5.1 Additional features of XPath & XSLT

- 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

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) = 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
  - id('sect1 sect2 sect3') selects 3 sections
    (if they have the corresponding ID values)

String manipulation

- Equality and inequality of strings can be tested with operators = and !=
  - "foo" = "foo", "foo" != "Foo"
- Testing for substrings:
  - starts-with("dogbert", "dog") = true()
  - contains("dogbert", "ge") = true()
- Concatenation (of two or more strings):
  - concat("dog", "bert") = "dogbert"

XPath: more string functions

- substring-before("dogbert", "bert") =
  substring-before("dogbert", "u") = "dog"
- substring-after("dogbert", "u") = "bert"
- substring(string, startpos, length?):
  - substring("dogbert", 1, 3) = "dog"
  - substring("dogbert", 3, 1) = "gert"
- string-length("dogbert")=7
- translate(Str, ReplacedChars, ReplacingChars):
  - translate("dogbert", "dps", "D11") = "D11bert"

Computing generated text

- The string-value of an expression can be inserted in the
  result tree by instruction
  <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
  <name alias="Bird">Charlie</name> Charlie's last name
  <name alias="Bird">Parker</name> Parker's last name
  to the form
  Charlie ("Bird") Parker
Computing generated text (2)

- This can be specified by template rule
  ```
  <xsl:template match="name">
    <xsl:value-of select="first" />
    <xsl:value-of select="last" />
  </xsl:template>
  ```
- Verbatim text (like the white space 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: `&quot;`.
- Consider transforming source element
  ```
  <photo>
    <file>Mary.jpg</file>
    <size width="300"/>
  </photo>
  ```
  into form
  ```
  <img src="/images/Mary.jpg" width="300"/>
  ```

Attribute value templates (2)

- This can be specified by template rule
  ```
  <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
  ```
  <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 (of `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</name>
      <last-David</last>
      <name>first-Dave</name>
      <last-Dobie</last>
      <name>first</name>
      <last-Brown</last>
    </section>
    <section id="Family affairs">
      <name>first-Bob</name>
      <last-Jane</last>
      <name>first</name>
      <last-Jim</last>
    </section>
    </document>
  ```
  - As we discussed in: `<xsl:for-each>`

Example: Table of contents

- A table of contents can be formed of section titles:
  ```
  <xsl:template match="/">
    <body>
      <head>
        <title>[xsl:value-of select="/document/title"]</title>
      </head>
      <ul id="table-of-contents">
        <li><xsl:apply-templates select="/li"/></li>
      </ul>
      <ul id="slightly-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(//'title-ref')">  
    <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="IsThis"/>
```

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="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:template match="/*">  
  <xsl:apply-templates select="*"/>  
  <xsl:for-each select="/*/name">  
    <xsl:value-of select="last"/>  
    <xsl:apply-templates select="name[1]"/>  
  </xsl:for-each>
</xsl:template>
```

- This creates an UL list with items

```xml
  <li>Brown, Bob</li>  
  <li>Brown, Helen</li>  
  <li>Obrih, Dave</li>
```

Conditional processing

- A template can be instantiated or ignored based on the value of a `test` Boolean expression, using `xsl:if`:

```xml
<xsl:if test="Expression">  
  <xsl:template match="name">  
    <xsl:apply-templates/>  
  </xsl:template>
</xsl:if>
```

- Example: a comma-separated list of names:

```xml
<xsl:template match="name">  
  <xsl:apply-templates/>  
  <xsl:if test="not(position()-last())">  
    ,  
  </xsl:if>
</xsl:template>
```

Conditional processing (2)

- Also a case-like construct (`xsl:choose` in Java):

```xml
<xsl:choose>  
  <!-- The first 'when' whose test-true() is instantiated: -->  
  <xsl:when test="Expr1"/>  
  <xsl:when test="Expr2"/>  
  <xsl:otherwise/>  
  <!-- If no 'when' applies, optional 'otherwise' is instantiated: -->  
  <xsl:otherwise/>  
</xsl:choose>
```
Example (cont: Eliminating duplicate names)

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

Generating Numbers

- Formatted numbers can be inserted in the result tree by element xsl:number
  - number can be specified by attribute value="Expr"
  - otherwise the number generated based on the position of the current node in the source tree

Example 1, a numbered list:

<i>... default: count like siblings (items) ...</i>

Generating numbers: Example 1

Example 2: Variation

- As above, but numbering titles within appendices with A, A.1, A.1.1, B.1 etc:

Example 2: Variation

Generating numbers: Example 2
Example 2: Variation

Generating numbers: Example 3

Sequential numbering of notes within chapters:
(more precisely: after the start of the preceding chapter)

```xml
<xsl:template match="note">
  <xsl:variable name="secnum" select="number from="chap"
    format="(i)"
  />
  <xsl:apply-templates/>
</xsl:template>
```