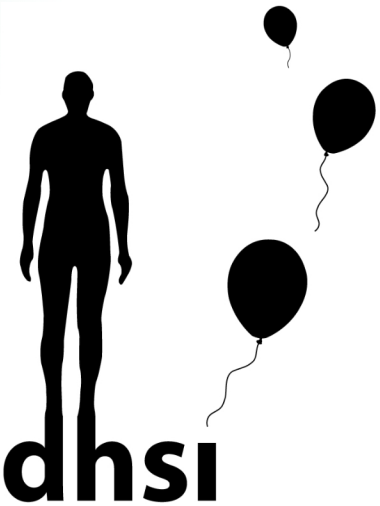
Introduction to XSLT for Digital Humanists

Syd Bauman
Martin Holmes



University of Victoria
etcl Electronic Textual Cultures Lab
Social Sciences and Humanities Research Council of Canada
Conseil de recherches en sciences humaines du Canada

Daily Schedule

Wednesday, 5 June 2013 [DHSI Registration]

4:00-6.00	DHSI Registration At UVic Housing / Residence Services Office (Craigdarroch Building) See the University of Victoria @ Google Maps After registration, many will wander to Cadboro Bay and the pub at Smuggler's Cove.
Recharge Room! (9.00-5.00, Thursday-Monday)	Hickman 116 will be available from 9-5 for the duration of DHSI for those who wish to take a moment to recharge amidst all the activities of DHSI. It is a space dedicated to individual peace and quiet, where people can go to eat / rest / read / whatever, and where we ask that mobile devices be silenced.

Thursday, 6 June 2013

7:45 to 8:15	Last-minute Registration MacLaurin Building, Room A100 See the University of Victoria @ Google Maps
8:30 to 9:30	Welcome, Overview and Orientation MacLaurin A144
9:30 to Noon	Classes in Session 1. Text Encoding Fundamentals and their Application (Clearihue A102, Lab) 2. Digitisation Fundamentals and their Application (Clearihue A051, Lab) 3. Introduction to XSLT for Digital Humanists (Clearihue A103, Lab) 4. Transcribing and Describing Primary Sources in TEI (Clearihue A015, Lab) 5. Multimedia Design for Research Creation, Community Engagement, and Knowledge Mobilization (Clearihue A108, Lab) 6. Geographical Information Systems in the Digital Humanities (Human and Social Development A170, Lab) 7. Digital Pedagogy in the Humanities (Hickman 120, Classroom) 8. Physical Computing and Desktop Fabrication for Humanists (MacLaurin D016, Classroom) 9. Creating Digital Humanities Projects for the Mobile Environment (Human and Social Development A270, Lab) 10. Digital Humanities Databases (Hickman 110, Classroom) 11. Augmented Reality: An Introduction (MacLaurin D010, Classroom) 12. Games for Digital Humanists (MacLaurin D111, Classroom [at times CLE A105]) 13. Visual Design for Digital Humanists (MacLaurin D287, Classroom) 14. Issues in Large Project Planning and Development (Hickman 128, Classroom) 15. Digital Editions (Clearihue A012, Lab) 16. Out-of-the-Box Text Analysis for the Digital Humanities (Human and Social Development A160, Lab) 17. SEASR Analytics (MacLaurin A326, Classroom) 18. Versioning & Collation in the Digital Environment (Human and Social Development A150, Lab) 19. 3D Modelling for the Digital Humanities and Social Sciences (Clearihue C115, Classroom) 20. Cultural Codes and Protocols for Indigenous Digital Heritage Management (MacLaurin A195, Classroom) 22. Understanding the Pre-Digital Book (McPherson Library A003 [at times A130], Classroom)
Noon to 1:15	Lunch break / Unconference Coordination Session MacLaurin A144 (Unconference discussions through the week are coordinated by Sarah Kremen-Hicks and Paige Morgan; discussion topics, scheduling, and room assignments from among all DHSI rooms will be handled at this meeting)
1:15 to 3:50	Classes in Session (Locations as above)
4:00 to 5:00	Institute Lecture: George Dyson MacLaurin A144
5:00 to 6:00	Electronic Literature Online Exhibit Conviviality and Light Reception MacLaurin A100 (just outside the lecture hall in A144)

Friday, 7 June 2013

8:00 to 9:20	DHSI Colloquium MacLaurin A144
9:30 to Noon	Classes in Session (Locations as above)
Noon to 1:15	Lunch break / Unconference, various locations
1:15 to 3:50	Classes in Session (Locations as above)
4:00 to 5:30/6.00	DHSI Colloquium MacLaurin A144

Saturday, 8 June 2013

8:00 to 9:20	DHSI Colloquium MacLaurin A144
9:30 to Noon	Classes in Session (Locations as above)
Noon to 1:15	Lunch break / Unconference, various locations
1:15 to 3:50	Classes in Session (Locations as above)
4:00 to 5:30/6.00	DHSI Colloquium MacLaurin A144

Sunday, 9 June 2013

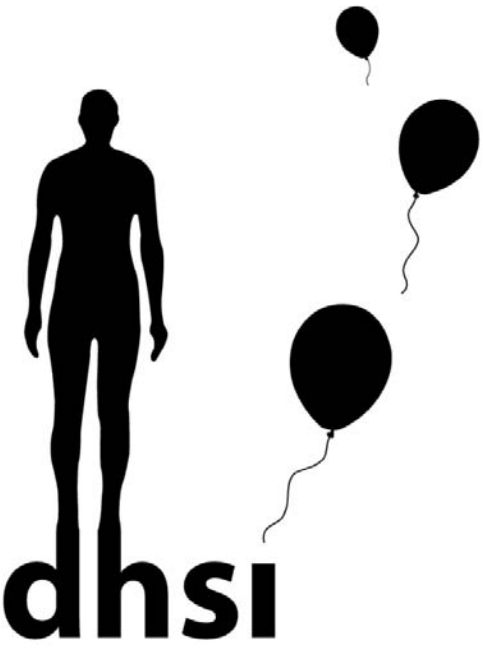
8:00 to 9:20	DHSI Colloquium MacLaurin A144
9:30 to Noon	Classes in Session (Locations as above)
Noon to 1:15	Lunch break / Unconference, various locations
1:15 to 3:50	Classes in Session (Locations as above)
4:00 to 5:30/6.00	DHSI Colloquium MacLaurin A144

Monday, 10 June 2013

8:00 to 9:20	DHSI Colloquium MacLaurin A144
9:30 to Noon	Classes in Session (Locations as above)
Noon to 1:15	Lunch break / Unconference, various locations
1:15 to 2:15	Institute Lecture: Kari Kraus (U Maryland) MacLaurin A144
2:15-4:30 (or so)	DHSI Wrap-up Session, Show and Tell MacLaurin A144

Contact info:
institut@uvic.ca ***P:*** *250-472-5401* ***F:*** *250-472-5681*


University
of Victoria
etcl
Electronic Textual
Cultures Lab


dhsi

INTRODUCTION TO XSLT FOR DIGITAL HUMANISTS

Syd Bauman, Brown University (Syd_Bauman@Brown.edu)
Martin Holmes, University of Victoria (mholmes@uvic.ca)

URL

The index page for the workshop can be found at
`http://www.wwp.brown.edu/outreach/seminars/uvic_xslt_2013/index.html`

SCHEDULE

Thu 06 Jun

Session 1, 09:30–12:00

- Welcome and introductions (SB); logistics, etc. (MH)
- Overview: where does XSLT fit in, and what we will (not) cover (SB; slides: Slidy, static)
- Introduction to XSLT (SB; slides: Slidy, static)
  - hello world input, hello world stylesheet
  - verse input, verse stylesheet

Session 2, 13:15–15:50

- Group Exploration: simple transformations (SB; slides: Slidy, static)
  - prose input, prose stylesheet
- Executing an XSLT program in oXygen (MH; slides: Slidy, static)
- Hands-on exercises
- Anatomy of an XSLT stylesheet (MH; slides: Slidy, static)
- Group Exploration: repetitive data (SB; exercises)
  - personography, personography stylesheet
  - structured personography, structured personography stylesheet
- hands-on practice

Fri 07 Jun

Session 3, 09:30–12:00

- Questions (SB)
- Quiz (MH)
- Navigating the XML tree and selecting nodes: XPath path expressions (SB; slides: Slidy, static — *OR* — Slidy, static; cribsheet: HTML, TEI)
- Group Exploration: more complex transformations (SB)
  - calculating rudimentary statistics: personography (again), statistics stylesheet
  - making a change to a file using an “identity” transform: personography (again), add-an-ID stylesheet

Sat 08 Jun

Session 5, 09:30–12:00
- Questions (SB)
- Quiz (MH)
- conditionals (MH; slides: Slidy, static)
- hands-on

Session 6, 13:15–15:50
- Template Modes (i.e., repeated use of input) (MH; slides: Slidy, static)
- hands-on

Sun 09 Jun

Session 7, 09:30–12:00
- Questions (SB)
- Quiz (MH)
- hands-on

Session 8, 13:15–15:50
- hands-on

Mon 10 Jun

Session 9, 09:30–12:00
- Questions (SB)
- Quiz (MH)
- final presentation prep
- wrap-up (SB; slides: Slidy, static)

BIBLIOGRAPHY

Many slides and handouts refer to the page or chapter of the reference work that we personally use:

Kay, Michael. *XSLT 2.0 and XPath 2.0 Programmer's Reference*. 4th edition. Indianapolis, IN: Wiley/Wrox, 2008. ISBN: 978-0-470-19274-0

References to "Kay" are to this book.

Extracts from *XSLT 2.0 and XPath 2.0 Programmer's Reference* 4th edition, by Michael Kay; published by Wiley Publishing, Inc. ISBN: 978-0-470-19274-0. (Try searching for that ISBN at www.wrox.com.)

Note: subsections have been taken out of context from chapters 1 & 2.

What Is XSLT?

XSLT (Extensible Stylesheet Language: Transformations) is a language that, according to the very first sentence in the specification (found at `http://www.w3.org/TR/xslt20/`), is primarily designed for transforming one XML document into another. However, XSLT is also capable of transforming XML to HTML and many other text-based formats, so a more general definition might be as follows:

XSLT is a language for transforming the structure and content of an XML document.

Why should you want to do that? In order to answer this question properly, we first need to remind ourselves why XML has proved such a success and generated so much excitement.

XML is a simple, standard way to interchange structured textual data between computer programs. Part of its success comes because it is also readable and writable by humans, using nothing more complicated than a text editor, but this doesn't alter the fact that it is primarily intended for communication between software systems. As such, XML satisfies two compelling requirements:

* *Separating data from presentation*: the need to separate information (such as a weather forecast) from details of the way it is to be presented on a particular device. The early motivation for this arose from the need to deliver information not only to the traditional PC-based Web browser (which itself comes in many flavors) but also to TV sets and handheld devices, not to mention the continuing need to produce print-on-paper. Today, for many information providers an even more important driver is the opportunity to syndicate content to other organizations that can republish it with their own look-and-feel.
* *Transmitting data between applications*: the need to transmit information (such as orders and invoices) from one organization to another without investing in one-off software integration projects. As electronic commerce gathers pace, the amount of data exchanged between enterprises increases daily, and this need becomes ever more urgent.

Of course, these two ways of using XML are not mutually exclusive. An invoice can be presented on the screen as well as being input to a financial application package, and weather forecasts can be summarized, indexed, and aggregated by the recipient instead of being displayed directly. Another of the key benefits of XML is that it unifies the worlds of documents and data, providing a single way of representing structure regardless of whether the information is intended for human or machine consumption. The main point is that, whether the XML data is ultimately used by people or by a software application, it will very rarely be used directly in the form it arrives: it first has to be transformed into something else.

In order to communicate with a human reader, this something else might be a document that can be displayed or printed: for example, an HTML file, a PDF file, or even audible sound. Converting XML to HTML for display is probably still the most common application of XSLT, and it is the one I will use in most of the examples in this book. Once you have the data in HTML format, it can be displayed on any browser.

In order to transfer data between different applications, we need to be able to transform information from the data model used by one application to the model used by another. To load the data into an application, the required format might be a comma-separated-values file, a SQL script, an HTTP message, or a sequence of calls on a particular programming interface. Alternatively, it might be another XML file using a different vocabulary from the original. As XML-based electronic commerce becomes widespread, the role of XSLT in data conversion between applications also becomes ever more important. Just because everyone is using XML does not mean the need for data conversion will disappear.

There will always be multiple standards in use. As I write there is a fierce debate between the protagonists of two different XML representations of office documents: the ODF specification from the Open Office community, and the OOXML specification from Microsoft and its friends. However this gets resolved, the prospects of a single XML format for all word processor documents are remote, so there will always be a need to transform between multiple formats.

Even within the domain of a single standard, there is a need to extract information from one kind of document and insert it into another. For example, a PC manufacturer who devises a solution to a customer problem will need to extract data from the problem reports and insert it into the documents issued to field engineers so that they can recognize and fix the problem when other customers hit it. The field engineers, of course, are probably working for a different company, not for the original manufacturer. So, linking up enterprises to do e-commerce will increasingly become a case of defining how to extract and combine data from one set of XML documents to generate another set of XML documents, and XSLT is the ideal tool for the job.

A Scenario: Transforming Music

As an indication of how far XML has now penetrated, Robin Cover's index of XML-based application standards at `http://xml.coverpages.org/xmlApplications.html` today runs to 594 entries. (The last one is entitled *Mind Reading Markup Language*, but as far as I can tell, all the other entries are serious.)

I'll follow just one of these 594 links, *XML and Music*, which takes us to `http://xml.coverpages.org/xmlMusic.html`. On this page we find a list of no less than 18 standards, proposals, or initiatives that use XML for marking up music.

This diversity is clearly unnecessary, and many of these initiatives are already dead or dying. Even the names of the standards are chaotic: there is a Music Markup Language, a MusicML, a MusicXML, and a MusiXML, all quite unrelated. There are at least three really serious contenders: the Music Encoding Initiative (MEI), the Standard Music Description Language (SMDL), and MusicXML The MEI derives its inspiration from the Text Encoding Initiative, and has a particular focus on the needs of music scholars (for example, the ability to capture features found in different manuscripts of the same score), while SMDL is related to the HyTime hypermedia standards and takes into account requirements such as the need to synchronize music with video or with a lighting script (it has not been widely implemented, but it has its enthusiasts). MusicXML, by contrast, is primarily focused on the needs of composers and publishers of sheet music.

Given the variety of requirements, it's unlikely that the number of standards in use will reduce any further. The different notations were invented with different purposes in mind: a markup language used by a publisher for printing sheet music has different requirements from the one designed to let you listen to the music from a browser.

In the first edition of this book, back in 2001, I introduced the idea of using XSLT to transform music as a theoretical possibility, something to make my readers think about the range of possibilities open for the language. By the time I published the second edition, PhD students were showing that it could actually be done. Today, MusicXML is a standard part of over 80 software applications including industry leaders such as Sibelius and Finale, and XSLT is routinely used to manipulate the output. The MEI website publishes XSLT stylesheets for converting between MEI and MusicXML in either direction.

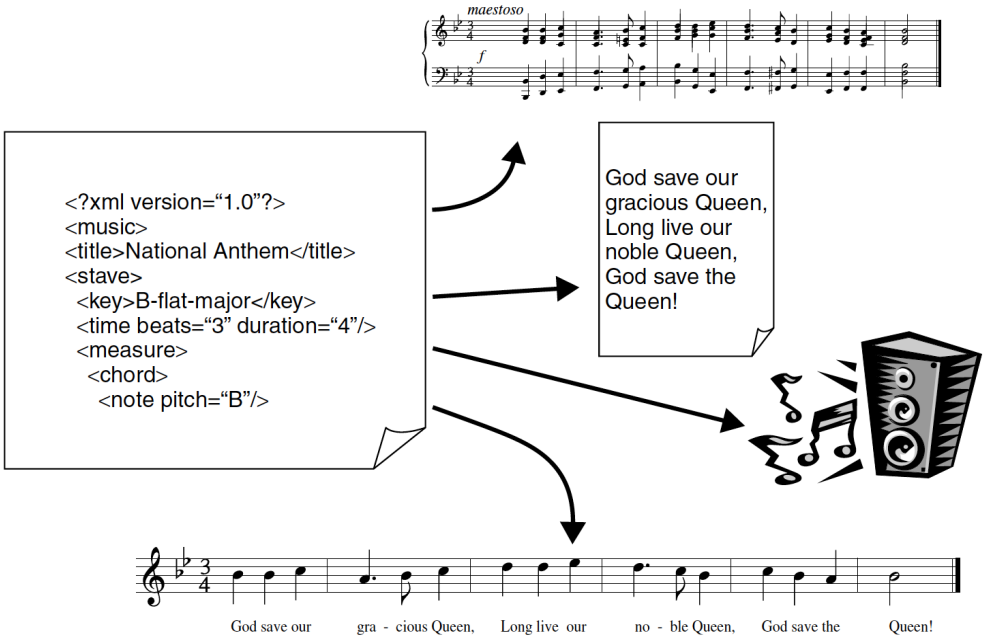


Figure 1-1

As it happens, MusicXML itself provides two ways of representing a score. In one the top-level subdivision in the XML hierarchy is by instrumental part or voice; in the other the top-level structure is the timeline of the music. XSLT stylesheets are provided to convert between the two formats.

Figure 1-1 shows some of the possibilities. You could use XSLT to:

* Convert music from one representation to another, for example from MEI to SMDL.
* Convert music from any of these representations into visual music notation, by generating the XML-based vector graphics format SVG.
* Play the music on a synthesizer, by generating a MIDI (Musical Instrument Digital Interface) file.
* Perform a musical transformation, such as transposing the music into a different key or extracting parts for different instruments or voices.
* Extract the lyrics, into HTML or into a text-only XML document.
* Capture music from non-XML formats and translate it to XML (XSLT 2.0 is especially useful here).

As you can see, XSLT is not just for converting XML documents to HTML.

