

XML GENERATION

1

XML Overview	2
XML and SPXML Library Overview	3
The SPXML Library	4
Reading and Writing XML Using The SPXML Library	5
Examples of XSL Transformations	6
Example 1: Creating a Vector	6
Example 2: Creating a Named Vector	8
Example 3: A List of Data Frames	10
Example 4: Import SAS XML as a Data Frame	13

XML OVERVIEW

Extensible Markup Language (XML) provides a powerful mechanism for both storing information and exchanging information between applications. S-PLUS includes two libraries of functions for working with XML: the SPXML library from Insightful, and the XML library from Bell Laboratories.

The effective use of XML involves more than just the XML code, however; an application using XML requires definitions for writing and reading XML. XML uses a *Document Type Definition* (DTD) file to define the elements and attributes allowed in an XML document. DTD files create specific markup languages that can be used for specific tasks. For example, *Predictive Modeling Markup Language* (PMML) has become an industry-standard for describing statistical models and serves as a way to exchange model information between applications. Once the language is defined, an XML parser is able to read the XML text file and interpret the tags to determine the type of data that is present. The data can then be put into an internal data structure.

One of the chief benefits of XML is that it provides a way to exchange data between applications whose internal data representations are different. The *Extensible Stylesheet Language* (XSL) provides standards and tools describing how to create other documents from an XML file. *XSL transforms* (XSLT) can be used to create an HTML document or to transform XML to another style of XML that uses different tags and organizes the data differently. *XSL formatting objects* (XSL-FO) describe how to create formatted documents such as PostScript, PDF, and RTF documents based on formatting specified in the XSL file and information specified in the XML file.

In the next section, we describe the differences between the XML and SPXML libraries. The remainder of the document provides more details about the SPXML library and shows examples of using its functions for data exchange.

XML AND SPXML LIBRARY OVERVIEW

As a convenience for S-PLUS users, the S-PLUS installation includes a copy of the XML library. This library is written and maintained by Duncan Temple Lang of Bell Laboratories.

The SPXML library supports reading and writing of XML by specifying a set of XML tags describing XML objects, and using C code that specifically writes and parses these tags. There is no such thing as an XML object within S-PLUS, and no manipulation of XML within S-PLUS. In this library, XML manipulation is done using XSL.

The XML library takes a different approach: It introduces a set of S-PLUS classes representing XML objects such as `XMLNode` and `XMLComment`. A *Document Object Model* (DOM) parser is used to create an S-PLUS object from the information in an XML file. This object can be traversed using S-PLUS subscripting, manipulated using S-PLUS functions, and written back to a file if desired.

Information on this library is available from

<http://www.omegahat.org/RFXML/>

Which library is most appropriate to use depends on the purpose and user preferences:

- The SPXML library is more efficient for serialization, as it uses a SAX parser and does computations almost exclusively in C code.
- When translating between S-PLUS objects and XML from other applications, the SPXML library does the transformations using XSL. This is preferable for users familiar with XSL.
- When translating between S-PLUS objects and XML from other applications, the XML library builds an S-PLUS object representing the XML document. S-PLUS functions are used to create a standard S-PLUS object such as a data frame from the XML document object. S-PLUS programmers may prefer to use S-PLUS functions rather than XSL.

The remainder of this paper focuses on the SPXML library. Consult the URL above for more information regarding the XML library.

THE SPXML LIBRARY

The SPXML library contains a small number of powerful functions:

- S-PLUS objects can be written to a file as XML using `createXMLFile` and read from a file using `parseXMLFile`.
- XML files can be transformed with XSLT or XSL-FO using `javaXMLTransform`.

These functions are all you need to create a file with an XML description of an S-PLUS object and generate a report from the information in this XML file. The formatting information is specified in an XSL file.

One reason for keeping the S-PLUS functions simple and focused on XSL as a transformation mechanism is that XSL is an industry standard with a wealth of documentation on its usage. This provides you with extensive reference materials that are not included as part of the S-PLUS documentation set.

