What I'm Learning About Learning, Learning About Elixir

Mike Stok — Toronto Perl Mongers — 2016-07-19 https://github.com/mikestok/tpm-2016-07.git

The talk is about learning and Elixir. This was a deliberate experiment to learn following Cameron Price's advice dispensed at EmpEx this year.

Slides & code at https://github.com/mikestok/tpm-2016-07.git



Conventional "Advice"

- Do a "real" project...
- Read *all* the docs...
- Translate something you've already done...
- ... OK if you're not learning something "different"

Risks with this include "writing Fortran in Java": prior experience can affect the way you structure something big, and that can lead to making the new language conform to your existing ways of thinking.

"Real" projects can bring in all kinds of extra libraries, more work in addition to learning the language and its toolchain.

These risks are amplified if you're taking an existing project and replicating it as the tools and libraries can affect the design of something big (especially opinionated frameworks).

Thanks Cameron & Dave

- Do smaller things
- Get a feel for the language on its own terms
- Let its patterns and idioms ingrain themselves

Cameron Price gave a talk at EmpEx (Empire City Elixir), and Dave Doyle used koans to acclimate himself to Ruby coming from a Perl background.

Both people showed me a way to familiarize myself with a new language.

I found it liberating to know that the code I write while learning will *not* end up in part of a system, and I don't have to care about beginner's mistakes and using the wrong thing from the libraries as I discover them.

Smaller steps are particularly helpful when making a big change (compare Perl \rightarrow Ruby, both 00-ish imperative languages with mutable state, to Ruby \rightarrow Elixir where we go to a functional language with no mutable state and deeply integrated support for reliable concurrency).

Baby steps... as they say.

Thanks Cameron & Dave

- https://medium.com/@cameronp/ functional-programming-is-not-weird-youjust-need-some-newpatterns-7a9bf9dc2f77#.d45mro7xx
- http://adventofcode.com
- http://elixirkoans.io

Links to Cameron Price's Medium a

The Recipe I'm Trying

- Do small exercises
 - Try multiple approaches
- Read books & documentation
- Contribute to a project
- ...all intermingled



As Elixir uses BEAM it's easy to call from Elixir to Erlang and vice versa, so we already have access to a lot of Erlang packages. https://hex.pm is the place to look for modules. The "hot" web framework is phoenix http://www.phoenixframework.org/.

Before I Get Going...

"I would like to add a slightly different perspective to functional programming in the Erlang VM: functional programming is not a goal in the Erlang VM. It is a means to an end.

When designing the Erlang language and the Erlang VM, Joe, Mike and Robert did not aim to implement a functional programming language, they wanted a runtime where they could build distributed, fault-tolerant applications. It just happened that the foundation for writing such systems share many of the functional programming principles. And it reflects in both Erlang and Elixir."

José Valim

Erlang Roots

- Uses Erlang's BEAM virtual machine
- Easy to call Erlang and use OTP
- Small core in Erlang...
- ...the rest of Elixir is written in Elixir

Pattern Matching

- What does a = 1 mean?
- \bullet 1 = a
- \bullet [a, b, c] = [4, 0, 17]
- \bullet [a, b, a] = [4, 1, 6]
- [h | t] = [4, 0, 17]
- "foo" <> rest = "foobar"

Elixir tries to make the left side match the right side.

Variable names are bound to the data, the variable isn't a "slot".

Only things on the left of = are bound, and a name can only be bound once in a match.

If left and right hand sides can't be reconciled the match fails. Fatal in code, not so in function definitions...

Types: Lists

- Similar to Lisp
- [1,2,3] is [1|[2|[3|[]]]]
- Cheap to add to head O(1)
- Expensive to add to tail O(n)
- ... this affects how we think of things!

