

Distributed Software Development

XSLT

Chris Brooks

Department of Computer Science
University of San Francisco

Department of Computer Science — University of San Francisco — p. 1/??

XSLT

- XSLT is an XML-based language that allows you to *declaratively* specify how a document should be changed or transformed.
 - You specify the output for a particular element; no need to manage tree traversal.
- Useful for:
 - Emitting an HTML display of an XML document
 - Converting between tag vocabularies
 - Extracting plain text from an XML document
 - Automatically modifying or filtering an XML document.

Department of Computer Science — University of San Francisco — p. 2/??

Output

- You can transform an XML document into:
 - Plain text
 - HTML
 - XML (or any flavor thereof)

Department of Computer Science — University of San Francisco

XSLT processing

- An XSLT engine will take two inputs:
 - An XML document (the source)
 - An XSLT stylesheet which describes the transformations
- The result is a new document, sometimes called the result tree.
- Engine traverses the source tree
 - At each node, look in the stylesheet to see if there is a rule describing how to transform this node.

Department of Computer Science — University of San Francisco — p. 4/??

Our CD database

```
<catalog>
  <song>
    <title> Tomorrow Never Knows </title>
    <artist> Beatles </artist>
    <album> Revolver </album>
    <genre> Rock </genre>
    <rating> 5 </rating>
    <length> 2:57 </length>
    <date>
      <day> 6 </day>
      <month> Feb </month>
      <year> 2005 </year>
    </date>
  </song>
  ...
</catalog>
```

Department of Computer Science — University of San Francisco — p. 5/??

Step 1: Using XSLT to emit plain text

- To begin, let's use XSLT to print a plaintext version of our catalog.
- We can run XSLT from the command line or within a browser.
 - `/usr/bin/4xslt` or `xsltproc` on lab machines
 - Most modern browsers have XSLT support
 - Debugging is easier from the command line

Department of Computer Science — University of San Francisco

Step 2: filtering elements

- That's fine, but pretty dull.
- Let's select just artist, title, and album to display.
- We do that through the use of apply-templates.
- What if we left out apply-templates in the song template?

Emitting HTML

- We can also emit other markup languages, such as HTML. (XHTML, actually).
- Just indicate the tags to be produced by a template match.

Emitting XML

- We can also use XSLT to create new XML documents with different tag names or contents.
- For example, let's say we want to change the tags to be in Spanish.

Copying Nodes

- When transforming from XML to XML, often, it's useful to copy sections of a document without changing it.
- copy makes a shallow copy of a node.
 - Useful if you want to change a bunch of values or attributes.
- copy-of makes a deep copy and lets you specify a path.
- For example, let's make a new database with just artist, album and title.

Incorporating CSS

- We can still use CSS to control presentational elements.
- With HTML, we can just embed a 'link' tag in the generated HTML.

Incorporating CSS

- If we're emitting XML, we can instead embed a *processing instruction* into the output document.
- Note: this will work best if we do the XSLT on the server side.

```
<xsl:processing-instruction name="xml-stylesheet"
href="songs.css" type="text/css"></xsl:processing-instruction>
```

Referencing a stylesheet

- The command line is great for debugging, but sometimes we want the client to do the work.
- Most web browsers have at least some support for XSLT.
 - More advanced features are not universally supported.
 - In particular, the browser's XSLT processor may make a single pass and not apply the CSS. (firefox)

Xpath

- The examples we've seen so far match templates to elements based solely on the element's tag name.
- Often, you want something more flexible:
 - Match the root element
 - Match all text nodes
 - Match all children of an author node
- Essentially, we want to specify matching rules based on an element's position in the DOM tree.
- XPath is a language for doing this.

XPath

- In XPath, everything is dealt with as a path from the root of the tree.
- To find a node, we'll use a *location path*, which consists of a series of *location steps*.
- A location step consists of:
 - An axis that tells us which direction to travel
 - A node test that specifies which types of nodes apply
 - predicates that use boolean tests to help filter nodes.

