Practical 4: XML input to Fortran

The first two practical sessions dealt with how to produce XML from Fortran code. The third session showed you how to use XPath to write scripts to analyse that data. However, there is as yet no XPath interface to Fortran, and sometimes you may need to read XML docu- ments into Fortran. In this practical session, you will learn one method for doing this.

In the earlier talk on the XML landscape, a number of interfaces were mentioned. The historically earliest was SAX. It is conceptually the simplest interface; and is also the easiest to implement, and there is a Fortran SAX interface in FoX.

Note
FoX also includes a DOM interface (an introduction to this is in Practical 5). For many purposes, the DOM interface will be considerably easier to use.

This practical is designed to give you an understanding of how to use the FoX SAX interface to import data from an XML file into a Fortran program.

The FoX documentation gives an introduction to the concepts behind SAX, and some initial examples of its use. SAX relies on an event-based callback programming model - the XML parser sends messages to the main program, which are received by handlers provided by the programmer. It can be tricky to get your head around programming in this style, so in this practical session, you will work through some example exercises to learn how to do this.

We will be using as input the earthquake data XML files that we taught `hypoDD` to produce in the second practical session, and the example programs you will write are designed to work towards the sort of data handling you might want to do in a real Fortran program using scientific data.

The directory `Practical_4` contains the necessary files. Inside you will find several directories - one for each practical. Each practical involves taking the skeleton of an existing program already set up for FoX SAX (in every case, this program is called `exercise_4_X.f90`), and adapting it to perform some task. If you get stuck on any of the exercises, complete solutions are available in the same directory, as a file called `solution_4_X.f90`.

Compiling the exercises
The exercises all come with Makefiles (pointing at the copy of FoX in your home directory), such that they can all be compiled by just typing `make`.
If you want to compile the solutions (or indeed any other file), then for any file called `filename.f90`, you can just type `make filename.exe`.

1 Exercise 4-1: Watching SAX events

A sample SAX test program is provided. It consists of two things; a module containing a list of "handlers", and a driver program. The handlers do nothing except print out when they are called; and the driver program does nothing but pass the handlers into FoX. Compile the program, and execute it. It will read in the file `input.xml`, and output a list of events. Compare the list of events to the file `input.xml`, and you should see the correspondence.

You may see rather more `Character` events than you were expecting. This is because SAX will report all characters it finds between tags - even if they are just whitespace. Most of those character events are reporting empty space.

Edit the handlers, so that the `startElement` and `endElement` handlers print out the name of the element, and the `characters` handler prints out the characters it receives.

2 Exercise 4-2: Attribute dictionaries

Attributes are handled in a slightly more complicated way within SAX. If you look at the `startElement` handler, you'll see that it takes an argument `atts`. This contains a dictionary of attributes - that is to say a list of name/value pairs. There is a long list of functions and subroutines to manipulate this dictionary, which are explained in full in the FoX documentation.

Attribute dictionaries
Attribute dictionaries consist of a series of entries, which hold information about each entry. If you need to worry about namespaced attributes, these can be quite complex, but that is fairly rare. For the purposes of these exercises, you need only think about attributes as name-value pairs.
`startElement` will pass a dictionary containing all the attributes for a given element. You can get at this list of attributes using the following Fortran calls:

- `len(atts)` gives the number of attributes
- `getQName(atts, i)` gives the name of the `i`th attribute
- `getValue(atts, i)` gives the value of the `i`th attribute
- `hasKey(atts, name)` gives true or false according to whether the attribute `name` exists
- `getValue(atts, name)` gets the value of the attribute called `name` if it exists.

Alter the `startElement_handler` subroutine so that it prints out all of the attribute name/value pairs.

Now, write a `startElement_handler` which looks for the units of latitude, and stores it in a module variable. At the end of the program, once the document has been parsed and the file closed, print out that module variable.

Fortran module variables
If you are not very familiar with Fortran 90/95, module variables may be new to you. Within a Fortran module, you can keep subroutines (as we have been doing for the event handlers) and also variables. Any variables declared in the first section of a module are accessible to all subroutines within that module (and to any other parts of the program which use that module.) You must remember to `save` the module variable, or its value may not be preserved.

```
module tiny
  integer, save :: i = 0
contains
  subroutine in
    i = 1
  end subroutine in
  subroutine out
    print*, i
  end subroutine out
end module tiny
```

In this module, both subroutines can read and write the variable i.

3 Exercise 4-3: Parsing a simple XML document

So far, all the XML data manipulation you have done has disposed of the data immediately. For a real program, usually you want to read in the data, and keep it around to do something with it later. In order to do this, you need to make use of persistent variables somewhere else to record the data, and also to record where you are in the process of reading the document.

For example, if you want to read the characters within a certain tag, you can't rely on the characters callback telling you where it is; it doesn't give you that information. So you need to keep track of which `startElement`/`endElement` events you have been received, and only read the character data when appropriate.

