XML

1 Some History

See discussion of what is a markup language in the context of HTML. In particular, see discussion on SGML.

HTML defined a nice set of tags suitable for controlling display on a browser. However, as its use became widespread, various website builders started asking for new features that they could implement using HTML. Companies building web server software responded, and a barrage of new constructs were added to HTML, independently by each vendor. Some of the new ideas were bad, and died quickly. Many met a real need, and were adopted by competing vendors, either in their entirety or in some modified form. While this inventive ferment was great for technical progress, it totally destroyed the concept of a standard language: web site builders started having to say things like "best viewed with IE 4.0 or later."

If one thinks a bit about what it means to add new functionality in a mark up language, the primary means of doing this is through the definition of a new tag type. If one wishes to support and manage innovation in functionality, one must create mechanisms within the language itself to define new tag types. This is exactly what XML set out to do.

XML stands for eXtensible Markup Language, and was explicitly designed from the ground up with extensibility in mind. In fact, it took extensibility to the extreme, and pre-defined no tag types at all: every element tag had to be explicitly defined. Thus a <P> in XML does not necessarily indicate a paragraph beginning – it could just as well stand for a phrase, a price, or a document title – which it is depends on what it was defined to mean.

As it turns out, HTML settled down enough, after an initial burst of innovation, that it remained a viable language for describing web display and the additional flexibility of XML was not required. In consequence XML did not displace HTML.

However, this did not result in the death of XML. It turned out that the flexibility of a lightweight markup language that permitted arbitrary tags to be defined was great for representing information in a wide variety of applications. In fact, XML morphed from being a web language, to a universal representation for sharing data.

The importance of this universal representation cannot be emphasized

enough. Whenever two parties have to share information, they have to agree on how this information is to be represented. This agreement can come at several levels. Consider two humans trying to share some written information. A first step is to agree on the alphabet used to write in. But that is not enough – if I write in German and you only know French, it is the same alphabet, but there can be no sharing. A second step is to agree on the language. Let us say we have settled on English. Even that is not quite enough – if I give you a document written in "legalese," or a scientific paper replete with medical terms, you are likely to have difficulty with many of the terms and language constructs I use. Finally, even having the same vocabulary may not be enough – think of how many times you have had misunderstandings because some meaning was misconstrued by the reader.

In the same vein, there is a hierarchy of levels at which standards can be established for the interchange of information. Each step up in the hierarchy makes sharing that much easier. (However, it is too much to expect perfect sharing). To begin with, all modern computation is performed using a binary system with ones and zeroes. That much has been standard. Until twentyfive or so years ago, each computer manufacturer had their own way of representing characters as zeroes and ones. Moving data from one brand of machine to another would involve painstaking recoding of individual characters. ASCII (and its successor universal character representations, recall Sec. *) came along and established a standard at the level of characters. But data was still shared as "streams" of characters. XML provides a standard syntax for representing arbitrary data structures. Now, computers and programs can share data structures instead of sharing character strings. Think of how you write programs in your favorite programming language. Chances are that you spend considerable effort reading in a stream of input characters, parsing this stream and populating data structures before you get to doing anything useful in the program. In turn, the output is written out as a stream of characters. With XML, you can directly read in relevant data structures, perform the manipulations desired, and then write out data structures into XML.

Having a shared syntax for data structures still does not mean that there is perfect sharing of information. The next level up is the sharing of terminology and structural constructs. This turns out to be hard to do in a global way. However, the extensibility of XML has permitted communities of shared interest to define their own tag sets and schemas in XML to create their own markup language. Thus, we have Chem ML, BioML, StatML, MathML, ... Hundreds of languages that are easy to create and modify on top of XML, and serve as effective local standards.

2 XML Syntax and Structure

In HTML, many elements are bracketed by start and end tags. E.g <A href="next.htm"> element-content </A> However, not all elements are so bracketed. For instance, we frequently will start a paragraph with a <P> tag, but never bother to mark the end of the paragraph explicitly. It is assumed that the current paragraph ends when the next one begins, or we encounter the start of a new section or list. Observe that this implicit ending of a paragraph is possible because of our understanding of what a paragraph is, and what other tags mean relative to it. Consequently, we know not to end the paragraph when we encounter a font change tag, but we do end the is when we encounter a new section start.

In XML, we do not have any in-built tags with known semantics. Therefore, we have no way of distinguishing between a tag that causes change of font and one that begins a new section, and knowing that the latter should end a paragraph whereas the former should not. Instead, XML requires that each start tag have a matching end tag. For this to work properly, tags must nest – a tag that starts later must end earlier. Think of it like balancing parantheses.

