Literate Programming with xmltangle

Jonathan Bartlett

Introduction

xmltangle is a program that converts a specially-formatted XML file into source code, implementing the tangle function of literate programming. It reads in a source XML file and parses out the <programlisting> sections, and uses the role attribute to determine which source file to put it in. For example, if you had the code

```
<programlisting role="mainsource.c">
#include &lt;stdio.h&lt;

int main()
{
printf("hello world\n!");
return 0;
}
</programlisting>
```

in a file `example.xml` and ran the command **xmltangle example.xml**, the result would be a file call `mainsource.c` which would be compilable. The first usage of a source file in a role attribute creates a new file, all other usages appends to that file. You may also make multiple source files from the same xml file by varying the role attribute. You can also intermix programlistings between separate source files. However, there is not currently any way to reorder the source listings within the same file. The order they appear in the xml file is the order they will appear in the output code.

Motivation

Documentation has always been a fascinating part of programming for me. I like documentation because it allows other people to become part of the programming process. It makes a program much more "living". Later, I read parts of Donald Knuths book on literate programming (http://www-cs-faculty.stanford.edu/~knuth/lp.html). In that book, Donald Knuth says that a program should not be written as a person describing to a computer how to do a particular job, but instead a program should be written as a person describing to another person how they would tell a computer to do a job. The difference is phenomenal. If you write a program as if you were talking to a person, you find that you have to explain yourself more. Writing a program as a document to someone else also inherently makes the code more maintainable, because just by reading your code, they are better able to understand why you programmed things the way you did.

In literate programming, there are two phases - tangle and weave. Tangling is the act of changing a web (the document describing/containing the literate program) into a source program for a machine. Weaving is the process of typesetting the document for a human to read. Originally, literate programming was done by using TeX for the document language, and Pascal for the programming language. Other people have written many other literate programming engines, but they mostly revolve around TeX. I am a big fan of TeX for the pioneering it made into computer typesetting, however, it really lacks as a general document language because the syntax is really too terrible. XML is much cleaner and easier to use. The only problem I've had is remembering to escape the <s, >s, and &s. You may be able to avoid that using CDATA sections, but I haven't tried that. After learning bits of TeX and LaTeX, I eventually found an XML DTD called DocBook, that was specifically for writing technical documentation. Using the Jade DSSSL stylesheet engine, one could even make use of TeX as a backend for DocBook. From there on out, all of the documentation I've ever written has been in DocBook format. However, one day I was writing some documentation, and I thought that it would be easiest if I just embedded my source code in the documentation like I read about in Donald Knuth's book. I tried it, and it worked great! I wrote a little perl script to do the extracting, and it worked wonderfully. This program is an extension of that perl script. Currently it only supports XML DocBook for its document language (although it supports any target programming language), however I plan on generalizing it so that can be easily changed by using simple command-line switches. It is also missing several of the features that Knuth suggested in his book.

If You Use xmltangle in Your Programs: If you use **xmltangle** in your programs (i.e. - you write an xml file and use **xmltangle** to generate your source files), please be kind to everyone you distribute the file to, and have the pre-generated source files in your tarball. This way, people don't have to already have **xmltangle** to build your program. We don't need useless dependencies floating around.

The program

Here I will describe how the program works. In our source files, we are going to use functions and data structures from both glib and libxml. The shared definitions will be in the file `xmltangle.h`

```
#include <glib.h>
#include <parser.h>
```

```
#include <glib.h>
#include <parser.h>
```

```
#include <parserInternals.h>
#include "xmltangle.h"
```

```
#include <glib.h>
#include <parser.h>
#include "xmltangle.h"
```

The State Machine

While parsing the XML document, the program looks for the element called `programlisting`.

```
/* this is the element used for generating program listings */
#define ELEMENT_NAME "programlisting"
```

It also checks the `role` attribute, to determine which file to put it in

```
/* this is the attribute that the file name is pulled from */
#define ATTRIBUTE_NAME "role"
```

The program essentially has two states:

- it’s in the target element
- it’s not in the target element

```
typedef enum TangleParserState { NOT_IN_ELEMENT, IN_ELEMENT } ParserState;
```

Also, we don’t want to output any of the character data on any elements withing our `programlisting` element (there shouldn’t be any, but we just want to make sure), so we need to always check how deep we are from the `programlisting` element, and only output on level 0. Also, we need to keep track of which file we are currently working on, as well as the files we have used previously, so we know when to open a new one.

```
typedef struct TangleParserData {
ParserState state;
gint in_element_depth;
gchar *file_name;
GSList *used_files;
} TangleParserData;
```

The Parser

The parse consists of callback functions for the SAX interface.

