xmlf90: A parser for XML in Fortran90

Alberto García
Departamento de Física de la Materia Condensada
Facultad de Ciencia y Tecnología
Universidad del País Vasco
Apartado 644 , 48080 Bilbao, Spain
http://lcdx00.wm.lc.ehu.es/ag/xml/

30 January 2004 — xmlf90 Version 1.1

1 Introduction

NOTE: This version of the User Guide and Tutorial does not cover either the WXML printing library or the new DOM API conceived by Jon Wakelin. See the html reference material and the relevant example subdirectories.

This tutorial documents the user interface of `xmlf90`, a native Fortran90 XML parser. The parser was designed to be a useful tool in the extraction and analysis of data in the context of scientific computing, and thus the priorities were efficiency and the ability to deal with very large XML files while maintaining a small memory footprint. There are two programming interfaces. The first is based on the very successful SAX (Simple API for XML) model: the parser calls routines provided by the user to handle certain events, such as the encounter of the beginning of an element, or the end of an element, or the reading of character data. The other is based on the XPATH standard. Only a very limited set of the full XPATH specification is offered, but it is already quite useful.
Some familiarity of XML is assumed. Apart from the examples discussed in this tutorial (chosen for their simplicity), the interested reader can refer to the `Examples/` directory in the `xmlf90` distribution.

2 The SAX interface

2.1 A simple example

To illustrate the working of the SAX interface, consider the following XML snippet

```
<item id="003">
   <description>Washing machine</description>
   <price currency="euro">1500.00</price>
</item>
```

When the parser processes this snippet, it carries out the sequence of calls:

1. call to `begin_element_handler` with name=”item” and attributes=(Dictionary with the pair (id,003))

2. call to `begin_element_handler` with name="description" and an empty attribute dictionary.
3. call to `pcdata_chunk_handler` with pcdata="Washing machine"
4. call to `end_element_handler` with name="description"
5. call to `begin_element_handler` with name="price" and attributes=(Dictionary with the pair (currency,euro))
6. call to `pcdata_chunk_handler` with pcdata="1500.00"
7. call to `end_element_handler` with name="price"
8. call to `end_element_handler` with name="item"

The handler routines are written by the user and passed to the parser as procedure arguments. A simple program that parses the above XML fragment (assuming it resides in file *inventory.xml*) and prints out the names of the elements and any *id* attributes as they are found, is:

```
program simple
use flib_sax

type(xml_t)        :: fxml      ! XML file object (opaque)
integer            :: iostat    ! Return code (0 if OK)

call open_xmlfile("inventory.xml",fxml,iostat)
if (iostat /= 0) stop "cannot open xml file"

call xml_parse(fxml, begin_element_handler=begin_element_print)

contains !---------------- handler subroutine follows

subroutine begin_element_print(name,attributes)
   character(len=*), intent(in)     :: name
   type(dictionary_t), intent(in)   :: attributes

   character(len=3)  :: id
   integer           :: status

   print *, "Start of element: ", name
   if (has_key(attributes,"id")) then
    call get_value(attributes,"id",id,status)
print *, "  Id attribute: ", id
   endif
end subroutine begin_element_print

end program simple
```

To access the XML parsing functionality, the user only needs to `use` the module `flib_sax`, open the XML file, and call the main routine `xml_parse`, providing it with the appropriate event handlers.
The subroutine interfaces are:

```
subroutine open_xmlfile(fname,fxml,iostat)
character(len=*), intent(in)  :: fname     ! File name
type(xml_t), intent(out)      :: fxml      ! XML file object (opaque)
integer, intent(out   )       :: iostat    ! Return code (0 if OK)


subroutine xml_parse(fxml,                    &
                     begin_element_handler,   &
                     end_element_handler,     &
                     pcdata_chunk_handler ....
                     .... MORE OPTIONAL HANDLERS  )
```

The handlers are OPTIONAL arguments (in the above example we just specify `begin_element_handler`). If no handlers are given, nothing useful will happen, except that any errors are detected and reported. The interfaces for the most useful handlers are:

```
subroutine begin_element_handler(name,attributes)
character(len=*), intent(in)     :: name
type(dictionary_t), intent(in)   :: attributes
end subroutine begin_element_handler

subroutine end_element_handler(name)
character(len=*), intent(in)     :: name
end subroutine end_element_handler

subroutine pcdata_chunk_handler(chunk)
character(len=*), intent(in) :: chunk
end subroutine pcdata_chunk_handler
```

The attribute information in an element tag is represented as a dictionary of name/value pairs, held in a `dictionary_t` abstract type. The information in it can be accessed through a set of dictionary methods such as `has_key` and `get_value` (full interfaces to be found in Sect. 5).

