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>`
    - `<xsl:template>` NB: well-formed XML ->
  - Rule applied to nodes of source tree matched by the **Pattern**
    - expressed using XML Path Language (XPath)
- **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
    - XLink and XPointer (drafts for hyperlinks in XML)
    - XQuery (draft, 12/2001, 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 version='1.0'?>
  <xsl: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 <em>simple</em> XML document</em>.</para>
    <figure>
      <title>My Figure</title>
      <graphic fileref="myfig.jpg"/>
    </figure>
  </doc>
  ```
Example style sheet begins

```xml
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/"> <!-- rule for root -->
    <HTML><HEAD><TITLE>A Document</TITLE></HEAD>
    <BODY>
      <!-- process root's children here: -->
      <xsl:apply-templates />
    </BODY>
  </xsl:template>

  <xsl:template match="doc/title"> <!-- rule for title element -->
    <H1><xsl:apply-templates /></H1>
  </xsl:template>

  <xsl:template match="para"> <!-- rule for paragraph -->
    <P><xsl:apply-templates /></P>
  </xsl:template>

  <xsl:template match="em"> <!-- rule for emphasis -->
    <I><xsl:apply-templates /></I>
  </xsl:template>

  <xsl:template match="em/em"> <!-- rule for nested emphasis -->
    <B><xsl:apply-templates /></B>
  </xsl:template>

  <xsl:template match="figure"> <!-- rule for 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" />  <!-- for 'graphic' -->
      <!-- then the 'title' child: -->
      <xsl:apply-templates select="title" />  <!-- for 'title' -->
    </DIV>
  </xsl:template>

  <xsl:template match="graphic"> <!-- rule for graphic -->
    <IMG> <!-- Create new attribute 'src': -->
      <xsl:attribute name="src">
        <!-- and assign the value of current element's fileref attribute to it: -->
        <xsl:value-of select="@fileref" />
      </xsl:attribute>
    </IMG>
  </xsl:template>

  <xsl:template match="figure/title"> <!-- rule for figure title -->
    <B> <xsl:apply-templates /></B>
  </xsl:template>  <!-- for 'graphic' -->
</xsl:stylesheet>
```

Example (paras and emphs)

```xml
<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>
```

Example (figures)

```xml
<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" />  <!-- for 'graphic' -->
    <!-- then the 'title' child: -->
    <xsl:apply-templates select="title" />  <!-- for 'title' -->
  </DIV>
</xsl:template>

<xsl:template match="graphic">
  <IMG> <!-- Create new attribute 'src': -->
    <xsl:attribute name="src">
      <!-- and assign the value of current element's fileref attribute to it: -->
      <xsl:value-of select="@fileref" />
    </xsl:attribute>
  </IMG>
</xsl:template>

<xsl:template match="figure/title">
  <B> <xsl:apply-templates /></B>
</xsl:template>  <!-- for 'graphic' -->
```

Example (end of style sheet)

```xml
<xsl:template match="graphic">
  <IMG> <!-- Create new attribute 'src': -->
    <xsl:attribute name="src">
      <!-- and assign the value of current element's fileref attribute to it: -->
      <xsl:value-of select="@fileref" />
    </xsl:attribute>
  </IMG>
</xsl:template>

<xsl:template match="figure/title">
  <B> <xsl:apply-templates /></B>
</xsl:template>  <!-- for 'graphic' -->
```

Result (edited for readability)

```xml
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html><head><title>A Document</title></head>
<body>  
  <h1>My Document</h1>  
  <p>This is a <i>short</i> document.</p>  
  <p>It only exists to <i>demonstrate a <b>simple</b> XML document</i>.</p>  
  <div>
    <b>Figure 1. </b>  
    <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 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) in the server

Data Model of XSLT and XPath

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

XSLT/XPath document trees

- Element nodes have elements, text nodes, comments and processing instructions of their (direct) content as child nodes
  - NB: attribute nodes 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: Example

- 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

- 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

XPath Expressions
Location paths (1)
- 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
- 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 paths (2)
- 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 paths (3)
- Abbreviations in location steps  
  - ‘child::’ can be omitted  
  - ‘attribute::’ can be shortened to ‘@’  
  - ‘//descendant-or-self::node()/’ shortened to ‘//’  
  - ‘self::node()’ can be shortened to ‘.’  
  - Predicate ‘[position()=n]’ for testing occurrence position n can be shortened to ‘[n]’
- ‘/’ in the beginning sets context to the root node
- Syntax resembles Linux/Unix file path names

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

- child:

  ![Child Axis Diagram]

- descendant:

  ![Descendant Axis Diagram]

- descendant-or-self:

  ![Descendant-or-Self Axis Diagram]

- preceding-sibling:

  ![Preceding-Sibling Axis Diagram]