Element Callbacks

The start element callback has three jobs

- Determine if the new element changes program state
- Determine the output file name if the parser changes into the IN_ELEMENT state
- Determine whether the output file should be overwritten or appended to

The callback is passed the parser state data, the element name, and the element attributes:

```
extern void startElementCallback(
gpointer user_data,
const CHAR *name,
const CHAR **atrs
);
```

```
void startElementCallback(
gpointer user_data,
const CHAR *name,
const CHAR **attrs)
{
TangleParserData *parser_data;
FILE *output_file;
gint attribute_index;

parser_data = (TangleParserData *) user_data;
```

Okay, so the first thing we check for is whether or not we are currently in a programlisting element

```
/* don’t process subelements of a source listing */
if(parser_data->state == IN_ELEMENT)
{
parser_data->in_element_depth++;
}
```

If we are not, we check to see if current element is programlisting. This should probably be changed to two callback functions in the parser state (isParserElement and getOutputFilename), but I’ll do that later. Also, if there are not attributes, we just assume it isn’t a valid program listing (because we don’t know what file to put it in). Also, this checks to make sure that the attributes pointer isn’t null

```
else
{
if(attrs == NULL)
{
/* dont process for null attributes */
return;
}

if(strcmp(name, ELEMENT_NAME) == 0)
{
```

Then we look for the role attribute because thats what has the output file name

```
/* look for attribute */
attribute_index = 0;
while(attrs[attribute_index] != NULL)
{
if(strcmp(attrs[attribute_index], ATTRIBUTE_NAME) == 0)
{
/* set up the parser to write the source file */
```

Now that we have the filename, we free the old filename if there was one

```
if(parser_data->file_name != NULL)
{
g_free(parser_data->file_name);
}
```

set up the parser state with the file name, the state, and the depth (0)

```
parser_data->file_name = g_strdup(attrs[attribute_index + 1]);
parser_data->state = IN_ELEMENT;
parser_data->in_element_depth = 0;
```

and rewrite the file if it hasn’t been seen before (this used to be in the characters callback, but it wouldn’t handle the case of having an empty programlisting element if someone wanted to write a blank file).

```
if(g_slist_find_custom(
parser_data->used_files,
parser_data->file_name,
(GCompareFunc)strcmp
) == NULL)
{
output_file = fopen(
parser_data->file_name,
"w"
);
fclose(output_file);
parser_data->used_files =
g_slist_prepend(
parser_data->used_files,
g_strdup(parser_data->file_name)
);
}
```

If the role wasn't right, just go to the next attribute. Afterwards, return

```
} /* end attribute check */

attribute_index++;
attribute_index++;
} /* end attribute search */
} /* end element check */
} /* end state checks */
} /* end function */
```

At the end of the element, all we need to do is check our state, our depth, and alter our state if necessary.

```
extern void endElementCallback(
gpointer user_data,
const CHAR *name
);

void endElementCallback(
gpointer user_data,
const CHAR *name)
{
TangleParserData *parser_data;

parser_data = (TangleParserData *)user_data;

if(parser_data->state == IN_ELEMENT)
{
if(parser_data->in_element_depth > 0)
{
parser_data->in_element_depth-;
}
else
{
parser_data->state = NOT_IN_ELEMENT;
}
}
}
```

Characters Callback

The characters callback is called whenever the XML parser encounters character data. Some entities are automatically converted and passed in as text. However, I am not sure how user-defined entities are handled (either regular or system entities). If anyone knows how to do this with the SAX interface, let me know. Anyway, the function definition is simple:

```
extern void charactersCallback(
gpointer user_data,
const CHAR *chars,
int lent
);
```

```
void charactersCallback(
gpointer user_data,
const CHAR *chars,
int len)
{
TangleParserData *parser_data;
gchar *string;
FILE *output_file;

parser_data = (TangleParserData *)user_data;
```

All this function really does is check to see if the parser is in the correct state, and if so, appends the text to the file (file creation was handled in the start element callback).

```
if(parser_data->state == IN_ELEMENT)
{
if(parser_data->in_element_depth == 0)
{
/* create a file if it hasn’t been mentioned before, otherwise ap-
pend to it */
output_file = fopen(parser_data->file_name, "a");
if(output_file != NULL)
{
/* the string duplication is to null
   terminate the string, because the
   string we get is not null terminated
*/
string = g_strndup(chars, len);
fputs(string, output_file);
fclose(output_file);
g_free(string);
}
else
{
g_printerr("warning: error writing to file %s\n", parser_data->file_name);
}
}
}
}
```