The easiest way to do this is using module variables. If you place a single logical variable within your module, you can switch it on and off as you enter and leave an element which you're interested in. Then, whenever you receive a character event, you can decide what to do with the characters based on the setting of that logical variable.

Similarly, if you retrieve some data from an event, and want to keep it around, you can put it into a module string variable, and then use it again later.

Using your input file and framework, write a program that reads the text of the latitude and longitude variables, as well as their units, and then prints it out like so:

`The earthquake location was latitude` *XXX UNITS* `and longitude` *YYY UNITS*

You'll need to keep track of your location in the file, in order to read the correct characters, and then preserve the character and attribute values.

4 Exercise 4-4: Namespaces and SAX

You have learnt that namespaces can be used to disambiguate various elements, and to distinguish different XML vocabularies.

SAX will provide data to do this namespace disambiguation for you. For every element it encounters, it will report both the Name of the element, and also its Namespace URI. When dealing with namespaced documents, you often don't care about the elements `name` any more - you care about the combination of its `localname` and its `namespaceURI`.

Namespaces in SAX

When you recieve a `startElement` or `endElement event`, it comes with three strings. So far, you've only paid attention to the name. However, in the presence of namespaces, names don't matter - what matters is the other two strings, namespaceURI and localname. Here are some examples of what SAX will report to you on finding various tags:

```
<description xmlns="http://purl.org/dc/elements/1.1/">
```

```
name         = "dc:description"
localname    = "description"
namespaceURI = "http://purl.org/dc/elements/1.1/"
```

```
<dc:description xmlns:dc="http://purl.org/dc/elements/1.1/">
```

```
name  =        "description:
localname =    "description"
namespaceURI = "http://purl.org/dc/elements/1.1/"
```

```
 <description xmlns="http://earth.google.com/kml/2.0">
```

```
name         = "description"
localname    = "description"
namespaceURI = "http://earth.google.com/kml/2.0"
```

```
<kml:description kml:xmlns="http://earth.google.com/kml/2.0">
```

```
name         = "kml:description"
localname    = "description"
namespaceURI = "http://earth.google.com/kml/2.0"
```

The first and second tag are equivalent; as are the third and fourth.
Note in particular how the first and third tag have the same QName, and are only distinguishable by their namespaceURI.
You can see from this how testing a combination of localname and namespaceURI will let you use SAX to check for the right tag.
The dictionary functions hasKey and getValue can both be used with namespaces:

- `hasKey(atts, uri, localname)` returns true or false according to whether the attribute with the specified `uri` and `localname` exist.
- `getValue(atts, uri, localname)` gets the value of the attribute with the specified `uri` and `localname`.

To illustrate this, the input.xml for this exercise has description elements from both KML and Dublin Core. Starting with the same skeleton program as before, write a program to separately extract and print out the description metadata from KML and from DC.

Tip
The KML namespace is "`http://earth.google.com/kml/2.0`", and the DC namespace is "`http://purl.org/dc/elements/1.1/`".

Handling multiple character events
SAX interfaces have a small, unavoidable, but very irritating, peculiarity with respect to character handlers. Namely, a given stretch of character data may be reported as multiple chunks. That is, in reading the following document,

```
<tag> 123456789 </tag>
```

a SAX interface may report 4 events:

- `startElement(tag)`
- `characters(" 12345")`
- `characters("6789 ")`
- `endElement(tag)`

or indeed any number of `character` events that will sequentially produce the same string.
You will have noticed this in this exercise - reading in the Dublin Core description, FoX returned three character events for one string.
Thus, to correctly handle a SAX interface, any program you write must do the following on encountering a `character` event.

- Remember the last event - was it a `character` event? If so, append current character data to a string buffer.

and it must not process the string buffer until it is sure that there are no more `character` events coming - which will happen at the next `startElement` or `endElement` event.
There is an example in `exercise_4_4b.f90` which illustrates the control flow necessary to handle this correctly, and a generic example as the skeleton for `exercise_4_5`.

5 Exercise 4-5: Extracting and presenting data from QuakeML

For this exercise, the input file is the final output from `hypoDD` as generated yesterday. It is often useful to take data from an XML format, and reformat it for human consumption in a more familiar way.

In exercise 4-4 you learnt how to use module variables to keep track of where you were in the document. You probably used a simple logical switch to see if you were in the correct element. However, that approach gets quite unwieldy when large numbers of tags are in evidence. In such cases, it is useful to use a "state" variable, which is a module variable which takes on different values according to the state of the parser - *i.e.* where it is in the document. For example, you could assign the values:

- `0`: not in an interesting tag.
- `1`: in a latitude tag
- `2`: in a longitude tag

etc.

Then, when receiving a `characters` event, you can simply check the value of the state variable to choose what to do with the data.

