First a tiny bit of background.

AS we know, the way to do “looping” in both XSLT and XQuery is through recursion. And if you know the common pattern for writing a recursive function or template, and follow it carefully, chances are high that the processor will turn your expression into a loop internally, using tail call optimization.

This is great, because at each stage of recursion you can explore a new input value, and computing the results happens only at that time, not all in advance, so e.g. if the calculation is really expensive and you only need a few values processed, but don’t in advance know how many, you can terminate early.

Example: Consider all of the nodes reachable in this RDF graph starting at a particular node, but only until you reach a particular result (e.g. can you get to Mornington Crescent from here?).

But if you don’t get the template or function right, you run out of stack space, or reach some limit on stack size or recursion depth imposed by the processor. This can be as low as 50 nested calls in some implementations.

So along came xsl:iterate. This XSLT instruction is the equivalent of a recursive template, but in a way that can always be optimized. It also supports early termination, which is tricky to get right with a recursive template.

There is nothing in xsl:iterate you couldn’t write yourself with a recursive named template and a lot of care. And yet, in a detailed and exhaustive survey of all XSLT 3 users in the entire universe [1] it was the second most popular feature introduced in 3.0.

[1] OK, I asked a few people.

But what about pure XPath, or what about XQuery?

There’s nothing similar built in. And as with recursive XSLT templates or functions, there is no guaranteed optimization of recursive functions.

So, on to generators…

I maintain an online version of a 32-volume dictionary of biography from 1812 or so. It covers over 9500 people, and often lists the most important books they wrote.

Many of the books are online in TEI XML, it turns out. The Early English Books Online project transcribed a lot of them (using slav… er I mean graduate students). Over sixty thousand books in all.

So I wrote a query (using BaseX) to match up authors of books in EEBO to people in the 1812 biographical dictionary. It found two or three thousand authors who were likely the same (you have to allow for spelling variations, and for dates being a year or two different partly because of calendar changes).

Now for each author I'd like to look up each book in turn in the British Library Short Title Catalogue (at least for the 18th Century English authors, but that's more than half of them I think), and be able to give visitors to the site better references with the more usual names of the books, and, ultimately, link to online images as well as the EEBO transcription.

So, I have 9,500 biographies in Chalmers, many with books they supposedly wrote, and I can look up the books they wrote in the catalogue. If I look up all of them at once the British Library will hate me, but in any case I don't need to. Some books will appear more than once (e.g. multiple authors, or a mention in a biography rather than an assertion or authorship). Many will fall outside the date period for the catalogue. And when I've found a likely match for one, I want to fetch the full catalogue record, save it in a cache for later processing, and move on.

Consider using xsl:iterate, but you are going to use a Web API on each iteration, and maybe you will stop when you have found a dozen matches for each author: the goal is partly to build confidence in the name of the author and their details, and fetching sixty works doesn't do that much better than fetching a dozen, if the details match well enough.

So, 9,600 times 12 is 115,200 queries at most.

Now, if you've worked with XSLT 3 streaming, you know that you have to avoid constructing long sequences of data from your input. Don't write

xsl:for-each select="/dictionary/letter-group/entry/title"

because that makes a list of 9,960 entry titles before it starts to process them. Instead, you write, e.g.

xsl:apply-templates select="/dictionary/letter-group/entry/title"

so that the processor can handle one title at a time.

In XQuery we don’t have template matching. We can only do the for-each. I can rewrite this as a recursive function, keeping an integer to track position in the list, or using ../following::entry[1]/title to fetch the next title given the current one.

This is easy to get wrong, and if I try to process the 60,000+ books in TEI XML this way I’m in real danger of running out of memory.

But what if I could write something like,

```
let $g := generator:new( {

 “current” : $doc/dictionary/letter-group[1]/entry[1]/title ,

 “next” : fn() {

   generator:new(

     “current” :  $g/../following::entry[1]/title.

    “next” : $g(“next”) (: copy the next function :)

} )

return process-books-by($g)
```

where process-books-by() is a function that processes $g→current() and then calls itself with $g→next() each time?

This is a lot less to write, and the generator class (OK, we don’t have classes, the generator code) is written once and (we hope) tested. (this example doesn’t use exactly the proposed generator API)

All that’s going on is that I have a map, an object if you prefer, except we don’t have objects, that knows how to find the next value, and returns the empty sequence when there are no more values. The *next* function returns a new generator with the current value set to the next one to process.

But this is exactly what an iterator looks like in procedural languages. The generator proposal defines the functional counterpart.

When I’ve found a list of candidates from the remote catalogue, I have a text similarity problem – which book is the best match (if any) from the list? Does it match any of the 60,000 EEBO books?

The thing about text similarity is you want to sort candidates based on a likelihood criterion, then do the more expensive similarity test on the smallest possible solution space, the fewest books possible, because it's computationally expensive. So when you find a good match you want to stop, but you don't know how many to process in advance, and the full list of EEBO books uses tens or hundreds of gigabytes in the database.

So this is another good case for generators: choose some candidates, process them until you find one you think is likely the right answer, then stop.

Again, you could do all this with a recursive function, or maybe with one of the new-fangled qt4 while-do-done() things. But you can't use fold-left as you can't construct the sequence to process, and even xsl:iterate is tricky if you don't know what you're iterating over exactly in advance.

This example does not use exactly the proposed generator API, because in a generator the current value can be the empty sequence, it doesn't have to mean 'there is no more'.

Some examples of good candidates for standardization might include:

1. Things many people need but have to write for themselves

2. Things people tend to write incorrectly, causing problems (e.g. contains-token() for matching HTML class values)

3. Things that are hard enough to write that most people will simply use a different language or wander off or fail or hate us :-) --- e.g. replace() could be written in terms of string processing functions, and the file:archive module for reading/writing zip files could be written with file:read-binary and file:write-binary and simply implementing Lempel-Ziv and Huffman compression in XPath, and then you could process epub files in XSLT or XQuery, but few people would do that. Having the extension makes things feasible.

4. Things that extend the reach of the language – this is social more than technical.

Generators meet all of these criteria, which is why I support the work.