Deep Comprehension of XPath and XSLT (handout)

ApacheCon Europe 05

Cliff Schmidt

Copyright 2005 Cliff Schmidt. Licensed under the Apache License, Version 2.0

Agenda

- Hello World!
- The Big Picture
- Review of XPath Specification
- XPath Exercises
- Review of XSLT Specification
- XSLT Exercises
- The Movie XML Files
- Groups in XSLT 1.0
- Using Apache Xalan
- XSLT 2.0
- Review

Hello, World!

- Input:

```
<container>
  <message greeting="Hello" to="World"/>
</container>
```

- Desired Output:

```
Hello, World!
----------------------- or ---------------------------
<HTML>
  <HEAD><TITLE>XSLT-HTML Page</TITLE></HEAD>
  <BODY><H1>Hello, World!</H1></BODY>
</HTML>
----------------------- or ---------------------------
<Hello>
  <to recipient="World"/>
</Hello>
```

Text Output

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text"/>
  <xsl:template match="message">
    <xsl:value-of select="@greeting"/>,
    <xsl:value-of select="@to"/>!
  </xsl:template>
</xsl:stylesheet>
```

HTML Output

```
<xsl:transform version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="html"/>
  <xsl:template match="message">
    <HTML>
      <HEAD><TITLE>XSLT-HTML Page</TITLE></HEAD>
      <BODY>
        <H1>
          <xsl:value-of select="@greeting"/>,
          <xsl:value-of select="@to"/>!
        </H1>
      </BODY>
    </HTML>
  </xsl:template>
</xsl:transform>
```

XML Output

```
<xsl:transform version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml" indent="yes"/>
  <xsl:template match="container">
    <xsl:element name="{message/@greeting}">
      <to recipient="{message/@to}"/>
    </xsl:element>
  </xsl:template>
</xsl:transform>
```

The Big Picture

- “Hello, World!” Observations
  - XSLT stylesheets are XML.
  - XSLT instructions are namespace-aware XML elements.
  - XSLTs start with either an xsl:stylesheet or xsl:transform element.
  - Input: XML; Output: varies (text, html, xml, etc).
    - xsl:output instruction formats content (and can insert XML/HTML content).
  - xsl:template: instantiated when matches some node of input.
  - Literals: directly output
    - must be well-formed XML.
  - Elements (and attributes) also created at run-time.
  - XPath: heavily used (e.g. match and select attributes)
    - Attribute Value Templates ({}) insert evaluation of XPath into XSLT attributes.
  - Warning: hidden default templates at work!

Hidden Default Templates

- This causes many of the beginner problems with XSLT!

-
  ```
  <xsl:template match="*|/">
     <xsl:apply-templates/>
  </xsl:template>
  ```
  - Also for each existing mode.

-
  ```
  <xsl:template match="text()|@*">
     <xsl:value-of select="."/>
  </xsl:template>
  ```

- See slide on XSLT 1.0 Specification, Section 5.8 for more on this.

Big Picture: XPath and XSLT

- XPath
  - Expression language based on namespace-aware data model
- XSLT
  - Uses XPath as expression language
  - Primarily based on three steps
    - Matching
    - Instantiation
    - Selection

XPath 1.0 Specification Outline

- See http://www.w3.org/TR/xpath
  - 1 Introduction
  - 2 Location Paths
  - 3 Expressions
  - 4 Core Function Library
  - 5 Data Model
  - 6 Conformance

1 Introduction

- From XSLT and XPointer
- Address parts of XML document
- Non-XML syntax
- Operates on abstract, logical structure
- Models XML as tree of ele/att/text nodes
- Expressions yield objects of the following types:
  - strings, numbers, booleans, **node-sets**
- Context provided by host language:
  - Context node/position/size
  - Functions, variables, and namespace bindings

1 Introduction (cont)

- Context node/position/size may change from one subexpression to the next.
- Extension functions by host language are allowed and expected.
- XPath grammar applies after XML normalization (e.g. <, ‘, “”).

2 Location Paths

- Most important type of expression
- Verbose and abbreviated syntax
  - Learn the Verbose form, then use abbreviated form!
  - See spec examples
- Relative and absolute paths
- Made up of 1+ location steps
- Selects set of nodes relative to context
- New context is union of resulting node sets

2.1 Location Steps

- Step ::=

  AxisSpecifier NodeTest Predicate*

  | AbbreviatedStep

2.2 Axes

- self, attribute, namespace,
- parent, child,
- ancestor, descendant,
- ancestor-or-self, descendant-or-self,
- preceding, following,
- preceding-sibling, following-sibling

- **NOTE:** The *ancestor*, *descendant*, *following*, *preceding* and *self* axes partition a document (ignoring attribute and namespace nodes).
  - They do not overlap and together they contain all the nodes in the document.

AxisSpecifier Production

- AxisSpecifier ::=

  AxisName ‘::’ | AbbreviatedAxisSpecifier

2.3 Node Tests

- Principal node type
  - = element, except for attribute and NS axes
- Test by QName (expanded-name)
  - Only tests for principal node.
  - <u>Default namespace (xmlns) is not used</u>.
  - '*' is true for any node of the principal node type. Also, NCName:* is allowed.
- Test by Type
  - text(), comment(),
  - processing-instruction(), node()

NodeTest Production

- NodeTest ::= NameTest

  | NodeType ‘(’ ‘)’

  | ‘processing-instruction’ ‘(’ <u>Literal</u> ‘)’

2.4 Predicates

- Forward and Reverse axes
  - underlined axes in 2.2 are reverse axes
  - Important: use document order to determine *proximity position*.
- Predicates filter the node-set
  - Evaluates the expression and converts to boolean. True stays, False is filtered out.
  - Number results are true if equal to context position; otherwise, false. As in, position().

Predicate Production

- Predicate ::= ‘[’ PredicateExpr ‘]’
- PredicateExpr ::= Expr

2.5 Abbreviated Syntax

Verbose	**Abbreviation**
child::	*omitted*
attribute::	@
self::node()	.
parent::node()	..
/descendant-or-self::node()/	//
[position()=number]	[number]

- See examples in spec

3 Expressions

- Implicit casting for functions
- Union node-sets with '|'
- Careful with booleans and node-sets (see NOTEs)
- Increasing order of precedence
  - /, //, |
  - or, and
  - =, !=
  - <=, <, >=, >
  - +, -, div, mod, *, - (unary)

- IEEE 754 for numbers

4 Core Function Library

- Describes required core functions
- Extended functions allowed
  - Must be namespace-qualified
  - Core functions belong to no namespace
- Functions for explicit casting:
  - string(), number(), and boolean()
  - Implicit conversion, otherwise

4.1 Node Set Functions

- id,
- lang,
- last,
- local-name,
- name,
- namespace-uri,
- position

4.2 String Functions

- concat,
- contains,
- normalize-space,
- starts-with,
- string,
- string-length,
- substring,
- substring-after,
- substring-before,
- translate

4.3 Boolean Functions

- boolean,
- false,
- not,
- true

4.4 Number Functions

- ceiling,
- count,
- floor,
- number,
- round,
- sum

5 Data Model

- Very important section!
- Defines a data model for XML, important supplement to Infoset spec.
- Differs from DOM data model
- Differs from XQuery/XPath2 data model

5 Data Model (cont)

- Seven types of nodes:
  - root, element, text, attribute,
  - namespace, processing instruction nodes,
  - comment nodes
- Attribute and Namespace nodes occur before children of element
- Definition of XML as tree

5.1 Root Node

- Children include document element node.
- Children might also include PIs or comment nodes

5.2 Element Nodes

- Has expanded-name by expanding QName IAW Namespaces spec
- Children include
  - element nodes
  - comment nodes
  - processing instructions
  - text nodes

5.3 Attribute Nodes

- Element is parent of each attribute node
  - But, attribute node is not a child of parent element
- NOTE that = operator test for value, not identity
- xml:lang, xml:space apply to descents, but do not repeat in tree
- Be careful with external DTDs containing defaults or entities

5.4 Namespace Nodes

- One for each namespace in scope, including xml prefix
- Element has namespace node for
  - Every attribute with xmlns:
  - Every attribute on an ancestor element with xmlns:, unless later redeclared.
  - xmlns default namespace attribute
    - Xmlns=“” undeclares the default namespace.

5.5 Processing Instruction Nodes

- Does not include PIs in DTD

5.6 Comment Nodes

- Does not include comments in DTD

5.7 Text Nodes

- Never has an immediately following or preceding sibling as text node.

6 Conformance

- Up to the host language
  - Allows subsetting/profiling

XPath Exercises: Input XML

