a backslash followed by a digit, where `\1` means the first captured group, `\2` means the second, etc. Since in this case we're capturing only one group (we have only one set of parentheses), wherever we write `\1` in our replacement string, we'll insert the Roman numeral that we captured. For this task we'd build a replacement string that starts with a `</sonnet>` end tag (since the Roman numeral appears after the end of the preceding sonnet), then a new line, and then a `<sonnet>` start tag, and inside that start tag we'd include the `number` attribute and use the captured string (that is, `\1`) as its value, etc. Try it.

Clean up

You may have to clean up the beginning and end of the document manually, including the title and author, and you'll also need to add a root element.

Checking your results

Although you've added XML markup to the document, <oXygen/> remembers that you opened it as plain text, which means that you can't check it for well-formedness. To fix that, save it as XML with File → Save as and give it the extension ".xml". Even that doesn't tell <oXygen/> that you've changed the file type, though; you have to close the file and reopen it. When you do that, <oXygen/> now knows that it's XML, so you can verify that it's well formed in the usual way: Control+Shift+W on Windows, Command+Shift+W on Mac, or click on the arrow next to the red check mark in the icon bar at the top and choose "Check well-formedness".

General

As we mention above, there are several ways to get to the target output, and whatever works is legitimate, as long as you make meaningful use of computational tools, including regular expressions (where appropriate), and don't just tag everything manually. As you saw in class, there are ways to build your own regular expressions to match whatever patterns you need to identify, and the regex languages is complex and often difficult to read. The way we would approach this task is by figuring out what we need to match and then looking up how to match it. In addition to the mini-tutorial above, there is a more comprehensive description in the regex section of Michael Kay's book and more detailed tutorial information at http://www.regular-expressions.info/tutorialcnt.html. If you decide to look around for alternative reference sites and find something that seems especially useful, please post the URL on the discussion boards, so that your classmates can also consult it.

What to submit (reminder)

We don’t need to see the XML that you produce as the output of your transformation because we’re going to recreate it ourselves anyway, but you do need to upload a step-by-step description of what you did (see the explanation at the top of the page of why and how to create a plain text write-up). Your write-up can be brief and concise, but it should provide enough information to enable us to duplicate the procedure you followed.

If you don’t get all the way to a solution, just upload the description of what you did, what the output looked like, and why you were not able to proceed any further. As always, you’re encouraged to post any questions on the discussion boards, in this case in the regex forum.

<oo>→<dh> Digital humanities

Maintained by: David J. Birnbaum (djbpitt@gmail.com)
Last modified: 2015-09-21T12:41:08+0000

Regex assignment #2

The task

Assume that we've been given a plain-text file like the Project Gutenberg EBook of *The Blithedale Romance*, by Nathaniel Hawthorne, and we want to convert it to XML, but we don't want to type all of the angle brackets manually. (Note that this site sometimes shows you a pop-up welcome screen and a list of different versions of the file, instead of taking you to the plain text one directly. If that happens, click "OK" on the "Welcome" pop-up and then select the version labeled "Plain Text UTF-8".) In this case Project Gutenberg makes the same book available in HTML, and in Real Life we'd probably convert from HTML to XML (using XSLT, which we'll learn later in the semester) rather than from plain text, but since there are situations where all we have is plain text, we'll ignore the HTML version on the Gutenberg site, pretend that all they provide is plain text, and work with that. So what can we tag automatically, with global find-and-replace operations? Some of the markup we might want to introduce for analytical purposes might require us to touch every word of the text, but, at a minimum, we can autotag chapters, chapter titles, paragraphs, and quotations using regex tools, and that's the goal of the present assignment.

Preliminaries

Select the plain-text version of the document and open it in <oXygen/> as a plain text file. Then cut out the front matter (before the main title title) and the back matter (after the last line of the text of the novel, which is "I--I myself--was in love--with--Priscilla! "). In Real Life we might want to mark those parts up eventually and reintroduce them into the XML as metadata, but for this assignment we'll just delete everything that isn't part of the text of the novel.

Step by step

There's more than one way to accomplish this task, but one way to approach the problem is as follows:

Reserved characters

The plain text file could, at least in principle, contain characters that have special meaning in XML: the ampersand and the angle brackets. You need to search for those and replace them with their corresponding XML entities; if you don't remember the entity strings, you can look them up in the "Entities and numerical character references" section of http://dh.obdurodon.org/what-is-xml.xhtml. Note that you need to process them in the correct order. What is that order, and why is it important?

Extra blank lines

The blank lines are pseudo-markup that tell us where titles and paragraphs begin and end, but in some cases there are multiple blank lines in a row (for example, there are two blank lines between the title and the word "by"). Those extra blank lines don't tell us anything useful, so we'll start by getting rid of them. We want to retain one blank line between titles and paragraphs, etc., but not more than one.

To perform regex searching, you need to check the box labeled "Regular expression" at the bottom of the <oXygen/> find-and-replace dialog box, which you open with Control-f (Windows) or Command-f (Mac). If you don't check the "Regular expression" box, <oXygen/> will just search for what you type literally, and it won't recognize that some characters in regex have special meaning. You don't have to check anything else yet.

The regex escape code that matches a new line is `\n`, so you want to search for more than two of those in succession, and you want to replace them with exactly two. You can search for three blank lines and replace them with two and then keep repeating the process until there are no instances of three blank lines left, or, more elegantly and efficiently, you can search for `\n{3,}`, which matches three or more new line characters in succession (see the "Limiting repetition" section of http://www.regular-expressions.info/repeat.html) and replace them with `\n\n` (the quantifiers work only in matches, but not in replacements, so you have to write it this way).

Note that a transformation that searches for a sequence of two end-of-line characters depends on their being immediately adjacent to each other. If what looks like a blank line to you actually has (invisible) spaces or tabs, the pattern won't match and the replacement won't happen because there will be spaces or tabs between the end-of-line characters, which is to say that they won't be adjacent. If you think that might be the case, you can make those characters visible by going into the <oXygen/> preferences (Preferences → Editor) and checking the boxes labeled "Show TAB/NBSP/EOL/EOF marks" and "Show SPACE marks" under Whitespaces. If you do have whitespace characters interfering with your ability to find a blank line (that is, two consecutive new line characters), you can use regex processing to replace them: the pattern `\t` matches a tab character, a space matches a space, and `\s+` matches one or more white-space characters of any sort (including new lines). You can use the "Find" or "Find all" options in the find-and-replace dialog to explore the document and make sure that you're matching what you want to match before you use "Replace all" to make the changes.

Paragraphs

What's left after deleting the beginning and ending metadata and extra blank lines is mostly (except for the stuff at the top) a bunch of chapter titles and paragraphs, separated from one another by a single blank line, and we can use a regex to find all blank lines and replace them with the sequence `</p><p>`. XML doesn't care about the following, but for human legibility, we'd suggest inserting a new line character between the tags, instead of just outputting the end tag followed immediately by the start tag, so that each paragraph will start on a new line. You'll have to add the `<p>` start tag before the first paragraph and the `</p>` end tag after the last one manually, but you can enter all of the rest automatically with a single regex-aware find-and-replace operation. At this point the document looks like a bunch of `<p>` elements. Some may contain chapter titles, rather than paragraphs. We'll fix that below. At the top of the file, the title, author, and list of chapter titles will need special handling. We'll talk about those below, too.

Chapter titles

The title of the first chapter within the body looks like:

```
<p>I. OLD MOODIE</p>
```

the second looks like:

```
<p>II. BLITHEDALE</p>
```

and we can see easily, from the list of chapter titles at the top, that there are twenty-nine chapter titles, each of which begins with a Roman numeral, then a period, and then a single space character, and each of which runs until the end of the line. No real textual paragraph looks like that, although some paragraphs could begin with the pronoun "I", which looks like a Roman numeral, and some paragraphs might be only one line long. If we can write a regex that matches chapter titles and only chapter titles, then, we can replace the paragraph markup with title markup, retaining the part in the middle.

We're not going to write that regex for you, but we will tell you the pieces we used. Try building a regex and running "Find all" to verify that it is matching all of the chapter titles and nothing else. When you can match what you need, then you can think about how to craft the replacement string. Here are the pieces:

- First make sure that, under "Options", "Case sensitive" is checked and "Dot matches all" is unchecked. You want to do case sensitive matching because the Roman numeral characters here are all upper case, so you want to be able to distinguish those from lower case "i", "v", "x", etc. We'll discuss when to use "Dot matches all" below, but for now, make sure that it's unchecked.
- You want to match the entire content of a line, and you can do that by using the `^` (line start) and `$` (line end) *anchor* metacharacters. If you type, say, "A" into the "Find" box and hit "Find all", you'll match every upper-case "A" in the document (try it). But if you type "^A", you'll only find an "A" at the very beginning of a line. In other words `^` doesn't match a caret character; what it does is anchor the match so that it succeeds only if it falls at the beginning of a line. Similarly, the `$` doesn't match a literal dollar sign; what it does instead is anchor a match so that it succeeds only if it falls at the end of a line. This also means that `^A$` will match only lines that consist of nothing except the letter "A". You don't want to match lines that consist of nothing but the letter "A", but with the two anchors (`^`, `$`), the knowledge that the dot (`.`) matches any character except a new line, andyour knowledge of the regex repetition indicators (`?`, `+`, `*`), you have all the pieces you need to craft a regular expression that will match an entire line, no matter what the contents.
- A chapter title is (now) wrapped (misleadingly) in `<p>` tags and fills a single line. That may not always be the case with other texts you'll need to process, but you can see that it is here, and you can take advantage of that fact by searching for lines that begin with `<p>` and end with `</p>` (using the line start and line end metacharacters to match those tags only at the beginning and end of lines). But you also need to match the stuff between the tags, and that's different for every chapter. You can handle that situation, as you did when you matched entire lines above, by using the regex dot (`.`) metacharacter, which matches any character except a new line. And since you don't know the exact number of characters in each title, you can match one or more characters by using the plus sign (`+`) *repetition indicator*, which means "one or more". Now try putting these pieces together and matching all lines that begin with a `<p>` start tag, continue with one or more characters (any characters except a new line), and end with a `</p>` end tag. When you look at the results, you'll see that you've matched all of the chapter titles, but also all other one-line paragraphs. You've also matched the title, author, and a few other lines near that top, but you'll need to repair those manually at the end anyway.
- You now need to refine your regex so that you'll continue to match chapter titles, but not other one-line paragraphs. Since chapter titles begin with a Roman numeral, you can modify your regex to match only if a Roman numeral immediately follows the `<p>` start tag. To do that you'll use a *character class*, which you can read about at http://www.regular-expressions.info/charclass.html. You want to match any sequence of "I", "V", and "X" characters in any order. This will match

sequences that aren't Roman numerals, like "XVX", but those don't occur, so you don't have to worry about them. This illustrates a useful strategy: a simple regex that overgeneralizes vacuously may be more useful than a complex one that avoids matching things that won't occur anyway. You can use the character class (wrapped in square brackets) followed by a plus sign (meaning "one or more") to enhance your regex and match only one-line paragraphs that begin with something that looks like a Roman numeral. Try it.

- This almost works, but it also matches one-line paragraphs that begin with the first person singular pronoun "I", such as:

```
<p>I mentioned those rumors to Hollingsworth in a playful way.</p>
```

  To weed those out, you want to match a Roman numeral only if it's followed immediately by a period. Since the dot in regex is a metacharacter that matches any character except a new line, if you want to match a literal period, you have to *escape* the dot character by preceding it with a backslash (`\`). Add that after the Roman numeral part of your regex, and you should be matching only the twenty-nine chapter-title lines.
- Matching the chapter titles is necessary but not sufficient: you now need to replace the paragraph tags with `<title>` tags. To do that we need to *capture* the part of the title line that's between the paragraph tags and write that captured text into the replacement. To capture part of a regex, you wrap it in parentheses; this doesn't match parenthesis characters, but it does make the part of the regex that's between the parentheses available for reuse in the replacement string. For example, `a(b)c` would match the sequence "abc" and capture the "b" in the middle, so that it could be written into the replacement. Capturing a single literal character value isn't very useful because you could have just written the "b" into the replacement literally, but you can also capture wildcard matches. For example, `a(.)c` matches a sequence of a literal "a" character followed by any single character except a new line followed by a literal "c" character. You can use that information to capture everything between the paragraph tags in the matched string. To write a captured pattern into the replacement, use a backslash followed by a digit, where `\1` means the first capture group, `\2` means the second, etc. In this case you're capturing only one group, so you can build a replacement string that starts with `<title>`, ends with `</title>`, and puts `\1` between them. You don't need to do anything about the line start and line end anchors; since you've matched an entire line, the replacement will automatically be an entire line.
- Putting this all together, you should be able to retag your titles automatically, distinguishing them from the paragraphs. Try it.

Chapters

A book isn't just a series of paragraphs with titles strewn among them; the book has logical chapters, which must begin with a title, and you want to represent this part of the logical document hierarchy in your markup by inserting `<chapter>` tags. Much as you used blank lines as *milestone* delimiters between paragraphs earlier, you can now use your `<title>` elements as delimiters between chapters. Use a find-and-replace operation to do this; you'll have to clean up the markup before the first chapter and after the last one manually, since in those cases the `<title>` element doesn't have the same milestone function as elsewhere.

Quotes

How are quotations represented in the plain text? How would you find the text of a quotation, that is, how would you find where it starts, where it ends, and what goes between the start and the end? Files on the

Internet sometimes have errors and inconsistencies; if you're relying on cues in the text to identify the beginnings and ends of quotations, what can happen if you miss one?

If we assume that a quotation is text between opening and closing quotation marks (which are the same in this text, which has straight quotation marks, instead of the curly typographic ones where the opening and closing shapes are different), we have at least two concerns:

- A line may have more than one quotation. If we write a regex like `".+"` (including the quotation marks), will we match each quotation individually, or will we match the first quotation mark on the line and the last, erroneously gobbling up everything between into one spurious quotation? Try it and see.
- Some quotations span multiple lines. Since the dot matches any character except a new line, if we write `".+"` and the start and end quotation marks are on different lines, we'll fail to match those quotations, and we may erroneously match material between ending and starting quotation marks, instead of only between starting and ending ones. Try it and see.

Let's address the second problem first. There's a line in the text that reads:

```
without further question, only," added she, "it would be a convenience
```

which represents the end of one quotation and the beginning of another. If we write `".+"`, the system will incorrectly think that the first quotation mark opens a quotation and the second closes one, and it will also fail to recognize that the material before and after that line is really part of a quotation. We can fix this by checking the "Dot matches all" box, which changes the meaning of the dot metacharacter from "any character except a new line" to "any character including a new line". This means that we should be able to match quotations that cross line boundaries. Try it and notice the different results. Uh-oh!

So what went wrong? By default regular expressions are *greedy*, which means that they make the longest possible match. Turning on *dot all* mode causes the regex to match everything from the very first quotation mark in the entire text through the very last (since quotation mark characters are also characters, the dot in the regex `".+"` matches the quotation marks between the first and last ones in the document, just like it matches any other character). Turning off dot all mode won't fix this because some quotations do cross line boundaries, and we need to be able to recognize and match them.

We can resolve the problem by turning on dot-all mode (since we have to match quotations that span line breaks) but also specifying that the match should be *non-greedy*, that is, that we should make the *shortest* possible match (instead of the longest, which is the default), and we do this by following the repetition indicator (the plus sign) with a question mark. (Note that the question mark you met earlier is a repetition indicator that means "zero or one instance" of whatever it follows. Here is isn't a repetition indicator, though; here it means "don't be greedy". So if the same symbol can have two such different meanings, how does a regex processor know which meaning to apply?) In other words `".+?"` will correctly treat two full quotations on the same line as separate quotations. Try it. You should now correctly be matching each quotation fully, regardless of whether it spans a new line character and regardless of the number of quotations on a line.

Once you can do that, you can capture the text between the quotation marks and write it into the output between `<quote>` tags. Don't include the quotation mark characters themselves in the capture group; those are plain-text pseudo-markup, and now that you're going to be tagging quotations with real markup, you don't want the quotation mark characters included.

Cleanup

At this point you can fix the title and author lines manually (we'd just delete the line that reads "by", since the new `<author>` tags will make that explicit), as well as the table of contents, and you need to wrap the entire document in a root element (such as `<book>`). If you'd like a little more regex practice, instead of fixing the table of contents manually, you can use regex find-and-replace to tag it. If you select the table of contents and then open the find-and-replace dialog, you can check the "Only selected lines" box under "Scope" to say that instead of applying find-and replace operations to the entire file, you'll apply them only to the selected lines. You may want to start by stripping out incorrect markup that you've inserted when your global find-and-replace operations earlier changed these lines, as well—and of course you'll want to do that with a regex that matches any tag and replaces it with nothing (that is, deletes it). Once you've done that, these lines look like title lines, except that they have space characters before them, and you can use a regex that matches one or more space characters to help match them. You can then capture each line (throwing away the leading white space by excluding it from the capture) and wrap it in `<title>` tags. You'll want to get rid of the paragraph tags that are wrapping the whole table of contents, since it isn't a paragraph, and replace it with something like `<toc>` (for "table of contents").

Checking your results

Although you've added XML markup to the document, <oXygen/> remembers that you opened it as plain text, which means that you can't check it for well-formedness. To fix that, save it as XML with File → Save as and give it the extension ".xml". Even that doesn't tell <oXygen/> that you've changed the file type, though; you have to close the file and reopen it. When you do that, <oXygen/> now knows that it's XML, so you can verify that it's well formed in the usual way: Control+Shift+W on Windows, Command+Shift+W on Mac, or click on the arrow next to the red check mark in the icon bar at the top and choose "Check well-formedness".

What to submit

We don't need to see the XML that you produce as the output of your transformation because we're going to recreate it ourselves anyway, but you do need to upload a step-by-step description of what you did. Your write-up can be brief and concise, but it should provide enough information to enable us to duplicate the procedure you followed.

If you don't get all the way to a solution, just upload the description of what you did, what the output looked like, and why you were not able to proceed any further. As always, you're encouraged to post any questions on the discussion boards, in this case in the regex forum.

<oo>→<dh> Digital humanities

Maintained by: David J. Birnbaum (djbpitt@gmail.com)
Last modified: 2015-07-29T13:11:20+0000

Regex assignment #3

The text

Oscar Wilde's *The importance of being Earnest* is available in plain text from Project Gutenberg at http://www.gutenberg.org/cache/epub/844/pg844.txt. Download the text and manually remove the Project Gutenberg boilerplate from the beginning and end, so that all that remains is the text as Oscar Wilde wrote it.

The task

Your task is to prepare an XML-encoded digital edition of this play from the plain text using search and replace operations to introduce the markup. The specific markup you use is up to you, but as is appropriate for a play, you will want your XML to identify at least acts, scenes, speeches, speakers, and stage directions. Note that your goal is to use search and replace operations, with or without regular expressions, to create *descriptive* well-formed XML markup (rather than, for example, to create a presentational HTML editon). You should not use manual tagging except in situations that occur so rarely that they don't justify search and replace operations or stylesheet transformations (such as tagging the title of the play or creating a root element).

When you have completed your tagging, you should upload the XML document you create along with a separate page describing any global search and replace operations you used (through the search and replace dialog box) to introduce markup.

There is no single target output for this assignment. Any well-formed markup you create that is appropriate and sensible for the play is fine.

<oo>→<dh> Digital humanities

Maintained by: David J. Birnbaum (djbpitt@gmail.com)
Last modified: 2018-02-16T17:35:24+0000

Test #3: Regular Expressions

The task

For this test, we are asking you to up-convert a one-act play called *The bicyclers and three other farces* using regular expressions, which is available at http://dh.obdurodon.org/2018-02-16_regex-test.txt. Please copy and paste the text of the play into a plain text file in <oXygen/> and use the regex search and replace function to add structural tags to the play.

We encourage you to familiarize yourself with the format of the document (e.g., the division of scenes, the format of speeches and speaker names, etc.) before beginning your tagging. As was the case with the regex homework assignments, we recommend that you first search for any reserved characters and normalize the new line characters in the play, and then use regex to tag specific textual components. Our solution tagged the following:

- Each of the four scenes should be wrapped in `<scene>` tags, and the title of each scene (inside the `<scene>` element) should be wrapped in `<title>` tags.
- The setting description that precedes each individual scene should be wrapped in `<sceneDesc>` tags.
- The list of characters at the beginning of each scene should be wrapped, all together, in a single set of `<characters>` tags. (See also the bonus question, below.)
- All stage directions should be wrapped in `<stage>` tags.
- Each speech in the play, comprising both the speaker name and the spoken lines, should be wrapped in `<speech>` tags.
- Inside each `<speech>` element, the name of the speaker should be wrapped in `<speaker>` tags and the spoken text (all of it, together) should be wrapped in a single pair of `<line>` tags.

A sample speech might be tagged as follows:

```
<speech>
   <speaker>Mrs. Perkins</speaker>
   <lines><stage>foreseeing a quarrel</stage> Thaddeus! 'Sh! Ah, by-the-
   way, Mr. Bradley, where is Emma this evening? I never knew you to be
   separated before.</lines>
</speech>
```

Bonus: Within the list of character names at the beginning of each scene, wrap each item in the list in `<character>` tags, with the character's name in `<name>` tags and his or her character description in `<desc>` tags.

In our solution, we performed each of these steps with a single search and replace, but any solution that makes meaningful use of regex to tag your text is fine. **Remember to tell us explicitly when you use "dot matches all".**

You should pseudo mark-up whenever possible (for example,, the period that follows each speaker's name and the "CURTAIN" string that denotes the end of a scene). You should manually tag the play title and contents at the very beginning before you begin your up-conversion, or you should remove them until you've completed your up-conversion, tagging them at the end.

What to submit

Upload to CourseWeb only a step-by-step description of how you performed your up-conversion. We will then apply the steps to the plain text play to recreate your XML result. **Don't forget to save your tagged file as XML file after you finish your work and then reopen it in <oXygen/> to check for well-formedness!**

<oo>→<dh> Digital humanities

Maintained by: David J. Birnbaum (djbpitt@gmail.com)
Last modified: 2015-02-08T19:27:00+0000

XPath assignment #1

You can find an XML (TEI) version of Shakespeare's *Hamlet* at http://dh.obdurodon.org/bad-hamlet.xml. We've deliberately damaged some of the markup in this edition to introduce some inconsistencies, but the file is well-formed XML, which means that you can use XPath to explore it. You should download this file to your computer (typically that means right-clicking on the link and selecting "save as") and open it in <oXygen/>.

After you've completed your homework, save your answers to a file and upload it to CourseWeb as an attachment. (Please use an attachment! If you paste your answer into the text box, CourseWeb may munch the angle brackets.) Some of these tasks are thought-provoking, and even difficult. If you get stuck, do the best you can, and if you can't get a working answer, give the answers you tried and explain where they failed to get the results you wanted. Sometimes doing that will help you figure out what's wrong, and even when it doesn't, it will help us identify the difficult moments. These tasks require the use of path expressions, predicates, and the functions count() and not(), but they should not require any other XPath functions. There may be more than one possible answer.

Using the *Bad Hamlet* document and the XPath browser window in <oXygen/>, construct XPath expressions that will do the following (give the full XPath expressions in your answers, and not just the results):

1. Hamlet, like a typical Shakespearean tragedy, contains five acts, each of which contains scenes. Both acts and scenes are encoded as division (`<div>`) elements.
    a. How can XPath tell them apart?
    b. What XPath would find just the acts?
    c. What XPath would find just the scenes?
    d. What XPath would find just the scenes in Act III?
2. Stage directions (`<stage>`) occur in a variety of contexts.
    a. What XPath would find all of the stage directions that are inside a metrical line (`<l>`), that is, between the starting `<l>` and the ending `</l>`. How many are there?
    b. What XPath would find all of the stage directions that are directly inside a speech (`<sp>`), that is, inside a speech but not inside a line within a speech?
    c. What XPath would find all of the stage directions that are not directly inside a speech or a line. How many are there?
    d. For the stage directions you identified in #2c, above, write an XPath expression that will return not the `<stage>` elements themselves, but their parent elements, whatever they might be. What are those parent elements? (You haven't yet learned the XPath to return just the names of the parent elements [rather than the elements themselves], but you can locate them, click on each one in the list <oXygen/> returns, and look at it directly.)

<oo>→<dh> Digital humanities

Maintained by: David J. Birnbaum (djbpitt@gmail.com)
Last modified: 2015-08-24T19:05:34+0000

XPath assignment #2

You can find an XML (TEI) version of Shakespeare's *Hamlet* at http://dh.obdurodon.org/bad-hamlet.xml. We've deliberately damaged some of the markup in this edition to introduce some inconsistencies, but the file is well-formed XML, which means that you can use XPath to explore it. You should download this file to your computer (typically that means right-clicking on the link and selecting "save as") and open it in <oXygen/>.

After you've completed your homework, save your answers to a file and upload it to CourseWeb as an attachment. (Please use an attachment! If you paste your answer into the text box, CourseWeb may munch the angle brackets.) Some of these tasks are thought-provoking, and even difficult. If you get stuck, do the best you can, and if you can't get a working answer, give the answers you tried and explain where they failed to get the results you wanted. Sometimes doing that will help you figure out what's wrong, and even when it doesn't, it will help us identify the difficult moments. These tasks require the use of path expressions, predicates, and the functions `count()` and `not()`, but they should not require any other XPath functions. There may be more than one possible answer.

Using the *Bad Hamlet* document and the XPath browser window in <oXygen/>, construct XPath expressions that will do the following (give the full XPath expressions in your answers, and not just the results):

1. Most (not all) speeches in Hamlet contain, as their immediate children, mostly metrical line (`<l>`) and "anonymous block" (`<ab>`) elements (an anonymous block is the TEI element that the tagger used to represent a non-metrical speech line). Speeches also typically contain, as immediate child elements, `<speaker>` elements, and may also contain stage directions (`<stage>`). We have deliberately left out at least one other type of subelement found as an immediate child of speech elements. Based on this understanding:
    - What XPath would find all of the speeches that do not contain any metrical lines as immediate children? How many are there?
    - What XPath would find all of the speeches that do not contain either any metrical lines (`<l>`) or any anonymous blocks (`<ab>`) as immediate children? How many are there? What do they contain instead? (As in #2d in XPath assignment #1, you haven't yet learned the XPath to return a list of the types of elements they do contain, but if you find them all, you can scan the brief list that <oXygen/> returns, click on each one to see it in context, and see what's going on.)
2. Explain why the following four XPath expressions return two different results, and describe in prose what each of them does return, and why:
    - //sp[@who="Hamlet"]/l[1]
    - /descendant::sp[@who="Hamlet"]/l[1]
    - (//sp[@who="Hamlet"]/l)[1]
    - (/descendant::sp[@who="Hamlet"]/l)[1]

<oo>→<dh> Digital humanities

Maintained by: David J. Birnbaum (djbpitt@gmail.com)
Last modified: 2015-10-11T18:32:10+0000

XPath assignment #3

You can find an XML (TEI) version of Shakespeare's *Hamlet* at http://dh.obdurodon.org/bad-hamlet.xml. We've deliberately damaged some of the markup in this edition to introduce some inconsistencies, but the file is well-formed XML, which means that you can use XPath to explore it. You should download this file to your computer (typically that means right-clicking on the link and selecting "save as") and open it in <oXygen/>.

After you've completed your homework, save your answers to a file and upload it to CourseWeb as an attachment. (Please use an attachment! If you paste your answer into the text box, CourseWeb may munch the angle brackets.) Some of these tasks are thought-provoking, and even difficult. If you get stuck, do the best you can, and if you can't get a working answer, give the answers you tried and explain where they failed to get the results you wanted. Sometimes doing that will help you figure out what's wrong, and even when it doesn't, it will help us identify the difficult moments. These tasks require the use of path expressions, predicates, and functions. There may be more than one possible answer.

Using the *Bad Hamlet* document and the XPath browser window in <oXygen/>, construct XPath expressions that will do the following (give the full XPath expressions in your answers, and not just the results):

1. What XPath expressions will find the last stage direction `<stage>` in the entire document? (Note: there should be only one!)
2. What XPath expression will find the last member in the cast list at the beginning of the document and return the value of the `@xml:id` attribute that is associated with it?
3. What XPath expression will find all `<sp>` elements with more than 8 line (`<l>`) subelements? You'll need to use the `count()` function (Kay 733–34).
4. Building on your answer to the preceding question, what XPath expression will tell you how many line subelements each of those speeches actually has?
5. Building on your answers to the preceding two questions, what XPath expression will find the *speakers* of all speeches that have more than 8 line subelements? Once you've found the speeches that have more than 8 lines, you can find the speakers of those speeches by just adding another path step, but you'll get some duplication, since a single person may have more than one long speech. Your answer to this question should get rid of the duplicates, and return just a list of names of speakers without duplication. You'll need to use the `distinct-values()` function (Kay 749–50).

Optional bonus questions

1. Question #1, above, asked how you to provide an XPath that would find the last stage direction (`<stage>`) in the play. What XPath would find the last line (`<l>`) in the play? What XPath would find the last stage direction *or* line (that is, whichever of the last stage direction and last line comes last)? You'll need to use the *union operator* (Kay 628–31).

2. Question #2, above, asked you to provide an XPath that would find the `@xml:id` associated with the last cast member in the cast list. What's the difference between an XPath that returns the `@xml:id` attribute itself and an XPath that returns just the *value* of the `@xml:id` attribute? That is, what are the two XPath expressions and what object does each of them return? You'll need to use the `data()` or `string()` function (Kay 741–43, 877–79).
3. Question #3, above, asked you to provide an XPath that would find all of the speeches (`<sp>` elements) with more than 8 line (`<l>`) subelements. What are the XPaths to find speeches with more than 8 line *child* elements and speeches with more than 8 *descendant* line elements? How do those results differ? If there are descendant line elements that are not children of a speech, what is their parent? If you don't know the types of their parent elements in advance, what XPath expression will tell you?

<oo>→<dh> Digital humanities

Maintained by: David J. Birnbaum (djbpitt@gmail.com)
Last modified: 2014-09-28T20:57:54+0000

XPath assignment #4

You can find an XML (TEI) version of Shakespeare's *Hamlet* at http://dh.obdurodon.org/bad-hamlet.xml. We've deliberately damaged some of the markup in this edition to introduce some inconsistencies, but the file is well-formed XML, which means that you can use XPath to explore it. You should download this file to your computer (typically that means right-clicking on the link and selecting "save as") and open it in <oXygen/>.

After you've completed your homework, save your answers to a file and upload it to CourseWeb as an attachment. (Please use an attachment! If you paste your answer into the text box, CourseWeb may munch the angle brackets.) Some of these tasks are thought-provoking, and even difficult. If you get stuck, do the best you can, and if you can't get a working answer, give the answers you tried and explain where they failed to get the results you wanted. Sometimes doing that will help you figure out what's wrong, and even when it doesn't, it will help us identify the difficult moments. These tasks require the use of path expressions, predicates, and functions. There may be more than one possible answer.

Using the *Bad Hamlet* document and the XPath browser window in <oXygen/>, construct XPath expressions that will do the following (give the full XPath expressions in your answers, and not just the results):

1. What XPath will return a hyphen-separated list of all characters without duplicates. The resulting list will look something like:

   ```
   Claudius-Hamlet-Polonius ...
   ```

   Our solution uses `distinct-values()` and `string-join()`. Note that there are several ways to identify the characters in this markup, including the `<castList>` element, the `<speaker>` elements, and the `@who` atribute on the `<sp>` element. Which should you use and why?
2. Most metrical lines (`<l>`) have an `@xml:id` attribute with a value like "sha-ham101010", ending in a six-digit number. The first digit is the act, the next two the scene, and the last three the line in the scene. Some metrical lines are split across multiple speakers, and in that case the six-digit number in the `@xml:id` value is followed by "I" (initial part), "M" (middle part), or "F" (final part). In a few places there may be more than one middle part, and in those cases the "M" is followed by a one-digit number. For example, one of Hamlet's lines is:

   ```
   <l xml:id="sha-ham502277M2" n="277">One.</l>
   ```

   which is the second middle part. What XPath will return the number of `<l>` elements that are middle parts? Our solution uses `count()` and `contains()`.

3. Sometimes Rosencrantz speaks by himself and sometimes he speaks in unison with Guildenstern.
    a. What XPath finds all of the speeches by Rosencrantz, whether alone or together with Guildenstern? Our solution uses a single instance of `contains()`.
    b. Can you think of an alternative solution that doesn't use any functions (just a predicate)?
4. The `string-length()` function can be used in two ways. You can wrap it around an argument, so that, for example, `string-length('Hi, Mom!')` will return 8, the length in character count of the string inside the quotation marks. It can also be used as part of a path expression, so that, for example, if the XPath `//sp` returns a sequence of all `<sp>` elements, `//sp/string-length(.)` returns a sequence of the lengths of all `<sp>` elements as measured by counting characters. This works by finding all of the `<sp>` elements and then (next path step) getting the string length of each one. Remember that the dot inside the parentheses refers to the current context node, which is the member of the sequence of `<sp>` nodes that is being processed at the moment. We need to use the subterfuge because `string-length(//sp)` generates an error. The problem is that `string-length()` can take only a single argument, and `//sp` returns more than one item. Putting the `string-length()` function on its own path step with a dot inside means that it applies once for every `<sp>` element, and that each time it applies, it has just a single argument.

    Use this information to identify an XPath that finds the length of the longest speech. What length does it return? Our solution uses `string-length()` and `max()`.
5. *Optional, challenging question:* Given the preceding solution, how can you use that XPath to retrieve the longest `<sp>` itself? No fair checking the length and then writing a separate XPath that looks for that number. Your answer must find the longest speech without your knowing how long it is. Our solution doesn't require any additional functions beyond the ones used in #4, but it does use a complicated predicate.
6. *Optional, very challenging question:* What XPath produces a numbered list of all characters, without any duplicates, which should look something like:
    1. Claudius
    2. Hamlet
    3. Polonius
    4. ...

    There are several possible solutions, each of which raises issues that you may not have seen before. If you get an error message, try to figure out what it means and how to resolve it.

<oo>→<dh> Digital humanities

Maintained by: David J. Birnbaum (djbpitt@gmail.com)
Last modified: 2018-03-02T00:17:53+0000

Test #4: XPath

The task

Using *Bad Hamlet*, provide an XPath expression that retrieves:

1. All speeches (`<sp>`) by Ophelia that contain Hamlet's name. Requires, at least in our solution, `contains()`. (There are two such speeches.)
2. A semicolon-separated list of all unique speakers (`<speaker>`) in Act IV, without duplicates. Requires, at least in our solution, `string-join()` and `distinct-values()`. Your list will include, among other, "Rosencrantz", "Guildenstern", and, because they sometimes speak together, "Rosencrantz and Guildenstern". For the purpose of this test, you don't have to get rid of the last of these (but see the bonus question, below).
3. The number of speeches (`<sp>`) in each act (`//body/div`). Our solution requires `count()`. (The number of speeches you should find are 251 for Act 1, 201 for Act 2, 249 for Act 3, 179 for Act 4, and 257 for Act 5.)
4. The speaker elements (`<speaker>`) for all speeches (`<sp>`) that are greater than 4000 characters long. Requires, at least in our solution, `string-length()`. Hint: to approach this in stages: 1) find all speeches; 2) filter them to keep just the ones that are more than 4000 characters in length; 3) find the speakers of those speeches. (There are two such speeches, one by "Hamlet" and one by "Ghost".)
5. This question has four parts, which build incrementally on one another:
    a. The number of lines (`<l>` elements) in each speech (`<sp>` element). Here and in the other parts of this question, count the descendants of the speech, not just the children.
    b. The number of lines in the longest speech (measured in `<l>` elements). The answer is 58.
    c. The longest speech (`<sp>`) itself. It's the speech by Hamlet that begins with the line that has the `@xml:id` value "sha-ham202553".
    d. The `<head>` of the scene that contains that speech. The answer is "Act 2, Scene 2".

Bonus

How can you use XPath to get the semi-spurious "Rosencrantz and Guildenstern" out of the answer to #2? Your answer should cater to the following possibilities:

- There could be other pairs of simultaneous speakers, that is, Horatio and Ophelia might, in principle also speak together.
- Although Rosencrantz and Guildenstern both happen to speak individually in this act, in principle there might be characters who speak together but not separately, and you need to list them individually in your list. For example, if Rosencrantz spoke separately and with Guildenstern, and Guildenstern spoke only together with Rosencrantz, your list of speakers should include both of them individually.

- It's possible in principle for there to be more than two simultaneous speakers, and in that case they might be joined with both commas and "and", along the lines of "King, Gertrude, Hamlet, and Servant". These persons may or may not also speak individually, so you need to be sure that they all make it into your list as separate entries.

<oo>→<dh> Digital humanities

Maintained by: David J. Birnbaum (djbpitt@gmail.com)
Last modified: 2015-03-23T03:47:54+0000

XQuery assignment #1

Use the 42 Shakespeare plays that have been uploaded to Obdurodon to do the following:

1. Find all of the titles of all of the Shakespeare texts in the corpus. You'll need to read our posting on the main course page on Obdurodon for information about how to address the collection of plays, and also about how to retrieve the full text of one of the plays so that you can look at it and see where the title is, which you'll need to know in order to construct the XPath to retrieve it. The simplest answer is a single XPath expression. The output should look something like (there are 42 of them):

   ```
   <title xmlns="http://www.tei-c.org/ns/1.0">Othello, the Moor of Venice</title>
   <title xmlns="http://www.tei-c.org/ns/1.0">The Second Part of King Henry the Fourth</title>
   <title xmlns="http://www.tei-c.org/ns/1.0">The Taming of the Shrew</title>
   ```

   Here are two important issues:
   - The Shakespeare texts are in the TEI namespace. To use namespaces in XQuery, you declare them as follows:

     ```
     declare namespace tei="http://www.tei-c.org/ns/1.0";
     ```

     This should go on the second line of your XQuery, just below the XQuery declaration. It declares that you will use the prefix `tei:` to refer to elements in the TEI namespace. That means, for example, that you should describe the TEI header as `tei:teiHeader`.
   - There are several `<title>` elements in the plays, and not all of them are titles of plays. Some, for example, may be titles of acts or scenes. You can find the titles of plays by using XPath, and you may want to examine a sample play to remind yourself of how the TEI encodes that type of title.
2. Modify your XPath above to return just the text of the titles, without the tags. You can do that by using `text()` or `data()` or `string()` (which you might want to look up in Kay or at w3schools). Your answer should look something like:

   ```
   Othello, the Moor of Venice
   The Second Part of King Henry the Fourth
   The Taming of the Shrew
   ```

3. Fourteen of the 42 plays have more than 40 unique speakers. Find those plays and return their titles. You will need to use `count()` and `distinct-values()` (and don't forget the TEI namespace!). Find the collection, drill down to the `<TEI>` elements in the collection (you know there are 42 of them), then filter them based on whether or not they contain more than 40 distinct `<speaker>` element values. Once you're getting the 14 plays that meet that description, you can add a path step to retrieve their titles.
4. Modify your solution to the preceding question #3 to return just the text of the play titles, without the `<title>` tags. You can take the same approach as you did for the transition from question #1 to question #2.

Copy and paste your XQuery expressions from eXide into a plain-text document (you can create one in <oXygen/> or in the plain-text editor of your choice) and upload that document as your homework submission. *Do not use Word, which may turn your well-formed straight apostrophes and quotation marks into curly ones, which don't have the same meaning and which will result in invalid XQuery code.* We do not need the results returned by your query; all we need is the XQuery expression itself.

<oo>→<dh> Digital humanities

Maintained by: David J. Birnbaum (djbpitt@gmail.com)
Last modified: 2017-04-03T15:31:32+0000

XQuery assignment #2

About FLWOR constructions

This assignment requires the use of a FLWOR construction, so you might want to review the assigned reading before tackling it. Be sure that you understand what each of the components of a FLWOR does, and how they interact. For example, you can get the title of every play in the corpus with a single XPath statement:

```
xquery version "3.0";
declare namespace tei="http://www.tei-c.org/ns/1.0";
collection('/db/apps/shakespeare/data')//tei:titleStmt//tei:title
```

To get the same result with a FLWOR expression (alphabetizing the titles on the way), you might use:

```
xquery version "3.0";
declare namespace tei="http://www.tei-c.org/ns/1.0";
let $plays := collection('/db/apps/shakespeare/data')
for $play in $plays
order by $play
return $play//tei:titleStmt/tei:title
```

We use the `let` statement to set a variable `$plays` equal to the collection of all 42 plays. We then use a `for` statement to iterate over the plays, doing something once for each play. Remember that the components of a FLWOR expression must occur in FLWOR (for, let, where, order by, return) order, except that you can have as many `for` and `let` statements as you want, and they can occur in any order with respect to one another. If there is a `where` statement, it must follow all of the `for` and `let` statements; if there is an `order by` statement, it must come next; and the `return` statement must come last. There must be at least one `for` or `let` statement and exactly one `return` statement; everything else is optional.