XSLT and SQL: An Analogy

In a relational database, the data consists of a set of tables. By themselves, the tables are not of much use, the data might as well be stored in flat files in comma-separated values format. The power of a relational database doesn't come from its data structure; it comes from the language that processes the data, SQL. In the same way, XML on its own just defines a data structure. It's a bit richer than the tables of the relational model, but by itself it doesn't actually do anything very useful. It's when we get a high-level language expressly designed to manipulate the data structure that we start to find we've got something interesting on our hands, and for XML data the main language that does that is XSLT.

Superficially, SQL and XSLT are very different languages. But if you look below the surface, they actually have a lot in common. For starters, in order to process specific data, be it in a relational database or an XML document, the processing language must incorporate a declarative query syntax for selecting the data that needs to be processed. In SQL, that's the `SELECT` statement. In XSLT, the equivalent is the *XPath expression*.

The XPath expression language forms an essential part of XSLT, though it is actually defined in a separate W3C Recommendation (`http://www.w3.org/TR/xpath`) because it can also be used independently of XSLT (the relationship between XPath and XSLT is discussed further on page 21).

The XPath syntax is designed to retrieve nodes from an XML document, based on a path through the XML document or the context in which the node appears. It allows access to specific nodes, while preserving the

hierarchy and structure of the document. XSLT instructions are then used to manipulate the results of these queries, for example by rearranging selected nodes and constructing new nodes.

There are further similarities between XSLT and SQL:

* Both languages augment the basic query facilities with useful additions for performing arithmetic, string manipulation, and comparison operations.
* Both languages supplement the declarative query syntax with semiprocedural facilities for describing the processing to be carried out, and they also provide hooks to escape into conventional programming languages where the algorithms start to get too complex.
* Both languages have an important property called *closure*, which means that the output has the same data structure as the input. For SQL, this structure is tables, for XSLT it is trees—the tree representation of XML documents. The closure property is extremely valuable because it means operations performed using the language can be combined end-to-end to define bigger, more complex operations: you just take the output of one operation and make it the input of the next operation. In SQL you can do this by defining views or subqueries; in XSLT you can do it by passing your data through a series of stylesheets, or by capturing the tree produced by one transformation phase in a variable and using that variable as the input of another transformation phase. This last feature is new in XSLT 2.0, though most XSLT 1.0 processors offered a similar capability as a language extension.

In the real world, of course, XSLT and SQL have to coexist. There are many possible relationships, but typically, data is stored in relational databases and transmitted between systems in XML. The two languages don't fit together as comfortably as one would like, because the data models are so different. But XSLT transformations can play an important role in bridging the divide. All of the major relational database vendors have released extensions that allow XML data to be stored and manipulated directly within what is still nominally a relational database.

XSLT and XPath

Halfway through the development of XSLT 1.0, it was recognized that there was a significant overlap between the expression syntax in XSLT for selecting parts of a document and the XPointer language being developed for linking from one document to another. To avoid having two separate but overlapping expression languages, the two committees decided to join forces and define a single language, *XPath*, which would serve both purposes. XPath 1.0 was published on the same day as XSLT 1.0, November 16, 1999.

XPath acts as a sublanguage within an XSLT stylesheet. An XPath expression may be used for numerical calculations or string manipulations, or for testing Boolean conditions, but its most characteristic use (and the one that gives it its name) is to identify parts of the input document to be processed. For example, the following instruction outputs the average price of all the books in the input document:

```
<xsl:value-of select="avg(//book/@price)"/>
```

Here, the `<xsl:value-of>` element is an instruction defined in the XSLT standard, which causes a value to be written to the output document. The `select` attribute contains an XPath expression, which calculates the value to be written: specifically, the average value of the `price` attributes on all the `<book>` elements. (The `avg()` function too is new in XPath 2.0.)

Following its publication, the XPath specification increasingly took on a life of its own, separate from XSLT. Several DOM implementations (including Microsoft's) allowed you to select nodes within a DOM tree structure, using a method such as `selectNodes(XPath)`, and this feature is now included in the current version of the standard, DOM3. Subsets of XPath are used within the XML Schema language and in XForms for defining validation conditions, and bindings of XPath to other languages such as Perl are multiplying. Perhaps most important of all, the designers of XQuery decided to make their language a pure superset of XPath. The language has also proved interesting to academics, and a number of papers have been published analyzing its semantics, which provides the basis for optimized implementations.

XSLT and XML Namespaces

XSLT is designed on the basis that *XML namespaces* are an essential part of the XML standard. So when the XSLT standard refers to an XML document, it always means an XML document that conforms to the XML Namespaces specification, which can be found at `http://www.w3.org/TR/REC-xml-names`.

Namespaces play an important role in XSLT. Their purpose is to allow you to mix tags from two different vocabularies in the same XML document. We've already seen how a stylesheet can mix elements from the target vocabulary (for example HTML or XSL-FO) with elements that act as XSLT instructions. Here's a quick reminder of how namespaces work:

* Namespaces are identified by a Uniform Resource Identifier (URI). This can take a number of forms. One form is the familiar URL, for example `http://www.wrox.com/namespace`. Another form, not fully standardized but being used in some XML vocabularies, is a URN, for example `urn:biztalk-org:biztalk:biztalk_1`. The detailed form of the URI doesn't matter, but it is a good idea to choose one that will be unique. One good way of achieving this is to use the domain name of your own website. But don't let this confuse you into thinking that there must be something on the website for the URI to point to. The namespace URI is simply a string that you have chosen to be different from other people's namespace URIs; it doesn't need to point to anything.
* The latest version, XML Namespaces 1.1, allows you to use an International Resource Identifier (IRI) rather than a URI. The main difference is that this permits characters from any alphabet (for example, Chinese); it is no longer confined to ASCII. In practice, most XML parsers have always allowed you to use any characters you like in a namespace URI.
* Since namespace URIs are often rather long and use special characters such as «/», they are not used in full as part of the element and attribute names. Instead, each namespace used in a document can be given a short nickname, and this nickname is used as a prefix of the element and attribute names. It doesn't matter what prefix you choose, because the real name of the element or attribute is determined only by its namespace URI and its local name (the part of the name after the prefix). For example, all my examples use the prefix `xsl` to refer to the namespace URI `http://www.w3.org/1999/XSL/Transform`, but you could equally well use the prefix `xslt`, so long as you use it consistently.
* For element names, you can also declare a default namespace URI, which is to be associated with unprefixed element names. The default namespace URI, however, does not apply to unprefixed attribute names.

A namespace prefix is declared using a special pseudo-attribute within any element start tag, with the form:

```
xmlns:prefix = "namespace-URI"
```

This declares a namespace prefix, which can be used for the name of that element, for its attributes, and for any element or attribute name contained in that element. The default namespace, which is used for elements having no prefix (but not for attributes), is similarly declared using a pseudo-attribute:

```
xmlns = "namespace-URI"
```

XML Namespaces 1.1 became a Recommendation on February 4, 2004, and the XSLT 2.0 specification makes provision for XSLT processors to work with this version, though it isn't required. Apart from the largely cosmetic change from URIs to IRIs mentioned earlier, the main innovation is the ability to undeclare a namespace, using syntax of the form «`xmlns:prefix=""`». This is particularly intended for applications like SOAP messaging, where an XML payload document is wrapped in an XML envelope for transmission. Without namespace undeclarations, there is a tendency for namespaces used in the SOAP envelope to stick to the payload XML when this is removed from the envelope, which can cause problems—for example, it can invalidate a digital signature attached to the document.

XSLT and CSS

Why are there two stylesheet languages, XSL (that is, XSLT plus XSL Formatting Objects) as well as Cascading Style Sheets (CSS and CSS2)?

It's only fair to say that in an ideal world there would be a single language in this role, and that the reason there are two is that no one was able to invent something that achieved the simplicity and economy of CSS for doing simple things, combined with the power of XSL for doing more complex things.

CSS is mainly used for rendering HTML, but it can also be used for rendering XML directly, by defining the display characteristics of each XML element. However, it has serious limitations. It cannot reorder the elements in the source document, it cannot add text or images, it cannot decide which elements should be displayed and which omitted, neither can it calculate totals or averages or sequence numbers. In other words, it can only be used when the structure of the source document is already very close to the final display form.

Having said this, CSS is simple to write, and it is very economical in machine resources. It doesn't reorder the document, and so it doesn't need to build a tree representation of the document in memory, and it can start displaying the document as soon as the first text is received over the network. Perhaps, most important of all, CSS is very simple for HTML authors to write, without any programming skills. In comparison, XSLT is far more powerful, but it also consumes a lot more memory and processor power, as well as training budget.

It's often appropriate to use both tools together. Use XSLT to create a representation of the document that is close to its final form, in that it contains the right text in the right order, and then use CSS to add the finishing touches, by selecting font sizes, colors, and so on. Typically, you would do the XSLT processing on the server and the CSS processing on the client (in the browser); so, another advantage of this approach is that you reduce the amount of data sent down the line, which should improve response time for your users and postpone the next expensive bandwidth increase.

A Simplified Overview

The core task of an XSLT processor is to apply a stylesheet to a source document and produce a result document. This is shown in Figure 2-1.

As a first approximation we can think of the source document, the stylesheet, and the result document as each being an XML document. XSLT performs a *transformation* process because the *output* (the result document) is the same kind of object as the *input* (the source document). This has immediate benefits: for example, it is possible to do a complex transformation as a series of simple transformations, and it is possible to do transformations in either direction using the same technology.

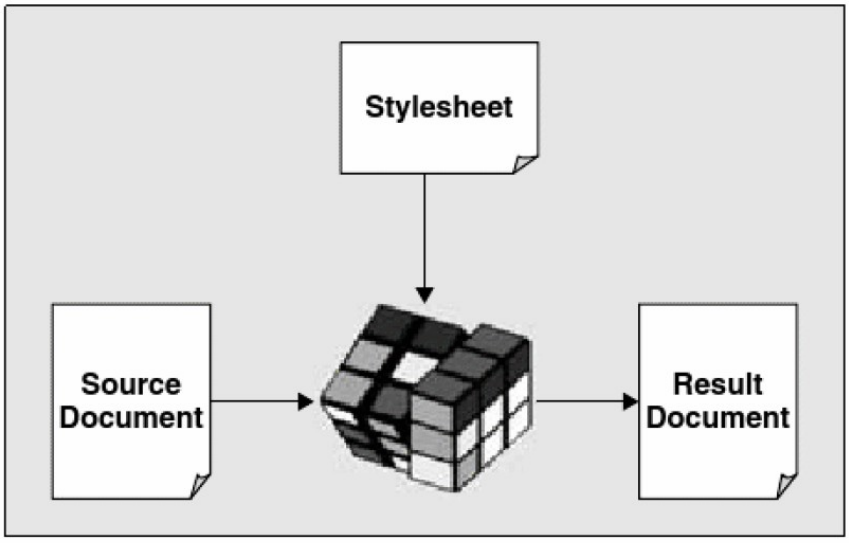


Figure 2-1

The choice of Rubik's cube to illustrate the transformation process is not entirely whimsical. The mathematics of Rubik's cube relies on group theory, which is where the notion of closure comes from: every operation transforms one instance of a type into another instance of the same type. We're transforming XML documents rather than cubes, but the principle is the same.

The name *stylesheet* has stuck for the document that defines the transformation, despite the fact that XSLT is often used for tasks that have nothing to do with styling. The name reflects the reality that a very common kind of transformation performed using XSLT is to define a display style for the information in the source document, so that the result document contains information from the source document augmented with information controlling the way it is displayed on some output device.

Trees, Not Documents

In practice, we don't always want the input or output to be XML in its textual form. If we want to produce HTML output (a very common requirement), we want to produce it directly, rather than having an XML document as an intermediate form. When the Firefox browser displays the result of an XSLT transformation, it doesn't serialize the result to HTML and then parse the textual HTML; rather, it works directly from the result tree as a data structure in memory. Similarly, we might want to take input from a database or (say) an LDAP directory, or an EDI message, or a data file using comma-separated-values syntax. We don't want to spend a lot of time converting these into serial XML documents if we can avoid it, nor do we want another raft of converters to install.

Instead, XSLT defines its operations in terms of a data model (called XDM) in which an XML document is represented as a *tree*. The tree is an abstract data type. There is no defined application programming interface (API) and no defined data representation, only a conceptual model that defines the objects in the tree, their properties, and their relationships. The XDM tree is similar in concept to the W3C DOM, except that the Document Object Model (DOM) does have a defined API. Some implementors do indeed use the DOM as their internal tree structure. Others use a data structure that corresponds more closely to the XDM specification, while some use optimized internal data structures that are only distantly related to this model. It's a conceptual model we are describing, not something that necessarily exists in an implementation.

The data model for XSLT trees is shared with the XPath and XQuery specifications, which ensures that data can be freely exchanged between these three languages (it also means you can take your pick as to what the "X" in "XDM" stands for). With XSLT and XPath, this is of course essential, because XSLT always retrieves data from a source document by executing XPath expressions. There is a description of this data model later in this chapter, and full details are in Chapter 4.

Taking the inputs and output of the XSLT processors as trees produces a new diagram (see Figure 2-2). The formal conformance rules say that an XSLT processor must be able to read a stylesheet and use it to transform a source tree into a result tree. This is the part of the system shown in the oval box. There's no official requirement to handle the parts of the process shown outside the box, namely the creation of a source tree from a source XML document (known as *parsing*), or the creation of a result XML document from the result tree (called *serialization*). In practice, though, most real products are likely to handle these parts as well.

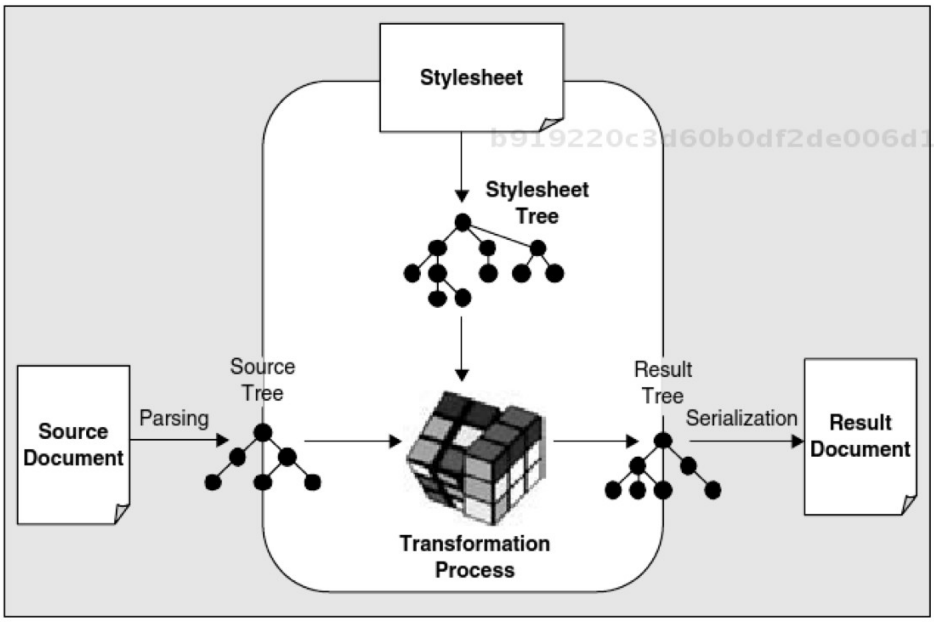


Figure 2-2

Different Output Formats

Although the final process of converting the result tree to an output document is outside the conformance rules of the XSLT standard, this doesn't mean that XSLT has nothing to say on the subject.

The main control over this process is the `<xsl:output>` element. This element defines four output formats or methods, namely `xml`, `html`, `xhtml`, and `text`. In each case a result tree is written to a single output file.

* With the `xml` output method, the output file is an XML document . We'll see later that it need not be a complete XML document; it can also be an XML fragment. The `<xsl:output>` element allows the stylesheet writer some control over the way in which the XML is written, for example, the character encoding used and the use of `CDATA` sections.
* With the `html` output method, the output file is an HTML document, typically HTML 4.0, though products may support other versions if they wish. With HTML output, the XSLT processor recognizes many of the conventions of HTML and structures the output accordingly. For example, it recognizes elements such as `<hr>` that have a start tag and no end tag, as well as the special rules for escape characters within a `<script>` element. It may also (if it chooses) generate references to built-in entities such as «`&eacute;`».

  Selecting `html` as the output method doesn't in any way automate the process of creating valid HTML, nor does it cause the processor to check that the output is valid HTML (with one exception—it will report the presence of characters that aren't allowed in HTML). All it does is to tell the serializer to use HTML conventions when turning the nodes in the tree back into markup.
* The `xhtml` output method, as one might expect, is a compromise between the `xml` and `html` output methods. Generally speaking, it follows the rules of the `xml` output method, but sticks to the conventions described in the XHTML specification that are designed to make the output display properly in browsers that were written to handle HTML. Such conventions include, for example, outputting an empty `<br>` element as `<br />` (with a space before the «/»), and outputting an empty `<p>` element as `<p></p>`.
* The `text` output method is designed to allow output in any other text-based format. For example, the output might be a comma-separated-values file, a document in Microsoft's Rich Text Format (RTF), or in Adobe's Portable Document Format (PDF); or, it might be an electronic data interchange message, or a script in SQL or JavaScript. It's entirely up to you.

If the `<xsl:output>` element is omitted, the processor makes an intelligent guess, choosing HTML if the output starts with an `<html>` element in the null namespace, XHTML if it starts with an `<html>` element in the XHTML namespace, and XML otherwise.

Implementations may include output methods other than these four, but this is outside the scope of the standard. One mechanism provided by several products is to feed the result tree to a user-supplied document handler. In the case of Java products, this will generally be written to conform to the `ContentHandler` interface defined as part of the SAX2 API specification (which is part of the core class library in Java). Most implementations also provide mechanisms to capture the result as a DOM tree. Remember that if you use the result tree directly in this way, the XSLT processor will not serialize the tree, and therefore nothing you say in the `<xsl:output>` declaration will have any effect.

