XML Plugin 2.0

Parse XML documents with Hollywood

Andreas Falkenhahn

Table of Contents

1 General information

1.1 Introduction

The XML plugin allows Hollywood scripts to parse XML files easily and efficiently enabling you to make use of this extremely flexible universal markup language that can be used for so many different purposes. The plugin offers a powerful library interface that allows you to access all kinds of data in XML documents like nodes, attributes, namespaces, entities, attlists, CDATA and more.

On top of that, the XML plugin also supports Hollywood's serialization interface which means that you can conveniently serialize Hollywood tables to XML documents by just a single call to Hollywood's `SerializeTable()` function. In the same manner you can also deserialize whole XML documents into Hollywood tables by a single call to Hollywood's `DeserializeTable()` function. It just doesn't get any easier!

1.2 Terms and conditions

1.3 Requirements

- Hollywood 9.0 or better

1.4 Installation

Installing xml.hwp is straightforward and simple: Just copy the file `xml.hwp` for the platform of your choice to Hollywood's plugins directory. On all systems except on AmigaOS and compatibles, plugins must be stored in a directory named `Plugins` that is in the same directory as the main Hollywood program. On AmigaOS and compatible systems, plugins must be installed to `LIBS:Hollywood` instead. On macOS, the `Plugins` directory must be inside the `Resources` directory of the application bundle, i.e. inside the `HollywoodInterpreter.app/Contents/Resources` directory. Note that `HollywoodInterpreter.app` is stored inside the `Hollywood.app` application bundle itself, namely in `Hollywood.app/Contents/Resources`.

On Windows you should also copy the file `xml.chm` to the `Docs` directory of your Hollywood installation. Then you will be able to get online help by pressing F1 when the cursor is over a xml.hwp function in the Hollywood IDE.

On Linux and macOS copy the `xml` directory that is inside the `Docs` directory of xml.hwp's distribution archive to the `Docs` directory of your Hollywood installation. Note that on macOS the `Docs` directory is within the `Hollywood.app` application bundle, i.e. in `Hollywood.app/Contents/Resources/Docs`.

2 About xml.hwp

2.1 Credits

xml.hwp is one of the first Hollywood plugins and was written by Andreas Falkenhahn. It was originally written as a proof-of-concept for Hollywood 5's new library plugin interface. Later it was expanded to support Hollywood 9's new serialization interface to conveniently serialize XML documents to Hollywood tables and the other way round. Thanks have to go to Roberto Ierusalimschy, Andre Carregal and Tomas Guisasola for their LuaExpat plugin on which the library interface of xml.hwp is based.

If you want to contact me, you can either send an e-mail to `andreas@airsoftsoftwair.de` or use the contact form on `http://www.hollywood-mal.com`.

2.2 Frequently asked questions

This section covers some frequently asked questions. Please read them first before asking on the forum because your problem might have been covered here.

Q: Is there a Hollywood forum where I can get in touch with other users?

A: Yes, please check out the "Community" section of the official Hollywood Portal online at `http://www.hollywood-mal.com`.

Q: Where can I ask for help?

A: There's an active forum at `http://forums.hollywood-mal.com`. You're welcome to join it and ask your question there.

Q: I have found a bug.

A: Please post about it in the "Bugs" section of the forum.

2.3 Known issues

Here is a list of things that xml.hwp doesn't support yet or that may be confusing in some way:

– tbd

2.4 Future

Here are some things that are on my to do list:

– add more examples

Don't hesitate to contact me if xml.hwp lacks a certain feature that is important for your project.

2.5 History

Please see the file `history.txt` for a complete change log of xml.hwp.

3 Usage

3.1 Interfaces

There are two ways of using this plugin: Either through the library interface or through the serialization interface. Using the plugin through the serialization interface is easier and very convenient but it comes at the expense of flexibility. Using the plugin through the library interface is a bit more difficult but offers full flexibility. Please see the next two chapters for a brief overview of the two different interfaces.

3.2 Library interface

Using the library interface of xml.hwp gives your script the most flexibility and allows you to access all features of XML documents. The library interface is based on the SAX parsing model which means that you specify a number of callbacks which the XML parser will call while it is parsing your document. One of those callbacks is the `StartElement()` callback. It will be called whenever the parser encounters a new XML tag in the document. As described in its manual page, `StartElement()` receives three arguments: The parser handle, the element name and a table containing the XML attributes specified in the tag. Thus, a `StartElement()` callback could look like this:

```
Function p_StartElement(p, name$, attrs)
   DebugPrint("New tag found:", name$)
   For k,v In Pairs(attrs) Do DebugPrint(k .. "=" .. v)
EndFunction
```

The code above will print the name of the XML tag the parser has just handled as well as all the attributes specified in the XML tag declaration. After having defined the callback, we now have to create a XML parser. This is done using the `xml.CreateParser()` function. You have to pass all callbacks you want to use to it. For our little example, we only want to use the `StartElement()` callback so we create the parser like this:

```
p = xml.CreateParser({StartElement = p_StartElement})
```

As a next thing, we can feed XML data to the parser. This is done by calling the `parser:Parse()` method:

```
p:Parse([[<plugin name="XML" author="A. Falkenhahn" version="1.0"/>]])
```

Running this code will print the plugin's name as well as the attributes `name`, `author` and `version`. You can call `parser:Parse()` as often as you want. When you're finished, don't forget to call `parser:Free()` to free the parser handle. So here's the full example code of our minimal XML parser from above, ready to be copy and pasted into the Hollywood IDE:

```
@REQUIRE "xml"

Function p_StartElement(p, name$, attrs)
   DebugPrint("New tag found:", name$)
   For k,v In Pairs(attrs) Do DebugPrint(k .. "=" .. v)
EndFunction

p = xml.CreateParser({StartElement = p_StartElement})
```

```
p:Parse([[<plugin name="XML" author="A. Falkenhahn" version="1.0"/>]])
p:Free()
```

Of course, there are many other callback types besides `StartElement()`. See Section 4.1 [xmlCreateParser], page 11, for details.

