**XML Data Processing**

Prof. Weining Zhang

---

**What to Do with XML Data**

- Present the data in some form
  - Display on the Web
- Obtain and process the data content
  - Process an order in e-business
- Transform XML data from one form to another
  - From one DTD to another
  - From XML to non-XML
- Query the content of XML data

---

**Basic Tasks**

- Choose the type of documents to be processed
  - Verify the conformity to a DTD or XML Schema w/ a verification parser
- Obtain XML information items
  - Parsing using SAX or DOM API
- Transform XML data
  - Using XSLT & XPath
- Query XML data
  - Better use an XML database w/ XQuery

---

**Simple Application Using SAX**

- Implement event handler interface(s)
  - Both methods predefined in the interface and methods of your special needs can be implemented
  - When the predefined methods are called, you will get an info item. You just need to decide what to do with the information
- Write an application driver
  - Create event handler objects
  - Setup the SAX XMLReaderFactory to generate an XMLReader (the parser)
  - Set parser features to control the parser behavior
  - Set event handlers for the parser
  - Use the parser to parse an XML document

---

**Simple Application Using DOM**

- Create the application code
  - Your code should assume the availability of DOM objects: Node, Document, Element, …
  - Your code will deal with DOM trees that will be generated by DOM parsers
- Create an application driver
  - Setup a DOM ParserWrapper to create a parser
  - Set parser features to control the parser behavior
  - Use the parser to parse an XML document into a DOM Document node
  - Pass the document node to the application code for processing

---

**Transform XML Documents**

- Tasks
  - Transform data from one form to another, either another DTD type XML or non-XML form
- Standard XML querying/transform languages
  - XPath
    - Simple language consisting of path expressions
  - XSLT
    - Simple language designed for translation from XML to XML and XML to HTML
**XPath: XML Path Language**

- Current version 2.0
- Functional language
  - Queries are path expressions
  - Strongly typed
  - Based on a tree model
  - Has many built-in functions
  - To be used within (embedded in) other languages

**Path Expression**

- Used to navigate to desired parts of documents
- Consists of a sequence of location steps separated by “/”
  - inputFunction()/LStep1/LStep2/…/LStepN
- Result of a path expression is a sequence of items (nodes or values)
  - A node in XML data model contains pointers to its children, therefore represents the root of a subtree

**Bank Example**

```xml
<!DOCTYPE bank-2]
  <!ELEMENT bank-2 (account | customer | depositor)>  
  <!ELEMENT account (branch, balance)> 
  <!ATTLIST account acct-num ID # REQUIRED
  owners IDREFS # REQUIRED>
  <!ELEMENT customer(name, street, city)> 
  <!ATTLIST customer cust-id ID # REQUIRED
  accounts IDREFS # REQUIRED>
  ... declarations for depositor, branch, balance, customer-
  name, customer-street, customer-city, etc.
]
```

**Bank Example (cont.)**

```xml
  <bank-2>
    <account acct-num="A-401" owners="C100 C102"/>
    <branch-name>Downtown</branch-name>
    <balance>500</balance>
  </account>
  <customer cust-id="C100" accounts="A-401"> 
    <name>Joe</name>
    <street>Monroe</street>
    <city>Madison</city>
  </customer>
  <customer cust-id="C102" accounts="A-401 A-402">
    <name>Mary</name>
    <street>Erin</street>
    <city>Newark</city>
  </customer>
</bank-2>
```

**XPath (Cont.)**

- E.g., `/bank-2/customer/name` evaluated on the previous bank example returns
  <name>Joe</name>
  <name>Mary</name>
- E.g., `/bank-2/customer/name/text()` returns
  Joe
  Mary

**Path Expression Syntax**

- Input functions
  - Input(), collection(), document()
- Provide access to input data, local or remote
- Location Step
  - Axes::NodeTest[predicates]
    - Child::account[attribute::acct-num=A401]
  - Axes
    - Specify navigation direction
    - Forward: child::, descendant::, self::, ...
    - Backward: parent::, ancestor::, ...
Path Expression Syntax

- Node Test
  - Condition on nodes
  - Test for node type and node name
    - Type may be text(), …
    - Name may be Qname, *, …
- Predicates
  - Further conditions on nodes
  - Is a Boolean expression
  - Can test order position of a node, children or descendants of the node, value of a node, etc.

Evaluation of Path Expressions

- Document order is the depth-first traversal of the tree model
- Evaluation context
  - The input to a step is a sequence of items generated from the previous step
  - A context item is the item currently being evaluated
  - The outcome of evaluation of a context item is a sequence of items as well
  - The output of a step is the concatenation of the outcomes of the input items

Evaluation of Path Expression

- Path expressions are evaluated left to right
- `/bank-2/account[balance > 400]`
  - returns account elements with a balance value greater than 400
- `/bank-2/account[balance]`
  - returns account elements containing a balance subelement

Abbreviated Syntax

- Child: e.g. A/B
- Descendant-or-self: e.g. //A
- Parent: e.g., ../../../B
- Attributes are accessed using "@"
  - E.g. `/bank-2/account[balance > 400]/@account-number`
    - returns the account numbers of those accounts with balance > 400
  - IDREF attributes are not dereferenced automatically (more on this later)

