Intro to Erlang

About Me

- Founder and Organizer of the Dallas/Fort Worth Erlang User Group
- Functional Geekery (**http://www.functionalgeekery.com/**)
- Planet Erlang (**http://www.planeterlang.com/**)

Background

- Created by Joe Armstrong, Robert Virding, and Mike Williams
- Born in Ericsson for telecom switches in 1986
- Open Sourced in 1998

Small Syntax

- Predefined Data Types a.k.a Terms
- Pattern Matching
- Variables (but not really)
- Function Clauses
- Modules

Data Types

- Immutable
- Limited Set

Data Types

- Number
- Atom
- Bit strings and Binaries
- Reference
- Function Identifier
- Port Identifier
- Pid
- List
- Tuple
- Map

Where are my...

Booleans?

Atoms.

- `true`
- `false`

Where are my...

Strings?

Lists of Integers.

```
[72, 101, 108, 108, 111, 32, 87, 111, 114, 108, 100].
% "Hello World"
```

Where are my...

Custom Data Types

Tagged/Named Tuples.

```
{muppet, "Kermit", "frog"}.
% {muppet,"Kermit","frog"}
{muppet, "Fozzie", "bear"}.
% {muppet,"Fozzie","bear"}
```

Where are my...

Custom Data Types (cont)

Records

```
-record(muppet, {name, type}).

Rolf = #muppet{name="Rolf", type="Dog"}.
% #muppet{name = "Rolf",type = "Dog"}

tuple_to_list(Rolf).
% [muppet,"Rolf","Dog"]
```

Variables

- Not Really...
- Can only be bound once
- Start with a capital letter

Variables

```
> Q.
* 1: variable 'Q' is unbound
> Q = 42.
42
> Q.
42
```

Pattern Matching

```
1> 13 = 13.

13

2> a = a.

a

3> A = 13.

13

4> 13 = A.

13
```

Pattern Matching

```
5> A = 15.

** exception error: no match of right hand
     side value 15

6> List = [1, 2, 3].

[1,2,3]
```

Pattern Matching

```
7> [First, y, Third] = [x, y, z].

[x,y,z]

8> First.

x

9> Third.

z
```

Pattern Matching

```
10> SomeList = [Foo, Bar, Baz] = [a, b, c].
[a,b,c]
11> SomeList.
[a,b,c]
```

Pattern Matching

```
12> Foo.
a
13> Bar.
b
14> Baz.
c
```

Pattern Matching

```
15> [Head | Rest] = [1, 2, 3, 4, 5].
[1,2,3,4,5]
16> Head.
1
17> Rest.
[2,3,4,5]
```

Modules

- Used for name-spacing/organization
- Level of code reuse
- Must match with filename
- Declaration is first line of file

Modules

```
-module(markov_chain).
```

Exports

- Preprocessor directive to declare API of a module
- List of function identifiers
- Multiple export declarations are allowed

Exports

```
-export([sum/1]).
```

Exports

```
%% API
-export([start_link/1,
         add_following_word/2,
         pick_next_word_after/1]).

%% gen_server callbacks
-export([init/1,
         handle_call/3,
         handle_cast/2,
         handle_info/2,
         terminate/2,
         code_change/3]).
```

Functions

- Identified by name and arity (number of arguments)
- Allows for multiple function clauses
  - Uses pattern matching to match the clause
  - Order is important, first clause to match wins

my_list.erl

```
-module(my_list).

-export([sum/1]).

sum(List) ->
    sum(List, 0).

sum([], Sum) ->
    Sum;
sum([Head | Rest], Sum) ->
    sum(Rest, Sum + Head).
```

Markov Chain

- **https://en.wikipedia.org/wiki/Markov_chain**
- Random state transitions based off current state and probabilities of next state.
- Similar to iPhone's predictive typing

Markov Chain

Priming the Markov Chain

- Parse out words
- Iterate over words, and create an association between a word and the word following it

Markov Chain

Generating the Markov Chain

- Given a word, we want to pick from the next word by probability of occurance
- Keep picking words until we reach the number of words to generate

Code Time!

- Function to add a word to list of following words
- Function to pick the next word
  - Pick a random word from a list of words

src/markov_generator.erl

```
tokenize(Text) ->
    string:tokens(Text, " \t\n").
```

src/markov_generator.erl

```
parse_text(Text) ->
    [FirstWord | Words] = tokenize(Text),
    load_words(FirstWord, Words).
```

src/markov_generator.erl

```
load_words(_Word, []) ->
    ok;
load_words(Word, [Following | Words]) ->
    markov_word:add_following_word(Word, Following),
    load_words(Following, Words).
```

src/markov_generator.erl

```
add_word_to_list(Words, Word) ->
    [Word | Words].
```

```
pick_next_word(Words) ->
    pick_random(Words).
```

```
pick_random(List) ->
    Length = length(List),
    Index = random:uniform(Length),
    lists:nth(Index, List).
```

OTP

Why OTP?

