

Argot – version 1.0-beta

<http://argot.x9c.fr>

Copyright © 2010-2011 Xavier Clerc – argot@x9c.fr
Released under the GPL v3

July 14, 2011

Introduction

Argot is an enhanced HTML generator for the `ocamldoc` comment-extraction tool for the Objective Caml language¹. Its name stems from the following acronym: *Argot is a Raised Generator for the Ocamldoc Tool*.

Argot, in its 1.0-beta version is designed to work with version 3.12.1 of Objective Caml. Argot is released under the GPL version 3. This licensing scheme should not cause any problem, as documentation generation will not contaminate code.

Bugs and requests for enhancement should be reported at <http://bugs.x9c.fr>.

Building Argot

Argot can be built from sources using `make` (in its `GNU Make 3.81` flavor), and Objective Caml version 3.12.1. No other dependency is needed. Following the classical Unix convention, the build and installation process consists in these three steps:

1. `sh configure`
2. `make all`
3. `make install`

During the first step, one can specify elements if they are not correctly inferred by the `./configure` script; the following switches are available:

- `-ocaml-prefix` to specify the prefix path to the Objective Caml installation (usually `/usr/local`);
- `-ocamlfind` to specify the path to the `ocamlfind` executable (notice that the presence of `ocamlfind`² is optional, and that the tool is used only at installation if present);
- `-no-native-dynlink` to disable the build of the native version of the generator, even if native dynamic linking is available.

¹The official Caml website can be reached at <http://caml.inria.fr> and contains the full development suite (compilers, tools, virtual machine, *etc.*) as well as links to third-party contributions.

²Findlib, a library manager for Objective Caml, is available at <http://projects.camlcity.org/projects/findlib.html>

During the third and last step, according to local settings, it may be necessary to acquire privileged accesses, running for example `sudo make install`.

Running Argot

Through direct call to `ocamldoc`

Once installed, using Argot is as simple as switching from:

```
ocamldoc -html -d destination-path files
```

to:

```
ocamldoc -i argot-path -g argot.cmo -d destination-path files
```

where *argot-path* is ‘`ocamlfind query argot`’ when installed through `ocamlfind`, and to either:

```
ocamldoc -i +custom -g argot.cmo -d destination-path files  
or ocamldoc -g argot.cmo -d destination-path files
```

if not installed through `ocamlfind`.

Through `ocamlbuild`

First, one needs to create a *modules.odocl* file containing the list of modules to generate documentation for. Then, the following line should be added to the `_tags` file:

```
‘‘modules.docdir/index.html’’: argot
```

To handle the `argot` flag, it is necessary to define an `ocamlbuild` plugin, like the one presented by code sample 1. Finally, it is possible to call `ocamlbuild`, taking care to specify an `ocamldoc` version compatible with the generator. The example using ‘`argot.cmo`’, it is necessary to pass the `-ocamldoc /path/to/ocamldoc` command-line switch in order to ensure that this is the bytecode version of `ocamldoc` that will be actually called by `ocamlbuild`.

Using Argot

Text formatting

In addition to the already available `{b ...}` (for bold), `{i ...}` (for italic), and `{e ...}` (for emphasized), Argot provides the following text formatting modifiers:

- `{s ...}` for stroke;
- `{u ...}` for underline;
- `{h ...}` for highlight.

Code sample 1 Example of ocamlbuild plugin.

```
open Ocamlbuild_plugin
open Ocamlbuild_pack

let () =
  dispatch begin function
    | After_rules ->
      flag ["argot"] (S[A"-i"; A"+custom"; A"-g"; A"argot.cmo"]);
      let myocamldoc tags =
        Ocaml_tools.ocamldoc_l_dir (tags -- "extension:html") in
      rule "ocamldoc: argot"
        ~prod:"%.docdir/index.html"
        ~dep:"%.odocl"
        ~stamp:"%.docdir/html.stamp"
        ~insert:'top
        (Ocaml_tools.document_ocaml_project ~ocamldoc:myocamldoc
         "%.odocl"
         "%.docdir/index.html"
         "%.docdir")
    | _ -> ()
  end
```

Tables

In order to define tables, the following elements can be used:

- `{table ...}` to actually define a table;
- `{caption ...}` to define its associated caption;
- `{row ...}` to add a row to the table;
- `{header ...}` to add an header cell to the row;
- `{data ...}` to add a data cell to the row;
- `{span n ...}` (where n should be a positive integer) to add a data cell spanning n columns to the row.

Here is an example of a complete table definition:

```
{table {caption the caption}
  {row {header key} {header value}}
  {row {data key1} {data {i data1}}}}
  {row {data key2} {data {i data2}}}}
  {row {span 2 summary}}}}
```

Token substitution

Token substitution allows one to use the value of either an environment variable or a command-line switch into an HTML page. This may for example be useful to insert the current date, or to specify the path of an element at documentation generation time:

```
{token DATE}  
file {token FILE_PATH}/file.ext
```

The value of a token is first searched in `-define id value` switches passed to the `ocamldoc` tool, and then searched among the environment variables.

Images

It is possible to include images into the generated pages through `{image path}`. The image data will be directly embedded into the page using base64 format, in such a way that no external link remains in the generated HTML. There is thus no need to package the image along with the HTML pages.

To avoid to use the full path to the image, it is possible to use the aforementioned token substitution inside the `image` formatter:

```
{image {token IMAGE_PATH}/img.png}
```

Then, one has to specify the value for `IMAGE_PATH` on the command-line through the `-define` switch seen above:

```
ocamldoc -define IMAGE_PATH /path/to/images ...
```

Folding









When some explanation, albeit useful, is long and/or may appear as a digression, it is possible to *fold* it. It means that the text inside a `{fold digression}` will appear as an ellipsis (*i.e.* ...) and will be *unfolded* (that is revealed) when the ellipsis will be clicked. At the opposite, clicking on the ellipsis while the *foldable* text is visible will make it disappear. Any formatting instruction can be used in the digression.

Additional tags

Argot also defines a bunch of new tags that can be used to enhance documentation. Some of these tags come with image icons; these have been designed by Mark James, released under the Creative Commons Attribution 2.5 License, and are available at <http://www.famfamfam.com/lab/icons/silk/>.

The additional tags are:

- `@obvious`, a bare placeholder;
- `@typevar`, to document type variables in the same way `@param` is used for parameters;
- `@copyright` and `@license` to be used along with the `@author` parameter;
- `@alias`, `@synonym`, and `@equivalent`, to define synonyms or equivalences;

- `@todo`, or `@unimplemented`, to mark an implementation-in-progress;
- `@todoc`, or `@docme`, to mark a documentation-in-progress;
- `@tofix`, or `@fixme`, to mark a fix-in-progress;
- `@stateful`, to mark that a given function relies on a state;
- `@threadsafe`, to mark that a given function can be used in a multithread context;
- `@threadsafe`, to mark that a given function cannot be used in a multithread context;
- `@attention`, to introduce text by the  icon;
- `@bug`, to introduce text by the  icon;
- `@error`, to introduce text by the  icon;
- `@info`, to introduce text by the  icon;
- `@new`, to introduce text by the  icon;
- `@note`, to introduce text by the  icon;
- `@remark`, to introduce text by the  icon;
- `@warning`, to introduce text by the  icon.