2.2 Monitoring the sequence of events

The above example is too simple and not very useful if what we want is to extract information in a coherent manner. For example, assume we have a more complete inventory of appliances such as

```
<inventory>
<item id="003">
   <description>Washing machine</description>
   <price currency="euro">1500.00</price>
</item>
<item id="007">
   <description>Microwave oven</description>
   <price currency="euro">300.00</price>
</item>
<item id="011">
   <description>Dishwasher</description>
```

```
   <price currency="swedish crown">10000.00</price>
</item>
</inventory>
```

and we want to print the items with their prices in the form:

```
003 Washing machine : 1500.00 euro
007 Microwave oven : 300.00 euro
011 Dishwasher : 10000.00 swedish crown
```

We begin by writing the following module

```
module m_handlers
use flib_sax
private
public :: begin_element, end_element, pcdata_chunk
!
logical, private            :: in_item, in_description, in_price
character(len=40), private  :: what, price, currency, id
!
contains !-----------------------------------------
!
subroutine begin_element(name,attributes)
   character(len=*), intent(in)     :: name
   type(dictionary_t), intent(in)   :: attributes

   integer  :: status

   select case(name)
     case("item")
       in_item = .true.
       call get_value(attributes,"id",id,status)

     case("description")
       in_description = .true.

     case("price")
       in_price = .true.
       call get_value(attributes,"currency",currency,status)

   end select

end subroutine begin_element
!-------------------------------------------------------------
subroutine pcdata_chunk_handler(chunk)
   character(len=*), intent(in) :: chunk

   if (in_description) what = chunk
   if (in_price) price = chunk

end subroutine pcdata_chunk_handler
!-------------------------------------------------------------
```

```
subroutine end_element(name)
   character(len=*), intent(in)     :: name

   select case(name)
     case("item")
       in_item = .false.
       write(unit=*,fmt="(5(a,1x))") trim(id), trim(what), ":", &
                                     trim(price), trim(currency)

     case("description")
       in_description = .false.

     case("price")
       in_price = .false.

   end select

end subroutine end_element
!-----------------------------------------------------------------
end module m_handlers
```

PCDATA chunks are passed back as simple fortran character variables, and we assign them to `what` or `price` depending on the context, which we monitor through the logical variables `in_description, in_price`, updated as we enter and leave different elements. (The variable `in_item` is not strictly necessary.)
The program to parse the file just needs to use the functionality in the module `m_handlers`:

```
program inventory
use flib_sax
use m_handlers

type(xml_t)        :: fxml      ! XML file object (opaque)
integer            :: iostat

call open_xmlfile("inventory.xml",fxml,iostat)
if (iostat /= 0) stop "cannot open xml file"

call xml_parse(fxml, begin_element_handler=begin_element, &
                     end_element_handler=end_element,     &
                     pcdata_chunk_handler=pcdata_chunk )

end program inventory
```

2.2.1 Exercises

1. Code the above fortran files and the XML file in your computer. Compile and run the program and check that the output is correct. (Compilation instructions are provided in Sect. 8).

2. Edit the XML file and remove one of the `</item>` lines. What happens? This is an example of a *mal-formed* XML file. The parser can detect it and complain about it.

3. Edit the XML file and remove the `currency` attribute from one of the elements. What happens? In this case, the parser cannot detect the missing attribute (it is not a *validating parser*). However, it could be possible for the user to detect early that something is wrong by checking the value of the `status` variable after the call to `get_value`.

4. Modify the program to print the prices in euros (1 euro buys approximately 9.2 swedish crowns).

2.3 Other tags and their handlers

The parser can also process comments, XML declarations (formally known as "processing instructions"), and SGML declarations, although the latter two are not acted upon in any way (in particular, no attempt at validation of the XML document is done).

