PEARLS OF XSLT AND XPATH 3.0 DESIGN

PREFACE

XSLT 3.0 and XPath 3.0 contain a lot of powerful and exciting new capabilities. The purpose of this paper is to highlight the new capabilities.

Have you got a pearl that you would like to share? Please send me an email and I will add it to this paper (and credit you). I ask three things:

1. The pearl highlights a capability that is new to XSLT 3.0 or XPath 3.0.
2. Provide a short, complete, working stylesheet with a sample input document.
3. Provide a brief description of the code.

This is an evolving paper. As new pearls are found, they will be added.

TABLE OF CONTENTS

CHAPTER 1: XPATH 3.0 IS A COMPOSABLE LANGUAGE

The XPath 3.0 specification says this:

XPath 3.0 is a composable language

What does that mean?

It means that every operator and language construct allows any XPath expression to appear as its operand (subject only to operator precedence and data typing constraints).

For example, take this expression:

```
3 + ____
```

The plus (+) operator has a left-operand, 3. What can the right-operand be? Answer: any XPath expression! Let's use the max() function as the right-operand:

```
3 + max(___)
```

Now, what can the argument to the max() function be? Answer: any XPath expression! Let's use a for-loop as its argument:

```
3 + max(for $i in 1 to 10 return ___)
```

Now, what can the return value of the for-loop be? Answer: any XPath expression! Let's use an if-statement:

```
3 + max(for $i in 1 to 10 return (if ($i gt 5) then ___ else ___)))
```

And so forth.

The fact that XPath is a composable language is very cool and very powerful.

Contrast this with XSLT, which is not fully composable; for example, XPath expressions can be used as operands to XSLT instructions, but not the other way around. Similarly, Java has constructs called statements that cannot be nested inside expressions.

Credit to Michael Kay

CHAPTER 2: HIGHER-ORDER FUNCTIONS

In almost all programming languages you can create a function and pass to it an integer or a string (or some other data type). You can create functions that return an integer or a string. And you can assign to a variable an integer or a string. Thus, integers and strings are *first-class values*. For a long time many of the popular programming languages did not allow you to pass a function to a function nor create a function that returned a function nor assign a function to a variable. Thus functions were treated as *second-class values*. But many of today's programming languages have elevated functions to first-class values. In fact, XSLT 3.0 and XPath 3.0 have done so. XSLT 3.0 and XPath 3.0 now allow you to pass functions to a function, create functions that return a function, and assign a function to a variable.

Functions that manipulate functions are called *higher-order functions*.

Let's see how to implement higher-order functions using XSLT 3.0 and XPath 3.0.

Example: the following variable holds the altitude of an aircraft in feet:

```
<xsl:variable name="altitude-in-feet" select="1200" />
```

I would like to output this value in other units such as meters and yards. So I created an anonymous function that is invoked by passing to it a value and a function:

```
function(
            $value as xs:decimal,
            $f as function(item()*) as item()*
        )
```

It returns a decimal:

```
function(
                $value as xs:decimal,
                $f as function(item()*) as item()*
        )
        as xs:decimal
```

What does it do? Answer: it applies $f to $value:

```
function(
                $value as xs:decimal,
                $f as function(item()*) as item()*
        )
        as xs:decimal
   {$f($value)}
```

We have created a higher-order function!

Now let’s assign it to a variable:

```
<xsl:variable name="unit-converter" select="function(
                                                              $value as xs:decimal,
                                                              $f as function(item()*) as item()*
                                                           )
                                                          as xs:decimal
                                                   {$f($value)}" />
```

So $unit-converter is a function that takes two arguments:

(1) a decimal value

(2) a function

and it applies the function to the value.

Let’s create another function. This function converts feet to meters:

```
<xsl:variable name="feet-to-meters" select="function(
                                                              $a as xs:decimal
                                                           )
                                                          as xs:decimal
                                                   {$a * 0.3048}" />
```

It takes a decimal argument and multiplies it by 0.3048.

Now let’s convert $altitude-in-feet to meters and output the result:

```
<xsl:value-of select="$unit-converter($altitude-in-feet, $feet-to-meters)" />
```

$unit-converter is invoked with two arguments: the value 1200 and a function that converts the value from feet to meters.

Okay, we can convert to meters; let’s now create a function that converts feet to yards:

```
<xsl:variable name="feet-to-yards" select="function(
                                                            $a as xs:decimal
                                                         )
                                                        as xs:decimal
                                                 {$a div 3}" />
```

It takes a decimal argument and multiplies it by 3.

Now let’s convert $altitude-in-feet to yards and output the result:

```
<xsl:value-of select="$unit-converter($altitude-in-feet, $feet-to-yards)" />
```

Again, $unit-converter is invoked with two arguments: the value 1200 and a function that converts the value from feet to yards.

Notice that function $unit-converter can convert any value to any units simply by providing it a conversion function.

$unit-converter is a *higher-order function*.

CHAPTER 3: PARTIAL FUNCTIONS

“Partial functions” is an important concept in functional programming.

A function that takes two arguments may be rewritten as a function that takes one argument and returns a function. The function returned also takes one argument. For example, consider a min function which returns the minimum of two integers (I show this in pseudocode):

```
min 2 3                    -- returns 2
```

min is a function that takes two arguments.

