Computer Science 75 Lecture 4: October 5, 2009
Fall 2009 Andrew Sellergren
Scribe Notes

Contents

1 Announcements (0:00–2:00)

- Tonight, we have the privilege of listening to a guest lecturer, David Heitmeyer, who teaches CSCI E-153, a entire course devoted to XML-based web development, as well as CSCI E-12, dedicated to the fundamentals of web development. David also works as a software engineer for iCommons. Thanks, David, for giving us a break from listening to the other David!

2 XML (2:00–110:00)

2.1 Introduction

- XML celebrated its tenth anniversary last year, which means that while it's not brand new and not the latest and greatest technology, it *is* mature and thus tried and true.
- XML is very similar to HTML, as we've already seen. Here's a few other properties and uses of XML:
  - "self-describing" data
  - interoperable (languages, platforms, applications)
    - e.g., between Java and Python or between PCs and Macs
  - content and data
    - can be used for both data-centric and narrative purposes, e.g. astronomical datasets and ancient manuscripts, respectively
  - machine-readable
  - presentation and display for people
- Let's take a look a simple example of XML which implements an address book:

```
<addressbook>
  <contact>
    <name>David Heitmeyer</name>
    <address>8 Story Street</address>
    <city>Cambridge</city>
    <state>Massachusetts</state>
    <zip>02138</zip>
    <phone type="office">617-384-6656</phone>
    <email>david_heitmeyer@harvard.edu"</email>
  </contact>
  <contact>
    <name>Drew Gilpin Faust</name>
    <address>Massachusetts Hall</address>
    <city>Cambridge</city>
    <state>Massachusetts</state>
    <zip>02138</zip>
    <email>president@harvard.edu</email>
  </contact>
</addressbook>
```

As you can see, `addressbook` is the single root node, which is a requirement for well-formedness. This XML can very easily be translated into XHTML so as to display more appropriately in a browser:

```
<table class="addressbook">
    <tr>
        <th>Name</th>
        <td>David Heitmeyer</td>
    </tr>
    <tr>
        <th>Address</th>
        <td>8 Story Street<br/>Cambridge, Massachusetts 02138
        </td>
    </tr>
    <tr>
        <th>Phone</th>
        <td>617-999-5870</td>
    </tr>
    <tr>
        <th>Email</th>
        <td><a href="mailto:david_heitmeyer@harvard.edu">david_heitmeyer@harvard.edu</a
    </tr>
    <tr>
        <th>Name</th>
        <td>Drew Gilpin Faust</td>
    </tr>
    <tr>
        <th>Address</th>
        <td>Massachusetts Hall<br/>Cambridge, Massachusetts 02138
        </td>
    </tr>
    <tr>
        <th>Email</th>
        <td><a href="mailto:president@harvard.edu">president@harvard.edu</a></td>
    </tr>
</table>
```

Although a trained programmer could easily extract the data he needs from this XHTML, it is much easier to parse in pure XML format.

- Recommended reading: XML in 10 points from the W3C. To emphasize 2 of these 10 points:

  - XML is a family of technologies.

