Practical Perl Tools

Taking the XPath Less Traveled

DAVID N. BLANK-EDELMAN

David Blank-Edelman is the director of technology at the Northeastern University College of Computer and Information Science and the author of the O'Reilly book *Automating System Administration with Perl* (the second edition of the Otter book), available at purveyors of fine dead trees everywhere. He has spent the past 24+ years as a system/network administrator in large multi-platform environments, including Brandeis University, Cambridge Technology Group, and the MIT Media Laboratory. He was the program chair of the LISA '05 conference and one of the LISA '06 Invited Talks co-chairs. David is honored to have been the recipient of the 2009 SAGE Outstanding Achievement Award and to serve on the USENIX Board of Directors beginning in June of 2010.
dnb@ccs.neu.edu

In my last column I made a statement about liking XPath so much that I just might be tempted to make it the subject of the next column. I've decided to make good on that threat and talk a bit about XPath and how its power can be harnessed from Perl. One strange thing about this choice is it will force the column to include more details about another language that isn't Perl. Hopefully, you, faithful reader, are willing to hang with that. The plus is the language in question is XPath, something you'll be able to use with Perl or virtually any other language that is currently wetting your whistle.

What's XPath and Why Do I Care?

So what is XPath? I think you'd be a bit hard-pressed to figure out the answer to this question, and why XPath is so cool, if you just read the introduction to the W3C's technical report on the subject ("XML Path Language (XPath), Version 1.0"):

> *XPath is the result of an effort to provide a common syntax and semantics for functionality shared between XSL Transformations and XPointer. The primary purpose of XPath is to address parts of an XML document. In support of this primary purpose, it also provides basic facilities for manipulation of strings, numbers and booleans. XPath uses a compact, non-XML syntax to facilitate use of XPath within URIs and XML attribute values. XPath operates on the abstract, logical structure of an XML document, rather than its surface syntax. XPath gets its name from its use of a path notation as in URLs for navigating through the hierarchical structure of an XML document.*

Let me try to provide a different, pragmatic, and perhaps overly blunt explanation of what XPath is and why you might care, as seen from the porthole of a Perl programmer. XPath provides a concise, elegant syntax for referencing a specific thing or things in a document. Let's say you want your program to retrieve the text of the third headline in a document. Or maybe you need to extract the weather forecast part of an XML document (as we did in the last column). Both of these things are pretty easy to do with the XPath syntax, provided your document is marked up well. XPath was originally invented to interact with XML documents, but it works quite well for well-formed HTML documents as well.

To understand how XPath works, there's one very important idea you must wrap your head around: documents as trees. And just to be clear, when I say "trees" I don't mean the "I think that I shall never see. A poem lovely as" kind of tree. I'm

referring to the computer sciency, data structure/hierarchical organization like "directory tree" kind of tree. Going from an XML or HTML document to a tree structure may require a bit of squinting and pondering if you've never heard of this idea before, but let me see if a couple of examples might help. Take for instance the following XML document:

```
<poem>
  <poet>
    <name>Joyce Kilmer</name>
    <born>1886</born>
    <died>1918</died>
    <movement locale="American">modernist</movement>
  </poet>
  <title>Trees</title>
  <excerpt>I think that I shall never see
           A poem lovely as a tree.
  </excerpt>
</poem>
```

In XML-speak, the "root" element of this XML document is the <poem> element. And when I say "element" in this case I'm referring to the <poem> tag and everything inside it up until the closing </poem> tag. The <poem> element has three sub-elements, namely <poet>, <title>, and <excerpt>. Within <poet>, there are four sub-elements: <name>, <born>, <died>, and <movement>.

So far, so good, right? Okay, let me turn this document into a tree for you by massaging that last descriptive paragraph a bit: At the top of the tree (at the root node) is the <poem> element. It has three child nodes: <poet>, <title>, and <excerpt>. One of these nodes, <poet>, has four children of its own: <name>, <born>, <died>, and <movement>. Tah-dah. It's a tree.

How would this work with an HTML document? Same sleight-of-hand applies. Take for example this document:

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
     <title>Joyce Kilmer</title>
     <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
  </head>
  <body>
    <h1>Trees<h1>
    <p>I think that I shall never see...</p>
  </body>
</html>
```

I could start speaking in tree-language and say something like: the root node of the document is the <html> element. It has two children, <head> and <body>. The <head> child has two children of its own, <title> and <meta>. Similarly, the <body> node has an <h1> element and a paragraph element (<p>) as its children.

Once we're certain that we're dealing with data that is in a tree structure, then doing all of the things we talked about before having to do with referencing parts of the document or extracting certain bits all become tree operations. We need a way to take a walk up and down the tree structure, find certain elements in the tree, extract parts of the tree, and so on. And that's where XPath comes in.

Before we actually see any XPath syntax, I feel that I must confess that we're only going to stick with the simple, but highly useful, parts of XPath in this column. Let me unburden my soul by mentioning the syntax we'll be seeing will be all XPath 1.0 syntax, even though a 2.0, and a 2.0 schema-aware specification (both not nearly as widely implemented as 1.0) exist. XPath lets you specify things using an abbreviated or an unabbreviated form; we're going to use the former. XPath defines eight separate ways to specify the direction a parser should go when walking the tree (called an axis), but we're not going to use most of them in this column (and you may find you never use most of them in real life either). All of this is just to say that there's considerable depth to explore when dealing with XPath. Consider doing more reading on the subject if you decide XPath is a tool you'd like to use with proficiency.

XPath Path Time

Time to put the "Path" in XPath. For these examples let's use the sample XML document from above so we have something simple for demonstration purposes. As a first test, let's start by figuring out how to reference the <movement> element. In XPath we walk down the tree from the root node, just like you would if you were dealing with a file system path. The resulting XPath specification is:

```
/poem/poet/movement
```

Yup, it is that simple to reference the <movement> element. Note: this lets us point to that node in the tree. If we want the textual contents of that node (i.e., "modernist") we would add the text() function to our specification (as in "/poem/poet/movement/text()").

Let's follow the file system path analogy a little farther. Like most common file-system semantics, you can also use the . (dot) and .. (dot-dot) characters to refer to relative places in the tree. A dot by itself refers to the current node, while dot-dot refers to the parent node. Given our example document:

```
/poem/poet/death/..
```

refers to the <poet> element. That example may be a bit obtuse because it isn't clear why you'd want to go that deep in the tree before coming back up a level. We'll see when you might want to do this in a moment once we discuss a few other path-related items. The first is the use of an at (@) character in a path. The @ gets used to refer to an attribute of an element. So if we wanted to access the "locale" part of <movement locale="American"> we could write:

```
/poem/poet/movement/@locale
```

The last part of the path syntax I want to mention also has its roots in file system syntax. XPath lets you include wildcards in your expressions, so I could write:

```
/poem/poet/*
```

to refer to all of the child nodes of the <poet> element or even something like:

```
/poem/*/born
```

to reference all of the nodes that have a <born> sub-element. Or I could write:

```
/poem/poet/*/@locale
```

to find all of the elements under <poet> that have locale attributes.

Or even:

```
/poem/poet/movement/@*
```

to reference all of the attributes of the <movement> element.

This globbing-like syntax is similar to the way we use globbing with file systems. With a file system, if you say *.txt, you expect to get back a list of files that end in .txt. With XPath, when you use a wild card, your program gets back a list of nodes that match the specification. I'll be sure to show you an example of this in Perl so you can see how we might handle dealing with a list of nodes (foreshadow: the same way we deal with any other list in Perl).

There's one more very important path-related piece of syntax to show you that is kind of like globbing, only cooler. It doesn't have a direct file system analog, although I sometimes wish it did. If you use two slashes (//) in your specification, an XPath parser will start at that place in the tree and walk down the tree until it finds a match. Our tree isn't particular deep or complex, so examples of this may seem a bit weak. But when you start dealing with more complex documents, it can be awfully convenient to tell the parser "Go find this specification any place in your tree" without having to do that searching yourself. For our document, we could write something like:

```
//died
```

and it would walk down from the root until it found that node deep in the <poet> element. The double-slash notation also works in the middle of a specification, as in:

```
/poem//@locale
```

Remember the mention above about the "dot-dot" operator? Here's one place where it makes more sense. If we wanted to extract the textual contents of all of the nodes that contained a `@locale` attribute (vs. just referencing the locale attributes themselves as we just wrote), we could write:

```
/poem//@locale/../text()
```

An XPath parser would return the string "modernist" if asked to parse our current sample document with this specification.

XPath has a ton of other "navigational" filigrees that let you say things such as "move to the next sibling node in the tree," but that gets you into talking about the different XPath axes (plural of axis). Rather than bore you with any more of them, there's just one more concept we should discuss before we actually write some Perl code.

XPath Predicates and Functions

You may not have noticed, but something about our sample document has made our XPath specifications easier than they might ordinarily be. Every one of our elements has been unique at its level of the tree. For example, there's only one <poem> element. That seldom happens with real documents. Most documents contain more than one of a certain piece of information. Let's construct a new sample that incorporates our previous sample so we have something more interesting to work with. Imagine we had a new document with this in it:

```
<anthology>
  <poem>
   <poet>
       <name>Joyce Kilmer</name>
       <born>1886</born>
       <died>1918</died>
       <movement locale="American">modernist</movement>
    </poet>
    <title>Trees</title>
    <excerpt>I think that I shall never see
             A poem lovely as a tree.
   </excerpt>
  </poem>
  <poem>
   <poet>
       <name>Ogden Nash</name>
       <born>1902</born>
       <died>1971</died>
       <movement locale="American">light verse</movement>
    </poet>
    <title>Song of the Open Road</title>
    <excerpt>I think that I shall never see
             A billboard lovely as a tree.
   </excerpt>
  </poem>
</anthology>
```

Given this document, how do we walk down the tree? We can start off with "/anthology", but then how do we tell the parse which <poem> element we want to reference? This is where XPath predicates come into play. XPath predicates are specified using square brackets right at the decision point in the path. XPath has a whole bunch of predicates, the simplest of which is simply to use the number of the branch, starting at the number 1 (yup, it is 1-based not 0-based as in Perl):

```
/anthology/poem[2]/poet/name/text()  # Ogden Nash
```

You can also use string comparison predicates against attributes and text contents of nodes. If we wanted to find the names of all of the poets who died in 1971, we could write something like this:

```
//died[text()="1971"]/../name/text() # Ogden Nash
```

This example uses a combination of the different things we've covered up until this point, so let me go over it once just so it is clear. The double slash at the beginning says "walk down the tree until you find a match for what follows." In this case, it walks down until it finds a node called <died> whose contents equal 1971. When it finds a node that matches this condition, it walks back up one level to find a <name> element and returns the textual contents of that <name> element. Given our document, this will return "Ogden Nash."

If we wanted to return the names of all of the American poets, we could write:

```
//movement[@locale="American"]/../name/text() # Joyce Kilmer & Ogden Nash
```

and we would get back two strings: "Joyce Kilmer" and "Ogden Nash."

XPath also has a bunch of functions you can use. For example, if we wanted to know how many American poets were in the document, we could instead write:

```
count(//movement[@locale="American"]) # 2
```

So far I've just been describing the XPath syntax without providing much commentary, so let me be sure my bias is clear. I *love* this language. It may just be that my brain is wired strangely, but I find the path analogy lets you write concise and elegant specifications for document references and document extraction queries.

Forget Anything?

Given that this is a Perl column, I'm pretty sure I'd be remiss if I didn't include any Perl code. Let me warn you ahead of time, the Perl samples we're about to see won't themselves be anything exciting. This is not because Perl isn't exciting (hey, hey, quiet down, peanut gallery), but it is because all of the magic super powers reside in the XPath language itself. Perl programs that use XPath largely get to say, "Here's a document. Here's an XPath specification. Go to town and hand me back the results when you are done."

There are a number of different Perl modules that understand XPath or a reasonable subset of it (in fact, in the last column, I mentioned Class::XPath, which lets you graft on an XPath-lite interface to an object tree of your own making). Even though there is actually an XML::XPath module (last touched in 2003), the two modules I use for XML and HTML XPath parsing are XML::LibXML and XML::Twig. I've also used HTML::TreeBuilder::XPath for my HTML parsing. For our big anti-climactic use of XPath in Perl, we'll use XML::LibXML. Here's some code that returns all of the American poet names from our document:

```
use XML::LibXML;

my $prsr = XML::LibXML->new();
$prsr->keep_blanks(0);

my $doc = $prsr->parse_file('poetry.xml');

foreach my $tnode (
   $doc->findnodes('//movement[@locale="American"]/../name/text()') )
{
   print $tnode->data . "\n";
}
```

First we load the module and initialize a parser object. We tell the parser object it should feel free to discard any "blank" nodes it would create during parsing (i.e., a node that gets created from whitespace). The parser gets pointed at our document, and XML::LibXML goes to work parsing it and bringing it into memory. At this point we can execute the XPath query we described above using the findnodes() method. This method performs a query and returns a list of nodes returned by that query. We iterate over each returned node (we're going to get back a list of text nodes holding the textual contents of each element), finally printing out the data in the node. It prints:

```
Joyce Kilmer
Ogden Nash
```

as expected. Code that queries the value of an element's attribute looks quite similar:

```
use XML::LibXML;

my $prsr = XML::LibXML->new();
$prsr->keep_blanks(0);

my $doc = $prsr->parse_file('poetry.xml');

# yes, this can be written as just //@locale, but I think it is good
# form to specify a bit more context so it is clear which elements'
# attributes you are targeting
foreach my $attrib ( $doc->findnodes('//movement/@locale') ) {
   print $attrib->value . "\n";
}

# output:
# American
# American
```

XML::LibXML also has a find() method that can be used to execute XPath queries that don't return nodes. This would be used for something like retrieving the result of the count() function example from above:

```
use XML::LibXML;

my $prsr = XML::LibXML->new();
$prsr->keep_blanks(0);

my $doc = $prsr->parse_file('poetry.xml');

print $doc->find('count(//movement[@locale="American"])'),"\n"; # 2
```

That's mostly all there is to it—feed the right XPath 1.0 specification to either the findnodes() or find() method and cope with what is returned.

And with that, I think it is time to bring this column to a close. Hopefully, having gotten a taste of XPath, you're going to rush right out to try it from Perl. Take care and I'll see you next time.