Practical 1: Writing xml with wxml

The aims of this exercises are to familiarize you with the process of compiling the FoX library and using its wxml API to produce simple xml documents. The tasks revolve around producing a simple HTML document, then modifying a program that evaluates a simple mathematical function to output HTML before finally building an HTML report containing mathematical notation.

Exercise 1.1

In the directory "`~/Practical_1/exercise_1`" you should find a file named `simple.f90`. As the name implies this simple fortran program is designed to produce an equally simple XML document. Without compiling the code sketch out what you expect the XML document produced by this code to look like.

Space for your outline of the XML document produced by simple.f90:

Before moving on and compiling the code there are some aspects you should note (you may have already spotted some or all of these, but they are listed below anyway).

1. There are two important stylistic points to note. First most of the arguments passed to subroutines in the FoX library make use of keywords. This is useful as it makes the code more readable, is less likely to break if you upgrade to a new version of the FoX library and is the only way to deal with the many optional arguments in many of the subroutines. Second, it is worth noting the indentation scheme used. Whenever a new element is opened the level of indentation is increased, as if the call was opening an if or do block. This helps prevent the creation of code that leads to the production of unbalanced or overlapping tags (things that are forbidden in XML).

2. There are two calls to `xml_AddAttribute`, one for each "MyXMLElement" element and they are passing different data types as the value argument of the call. To anybody versed in Fortran 77 it may come as a surprise that this can be done in Fortran, but it is permitted for many attributes of the subroutines in FoX. If you are interested in know-

ing how this works you could take a look at 'module', 'generic' or 'type bound' procedures in a Fortran 90 reference book.

3. Take a close look at the string passed to the second `xml_AddAttribute` call. Some of these characters are not permitted within an XML document (at least not permitted outside of CDATA) as they have special meanings. The problematic characters are "<" and "&" - both of these are correctly escaped by FoX so they can be passed in as part of the string.

Exercise 1.2
Now it is time to compile `simple.f90`, link it against the FoX library and see if it produces the results you expect. To do this you will need to download, configure and make FoX (see the reminder below). We suggest that this is done in your homespace or on the desktop so it can be accessed easily for the whole practical.

Once you have compiled the code you can able to run it by typing "`./simple`" at the command prompt. A new file should be produced called `simple.xml`. Look at this file - does it look like the outline you sketched in exercise 1? There are a couple of things you should note related to the calls to `xml_AddAttribute`. The floating point number "`2.0`" is expressed in scientific notation, FoX does this by default for all floating point numbers it writes out, and the problematic characters in the second call have been encoded.

Reminder - compiling and linking FoX
1) Download the latest FoX source code from:

    http://www.uszla.me.uk/software/source/FoX/

2) Uncompress to your home directory, and build the libraries:

```
cd ~
tar xzf FoX-3.1.2.tgz
cd FoX-3.1.2
./configure
make
```

3) For a single file, the compiler command to compile and link while including FoX is:

```
g95 -o simple simple.f90 `~/FoX-3.1.2/FoX-config`
```

4) Or, compiling and linking in separate steps: first compile source code with relevant include flags:

```
cd Practical_1/exercise_1
g95 -c `~/FoX-3.0/FoX-config --fcflags` simple.f90
```

5) Then link source code with library and library search flags:

```
g95 simple.o `~/FoX-3.0/FoX-config --libs` -o simple
```

Exercise 1.3
Before moving on to write your own fortran program it's worth looking to see what FoX does when a program attempts to output XML that is not well formed. There are many ways that an XML document can be badly formed and the aim of this exercise is to write a fortran code that can be compiled, but that produces badly formed XML. Modify `simple.f90`, recompile the code and run it to see what happens in the following cases:

1. Closing an unopened tag. What happens if you attempt to close an element that has not been opened. It may be particularly instructive to change the case of the string representing the final element name from "MyXmlRoot" to "MyXMLRoot"?