An XML document is said to be *well-formed* if (i) it has a matching end tag for every start tag, and if this start-end pair is properly nested either completely included in, completely including, or completely non-overlapping with every other start-end tag pair, and (ii) it has a "root" tag pair enclosing the entire document. Note that well-formedness is a purely syntactic property – it says nothing about what the tags are or what they mean.

XML has been designed as a lightweight version of SGML. A key innovation in SGML was the idea of a *Document Type Definition*. The idea was that there would be a small number of well-known document types, and the structure of each could be stated in a DTD. XML adopted this idea of a DTD. Each XML document has a type specified in a DTD. This description could be included directly with the document itself, in a preamble; or it could merely be a reference to (the URL of) a DTD defined elsewhere. Think of this the way you treat variable declarations in software. Most of the time, you declare variables in separate header files that are then included into your source files. But occasionally you may have additional declarations to make in your file itself, e.g. for some local variables. Also, for small projects, you may choose to do everything in one file without pulling out the declarations into a separate include file. In a similar vein, one expects that in most situations, documents will use known DTDs from some agreed upon (within some community of interest) standard source. But occasionally, the

creator of an XML document may wish to define their own DTD.

An XML document is said to be *valid* if it follows the rules specified in its DTD. Note that an XML document must be well-formed before we can even begin to check its validity. Note also, that there can be well-formed XML documents that either do not specify a DTD at all, or ar invalid with respect to a specified DTD. For certain types of processing, where only the syntax matters, we may not care. For instance, a browser should still be able to display such a document, and we should still be able to evaluate Xpath or XQuery expressions against it. However, the meaning of such an invalid document may not be clear – so any semantic processing is unlikely.

3 Parsing XML

There are two quite distinct ways in which XML could be parsed: event-based, and object-based. An event-based parser is "low" level in that it responds to "events" in the input XML stream. The parser itself maintains very little state, and hence is very fast and requires little memory to run. The drawback is that using such a parser requires more work on the part of the programmer. Conversely, an object-based parser actually extracts a set of interconnected objects from the XML. The programmer now only needs to work with methods on these objects. However, constructing this set of objects can be quite expensive, and can use up huge amounts of memory. If all the extracted information will not be used, an object-based parse may represent wasted resources.

We discuss below the primary standards for these two types of XML parsing.

Given any well-formed XML, a parser should be able to parse it. However, we may often wish to make sure that it is valid as well. Doing so requires a *validating* parser. Parsers of both types can be rendered validating, simply by having them check the rules specified in the DTD. Typically, there is a flag that permits the same parser to be used in validating mode and in non-validating mode.

3.1 SAX

The Simple API for XML (SAX) defines an API for an event-based parser. Being event-based means that the parser reads an XML document from beginning to end, and each time it recognizes a syntax construction, it notifies the application that is running it. The SAX parser notifies the application by calling methods from the ContentHandler interface. For example,

when the parser comes to a less than symbol ("¡"), it calls the startElement method; when it comes to character data, it calls the characters method; when it comes to the less than symbol followed by a slash ("¡/"), it calls the endElement method, and so on. To illustrate, let's look at part of the example XML document from the first section and walk through what the parser does for each line. (For simplicity, calls to the method ignorableWhiteSpace are not included.)

```
<priceList>   [parser calls startElement]
  <coffee>    [parser calls startElement]
    <name>Mocha Java</name>    [parser calls startElement,
                characters, and endElement]
    <price>11.95</price>    [parser calls startElement,
                characters, and endElement]
  </coffee>    [parser calls endElement]
```

The default implementations of the methods that the parser calls do nothing, so you need to write a subclass implementing the appropriate methods to get the functionality you want. For example, suppose you want to get the price per pound for Mocha Java. You would write a class extending DefaultHandler (the default implementation of ContentHandler) in which you write your own implementations of the methods startElement and characters.

