Meaningful Markup

XML and Web Information

Gary Stringer

Creative Media and Information Technology

University of Exeter, UK

Meaningful Markup: XML and Web Information

Gary Stringer

Copyright © 2006-2008 Gary Stringer / University of Exeter

Legal Notice

The right of Gary Stringer to be identified as the author of this work has been asserted by him in accordance with the Copyright, Designs and Patents Act 1988.

This work is licensed under a Creative Commons License. Some rights reserved:

Attribution-NonCommercial-ShareAlike 2.0 United Kingdom .

You are free:

- to copy, distribute, display, and perform the work
- to make derivative works

Under the following conditions:

BY - Attribution	You must give the original author credit.
NC - Non-Commercial	You may not use this work for commercial purposes.
SA - Share Alike	If you alter, transform, or build upon this work, you may distribute the resulting work only under a licence identical to this one.

- For any reuse or distribution, you must make clear to others the licence terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

This is a human-readable summary of the Legal Code (the full licence) [http://creativecommons.org/licenses/by-nc-sa/2.0/uk/legalcode].

Please also read the Disclaimer [http://creativecommons.org/licenses/disclaimer-popup?lang=en_GB]

Table of Contents

About this Module ... ix
- Prerequisites ... ix
- Essential Reading ... ix

1. Introducing XML ... 1
- What is XML? ... 1
- Strictly XML ... 2
- The X Word: XML-related technologies and acronyms ... 4
- SGML to XML, HTML to XHTML ... 5

2. Different Types of Document ... 7
- Hypertext Markup Language ... 7
- Mathematical Markup Language (MathML) ... 9
- RSS - Really Simple Syndication ... 9
- DocBook ... 11
- The Text Encoding Initiative ... 11
- Scalable Vector Graphics (SVG) ... 12
- Declaring a DTD ... 13
- Namespaces ... 15

3. Defining Documents ... 17
- What is a DTD? ... 17
- Defining the Document ... 18
- Defining Elements ... 19
- Adding Attributes ... 22
- Good Document Design ... 23
- Entities and Other Useful(?) Stuff ... 24

4. Cascading Stylesheets ... 29
- Associating a CSS file with an XML document ... 30
- Using the Cascade ... 31
- Advanced CSS Selectors ... 32

5. Introducing XSL Transformations ... 35
- A Simple Stylesheet ... 35
- A More Detailed Look ... 38
- Outputting Markup ... 39
- Tracking the current node ... 41

6. Further XSLT and XPath ... 43
- Control structures ... 43
- Creating new elements - <xsl:element> and <xsl:attribute> ... 44
- Storing values - <xsl:variable> and <xsl:param> ... 45
- More XPath: Tests and operators ... 45
- Some Useful XPath Functions ... 46

7. Using XSLT 2.0 ... 47
- Namespaces ... 47
- Sequences ... 48
- XPath enhancements ... 48
- Functions ... 48
- Frequently Asked Questions ... 49

8. Schemas of various types 51
Why use a schema? 51
So which language should I use? 51
Using the W3C's Schema Description Language 51
9. Making Connections 55
Adding simple links 55
Extended Links 57
Bibliography 59
Glossary 61
Index 63

List of Figures

5.1. The XSLT Processing Model 36

List of Examples

1.1. An example of XML looking suspiciously like HTML 1
1.2. An example of a home-made markup language in XML 2
2.1. A basic XHTML page with minimal presentational markup 8
2.2. A very simple example of MathML 9
2.3. An instance of RSS 2.0 with iTunes and Yahoo media extensions 10
2.4. A very simple TEI P5 document 12
2.5. The simplest of SVG examples 13
2.6. Declaring a DTD internally 13
2.7. Declaring an external DTD 14
3.1. A simple DTD for a jokebook 18
3.2. A minimal jokebook document 18
4.1. Simple CSS stylesheet for jokebook (excerpt) 29
5.1. A simple XSLT stylesheet for jokebook (excerpt) 37
7.1. A namespace-aware XSLT2 stylesheet header 47
7.2. A basic function to lowercase text and remove (some) accents 49
7.3. XInclude example 50
8.1. Starting a new schema 52
8.2. Using a schema to validate XML 52
8.3. Adding documentation to a schema 52
8.4. Some simple element definitions in XSD 53
8.5. XSD: A simple element container 53
8.6. XSD: A user-defined complex type 54
8.7. XSD: Constraining data with a pattern - postcodes 54
9.1. Some XLink/XHTML Equivalences 56
9.2. An image gallery as an extended XLink 57

About this Module

XML is the future of the web; it's a more flexible language than HTML, and can carry much more information. XML separates content and presentation, so that the same document can be viewed on high-resolution displays or mobile phones, and still appear designed for the medium. XML also allows the document author to create their own tags with meaningful names, incorporating semantics into the document structure.

This module is designed to build on the skills you've acquired in The Internet (MIT2114/2214) or Aspects of Web Design (MIT2100/2200), and looks at modern website creation through the use of XML and its related technologies. You'll have the opportunity to create a website using scalable document management techniques, developing for a variety of browsing formats and creating 'house styles' using stylesheets and transformations. The module is practically based, looking at each technology from a 'how-to' perspective, whilst analysing the technologies usefulness and role in document management at each stage.

Prerequisites

We'll begin our look at XML by examining the differences between XML and HTML, so a basic knowledge of how HTML tags work, and a little experience in hand-coding HTML pages, will be useful if not essential.

A knowledge of file management using Microsoft Windows (or other Operating System) will be essential.

The module is taught from a non-technical viewpoint, so skills in computer programming and other advanced subjects are *not* required nor expected!

Essential Reading

The recommended book for the practical side of the module is Elizabeth Castro's “XML for the World Wide Web: Visual Quickstart Guide” [1]. This was chosen as it focusses on creating XML documents rather than more advanced applications. It covers the basics well, and as you progress through the module, will provide a handy reference for many of the tasks you'll be required to complete. It's available from amazon.co.uk [http://www.amazon.co.uk/exec/obidos/ASIN/0201710986/qid=1138789323] for around £10 - you may want to order collectively to save postage.

Chapter 1. Introducing XML

With this module, we will be constantly referring to standards and documentation produced by the World Wide Web Consortium [http://www.w3.org/], the W3C, who organise and shape the future of many web-based technologies, including XML and its friends. So, to begin, take a look at the W3C's summary of what XML is, their XML in 10 points [http://www.w3.org/XML/1999/XML-in-10-points].

The remainder of this chapter will look at these points, and look at some real examples of XML along the way.

What is XML?

Well, let's start by looking at a simple example.

Example 1.1. An example of XML looking suspiciously like HTML

```
<?xml version="1.0"?>
<html>
  <head>
    <title>My Title</title>
  </head>
  <body>
    <h1>My Title</h1>
    <p>The text of my document</p>
    <hr/>
    <p>Some more text</p>
  </body>
</html>
```

Hmmmm, looks familiar. Apart from the first line, it looks just like HTML, which we all know and love. That's because HTML can be written as an *application* of XML - this has come to be known as XHTML.

Here's another example:

Example 1.2. An example of a home-made markup language in XML

```
<?xml version="1.0" encoding="UTF-8"?>
<jokebook>
  <bookinfo>
    <title>My Best Jokes</title>
    <editor>Fred Bloggs</editor>
  </bookinfo>
  <joke>
    <simplejoke author="Traditional" rating="U">
      <question>Why did the chicken cross the road?</question>
      <punchline>To get to the other side!</punchline>
    </simplejoke>
  </joke>
</jokebook>
```

You'll notice here that, though the structure is similar, the tags (elements) have changed, and are more “meaningful”, in that they tell you about the specific role of each part of the document. So, `<punchline>` clearly indicates the (not very funny) response to the joke.

Also notice that there's very little about how the document is presented. The separation of *content* from *presentation* is an important theme in XML, and one we'll return to over and again.

Strictly XML

Or, why we need more rigorous markup

One of the problems with HTML is it's too laid back and relaxed - you can get away with almost anything. This is in part due to the browsers we have, which will always make the best of whatever is thrown their way, displaying on screen as much as they can decipher from the messy HTML they have grabbed from the webserver.

We, also, have played our part. It's difficult for humans to write in a mechanical, accurate manner. Think of how differently we all use English, how we flout the laws of spelling and grammar or use idioms or figures of speech - all very messy for the pedantic and rigidly-minded to understand. We get an impression of what is meant, but it may not be exactly what the speaker intended. This is a little like the use of HTML - because it is sloppy and loosely defined, we don't always get what we intended when the web browser displays it.

But XML is coming to the rescue! XML is well-defined and precise, and encodes accurately the structure and content of a document. It is easily read by machines, and fairly easily read by humans too. And it's easy to translate into more useful forms, such as HTML itself, printed pages, summaries, lists, etc.

Practical task

Create a new DocBook article using the <oXygen/> software, and use it to document any notes or ideas you have on the topics we discussed in today's class. Remember to create the document by choosing **`File > New from Templates...`**, and add some more sections, and some more paragraphs.

Experiment with the element drop-down list, which you can access by typing the first character of a new element (<) or by pressing **Ctrl**+**Space**. Read the descriptions of the elements - remember that what is important is not presentation but meaning. Choose the element that is closest in meaning to the intended text.

Add some of the following elements, working out how to structure them by following clues provided by the drop-down element list:

- `<emphasis>`
- `<quote>`
- `<itemizedlist>`
- `<indexterm>`
- `<code>`
- `<computeroutput>`
- `<programlisting>`
- `<example>`

Writing well-formed XML

As an example, let's look at a few of the differences between the HTML you're familiar with, and the XML version of HTML called XHTML. This XHTML follows very similar rule of syntax to conventional HTML, though the syntax is more strictly controlled. The main differences are:

- an XML version *must* be declared, e.g. `<?xml version="1.0"?>`
- a *root element* is required
  - this must enclose all other elements in the document;
  - anything outside the root element is considered to be a processing instruction;
- elements must always be properly closed
  - e.g. `<element></element>` or `<element/>` are both valid
  - but `<element><element2></element>` is not valid
- elements must be properly *nested*
- tags are always case-sensitive
  - and are conventionally lower-case throughout
- all attributes and values must be quoted
  - `<joke type="pun">` is valid XML
  - but `<joke type=pun>` is not
- all character entities (special characters) must be explicitly declared, except
  - ampersand: &
  - greater-than: >

  - less-than: <
  - quotes: "
  - apostrophe: '
- literal text containing any markup must be entered as CDATA, e.g. `<!CDATA[This is a piece of <i>tagged</i> text.]>`

This rigid adherence to the strict syntax rules means that, though it seems pedantic to us humans, the text requires much less work to parse in software, because the computer is not having to cope with mistakes and unexpected tagging.

