Introduction to XSLT 3.0

While many W3C specifications take years to reach the recommendation state, XSLT has evolved quickly and deterministically, thanks not in small part to the great talent and sobriety of its spec. chair and a dedicated board committee.

The Stylus Studio team decided to be on the cutting edge, introducing support for the current XSLT 3.0 working draft in version X14 in order to give a chance to the community to start developing using the new language edition.

A variety of exciting new features have been introduced to make the language modern and to allow implementers to take advantage of modern hardware for transforming large data sets.

Support for Streaming

The need to process XML in streaming fashion, in other words, without loading the entire input document in memory, has risen over the years. Several use cases require processing very large streams of XML events, for example stocking tickers or social media user's stream.

Here ishow the specification formally defines streaming:

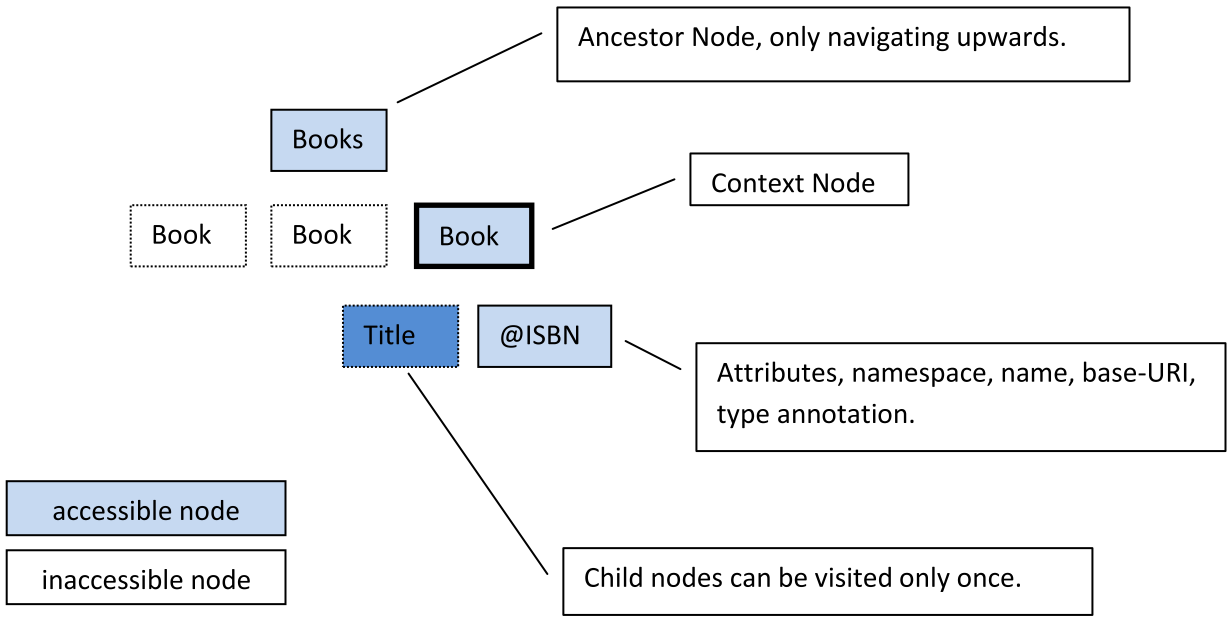
<<" A processor that claims conformance with the streaming option offers a guarantee that ... an algorithm will be adopted ... allowing documents to be processed that are orders-of-magnitude larger than the physical memory available.">>

In 2007, a team of XML experts came up with a dedicated language called STX, Streaming Transformation for XML, to tackle the problem. Even if the language did not gain significant popularity, it was a valuable exercise to identify use cases and come up with a declarative approach. Such experience has been an important inspiration for introducing the streaming feature in XSLT 3.0.

XSLT 3.0 introduces new constructs (*xsl:stream, xsl:mode streamable="yes"*) to explicitly indicate to stream the execution of its instruction body. Under streaming mode, there are a number of restrictions to be aware of:

- You have access only to the current element attributes and namespace declaration.
- Sibling nodes and ancestor sibling are not reachable.
- You can visit child nodes only once.

The following diagram illustrates which nodes are accessible while processing an xml document that contains a list of books.


Ancestor Node, only navigating upwards.
Books
Context Node
Book
Book
Book
Title
@ISBN
Attributes, namespace, name, base-URI, type annotation.
accessible node
inaccessible node
Child nodes can be visited only once.

