Transforming JSON using XSLT 3.0

Michael Kay
Saxonica
<mike@saxonica.com>

Abstract

The XSLT 3.0 and XPath 3.1 specifications, now at Candidate Recommendation status, introduces capabilities for importing and exporting JSON data, either by converting it to XML, or by representing it natively using new data structures: maps and arrays. The purpose of this paper is to explore the usability of these facilities for tackling some practical transformation tasks.

Two representative transformation tasks are considered, and solutions for each are provided either by converting the JSON data to XML and transforming that in the traditional way, or by transforming the native representation of JSON as maps and arrays.

The exercise demonstrates that the absence of parent or ancestor axes in the native representation of JSON means that the transformation task needs to be approached in a very different way.

1. Introduction

JSON [2] has become a significant alternative to XML as a syntax for data interchange. The usually-cited reasons[1] include:

- JSON is simpler: the grammar is smaller. The extra complexity of XML might be justified for some applications, but there are many others for which it adds costs without adding benefits.
- JSON is a better fit to the data models of popular programming languages like Javascript, and this means that manipulating JSON in such languages is easier than manipulating XML.
- JSON is better supported for web applications (for example, for reasons that are hard to justify, JSON is not subject to the same security restrictions as XML for cross-site scripting).

However, some of the transformation tasks for which XSLT is routinely used (for example, hierarchic inversion) are difficult to achieve in general-purpose languages like JavaScript.

[1] I include here only the reasons that I consider to be credible. Many comments on the topic also claim that XML is more verbose or that its performance is worse, but this appears to be folklore rather than fact.

XSLT 3.0 [4] (together with XPath 3.1 [5]) provides capabilities for handling JSON data. These capabilities include:

Two new functions `json-to-xml()` and `xml-to-json()` to convert between JSON and XML. These perform lossless conversion. The `json-to-xml()` function delivers XML using a custom XML vocabulary designed for the purpose, and the `xml-to-json()` function requires the input XML to use this vocabulary, though this can of course be generated by transforming XML in a different vocabulary.

Two new data types are introduced: maps and arrays. These correspond to the "objects" and "arrays" of the JSON model. In fact they are generalizations of JSON objects and arrays: for example, the keys in map can be numbers or dates, whereas JSON only allows strings, and the corresponding values can be any data type (for example, a sequence of XML nodes), whereas JSON only allows objects, arrays, strings, numbers, or booleans.

A new function `parse-json()` is provided to convert from lexical JSON to the corresponding structure of maps and arrays. (There is also a convenience function `json-doc()` which does the same thing, but taking the input from a file rather than from a string.)

A new JSON serialization method is provided, allowing a structure of maps and arrays to be serialized as lexical JSON, for example by selecting suitable options on the `serialize()` function.

While XSLT 3.0 offers all these capabilities [2], it does not have any new features that are specifically designed to enable JSON transformations — that is, conversion of one JSON structure to another. This paper addresses the question: can such transformations be written in XSLT 3.0, and if so, what is the best way of expressing them?

Note that I'm not trying to suggest in this paper that XSLT should become the language of choice for transforming any kind of data whether or not there is any relationship to XML. But the web is a heterogeneous place, and any technology that fails to handle a diversity of data formats is by definition confined to a niche. XSLT 2.0 added significant capabilities to transform text (using regular expressions); the EXPath initiative has added function libraries to process binary data[1]; and the support for JSON in XSLT 3.0 continues this trend. XSLT will always be primarily a language for transforming XML, but to do this job well it needs to be capable of doing other things as well.

2. Two Transformation Use Cases

We'll look at two use cases to study this question, in the hope that these are representative of a wider range of transformation tasks.

The first is a simple "bulk update": given a JSON representation of a product catalogue, apply a price change to a selected subset of the products.

[2]Some of these features are optional, so not every XSLT 3.0 processor will provide them.

The second is a more complex structural transformation: a hierarchic inversion. We'll start with a dataset that shows a set of courses and lists the students taking each course, and transform this into a dataset showing a set of students with the courses that each student takes.

For each of these problems, we'll look first at how it can be tackled by converting the data to XML, transforming the XML, and then converting back to JSON. Then we'll examine whether the problem can be solved entirely within the JSON space, without conversion to XML: that is, by manipulating the native representation of the JSON data as maps and arrays. We'll find that this isn't so easy, but that the difficulties can be overcome.

