atangle

COLLABORATORS

	TITLE : atangle		
ACTION	NAME	DATE	SIGNATURE
WRITTEN BY	Jordi Fita	February 6, 2018	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
8c8df57a6f06	2011-08-20	Moved the download links for atangle to big buttons.	jfita
34b7522b4f97	2011-03-28	atangle is now using a new style for directives which don't collide with XML tags. I had to update all games and programs as well in order to use the new directive syntax.	jfita
6cc909c0b61d	2011-03-07	Added the comments section.	jfita
d5cc1bcb5948	2010-10-28	The correct source language for Makefiles is make.	jfita
854feca7a1b6	2010-10-27	Added the source style and thus highlighting to the Makefile.	jfita
658793e2ad92	2010-10-25	Added the Makefile to atangle.	jfita
e752ea890166	2010-10-22	atangle's license is now longer syntax highligthed.	jfita
05a1b32f8b4a	2010-10-22	The appendix sections now aren't actual appendix when making a book.	jfita
0ab76df46149	2010-10-20	Added the download links.	jfita
a47b37b23119	2010-10-19	Fixed while block's indentation.	jfita

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
24c420624034	2010-10-18	Reworded some paragraphs that were difficult to understand.	jfita
38abe4173b21	2010-10-18	Fixed typo in atangle.txt.	jfita
8fcc5ea10635	2010-10-18	Changed atangle's main source file description.	jfita
91aaff328f42	2010-10-18	Merged the options section of atangle inside the root snippet's description.	jfita
1ac07ccc50b2	2010-10-18	Atangle's license is now an appendix.	jfita
b8cefc9d0ec4	2010-10-18	atangle now supports dots ('.') as snippets' name.	jfita
fb94ebbd979d	2010-10-15	Added a missing source tag to atangle.	jfita
3eb59ece1e5c	2010-10-14	Added the root snippet options to atangle.	jfita
1f0c91d3ebab	2010-10-14	atangle now can load documents from files.	jfita
84f0fad36ff1	2010-10-14	I've reworded most of atangle. I also changed the main to be a standalone function and uses AWeb.ATangle as an object instead of a bunch of functions.	jfita
f65df20b9ff8	2010-10-14	atangle's references now can contain spaces.	jfita
ff481182607e	2010-10-14	Fixed typo in atangle.	jfita
8af0238ef205	2010-10-13	"Fixed" a line too long.	jfita
d25a52536ad7	2010-10-13	Remplaced the code filter to source filter in atangle.	jfita
fff0e129835f	2010-10-13	Added the functional atangle application written in aweb.	jfita

Contents

1   Introduction

atangle is a simple literate programing tool intended to extract blocks of source code, or *code snippets*, from literate programs written using AsciiDoc, and then reorder the snippets into source code modules suitable to be feed into a compiler and produce a binary application or library.

2   Literate Programming Blocks

In every literate application there are two kinds of blocks: *documentation blocks* and *code blocks*.

The documentation block is the main text with which the logic and decisions taken during the development of the program are explained in enough detail for a human being to understand completely.

Between the documentation blocks there are code blocks, also called code snippets, in where the programmer writes the actual compilable code. Each of these snippets has a name that can be used within other snippets as *references*.

Once atangle has extracted all code snippets from the input file, it starts writing out each line of the *root snippet*, by convention named *, replacing every reference by the snippet's contents. In turn, the references in the referenced snippet are also recursively replaced until all references are resolved. The output of atangle is thus a valid source module as the compiler or interpreter expects it to be written.

To embed code snippets within the documentation, we will write the code inside AsciiDoc's *listing blocks*, which consist of any number of lines between two delimiters. The two delimiters are identical lines, usually four dashes, one at the beginning and one at the end of the block. When rendering, AsciiDoc simply copies the contents of any listing block without formating and therefore are the ideal place to write the snippets, since AsciiDoc will still render the blocks correctly without the use of any other tool.

Besides the actual code, in order to identify a listing block as a valid code snippet, the first line must be a directive that assigns a name to the snippet.

2.1   Directives

While scanning a listing block, atangle looks for the following recognized directives:

- The default root code snippet definition-extension: `<<*>>=`

- A named code snippet definition-extension: `<<name>>=`

- A named code snippet reference: `<<name>>`

3   Collecting code snippets

In order to find the code snippets atangle will read each input's line and look for the start of an listing block. We will use AsciiDoc's default starting block markup of four dashes (`----`) although it is possible to change AsciiDoc's configuration to use a different markup.

As we are reading the AsciiDoc document line by line, we can use `FileStream`'s `read_line()` member function. This function strips out the end of line character from the returned string and when it reaches the end of the stream it returns `null`. Therefore just checking the string's nullity we know when to stop without a call to `eof()`.

```
<<look for listing blocks>>=
public void process_document(GLib.FileStream input) {
    string line = input.read_line();
    while (line != null) {
        if (line == "----") {
            process_listing_block(input);
        }
        line = input.read_line();
    }
}
```

Once we have identified a listing block, we must make sure this is a valid code snippet by looking at the first line for the named snippet definition-extension directive. We can express this directive using a regular expression, but summarizing the directive has a name part, which can't be empty, in between angle brackets (<< and >>) followed by the definition-extension action identified by a single equal sign at the end (=).

