DoccoTeX is totally based on Docco. The output produced can be included in some LaTex document. The silly LaTeX files produced might or might not be a good fit for you, just modify `resources/docco.jst` to adapt it to your needs.
Comments are passed through a Markdown to LaTeX converter, and code is listed using Minted.
If you install DoccoTeX, you can run it from the command-line:

```
doccotex src/*.coffee
```

. . . will generate LaTeX files for each source file in a `docs` folder.
The source for Docco is available on GitHub, and released under the MIT license.
To install Docco, first make sure you have Node.js, Pandoc , and CoffeeScript. Then:

```
git clone git://github.com/ogf/doccotex.git
npm install <directory-doccotex-has-been-cloned>
```

Docco can be used to process CoffeeScript, JavaScript, Ruby, Python, or TeX files. Only single-line comments are processed — block comments are ignored.

0.0.1 Main Documentation Generation Functions

Generate the documentation for a source file by reading it in, splitting it up into comment/code sections, converting the comments from Markdown to LaTeX and merging them into the `resources/docco.jst` template.

```
generate_documentation = (source, callback) ->
  fs.readFile source, "utf-8", (error, code) ->
    throw error if error
    sections = parse source, code
    markdown_to_latex source, sections, ->
      generate_latex source, sections
      callback()
```

Given a string of source code, parse out each comment and the code that follows it, and create an individual **section** for it. Sections take the form:

```
{
  docs_text: ...
  docs_latex: ...
  code_text: ...
}
```

```
parse = (source, code) ->
  lines    = code.split '\n'
  sections = []
  language = get_language source
  has_code = docs_text = code_text = ''

  save = (docs, code) ->
    sections.push docs_text: docs, code_text: code

  for line in lines
    if line.match(language.comment_matcher) and not line.match(language.comment_filter)
      if has_code
        save docs_text, code_text
        has_code = docs_text = code_text = ''
      docs_text += line.replace(language.comment_matcher, '') + '\n'
    else
      has_code = yes
      code_text += line + '\n'
  save docs_text, code_text
  sections
```

We process the entire file in a single call to Pandoc by inserting little marker comments between each section and then splitting the result string wherever our markers occur.

```
markdown_to_latex = (source, sections, callback) ->
  language = get_language source
  pandoc = spawn 'pandoc', ['-f', 'markdown', '-t', 'latex']
  output   = ''

  pandoc.stderr.addListener 'data',  (error)  ->
    console.error error.toString() if error

  pandoc.stdin.addListener 'error',  (error)  ->
    console.error "Could not use Pandoc to convert the docs to LaTeX."
    process.exit 1

  pandoc.stdout.addListener 'data', (result) ->
    output += result if result

  pandoc.addListener 'exit', ->
    fragments = output.split section_marker
    for section, i in sections
      section.docs_text = fragments[i]
    callback()

  if pandoc.stdin.writable
```

Two newlines added after section_marker. Otherwise it interferes with the output generated by Pandoc

```
    to_write = (section.docs_text for section in sections)
                       .join(section_marker+"\n\n")
    pandoc.stdin.write(to_write)
    pandoc.stdin.end()
```

Once all of the code has been splitted and the docs converted to LaTeX, we can generate the LaTex file and write out the documentation. Pass the completed sections into the template found in `resources/docco.jst`.

```
generate_latex = (source, sections) ->
  title = path.basename source
  dest  = destination source
  html  = docco_template {
    title: title, sections: sections, sources: sources, path: path,
    destination: destination, language: get_language(source)
  }
  console.log "doccotex: #{source} -> #{dest}"
  fs.writeFile dest, html
```

0.0.2 Helpers & Setup

Require our external dependencies

```
fs       = require 'fs'
path     = require 'path'
{spawn, exec} = require 'child_process'
```

A list of the languages that Docco supports, mapping the file extension to the name of the Pygments lexer and the symbol that indicates a comment. To add another language to Docco's repertoire, add it here.

```
languages =
  '.coffee':
    name: 'coffee-script', symbol: '#'
  '.js':
    name: 'javascript', symbol: '//'
  '.rb':
    name: 'ruby', symbol: '#'
  '.py':
    name: 'python', symbol: '#'
  '.tex':
    name: 'tex', symbol: '%'
  '.latex':
    name: 'tex', symbol: '%'
  '.c':
    name: 'c', symbol: '//'
  '.h':
    name: 'c', symbol: '//'
```

Build out the appropriate matchers and delimiters for each language.

```
for ext, l of languages
```

Does the line begin with a comment?

```
  l.comment_matcher = new RegExp('^\\s*' + l.symbol + '\\s?')
```

Ignore hashbangs and interpolations...

```
  l.comment_filter = new RegExp('(^#![/]|^\\s*#\\{)')
```

Get the current language we're documenting, based on the extension.

```
get_language = (source) -> languages[path.extname(source)]
```

Compute the destination HTML path for an input source file path. If the source is `lib/example.coffee`, the HTML will be at `docs/example.html`

```
destination = (filepath) ->
  'docs/' + path.basename(filepath, path.extname(filepath)) + '.tex'
```

Ensure that the destination directory exists.

```
ensure_directory = (dir, callback) ->
  exec "mkdir -p #{dir}", -> callback()
```

Micro-templating, originally by John Resig, borrowed by way of Underscore.js.
Some fixes added so typical LaTeX backslashes are not lost. Newlines are also preserved wherever possible.

```
template = (str) ->
  new Function 'obj',
    'var p=[],print=function(){p.push.apply(p,arguments);};' +
    'with(obj){p.push(\'' +
    str.replace(/[\r\t]/g, " ")
       .replace(/\\(?=[^<]*%>)/g,"\t")
       .replace(/\\/g, "\\\\")
       .replace(/\t/g, "\\")
       .replace(/%>\s*?\n/g, "%>")
       .replace(/\n/g, "\\n")
       .replace(/'(?=[^<]*%>)/g,"\t")
       .split("'").join("\\'")
       .split("\t").join("'")
       .replace(/<%=(.+?)%>/g, "',$1,'")
       .split('<%').join("');")
       .split('%>').join("p.push('") +
       "');}return p.join('');"
```

Create the template that we will use to generate the Docco HTML page.

```
docco_template  = template fs.readFileSync(__dirname + '/../resources/docco.jst').toString()
```

The dividing token we feed into Pandoc, to delimit the boundaries between sections.

```
section_marker = '\ndoccotexdivider'
```

Run the script. For each source file passed in as an argument, generate the documentation.

```
sources = process.ARGV.sort()
if sources.length
  ensure_directory 'docs', ->
    files = sources.slice(0)
    next_file = -> generate_documentation files.shift(), next_file if files.length
    next_file()
```