# 3 Common Patterns in XML Processing

Roger L. Costello
February 2011

## XML Documents Contain Lists

Most XML documents contain lists of elements. For example, this document contains a list of `<book>` elements:

```
<?xml version="1.0" encoding="UTF-8"?>
<books>
    <book>
        <title>The Origin of Wealth</title>
        <author>Eric D. Beinhocker</author>
        <date>2006</date>
        <ISBN>1-57851-777-X</ISBN>
        <publisher>Harvard Business School Press</publisher>
        <cost currency="USD">29.95</cost>
    </book>
    <book>
        <title>DOM Scripting</title>
        <author>Jeremy Keith</author>
        <date>2005</date>
        <ISBN>1-59059-533-5</ISBN>
        <publisher>friends of ed</publisher>
        <cost currency="USD">34.99</cost>
    </book>
    …
</books>
```

Consequently, processing XML documents typically involves processing lists.

## 3 Common Ways of Processing Lists

There are 3 things that are commonly done to lists:

1. **Map**: a new list is created with the same length as the original, and each item in the new list is the result of doing one piece of work on the corresponding item in the original list.
2. **Filter**: select pieces of a list—walk through a list, selecting those that satisfy a criterion.
3. **Fold**: reduce a list to a single value—do something to every element of a list, updating an accumulator along the way, and returning the accumulator when done.

They are common list processing patterns. *It is beneficial to abstract out those common patterns so that they can be reused and thereby avoid repetitive, mundane, error-prone boilerplate code.*

This article describes the 3 patterns and then provides a link to XSLT 1.0 code that implements the patterns.

## Map

A common list processing pattern is to create a new list that is the same length as the original, where each item in the new list is the result of doing one piece of work on the corresponding item in the original list.

For example, map the above list of `<book>` elements to this list of HTML table rows:

| | | | | | |
|---|---|---|---|---|---|
| Origin of Wealth | Eric D. Beinhocker | 2006 | 1-57851-777-X | Harvard Business School Press | 2 |
| M Scripting | Jeremy Keith | 2005 | 1-59059-533-5 | friends of ed | 3 |
| s, Germs, and Steel | Jared Diamond | 2005 | 0-393-06131-0 | W. W. Norton & Company, Ltd. | 2 |
| nomics in One Lesson | Henry Hazlitt | 1946 | 0-517-54823-2 | Three Rivers Press | 1 |
| to Read a Book | Mortimer J. Adler | 1940 | 0-671-21280-x | Simon & Schuster, Inc. | 1 |
| t Make Me Think | Steve Krug | 2006 | 0-321-34475-8 | New Riders | 4 |
| tproof Ajax | Jeremy Keith | 2007 | 0-321-47266-7 | New Riders | 3 |

Notice that this HTML list has the same length as the `<book>` list and each HTML row is the result of performing one operation on the corresponding `<book>` element. The terminology is that the `<book>` list has been mapped to an HTML list.

Another example: consider this list of `<aircraft>` elements:

```
<?xml version="1.0" encoding="UTF-8"?>
<aircraft-list>
    <aircraft>
        <altitude units="feet">100</altitude>
    </aircraft>
    <aircraft>
        <altitude units="feet">200</altitude>
    </aircraft>
    <aircraft>
        <altitude units="feet">300</altitude>
    </aircraft>
    <aircraft>
        <altitude units="feet">400</altitude>
```

```
    </aircraft>
    <aircraft>
        <altitude units="feet">500</altitude>
    </aircraft>
</aircraft-list>
```

Map that list to a new list that is identical, except the `<altitude>` values are expressed in meters:

```
<?xml version="1.0" encoding="UTF-8"?>
<aircraft-list>
    <aircraft>
        <altitude units="meters">30.48</altitude>
    </aircraft>
    <aircraft>
        <altitude units="meters">60.96</altitude>
    </aircraft>
    <aircraft>
        <altitude units="meters">91.44</altitude>
    </aircraft>
    <aircraft>
        <altitude units="meters">121.92</altitude>
    </aircraft>
    <aircraft>
        <altitude units="meters">152.4</altitude>
    </aircraft>
</aircraft-list>
```

The two examples illustrate a common list processing pattern. The pattern is so common it has a name: `map`.

