5 Document Transformations

- XSLT (1.0 W3C Rec. 11/1999; XSLT 2.0 Rec. 1/07)
  - A language for transforming XML documents
  - Initial main purpose to support XSLT formatting
  - Used mainly for independent transformations
    (esp. XML --> HTML)
- Our goal: to understand the basic model and central features of XSLT
  - Overview and an example
  - Data model and processing model

XSLT: Overview

- XSLT uses XML syntax for expressing transformations
  - A document source tree into a result tree
    » result and source are separate trees
  - By template rules
- Each 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

- An xsl:stylesheet (or xsl:transform) consists of template rules:
  <xsl:stylesheet namespace prefix = "conventional XSLT"
                  version = "1.1"
                  xmlns = "http://www.w3.org/1999/XSL/Transform"
                  >
  — Rule applied to nodes of the source tree matched by the Pattern
  — Expressed using XPath (XML Path Language)
  — Template consists of
    » literal result tree fragments (elements, text), and
    » XSLT instructions for controlling further processing

Style Sheets and Template Rules

XPath in a Nutshell

- XPath 1.0 W3C Rec. 11/99 (2.0 Rec. 1/07)
  - A compact non-XML syntax for addressing parts of XML documents (as node-sets)
  - Used also in other W3C languages
    » Specs for hyperlinks in XML:
      XLink (Rec. '01) and XPointer (Rec. '03)
    » XPath (Rec. 1.07; extends XPath 2.0)
  - Also typical operations on strings, numbers and truth values

Early impressions of XSLT 2.0

- Somewhat more advanced & flexible than 1.0
- Complexity of 1.0 vs. 2.0 specs (as pages):
  XSLT 1.0 ~ 90
  XPath 1.0 ~ 30
  Funcs & opers ~ 160
  XSLT 2.0 ~ 280
  Total ~ 120

An XSL transformation example

- Transform below document to HTML:
  <xsl:stylesheet type = "text/xsl" href = "walsh.xsl" >
</xsl:stylesheet>
  <html>
  <head>
  <title>My Document</title>
  </head>
  <body>
  <para>
  This is a &lt;para&gt; document.&lt;/para&gt;
  <para>
  It only exists to &lt;para&gt; demonstrate a
  &lt;em&gt;simple&lt;/em&gt; XML document.&lt;/para&gt;
  "<figure>
  &lt;title&gt;My Figure&lt;/title&gt;
  &lt;graphic filesrc = "myfig.jpg" /&gt;
  </figure>
  "</para>
  </body>
  </html>
Example style sheet begins

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

Example (figures)

```xml
<xsl:template match="figure">
  <!-- Insert a bold caption of form 'Figure Num. ' by counting all figures in the document: -->
  <div><h1><xsl:apply-templates select="title" /></h1></div>
  <!-- Process the children of figure, -->
  <div>
    <!-- the 'graphic' child first: -->
    <xsl:apply-templates select="graphic" />
    <!-- then the 'title' child: -->
    <xsl:apply-templates select="title" />
  </div>
</xsl:template>
```

Example end of style sheet

```xml
<xsl:template match="para">
  <p><xsl:apply-templates /></p>
</xsl:template>
```