Suppose min is given only one argument:

```
min 2
```

[Definition: When fewer arguments are given to a function than it can accept it is called partial application of the function. That is, the function is partially applied.]

min 2 is a function. It takes one argument. It returns the minimum of the argument and 2. For example:

```
(min 2) 3          -- returns 2
```

To see this more starkly, let's assign g to be the function min 2:

```
let g = min 2
```

g is now a function that takes one argument and returns the minimum of the argument and 2:

```
g 3                -- returns 2
```

Let's take a second example: the addition operator (+) is a function and it takes two arguments. For example:

```
2 + 3              -- returns 5
```

To make it more obvious that (+) is a function, here is the same example in prefix notation:

```
(+) 2 3            -- returns 5
```

Suppose we provide (+) only one argument:

```
(+) 2
```

That is a partial application of the (+) function.

(+) 2 is a function. It takes one argument. It returns the sum of the argument and 2. For example:

```
((+) 2) 3           -- returns 5
```

We can succinctly express (+) 2 as:

```
(+2)
```

Thus,

```
(+2) 3              -- returns 5
```

Now that you have intuition about what partial functions are and how to use them, let's see how to express partial functions in XPath 3.0. To express, "I am omitting an argument because I will provide it latter" you use the question mark (?). For example, suppose that we wish to use the concat function to concatenate three values, but we provide only the first value:

```
let $concat3 := concat#3('Section', ?, ?)
```

That says, "Create a new variable, $concat3, and assign to it the concat function. The concat function's first argument is 'Section' and the next two arguments are to be provided latter." So the concat function is partially defined.

Notice the #3 after concat. It indicates the "arity" of the function. In this case it indicates that the concat function is to have three arguments. The arity must be specified when a partially defined function is assigned to a variable.

The XPath 3.0 "let" statement has a matching "return" statement. In the return statement the remaining two values are provided:

```
return $concat3(': ', 1)
```

The result is: Section: 1

Here is a stylesheet that shows 4 equivalent ways to concatenate three values; the latter 3 ways use partial functions:

```
<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="concat('Section', ': ', 1)" />

      <xsl:value-of select="
         let $concat3 := concat#3     (: You must specify the arity
                                 of the function :)
         return $concat3('Section', ': ', 1)
         " />

      <xsl:value-of select="
         let $concat3 := concat(?,?,?)
         return $concat3('Section', ': ', 1)
         " />

      <xsl:value-of select="
         let $concat3 := concat#3('Section', ?, ?)
         return $concat3(': ', 1)
         " />
   </xsl:template>

</xsl:stylesheet>
```

Above we used the min function to introduce partial functions. How is that expressed using XPath 3.0? Interestingly, partial functions are not used:

```
<xsl:value-of select="
     let $min2 := function($x) { min(($x, 2)) }
     return $min2(3)
     " />
```

Understanding that code will require an understanding of closures -- see chapter 6. Credit Michael Kay for this implementation of the min function.

CHAPTER 4: FUNCTION COMPOSITION

The ability to compose functions is arguably the most important feature of any functional programming language. So it is important to know how to compose functions in XPath 3.0.

The easiest way to explain function composition is with an example.

Example: Let's compose these two functions:

1. *increment*: this function adds 1 to its argument.

2. *double*: this function multiplies its argument by 2.

In the Haskell programming language the two functions are composed like so:

```
f = double . increment
```

The dot (.) is the composition operator. Function composition means the output of the *increment* function is to be used as the input to the *double* function.

f is a new function (it is the composition of *double* and *increment*). Function *f* can be applied to an argument:

```
f 2
```

The result is 6. That is the expected result:

- add 1 to the argument: (2 + 1)

- then multiply by 2: (2 + 1) * 2

- to produce the result: (2 + 1) * 2 = 6

XPath 3.0 Implementation

Function *increment* is implemented in XPath 3.0 like so:

```
$increment := function($x as xs:integer) {$x + 1}
```

Read as: the variable *increment* is assigned a value that is an (unnamed) function which takes as its argument an (XML Schema) integer *x*. The function returns the value of *x* plus 1.

Function *double* is implemented:

```
$double := function($y as xs:integer) {$y * 2}
```

Read as: the variable *double* is assigned a value that is an (unnamed) function which takes as its argument an (XML Schema) integer *y*. The function returns the value of *y* multiplied by 2.

Next, we create an (unnamed) function that composes two functions, *a* and *b*. It returns an (unnamed) function. The returned function takes one argument, *c*, and returns the result of applying *a* to *c* and then *b* to *a(c)*:

```
$compose := function(
                                    $a as function(item()*) as item()*,
                                    $b as function(item()*) as item()*
                                 )
                                as function(item()*) as item()*
                        {function($c as item()*) as item()* {$b($a($c))}}
```

Read as: the variable *compose* is assigned a value that is an (unnamed) function which takes two arguments, *a* and *b*. Both *a* and *b* are functions. The unnamed function returns an unnamed function.

The returned function takes one argument, *c*. The implementation of the returned function is to apply *a* to *c* and then apply *b* to the result.

