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 Archeology and Project Management

course number 44

when: offered in week 2 of the Digital Humanities Summer Institute: 11-15 June, 2018

Link to Register for DHSI Courses

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.
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

XPath for Document Archaeology and Project Management: Syllabus

Digital Humanities Summer Institute Week 2 (June 11–15, 2018)

Daily schedule

Time	Session
9:30am–12:00pm	Morning session
12:00pm–1:15pm	Lunch
1:15pm–3:50pm	Afternoon session

Classes start at 10:00 on Monday and end at 12:00 on Friday.

Before you arrive

Please install on a laptop that you bring with you:

- <oXygen/> XML Editor
- Optional: eXist-db

Monday, June 11: XPath

Morning (10:00am–12:00pm)

- Introduction to XPath in eXist-db
- Simple XPath expressions
- XPath axes and context nodes
- XPath path steps

Afternoon (1:15pm–3:50pm)

- Explore document structures and data
- XPath functions
- XPath function signatures and cardinality
- XPath predicates
- Read and evaluate XML projects with XPath

Tuesday June 12: XPath and XQuery

Morning (9:30am–12:00pm)

- The seven types of nodes
- Neglected XPath Axes (`ancestor::` , `preceding::` and `following::` , `self::` , etc.)
- Don’t confuse XPath predicates with path steps
- Regex in XPath
- XPath in XQuery

Afternoon (1:15pm–3:50pm)

- XPath expressions with functions and predicates
- `for` loops; sequence and range variables
- FLWOR statements (XQuery)
- XPath predicates and the `where` statement

Wednesday June 13: XPath and XSLT

Morning (9:30am–12:00pm)

- Introduction to XSLT
- “It’s always a namespace issue”: working with multiple namespaces

- XSLT template matching and context nodes
- Identity transformation

Afternoon (1:15pm–3:50pm)

- `<xsl:apply-templates>` and the `@select` attribute
- XSLT push and pull processing
- When to use `<xsl:value-of>`
- Comparing XSLT and XQuery

Thursday June 14: XPath and Schematron

Morning (9:30am–12:00pm)

- Schematron: using XPath path to constrain your markup
- XPath functions practice
- Grouping; looping over distinct values, mapping back to the tree

Afternoon (1:15pm–3:50pm)

- Writing Schematron with XPath functions
- Lab session (sample activities provided, or work on your own data)

Friday June 15: Taking stock

Morning (9:30am–12:00pm)

- Review of XPath and its applications
- What next?

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)
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:

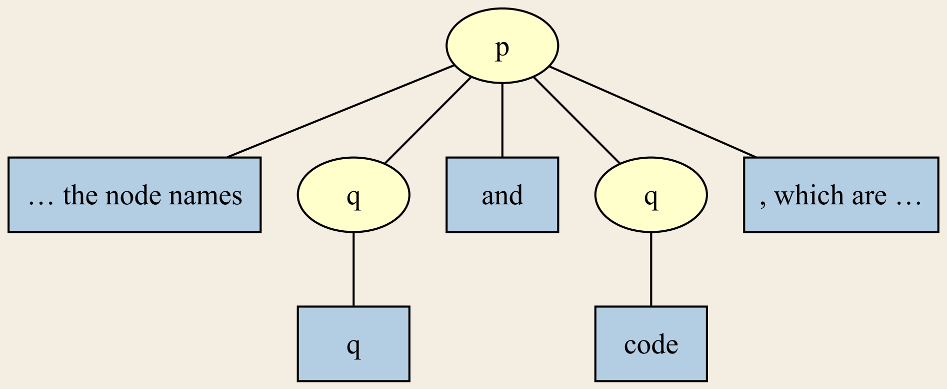
1. XML is a formal model designed to represent an ordered hierarchy, and to the extent that human documents are logically ordered and hierarchical, they can be formalized and represented easily as XML documents.
2. Computers can operate very quickly and efficiently on trees (ordered hierarchies), much 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:

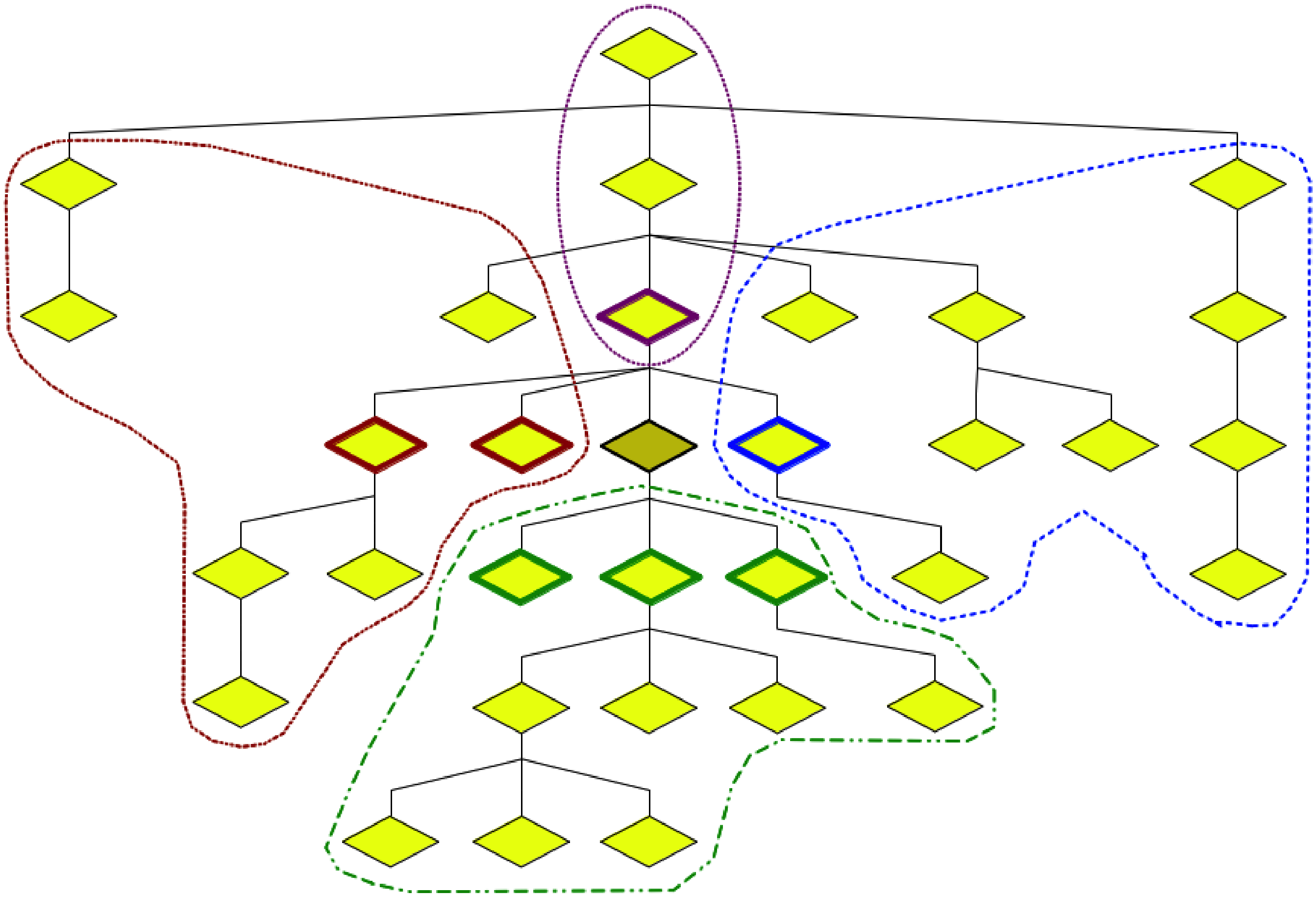
- `child`: All nodes contained directly by the current context node.
- `descendant`: All nodes contained directly by the current context node, recursively, that is, all the way down the tree. In other words, the descendants of a node are its children, its children's children, etc.
- `parent`: The node 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).

