

# Cómo conocí a la metaprogramación (y sobreviví a ello)

Miguel E. Ruiz

29 de febrero de 2024

# Contenidos

- 1 Introducción
- 2 Metaprogramación
- 3 Mi TFM: LogicElixir

# IO.puts("Hello, World!")

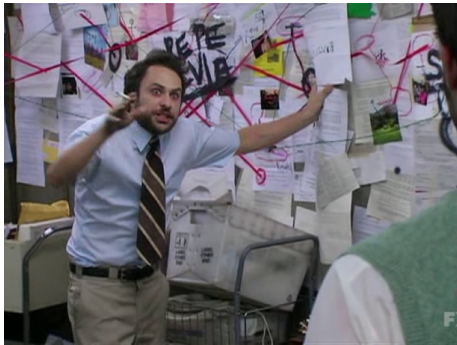


- Miguel [Emilio]
- Ingeniero Informático
- Intereses: Programación funcional y concurrente
- Erlang y Elixir

# La metaprogramación y yo

- Al acabar el Grado (2018), interés en Elixir
  - Ejercicios en Codewars
  - Proyecto Pokedex
- Interés en el libro *Metaprogramming Elixir*
- Sin embargo...
  - Falta de tiempo (y de motivación)
- Arranca el máster (2021) y...
  - recupero el interés en la metaprogramación
  - posibilidad de hacer el TFM sobre ello

# Una cosa os prometo...



No acabaremos así (o al menos, no mucho. . .)

# Finalizadas las presentaciones...



¿KLK?

# Metaprogramación

Definicion(es)

## Elixir School

*Metaprogramming is the process of using code to write code.*



# Metaprogramación

Definición(es)

## Elixir School

*Metaprogramming is the process of using code to write code.*

## Rust Web Programming

*Metaprogramming can generally be described as a way in which the program can manipulate itself based on certain instructions.*

# Metaprogramación

Definicion(es)

## Elixir School

*Metaprogramming is the process of using code to write code.*

## Rust Web Programming

*Metaprogramming can generally be described as a way in which the program can manipulate itself based on certain instructions.*

## ChatGPT

*Metaprogramming is a programming technique where a program can manipulate its own structure or behavior at runtime.*

# Metaprogramación

## Técnicas

- Metaclases (Ruby, Python)
- Macros (Scala, Rust)
- Templates (C, C++)

# Metaprogramación en Elixir

## Mi definición

- Metaprogramación: mecanismo para generar programas a partir de otros programas (en tiempo de compilación).
- Elixir tiene la capacidad de tratar los programas como un tipo de dato y modificarlos o transformarlos.
- Ejemplos de uso:
  - Extensión del lenguaje a través de DSL.
  - Creación de nuevas librerías como *Phoenix* y *Ecto*.

# Metaprogramación en Elixir

## Ejemplos

```
defmodule Sample.Weather do
  use Ecto.Schema
  schema "weather" do
    field :city      # String by default
    field :temp_lo, :integer
    field :temp_hi, :integer
    field :prcp,     :float, default: 0.0
  end
end
```

# Metaprogramación en Elixir

## Ejemplos

```
defmodule MyAppWeb.Router do
  use Phoenix.Router

  get "/login", LoginController, :show

  scope "/" do
    pipe_through [:browser]
    get "/posts", PostController, :index
    get "/posts/:id", PostController, :show
  end
end
```

# Metaprogramación en Elixir

## Ejemplos

```
defmodule Calculator do
  use ExActor.GenServer

  defstart start_link, do: initial_state(0)

  defcast inc(x), state: state, do: new_state(state + x)
  defcast dec(x), state: state, do: new_state(state - x)

  defcall get, state: state, do: reply(state)

  defcast stop, do: stop_server(:normal)
end
```

# Metaprogramación en Elixir

¿Cómo?

- A través de macros, las cuales permiten **extender** el lenguaje

¿Cuántas palabras reservadas hay en Elixir?

- 8
- 15
- 37
- ninguna



# Metaprogramación en Elixir

¿Cómo?