```
<A>
  <B>
    <C>C-text
      <D>D-text</D>
    </C>
    <E start="1">Position 1.
      <F>
        <G/>
      </F>
   It is also mixed text.
      <H>H-text</H>
    </E>
    <I>I-text</I>
    <J color="many">J-text
   <K start="2">Position 2</K>
      <fruit color="red" start="2">apple</fruit>
      <fruit color="yellow">banana</fruit>
      <fruit color="yellow">pineapple</fruit>
      <fruit color="red" start="2">strawberry</fruit>
      <L/>
      more J-text
    </J>
  </B>
  <M xmlns="foo">
    <N>N-text</N>
  </M>
</A>
```

XPath Exercises: Sample

- First set two variables to different nodes in the XML tree.
  - Start #1: E
    - *name="pos1" select="/descendant-or-self::node()[attribute::start='1']"*
  - Start #2: K
    - *name="pos2" select="//*[@start='2']"*
- Samples
  - Text of the child of the preceding node
    - $pos1/preceding-sibling::node()[position()=1]/ child::*[position()=1]/child::text()
  - Abbreviation for the same expression
    - $pos1/preceding-sibling::*[1]/*[1]

XPath Exercises: Questions

- How many children does each parent have?
- What is the value of the first and last child of each parent?
- What is the string length of the text within the N-element? Use the root as the starting point this time.
- Find the second red fruit from starting point #1.
- Bonus: Can you figure out why the following two expressions do not give the same result?
  - /A/B/C/*[1]: D-text
  - /A/B/C/node()[1]: C-text

XPath Exercises: Answers

- How many children does each parent have?
  - count($pos1/../node())
- What is the value of the first and last child of each parent?
  - $pos1/../node()[1]
  - $pos1/../node()[last()]
- What is the string length of the text within the N-element? Use the root as the starting point this time.
  - xmlns:f="foo"   string-length(/A/f:M/f:N)
- Find the second red fruit from starting point #1.
  - $pos1/following::fruit [@color='red' and preceding-sibling::fruit[@color='red']]
- Bonus: Can you figure out why the following two expressions do not give the same result?
  - /A/B/C/*[1]: D-text
  - /A/B/C/node()[1]: C-text

XSLT Specification Outline

See http://www.w3.org/TR/xslt

- 0 Status of this Document
- 1 Introduction
- 2 Stylesheet Structure
- 3 Data Model
- 4 Expressions
- 5 Template Rules
- 6 Named Templates
- 7 Creating the Result Tree
- 8 Repetition
- 9 Conditional Processing
- 10 Sorting
- 11 Variables and Parameters
- 12 Additional Functions
- 13 Messages
- 14 Extensions
- 15 Fallback
- 16 Output
- 17 Conformance
- 18 Notation

0 Status of this Document

- Recommendation 16 November 1999
- Plans for Version 1.1 were canceled.

1 Introduction

- Format of XSLT
  - Expressed as well-formed XML (with Namespaces)
  - May include elements defined by XSLT as well as those not defined.
  - XSLT is called a stylesheet or a transform.

- Transformation
  - Changes source tree into a result tree.
  - Associates patterns with templates.
    - Patterns are matched against nodes in source tree.
    - Templates are instantiated to for result tree.

1 Introduction (continued)

- Templates can contain
  - literal results elements.
  - XSLT namespace elements – instructions
    - Each instruction is replaced by result tree fragment.
- Processing
  - Elements processed when selected by execution of an instruction.
  - Result tree constructed by finding template rule for root node and instantiating its template.
  - More than one template rule may have a pattern that matches a given element, but only one will be applied.

1 Introduction (continued)

- Templates always instantiated with respect to a current node and node list.
  - Most instructions pass on same context.
  - A few will change context until instruction is complete, and then revert back.
- XSLT Expression Language = XPath
- XSLT Extensions
  - Extension instruction elements
  - Extension functions

2 Stylesheet Structure

- 2.1 XSLT Namespace
- 2.2 Stylesheet Element
- 2.3 Literal Result Element as Stylesheet
- 2.4 Qualified Names
- 2.5 Forwards-Compatible Processing
- 2.6 Combining Stylesheets
- 2.7 Embedding Stylesheets

2.1 XSLT Namespace

- xmlns:xsl=http://www.w3.org/1999/XSL/Transform
- Versioning based on “version” attribute, not on XSLT namespace.
- Elements from XSLT namespace are only recognized in stylesheet, not in source doc.
- Elements and attributes from other namespaces may be used in stylesheets.
  - Attributes must not have null namespace.

2.2 Stylesheet Element

- <xsl:stylesheet
  id = *id*
  extension-element-prefixes = *tokens*
  exclude-result-prefixes = *tokens*
  version = *number*>
  <!-- Content: (xsl:import*, *top-level-elements*) -->
  </xsl:stylesheet>

  - xsl:transform is synonym

2.2 Stylesheet Element (continued)

- May contain following “top-level” elements:
  - xsl:import, xsl:include
  - xsl:strip-space, xsl:preserve-space
  - xsl:output, xsl:key, xsl:decimal-format,
  - xsl:namespace-alias, xsl:attribute-set,
  - xsl:variable, xsl:param,
  - xsl:template

  - Order is not significant, except xsl:import must be first.

2.3 Literal Result Element as Stylesheet

- "Simplified Syntax" consisting of only a single template for the root node.
- Equivalent to a stylesheet with xsl:stylesheet element containing template rule matching pattern, "/".
- See example in spec.
- Document element must have xsl:version attribute.

