Lua bindings for libxml2

Peter S. Housel
August 2003–January 2004

Copyright

Table of Contents

Introduction

Many bindings have been provided for the Lua extension language. This library provides a rudimentary Lua binding for Daniel Veillard's libxml (also known as gnome-xml). This binding focuses on the DOM-style interface to libxml, and provides a “DOM-inspired” interface. (The gdome library was used as a reference for implementing DOM methods on top of the libxml base.)

⟨Module `lua-libxml.h` 1⟩≡

```
#ifndef LUA_LIBXML_H
#define LUA_LIBXML_H

int luaopen_libxml(lua_State *L);

#endif
```

⟨Module `lua-libxml.c` 1⟩≡

```
#include <stddef.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>

#ifdef DEBUG
#include <stdio.h>
#endif

#include <lua.h>
#include <lauxlib.h>

#include <libxml/tree.h>
#include <libxml/parser.h>
#include <libxml/globals.h>
```

⟨Other includes 17⟩

```
#include "lua-libxml.h"
```

⟨Implementation declarations 2⟩

⟨C implementation definitions 4⟩

```
int
luaopen_libxml(lua_State *L) {
  static const struct luaL_reg libxml[] = {
```

⟨Functions registered in the `libxml` table 12⟩

```
    {NULL, NULL}
  };
  static const struct luaL_reg libxslt[] = {
```

⟨Functions registered in the `libxsltl` table 20⟩

```
    {NULL, NULL}
  };
```

⟨Initialize libxml 2⟩

```
  luaL_openlib(L, "libxml", libxml, 0);
```

```
  luaL_openlib(L, "libxslt", libxslt, 0);
  return 2;
}
```

DTDs are good. Fully expanded entities are also good.

⟨Initialize libxml2⟩≡
```
xmlLoadExtDtdDefaultValue = 1;
xmlSubstituteEntitiesDefault(1);
```

This section continues on pages 2, 3 and 19.

This section is referenced on page 1.

Namespace Construction

The Document Object Model (DOM) defines a module named `dom` defining a number of interfaces such as `Node`, `NodeList`, `Document`, and so on. Here we define our equivalent of this namespace layout, with a global `dom` table.

⟨Initialize libxml 2⟩+≡
```
lua_pushstring(L, "dom");
lua_newtable(L);
```
⟨Insert namespace items into the `dom` table 3⟩
```
lua_settable(L, LUA_GLOBALSINDEX);
```

Memory Management

Pointers from Lua to C data structures require the use of either light or full userdata. In order to allow the use of metatable methods, full userdata items must be used.

Since libxml handles allocating its own data structures, the only thing we need to store in our userdata items is a pointer to these structures—in effect a “boxed pointer”.

⟨Implementation declarations2⟩≡
```
struct box {
  void *ptr;
};
```

This section continues on pages 2, 3, 3, 5, 6, 8, 11, 13, 17, 18, 19, 19, 20 and 21.

This section is referenced on page 1.

We want to insure that a unique boxed pointer full userdata exists for any given libxml object. We'll do this using a weak table, keyed on a light userdata also pointing to the object. We'll locate the weak table by storing it in the registry, keyed on a light userdata pointer to some random address.

⟨Implementation declarations 2⟩+≡
```
static int unique_table;
```

⟨Initialize libxml 2⟩+≡
```
lua_pushlightuserdata(L, &unique_table);
lua_newtable(L);                /* weak table */
lua_newtable(L);                /* metatable for weak table */
lua_pushliteral(L, "__mode");
lua_pushliteral(L, "kv");       /* both keys and values are weak */
lua_settable(L, -3);
lua_setmetatable(L, -2);
lua_settable(L, LUA_REGISTRYINDEX);
```

There is no automatic memory managment in libxml; nodes belong to the document they are a part of and are usually only freed when the entire document is freed. This means that we can not free anything when the `"gc"` method of an `xmlNode` internal to a document is called. Furthermore, we cannot free the document when its `"gc"` method is called if there are still any userdata containing internal pointers.

Fortunately each node contains a pointer to the document it belongs to. We can use this to maintain a reference count of userdata items pointing to internal nodes. When the last reference to a node in the document is collected, then we can free the entire document.

⟨Implementation declarations 2⟩+≡
```
struct document_private {
  unsigned refcount;
};
```

The DOM Node and NodeList Interfaces

The base interface for most of the DOM interfaces we're interested in is the `Node` interface.

⟨Insert namespace items into the `dom` table3⟩≡
```
lua_pushstring(L, "Node");
lua_newtable(L);
```
⟨Insert namespace items into the `dom.Node` table 4⟩

⟨Register items that are subtypes of `dom.Node` 9⟩
```
lua_settable(L, -3);
```

This section continues on page 6.

This section is referenced on page 2.