This is a great example of a *higher-order function*. A higher-order function is a function that takes as its arguments function(s) and/or returns as its result a function. In this example, the function takes *two* functions as arguments *and* returns a function.

Next, we assign to variable *f* the result of composing function *increment* and *double*:

```
$f := $compose($increment, $double)
```

Lastly, apply *f* to some value, say 2:

```
$f(2)
```

Here is the complete XPath 3.0 program:

```
let $increment  :=  function($x as xs:integer) {$x + 1},
    $double      :=  function($y as xs:integer) {$y * 2},
    $compose    :=  function(
                                        $a as function(item()*) as item()*,
                                        $b as function(item()*) as item()*
                                     )
                                     as function(item()*) as item()*
                            {function($c as item()*) as item()* {$b($a($c))}},
     $f  :=  $compose($increment, $double)
```

```
return $f(2)
```

Observe these things:

- Each statement is separated from the next by a comma

- The first statement is initiated with the keyword, *let*

- The format is: *let statement, statement, …, statement return expression*

That XPath program can be used in an XSLT 3.0 program, an XQuery 3.0 program, and any language that supports XPath 3.0.

That is a beautiful program.

Credit to Michael Kay.

CHAPTER 5: RECURSION WITH ANONYMOUS FUNCTIONS

XPath 3.0 does not support named functions. It only provides unnamed (anonymous) functions. So recursion is a bit tricky – how do you recurse in a function that has no name? This chapter shows how.

The easiest way to explain the technique is with an example.

Example: Let's implement a recursive *until* function (loop until some condition is satisfied).

Function *until* takes three arguments:

1. *p* is a boolean function
2. *f* is a function on *x*
3. *x* is the value being processed

Here is an example using the function:

```
$until ($isNegative, $decrement, 3)
```

Read as: decrement 3 until it is negative. The result is (-1).

XPath 3.0 Implementation

Function *isNegative* is implemented in XPath 3.0 like so:

```
$isNegative  :=  function($x as xs:integer) {$x lt 0}
```

Read as: the variable *isNegative* is assigned a value that is an (unnamed) function which takes as its argument an (XML Schema) integer *x*. The function returns True if *x* is less than 0 and False otherwise.

Function *decrement* is implemented:

```
$decrement  :=  function($x as xs:integer) {$x - 1}
```

Read as: the variable *decrement* is assigned a value that is an (unnamed) function which takes as its argument an (XML Schema) integer *x*. The function returns the value of *x* minus 1.

Now it is time to implement function *until*. Here it is:

```
$until := function(
                        $p as function(item()*) as xs:boolean,
                        $f as function(item()*) as item()*,
                        $x as item()*
```

```
                         ) as item()*
             {$helper($p, $f, $x, $helper)}
```

Read as: the variable *until* is assigned a value that is an (unnamed) function which takes 3 arguments: a function, *p*, that takes an argument and returns a boolean value, a function, *f*, that takes an argument and returns a result, and a value, *x*. The function invokes another function that I called *helper*. Function *helper* is invoked with *p*, *f*, *x*, and itself.

This is key: to implement recursion with unnamed functions you must assign the unnamed function to a variable and invoke the function by passing it the variable.

Okay, let's look at the helper function:

```
$helper  := function(
                                  $p as function(item()*) as xs:boolean,
                                  $f as function(item()*) as item()*,
                                  $x as item()*,
                                  $helper as function(function(), function(), item()*, function()) as item()*
                             ) as item()*
               {if ($p($x)) then $x else $helper($p, $f, $f($x), $helper)}
```

Read as: the variable *helper* is assigned a value that is an (unnamed) function which takes 4 arguments: *p*, *f*, *x*, and *helper*. The function applies p to x and if the result is True then it returns x, otherwise it recurses, calling *helper* with *p*, *f*, *f(x)*, and *helper*.

Lastly, let's apply *until* to some value, say 3: :

```
$until($isNegative, $decrement, 3)
```

The result is the value (-1).

Here is the complete XPath 3.0 program:

```
let $isNegative := function($x as xs:integer) {$x lt 0},
    $decrement  := function($x as xs:integer) {$x - 1},
    $helper  := function(
                                 $p as function(item()*) as xs:boolean,
                                 $f as function(item()*) as item()*,
                                 $x as item()*,
                                 $helper as function(function(), function(), item()*, function()) as item()*
                              ) as item()*
                      {if ($p($x)) then $x else $helper($p, $f, $f($x), $helper)},
     $until := function(
                               $p as function(item()*) as xs:boolean,
                                    $f as function(item()*) as item()*,
                                    $x as item()*
                                  ) as item()*
                            {$helper($p, $f, $x, $helper)}
return $until($isNegative, $decrement, 3)
```

This is a fantastic technique.

Credit to Dimitre Novatchev for discovering this technique.

CHAPTER 6: CLOSURES

Recall that a higher-order function can return a function. Suppose a function *f* is invoked and it returns a function *g*. And into that returned function *f* stores some of its local data. That returned function *g* is called a closure.

Definition: A closure is a function that was returned from another function and the returned function contains data from the returning function.

Let's take an example.

