4. The SAX module

4.1. Description

Parsing XML streams can be done with two different methods. They each have their pros and cons. Although the simplest, and probably most usual way to manipulate XML files is to represent them in a tree and manipulate it through the DOM interface (see next chapter).

The **Simple API for XML** is an other method that can be used for parsing. It is based on a callbacks mechanism, and doesn't store any data in memory (unless of course you choose to do so in your callbacks). It can thus be more efficient to use SAX than DOM for some specialized algorithms. In fact, this whole Ada XML library is based on such a SAX parser, then creates the DOM tree through callbacks.

Note that this module supports the second release of SAX (SAX2), that fully supports namespaces as defined in the XML standard.

SAX can also be used in cases where a tree would not be the most efficient representation for your data. There is no point in building a tree with DOM, then extracting the data and freeing the tree occupied by the tree. It is much more efficient to directly store your data through SAX callbacks.

With SAX, you register a number of callback routines that the parser will call them when certain conditions occur.

This documentation is in no way a full documentation on SAX. Instead, you should refer to the standard itself, available at http://sax.sourceforge.net.

Some of the more useful callbacks are *Start_Document*, *End_Document*, *Start_Element*, *End_Element*, *Get_Entity* and *Characters*. Most of these are quite self explanatory. The characters callback is called when characters outside a tag are parsed.

Consider the following XML file:

```
<?xml version="1.0"?>
<body>
  <h1>Title</h1>
</body>
```

The following events would then be generated when this file is parsed:

```
Start_Document           Start parsing the file
Start_Prefix_Mapping     (handling of namespaces for "xml")
Start_Prefix_Mapping     Parameter is "xmlns"
Processing_Instruction   Parameters are "xml" and "version="1.0""
Start_Element            Parameter is "body"
Characters               Parameter is ASCII.LF & "  "
Start_Element            Parameter is "h1"
Characters               Parameter is "Title"
End_Element              Parameter is "h1"
Characters               Parameter is ASCII.LF & "  "
End_Element              Parameter is "body"
End_Prefix_Mapping       Parameter is "xmlns"
End_Prefix_Mapping       Parameter is "xml"
End_Document             End of parsing
```

As you can see, there is a number of events even for a very small file. However, you can easily choose to ignore the events you don't care about, for instance the ones related to namespace handling.

4.2. Examples

There are several cases where using a SAX parser rather than a DOM parser would make sense.

Here are some examples, although obvisouly this doesn't include all the possible cases. These examples are taken from the documentation of libxml, a GPL C toolkit for manipulating XML files.

- Using XML files as a database

  One of the common usage for XML files is to use them as a kind of basic database, They obviously provide a strongly structured format, and you could for instance store a series of numbers with the following format:

  ```
  <array> <value>1</value> <value>2</value> ....</array>
  ```

  In this case, rather than reading this file into a tree, it would obviously be easier to manipulate it through a SAX parser, that would directly create a standard Ada array while reading the values.

  This can be extended to much more complex cases that would map to Ada records for instance.

- Large repetitive XML files

  Sometimes we have XML files with many subtrees of the same format describing different things. An example of this is an index file for a documentation similar to this one. This contains a lot (maybe thousands) of similar entries, each containing for instance the name of the symbol and a list of locations.

  If the user is looking for a specific entry, there is no point in loading the whole file in memory and then traverse the resulting tree. The memory usage increases very fast with the size of the file, and this might even be unfeasible for a 35 megabytes file.

- Simple XML files

  Even for simple XML files, it might make sense to use a SAX parser. For instance, if there are some known constraints in the input file, say there are no attributes for elements, you can save quite a lot of memory, and maybe time, by rebuilding your own tree rather than using the full DOM tree.

However, there are also a number of drawbacks to using SAX:

- SAX parsers generally require you to write a little bit more code than the DOM interface
- There is no easy way to write the XML data back to a file, unless you build your own internal tree to save the XML. As a result, SAX is probably not the best interface if you want to load, modify and dump back an XML file.

  Note however that in this Ada implementation, the DOM tree is built through a set of SAX callbacks anyway, so you do not lose any power or speed by using DOM instead of SAX.

4.3. The SAX parser

The basic type in the SAX module is the **SAX.Readers** package. It defines a tagged type, called *Reader*, that represents the SAX parser itself.

Several features are define in the SAX standard for the parsers. They indicate which behavior can be expected from the parser. The package *SAX.Readers* defines a number of constant strings for each of these features. Some of these features are read-only, whereas others can be modified by the user to adapt the parser. See the *Set_Feature* and *Get_Feature* subprograms for how to manipulate them.

The main primitive operation for the parser is *Parse*. It takes an input stream for argument, associated with some XML data, and then parses it and calls the appropriate callbacks. It returns once there are no more characters left in the stream.

Several other primitive subprograms are defined for the parser, that are called the **callbacks**.

They get called automatically by the *Parse* procedure when some events are seen.

