XML/Ada: the XML Library for Ada Documentation

Release 2018

AdaCore

May 24, 2018

CONTENTS

CHAPTER

ONE

INTRODUCTION

The Extensible Markup Language (XML) is a subset of SGML Its goal is to enable generic SGML to be served, received, and processed on the Web in the way that is now possible with HTML. XML has been designed for ease of implementation and for interoperability with both SGML and HTML.

This library includes a set of Ada95 packages to manipulate XML input. It implements the XML 1.0 standard (see the references at the end of this document), as well as support for namespaces and a number of other optional standards related to XML.

We have tried to follow as closely as possible the XML standard, so that you can easily analyze and reuse languages produced for other languages.

This document isn't a tutorial on what XML is, nor on the various standards like DOM and SAX. Although we will try and give a few examples, we refer the reader to the standards themselves, which are all easily readable.

CHAPTER

TWO

THE UNICODE MODULE

Unicode provides a unique number for every character, no matter what the platform, no matter what the program, no matter what the language.

Fundamentally, computers just deal with numbers. They store letters and other characters by assigning a number for each one. Before Unicode was invented, there were hundreds of different encoding systems for assigning these numbers. No single encoding could contain enough characters: for example, the European Union alone requires several different encodings to cover all its languages. Even for a single language like English no single encoding was adequate for all the letters, punctuation, and technical symbols in common use.

These encoding systems also conflict with one another. That is, two encodings can use the same number for two different characters, or use different numbers for the same character. Any given computer (especially servers) needs to support many different encodings; yet whenever data is passed between different encodings or platforms, that data always runs the risk of corruption.

Unicode provides a unique number for every character, no matter what the platform, no matter what the program, no matter what the language. The Unicode Standard has been adopted by such industry leaders as Apple, HP, IBM, JustSystem, Microsoft, Oracle, SAP, Sun, Sybase, Unisys and many others. Unicode is required by modern standards such as XML, Java, ECMAScript (JavaScript), LDAP, CORBA 3.0, WML, etc., and is the official way to implement ISO/IEC 10646. It is supported in many operating systems, all modern browsers, and many other products. The emergence of the Unicode Standard, and the availability of tools supporting it, are among the most significant recent global software technology trends.

The following sections explain the basic vocabulary and concepts associated with Unicode and encodings.

Most of the information comes from the official Unicode Web site, at http://www.unicode.org/unicode/reports/tr17.

Part of this documentation comes from http://www.unicode.org, the official web site for Unicode.

2.1 Glyphs

A glyph is a particular representation of a character or part of a character.

Several representations are possible, mostly depending on the exact font used at that time. A single glyph can correspond to a sequence of characters, or a single character to a sequence of glyphs.

The Unicode standard doesn't deal with glyphs, although a suggested representation is given for each character in the standard. Likewise, this module doesn't provide any graphical support for Unicode, and will just deal with textual memory representation and encodings.

Take a look at the **GtkAda** library that provides the graphical interface for unicode in the upcoming 2.0 version.

2.2 Repertoires and subsets

A repertoire is a set of abstract characters to be encoded, normally a familiar alphabet or symbol set. For instance, the alphabet used to spell English words, or the one used for the Russian alphabet are two such repertoires.

There exist two types of repertoires, close and open ones. The former is the most common one, and the two examples above are such repertoires. No character is ever added to them.

Unicode is also a repertoire, but an open one. New entries are added to it. However, it is guaranteed that none will ever be deleted from it. Unicode intends to be a universal repertoire, with all possible characters currently used in the world. It currently contains all the alphabets, including a number of alphabets associated with dead languages like hieroglyphs. It also contains a number of often used symbols, like mathematical signs.

The goal of this Unicode module is to convert all characters to entries in the Unicode repertoire, so that any applications can communicate with each other in a portable manner.

Given its size, most applications will only support a subset of Unicode. Some of the scripts, most notably Arabic and Asian languages, require a special support in the application (right-to-left writing,...), and thus will not be supported by some applications.

The Unicode standard includes a set of internal catalogs, called collections. Each character in these collections is given a special name, in addition to its code, to improve readability.

Several child packages (**Unicode.Names.***) define those names. For instance:

Unicode.Names.Basic_Latin This contains the basic characters used in most western European languages, including the standard ASCII subset.

Unicode.Names.Cyrillic This contains the Russian alphabet.

Unicode.Names.Mathematical_Operators This contains several mathematical symbols

More than 80 such packages exist.

2.3 Character sets

A character set is a mapping from a set of abstract characters to some non-negative integers. The integer associated with a character is called its code point, and the character itself is called the encoded character.

There exist a number of standard character sets, unfortunately not compatible with each other. For instance, ASCII is one of these character sets, and contains 128 characters. A super-set of it is the ISO/8859-1 character set. Another character set is the JIS X 0208, used to encode Japanese characters.

Note that a character set is different from a repertoire. For instance, the same character C with cedilla doesn't have the same integer value in the ISO/8859-1 character set and the ISO/8859-2 character set.

Unicode is also such a character set, that contains all the possible characters and associate a standard integer with them. A similar and fully compatible character set is ISO/10646. The only addition that Unicode does to ISO/10646 is that it also specifies algorithms for rendering presentation forms of some scripts (say Arabic), handling of bi-directional texts that mix for instance Latin and Hebrew, algorithms for sorting and string comparison, and much more.

Currently, our Unicode package doesn't include any support for these algorithms.

Unicode and ISO 10646 define formally a 31-bit character set. However, of this huge code space, so far characters have been assigned only to the first 65534 positions (0x0000 to 0xFFFD). The characters that are expected to be encoded outside the 16-bit range belong all to rather exotic scripts (e.g., Hieroglyphics) that are only used by specialists for historic and scientific purposes

The Unicode module contains a set of packages to provide conversion from some of the most common character sets to and from Unicode. These are the **Unicode.CCS.*** packages.

All these packages have a common structure:

- They define a global variable of type *Character_Set* with two fields, ie the two conversion functions between the given character set and Unicode.

  These functions convert one character (actually its code point) at a time.