The following anonymous function is invoked by passing to it a greeting; it returns a function:

```
function(
                $greeting as xs:string
              )
             as function(xs:string) as xs:string
```

To make the returned function a closure we store data into the body of the returned function (in this case we store the value of $greeting into the returned function):

```
function(
                $greeting as xs:string
              )
             as function(xs:string) as xs:string
   {function($name as xs:string) as item() {$greeting || $name}}" />
```

Note: XPath 3.0 has a new operator, ||, for concatenating two strings.

Look at the body of the returned function (i.e., everything between the curly braces). Read it as: return a function that takes one argument, $name, and it concatenates $greeting with $name. The body of the returned function *up-references* $greeting. Thus, $greeting is kept alive after the function returns.

Let's assign the whole thing to a variable:

```
<xsl:variable name="f" select="function(
                                                   $greeting as xs:string
                                                 )
                                                as function(xs:string) as xs:string
                                       {function($name as xs:string) as item() {$greeting || $name}}" />
```

We invoke the function $f with a string:

```
$f('Hello: ')
```

As we’ve seen, $f returns this function:

```
function($name as xs:string) as item() {$greeting || $name}
```

That is a closure -- the body is up-referencing a value from $f.

Let’s assign the returned function to a variable:

```
<xsl:variable name="sayHello" select="$f('Hello: ')" />
```

Remember, $sayHello is a function and we invoke it by passing to it a name. So let’s pass to it “John” and then output the result:

```
<xsl:value-of select="$sayHello('John')" />
```

Output: Hello: John

We invoked $sayHello with a simple name and it operated on it using data from $f. This was accomplished using a closure.

Here is the code in its entirety:

```
<?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">

  <xsl:output method="text"/>

  <xsl:variable name="f" select="function(
                                          $greeting as xs:string
                                        )
                                        as function(xs:string) as xs:string
                                 {function($name as xs:string) as item() {$greeting || $name}}" />

  <xsl:variable name="sayHello" select="$f('Hello: ')" />

  <xsl:template match="Name">
    <xsl:value-of select="$sayHello('John')" />
  </xsl:template>

</xsl:stylesheet>
```

In the example, the data inserted into the closure was a hardcoded string. Let’s create a closure that contains data from an XML document. That’s next.

How to create a function that has context, without passing it context

Suppose you would like to create a function that, given the title of a book, it returns the author.

For example, the function call:

```
author('The Society of Mind')
```

returns the author:

```
Marvin Minsky
```

Notice that the function, author(), was just provided a string and no context. How can the function obtain the author without any context? Recall that functions supposedly have no context and you must give it (through parameters) all the context it needs.

I show how to create functions that have context, but you don't have to give it the context.

It is accomplished using closures.

Here is the XML document that the function will operate on:

```
<Books>
   <Book>
      <Title>Six Great Ideas</Title>
      <Author>Mortimer J. Adler</Author>
   </Book>
   <Book>
      <Title>The Society of Mind</Title>
      <Author>Marvin Minsky</Author>
   </Book>
</Books>
```

I create a variable, author, that is defined to be an XPath 3.0 anonymous function:

```
<xsl:variable name="author" select="... define author as an anonymous function ... " />
```

Then I invoke the "author function" with a string representing the title of a book:

```
<xsl:value-of select="$author('The Society of Mind')" />
```

Output: Marvin Minsky

This is key:

> *Create a variable that is defined to be a function that returns a function and*
>
> *store context into the returned function.*

Let’s step through this very carefully.

I create a variable, root, to which I pass the root element of the XML document (Books):

```
<xsl:variable name="root" select="function(
                                            $root_ as element(Books)
                                        )
```

The function returns a function:

```
<xsl:variable name="root" select="function(
                                            $root_ as element(Books)
                                        )
                                        as function(xs:string) as item()
```

The function that is returned takes an argument that is a string representing the title of a book and it returns the author of the book:

```
<xsl:variable name="root" select="function(
                                            $root_ as element(Books)
                                        )
                                        as function(xs:string) as item()
                          {function($title as xs:string) as item() {$root_/Book[Title eq $title]/Author}}" />
```

Observe that the returned function is up-referencing the <Books> element (which is in the variable $roots_).

The returned function is a closure.

Remember the author variable? It is assigned the returned function:

```
<xsl:variable name="author" select="$root(/Books)" />
```

Okay, let's put it all together:

```
<?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">

   <xsl:output method="text"/>

   <xsl:variable name="root" select="function(
                                                         $root_ as element(Books)
                                                       )
                                                      as function(xs:string) as item()
                       {function($title as xs:string) as item() {$root_/Book[Title eq $title]/Author}}" />

   <xsl:variable name="author" select="$root(/Books)" />

   <xsl:template match="Books">
      <xsl:value-of select="$author('The Society of Mind')" />
   </xsl:template>

</xsl:stylesheet>
```

CHAPTER 7: BINARY SEARCH TREES

This chapter contains code for creating and processing binary search trees. The code is implemented entirely in XPath. An advantage of an "XPath-only" solution is that it can be used (hosted) in many programs—it can be hosted in XSLT programs, in XQuery programs, and in any other programming language that hosts XPath. Thus "XPath-only" solutions are highly portable and reusable.

The XPath code inputs data from an XML document and stores the data into a binary search tree. By storing the data in a binary tree there are many operations that can be efficiently performed which would otherwise be difficult or inefficient.

