summer school

XSLT 3.0 for Daily Coding

Jirka Kosek
<jirka@kosek.cz>

XML Summer School, Oxford

September 20th, 2019

Slides: https://kosek.cz/xml/2019xmlss/
Examples: https://kosek.cz/xml/2019xmlss/examples.zip
XMLGuru.cz & XML Prague

Table of Contents

Agenda

- become familiar with a lot of small XSLT 3.0 features that improve productivity and readability
- practice, practice, practice, …

More readable XPath expressions

New XPath operators

- simple map operator (!)

```
('a', 'b', 'c') ! upper-case(.)    --> 'A', 'B', 'C'
```

- arrow operator (⇒)

```
'hello' => upper-case()            --> 'HELLO'
```

- string concatenation (||)

```
'a' || 'b' || 'c'                  --> 'abc'
```

Exercise

- in practice it’s necessary to apply chain of function to string
- for example getting filename from article title

```
<title>Kvůli hrozící srážce změnila ESA dráhu své družice.</title>
                          ⇩
kvuli_hrozici_srazce_zmenila_esa_drahu_sve_druzice.html
```

- this is a chain of the following simple operations:
  - remove any diacritics
  - remove any punctuation
  - convert spaces to underscores
  - convert to lowercase
  - add extension .html

Exercise

- easy to write in XPath, not so easy to read

```
concat(
  lower-case(
    replace(
      replace(
        replace(
          normalize-unicode(
            $item/title,
            'NFKD'),
          '\p{Mn}', ''),
        '\p{P}', ''),
      '\s+', '_')
    ),
  '.html')
```

- try to make `feed2files.xsl` more readable using new operators

More readable templates

Text value templates

- in XSLT 2.0 content of a text node could be computed using `xsl:value-of`

```
<h1><xsl:value-of select="title"/></h1>
```

- for values in attributes it was possible to use attribute value templates inside curly braces

```
<h1 id="{generate-id()}"><xsl:value-of select="title"/></h1>
```

- in XSLT 3.0 curly braces can be also used to produce text nodes if enabled by using `expand-text="yes"`

```
<xsl:stylesheet ...
                expand-text="yes">
    ...
  <h1 id="{generate-id()}">{title}</h1>
```

- text value templates can be enabled/disabled on any element
- on literal result elements prefixed version `xsl:expand-text="…"` must be used
- be careful with embedded JS and CSS code

```
<style type="text/css" xsl:expand-text="no">
  h1 { color: navy; }
</style>
```

Exercise

- try to make `po2html.xsl` more concise using text value templates
- check that `{` is properly handled inside any JS/CSS blocks
- try to make stylesheet even more concise

Conditional content construction (xsl:where-populated)

- sometimes wrapper elements should be output only when there is content

```
<xsl:if test="item">
  <table>
    <xsl:for-each select="item">
      <tr>...</tr>
    </xsl:for-each>
  </table>
</xsl:if>
```

- such approach prevents streaming and could be verbose
- content of `xsl:where-populated` is output only when there are some grandchild nodes

```
<xsl:where-populated>
  <table>
    <xsl:for-each select="item">
      <tr>...</tr>
    </xsl:for-each>
  </table>
</xsl:where-populated>
```

Conditional content construction (xsl:on-empty/xsl:on-non-empty)

- content of `xsl:on-empty` is output only when preceding siblings do not produce anything

```
<xsl:sequence>
  <xsl:for-each select="item">
    ... process items ...
  </xsl:for-each>
  <xsl:on-empty>
    <p>No items to process!</p>
  </xsl:on-empty>
</xsl:sequence>
```

  - `xsl:on-empty` can be used only once in one sequence constructor
  - `xsl:on-empty` must be at the end of sequence constructor
- content of `xsl:on-non-empty` is output only when siblings (other than `xsl:on-empty`/`xsl:on-non-empty`) produce anything
  - `xsl:on-non-empty` can be used multiple times in one sequence constructor
  - `xsl:on-non-empty` can be positioned anywhere in a sequence constructor

