Practical 5: Reading XML data into a scientific application

1 Introduction

The last practical session showed you how to read XML data into Fortran using the SAX interface. As discussed, SAX is well suited to large but simply-structured documents - it is fast, and has minimal memory usage, but extracting information from moderately complex documents can quickly become tedious.

Note how, for example, when provided with a piece of character data, you are not told where in the document it occurs, you must keep track of that yourself. There are also a number of gotchas that you have to be aware of; for example, the fact that one segment of character data may not all come in one chunk.

For this reason, it is often worth using an alternative interface, called the DOM - the Document Object Model. This is slower, and much more memory-intensive - the whole document must be read into memory and held there for the duration of any interaction with it. This makes it very ill-suited to large documents. However, as long as the document fits in memory, then this is more than outweighed by the ease of data extraction.

Recall the mapping of an XML document onto a tree-shaped structure that you were shown in other lecture. It is this tree that is presented to you by the DOM interface, along with a series of subroutines and functions for navigating and manipulating it.

The DOM interface is a W3C standard (that is, it is designed and codified by the same people who standardized XML itself.) As such, it is independent of any programming language, and for most languages available to day there exists at least one, if not more, implementations of the DOM interface. For Fortran, however, FoX is the only available implementation.

Because the DOM is a standard interface, there are a multitude of tutorials and examples of its use available on the web. Most of these are written in Javascript or Java - nevertheless, there is a direct mapping of the relevant DOM calls onto the FoX DOM, and you will see how to do this.

This means that if you already have access to an algorithm for extracting information from an XML tree, expressed as DOM calls, you can immediately translate it to Fortran.

The FoX DOM documentation is currently quite sparse, though it does provide a list of all the DOM functions and subroutines implemented. For fuller documentation, then w3schools is a very good overview of the API.

So - if you find this tutorial insufficient, or you wish to learn more, there is no shortage of further resources - please see the appendix for further information.

2 Practical exercises

For these exercise, please see the directory `Practical_5/`, within which there are a number of directories, one for each exercise below. Each directory contains an example Fortran program for the exercise, and is set up with a Makefile so you can make the executable by simply typing `make` in each directory.

3 Exercise 5-1: Parse a file.

Look at `exercise_5_1.f90`. This is the very simplest DOM program, which simply opens and then closes an XML file.

This shows you:

- the module you need to USE at the top.
- the declaration of the document object you need to make.
- how to retrieve the document object from a file
- how to clean up memory afterwards.

As you can see, you need to use a `Node` object, which is used to hold the document parse tree. You will use this `Node` object a lot in the DOM; nearly everything is represented as a `Node`.

Node pointers
You never use a `Node` object directly; only ever as a `pointer`. If you've never used pointers in Fortran or other languages, you don't need to worry too much about this. Just remember that you must *always* declare DOM objects as pointers - your programs will fail to work otherwise.
In most cases, your programs will simply fail to compile if you haven't declared your objects as pointer. Remember to check declarations if you have troubles compiling.
Try this - declare the `doc` variable as a normal variable, without it being a pointer. Notice the error message.

Cleaning up memory
As a result of the use of pointers, it is also necessary to explicitly tell FoX to clear up its memory once you're finished. If you don't, your program will still run correctly, but it will leak memory - that is, in a long-running program which opens lots of files, you will use more and more memory unnecessarily.
Try commenting out the `destroy()` call. If you are using g95, you will see that lots of memory isn't getting deallocated.

The `parseFile` call opens the file and extracts all the XML information into memory, all before doing anything else in the program. This means that if the file doesn't exist, or is not correct XML, then your program will fail with an error immediately - try this and see.

However, you can check whether the call succeeds, and carry on rather than failing. You do this in the same way as with a normal Fortran `open()` statement. There is an optional integer argument, `iostat`. If this is passed in, then `iostat` is set to a non-zero number if the file failed to open for any reason. Try editing the program so it does this

```
doc => parseFile("output.xml", iostat=i)
if (i/=0) then
  print*, "Could not open XML file"
  !  call goAndDoSomethingElseInstead()
endif
```

4 Exercise 5-2: Explore the document object, and look at a node.

Look at `exercise5-2.f90`.

In this program, we parse the file as before, and the parse tree is placed into the `document` variable, which is a `Node` pointer. In order to start getting information from this tree, we first get the root node. This is called the "document element" in DOM terminology (because it is the root element of the document), and, logically enough, we get it by calling the `getDocument-Element` function on the document node. The result (the **document element** itself) is stored in another `Node` pointer, called `element`.