- A través de macros, las cuales permiten **expandir** el lenguaje

¿Cuántas palabras reservadas hay en Elixir?<sup>1</sup>

- 8
- 15
- 37
- ninguna

true, false, nil, when, and, or, not, in  
fn, do, end, catch, rescue, after, else

<sup>1</sup><https://hexdocs.pm/elixir/1.16/syntax-reference.html#reserved-words>

# AST (de Elixir)

- Abstract Syntax Tree
- La estructura interna del código Elixir
- Se representa mediante una tupla de 3 elementos:
  - Un átomo que representa la función invocada u otra tupla que representa un nodo interno del árbol.
  - Una lista de términos de metadatos de la función invocada.
  - Una lista de argumentos de la función del primer argumento.
- El tipo `Macro.t()` representa el AST de Elixir

# AST de Elixir

Para obtener el AST de una expresión de Elixir, se utiliza la función `quote/2`:

```
iex> quote do: sum(1, 2, 3)
{:sum, [], [1, 2, 3]}
```

Si se requiere introducir valores dentro de un AST, se dispone de la función `unquote/1`:

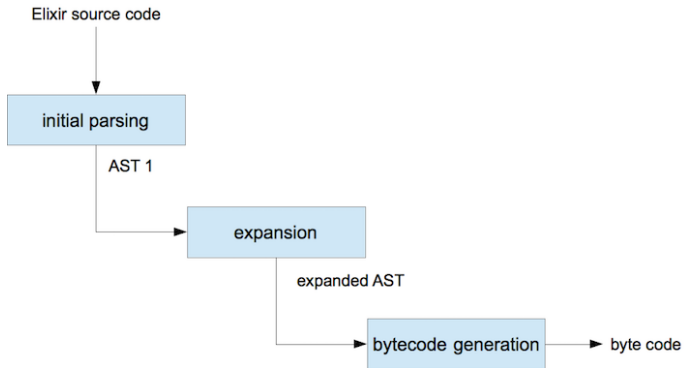
```
iex> a = 3
iex> quote do: sum(1, 2, a)
{:sum, [], [1, 2, {:a, [], Elixir}]}
iex> quote do: sum(1, 2, unquote(a))
{:sum, [], [1, 2, 3]}
```

# DEMO

(Desmembrando ASTs)

# Metaprogramación en Elixir

## Proceso de compilación



# Metaprogramación en Elixir

¿Cómo definimos macros?

- Con el operador `defmacro` podremos definir nuestras macros, respetando las reglas<sup>2</sup>:
  - Don't Write macros (WTF?)
  - Use Macros Gratuitously
- El operador **tiene que** devolver un AST

---

<sup>2</sup>Metaprogramming Elixir: Macro Rules

# Definición de macros

## Ejemplo básico

```
defmodule ControlFlow do
  defmacro unless(expression, do: block) do
    quote do
      if !unquote(expression), do: unquote(block)
    end
  end
end
```

# DEMO

## Un DSL para definir mascotas

```
defmodule Template do
  use Pets

  pet "Bucky" do
    species :dog
    hobbies ["sniffing", "eating birds"]
  end
  pet "Gardfield" do
    species :cat
    hobbies ["hating mondays"]
  end
end
```

```
iex> Bucky.greet
"Hello my name is Bucky and I'm a dog"
iex> Gardfield.hobbies
"Gardfield's hobbie is hating mondays"
```



Sobreutilizar esta técnica podría llevarnos a algunos de estos casos<sup>3</sup>:

- *Large code generation*: Cuando la macro genera demasiado código
- *Unnecessary macros*: ¿Estás seguro que este código debe ser una macro?
- *use instead of import*: Propagación de dependencias

---

<sup>3</sup><https://hexdocs.pm/elixir/macro-anti-patterns.html> ( $\geq 1.16$ )

# Mi TFM: LogicElixir

## Motivación

- La mayor de todas: aprender sobre metaprogramación (en Elixir).
- Hacer mi primer proyecto (serio) en Elixir.
- Acabar el Máster :D

# LogicElixir

## Objetivo principal

- Desarrollar una librería para Elixir que permita a un programador especificar un programa lógico
- Proporcionar mecanismos para realizar consultas sobre ese programa lógico

# LogicElixir: Prerrequisitos

## Programación Lógica

*% Hechos*

```
human(fry).
```

```
mutant(leela).
```

```
likes(fry, pizza).
```

*% Reglas*

```
pizza_lover(X):- likes(X, pizza).
```

```
member(X, [X | _]).
```

```
member(X, [_ | T]):- member(X, T).
```