Alternatively, you can also use the plugin's serialization interface. This is easier because it only requires a single function call to convert Hollywood tables to XML documents and vice versa but you won't have fine-tuned control over everything as you have when using the library interface.

See the next chapter for more details on the plugin's serialization interface.

3.3 Serialization interface

If you don't want to use xml.hwp's library interface (see above) for some reason, you can also use the plugin's serialization interface. This is easier to use because it only requires a single function call to convert Hollywood tables to XML documents and vice versa but you won't have fine-tuned control over everything as you have when using the library interface.

Access to the xml.hwp's serialization interface is through Hollywood's `SerializeTable()` and `DeserializeTable()` functions, or, alternatively, through the `ReadTable()` and `WriteTable()` functions. By using the serialization interface, you can convert an XML document into a Hollywood table through just a single function call:

```
t = DeserializeTable(FileToString("test.xml"), "xml")
```

The code above will read all nodes and attributes from `test.xml` and store them in the Hollywood table `t`.

The way the XML data is stored inside the table depends on the serialization mode you have set using `xml.SetSerializeMode()`. The XML plugin supports three different serialization modes:

1. List mode: This will store all XML nodes as sequential list items inside the table. The first XML node will be at index 0, the second at index 1, and so on. This is the default mode. See Section 3.4 [List serialization], page 7, for details.
2. Named mode: This will store all XML nodes as named items inside the table. This means that you can conveniently access nodes by their name instead of having to use numeric indices. The disadvantage of this mode is that you can't have multiple nodes of the same name on the same level because the nodes are stored by name and each name is only available once per node level. Another disadvantage is that you don't have control over the order of the nodes when serializing them back to an XML document. See Section 3.5 [Named serialization], page 8, for details.
3. Hollywood mode: This is a special mode that allows you to serialize arbitrary Hollywood tables. In contrast to the first two modes, Hollywood mode doesn't require the table to follow a certain layout. You can serialize any table in this mode, just as you can with Hollywood's `ReadTable()` and `WriteTable()` functions. The table can even contain binary data or code like Hollywood functions. See Section 3.6 [Hollywood serialization], page 9, for details.

After you have converted an XML file to a Hollywood table, you could then make any modifications you like directly to the Hollywood table. When you're done with all modifications, you can simply convert your Hollywood table back into an XML document in just a single line like this:

```
StringToFile(SerializeTable(t, "xml"), "test2.xml")
```

The code above will convert the table `t` to an XML document using the xml.hwp plugin and save the XML document as `test2.xml`.

As you can see, the serialization interface is very easy to use but doesn't offer as much flexibility as the library interface which gives you fine-tuned control over many XML documents features.

3.4 List serialization

The list (de)serialization mode, which is also the default mode, will store all XML nodes as sequential list items inside the table. The first XML node will be at index 0, the second at index 1, and so on. The table generated by the list deserializer will be a table of tables that contains the whole XML document. The following table fields will be initialized for each node:

`Name`	Name of the XML node.
`Text`	Character data of the XML node.
`Attrs`	Set to a table of subtables containing all attributes as key and value pairs. Each subtable in the `Attrs` table will have the following fields initialized: `Key` The attribute's name. `Value` The attribute's value.
`Nodes`	Set to a table containing the children of the node.

To print all items of the table deserialized from an XML document using `DeserializeTable()` you could use the following recursive function:

```
Function p_PrintNodes(t, indent)
    DebugPrint(RepeatStr(" ", indent) .. "+" .. t.name)
    For Local k = 0 To ListItems(t.attrs) - 1
       DebugPrint(RepeatStr(" ", indent + 1) ..
          "attr: " .. t.attrs[k].key .. "=" .. t.attrs[k].value)
    Next
    If Not EmptyStr(t.text)
       DebugPrint(RepeatStr(" ", indent + 1) .. t.text)
    EndIf
    For Local k = 0 To ListItems(t.nodes) - 1
       p_PrintNodes(t.nodes[k], indent + 4)
    Next
EndFunction

t = DeserializeTable(FileToString("test.xml"), "xml")
p_PrintNodes(t[0], 0)
```

To modify a node, you can then simply change the desired value in the Hollywood table and then serialize it back to XML using Hollywood's `SerializeTable()` function. For example, let's suppose this is your XML file:

```
<?xml version="1.0"?>
<catalog>
   <book id="bk101">
      <author>Gambardella, Matthew</author>
      <title>XML Developer's Guide</title>
      <genre>Computer</genre>
      <price>44.95</price>
      <publish_date>2000-10-01</publish_date>
      <description>An in-depth look at creating applications
      with XML.</description>
   </book>
</catalog>
```

To change the price to 39.95 and save the updated XML to disk, you could do the following:

```
t = DeserializeTable(FileToString("catalog.xml"), "xml")
t[0].nodes[0].nodes[3].text = "39.95"
StringToFile(SerializeTable(t, "xml"), "catalog_new.xml")
```

Note that when using the list serializer, the table you pass to `SerializeTable()` must exactly follow the conventions described above, i.e. the table must contain a number of subtables stored at sequential numeric indices and the fields described above must be initialized for all nodes or there will be an error.

For the example XML shown above, it might be more convenient to use the named serializer, though, because no node names are used twice. See Section 3.5 [Named serialization], page 8, for details.

3.5 Named serialization

The named (de)serialization mode will store all XML nodes as named items inside the table. This means that you can conveniently access nodes by their name instead of having to use numeric indices. The disadvantage of this mode is that you can't have multiple nodes of the same name on the same level because the nodes are stored by name and each name is only available once per node level. Another disadvantage is that you don't have control over the order of the nodes when serializing them back to an XML document.