Functions in XPath

- The function `count()` at the end of a path counts the number of elements in the set generated by the path
  - E.g. `/bank-2/account[customer/count() > 2]`
    - Returns accounts with > 2 customers
  - Also function for testing position (1, 2, ..) of node w.r.t. siblings
  - Boolean connectives and and or and function not() can be used in predicates

Functions in XPath

- Function `id()` is used to reference IDREFs
  - `id()` can also be applied to sets of references such as IDREFS and even to strings containing multiple references separated by blanks
  - E.g. `/bank-2/account/id(@owner)`
    - returns all customers referred to from the owners attribute of account elements.
More XPath Features
- Operator "|") used to implement union
  E.g. /bank-2/account/id/@owner) | /bank-2/loan/id/@borrower
  - gives customers with either accounts or loans
  - However, "|" cannot be nested inside other operators.
- Can bind variables $var to path expressions
- Many more expressions
  - Sequence, arithmetic, logic, for, if-then-else, quantifier, ...

XSLT: XSL Transformations
- A language to transform XML documents to other XML documents
  - Describe stylesheet that transform input tree to an output tree
  - Transformation based on templates
    - Match input tree with a pattern
    - Create output tree using matched portion of the input tree together with new structure created in a template
  - Stylesheets are defined as a special type of XML document using tags in xsl namespace

Syntax of Stylesheets
- Contains one xsl:stylesheet element
  - Its children (i.e. top-level elements) are typically elements defined in xsl namespace
- Top-level elements
  - Defines variables, parameters, templates, ...
  - Order of these elements are not significant
- A stylesheet can be simply a literal (constant value)

XSLT Templates
- XSLT transformations are expressed using rules called templates
  - Templates combine selection using XPath with construction of results
  - Example of XSLT template with match and select part
    `<xsl:template match="/bank-2/customer">`  
    `<xsl:value-of select="name"/>`  
    `</xsl:template>`  
    `<xsl:template match="*"/>`

XSLT Templates
- The match attribute of xsl:template specifies a pattern in XPath
- Elements in the XML document matching the pattern are processed by the actions within the xsl:template element
  - `xsl:value-of` selects (outputs) specified values (here, name)

XSLT Templates
- For elements that do not match any template
  - Attributes and text contents are output as is
  - Templates are recursively applied on subelements
  - The `<xsl:template match="*"/>` template matches all elements that do not match any other template
  - Used to ensure that their contents do not get output.
XSLT Templates (Cont.)
- If an element matches several templates, only one is used
  ▲ Which one depends on a complex priority scheme/user-defined priorities
  ▲ We assume only one template matches any element

Processing Model
- The input is a sequence of source nodes
- The output starts with a root node
- The result of processing a source node is appended to results of processing source nodes ahead of it
- For each source node, the best matching template is used to generate its output
- A template may select an additional list of source nodes for further processing
  ▲ recursively apply templates to the children of the current matching node

Creating XML Output
- Any text or tag in the XSL stylesheet that is not in the xsl namespace is output as is
- Example: To wrap results in new XML elements.
  `<xsl:template match="/bank-2/customer">
    <customer>
      <xsl:value-of select="name"/>
    </customer>
    <xsl:template>
      <xsl:template match="*"/>
  </xsl:template>
  </xsl:template>`
  Example output:
  `<customer> John </customer>`
  `<customer> Mary </customer>`

Creating XML Output (Cont.)
- Note: Cannot directly insert an xsl:value-of tag inside another tag
  ▲ For example, cannot create a name attribute for <customer> using
    `<customer name="xsl:value-of select="customer-name">
    ...`
  ▲ XSLT provides a construct xsl:attribute to handle this situation

Creating XML Output (Cont.)
- xsl:attribute adds attribute to the preceding element
  ▲ E.g. `<customer>
    <xsl:attribute name="customer-id">
      <xsl:value-of select="customer-id"/>
    </xsl:attribute>
  </customer>`
  results in output of the form
  `<customer customer-id="..."> ...`
- xsl:element is used to create output elements with computed names

Structural Recursion
  `<xsl:template match="/bank-2">
    <customers>
      <xsl:template apply-templates/>
    </customers>
  </xsl:template>`
  Example output:
  `<customer> John </customer>`
  `<customer> Mary </customer>`
XSLT Keys

- XSLT keys allow elements to be looked up (indexed) by values of subelements or attributes
  - Keys must be declared (with a name) and, the key() function can then be used for lookup.
    E.g.
    ```xml
    <xsl:key name="acctno" match="account" use="acct-num"/>
    <xsl:value-of select='key("acctno", "A-101")'/>
    ```

Joins in XSLT

- Keys permit (some) joins to be expressed in XSLT
  ```xml
  <xsl:template match="depositor">
    <cust-acct>
      <xsl:value-of select='key("custno", "name")'/>
      <xsl:value-of select='key("acctno", "acct-num")'/>
    </cust-acct>
  </xsl:template>
  ```

Sorting in XSLT

- Using an xsl:sort directive inside a template causes all elements matching the template to be sorted
  - Sorting is done before applying other templates
  ```xml
  <xsl:template match="/bank">
    <xsl:apply-templates select="customer"/>
    <xsl:sort select="name"/>
  </xsl:template>
  ```