

Implementing a Data-Driven Courseware System with SAS®

Don Boudreaux, SAS® Institute Inc., Austin, TX

ABSTRACT

This investigation presents a courseware system that is designed to facilitate the development and rapid deployment of web-based courses. Conceptually the system requires a collection of instructional documents and an overall course schema. Specifically, the instructional documents are source HTML files that contain the course lessons. The course schema describes how the instruction lessons are organized into a class. It is defined within the hierarchical structure of an XML document. A SAS®9 program is used read the XML course schema, filter the lesson source documents into a set of final course files, and build an HTML navigation page. A number of design considerations for building courseware are presented, several SAS®9 coding tricks for parsing XML are shown, and some pros and cons for using XML are discussed.

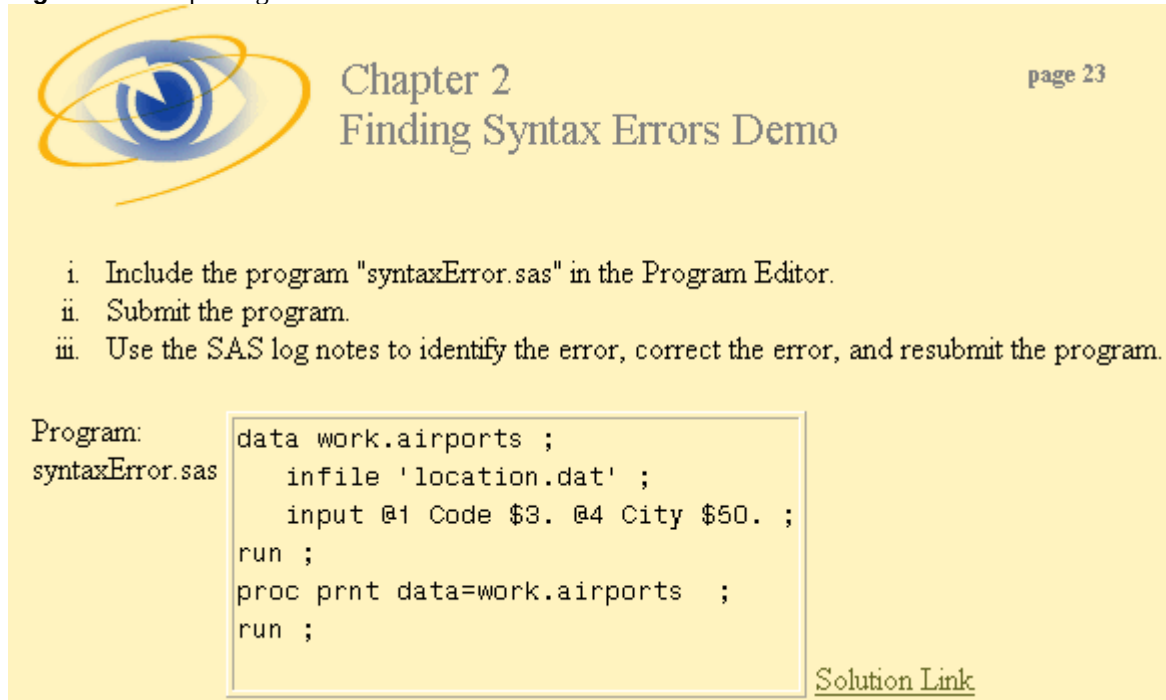
INTRODUCTION

A course can be defined as a collection of instructional documents or pages that are generally organized into a hierarchical grouping or course schema. If the intent of the course developer is to go beyond just producing simple fixed-content documents then a data-driven methodology that allows more flexibility in development and deployment should be considered. This investigation attempts to explore a SAS prototype courseware system that uses XML to create web-based courses. It is assumed that the reader is familiar with CSS specifications, HTML tags, and to some degree XML.

A DATA PERSPECTIVE

To introduce an XML data-driven view of course pages, consider the document shown in Figure 1. It presents a demo that is similar to one of the exercises in the Programming 1 class at SAS. For our purposes, we can consider it a generic course page.