2.4 Qualified Names

- All XSLT objects are named with QNames using in-scope namespace declarations.

- Default namespace is not used for unprefixed names.

2.5 Forwards-Compatible Processing

- Forwards-compatible mode occurs when version attribute does not equal “1.0”.
  - within stylesheet element
  - or document element of simplified syntax.
- New top-level elements are ignored.
- New elements in templates perform fallback if instantiated.
- New attributes must be ignored.
- See example in spec and NOTE regarding terminating processing when involving crucial elements.

2.6 Combining Stylesheets

- Stylesheet Inclusion
  - <xsl:include
    href = *uri-reference* />
  - Relative URI resolved to base (not XML Base).
  - Works at tree level.
  - Children of the xsl:stylesheet element replace xsl:include element.
  - Error if directly or indirectly includes self.
  - See diamond problem (D includes B and C, which each include A)

2.6 Combining Stylesheets (continued)

- Stylesheet Import
  - <xsl:import
    href = *uri-reference* />
  - Same as include, except definitions and template rules of importing document take precedence over imported document.
  - Must precede all other top-level children.
  - Including import instructions moves them up prior to rest of include.
  - Include elements are resolved before constructing import tree.
  - Import tree defines import precedence as post-order.

2.7 Embedding Stylesheets

- xsl:stylesheet tag may not be top level element.
- Reference to stylesheet may include fragment identifier with ID attribute.
- See example in spec.

3 Data Model

- 3.1 Root Node Children
- 3.2 Base URI
- 3.3 Unparsed Entities
- 3.4 Whitespace Stripping

- Data model is same as XPath, with a few additions.
- XSLT operates on source, result, and stylesheet documents using same model.
- PIs and comments are ignored in stylesheets.

3.1 Root Node Children

- Normal restriction of children of root node is relaxed for result tree.
- When writing out to XML, possible to not be well-formed XML.
- When source tree is parsed, it will automatically satisfy normal restrictions;
  - but, if created synthetically, restrictions are also relaxed.

3.2 Base URI

- Every node as an associated base URI.
- Base URI of elements and PIs is the base URI of document.
  - unless in an external entity, use that one.
- Base URI of text, comment, attribute, and namespace node is base URI of parent node.
- None of this refers to the XML Base spec, which became a recommendation two years later.

3.3 Unparsed Entities

- Root node has a mapping giving URI for each unparsed entity declared in DTD.
  - URI generated from system identifier and public identifier.

3.4 Whitespace Stripping

- After a tree for a source document or stylesheet is constructed, but before processing, some text nodes are stripped:
  - must contain only whitespace
  - must not occur on elements whose names have been identified for whitespace preservation.
  - must not occur on elements with ancestors having xml:space = “preserve” (closer than an xml:space=“default”.
  - otherwise, text node is stripped.

- <xsl:strip-space **elements** = *tokens* />
- <xsl:preserve-space **elements** = *tokens* />

4 Expressions

- XSLT uses expression language defined by XPath:
  - selecting nodes for processing;
  - specifying conditions for different ways of processing a node;
  - generating text to be inserted in the result tree.
- Outermost expression gets context:
  - Context node/position/size comes from current node and node list.
  - Variable bindings from those in-scope on elements/attributes containing expression.
  - Namespaces treated same way, including xml. Default namespace is not passed on.
  - Function library includes XSLT Additional Functions and Extension Functions.

5 Template Rules

- 5.1 Processing Model
- 5.2 Patterns
- 5.3 Defining Template Rules
- 5.4 Applying Template Rules
- 5.5 Conflict Resolution for Template Rules
- 5.6 Overriding Template Rules
- 5.7 Modes
- 5.8 Built-in Template Rules

5.1 Processing Model

- Result tree starts by processing a list containing just the root node.
- A list of source nodes is processed and appended to the result tree structure.
- Each node processed by finding all template rules with matching patterns and choosing best.
  - Chosen rule's template is then instantiated with node as current node and list of source nodes as current node list.
  - Template typically contains instructions to select additional list of source nodes.
- Matching, instantiation, and selection continues recursively.

5.2 Patterns

- Template rules identify nodes by applying a pattern.
- Syntax for patterns is a subset of expressions:
  - Set of location paths separated by "|"
  - Only use child or attribute axes, or "/" and "//".
  - Predicates can still contain full expression.
- Patterns evaluate to node-sets.
- Node matches pattern if included in node-set for some possible context.
- See examples in spec.

5.3 Defining Template Rules

- <xsl:template
  match = *pattern*   name = *qname*
  priority = *number*   mode = *qname*>
  <!-- Content: (xsl:param*, *template*) -->
  </xsl:template>

- "match" attribute identifies applicable source nodes.
  - Required unless template has a "name".
  - Error for match to contain variable reference.

5.4 Applying Template Rules