- They also define a number of standard names associated with this character set. For instance, the ISO/8859-1 set is also known as Latin1.

  The function *Unicode.CCS.Get_Character_Set* can be used to find a character set by its standard name.

Currently, the following sets are supported:

ISO/8859-1 aka Latin1

> This is the standard character set used to represent most Western European languages including: Albanian, Catalan, Danish, Dutch, English, Faroese, Finnish, French, Galician, German, Irish, Icelandic, Italian, Norwegian, Portuguese, Spanish and Swedish.

ISO/8859-2 aka Latin2

> This character set supports the Slavic languages of Central Europe which use the Latin alphabet. The ISO-8859-2 set is used for the following languages: Czech, Croat, German, Hungarian, Polish, Romanian, Slovak and Slovenian.

ISO/8859-3

> This character set is used for Esperanto, Galician, Maltese and Turkish

ISO/8859-4

> Some letters were added to the ISO-8859-4 to support languages such as Estonian, Latvian and Lithuanian. It is an incomplete precursor of the Latin 6 set.

2.4 Character encoding schemes

We now know how each encoded character can be represented by an integer value (code point) depending on the character set.

Character encoding schemes deal with the representation of a sequence of integers to a sequence of code units. A code unit is a sequence of bytes on a computer architecture.

There exists a number of possible encoding schemes. Some of them encode all integers on the same number of bytes. They are called fixed-width encoding forms, and include the standard encoding for Internet emails (**7bits**, but it can't encode all characters), as well as the simple **8bits** scheme, or the **EBCDIC** scheme. Among them is also the **UTF-32** scheme which is defined in the Unicode standard.

Another set of encoding schemes encode integers on a variable number of bytes. These include two schemes that are also defined in the Unicode standard, namely **Utf-8** and **Utf-16**.

Unicode doesn't impose any specific encoding. However, it is most often associated with one of the Utf encodings. They each have their own properties and advantages:

Utf32

> This is the simplest of all these encodings. It simply encodes all the characters on 32 bits (4 bytes). This encodes all the possible characters in Unicode, and is obviously straightforward to manipulate. However, given that the first 65535 characters in Unicode are enough to encode all known languages currently in use, Utf32 is also a waste of space in most cases.

Utf16

For the above reason, Utf16 was defined. Most characters are only encoded on two bytes (which is enough for the first 65535 and most current characters). In addition, a number of special code points have been defined, known as *surrogate pairs*, that make the encoding of integers greater than 65535 possible. The integers are then encoded on four bytes. As a result, Utf16 is thus much more memory-efficient and requires less space than Utf32 to encode sequences of characters. However, it is also more complex to decode.

Utf8

This is an even more space-efficient encoding, but is also more complex to decode. More important, it is compatible with the most currently used simple 8bit encoding.

Utf8 has the following properties:

- Characters 0 to 127 (ASCII) are encoded simply as a single byte. This means that files and strings which contain only 7-bit ASCII characters have the same encoding under both ASCII and UTF-8.
- Characters greater than 127 are encoded as a sequence of several bytes, each of which has the most significant bit set. Therefore, no ASCII byte can appear as part of any other character.
- The first byte of a multibyte sequence that represents a non-ASCII character is always in the range 0xC0 to 0xFD and it indicates how many bytes follow for this character. All further bytes in a multibyte sequence are in the range 0x80 to 0xBF. This allows easy resynchronization and makes the encoding stateless and robust against missing bytes.
- UTF-8 encoded characters may theoretically be up to six bytes long, however the first 16-bit characters are only up to three bytes long.

Note that the encodings above, except for Utf8, have two versions, depending on the chosen byte order on the machine.

The Ada95 Unicode module provides a set of packages that provide an easy conversion between all the encoding schemes, as well as basic manipulations of these byte sequences. These are the **Unicode.CES.*** packages. Currently, four encoding schemes are supported, the three Utf schemes and the basic 8bit encoding which corresponds to the standard Ada strings.

It also supports some routines to convert from one byte-order to another.

The following examples show a possible use of these packages:

```
Converting a latin1 string coded on 8 bits to a Utf8 latin2 file
involves the following steps:

   Latin1 string  (bytes associated with code points in Latin1)
     |    "use Unicode.CES.Basic_8bit.To_Utf32"
     v
   Utf32 latin1 string (contains code points in Latin1)
     |    "Convert argument to To_Utf32 should be
     v         Unicode.CCS.Iso_8859_1.Convert"
   Utf32 Unicode string (contains code points in Unicode)
     |    "use Unicode.CES.Utf8.From_Utf32"
     v
   Utf8 Unicode string (contains code points in Unicode)
     |    "Convert argument to From_Utf32 should be
     v         Unicode.CCS.Iso_8859_2.Convert"
   Utf8 Latin2 string (contains code points in Latin2)
```

2.5 Unicode_Encoding

XML/Ada groups the two notions of character sets and encoding schemes into a single type, *Unicode.Encodings.Unicode_Encoding*.

This package provides additional functions to manipulate these encodings, for instance to retrieve them by the common name that is associated with them (for instance "utf-8", "iso-8859-15",...), since very often the encoding scheme is implicit. If you are speaking of utf-8 string, most people always assume you also use the unicode character set. Likewise, if you are speaking of "iso-8859-1", most people will assume you string is encoded as 8 byte characters.

The goal of the *Unicode.Encodings* package is to make these implicit associations more obvious.

It also provides one additional function *Convert*, which can be used to convert a sequence of bytes from one encoding to another. This is a convenience function that you can use when for instance creating DOM trees directly through Ada calls, since XML/Ada excepts all its strings to be in utf-8 by default.

2.6 Misc. functions

The package **Unicode** contains a series of *Is_** functions, matching the Unicode standard.

Is_White_Space

> Return True if the character argument is a space character, ie a space, horizontal tab, line feed or carriage return.

Is_Letter

> Return True if the character argument is a letter. This includes the standard English letters, as well as some less current cases defined in the standard.

Is_Base_Char Return True if the character is a base character, ie a character whose meaning can be modified with a combining character.

