5.1 XPath & XSLT: Additional features

- XPath support for
  - arithmetical operations
  - processing ID/IDREF cross-references
  - manipulation of strings
- Generating text
  - for content
  - for attribute values
- Repetition, sorting and conditional processing
- Generating numbers
- Computational power of XSLT

XPath: Arithmetical Operations

- Operations for double-precision (64 bit) floating-point numbers
  - +, -, *, div, mod (same as % in Java)
  - functions to map numbers to integers:
    - floor(-1.1) = -2, floor(1.1)=floor(1.5)=1
    - ceiling(-1.1) = -1, ceiling(1.1)=ceiling(1.5)=2
    - round(-1.1) = -1, round(1.1) = 1, round(-1.5) = -1, round(1.5) = 2

Cross-referencing

- Function id selects elements by their unique ID
  - NB: ID attributes need to be declared
    (in DTD or its internal subset; See an example later)
- Examples
  - id('sect:intro') selects the element with unique ID "sect:intro"
  - id('sect:intro')/para[5] selects the fifth para child of the above element

String manipulation

- Equality and inequality of strings can be tested with operators = and !=
  - "foo" = 'foo'; "foo" != "Foo"
- Functions for concatenation, and for testing for substrings:
  - concat("dog", "bert") = "dogbert"
  - starts-with("dogbert", "dog") = true()
  - contains("dogbert", "gbe") = true()

XPath: more string functions

- substring-before("dogbert", "bert") = "dog"
- substring-after("dogbert", "b") = "ert"
- substring(\text{"string", startpos, length?});
  - substring("dogbert", 1, 3) = "dog"
  - substring("dogbert", 3) = "gbert"
- string-length("dogbert") = 7
- replace("Str", ReplacedChars, ReplacingChars)
  - translate("dogbert", "dog", "gil") = "gilbert"

Computing generated text

- The string-value of an expression can be inserted in the result tree by instruction
  \text{<xsl:value-of select="Expr"/>}
  - if the expression evaluates to a node-set, the value of the first node in document order is used
- Consider transforming source elements like
  - \text{<name alias="Bird">}
    - \text{<first>Charlie</first>\text{<last>Parker</last>}}
    - \text{</name>}
  - to the form
    - Charlie ("Bird") Parker
Computing generated text (2)

- This can be specified by template rule
  ```xml
  <xsl:template match="name">
    <xsl:value-of select="first" />
    (<xsl:value-of select="@alias" />)
    <xsl:value-of select="last" />
  </xsl:template>
  </xsl:template>
  ```

- Verbatim text (like the linefeed above) can be inserted using `xsl:text`.

Attribute value templates

- The string-value of an expression can be inserted in an attribute value by surrounding the expression by braces `{ and }`.

- Consider transforming source element

  ```xml
  <photo>
    <file>Mary.jpg</file>
    <size width="300" />
  </photo>
  ```

  into form

  ```xml
  <img src="/images/Mary.jpg" width="300" />
  ```

Attribute value templates (2)

- This can be specified by template rule

  ```xml
  <xsl:template match="photo">
    <img src="/images/{file}
    width="{size/@width}" />
  </xsl:template>
  ```

- Expressions `{file}` and `{size/@width}` are evaluated in the context of the current node (the photo element).

XSLT: Repetition

- Nodes can be "pulled" from source for processing using instruction

  ```xml
  <xsl:for-each select="Expr">
    Template
  </xsl:for-each>
  ```

  – the template is applied to each of the selected nodes (0, 1 or more), each node in turn as the current node
  
  » in document order, unless sorted using `xsl:sort` instructions (see later)

Example (xsl:for-each)

- Consider formatting the below document as HTML:

  ```xml
  <!DOCTYPE document [<!ATTLIST section id ID #IMPLIED>]]>
  <document>  
    <title>The Joy of XML</title>
    <section id="Intro"> 
      <title>Getting Started</title>
      <name> 
        <first>Helen</first> <last>Brown</last>
      </name>
      says that processing XML documents is fun.
      <name> 
        <first>Dave</first> <last>Dobrik</last>
      </name> agrees.
    </section>
    <section>
      <title>Family affairs</title>
      <name> 
        <first>Bob</first> <last>Brown</last>
      </name> is the husband of 
      <name> 
        <first>Helen</first> <last>Brown</last>
      </name>.
    </section>
    <section>
      <title>Finishing Up</title>
      As we discussed in <title- ref idref="Intro" />, processing XML documents is fun. 
    </section>
  </document>
  ```

Example: Table of contents

- A table of contents can be formed of section titles:

  ```xml
  <xsl:template match="/">
    <HTML><HEAD><TITLE><xsl:value-of select="/document/title"/></TITLE></HEAD></HTML>
    <BODY>
      <H2>Table of Contents</H2>
      <OL>
        <xsl:for-each select="/section">
          <LI><xsl:apply-templates /></LI>
        </xsl:for-each>
      </OL>
      <xsl:apply-templates select="/section" />
    </BODY>
  </xsl:template>
  ```
Example (cont; Cross references)