Here is an example of how to split a very large document into small fragments:

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

   <xsl:template match="/">
      <xsl:stream href="books.xml">
         <xsl:iterate select="/books/book">
            <xsl:result-document href="{concat('book', position(),'.xml')}">
               <xsl:copy-of select="."/>
            </xsl:result-document>
            <xsl:next-iteration/>
         </xsl:iterate>
      </xsl:stream>
   </xsl:template>
</xsl:stylesheet>
```

Also of interest is the new instruction xsl:fork which declares that an XSLT block can be executed independently, during a single pass of a streamed input document.

Unfortunately, Saxon does not implement declarative streaming at the time of this writing.

Higher-Order Functions

Higher order functions are functions that either take functions as parameters or return a function.

XPath 3.0 introduces the ability to define anonymous functions and the XDM has been extended with the function item type. Such changes open the door to meta-programming using lambda expressions.

Let us start with an example: here is a lambda expression that calculates the square of two numbers and sums them.

(x, y) ↦ x*x + y*y

Such expressions can be can be reworked into an equivalent function that accepts a single input, and as output returns *another* function, that in turn accepts a single input .

x ↦ (y ↦ x*x + y*y)

The variable f1 is assigned to an anonymous function that takes an integer and returns a function that takes an integer and returns an integer.

```
<?xml version='1.0'?>
<xsl:stylesheet
   version="3.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xsl:template match="/">
      <xsl:variable name="f1" select="
         function($x as xs:integer) as (function(xs:integer) as xs:integer){
               function ($y as xs:integer) as xs:integer{
                  $x*$x + $y * $y
               }
         }
      "/>
      <xsl:value-of select="$f1(2)(3)"/>
</xsl:template>
</xsl:stylesheet>
```

XPath 3.0 provides built-in support for common lambda patterns such as map, filter, fold-left, fold-right, map-pairs. Here is an example of folding that sums only positive numbers from a list:

```
<?xml version="1.0"?>
<xsl:stylesheet  version="3.0"xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
   <xsl:variable name="list" select="(10,-20,30,-40)"/>
   <xsl:template match="/">
      <xsl:variable name="f1" select="
      function($accumulator as item()*, $nextItem as item()) as item()*
      {
         if($nextItem &gt; 0) then
            $accumulator + $nextItem
         else
            $accumulator
      }"/>
      <xsl:value-of select="fold-left($f1, 0, $list)"/>
   </xsl:template>
</xsl:stylesheet>
```

Text Manipulations

The language designers had always considered text manipulation an important feature, starting from XSLT 1. Functions for formatting numbers, date and time played an important role in building html content and eventually were moved to XPath in order to be shared with XQuery. XPath 2.0 introduced a large number of functions for manipulating strings: tokenize, matches, replace, string-join, upper-case, and lower-case.

Version 3 introduces a variety of new built-in functions for manipulating text, which are very useful when dealing with CSV data such as unparsed-text-lines, unparsed-text-available.

The following example shows how to implement a simple CSV to XML converter:

```
<?xml version="1.0"?>
<xsl:stylesheet version="3.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      xmlns:xs="http://www.w3.org/2001/XMLSchema"
      xmlns:hd="urn:header">

   <xsl:param name="csv" select="'one.csv'"/>
   <xsl:param name="sep" select="','"/>
   <xsl:param name="rootElement" select="'root'"/>
   <xsl:param name="rowElement" select="'row'"/>
   <xsl:param name="firstRow" select="true()"/>

   <xsl:variable name="header" select="tokenize(unparsed-text-lines($csv)[1], $sep)"/>

   <xsl:function name="hd:header" as="xs:string">
      <xsl:param name="col"/>
      <xsl:choose>
         <xsl:when test="$firstRow">
            <xsl:value-of select="$header[$col]"/>
         </xsl:when>
         <xsl:otherwise>item</xsl:otherwise>
   </xsl:choose>
   </xsl:function>

   <xsl:template match="/">
      <xsl:element name="{$rootElement}">
         <xsl:for-each select="unparsed-text-lines($csv)[position() &gt; 1]">
            <xsl:element name="{$rowElement}">
               <xsl:for-each select="tokenize(., $sep)">
                  <xsl:variable name="pos" select="position()"/>
                  <xsl:element name="{hd:header($pos)}">
                     <xsl:value-of select="."/>
                  </xsl:element>
               </xsl:for-each>
            </xsl:element>
         </xsl:for-each>
      </xsl:element>
   </xsl:template>
</xsl:stylesheet>
```

When processing in input a file like the following:

```
make,model,year,mileage
BMW,R1150RS,2004,14274
Kawasaki,GPz1100,1996,60234
Ducati,ST2,1997,24000
Moto Guzzi,LeMans,2001,12393
BMW,R1150R,2002,17439
Ducati,Monster,2000,15682
Aprilia,Futura,2001,17320
```

Produces as output

```
<?xml version='1.0' ?>
<root>
  <row>
    <make>BMW</make>
    <model>R1150RS</model>
    <year>2004</year>
    <mileage>14274</mileage>
  </row>
...
</root>
```

Conclusions

As you can see, there are many changes to look forward to in the upcoming XSLT 3.0 version. The specification is still under discussion and has not been finalized. The Stylus Studio Team will follow this closely and will release intermediate builds to provide a reference implementation in order to be prepared when version 3.0 goes live.