In addition to the functions listed above, the SPXML library provides some useful utility functions:

- The functions `xml2html`, `xml2pdf`, `xml2ps`, `xml2rtf`, and `xml2xml` are wrappers for `javaXMLTransform` that specify the corresponding output file type.
- The `createXMLString` function returns the generated XML as a string rather than writing it to a file.
- The `summaryReport` function implements a particular kind of summary report. This function, along with the corresponding XSL files, are good examples of how to combine S-PLUS and XSL to create a sophisticated report.

This document focuses on using XML as a tool for data exchange. The `summaryReport` function and report generation in general are discussed in the separate technical paper titled *XML Reporting*.

READING AND WRITING XML USING THE SPXML LIBRARY

The `createXMLFile` and `parseXMLFile` functions provide a simple mechanism for S-PLUS objects to be saved and restored using XML.

In the simplest case, this allows you to save any S-PLUS object to a readable file and access it at a future time.

The following shows a simple example:

```
> library(SPXML)
> xmlFile <- "output.xml"
> orig.list <- list(fuel.frame, c(1:50), hist)
> createXMLFile(orig.list, xmlFile)
> new.list <- parseXMLFile(xmlFile)
> all.equal(orig.list, new.list)
[1] T
```

The XML tags for S-PLUS objects are described in the **SPLUS_1_0.dtd** file in the **library/SPXML/xml** directory of your S-PLUS installation.

The objective in defining these tags is to allow reading and writing of all S-PLUS objects (data frames, matrices, vectors, and multidimensional arrays). The XML descriptions of these objects are rectangular and hyper-rectangular data structures which can be easily interpreted.

EXAMPLES OF XSL TRANSFORMATIONS

To import XML from another application into S-PLUS, you need to transform the application's XML into XML describing S-PLUS objects. You can do this easily through the use of an XSL file, and in this section, we use four examples to illustrate the use of XSL.

In these examples, we start with PMML files generated by Insightful Miner, a data mining and statistical data analysis product designed to handle large amounts of data. It stores information on fitted models in its own XML format and generates industry-standard PMML, describing the models as a way to exchange model information with other applications. In addition to standard PMML element tags, Insightful Miner uses the Extension mechanism to include additional information about the models.

One type of output that Insightful Miner computes for models is a column importance measure for each independent column used in the model. By reading this information into S-PLUS, we can use S-PLUS functions to create plots and reports that are not available in Insightful Miner.

Example 1: Creating a Vector

The first PMML file we use describes a logistic regression model. The file is located in the `library/SPXML/examples/xml_generation` directory and is named `logRegPMML.xml`. The column importance measure used for logistic regression is the Wald statistic.

PMML

In the XML for the logistic regression model, the column importance information is stored in one place:

```
<Extension extender="Insightful"
  name="X-IMML-GeneralRegressionModel-Importance">
  <X-IMML-GeneralRegressionModel-Importance count="29"
    targetCategory="credit.card.owner">
    <X-IMML-GeneralRegressionModel-Effect name="mean.check.cash.withdr"
      value="169.99539009465" df="1" Pr="0"/>
    <X-IMML-GeneralRegressionModel-Effect name="mean.check.debits"
      value="165.004662158705" df="1" Pr="0"/>
  ...
```

XSL

To create a vector of column importances in S-PLUS, the first step is to design an XSL transformation file that extracts the desired information from the PMML file, and writes it to a new XML file using S-PLUS XML tags.

The file **library/SPXML/examples/xml_generation/logReg_ColImp_Vec.xml** contains the following XSL to create an S-PLUS vector of column importance values:

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml" standalone="yes" indent="yes"/>
  <xsl:template match="PMML">
    <xsl:apply-templates select="GeneralRegressionModel/Extension/
      X-IMML-GeneralRegressionModel-Importance"/>
  </xsl:template>