   Answer: ____

2. Unbalanced tags. What happens when an element is closed before an element that it encloses is closed (change the order of the last two `xml_EndElement` calls, for example).

   Answer: ____

3. Unclosed tags. What happens if a program terminates correctly without closing all the tags (remove the last two `xml_EndElement` calls, for example).

   Answer: ____

4. Adding elements to an unopened file. What happens when you attempt to create a new element before calling `xml_OpenFile`.

   Answer: ____

Exercise 1.4
Write a fortran program to produce a simple home page in html. The code should be simple (a single program block without flow control) and use FoX's wxml interface to output XHTML. You could use `simple.f90` as a starting point. There should be an `<html>` root element with two children, `<head>` and `<body>`, the `<head>` element should have at least a `<title>` child while the `<body>` element should contain one or more `<p>` and `<h1>` elements. Compile the code and link it to the FoX libraries.

Confirm that the code writes a valid html file and check that you see the expected rendering when it is loaded into a web browser (use the "File → Open File" menu in Firefox). The example solution produces XHTML like that shown in the "quick introduction to HTML", on the next page.

You should also check that your file declares the correct namespace - you should see `xmlns='http://www.w3.org/1999/xhtml'` within the root `<html>` tag. If not you will need to add a namespace declaration with a call to the `xml_DeclareNamespace` subroutine immediately before your first `xml_AddElement` call. The meaning and usefulness of XML namespaces will be explained later in the course.

Quick guide to (x)html
HTML is the markup language used to encode the structure and to a lesser extent the meaning of pages on the web. Web browsers load, parse and render a page for display based on content of the document. There are several versions of HTML and one of these (XHTML) is strictly an XML language. XHTML documents have `<html>` as the document root, one `<head>` element and one `<body>` element. The document head contains metadata about the document and the body contains the data that will actually be displayed by the browser. A (very) simple example looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<html xmlns=”http://www.w3.org/1999/xhtml”>
  <head>
    <title>Andrew&apos;s home page</title>
  </head>
  <body>
    <h1>Andrew&apos;s home page</h1>
    <p>I&apos;m a postdoctoral research fellow and was
    involved in the <i>e</i>Minerals project. This is not
    my web site.</p>
  </body>
</html>
```

Note that this is a well-formed XML document. Other tags to note are `<p>`, which delimits a paragraph, `<i>` which marks text to be italicized and `<b>`, marking bold text. In order to make sure that Firefox correctly understands the document is XHTML, files loaded from the local machine should have a ‘`.xhtml`’ extension and the `xmlns` attribute should be included as shown above.

Exercise 1.5
The files for exercise 1.5 and 1.6 can be found in “~/Practical_1/exercise_5”. The Fortran program contained in the file erf.f90 calculates the error function:

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt.$$

using the Taylor series expansion:

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{(2n+1)n!} = \frac{2}{\sqrt{\pi}} \left(x - \frac{x^3}{3} + \frac{x^5}{10} - \frac{x^7}{42} + \frac{x^9}{216} - \cdots \right)$$

for input values of the argument (x) and the truncation of the expansion (n). The series itself is implemented in function `erf_loop`.

Currently the program only writes output to the standard output stream (you should see it in the terminal when you compile and run the program). Your task for this part of the exercise is to have the program output an XHTML file containing the same data as is currently output to the terminal in addition to the current output. You should be able to view the result in a web browser.

Exercise 1.6
The aim of this exercise is to write an (XML encoded) record of the actual calculation used to produce the result of the error function. Ideally we would like to write this record in a format that can be 'understood' by a computer in a way that simply writing the appropriate symbols into an HTML file is not. For example, we may like a computer to be able to parse the XML document and extract the expression then produce some executable code to enable the expression to be evaluated. In short, the aim is to encode the semantics of the summation into the XML document. Do this by modifying the `erf_loop` functions to write content MathML to the same file as the HTML is being written to. Content MathML is an XML standard for expressing the meaning of a mathematical expression - and Firefox is capable of parsing MathML and producing human readable output.

A short guide to MathML is provided over the page. A good starting point is to write a small standalone Fortran program to write some of the MathML you will need to form the full expression. For example *2n+1* should be encoded as shown on the right.