The X Word: XML-related technologies and acronyms

One of the hardest parts of learning XML is remembering all the abbreviations for the bewildering array of tools and additions that associate themselves with XML. Here's a quick guide to the most important ones that we'll look at in this module.

XML, DTDs and Schema

XML is the core technology here, and almost every related technology is written in XML, and either extends its usefulness, or allows us to manipulate it in some way.

When we write new document types in XML, we say we are creating a new *application* of XML. We can be informal about how we create this application, making up new elements (tags) as we go, or we can specify which elements we want to use in detail first.

If we use the latter method, then we'll need to write a definition of our new application. This definition will not only create new elements for us to use, but will specify the structure that they will fit into, and the order we must use them in. This rigidity seems very restrictive at first, but we'll see later why this is necessary.

To write our definition we'll use either a *Document Type Definition* (a DTD) or an *XML Schema*. DTDs have been around for a very long time, and are rather clunky and inflexible - the specification for HTML is written in a series of DTDs. The format of a DTD can be very messy and unstructured, and is not in XML format.

XML Schema are a modernised version of the DTD. Written in XML, they are highly structured, and allow you to specify new applications very accurately. We'll look at both of these formats as we progress.

XPath

The first major addition to XML that we'll need to examine is *XPath*, which gives us a way to refer to and examine specific parts of an XML document. Using XPath, we can give a

precise identifier to an individual element within a document, or even a pattern or group of elements, useful when we want to examine only certain details recorded in an XML file.

Specifying a location in XPath is a little like writing down someone's postal address. We can write a full address, with everything from street number and name through to county and country, or in some contexts, we can just say “35 Juniper Avenue”, where it's clear that we mean a particular town or city.

XSL Transformations and Formatting Objects

When we want to convert our XML document into some other format, we have a couple of tools that fit the purpose, grouped under the banner of *XML Stylesheet Languages* (XSL).

XSL Transformations or XSLT are used very widely, and are especially useful for converting your XML documents into HTML (or rather, XHTML). The main focus of this module is on using XSLT effectively to create multipurpose documents, that can be viewed in a variety of formats and styles. XSLT is often built into web browsers, which makes using XSLT to create HTML a simple process.

Whereas XSLT is used to convert XML into other markup languages, its sister technology *XSL Formatting Objects* or XSL-FO is used to convert documents into more presentational formats, such as print documents.

XLink and XPointer

These are the linking mechanisms that allow us to insert the equivalent of the `<a href="url">text</a>` in HTML.

Generally, *XLink* gives us the ability to link to other locations on the web, and *XPointer* is used to link internally within a document.

SGML to XML, HTML to XHTML

XML has evolved from a previous generation of technologies, based around *Standard Generalised Markup Language* or SGML. XML can be considered the “little brother” of XML, in that it's more compact, easier to learn, and simpler to write applications and software for. We're not going to go into the technical details of SGML or the differences between SGML and XML, as they are mostly only of historical interest.

Chapter 2. Different Types of Document

This chapter will examine some of the applications of XML that are widely used in various communities today. Most have been designed to fulfill a single purpose; to encode a certain type of information, usually very rigidly constrained and predictable in nature. However, we'll start by looking at the most widely used, and least constrained of applications, XHTML.

Hypertext Markup Language

- General purpose markup language
- Originated as text-only, but quickly gained media extensions
- Often produced automatically by web authoring software

Example 2.1. A basic XHTML page with minimal presentational markup

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=iso-8859
    <title>Layout example</title>
    <link rel="stylesheet" type="text/css" href="default.css" />
  </head>
  <body>
    <div xml:id="header">
      <h1>The Title of the Site </h1>
    </div>
    <div xml:id="menu">
      <ul>
        <li><a href="overview.html">Overview</a></li>
        <li><a href="background.html">Background</a></li>
        <li><a href="filename.html">Category</a></li>
        <ul>
          <li><a href="filename.html">Subcategory</a>
          <li><a href="filename.html">Subcategory</a>
        </ul>
        <li><a href="filename.html">Category</a></li>
        <ul>
          <li><a href="filename.html">Subcategory</a></li>
          <li><a href="filename.html">Subcategory</a></li>
        </ul>
        <li><a href="filename.html">Category</a></li>
        <li><a href="ratings.html">Ratings</a></li>
      </ul>
    </div>
    <div xml:id="content">
      <h1>Section Title</h1>
      <img src="web.jpg" alt="A Spider's Web with its creator" />
      <p>A paragraph of text. A paragraph of text. A paragraph
         of text. A paragraph of text. A paragraph of text. A
         paragraph of text. A paragraph of text. </p>
      <h2>Subsection Title</h2>
      <p>A paragraph of text. A paragraph of text. A paragraph
         of text. A paragraph of text. A paragraph of text. A
         paragraph of text. A paragraph of text. </p>
    </div>
    <div xml:id="footer">
      <p>Student ID: 987654321</p>
    </div>
  </body>
</html>
```

Mathematical Markup Language (MathML)

- Special purpose: encoding mathematical equations and notation
- Designed to be embedded in other markup languages
- Rendered automatically by latest browsers

Specifications: http://www.w3.org/TR/2003/REC-MathML2-20031021/

Example 2.2. A very simple example of MathML

```
<math xmlns="http://www.w3.org/1998/Math/MathML">
  <msup>
    <msqrt>
      <mrow>
        <mi>a</mi>
        <mo>+</mo>
        <mi>b</mi>
      </mrow>
    </msqrt>
    <mn>27</mn>
  </msup>
  </math>
```

Source: http://www.w3.org/Math/XSL/pmathml2.xml [accessed 2007-02-13] (which can be used to check if your browser can render MathML embedded into XHTML).

RSS - Really Simple Syndication

- Special purpose: distribution of news items
- Simple syntax which handles text only
- Extended in several ways to encompass different media

Example 2.3. An instance of RSS 2.0 with iTunes and Yahoo media extensions

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet title="XSL_formatting" type="text/xsl"
    href="http://downloads.bbc.co.uk/rmhttp/downloadtrial/common/rss_rm.
<rss version="2.0" xmlns:itunes="http://www.itunes.com/dtds/podcast-1.0.
 xmlns:media="http://search.yahoo.com/mrss">
 <channel>
  <title>Digital Planet</title>
  <link>http://news.bbc.co.uk/1/hi/technology/1478157.stm</link>
  <description>Find out how the digital revolution is changing our lives
  <itunes:author>BBC World Service</itunes:author>
  <language>en-gb</language>
  <ttl>720</ttl>
  <itunes:image
   href="http://www.bbc.co.uk/radio/downloadtrial/images/programmes/300x
  <copyright>(C) BBC 2006</copyright>
  <pubDate>Tue, 13 Feb 2007 03:00:15 +0000</pubDate>
  <itunes:category text="Technology" />
  <itunes:keywords>technology digital science world</itunes:keywords>
  <itunes:explicit>No</itunes:explicit>
  <item>
   <title>BBC Digital Planet February 13 2007</title>
   <description>In this week's Digital Planet, ...</description>
   <itunes:subtitle>BBC Digital Planet February 13 2007</itunes:subtitle
   <itunes:summary>In this week's Digital Planet, ...</itunes:summary>

   <pubDate>Mon, 12 Feb 2007 16:00:00 +0000</pubDate>
   <itunes:duration>00:26:49</itunes:duration>
   <guid isPermaLink="false"
    >http://downloads.bbc.co.uk/rmhttp/downloadtrial/worldservice/digita
   <enclosure length="11339863" type="audio/mpeg"
    url="http://downloads.bbc.co.uk/rmhttp/downloadtrial/worldservice/di
   <media:content bitrate="40" duration="1609" expression="full"
    fileSize="11339863" type="audio/mpeg"
    url="http://downloads.bbc.co.uk/rmhttp/downloadtrial/worldservice/di

  </item>
 </channel>

</rss>
```

Source: http://downloads.bbc.co.uk/rmhttp/downloadtrial/worldservice/digitalplanet/rss.xml [accessed 2007-02-13].

Specifications: http://www.rssboard.org/rss-specification

DocBook

- Special purpose: computer documentation
- Very strictly specified and not usually extended
- Widely used in the publishing industry
- Highly customisable stylesheets to output to different media

For an example of DocBook, see the source XML [http://www.services.ex.ac.uk/cmit/modules/meaningful_markup/mit3112-notes.xml] for these notes.

Specifications:

- Version 5.0 (schema): http://www.docbook.org/specs/docbook-5.0b6-spec-wd-01.html [accessed 2007-02-12].
- Version 4.5 (DTD): http://www.docbook.org/specs/docbook-4.5-spec.html [accessed 2007-02-12].

The Text Encoding Initiative

- Special purpose: encoding of humanities texts
- Also used for creating websites and general documentation
- Very rich set of modular features to encode “messy” data
- Widely used as an archival format

Website: http://www.tei-c.org/

The TEI tagset is extremely feature-rich, and it is usual to customise the features available so as to limit the types of tags according to the nature of the document being encoded, e.g. poetry requires a different set of tags to prose or to performance texts. There are also features that allow the encoding of variations between manuscripts, uncertainty and omissions, unclear passages in audio transcriptions, etc.

