Data-centric XML

XSLT

The Extensible Stylesheet Language

- XSL (Extensible Stylesheet Language) is divided into two parts
  - XSL-FO: Formatting Objects (out of context for data-centric XML)
  - XSLT: Transformations
- XSLT is an XML application
  - Documents are called *stylesheets*
- Specified in http://www.w3.org/TR/xslt (XSLT 1.0)
- XSLT 2.0: http://www.w3.org/TR/xslt20/
  - based on XPath 2
  - includes support for Schema types
  - not discussed here; read Annex J in spec for a list of changes

XSLT Overview

- XSLT forms a programming language
  - Stylesheet is the program, operates on input XML document
  - structured into *template rules*
    - Each template rule has a pattern an a template
    - processor compares input elements with each pattern
    - if a template rule matches, the template is written to the output tree
    - resulting output tree is serialized as XML, HTML, or plain text
- All XSLT elements are in the xsl namespace
  - http://www.w3.org/1999/XSL/Transform
  - Typically associated with “xsl” prefix
- Document elements are either xsl:stylesheet or xsl:transform
  - Both elements have identical meaning

Minimum Stylesheet

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
</xsl:stylesheet>
```

- Version attribute is required
- Stylesheet produces as output all text nodes, omitting all markup

Stylesheet Processors

- XSLT processors read templates and input documents, produce output documents
- Can be stand-alone, embedded in programming language, embedded into web browser, web server
- Multiple implementations
  - SAXON (Michael Kay)
  - Xalan (Apache Foundation)
  - MSXML (Microsoft)
  - xsltproc (Daniel Veillard, GNOME)
  - 4XSLT (Fourthought)
  - …
- Web browsers honor xsl-stylesheet processing instruction in XML prolog
  - <?xml-stylesheet type="application/xml" href="http://www.oreilly.com/styles/people.xsl"?>

Template Rules

- xsl:template
- Attribute “match” is an XPath expression (the *pattern*)
  - More strictly, matches “Pattern” production
  - A node *matches* a pattern if it is member of the selected node-set, with the “current” node or one of its ancestors as the context node
- Content is the *template*
  - Could be either *literal character data*, or *literal result element*
- Literal character data is Unicode text copied to the output

  <xsl:template match=“person”>A Person</xsl:template>
- Literal result elements are elements copied to output tree
  - Element is result element if it does not belong to the XSL namespace, and not to an extension namespace
  - Must be well-formed

  <xsl:template match=“person”> A Person </xsl:template>

xsl:value-of

- Computes the value of a node
- Attribute “select” is an XPath expression
  - expression result is converted to its string value

```
<xsl:template match=“person”>
  <p>
    <xsl:value-of select=“name”/>
  </p>
</xsl:template>
```

xsl:apply-templates

- By default, templates are processed top-to-bottom, starting with the root element
  - multiple matching templates are sorted by priority
  - If none matches, recursively traverses nested elements (by means of builtin template rules)
- Stylesheet can control recursive invocation using xsl:apply-templates
- select attribute defines node-set to recurse to

```
<xsl:template match="person>
  <xsl:apply-templates select="name">
</xsl:template>
```

Built-in template rules

```
<xsl:template match="*|/">
  <xsl:apply-templates/>
</xsl:template>

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

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

Modes

- Depending on context, the same node may be output in different ways several times
  - e.g. each chapter is output once in the table of contents, and once in the body of the book
- Both xsl:template and xsl:apply-templates take a “mode” attribute
  - when operating in a mode, only templates of the current mode are considered
  - There are built-in rules for each mode:

```
<xsl:template match="*|/" mode="m">
  <xsl:apply-templates mode="m"/>
</xsl:template>
```

Attribute Value Templates

- Generating attribute values does not work using xsl:value-of
  - Cannot put XSL elements into attribute values
- Attribute values in literal elements are treated as attribute value templates
- Curly braces denote XPath expression

<name first="{first_name}" initial="{middle_initial}" last="{last_name}">

Namespaces

- Location steps always compare namespace URI and local name
- If names are in a namespace, namespace prefixes must be declared in the template
  - E.g. for document

