5. Document Transformations

- XSLT (W3C Rec. November 1999)
  - A language for transforming XML documents
  - Representative of tree-based transformation languages (DSSSL, MetaMorphosis, TranSID, ...)
  - Designed to be used
    - Primarily as part of XSL formatting
    - Also as an independent transformation language
- Our goal: to understand the basic model and central features of the language
  - Overview and an example
  - Data model and processing model

XSLT: Overview

- XSLT uses XML syntax for expressing transformations
  - Of a document source tree into a result tree
  - Result and source are separate trees
  - By template rules
- Each (ordinary) template rule has
  - A pattern (matched against nodes of the source tree)
  - A template as a body
    - Instantiated to create fragments of the result tree

Overview of XSLT Transformation

Style Sheets and Template Rules

- An xsl:stylesheet (or xsl:transform)
  - Consists of template rules:
    - <xsl:template match="Pattern"> Template</xsl:template>
      - Rule applied to nodes of source tree matched by the Pattern
        - Expresses using XPath (XML Path Language)
    - Template consists of
      - Literal result tree fragments (elements, text), and
      - XSLT instructions for creating further result tree fragments

XPath in Short

- W3C Recommendation (November 1999)
  - A compact non-XML syntax for addressing parts of XML documents
  - Used also in other W3C languages
    - Specs for hyperlinks in XML
    - XLink (Rec.) and XPointer (Draft)
    - XQuery (draft, 11/2003, for querying XML)
  - Provides also typical primitives for manipulating strings, numbers and truth values

An XSL transformation example

- Transform below document to HTML:

```xml
<?xml-stylesheet type="text/xsl" href="walsh.xsl" ?>
<!-- Modified from an example by Norman Walsh -->
<doc><title>My Document</title>
  <para>This is a <em>short</em> document.</para>
  <para>It only exists to <em>demonstrate a simple</em> XML document</para>
  <figure><title>My Figure</title><graphic fileref="myfig.jpg"></graphic></figure>
</doc>
```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>A Document</title>
</head>
<body>
<h1>My Document</h1>
<p>This is a <i>short</i> document.<i></i></p>
<p>It only exists to <i>demonstrate a <b>simple</b> XML document</b>.</p>
<div>
<b>Figure 1. </b> <br />
<img src="myfig.jpg"><b>My Figure</b>
</div>
</body>
</html>

<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
<html><head><title>A Document</title></head>
<body>
<!-- process root's children here: -->
<xsl:apply-templates />
</body>
</html>
</xsl:template>
<xsl:template match="doc/title">
<h1><xsl:apply-templates /></h1>
</xsl:template>
<xsl:template match="para">
<p><xsl:apply-templates /></p>
</xsl:template>
<xsl:template match="em">
<i><xsl:apply-templates /></i>
</xsl:template>
<xsl:template match="em/em">
<b><xsl:apply-templates /></b>
</xsl:template>
<xsl:template match="figure">
<!-- Insert a bold caption of form 'Figure Num.' by counting all figures in the document: -->
<div><b>Figure <xsl:number level="any" count="figure" />.</b> 
<!-- Process the children of figure, -->
<!-- the 'graphic' child first: -->
<xsl:apply-templates select="graphic" />
<!-- then the 'title' child: -->
<xsl:apply-templates select="title" /></div>
</xsl:template>
<xsl:template match="graphic">
<img src="{@fileref}" />
<!-- Assign the value of attribute 'fileref' to attribute 'src' -->
</xsl:template>
<xsl:template match="figure/title">
<b> <xsl:apply-templates /></b>
</xsl:template>
</xsl:stylesheet>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>A Document</title>
</head>
<body>
<h1>My Document</h1>
<p>This is a <i>short</i> document.<i></i></p>
<p>It only exists to <i>demonstrate a <b>simple</b> XML document</b>.</p>
<div>
<b>Figure 1. </b> <br />
<img src="myfig.jpg"><b>My Figure</b>
</div>
</body>
</html>
What use of XSL or XSLT?