The following table fields will be initialized for each node when using the named serialization mode:

`Text`	Character data of the XML node.
`Attrs`	Set to a table of subtables containing all attributes as key and value pairs. Each subtable in the `Attrs` table will have the following fields initialized: `Key` The attribute's name. `Value` The attribute's value.
`Nodes`	Set to a table containing the children of the node.

For example, let's suppose this is your XML file:

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

```
      <book id="bk101">
         <author>Gambardella, Matthew</author>
         <title>XML Developer's Guide</title>
         <genre>Computer</genre>
         <price>44.95</price>
         <publish_date>2000-10-01</publish_date>
         <description>An in-depth look at creating applications
         with XML.</description>
      </book>
   </catalog>
```

To change the price to 39.95 and save the updated XML to disk, you could do the following:

```
xml.SetSerializeMode(#XML_SERIALIZEMODE_NAMED)
t = DeserializeTable(FileToString("catalog.xml"), "xml")
t.catalog.nodes.book.nodes.price.text = "39.95"
StringToFile(SerializeTable(t, "xml"), "catalog_new.xml")
```

Note that when using the named serializer, the table you pass to `SerializeTable()` must exactly follow the conventions described above, i.e. the table must contain a number of subtables stored at named indices and the fields described above must be initialized for all nodes or there will be an error.

Also note that since the named serializer stores XML nodes as named table fields, you can only use the same name once per tree level. Thus, you can't use the named serializer to deserialize XMLs that have multiple nodes of the same name, e.g. like this:

```
<?xml version="1.0"?>
<catalog>
   <item>First book</item>
   <item>Second book</item>
   <item>Third book</item>
</catalog>
```

In the XML shown above, there are three `<item>` tags on the same tree level. This can't be deserialized with the named serializer because in Hollywood tables each name can only be used once. To deserialize such tables you have to use the list serializer. See Section 3.4 [List serialization], page 7, for details.

3.6 Hollywood serialization

The Hollywood (de)serialization mode is a special mode that allows you to serialize arbitrary Hollywood tables. In contrast to the first two modes, Hollywood mode doesn't require the table to follow a certain layout. You can serialize any table to XML in this mode, just as you can with Hollywood's `ReadTable()` and `WriteTable()` functions. The table can even contain binary data or code like Hollywood functions.

The disadvantage is that when deserializing XML data back to a Hollywood table in this mode, the XML must follow a certain convention because Hollywood needs to know the type of the data stored inside XML elements. Thus, even though it will probably work in most cases, it's not possible to generally deserialize any arbitrary XML data using the Hollywood serializer. You can only safely deserialize XMLs that have been serialized using the Hollywood serializer before. Here is an example, consider this Hollywood table:

```
t = {1, 2, 3, 4, 5, test = "Hello", subtable = {x = 5, y = 6, z = "8"}}
```

When serializing this table to XML using the Hollywood serializer, it will look like this:

```
<?xml version="1.0" encoding="utf-8"?>
<root>
   <item.0>1</item.0>
   <item.1>2</item.1>
   <item.2>3</item.2>
   <item.3>4</item.3>
   <item.4>5</item.4>
   <test>Hello</test>
   <subtable>
      <y>6</y>
      <x>5</x>
      <z type="string">8</z>
   </subtable>
</root>
```

You can see that sequential table indices are stored in the form of `<item.n>` and that the `<z>` tag has an additional `type` attribute that tells the deserializer that the value is to be stored as a string in the Hollywood table instead of a number. All these things are special conventions of the Hollywood serializer which is why it can't be used to deserialize arbitrary XML documents. The `<root>` tag that is at the top level can be changed to a different tag through the `xml.SetSerializeOptions()` function.

Note that the Hollywood serializer mode also supports binary data. So in case the table contains Hollywood functions or strings that contain binary data, that data will be encoded as Base64 in the XML.

To deserialize an arbitrary Hollywood table to an XML document and then serialize it back to a Hollywood table, you could do the following:

```
xml.SetSerializeMode(#XML_SERIALIZEMODE_HOLLYWOOD)
StringToFile(SerializeTable(t, "xml"), "test.xml")
copy_of_t = DeserializeTable(FileToString("test.xml"), "xml")
```

4 Function reference

4.1 xml.CreateParser

NAME

xml.CreateParser – create new parser

FORMERLY KNOWN AS

xmlparser.New (V1.x)

SYNOPSIS

```
p = xml.CreateParser(t[, sep$, merge])
```

FUNCTION

This function will create a new parser that can be used to iterate over XML documents. You have to pass a table that contains a selection of callback functions that should be called while the XML document is being parsed. The following callback functions can currently be passed in the table argument:

`AttlistDecl`
: The function you specify here will be called for attlist declarations in the DTD. See Section 6.1 [AttlistDecl], page 21, for details. (V2.0)

`CharacterData`
: The function you specify here will be called for all data that is not part of any XML entity. See Section 6.2 [CharacterData], page 22, for details.

`Comment`
: The function you specify here will be called when the parser encounters a comment in the XML. See Section 6.3 [Comment], page 22, for details.

`DefaultExpand`
: This is the same as `DefaultHandler` except that the handler doesn't inhibit the expansion of internal entity references. See Section 6.4 [DefaultExpand], page 23, for details.

`DefaultHandler`
: The function you specify here will be called when the parser encounters characters in the document which wouldn't otherwise be handled. See Section 6.5 [DefaultHandler], page 23, for details.

`ElementDecl`
: The function you specify here will be called for element declarations in a DTD. See Section 6.6 [ElementDecl], page 24, for details. (V2.0)

`EndCDataSection`
: The function you specify here will be called when the parser detects the end of an XML CDATA section. See Section 6.7 [EndCDataSection], page 25, for details.

