XMLTangle - Literate Programming in XML

Jonathan Bartlett

Table of Contents

Literate Programming is a style of programming in which the programmer writes an essay instead of a program. The essay's code fragments are then merged together to form a full program which can be compiled or interpretted. This article is a literate program designed to perform this task with XML documents.

Introduction

Donald Knuth's Literate Programming is a wonderful system for writing programs which are understandable and maintainable. It allows the programmer to not just communicate to the computer, but also communicate the ideas behind the program to current and future programmers. This idea has not caught on, but I believe it is still a worthy goal.

The current literate programming tools are problematic, however. They are still too wedded to individual programming languages and document formats. This program is a version of the tangle program which has the following features:

- Works with any programming language
- Uses XML as the documentation language
- Is not tied to any specific DTD - instead it relies on processing instructions to perform it's tasks

It does not include every feature of literate programming - specifically it does not include any macro facility.

Originally this program was written in C, only worked with the DocBook DTD, and only had a very primitive subset of the literate programming paradigm. Specifically, the code could only be broken up into files - it was not possible to include named code fragments which would be defined elsewhere - you could only append to files. This version is written in Python and captures much more of the literate paradigm.

> **Python and Tangle:** Although this program is technically language-agnostic, it does have some practical problems with languages such as Python. Specifically, since Python's indentation is part of the language itself, it makes it difficult to write literate programs in Python. For example, if you have to insert code into a block, you have to know for sure how indented it is. In a future version, I will add language-specific extensions, including general <?lp-begin-block?> and <lp-end-block> processing instructions, and let the tangle program automatically indent the proper amount. In other languages, this can add the appropriate braces as well, although there would be less need for such a facility in those languages.

* *Need to include info on how it will actually work*

Outline of the program

This program uses Python's SAX parser for handling the data. For more information about Python, XML, and the SAX parser, see http://www.python.org/.

* *Need a more specific link*

The program's outline is as follows:

```
Main Listing =

#!/usr/bin/python

#Import needed libraries
from xml.sax import saxlib, saxexts, saxutils
```

```
import sys, urllib, string, sre, types

{Class to handle SAX parsing functions}
{Class to handle SAX error conditions}
{Instantiate class and run parser}
```

The actual code to run the parser is quite simple:

```
Instantiate class and run parser =

#Create a SAX XML parser
parser = saxexts.make_parser()

#Instantiate our handler and error classes
ldh = LiterateDocumentHandler()
leh = LiterateErrorHandler()

#prepare parser
parser.setDocumentHandler(ldh)
parser.setErrorHandler(leh)

#parse the first argument
parser.parse(sys.argv[1])

#write out the files indicated
ldh.write_files()
```

Notice that we do not care about validating against a DTD, since our program is operating on processing instructions and doesn't care much about the DTD or the elements.

The Literate Document Handler

Here is the outline of the class:

```
Class to handle SAX parsing functions =

class LiterateDocumentHandler(saxlib.DocumentHandler):
 {Class-wide constants}

 def __init__(self):
  {Initialize object variables}

 {Overrided document handling methods}
 {Auxillary document methods}
```

Handling Processing Instructions

Since our program is based on processing instructions, the SAX processing instruction handler is the key function in the program.

```
Overrided document handling methods =

 def processingInstruction(self, target, data):
  {Initialize processing instruction variables}
  {Parse processing instruction into attribute-value pairs}

  {Call appropriate method for processing instruction target}
```

Processing instructions present a parsing problem. Although we want to structure our processing instructions like elements, with attribute-value pairs, XML does not specify anything about how they are formatted, so the parser just hands you the entire content in one string. Therefore, we need to write code to parse the data string into attribute-value pairs. To make this simple, we will use regular expressions. To simplify, we will also force that the attributes be in double-quotes, not single quotes.[2] The following code will parse the variable data into the dictionary pi_attrs.

```
Parse processing instruction into attribute-value pairs =

  while 1:
   try:
    match = self.PIRegex.search(data, regex_start)
    pi_attrs[match.group(1)] = match.group(2)
    regex_start = regex_start + match.end() + 1
   except:
    break
```

The variables used here are initialized in Initialize processing instruction variables. Here is what each variable does -

data

This is the character string after the processing instruction target. This is passed as a parameter

match

This object holds all of the information about the match made.

regex_start

This is the position in the data string that we are currently searching. It starts at 0, so we have to initialize it at the beginning of the function:

```
  regex_start = 0
```

pi_attrs

This is the dictionary that holds the result of our parsing. It has to be initialized at the beginning of the function.

```
  pi_attrs = {}
```

self.PIRegex

This is a precompiled regular expression object. This is initialized when the object is initialized of the object.[3] It is initialized as follows:

```
Initialize object variables +=

  self.PIRegex = sre.compile('([a-z-]+)="([^"]*)"')
```

As you can see it matches any alphabetic character string (which can include dashes as well), followed by an equal sign and a quoted expression. It would be nice to get this closer to the actual parsing of element attributes, but I don't have the XML spec handy.