- following-sibling:

  ![Following-Sibling Axis Diagram]

---

Semantics of Location Paths (example)

A

B

C

“txt”

D

E

F

G

H

I

J

K

/*[child::A]

{2, 5, 7} \{3, 4, 6, 8\} \{2, 7\}

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
  - 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`:
  ```xml```
  `*[@type]`
  ```
- NB: use of node sets as truth values:
  ```xml```
  `empty` - `false`; non-empty - `true`
  ```
- All child elements of any `author` child:
  ```xml```
  `author/*`
  ```
- Sections whose `type` attribute equals the `style` attribute of the document element:
  ```xml```
  `//sect[@type = /*/@style]`
  ```
- First author child, and previous to the last:
  ```xml```
  `author[1], author[last()-1]`
  ```
Location path examples (3)

- Predicate expressions can be Boolean combinations:
  ```xml```
  `– 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):
  ```xml```
  `ancestor::chap[1]` (or `ancestor::chap[last()]`)
  ```
- `author` of the highest `sect` ancestor that is contained in an `appendix`:
  ```xml```
  `ancestor::sect[ancestor::app][last()]//author`
  ```

XSLT Patterns

- Main use in match attributes of template rules:
  ```xml```
  `<xsl:template match="Pattern">`
  ```
- Restricted location path expressions:
  – may use only `child` and `attribute` axes and operators `/` and `//`
  – can use arbitrary predicate expressions `[Expr]`
  – 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
  ```xml```
  `Step1§Step2§…§Stepn`,
  where each separator `§` is either `/` or `//`
  ```
- Pattern `P` matches a node `n` if the rightmost step `Stepn` matches `n`, and a suitable node `v` matches the prefix `Step1§…§Stepn-1` of `P`:
  – if the rightmost `§` is `/`, `v` must be the parent of `n`
  – if the rightmost `§` is `//`, `v` may be any ancestor of `n`
  ```
- That is, there’s a path `(Stepn,…,Step1)` from `n` towards root

XSLT Patterns: Examples

- Pattern
  ```xml```
  `sect-head | section/head`
  ```
matches any element with name `sect-head`, and any `head` elements directly below a section
  ```
- For a match with pattern
  ```xml```
  `/appendix//ulist/item[1]`
  ```
matches any element which is contained in an `appendix`, which is the document element
  ```

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, formatting objects and arbitrary elements
2. Serialize the result tree (as XML, HTML or text), or interpret it to produce formatted output
   (in case of full XSL processing)
### XSL Tree Construction

Start at the source tree root:

```java
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:**
  ```xml
  <xsl:template priority="2.0">
  ... 
  </xsl:template>
  ```

- **Default priorities based on the match pattern:**
  - 0 for simple name tests (like `@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**
  ```xml
  <xsl:apply-templates />
  ```
  processes all children of current node

- **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:**
  ```xml
  <xsl:template match="/ | *">
  <xsl:apply-templates/>
  </xsl:template>
  ```

- **Built-in rule for text and attribute nodes:**
  ```xml
  <xsl:template match="text() | @*">
  <!-- Insert the string value of current node: -->
  <xsl:value-of select="."/>
  </xsl:template>
  ```

### A (Tricky) Processing Example

- **Consider transforming document**

```xml
<A>
  <C>c1</C><B>bb</B><C>c2</C></B><D> dd</D><C>c3</C>
</A>
```

with the below rules:

```xml
<xsl:template match="/"> <!-- Rule 1 -->
  <R><xsl:apply-templates select="//B" /></R>
</xsl:template>
```

```xml
<xsl:template match="B"> <!-- Rule 2 -->
</xsl:template>
```

### Processing example (2)

- **The result**

```xml
<R><NewB>New: c1c3bbc2</NewB></R>
```

is obtained as follows:

1. Rule 1 matches the root node → Element node `<R>` is added to the result; Instruction `<xsl:apply-templates select="/B" />` Selects the (only) B element for processing (which will produce the contents of node R).
2. Rule 2 with pattern "*" creates into result tree a `<NewB>` element with text node "New:" as its first child.
Processing example (3)

3. Instruction `<xsl:apply-templates select="../C"/>` selects element `C` siblings of current node (3). The built-in element rule applies to these, and the built-in text rule to their children. Result: text nodes "c1" and "c3" become the next children of `NewB`.

4. Instruction `<xsl:apply-templates />` in the context of element node `B` selects its children, "bb" and `<C>c2</C>`, for processing. The built-in text rule inserts value "bb" to the result tree, and the `C` element node becomes "c2" 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 about execution in so much detail; Often sufficient just to specify declarative rules to handle different cases, like
    ```xml
    <xsl:template match="para">
        <p><xsl:apply-templates /></p>
    </xsl:template>
    ```