- Cross references (to sections) can also be processed using `xsl:for-each`:
  ```xml
  <xsl:template match="title-ref">
    <xsl:for-each select="id(@idref)">
      Section \(<xsl:value-of select="substring(title, 1, 8)" />...\)
    </xsl:for-each>
  </xsl:template>

- With this rule the source fragment
  ```xml
  As we discussed in <title-ref idref="Intro"/>
  ```
  becomes
  ```xml
  As we discussed in Section (Getting ...)
  ```

XSLT Sorting

- A sorted order for the processing of nodes with `xsl:for-each` and `xsl:apply-templates` can be specified by `xsl:sort`:
  ```xml
  <xsl:sort select="" />
  <xsl:sort select="" order="descending" />
  ```

- controlled by attributes of `xsl:sort` like
  - `select`: expression for the sort key
  - `data-type`: "text" (default) or "number"
  - `order`: "ascending" (default) or "descending"

- The first `xsl:sort` specifies the primary sort key, the second one the secondary sort key, and so on.

Example (cont; Sorted index of names)

- All names can be collected in a last-name-first-name order using the below template
  ```xml
  <xsl:for-each select="//name">
    <xsl:sort select="last" />
    <xsl:sort select="first" />
    <li><xsl:value-of select="last" />, <xsl:value-of select="first" /></li>
  </xsl:for-each>
  ```

- This creates an UL list with items
  ```xml
  <li>Brown, Bob</li>
  <li>Brown, Helen</li>
  <li>Brown, Helen</li>
  <li>Dobrik, Dave</li>
  ```

What about duplicates?

- Is it possible to eliminate duplicate values like
  ```xml
  <li>Brown, Helen</li>
  <li>Brown, Helen</li>
  ```

- Yes (but not that straightforward)

- Using conditional instructions
  ```xml
  <xsl:apply-templates />
  <xsl:if test="not(position()=last())">, </xsl:if>
  ```

Conditional processing

- A template can be instantiated or ignored based on the value of a test Boolean expression, using
  ```xml
  <xsl:if test="Expression">
    Template
  </xsl:if>
  ```

- Example: a comma-separated list of names:
  ```xml
  <xsl:template match="namelist/name">
    <xsl:apply-templates/>
    <xsl:if test="not(position()=last())">,</xsl:if>
  </xsl:template>
  ```

- Also a case-like construct:
  ```xml
  <xsl:choose>
    <xsl:when test="Expr1" > ... </xsl:when>
    <xsl:when test="Expr2" > ... </xsl:when>
    ...<xsl:otherwise> ... </xsl:otherwise>
  </xsl:choose>
  ```

- If no 'when' applies, optional 'otherwise' is instantiated:
  ```xml
  <xsl:if test="not(position()=last())">, </xsl:if>
  ```

Conditional processing (2)
Example (cont; Eliminating duplicate names)

- No access to other nodes (except \texttt{current()}) in the list of \texttt{xsl:for-each}.
  - But can refer to other nodes in the source tree.
  - Process just the first one of duplicate names:
    \[
    \texttt{xsl:for-each select="//name"} \\
    \texttt{xsl:if test="not\texttt{(preceding::name[first=current()/first}} \texttt{and last=current()/last])"}} \\
    \texttt{xsl:if test="not\texttt{(first=current()/first}} \texttt{and last=current()/last})"}} \\
    \texttt{xsl:for-each select="//name"} \\
    \texttt{xsl:sort select="last"} \\
    \texttt{xsl:sort select="first"} \\
    \texttt{<LI><xsl:value-of select="last"/>}, \texttt{<xsl:value-of select="first"/>}\texttt{</LI>}
    \texttt{</xsl:if>}
    \texttt{</xsl:for-each>}
\]

Generating numbers

- Formatted numbers can be inserted in the result tree by element \texttt{xsl:number}.
  - Can be specified by attribute \texttt{value="Expr"}
  - Otherwise the number generated based on the position of the current node in the source tree.

Example 1, a numbered list:

\[
\texttt{xsl:template match="ol/item"} \\
\texttt{<xsl:apply-templates/>}
\]

Generating numbers (2)

Example 2. Hierarchical numbering (1.1, 1.1.1, 1.1.2, ...) for titles of chapters, titles of sections of chapters, and titles of subsections of sections:

\[
\texttt{xsl:template match="title"} \\
\texttt{xsl:apply-templates/>}
\]

Generating numbers (3)

Example 3. Sequential numbering of \texttt{notes} within chapters:

\[
\texttt{xsl:template match="note"} \\
\texttt{xsl:apply-templates/>}
\]

Computational power of XSLT

- Programming-like features, for example,
  - Limited variables (names bound to values):
    \[
    \texttt{xsl:for-each select="//name"} \\
    \texttt{xsl:variable name="LandF" select="concat(last, ', ', first)"} \\
    \texttt{\ldots} \\
    \texttt{xsl:call-template name="process-name"} \\
    \texttt{xsl:with-param name="pname" select="$LandF"} \\
    \texttt{\ldots}
    \texttt{</xsl:for-each>}
\]

- Named templates which can be explicitly called with parameters:
  \[
  \texttt{xsl:call-template name="process-name"} \\
  \texttt{xsl:apply-templates/>}
  \]

- Conditional instructions, string functions, and named templates sufficient to simulate Turing machines.

Implications:
- XSLT has full algorithmic programming power
  - Is this intentional?
  - Inconvenient as a general-purpose programming language!
- Impossible to recognize non-terminating transformations automatically
  - Malicious hacker could cause “denial-of-service” through non-terminating style sheets