While we iterate over the plays with our `for` statement, we use an `order by` statement to sort them alphabetically, taking advantage of the fact that the first textual content of each play is the title, so alphabetizing by the entire text of the play is equivalent to alphabetizing by the title. Note that when we return the title of each play, the XPath in the `return` statement has to begin with the variable `$play`, so that on each of the 42 iterations we'll be getting the title of the play we're looking at at that moment.

The texts

Obdurodon contains the text of forty-two Shakespearean plays. In the TEI markup for these plays, speeches are `<sp>` elements and the speakers are `<speaker>` elements that are their first children. For example:

```
<sp who="Roderigo">
    <speaker>Roderigo</speaker>
    <l xml:id="sha-oth101040F" n="40">I would not follow him then.</l>
</sp>
```

There are 966 distinct speaker names (values of the `<speaker>` element) in the entire corpus, some of which show up in more than one play. Three of the names show up in more than ten plays: there is a character called “Messenger” in 22 plays, one called “All” in 20 plays, and one called “Servant” in 18. (These aren’t the same messenger or all or servant, of course!) Here are the details:

- Messenger appears in 22 plays: Antony and Cleopatra; Coriolanus; Cymbeline; Hamlet, Prince of Denmark; Julius Caesar; King Lear; Macbeth; Much Ado About Nothing; Othello, the Moor of Venice; Pericles, Prince of Tyre; The First Part of King Henry the Fourth; The First Part of King Henry the Sixth; The Life and Death of King John; The Life of King Henry the Eighth; The Merchant of Venice; The Second Part of King Henry the Fourth; The Second Part of King Henry the Sixth; The Taming of the Shrew; The Third Part of King Henry the Sixth; The Tragedy of King Richard the Third; Timon of Athens; Titus Andronicus
- All appears in 20 plays: A Midsummer Night’s Dream; All’s Well That Ends Well; Antony and Cleopatra; As You Like It; Coriolanus; Cymbeline; Hamlet, Prince of Denmark; Julius Caesar; Macbeth; Pericles, Prince of Tyre; The First Part of King Henry the Sixth; The Life of King Henry the Eighth; The Life of King Henry the Fifth; The Merry Wives of Windsor; The Second Part of King Henry the Sixth; The Taming of the Shrew; The Third Part of King Henry the Sixth; The Tragedy of King Richard the Second; Timon of Athens; Titus Andronicus
- Servant appears in 18 plays: All’s Well That Ends Well; Hamlet, Prince of Denmark; Julius Caesar; Macbeth; Measure for Measure; Pericles, Prince of Tyre; Romeo and Juliet; The Comedy of Errors; The First Part of King Henry the Fourth; The First Part of King Henry the Sixth; The Life of King Henry the Eighth; The Merry Wives of Windsor; The Second Part of King Henry the Sixth; The Taming of the Shrew; The Tragedy of King Richard the Second; The Winter’s Tale; Timon of Athens; Twelfth Night or What You Will

Assignment

Your assignment is to write XQuery that will query the collection of plays, find the three `<speaker>` element values that occur in more than ten plays, and return a list of those elements along with the names of the plays in which they occur. That is, you should write XQuery that generates the basic information in the preceding list. Once your XQuery is working, copy it into a plain-text document (not a Word document), save it, and upload it to CourseWeb. You can create a plain-text (that is, “.txt”) document in <oXygen/> by creating a new document of type “Text”. We don’t need to see the output of your XQuery; all we need is the XQuery code itself.

Our list includes some enhancements that you can consider optional for this assignment, but if you get the basic solution, we hope you’ll try them:

- We’ve added some plain text (“ appears in ”, “ plays: ”) and punctuation (the list of play titles is preceded by a colon and the individual titles are separated by semicolons). Whether you use this specific formatting is optional, but you have to use some sort of formatting to produce a readable list. An alternative might be to output HTML nested lists, where the outer list gives the character

names, with the frequency values after the names in parentheses, and the sublists under each name contain the titles of the relevant plays. The formatting is up to you, as long as it's clear, legible, and sensible.
- We've sorted the results in descending order of frequency, so that the character name that appears 22 times is first, then the one that appears 20 times, then the one that appears 18 times.
- We've alphabetized the titles of the play. We had to do that explicitly in our XQuery; by default they come out in no particular order (well, there is a particular order, but it's the order in which they were uploaded into eXist, so it isn't meaningful to humans).
- The XML uses straight apostrophes in the titles, but we've replaced them with curly (typographic) ones.

Our solution used several `let` statements to configure "convenience" variables. We used `distinct-values()` and a `for` statement to find and iterate over all of the distinct `<speaker>` names in the collection. We used the `count()` function and a `where` statement to find all of the names of speakers that occur in more than ten plays. Finally, we used both `concat()` and `string-join()` in our `return` statement to glue the pieces together. For the optional parts, we used an `order by` statement to sort the results in descending order of frequency, and to alphabetize the play titles we used an embedded FLWOR statement. That means that we set a variable equal to the return of a FLWOR statement, which is a powerful way of using FLWOR not just for the main program flow, but also to control subcomponents. We used the XPath `translate()` function to fix the apostrophe.

This is not a difficult problem intellectually: you can paraphrase it easily as "find all of the speaker names that show up in more than ten plays." It is, however, a complicated pattern to formalize, and because the XQuery version has to be explicit, it is longer than the prose formulation. We would suggest breaking the problem down into subcomponents like

- How would you find all of the plays in the corpus?
- How would you find the title of a play, or of all of the plays in the corpus?
- How would you find all all of the speakers (that is, all of the <speaker> elements) in a play or in the corpus?
- How would you find all of the *distinct* speakers in a play or in the corpus?
- How would you find all the plays in the corpus in which a speaker with a certain name appears, and then count them?
- How would you find all the plays in the corpus in which a speaker with a certain name appears, and then list their titles?
- If you can count the number of plays in which each distinct speaker name occurs in the corpus, how can you use that information to sort your results in descending order by frequency?
- If you can find the titles of the plays in which a particular speaker name appears, how can you sort them alphabetically?

Some of the preceding questions are easier than others, but getting all the way to a solution means getting them all correct and stitching them together effectively, which is why we recommend developing your XQuery step by step and testing it after every step, so that when it breaks, you'll know which statement caused the problem. If you can't get all the way to a solution, that's okay, but you can't just give up; what you need to do instead is submit your solutions to as many of the pieces as you can (both the ones we list above and any others that you recognize as important). While this particular assignment is a bit (okay, more than a bit) artificial, the techniques required are very common in Real Life: find information across documents and create a report based on an aggregated result.

Watch out for namespaces!

The plays are all in the TEI namespace. This means that you'll want to declare a namespace prefix and map it to the TEI namespace, as you did in the last assignment. What tripped up several people in the last assignment, though, is that *all* element names in your XPath expressions require the namespace prefix. For example, if you've set the value of a variable `$plays` to the 42 play documents in the corpus and if you've also set the prefix `tei:` to point to the TEI namespace, you can get the titles of all of the plays with the XPath expression `$plays//tei:titleStmt/tei:title`. If you forget the prefix on either `<titleStmt>` or `<title>`, though, you won't get any results because you'll be looking for one or both of those elements in no namespace, and every element in this corpus is in the TEI namespace.

<oo>→<dh> Digital humanities

Maintained by: David J. Birnbaum (djbpitt@gmail.com)
Last modified: 2018-02-28T14:12:07+0000

XSLT assignment #1

The assignment

Your assignment is to create an XSLT stylesheet that will transform *Bad Hamlet* into a hierarchical outline of the titles of acts and scenes in HTML. This isn't very interesting on its own, of course, but if you were transforming the entire document into HTML for publication on the web, this might serve as the skeleton. It might also stand on its own as a table of contents at the top of such a publication, so that the reader could click on the title of a scene to jump to that location in the file.

If you're feeling adventurous, you're welcome to include more information, whether of a publication-oriented sort (e.g., speakers, speeches, stage directions, etc., as if you were publishing the entire play) or as a foray into exploration and analysis (e.g., list of characters who speak in each scene, perhaps with a count of their speeches, length of speeches, etc). The only required content of your homework, though, is the HTML outline of act and title chapters, which might look something like:

- Act 1
  - Act 1, Scene 1
  - Act 1, Scene 2
  - Act 1, Scene 3
  - Act 1, Scene 4
  - Act 1, Scene 5
- Act 2
  - Act 2, Scene 1
  - Act 2, Scene 2
- Act 3
  - Act 3, Scene 1
  - Act 3, Scene 2
  - Act 3, Scene 3
  - Act 3, Scene 4
- Act 4
  - Act 4, Scene 1
  - Act 4, Scene 2
  - Act 4, Scene 3
  - Act 4, Scene 4
  - Act 4, Scene 5
  - Act 4, Scene 6
  - Act 4, Scene 7
- Act 5
  - Act 5, Scene 1
  - Act 5, Scene 2

The underlying HTML, which we generated using XSLT, is:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html
  SYSTEM "about:legacy-compat">
<html xmlns="http://www.w3.org/1999/xhtml">
   <head>
      <title>Hamlet</title>
   </head>
   <body>
      <ul>
         <li>Act 1
            <ul>
               <li>Act 1, Scene 1</li>
               <li>Act 1, Scene 2</li>
               <li>Act 1, Scene 3</li>
               <li>Act 1, Scene 4</li>
               <li>Act 1, Scene 5</li>
            </ul>
         </li>
         <li>Act 2
            <ul>
               <li>Act 2, Scene 1</li>
               <li>Act 2, Scene 2</li>
            </ul>
         </li>
         <li>Act 3
            <ul>
               <li>Act 3, Scene 1</li>
               <li>Act 3, Scene 2</li>
               <li>Act 3, Scene 3</li>
               <li>Act 3, Scene 4</li>
            </ul>
         </li>
         <li>Act 4
            <ul>
               <li>Act 4, Scene 1</li>
               <li>Act 4, Scene 2</li>
               <li>Act 4, Scene 3</li>
               <li>Act 4, Scene 4</li>
               <li>Act 4, Scene 5</li>
               <li>Act 4, Scene 6</li>
               <li>Act 4, Scene 7</li>
            </ul>
         </li>
         <li>Act 5
            <ul>
               <li>Act 5, Scene 1</li>
               <li>Act 5, Scene 2</li>
            </ul>
         </li>
      </ul>
   </body>
</html>
```

The DOCTYPE declaration at the top (`<!DOCTYPE html SYSTEM "about:legacy-compat">`) is a schema declaration. It is similar to the declaration that <oXygen/> creates when you author a new XHTML document by hand (`<!DOCTYPE html>`), except that, for obscure reasons that aren't very interesting, it takes

a different form when you generate it as part of an XSLT transformation. You can think of it as a magic incantation, and we explain below how to invoke it.

We've used HTML unordered lists (`<ul>`) elements. The only content allowed inside a `<ul>` element is list items (`<li>`), and we've nested them, so that each each list item that represents an act contains the title of that act followed by an embedded `<ul>` that contains, in turn, a separate list item for the title of each scene. This isn't the only way to format this type of outline and you're welcome to take a different approach. For example, if you'd like to include the full text of the play, that is, the stage directions and speeches, the embedded list format isn't really appropriate. In that case, we would use the HTML header elements (`<h1>` through `<h6>`) to create hierarchical headers. You can read more about HTML lists and headers at http://www.w3schools.com.

Before you begin

Both your input document and your output documents have to be in the correct namespace, and you need to tell your XSLT stylesheet about that.

- **Input namespace:** Your input document, *Bad Hamlet*, is in the TEI namespace, which means that your XSLT stylesheet must include an instruction specifying that when it tries to match elements, it needs to match them in that namespace. You specify the namespace of the input document by creating a `@xpath-default-namespace` attribute on the `<xsl:stylesheet>` root element of your XSLT stylesheet and setting it equal to the `@xmlns` value of the input document. When you create a new XSLT document in <oXygen/> it won't contain that instruction, so you need to add it (in blue below).
- To ensure that the output will be in the HTML namespace, which is required for any HTML web page, you need to specify a default output namespace declaration (in fuchsia below).

Finally, to output the required DOCTYPE declaration, we also create an `<xsl:output>` element as the first child of our root `<xsl:stylesheet>` element (in green below). Our modified skeleton looks like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="3.0"
    xpath-default-namespace="http://www.tei-c.org/ns/1.0"
    xmlns="http://www.w3.org/1999/xhtml">
    <xsl:output method="xml" indent="yes" doctype-system="about:legacy-compat"/>

</xsl:stylesheet>
```

Guide to approaching the problem

Our XSLT transformation (after all this housekeeping) has three template rules:

1. We have a template rule for the document node (`<xsl:template match="/">`), in which we create the basic HTML output: the `<html>` element, `<head>` and its contents, and `<body>`. Inside the `<body>` element that we're creating, we use `<xsl:apply-templates>` and select the acts (using an XPath expression as the value of the `@select` attribute).
2. We have a separate template rule that matches acts, so it will be invoked as a result of the preceding `<xsl:apply-templates>` instruction, and will fire once for each act. Inside that template rule we create a new list item (`<li>`) for the act being processed and inside the tags for that new list item we do two things. First, we apply templates to the `<head>` for the act, which will eventually cause its title to be output. Second, we create wrapper `<ul>` tags for the nested list that will contain the titles of the scenes. Inside that new `<ul>` element, we use an `<xsl:apply-templates>` rule to apply templates to (that is, to process) the scenes of that act.

3. We have a separate template rule that matches scenes, and that just applies templates to the `<head>` element in each scene, which ultimately causes the textual content of the `<head>` element to be output. This rule will fire once for each scene in the play, and it will be called separately for the scenes of each act, so that the scenes will be rendered properly under their acts.

We don't need a template rule for the `<head>` elements themselves because the built-in (default) template rule in XSLT for an *element* that doesn't have an explicit, specified rule is just to apply templates to its children. The only child of the `<head>` elements is a `text()` node, and the built-in rule for `text()` nodes is to output them literally. In other words, if you apply templates to `<head>` and you don't have a template rule that matches that element, ultimately the transformation will just output the textual content of the head, that is, the title that you want.

Important

- Those who like to read ahead or already have some programming experience with other languages may have noticed that XSLT includes an `<xsl:for-each>` instruction that *could* be used to solve this problem. *We are prohibiting its use for now;* your solution must use `<xsl:template>` and `<xsl:apply-templates>` rules instead. There's a Good Reason for this, which we'll explain later, when we talk about situations where you *should* use `<xsl:for-each>`.
- *Before submitting your homework, you must run the transformation* to make sure the results are what you expect them to be. There's a guide to running XSLT transformations inside <oXygen/> at http://dh.obdurodon.org/oxygen-xslt-configuration.html. If you don't get the results you expect and can't figure out what you're doing wrong, you're welcome to post an inquiry to the discussion board. You can't just ask for the answer, though; you need to describe what you tried, what you expected, what you got, and what you think the problem is. We often find, just as we're preparing to post our own queries to programming discussion boards, that having to write up a description of the problem helps us think it through and solve it ourselves (the technical term for this phenomenon is rubber duck debugging). We're encouraging you to discuss the homework on the discussion boards because that's also helpful for the person who responds; we've found that answering someone else's inquiry and troubleshooting someone else's problem helps us clarify matters for ourselves.

<oo>→<dh> Digital humanities

Maintained by: David J. Birnbaum (djbpitt@gmail.com)
Last modified: 2018-03-12T15:40:38+0000

XSLT assignment #2

The input text

For this assignment we'll be using an XML file originally prepared by a member of this class in spring 2012. We've modified it for use as an XSLT exercise, and the new version is available at http://dh.obdurudon.org/skyrim.xml. You should right-click on this link, download the file, and open it in <oXygen/>. We'll be using it for subsequent XSLT assignments, so keep a copy when you're done with this one. You don't need the Relax NG schema, but if you'd like to look at it, it's available at http://dh.obdurodon.org/skyrim.rnc.

Because this document (unlike our version of *Hamlet*) is not in a namespace, you should not add the `@xpath-default-namespace` attribute.

Overview of the assignment

For this assignment you want to write an XSLT stylesheet that will transform the XML input document into an HTML document that consists entirely of tables of characters and factions. You can see the desired output at http://dh.obdurodon.org/skyrim-02.xhtml.

Analysis of the task

The information that you want to output is all found near the top of the input document, inside the `<cast>` element. For the moment we're going to ignore everything else, including the `<body>`. You want to generate one HTML table to contain information that you'll extract from the `<character>` elements and a second table to contain information that you'll extract from the `<faction>` elements. Your XSLT, then, should proceed along the following lines:

1. In a template rule for the document node (`<xsl:template match="/">`), create the HTML superstructure. Between the start and end `<body>` tags that you'll be creating you should insert the main tags for two HTML tables, one for characters and one for factions. Between the start and end `<table>` tags for each table, you should then insert `<xsl:apply-template>` elements that select the nodes you want to process to build those tables. Be careful, though; there may be `<character>` or `<faction>` elements elsewhere in the document, such as inside `<paragraph>` elements, and for the tables you're producing you want only the ones that are inside the `<cast>` element.
2. In XSLT, processing something normally happens in two parts. You normally have an `<xsl:apply-templates>` element that tells the system what elements (or other nodes) you want to process, and you then have an `<xsl:template>` element that tells the system exactly how you want to process those elements, that is, what you want to do with them. If you find the image helpful, you

can think of this as a situation where the `<xsl:apply-templates>` elements *throw some nodes out into space and say "would someone please process these?"* and the various `<xsl:template>` elements sit around watching nodes fly by, and when they match something, they *grab it and process it*. In this case, then, your `<xsl:apply-template>` elements inside the template rule for the document node will tell the system that you want to process `<character>` and `<faction>` elements, at which point the template rule for the document node will have done its work by announcing what needs to be done. That work actually gets done by other `<xsl:template>` rules, the ones that you've written that match the `<character>` and `<faction>` elements.

3. In the `<xsl:template>` rules for `<character>` or `<faction>` elements you'll need to output something for each one. That is, for each `<character>` or `<faction>` element, you'll need to output a line in your table.
4. The values you use to populate your table cells come from attributes. Remember that attributes are on the *attribute axis*, which you can address by prefixing the name of the attribute with an "at" sign. For example, open skyrim.xml in <oXygen/>, go into the XPath browser box in the upper left, and search for `//cast/character/@id`, and you'll retrieve all of the `@id` attribute values associated with `<character>` elements. You won't use this exact XPath in your assignment, but it can serve to remind you how you address an attribute in XPath.

What goes where

You'll want to create three template rules, one for the document node (`/`), one for each type of element you need to process.

In the template for the document node you create the HTML superstructure, as well as the wrappers for the tables (that is, the `<table>` elements themselves). Then, inside the `<table>` elements you'll create the header rows, with the labels for the columns in each table, and after the header rows you'll need to use an `<xsl:apply-templates>` element and select the data you want to use to populate that table. For the character table, the data will be the characters, and for the faction table, it will be the factions.

The actual rows in the tables for each of the characters and factions should be created in the template rules that you'll write for those elements. Note that the stuff you do just once per table (create the table and the header row) should be in the template rule for the document node, but the stuff that has to happen repeatedly, once for each row of data, should be in the template rule for those elements (`<character>` and `<faction>`). That's the XSLT way to ensure that you create a new row for every `<character>` or `<faction>` element in the input.

How to develop an XSLT stylesheet

The problem with trying to code everything at once and then trying to run it is that if it doesn't work, it can sometimes be hard to find just where the error is. When we develop an XSLT stylesheet, we begin by writing a template rule for the document node (`/`), and inside that we do basic housekeeping (e.g., HTML superstructure) and include whatever `<xsl:apply-templates>` elements we need. Initially for the templates that get called by those `<xsl:apply-templates>` elements we put in a simplified placeholder, something that will produce output that may not be what we want eventually, but that will let us confirm that our templates are being called. Once the basic framework is in place (we're calling the right templates in the right places), we then start fine-tuning the individual template rules, replacing the placeholder code with code that produces the results we actually want. The technical term for this type of placeholder is *stub*.

In this case, in the template rules for characters and faction we might start by just outputting some plain text. That won't be valid in HTML, but it will tell you whether the templates are being called when you

want. Once that's working, you can expand it by creating real HTML rows and cells and filling each one with fixed text of some sort. That's now valid HTML, but it isn't the real content. Once you've determined that the cells are being created in the right place, you can then replace that fixed text with XSLT code that retrieves the information you actually want in your table. It may be tempting to write all of the code at once, but typically it won't all be correct the first time, and you'll save time in the long run by proceeding one step at a time. That isn't just for beginners; it's normal professional practice, and it's what we do in our own development, as well.

<oo>→<dh> Digital humanities

Maintained by: David J. Birnbaum (djbpitt@gmail.com)
Last modified: 2018-03-12T15:41:00+0000

XSLT assignment #3

The input text

For this assignment we'll continue to use http://dh.obdurudon.org/skyrim.xml. You should right-click on this link, download the file, and open it in <oXygen/>. You don't need the Relax NG schema, but if you'd like to look at it, it's available at http://dh.obdurodon.org/skyrim.rnc.

Overview of the assignment

For this assignment we're going to work with the `<body>` element, concentrating on processing the in-line elements to style the text. You can use some of the basic HTML in-line elements, like `<em>` for emphasis or `<strong>` for strong emphasis, but you'll also want to use CSS to set some elements to different colors or background colors or borders or fonts or font sizes or font styles (e.g., italic) or font weights (e.g., bold) or text decoration (e.g., underlining) or text transformation (e.g., convert to all upper case) … well … anything else. We describe below how to do that.

There are six types of in-line elements in the input XML document:

- `<QuestEvent>`
- `<QuestItem>`
- `<character>`
- `<epithet>`
- `<faction>`
- `<location>`

Some are immediately inside a `<paragraph>` and some are inside other elements that are inside paragraphs. You may not know at the outset which ones can be inside which other ones, or how deeply they can nest. Happily, with XSLT, unlike with many other programming languages, you don't need to care about those questions!

How to process richly mixed content

Prose paragraphs with in-line elements that might contain other in-line elements are richly mixed content, with varied and unpredictable combinations of elements and plain text. This is the problem that XSLT was designed to solve. With a traditional procedural programming language, you'd have to write rules like "inside this paragraph, if there's a `<QuestEvent>` do X, and, oh, by the way, check whether there's a `<QuestItem>` or a `<location>` inside the `<QuestEvent>`, etc." That is, most programming languages have to tell you what to look for at every step. The elegance of XSLT when dealing with this type of data is that all you have to say inside paragraphs and other elements is "I'm not worried about what I'll find here; just process (apply templates to) all my children, whatever they might be."

The way to deal with mixed content in XSLT is to have a template rule for every element and use it to output whatever HTML markup you want for that element and then, inside that markup, to include a general `<xsl:apply-templates/>`, not specifying a `@select` attribute. For example, if you want your `<QuestEvent>` to be tagged with the HTML `<strong>` tags, which means "strong emphasis" and which is usually rendered in bold, you could have a template rule like:

```
<xsl:template match="QuestEvent">
  <strong>
      <xsl:apply-templates/>
  </strong>
</xsl:template>
```

You don't know or care whether `<QuestEvent>` has any child nodes or, if it does, what they are. Whatever they are, this rule tells the system to try to process them, and as long as there's a template rule for them, they'll get taken care of properly somewhere else in the stylesheet. If there are no child nodes, the `<xsl:apply-templates/>` will apply vacuously and harmlessly. As long as every element tells you to process its children, you'll work your way down through the hierarchy of the paragraph without having to know which elements can contain which other elements or text nodes.

Taking stock: when to use `@select`

In an earlier XSLT assignment, where you built HTML tables of characters and factions, you used `<xsl:apply-templates select="…"/>`, specifying exactly what you wanted to process where. That makes sense when your input (the `<character>` and `<faction>` elements inside the `<cast>` element at the beginning of the document) and output (an HTML table) are very regular in structure. *Use the `@select` attribute when you know exactly what you're looking for and where you want to put it.*

In this assignment, on the other hand, you don't know (and don't need to know) the order and nesting hierarchy of whatever salad of elements and plain text you might find inside a paragraph or its subelements. You just want to process whatever comes up whenever it comes up. `<xsl:apply-templates/>` without the `@select` attribute says "apply templates to whatever you find." *Omit the `@select` attribute where you don't want to have to think about and cater to every alternative individually.* (You can still treat them all differently because you'll have different template rules to "catch" them, but when you assert that they should be processed, you don't have to know what they actually are.)

What should the output look like

HTML provides a limited number of elements for styling in-line text, which you can read about at http://www.w3schools.com/html/html_formatting.asp. You can use any of these in your output, but note that presentational elements, the kind that describe how text looks (e.g., `<i>` for "italic"), are generally regarded as less useful than descriptive tags, which describe what text means (e.g., `<em>` for "emphasis"). Both of the preceding are normally rendered in italics in the browser, but the semantic tag is more consistent with the spirit of XML than the presentational one.

The web would be a dull world if the only styling available were the handful of presentational tags available in vanilla HTML. In addition to those options, there are also ways to assign arbitrary style to a snippet of in-line text, changing fonts or colors or other features in mid-stream. To do that:

1. Before you read any further in this page, read our Using `<span>` and `@class` to style your HTML page.

2. To use the strategies described at that page, create an XSLT template rule that transforms the element you want to style to an HTML `<span>` element with a `@class` attribute. For example, you might transform `<faction ref="MythicDawn">assassins</faction>` in the input XML to `<span class="MythicDawn">assassins</span>` in the output HTML. You can then specify CSS styling by reference to the `@class` attribute, as described in the page we link to above.
    - Note that you can make your transformations very specific. For example, instead of setting all `<faction>` elements to the same HTML `@class`, you can create separate template rules to match on factions according to their attribute values. For example, `<xsl:template match="faction[@ref='MythicDawn']">` is a normal XPath expression to match `<faction>` elements only if they have a `@ref` attribute with the value "MythicDawn".
    - If you really want to exercise your XPath skills, note that in the header some factions are described (with an `@alignment` attribute) as "evil", "good", or "neutral". You can write a matching rule that will *dereference* the `@ref` attribute on, say, `<faction ref="MythicDawn">assassins</faction>`, look up whether this is an evil, good, or neutral faction, and set the `@class` value accordingly. You could make all good factions one color and all evil factions a different color, letting XPath look up the moral alignment of a faction for you.
3. Setting the `@class` attributes in the output HTML makes it possible to style the various `<span>` elements differently according to the value of those attributes, but you need to create a CSS stylesheet to do that. Create the stylesheet (just as you've created CSS in the past), and specify how you want to style your `<span>` elements. Link the CSS stylesheet to the HTML you are outputting by creating the appropriate `<link>` element in your output HTML (you can remind yourself how to do that at the bottom of our Introduction to CSS).

When the smoke clears

What you should produce, then, is:

- An XSLT stylesheet that transforms the `<body>` element and its contents into HTML.
- The resulting HTML should style the six types of in-line elements listed above. At least some of those styles should be set using `<span>` elements with the `@class` attribute.
- You need to create a CSS file, linked to your output HTML, that specifies how to style the output document. You can look up the most useful of the available CSS properties at http://www.w3schools.com/css/. We'd suggesting following the links on the left under "CSS styling" for styling backgrounds, text, and fonts, as well as the link for borders under "CSS box model".

Please upload your XSLT, HTML, and CSS files to CourseWeb.

<oo>→<dh> Digital humanities

Maintained by: David J. Birnbaum (djbpitt@gmail.com)
Last modified: 2016-10-20T21:24:33+0000

XSLT assignment #4

The input text

For this assignment you will be working with Shakespearean sonnets, which you can download from http://dh.obdurodon.org/shakespeare-sonnets.xml. You should right-click on this link, download the file, and open it in <oXygen/>.

Using modal XSLT

What happens if you need to process the same nodes in your document in two different ways? For example, what happens if you need to output them as list items in a table of contents, but also as headers or text in the main body of your document, below the table of contents? Wouldn't it be handy to be able to have two completely different template rules that match exactly the same elements, one rule to output the data as list items in the table of contents and the other to output the same data as headers? You can write two template rules that will match the same nodes (have the same value for their `@match` attribute), but how do you make sure that the correct one is handling the data in the correct place?

For this assignment we would like you to get some experience working with modal XSLT. As is explained at http://dh.obdurodon.org/modal-xslt.html, modal XSLT allows you to output the same parts of the input XML document in multiple locations and treat them differently each time. That is, it lets you have two different template rules for processing the same elements or other nodes in different ways, and you use the `@mode` attribute to control how the elements are processed *at a particular place* in the transformation.

Overview of the assignment

For this assignment you want to produce an HTML version of the sonnets with a table of contents at the top. The table of contents should have one entry for each sonnet, which gives the number of the sonnet and the first line. Below the full table of contents (one line for each sonnet) you should render the complete text of all of the sonnets. You can see our output at http://dh.obdurodon.org/shakespeare-sonnets.xhtml.

How to begin

Begin by forgetting about the table of contents, and concentrate on just outputting the full text of the sonnets. This is just like the XML-to-HTML transformations you have already written, and you'll use regular template rules (without a `@mode` attribute) to perform the transformation. In our HTML output (scroll down past the table of contents, to where the full text of the sonnets is rendered), the roman numeral before each sonnet is an HTML `<h2>` element and the body of each sonnet is an HTML `<p>`

element. To make each line of the poems start on a new line, we add an HTML empty `<br/>` ("[line] break") element at the end of each line within the stanza. If you don't include the `<br/>` elements, the lines will all wrap together in the browser. Here's the HTML output for one of our sonnets:

```
<h2>VI</h2>
<p>Then let not winter's ragged hand deface,<br/>
    In thee thy summer, ere thou be distill'd:<br/>
    With beauty's treasure ere it be self-kill'd.<br/>
    Make sweet some vial; treasure thou some place<br/>
    That use is not forbidden usury,<br/>
    Which happies those that pay the willing loan;<br/>
    That's for thy self to breed another thee,<br/>
    Or ten times happier, be it ten for one;<br/>
    Ten times thy self were happier than thou art,<br/>
    If ten of thine ten times refigur'd thee:<br/>
    Then what could death do if thou shouldst depart,<br/>
    Leaving thee living in posterity?<br/>
    Be not self-will'd, for thou art much too fair<br/>
    To be death's conquest and make worms thine heir.
</p>
```

The fine print: Don't worry if your HTML output isn't wrapped the same way ours is, if it puts the empty line break elements at the beginnings of lines instead of at the ends, or if it serializes (spells out) those empty line break elements as `<br></br>` instead of as `<br/>`. Those differences are not *informational* in an XML context. You can open your HTML output in <oXygen/> and pretty-print it if you'd like, which may make it easier to read, but as long as what you're producing is valid HTML and renders the text appropriately, you don't have to worry about non-informational differences between your markup and ours.

More fine print: You need a line break only between lines, which is to say that you don't need a `<br/>` element at the end of the last line of the sonnet because that's the end of the containing `<p>`, and not between lines. In our solution we used an `<xsl:if>` element to check the position of the line and output the `<br/>` only for non-final lines. If you're feeling ambitious, you can look up `<xsl:if>` at http://www.w3schools.com/xml/xsl_if.asp or in Michael Kay and perform this check yourself. If not, you can just output the `<br/>` element after all lines of the sonnet, including the last. That's not really considered good HTML style, and you don't want to do it in your own projects, but it won't interfere with the legibility in the browser and we'll let it pass for homework purposes.

Once your sonnets are all being formatted correctly in HTML, you can add the functionality to create the table of contents at the top.

Adding the table of contents

The template rule for the document node in our solution, revised to output a table of contents before the text of the sonnets, looks like the following:

```
<xsl:template match="/">
    <html>
        <head>
            <title>Shakespearean sonnets</title>
        </head>
        <body>
            <h1>Shakespearean sonnets</h1>
            <h2>Contents</h2>
            <ul>
                <xsl:apply-templates select="//sonnet" mode="toc"/>
            </ul>
            <hr/>
```

```
            <xsl:apply-templates/>
        </body>
    </html>
</xsl:template>
```

The highlighted code is what we added to include a table of contents, and the important line is `<xsl:apply-templates select="//sonnet" mode="toc"/>`. This is going to apply templates to each sonnet *with the `@mode` attribute value set to "toc".* The value of the `@mode` attribute is up to you (we used "toc" for "table of contents"), but whatever you call it, setting the `@mode` to any value means that only template rules that also specify a `@mode` with that same value will fire in response to this `<xsl:apply-templates>` element. Now we have to go write those template rules!

What this means is that when you process the `<sonnet>` elements to output the full poems, you use `<xsl:apply-templates>` and `<xsl:template>` elements without any `@mode` attribute. To create the table of contents, though, you can have `<xsl:apply-templates>` and `<xsl:template>` elements that select or match the same elements, but that specify a mode and apply completely different rules. A template rule for `<sonnet>` elements in table-of-contents mode will start with `<xsl:template match="sonnet" mode="toc">`, and you need to tell it to create an `<li>` element that contains a roman numeral and a first line, both fetched from the sonnet in the input XML file. The rule for those same elements not in any mode will start with `<xsl:template match="sonnet">` (without the `@mode` attribute). That rule will create the `<h2>` header to hold the roman numeral and then output the full text of the poem in a `<p>`, with `<br/>` elements between the lines. In this way, you can have two sets of rules for sonnets, one for the table of contents and one for the body, and use modes to ensure that each is used only in the correct place.

Remember: both the `<xsl:apply-templates>`, which tells the system to process certain nodes, and the `<xsl:template>` that responds to that call and does the processing must agree on their mode values. For the main output of the full text of every poem, neither the `<xsl:apply-templates>` nor the `<xsl:template>` elements specifies a mode. To output the table of contents, both specify the same mode.

<oo>→<dh> Digital humanities

Maintained by: David J. Birnbaum (djbpitt@gmail.com)
CC BY NC SA
Last modified: 2017-03-01T03:29:03+0000

XSLT assignment #5

The input text

For this assignment you will be working with Shakespearean sonnets, which you can download from http://dh.obdurodon.org/shakespeare-sonnets.xml. You should right-click on this link, download the file, and open it in <oXygen/>. You will be building on our XSLT assignment #4, and you can take your stylesheet from that assignment and modify it for this one.

Overview of the assignment

For your last assignment you used the XSLT `@mode` attribute to create a table of contents for the Shakespearean sonnets, using the first line of each sonnet as a surrogate for the title (since they don't have real titles). Our output is at http://dh.obdurodon.org/shakespeare-sonnets.xhtml.

What's a table of contents good for anyway

In a digital edition, we can just do a full-text search and scroll in the browser, so we don't really need a table of contents at all. We can search for a roman numeral, we can search for the text of the first line of a sonnet, or we can search for a memorable phrase. But suppose we want to produce a paper edition, where the only organized access our users will get is the organization we decide to give them. What would be a useful table of contents or index?

A table of contents in the same order as the full text (numerical order), which is what we produced in the last assignment, duplicates the ordering information in the plain text. How useful is that? If we want to find a sonnet with a low number, we already know without a table of contents that we should look near the beginning. On the other hand, it's very common in published poetry collections to include an index of first lines, sorted in alphabetical order, so that a user who remembers just the first line of a poem can find it easily.

For this assignment we're going to enhance our output from the last assignment in the following ways:

- We're going to create links between the items in the table of contents and the sonnets, so that you can click on a line and be taken immediately to the corresponding sonnet.
- We're going to alphabetize our list of first lines, so that the table of contents will be sorted alphabetically, instead of in numerical order.
- As long as we were sorting the lines by order of appearance in the collection, that is, by numerical order of sonnet, it made sense to put the sonnet number (the roman numeral) first. For example, the sixth entry in our original list read "VI. Then let not winter's ragged hand deface,". If we're now going to sort by the first line of text, though, having those roman numerals at the far left edge of the entry will be disorienting, since they'll obscure the fact that we're using alphabetical order. We do want to retain the roman numerals (after all, the sonnet numbers are meaningful for Shakespeare scholars), but we're

going to move them to the *end* of the line, so that when the user reads down the left edge of the table of contents, scanning for a particular first line, the alphabetic order will be immediately accessible.

Our HTML output is at http://dh.obdurodon.org/shakespeare-sonnets-sorted.xhtml.

The tools we need

To create links between the first lines in the table of contents and the sonnets in the full text section of the page below we're going to use *attribute value templates* (AVT). If you haven't done so already, you should read about AVTs at http://dh.obdurodon.org/avt.xhtml.

To sort the table of contents we're going to use `<xsl:sort>`.

When we sort the first lines, they won't sort correctly for a quirky reason. We're going to fix that using the XPath `translate()` function, which we discuss below.

How HTML linking works

The `<li>` items in the table of contents should include `<a>` ("anchor") elements, which is how HTML identifies a clickable link. An anchor that is a clickable link has an `@href` attribute, which points to the target to which you want to move when you click on the link. For example, the table of contents might contain the following list item for Sonnet VI:

```
<li><a href="#sonnetVI">Then let not winter's ragged hand deface, (VI)</a></li>
```

HTML `<a>` elements that have `@href` attributes normally appear blue and underlined in the browser, to advertise that they are links. The *target* of a link can be any element that has an `@id` attribute that identifies it uniquely. If you click on this line in the browser, the window will scroll to the element elsewhere in the document that has an `@id` attribute with the value "sonnetVI". In our case, we've assigned that `@id` attribute value to the `<h2>` for that sonnet in the main body:

```
<h2 id="sonnetVI">VI</h2>
```

Adding links to your output

You should first read our page on Attribute value templates (AVT), which describes a strategy you can use to create a unique `@id` attribute for each sonnet. For this task we gave our sonnets `@id` values that were a concatenation of the string "sonnet" and the roman numeral of the sonnet, e.g., "sonnetVI" for Sonnet #6. We attached those `@id` attributes to the `<h2>` elements that we used as titles for each sonnet in the body of our page, e.g., `<h2 id="sonnetVI">`. Meanwhile, in the table of contents at the top we created `<a>` elements with `@href` attributes that point to these `@id` values. *The value of the `@href` attribute must begin with a leading "#" character, but that "#" must not be part of the value of the `@id` attribute to which it points.* For example,

```
<li><a href="#sonnetVI">Then let not winter's ragged hand deface, (VI)</a></li>
```

means if the user clicks on this line, the browser will scroll to the line that reads `<h2 id="sonnetVI">` in the main body of the page. *Remember: the value of the `@href` attribute begins with "#", but the value of the*

corresponding `@id` attribute on the `<h2>` element you want to scroll to doesn't.

Armed with that information, you can take your answer to the main assignment and, using AVTs, modify it to create the `<a>` elements with the `@href` attributes and the `@id` attributes for the targets.

Sorting

An index of first lines in a collection of poems is usually alphabetized, because that's how humans look things up in that kind of list. To learn how to sort your table of contents before you output it, start by looking up `<xsl:sort>` at https://www.w3schools.com/xml/xsl_sort.asp or in Michael Kay. So far, if we've wanted to output, say, our table of contents in the order in which they occur in the document, we've used a self-closing empty element to select them with something like `<xsl:apply-templates select="//sonnet"/>`. We've also said, though, that the self-closing empty element tag is informationally identical to writing the start and end tags separately with nothing between them, that is, `<xsl:apply-templates select="//sonnet></xsl:apply-templates>`. To cause the elements being processed to be sorted first, you need to use this alternative notation, with separate start and end tags, because you need to put the `<xsl:sort>` element between the start and end tags. If you use the first notation, the one with a single self-closing tag, there's no "between" in which to put the `<xsl:sort>` element. In other words, you want something like:

```
<xsl:apply-templates select="//sonnet">
  <xsl:sort/>
</xsl:apply-templates/>
```

As written, the preceding will sort the `<sonnet>` elements alphabetically by their text value. As you'll see at the sites mentioned above, though, it's also possible to use the `@select` attribute on `<xsl:sort>` to sort a set of items by properties other than alphabetic order of their textual content.

After the sort

At this point we'd make other adjustments in the output. The original table of contents begins with a roman numeral, but if you're going to sort the table of contents, you want the text of the first line of the poem at the left side of the line, not preceded by the roman numeral, so that you can see the alphabetic order easily. Putting the roman numeral first would make it harder to discern the alphabetization, since the user wouldn't be able to see it by just glancing down the left margin. For that reason, you should now adjust the output to put the roman numeral after the text of the line, in parentheses.

Using `translate()` to fix the sort order

If you sort the first lines alphabetically according to their textual value, there will be one error. The first line of Sonnet #121, "'Tis better to be vile than vile esteem'd," will show up first because in the internal representation of characters in the computer, the single straight apostrophe is "alphabetically" earlier than all of the letters. We can fix this by using `translate()` to strip the apostrophe for sorting purposes, but not for rendering. That is, we can sort as if there were no apostrophe, while still printing the apostrophe when we render the line.

We can't easily translate away an apostropohe, though, because quotation marks have special meaning in XPath. For the purpose of this assignment, you can ignore this one missorted line. If you're feeling ambitious, though, read Michael Kay's answer at http://p2p.wrox.com/xslt/50152-how-do-you-translate-apostrophe.html and see whether you can apply it to fixing this problem.