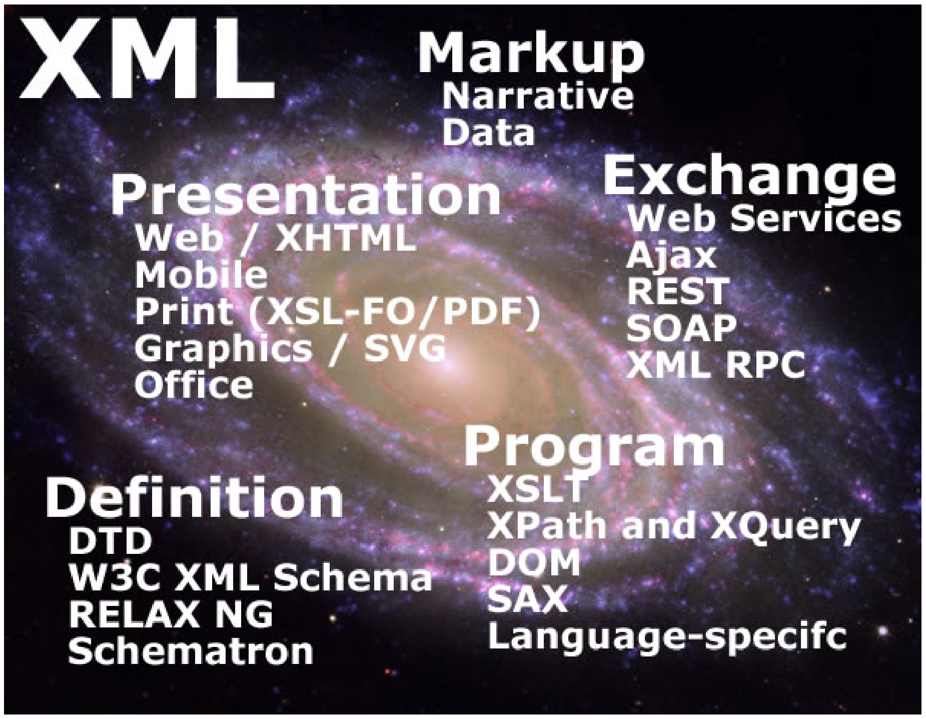
  This can be somewhat confusing for beginners in XML, but it will eventually be empowering as you become more comfortable with the language.

  - XML is new, but not that new.

  XML was actually born as a simplification of SGML (standard generalized markup language), which was developed in the mid-1980s.

- To quote W3C: "XML isn't always the best solution, but it is always worth considering."

- XML is more like a family or a galaxy of languages than a single one:



We've already discussed how XML can be used as Markup, either for narrative or data-centric purposes, as well as for Presentation, whether it be on the web or mobile devices or in print (extensible stylesheet language formatting object (XSL-FO)). Later in the course, we'll be talking more about using XML for Exchange, particularly in the context of Ajax. This Exchange can take place entirely server-side or might take place both client-side and server-side. We've already seen that it's possible to validate our webpages as XHTML, which means we've dealt with the Definition aspect of XML. DTDs and schemas, as well as RELAX NG and Schematron, are all used to describe what elements and attributes are available and how those elements can be nested. In terms of the Program side of XML, XSLT (emphasis on T for transformation), defines templates which are used to translate one XML format to another. In addition, XPath, a subset of XQuery, is used to quickly access information in XML.

2.2 Presentation

- XML can be used to present content in a number of different formats:
  - XHTML/HTML
  - PDF
  - Graphics
  - Mobile Devices (XHTML MP, WML)
  - Text
  - Office (Word, Excel, RTF)

  As an example, take a look at how a simple XML document describing a course can be transformed via XSLT into multiple different presentation formats. In each case, the presentation is being generated on the fly according to a template which dictates how the XML document is to be parsed. In this way, if any piece of information about the course changes, we need only change it in the source XML in order for all the different presentations to be updated. This way of describing a course in XML (although it could be improved), can also be extended to courses other than E-153.

- Weather data lends itself quite naturally to being represented in XML. The `my.harvard` Portal, for example, makes use of an XML feed from weather.com. The National Weather Service (NWS) also distributes live data in XML format. Feel free to play around with the Massachusetts data. Note that although this page looks highly stylized, it is still nothing more than basic XML. If you view its source, you'll see that the second line defines a stylesheet which the browser is using to display it. Even without a stylesheet specified, Firefox will display XML so that its elements are collapsible. While convenient, this feature is not built in to XML.

- Although we won't go into details, another interesting example of XML used for Presentation is maps. This map, depicting the precincts of Massachusetts and which party they voted for in the Presidential Election of 2008, was created using XSLT and XPath to merge data from an SVG map of Massachusetts and voting data from Boston.com. KML is a specific flavor of XML which is used to define placemarks and locations. We could pass a KML document to Google Earth or Google Maps to have it display markers at our points of interest.