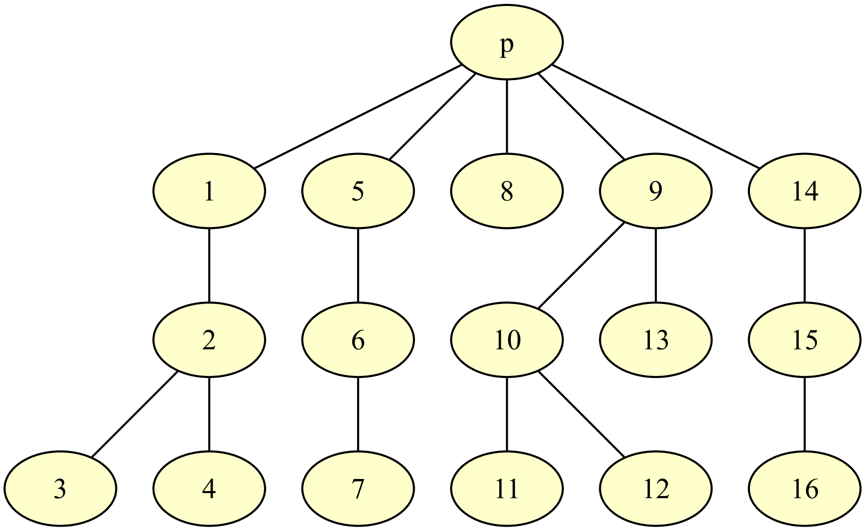
A double slash (`//`) is shorthand for the `descendant` axis, so that `chapter//quote` would first find all of the `<chapter>` elements that are children of the current context and then find all of the `<quote>` elements anywhere within them, at any depth (children, children's children, etc.). When used 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

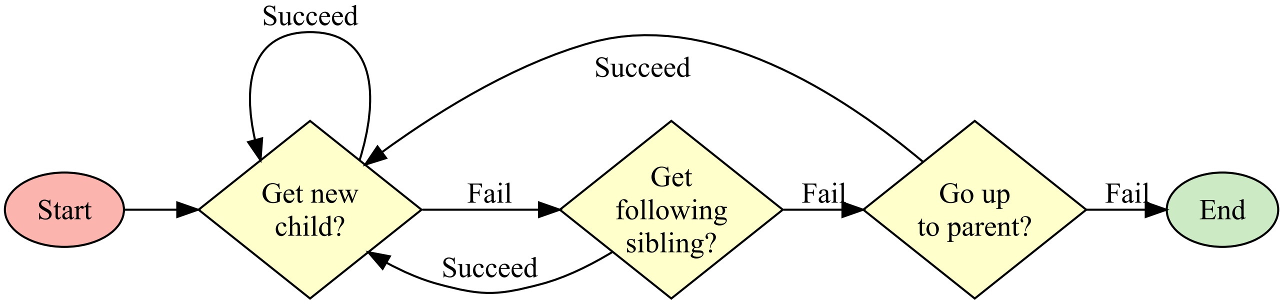
Predicates are used to filter the results of path expressions. The sequences that are returned by path expressions have an inherent and stable order, which is called *document order*. In XPath, document order is defined as *depth first*, which means that when the system has to return nodes in order, 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 **&gt;**)
- **>=** greater than or equal to (not less than; may also be written **&gt;=**)
- **<** less than (may also be written **&lt;**)
- **<=** less than or equal to (not greater than; may also be written **&lt;=**)

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	> (&gt;)
Greater than or equal to (not less than)	ge	>= (&gt;=)
Less than	lt	< (&lt;)
Less than or equal to (not greater than)	le	<= (&lt;=)

