



# XPath 2.0 and XSLT 2.0

**Norman Walsh**

*XML Standards Architect*

**Extreme Markup Languages 2004**

**01-06 August 2004**

<http://www.sun.com/>



Version 1.0

# Table of Contents

Introduction  
Speaker Qualifications  
A Running Example  
Background Material  
XPath 2.0  
XSLT 2.0  
Challenges  
Closing Thoughts

# Introduction

- This tutorial covers XPath 2.0 and XSLT 2.0 with only a passing glance at XML Query 1.0
- Focus on describing and demonstrating new features
- Assume some familiarity with XPath 1.0 and XSLT 1.0
- Mixture of slides and examples. Ask questions!

# Speaker Qualifications

- Elected member of the *W3C Technical Architecture Group*; co-chair of the *XML Core Working Group*; member of the *XSL WG*. Joint editor of several *XSL/XML Query Specs*.
- Chair of the *OASIS DocBook Technical Committee*, member of the *Entity Resolution TC* and the *RELAX NG TC*.
- Co-Spec Lead for *JSR 206: Java API for XML Processing*

# A Running Example

Everyone's third favorite toy example, the recipe collection. Our recipe list is defined by the schema described on the following slides.

RecipeList

Servings

Recipe

Recipe Subtypes

Recipe Elements

Prose Types

Prose Elements

IngredientList

Ingredient

# RecipeList

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  targetNamespace="http://nwalsh.com/xmlns/extreme2004/recipes/"
  xmlns:r="http://nwalsh.com/xmlns/extreme2004/recipes/">

  <xs:complexType name="RecipeList">
    <xs:sequence>
      <xs:element ref="r:recipe" minOccurs="1" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>

  <xs:element name="recipeList" type="r:RecipeList"/>
```

# Servings

```
<xs:simpleType name="Servings">  
  <xs:restriction base="xs:integer">  
    <xs:minInclusive value="1"/>  
    <xs:maxInclusive value="12"/>  
  </xs:restriction>  
</xs:simpleType>
```

# Recipe

```
<xs:complexType name="Recipe">
  <xs:sequence>
    <xs:element ref="r:name"/>
    <xs:element ref="r:source" minOccurs="0" maxOccurs="1"/>
    <xs:element ref="r:description" minOccurs="0" maxOccurs="1"/>
    <xs:element ref="r:ingredientList" minOccurs="1" maxOccurs="unbounded"/>
    <xs:element ref="r:preparation"/>
  </xs:sequence>
  <xs:attribute name="servings" type="r:Servings"/>
  <xs:attribute name="time" type="xs:duration"/>
  <xs:attribute name="calories" type="xs:positiveInteger"/>
</xs:complexType>
```

# Recipe Subtypes

```
<xs:complexType name="FoodRecipe">
  <xs:complexContent>
    <xs:extension base="r:Recipe"/>
  </xs:complexContent>
</xs:complexType>
```

```
<xs:complexType name="CandyRecipe">
  <xs:complexContent>
    <xs:extension base="r:FoodRecipe">
      <xs:attribute name="sugarfree" type="xs:boolean"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

```
<xs:complexType name="DrinkRecipe">
  <xs:complexContent>
    <xs:extension base="r:Recipe">
      <xs:attribute name="virgin" type="xs:boolean"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

# Recipe Elements

```
<xs:element name="recipe" type="r:Recipe" abstract="true"/>
<xs:element name="beverage" type="r:DrinkRecipe" substitutionGroup="r:recipe"/>
<xs:element name="appetizer" type="r:FoodRecipe" substitutionGroup="r:recipe"/>
<xs:element name="entrée" type="r:FoodRecipe" substitutionGroup="r:recipe"/>
<xs:element name="sidedish" type="r:FoodRecipe" substitutionGroup="r:recipe"/>
<xs:element name="dessert" type="r:FoodRecipe" substitutionGroup="r:recipe"/>
<xs:element name="candy" type="r:CandyRecipe" substitutionGroup="r:recipe"/>
```

# Prose Types

```
<xs:complexType name="Prose">
  <xs:choice minOccurs="1" maxOccurs="unbounded">
    <xs:element ref="r:p"/>
    <xs:element ref="r:list"/>
  </xs:choice>
</xs:complexType>
```

```
<xs:complexType name="Para" mixed="true">
  <xs:choice minOccurs="0" maxOccurs="unbounded">
    <xs:element name="em" type="xs:string"/>
  </xs:choice>
</xs:complexType>
```

```
<xs:complexType name="NumberedList">
  <xs:sequence>
    <xs:element name="item" minOccurs="1" maxOccurs="unbounded" type="r:Prose"/>
  </xs:sequence>
</xs:complexType>
```

# Prose Elements

```
<xs:element name="description" type="r:Prose"/>  
<xs:element name="preparation" type="r:Prose"/>  
<xs:element name="p" type="r:Para"/>  
<xs:element name="list" type="r:NumberedList"/>  
<xs:element name="title" type="r:Para"/>  
<xs:element name="name" type="r:Para"/>  
<xs:element name="source" type="r:Para" nillable="true"/>
```

# IngredientList