-
  ```
  <xsl:apply-templates
    select = node-set-expression    mode = qname>
    <!-- Content: (xsl:sort | xsl:with-param)* -->
  </xsl:apply-templates>
  ```

- In the absence of the “select” attribute, xsl:apply-templates processes all children of the current node, including text nodes.

- Value of “select” attribute is an expression, which must evaluate to a node-set.

- May select descendents or ancestors (unusual and dangerous).

- See examples in spec.

5.5 Conflict Resolution for Template Rules

- Lower import precedence rules are thrown out.
- Next, highest template priority is found. If not explicit, default is used:
  - Multiple alternatives ("|") are treated as separate rules.
  - QName or PI test: priority = 0
  - NCName:*: priority = -0.25
  - NodeTest: priority = -0.50
  - Otherwise, priority = 0.50
- If still more than one matching rule, either error or choose last occurrence in stylesheet.

5.6 Overriding Template Rules

- <xsl:apply-imports />
- Invokes overridden template rule.
  - Similar to derived class calling base class in OOP.
- Current template rule
  - becomes null during xsl:for-each iterations.
- See example in spec.

5.7 Modes

- Modes allow an element to be processed multiple times, each time producing a different result.
- “mode” attribute is QName, only allowable with “match” attribute.
- Template only applied if “mode” value of caller and template match, or if both absent.

5.8 Built-in Template Rules

- <xsl:template match="*|/">
     <xsl:apply-templates/>
  </xsl:template>
  - Also for each existing mode.

- <xsl:template match="text()|@*">
     <xsl:value-of select="."/>
  </xsl:template>

- <xsl:template match="processing-instruction()|comment()"/>

- Also, built-in namespace node template does nothing. No way to match a namespace node.

- Built-in templates treated as if imported and can be overridden.

6 Named Templates

- <xsl:call-template
  name = *qname*>
  <!-- Content: xsl:with-param* -->
  </xsl:call-template>

- Templates can be invoked by name.

- May or may not also have a “match” attribute.

- Xsl:call-template does not change the current node or current node list (as is true for xsl:apply-templates).

- “match” and “name” do not interfere with each other.

7 Creating the Result Tree

- 7.1 Creating Elements and Attributes
- 7.2 Creating Text
- 7.3 Creating Processing Instructions
- 7.4 Creating Comments
- 7.5 Copying
- 7.6 Computing Generated Text
- 7.7 Numbering

7.1 Creating Elements and Attributes

- 7.1.1 Literal Result Elements
- 7.1.2 Creating Elements with xsl:element
- 7.1.3 Creating Attributes with xsl:attribute
- 7.1.4 Names Attribute Sets

7.1.1 Literal Result Elements

- Any element in a template that
  - does not belong to XSLT namespace (nor extension)
  - is instantiated to create an element node
  - with same expanded-name
  - and a copy of namespace nodes,
  - unless included in exclude-result-prefixes attribute (especially useful when only needed to address source tree).

- Attributes of literal result elements can be interpreted as {attribute value templates}.

- <xsl:namespace-alias
  stylesheet-prefix = *prefix* | "#default"
  result-prefix = *prefix* | "#default" />
  - Translates from stylesheet use to result tree.
    See examples in spec.

7.1.2 Creating Elements with xsl:element

- <xsl:element
  name = { *qname* }
  namespace = { *uri-reference* }
  use-attribute-sets = *qnames*>
  <!-- Content: *template* -->
  </xsl:element>

- Allows elements to be created with a computed name.

- “name” and “namespace” attributes can be {attribute value templates}.

7.1.3 Creating Attributes with xsl:attribute

- <xsl:attribute
  name = { *qname* }
  namespace = { *uri-reference* }>
  <!-- Content: *template* -->
  </xsl:attribute>

- Adds attribute node to containing result element, regardless of whether created by literals or xsl:element.

- Errors:
  - Adding to element after children have been added.
  - Adding to a node that is not an element.
  - Creating nodes other than text nodes during instantiation of content of xsl:attribute.

7.1.4 Names Attribute Sets