  <xsl:template match="X-IMML-GeneralRegressionModel-Importance">
    <xsl:element name="S-PLUS">
      <xsl:element name="Vector">
        <xsl:attribute name="length"><xsl:value-of select="@count"/></xsl:attribute>
        <xsl:attribute name="type">numeric</xsl:attribute>
        <xsl:element name="Items">
          <xsl:for-each select="X-IMML-GeneralRegressionModel-Effect">
            <xsl:element name="Item">
              <xsl:value-of select="@value"/>
            </xsl:element>
          </xsl:for-each>
        </xsl:element>
      </xsl:element>
    </xsl:element>
  </xsl:template>
</xsl:stylesheet>
```

S-PLUS Functions

To create the S-PLUS vector, first use `xml2xml` to transform the file, and then use `parseXMLFile` to create an S-PLUS object:

```
> library(SPXML)
> logRegPMMLFile <- paste(getenv("SHOME"),
+   "/library/SPXML/examples/xml_generation
+   /logRegPMML.xml", sep="")
> logRegXSLFile <- paste(getenv("SHOME"),
+   "/library/SPXML/examples/xml_generation
+   /logReg_ColImp_Vec.xml", sep="")
> splusVecXMLFile <- "Splus_ColImp_Vec.xml"
> xml2xml(logRegPMMLFile, splusVecXMLFile, logRegXSLFile)
> colImpVec <- parseXMLFile(splusVecXMLFile)
> colImpVec

[1] 169.995390095 165.004662159 109.405532767 97.18724276
[5] 67.254514683 31.503267187 20.055698417 14.51615874
[9] 9.359599672 8.369635041 7.632052511 4.176602535
...

```

Example 2: Creating a Named Vector

While it is useful to have the column importance values in S-PLUS, it would be even more practical to include the column names. With more detailed XSL, we can create a named vector with the column names and importance values.

XSL

The `library/SPXML/examples/xml_generation/logReg_ColImp_NamedVec.xml` file contains XSL to create a named vector from the logistic regression PMML. The resulting S-PLUS object is represented in the XML as a Generic object with class named.

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml" standalone="yes" indent="yes"/>
  <xsl:template match="PMML">
    <xsl:apply-templates select="GeneralRegressionModel/Extension/
      X-IMML-GeneralRegressionModel-Importance"/>
  </xsl:template>

  <xsl:template match="X-IMML-GeneralRegressionModel-Importance">
    <xsl:element name="S-PLUS">
      <xsl:element name="Generic">
        <xsl:attribute name="class">named</xsl:attribute>
        <xsl:element name="Attrs">
          <xsl:attribute name="length">1</xsl:attribute>
          <xsl:element name="Attr">
            <xsl:attribute name="name">.Names</xsl:attribute>
            <xsl:element name="Vector">
              <xsl:attribute name="length">
                <xsl:value-of select="@count"/></xsl:attribute>
              <xsl:attribute name="type">character</xsl:attribute>
              <xsl:attribute name="name">.Names</xsl:attribute>
              <xsl:element name="Items">
                <xsl:for-each select=
                  "X-IMML-GeneralRegressionModel-Effect">
                  <xsl:element name="Item">
                    <xsl:value-of select="@name"/>
                  </xsl:element>
                </xsl:for-each>
              </xsl:element>
            </xsl:element>
          </xsl:element>
        </xsl:element>
      </xsl:element>
      <xsl:element name="Vector">
        <xsl:attribute name="length">
          <xsl:value-of select="@count"/></xsl:attribute>
        <xsl:attribute name="type">numeric</xsl:attribute>
        <xsl:element name="Items">
          <xsl:for-each select=
            "X-IMML-GeneralRegressionModel-Effect">
            <xsl:element name="Item">
              <xsl:value-of select="@value"/>
            </xsl:element>
          </xsl:for-each>
        </xsl:element>
      </xsl:element>
    </xsl:element>
  </xsl:template>
</xsl:stylesheet>
```

```
</xsl:element>  
</xsl:template>
```