Pointer assignment
When we use the `getDocumentElement` function, we use the following notation:

```
elem => getDocumentElement(doc)
```

that is, with "=>" instead of "=". This is called "pointer assignment". Whenever the **Left Hand Side** of an expression is a FoX pointer, you need to use pointer assignment instead of a normal equals sign.
If you fail to do this, your program will crash.

We then inspect this node by getting its name, and printing it out - and then doing the same for its namespaceURI and localname. Again, to do this you use the logically-named `getNodeName`, `getNamespaceURI`, and `getLocalName` functions.

This is a theme throughout the FoX DOM API. In order to extract information about a DOM object, you use a function called `getXXX`, where `XXX` is the property you are interested in. For a full list of properties accessible in this way, see the FoX documentation.

5 Exercise 5-3: Look at the attributes of a node.

Look at `exercise 5-3.f90`.

Here we extract the root element again, and then look at its attributes. We do this in three different ways:

1. by simply asking directly (with `getAttribute`) for the value of a particular attribute. Note that when we ask for the value of a value of a non-existent attribute, we just get the empty string (so how would you distinguish between a non-existent attribute, and an attribute whose value is "" ?)
2. by asking for an attribute (with `getAttributeNode`) of a given name, and querying the node we are given. Note that if we ask for a non-existent attribute, then we get a non-`associated` pointer back. This is how you can check if a given attribute is present or not. What happens if you try and get the value of the non-`associated` pointer? Try it and see, you will get a segfault.

Note
An attribute is a `Node`, just like an element or a document.

Advanced usage . . .
The third way of accessing attribute values is by walking over the full list of attributes and inspecting them. We retrieve the attribute list (with the `getAttributes` function), find out how long it is (with the `getLength` function), and then loop over the list, extracting and inspecting each attribute in turn (with the `item` function). Each attribute node is then inspected by querying its `Name` and `Value` properties.
When you retrieve the list of `Node`s, it is put into a type called the `NamedNodeMap`.
Be aware, because the DOM interface was designed by Java/JS programmers, the loop has to go from a `0` index rather than the `1` index a Fortran programmer might expect.

These three methods may be useful in different circumstances. Most often, the first will be what you want. If you need to distinguish between empty attributes and non-existent attributes, then the second method is best, while the last method is available in extremis if you are not sure what's there, and you're simply exploring.

Note
Despite the fact that we are now dealing with multiple node and nodelist pointers, we still only do one `destroy()` call to clear up memory. You should never try to `destroy()` any node other than the document node. To see, try `destroy()`ing the element node as well as the document node - you will see a segfault or a memory leak.

---

6 Exercise 5-4 Retrieving particular elements.

Look at `exercise5-4.f90`.

This exercise shows you how to get at particular elements within a tree.

We introduce a `NodeList` here. This is the third derived type you need to know about - while a `NamedNodeMap` holds a list of attributes attached to a particular node, a `NodeList` is just a catch-all list for a list of nodes.

Note
We are now using the `hypoDD.kml` file (created at the end of Practical 3) as the input file. As this is quite a large XML file, it might take a couple of seconds to parse - don't worry about this!

You will see in the program how you collect a list of relevant nodes from a document - you use the `getElementsByTagName` call, which returns to you the list of all elements with that name.

You can retrieve the number of nodes in that list with `getLength`, and inspect each node in it with `item`.

The first loop simply walks through the list of `Placemark` nodes, and prints out the name of each node. (Obviously, the name is simply `"Placemark"` each time.)

Note
Again, as with the `NamedNodeMap` in the previous exercise, you must count from `0` to `getLength()-1` when extracting an `item` from the list.

Now, extend the program to find all the `location` elements in the document, and print out the values of their `unique_id` attributes.

If you need to deal with namespaced XML data, there is a parallel call, `getElementsByTagNameNS`. Since this is a namespaced document, try searching for the same set of nodes, but specifying their namespaces. Make sure you get the same result.

Tip
the syntax is `getElementsByTagNameNS(document, namespaceURI, localName)`

7 Exercise 5-5 Finding nodes within nodes

Sometimes the information you require isn't an attribute of the node you've found, it's held as part of a child element.

