litlxml: A source extractor for lightweight literate programming

John W. Shipman

2011-04-12 19:15

Abstract

Describes a script that extracts the source code from a program presented in lightweight literate programming form, using the DocBook documentation toolchain, the Python programming language, and the `lxml` module for XML processing.

This publication is available in Web form[1] and also as a PDF document[2]. Please forward any comments to **`tcc-doc@nmt.edu`**.

Table of Contents

1. Introduction

Programs must be written for people to read, and only incidentally for machines to execute.

— *Structure and interpretation of computer programs*, Harold Abelson and Gerald Jay Sussman, p. xvii

[1] http://www.nmt.edu/tcc/help/lang/python/examples/litlxml/
[2] http://www.nmt.edu/tcc/help/lang/python/examples/litlxml/litlxml.pdf

By literate programming, we mean programs that are intended to be readable. The idea comes from Dr. Donald E. Knuth and has a long history. For background, see the Literate Programming web site[3].

Knuth's `cweb` system interwove narrative about the program with the actual source code of the program. One then runs a tool named `ctangle` to generate the source code an a different tool named `cweave` to generate the online documentation.

The present effort was inspired by similar efforts of Dr. Allan M. Stavely[4], who suggested using DocBook as a general framework for literate programming. Refer to *Writing documentation with DocBook-XML 4.3*[5] for more information on DocBook.

Stavely's idea was to use DocBook's existing `programlisting` element to hold the program fragments, adding a `role='executable'` attribute to that element to distinguish executable source code from other uses of the `programlisting` element. This means that the regular processing of DocBook into HTML and PDF forms is the equivalent of Knuth's `cweave` step.

The remaining half of the problem, the extraction of the executable code from the DocBook source file, is the subject of this document.

The *litlxml* script is embedded in this document. Relevant online files include:

- The *litlxml* script itself.[6]
- The DocBook source for this document.[7]

See also the author's literate programming site[8] for more discussion of the practice and several dozen examples in assorted languages and sizes.

2. Encoding the literate program

One limitation of Stavely's approach was that it assembled all the executable code fragments into a single file for execution. But the literate exposition of a C program, for example, might require the discussion of two source files, a header file named `foo.h` and a code file named `foo.c`. We get around this problem by using the `role` attribute of the `programlisting` element in a more flexible way.

The general form of a literate program source is a valid DocBook-XML file, except that each fragment of executable code is wrapped in a `programlisting` element with this general format:

```
<programlisting role='outFile:F'>
  (source text)
</programlisting>
```

where *F* is the name of the output file to which that source text should be written.

We can then handle the above example by using a `role='outFile:foo.h'` attribute on fragments of the header file and a `role='outFile:foo.c'` attribute on fragments of the code file. For example:

```
<programlisting role='outFile:foo.h'>
  (stuff to be written to foo.h)
</programlisting>
   ...
<programlisting role='outFile:foo.c'>
```

[3] http://www.literateprogramming.com/
[4] http://www.nmt.edu/~al/
[5] http://www.nmt.edu/tcc/help/pubs/docbook43/
[6] http://www.nmt.edu/tcc/help/lang/python/examples/litlxml/litlxml
[7] http://www.nmt.edu/tcc/help/lang/python/examples/litlxml/litlxml.xml
[8] http://www.nmt.edu/~shipman/soft/litprog/

```
  (stuff to be written to foo.c)
</programlisting>
```

Of course, either of those files can be broken into many fragments spread throughout the document. They can even be intermingled.

There are two important refinements to mention:

- You can use a CDATA section to enclose the source fragment. This XML convention uses special delimiters to tell processing programs not to mess with anything between "`<![CDATA[`" and "`]]>`". This is especially convenient for enclosing XML fragments, because you can use "`<`" and "`>`" characters without having to escape them.
- If your text is not enclosed in a CDATA section, you can use DocBook tags inside the `programlisting` element.

  For example, you can enclose a function call inside a `link` element that links to the definition of that function. In both the HTML and PDF generated from the DocBook file, that function name will then be clickable.

  Another element you might want to use inside a code fragment is the `co` element, to label lines of the code with callouts that are defined later inside DocBook `callout` elements.

Here's an example of the use of callouts, as it would be encoded in the DocBook source. This is from the exposition of a schema using Relax NG Compact Format (RNC)[9].

```
      <programlisting role='outFile:trails.rnc'>
park = element park
{ attribute name { text }?,   <co id='park.name'>
  trail*                      <co id='park.trail'>
}
</programlisting>
      <calloutlist>
        <callout arearefs='park.name'>
          <para>
            This optional attribute contains the name of the park.
          </para>
        </callout>
        <callout arearefs='park.trail'>
          <para>
            The content of a <code>park</code> element
            consists of one or more <code>trail</code>
            elements.
          </para>
        </callout>
      </calloutlist>
```

3. Operation of the *litlxml* script

A script in the Python language extracts the various output files from DocBook source files. Command line arguments are:

[9] http://www.nmt.edu/tcc/help/pubs/rnc/

```
litlxml file ...
```

Each DocBook-XML source file named on the command line is read, and all the `programlisting` elements with the correct `role` attribute are assembled and written to the corresponding files.

