Here we describe the basics of the XML parsing facilities in theOmegahat package for R and S. There are two styles of parsing – document and event based.

Document The document approach reads an entire file into memory as a hierarchical tree (i.e. a lists of lists in R and S) of XML tags or nodes. These XML elements in the tree contain the name of the XML tag, its attributes and a list of the sub-elements.

Event The event based style invokes handlers as each XML element (start of a tag, end of a tag, comment, text tag, etc.) is encountered in the parsing stream. The document approach is usually simpler to understand and use for standard situations, while the event driven style provides greater control and the potential for more efficient handling of very large XML data sources.

We discuss R functions for the document based approach first and then outline the event-based approach. In each of the two approaches, there is a single function that one calls to perform the parsing.

As well as parsing regular XML documents, the facilities in the package allow one to read and manipulate (within R or S) DTDs – Document Type Definitions – which define or provide a general template for different groups of documents. This facility can be used to perform meta-programming on documents, both creating valid XML documents programmatically and also providing mappings between tags.

This is more of a “how to” than a “why”, or “how does it work” document.

1 Basics Idea

Each XML document is made up of XML tags organized hierarchically, where tags are nested within other tags and some tags just have text.

2 Document-based Parsing

The fuction for parsing an XML document and returning it as a list of nodes is *xmlTreeParse()*

We will consider the following Structure Vector Graphics (SVG) file (in data/svg.xml)**??** and we will setup a mechanism to process and render it.

2.1 Processing the Nodes

When we build the R document tree having generated the internal DOM version in C, we create the different nodes by first creating the default version with classes *XMLNode*, *XMLComment*, *XMLEntity*, *XMLText*, etc. However, the user can provide functions that post-process these nodes before they are added to the R document tree. To do this, one provides a named list of functions as the value of the parameter *handlers()*. The *C*-level conversion takes each XML element, converts it to an *XMLNode* (or similar class) and then calls the appropriate function from this list. These functions return either NULL indicating that the node should be discarded and not added to the document tree, or an object to be added to the tree. Currently, the only argument to these functions is the node itself. (The parent may be added in the future.) This node contains its children nodes. (If this proves to be excessively expensive as many nodes are discarded or modified to discard their child information, one should consider event driven parsing.)

How is the appropriate function selected?

A simple example of utilizing these post-processing node handlers is to discard all comment elements.

?? ⟨* **??**⟩≡ **??**▷

```
doc <- xmlTreeParse("data/mtcars.xml",
                      handlers = list(startElement =function(node) {
                       ifelse(class(node) == "XMLComment", NULL, node)
                      })
                   )
```

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg SYSTEM "SVG-19990812.dtd">
<svg width="120" height="120">
<!-- define the outside border as a black square with a smaller white square on top of it
->
<rect x="1" y="1" width="120" height="120" style="fill: black"/>
<rect x="10" y="10" width="102" height="102" style="fill: white"/>
<!-- position the "ia" near the center of the image -->
<text style="font-size: 70; font-family: serif; font-weight: bolder; color: black" x="28"
<!-- build a black triangle that covers the dot of the "i" and a black rect-
angle for the base. note that the "g" tag groups the two objects and ap-
plies the black fill to both objects at the same time -->
<g style="fill: black">
<polygon points="60 12 106 51 14 51 60 12" />
<rect x="14" y="87" width="92" height="19" />
</g>
<!-- create the white dot for the "i" -->
<ellipse cx="40" cy="44" rx="7" ry="4" style="fill:white" />
</svg>
```

Figure 1: SVG File

A slightly more advanced version uses a closure to store a list of tag names that we wish to keep in the document tree, discarding all others. (Event driven parsin would be better for this application.)

?? ⟨* ??⟩+≡ ◁?? ??▷

```
   xmlKeepTags <- function(tagNamesToKeep) {
                  startElement <- function(node) {
 #                             ifelse(!any(is.na(match(xmlName(node), tagNamesToKeep))), no
                      if(any(xmlName(node) == tagNamesToKeep)) {
                        cat("Keeping",xmlName(node),"\n")
                        return(node)
                      }
                      else {
                        cat("Discarding",xmlName(node),"\n")
                        return(NULL)
                      }
                   }
                    return(list(startElement=startElement))
                 }
   doc <- xmlTreeParse("data/mtcars.xml", xmlKeepTags(c("variables","dataset")))
```

Note that the nodes are processed upwards (i.e. from leaf to root node) rather than from the root down through the child nodes. Thus, if we discard the parent node of a node we are trying to preserve, e.g variables when trying to preserve variable, we will throw away the children nodes also and discard all the variable elements.

One can query the DTD to find what nodes allow the ones in which we are interested as sub-elements.

?? ⟨* ??⟩+≡ ◁?? ??▷