3. Use Case 1: Bulk Update

Rather than invent our own example, we'll take this one from `json-schema.org`:

```
[
    {
        "id": 2,
        "name": "An ice sculpture",
        "price": 12.50,
        "tags": ["cold", "ice"],
        "dimensions": {
            "length": 7.0,
            "width": 12.0,
            "height": 9.5
        },
        "warehouseLocation": {
            "latitude": -78.75,
            "longitude": 20.4
        }
    },
    {
        "id": 3,
        "name": "A blue mouse",
        "price": 25.50,
        "dimensions": {
            "length": 3.1,
            "width": 1.0,
            "height": 1.0
        },
        "warehouseLocation": {
            "latitude": 54.4,
            "longitude": -32.7
        }
    }
]
```

The transformation we will tackle is: for all products having the tag "ice", increase the price by 10%, leaving all other data unchanged.

First we'll do this by converting the JSON to XML, then transforming the XML in the traditional XSLT way, and then converting back. If we convert the above JSON to XML using the `json-to-xml()` function in XSLT 3.0, the result (indented for readability) looks like this:

```
[
<?xml version="1.0" encoding="UTF-8"?>
<array xmlns="http://www.w3.org/2005/xpath-functions">
   <map>
      <number key="id">2</number>
      <string key="name">An ice sculpture</string>
      <number key="price">12.50</number>
      <array key="tags">
         <string>cold</string>
         <string>ice</string>
      </array>
      <map key="dimensions">
         <number key="length">7.0</number>
         <number key="width">12.0</number>
         <number key="height">9.5</number>
      </map>
      <map key="warehouseLocation">
         <number key="latitude">-78.75</number>
         <number key="longitude">20.4</number>
      </map>
   </map>
   <map>
      <number key="id">3</number>
      <string key="name">A blue mouse</string>
      <number key="price">25.50</number>
      <map key="dimensions">
         <number key="length">3.1</number>
         <number key="width">1.0</number>
         <number key="height">1.0</number>
      </map>
      <map key="warehouseLocation">
         <number key="latitude">54.4</number>
         <number key="longitude">-32.7</number>
      </map>
   </map>
</array>
```

And we can now achieve the transformation by converting the JSON to XML, transforming it, and then converting back:

```
<?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"
  version="3.0"
  xpath-default-namespace="http://www.w3.org/2005/xpath-functions">

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

  <xsl:param name="input"/>

  <xsl:output method="text"/>

  <xsl:template name="xsl:initial-template">
    <xsl:variable name="input-as-xml" select="json-to-xml(unparsed-
text($input))"/>
    <xsl:variable name="transformed-xml" as="document-node()">
      <xsl:apply-templates select="$input-as-xml"/>
    </xsl:variable>
    <xsl:value-of select="xml-to-json($transformed-xml)"/>
  </xsl:template>

  <xsl:template match="map[array[@key='tags']/string='ice']/▶
number[@key='price']/text()">
        <xsl:value-of select="xs:decimal(.)*1.1"/>
  </xsl:template>

</xsl:stylesheet>
```

Sure enough, when we apply the transformation, we get the required output (indented for clarity):

```
[
    {
        "id": 2,
        "name": "An ice sculpture",
        "price": 13.75,
        "tags": [
            "cold",
            "ice"
        ],
        "dimensions": {
            "length": 7,
            "width": 12,
            "height": 9.5
        },
        "warehouseLocation": {
            "latitude": -78.75,
```

```
            "longitude": 20.4
        }
    },
    {
        "id": 3,
        "name": "A blue mouse",
        "price": 25.5,
        "dimensions": {
            "length": 3.1,
            "width": 1,
            "height": 1
        },
        "warehouseLocation": {
            "latitude": 54.4,
            "longitude": -32.7
        }
    }
]
```

Now, the question arises, how would we do this transformation without converting the data to XML and back again?

