The idea here is to provide simple examples of how to get started with processing XML in R using some reasonably straightforward "flat" XML files and not worrying about efficiency.

An Example: Grades

Here is an example of a simple file in XML containing grades for students for three different tests.

```
<?xml version="1.0" ?>
<TABLE>
   <GRADES>
      <STUDENT> Fred </STUDENT>
      <TEST1> 66 </TEST1>
      <TEST2> 80 </TEST2>
      <FINAL> 70 </FINAL>
   </GRADES>
   <GRADES>
      <STUDENT> Wilma </STUDENT>
      <TEST1> 97 </TEST1>
      <TEST2> 91 </TEST2>
      <FINAL> 98 </FINAL>
   </GRADES>
</TABLE>
```

We might want to turn this into a data frame in R with a row for each student and four variables, the name and the scores on the three tests.

Since this is a small file, let's not worry about efficiency in any way. We can read the entire document tree into memory and make multiple passes over it to get the information. Our first approach will be to read the XML into an R tree, i.e. R-level XML node objects. We do this with a simple call to *xmlTreeParse*() .

```
doc = xmlRoot(xmlTreeParse("generic_file.xml"))
```

We use *xmlRoot*() to get the top-level node of the tree rather than holding onto the general document information since we won't need it.

Since the structure of this file is just a list of elements under the root node, we need only process each of those nodes and turn them into something we want. The "easiest" way to apply the same function to each child of an XML node is with the *xmlApply*() function. What do we want to do for each of the <GRADES> node? We want to get the value, i.e. the simple text within the node, of each of its children. Since this is the same for each of the child nodes in <GRADES>, this is again another call to *xmlApply*() . And since this is all text, we can simplify the result and get back a character vector rather than a list by using *xmlSApply*() which will perform this extra simplication step.

So a function to do the initial processing of an individual <GRADES> node might be

```
function(node)
     xmlSApply(node, xmlValue)
```

since *xmlValue*() returns the text content within an XML node. Let's check that this does what we want by calling it on the first child of the root node.

```
xmlSApply(doc[[1]], xmlValue)
```

And indeed it does.

So we can process all the <GRADES> nodes with the command

```
tmp = xmlSApply(doc, function(x) xmlSApply(x, xmlValue))
```

The result is a character matrix in which the rows are the variables and the columns are the records. So let's transpose this.

```
tmp = t(tmp)
```

Now, we have finished working with the XML; the rest is regular R programming.

```
grades = as.data.frame(matrix(as.numeric(tmp[,-1]), 2))
 names(grades) = names(doc[[1]])[-1]
 grades$Student = tmp[,1]
```

There seems to be more messing about after we have got the values out of the XML file. There are several things that might seem more complex but that actually just move the work to different places, i.e. when we are traversing the XML tree.

Here's another alternative using XPath.

```
doc = xmlTreeParse("generic_file.xml", useInternal = TRUE)

ans = lapply(c("STUDENT", "TEST1", "TEST2", "FINAL"),
             function(var)
               unlist(xpathApply(paste("//", var, sep = ""), xmlValue)))
```

And this gives us a list containing the variables with the values as character vectors.

```
as.data.frame(lapply(names(ans),
                     function(x) if(x != "STUDENT") as.integer(x) else x ))
```

Another Example: Customer Information List

The second example is another list, this time of description of customers. The first two nodes in the document are shown below:

```
<dataroot xmlns:od="urn:schemas-microsoft-com:officedata">
<Customers>
<CustomerID>ALFKI</CustomerID>
<CompanyName>Alfreds Futterkiste</CompanyName>
<ContactName>Maria Anders</ContactName>
<ContactTitle>Sales Representative</ContactTitle>
<Address>Obere Str. 57</Address>
<City>Berlin</City>
<PostalCode>12209</PostalCode>
<Country>Germany</Country>
<Phone>030-0074321</Phone>
```

```
<Fax>030-0076545</Fax>
</Customers>
<Customers>
<CustomerID>ANATR</CustomerID>
<CompanyName>Ana Trujillo Emparedados y helados</CompanyName>
<ContactName>Ana Trujillo</ContactName>
<ContactTitle>Owner</ContactTitle>
<Address>Avda. de la Constitución 2222</Address>
<City>México D.F.</City>
<PostalCode>05021</PostalCode>
<Country>Mexico</Country>
<Phone>(5) 555-4729</Phone>
<Fax>(5) 555-3745</Fax>
</Customers>
```

We can quickly verify that all the nodes under the root are customers with the command

```
doc = xmlRoot(xmlTreeParse("Cust-List.xml"))
 table(names(doc))
```

We see that these are all "Customers". We could further explore to see if each of these nodes has the same fields.

```
fields = xmlApply(doc, names)
table(sapply(fields, identical, fields[[1]]))
```

And the result indicates that about half of them are the same. Let's see how many unique field names there are:

```
unique(unlist(fields))
```

This gives 11. And we can see how may fields are in each of the Customers nodes with

```
xmlSApply(doc, xmlSize)
```

So most of the nodes have most of the fields.

So let's think about a data frame. What we can do is treat each of the fields as having a simple string value. Then we can create a data frame with the 11 character columns and with NA values for each of the records. Thne we will fill this in record at a time.

```
ans = as.data.frame(replicate(11, character(xmlSize(doc))),
                      stringsAsFactors = FALSE)
names(ans) = unique(unlist(fields))
```

Now that we have the skeleton of the answer, we can process each of the Customers nodes.

```
sapply(1:xmlSize(doc),
          function(i) {
             customer = doc[[i]]
             ans[i, names(customer)] <<- xmlSApply(customer, xmlValue)
          })
```

Note that we used a global assignemnt in the function to change the **`ans`** in the global environment rather than the local version within the function call. Also, we loop over the indices of the nodes in the tree, i.e. use `sapply(1:xmlSize(doc), )` rather than `xmlSApply(doc, )` simply because we need to know which row to put the results for each node.

There are various other ways to process these two XML files. One is to use handler functions to process the internal nodes as they are being converted from C-level data structures to R objects in a call to `xml-TreeParse()`. This avoids multiple traversal of the tree but can seem a little indirect until you get the hang of it. And some transformations can be cumbersome using this approach as it is a bottom up transformation.

The event-driven parsing provided by `xmlEventParse()` is a SAX style approach. This is quite low level and used when reading the entire XML document into memory and then processing it is prohibitive, i.e. when the XML file is very, very large.

The use of XPath to perform queries and get subsets of nodes involves a) learning XPath and b) potentially multiple passes over the tree. If one has to do many queries, this can be slow overall eventhough each is very fast. However, if you know XPath or are happy to learn the basics, this can be quite convenient, avoiding having to write recursive functions to search for the nodes of interests. Using the internal nodes (as you must for XPath) also gives you the ability to go up the tree, i.e. find parent, ancestor and sibling nodes, and not just down to children. So we have more flexibility in how we traverse the tree.