Axes

- Axes consist of:
- Children and parents, which have their usual meanings.
- Ancestor, which means any node above the node of interest.
- Descendant: any node below the node of interest.
- Following: following siblings and their descendants.
- Preceding: preceding siblings and their descendants.
- Self

Node test

- The second component of the location step is the node test.
- This is joined to the axis by a ::
- Some tests:
 - / - root node
 - * - any element
 - author - any node named "author"
 - text() - any text node
- In our Tolkien example, we might use `book/volumes/volume:"The Two Towers"`

Shortcuts

- // - descending from the root. `//volume` matches all volume nodes below the root.
- ../* - all siblings
- .. - parent
- /* - document element
- @name - matches attribute named 'name'

Examples

- `/songlist/child::node()` - matches all song elements, plus the comment.
- `//comment()/following-sibling::*[title]` - matches 'Tomorrow Never Knows'.
- `/*/*` - matches all song elements
- `id('s1')/..` - matches the songlist element
- `id('s1')/ancestor-or-self::*` - matches the songlist element and the song element for 'Tomorrow Never Knows'
- `id('s1')/genre::country` - matches nothing. (lets you test node type).

Predicates

- If you need more flexibility in specifying nodes of interest, you can use a predicate.
- Predicates are contained inside square brackets.
- To be included in final node set, a node must pass both axis and predicate tests.

Examples

- `//song/[id='s1']/title/text` - text for all 's1' songs.
- `//song[title]` - all quotations that have a source subelement.
- `//song[not(source)]` - songs that do not have a title sub-element.
- `//song[position() == 2]` or `//song[2]` - the second quotation.

So what's all this good for?

- XPath is very useful for allowing users to query an XML document.
- Even more useful for specifying which transformations should be applied in an XML document.
- gives us a way to easily specify transformations that should take place based on a node's context.

Programmatic Control

- XSLT also gives you the sort of control we'd expect from programming languages:
 - if, for-each
 - We can also test and access node values.
- Keep in mind that this still works within a declarative, tree-based context.
- Use these things sparingly and try to work with the XSLT processor, rather than against it.

Tables Example

- Let's use XSLT to take the starship XML data and generate a table.
- We'll use value-of to pull out the text fields.
- Notice that we can use value-of with XPath to test the value of child nodes.

Iteration in XSLT

- `xsl:for-each` lets you iterate over element types within a template match.
- You select nodes and then specify a transformation to happen for each selected node.

Example

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>
<xsl:template match="/">
  <html><body><xsl:apply-templates/>
  </body></html>
</xsl:template>
<xsl:template match="purchases">
  <xsl:for-each select="product">
    <p>Product: <xsl:value-of select="./@name"/>
      Price: <xsl:value-of select="./@price"/></p>
  </xsl:for-each>
</xsl:template>
</xsl:stylesheet>
```

Conditional testing

- XSLT also has an if-style construct.

```
<xsl:template match="nation">
  <xsl:value-of select="name"/>
  <xsl:if test="position( ) != last( )">, </xsl:if>
  <xsl:if test="position( ) mod 5 = 0">
    <xsl:text> </tr><tr></xsl:text>
  </xsl:if>
  <xsl:if test="position( ) = (last( ) - 1)">and </xsl:if>
  <xsl:if test="position( ) = last( )"></xsl:if>
</xsl:template>
```

- This example is not correct; it is not a well-formed XML document. Why?
- Figuring out how to fix this is left as a lab exercise for you.

Sorting

- XSLT also has built-in support for sorting and processing your elements.

Parameters in XSLT

- You can also pass parameters into an XSLT stylesheet.
- You can also define them at the top of your XSLT program.
- Parameters can be referenced with a `$`.

```
<xsl:param name="discount" select="0.10"/>
...
<discount><xsl:value-of select="$discount"/></discount>
  <discountPrice>
    <xsl:value-of select="price - (price * $discount)"/>
  </discountPrice>
```

Using XSLT from Python

```
#!/usr/bin/python

from Ft.Xml.Xslt.Processor import Processor
from Ft.Xml import InputSource

xsltproc = Processor( )
sweaterdata = InputSource.DefaultFactory.fromUri('sweater.xml')
sweaterxsl = InputSource.DefaultFactory.fromUri('sweater.xsl')
xsltproc.appendStylesheet(sweaterxsl)
sweaterXML = xsltproc.run(sweaterdata)

print sweaterXML
```

Summary

- XSLT allows you to declaratively specify how a document should be transformed.
- Works on the parse tree.
 - Can emit text or XML
- XPath allows you to easily identify nodes.
- Makes it easy to have separate representations for storage or network transmission and for display.