- XSL can be used in different ways
  - for offline document formatting
    - produce, say, PDF from XML by an XSL style sheet (using XSLT + XSL formatting objects)
  - for offline document manipulation
    - transform XML into other form (XML/HTML/text) using XSLT
  - for online document delivery
    - in a Web server
    - in a Web browser (if the browser supports)

XSLT in online document delivery

- XSLT in a browser
  - defines rendering of XML documents
  - approach of Microsoft Internet Explorer
    - transformation of XML to HTML on the fly in browser
    - NB: Microsoft’s XSLT implementation differs from XSLT 1.0 (a new one exists but has to be installed separately)
- XSLT in a Web server
  - an HTTP request for an XML document served by transforming XML on the fly to HTML (or other format) on the server

Main Aspects of XSLT

- Data model
  - How document data is viewed in XSLT?
- Selection mechanism
  - How document parts are selected for processing?
- Matching
  - How are the template rules selected?
- Processing model
  - How XSLT execution proceeds?

Data Model of XSLT and XPath

- Documents viewed as abstract tree structures
- Seven types of tree nodes
  - root (additional parent of document element)
  - element nodes
  - attribute nodes
  - text nodes
  - comments, processing instructions and namespaces
- NB: Entity references are expanded
  - no entity nodes

XSLT/XPath document trees

- Element nodes have elements, text nodes, comments and processing instructions of their (direct) content as their children
  - NB: attribute nodes are not children
  - the value of an element node is the concatenation of its text-node descendants
- Nodes have a complete document order
  - root node first, otherwise according to the order of the first character of the XML markup for each node
  - element node precedes its attribute nodes, which precede any content nodes of the element

XSLT/XPath Trees

- Similar to the DOM structure model, with slight differences
  - value of an element: its full textual content (in DOM: null)
  - no names for text nodes, comment nodes, etc. (in DOM: "#text", "#comment", etc.)
- Example document:

  `<article>Written by <fig file="pekka.jpg" caption="The Lecturer"/> the lecturer.</article>`
XSLT/XPath trees: Example

Main Aspects of XSLT

- Data model
- Selection mechanism
  - How document parts are selected for processing?
  - A: With XPath expressions
- Matching
- Processing model

XPath Expressions

- Used for selecting source tree nodes, conditional processing, and generating new text content
  - return node-sets, truth values, numbers or strings
  - can select any parts of source tree (node-set) for processing, using...

Location paths

- Consist of location steps separated by ‘/’
  - each step produces a set of nodes
  - steps evaluated left-to-right
  - path begins with ‘/’ -> root is the first context node

Location steps: Axes

- In total 13 axes
  - for staying at the context node:
    - self
  - for going downwards:
    - child, descendant, descendant-or-self
  - for going upwards:
    - parent, ancestor, ancestor-or-self
  - for moving towards beginning or end of the document:
    - preceding-sibling, following-sibling, preceding, following
  - “Special” axes
    - attribute, namespace

XPath Axes and Their Orientation

- Axes are oriented away from the context node (except attribute and namespace axes, which are unordered sets)
  - the position() for the closest node = 1
  - for the most remote node, position() = last()
- The simplest axis, self::
XPath Axes and Their Orientation

- **parent::**
  ![Parent Diagram]

- **ancestor::**
  ![Ancestor Diagram]

- **ancestor-or-self::**
  ![Ancestor-or-Self Diagram]

- **child::**
  ![Child Diagram]

- **descendant::**
  ![Descendant Diagram]

- **descendant-or-self::**
  ![Descendant-or-Self Diagram]

- **preceding-sibling::**
  ![Preceding-Sibling Diagram]

- **following-sibling::**
  ![Following-Sibling Diagram]

- **following::**
  ![Following Diagram]