In Vala the regular expression becomes:

```
<<web definition-extension directive regexp>>=
web_definition_extension = new GLib.Regex("^<<(\\\*|[-\\s\\w\\.]+)>>=\\s*$");
```

If the first listing block's line matches the definition-extension directive, then we can get the snippet's name from the first subexpression group. Otherwise, we just ignore this listing block.

```
<<process listing block>>=
private void process_listing_block(GLib.FileStream input) {
    GLib.MatchInfo nameMatch;
    string line = input.read_line();
    if (line == null) {
        return;
    }
    if (!web_definition_extension.match(line, 0, out nameMatch)) {
        return;
    }
    string name = nameMatch.fetch(1);
```

Each snippet is nothing but a list of lines where each line is either the actual code that we want to compile or a reference to another snippet. Since the number of lines that a snippet can have is arbitrary, we store the snippet as a dynamic list of strings.

```
<<snippet class definition>>=
class AWeb.Snippet: Gee.ArrayList<string>{}
```

Therefore, for each snippet we just read from the input stream and store each line to the list, except for the first line, which is the snippet's name directive, and the last line, which marks the end of the listing block.

```
<<process listing block>>=
    Snippet snippet = new Snippet();
    line = input.read_line();
    while (line != null && line != "----") {
        snippet.add(line);
        line = input.read_line();
    }
```

3.1 Storing the snippets

Since later we need to refer to the code snippets by name the most natural way of storing the snippets is by using a *dictionary* data structure. In Vala, the best solution is to use a `HashMap` where the key is the snippet's name and the value its lines.

```
<<snippets dictionary>>=
private Gee.HashMap<string, Snippet> snippets =
    new Gee.HashMap<string, Snippet>();
```

Once we extracted the snippet from the listing block, we must **append** the snippet's contents if there is already another snippet with the same name. Otherwise, we just add the snippet to the dictionary.

```
<<process listing block>>=
    if (snippets.has_key(name)) {
        snippets[name].add_all(snippet);
    } else {
        snippets.set(name, snippet);
    }
}
```

4 Output

With all the snippets collected inside the snippets dictionary, we are now able to output the actual source code in the proper order starting from the *root* snippet. Usually, the root snippet is identified by $\star$ (asterisk.)

The nature of printing snippets is recursive. For each line of the snippet, we must analyze if the line is a reference directive, in which case we must retrieve the referenced code snippet and recursively resolve any references within, or is **not** a directive, in which case we just print it.

An reference directive is a line whose content's is just a snippet's name is delimited by angle brackets, optionally preceded by spaces. This can be described by the following regular expression.

```
<<web reference directive regexp>>=
web_reference = new GLib.Regex("^(\\s*)<<([-\\s\\w\\.]+)>>\\s*$");
```

Notice how we must get the space in front of the reference to be able to set the correct indentation when printing the dereferenced lines of code. We, therefore, need to start from the root snippet without any indentation at all.

By default, the root snippet is named $\star$ (asterisk), but there is a command line option to specify a different root snippet. This is useful, for example, when we are working with C++ modules and we have different root snippets for the header and the actual code module.

We will create a new Options class that only have attributes members to be used as references to the GLib.OptionEntry. Currently, the only option available is the root_snippet string.

```
<<command line parameters>>=
public class Options: GLib.Object {
    public static string root_snippet = null;
}

const GLib.OptionEntry[] options = {
    { "root-snippet", 'r', 0, GLib.OptionArg.STRING, ref Options.root_snippet, "The name of ↩
        the root snippet", "name"},
    { null }
};
```

Then, we just need to create a new OptionContext and parse the arguments passed in the command line options. If the -r options is passed, then the root snippet's name gets stored in Options.root_snippet.

```
<<parse command line>>=
var context = new GLib.OptionContext("<document>");
context.add_main_entries(options, null);
try {
    context.parse(ref args);
```

```
}
catch (GLib.OptionError e) {
    stderr.printf("\n%s\n\n", e.message);
    stderr.printf("%s", context.get_help(true, null));
    return -1;
}
```

If the root snippet is not specified, then `Options.root_snippet` is left to `null` and we use the default name of `*` (asterisk.)

```
<<dereferencing root snippet>>=
atangle.print_snippet(Options.root_snippet ?? "*", "", stdout);
```

Notice how the `indent` parameter is an empty string to start without indentation. Then, as we print each line and resolve references we need to append the reference's indentation before resolving.

```
<<print snippet>>=
public void print_snippet(string name, string indent, GLib.FileStream output)
{
    if (snippets.has_key(name)) {
        foreach(string line in snippets[name]) {
            GLib.MatchInfo referenceMatch;
            if (web_reference.match(line, 0, out referenceMatch)) {
                print_snippet(referenceMatch.fetch(2), indent +
                    referenceMatch.fetch(1), output);
            } else {
                output.printf("%s%s\n", indent, line);
            }
        }
    }
```

We should also warn whenever we try to use an unknown reference.

```
<<print snippet>>=
    else {
        stderr.printf("Unknown reference: %s\n", name);
    }
}
```

5 Input Document

atangle can read the document either from the standard input or from a file. From the point of view of Vala, both are instances of `FileStream`.