The following method definitions show one way to implement the methods characters and startElement so that they find the price for Mocha Java and print it out. Because of the way the SAX parser works, these two methods work together to look for the name element, the characters "Mocha Java", and the price element immediately following Mocha Java. These methods use three flags to keep track of which conditions have been met. Note that the SAX parser will have to invoke both methods more than once before the conditions for printing the price are met.

```
public void startElement(..., String elementName, ...){
  if(elementName.equals(``name''')){
    inName = true;
  } else if(elementName.equals(``price'') && inMochaJava ){
    inPrice = true;
    inName = false;
  }
}
```

```
public void characters(char [] buf, int offset, int len) {
  String s = new String(buf, offset, len);
  if (inName && s.equals(``Mocha Java'')) {
    inMochaJava = true;
    inName = false;
  } else if (inPrice) {
    System.out.println(``The price of Mocha Java is: `` + s);
    inMochaJava = false;
    inPrice = false;
    }
  }
}
```

Once the parser has come to the Mocha Java coffee element, here is the relevant state after the following method calls:

next invocation of startElement – inName is true next invocation of characters – inMochaJava is true next invocation of startElement – inPrice is true next invocation of characters – prints price

3.2 DOM

The Document Object Model (DOM), defined by the W3C DOM Working Group, is a set of interfaces for building an object representation, in the form of a tree, of a parsed XML document. Once you build the DOM, you can manipulate it with DOM methods such as insert and remove, just as you would manipulate any other tree data structure. Thus, unlike a SAX parser, a DOM parser allows random access to particular pieces of data in an XML document. Another difference is that with a SAX parser, you can only read an XML document, but with a DOM parser, you can build an object representation of the document and manipulate it in memory, adding a new element or deleting an existing one. DOM libraries are available for most popular programming languages. For example, you can use a DOM Java library to manipulate DOM objects as Java objects using standard methods defined in the library.

The DOM model represents an XML document as a tree. Each node in the tree corresponds to an element. The root of the tree is the element whose start and end tags include the full document. There is an edge in the tree corresponding to each inclusion relationship between elements. Whenever the start and end tags of an element c occur after the start tag of an element

p but before the end tag of p, then we say that p includes c and make the node p in the DOM tree a parent of child c.

The data model for an XML document is that of an ordered, labeled, tree. The tree is said to be labeled because each node in the tree has a label corresponding to the tag of the associated element. The tree is ordered because the order in which sibling nodes are placed beneath a parent is material. We actually have the notion of a first child, second child, etc. Something that would not be possible if it were an unordered tree.

Our primary focus thus far has been on XML elements. There are several other types of objects in the XML DOM, of which we mention one here: attributes. We are familiar with the notion of element and attribute from HTML. These ideas are carried over into XML. Specifically, we have zero or more attributes associated with each element. Syntactically, these are written as name=value pairs in the start element tag.

4 XML Design

XML provides great flexibility in structuring information – the syntax itself imposes no restrictions, and permits the complete individualization of each element occurrence. However, if any one is to use XML data, it is important to use this flexibility in a responsible way – there should be some sense to the structure, some pattern in which the information is represented. These structural patterns are captured in an XML DTD or schema definition. In this section, we consider some of the issues to keep in mind while creating such patterns.

In a relational table, each row represents a relationship, which could be rendered in an English sentence. Consider a table ORDERS with columns (partnum, supplierID, price, quantity). A row in it, with the tuple of values (123, ABC, 5, 10) can be read as "10 units of part 123 are ordered from supplier ABC at a price of 5 dollars each." (The astute reader will notice that the English sentence includes a great deal of semantics not present in the column names: that the price is in dollars, that it applies per unit and not to the whole order, and so on.

Now consider the same data in XML. We may have a supplier element with a part element below it, and price and quantity as sub-elements of part. There isn't a single unique tuple that is pulled out. However, any ancestor-descendant path in the graph should "make sense", in that it should be interpretable as an English sentence. Begin with single element "sentences", such as "There is a part." These obviously are OK. The ID of the part has

to be made an attribute of the element. Our one-element sentence then reads, "There is a part with ID 123." The maximum path length is now 3, and we can form a sentence that reads, "10 units of part 123 are ordered from supplier ABC," and another sentence that reads, "Part 123 is ordered from supplier ABC at a price of 5." Notice that the price is determined by part and supplier, and not by the order or order quantity. If we expect the price to be different for different orders, even from the same supplier, we may need to introduce an additional node order below supplier, and then make part, price and quantity all children of order.

The main point of the above example is to show that there is an issue with database design in the XML context. There are many different types of errors possible. A few common ones are listed below:

- Incorrectly promote element to attribute. An attribute at any level in the XML tree should apply to the entire sub-tree below it. If an attribute of an element x is irrelevant with respect to an element y, descendant of x, then it is likely that the attribute really should have been rendered as an element, child of x.

  In most XML implementations, attributes are dereferenced much more quickly than child sub-elements. So there is an efficiency reason to use attributes rather than sub-elements.

- Incorrect Use of attribute as element. Essential information about an element, of possible concern to its descendants, should be included in the element itself, as an attribute, rather than being pulled out in a sub-element. For example, the ID of a supplier should be part of the supplier element, and not in a separate supplID sub-element.

