XSLT Three
Clearer Faster Wider Stronger

Liam Quin
Delightful Computing

New In XSLT 3

- New data structures & types
- Dynamic First-Class Functions
- More than XML: text, HTML 5, JSON
- New XSLT instructions
- More succinct syntax (shorter, often clearer)
- And...

More Highlights

- Streaming: Making the impossible possible
- Packages, and load dynamic XSLT or XQuery
- Many restrictions relaxed (shadow attributes, non-node steps more)
- Try/Catch for greater robustness
- Very up-to-date, much goodness.

Before We Start

- There's new features in XSLT that lend themselves to a new style of writing stylesheets; it can be less like text processing and more like mathematics.
- When you use the new features, be aware of who will read and maintain the stylesheets. It might be you, a year or a decade from now.
- I call this the *rhetorical nature of XSLT*.

XSLT 3 Overview

- Builds on XSLT 2 with *xsl:sequence* and types;
- Adds streaming, packaging, new data types, new ways of working, new ways to combine stylesheets;
- XPath got terser (both good and bad)
- Let's start with the best of all: expand-text

XSLT 1.0 Message

```
<xsl:message>
    <xsl:text>Darlings, I lost </xsl:text>
    <xsl:value-of select="count($s1) – count(item/found)" />
    <xsl:text> </xsl:text>
    <xsl:value-of select="$garment-plural" />
    <xsl:text>.</xsl:text>
</xsl:message>
```

Now With XSLT 3

```
<xsl:message>Darlings, I lost {
    count($s1) – count(item/found)
} {$garment-plural}.</xsl:message>
```

Darlings, I lost 49 pairs of socks.

Element Example

```
<xsl:template match="anné">
   <year>{ . }</year>
</xsl:template>
```

- {Computed text values} always make text nodes.

Turn it on

- Add the attribute expand-text="yes" to any XSLT element (including xsl:stylesheet);
- Turn it off with expand-text="no" for a particular element and its children (e.g. one template);
- Use xsl:expand-text on a direct element constructor or extension element.

Relaxing Restrictions

- XSLT 3 is more orthogonal - e.g. more instructions can have *select* attributes, and you can use self::foo in match patterns;
- Places where constant strings couldn't be made into expressions (for not breaking styesheets) can now all take *shadow attributes* computed at compile time:

Shadow Attributes

- Put an underscore (_) before an attribute name and it becomes an *attribute value template* evaluated at compile time, supplying the actual attribute value.
- Any parameters or variables referred to must be declared with static="yes"
- Can use this e.g. to parameterize xsl:output doctype.

A New XPath Operator

- XSLT 3 introduces =>

  "David" => upper-case() => string-to-codepoints() => reverse() => codepoints-to-string()

  Same as

  codepoints-to-string(reverse(string-to-codepoints(upper-case("David"))))

- Easier to read, for people who remember what => does.

Don't overdo it

- `$input => upper-case() => string-to-codepoints() => reverse() => codepoints-to-string()`
- Compare:

```
upper-case(my:string-reverse($input))
```

- This is about naming abstractions and making them explicit.

The ! operator

- string-to-codepoints("David") ! count(.) *produces:*

  (1 1 1)

- string-to-codepoints("David") => count() *produces:*

  5

- This shows, ! works on each item in turn, like [ ], and => works on the entire value at a time.

New expression: for

- for $i in (1 to 30) return $i * $i
- for $a in /nuts, $b in (‘flour’, ‘surprise’)

  return $a || ‘ ‘ || $b
  - Hazelnut flour, Hazelnut surprise, Almond flour…
- if (//weather/snow) then “boots” else “barefoot”
  - This was also in XSLT 2

New Structures: Maps

- A *map* is an extensional function (mathematics) that says how you get from one set of values to another by explicitly listing all possibilities:

  1 ⇒ 1, 2 ⇒ 4, 3 ⇒ 9, 4 ⇒ 16, 5 ⇒ 25, 6 ⇒ 36

- The *keys* and *values* can be anything:

  "Toronto", ("416,", "905")　　　　"DC", "202"

- Maps are light-weight compared to element nodes.

