XSLT repetition constructs

Author: Mukul Gandhi

Last modified: 2019-03-21

Introduction

XSLT is a language primarily used to transform input XML / text documents to other XML / HTML / text documents. XSLT 3.0 is the latest version of XSLT language, released as a W3C Recommendation during June 2017. In this document, I'll attempt to compare XSLT constructs specified, to write repetition logic in the XSLT stylesheets. By repetition logic, we mean repeated execution of a block of code, similar in concept to a procedural language loop (for e.g the C language's "for" loop). My analysis and comparison, is based on assumptions of 'non schema aware' and 'non streaming' stylesheets. XML Schema aware stylesheets were first introduced in the XSLT 2.0 language, whereas the ability to write streaming stylesheets was introduced in the XSLT 3.0 language. XML Schema aware stylesheets mean, that we can make use of XML Schema data types and the whole of XML Schema documents, into XSLT transformations. Whereas streaming XSLT transformations mean, that we can transform XML input documents that can be very large or even large than what can be stored in the computer's local storage.

Discussed below, are various XSLT constructs that are available to write repetition logic in the XSLT stylesheets.

1) xsl:iterate

This instruction is newly introduced in the XSLT 3.0 language. It's formally discussed at, https://www.w3.org/TR/xslt-30/#iterate in the XSLT 3.0 specification. Borrowing the text from XSLT 3.0 specification, syntax of xsl:iterate instruction is defined as following,

```
<!-- Category: instruction -->
<xsl:iterate
    select = expression >
    <!-- Content: (xsl:param*, xsl:on-completion?, sequence-constructor) -->
</xsl:iterate>

<!-- Category: instruction -->
<xsl:next-iteration>
    <!-- Content: (xsl:with-param*) -->
</xsl:next-iteration>
```

```
<!-- Category: instruction -->
<xsl:break
    select? = expression >
    <!-- Content: sequence-constructor -->
</xsl:break>

<xsl:on-completion
    select? = expression >
    <!-- Content: sequence-constructor -->
</xsl:on-completion>
```

The execution of xsl:iterate instruction starts from the xsl:iterate element, and every other part of this instruction is either an attribute of xsl:iterate or child / descendant of xsl:iterate. *The XSLT 3.0 specification further tells us following about xsl:iterate instruction,*

"The 'select' attribute is required. It contains an expression which is evaluated to produce a sequence, called the input sequence.

The 'sequence constructor' contained in the xsl:iterate instruction is evaluated once for each item in the input sequence, in order, or until the loop exits by evaluating an xsl:break instruction, whichever is earlier. Within the sequence constructor that forms the body of the xsl:iterate instruction, the 'context item' is set to each item from the value of the select expression in turn, the 'context position' reflects the position of this item in the input sequence, and the 'context size' is the number of items in the input sequence."

Let us see an example below of xsl:iterate instruction.

XML input document:

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
    <val>1</val>
    <val>2</val>
    <val>3</val>
    <val>0</val>
    <val>0</val>
    <val>4</val>
</root>
```

XSLT stylesheet:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="3.0">

    <xsl:output method="xml" indent="yes"/>

    <xsl:template match="root">
        <result>
            <xsl:iterate select="val">
                <xsl:choose>
                    <xsl:when test=". = 0">
                        <xsl:break/>
                    </xsl:when>
                    <xsl:otherwise>
                        <xsl:copy-of select="."/>
                    </xsl:otherwise>
                </xsl:choose>
            </xsl:iterate>
        </result>
    </xsl:template>

</xsl:stylesheet>
```

The XSLT transformation for the above inputs produces following result,

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
    <val>1</val>
    <val>2</val>
    <val>3</val>
</result>
```

The XSLT transformation logic for above example, essentially does following:
Iterate through each “val” element, and copy the “val” element to output if value contained in it is not 0, otherwise break from the xsl:iterate loop. Although, the input sequence for an xsl:iterate contains all the “val” elements in order, just when a “val” element is found that contains the value 0, the execution moves to an instruction just after xsl:iterate (i.e, execution exits from xsl:iterate).

When using XSLT 1.0 or 2.0 processor, can we achieve iteration and break as illustrated in above XSLT 3.0 example? A XSLT stylesheet writer, may naively try to write following 1.0 or 2.0 stylesheet, to achieve same objective,

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">

  <xsl:output method="xml" indent="yes"/>

  <xsl:template match="root">
      <result>
        <xsl:for-each select="val[. != 0]">
          <xsl:copy-of select="."/>
        </xsl:for-each>
      </result>
  </xsl:template>

</xsl:stylesheet>
```