  Note that attributes must be single-values, and cannot have structure. These restrictions are significant, and can require that certain attributes be treated as sub-elements, even though they are logically attributes based on the argument above.

- Inadequately grouped data. Information that forms a single logical unit should be grouped together under a common parent element. E.g. if a supplier's address has a street, city, state, and zip, recorded as four different elements, these should be grouped together as children of an address element. In the relational world, such grouping is often not performed. E.g. in a single table, we would have these four fields and also supplier name, telephone number, year established, etc. Looking at the table structure, all these fields are co-equal without any additional structure among them.

- Promoting Data to Meta-Data. Since XML permits the schema to change from one part of the database to another, it is easy to fall into the trap of making "everything" into an element tag. Things that are data should remain data. E.g. it would be bad design to have fifty different tags, one for each state, rather than reord the state name as data.

- Demoting Meta-Data into Data. This is an error that is less likely in XML databases. However, it is very common in relational databases that have to represent semi-structured data. E.g. One could create a table with three columns, objectid, attribute name, and attribute value. With such a table you can represent anything you wish! But note that attribute name is a column that stores as values information that is better represented as real names of attributes.

5 XSLT

XML Stylesheet Language for Transformations (XSLT), defined by the W3C XSL Working Group, describes a language for transforming XML documents into other XML documents or into other formats. To perform the transformation, you usually need to supply a style sheet, which is written in the XML Stylesheet Language (XSL). The XSL style sheet specifies how the XML data will be displayed, and XSLT uses the formatting instructions in the style sheet to perform the transformation.

A common way to describe the transformation process is to say that XSLT transforms an XML source tree into an XML result tree.

XSLT Uses XPath XSLT uses XPath to define the matching patterns for transformations.

If you want to study XPath first, please read our XPath Tutorial.

How does it Work? In the transformation process, XSLT uses XPath to define parts of the source document that match one or more predefined templates. When a match is found, XSLT will transform the matching part of the source document into the result document. The parts of the source document that do not match a template will end up unmodified in the result document.

XSLT is a Web Standard XSLT became a W3C Recommendation 16. November 1999.

The root element that declares the document to be an XSL style sheet is <xsl:stylesheet> or <xsl:transform>.

Note: These two declarations are completely synonymous and either can be used!

The correct way to declare an XSL style sheet according to the W3C XSLT Recommendation is:

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

or:

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

Note: The xmlns:xsl="http://www.w3.org/1999/XSL/Transform" identifies the official W3C XSL recommendation namespace. If you use this namespace, you must also include the attribute version="1.0".

Start with your XML Document We want to transform the following XML document ("cdcatalog.xml") into XHTML:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<catalog>
  <cd>
    <title>Empire Burlesque</title>
    <artist>Bob Dylan</artist>
    <country>USA</country>
    <company>Columbia</company>
    <price>10.90</price>
    <year>1985</year>
  </cd>
</catalog>
```

Create an XSL Style Sheet Then you create an XSL Style Sheet ("cdcatalog.xsl") with a transformation template:

```
<?xml version="1.0" encoding="ISO-8859-1"?><xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"><xsl:template match="/">
  <html>
  <body>
    <h2>My CD Collection</h2>
```

```
    <table border="1">
    <tr bgcolor="9acd32">
      <th align="left">Title</th>
      <th align="left">Artist</th>
    </tr>
    <xsl:for-each select="catalog/cd">
    <tr>
      <td><xsl:value-of select="title"/></td>
      <td><xsl:value-of select="artist"/></td>
    </tr>
    </xsl:for-each>
    </table>
  </body>
  </html>
</xsl:template></xsl:stylesheet>
```

Link the XSL Style Sheet to the XML Document Finally, add an XSL Style Sheet reference to your XML document ("cdcatalog.xml"):

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="cdcatalog.xsl"?>
<catalog>
  <cd>
    <title>Empire Burlesque</title>
    <artist>Bob Dylan</artist>
    <country>USA</country>
    <company>Columbia</company>
    <price>10.90</price>
    <year>1985</year>
  </cd>
</catalog>
```

If you have an XSLT compliant browser it will nicely transform your XML into XHTML!

An XSL style sheet consists of a set of rules called templates.

Each <xsl:template> element contains rules to apply when a specified node is matched.

XSLT uses Templates The <xsl:template> element contains rules to apply when a specified node is matched.

The match attribute is used to associate the template with an XML element. The match attribute can also be used to define a template for a whole branch of the XML document (i.e. match="/" defines the whole document).