The following values can be returned for the `nodeType` attribute:

⟨Implementation declarations 2⟩+≡
```
#define NodeTypeDefs \
  NODETYPE(ELEMENT_NODE, 1) \
  NODETYPE(ATTRIBUTE_NODE, 2) \
  NODETYPE(TEXT_NODE, 3) \
  NODETYPE(CDATA_SECTION_NODE, 4) \
  NODETYPE(ENTITY_REFERENCE_NODE, 5) \
  NODETYPE(ENTITY_NODE, 6) \
```

```
  NODETYPE(PROCESSING_INSTRUCTION_NODE, 7) \
  NODETYPE(COMMENT_NODE, 8) \
  NODETYPE(DOCUMENT_NODE, 9) \
  NODETYPE(DOCUMENT_TYPE_NODE, 10) \
  NODETYPE(DOCUMENT_FRAGMENT_NODE, 11) \
  NODETYPE(NOTATION_NODE, 12)

enum NodeType {
#define NODETYPE(name, value) name = value,
NodeTypeDefs
#undef NODETYPE
};
```

⟨Insert namespace items into the `dom.Node` table4⟩≡

```
#define NODETYPE(name, value) \
  lua_pushstring(L, #name); \
  lua_pushnumber(L, (lua_Number) value); \
  lua_settable(L, -3);
NodeTypeDefs
#undef NODETYPE
```

This section is referenced on page 3.

This routine takes an `xmlNode` pointer and pushes it onto the Lua stack. If it is necessary to create a new userdata, it determines the appropritate metatable..

⟨C implementation definitions4⟩≡

```
static int
luaX_pushxmlNode(lua_State *L, xmlNode *node) {
  if(node == NULL) {
    lua_pushnil(L);
    return 1;
  } else {
    lua_pushlightuserdata(L, &unique_table);
    lua_gettable(L, LUA_REGISTRYINDEX);
    lua_pushlightuserdata(L, node);
    lua_rawget(L, -2);
    if(!lua_isnil(L, -1)) {
      lua_remove(L, -2);
#ifdef DEBUG
      fprintf(stderr, "Userdata for %p is old\n", node);
#endif
      return 1;
    } else {
      struct box *sb;
      char *mtname = NULL;
      switch(node->type) {
```

⟨Cases for `xmlNode` types in `luaX_pushxmlNode` 10⟩

```
      default:
        luaL_error(L, "can't push XML node type %d", node->type);
        return 0;
      }
      sb = lua_newuserdata(L, sizeof(struct box));
      sb->ptr = node;
      luaL_getmetatable(L, mtname);
      lua_setmetatable(L, -2);
```

⟨Add the new userdata to the `unique_table` 5⟩
```
lua_replace(L, -3);
lua_pop(L, 1);
```

⟨Increment the document's count of internal references 5⟩
```
#ifdef DEBUG
      fprintf(stderr, "Userdata for %p (type %s) is new\n", node, mtname);
#endif
      return 1;
    }
  }
}
```

This section continues on pages 5, 6, 7, 7, 8, 8, 10, 11, 11, 13, 14, 15, 16, 17, 17, 18, 18, 19, 20, 21 and 22.

This section is referenced on page 1.

We need to add the newly-created userdata to the `unique_table`. At this point the Lua stack contains the `unique_table`, a `nil`, and the new full userdata.

⟨Add the new userdata to the `unique_table`5⟩≡
```
lua_pushlightuserdata(L, node);
lua_pushvalue(L, -2);
lua_rawset(L, -5);
```

This section is referenced on page 5.

When the userdata is new then we can increment the reference count.

⟨Increment the document's count of internal references5⟩≡
```
{
  xmlDoc *doc = node->doc;
  struct document_private *private = doc->_private;
  ++(private->refcount);
#ifdef DEBUG
  fprintf(stderr, "document %p now has %u references\n",
          doc, private->refcount);
#endif
}
```

This section is referenced on page 5.

When a reference to a node is freed, we decrement the refernece count.

⟨Implementation declarations 2⟩+≡
```
static int dom_Node__gc(lua_State *L);
```

⟨C implementation definitions 4⟩+≡
```
static int
dom_Node__gc(lua_State *L) {
  struct box *sb = lua_touserdata(L, 1);
  xmlNode *node = sb->ptr;
  xmlDoc *doc = node->doc;
  struct document_private *private = doc->_private;
```

```
  private->refcount -= 1;
#ifdef DEBUG
  fprintf(stderr, "Userdata for %p is going away\n", node);
  fprintf(stderr, "Document %p now has %u internal nodes\n",
          doc, private->refcount);
#endif

   if(private->refcount == 0) {
#ifdef DEBUG
     fprintf(stderr, "Freeing document %p\n", doc);
#endif
     free(private);
     xmlFreeDoc(doc);
   }

  return 0;
}
```

The libxml library does not have any data structures corresponding to the DOM `NodeList` abstraction (since `xmlNode` objects contain internal linked list fields). Instead, we'll use a table with a custom metatable to implement the `NodeList` interface.