To know whether we must read from the standard input or a file, we will use the parameters passed to `main`. If after parsing the command line options, there is any parameter, besides the first parameter which is the call to the application, we assume that parameter is the file we should read from instead of the standard input.

```
<<main>>=
<<command line parameters>>
int main(string[] args) {
    <<parse command line>>
    unowned FileStream input = GLib.stdin;
    GLib.FileStream file;
    if (args.length > 1) {
        file = GLib.FileStream.open(args[1], "rt");
        if (file == null) {
            stderr.printf("Couldn't open '%s'\n", args[1]);
        }
        input = file;
```

```
    }
    AWeb.Atangle atangle = new AWeb.Atangle();
    atangle.process_document(input);
    <<dereferencing root snippet>>
    return 0;
}
```

6   The program's structure

The final program looks like this:

```
<<*>>=
/*
   atangle - Extracts and sorts the code snippets from aweb documents.
   Copyright 2010 Jordi Fita <jfita@geishastudios.com>

   <<license>>
*/

<<snippet class definition>>
class AWeb.Atangle: GLib.Object {
    <<snippets dictionary>>
    private GLib.Regex web_definition_extension;
    private GLib.Regex web_reference;

    public Atangle()
    {
        <<web definition-extension directive regexp>>
        <<web reference directive regexp>>
    }

    <<look for listing blocks>>
    <<process listing block>>
    <<print snippet>>
}

<<main>>
```

A   Makefile

Being a simple application, an small Makefile would be sufficient to build and link `atangle` from the source document.

The first thing that needs to be done is to extract the Vala source code from the AsciiDoc document using `atangle` itself. It is necessary, therefore, to have a prebuilt version of `atangle` to extract its source code.

```
<<extract vala source code>>=
atangle.vala: atangle.txt
        atangle $< > $@
```

Even though it is possible to compile and build directly the `atangle` executable, in order to avoid a dependency to Vala's compiler when distributing the source code, it is custom to generate the C source from the Vala's. In this case, I need to use the `gee-0.8` package because `atangle` uses `ArrayList` and `HashMap`, which belong to this package.

```
<<generate C source code>>=
atangle.c: atangle.vala
        valac --pkg gee-0.8 -C -o $@ $<
```

Finally is possible to link the executable from the generated C source code. Although, I have to take into account the platform executable suffix. For Linux and other UNIX systems, the suffix is the empty string, but for Windows I need to append *.exe* to the executable.

To know which system is the executable being build, I'll use the `uname -s` command, available both in Linux and also in MinGW or Cygwin for Windows. In this case, I only detect the presence of MinGW because I don't want to add yet another dependency to Cygwin's DLL.

```
<<determine executable suffix>>=
UNAME = $(shell uname -s)
MINGW = $(findstring MINGW32, $(UNAME))
```

Later, I just need to check if the substring *MINGW* is contained in the output of `uname`. If the `findstring` call's result is the empty string, then we assume we are building in a platform that doesn't have executable suffix.

```
<<determine executable suffix>>=
ifneq ($(MINGW),)
EXE := .exe
endif
```

With this suffix, I can now build the final executable. In this case, I need to link against `glib-2.0`, which is a dependency for every Vala application, and `gee-0.8`, as already explained.

```
<<build atangle executable>>=
atangle$(EXE): atangle.c
        gcc -o $@ $< `pkg-config --cflags --libs glib-2.0 gee-0.8`
```

Sometimes, it is necessary to remove the executable as well as the intermediary building artifacts. For this, I'll add a target named `clean` that will build all the files built by the Makefile and only left the original document. I have to mark this target as `PHONY` in case there is a file named `clean` in the same directory as the Makefile.

```
<<clean build artifacts>>=
.PHONY: clean

clean:
        rm -f atangle$(EXE) atangle.vala atangle.c
```

As the first defined target is the Makefile's default target, I'll place the executable first and then the dependences until the original document. After all source code targets, I'll put the `clean` target. This is not required, but a personal choice. The final Makefile's structure is thus the following.

```
<<Makefile>>=
<<determine executable suffix>>

<<build atangle executable>>

<<generate C source code>>

<<extract vala source code>>

<<clean build artifacts>>
```

B License

This program is distributed under the terms of the GNU General Public License (GPL) version 2.0 as follows:

```
<<license>>=
This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License version 2.0 as
```