In `exercise_5_5.f90`, you can see how to retrieve such an element; in this case, the name of a placemark.

Similarly to the previous exercise, we execute a `getElementsByTagName` query, this time extracting all the `Placemark` tags.

Placemarks in KML
Recall that a `Placemark` in KML is described like so:

```
<Placemark>
  <description>Empire State Building</description>
</Placemark>
```

The program then walks across this list, looking at each placemark in turn (using the `item`) function. To extract the `name` child of this `Placemark`, we then execute another `getElementsByTagName` - but now we execute it relative to the current placemark, rather than the whole document. To do this, we pass in the node of interest as the first argument.

This returns another `NodeList` - but in this case the nodelist only has one child, the single `name` child element of the current `Placemark`. We can select that element from the `NodeList` with the `item` call - and print out its name. (Again, of course, we know its name anyway.)

Chaining DOM calls together
We can save a little bit of effort in this program by chaining DOM calls together. Since in this case we know there is only one node in the list, and we only care about that node rather than the list, we could replace:

```
descriptionList => getElementsByTagname(placemark, "description")
description => item(descriptionList, 0)
```

with this:

```
description => item( getElementsByTagname(placemark, "description"), 0)
```

and avoid the need for a `NodeList` variable.

Error checking
We didn't check that the `NodeList` actually contained a node before extracting it. How would we have checked it? What happens if you try and extract a node which isn't there (try searching for "prescription" instead of "description").

8 Exercise 5-6 Retrieving text value of elements,

Look at `exercise_5_6.f90`.

The previous exercises showed you how to to find particular elements within a document and look at their attributes. The final part of an XML document you might want to inspect is the text content.

In this program, you can see how to extract the text content of an element. Unsurprisingly, you use the `getTextContent` call.

It finds all the `description` elements within `Placemark`s, and prints out the contents.

A useful thing about the `getTextContent` call is how it behaves when an element has tags as well as text inside it. Try changing the name of one of the `description` tags to include some HTML markup. (for example, you could do:

```
<description> This is <i>my</i> house. </description>
```

Note
This is only for the purpose of this exercise; this would not be valid KML - KML requires you to include HTML in a different way.

What happens when you get the text content of such a `description`?

You should find that the text content is simply **all** of the text within it, including inside any child elements.

`getTextContent` faithfully retrieves all of the text for you without removing any spaces. Occasionally this can be annoying. Try editing one of the `description` tags to include a newline:

```
<description> This is
<i>my</i> house.
</description>
```

You should get a rather confusing result on printing that out. This is because the newline character is being preserved, and printing out a newline character from Fortran can have odd, platform-dependent, consequences.

Note
If you manipulate the string in memory, it will be correct, but printing out will look strange. And a newline is very rarely what you are interested in; see the note at the end of exercise 5-7.

9 Exercise 5-7. Retrieving data from XML.

FoX extensions to the DOM
Up until now, all of the functions you have used have been standard W3C DOM functions; that is, you have been programming to a standard, language-agnostic API.
The disadvantage of such a standard API is that it avoids interacting with any potentially language-specific features - like data types, or arrays. So you can't extract numerical data using DOM calls (or rather, you can, but only in a very cumbersome fashion by reading strings as numbers and converting them yourselves.)
However, FoX offers several functions which let you automatically extract data into a numerical variable.

Look at `exercise_5_7.f90`.

We are now dealing with data extraction, so we turn our attention to the quakeML parts of the file.

We use two new functions here, `extractDataContent` and `extractDataAttribute`.

- `extractDataContent` extracts data from the text content of an element (that is, it converts the result of `getTextContent` to an appropriate data type). It needs two arguments - the first is the node to extract data from, and the second is the variable into which to put your data.

```
call extractDataContent(element, data)
```

- `extractDataAttribute` extracts data from a named attribute of an element (that is, it converts the result of `getAttribute` to an appropriate data type). It takes three arguments; the first is the relevant element, the second is the name of the relevant attribute, and the third is the variable into which to put your data.

```
call extractDataAttribute(element, att_name, data)
```

Note
There is a third parallel function, `extractDataAttributeNS`, which works in the same way for namespaced attributes. These are much less frequently used.

In this case, we extract two items of integer data - the location ID, and the year. We indicate that we expect integer data by the fact that we pass integer variables as the data-containing arguments.

