


```

<?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" y="70">ia</text>
<!-- build a black triangle that covers the dot of the "i" and a black rectangle for the base. note that the "g" tag groups the two objects and applies 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))), node, NULL)
    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, i)}

```

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 **F**.

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()
```