# XPath and XSLT

Michael B. Spring
Department of Information Science and Telecommunications
University of Pittsburgh
spring@imap.pitt.edu
http://www.sis.pitt.edu/~spring

# Overview

- Context
- The Basics of XPath
  - Nodes
  - Axes
  - Expressions
- XPath and XSLT
  - Stylesheet templates
  - Transformations

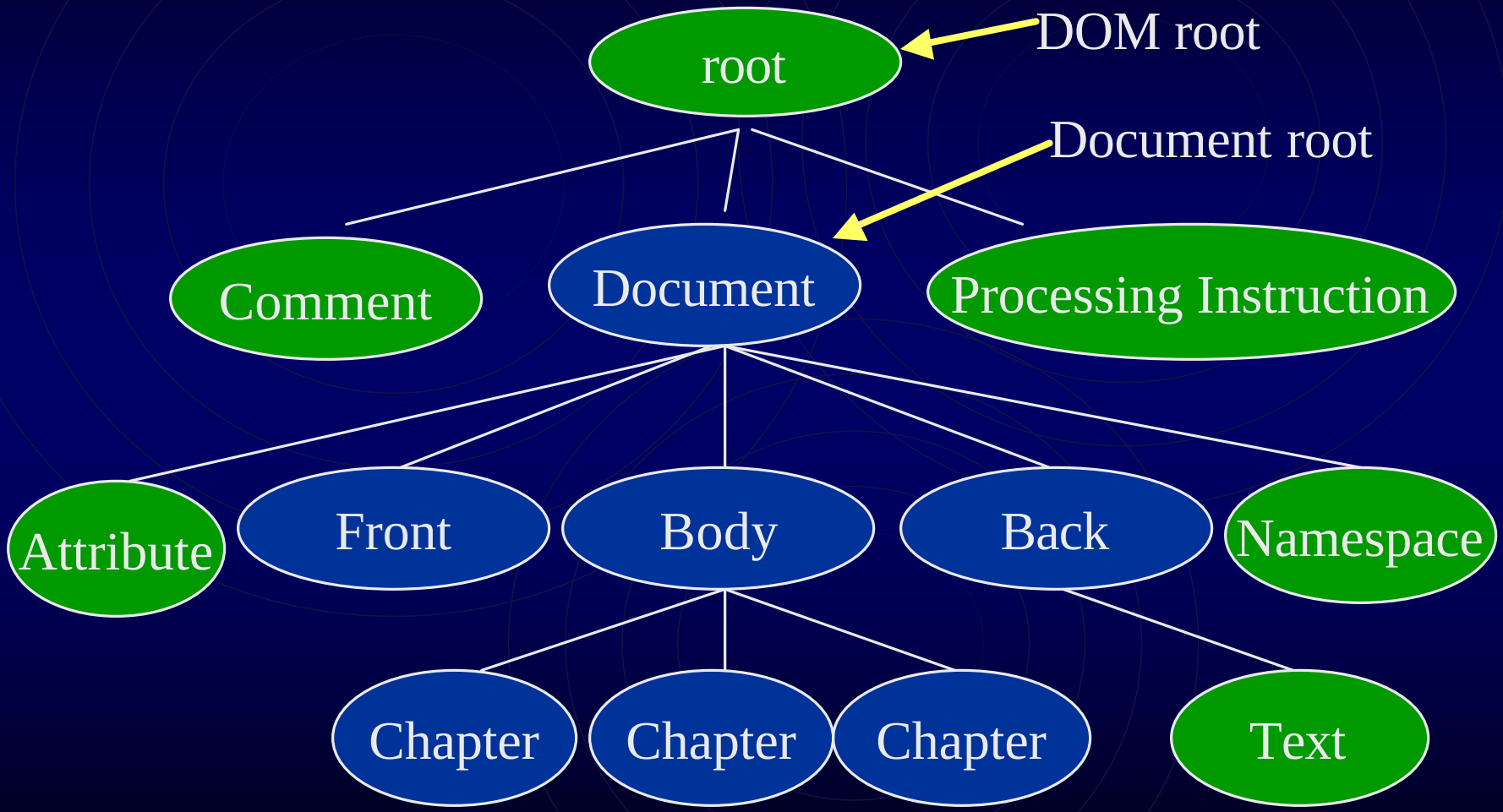
# Context

- An XML document is a directed graph or a tree.
- The XPath language allows a set of nodes of this tree to be identified
  - XPath allows for a number of special manipulations of the tree – these are called axes
  - XPath allows for a number of Node tests
- XPath is used by many other tools, but most notably by:
  - XSLT which is used to transform an XML document into another form
  - XPointer which is used in turn by XLink to identify a particular location within a document based on the tree structure.