⟨Implementation declarations 2⟩+≡
```
static int luaX_newNodeList(lua_State *L, xmlNode *head);
static int dom_NodeList__index(lua_State *L);
static int dom_NodeList__newindex(lua_State *L);
```

⟨Insert namespace items into the `dom` table 3⟩+≡
```
lua_pushstring(L, "NodeList");
lua_newtable(L);
```
⟨Items in, and functions using, the `dom.NodeList` namespace 6⟩
```
lua_settable(L, -3);
```

⟨Items in, and functions using, the `dom.NodeList` namespace6⟩≡
```
luaL_newmetatable(L, "dom.NodeList");
lua_pushstring(L, "__index");
lua_pushvalue(L, -3);
lua_pushcclosure(L, dom_NodeList__index, 1);
lua_settable(L, -3);
lua_pushstring(L, "__newindex");
lua_pushcfunction(L, dom_NodeList__newindex);
lua_settable(L, -3);

lua_pop(L, 1);
```

This section continues on page 8.

This section is referenced on page 6.

The `luaX_newNodeList` function creates a `NodeList` given the node at the head of the list.

⟨C implementation definitions 4⟩+≡
```
static int
luaX_newNodeList(lua_State *L, xmlNode *head) {
```
⟨Create a new table and tag it as a `NodeList` 7⟩

```
    ⟨Fill in the "head" field 7⟩
    return 1;
  }
```

⟨Create a new table and tag it as a `NodeList`7⟩≡

```
  lua_newtable(L);
  luaL_getmetatable(L, "dom.NodeList");
  lua_setmetatable(L, -2);
```

This section is referenced on page 6.

If the list is non-empty, we set the value of the first index.

⟨Fill in the `"head"` field7⟩≡

```
  if(head != NULL) {
    luaX_pushxmlNode(L, head);
    lua_rawseti(L, -2, 1);
  }
```

This section is referenced on page 7.

The `newindex` method does nothing but say “no”.

⟨C implementation definitions 4⟩+≡

```
  static int
  dom_NodeList__newindex(lua_State *L) {
    lua_pushstring(L, "NodeList objects are read-only");
    lua_error(L);
    return 0;
  }
```

The `index` method, on the other hand, does a lot of things. First, it provides the usual Lua interface to array-like tables (with 1-based indexing and an `"n"` field. It also provides the DOM `item` method (which uses 0-based indexing) and a `length` attribute.

⟨C implementation definitions 4⟩+≡

```
  static int
  dom_NodeList__index(lua_State *L) {
    if(lua_isnumber(L, 2)) {
      int i = (int) lua_tonumber(L, 2);
      return luaX_getiNodeList(L, 1, i);
    } else if(lua_isstring(L, 2)) {
      const char *field = lua_tostring(L, 2);
      if(strcmp(field, "n") == 0) {
        lua_pop(L, luaX_getiNodeList(L, 1, INT_MAX));
        lua_pushstring(L, "length");
        lua_rawget(L, 1);
        return 1;
      } else if(strcmp(field, "length") == 0) {
        lua_pop(L, luaX_getiNodeList(L, 1, INT_MAX));
        lua_pushstring(L, "length");
        lua_rawget(L, 1);
        lua_pushnumber(L, lua_tonumber(L, -1));
        return 1;
      } else {
```

```
        lua_gettable(L, lua_upvalueindex(1));
        return 1;
      }
    } else {
      lua_gettable(L, lua_upvalueindex(1));
      return 1;
    }
  }
```

⟨Items in, and functions using, the `dom.NodeList` namespace 6⟩+≡

```
  lua_pushstring(L, "item");
  lua_pushcfunction(L, dom_NodeList_item);
  lua_settable(L, -3);
```

⟨C implementation definitions 4⟩+≡

```
  static int
  dom_NodeList_item(lua_State *L) {
    int i = (int) lua_tonumber(L, 2);
    return luaX_getiNodeList(L, 1, i + 1);
  }
```

Our indexing function works lazily, computing entries as necessary.

⟨Implementation declarations 2⟩+≡

```
  static int luaX_getiNodeList(lua_State *L, int index, int i);
```

⟨C implementation definitions 4⟩+≡

```
  static int
  luaX_getiNodeList(lua_State *L, int index, int i) {
    int n = lua_objlen(L, index);
    if(i <= 0) {
       lua_pushstring(L, "invalid index for NodeList item");
       lua_error(L);
       return 0;
    }
    if(i <= n) {
      lua_rawgeti(L, index, i);
      return 1;
    }
```

⟨Check to see if `"length"` is already defined 8⟩

⟨Traverse items starting with item `n` 9⟩

```
    return 1;
  }
```

If the `"length"` field has already been computed, it means we know how many items are in the nodelist, and we can rule out indices that aren't explicitly in the table already.

⟨Check to see if `"length"` is already defined8⟩≡

```
  lua_pushstring(L, "length");
  lua_rawget(L, index);
  if(lua_isnumber(L, -1)) {
    lua_pop(L, 1);
    return 0;
```

```
  }
  lua_pop(L, 1);
```

This section is referenced on page 8.