Is_Digit Return True if the character is a digit (numeric character)

Is_Combining_Char Return True if the character is a combining character. Combining characters are accents or other diacritical marks that are added to the previous character.

> The most important accented characters, like those used in the orthographies of common languages, have codes of their own in Unicode to ensure backwards compatibility with older character sets. Accented characters that have their own code position, but could also be represented as a pair of another character followed by a combining character, are known as precomposed characters. Precomposed characters are available in Unicode for backwards compatibility with older encodings such as ISO 8859 that had no combining characters. The combining character mechanism allows to add accents and other diacritical marks to any character
>
> Note however that your application must provide specific support for combining characters, at least if you want to represent them visually.

Is_Extender True if Char is an extender character.

Is_Ideographic True if Char is an ideographic character. This is defined only for Asian languages.

CHAPTER

THREE

THE INPUT MODULE

This module provides a set of packages with a common interface to access the characters contained in a stream. Various implementations are provided to access files and manipulate standard Ada strings.

A top-level tagged type is provided that must be extended for the various streams. It is assumed that the pointer to the current character in the stream can only go forward, and never backward. As a result, it is possible to implement this package for sockets or other strings where it isn't even possible to go backward. This also means that one doesn't have to provide buffers in such cases, and thus that it is possible to provide memory-efficient readers.

Two predefined readers are available, namely *String_Input* to read characters from a standard Ada string, and *File_Input* to read characters from a standard text file.

They all provide the following primite operations:

Open

Although this operation isn't exactly overriden, since its parameters depend on the type of stream you want to read from, it is nice to use a standard name for this constructor.

Close This terminates the stream reader and free any associated memory. It is no longer possible to read from the stream afterwards.

Next_Char Return the next Unicode character in the stream. Note this character doesn't have to be associated specifically with a single byte, but that it depends on the encoding chosen for the stream (see the unicode module documentation for more information).

The next time this function is called, it returns the following character from the stream.

Eof This function should return True when the reader has already returned the last character from the stream. Note that it is not guarantee that a second call to Eof will also return True.

It is the responsability of this stream reader to correctly call the decoding functions in the unicode module so as to return one single valid unicode character. No further processing is done on the result of *Next_Char*. Note that the standard *File_Input* and *String_Input* streams can automatically detect the encoding to use for a file, based on a header read directly from the file.

Based on the first four bytes of the stream (assuming this is valid XML), they will automatically detect whether the file was encoded as Utf8, Utf16,... If you are writing your own input streams, consider adding this automatic detection as well.

However, it is always possible to override the default through a call to *Set_Encoding*. This allows you to specify both the character set (Latin1, ...) and the character encoding scheme (Utf8,...).

The user is also encouraged to set the identifiers for the stream they are parsing, through calls to *Set_System_Id* and *Set_Public_Id*. These are used when reporting error messages.

CHAPTER

FOUR

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 another 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 a 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 traversing 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 defined in the SAX standard for the parsers. They indicate what 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 guaranteed 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 guaranteed 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 is encountered. 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  --
2  --  Copyright (C) 2017, AdaCore
3  --
4
5  with Sax.Readers;         use Sax.Readers;
6  with Input_Sources.File; use Input_Sources.File;
7  with SaxExample;          use SaxExample;
8
9  procedure SaxExample_Main is
10    My_Reader : SaxExample.Reader;
11    Input     : File_Input;
12 begin
13    Set_Public_Id (Input, "Preferences file");
14    Set_System_Id (Input, "pref.xml");
15    Open ("pref.xml", Input);
16
17    Set_Feature (My_Reader, Namespace_Prefixes_Feature, False);
18    Set_Feature (My_Reader, Namespace_Feature, False);
19    Set_Feature (My_Reader, Validation_Feature, False);
20
21    Parse (My_Reader, Input);
22
23    Close (Input);
24 end SaxExample_Main;
```

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

```
with Sax.Readers;

package SaxExample is

   type Reader is new Sax.Readers.Reader with null record;

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:

```
--
--  Copyright (C) 2017, AdaCore
--

with Sax.Readers;
with Unicode.CES;
with Sax.Attributes;
with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;

package SaxExample is

   type String_Access is access String;

   type Reader is new Sax.Readers.Reader with record
       Current_Pref  : Unbounded_String;
       Current_Value : Unbounded_String;
   end record;
```

```
18
19    procedure Start_Element
20      (Handler       : in out Reader;
21       Namespace_URI : Unicode.CES.Byte_Sequence := "";
22       Local_Name    : Unicode.CES.Byte_Sequence := "";
23       Qname         : Unicode.CES.Byte_Sequence := "";
24       Atts          : Sax.Attributes.Attributes'Class);
25
26    procedure End_Element
27      (Handler : in out Reader;
28       Namespace_URI : Unicode.CES.Byte_Sequence := "";
29       Local_Name    : Unicode.CES.Byte_Sequence := "";
30       Qname         : Unicode.CES.Byte_Sequence := "");
31
32    procedure Characters
33      (Handler : in out Reader;
34       Ch      : Unicode.CES.Byte_Sequence);
35
36 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.

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;
```

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:

```
procedure Characters
   (Handler : in out Reader;
    Ch      : Unicode.CES.Byte_Sequence) is
begin
    if Handler.Current_Pref /= Null_Unbounded_String then
       Handler.Current_Value := Handler.Current_Value & Ch;
    end if;
end Characters;
```