The following XSL Style Sheet contains a template to output the XML CD Catalog from the previous chapter:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"><xsl:template match="/">
 <html>
 <body>
   <h2>My CD Collection</h2>
   <table border="1">
     <tr bgcolor="9acd32">
       <th>Title</th>
       <th>Artist</th>
     </tr>
     <tr>
       <td>.</td>
       <td>.</td>
     </tr>
   </table>
 </body>
 </html>
</xsl:template></xsl:stylesheet>
```

Since the style sheet is an XML document itself, the document begins with an xml declaration: <?xml version="1.0" encoding="ISO-8859-1"?>.

The <xsl:stylesheet> tag defines the start of the style sheet.

The <xsl:template> tag defines the start of a template. The match="/" attribute associates (matches) the template to the root (/) of the XML source document.

The rest of the document contains the template itself, except for the last two lines that defines the end of the template and the end of the style sheet.

The result of the transformation will look (a little disappointing) like this:

My CD Collection
Title Artist
. .
w
The <xsl:value-of> element extracts the value of a selected node.

The <xsl:value-of> Element
The <xsl:value-of> element can be used to select the value of an XML element and add

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"><xsl:template match="/">
 <html>
 <body>
   <h2>My CD Collection</h2>
   <table border="1">
     <tr bgcolor="#9acd32">
       <th>Title</th>
       <th>Artist</th>
     </tr>
     <tr>
      <td><xsl:value-of select="catalog/cd/title"/></td>
      <td><xsl:value-of select="catalog/cd/artist"/></td>
     </tr>
   </table>
 </body>
 </html>
</xsl:template></xsl:stylesheet>
```

Note: The value of the required select attribute contains an XPath expression. It wor

---

The Result

The result of the transformation will look like this:

My CD Collection
Title Artist
Empire Burlesque Bob Dylan

The <xsl:for-each> element allows you to do looping in XSLT.

The <xsl:for-each> Element
The XSL <xsl:for-each> element can be used to select every XML element of a specified

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"><xsl:template match="/">
  <html>
  <body>
    <h2>My CD Collection</h2>
    <table border="1">
      <tr bgcolor="#9acd32">
        <th>Title</th>
        <th>Artist</th>
      </tr>
      <xsl:for-each select="catalog/cd">
      <tr>
        <td><xsl:value-of select="title"/></td>
        <td><xsl:value-of select="artist"/></td>
      </tr>
      </xsl:for-each>
    </table>
  </body>
  </html>
</xsl:template></xsl:stylesheet>
```

Note: The value of the required select attribute contains an XPath expression. It wor

Filtering the Output
We can filter the output from an XML file by adding a criterion to the select attribu

```
<xsl:for-each select="catalog/cd[artist='Bob Dylan']">
```

Legal filter operators are:

```
=  (equal)
!= (not equal)
&lt; less than
&gt; greater than
```

Take a look at the adjusted XSL style sheet:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"><xsl:template match="/">
 <html>
  <body>
  <h2>My CD Collection</h2>
  <table border="1">
   <tr bgcolor="#9acd32">
      <th>Title</th>
      <th>Artist</th>
   </tr>
   <xsl:for-each select="catalog/cd[artist='Bob Dylan']">
   <tr>
      <td><xsl:value-of select="title"/></td>
      <td><xsl:value-of select="artist"/></td>
   </tr>
   </xsl:for-each>
  </table>
 </body>
 </html>
</xsl:template></xsl:stylesheet>
```

The Result

The result of the transformation will look like this:

My CD Collection

```
Title Artist
Empire Burlesque Bob Dylan
The <xsl:apply-templates> element applies a template rule to the current element or t


--------------------------------------------------------------------------------

The <xsl:apply-templates> Element
The <xsl:apply-templates> element applies a template rule to the current element or t

If we add a select attribute to the <xsl:apply-templates> element it will process onl

Look at the following XSL style sheet:

<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"><xsl:template match="/">
<html>
<body>
<h2>My CD Collection</h2>
<xsl:apply-templates/>
</body>
</html>
</xsl:template><xsl:template match="cd">
<p>
<xsl:apply-templates select="title"/>
<xsl:apply-templates select="artist"/>
</p>
</xsl:template><xsl:template match="title">
Title: <span style="color:#ff0000">
<xsl:value-of select="."/></span>
<br />
</xsl:template><xsl:template match="artist">
Artist: <span style="color:#00ff00">
<xsl:value-of select="."/></span>
<br />
</xsl:template></xsl:stylesheet>


--------------------------------------------------------------------------------
```