`EndDocTypeDecl`
: The function you specify here will be called when the parser reaches the end of a DOCTYPE declaration. See Section 6.8 [EndDocTypeDecl], page 26, for details. (V2.0)

`EndElement`
: The function you specify here will be called when an XML node is closed. See Section 6.9 [EndElement], page 26, for details.

`EndNamespaceDecl`
: The function you specify here will be called when the parser detects the end of an XML namespace declaration. Note that you must specify the optional `sep$` parameter if you want the parser to handle namespaces. See Section 6.10 [EndNamespaceDecl], page 26, for details.

`EntityDecl`
: The function you specify here will be called when the parser detects an entity declaration. See Section 6.11 [EntityDecl], page 27, for details. (V2.0)

`ExternalEntityRef`
: The function you specify here will be called when the parser detects an external entity reference. See Section 6.12 [ExternalEntityRef], page 27, for details.

`NotationDecl`
: The function you specify here will be called when the parser encounters a notation declaration. See Section 6.13 [NotationDecl], page 28, for details.

`NotStandalone`
: The function you specify here will be called if the document is not standalone without indicating so in the XML declaration. See Section 6.14 [NotStandalone], page 29, for details.

`ProcessingInstruction`
: The function you specify here will be called when the parser detects XML processing instructions. See Section 6.15 [ProcessingInstruction], page 29, for details.

`SkippedEntity`
: The function you specify here will be called when entities are skipped. See Section 6.16 [SkippedEntity], page 30, for details. (V2.0)

`StartCDataSection`
: The function you specify here will be called when the parser detects the beginning of an XML CDATA section. See Section 6.17 [StartCDataSection], page 30, for details.

`StartDocTypeDecl`
: The function you specify here will be called when the parser reaches the DOCTYPE declaration. See Section 6.18 [StartDocTypeDecl], page 30, for details.

`StartElement`
: The function you specify here will be called when the parser encounters a new XML node. See Section 6.19 [StartElement], page 31, for details.

`StartNamespaceDecl`
: The function you specify here will be called when the parser detects an XML namespace declaration. Note that you must specify the optional `sep$`

parameter if you want the parser to handle namespaces. See Section 6.20 [StartNamespaceDecl], page 32, for details.

`UnparsedEntityDecl`
: The function you specify here will be called when the parser detects declarations of unparsed entities. See Section 6.21 [UnparsedEntityDecl], page 32, for details. Note that `UnparsedEntityDecl` is obsolete and you should use `EntityDecl` instead (see above).

`XMLDecl`
: The function you specify here will be called when the parser detects XML declarations. See Section 6.22 [XMLDecl], page 33, for details. (V2.0)

The optional separator character `sep$` can be used to define the character used in the namespace expanded element names. If `sep$` isn't defined, the parser will not handle namespaces.

The optional `merge` parameter can be used to configure whether or not the `CharacterData` callback should try to merge as much data as possible into a single string. This defaults to `True`. If you set this to `False`, your `CharacterData` callback could get called more often because the character data isn't merged into larger chunks.

This function will return a parser object. You can call `parser:Parse()` to parse some XML using the parser object. Once you're done with parsing, call `parser:Free()` to free the parser object.

INPUTS

`t` table containing one or more callback functions (see above)

`sep$` optional: separator character

`merge` optional: whether or not the parser should try to merge character data into bigger chunks of text (defaults to `True`) (V2.0)

RESULTS

`p` a parser object

EXAMPLE

See Section 6.19 [StartElement], page 31.

4.2 xml.SetSerializeMode

NAME

xml.SetSerializeMode – set serialization mode (V2.0)

SYNOPSIS

```
xml.SetSerializeMode(mode)
```

FUNCTION

This can be used to set the desired serialization mode when using the plugin through its serialization interface. You have to pass the desired serialization mode in the `mode` argument. The following serialization modes are currently supported:

`#XML_SERIALIZEMODE_LIST`
: This will store all XML nodes as sequential list items inside the table. The first XML node will be at index 0, the second at index 1, and so on. This is the default mode. See Section 3.4 [List serialization], page 7, for details.

`#XML_SERIALIZEMODE_NAMED`
: This will store all XML nodes as named items inside the table. This means that you can conveniently access nodes by their name instead of having to use numeric indices. The disadvantage of this mode is that you can't have multiple nodes of the same name on the same level because the nodes are stored by name and each name is only available once per node level. Another disadvantage is that you don't have control over the order of the nodes when serializing them back to an XML document. See Section 3.5 [Named serialization], page 8, for details.

`#XML_SERIALIZEMODE_HOLLYWOOD`
: This is a special mode that allows you to serialize arbitrary Hollywood tables. In contrast to the first two modes, Hollywood mode doesn't require the table to follow a certain layout. You can serialize any table in this mode, just as you can with Hollywood's `ReadTable()` and `WriteTable()` functions. The table can even contain binary data or code like Hollywood functions. See Section 3.6 [Hollywood serialization], page 9, for details.

Further options can be configured using the `xml.SetSerializeOptions()` command. See Section 4.3 [xml.SetSerializeOptions()], page 14, for details.

INPUTS

`mode` desired serialization mode (see above for possible values)

4.3 xml.SetSerializeOptions

NAME

xml.SetSerializeOptions – set serialization options (V2.0)

SYNOPSIS

`xml.SetSerializeOptions(t)`

FUNCTION

This can be used to set certain options when using the plugin through its serialization interface. The sole argument taken by this function is a table that can contain one or more of the following items:

`RootNode` This can be used to set the name of the XML's root node. This is only used if the serialization mode is `#XML_SERIALIZEMODE_HOLLYWOOD`. It defaults to `"root"`.

`ForceLowerCase`
: When using the list or named serialization modes, the XML plugin will always convert all node names to lower case. If you don't want that, set this table item to `False`. The default value is `True`. This tag is ignored for the Hollywood serialization mode.