# XPath

- XPath views a document as a tree of nodes, using the Document Object Model (DOM).
- The topmost part of the DOM tree is the root node
  - The root of the DOM is not the same as the document root element, but it contains the document root element which is abbreviated here as the document element.
- Nodes are of different types
  - Nodes of different types (e.g. element versus attribute) have logical restrictions on the children they may have.
  - For example, children of the root node may include comments, elements, and PI.
- XPath provides a variety of mechanisms that allows sets of nodes to be identified.

# XPath View of a Document



# XPath Terminology
# Nodes

- Nodes are the atomic entities in an XPath.
- Nodes may be of the following types:
  - Root
  - Element
  - Attribute
  - Text
  - Comment
  - Processing Instruction
  - Namespace
- Each type of node allows for specific children
- The currently “selected” node in an XPath is the context node.

# XPath Terminology
# Axes

- Axes represent the direction in which search of an XPath proceeds from the context node.
- There are thirteen Axes in XPath, with about five (*) that are used frequently:
  - self*
  - parent*, ancestor, ancestor-or-self
  - child*, descendant, descendant-or-self*
  - following, preceding
  - following-sibling, preceding sibling
  - attribute*
  - namespace

# XPath Terminology: Abbreviated Axes Names

- A document tree can be navigated using axes, but describing nodes using “descendant-or-self can be tedious. The more common axes have abbreviated terminology
  - the descendent-or-self axis is abbreviated as “//”
    - “//footnotes” would find footnotes anywhere in the tree
  - the parent axis is abbreviated “..”. “..” is the parent of the context node.
    - “//footnotes/..” would find the the parents of all footnotes
  - the attribute axis is abbrevated as @
  - The self node – the context node is abbreviated “.”

# XPath Expression

- an instance of an XPath is called an expression, or a location path
- A location path is a sequence of location steps – each step separated from the next by a “/”
- A location step is an axis specification followed by an optional node test (separated by “::”) followed by a predicate (enclosed in “[]”)
- when a system processes an expression, it builds a node set
  - The node set may then be processed by the application

# Node Tests and Functions

- There are a variety of node tests including:
  - node() – selects all nodes
  - text() – selects text nodes
  - comment() -- selects comment nodes
  - processing-instructions() – selects all processing instructions
- There are a variety of node set functions including
  - last() returns the last node of a set
  - count() returns the number of nodes in a set
  - id(string) returns the element node whose id matches string

# XPath Operators

- All of the operators, except one have already been seen:the simplest predicates test context
  - “/” is the XPath separator operator
  - “//” is the XPath abbreviation operator for all descendant children
  - “|” is the union operator which allows two sets of nodes to be combined

# XPath Predicates

- Predicates are used to test given node sets and are enclosed in “[]”s
  - /chap/para[7] selects the seventh para of the chap element which must be the document element
  - //chap/para[last()] selects the last para of all chap elements
  - //para[footnote] selects  para's that have footnotes
  - //para[footnote][@status] selects para's that have footnotes and and attribute called “status”
  - //chap[title = 'Introduction'] selects chapter(s) that have a title subelement whose value is “Introduction”
  - //chap/para[@type='ordered'] selects all paras of all chaps that have an attribute called type with a value of “ordered”

# Gathering a Node Set

- Path expressions may be relative or absolute
  - Keep in mind that in many applications using XPath, there will be a location in the tree – a context node, from which the path begins
- An absolute expression begins from the root node
  - “/book/front/author” identifies all the author nodes of the front element of the book element of the root node, if any
- A relative expression begins from the context node
  - “front/author” identifies all of the author nodes of the front element of the current node, if any

