

# A source extractor for lightweight literate programming



John W. Shipman

2006-06-11 14:01

## Table of Contents

1. Introduction .....	1
1.1. How to get this publication .....	2
2. Encoding the literate program .....	2
3. Operation of the <b>litsource</b> script .....	3
3.1. Suggested <b>Makefile</b> rules .....	3
4. Literate exposition of the <b>litsource</b> program itself .....	4
4.1. Design notes .....	4
4.2. The prologue .....	5
4.3. Modules required .....	5
4.4. Global declarations .....	6
4.5. The main program .....	6
4.6. <b>processFile</b> : Process one input file .....	6
4.7. <b>class ArticleHandler</b> : The customized content handler .....	8
4.8. <b>ArticleHandler.__init__()</b> : Constructor .....	8
4.9. <b>ArticleHandler.startElement()</b> : Observe a start tag .....	9
4.10. <b>ArticleHandler.characters()</b> : Observe text content .....	10
4.11. <b>ArticleHandler.endElement()</b> : Observe an end tag .....	11
4.12. Epilogue .....	11

## 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

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<sup>1</sup>.

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, and a different tool named **cweave** to generate the online documentation.

---

<sup>1</sup> <http://www.literateprogramming.com/>

The present effort was inspired by similar efforts of Dr. Allan M. Stavely<sup>2</sup>, who suggested using DocBook as a general framework for literate programming. Refer to *Writing documentation with DocBook-XML 4.2*<sup>3</sup> 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 becomes the new 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.

## 1.1. How to get this publication

This document is available in Web form<sup>4</sup> and also as a PDF document<sup>5</sup>. See also the executable Python source<sup>6</sup> and the XML source of this document<sup>7</sup>.

## 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'>
  (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 “]]>”.

---

<sup>2</sup> <http://www.nmt.edu/~al/>

<sup>3</sup> <http://www.nmt.edu/tcc/help/pubs/docbook42/>

<sup>4</sup> <http://www.nmt.edu/tcc/help/lang/python/examples/litsource/>

<sup>5</sup> <http://www.nmt.edu/tcc/help/lang/python/examples/litsource/litsource.pdf>

<sup>6</sup> <http://www.nmt.edu/tcc/help/lang/python/examples/litsource/litsource>

<sup>7</sup> <http://www.nmt.edu/tcc/help/lang/python/examples/litsource/litsource.xml>

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)<sup>8</sup>.

```
<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 <userinput>park</userinput> element
      consists of one or more <userinput>trail</userinput>
      elements.
    </para>
  </callout>
</calloutlist>
```

## 3. Operation of the `litsource` script

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

```
litsource 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.

<sup>8</sup> <http://www.nmt.edu/tcc/help/pubs/rnc/>

First, in the part of your `Makefile` that defines variables, define a variable named `CODE_TARGET` that contains the name of the source file you want to build. If you are building multiple source files, any of them will work. For instance, if your source file is called `run.c`, the rule would look like this:

```
CODE_TARGET    =    run.c
```

Then, in the rules part of your `Makefile`, add this rule:

```
code: $(CODE_TARGET)
$(CODE_TARGET): $(TARGET).xml
    litsource $<
```

Make sure that the last line starts with an actual *tab* character.

Here's one more refinement. Suppose you are generating an executable script in some scripting language like Perl or Python, and you need to make that script executable under Unix. Assume further that the script's name is in the `$(CODE_TARGET)` variable. You can automate making the script executable with a rule like this:

```
$(CODE_TARGET): $(TARGET).xml
    litsource $<; \
    chmod +x $(CODE_TARGET)
```

## 4. Literate exposition of the `litsource` program itself

The `litsource` 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

An earlier version of this script used the Document Object Model (DOM) to build a tree representation of the entire DocBook document. It then used XPath to pull from this tree the set of **programlisting** elements that had a **role** attribute whose value started with **"outFile:"**. The code was straightforward and quite short. See the literate exposition of the DOM version<sup>9</sup>.

However, for large Docbook files, the DOM technique became somewhat time-consuming. For example, a 5400-line DocBook file took about 60 seconds to process. The current version, using Python's SAX interface (Simple API for XML), processed this same file in 0.11 seconds, a better than 500-fold performance improvement.

SAX is a completely different approach to XML processing. It is a serial, event-based technique. The SAX interface reads through the XML and classifies each bit as a start tag, end tag, chunk of text, comment, and so on. The programmer defines a set of "handlers" that are called whenever specific types of content are encountered.

Because the `litsource` script cares only about the text inside selected **programlisting** elements, there is no need to build a tree of the entire DocBook file. All we need is a SAX interface with three handlers:

1. One handler observes each start tag that goes by. When it sees a **<programlisting role='out-File: filename'>** tag, it remembers that we are now inside a code fragment, and it also remembers the *filename*, and opens an output file by that name.

<sup>9</sup> <http://www.nmt.edu/tcc/help/lang/python/examples/litdom>

2. Another handler is called whenever the SAX interface sees text content. If we are currently inside a code fragment, that text content is written to the current output file.
3. A third handler observes each end tag. If it is a `</programlisting` tag, we note that we're no longer inside a code fragment.

Here are some good resources for learning about Python's XML libraries:

- See the Python web site<sup>10</sup> for information and downloads for the Python language.
- *Python & XML*, by Christopher A. Jones and Fred L. Drake, Jr. (O'Reilly Press, 2002, ISBN 0-596-00128-2) is an excellent overview of the major approaches to Python XML processing, with copious examples.
- In the online Python Library Reference, see the documentation for the `xml.sax` module<sup>11</sup>.

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 my Cleanroom pages<sup>12</sup> for a discussion of how I practice the methodology.

## 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.

litsource