    <people xmlns="http://www.cafeconleche.org/namespaces/people">…

    the template should read

    <xsl:stylesheet version="1.0" xmlns:xsl="…"

    xmlns:pe="http://www.cafeconleche.org/namespaces/people">

    <xsl:template match="pe:person">

    …

    </xsl:template

    </xsl:stylesheet>

Non-literal Elements and Attributes

- xsl:element
- Creating elements with computed names
- Attribute “name” is the element name (an attribute template)
- Optional attribute “namespace” gives namespace URI

<xsl:element name=“p”>Some Text</xsl:element>

- xsl:attribute can be used both inside literal elements and xsl:element

<xsl:attribute name=“href”>http://www.heise.de</xsl:attribute>

Attribute Sets

- Repeated inclusion of multiple attributes into different elements

```
<xsl:attribute-set name="head-style">
  <xsl:attribute name="font-size">14pt</xsl:attribute>
  <xsl:attribute name="font-weight">bold</xsl:attribute>
</xsl:attribute-set>
```

- Usage through xsl:use-attribute-sets attribute on xsl:element, xsl:copy, xsl:attribute-set, or literal elements

```
<xsl:template match="chapter">
    <h1 xsl:use-attribute-sets="common head-style">
      <xsl:apply-templates/>
    </h1>
</xsl:template>
```

xsl:for-each

- Repeated execution of an embedded template
  - Short-hand for creation of a separate template

```
<xsl:template match="/">
    <html>
    <ul>
     <xsl:foreach select=".//chapter">
        <li><xsl:value-of select="."/></li>
      </xsl:foreach>
    </ul>
    <xsl:apply-templates/>
    </html>
</xsl:template>
```

Sorting

- xsl:sort is applicable to xsl:apply-templates and xsl:foreach
- “select” determines sorting key, default “.”
- “data-type” determines data type to which keys are converted
  - Default is “text”
  - Possible values are “text”, “number”, and QNames (identifying extensions)
- “order” has values “ascending”, “descending” (default ascending)
- “lang” determines language for collation (default from the environment)
- “case-order” has values “upper-first”, “lower-first”

```
<xsl:for-each select=“person”>
  <xsl:sort select=“@born” data-type=“number”/>
  <xsl:value-of select=“last_name”>
</xsl:for-each>
```

Conditional Processing

- xsl:if defines conditional inclusion
  - Attribute “test” determines condition (an XPath Expr)
  - No else-part

```
<xsl:if test=“middle_initial”>
    <xsl:value-of select=“middle_initial”/>.
</xsl:if>
```

- xsl:choose offers choice among a number of alternatives
  - Multiple nodes with xsl:when test=“expr”
  - At most one xsl:otherwise

Other Nodes

- Text nodes can be created with xsl:text
  - Attribute disable-output-escaping (“yes”,”no”) can cause “<“ not to be escaped
  - xsl:text can be used to guarantee white space in the output
- xsl:processing-instruction generates PIs

<xsl:processing-instruction
name=“xml-stylesheet”>href=“book.css” type=“text/css”</xsl:processing-instruction>

- xsl:comment generates comments
- xsl:copy copies the context node to the output

<xsl:template match=“*|@*”> <!--identity transformation-->
<xsl:copy><xsl:apply-templates select=“*|@*”></xsl:copy>
<xsl:template>

Named Templates

- Attribute “name” gives a name to a template
  - Template names must be QNames
- xsl:call-template name=“*name*” invokes the named template
  - Template is invoked with the current context node
  - Caller can pass parameters to the template

Template Parameters

- Declared in the template with xsl:param
  - Referred-to in XPath expressions as a variable
  - Default value given in “select” attribute (an XPath expression)
    - Alternatively, specify default value as content
    - Parameters without a default value are empty strings by default

```
<xsl:template name=“calculateArea”>
    <xsl:param name=“width”/>
    <xsl:param name=“height” select=“150”/>
    <xsl:value-of select=“$width * $height”/>
</xsl:template>
```

- Parameters introduce another data type: result tree fragment
  - Parameter value can be a list of nodes, to be copied into the result
  - xsl:copy-of select= can be used to copy a result tree fragment into the target tree

Passing Parameters

- Element xsl:with-param in xsl:call-template and xsl:apply-template
- Attributes name and select

```
<xsl:call-template name="calculateArea">
   <xsl:with-param name="width" select="100"/>
</xsl:call-template>
```