Result (edited for readability)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <title>My Document</title>
</head>
<body>
  <h1>My Document</h1>
  <p>It only exists to demonstrate a simple XML document.</p>
  <p>This is a short document.</p>
</body>
</html>
```

What use of XSL(T)?

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

Main Aspects of XSLT

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

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

XSLT/XPath Trees

Similar to the DOM, with slight differences:
- 7 vs 12 node types
- value of an element: its full textual content
  (in DOM: null)
- no names for text nodes, comment nodes, etc.
  (in DOM: "#text", "#comment", etc.)

Document order of nodes:
- 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 document trees

Defined in Sect. 5 of XPath 1.0 spec
- for XSLT/XPath 2.0 & XQuery in their joint Data Model

Element nodes have elements, text nodes, comments and processing instructions of their (direct) content as their children
- NB: attribute nodes are not children (but have a parent)
- the value of an element node is the concatenation of its text-node descendants

Data Model of XSLT and XPath

Documents are viewed as trees made of seven types of nodes:
- root (additional parent of document element)
- element nodes
- attribute nodes
- text nodes
- comments, processing instructions and namespaces
- NB: Entity references are expanded

NB

XSLT/XPath trees: Example

<fig file="pekka.jpg" caption="The Lecturer" />

XSLT/XPath document trees

Defined in Sect. 5 of XPath 1.0 spec
- for XSLT/XPath 2.0 & XQuery in their joint Data Model

Element nodes have elements, text nodes, comments and processing instructions of their (direct) content as their children
- NB: attribute nodes are not children (but have a parent)
- the value of an element node is the concatenation of its text-node descendants

Location steps: Axes

In total 13 axes (~ directions in tree)
- 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 start/end of the document:
  » preceding-sibling, following-sibling, preceding, following
- "Special" axes
  » attribute, namespace

Location paths

Consist of location steps separated by '/'
- each step produces a set of nodes
- steps evaluated left-to-right, each node in turn as context node
  » path begins with '/' -> root is the first context node

Complete location step:
AxisName: (PredicateExpr)*
- axis specifies the tree relationship between the context node and the selected nodes
- node test restricts the type and and name of nodes
- filtered further by 0 or more predicates

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
- the most characteristic of XPath expressions
- evaluated with respect to a context node
  » often the current node matched by the template pattern
- result: set of nodes selected by the location path

Document order of nodes:
- 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 document trees

Defined in Sect. 5 of XPath 1.0 spec
- for XSLT/XPath 2.0 & XQuery in their joint Data Model

Element nodes have elements, text nodes, comments and processing instructions of their (direct) content as their children
- NB: attribute nodes are not children (but have a parent)
- the value of an element node is the concatenation of its text-node descendants

Location steps: Axes

In total 13 axes (~ directions in tree)
- 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 start/end of the document:
  » preceding-sibling, following-sibling, preceding, following
- "Special" axes
  » attribute, namespace

Location paths

Consist of location steps separated by '/'
- each step produces a set of nodes
- steps evaluated left-to-right, each node in turn as context node
  » path begins with '/' -> root is the first context node

Complete location step:
AxisName: (PredicateExpr)*
- axis specifies the tree relationship between the context node and the selected nodes
- node test restricts the type and and name of nodes
- filtered further by 0 or more predicates
XPath Axes and Their Orientation

- Ordinary axes oriented away from context node (attribute and namespace axes are unordered)
  - the position() for the closest node = 1
  - for the most remote node, position() = last()

- The simplest axis, self::

  ![Diagram of self::]

- parent:: (exists for every node except the root)

  ![Diagram of parent::]

- ancestor::

  ![Diagram of ancestor::]

- ancestor-or-self::

  ![Diagram of ancestor-or-self::]

- descendant::

  ![Diagram of descendant::]

- descendant-or-self::

  ![Diagram of descendant-or-self::]

- preceding::

  ![Diagram of preceding::]

- following::

  ![Diagram of following::]

Location paths: Node tests

- Location paths: /[/Step]/.../Step, where
  - Step = [axis::]NodeTest(['Pred'])*

- Node tests (slightly simplified)
  - Name: any element node with thatName
    (on an attribute axis, any attribute node with that Name)
