COMP 348: Principles of Programming Languages

Section 20: Erlang programs

Introduction

- Function syntax
- Basic display functionality
- Function arity
- Pattern matching
- Higher order functions
- Modules
- Importing and exporting functions
- Compiling and running code

Erlang functions

- Functions in Erlang share the basic properties of those in other functional languages.
  - All functions return a value
    - Typically this is a data value but some functions will return the "ok" atom (just like Clojure's print returned nil).
  - For multi-expression function bodies, the return value is that of the last expression.
  - It is possible to define multi-arity functions.
- The basic syntax is as follows:

```
func_name (parm1, parm2, …parmN) ->
   expression_1,
   expression_2,
   ...
   expression_N.
```

Function…cont'd

- Below, we see the trivial `hello_world` function.
- There are no special keywords to introduce a new function.
- Function definitions always end with a "."
- Invoking a function looks a lot like the invocation in imperative languages.
  - Again, don't forget the period.

```
hello_world () ->
   io:fwrite("Hello World\n").

...
hello_world().
```

fwrite

- Note the use of `io:fwrite` in the function body.
  - This is the most basic method for printing to the console.
- fwrite is a function in the `io` module provided by the Erlang framework.
- It has many options for specifying output of various forms.
- But, at its most basic, it works a lot like C's printf function.
- In other words, it uses a format string, including various special substitution characters, followed by zero or more data values.

fwrite…cont'd

- There are a couple of basic things to keep in mind.
  - The substitution character is the tilde (~), as opposed to the "%" in C.
  - In most cases, you can used "~w" to display a value in its "natural" form (int, float, etc)
  - Strings, however, should use the "s" char.
    - Otherwise, you will get ascii values
  - Newlines can either be \n or ~n
  - The list of data values must be delimited by square brackets [ ] (i.e., an Erlang list)
- See a couple of basic examples below.

```
io:fwrite("Hello World\n").
io:fwrite("An int: ~w\n", [43]).
io:fwrite("A float: ~w~n", [77.67]).
io:fwrite("A string: ~s~n", ["boo"]).
io:fwrite("Stuff: ~w, ~w, ~s~n", [3, 4.5, "cat"]).
```

Function arity

- Like Clojure (and other languages), it is possible to create functions with multiple signatures.
- However, the mechanism is a little different.
- As was the case with “assignment”, Erlang uses pattern matching when more than one option is possible.
  - In short, if more than one option is available, it identifies the valid match from the group of possibilities.
  - An error will be generated if there are no matches.

Arity…cont'd

- If we actually have versions of a function with a different number of parameters, we can create multi-arity versions more or less like Clojure.
- An example is given below. Note:
  - Each definition is terminated by a period. In effect, these are completely separate definitions.
  - It is possible for one version to call another.

```
foo(X) -> X * X.
foo(X, Y) -> X * Y.
foo(X, Y, Z) -> foo(X) * foo(Y, Z).
```

Arity..cont’d

- But Erlang can do a little more than this by utilizing its pattern matching mechanism.
- In short, we can have multiple variations of a function, all with the same arity.
  - Erlang will use pattern matching to distinguish between them
- Note that each variation is terminated by a semi-colon (except the last one)
  - This is a single function definition!
- Erlang will select the first variation that matches

```
foo({dog, Name}) -> io:fwrite(“Dog: ~s~n”, [Name]);

foo({cat, Name}) -> io:fwrite(“Cat: ~s~n”, [Name]);

foo({pig, Name, Weight}) ->
  io:fwrite(“Pig: ~s, Weight: ~w ”, [Name, Weight]).
```

Higher order functions

- Like Clojure and other functional languages, functions can be passed as arguments and can be returned as well.
- Again, like Clojure, we often do this with anonymous functions
- In Erlang, we do this with `fun`.
  - We terminate the anonymous definition with `end`
- An example, using Erlang's version of the map function, is given below
  - map can be found in the `lists` module, along with `filter` and `fold` (like Clojure's reduce)
  - You will also find a `foreach` function, that works like a compact loop

```
lists:map( fun(X)->X*2 end, [1, 2, 3, 4]).
; [2,4,6,8]
```

Erlang modules

- Erlang applications are organized into modules.
- In short, the module is defined with the `-module` *attribute*.
- Attributes are prefixed with a "-"
- Modules are searched as per Erlang's Code Path.
  - The default code path consists of the current working directory and Erlang's system library location
- If you want to add directories to the code path, these should be defined in an environment variable called ERL_LIBS
  - This is similar to Java/Clojure's CLASSPATH.

```
-module(foo).
…
foo code
```

Exporting functions

- Functions must be *exported* in order to be call-able in other modules.
- We use the `-export` attribute for this purpose.
- If a function is not specified in the export list, it is considered to be a private function.
- When a function is exported, we must also indicate the arity of the function to be exported.
  - We use a "/" to separate the function name from the arity.

```
-export(baz/1, bar/1, bar/2).
…
Code for baz and bar functions
```

Importing functions

- We can also specify an `-import` attribute.
- Note, however, that the only purpose of import is to eliminate the need to full qualify the function with the module name.
- Without it, the function is still accessible, as long as the module name is specified.
- As such, the import attribute is not that common in Erlang programs.
  - The notable exception would be functions in the lists module, as they are called very frequently.

```
-import(lists, [append/1, map/2]).

map(fun(X)->X*2 end, [1,2, 3,4]). %not qualified

lists:max(MyList). % needs to be qualified
```

Compiling code

- To run Erlang programs from the command line, it is necessary to first compile them into .beam files
  - This is essentially like creating .class files for java.
- The simplest way to do this is with the erlc compiler.
  - Multiple files can be listed at once

```
erlc foo.erl

; will produce a foo.beam bytecode file
```

Running the code

- Now that the .beam files have been created, you can run the code.
- To do this we use the erl emulator, along with a couple of options
  - noshell: this ensures that we don't drop into the interactive Eshell.
  - s or run: this indicates a module/function combination to load and run
    - If command line args are provided, `s` will pass them as atoms (constants), while `run` will pass them as strings
    - If no function name is given, start() is assumed
    - Typically we also use a second "-s init stop" to tell erl to shutdown the Erlang environment once the program is finished

```
erl -noshell -s foo hello -s init stop
; will run the hello function in the foo module

erl -noshell -s foo -s init stop
; will run the start function in the foo module

erl -noshell -run foo bar baz -s init stop
; will run the bar function in foo, passing "baz"
```

A simple example

- This has been a fairly brief tour of the basic syntax of an Erlang program.
- There is a lot more that can be explored, of course, but this is enough to get us going.
- Most of the things that we have discussed can be seen in the sample files listed below:

bar.erl, foo.erl