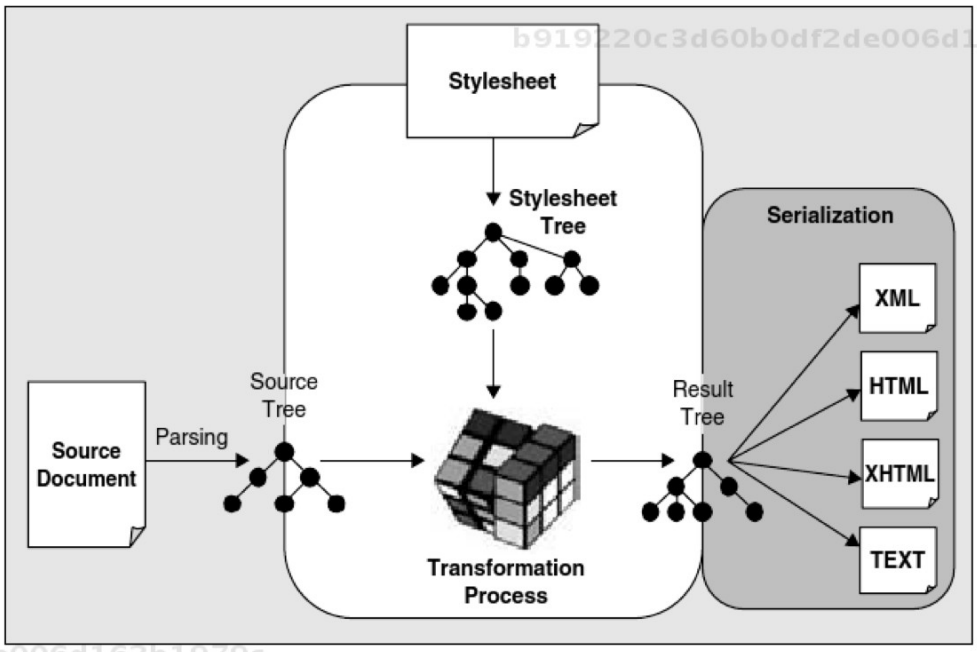


Figure 2-3

So, while the bulk of the XSLT Recommendation describes the transformation process from a source tree to a result tree, there is one section that describes another process, serialization. Because the serialization of a document tree as a file is a process that is relevant not only to XSLT but also to XQuery and potentially other applications in the future, the detail is no longer in the XSLT 2.0 Recommendation, but forms a W3C specification in its own right (see `http://www.w3.org/TR/xslt-xquery-serialization/`). For similar reasons, it has a chapter of its own in this book (Chapter 15). The name *serialization* is used because it turns a tree structure into a stream of characters or bytes, but it mustn't be confused with serialization in distributed object systems such as Component Object Model (COM) or Java, which produces a serial file representation of a COM or Java object. XSLT processors can implement this at their discretion, and it fits into our diagram as shown in Figure 2-3.

Nodes in the Tree Model

- An XDM tree is made up of nodes. There are seven kinds of node. The different kinds of node correspond fairly directly to the components of the source XML document:

Node Kind	Description
Document node	The document node is a singular node; there is one for each document. Do not confuse the document node with the document element, which in a well-formed document is the outermost element that contains all others. A document node never has a parent, so it is always the root of a tree.
Element node	An element is a part of a document bounded by start and end tags, or represented by a single empty-element tag such as `<TAG/>`. Try to avoid referring to elements as tags: elements generally have two tags, a start tag and an end tag.
Text node	A text node is a sequence of consecutive characters in a `PCDATA` part of an element. Text nodes are always made as big as possible: there will never be two adjacent text nodes in the tree, because they will always be merged into one. (This is the theory. Some implementations don't always follow this rule)
Attribute node	An attribute node includes the name and value of an attribute written within an element start tag (or empty element tag). An attribute that was not present in the tag, but which has a default value defined in the DTD or Schema, is also represented as an attribute node on each separate element instance. A namespace declaration (an attribute whose name is «`xmlns`» or whose name begins with «`xmlns:`») is, however, *not* represented by an attribute node in the tree.
Comment node	A comment node represents a comment written in the XML source document between the delimiters «`<!--`» and «`-->`»
Processing instruction node	A processing instruction node represents a processing instruction written in the XML source document between the delimiters «`<?`» and «`?>`». The `PITarget` from the XML source is taken as the node's name and the rest of the content as its value. Note that the XML declaration `<?xml version="1.0"?>` is not a processing instruction, even though it looks like one, and it is not represented by a node in the tree.
Namespace node	A namespace node represents a namespace declaration, except that it is copied to each element that it applies to. So each element node has one namespace node for every namespace declaration that is in scope for the element. The namespace nodes belonging to one element are distinct from those belonging to another element, even when they are derived from the same namespace declaration in the source document.

The String Value of a Node

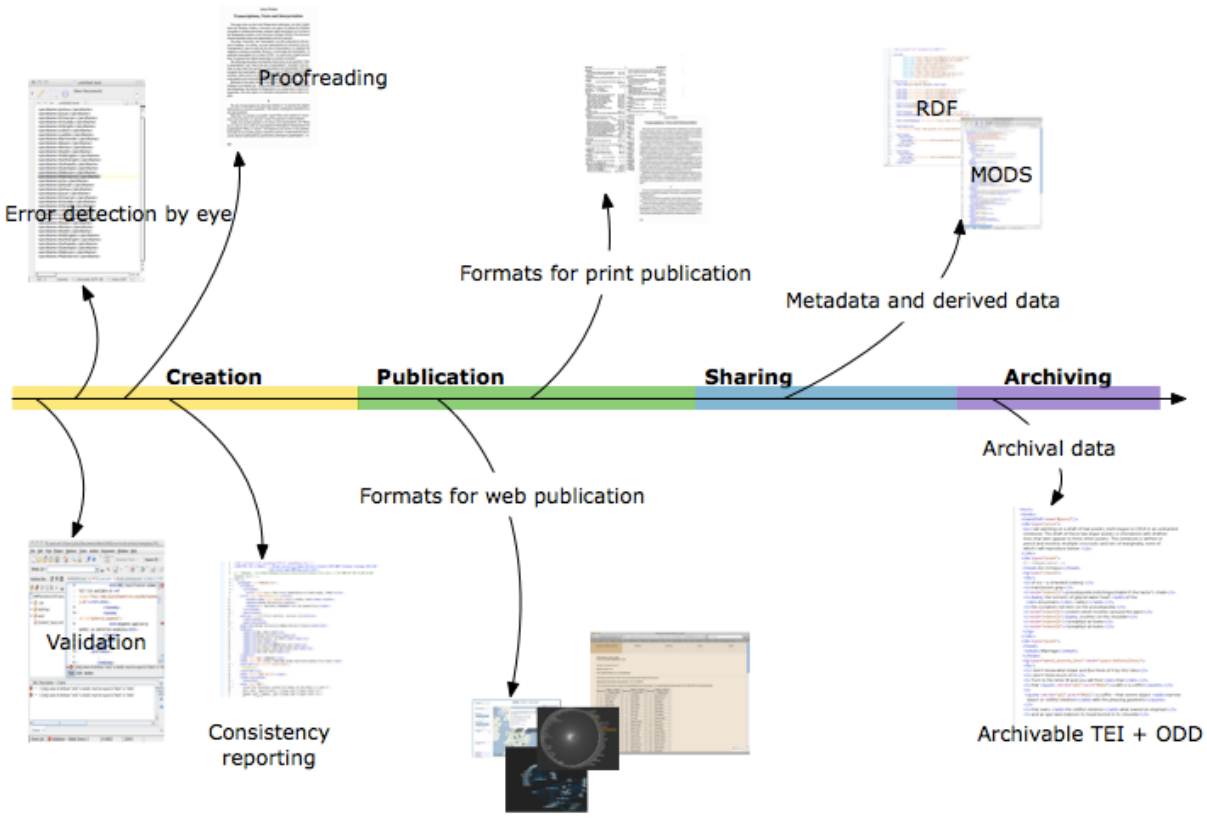
Every node has a string value, which is a sequence of Unicode characters. The string value depends on the kind of node, as shown in the table below:

Node Kind	String Value
Text	The text as it appears in the source XML document, except that the XML parser will have replaced every end-of-line sequence (for example, CRLF as used on Windows platforms) by a single newline (`x0A`) character.
Comment	The text of the comment, minus the delimiters.
Processing instruction	The `data` part of the source processing instruction, not including the whitespace that separates it from the `PITarget`. For example, given the processing instruction `<?ignore this?>`, the string value is `«this»`.
Attribute	The string value is the value of the attribute as written, modified by any whitespace normalization done by the XML parser and schema processor. The detailed rules for whitespace normalization of attributes depend on the attribute type.
Document or element	The concatenation of the string values of all the element and text children of this node. Or, to look at it another way: the concatenation of all the `PCDATA` contained in the element (or for the document node, the entire document) after stripping out all markup. (This again differs from the DOM, where the `nodeValue` property in these cases is null.)
Namespace	By convention, the URI of the namespace being declared.

XSLT: where does it fit in?

Julia Flanders, Syd Bauman

"XML workflow"



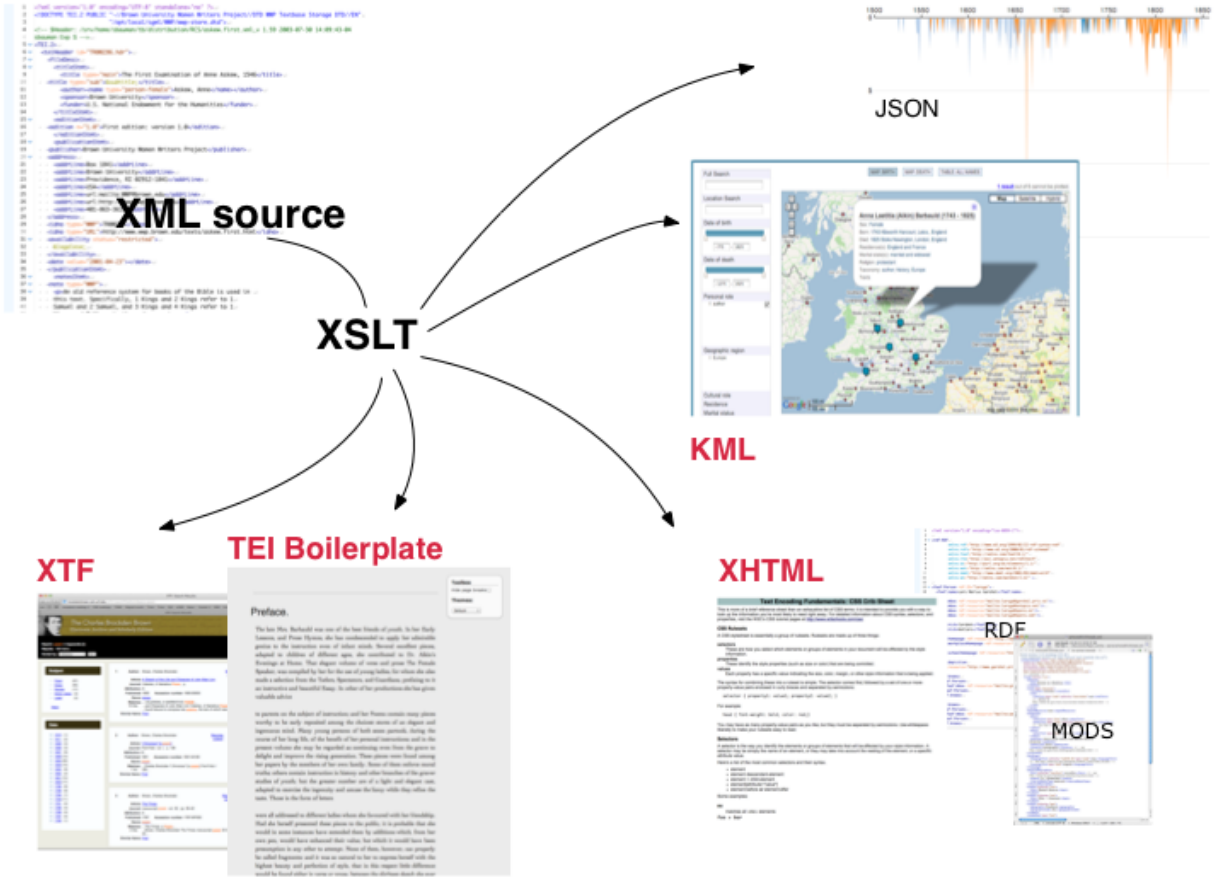
No one needs to convince us of the importance of the overall topic here: "transforming and publishing TEI". It's why we create TEI data. However, we may need to do some preliminary clarification and scoping to get a full sense of what we mean, of what the possibilities are and what kinds of "publishing and transforming" they entail.

One way to orient ourselves in the landscape of "publishing and transforming" is to think about how we use our own data. If we think of the life cycle of a TEI project, there are numerous places along the timeline where we want to express different views of our data, for internal or external viewing:

- proofreading and error catching

- formats that extract specific structures to let us catch inconsistencies in the encoding
- web publication
- print publication
- formats for contribution to collaborative projects (where we might want to simplify or alter our markup to match the target encoding of those projects)
- metadata formats (e.g. to expose to metadata harvesters)
- archival formats for committing to a repository

Single-source publishing and XSLT



Another way to approach the topic of this workshop is to think about transformation and publishing as a variety of informational avenues that radiate out from our TEI data. Even though it's probably a familiar concept to many of you, it's worth noting an important assumption that underlies much of our work with XML: we're creating a single XML source from which we are going to generate many different kinds of output.

This is important because the XML source is an expensive and valuable information object: it represents a careful modeling of our research materials, we've put a lot of work into it (transcription, encoding, proofreading, correction, annotation, other kinds of enhancement) and we want to exploit it in many different ways, automatically, not by hand.

When we generate these different varieties of output, we are often losing information: erasing distinctions that are present in the source (but unnecessary in the output), or moving from a representationally rich language (like TEI) to a representationally impoverished language (like HTML)

But since these output formats are generated automatically, rather than by hand, this information loss doesn't matter: the source retains its informational richness: it represents the full set of possibilities from which any specific option can be generated.

Some examples

Charles Brockden Brown Archive

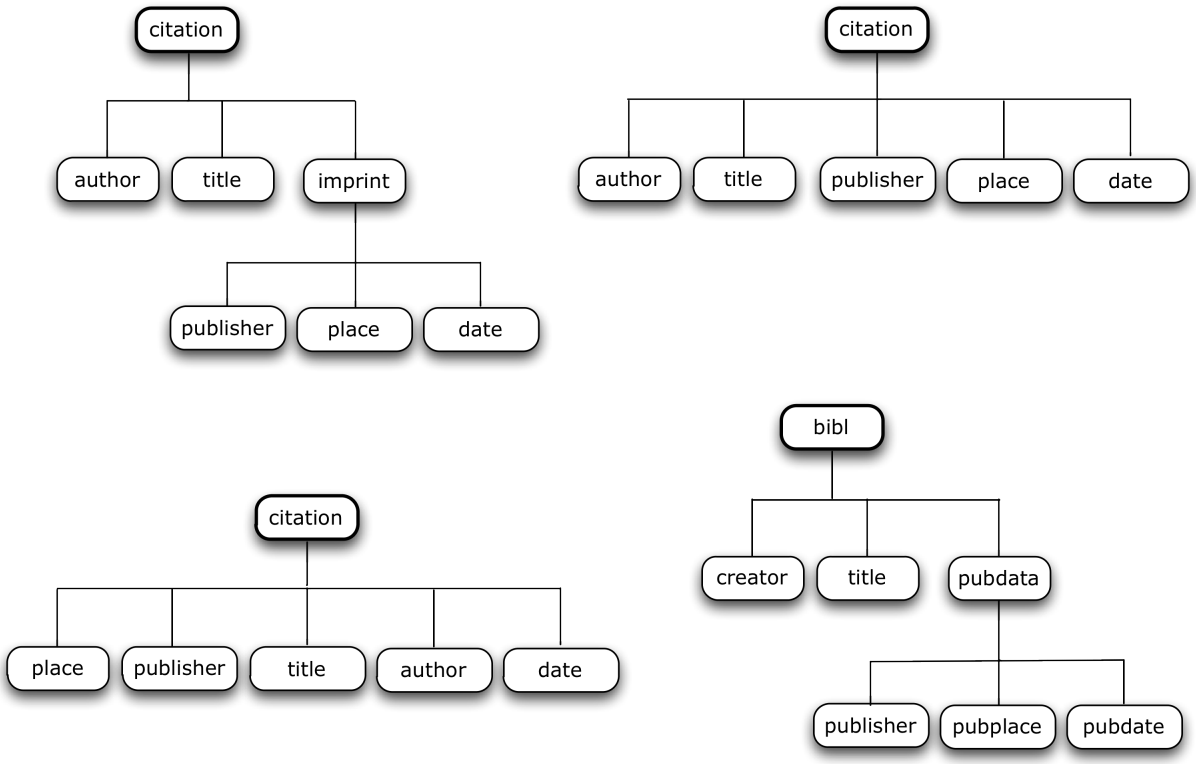
Mark Twain Project

Paul the Simple, XML source

A few examples:

- Charles Brockden Brown
- Mark Twain Project
- Paul the Simple

Transformation as a power tool



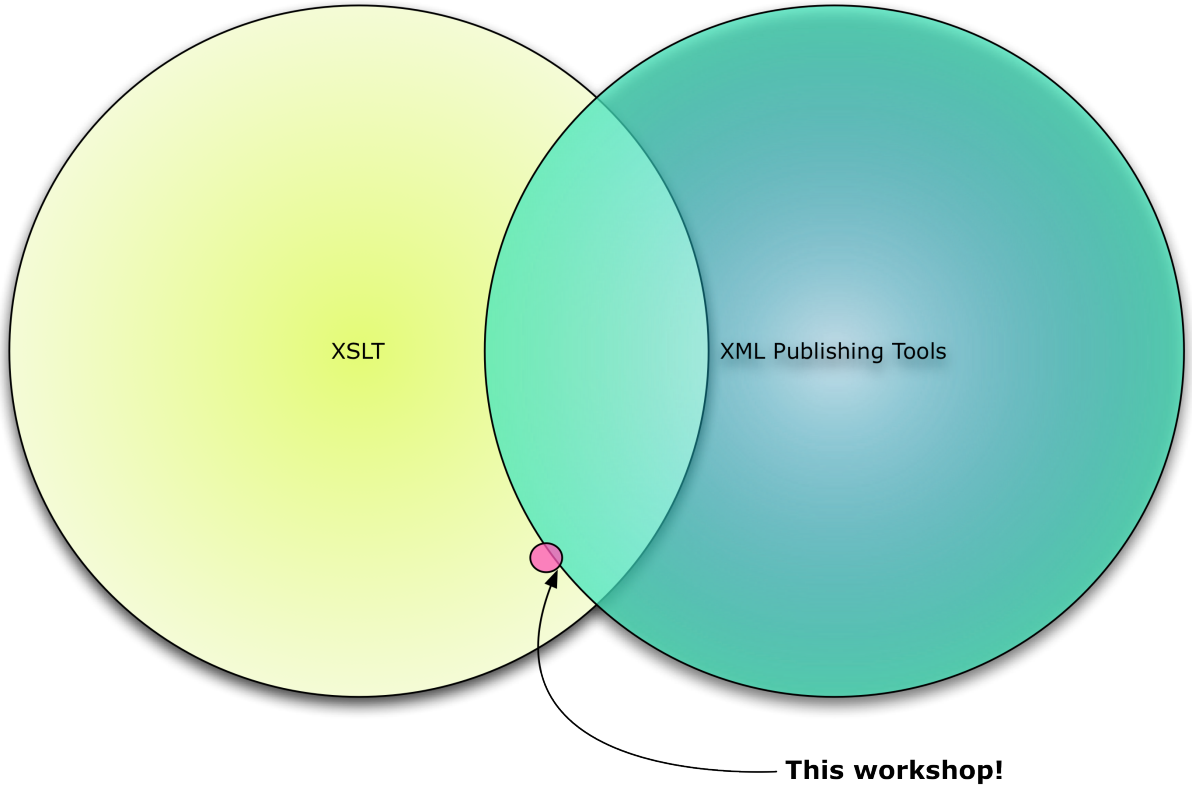
A third important aspect of our topic is the idea of data as a mutable, protean substance: as a kind of plastic informational model that we can reshape and manipulate as needed.