```
#!/usr/bin/env python
#=====
# litsource: Extract code from literate-programming source files.
#   For documentation, see:
#       http://www.nmt.edu/tcc/help/lang/python/examples/litsource/
#-----
# 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 two items from the Python's SAX library:

- The `ContentHandler` class is a base class used to write SAX content handlers. See the documentation for `xml.sax.handler`<sup>13</sup>.
- The `make_parser()` function is used to create a parser using our content handler. See the documentation for `xml.sax`<sup>14</sup>.

litsource

```
import sys
from xml.sax.handler import ContentHandler
from xml.sax import make_parser
```

<sup>10</sup> <http://www.python.org>

<sup>11</sup> <http://www.python.org/doc/2.4/lib/module-xml.sax.html>

<sup>12</sup> <http://www.nmt.edu/~shipman/soft/clean>

<sup>13</sup> <http://www.python.org/doc/2.4/lib/module-xml.sax.handler.html>

<sup>14</sup> <http://www.python.org/doc/2.4/lib/module-xml.sax.html>

## 4.4. Global declarations

These manifest constants are defined globally.

### **PROG\_ELT**

The element for the **programlisting** element.

litsource

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

PROG_ELT      = "programlisting"
```

### **ROLE\_ATTR**

The name of the **role** attribute.

litsource

```
ROLE_ATTR     = "role"
```

### **ROLE\_PREFIX**

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

litsource

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

## 4.5. 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 4.6, “**processFile**: Process one input file” (p. 6).

litsource

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

def main():
    """Main program for litsource."""

    #-- 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 )
```

## 4.6. processFile: Process one input file

The **processFile()** function handles all the processing for one DocBook source file.

litsource

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

def processFile ( inFileName ):
    """Process one input file.
```

```

[ inFileName is a string ->
  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 ]
"""

```

The first step is to open the input file, and report errors if that fails.

litsource

```

#-- 1 --
# [ if inFileName names a readable file ->
#   inFile := that file opened for reading
#   else ->
#     sys.stderr += error message
#   return ]
try:
    inFile = open ( inFileName )
except IOError, detail:
    sys.stderr.write ( "*** Can't open file '%s' for reading: %s\n" %
                      (inFileName, detail) )
return

```

The next step is to create a SAX parser. We first create a content handler object, an **ArticleHandler** object. This object contains the three handlers that observe start tags, text content, and end tags. See Section 4.7, “**class ArticleHandler**: The customized content handler” (p. 8).

litsource

```

#-- 2 --
# [ ch := an ArticleHandler instance ]
ch = ArticleHandler()

```

The remaining steps obey the usual SAX protocol. We create a new SAX parser with the **make\_parser()** function, associate our content handler with it using its **.setContentHandler()** method, and then use its **.parse()** method to read **inFile**.

This process is fairly well-described on page 53 of the O'Reilly *Python and XML* book. (However, the example on this page does not run unless you first import the **ContentHandler** class.)

litsource

```

#-- 3 --
# [ ch is an ArticleHandler object ->
#   if inFile contains a readable, well-formed XML file ->
#     output files named in inFile := code fragments
#     designated for those files
#     sys.stderr += error message(s), if any ]
saxparser = make_parser()
saxparser.setContentHandler ( ch )
saxparser.parse ( inFile )

```

## 4.7. class ArticleHandler: The customized content handler

This class represents our content handler. It inherits from the SAX **ContentHandler** class; all we have to do is define three handlers, with given names.

These state items in the instance manage the process of extracting code fragments:

### .outFileName

Initially set to **None**, whenever we are inside a code fragment, this attribute holds the name of the output file.

### .outFile

When we are inside a code fragment, this attribute holds a writeable file handle that writes to **self.outFileName**.

### .fileMap

Because we want each output file to be the concatenation of all the code fragments assigned to that file, we want to open each output file only once. Hence, the **.fileMap** attribute holds a dictionary whose keys are the names of output files we have seen so far, and each corresponding value is a writeable file handle for that file.

litsource

```
# - - - - c l a s s   A r t i c l e H a n d l e r   - - - - -
class ArticleHandler(ContentHandler):
    """Content handler object.