... but measure performance, the Big O notation

Types: Tuples, Maps, and Structs*

- Tuple: {"a", 10, hello}
- Map: $%\{:f => "x", "b" => 10\}$
- Struct: %Foo{bar: "baz"}
 - Structs are based on maps

Types: Odds and Ends

- String "Hello"
- List 'Hello'
- Atom:foo
- Numbers

Functions and Arity

- For example: in the Enum module the function sum called with a single enumerable argument would be called Enum.sum/1
- 10 = Enum.sum([1, 2, 3, 4])

Functions and Arity

• Example.func/1 & Example.func/2 are *different* functions:

```
defmodule Example do
    # Example.func/1
    def func(arg), do: func(arg, 0)

# Example.func/2
    def func(arg, n) do
        IO.puts("Called with #{arg} and #{n}")
        end
end
```

Artity and pattern matching are the tools for deciding which function to call. You'll see a lot of the Module.function/arity in the Elixir docs.

We'll see an example of using both arity and simple pattern matching in the the Private "Helpers" slide.

Functions and Privacy

- All functions are defined in modules, there's no default or top level module.
- def
- defp
- that's it!

In Elixir the function's documentation is compiled into the byte code, this is only done for visible functions.

I don't think there's a way to get around

```
fact(0) 1
fact(n) n * fact(n - 1)
```

```
fact(0) 1
fact(n) n * fact(n - 1)
```

```
def fact(0), do: 1
def fact(n), do: n * fact(n - 1)
```

```
defmodule Example do
  def fact(0), do: 1
  def fact(n), do: n * fact(n - 1)
end
```

```
defmodule Example do
  def fact(0), do: 1
  def fact(n)
    n * fact(n - 1)
  end
end
```

do

```
defmodule Example do
  def fact(0), do: 1
  def fact(n) when is_integer(n) and n > 0 do
    n * fact(n - 1)
  end
end
```

Recursion (not looping) defmodule Example do @doc """ Calculate the factorial of a number. iex> Example.fact(10) 3628800 """ @spec fact(non_neg_integer) :: pos_integer def fact(0), do: 1 def fact(n) when is_integer(n) and n > 0 do n * fact(n - 1) end end

The documentation is compiled in, so if you're in IEx you can say:

h Example.fact

and see the docs.

The @spec notation makes explicit the contract for the function which can be inferred by the dialyzer tool (a static analysis tool).

Specs show up in generated html documentation.

Private "Helpers" defmodule Example do @doc """ Calculate the factorial of a number using tail recursion. iex> Example.fact(10) 3628800 """ @spec fact(non_neg_integer) :: pos_integer def fact(n) when is_integer(n) and n >= 0 do fact(n, 1) end defp fact(0, acc), do: acc defp fact(n, acc), do: fact(n - 1, n * acc) end

Common pattern is to use recursion with an accumulator and private "helper" functions.

The arity lets us know which of the fact functions we're calling, and users of the module can't get at Example.fact/2.

Syntax Sugar: |>

- func(arg1, arg2) can be written as arg1 |> func(arg2)
- Idea from F#
- Useful for avoiding lots of irritating single parentheses!

This leads for the first argument to a function being its subject.

Important because we want to compose little functions into bigger units, and this can help avoid deep calls or lots of intermediate variables. To me it makes the code read better...

Syntax Sugar: | >

```
result =
  last_func(
    third_func(
        second_func(
        first_func(value, arg1),
        arg2),
        arg3, arg4),
        arg5)
```

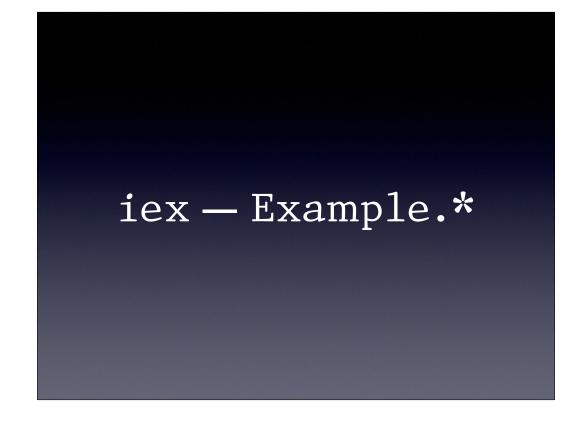
Syntax Sugar: |>

```
result =
  value
  |> first_func(arg1)
  |> second_func(arg2)
  |> third_func(arg3, arg4)
  |> last_func(arg5)
```

Streams

```
defmodule Example do
    @doc """
    Calculate the *n*th Fibonacci number.

        iex> Example.fib(20)
        6765
"""
    @spec fib(non_neg_integer) :: non_neg_integer
    def fib(n) when is_integer(n) and n >= 0 do
        [0, 1]
        |> Stream.iterate(fn([a, b]) -> [b, a + b] end)
        |> Stream.drop(n)
        |> Enum.take(1)
        |> List.flatten
        |> List.first
        end
end
```



This is to show iex and some of the examples from the code. Simple REPL and can run code iex example.exs (exs = elixir script, compiled in memory not to file...)