Figure 1: Example Page



The screenshot shows a web page with a yellow background. At the top left is a logo consisting of a blue circle with a white arrow pointing clockwise, surrounded by two orange elliptical orbits. To the right of the logo, the text 'Chapter 2' is in a large, dark blue serif font, and 'Finding Syntax Errors Demo' is in a smaller, dark blue serif font. In the top right corner, 'page 23' is written in a small, dark blue sans-serif font. Below the title, there is a list of three instructions: 'i. Include the program "syntaxError.sas" in the Program Editor.', 'ii. Submit the program.', and 'iii. Use the SAS log notes to identify the error, correct the error, and resubmit the program.' Below the instructions, the text 'Program: syntaxError.sas' is followed by a code block containing SAS code: 'data work.airports ;', 'infile 'location.dat' ;', 'input @1 Code \$3. @4 City \$50. ;', 'run ;', 'proc prnt data=work.airports ;', and 'run ;'. At the bottom right of the code block, there is a link labeled 'Solution Link'.

To view this document from a data-driven document perspective, consider that there are different types of content on this page. This page contains organizational data. That is information about where this page exists within the overall organizational schema of the course. Specifically the organizational data elements inside of the example page include a chapter label: **Chapter 2**, a page label: **page 23**, and a page title: **Finding Syntax Errors Demo**. The instructional content of this page for the most part would be considered static text, but the code segment could be seen as dynamic data. Under different instructional scenarios it may be necessary to have different code segments used within this page. It might be important that the code reflect a specific version of a given software product. The course might be given at a site that requests use of their own data for the examples and exercises. It might be considered critical that procedures shown in the examples and exercises more closely reflect site-specific usage patterns. So, the content of this page can be thought of and separated

into static text, organizational data, and dynamic content. If this determination can be made all the pages within a course, then all of the non-static parts can be separated out as data into an XML definition file and easily manipulated.

Now, think of a hierarchical data file that contains user-defined tagged data elements. This is essentially what an XML file is. An XML file that would contain tagged values for the non-static parts described within Figure 1 would (along with other tags) look like the following:

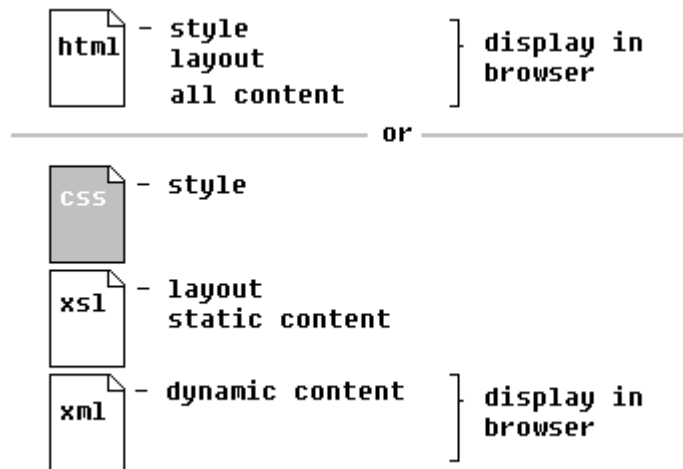
```
<?xml version="1.0"?>
<course_name> SAS programming I: Essentials </course_name>
<course_code> Prog1 </course_code>
<chapt label="Chapter 2"
      title="Getting Started with the SAS System">
  <page id="dcErr"
        label="page 23"
        title="Finding Syntax Errors Demo"
        file ="dcErr_File" />
...
</chapt>
...
<macro id="syntaxErr_sas" type="code">
<pre>
data work.airports ;
  infile 'location.dat' ;
  input @1 Code $3. @4 City $50. ;
run ;
proc prnt data=work.airports ;
run ; </pre>
</macro>
```

The user-defined tags in this file look like HTML tags. These tags provide a natural hierarchical structure for the organizational data that defines the course schema. There are also tags that can contain dynamic instructional content. The data values shown in bold from the above tags are the actual values that get substituted into the example page. The complete XML file, which will be referred to as the XML course definition file, would contain all the non-static content for each page in a course. The versatility and efficiency that is gained by this approach become obvious when changes need to be made in the course. Modifying the data for either a course's dynamic content or course organization would only involve changing the values within the initial XML course definition file or producing an alternate definition file.

A WEB PERSPECTIVE

Now, to add in more specific detail from a web-based document perspective, let's back up and start with a simple HTML file. By itself, an HTML file uses text and tags to provide style, layout, and content for a given document. A more modular approach would be to separate these components out into a CSS (cascading style sheet) file - to define style, a XSL (extensible style sheet) file - to define layout and static content, and an XML file - to define dynamic content as data. The CSS file allows for a great deal of flexibility in controlling style within a document. Changes in fonts, font sizes, colors, and background are easily defined and easily changed within a CSS file. The XSL file can be thought of as an HTML document that contains the static content along with special XSL tags that connect it to an XML data source. An alternate way to look at an XSL file is as a data-driven page template. The XML file is the data source that contains the dynamic instructional content and organizational information. The XML file is also the file that the browser would access to render the page. A visual representation of this explanation can be seen in Figure 2.