⟨Traverse items starting with item n9⟩≡

```
  {
    struct box *sb = (lua_rawgeti(L, index, n), lua_touserdata(L, -1));
    xmlNode *item = (sb != NULL) ? sb->ptr : NULL;
    lua_pop(L, 1);

    if(item == NULL)
      return 0;

    while(n < i) {
      item = item->next;
      if(item == NULL) {
        ⟨Set the "length" field to n 9⟩
        return 0;
      }
      ++n;
      luaX_pushxmlNode(L, item);
      lua_rawseti(L, index, n);
    }
    luaX_pushxmlNode(L, item);
  }
```

This section is referenced on page 8.

⟨Set the "length" field to n9⟩≡

```
  lua_pushstring(L, "length");
  lua_pushnumber(L, (double) n);
  lua_rawset(L, index);
```

This section is referenced on page 9.

The Document Node

Parsing a document yields a `Document` node, which is distinct from the root element. The `dom.Document` interface inherits from `dom.Node`.

⟨Register items that are subtypes of `dom.Node`9⟩≡

```
  lua_pushstring(L, "Document");
  lua_newtable(L);
```

⟨Set the metatable for `dom.Document` 10⟩

⟨Items in, and functions using, the `dom.Document` namespace 10⟩

```
  lua_settable(L, -5);
```

This section continues on pages 12, 14 and 15.

This section is referenced on page 3.

The `dom.Document` namespace table has a metatable that allows it to inherit from the `dom.Node` interface.

⟨Set the metatable for `dom.Document`10⟩≡
```
lua_newtable(L);
lua_pushstring(L, "__index");
lua_pushvalue(L, -5); /* dom.Node */
lua_settable(L, -3);
lua_setmetatable(L, -2);
```

This section is referenced on page 9.

The metatable for the boxed userdata itself contains methods for `gc`, `index`, and `newindex` methods.

⟨Items in, and functions using, the `dom.Document` namespace10⟩≡
```
luaL_newmetatable(L, "dom.Document");
lua_pushstring(L, "__gc");
lua_pushcfunction(L, dom_Node__gc);
lua_settable(L, -3);
lua_pushstring(L, "__index");
lua_pushvalue(L, -3);
lua_pushcclosure(L, dom_Document__index, 1);
lua_settable(L, -3);

lua_pop(L, 1);
```

This section is referenced on page 9.

⟨Cases for `xmlNode` types in `luaX_pushxmlNode`10⟩≡
```
case XML_DOCUMENT_NODE:
case XML_HTML_DOCUMENT_NODE:
  mtname = "dom.Document";
  break;
```

This section continues on pages 13, 15 and 16.

This section is referenced on page 4.

In addition to the `Node` constants, the `Document` interface allows several read-only values to be retrieved. Other values can be retrieved from the `dom.Document` namespace.

⟨C implementation definitions 4⟩+≡
```
static int
dom_Document__index(lua_State *L) {
  struct box *sb = luaL_checkudata(L, 1, "dom.Document");
  xmlDoc *document = sb->ptr;
  const char *field = lua_tostring(L, 2);
  if(strcmp(field, "nodeType") == 0) {
    lua_pushnumber(L, DOCUMENT_NODE);
    return 1;
  } else if(strcmp(field, "nodeName") == 0) {
    lua_pushstring(L, "#document");
    return 1;
  } else if(strcmp(field, "childNodes") == 0) {
    luaX_newNodeList(L, document->children);
```

```
    return 1;
  } else if(strcmp(field, "documentElement") == 0) {
    xmlNode *rootElement = xmlDocGetRootElement(document);
    luaX_pushxmlNode(L, rootElement);
    return 1;
  /* readonly attribute DocumentType doctype; */
  /* readonly attribute DOMImplementation implementation; */
  /* readonly attribute Element documentElement; */
  /* readonly attribute DOMString actualEncoding; */
  /* attribute DOMString xmlEncoding; */
  /* attribute boolean xmlStandalone; */
  /* attribute DOMString xmlVersion; */
  /* attribute boolean strictErrorChecking; */
  /* attribute DOMString documentURI; */
  /* readonly attribute DOMConfiguration config; */
  } else {
    lua_gettable(L, lua_upvalueindex(1));
  }
  return 1;
}
```

Right now, the only way to create a `Document` node is to parse an XML source document, either from a file or from memory.