# Examples

- /Doc/Body/Chap
  - would select the Chap nodes in all the Body nodes of the Doc node which is THE Document element under the DOM root node. This could be written more formally as: “/child::Doc/child::Body/Chap”
- Chap//footnote
  - Would select all the footnote elements in all elements that are in the Chap elements of the current node
- //footnote/..
  - Would select the parent elements of all footnote elements anywhere in a document. This could be written more formally as:/descendant-or-self::node()/child::footnote/parent::node()

# More Examples

- /doc/(front | back)//text()
  - would select all of the text nodes in the front and back elements of the doc element which is the document root
- //comment()
  - Would select all comment nodes
- /book/chapter[1]/section[1]/para[3]
  - Would select the third para of the first section of the first chapter of the book element. More formally /child::book/child::chapter[position()=1]/child::section[position()==1/para[position()=3]

# Still More Examples

- And finally, two last examples:
- .//footnote/../title
  - Would select all of the title nodes that were children of all of the nodes that were parents of the footnote nodes that were descendants of the current node. More formally self::node()/self - descendant::footnote/parent::node()/child::title
- //chapter/section[position()=1 and position()=last()/title
  - Would select all of the title nodes of the first and last sections of all of the chapter nodes anywhere in the tree.

# The Uses of XPath

- Why all this functionality?
  - The basic answer is that XPath is the foundation on which all tree manipulation is done.
- The most fundamental tree manipulator is XSLT. It is used to select and filter sets of nodes from an input document creating an output document.
  - XSLT is a very complex standard, and we will only overview it here.
  - XSLT is also used in conjunction with formatting objects which are described separately
- XPointer and XLink also use XPath and are described separately.

# XSLT

- XSLT defines a class of applications that provide for the transformation of documents using XPath
- While XSLT is a part of the “XSL” standard, and can be used to “style” documents, it is different enough that it is dealt with separately
- We can match, filter, and select nodes in sets that are returned by XPath
- XSLT has many function.  Two main functions are:
  - As a tool to convert XML documents to html documents
  - As a tool to convert a base XML document to multiple forms

# How XSLT is Used

- XSLT is used to transform documents. It accomplishes this by attaching a “style sheet” to the document.

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="my.xsl"?>
```

- The style sheet is an XML document whose elements are come from the xsl namespace.

```
<xsl:stylesheet xmlns:xsl=http://www.w3.org/Transform/1.0>
.... templates...
</xsl:stylesheet>
```

- The templates define how given elements are transformed

# XSLT Processors

- Formally, a style sheet should begin to convert XML to HTML should begin:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl=http://www.w3.org/Transform/1.0
    xmlns:html=http://www.w3.org/TR/REC-html40
```

- Each processor, non of which is yet fully conforming requires a different start:
  - IE5 accepts XML documents that begin:

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

  - While James Clark's XT accepts documents that begin:

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

# XSLT Elements Overview

- Stylesheet structuring
  - <xsl:stylesheet>
  - <xsl:include>
  - <xsl:import>
- Template structuring
  - <xsl:template>
  - <xsl:apply-template>
  - <xsl:call-template>
- Conditional processing
  - <xsl:if>
  - <xsl:choose>
  - <xsl:when>
  - <xsl:otherwise>
  - <xsl:for-each>
- Generating output
  - <xsl:value-of>
  - <xsl:element>
  - <xsl:attribute>
  - <xsl:comment>
  - <xsl:processing-instruction>
  - <xsl:text>
- Variables and parameters
  - <xsl:variable>
  - <xsl:param>
  - <xsl:with-param>
- Sorting and numbering
  - <xsl:sort>
  - <xsl:number>

# XSLT Elements, Simplified

- A basic XSLT style sheet consists of:
  - The main element -- <xsl:stylesheet>
    - Transformation elements -- <xsl:template>
      - Access elements -- <xsl:value-of>
      - Further processing -- <xsl:apply-templates>
      - Loops -- <xsl:for-each>
      - Sorting -- <xsl:sort>
      - Testing -- <xsl:if>
      - Numbering -- <xsl:number>
    - Other high level elements for dealing with whitespace and combining stylesheets
- Each of the transformation elements uses XPath in some form to do its work

# The Basic Idea

- An input tree is processed recursively through the use of templates.
- Each template, starting with the root element is processed producing a set of nodes.
  - The internal instructions in the template describe how the set is processed by other templates in the stylesheet.
  - Ideally, there is a template for each element of the input document that takes the text and wraps it with the appropriate tags for the output document
- Keep in mind that for template processing, only element, text, comment and processing instruction nodes are processed. Attribute and namespace nodes are not passed on