In the example here, all four of these examples represent pretty much the same pieces of data—any one of them could be generated from any of the others. And yet these differences might matter in the context of some particular tool or standard way of doing things.

The point is that our data is almost never trapped in its current format: when we understand it as transformable, we gain power over it and we can use it more flexibly. If a collaborator

needs some information extracted from our data, or if they put their fields in a slightly different order, or whatever, it's not a problem.

Scope and ambition



The chief tool for doing all of these kinds of work is a programming language called XSLT, the Extensible Stylesheet Language for Transformations:

- it can be used on its own to generate different kinds of transformed and manipulated data (such as HTML, KML, JSON, other XML formats)
- and it also is built into many (most? all?) of the XML publication systems that we use, such as XTF, as the way that they take XML data and manipulate it as part of their publication activities

Either way, what it does is give us a way of manipulating our XML data: to extract pieces of it, reshape them, change their format, generally do whatever we want to do with them.

Let's talk for a moment about what we're going to cover in this seminar (and what we're not going to cover).

This seminar is aimed at people who have TEI data and not much else: we aren't assuming familiarity with programming, or with XML publishing tools

Our goal is to help you learn about what's involved in using your TEI data: in publishing it, in manipulating and transforming it into other formats, exploiting its informational potential; we'd like you to come away, first of all, with a sense of what is possible.

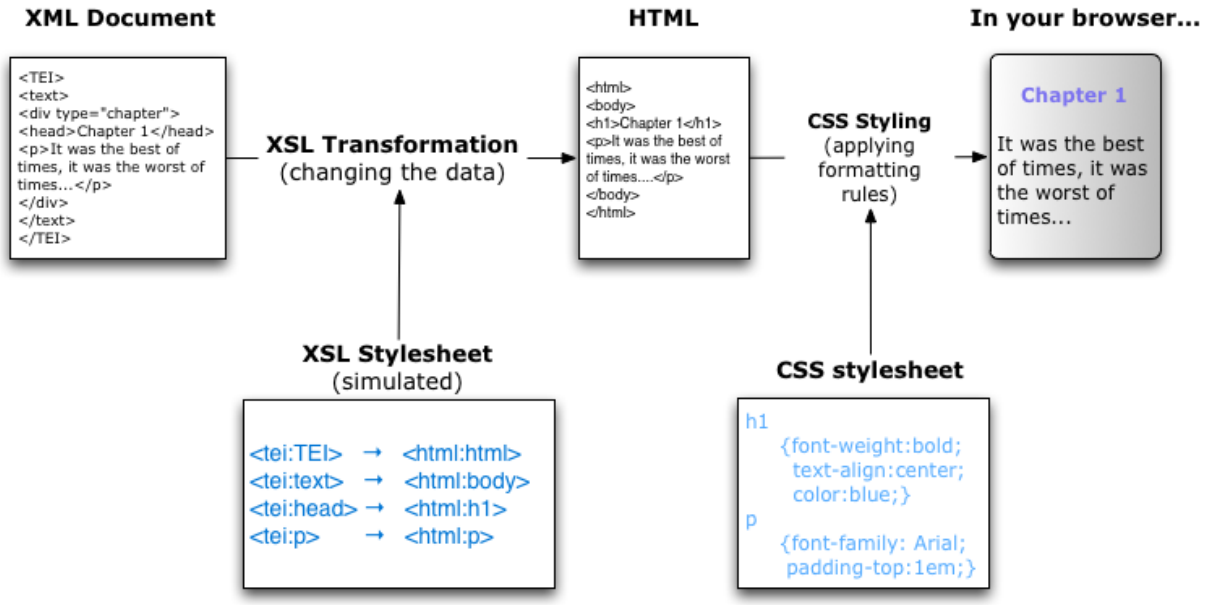
How about in concrete, practical terms? What are we actually going to cover? XSLT is hugely powerful--it is a full-fledged programming language--but as a result it's a big topic:

- hence we are not aiming here to teach you XSLT in any kind of comprehensive way
- what we are aiming to do is give you an understanding of how XSLT works, both on its own and in the context of XML publishing systems
- on the first two days, we are going to look at a lot of examples, and we're going to experiment with a lot of things that XSLT can do, including generating HTML and KML
- on the third day, we are going to install and set up two fairly simple XML publishing tools: XTF and TEI Boilerplate
- so at a minimum, by the end of the workshop you will be able to take your TEI data and publish it on the web in some basic ways.

By the end of the workshop, you should also have a good sense of whether XSLT is something you want to know more about and learn in a more systematic way, and if it is, we encourage you to take a more intensive XSLT workshop: Syd teaches one at DHSI, and Syd and David teach one at Brown every so often. This workshop is a good starting point for either of those workshops.

Simple Publication with XSLT

Extensible Stylesheet Language transformations allow you to transform XML documents into other formats



The Extensible Stylesheet Language allows you to transform XML documents into other XML formats

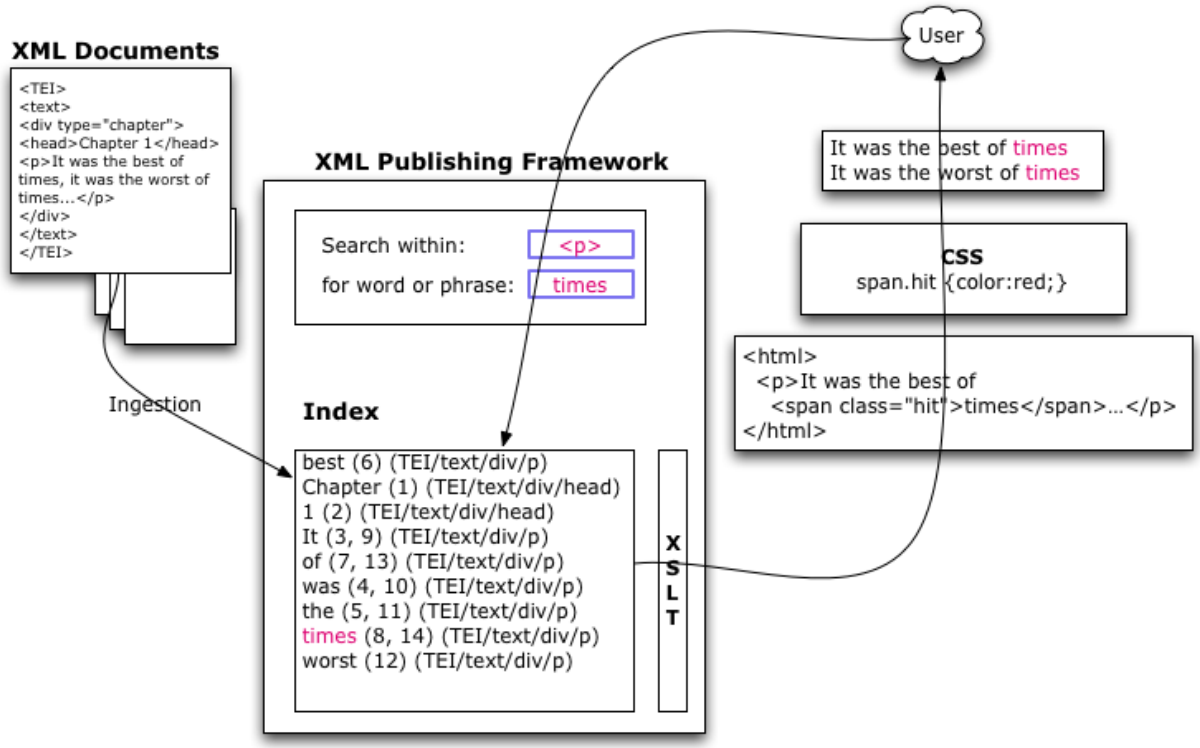
Essentially XSLT allows you to map a given XML element onto another XML element: saying "take in the following construct, and put out this other construct"

It could be a construct in the same language, or in a different language such as XHTML, as in the example here

XML Databases and Publication Frameworks

Tools designed to manage large groups of XML files, with more advanced functionality:

- fast, efficient searching
- transformations involving groups of files
- eXist, DBXML, Xindice, XTF, MarkLogic



The XML database and publication framework universe

These kinds of tools are designed to manage large groups of XML files, and to provide certain kinds of advanced functionality:

- fast, efficient searching
- transformations involving groups of files: not just transforming each file separately, but doing transformations that involve taking parts of different files and creating new results files: for instance, a sorted list of the first lines from all the poems in a collection.

How do databases fit into a larger XML publication framework? What do they do?

- they create and store indexed information: that is, information from the source XML files that has been preprocessed to make it more accessible and easier to manipulate. For instance, they might store tables of all the document metadata (author, title, genre, date, etc.) so that it can be searched and sorted more quickly

- they contain a representation of the document's structure in a format that makes it easier to process, so that certain kinds of navigation are easier

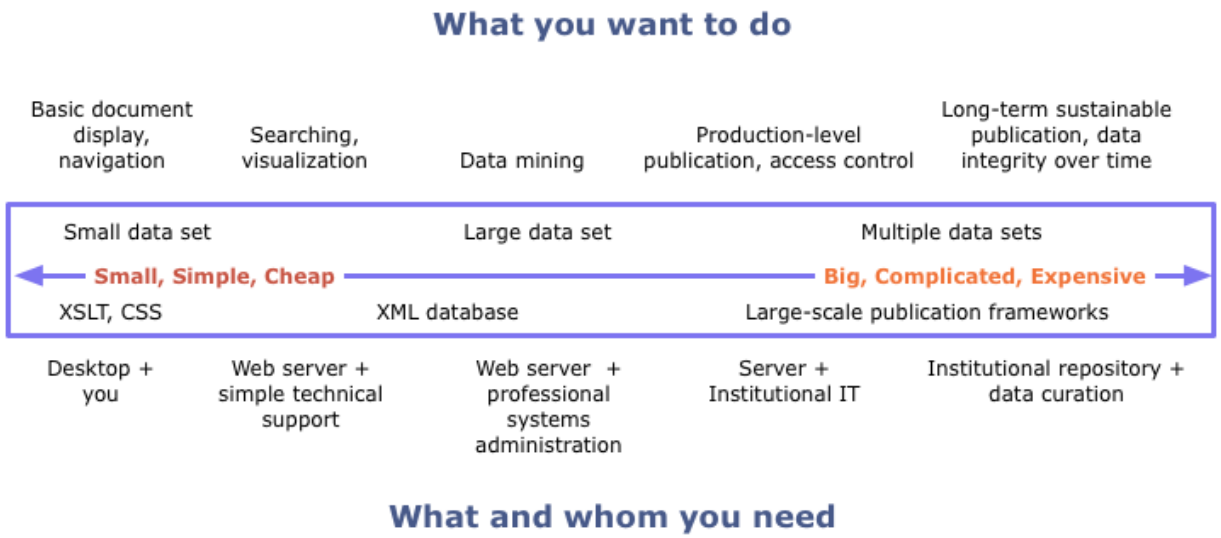
Within the XML publication framework, the database sits and waits for queries to come in.

- when it receives a query, it performs the necessary searching and returns a result (in the form of an XML fragment, or a node set, or some proprietary structure)
- the result can then be transformed (e.g. into HTML for delivery to a browser, or into some other XML format for other processing) using XSLT

XML databases exist as separate modules that can be used as the basis for XML publishing systems, for instance:

- eXist
- DBXML
- Xindice (Apache)

The Bigger Picture



The tools you need, and the people you need, can be imagined as a rough continuum of increasing scale, complexity, difficulty, and cost:

- at the simplest level, there are things you can do (or learn to do) by yourself, with very little in the way of equipment or software: tools like XSLT and CSS will go a long way towards producing simple, effective interfaces for browsing and reading small sets of documents
- at a slightly more complex level, as the number of documents increases and as you want to do more ambitious things with them (such as visualizations, complex searching), you need software tools that are a little more challenging to manage: perfectly within the capabilities of a humanist, but requiring more time: not something you can do on the side of another job; this becomes someone's major job responsibility
- Going a bit further, we get to things that require XML publication frameworks that require a professional systems administrator, someone who really understands the installation and configuration of things like web servers, XML databases, etc. These are the kinds of tools we need to build things like data mining or text/topic analysis into our publications, and also if we want to publish larger collections of documents that require more server power/speed
- For production-level publication, where you may be actually charging money for access (and hence need to do things like authentication) and hence may have higher standards of performance and reliability, you need to start engaging with your institutional IT organization to make sure that things like backups, server maintenance, etc. are being handled at the appropriate level of professionalism; this is also the level of scale at which we start to be able to really work effectively with multiple large data sets: for instance, multiple projects of substantial size
- Finally, if we want to be able to ensure the long-term sustainability of projects, we need to engage with systems like institutional repositories and the data curators who can help us ensure that data will be maintained, migrated, etc. after the project itself is no longer funded.

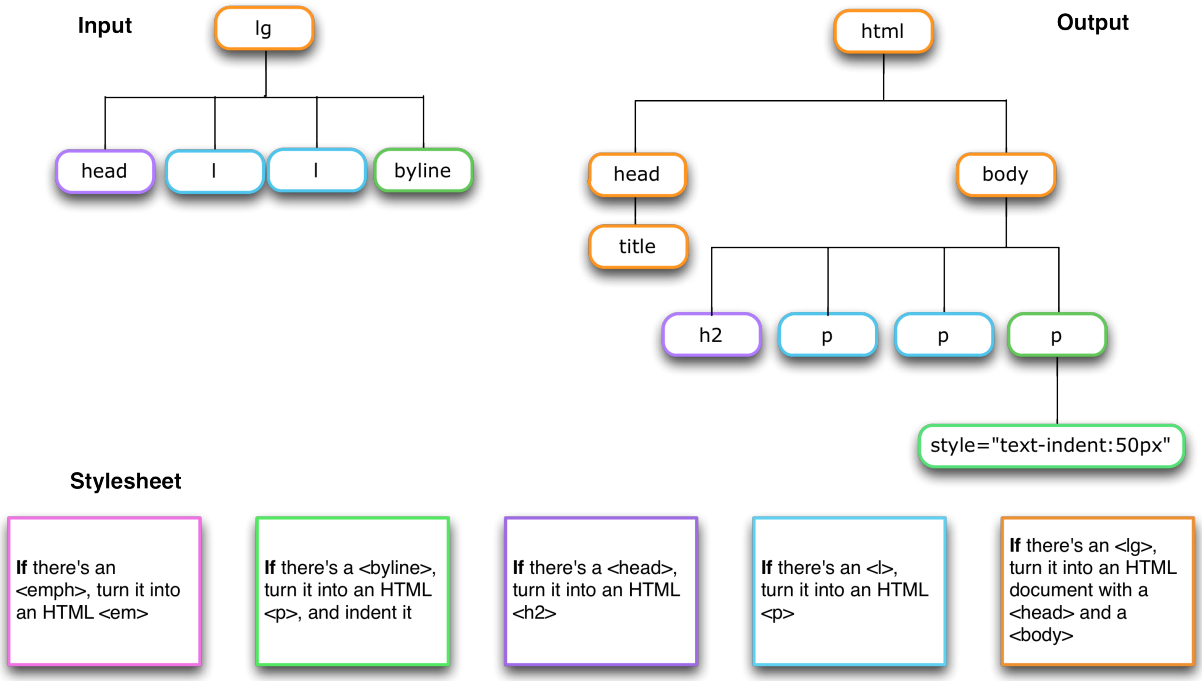
So considering where the three examples we looked at earlier might fit in:

- Paul the Simple: a single scholar, acting alone
- Charles Brockden Brown: a small amount of professional systems adminstration
- Mark Twain Project: a much larger staff, embedded in the CDL (which is where XTF was built)

A Gentle Introduction to XSLT

Julia Flanders, Syd Bauman

What does an XSLT stylesheet do?