`IgnoreWhiteSpace`
When using the list or named serialization modes, the XML plugin will set the `Text` field of nodes that contain nothing but whitespace characters (e.g. space, tab, line breaks...) to an empty string. If you don't want that, set this item to `False`. The default value is `True`. This tag is ignored for the Hollywood serialization mode.

INPUTS

`t`	table containing desired serialization options

5 Parser methods

5.1 parser:Free

NAME

parser:Free – free parser

FORMERLY KNOWN AS

parser:Close (V1.x)

SYNOPSIS

`parser:Free()`

FUNCTION

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

INPUTS

none

EXAMPLE

See Section 6.19 [StartElement], page 31.

5.2 parser:GetBase

NAME

parser:GetBase – get base for relative URIs

SYNOPSIS

`s$ = parser:GetBase()`

FUNCTION

Gets the base to be used for resolving relative URIs in system identifiers.

INPUTS

none

RESULTS

`s$`	base string

5.3 parser:GetCallbacks

NAME

parser:GetCallbacks – get parser callbacks

SYNOPSIS

`cb = parser:GetCallbacks()`

FUNCTION

This simply returns a table containing all the callbacks that were specified when creating the parser using `xml.CreateParser()`.

INPUTS

none

RESULTS

`cb`	table containing parser callbacks

5.4 parser:GetError

NAME

parser:GetError – get extended error information (V2.0)

SYNOPSIS

```
s$, line, col, pos = parser:GetError()
```

FUNCTION

If `parser:Parse()` returns an error, you can use this method to get extended error information. The method returns a human-readable error message in `s$`, the line and column index of the error in `line` and `col` and the byte index of the error in `pos`.

INPUTS

none

RESULTS

`s$`	error message
`line`	line index (starting from 1)
`col`	column index (starting from 1)
`pos`	byte index of the error (starting from 1)

5.5 parser:GetPosition

NAME

parser:GetPosition – get current parsing position

FORMERLY KNOWN AS

parser:Pos (V1.x)

SYNOPSIS

```
line, col, pos = parser:GetPosition()
```

FUNCTION

This method returns the current parser position. The line and column indices are returned in `line` and `col` and the byte index is returned in `pos`.

INPUTS

none

RESULTS

`line`	current line index (starting from 1)

`col`	current column index (starting from 1)
`pos`	current byte index (starting from 1)

5.6 parser:Parse

NAME

parser:Parse – parse XML code

SYNOPSIS

```
ok = parser:Parse(s$)
```

FUNCTION

This method parses XML code. The string `s$` contains part (or perhaps all) of the document to be parsed. The method returns a boolean value indicating success or failure. If `parser:Parse()` fails, you can use `parser:GetError()` to get extended error information.

INPUTS

`s$`	XML code to parse

RESULTS

`ok`	boolean value indicating success or failure

EXAMPLE

See Section 6.19 [StartElement], page 31.

5.7 parser:SetBase

NAME

parser:SetBase – set base for relative URIs

SYNOPSIS

```
parser:SetBase(s$)
```

FUNCTION

Sets the base to be used for resolving relative URIs in system identifiers to the string passed in `s$`.

INPUTS

`s$`	desired base

5.8 parser:SetEncoding

NAME

parser:SetEncoding – set parser encoding

SYNOPSIS

```
parser:SetEncoding(e$)
```

FUNCTION

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". `parser:SetEncoding()` must not be called after parsing has already been started with `parser:Parse()`.

INPUTS

`e$`	encoding that should be used by the parser

5.9 parser:Stop

NAME

parser:Stop – stop the parser

SYNOPSIS

```
ok = parser:Stop()
```

FUNCTION

Abort the parser and prevent it from parsing any further through the data it was last passed. Use this to halt parsing the document when an error is discovered inside a callback, for example. The parser object cannot accept more data after this call. The method returns a boolean value indicating success or failure.

INPUTS

none

RESULTS

`ok`	boolean value indicating success or failure

6 Callback reference

6.1 AttlistDecl

NAME

AttlistDecl – handler for attlist declarations in the DTD (V2.0)

SYNOPSIS

`AttlistDecl(p, elname$, attname$, atttype$, dflt$, isrequired)`

FUNCTION

This function is called for attlist declarations in the DTD. It is called for each attribute. So a single attlist declaration with multiple attributes declared will generate multiple calls to this handler. The `elname$` parameter returns the name of the element for which the attribute is being declared. The attribute name is in the `attname$` parameter. The attribute type is in the `atttype$` parameter. It is the string representing the type in the declaration with whitespace removed.

The `dflt$` parameter holds the default value. It will be `Nil` in the case of `#IMPLIED` or `#REQUIRED` attributes. You can distinguish these two cases by checking the `isrequired` parameter, which will be `True` in the case of `#REQUIRED` attributes. Attributes which are `#FIXED` will have also have a `True isrequired`, but they will have the non-Nil fixed value in the `dflt` parameter.

PARAMETERS

`p`	parser handle
`elname$`	name of the element for which the attribute is being declared
`attname$`	attribute name
`attype$`	attribute type
`dflt$`	default value
`isrequired`	`True` or `False` depending on whether the attribute is required

EXAMPLE

```
Function p_AttlistDecl(p, elname$, attname$, atttype$, dflt$, isreq)
   DebugPrint(elname$, attname$, atttype$, dflt$, isreq)
EndFunction

p = xml.CreateParser({AttlistDecl = p_AttlistDecl})
p:Parse([[
<?xml version="1.0" standalone="yes"?>
<!DOCTYPE lab_group [
   <!ELEMENT student_name (#PCDATA)>
   <!ATTLIST student_name student_no ID #REQUIRED>
   <!ATTLIST student_name tutor_1 IDREF #IMPLIED>
   <!ATTLIST student_name tutor_2 IDREF #IMPLIED>
]>
<root/>
]])
p:Free()
```

The code above shows how to handle attlist declarations.