Figure 2: HTML or CSS, XSL, and XML



The XML approach not only provides a single-source repository of dynamic instructional content, it also allows for a natural hierarchical data structure in which to define the organizational structure of a course. Assume that chapter 2 is made up only from the material contained within the files: **dcErr_File.xml**, **dcErr_Solution.xml**, **c2ex2.xml**, and **c2ex3.xml**. Let's display more of the definition file that was used to create the example page:

```
<chapt label="Chapter 2"
      title="Getting Started with the SAS System">
  <page id="dcErr"
        label="page 23"
        title="Finding Syntax Errors Demo"
        file ="dcErr_File" />
  <page id="dcErr_s"
        label="Solution Link"
        title="Exercise 1 Solution"
        file =" dcErr_Solution" type ="link" />
  <page id="runSAS"
        label="page 24"
        title="Running SAS Programs - Exercise 2"
        file ="c2ex2" />
  <page id="funCon"
        label="page 25"
        title="Mastering Fundamental Concepts - Exercise 3"
        file="c2ex3" />
</chapt>
```

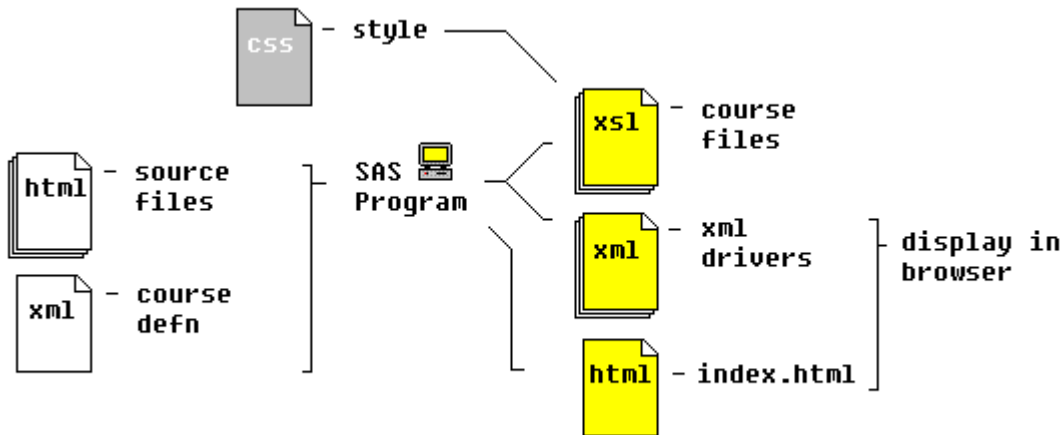
Note that the XML file contains a beginning and ending **chapt** tag for defining course chapters. These tags have label and title attributes for defining a short and long designation for chapter 2. The **chapt** tags are container tags that can encompass a collection of **page** tags. The position of the **page** tags indicates how they are arranged within the chapter and provide a page id, label, title, and file name. But, unlike the XML tag containing the SAS code that only facilitated substituting a code segment into the example page, the use of the organizational information will involve more processing. Actually, the prototype courseware program will use the XML data to build:

1. a set of XSL course files based on the original HTML source files,
2. a set of XML driver files – to display each XSL course file,
3. an **index.html** navigation file.

The original HTML source files are assumed to contain the static text and any XSL tag links to the dynamic XML content (see Appendix A). Given that the XSL tags used to insert the dynamic content do not work in a simple HTML file, these original files need to be converted to XSL files (see Appendix B). This can be accomplished by inserting an appropriate set of XSL header and footer tags around of the HTML text and saving the results in a file with the XSL extension. The main technical issue that needs to be addressed with these new XSL files is that they will not render in a browser. A browser will

render an XML file using an XSL file as a template, but not an XSL file by itself. Therefore, a set of XML “driver files” that these XSL files are linked to will also need to be created (see Appendix C). Finally an **index.html** navigation file can be constructed to link all of the pages together as a web-based course. Now, when processing XML information for these types of tasks, it is common to use Perl scripts, PHP coding, or Java. In our prototype a SAS®9 courseware program is used. Figure 3 illustrates the flow-through of this process:

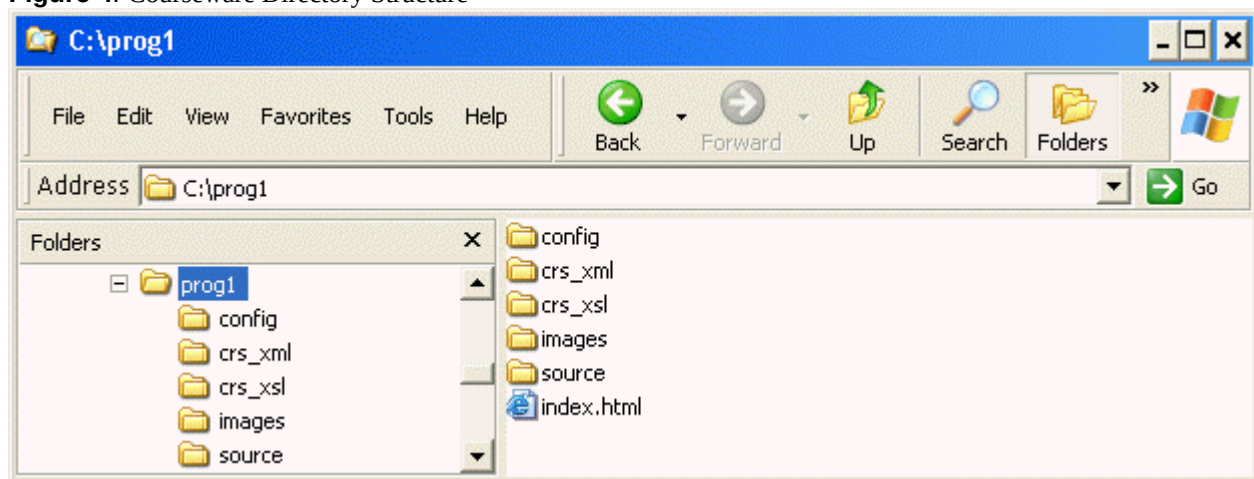
Figure 3: CSS, XML, and HTML source Files to Final Course Pages



COURSEWARE PROTOTYPE

Listed below is the SAS®9 program courseware prototype. It is a macro named **crs_build** that uses the data in the XML course definition file to build all of the course XSL files, the XML driver files, and the course index.html navigation file. The defaults of the macro assume that the subdirectory structure shown in Figure 4 preexist.

Figure 4: Courseware Directory Structure



The location of the course’s root directory is set within a parameter call to the courseware macro. In Figure 4 the root is defined as the folder **C:\prog1**. The names and locations of the subdirectories (folders within the root) in this prototype are fixed. It is assumed that there is a folder named **config** that contains the XML course definition file, the document type definition file for the XML course definition file, and the course CSS file. A folder named **crs_xml** is assumed to exist where all the created course XML driver files will be located. A folder named **crs_xsl** is assumed to exist where all the created course XSL files will be written. These XSL files are created with the SAS courseware program using the original HTML instructional content files located in a folder named **source**. In addition, the HTML course files that contain image tags are assumed to be reading their graphics from a folder named **images**. The only other entity that is shown in Figure 4 is the **index.html** file. It is also built by the SAS courseware program and written, by default, to the root directory. This assumed directory structure is enforced within a root directory by a set of local macro variables defined within the **crs_build** macro. Note that the location of the root directory is the only parameter that this macro takes.

```
%macro crs_build (dir = C:\dir) ;
```

/* These macro variables are used to define the subdirectories and names of the files that will be read or written by the courseware program. They enforce the directory structure that is shown in Figure 4. The initial root directory is the only input parameter that is needed to run the macro.

```
*/  
%let dir      = &dir ;  
%let dir_src  = &dir\source ;  
%let fn_index = &dir\index.html ;  
%let sub_xsl  = crs_xsl ;  
%let sub_xml  = crs_xml ;  
%let fn_css   = config\style.css ;  
%let fn_defn  = config\crs_defn.xml ;  
%let fn_dtd   = config\crs_defn.dtd ;
```