- **preceding::**
  ![Preceding Diagram]
Location paths: Node tests

- **Name**: any element node with name Name (on an axis or any attribute node with name Name)
- **:***: any element node (on an axis, any attribute node, any value node)
- **text()**: any text node
- **comment()**: any comment node
- **processing-instruction()**: any processing instruction node
- **node()**: any node of any type

Location paths: Abbreviations

- **'child::'** can be omitted
- **'attribute::'** can be shortened to '='
- **'self::node()'** can be shortened to '
- **'parent::node()'** can be shortened to '
- **'descendant-or-self::node()'** can be shortened to '/'
- **'position()=n'** for testing occurrence
- **'//node()'** can be shortened to '

Semantics of Location Paths (example)

Value after each step:

1. value

Location path examples (1)

- chap children of current node:
  .//chap (or equivalently: chap, or .//child::node()[name()='chap'])
- The document element (child element of root node): /'
- Elements chapter anywhere (below the root):
  //chapter (//chapter -> anywhere below the current node)
- All chapters with attribute style: '
- the previous chapter sibling:
  preceding-sibling::chapter[1]

Location path examples (2)

- All child elements having an attribute type:
  *[@type]
- All child elements of any author child:
  author/*
- Sections whose style attribute equals style attribute of the document element:
  //sect[@style = '/@style']
- First author child, and previous to the last:
  author[1], author[last()-1]

Location path examples (3)

- Predicate expressions can be Boolean combinations:
  - child elements author with a publ child, but without degree or award children:
    author[publ and not(degree or award)]
- Closest chap ancestor (and highest):
  ancestor:chap[1] (ancestor:chap[last()])
- Author of the highest sect ancestor that is contained in an appendix:
  ancestor:sect[ancestor:app[last()]][ancestor:author]
Main Aspects of XSLT

- Data model
- Selection mechanism
- Matching
- Processing model

Matching

- How are the rules selected?
- A: With Patterns

XSLT Patterns

- Main use in match attributes of template rules:
  `<xsl:template match='Pattern'>`  

- Restricted location path expressions:
  - with steps using only `child` and `attribute` axes and separated by `/` or `//`
  - but arbitrary predicate expressions `([Expr])` allowed
  - may begin with `id(`/`idVal`)
    (for selecting element nodes by ID attribute values)
  - alternative patterns separated by `|` (node-set union operator)

XSLT Patterns: Semantics

- A location path pattern \( P \) is of form
  \[ \text{Step}_1 \odot \text{Step}_2 \odot \ldots \odot \text{Step}_{n-1} \odot \text{Step}_n \]
  where each separator \( \odot \) is either `//` or `/`
  - may also begin with `//`; Pattern `//` matches only the root

- Else \( P \) matches a node \( v \), if there are nodes \( v_i \ldots v_n \) such that each \( v_i \) satisfies the node test and possible predicates of \( \text{Step}_i \), and which form a path towards the root.
  - If \( P \) begins with a single `/'`, node \( v_1 \) must be child of the root
  - in case of \( \text{Step}_1 // \), node \( v_n \) must be parent of \( v \)
  - in case of \( \text{Step}_n // \), node \( v_i \ldots v_n \) may be any ancestor of \( v \)

XSLT Patterns: Examples

- `match='sect-head | section/head'`
  - matches any element with name `sect-head`, and any `head` elements directly below a `section`

- A node matches the pattern
  `/appendix//ulist/item[1]`
  if it is the first `item` element in a `ulist` element which is contained in an `appendix`, which is the document element

Main Aspects of XSLT

- Data model
- Selection mechanism
- Matching
- Processing model

- How does the XSLT execution proceed?