S-PLUS Functions

Again, use `xml2xml` to translate the XSL and `parseXMLFile` to create the object:

```
> logRegXSLFile2 <- paste(getenv("SHOME"),  
+   "/library/SPXML/examples/xml_generation  
+   /logReg_ColImp_NamedVec.xsl", sep="")  
> splusNamedVecXMLFile <- "Splus_ColImp_NamedVec.xml"  
> xml2xml(logRegPMMLFile, splusNamedVecXMLFile,  
+   logRegXSLFile2)  
> colImpNamedVec <- parseXMLFile(splusNamedVecXMLFile)  
> colImpNamedVec
```

```
mean.check.cash.withdr mean.check.debits cust.age  
169.9954 165.0047 109.4055
```

```
mean.saving.balance mean.amnt.atm.withdr  
97.18724 67.25451
```

```
...
```

Example 3: A List of Data Frames

In the logistic regression PMML, the column importance is already computed for each column and is stored in a single place in the XML. This makes it easy to access and transform.

Insightful Miner also has an *ensemble tree* model. In this model, multiple classification (or regression) trees are fit, and the results averaged together for prediction. This is similar to the use of multiple models in bagging or boosting.

With the ensemble tree, the column importance measure is determined by taking the change in goodness-of-fit at each split, and attributing that change as being due to the independent column used for the split. These changes in goodness-of-fit are summed over all splits and all trees to get a single change in goodness-of-fit attributed to each independent column. This is used as the importance measure.

Unlike logistic regression, this column importance is not a quantity that is stored in the PMML. Instead, it is computed by Insightful Miner from information stored at the split level of the XML tree description. For example, the `library/SPXML/examples/xml_generation/ensembleTreePMML.xml` file has the following:

```
<Node score="0" recordCount="6806">
  <Extension extender="Insightful" name="X-IMML-XTProps">
    <X-IMML-XTProps>
      <X-IMML-Property name="id" value="1"/>
      <X-IMML-Property name="group" value="0"/>
      <X-IMML-Property name="deviance"
        value="5750.79288765702"/>
      <X-IMML-Property name="risk" value="1020"/>
      <X-IMML-Property name="yprob"
        value="0.850132236262122 0.149867763737878"/>
      <X-IMML-Property name="improvement"
        value="435.586807320647"/>
    </X-IMML-XTProps>
  </Extension>
  <SimplePredicate field="mean.salary.deposits"
    operator="lessThan" value="4426.431109375"/>

```

In this example, we create an S-PLUS list of data frame objects, with one data frame for each tree in the ensemble. The data frame has two columns: one for the name of the independent column used to split, and one for the change in goodness-of-fit. Once we have this information in S-PLUS, we could use other S-PLUS functions to create separate column importance vectors for each tree, a single column importance vector over all trees, or some other quantity.

XSL

The file `library/SPXML/examples/xml_generation/tree_ColImp_DFList.xml` contains the following XSL transform:

```

<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml" standalone="yes" indent="yes"/>
  <xsl:template match="PMML">
    <xsl:element name="S-PLUS">
      <xsl:element name="List">
        <xsl:attribute name="length"><xsl:value-of select="
          count(../TreeModel)"/></xsl:attribute>
        <xsl:attribute name="named">F</xsl:attribute>
        <xsl:element name="Components">
          <xsl:apply-templates select="//TreeModel"/>
        </xsl:element>
      </xsl:element>
    </xsl:element>
  </xsl:template>
  <xsl:template match="TreeModel">
    <xsl:param name="numSplits" select="count(../Node[count(../True)=0])"/>
    <xsl:element name="DataFrame">
      <xsl:element name="DataFrame">
        <xsl:attribute name="numRows"><xsl:value-of select="$numSplits"/>
        </xsl:attribute>
        <xsl:attribute name="numCols">2</xsl:attribute>
        <xsl:element name="RowNames">
          <xsl:attribute name="length"><xsl:value-of select="$numSplits"/>
          </xsl:attribute>
          <xsl:element name="Items">
            <xsl:for-each select="//Node[count(../True)=0]">
              <xsl:element name="Item">
                <xsl:value-of select="position()"/>
              </xsl:element>
            </xsl:for-each>
          </xsl:element>
        </xsl:element>
        <xsl:element name="Columns">
          <xsl:element name="Column">
            <xsl:attribute name="length"><xsl:value-of
              select="$numSplits"/></xsl:attribute>
            <xsl:attribute name="type">character</xsl:attribute>
            <xsl:attribute name="name">Split.Column</xsl:attribute>
            <xsl:element name="Items">
              <xsl:for-each select="//Node[count(../True)=0]">
                <xsl:element name="Item">
                  <xsl:if test="count(SimplePredicate) &gt; 0">
                    <xsl:value-of select="SimplePredicate[1]
                      /@field"/>
                  </xsl:if>
                  <xsl:if test="count(CompoundPredicate) &gt; 0">
                    <xsl:value-of select="CompoundPredicate
                      /SimplePredicate[1]/@field"/>
                  </xsl:if>
                </xsl:element>
              </xsl:for-each>
            </xsl:element>
          </xsl:element>
          <xsl:element name="Column">
            <xsl:attribute name="length"><xsl:value-of
              select="$numSplits"/></xsl:attribute>
            <xsl:attribute name="type">numeric</xsl:attribute>
            <xsl:attribute name="name">Change.in.Fit</xsl:attribute>
            <xsl:element name="Items">