/* This data step reads the tags within the XML course definition file and builds a SAS data set called WORK.TAG_INFO. This data set contains all of the chapter and page organizational data within two variables: tag and text. There are a number of tags in the definition file that identify information about the course: course_name, course_code, book_code, prepared_date, and course_code. These tags are also read, but their data is loaded directly into local macro variables for use within the index.html file.

```
*/  
data tag_info(keep=tag text) ;  
  infile "&dir\&fn_defn" dlm="< >" ;  
  input tag : $16. @ ;  
  length text $ 256 p c $ 1 ;  
  text = " " ;  
  select (tag) ;  
    when ("course_name", "course_code", "book_code", "prepared_date",  
          "chapt") do ;  
      do until (c EQ ">") ;  
        input c $char1. @@ ;  
        if c EQ " " then c = "^" ;  
        text = trim(text) !! c ;  
      end ;  
      text = scan(text,1,"<>") ;  
    end ;  
    when ("page") do ;  
      do until (p EQ "/" AND c EQ ">") ;  
        p = c ;  
        input c $char1. @@ ;  
        if c EQ " " then c = "^" ;  
        text = trim(text) !! c ;  
      end ;  
      text = tranwrd(text,"/>", " ") ;  
    end ;  
    otherwise delete ;  
  end ;  
  text = tranwrd(text,"^"," ") ;  
  text = left(text) ;  
  text = compbl(text) ;  
  if tag in ("chapt","page") then output ;  
  else do ;  
    call symput (trim(tag),trim(text)) ;  
  end ;  
run ;
```

/* This data step reads the data set WORK.TAG_INFO and extracts out the course organizational schema data. That information is then saved within a data set named WORK.PAGE_INFO. Note that a page id and the page file names are all saved in a series of local macro variables: _id_1... _id_n, ... and _fn_1... _fn_n . A count of how many page files are

involved is also put in a local macro variable: `_n`. These local macro variables will be used within a macro level loop directly following this step. The data set will be used later on by the code that creates the `index.html` navigation file.

```

*/
data page_info ;
  set tag_info ;
  length chapt_label $ 16 chapt_title $ 64 ;
  length page_label $ 16 page_title $ 64 ;
  length page_id $ 8 type $ 8 ;
  select (tag) ;
    when ("chapt") do ;
      id_1 = prxparse('/label *= *"\([\w ]*)\"/') ;
      pm_1 = prxmatch(id_1,text) ;
      chapt_label = prxposn(id_1,1,text) ;
      id_2 = prxparse('/title *= *"\([\w ]*)\"/') ;
      pm_2 = prxmatch(id_2,text) ;
      chapt_title = prxposn(id_2,1,text) ;
    end ;
    when ("page") do ;
      id_3 = prxparse('/id *= *"\([\w ]*)\"/') ;
      pm_3 = prxmatch(id_3,text) ;
      page_id = prxposn(id_3,1,text) ;
      id_4 = prxparse('/label *= *"\([\w ]*)\"/') ;
      pm_4 = prxmatch(id_4,text) ;
      page_label = prxposn(id_4,1,text) ;
      id_5 = prxparse('/title *= *"\([\w ]*)\"/') ;
      pm_5 = prxmatch(id_5,text) ;
      page_title = prxposn(id_5,1,text) ;
      id_6 = prxparse('/file *= *"\([\w.]*\")\"/') ;
      pm_6 = prxmatch(id_6,text) ;
      file = prxposn(id_6,1,text) ;
      id_7 = prxparse('/type *= *"\([\w.]*\")\"/') ;
      pm_7 = prxmatch(id_7,text) ;
      type = prxposn(id_7,1,text) ;
      output ;
      n + 1 ;
      call symput ("_n",left(n)) ;
      call symput ("_id_"!!left(n), trim(page_id)) ;
      call symput ("_fn_"!!left(n), trim(file)) ;
    end ;
  otherwise delete ;
end ;
retain chapt_label chapt_title ;
keep chapt_label chapt_title
page_label page_title
page_id file type ;
run ;

```

%do i = 1 **%to** &_n ; *** for each course file ;

/* for each original instructional page an XSL course file is built with three data steps:

- data step 1: starts with a collection of XSL “header” tags put into a new XSL file.
- data step 2: add in the instructional content from the HTML source file – beginning after its <body> tag
- data step 3: finish with a small set of XSL “ending” tags

```

*/
data _null_ ; *** .xsl: xsl header tags ;
  file_out = "&dir\&sub_xsl\&&_fn_&i"!!".xsl" ;
  file fv_1 filevar = file_out ;
  put '<?xml version="1.0"?>' ;
  put '<xsl:stylesheet version="1.0" ' ;
  put '      xmlns:xsl="http://www.w3.org/1999/XSL/Transform">' ;
  put '<xsl:template match="/defn">' ;