The parsing section is wrapped in a try/except block. This could be avoided with boundary checking and "no-match" checking, but simply doing it this way meant

I could avoid dealing with these issues. The one drawback to this method is that errors within the processing instructions neither caught nor reported. This could be improved.

Finally, after the processing instruction is parsed, a method is dispatched based on the processing instruction target. Right now, this is just a sequence of ifs. I think I will move it to a target-method dictionary in a future version. Also, I need to move the string constants to the constants section of the program.

```
Call appropriate method for processing instruction target =

  if target == 'lp-section-id':
   self.start_section_id(pi_attrs, data)
  elif target == 'lp-section-id-end':
   self.end_section_id(pi_attrs, data)
  elif target == 'lp-code':
   self.start_code(pi_attrs, data)
  elif target == 'lp-code-end':
   self.end_code(pi_attrs, data)
  elif target == 'lp-ref':
   self.start_ref(pi_attrs, data)
  elif target == 'lp-ref-end':
   self.end_ref(pi_attrs, data)
  elif target == 'lp-file':
   self.match_filename_to_section(pi_attrs, data)
```

Storing Code Sections

This section will concentrate on how the sections of code are read and stored. The basic data structure for storage consists of lists of code fragments, which can also contain lists. Then, there is a dictionary matching each section id to the appropriate code fragment list for that section. This list is later walked to produce the actual code for output. Therefore, we need to initialize our section id to code fragment list dictionary at object-creation time.

```
Initialize object variables +=

  self.sections = {}
```

However, not only do we need to be able to find sections, we also need to find out what section id should be the top-level section of each file. Therefore, we have the declaration

```
Initialize object variables +=

  self.files = {}
```

Now, when a programmer specifies the name of a section, they will probably do it in a nice, human-readable form. However, we need to normalize that into a form that can be keyed off of. The reason that the human-readable form can't be keyed off of is because of problems with spacing, capitalization, and potential symbols within the text. Therefore, we have the following method to normalize the data.[4]

```
Overrided document handling methods +=

 def normalize_id(self, id):
  id = self.matchNonLetterRegex.sub(", id)
  #sre.gsub('[^a-zA-Z]+', ", id)
  id = string.lower(id)
  return id
```

```
Initialize object variables +=

  self.matchNonLetterRegex = sre.compile('[^a-zA-Z]+')
```

This first removes any non-alphabetic character, and then converts it all to lower-case, thus giving the normalized version of the id.

Reading in Code Sections

In order for a section to contain code, it has to be able to read in both a section ID and the code that goes with it. In addition, it has to be able to append multiple code fragments and references to other sections within its text. Therefore, we need to modify what the characters callback function is doing based on what the last processing instruction was. The way that we modify the characters callback is simply by having our standard characters callback only be a dispatch method. It is simply this:

```
Overrided document handling methods +=

 def characters(self, ch, start, length):
  func = self.characters_cb
  func(ch)
```

The instance variable characters_cb is the function that handles the callbacks (usually a bound method) which takes one parameter - the character string. However, this means that we need a default callback initialized when the document handler object is created.

```
Initialize object variables +=

  self.characters_cb = self.default_ch_cb
```

The default characters callback does nothing.

```
Overrided document handling methods +=

 def default_ch_cb(self, ch):
  pass
```

Now, when the processing instruction method gets an lp-section-id processing instruction, it dispatches to the function start_section_id, which sets up the characters callback to read in the current section id.

```
Auxillary document methods +=

 def start_section_id(self, attrs, data):
  self.current_section_id = ''
  self.characters_cb = self.read_section_id_ch_cb
```

The current_section_id instance variable is where the read_section_id_ch_cb will read the section name into.

```
Auxillary document methods +=

 def read_section_id_ch_cb(self, ch):
  self.current_section_id = self.current_section_id + ch
```

Finally, when we hit the lp-section-id-end processing instruction, that turns off the section id reader.

```
Auxillary document methods +=

 def end_section_id(self, attrs, data):
  self.characters_cb = self.default_ch_cb
  self.current_section_id = self.normalize_id(self.current_section_id)
```