- An **empty element** tag of the form

  ```
  <name att="value"...  />
  ```

  can be handled as successive calls to `begin_element_handler` and `end_element_handler`. However, if the optional handler `empty_element_handler` is present, it is called instead. Its interface is exactly the same as that of `begin_element_handler`:

  ```
  subroutine empty_element_handler(name,attributes)
  character(len=*), intent(in)     :: name
  type(dictionary_t), intent(in)   :: attributes
  end subroutine empty_element_handler
  ```

- **Comments** are sections of the XML file contained between the markup `<!--` and `-->`, and are handled by the optional argument `comment_handler`

  ```
  subroutine comment_handler(comment)
  character(len=*), intent(in) :: comment
  end subroutine comment_handler
  ```

- **XML declarations** can be processed in the same way as elements, with the "target" being the element name, etc. For example, in

  ```
  <?xml version="1.0"?>
  ```

  xml would be the "element name", *version* an attribute name, and *1.0* its value. The optional handler interface is:

  ```
  subroutine xml_declaration_handler(name,attributes)
  character(len=*), intent(in)     :: name
  type(dictionary_t), intent(in)   :: attributes
  end subroutine xml_declaration_handler
  ```

- **SGML declarations** such as entity declarations or doctype specifications are treated basically as comments. Interface:

  ```
  subroutine sgml_declaration_handler(sgml_declaration)
  character(len=*), intent(in) :: sgml_declaration
  end subroutine sgml_declaration_handler
  ```

In the current version of the parser, overly long comments and SGML declarations might be truncated.

3 The XPATH interface

NOTE: The current implementation gets its inspiration from XPATH, but by no means it is a complete, or even a subset, implementation of the standard. Since it is built on top of the SAX interface, it uses a "stream" paradigm which is completely alien to the XPATH specification. It is nevertheless still quite useful. The author is open to suggestions to refine the interface.

This API is based on the concept of an XML path. For example:

```
/inventory/item
```

represents a 'item' element which is a child of the root element 'inventory'. Paths can contain special wildcard markers such as `//` and `*`. The following are examples of valid paths:

```
  //a       : Any occurrence of element 'a', at any depth.
  /a/*/b    : Any 'b' which is a grand-child of 'a'
  ./a       : A relative path (with respect to the current path)
  a         : (same as above)
  /a/b/./c  : Same as /a/b/c (the dot (.) is a dummy)
  //*       : Any element.
  //a/*//b  : Any 'b' under any children of 'a'.
```

3.1 Simple example

Using the XPATH interface it is possible to search for any element directly, and to recover its attributes or character content. For example, to print the names of all the appliances in the inventory:

```
program simple
use flib_xpath

type(xml_t) :: fxml

integer  :: status
character(len=100)  :: what

call open_xmlfile("inventory.xml",fxml,status)
!
do
      call get_node(fxml,path="//description",pcdata=what,status=status)
      if (status < 0)   exit
      print *, "Appliance: ", trim(what)
enddo
end program simple
```

Repeated calls to `get_node` return the character content of the 'description' elements (at any depth). We exit the loop when the `status` variable is negative on return from the call. This indicates that there are no more elements matching the `//description` path pattern.[1]
Apart from path patterns, we can narrow our search by specifying conditions on the attribute list of the element. For example, to print only the prices which are given in euros we can use the `att_name` and `att_value` optional arguments:

[1] Returning a negative value for an end-of-file or end-or-record condition follows the standard practice. Positive return values signal malfunctions

```
program euros
use flib_xpath

type(xml_t) :: fxml

integer  :: status
character(len=100)  :: price

call open_xmlfile("inventory.xml",fxml,status)
!
do
  call get_node(fxml,path="//price", &
                att_name="currency",att_value="euro", &
                pcdata=price,status=status)
  if (status < 0)   exit
  print *, "Price (euro): ", trim(price)
enddo
end program euros
```

We can zero in on any element in this fashion, but we apparently give up the all-important context. What happens if we want to print *both* the appliance description and its price?

```
program twoelements
use flib_xpath

type(xml_t) :: fxml

integer  :: status
character(len=100)  :: what, price, currency

call open_xmlfile("inventory.xml",fxml,status)
!
do
  call get_node(fxml,path="//description", &
                pcdata=what,status=status)
  if (status < 0)   exit                  ! No more items
  !
  ! Price comes right after description...
  !
  call get_node(fxml,path="//price", &
                attributes=attributes,pcdata=price,status=status)
  if (status /= 0) stop "missing price element!"

  call get_value(attributes,"currency",currency,status)
  if (status /= 0) stop "missing currency attribute!"

  write(unit=*,fmt="(6a)") "Appliance: ", trim(what), &
                           ". Price: ", trim(price), " ", trim(currency)
enddo
end program twoelements
```

3.1.1 Exercises

1. Modify the above programs to print only the appliances priced in euros.

2. Modify the order of the 'description' and 'price' elements in a item. What happens to the 'twoelements' program output?

3. The full XPATH specification allows the query for a particular element among a set of elements with the same path, based on the ordering of the element. For example, "/inventory/item[2]" will refer to the second 'item' element in the XML file. Write a routine that implements this feature and returns the element's attribute dictionary.