If we wanted to extract double precision data, we'd need to simply pass in double precision arguments. Try doing this, and extracting the `latitude`, `longitude`, `depth`, and `magnitude` of each earthquake.

Extracting data of other types, and making life easier for strings.
The `extractData` functions will work for any of the standard Fortran data tyeps; *i.e.* `character`, `logical`, `integer`, `real`, `double precision`, `complex` or `double complex`.
As a handy shortcut, when you `extractData` to a string, then the string will be processed to remove all new lines, remove initial space, and condense multiple spaces. So, for example:

```
<string>
This is
a        string
containing newlines
and whitespace.
</string>
```

will result in the string:
`"This is a string containing newlines and whitespace"`
This solves the problem shown at the end of exercise 5-6. (Of course, if you want the unmodified string, you are free to retrieve it with `getTextContent`).

---

10 Exercise 5-8: Extracting array information.

If you're dealing with Fortran programs, then you probably care just as much about arrays and matrices of numbers as you do about individual numbers. Fortunately, `extractData` knows about these as well.

In `exercise_5_8.f90`, we extract the length-`2` arrays containing the coordinates of each `Placemark`, and print them out.

Array syntax in XML
There is no one syntax for representing arrays or matrices in XML; each XML language may have its own. However, FoX will understand any arrays where the elements are separated by whitespace or commas.

Arrays and matrices in XML; error checking.
When extracting data to arrays and matrices, FoX simply reads through the XML data, filling the array or matrix as it goes. This means that from FoX's point of view, if you pass it a length-`4` array, or a `2x2` matrix, the result is the same either way - the first 4 numbers it finds are placed in the data variable.
If more numbers are present, they are ignored; if FoX can't find enough numbers, it will fill the rest of the array with zeroes, and stop with an error message.
If necessary, you can catch this error with `iostat`, and carry on regardless (in the same way as when you try and read too many numbers from a normal file with `read`)

You are now able to replicate the program which you wrote at the end of Practical 3. There it was done in Python, using XPath expressions; here you can do it in Fortran, extracting information by DOM calls. This will be more verbose, but will be more familiar to a Fortran programmer.

You can do this by creating two `NodeLists`; one containing the `Points` from the first `Folder` (with the `name "Initial Positions"`); and one containing all the `Points` from the second `Folder`.

Note
This illustrates the importance of organizing your XML files hierarchically. If you simply had a series of data organized in order, it would be just as easy to look down them with a human eye, but much more difficult to extract them from a DOM tree.

If you simply walk through both `NodeLists` at the same time, you can extract the coordinates for the initial and final positions and compare them. A function to calculate great-circle distances is provided in `example_5_8.f90`.

Do so, and print out the distance that each calculated earthquake location has moved during the `hypoDD` run.

For comparison, a working solution is included in `solution_5_8.f90`.

Note
Our solution here is different from the XPath solution in two ways:

1. When we solved this problem with XPath, we tested on equality of `unique_id` to match up the different `Points`. This isn't directly possible in DOM, but it is achievable. How would you do this, and why is it a bad idea?
2. When we solved this problem with XPath, we extracted the data from the `qml:latitude` and `qml:longitude` elements - here we are extracting them from the `kml:Point` element. Why are we doing it differently - does it make any difference?

11 Conclusion

This practical introduced you to the use of the FoX DOM. You have learnt

- What you need to add to your Fortran program to use the DOM, and open and close a file.
- How to pick out particular elements from a DOM tree.
- How to get at the value of attributes, and the text content of elements
- How to extract these values and contents to native Fortran datatypes.

The FoX DOM includes a full implementation of all of the W3C DOM levels 1, 2, and (most of) 3. This includes well over a hundred different functions and subroutines, which are available for use; and if you need to use parts of a program from a DOM in another language, the translation will be straightforward.

Nevertheless, most of what you need to accomplish can be done using only the following functions:

- `parseFile`
- `getElementsByTagName(NS)`
- `getLength`
- `item`
- `extractDataContent`
- `extractDataAttribute`
- `destroy`

and the DOM part of a typical program will have a skeleton looking very much like the example programs shown in this exercise.

```
doc => parseFile("input.xml")

nlist => getElementsByTagname(doc, "interestingElement")

do i = 0, getLength(nlist)-1
  np => item(nlist, i)

  call extractDataAttribute(np, "att", value)

  if (value==somethingInteresting) then
    call extractDataContent(np, data)
  endif
enddo

call destroy(doc)
```