```
<xs:complexType name="IngredientList">
  <xs:sequence>
    <xs:element ref="r:title" minOccurs='0' maxOccurs='1' />
    <xs:element ref="r:ingredient" minOccurs='1' maxOccurs='unbounded' />
  </xs:sequence>
</xs:complexType>

<xs:element name="ingredientList" type="r:IngredientList" />
```

# Ingredient

```
<xs:complexType name="Ingredient">
  <xs:sequence>
    <xs:element name="quantity" minOccurs="0" maxOccurs="1">
      <xs:complexType>
        <xs:simpleContent>
          <xs:extension base="xs:double">
            <xs:attribute name="units"/>
          </xs:extension>
        </xs:simpleContent>
      </xs:complexType>
    </xs:element>
    <xs:element name="name" type="xs:string"/>
  </xs:sequence>
</xs:complexType>

<xs:element name="ingredient" type="r:Ingredient"/>
```

# Background Material

Specifications

Fitting the Pieces Together

Data Model

Functions and Operators

Language Semantics

Static Semantics

Language Semantics (Continued)

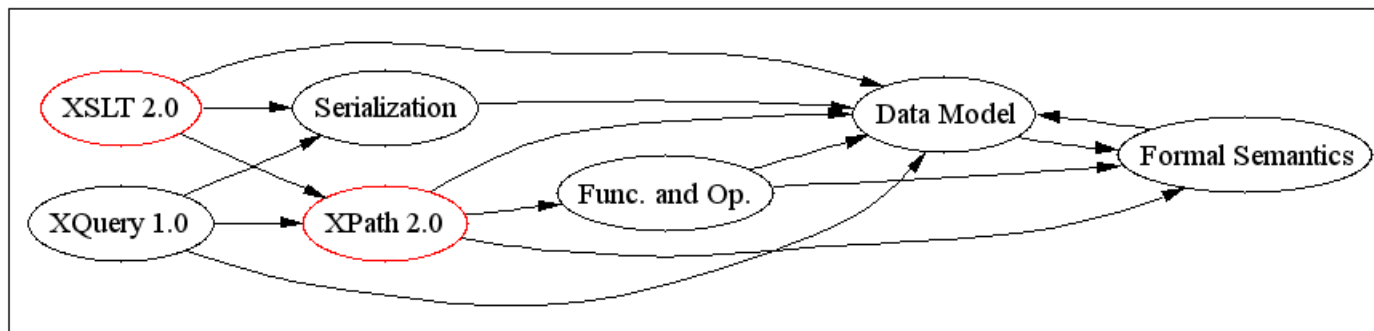
# Specifications

Seven core specifications; new Working Drafts published 23 Jul 2004.

- XQuery 1.0 and XPath 2.0 Data Model
- XQuery 1.0 and XPath 2.0 Functions and Operators
- XQuery 1.0 and XPath 2.0 Formal Semantics
- XML Path Language (XPath) 2.0
- XSL Transformations (XSLT) Version 2.0
- XSLT 2.0 and XQuery 1.0 Serialization
- XQuery 1.0: An XML Query Language

# Fitting the Pieces Together

The family of XSL and XML Query specifications are closely related. Many of the specifications depend on each other.



XSL/XML Query Specifications

(This diagram is only illustrative, not complete or exhaustive.)

## Data Model

- XPath 2.0 has nodes and typed values. Colloquially, there are three kinds of things: *nodes*, *simple* or *atomic* values, and *items*. An item is either a node or an atomic value.
- XPath 2.0 has sequences where XPath 1.0 had node sets:
  - Sequences can be in arbitrary order
  - Sequences can contain duplicates
  - Sequences can be heterogenous

# Functions and Operators

Functions. Lots of functions.

- String and numeric functions
- Date and time functions
- Sequence manipulation functions
- Casting and type-related functions

# Language Semantics

XPath 2.0 has both static and dynamic semantics:

- Static semantics define, informally, what a language means without reference to any particular input.
- Dynamic semantics, again informally, define how a language behaves presented with inputs of various sorts.

# Static Semantics

- The static type of “`1 + 1`” is `xs:integer`.
- The static type of “`r:recipeList/r:recipe`” is `r:Recipe+`.
- The static type of “`r:ingredient/r:quantity * 2`” is `xs:double`.
- The static type of “`r:name`” is `element()`.
- The static type of “`r:recipe/@time + 5`” is a type error.

## Language Semantics (Continued)

- The Formal Semantics specification describes the static semantics of XPath.
- Support for static analysis is optional.
- The XPath 2.0 specification describes the dynamic semantics of XPath.
- The XSLT 2.0 specification describes all of the semantics of XSLT.

# XPath 2.0

XML Schema Type System

XML Schema Type System (Continued)

Type Names and Type Matching

Atomization

New Types

New Duration Types

New Node Types

Element Tests (1)

Element Tests (2)

Element Test Examples

Schema Element Test

Attribute Tests

...

# XML Schema Type System

- Probably the most significant semantic change to XPath
- The XPath 1.0 type system is very simple: nodes, strings, numbers, and booleans.
- XPath 2.0 adds W3C XML Schema simple and complex types.
- XPath 2.0 has nodes and *atomic values*.
- Atomic values have simple types: xs:string, xs:integer, xs:dateTime, etc.

## XML Schema Type System (Continued)

- Allows matching and selection of elements, attributes, and atomic values by type.
- Supports a set of primitive simple types.