-
  ```
  <xsl:attribute-set
    name = qname
    use-attribute-sets = qnames>
    <!-- Content: xsl:attribute* -->
  </xsl:attribute-set>
  ```
- Defines a named set of attributes.
- Content consists of zero or more xsl:attribute elements.
- Used by specifying a “use-attribute-sets” attribute on xsl:element, xsl:copy, and xsl:attribute-set, or even literal result element.
- Error to use directly/indirectly on same set.

7.1.4 Names Attribute Sets (continued)

- Multiple occurrences of same name are merged:
  - First, by order of use-attributue-sets listed in element.
  - Next, by attributes specified on literal result elements.
  - Finally, by any attributes specified by xsl:attribute.
  - Multiple copies will be overridden with latest, according to order above.
- Only variables and parameters declared top-level are visible.
- Multiple copies of same attribute set are merged.

7.2 Creating Text

- Template can contain text nodes, which gets created in result tree.
  - Processed at tree level. “&lt;” => “<”

- May also wrap text in xsl:text element instruction.
  - Wrapping may change whitespace processing.

  -
    ```
    <xsl:text
      disable-output-escaping = "yes" | "no">
      <!-- Content: #PCDATA -->
    </xsl:text>
    ```

7.3 Creating Processing Instructions

- <xsl:processing-instruction **name** = { *ncname* }>
  <!-- Content: *template* -->
  </xsl:processing-instruction>

- For example, this
  - <xsl:processing-instruction name="xml-stylesheet">
    href="book.css" type="text/css"
    </xsl:processing-instruction>

  would create the processing instruction
  - <?xml-stylesheet href="book.css" type="text/css"?>

- Content may not contain “?>”

7.4 Creating Comments

- <xsl:comment>
  <!-- Content: *template* -->
  </xsl:comment>

- For example, this
  - <xsl:comment>This file is automatically generated. Do not edit!
    </xsl:comment>

  would create the processing instruction
  - <!--This file is automatically generated. Do not edit!-->

- Content may not contain “--”

7.5 Copying

- <xsl:copy
  use-attribute-sets = *qnames*>
  <!-- Content: *template* -->
  </xsl:copy>

- Instantiating the xsl:copy element creates a copy of the current node,
  - including in-scope namespace nodes,
  - not including attributes and children.

- Content of element is a template for the attributes and children of the created node.
  - (instantiated only for node types that can have children)

7.6 Computing Generated Text

- <xsl:value-of
  select = *string-expression*
  disable-output-escaping = "yes" | "no" />

- Instantiated to create text node in result tree.

- Used to compute generated text. Expressions can also be used inside attribute values of literal result elements by enclosing expression in curly braces ({}).

- See security example in spec.

7.6.2 Attribute Value Templates

- Surrounded by curly braces ({})
- Instantiated by replacing expression and braces with result of evaluating expression and casting to string.
- Attributes cannot use attribute value templates when:
  - Attribute value is expression or pattern.
  - Attribute is of top-level element.
  - Attributes referring to XSLT objects.
  - xmlns
- See example in spec.

7.7 Numbering

- <xsl:number
  level = "single" | "multiple" | "any"
  count = *pattern*
  from = *pattern*
  value = *number-expression*
  format = { *string* }
  lang = { *nmtoken* }
  letter-value = { "alphabetic" | "traditional" }
  grouping-separator = { *char* }
  grouping-size = { *number* } />
- Inserts a formatted number into the result tree.
- Value attribute contains expression, if missing, inserts number of current node position.
- See spec for details.

8 Repetition

- <xsl:for-each
  select = *node-set-expression*>
  <!-- Content: (xsl:sort*, *template*) -->
  </xsl:for-each>

- Used when result has a known regular structure.

- Contains a template, which is instantiated for each node selected by “select” expression.

9 Conditional Processing

- 9.1 Conditional Processing with xsl:if
- 9.2 Conditional Processing with xsl:choose

9.1 Conditional Processing with xsl:if

- <xsl:if
  test = *boolean-expression*>
  <!-- Content: *template* -->
  </xsl:if>

- If “test” expression is true, template is instantiated.

9.2 Conditional Processing with xsl:choose

- <xsl:choose>
  <!-- Content: (xsl:when+, xsl:otherwise?) -->
  </xsl:choose>

- <xsl:when
  test = *boolean-expression*>
  <!-- Content: *template* -->
  </xsl:when>

- <xsl:otherwise>
  <!-- Content: *template* -->
  </xsl:otherwise>

- Content of the first, and only the first, xsl:when element whose test is true is instantiated.

10 Sorting

- <xsl:sort
  select = *string-expression*
  lang = { *nmtoken* }
  data-type = { "text" | "number" | *qname-but-not-ncname* }
  order = { "ascending" | "descending" }
  case-order = { "upper-first" | "lower-first" } />

- Exists as child of
  - xsl:apply-templates
  - or xsl:for-each (must occur as first child for xsl:for-each).

- Sorts according to specified sort keys, and then processes the in sorted order.

11 Variables and Parameters

- 11.1 Result Tree Fragments
- 11.2 Values of Variables and Parameters
- 11.3 Using Values of Variables and Parameters with xsl:copy-of
- 11.4 Top-level Variables and Parameters
- 11.5 Variables and Parameters within Templates
- 11.6 Passing Parameters to Templates

11 Variables and Parameters (continued)

- <xsl:variable
  name = *qname*
  select = *expression*>
  <!-- Content: *template* -->
  </xsl:variable>

- <xsl:param
  name = *qname*
  select = *expression*>
  <!-- Content: *template* -->
  </xsl:param>

- Accepts object of any type that can be returned by an expression.
- Difference between xsl:variable and xsl:param is that xsl:param is only a default value for the binding.

11.1 Result Tree Fragments

- Variables introduce an additional data-type into the expression language:
  - result tree fragment
  - (in addition to string, number, boolean, node-set)
  - Treated equivalently to node-set with a single root node.
  - Operations permitted are subset:
    - Only those that would be permitted on a string.
    - (no “/”, “//”, or “[ ]” on result tree fragments.
- Expressions return result tree fragment by
  - Referencing variables of type result tree fragment
  - Calling extension functions
  - Getting system property whose value is a result tree fragment.

11.2 Values of Variables and Parameters

- Three ways to specify value of variable:
  - “select” attribute expression;
  - content of variable binding element;
  - otherwise, empty string.
- See NOTE in spec of what not to do.

11.3 Using Values of Variables and Parameters with xsl:copy-of

- <xsl:copy-of
  select = *expression* />

- Inserts result tree fragment into result tree without convert to string as xsl:value-of does.
- Copying includes the attributes, namespaces, and children of element.
- deep-copy, unlike xsl:copy

11.4 Top-level Variables and Parameters

- Allowed as top-level elements, making them global.
- Top-level xsl:param elements serve as parameters to the stylesheet itself.
- Implementation-dependent how this is passed in.
- Circular definitions not allowed.

11.5 Variables and Parameters within Templates

- xsl:param is allowed as a child at the beginning of an xsl:template.

11.6 Passing Parameters to Templates

- <xsl:with-param
  name = *qname*
  select = *expression*>
  <!-- Content: *template* -->
  </xsl:with-param>

- Allowed within
  - xsl:apply-templates
  - xsl:call-template

12 Additional Functions

- 12.1 Multiple Source Documents
- 12.2 Keys
- 12.3 Number Formatting
- 12.4 Miscellaneous Additional Functions

12.1 Multiple Source Documents

- **Function:** *node-set* **document**(*object*, *node-set*?)

- The document function allows access to XML documents other than the main source document.

12.2 Keys

- <xsl:key
  name = *qname*
  match = *pattern*
  use = *expression* />

- **Function:** *node-set* **key**(*string*, *object*)

12.3 Number Formatting

- **Function:** *string* **format-number**(*number*, *string*, *string*?)