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:

```
procedure End_Element
   (Handler : in out Reader;
    Namespace_URI : Unicode.CES.Byte_Sequence := "";
    Local_Name    : Unicode.CES.Byte_Sequence := "";
    Qname         : Unicode.CES.Byte_Sequence := "")
is
begin
    if Local_Name = "pref" then
       Put_Line ("Value for """ & To_String (Handler.Current_Pref)
                 & """ is " & To_String (Handler.Current_Value));
    end if;
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.

CHAPTER

FIVE

THE DOM MODULE

DOM is another standard associated with XML, in which the XML stream is represented as a tree in memory. This tree can be manipulated at will, to add new nodes, remove existing nodes, change attributes,...

Since it contains the whole XML information, it can then in turn be dump to a stream.

As an example, most modern web browsers provide a DOM interface to the document currently loaded in the browser. Using javascript, one can thus modify dynamically the document. The calls to do so are similar to the ones provided by XML/Ada for manipulating a DOM tree, and all are defined in the DOM standard.

The W3C commitee (http://www.w3c.org) has defined several version of the DOM, each building on the previous one and adding several enhancements.

XML/Ada currently supports the second revision of DOM (DOM 2.0), which mostly adds namespaces over the first revision. The third revision is not supported at this point, and it adds support for loading and saving XML streams in a standardized fashion.

Although it doesn't support DOM 3.0, XML/Ada provides subprograms for doing similar things.

Only the Core module of the DOM standard is currently implemented, other modules will follow.

Note that the `encodings.ads` file specifies the encoding to use to store the tree in memory. Full compatibility with the XML standard requires that this be UTF16, however, it is generally much more memory-efficient for European languages to use UTF8. You can freely change this and recompile.

5.1 Using DOM

In XML/Ada, the DOM tree is build through a special implementation of a SAX parser, provided in the *DOM.Readers* package.

Using DOM to read an XML document is similar to using SAX: one must setup an input stream, then parse the document and get the tree. This is done with a code similar to the following:

```
1  --
2  --  Copyright (C) 2017, AdaCore
3  --
4
5  with Input_Sources.File; use Input_Sources.File;
6  with Sax.Readers;        use Sax.Readers;
7  with DOM.Readers;        use DOM.Readers;
8  with DOM.Core;           use DOM.Core;
9
10 procedure DomExample is
11    Input  : File_Input;
12    Reader : Tree_Reader;
```

```
   Doc    : Document;
begin
   Set_Public_Id (Input, "Preferences file");
   Open ("pref.xml", Input);

   Set_Feature (Reader, Validation_Feature, False);
   Set_Feature (Reader, Namespace_Feature, False);

   Parse (Reader, Input);
   Close (Input);

   Doc := Get_Tree (Reader);

   Free (Reader);
end DomExample;
```

This code is almost exactly the same as the code that was used when demonstrating the use of SAX (*Using SAX*).

The main two differences are:

- We no longer need to define our own XML reader, and we simply use the one provided in *DOM.Readers*.
- We therefore do not add our own callbacks to react to the XML events. Instead, the last instruction of the program gets a handle on the tree that was created in memory.

The tree can now be manipulated to get access to the value stored. If we want to implement the same thing we did for SAX, the code would look like:

```
--
--  Copyright (C) 2017, AdaCore
--

with Input_Sources.File; use Input_Sources.File;
with Sax.Readers;        use Sax.Readers;
with DOM.Readers;        use DOM.Readers;
with DOM.Core;           use DOM.Core;
with DOM.Core.Documents; use DOM.Core.Documents;
with DOM.Core.Nodes;     use DOM.Core.Nodes;
with DOM.Core.Attrs;     use DOM.Core.Attrs;
with Ada.Text_IO;        use Ada.Text_IO;

procedure DomExample2 is
   Input  : File_Input;
   Reader : Tree_Reader;
   Doc    : Document;
   List   : Node_List;
   N      : Node;
   A      : Attr;
begin
   Set_Public_Id (Input, "Preferences file");
   Open ("pref.xml", Input);

   Set_Feature (Reader, Validation_Feature, False);
   Set_Feature (Reader, Namespace_Feature, False);

   Parse (Reader, Input);
   Close (Input);

   Doc := Get_Tree (Reader);
```

```
32
33      List := Get_Elements_By_Tag_Name (Doc, "pref");
34
35      for Index in 1 .. Length (List) loop
36          N := Item (List, Index - 1);
37          A := Get_Named_Item (Attributes (N), "name");
38          Put_Line ("Value of """ & Value (A) & """ is "
39                    & Node_Value (First_Child (N)));
40      end loop;
41
42      Free (List);
43
44      Free (Reader);
45  end DomExample2;
```

The code is much simpler than with SAX, since most of the work is done internally by XML/Ada. In particular, for SAX we had to take into account the fact that the textual contents of a node could be reported in several events. For DOM, the tree is initially normalized, ie all text nodes are collapsed together when possible.

This added simplicity has one drawback, which is the amount of memory required to represent even a simple tree.

XML/Ada optimizes the memory necessary to represent a tree by sharing the strings as much as possible (this is under control of constants at the beginning of `dom-core.ads`). Still, DOM requires a significant amount of information to be kept for each node.

For really big XML streams, it might prove impossible to keep the whole tree in memory, in which case ad hoc storage might be implemented through the use of a SAX parser. The implementation of *dom-readers.adb* will prove helpful in creating such a parser.

5.2 Editing DOM trees

Once in memory, DOM trees can be manipulated through subprograms provides by the DOM API.

Each of these subprograms is fully documented both in the Ada specs (the `*.ads` files) and in the DOM standard itself, which XML/Ada follows fully.

One important note however is related to the use of strings. Various subprograms allows you to set the textual content of a node, modify its attributes,.... Such subprograms take a Byte_Sequence as a parameter.

This Byte_Sequence must always be encoded in the encoding defined in the package *Sax.Encoding* (as described earlier, changing this package requires recompiling XML/Ada). By default, this is UTF-8.

Therefore, if you need to set an attribute to a string encoded for instance in iso-8859-15, you should use the subprogram *Unicode.Encodings.Convert* to convert it appropriately. The code would thus look as follows:

```
Set_Attribute (N, Convert ("å", From => Get_By_Name ("iso-8859-15")));
```

5.3 Printing DOM tress

The standard DOM 2.0 does not define a common way to read DOM trees from input sources, nor how to write them back to output sources. This was added in later revision of the standard (DOM 3.0), which is not yet supported by XML/Ada.

However, the package `DOM.Core.Nodes` provides a *Write* procedure that can be used for that purpose. It outputs a given DOM tree to an Ada stream. This stream can then be connected to a standard file on the disk, to a socket, or be used to transform the tree into a string in memory.

An example is provided in the XML/Ada distribution, called `dom/test/tostring.adb` which shows how you can create a stream to convert the tree in memory, without going through a file on the disk

5.4 Adding information to the tree

The DOM standard does not mandate each node to have a pointer to the location it was read from (for instance *file:line:column*). In fact, storing that for each node would increase the size of the DOM tree (not small by any means already) significantly.

But depending on your application, this might be a useful information to have, for instance if you want to report error messages with a correct location.

Fortunately, this can be done relatively easily by extending the type *DOM.Readers.Tree_Reader*, and override the *Start_Element*. You would then add a custom attribute to all the nodes that contain the location for this node. Here is an example.