4. Queries for paths can be issued in any order, and so some mechanism for "rewinding" the XML file is necessary. It is provided by the appropriately named `rewind_xmlfile` subroutine (see full interface in the Reference section). Use it to implement a silly program that prints items from the inventory at random. (Extra points for including logic to minimize the number of rewinds.)

3.2 Contexts and restricted searches

The logic of the `twoelements` program in the previous section follows from the assumption that the 'price' element follows the 'description' element in a typical 'item'. If the DTD says so, and the XML file is valid (in the technical sense of conforming to the DTD), the assumption should be correct. However, since the parser is non-validating, it might be unreasonable to expect the proper ordering in all cases. What we should expect (as a minimum) is that both the price and description elements are children of the 'item' element. In the following version we make use of the **context** concept to achieve a more robust solution.

```
program item_context
use flib_xpath

type(xml_t) :: fxml, contex

integer  :: status
character(len=100)  :: what, price, currency

call open_xmlfile("inventory.xml",fxml,status)
!
do
  call mark_node(fxml,path="//item",status=status)
  if (status < 0)   exit      ! No more items
  context = fxml               ! Save item context
  !
  ! Search relative to context
  !
  call get_node(fxml,path="price", &
                attributes=attributes,pcdata=price,status=status)
  call get_value(attributes,"currency",currency,status)
  if (status /= 0) stop "missing currency attribute!"
  !
  ! Rewind to beginning of context
  !
```

```
  fxml = context
  call sync_xmlfile(fxml)
  !
  ! Search relative to context
  !
  call get_node(fxml,path="description",pcdata=what,status=status)
  write(unit=*,fmt="(6a)") "Appliance: ", trim(what), &
                           ". Price: ", trim(price), " ", trim(currency)
enddo
end program item_context
```

The call to `mark_node` positions the parser's file handle `fxml` right after the end of the starting tag of the next 'item' element. We save that position as a "context marker" to which we can return later on. The calls to `get_node` use path patterns that do not start with a `/`: they are **searches relative to the current context**. After getting the information about the 'price' element, we restore the parser's file handle to the appropriate position at the beginning of the 'item' context, and search for the 'description' element. In the following iteration of the loop, the parser will find the next 'item' element, and the process will be repeated until there are no more 'item's.
Contexts come in handy to encapsulate parsing tasks in re-usable subroutines. Suppose you are going to find the basic 'item' element content in a whole lot of different XML files. The following subroutine extracts the description and price information:

```
subroutine get_item_info(context,what,price,currency)
type(xml_t), intent(in)       :: contex
character(len=*), intent(out) :: what, price, currency

!
! Local variables
!
type(xml_t)        :: ff
integer            :: status
type(dictionary_t) :: attributes

  !
  ! context is read-only, so make a copy and sync just in case
  !
  ff = context
  call sync_xmlfile(ff)
  !
  call get_node(ff,path="price", &
                attributes=attributes,pcdata=price,status=status)
  call get_value(attributes,"currency",currency,status)
  if (status /= 0) stop "missing currency attribute!"
  !
  ! Rewind to beginning of context
  !
  ff = context
  call sync_xmlfile(ff)
  !
  call get_node(ff,path="description",pcdata=what,status=status)
```

```
end subroutine get_item_info
```

Using this routine, the parsing is much more compact:

```
program item_context
use flib_xpath

type(xml_t) :: fxml

integer  :: status
character(len=100)  :: what, price, currency

call open_xmlfile("inventory.xml",fxml,status)
!
do
  call mark_node(fxml,path="//item",status=status)
  if (status /= 0)   exit      ! No more items
  call get_item_info(fxml,what,price,currency)
  write(unit=*,fmt="(6a)") "Appliance: ", trim(what), &
                           ". Price: ", trim(price), " ", trim(currency)
  call sync_xmlfile(fxml)
enddo
end program item_context
```

It is extremely important to understand the meaning of the call to `sync_xmlfile`. The file handle `fxml` holds parsing context **and** a physical pointer to the file position (basically a variable counting the number of characters read so far). When the context is passed to the subroutine and the parsing carried out, the context and the file position get out of sync. Synchronization means to re-position the physical file pointer to the place where it was when the context was first created.

