Abstract

This provides a brief tour of how to use the XML parsing package. It starts by showing how to read an XML document into R and access the elements of the tree that represents the structured contents of the document. Next, it discusses how to govern the creation of the tree as it is being read from the file. And finally, it discusses the use of event-driven (or SAX) parsing.

Suppose we have the following text in a file named `as.Sxml`. A quick look at the contents shows that it contains some S-language functions and some documentation for each of them. This is similar to the self-documenting facility in S4.

Note also that each function definition consists of 3 elements: a name, some comments and the function definition itself. In this particular form, we have been able to specify the structure of the elements and not use the R or S syntax to assign the function definition to the name, i.e. as <- function(..) .... Why is this useful? Because we have expressed what mean – the relationships between the elements in a system-neutral format. I could add a fourth element to some or all of these functions which provides a Matlab implementation of the same function. The structure and syntax of the document would not have to change. Instead, the software that reads the document would know which bits to read.

Note also that the S syntax causes complications for XML because certain characters S uses (e.g. <) have special significance in XML. To avoid processing these S language elements as XML, one must escape them. This is why the function definition (i.e. the right hand side) is enclosed within a <![CDATA[...]]> construction. This indicates the XML parser that the text within the [ ] pair is to be read verbatim.

Note also that this is not needed in place. For example, the R *processing instruction* (<?R ..> near the end of the document) does not require escaping the R command(s). This is because XML parsers recognize this tag type as being special and escape its contents automatically.

We are definitely giving up some readability in this format. For this to work, we must provide good tools that are easy to use to generate and work with these types of documents.

?? ⟨ ??⟩≡

```
<?xml version="1.0"?>
<!DOCTYPE functions>
<functions>
<functionDef access="protected">
<name>as</name>
<comments>
 return a the version of this object coerced to be the given Class

 If the corresponding ‘is’ relation is true, it will be used.  In particular,
 if the relation has a coerce method, the method will be invoked on ‘object’.

 If the ‘is’ relation is FALSE, and the coerceFlag is TRUE,
 the coerce function will be called (which will throw an error if there is
 no valid way to coerce the two objects).  Otherwise, NULL is returned.
</comments>
<def>
<![CDATA[
  function(object, Class, coerceFlag = T) {
    thisClass <- Class(object)
    if(thisClass == Class)
      return(object)
    if(is(object, Class)) {
      ## look for coerce method or indirection
      thisClass <- Class(object)
      coe <- extendsCoerce(thisClass, Class)
      coe(object)
    }
    else if(coerceFlag)
```

```
      coerce(object, new(Class, force=T))
    else
      NULL
  }
]]>
</def>
</functionDef>
<functionDef>
<name>extendsCoerce</name>
<comments>
  the function to perform coercion based on the is relation
  between two classes.  May be explicitly stored in the metadata or
  inferred.  If the latter, the inferred result is stored in the session
  metadata for fromClass, to save recomputation later.
</comments>
<def>
<![CDATA[
  function(fromClass, Class) {
    ext <- findExtends(fromClass, Class)
    f <- NULL
    if(is.list(ext)) {
      coe <- ext$coerce
      if(is.function(coe))
        return(coe)
      by <- list$by
      if(length(by) > 0)
        f <- substitute(function(object)
                        as(as(object, BY), CLASS), list(BY = by, CLASS=Class))
      ## else, drop through
    }
    if(is.null(f)) {
      ## Because 'is' was TRUE, must be a direct extension.
      ## Copy slots if the slots are a subset.  Else, just set the
      ## class.  For VIRTUAL targets, never change the object.
      virtual <- isVirtualClass(Class)
      if(virtual)
        f <- function(object)object
      else {
        fromSlots <- slotNames(fromClass)
        toSlots <-  slotNames(Class)
        sameSlots <- (length(toSlots) == 0
            || (length(fromSlots) == length(toSlots) &&
                !any(is.na(match(fromSlots, toSlots)))))
      if(sameSlots)
          f <- substitute(function(object){Class(object) <- CLASS; object},
                          list(CLASS = Class))
      else
        f <- substitute(function(object) {
          value <- new(CLASS)
          for(what in TOSLOTS)
            slot(value, what) <- slot(object, what)
          value }, list(CLASS=Class, TOSLOTS = toSlots))
      }
```

```
        ## we dropped through because there was no coerce function in the
        ## extends object.  Make one and save it back in the session metadata
        ## so no further calls will require constructing the function
        if(!is.list(ext))
          ext <- list()
        ext$coerce <- f
        ClassDef <- getClass(fromClass)
        allExt <- getExtends(ClassDef)
        allExt$Class <- ext
        setExtends(ClassDef, allExt)
      }
      f
    }
  ]]>
  </def>
  </functionDef>
  <?R   x <- 1:10?>
  </functions>
```

We parse the document and create the tree that contains the different elements within the XML document. Note that we are not interested in the DTD at this point, so we instruct the parsing function to omit translating it to S.