OTP is a set of libraries for applying lessons learned in building distributed, asynchronous, concurrent applications with high availability

But, First

Actor Model

- No Shared State
- Message Passing
  - Asynchronous Communication

Actor Model

Implemented via Erlang Processes

Processes

- Cheap
- Isolated
- Message Passing
- Links and Monitors
- Garbage Collection

Processes

Cheap

Processes are not

- Platform Process
- Threads

Processes

Cheap

Processes are

- Green threads
- Managed by the Erlang Runtime
- Small
- Quick to create

Processes

Cheap

> *Processes are not threads. Processes are cheap. Ommmmm.*
>
> *-- Martin J. Logan*

Processes

Isolated Processes

- No shared state
- The only state they have access to is their own

Processes

Message Passing

- Mailboxes
- Messages
  - Must provide all the information a process would need
    - Includes "Return addresses"
- No View of outside world (mostly)
  - Process registry
- Asynchronous

Processes

Links and Monitors

- Monitors
  - Monitor status of another process
- Links
  - Process is dependent on another process

Processes

Garbage Collection

Processes live on their own, and don't share state, so easy to reclaim memory when process is no longer being used.

gen_server

- Generic Server
- The base behavior other OTP behaviors are built on
- Takes care of the different concerns you would have to write yourself
  - Handles maintaining state in your processes
  - Allows for synchronous communication on top of asynchronous message passing

gen_server

Implement a behavior and expected set of callbacks

```
-behavior(gen_server).

%% gen_server callbacks
-export([init/1,
         handle_call/3,
         handle_cast/2,
         handle_info/2,
         terminate/2,
         code_change/3]).
```

gen_server

handle_info/2

```
handle_info(Info, State) -> {noreply, State} |
                            {noreply, State, Timeout} |
                            {stop, Reason, State}
```

gen_server

Calling a gen_server Asynchronously

```
cast(ServerRef, Request) -> ok
```

gen_server

Creating a API for you gen_server

- Why?
  - How would your client know your ServerRef?
  - What if you change your Request format?
  - All consumers would need to know format

gen_server

Creating a API for you gen_server

```
-export([add_following_word/1]).


add_following_word(Word, FollowingWord) ->
    WordPid = find_process_for_word(Word),
    gen_server:call(WordPid, {add_following_word, Follo
```

Code Time!

- Generate a reference for a process for a word
- Start a process for a word seen
- Add following word to state of process

Generate a reference for a word

src/markov_word.erl

```
find_process_for_word(Word) ->
    WordKey = get_registered_name_for_word(Word),
    case whereis(WordKey) of
        undefined -> register_word(WordKey);
        Pid when is_pid(Pid) -> Pid
    end.
```

src/markov_word.erl

```
add_following_word(Word, FollowingWord) ->
    WordPid = find_process_for_word(Word),
    gen_server:call(WordPid, {add_following_word, Follo
```

src/markov_word.erl

```
pick_next_word_after(Word) ->
    WordPid = find_process_for_word(Word),
    gen_server:call(WordPid, {pick_next_word}).
```

src/markov_word.erl

```
handle_call({add_following_word, Word}, _From, #state{fo
        NewState = #state{following_words=add_word_to_li
        {reply, ok, NewState};
handle_call({pick_next_word}, _From, State=#state{follow
        Reply = pick_next_word(FollowingWords),
        {reply, Reply, State}.
```

Starting a process for a word

```
start_link(WordKey) when is_atom(WordKey) ->
    gen_server:start_link({local, WordKey}, ?MODULE, []
```

Let it Crash!

- Supervisors

supervisor

- Monitors child processes
- Can supervise other supervisor processes
- Handles restarting of child processes
  - Independent or cascading
- What happens if a child process dies

supervisor

```
-behaviour(supervisor).

%% Supervisor callbacks
-export([init/1]).
```

supervisor

init

```
init(Args) -> {ok, {SupFlags, [ChildSpec]}} |
              ignore |
              {error, Reason}
```

supervisor

init - example

```
init([]) ->
        RestartStrategy = simple_one_for_one,
        MaxRestarts = 1000,
        MaxSecondsBetweenRestarts = 3600,

        SupFlags = {RestartStrategy, MaxRestarts, MaxSe

        Restart = permanent,
        Shutdown = 2000,
        Type = worker,

        Child = {markov_word, {markov_word, start_link,
                               Restart, Shutdown, Type, [mar

        {ok, {SupFlags, [Child]}}.
```

supervisor

restart strategies

- one_for_one
- one_for_all
- rest_for_one
- simple_one_for_one

Code Time!

- Start a process for a word seen
- Add following word to state of process

src/markov_word.erl

```
register_word(Word) ->
    {ok, Pid} = markov_word_sup:start_child(Word),
    Pid.
```

Questions?

Contact me

- @stevenproctor on Twitter
- steven.proctor@gmail.com
- **http://www.proctor-it.com/**

Challenges

- Limit to 140 characters for a tweet
  - Limit to characters or words (tagged tuple)

Challenges

- Read text in given a file name

Challenges

- Add in ability to clear primed text
  - List of process ids to stop nicely?
    - Another Process
    - ETS tables
    - mnesia
  - Supervisor?

Challenges

- Go out there and have fun!!!