The `map` function takes two arguments, *`function`* and *`list`*:

```
map (function, list)
```

The `map` function applies *function* to each item in *list* and returns a new list that has the same length as the original list.

The advantage of using the `map` function is that it implements the machinery for applying *`function`* to each item in *`list`*. That machinery is boilerplate code. Let's write it once and never again. That way you can focus on the real problem -- what processing do you want to apply to each item in the list.

## Filter

A second common list processing pattern is to select pieces of a list—walk through a list, selecting those that satisfy a criterion.

For example, filter the above list of `<book>` elements, selecting only those authored by Jeremy Keith:

```
<?xml version="1.0" encoding="UTF-8"?>
<Jeremy-Keith-Books>
    <book>
        <title>DOM Scripting</title>
        <author>Jeremy Keith</author>
        <date>2005</date>
        <ISBN>1-59059-533-5</ISBN>
        <publisher>friends of ed</publisher>
        <cost currency="USD">34.99</cost>
    </book>
    <book>
        <title>Bulletproof Ajax</title>
        <author>Jeremy Keith</author>
        <date>2007</date>
        <ISBN>0-321-47266-7</ISBN>
        <publisher>New Riders</publisher>
        <cost currency="USD">34.99</cost>
        <cost currency="CAD">43.99</cost>
        <cost currency="GPB">24.99</cost>
    </book>
</Jeremy-Keith-Books>
```

The example illustrates a common list processing pattern. The pattern is so common it has a name: `filter`.

The `filter` function takes two arguments, *`function`* and *`list`*:

```
filter (function, list)
```

*`function`* is Boolean – it returns either `true` or `false`. The `filter` function walks through *`list`*, applying *function* to each item, and creating a new list composed of a copy of the items for which *`function`* returns `true`.

The advantage of using the `filter` function is that it implements the machinery for walking through *`list`*, applying *`function`* to each item, and copying the items for which *`function`*

returns `true`. That machinery is boilerplate. Let's write it once and never again. That way you can focus on the real problem -- what are the items you want from *`list`*.

## Fold

A third common list processing pattern is to reduce a list to a single value—do something to every element of a list, updating an accumulator along the way, and returning the accumulator when done.

A list may be "folded up" from the left or right. A left fold is called `foldl` and a right fold is called `foldr`.

Let's start with a `foldl` example. Fold up all the `<item>` elements in this purchase order and generate a total cost:

```
<?xml version="1.0" encoding="UTF-8"?>
<purchase-order>
    <item>
        <cost>10</cost>
    </item>
    <item>
        <cost>20</cost>
    </item>
    <item>
        <cost>19</cost>
    </item>
    <item>
        <cost>25</cost>
    </item>
    <item>
        <cost>17</cost>
    </item>
</purchase-order>
```

The list of `<item>` elements is processed from top to bottom (left to right if they were displayed on one line). The accumulator is initialized to zero (0), and the `<cost>` values are folded into the accumulator one by one to generate:

```
<Total-Cost>91</Total-Cost>
```

To distinguish between `foldl` and `foldr`, let's trace the example's execution. I will use pseudo code. Let `(+)` denote the addition operation and let `[10, 20, 19, 25, 17]` denote the `<item>` list. The accumulator is initialized to zero (0):

```
foldl (+) 0 [10, 20, 19, 25, 17]

          == fold1 (+) (0 + 10)          [20, 19, 25, 17]
          == foldl (+) (10 + 20)         [19, 25, 17]
          == foldl (+) (30 + 19)         [25, 17]
          == foldl (+) (49 + 25)         [17]
          == foldl (+) (74  + 17)        []
          == 91
```

The accumulator is added to the first item in the list and the result is recursively folded into the remaining list.