Entity Callback

This was ripped straight from the gnorpm source, and I have no idea what it does or how it does it.

```
extern xmlEntityPtr getEntityCallback(
gpointer user_data,
const CHAR *name
);
```

```
xmlEntityPtr getEntityCallback(
gpointer user_data,
const CHAR *name)
{
return xmlGetPredefinedEntity(name);
}
```

Parser Driver

The parser driver is very simple. It just initializes the SAX XML interface with the first command line argument, and tells it to run, and then frees the memory. Someday I will add real command-line options. But that day isn't today.

Okay, first we need to tell SAX what callbacks we are using. It would probably be easier to just make an instance, zero it, and then set the ones we needed, but I was just ripping gnorpm source code blindly, and this is what I came up with:

```
/* set up callback structure */
xmlSAXHandler tangleParserTemplate = {
NULL, /* internalSubset */
NULL, /* isStandalon */
NULL, /* hasInternalSubset */
NULL, /* hasExternalSubset */
NULL, /* resolveEntity */
getEntityCallback,
NULL, /* entityDecl */
NULL, /* notationDecl */
NULL, /* attributeDecl */
NULL, /* elementDecl */
NULL, /* unparsedEntityDecl */
NULL, /* setDocumentLocator */
NULL, /* startDocument */
NULL, /* endDocument */
startElementCallback,
endElementCallback,
NULL, /* reference */
charactersCallback,
NULL, /* ignorableWhitespace */
NULL, /* processingInstruction */
NULL, /* comment */
NULL, /* warning */
NULL, /* error */
NULL  /* fatal */
};
```

Note that this is a template. That’s because I don’t believe in global variables, except for holding command-line options and that such thing. This will be copied to the variable that SAX is actually sent. Okay, here’s the main function:

```
gint main(gint argc, gchar *argv[])
{
xmlParserCtxtPtr ctxt;          /* this is the SAX parser */
TangleParserData parser_data;   /* this is my parser state data */
xmlSAXHandler theParser;        /* this is my callback struct */
```

So, first I need to initialize my structures:

```
/* copy my template */
theParser = tangleParserTemplate;

parser_data.state = NOT_IN_ELEMENT;
parser_data.in_element_depth = 0;
parser_data.file_name = NULL;
parser_data.used_files = NULL;
```

Now, I need to attempt to parse the XML file passed on the command line

```
if(argc < 1)
{
g_printerr("No file specified!");
exit(1);
}

ctxt = xmlCreateFileParserCtxt(argv[1]);
if(ctxt == NULL)
{
g_printerr("File %s could not be parsed", argv[1]);
exit(1);
}
```

Now, that I have my XML parser set up, I need to given it my callback handler and my state data.

```
ctxt->sax = &theParser;
ctxt->userData = (gpointer) &parser_data;
```

Now, the parser just needs to run, and I need to print errors if there are any

```
xmlParseDocument(ctxt);

if(! ctxt->wellFormed)
{
g_printerr("File %s is not a well-formed xml file -
parsing may have been incomplete\n", argv[1]);
exit(1);
}
```

And now, I just free my structures and exit

```
/* this prevents the next statement from freeing my structure */
ctxt->sax = NULL;
xmlFreeParserCtxt(ctxt);

g_slist_free(parser_data.used_files);

return 0;
} /* end of program */
```

Bugs and Potential Improvements

- As already mentioned, be able to customize which elements and attributes are used to generate the source files
- Allow a method to reorder source elements
- Integrate as an option to gcc
- For C files, generate #file and #line instructions
- Need to make an autoconf script, so that this only utilizes libxml and glib (removing any other gnome dependencies, like using gnome-config for makefiles)
- Need to make some autoconf and automake rules for using **xmltangle**
- Need to incorporate the build process into xmltangle.xml
- Need to figure out how to access user-defined and system entities with the SAX interface
- Need to autocreate directories when the file name contains slashes in it (currently it will write the file only if the directory is available
- Need to have a command-line option to include a license in every source file, or a generic interface to have it only expand certain entities when writing to the source file or in special circumstances
- Need to tailor to large-scale projects. Since this program is as yet the only thing written like this, I'm certain that changes will need to be made for this to accomodate projects on a larger scale.
- Need to figure out why SAX gives a well-formed error when given a DOCTYPE declaration
- Need to standardize a way to list bugs and fixes within the code, so later maintainers can see exactly why options were chosen