- See spec for details.

12.4 Miscellaneous Additional Functions

- **Function:** *node-set* **current**()
- **Function:** *string* **unparsed-entity-uri**(*string*)
- **Function:** *string* **generate-id**(*node-set*?)
- **Function:** *object* **system-property**(*string*)

13 Messages

-
  ```
  <xsl:message
    terminate = "yes" | "no">
    <!-- Content: template -->
  </xsl:message>
  ```

- Implementation may choose to pop up an alert box or simply write to a log.
- If terminate = “yes”, XSLT processor must terminate. Default is “no”.

14 Extensions

- Extension Elements
- Extension Functions

15 Fallback

- <xsl:fallback>
  <!-- Content: *template* -->
  </xsl:fallback>

- **Function:** *boolean* **element-available**(*string*)

- **Function:** *boolean* **function-available**(*string*)

16 Output

- <xsl:output
  method = "xml" | "html" | "text" | *qname-but-not-ncname*
  version = *nmtoken*
  encoding = *string*
  omit-xml-declaration = "yes" | "no"
  standalone = "yes" | "no"
  doctype-public = *string*
  doctype-system = *string*
  cdata-section-elements = *qnames*
  indent = "yes" | "no"
  media-type = *string* />

17 Conformance

- Conforming XSLT processor need not be able to output result in XML or in any other form.

- Must signal errors except for those disallowed in spec.

18 Notation

- Attribute is required only if its name is in bold.

XSLT Exercises: Document Example

```
<doc>
<!--inside beginning of doc-->
  <title>Document Title</title>
  <chapter>Chapter One Text
  <?my-pi info="here"?>
    <title>Intro</title>
    <section>
       <title>Section A</title>
       <para>This is a test.</para>
       <note>a note.</note>
    </section>
    <section>
      <title>Section B</title>
      <para>This is <emph>another</emph> test.</para>
      <note>another note.</note>
    </section>
  </chapter>
```

XSLT Exercises

```
<chapter>Chapter Two Text
  <?my-pi info="here"?>
  <title>Middle</title>
  <section>
    <title>Section A</title>
    <para>This is a test.</para>
    <note>a note.</note>
  </section>
  <section>
     <title>Section B</title>
     <para>This is <emph>another</emph> test.</para>
     <note>This is another note.</note>
   </section>
 </chapter>
```

XSLT Exercises

```
<chapter>Chapter Three Text
  <?my-pi info="here"?>
    <title>Conclusion</title>
    <section>
      <title>Section A</title>
      <para>This is a test.</para>
      <note>This is a note.</note>
    </section>
    <section>
      <title>Section B</title>
      <para>This is <emph>another</emph> test.</para>
      <note>This is another note.</note>
    </section>
  </chapter>
</doc>
```

Transform to HTML

```
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns="http://www.w3.org/TR/xhtml1/strict">
<xsl:strip-space elements="doc chapter section"/>
<xsl:output method="xml” indent="yes‟ encoding="iso-8859-1"/>
<!--    BEGIN  Built-in templates -->
<xsl:template match="*|/"> <!-- Also for each existing mode. -->
  <xsl:apply-templates/>
</xsl:template>
<xsl:template match="text()|@*">
  <xsl:value-of select="."/>
</xsl:template>
<xsl:template match="processing-instruction()|comment()"/>
<!--   END   Built-in templates -->
```

Transform to HTML

```
<xsl:template match="doc">
 <html>
   <head><title>
       <xsl:value-of select="title"/>
     </title></head>
   <body>
     <xsl:apply-templates/>
   </body>
 </html>
</xsl:template>
<xsl:template match="doc/title">
  <h1>
    <xsl:apply-templates/>
  </h1>
</xsl:template>
<xsl:template match="chapter/title">
  <h2>
    <xsl:apply-templates/>
  </h2>
</xsl:template>
```

Transform to HTML

```
<xsl:template match="section/title">
  <h3>
    <xsl:apply-templates/>
  </h3>
</xsl:template>
<xsl:template match="section/title/text()">
   There was a section here
</xsl:template>
<xsl:template match="para">
  <p><xsl:apply-templates/></p>
</xsl:template>
<xsl:template match="note">
  <p class="note"><b>NOTE: </b><xsl:apply-templates/></p>
</xsl:template>
<xsl:template match="emph">
  <em>
    <xsl:apply-templates/>
  </em>
</xsl:template></xsl:stylesheet>
```

XSLT Exercises: Data Example

```
<sales>
        <division id="North">
                <revenue>10</revenue>
                <growth>9</growth>
                <bonus>7</bonus>
        </division>
        <division id="South">
                <revenue>4</revenue>
                <growth>3</growth>
                <bonus>4</bonus>
        </division>
        <division id="West">
                <revenue>6</revenue>
                <growth>-1.5</growth>
                <bonus>2</bonus>
        </division>
</sales>
```

Transform to HTML

```
<html xsl:version="1.0"
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
     lang="en">
    <head>
   <title>Sales Results By Division</title>
    </head>
    <body>
   <table border="1">
      <tr>
          <th>Division</th>
          <th>Revenue</th>
          <th>Growth</th>
          <th>Bonus</th>
      </tr>
```

Transform to HTML

```
        <xsl:for-each select="sales/division">
            <!-- order the result by revenue -->
            <xsl:sort select="revenue"
                      data-type="number"
                      order="descending"/>
            <tr>
                <td><em><xsl:value-of select="@id"/></em></td>
                  <td><xsl:value-of select="revenue"/></td>
                  <td>
                      <!-- highlight negative growth in red -->
                      <xsl:if test="growth &lt; 0">
                          <xsl:attribute name="style">
                                <xsl:text>color:red</xsl:text>
                          </xsl:attribute>
                      </xsl:if>
                      <xsl:value-of select="growth"/>
                  </td>
                  <td><xsl:value-of select="bonus"/></td>
             </tr>
        </xsl:for-each>
    </table>
     </body>
</html>
```

Final Exam: Normalized Movie XML

```
<normalizedMovies>
  <myTopMovies>
    <row title_id="89380" Rating="1"/>
 </myTopMovies>
 <ratingLabels>
    <row rating_id="1" rating_label="Top 25"/>
    <row rating_id="2" rating_label="Top 50"/>
 </ratingLabels>
  <movieIdTitle>
    <row title_id="2374" title="8 1/2 (1963)"/>
 </movieIdTitle>
  <peopleIdName>
    <row name_id="955" name="Abraham, F. Murray"/>
 </peopleIdName>
  <peopleMovies>
    <row name_id="955" title_id="10648" role="1"/>
 </peopleMovies>
</normalizedMovies>
```

Relatively simple mapping from database tables using SQL/XML.