3.1. Suggested `Makefile` rules

If you are using the Unix *make* utility to build your document and source files, you can add lines to your `Makefile` to take care of building the program source files.

The exact rules depend on whether your literate programs are executable or not. We'll assume that both executable and non-executable programs are produced, and that the variable `BASENAME` is the name of your DocBook file minus its "`.xml`" extension. below.

First, define three variables like this:

```
MODULES        =  m1 m2 ...
EXECUTABLES    =  e1 e2 ...
CODE_TARGETS   =  $(EXECUTABLES) $(MODULES)
```

where *`m1`*, *`m2`*, and so forth are the names of non-executable files, and *`e1`*, *`e2`*, and so on are the names of your executable files.

Then, in the rules part of your `Makefile`, change the first (default) target to read like this:

```
all: web pdf code
```

Add these rules:

```
code: $(CODE_TARGETS)

$(CODE_TARGETS): $(BASENAME).xml
        litlxml $<; \
        chmod +x $(EXECUTABLES)
```

If no executables are produced, change the latter rule to:

```
$(CODE_TARGETS): $(BASENAME).xml
        litlxml $<
```

A model `Makefile` is included in *Writing documentation with DocBook-XML 4.3*[10].

4. Literate exposition of the *litlxml* program itself

The *litlxml* program is worth study as an example not only of literate programming but also of how easy it is to process XML files in Python.

4.1. Design notes

This program has gone through several rewrites as techniques for XML processing in Python have evolved.

[10] http://www.nmt.edu/tcc/help/pubs/docbook43/make-lit.html

The current version uses the `lxml` package. For more details, see the *Python processing with `lxml`*[11]. This package yields much higher performance than earlier approaches.

This program was written using the Cleanroom or zero-defect methodology. The best introduction to the method is given in Stavely, Allan M., *Toward Zero-defect Programming*, Addison-Wesley, 1999, ISBN 0-201-38595-3. Also see the author's Cleanroom pages[12] for a discussion of methods and dozens of examples

4.2. The prologue

The script starts with the usual Python prologue. The first line makes the script self-executing. This is followed by minimal comments pointing to the online form of the literate programming document, and the Cleanroom intended function for the program as a whole.

`litlxml`

```
#!/usr/bin/env python
#================================================================
# litlxml:  Extract code from literate-programming source files.
#   Do not edit this file.  It is extracted automatically from
#   the documentation:
#       http://www.nmt.edu/tcc/help/lang/python/examples/litlxml/
#----------------------------------------------------------------
# Overall intended function:
#   [ output files named in input files given on the command line
#         :=  code fragments designated for those files
#     sys.stderr  +:=  error messages if any ]
#----------------------------------------------------------------
```

4.3. Modules required

Aside from the standard Python `sys` module that gives programs access to their standard I/O streams and command line arguments, the program needs the `etree` library from the `lxml` module.

`litlxml`

```
import sys
from lxml import etree
```

4.4. Global declarations

These manifest constants are defined globally.

`PROG_ELT`
The element for the `programlisting` element.

`litlxml`

```
#================================================================
# Manifest constants
#----------------------------------------------------------------

PROG_ELT     =  "programlisting"
```

[11] http://www.nmt.edu/tcc/help/pubs/pylxml/
[12] http://www.nmt.edu/~shipman/soft/clean/

ROLE_ATTR

The name of the `role` attribute.

litlxml

```
ROLE_ATTR    =  "role"
```

ROLE_PREFIX

The prefix of the `role` attribute that identifies this `programlisting` element as a code fragment.

litlxml

```
ROLE_PREFIX  =  "outFile:"
```

4.5. Verification functions

In the Cleanroom methodology, a verification function is a shorthand notation for describing various program entities. The author's preference is to use names for these functions that contain a hyphen ("`-`"), so that it is clear that these are not Python functions.