```

```

<xsl:for-each select="//Node[count(/.True)=0]">
  <xsl:element name="Item">
    <xsl:for-each select="Extension/X-IMML-XTProps
/X-IMML-Property[@name='improvement']">
      <xsl:value-of select="@value"/>
    </xsl:for-each>
  </xsl:element>
</xsl:for-each>
</xsl:element>
</xsl:element>
</xsl:element>
</xsl:element>
</xsl:element>
</xsl:template>
</xsl:stylesheet>

```

Note that a complex S-PLUS object can be created from a complex XML document in a little over a single page of XSL.

S-PLUS Functions

Once the XSL file has been created, the following code can be run from S-PLUS:

```

> ensembleTreePMMLFile <- paste(getenv("SHOME"),
+   "/library/SPXML/examples/xml_generation
+   /ensembleTreePMML.xml", sep="")
> treeXSLFile <- paste(getenv("SHOME"),
+   "/library/SPXML/examples/xml_generation
+   /tree_ColImp_DFList.xsl", sep="")
> splusDFListXMLFile <- "Splus_ColImp_DFList.xml"
> xml2xml(ensembleTreePMMLFile, splusDFListXMLFile,
+   treeXSLFile)
> colImpDFList <- parseXMLFile(splusDFListXMLFile)
> length(colImpDFList)
[1] 2
> colImpDFList[[1]][1:5,]

```

	Split.Column	Change.in.Fit
1	mean.salary.deposits	435.58681
2	mean.num.check.cash.withdr	90.42567
3	gender	110.75756
4	mean.saving.balance	46.77268
5	mean.amnt.transfers	16.38389

Example 4: Import SAS XML as a Data Frame

SAS has the capability to create data sets from XML. Here is an example of SAS-friendly XML code:

```
<?xml version="1.0" encoding="UTF-8"?>
<LIBRARY>
  <STUDENTS>
    <ID> 0755 </ID>
    <NAME> Brad Martin </NAME>
    <CITY> Huntsville </CITY>
    <STATE> Texas </STATE>
  </STUDENTS>
  <STUDENTS>
    <ID> 1522 </ID>
    <NAME> Michelle Harvell </NAME>
    <CITY> Houston </CITY>
    <STATE> Texas </STATE>
  </STUDENTS>
  <STUDENTS>
    <ID> 1523 </ID>
    <NAME> Terry Glynn </NAME>
    <CITY> Chicago </CITY>
    <STATE> Illinois </STATE>
  </STUDENTS>
  <PROFESSORS>
    <ID> 9122 </ID>
    <NAME> Sue Clayton </NAME>
    <TENURE> YES </TENURE>
    <CITY> Huntsville </CITY>
    <STATE> Texas </STATE>
  </PROFESSORS>
  <PROFESSORS>
    <ID> 9453 </ID>
    <NAME> Todd Cantrell </NAME>
    <TENURE> NO </TENURE>
    <CITY> Houston </CITY>
    <STATE> Texas </STATE>
  </PROFESSORS>
  <PROFESSORS>
    <ID> 9562 </ID>
    <NAME> Larry Anders </NAME>
    <TENURE> YES </TENURE>
    <CITY> Chicago </CITY>
    <STATE> Illinois </STATE>
  </PROFESSORS>
</LIBRARY>
```