3.2.1 Exercises

1. Modify the above programs to print only the appliances priced in euros.
2. Write a program that prints only the most expensive item. (Assume that the inventory is very large and it is not feasible to hold everything in memory...)
3. Use the `get_item_info` subroutine to print descriptions and price information from the following XML file:

```
<vacations>
<trip>
   <description>Mediterranean cruise</description>
   <price currency="euro">1500.00</price>
</trip>
<trip>
   <description>Week in Majorca</description>
   <price currency="euro">300.00</price>
</trip>
<trip>
```

```
   <description>Wilderness Route</description>
   <price currency="swedish crown">10000.00</price>
</trip>
</vacations>
```

(Note that the routine does not care what the context name is (it could be 'item' or 'trip'). It is only the fact that the children ('description' and 'price') are the same that matters.

4 Handling of scientific data

4.1 Numerical datasets

While the ASCII form is not the most efficient for the storage of numerical data, the portability and flexibility offered by the XML format makes it attractive for the interchange of scientific datasets. There are a number of efforts under way to standardize this area, and presumably we will have nifty tools for the creation and visualization of files in the near future. Even then, however, it will be necessary to be able to read numerical information into fortran programs. The `xmlf90` package offers limited but useful functionality in this regard, making it possible to build numerical arrays on the fly as the XML file containing the data is parsed. As an example, consider the dataset:

```
<data>
 8.90679398599 8.90729421510 8.90780189594 8.90831710494
 8.90883991832 8.90937041202 8.90990866166 8.91045474255
 8.91100872963 8.91157069732 8.91214071958 8.91271886986
 8.91330522098 8.91389984506 8.91450281355 8.91511419713
 8.91573406560 8.91636248785 8.91699953183 8.91764526444
 8.91829975142 8.91896305734 8.91963524555 8.92031637799
 8.92100651514 8.92170571605 8.92241403816 8.92313153711
 8.92385826683 8.92459427943 8.92533962491 8.92609435120
 8.92685850416 8.92763212726 8.92841526149 8.92920794545
</data>
```

and the following fragment of a `m_handlers` module for SAX parsing:

```
real, dimension(1000)  :: x    ! numerical array to hold data

subroutine begin_element(name,attributes)
 ...
   select case(name)
     case("data")
       in_data = .true.
       ndata = 0
     ...
   end select

end subroutine begin_element
!------------------------------------------------------------
subroutine pcdata_chunk_handler(chunk)
   character(len=*), intent(in) :: chunk
```

```
   if (in_data) call build_data_array(chunk,x,ndata)
   ...

end subroutine pcdata_chunk_handler
!------------------------------------------------------------
subroutine end_element(name)
 ...
   select case(name)
     case("data")
       in_data = .false.
       print *, "Read ", ndata, " data elements."
       print *, "X: ", x(1:ndata)
     ...
   end select

end subroutine end_element
```

When the `<data>` tag is encountered by the parser, the variable `ndata` is initialized. Any PCDATA chunks found from then on and until the `</data>` tag is seen are passed to the `build_data_array` generic subroutine, which converts the character data to the numerical format (integer, default real, double precision) implied by the array `x`. The array is filled with data and the `ndata` variable increased accordingly.
If the data is known to represent a multi-dimensional array (something that could be encoded in the XML as attributes to the 'data' element, for example), the user can employ the fortran `reshape` intrinsic to obtain the final form.
There is absolutely no limit to the size of the data (apart from filesystem size and total memory constraints) since the parser only holds in memory at any given time a small chunk of character data (the default is to split the character data stream and call the `pcdata_chunk_handler` routine at the end of a line, or at the end of a token if the line is too long). This is one of the most useful features of the SAX approach to XML parsing.
In order to read numerical data with the XPATH interface in its current implementation, one must first read the PCDATA into the `pcdata` optional argument of `get_node`, and then call `build_data_array`. However, there is an internal limit to the size of the PCDATA buffer, so this method cannot be safely used for large datasets at this point. In a forthcoming version there will be a generic subroutine `get_node` with a `data` numerical array optional argument which will be filled by the parser on the fly.

4.1.1 Exercises

1. Generate an XML file containing a large dataset, and write a program to read the information back. You might want to include somewhere in the XML file information about the number of data elements, so that an array of the proper size can be used.

2. Devise a strategy to read a dataset without knowing in advance the number of data elements. (Some possibilities: re-sizable allocatable arrays, two-pass parsing...).

3. Suggest a possible encoding for the storage of two-dimensional arrays, and write a program to read the information from the XML file and create the appropriate array.

4. Write a program that could read a 10Gb Monte Carlo simulation dataset and print the average and standard deviation of the data. (We are not advocating the use of XML for such large datasets. NetCDF would be much more efficient in this case).