Finishing touches

Some lists of first lines of poetry put quotation marks around the lines. We haven't done that in our solution, but if you'd like to add it, you should use the HTML `<q>` ("quoted text") element, instead of outputting the raw quotation marks as plain text.

Oh, and did we mention CSS? Can you attach a CSS stylesheet to your output to make it look more interesting than what you get by default in a web browser?

<oo>→<dh> Digital humanities

Maintained by: David J. Birnbaum (djbpitt@gmail.com)
Last modified: 2018-03-16T15:31:16+0000

XSLT assignment #6

The texts

Beginning in spring 2014 (2144), XML texts prepared by students in this course for their projects have been made available on GitHub (unless prohibited by copyright). You can find them by loading our Course projects page and clicking on the bracketed links labeled "GitHub" after the project titles. Because the XML texts for current (and some older) projects are under development, they may be inconsistently or incompletely marked up. As long as they are well formed, however, they can be explored with XML tools, including XSLT.

The assignment

Select an XML file from one of the projects and spend a few minutes looking at it to familiarize yourself with its overall structure. (Notice whether it's in a namespace!) Explore the project GitHub repo and site to learn more about it. You may use your own XML files or someone else's. Download an XML file from whichever project you choose and open it in <oXygen/>.

Digression: Downloading files from GitHub

The easiest way to download a file from GitHub is to clone the project onto your own machine, which will copy all files, and then open the file you need. If you want to download just one file (which is all you need for this assignment), *you can't just right-click and download* because you'll download a version with extraneous GitHub specific markup mixed into the file, which will render the file unusable for your purposes. What you can do instead is 1) connect to the repo in a browser; 2) click on the file you want, which will display its contents; 3) click on the button labeled "Raw" at the top of the code window, which will display its contents without any extraneous GitHub-specific material; and 4) either right-click and do a "Save as" or select all the text in the window, copy it to the clipboard, and paste it into a new XML document in <oXygen/>.

What to do with the file once you've downloaded it

Transform the XML into some form of HTML using XSLT, whether that's a reading view or some sorts of lists or tables or other reports. You should decide yourself on the type of output you would find interesting or useful, but so that you'll gain practice with some of the techniques we've introduced recently, your transformation must require meaningful use of at least two of the following:

- Conditional: an `<xsl:if>` and/or `<xsl:choose>` statement(s) (see XSLT, part 2: conditionals and push and pull)
- An `<xsl:for-each>` statement (see XSLT, part 2: conditionals and push and pull)

- Attribute value templates (see Attribute value templates (AVT))
- Modal XSLT (see Modal XSLT)
- Linking (e.g., as in XSLT assignment #5)

Please upload your XSLT and the XML to CourseWeb. You do not have to upload the HTML output of your transformation (we're going to run the transformation and generate the HTML ourselves anyway). If your HTML is going to be styled with CSS, though, be sure that your XSLT generates the necessary <link> element inside HTML document, and upload the CSS file along with the XSLT.

<oo>→<dh> Digital humanities

Maintained by: David J. Birnbaum (djbpitt@gmail.com)
Last modified: 2018-03-23T21:40:13+0000

Test #5: XSLT

The task

For this test, we are revisiting the play *The bicyclers and three other farces* from the Regex test, this time to use XSLT to transform an XML version of the entire text into a reading view in HTML. The input XML is available at http://dh.obdurodon.org./xslt-test_input.xml. One possible HTML output file is available at http://dh.obdurodon.org/xslt-test_output.xhtml, but yours does not have to look exactly like ours.

Before you start

Before writing any XSLT, you should explore the XML file. Part of that exploration involves reading through it quickly, but you can also use the XPath browser interface in <oXygen/> to learn about the markup. What elements are used in the document? How are they structured, that is, which elements can occur where? To get you started, you can get the names of all of the distinct element types with the XPath expression `distinct-values(//*/name())`. This finds all elements anywhere in the document (`//*`); uses the `name()` function to get their names, instead of the elements themselves; and then uses the `distinct-values()` function to remove the duplicates and make the list easier to read. Before you write any XSLT, be confident that you know how each of these element types is used in the document. Hint: the `<stage>` element appears in two different contexts (you can find them using XPath), and you may want to process it differently according to where it appears.

HTML requirements

Your HTML must look like the script for a play. That gives you a lot of flexibility, since the typography of play scripts is not rigid, but it should be something that a reader would recognize as a script, and the expected parts of the script (such as scenes, speakers, spoken text, stage directions) should be recognizable.

Your HTML must be valid. This means that you should save the results of the transformation by specifying a filename in the "Output" dropdown in <oXygen/>, and, after running the transformation, you should open the output HTML in <oXygen/> and validate it. You are not required to apply CSS to the file, but if you do (it's an optional bonus task, about which see below), the CSS needs to be valid, it needs to be linked to the HTML, and the link needs to be created during the transformation.

Required output features

Your output must include the following:

- Titles for the play and for the four constituent scenes.

- Cast of characters for each scene
- Speeches should be recognizable as speeches, and the speakers should be represented in a way that makes sense in a script.
- Stage directions should be recognizable as stage directions.

You should create this output by using XSLT in an algorithmic fashion, as described in our http://dh.obdurodon.org/algorithms.xhtml.

Successful completion of everything above this line earns an “A” grade.

Bonus tasks

The following features earn extra credit:

- Create a table of contents at the top of the file with links to the four scenes.
- Use attribute value templates where they improve your XSLT.
- Use appropriate XPath functions where they improve your XSLT.
- Use conditional statements (`<xsl:if>` or `<xsl:choose>`) to streamline your XSLT.
- Use CSS to improve the appearance of the output.
- If you use CSS, use `@class` attributes in the HTML to assist with the CSS styling.
- Add comments to your XSLT to document your code.
- Use `<xsl:text>` where appropriate to manage plain text.

What to submit

Upload your XSLT file and, if you created CSS, your CSS stylesheet. Do not upload either the input XML (we already have it) or the output HTML (we’ll run your XSLT transformation to create it).

<oo>→<dh> Digital humanities

Maintained by: David J. Birnbaum (djbpitt@gmail.com)
Last modified: 2013-03-04T17:46:12+0000

Schematron assignment #1

In a three-way election for Best Stooge Ever, each candidate (Curly, Larry, Moe) wins between 0% and 100% of the votes. Assume that all votes are cast for one of the three candidates (no abstentions, write-ins, invalid ballots, etc.), which means that when you add the percentages for the three candidates, the result must be exactly 100%. Assume also that we're recording percentage of the vote, not raw votes, and that the percentages are all integer values. (In Real Life we'd probably record the raw count and *calculate* the percentages, but in real life we wouldn't be voting for Best Stooge Ever in the first place!) Here's a Relax NG schema for the results of the election:

```
start = results
results = element results { stooge+ }
stooge = element stooge { name, xsd:int }
name = attribute name { "Curly" | "Larry" | "Moe" }
```

Here's a sample XML document that is valid against the preceding schema:

```
<results>
    <stooge name="Curly">50</stooge>
    <stooge name="Larry">35</stooge>
    <stooge name="Moe">15</stooge>
</results>
```

We could have written a better Relax NG schema, but we didn't, and although our sloppy schema works with the results above, it also allows erroneous results like the following:

```
<results>
    <stooge name="Curly">55</stooge>
    <stooge name="Larry">38</stooge>
    <stooge name="Moe">11</stooge>
</results>
```

The problem here is that the three "percentage" values total 104%, and no matter how good our coding, it is not possible to prevent this type of error by using Relax NG alone. Your assignment is to write a Schematron schema that verifies that the three percentages always total exactly 100%. Test your results by creating the Relax NG schema, your Schematron schema, and a sample XML document that you can validate against both schemas in <oXygen/>. Enter correct and incorrect values and verify that the Schematron schema is working correctly. For homework, upload only your Schematron schema.

You can stop here and consider the assignment complete, but for more Schematron practice, you're welcome to add additional rules to check for additional types of error. The following types of errors could have been controlled by writing a better Relax NG schema, but for the purpose of learning Schematron, let's do it in Schematron:

1. There should be exactly three votes, with exactly one for each Stooge. No duplicate Stooges and no missing Stooges.
2. Each individual Stooge's vote should range from 0 to 100. No negative integers and no integers greater than 100. (The Relax NG schema is ensuring that all values are integers, so you don't have to worry about that.)

<oo>→<dh> Digital humanities

Maintained by: David J. Birnbaum (djbpitt@gmail.com)
Last modified: 2012-10-27T14:34:01+0000

Schematron assignment #2

Preamble

This assignment is situated in the context of Real Life linguistic documentation project in which we were asked to provide some XML assistance. Your assignment involves a bit of Schematron in the middle, but we describe below both the linguistic project itself and the eventual XML conversion that the Schematron was ultimately used to facilitate.

The problem

Here's a quote from http://dh.obdurodon.org/schematron-class-01.html (simplified slightly):

> Linguistic corpora often record transcriptions in multiple tiers, such as a transcription of the original utterance, a word-by-word gloss with grammatical information, and a more fluid, natural-language translation. The set of notational conventions most commonly used for this purpose by corpus linguists have been codified in the Leipzig Glossing Rules (http://www.eva.mpg.de/lingua/resources/glossing-rules.php). Here is a Russian example based on that document:
>
> | | | | | | | | |
> |---|---|---|---|---|---|---|---|
> | **Orth** | Мы | с | Марко | поеха-л-и | автобус-ом | в | Переделкино |
> | **Translit** | My | s | Marko | poexa-l-i | avtobus-om | v | Peredelkino. |
> | **ILG** | we | with | Marko | go-PST-PL | bus-by | to | Peredelkino. |
> | **Free** | 'Marko and I went to Perdelkino by bus.' | | | | | | |
>
> Other tiers might include International Phonetic Alphabet (IPA) and interlinear glossing or free translation into other languages.
>
> Each of the computationally tractable tiers should have the same number of words, and each word should have the same number of hyphens.

From field notes to markup

Field linguists often type up this information in plain text, so that their starting point is something like:

Orth: Мы с Марко поеха-л-и автобус-ом в Переделкино
Translit: My s Marko poexa-l-i avtobus-om v Peredelkino.
ILG: we with Marko go-PST-P bus-by to Peredelkino.
Free: Marko and I went to Perdelkino by bus.

Assume that you can get the raw text into the following XML easily:

```
<sentence>
    <orth>Мы с Марко поеха-л-и автобус-ом в Переделкино</orth>
    <translit>My s Marko poexa-l-i avtobus-om v Peredelkino.</translit>
    <ilg>we with Marko go-PST-P bus-by to Peredelkino.</ilg>
    <free>Marko and I went to Perdelkino by bus.</free>
</sentence>
```

You don't have to do the following for this assignment, but now that you've learned a bit about regular expressions, `<xsl:analyze-string>`, and the XPath `tokenize()` function, you would be able to write XSLT to convert this XML to a different XML structure, one where the pieces are aligned properly, that is, so that every word and morpheme on the Orth, Translit, and ILG (interlinear gloss) tier is associated with the corresponding word or morpheme on the other tiers (except the Free tier, which isn't expected to match up; it's a free translation, after all). But that works only if the person who entered the data originally got the spaces and hyphens right! If the number of spaces and hyphens doesn't match up in the Orth, Translit, Gram, and ILG tiers , you can't automate the alignment.

The task: using Schematron to get your data ready for XML-to-XML conversion

When we had to perform this type of plain-text-to-XML converstion for a real linguistic documentation project, the linguists' initial, raw field notes had lots of error: spaces instead of hyphens and vice versa, as well as other punctuation (periods, hash marks, etc.) in place of both spaces and hyphens. This is typical field data; it's very hard for a human to pay attention to counting spaces and punctuation marks, which is why we use markup languages in projects of this sort in the first place. Before we even tried to transform the data with XSLT to something that formalized the word-by-word and morpheme-by-morpheme alignment, we used Schematron to verify that the number of spaces and hyphens matched where it needed to. That doesn't mean that we can't still have a mistake, of course, but it greatly reduces the opportunity that we won't notice an error, since only if we were to make the same error (or the same type of error) in every associated tier would we fool the counter.

Your assignment, then, is to *write a Schematron schema* that will take input like:

```
<sentence>
    <orth>Мы с Марко поеха-л-и автобус-ом в Переделкино</orth>
    <translit>My s Marko poexa-l-i avtobus-om v Peredelkino.</translit>
    <ilg>we with Marko go-PST-P bus-by to Peredelkino.</ilg>
    <free>Marko and I went to Perdelkino by bus.</free>
</sentence>
```

and verify that the first three lines (Orth, Translit, and ILG) all have the same number of spaces and the same number of hyphens. *You do not have to convert this XML to word-aligned or morpheme-aligned XML;*

all you have to do is write the Schematron that will verify whether the spaces and hyphens match. The verification is a prerequisite for the transformation, which would be the next step in Real Life, but for a Schematron assignment all you have to do is … well … write the Schematron.

To test your Schematron rules, create your own small sample XML document, with a handful of sentences formatted like the example above, with each tier in its own element but no internal markup separating words or morphemes. You can make up your own examples in a language of your choice or copy examples from http://www.eva.mpg.de/lingua/resources/glossing-rules.php. If you make up your own examples, don't worry about the precision of your linguistic annotations; this is an exercise in Schematron, and not in field linguistics. It doesn't matter what tiers you use, as long as you have at least two that have spaces and hyphens in them that are supposed to correspond. You should also make copies of some of your examples, muck up the spaces and hyphenation, and use that bad data to test whether your Schematron schema can catch the errors.

If that's too easy

The following isn't required, but if you feel like exercising your XSLT and XPath skills, you're welcome to transform the input XML, which you've verified with Schematron, to a different XML structure, one that formalizes the word-by-word or morpheme-by-morpheme associations. There's no single right output XML structure (schema) for this purpose, so should you choose to try it, you should first decide what the XML output of the transformation should look like, and then write the XSLT to produce it.

newtFire {dhlds}
Maintained by: Elisa E. Beshero-Bondar (ebb8 at pitt.edu) (cc) BY-NC-SA Last modified: Sunday, 06-Aug-2017 18:16:13 EDT. Powered by firebellies.

Regex Exercise: Convert the text of a voyage narrative into XML

Consult the following resources as you work with Regular Expressions:

- Our newtFire tutorial on Autotagging with Regular Expressions (Regex)
- Regular-Expressions.info Tutorial: a mine of helpful detail on regular expression matching,

Your challenge is to up-convert to XML the complete plain text file of *A VOYAGE round the WORLD* by Georg Forster, using the Find and Replace window . This file is quite large, so autotagging using Regular Expressions (regex) is really the only option we have to make this text into an XML document. Begin by downlading the text file and opening it in <oXygen/>. Use the Find and Replace window in <oXygen/> to autotag the document, and consult our Guide to Autotagging with Regular Expressions and the Regular Expressions Quick Start tutorial as you work.

Record each step of your process carefully, in a separate plain-text file. This plain text file is what you will submit for your homework. Record step-by-step your global Find-and-Replace operations with Regular Expressions in oXygen. Your goal is to produce an XML document like our model XML file but even if you have have trouble, what is most important is that you document the steps you took.

Your XML markup should accomplish the following:

1. Indicate the structure of the file by marking book divisions, chapter divisions, and paragraphs. (You do not necessarily want to do this in that order! You might want to start "from the inside out", with the paragraphs first, and then work your way up.) Think about a strategy that makes sense to you to help you match the distinctive patterns that designate the structure of this document.
2. Tag the dates, at least the dates that are sitting in square brackets. Ideally, you should remove any pseudo-markup around them.

Your complete text should look like our model, only you could go one better by removing the pseudo-markup (the brackets) around the dates. Can you locate and tag more dates than those in the square brackets?

Upload two files on Courseweb for this exercise:

1. a plain-text file in which you recorded your steps, and
2. your end result: the XML file you have created.

newtFire {dhlds}
Maintained by: Elisa E. Beshero-Bondar (ebb8 at pitt.edu) (cc) BY-NC-SA Last modified: Tuesday, 03-Apr-2018 23:59:21 EDT. Powered by firebellies.

Regex Exercise (Short Test): Autotag the Radio Script of *The War of the Worlds*

- Our newtFire tutorial on Autotagging with Regular Expressions (Regex)
- Regular-Expressions.info Tutorial: a mine of helpful detail on regular expression matching,

The test

- For this test you need to download the War-of-the-World-1938.txt file from the Newtfire site.
- After you have the file downloaded and opened the file in oXygen, open the Find/Replace window.
- Open new text file to record your steps. **Record each step of your process on the following tasks carefully**, since this is the file we will be evaluating. These will include global Find-and-Replace operations or Regular Expressions in oXygen (using Ctrl-F on Windows or command-F on Mac). Your goal is to produce a well-formed XML document, but even if you have have trouble, what's most important is that you document the steps you took.
- We have already verified for you that there are no reserved characters.
- Also there are no groups of blank lines exceeding 2 (\n{2}).

Your Tasks:

1. Find all of the speakers. Use `<spkr>` in your replace window to wrap all of the speakers. Record your Find and Replace expressions with a brief description and any additional alterations you made to the text file.

   Bonus: Tag all of the speeches and corresponding speakers. Use `<sp>` for speech and `<spkr>` for speaker. Record your Find and Replace expressions with a brief description and any additional alterations you made to the text file. [Hint: remember how you wrapped chapters or acts!]

2. Find all of the stage directions in parenthesis. Tag all of the stage directions with `<sd>` removing the pseudo-markup (a.k.a. the parentheses). Record your Find and Replace expressions with a brief description and any additional alterations you made to the text file.
3. Make sure you add a root element and verify your new XML file is green in oXygen.
4. **Upload two files on Courseweb for this exercise:**
   1. a plain-text file in which you recorded your steps
   2. your end result: the XML file you create

newtFire {dhlds}
Maintained by: Elisa E. Beshero-Bondar (ebb8 at pitt.edu) (cc) BY-NC-SA Last modified: Sunday, 06-Aug-2017 18:16:42 EDT. Powered by firebellies.

XPath Exercise 1

First of all, download the XML file I have linked here: ForsterGeorgComplete.xml. Open the file in oXygen and work with the XPath 2.0 Window. Respond to the XPath questions below in a text file, and upload to Courseweb for this assignment when you're finished. (Please use an attachment! If you paste your answer into the text box, CourseWeb may munch the angle brackets.) Some of these tasks are thought-provoking, and even difficult. If you get stuck, do the best you can, and if you can't get a working answer, give the answers you tried and explain where they failed to get the results you wanted. Sometimes doing that will help you figure out what's wrong, and even when it doesn't, it will help us identify the difficult moments. These tasks require the use of path expressions and predicates, and there may be more than one possible answer. You may opt to try the functions count() and not(), but they should not require any other XPath functions. Consult our introductory guide *Follow the XPath!* for help with constructing your expressions.

With the Georg Forster file open in oXygen and using the XPath 2.0 browser window in oXygen, construct XPath expressions that will do the following. **Be sure to give the FULL XPath expression you used in your answer, and don't just report your results.** This way, if the answer is incorrect, we can help explain what went wrong.

1. Like most of the long voyage publications, Georg Forster's voyage account is produced in multiple books, and inside each books we find multiple chapters. Both books and chapters are coded with elements. Take a look at the outline view of the document before you begin to familiarize yourself with the structure of this file, and answer the following:
    - How can XPath tell apart the books from the chapters?
    - What XPath would find ONLY the books in the file?
    - What XPath would find ONLY the chapters in the file?
    - What XPath would find ONLY the chapters in Book 2?

2. Look at the outline structure of the document to help you with these: What's the XPath to identify the <head> element inside a chapter ? How would we locate a <l> (or line) element inside a chapter ?

3. Georg Forster used a lot of footnotes in his document: These are coded inside <ref> elements throughout the body paragraphs of the text. What's the XPath to locate all the notes in the document?

4. We've encoded lots of <placeName> elements in this document to mark names of places, and these may occur in lots of positions. Sometimes they're in the <head> elements that start the book or chapter divs, positioned inside lines of texts (coded with <l>). Most often they're nested in the body paragraphs (<p>), and they're frequently coded in Forsters notes, which you've just located.
    - What's the XPath to determine the number of placeNames that appear inside ONLY the <head> elements, and not in the rest of the document? (Notice where these are located in the heads).
    - What's the XPath to find the placeNames that are only mentioned in the notes?
    - What's the XPath to find the placeNames that are only mentioned in the notes of Book I? How many of these are there?

newtFire {dhlds}
Maintained by: Elisa E. Beshero-Bondar (ebb8 at pitt.edu) (cc) BY-NC-SA Last modified: Sunday, 06-Aug-2017 18:16:42 EDT. Powered by firebellies.

XPath Exercise 2

Again for this exercise, we'll be working with our XML of Georg Forster's voyage narrative. Download the XML file I have linked here: ForsterGeorgComplete.xml. Open the file in oXygen and work with the XPath 2.0 Window. Respond to the XPath questions below in a text file, and upload to Courseweb for this assignment when you're finished. (Please use an attachment! If you paste your answer into the text box, CourseWeb may munch your brackets.) Some of these tasks are thought-provoking, and even difficult. If you get stuck, do the best you can, and if you can't get a working answer, give the answers you tried and explain where they failed to get the results you wanted. Sometimes doing that will help you figure out what's wrong, and even when it doesn't, it will help us identify the difficult moments.

These tasks require the use of **path expressions, predicates, and the functions count(), not(), name(), and distinct-values()**, but they should not require any other XPath functions. There may be more than one possible answer. You may find class notes and our introductory guide *Follow the XPath!* a helpful resource as you proceed. With the Georg Forster file open in oXygen and using the XPath 2.0 browser window in oXygen, construct XPath expressions that will do the following. **Be sure to give the FULL XPath expression you used in your answer, and don't just report your results.** This way, if the answer is incorrect, we can help explain what went wrong.

Georg Forster often took note of when latitude and longitude coordinates were measured during his trip with Captain Cook. We've spent some time capturing and coding these (with help from regular expressions!), and now we can use XPath to work with our geographic coordinates. We've used <geo select="lat"> for latitude readings, and <geo select="lon"> for longitude readings.

1. Write an XPath expression to locate all the geo elements in Book I that contain latitude measurements. How many are there (only in Book I)? Check the number in the oXygen result window (Description line) if you like. Be careful if you use the count() function here that you're getting only the count in Book I.
2. These latitude measurements you've just looked up are all held inside paragraphs, or the <geo select="lat"> element. What would you add to the previous XPath expression to return the paragraphs that hold latitude measurements in Book I? Give your complete XPath expression here.
3. Write an XPath expression to find the first paragraph in Book III, Chapter 1 that contains a latitude reading. What's the number of this paragraph as coded in the file?
4. Write an XPath to bring up all the paragraphs in this WHOLE file that contain both latitude AND longitude readings. How many of these paragraphs are there?
5. Are there any paragraphs in this WHOLE file that do NOT have a latitude measurement, but DO have a longitude? What XPath expression reveals these? And how many of these paragraphs are there?

6. Explain why the following two XPath expressions return different results. Run each XPath expression, review the results, and explain what you think each expression is returning.
    - //p[geo]/placeName[1]
    - (//p[geo]/placeName)[1]

newtFire {dhlds}
Maintained by: Elisa E. Beshero-Bondar (ebb8 at pitt.edu) (cc) BY-NC-SA Last modified: Sunday, 06-Aug-2017 18:16:43 EDT. Powered by firebellies.

XPath Exercise 3

To begin, download this XML file from our Pacific Voyage project that I have linked here: ForsterGeorgComplete.xml. Open the file in <oXygen/> and work with the XPath 2.0 Window. Respond to the XPath questions below in a text file, and upload to Courseweb for this assignment when you're finished. (Please use an attachment! If you paste your answer into the text box, CourseWeb may munch your brackets.) Some of these tasks are thought-provoking, and even difficult. If you get stuck, do the best you can, and if you can't get a working answer, give the answers you tried and explain where they failed to get the results you wanted. Sometimes doing that will help you figure out what's wrong, and even when it doesn't, it will help us identify the difficult moments.

These tasks require the use of path expressions, predicates, and the functions count(), not(), name(), position(), last(), and distinct-values(), but they should not require any other XPath functions. There may be more than one possible answer. To read about these functions, you should consult *The XPath Functions We Use Most* page and if you have the Michael Kay text, it may be useful to you here. As always, consult our class notes and our introductory guide *Follow the XPath!*.

With the Georg Forster file open in oXygen and using the XPath 2.0 browser window in oXygen, construct XPath expressions that will do the following. **Be sure to give the FULL XPath expression you used in your answer, and don't just report your results.** This way, if the answer is incorrect, we can help explain what went wrong.
** Notation: For ease in recognition, from now on when we refer in discussion to an attribute name, we'll precede it with an at sign (@). In other words, when we write about the @type attribute below, the name of the attribute is actually type (without an at sign).

1. Working with @type:
    1. Write an XPath using a function that returns a count of number of times we've used @type attributes in the Georg Forster file.
    2. What's the XPath to return the parent elements (whatever they are) of @type?
    3. Modify the XPath in your previous statement to return in the bottom results window the **names** of those parent elements
    4. Modify the XPath expression once more to return a list of only the distinct-values of those parent elements.

2. Working with attributes of ANY kind:
    1. Write an XPath expression to return all the attributes of any kind, anywhere in this file.
    2. Using the name() function, build on your previous XPath expression to return the names of these attributes.
    3. Now, return only the distinct-values of those attribute names: What XPath expression does this?
    4. Now, what if we wanted to find all the **parent elements** (without knowing what they are) of any attributes in use within the **body** element of the file? Write the XPath expression.
    5. What's the XPath to return the distinct-values of the **names** of those parent elements in the body of the file.
    6. How many distinctly different element names are holding attributes of any kind? (What expression returns this as a count?)
3. Working with the count() and position() functions in predicates: (count(), position(), last()): (You'll need to look up how to look for a count() of something and set it equal to, greater than, or less than a particular number.)
    1. Write an XPath expression that returns the **last** paragraph in the ENTIRE Georg Forster file that contains **more than one** latitude record, coded as <geo select="lat">. (Hint: This builds on things we showed you in the XPath Exercise 2 homework. Note that you should only get ONE result here!)
    2. Modify the expression so you return the **first** paragraph in the ENTIRE Georg Forster file that contains **more than two** latitude records. (There's no such thing as a first function, but remember how we found the first, second, and third books and chapters in past XPath exercises? The same working with position numbers applies here.) Again, you should only get one result in your results window.
    3. Now, how would we write XPath to find **the very last paragraph in Book 2** that contains **more than 1 latitude record**? As before, you should only get one result for this.
    4. Now, can you write an XPath expression that finds **the very first** paragraph holding **more than two latitude** records, that also holds **more than one placeName element**?
    5. How would you modify the previous expression to return the contents of the placeName elements in that paragraph? What are the placeNames?
4. **Optional Bonus Challenge Question:** Try out an XPath function on this file that we haven't yet assigned. Look up functions in the links posted here in our instructions, or in the Michael Kay book if you have it. Explain what you tried, give your XPath expression, and describe its results in your return window.

newtFire {dhlds}
Authored by: Alexandra Krongel (alk157 at pitt.edu) Edited and maintained by: Elisa E. Beshero-Bondar (ebb8 at pitt.edu) BY-NC-SA Last modified: Tuesday, 10-Oct-2017 15:30:24 EDT. Powered by firebellies.

XPath Exercise 4

You can find an XML (TEI) version of our Fall 2015 DH class syllabus at http://newtfire.org/dh/dhCDA-2015.xml. Right-click to download and save this file locally on your computer, and open it in <oXygen/>.

You should consult *The XPath Functions We Use Most* page and especially its section III. on Strings. Also, if you have the Michael Kay text, it may be useful to you here. As always, consult our class notes and our introductory guide *Follow the XPath!*. After you've completed your homework, save your answers to a file and upload it to CourseWeb as an attachment. (Please use an attachment! If you paste your answer into the text box, CourseWeb may munch the angle brackets.) Some of these tasks are thought-provoking, and even difficult. If you get stuck, do the best you can, and if you can't get a working answer, give the answers you tried and explain where they failed to get the results you wanted. Sometimes doing that will help you figure out what's wrong, and even when it doesn't, it will help us identify the difficult moments. These tasks require the use of path expressions, predicates, and functions, and in this exercise we concentrate on functions that manipulate strings. There may be more than one possible answer.

Using the syllabus XML document and the XPath browser window in <oXygen/>, construct XPath expressions that will do the following (give the full XPath expressions in your answers, and not just the results):

1. There are two books referenced in the syllabus using the tag `<bibl>`. What Xpath will return a semicolon-separated list of the authors?
2.
    1. Which 'div' elements contain references to 'homework'? How many results do you return?
    2. Can you figure out how to retrieve the immediate parent element containing the word "homework"? (Hint: it involves looking for any element below the div and then its text() node, since we need the element whose *text* contains the word "homework.")
3.
    1. What XPath returns all the Fridays on the syllabus? (Scroll through the document looking for the date elements to help determine this.)
    2. Now, what if we want to return those dates in their ISO format, as yyyy-mm-dd? Can you retrieve that with XPath? (Hint: Look at the attribute values on the date elements.)
    3. Return a string-joined list of all these dates, separated with a comma and a space.
4.
    1. How many `div` elements of @type 'assign' contain references to word "GitHub"?
    2. Find the longest and shortest div elements of this type (that contain the word "GitHub") in the document. How long and short are they? Hint: You will need to use min(), max(), and the string-length() function here, as well as some complex predicates.

5. **Reformatting Dates:** In Question 3, you located the Fridays on the syllabus, most likely through XPath to reach element text contents. What if you only worked with the `@when` attribute values on `<date>`, without looking at the text element content? Could you determine the Fridays on the syllabus just from that content alone? Yes, you can, and though this may not seem practical since we already have days of the week noted in our document, consider this a challenge, *as if those M, W, and F designations were missing*, or as if you had to work with a list of numerical dates without knowing their days of the week. To retrieve this, you need an XPath function you probably have not seen before: `format-date()`, to use with `xs:date`. You can read about these in the Michael Kay book on pages 781-788, or on the W3C specifications page for XSLT functions, as well as in the Safari XSLT book online.

   The function `format-date()` can take a string of text that it identifies as a date and can be set to reformat that date in many different ways: It could give the name of the month, the year as the phrase "Two Thousand and Fifteen", and, yes, the day of the week to accompany a day, output in upper or lower case letters as you wish. It might also be converted and expressed as a Buddhist, Mohammedan, or Japanese calendar date (among many others).

   For this last task, do some reading on format-date in one or more of the resources we've linked here. Then try the following:

   1. Working **only** with the `@when` attribute on the date elements on our syllabus file, convert those dates in the return window so that they display days of the week (and anything else you wish from the various available date-formatting codes. To do this, you'll need to see how to work with `format-date()`. It works with at minimum, two arguments (though it can take up to five). At minimum it needs to contain 1) something that it understands **as a date**, and 2) a **picture string** to designate (or "picture" if you will) the output, using a special notation of letter codes inside square brackets, examples of which you can look up in the references we've mentioned and linked here. Here is a model of how `format-date` looks with the minimum two arguments, working on a string of text marked as a dot: `.` (to indicate the self:: axis—the current context node in an XPath expression):
      ```
      //some-XPath-here-that-leads-to-the-dates-you-want/format-
      date(xs:date(.), '[FNn]/ [MNn]/ [Dwo] / [YWw]')
      ```
      The first argument of format-date() takes the current XPath date (each date attribute on the syllabus) and recognizes it as a date via xs:date(.) Then, the second argument (after the comma) sits inside the single quotation marks. The values inside `[ ]`s or square brackets are the picture strings which designate FNn (day of the week--initially capitalized to lower case, then Month (same thing with the capitalization), then a day of the month spelled out as a word, and finally the year converted from a number to a string of words. We have positioned a `/` in between each picture string as a separator. Try appending the format-date() code from this example into your XPath that reaches into the `@when` attribute on date, plug it into your XPath window and notice what it outputs in the return window. And try tinkering with it to change it, using different picture strings. **Record at least two different ways you adjusted this code to output different formats of date that you tried here, and their output.**
   2. Now that you see how picture strings work in the `format-date()` function, we think you now know enough to take a numerical string of text from the `@when` attribute, convert it to retrieve days of the week, and then output **only the Friday dates on the syllabus**. We don't care how you decide to format the rest of the date, as that is up to you, but we would like you to write an XPath that first filters to find the dates you need, and then outputs those dates however you wish to output them. **Hints:** You will need to use a predicate, and you want to put the contains() or matches() function around the format-date() function, after the part of the XPath expression where you drill down for dates.

3. One last challenge: Modifying your functions, can you output **all of the Friday dates in October only**? Record the XPath. Hint: You probably want multiple predicates for this.

newtFire {dhlds}
Maintained by: Elisa E. Beshero-Bondar (ebb8 at pitt.edu) (cc) BY-NC-SA Last modified: Monday, 19-Feb-2018 17:57:25 EST. Powered by firebellies.

XQuery Exercise 2

Our second XQuery homework exercise works with our Digital Mitford project files, coded in TEI. Here's how to locate this collection in our eXist at the Pittsburgh Supercomputing Center: `collection('/db/mitford').`

IMPORTANT: You'll need to **declare the TEI namespace** at the top of any XQuery you run on these TEI files or you won't see any output. (Declare the namespace in the line just under `xquery version "3.1";`). Here's the line you need for that:

declare default element namespace "http://www.tei-c.org/ns/1.0";

Write XQuery expressions for each of the following tasks. Paste these into a text file or a document, and where we ask you to do so, record your return or output. Save your file using our standard homework filenaming conventions (as in `besheroXQuery2.txt`), and upload your assignment to Courseweb.

1. First, let's do some exploring. This is a large collection of files, so we might want to look at a complete list of their file names and get a count of the number of files. For this, we use the XPath function `base-uri()`. Write a statement in XQuery that returns the base-uri() (or filename) for each file in the Mitford collection. How many files are there? (Record the number.) Wrap the expression in a `count()` function so you return the number, and record the expression you used.
2. Starting from the `collection()`, write a basic XQuery expression to show you the coding of the files, using `/*`, so that you can see how to locate the `<title>` element inside the `<teiHeader>` and `<titleStmt>`. Copy this into your text file recording this homework exercise.
3. Begin working on FLWOR expressions. First, write a very simple FLWOR statement to define variables that will return the following:
    - the whole collection
    - the particular texts in the collection, starting from the `<body>` element in our TEI files. (You'll need this later.)
    - the main title of the files (as described in #2: up in the `<teiHeader>`, inside the `<titleStmt>`).
    - Write a return statement to return the text ONLY of the main titles of the files in this collection. Refer to what you learned in XQuery Exercise 1 about the differences between text() vs. string(). Which one of these should you use here and why? (Copy your FLWOR into your text file for this homework exercise, together with your explanation.)
    - How many titles did you return? (Record the number.) Note: You should actually return one extra title compared to the number of file names you returned, which surprised us as we worked with the collection, until we realized that one file here actually has *two* `<title>` elements inside its `<titleStmt>`.
    - **Bonus:** In Real Life we would write some more XQuery to figure out what is going on when we receive a puzzling result like this. To find out which file is the culprit, with two titles instead of one, we wrote a little more XQuery code, using a "for loop" and another variable to return the file in question. See if you can write the code to locate that file for a bonus on this assignment. Record your XQuery and give the two titles of the file.

4. Build on your FLWOR expression. We are looking for some very important personal contacts of Mitford whose names turn up in the `//body` of **more than 15 files** in the Digital Mitford collection. Note:
    - We don't want results from the TEI Header because that would include the current Mitford editors, so we want to define our variable with an XPath that drills into the `body` element of each file.
    - Each person is coded with a distinct identifier held in an @ref attribute: `<persName ref="#id">`, which helps us to keep track of people when they are referred to by different names in the archive.
    - Define a variable to collect a list of the distinct values of this `@ref` attribute. **Note:** To get output in XQuery when you collect a list of attribute values, you need to return the `string()` value of the attribute. (Check your results here. We returned 711 distinct values of the `@ref` attribute on the `<persName>` element across the collection.)
    - What we need to do next is something like building an index in a book. We have isolated the 711 distinct values of all the identifiers for people across the collection. We now need a way to check and see if a file contains a `<persName @ref>` that matches each single entry in that list of distinct values, since those entries are now just a simple *dereferenced* list, and pulled out of their XPath context. A good example of what we are doing is the index of a printed book: You go to the back of a book and look up a word or phrase that is only listed once, and find out all the different places in the body of the book that mention it, so you can flip back and find what you need. What we are doing here is similar: We want to define a new variable in XQuery to look up for us *which files are holding each entry* on our list of distinct values. We'll need to test each entry in our list of distinct values (using a "**for loop**") and *map it back into the XML tree* to locate the files holding it. You will need a predicate expression that uses a comparison operator (and we used a General Comparison operator, the equal sign = to filter *which files contain `<persName>` elements with `@ref` attributes* ***equal to*** *the current distinct value in the "for loop"*. Here is a review of comparison operators, which you used in the XPath assignments.
    - Now, use the `where` statement in the FLWOR to filter your results so that you return *only the distinct `@ref` attributes that are seen in more than 15 files*. **Note:** This make take about 15 seconds to run, so do not be alarmed if eXist seems to pause a little while. Our collection of files is pretty large and you are testing a list of 711 distinct values in your "for loop"! When the dust settles, you should return a list of 10 name ids, a "top 10" list of popular names in the Digital Mitford collection.
    - We'd like to return the output without the hashtag (#) in front, and we want to output the results in alphabetical order by the `@ref` (without hashtag). To eliminate the hashtag, we recommend either the `tokenize()` function we used in the previous assignment, or the `translate()` function (read about these in Michael Kay or look it up in The XPath Functions We Use the Most.
    - Reading our Explain XQuery Guide to look up the details, write the FLOWR statement to `order by` the distinct @ref that is most frequently referenced.
    - Add the appropriate word to the "order" statement to generate these results in reverse order. (Refer to our guide linked here.)

5. Finally, we will build an HTML file around your XQuery results using curly braces `{ }` where necessary. Consult our **XQuery Intro Guide on building HTML with Curly Braces.** We need to make some changes to our file, though, because we are working with two different namespaces now, HTML and TEI. Alter the top lines of the XQuery so they look like this:

```
            xquery version "3.1";
declare default element namespace "http://www.w3.org/1999/xhtml";
declare namespace tei="http://www.tei-c.org/ns/1.0";
```

   And begin building your HTML around your FLWOR so the basic outermost structure of the file is set:

```
   xquery version "3.1";
declare default element namespace "http://www.w3.org/1999/xhtml";
declare namespace tei="http://www.tei-c.org/ns/1.0";
  <html>
<head><title>Top Ten Most Referenced People in the Digital Mitford Project</title></head>
<body>


{
. . . FLWOR HERE. . .
[use the tei:prefix on any TEI element names here]
Return $something

}
</body>
</html>
```

6. Build an HTML table in the HTML body part of the file to contain table rows and two columns of cells (two cells side by side in each column).
    - The first cell will contain each of the top ten the translated and sorted `@ref` value (without hashtag) that you retrieved in number 4.
    - In the other cell, return a string-joined list of the `base-uri()` or filenames of each file that contains the match, separated by commas. You might just want to trim down that `base-uri()` so you only return the filename at the end (like we did in XQuery Exercise 1. Here is our HTML table output to view in a web browser. (View Page Source to look under the hood at the HTML code.)
7. **Bonus**: Instead of using `string-join()` to list out the multiple filenames in your second table cell, can you work out how to output that list of names in its own table, nested inside that second table cell? Inside the table cell, you will need to nest a new FLWOR statement inside a new set of curly braces, in which you'll make another `for` statement. Return your output in table rows with a single column of cells.

Copy your HTML and FLWOR constructions into your document to upload to Courseweb.

newtFire {dhlds}.
Maintained by: Elisa E. Beshero-Bondar (ebb8 at pitt.edu) BY-NC-SA Last modified: Monday, 19-Mar-2018 06:42:56 EDT.
Powered by firebellies.

XSLT Exercise 2

The input text

For this assignment we'll be producing HTML from a TEI XML file developed by the Akira project team on newtFire in the spring of 2018. The XML file is available here: http://newtfire.org/dh/Akira_tei.xml. You should right-click on this link, download the file, and open it in <oXygen/> (or you can pull it in locally from the DHClass-Hub where it is in Class Examples --> XSLT).