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 &amp;
2. Then change < to &lt; and > to &gt;
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>
```

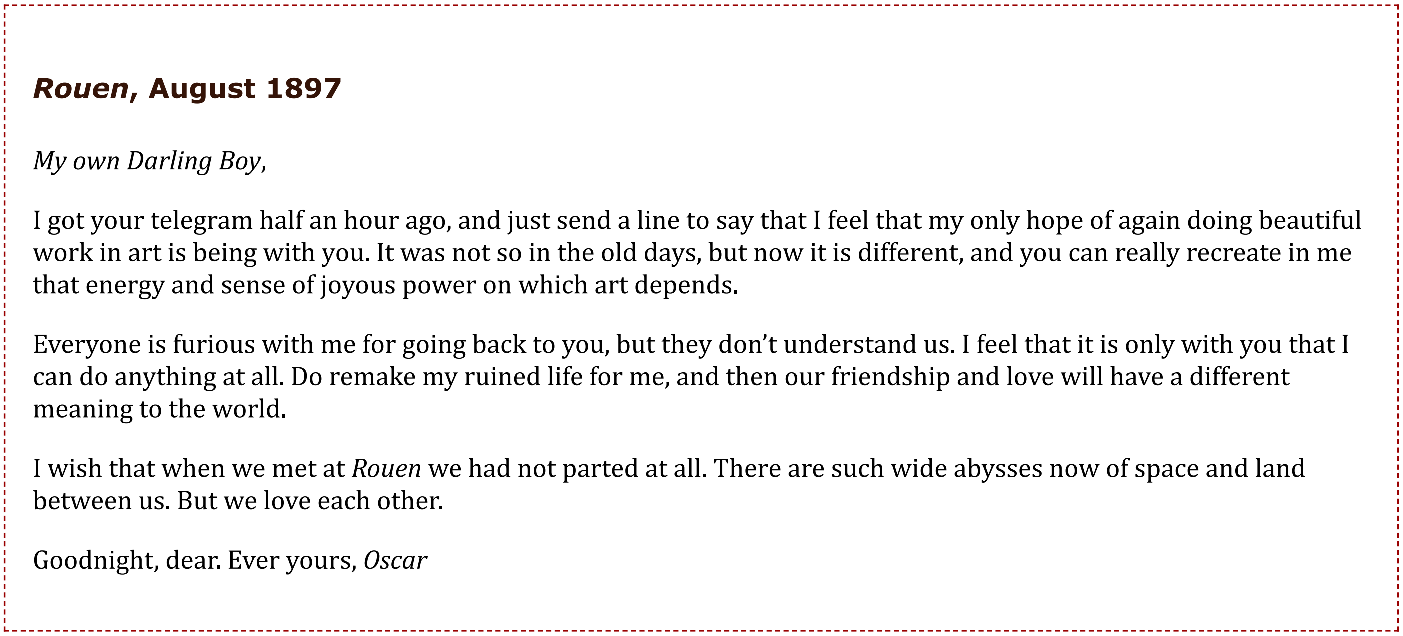
Putting it all together

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.

```
<letter>
    <head>
        <context>The following letter was written shortly after Wilde's
        release from prison:</context>
    </head>
    <content>
        <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)
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.

newtFire {dhlds}
Maintained by: Elisa E. Beshero-Bondar (ebb8 at pitt.edu) 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 <div type="chapter"> 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 <div> elements for Books, and the <div> 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)
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”.

<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`.

newtFire {dh|ds}
Authored by: Rob Spadafore (spadafour at gmail.com) Edited and maintained by: Elisa E. Beshero-Bondar (ebb8 at pitt.edu)
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 `&lt;`
- 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

Supplemental Readings

O'REILLY®

2nd Edition

XQuery

SEARCH ACROSS A VARIETY
OF XML DATA

Free Sampler

Priscilla Walmsley

SECOND EDITION

XQuery

Search Across a Variety of XML Data

Priscilla Walmsley

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

XQuery
by Priscilla Walmsley

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

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.

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.

Table of Contents

Contents

Part I: Foundations

Part II: XSLT and XPath Reference

Part III: Exploitation

Part IV: Appendices

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.

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.

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.

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 . . .*

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`

    - 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».

with a schema that defines the colors attribute with type `xs:NMTOKENS` (that is, it allows a list of colors to be specified, but our sample document only specifies one).

Expression	Result
`string(@colors)`	Succeeds unless the processor is doing static type checking, in which case it gives a compile-time error because the argument to `string()` must be a sequence of zero or one items

`string(`
`zero-or-`
`one(@colors))`
	Succeeds whether the processor is doing static type checking or not, because the check that the typed value of `@colors` contains at most one item is deferred until runtime

Usage

This function is never needed unless you are using a processor that does static type checking.

However, you may still find it useful as a way of inserting runtime checks into your XPath expressions, and documenting the assumptions you are making about the input data.

See Also

`exactly-one()` on page 777
`one-or-more()` on page 853

Summary

Much of the power of any programming language comes from its function library, which is why this chapter explaining the function library is one of the longest in the book. The size of the function library has grown greatly since XSLT and XPath 1.0, largely because of the richer set of types supported.

The next chapter defines the syntax of the regular expressions accepted by the three functions `matches()`, `replace()`, and `tokenize()`, and by the XSLT instruction `<xsl:analyze-string>`.