- RSS is a flavor of XML used to syndicate news articles or other information updates. Users will access this feed using a reader which might be standalone or web-based. The core elements of RSS feeds are the `item` elements, which generally contain `title`, `description`, and `link` elements as children. Apple uses RSS to implement Podcasts. In the `enclosure` element (which is not Apple-specific), the actual media file (e.g. an MP3), is specified. Several other elements which are Apple-specific, including `itunes:summary`, `itunes:keywords`, and `itunes:duration`, extend the RSS format. Still, a normal RSS reader will be able to understand the Podcast. Apple's extension of the RSS format speaks to the modularity of XML.

- Question: XSLT would be able to process the Apple-specific elements so long as it was made aware of them at the outset.

- Using RSS, we could also depict information like David's Favorite Lunch Spots on Yahoo! Maps.

- MathML offers a quick and easy way of presenting equations and formulas in browsers. The less appealing alternative is TEX which is quite cumbersome.[1] In order to use MathML within HTML, we must declare our namespaces:

```
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:mml="http://www.w3.org/1998/Math/MathML">
```

  The first namespace declaration is standard for all HTML and is required for it to validate. The second namespace declaration is for MathML. This defines a prefix `mml`, which, when present at the start of any tag in the document, will refer to the MathML namespace. Note that `mml` is a prefix that we ourselves are specifying. We could've just as easily named it `math`, for example. It's not tied to the actual namespace.

2.3 History

- XML developed out of SGML, which displays tremendous flexibility. We see this flexibility in HTML, which is a specific manifestation of SGML. For example, some end tags are optional in HTML and attributes don't need to be quoted if they consist of a single word.

- This same flexibility, however, was also the undoing of SGML, given the difficulty of writing cross-platform parsers for it. XML, then, is a simplified, stricter subset of SGML. Because it is stricter, its parsers are simpler to write.

[1]Yes, I realize the irony of writing that within a LATEX document. (sigh)

- Question: what do you give up in using XML as opposed to SGML? Pretty much nothing. SGML DTDs allow for defining exclusions (e.g. a certain tag cannot be nested within another tag) whereas XML DTDs do not, but XML Schemas can accomplish the same thing.

- XML is not a markup language in and of itself, but rather a structure for creating markup languages, whether general ones like XHTML or more specific ones like MathML. For a good (but not comprehensive) list of markup languages that have evolved from XML during its history, see Oasis Open or XML.org.

2.4 Structure

- In order for parsers to work with XML documents, the documents must be well-formed (e.g. have open and close tags for all elements, including empty ones; use character entities to properly escape ampersands, etc.; have one root element; quote attributes; show proper nesting). XML documents may or may not be valid, however.

- XML documents can very effectively be visualized as trees. So in the example address book XML snippet we looked at earlier, `addressbook` is the root, within which we have two `contact` elements, which are children of `addressbook` and siblings of each other. Each `contact` element has `name`, `address`, `city`, `state`, `zip`, and `email` child elements, each of which is a descendant of `addressbook`. Conversely, `contact` is a parent and `addressbook` is an ancestor of `name`, `address`, etc.

- Question: can you have multiple `phone` elements if they have different `type` attributes? Actually, they don't even need to have different `type` attributes. We can have multiple `phone` elements within each `contact` and the document will still be well-formed. We might, however, choose to constrain that when we are defining our grammar.

- David recommends that you download an XML editor or otherwise enable XML syntax highlighting in your favorite text editor.

- XPath is a language which allows us to quickly access information in an XML document based on our knowledge of its structure. Recall this snippet of XPath from last lecture:

  ```
  /child::lectures/child::lecture[@number='0']
  ```

  From this snippet, we can make a guess at the structure of the XML:

  ```
  <lectures>
      <lecture number="0">
          ...
      </lecture>
  ```