Housekeeping: Setting Up a TEI to HTML Transformation

When you create an new XSLT document in <oXygen/> it won't contain that instruction by default, so whenever you are working with TEI you need to add it (See the text in blue below). To ensure that the output would be in the XHTML namespace, we added a default namespace declaration (in purple below). To output the required DOCTYPE declaration, we also created `<xsl:output>` element as the first child of our root `<xsl:stylesheet>` element (in green below), and we needed to include an attribute there to omit the default XML declaration because if we output that XML line in our XHTML output, it will not produce valid HTML with the w3C and might produce quirky problems with rendering in various web browsers. So, you should copy our modified stylesheet template and xsl:output line here into your stylesheet:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xpath-default-namespace="http://www.tei-c.org/ns/1.0"
         xmlns:math="http://www.w3.org/2005/xpath-functions/math"
    exclude-result-prefixes="xs math"
    xmlns="http://www.w3.org/1999/xhtml"
         version="3.0">

         <xsl:output method="xhtml" encoding="utf-8" doctype-system="about:legacy-compat"
        omit-xml-declaration="yes"/>

</xsl:stylesheet>
```

Overview of the assignment

We're going to work with this entire XML document (on all levels of the hierarchy), concentrating on processing the XML "salad" of mixed text and in-line elements to style them for presentation on the web in HTML's limited tagset. You can use some of the basic HTML in-line elements, like `<em>` for emphasis or `<strong>` for strong emphasis, but you'll also want to use CSS to set some elements to have different colors or background colors or to alter borders or fonts or font sizes or font styles (e.g., italic) or font weights (e.g., bold) or add text decoration (e.g., underlining) or text transformation (e.g., convert to all upper case), really anything stylistically possible.

For this assignment, we aim to produce an HTML reading view of the Akira script, to help orient readers to the cast of characters and to help visualize some special markup the team has applied to help locate special scenes. The exercise will help orient you to styling and layout decisions connected with transforming XML to HTML, and it will also give you a chance to "remix" the XML creatively into something designed for display in a web browser.

Surveying the "input" TEI document, you will see it has a TEI header to hold information *about* the Akira document, its source, and the work of the project team. We will pick and choose which portions of this to process and output in the body of an HTML document (remembering that the `body` element in HTML is the part that is visible in a web browser), and we may want to change the order it appears in the new document. We will take some material from the TEI header to display in the HTML body, for example, and we will try sorting the list of characters that appears in the TEI `profileDesc` to present a title and cast list at the top of the HTML file.

The following portions of the input document are especially important to us to display in HTML:

- The `profileDesc` in the `teiHeader` contains a list of characters we will want to sort alphabetically. We will want to output make a cast list or key of names and abbreviations.
- We will want to process the script itself, with its `sp` elements and the information coded in the attributes, including the speaker id, the number, and, where available, the time segment.
- Within those `sp` elements we want to process the `l` elements to hold them on separate lines.
- When `sp` elements are inside special sections coded as `spGrp`, we want to hold these in their own HTML `div` elements to distinguish them from other parts of the script.

Some of these elements are located inside the `<teiHeader>`. Some are nested unevenly at different levels of the XML tree hierarchy, like some of the `<sp>` elements nested inside `<spGrp>` elements, and the `<l>` elements sitting inside of `<sp>` elements. You may not be sure at the outset which elements can be inside which other ones, or how deeply they can nest. Happily, with XSLT, unlike with many other programming languages, you don't need to care about those questions!

An example of possible desired output can be found here http://newtfire.org/dh/akiraSample.html, though we did not style the body paragraphs in this output file. It is important to note that the majority of the styling choices on this file are controlled with a CSS file. You will make your own CSS and relate it to your XSLT; therefore, your stylistic choices might vary greatly from ours and your output may look completely different. What should look relatively similar is the underlying raw HTML, which is generated by running the XSLT. By viewing the page source of our output you can review the underlying raw HTML (http://newtfire.org/dh/akiraSample.html).

Guide to Approaching the Problem

In XSLT, processing something normally happens in two parts. You normally have an <xsl:apply-templates> element that tells the system what elements (or other nodes) you want to process, and you then have an <xsl:template> element that tells the system exactly how you want to process those elements, that is, what you want to do with them. If you find the image helpful, you can think of this as a situation where the <xsl:apply-templates> elements *throw some nodes out into space and say "would someone please process these?"* and the various <xsl:template> elements sit around watching nodes fly by, and when they **match** something, they *grab it and process it*.

Therefore, for this assignment, your XSLT transformation (after all the housekeeping) should have several template rules:

1. Begin with **a special template rule for the document node** (<xsl:template match="/">), in which you set up the basic HTML structure: the <html> element, <head> and its contents, and <body>.
2. Inside the <body> element that just created, write an HTML h1 element to hold the title you want viewers to see on the web page, and use xsl:apply-templates inside to select the part of the input TEI that will give you the title. When you need to set the value of the @select attribute on xsl:apply-templates, you are being choosy, pulling *just* what you need into position where you want it.
3. Beneath the main title, create a secondary header in HTML (an h2 element), just type the words "Cast List" into it. You are basically writing some HTML code within the XSLT document!
4. Set up an HTML table, and give it a row of two th (table header) cells to help start a table of abbreviated IDs and values. After the table header, you will want to simply apply-templates to select each TEI person element to process in a new template rule. Anything that is going to have to be processed multiple times needs to just be called once in the special template match on the document node, to apply-templates selecting these elements.
5. Write a new template rule to match on the TEI person element and output (each time it finds person) an HTML table row (tr) containing two cells (td). Inside each cell, pull the relevant information from the person element that you wish to present. (We recommend outputting the @xml:id in one table cell, and just the *first* persName element.)
6. Then create separate template rules that match on each of the inline elements we planned above to match and style. Each rule will be "called" or "fired" as a result of the preceding <xsl:apply-templates> selection from our first template rule.

In this case, then, your @select on the <xsl:apply-template> elements inside the template rule for the document node will tell the system what specific elements (using their XPath location in the source XML) you want to appear and where in your output HTML you wish for them to appear. You create the order each selection appears by placing the various <xsl:apply-template> elements in the desired order inside of that first template rule matching on the document node. This will tell the system that you want to **select** only certain elements, at which point the template rule for the document node will call out what portions of the document need to be processed at this particular point. The processing work actually gets done by the other <xsl:template> rules, the ones that you write to then **match** on the elements that need styled.

The elegant simplicity of <xsl:apply-templates>

Akira's spGrp elements wrap around clusters of speeches unpredictably. Similarly, if you were processing prose paragraphs with markup floating around unpredictably with the text, you would have an unpredictable combination of elements that we call "mixed content", with varied and unpredictable combinations of elements. This is the problem that *declarative* XSLT was designed to solve. With a traditional *procedural* programming language, you'd have to write rules like "inside the body, if there's a <spGrp> do X, but if there isn't do Y, and, oh, by the way, check whether there's a <l> or a <p> inside the <sp> elements, etc." That is, most programming languages have to tell you what to look for at every step. The elegance of XSLT is that all you have to say inside paragraphs and other elements is "I'm not worried about what I'll find here; just process **(apply templates to)** all my children, *whatever they might be*."

The way to deal with mixed content in XSLT is to create a template rule for every element you care about and use it to output whatever HTML markup you want for that element. Then, inside that markup, you can include a general <xsl:apply-templates/>, not specifying a @select attribute. For example, if you want your <persName> elements to be tagged with the HTML tags, which means "strong emphasis" and which is usually rendered in bold, you could have a template rule like:

```
<xsl:template match="persName">
  <strong>
      <xsl:apply-templates/>
  </strong>
</xsl:template>
```

You don't know or care whether <persName> has any children nodes or, if it does, what they are. Whatever they are, this rule tells the system to try to process them, and as long as there's a template rule for them, they'll be taken care of properly somewhere else in the stylesheet. If there are no children nodes, the <xsl:apply-templates/> will apply vacuously and harmlessly. As long as every element tells you to process its children, you'll work your way down through the hierarchy of the document without having to know which elements can contain which other elements or text nodes.

Taking stock: when to use @select

In our XSLT tutorial we describe the use of <xsl:apply-templates select="…"/> which specifies exactly what you want to process and where. That makes sense when your input and output are very regular in structure. *Use the @select attribute when you know exactly what you're looking for and where you want to put it*. We will want to use <xsl:apply-templates select="…"/> in order to grab all of the <person> elements sitting inside of the <particDesc> element so we can output them up in a Cast List near the top of our HTML file, separate from the Akira script that we want to come out below, selected from the TEI <body> element. We will also want to use the <xsl:apply-templates select="…"/> in order to reach for attribute values like the @xml:id on person to output them inside HTML table cell (td) elements. By setting up these very specific selections of these elements and attributes, we are paring down or "trimming" the XML tree of the input document to designate exactly and only what we want. Remember. what is represented in the <html> element of your XSLT is the basic superstructure of your output HTML document. The content inside the HTML <head> element, including the <title> element, will not appear in the web browswerunless someone is reading your HTML source code. Hence the importance in creating visible body headings with elements (<h1>, <h2>, etc.) that contain the document title information.

After you have selected the portions of the document to process for your Cast of Characters table, and to output the script, for the rest of this assignment you don't need to write the template rules in any particular order. Those template matches will fire as elements in the document turn up to be processed, whenever it comes up. Basically, <xsl:apply-templates/> without the @select attribute says "apply templates to whatever you find." *Omit the @select attribute where you don't want to have to think about and cater to every alternative individually*. (You can still treat them all differently because you'll have different template rules to "catch" them, but when you assert that they should be processed, you don't have to know what they actually are.)

Sorting

An alphabetically sorted Cast of Characters may be useful for humans who want to look up more information about a speaker they see in the script. We want to make an alphabetized list sorted by the abbreviated name given in the `person/@xml:id`. Start by looking up `<xsl:sort>` in the Michael Kay book or at https://www.w3schools.com/xml/xsl_sort.asp. So far, if we want to output our cast in the order in which they occur *Akira* script, we've used a self-closing empty `<xsl:apply-templates/>` to select them with something like `<xsl:apply-templates select="descending::particDesc//person"/>`. But the self-closing empty element tag is informationally identical to writing the start and end tags separately with nothing between them, that is:

```
<xsl:apply-templates select="descendant::particDesc//person>
</xsl:apply-templates>
```

To cause the elements being processed to be sorted first, you need to use this alternative notation, with separate start and end tags, because you need to put the `<xsl:sort>` element between the start and end tags. If you use the first notation, the one with a single self-closing tag, there's no "between" in which to put the `<xsl:sort>` element. In other words, you want something like:

```
<xsl:apply-templates select="descendant::particDesc//person">
  <xsl:sort select="what specific aspect of the person element you want to sort on, such as an attribute or child element"/>
</xsl:apply-templates/>
```

Without an `@select` attribute on `<xsl:sort>` this would sort on child text content of the `<person>` elements alphabetically by their text value (and if there is no text, there won't be anything to sort, so the sort will fail). Since our `person` elements only contain other elements, we need to use the `@select` attribute on `<xsl:sort>`. Note that you can set an `@order` attribute to sort in ascending or descending order. Also you do not have to sort alphabetically. You can sort by numerical counts of something, for example, how often a particular character appears in the script. (We sorted our Cast Table in both ways.) **Challenge:** Can you figure out how to sort based on a count of the number of appearances, or the number of times the character speaks in the production? **Hint:** To do this requires searching the XML tree for the `sp` elements whose `@who` attribute values match up with the current person element. You will need a string-matching function, because the `@who` attributes have a `#` in front of the id. Typically we strip that off using the `substring-after()` function, so we look for the `substring-after(@who, '#')` to see where that substring = the `current()` xml:id. We need to use `current()` to designate the specific person element being processed (it's a little like processing $i in a for-loop).

What should the output look like

We are sure you can do better than our sample output! HTML provides a limited number of elements for styling in-line text, which you can read about at http://www.w3schools.com/html/html_formatting.asp. You can use any of these in your output, but think about your decisions. For layout purposes, *block elements* like `div` or `h1` or `p` literally take up a rectangular "block" on the page and can be styled accordingly (given padding, etc. *Inline elements*, like `span` or `em` or `strong` are meant to run within blocks (inside paragraphs, for example), and are good for highlighting within the line, for example to style speaker names or speech numbers to introduce each speech. Finally, presentational elements, the kind that describe how text looks (e.g., `<i>` for "italic"), are generally regarded as less useful than descriptive tags, which describe what text means (e.g., `<em>` for "emphasis"). Both of the preceding are normally rendered in italics in the browser, but the semantic tag is more consistent with the spirit of XML than the presentational one.

The web would be a dull world if the only styling available were the handful of presentational tags available in vanilla HTML. In addition to those options, there are also ways to assign arbitrary style to a snippet of in-line text, changing fonts or colors or other features in mid-stream. To do that:

1. Before you read any further in this page, read Obdurodon's Using `<span>` and `@class` to style your HTML page.
2. To use the strategies described on that page, create an XSLT template rule that transforms the element you want to style to an HTML `div` or `<span>` element with a `@class` attribute. For example, you might transform `<spGrp>` in the input XML to `<div class="spGrp">`...child nodes (processed in XSLT with `<xsl:apply-templates/>`) ...`</span>` in the output HTML. You can then specify CSS styling by reference to the `@class` attribute, as described in the page we link to above.

   Note that you can make your transformations very specific. For example, instead of setting all `<sp>` elements to the same HTML `@class`, you can create separate template rules to **match** on special `sp[@who="#colonel"]` and `sp[@who="#doctor"]` according to their attribute values. (You can even use the pipe (|) to unify these as two options for a template match:

   ```
   <xsl:template match="sp[@who='#doctor'] | sp[@who='#colonel']">
             <span class="commanders">
             <strong><xsl:apply-templates select="@who"></strong>
             <xsl:apply-templates/>
             </span class="commanders">
   </xsl:template>>
   ```

   Notice how we used two `<xsl:apply-templates/>` statements here, one which selected an attribute value to output, and the other just to process whatever child contents of the <sp> elements turn up. Around both of them, we set a special `<span>` element with a logical `@class` (we used the value "commanders" to help associate these two controlling figures in *Akira*). In our CSS we make reference to the `@class`, again as described in the page we link to above.

3. Setting `@class` attributes in the output HTML makes it possible to style the various `<span>` elements differently according to the value of those attributes, but you need to create a CSS stylesheet to do that. Create the stylesheet (just as you've created CSS in the past), and specify how you want to style your `<span>` elements. Link the CSS stylesheet to the XSLT by creating the appropriate `<link>` element inside of the HTML `<head>` element of your XSLT (you can remind yourself of the `<link>` element format by referencing our CSS Tutorial).
4. Besides wrapping your `<xsl:apply-templates/>` in `<span>` elements and other HTML elements, you might consider adding extra spaces or text outside some of these as well. To do this, experiment with inserting `<xsl:text>...</xsl:text>` where you would like spaces or characters (say a colon and some white space to follow a speaker name in the script).
5. You may want to style your table so you can see the outlines of the table cells, and add colors and styling. For some guidance, see the w3schools CSS tutorial on tables, which shows you some nifty tricks like how to style every other row to shade it differently.

Your Final Results

What you should produce, then, is:

- An XSLT stylesheet that transforms the contents of the source document into HTML, giving us at least one sorted Cast List and a reading view of the *Akira* script.
- The resulting HTML should also style and at least some of those styles should be set using block `<div>` and inline `<span>` elements with the `@class` attribute to group related kinds of elements visually.
- You need to create a CSS file, **linked to your output HTML**, that specifies how to style the output document. You can look up the most useful of the available CSS properties at http://www.w3schools.com/css/. We'd suggesting following the links on the left under "CSS styling" for styling backgrounds, text, and fonts, as well as the link for borders under "CSS box model".

Important

- *Before submitting your homework, you must run the transformation at home, and open the results as a new file in <oXygen/>* to make sure the results are what you expect them to be. (There's a guide to running XSLT transformations inside <oXygen/> on Obdurodon at http://dh.obdurodon.org/oxygen-xslt-configuration.html.) If you don't get the results you expect and can't figure out what you're doing wrong, remember that you can post a query to our DHClass-Hub Issues board. Don't just ask for the answer, though; you need to describe what you tried, what you expected, what you got, and what you think the problem is. We often find, just as we're preparing to post our own queries to coding discussion boards, that having to write up a description of the problem helps us think it through and solve it ourselves. We're also encouraging you to discuss the homework on DHClass-Hub Issues because that's also helpful for the person who responds. Answering someone else's inquiry and troubleshooting someone else's problem often helps us clarify matters for ourselves!
- When you complete this assignment, submit your XSLT file and CSS file to Courseweb, following our usual homework file-naming conventions. We will run your XSLT transformation to see what output it generates, so you do not need to submit your output file. However, it is important that you include your CSS so we can locally associate it to your XSLT (keeping them in the same folder space) and see your final output. **Link the CSS in the XSLT for us, so that when we run the XSLT it generates the `<link>` element automatically.**

newtFire {dhlds}
Maintained by: Elisa E. Beshero-Bondar (ebb8 at pitt.edu) (cc) BY-NC-SA Last modified: Sunday, 22-Oct-2017 21:10:17 EDT. Powered by firebellies.

XSLT Exercise 2

Overview of the Assignment

The Digital Mitford Site Index stores lists of names and information on people, places, organizations, and texts, among other kinds of named entities referenced in files throughout the Digital Mitford project. For this assignment, we will work with a slightly modified version of the Digital Mitford Site Index, which you should download from here and open in <oXygen>. Our goal is to create a structured outline in HTML of all the information about organizations in the site index. We want to output that in HTML in the form of a list with nested lists inside, representing an outline of first the *categories* of organization, and then inside each category, a new list of the organization names. This is something we actually need to do in the Mitford project: to process portions of the Site Index file to make it readable on the web as a list. One possible use of a webpage like this is as a list of links, so that each organization name might link to a page of information on each organization. We don't have to generate those links now. For this assignment, we just want to learn how to transform XSLT to HTML and to generate the lists themselves by pulling the right content out of our XML.

If you're feeling adventurous, once you obtain the output we're seeking, you may go on to build other HTML lists, working with other portions of the XML document, such as the `<listBibl>` or `<listPerson>` sections, which are formatted a little differently. The only required content of your homework, though, is the HTML outline of **organization types** and **organization names**. For the organization types or categories, we need to pull from the <head> element sitting inside at the top of each <listOrg> elements in our TEI file. For the organization names, we reach in to find the individual entries for <org> and their child <orgName> elements inside each <listOrg> element. Each <org> element contains one <orgName> inside that holds the best-known name of a particular organization. You may first want to experiment with XPath on the Site Index file to locate the <listOrg> elements and study the XML hierarchy of the lists. Let's make the outer list be **ordered** (or numbered) list in HTML, using the HTML <ol> element, and then make the inner list be an **unordered** (bulleted) list, using the HTML <ul> element.

Your lists in HTML should come out looking something like this, only yours will have a few more entries in each category, because your XML document contains some new material.

1. Archives Holding Mitford's Papers
    - Baylor University, Armstrong Browning Library
    - Berkshire Record Office
    - British Library
    - Boston Public Library
    - Cambridge University: Fitzwilliam Museum
    - Duke University Rubenstein Library
    - Eton College
    - Florida State University Special Collections
    - The Women's Library, Glasgow
    - Houghton Library, Harvard
    - Huntington Library
    - University of Iowa Special Collections
    - Massachusetts Historical Society
    - New York Public Library
    - Oxford University, Balliol College Archives
    - Oxford University, Bodleian Library
    - Reading Central Library The principal archive of Mary Russell Mitford's personal papers and related documents, holding approximately 1,000 manuscripts and a nearly comprehensive collection of her publications.
    - John Ruskin Library, Lancaster
    - The John Rylands Library
    - National Library of Scotland, Manuscript Collections
    - University of Texas, Ransom Center
    - University of Reading Special Collections
    - University of Virginia Special Collections
    - Wellesley College, Margaret Clapp Library, Special Collections
    - Wordsworth Trust
    - Yale University, Beineke Library
2. Organizations Relevant to Mitford's World
    - Billiard Club
    - House of Bourbon
    - Cavaliers
    - Court of Chancery
    - Church of England
    - the Cockney School
    - Drover
    - Eton College
    - High Court of Justice
    - House of Commons
    - the Kembles
    - House of Medici
    - Mitford
    - Mr.and Mrs.Mitford
    - the Moncks, family of John Berkeley Monck
    - New Model Army
    - Palmerite
    - Parliament
    - Court of Pope Pius VII
    - Prelacy
    - the Presybterian faction
    - Privy Council
    - Richmond Coach or Stage
    - Scriblerus Club
    - Slade family
    - Taylor and Hessey (publishers)
    - Tory Party
    - Twickenham Coach or Stage
    - Valpy family
    - Webb family
    - Weylandite

3. Fictional Organizations Referenced by Mitford
    - Attendants &c.
    - Citizens
    - Guards
    - Guards
    - Ladies
    - Nobles (in Julian)
    - Nobles (in Rienzi)
    - officers in Charles I
    - Prelates

The underlying HTML, which we generated by running XSLT, should look like this:

```
<ol>
      <li>Archives Holding Mitford's Papers<ul>
            <li>Baylor University, Armstrong Browning Library</li>
            <li>Berkshire Record Office</li>
            <li>British Library</li>
            <li>Boston Public Library</li>
            <li>Cambridge University: Fitzwilliam Museum</li>
            <li>Duke University Rubenstein Library</li>
            <li>Eton College</li>
            <li>Florida State University Special Collections</li>
            <li>The Women's Library, Glasgow</li>
            <li>Houghton Library, Harvard</li>
            <li>Huntington Library</li>
            <li>University of Iowa Special Collections</li>
            <li>Massachusetts Historical Society</li>
            <li>New York Public Library</li>
            <li>Oxford University, Balliol College Archives</li>
            <li>Oxford University, Bodleian Library</li>
            <li>Reading Central Library The principal archive of Mary
                  Russell Mitford's personal papers and related documents, holding
                  approximately 1,000 manuscripts and a nearly comprehensive collection of her
                  publications.
               </li>
            <li>John Ruskin Library, Lancaster</li>
            <li>The John Rylands Library</li>
            <li>National Library of Scotland, Manuscript Collections</li>
            <li>University of Texas, Ransom Center</li>
            <li>University of Reading Special Collections</li>
            <li>University of Virginia Special Collections</li>
            <li>Wellesley College, Margaret Clapp Library, Special Collections</li>
            <li>Wordsworth Trust</li>
            <li>Yale University, Beineke Library</li>
         </ul>
      </li>
      <li>Organizations Relevant to Mitford's World<ul>
            <li>Billiard Club</li>
            <li>House of Bourbon</li>
            <li>Cavaliers</li>
            <li>Court of Chancery</li>
            <li>Church of England</li>
            <li>the Cockney School</li>
            <li>
                  Drover
               </li>
            <li>Eton College</li>
            <li>High Court of Justice</li>
            <li>House of Commons</li>
            <li>the Kembles</li>
            <li>House of Medici</li>
            <li>
                  Mitford
               </li>
            <li>Mr.and Mrs.Mitford</li>
            <li>the Moncks, family of John Berkeley
                  Monck
               </li>
            <li>New Model Army</li>
            <li>Palmerite</li>
            <li>Parliament</li>
            <li>Court of Pope Pius VII</li>
```

```
            <li>Prelacy</li>
            <li>the Presybterian faction</li>
            <li>Privy Council</li>
            <li>Richmond Coach or Stage</li>
            <li>Scriblerus Club</li>
            <li>
                  Slade family
               </li>
            <li>Taylor and Hessey (publishers)</li>
            <li>Tory Party</li>
            <li>Twickenham Coach or Stage</li>
            <li>
                  Valpy family
               </li>
            <li>
                  Webb family
               </li>
            <li>Weylandite</li>
         </ul>
      </li>
      <li>Fictional Organizations Referenced by Mitford<ul>
            <li>Attendants &amp;amp;c.</li>
            <li>Citizens</li>
            <li>Guards</li>
            <li>Guards</li>
            <li>Ladies</li>
            <li>Nobles (in Julian)</li>
            <li>Nobles (in Rienzi)</li>
            <li>officers in Charles I
               </li>
            <li>Prelates</li>
         </ul>
      </li>
   </ol>
```

In HTML ordered and unordered lists, the only elements permitted inside are list items or <li> elements. We've nested them so that each list item in the outside numbered list contains a category type (designating what kind of organization), followed by an embedded <ul> that contains, in turn, a separated bulleted list series, listing the name of each organization in the list.

Before You Begin: Set up the XSLT Stylesheet to Read TEI

The Digital Mitford's Site Index file is coded in the TEI namespace, which means that your XSLT stylesheet (much as in the last assignment) requires an instruction at the top to specify that when it tries to match elements, it needs to match them in the TEI namespace. (When you create an new XSLT document in <oXygen/> it won't contain that instruction by default, so whenever you are working with TEI you need to add it (See the text in blue below). To ensure that the output would be in the XHTML namespace, we added a default namespace declaration (in purple below). To output the required DOCTYPE declaration, we also created `<xsl:output>` element as the first child of our root `<xsl:stylesheet>` element (in green below), and we needed to include an attribute there to omit the default XML declaration because if we output it that XML line in our XHTML output, it will not produce valid HTML with the w3C and might produce quirky problems with rendering in various web browsers. So, our modified stylesheet template and xsl:output line is this, and you should copy this into your stylesheet:

```
<?xml version="1.0" encoding="UTF-8"?>
         <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="2.0"
    xmlns="http://www.w3.org/1999/xhtml"
    xpath-default-namespace="http://www.tei-c.org/ns/1.0">

    <xsl:output method="xhtml" encoding="utf-8" doctype-system="about:legacy-compat"
        omit-xml-declaration="yes"/>

    </xsl:stylesheet>
```

Guide to Approaching the Problem

Our XSLT transformation (after all this housekeeping) has three template rules:

1. We have a template rule for the **document node** (`<xsl:template match="/">`), in which we create the basic HTML file structure: the `<html>` element, `<head>` and its contents, and `<body>`—anything that appears just once in the HTML document (one to one relationship with the root node). Inside the `<body>` element that we're creating, we use `<xsl:apply-templates>` and select the `<listOrg>` elements (using an XPath expression as the value of the `@select` attribute). And we create our wrapper `<ol>` tags to set up the ordered list of organization types.
2. We have a separate template rule that matches the `<listOrg>` elements (holding the lists of organizations), so it will be invoked as a result of the preceding `<xsl:apply-templates>` instruction, and will fire once for each `<listOrg>` element in our Site Index. Inside that template rule we create a new list item (`<li>`) for the particular `<listOrg>` being processed and inside the tags for that new list item we do two things. First, we apply templates to the `<head>` for the `<listOrg>`, which will cause its category description to be output when we run the transformation. Second, we create wrapper `<ul>` tags for the nested list that will contain the names of the organizations within that category. Inside that new `<ul>` element, we use an `<xsl:apply-templates>` rule to apply templates to (that is, to process) the `<org>` elements of that `<listOrg>`.
3. We have a separate template rule that matches the `<org>` elements, which make up the items in the list of organizations, and that just applies templates to the `<orgName>` element within each `<org>`. This rule will fire once for each `<org>` element inside the `<listOrg>`, and it will be called separately for the `<org>` elements within each `<listOrg>`, so that the orgs will be rendered properly in their respective lists.

We don't need a template rule for the `<head>` elements themselves because the built-in (default) template rule in XSLT for an *element* that doesn't have an explicit, specified rule is just to apply templates to its children. The only child of the `<head>` elements is a text node, and the built-in rule for *text nodes* is to output them literally. In other words, if you apply templates to `<head>` and you don't have a template rule that matches that element, ultimately the transformation will just output the textual content of the head, that is, the title that you want.

Important

- Those who like to read ahead or already have some programming experience with other languages may have noticed that XSLT includes an `<xsl:for-each>` instruction that *could* be used to solve this problem. *We are prohibiting its use for now;* your solution must use `<xsl:template>` and `<xsl:apply-templates>` rules instead. There's a Good Reason for this, which we'll explain later, when we talk about situations where you *should* use `<xsl:for-each>`.
- You may notice that two or three of your output bulleted list items show multiple related organization names squished together. This is because our editors occasionally provided more than one name used for an organization. Our standing rule is that the most definitive `orgName` be listed *first* in the list of names, so we recommend that you tidy up your list by selecting just the very first available orgName, that is, the first element child named `orgName` of `org` elements you are processing. Alternatively, you may try applying an XPath `string-join()` function to output the entries, but you will need to use `xsl:value-of` instead of `xsl:apply-templates` because we need to use `xsl:value-of` to calculate the results of functions (which removes us from the XML tree). Either approach is fine with us, and you would use the same `@select` attribute to indicate what you would like to output.
- *Before submitting your homework, you must run the transformation at home* to make sure the results are what you expect them to be. Remember, there's a guide to running XSLT transformations inside <oXygen/> in our Intro to XSLT tutorial. If you don't get the results you expect and can't figure out what you're doing wrong, remember that you can post a query to our DHClass-Hub Issues board. You can't just ask for the answer, though; you need to describe what you tried, what you expected, what you got, and what you think the problem is. We often find, just as we're preparing to post our own queries to coding discussion boards, that having to write up a description of the problem helps us think it through and solve it ourselves. We're also encouraging you to discuss the homework on the discussion boards because that's also helpful for the person who responds. Answering someone else's inquiry and troubleshooting someone else's problem often helps us clarify matters for ourselves!
- When you complete this assignment, submit your XSLT file and your output HTML file to Courseweb, following our usual homework file-naming conventions.

newtFire {dhlds}
Maintained by: Elisa E. Beshero-Bondar (ebb8 at pitt.edu)  Last modified: Sunday, 22-Oct-2017 21:10:14 EDT.
Powered by firebellies.

XSLT Exercise 5

The input collection on our DHClass-Hub

For this assignment and the next, you will be working with a digitized XML collection of Emily Dickinson's poems and you will need to access this collection in GitHub. This assignment requires you to use our **DHClass-Hub** so you can work with a **a local directory of files** rather than just one at a time as we have been doing up to this point. Here is how to access the directory:

- Sync the DHClass-Hub to your computer, or clone the repository if it is not already on your computer. (If you have not synced or cloned a GitHub repository in a while; please see the instructions posted in our Readme file).
- When you have synced the repository, open the DHClass-Hub locally on your computer, and find in it the **Assignment-Files** directory. Inside it is a directory named **Dickinson** that contains a eleven XML files that we are working with as a collection in this assignment.
- **Copy this Dickinson directory to some other location on your computer** outside of your GitHub directories. (We do not want you to push your homework to the whole class over our DHClass-Hub, so we just need you to make your own private copy of this directory to work with in the same folder in which you do your homework for this assignment and the next.
- Do not rename the file folder or the files inside, as we need to refer to them as a coherent collection.

Please be careful to **copy** rather the move the directory out of GitHub! If you move it out of the directory, the next time you sync our DHClass-Hub, GitHub will prompt you to commit the change and push it, which will effectively eliminate the Dickinson folder. One of us instructors can easily put it back if that happens, but please alert us ASAP if something goes awry!

Working with a Collection of Files in XSLT

Emily Dickinson made little bundles of her manuscript poems with a needle and thread, and these have come to be known as *fascicles* by Dickinson scholars. We have digitally reproduced a bundle that Dickinson scholars have named Fascicle 16 by using a folder or directory, which holds a digital collection of files together. We can process a whole directory of files using the `collection()` function in XSLT, so we can represent content from a whole collection of XML files in one or more output HTML files. One useful application for working with a collection is to process several short XML files and unify them on a single HTML page designed to merge their content. In this case, we will be representing the poems encoded in eleven small XML files inside one HTML page, which we will produce with a table of contents giving poems by number and first lines, followed by the full text of the poems themselves, formatted in HTML with numbered lines. Since these poems are all encoded with the same structural elements, we can use the `collection()` function to reach into them as a group, and output their content one by one based on their XML hierarchy. Really, we are treating the collection itself as part of the hierarchy as we write our XSLT, so we move from the directory down into the document node of each file to do our XSLT processing.

Using modal XSLT

Besides working with a collection of files, the other interesting new application in this assignment is **modal XSLT**, which lets you process the same nodes in your document in two different ways. How can you output the same element contents to sit as list items in a table of contents at the top of an HTML page, *and also* as headers positioned throughout the body of your document, below the table of contents? Wouldn't it be handy to be able to have two completely different template rules that match exactly the same elements: one rule to output the data as list items in the table of contents, and the other to output the same data as headers? You can write two template rules that will match the same nodes (have the same value for their `@match` attribute), but how do you make sure that the correct template rule is handling the data in the correct place?

To permit us to write multiple template rules that process the same input nodes in different ways for different purposes, we write **modal XSLT**, and that is what you will be learning to write with this assignment. Modal XSLT allows you to output the same parts of the input XML document in multiple locations and treat them differently each time. That is, it lets you have two different template rules for processing the same elements or other nodes in different ways, and you use the `@mode` attribute to control how the elements are processed *at a particular place* in the transformation. Please read the explanation and view the examples in Obdurodon's tutorial on Modal XSLT before proceeding with the assignment, so you can see where and how to set the `@mode` attribute and how it works to control processing.

Overview of the assignment

For this assignment you want to produce in one HTML page our collection of Emily Dickinson's eleven poems in Fascicle 16, and that page needs to have a table of contents at the top. The table of contents should have one entry for each poem, which produces the information we have encoded in `<title>` element that is a descendant of the `<body>` element in our XML source code, together with the first line, and a count of the number of variants we have recorded in each poem. Below the full table of contents (one line for each poem) you should render the complete text of all eleven poems, and wrap span elements around the text we have marked as variants, ideally by using a `@class` attribute that holds the same information as the `@wit` attribute on the `<rdg>` element in our source texts. To generate the attribute value on @class, we used an Attribute Value Template, which you should read about here. You can see our output at http://newtfire.org/dh/dickinson-5.html.

Housekeeping with the stylesheet template: From TEI to XHTML

Our Emily Dickinson collection is coded in the TEI namespace, which means that your XSLT stylesheet must include an instruction at the top to specify that when it tries to match elements, it needs to match them in that TEI namespace. When you create an new XSLT document in <oXygen/> it won't contain that instruction by default, so whenever you are working with TEI you need to add it (See the text in blue below). To ensure that the output would be in the XHTML namespace, we added a default namespace declaration (in purple below). To output the required DOCTYPE declaration, we also created `<xsl:output>` element as the first child of our root `<xsl:stylesheet>` element (in green below), and we needed to include an attribute there to omit the default XML declaration because if we output it that XML line in our XHTML output, it will not produce valid HTML with the w3C and might produce quirky problems with rendering in various web browsers. So, our modified stylesheet template and xsl:output line is this, and you should copy this into your stylesheet:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xpath-default-namespace="http://www.tei-c.org/ns/1.0"
         xmlns:math="http://www.w3.org/2005/xpath-functions/math"
    exclude-result-prefixes="xs math"
    xmlns="http://www.w3.org/1999/xhtml"
         version="3.0">

         <xsl:output method="xhtml" encoding="utf-8" doctype-system="about:legacy-compat"
        omit-xml-declaration="yes"/>

</xsl:stylesheet>
```

How to begin

Begin by forgetting about the table of contents, and concentrate on just outputting the full text of the poems. Except for having to pull the poems from a collection of files, this is just like the XML-to-HTML transformations you have already written, and you'll use regular template rules (without a `@mode` attribute) to perform the transformation.

The collection() function: Here is how we write and run XSLT to process a collection of files. Just ahead of the first template match, after the `<xsl:output method>` statement, we define a **variable** in XSLT, which simply sets up a convenient shorthand for something complicated that we need to use more than once, so we don't have to keep retyping it.

<xsl:output method="xml" encoding="utf-8" indent="yes" doctype-system="about:legacy-compat"/>
```
<xsl:variable name="dickinsonColl" select="collection('Dickinson')"/>
```

An `xsl:variable` works by designating an `@name` which holds any name you like to refer to it later (we have used "dickinsonColl" here to refer to the Dickinson collection of files), and with `@select` it holds anything you wish: a complicated XPath expression or a function, or whatever it is that is easier to store or process in a variable rather than typing it out multiple times. We use variables to help keep our code easy to read! In this case, we are using a variable to define our collection, using the `collection()` function in the `@select` attribute. The `collection()` function is set to *designate the directory location of the collection of poems in relation to the stylesheet I am currently writing*. My XSLT is saved in the directory immediately above the Dickinson collection, so I am simply instructing the XSLT parser to take a path-step down to it by designating Dickinson inside the collection function. (You may wish to save your stylesheet in relation to the Dickinson collection just as I did, but in case you did not, you will simply need to figure out how to step up or down your file directory structure to reach the Dickinson folder, using `..` to climb up and `/` or `//` to step down.)

Within the stylesheet as we will see below, we will call this variable whenever we need it, to show how we are stepping into our collection of poems. That will happen in the first template rule that matches on the root element. Open any one of the input XML files in the Dickinson collection in <oXygen/> and you will see that the title and content of the poems are all coded within the `<body>` element, so we can write this stylesheet to look through the whole collection of files and process only the elements below `<body>`. You call or invoke the variable name for the collection by signalling it first with a dollar sign `$`, giving the variable name, and then simply step down the descendant axis straight to the `<body>` element in each file. Here is how the code looks to call or invoke the variable in our first template match:

```
<xsl:apply-templates select="$dickinsonColl//body"/>
```

Note on running the transformation: Unlike other transformations we do on single XML files, when we run the XSLT in <oXygen/> it actually doesn't matter what file we have selected in the XML input, because we have indicated in the stylesheet itself what we are processing, with the `collection()` function. We can actually set even a file that is outside of our collection as the input XML file (and we ran it successfully with the HTML file of the previous exercise selected). You do need to enter something in the input window, but when you work with the `collection()` function, your input file is just a dummy or placeholder that <oXygen/> needs to have entered so it can run your XSLT transformation.

In our HTML output (scroll down past the table of contents, to where the full text of the poems is rendered), the Poem number (and publication info in parentheses) are inside an HTML `<h2>` element and the stanzas of each poem are held and spaced apart using HTML `<p>` elements. To make each line of the poems start on a new line, we add an HTML empty `<br/>` ("[line] break") element at the end of each line within the stanza. If you don't include the `<br/>` elements, the lines will all wrap together in the browser. Numbering the lines is optional for our assignment, but we have done so in our sample output by using the `count()` function over the `<l>` elements on the `preceding::` axis (which we used instead of `preceding-sibling::`, because we wanted to number lines by counting them consecutively within each file rather than within each line group. (You can read about the `preceding::` axis in the Michael Kay book on page 612.) Here's the HTML output for one of our poems:

```
 <h2 id="p1611">Poem 11 </h2>
<p>
   1: He showed me Hights I never saw—<br/>
   2: "Would'st Climb," —He said?<br/>
   3: I said—"Not so"—<br/>
   4: "With me—" He said—"With me"?<br/>
   5: He showed me Secrets—Morning's Nest—<br/>
   6: The Rope the Nights were put across—<br/>
   7: "And now—"Would'st have me for a Guest"?<br/>
   8: I could not find my "Yes".
</p>
<p>
   9: And then, He brake His Life—And lo,<br/>
   10: A Light, for me, did solemn glow,<br/>
   11:
         <span class="var0">The steadier, as my face withdrew</span>
         <span class="var1">The larger—as my face withdrew</span>

   <br/>
   12: And could I, further, "No"?
</p>
```

The fine print: Don't worry if your HTML output isn't wrapped the same way ours is, if it puts the empty line break elements at the beginnings of lines instead of at the ends, or if it serializes (spells out) those empty line break elements as `<br></br>` instead of as `<br/>`. Those differences are not *informational* in an XML context. You can open your HTML output in <oXygen/> and pretty-print it if you'd like, which may make it easier to read, but as long as what you're producing is valid HTML and renders the text appropriately, you don't have to worry about non-informational differences between your markup and ours.

More fine print: You need a line break only between lines, which is to say that you don't need a `<br/>` element at the end of the last line of the poem because that's the end of the containing `<p>`, and not between lines. In our solution we used an `<xsl:if>` element to check the position of the line and output the `<br/>` only for non-final lines. If you're feeling ambitious, you can look up `<xsl:if>` at http://www.w3schools.com/xsl/xsl_if.asp or by searching for xsl:if on Obdurodon's XSLT Advanced Features tutorial, or looking it up in Michael Kay so you can perform this check yourself. If not, you can just output the `<br/>` element after all the concluding lines of line-groups in the poems. That's not really considered good HTML style, and you don't want to do it in your own projects, but it won't interfere with the legibility in the browser and we'll let it pass for homework purposes.

Once your poems are all being formatted correctly in HTML, you can add the functionality to create the table of contents at the top, using modal XSLT.