Note!! at this stage we are talking very imprecisely and impressionistically about XSLT, for the sake of making it clearer and learning things one at a time. We'll add precision later.

At its simplest: it takes an input file (an XML tree structure) and transforms it: changes its form somehow.

It does this by taking the input tree, piece by piece, and applying transformation rules to each piece: these rules say what to do with each piece of input

We call these transformation rules "templates", and an XSLT stylesheet is basically a set of templates, each one with a condition of operation (when do I happen?) and its own specific task

These tasks will take place if the conditions of operation are met, and not if not.

A very simple example

The Input Document

```
<?xml version="1.0" encoding="UTF-8"?>
<paragraph>Hello, world!</paragraph>
```

The Stylesheet

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

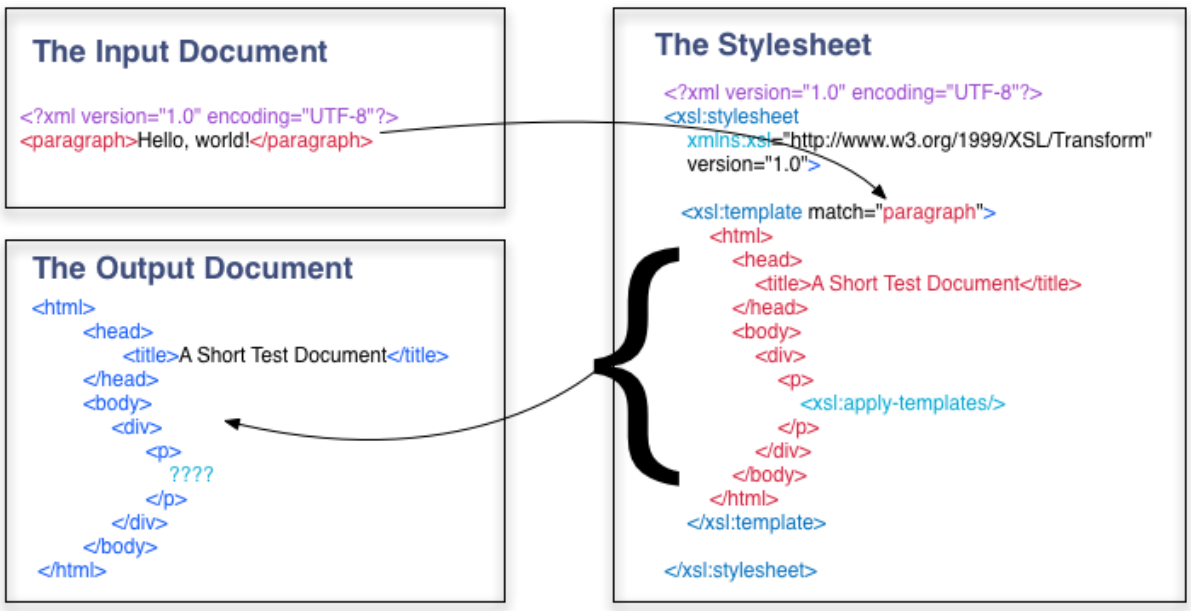
  <xsl:template match="paragraph">
     <html>
        <head>
           <title>A Short Test Document</title>
        </head>
        <body>
           <div>
              <p>
                 <xsl:apply-templates/>
              </p>
           </div>
        </body>
     </html>
  </xsl:template>

</xsl:stylesheet>
```

Here's a very, very simple XML document, and a very very simple stylesheet. Looking at this stylesheet, what do you think it's going to do?

OK, so let's look at how it does this, breaking it down into steps.

Build an output tree ...

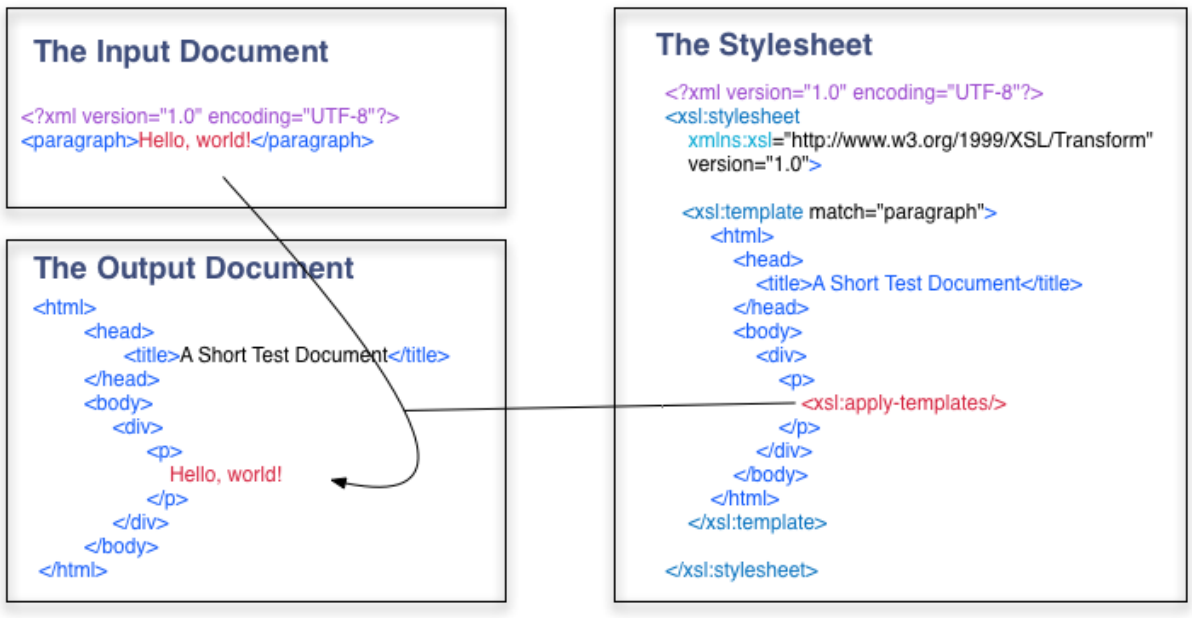


The first thing that happens is that we build an output tree--a structural skeleton for the file we are creating. The output tree is made up of pieces of code that are embedded in the XSLT stylesheet

So if we look at this diagram, we can see that the stylesheet contains a whole little HTML document;

- when we run the stylesheet, we say in effect "I have a paragraph here, what do I do when I have a paragraph?"
- the stylesheet answers "aha! when I have a paragraph, I create an output tree like this..."
- Notice that our output tree can include some pre-defined content (like boilerplate) but at this point there's still no content from the input tree present in the output.

... then fill it with data



Go get that content, apply templates to it, and put it in the output tree!!

Once we have the output tree, we next have to populate it with content from the input. For this step, there's a special part of the stylesheet that serves as a kind of conduit for content to come from the input and get placed in the output. We can think of this `<xsl:apply-templates>` component as being sort of like a variable or a placeholder within the stylesheet: it says "right here, content is going to flow from the input tree into this place in the output tree." Or maybe it says, more imperatively "Go get that content, apply templates to it, and put it in the output tree!" In a very simple scenario like this one, that content is very simple and the templates being applied are very simple, so the process is easy to follow:

- we go back to the input document, to the element that we matched earlier (i.e. `<paragraph>`) and we get the content from there
- then we put it into the appropriate place in the output tree

http://2.bp.blogspot.com/-l23P6-iCPNc/TfDqRvDzWsI/AAAAAAAAAUw/JjaV5_QVzRg/s1600/fetchblog3.jpg

A more detailed example

The Input Document

```
<?xml version="1.0" encoding="UTF-8"?>
<lg>
  <opener>Eggs</opener>
  <l>Let's think of eggs</l>
  <l>They have no legs</l>
  <l>Chickens come from eggs,</l>
  <l>But they have legs.</l>
  <l>The plot thickens—</l>
  <l>Eggs come from chickens</l>
  <l>But have no legs under 'em:</l>
  <l>What a conundrum!</l>
  <byline>Ogden Nash</byline>
</lg>
```

The Output Document

```
<html>
  <head>
    <title>A short test document</title>
  </head>
  <body>
    <h2>Eggs</h2>
    <p>Let's think of eggs</p>
    <p>They have no legs</p>
     ...
    <p style="text-indent:50px">Ogden Nash</p>
  </body>
</html>
```

The Stylesheet

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

   <xsl:template match="lg">
      <html>
         <head>
            <title>A short test document</title>
         </head>
         <body><xsl:apply-templates/></body>
      </html>
   </xsl:template>

   <xsl:template match="opener">
      <h2><xsl:apply-templates/></h2>
   </xsl:template>

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

   <xsl:template match="byline">
      <p style="text-indent:50px">
         <xsl:apply-templates/>
      </p>
   </xsl:template>

</xsl:stylesheet>
```

So now let's take a closer look at the stylesheet...

Here's another view of our first scenario, this time with the actual data showing in XML notation

This slide will be useful to come back to, but let's look closely at a few specific pieces in an introductory way...we'll come back to them in more detail later on.

First, let's look at the templates themselves:

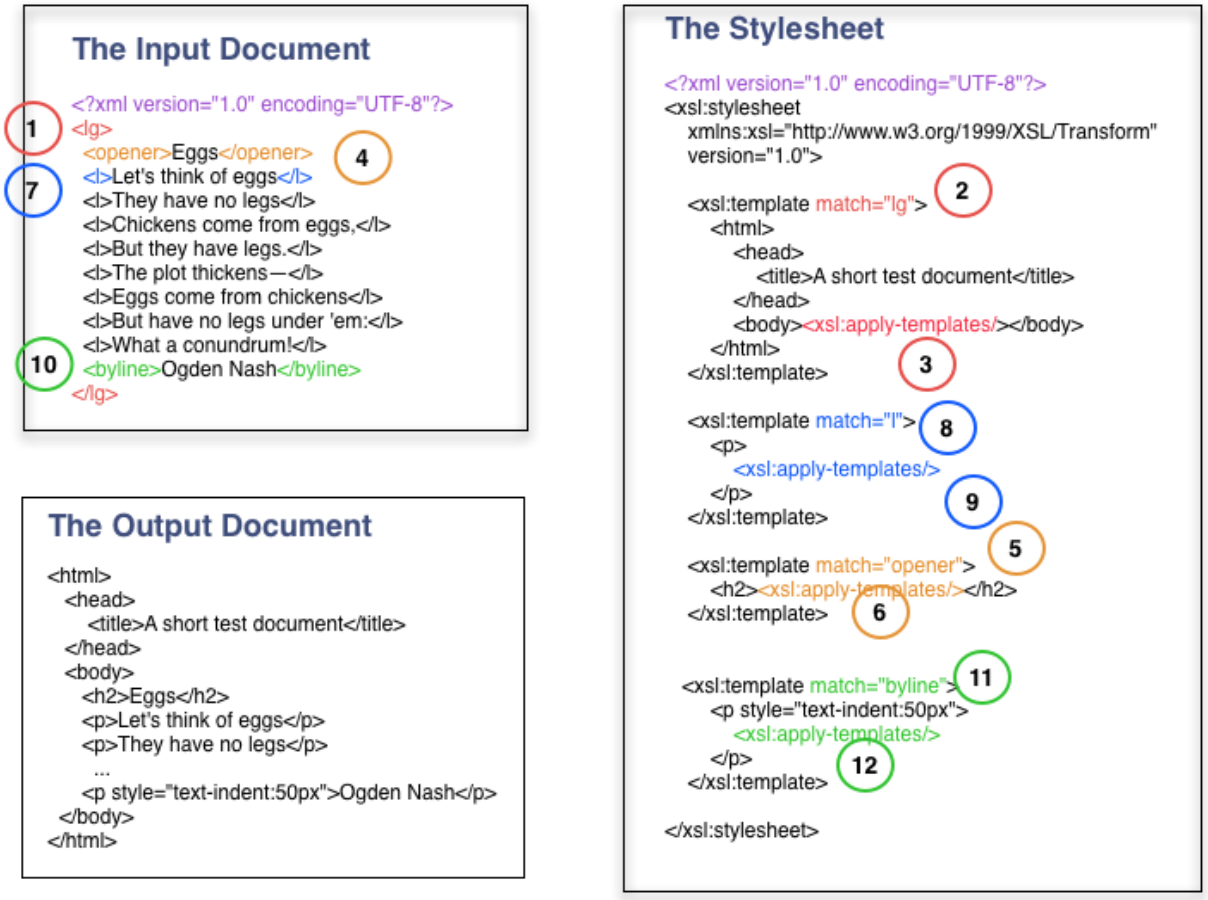
- templates in the stylesheet (those instructions we saw), represented by `<xsl:template>`
- the `@match` attribute: tells us what element in the input tree we are dealing with: this is also the "if", the "condition" that has to be met. The `@match` attribute says basically "is there such an element? If so..."
- let's also note the fact that the value of @match is a location in the input tree; we can specify a context as well and we'll see how to do this later (don't gloss this yet)

How about what a template does? Inside these templates here we have basically two things:

- a snippet of the output tree (in this case, HTML): these are basically the information skeleton of the output document
- an instruction to "apply templates": what this means is, "keep going!" "don't stop here". Another more precise way of interpreting this element is to say "go ahead and process any children of the matched element": in other words, keep drilling down into the input tree. Without this instruction, the stylesheet logic reaches a dead end.

Note that both of these are optional: might be absent...

What order do things happen?

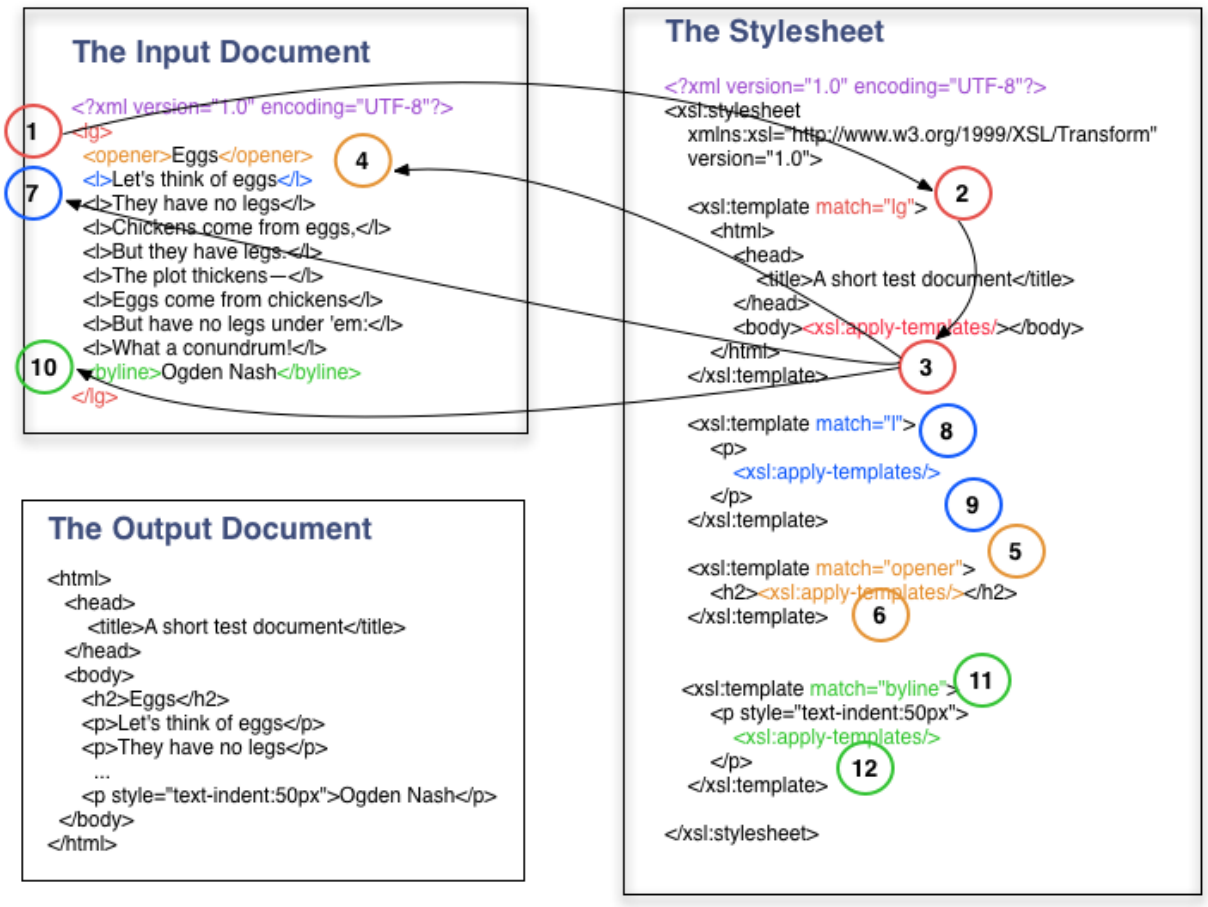


The trickiest thing about XSLT is perhaps understanding the order in which things happen, and the logic that determines what happens next. It's important to understand this because otherwise you can get very puzzling results.

We're used to thinking of computer programs as sets of steps that will happen in order, where the sequence of events is determined by the computer program itself. XSLT is different, in that the order of steps is primarily determined by the input tree:

- You start with the root of the input tree (for now, we'll pretend that's the root element)
- Then you go to the stylesheet to look for the template that matches that root.
- When you find the template that matches, you do what it says
- And if it contains an instruction to continue applying templates, then you go back to the input tree and consider the children of that matched element.
- And for each child, you go back to the stylesheet and see whether there are any templates that match it.