⟨Implementation declarations 2⟩+≡

```
static int luaX_xmlParseFile(lua_State *L);

static int luaX_xmlParseMemory(lua_State *L);
```

⟨C implementation definitions 4⟩+≡

```
static int
luaX_xmlParseFile(lua_State *L) {
  const char *pathname = lua_tostring(L, 1);
  xmlDoc *document = xmlParseFile(pathname);
#ifdef DEBUG
  fprintf(stderr, "parsed document %p\n", document);
#endif
  if(document) {
    struct document_private *private
       = malloc(sizeof(struct document_private));
    private->refcount = 0;
    document->_private = private;
    luaX_pushxmlNode(L, (xmlNode *) document);
    return 1;
  } else {
    return 0;
  }
}
```

⟨C implementation definitions 4⟩+≡

```
static int
luaX_xmlParseMemory(lua_State *L) {
  const char *buffer = lua_tostring(L, 1);
  size_t len = lua_strlen(L, 1);
  xmlDoc *document = xmlParseMemory(buffer, len);
```

```
#ifdef DEBUG
  fprintf(stderr, "parsed document %p\n", document);
#endif
  if(document) {
    struct document_private *private
       = malloc(sizeof(struct document_private));
    private->refcount = 0;
    document->_private = private;
    luaX_pushxmlNode(L, (xmlNode *) document);
    return 1;
  } else {
    return 0;
  }
}
```

⟨Functions registered in the `libxml` table12⟩≡

```
{ "xmlParseFile", luaX_xmlParseFile },
{ "xmlParseMemory", luaX_xmlParseMemory },
```

This section continues on pages 18 and 19.

This section is referenced on page 1.

Element Nodes

⟨Register items that are subtypes of `dom.Node` 9⟩+≡

```
lua_pushstring(L, "Element");
lua_newtable(L);
```

⟨Set the metatable for `dom.Element` 12⟩

⟨Items in, and functions using, the `dom.Element` namespace 12⟩

```
lua_settable(L, -5);
```

⟨Set the metatable for `dom.Element`12⟩≡

```
lua_newtable(L);
lua_pushstring(L, "__index");
lua_pushvalue(L, -5); /* dom.Node */
lua_settable(L, -3);
lua_setmetatable(L, -2);
```

This section is referenced on page 12.

⟨Items in, and functions using, the `dom.Element` namespace12⟩≡

```
luaL_newmetatable(L, "dom.Element");
lua_pushstring(L, "__gc");
lua_pushcfunction(L, dom_Node__gc);
lua_settable(L, -3);
lua_pushstring(L, "__index");
lua_pushvalue(L, -3);
lua_pushcclosure(L, dom_Element__index, 1);
lua_settable(L, -3);

lua_pop(L, 1);
```

This section continues on page 13.

This section is referenced on page 12.

⟨Cases for `xmlNode` types in `luaX_pushxmlNode` 10⟩+≡

```
case XML_ELEMENT_NODE:
  mtname = "dom.Element";
  break;
```

⟨C implementation definitions 4⟩+≡

```
static int
dom_Element__index(lua_State *L) {
  struct box *sb = luaL_checkudata(L, 1, "dom.Element");
  xmlElement *element = sb->ptr;
  const char *field = lua_tostring(L, 2);
  if(strcmp(field, "nodeType") == 0) {
    lua_pushnumber(L, ELEMENT_NODE);
    return 1;
  } else if(strcmp(field, "nodeName") == 0
           || strcmp(field, "tagName") == 0) {
    lua_pushstring(L, element->name);
    return 1;
  } else if(strcmp(field, "parentNode") == 0) {
    luaX_pushxmlNode(L, (xmlNode *) element->parent);
    return 1;
  } else if(strcmp(field, "childNodes") == 0) {
    luaX_newNodeList(L, element->children);
    return 1;
  } else if(strcmp(field, "firstChild") == 0) {
    luaX_pushxmlNode(L, element->children);
    return 1;
  } else if(strcmp(field, "lastChild") == 0) {
    luaX_pushxmlNode(L, element->last);
    return 1;
  } else if(strcmp(field, "previousSibling") == 0) {
    luaX_pushxmlNode(L, element->prev);
    return 1;
  } else if(strcmp(field, "nextSibling") == 0) {
    luaX_pushxmlNode(L, element->next);
    return 1;
  } else if(strcmp(field, "ownerDocument") == 0) {
    luaX_pushxmlNode(L, (xmlNode *) element->doc);
    return 1;
  /* readonly attribute TypeInfo schemaTypeInfo; */
  } else {
    lua_gettable(L, lua_upvalueindex(1));
  }
  return 1;
}
```

⟨Implementation declarations 2⟩+≡

```
static int dom_Element_getAttribute(lua_State *L);
```

⟨Items in, and functions using, the `dom.Element` namespace 12⟩+≡

```
lua_pushstring(L, "getAttribute");
```

```
  lua_pushcfunction(L, dom_Element_getAttribute);
  lua_settable(L, -3);
```

⟨C implementation definitions 4⟩+≡

```
  static int
  dom_Element_getAttribute(lua_State *L) {
    struct box *sb = luaL_checkudata(L, 1, "dom.Element");
    const char *name = lua_tostring(L, 2);
    xmlNode *element = sb->ptr;
    char *value = xmlGetProp(element, name);
    if(value == NULL)
      lua_pushnil(L);
    else
      lua_pushstring(L, value);
    return 1;
  }
```

Text Nodes

FIXME Text and Comment interfaces should actually inherit from CharacterData.