Adding the table of contents

The template rule for the document node in our solution, revised to output a table of contents with all the information we wish to show before the text of the poems, looks like the following:

```
          <xsl:variable name="dickinsonColl" select="collection('Dickinson')"/>

    <xsl:template match="/">
     <html>
         <head><title>Emily Dickinson's Fascicle 16</title></head>
         <body>

         <h1>Emily Dickinson's Fascicle 16</h1>
            <h2>Table of Contents</h2>
       <ul><xsl:apply-templates select="$dickinsonColl//body" mode="toc"/></ul>
            <hr/>
<!--ebb: This template rule sets up my "toc" mode for the table of contents,
   so that in the top part of the document we'll output a selection of the body elements specially formatted for my Table of
   and so that in another section of my document below, which I've put inside a <div> element, we can also output the full t
   Notice how I have invoked my variable multiple times here with the $ notation: $dickinsonColl -->
         <div id="main">
            <xsl:apply-templates select="$dickinsonColl//body"/>

         </div>

         </body>

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

The highlighted code is what we added to include a table of contents, and the important line is `<xsl:apply-templates select="$dickinsonColl//body" mode="toc"/>`. This is going to apply templates to each poem *with the `@mode` attribute value set to "toc"*. The value of the `@mode` attribute is up to you (we used "toc" for "table of contents"), but whatever you call it, setting the `@mode` to any value means that only template rules that also specify a `@mode` with that same value will fire in response to this `<xsl:apply-templates>` element. Now we have to go write those template rules!

What this means is that when you process the `<body>` elements to output the full text of the poems, you use `<xsl:apply-templates>` and `<xsl:template>` elements without any `@mode` attribute. To create the table of contents, though, you can have `<xsl:apply-templates>` and `<xsl:template>` elements that select or match the same elements, but that specify a mode and apply completely different rules. A template rule for `<body>` elements in table-of-contents mode will start with `<xsl:template match="$dickinsonColl//body" mode="toc">`, and you need to tell it to create an `<li>` element that contains the text of the `<title>` element and a first line, both fetched from the poem in the input XML collection of files. The rule for those same elements not in any mode will start with `<xsl:template match="$dickinsonColl//body">` (without the `@mode` attribute). That rule will create the `<h2>` header to hold the text of the `<title>` element and then output the full text of the poem in a `<p>`, with `<br/>` elements between the lines. In this way, you can have two sets of rules for the poems, one for the table of contents and one to output the full text, and we use modes to ensure that each is used only in the correct place.

Remember: both the `<xsl:apply-templates>`, which tells the system to process certain nodes, and the `<xsl:template>` that responds to that call and does the processing must agree on their mode values. For the main output of the full text of every poem, neither the `<xsl:apply-templates>` nor the `<xsl:template>` elements specifies a mode. To output the table of contents, both specify the same mode.

Completing and checking your work

- *Before submitting your homework, you must run the transformation at home* to make sure the results are what you expect them to be. Remember, there's a guide to running XSLT transformations inside <oXygen/> in our Intro to XSLT tutorial. If you don't get the results you expect and can't figure out what you're doing wrong, remember that you can post a query to our DHClass-Hub Issues board. You can't just ask for the answer, though; you need to describe what you tried, what you expected, what you got, and what you think the problem is. We often find, just as we're preparing to post our own queries to coding discussion boards, that having to write up a description of the problem helps us think it through and solve it ourselves. We're also encouraging you to discuss the homework on the discussion boards because that's also helpful for the person who responds. Answering someone else's inquiry and troubleshooting someone else's problem often helps us clarify matters for ourselves!
- When you complete this assignment, submit your XSLT file and your output HTML file to Courseweb, following our usual homework file-naming conventions. You need not generate CSS for this, because we will ask you to create one for a modified version of this output in the next assignment.

newtFire {dhlds}
Maintained by: Elisa E. Beshero-Bondar (ebb8 at pitt.edu) (cc) BY-NC-SA Last modified: Sunday, 29-Oct-2017 00:00:13 EDT. Powered by firebellies.

XSLT Exercise 6

The input collection on our DHClass-Hub

You will be working with same digitized XML collection of Emily Dickinson's poems that you worked with in the last assignment, the collection we posted on our DHClass-Hub GitHub. If you need a new copy of the files, please refer to our instructions on the previous exercise for accessing them by syncing, cloning, and copying the directory out of your local GitHub directory. You will be building on our XSLT Exercise 5, and you can take your stylesheet from that assignment and modify it for this one.

Overview of the assignment

For your last assignment you used the XSLT `@mode` attribute to create a table of contents for the Dickinson poems in Fascicle 16, using the poem number and first line of each poem as a surrogate for the title (since they don't have real titles). Our output for that previous assignment is at http://newtfire.org/dh/dickinson-5.html.

What's a table of contents good for anyway?

In a digital edition, we can just do a full-text search and scroll in the browser, so we don't really need a table of contents at all. We can search for a poem by number, we can search for the text of the first line, or we can search for a memorable phrase. But suppose we want to produce a paper edition, where the only organized access our users will get is the organization we decide to give them. What would be a useful table of contents or index?

A table of contents in the same order as the full text (numerical order), which is what we produced in the last assignment, duplicates the ordering information in the plain text. How useful is that? If we want to find a poem with a low number, we already know without a table of contents that we should look near the beginning. On the other hand, it's very common in published poetry collections to include an index of first lines, sorted in alphabetical order, so that a user who remembers just the first line of a poem can find it easily. It is a little less common to sort the order of a series of poems by an interesting feature inside them, but since one of our interests in digitizing this collection is to study Dickinson's use of variants, we would also like to sort the poems by a count of the variants in each, so that the poems with the highest number of variants come first in the sort order.

For this assignment we're going to enhance our output from the last assignment in the following ways:

- We're going to create links between the items in the table of contents and the poems, so that you can click on a poem's identifying information and be taken immediately to the corresponding poem.
- We're going to produce a list of the poems organized by the total number of the variants that Dickinson marked in them, from most variants to least variants.

- We're then going to alphabetize our list of first lines, so that the table of contents will be sorted alphabetically, instead of in numerical order.

Our HTML output for this assignment is at http://newtfire.org/dh/dickinson-6.html.

The tools we need

To create links between the first lines in the table of contents and the poems in the full text section of the page below we're going to use *attribute value templates* (AVT). We have been working with these in earlier XSLT assignments, but you may want to review Obdurodon's page on how they are written: http://dh.obdurodon.org/avt.xhtml.

To sort the table of contents we're going to use `<xsl:sort>`.

When we sort the first lines, they won't sort correctly for a quirky reason. We're going to fix that using the XPath `translate()` function, which we discuss below.

How HTML linking works

The `<li>` items in the table of contents should include `<a>` ("anchor") elements, which is how HTML identifies a clickable link. An anchor that is a clickable link has an `@href` attribute, which points to the target to which you want to move when you click on the link. For example, the table of contents might contain the following list item for Poem 6:

```
<li><a href="#p1606"><strong>Poem 6 (J 281: 1861/1935)</strong></a>:
                1: 'Tis so appalling—it exhilarates— <br /> [Variants: 4]</li>
```

HTML `<a>` elements that have `@href` attributes normally appear blue and underlined in the web browser, to advertise that they are links. The *target* of a link can be any element that has an `@id` attribute that identifies it uniquely. (This is why you need to use a hashtag (`#`) in the `@href` on the Table of Contents that links to an `@id`, because the `#` indicates you are pointing to the unique identifier that follows.) If you click on this line in the browser, the window will scroll to the element elsewhere in the document that has an `@id` attribute with the value "p1606". In our case, we've assigned that `@id` attribute value to the `<h2>` for that poem in the main body:

```
<h2 id="p1606">Poem 6 </h2>
```

Adding links to your output

You should first review Obdurodon's page on Attribute value templates (AVT), which describes a strategy you can use to create a unique `@id` attribute for each poem. For this task we gave our poems `@id` values that were a concatenation of the letter "p" and the distinct identifying number given in the `idno` element in the TEI header of each poem file: "1606" for Poem 6. We attached those `@id` attributes to the `<h2>` elements that we used as titles for each poem in the body of our page, e.g., `<h2 id="p1606">`. Meanwhile, in the table of contents at the top we created `<a>` elements with `@href` attributes that point to these `@id` values. *The value of the `@href` attribute must begin with a leading "#" character, but that "#" must not be part of the value of the `@id` attribute to which it points*. For example,

```
<li><a href="#p1606"><strong>Poem 6 (J 281: 1861/1935)</strong></a>:
                1: 'Tis so appalling—it exhilarates— <br /> [Variants: 4]</li>
```

means if the user clicks on the linked content in this list item, the browser will scroll to the line that reads `<h2 id="p1606">` in the main body of the page. *Remember: the value of the `@href` attribute begins with "#", but the value of the corresponding `@id` attribute on the `<h2>` element you want to scroll to doesn't.*

Armed with that information, you can take your answer to the main assignment and, using AVTs, modify it to create the `<a>` elements with the `@href` attributes and the `@id` attributes for the targets.

Sorting

An index of first lines in a collection of poems is usually alphabetized, because that's how humans look things up in that kind of list. We want to make an alphabetized list by first line, as well as sorted list by count of the variant phrases we have marked in these poems, so we wish to do two kinds of sorting in this assignment: one that is alphabetical and the other based on numbers derived from a `count()`. To learn how to sort your table of contents before you output it, start by looking up `<xsl:sort>` at https://www.w3schools.com/xml/xsl_sort.asp or in Michael Kay. So far, if we've wanted to output, say, our table of contents in the order in which they occur in the document, we've used a self-closing empty element to select them with something like `<xsl:apply-templates select="$dickinsonColl//body"/>`. We've also said, though, that the self-closing empty element tag is informationally identical to writing the start and end tags separately with nothing between them, that is, `<xsl:apply-templates select="$dickinsonColl//body"></xsl:apply-templates>`. To cause the elements being processed to be sorted first, you need to use this alternative notation, with separate start and end tags, because you need to put the `<xsl:sort>` element between the start and end tags. If you use the first notation, the one with a single self-closing tag, there's no "between" in which to put the `<xsl:sort>` element. In other words, you want something like:

```
<xsl:apply-templates select="$dickinsonColl//body">
  <xsl:sort/>
</xsl:apply-templates/>
```

As written, the preceding will sort the `<body>` elements alphabetically by their text value. As you'll see at the sites mentioned above, though, it's also possible to use the `@select` attribute on `<xsl:sort>` to sort a set of items by properties other than alphabetic order of their textual content, which is what we will be doing in sorting on a `count()` of the `<rdg>` elements that we used to signal variant words and phrases in Dickinson's text.

Using `translate()` to fix the alphabetical sort order

If you sort the first lines alphabetically according to their textual value, there will be two errors. The first lines of Poem 6 and Poem 9, "'Tis so appalling—it exhilarates—" and "'Twas just this time, last year, I died.", will show up first because in the internal representation of characters in the computer, the single straight apostrophe is "alphabetically" earlier than all of the letters. We can fix this by using `translate()` to strip the apostrophe for sorting purposes, but not for rendering. That is, we can sort as if there were no apostrophe, while still printing the apostrophe when we render the line.

We can't easily translate away an apostrophe, though, because quotation marks have special meaning in XPath. For the purpose of this assignment, you can ignore these two missorted lines or let the apostrophe be sorted first. If you're feeling ambitious, though, read Michael Kay's answer at http://p2p.wrox.com/xslt/50152-how-do-you-translate-apostrophe.html and see whether you can apply it to fixing this problem.

Another Optional Challenge, for either of the table of contents or the body output, or both: You may have observed in your HTML output that some of our titles are inconsistently formatted. Some poem

numbers have a period after them, and some only white space before the parenthetical information that summarizes each poem's publication history. You might see if you can find a way to: a) output only the poem and its number in the part of the document where you reproduce the poems, and/or b) remove the rogue period from the output, using the `replace()` function, which takes three arguments (for "finding a needle in a haystack" and then changing or removing it): an XPath leading to a string you want to alter (your "haystack"), a regular expression for the "needle" you want to find and change, and whatever you wish to convert it to (including nothing, to delete it). Read about `replace()` in the Strings section of The XPath functions we use the most and learn about its syntax by looking it up in the index of the Michael Kay book.

Finishing touches

Some lists of first lines of poetry put quotation marks around the lines. We haven't done that in our solution, but if you'd like to add it, you should use the HTML `<q>` ("quoted text") element, instead of outputting the raw quotation marks as plain text.

Oh, and did we mention CSS? Can you associate a CSS stylesheet to your output (write the CSS file link into your XSLT) to make it look more interesting than what you get by default in a web browser? See if you can find an interesting way to style the `<span>` elements surrounding the variants.

newtFire {dhlds}

Authored by: Nicole L. Lottig (nll29 at pitt.edu), Brooke A. Stewart (bas160 at pitt.edu), and Rebecca Parker (rjp43 at pitt.edu | Twitter: @bcpkr396)
Edited and maintained by: Elisa E. Beshero-Bondar (ebb8 at pitt.edu)
(cc) BY-NC-SA
Last modified: Sunday, 06-Aug-2017 18:16:14 EDT. Powered by firebellies.

Schematron Exercise 1

Preliminaries

Before beginning this assignment, please thoroughly read our introduction to Schematron. This tutorial will be useful to you during this assignment and the Schematron Exercise 2. To begin this assignment, you will need to open a new Schematron document in <oXygen/> under **File → New → New Document → (scroll to Schematron in the alphabetized list) → Schematron**. Once opened, you will keep the default xml line at the top, but you will delete everything from `<sch:schema>` down. You will then replace this with:

```
<schema xmlns:sch="http://purl.oclc.org/dsdl/schematron" queryBinding="xslt2"
    xmlns:sqf="http://www.schematron-quickfix.com/validator/process"
    xmlns="http://purl.oclc.org/dsdl/schematron">


        </schema>
```

You will be writing your Schematron **inside** the `<schema>` root element.

Analysis of the task

Background:

For this assignment, we are looking at votes for what place the Pitt-Greensburg DH Class will go for Spring Break. The options include: New York City, Mexico, London, and Rome. Each place gets between 0% to 100% of the votes. Assume here that this is the final voting poll, and there are no other options. This means that when you add the four percentages together, the result must be exactly 100%. Also assume that this is recording the already calculated percentage of the votes, not the raw count of the votes. All of these percentages are to be integer values.

Here is a Relax NG schema for the results of the Spring Break votes:

```
start = results
results = element results {place+}
place = element place {name, xsd:int}
name = attribute name {"NYC" | "Mexico" | "London" | "Rome"}
```

Here is a sample XML document that is valid against the above schema:

```
<results>
    <place name="NYC">34</place>
    <place name="Mexico">24</place>
    <place name="Rome">30</place>
```

```
    <place name="London">12</place>
</results>
```

Our Relax NG schema is a little sloppy and doesn't constrain the XML as thoroughly as it could have been better written (as we will discuss below). It lets us set a rule that the content of the element `<place>` must be a number (or xsd:int for integer), but the rule isn't really good enough as we will see from the from the following example:

<results>
<place name="NYC">27</place>
<place name="Mexico">39</place>
<place name="Rome">12</place>
<place name="London">15</place>
</results>

Do you see the problem? The four percentage values only total 93%! No matter how good our coding is, it is not possible to keep this type of error from happening by using Relax NG alone. That is why we use Schematron.

Task:

First, re-create the Relax NG schema file and the XML document by copying and pasting the **blue sample code above** into files with the appropriate file extensions. Associate your newly created Schematron and the Relax NG schema with your XML. As you write the following rules, "munge" (aka mess up) the XML to verify your rules are firing by entering correct and incorrect values into the XML.

1. Write a Schematron rule that verifies the four percentages always equal 100%.
2. Write a Schematron rule that fires an error when any location's voting percentage sits outside of the 0 to 100 range. There should be no negative integers and no integers greater than 100. (Hint: the Relax NG schema states that these values must be integers, so you will not have to worry about making sure of that; however, the computer parser will not recognize the values in each `<place>` as integers and instead will try to process them as strings of text. Use the `number()` function so the computer parses the values as numbers.)
3. Write a Schematron rule that tests there are only ever four place elements in our list of locations to visit for Spring Break.
4. Write a Schematron rule that tests if any of the `@name` values are repeated. It should not be possible for there to be any places that appear more than once in the XML. (Hint: Think about using the `count()` function for this. How many different values for `@name` should there be? How would you make sure each value is not repeated?

Optional Task:

Write a Schematron rule that tests whether the places are listed in order from greatest to least number of votes. (Hint: You will need to check the **numerical** value of each place with their **sibling** place's **numerical** value. Depending on your rule context you may need to clarify the position of the immediate sibling using the [1] position notation.)

Submission:

Upload your completed Schematron schema and your re-created XML document (**with your associated Schematron line**) on Courseweb. Please follow our standard filenaming conventions for

homework assignments uploaded to Courseweb.

newtFire {dhlds}
Authored by: Nicole L. Lottig (nll29 at pitt.edu) and Brooke A. Stewart (bas160 at pitt.edu) Edited and maintained by: Elisa E. Beshero-Bondar (ebb8 at pitt.edu)
(cc) BY-NC-SA
Last modified: Sunday, 06-Aug-2017 18:16:15 EDT. Powered by firebellies.

Schematron Exercise 2

Preliminaries

To work on this assignment, you will need to to find and do the following:

- **Information resources at the ready:** Review our Schematron tutorial, and read more about the XPath functions and syntax we describe below either on the web (see w3Schools' "XSLT, XPath, and XQuery Functions", Obdurodon's "The XPath Functions We Use the Most") or through offline searching with the index of the Michael Kay book.
- **XML file to test:** Save this TEI file locally and open it in <oXygen/>: Emily Dickinson's Fascicle 16 poems. You will need to associate your Schematron file with this document **in addition to** the currently associated TEI schema lines.
- Open a new Schematron document in <oXygen/> by going to **File → New** and typing "Schematron" in the "Type filter text" box, or by going to **File → New → New Document → (scroll to Schematron in the alphabetized list) → Schematron**. Once opened, you will keep the default xml line at the top, but you will delete everything from `<sch:schema>` down. To write Schematron rules for a document in the TEI namespace, you will then replace this with:

```
<schema xmlns:sch="http://purl.oclc.org/dsdl/schematron" queryBinding="xslt2"
    xmlns:sqf="http://www.schematron-quickfix.com/validator/process"
    xmlns="http://purl.oclc.org/dsdl/schematron">
        <ns uri="http://www.tei-c.org/ns/1.0" prefix="tei"/>

    </schema>
```

- Write your Schematron patterns **inside** the `</schema>` root element.
- Use the `tei:` prefix before each of your elements since we are working with a document in the TEI namespace. Remember that we do **not** use that prefix before attributes because attributes are in no namespace.

Analysis of the task

The goal:

The Dickinson project team is using TEI `<app>` elements inside the lines of Dickinson's poems when they need to encode a set of *variant* words or phrases that appear in different publications, labeled in the `<rdg>` elements with their `@wit` attributes. You are working with a single file representing a set of poems from a collection of manuscripts or *fascicle* that Emily Dickinson bundled and bound together herself. For this assignment, you will write Schematron to function on top of the established TEI Relax NG Schema to help ensure that the `<app>` and `<rdg>` elements are written properly according to the rules of the team. You will need to write a few rules to make sure that particular elements and attributes are appearing where we need them to, to make sure the poems are appearing in the proper order in this document (Poem 1 through Poem 11), and to control for missing or additional white spaces around our tags that might be distorting our representation of the poems.

Creating the rules step by step:

1. **Make sure each `<rdg>` element has no other attribute but `@wit`.** We want to make sure that the `<rdg>` element has nothing but an `@wit` attribute, but it *must* have this attribute. (The TEI schema by itself will allow other attributes or no attributes at all on this element, but we want to make sure that our project team only uses just this `@wit` attribute and not others.) Consider that we want the Schematron to tell us when `@wit` is *missing* and decide whether you need to write an `<assert>` or a `<report>` rule for this. Our solution uses the `not()` function in `@test` to fire if the `<rdg>` does not have `@wit` (including if it has any attribute other than `@wit`).

   Remember to use the `tei:` prefix before your element names! (Examples: `tei:app` and `tei:rdg`)

2. **Make sure there are one or more `<rdg>` elements inside an `<app>` element.** For this rule we want to check that an `<app>` element has a *count* of at least one or more than one `<rdg>` element. **Note:** For every new rule that matches in some way on the `<rdg>` context, you need to position it inside a new Schematron `<pattern>` element because otherwise only the first rule at a given context will fire and the others will remain passive.

3. **Make sure all of the poems are in the correct counting order within the document.** The team used XSLT to combine eleven separate documents, each holding a single poem, into the one XML file you are working with to hold the entire collection. However, the poems may not have transferred over in the correct order. For example, maybe Poem 6 comes after Poem 7 instead of coming directly after Poem 5. The rule that we will create now will help to check if the poems are in order so you can rearrange them if they are not. For this, we need to do the following:

    - First we need to look over our XML document that is holding all of the poems and find out *where* all of the poem titles are located in the hierarchy. Those poem titles each hold a number (1 to 11) that indicates where they properly sit in the sequence of the collection. We know that these `<title>` elements are positioned inside the `div/head` of each poem. Notice that each begins with the same pattern of text: the word "Poem" followed by a white space and a one or two-digit number.

    - Think about what we need to do: We want to make sure that these poems are in the correct order, based on the number given in their title. Is the poem titled Poem 2 immediately following the one titled Poem 1? If we look ahead and evaluate each poem's title in relation to the one that follows it, we only want to look at poems *that are followed by another poem*. (The last poem will not have a poem following after it to compare, but it will already be worked into the test because it will be the poem following the second-to-last poem.) Write your Schematron `<rule>` element and set its `@context` accordingly. (Our solution sets the `@context` at the `title` position. Think about whether you want to use the `following-sibling::` or the `following::` axis. Either way, you will need to compare a number in the `<title>` element of a current poem to that of the **first**, immediately-following poem.)
    - Decide whether you want to write an `<assert>` or `<report>` test that isolates the number of the poem inside the title element at your context. Our solution uses the `number()` function to convert the numeral(s) into a literal number, and then adds `+ 1` to test if that value equals the number of the poem given in the next following poem `div` (stepping down into *its* `title` element to isolate and convert and read *its* number). (Think about why we need to add **+ 1** here, or perhaps alternative ways you could write this test.)
    - You need to isolate just the number after the word “Poem” in the title, and to do this you need the `substring()` function (which you may wish to look up in Michael Kay or w3schools to see how this is formatted). The `substring()` function takes three arguments. The first argument indicates the XPath node (so if you set your rule context at the title, you would just invoke the `self::*` or dot(`.`) as the first argument). The second and third arguments are numbers: The second argument gives the numerical position of the character in the whole string of text that indicates the point where you want to start extracting your substring (so for this, count over from the start of the title to the first digit you want). The third argument indicates *how many* characters you want to extract into your substring. So the function is set up like this:

      ```
      substring(XPath, character-position-number-to-start, number-of-characters-to-extract)
      ```

      Note: since we have 11 poems, we are going to need to extract two characters to deal with Poem 10 and Poem 11.
    - Wrap your `substring()` in a `number()` function to convert it, and now work with it *as* a number. Add `+ 1` to it, and see if that value, `(substring() + 1)` equals the `substring()` in the title of *just* the very next poem in the sequence.
    - Test your rule. Our file is deliberately out of sequence, so you can expect to see errors if your rule is firing correctly.
4. **Test the values of the `@wit` attributes sitting on the `rdg` elements to be sure they are not mistyped.** This is something you are likely to need in your projects, so we direct you to our special Schematron tutorial on testing unique identifiers, which shows you how to work with `@xml:id`s (unique identifiers) and their corresponding referencing attributes. Can you adapt the code in our tutorial to work with this file and its positioning of the list of witnesses in this document?

5. **Optional Challenge: Control the white space around the <app> and <rdg> elements in a line of poetry.** As the team works on coding these poems, it is very easy for them to accidentally remove or add white space in applying <app> and <rdg> elements. It is very easy to make two words run together by accident, for example, by coding like this:

```
<l n="1">When we stand on the tops of<app>
 <rdg wit="#df16">Things—</rdg>
 <rdg wit="#bm">things</rdg>
</app>
</l>
```

Notice that there is no space before the opening <app> tag and no space inside the opening <rdg> tag, so when the team transformed this to view the first witness in HTML, we saw something like this:
When we stand on the tops ofThings—
To deal with this, we need to recognize that sometimes we want a white space in between the main line of text and the starting <app> element, and sometimes we do not.

- We **do not want to add a space** when the line of text before <app> ends with white space already, when it has a special punctuation mark, a dash (–) or a quotation mark (`&quot;`) designed to connect with the text in the <rdg> element(s).
- We need to **add** a white space whenever the line of text before <app> ends with something *other than* the three characters we described above *and* the `rdg` element inside begins with a letter (another alphabet character).
- We might have to **remove** an extra white space when the line of text before <app> ends with *any non-space character followed by a space*, and the <rdg> element opens with a white space.
- You may also want to test for white space at the end of an rdg element when its `parent::app` is *followed by* a string of text.

For our purposes, if you can write a Schematron rule that addresses even just one of the above scenarios, that is sufficient, though we hope that if you succeed with one test, you will figure out how to write one or two others! To control for white space, we created a pattern with a single rule set on the `@context` of the `tei:rdg` element, because we need to look at each <rdg> element in turn to see if we have a white space problem, and there are often multiple <rdg> elements inside each line. When we set the context to the whole line of poetry, it might have multiple sets of <app> elements inside, and we cannot write a precise enough rule to address the spans of text we need. To proceed, we need to understand something about **mixed content**: When an element like the TEI `l` (for a line of poetry) contains a mixture of `text()` and other elements, the `text()` node is sitting in a **sibling** relationship to the nested elements, so that a span of text in a line of a Dickinson poem is sitting on the `preceding-sibling::` axis in relation to the <app> element that follows it. If you write your rule as we did, from the context of the <rdg> elements, you will need to write your test to reach up to to the parent <app> elements and walk over to the `preceding-sibling::text()` node. We use the `matches()` function in our Schematron `@test` because it works with regular expressions and helps us to identify the particular conditions we are looking for. (Look up this function in one of the sources we list in the Preliminaries section of this assignment to be sure you understand how to write it.) Specifically, we are going to need a *two-part test*, and we can use the `matches()` function twice, joined by the word `and` to see first if a) the line of text that is the first preceding-sibling of our parent <app> *ends with* something in a regex character set, **and** b) the contents of our context <rdg> element *starts with* something in a regex character set, like this:

```
test="matches(. . .) and matches(. . .)"
```

or

```
test="not(matches(. . .) and matches(. . .)"
```

or some combination of these.

Note that the `matches()` function takes two arguments like this: `matches(Xpath-location, 'regex-pattern')`. You might be wondering why we aren't using the functions `starts-with()` or `ends-with()`. The answer is that these do not help us with finding regular expressions, but `matches()` can look for a regex pattern *wherever we need it*. To designate the **start of a line** in regex (or the start of the text in a given XPath node, use the regex caret, `^`, at the start of the regex pattern you are hunting for, and to designate **the end of the text**, use the regex dollar sign, `$` at the end of your regex pattern.

Bonus task: You will likely have difficulty with matching on a quotation mark, because if you try to include it literally in the character set (or even escape it), it will be interpreted as the end of the schematron attribute and will result in a formedness error, munging your Schematron code. Consider it a bonus task on this assignment to find a way to match on a straight quotaton mark. Hint: you will need to escape the literal quotation mark using `&quot;`, but you won't be able to include it in a `[ ]` character set.

See how far you can get with this Optional Challenge Task and if you get stuck, record what you tried and what didn't work. Do your tests fire? You should see some white space errors in the file as we presented it, but you should also tinker with the white space just before an `<app>` tag and at the start of an `<rdg>` element.

Submission

Upload your completed Schematron schema AND the Dickinson poems XML ***with your Schematron associated*** to Courseweb, and follow our standard filenaming conventions for homework assignments uploaded to Courseweb.

newtFire {dhlds}
Maintained by: Elisa E. Beshero-Bondar (ebb8 at pitt.edu) (cc) BY-NC-SA Last modified: Monday, 16-Oct-2017 20:01:37 EDT. Powered by firebellies.

Schematron Exercise 2

Meet Schematroll, the Schematron mascot! Schematroll is a cross between a bilby and a bettong.

Preliminaries

To work on this assignment, you will need to to find and do the following:

- **Information resources at the ready:** Review our Schematron tutorial, and read more about the XPath functions and syntax we describe below either on the web (see w3Schools' "XSLT, XPath, and XQuery Functions", Obdurodon's "The XPath Functions We Use the Most") or through offline searching with the index of the Michael Kay book. You also want to read our tutorial on validating id attributes.
- **XML file to test:** Right-click to save this TEI file locally and open it in <oXygen/>: **Sample for Digital Mitford Site Index**. You will need to associate your Schematron file with this document **in addition to** the currently associated TEI schema lines.
- Open a new Schematron document in <oXygen/> by going to **File → New** and typing "Schematron" in the "Type filter text" box, or by going to **File → New → New Document → (scroll to Schematron in the alphabetized list) → Schematron**. Once opened, you will keep the default xml line at the top, but you will delete everything from `<sch:schema>` down. To write Schematron rules for a document in the TEI namespace, you will then replace this with:

```
<schema xmlns:sch="http://purl.oclc.org/dsdl/schematron" queryBinding="xslt2"
    xmlns:sqf="http://www.schematron-quickfix.com/validator/process"
    xmlns="http://purl.oclc.org/dsdl/schematron">
        <ns uri="http://www.tei-c.org/ns/1.0" prefix="tei"/>