6.2 CharacterData

NAME

CharacterData – parser has encountered character data

SYNOPSIS

```
CharacterData(p, s$)
```

FUNCTION

This function is called whenever the parser encounters data that is not part of any XML entity. The data is passed to your callback as a string in the `s$` parameter.

Note that the character data for a single XML element might be passed to your callback in several separate chunks so make sure your code doesn't depend on getting all of the character data at once. By default, the XML plugin tries to merge character data into as few chunks of text as possible. You can turn off the merging functionality by passing `False` in the optional third argument in your call to `xml.CreateParser()`. Note that even with the merge data option enabled (which is the default) it isn't guaranteed that all data will come in at once so be prepared to handle this correctly.

PARAMETERS

`p`	parser handle
`s$`	the character data

EXAMPLE

```
Function p_CharacterData(p, s$)
   DebugPrint(s$)
EndFunction

p = xml.CreateParser({CharacterData = p_CharacterData})
p:Parse([[<app>Hollywood</app>]])
p:Free()
```

The code above will print "Hollywood".

6.3 Comment

NAME

Comment – parser has encountered a comment

SYNOPSIS

```
Comment(p, s$)
```

FUNCTION

This function is called whenever the parser encounters a comment in the XML. The comment data is passed to your callback as a string in the `s$` parameter.

PARAMETERS

`p`	parser handle
`s$`	the comment data

EXAMPLE

```
Function p_Comment(p, s$)
   DebugPrint(s$)
EndFunction

p = xml.CreateParser({Comment = p_Comment})
p:Parse([[
    <!--I'm a comment-->
    <app>Hollywood</app>
]])
p:Free()
```

The code above will print "I'm a comment".

6.4 DefaultExpand

NAME

DefaultExpand – default handler with expansions

SYNOPSIS

```
DefaultExpand(p, s$)
```

FUNCTION

This does the same as `DefaultHandler()` except that this handler doesn't inhibit the expansion of internal entity references. The entity reference will not be passed to the default handler.

PARAMETERS

`p`	parser handle
`s$`	the character data

6.5 DefaultHandler

NAME

DefaultHandler – default handler for other characters

SYNOPSIS

```
DefaultHandler(p, s$)
```

FUNCTION

This function is called for any characters in the document which wouldn't otherwise be handled. This includes both data for which no handlers can be set (like some kinds of DTD declarations) and data which could be reported but which currently has no handler set. The characters are passed exactly as they were present in the XML document except that they will be encoded in UTF-8. Line boundaries are not normalized. Note that a byte order mark character is not passed to the default handler. There are no guarantees about how characters are divided between calls to the default handler: for example, a comment might be split between multiple calls. Setting the handler with this call has the side effect of turning off expansion of references to internally defined general entities. Instead these references are passed to the default handler.

PARAMETERS

`p`	parser handle
`s$`	the character data

6.6 ElementDecl

NAME

ElementDecl – handler for element declarations in the DTD (V2.0)

SYNOPSIS

`ElementDecl(p, name$, type, quantifier, children)`

FUNCTION

This function is called for element declarations in a DTD. The name of the element is passed in the `name$` parameter. If `type` equals `#XML_CTYPE_EMPTY` or `#XML_CTYPE_ANY`, then `quantifier` will be `#XML_CQUANT_NONE`, and the `children` table will be `Nil`.

If `type` is `#XML_CTYPE_MIXED`, then `quantifier` will be `#XML_CQUANT_NONE` or `#XML_CQUANT_REP` and the `children` table will contain the elements that are allowed to be mixed in. All those children will have type `#XML_CTYPE_NAME` with no quantification then. Only the root node can be of type `#XML_CTYPE_EMPTY`, `#XML_CTYPE_ANY`, or `#XML_CTYPE_MIXED`.

For type `#XML_CTYPE_NAME`, the `name$` parameter will contain the name and the `children` table will be `Nil`. The `quantifier` parameter will indicate any quantifiers placed on the name.

Types `#XML_CTYPE_CHOICE` and `#XML_CTYPE_SEQ` indicate a choice or sequence respectively. The `children` table then contains the nodes in the choice or sequence.

The children passed in the `children` table will have the fields `name`, `type`, `quantifier`, and `children` set. They correspond to the parameters of the same name passed to `ElementDecl()`.

Possible values for the `type` parameter/field:

```
#XML_CTYPE_EMPTY
#XML_CTYPE_ANY
#XML_CTYPE_MIXED
```

```
#XML_CTYPE_NAME
#XML_CTYPE_CHOICE
#XML_CTYPE_SEQ
```

Possible values for the `quantifier` parameter/field:

```
#XML_CQUANT_NONE
#XML_CQUANT_OPT
#XML_CQUANT_REP
#XML_CQUANT_PLUS
```

PARAMETERS

`p`	parser handle
`name$`	name of the element
`type`	element type
`quantifier`	element quantifier
`children`	child nodes or `Nil`

EXAMPLE

```
Function p_ElementDecl(p, elname$, type, quant, children)
   DebugPrint(elname$, type, quant, children)
EndFunction

p = xml.CreateParser({ElementDecl = p_ElementDecl})
p:Parse([[
<!DOCTYPE student [
   <!ELEMENT student (id|surname)>
   <!ELEMENT id (#PCDATA)>
]>
<student>
   <id>9216735</id>
</student>
]])
p:Free()
```

The code above shows how to handle attlist declarations.

6.7 EndCDataSection

NAME

EndCDataSection – end of CDATA section has been reached

SYNOPSIS

```
EndCDataSection(p)
```

FUNCTION

This function is called when the parser detects the end of an XML CDATA section. In the callback, you can use `parser:GetPosition()` to get the current parser position.