In the above XSLT stylesheet, xsl:for-each is used to do the repetition, with an expression val[. != 0] as input. i.e all “val” elements containing anything but 0 will be processed by the body of xsl:for-each. When we process the same input XML, with the above XSLT stylesheet with an XSLT 1.0 processor, we get following output,

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
   <val>1</val>
   <val>2</val>
   <val>3</val>
   <val>4</val>
</result>
```

We can see that, above output contains “val” elements that occur after those “val” elements that contain value 0, and which contain non 0 values. This is not expected, since we wanted to ignore all “val” elements starting from the one containing value 0.

The following XSLT 1.0 or 2.0 stylesheet, achieves what xsl:iterate has achieved in the above example,

```
<?xml version="1.0" encoding="UTF-8"?>
```

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

    <xsl:output method="xml" indent="yes"/>

    <xsl:template match="root">
        <result>
          <xsl:call-template name="emitVal">
              <xsl:with-param name="val" select="val[1]"/>
          </xsl:call-template>
        </result>
    </xsl:template>

    <xsl:template name="emitVal">
        <xsl:param name="val"/>

        <xsl:if test="$val != 0">
            <xsl:copy-of select="$val"/>
            <xsl:call-template name="emitVal">
               <xsl:with-param name="val" select="$val/following-sibling::val[1]"/>
            </xsl:call-template>
        </xsl:if>
    </xsl:template>

</xsl:stylesheet>
```

The above stylesheet uses, a recursive named template ('emitVal') to process each following sibling "val" items in turn. The recursion is exited, when a "val" element with value 0 is encountered. The above XSLT 1.0 stylesheet produces following output, when using the same input XML document,

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
   <val>1</val>
   <val>2</val>
   <val>3</val>
</result>
```

The above output is same, as that produced by the XSLT 3.0 example mentioned above.

Can we now envision, if XSLT 3.0's xsl:iterate instruction has greater value than XSLT 1.0 or 2.0's recursive named template? I can think of following two improvements available in xsl:iterate, as per the above use cases that are discussed:

a) xsl:iterate instruction is not recursive. Generally speaking, recursion imposes more memory overheads as compared to a linear 'for' loop.
b) While writing a XSLT stylesheet using xsl:iterate, we don't have to come up with a right XPath axis (following-sibling in case of recursive named template example above) for processing. Thinking along the XPath axes, is usually harder than thinking about a linear xsl:iterate instruction.

But if there are any business or technical reasons, that the logic has to be written in XSLT 1.0, then recursive named template would be the only choice for the use case that is discussed above.

Lets look at another xsl:iterate example, where the current iteration can pass values (usually different than the current iteration) as parameters to the next iteration. Following is an XSLT 3.0 stylesheet, working on the same input XML document as examples above,

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="3.0">

  <xsl:output method="xml" indent="yes"/>

  <xsl:template match="root">
      <result>
        <xsl:iterate select="val">
          <xsl:param name="running-total" select="0"/>
          <val running-total="{$running-total + .}"><xsl:value-of select="."/></val>
          <xsl:next-iteration>
              <xsl:with-param name="running-total" select="$running-total + ."/>
          </xsl:next-iteration>
        </xsl:iterate>
      </result>
  </xsl:template>

</xsl:stylesheet>
```

With the above stylesheet, during a particular iteration with xsl:iterate, the output "val" element has an attribute with value equal to sum of previous "val" values and the current "val" element.

When the above stylesheet is run with input XML, the following output is produced,

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
    <val running-total="1">1</val>
    <val running-total="3">2</val>
    <val running-total="6">3</val>
    <val running-total="6">0</val>
    <val running-total="6">0</val>
    <val running-total="10">4</val>
 </result>
```

Interestingly, the above mentioned stylesheet logic can be achieved with following XSLT 1.0 or 2.0 stylesheet, using a recursive named template,

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">

  <xsl:output method="xml" indent="yes"/>

  <xsl:template match="root">
      <result>
          <xsl:call-template name="emitValWithRunningTotal">
             <xsl:with-param name="val" select="val[1]"/>
             <xsl:with-param name="running-total" select="0"/>
           </xsl:call-template>
      </result>
  </xsl:template>

  <xsl:template name="emitValWithRunningTotal">
      <xsl:param name="val"/>
      <xsl:param name="running-total"/>

      <xsl:if test="$val">
          <val running-total="{$running-total + $val}"><xsl:value-of select="$val"/></val>
          <xsl:call-template name="emitValWithRunningTotal">
             <xsl:with-param name="val" select="$val/following-sibling::val[1]"/>
             <xsl:with-param name="running-total" select="$running-total + $val"/>
          </xsl:call-template>
      </xsl:if>
```

```
  </xsl:template>

</xsl:stylesheet>
```

The above mentioned XSLT stylesheet, has same pros and cons as mentioned in the previous XSLT example using recursive named template.