Recursion and dead ends



So the process is recursive, in that you keep drilling down into the input tree and processing the children, and then their children, and then their children

But it can also come to a dead end: that drilling process requires that we receive an instruction to keep applying templates

So in this diagram what we're seeing is how in fact the "apply templates" at step 3 is really the key to accessing all of the other templates in the stylesheet. If that instruction isn't there, nothing else happens. We never get to find out whether the `<opener>` or the `<l>` or `<byline>` is matched.

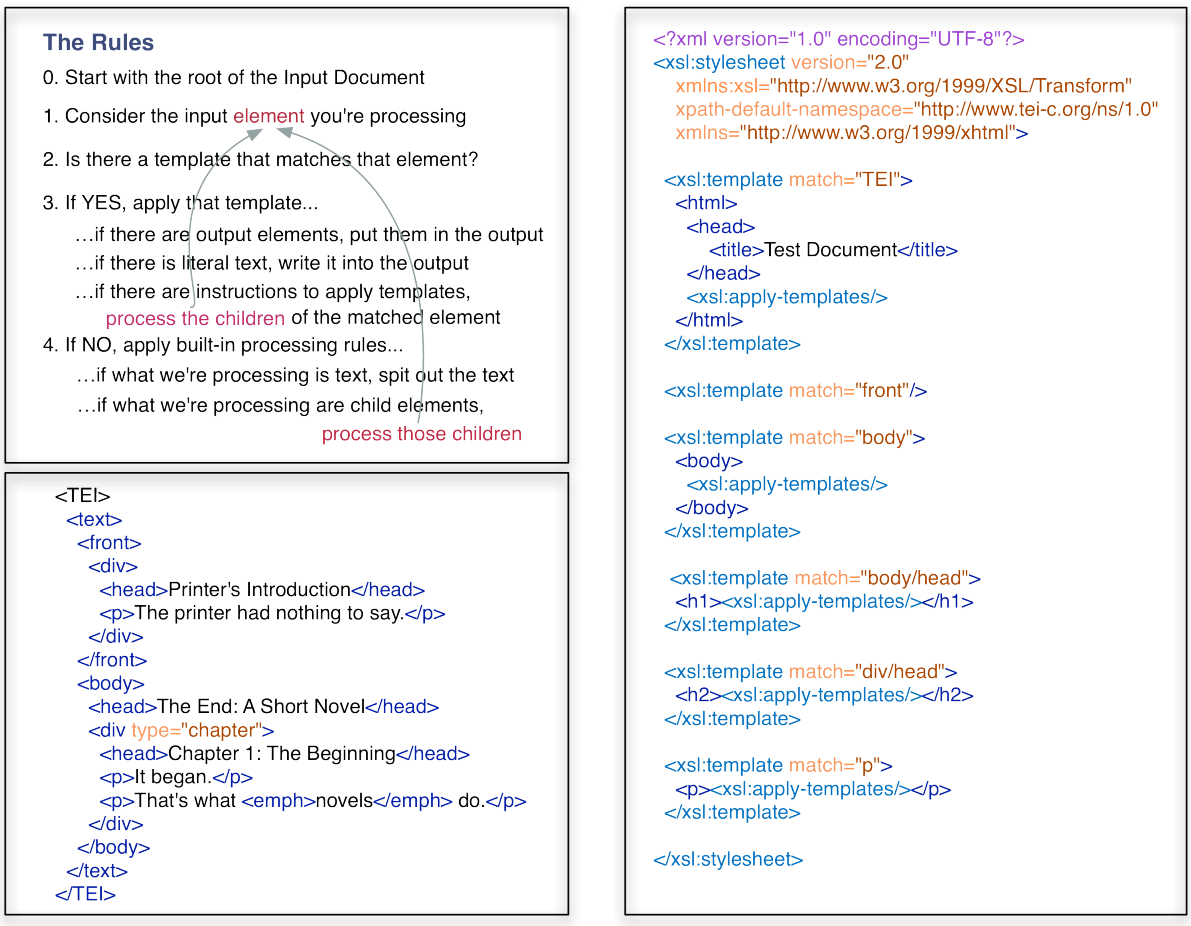
Over to a real example ...

Let's look at this same example in Oxygen.

The XSLT Processing Model

Julia Flanders and Syd Bauman

The XSLT processing model: overview



So let's come back now and consider in more detail how an XSLT stylesheet runs and what it does. What we're talking about here is something called the *XSLT processing model* and it is essentially the set of rules that direct how the stylesheet will run and what it will do, in what order. These rules are actually fairly simple once you're familiar with them, although they're not perfectly intuitive at the outset--so we're going to step through them in detail with an actual example and follow how they work.

Essentially we are starting with the root of the input document, and working our way through that tree, based on the templates we find in the stylesheet.

The XSLT processing model: matching the root

The Rules

0. Start with the root of the Input Document

1. Consider the input element you're processing

2. Is there a template that matches that element?

3. If YES, apply that template...
   - …if there are output elements, put them in the output
   - …if there is literal text, write it into the output
   - …if there are instructions to apply templates, process the children of the matched element

4. If NO, apply built-in processing rules...
   - …if what we're processing is text, spit out the text
   - …if what we're processing are child elements, process those children

```
<TEI>
 <text>
  <front>
   <div>
    <head>Printer's Introduction</head>
    <p>The printer had nothing to say.</p>
   </div>
  </front>
  <body>
   <head>The End: A Short Novel</head>
   <div type="chapter">
    <head>Chapter 1: The Beginning</head>
    <p>It began.</p>
    <p>That's what <emph>novels</emph> do.</p>
   </div>
  </body>
 </text>
</TEI>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="2.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xpath-default-namespace="http://www.tei-c.org/ns/1.0"
   xmlns="http://www.w3.org/1999/xhtml">

 <xsl:template match="TEI">
  <html>
   <head>
     <title>Test Document</title>
   </head>
   <xsl:apply-templates/>
  </html>
 </xsl:template>

 <xsl:template match="front"/>

 <xsl:template match="body">
  <body>
   <xsl:apply-templates/>
  </body>
 </xsl:template>

 <xsl:template match="body/head">
  <h1><xsl:apply-templates/></h1>
 </xsl:template>

 <xsl:template match="div/head">
  <h2><xsl:apply-templates/></h2>
 </xsl:template>

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

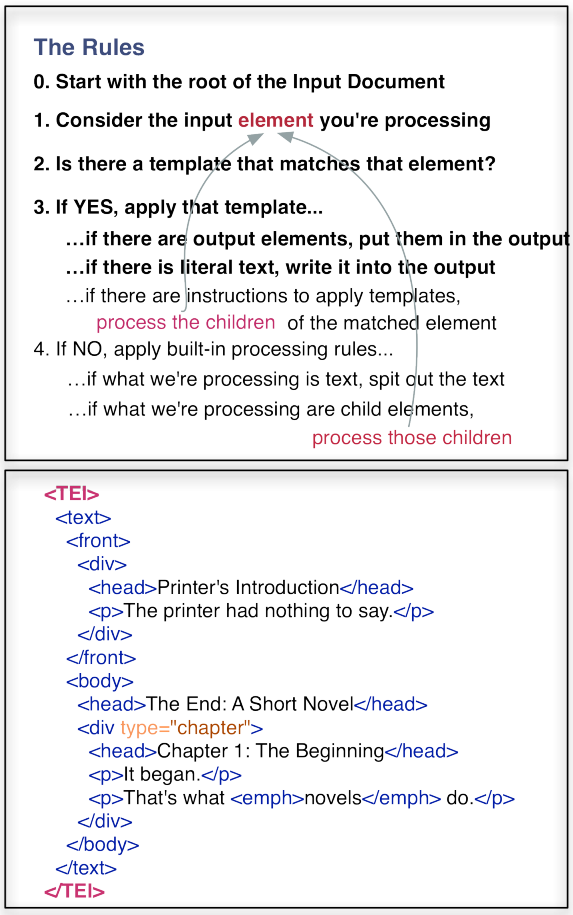
</xsl:stylesheet>
```

So what's the first rule? we remember this from our earlier, more informal look:

- start with the root of the input document (what is it?)

- and then ask: is there a template that matches this element I'm considering? (is there?)

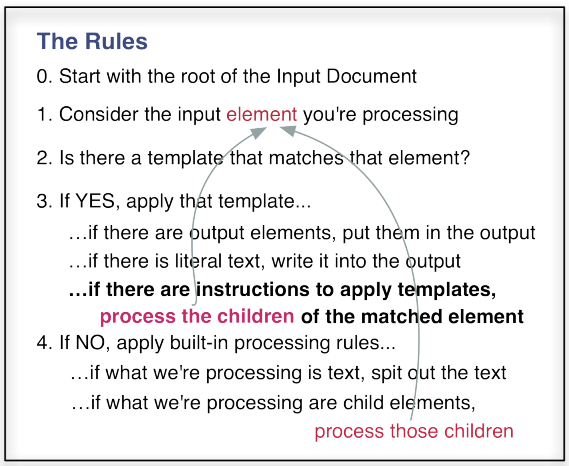
The XSLT processing model: applying a template

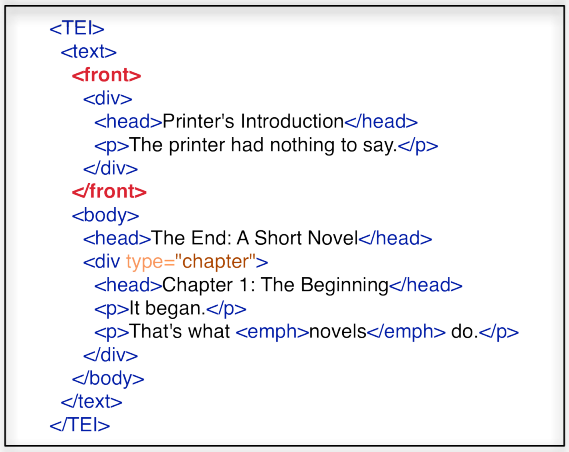


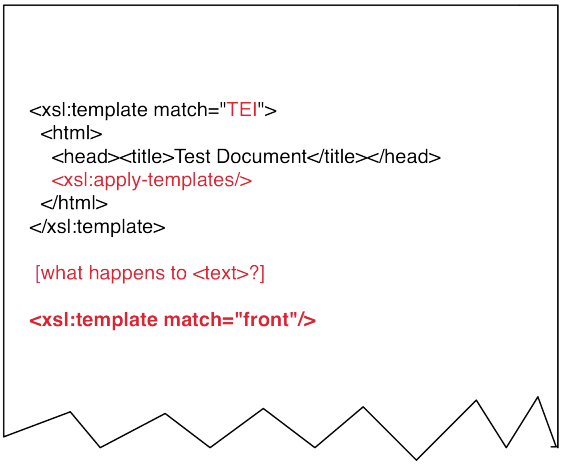
If I do find a template that matches the element I'm considering, I apply that template:

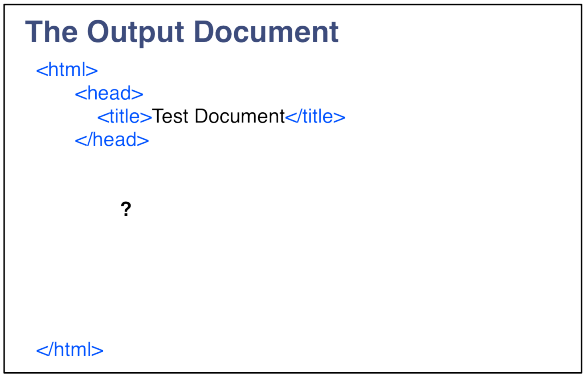
- in this case, what is it doing? (writing out the first few layers of the output tree, and including a little bit of literal text)

The XSLT processing model: processing children









So what did we just do? We applied a template, which entails:

- putting out the output elements
- writing out any literal text
- and one thing further: if there are instructions to apply templates, then process the children of the matched element

So what is the element we just matched? (`<TEI>`)

And it does include instructions to apply templates, which means that we... (process the children of the matched element, i.e. the children of `<TEI>`

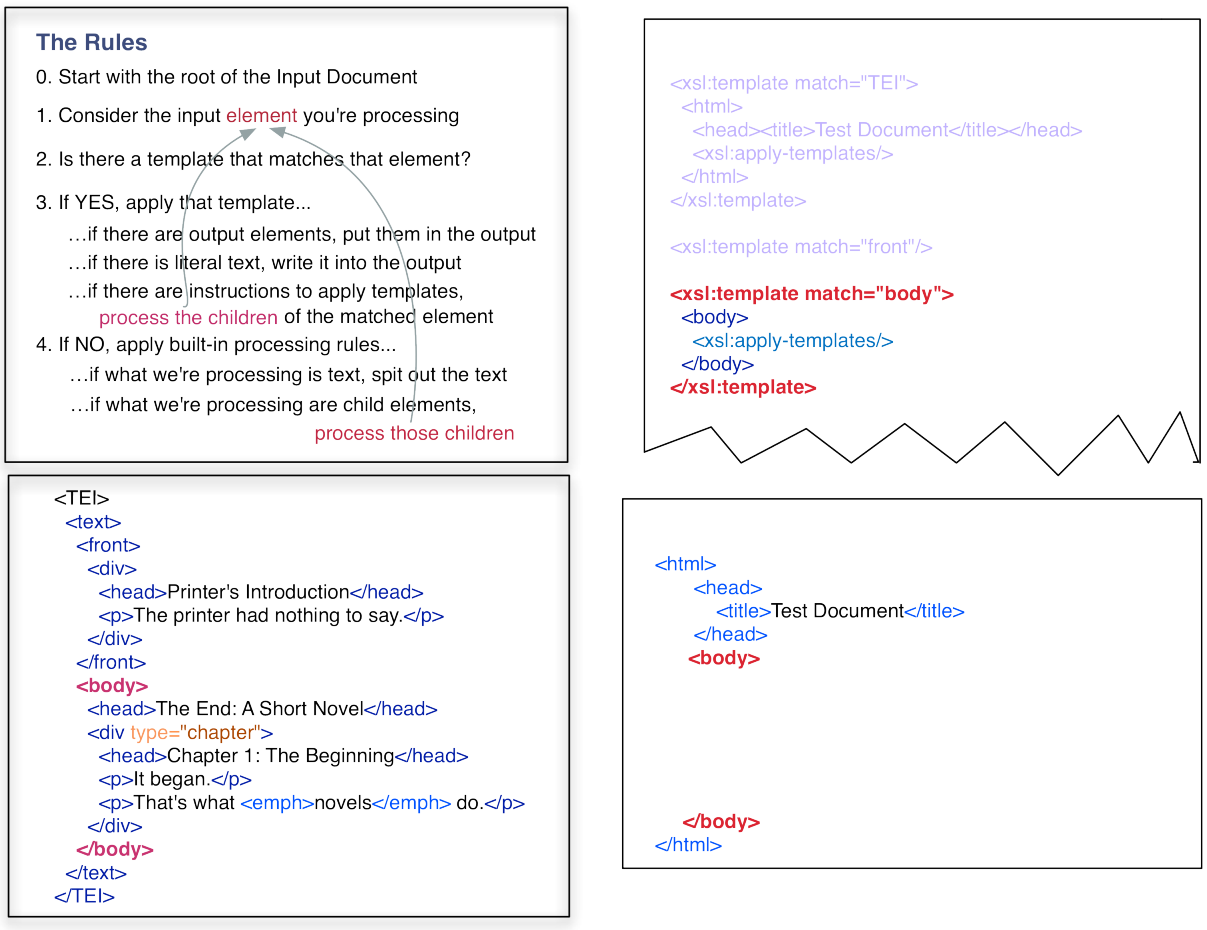
So what are these children? `<text>`: what do we do with this? What rule applies in this case? (no template matches it: so we apply built-in processing rules, which say that we... (spit out any text, and process any children)

So what are the children of `<text>`?...`<front>`: what do we do with this? What rule applies in this case?

There's a template that matches `<front>`, but what does it tell us to do? What rule applies in this case?... there are no instructions to apply templates, so this part of the process stops there. There's no output from `<front>`.

Does it stop altogether? Or are there other loose ends that will keep the process going?