Regarding efficiency, we can provide some timing data here: without using a binary-search tree the time complexity for searching is O(N). With a balanced binary tree the time complexity is O($\log_2$N). To be even more concrete: $\log_2$(1000000) = 19 – this gives us a 50,000 time speed-up.

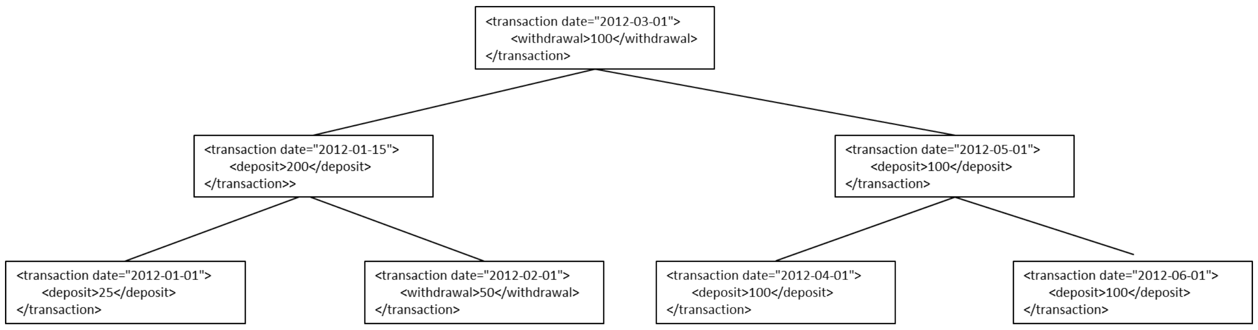
Note: if a binary search tree is severely malformed, then the search becomes O(N) – this is the *worst case* scenario. That said, there are algorithms for ensuring a "perfect" population of a binary tree from a sequence of items; in the resulting tree the height-difference of any left and right subtree is at most 1.

We present an example for which a binary tree is well-suited: find all the bank transactions within a range of dates.

Example: the following XML document contains a list of bank transactions (withdrawals and deposits). Each transaction is stamped with a date. The transactions are in no particular order:

```
<Transactions>
    <transaction date="2012-03-01">
        <withdrawal>100</withdrawal>
    </transaction>
    <transaction date="2012-01-15">
        <deposit>200</deposit>
    </transaction>
    <transaction date="2012-05-01">
        <deposit>100</deposit>
    </transaction>
    <transaction date="2012-02-01">
        <withdrawal>50</withdrawal>
    </transaction>
    <transaction date="2012-06-01">
        <deposit>100</deposit>
    </transaction>
    <transaction date="2012-04-01">
        <deposit>100</deposit>
    </transaction>
    <transaction date="2012-01-01">
        <deposit>25</deposit>
    </transaction>
</Transactions>
```

We will input the list of transactions and insert them into a binary tree as follows: for each transaction, if its date is less than the root node's date then it is inserted into the left subtree, if its date is greater than the root node's date then it is inserted into the right subtree. Using that algorithm, here is the binary tree for the above set of transactions:



<transaction date="2012-03-01"> <withdrawal>100</withdrawal> </transaction>
<transaction date="2012-01-15"> <deposit>200</deposit> </transaction>>
<transaction date="2012-05-01"> <deposit>100</deposit> </transaction>
<transaction date="2012-01-01"> <deposit>25</deposit> </transaction>
<transaction date="2012-02-01"> <withdrawal>50</withdrawal> </transaction>
<transaction date="2012-04-01"> <deposit>100</deposit> </transaction>
<transaction date="2012-06-01"> <deposit>100</deposit> </transaction>

Task: Find all the transactions in the range 2012-03-15 to 2012-05-15. This task is efficiently accomplished now that the transactions are stored in a binary tree. (The more balanced the binary search tree is, the more efficient will be the implementation to produce the results.)

Here is pseudocode that shows how to find the relevant transactions (start date = 2012-03-15, end date = 2012-05-15):

```
If the value of the root node equals the start date, then
    - return the root node
    - recurse on the right subtree

If the value of the root node equals the end date, then
    - recurse on the left subtree
    - return the root node

If the value of the root node is between the start and end date, then
    - recurse on the left subtree
    - return the root node
    - recurse on the right subtree

If the value of the root node is less than the start date, then
    - recurse on the right subtree

If the value of the root node is greater than the end date, then
    - recurse on the left subtree
```

Below is the actual XPath code.

XSLT or XPath?

All the code in this chapter could have been implemented using XSLT functions. However, there are practical reasons for favoring XPath: the XPath code is shorter and easier to understand, it can be used in XSLT, and it can be "ported" to XQuery just with a simple copy and paste operation.