  - : any element (on an attribute axis)
    text(): any text node
    comment(): any comment node
    processing-instruction(): any processing instruction
    node(): any node of any type
Location paths: Abbreviations

- Abbreviations in location steps
  - ’child:’ can be omitted
  - ’attribute:’ can be shortened to ’@’
  - ’self::node()’ can be shortened to ’.’ (period)
  - ’parent::node()’ can be shortened to ’../’
  - Predicate [position()=n]’ for testing occurrence position n can be shortened to ’[n]’
  - /descendant-or-self::node()’/ shortened to ’/’

-> Syntax resembles slightly Linux/Unix file path names

Semantic of Location Paths (example)

Location path examples (1)

- chap: children of current node:
  .//chap (or simply chap, or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or
  or

Location path examples (2)

- All child elements having an attribute type:
  *[type]
  NB: Node sets as truth values: empty = false; non-empty = true
  - All child elements of any author child:
  author/*
  - sections whose type attribute equals style
  attribute of the document element:
  //sect[@type = /*/@style]
  - First author child, and previous to the last:
  author[1], author[last()-1]

Main Aspects of XSLT

- Data model
- Selection mechanism
- Matching
- How are the rules selected?
- A: With Patterns
- Processing model

XSLT Patterns

- Main use in match attributes of template rules:
  <xsl:template match="Pattern"> 
  - also used for numbering (Which parts are counted?)
- Restricted location path expressions:
  - steps with child and attribute axes only, separated by ’/’ or ’/’
  - but arbitrary predicates in [Expr] allowed
  - may begin with id()/id(val)
  (for selecting element nodes by ID attribute values)
  - alternative patterns separated by ’|’ (~ node-set union)

XSLT Patterns: Semantics

- A node n matches a pattern P if n would be
  selected by P in some context
  - > the context has to be an ancestor of n
  - can be tested by considering Steps of P right-to-left and
  checking that they can be satisfied along a path from n
  towards root

XSLT Patterns: Examples

- match="sect-head | section/head"
  - matches any element with name sect-head, and any
  head elements directly below a section
- Pattern
  /appendix/ulist/item[1]

matches the first item element in a ulist 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?

Overview of XSLT Transformation

XSLT Processing Model

0. Parse the document into a source tree
1. Construct the result tree by applying template rules to the source tree
2. Serialize the result tree (as XML, HTML or text)

Selecting one of matching rules

- Priority of a rule can be specified explicitly:
  - 0 for simple name tests (like para, @href)
  - negative for less specific patterns
  - 0.5 for more complex patterns
- Multiple matching rules with the same maximum priority is an error - Processor may (quietly) choose the last one of them

Result Tree Construction (roughly)

Application of template rules

- Without a select attribute (~ select="node()")
  - `<xsl:apply-templates />`
  - processes all children of current node
  - `->` = "default traversal": top-down
- Selected nodes are processed in document order (if not sorted with `xsl:sort`)
- Built-in rules support the top-down traversal if no matching rules are found

Built-In Default Rules

- For the root and element nodes:
  - `<xsl:template match="/ | *">`  
  - `<xsl:apply-templates />`  
  - `/xsl:template>`
- For text and attribute nodes:
  - `<xsl:template match="text() | @*">`  
  - `<xsl:value-of select="." />`
  - `/xsl:template>`
- Low priority `->` can be overridden

A (Tricky) Processing Example

- Consider transforming document
  - `<A>`  
  - `<B>`  
  - `<C>`
  - `<D>`
  - `<E>`
  - `<F>`
  - `/A>`
  - `</F>`
  - `</E>`
  - `</D>`
  - `</C>`
  - `</B>`
  - `</A>`

with the below rules:

- `<xsl:template match="/">`  
  - `<!-- Rule 1 -->`
  - `<D>:<xsl:apply-templates select="//C" /></D>`
  - `</xsl:template>`
- `<xsl:template match="C">`  
  - `<!-- Rule 2 -->`
  - `</New>:<xsl:apply-templates select="..//B" />`
  - `</New>`
  - `</xsl:template>`
  - `/xsl:template>`
1. Rule 1 matches the root node. Element node \( R \) is added to the result; Instruction \(<\text{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 \( "C" \) creates into result tree a \( \text{NewC} \) element node with text node \( \text{New: } " \) as its first child.

3. Instruction \(<\text{xsl:apply-templates select="../B" />\> selects element \( B \) 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 \( \text{NewC} \).

4. Instruction \(<\text{xsl:apply-templates />}\) in the context of element node \( C \) selects its children, \( "cc" \) and \( <\text{B}>b2</\text{B}> \), for processing. The built-in text rule inserts value \( "cc" \) to the result tree, and the \( \text{B} \) element node becomes \( "b2" \) in the result (similarly to step 3).