In this example, SAS uses the second-level tags (STUDENTS and PROFESSORS) to denote individual data sets. Repeated second-level tags (note that STUDENTS is repeated three times) represent separate rows in the resulting data set. Among the repeated second-level tags, common children represent data set columns (ID, NAME, CITY and STATE in STUDENTS).

When this XML file is imported into SAS, it creates two data sets, STUDENTS and PROFESSORS, with several columns ID, NAME, TENURE (for PROFESSORS), CITY, and STATE.

XSL

To import this file into S-PLUS, we need to create an XSL file that creates XML versions of data frames (analogous to the SAS data sets). Here is an example of such an XSL file:

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml" standalone="yes" indent="yes"/>

  <xsl:template match=".">
    <xsl:variable name="root" select="." />
    <xsl:variable name="level1Children" select="/*" />
    <xsl:variable name="level1Child1" select="$level1Children[1]" />

    <xsl:variable name="children" select="$level1Child1/*" />
    <xsl:variable name="nUniqueNames">
      <xsl:call-template name="CountLevel2Nodes">
        <xsl:with-param name="level2" select="$children" />
      </xsl:call-template>
    </xsl:variable>
    </xsl:template>

    <SPLUS>
      <xsl:choose>
        <xsl:when test="$nUniqueNames > 1">
          <xsl:element name="List">
            <xsl:attribute name="length">
              <xsl:value-of select="$nUniqueNames"/>
            </xsl:attribute>
            <xsl:attribute name="named">F</xsl:attribute>

            <Components>
              <xsl:call-template name="Level2Nodes">
                <xsl:with-param name="level2" select="$children"/>
                <xsl:with-param name="needList" select="1" />
              </xsl:call-template>
            </Components>
          </xsl:element>
        </xsl:when>
        <xsl:otherwise>
          <xsl:call-template name="Level2Nodes">
            <xsl:with-param name="level2" select="$children"/>
          </xsl:call-template>
        </xsl:otherwise>
      </xsl:choose>
    </SPLUS>
  </xsl:template>

  <xsl:template name="CountLevel2Nodes">
    <xsl:param name="level2" select=""/>
    <xsl:param name="rCount" select="1"/>

    <xsl:variable name="firstChild" select="$level2[1]" />
    <xsl:variable name="child1Name" select="name($firstChild)" />
    <xsl:variable name="same" select="$level2[name(.) = $child1Name]" />
    <xsl:variable name="other" select="$level2[name(.) != $child1Name]" />

    <xsl:choose>
      <xsl:when test="boolean($other)">
        <xsl:call-template name="CountLevel2Nodes">
          <xsl:with-param name="level2" select="$other"/>
          <xsl:with-param name="rCount" select="$rCount + 1" />
        </xsl:call-template>
      </xsl:when>
    </xsl:choose>
  </xsl:template>

```

```

        <xsl:otherwise>
            <xsl:value-of select="$rCount"/>
        </xsl:otherwise>
    </xsl:choose>
</xsl:template>

<xsl:template name="Level2Nodes">
    <xsl:param name="level2" select=""/>
    <xsl:param name="needList" select=""/>

    <xsl:variable name="firstChild" select="$level2[1]"/>
    <xsl:variable name="child1Name" select="name($firstChild)" />
    <xsl:variable name="same" select="$level2[name(.) = $child1Name]" />
    <xsl:variable name="other" select="$level2[name(.) != $child1Name]" />