PARAMETERS

`p`	parser handle

6.8 EndDocTypeDecl

NAME

EndDocTypeDecl – parser has reached the end of a DOCTYPE declaration (V2.0)

SYNOPSIS

`EndDocTypeDecl(p)`

FUNCTION

This function is called when the parser detects the end of the DOCTYPE declaration when the closing `>` is encountered, but after processing any external subset.

PARAMETERS

`p`	parser handle

6.9 EndElement

NAME

EndElement – XML element has been closed

SYNOPSIS

`EndElement(p, name$)`

FUNCTION

This function is called whenever an XML element is closed. The name of the XML element is passed in `name$`.

PARAMETERS

`p`	parser handle
`name$`	name of the XML element

EXAMPLE

See Section 6.19 [StartElement], page 31.

6.10 EndNamespaceDecl

NAME

EndNamespaceDecl – parser has detected the end of a namespace declaration

SYNOPSIS

`EndNamespaceDecl(p, name$)`

FUNCTION

This function is called when leaving the scope of the namespace declaration specified by `name$`. This handler will be called, for each namespace declaration, after the handler for the end tag of the element in which the namespace was declared.

Note that you must specify the optional `sep$` parameter in your call to `xml.CreateParser()` if you want the parser to handle namespaces.

PARAMETERS

`p`	parser handle
`name$`	namespace name

6.11 EntityDecl

NAME

EntityDecl – parser has detected an entity declaration

SYNOPSIS

```
EntityDecl(p, name$, isparm, value$, base$, sysid$, pubid$, notation$)
```

FUNCTION

This function is called for entity declarations. The `isparm` argument will be `True` if the entity is a parameter entity, `False` otherwise. For internal entities, e.g.

```
(<!ENTITY foo "bar">)
```

`value$` will be a string and `sysid$`, `pubid$`, and `notation$` will be `Nil`. The `value$` string can be `Nil`, as well as an empty string, which is a valid value. For external entities, `value$` will be `Nil` and `sysid$` will be a string. The `pubid$` argument will be `Nil` unless a public identifier was provided. The `notation$` argument will have a string value only for unparsed entity declarations.

The `base$` parameter will be set to whatever has been set with `parser:SetBase()`. If not set, it will be `Nil`.

PARAMETERS

`p`	parser handle
`name$`	entity name
`isparm`	`True` if the entity is a parameter entity
`base$`	base to use for relative system identifiers
`sysid$`	system ID
`pubid$`	public ID
`notation$`	notation name

6.12 ExternalEntityRef

NAME

ExternalEntityRef – parser has detected an external entity reference

SYNOPSIS

```
ExternalEntityRef(p, subparser, base$, sysid$, pubid$)
```

FUNCTION

This function is called when the parser detects an external entity reference. The `subparser` parameter contains a parser handle created with the same callbacks and context as the main parser and should be used to parse the external entity. The `base$` parameter is the base to use for relative system identifiers. It is set by `parser:SetBase()` and may be `Nil`. The `sysid$` parameter is the system identifier specified in the entity declaration and is never `Nil`. The `pubid$` parameter is the public ID given in the entity declaration and may be `Nil`.

PARAMETERS

`p`	parser handle
`subparser`	subparser handle for parsing the external entity
`base$`	base to use for relative system identifiers
`sysid$`	system ID
`pubid$`	public ID

6.13 NotationDecl

NAME

NotationDecl – parser has detected a notation declaration

SYNOPSIS

```
NotationDecl(p, name$, base$, sysid$, pubid$)
```

FUNCTION

This function is called when the parser detects an XML notation declaration. The notation name is passed in the `name$` parameter. The `base$` parameter is the base to use for relative system identifiers. It is set by `parser:SetBase()` and may be `Nil`. The `sysid$` parameter is the system identifier specified in the entity declaration and is never `Nil`. The `pubid$` parameter is the public ID given in the entity declaration and may be `Nil`.

PARAMETERS

`p`	parser handle
`name$`	notation name
`base$`	base to use for relative system identifiers
`sysid$`	system ID
`pubid$`	public ID

6.14 NotStandalone

NAME

NotStandalone – handle non-standalone documents

SYNOPSIS

```
ok = NotStandalone(p)
```

FUNCTION

This function is called when the parser detects that the document is not "stand-alone". 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. This callback expects a return value, if the callback returns `False`, parsing will be aborted.

PARAMETERS

`p`	parser handle

RESULTS

`ok`	return `True` to continue parsing, `False` to abort

6.15 ProcessingInstruction

NAME

ProcessingInstruction – parser is processing instruction

SYNOPSIS

```
ProcessingInstruction(p, target$, data$)
```

FUNCTION

This function is 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.

PARAMETERS

`p`	parser handle
`target$`	first word in the processing instruction
`data$`	rest of the characters after the initial word

EXAMPLE

```
Function p_ProcessingInstruction(p, target$, data$)
   DebugPrint(target$, data$)
EndFunction

p = xml.CreateParser({ProcessingInstruction = p_ProcessingInstruction})
p:Parse([[<foo><?Hollywood rocks?></foo>]])
p:Free()
```

The code above prints "Hollywood rocks".

6.16 SkippedEntity

NAME

SkippedEntity – parser has skipped an entity (V2.0)

SYNOPSIS

`SkippedEntity(p, name$, isparm)`

FUNCTION

This function is called in two situations:

1. An entity reference is encountered for which no declaration has been read and this is not an error.
2. An internal entity reference is read, but not expanded, because `DefaultHandler()` has been called.

The `isparm` argument will be `True` for a parameter entity and `False` for a general entity.

Note: Skipped parameter entities in declarations and skipped general entities in attribute values cannot be reported, because the event would be out of sync with the reporting of the declarations or attribute values.

