summer school

Improving Existing XSLT Stylesheets

19 September 2013

Improving Existing XSLT Stylesheets

Priscilla Walmsley

Datypic, Inc.

summer school

Contents

1. Grouping Structure
2. Regular Expression Matching
3. Modularization
4. Using Types

summer school

GROUPING

Grouping

- Difficult in XSLT 1.0; usually used keys
- **`xsl:for-each-group`** element allows you to iterate through groups
  - **`select`** attribute identifies items to group
  - **`group-by`** attribute specifies the grouping key
- Two functions can be used within **`for-each-group`**:
  - **`current-group()`** returns members of current group
  - **`current-grouping-key()`** returns the current grouping key

Grouping by Value

summer school

```
<catalog>
  <product dept="WMN">...</product>
  <product dept="ACC">...</product>
  <product dept="ACC">...</product>
  <product dept="MEN">...</product>
</catalog>
```

```
<xsl:template match="/">
  <RESULTS>
    <xsl:for-each-group select="catalog/product"
                        group-by="@dept" >
      <xsl:sort select="current-grouping-key()" />
      <DEPT name="{current-grouping-key()}"
            prodCount="{count(current-group())}" />
    </xsl:for-each-group>
  </RESULTS>
</xsl:template>
```

```
<RESULTS>
   <DEPT name="ACC" prodCount="2"/>
   <DEPT name="MEN" prodCount="1"/>
   <DEPT name="WMN" prodCount="1"/>
</RESULTS>
```

Other Grouping Options

- **`group-adjacent`**
  - groups adjacent items with the same key together
- **`group-starting-with`**
  - creates a group of items starting with the specified element
- **`group-ending-with`**
  - creates a group of items ending with the specified element

summer school

Grouping by Starting Element

Input document

```
<body>
 <h1>Chapter 1</h1>
 <h2>Section 1.1</h2>
 <p>In this section...</p>
 <p>More text</p>
 <h2>Section 1.2</h2>
 <p>In this section...</p>
</body>
```

Desired output document

```
<section level="1">
  <heading>Chapter 1</heading>
  <section level="2">
    <heading>Section 1.1</heading>
    <p>In this section...</p>
    <p>More text</p>
  </section>
  <section level="2">
    <heading>Section 1.2</heading>
    <p>In this section...</p>
  </section>
</section>
```

summer school

Grouping by Starting Element

```
<xsl:template match="/">
  <xsl:for-each-group select="body/*" group-starting-with="h1">
   <section level="1">
    <xsl:for-each-group select="current-group()"
                        group-starting-with="h2">
    <xsl:choose>
      <xsl:when test="current-group()[self::h2]">
        <section level="2">
         <xsl:apply-templates select="current-group()"/>
        </section>
      </xsl:when>
      <xsl:otherwise>
        <xsl:apply-templates select="current-group()"/>
      </xsl:otherwise>
    </xsl:choose>
    </xsl:for-each-group>
   </section>
  </xsl:for-each-group>
</xsl:template>
```

Grouping Adjacent Items

Input document

```
<body>
 <p>The following...:</p>
 <p>1. Open the file.</p>
 <p>2. Change it to...</p>
 <p>3. Save the file.</p>
 <p>As you can see...</p>
</body>
```

Desired output document

```
<body>
  <p>The following...:</p>
  <ol>
    <li>Open the file.</li>
    <li>Change it to...</li>
    <li>Save the file.</li>
  </ol>
  <p>As you can see...</p>
</body>
```

Grouping Adjacent Items

```
<xsl:template match="body">
 <body>
  <xsl:for-each-group select="*" group-adjacent="my:is-a-list-item(.)">
   <xsl:choose>
    <xsl:when test="current-grouping-key() = true()">
     <ul>
      <xsl:for-each select="current-group()">
       <li><xsl:apply-templates/></li>
      </xsl:for-each>
     </ul>
    </xsl:when>
    <xsl:otherwise>
     <xsl:copy-of select="."/>
    </xsl:otherwise>
   </xsl:choose>
  </xsl:for-each-group>
 </body>
</xsl:template>
```