4.2 Mapping of XML elements to derived types

After the parsing, the data has to be put somewhere. A good strategy to handle structured content is to try to replicate it within data structures inside the user program. For example, an element of the form

```
<table units="nm" npts="100">
<description>Cluster diameters</description>
<data>
2.3 4.5 5.6 3.4 2.3 1.2 ...
...
...
</data>
</table>
```

could be mapped onto a derived type of the form:

```
type :: table
  character(len=50)            :: description
  character(len=20)            :: units
  integer                      :: npts
  real, dimension(:), pointer  :: data
end type table
```

There could even be parsing and output subroutines associated to this derived type, so that the user can handle the XML production and reading transparently. Directory `Examples/` in the `xmlf90` distribution contains some code along these lines.

4.2.1 Exercises

1. Study the `pseudo` example in `Examples/sax/` and `Examples/xpath/`. Now, with your own application in mind, write derived-type definitions and parsing routines to handle your XML data (which would also need to be *designed* somehow).

5 REFERENCE: Subroutine interfaces

5.1 Dictionary handling

Attribute lists are handled as instances of a derived type `dictionary_t`, loosely inspired by the Python type. The terminology is more general: keys and entries instead of names and attributes.

- ```
  function number_of_entries(dict) result(n)
  !
  ! Returns the number of entries in the dictionary
  !
  type(dictionary_t), intent(in)   :: dict
  integer                          :: n
  ```

- ```
  function has_key(dict,key) result(found)
  !
  ! Checks whether there is an entry with
  ! the given key in the dictionary
  !
  ```

```
type(dictionary_t), intent(in)   :: dict
character(len=*), intent(in)     :: key
logical                          :: found
```

- ```
  subroutine get_value(dict,key,value,status)
  !
  ! Gets values by key
  !
  type(dictionary_t), intent(in)            :: dict
  character(len=*), intent(in)              :: key
  character(len=*), intent(out)             :: value
  integer, intent(out)                      :: status
  ```

- ```
  subroutine get_key(dict,i,key,status)
  !
  ! Gets keys by their order in the dictionary
  !
  type(dictionary_t), intent(in)            :: dict
  integer, intent(in)                       :: i
  character(len=*), intent(out)             :: key
  integer, intent(out)                      :: status
  ```

- ```
  subroutine print_dict(dict)
  !
  ! Prints the contents of the dictionary to stdout
  !
  type(dictionary_t), intent(in)   :: dict
  ```

5.2 SAX interface

- ```
  subroutine open_xmlfile(fname,fxml,iostat)
  !
  ! Opens the file "fname" and creates an xml handle fxml
  ! iostat /= 0 on error.
  !
  character(len=*), intent(in)      :: fname
  integer, intent(out)              :: iostat
  type(xml_t), intent(out)          :: fxml
  ```

- ```
  subroutine xml_parse(fxml, begin_element_handler,    &
                             end_element_handler,      &
                             pcdata_chunk_handler,     &
                             comment_handler,          &
                             xml_declaration_handler,  &
                             sgml_declaration_handler, &
                             error_handler,            &
                             signal_handler,           &
                             verbose,                  &
                             empty_element_handler)

  type(xml_t), intent(inout), target  :: fxml
  ```
  ```

```
optional                              :: begin_element_handler
optional                              :: end_element_handler
optional                              :: pcdata_chunk_handler
optional                              :: comment_handler
optional                              :: xml_declaration_handler
optional                              :: sgml_declaration_handler
optional                              :: error_handler
optional                              :: signal_handler  ! see XPATH code
logical, intent(in), optional         :: verbose
optional                              :: empty_element_handler
```

- Interfaces for handlers follow:

```
subroutine begin_element_handler(name,attributes)
character(len=*), intent(in)     :: name
type(dictionary_t), intent(in)   :: attributes
end subroutine begin_element_handler

subroutine end_element_handler(name)
character(len=*), intent(in)     :: name
end subroutine end_element_handler

subroutine pcdata_chunk_handler(chunk)
character(len=*), intent(in) :: chunk
end subroutine pcdata_chunk_handler

subroutine comment_handler(comment)
character(len=*), intent(in) :: comment
end subroutine comment_handler

subroutine xml_declaration_handler(name,attributes)
character(len=*), intent(in)     :: name
type(dictionary_t), intent(in)   :: attributes
end subroutine xml_declaration_handler

subroutine sgml_declaration_handler(sgml_declaration)
character(len=*), intent(in) :: sgml_declaration
end subroutine sgml_declaration_handler

subroutine error_handler(error_info)
type(xml_error_t), intent(in)            :: error_info
end subroutine error_handler

subroutine signal_handler(code)
logical, intent(out) :: code
end subroutine signal_handler

subroutine empty_element_handler(name,attributes)
character(len=*), intent(in)     :: name
```