Recommendation: given the choice between implementing some functionality using XSLT or XPath, it may be wiser to use an XPath-only solution especially if there is even a remote possibility that the functionality would be needed in more than one different languages.

```
$find-range-of-transactions :=
            function( $tree as function()*,
                      $start-date as xs:date,
                      $end-date as xs:date
                    )
            {
              $find-range-of-transactions-helper
                                        ( $tree,
                                               $start-date,
                                         $end-date,
                                         $find-range-of-transactions-helper)
            }
```

The function's name is find-range-of-transactions. It is a recursive function. As described in Chapter 5, implementing recursion using anonymous functions requires a "helper" function. Here is the helper function find-range-of-transactions-helper:

```
$find-range-of-transactions-helper :=
            function( $tree as function()*,
                      $start-date as xs:date,
                      $end-date as xs:date,
                      $find-range-of-transactions-helper
                    )
                    as element(transaction)*
            {
              if (empty($tree)) then ()
              else
                if (xs:date($root($tree)/@date) eq $start-date) then
                  (
                    $root($tree),
                    $find-range-of-transactions-helper
                              ( $right($tree),
                                $start-date,
                                $end-date,
                  $find-range-of-transactions-helper)
      )
     else
       if (xs:date($root($tree)/@date) eq $end-date) then
        (
          $find-range-of-transactions-helper
                    ( $left($tree),
                     $start-date,
                     $end-date,
                     $find-range-of-transactions-helper),
          $root($tree)
        )
       else
         if ((xs:date($root($tree)/@date) gt $start-date)
            and
           (xs:date($root($tree)/@date) lt $end-date)) then
          (
            $find-range-of-transactions-helper
                       ( $left($tree),
                        $start-date,
                        $end-date,
                        $find-range-of-transactions-helper),
            $root($tree),
            $find-range-of-transactions-helper
                       ( $right($tree),
                        $start-date,
                        $end-date,
                        $find-range-of-transactions-helper)
          )
         else
           if (xs:date($root($tree)/@date) lt $start-date) then
            (
              $find-range-of-transactions-helper
                         ( $right($tree),
                          $start-date,
                          $end-date,
                          $find-range-of-transactions-helper)
            )
           else
             if (xs:date($root($tree)/@date) gt $end-date) then
              (
                 $find-range-of-transactions-helper
                           ( $left($tree),
                            $start-date,
                            $end-date,
                            $find-range-of-transactions-helper)
              )
             else ()
}
```

As items are inserted into the binary tree, a comparison is made between the item being inserted and the item in the tree's root node. How should the comparison be done? That depends on the particular items stored in the tree. Consequently, the "insert function" must be provided an appropriate "comparator function." For our bank transactions example, the comparison is done based on the date attribute. Here is an appropriate comparator function:

```
$transaction-less-than-comparator :=
            function($lhs as element(transaction),
                        $rhs as element(transaction)
                       ) as xs:boolean
            {
                    xs:date($lhs/@date) lt xs:date($rhs/@date)
             }
```

The value of the variable is an anonymous function. The function takes two arguments – both transaction elements – and returns true if the date in the first transaction is less than the date in the second transaction.

A binary tree is a collection of these functions:

1. **create**: create an empty tree
2. **root**: return the value of the root node
3. **left**: return the left subtree
4. **right**: return the right subtree
5. **empty**: return true if the tree is empty, false otherwise
6. **insert**: insert an item into the tree, with comparison done using a comparator (see above)
7. **print**: serialize the tree as XML
8. **populate**: insert a sequence of items into the binary tree

Other functions may be created but this collection is sufficient for our task. (The most glaring omission from this list is a function to delete nodes from the tree. See Dimitre Novatchev's blog for the delete function and others.)