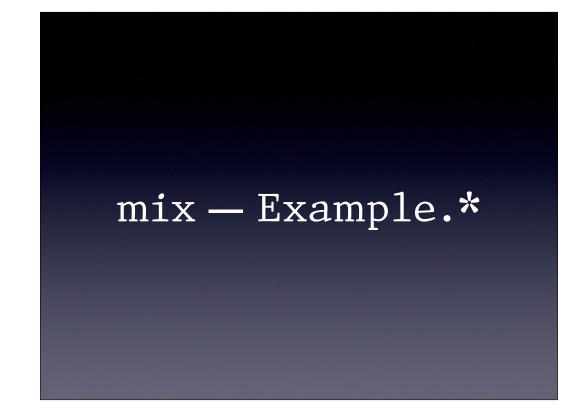
:timer.tc(Example, :fact, [100_000])
:observer.start

mix Tool

• Mix is a build tool that provides tasks for creating, compiling, and testing Elixir projects, managing its dependencies, and more (e.g. escript).

http://hex.pm is the home of the hex package manager for Elixir and Erlang.

As Elixir compiles down to BEAM byte-code escript is particularly useful for packaging an application so it can be installed on any machine with Erlang installed. mix.exs is used to specify the module whose main function is to be called when the escript is executed (used in look and say later...)



A quick tour of mix -

mix test mix dialyzer mix docs

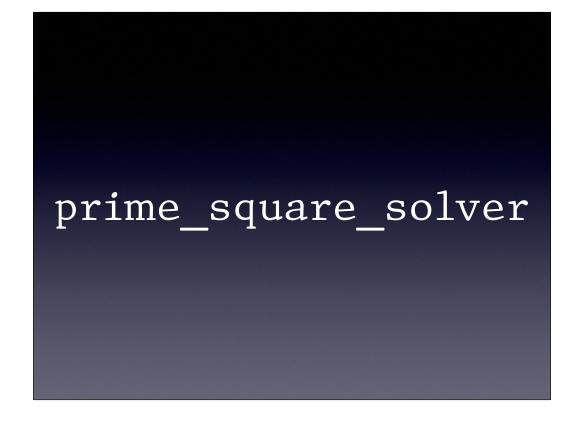
add credo



Speed of elixir vs perl5 (the problem is pretty much what perl is aimed optimized for, elixir really shines when you have multiple processes and use the language and OTP... right tool for the job etc.)

Note the executables made by mix escript

Evolution of elixir version, work through elixir2 look_and_say('112') if there's interest (interaction between destructuring, matching, and dealing with reversed lists)



Elixir vs perl6 - I needed to add Square and Math modules. Code is more verbose in elixir but much faster. Naïve elixir though.

Standard Library

- Good modules to start with:
 - <u>List</u>
 - Enum
 - <u>Stream</u>
 - <u>Regex</u>



Learning About Learning

- For me reading, simple exercises, and contributing to a project are complementary.
- Elixir in Action, advent of code exercises, and credo are the ones I'm using.
- The credo author is mentoring me and helping me make a contribution to the community.

Resources (Elixir)

- http://elixir-lang.org
- http://hex.pm
- http://blog.plataformatec.com.br/2016/05/beyond-functional-programming-with-elixir-and-erlang/
- <u>Programming Elixir</u> (Pragmatic Programmers book)
- Elixir in Action (Manning book)

Resources (Exercises)

- http://adventofcode.com
- http://exercism.io
- http://elixirkoans.io
- See Cameron Pierce's article https://medium.com/@cameronp/functional-programming-is-not-weird-you-just-need-some-new-patterns-7a9bf9dc2f77#.h2kck1xjb