```
   dtd <- parseDTD("DatasetByRecord.dtd")
   which <- character()
   for(i in names(dtd$elements)) {if(dtdValidElement("variable", i, dtd)) which <<- c(which
```

3 Event-based Parsing

Processing each XML element as the parser occurs can be a very useful and more flexible approach than reading an entire document, maintaining an intermediate form in memory and then processing. Firstly, the amount of memory required is smaller, often significantly. In other cases, the source of the XML may not be a complete document, but may be a source that periodically generates more output. For example, one might be monitoring a device in a factory, etc. where the data is an "infinite" stream. By processing the XML units as they arrive, one can provide dynamic updating of the intermediate or current results. This approach allows users to decide whether to continue, monitor for strange events, perform quality control procedures and generally perform statistical analysis on the process, not static data. This is similar to the idea of triggers in databases. A third case where the element-wise approach works well is when one wishes to extract rows or cells that fit particular criteria. Rather than reading all the data and then processing it, one can discard those records that do not satisfy the criteria. This record-wise processing works well when a transformation of the record is required as the transformation can be done in-line before assigning the value(s) and hence avoids a copy of the data.

We will look at an example of the dynamic event-driven processing which reads a data set and keeps certain records. The first example will keep rows based on their order or index. The second example examines the contents of the record to determine whether it should be discarded or kept. These are different in that in the first, we can determine this when we handle the record tag, whereas the second case waits for the text value within the record tag and must be done differently. This uses the basic event handler in **dataFrameEvent** and provides alternative versions of the record and text functions in that closure. The record closure

?? ⟨* **??**⟩+≡ ◁**?? ??**▷

```
    record <- function(x, atts) {
     if(is.na(match(atts[["id"]], desiredRowNames))) {
        # discard this entry
       return()
     }

      processRow <<- 1
        # advance the current record index.
        # (Same as previous version).
      currentRecord <<- currentRecord + 1
      rowNames <<- c(rowNames, atts[["id"]])
    }
```

The definition of the *text()* changes so that it returns if we are expecting a record (i.e. not expecting a variable name) and processRow is .

One other small changes relate to how we set the dimensions and the row names of the resulting dataframe. Rather than using the number of records reported in the XML file, we use the length of the desired row names specified when creating the closure. This can be handled more dynamically if we cannot assume uniqueness, etc.

3.1 Filtering on a Record's Values

We now make the filtering slightly more complicated. We will create an event filter to which the user supplies a function expecting the record as its only argument and returning a logical value indicating whether the record should be accepted or not. The argument is a named list of values.

```
function(data) {
  as.numeric(data["cyl"]) >= 6 & as.integer(data[2]) < 100
}
```

Here, we change the logic slightly from the way we read the entire dataframe. Firstly, we do not want to allocate a matrix or data frame to store the number of records that the dataset tag indicates. We are trying to be conservative in the amount of memory we use. So, instead, we append each record that we accept to a list and at the conclusion of the XML stream, we convert the list of records to a dataframe. This involves changing the segment in the *text()* function

?? ⟨* ??⟩+≡ ◁?? ??▷

```
        for(i in els) {
          data[currentRecord, currentColumn] <<- as.numeric(i)
          currentColumn <<- currentColumn + 1
        }
```

to read

?? ⟨* ??⟩+≡ ◁?? ??▷

```
       data[[length(data)+1]] <<- els
```

Another change is how we handle the record id attribute. We can discard the current record count (currentRecord) and change the definition of the record handler to store the record id. The text handler can then access this if it accepts the record, and append it to the rowNames vector.

?? ⟨* ??⟩+≡ ◁?? ??▷

```
        names(els) <- varNames
        if(accept(els)) {
          data[[length(data)+1]] <<- els
          rowNames <<- c(rowNames, currentRowName)
        }
```

And finally, the *endElement()* function in the closure is changed to convert the list of records stored in data to a data frame.

?? ⟨* ??⟩+≡ ◁?? ??▷

```
  if(x == "dataset") {
    data <- data.frame(matrix(unlist(data),length(data),length(varNames), byrow=T))
    names(data) <- varNames
    rownames(data) <- rowNames
   }
  After all this, we can use the filter
```

?? ⟨* ??⟩+≡ ◁??

```
   accept  <-  function(data) {
                   as.numeric(data["cyl"]) >= 6 & as.integer(data[2]) < 100
               }
   myData <- xmlEventParse("data/mtcars.xml", valueDataFrameFilter(accept))$data()
```