Example 2.4. A very simple TEI P5 document

```
<?xml version="1.0" encoding="UTF-8"?>
<?oxygen
    RNGSchema="http://www.tei-c.org/release/xml/tei/custom/schema/relaxn
    type="xml"?>
<TEI
 xmlns:xi="http://www.w3.org/2001/XInclude"
 xmlns:svg="http://www.w3.org/2000/svg"
 xmlns:math="http://www.w3.org/1998/Math/MathML"
 xmlns="http://www.tei-c.org/ns/1.0">
 <teiHeader>
  <fileDesc>
   <titleStmt>
    <title>A Limerick</title>
   </titleStmt>
   <publicationStmt>
    <p>Privately published</p>
   </publicationStmt>
   <sourceDesc>
    <p>Uncertain source</p>
   </sourceDesc>
  </fileDesc>
 </teiHeader>
 <text>
  <body>
   <div type="limerick">
    <lg rhyme="aabba">
     <l>I went with the Duchess to <rhyme label="a">tea</rhyme>,</l>
     <l>Her manners were shocking to <rhyme label="a">see</rhyme>;</l>
     <l>Her rumblings abd<rhyme label="b">ominal</rhyme></l>
     <l>Were simply phen<rhyme label="b">omenal</rhyme></l>
     <l>And everyone thought it was <rhyme label="a">me</rhyme>.</l>
    </lg>
   </div>
  </body>
 </text>
</TEI>
```

Scalable Vector Graphics (SVG)

- Special purpose: vector graphics language

Example 2.5. The simplest of SVG examples

```
<?xml version="1.0" standalone="no"?>

<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">

<svg width="100%" height="100%" version="1.1"
xmlns="http://www.w3.org/2000/svg">

<circle cx="100" cy="50" r="40" stroke="black"
stroke-width="2" fill="red"/>

</svg>
```

Source: http://www.w3schools.com/svg/svg_example.asp [accessed 2007-02-13].

Declaring a DTD

You'll have noticed that in many of these formats, there are a few lines at the top of the document that contain URLs or other references, to standards sites and documents. These lines declare the standard being adhered to, and determine what 'validity' means for that document.

Like CSS stylesheets, DTDs can be defined within the XML document itself, or within a separate DTD file. Obviously, the latter is more flexible, since the DTD can be used easily for multiple files and is separated from the content of the document.

To declare an internal DTD, use something like this:

Example 2.6. Declaring a DTD internally

```
                    <?xml version="1.0" ?>
                    <!DOCTYPE jokebook [
                    ... DTD goes here ...
                    ]>
                    ... markup goes here ...
```

To create an external DTD, the definitions are simply placed in a separate file, usually with a `.dtd` extension, and the DTD is declared within the document:

Example 2.7. Declaring an external DTD

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE jokebook SYSTEM "jokebook.dtd">
... markup goes here ...
```

For most document types, especially those created for your personal use, this will be sufficient. If, however, your DTD is distributed to other authors and used widely, then you should consider "going public" and creating a Formal Public Identifier for your document type. This allows other authors to be certain that they're using the same DTD as everyone else, since FPIs are intended to uniquely identify a DTD.

An example of an FPI is that used by DocBook; here's the declaration for a DocBook 4.1.2 document:

```
<!DOCTYPE book PUBLIC "-//OASIS//DTD DocBook XML V4.1.2/
"http://www.oasis-open.org/docbook/xml/4.1.2/docbookx.dt
```

As you can see, it contains information about the creating organisation (or author), the version of the DTD, the language (EN), and the URL of the DTD itself. In most authoring systems, you'll need to download the DTD and install it before you can use it, even though you've declared where it can be found on the web.

XHTML has three different DTDs defined for it. A "strict" DTD forces very rigid rules on the document, which makes it more compatible with XML applications and processors. A "transitional" DTD creates more HTML-like relaxed documents, and is often used as an intermediate version when moving documents from HTML to XML. The "frameset" DTD defines documents which create frame-based layouts. The FPIs for these DTDs are:

```
<!DOCTYPE html
  PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!DOCTYPE html
  PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<!DOCTYPE html
  PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">
```

With XHTML, the declaration of the standard is important, as if it is not recognised, the browser will switch to a compatibility mode (known as “Quirks” mode, which will

render the document much less effectively. For a full discussion on this, see Eric Meyer's page on rendering modes [http://www.ericmeyeroncss.com/bonus/render-mode.html], and the material at quirksmode.org [http://www.quirksmode.org/css/quirksmode.html]. For any new websites, it's advisable to work with the Strict XHTML standard, as this will provide the best support for the latest CSS standards in most modern browsers.

Namespaces

An alternative method of specifying the standard in use is with a namespace. We've seen a couple of examples already:

```
<math xmlns="http://www.w3.org/1998/Math/MathML">
```

Here, the MathML standard is declared with a simple URL. These URLs don't have to point to any actual resource, as they are really only unique identifying strings, but it's usual for them to point to the standards definition or some related documentation.

```
<TEI
    xmlns:xi="http://www.w3.org/2001/XInclude"
    xmlns:svg="http://www.w3.org/2000/svg"
    xmlns:math="http://www.w3.org/1998/Math/MathML"
    xmlns="http://www.tei-c.org/ns/1.0">
```

This is a more complex example, where several different standards have been used in a document. The main body of the document is written to the TEI standard, which is declared as the default, with the `xmlns` attribute. Other standards are defined with prefixes, so that elements from those standards are distinguished from the defaults. As an example of this mixing of standards, take a look at this markup fragment:

```
<para>With XHTML, the declaration of the standard is important, as if
  recognised, the browser will switch to a compatibility mode (known a
  <quote>Quirks</quote> mode, which will render the document much less
  For a full discussion on this, see Eric Meyer's <link
    xlink:href="http://www.ericmeyeroncss.com/bonus/render-mode.html">
  rendering modes</link>, and the material at <link
    xlink:href="http://www.quirksmode.org/css/quirksmode.html"
  >quirksmode.org</link>. For any new websites, it's advisable to work
  XHTML standard, as this will provide the best support for the latest
  in most modern browsers.</para>
```

This illustrates a mixture of DocBook and XLink standards, where the attributes to the `link` element are defined by the XLink standard, not the DocBook standard. The namespace declaration for this document looks like this:

```
<book xmlns="http://docbook.org/ns/docbook"
```

```
        xmlns:xlink="http://www.w3.org/1999/xlink"
        xmlns:xi="http://www.w3.org/2001/XInclude"
        version="5.0">
```

Chapter 3. Defining Documents

Document Type Definitions

So far, we've looked primarily at existing document standards, such as XHTML and DocBook. These are defined through extensive discussion by groups of experts, and are extensively documented and described in various human-readable ways. However, their “canonical” definition lies within their Document Type Definition (DTD), a document (or more likely, a set of documents) that can be read by editors, validators and other software and used to determine whether any given document conforms to that standard.

So, the *DTD* is a formal specification of a particular document type; it defines what is *valid* for each *instance* of that document type. It defines the sequence of allowable elements and controls which entities and attributes may be used within each element. Most XML authoring software can read DTDs and can validate your documents by comparing them to your DTD. Some software will also use the DTD to restrict you to writing only valid markup as you author your documents.

For some uses of XML, there's no available standard covering the specific markup we need to store our data with. Perhaps we need to store very detailed data for a specific subject domain. Or perhaps we want to create documents that have a specific structure tied to their purpose. For these types of applications, we need to create a new definition, and write our own DTD.

DTDs are quite simple to write, but only allow very basic definitions of the elements and attributes that make up the application. For more detailed control over the structure of our instance documents, we can use more modern definition languages such as W3C Schemas or RelaxNG, both of which we'll cover in a later section. But as DTDs are easier to read and write, we'll start with them for our new XML application.

What is a DTD?

To begin with, let's have a look at the example DTD shown in Example 3.1, “A simple DTD for a jokebook”. This gives us a very simple document structure with some metadata (the bookinfo element) and a series of one or more jokes, all contained within the root node, called jokebook. The simplest document that can be created with this DTD is shown in Example 3.2, “A minimal jokebook document”.

Example 3.1. A simple DTD for a jokebook

```
<!ELEMENT jokebook (bookinfo, joke+)>
<!ELEMENT bookinfo (title, editor+)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT editor (#PCDATA)>
<!ELEMENT joke (simplejoke+)>
<!ELEMENT simplejoke (question, punchline)>
<!ELEMENT question (#PCDATA)>
<!ELEMENT punchline (#PCDATA)>
```

Example 3.2. A minimal jokebook document

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jokebook SYSTEM "jokebook.dtd">
<jokebook>
  <bookinfo>
    <title/>
    <editor/>
  </bookinfo>
  <joke>
    <simplejoke>
      <question/>
      <punchline/>
    </simplejoke>
  </joke>
</jokebook>
```

You should be able to roughly match the element names in the XML document with the element definitions in the DTD - in the next few sections we'll look at the exact syntax that allows us to define this document structure.

Defining the Document

The basic structure of a DTD comprises a sequence of definitions, of four types:

- Elements (or tags): `<!ELEMENT ....>`
- Attributes: `<!ATTLIST ....>`
- Entities: `<!ENTITY ....>`
- Notations: `<!NOTATION ....>`

The order of the definitions is generally not important, unless you're importing definitions from elsewhere (more on this another time).

The DTD doesn't follow the usual XML rules of well-formedness, since it is not an XML document itself. However, the syntax is strict and case-sensitive.

Defining Elements

As you've probably guessed, the element definitions provide the basic tagset to be used in the document. They also define the strict order and repetition of elements within the document.

A typical element definition may look like this:

```
<!ELEMENT simplejoke (question, punchline)>
```

This gives us the following markup:

```
<simplejoke>
  <question></question>
  <punchline></punchline>
</simplejoke>
```

This says that every <simplejoke> must contain *one and only one* <question>, followed by *one and only one* <punchline>. The simplejoke is the *parent* element, and question and punchline are its *children*.

Repetition+

Obviously, you are able to allow more than one of each element:

```
<!ELEMENT jokebook (joke+)>
```

which means that a jokebook may contain *one or more* jokes:

```
<jokebook>
  <joke></joke>
  ... more <joke> elements ...
</jokebook>
```

Similarly, elements may be marked as *optional* (`?`) or as appearing *none or more* times (`*`). You can also use the keyword `ANY` to allow any defined tag or text at that point (though this is rarely used).

Choices | choices | choices

Alternatives can also be marked:

```
<!ELEMENT joke (simplejoke | knockknockjoke |
    doctordoctorjoke | limerick)>
```

so that a joke may contain one (and only one) of several types. A valid pattern might be:

```
<joke><limerick></limerick></joke>
```

We can also allow multiple choices from a list *in any order*, by saying:

```
<!ELEMENT myjokes (simplejoke | knockknockjoke |
    doctordoctorjoke | limerick)+ >
```

which creates an *unordered list* of one or more jokes of the listed type, perhaps like this:

```
<myjokes>
  <knockknockjoke></knockknockjoke>
  <limerick></limerick>
  <limerick></limerick>
  <simplejoke></simplejoke>
  <limerick></limerick>
</myjokes>
```

Remember that, for these to validate, each of these individual element types will also need to be defined, even if only as #PCDATA.