- Implementations may support user-defined simple and complex types.
- Implementations may support additional, non-W3C XML Schema types.

# Type Names and Type Matching

- Types are identified by name.
- Available type names are determined by schema import.
- Values are “atomized” before most comparisons.

# Atomization

- Atomization transforms a sequence into a sequence of atomic values.
- For each item in the sequence:
  - If the item is an atomic value, use it.
  - Otherwise, use the typed value of the item.
- An error occurs if the item does not have a typed value.

## New Types

xdt:anyAtomicType

The base type of all atomic types.

xdt:untypedAtomic

The type name that identifies an atomic value with no known type.

xdt:untyped

The type name that identifies any value (simple or complex) with no known type.

## New Duration Types

xdt:yearMonthDuration

A duration that consists of only years and months.

xdt:dayTimeDuration

A duration that consists of only days and times.

These duration types have the feature that they can be totally ordered; **xs:durations** are only partially ordered. (e.g., is one month and five days more or less than five weeks?)

## New Node Types

There are several new node tests in addition to the familiar `text()`, `comment()`, etc.

- `item()` matches any node *or* any atomic value.
- `document-node()` matches a document node.
- `document-node(ElementTest)` matches a document with a document element that matches *ElementTest*.

# Element Tests (1)

An **ElementTest** matches elements.

- `element()` (or `element(*)`) matches any element.
- `element(ElementName)` matches any element named *ElementName* regardless of type or nilled property.

## Element Tests (2)

- `element(ElementName, TypeName)` matches a (non-nilled) element named *ElementName* with the type *TypeName*.
- `element(ElementName, TypeName?)` is the same, but will also match nilled elements.

In element tests, a type matches the specified type or any type derived from it. So `r:DrinkRecipe` matches `r:Recipe`, for example.

## Element Test Examples

“`element(*,r:FoodRecipe)`” matches any non-nilled elements that have the type `r:FoodRecipe`.

“`element(r:name, r:Para)`” matches non-nilled elements named `r:name` that have the type `r:Para`.

“`element(r:source, r:Para?)`” matches elements named `r:source` that have the type `r:Para`, even if they have been `nilled`.

## Schema Element Test

- **schema-element(*ElementName*)** matches an element named *ElementName* or any element in the substitution group headed by *ElementName*.

“**schema-element(r:recipe)**” matches **r:recipe** and **r:beverage**, **r:appetizer**, **r:entree**, and all the other elements that can be substituted for **r:recipe**.

# Attribute Tests

An **AttributeTest** matches attributes.

It has the same forms as an **ElementTest**:

- **attribute()** (or **attribute(\*)**) matches any attribute.
- **attribute(*AttributeName*, *TypeName*)** matches an attribute by name and type.
- **attribute(\*, *TypeName*)** matches an attribute by type.

## Type Errors

- XPath 1.0 had almost no type errors: if an expression was syntactically valid, it returned a result. `"3" + 1` = 4, `"Tuesday" + 1` = NaN, etc.
- In XPath 2.0 this is not the case. Errors will terminate the evaluation of an expression, stylesheet, or query.
- XPath 2.0 adds new operators that allow you to test if an operation will succeed.

# Sequences

- Sequence construction:
  - `(1 to 10)[. mod 2 = 1]=(1,3,5,7,9)`
  - `($preamble, .//item)`
- Combining Node Sequences
  - `$seq1 union $seq2` returns all the nodes in at least one sequence.
  - `$seq1 intersect $seq2` returns all the nodes in both sequences.
  - `$seq1 except $seq2` returns all the nodes in the first sequence that aren't in the second.
- Quantified expressions (**some** and **every**).

## “For” Expressions

XPath 2.0 adds a “for” expression:

```
for $varname in (expression) return (expression)
```

N.B. This is *in XPath*. For example, it might appear in a **select** attribute.

Use cases?

```
fn:sum(for $i in order-item return $i/@price * $i/@qty)
```

XSLT 2.0 retains the **xsl:for-each** instruction.

## “If” Expressions

XPath 2.0 also adds an “if” expression:

```
if ($part/@discount)
  then $part/retail * $part/@discount
  else $part/retail
```

Again, this is *in XPath* and might appear in a **select** attribute.

```
if ($drink/@virgin) then $drink/@virgin else false()
```

XSLT 2.0 retains the **xsl:if** and **xsl:choose** instructions.

# “If” Example Questions

Consider this fragment:

```
<r:recipe>
  <dc:title>Recipe Title</dc:title>
  <r:name>Recipe Name</r:name>...
</r:recipe>
```

How are these three different?

```
<xsl:variable name="title" select="(r:name|dc:title)[1]"/>
```

```
<xsl:variable name="title">
  <xsl:choose>
    <xsl:when test="r:name"><xsl:copy-of select="r:name"/></xsl:when>
    <xsl:otherwise><xsl:copy-of select="dc:title"/></xsl:otherwise>
  </xsl:choose>
</xsl:variable>
```

```
<xsl:variable name="title"
  select="if (r:name) then r:name else dc:title"/>
```

## “If” Example Answers

Consider this fragment:

```
<r:recipe>  
  <dc:title>Recipe Title</dc:title>  
  <r:name>Recipe Name</r:name>  
  ...  
</r:recipe>
```

How are these three different?

```
<xsl:variable name="title" select="(r:name|dc:title)[1]"/>
```

Returns “**dc:title**”.

## “If” Example Answers (Continued)

```
<xsl:variable name="title">
  <xsl:choose>
    <xsl:when test="r:name">
      <xsl:copy-of select="r:name"/>
    </xsl:when>
    <xsl:otherwise>
      <xsl:copy-of select="dc:title"/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:variable>
```

Returns *a copy of* “r:name”.

## “If” Example Answers (Continued)

```
<xsl:variable name="title"  
              select="if (r:name) then r:name  
              else dc:title"/>
```

Returns “**r:name**”.

## instance of

The **instance of** operator tests if an item is an instance of a given type (or is derived by restriction from it):

```
$div instance of element(*, eg:Chapter)
```

returns true if **\$div** is a chapter.

```
5 instance of xs:decimal
```

returns true because 5 is an integer and integers are a restriction of decimals.

## treat as

The **treat as** operator fools the static type checker.

Suppose, for example, that you have a function that operates on UK addresses. If the static type of **\$curAddr** isn't a UK address, you can still pass it to the function as follows:

```
$curAddr treat as element(*, eg:UKAddress)
```

A dynamic error will occur if the address isn't a UK address.

## cast as

The **cast as** operator coerces the type of an item.

**`$x cast as eg:HatSize`**

returns the value of **`$x`** as an item of type **`eg:HatSize`**.

## castable as

Attempts to make invalid casts, for example string to integer, will raise dynamic errors. The **castable as** operator allows you to test if the cast will succeed.

**`$x castable as eg:HatSize`**

returns true if the value of **`$x`** can be cast to **`eg:HatSize`**.

## General Comparisons

General comparisons: =, !=, <, <=, >, >=

True if any pair of values satisfies the comparison. (XPath 1.0 semantics.)

```
$recipelist/*/r:source = "My Mom"
```

is true if **\$book** has one or more authors and at least one of those authors is “Kennedy”.

## Value Comparisons

Value comparisons: **eq**, **ne**, **lt**, **le**, **gt**, **ge**

Compare exactly two atomic values.

```
$recipe/r:source eq "My Mom"
```

is true if and only if **\$book** has exactly one author and that author is “Kennedy”.

Errors are raised if the comparison is not between two values or if the values cannot be compared.

## Node Comparisons

Node comparisons: **is**, **<<**, **>>**

The **is** operator compares two nodes for equality (are they *the same* node?). The operators **<<** and **>>** compare document order of nodes.

```
$book/author is key('authors', 'kennedy')
```

is true if and only if **\$book** has exactly one author and that author element is the same as the one returned by the key expression.

# XSLT 2.0

Schema Support

Types

Declaring Types

Declaring Types (Continued)

Constructor Functions vs. “as”

Type Errors

Implicit Casting

Sorting and Collation

Regular Expression Functions

Regular Expression Instructions

Regular Expression Example

Grouping

...

# Schema Support

- Many of the W3C XML Schema simple types are always available.
- In order to refer to additional types, you must import the schema that defines them:

```
<xsl:import-schema  
  namespace="http://nwalsh.com/xmlns/extreme2004/recipes/"  
  schema-location="recipes.xsd"/>
```

You must specify at least the namespace or the schema location, if not both.

# Types

XSLT 2.0 allows you to declare:

- The type of variables.
- The return type of templates.
- The type of sequences (constructed with **`xsl:sequence`**)
- The return type of (user-declared) functions.
- Both the type and required type of parameters.

## Declaring Types

The “**as**” attribute is used to declare types.

```
<xsl:variable name="i" select="1"/>
```

is the integer value 1.

```
<xsl:variable name="fp" select="1"  
              as="xs:double" />
```

is the double value 1.0.

## Declaring Types (Continued)

```
<xsl:variable name="date"  
              select="'2003-11-20'"/>
```

is the *string* “2003-11-20”.

```
<xsl:variable name="date"  
              select="xs:date('2003-11-20')"/>
```

is the date November 20, 2003.