As a result, you should always override at least some of these subprograms to get something done. The default implementation for these is to do nothing, except for the error handler that raises Ada exceptions appropriately.

An example of such an implementation of a SAX parser is available in the DOM module, and it creates a tree in memory. As you will see if you look at the code, the callbacks are actually very short.

Note that internally, all the strings are encoded with a unique character encoding scheme, that is defined in the file `sax-encodings.ads`. The input stream is converted on the fly to this internal encoding, and all the subprograms from then on will receive and pass parameters with this new encoding. You can of course freely change the encoding defined in the file `sax-encodings.ads`.

The encoding used for the input stream is either automatically detected by the stream itself (*The Input module*), or by parsing the:

```
<?xml version='1.0' encoding='UTF-8' ?>
```

processing instruction at the beginning of the document. The list of supported encodings is the same as for the Unicode module (*The Unicode module*).

4.4. The SAX handlers

We do not intend to document the whole set of possible callbacks associated with a SAX parser. These are all fully documented in the file `sax-readers.ads`.

here is a list of the most frequently used callbacks, that you will probably need to override in most of your applications.

Start_Document
: This callback, that doesn't receive any parameter, is called once, just before parsing the document. It should generally be used to initialize internal data needed later on. It is also garanteed to be called only once per input stream.

End_Document
: This one is the reverse of the previous one, and will also be called only once per input stream. It should be used to release the memory you have allocated in Start_Document.

Start_Element
: This callback is called every time the parser encounters the start of an element in the XML file. It is passed the name of the element, as well as the relevant namespace information. The attributes defined in this element are also passed as a list. Thus, you get all the required information for this element in a single function call.

End_Element
: This is the opposite of the previous callback, and will be called once per element. Calls to *Start_Element* and *End_Element* are garanteed to be properly nested (ie you can't see the end of an element before seeing the end of all its nested children.

Characters and Ignore_Whitespace
: This procedure will be called every time some character not part of an element declaration are encounted. The characters themselves are passed as an argument to the callback. Note that the white spaces (and tabulations) are reported separately in the Ignorable_Spaces callback in case the XML attribute *xml:space* was set to something else than *preserve* for this element.

You should compile and run the `testsax` executable found in this module to visualize the SAX events that are generated for a given XML file.

4.5. Using SAX

This section will guide you through the creation of a small SAX application. This application will read an XML file, assumed to be a configuration file, and setup some preferences according to the contents of the file.

The XML file is the following:

```
<?xml version="1.0" ?>
<preferences>
   <pref name="pref1">Value1</pref>
   <pref name="pref2">Value2</pref>
</preferences>
```

This is a very simple example which doesn't use namespaces, and has a very limited nesting of nodes. However, that should help demonstrate the basics of using SAX.

4.5.1. Parsing the file

The first thing to do is to declare a parser, and parse the file. No callback is put in place in this first version, and as a result nothing happens.

The main program is the following:

```
1  with Sax.Readers;         use Sax.Readers;
2  with Input_Sources.File; use Input_Sources.File;
3  with SaxExample;          use SaxExample;
4
5  procedure SaxExample_Main is
6     My_Reader : SaxExample.Reader;
7     Input     : File_Input;
8  begin
9     Set_Public_Id (Input, "Preferences file");
10    Set_System_Id (Input, "pref.xml");
11    Open ("pref.xml", Input);
12
13    Set_Feature (My_Reader, Namespace_Prefixes_Feature, False);
14    Set_Feature (My_Reader, Namespace_Feature, False);
15    Set_Feature (My_Reader, Validation_Feature, False);
16
17    Parse (My_Reader, Input);
18
19    Close (Input);
20 end SaxExample_Main;
```

A separate package is provided that contain our implementation of an XML parser:

```
1  with Sax.Readers;
2
3  package SaxExample is
4
5     type Reader is new Sax.Readers.Reader with null record;
6
7  end SaxExample;
```

There are two steps in setting up an XML parser:

- Create an input stream

  This input stream is in charge of providing the XML input to the parser. Several input streams are provided by XML/Ada, including the one we use in this example to read the XML data from a file on the disk. The file is called `pref.xml`.

  It has two properties, that should generally be set: the public id will be used by XML/Ada in its error message to reference locations in that file; the system id should be the location of the file on the system. It is used to resolve relative paths found in the XML document.

- Setup the parser

  The behavior of an XML parser can be changed in several ways by activating or deactivating some features. In the example above, we have specified that the XML document doesn't contain namespaces, and that we do not intend to validate the XML file against a grammar.

Once the two steps above are done, we can simply call the procedure *Parse* to perform the actual parsing. Since we are using SAX, XML/Ada will call the primitive operations of *My_Reader*, which, so far, are inherited form the default ones provided by XML, and do nothing.

4.5.2. Reacting to events

We are now going to enhance the example a little, and make it react to the contents of the XML file.