Tricks and tips

Controlling and limiting multiple items is not always easy with DTDs, especially where there's a need to have a finite maximum or minimum that isn't zero or one. There are various tricks needed to work using these conventions, such as:

```
<!ELEMENT limerick (line, line, line, line, line)>
```

to define a strictly five-line verse, or

```
<!ELEMENT double-entendre (phrase, meaning, meaning+)>
```

to define it as a phrase followed by *two or more* meanings:

```
<double-entendre>
  <phrase></phrase>
  <meaning></meaning>
  <meaning></meaning>
  ... more <meaning>s if required ...
</double-entendre>
```

This method is quite a limitation if you wanted to define an element with, perhaps, twenty child elements. Both W3C Schemas and RelaxNG provide more effective ways of defining multiple child elements.

Adding text

So far, we've only defined tags which contain other tags, so we need a way of allowing tags to contain arbitrary text. We can do this using a #PCDATA clause:

```
<!ELEMENT phrase (#PCDATA)>
```

which allows any text *except tags* to be included as part of the phrase. Allowing a mixture of tags and text is trickier, and often confuses parsers and validators. You can mix #PCDATA with other elements in a choice list, but not in a sequence list, so:

```
<!ELEMENT line (#PCDATA | emphasis | rhyme)*>
```

is valid, and could give the following markup:

```
<line>There <emphasis>was</emphasis> a
  young lady from <rhyme>Crewe</rhyme>
</line>
```

but the next mixes #PCDATA within a sequence, so isn't a valid DTD definition:

```
<!ELEMENT double-entendre (phrase, meaning, meaning+, #PCDATA)>
```

In this case, you'll need to define another element which then contains the arbitrary text itself. Generally, though, mixing text and elements is frowned upon as poor document design, unless the document calls for “inline” markup (as, for example, the inline elements in XHTML such as <strong>, <a>, etc.).

Adding Attributes

Often it's preferable to define attributes to specify more detailed information about an element. In HTML, attributes were often used to modify the *presentation* of the element on the screen; in XML, the attributes should only be used to describe *content*.

Attributes are often used to provide *metadata*, information about the data contained in the element, such as its source, language or accuracy.

So we may want to specify in our XML document thus:

```
<joke author="Lee Evans" cert="18"> ...... </joke>
```

which we can define as:

```
<!ATTLIST joke author CDATA #IMPLIED>
```

The `ATTLIST` is followed by the element it applies to, then the name of the attribute it is defining. In this case it is an arbitrary text field (`CDATA` -- note no `#P`!), and is optional (`#IMPLIED`).

Explicit values can also be specified here:

```
<!ATTLIST joke cert (U | 12 | 15 | 18) #REQUIRED>
```

so that the author must choose one of the certificate values.

And a default value may be given:

```
<!ATTLIST joke author CDATA "anonymous">
```

where, if otherwise unspecified, the value will default to `"anonymous"`.

You can also specify a fixed value:

```
<!ATTLIST joke language CDATA #FIXED "English">
```

where the attribute will be defined as a specific constant whether or not it's encoded in the document.

Unique Identifiers

It's common to want to refer to a certain section of a document using some kind of identifier, so provision has been made for this using the special attribute 'ID'. This is reserved for use as a unique marker within each document (HTML has this feature, though it is little used). In particular, many database applications of XML will generate or read this attribute as a part of its key for the data.

```
<!ATTLIST joke code ID #REQUIRED>
```

You can also define an attribute to be a list of such IDs, as cross-references:

```
<!ATTLIST joke seealso IDREFS #IMPLIED>
```

Practical task

Develop a document type of your own. To begin, think of a type of document that has a simple structure, and that records the same information regularly within that structure. A simple example might be an address book - each entry follows a regular pattern and records the same type of information for each person.

Begin by marking up a “typical” document, creating the tags as you go. Then deconstruct the marked up document into a series of element definitions in a separate file. As you go, try creating a new document using your DTD in <oXygen/>, using the validation facility to test your DTD.

This works best with regular, well-structured documents, such as meeting minutes, restaurant menus, recipes, product specifications (e.g. cars, computers,...) and so on.

As an example, I've created a DTD [http://pallas.ex.ac.uk/pallas/teaching/mit3112/outlines/outlines.dtd] for the module outlines that we produce to define each module (e.g. this one [http://pallas.ex.ac.uk/pallas/teaching/mit3112/outlines/mit3112.xml]) that you take (see the outlines [http://pallas.ex.ac.uk/pallas/teaching/mit3112/outlines/] folder for other related files).

Good Document Design

Over the coming weeks, we'll build up a picture of how to analyse data and build a document design that models the data accurately and flexibly. For now, we need to bear in mind a few steps towards this process:

- identify your basic data items

- group related data items together
- organise these groups into a hierarchical (tree-like) structure
- examine the level of detail recorded for each data item - can it be broken down into further elements?
- determine whether inline markup is needed within text elements?
- look at the metadata (attributes) needed for each data item

Try to design for reusability; don't be restricted by presentation issues or by a specific output format for the document. Most XML documents have a life after their initial purpose, often unexpected, and planning for this should be part of your design. For example, if recording personal names, separating the first and surnames into separate data elements can allow more flexible processing, perhaps producing lists sorted by surname *or* firstname:

```
<personname>
  <firstname>Gary</firstname>
  <surname>Stringer</surname>
</personname>
```

Above all, your document design should be a *semantic* representation of the structured data contained within your documents.

We'll revisit this topic in more depth in the next session.

Entities and Other Useful(?) Stuff

Warning!

Most of the techniques described in the rest of this chapter are gradually being replaced with XML-related standards.

For example, the use of entities for accented characters is made redundant by the adoption of Unicode; notations are more commonly dealt with by embedding markup (via namespaces), etc.

What are Entities?

Entities are used for several reasons:

- to create a shorthand for entering often-used text;

- to define user-friendly names for special characters;
- to include material from an external file;
- to define material that should not be parsed.

Defining a text entity (an *internal general entity*) in the DTD:

```
<!ENTITY uoe "University of Exeter">
```

then using the entity in an XML document:

```
<p>The &uoe; is a very rainy place</p>
```

Inserting Accents and Symbols

Remember the encoding parameter in the XML declaration, which we mentioned way back in the introduction in Example 1.2, “An example of a home-made markup language in XML”?

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

An XML document can be written as a Unicode document, which allows the use of a vast array of multinational characters. Unicode should be used wherever possible, and should cope with most situations you come across. However, when dealing with legacy data, it's sometimes necessary to deal with other encodings.

In the past, it was more usual to write XML as plain ASCII, a standard format that almost all text editors use. Since ASCII doesn't have provision for multinational and symbol characters, we need a way of defining them, and inserting them easily into our text.

Special characters are already part of the XML standard; any Unicode character can be inserted using a character reference, which looks similar to an entity and uses the character's Unicode reference number, e.g. an e-acute is `é`

As you can see, this isn't exactly an easy-to-remember way of inserting special characters, so we usually define entities as more memorable references. In XHTML, for example, we can use the entity `é` which is defined as:

```
<!ENTITY eacute CDATA "&#233;" -- small e, acute accent -->
```

There's a list of character references for commonly used characters in Appendix C of Castro (2001), and there are numerous more complete lists on the web.

Exercises

Using the DTD and XML data documents you created for the previous exercise, try the following:

1. Add a few standard accented characters (e.g. umlauts or acute/grave accents) as entities to your DTD.
2. Insert a `lang="_____"` attribute to one of the elements in your DTD.
3. Add some multilingual text to your documents, with the relevant language attribute set.

   Hint: you'll also need to allow multiple instances of the tag containing the multilingual text.

External Entities

An *external entity* defines a binary chunk of data for later use. It's commonly employed for regularly used data such as graphical logos or icons that are used frequently within all documents of the type being defined; it's not normally used for one-off graphics such as diagrams, illustrations or other “content-related” items. Here's how it works: :

```
                            <!ENTITY unilogo SYSTEM logo-large.jpg NDA
<!ENTITY cmitlogo SYSTEM cmit-logo.gif NDATA gif>
```

Creating an attribute that refers to that data:

```
<!ELEMENT logo (alternatetext?)>
<!ATTLIST logo image ENTITY #REQUIRED>
<!ELEMENT alternatetext (#PCDATA)>
```

then referring to that picture in the XML file:

```
<logo image="cmitlogo">
  <alternatetext>Creative Media and
    Information Technology.</alternatetext>
</logo>
```

Including images as entities

In practice, this method of including an image is clumsy and very restrictive, since the location of the image file must be defined in the DTD. It's much more

usual to merely indicate a filename as a standard attribute and use a stylesheet or script to insert the image.

The entity method can, however, be useful to include very commonly used items such as a corporate logo, or regularly used icons, as part of a the text of the document.

Notations

When including unparsed content, the applications processing the data need to know something about the format of the data included, in order to process it correctly. For this we need to create a `<!NOTATION>` entry. So for example, we might have:

```
<!NOTATION jpg SYSTEM "image/jpeg">
<!NOTATION svg SYSTEM "image/svg-xml">
```

to allow us to use two different formats for graphical data. Note that the second, SVG, is also an XML document, though we don't want to parse it - it should be passed directly to the application.

The value given for each type of file is called a *MIME-type*, and is a standard code that most web browsers and data processing systems can use. There are numerous lists of MIME-types on the web; the most authoritative is at IANA [http://www.iana.org/assignments/media-types/], and a slightly friendlier list can be seen at W3Schools [http://www.w3schools.com/media/media_mimeref.asp].

Chapter 4. Cascading Stylesheets

When read by a browser, an XML document bears little relation to the HTML that it knows about by default, so we have to give it clues as to how it should display the elements we've created. We can do this in one of two ways, either giving it information on displaying existing tags, or translating the document into tags it knows about. To do the former, we use CSS, which should be a familiar technique from standard HTML.

A typical CSS stylesheet for an XML document might look like this:

Example 4.1. Simple CSS stylesheet for jokebook (excerpt)

```
jokebook   {display:block}

joke       {display:block}

simplejoke {display:block;
            border:thin inset blue}

question   {display:inline;
            font-size:12pt;
            font-weight:bold}

punchline  {display:inline;
            font-size:12pt;
            font-style:italic}
...
```

Each entry in the stylesheet consists of a selector (e.g. jokebook with one or more properties (e.g. display:block) attached to it.