Final Exam: Desired XML

```
<denormalizedMovies>
  <movie rating="Top 100" id="2374" year="1963"
          <title>8 1/2 (1963)</title>
          <directors>
            <director id="125072">Fellini, Federico</director
          </directors>
          <actors>
           <actor id="257174">Mastroianni, Marcello</actor>
           <actor id="482403">Cardinale, Claudia</actor>
          </actors>
  </movie>
  <movie rating="Top 25" id="89380" year="2003">
          <title>Eternal Sunshine of the Spotless Mind (2003)</title>
          <directors>
           <director id="800447">Gondry, Michel</director>
          </directors>
          <actors>
           <actor id="63170">Carrey, Jim</actor>
           <actor id="708982">Winslet, Kate</actor>
          </actors>
  </movie>
</denormalizedMovies>
```

Final Exam: Solution

```
<xsl:stylesheet version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform" xmlns="">
<xsl:output
   method="xml"
   indent="yes"
   encoding="iso-8859-1"
/>

<xsl:template match="normalizedMovies">
  <denormalizedMovies>
    <xsl:apply-templates select="movieIdTitle"/>
  </denormalizedMovies>
</xsl:template>
```

Final Exam: Solution

```
<xsl:template match="movieIdTitle/row">
  <xsl:variable name="rating"  select="../../myTopMovies/row
                                              [@title_id=current()/@title_id]/@Rating"/>
  <movie id="{@title_id}"
    rating="{../../ratingLabels/row[@rating_id=$rating]/@rating_label}">
    <xsl:attribute name="year">
      <xsl:call-template name="outputYearFromTitle">
        <xsl:with-param name="titleString" select="@title"/>
      </xsl:call-template>
    </xsl:attribute>
    <title><xsl:value-of select="@title"/></title>
    <directors>
      <xsl:apply-templates select="../../peopleMovies/row
                                    [@title_id=current()/@title_id and @role=0]"/>
    </directors>
    <actors>
      <xsl:apply-templates select="../../peopleMovies/row
                                    [@title_id=current()/@title_id and @role=1]"/>
    </actors>
  </movie>
</xsl:template>
```

Final Exam: Solution

```
<xsl:template name="outputYearFromTitle">
  <!--  This template uses recursion to remove all, but the text inside last parentheses.
        It should work fine on titles such as:
  Heaven Is What I've Done (For My Fellow Beings) (1984)
  Hollywood (and Vine) (1993)
  How to Save a Marriage (And Ruin Your Life) (1968)
  Slapstick (Of Another Kind) (1982)   -->
  <xsl:param name="titleString"/>
  <xsl:choose>
    <xsl:when test="contains($titleString,'(')">
      <xsl:call-template name="outputYearFromTitle">
        <xsl:with-param name="titleString" select="substring-after($titleString, '(')"/>
      </xsl:call-template>
    </xsl:when>
    <xsl:otherwise>
      <xsl:value-of select="substring-before($titleString, ')')"/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

Final Exam: Solution

```
<xsl:template match="peopleMovies/row">
  <xsl:choose>
    <xsl:when test="@role=0">
      <director id="{@name_id}">
        <xsl:value-of select="ancestor::normalizedMovies/peopleIdName/row
                         [@name_id=current()/@name_id]/@name"/>
      </director>
    </xsl:when>
    <xsl:when test="@role=1">
      <actor id="{@name_id}">
        <xsl:value-of select="ancestor::normalizedMovies/peopleIdName/row
                         [@name_id=current()/@name_id]/@name"/>
      </actor>
    </xsl:when>
  </xsl:choose>
</xsl:template>
</xsl:stylesheet>
```

The Problem with Grouping

- Grouping is not easy in XSLT 1.0!
  - Much more straight-forward in XSLT 2.0
- Cannot group using a combination of:

```
<xsl:sort select="@rating">

<xsl:if test="@rating!=preceding-
  sibling::movie[1]/@rating">
```

- Why?
  - preceding-sibling axis is based on document order, not sorted order.

Grouping Solution

- Muench Method
  - Define a key for the node to be grouped.
  - Select all nodes to group.
  - Generate a unique value for each group node and select all other matching nodes.

Advanced Final Exam: Desired XML

```
<moviesGroupedByRating>
   <Top-25 debugRatingId="1">
            <movie year="1934" id="140838">
                        <title>It Happened One Night (1934)</title>
                        <directors>
                             <director id="61043">Capra, Frank</director>
                        </directors>
                        <actors>
                             <actor id="138146">Gable, Clark</actor>
                             <actor id="492357">Colbert, Claudette</actor>
                        </actors>
            </movie>…
   </Top-25>
   <Top-50>…
   </Top-50>…
```

Advanced Final Exam: Solution

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:key name="ratingKey" match="movie" use="@rating"/>
  <xsl:template match="/">
    <xsl:variable name="uniqueRatings" select="/denormalizedMovies/movie[generate-
   id(.)=generate-id(key('ratingKey',@rating))]/@rating"/>
    <moviesGroupedByRating>
      <xsl:for-each select="$uniqueRatings">
                    …see next slide
       </xsl:for-each>
    </moviesGroupedByRating>

  </xsl:template>
    <xsl:template match="movie">
    <movie id="{@id}" year="{@year}">
      <xsl:copy-of select="node()"/>
    </movie>
  </xsl:template>
</xsl:stylesheet>
```