```

```

put '<xsl:variable name="pg_id" ' ;
put '      select="' '&&_id_&i" "' '>' ;
put '<html>' ;
put '<head>' ;
put '<link rel="stylesheet" href="' '..\&fn_css" "' />' ;
put '<title>' ;
put '<xsl:value-of select="chapt/page[@id=$pg_id]/@title"/>' ;
put '</title>' ;
put '</head>' ;
put '<body>' ;
put '<span id="pg_n">' ;
put '<xsl:value-of select="chapt/page[@id=$pg_id]/@label"/>' ;
put '</span>' ;
put '<table width="100%">' ;
put '<tr>' ;
put '<td width="150px">' ;
put '' ;
put '</td>' ;
put '<td align="left">' ;
put '<span id="pg_hdr">' ;
put '<xsl:value-of select="chapt/page[@id=$pg_id]/../@label"/> <br/>' ;
put '<xsl:value-of select="chapt/page[@id=$pg_id]/@title"/>' ;
put '</span>' ;
put '</td>' ;
put '</tr>' ;
put '</table>' ;
put '<!-- instruction ..... -->' ;

```

```

run ;
data _null_ ; *** .xsl: attach html instruction ;
retain chk 0 ;
file_in = "&dir_src\&&_fn_&i"!!".html" ;
infile fv_1 filevar = file_in end=EOF trunccover ;
input line $char120. ;
file_out = "&dir\&sub_xsl\&&_fn_&i"!!".xsl" ;
file fv_2 filevar = file_out mod ;
*** attach html content after original body tag ;
if substr(left(line),1,5) EQ "<body" then do ;
    chk = 1 ;
    input line $char120. ;
end ;
if chk EQ 1 then put line $char120. ;

```

```

run ;
data _null_ ; *** .xsl: attach xsl footer tags ;
file_out = "&dir\&sub_xsl\&&_fn_&i"!!".xsl" ;
file fv_1 filevar = file_out mod ;
put '</xsl:template>' ;
put '</xsl:stylesheet>' ;
run ;

```

/* For each XSL file an XML driver file is built. The XML driver files are only different from one another because they each access a particular XSL file.

*/

```

data _null_ ; *** .xml: driver files ;
file_out = "&dir\&sub_xml\&&_fn_&i"!!".xml" ;
file fv_1 filevar = file_out ;
length xsl_href $ 120 ;
xsl_href = 'href="' !! "..\&sub_xsl\&&_fn_&i" !! '.xsl"' ;
put '<?xml version="1.0"?>' ;
put '<?xml-stylesheet type="text/xsl" ' xsl_href +(-1) "?>' ;
put '<!DOCTYPE macro SYSTEM "' '..\&fn_dtd" ' ">' ;
put '<defn> &inc_file; </defn>' ;

```

```
run ;
```

```
%end ;
```

```
/* An index.html navigation file is also built with three data steps:
```

- data step 1: starts with a collection of HTML “header” tags
- data step 2: add in a navigational link to each XSL file through its XML driver
- data step 3: finish with a small set of HTML “ending” tags

```
*/
```

```
data _null_ ; *** index.html: initial tags ;
```

```
  file "&fn_index" ;  
  put '<html>' ;  
  put '<head>' ;  
  put '<link rel="stylesheet" href="' "&fn_css" '"/>' ;  
  put '<title> index.html </title>' ;  
  put '</head>' ;  
  put '<body>' ;  
  put "<b> &course_name </b><br/>" ;  
  put '<b>Exercises</b>' ;  
  put '<hr/>' ;  
  put '<table>' ;
```

```
run ;
```

```
data _null_ ; *** index.html: page anchor tags ;
```

```
  set page_info ;  
  by chapt_label notsorted ;  
  file "&fn_index" mod ;  
  if first.chapt_label EQ 1 then do ;  
    put '<tr> <th> ' chapt_label '<th>' ;  
    put '<td> </td><td> </td><td> </td> </tr>' ;  
  end ;  
  if type NE "link" then do ;  
    put '<tr> <td align="right"> ' page_label '<td>' ;  
    xml_href = "&sub_xml\"!!trim(file)!!".xml" ;  
    put '<td> <a href="' xml_href +(-1) '">' page_title '</a> </td>' ;  
    put '<td> &nbsp; </td>' ;  
    file = "["!!trim(file)!!"]" ;  
    put '<td> ' file '</td>' ;  
    put '</tr>' ;  
  end ;
```

```
run ;
```

```
data _null_ ; *** index.html: footnote, end tags ;
```

```
  file "&fn_index" mod ;  
  put '</table>' ;  
  put '<br/>' ;  
  put '<hr/>' ;  
  put '<span id="crs_info">' ;  
  put "course code: &course_code |&nbsp; " ;  
  put "book code: &book_code |&nbsp; " ;  
  put "prepared date: &prepared_date " ;  
  put '</span>' ;  
  put '<hr/>' ;  
  put '</body>' ;  
  put '</html>' ;
```

```
run ;
```

```
%mend ;
```

```
/* This is the trigger to run the macro against the c:\prog1 root directory files.
```

```
*/
```