    Exports:
        ArticleHandler(): [ return a new ArticleHandler ]

    State/Invariants:
        .fileMap:
            [ a dictionary whose keys are the names of files in
              fragments seen so far; each value is a writeable
              file handle for that file ]
        .outFileName:
            [ if currently within a fragment ->
              the output file name for that fragment
              else -> None ]
        .outFile:
            [ if currently within a fragment ->
              the output file handle for that fragment
              else -> None ]
    """
```

## 4.8. ArticleHandler.\_\_init\_\_(): Constructor

The constructor for the **ArticleHandler** has only two duties. First, it calls the parent class constructor.

litsource

```
# - - -   A r t i c l e H a n d l e r . _ _ i n i t _ _   - - -
def __init__ ( self ):
    """Constructor for ArticleHandler.
    """
```

```

#-- 1 --
# [ self := a new ContentHandler instance ]
ContentHandler.__init__ ( self )

```

Then it initializes the instance variables.

litsource

```

#-- 2 --
self.fileMap = {}
self.outFileName = self.outFile = None

```

## 4.9. ArticleHandler.startElement(): Observe a start tag

The SAX interface requires that the content handler class define a method named `.startElement()` to observe start tags.

litsource

```

# - - -   A r t i c l e H a n d l e r . s t a r t E l e m e n t   - - -

def startElement ( self, name, attrs ):
    """Handle a start tag.

    [ (name is the element name) and
      (attrs is a dictionary containing the attribute names |->
      attribute values) ->
      if this tag starts a fragment ->
        self.outFileName := the fragment's file name
        self.outFile     := the fragment's output file
        self.fileMap     := self.fileMap with an entry
                          mapping the fragment's file name |-> the
                          fragment's output file
      else -> I ]
    """

```

If this start tag isn't a **programlisting** element, or it doesn't have a **role** attribute, we don't care about it. Otherwise, we save the **role** attribute the variable **role**.

litsource

```

#-- 1 --
if name != PROG_ELT:
    return

#-- 2 --
# [ if attrs has a key ROLE_ATTR ->
#   role := that attribute's value
#   else -> return ]
try:
    role = attrs [ ROLE_ATTR ]
except KeyError:
    return

```

If the **role** attribute starts with **outFile:**, we store the rest in **self.outFileName**, signifying that we are now inside a code fragment. If it's not one of our **role** attributes, we return to the caller.

litsource

```

#-- 3 --
# [ if role starts with ROLE_PREFIX ->

```

```

#     self.outFileName := the rest of role
# else -> return ]
if role.startswith ( ROLE_PREFIX ):
    self.outFileName = role [ len ( ROLE_PREFIX ) : ]
else:
    return

```

Next we need an output file handle so that the `.characters()` method will know where to write the code content. If the `self.fileMap` dictionary already has an output file handle in it for this file name, we use that, saving it in `self.outFile`. Otherwise, we open it now and save the file handle in `self.outFile` and also in the `self.fileMap`. Failure to open the output file is a fatal error.

litsource

```

#-- 4 --
# [ if self.fileMap has no key self.outFileName ->
#     self.fileMap := self.fileMap with an entry mapping
#         self.outFileName |-> a writeable file handle for
#             self.outFileName
#     self.outfile := that same file handle
# else ->
#     self.outFile := the corresponding value from
#                     self.fileMap ]
try:
    self.outFile = self.fileMap [ self.outFileName ]
except KeyError:
    try:
        self.outFile = open ( self.outFileName, "w" )
        self.fileMap[self.outFileName] = self.outFile
    except IOError, detail:
        print >> sys.stderr, ( "*** Can't open file "
            "'%s' for writing." % self.outFileName )
        sys.exit(1)

```

## 4.10. ArticleHandler.characters(): Observe text content

The SAX interface stipulates that the content handler have a method called `.characters()` that is called to pass it all textual content. We care about such content only if we are currently inside a code fragment.

litsource

```

# - - -   A r t i c l e H a n d l e r . c h a r a c t e r s   - - -

def characters ( self, text ):
    """Handle text within an element.

    [ text is a string ->
      if self.outFile is not None ->
        self.outFile += text
      else -> I ]
    """

#-- 1 --
if self.outFile is not None:
    self.outFile.write ( text )

```

## 4.11. ArticleHandler.endElement(): Observe an end tag

Again, the name of this method is mandated by the SAX interface as the one it calls when it sees an end tag. If it's a `</programlisting>` end tag, we clear the values of `self.outFileName` and `self.outFile` to signify that we're no longer in a code fragment. Other end tags are ignored.

litsource

```
# - - -   A r t i c l e H a n d l e r . e n d E l e m e n t   - - -

def endElement ( self, name ):
    """Handle the end of an element.

    [ name is an element name ->
      if name==PROG_ELT ->
        self.outFile      := None
        self.outFileName := None
      else -> I ]
    """
    if name == PROG_ELT:
        self.outFile = self.outFileName = None
```

## 4.12. Epilogue

Rather than placing the main at the end of the script, we defined it above (Section 4.5, “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 `litsource` is the main script. Python sets global variable `__name__` to the string `'__main__'` for the outermost script.

litsource

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

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

