CS11 – Erlang

Winter 2012–2013

Lecture 4

Erlang Documentation Generator

- Erlang includes a documentation-generator tool called **edoc**
  - **edoc** is actually an Erlang module containing various entry-points, functions, etc.
- **edoc** is inspired by Javadoc
  - Write specially formatted comments in your source files
  - Run **edoc** tool against your sources to get HTML etc.
- Many common themes between **edoc** and Javadoc
  - Architecture and documentation tags are very similar
  - **edoc** includes several additional capabilities over Javadoc
  - (…and it's also harder to use than Javadoc…)

What Can Be Documented?

- **edoc** allows you to document modules and functions
  - The two main units of code supported in Erlang
- Can't directly document macros or records ☹
- **edoc** has a way of specifying abstract types
  - Use this to describe records, tuples, or other commonly used structures this way
- Function docs include a specification:
  - function_name(arg1_type, …) -> result_type
  - Can refer to the abstract types in these specs

Module Documentation

- Documentation is specified using @tags in Erlang comments
  - Docs must appear *before* what is being documented
- Example:

```
% @doc This module generates prime numbers using a
%      sieve of Eratosthenes.  The implementation
%      completely abuses the Erlang process model
%      because Donnie thought it would be cool.
%      But it's not.
-module(proc_sieve).
```

  - First sentence is used for brief description in indexes
  - Full text used in detailed documentation for module

Module Documentation (2)

- Can specify many other tags as well, e.g.
  - Author:

    ```
    @author Donnie Pinkston
    @author <a href="mailto:...">Donnie Pinkston</a>
    ```

  - References:

    ```
    @reference <a href="http://...">Sieve of Eratosthenes</a>
    ```

  - Private or hidden modules:
    - **`@private`** excludes module from doc-generation
      - For modules that aren't part of the public interface
      - Can tell **`edoc`** to generate private docs as well
    - **`@hidden`** is even more private than **`@private`** is
      - Will never be included in **`edoc`** output

Function Documentation

- Similar to module documentation
  - Must appear before the function being documented
  - Main documentation tag is **`@doc`**
  - Can also use **`@private`** or **`@hidden`** tags
    - e.g. for exported server functions used by processes
- Function docs include a specification
  - Specification is generally of the form:
    - function_name(ArgName::type(), ...) -> type()
  - edoc makes one up if you don't specify one
    - Probably won't include useful type information
    - If function has multiple clauses, spec will mirror the first clause

Function Specifications

- Can give a function spec using **`@spec`** tag
- Example: **`proc_sieve:generate(MaxN)`**

```
% @doc Generates all prime numbers in the
%       range [2..MaxN] and returns them
%       as a list.
%
% @spec generate(integer()) -> [integer()]
%
generate(MaxN) when MaxN >= 2 -> ...
```

  - Result: HTML docs will contain generate(MaxN::integer()) -> [integer()]
  - **`edoc`** picks up argument name from the code

Function Specifications (2)

- Can specify or override names of arguments, return values with **`Name::`** syntax

```
% @doc Generates all prime numbers in the
%      range [2..MaxN] and returns them as a list.
%
% @spec generate(MaxN::integer()) -> Primes::[integer()]
%
generate(MaxN) when MaxN >= 2 -> ...
```

  - Result: HTML docs will contain generate(MaxN::integer()) -> Primes::[integer()]
- Can also specify lists of values, e.g. [integer()]
- Can also specify tuples, atoms, records, etc.

Specification Types

- Function specifications can use various built-in types
  - atom() – Any atom value
  - bool() – Boolean data type (true or false)
  - integer(), float(), number() – Numeric types
    - number() means “integer or float”
  - string() – a list of characters
  - function()
  - pid()
  - any(), term() – “Any Erlang data type”
  - none() – for functions that don’t return
    - e.g. server process functions

User-Defined Types

- Can also define your own types using `@type` tag

```
% @type xmlElem = #xmlElement{}.
%       An #xmlElement record from the xmerl library.

% @type xmlAttr = #xmlAttribute{}.
%       An #xmlAttribute record from xmerl library.
...
% @type xmlAny = xmlElem() | xmlAttr() | (etc.) .
%       Any XML node type produced by xmerl library.
```

- Types are documented in a "Data Types" section at top of module documentation
- Function specs can also refer to these types

```
% @spec find_xml_attr(atom(), Attributes::[xmlAttr()]) ->
%           xmlAttr() | none
```

User-Defined Types (2)

- Can even specify types/values of record fields

```
% @type rssDoc() = #xmlElement{name=rss,
%                               attributes=[xmlAttr()],
%                               content=[xmlAny()]}.
%       The root xmerl element of an RSS 2.0 feed.

% @type rssItem() = #xmlElement{name=item, (etc.)}.
%       An RSS 2.0 feed item XML fragment.
```

- Can use these types to specify your functions

```
% @spec get_feed_items(rssDoc()) -> [rssItem()]
```

- Generated documentation will contain links for rssDoc() and rssItem()
  - Links go back to “Data Types” section of docs

TODO Tags

- Can put `@todo` tags anywhere in your module documentation

```
% @todo Finish implementing this.
get_feed_items(RSS2Feed) -> [].
```

- Can also specify as `@TODO` or just `TODO:`
- Can optionally include “to-do” items in resulting documentation
  - Specify an option to edoc

XHTML and Wiki Formatting

- Documentation may contain XHTML or Wiki markup
- XHTML markup:
  - HTML tags must have a corresponding closing-tag
  - **`edoc`** uses **`xmerl`** to parse XHTML markup (woo!)
- Wiki markup:
  - A subset of Wiki markup for headings, links to headings, external links, verbatim text, etc.
- Don't need to manually encode paragraphs
  - Even with XHTML, **`edoc`** inserts paragraph breaks when documentation contains a blank line

Running edoc

- edoc can be run from Erlang shell, or from command line
- Erlang shell:
  - `1> edoc:files(["proc_sieve.erl"]).`
  - `ok` *(or info on errors in documentation syntax)*
  - Doc-generator contained in **`edoc`** module
  - **`files/1`** takes a list of strings specifying filenames
  - Generates results into the local directory ☹

Running edoc (2)

- Can also specify a list of config options as second argument
  - Options are of form: **`{name, value}`**
- Example: Generate docs into a directory

```
1> edoc:files(["proc_sieve.erl"], [{dir,"./docs"}]).
```

  - Directory will be created, if necessary
- Example: Include **`@todo`** and private docs

```
1> edoc:files(["proc_sieve.erl"],
              [{dir,"./docs"}, {private,true}, {todo,true}]).
```

  - In the generated docs, private functions and modules have a star next to them

Running edoc (3)

- Can also use Erlang shell to run **edoc** from command-line
  - Shell provides a way to invoke a single module-function, and pass a series of arguments
- Example:

```
erl -noshell -run edoc_run files \
        '["proc_sieve.erl"]' '[{dir,"./docs"}]'
```

  - Calls the **files** function on **edoc_run** module
  - Passes two arguments:
    - A string specifying a list of filenames
    - A string specifying a list of options
    - **edoc_run** converts to Erlang types then invokes **edoc**

More Information

- Much more documentation available on edoc
  - http://www.erlang.org/doc/apps/edoc/index.html
  - See links in top-left area of page

- edoc User's Guide:
  - More "user-friendly" documentation of **`edoc`**
  - Good examples, general details
- edoc Reference Manual:
  - Generated module docs from **`edoc`** sources
  - See **`edoc:files`** and **`edoc:run`** for options, etc.