```
<math xmlns="http://www.w3.org/1998/Math/MathML">
    <apply>
      <plus/>
      <ci>
        <apply>
          <times/>
          <cn> 2 </cn>
          <ci> n </ci>
        </apply>
      </ci>
      <cn> 1 </cn>
    </apply>
</math>
```

I'm not going to write very much more in the way of explicit instructions for this exercise - the information below and over the page should help get you started and you should feel free to ask one of us if you are having trouble.

A short guide to content MathML
Perhaps the best way to explain how content MathML works is by starting with an example of the encoding for *(a+b)*. The MathML to express this is

```
<apply>
  <plus/>
  <ci> a </ci>
  <ci> b </ci>
</apply>
```

We use reverse polish notation and embed operations within `<apply>` elements. The fragment says "apply the addition operator (plus) to the variables (ci) *a* and *b*". Expressions can be embedded within other expressions - so x^{2n+1} can be expressed:

continued over...

```
<apply>
  <power/>
  <ci> x </ci>
  <ci>
    <apply>
      <plus/>
      <ci>
        <apply>
          <times/>
          <cn> 2 </cn>
          <ci> n </ci>
        </apply>
      </ci>
      <cn> 1 </cn>
    </apply>
  </ci>
</apply>
```

Note that `<ci>` represents a variable and `<cn>` represents a number. You can find a list of the other elements you will need to use are over the page. In order to put mathematical expressions within an XHTML document, you must embed them as shown below, and, just before you start adding MathML elements, do:

```
call xml_DeclareNamespace(xf, “http://www.w3.org/1998/Math/MathML”)
```

You will also need to include a reference to an XML style sheet at the top of the document. The style sheets are included in the exercise_5 directory. To do this include:

```
call xml_AddXMLStylesheet(xf, type="text/xsl", href="MathML_XSLT/mathml.xsl")
```

between opening the XML document and writing any XML tags. You should end up with a document something like this:

```
<?xml-stylesheet type="text/xsl" href="MathML_XSLT/mathml.xsl"?>
<html xmlns="http://www.w3.org/1999/xhtml”>
  <head>
    <title>MathML in an HTML document</title>
  </head>
  <body>
    <p> Expressing (a+b) is easy:</p>
    <math xmlns="http://www.w3.org/1998/Math/MathML">
      <apply>
        <plus/>
        <ci> a </ci>
        <ci> b </ci>
      </apply>
    </math>
  </body>
</html>
```

continued over...

The mathML elements you will need to use are:

`<math>`: This must be the root element of any MathML fragment.

`<root>`: the nth root of a is is given by:

```
<apply>
  <root/>
  <degree><ci type='integer'> n </ci></degree>
  <ci> a </ci>
</apply>
```

`<power>`: The power element is a generic exponentiation operator. That is, when applied to arguments a and b, it returns the value of a^b.

```
<apply>
  <power/>
  <ci> a </ci>
  <ci> b </ci>
</apply>
```

If this were evaluated at a= 5 and b=3 it would yield 125.

`<factorial>`: *n!* would be represented by:

```
<apply>
  <factorial/>
  <ci> n </ci>
</apply>
```

If this were evaluated at n = 5 it would evaluate to 120.

`<minus>`: This can be used as a unary arithmetic operator (e.g. to represent "*-x*") or a binary arithmetic operator (e.g. "*x-y*"). These examples would be:

```
<apply>
  <minus/>
  <ci> x </ci>
</apply>
```

and:

```
<apply>
  <minus/>
  <ci> x </ci>
  <ci> y </ci>
</apply>
```

respectively.

`<pi/>`: represents π.

A much longer list of operators can be found in the MathML specification at: http://www.w3.org/TR/2003/REC-MathML2-20031021/chapter4.html