# LogicElixir

## Ejemplo

```
defmodule LogicModule do
  use LogicElixir
  defpred human(:fry)
  defpred mutant(:leela)
  defpred likes(:fry, :pizza)
  defpred pizza_lover(X), do: likes(X, :pizza)
  defpred member(X, [X | T])
  defpred member(X, [H | T]) do
    member(X, T)
  end
end
```

### Términos lógicos

$T$	$::=$	$lit$	<i>literales</i>
		$X$	<i>variables lógicas</i>
		$[]$	
		$[T_1 \mid T_2]$	
		$\{T_1, T_2, \dots, T_n\}$	
		$f(T_1, T_2, \dots, T_n)$	<i>llamadas a funciones de Elixir</i>

### Patrones

$$\begin{array}{lcl} P & ::= & lit \\ & | & X \\ & | & [] \\ & | & [P_1 \mid P_2] \\ & | & \{P_1, P_2, \dots, P_n\} \end{array} \quad \text{variables lógicas}$$

### Objetivos y secuencias de objetivos

$G$	$::=$	$\epsilon$	
		$g, G$	
$g$	$::=$	$p(T_1, T_2, \dots, T_n)$	<i>predicados lógicos</i>
		$T_1 = T_2$	<i>unificación de términos</i>
		$G_1 ; \dots ; G_n$	<i>disyunción de secuencias</i>
		$@(e)$	<i>expresiones de Elixir</i>



### Reglas

$$R ::= p(P_1, \dots, P_n) [: - G]$$

### Definiciones de predicados

$$D = \left[ \begin{array}{cc} p(P_{1,1}, \dots, P_{1,n}) & [: - G_1] \\ p(P_{2,1}, \dots, P_{2,n}) & [: - G_2] \\ \vdots & \\ p(P_{m,1}, \dots, P_{m,n}) & [: - G_m] \end{array} \right]$$

# Lenguaje Core

- El lenguaje Core es el lenguaje de bajo nivel que servirá de nexo entre LogicElixir y Elixir.
- Una función Core agrupa todos los hechos y reglas de un predicado `defpred` en una única función `defcore`.
- Posee una sintaxis parecida a LogicElixir, pero con matices:
  - No existen los patrones.
  - Ausencia de sintaxis azucarada en las listas.
  - Los predicados tienen que tener un cuerpo de función.

# Traducción de LogicElixir a Core

Se considera una definición de predicado  $D$ :

$$D = \left[ \begin{array}{ccc} p(P_{1,1}, \dots, P_{1,n}) & : - & G_1 \\ p(P_{2,1}, \dots, P_{2,n}) & : - & G_2 \\ \vdots & & \\ p(P_{m,1}, \dots, P_{m,n}) & : - & G_m \end{array} \right]$$

# Traducción de LogicElixir a Core

*generate\_defcore*(*D*) =

```
defcore  $p(X_1, \dots, X_n)$  do  
  choice do  
     $P_{1,1} = X_1$   
    ...  
     $P_{1,n} = X_n$   
     $G_1$   
  else  
    ...  
  else  
     $P_{m,1} = X_1$   
    ...  
     $P_{m,n} = X_n$   
     $G_m$   
  end  
end
```

# Traducción de LogicElixir a Core

## Ejemplos

```
defpred planet(:tatooine)
defpred planet(:hoth)
defpred planet(:dagobah)
```

```
defcore planet(X1) do
  choice do
    :tatooine = X1
  else
    :hoth = X1
  else
    :dagobah = X1
  end
end
```

# Traducción de LogicElixir a Core

## Ejemplos

```
defpred member(X, [X | T])
defpred member(X, [H | T]) do
  member(X, T)
end
```

```
defcore member(X1, X2) do
  choice do
    X = X1
    [X | T] = X2
  else
    X = X1
    [H | T] = X2
    member(X, T)
  end
end
```

# Implementación

logic\_elixir.ex

¿Qué supone la expresión `use LogicElixir`?

Se inyecta un AST a través de la función `__using__/1`:

```
defmodule LogicElixir do
  defmacro __using__(_params) do
    quote do
      use LogicElixir.Defpred
      use LogicElixir.Findall
    end
  end
end
```

# Implementación

logic\_elixir.ex

¿Qué supone la expresión `use LogicElixir`?

