LuaExpat

XML Expat parsing for the Lua programming language

Introduction

LuaExpat is a **SAX** XML parser based on the **Expat** library. SAX is the *Simple API for XML* and allows programs to:

- process a XML document incrementally, thus being able to handle huge documents without memory penalties;
- register handler functions which are called by the parser during the processing of the document, handling the document elements or text.

With an event-based API like SAX the XML document can be fed to the parser in chunks, and the parsing begins as soon as the parser receives the first document chunk. LuaExpat reports parsing events (such as the start and end of elements) directly to the application through callbacks. The parsing of huge documents can benefit from this piecemeal operation.

LuaExpat is distributed as a library and a file `lom.lua` that implements the **Lua Object Model**.

Building

LuaExpat could be built to Lua 5.0 or to Lua 5.1. In both cases, the language library and headers files for the desired version must be installed properly. LuaExpat also depends on Expat 2.0.0 which should also be installed.

LuaExpat offers a Makefile and a separate configuration file, `config`, which should be edited to suit the particularities of the target platform before running `make`. The file has some definitions like paths to the external libraries, compiler options and the like. One important definition is the version of Lua language, which is not obtained from the installed software.

Installation

The compiled binary file should be copied to a directory in your **C path**. Lua 5.0 users should also install **Compat-5.1**.

Windows users can use the binary version of LuaExpat (`lxp.dll`, compatible with **LuaBinaries**) available at **LuaForge**.

The file `lom.lua` should be copied to a directory in your **Lua path**.

Parser objects

Usually SAX implementations base all operations on the concept of a parser that allows the registration of callback functions. LuaExpat offers the same functionality but uses a different registration method, based on a table of callbacks. This table contains references to the callback functions which are responsible for the handling of the document parts. The parser will assume no behaviour for any undeclared callbacks.

Constructor

lxp.new(*callbacks [, separator]*)

The parser is created by a call to the function **lxp.new**, which returns the created parser or raises a Lua error. It receives the callbacks table and optionally the parser **separator character** used in the namespace expanded element names.

Methods

parser:close()
Closes the parser, freeing all memory used by it. A call to parser:close() without a previous call to parser:parse() could result in an error.

parser:getbase()
Returns the base for resolving relative URIs.

parser:getcallbacks()
Returns the callbacks table.

parser:parse(s)
Parse some more of the document. The string *s* contains part (or perhaps all) of the document. When called without arguments the document is closed (but the parser still has to be closed).
The function returns a non nil value when the parser has been succesfull, and when the parser finds an error it returns five results: nil, *msg*, *line*, *col*, and *pos*, which are the error message, the line number, column number and absolute position of the error in the XML document.

parser:pos()
Returns three results: the current parsing line, column, and absolute position.

parser:setbase(base)
Sets the *base* to be used for resolving relative URIs in system identifiers.

parser:setencoding(encoding)
Set the encoding to be used by the parser. There are four built-in encodings, passed as strings: "US-ASCII", "UTF-8", "UTF-16", and "ISO-8859-1".

Callbacks

The Lua callbacks define the handlers of the parser events. The use of a table in the parser constructor has some advantages over the registration of callbacks, since there is no need for for the API to provide a way to manipulate callbacks.

Another difference lies in the behaviour of the callbacks during the parsing itself. The callback table contains references to the functions that can be redefined at will. The only restriction is that only the callbacks present in the table at creation time will be called.

The callbacks table indices are named after the equivalent Expat callbacks:
CharacterData, *Comment*, *Default*, *DefaultExpand*, *EndCDataSection*, *EndElement*, *EndNamespaceDecl*, *ExternalEntityRef*, *NotStandalone*, *NotationDecl*, *ProcessingInstruction*, *StartCDataSection*, *StartElement*, *StartNamespaceDecl*, and *UnparsedEntityDecl*.

These indices can be references to functions with specific signatures, as seen below. The parser constructor also checks the presence of a field called *_nonstrict* in the callbacks table. If *_nonstrict* is absent, only valid callback names are accepted as indices in the table (Defaultexpanded would be considered an error for example). If *_nonstrict* is defined, any other fieldnames can be used (even if not called at all).

The callbacks can optionally be defined as `false`, acting thus as placeholders for future assignment of functions.