Our first verification function is `lit-elt`: an XML element that contains literate code.

litlxml

```
#================================================================
# Verification functions
#----------------------------------------------------------------
# lit-elt  ==  an XML element whose GID is PROG_ELT, and which
#    has an attribute ROLE_ATTR whose value starts with
#    ROLE_PREFIX
```

Next is the `lit-dest` function. This describes the destination file to which a literate fragment is to be written.

litlxml

```
#----------------------------------------------------------------
# lit-dest(elt)  ==  the part of the ROLE_ATTR value after
#    ROLE_PREFIX in a lit-elt
```

The `lit-content` verification function describes the text inside the literate fragment. Note that literate code can contain XML tags: in some of the author's source code, the DocBook `link` tag is used so that the name of a called function or method is a link to the definition of that function or method. However, the source text should not include any tags.

litlxml

```
#----------------------------------------------------------------
# lit-content(elt)  ==  The text content of element (elt) and
#    any descendants
#----------------------------------------------------------------
```

4.6. The main program

The only thing the main does is iterate over the list of files given as command line arguments, processing each one in turn by calling Section 4.7, " `processFile()`: Process one input file " (p. 7).

litlxml

```
# - - - - -   m a i n

def main():
    """Main program for litlxml."""
    #-- 1 --
    for inFileName in sys.argv[1:]:
        #-- 1 body --
        # [ if inFileName names a readable, valid DocBook XML file ->
        #     output files named in that file  :=  code fragments
        #       designated for those files
        #     sys.stderr  +:=  error messages from processing that file,
        #                      if any
        #   else ->
        #     sys.stderr  +:=  error message ]
        processFile ( inFileName )
```

4.7. `processFile()`: Process one input file

This function handles all the processing for one DocBook source file.

litlxml

```
# - - -   p r o c e s s F i l e

def processFile ( fileName ):
    """Process one input file.

      [ inFileName is a string ->
          if inFileName names a readable, valid DocBook XML file ->
            output files named in lit-elts from that file  :=
              lit-content of those lit-elts
            sys.stderr  +:=  error messages from processing that file,
                             if any
          else ->
            sys.stderr  +:=  error message ]
    """
```

The fragments in a given file may be directed to several different output files. To keep track of the output files we have seen so far, we'll use a dictionary named `fileMap`, whose keys are file names, and each corresponding value is an open, writeable file handle for that file. We'll write the text to each file as it is encountered, and leave all the files open until the end, at which point we'll close them all.

litlxml

```
    #-- 1 --
    fileMap  =  {}
```

Next we call the `etree` package to parse the XML file and make it into an element tree. This may raise either of two exceptions:

- If the file can't even be opened, it will raise an `IOError` exception.
- If the file is not well-formed XML, the `etree` package will raise its `XMLSyntaxError` exception.

litlxml

```
    #-- 2 --
    # [ if fileName names a readable, valid XML file ->
    #     doc  :=  an ElementTree representing that file
    #   else ->
    #     sys.stderr  +:=  error message(s)
    #     return ]
    try:
        doc  =  etree.parse ( fileName )
    except IOError, detail:
        print >>sys.stderr, ( "*** I/O error opening '%s': %s" %
                              (fileName, detail) )
        return
    except etree.XMLSyntaxError, detail:
        print >>sys.stderr, ( "*** Syntax error opening '%s': %s" %
                              (fileName, detail) )
        return
```

Now that `doc` contains the document tree, send it off for processing to Section 4.8, "`processDoc()`: Process one document tree" (p. 8).

litlxml

```
    #-- 3 --
    # [ (doc is an etree Document) and
    #   (fileMap is a dictionary whose keys are file names and
    #   each corresponding value is a writeable file handle
    #   for that file) ->
    #     fileMap  :=  fileMap with new file names added from
    #         lit-dests in doc
    #     file handles in fileMap  :=  lit-content of those files
    #     sys.stderr  +:=  error messages from processing doc,
    #         if any ]
    processDoc ( fileMap, doc )
```

Finally, we close all the output files that are values in the `fileMap` dictionary.

litlxml

```
    #-- 4 --
    # [ fileMap is a dictionary whose values are file objects ->
    #     those values  :=  those values, closed ]
    for  outFile in fileMap.values():
        outFile.close()
```

4.8. `processDoc()`: Process one document tree

Given a document tree, this function finds all the literate program elements, attempts to open output files for them if they are not already open, and writes the code fragments to those files.

litlxml

```
# - - -   p r o c e s s   D o c

def processDoc ( fileMap, doc ):
    """Process one document tree

      [ (fileMap is a dictionary whose keys are file names and
        each corresponding value is a writeable file handle
        for that file) and
        (doc is an etree.ElementTree) ->
          fileMap  :=  fileMap with new output files added from
              lit-elts in doc whose lit-dests can be opened anew
          files named in fileMap  :=  lit-content of those files
          sys.stderr  +:=  error messages for lit-dests that
              cannot be opened for output, if any ]
    """
```

To find the root element of `doc`, we use its `.getroot()` method. Then we use an XPath expression to find all the `PROG_ELT` elements. The XPath expression `"//programlisting"` means to find all `programlisting` elements no matter where they are in the tree; it returns a list of matching elements.

litlxml

```
    #-- 1 --
    # [ eltList  :=  a list of all the PROG_ELT elements in doc,
    #                in document order ]
    root  =  doc.getroot()
    eltList  =  root.xpath ( "//" + PROG_ELT )
```