```
--
--  Copyright (C) 2017, AdaCore
--

with DOM.Readers;       use DOM.Readers;
with Sax.Utils;         use Sax.Utils;
with Sax.Readers;       use Sax.Readers;
with Sax.Symbols;       use Sax.Symbols;

package DOM_With_Location is

   type Tree_Reader_With_Location is new Tree_Reader with null record;
   overriding procedure Start_Element
      (Handler     : in out Tree_Reader_With_Location;
       NS          : Sax.Utils.XML_NS;
       Local_Name  : Sax.Symbols.Symbol;
       Atts        : Sax.Readers.Sax_Attribute_List);

end DOM_With_Location;
```

```
--
--  Copyright (C) 2017, AdaCore
--

with DOM.Core;             use DOM.Core;
with DOM.Core.Attrs;       use DOM.Core.Attrs;
with DOM.Core.Documents;   use DOM.Core.Documents;
with DOM.Core.Elements;    use DOM.Core.Elements;
with Sax.Locators;         use Sax.Locators;

package body DOM_With_Location is

   overriding procedure Start_Element
      (Handler     : in out Tree_Reader_With_Location;
       NS          : Sax.Utils.XML_NS;
       Local_Name  : Sax.Symbols.Symbol;
```

```
17          Atts         : Sax_Attribute_List)
18    is
19       Att, Att2 : Attr;
20    begin
21       --  First create the node as usual
22       Start_Element (Tree_Reader (Handler), NS, Local_Name, Atts);
23
24       --  Then add the new attribute
25       Att := Create_Attribute_NS
26          (Get_Tree (Handler),
27           Namespace_URI  => "http://mydomain.com",
28           Qualified_Name => "mydomain:location");
29       Set_Value (Att, To_String (Current_Location (Handler)));
30
31       Att2 := Set_Attribute_Node (Handler.Current_Node, Att);
32    end Start_Element;
33
34 end DOM_With_Location;
```

CHAPTER

SIX

THE SCHEMA MODULE

6.1 XML Grammars

There are several steps that applications must go through when they have to use XML files:

- Make sure the XML file is well-formed.

  This is a basic step where we ensure that XML tags are correctly nested, that closing tags have the same names as the matching opening tags, that attribute values are quoted,.... This corresponds to a syntactic parser in a compiler.

  This step does not depend on the application domain. One file that is well-formed will always be so, no matter in what context you use it.

- Make sure the contents of the XML file is semantically valid.

  Depending on the application domain, we must ensure that the content of the file makes sense. This step is highly application dependent, and a file that is usable in one application might not be usable in another one.

  This is the phase in which the application needs to check whether a given XML file has all its required attributes, whether the children of an XML tag are the expected ones, whether the type of the attributes is valid,....

- Use the XML file in the application.

  This is done through the already-described SAX or DOM parsers

The first phase is mandatory, and necessarily enforced by XML/Ada. You will not be able to access the contents of the XML file if it isn't well-formed.

The second phase is provided by the Schema module in XML/Ada. Although such constraints can be checked at the application level, with ad hoc code, it is generally easier to maintain a separate file that describes the valid semantic contents of the file, that maintain specific code when the semantic changes. It is also difficult not to forget special cases when doing the validating through a set of *if* statements in the Ada core.

XML provides two ways to describe additional constraints that a file must satisfy in order to be considered as valid.

- DTD

  The Document Type Description is the original way to do this. They come directly from the ancestor of XML, SGML. All XML parsers must parse the DTD, and report events if the user is using SAX. However, not all parsers are able to validate the document against a DTD (XML/Ada doesn't).

  Their use tend to greatly diminish. Among their limitation are a limit capability to express constraints on the order of tag children, the fact they the DTD themselves are written in a separate language totally different from XML, and that users must learn as a result.

- XML Schema

The XML schemas are replacing the DTDs. They are written in XML, and provide an extensive capability to describe what the XML document should look like. In fact, almost all Ada types can be described in an XML schema, including range constraints, arrays, records, type inheritance, abstract types,....

It is for instance possible to indicate that the value of a preference, in our example, must be a string of length 6. Any other length will result in a validation error.

6.2 XML Schema Syntax

The Schema modules provides subprograms and types to parse an XML schema and validate an XML document with this schema.

This document does not provide a full documentation on the format of XML Schemas. This is extensive, has several obscure features, which, although supported by XML/Ada, are of little use in most pratical uses. We refer the reader to the first part of the XML Schema specification, which is designed as a tutorial (http://www.w3.org/TR/xmlschema-0/).

The typical extension for a schema file is `.xsd`.

A schema file must be a valid XML file, and thus start with the usual *<?xml version="1.0" ?>* line. The root node must be named *schema*, and belong to the namespace (http://www.w3.org/2001/XMLSchema/). The handling of namespaces is fairly powerful, but also complex. A given XML document might have nodes belonging to several namespaces, and thus several schema files might have to be loaded, each defining one of the namespaces.

In the following simple example, we will not define our schema for a specific namespace, and thus no special attribute is needed for the root node. Thus, our document will be organized as:

```
<?xml version="1.0" ?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    ... rest of the description goes here ...
</xsd:schema>
```

An XML schema does not enforce a specific root node in the XML documents it validates. However, it must define all the valid elements that can be used in the XML file. This is done through the *<element>* tag, which takes one mandatory attribute, the name of the element we are defining.

The contents of the element is then defined in one of two ways:

- Through a *type* attribute.

  Schemas come with a number of predefined simple types. A simple type is such that an element of that type accepts no child node, and that its contents must satisfy additional constraints (be an integer, a date, ...).

  Among the predefined simple type (which are all defined in the namespace http://www.w3.org/2001/XMLSchema/), one can find: *string*, *integer*, *byte*, *date*, *time*, *dateTime*, *boolean*,...

  If no additional constraint should be enforced on this simple type when applied to the element, the type of the element is given through a *type* attribute, as in:

  ```
  <xsd:element name="tag1" type="xsd:string"  />
  <xsd:element name="tag2" type="xsd:boolean" />
  ```

  which would accept the following XML files:

  ```
  <tag1>Any string is valid here</tag1>
  ```

  and:

  ```
  <tag2>true</tag2>
  ```

but not:

```
<tag2>String</tag2>
```

As will be described later, it is possible to create new types in XML schema, which are created with a name. Such new types can also be associated with the element through the *type* attribute.

- Through an inline type definition

  If the element must accept child elements, or if a further constraint needs to be enforced on the list of valid values, one must create the type. As mentioned above, this can be done by creating a type separately and referencing it by name, or through an inline type definition.

  The syntax is mostly the same in both cases. Schemas distinguish between the notion of simple types (that accept no child element) and complex types (that accept child elements, and possibly text value).

  To define a simple type, based on string, but that only allows a limited set of value (similar to an Ada enumeration), one would create a restriction of the standard string type, as in:

```
<xsd:element name="tag3">
 <xsd:simpleType>
   <xsd:restriction base="xsd:string">
     <xsd:enumeration value="value1" />
     <xsd:enumeration value="value2" />
   </xsd:restriction>
 </xsd:simpleType>
</xsd:element>
```

  Similarly, we could create an integer type whose valid range of values is between 10 and 20, as in:

```
<xsd:element name="tag4">
 <xsd:simpleType>
   <xsd:restriction base="xsd:byte">
     <xsd:minInclusive value="10" />
     <xsd:maxInclusive value="20" />
   </xsd:restriction>
 </xsd:simpleType>
</xsd:element>
```

  Complex types allow elements to have child nodes, as well as attributes. The list of valid attributes is created by a set of <*xsd:attribute*> tags, and the list of valid child nodes is generally defined either through a <*xsd:choice*> or a <*xsd:sequence*> node (although it is possible to indicate that any child node is authorized, among other things).

  <*xsd:choice*> indicate the children can appear in any order, whereas <*xsd:sequence*> enforces a specific order on children.

  In both cases, extra attributes can be specified to indicate the number of times the sequence or choice itself can be repeated, or that each child node can appear.

  For instance, we can indicate that *tag5* accepts between 1 and 4 child nodes, chosen among *tag6* and *tag7*, but that the latter, if present, can only appear once. In addition, tag5 accepts one optional attribute. Note that the type of tag6 and tag7 is here specified through a *type* attribute, although it could in turn be defined inline:

```
<xsd:element name="tag5">
  <xsd:complexType>
    <xsd:choice>
      <xsd:element name="tag6" type="xsd:string"
                   minOccurs="1" maxOccurs="3/>
      <xsd:element name="tag7" type="xsd:string" maxOccurs="1" />
```

```
    </xsd:choice>
    <xsd:attribute name="attr" type="xsd:boolean" use="optional" />
  </xsd:complexType>
</xsd:element>
```

In the example above, if *tag6* was defined elsewhere in the schema, we could use a reference to it, instead of duplicating its type definition, as in:

```
<xsd:element ref="tag6" />
```

If you need an element with no child element (just a string value), but that accepts attributes, this also must be defined through a complex type, as in:

```
<xsd:element name="tag8" />
  <xsd:complexType>
    <xsd:simpleContent>
      <xsd:extension base="xsd:string">
         <xsd:attribute name="attr" type="xsd:boolean" />
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
</xsd:element>
```

As mentioned before, instead of defining inline types, we could explicitly declare them, and reference them in the element declaration later on:

```
<xsd:simpleType name="string_of_length_10">
  <xsd:restriction base="xsd:string" />
    <xsd:length value="10"/>
  </xsd:restriction>
</xsd:simpleType>
<xsd:element name="tag9" type="string_of_length_10" />
```

6.3 Connecting XML documents and schemas

There are several ways that XML/Ada uses to find what schema to use when validating a file.

- Manually creating the grammar.

  The schema module contains the package *Schema.Validators* which allows you to create a grammar by hand. It is very low-level, and it is likely that you will never need to use it. It is used internally mostly, and when creating the schema which is used to validate schema files themselves.

- Explicitly parsing a schema file

  Parsing a schema file can be done through a call to parse for a reader derived from *Schema.Schema_Readers.Schema_reader*. As usual, you call *Parse*, and pass it an input source. As output, you get access to a grammar, that can then be given to another instance of a *Schema.Readers.Validating_Reader*.

  This technique will generally be used when you need to validate several XML files with the same grammar: you parse the grammar only once, and then reuse its instance, instead of reparsing the `.xsd` file every time:

```
1 with Ada.Text_IO;   use Ada.Text_IO;
2 with Schema.Schema_Readers, Schema.Validators, Input_Sources.File;
3 use  Schema.Schema_Readers, Schema.Validators, Input_Sources.File;
4
```

```
procedure SchemaExample2 is
   Grammar : XML_Grammar;
   Schema  : Schema_Reader;
   Read    : File_Input;
begin
   Open ("file.xsd", Read);
   Parse (Schema, Read);
   Close (Read);

   Grammar := Get_Grammar (Schema);

 exception
    when XML_Validation_Error | XML_Not_Implemented =>
       Put_Line ("ERROR: " & Get_Error_Message (Schema));
end SchemaExample2;
```

In the example above, the schema file itself is validated against the official schema for schema files.

The resulting grammar object is in fact a collection of parsed schema files, each associated with its own namespace. It can be kept as long as you need it in your application. Memory will automatically be reclaimed when no longer needed.

Every time you parse an XML file later on, you must associated the Grammar with the parser:

```
declare
   Read      : File_Input;
   My_Reader : Validating_Reader;
begin
   Set_Grammar (My_Reader, Grammar);
   Set_Feature (My_Reader, Schema_Validation_Feature, True);
   Open (Xml_File.all, Read);
   Parse (My_Reader, Read);
   Close (Read);

exception
   when XML_Validation_Error | XML_Not_Implemented =>
      Put_Line ("ERROR: " & Get_Error_Message (My_reader));
end;
```

- Implicitly parsing the schema

  Two special attributes, defined in the Schema standard, can be used to indicate, in an XML document itself, that it should be validated with a specific schema.

  These attributes are both defined in a special namespace, http://www.w3.org/2001/XMLSchema-instance.

  – *xsi:noNamespaceSchemaLocation*

    The value of this attribute is the name of a file that contains the schema to use for elements that are not associated with a specific namespace.

  – *xsi:schemaLocation*

    This attribute is a list of strings, alternatively the prefix of a namespace and the name of an xsd file to use for that namespace. For instance, *"ns1 file1.xsd ns2 file2.xsd"*.

  When it encounters any of these two attributes, XML/Ada will automatically parse the corresponding schema files, and use the result to validate the file.

  See the section below on optimizing the parsing of the grammars, as a way to avoid parsing the same grammar multiple times.

6.4 Validating documents with SAX

XML/Ada is quite unique in the category of XML parsers, since it allows the validation of XML files when you are using an event-based parser with SAX. Most other XML parsers only work on DOM trees.

Basing the validation on SAX is more efficient, since there is no need to read the whole XML stream (or even the grammar) in memory before starting the validation, and errors can be reported immediatly.

It also requires less memory to run, and thus can validate large XML documents.

It also means that even if you are using SAX, and not DOM, you still have access to the validation features.

Validating a XML document while parsing it is basically done the same as when using SAX itself. Instead of inheriting from *Sax.Readers.Reader*, your tagged type must inherit from *Schema.Readers.Validating_Reader*.

As usual, you can still override the predefined primitive operations like *Start_Element*, *End_Element*, ...

Note the activation of the *Schema_Validation_Feature* feature, without which no validation takes place:

```
--
--  Copyright (C) 2017, AdaCore
--

with Ada.Text_IO;         use Ada.Text_IO;
with Sax.Readers;         use Sax.Readers;
with Schema.Readers;      use Schema.Readers;
with Schema.Validators;
with Input_Sources.File; use Input_Sources.File;

procedure SchemaExample is
   Input : File_Input;
   My_Reader : Validating_Reader;
begin
   Set_Public_Id (Input, "Preferences file");
   Open ("pref.xml", Input);

   Set_Feature (My_Reader, Schema_Validation_Feature, True);
   Parse (My_Reader, Input);

   Close (Input);

exception
   when Schema.Validators.XML_Validation_Error =>
      Put_Line ("ERROR: " & Get_Error_Message (My_Reader));
end SchemaExample;
```

6.5 Validating documents with DOM

This is very similar to using DOM itself, except the base class of your reader should be *Schema.Dom_Readers.Tree_Reader*. Going back to the example described in *Using DOM*, you would use the following to validate XML streams before generating the DOM tree.

```
--
--  Copyright (C) 2017, AdaCore
--

with Input_Sources.File; use Input_Sources.File;
```

```
with Sax.Readers;         use Sax.Readers;
with DOM.Core;            use DOM.Core;
with Schema.Dom_Readers; use Schema.Dom_Readers;

procedure DomSchemaExample is
   Input  : File_Input;
   Reader : Schema.Dom_Readers.Tree_Reader;
   Doc    : Document;
begin
   Set_Public_Id (Input, "Preferences file");
   Open ("pref_with_xsd.xml", Input);

   Set_Feature (Reader, Validation_Feature, False);

   Parse (Reader, Input);
   Close (Input);

   Doc := Get_Tree (Reader);

   Free (Reader);
end DomSchemaExample;
```

6.6 Unsupported schema elements

Not all aspects of XML schemas are supported by XML/Ada. In particular, it does not currently support XPath, so any part of the schema that is related to XPath expressions (for instance <*xsd:key*> and <*xsd:unique*>) are not supported currently.

6.7 Optimizing the parsing of grammars

It is often the case that a given `.xsd` file will be reused multiple times to validate XML documents. In such case, you do not want to parse the file multiple times, but instead reuse an already existing *XML_Grammar* object. Of course, this is a tradeoff between memory used to keep the grammar in memory, and the time it would take to reparse the grammar.

This is easily done when you have a single `.xsd` file to reuse for all the XML files. Simply call *Set_Grammar* on the parser before you parse the file, as in:

```
declare
   G : constant XML_Grammar := ...;  --  parsed earlier
   R : Validating_Reader;
   F : File_Input;
begin
   R.Set_Grammar (G);
   Open ("file.xml", F);
   R.Parse (F);
   Close (F);
   ...;  --  Do something with the resulting tree
end;
```

The second use case is a bit more complex: you have several XSD files to parse, and the XML files will need either of these. If you are using namespaces, there is nothing special to do, and the same code as above applies: you can simply parse each of the XSD file into the same XML_Grammar, and then use that grammar to parse all the XML files, as in:

```
declare
   G : XML_Grammar;
   S : Schema_Reader;
   F : File_Input;
   R : Validating_Reader;
begin
   Open ("grammar1.xsd", F);
   S.Parse (F);
   F.Close;

   Open ("grammar2.xsd", F);
   S.Parse (F);
   F.Close;

   G := S.Get_Grammar;

   R.Set_Grammar (G);
   Open ("file.xml", F);
   R.Parse (F);
   F.Close;
end;
```

If however you are not using namespaces, you cannot use this technics, since the grammar from the various XSD files would end up mixed up, and validation will most likely fail. So instead you need to have one *XML_Grammar* per XSD file, and then set the grammar on the reader dynamically. A full example is given in the XML/Ada source distribution, in `tests/schema/multiple_xsd`. Here is an overview.

We first need to parse each of the XSD files into its own grammar:

```
declare
   Symbols : Symbol_Table;
   G1, G2 : XML_Grammar;
   S      : Schema_Grammar;
   F      : File_Input;
begin
   --  Since we are going to reuse grammars, we need to ensure their
   --  symbol tables (where internal strings are stored) across all
   --  involved parsers).
   Symbols := Allocate;
   S.Set_Symbol_Table (Symbols);

   --  Now we can parse each of the XSD file
   Open ("algo1.xsd", F);
   S.Parse (F);
   F.Close;
   G1 := S.Get_Grammar;

   S.Set_Grammar (No_Grammar);  --  reset
   Open ("algo2.xsd", F);
   S.Parse (F);
   F.Close;
   G2 := S.Get_Grammar;
end;
```

We then need to create a custom validating reader, which knows how to set the grammar based on its name. This is done by overriding one of the primitive operations of the parser:

```
declare
   type My_Reader is new Validating_Reader with null record;
   overriding procedure Parse_Grammar
      (Self          : not null access Reader_With_Preloaded_XSD;
       URI, Xsd_File : Sax.Symbols.Symbol;
       Do_Create_NFA : Boolean := True) is
   begin
      if Xsd_File = "algo1.xsd" then
         Self.Set_Grammar (G1);
      elsif Xsd_File = "algo2.xsd" then
         Self.Set_Grammar (G2);
      end if;
   end Parse_Grammar;

   R : My_Reader;
begin
   --  Also share the same symbol table
   R.Set_Symbol_Table (Symbols);
   R.Set_Feature (Schema_Validation_Feature, True);

   Open ("test1.xml", F);
   R.Parse (F);
   F.Close;
end;
```

Where for instance `test1.xml` contains:

```
<?xml version="1.0" encoding="UTF-8"?>
<root xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:noNamespaceSchemaLocation="algo1.xsd">
   <child>102</child>
</root>
```

CHAPTER

SEVEN

USING THE LIBRARY

XML/Ada is a library. When compiling an application that uses it, you thus need to specify where the spec files are to be found, as well as where the libraries are installed.

There are several ways to do it:

- The simplest is to use the *xmlada-config* script, and let it provide the list of switches for *gnatmake*. This is more convenient on Unix systems, where you can simply compile your application with:

```
gnatmake main.adb `xmlada-config`
```

  Note the use of backticks. This means that *xmlada-config* is first executed, and then the command line is replaced with the output of the script, thus finally executing something like:

```
gnatmake main.adb -Iprefix/include/xmlada -largs -Lprefix/lib \\
  -lxmlada_input_sources -lxmlada_sax -lxmlada_unicode -lxmlada_dom
```

  Unfortunately, this behavior is not available on Windows (unless of course you use a Unix shell). The simplest in that case is to create a `Makefile`, to be used with the *make* command, and copy-paste the output of *xmlada-config* into it.

  xmlada-config has several switches that might be useful:

  - *–sax*: If you this flag, your application will not be linked against the DOM module. This might save some space, particularly if linking statically.
  - *–static*: Return the list of flags to use to link your application statically against Xml/Ada. Your application is then standalone, and you don't need to distribute XMl/Ada at the same time.
  - *–static_sax*: Combines both of the above flags.

  If you are working on a big project, particularly one that includes sources in languages other than Ada, you generally have to run the three steps of the compilation process separately (compile, bind and then link). *xmlada-config* can also be used, provided you use one of the following switches:

  - *–cflags*: This returns the compiler flags only, to be used for instance with *gcc*.
  - *–libs*: This returns the linker flags only, to be used for instance with *gnatlink*.

  This *xmlada-config* method doesn't provide access to the `xml_gtk` module, which is only available when using project files (see below).

- The preferred method, however, is to use the GNAT project files. See the GNAT user's guide for more information on the project files and how to create them for your application.

  Basically, a project file contains the description of your build environment (source directories, object directories, libraries,...).

The very simple case is when you have all your sources in the same directory (say `src/`), and the object files are all generated in the `obj/` directory.

In this case, your project file would look like:

```
with "xmlada";
project Default is
   for Source_Dirs use ("src/");
   for Object_Dir use "obj/";
end Default;
```

and you build your application with:

```
gprbuild -Pdefault main.adb
```

Note in the project file the first line, which indicates that your application requires XML/Ada to build. This will automatically set the appropriate compiler and linker switches to use XML/Ada. Your application will be linker against all modules of XML/Ada (DOM, SAX, ...).

If your application doesn't use DOM, you can replace the first line with something like:

```
with "xmlada_sax";
```

which will reduce the number of libraries that your application is linked with.

WHen you are using project files, you need to let GNAT know where to find the project files. This is done by setting the *ADA_PROJECT_PATH* environment variable, by adding to it the installation directory of XML/Ada, ie the one that contains xmlada.gpr

If the installation prefix is the same as your GNAT installation, and you are using GNAT more recent than 5.03a, then it will automatically find XML/Ada's project files.

Check the `dom/test` directory in the XML/Ada package, which contains both code examples and project files that you can use as a basic for your own code.

The default type of library depends on the way you installed XML/Ada. In all cases, and assuming you installed both static and shared libraries, you can choose among the two by setting the environment variable:

```
LIBRARY_TYPE=static
```

or:

```
LIBRARY_TYPE=relocatable
```

Whatever method you used to build your application, you might have to change, at least one UNIX systems, the environment variable *LD_LIBRARY_PATH* so that it contains the `lib/` directory in the XML/Ada installation, so that the dynamic libraries are correctly found.

This is not needed if you build XML/Ada as a static directory.

7.1 Running on VxWorks

On VxWorks, XML Ada processing might require more stack space than what is typically available from the VxWorks shell, the tasks spawned from there with "sp", or Ada tasks with no or a too small Storage_Size value attached.

Such stack overflow conditions are typically characterized by non-deterministic erratic behavior and can be cured by allocating more stack space for the tasks involved.

INDEX