In general, using xsl:iterate would simplify XSLT iteration logic for numerous use cases, that would otherwise require recursive processing or a XSLT for-each loop written in a special way (which will usually be harder).

2) xsl:for-each

This is another well defined way in the XSLT language (available in all the versions of XSLT language), to write repetition logic.

The XSLT language says, that each iteration of xsl:for-each loop can even be computed in parallel and the results can be concatenated later. The specification of xsl:iterate in the XSLT 3.0 language, doesn't mention any such features.

The section of xsl:iterate above, specifies examples of XSLT for-each loops as well.

3) Recursive functions, and recursive named templates

Both, recursive functions (using xsl:function) and recursive named templates (using xsl:template with a 'name') can be used to write repetition logic in the XSLT language. xsl:function has been available since XSLT 2.0 and is also available in XSLT 3.0, whereas recursive named templates has been available since 1.0 version of the XSLT language. Few examples of recursive named templates, have been mentioned in sections above.

4) xsl:apply-templates

The XSLT xsl:apply-templates instruction can also be thought to be performing repetition logic. This instruction is available in all versions of the XSLT language. Below is an XSLT 1.0 stylesheet, illustrating use of xsl:apply-templates instruction,

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">

    <xsl:output method="xml" indent="yes"/>
```

```
    <xsl:template match="root">
        <result>
            <xsl:apply-templates select="val"/>
        </result>
     </xsl:template>

    <xsl:template match="val">
       <v><xsl:value-of select="."/></v>
    </xsl:template>

</xsl:stylesheet>
```

The above XSLT stylesheet, when applied to an XML input document mentioned in the earlier examples, produces following output,

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
   <v>1</v>
   <v>2</v>
   <v>3</v>
   <v>0</v>
   <v>0</v>
   <v>4</v>
</result>
```

The XSLT stylesheet mentioned in example above, loops through (via xsl:apply-templates instruction) the sibling “val” elements and applies/invokes the same template rule for each individual “val” element. The invoked template rule in this example, transforms the “val” input element to another element “v”.

In the general case, xsl:apply-templates instruction may not exhibit repetition behavior. Following is an example XSLT stylesheet depicting this, and applied to the same XML input document,

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">

     <xsl:output method="xml" indent="yes"/>

     <xsl:template match="root">
```

```
        <result>
          <xsl:apply-templates select="val"/>
        </result>
    </xsl:template>

    <xsl:template match="val[position() mod 2 = 0]">
        <v2><xsl:value-of select="."/></v2>
    </xsl:template>

    <xsl:template match="val[position() mod 3 = 0]">
        <v3><xsl:value-of select="."/></v3>
    </xsl:template>

    <xsl:template match="val">
        <v><xsl:value-of select="."/></v>
    </xsl:template>

</xsl:stylesheet>
```

The above XSLT stylesheet, loops through (via xsl:apply-templates instruction) each “val” input element and invokes a specific template rule based on position of input “val” element in an input XML document.

The above XSLT stylesheet when run, produces following output when applied to an XML input document,

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
    <v>1</v>
    <v2>2</v2>
    <v3>3</v3>
    <v2>0</v2>
    <v>0</v>
    <v3>4</v3>
</result>
```

In the above mentioned example, for evenly positioned input “val” elements, elements named “v2” are generated. For input “val” elements whose positions are divisible by 3, elements named “v3” are generated. For all other input “val” elements, elements named “v” are generated.

5) fold-left() and fold-right()

The fold-left() and fold-right() functions have been introduced in XPath 3.0, and exhibit repetition / iterative behavior. In an XSLT environment, it'll be required to use XSLT 3.0 to make use of these functions. These functions support a concept known as higher-order functions. By definition, higher-order functions can accept functions (known as function items in XSLT 3.0) as input arguments or may return functions (i.e function items) as result. Lets discuss both of these functions.

5.1 fold-left()

According to XPath 3.1 Functions & Operators specification,
The fold-left function, "processes the supplied sequence from left to right, applying the supplied function repeatedly to each item in turn, together with an accumulated result value."

