XML is very convenient for a variety of different applications. It is language neutral and provides a form of persistence. It can contain directives (*processing instructions*) for different systems to aid interpretation of a document. It is a convenient form for specifying and storing configuration files. This is because it provides a richer form than simple ASCII file, havin significant semantic information, but simpler than most languages.

We need the facilities for reading XML in both R and S. Ideally they should be similar. There are a myriad of good XML parsers written in *C*, *C++*, *Java*^{TM}, etc. We can use these to provide *R*-level user programming API for reading XML.

The reason for wanting user-level programming capabilities are reasonably obvious. Firstly, the focus of these languages is high-level operations and for this reason they are usually easier to use. Secondly, the mechanics of parsing and reading an XML document are reasonably unimportant. What is more complex is the conversion from the XML form to a user-level datastructure. While XML documentss contain more information than simple ASCII files, they do not necessarily contain information about what type to map them to in each target system and how to do so. The example below of a job has no counterpart in R and so the user must decide how to arrange the resulting data read from the XML file.

As a first start, we use the expat parser. The Gnome-related parser didn't compile for me and that makes me hesitant. Additionally, it is heavily DOM oriented which requires the entire tree to be in memory. We may have more constraints for large datasets, etc.. It is a validating parser however.

The basic approach is reasonable simple. As elements/nodes in the document are located, the parser notifies our routines. These in turn call *R* functions associated with that node type (e.g element, character data, entity, etc.) and potentially even the element name (e.g. body, h1, h2, etc.). The user initiates the parsing by specifying the name of the "file" or stream and a collection of functions which are used for the different "events". The collection of functions can also refer to local data by using a closure.

The following is a simple example of an XML document taken from the GNOME xml distribution. We use it to illustrate how we might generate an R structure from it.

2a ⟨*job.xml* 2a⟩≡

```
<?xml version="1.0"?>
<gjob:Helping xmlns:gjob="http://www.gnome.org/some-location">
  <gjob:Jobs>

    <gjob:Job>
      <gjob:Project ID="3"/>
      <gjob:Application>GBackup</gjob:Application>
      <gjob:Category>Development</gjob:Category>

      <gjob:Update>
        <gjob:Status>Open</gjob:Status>
        <gjob:Modified>Mon, 07 Jun 1999 20:27:45 -0400 MET DST</gjob:Modified>
        <gjob:Salary>USD 0.00</gjob:Salary>
      </gjob:Update>

      <gjob:Developers>
        <gjob:Developer>
        </gjob:Developer>
      </gjob:Developers>

      <gjob:Contact>
        <gjob:Person>Nathan Clemons</gjob:Person>
        <gjob:Email>nathan@windsofstorm.net</gjob:Email>
        <gjob:Company>
        </gjob:Company>
        <gjob:Organisation>
        </gjob:Organisation>
        <gjob:Webpage>
        </gjob:Webpage>
        <gjob:Snailmail>
        </gjob:Snailmail>
        <gjob:Phone>
        </gjob:Phone>
      </gjob:Contact>

      <gjob:Requirements>
      The program should be released as free software, under the GPL.
      </gjob:Requirements>

    </gjob:Job>

  </gjob:Jobs>
</gjob:Helping>
```

Basically, as we start each tag, we call the appropriate function and give it a string name and a named list of XML attributes.

2b ⟨ 2b⟩≡

The parse function looks something like the following. The *handlers* object is assumed to be a closure. Its functions modify its own data internally and then we return it so that one can access the new data contents.

The addContext argument instructs the internal *C* code whether to build the index path of the position in the XML tree for this node. In the absence of references, it can be used to identify sub elements of a tree.

The useTagName argument allows the caller to specify whether there are functions per element name or to use the standard event driven interface which is startElement, endElement, externalEntity, etc.

startElement
endElement
text
externalEntity
startCdata
endCdata
comment
default
startNameSpace
endNameSpace

Table 1: Standard Event Handlers

⟨*XML.R* ⟩≡

```
xmlEventParse <- function(file, handlers=xmlHandler(), addContext = T, useTagName = F) {
 handlers <- .Call("R_XMLParse", file, handlers)
 return(handlers)
}
```

In our simple setup, we will provide a single handler for the start of a new element.

4a ⟨*Example.r* 4a⟩≡

```
xmlHandler <- function() {
  data <- list()
  startElement <- function(name, atts,...) {
    if(is.null(atts))
      atts <- list()
    data[[name]] ■- atts
  }
  text <- function(x,...) {
    cat("MyText:",x,"\n")
  }
  comment <- function(x,...) {
    cat("comment", x,"\n")
  }
  externalEntity <- function(ctxt, baseURI, sysId, publicId,...) {
    cat("externalEntity", ctxt, baseURI, sysId, publicId,"\n")
  }
  entityDeclaration <- function(name, baseURI, sysId, publicId,notation,...) {
    cat("externalEntity", name, baseURI, sysId, publicId, notation,"\n")
  }

  foo <- function(x,attrs,...) { cat("In foo\n")}
  return(list(startElement=startElement, getData=function() {data},
              comment=comment, externalEntity=externalEntity,
               entityDeclaration=entityDeclaration,
               text=text, foo=foo))
}
```

4b ⟨ 2b⟩+≡

```
h <- xmlParse("Docs/test.xml")
```

An entirely different approach involves reading the entire document into memory and then

4c ⟨ 2b⟩+≡

```
characterOnlyHandler <- function() {
  txt <- NULL
  text <- function(val,...) {
   i
    txt ■- c(txt, val)
  }

  getText <- function() { txt }

  return(list(text=text, getText=getText))
}
```

⟨ 2b⟩
⟨*Example.r* 4a⟩
⟨*XML.R*
⟩
⟨*job.xml* 2a⟩