5.1 Additional Features

- XPath for arithmetics, cross-references, and string manipulation
- Generating text
  - for content
  - for attribute values
- Repetition, sorting and conditional processing
- Numbering document contents

5.2 Computing with XSLT

XSLT: Arithmetics

- Double-precision floating-point operators
  - +, -, *, div, mod (same as % in Java)
  - e.g. 2.3 mod 1.1 = 0.1
- 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 1.0 function; uses Java decimal format patterns

Aggregate Functions

- Counting nodes
  - count(node-set) — and summing them as numbers
  - sum(node-set)
- Example:
  - Average of observed temps below current node:
    sum(//obs/temperature) / div count(//obs)

String manipulation

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

Generating Text

- Insert text nodes with a computed value in the result:
  <xsl:value-of select="Expr" />
  - If Expr gives a node-set, the value of the first node in document order is used (in XSLT 2.0 all, space-separated)
- Example: Transform elements like
  <name alias="Bird"> <first>Charlie</first> <last>Parker</last> </name>
  to the form
  Charlie ("Bird") Parker

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 third auth of the above element
  - id('sect1 sect2 sect3') selects 3 sections
    (with corresponding ID values)

More XPath string functions

- substring-before("ftp://a", "/") =
  - substring-before("ftp://a", "/") = "ftp:
- substring-after("ftp://a", "/") = "/a"
- string-length("dogbert") = 7
- substring(string, start, length?):
  - substring("dogbert", 1, 3) = "dog"
  - substring("dogbert", 3) = "bert"
- translate(Str, Replaced, Replacing):
  - translate("doggy", "dgo", "Ssi") = "Sissy"

Computing generated text (2)

- This can be specified by template rule
  <xsl:template match="name">
    <xsl:value-of select="Expr" />
  </xsl:template>

- Verbatim text (like the white-space above) can be inserted using xsl:text

Examples:

- translate("dogg y", "dgo", "Ssi") = "Sissy"
- floor(x), ceiling(x), round(x)
- Example: Transform elements like
  <name alias="Bird"> <first>Charlie</first> <last>Parker</last> </name>
  to the form
  Charlie ("Bird") Parker

- substring-before("ftp://a", "/") =
  - substring-before("ftp://a", "/") = "ftp:
- substring-after("ftp://a", "/") = "/a"
- string-length("dogbert") = 7
- substring(string, start, length?):
  - substring("dogbert", 1, 3) = "dog"
  - substring("dogbert", 3) = "bert"
- translate(Str, Replaced, Replacing):
  - translate("doggy", "dgo", "Ssi") = "Sissy"
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
  
  ```xml
  <photo>
    <file> Mary.jpg </file>
    <size width="300"/>
  </photo>
  
  into form
  
  `<img src="/images/Mary.jpg" width="300"/>
  ```

XSLT: Repetition

- Nodes can be "pulled" from source for processing using
  
  ```xml
  <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:
  
  ```xml
  <xsl:template match="title-ref"> 
    `<li><xsl:value-of select="last" /></li>` 
  </xsl:template>
  ```

Example (of for-each)

- Format the below document as HTML:
  
  ```xml
  <!DOCTYPE document [<!ATTLIST section id ID #IMPLIED>]> 
  <document>
    <section id="Intro">Getting Started</section>
  </document>
  ```

Example (cont; Cross references)

- Cross-ref can also be processed for-each:
  
  ```xml
  <xsl:for-each select="id(@idref)">
    Section (current value):
    `<li><xsl:value-of select="substring(title, 1, 8)" /></li>`
  </xsl:for-each>
  ```

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:for-each select="/html/list">
    `<li><xsl:value-of select="last" /></li>`
  </xsl:for-each>
  ```

Example (cont; Sorted index of names)

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

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

- Possible to eliminate duplicates? Yes, but a bit tricky. See next slide...
Conditional processing

- A template can be instantiated or ignored with
  ```xml
  <xsl:if test="BooleanExpr">Template</xsl:if>
  ```
- Example: a comma-separated list of names:
  ```xml
  <xsl:template match="nameList">  
      <xsl:if test="position() &lt; last()">  
          <name>&lt;name2>&lt;/name2>&lt;name3>&lt;/name3>&lt;name4>&lt;/name4>&lt;/nameList>
      </xsl:if>
  </xsl:template>
  ```

Conditional processing (2)

- A case construct (~ switch in Java):
  ```xml
  <xsl:choose>  
      <xsl:when test="expr1"> <expr2> <expr3> </expr2> </xsl:when>  
      <xsl:otherwise> <expr4> </xsl:otherwise>  
  </xsl:choose>
  ```

Example (cont; Eliminating duplicate names)

- Only the current() node accessible in current node list — but can refer to nodes in the source tree
  - Process just the first one of duplicate names:
    ```xml
    <xsl:for-each select="//name">  
        <xsl:sort select="first"/>  
        <xsl:sort select="last"/>  
        <xsl:sort-select select="first"/>  
    </xsl:for-each>
    ```

Generating numbers: Example 1

- 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
  - supports common numbering schemes: single-level, hierarchical, and sequential through levels
- Typical cases in following examples
  - (Complete specification is rather complex)
- Example 1: Numbering list items