```
type(dictionary_t), intent(in)   :: attributes
end subroutine empty_element_handler
```

Other file handling routines (some of them really only useful within the XPATH interface):

- subroutine REWIND_XMLFILE(fxml)

```
!
! Rewinds the physical file associated to fxml and clears the data
! structures used in parsing.
!
type(xml_t), intent(inout) :: fxml
```

- subroutine SYNC_XMLFILE(fxml,status)

```
!
! Synchronizes the physical file associated to fxml so that reading
! can resume at the exact point in the parsing saved in fxml.
!
type(xml_t), intent(inout) :: fxml
integer, intent(out)       :: status
```

- subroutine CLOSE_XMLFILE(fxml)

```
!
! Closes the file handle fmxl (and the associated OS file object)
!
type(xml_t), intent(inout) :: fxml
```

5.3 XPATH interface

- subroutine MARK_NODE(fxml,path,att_name,att_value,attributes,status)

```
!
! Performs a search of a given element (by path, and/or presence of
! a given attribute and/or value of that attribute), returning optionally
! the element's attribute dictionary, and leaving the file handle fxml
! ready to process the rest of the element's contents (child elements
! and/or pcdata).
!
! Side effects: it sets a "path_mark" in fxml to enable its use as a
! context.
!
! If the argument "path" is present and evaluates to a relative path (a
! string not beginning with "/"), the search is interrupted after the end
! of the "ancestor_element" set by a previous call to "mark_node".
! If not earlier, the search ends at the end of the file.
!
! The status argument, if present, will hold a return value,
! which will be:
!
!   0 on success,
!   negative in case of end-of-file or end-of-ancestor-element, or
!   positive in case of other malfunction
```

```
!
type(xml_t), intent(inout), target           :: fxml
character(len=*), intent(in), optional       :: path
character(len=*), intent(in), optional       :: att_name
character(len=*), intent(in), optional       :: att_value
type(dictionary_t), intent(out), optional    :: attributes
integer, intent(out), optional               :: status
```

- subroutine GET_NODE(fxml,path,att_name,att_value,attributes,pcdata,status)

```
!
! Performs a search of a given element (by path, and/or presence of
! a given attribute and/or value of that attribute), returning optionally
! the element's attribute dictionary and any PCDATA characters contained
! in the element's scope (but not child elements). It leaves the file handle
! physically and logically positioned:
!
!     after the end of the element's start tag if 'pcdata' is not present
!     after the end of the element's end tag if 'pcdata' is present
!
! If the argument "path" is present and evaluates to a relative path (a
! string not beginning with "/"), the search is interrupted after the end
! of the "ancestor_element" set by a previous call to "mark_node".
! If not earlier, the search ends at the end of the file.
!
! The status argument, if present, will hold a return value,
! which will be:
!
!   0 on success,
!   negative in case of end-of-file or end-of-ancestor-element, or
!   positive in case of a malfunction (such as the overflow of the
!   user's pcdata buffer).
!
type(xml_t), intent(inout), target           :: fxml
character(len=*), intent(in), optional       :: path
character(len=*), intent(in), optional       :: att_name
character(len=*), intent(in), optional       :: att_value
type(dictionary_t), intent(out), optional    :: attributes
character(len=*), intent(out), optional      :: pcdata
integer, intent(out), optional               :: status
```

5.4 PCDATA conversion routines

- subroutine build_data_array(str,x,n)

```
!
! Incrementally builds the data array x from
! character data contained in str. n holds
! the number of entries of x set so far.
!
character(len=*), intent(in)                ::  str
NUMERIC TYPE, dimension(:), intent(inout)   ::    x
integer, intent(inout)                      ::    n
```

```
!
! NUMERIC TYPE can be any of:
!            integer
!            real
!            real(kind=selected_real_kind(14))
!
```

5.5 Other utility routines