    </schema>
```

- Write your Schematron patterns **inside** the `</schema>` root element.
- **Important:** You must use the `tei:` prefix before each of your elements since we are working with a document in the TEI namespace; otherwise none of your schema rules involving elements will fire! Remember that we do **not** use that prefix before attributes because attributes are in no namespace.

Analysis of the task

The goal:

The Digital Mitford project is working on a collection of prosopography data, that is, a record of people, places, organizations, published works, and other named entities relevant to British author Mary Russell Mitford's world in the nineteenth century. After some years of collaborative research the collection (which we call our "Site Index") contains thousands of entries, all contributed in batches by members of the editing team in the course of their research. It's common for our editors to make typographical errors as they enter details about historical people in particular, since these entries can be especially complicated! Your task is to write a helpful Schematron file to guide the editors in their process, flag errors if they reverse date ranges like birth and death dates, check for white space errors and other common problems, and check to see that the referencing of `@xml:id` attributes is correct. We hope that learning these things will give you ideas for writing Schematron to guide your own projects.

As you work on the rules below, think about how to group them logically into related `pattern` elements. You can use an `@id` on `pattern` elements to help label them and organize your work. Also, be sure to associate your Schematron file with the XML file you are testing *as soon as you write your first rule* so you can test it to make sure it is working.

A little orientation

Skim through the Digital Mitford project XML you downloaded, and get a sense of how it is organized and the way we have nested information about individuals inside each `person` element. You will see that each `person` has an `@xml:id` whose value is a distinct identity marker. Inside the `person` elements you will see `persName` elements, some of which contain nested `surname`, and `forename` elements. You will also see elements for `birth` and `death` with attributes and contents telling us about when and where a person was born and died. And most `person` elements contain a biographical `note` element with more information. These notes sometimes include references (made with `@ref` attributes) to people, places, books, and more listed elsewehere in the site index.

Rules to write and test

1. We want to close up extra white spaces that our editors inevitably type at the start of their elements. Write a Schematron rule that checks for leading white space inside the `tei:persName` element in particular. (That is, raise a warning when an element *starts with* a white space.) **Hints**:
    - You may want to look up the `starts-with()` function, one of the family related to `contains()`. If you would rather "play with `matches()`", the `matches()` function can handle this too, as long as you know how to write regex to find the start of a node. (Hint for safely "playing with matches": Remember the regular expression `^` and `$`? In XPath contexts, they refer to the start or end of an XML node, instead of the start or end of a line of text.)
    - One thing you will notice in writing these string-matching functions is that you need to represent the *haystack* (in this case, each XML node you're checking), followed by the *needle* (or the thing you're looking to find inside), and when that needle is a *literal string* as in with starts-with() , or a *regex pattern* as in matches(), you need to wrap it in quotation marks. But in the context of writing Schematron, your tests are written as the **value** of the attribute `@test`, so they must *already* be inside quotation marks: a NEW set of quotation marks inside is going to throw your computer off so it will not know how to find the end of your attribute value: and your computer will throw a *well-formedness* error if you use the same *kind* of quotation marks. So, we switch over to **single** quotation marks when we need to use quotes inside functions like we do here:

      ```
      <report test="starts-with(., ' ')">
      ```

      This practice is called *nesting* your quotation marks, and we use it in ordinary writing, too! In XML code and in formal editorial practice, we alternate between double and single quotation marks to nest them in layers.

2. Let's work on some Schematron tests for the `tei:person` element. We want to check the way its *`@xml:id`* is written. In our project when a historical person is given a unique identifier, that `@xml:id` value is supposed to begin with the most distinctive part of the person's name, their *last* name. Since we code the `tei:surname` element as a descendant of `tei:person`, you may write a Schematron rule that tests whether the `@xml:id` *starts with* the contents of the TEI's surname element. **Hint:** You are used to writing `starts-with()` and related functions so that they look for literal strings of text or regex patterns, but you can *also* use these functions to locate the contents of an element and make sure it matches up to what you see in an attribute. To locate *whatever is in an XML node* (element or attribute) instead of a specific string of text, simply do not use the quotation marks that indicate a string.

3. Sometimes our editors don't capitalize proper names! Check that all the `tei:forename`, `tei:surname`, and `tei:placeName` elements, as well as any `tei:persName` elements that hold text and do not wrap around forename and surname elements start with capital letters. **Hints:**
    - You can do that with one rule, and you can set multiple contexts using the **union operator** or pipe: `|` to join these together. You last used the pipe when writing Relax NG. You can use it in Schematron (and XSLT) contexts here specifically to join together multiple context items in one rule.
    - You actually DO need to "play with `matches()`" this time, because you need to find a regular expression pattern at the start of each node. The `starts-with()` function looks only for literal strings, not regex patterns. (We'll repeat our Hint for safely "playing with matches" in case you didn't read it on number 1: Remember the regular expression `^` and `$`? In XPath contexts, they refer to the start or end of an XML node, instead of the start or end of a line of text.)

4. Now let's take a look at the dates coded in this file, coded in the `tei:birth` and `tei:death` elements. All death dates need to be later than birth dates, but surprisingly, the TEI does not have a built-in way of checking this. Write a Schematron rule to flag when the dates coded in the `@when` attributes on any `tei:birth` and `tei:death` elements don't make sense. **Hints:**
    - We use a few different kinds of dating attributes here: `@notBefore`, `@notAfter`, and `@when`, depending on how certain we are of when a birth or death occurred. For the purposes of this homework, it is fine to **concentrate only on the `@when` attributes** coded on `tei:birth` and `tei:death`.
    - **How to test for this:** Some dates are given as full ISO years (`yyyy-mm-dd`) and others are only partial and those, alas, will NOT convert to a machine-readable date with `xs:date()`, so we do not want to use that function here. Instead, we recommend that you work with the `tokenize()` function to isolate the year as the piece that we really need to look at, that is, the four-digit year that sits in front of the first hyphen. To reliably capture this piece, write the `tokenize()` function to break the attribute values in pieces around hyphens ("tokenize on the hyphen") and write a position predicate to grab the *first* of the tokens. (Note: tokenize() is a wonderfully adaptable function! Even if the date value lacks any hyphens and only contains a year, this will still return that year since the token just won't break off!)
    - Remember, you are testing to see when a *birth year* is later than a *death year*, so you need to write a test that uses comparison operators, like you did in Schematron Exercise 1.
5. For the last required task in this assignment, it is very important for our site index file that `@ref` attributes must begin with a leading hashtag (#), since (as we explain more fully in our guide on "Coding with Unique Identifiers and Testing Them with Schematron"), the hashtag is reserved for `@ref` attributes that *point) to* `@xml:ids`, so they do not duplicate those ids (whose values should only ever turn up once in a project). Write Schematron rule(s) to test and flag those errors on our `@ref` attributes, to help us find where these are missing their required hashtags.
6. **Optional Bonus Challenge:** These last two tasks are challenging, but may be useful to adapt in projects, so if you do not have time to write them now, you may wish to come back to them later on. To work on these, you need to consult our guide on "Coding with Unique Identifiers and Testing Them with Schematron". Finally, carefully following our guide, adapt the code we provide there to write a test that checks whether the `@ref` and `@resp` attribute values, *following their hashtags*, actually match up to a defined `@xml:id` in this file *or* in the Digital Mitford Site Index at `http://digitalmitford.org/si.xml`. (Note that this rule will also ensure that these values actually begin with a hashtag!) Following our guide, you will learn how to write a `let` statement to define a variable that points to another file's `@xml:ids`, and then *refer* to that variable in your Schematron test. Also, it is perfectly legal in our project for there to be *multiple* values on an `@ref` or `@resp`, separated by white space, just as you see in our guide, so you should follow our lead to adapt our code there.

7. **Optional Bonus Challenge:** We need a more sophisticated way than we used in number 3 to check the way people type out full names in the `persName` elements. Can we test for errors like these?

   ```
   Dorothy wordsworth
   ```

   or

   ```
   Percy bysshe Shelley
   ```

   Of course we can, by adapting the `tokenize()` we have been using here to break on white space, and to test **each** token in turn to see if it is capitalized. You can do this by applying the `for $i in (sequence) return …` (or "**for-loop**" XPath feature) so we can walk through each token in the full sequence. To see how to write the code, consult our our guide on testing unique identifiers: Look at our `let` statement, defining a variable containing a sequence of tokens, and then consider how we processed each one in turn in our `assert @test`. Can you adapt that code to tokenize the parts of a name, and test to see if each part is capitalized? Write your Schematron rule!

Submission

Upload your completed Schematron schema AND the si-Add-MRMsample.xml file **with your Schematron associated** to Courseweb, and follow our standard filenaming conventions for homework assignments uploaded to Courseweb.

Mulberry Classes Guide to Using the Oxygen XML Editor (v20.0)

Mulberry Technologies, Inc.
17 West Jefferson Street, Suite 207
Rockville, MD 20850
Phone: 301/315-9631
Fax: 301/315-8285
info@mulberrytech.com
http://www.mulberrytech.com

Version 1.8 (March 23, 2018)
©Copyright 2015-2018 Mulberry Technologies, Inc.

Mulberry Classes Guide to Using the Oxygen XML Editor (v20.0)

Exhibits

Guide to Using Oxygen XML Editor (v20.0)

NOTE: This is a reference, not a list of instructions!

Oxygen is both an XML editor and a development tool. We will be using it to run XML transforms using XSLT, to validate documents according to a DTD or schema, and to run Schematron, XQuery, XSLT-FO, and other processes.

Key Oxygen Icons

 check well-formedness (blue checkmark)

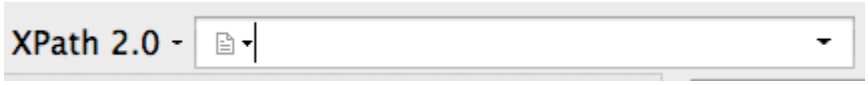
 validate document (red checkmark)

 associate schema (red push pin)

 apply transformation scenario (triangle in circle)

 configure transformation scenario (wrench)

XPath 2.0 search window



Open Oxygen XML Editor

- Double click the icon 

Naming Files

When you create a file, it is considered best practice to name your files using the following file extensions:

- XML filenames end in "`.xml`"
- XSLT filenames end in "`.xsl`"
- XML Schema filenames end in "`.xsd`"

- DTD filenames end in "`.dtd`"
- DTD modules (DTD fragments) end in "`.ent`" or "`.mod`"
- Schematron filenames end in "`.sch`"
- PDF files end in "`.pdf`"
- HTML an XHTML files end in "`.html`" or "`.htm`"
- RELAXNG files end in "`.rng`"

Create a New XML Document

1. ***First Time Opening Oxygen***
   - If a "Welcome to Oxygen" screen appears, under `Create New`
     - Choose `New Document`
     - Choose `XML Document`
     - Then finish as explained below
   - If there is no "Welcome to Oxygen" screen, on the top bar choose `File`
     - Choose `New`
     - Under `New Document`, choose `XML Document`
     - Then finish as explained below
2. ***If Oxygen is Already Open***
   - On the top bar choose `File`
   - Choose `New`
   - Under `New Document`, choose `XML Document`
   - Then finish as explained below
3. ***Finish New Document: Associate a Schema***
   - Click `Customize`
     - On the fill-in line `Schema URL:`, click the small down arrow 
     - Choose `Browse for local file`

- An `Open` window will pop up
- Select the DTD, RNG, or XSD schema you want
- Click `Open`
- The `Schema type` and the `Root element:` should fill in automatically (If the `Root element:` does not show the correct root element, use the drop-down menu to scroll to the root.)
- Click `Create`

Outline View (See the Tree)

The Outline view shows the tree view of your document, and collapses and expands like a word-processing outliner by clicking on the plus right-facing triangles.

- Open your `.xml` document in Oxygen
- From the top line options choose `Window`
- Pull down to `Show View`
- Choose `Outline`

Check XML Document for Well-formedness

A well-formed XML document, follows all the syntax rules of XML

- Open your `.xml` document in Oxygen
- To the right of the red check mark  is a tiny down-pointing arrow. Click that arrow and then choose the Well-formedness icon 
- Test results show at the very bottom of the screen:
  - A green box with the words "`Document is well formed`" tells you that the document is well-formed.
  - A red box with words like "`Wellformed test - failed`" says there are errors. Each error will be described in the error window below the screen and by a red bar on the vertical status line. Clicking on the error message or the red bar will take you to the location of the error.

Check XML Document for Validity

- If there is a schema associated with your document, click on the Validation icon  . For DTD validation, this means there is a DOCTYPE declaration.
- If there is no schema associated, associate an XSD, RNG, or DTD schema with the document as follows:
  - Click the Associate Schema  icon just above the document
  - Click on the folder icon  the right of the `URL:` fill-in box
  - Choose `Browse for local file`
  - An `Open` window will pop up
  - Select the DTD, RNG, or XSD schema you want
  - Click `Open`
  - Click `OK` to choose your DTD or Schema
  - Click on the Validation icon 
- Validation messages
  - A green box on the bottom line with the words "`Validation successful`" will tell you that there were no parsing errors.
  - A red box on the bottom line with words like "`Validation failed`" will tell you that you have errors. Each error will be described in the error window below the screen and by a red bar on the vertical the status line. Clicking on the error message or the red bar will take you to the location of the error.

GO TO an Error in an XML File

Each error is described at the bottom of the screen in an error window and as a red bar on the status bar to the right of the main window. Go directly to the error by:

- Clicking on the error message, or
- Clicking on the red bar

See All of an Error Message

Each error is described at the bottom of the screen in an error window and as a red bar on the status bar to the right of the main window. To see the full message you may:

- Right click on the error message and choose `Show message`, or
- Scroll the error bar

Associate a Stylesheet (Run XSLT transform through Oxygen)

To run a XSLT transformation (for example, to transform your XML into HTML that you can see in a browser):

- Open your XML document in Oxygen
- ***First time setup*** —The first time you create the transform: Click on the Configure Transformation Scenario icon [a crescent wrench with small right-pointing red triangle] 
  - Click the `New` button near the bottom of the menu
  - Then choose `XML transformation with XSLT`
  - A `New scenario` window will pop up
  - In the `Name:` fill-in, give your scenario a name
  - Under the `XSLT` tab:
    - Leave the `XML URL` alone (`${currentFileURL}` names the file that you have open)
    - In the `XSL URL` box, click on the open folder  and choose the appropriate `.xsl` stylesheet file
  - Ignore the `FO Processor` tab
  - Under the `Output` tab:
    - Click `Prompt for file`
    - Click `OK`
  - Click the button `Apply associated(1)`

- You will be prompted for a file name (which will have your results in it) and a preview of your results will show in the bottom window

- ***Run an existing transform*** — *Two* ways to run a transform that has already been set up:

1. For the *last scenario used* (where the scenario still has a check mark next to its name):
   - Click on the Apply Transformation Scenario icon [large right pointing red triangle] 
   - The previously selected scenario will run
2. For a scenario listed among Oxygen's transforms, but *not previously selected* (no check mark next to the scenario name):
   - Click on the Apply Transformation Scenario icon [large right pointing red triangle] 
   - A `Transform With` window will appear
   - Choose the scenario you want (click on the name)
   - Click `Edit`
   - Look at the scenario to see what it is doing, then click `OK`
   - Add a check mark to the left of the scenario you have selected (click in the box to the left of the name)
   - Click the button `Apply associated(1)`

Checking Selected Aspects using Schematron

(This is one way; there are many others, including techniques for running Schematron against many files at one time.)

- Open your XML file in Oxygen
- Click on the small gray down-arrow just to the right of the Validate Document red check mark  . Choose `Validate with`.
- In the `Schema type` tab, select `Schematron`.

- Choose the folder icon to the right of the URL box and navigate to the file that is your Schematron schema. Double click on that filename to choose the `.sch` file. Use relative paths.
- Click `OK`
- Any Schematron error (`E [ISO Schematron]`) and warning (`W [ISO Schematron]`) messages will appear in a window at the bottom of your screen; otherwise, you will see a green box and "`Validation successful`" message.

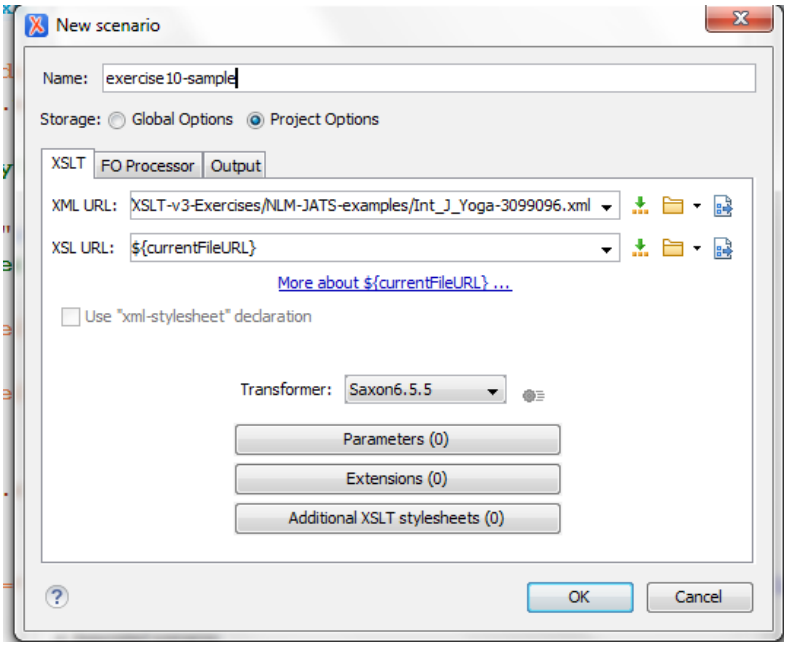
Turn Content Completion On/OFF in Oxygen

- Select `Options` in the top bar
- Choose `Preferences`
- Scroll down to and click on `Editor`, then find `Content Completion`
  - To turn `Content Completion` on: Select the top three items (other options will appear), select `Apply`, and then select `OK`
  - To turn `Content Completion` off: Deselect the top three items (other options in the pane will then also be grayed out), select `Apply`, and then select `OK`

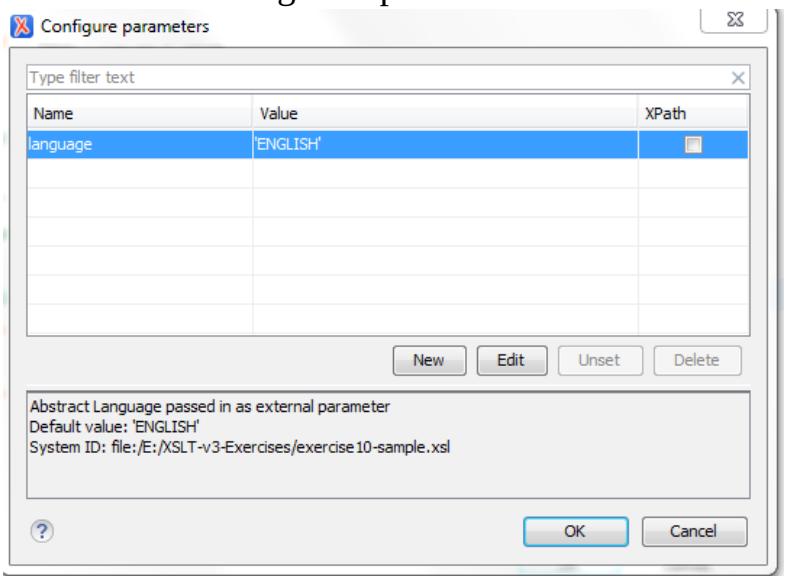
To Pass a Global Parameter to a Stylesheet in Oxygen

Often, we wish to pass global, externally-supplied parameters to XSLT transformations. This can be done when setting up or editing an Oxygen XSLT transformation.

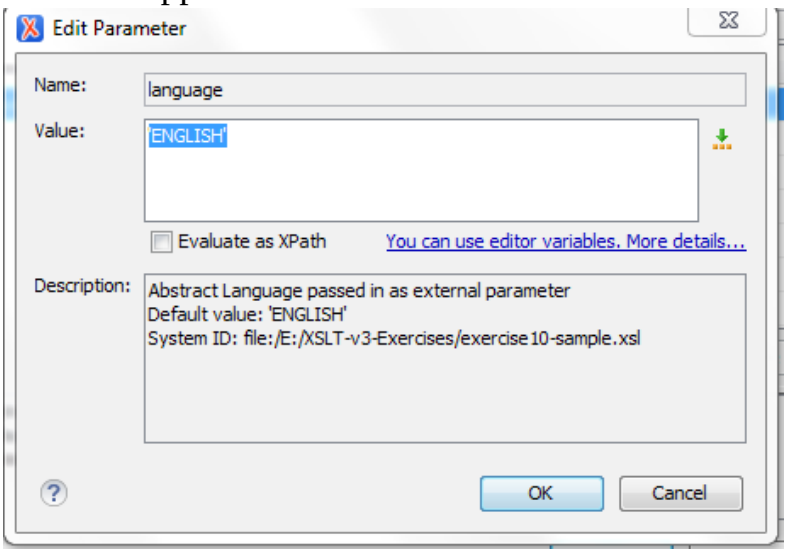
- When constructing the transformation, select the `Parameters` button (near the bottom of the `New Scenario` window).



- A `Configure parameters` window will appear. If supplied in the XSLT code, the name of a parameter and its default value will be listed. (Oxygen knows to read the global parameter values from the XSLT code.)

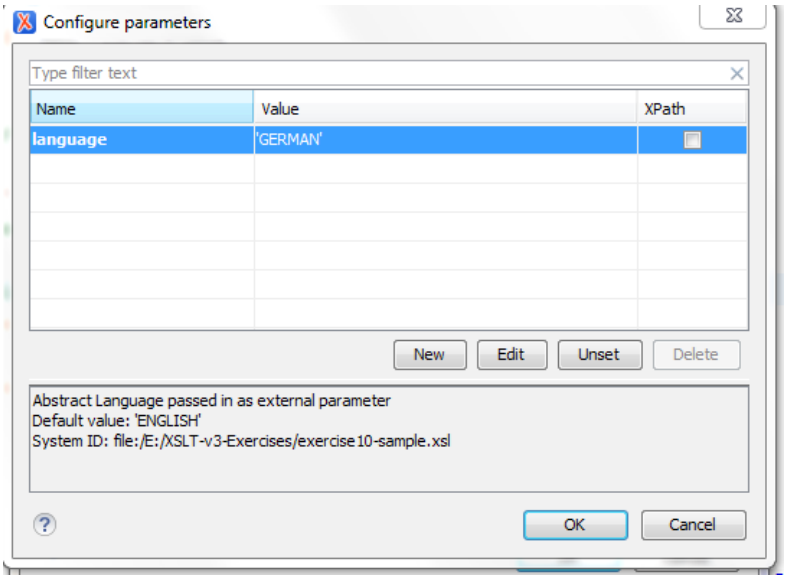


- To change the value of the parameter, place the cursor on the desired parameter line, select it, and click the `Edit` button. An `Edit parameter` window will appear:

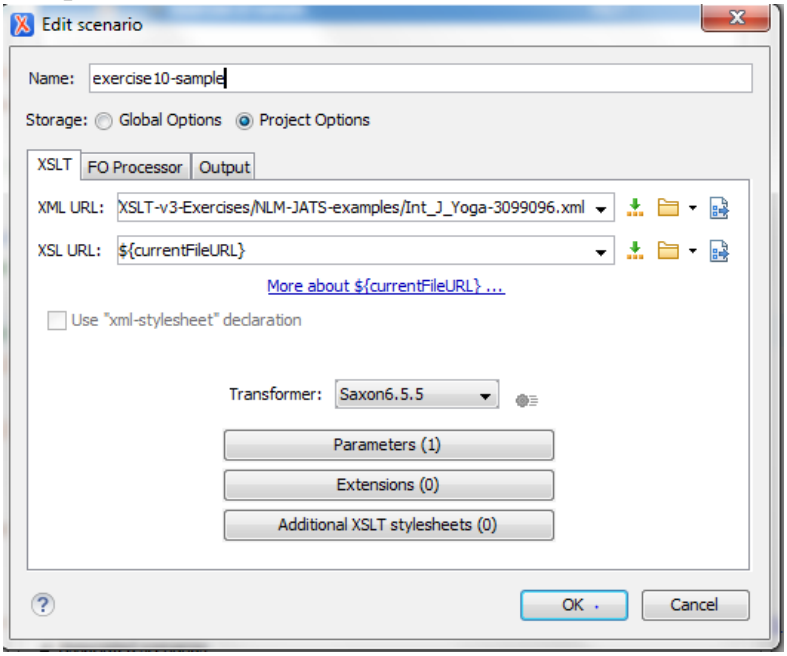


- In the `Value:` pane, select the text and type the desired value over the existing value, e.g., type the new value “GERMAN” over the existing value “ENGLISH”. (The value does NOT need quotation marks — Oxygen will add those.) Click `OK`.

- The `Configure parameters` window will reappear and include the new default value. Click `OK`.



- The `Edit scenario` window will reappear and indicate that one parameter is now being passed in from Oxygen. Click `OK` to exit the transformation setup.



Date/Time Functions

adjust-date-to-timezone(xs:date?) **as** xs:date?

adjust-date-to-timezone(xs:date?, xs:dayTimeDuration?) **as** xs:date?

adjust-dateTime-to-timezone(xs:dateTime?) **as** xs:dateTime?

adjust-dateTime-to-timezone(xs:dateTime?, xs:dayTimeDuration?) **as** xs:dateTime?

adjust-time-to-timezone(xs:time?) **as** xs:time?

adjust-time-to-timezone(xs:time?, xs:dayTimeDuration?) **as** xs:time?

dateTime(xs:date?, xs:time?) **as** xs:dateTime?

day-from-date(xs:date?) **as** xs:integer?

day-from-dateTime(xs:dateTime?) **as** xs:integer?

days-from-duration(xs:duration?) **as** xs:integer?

hours-from-dateTime(xs:dateTime?) **as** xs:integer?

hours-from-duration(xs:duration?) **as** xs:integer?

hours-from-time(xs:time?) **as** xs:integer?

implicit-timezone() **as** xs:dayTimeDuration

minutes-from-dateTime(xs:dateTime?) **as** xs:integer?

minutes-from-duration(xs:duration?) **as** xs:integer?

minutes-from-time(xs:time?) **as** xs:integer?

month-from-date(xs:date?) **as** xs:integer?

month-from-dateTime(xs:dateTime?) **as** xs:integer?

months-from-duration(xs:duration?) **as** xs:integer?

seconds-from-dateTime(xs:dateTime?) **as** xs:decimal?

seconds-from-duration(xs:duration?) **as** xs:decimal?

seconds-from-time(xs:time?) **as** xs:decimal?

timezone-from-date(xs:date?) **as** xs:dayTimeDuration?

timezone-from-dateTime(xs:dateTime?) **as** xs:dayTimeDuration?

timezone-from-time(xs:time?) **as** xs:dayTimeDuration?

year-from-date(xs:date?) **as** xs:integer?

year-from-dateTime(xs:dateTime?) **as** xs:integer?

years-from-duration(xs:duration?) **as** xs:integer?

XPath 2.0:
http://www.w3.org/TR/xpath20/
XQuery 1.0:
http://www.w3.org/TR/xquery/
XQuery 1.0 & XPath 2.0 Functions & Operators:
http://www.w3.org/TR/xpath-functions/

XSLT-Only Functions

current() **as** item()

current-group() as item()*

current-grouping-key() **as** xs:anyAtomicType?

document(item()*) **as** node()*

document(item()*, node()) **as** node()*

element-available(xs:string) **as** xs:boolean

format-dateTime(xs:dateTime?, xs:string, xs:string?, xs:string?, xs:string?) **as** xs:string?

format-dateTime(xs:dateTime?, xs:string) **as** xs:string?

format-date(xs:date?, xs:string, xs:string?, xs:string?, xs:string?) **as** xs:string?

format-date(xs:date?, xs:string) **as** xs:string?

format-number(numeric?, xs:string) **as** xs:string

format-number(numeric?, xs:string, xs:string) **as** xs:string

format-time(xs:time?, xs:string, xs:string?, xs:string?, xs:string?) **as** xs:string?

format-time(xs:time?, xs:string) **as** xs:string?

function-available(xs:string) **as** xs:boolean

function-available(xs:string, xs:integer) **as** xs:boolean

generate-id() **as** xs:string

generate-id(node()?) **as** xs:string

key(xs:string, xs:anyAtomicType*) **as** node()*

key(xs:string, xs:anyAtomicType*, node()) **as** node()*

regex-group(xs:integer) **as** xs:string

system-property(xs:string) **as** xs:string

type-available(xs:string) **as** xs:boolean

unparsed-text(xs:string?) **as** xs:string?

unparsed-text(xs:string?, xs:string) **as** xs:string?

unparsed-text-available(xs:string?) **as** xs:boolean

unparsed-text-available(xs:string?, xs:string?) **as** xs:boolean

unparsed-entity-uri(xs:string) **as** xs:anyURI

unparsed-entity-public-id(xs:string) **as** xs:string

Argument Notation

numeric	Any of xs:integer, xs:decimal, xs:float or xs:double.
*	A sequence of the indicated type.
?	The indicated type or empty sequence.
~	The result type varies depending on the arguments.
xs:	http://www.w3.org/2001/XMLSchema

2008-07-21

XQuery 1.0 & XPath 2.0 Functions & Operators Quick Reference

Sam Wilmott
sam@wilmott.ca
http://www.wilmott.ca

and

Mulberry Technologies, Inc.
17 West Jefferson Street, Suite 207
Rockville, MD 20850 USA
Phone: +1 301/315-9631
Fax: +1 301/315-8285
info@mulberrytech.com
http://www.mulberrytech.com

© 2007-2008 Sam Wilmott and Mulberry Technologies, Inc.

Date/Time Operators

(xs:date) + (xs:dayTimeDuration) **as** xs:date

(xs:date) + (xs:yearMonthDuration) **as** xs:date

(xs:dateTime) + (xs:dayTimeDuration) **as** xs:dateTime

(xs:dateTime) + (xs:yearMonthDuration) **as** xs:dateTime

(xs:dayTimeDuration) + (xs:dayTimeDuration) **as** xs:dayTimeDuration

(xs:time) + (xs:dayTimeDuration) **as** xs:time

(xs:yearMonthDuration) + (xs:yearMonthDuration) **as** xs:yearMonthDuration

(xs:date) - (xs:date) **as** xs:dayTimeDuration

(xs:date) - (xs:dayTimeDuration) **as** xs:date

(xs:date) - (xs:yearMonthDuration) **as** xs:date

(xs:dateTime) - (xs:dateTime) **as** xs:dayTimeDuration

(xs:dateTime) - (xs:dayTimeDuration) **as** xs:dateTime

(xs:dateTime) - (xs:yearMonthDuration) **as** xs:dateTime

(xs:dayTimeDuration) - (xs:dayTimeDuration) **as** xs:dayTimeDuration

(xs:time) - (xs:dayTimeDuration) **as** xs:time

(xs:time) - (xs:time) **as** xs:dayTimeDuration

(xs:yearMonthDuration) - (xs:yearMonthDuration) **as** xs:yearMonthDuration

(xs:dayTimeDuration) * (xs:double) **as** xs:dayTimeDuration

(xs:yearMonthDuration) * (xs:double) **as** xs:yearMonthDuration

(xs:dayTimeDuration) **div** (xs:dayTimeDuration) **as** xs:decimal

(xs:dayTimeDuration) **div** (xs:double) **as** xs:dayTimeDuration

(xs:yearMonthDuration) **div** (xs:double) **as** xs:yearMonthDuration

(xs:yearMonthDuration) **div** (xs:yearMonthDuration) **as** xs:decimal

The **eq, ne, lt, gt, le** and **ge** comparisons are suppoted for the types: **xs:date** and **xs:time**.

The **eq** and **ne** (only) comparisons are supported for the types: **xs:duration**, **xs:gDay**, **xs:gMonth**, **xs:gMonthDay**, **xs:gYear** and **xs:gYearMonth**.

The **lt, gt, le** and **ge** (only) comparisons are supported for the types: **xs:dayTimeDuration** and **xs:yearMonthDuration**.

Other Comparisons

The **eq** and **ne** (only) comparisons are supported for the types: **xs:base64Binary**, **xs:hexBinary**, **xs:NOTATION** and **xs:QName**.

Text/String Functions

codepoint-equal(xs:string?, xs:string?) **as** xs:boolean?
codepoints-to-string(xs:integer*) **as** xs:string
compare(xs:string?, xs:string?) **as** xs:integer?
compare(xs:string?, xs:string?, xs:string) **as** xs:integer?
concat(xs:anyAtomicType?, xs:anyAtomicType?,) **as** xs:string
contains(xs:string?, xs:string?) **as** xs:boolean
contains(xs:string?, xs:string?, xs:string) **as** xs:boolean
current-date() **as** xs:date
current-dateTime() **as** xs:dateTime
current-time() **as** xs:time
default-collation() **as** xs:string
encode-for-uri(xs:string?) **as** xs:string
ends-with(xs:string?, xs:string?) **as** xs:boolean
ends-with(xs:string?, xs:string?, xs:string) **as** xs:boolean
escape-html-uri(xs:string?) **as** xs:string
lower-case(xs:string?) **as** xs:string
normalize-space() **as** xs:string
normalize-space(xs:string?) **as** xs:string
normalize-unicode(xs:string?) **as** xs:string
normalize-unicode(xs:string?, xs:string) **as** xs:string
starts-with(xs:string?, xs:string?) **as** xs:boolean
starts-with(xs:string?, xs:string?, xs:string) **as** xs:boolean
string() **as** xs:string
string(item()?) **as** xs:string
string-join(xs:string*, xs:string) **as** xs:string
string-length() **as** xs:integer
string-length(xs:string?) **as** xs:integer
string-to-codepoints(xs:string?) **as** xs:integer*
substring(xs:string?, xs:double) **as** xs:string
substring(xs:string?, xs:double, xs:double) **as** xs:string
substring-after(xs:string?, xs:string?) **as** xs:string
substring-after(xs:string?, xs:string?, xs:string) **as** xs:string
substring-before(xs:string?, xs:string?) **as** xs:string
substring-before(xs:string?, xs:string?, xs:string) **as** xs:string
translate(xs:string?, xs:string, xs:string) **as** xs:string
upper-case(xs:string?) **as** xs:string

XSL-List:
http://www.mulberrytech.com/xsl/xsl-list

REGEX Functions

matches(xs:string?, xs:string) **as** xs:boolean
matches(xs:string?, xs:string, xs:string) **as** xs:boolean
replace(xs:string?, xs:string, xs:string) **as** xs:string
replace(xs:string?, xs:string, xs:string, xs:string) **as** xs:string
tokenize(xs:string?, xs:string) **as** xs:string*
tokenize(xs:string?, xs:string, xs:string) **as** xs:string*

Arithmetic Operators

+ (numeric) **as** ~numeric
(numeric) **+** (numeric) **as** ~numeric
- (numeric) **as** ~numeric
(numeric) **-** (numeric) **as** ~numeric
(numeric) ***** (numeric) **as** ~numeric
(numeric) **div** (numeric) **as** ~numeric
(numeric) **idiv** (numeric) **as** xs:integer
(numeric) **mod** (numeric) **as** ~numeric

Arithmetic Functions

abs(numeric?) **as** ~numeric?
avg(xs:anyAtomicType*) **as** ~xs:anyAtomicType?
ceiling(numeric?) **as** ~numeric?
floor(numeric?) **as** ~numeric?
number() **as** xs:double
number(xs:anyAtomicType?) **as** xs:double
round(numeric?) **as** ~numeric?
round-half-to-even(numeric?) **as** ~numeric?
round-half-to-even(numeric?, xs:integer) **as** ~numeric?
sum(xs:anyAtomicType*) **as** ~xs:anyAtomicType
sum(xs:anyAtomicType*, xs:anyAtomicType?) **as** ~xs:anyAtomicType?

The **eq, ne, lt, gt, le** and **ge** comparisons are supported for the numeric types.

Sequence Operators

(item()*) **,** (item()*) **as** ~item()*
(node()*) **union** (node()*) **as** ~node()*
(node()*) **intersect** (node()*) **as** ~node()*
(node()*) **except** (node()*) **as** ~node()*
(xs:integer) **to** (xs:integer) **as** xs:integer*

Node Comparisons

(node()) **is** (node()) **as** xs:boolean
(node()) **<<** (node()) **as** xs:boolean
(node()) **>>** (node()) **as** xs:boolean

Sequence and Node Functions

collection() **as** node()*
collection(xs:string?) **as** node()*
count(item()*) **as** xs:integer
data(item()*) **as** ~xs:anyAtomicType*
deep-equal(item()*, item()*) **as** xs:boolean
deep-equal(item()*, item()*, string) **as** xs:boolean
distinct-values(xs:anyAtomicType*) **as** ~xs:anyAtomicType*
distinct-values(xs:anyAtomicType*, xs:string) **as** ~xs:anyAtomicType*
doc(xs:string?) **as** document-node()?
empty(item()*) **as** xs:boolean
exactly-one(item()*) **as** ~item()
exists(item()*) **as** xs:boolean
index-of(xs:anyAtomicType*, xs:anyAtomicType) **as** xs:integer*
index-of(xs:anyAtomicType*, xs:anyAtomicType, xs:string) **as** xs:integer*
insert-before(item()*, xs:integer, item()*) **as** ~item()*
last() **as** xs:integer
nilled(node()?) **as** xs:boolean?
node-name(node()?) **as** xs:QName?
one-or-more(item()*) **as** ~item()+
position() **as** xs:integer
remove(item()*, xs:integer) **as** ~item()*
reverse(item()*) **as** ~item()*
root() **as** node()
root(node()?) **as** node()?
subsequence(item()*, xs:double) **as** ~item()*
subsequence(item()*, xs:double, xs:double) **as** ~item()*
unordered(item()*) **as** ~item()*
zero-or-one(item()*) **as** ~item()?

Miscellaneous Functions

error() **as** none
error(xs:QName) **as** none
error(xs:QName?, xs:string) **as** none
error(xs:QName?, xs:string, item()*) **as** none
lang(xs:string?) **as** xs:boolean
lang(xs:string?, node()) **as** xs:boolean
max(xs:anyAtomicType*) **as** ~xs:anyAtomicType?
max(xs:anyAtomicType*, string) **as** ~xs:anyAtomicType?
min(xs:anyAtomicType*) **as** ~xs:anyAtomicType?
min(xs:anyAtomicType*, string) **as** ~xs:anyAtomicType?
trace(item()*, xs:string) **as** ~item()*

Boolean Functions

boolean(item()*) **as** xs:boolean
false() **as** xs:boolean
not(item()*) **as** xs:boolean
true() **as** xs:boolean

The **eq, ne, lt, gt, le** and **ge** comparisons are supported for the **xs:boolean** type.

URI, ID and XML Name Functions

base-uri() **as** xs:anyURI?
base-uri(node()?) **as** xs:anyURI?
document-uri(node()?) **as** xs:anyURI?
doc-available(xs:string?) **as** xs:boolean
in-scope-prefixes(element()) **as** xs:string*
id(xs:string*) **as** element()*
id(xs:string*, node()) **as** element()*
idref(xs:string*) **as** node()*
idref(xs:string*, node()) **as** node()*
iri-to-uri(xs:string?) **as** xs:string
local-name() **as** xs:string
local-name(node()?) **as** xs:string
local-name-from-QName(xs:QName?) **as** xs:NCName?
name() **as** xs:string
name(node()?) **as** xs:string
namespace-uri() **as** xs:anyURI
namespace-uri(node()?) **as** xs:anyURI
namespace-uri-for-prefix(xs:string?, element()) **as** xs:anyURI?
namespace-uri-from-QName(xs:QName?) **as** xs:anyURI?
prefix-from-QName(xs:QName?) **as** xs:NCName?
QName(xs:string?, xs:string) **as** xs:QName
resolve-QName(xs:string?, element()) **as** xs:QName?
resolve-uri(xs:string?) **as** xs:anyURI?
resolve-uri(xs:string?, xs:string) **as** xs:anyURI?
static-base-uri() **as** xs:anyURI?

Built-In Schema Types

These types are available in all implementations.

xs:anyAtomicType
xs:anySimpleType
xs:anyType
xs:base64Binary
xs:boolean
xs:date
xs:dateTime
xs:dayTimeDuration
xs:decimal
xs:double
xs:duration
xs:float
xs:gDay
xs:gMonth
xs:anyURI
xs:gMonthDay
xs:gYear
xs:gYearMonth
xs:hexBinary
xs:integer
xs:QName
xs:string
xs:time
xs:untyped
xs:untypedAtomic
xs:yearMonthDuration

Escaping Characters

Characters that have special meaning in regular expressions need to be escaped if they are to be represented "as is". These characters are:

\ | . ? * + ( ) { } [ ] - ^ $

In addition, the following escapes represent single characters:

\n	newline or line-feed character (
)
\r	carriage return character (
)
\t	tab character ( )

Multi-Character Escapes

.	(dot) Any Non-Line-End Character
\s	Any Space Character
\i	Any Initial Name Character (including '_' and ':')
\c	Any Name Character (including '.', '-', '_' and ':')
\d	Any Decimal Digit
\w	Any "Word" Character (anything other than Punctuation, Separator or "Other")

An upper-case multi-character escape matches any character not described by the lower-case escape. The upper-case escapes are:

\S \I \C \D \W

Character Class Expressions

A character class expression matches a single character. It's wrapped in square brackets and consists of three parts:

1. an optional negation indicator, ^.
2. one or more characters or ranges, and
3. an optional character class subtraction.

If the negation indicator is used, the single character matched is any character not given following it or in a given range.

A character range consists of two characters separated by a dash, as in:

[-a-zA-Z0-9_]

A leading dash (-) is a dash, not a range.

A character class subtraction consists of a dash followed by a character, category escape or nested character class expression, as in:

[a-z-[aeiou]]

i.e. Match lower-case letters but not the vowels.

XPath 2.0 and XQuery 1.0 Functions That Use Regular Expressions

matches(xs:string?, xs:string) **as** xs:boolean

matches(xs:string?, xs:string, xs:string) **as** xs:boolean

replace(xs:string?, xs:string, xs:string) **as** xs:string

replace(xs:string?, xs:string, xs:string, xs:string) **as** xs:string

tokenize(xs:string?, xs:string) **as** xs:string*

tokenize(xs:string?, xs:string, xs:string) **as** xs:string*

XSLT 2.0 Instructions That Use Regular Expressions

<xsl:analyze-string select = expression
regex = { string }
flags = { string }**>**
<xsl:matching-substring>
sequence-constructor
</xsl:matching-substring>
<xsl:non-matching-substring>
sequence-constructor
</xsl:non-matching-substring>
xsl:fallback*
</xsl:analyze-string>

One but not both of **xsl:matching-substring** and **xsl:non-matching-substring** can be omitted.

Inside **xsl:matching-substring**, the **regex-group(N)** function returns the **N**th group captured by the regular expression.

Regular Expression Matching Flags

Flags are letters used to indicate how Regular Expression matching is to be done:

s	Dot (.) matches any character, line-end characters included.
m	^ and $ match at the start and end of all lines, not just the start and end of the selected string as a whole.
i	Match case insensitive.
x	Remove white-space (space, tab and line-end) characters from the regular expression before using it.

Zero or more flags are specified as a string using the optional **flags=** attribute of **xsl:analyze-string** or the optional last argument of the **matches**, **replace** and **tokenize** functions.

2008-07-21

Regular Expressions in XSLT 2.0, XQuery 1.0 and XPath 2.0

Sam Wilmott
sam@wilmott.ca
http://www.wilmott.ca

and

Mulberry Technologies, Inc.
17 West Jefferson Street, Suite 207
Rockville, MD 20850 USA
Phone: +1 301/315-9631
Fax: +1 301/315-8285
info@mulberrytech.com
http://www.mulberrytech.com

Mulberry Technologies, Inc.

© 2007-2008 Sam Wilmott and Mulberry Technologies, Inc.

Regular Expression Basics

A regular expression is:

oneThing | anotherThing | yetAnother
Match one thing or another or another (one or more things).

oneThing anotherThing yetAnother
Match one thing followed by another etc. (one or more things)

atom quantifier
Match **atom** the number of times indicated by **quantifier**; once if **quantifier** is omitted.

Where **atom** is any of:

- an unescaped character,
- an escaped character,
- a parenthesized regular expression, or
- a character class expression.

Where **quantifier** is any of:

?	zero or one times (i.e. optional)
*****	zero or more times
+	one or more times
{N}	exactly N times
{N,}	N or more times
{N,M}	between N and M times inclusive.

An extra trailing **?**, as in **??**, **+?** or **{N,M}?** means match the shortest possible number of repetitions rather than the (default) longest.

Line Starts and Ends

A regular expression can be anchored at the start and/or end of a string using **^** (the start) and **$** (the end). If a regular expression is used with the **m** flag, **^** and **$** match at the start and end of each line.

In the absence of **^** or **$**, a regular expression matches unanchored: anywhere within the string.

Subexpressions and Back References

Each parenthesized group in a regular expression is assigned a group number counting unescaped left parentheses starting from the left.

Group numbers can be used in three ways:

1. Within a regular expression, to match what was matched by a previous subexpression. A previously matched group is identified by backslash and a number: **\1**, **\2** etc.
2. Within a **replace** replacement expression to match what was matched by a previous subexpression. A group is identified by a numeric name: **$1**, **$2** etc. As well, **$0** identifies the whole matched substring.
3. within a XSLT **regex-group(N)** to access the matched subexpression.

Category Escapes

A category escape matches a character from a set specified by a property or using a block:

\p indicates match any character in the set.

\P indicates match any character not in the set.

Categories and Properties

Any character can be matched by its properties using a category escape consisting of a Category code followed by an optional Property code:

\p{L}	Any Letter
\p{Lu}	Any Upper-case Letter
\p{Ll}	Any Lower-case Letter
\p{Lt}	Any Title-case Letter
\p{Lm}	Any Letter Modifier
\p{Lo}	Any "Other" Letter
\p{M}	Any Mark
\p{Mn}	Any Non-Spacing Mark
\p{Mc}	Any Combining Mark
\p{Me}	Any Enclosing Mark
\p{N}	Any Digit
\p{Nd}	Any Decimal Digit
\p{Nl}	Any Letter Digit
\p{No}	Any "Other" Digit
\p{P}	Any Punctuation Character
\p{Pc}	Any Connector Character
\p{Pd}	Any Dash Character
\p{Ps}	Any Open Character
\p{Pe}	Any Close Character
\p{Pi}	Any Initial Quote Character
\p{Pf}	Any Final Quote Character
\p{Po}	Any "Other" Punctuation
\p{Z}	Any Separator Character
\p{Zs}	Any Space Separator
\p{Zl}	Any Line Separator
\p{Zp}	Any Paragraph Separator
\p{S}	Any Symbol Character
\p{Sm}	Any Math Symbol
\p{Sc}	Any Currency Symbol
\p{Sk}	Any Modifier Symbol
\p{So}	Any "Other" Symbol
\p{C}	Any "Other" Character
\p{Cc}	Any Control Character
\p{Cf}	Any Format Character
\p{Co}	Any Private Use Character
\p{Cn}	Any "Not Assigned" Character

Character Blocks

Any character within a Unicode character block can be matched using a category escape consisting of "Is" followed by the block's name. For example: **\p{IsBasicLatin}**

Block Start	Block End	Block Name
0000	007F	BasicLatin
0080	00FF	Latin-1Supplement
0100	017F	LatinExtended-A
0180	024F	LatinExtended-B
0250	02AF	IPAExtensions
02B0	02FF	SpacingModifierLetters
0300	036F	CombiningDiacriticalMarks
0370	03FF	Greek
0400	04FF	Cyrillic
0530	058F	Armenian
0590	05FF	Hebrew
0600	06FF	Arabic
0700	074F	Syriac
0780	07BF	Thaana
0900	097F	Devanagari
0980	09FF	Bengali
0A00	0A7F	Gurmukhi
0A80	0AFF	Gujarati
0B00	0B7F	Oriya
0B80	0BFF	Tamil
0C00	0C7F	Telugu
0C80	0CFF	Kannada
0D00	0D7F	Malayalam
0D80	0DFF	Sinhala
0E00	0E7F	Thai
0E80	0EFF	Lao
0F00	0FFF	Tibetan
1000	109F	Myanmar
10A0	10FF	Georgian
1100	11FF	HangulJamo
1200	137F	Ethiopic
13A0	13FF	Cherokee
1400	167F	UnifiedCanadianAboriginalSyllabics
1680	169F	Ogham
16A0	16FF	Runic
1780	17FF	Khmer
1800	18AF	Mongolian
1E00	1EFF	LatinExtendedAdditional
1F00	1FFF	GreekExtended
2000	206F	GeneralPunctuation
2070	209F	SuperscriptsandSubscripts
20A0	20CF	CurrencySymbols
20D0	20FF	CombiningMarksforSymbols
2100	214F	LetterlikeSymbols
2150	218F	NumberForms
2190	21FF	Arrows
2200	22FF	MathematicalOperators
2300	23FF	MiscellaneousTechnical
2400	243F	ControlPictures
2440	245F	OpticalCharacterRecognition
2460	24FF	EnclosedAlphanumerics
2500	257F	BoxDrawing
2580	259F	BlockElements
25A0	25FF	GeometricShapes
2600	26FF	MiscellaneousSymbols
2700	27BF	Dingbats
2800	28FF	BraillePatterns
2E80	2EFF	CJKRadicalsSupplement
2F00	2FDF	KangxiRadicals
2FF0	2FFF	IdeographicDescriptionCharacters
3000	303F	CJKSymbolsandPunctuation
3040	309F	Hiragana
30A0	30FF	Katakana
3100	312F	Bopomofo
3130	318F	HangulCompatibilityJamo
3190	319F	Kanbun
31A0	31BF	BopomofoExtended
3200	32FF	EnclosedCJKLettersandMonths
3300	33FF	CJKCompatibility
3400	4DB5	CJKUnifiedIdeographsExtensionA
4E00	9FFF	CJKUnifiedIdeographs
A000	A48F	YiSyllables
A490	A4CF	YiRadicals
AC00	D7A3	HangulSyllables
E000	F8FF	PrivateUse
F900	FAFF	CJKCompatibilityIdeographs
FB00	FB4F	AlphabeticPresentationForms
FB50	FDFF	ArabicPresentationForms-A
FE20	FE2F	CombiningHalfMarks
FE30	FE4F	CJKCompatibilityForms
FE50	FE6F	SmallFormVariants
FE70	FEFE	ArabicPresentationForms-B
FEFF	FEFF	Specials
FF00	FFEF	HalfwidthandFullwidthForms
FFF0	FFFD	Specials

XSLT 2.0:
http://www.w3.org/TR/xslt20/

XQuery 1.0:
http://www.w3.org/TR/xquery/

XPath 2.0:
http://www.w3.org/TR/xpath20/

Unicode:
http://www.unicode.org

Regular Expression Examples

^[A-Za-z]
An Ascii letter at the start of a string or line.

^\p{Lu}
An upper-case Unicode letter at the start of a string or line.

\.$
A period at the end of a string or line.

\p{IsGreek}+
One or more Greek letters.

\p{IsGreek}{1,}
One or more Greek letters.

.*?;
Up to and including the next semicolon.

.*;
Up to and including the last semicolon.

^\c+$
Match only if the string consists entirely of XML name characters.

[-~-[\[\]]]+
Any Ascii printable character except the square brackets.

\w+
A "word".

[^\s]+
Non-white-space characters.

\S+
Non-white-space characters.

(["'])(.*?)\1
A string delimited by single or double quotes. $2 or regex-group(2) will return the unquoted substring. (\1 is the quote character used.)

\s*(\i\c*)\s*=\s*(["'])(.*?)\2
An XML-attribute-like name, equal and quoted value (with optional leading and intervening white space). $1 is the name and $3 is the value.

\((\d+|\p{L}+)\)
A parenthesized sequence either of digits or of letters (but not a mixture of both).

\p{Sc}(\d+(\.\d*)?|\.\d+)
A decimal number with a leading currency symbol.

Schematron 1.5

Schamatron 1.5 differs from ISO Schmatron in the following ways:

Overall:

- The namespace for Schematron 1.5 is: "http:/www.ascc.net/xml/schematron"
- <let> and <include> elements are not supported.
- <key> element is supported:

  <**key name**="NAME" **path**="PATH" icon="URI"/>

  <key> is allowed anywhere in the content of <rule>. (In ISO Schematrons implementations supporting the use of XSLT "foreign" elements, <xsl:key> can be used in place of Schematron 1.5's <key>.)
- Abstract <pattern>s are not supported.
- Attribute pattern/@name used to name <pattern>s rather than @id. It's a required attribute.

Unsupported Attributes:

- These attributes are not supported anywhere: @xml:space, @flag.
- These attributes are not supported on <rule>: @see, @xml:lang, @icon, @fpi, @subject.
- These attributes are not supported on <diagnostics>: @see, @fpi.
- In addition, attribute @see is not supported on <schema>, <assert> or <report>.

Other Differences:

- <value-of> isn't allowed as a child of <assert> or <report>.
- Attribute @version is allowed on <schema>. (Default value is "1.5".)
- The following attributes are optional: ns/@uri, dir/@value and span/@class.

Schematron 1.6

Schematron 1.6 differs from Schematron 1.5 in supporting most ISO Schematron features, including <let>, <include>, abstract <pattern>s and <value-of> in <assert> and <report>.

Schematron 1.5/1.6 Resources:
http://xml.ascc.net/schematron/

Schematron Validation Report Language

The Schematron Validation Report Language is the standard for the output of an ISO Schematron processor. It can be post-processed to produce more readable output, if required.

<**schematron-output** title="TEXT"
phase="NMTOKEN" schemaVersion="TEXT"
xmlns="http://purl.oclc.org/dsdl/svrl">
<text>*, <ns-prefix-in-attribute-values>*,
(<active-pattern>, (<fired-rule>,
(<failed-assert> |
<successful-report>)*)+)+
</**schematron-output**>

<**ns-prefix-in-attribute-values**
prefix="NMTOKEN" **uri**="URI"/>

Only namespaces from <ns> need to be reported.

<**active-pattern** id="ID" name="TEXT"
role="NMTOKEN"/>

Only active <pattern>s are reported.

<**fired-rule** id="ID" **context**="TEXT"
role="NMTOKEN" flag="NMTOKEN"/>

Only <rule>s that are fired are reported.

<**diagnostic-reference**
diagnostic="NMTOKEN">
<text>
</**diagnostic-reference**>

Only references are reported, not the <diagnostic>.

<**failed-assert** id="ID" **location**="TEXT"
test="TEXT" role="NMTOKEN"
flag="NMTOKEN">
<diagnostic-reference>*, <text>
</**failed-assert**>

Only failed <assert>s are reported.

<**successful-report** id="ID" **location**="TEXT"
test="TEXT" role="NMTOKEN"
flag="NMTOKEN">
<diagnostic-reference>*, <text>
</**successful-report**>

Only successful <report>s are reported.

<**text**>
text
</**text**>

See other Quick References for at:
http://www.mulberrytech.com/quickref

2012-03-05

ISO Schematron Quick Reference

Sam Wilmott
sam@wilmott.ca
http://www.wilmott.ca

and

Mulberry Technologies, Inc.
17 West Jefferson Street, Suite 207
Rockville, MD 20850 USA
Phone: +1 301/315-9631
Fax: +1 301/315-8285
info@mulberrytech.com
http://www.mulberrytech.com

© 2009-2012 Sam Wilmott and Mulberry Technologies, Inc.

ISO Schematron Examples

Checking a document for good practice:

```
<schema xmlns=
        "http://purl.oclc.org/dsdl/schematron"
        queryBinding="xslt2">
   <pattern>
      <title>Check paragraphs and titles for
            content</title>
      <rule context="title">
         <report test="*">A title can only contain
            text.</report>
         <assert test="normalize-space()">A title
            must have content.</assert>
      </rule>
      <rule context="p">
         <assert test="* or normalize-space()">A
            p must have content.</assert>
      </rule>
   </pattern>
   <pattern>
      <title>Report use of HTML formatting
            elements.</title>
      <rule context="b | i">
      <report test="true()">HTML <name/>
            elements shouldn't be used (found
            in<name path=".."/>).</report>
      </rule>
   </pattern>
   <pattern>
      <title>Check that titles precede
            something.</title>
      <rule context="title">
         <assert test=
         "following-sibling::*[1][not(self::title)]"
         >A title should be followed by a
            non-title element.</assert>
      </rule>
   </pattern>
</schema>
```

ISO Schematron:
Go to:
http://www.iso.org/PubliclyAvailableStandards
and search for "Schematron".

Other Schematron resources:
http://www.schematron.com

Top-Level Schema

This Quick Reference primarily describes ISO Schematron. See the "Difference" panel for Schamatron 1.5 and 1.6.

<schema id="ID" icon="URI" see="URI"
fpi="FORMAL-PUBLIC-ID" xml:lang="LANG"
xml:space={"preserve" | "default"}
schemaVersion="VERSION"
defaultPhase="IDREF"
queryBinding="BINDING-NAME"
xmlns=
"http://purl.oclc.org/dsdl/schematron">
<title>?, <ns>*, *, <let>*, <phase>*,
<pattern>+, *, <diagnostics>?, plus
interspersed <include>
</schema>

<ns prefix="NMTOKEN" **uri**="URI"/>

All namespaces used in validated documents, and referenced in the schema, must be declared using <ns>.

<**let name**="NAME" **value**="VALUE"/>

<include href="URI"/>

Patterns

<pattern abstract="false" id="ID"
icon="URI" see="URI"
fpi="FORMAL-PUBLIC-ID" xml:lang="LANG"
xml:space={"preserve" | "default"}>
, <let>, <rule>*, plus interspersed
<include>
</pattern>

Within each pattern, only the first non-abstract <rule> whose **@context** matches is used.

Abstract patterns

<pattern abstract="true" id="ID"
icon="URI" see="URI"
fpi="FORMAL-PUBLIC-ID" xml:lang="LANG"
xml:space={"preserve" | "default"}>
, <let>, <rule>*, plus interspersed
<include>
</pattern>

Using abstract patterns

<pattern abstract="false" **is-a**="IDREF" id="ID"
icon="URI" see="URI"
fpi="FORMAL-PUBLIC-ID" xml:lang="LANG"
xml:space={"preserve" | "default"}>
, <param>, and interspersed <include>
</pattern>

<param name="NCNAME" **value**="VALUE"/>
@value must be non-empty-string

Phases

<phase id="ID" icon="URI" see="URI"
fpi="FORMAL-PUBLIC-ID" xml:lang="LANG"
xml:space={"preserve" | "default"}>
, <let>, <active>*, plus interspersed
<include>
</phase>

<active pattern="IDREF">
any number of text, <dir>, <emph> and

</active>

Rules, Assertions and Reports

<rule flag="NAME" abstract="false"?
context="PATH" id="ID" icon="URI"
fpi="FORMAL-PUBLIC-ID" xml:lang="LANG"
xml:space={"preserve" | "default"}
see="URI" role="ROLE" subject="PATH">
any number of <let>, followed by any number
(at least one) of <assert>, <report> and
<extends>, plus interspersed <include>
</rule>

<extends rule="IDREF"/>
plus any foreign attributes

<assert test="EXPR" flag="NAME" id="ID"
diagnostics="IDREFS" icon="URI" see="URI"
fpi="FORMAL-PUBLIC-ID" xml:lang="LANG"
xml:space={"preserve" | "default"}
role="ROLE" subject="PATH">
any number of text, <name>, <value-of>,
<emph>, <dir> and
</assert>

<report test="EXPR" flag="NAME" id="ID"
diagnostics="IDREFS" icon="URI" see="URI"
fpi="FORMAL-PUBLIC-ID" xml:lang="LANG"
xml:space={"preserve" | "default"}
role="ROLE" subject="PATH">
any number of text, <name>, <value-of>,
<emph>, <dir> and
</report>

Abstract rules (used to <extends> others)

<rule flag="NAME" **abstract="true"**
id="ID" icon="URI"
fpi="FORMAL-PUBLIC-ID" xml:lang="LANG"
xml:space={"preserve" | "default"}
see="URI" role="ROLE" subject="PATH">
any number of <let>, followed by any number
(at least one) of <assert>, <report> and
<extends>, plus interspersed <include>
</rule>

XSL-List:
http://www.mulberrytech.com/xsl/xsl-list

Diagnostics

<diagnostics>
any number of <diagnostic> and <include>
</diagnostics>

<diagnostic id="ID" icon="URI" see="URI"
fpi="FORMAL-PUBLIC-ID" xml:lang="LANG"
xml:space={"preserve" | "default"}>
any number of text, <value-of>, <emph>,
<dir> and
</diagnostic>

Formatting Output

<title>
any number of <dir> and text
</title>

<p id="ID" class="CLASS" icon="URI">
any number of text, <dir>, <emph> and

** **

<dir value={"ltr" | "rtl"}>
text
</dir>

<emph>
text
</emph>

<span class="CLASS">
text
****

<value-of select="PATH"/>

<name path="PATH"/>

If **@path** not specified, <name> returns the name of the current node.

Attribute Specification Options

{ }	alternate allowed values
bold =	required attribute
non-bold =	optional attribute

W3C XSLT 1.0 Specification:
http://www.w3.org/TR/xslt

W3C XPath 1.0 Specification:
http://www.w3.org/TR/xpath

W3C XSLT 2.0 Specification:
http://www.w3.org/TR/xslt20

W3C XPath 2.0 Specification:
http://www.w3.org/TR/xpath20

Which Patterns Are Used?

All non-abstract <pattern>s are used if:

- there's no <phase> in the <schema>,
- there's no <phase> selected by its @id attribute, or
- the <schema> is invoked with the #ALL option.

If there's a **@defaultPhase**, and the <schema> is invoked with the #DEFAULT option, then all <pattern>s referenced in the <active> children of the default <phase> are used.

If the implementation selects a <phase> using its **@id** attribute, then all <pattern>s referenced in the <active> children of that <phase> are used.

How #ALL, #DEFAULT and named phases are specified is implementation-determined.

More About Attributes

@abstract indicates whether a <pattern> or <rule> is to be used as-is (if "false") or by another <pattern> or <rule> (if "true").

@defaultPhase (on <schema>) indicates which <phase> is used to determine which <pattern>s are selected by the #DEFAULT option.

@flag on a fired <rule>, on a failing <assert> or on a succeeding <report> sets a flag for further processing.

@fpi is a public identifier associated with the element it appears on.

@icon is the URI of the location of a graphic.

@queryBinding (on <schema>) indicates which query language is to be used. The default is "xslt" — for XSLT/XPath 1.0. Other appropriate values are: "stx", "xslt1.1", "exslt", "xslt2", "xpath", "xpath2", "xquery".

@role is a name classifying the <rule>, <assert> or <report>, or the **@subject**, if any.

@see is the URI of information about the schema itself.

@subject is a path describing related elements and/or attributes, if other than the context of the current <rule>.

Foreign Elements and Attributes

Schema elements can have "foreign" attributes, and non-empty schema elements can contain "foreign" child elements. Foreign attributes and elements are those in a namespace other than "http://purl.oclc.org/dsdl/schematron".

Relative Location Paths

Relative Location Paths traverse the document from the context node

para
para element children
Also – **child::para**

@type
the **type** attribute
Also – **attribute::type**

../title
the **title** element children of the parent

*** except title**
child elements except **title** elements
Also – ***[not(self::title)]** (works in XPath 1.0)

ancestor::sec
all **sec** ancestor elements

ancestor::sec/@n
all **n** attributes on **sec** ancestor elements

list/(item | step)
item and **step** element children of **list** children, in document order

list/item, list/step
item element children of **list** children followed by **step** children of **list** children

preceding-sibling::step
all preceding sibling **step** elements

preceding-sibling::*[1][self::step]
the directly preceding sibling element, if it is a **step** (otherwise nothing)

descendant::div[last()]
the last **div** descendant of the current node

.//div[last()]
div descendants that are the last child **div** of each of their parents

preceding::pb[1]
the first (most immediate) preceding **pb**

ancestor::sec//pb intersect preceding::pb
pb elements inside the same **sec** element as the context node, preceding it

p[normalize-space()]
p child elements that have a non-whitespace value (text content)

***[not(node())]**
empty element children (i.e., element children with no node children)

***[not(node() except (comment()|processing-instruction()))]**
element children that are empty (have no children) except for comments or processing instructions

step[position() gt 1]
all **step** element children but the first

step except *[1]
step element children but the first

step[position() le 4]
the first four **step** element children
Also – **step[position() = (1 to 4)]**

step[position() mod 2]
odd-numbered **step** children

step[not(position() mod 2)]
even-numbered **step** children

***[position() le 4] intersect step**
from the first four element children, the **step** children

ancestor-or-self::*[exists(@lang)][1]/@lang
the closest **lang** attribute on the context node or an ancestor element

Expressions that are not Location Paths

(@class,'none')[1]
the **class** attribute, or if it does not exist, the string **"none"**.
Also –
if (exists(@class)) then @class else "none"

//*/name()
the names of all elements, in document order

distinct-values(//*/name())
the names of all elements, in document order, with duplicates removed

//name/string-join((first, last),' ')
a sequence of strings constructed from the **name** elements in the document, each one concatenating the values of its **first** and **last** element children, in that order, joining them with spaces
Also – **for $n in //name return string-join(($n/first,$n/last),' ')**

//*/count(ancestor-or-self::*)
a sequence of numbers representing the depth of each element in the document

max(//*/count(ancestor-or-self::*))
the maximum depth of all elements in the document (a number in a singleton sequence)

for $stooge in ('Moe','Larry','Curly') return count(//p[contains(.,$stooge)])
the counts of all **p** elements in the document mentioning each of "Moe", "Larry" and "Curly", in that order

index-of(('Moe','Larry','Curly'), speaker[1])
if the first **speaker** element child has the value "Moe", then 1; if "Larry", then 2; if "Curly", then 3; otherwise the empty sequence (i.e., no value)

(: You've got to be kidding me. :)
do nothing. A comment is just a comment.

2008-07-21

XPath 2.0 Quick Reference

See also the "XQuery 1.0 & XPath 2.0 Functions & Operators Quick Reference"

Sam Wilmott
sam@wilmott.ca
http://www.wilmott.ca

and

Mulberry Technologies, Inc.
17 West Jefferson Street, Suite 207
Rockville, MD 20850 USA
Phone: +1 301/315-9631
Fax: +1 301/315-8285
info@mulberrytech.com
http://www.mulberrytech.com

© 2007-2008 Sam Wilmott and Mulberry Technologies, Inc.

Absolute Location Paths

Absolute Location Paths traverse the document starting at the top (the root), and can be recognized by their initial / (forwardslash).

/book/bookinfo/abstract
an **abstract** element child of a **bookinfo** child of the **book** document element
Also – **/child::book/child::bookinfo/child::abstract**

//para
all **para** elements in the document
Also – **/descendant-or-self::*/child::para**
Also – **/descendant::para**

/descendant::para[1]
the first **para** element in the document
Also – **(//para)[1]**

//@order-by
all **order-by** attributes in the document

//list[exists(ancestor::list)]
all **list** elements that have ancestor **list**elements

//list[not(ancestor::list)]
all **list** elements that do not have ancestor **list** elements
Also – **//list[not(exists(ancestor::list))]**
Also – **//list[empty(ancestor::list)]**

//(* except title)
all elements except **title** elements
Also – **//*[not(self::title)]** (works in XPath 1.0)

//processing-instruction()[not(ancestor::sec/@n = 1)]
all processing instructions with no **sec** ancestor elements with **n** attributes equal to 1

//para[matches(.,'[X|x]{3}')]
all **para** elements whose value includes the regular expression **[X|x]{3}**
Tip – **[X|x]{3}** matches three **X** or **x**characters appearing in a row

//sec[@id = //@rid/tokenize(.,'\s+')]
all **sec** elements with **id** attributes whose values are also given as a value by a tokenized **rid** attribute anywhere in the document
Also – **//sec[@id = $rid-values]** where **$rid-values** is **distinct-values(//@rid/tokenize(.,'\s+'))**
Tip – use **distinct-values(//@rid/tokenize(.,'\s+'))** to remove duplicates from the list of tokenized **@rid** values
Tip – the regular expression **\s+** matches any contiguous sequence of spaces (space, linefeed or tab characters)

Simple Expressions

$VarName
(Expr **)**
( )
. (one dot: self)
QName **(** Expr **, ...)**
QName **( )**
IntegerLiteral
DecimalLiteral
DoubleLiteral
StringLiteral

Arithmetic Expressions

+ Expr	Expr + Expr
- Expr	Expr - Expr
Expr ***** Expr	Expr **div** Expr
Expr **idiv** Expr	Expr **mod** Expr

Creating Sequences

Create a sequence from a list of items:
Expr **,** ...
Note: A sequence list must usually be parenthesized.

Repeat over one or more sequences, returning a sequence of results:
for VariableBinding **,** ... **return** Expr

where a VariableBinding is:
$VarName **in** Expr

Create a numeric sequences, from lower bound to upper bound:
Expr **to** Expr

All the items appearing in either sequence:
Expr **union** Expr
Expr | Expr

Only items appearing in both sequences:
Expr **intersect** Expr

All items in the first sequence not in second:
Expr **except** Expr

Comments in XPath Expressions

(: This is a comment within an XPath expr **:)**

Testing

Test if the condition is satisfied for at least one combination of the bound expressions:
some VariableBinding **,** ... **satisfies** Expr

Test if the condition is satisfied for all of the bound expressions:
every VariableBinding **,** ... **satisfies** Expr

Select one or the other of two possibiliites:
if (Expr **) then** Expr **else** Expr

Either or both of two tests:
Expr **or** Expr Expr **and** Expr

Test if they are the same node:
Expr **is** Expr

Test if a node appears before or after another:
Expr << Expr Expr >> Expr

Test an expression's dynamic type:
Expr **instance of** SequenceType

Test if an expression can be converted to a type:
Expr **castable as** AtomicType
Expr **castable as** AtomicType**?**

Compare two atomic values:

Expr **eq** Expr	Expr **ne** Expr
Expr **lt** Expr	Expr **le** Expr
Expr **gt** Expr	Expr **ge** Expr

Compare all items in one sequence to all items in a second, and return if true for any pair of values:

Expr = Expr	Expr **!=** Expr
Expr < Expr	Expr <= Expr
Expr > Expr	Expr >= Expr

Type Modification Expressions

Use as without converting:
Expr **treat as** SequenceType

Use as, converting as needed and doable:
Expr **cast as** AtomicType
Expr **cast as** AtomicType**?**

XPath 2.0:
http://www.w3.org/TR/xpath20/
XSL-List:
http://www.mulberrytech.com/xsl/xsl-list

Path Expressions

/	Top level, document root
/ Step	At top level
Step	Relative to current node
// Step	Anywhere within document
Path / Step	Immediately within Path
Path // Step	Anywhere within Path

Where a Step is one of:
Expr
AxisName**::**NameTest
AxisName**::**KindTest
@NameTest (attribute test)
NameTest (child element test)
KindTest (child node test)
.. (two dots: parent test)
Followed by zero or more predicates:
[Expr **]**

Where an AxisName is one of:

ancestor	**ancestor-or-self**
attribute	**child**
descendant	**descendant-or-self**
following	**following-sibling**
namespace	**parent**
preceding	**preceding-sibling**
self	

Where a NameTest is one of:
QName
*****
NCName**:***
***:**NCName

Where a KindTest is one of:
attribute (AttributeName **)**
attribute (AttributeName **,** TypeName **)**
attribute (*)
attribute (* , TypeName **)**
attribute ( )
comment ( )
document-node (element ... **)**
document-node (schema-element ... **)**
document-node ( )
element (ElementName **)**
element (ElementName **,** TypeName **)**
element (*)
element (* , TypeName **)**
element ( )
node ( )
processing-instruction (NCName **)**
processing-instruction (StringLiteral **)**
processing-instruction ( )
schema-attribute (AttributeName **)**
schema-element (ElementName **)**
text ( )

Names and Types

XML QNames, with or without a colon-separated prefix, is use for all of:
VarName
AttributeName
ElementName
TypeName
AtomicType

A SequenceType is one of:
empty-sequence ( )
KindTest
item ( )
AtomicType

Where KindTest, **item()** or AtomicType can be optionally followed by:

?	(may be empty sequence)\
+	(is a non-empty sequence of the type)
*****	(is a sequence of the type, empty or not)

Operator Precedence:

1 **,** (comma)
2 **for some every if**
3 **or**
4 **and**
5 **= != < <= > >= eq ne lt le gt ge is << >>**
6 **to**
7 (two-argument) **+ -**
8 *** div idiv mod**
9 **union |**
10 **intersect except**
11 **instance of**
12 **treat as**
13 **castable as**
14 **cast as**
15 (one-argument) **+ -**
16 **/ //**
17 step node-test **$**name **(Expr)** function-call literal

Testing

Select based on the type of an expression (one or more **case**s plus a **default**):

typeswitch (Expr **)** **case** ... **default** ...

where **case** and **default** are:

case SequenceType **return** Expr

case $VarName **as** SequenceType **return** Expr

default return Expr

default $VarName **return** Expr

Test if the condition is satisfied for at least one combination of the bound expressions:

some VariableBinding **,** ... **satisfies** Expr

Test if the condition is satisfied for all of the bound expressions:

every VariableBinding **,** ... **satisfies** Expr

where a VariableBinding is:

$VarName **in** Expr

$VarName **as** SequenceType **in** Expr

Select one or the other of two possibiliites:

if (Expr **) then** Expr **else** Expr

Either or both of two tests:

Expr **or** Expr Expr **and** Expr

Test if they are the same node:

Expr **is** Expr

Test if a node appears before or after another:

Expr << Expr Expr >> Expr

Test an expression's dynamic type:

Expr **instance of** SequenceType

Test if an expression can be converted to a type:

Expr **castable as** AtomicType
Expr **castable as** AtomicType**?**

Compare two item values:

Expr **eq** Expr	Expr **ne** Expr
Expr **lt** Expr	Expr **le** Expr
Expr **gt** Expr	Expr **ge** Expr

Compare all items in one sequence to all items in a second, and return if true for any pair of values:

Expr = Expr	Expr **!=** Expr
Expr < Expr	Expr <= Expr
Expr > Expr	Expr >= Expr

Names and Types

VarName AttributeName ElementName
TypeName AtomicType

are all XML QNames, with or without a colon-separated prefix.

A SequenceType is one of:

empty-sequence ( )	KindTest
item ( )	AtomicType

Where KindTest, **item()** or AtomicType can be optionally followed by:

?	(may be empty sequence)
+	(is a non-empty sequence of the type)
*	(is a sequence of the type, empty or not)

Operator Precedence:

1	**,** (comma)
2	**for let some every if typeswitch**
3	**or**
4	**and**
5	**= != < <= > >=** **eq ne lt le gt ge is << >>**
6	**to**
7	(two-argument) **+ -**
8	*** div idiv mod**
9	**union |**
10	**intersect except**
11	**instance of**
12	**treat as**
13	**castable as**
14	**cast as**
15	(one-argument) **+ -**
16	**/ //**
17	step node-test **$** name **(** Expr **)** function-call literal **validate** **(#** ... **#)** constructor **ordered** **unordered**

Predefined Namespace Names:

xml = http://www.w3.org/XML/1998/namespace
xs = http://www.w3.org/2001/XMLSchema
xsi = http://www.w3.org/2001/XMLSchema-instance
fn = http://www.w3.org/2005/xpath-functions
local = http://www.w3.org/2005/xquery-local-functions

2008-07-21

XQuery 1.0 Quick Reference

See also the "XQuery 1.0 & XPath 2.0 Functions & Operators Quick Reference"

Sam Wilmott
sam@wilmott.ca
http://www.wilmott.ca

and

Mulberry Technologies, Inc.
17 West Jefferson Street, Suite 207
Rockville, MD 20850 USA
Phone: +1 301/315-9631
Fax: +1 301/315-8285
info@mulberrytech.com
http://www.mulberrytech.com

© 2007-2008 Sam Wilmott and Mulberry Technologies, Inc.

FLWOR Expressions

FLWOR Expressions start with one or more **for** or **let**:

for SequenceVariableBinding **,** ...

let AssignedVariableBinding **,** ...

followed by:

where Expr	(optional)
OrderingInfo **,** ...	(one or more, optional)
return Expr	

where SequenceVariableBinding is one of:

$VarName **in** Expr

$VarName **as** SequenceType **in** Expr

$VarName **at $** VarName **in** Expr

$VarName **as** SequenceType **at** $ VarName **in** Expr

where AssignedVariableBinding is one of:

$VarName := Expr

$VarName **as** SequenceType := Exp

and where OrderingInfo consists of, in order:

stable	(optional)
order Expr	
ascending or **descending**	(optional)
empty greatest or **empty least**	(optional)
collation URILiteral	(optional)

Constructors

< QName ... />

< QName ... > ... </ QName >

<![CDATA[... **]]>**

<!-- ... -->

<? PITarget ... **?>**

document { Expr **}**

element QName **{** OptionalExpr **}**

element { Expr **} {** OptionalExpr **}**

attribute QName **{** OptionalExpr **}**

attribute { Expr **} {** OptionalExpr **}**

text { Expr **}**

comment { Expr **}**

processing-instruction NCName **{** OptionalExpr **}**

processing-instruction { Expr **} {** OptionalExpr **}**

Within a constructor's attribute values and element content, literal "{" and "}" need doubling. Anything within single "{" ... "}" is evaluated as an Expr.

XQuery Scripts

An XQuery script consists of:

1. A Version Declaration

xquery version StringLiteral

followed, optionally, by:

encoding StringLiteral

followed, optionally, by a semicolon (";").

2. If an XQuery script is a Library Module, then it's module namespace declaration comes next:

module namespace NCName = URILiteral ;

3. Default Declarations and Imports:

zero or more of:

declare default element namespace URILiteral ;
declare default function namespace URILiteral ;
declare boundary-space preserve ;
declare boundary-space strip ;
declare default collation URILiteral ;
declare base-uri URILiteral ;
declare construction strip ;
declare construction preserve ;
declare ordering ordered ;
declare ordering unordered ;
declare default order empty greatest ;
declare default order empty least ;
declare copy-namespaces preserve , inherit ;
declare copy-namespaces preserve , no-inherit ;
declare copy-namespaces no-preserve , inherit ;
declare copy-namespaces no-preserve , no-inherit ;
declare namespace NCName = URILiteral ;
import schema namespace NCName = URILiteralList ;
import schema default element namespace URILiteralList ;
import schema URILiteralList ;
import module namespace NCName = URILiteralList ;
import module URILiteralList ;

XQuery 1.0:
http://www.w3.org/TR/xquery/

4. Variable, Function and Option Declarations:

zero or more of:

declare variable VariableDeclaration := ExprSingle ;
declare variable VariableDeclaration **external** ;
declare function QName ParameterDeclarations ;
declare function QName ParameterDeclarations **external;**
declare function QName ParameterDeclarations **as** SequenceType **external** ;
declare option QName StringLiteral ;

where ParameterDeclarations is one of:

( ) (i.e. empty if no parameters)
(VariableDeclaration **)** (for one parameter)
(VariableDeclaration **, ...)** (when two or more)

where VariableDeclaration is one of:

$QName
$QName **as** SequenceType

and where URILiteralList is one of:

URILiteral
URILiteral **at** URILiteral
URILiteral **at** URILiteral **,** ... (two or more)

5. Finally, if the XQuery script is a Main module, not a Library module, an XQuery expression is required to specify the query being made:

Expr

Creating Sequences

Create a sequence from a list of items:

Expr **,** ...

Note: A sequence list must usually be parenthesized.

Repeat over one or more sequences, returning a sequence of results:

for VariableBinding **,** ... **return** Expr

Create a numeric sequences, from lower bound to upper bound:

Expr **to** Expr

All the items appearing in either sequence:

Expr **union** Expr Expr | Expr

Only items appearing in both sequences:

Expr **intersect** Expr

All items in the first sequence not in second:

Expr **except** Expr

Arithmetic Expressions

+ Expr	Expr + Expr
- Expr	Expr - Expr
Expr * Expr	Expr **div** Expr
Expr **idiv** Expr	Expr **mod** Expr

Type Modification Expressions

Use as without converting:

Expr **treat as** SequenceType

Use as, converting as needed and doable:

Expr **cast as** AtomicType

Expr **cast as** AtomicType**?**

Simple Expressions

$ VarName	**.** (one dot: self)
( )	**(** Expr **)**
QName **(** Expr **, ...)**	QName **( )**
IntegerLiteral	DecimalLiteral
DoubleLiteral	StringLiteral

Validating Nodes

validate { Expr **}** (defaults to **strict**)
validate lax { Expr **}**
validate strict { Expr **}**

Ordering Mode for Sequences

ordered { Expr **}**
unordered { Expr **}**

Implementation-Defined Instructions

(# QName ... **#)** ... **{** OptionalExpr **}**

Path Expressions

/	Top level, document root
/ Step	At top level
Step	Relative to current node
// Step	Anywhere within document
Path / Step	Immediately within Path
Path // Step	Anywhere within Path

Where a Step is one of:

Expr
AxisName :: NameTest
AxisName :: KindTest
@NameTest (attribute test)
NameTest (child element test)
KindTest (child node test)
.. (two dots: parent test)

Followed by zero or more predicates:

[Expr **]**

Where an AxisName is one of:

ancestor	**ancestor-or-self**
attribute	**child**
descendant	**descendant-or-self**
following	**following-sibling**
namespace	**parent**
preceding	**preceding-sibling**
self	

A NameTest is one of:

QName	*
NCName**:***	***:**NCName

And a KindTest is one of:

attribute (AttributeName **)**
attribute (AttributeName **,** TypeName **)**
attribute (* , TypeName **)**
attribute (*)
attribute ( )
comment ( )
document-node (element ... **)**
document-node (schema-element ... **)**
document-node ( )
element (ElementName **)**
element (ElementName **,** TypeName **)**
element (* , TypeName **)**
element (*)
element ( )
node ( )
processing-instruction (NCName **)**
processing-instruction (StringLiteral **)**
processing-instruction ( )
schema-attribute (AttributeName **)**
schema-element (ElementName **)**
text ( )

Conditional and Looping Instructions

```
<xsl:analyze-string select = expression
      regex = { string }
      flags = { string }>
   <xsl:matching-substring>
      sequence-constructor
      </xsl:matching-substring>
   <xsl:non-matching-substring>
      sequence-constructor
      </xsl:non-matching-substring>
   xsl:fallback*
   </xsl:analyze-string>
```

One but not both of **xsl:matching-substring** and **xsl:non-matching-substring** can be omitted.

regex-group(N) returns the **N**th group matched by the **regex** within **xsl:matching-substring**.

```
<xsl:choose>
   <xsl:when test = expression>
      sequence-constructor
      </xsl:when>
   <xsl:otherwise>
      sequence-constructor
      </xsl:otherwise>
   </xsl:choose>
```

One or more **xsl:when** and zero or one **xsl:otherwise** are alllowed.

```
<xsl:for-each select = expression>
   xsl:sort*,sequence-constructor
   </xsl:for-each>

<xsl:for-each-group select = expression
      group-by = expression
      group-adjacent = expression
      group-starting-with = pattern
      group-ending-with = pattern
      collation = { uri }>
   xsl:sort*,sequence-constructor
   </xsl:for-each-group>

<xsl:if test = expression>
   sequence-constructor
   </xsl:if>
```

Standard Attributes

Standard attributes are allowed on all elements. When not on **xsl:** elements, the **xsl:** prefix is required on the attribute name.

[xsl:]default-collation = uri

[xsl:]exclude-result-prefixes = tokens

[xsl:]extension-element-prefixes = tokens

[xsl:]use-when = expression

[xsl:]version = "1.0" | "2.0"

[xsl:]xpath-default-namespace = uri

Value/Copy Instructions

```
<xsl:copy copy-namespaces = "yes" | "no"
      inherit-namespaces = "yes" | "no"
      use-attribute-sets = qnames
      type = qname
      validation = "strict" | "lax" |
                   "preserve" | "strip">
   sequence-constructor
   </xsl:copy>

<xsl:copy-of select = expression
      copy-namespaces = "yes" | "no"
      type = qname
      validation = "strict" | "lax" |
                   "preserve" | "strip" />

<xsl:number value = expression
      select = expression
      level = "single" | "multiple" | "any"
      count = pattern
      from = pattern
      format = { string }
      lang = { nmtoken }
      letter-value = { "alphabetic" |
                       "traditional" }
      ordinal = { string }
      grouping-separator = { char }
      grouping-size = { number } />

<xsl:perform-sort select = expression>
   xsl:sort+, sequence-constructor
   </xsl:perform-sort>

<xsl:value-of select = expression
      separator = { string }
      disable-output-escaping = "yes" | "no" >
   sequence-constructor
   </xsl:value-of>
```

disable-output-escaping is deprecated.

```
<xsl:sort select = expression
      lang = { nmtoken }
      order = { "ascending" | "descending"}
      collation = { uri }
      stable = { "yes" | "no" }
      case-order = { "upper-first" | "lower-first" }
      data-type = { "text" | "number" |
                    qname-but-not-ncname } >
   sequence-constructor
   </xsl:sort>
```

xsl:sort is used in **xsl:for-each**, **xsl:for-each-group**, **xsl:apply-templates** and **xsl:perform-sort**.

XSLT 2.0:
http://www.w3.org/TR/xslt20/

XPath 2.0:
http://www.w3.org/TR/xpath20/

2008-07-21

XSLT 2.0 Quick Reference

Sam Wilmott
sam@wilmott.ca
http://www.wilmott.ca

and

Mulberry Technologies, Inc.
17 West Jefferson Street, Suite 207
Rockville, MD 20850 USA
Phone: +1 301/315-9631
Fax: +1 301/315-8285
info@mulberrytech.com
http://www.mulberrytech.com

© 2007-2008 Sam Wilmott and Mulberry Technologies, Inc.

The Stylesheet Element

```
<xsl:stylesheet id = id
      extension-element-prefixes = tokens
      exclude-result-prefixes = tokens
      version = "1.0" | "2.0"
      xpath-default-namespace = uri
      default-validation = "preserve" | "strip"
      default-collation = uri-list
      input-type-annotations = "preserve" |
                               "strip" | "unspecified"
      xmlns:xsl=
         "http://www.w3.org/1999/XSL/Transform">
   xsl:import*, top-level-declarations
   </xsl:stylesheet>
```

xsl:transform is a synonym for **xsl:stylesheet**.

```
<xsl:import href = uri />
```

A literal result element can be used in place of **xsl:stylesheet**, so long as it specifies attribute **xsl:version** and namespace **xmlns:xsl**.

Template Invocation Instructions

```
<xsl:apply-imports>
   xsl:with-param*
   </xsl:apply-imports>

<xsl:apply-templates select = expression
      mode = token>
   (xsl:sort | xsl:with-param)*
   </xsl:apply-templates>

<xsl:call-template name = qname>
   xsl:with-param*
   </xsl:call-template>

<xsl:next-match>
   (xsl:with-param | xsl:fallback)*
   </xsl:next-match>

<xsl:with-param name = qname
      select = expression
      as = sequence-type
      tunnel = "yes" | "no">
   sequence-constructor
   </xsl:with-param>
```

Exception-Handling Instructions