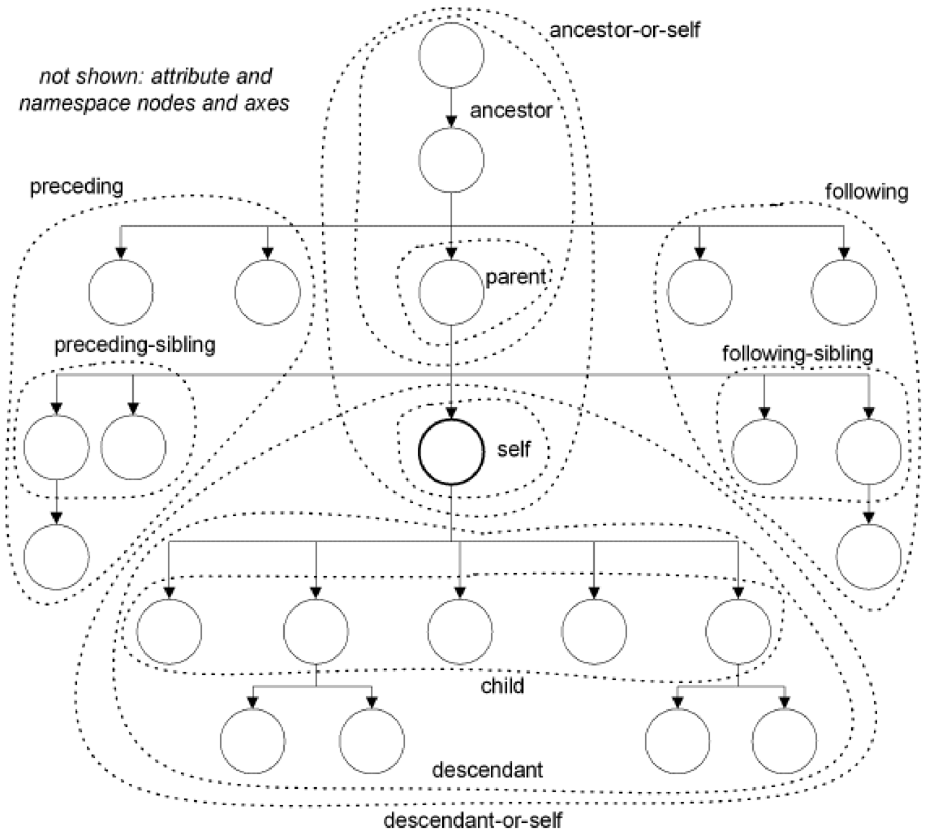
```
    <lecture number="1">
        ...
    </lecture>
</lectures>
```

The square brackets in the XPath expression comprise a *predicate* which specifies that we are interested only in the `lecture` element whose `number` attribute is 0.

- The following are all types of nodes that can appear in XML documents:
  - document node
  - element node
  - attribute node
  - text node
  - comment node
  - processing instruction node
  - namespace node

  The first four are much more common than the last three.

- Take a look at the following visualization of all the axes available via XPath:



In XPath expressions, axes are specified before two colons and are separated from each other by forward slashes. In the previous snippet, for example, we used the `child` axis twice. `child` is the default axis, so if no axis is specified, then `child` is assumed. We might've written the example above like so:

```
/lectures/lecture[@number='0']
```

Of the downward-looking axes, there is the verb/descendant/ axis, which includes immediate children as well as the children's children, and so on, as well as the `descendant-or-self` axis, which includes descendants and the self node. Of the upward-looking axes, there is the `parent` axis, the `ancestor` axis, and the `ancestor-or-self` axis. Other axes look left and right, so to speak, which attests to the fact that order *does* matter in XML documents (for elements, not attributes). Using the `preceding-sibling` and `following-sibling` axes, we can access siblings of the self node. Finally, the `preceding` axis, according to its informal definition, captures all elements which have closed before the self node and the `following` axis captures all elements which open after the self node.

- XPath expressions return sequences of nodes. You can think of them as lists, but know that lists are slightly different from sequences, at least conceptually.
- In predicates, when searching for specific values of attributes, be careful to enclose strings in single quotes, lest they be confused with parts of the XPath expression.
- Let's take a look at some examples of XPath expressions. The following two expressions are equivalent:

  ```
  /courses/course[@acad_year = 2009][@offered = 'Y']
  /courses/course[@acad_year = 2009 and @offered = 'Y']
  ```

