The Ada95 Unicode and XML Library

Version 1.0
Document revision level $Revision: 1.7 $
Date: $Date: 2003/01/10 13:07:44 $

Emmanuel Briot

1 Introduction

The Extensible Markup Language (XML) is a subset of SGML that is completely described in this document. 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.

??? Explain what XML is

2 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.

Some information was also extracted from the "UTF-8 and Unicode FAQ" by M. Kuhn, available at `???`.

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/8859-1 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 other 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:

1. 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.

2. 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 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.

3 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 called to `Set_System_Id` and `Set_Public_Id`. These are used when reporting error messages.

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 you 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://www.megginson.com/SAX/`.

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 than 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 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 subprogram to get something done. The default implementation for these is to do nothing, exception 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 (see Chapter 3 "The Input module," page 9), 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 (see Chapter 2 "The Unicode module," page 2).

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 standard itself, and there is little point in duplicating this information.

However, 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.

5 The DOM module

A default SAX implementation is provided in the tree_readers file, through its Parse function. This reads an XML stream and creates a tree in memory. The tree can then be manipulated through the DOM module.

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.

What is the Document Object Model?

The Document Object Model is a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents. The document can be further processed and the results of that processing can be incorporated back into the presented page. This is an overview of DOM-related materials here at W3C and around the web.

Why the Document Object Model?

"Dynamic HTML" is a term used by some vendors to describe the combination of HTML, style sheets and scripts that allows documents to be animated. The W3C has received several submissions from members companies on the way in which the object model of HTML documents should be exposed to scripts. These submissions do not propose any new HTML tags or style sheet technology. The W3C DOM WG is working hard to make sure interoperable and scripting-language neutral solutions are agreed upon.

The DOM (Document Object Model) is a set of subprograms to create and manipulate XML trees in memory.

You can create such a tree through the tree_readers.Parse function.

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

6 Using the library

XML/Ada is a library. When compiling an application that uses it, you thus need to specify where the specifications 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:

  1. '`--sax`': If you this flag, your application will not be linked against the DOM module. This might save some space, particularly if linking statically. This also reduces the dependencies on external tools.
  2. '`--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.
  3. '`--static_sax`': Combines both of the above flags.

- On Windows system, you might also simply want to register once and for all the library in the Windows registry, with the command `gnatreg`. This means that `GNAT` will automatically find the installation directory for the XML/Ada.
- 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:

  1. '`--cflags`': This returns the compiler flags only, to be used for instance with `gcc`.
  2. '`--libs`': This returns the linker flags only, to be used for instance with `gnatlink`.

Table of Contents