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:
  ```xml
  <xsl:template match="item">
      <item>
  ```
5.2 Computing with XSLT

- XSLT is a declarative rule-based language
  - for XML transformations
  - Could we use it for general computing?
  - What is the exact computational power of XSLT?
- We've seen some programming-like features:
  - iteration over source nodes (xsl:for-each)
    - in XSLT 2.0 iteration over arbitrary sequences
  - conditional evaluation (xsl:if and xsl:choose)

Further programming-like features:
- variables (names bound to non-updatable values):
  - select="concat(last, ', ', first)" /
  - (names bound to non-updatable values):
  - select="concat(last, ', ', first)" /
- callable named templates with parameters:
  - select="concat(last, ', ', first)" /
  - select="concat(last, ', ', first)" /

Visibility of Variable Bindings

- The binding is visible in following siblings of xsl:variable, and in their descendants:
  - select="concat(last, ', ', first)" /
  - select="concat(last, ', ', first)" /
  - select="concat(last, ', ', first)" /

Result Tree Fragments

- Result tree fragments built by templates can be stored, too:
  - select="concat(last, ', ', first)" /
- They can only be used as string values
  - substring(\$fooBar, 2, 2) = "AR"
- or inserted in the result:
  - select="concat(last, ', ', first)" /
- (XSLT 2.0 allows unlimited processing of (computed) sequences)
A Real-Life Example

- We used LaTeX to format an XML article. For this, we needed to map source table structures
to corresponding LaTeX environments:
  \begin{tabular}{lll} \hline
  ... \\
  \end{tabular}
- How to do this?

More General Solution (1/2)

- Pass the column-count to a named template which generates the requested number of 'l's:
  \begin{tabular}{l}
  \hline
  ... \\
  \end{tabular}
- How to support arbitrarily many columns?

Iteration in XSLT 2.0

- XSLT 2.0 is more convenient, by allowing iteration over generated sequences, too:
  \begin{tabular}{l}$\begin{tabular}{l}
  ... \\
  $\end{tabular}$
- OK, but inelegant!
- How to support arbitrarily many columns?

Stylesheet Parameters

- Stylesheet can get params from command line, or through JAXP with Transformer.setParameter(name, value):
  Transformer.transform(...).
- Example "database"

Simulating DB Queries

- Relational DB queries can be simulated easily, with variables and nested for-each loops
  - (but XQuery is designed especially for such)
- Example:
  - Join "tuples" by matching values of their fields
  - Assume information of fathers' addresses and children:

Possible solution (for up to 4 columns)

- XSLT: Additional features & computing 3
- OK, but inelegant!
- How to support arbitrarily many columns?
Computational Power of XSLT

- XSLT seems quite powerful, but how powerful is it?
  - Implementations provide extension mechanisms, e.g., to call arbitrary Java methods.
  - Are there limits to XSLT processing that we can do without extensions?
- We’ll see that any algorithmic computation can be simulated with plain XSLT.
  - Shown indirectly, through simulating Turing machines by XSLT.

Turing Machine

- Alan Turing 1936/37
- Formal model of algorithms
- Primitive but powerful enough to simulate any computation expressible in any algorithmic model (Church/Turing thesis).
- Turing machine:
  - A finite set of states.
  - Unlimited tape of cells for symbols, examined by a tape head.

Illustration of a Turing Machine

Control of a Turing machine

- Control defined by a transition function $\sigma(q, a) = (q', b, d)$, where $d \in \{\text{left, right}\}$.
  - Meaning: with current state $q$ and tape symbol $a$,
    - move to new state $q'$
    - replace symbol $a$ in the current cell by $b'$
    - move tape head one step in direction $d$.
- Such control can be simulated in XSLT with a recursive named-template; Call it transition.

The “transition” template

- Parameters:
  - state: the current state
  - left: contents of the tape up to the tape head
  - right: contents of the tape starting at the cell pointed by the tape head
- Transition simulates a single transition step; calls itself with updated parameters.

Overall structure of the simulation

- Join children with their daddies:
- Implementations provide extension mechanisms, e.g., to call arbitrary Java methods.
- Such control can be simulated in XSLT with a recursive named-template; Call it transition.

Updating the representation of the tape

- For each right-move $\sigma(q, a) = (q', b, \text{right})$, concatenate 'b' at the end of $\text{left}$ and drop the first character of $\text{right}$
- Left-moves $\sigma(q_i, a) = (q_j, b, \text{left})$ in a similar manner:
  - drop the last character of $\text{left}$, and concatenate it in front of $\text{right}$ whose first character has been replaced by 'b'
- Example: a TM for palindromes over alphabet $\{a, b\}$ ('#' used for denoting empty cells)

Simulating a single step (1/2)

- First update the parameters:
  - $\text{newstate} = \text{move}_a$;
  - $\text{newleft} = \text{concat}($left, '#')$;
  - $\text{newright} = \text{substring}($right, 2)$;

- Then call "transition" with new params:

Simulating a single step (2/2)

Sample trace of the simulation

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 (the "halting problem" has no algorithmic solution)
      - could cause "denial-of-service" through non-terminating style sheets