Here we immediately see a difficulty. We can't use the same approach because in the map/array representation of JSON, there is no parent axis. In the XML-based transformation above, the semantics of the pattern `map[array[@key='tags']/string='ice']/number[@key='price']/text()` depend on matching a text node according to properties of its parent (a `<number>` element) and grandparent (a `<map>` element). In the map/array model, we can't match a string by its context in the same way, because a string does not have a parent or grandparent.

However, all is not lost. With a little help from a general-purpose helper stylesheet, we can write the transformation like this:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:jlib="http://▶
saxonica.com/ns/jsonlib"
  xmlns:map="http://www.w3.org/2005/xpath-functions/map"
  xmlns:array="http://www.w3.org/2005/xpath-functions/array" ▶
version="3.0">

  <xsl:param name="input"/>

  <xsl:output method="json"/>

  <xsl:import href="maps-and-arrays.xsl"/>

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

```
  <xsl:template name="xsl:initial-template">
    <xsl:apply-templates select="json-doc($input)"/>
  </xsl:template>

  <xsl:template match=".[. instance of map(*)][?tags = 'ice']">
    <xsl:map>
      <xsl:sequence select="map:for-each(.,
                function($k, $v){ map{$k : if ($k = 'price') then $v*1.1 ▶
else $v }})"/>
    </xsl:map>
  </xsl:template>
</xsl:stylesheet>
```

This relies on the helper stylesheet, `maps-and-arrays.xsl`, containing default processing for maps and arrays that performs the equivalent of the traditional "identity template" (called shallow-copy processing in XSLT 3.0): specifically, processing an array that isn't matched by a more specific template rule should create a new array whose contents are the result of applying templates to the members of the array; while processing a map should similarly create a new map whose entries are the result of applying templates to the entries in the existing map. Unfortunately the shallow-copy mode in XSLT 3.0 doesn't work this way; it has the effect of deep-copying maps and arrays.

For maps, we can write a shallow-copy template like this (it's not actually needed for this use case):

```
<xsl:template match=".[. instance of map(*)]" mode="#all">
  <xsl:choose>
    <xsl:when test="map:size(.) le 1">
      <xsl:sequence select="."/>
    </xsl:when>
    <xsl:otherwise>
      <xsl:map>
        <xsl:variable name="entries" as="map(*)*"
           select="map:for-each(., function($k : $v) { map:entry($k, ▶
$v) })"/>
        <xsl:apply-templates select="$entries" mode="#current"/>
      </xsl:map>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

This divides maps into two categories. Applying templates to a map with less than two entries returns the map unchanged. Applying templates to a larger map splits the map into a number of singleton maps, one for each entry, and applies

templates recursively to each of these singleton maps. In the absence of overriding template rules for any of these entries, the entire map is deep-copied.

To make it easier to write a template rule that matches a singleton map with a given key, we can define a library function:

```
<xsl:function name="jlib:is-map-entry" as="xs:boolean">
    <xsl:param name="map" as="item()"/>
    <xsl:param name="key" as="xs:anyAtomicType"/>
    <xsl:sequence select=". instance of map(*) and map:size(*) eq 1 and ▶
map:contains($key)"/>
</xsl:function>
```

An overriding template rule can then be written like this:

```
<xsl:template match=".[jlib:is-map-entry(., 'price')]">...</xsl:template>
```

Writing a shallow-copy template rule for arrays is a little bit trickier because of the absence of XSLT 3.0 instructions for creating arrays: we hit the problem of composability, where XPath constructs such as `array{}` cannot directly invoke XSLT instructions like `<xsl:apply-templates/>`; and we also hit the problem that the only way of iterating over a general array (one whose members can be arbitrary sequences) is to use the higher-order function `array:for-each()`.

One way to write it might be like this:

```
<xsl:template match=".[. instance of array(*)]">
  <xsl:sequence select="array:for-each(., jlib:apply-templates#1)"/>
</xsl:template>

<xsl:function name="jlib:apply-templates">
  <xsl:param name="input"/>
  <xsl:apply-templates select="$input"/>
</xsl:function>
```

But this has the disadvantage that tunnel parameters are not passed through a stylesheet function call; in addition, the current template rule and current mode are lost. We can get around these problems using this more complicated formulation, which uses head-tail recursion:

```
<xsl:template match=".[. instance of array(*)]" mode="#all">
    <xsl:choose>
      <xsl:when test="array:size(.) = 0">
        <xsl:sequence select="[]"/>
      </xsl:when>
      <xsl:otherwise>
        <xsl:variable name="head" as="item()*">
          <xsl:apply-templates select="array:head(.)" mode="#current"/>
        </xsl:variable>
        <xsl:variable name="tail" as="array(*)">
          <xsl:apply-templates select="array:tail(.)" mode="#current"/>
```

```
        </xsl:variable>
        <xsl:sequence select="array:join((array{$head}, $tail))"/>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:template>
```

The complexity here doesn't really matter greatly, because the code only needs to be written once.

Returning to our specific use case, of updating prices in a product catalog, the main limitation of our solution is that all the update logic is contained in a single template rule, which works for this case but might not work for more complex cases. The match pattern for the template rule matches a map that needs to be changed, and this matching can only consider the content of the map, not the context in which it appears. Moreover, the template body does all the work of creating a replacement map monolithically without further calls on `<xsl:apply-templates>`; it would be possible to make such calls, but the syntax doesn't make it easy.

4. Use Case 2: Hierarchic Inversion

In our second case, we'll look at a structural transformation: changing a JSON structure with information about the students enrolled for each course to its inverse, a structure with information about the courses for which each student is enrolled.

Here is the input dataset:

```
[{
    "faculty": "humanities",
    "courses": [
        {
            "course": "English",
            "students": [
                {
                    "first": "Mary",
                    "last": "Smith",
                    "email": "mary_smith@gmail.com"
                },
                {
                    "first": "Ann",
                    "last": "Jones",
                    "email": "ann_jones@gmail.com"
                }
            ]
        },
        {
            "course": "History",
```

```
            "students": [
                {
                    "first": "Ann",
                    "last": "Jones",
                    "email": "ann_jones@gmail.com"
                },
                {
                    "first": "John",
                    "last": "Taylor",
                    "email": "john_taylor@gmail.com"
                }
            ]
        }
    ]
},
{
    "faculty": "science",
    "courses": [
        {
            "course": "Physics",
            "students": [
                {
                    "first": "Anil",
                    "last": "Singh",
                    "email": "anil_singh@gmail.com"
                },
                {
                    "first": "Amisha",
                    "last": "Patel",
                    "email": "amisha_patel@gmail.com"
                }
            ]
        },
        {
            "course": "Chemistry",
            "students": [
                {
                    "first": "John",
                    "last": "Taylor",
                    "email": "john_taylor@gmail.com"
                },
                {
                    "first": "Anil",
                    "last": "Singh",
                    "email": "anil_singh@gmail.com"
                }
```

```
            ]
        }
    ]
}]
```

The goal is to produce a list of students, sorted by last name then first name, each containing a list of courses taken by that student, like this:

```
[
    {
        "email": "ann_jones@gmail.com",
        "courses": [
            "English",
            "History"
        ]
    },
    {
        "email": "amisha_patel@gmail.com",
        "courses": ["Physics"]
    },
    {
        "email": "anil_singh@gmail.com",
        "courses": [
            "Physics",
            "Chemistry"
        ]
    },
    {
        "email": "mary_smith@gmail.com",
        "courses": ["English"]
    },
    {
        "email": "john_taylor@gmail.com",
        "courses": [
            "History",
            "Chemistry"
        ]
    }
]
```

As before, a stylesheet can be written that does this by converting JSON to XML, transforming the XML, and then converting back. The XML representation of our input dataset looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<array xmlns="http://www.w3.org/2005/xpath-functions">
   <map>
      <string key="faculty">humanities</string>
```

```
   <array key="courses">
      <map>
         <string key="course">English</string>
         <array key="students">
            <map>
               <string key="first">Mary</string>
               <string key="last">Smith</string>
               <string key="email">mary_smith@gmail.com</string>
            </map>
            <map>
               <string key="first">Ann</string>
               <string key="last">Jones</string>
               <string key="email">ann_jones@gmail.com</string>
            </map>
         </array>
      </map>
      <map>
         <string key="course">History</string>
         <array key="students">
            <map>
               <string key="first">Ann</string>
               <string key="last">Jones</string>
               <string key="email">ann_jones@gmail.com</string>
            </map>
            <map>
               <string key="first">John</string>
               <string key="last">Taylor</string>
               <string key="email">john_taylor@gmail.com</string>
            </map>
         </array>
      </map>
   </array>
</map>
<map>
   <string key="faculty">science</string>
   <array key="courses">
      <map>
         <string key="course">Physics</string>
         <array key="students">
            <map>
               <string key="first">Anil</string>
               <string key="last">Singh</string>
               <string key="email">anil_singh@gmail.com</string>
            </map>
            <map>
               <string key="first">Amisha</string>
```

```
                  <string key="last">Patel</string>
                  <string key="email">amisha_patel@gmail.com</string>
               </map>
            </array>
         </map>
         <map>
            <string key="course">Chemistry</string>
            <array key="students">
               <map>
                  <string key="first">John</string>
                  <string key="last">Taylor</string>
                  <string key="email">john_taylor@gmail.com</string>
               </map>
               <map>
                  <string key="first">John</string>
                  <string key="last">Taylor</string>
                  <string key="email">john_taylor@gmail.com</string>
               </map>
            </array>
         </map>
      </array>
   </map>
</array>
```

