5.1 Additional Features

- XPath support for
  - arithmetics
  - 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

- Operators for double-precision (64 bit) floating-point numbers
  - +, -, *, div, mod (same as \( \times \) in Java)
- Rounding numbers up, down, and to the closest integer:
  - floor(x), ceiling(x), round(x)
- Formatting numbers as strings (e.g.):
  - format-number(-1.2534, "0.0") = \( -1.3 \)
  - XSLT function; applies Java decimal format patterns

Aggregate Functions

- Counting nodes
  - \( \text{count}(\text{nodelist}) \)
  - and summing them as numbers
  - \( \text{sum}(\text{nodelist}) \)
- Example:
  - Average of course grades below current node:
    \[
    \text{sum}(.//\text{course}/@grade) \div \text{count}(.//\text{course})
    \]

Cross-referencing

- Function id selects elements by their unique ID
  - NB: ID attributes must be declared in DTD
    (See an example later)
- Examples:
  - id('sect:intro')
  - selects the element with unique ID "sect:intro"
  - id('sect:intro')/auth[3]
    - selects the element with unique ID and its 3rd children
  - id('sect1 sect2 sect3')
    - selects 3 sections with corresponding ID values

String manipulation

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

XPath: more string functions

- substring-before("ftp://a", "/") = substring-before("ftp://a", "/") = "ftp:
  - substring-after("ftp://a", "/") = "/a"
- substring(string, start, length?)
  - substrings("dogbert", 1, 3) = "dog"
  - substrings("dogbert", 3) = "gbert"
  - string-length("dogbert")
- translate(Str, Replaced, Replacing):
  - translate("doggy", "dgo", "Ssi") = "Ssey"

Generating Text

- The string-value of an expression can be inserted in the result tree by instruction
  - \(<\text{xsl:value-of select="Expr" } />\)
  - if \( \text{Expr} \) evaluates to a node-set, \text{value} of the first node in document order is used (XSLT 2.0: of all, space-separated)
- Example: Transform elements like
  - <name alias="Bird"> <first>Charlie</first> <last>Parker</last> </name>
  - to the form
    - Charlie ("Bird") Parker

Computing generated text (2)

- This can be specified by template rule
  - \(<\text{xsl:template match="name" } />
  - \(<\text{xsl:value-of select="first" } />\)
    - ("<\text{xsl:value-of select="alias" } />")
  - \(<\text{xsl:value-of select="last" } />
  - \(<\text{xsl:text } /
  - \</text>
  - \</xsl:template>
  - \</xsl:template>
- Verbatim text (like the white-space above) can be inserted using \text{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 }
- Example: Transform source element
  `<file>Mary.jpg</file>`
  `<size width="300"/>`
  into form
  `<img src="/images/Mary.jpg" width="300"/>`

XSLT: Repetition

- Nodes can be "pulled" from source for processing using
  `<xsl:for-each select="Expr">`
  Template
  `<xsl:for-each>`
  - Template is applied to the selected nodelist, each node in turn as the current() node
  - in document order, unless sorted using `xsl:sort` instructions (see later)

Example: Table of contents

- A table of contents can be formed of section titles:
  `<xsl:template match="/">`
  `<HTML><HEAD> <TITLE><xsl:value-of select="document/title"/></TITLE></HEAD> <BODY>`
  `<H2>Table of Contents</H2>`
  `<OL>`
  `<xsl:for-each select="//section/title">`
  `<LI>`
  `<xsl:apply-template /></LI>`
  `</xsl:for-each>`
  `</UL>`
  `</BODY>`
`</HTML>`

XSLT Sorting

- A sorted order for the processing of nodes with `xsl:for-each` and `xsl:apply-templates` can be specified by
  `<xsl:sort/>`
- controlled by attributes of `xsl:sort` like
  - `select`: expression for the sort key (default: ".")
  - `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.
Conditional processing

- A template can be instantiated or ignored based on the value of a Boolean expression:
  - `<xsl:if test="Expression">` Template
  - `</xsl:if>`
- Example: a comma-separated list of names:
  - `<xsl:template match="namelist/name">`<xsl:apply-templates/>
  - `<xsl:if test="position() &lt; last()">`<xsl:if>
  - `</xsl:if>`