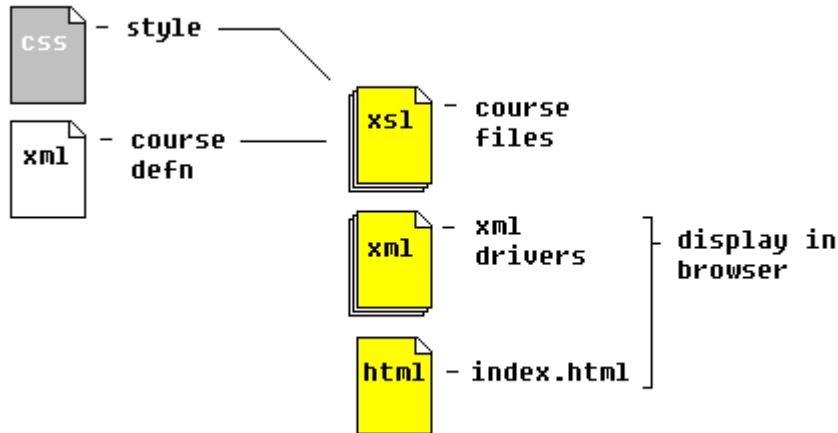
```
%crs_build (dir = C:\prog1)
```

The macro level programming in this prototype is rather simple, as is most of the data step code, but the creation of WORK.PAGE_INFO involves repeatedly parsing XML tag data with perl regular expressions. These types of expressions are processed in functions that are only available with SAS®9.

DEPLOYMENT

A further advantage of using HTML and XML technology is that the resulting pages can easily be presented interactively through a browser, shown from within a web cast, or printed into a PDF document as a stand-alone paper copy. To deploy the resulting course pages, it is only necessary to have the final XSL course files, the XML drivers, access to the original XML course definition file, and access to the CSS file:

Figure 5: Deployment Files



At this point, applying any change in the CSS file will automatically alter how the XSL course files and the index.html file display. It is also possible to automatically display modified dynamic instructional content by simply changing the tagged data within the XML course definition file. However, changes to the organizational schema in the XML definition file would require the courseware program to be rerun.

CONCLUSION

There are pros and cons for the use of this approach. On the pro side, the use of a CSS file for controlling and changing styles for any course content would be a major advantage. A CSS file allows for complete control over font styles, font sizes, color usage, and backgrounds (just to name a few stylistic elements). This would be highly beneficial for maintaining current style requirements or changing to new requirements. It would also allow existing courses to easily be adjusted for any site-specific style considerations. The use of HTML is also highly beneficial. It is accessible everywhere, free, and is very simple to learn. It circumnavigates most platform considerations and issues that might arise with the use of any second-party content display programs. The use of hyperlinks within HTML would also be considered a big plus when building any instructional content. In addition, XML provides two highly valuable capabilities. It allows dynamic instructional content to be automatically updated and displayed. The inherent hierarchical structure of XML is also advantageous when defining the organizational components of a course. But, the use of this approach would have several consequences. Content developers would have to be comfortable with HTML or HTML code generating editors in order to create the original source documents. Content developers or additional support personnel would have to get use to CSS, XSL, and XML syntax. Within an organization it could be argued that only a limited number of CSS files would be needed at a given time, but these files do require that a technically capable person create, maintain, and update them. The same would be true of the initial XSL tag set that is used in the SAS macro to create the course XSL files. And, the course definition file would also have to be carefully planned and constructed by someone familiar with XML and the course content. The major concern here is not the capabilities of the technology but the effort in time and training to effectively utilize it. For while the HTML part of this issue would really not be a problem, the technology behind the CSS, XSL, and XML components would be considered complex and involve a significant effort to learn and effectively use.