Following is an example, of the use of fold-left() function in an XSLT stylesheet,

Input XML document:

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
   <val>1</val>
   <val>2</val>
   <val>3</val>
   <val>4</val>
</root>
```

XSLT 3.0 stylesheet:

```
<?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"
                exclude-result-prefixes="xs"
                version="3.0">

    <xsl:output method="xml" indent="yes"/>

    <xsl:template match="root">
        <xsl:variable name="inpValues" select="val" as="xs:string*"/>
        <result>
            <xsl:sequence select="fold-left($inpValues, '', function($accResult as xs:string, $nextItem as xs:string) as xs:string {concat($accResult, if ($accResult != '') then '.' else '', $nextItem)})"/>
```

```
    </result>
  </xsl:template>

</xsl:stylesheet>
```

The above XSLT transformation on the given XML input document, produces following result,

```
<?xml version="1.0" encoding="UTF-8"?>
<result>1.2.3.4</result>
```

Interestingly, instead of using the fold-left() higher-order function the logic of mentioned XSLT stylesheet in this example can also be achieved with the following expression,

```
string-join($inpValues, '.')
```

The 'string-join' function was first introduced in XPath 2.0 language.

Note that, the intent of using the fold-left() function in above mentioned example, is to illustrate the mechanics of the function and not to produce a best possible solution for the use case.

A good simple use case for fold-left() function can be to accumulate a running total, i.e. turn (1,2,3,4) into (1,3,6,10).

Following are few possible solutions to this, using fold-left() function:

```
<xsl:variable name="inpValues" select="(1,2,3,4)" as="xs:integer*"/>

fold-left($inpValues, (), function($a, $b) {$a, ($a[last()], 0)[1] + $b})

fold-left($inpValues, (), function($a, $b) {$a, if (empty($a)) then $b else $b + $a[last()]})

fold-left($inpValues, (), function($a, $b) {$a, sum(for $idx in 1 to count($a) return $inpValues[$idx]) + $b})
```

5.2 fold-right()

According to XPath 3.1 Functions & Operators specification,
The fold-right function, "processes the supplied sequence from right to left, applying the supplied function repeatedly to each item in turn, together with an accumulated result value."

Following is an example, of the use of fold-right() function in an XSLT stylesheet,

With the same XML input document as for the above example, the following XSLT 3.0 stylesheet is used,

```
<?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"
                exclude-result-prefixes="xs"
                version="3.0">

    <xsl:output method="xml" indent="yes"/>

    <xsl:template match="root">
        <xsl:variable name="inpValues" select="val" as="xs:string*"/>
        <result>
            <xsl:sequence select="fold-right($inpValues, '', function($nextItem as xs:string, $accResult as xs:string) as xs:string {concat($accResult, if ($accResult != '') then '.' else '', $nextItem)})"/>
        </result>
    </xsl:template>

</xsl:stylesheet>
```

The above XSLT transformation on the given XML input document, produces following result,

```
<?xml version="1.0" encoding="UTF-8"?>
<result>4.3.2.1</result>
```

Interestingly, instead of using the fold-right() higher-order function the logic of mentioned XSLT stylesheet in this example can also be achieved with the following expression,

```
string-join(reverse($inpValues), '.')
```

Along with ‘string-join’ function, the ‘reverse’ function was also first introduced in XPath 2.0 language.

Note that, the intent of using the fold-right() function in above mentioned example, is to illustrate the mechanics of the function and not to produce a best possible solution for the use case.

Its also worth noting that, fold-left() and fold-right() functions don’t have break option as the xsl:iterate construct.

Notes: Most of the XSLT 3.0 examples in this document, were tested with Saxon XSLT 3.0 HE processor. The fold-left() and fold-right() functions were tested with Saxon XSLT 3.0 PE/EE processor (provided with <oXygen/> XML Editor). All the XSLT 1.0 examples in this document, were tested with Xalan-J 2.7.2 processor.

References:

- https://www.w3.org/TR/xslt-30 : W3C XSLT 3.0 Recommendation
- http://www.saxonica.com : XSLT and XQuery processors
- https://www.oxygenxml.com : <oXygen/> XML Editor
- http://xalan.apache.org : An XSLT 1.0 processor

The following persons have contributed, to the subject matter of this document:

Michael Kay

Geert Bormans

Martin Honnen

Liam Quin

Dimitre Novatchev

John Lumley

Wendell Piez