⟨Register items that are subtypes of `dom.Node` 9⟩+≡

```
  lua_pushstring(L, "Text");
  lua_newtable(L);
```

⟨Set the metatable for `dom.Text` 14⟩

⟨Items in, and functions using, the `dom.Text` namespace 14⟩

```
  lua_settable(L, -5);
```

⟨Set the metatable for `dom.Text`14⟩≡

```
  lua_newtable(L);
  lua_pushstring(L, "__index");
  lua_pushvalue(L, -5); /* dom.Node */
  lua_settable(L, -3);
  lua_setmetatable(L, -2);
```

This section is referenced on page 14.

⟨Items in, and functions using, the `dom.Text` namespace14⟩≡

```
  luaL_newmetatable(L, "dom.Text");
  lua_pushstring(L, "__gc");
  lua_pushcfunction(L, dom_Node__gc);
  lua_settable(L, -3);
  lua_pushstring(L, "__index");
  lua_pushvalue(L, -3);
  lua_pushcclosure(L, dom_Text__index, 1);
  lua_settable(L, -3);

  lua_pop(L, 1);
```

This section is referenced on page 14.

⟨Cases for `xmlNode` types in `luaX_pushxmlNode` 10⟩+≡

```
case XML_TEXT_NODE:
  mtname = "dom.Text";
  break;
```

⟨C implementation definitions 4⟩+≡

```
static int
dom_Text__index(lua_State *L) {
  struct box *sb = luaL_checkudata(L, 1, "dom.Text");
  xmlNode *text = sb->ptr;
  const char *field = lua_tostring(L, 2);
  if(strcmp(field, "nodeType") == 0) {
    lua_pushnumber(L, TEXT_NODE);
    return 1;
  } else if(strcmp(field, "nodeName") == 0) {
    lua_pushstring(L, "#text");
    return 1;
  } else if(strcmp(field, "nodeValue") == 0) {
    lua_pushstring(L, xmlNodeGetContent(text));
    return 1;
  } else if(strcmp(field, "parentNode") == 0) {
    luaX_pushxmlNode(L, text->parent);
    return 1;
  } else if(strcmp(field, "previousSibling") == 0) {
    luaX_pushxmlNode(L, text->prev);
    return 1;
  } else if(strcmp(field, "nextSibling") == 0) {
    luaX_pushxmlNode(L, text->next);
    return 1;
  } else if(strcmp(field, "ownerDocument") == 0) {
    luaX_pushxmlNode(L, (xmlNode *)text->doc);
    return 1;
    /* readonly attribute DOMString wholeText; */
  } else {
    lua_gettable(L, lua_upvalueindex(1));
  }
  return 1;
}
```

Comment Nodes

⟨Register items that are subtypes of `dom.Node` 9⟩+≡

```
lua_pushstring(L, "Comment");
lua_newtable(L);
```

⟨Set the metatable for `dom.Comment` 15⟩

⟨Items in, and functions using, the `dom.Comment` namespace 16⟩

```
lua_settable(L, -5);
```

⟨Set the metatable for `dom.Comment`15⟩≡

```
lua_newtable(L);
lua_pushstring(L, "__index");
```

```
  lua_pushvalue(L, -5); /* dom.Node */
  lua_settable(L, -3);
  lua_setmetatable(L, -2);
```

This section is referenced on page 15.

⟨Items in, and functions using, the `dom.Comment` namespace16⟩≡

```
  luaL_newmetatable(L, "dom.Comment");
  lua_pushstring(L, "__gc");
  lua_pushcfunction(L, dom_Node__gc);
  lua_settable(L, -3);
  lua_pushstring(L, "__index");
  lua_pushvalue(L, -3);
  lua_pushcclosure(L, dom_Comment__index, 1);
  lua_settable(L, -3);

  lua_pop(L, 1);
```

This section is referenced on page 15.

⟨Cases for `xmlNode` types in `luaX_pushxmlNode` 10⟩+≡

```
  case XML_COMMENT_NODE:
    mtname = "dom.Comment";
    break;
```

⟨C implementation definitions 4⟩+≡

```
  static int
  dom_Comment__index(lua_State *L) {
    struct box *sb = luaL_checkudata(L, 1, "dom.Comment");
    xmlNode *text = sb->ptr;
    const char *field = lua_tostring(L, 2);
    if(strcmp(field, "nodeType") == 0) {
      lua_pushnumber(L, COMMENT_NODE);
      return 1;
    } else if(strcmp(field, "nodeName") == 0) {
      lua_pushstring(L, "#comment");
      return 1;
    } else if(strcmp(field, "nodeValue") == 0) {
      lua_pushstring(L, xmlNodeGetContent(text));
      return 1;
    } else if(strcmp(field, "parentNode") == 0) {
      luaX_pushxmlNode(L, text->parent);
      return 1;
    } else if(strcmp(field, "previousSibling") == 0) {
      luaX_pushxmlNode(L, text->prev);
      return 1;
    } else if(strcmp(field, "nextSibling") == 0) {
      luaX_pushxmlNode(L, text->next);
      return 1;
    } else if(strcmp(field, "ownerDocument") == 0) {
      luaX_pushxmlNode(L, (xmlNode *)text->doc);
      return 1;
    } else {
      lua_gettable(L, lua_upvalueindex(1));
    }
    return 1;
```

```
}
```

Output Support

In libxml, custom output support is implemented using the `xmlOutputBuffer` structure, containing two callbacks: one for writes, and one for the final close. We will implement this abstraction as a table with `"write"` and `"close"` elements.