Notice that it sets the characters callback back to the default and normalizes the section id. However, this is worthless if no code sections are ever placed here. The lp-code processing instruction is used for that. It dispatches to the following function:

```
Auxillary document methods +=

 def start_code(self, attrs, data):
  id = self.current_section_id
  if self.sections.has_key(id):
   self.current_section = self.sections[id]
  else:
   self.current_section = []
   self.sections[id] = self.current_section

  self.characters_cb = self.read_section_data_ch_cb
```

This checks to see if the current id is yet in the sections instance dictionary. If it isn't, it creates a new list to hold the data, and then stores that list in the dictionary for that id. If it is in the dictionary, it simply pulls that list into the current_section instance variable. It then sets the characters callback function to read section data. Note that the current_section variable is never used except after we have assigned it a value. Therefore, we don't need to initialize it at object creation time. Anyway, the section reader function looks like this:

```
Auxillary document methods +=

 def read_section_data_ch_cb(self, ch):
  self.current_section.append(ch)
```

Finally, when the lp-code-end processing instruction is found, it simply resets the characters callback to the default.

```
Auxillary document methods +=

 def end_code(self, attrs, data):
  self.characters_cb = self.default_ch_cb
```

The current section list and section id are maintained in case the user wants to add additional lp-code sections later under the same id.

Within the code sections, there can also be references to other code sections. This is accomplished by switching the characters callback to read in the section id. If the section id does not yet exist, it is created as empty, and it is included as an object reference in the current list.

```
Auxillary document methods +=

 def start_ref(self, attrs, data):
  self.current_reference = ''
  self.characters_cb = self.read_ref_ch_cb
```

```
def read_ref_ch_cb(self, ch):
 self.current_reference = self.current_reference + ch

def end_ref(self, attrs, data):
 ref = self.current_reference
 ref = self.normalize_id(ref)
 self.characters_cb = self.read_section_data_ch_cb
 if not self.sections.has_key(ref):
  self.sections[ref] = []
 self.current_section.append(self.sections[ref])
```

Since this is only allowed to be called from lp-code sections, after we're done we simply reset the characters callback to read_section_data_ch_cb.

Matching Sections to Files

I decided to match sections to files with the lp-file processing instruction, which has a file attribute for the filename and an id attribute for the section id to put in the file. The processing instruction is dispatched to this function:

```
Auxillary document methods +=

 def match_filename_to_section(self, attrs, data):
  real_id = self.normalize_id(attrs['id'])
  if attrs.has_key('id') and attrs.has_key('file'):
   self.files[attrs['file']] = attrs['id']
```

Which normalizes the id, verifies that all the parameters are in place, and then makes the dictionary mapping.

At the end of the program, we have to write out all of the files to disk. Therefore, we have this handy-dandy method:

```
Auxillary document methods +=

 def write_files(self):
  for file in self.files.keys():
   ostream = open(file, "w")
   ostream.write(self.flatten_array_to_string(self.sections[self.files[file]]))
```

Which looks up the section associate with each file, flattens the code fragment list to a single string, and then writes it to the given file. In the future, I plan to do this so that it doesn't take up so much memory, like has a list walker, which goes through each element and executes a callback function. The flattening function iterates through each element, checks to see if it is a string or a list. If it is a string, it adds it onto the string it is building, otherwise it calls itself with the sublist and adds the result onto the string.

```
Auxillary document methods +=

 def flatten_array_to_string(self, array):
  new_str = ''
  for item in array:
   if type(item) == type([]):
    new_str = new_str + self.flatten_array_to_string(item)
   else:
    new_str = new_str + item

  return new_str
```

The Error Handler

This program does little error-checking anywhere, and is one of the main improvements needed. Therefore, the error handling class is equally short.

```
Class to handle SAX error conditions =

class LiterateErrorHandler:
 def error(self, exception):
  pass

 def warning(self, exception):
  pass

 def fatalError(self, exception):
  pass
```

Future Developments

The following features will be implemented before releasing version 1.0:

- Error checking and reporting to make sure no constructs are used out of order
- Verify that no reference is unused
- Verify that no section contains circular references before printing
- Make sure all sections belong to files
- Include easy-access features for DocBook (i.e. - make programlisting automatically set the first line to the section id and the remaining to lp-code if the role='literate' or something like that)
- Include easy-access features for Python (specifically for block indentation)

The following features are being thought about for some time in the future:

- Creating section indexes for DocBook or other DTDs
- Implementing a backend filter interface to record all top-level declarations in an index

Stylistically, I could also reduce a lot of my functions into lambda functions.

Notes

1. http://www.python.org/
2. This could be easily done with Perl-compatible regular expressions. However, Python's regular expressions make this difficult.
3. Actually, it has to be reused throughout the life of the class, and should be a class variable. This needs to be fixed.
4. Note that there is no reason for this to be a method rather than a class or module function.