For each potentially literate element, we look to see if it has a `ROLE_ATTR` attribute, and if so, whether that attribute's value starts with `ROLE_PREFIX`. If so, it is a literate element, and is sent for processing to Section 4.9, "`processElt()`: Process one literate element" (p. 10).

litlxml

```
    #-- 2 --
    # [ fileMap  :=  fileMap with new file names added from lit-dests
    #                in eltList whose lit-dest files could be opened
    #   files named in fileMap  :=  lit-content of those files
    #   sys.stderr  +:=  error messages from failures to open
    #      those files, if any ]
    for  elt in eltList:
        #-- 2 body --
        # [ if  (elt is a lit-elt) and
        #   (lit-dest(elt) is a key in fileMap) ->
        #     that value from fileMap  +:=  lit-content(elt)
        #   else if (elt is a lit-elt) and
        #   (lit-dest(elt) is not a key in fileMap) and
        #   (a new file named lit-dest(elt) can be opened for
        #   writing) ->
        #     fileMap[lit-dest(elt)]  :=  that new file
        #     that new file  +:=  lit-content(elt)
        #   else if (elt is a lit-elt) and
        #   (lit-dest(elt) is not a key in fileMap) and
        #   (a new file named lit-dest(elt) cannot be opened for
        #   writing) ->
        #     sys.stderr  +:=  error message
        #   else -> I ]
```

Several conditions must be met for a literate element. There must be a `ROLE_ATTR` attribute; if not, trying to extract the element's `.attrib` dictionary's value will raise `KeyError`. If there is such an attribute, it must start with `ROLE_PREFIX`; if it does, the output file name is the rest of the attribute after that prefix.

litlxml

```
        try:
            attrValue  =  elt.attrib[ROLE_ATTR]
            if  attrValue.startswith ( ROLE_PREFIX ):
                outName  =  attrValue[len(ROLE_PREFIX):]
                processElt ( fileMap, outName, elt )
        except KeyError:
            pass
```

4.9. `processElt()`: Process one literate element

This function takes three arguments:

1. The `fileMap` is the dictionary whose keys are the names of output files we've already seen, and each corresponding value is a writeable file handle for that file.
2. The `outName` is the name of the output file.
3. The `elt` is the actual literate element, as an `etree.Element` instance.

litlxml

```
# - - -   p r o c e s s E l t

def processElt ( fileMap, outName, elt ):
    """Process one element that may be literate.

      [ (fileMap is a dictionary whose keys are file names and
        each corresponding value is a writeable file handle
        for that file) and
        (outName is a file name as a string) and
        (elt is an etree.Element) ->
          if fileMap has a key (outName) ->
            fileMap[outName]  +:=  text of elt
          else if outName can be opened new for writing ->
            fileMap[outName]  :=  that file, so opened
            that file  :=  text of elt
          else ->
            sys.stderr  +:=  error message(s) ]
    """
```

First we check to see if this is a new output file. If so, we try to open it for writing. This can fail, in which case we'll need to send an error message to the standard error stream, and return prematurely.

litlxml

```
    #-- 1 --
    # [ if outName is a key of fileMap ->
    #     I
    #   else if outName can be opened new for writing ->
    #     fileMap[outName]  :=  that file, so opened
    #   else ->
    #     sys.stderr  +:=  error message(s)
    #     return ]
    if  not fileMap.has_key(outName):
        try:
            fileMap[outName]  =  open ( outName, "w" )
        except IOError, detail:
            print >>sys.stderr, ( "*** Can't open '%s': %s" %
                                  (outName, detail) )
            return
```

At this point we have a destination file handle, `fileMap[outName]`. We use another XPath expression to find all the text descendants of `elt`. In this expression, the "`descendant-or-self::`" part is an *axis specifier* that selects `elt`, its children, its children's children, and so forth all the way to the leaves of the document tree. The XPath "`text()`" function selects only text nodes (as opposed to element nodes).

litlxml

```
    #-- 2 --
    # [ textList  :=  a list of all text descendants of elt ]
    textList  =  elt.xpath ( "descendant-or-self::text()" )

    #-- 3 --
    # [ fileMap[outName]  +:=  elements of textList, concatenated ]
    fileMap[outName].write ( "".join ( textList ) )
```

4.10. Epilogue

Rather than placing the main at the end of the script, we defined it above (Section 4.6, "The main program" (p. 6)) as a function `main()` so that the code can be presented in top-down order.

The lines below cause `main()` to be called, assuming that *litlxml* is the main script. Python sets global variable `__name__` to the string `'__main__'` for the outermost script.

litlxml

```
# - - - - -   e p i l o g u e   - - - - -

if  __name__ == '__main__':
    main()
```