Below is the entire XPath code for both the binary tree functions and the functions to find all the bank transactions between a start date and an end date. There are comments before each function that hopefully suffices for understanding. As mentioned previously, this XPath code can be used in an XSLT program, an XQuery program, and any other programming language that hosts XPath.

```
let
        (:
           The purpose of create is to return an empty tree.

           It returns a sequence of functions,
              - the first function represents the root of a tree,
              - the second function represents the left subtree,
              - and the third function represents the right subtree.
           The value of each function is an empty sequence.
        :)
        $create := (
                        function() { () }     (: root :),
                        function() { () }     (: left :),
                        function() { () }     (: right :)
                      ),


        (:
           The purpose of empty is to return a boolean value,
           indicating whether $tree is empty.

           $tree is empty in these two circumstances:
           1. $tree is the empty sequence (it doesn't contain any functions).
           2. $tree contains a sequence of three functions, but the first
              function - representing the root - is empty (i.e., if you invoke the
              first function it returns the empty sequence).

           Note: if the first function (root) returns an empty sequence,
           then the other two functions (left and right) must also contain
           the empty sequence. You can't have an empty root but a non-empty
           left or right subtree.
        :)
        $empty := function($tree as function()*)
                     {
                        empty($tree) or empty($tree[1]())
                     },


        (:
           The purpose of root is to return the value of the root node.

           This function takes one argument, $tree. Since $tree
           is represented by a sequence of functions, returning the
           value of the root node actually means returning the value of
           the function that corresponds to the root node.

           If $tree is empty then it returns the empty sequence. Otherwise
           it returns the *value* of the first function in $tree (recall
   that a tree is represented by a sequence of functions, the first
   function representing the root of the tree).

   Note: $tree[1] returns the first function whereas
            $tree[1]() returns the *value* of the first function.
:)
$root := function($tree as function()*)
            {
                 if ($empty($tree)) then ()
                 else $tree[1]()
            },



(:
   The purpose of left is to return the value of the left subtree.

   This function takes one argument, $tree. Since $tree is represented
   by a sequence of functions, returning the value of the left subtree
   actually means returning the value of the function that corresponds
   to the left subtree.

   If $tree is empty then it returns the empty sequence. Otherwise
   it returns the *value* of the second function in $tree (recall
   that a tree is represented by a sequence of functions, the second
   function representing the left subtree).

   Note: $tree[2] returns the second function whereas
            $tree[2]() returns the *value* of the second function.
:)
$left := function($tree as function()*)
           {
                if ($empty($tree)) then ()
                else
                     if ($empty($tree[2])) then ()
                     else $tree[2]()
           },



(:
   The purpose of right is to return the value of the right subtree.

   This function takes one argument, $tree. Since $tree is represented
   by a sequence of functions, returning the right subtree actually
   means returning the value of the function that corresponds to the
   right subtree.

   If $tree is empty then it returns the empty sequence. Otherwise
   it returns the *value* of the third function in $tree (recall
    that a tree is represented by a sequence of functions, the third
    function representing the right subtree).
 :)
 $right := function($tree as function()*)
               {
                    if ($empty($tree)) then ()
                   else
                         if ($empty($tree[3])) then ()
                         else $tree[3]()
               },


(:
   As items are inserted into a binary tree, a comparison is made between
   the item being inserted and the item in the tree's root node. How should
   the comparison be done? That depends on the particular items in the tree.
   Comparing two integers will be different than comparing two tree fragments.
   So, the insert function must be provided an appropriate comparator. For
   the case of bank transactions, the comparison is done based on the date
   attribute. Here is an appropriate comparator function:
:)
$transaction-less-than-comparator :=
              function( $lhs as element(transaction),
                               $rhs as element(transaction)
                             ) as xs:boolean
              {
                    xs:date($lhs/@date) lt xs:date($rhs/@date)
              },


 $numeric-less-than-comparator :=
               function( $lhs as xs:decimal,
                                $rhs as xs:decimal
                              ) as xs:boolean
               {
                     number($lhs) lt number($rhs)
               },


 (:
    The purpose of insert is to insert $item into the proper place
    in $tree.

    Here's the proper place:
    - if $tree is empty then place it in the root node
    - if $item is less than the value in the root node then
      insert it into the left subtree (note the recursive definition)
    - else insert $item into the right subtree (again note the recursive
     definition)

    Here are the steps taken if $tree is empty:
    - $item is inserted into the root function. That is, the root function,
      if invoked, will return $item.
    - A left function is created such that, if invoked, will return an empty subtree.
    - A right function is created such that, if invoked, will return an empty subtree.

    The insert function is recursive. Recursion with anonymous functions requires
    a helper function. See chapter 5, Recursion, for details.
:)
$insert-helper :=
        function( $tree as function()*,
                  $item as item(),
                  $less-than-comparator as function(item(), item()) as xs:boolean,
                  $insert-helper
                 )
        {
        if ($empty($tree)) then
           (
              function() {$item}    (: root :),
              function() {()}        (: left :),
              function() {()}        (: right :)
           )
         else if ($less-than-comparator($item, $root($tree))) then
           (
              function() {$root($tree)}                  (: root :),
              function() {$insert-helper( $left($tree),
                                          $item,
                                          $less-than-comparator,
                                          $insert-helper)
                         }                               (: left :),
              function() {$right($tree)}                 (: right :)
            )
         else
           (
              function() {$root($tree)}                  (: root :),
              function() {$left($tree)}                  (: left :),
              function() {$insert-helper( $right($tree),
                                          $item,
                                          $less-than-comparator,
                                          $insert-helper)
                          }                               (: right :)
            )
        },


$insert :=
            function ( $tree as function()*,
                       $item as item(),
                       $less-than-comparator as function(item(), item()) as xs:boolean
                     )
            {
                 $insert-helper($tree, $item, $less-than-comparator, $insert-helper)
            },


(:
    The purpose print is to return as XML the contents of
    each node in $tree.

    Print returns a tree element that consists of three child elements:
    1. A root element containing the content of the root function.
    2. A left element containing the printing of the left subtree
       (note the recursive definition).
    3. A right element containing the printing of the right subtree
       (again, note the recursive definition).

    An interesting technical challenge arose in creating this function:
    There cannot be XML markup in the body of an XPath function.
    So how does an XPath function generate new XML nodes?
    The solution is to escape the markup (thus producing
    simply a string) and then invoke the parse-xml()
    function to interpret the string as XML markup.
    This is a brilliant and powerful insight by Dimitre.

    The $print-helper function returns the markup
    as a string and then the $printer function converts
    it into XML using the parse-xml() function.
:)
$print-helper :=
            function ( $tree as function()*,
                       $print-helper
                      )
                      as xs:string?
            {
            if (not($empty($tree))) then
                   concat('&lt;tree>',
                                '&lt;root>',
                                     $root($tree),
                                '&lt;/root>',
                                '&lt;left>',
                                     $print-helper($left($tree),$print-helper),
                                '&lt;/left>',
                                '&lt;right>',
                                     $print-helper($right($tree),$print-helper),
                                          '&lt;/right>',
                                   '&lt;/tree>'
                                  )
                else ()
                },


$print := function ($tree as function()*)
                {parse-xml($print-helper($tree, $print-helper))/*},


(:
   The purpose of populate is to sequentially insert
   the values in $items into $tree.
:)
$populate-helper :=
            function ( $tree as function()*,
                          $items as item()*,
                          $less-than-comparator as function(item(), item()) as xs:boolean,
                          $populate-helper
                        )
            {
               if (empty($items)) then $tree
               else
                   $populate-helper( $insert($tree, $items[1], $less-than-comparator),
                                             $items[position() gt 1],
                                             $less-than-comparator,
                                             $populate-helper
                                           )
            },


$populate :=
            function( $tree as function()*,
                          $items as item()*,
                          $less-than-comparator as function(item(), item()) as xs:boolean
                        )
            {$populate-helper($tree, $items, $less-than-comparator, $populate-helper)},


(:
   Task: find all the bank transactions in the range 2012-03-15 to 2012-05-15.
   This can be accomplished efficiently now that the transactions are stored
   in a binary tree. (The more balanced the binary search tree is, the
   more efficient will be the implementation.)
:)
$find-range-of-transactions-helper :=
            function( $tree as function()*,
                     $start-date as xs:date,
                     $end-date as xs:date,
                     $find-range-of-transactions-helper
                )
                as element(transaction)*
            {
              if (empty($tree)) then ()
              else
                if (xs:date($root($tree)/@date) eq $start-date) then
                   (
                       $root($tree),
                       $find-range-of-transactions-helper
                                   ( $right($tree),
                                     $start-date,
                                     $end-date,
                                     $find-range-of-transactions-helper)
                   )
                  else
                    if (xs:date($root($tree)/@date) eq $end-date) then
                     (
                         $find-range-of-transactions-helper
                                     ( $left($tree),
                                       $start-date,
                                       $end-date,
                                       $find-range-of-transactions-helper),
                         $root($tree)
                     )
                    else
                      if ((xs:date($root($tree)/@date) gt $start-date)
                          and
                         (xs:date($root($tree)/@date) lt $end-date)) then
                       (
                           $find-range-of-transactions-helper
                                       ( $left($tree),
                                         $start-date,
                                         $end-date,
                                         $find-range-of-transactions-helper),
                           $root($tree),
                           $find-range-of-transactions-helper
                                       ( $right($tree),
                                         $start-date,
                                         $end-date,
                                         $find-range-of-transactions-helper)
                       )
                      else
                        if (xs:date($root($tree)/@date) lt $start-date) then
                         (
                           $find-range-of-transactions-helper
```

```
                                                   ( $right($tree),
                                                     $start-date,
                                                     $end-date,
                                                     $find-range-of-transactions-helper)
                                )
                               else
                                  if (xs:date($root($tree)/@date) gt $end-date) then
                                   (
                                     $find-range-of-transactions-helper
                                                     ( $left($tree),
                                                                  $start-date,
                                                                  $end-date,
                                                                  $find-range-of-transactions-helper)
                                   )
                                  else ()
                 },


        $find-range-of-transactions :=
                  function($tree as function()*,
                              $start-date as xs:date,
                              $end-date as xs:date
                             )
                  {
                     $find-range-of-transactions-helper
                                   ( $tree,
                                     $start-date,
                                     $end-date,
                                     $find-range-of-transactions-helper)
                  }


   (:  At last, we finalize this big, outermost *let* clause with a *return* clause
      that expresses the intent of the users of our XPath-function-library.
      We want to get all transactions in the period:
      15th March 2012 to 15th May 2012.
   :)

return (
          $find-range-of-transactions
                        ( $populate((), //transaction, $transaction-less-than-comparator),
                          xs:date('2012-03-15'),
                          xs:date('2012-05-15')
                        )
         )
```