Each element must be defined in the stylesheet with its presentational properties. For each element in your DTD, you must define at least the display property, which governs how the element behaves on the page. If you set display:block then the element will be shown as an independent block of content, on its own line (behaving in a similar way to <p>, <h1>, etc.). Setting it to display:inline will make it behave more like characters on the page.

Other properties should be vaguely familiar to you if you've used CSS to format HTML before -- the actual formatting properties are the same for HTML and XML, and if you're used to using CSS with HTML then defining stylesheets for XML is much the same.

You'll also note that class selectors (the .myclass notation in CSS, which selects a class="myclass" attribute in the XHTML) are not commonly used, as there may be no class attribute defined in the DTD for the XML document. However, the xml:id attribute is often used for more specific cases, and as the markup is more semantic with better defined elements serving more specific purposes, so the absence of the class notation is rarely a problem.

Of course, you can also use any attribute as a selector, so for the following markup:

```
<joke cert="18"> .... </joke>
```

the following rule will prevent the joke being displayed:

```
joke[cert="18"] {
    display: none;
}
```

Associating a CSS file with an XML document

In HTML, you may be used to linking to a CSS stylesheet using something like this:

```
<link rel="stylesheet" href="default.css" type="text/css">
```

or through Internet Explorer's @import URL() directive:

```
<style type="text/css">
  @import URL(http://mysite.org/default.css);
</style>
```

In XML, these are replaced by *processing instructions*, which link the stylesheet in a similar way:

```
<?xml-stylesheet href="default.css" type="text/css"?>
```

In XML, as in HTML, you can also define alternate stylesheets:

```
 <?xml-stylesheet alternate="yes" title="Large Print"
href="largeprint.css" type="text/css"?>

 <?xml-stylesheet alternate="yes" title="High Contrast"
href="hicontrast.css" type="text/css"?>

 <?xml-stylesheet alternate="yes" title="Print Only"
href="print.css" type="text/css" media="print"?>
```

If viewing the XML document through the Firefox browser, you can then choose between these stylesheets using the View → Page Style → Stylesheet-name commands. The

media="print" parameter means that the "Print Only" stylesheet will be used for printing by default.

Note that the last of the alternate stylesheets above adds a media attribute, which restricts the application of the stylesheet to a particular display type, such as "print", "screen", "aural", etc.[i] CSS itself has a method for restricting rules to certain display types, with the `@media` rule:

```
@media print {
  body { font-size: 10pt; }
}
@media screen {
  body { font-size: 14px; }
}
@media print,screen {
  body { line-height: 1.2; }
}
```

Using the Cascade

The C in CSS stands for “Cascade”, which describes a property of stylesheets that allows properties defined in one part of a stylesheet to override those defined elsewhere, or to override default properties. We can use the cascade to create useful effects within our stylesheets.

How the Cascade Works

- The rules for the CSS cascade can be summarised as:
  - Designer *overrides* User *overrides* Browser
  - More specific *overrides* less specific
  - ID selectors *override* classes *overrides* tags
- Order is also important - later rules override earlier

The rules of the Cascade mean that it is possible to write very general CSS rules which cover most elements, then redefine or override those rules for more specific cases. So, where there are generic text elements, you can set up a rule for the general case, then redefine properties for specific cases.

For more details on the Cascade, see the explanation in the CSS Guidelines [http://www.w3.org/TR/REC-CSS2/cascade.html#cascade].

[i]See http://www.w3.org/TR/CSS21/media.html for a full list of media types.

Firstly, the cascade prioritises the rules created by the website designer over and above any that are defined in the browser, either as defaults or those overridden by the user themselves (it's possible to customise the default behaviour of most browsers).

Secondly, more specific selectors will allow a rule to take precedence. So, a rule which selects only p elements will be overridden by a rule which selects all p elements which are children of a div, so if the CSS says:

```
div>p { color:blue; }
p { color:red; }
```

then the paragraphs directly inside a div will be coloured blue, and those not inside a div will be coloured red, even though the order might suggest otherwise.

Thirdly, there is a precendence of ID-based selectors over other types, so using a #name selector will usually take priority over any general redefinitions of elements.

However, the most important feature of the cascade is that later rules will override earlier ones (if of the same type).

It's quite common to define a set of defaults in a common stylesheet, then to override some of those rules with more specific stylesheets which are included later in the document. And sometimes, web pages override the site's CSS with inline CSS rules specific to individual pages. This occurs with the CMIT web pages, where inline CSS colours specific subdivisions of the menus, to give a contextual highlighting of the menu for the 'current' module.

Advanced CSS Selectors

Some of the more advanced *pseudo-classes* and *psuedo-elements* available in recent versions of CSS (CSS 2.1 [http://www.w3.org/TR/CSS21/] and the proposed CSS 3 [http://www.w3.org/Style/CSS/current-work/]) are extremely useful when styling XML directly with CSS. Though not available in all browsers, their use is gradually becoming more widespread. Here's a selection of the most useful:

Styling the first of a group of elements - `:first-child`. Applies a style to an element if it is the first child of its parent. This can be used to emphasise the first line of a poem, for example, or to provide different formatting for the first item of a list.

```
chapter { display: block }

chapter:first-child { color: blue }
```

Here, the first chapter in the list will be coloured blue [Example: XML [http://www.ex.ac.uk/cmit/modules/meaningful_markup/examples/css-first-child/book.xml] | CSS [http://www.ex.ac.uk/cmit/modules/meaningful_markup/examples/css-first-child/css-first-child.css]].

Styling the first of a group of lines or letters - `:first-line` and `:first-letter`.

```
p              { font-size: 12pt; line-height: 1.2 }
p:first-letter { font-size: 200%; float:left }
```

Adding text before/after an element - `:before` and `:after`. The following will add the word 'Warning' to any paragraphs with class="warning":

```
p.warning:before { content: "Warning: " }
```

and this rule will add a number before each h1 element, counting up with roman numerals:

```
h1:before { content: counter(chapno, upper-roman) ". " }
```

note here that the chapno is an identifier for the counter; any other numbering within the document would need to be identified by a different name. You can also reset the counters within rules, so that if numbering sections within a chapter, you could reset the numbers at each new chapter.

There are many other new rule types in the newer CSS (2.1) standards, which allow many useful effects to be created. However, there are still limitations to using CSS to style XML, most notably that it's often useful to re-order or filter the document before display. Whilst the newer CSS layout techniques can help with this, there is a better tool for the job in XSLT, which is purpose-designed for this role. We'll begin examining XSLT next week.

Browser compatibility

Note that many of the CSS 2.1 functions are only available in a few browsers - Firefox seems the most reliable for this at the moment - though eventually, they should be supported in all browsers.

For the purposes of the assignment, I'm quite happy for you to develop CSS which uses the newer features, rather than trying to produce effects that work across all browsers - but don't forget to mention in your report which browsers you used to test the CSS in!

Further reading...

The W3C have produced a guide to adding style to XML [http://www.w3.org/Style/styling-XML].

Chapter 5. Introducing XSL Transformations

CSS is a good way of laying out your pages in a well-designed manner directly from the XML document. And it provides a way for you to allow most browsers to view your XML without falling over or drawing a blank. But CSS is very limited in what it can do to the data before it is shown on screen. And it certainly can't manipulate the data in any real way, or rewrite it into other markup languages. That's where XSL steps in.

The XSL standard comprises two basic parts:

XSL Transformations (XSLT)
: These are widely used, well-defined and stable as a standard, and are used primarily to rewrite XML into other XML applications (such as XHTML or WML).

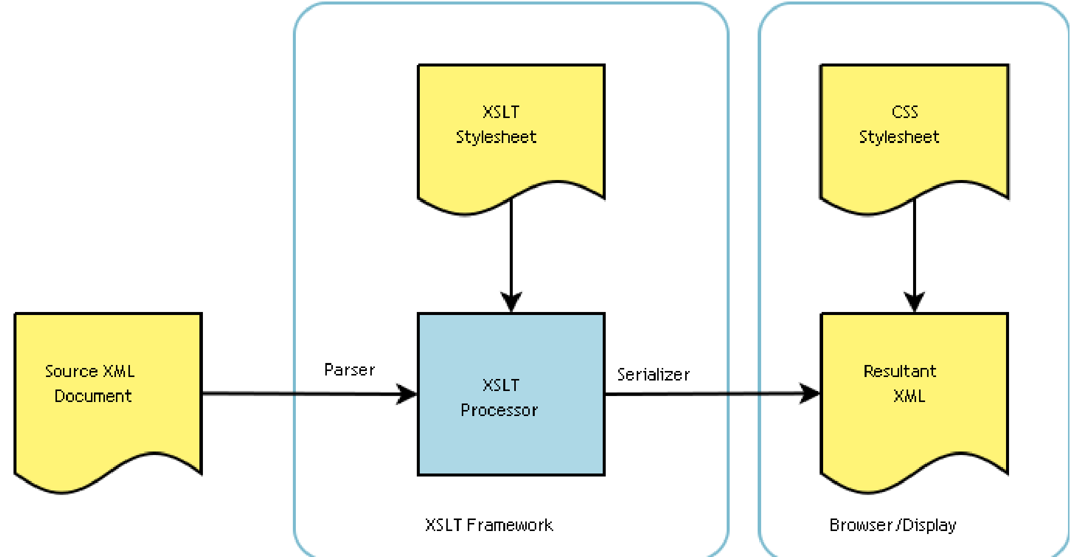
XSL Formatting Objects (XSL-FO)
: This standard is used to generate formatted output such as PDF files; they provide exact control of layout and presentation that isn't possible with either XSLT or CSS.

These two languages share many features, but are designed to achieve different tasks. XSLT is a general-purpose language for transforming one XML-based markup language into another, whereas XSL-FO is designed specifically for creating views of XML data in paged-media formats. Over the next couple of weeks, we're going to look at XSLT in depth, and maybe have a quick glance at XSL-FO if there's time.

A Simple Stylesheet

Let's begin by looking at our jokebook example to see why XSLT could be useful. One task that we might want to achieve is to produce an XHTML version of our jokes, for display on the web. XSLT is designed to this this job, as it can transform one XML-based markup language into any other.

Figure 5.1. The XSLT Processing Model



The XSLT stylesheet is applied using an XSLT processor (such as the ones built into the Oxygen editor), and converts documents from one XML application into another. CSS can still be applied to the resulting XML document.