Appendix A – The HTML Source File for the Example Page

The following is the contents of the HTML instructional source file for the example page shown in Figure1. It is assumed to consist of simple HTML tags, instructional text, and XSL tags pointing to data within the course XML definition file. The courseware program will use the contents of this file from the <body> tag down to construct the XSL file. The course content author creates this file.

```
<html>
<head>
<!-- html soure file -->
</head>
<body>

<ol start="i" type="i">
<li> Include the program "syntaxError.sas" in the Program Editor.</li>
<li> Submit the program. </li>
<li> Use the SAS log notes to identify the error, correct
      the error, and resubmit the program. </li>
</ol>
<table>
<tr>
  <td valign="top">Program: <br/>syntaxError.sas </td>
  <td>
    <table border="1">
      <tr>
        <td><code><xsl:copy-of select="macro[@id='syntaxErr_sas']"/></code> </td>
      </tr>
    </table>
  </td>
  <td valign="bottom"> <a href="dcErr_Solution.xml"> Solution Link</a> </td>
</tr>
</table>

</body>
</html>
```

Appendix B – The Generated XSL File for the Example Page

The following is the contents of the course XSL file for the example page shown in Figure1. It consists of a preset collection of XSL tags, the contents of the HTML source file from the <body> tag down, and several XSL ending tags. This file is generated by the courseware macro.

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/defn">
<xsl:variable name="pg_id"
  select="'dcErr'"/>
<html>
<head>
<link rel="stylesheet" href="..\config\style.css" />
<title>
<xsl:value-of select="chapt/page[@id=$pg_id]/@title"/>
</title>
</head>
<body>
<span id="pg_n">
<xsl:value-of select="chapt/page[@id=$pg_id]/@label"/>
</span>
<table width="100%">
<tr>
<td width="150px">
```

```


</td>
<td align="left">
<span id="pg_hdr">
<xsl:value-of select="chapt/page[@id=$pg_id]/../@label"/> <br/>
<xsl:value-of select="chapt/page[@id=$pg_id]/@title"/>
</span>
</td>
</tr>
</table>
<!-- instruction ..... -->

<ol start="i" type="i">
<li> Include the program "syntaxError.sas" in the Program Editor.</li>
<li> Submit the program. </li>
<li> Use the SAS log notes to identify the error, correct
      the error, and resubmit the program. </li>
</ol>
<table>
<tr>
  <td valign="top">Program: <br/>syntaxError.sas </td>
  <td>
    <table border="1">
      <tr>
        <td><code><xsl:copy-of select="macro[@id='syntaxErr_sas']"/></code> </td>
      </tr>
    </table>
  </td>
  <td valign="bottom"> <a href="dcErr_Solution.xml"> Solution Link</a> </td>
</tr>
</table>

</body>
</html>
</xsl:template>
</xsl:stylesheet>

```

Appendix C – The XML Driver File For The Example Page

The following is the contents of the XML driver file for the example page shown in Figure1. As the course XSL file will not directly render in a browser, it is necessary to have an associated XML file that will display. This file is also generated by the courseware macro.

```

<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="../../crs_xsl/dcErr_File.xsl"?>
<!DOCTYPE macro SYSTEM "../../config/crs_defn.dtd">
<defn> &inc_file; </defn>

```

REFERENCES

- [1] SAS Institute Inc. (2004), SAS Macro Language Course Notes,
Book Code 59958
Cary, NC: SAS Institute Inc.
- [2] SAS Institute Inc. (2005), SAS Programming III: Advanced Techniques Course Notes,
Book Code E70041
Cary, NC: SAS Institute Inc.
- [3] Keogh, Jim; Davidson, Ken (2005), XML Demystified,
A Self-Teaching Guide
Emeryville, CA: McGraw-Hill
- [4] Castro, Elizabeth (2001), XML For The World Wide Web,
Visual QuickStart Guide
Berkeley, CA: Peachpit Press
- [5] Tidwell, Doug (2001), XSLT,
Sebastopol, CA: O'Reilly Media, Inc.
- [6] McGrath, Mike (2003), XHTML
Warwickshire, United Kingdom: Computer Step.

CONTACT INFORMATION

Please forward comments and questions to:

Don Boudreaux, Ph.D.
SAS Institute Inc.
11920 Wilson Parke Ave
Austin, TX 78726
Phone: 512.258.5171 Ext. 3335
E-mail: don.boudreaux@sas.com

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.