Grouping Adjacent Items (cont.)

```
<xsl:template match="text()">
  <xsl:choose>
    <xsl:when test="my:is-a-list-item(parent::p)
                    and my:is-a-list-item(.)
                    and (. is parent::p/node()[1])">
      <xsl:value-of select="replace(.,'^\s*\d+\.\s*','')"/>
    </xsl:when>
    <xsl:otherwise>
       <xsl:copy-of select="."/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>

<xsl:function name="my:is-a-list-item">
  <xsl:param name="node"/>
  <xsl:sequence select="matches($node,'^\s*\d+\.')"/>
</xsl:function>
```

summer school

REGULAR EXPRESSION MATCHING

Recognizing Textual Patterns

- XSLT 2.0 has excellent regular expression support
- **`xsl:analyze-string`** element splits string into matching and non-matching parts, based on a regex
  - **`xsl:matching-substring`** child specifies what to do with matching parts
  - **`xsl:non-matching-substring`** child specifies what to do with non-matching parts

xsl:analyze-string Example

summer school

```
<xsl:function name="my:markUpPhone">
 <xsl:param name="theText"/>
 <xsl:analyze-string select="$theText"
                     regex="[0-9]{{3}}/[0-9]{{3}}-[0-9]{{4}}">
  <xsl:matching-substring>
   <phone>
    <xsl:value-of select="."/>
   </phone>
  </xsl:matching-substring>
  <xsl:non-matching-substring>
   <xsl:value-of select="."/>
  </xsl:non-matching-substring>
 </xsl:analyze-string>
</xsl:function>
```

can be reached at 231/555-1212 or...

can be reached at <phone>231/555-1212</phone> or...

summer school

The **regex-group** Function

```
<xsl:function name="my:markUpPhone">
 <xsl:param name="theText"/>
 <xsl:analyze-string select="$theText"
                regex="([0-9]{{3}})/([0-9]{{3}}-[0-9]{{4}})">
  <xsl:matching-substring>
   <phone>
    <areaCode><xsl:value-of select="regex-group(1)"/></areaCode>
    <number><xsl:value-of select="regex-group(2)"/></number>
   </phone>
  </xsl:matching-substring>  ....
```

can be reached at 231/555-1212 or...

can be reached at
<phone><areaCode>231</areaCode><number>555-1212</number></phone> or...

The **matches** Function

matches

whether a string matches a regular expression

```
matches("query", "^qu")  ==> true
```

uses the XML Schema regex syntax (similar to Perl)

optional third "flags" argument allows for interpretation of regular expression

- case sensitivity
- multi-line mode
- whitespace sensitivity

The **tokenize** Function

tokenize

delimiter specified as a regular expression

returns a sequence of strings

```
tokenize("a b c", "\s")
  ==> ("a", "b", "c")
```

```
tokenize("2006-12-25T12:15:00", "[\-T:]")
  ==> ("2006","12","25","12","15","00")
```

summer school

The **replace** Function

Arguments are:

- the string to be manipulated
- the regular expression
- the replacement string

replace("query", "r", "as")	queasy
replace("query", "qu", "quack")	quackery
replace("query", "[ry]", "l")	quell
replace("query", "[ry]+", "l")	quel

Replacing with Subexpressions

Use **$1, $2,** etc. in replacement string to insert strings that matched parenthesized subexpressions

```
replace("Chap 2...Chap 3...Chap 4...",
        "Chap (\d)", "Sec $1.0")
  ==> "Sec 2.0...Sec 3.0...Sec 4.0..."
```

```
replace("2006-10-18", "\d{2}(\d{2})-(\d{2})-(\d{2})", "$2/$3/$1")
  ==> "10/18/06"
```

summer school

MODULARIZATION

Temporary Trees

- Variables can contain element structures that can be accessed using XPath
- Useful for:
  - defining lookup tables
  - simplifying queries, especially with multi-step processes
- In 1.0 this was generally handled by a **node-set** extension function