Maps in XPath

- Create:

```
map { “name” : “Boris”, “is-greedy” : true(),
 “socks” : map { “left” : “black”, ”right” : “grey” }
}
```

- Type

```
map(xs:string, xs:integer)
```

use * to match any type, e.g. map(*)

Making a map in XSLT:

```
<xsl:variable name="Institutions" as="map(*)">
    <xsl:map>
        <xsl:map-entry key="BSI"
                select=" 'Bavarian Sock Inspector' " />
        <xsl:map-entry key="MARC"
                select="Make Archivists Retch and Cry" />
    </xsl:map>
</xsl:variable>
```

Getting Stuff Out of a Map

- map:get(*key*)
- $mymap?simplekey   note, no quotes
- $mymap(*key*)(*subkey*)   for nested maps
- $mymap?(*key*, *key*…)   for any keys
- $mymap?("key1")?("submapkey")?foo
- $mymap?*[?submapkey = "value"]?foo

New Data Type: Arrays

- Arrays are like sequences, except they do not get flattened automatically …

count((1, 2, 3)) ⇒ 3 *but* count([1, 2, 3]) ⇒ 1

array:size([1, 2, ['Ringo', 'Paul', 'John', 'George'], 2])
    ⇒ 4

JSON Example

```
“config” : {
    “users” : [
        { “name” : “Julia”, “wearsShoes” : “yes” },
        { “name” : “Tom” },
    ],
    “modules” : [ . . . .
```

Arrays, Maps, JSON

- You can load a JSON file with json-doc() and get back a mix of arrays and maps.
- You can use json-to-xml() to get an XML representation, but only if the XML was made with xml-to-json() or uses the same schema.
- These functions take a map with options…

JSON functions

- parse-json($string, $map)
- json-doc($href, $map)
  - Like unparsed-text($href) => parse-json($map)
- json-to-xml()
- xml-to-json()
  - Requires the use of the W3C/XSLT JSON XML schema.

Exploring

```
<fn:map xmlns:fn="http://www.w3.org/2005/xpath-functions">
  <fn:string key="test" escaped="false">foo/bar</fn:string>
</fn:map>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="3.0">
  <xsl:output method="text"/>
  <xsl:template match="/*">
    <xsl:value-of select="xml-to-json(.) => parse-json() =>
                 serialize(  map {'method': 'json'} ) "/>
  </xsl:template>
 </xsl:stylesheet>
```

https://xsltfiddle.liberty-development.net/bwdwrV/2

New Functions

- Streaming (≋) Functions
- Functions on maps and arrays
- Functions on Functions: apply(), fold-left() etc
- Collations, sorting;
- System: serialization, environment variables, etc
- Numeric (random numbers!) and other.

EXPath Extension Functions

- The functions in EXPath are *really* useful, e.g.
  - Read and write files
  - Process *binary* files
  - Read and write Zip archives (e.g. for epub files)
- They are *native*, not written in XSLT or XQuery
- Supported by BaseX and Saxon and others:
- https://expath.org/ …

EXPath Modules

- File http://expath.org/spec/file
- Binary http://expath.org/spec/binary
- Archive: http://expath.org/spec/archive
- Newer versions of some of them: https://www.w3.org/community/expath/

try/catch

- Use xsl:try to evaluate expressions that might raise errors, and take special action based on the errors.
  - E.g.: try casting an attribute to a dateTime or to an integer (better: use *castable as* or *instance of*);
  - Open a file that might not be well-formed XML, without exiting on the error
- Not a way to cover up programming errors!

New: xsl:iterate

- Like xsl:for-each, with a required *select* attribute;
- You can use xsl:break to end iteration;
- Call :xsl:next-iteration, possibly with new parameters, at any point, but only
  - As the last instrution in an *if* or *iterate* body, or of a *when* or *otherwise* or *try* or *catch*

Higher Order Functions

Inline Function Expressions

```
let $f := function($e as element(sup)) as element(*)? {
   if ($e/sub) then $e/sub/node() else $e/node()
} return $f(//reference)
```

- Use functions in expressions, in *select* attributes, sequences, etc.
- It’s *usually* better to use xsl:function, but this way you can share XPath expressions with XQuery too.

Function parts

```
let $f := function( parameters ) as type {
    function body goes here
}

$f(//reference)
```

The ? place-holder

- Use ? to mark the arguments that you have not supplied yet:

```
let $slashify := string-join(?, "/")
return $slashify( ("a", "b", "c") )
```

- You can use this new feature with => too
- You could use $slashify with sort().