```
<xsl:variable name="date" as="xs:date"  
              select="'2003-11-20'"/>
```

is an error.

## Constructor Functions vs. “as”

- The constructor functions, **`xs : type(string)`**, attempt to construct the typed value from the lexical form provided.
- The **`as`** attribute asserts that the value must have the required type. It performs simple type promotions but doesn't, for example, implicitly **`cast as`** the requested type.

## Type Errors

If a type cannot be cast to another type, attempting the cast will generate a type error. Some (perhaps many) of the things you're used to doing in XPath 1.0 will generate type errors in XPath 2.0:

- Math operations on strings (**@attr + 1** if **attr** is validated as a string type).
- Invalid lexical representations (“12/08/2003” is not a valid lexical form for dates, use “2003-12-08” instead).
- Incompatible casts (**100 cast as r:Servings**).

# Implicit Casting

- From subtypes to supertypes (**xs:NMTOKEN** where **xs:string** is required).
- Between numeric types (**xs:decimal** to **xs:double**, etc.)
- Computing effective boolean values (“NaN” to **false()**)
- From “untyped” values

# Sorting and Collation

- XPath 2.0 and XSLT 2.0 have *collations*
- Collations determine how sorting and collation work
- Collations are identified by URI
- All processors support “Unicode code-point collation”

# Regular Expression Functions

There are three regular-expression functions that operate on strings:

- **matches( )** tests if a regular expression matches a string.
- **replace( )** uses regular expressions to replace portions of a string.
- **tokenize( )** returns a sequence of strings formed by breaking a supplied input string at any separator that matches a given regular expression.

# Regular Expression Instructions

The **xsl:analyze-string** instruction uses regular expressions to apply markup to a string.

```
<xsl:analyze-string select="..." regex="...">
  <xsl:matching-substring>...</xsl:matching-substring>...
  <xsl:non-matching-substring>...</xsl:non-matching-substring>...
</xsl:analyze-string>
```

The **regex-group( )** function allows you to look back at matching substrings.

## Regular Expression Example

These instructions transform dates of the form “12/8/2003” into ISO 8601 standard form: “2003-12-08” using the regular expression instructions.

```
<xsl:analyze-string select="$date"
  regex="([0-9]+)/([0-9]+)/([0-9]{{4}})">
  <xsl:matching-substring>
    <xsl:number value="regex-group(3)"
      format="0001"/>
    <xsl:text>-</xsl:text> ...
```

Note that the curly braces are doubled in the regular expression. The **regex** attribute is an attribute value template, so to get a single “{” in the attribute value, you must use “{{” in the stylesheet.

# Grouping

Grouping in XSLT 1.0 is *hard*. XSLT 2.0 provides a new, flexible grouping instruction for grouping...

- on a specific key
- by transitions at the start of each group
- by transitions at the end of each group
- by adjacent key values

Groups can also be sorted.

## Grouping By Key (Data)

Group the following data by country:

```
<cities>
  <city name="Milano"    country="Italia" />
  <city name="Paris"     country="France" />
  <city name="München"   country="Deutschland" />
  <city name="Lyon"      country="France" />
  <city name="Venezia"   country="Italia" />
</cities>
```

## Grouping By Key (Code)

```
<xsl:for-each-group select="cities/city"
                    group-by="@country">
  <tr>
    <td><xsl:value-of select="position()" /></td>
    <td><xsl:value-of select="@country" /></td>
    <td>
      <xsl:value-of select="current-group() /@name"
                    separator=", " />
    </td>
  </tr>
</xsl:for-each-group>
```

## Grouping By Key (Results)

```
<tr>
  <td>1</td>
  <td>Italia</td>
  <td>Milano, Venezia</td>
</tr>
<tr>
  <td>2</td>
  <td>France</td>
  <td>Paris, Lyon</td>
</tr>
```

## Grouping By Starting Value (Data)

Group the following data so that the implicit divisions created by each **h1** are explicit:

```
<body>
  <h1>Introduction</h1>
  <p>XSLT is used to write stylesheets.</p>
  <p>XQuery is used to query XML databases.</p>
  <h1>What is a stylesheet?</h1>
  <p>A stylesheet is an XML document used to defi
  <p>Stylesheets may be written in XSLT.</p>
  <p>XSLT 2.0 introduces new grouping constructs.
</body>
```

## Grouping By Starting Value (Code)

```
<xsl:for-each-group select="*"
                    group-starting-with="h1">
  <div>
    <xsl:apply-templates
      select="current-group()" />
  </div>
</xsl:for-each-group>
```

## Grouping By Starting Value (Results)

```
<div>
  <h1>Introduction</h1>
  <p>XSLT is used to write stylesheets.</p>
  <p>XQuery is used to query XML databases.</p>
</div>
<div>
  <h1>What is a stylesheet?</h1>
  <p>A stylesheet is an XML document used to def
  <p>Stylesheets may be written in XSLT.</p>
  <p>XSLT 2.0 introduces new grouping constructs
</div>
```

## Grouping By Ending Value (Data)

Group the following data so that continued pages are contained in a **pageset**:

```
<doc>
  <page continued="yes">Some text</page>
  <page continued="yes">More text</page>
  <page>Yet more text</page>
  <page continued="yes">Some words</page>
  <page continued="yes">More words</page>
  <page>Yet more words</page>
</doc>
```

## Grouping By Ending Value (Code)