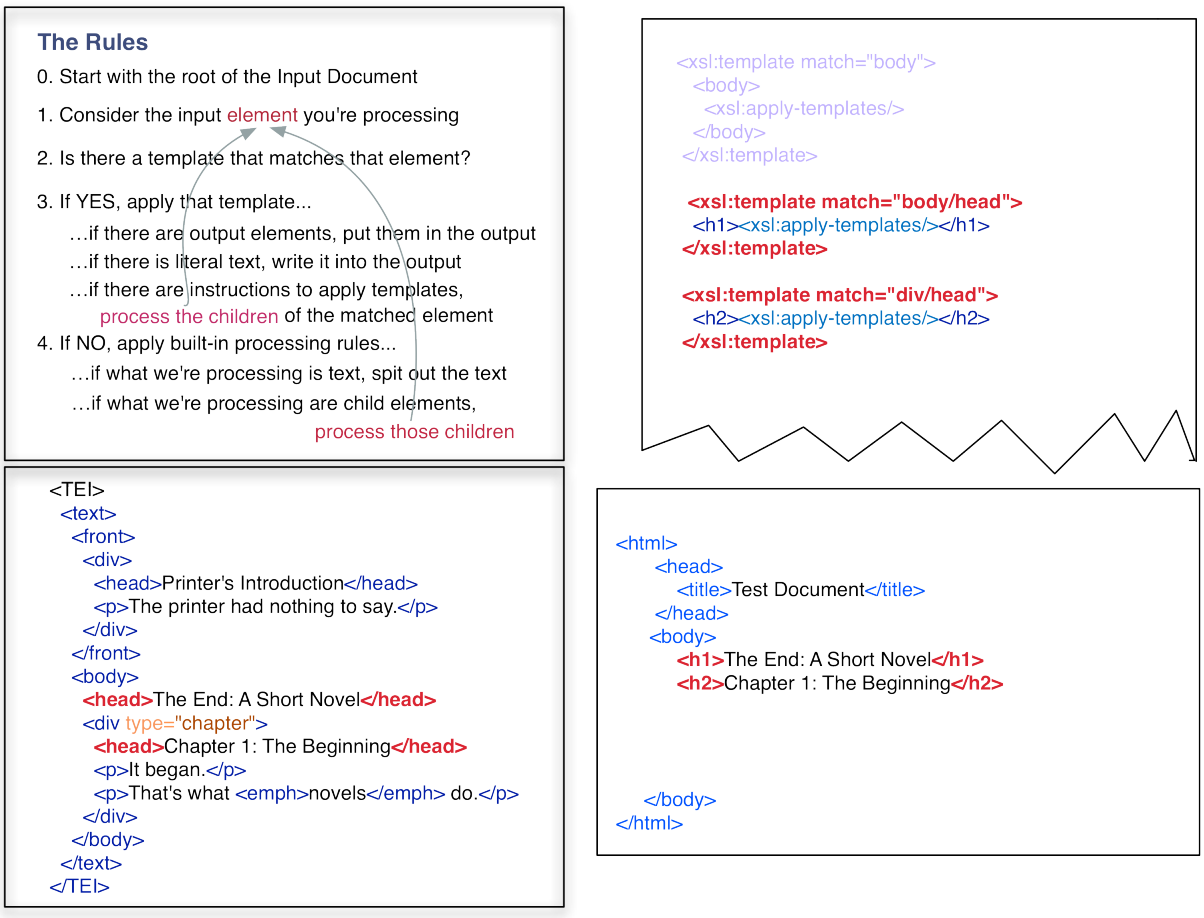
The XSLT processing model: chugging along



There's another child of `<text>`, namely `<body>`, so our built-in stylesheet rule of "process the children" applies here and allows us to proceed to the `<body>` element

So what happens here? (another output element is generated, and inside it, additional templates will be applied)

The XSLT processing model: processing more children

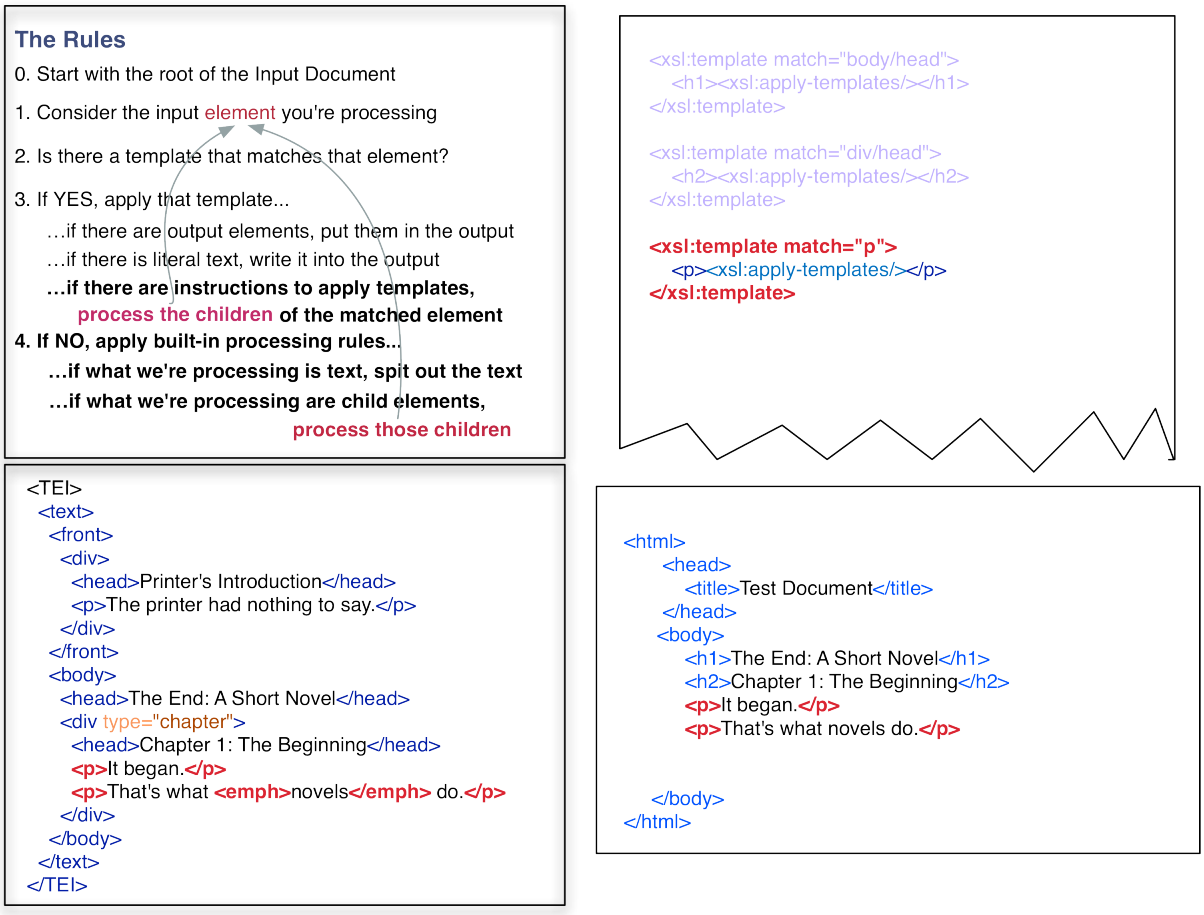


Next we start processing the children of `<body>`, and we have two templates here that do somewhat similar things; what are we matching here?

What if we had wanted to just match any `<head>` in the input document?

Why do it this way? Why distinguish between two different locations for `<head>`

The XSLT processing model: a final round



Finally we're getting to the last of the children in the input document... What is happening here? What rules apply when we get to `<emph>`? (There's no template that matches, so we apply the built-in rules, which say...if what we're processing is text, spit out the text)

The XSLT processing model: a last look

The Input Document

```
<TEI>
 <text>
  <front>
   <div>
    <head>Printer's Introduction</head>
    <p>The printer had nothing to say.</p>
   </div>
  </front>
  <body>
   <head>The End: A Short Novel</head>
   <div type="chapter">
    <head>Chapter 1: The Beginning</head>
    <p>It began.</p>
    <p>That's what <emph>novels</emph> do.</p>
   </div>
  </body>
 </text>
</TEI>
```

The Stylesheet

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="2.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xpath-default-namespace="http://www.tei-c.org/ns/1.0"
   xmlns="http://www.w3.org/1999/xhtml">

 <xsl:template match="TEI">
  <html>
   <head>
     <title>Test Document</title>
   </head>
   <xsl:apply-templates/>
  </html>
 </xsl:template>

 <xsl:template match="front"/>

 <xsl:template match="body">
  <body>
   <xsl:apply-templates/>
  </body>
 </xsl:template>

 <xsl:template match="body/head">
  <h1><xsl:apply-templates/></h1>
 </xsl:template>

 <xsl:template match="div/head">
  <h2><xsl:apply-templates/></h2>
 </xsl:template>

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

</xsl:stylesheet>
```

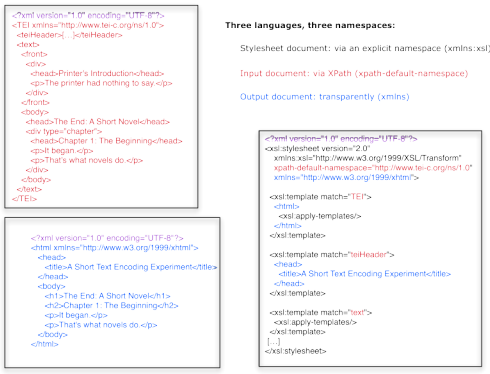
The Output Document

```
<html>
   <head>
      <title>Test Document</title>
   </head>
   <body>
      <h1>The End: A Short Novel</h1>
      <h2>Chapter 1: The Beginning</h2>
      <p>It began.</p>
      <p>That's what novels do.</p>
   </body>
</html>
```

Let's look at this in oxygen...

Now we can back up again and study the whole thing as a finished product: the input, the stylesheet, and the output. Any questions?

Namespaces and languages



You've probably already noticed that we're dealing here with three languages:

- The language of the XSLT stylesheet itself (which is a language containing elements like `<tempate>` and `<apply-templates>`)
- The language of the input document (in this case, TEI)
- The language of the output document (in this case, HTML)

Within the stylesheet itself, we need to keep these three different languages distinct from one another, so that the processor always knows what piece of what tree it is dealing with. We do this with something called namespaces. (Does everyone understand namespaces? Quick review on the next slide if necessary...)

Each of these languages plays a specific role in the stylesheet ecology and gets referenced in a distinctive way:

- Let's take the simplest first: the output tree, which is being treated transparently in our examples: it doesn't use a namespace prefix, and this is because we have declared that the entire stylesheet is in the HTML namespace (we did this with the namespace declaration attribute-like thingy: xmlns)
- The next fairly simple case is the input tree, which also looks as if it's not getting a namespace. How are we keeping this separate from the output tree? The trick here is that the input tree is always accessed via these `@match` and `@select` (and similar) attributes. These attributes all access the input tree via XPath, and up at the top, we provided a default namespace for all XPaths (via xpath-default-namespace)

- And finally, the stylesheet document has its namespace specified as the XSL namespace (the default namespace for the stylesheet is already set to HTML) so all of the stylesheet elements have a namespace prefix.

Namespaces review

Without the genus, we don't know what animal these species are:
- glauca: a pine tree (Picea glauca) or a small yellow flower (Agoceris glauca)?
- leucocephalus: a cactus (Pilosocereus leucocephalus) or a bald eagle (Haliaeetus leucocephalus)?

Without knowing the language, we don't know these words mean:
- the (English definite article or a French hot drink?)
- bad (English adjective or German noun for "bath"?)

Without a namespace designation, we don't know what these elements mean:
- `<p>` (TEI paragraph or HTML block element?)
- `<div>` (TEI textual division or HTML grouping element?)
- `<fileDesc>` (TEI or EAD?)

With the namespace, all is clear:
- ```
  <tei:p>
  ```
- ```
  <html:div>
  ```
- ```
  <ead:fileDesc>
  ```

The namespace prefix is somewhat like a genus or language name: it tells us more precisely what language we are speaking (and hence what the semantics of the element are)
  ```

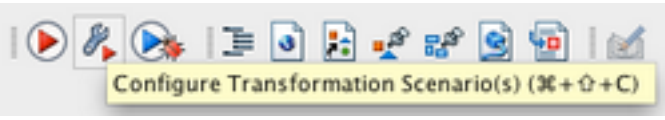
XSLT Workshop: Creating a Transformation Scenario in oXygen (version 14)

What is a transformation scenario?

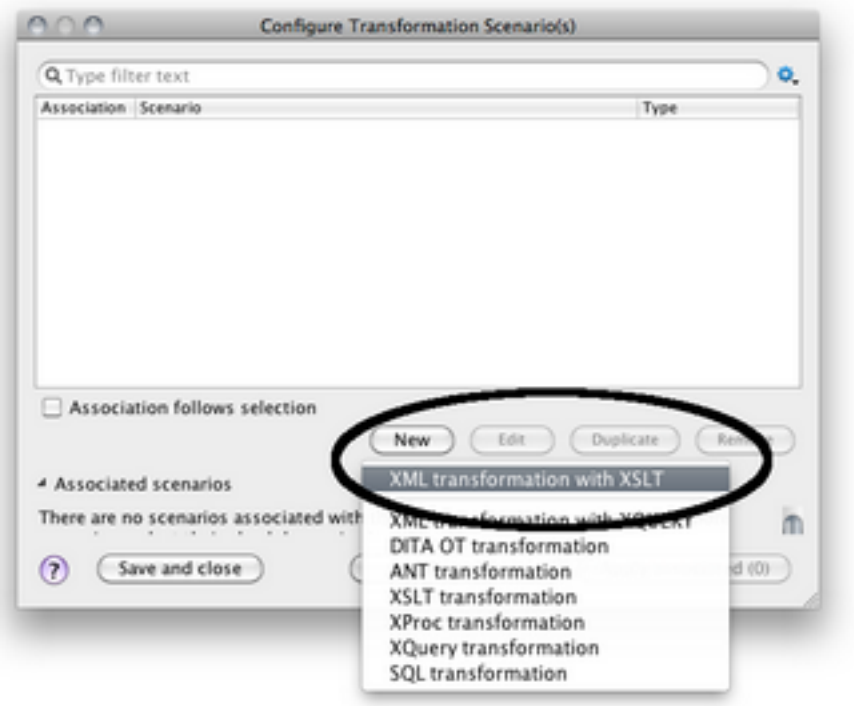
A transformation scenario is like a macro which enables you to run an XML file quickly through an XSLT transformation, save the resulting output, and then view it in an appropriate program such as a browser. So in a transformation scenario, you specify:

- An input file (this is optional)
- An XSLT file
- An XSLT engine to do the work
- An output file pattern
- An action to take when the transformation is complete (such as opening the result in a browser)

Getting started

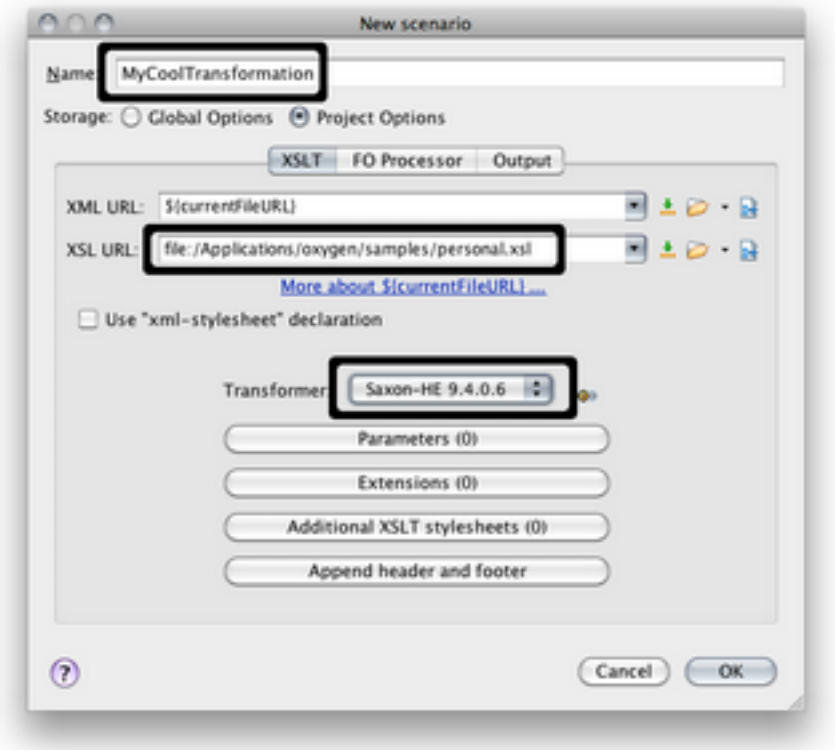


Click on the toolbar button to configure the transformation scenario. You'll see this dialog box, showing all the transformation scenarios that are available now:



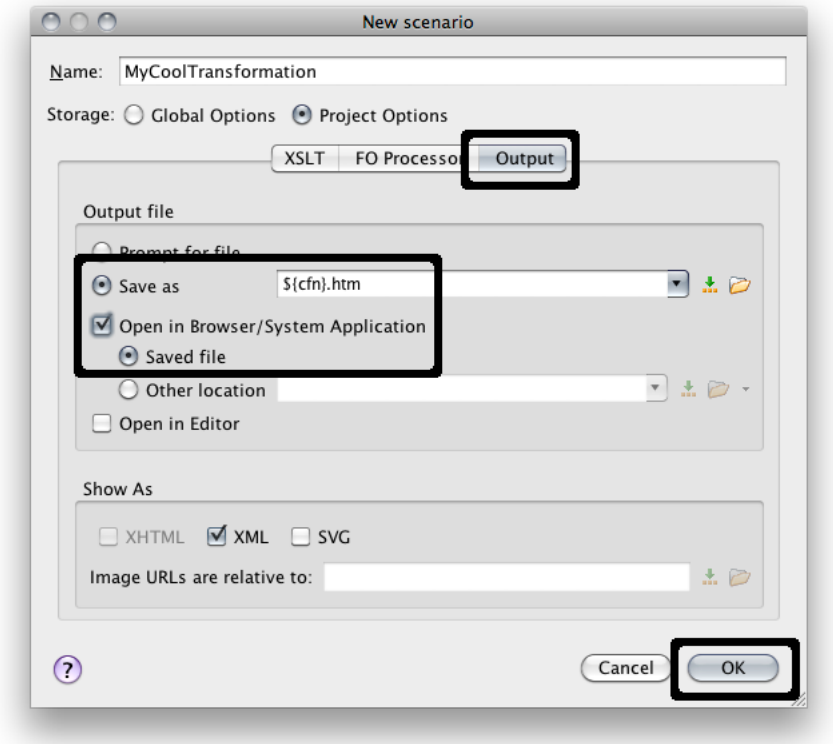
Click on the "New" button to create your new scenario.

Specify the scenario name, the XSLT file and the processor



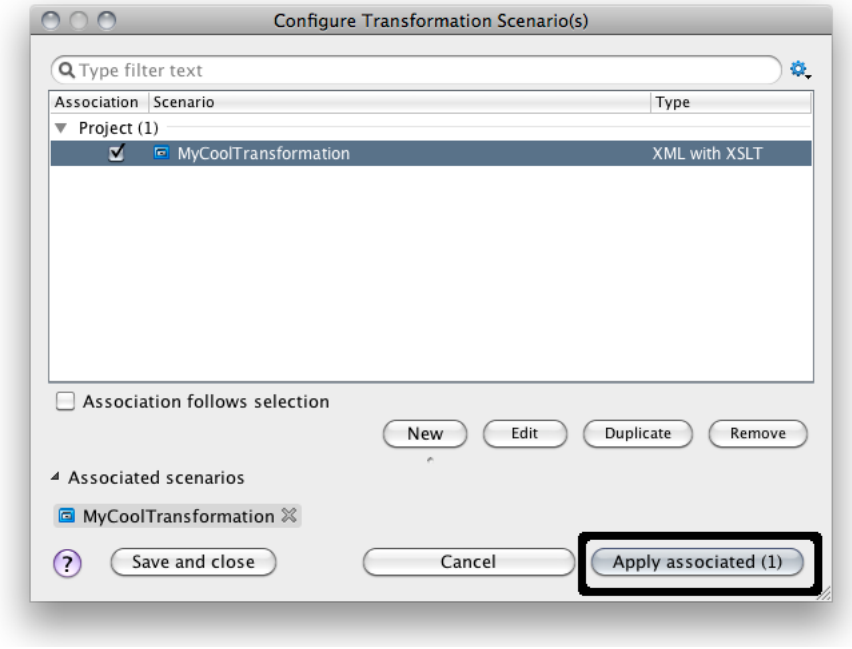
Give your transformation a name, then navigate to the XSLT file you want to use to do the transformation. Finally, make sure you select Saxon 9 as your transformer. The minor version may vary, but you want to use version 9 or above, not an older version.

