

XSLT 2.0

Not Your Mother's XSLT

Bonnie B. Ricca

Sun Microsystems

bonnie.ricca@sun.com

bonnie@bobrow.net



Contents

➡ Review of XPath/XSLT 1.0

- Datatypes
- Keyword Operators
- String Manipulation
- Expression Syntax
- Flexibility
- Grouping
- Conclusion



What Is XPath?

- “XML Path”
- World Wide Web Consortium (W3C) standard
 - 1.0 finalized in 1999
 - 2.0 currently in Working Draft stage
- Language for locating parts of an XML document
 - based on Unix-like paths that navigate XML tree
 - has set of standard library functions
- Designed to be used by XSLT

XPath 1.0 Examples

```

<orders>
  <order custId="77834">
    <id>101899</id>
    <item>
      <id>421</id>
      <price currency="dollars">21.99</price>
      <number>3</number>
    </item>
    <item> ← context node
      <id>325-A</id>
      <price currency="dollars">12.49</price>
      <number>21</number>
    </item>
  </order>
  <order custId="48803">
    <id>101900</id>
    <item>
      <id>579</id>
      <price currency="pounds">7.99</price>
      <number>8</number>
    </item>
  </order>
</orders>

```

`/orders/order/@custId`

custId attributes that belong to order elements that are children of orders elements that are children of the document root

`id`

id elements that are children of context node item

`//item[price > 10]/id`

id elements that are children of ALL item elements with child price > 10

`//order [count(item) > 1]`

ALL order elements with more than one child element

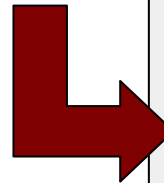


What Is XSLT?

- “XSL Transformations”
 - part of XSL (Extensible Stylesheet Language)
- W3C Recommendation
 - 1.0 finalized in 1999 (same day as XPath 1.0)
 - 1.1 never finalized
 - some implementations exist
 - 2.0 currently in Working Draft stage
- Enables transformation of XML doc structure
 - stylesheets define how to reorganize data
 - can output XML, HTML, plain text

Transforming XML

```
<order custId="77834">
  <id>101899</id>
  <item>
    <id>421</id>
    <price currency="dollars">21.99</price>
    <number>3</number>
  </item>
  <item>
    <id>325-A</id>
    <price currency="dollars">12.49</price>
    <number>21</number>
  </item>
</order>
```



```
<html>
Order ID:101899 <br>
<table border="2">
  <tr><th>Item</th>
  <th>Price</th>
  <th>Number</th>
</tr>
<tr>
  <td>325-A</td>
  <td><em>12.49</em></td>
  <td>21</td>
</tr>
<tr>
  <td>421</td>
  <td><b>21.99</b></td>
  <td>3</td>
</tr>
</table>
</html>
```

Order ID:101899

Item	Price	Number
325-A	12.49	21
421	21.99	3

Templates

```

<xsl:template match="/">
  <html>
    <xsl:apply-templates />
  </html>
</xsl:template>

<xsl:template match="order">
  Order ID:<xsl:value-of
              select="id"/><br/>
  <table border="2">
    <tr><th>Item</th><th>Price</th>
      <th>Number</th></tr>
    <xsl:apply-templates select="item">
      <xsl:sort select="id"/>
    </xsl:apply-templates>
  </table>
</xsl:template>

<xsl:template match="item">
  <tr><xsl:apply-templates /></tr>
</xsl:template>

```



Order ID:101899

Item	Price	Number
325-A	12.49	21
421	21.99	3

```

<xsl:template match="id | number">
  <td><xsl:apply-templates/></td>
</xsl:template>

<xsl:template match="price">
  <xsl:choose>
    <xsl:when test=". >= '20'">
      <td><b><xsl:apply-templates/></b></td>
    </xsl:when>
    <xsl:when test=". &lt; '10'">
      <td><xsl:apply-templates/></td>
    </xsl:when>
    <xsl:otherwise>
      <td><em><xsl:apply-templates/></em></td>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>