Assessment: This example reveals some things missing in XPath 3.0 that would make creating XPath-function-libraries more useful:

1. Our example contained one huge let-return expression. For better modularity it would have been nice to place all the binary search tree functions (create, left, right, insert, etc.) into its own "module" which could then be "imported" by the bank transaction functions. Unfortunately, XPath does not support this. *Recommendation to the XPath Working Group*: Support XPath expression files and an import clause to collect such expressions from files in a desired new, client XPath program.

2. Our example implemented binary search trees as a sequence of functions. That's okay, but it would be much nicer if XPath had a "tuple type" so that a tree could be simply defined as a tuple: tree is a tuple (root, left, right). *Recommendation to the XPath Working Group*: Support a tuple type, so that the result type of various functions (such as the creation of a tree) can be more precisely specified than just a sequence of function items.

3. This recommendation is along the lines of the first recommendation. *Recommendation to the XPath Working Group*: Support the capability to create a new XPath expression dynamically by combining a set of file-residing XPath expressions in a "let" clause and adding a user-defined "return" clause – and evaluating this as a new XPath ("let") expression.

Credits: Great credit goes to Dimitre Novatchev for this binary tree code. The code possesses several brilliant insights (such as using the parse-xml() function for enabling XPath functions to return markup). See Dimitre's blog for additional binary tree functions and example usages: http://dnovatchev.wordpress.com/2012/01/09/the-binary-search-tree-data-structurehaving-fun-with-xpath-3-0/

Credit to Michael Kay for the transaction example.