```
<xsl:for-each-group select="*"
  group-ending-with="page[not(@continued
                        ='yes')]">
  <pageset>
    <xsl:for-each select="current-group()">
      <page><xsl:value-of select="."/></page>
    </xsl:for-each>
  </pageset>
</xsl:for-each-group>
```

## Grouping By Ending Value (Results)

```
<doc>
  <pageset>
    <page>Some text</page>
    <page>More text</page>
    <page>Yet more text</page>
  </pageset>
  <pageset>
    <page>Some words</page>
    <page>More words</page>
    <page>Yet more words</page>
  </pageset>
</doc>
```

## Grouping By Adjacent Key Values (Data)

Group the following data so that lists do not occur *inside* paragraphs:

```
<p>Do <em>not</em> :  
  <ul>  
    <li>talk,</li>  
    <li>eat, or</li>  
    <li>use your mobile telephone</li>  
  </ul>  
while you are in the cinema.</p>
```

## Grouping By Adjacent Key Values (Code)

```
<xsl:for-each-group select="node()"
  group-adjacent="self::ul or self::ol">
  <xsl:choose>
    <xsl:when test="current-grouping-key()">
      <xsl:copy-of select="current-group()" />
    </xsl:when>
    <xsl:otherwise>
      <p>
        <xsl:copy-of
          select="current-group()" />
      </p>
    </xsl:otherwise>
  </xsl:choose>
</xsl:for-each-group>
```

## Grouping By Adjacent Key Values (Results)

```
<p>Do not:
```

```
</p>
```

```
<ul>
```

```
  <li>talk,</li>
```

```
  <li>eat, or</li>
```

```
  <li>use your mobile telephone</li>
```

```
</ul>
```

```
<p>
```

```
  while you are in the cinema.
```

```
</p>
```

# Functions

At the instruction level, XSLT 1.0 and 2.0 provide mechanisms for calling named templates. They also provide mechanisms for calling user-defined extension functions. What's new is the ability to declare user-defined functions *in XSLT*.

## Function Example

```
<xsl:function name="str:reverse" as="xs:string">
  <xsl:param name="sentence" as="xs:string"/>
  <xsl:sequence select="
if (contains($sentence, ' '))
  then concat(str:reverse(
                substring-after(
                  $sentence, ' '),
                ' '),
                substring-before($sentence, ' '))
  else $sentence"/>
</xsl:function>
```

This can be called *from XPath*: `select="str:reverse('DOG BITES MAN')"`.

# Stylesheet Modularity

- Stylesheets can be included or imported:
- **<xsl:include>** provides source code modularity.
- **<xsl:import>** provides logical modularity.
- **<xsl:apply-imports>** allows an importing stylesheet to apply templates from an imported stylesheet.
- What's new is **<xsl:next-match>**

## Next Match

- If several templates match, they are sorted by priority
- The highest priority template is executed, the others are not
- `<xsl:next-match>` allows a template to evaluate the *next highest* priority template.
- This is independent of stylesheet import precedence

## Access to Result Trees

In XSLT 1.0, result trees are read-only. If you construct a variable that contains some computed elements, you cannot access those elements.

Almost every implementation of XSLT 1.0 provided some sort of extension function to circumvent this limitation.

XSLT 2.0 removes this limitation. It is now possible to perform the same operations on result trees that you can perform on input documents.

# Result Documents

- XSLT 1.0 provides only a single result tree.
- Almost all vendors provided an extension mechanism to produce multiple result documents.
- XSLT 2.0 provides a **<xsl:result-document>** instruction to create multiple result documents.
- Provides support for validation.

# Sequences

The `<xsl:sequence>` instruction is used to construct sequences of nodes or atomic values (or both).

- `<xsl:sequence select='(1,2,3,4)'/>`  
returns a sequence of integers.
- `<xsl:sequence select='(1,2,3,4)'  
as="xs:double"/>`  
returns a sequence of doubles.

## Sequences (Continued)

The following code defines **\$prices** to contain a sequence of decimal values computed from the prices of each product.

```
<xsl:variable name="prices">
  <xsl:for-each select="$products/product">
    <xsl:choose>
      <xsl:when test="@price">
        <xsl:sequence select="xs:decimal(@price)"/>
      </xsl:when>
      <xsl:otherwise>
        <xsl:sequence select="xs:decimal(@cost) * 1.5"/>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:for-each>
</xsl:variable>
```

## Sequences (Continued)

So does this:

```
<xsl:value-of
  select="for $p in products return
        if ($p/@price)
        then xs:decimal($p/@price)
        else (xs:decimal($p/@cost) * 1.5)"/>
```

## Sequences (Continued)

Why is the former better? Because if you run the latter *stylesheet* through a processor, you'll get:

```
<xsl:value-of select="for $p in products  
return if ($p/@price) then xs:decim-  
al($p/@price) else (xs:decimal($p/@cost) *  
1.5)"/>
```

Which I find a little hard to read. My recommendation: use XSLT whenever you can.

# Values, Copies, and Sequences

What's the difference between **xsl:value-of**, **xsl:copy-of**, and **xsl:sequence**?

- **xsl:value-of** always creates a text node.
- **xsl:copy-of** always creates a copy.
- **xsl:sequence** returns the nodes selected, subject possibly to atomization. Sequences can be extended with **xsl:sequence**.

## Sequence Separators

You can add a separator when taking the value of a sequence:

```
<xsl:value-of select="(1, 2, 3, 4)"  
              separator="; "/>
```

produces “1; 2; 3; 4” (as a single text node).

## Formatting Dates

XPath 2.0 adds date, time, and duration types. XSLT 2.0 provides **format-date** to format them for presentation.

- **format-date()**

This is analogous to **format-number()**.

## Character Maps

Character maps give you greater control over serialization. They map a Unicode character to *any* string in the serialized document.

- For XML and HTML output methods, the resulting character stream does not have to be well-formed.
- The mapping occurs *only* at serialization: it is not present in result tree fragments.
- This facility can be used instead of “disabled output escaping” in most cases.

# Creating Entities

Suppose you want to construct an XHTML document that uses `&nbsp;` for non-breaking spaces, `&eacute;` for “é”, etc.

```
<xsl:output method="xml"
  doctype-public="-//W3C//DTD XHTML 1.0 Transitional//EN"
  doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"
  use-character-maps="example-map"/>

<xsl:character-map name="example-map">
  <xsl:output-character character="#233;"
    string="&eacute;"/>
  <xsl:output-character character="#160;"
    string="&nbsp;"/>
</xsl:character-map>
```

## Avoiding Disable Output Escaping

Suppose you want to construct a JSP page that contains:

```
<jsp:setProperty name="user" property="id"
                 value="'<%= "id" + idValue %>'"/>
```

Pick some otherwise unused Unicode characters to represent the character sequences that aren't valid XML. For example, “«” for “<%=”, “»” for “%>”, and “.” for the explicit double quotes:

```
<jsp:setProperty name="user" property="id"
                 value='« .id. + idValue »' />
```

## Avoiding D-O-E (Continued)

Construct a character map that turns those characters back into the invalid strings:

```
<xsl:character-map name="jsp">
  <xsl:output-character character="<"
    string="&lt;%" />
  <xsl:output-character character=">"
    string="%&gt;" />
  <xsl:output-character character="."
    string="' "' />
</xsl:character-map>
```

# Compatibility

- An XSLT 1.0 processor handles 2.0 stylesheets in *forwards compatibility mode*
- An XSLT 2.0 processor handles 1.0 stylesheets:
  - As a 1.0 processor, or
  - in *backwards compatibility mode*
- A mixed-mode stylesheet uses the appropriate mode.

Although the goal is that a 2.0 processor running a 1.0 stylesheet in backwards compatibility mode should produce the same results as a 1.0 processor, this is not guaranteed to be the case for all stylesheets.

# Challenges

Q: Function Results

A: Function Results

E: Function Results

Q: Counting Elements

A: Counting Elements

E: Counting Elements

Q: Matching Elements

A: Matching Elements

Q: Matching Elements 2

A: Matching Elements 2

## Q: Function Results

What does this stylesheet fragment produce?

```
<xsl:template match="/">
  <xsl:value-of select="xf:compare('apple', 'apple')"/>
  <xsl:text>#10;</xsl:text>
  <xsl:value-of select="xf:compare('apple', 'orange')"/>
  <xsl:text>#10;</xsl:text>
</xsl:template>
```

```
<xsl:function name="xf:compare">
  <xsl:param name="word1" as="xs:string"/>
  <xsl:param name="word2" as="xs:string"/>
  <xsl:text>"</xsl:text>
  <xsl:value-of select="$word1"/>
  <xsl:text>" and "</xsl:text>
  <xsl:value-of select="$word2"/>
  <xsl:text>" are </xsl:text>
  <xsl:if test="$word1 != $word2">not</xsl:if>

  <xsl:text> the same.</xsl:text>
</xsl:function>
```

## A: Function Results

It produces:

```
" apple " and " apple " are the same.
```

```
" apple " and " orange " are not the same.
```

Why?

## E: Function Results

Because:

1. The function returns a sequence of text nodes.
2. Atomization turns that into a sequence of strings.
3. And **xsl:value-of** uses the default separator, “ ”, between those strings.

One way to eliminate the “extra” spaces is to explicitly set the separator to the empty string:

```
<xsl:value-of select="xf:compare('apple', 'apple')"  
              separator="" />
```

# Q: Counting Elements

What does this stylesheet produce?

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns="http://www.w3.org/1999/xhtml"
  version="2.0">

  <xsl:output method="text"/>

  <xsl:param name="doc">
    <doc>
      <p>One</p>
      <p>Two</p>
      <p>Three</p>
    </doc>
  </xsl:param>

  <xsl:template match="/">
    <xsl:text>There are </xsl:text>
    <xsl:value-of select="count($doc//p)"/>
    <xsl:text> paras.&#10;</xsl:text>
  </xsl:template>

</xsl:stylesheet>
```

# A: Counting Elements

It produces:

**There are 0 paras.**

Why?

## E: Counting Elements

Because “**doc**” is in the default namespace and “**//p**” matches elements in no namespace.