Exercise

- try to rewrite `po2html-list-empty.xsl` using instructions for conditional content construction
  - for testing use both `po.xml` and `po-empty.xml` files
- try to rewrite `po2html-empty.xsl` using instructions for conditional content construction
  - result will be more likely less readable then original code
  - such approach still has use when doing streaming processing

Easily parameterized stylesheets

Shadow attributes

- for legacy and backward compatibility reasons not all attributes could use attribute value templates
  - legal use:

```
<xsl:sort order="{$sort-order}"/>
```

  - illegal use:

```
<xsl:sort select="{$key}"/>
```

    `select` attribute is not AVT it should contain expression directly — in such case all items will be sorted using the same key — value inside `$key` parameter
- shadow attribute name starts with _ (underscore) and replaces value of "normal" attribute
- only static expressions could be used in value templates for shadow attributes

```
<xsl:param name="key" static="yes" select="'price'"/>

<xsl:sort _select="{$key}"/>
```

- `xsl:evaluate` can be used if static expressions are too limiting

Dynamic evaluation

- `xsl:evaluate` takes string and evaluates it as XPath expression
- be very careful about security implications

```
<xsl:sort>
  <xsl:evaluate xpath="$key" context-item="."/>
</xsl:sort>
```

More useful bits

Random number generator

- `random-number-generator()?number` returns random number between 0 and 1
- `random-number-generator()?permute(`*sequence*`)` returns *sequence* in a randomized order

Initial template

- template named `xsl:initial-template` will be invoked first

```
<xsl:template name="xsl:initial-template">
  <!-- Do something -->
</xsl:template>
```

- useful when input is not XML or when stylesheet is just generating output

JSON

- more and more data sets are available in JSON instead of XML
- XSLT 3.0 offers two ways how to process JSON
- `json-to-xml()` and `xml-to-json()` can map JSON to XML (and vice versa)
  - mapping is generic and universal, but not always concise and intuitive
  - once JSON is converted to XML it can be processed by templates and accessed by XPath
- `parse-json()` and `json-doc()` functions map JSON to maps and arrays
  - more natural and 'JSONish' approach
  - lookup operator (`?`) provides easy navigation

```
JS:    $json.results[0].geometry.location.lat
XPath: $json?results(1)?geometry?location?lat
```

caution: array index in XPath is not zero-based

Exercise

- write transformation that would display random quote from `quotes.json`
- **Hint:**
  - JSON file contains array of quotes
  - function `array:flatten()` can be used for converting array to sequence
  - array functions are available in the separate namespace `http://www.w3.org/2005/xpath-functions/array`

Identity transform reloaded

- empty transformation copies just text nodes
- for identity transformation we have to add identity template

```
<xsl:template match="node()">
  <xsl:copy>
    <xsl:copy-of select="@*"/>
    <xsl:apply-templates/>
  </xsl:copy>
</xsl:template>
```

- `xsl:mode` can be used to change default behaviour

```
<xsl:mode on-no-match="shallow-copy"/>
```

  - `deep-copy` — tree is copied at once
  - `shallow-copy` — tree is copied node by node (individual nodes could be processed by custom templates)
  - `deep-skip` — whole tree is skipped
  - `shallow-skip` — tree is skipped node by node
  - `text-only-copy` — only text nodes are copied (default)
  - `fail` — transformation will fail if there is no template for some node

Another goodies

- `contains-token()` — easy matching of HTML/DITA classes

```
<div class="dark-theme navbar cols-2">
 ...
</div>

<xsl:template match="*[contains-token(@class, 'navbar')]>
```

- finally there are trigonometric and exponential functions — https://www.w3.org/TR/xpath-functions-31/#trigonometry
- `head()` and `tail()` — easier recursive processing of sequences
- `parse-xml()` and `serialize()` — parsing and serialization from/to string
- `load-xquery-module()` — access functions and variables from XQuery module
- `transform()` — invokes dynamically-loaded stylesheet
- `path()` — returns XPath selecting supplied node
- `analyze-string()` — function that matches text against regular expression and returns result as an XML structure suitable for further processing