?? ⟨ ??⟩+≡
```
  doc <- xmlTreeParse("/tmp/as.S", getDTD = F)
```

So now we want to get at the contents of the data. We get the top node of the document using the *xmlRoot()* function.

?? ⟨ ??⟩+≡
```
  r <- xmlRoot(doc)
```

This is the node which is referenced in the `DOCTYPE` and whose name is `functions`. We can find its name using the *xmlName()* function.

?? ⟨ ??⟩+≡
```
  xmlName(r)
```

We can determine how many sub-nodes this root node has by calling the function *xmlSize()*.

?? ⟨ ??⟩+≡
```
  xmlSize(r)
```

In this case, the result is 3. That means that it has 3 children which are themselves *XMLNode* objects. We can access the different child nodes using the [[operator. For example, we can get the first child node with the command.

?? ⟨ ??⟩+≡
```
   r[[1]]
```

This is also an object of class `XMLNode`.
We can ask it for its name and number of children, i.e. size.

?? ⟨ ??⟩+≡
```
xmlName(r[[1]])
xmlSize(r[[1]])
```

Most all XML nodes have attributes corresponding to the
name="value" pairs within the tag start element. For
example, the first function definition in our example (i.e. the first
sub-child of the root node) has a access attribute with a
value protected. We can retrieve a name character vector of
a node's attributes via the function *xmlAttrs()*.

?? ⟨ ??⟩+≡
```
xmlAttrs(r)
```

Lets get the first function definition object.
This is the first child of the top-level document object.

?? ⟨ ??⟩+≡
```
r[[1]]
```

This is itself an object of class *XMLNode* and so has a tag
name, attributes and children.
It has 3 children whose tag names are
given by

?? ⟨ ??⟩+≡
```
> sapply(xmlChildren(r[[1]]), xmlName)
      name   comments        def
    "name" "comments"      "def"
```

Applying an operation to children of a node is so common that we
provide functions *xmlApply()* and *xmlSApply()* which
are simple wrappers whose primary role is to fetch the list of
children of the specified node. (The apply functions are not generic.)

?? ⟨ ??⟩+≡
```
xmlSApply(r[[1]], xmlName)

xmlApply(r[[1]], xmlAttrs)

xmlSApply(r[[1]], xmlSize)
```

Let's look further at this *functionDef* element in the tree.
As we see, it has three sub-nodes named *name*,
comments and *def*.
Let's grab the *name*
element first.

?? ⟨ ??⟩+≡
```
r[[1]][[1]]
```

Again, this is of class *XMLNode*.

?? ⟨ ??⟩+≡
```
> class(r[[1]][[1]])
[1] "XMLNode"
```

and it has a single child whose class is *XMLTextNode*. This basically means we have a leaf node. Objects of class *XMLTextNode* have no children (but they are *XMLNode* so they have a slot for children!)

?? ⟨ ??⟩+≡
```
> class(r[[1]][[1]][[1]])
[1] "XMLTextNode" "XMLNode"
```

This leaf node is not the text itself, but contains that text. We get it using the *xmlValue()* function.

?? ⟨ ??⟩+≡
```
xmlValue(r[[1]][[1]][[1]])
```

We should not that the lengthy subscripting to access nodes within nodes $...$ is ugly and tedious. Of course, one can assign these intermediate nodes to variables and work on these

?? ⟨ ??⟩+≡
```
x <- r[[1]]
x <- x[[1]]
xmlValue(x[[1]])
```

I personally find this sometimes more difficult to follow. But there are times that it is more readable. The key point to remember here is that these intermediate variables are *copies* of the element in the tree. This is obvious to users of the S language, but is slightly unexpected for those coming with backgrounds in *C/C++*, *Java*^{TM}, *Perl*, etc. These languages (can) use references to operate on XML trees and so changes to sub-nodes are reflected in the bigger tree in which that sub-node resides. S is not ideally suited to operating on highly recursive, deep objects. But it is more than sufficient.

We have seen how we can extract individual sub-nodes from an object of class *XMLNode* using the [[operator and giving its index. It will come as no surprise that we can use the [(a single bracket) operator to extract a list of nodes. For example,

?? ⟨ ??⟩+≡
```
 r[[1]][1:2]
```

returns the first two elements of the root node,
i.e. the *name* and *comments* nodes.
Similarly to using indices, one can identify
nodes by name.

