A Short Introduction to the *XML* package for R

Duncan Temple Lang, UC Davis

This is intended to be a short document that gets you started with the R package XML. There are two main things that one does with the XML package: read XML documents and create XML documents. We'll start with the first of these.

Reading XML

To parse an XML document, you can use *xmlInternalTreeParse*() or *xmlTreeParse*() (with *useInternalNodes* specified as **TRUE** or **FALSE**) or *xmlEventParse*() . If you are dealing with HTML content which is frequently malformed (i.e. nodes not terminated, attributes not quoted, etc.), you can use *htmlTreeParse*() . You can give these functions the name of a file, a URL (HTTP or FTP) or XML text that you have previously created or read from a file.

Tree/DOM-based parsing

If you are working with small to moderately sized XML files, it is easiest to use *xmlInternal-TreeParse*() to first read the XML tree into memory.

```
#"http://www.omegahat.org/RSXML/index.html"
doc = xmlInternalTreeParse("Install/Web/index.html.in")
```

Then you can traverse the tree looking for the information you want and putting it into different forms. There are two ways to do this iteration. One is to recursively "walk" the tree yourself by starting at the root node and processing it, and then process each child node in the same manner, working on its name and attributes and then its children, and so on. In order to collect the information at different levels into an data structure, it is often convenient to use global variables. This works for interactive computations. When writing functions for this, make certain to use closures/lexical scoping to obtain non-local variables that are not truly global.

Many people find recursion confusing, and when coupled with the need for non-local variables and mutable state, a different approach can be welcome. If we know what parts of the tree that we are interested in, then it is convenient to just fetch them, process them and discard them to move on to the other pieces. XPath is an XML technology that provides a language for accessing subsets of an XML tree. It allows us to express things such as "find me all nodes named a" or "find me all nodes name a that have no attribute named b" or "nodes a that have an attribute b equal to 'bob'" or "find me all nodes a which have c as an ancestor node". It has a similar feeling to R's subsetting capabilities and works for trees rather than vectors and data frames. It is also very powerful and efficient. But it takes a little time to learn. Some decent tutorials are available on the Web (e.g. Zvon [http://www.zvon.org/xxl/XPathTutorial/General/examples.html] and w3schools [http://www.w3schools.com/xpath]) and there are books that cover this subject, e.g. [XML in a Nutshell], [XPathXPointer].

The XPath functions in the *XML* package are *getNodeSet*() and *xpathApply*() . Basically, you specify the document returned from *xmlInternalTreeParse*() and the XPath expression to identify the nodes. *getNodeSet*() returns a list of the matching nodes. *xpathApply*() is used to apply a function to each

of those nodes, e.g. find nodes named "a anywhere in the tree that have an "href" attribute and get the value of that attribute

```
src = xpathApply(doc, "//a[@href]", xmlGetAttr, "href")
```

Of course, once we have the nodes of interest, we need to be able to extract their information. There are several functions to do this: *xmlName*() , *xmlAttrs*() , *xmlGetAttr*() , *xmlChildren*() and *xmlValue*() . *xmlName*() gets the name of the node/element. *xmlAttrs*() returns all the attribute name-value pairs as a character vector while *xmlGetAttr*() is used to query the value of a single attribute with facilities for providing a default value if it is not present and converting it if it is. We tend to use *xmlGetAttr*() as we typically know which attributes we are looking for. *xmlAttrs*() is used when doing general/meta- computations.

That's essentially all the information that is available directly from the node. Other information is available from the child nodes. If you are dealing with a "simple" node that contains no XML child nodes but simply text, e.g.

```
<emphasis>text to be emphasized</emphasis>
```

then the text is actually a child node. We can deal with it in the way we deal with arbitrary children nodes, but the function *xmlValue*() is convenient for retrieving the text value of a node. So we could get the string "text to be emphasized" via the call

```
xmlValue(node)
```

assuming **node** referred to the node. *xmlValue*() works on arbitrary nodes, not just simple text nodes and operates recursively.