```
<xsl:fallback>
   sequence-constructor
   </xsl:fallback>

<xsl:message select = expression
      terminate = { "yes" | "no" }>
   sequence-constructor
   </xsl:message>
```

Top-Level Declarations

<xsl:attribute-set name = qname
    use-attribute-sets = qnames**>**
  xsl:attribute*
  </xsl:attribute-set>

<xsl:character-map name = qname
    use-character-maps = qnames**>**
  xsl:output-character*
  <xsl:output-character character = char
    string = string **/>**
  </xsl:character-map>

One or more **xsl:output-character** is allowed.

<xsl:decimal-format name = qname
    decimal-separator = char
    grouping-separator = char
    infinity = string
    minus-sign = char
    NaN = string
    percent = char
    per-mille = char
    zero-digit = char
    digit = char
    pattern-separator = char **/>**

<xsl:function name = qname
    as = sequence-type
    override = "yes" | "no"**>**
  xsl:param*, sequence-constructor
  </xsl:function>

<xsl:import-schema namespace = uri
    schema-location = uri**>**
    xs:schema?
  </xsl:import-schema>

<xsl:include href = uri **/>**

<xsl:key name = qname
    match = pattern
    use = expression
    collation = uri**>**
  sequence-constructor
  </xsl:key>

<xsl:namespace-alias
    stylesheet-prefix = prefix | "#default"
    result-prefix = prefix | "#default" **/>**

Content Specification Options

?	optional
*	zero or more
+	one or more
#PCDATA	just text
sequence-constructor	Instructions and text

<xsl:output name = qname
    method = "xml" | "html" | "xhtml" |
      "text" |qname-but-not-ncname
    byte-order-mark = "yes" | "no"
    cdata-section-elements = qnames
    doctype-public = string
    doctype-system = string
    encoding = string
    escape-uri-attributes = "yes" | "no"
    include-content-type = "yes" | "no"
    indent = "yes" | "no"
    media-type = string
    normalization-form = "NFC" | "NFD" |
      "NFKC" | "NFKD" | "none" |
    "fully-normalized" | nmtoken
    omit-xml-declaration = "yes" | "no"
    standalone = "yes" | "no" | "omit"
    undeclare-prefixes = "yes" | "no"
    use-character-maps = qnames
    version = nmtoken **/>**

<xsl:param name = qname
    select = expression
    as = sequence-type
    required = "yes" | "no"
    tunnel = "yes" | "no"**>**
  sequence-constructor
  </xsl:param>

xsl:param is also allowed in **xsl:function** and **xsl:template**.

<xsl:preserve-space elements = tokens **/>**

<xsl:strip-space elements = tokens **/>**

<xsl:template match = pattern
    name = qname
    priority = number
    mode = tokens
    as = sequence-type**>**
  xsl:param*, sequence-constructor
  </xsl:template>

<xsl:variable name = qname
    select = expression
    as = sequence-type**>**
  sequence-constructor
  </xsl:variable>

xsl:variable is also allowed in sequence-constructor contexts.

Attribute Specification Options

{ }	specified using an attribute value template
bold =	required attribute
non-bold =	optional attribute

Node Constructing Instructions

<xsl:attribute name = { qname }
    namespace = { uri }
    select = expression
    separator = { string }
    type = qname
    validation = "strict" | "lax" |
      "preserve" | "strip"**>**
  sequence-constructor
  </xsl:attribute>

<xsl:comment select = expression**>**
  sequence-constructor
  </xsl:comment>

<xsl:document type = qname
    validation = "strict" | "lax" |
      "preserve" | "strip" **>**
  sequence-constructor
  </xsl:document>

<xsl:element name = { qname }
    namespace = { uri}
    inherit-namespaces = "yes" | "no"
    use-attribute-sets = qnames
    type = qname
    validation = "strict" | "lax" |
      "preserve" | "strip"**>**
  sequence-constructor
  </xsl:element>

Element nodes can also be constructed using XML elements not in the **xsl:** namespace, which can also specify **xsl:type**, **xsl:validation** and **xsl:use-attribute-sets** attributes.

<xsl:namespace name = { ncname }
    select = expression**>**
  sequence-constructor
  </xsl:namespace>

<xsl:processing-instruction
    name = { ncname }
    select = expression**>**
  sequence-constructor
  </xsl:processing-instruction>

<xsl:sequence select = expression**>**
  xsl:fallback*
  </xsl:sequence>

<xsl:text disable-output-escaping = "yes" | "no" **>**
  #PCDATA
  </xsl:text>

disable-output-escaping is deprecated.

Text also constructs text nodes.

XSL-List:
http://www.mulberrytech.com/xsl/xsl-list

<xsl:result-document format = { qname }
    href = { uri }
    validation = "strict" | "lax" |
      "preserve" | "strip"
    type = qname
    method = { "xml" | "html" | "xhtml" |
      "text" | qname-but-not-ncname }
    byte-order-mark = { "yes" | "no" }
    cdata-section-elements = { qnames }
    doctype-public = { string }
    doctype-system = { string }
    encoding = { string }
    escape-uri-attributes = { "yes" | "no" }
    include-content-type = { "yes" | "no" }
    indent = { "yes" | "no" }
    media-type = { string }
    normalization-form = { "NFC" | "NFD" |
      "NFKC" | "NFKD" | "none" |
      "fully-normalized" | nmtoken }
    omit-xml-declaration = { "yes" | "no" }
    standalone = { "yes" | "no" | "omit" }
    undeclare-prefixes = { "yes" | "no" }
    use-character-maps = qnames
    output-version = { nmtoken } **>**
  sequence-constructor
  </xsl:result-document>

Allowed Attribute Values:

char	a single character
expression	an XPath expression
id	an ID attribute value
ncname	a name with no namespace prefix
nmtoken	a number token
number	a number (only digits)
pattern	an XPath expression conforming to pattern syntax
prefix	a namespace prefix
qname-but-not-ncname	a name with a namespace prefix
qname	a name with or without a namespace prefix
sequence-type	an XML Schema sequence type (with *)
string	just text
token	specific to its use
uri-list	white-space separated list of URIs
uri	a uniform resource identifier

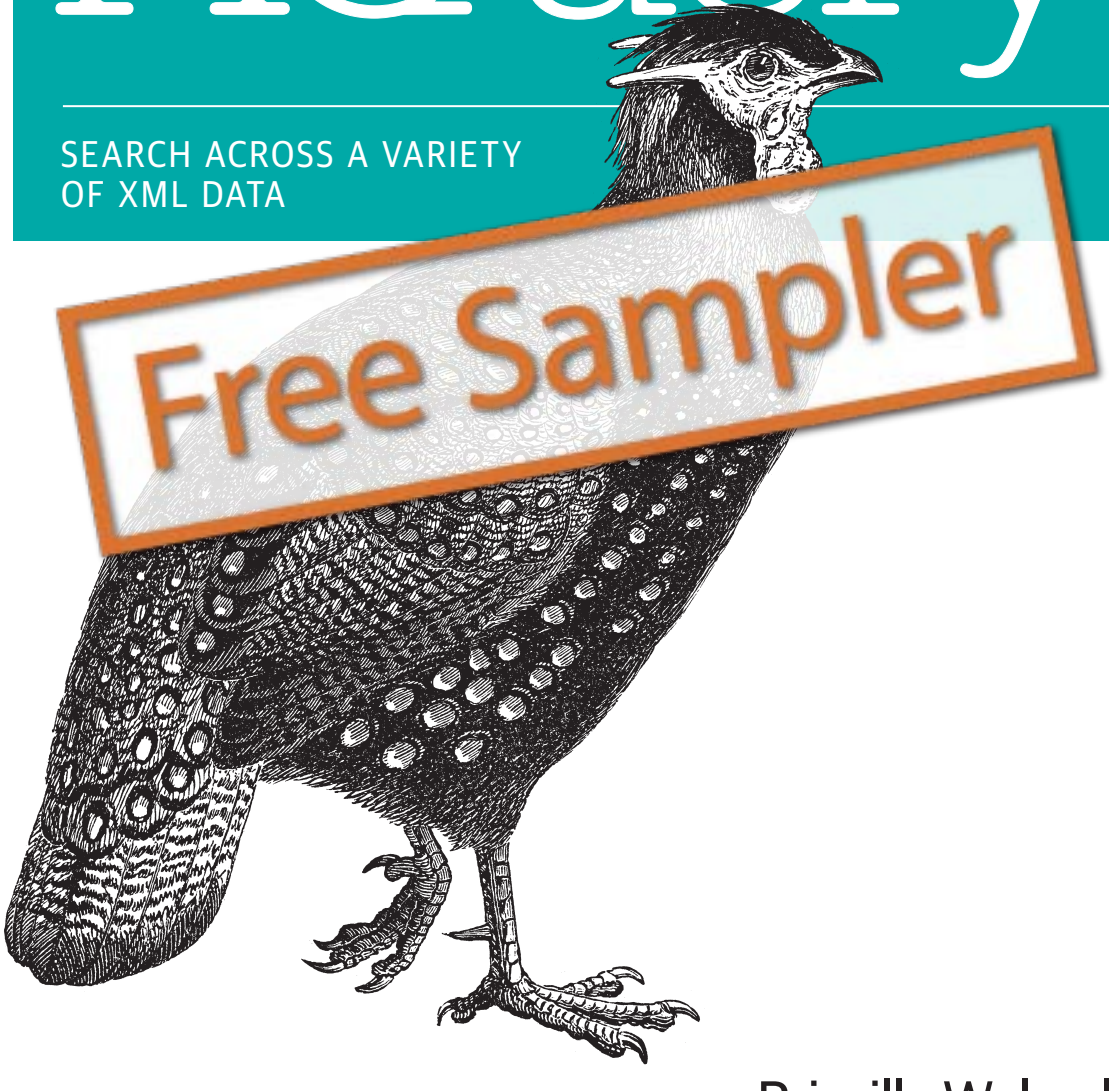
Supplemental Readings

O'REILLY®

2nd Edition

XQuery

SEARCH ACROSS A VARIETY OF XML DATA



Priscilla Walmsley

Table of Contents

SECOND EDITION

XQuery

Search Across a Variety of XML Data

Priscilla Walmsley

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

XQuery
by Priscilla Walmsley

Copyright © 2016 Priscilla Walmsley. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Meg Foley
Production Editor: Shiny Kalapurakkel
Copyeditor: Nan Reinhardt
Proofreader: Sonia Saruba
Indexer: Priscilla Walmsley
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

March 2007: First Edition
December 2015: Second Edition

Revision History for the Second Edition
2015-11-30: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491915103* for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *XQuery,* the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-491-91510-3

[LSI]

CHAPTER 1

Introduction to XQuery

This chapter provides background on the purpose and capabilities of XQuery. It also gives a quick introduction to the features of XQuery that are covered in more detail later in the book. It is designed to provide a basic familiarity with the most commonly used kinds of expressions, without getting too bogged down in the details.

What Is XQuery?

The use of XML has exploded in recent years. An enormous amount of information is now stored in XML, both in XML databases and in documents on a filesystem. This includes highly structured data such as sales figures, semi-structured data such as product catalogs and yellow pages, and relatively unstructured data such as letters and books. Even more information is passed between systems as transitory XML documents.

All of this data is used for a variety of purposes. For example, sales figures may be useful for compiling financial statements that may be published on the Web, reporting results to the tax authorities, calculating bonuses for salespeople, or creating internal reports for planning. For each of these uses, we are interested in different elements of the data and expect it to be formatted and transformed according to our needs.

XQuery is a query language designed by the W3C to address these needs. It allows you to select the XML data elements of interest, reorganize and possibly transform them, and return the results in a structure of your choosing.

Capabilities of XQuery

XQuery has a rich set of features that allow many different types of operations on XML data and documents, including:

- Selecting information based on specific criteria
- Filtering out unwanted information
- Searching for information within a document or set of documents
- Joining data from multiple documents or collections of documents
- Sorting, grouping, and aggregating data
- Transforming and restructuring XML data into another XML vocabulary or structure
- Performing arithmetic calculations on numbers and dates
- Manipulating strings to reformat text

As you can see, XQuery can be used not just to extract sections of XML documents, but also to manipulate and transform the results for output. In fact, XQuery is a Turing-complete functional programming language, which means you can also use it for general-purpose programming and application development, not just for querying data.

Uses for XQuery

There are as many reasons to query XML as there are reasons to use XML. Some examples of common uses for the XQuery language are:

- Finding textual documents in a native XML database and presenting styled results
- Generating reports on data stored in a database for presentation on the Web as HTML
- Extracting information from a relational database for use in a web service
- Pulling data from databases or packaged software and transforming it for application integration
- Combining content from traditionally non-XML sources to implement content management and delivery
- Ad hoc querying of standalone XML documents for the purposes of testing or research
- Building entire complex web applications

Processing Scenarios

XQuery's sweet spot is querying bodies of XML content that encompass many XML documents, often stored in databases. For this reason, it is sometimes called the "SQL of XML." Some of the earliest XQuery implementations were in native XML database products. The term "native XML database" generally refers to a database that is designed for XML content from the ground up, as opposed to a traditionally relational database. Rather than being oriented around tables and columns, its data model is based on hierarchical documents and collections of documents.

Native XML databases are most often used for narrative content and other data that is less predictable than what you would typically store in a relational database. Many of these products are now known by the broader term *NoSQL database* and provide support for not just XML but also JSON and other data formats. Examples of these database products that support XQuery are eXist, MarkLogic Server, BaseX, Zorba, and EMC Documentum xDB. Of these, all but MarkLogic Server and EMC Documentum xDB are open source. These products provide the traditional capabilities of databases, such as data storage, indexing, querying, loading, extracting, backup, and recovery. Most of them also provide some added value in addition to their database capabilities. For example, they might provide advanced full-text searching functionality, document conversion services, or end-user interfaces.

Major relational database products, including Oracle (via its XML DB), IBM DB2 (via pureXML), and Microsoft SQL Server, also have support for XML and various versions of XQuery. Early implementations of XML in relational databases involved storing XML in table columns as blobs or character strings and providing query access to those columns. However, these vendors are increasingly blurring the line between native XML databases and relational databases with new features that allow you to store XML natively.

Other XQuery processors are not embedded in a database product, but work independently. They might be used on physical XML documents stored as files on a file system or on the Web. They might also operate on XML data that is passed in memory from some other process. The most notable product in this category is Saxon, which has both open source and commercial versions. Altova's RaptorXML also provides support for standalone XQuery queries.

XML editors provide support for editing and running XQuery queries and displaying the results. Some, like Altova's XMLSpy, have their own embedded XQuery implementations. Others, like oXygen XML Editor, allow you to run queries using one or more separate XQuery processors. If you are new to XQuery, a free trial license to a product like oXygen or XMLSpy is a good way to get started running queries.

Easing into XQuery

The rest of this chapter takes you through a set of example queries, each of which builds on the previous one. Three XML documents are used repeatedly as input documents to the query examples throughout the book. They will be used so frequently that it may be worth printing them from the companion web site at *http://www.datypic.com/books/xquery/chapter01.html* so that you can view them alongside the examples.

These three examples are quite simplistic, but they are useful for educational purposes because they are easy to learn and remember while looking at query examples. In reality, most XQuery queries will be executed against much more complex documents, and often against multiple documents as a collection. However, in order to keep the examples reasonably concise and clear, this book will work with smaller documents that have a representative mix of XML characteristics.

The *catalog.xml* document is a product catalog containing general information about products (Example 1-1).

Example 1-1. Product catalog input document (catalog.xml)

```
<catalog>
  <product dept="WMN">
    <number>557</number>
    <name language="en">Fleece Pullover</name>
    <colorChoices>navy black</colorChoices>
  </product>
  <product dept="ACC">
    <number>563</number>
    <name language="en">Floppy Sun Hat</name>
  </product>
  <product dept="ACC">
    <number>443</number>
    <name language="en">Deluxe Travel Bag</name>
  </product>
  <product dept="MEN">
    <number>784</number>
    <name language="en">Cotton Dress Shirt</name>
    <colorChoices>white gray</colorChoices>
    <desc>Our <i>favorite</i> shirt!</desc>
  </product>
</catalog>
```

The *prices.xml* document contains prices for most of the products, based on an effective date (Example 1-2).

Example 1-2. Price information input document (prices.xml)

```
<prices>
  <priceList effDate="2015-11-15">
    <prod num="557">
      <price currency="USD">29.99</price>
      <discount type="CLR">10.00</discount>
    </prod>
    <prod num="563">
      <price currency="USD">69.99</price>
    </prod>
    <prod num="443">
      <price currency="USD">39.99</price>
      <discount type="CLR">3.99</discount>
    </prod>
  </priceList>
</prices>
```

The *order.xml* document is a simple order containing a list of products ordered (referenced by a number that matches the number used in *catalog.xml*), along with quantities and colors (Example 1-3).

Example 1-3. Order input document (order.xml)

```
<order num="00299432" date="2015-09-15" cust="0221A">
  <item dept="WMN" num="557" quantity="1" color="navy"/>
  <item dept="ACC" num="563" quantity="1"/>
  <item dept="ACC" num="443" quantity="2"/>
  <item dept="MEN" num="784" quantity="1" color="white"/>
  <item dept="MEN" num="784" quantity="1" color="gray"/>
  <item dept="WMN" num="557" quantity="1" color="black"/>
</order>
```

Path Expressions

The most straightforward kind of query simply selects elements or attributes from an input document. This type of query is known as a path expression. For example, the path expression:

```
doc("catalog.xml")/catalog/product
```

will select all the `product` elements from the *catalog.xml* document.

Path expressions are used to traverse an XML tree to select elements and attributes of interest. They are similar to paths used for filenames in many operating systems. They consist of a series of steps, separated by slashes, that traverse the elements and attributes in the XML documents. In this example, there are three steps:

1. `doc("catalog.xml")` calls an XQuery function named `doc`, passing it the name of the file to open
2. `catalog` selects the `catalog` element, the outermost element of the document
3. `product` selects all the `product` children of `catalog`

The result of the query will be the four `product` elements, exactly as they appear (with the same attributes and contents) in the input document. Example 1-4 shows the complete result.

Example 1-4. Four `product` elements selected from the catalog

```
<product dept="WMN">
  <number>557</number>
  <name language="en">Fleece Pullover</name>
  <colorChoices>navy black</colorChoices>
</product>
<product dept="ACC">
  <number>563</number>
  <name language="en">Floppy Sun Hat</name>
</product>
<product dept="ACC">
  <number>443</number>
  <name language="en">Deluxe Travel Bag</name>
</product>
<product dept="MEN">
  <number>784</number>
  <name language="en">Cotton Dress Shirt</name>
  <colorChoices>white gray</colorChoices>
  <desc>Our <i>favorite</i> shirt!</desc>
</product>
```

The asterisk (*) can be used as a wildcard to indicate any element name. For example, the path expression:

```
doc("catalog.xml")/*/product
```

will return any `product` children of the outermost element, regardless of the outermost element's name. Alternatively, you can use a double slash (//) to return `product` elements that appear anywhere in the catalog document, as in:

```
doc("catalog.xml")//product
```

In addition to traversing the XML document, a path expression can contain predicates that filter out elements or attributes that do not meet a particular criterion. Predicates are indicated by square brackets. For example, the path expression:

```
doc("catalog.xml")/catalog/product[@dept = "ACC"]
```

contains a predicate. It selects only those `product` elements whose `dept` attribute value is `ACC`. The `@` sign is used to indicate that `dept` is an attribute as opposed to a child element.

When a predicate contains a number, it serves as an index. For example:

```
doc("catalog.xml")/catalog/product[2]
```

will return the second `product` element in the catalog.

Path expressions are convenient because of their compact, easy-to-remember syntax. However, they have a limitation: they can only return elements and attributes as they appear in input documents. Any elements selected in a path expression appear in the results with the same names, the same attributes and contents, and in the same order as in the input document. When you select the `product` elements, you get them with all of their children and with their `dept` attributes. Path expressions are covered in detail in Chapter 4.

FLWORs

The basic structure of many (but not all) queries is the FLWOR expression. FLWOR (pronounced "flower") stands for "for, let, where, order by, return," the most common keywords used in the expression.

FLWORs, unlike path expressions, allow you to manipulate, transform, and sort your results. Example 1-5 shows a simple FLWOR that returns the names of all products in the ACC department.

Example 1-5. Simple FLWOR

Query

```
for $prod in doc("catalog.xml")/catalog/product
where $prod/@dept = "ACC"
order by $prod/name
return $prod/name
```

Results

```
<name language="en">Deluxe Travel Bag</name>
<name language="en">Floppy Sun Hat</name>
```

As you can see, the FLWOR is made up of several parts:

`for`
: This clause sets up an iteration through the `product` elements, and the rest of the FLWOR is evaluated once for each of the four products. Each time, a variable

named `$prod` is bound to a different `product` element. Dollar signs are used to indicate variable names in XQuery.

`where`
: This clause selects only products in the ACC department. This has the same effect as a predicate (`[@dept = "ACC"]`) in a path expression.

`order by`
: This clause sorts the results by product name, something that is not possible with path expressions.

`return`
: This clause indicates that the `product` element's `name` children should be returned in the query result.

The `let` clause (the L in FLWOR) is used to bind the value of a variable. Unlike a `for` clause, it does not set up an iteration. Example 1-6 shows a FLWOR that returns the same result as Example 1-5. The second line is a `let` clause that binds the `product` element's `name` child to a variable called `$name`. The `$name` variable is then referenced later in the FLWOR, in both the `order by` clause and the `return` clause.

Example 1-6. Adding a `let` clause

```
for $prod in doc("catalog.xml")/catalog/product
let $name := $prod/name
where $prod/@dept = "ACC"
order by $name
return $name
```

The `let` clause serves as a programmatic convenience that avoids repeating the same expression multiple times. With some implementations, it may improve performance because the expression is evaluated only once instead of each time it is needed.

This chapter has provided only very basic examples of FLWORs. In fact, FLWORs can become quite complex. Multiple `for` clauses are permitted, which set up iterations within iterations. Additional clauses such as `group by`, `count`, and `window` are available. In addition, complex expressions can be used in any of the clauses. FLWORs are discussed in detail in Chapter 6. Even more advanced examples of FLWORs are provided in Chapter 9.

Adding XML Elements and Attributes

Sometimes you want to reorganize or transform the elements in the input documents into differently named or structured elements. XML constructors can be used to create elements and attributes that appear in the query results.

Adding Elements

Suppose you want to wrap the results of your query in a different XML vocabulary, for example, XHTML. You can do this using a familiar XML-like syntax. To wrap the `name` elements in a `ul` element, for instance, you can use the query shown in Example 1-7. The `ul` element represents an unordered list in HTML.

Example 1-7. Wrapping results in a new element

Query

```
<ul>{
  for $prod in doc("catalog.xml")/catalog/product
  where $prod/@dept='ACC'
  order by $prod/name
  return $prod/name
}</ul>
```

Results

```
<ul>
  <name language="en">Deluxe Travel Bag</name>
  <name language="en">Floppy Sun Hat</name>
</ul>
```

This example is the same as Example 1-5, with the addition of the first and last lines. In the query, the `ul` start tag and end tag, and everything in between, is known as an element constructor. The curly braces around the content of the `ul` element signify that it is an expression (known as an enclosed expression) that is to be evaluated. In this case, the enclosed expression returns two elements, which become children of `ul`.

Any content in an element constructor that is not inside curly braces appears in the results as is. For example:

```
<h1>There are {count(doc("catalog.xml")//product)} products.</h1>
```

will return the result:

```
<h1>There are 4 products.</h1>
```

The content outside the curly braces, namely the strings `"There are "` and `" products."`, appear literally in the results, as textual content of the `h1` element.

The element constructor does not need to be the outermost expression in the query. You can include element constructors at various places in your query. For example, if you want to wrap each resulting `name` element in its own `li` element, you could use the query shown in Example 1-8. An `li` element represents a list item in HTML.

Example 1-8. Element constructor in FLWOR `return` clause

Query

```
<ul>{
  for $prod in doc("catalog.xml")/catalog/product
  where $prod/@dept='ACC'
  order by $prod/name
  return <li>{$prod/name}</li>
}</ul>
```

Results

```
<ul>
  <li><name language="en">Deluxe Travel Bag</name></li>
  <li><name language="en">Floppy Sun Hat</name></li>
</ul>
```

Here, the `li` element constructor appears in the `return` clause of a FLWOR. Since the `return` clause is evaluated once for each iteration of the `for` clause, two `li` elements appear in the results, each with a `name` element as its child.

However, suppose you don't want to include the `name` elements at all, just their contents. You can do this by calling a built-in function called `data`, which extracts the contents of an element. This is shown in Example 1-9.

Example 1-9. Using the `data` function

Query

```
<ul>{
  for $prod in doc("catalog.xml")/catalog/product
  where $prod/@dept='ACC'
  order by $prod/name
  return <li>{data($prod/name)}</li>
}</ul>
```

Results

```
<ul>
  <li>Deluxe Travel Bag</li>
  <li>Floppy Sun Hat</li>
</ul>
```

Now no `name` elements appear in the results. In fact, no elements at all from the input document appear.

Adding Attributes

You can also add your own attributes to results using an XML-like syntax. Example 1-10 adds attributes to the `ul` and `li` elements.

Example 1-10. Adding attributes to results

Query

```
<ul type="square">{
  for $prod in doc("catalog.xml")/catalog/product
  where $prod/@dept='ACC'
  order by $prod/name
  return <li class="{$prod/@dept}">{data($prod/name)}</li>
}</ul>
```

Results

```
<ul type="square">
  <li class="ACC">Deluxe Travel Bag</li>
  <li class="ACC">Floppy Sun Hat</li>
</ul>
```

As you can see, attribute values, like element content, can either be literal text or enclosed expressions. The `ul` element constructor has an attribute `type` that is included as is in the results, while the `li` element constructor has an attribute `class` whose value is an enclosed expression delimited by curly braces. In attribute values, unlike element content, you don't need to use the `data` function to extract the value: it happens automatically.

The constructors shown in these examples are known as direct constructors, because they use an XML-like syntax. You can also construct elements and attributes with dynamically determined names, using computed constructors. Chapter 5 provides detailed coverage of XML constructors.

Functions

Almost 200 functions are built into XQuery, covering a broad range of functionality. Functions can be used to manipulate strings and dates, perform mathematical calculations, combine sequences of elements, and perform many other useful jobs. You can also define your own functions, either in the query itself, or in an external library.

Both built-in and user-defined functions can be called from almost any place in a query. For instance, Example 1-9 calls the `doc` function in a `for` clause, and the `data` function in an enclosed expression. Chapter 8 explains how to call functions and also

describes how to write your own user-defined functions. Appendix A lists all the built-in functions and explains each of them in detail.

Joins

One of the major benefits of FLWORs is that they can easily join data from multiple sources. For example, suppose you want to join information from your product catalog (*catalog.xml*) and your order (*order.xml*). You want a list of all the items in the order, along with their number, name, and quantity.

The name comes from the product catalog, and the quantity comes from the order. The product number appears in both input documents, so it is used to join the two sources. Example 1-11 shows a FLWOR that performs this join.

Example 1-11. Joining multiple input documents

Query

```
for $item in doc("order.xml")//item
let $name := doc("catalog.xml")//product[number = $item/@num]/name
return <item num="{$item/@num}"
             name="{$name}"
             quan="{$item/@quantity}"/>
```

Results

```
<item num="557" name="Fleece Pullover" quan="1"/>
<item num="563" name="Floppy Sun Hat" quan="1"/>
<item num="443" name="Deluxe Travel Bag" quan="2"/>
<item num="784" name="Cotton Dress Shirt" quan="1"/>
<item num="784" name="Cotton Dress Shirt" quan="1"/>
<item num="557" name="Fleece Pullover" quan="1"/>
```

The `for` clause sets up an iteration through each `item` from the order. For each item, the `let` clause goes to the product catalog and gets the name of the product. It does this by finding the `product` element whose `number` child equals the item's `num` attribute, and selecting its `name` child. Because the FLWOR iterated six times, the results contain one new `item` element for each of the six `item` elements in the order document. Joins are covered in Chapter 6.

Aggregating and Grouping Values

One common use for XQuery is to summarize and group XML data. It is sometimes useful to find the sum, average, or maximum of a sequence of values, grouped by a particular value. For example, suppose you want to know the number of items contained in an order, grouped by department. The query shown in Example 1-12 accomplishes this. It uses a `group by` clause to group the items by department, and

the `sum` function to calculate the totals of the `quantity` attribute values for the items in each department.

Example 1-12. Aggregating values

Query

```
xquery version "3.0";
for $i in doc("order.xml")//item
let $d := $i/@dept
group by $d
order by $d
return <department name="{$d}" totQuantity="{sum($i/@quantity)}"/>
```

Results

```
<department name="ACC" totQuantity="3"/>
<department name="MEN" totQuantity="2"/>
<department name="WMN" totQuantity="2"/>
```

Chapter 7 covers sorting, grouping, and aggregating values in detail. The version declaration on the first line of this example is used to show that use of the `group by` clause requires at least version 3.0 of XQuery.

Want to read more?

You can buy this book at oreilly.com
in print and ebook format.

Buy 2 books, get the 3rd FREE!

Use discount code OPC10

All orders over $29.95 qualify for **free shipping** within the US.

It's also available at your favorite book retailer, including the iBookstore, the Android Marketplace, and Amazon.com.

O'REILLY®

©2015 O'Reilly Media, Inc. The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. 15055

Contents

Copyright © 2010. John Wiley & Sons, Incorporated. All rights reserved.

Kay, Michael. XSLT 2.0 and XPath 2.0 Programmer's Reference, John Wiley & Sons, Incorporated, 2010. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/pitt-ebooks/detail.action?docID=514366.
Created from pitt-ebooks on 2018-04-03 11:18:24.

Copyright © 2010. John Wiley & Sons, Incorporated. All rights reserved.

Kay, Michael. XSLT 2.0 and XPath 2.0 Programmer's Reference, John Wiley & Sons, Incorporated, 2010. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/pitt-ebooks/detail.action?docID=514366.
Created from pitt-ebooks on 2018-04-03 11:18:24.

Copyright © 2010. John Wiley & Sons, Incorporated. All rights reserved.

Kay, Michael. XSLT 2.0 and XPath 2.0 Programmer's Reference, John Wiley & Sons, Incorporated, 2010. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/pitt-ebooks/detail.action?docID=514366.
Created from pitt-ebooks on 2018-04-03 11:18:24.

Copyright © 2010. John Wiley & Sons, Incorporated. All rights reserved.

Kay, Michael. XSLT 2.0 and XPath 2.0 Programmer's Reference, John Wiley & Sons, Incorporated, 2010. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/pitt-ebooks/detail.action?docID=514366.
Created from pitt-ebooks on 2018-04-03 11:18:24.

Copyright © 2010. John Wiley & Sons, Incorporated. All rights reserved.

Copyright © 2010. John Wiley & Sons, Incorporated. All rights reserved.

Kay, Michael. XSLT 2.0 and XPath 2.0 Programmer's Reference, John Wiley & Sons, Incorporated, 2010. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/pitt-ebooks/detail.action?docID=514366.
Created from pitt-ebooks on 2018-04-03 11:18:24.

Copyright © 2010. John Wiley & Sons, Incorporated. All rights reserved.

Kay, Michael. XSLT 2.0 and XPath 2.0 Programmer's Reference, John Wiley & Sons, Incorporated, 2010. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/pitt-ebooks/detail.action?docID=514366.
Created from pitt-ebooks on 2018-04-03 11:18:24.

Copyright © 2010. John Wiley & Sons, Incorporated. All rights reserved.

Kay, Michael. XSLT 2.0 and XPath 2.0 Programmer's Reference, John Wiley & Sons, Incorporated, 2010. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/pitt-ebooks/detail.action?docID=514366.
Created from pitt-ebooks on 2018-04-03 11:18:24.

Copyright © 2010. John Wiley & Sons, Incorporated. All rights reserved.

Kay, Michael. XSLT 2.0 and XPath 2.0 Programmer's Reference, John Wiley & Sons, Incorporated, 2010. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/pitt-ebooks/detail.action?docID=514366.
Created from pitt-ebooks on 2018-04-03 11:18:24.

Contents

Copyright © 2010. John Wiley & Sons, Incorporated. All rights reserved.

Kay, Michael. XSLT 2.0 and XPath 2.0 Programmer's Reference, John Wiley & Sons, Incorporated, 2010. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/pitt-ebooks/detail.action?docID=514366.
Created from pitt-ebooks on 2018-04-03 11:18:24.

Copyright © 2010. John Wiley & Sons, Incorporated. All rights reserved.

Kay, Michael. XSLT 2.0 and XPath 2.0 Programmer's Reference, John Wiley & Sons, Incorporated, 2010. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/pitt-ebooks/detail.action?docID=514366.
Created from pitt-ebooks on 2018-04-03 11:18:24.

Copyright © 2010. John Wiley & Sons, Incorporated. All rights reserved.

Kay, Michael. XSLT 2.0 and XPath 2.0 Programmer's Reference, John Wiley & Sons, Incorporated, 2010. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/pitt-ebooks/detail.action?docID=514366.
Created from pitt-ebooks on 2018-04-03 11:18:24.

Part III: Exploitation

Copyright © 2010. John Wiley & Sons, Incorporated. All rights reserved.

Kay, Michael. XSLT 2.0 and XPath 2.0 Programmer's Reference, John Wiley & Sons, Incorporated, 2010. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/pitt-ebooks/detail.action?docID=514366.
Created from pitt-ebooks on 2018-04-03 11:18:24.

Copyright © 2010. John Wiley & Sons, Incorporated. All rights reserved.

Kay, Michael. XSLT 2.0 and XPath 2.0 Programmer's Reference, John Wiley & Sons, Incorporated, 2010. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/pitt-ebooks/detail.action?docID=514366.
Created from pitt-ebooks on 2018-04-03 11:18:24.

Part IV: Appendices

Copyright © 2010. John Wiley & Sons, Incorporated. All rights reserved.

Kay, Michael. XSLT 2.0 and XPath 2.0 Programmer's Reference, John Wiley & Sons, Incorporated, 2010. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/pitt-ebooks/detail.action?docID=514366.
Created from pitt-ebooks on 2018-04-03 11:18:24.

Copyright © 2010. John Wiley & Sons, Incorporated. All rights reserved.

Kay, Michael. XSLT 2.0 and XPath 2.0 Programmer's Reference, John Wiley & Sons, Incorporated, 2010. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/pitt-ebooks/detail.action?docID=514366.
Created from pitt-ebooks on 2018-04-03 11:18:24.

Copyright © 2010. John Wiley & Sons, Incorporated. All rights reserved.

Kay, Michael. XSLT 2.0 and XPath 2.0 Programmer's Reference, John Wiley & Sons, Incorporated, 2010. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/pitt-ebooks/detail.action?docID=514366.
Created from pitt-ebooks on 2018-04-03 11:18:24.

I've arranged the functions in alphabetical order (combining the XPath-defined and XSLT-defined functions into a single sequence), so you can find a function quickly if you know what you're looking for. However, in case you only know the general area you are interested in, you may find the classification that follows in the section *Functions by Category* useful. This is followed by a section called *Notation*, which describes the notation used for function specifications in this chapter. The rest of the chapter is taken up with the functions themselves, in alphabetical order.

A Word about Naming

Function names such as `current-dateTime()` seem very strange when you first come across them. Why the mixture of camelCasing and hyphenation? The reason they arise is that XPath 1.0 decided to use hyphenated lower-case names for all functions, while XML Schema decided to use camelCase for the names of built-in types. Wherever the XPath 2.0 function library uses a schema-defined type name as part of a function name, it therefore uses the camelCase type name as a single word within the hyphenated function name.

So it may be madness, but there is method in it!

Throughout this book, I write these function names without a namespace prefix. In fact the functions are defined to be within the namespace `http://www.w3.org/2005/xpath-functions`, which is often referred to using the namespace prefix «`fn`». (Earlier drafts of the specification used different namespaces, which you may still encounter). In XSLT this is the default namespace for function names, so you will never need to write them with a namespace prefix. I have therefore omitted the prefix when referring to the names in this book. In the W3C specifications, however, you will often see the functions referred to by names such as `fn:position()` or `fn:count()`.

Functions by Category

Any attempt to classify functions is bound to be arbitrary, but I'll attempt it anyway. A few functions appear in more than one category. The number after each function is a page reference to the entry where the function is described. Functions marked † are available in XSLT only (that is, they are not available when executing freestanding XPath expressions or in XQuery).

Boolean Functions

`boolean()` 721, `false()` 779, `not()` 850, `true()` 899.

Numeric Functions

`abs()` 714, `avg()` 718, `ceiling()` 723, `floor()` 779, †`format-number()` 788, `max()` 830, `min()` 830, `number()`851, `round()` 870, `round-half-to-even()` 872, `sum()` 889.

String Functions

`codepoints-to-string()` 725, `compare()` 727, `concat()` 729, `contains()` 730, `ends-with()` 773, `lower-case()` 827, `matches()` 828, `normalize-space()` 845, `normalize-unicode()` 847, `replace()` 862, `starts-with()` 875, `string()` 877, `string-join()` 879, `string-length()` 880, `string-to-codepoints()` 881, `substring()` 883, `substring-after()` 885, `substring- before()` 887, `tokenize()` 894, `upper-case()` 910.

Copyright © 2010. John Wiley & Sons, Incorporated. All rights reserved.

Kay, Michael. XSLT 2.0 and XPath 2.0 Programmer's Reference, John Wiley & Sons, Incorporated, 2010. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/pitt-ebooks/detail.action?docID=514366.
Created from pitt-ebooks on 2018-03-29 08:01:33.

Date and Time Functions

adjust-date-to-timezone() 715, adjust-dateTime-to-timezone() 715, adjust-time-to-timezone() 715, current-date() 738, current-dateTime() 738, current-time() 738, day-from-date() 744, day-from-dateTime() 744, †format-date() 781, †format-dateTime() 781, †format-time() 781, hours-from-dateTime() 800, hours-from-time() 800, implicit-timezone() 806, minutes-from-dateTime() 832, minutes-from-time() 832, month-from-date() 833, month-from-dateTime() 833, seconds-from-dateTime() 873, seconds-from-time() 873, timezone-from-date() 893, timezone-from-dateTime() 893, timezone-from-time() 893, year-from-date() 911, year-from-dateTime() 911.

Duration Functions

days-from-duration() 745, hours-from-duration() 801, minutes-from-duration() 832, months-from-duration() 834, seconds-from-duration() 874, years-from-duration() 911.

Aggregation Functions

avg() 718, count() 733, max() 830, min() 830, sum() 889.

Functions on URIs

base-uri() 719, collection() 726, doc() 750, doc-available() 750, document-uri() 764, encode-for-uri() 771, escape-html-uri() 775, iri-to-uri() 811, resolve-uri() 867, static-base-uri() 876, †unparsed-text() 904, †unparsed-text-available() 904.

Functions on QNames

local-name-from-QName()826, namespace-uri-from-QName()841, node-name() 843, prefix-from-QName() 857, QName() 858, resolve-QName() 864.

Functions on Sequences

count() 733, deep-equal() 745, distinct-values() 749, empty() 770, exists() 778, index-of() 807, insert-before() 810, remove() 861, subsequence() 882, unordered() 901.

Functions That Return Properties of Nodes

base-uri() 719, data() 741, document-uri() 764, †generate-id() 797, in-scope-prefixes() 808, lang() 819, local-name() 824, name() 835, namespace-uri() 837, namespace-uri-for-prefix() 839, nilled() 842, node-name() 843, root() 870, string() 877, †unparsed-entity-public-id() 902, †unparsed-entity-uri() 902.

Functions That Find Nodes

collection() 726, doc() 750, †document() 754, id() 802, idref() 804, †key() 812, root() 870.

Functions That Return Context Information

base-uri() 719, collection() 726, †current() 734, current-date() 738, current-dateTime() 738, †current-group() 739, †current-grouping-key() 740, current-time() 738, default-collation() 748, doc() 750, implicit-timezone() 806, last() 820, position() 854, †regex-group() 860.

Copyright © 2010. John Wiley & Sons, Incorporated. All rights reserved.

Kay, Michael. XSLT 2.0 and XPath 2.0 Programmer's Reference, John Wiley & Sons, Incorporated, 2010. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/pitt-ebooks/detail.action?docID=514366.
Created from pitt-ebooks on 2018-03-29 08:01:43.

Diagnostic Functions

`error()` 774, `trace()` 896.

Functions That Return Information about the XSLT Environment

`†element-available()` 764, `†function-available()` 792, `†system-property()`890, `†type-available()`899

Functions That Assert a Static Type

`exactly-one()` 777, `one-or-more()` 853, `zero-or-one()` 912.

Notation

For each function (or for a closely related group of functions) there is an alphabetical entry in this chapter containing the following information:

- The name of the function
- A summary of the purpose of the function, often with a quick example
- *Changes in 2.0*. In cases where a function was present in XSLT 1.0 or XPath 1.0, the entry for the function in this chapter contains a section that describes any changes in behavior introduced in the 2.0 version of the specs. If there are no changes, this section will say so. In cases where the function is new in XPath 2.0 or XSLT 2.0, this section is omitted.
- The function signature, described below
- A section entitled *Effect*, which describes in fairly formal terms what the function does
- Where appropriate, a section entitled *Usage*, which give advice on how to make best use of the function
- A set of simple examples showing the function in action
- Cross-references to other related information in this book

Technically, a function in XPath is identified by its name and arity (number of arguments). This means that there is no formal relationship between the function `substring()` with two arguments and the function `substring()` with three arguments. However, the standard function library has been designed so that in cases like this where there are two functions with different arity, the functions in practice have a close relationship, and it is generally easier to think of them as representing one function with one or more of the arguments being optional. So this is how I have presented them.

The signatures of functions are defined with a table like the one that follows:

Argument	Type	Meaning
input	`xs:string?`	The containing string
start	`xs:double`	The position in the containing string of . . .
length (optional)	`xs:double`	The number of characters to be included . . .
Result	*`xs:string`*	*The required substring . . .*

Copyright © 2010. John Wiley & Sons, Incorporated. All rights reserved.

Kay, Michael. XSLT 2.0 and XPath 2.0 Programmer's Reference, John Wiley & Sons, Incorporated, 2010. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/pitt-ebooks/detail.action?docID=514366.
Created from pitt-ebooks on 2018-03-29 08:01:52.

The first column here gives a conventional name for the argument (or ''Result'' to label the row that describes the result of the function). Arguments to XPath functions are supplied by position, not by name, so the name given here is arbitrary; it is provided only to allow the argument to be referred to within the descriptive text. The text ''(optional)'' after the name of an argument indicates that this argument does not need to be supplied; in this case, this means that there is one version of the function with two arguments, and another version with three.

The second column gives the required type of the argument. The notation is that of the `SequenceType` syntax in XPath, introduced in Chapter 11. This consists of an item type followed optionally by an occurrence indicator («?», «*», or «+»). The item type is either the name of a built-in atomic type such as `xs:integer` or `xs:string`, or one of the following:

Item type	Meaning
`item()`	Any item (either a node or an atomic value)
`node()`	Any node
`element()`	Any element node
`xs:anyAtomicType`	Any atomic value
`Numeric`	An `xs:double`, `xs:float`, `xs:decimal`, or `xs:integer`

The occurrence indicator, if it is present, is either «?» to indicate that the supplied argument can contain zero or one items of the specified item type, or «*» to indicate that it can be a sequence of zero or more items of the specified item type. (The occurrence indicator «+», meaning one or more, is not used in any of the standard functions.)

Note the difference between an argument that is optional, and an argument that has an occurrence indicator of «?». When the argument is optional, it can be omitted from the function call. When the occurrence indicator is «?», the value must be supplied, but the empty sequence «()» is an acceptable value for the argument.

Many functions follow the convention of allowing an empty sequence for the first argument, or for subsequent arguments that play a similar role to the first argument, and returning an empty sequence if any of these arguments is an empty sequence. This is designed to make these functions easier to use in predicates. However, this is only a convention, and it is not followed universally. Most of the string functions instead treat an empty sequence the same way as a zero-length string.

When these functions are called, the supplied arguments are converted to the required type in the standard way defined by the XPath 2.0 function calling mechanism. The details of this depend on whether XPath 1.0 backward compatibility is activated or not. In XSLT this depends on the value of the `[xsl:]version` attribute in the stylesheet, as follows:

- In 2.0 mode, the standard conversion rules apply. These rules appear in Chapter 6 on page 505, under the heading *Converting the Arguments and the Result*. They permit only the following kinds of conversion:
  - Atomization of nodes to extract their numeric values
  - Promotion of numeric values to a different numeric type; for example, `xs:integer` to `xs:double`
  - Promotion of `xs:anyURI` values to `xs:string`

Copyright © 2010. John Wiley & Sons, Incorporated. All rights reserved.

Kay, Michael. XSLT 2.0 and XPath 2.0 Programmer's Reference, John Wiley & Sons, Incorporated, 2010. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/pitt-ebooks/detail.action?docID=514366.
Created from pitt-ebooks on 2018-03-29 08:01:52.

- Casting of a value of type `xs:untypedAtomic` to the required type. Such values generally arise by extracting the content of a node that has not been schema-validated. The rules for casting from `xs:untypedAtomic` values to values of other types are essentially the rules defined in XML Schema for conversion from the lexical space of the type to the value space: more details are given in Chapter 11 (see *Converting from string* on page 663).

- In 1.0 mode, two additional conversions are allowed:
  - If the required type is `xs:string` or `xs:double` (perhaps with an occurrence indicator of «?»), then the first value in the supplied sequence is converted to the required type using the `string()` or `number()` function as appropriate, and other values in the sequence are discarded.
  - If the required type is `node()` or `item()` (perhaps with an occurrence indicator of «?»), then if the supplied value contains more than one item, all items except the first are ignored.

The effect of these rules is that even though the function signature might give the expected type of an argument as `xs:string`, say, the value you supply can be a node containing a string, or a node whose value is untyped (because it has not been validated using a schema), or an `xs:anyURI` value. With 1.0 compatibility mode on, you can also supply values of other types; for example, an `xs:integer` or an `xs:date`; but when compatibility mode is off, you will need to convert such values to an `xs:string` yourself, which you can achieve most simply by calling the `string()` function.

Code Samples

Most of the examples for this chapter are single XPath expressions. In the download file for this book, these code snippets are gathered into stylesheets, which in turn are organized according to the name of the function they exercise. In many cases the examples use no source document, in which case the stylesheet generally has a single template named `main`, which should be used as the entry point. In other cases the source document is generally named `source.xml`, and it should be used as the principal input to the stylesheet. Any stylesheets that require a schema-aware processor have names of the form `xxx-sa.xsl`.

Function Definitions

The remainder of this chapter gives the definitions of all the functions, in alphabetical order.

abs

The `abs()` function returns the absolute value of a number. For example, «`abs(-3)`» returns 3.

Signature

Argument	Type	Meaning
input	`Numeric?`	The supplied number.
Result	*`Numeric?`*	*The absolute value of the supplied number. The result has the same type as the input.*

Effect

If the supplied number is positive, then it is returned unchanged. If it is negative, then the result is «`-$input`».

Copyright © 2010. John Wiley & Sons, Incorporated. All rights reserved.

Kay, Michael. XSLT 2.0 and XPath 2.0 Programmer's Reference, John Wiley & Sons, Incorporated, 2010. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/pitt-ebooks/detail.action?docID=514366.
Created from pitt-ebooks on 2018-03-29 08:02:02.