```



Goals of XPath/XSLT 2.0

1.0 Drawbacks

- Steep learning curve
- Minimal datatype support
- Poor string manipulation
- No grouping mechanism
- Only one output tree

2.0 Goals

- Increase usability and readability
 - add more functions
 - improve syntax
- Expand datatyping
- Simplify string manipulation
- Add grouping functionality
- Allow multiple output docs
- *Maintain backward compatibility



Contents

- Review of XPath/XSLT 1.0

➡ **Datatypes**

- Keyword Operators
- String Manipulation
- Expression Syntax
- Flexibility
- Grouping
- Conclusion



1.0 Datatypes

- XSLT/XPath 1.0 has 6 datatypes:
 - Strings
 - Booleans
 - Numbers (double-precision floating point)
 - Node-sets, contained:
 - Element
 - Text
 - Comment
 - Attribute
 - Document
 - Processing Instruction
 - Namespace
 - Result Tree Fragments
 - External Objects



New 2.0 Datatypes

- Supports XML-Schema types
 - 19 primitive built-in types
 - 25 derived datatypes
 - custom datatypes
 - *Careful:* XSLT processors not required to be “schema-aware”
- Sequence
 - new collection type
 - replaces the node-set and result tree fragment



Using Schema Types

- Create types from strings using constructors

```
xs:date('2002-03-11')
```

- Elements and attributes can be cast to types

```
<xsl:sort select="xs:date(@dob)" />
```



2.0 DateTime Support

➤ 10 Datatypes

xs:dateTime
xs:date
xs:time
xs:gYearMonth
xs:gYear
xs:gMonthDay
xs:gMonth
xs:gDay
xdt:yearMonthDuration
xdt:dayTimeDuration

➤ 3 Formatting Functions

format-time()
format-date()
format-dateTime()

➤ 21 Extraction Functions

get-years-from-yearMonthDuration()
get-month-from-dateTime()
get-timezone-from-date()
get-seconds-from-time()

➤ 3 Timezone Adjustment Functions

adjust-time-to-timezone()
adjust-date-to-timezone()
adjust-dateTime-to-timezone()



Example

- datatypeErrors.xsl



Sequences

- An ordered list that holds zero or more items
 - nodes
 - atomic values
 - *e.g.*, XML-Schema types
- Like node-set, the result of an expression



1. Everything Is a Sequence

- Every expression returns a sequence

Expression	Sequence
<code>(item, number)</code>	all the <code>item</code> and <code>number</code> children of the context node
<code>(1 to 12)</code>	all integers from 1 to 12, inclusive
<code>()</code>	empty sequence



Example

- sequence1.xsl

```
<xsl:for-each select="(1 to 100)[. mod 5 = 0]">
```



2. Sequences Are Shallow

- They get flattened
- These are all equivalent:

(a, b, (c, (d, e)), f)

((a), b, c, d, (e, (f)))

(a, b, c, d, e, f)

3, 4. Ordered with Duplicates

- No dependence on document order
- Can contain duplicates
 - node-sets could not

```
<xsl:for-each select="(following-sibling::*[1],  
following-sibling::*[2], following-sibling::*[1])">
```

- Simulate node-sets for location paths
 - returned in document order
 - duplicates are removed



Example

- `sequence2.xsl`



5. Mixed Types

- Can contain different types in one sequence

```
(xs:date('2002-03-11'), 'Hello', //order)
```

Value of a Sequence

- The value of a sequence is its first item

```
<xsl:value-of select="(xs:date('2002-03-11'), 'Hello', //order) " />
```

↓

```
2002-03-11
```

- Can declare a separator value to include all values in output:

```
<xsl:value-of select="(1 to 4)" separator=" + " />
```

↓

```
1 + 2 + 3 + 4
```

Sequences with One Item

- A sequence of one item is equivalent to that one item

```
<xsl:value-of select="(.) = ."/>
```

true

Sequence Functions

➤ 14 Sequence Functions

insert-before()

remove()

replace()

zero-or-one()

one-or-more()

exactly-one()

empty()

item-at()

index-of()

distinct-nodes()