The child nodes are accessed by *xmlChildren*() and each of these is also a node and so amenable to *xmlName*() , *xmlAttrs*() and *xmlGetAttr*() . *xmlChildren*() gives you a regular R list containing all of the child nodes. You can then access individual elements or subsets of these using regular R subscripting. For example, suppose we have a node with name "A" and it has children with node names "X", "Y" and "Z", and "X", "Y" and "Z", i.e.

```
<A>
  <X/>
  <Y/>
  <Z/>
  <X/>
  <Y/>
</A>
```

Then we can get the first or the last two children with

```
xmlChildren(node)[[1]]
xmlChildren(node)[2:3]
```

We can determine how many children a node has with

```
length(xmlChildren(node))
```

or

```
xmlSize(node)
```

You can also use names corresponding to the node names. Then we could get all the nodes named "Y" and "Z" with

```
xmlChildren(node)[c("Y", "Z")]
```

You can also index the children directly without having to use *xmlChildren*() to get the list first. For example, we can do the subsetting above more conveniently as

```
node[1:3]
```

Similarly, we can use names directly

```
node[c("Y", "Z")]
```

We frequently apply the same operation on all the children, for example, get their class or get an attribute of each. We can do this as

```
sapply(xmlChildren(node), xmlGetAttr, "id")
```

but again, we can do it more tersely with either of the functions *xmlApply*() and *xmlSApply*() . So the above becomes

```
xmlSApply(node, xmlGetAttr, "id")
```

If you us *xmlInternalTreeParse*() (or `xmlTreeParse(.., useInternalNodes = TRUE)`), you will end up with "internal" nodes that are references to the C data structures representing nodes. Otherwise, you will end up with XML nodes represented as lists of lists in R. With the internal nodes, you can "walk" the tree by going up and sideways, not just down through the children. The function *xmlParent*() gets the parent node of an XML node, or returns **NULL**. You can use this to iteratively walk to the top of the tree

```
while(!is.null(node)) {
   node = xmlParent(node)
 }
```

Given a node, we can also use *getSibling*() to move sideways. This gets the next sibling to right or left of a particular node in the list if children.

SAX & Event-driven parsing

If you have a very large XML file, you probably want to use the *xmlEventParse*() function to parse the file. This is quite low-level and you have to provide functions that are invoked when the parser encounters events within the XML stream such as the start of a node, the end of a node, a text chunk, a processing instruction, and so on. There is no tree so you can't find the children of a node directly but your code has to remember where it is based on the open and close node event so that one can understand the hierarchy. This is a state machine and a quite different style of programming than that involved in pulling information out of a tree.

If you are lucky enough to be interested in reasonably-sized subsets of the tree, then you can use "branches" to make things a little simpler. Otherwise, you have to define handler functions for processing start and end of nodes, and maintain the state of where the parser is to make sense of the information. This is the most efficient way to read an XML file, but is not the simplest. So we tend to try to work with *xmlTreeParse*() unless we know that we have to deal with large data files.

SAX is very memory efficient as it doesn't build the tree. However, for quick results, you can try use *xmlInternalTreeParse*() and XPath queries to get results even on very large files. If the tree can be read into memory, it can be queried efficiently. So it is always worth a try.

Creating XML

We often want to generate XML. For example, we want to create an HTML document to view in a browser. Or we want to generate input for Google Earth to display. Or we want to create XML nodes for dynamic documents. Again, the XML package provides several different ways to go about doing this. We'll focus on using internal nodes directly. There are higher-level functions to aid in this also, and alternative representations using R-level objects rather than C objects.

To create a regular node, we use *newXMLNode*() . This takes the name of the XML element/node, e.g. "img" for an image in HTML. Attributes are given by the *attrs* argument. And children can be added via the *...* mechanism. So for example, we can create the tree we discussed 'simple tree' above

make-nodes
```
node = newXMLNode("A")
sapply(c("X", "Y", "Z", "X", "Y"),
        newXMLNode, parent = node)
cat(saveXML(node))
```

W can change a odes attributes using *xmlAttrs*() as in

```
xmlAttrs(node)["src"]  = "http://www.omegahat.org"
```

Further Topics

We haven't mentioned name spaces, DTDs, schema, XSL or any of the advanced aspects of XPath.

Bibliography

[XML in a Nutshell] *XML in a Nutshell*. A Desktop Quick Reference. O'Reilly & Associates, Inc.. Elliotte Rusty Harold. W. Scott Means. third. 2004.

[XPathXPointer] *XPath and XPointer*. O'Reilly & Associates, Inc.. John E. Simpson.