Example (cont; Eliminating duplicate names)

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

Generating Numbers

- Formatted numbers can be inserted in the result tree by element `<xsl:number />
  - by the position of the current node in the source tree
  - nodes to be counted specified by a count pattern
  - common numbering schemes supported: single-level, hierarchical, and sequential ignoring levels
- Typical cases in following examples
  - (Complete specification rather complex)
- Example 1: Numbering list items

Generating numbers: Example 2

- Hierarchical numbering (1.1, 1.1.1, 1.1.2, ...) for titles of chapters, titles of their sections, and titles of subsections:
  - `<xsl:template match="title">`<xsl:template>
  - `<xsl:number level="multiple" count="chap|sect|subsect" format="1.1 " />`<xsl:number>
  - `<xsl:apply-templates/>`<xsl:apply-templates>

Conditional processing (2)

- Also a case-like construct (~ switch in Java):
  - `<xsl:choose>`<xsl:choose>
  - `<! -- The first 'when' whose test=true() is instantiated: -- >`<xsl:when>
  - `<xsl:when test="Expr">`<xsl:when test="Expr">
  - `<! -- If no 'when' applies, an optional 'otherwise' is instantiated: -->`<xsl:when>
  - `<xsl:otherwise>`. `<xsl:otherwise>`
  - `</xsl:choose>`<xsl:choose>

Example (cont; Eliminating duplicate names)

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

Generating numbers: Example 1

- `<xsl:template match="/item">`<xsl:template>
  - `<! -- default: count similar siblings (items) -->`<xsl:apply-templates/>
  - `<xsl:template>`<xsl:template>

Generating numbers: Example 2

- Hierarchical numbering (1, 1.1, 1.1.1, 1.1.2, ...) for titles of chapters, titles of their sections, and titles of subsections:
  - `<xsl:template match="title">`<xsl:template>
  - `<xsl:number level="multiple" count="chap|sect|subsect" format="1.1 " />`<xsl:number>
  - `<xsl:apply-templates/>`<xsl:apply-templates>

An aside: Meaning of position()
5.2 Computing with XSLT

XSLT is a declarative rule-based language
- for a special purpose: XML transformations
- Could we use XSLT for procedural computing?
- What is the exact computational power of XSLT?

We’ve seen some programming-like features:
- iteration over source nodes (xsl:for-each)
- conditional evaluation (xsl:if and xsl:choose)

Further programming-like features:
- variables (names bound to non-updatable values):
  <xsl:variable name="LAndF"
  select="concat(last, ", ", first)" />
  ...
- callable named templates with parameters:
  <xsl:call-template name="process-name">
    <xsl:with-param name="name" select="$(LAndF)" />
  </xsl:call-template>
  ...
  <xsl:for-each>
    <TABLE>...</TABLE>

Visibility of Variable Bindings

The binding is visible in following siblings of xsl:variable, and in their descendants:

<xsx:for-each select="/name">
  <xsl:variable name="LAndF"
  select="concat(last, ", ", first)" />
  ...
  <xsl:call-template name="process-name">
    <xsl:with-param name="name" select="$(LAndF)" />
  </xsl:call-template>
  ...
</xsl:for-each>

A Real-Life Example

We used LaTeX to format an XML article. For this, we needed to map source table structures
<group cols="3">
  ...
</group>
to corresponding LaTeX environments:
\begin{tabular}{l|l|l} ... \end{tabular}

How to do this?
The "halting problem" has no algorithmic solution.

Are there limits to XSLT processing that we can do?

Implementations provide extension mechanisms, e.g., to call arbitrary Java methods.

Inconvenient as a general purpose programming language!

Any algorithm can be shown computable with plain XSLT

– by simulating Turing machines, by a recursive named template with string parameters

What does this mean?

XSLT has full algorithmic power

– (It is "Turing-complete")

– Is this intentional?

> Inconvenient as a general-purpose programming language!

> Impossible to recognise non-terminating transformations automatically

> Could attempt "denial-of-service" attacks with non-terminating style sheets

Output: 2006 XSLT: Additional Features and Computing 37