To perform this transformation, we need to write stylesheets (though they are rather different from the CSS stylesheets we'e already examined). The stylesheet defines a set of templates that can be applied to the XML data document, and which contain rules to manipulate that data into the target format.

XSLT stylesheets are, of course, written as well-formed valid XML, so the basic syntax rules apply. They are usually a mixture of markup languages, including elements from the target language(s), together with statements from the XSLT language itself.

Here's a subset of a simple example:

Example 5.1. A simple XSLT stylesheet for jokebook (excerpt)

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/xsl">

  <xsl:template match="jokebook">
 <html>
   <head>
  <title>My Jokes</title>
   </head>
   <body>
  <h1>My Jokes</h1>
  <xsl:apply-templates />
   </body>
 </html>
  </xsl:template>

  ....

  <xsl:template match="double-entendre">
 <h3>Double-entendre:</h3>
 <p>The phrase
   <xsl:value-of select="phrase" />
   can mean:</p>
 <ol>
   <xsl:for-each select="meaning">
  <li>
    <xsl:value-of />
  </li>
   <xsl:for-each />
 </ol>

  </xsl:template>

  ....

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

</xsl:stylesheet>
```

As you can see, it's written as an XML document, which means it must be well-formed according to the XML rules. The repercussion of this is that any embedded markup in the

stylesheet must also be well-formed XML, which in turn means that our results will always be well-formed XML too. This can be difficult to work with at first, but will yield benefits in the quality of markup that is produced.

A More Detailed Look

The first thing to notice in the example is the mixture of XSL and XHTML markup. We're generating an XHTML page from an XML document, so the XHTML markup is inserted at the relevant output stage.

We begin with the declarations:

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

which define the verion of XML and XSL we're using. The xmlns:xsl clause is a 'namespace' declaration, and defines the standard we're using for the <xsl:_____> tags. We're using the convention of an xsl: prefix, but you could use another prefix such as x: or xslt:.

There are various locations used for this namespace declaration, the above is the current recommendation, though for some processors you may have to change this; for example, the built-in XSLT processor in some versions of IE6 require:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
```

The 'official' declaration, as recommended by the W3C, for an XSLT stylesheet outputting XHTML is:

```
 <xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns="http://www.w3.org/TR/xhtml1/strict">
```

Note that this also defines (with the default namespace) the version of XHTML that you're using (the strict version, in this case) within the second namespace (xmlns) declaration. Any markup which has no prefix will thus be XHTML Strict.

The body of the stylesheet consists mostly of <xsl:template> elements, which are designed to match the elements in our source document. So if we have a sequence of <joke> elements in our source document, we can use a <xsl:template match="joke"> element to process each of these with XSLT.

The last section, common to many XSLT 1.0 stylesheets, is the root template declaration:

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

This is really just saying that the templates defined in this stylesheet apply to the whole of the XML document tree. If you don't include it, most processors will use a default template which matches this definition, but it's useful to be able to override this default. Most version 1 stylesheets include the root template definition for completeness.

If we examine this more carefully, we can see that we're defining a new template within the <xsl:template> which matches a particular node or set of nodes in the document tree, in this case the topmost or root node. ("/"), which is the container for the entire XML document. We're then defining this template to merely look for and process other templates for the child nodes of the document.

XSLT Version 1.0 or 2.0?

On recent versions of the Oxygen editor, you'll be asked to choose between XSLT 1.0 and 2.0 when creating a new stylesheet. At this stage, I'd *strongly* recommend creating XSLT 1.0, as it's simpler and has fewer “gotchas” than 2.0, which will cause much head-scratching!

Once you're confident with 1.0, then you can take the step towards the newer standard. We'll look next week at some of the new features and complications introduced with XSLT 2.0.

Outputting Markup

In the stylesheet shown in Example 5.1, “A simple XSLT stylesheet for jokebook (excerpt)”, each template (i.e. within the <xsl:template> elements) could also include XHTML markup, since that's what we're outputting to our result document in this case.

The template for 'jokebook', which is the root element of our XML file in each case, is defined first.

```
  <xsl:template match="jokebook">
 <html>
   <head>
  <title>My Jokes</title>
   </head>
   <body>
  <h1>My Jokes</h1>
  <xsl:apply-templates />
```

```
  </body>
</html>
 </xsl:template>
```

Here, we're adding some structural XHTML markup which will form the basis of the XHTML result document, and in the midst of this asking the processor to apply any relevant templates for the contents of the <jokebook> tag, which is achieved with the <xsl:apply-templates / > element.

Skip down now to the <double-entendre> template:

```
 <xsl:template match="double-entendre">
<h3>Double-entendre:</h3>
<p>The phrase
  <xsl:value-of select="phrase" />
  can mean:</p>
<ol>
  <xsl:for-each select="meaning">
 <li><xsl:value-of /></li>
  <xsl:for-each />
</ol>
 </xsl:template>
```

This again outputs some HTML, but then outputs the entire contents of the <phrase> tag using a <xsl:value-of> statement. Note that this will output the whole of the contents of the element specified, so if it contains sub-elements, you'll get the values of all the #PCDATA elements with no surrounding tags splurged into the output.

You can also see an <xsl:for-each> statement which processes each of the <meaning> tags in the structure. We could, in fact, create a new template for <meaning> and do an <xsl:apply-templates>, which would work just as well here. However, sometimes we need to keep count of how many meanings we've processed (say we wanted to print the total number of meanings at the end), and using the <xsl:for-each> will allow us to do this.

The <xsl:value-of /> without a select attribute just means 'output the value of the current node', in this case, the whole of the currently processed <meaning> tag.

There are a number of other processing elements which can be used, together with a way of defining variables for temporarily storing values, which we'll look at in detail next time.

Hints and Gotchas

As the stylesheet must be well-formed XML, we have to use XHTML-style closing tags to all our HTML markup.

Though it's common to do so, it's not necessary to create a template for *every* element you've defined in your DTD.

If a template can't be found for a node in the document tree, then the default action is to apply-templates to its contents, or to output the contents if it's a #PCDATA node.

Order of definitions isn't important, except for your own sanity!

Tracking the current node

This illustrates an important concept in XSLT processing. The 'current node' is the element that we're working on at the moment, the context of the statement. The expressions use in referring to nodes (XPath expressions), like URLs in an <a> tag in HTML, can be relative or absolute, though the syntax differs slightly for XSLT.

As you might expect, you can begin an XPath expression with a / to start at the top of the tree, e.g. /jokebook/joke/simplejoke.

A '//' usually means 'all of...' in some way, so:

- "//" means "all descendents of the root node"
- ".//" means "all descendents of the current node"
- "//name" means "any descendents called 'name' in the whole document"
- "path//name" means "any descendents of 'path' called 'name'"

The @ symbol can be used to refer to an attribute of the current node, so @author is the value of the author attribute of the current node.

These expressions used in referring to parts of the document are part of a standard called 'XPath', which we'll cover in more detail later.

Chapter 6. Further XSLT and XPath

This lecture will look at some of the ways of generating loops, repetitions and choices within your stylesheets. Remember that, though these look similar to the control structures present in procedural programming languages, they are rather more limited in the way they behave. We'll also look at other XSLT elements that control such things as sorting, importing and the characteristics of the output.

Finally, we'll revisit XPath, looking at some of the tests, expressions and functions that can be used in conjunction with the above.

Control structures

There are a number of simple ways to make choices and perform repetitions within the XSLT language.

The <xsl:if> element

```
<xsl:if test="condition"> ...</xsl:if>
```

This element provides a simple `if ... then` control, which applies the elements it contains *if and only if* the condition is true. [i]

```
 <xsl:if test="@cert='18'">
<xsl:text>Warning:
         the following joke may be offensive</xsl:text>
 </xsl:if>
```

`<xsl:choose>`

The choose element makes a choice between one of several alternatives. A test is given for each defined choice, and there is a "catch-all" `<otherwise>` element which occurs if no other choice is selected. Equivalent to a `switch` or `case` structure in procedural languages.

```
 <xsl:choose>
   <xsl:when test="@type='limerick'">
     <xsl:text>A Limerick:</xsl:text>
   </xsl:when>
       ...
   <xsl:otherwise>
```

[i] Programmers should note that there's no `else` currently defined, so we'd need to use an `<xsl:choose>` element instead.

```
    <xsl:text>A generic joke:</xsl:text>
  </xsl:otherwise>
</xsl:choose>
```

<xsl:for-each>

Processes each of the child nodes in turn.

```
<xsl:for-each select="jokes">
  <xsl:value-of select="position()"/>
  <xsl:text>. </xsl:text>
  <xsl:apply-templates />
</xsl:for-each>
```

<xsl:sort>

Sort is used after a `<xsl:for-each>` or a `<xsl:apply-templates>` element to sort the child nodes that is selected by that element.

```
<xsl:for-each select="/jokebook/jokes">
  <xsl:sort select="joke/@type"  data-type="text"/>
    ...
</xsl:for-each>
```

Creating new elements - <xsl:element> and <xsl:attribute>

The Element and Attribute commands are used to output specific elements and attributes to the result document, and can be used to construct HTML or other tags using parts of the XML source document. So, for example, where a cartoon has a filename encoded for it and you wish to display this with an `<img>` tag in HTML, you could use:

```
<img>
  <xsl:attribute name="src">
    <xsl:value-of select="cartoon/filename" />
  </xsl:attribute>
</img>
```

Storing values - <xsl:variable> and <xsl:param>

Creates a variable. Note that variables in XSLT differ fundamentally from variables in procedural languages; they have more in common with constants or finals. For example, the variable with the most local scope overrides any others with the same name. So a variable created in the root element, and thus having 'global' scope, is overridden by any variables defined in child nodes with the same name.

```
<xsl:variable name="myFavourite" value="limerick"/>
    ...
<xsl:if select="$myFavourite='limerick'">
  <xsl:apply-templates/>
</xsl:if>
```

Parameters can be used in a similar way, but are effectively “default” values, which can be overridden when the stylesheet is applied.

To see parameters in action...

...try editing the DocBook parameters to change the type of output that's produced. You can change parameters of a particular stylesheet in <oXygen/> by editing the transformation scenario for the stylesheet.