We are only interested in two particular type of events, which are the opening and closing of an XML tag, and finding the value of each preference.

The way to react to these events is to override some of the primitive subprograms in the package `saxexample.ads` as follows:

```
1  with Sax.Readers;
2  with Unicode.CES;
3  with Sax.Attributes;
4  with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
5
6  package SaxExample is
7
8     type String_Access is access String;
9
10    type Reader is new Sax.Readers.Reader with record
11        Current_Pref  : Unbounded_String;
12        Current_Value : Unbounded_String;
13    end record;
14
15    procedure Start_Element
16      (Handler       : in out Reader;
17       Namespace_URI : Unicode.CES.Byte_Sequence := "";
18       Local_Name    : Unicode.CES.Byte_Sequence := "";
19       Qname         : Unicode.CES.Byte_Sequence := "";
20       Atts          : Sax.Attributes.Attributes'Class);
21
22    procedure End_Element
23      (Handler : in out Reader;
24       Namespace_URI : Unicode.CES.Byte_Sequence := "";
25       Local_Name    : Unicode.CES.Byte_Sequence := "";
26       Qname         : Unicode.CES.Byte_Sequence := "");
27
28    procedure Characters
29      (Handler : in out Reader;
30       Ch      : Unicode.CES.Byte_Sequence);
31
32 end SaxExample;
```

The primitive operations will be called automatically when the corresponding events are detected in the XML file.

The implementation for these subprograms is detailed below.

4.5.2.1. Start of XML tags

When an XML tag is started, we need to check whether it corresponds to the definition of a preference value. If that is the case, we get the value of the *name* attribute, which specifies the name of a preference:

```
1  with Unicode.CES;    use Unicode.CES;
2  with Sax.Attributes; use Sax.Attributes;
3  with Ada.Text_IO;    use Ada.Text_IO;
```

```
4
5  package body SaxExample is
6
7     procedure Start_Element
8       (Handler       : in out Reader;
9        Namespace_URI : Unicode.CES.Byte_Sequence := "";
10       Local_Name    : Unicode.CES.Byte_Sequence := "";
11       Qname         : Unicode.CES.Byte_Sequence := "";
12       Atts          : Sax.Attributes.Attributes'Class)
13    is
14    begin
15       Handler.Current_Pref  := Null_Unbounded_String;
16       Handler.Current_Value := Null_Unbounded_String;
17
18       if Local_Name = "pref" then
19          Handler.Current_Pref :=
20            To_Unbounded_String (Get_Value (Atts, "name"));
21       end if;
22    end Start_Element;
```

4.5.2.2. Characters

XML/Ada will report the textual contents of an XML tag through one or more calls to the *Characters* primitive operation. An XML parser is free to divide the contents into as many calls to *Characters* as it needs, and we must be prepared to handle this properly. Therefore, we concatenate the characters with the current value:

```
1  procedure Characters
2    (Handler : in out Reader;
3     Ch      : Unicode.CES.Byte_Sequence) is
4  begin
5     if Handler.Current_Pref /= Null_Unbounded_String then
6        Handler.Current_Value := Handler.Current_Value & Ch;
7     end if;
8  end Characters;
```

4.5.2.3. End of tag

Once we meet the end of a tag, we know there will be no more addition to the value, and we can now set the value of the preference. In this example, we simply display the value on the standard output:

```
1  procedure End_Element
2    (Handler : in out Reader;
3     Namespace_URI : Unicode.CES.Byte_Sequence := "";
4     Local_Name    : Unicode.CES.Byte_Sequence := "";
5     Qname         : Unicode.CES.Byte_Sequence := "")
6  is
7  begin
8     if Local_Name = "pref" then
9        Put_Line ("Value for """ & To_String (Handler.Current_Pref)
10                 & """ is " & To_String (Handler.Current_Value));
11    end if;
12 end End_Element;
```

In a real application, we would need to handle error cases in the XML file. Thankfully, most of the work is already done by XML/Ada, and the errors will be reported as calls to the primitive operation *Fatal_Error*, which by default raises an exception.

4.6. Understanding SAX error messages

XML/Ada error messages try to be as explicit as possible. They are not, however, meant to be understood by someone who doesn't know XML.

In addition to the location of the error (line and column in the file), they might contain one of the following abbreviations:

- *[WF]* .. index:: WF

  This abbreviation indicates that the error message is related to a well-formedness issue, as defined in the XML standard. Basically, the structure of the XML document is invalid, for instance because an open tag has never been closed. Some of the error messages also indicate a more precise section in the XML standard.

- *[VC]* .. index:: VC .. index:: DTD

  This abbreviation indicates that the error message is related to an unsatisfied validity-constraint, as defined in the XML standard. The XML document is well formed, although it doesn't match the semantic rules that the grammar defines. For instance, if you are trying to validate an XML document against a DTD, the document must contain a DTD that defines the name of the root element.