Configure the output

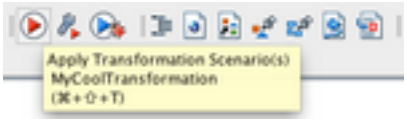


Choose the Output tab, then specify the name to save the file to. In the screenshot, you see ${cfn}.htm; this means "use the same file path and name as the input file, but give it an extension of .htm". This is appropriate if (for instance) you are transforming an XML file into XHTML. Next, select the "Open in browser" option so that you can see the result in your browser automatically. Finally, press "OK" to save your scenario into the list.

Using your scenario



You should now see your scenario in the list of scenarios available. You can run it immediately by clicking on "Transform now", or you can use the scenario on the XML file you happen to be working on by clicking on the appropriate button in the toolbar:



Crib Sheet for oXygen

Julia Flanders
Syd Bauman

in order to …	in oXygen …
insert an element	type **<**, and then scroll through the pop-up list of available element names; note that typing the first few letters of an element's name will select it
insert an attribute	position your cursor before the **>** of a start-tag (or before the **/** of an empty-element tag) and type a space, then scroll through the pop-up list of available attribute names; note that typing the first few letters of an attribute's name will select it
validate	press **cmd-shift-v** (or **ctl-shift-v** on PCs)
indent (i.e., make pretty)	press **cmd-shift-p** (or **ctl-shift-p** on PCs)
request *'completion'*	press **ctl-space**
request *'code template'*	press **ctl-shift-space**
configure transformation scenario	press **cmd-shift-c**
run configured transformation	press **cmd-shift-t**
switch to XSLT Debugger perspective	choose `Window > Open perspective > XSLT Debugger`
switch to Editor perspective	choose `Window > Open perspective > Editor`
make text into an element (i.e., surround it with tags)	select text of interest, and press **cmd-e** (or **ctl-e** on PCs), then scroll through the `surround` window's list of available element names; note that typing the first few letters of an element's name will select it
rename an element	place cursor on either start- or end-tag, and press **opt-cmd-r** (or **alt-shift-r** on PCs)
split element	press **ctl-opt-d**

XSLT Workshop: Basic XHTML Crib Sheet

This is a small subset of XHTML elements and attributes. For a full reference, check out http://www.w3schools.com/xhtml/.

XHTML elements: Main document divisions

`<html xmlns="http://www.w3.org/1999/xhtml">`
: The root element for your document. This should always have the XHTML namespace in it.

`<head>`
: Place for metadata, like the `<teiHeader>`.

`<body>`
: Place for page content, like TEI `<text>`.

XHTML elements: Header elements

`<title>`
: The document title (displayed in the browser caption bar).

`<link rel="stylesheet" type="text/css" href="style.css"/>`
: Links the document to a CSS stylesheet. All three of these attributes are required.

`<meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>`
: Metadata tag showing the MIME type and the character encoding. Include this, but don't worry about it.

XHTML elements: Block elements

`<div>`
: A division or section (much like in TEI).

`<h1>, <h2>, ... <h6>`
: Six levels of heading (higher numbers mean lower level or less important).

`<p>`
: A paragraph, i.e. a block of "flowed" text.

`<blockquote>`
: An indented block, suitable for a block quotation.

`<ul>`
: An unordered list (i.e. a bulleted list).

`<ol>`
: An ordered list (i.e. a numbered list).

`<li>`
: A list item, inside either of the two list types above.

XHTML elements: Inline elements

`<span>`
: An anonymous span of text (used for highlighting, identification, styling, etc.).

`<q>`
: A short inline quotation. (I.e., surrounded by quotation marks.)

`<a href="http://www.tei-c.org">`
: An “anchor” or link tag (requires an `href=` attribute if intended to be a link; requires an `id=` attribute if intended to be a target).

`<em>`
: Emphasized text (defaults to italics).

`<strong>`
: Strongly emphasized text (defaults to bold).

`<code>`
: Computer code.

XHTML attributes

`id=`
: Like an `xml:id=`, uniquely identifies a particular occurence of an element. Any element may have one, but the value must be unique within the document.

`title=`
: “Tooltip” text which shows up when mousing over an element. Any displayed element may have this.

`class=`

One or more classnames (separated by spaces). Classnames can be used to apply CSS styling to elements in an efficient manner. Any displayed element may use this.

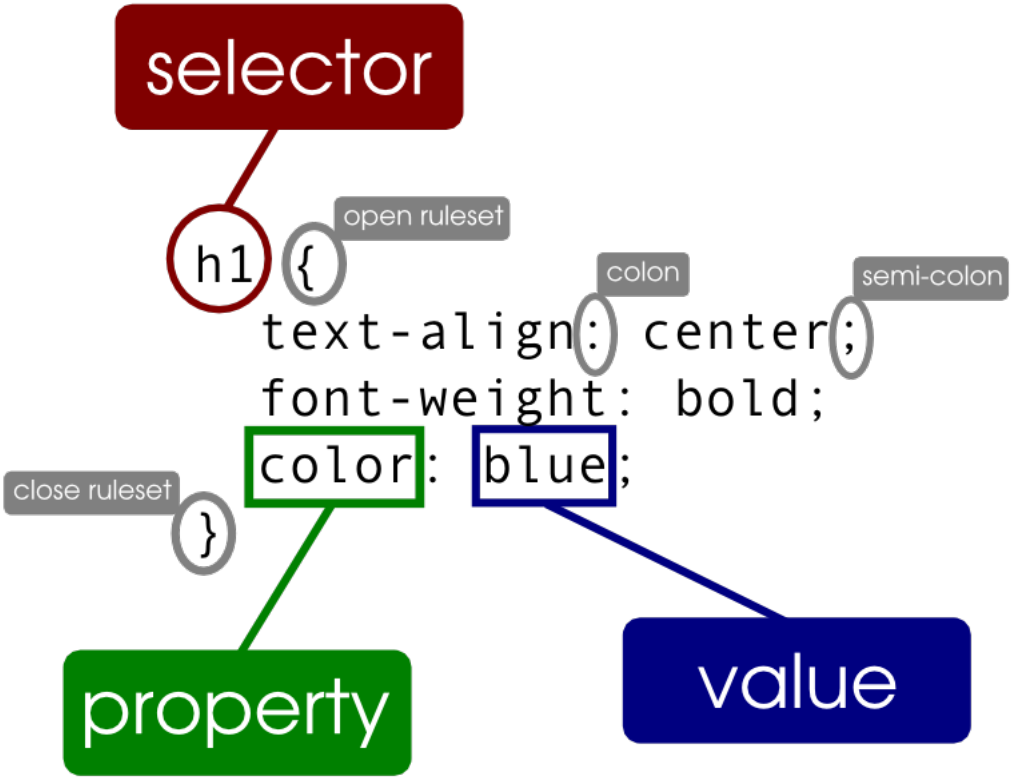
`href=`

The “hypertext reference” attribute’s value is the URI of the target of the link (when used on an `<a>`) or of the stylesheet to be used (when used on a `<link>`).

XSLT Workshop: Basic CSS Crib Sheet

This is a very brief reference sheet than an exhaustive list of CSS terms: it is intended to provide you with a way to look up the information you're most likely to need right away. For detailed information about CSS syntax, selectors, and properties, visit the W3C's CSS tutorial pages at http://www.w3schools.com/css/.

CSS Rulesets



The components of a ruleset.

A CSS stylesheet is essentially a group of rulesets. Rulesets are made up of three things:

selectors
: These are how you select which elements or groups of elements in your document will be affected by the style information.

properties
: These identify the style properties (such as size or color) that are being controlled.

values
: Each property has a specific value indicating the size, color, margin, or other style information that is being applied.

Selectors

A selector is the way you identify the elements or groups of elements that will be affected by your style information. A selector may be simply the name of an element, or they may take into account the nesting of the element, or a specific attribute value.

Here's a list of the most common selectors and their syntax.

element:
: h1{font-size: 125%;}

element descendant-element:
: p strong{font-style: italic;}

element.class:
: span.speakerName{font-weight: bold;}

element #id:
: div#menu{background-color: grey;}

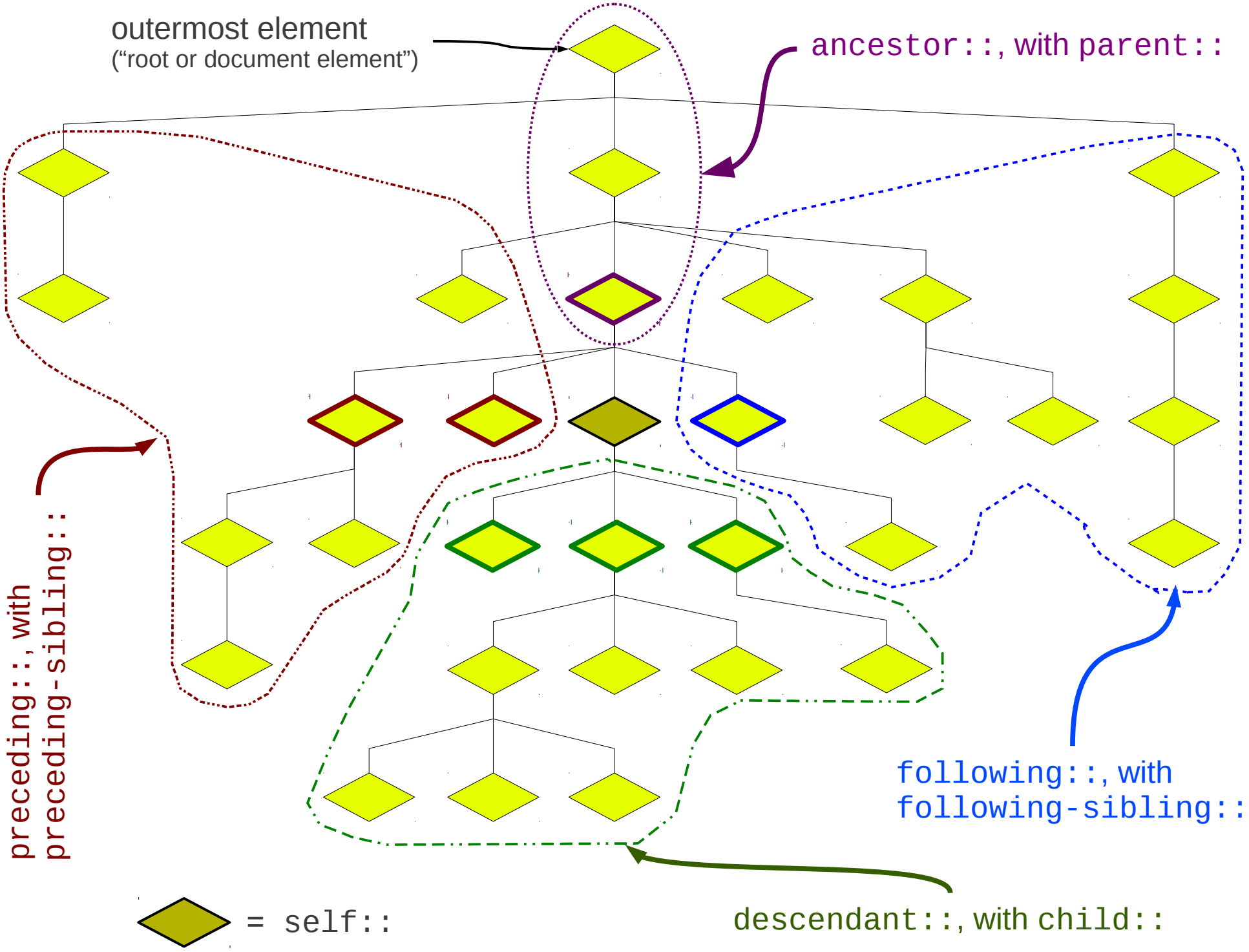
Properties and values

Here's a list of common properties and their most common kinds of values. Note that some properties (especially those that govern size) can be measured in several different ways. Some of the measurements are in absolute units such as pixels; others are relative to the font size (for instance, ems), and others are relative to the surrounding text (for instance, percentages or "larger" or " smaller").

- display: none | block | inline
- margin-left, margin-right, margin-top, margin-bottom: 1em | 2ex
- padding-left, padding-right, padding-top, padding-bottom: 1em | 2ex
- border: thin solid blue;
- text-align: left | right | center | justify
- line-height: %
- text-indent: 1em | 48px
- text-decoration: underline | overline
- font-size: xx-small | x-small | small | medium | large | x-large | xx-large

- font-size: % | smaller | larger
- font-style: normal | italic
- font-weight: normal | bold
- font-family: serif | sans-serif | cursive | fantasy | monospace
- color: red | #99AABB
- background-color: red | #99AABB
- background-image: url("./image.png")
- background-repeat: repeat | repeat-x | repeat-y | no-repeat
- background-attachment: scroll | fixed
- background-position: (top | center | bottom) (left | center | right)
- background-position: 20% 70%

For more detailed information on CSS colors, see
http://www.w3schools.com/css/css_colornames.asp


outermost element
(“root or document element”)
ancestor::, with parent::
preceding::, with
preceding-sibling::
following::, with
following-sibling::
= self::
descendant::, with child::

XSLT Workshop: Basic XPath Functions Crib Sheet

This is a small subset of useful XPath 2.0 functions. For a full reference, check out http://www.w3schools.com/xpath/xpath_functions.asp and http://www.w3.org/TR/xpath-functions/.

String functions (operations on text)

normalize-space(*string*)

Trims leading and trailing spaces, and collapses all internal sequences of spaces to a single space.

```
normalize-space('  this   has lots    of spaces    ')
  returns:
  "this has lots of spaces"
```

contains(*haystack*, *needle*)

Returns true or false, depending on whether the first string contains the second string or not.

```
contains(//TEI/text/body/div[1]/head[1], 'Chapter')
  returns:
  true() if the first head in the first div in the body contains "Chapter",
  false() if it doesn't.
  Beware: this is case-sensitive.
```

replace(*string*, *pattern*, *replacement*)

A regular-expression replace function. Returns a copy of string in which all instances of the pattern have been replaced by the replacement.

```
replace('XPath is cool.', 'cool', 'very cool')
  returns:
  "XPath is very cool."
```

starts-with(*string1*, *string2*)

Returns true if string1 starts with string2, and false if not.

```
starts-with(//TEI/text/body/div[1]/head[1], 'Chapter')
  returns:
  true() if the first <head> in the first <div> in the <body> begins with "Chapter",
  false() if it doesn't.
  Beware: this is case-sensitive.
```

ends-with(*string1*, *string2*)

Returns true if string1 ends with string2, and false if not.

```
ends-with(//TEI/text/body/div[1]/head[1], 'ide')
  returns:
  true() if the first <head> in the first <div> in the <body> ends with "ide",
  false() if it doesn't.
  Beware: this is case-sensitive.
```

Functions for handling sequences

Sequences are rather like arrays in traditional programming. In XPath, they are ubiquitous; most simple XPath expressions will return a sequence. For instance, `//div/@type` will return a sequence of all the `type=` attributes on `<div>` elements in the document.

distinct-values(*sequence…*)

This returns a sequence of only the **distinct** values in a sequence. In other words, it removes duplicates.

```
distinct-values(//div/@type)
  returns:
  a list of the unique values of the @type attribute on <div> elements.
```

empty(*sequence…*)

Returns true or false depending on whether the sequence is empty.

```
empty(//div[parent::back])
  returns:
  true() if there are any <div> elements in the <back>,
  false() if not.
```

Mathematical functions

count(*sequence…*)

Counts the items fed to it, and returns the total.

```
count(//div)
```

avg(*sequence…*)

Calculates the average from a sequence of numbers.

```
avg(//person/age/@value)
```

```
avg((1,2,3,4,5,6))
```

NOTE: the second example includes two sets of parentheses. The outer set contains the argument to the avg() function, and the inner set constructs a sequence which constitutes the argument to the function. In other words, avg() takes a single argument, which is a sequence; it cannot take a series of separate arguments.

sum(*sequence…*)

Calculates the sum of a list of numbers.

```
sum((count(//person), count(//org)))
```

Note the proliferating parentheses...

min(*sequence…*)

Returns the smallest from a list of numbers.

```
min(//person/age/@value)
```

max(*sequence…*)

Returns the largest a list of numbers.

```
max(//person/age/@value)
```

round(*number*)

Rounds a number to the nearest integer. .5 is rounded up.

```
round(1.6)
  returns:
  2
```

Boolean functions (dealing with true or false values)

not(*expression*)

returns true if the expression is false, and false if it is true.

```
not(count(//p) gt 5)
  returns:
  false() if there are 6 or more <div> in the document,
  true() if there are 5 or fewer.
```

true(), false()

`true()` and `false()` are functions in XPath. if you're checking the result of a boolean operation, don't forget the parentheses!

```
<xsl:if test="(count(//div) lt 5) = false()">
```

Context functions

position(*node*)

returns the position of the node in its sequence.

```
//div[position() = 1]
  returns:
  the first <div>.
```

last()

returns the number of the last item in the sequence.

```
//div[position() = last()]
  returns:
  the last <div>.
```