# A Simple Example

- An XML document “programs” is made up of multiple subelements (foo and bar.)
- The style sheet that has two templates as follows:

```
<xsl:template match=“/program">
    <ProgramListing>
    <xsl:apply-templates />
    </ProgramListing>
/xsl:template>
<xsl:template match=“foo">
    <Module>
    <xsl:value-of select = “.” />
    </Module>
</xsl:template>
```

- This style sheet processes the “program” element outputting new tags and calling apply-templates. The processor finds foo and bar elements. Because there is no template for “bar”, these elements are ignored. “foos” are output as “Modules”

# Anatomy of a Template

- Within a template, the simplest set of rules would be something along the following lines:

```
literal text to be output
<xsl:value-of select=" ."/>
<xsl:apply-templates />
literal text to be output
```

- If we were converting a “book” to “html”, we might use something like this at the top level

```
<xsl:template match = “/book”>
<HTML><HEAD>
<TITLE><xsl:value-of select=“title/text()”/></TITLE>
</HEAD><BODY>
<H1><xsl:value-of select=“title/text()”/></H1>
<xsl:apply-templates />
</BODY></HTML>
</xsl:template>
```

# Changing Order

- Imagine a book that has a front, middle, and end. The following template would reverse the order of these parts in the output:

```
<xsl:template match = “/book”>
<xsl:apply-templates select = “end”/>
<xsl:apply-templates select = “middle”/>
<xsl:apply-templates select = “front”/>
</xsl:template>
```

- Of course, templates would be needed to handle the nodes generated by each of these selects

# Conditional Processing

- A template can do conditional processing.
- Consider a “program” made up of “modules” made up of “lines” of code.
- One way to mark the beginning and end of the modules would be in the module template. But the line template could also handle the task:

```
<xsl:template match = “line”>
<xsl:if test=“position()=first()">
/* Start of Module */
</xsl:if>
<xsl:value-of select = “.”/>
<xsl:if test=“position()=last()">
/* End of Module */
</xsl:if>
</xsl:template>
```

# For-each

- Normally, a template would hand off processing of subelements to other templates. But if might be logically more clear to work within a template. Consider a pricelist, in which has a series of items each of which has an number, description, and price. We want to build a table which is easier to see in a single template.
- The second example shows how the same for-each might make use of a sort on the price of the items.

# Example of a for-each

```
<xsl:template match=“pricelist">
<table>
<xsl:for-each select=“item"/>
<tr>
<td><xsl:value-of select=“./number”></td>
<td><xsl:value-of select=“./description”></td>
<td><xsl:value-of select=“./price”></td>
</tr>
</xsl:for-each>
</table>
</xsl:template>
```

# Example of a for-each with a sort

```
<xsl:template match=“pricelist">
<table>
<xsl:for-each select=“item"/>
<xsl: sort select =“./price”>
<tr>
<td><xsl:value-of select=“./number”></td>
<td><xsl:value-of select=“./description”></td>
<td><xsl:value-of select=“./price”></td>
</tr>
</xsl:for-each>
</table>
</xsl:template>
```

# Numbering

- Within a template, xsl:number may be used to number items

```
<xsl:template match="/program">
        <ProgramListing>
        <xsl:apply-templates />
        </ProgramListing>
/xsl:template>
<xsl:template match="foo">
        <Module>
        <xsl:number expr="position()"/>
        <xsl:value-of match = "." />
        </Module>
</xsl:template>
```

- Number allows very complex manipulations within one level, across levels, or based on a hierarchy

# Explicit Elements, Attributes, etc.

- Within a template, xsl:element, xsl:attribute, and other explicit copymarking schemes may be used to convert elements in the input document to attributes in the output document and the reverse.

```
<xsl:template match="pricelist">
<xsl:for-each select="item"/>
<item>
<xsl:attribute name = "num"><xsl:value-of select="./number"></xsl:attribute>
<xsl:attribute name = "desc"><xsl:value-of select="./description"></xsl:attribute>
<xsl:attribute name = "price"><xsl:value-of select="./price"></xsl:attribute>
<xsl:value-of select=".">
</item>
</xsl:for-each>
</xsl:template>
```