- For document with 3 `item` elements:

```
<xsl:value-of select="count(//item, //item)" />
```

6

```
<xsl:value-of select="count(distinct-nodes(//item, //item))" />
```

3



Contents

- Review of XPath/XSLT 1.0
- Datatypes
- ➡ **Keyword Operators**
 - String Manipulation
 - Expression Syntax
 - Flexibility
 - Grouping
 - Conclusion



Some New Keywords

➤ Iteration

for
in
return

➤ Comparisons

every
some
satisfies

➤ Conditionals

if
else
then

➤ Sets

except
intersection
union

Iterative Expressions

Syntax: `for $var1 in expression (, $var2 in expression)
return expression`

- Iterates through a sequence
 - Allows navigation of node tree from within expression

```
<xsl:value-of select="for $x in ./item  
return $x/id" separator="," />
```

→ 421, 325-A, 113

- Can join more than one sequence

```
<xsl:value-of select="for $i in (10, 20),  
$j in (1, 2)  
return ($i + $j)" separator=' ' />
```

→ 11, 12, 21, 22

Subtotals/Total, 1.0

- Seemingly simple task

```
<order custId="77834">
  <id>101899</id>
  <item>
    <id>421</id>
    <price currency="dollars">21.99</price>
    <number>3</number>
  </item>
  <item>
    <id>325-A</id>
    <price currency="dollars">12.49</price>
    <number>21</number>
  </item>
  ...
</order>
```



Item	Price	Units	Subtotal
421	21.99	3	65.97
325-A	12.49	21	262.29
113	20.00	14	280
Total			608.26

*total of
calculated subtotals*



Example

- Subtotals/Total, Take 1
 - totals1-1.xsl

```
<xsl:value-of select="sum( item/price * item/number )" /> *
```

** This doesn't work!*



Example

- Subtotals/Total, Take 2
 - totals1-2.xsl

```
<xsl:variable name="total-rtf" >
  <xsl:for-each select="item">
    <subtotal>
      <xsl:value-of select="price * number" />
    </subtotal>
  </xsl:for-each>
</xsl:variable>
<xsl:value-of select="sum( exsl:node-set($total-rtf)/subtotal )" />
```



Example

- Subtotals/Total, XSLT 2.0
 - totals2.xsl

```
<xsl:value-of select="
    sum( for $x in /order/item
          return $x/price * $x/number ) " />
```

Conditional Expressions

Syntax: **if** (expression) **then** expression **else** expression

- Assigning the larger of two values to a variable:

➤ 1.0:

```
<xsl:choose>  
  <xsl:when test="$a > $b">  
    <xsl:value-of select="$a"/>  
  </xsl:when>  
  <xsl:otherwise>  
    <xsl:value-of select="$b"/>  
  </xsl:otherwise>  
</xsl:choose>
```

➤ 2.0:

```
<xsl:value-of select=  
  "if ($a > $b) then $a else $b"/>
```

Comparative Expressions, 1.0

- “=” operator
 - compares values or node-sets
 - returns “true” if at least one item meets condition

```
//price/@currency = "pounds"
```

```
//price/@currency != "pounds"
```

true

```
not(//price/@currency = "pounds")
```

```
not(//price/@currency != "pounds")
```

false

- Powerful, but
 - not always intuitive
 - not always what you want

Comparative Expressions, 2.0

- XPath 2.0 preserves “=” behavior
- Adds two new keywords:
 - some
 - default for “=” in 1.0

some \$x **in** //price/@currency **satisfies** \$x = “pounds”

➤ every

- “universal qualification” ability not available in 1.0

every \$x **in** //price/@currency **satisfies** \$x = “pounds”



Universal Qualification

- “If all values are above 0...”