  They both return the `course` elements which have an `acad_year` attribute value of 2009 and an `offered` attribute value of Y. In other words, all the courses which are offered in the 2009 academic year.
- The following two expressions select on the attributes of `role` elements, but ultimately returns a sequence of `person` elements, which are the parents of the `role` elements:

  ```
  /congress/person[role/@type='sen']
  /congress/person[role/@state='CA']
  ```

- We can use XPath to query RSS feeds as well:

  ```
  /rss/channel/item[position() = 1]
  /rss/channel/item[position() = last()
  /rss/channel/item[position() mod 2 = 0]
  /rss/channel/item[position() mod 2 = 0]/title
  /rss/channel/item[1]/title
  ```

  Unfortunately, although its confusing, XPath numbering is 1-indexed, so the first expression above selects the first `item` element. The second expression selects the last `item`. The third expression selects all the even `item` elements. The fourth expression illustrates that predicates don't have to appear on the end of XPath expressions. Finally, the fifth expression shows that `1` can be used as shorthand for `position() = 1`.
- Question: is it possible to find unique items within XML using XPath? This is actually very easy using XSLT and XPath 2.0, but not so easy using XPath 1.0. If you're using XPath alone and not XSLT along with XPath, you're better off searching for unique items programmatically using PHP.
- So long as a web page is written in XHTML, we can use XPath to traverse its DOM:

```
/html/head/title
/html/body//a[@href]
```

The first would select the `title` of the page and the second would select all the `a` elements that have `href` attributes—in other words, all the links. We should highlight an important but subtle syntactic difference between the following two XPath expressions:

```
/html/body//a[@href]
/html/body//a/@href
```

The first will select all the `a` elements that have `href` attributes. The second will select all the `href` attributes themselves. The predicates filter what we're selecting, they don't change what we're selecting.

- We might use XML to express classification of species by taxonomy. Take a look at this snippet which represents the family of even-toed ungulates:

```
<?xml version="1.0" encoding="UTF-8"?>
<taxonomy>
    <kingdom name="Animalia" common="animals">
        <phylum name="Acanthocephala"/>
        <phylum name="Chordata" common="cordates">
            <class name="Actinopterygii"/>
            <class name="Amphibia"/>
            <class name="Reptilia"/>
            <class name="Mammalia" common="mammals">
                <order name="Afrosoricida" />
                <order name="Artiodactyla" common="even-toed ungulates">
                    <family name="Antilocapridae">
                        <genus name="Antilocapra">
                            <species name="Antilocapra americana"/>
                        </genus>
                    </family>
                    <family name="Bovidae">
                        <genus name="Addax"/>
                        <genus name="Antilope"/>
                        <genus name="Bison"/>
                        <genus name="Gazella"/>
                        <!-- more not shown -->
                    </family>
                    <family name="Camelidae">
                        <genus name="Camelus" common="camels">
                            <species name="Camelus bactrianus"/>
                            <species name="Camelus dromedarius"/>
                        </genus>
```

```
                            <genus name="Lama">
                                <species name="Lama glama"/>
                                <species name="Lama guanicoe"/>
                                <species name="Lama pacos"/>
                            </genus>
                            <genus name="Vicugna">
                                <species name="Vicugna vicugna"/>
                            </genus>
                        </family>
                        <family name="Cervidae"/>
                        <family name="Giraffidae" common="giraffes">
                            <genus name="Giraffa">
                                <species name="Giraffa camelopardalis"/>
                            </genus>
                            <genus name="Okapia">
                                <species name="Okapia johnstoni"/>
                            </genus>
                        </family>
                        <family name="Hippopotamidae"/>
                        <family name="Moschidae"/>
                        <family name="Suidae"/>
                        <family name="Tayassuidae"/>
                        <family name="Tragulidae"/>
                    </order>
                    <order name="Carnivora" />
                    <order name="Cetacea" />
                    <order name="Chiroptera" />
                    <order name="Cingulata" />
                    <!-- more not shown -->
                </class>
                <!-- more not shown -->
            </phylum>
            <phylum name="Mollusca"/>
            <!-- more not shown -->
        </kingdom>
        <kingdom name="Archaea"/>
        <kingdom name="Bacteria"/>
        <kingdom name="Chromista"/>
        <kingdom name="Fungi"/>
        <kingdom name="Plantae"/>
        <kingdom name="Protozoa"/>
        <kingdom name="Viruses"/>
    </taxonomy>
```