Global Parameters

- xsl:param elements inside xsl:stylesheet
- Parameter passing implementation-defined
  - Xalan: -param name value
  - xsltproc: --param, --stringparam
  - XT, msxsl, Saxon: name=value
  - Java TrAX API: Transformer.setParameter

Variables

- xsl:variable name="variablename" select="expression"
  - Alternatively, specify value using xsl:variable content

```
<xsl:variable name="y">
    <xsl:choose>
      <xsl:when test="$x &gt; 7">
     <xsl:text>13</xsl:text>
      </xsl:when>
      <xsl:otherwise>15</xsl:otherwise>
    </xsl:choose>
</xsl:variable>
```

- Variables are bound at the point of definition, cannot be changed afterwards
  - Use recursion to rebind a variable in a nested invocation
- Variable scope is the element in which it is contained

Processor Parametrization

- Configuration by attributes of and elements inside xsl:stylesheet
  - Attribute “extension-element-prefixes” gives a list of namespace prefixes for extension elements
  - Attribute “exclude-result-prefixes” lists namespaces which should not occur in the output

xsl:output

- xsl:output specifies how to generate a byte stream
  - method="xml" | "html" | "text" | qname
  - version = <version of the output method>
    - XML: default is 1.0, HTML: default is 4.0
  - encoding=<name of output encoding>
  - omit-xml-declaration="yes"|"no"
  - standalone="yes"|"no"
  - doctype-public=<public id>
  - doctype-system=<system id>
  - cdata-section-elements=<list of elements whose text content should use CDATA sections>
  - indent="yes"|"no"
  - media-type=<MIME type> (text output: default is text/plain)
- multiple xsl:output declarations are merged
  - default values for unspecified flags are set after merging, depending on output method

Default Output Method

- Default method is html if
  - root node has an element child,
  - localname of first such element is “html” (any case); namespace is null
  - text nodes preceding that element contain only white-space
- Otherwise, default method is xml
- Default method should be used if there is no xsl:output, or none of the xsl:output elements contain method=

Whitespace Stripping

- Sometimes, text nodes consisting only of white-space are stripped
- White-space is preserved if:
  - the parent element is listed as preserving
  - the text node contains non-whitespace characters
  - an ancestor has xml:space="preserve", and no closer ancestor has xml:space="default"
- In the input document, elements are listed as preserving in the xsl:strip-space/xsl:preserve-space element
  - Initial default is to preserve white space

  <xsl:strip-space elements="tr td th tbody"/>
- In the template, only xsl:text is listed as preserving

Combining Stylesheets

- xsl:include incorporates a stylesheet on the XML tree level

  <xsl:include href="toot-o-matic-variables.xsl"/>
- xsl:import incorporates stylesheets "logically"
  - imported stylesheets have lower precedence
  - templates, strip-space,etc. can be overridden in the importing style sheet
  - xsl:apply-imports works like xsl:apply-templates, but only considers imported templates

Priority

- Each template has a priority
- Templates without explicit priority have builtin priority
  - -0.5 for general node tests (node(), text(), *, …) without predicate and other location steps
  - -0.25 for tests of the form prefix:* or @prefix:*
  - 0 for tests consisting just of a name (elem, @attr)
  - 0.5 for all other patterns
- Attribute priority= on a template specifies explicit priority

Following Links

- Core functions from XPath are available
  - id() function looks useful, but requires the DTD to be available
- XSLT supports arbitrary additional keys into the document
  - Declared through xsl:key

    <xsl:key name="person-by-year" match="person" use="@born"/>
    - name= specifies the parameter to the key() function
    - match= specifies the candidate nodes that this key indexes
    - use= specifies the key value
  - key() function takes key name and key value, returns selected nodes

    key('person-by-year', '1912')

Numbering

- xsl:number element generates numbers, following various schemes
- value= specifies the value to print

  <xsl:number value="position()" format="1. "/>
- Otherwise, count= specifies the things to be counted
  - optionally, from= specifies where to start counting
- level= specifies nesting of numbers
  - 'single' produces a single level of numbers (1., 2., 3., …), counting siblings
  - 'multiple' numbers different levels (1.1., 1.2., 1.3., 2.1., …), according to nesting
  - 'any' numbers a single level, counting all preceding elements in the document