➤ 1.0:

```
<xsl:if test="count( //question[number(@value) > 0] )  
= count( //question )">
```

➤ 2.0:

```
<xsl:if test="every $question in //question  
satisfies $question/@value > 0">
```



Set Operators, 1.0

- XPath 1.0 had one set operator:
 - | (union)
- Could not easily
 - determine a node's membership in a set
 - exclude certain nodes from a set



Set Operators, 2.0

- **union**

- can still write it as “|”
- no change in behavior from 1.0
 - removes duplicates
 - preserves document order

- **intersect**

- intersection of two sets

- **except**

- difference between two sets

Intersection, 1.0

- “Do the prices over 20 include '21.99'?”

```
count( //price[. > 20] | //price[. = '21.99'] ) = count( //price[. > 20] )
```

```
<price>21.99</price>
<price>20.00</price>
```

union

```
<price>21.99</price>
```

```
<price>21.99</price>
<price>20.00</price>
```

??

```
<price>21.99</price>
<price>20.00</price>
```

Intersection, 2.0

- “Do the prices over 20 include '21.99'?”

```
//price[. = '21.99'] intersect //price[. > 20]
```

```
<price>21.99</price>
```

?
intersect

```
<price>21.99</price>  
<price>20.00</price>
```



Differences

- “All attributes except @css:temp”

➤ 1.0:

```
@*[ not( namespace-uri() = 'http://www.softwaresummit.com/examples'  
and local-name() = 'temp' ) ]
```

• or

```
@*[ not( generate-id(.) = generate-id(../@css:temp) ) ]
```

➤ 2.0:

```
@* except @css:temp
```



Contents

- Review of XPath/XSLT 1.0
- Datatypes
- Keyword Operators
- ➔ **String Manipulation**
 - Expression Syntax
 - Flexibility
 - Grouping
 - Conclusion



Regular Expressions

- XPath 2.0 introduces regular expressions
 - not quite identical to Perl's
 - extension of XML-Schema's
- New string manipulation functions
 - use with regular expressions

```
matches()  
replace()  
tokenize()
```

Example

- Search and replace, 1.0

- replace1.xsl

```
<xsl:template name="replace-string">
  <xsl:param name="text"/> <xsl:param name="from"/> <xsl:param name="to"/>
  <xsl:choose>
    <xsl:when test="contains($text, $from)">
      <xsl:variable name="before" select="substring-before($text, $from)"/>
      <xsl:variable name="after" select="substring-after($text, $from)"/>
      <xsl:value-of select="concat($before, $to)"/>
      <xsl:call-template name="replace-string">
        <xsl:with-param name="text" select="$after"/>
        <xsl:with-param name="from" select="$from"/>
        <xsl:with-param name="to" select="$to"/>
      </xsl:call-template>
    </xsl:when>
    <xsl:otherwise> <xsl:value-of select="$text"/> </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```



Example

- Search and replace, 2.0
 - replace2.xsl

```
replace( text, from, to )
```



Contents

- Review of XPath/XSLT 1.0
- Datatypes
- Keyword Operators
- String Manipulation
- ➔ **Expression Syntax**
 - Flexibility
 - Grouping
 - Conclusion

Improved Location Steps

- Location steps can include:
 - functions

```
<xsl:for-each select="doc('cities.xml')//city/@name">  
  ...  
</xsl:for-each>
```

- parenthesized expressions

```
<xsl:value-of select="sum( /inventory/(clothing | officeSupplies)/item/units ) "/>
```

XPath Comments

- Can still use XSLT comments
 - standard XML comments

```
<!-- calculate the total price of all items ordered -->  
<xsl:value-of select="  
    sum(for $x in /order/item return $x/price * $x/number)" />
```

- Can add comments to XPath expressions, too

```
<xsl:value-of  
    select="(: total all subtotals :)  
        sum(  
            (: multiply number by price for each item :)  
            for $x in /order/item return $x/price * $x/number)" />
```



Contents

- Review of XPath/XSLT 1.0
- Datatypes
- Keyword Operators
- String Manipulation
- Expression Syntax
- ➔ **Flexibility**
- Grouping
- Conclusion

Debugging Functions

- Can now debug inside XPath expressions:

`error()`
`trace()` ← terminates processing