Xml.com has a useful overview [http://www.xml.com/pub/a/2001/02/07/trxml9.html] of the differences between variables and parameters, and examples of how to use them. There are also excellent articles on this site which cover most areas of XML usage.

More XPath: Tests and operators

The basic test operators should be fairly familiar to you, they are similar to most programming languages. So:

```
=     !=     >     >=     <     <=
```

all take the usual (and hopefully obvious) meanings.

There's a slight difference in the basic arithmetic operators:

```
+    -    *    div    mod
```

in that the divided-by operator must be spelt out - the / symbol has far too much significance in other contexts to be reused.

Some Useful XPath Functions

There are also some useful pre-defined functions, the best of which are discussed below.

`position()`. Returns the current node's position in relation to its siblings, so the fourth `<joke>` in the `<jokebook>` will have `position()=4`

`last()`. Returns the number of nodes in the current context (i.e. the position() of the last sibling).

```
<xsl:choose>
  <xsl:when test="position()=last()">
    <li>And finally,
      <xsl:value-of select="."/></li>
  </xsl:when>
  <xsl:otherwise>
    <li><xsl:value-of select="."/></li>
  </xsl:otherwise>
</xsl:choose>
```

`sum(expr)`, `count(expr)`, `ceiling(expr)`, `floor(expr)`, `round(expr)`. Various mathematical functions - sum and count should be obvious; floor, ceiling and round are used to convert floating-point decimal numbers to integers (rounding up, down and mathematically)

`substring(expr,start,len)`. Returns a substring of the string *expr*, starting at character number *start*, with length of *len* characters.

This has (I hope) covered the most useful XSLT and XPath features, and should certainly be sufficient to perform most basic tasks. I'll be adding a few more features I feel would be useful to you over the next few weeks, so keep an eye on this lecture's notes. And don't forget, there are many more functions defined, which you can glean from the XML/XSLT/XPath standards documents.

Chapter 7. Using XSLT 2.0

XSLT 2.0 introduces a number of new and more powerful features to your stylesheets. The specification also adds an updated XPath 2.0, which introduces more complex queries and some procedural functions to the language.

The first change that is needed is to switch to an XSLT2-aware processor. In Oxygen 11, the default processor is Saxon 6.5.5, which will not parse XSLT2 stylesheets, so you should change it to SaxonPE or SaxonB. There are other options available, and often the choice of processor is dependent on the server technologies you're using - if you're developing an application for an MSWindows server, then a .NET-based processor may be needed.

Namespaces

XSLT2 and XPath2 are both namespace aware, and expect namespaces to be properly declared and prefixed throughout. Although it's possible to leave the default namespace undefined, it's usual to add it to make the output format clear.

As an example, here's the header for a stylesheet that converts TEI markup into XHTML:

Example 7.1. A namespace-aware XSLT2 stylesheet header

```
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:tei="http://www.tei-c.org/ns/1.0"
    xmlns:cite="http://citations.ex.ac.uk/ns/"
    xmlns="http://www.w3.org/1999/xhtml"
    xpath-default-namespace="http://www.tei-c.org/ns/1.0"
    exclude-result-prefixes="xs cite tei" version="2.0">
```

In this example, we're effectively declaring our *input language* as the TEI namespace - the **xpath-default-namespace** implies that most of our xpath statements will be referring to TEI elements. If this isn't included, you'll need to prefix all TEI elements within xpaths to identify them.

Our output language is declared by the default namespace (**xmlns** without suffix).

We're also excluding some of the intermediate prefixes from output, including one that we've defined locally (**xmlns:cite**). This prevents spurious **xmlns** attributes from cluttering up the root node of our resultant XHTML.

Sequences

In XSLT 1.0, most operations or templates resulted in the production of either single items of data, or *result tree fragments*, which had to be well formed XML (in particular, with a single root element).

In XSLT 2.0, there is a third possible result, a *sequence*, which is simple an ordered list of items (which may be single atomic values, or result tree fragments. This allows simpler sequential processing of arbitrary items of data, without having to construct a result tree that wraps them in a spurious root node element.

XPath enhancements

In general, XPath 2.0 introduces new pre-defined operators and functions that are merely more powerful versions of tasks that were possible in the older version. So the new **tokenize()** function performs a task (splitting a string into constituent parts) that needed a custom-written stylesheet and significant levels of recursion to perform in XSLT1.

However, some new features take XPath into a significantly more powerful language, adding basic flow control into the XPath itself. So we now have XPath expressions such as:

```
for $thing in SEQ return (EXPR)
```

which will perform the expression EXPR on each of the items in the sequence SEQ, and more powerful conditional expressions such as:

```
some $thing in SEQ satisfies (EXPR)
```

which will resolve as **true** if the expression EXPR is true for any of the items in the sequence SEQ.

As you can see, this moves some of the processing overhead into the XPath expressions, which allows for more efficient and compact stylesheets.

Functions

XSLT2 also introduces the named function, which can be defined to perform tasks which are not suited to the template structure.

Example 7.2. A basic function to lowercase text and remove (some) accents

```
<xsl:function name="cite:lower-remove-accents">
    <xsl:param name="input"/>
    <xsl:variable
      name="ac">àáâãçèéêëìíîïùúûü`´###</xsl:variable>
    <xsl:variable
      name="un">aaaaceeeeiiiiuuuu'''''</xsl:variable>
    <xsl:value-of
      select="translate(lower-case($input),$ac,$un)"/>
</xsl:function>
```

Here, we can see the parameter passed to the function being processed by a simple character substitution. The function is called like this:

```
<xsl:value-of select="cite:lower-remove-accents($string)"/>
```

Note also that we're using a defined namespace (**cite:**) taken from the header in Example 7.1, “A namespace-aware XSLT2 stylesheet header”. All functions must be tied to a namespace, even if it's a fictitious one such as this!

Frequently Asked Questions

The intention of this session is really to clear up any outstanding problems you may be having, especially with XSLT. Below are a few questions that I've been asked so far; if you have more then email me or ask me in the practical.

How do I process multiple XML documents

If you have a number of source documents that need to be summarised or collated, you can read the documents into the XSL process tree using the `document()` function. The most common way of doing this is by creating a master document, which details the files to be included in some way, e.g.

```
<master>
  <doc filename="file1.xml" />
  <doc filename="file2.xml" />
</master>
```

Each of these included files contains consistent data to be processed. The XSL transformation can then refer to the data within these like this:

```
<xsl:template match="/">
  <xsl:for-each select="/master/doc">
    <xsl:apply-templates select="document(@filename)" />
  </xsl:for-each>
</xsl:template>
```

As you can probably guess, this reads each file in turn, and applies its templates to the elements within the individual documents. Note that the argument to the `document()` function can be a URL in most implementations of XSLT. This method will also work in XSLT 1.0.

A more satisfactory solution in XSLT2 stylesheets is to use XInclude. This is a basic linking standard that allows inclusion of trees of XML markup without recourse to special handling within the stylesheet - most XSLT2 aware processors are able to transparently handle XIncludes. There's a good example included at the start of these notes, where a legal notice is included from a separate file:

Example 7.3. XInclude example

```
<xi:include href="../mit0000/legalnotice-by-nc-sa.xml"
            xmlns:xi="http://www.w3.org/2001/XInclude">
    <xi:fallback>
        <para>All rights reserved.</para>
     </xi:fallback>
</xi:include>
```

Chapter 8. Schemas of various types

Why use a schema?

As XML developed, the one standard that didn't evolve easily into an XML form is the DTD. This is perhaps because the DTD is at the heart of any XML application - everything else is built around it - and changing the way the document is defined would require major rewriting of almost all XML applications.

However, the bullet has been bitten, and the DTD has evolved (or continues to evolve) into the Schema. Schemas are XML-formatted, so are easy to machine-read and process, and do everything that the DTD can in specifying the structure and ordering of XML documents. But Schemas can go a lot further in the detailed specification of documents, right down to the individual data elements.

So which language should I use?

Confusingly, there are several different schema languages which can be used to describe XML document types. The two main contenders are the "official" XML Schema Description Language as defined by the W3C, and a language called *Relax NG*, which has both an XML-based format and a briefer, easier to learn and read "compact" format.

The course textbook goes into great detail on using XSD to create documents, so that's the language we'll look at here. However, there is strong support in parts of the publishing industry for the RelaxNG form, and it's too close to call which will win in the long term.

The main differences between the two lie in the philosophy of how to describe elements. XSD goes for the "describe everything in as much detail as possible" approach, whereas RelaxNG tries to simplify and create more readable definitions. Personally, I find RelaxNG easier to sketch out a document in, but for complex documents with very specific data to encode, XSD provides a better specification.

As the most complex standard, we'll look briefly here at XML Schema Descriptions, but the overall principles are similar. You can find more documentation on either language in the O'Reilly series (van der Vlist 2002, 2004), available to borrow from the CMIT office.

Using the W3C's Schema Description Language

XSD: Getting Started

Example 8.1. Starting a new schema

```
<?xml version="1.0" ?>
<xsd:schema xmlns: xsd="http://www.w3.org/2001/XMLSchema">
  ....
</xsd:schema>
```

Example 8.2. Using a schema to validate XML

```
<?xml version="1.0" ?>
<jokebook
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:noNamespaceSchemaLocation="jokebook.xsd">
    ....
</jokebook>
```

XSD: Documentation

Where a schema is to be used widely, it's useful to add annotations to the definitions, to provide clues to the user regarding the structure of the markup and the data it defines. This is particularly helpful when used with a schema-aware editor such as oXygen, which will show these annotations in context as the user creates a document.

Example 8.3. Adding documentation to a schema

```
<xsd:annotation>
  <xsd:documentation>
    Documentation text goes here.
  </xsd:documentation>
</xsd:annotation>
```

Annotations placed at the top of the schema before any type definitions apply to the schema as a whole; annotations place directly after an element definition will apply to that element only.

Once annotations are added, `<oXygen/>` should pick them up and display them as tool tips. For further examples, see the XSD version of the Jokebook listed on the examples page on the module website.

XSD: Simple types

Whereas a DTD would use the `#PCDATA` to define a generic “leaf node” to contain data, the schema can specify the type of that data very precisely.

The simplest use of this is with the standard built-in types, as defined in the datatypes [http://www.w3.org/TR/xmlschema-2/] section of the W3C specification.