XSLT Processing Model

0. Parse the document into a source tree
1. Construct the result tree by applying template rules to the source tree
   - may contain text and arbitrary elements (including XSL FO formatting objects)
2. Serialize the result tree (as XML, HTML or text), or interpret it to produce formatted output
   (in case of full XSL processing)
Overview of XSLT Transformation

Result Tree Construction Algorithm

Start at the source tree root;
while (unprocessed source nodes left) do
  Find matching template rules;
  Choose one rule (the one with highest priority);
  Instantiate rule template in the context of the current node; Add the result to the result tree;
  Recursively process any source nodes selected by XSLT instructions (apply-templates, for-each) in the template;
endwhile;

Selecting among multiple matching rules

- Priority of a rule can be specified explicitly:
  `<xsl:template priority="2.0" ...`
- Default priorities based on the match pattern:
  - 0 for simple name tests (@para, @href)
  - negative for less specific patterns e.g., `*, @*, node()`
  - 0.5 for more complex patterns
- Multiple matching rules with the same maximum priority is an error - Processor may recover by (quietly!) choosing the last one of those rules

Application of template rules

- Without a select attribute
  `<xsl:apply-templates />`
  processes all children of current node
- `->` Default behaviour: top-down traversal of source tree
- Otherwise the selected nodes are processed
  `in document order (if not sorted with xsl:sort)`
- Built-in rules allow recursive traversal to proceed gracefully in the absence of matching rules

Default rules for elements and content

- Built-in rule for the root and element nodes:
  `<xsl:template match="/ | *">
   `<xsl:apply-templates />`
  </xsl:template>`
- Built-in rule for text and attribute nodes:
  `<xsl:template match="text() | @*">
   <!-- Insert the string value of current node: -->
   `<xsl:value-of select="." />`
  </xsl:template>`
- Low priority `-->` can be overridden

A (Tricky) Processing Example

- Consider transforming document `A` with the below rules:
  `<xsl:template match="/">
   <R><xsl:apply-templates select="//C" /></R>
  </xsl:template>`
  `<xsl:template match="/">
  </xsl:template>`
  `<xsl:template match="/">
   <xsl:apply-templates />
  </xsl:template>`
Processing example (2)

The result

\[
<\text{R}><\text{NewC}>\text{New: }\text{b1b3ccb2}/\text{NewC}</\text{R}>
\]

is obtained as follows:

1. Rule 1 matches the root node. Element node R is added to the result; Instruction \texttt{<xsl:apply-templates select="//C"/>} selects the (only) C element for processing (which will produce the contents of node R).
2. Rule 2 with pattern ".*" creates into result tree a NewC element node with text node "New: " as its first child.

Processing example (3)

3. Instruction \texttt{<xsl:apply-templates select="../B"/>} selects element siblings of current node (C). The built-in element rule applies to these, and the built-in text rule to their children.
   - Result: text nodes "b1" and "b3" become the next children of NewC.
4. Instruction \texttt{<xsl:apply-templates />} in the context of element node NewC selects its children, "cc" and \texttt{<B>b2</B>}, for processing. The built-in text rule inserts value "cc" to the result tree, and the B element node becomes "b2" in the result (similarly to step 3).

Processing example (4)

<table>
<thead>
<tr>
<th>Source</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>R</td>
</tr>
<tr>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>B</td>
<td>NewC</td>
</tr>
<tr>
<td>b1</td>
<td>cc</td>
</tr>
<tr>
<td>cc</td>
<td>dd</td>
</tr>
<tr>
<td>b3</td>
<td>b1</td>
</tr>
<tr>
<td>b2</td>
<td>b3</td>
</tr>
<tr>
<td>cc</td>
<td>b2</td>
</tr>
</tbody>
</table>

Is it Really So Tricky?

- In practice: Seldom
  - A computer scientist wants to understand how a model really works ...
- XSLT is a high-level declarative language for describing transformations
  - Normally no need to think so hard about execution;
  - Often sufficient just to specify declarative rules to handle different cases, like
    \[
    \texttt{<xsl:template match="para">}
    \texttt{<P><xsl:apply-templates /></P>}</xsl:template>
    \]

[More SDPL Notes 5: XSLT](#)