Having represented this family in XML—or at least a small portion of it which includes camels, llamas, giraffes, and okapis—we can gather some interesting data using XPath. For example:

- For given species, find others in same genus
  * Find genus, then species
    `//genus[species/@name='Lama guanico']/species`
  * Find species with parent genus
    `//species[parent::genus/species/@name = 'Lama guanicoe']`
  * Find siblings and self
    ```
    //species[@name = 'Lama guanicoe']/preceding-sibling::species
    /taxonomy//species[@name = 'Lama guanicoe']
    /taxonomy//species[@name = 'Lama guanicoe']/following-sibling::species
    ```
- Tree pruned to a specific species

  `/taxonomy//*[descendant-or-self::species/@name = 'Giraffa cameloparadalis']`

In all of these examples, we end up flattening the hierarchy by querying it using XPath. If we wish to retain the hierarchical structure, we'll need to use XSLT.

• In order to more specifically define the structure of an XML document, we can use a DTD, an XML Schema, RELAX NG (RNG), or Schematron. DTDs, unfortunately, are not intuitive in the least, but they are necessary in certain contexts. More user-friendly are Schema and RNG, both of which are themselves expressed in XML, have data types (i.e. specifying that a certain node must be a date), and are namespace-aware. Schematron is useful in certain contexts because it is rules-based and expressed through XPath. For example, a site administrator might require that all pages on a site refer to a common CSS file. This would be difficult to accomplish in Schema and RNG, but is relatively simple to accomplish in Schematron.

• Why should we bother with definitions and validation? In some cases, they will make our XML documents more readable, both to machines and to people, and more easily editable (since the definitions can provide a guide to XML editors). In other cases, definitions and validations will not be worth it. For example, you may want to accept whatever data you can get from others, even if it's not perfectly formatted. Validation is also a costly process and is less of a concern if you trust your source (e.g. if you produce and consume your own XML).

• Some questions you should consider when you're designing a structure for an XML document:

  - How is XML created?
  - How is XML used?
  - What is the underlying data?

And some things you should avoid when you're designing a structure for an XML document:

- Unthinkingly tying to underlying model—it shouldn't necessarily perfectly reflect our database structure or object model
- Over-stuffing—we don't need an element for *everything*
- Pointer-heavy documents—better to leverage the hierarchy of XML

• Some recommended reading: Bad XML by Jeni Tennison.

• Some characteristics of good XML:

  - custom names
  - mixed content
  - nesting
  - attributes and elements
  - untyped data
  - namespaces

• When should data be stored as an attribute and when should it be stored as an element? And when should we mix the two? For very large XML documents which have tens of thousands of elements, storing as attributes is about 30% faster. For PizzaML, this won't really be a concern. Generally speaking, attributes represent metadata for the elements which represent the actual content. For example, the currency of a stock quote would probably best be stored as an attribute since it is metadata which describes the actual data, namely the stock quote. Lengthy text, as well as anything that relies on ordering or will itself have children, should probably be stored as elements.

• If we revisit the XML weather data provided by the NWS, we can begin to make some improvements. First, for dates and times, we should use ISO standard formats. Second, the timestamp for the observation is really metadata, so we could store it as an attribute rather than a child element of the root element. Third, we could switch to the `geo` namespace for describing locations and attach latitude and longitude coordinates to the `location` element either as attributes or child elements. Fourth, we could add more nesting: all of the elements with the prefix `wind_` could really be represented as children of a single `wind` element. Fifth, units for measurements could be changed to attributes. Regarding measurements that differ only in units, we might simply create an attribute named `system` that would specify English, metric, SI, or whatever system the units belong to. In that way, if we wanted to add temperature in Kelvin, we need not create an entirely new element named `temp_k`, but rather can add another instance of an element we've already created, namely `temperature`. Another option would be to nest measurements with different units as `reading` children of a single `temperature` parent.