Se inyecta un AST a través de la función `__using__/1`:

```
defmodule LogicElixir do
  defmacro __using__(_params) do
    quote do
      use LogicElixir.Defpred # <= Dependency injection
      use LogicElixir.Findall
    end
  end
end
```



# Implementación

defpred.ex

```
defmodule LogicElixir.Defpred do
  defmacro defpred(head) do
    {pred_name, args_ast} = Macro.decompose_call(head)
    Module.put_attribute(__CALLER__.module, :definitions,
                        {pred_name, args_ast})
  end

  defmacro defpred(head, do: block) do
    {pred_name, args_ast} = Macro.decompose_call(head)
    Module.put_attribute(__CALLER__.module, :definitions,
                        {pred_name, {args_ast, block}})
  end
end
```

# Implementación

defpred.ex

```
defmacro __before_compile__(env) do
  VarBuilder.start_link()
  definitions = Module.get_attribute(env.module, :definitions)
    |>
    Enum.reverse()
  grouped_defs = definitions |>
    Enum.group_by(&elem(&1, 0), &elem(&1, 1))
  for {name, args} <- grouped_defs do
    generate_defcore(name, args)
  end
end
```

# Traducción de Core a función de Elixir

- El paso final es convertir la definición Core intermedia a código de Elixir.
- Para ello, es necesario definir una serie de conjuntos, así como una gramática BNF de estos términos disponibles en tiempo de ejecución.

# Traducción de Core a función de Elixir

## Predicado Core a función Elixir

$tr\_def(p(X_1, \dots, X_n): -G) =$

```
def p( $t_1, \dots, t_n$ ) do
   $x_1 = \text{VarBuilder.gen\_var}()$ 
   $\vdots$ 
   $x_n = \text{VarBuilder.gen\_var}()$ 
   $y_1 = \text{VarBuilder.gen\_var}()$ 
   $\vdots$ 
   $y_m = \text{VarBuilder.gen\_var}()$ 
  fn  $th_1 \rightarrow$ 
     $th_2 = \text{Map.merge}(th_1,$ 
       $\text{Map.new}([\{x_1, t_1\}, \dots, \{x_n, t_n\}]))$ 
    ( $tr\_goals(\Delta, G)$ ).( $th_2$ )
    |>
     $\text{Stream.map}(\&\text{Map.drop}(\&1, [$ 
       $x_1, \dots, x_n, y_1, \dots, y_m]))$ 
  end
end
```

# Consulta de predicados

- LogicElixir ofrece un mecanismo de consulta sobre los predicados que haya declarado el usuario en su proyecto.
- La macro `findall` sirve de interfaz entre LogicElixir y Elixir.

# Consulta de predicados

## Lenguaje

$$F ::= \text{findall } T \text{ [into } E \text{ ] do } G$$

Donde:

- $T$  es un término LogicElixir.
- $G$  es una secuencia de objetivos LogicElixir.
- $E$  es un término opcional que implementa el protocolo Enumerable.

# DEMO

(Probando mi TFM)

# Dificultades encontradas

- Falta de (buena) documentación
- Falta de código de apoyo (o desactualizado)
- Contramedidas: test unitarios



# Conclusiones

- La metaprogramación es una área bonita (y MUY compleja)
- Seguramente nunca tengas que usar esta técnica
- No está de más conocer qué hace la “magia”
- Los tests son tus amigos

# Bibliografía



Chris McCord (2015)

Metaprogramming Elixir



José Valim

Kernel.SpecialForms, Macro, Module



Leon Sterling and Ehud Shapiro (1986)

The Art of Prolog



[https://github.com/MiguelERuiz/logic\\_elixir](https://github.com/MiguelERuiz/logic_elixir)

Repo de la librería LogicElixir



[https://github.com/MiguelERuiz/logic\\_elixir\\_examples](https://github.com/MiguelERuiz/logic_elixir_examples)

Repo de uso de la librería LogicElixir

# Otros recursos

-  <https://chrismccord.com/blog/2014/03/13/write-less-do-more-and-have-fun-with-elixir-macros/>  
Charla de Chris McCord en la ElixirConf (2014)
-  [https://www.theerlangelist.com/article/macros\\_1](https://www.theerlangelist.com/article/macros_1)  
Blog de Sasa Juric sobre Macros (2014)
-  <https://github.com/MiguelERuiz/pets>  
Repo de Pets
-  <https://github.com/MiguelERuiz/presentations>  
Repo de Presentaciones