Example 8.4. Some simple element definitions in XSD

```
<xsd:element name="author" type="xsd:string" />
<xsd:element name="pubdate" type="date" />
<xsd:element name="isPublished" type="boolean" />
```

XSD: Defining element structure

If we want to specify nodes or elements which are more complex than just a simple string, number, date or true/false value, the Schema language allows us to do this. For example, to define the actual structure of the document, the hierarchy of nested elements, we need to use the `<complexType>` construction.

Example 8.5. XSD: A simple element container

```
  <xs:element name="jokebook">
      <xs:complexType>
          <xs:sequence>
              <xs:element name="bookinfo" type="bookinfoType" />
              <xs:element name="joke" maxOccurs="unbounded" type="joke
          </xs:sequence>
      </xs:complexType>
  </xs:element>
```

This defines a complex element called jokebook, which contains a sequence of a bookinfo element, followed by one or more joke elements. Note that we define the repeated element using a maxOccurs="unbounded", which allows an unlimited number of repetitions. There is also a minOccurs attribute, which defaults to a value of "1", so we don't need to define it here.

We've also chosen to create these joke and bookinfo elements with user-defined types (bookinfoType and jokeType), which isn't necessary, but does make the definition more readable (otherwise the element structure would be a very deep XML hierarchy of definitions. Here's one of those types further specified:

Example 8.6. XSD: A user-defined complex type

```
<xs:complexType name="bookinfoType">
  <xs:sequence>
    <xs:element name="title" type="xs:string" />
    <xs:element minOccurs="0" name="editor" type="xs:string" />
    <xs:element minOccurs="0" name="pubdate" type="xs:date" />
  </xs:sequence>
</xs:complexType>
```

This user-defined type uses only simple types in its element definitions, but does make two of these elements optional (the minOccurs="0").

XSD: More powerful specification

One of the key advantages over the older DTD specifications of an XML application is the ability to define the data items more accurately. This can be as simple as restricting the length of a data item, or constraining it to numeric data only. If the data has a more complex structure, it's possible to define a pattern (or regular expression) that describes the data accurately. Here's an example:

Example 8.7. XSD: Constraining data with a pattern - postcodes

```
<xs:simpleType name="uk-postcode">
  <xs:restriction base="xs:string">
    <xs:pattern
      value="[A-Z]{1,2}[0-9R][0-9A-Z]? [0-9][A-Z-[CIKMOV]]{2}" />
  </xs:restriction>
</xs:simpleType>
```

Chapter 9. Making Connections

XPointer and XLink

As we have so often seen with HTML, the power of online documents is often due to the ability to link them together with hyperlinks. In HTML, we have a simple mechanism, the `<a>` element, which coupled with a URI in an `href` attribute, gives us a simple one-way link to another document or resource. In XML, we have two technologies that replicate this function. XLink acts as the `<a>` element, defining where the link appears and how it is used, and the XPointer standard gives us a method of accurately pinpointing the link destination to a document or a point within a document.

Adding simple links

The simplest type of XLink is very similar to the familiar hyperlink used in XHTML, though it's a little more verbose. In DocBook, a link might look like this:

```
<uri xmlns:xlink="http://www.w3.org/1999/xlink"
    xlink:type="simple"
    xlink:href="http://www.distributed.net/"
    >http://www.distributed.net/</uri>
```

Note that we're using a new namespace, xlink, which needs to be defined previously, in this case, in the uri element itself. It's more common (especially for documents with many links) to define the namespace in the root node for the document:

```
<book xmlns="http://docbook.org/ns/docbook" version="5.0"
  xmlns:xlink="http://www.w3.org/1999/xlink">
```

which allows us then to omit the xmlns:xlink declaration in each link element:

```
<uri xlink:href="http://www.exeter.ac.uk/">University</uri>
```

Here, we've also omitted the xlink:type="simple", which is generally assumed if not present.

Show and Actuate

In HTML, we had an additional link attribute, target, which allowed us to specify how the link should be displayed, so target="_blank" would give us a new window, or target="_parent" would open in the parent window, replacing whatever was loaded there.

In XLink, there is a similar attribute, show. This has the following values:

embed	Load the resource into the context of the linking element. In the examples above,

	an xlink:show="embed" would replace the <uri> element.
new	Load the resource into a new window, frame or pane.
replace	Load the resource into the current window, replacing the entire contents (default).
other	Behaviour is defined in other markup or scripts.
none	Behaviour is left for the browser to decide upon.

We can also decide when this action happens, with an actuate attribute. This is rather different to the standard HTML link, where a user action (clicking on the link) is the only way to actuate the link. Possible values include:

onLoad	Link should be actuated when the document is loaded; no user action is required to initiate the link in this case.
onRequest	Link should be actuated only when a user-initiated event occurs. This is normally when the user clicks on the link text, but may be triggered by other events.
other	Behaviour is defined in other markup or scripts.
none	Behaviour is left for the browser to decide upon.

Example 9.1. Some XLink/XHTML Equivalences

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

<a xlink:type="simple"
   xlink:href="http://www.ex.ac.uk/"
   xlink:show="replace"
   xlink:actuate="onRequest">Exeter</a>

<img src="uoe-logo.png" alt="The University Logo" />

<img xlink:type="simple"
     xlink:href="uoe-logo.png"
     xlink:title="The University Logo"
     xlink:show="embed"
     xlink:actuate="onLoad" />
```

Extended Links

The XLink specification provides much greater flexibility in creating links between documents than the simple XHTML-style model, though. Its concept of extended links gives the ability to link multiple documents and to provide links that are externally defined, rather than embedded within a document. In mathematical terms, XLink can define a directed graph, which can be internally or externally referenced. Here's an example:

Example 9.2. An image gallery as an extended XLink

```
<gallery xlink:type="extended" xlink:title="Image Gallery">
  <photo xlink:type="locator" xlink:href="sunset.png"
         xlink:label="sunset"
         xlink:title="A sunset over the bay" />
  <photo xlink:type="locator" xlink:href="beach.png"
         xlink:label="beach-01"
         xlink:title="A view south over the beach" />
  <photo xlink:type="locator" xlink:href="beachnw.png"
         xlink:label="beach-02"
         xlink:title="Looking northwest across the beach" />
</gallery>
```

Bibliography

[1] Elizabeth Castro. 2001. *XML for the World Wide Web*. Peachpit Press. Berkeley, CA. Companion website: http://www.cookwood.com/xml/. Publisher's website: http://www.peachpit.com/bookstore/product.asp?isbn=0201710986.

[2] World Wide Web Consortium. *Extensible Markup Language (XML)*. 1.0. . Third. W3C Recommendation 04 February 2004. http://www.w3.org/TR/2004/REC-xml-20040204/ . 2006-02-14.

[3] World Wide Web Consortium. *XML Linking Language (XLink)*. 1.0. . W3C Recommendation 27 June 2001. http://www.w3.org/TR/xlink/ . 2006-02-14.

[4] World Wide Web Consortium. *XML Path Language (XPath)*. 1.0. . W3C Recommendation 16 November 1999. http://www.w3.org/TR/xpath . 2006-02-14.

[5] World Wide Web Consortium. *XSL Transformations (XSLT)*. 1.0. W3C Recommendation 16 November 1999. http://www.w3.org/TR/xslt . 2006-02-14.

[6] Jeni Tennison. 2002. *Beginning XSLT*. Wrox Press. Birmingham, UK.

[7] Eric van der Vlist. 2002. *XML Schema: The W3C's Object-Oriented Descriptions for XML*. O'Reilly. Sebastopol, CA. [Copy available from CMIT office].

[8] Eric van der Vlist. 2004. *RELAX NG: A Simpler Schema Language for XML*. O'Reilly. Sebastopol, CA. [Copy available from CMIT office].

Glossary

Application
: Specifically, a language defined using XML. More generally, this language and its associated stylesheets, related documentation and any server programs that is uses.

Child element
: An element enclosed within another element. For example, in an XHTML document, the <html> element will always have exactly two children: <head> and <body>.

Document Type Definition
: A language used to describe document structures in *SGML*. Before XML Schema were standardised, DTDs were also widely used to specify the structure of XML documents.

Element
: The XML term for the basic unit of markup, often called a *tag* in other contexts.

Extensible Markup Language
: A rigorous and exacting syntax for markup languages.

Formal Public Identifier
: The unique reference to some kind of XML entity, usually a *DTD* or *schema*, which is cited in the DOCTYPE declaration. The FPI is normally presented as a URI, which may point to the location of the *DTD* on the web, but does not need to.

Instance
: For any XML *application*, an instance is any document that conforms to that application's DTD or Schema. So, if you write a valid DocBook document, then that is an instance of the DocBook application.

Metadata
: Data about data. Metadata in XHTML is included as <meta /> elements, which describe internally such information as the author, copyright, desired search engine behaviour, etc. In defining an application in XML, decisions must be made as to what information is real data, and what is metadata; it's usual to include metadata as attributes rather than individual elements. A common example is where an amount of money must be recorded, where the actual figure is the data, and the currency unit is metadata, e.g. <value units="GBP">200</value>.

Parent Element
: The element which encloses another element within a markup scheme. For example, the parent of the <body> element in an XHTML document is always the <html> element.

Result Tree
: The resulting tree of XML markup that's produced from a stylesheet in XSLT. In XSLT2, multiple result trees can be produced as a *sequence*, which must be *serialised* as they are written to the output. A *result tree fragment* can be created by a template, expression or function, which is an intermediate result not usually serialised.

Sequence	A ordered list of result tree fragments or atomic values. Sequences were introduced in XSLT2 to handle and process multiple results from expressions and templates .
Standard Generalised Markup Language	The predecessor to *XML*; bigger, bulkier and much harder to understand and manage. SGML evolved into XML through the pruning of many under-used features, and by enforcing more rigorous syntax on its applications. Versions of HTML up to HTML4 were defined using SGML.
XLink	An extension to XML which allows links between XML-encoded documents to be easily defined.
XML	See Extensible Markup Language.
XPath	A method of specifying locations or patterns of locations within an XML document.
XPointer	An extension to XML which allows internal links within XML documents to be defined easily.
XML Schema	Languages used to specify the structure of XML documents.
XML Stylesheet Language Formatting Objects	A language used to write templates to convert XML-compliant documents to presentational (usually print-based) media.
XML Stylesheet Language Transformations	A language used to write transformation templates which convert between XML-compliant markup languages.

Index

F

Formal Public Identifier, 14

X

XHTML
    differences between HTML and, 3-4