5. Document Transformations

- **XSLT (W3C Rec. November 1999)**
  - A language for transforming XML documents
    - designed to be used
    - as part of XSL formatting
    - 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 rules 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" />
    <result>
      ... NB. well-formed ...
    </resulttemplate>`
  - Rule applied to nodes of source tree matched by the **Pattern**
    - expressed using XML Path Language (XPath)
  - **Template consists of**
    - literal result tree fragments (element, 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, 11/2002, 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"/>...This is a <em>short</em> document.
<para>It only exists to <em>demonstrate a</em> <em>small</em> XML document</para>
<fig title="My Figure"/>
<graphic fileref="myfig.jpg"/>
</doc>
```
XSLT in online document delivery

- XSLT in a browser
  - defines rendering of XML documents
  - approach of Microsoft Internet Explorer 5
    - 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

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
- NB: Entity references are expanded
  -> no entity nodes

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?

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 considered 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 it's 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
  - 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

Location paths

- Consist of location steps separated by ‘/’
  - each step produces a set of nodes
  - steps evaluated left-to-right
  - each node in turn acting as a context node
  - path begins with ‘/’ – root is the first context node
- Complete form of a location step:
  - AxisName::NodeTest[(PredicateExpr)]*
    - axis specifies the tree relationship between the context node and the selected nodes
    - node test restricts the type and name of nodes
    - further filtered by 0 or more predicates

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

- **ancestor:**
  - Diagram showing the relationship between an ancestor and its descendants.

- **ancestor-or-self:**
  - Diagram showing the relationship between an ancestor, its descendants, and itself.

XPath Axes and Their Orientation

- **child:**
  - Diagram showing the relationship between a parent and its children.

- **preceding-sibling:**
  - Diagram showing the relationship between elements that are siblings and have a preceding element.

- **following-sibling:**
  - Diagram showing the relationship between elements that are siblings and have a following element.

XPath Axes and Their Orientation

- **descendant:**
  - Diagram showing the relationship between a parent and its descendants.

- **descendant-or-self:**
  - Diagram showing the relationship between a parent, its descendants, and itself.

XPath Axes and Their Orientation

- **following:**
  - Diagram showing the relationship between an element and its following elements.

- **preceding:**
  - Diagram showing the relationship between an element and its preceding elements.

Location paths: Node tests

- **Node tests (slightly simplified):**
  - `[ Name ]`: any element node with name Name
  - `[ any attribute ]`: any attribute node
  - `[ text ]`: any text node
  - `[ comment ]`: any comment node
  - `[ processing-instruction ]`: any processing instruction node
  - `[ node ]`: any node of any type
Location paths: Abbreviations

- Abbreviations in location steps
  - ‘child::’ can be omitted
  - ‘attribute()’ can be shortened to ‘.’ (period)
  - ‘parent::node()’ can be shortened to ‘.’ (period)
  - ‘Parent::node()’ can be shortened to ‘.’ (period)
  - 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

Semantics of Location Paths (example)

Location path examples (1)

- chap children of current node:
  - ./chap (or equivalently, chap, or
    ./child::node(hWnd::="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 type="intro";
  //chapter[@type="intro"]
- the previous chapter sibling
  preceding-sibling::chapter[1]

Location path examples (2)

- All child elements having an attribute type:
  *[@type] (attribute) = type
  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[2], author[last()-1]

Location path examples (3)

- Predicate expressions can be Boolean
  combinations:
  - child elements author with a publ child, but without
    degree of 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()]/author

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:
  \(<xsl:template match="Pattern">\)
- Restricted location path expressions:
  - with steps using only child and attribute axes and
    separated by "/" or "/\"/
  - but arbitrary predicate expressions (\(\text{Expr}\)) allowed
  - may begin with an "id" value
  - 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 \(\text{Step}_0 / \text{Step}_1 / \ldots / \text{Step}_n\),
  where each separator is either "/" or "/\"
  - may also begin with "/\"
  - \(\text{Pattern} /\"\)
    matches only the root
- Else \(P\) matches a node \(v_i\) if there are nodes \(v_0, \ldots, v_i\) such
  that each \(v_j\) satisfies the node test and possible predicates of \(\text{Step}_j\), and which form a path towards the root:
  - if \(P\) begins with a single "/\"
    node \(v_i\) must be child of the root
  - in case of \(\text{Step}_i /\"\)
    node \(v_i\) must be parent of \(v_j\)
  - in case of \(\text{Step}_i /\\text{Step}_j\)
    node \(v_i\) may be any ancestor of \(v_j\)

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
  \(<\text{appendix//ulist/item}[1]/>\)
  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)

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 (like `text`, `@ref`) 
  - negative for less specific patterns 
    e.g. `*`, `0*`, `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() | @*">` 
  `<xsl:apply-templates select="." />` 
  `<xsl:template>` 
- Low priority `-->` can be overridden

A (Tricky) Processing Example

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

The result 
  `<A><B><C><D>B</D></C><B></A>` 
  is obtained as follows: 
1. Rule 1 matches the root node `-->` Element node `B` is added to the result; Instruction `xsl:apply-templates select="/C" />` selects the (only) `C` element for processing (which will produce the contents of node `B`). 
2. Rule 2 with pattern `=".*"` creates into result tree a `NodeC` element node with text node `"B: "` as its first child.

Processing example (3)

3. Instruction `xsl:apply-templates select=".."` 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 `"B: "` and `"C: "` become the next children of `NodeB`. 
4. Instruction `xsl:apply-templates />` in the context of element node `C` selects its children, `"B: "` and `"C: "` for processing. The built-in text rule inserts value `"C: "` to the result tree, and the `B: "` element node becomes `"B: C: "` in the result (similarly to step 3).
Is it Really So Tricky?

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