This would work:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns="http://www.w3.org/1999/xhtml"
  xpath-default-namespace="http://www.w3.org/1999/xhtml"
  version="2.0">
  ...
```

So would this:

```
<xsl:value-of xmlns:x="http://www.w3.org/1999/xhtml"
  select="count($doc//x:p)"/>
```

## Q: Matching Elements

Write an XSLT stylesheet function that returns true if two elements “match”:

**f:element-matches(\$srcElem, \$targetElem)**

Two elements match if:

1. They have the same name (use the **node-name** function).
2. Every attribute on **\$srcElem** that is not in a namespace is also on **\$targetElem** and has the same value.
3. Namespace qualified attributes on **\$srcElem** are ignored.
4. Extra attributes are allowed on **\$targetElem**.

# A: Matching Elements

Here's one approach:

```
<xsl:function name="f:element-matches" as="xs:boolean">
  <xsl:param name="srcElement" as="element()"/>
  <xsl:param name="targetElem" as="element()"/>
  <xsl:choose>
    <xsl:when test="node-name($srcElement) = node-name($targetElem)">
      <xsl:variable name="attrMatch">
        <xsl:for-each select="$srcElement/@*[namespace-uri(.) = '']">
          <xsl:variable name="aname" select="local-name(.)"/>
          <xsl:variable name="attr" select="$targetElem/@*[local-name(.) = $aname]"/>
          <xsl:choose>
            <xsl:when test="$attr = .">1</xsl:when>
            <xsl:otherwise>0</xsl:otherwise>
          </xsl:choose>
        </xsl:for-each>
      </xsl:variable>
      <xsl:value-of select="not(contains($attrMatch, '0'))"/>
    </xsl:when>
    <xsl:otherwise><xsl:value-of select="false()"/></xsl:otherwise>
  </xsl:choose>
</xsl:function>
```

## Q: Matching Elements 2

That function can be replaced by a single XPath 2.0 expression.  
What is it?

## A: Matching Elements 2

```
(node-name($srcElem) = node-name($targetElem))
and (every $i in $srcElem/@*[namespace-uri(.) = '']
    satisfies
    for $j in $targetElem/@*[local-name(.) = local-name($i)]
    return $i = $j
```

# Closing Thoughts

Is it Worth It?

Acknowledgements

References

## Is it Worth It?

XPath 2.0 and XSLT 2.0 are larger and more complex than their predecessors. Schema support and “strong” typing will catch errors, but they will also force more explicit casting.

Does it make sense to start developing for 2.0?

Yes, I think so. Grouping, regular expressions, user-defined functions, character maps, standardized access to result documents and multiple result documents are all going to make stylesheet writing easier.

# Acknowledgements

Many of the examples in this tutorial are taken directly from the XSLT 2.0 Specification.

Michael Kay generously shared a number of examples that he uses in his own tutorials.

Jeni Tennison's *Typing in Transformations* paper from *Extreme Markup Languages 2003* was instrumental in refreshing my memory about the issues surrounding XPath 2.0 casting rules.

David Carlisle suggested the XPath replacement for matching elements.

## References

*XQuery 1.0 and XPath 2.0 Data Model*, <http://www.w3.org/TR/xpath-datamodel/>. Mary Fernández, Ashok Malhotra, Jonathan Marsh, *et. al.*, editors. World Wide Web Consortium. 2003.

*XQuery 1.0 and XPath 2.0 Functions and Operators*, <http://www.w3.org/TR/xpath-functions/>. Ashok Malhotra, Jim Melton, and Norman Walsh, editors. World Wide Web Consortium. 2003.

*XQuery 1.0 and XPath 2.0 Formal Semantics*, <http://www.w3.org/TR/xquery-semantics/>. Denise Draper, Peter Frankhauser, Mary Fernández, *et. al.*, editors. World Wide Web Consortium. 2003.

## References (Continued)

*XML Path Language (XPath) 2.0*, <http://www.w3.org/TR/xpath20/>. Anders Berglund, Scott Boag, Don Chamberlin, *et. al.*, editors. World Wide Web Consortium. 2003.

*XQuery 1.0: An XML Query Language*, <http://www.w3.org/TR/xquery/>. Scott Boag, Don Chamberlin, Mary Fernández, *et. al.*, editors. World Wide Web Consortium. 2003.

*XSL Transformations (XSLT) Version 2.0*, <http://www.w3.org/TR/xslt20/>. Michael Kay, editor. World Wide Web Consortium. 2003.

*XSLT 2.0 and XQuery 1.0 Serialization*, <http://www.w3.org/TR/xslt-xquery-serialization/>. Michael Kay,



## References (Continued)

Norman Walsh, and Henry Zongaro, editors. World Wide Web Consortium. 2003.

*Typing in Transformations*, <http://www.idealliance.org/papers/extreme03/html/2003/Tennison01/EML2003Tennison01-toc.html>. Jeni Tennison. Extreme Markup Languages. 2003.



# XPath 2.0 and XSLT 2.0

**Norman Walsh**  
**Norman.Walsh@Sun.COM**

<http://www.sun.com/>



Version 1.0