- ```
  function xml_char_count(fxml) result (nc)
  !
  ! Provides the value of the processed-characters counter
  !
  type(xml_t), intent(in)          :: fxml
  integer                          :: nc

  nc = nchars_processed(fxml%fb)

  end function xml_char_count
  ```

6 Other parser features, limitations, and design issues

6.1 Features

- The parser can detect badly formed documents, giving by default an error report including the line and column where it happened. It also will accept an `error_handler` routine as another optional argument, for finer control by the user. In the SAX interface, if the optional logical argument "verbose" is present and it is ".true.", the parser will offer detailed information about its inner workings. In the XPATH interface, there are a pair of routines, `enable_debug` and `disable_debug`, to control verbosity. See `Examples/xpath/` for examples.
- It ignores PCDATA outside of element context (and warns about it)
- Attribute values can be specified using both single and double quotes (as per the XML specs).
- It processes the default entities: > &amp; < &apos; and &quot; and decimal and hex character entities (for example: &#123; &#4E;). The processing is not "on the fly", but after reading chunks of PCDATA.
- Understands and processes CDATA sections (transparently passed as PCDATA to the handler).

See `Examples/sax/features` for an illustration of the above features.

6.2 Limitations

- It is not a validating parser.
- It accepts only single-byte encodings for characters.
  ```

- Currently, there are hard-wired limits on the length of element and attribute identifiers, and the length of attribute values and unbroken (i.e., without whitespace) PCDATA sections. The limit is set in `sax/m_buffer.f90` to `MAX_BUFF_SIZE=300`.
- Overly long comments and SGML declarations can also be truncated, but the effect is currently harmless since the parser does not make use of that information. In a future version there could be a more robust retrieval mechanism.
- The number of attributes is limited to `MAX_ITEMS=20` in `sax/m_dictionary.f90`:
- In the XPATH interface, returned PCDATA character buffers cannot be larger than an internal size of `MAX_PCDATA_SIZE=65536` set in `xpath/m_path.f90`

6.3 Design Issues

See {`sax,xpath`}`/Developer.Guide`.
The parser is actually written in the `F` subset of Fortran90, for which inexpensive compilers are available. (See `http://fortran.com/imagine1/`).
There are two other projects aimed at parsing XML in Fortran: those of Mart Rentmeester (`http://nn-online.sci.kun.nl/fortran/`) and Arjen Markus (`http://xml-fortran.sourceforge.net/`). Up to this point the three projects have progressed independently, but it is anticipated that there will be a pooling of efforts in the near future.

7 Installation Instructions

There is extensible built-in support for arbitrary compilers. The setup discussed below is taken from the author's `flib` project[2] The idea is to have a configurable repository of useful modules and library objects which can be accessed by fortran programs. Different compilers are supported by tailored macros.
`xmlf90` is just one of several packages in `flib`, hence the `flib_` prefix in the package's visible module names.
To install the package, follow this steps:

```
 * Create a directory somewhere containing a copy of the stuff in the
   subdirectory 'macros':

cp -rp macros $HOME/flib

 * Define the environment variable FLIB_ROOT to point to that directory.

FLIB_ROOT=$HOME/flib  ; export FLIB_ROOT     (sh-like shells)
setenv FLIB_ROOT $HOME/flib                  (csh-like shells)


 * Go into $FLIB_ROOT, look through the fortran-XXXX.mk files,
   and see if one of them applies to your computer/compiler combination.
   If so, copy it or make a (symbolic) link to 'fortran.mk':

ln -sf fortran-lf95.mk fortran.mk

   If none of the .mk files look useful, write your own, using the
   files provided as a guide. Basically you need to figure out the
   name and options for the compiler,  the extension assigned to
   module files, and the flag used to identify the module search path.

 The above steps need only be done once.

 * Go into subdirectory 'sax' and type 'make'.
 * Go into subdirectory 'xpath' and type 'make'.
 * Go into subdirectory 'Tutorial' and try the exercises in this guide
  (see the next section for compilation details).
 * Go into subdirectory 'Examples' and explore.
```

8 Compiling user programs

After installation, the appropriate modules and library files should already be in `$FLIB_ROOT/modules` and `$FLIB_ROOT/lib`, respectively. To compile user programs, it is suggested that the user create a separate directory to hold the program files and prepare a `Makefile` following the template (taken from `Examples/sax/simple/`):

[2]There seems to be other projects with that very obvious name...

```
#-----------------------------------------------------------------
#
default: example
#
#---------------------------
MK=$(FLIB_ROOT)/fortran.mk
include $(MK)
#---------------------------
#
# Uncomment the following line for debugging support
#
FFLAGS=$(FFLAGS_DEBUG)
#
LIBS=$(LIB_PREFIX)$(LIB_STD) -lflib
#
OBJS= m_handlers.o example.o

example:  $(OBJS)
        $(FC) $(LDFLAGS) -o $@ $(OBJS)  $(LIBS)
#
clean:
        rm -f *.o example *$(MOD_EXT)
#
#-----------------------------------------------------------------
```

Here it is assumed that the user has two source files, `example.f90` and `m_handlers.f90`. Simply typing `make` will compile `example`, pulling in all the needed modules and library objects.