```
<xsl:value-of select="
  if ( //order )
  then count( //order )
  else error('No orders found.')" />
```



Example

- `error.xsl`



Custom Functions

- XSLT 1.0 has “named templates”
 - create template with a name and parameters
 - call it from other templates
 - still available in 2.0
- XSLT 2.0 adds `<xsl:function>` directive
 - define a custom function
 - use it in XPath expressions



Defining a Function

- Rules:
 - cannot be defined in default namespace
 - cannot include optional parameters
 - all parameters must be required
 - no *select* attribute for default values



Example

- function.xsl

```
<xsl:stylesheet version="2.0" ...  
  xmlns:ss="http://www.softwaresummit.com/examples">  
  
  <xsl:function name="ss:factorial">  
    <xsl:param name="n"/>  
    <xsl:value-of select="  
      if ($n = 0)  
      then 1  
      else $n * ss:factorial($n - 1)" />  
  </xsl:function>  
  
  ...  
</xsl:stylesheet>
```



Multiple Output Documents

- Available in 1.0 only through extensions
- Now part of XSLT 2.0

```
<xsl:result-document  
  format = "QualifiedName"  
  href   = "uri-reference">  
</xsl:result-document>
```

- *format* refers to `<xsl:output>` declaration
 - if not stipulated, uses the unnamed `<xsl:output>`



Sending XML Messages

- Set *href* attribute to Web service address
- Send SOAP message during transformation
 - `xs1:result-document` does not block
 - no return message received
 - no acknowledgement of receipt



Contents

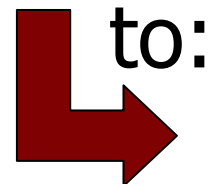
- Review of XPath/XSLT 1.0
- Datatypes
- Keyword Operators
- String Manipulation
- Expression Syntax
- Flexibility
- ➔ **Grouping**
- Conclusion



What Is Grouping?

Allows you to go from:

```
<cities>
  <city name="Buffalo" state="New York" />
  <city name="Yonkers" state="New York" />
  <city name="Paramus" state="New Jersey" />
  <city name="Brooklyn" state="New York" />
  <city name="Summit" state="New Jersey" />
  <city name="Stamford" state="Connecticut" />
  <city name="Mamaroneck" state="New York" />
  <city name="White Plains" state="New York" />
  <city name="Norwalk" state="Connecticut" />
  <city name="Bayonne" state="New Jersey" />
</cities>
```



to:

```
<states>
  <state name="New York">
    <city>Buffalo</city>
    <city>Yonkers</city>
    <city>Brooklyn</city>
    <city>Mamaroneck</city>
    <city>White Plains</city>
  </state>
  <state name="New Jersey">
    <city>Paramus</city>
    <city>Summit</city>
    <city>Bayonne</city>
  </state>
  <state name="Connecticut">
    <city>Stamford</city>
    <city>Norwalk</city>
  </state>
</states>
```



Grouping 1.0

1. Sort the city nodes before doing anything else:

```
<xsl:variable name="sorted-cities">  
  <xsl:for-each select="//city">  
    <xsl:sort select="@state" />  
    <xsl:copy-of select="." />  
  </xsl:for-each>  
</xsl:variable>
```



Grouping 1.0 *(Continued)*

2. Get unique states from sorted list
 - compare each city with previous one

```
<xsl:variable name="unique-states"  
  select="$sorted-cities/city  
    [not( @state=preceding-sibling::city[1]/@state )]/@state " />
```



Grouping 1.0 *(Continued)*

3. Generate state elements for each unique state with city child elements:

```
<states>
  <xsl:for-each select="$unique-states">
    <state name="{.}" >
      <xsl:for-each select="//city[@state=current()]">
        <city><xsl:value-of select="@name" /></city>
      </xsl:for-each>
    </state>
  </xsl:for-each >
</states>
```



Example

- grouping1-1.xsl



Grouping 1.0 Drawbacks

- Need to iterate thru each city twice:
 - to find unique ones
 - to find ones that match current state
- Requires optimized processor
 - most should optimize `preceding-sibling::city[1]`
 - otherwise it searches through all preceding siblings to find ones with `position()=1`

Grouping 1.0: “Muenchian Technique”

- Most efficient technique
- Developed by engineer Steve Muench
- Uses keys
 - allows for quick access to nodes
 - like a hash table
- Keys have attributes:
 - name – unique name you give it
 - match – XPath expr. for node(s) key identifies
 - use – XPath expr. for node value used as key



Muenchian Technique

1. Create key to access nodes by state attr

```
<xsl:key name="cities-by-state" match="city" use="@state" />
```

- will use it later to access city elements by state:

```
key('cities-by-state', 'New Jersey')
```



Muenchian Technique *(Continued)*

2. Get unique states using the key:
 - a. check each city against the first city in its state's key

```
//city[ generate-id(.) =  
        generate-id( key('cities-by-state', @state )[1] ) ]
```

- b. if it's the first one, process it
 - still going through each city, but using key is much faster

Muenchian Technique *(Continued)*

3. Use key() function to access group for each unique state

```
<xsl:key name="cities-by-state" match="city" use="@state" />
<xsl:template match="/">
  <states>
    <xsl:for-each select="//city[ generate-id(.) =
      generate-id( key('cities-by-state', @state )[1]) ] ">
      <state name="{@state}">
        <xsl:for-each select="key('cities-by-state', @state)">
          <city><xsl:value-of select="@name" /> </city>
        </xsl:for-each>
      </state>
    </xsl:for-each>
  </states>
</xsl:template>
```



Example

- grouping1-2.xsl



Muenchian Technique Drawbacks

- Still a little slow
 - building keys is time-consuming
 - but done just once
 - subsequent access is instantaneous
- Performance boost => resource loss
 - takes tons of memory
 - underlying key datatype stored in memory
 - not always feasible for multiple or large docs



Grouping, 2.0

- New directive:
 - `<xsl:for-each-group>`
 - builds groups for you
- New function:
 - `current-group()`
 - allows access to groups during iteration
- Used in conjunction, they solve the problem

Grouping 2.0 *(Continued)*

- This is all you need:

```
<xsl:template match="/">
  <states>
    <xsl:for-each-group select="cities/city" group-by="@state">
      <state name="{@state}">
        <xsl:for-each select="current-group()">
          <city><xsl:value-of select="@name" /> </city>
        </xsl:for-each>
      </state>
    </xsl:for-each-group>
  </states>
</xsl:template>
```



Example

- grouping2.xsl



Contents

- Review of XPath/XSLT 1.0
- Datatypes
- Keyword Operators
- String Manipulation
- Expression Syntax
- Flexibility
- Grouping



Conclusion



Conclusion

- More powerful
- More accessible
 - less verbose
 - more intuitive to write
 - easier to read
- More customizable
 - enables comprehensive library development
 - custom datatypes
- Faster development time
- Should be easier to maintain



Using XSLT 2.0

- Michael Kay's Saxon processor
 - <http://saxon.sourceforge.net/>
 - version 7 for XSLT 2.0
 - version 6 implements XSLT 1.0 plus some 1.1
 - Java libraries for programmatic transformation



Specifications

- XSL Transformations (XSLT) Version 2.0
 - <http://www.w3.org/TR/xslt20/>
- XML Path Language (XPath) 2.0
 - <http://www.w3.org/TR/xpath20/>
- XQuery 1.0 and XPath 2.0 Functions and Operators
 - <http://www.w3.org/TR/xquery-operators/>
- XQuery 1.0 and XPath 2.0 Data Model
 - <http://www.w3.org/TR/xpath-datamodel/>