Here is the stylesheet:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  version="3.0"
  xmlns="http://www.w3.org/2005/xpath-functions"
  xpath-default-namespace="http://www.w3.org/2005/xpath-functions"
  expand-text="yes">

  <xsl:param name="input"/>

  <xsl:output method="text"/>

  <xsl:template name="xsl:initial-template">
    <xsl:variable name="input-as-xml" select="json-to-xml(unparsed-
text($input))"/>
    <xsl:variable name="transformed-xml" as="element(array)">
      <array>
      <xsl:for-each-group select="$input-as-xml//string[@key='email']" ▶
group-by=".">
        <xsl:sort select="../string[@key='last']"/>
        <xsl:sort select="../string[@key='first']"/>
        <map>
```

```
          <string key="email">{current-grouping-key()}</string>
          <array key="courses">
            <xsl:for-each select="current-group()">
              <string>{../../../*[@key='course']}</string>
            </xsl:for-each>
          </array>
        </map>
      </xsl:for-each-group>
      </array>
    </xsl:variable>
    <xsl:value-of select="xml-to-json($transformed-xml)"/>
  </xsl:template>

</xsl:stylesheet>
```

Is it possible to write this as a transformation on the maps-and-arrays representation of JSON, without converting first to XML? The challenge is again that we can't use the parent axis to find the course associated with each student. Instead, the approach we will use is to flatten the data into a simple sequence of tuples containing the values that we need (last name, first name, email, and course), and then use XSLT grouping on this sequence of tuples. We'll represent the intermediate form as a sequence of maps.

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  version="3.0"
  xmlns="http://www.w3.org/2005/xpath-functions"
  xpath-default-namespace="http://www.w3.org/2005/xpath-functions"
  expand-text="yes">

  <xsl:param name="input"/>

  <xsl:output method="json"/>

  <xsl:template name="xsl:initial-template">
    <xsl:variable name="input-as-array" select="json-doc($input)" ▶
as="array(*)"/>
    <xsl:variable name="flattened" as="map(*)*">
      <xsl:for-each select="$input-as-array?*?courses?*">
        <xsl:variable name="course" select="?course"/>
        <xsl:for-each select="?students?*">
          <xsl:map>
            <xsl:map-entry key="'course'" select="$course"/>
            <xsl:map-entry key="'last'" select="?last"/>
            <xsl:map-entry key="'first'" select="?first"/>
            <xsl:map-entry key="'email'" select="?email"/>
          </xsl:map>
```

```
        </xsl:for-each>
      </xsl:for-each>
    </xsl:variable>
    <xsl:variable name="groups" as="map(*)*">
      <xsl:for-each-group select="$flattened" group-by="?email">
        <xsl:sort select="?last"/>
        <xsl:sort select="?first"/>
        <xsl:map>
          <xsl:map-entry key="'email'" select="current-grouping-key()"/>
          <xsl:map-entry key="'courses'" select="array{ current-group()?
course }"/>
        </xsl:map>
      </xsl:for-each-group>
    </xsl:variable>
    <xsl:sequence select="array{$groups}"/>
  </xsl:template>

</xsl:stylesheet>
```

Interestingly, this technique of flattening the data into a sequence of maps (turning it into first normal form) and then rebuilding a hierarchy using XSLT grouping is probably a very general one; it could equally have been used for our first use case.