Starting from the same skeleton SAX program, write a program to read data about the first ten earthquakes, and produce a table for presentation something like this:

```
ID      | LATITUDE           | LONGITUDE           | DEPTH     | MAGNITUDE
16484   |3.728530120850e1    |-1.216628036499e2    |3.60000e0  |6.30000e0
16527   |3.728570175171e1    |-1.216644973755e2    |3.50000e0  |6.01000e0
17496   |3.729219818115e1    |-1.216661987305e2    |1.20000e0  |6.61000e0
16521   |3.728530120850e1    |-1.216681976318e2    |3.00000e0  |3.82000e0
17842   |3.728829956055e1    |-1.216688003540e2    |1.89999e0  |3.73000e0
16635   |3.729570007324e1    |-1.216701965332e2    |1.40000e0  |5.90000e0
16576   |3.728699874878e1    |-1.216628036499e2    |1.60000e0  |5.79000e0
17929   |3.728919982910e1    |-1.216658020020e2    |1.80000e0  |5.24000e0
16659   |3.728929901123e1    |-1.216651992798e2    |1.50000e0  |6.30000e0
16701   |3.729169845581e1    |-1.216662979126e2    |1.60000e0  |6.06000e0
```

Tip
In addition to the state variable, you'll need to keep account of how many values you've read - use another module variable for that.

Converting strings to numbers

```
read(string,*) number
```

Alternatively, FoX provides an XML-aware shorthand which works for arrays and matrices as well:

```
call rts(string_in, number_out)
```

Query: how might you go about doing this if you didn't know ahead of time the number of data items you wanted to read?

6 Exercise 4-6: Perform a calculation

Above and beyond simple presentation, a real program will probably want to manipulate the data it's read in. This presents a new problem though - so far, we've been treating the data only as strings, but if you want to manipulate it, you'll need to treat it as numerical data.

Using your answer from exercise 4-5, at the end of the run, once you've collected all the data for presentation, calculate the centroid of the earthquake locations.

Note
The centroid of a set of points in 2D space is defined as that point whose x-coordinate is the average of the x-coordinates of all the points, and whose y-coordinate is the average of the y-coordinates of all the points.

You could extend your program so that, instead of getting the centroid for just the first ten points, it calculates the centroid of all the points in the first KML `Folder`. To do this, you need to keep track of the `startElement` events for the `Folders` as well.

Note
You won't know how many points are in the Folder until you've finished reading it. So instead of storing all the points until the end, you can calculate a progressive average as you read through the document.

7 Exercise 4-8

In the previous exercise, all the data you needed was found in the first half of the document. Once you'd retrieved the data you needed, you simply set the state variables so that when you received any further events, no action was taken.

This means that even though you'd finished, the parser carried on going, and read through the document to the end. If this was a particularly large document, then that might involve wasting a lot of time reading data you don't care about.

So, the SAX parser will let you terminate processing of the document at any point, though in a slightly awkward way. This can be done from any of the callback functions - you simply need to do:

```
call stop_parser(xf)
```

where `xf` is the same `xf` that you have called the parser on. This has the effect that the rest of the current callback subroutine will be executed, but as soon as it finishes, the parser will finish, and control will be returned to the main program. That is simple enough - however, you need to rearrange your program.

If you try and put this in one of your callback subroutines, you'll realise that the arguments to the subroutine don't include any way to get at the parser. So what you need to do is have the parser object accessible some other way.

The best way to do this is have your parser object be part of the `m_handlers` module, as a module variable. This gives the module subroutines access to the variable. Your main program can then import that variable by a `use` statement, in order to be able to call `parse` on it.

If you look at `exercise_4_7.f90`, you'll find your skeleton program rearranged into this form. Now you can call `stop_parser(xf)` at any time from within the callback subroutines.

Implement your solutions to exercise 4-6 in this new skeleton; and call `stop_parser` as soon as you've collected all your data; in the first case because you've collected 10 earthquakes; in the second case because you've collected *all* of the initial earthquakes.

8 Exercise 4-8: Teaching `hypoDD` to read QuakeML

You are probably not going to get this far through the practical session with much time to spare; but having got here, you understand how to read data from an XML file into Fortran data structures.

`hypoDD` could be adapted, instead of reading its input data from text files as it does now, to read its data from XML files of the sort we have been dealing with. Think about how this might be done, and look at the `hypoDD` source code again. Note that earthquake event data is read in at about line 150 of `getdata.f`.

9 Conclusions

This practical was designed to do three things

- Understand how to program against a callback API.
- Understand how to program with the FoX SAX API.
- Understand how to use a SAX API to build up data structures from an XML document.

It is important to realise, though, that although SAX is a conceptually simple API to work with, it becomes rapidly very complicated to use on complicated XML documents with lots of variation in structure, many different child elements and attributes etc. It was usable in this case only because our XML format is simple and predictable.

It is also important to understand that we could only do this because we knew the format of the XML we were expecting, so we didn't do any checking that the XML was of the right structure, we just assumed it. In XML-reading applications which will accept input from a wider variety of sources, much more robust error checking is necessary.

In either case, it might well be more sensible to do the XML input in a higher-level-language, using higher-level APIs, like XPath, as you were shown in Practical 3, or DOM as you will be shown in Practical 5.