If the purchase order is processed using `foldr`, the same result is generated. However, a trace of its execution shows that the list is folded up from the right:

```
foldr (+) 0 [10, 20, 19, 25, 17]

          == 10 +                              foldr (+) 0 [20, 19,25, 17]
          == 10 + 20 +                         foldr (+) 0 [19, 25, 17]
          == 10 + 20 + 19 +                    foldr (+) 0 [25, 17]
          == 10 + 20 + 19 + 25 +               foldr (+) 0 [17]
          == 10 + 20 + 19 + 25 + 17 +          foldr (+) 0 []
          == 10 + 20 + 19 + 25 + 17 +          0
          == 10 + 20 + 19 + 25 + 17
          == 10 + 20 + 19 + 42
          == 10 + 20 + 61
          == 10 + 81
          == 91
```

The addition operation is applied to the first element of the list and the result of folding the remaining list.

The `foldr` function is preferred over the `foldl` function. The reason for this is explained next.

Both fold functions step through the list and accumulate a value. The example above showed a single number as the accumulated value. However, what you accumulate could be anything – it could be a new list!

For example, we can use folds to do an identity transform – the accumulated value is an identical list. Let's trace an application of `foldl` with a *`function`* that simply copies the accumulator and the list item. I will use `(I)` to denote this identity *`function`*:

```
foldl (I) '' [10, 20, 19, 25, 17]

          == fold1 (I) [10]                [20, 19, 25, 17]
          == foldl (I) [10, 20]            [19, 25, 17]
          == foldl (I) [10, 20, 19]        [25, 17]
          == foldl (I) [10, 20, 19, 25]    [17]
          == foldl (I) [10, 20, 19, 25, 17] []
          == [10, 20, 19, 25, 17]
```

Consider this step in the application of `foldl`:

```
foldl (I) [10, 20, 19, 25]    [17]
```

The `[17]` is appended to the list `[10, 20, 19, 25]`. In many functional programming languages, that is an expensive operation. Dimitre Novatchev has an excellent explanation for why appending to a list is an expensive operation:

> *Appending to a list causes the whole list to be copied and is O(N).*

Now, let's trace an application of `foldr`:

```
foldr (I) 0 [10, 20, 19, 25, 17]

          == [10]                 foldr (I) '' [20, 19,25, 17]
          == [10, 20]             foldr (I) '' [19, 25, 17]
          == [10, 20, 19]         foldr (I) '' [25, 17]
          == [10, 20, 19, 25]     foldr (I) '' [17]
          == [10, 20, 19, 25, 17] foldr (I) '' []
          == [10, 20, 19, 25, 17] []
          == [10, 20, 19, 25]     [17]
          == [10, 20, 19] [25, 17]
          == [10, 20] [19, 25, 17]
          == [10] [20, 19, 25, 17]
          == [10, 20, 19, 25, 17]
```

Consider this step in the application of `foldr`:

```
[10, 20, 19, 25]    [17]
```

The item `25` is prepended to the list `[17]`. In many functional programming languages, that is an instant operation. Again, Dimitre Novatchev has an excellent explanation for why it is an inexpensive operation to prepend a single item to a list:

> *Prepending a list is making just the "next pointer" of an item point to the list -- an O(1) operation.*

To recap: when using folds, use `foldr` not `foldl`.

## Higher-Order Functions

Higher-order functions are  functions that can accept functions as arguments or return functions as values.

`map`, `filter`, `foldl`, and `foldr` are higher-order functions. Make them part of your library.

## Fabulous Book

This book has a fantastic description of the `map`, `filter`, `foldl`, and `foldr` function:

> *Real World Haskell* by Bryan O'Sullivan, John Goerzen, and Don Stewart

## XSLT 1.0 Implementation of map, filter, foldl, foldr

I created an XSLT 1.0 implementation of `map`, `filter`, `foldl`, and `foldr`:

http://www.xfront.com/higher-order-functions.zip

All of the examples described in this article are contained in that zip file.