PARAMETERS

`p`	parser handle
`name$`	entity name
`isparm`	`True` for parameter entities, `False` for general entities

6.17 StartCDataSection

NAME

StartCDataSection – CDATA section is about to be parsed

SYNOPSIS

`StartCDataSection(p)`

FUNCTION

This function is called when the parser detects the beginning of an XML CDATA section. In the callback, you can use `parser:GetPosition()` to get the current parser position.

PARAMETERS

`p`	parser handle

6.18 StartDocTypeDecl

NAME

StartDocTypeDecl – parser has reached the DOCTYPE declaration

SYNOPSIS

`StartDocTypeDecl(p, name$, sysid$, pubid$, subset)`

FUNCTION

This function is called at the start of a DOCTYPE declaration, before any external or internal subset is parsed. The callback receives the DOCTYPE name in `name$`, the system ID in `sysid$`, the public ID in `pubid$`. The `subset` parameter will be `True` if the DOCTYPE declaration has an internal subset.

PARAMETERS

`p`	parser handle
`name$`	DOCTYPE name
`sysid$`	system ID
`pubid$`	public ID
`subset`	`True` if DOCTYPE declaration has an internal subset

6.19 StartElement

NAME

StartElement – an XML element has been found

SYNOPSIS

```
StartElement(p, name$, attrs)
```

FUNCTION

This function is called whenever a new XML element is opened. The name of the XML element is passed in `name$`. The `attrs` parameter is a table with all the element attribute names and values. The table contains an entry for every attribute in the element start tag. The attribute's name is used as the table index.

PARAMETERS

`p`	parser handle
`name$`	name of the XML element
`attrs`	table containing all element attributes

EXAMPLE

```
Function p_StartElement(p, name$, attrs)
   DebugPrint("Open:", name$, attrs.name, attrs.author)
EndFunction
Function p_EndElement(p, name$)
   DebugPrint("Close:", name$)
EndFunction

p = xml.CreateParser({StartElement = p_StartElement,
  EndElement = p_EndElement})
p:Parse([[<plugin name="XML" author="Andreas Falkenhahn"/>]])
p:Free()
```

The code above will print "Open: plugin XML Andreas Falkenhahn" and then "Close: plugin".

6.20 StartNamespaceDecl

NAME

StartNamespaceDecl – parser has detected a namespace declaration

SYNOPSIS

```
StartNamespaceDecl(p, name$, uri$)
```

FUNCTION

This function is called when a namespace is declared. The callback will receive the name and URI of the namespace in the `name$` and `uri$` parameters. Note that even though namespace declarations occur inside start tags the namespace declaration start handler is called before the start tag handler for each namespace declared in that start tag.

Note that you must specify the optional `sep$` parameter in your call to `xml.CreateParser()` if you want the parser to handle namespaces.

PARAMETERS

`p`	parser handle
`name$`	namespace name
`uri$`	namespace URI

EXAMPLE

```
Function p_StartNamespaceDecl(p, name$, uri$)
   DebugPrint(name$, uri$)
EndFunction

p = xml.CreateParser({StartNamespaceDecl = p_StartNamespaceDecl}, "?")
p:Parse([[<foo xmlns:space='a/namespace'/>]])
p:Free()
```

The code above will print "space a/namespace".

6.21 UnparsedEntityDecl

NAME

UnparsedEntityDecl – parser has detected an unparsed entity declaration

SYNOPSIS

```
UnparsedEntityDecl(p, name$, base$, sysid$, pubid$, notation$)
```

FUNCTION

This function is 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>
```

the `name$` parameter would be "logo", `sysid$` would be "images/logo.gif" and `notation$` would be "gif". For this example the `pubid$` parameter would be `Nil`. The `base$` parameter would be whatever has been set with `parser:SetBase()`. If not set, it would be `Nil`.

Note that `UnparsedEntityDecl()` is obsolete and you should use `EntityDecl()` instead.

PARAMETERS

`p`	parser handle
`name$`	entity name
`base$`	base to use for relative system identifiers
`sysid$`	system ID
`pubid$`	public ID
`notation$`	notation name

6.22 XMLDecl

NAME

XMLDecl – parser has reached an XML declaration (V2.0)

SYNOPSIS

`XMLDecl(p, version$, encoding$, standalone)`

FUNCTION

This function is called s called for XML declarations and also for text declarations discovered in external entities. The way to distinguish is that the `version$` parameter will be `Nil` for text declarations. The `encoding$` parameter may be `Nil` for an XML declaration. The `standalone` argument will contain -1, 0, or 1 indicating respectively that there was no standalone parameter in the declaration, that it was given as no, or that it was given as yes.

PARAMETERS

`p`	parser handle
`version$`	version in XML declaration
`encoding$`	encoding in XML declaration
`standalone`	stand-alone state of the declaration, can be 0, -1 or 1 (see above)

EXAMPLE

```
Function p_XMLDecl(p, version$, encoding$, standalone)
   DebugPrint(version$, encoding$, standalone)
EndFunction

p = xml.CreateParser({XMLDecl = p_XMLDecl})
p:Parse([[<?xml version="1.0" encoding="ISO-8859-1"?><body/>]])
p:Free()
```

The code above will prints "1.0 ISO-8859-1 -1".

Appendix A Licenses

A.1 Expat license

A.2 LuaExpat license

LuaExpat is free software: it can be used for both academic and commercial purposes at absolutely no cost. There are no royalties or GNU-like "copyleft" restrictions. LuaExpat qualifies as Open Source software. Its licenses are compatible with GPL. LuaExpat is not in the public domain and the Kepler Project keep its copyright. The legal details are below.

The spirit of the license is that you are free to use LuaExpat for any purpose at no cost without having to ask us. The only requirement is that if you do use LuaExpat, then you should give us credit by including the appropriate copyright notice somewhere in your product or its documentation.

The LuaExpat library is designed and implemented by Roberto Ierusalimschy. The implementation is not derived from licensed software.

Index