Lookup Table Example

```
<xsl:variable name="dept_names" >
  <Dep code="ACC" name="Accessories" />
  <Dep code="MEN" name="Men's" />
  <Dep code="WMN" name="Women's" />
</xsl:variable>
<xsl:for-each-group select="catalog/product"
                    group-by="@dept" >
  <DEPT name="{$dept_names/Dep[@code=current-grouping-key()]/@name}">...</DEPT>
 </xsl:for-each-group>
```

```
<DEPT name="Accessories">...</DEPT>
<DEPT name="Men's">...</DEPT>
<DEPT name="Women's">...</DEPT>
```

Making Multiple Passes Example

```
<xsl:variable name="after-pass1" >
 <xsl:apply-templates mode="pass1"/>
</xsl:variable>

<xsl:variable name="after-pass2" >
 <xsl:apply-templates select="$after-pass1" mode="pass2"/>
</xsl:variable>

<xsl:apply-templates select="$after-pass2"/>
```

Modes in XSLT 2.0

Multiple modes can be specified in template

#current can be used to pass current mode

```
<xsl:template match="topic" mode="pass1 pass2 #default">
  <!-- do something -->
  <xsl:apply-templates mode="#current"/>
</xsl:template>
```

#all keyword can be used to match all modes

```
<xsl:template match="topic" mode="#all">
  <!-- do something -->
</xsl:template>
```

xsl:apply-imports and xsl:next-match

Used to call an overridden template

- applies imported templates to the *current* node (not the children)

Usually used to create new preceding or wrapping elements and then process the elements normally

xsl:apply-imports (available in 1.0)

- only looks at imported templates

xsl:next-match (2.0 only)

- looks at all templates of lower precedence

summer school

xsl:apply-imports

```
<xsl:stylesheet version="1.0" xmlns:xsl= "...">
<xsl:import href="t2.xsl"/> ...
   <xsl:template match="example">
    <a name="xx"/>
    <div style="border: solid red">
        <xsl:apply-imports/>
    </div>
   </xsl:template>
</xsl:stylesheet>
```

```
<div style="border: solid red">
 <pre>...</pre>
</div>
```

```
<xsl:stylesheet version="1.0" xmlns:xsl= "...">
   <xsl:template match="example">
    <pre><xsl:value-of select="."/></pre>
   </xsl:template>
</xsl:stylesheet>
```

summer school

xsl:next-match

```
<xsl:stylesheet version="2.0" xmlns:xsl= "...">
  <xsl:template match="example">
    <example-wrap>
      <xsl:next-match/>
    </example-wrap>
  </xsl:template>

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

</xsl:stylesheet>
```

```
<example-wrap>
 <example>...</example>
</example-wrap>
```

summer school

USING TYPES

summer school

Specifying Types in XSLT

You can specify types in your XSLT using an **as** attribute

- Values must conform to that type
- Helpful for debugging

```
<xsl:variable name="prods" as="node()*" select="product"/>

<xsl:param name="prodCount" as="xs:integer"/>

<xsl:function name="getPrice" as="xs:decimal">...

<xsl:template name="createList" as="element(ul)">...
```

Sequence Types

The **as** attribute is a *sequence type*

Most common are simple type names

e.g. **xs:string**, **xs:integer**, **xs:double, xs:date**

argument must be an atomic value of that type

Other sequence types:

item(), node(), element(), attribute()

Sequence Types and Occurrence Indicators

Occurrence indicators indicate how many items are allowed:

? for zero or one items

* for zero, one or many items

+ for one or many items

no indicator for one and only one item

```
<xsl:function name="getProdCount" as="xs:integer">
  <xsl:param name="prods" as="element()*"/>...
```

param can be zero, one or more elements

return type is always a single integer

Conclusions

- Using advanced features of XSLT 2.0 can:
  - greatly simplify your XSLT code
  - make it easier to maintain
  - make it easier to debug
  - make it perform faster
  - allow it to be more flexible
  - produce better output
- Questions?

Any questions?

Thank you for your attention.

Priscilla Walmsley
http://www.datypic.com
pwalmsley@datypic.com