5. On the Question of Parent Pointers

I'm not sure if it was ever a conscious decision that XML structures should be navigable in all directions (in particular, in the parent/ancestor direction), while JSON structures should only be navigable downwards. It's not only the XDM model (used by XSLT and XPath) that makes this choice; the same divergence of approach applies equally when processing XML or JSON in Javascript. Both XML and JSON are specified primarily in terms of the lexical grammar rather than the tree data model, and it's not obvious from looking at the two grammars why this difference in the tree models should arise.

The ability to navigate upwards (and to a lesser extent, sideways, to preceding and following siblings) clearly has advantages and disadvantages. Without upwards navigation, a transformation process that operates primarily as a recursive tree walk cannot discover the context of leaf nodes (for example, when processing a price, what product does it relate to?), so this information needs to be passed down in the form of parameters. However, the convenience of being able to determine the context of a node comes at a significant price. Most notably, the existence of owner pointers means that a subtree cannot be shared: it is difficult to implement the `xsl:copy-of` instruction without making a physical copy of the affected subtree. This means that each phase of a transformation typically incurs cost proportional to document size. It is difficult to implement iterative transfor-

mations, consisting of small incremental changes to localized parts of the tree. This difficulty was reported a while ago [3] in a project that attempted to use the XSLT rules engine to perform optimization on the XSLT abstract syntax tree; the high performance cost of making small changes to the tree made this infeasible in practice.

The ability to navigate freely in the tree also seems to imply a need to maintain a concept of node identity (whereby two nodes that are independently created differ in identity even if they are otherwise indistinguishable). Node identity also comes at a considerable price, in particular by imbuing the language semantics with subtle side-effects: calling the same function twice with the same arguments does not produce the same result.

The model that has been adopted for JSON, with no node identity and no parent navigation, makes certain kinds of transformation more difficult to express, but it may also make other kinds of transformation (especially the kind alluded to, involving many incremental and localized changes to the tree structure) much more feasible.

6. Conclusions

From these two use cases, we seem to be able to draw the following tentative conclusions:

- Transformation of JSON structures is possible in XSLT 3.0 either by first converting to XML trees, then transforming the XML trees in the traditional way, then transforming back to JSON; or by directly manipulating the maps-and-arrays representation of JSON in the XDM 3.0 data model.
- When transforming the maps-and-arrays representation, the use of traditional rule-based recursive-descent pattern matching is inhibited by the fact that no parent or ancestor axis is available. This problem can be circumvented by first flattening the data – moving data from upper nodes in the hierarchy so that it is held redundantly in leaf nodes.
- The absence of built-in shallow-copy templates for maps and arrays is an irritation, but is not a real problem because these only need to be written once and can be imported from a standard stylesheet module.
- The lack of an instruction, analogous to `<xsl:map>`, for constructing arrays at the XSLT level is a further inconvenience; it means that data constructed at the XSLT level has to be captured in a variable so that the XPath array constructors can be used to create the array.
- Similarly, it would be useful to be able to invoke `<xsl:apply-templates>` as a function, to allow its use within the function supplied to `map:for-each()` or `array:for-each()` – preferably without losing tunnel parameters.

References

[1] Binary Module 1.0 EXPath Module, 3 December 2013. http://expath.org/spec/binary

[2] Introducing JSON http://json.org

[3] Writing an XSLT Optimizer in XSLT Proc. Extreme Markup Languages, Montreal, 2007. Available at http://conferences.idealliance.org/extreme/html/2007/Kay01/EML2007Kay01.html and with improved rendition at http://www.saxonica.com/papers/Extreme2007/EML2007Kay01.html

[4] XSL Transformations (XSLT) Version 3.0. W3C Candidate Recommendation, 19 November 2015. Ed. Michael Kay. http://www.w3.org/TR/xslt-30

[5] XML Path Language (XPath) 3.1. W3C Candidate Recommendation, 17 December 2015. Ed. Jonathan Robie, Michael Dyck, and Josh Spiegel. http://www.w3.org/TR/xpath-31