Advanced Final Exam: Solution

```
<xsl:for-each select="$uniqueRatings">
   <xsl:sort select="document('movies.xml')/normalizedMovies/
                     ratingLabels/row[@rating_label=current()]/@rating_id”
                     order="ascending" data-type="number"/>
  <xsl:element name="{translate(.,' ','-')}">
          <xsl:attribute name="debugRatingId">
                  <xsl:value-of select="document('movies.xml')/
                          normalizedMovies/ratingLabels/row
                          [@rating_label=current()]/@rating_id"/>
          </xsl:attribute>
          <xsl:apply-templates select="key('ratingKey',.)">
                  <xsl:sort select="@year" order="ascending" data-type="number"/>
          </xsl:apply-templates>
   </xsl:element>
</xsl:for-each>
```

Apache Xalan: Programmatically

```
import org.apache.xalan.xslt.*;
import org.apache.xalan.xpath.*;
import org.apache.xalan.xpath.xml.*;
import org.apache.xalan.xpath.xdom.*;
XSLTProcessor proc = XSLTProcessorFactory.getProcessor();
XSLTInputSource xmlSource =
  new XSLTInputSource(“normalizedMovieList.xml");
XSLTInputSource xslt = new
  XSLTInputSource(“denomalizeMovies.xsl");
xdom.XercesLiaison xl = new XercesLiaison();
Document xmlTarget = xl.createDocument();
processor.process(xmlSource, xslt, xmlTarget);
```

Apache Xalan: Command Line

- Precompiling a stylesheet

```
java org.apache.xalan.xslt.Process
  -in input.xml
  -xsl transformToHtml.xsl
  -out output.html
  -lxcout transformToHtml.style

java org.apache.xalan.xslt.Process
  -in input.xml
  -lxcin transformToHtml.style
  -out output.html


java org.apache.xalan.xslt.Process -IN movies.xml -
  XSL denormalize.xsl -OUT outmovies.xml
```

Apache Cocoon Overview

- XML Publishing Framework
- Use XML and XSLT to generate format of choice (HTML,SVG,PDF)
- Built on Avalon
- Separation of Concerns:
  - Logic
  - Content
  - Style
  - Management
- For
  - Developers (logic)
  - Business Analysts (content)
  - Designers (style)
  - Administrators (management)

Apache Cocoon Basics

- Fundamental Model
  - Request is analyzed by sitemap against matches
  - Response pipeline is selected and constructed
  - Response is generated and returned
- Pipelines – connected by SAX events
  - Generators: generate events from source
  - Transformers: transform incoming events into outgoing events
  - Serializers: consume events and generate stream
- SiteMaps
  - see next slide

Cocoon: SiteMap (management)

```
<map:pipeline>
  <map:match pattern="">
    <map:redirect-to uri="home.html"/>
  </map:match>
  <map:match pattern="**.xml">
    <map:generate src="docs/{1}.xml"/>
    <map:serialize type="xml"/>
  </map:match>
  <map:match pattern="**.html">
    <map:generate src="docs/{1}.xml"/>
    <map:transform src="stylesheets/apache.xsl"/>
    <map:serialize/>
  </map:match>
  <map:match pattern="images/**.gif">
   <map:read src="resources/images/{1}.gif" mime-type="image/gif"/>
  </map:match>
</map:pipeline>
```

Cocoon: Creating a Page

```
<document>
  <header>
    <title>Department</title>
  </header>
  <body>
    <s1 title="Create a Department">
      <form handler="create-dept.html">
        <p>
          You can create a department by typing in the
          name and pressing the "submit" button.
        </p>
        <p>
          Name: <text name="name" size="30" required="true"/>
        </p>
        <submit name="Create Department"/>
        <note>
          * These fields are required.
        </note>
      </form>
    </s1>
  </body>
</document>
```

Cocoon: Mapping to Server Pages

ApacheCon Europe 05

```
<map:match pattern="*-dept.html">
  <map:act set="process">
    <map:parameter name="descriptor"
                   value="context://docs/department-form.xml"/>
    <map:parameter name="form-descriptor"
                   value="context://docs/department-form.xml"/>
    <map:generate type="serverpages" src="docs/confirm-dept.xsp"/>
    <map:transform src="stylesheets/apache.xsl"/>
    <map:serialize/>
  </map:act>
  <map:generate type="serverpages" src="docs/{1}-dept.xsp"/>
  <map:transform src="stylesheets/apache.xsl"/>
  <map:serialize/>
</map:match>
```

XSLT version 2.0

- Status
  - Latest Working Draft: 5 November 2004
  - Previously, Last Call: 12 November 2003 (109 comments)
- Based on XPath 2.0 Data Model
  - sequence of nodes and atomic types
  - Schema-aware; may declare types of variables/params/return types
  - Stylesheet can attach type annotations to output for validation
  - wide range of functions and operators
- Sequences
  - similar to node sets, but
    - ordered
    - allow duplicates
    - can contain atomic values as well as nodes
  - use xsl:sequence to build by iteration – less need for recursion

XSLT version 2.0

- Grouping
  - new xsl:for-each-group instruction
  - new current-group() and current-grouping-key() functions
- User-defined functions within stylesheet
  - can call from XPath expressions
- Regular Expression Matching
  - xsl:analyze-string
- New collation attribute in xsl:sort element
  - allows sorting using a user-defined collation
- Temporary Trees replace Result tree fragments
  - can now be accessed with location path
- Multiple outputs
- New XHTML output method

XSLT version 2.0

- New ‘use-when’ attribute
  - allows compile-time conditional inclusion of sections of the stylesheet depending on the processing environment
  - e.g., for schema-aware or non-schema-aware processing
- New xsl:next-match
  - allows multiple template rules to be applied to the same source node
- New xsl:character-map declaration
  - control the serialization of individual characters
  - replacement for some use cases requiring disable-output-escaping
- Tunnel Parameters
  - affect an entire phase of the transformation
  - without requiring them to be passed explicitly in every template call
- Transformation invoked by calling named template
  - potential for transformation to process large collections of input documents

Review & Questions

- XPath and XSLT. Remember:
  - Matching, Instantiation, Selection
  - XPath expressions, data model, and short cuts – know them!
  - Built-in templates – know them!
  - Recursion in named templates
  - Grouping is hard, but can be done with Muench method
- Xalan
- XSLT 2.0