Power of xsl:iterate

Functional nature of XSLT

- XSLT is a functional programming language
- functions/instructions should not have any side-effects
- data structures are immutable
- recursion is often used to overcome limitations of the language
- issues with recursion
  - harder to understand for some developers
  - can be memory ineffective (unless it is tail-recursion)
- in XSLT 1.0 even simple tasks often required recursive named templates
- in XSLT 2.0/3.0 need for recursion is smaller as more can be computed directly in XPath
- still complex tasks require recursion

Example — calculating factorial

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:f="http://example.com/functions"
  exclude-result-prefixes="xs f"
  version="3.0">

  <xsl:output method="text"/>

  <!-- Classical recursive approach for calculating factorial -->
  <xsl:function name="f:factorial" as="xs:integer">
    <xsl:param name="n" as="xs:integer"/>

    <xsl:sequence select="if ($n gt 1) then $n * f:factorial($n - 1) else 1"/>
  </xsl:function>

  <!-- Factorial can be calculated without recursion with HoF -->
  <xsl:function name="f:factorial-hof" as="xs:integer">
    <xsl:param name="n" as="xs:integer"/>

    <xsl:sequence select="fold-left((1 to $n), 1, function($a, $b) { $a * $b})"/>
  </xsl:function>

  <!-- Factorial using xsl:iterate -->
  <xsl:function name="f:factorial-iterate" as="xs:integer">
```

Example — calculating factorial (Continued)

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

    <xsl:iterate select="1 to $n">
      <xsl:param name="result" select="1"/>
      <xsl:on-completion select="$result"/>
      <xsl:next-iteration>
        <xsl:with-param name="result" select="$result * ."/>
      </xsl:next-iteration>
    </xsl:iterate>
  </xsl:function>

  <xsl:template name="xsl:initial-template">
    <xsl:value-of select="f:factorial(30)"/>
    <xsl:text>&#xA;</xsl:text>
    <xsl:value-of select="f:factorial-hof(30)"/>
    <xsl:text>&#xA;</xsl:text>
    <xsl:value-of select="f:factorial-iterate(30)"/>
  </xsl:template>

</xsl:stylesheet>
```

xsl:iterate

- just syntactic sugar
  - simplicity of `xsl:for-each` combined with parameter passing
  - guarantees tail-recursion (memory efficiency)
  - can't be always used

```
<xsl:iterate select="sequence">
  <!-- Parameters that are passed from iteration to iteration -->
  <xsl:param name="a" value="initial value"/>

  <xsl:on-completion>
     <!-- content returned when iteration is done -->
  </xsl:on-completion>

  <!-- sequence constructor -->

  <xsl:if test="premature termination is needed?">
    <xsl:break/>
  </xsl:if>

  <!-- For next iteration value inside parameters can be updated -->
  <xsl:next-iteration>
    <xsl:with-param name="a" select="new value"/>
  </xsl:next-iteration>
</xsl:iterate>
```

Example — running totals

```
<xsl:iterate select="purchase-order/item">
  <xsl:param name="total" select="0"/>
  <xsl:on-completion>
    <tr>
      <th colspan="3">Total Sum</th>
      <th>
        £ <xsl:value-of select="$total"/>
      </th>
    </tr>
  </xsl:on-completion>
  <tr>
    <td><xsl:value-of select="name"/></td>
    <td><xsl:value-of select="qty"/></td>
    <td>£ <xsl:value-of select="price"/></td>
    <td>£ <xsl:value-of select="qty * price"/></td>
  </tr>
  <xsl:next-iteration>
    <xsl:with-param name="total" select="$total + qty * price"/>
  </xsl:next-iteration>
</xsl:iterate>
```

Exercise

- study recursive function in `text2lines.xsl` for splitting long text into lines
- try to write more effective version of this code using `xsl:iterate`

Questions & References

- https://www.w3.org/TR/xslt-30/
- https://www.w3.org/TR/xpath-functions-31/
- https://www.w3.org/TR/xpath-31/