Every callback function receives as the first parameter the calling parser itself, thus allowing the same functions to be used for more than one parser for example.

callbacks.CharacterData = function(parser, string)
Called when the *parser* recognizes an XML CDATA *string*.

callbacks.Comment = function(parser, string)
Called when the *parser* recognizes an XML comment *string*.

callbacks.Default = function(parser, string)
Called when the *parser* has a *string* corresponding to any characters in the document which wouldn't otherwise be handled. Using this handler has the side effect of turning off expansion of references to internally defined general entities. Instead these references are passed to the default handler.

callbacks.DefaultExpand = function(parser, string)
Called when the *parser* has a *string* corresponding to any characters in the document which wouldn't otherwise be handled. Using this handler doesn't affect expansion of internal entity references.

callbacks.EndCdataSection = function(parser)
Called when the *parser* detects the end of a CDATA section.

callbacks.EndElement = function(parser, elementName)
Called when the *parser* detects the ending of an XML element with *elementName*.

callbacks.EndNamespaceDecl = function(parser, namespaceName)
Called when the *parser* detects the ending of an XML namespace with *namespaceName*. The handling of the end namespace is done after the handling of the end tag for the element the namespace is associated with.

callbacks.ExternalEntityRef = function(parser, subparser, base, systemId, publicId)
Called when the *parser* detects an external entity reference.

The *subparser* is a LuaExpat parser created with the same callbacks and Expat context as the *parser* and should be used to parse the external entity.
The *base* parameter is the base to use for relative system identifiers. It is set by parser:setbase and may be nil.
The *systemId* parameter is the system identifier specified in the entity declaration and is never nil.
The *publicId* parameter is the public id given in the entity declaration and may be nil.

callbacks.NotStandalone = function(parser)
Called when the *parser* detects that the document is not "standalone". This happens when there is an external subset or a reference to a parameter entity, but the document does not have standalone set to "yes" in an XML declaration.

callbacks.NotationDecl = function(parser, notationName, base, systemId, publicId)
Called when the *parser* detects XML notation declarations with *notationName*
The *base* parameter is the base to use for relative system identifiers. It is set by parser:setbase and may be nil.
The *systemId* parameter is the system identifier specified in the entity declaration and is never nil.
The *publicId* parameter is the public id given in the entity declaration and may be nil.

callbacks.ProcessingInstruction = function(parser, target, data)
Called when the *parser* detects XML processing instructions. The *target* is the first word in the processing instruction. The *data* is the rest of the characters in it after skipping all whitespace after the initial word.

callbacks.StartCdataSection = function(parser)
Called when the *parser* detects the begining of an XML CDATA section.

callbacks.StartElement = function(parser, elementName, attributes)
Called when the *parser* detects the begining of an XML element with *elementName*.
The *attributes*
parameter is a Lua table with all the element attribute names and values. The table contains an entry for every attribute in the element start tag and entries for the default attributes for that element.

The attributes are listed by name (including the inherited ones) and by position (inherited attributes are not considered in the position list).
As an example if the *book* element has attributes *author*, *title* and an optional *format* attribute (with "printed" as default value),

```
<book author="Ierusalimschy, Roberto" title="Programming in Lua">
```

would be represented as

```
{[1] = "Ierusalimschy, Roberto",
 [2] = "Programming in Lua",
 author = "Ierusalimschy, Roberto",
 format = "printed",
 title = "Programming in Lua"}
```

callbacks.StartNamespaceDecl = function(parser, namespaceName)
Called when the *parser* detects an XML namespace declaration with *namespaceName*. Namespace declarations occur inside start tags, but the StartNamespaceDecl handler is called before the StartElement handler for each namespace declared in that start tag.

callbacks.UnparsedEntityDecl = function(parser, entityName, base, systemId, publicId, notationName)
Called when the *parser* receives declarations of unparsed entities. These are entity declarations that have a notation (NDATA) field.
As an example, in the chunk

```
<!ENTITY logo SYSTEM "images/logo.gif" NDATA gif>
```

entityName would be "logo", *systemId* would be "images/logo.gif" and *notationName* would be "gif". For this example the *publicId* parameter would be nil.
The *base* parameter would be whatever has been set with `parser:setbase`. If not set, it would be nil.

The separator character

The optional separator character in the parser constructor defines the character used in the namespace expanded element names. The separator character is optional (if not defined the parser will not handle namespaces) but if defined it must be different from the character '\0'.

$Id: manual.html,v 1.27 2007/06/05 20:03:12 carregal Exp $