Numbering (2)

- lang= specifies the language for numbering
- format= specifies the format of numbers
  - format string is split into alphanumeric parts
  - each part formats a level in numbering
  - decimal numbers can specify minimum width (e.g. 01 specifies 01, 02, 03, …, 99, 100, …)
  - ‘A’ generates ‘A’, ‘B’, ‘C’, ‘D’, ‘E’, …, ‘AA’,…
  - ‘a’ likewise
  - ‘i’ and ‘I’ generate Roman numerals
- letter-value= can be ‘alphabetic’ or ‘traditional’
- grouping-separator (‘,’) and grouping-size (‘3’) specify numeric grouping

Numbering (3)

- Examples:

```
<xsl:number level=“multiple” count=“chapter|sect1|sect2” format=“1.1.1. “>
```

counts chapters, then sect1 within a chapter, sect2 within sect1

```
<xsl:template match=“note”>
    <fo:block>
      <xsl:number level=“any” from=“chapter” format=“(1) “/>
      <xsl:apply-templates/>
    </fo:block
</xsl:template>
```

numbers all notes subsequently within a chapter

Multiple Input Documents

- document(‘url’) gives a root node of an external document
- Can be combined with location steps

```
<xsl:for-each select=“/report/po”>
  <xsl:apply-templates select=“document(@filename)/purchase-order”/>
</xsl:for-each>
```

XSLT Extensions

- Extension elements:
  - Declared through extension-element-prefix
  - Implemented in the XSLT processor
  - Presence can be detected through element-available-function
    - element-available('redirect:write')
  - Multiple vendors can be supported by checking system-property('xsl:vendor')
    - element-available not implemented in XT
- Extension functions
  - Likewise declared through extension-element-prefix
  - Can be typically implemented by providing additional source code to the processor

Multiple Output Documents

- Xalan: Namespace redirect="org.apache.xalan.xslt.extensions.Redirect"
  - Elements redirect:open, redirect:close, redirect:write
  - Attribute select= specifies file name
  - Content of xsl:write is written to the output file
- SAXON: Namespace saxon="http://icl.com/saxon"
  - Element saxon:output
  - Attribute href= specifies output file, attribute value templates allowed
  - Content of saxon:output is written to a file
- XT: Namespace xt="http://www.jclark.com/xt"
  - Element xt:document
  - Attribute method= specifies output method
  - Attribute href= specifies output file

Pattern Syntax

```
[1]   Pattern    ::=   LocationPathPattern
                     | Pattern '|' LocationPathPattern
[2]   LocationPathPattern    ::=   '/' RelativePathPattern?
                     | IdKeyPattern (('/' | '//') RelativePathPattern)?
                     | '//'? RelativePathPattern
[3]   IdKeyPattern    ::=   'id' '(' Literal ')'
                     | 'key' '(' Literal ',' Literal ')'
[4]   RelativePathPattern    ::=   StepPattern
                     | RelativePathPattern '/' StepPattern
                     | RelativePathPattern '//' StepPattern
[5]   StepPattern    ::=   ChildOrAttributeAxisSpecifier NodeTest Predicate*
[6]   ChildOrAttributeAxisSpecifier    ::=   AbbreviatedAxisSpecifier
                     | ('child' | 'attribute') '::'
```

Additional Functions

- string format-number(number, string, string?)
  - format the number according to the JDK 1.1 DecimalFormat string
  - third argument specifies meaning of format characters
    - defaults from xsl:decimal-format element

  e.g. format-number(528.3, '#.#;-#.#') → 528.3
- node-set current() gives the 'current' node
  - in a Filter expressions, this might be different from the context node
- string unparsed-entity-uri(string)
- string generate-id(node-set?)
- object system-property(string)
  - supported are xsl:version, xsl:vendor, xsl:vendor-url

Additional Functions (2)

- boolean element-available(string)
- boolean function-available(string)

Additional Elements

- xsl:message terminate="yes"|"no" displays debug information
- xsl:fallback is executed if its parent element is an extension element that cannot be found

Literal Result as a Stylesheet

- Simplified form of a style-sheet
- Top-Level node can be literal element
  - style-sheet does not contain any templates, just a literal result
- Can use xsl:for-each, xsl:value-of inside elements