?? ⟨ ??⟩+≡
```
r[[1]]["comments"]
```

?? ⟨ ??⟩+≡
```
h <- xmlTreeParse(system.file("treeParseHelp.xml", pkg="XML"))
xmlRoot(h)[c("name", "author")]
```

?? ⟨ ??⟩+≡
```
> names(xmlRoot(doc))
 [1] "name"        "title"       "description" "usage"       "arguments"
 [6] "details"     "value"       "references"  "author"      "notes"
[11] "seealso"     "examples"    "keywords"
> names(xmlRoot(doc)[["examples"]])
[1] "example" "example" "example" "example" "example"
```

We find out which examples have explicit descriptions stanzas by
looking at the number of sub-nodes they have.

?? ⟨ ??⟩+≡
```
> xmlSApply(xmlRoot(doc)[["examples"]], xmlSize)
example example example example example
      1       2       1       1       2
```

So we can get the last one and look at its description.

?? ⟨ ??⟩+≡
```
xmlRoot(doc)[["examples"]][[5]][["description"]]
```

We can take the code from the example
and execute it.

?? ⟨ ??⟩+≡
```
eg <- xmlRoot(doc)[["examples"]][[5]]
canRun <- length(xmlAttrs(eg)) == 0  | is.na(match("dontRun", names(xmlAttrs(eg))))
if(!canRun)
  canRun <- !as.logical(xmlAttrs(eg)["dontRun"])
if(canRun)
   eval(parse(text=eg[[2]]))
```

?? ⟨ ??⟩+≡

```
xmlTagByName <-
function(node, name)
{
 which <- (1:xmlSize(node))[sapply(node$children, xmlName) == name]
 node$children[which]
}
```

?? ⟨ ??⟩+≡

```
lapply(xmlTagByName(d$children[[1]],"functionDef"), function(x) x$children[[2]][[1]])
```

Reading the entire XML document into a tree and then processing this tree works well in most situations. There are cases however, where it is more convenient to process the nodes in the tree as they are being created and inserted into the tree. This allows us to modify the node or even to discard it from the tree altogether.

?? ⟨ ??⟩+≡

```
fileName <- system.file("data/mtcars.xml")
doc <- xmlTreeParse(fileName,  handlers = (function() {
                                 vars <- character(0) ;
                                list(variable=function(x, attrs) {
                                                vars ■- c(vars, xmlValue(x[[1]]));
                                                NULL},
                                     startElement=function(x,attr){
                                                   NULL
                                                 },
                                     names = function() {
                                                 vars
                                             }
                                    )
                                })()
                       )
```

Now, suppose we want to make post-processing the tree easier. We can start by providing additional class information. For example, when reading a R source code in XML format (see xml2tex.Sxml) document, we might process fragment chunks by giving them an additional class name *XMLFragmentNode*. This would then allow us to process the resulting objects in a simpler manner, dispatching to different functions.

?? ⟨ ??⟩+≡

```
 h <- list(fragment=function(x, attr){ class(x) <- c("XMLFragmentNode", class(x))
                                       x
                                     })
 doc <- xmlTreeParse(file, handlers=h, asTree=T)
```

1 Creating XML Nodes

There are two styles that can be used for creating
nodes:

a) top-down create a top-level object
and assign children to it,

b) bottom-up create child nodes and group them
together into container/parent nodes, and recursively
work ones way up the tree.

?? ⟨ ??⟩+≡
```
a <- xmlNode("arg", attrs = c(default="T"), xmlNode("name", "foo"), xmlNode("defaultValue","1:10"))

a$children[[3]] <- xmlNode("duncan")
```

The resulting tree is

?? ⟨ ??⟩+≡
```
<arg default="T">
<name>
foo
</name>
<defaultValue>
1:10
</defaultValue>
<duncan>
</duncan>
</arg>
```

The worse form of generating a is

?? ⟨ ??⟩+≡
```
a <- xmlNode("arg", attrs = c(default="T"),
               xmlNode("name", xmlTextNode("foo")), xmlNode("defaultValue",xmlTextNode("1:10")))
```

%

2 Writing XML Output

If one has a tree of *XMLNode* objects in S, then the basic
print methods for these classes will generate XML that can be put in a
file or generally used outside of S. But what about translating data
in S into XML so that it can be used elsewhere, e.g sent to Matlab,
put on the web, communicated directly to another application via SOAP,
etc. How do we go about generating the XML text to represent
an object?
Well, take a look at StatDataML