⟨Other includes17⟩≡
```
#include <libxml/xmlIO.h>
```

This section continues on pages 18 and 19.

This section is referenced on page 1.

⟨Implementation declarations 2⟩+≡
```
static int luaX_xmlSaveFileTo(lua_State *L);
static int write_callback(void *context, const char * buffer, int len);
static int close_callback(void *context);
```

The `xmlSaveFileTo` function instantiates an output buffer and prepares the two callbacks.

⟨C implementation definitions 4⟩+≡
```
static int
luaX_xmlSaveFileTo(lua_State *L) {
  struct box *sb = luaL_checkudata(L, 2, "dom.Document");
  xmlDoc *doc = sb->ptr;
  xmlOutputBuffer *buf = xmlAllocOutputBuffer(NULL);

  if(doc == NULL)
    luaL_typerror(L, 1, "dom.Document");

  if(buf) {
    buf->context = L;
    buf->writecallback = write_callback;
    buf->closecallback = close_callback;
  }
  lua_pushnumber(L, xmlSaveFileTo(buf, doc, NULL));
  return 1;
}
```

The write callback gets the first argument to `xmlSaveFileTo` from the stack, retrieves the `"write"` element from it, and calls it with the first argument and the string to be written.

⟨C implementation definitions 4⟩+≡
```
static int
write_callback(void *context, const char *buffer, int len) {
  lua_State *L = context;
  int top = lua_gettop(L);

  lua_pushstring(L, "write");
```

```
    lua_gettable(L, 1);

    lua_pushvalue(L, 1);
    lua_pushlstring(L, buffer, len);
    lua_call(L, 2, 0);
    lua_settop(L, top);
    return len;
  }
```

The close callback is similar.

⟨C implementation definitions 4⟩+≡

```
  static int
  close_callback(void *context) {
    lua_State *L = context;
    int top = lua_gettop(L);

    lua_pushstring(L, "close");
    lua_gettable(L, 1);

    lua_pushvalue(L, 1);
    lua_call(L, 1, 0);
    lua_settop(L, top);
    return 0;
  }
```

⟨Functions registered in the `libxml` table 12⟩+≡

```
  { "xmlSaveFileTo", luaX_xmlSaveFileTo },
```

Catalog Support

Support for SGML and XML catalogs is available in libxml.

⟨Other includes 17⟩+≡

```
  #include <libxml/catalog.h>
```

⟨Implementation declarations 2⟩+≡

```
  static int luaX_xmlLoadCatalog(lua_State *L);
  static int luaX_xmlLoadCatalogs(lua_State *L);
```

⟨C implementation definitions 4⟩+≡

```
  static int
  luaX_xmlLoadCatalog(lua_State *L) {
    const char *pathname = lua_tostring(L, 1);
    if(xmlLoadCatalog(pathname) == 0) {
      lua_pushvalue(L, 1);
      return 1;
    } else {
      return 0;
    }
```

```
}

static int
luaX_xmlLoadCatalogs(lua_State *L) {
  const char *pathnames = lua_tostring(L, 1);
  xmlLoadCatalogs(pathnames);
  return 0;
}
```

⟨Functions registered in the `libxml` table 12⟩+≡
```
{ "xmlLoadCatalog", luaX_xmlLoadCatalog },
{ "xmlLoadCatalogs", luaX_xmlLoadCatalogs },
```

XSLT Support

We also support libxslt.

⟨Other includes 17⟩+≡
```
#include <libxslt/xslt.h>
#include <libxslt/transform.h>
#include <libxslt/xsltutils.h>
```

Parsing Stylesheets

We will tag pointers to parsed XSLT stylesheets with their own metatable.

⟨Implementation declarations 2⟩+≡
```
static int Stylesheet_gc(lua_State *L);
```

⟨Initialize libxml 2⟩+≡
```
luaL_newmetatable(L, "libxslt Stylesheet");
lua_pushstring(L, "__gc");
lua_pushcfunction(L, Stylesheet_gc);
lua_settable(L, -3);
```

Stylesheets are created by calling `xsltParseStylesheetFile`.