Working With HTML

- Still no direct standard support for reading HTML
- You can write HTML 5 with xsl:output
- You can make an HTML 5 string with serialize()
- There are some new functions that make life a little easier.

The HTML Collation

- In HTML, ASCII characters are case insensitive and others are not:

  XRef eq xref

  XRÉF **ne** xréf

- XSLT 3 introduces this as the HTML collation.

  contains-token(@class, 'to-ref')

  contains-token(@class, $token, $collation)

Other new features for Web work

- Use parse-ietf-date() to convert an IETF-style timestamp date (Wed Nov 6 13:58:49 EST 2019) into a dateTime object;
- These dates are found in HTTP headers, email headers and so forth;
- Use expand-text="no" for embedded JavaScript and CSS, so { } are not special. ...

Web features continued

- New function get-environment-variable() helpful with the CGI interface in some environments;
- Can now process text documents a line at a time with unparsed-text-lines()
- “http://www.w3.org/1999/xhtml”body syntax (EQNames) and *:body

Matching Any Type

- You can match any sort of item now, not just nodes;
- A template that matches integers? For-each that iterates over a sequence of tokens from @class?
- Combine with Schema Typing and have templates matching e.g. element(*, my:explainer)
- Watch that there's not always a useful context item

Stronger type-checking

- Declare the required type of the context item in a template with xsl:context-item, to get errors if a template is called unexpectedly;
- All built-in XSD types available, along with schema-less lax validation
- Use *as* attributes widely and find problems sooner
- xsl:message terminate="yes"

Reminder; New expressions

- Map constructors map { .... }
- Array constructors [...]
- Named function references and inline function expressions dynamic function call
- for $town in (.....) return ....
- Reminder: XPath 2 already had if (a) then b else c

The most powerful new function

fn:transform()

What is fn:transform()?

- A function in XPath that calls XSLT, runs a transformation, and returns the result.
- So you can write, for example,

```
<xsl:sequence select="fn:transform(...., .)" />
```

Some uses

- Processing lots of files (e.g. test suite) without restarting Java on each one;
- XProc-like pipelines;
- Simplifying stylesheets by replacing modes;
- Replacing ant or other build systems.

Streaming

- A *non*-streaming processor reads its input and then processes it.
- A *streaming* processor reads input as it arrives, e.g. over a network or from disk, and processes it as it becomes available.

Going Further

- xsl:stream
- xsl:source-document and xsl:iterate
- xsl:where-populated, on-empty, on-non-empty

xsl:where-populated

Wrapper appears only if it is not empty:

```
<xsl:where-populated>
  <fn-wrap>
    <xsl:apply-templates select=”fn”/>
  </fn-wrap>
</xsl:where-populated>
```

Xsl:on-empty

- Triggered if nothing before it made anything

  Must be last in its sequence constructor.

Xsl:on-non-empty

- Only evaluated if a sibling made something.
- Does not have to be last.
- See https://www.w3.org/TR/xslt-30/#iteratewher for an example combining where-populated, on-empty and on-non-empty.
- Useful outside streaming too!

Packages

- Not yet widely used in public;
- Can be a way to help manage configurations and versions in a corporate/enterprise or large closed environment;
- Packages can be compiled separately & reused
- Packages located using inplementation-specific mechanism (e.g. conf file for Saxon)

xsl:use-package

See *https://stackoverflow.com/questions/57478467/xslt-3-how-to-write-a-package* for a worked example with Saxon and the Saxon configuration file.

stack overflow page

load-xquery-module()

- Although there's no fn:query() you can load an XQuery module; it appears as a map, and you can ask it for functions and call them.
- This depends on your XSLT implementation also supporting XQuery.
- Saxon does, but not with a database.

Packages and system dependencies

- You can control system dependencies by adding a *use-when* attribute to any XSLT element, or *xsl:use-when* to other elements.
- The *use-when* attribute value is a *static* expression. You can use *system-property* but not parameters.
- You can also use XSLT 3 “static variables” …

Summary: XSLT 3 Brings

- New readability features (*esp.* expand-text)
- New functions and operators
- Ability to call XSLT and XQuery with fn:transform and fn:load-query-module
- Streaming
- A more complete language

Thank you

Liam Quin, Delightful Computing

Milford, Ontario