    <xsl:if test="boolean($same)">
        <xsl:choose>
            <xsl:when test="boolean($needList)">
                <Component>
                    <xsl:call-template name="NodesOfOneName">
                        <xsl:with-param name="nodes" select="$same"/>
                    </xsl:call-template>
                </Component>
            </xsl:when>
            <xsl:otherwise>
                <xsl:call-template name="NodesOfOneName">
                    <xsl:with-param name="nodes" select="$same"/>
                </xsl:call-template>
            </xsl:otherwise>
        </xsl:choose>
    </xsl:if>

    <xsl:if test="boolean($other)">
        <xsl:call-template name="Level2Nodes">
            <xsl:with-param name="level2" select="$other"/>
            <xsl:with-param name="needList" select="$needList"/>
        </xsl:call-template>
    </xsl:if>
</xsl:template>

<xsl:template name="NodesOfOneName">
    <xsl:param name="nodes" select=""/>

    <xsl:variable name="firstChild" select="$nodes[1]"/>
    <xsl:variable name="child1Name" select="name($firstChild)" />
    <xsl:variable name="firstChildChildren" select="$firstChild/*" />

    <xsl:variable name="nRows" select="count($nodes)"/>
    <xsl:variable name="nCols" select="count($firstChildChildren)" />

    <xsl:element name="DataFrame">
        <xsl:attribute name="numRows">
            <xsl:value-of select="$nRows"/>
        </xsl:attribute>
        <xsl:attribute name="numCols">
            <xsl:value-of select="$nCols"/>
        </xsl:attribute>

        <xsl:element name="RowNames">
            <xsl:attribute name="length">
                <xsl:value-of select="$nRows"/>
            </xsl:attribute>
            <Items>

```

```

        <xsl:for-each select="$nodes">
          <Item><xsl:value-of select="position()"/></Item>
        </xsl:for-each>
      </Items>
    </xsl:element>

    <Columns>
      <xsl:for-each select="$firstChildChildren">
        <xsl:variable name="cName" select="name(.)" />
        <xsl:element name="Column">
          <xsl:attribute name="length">
            <xsl:value-of select="$nRows"/>
          </xsl:attribute>
          <xsl:attribute name="type">character</xsl:attribute>
          <xsl:attribute name="name">
            <xsl:value-of select="$cName"/>
          </xsl:attribute>

          <Items>
            <xsl:for-each select="$nodes">
              <xsl:variable name="allColumns" select="./**"/>
              <xsl:variable name="thisColumn" select="$allColumns[name(.) = $cName]" />
              <Item>
                <xsl:choose>
                  <xsl:when test="not(boolean($thisColumn))">
                    <xsl:text></xsl:text>
                  </xsl:when>
                  <xsl:otherwise>
                    <xsl:value-of select="$thisColumn/text()"/>
                  </xsl:otherwise>
                </xsl:choose>
              </Item>
            </xsl:for-each>
          </Items>
        </xsl:element>
      </xsl:for-each>
    </Columns>
  </xsl:element>
</xsl:stylesheet>

```

S-PLUS Functions Once this XSL file has been created, we can run the following S-PLUS commands to import the SAS XML into S-PLUS:

```

> sasFile <- paste(getenv("SHOME"),
+   "/library/SPXML/examples/xml_generation/ImportSAS.xml",
+   sep="")
> xlsFile <- paste(getenv("SHOME"),
+   "/library/SPXML/examples/xml_generation/ImportSAS.xls",
+   sep="")
> xmlFile <- "ImportSPLUS.xml"
> xml2xml(sasFile, xmlFile, xlsFile)
> parseXMLFile(xmlFile)
[[1]]:
      ID          NAME          CITY          STATE
1  0755      Brad Martin  Huntsville      Texas

```

Examples of XSL Transformations

2	1522	Michelle Harvell	Houston	Texas
3	1523	Terry Glynn	Chicago	Illinois

[[2]]:

	ID	NAME	TENURE	CITY	STATE
1	9122	Sue Clayton	YES	Huntsville	Texas
2	9453	Todd Cantrell	NO	Houston	Texas
3	9562	Larry Anders	YES	Chicago	Illinois