⟨Implementation declarations 2⟩+≡
```
static int luaX_xsltParseStylesheetFile(lua_State *L);
```

⟨C implementation definitions 4⟩+≡
```
static int
luaX_xsltParseStylesheetFile(lua_State *L) {
  const char *pathname = lua_tostring(L, 1);
  xsltStylesheet *stylesheet = xsltParseStylesheetFile(pathname);
  if(stylesheet) {
```

```
    struct box *sb = lua_newuserdata(L, sizeof(struct box));
    sb->ptr = stylesheet;
    luaL_getmetatable(L, "libxslt Stylesheet");
    lua_setmetatable(L, -2);
    return 1;
  } else {
    return 0;
  }
}
```

⟨Functions registered in the `libxsltl` table20⟩≡

```
{ "xsltParseStylesheetFile", luaX_xsltParseStylesheetFile },
```

This section continues on pages 21 and 22.

This section is referenced on page 1.

Applying Stylesheets

To run a spreadsheet, we use the `xsltApplyStylesheet` function. It takes a stylesheet, a document, and a table mapping parameter names to parameter values, and returns a Document node.

⟨Implementation declarations 2⟩+≡

```
static int luaX_xsltApplyStylesheet(lua_State *L);
```

The wrapping of `xsltApplyStylesheet` is straightforward, except that the parameters and parameter values need to be collected into a vector.

⟨C implementation definitions 4⟩+≡

```
static int
luaX_xsltApplyStylesheet(lua_State *L) {
  struct box *ssb = luaL_checkudata(L, 1, "libxslt Stylesheet");
  xsltStylesheet *stylesheet = ssb->ptr;
  struct box *dsb = luaL_checkudata(L, 2, "dom.Document");
  xmlDoc *document = dsb->ptr;
  xmlDoc *result = NULL;

  const char **paramvec = NULL;
  unsigned nparams = 0;

  ⟨Count the number of supplied parameters 21⟩
  paramvec = malloc((nparams * 2 + 1) * sizeof(const char *));
  ⟨Fill in paramvec 21⟩

  result = xsltApplyStylesheet(stylesheet, document, paramvec);
  free(paramvec);

  if(result) {
    struct document_private *private = malloc(sizeof(struct document_private));
    private->refcount = 0;
    result->_private = private;
    luaX_pushxmlNode(L, (xmlNode *) result);
    return 1;
```

```
    } else {
      return 0;
    }
  }
```

Before allocating a vector for the parameters to pass to `xsltApplyStylesheet`, we need to count how many elements there are.

⟨Count the number of supplied parameters21⟩≡
```
  lua_pushnil(L);
  while(lua_next(L, 3) != 0) {
    ++nparams;
    lua_pop(L, 1);
  }
```

This section is referenced on page 20.

Once `paramvec` is allocated we can store parameter names and parameter values (which are XPath expressions) into it.

⟨Fill in `paramvec`21⟩≡
```
  {
    const char **p = paramvec;
    lua_pushnil(L);
    while(lua_next(L, 3) != 0) {
      *p++ = lua_tostring(L, -2);
      *p++ = lua_tostring(L, -1);
      lua_pop(L, 1);
    }
    *p++ = NULL;
  }
```

This section is referenced on page 20.

⟨Functions registered in the `libxsltl` table 20⟩+≡
```
  { "xsltApplyStylesheet", luaX_xsltApplyStylesheet },
```

Saving Results

The `xsltSaveResultTo` function is handled in a similar manner to `xmlSaveFileTo` above.

⟨Implementation declarations 2⟩+≡
```
  static int luaX_xsltSaveResultTo(lua_State *L);
```

⟨C implementation definitions 4⟩+≡
```
  static int
  luaX_xsltSaveResultTo(lua_State *L) {
    struct box *rsb = luaL_checkudata(L, 2, "dom.Document");
    xmlDoc *result = rsb->ptr;
    struct box *ssb = luaL_checkudata(L, 3, "libxslt Stylesheet");
    xsltStylesheet *stylesheet = ssb->ptr;
```

```
    xmlOutputBuffer *buf = xmlAllocOutputBuffer(NULL);

    if(buf) {
      buf->context = L;
      buf->writecallback = write_callback;
      buf->closecallback = close_callback;
    }
    xsltSaveResultTo(buf, result, stylesheet);
    lua_pushnumber(L, xmlOutputBufferClose(buf));
    return 1;
  }
```

⟨Functions registered in the `libxsltl` table 20⟩+≡

```
  { "xsltSaveResultTo", luaX_xsltSaveResultTo },
```

Stylesheet Garbage Collection

We can free the stylesheet when the last userdata reference to it is gone.

⟨C implementation definitions 4⟩+≡

```
  static int
  Stylesheet_gc(lua_State *L) {
    struct box *sb = lua_touserdata(L, 1);
    xsltStylesheet *stylesheet = sb->ptr;

  #ifdef DEBUG
    fprintf(stderr, "Freeing stylesheet %p\n", stylesheet);
  #endif

    xsltFreeStylesheet(stylesheet);
    return 0;
  }
```