

# 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

# Finalizadas las presentaciones...



¿KLK?

# Metaprogramación

Definicion(es)

## Elixir School

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

# 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.*



# 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

¿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

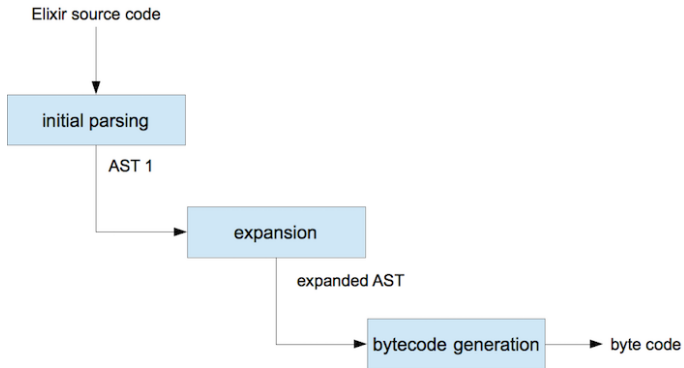
¿Cuántas palabras reservadas hay en Elixir?

- 8
- 15
- 37
- ninguna

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

# Metaprogramación en Elixir

## Proceso de compilación



# Metaprogramación en Elixir

- Con el operador `defmacro` podremos definir nuestras macros, respetando las reglas<sup>1</sup>:
  - Don't Write macros (WTF?)
  - Use Macros Gratuitously

---

<sup>1</sup>Metaprogramming Elixir: Macro Rules

# 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
```

# 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]}
```

# Módulos y atributos

- Los módulos de Elixir pueden registrar atributos para:
  - definición de constantes
  - estructuras de almacenamiento temporales

Traer ejemplo del TFM del `Module.register_attribute`, etc

# Cosas para hablar

- Hooks using `y` `before_compile`
- Formas especiales `__MODULE__` y `__ENV__` y `__CALLER__`
- `Macro.unique_var` vs `Macro.var`
- `Macro.escape`

# 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 Example do
  use Pets

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

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



# Antipatrones

Cosas que mirar en <https://hexdocs.pm/elixir/macro-anti-patterns.html>

# Mi TFM: LogicElixir

## Motivación

- La mayor de todas: aprender sobre metaprogramación (en Elixir).
- Hacer mi primero 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 del conjunto <math>Var</math></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

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

### 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

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

# 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.
- Los términos de este lenguaje estarán contenidos dentro del conjunto `Term`.

# 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

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

## Definición de conjuntos

Además de Var y Term, se consideran los conjuntos:

- EVar, el conjunto de variables de Elixir.
- Subst, el conjunto de sustituciones.
- Expr, el conjunto de expresiones de Elixir.
- Eval, el conjunto de valores de Elixir disponibles en tiempo de ejecución.
- RTerm, el conjunto de términos disponibles en tiempo de ejecución.

# Traducción de Core a función de Elixir

## Términos *ground*

- Para determinar en tiempo  $O(1)$  si un término es *ground* o no, utilizamos tuplas  $\{:\text{ground}, \_ \}$  en nuestra representación de los elementos `RTerm`.
- Para ello, han de cumplirse los siguientes invariantes:
  - Si está representado de la forma  $\{:\text{ground}, R\}$ , entonces no contiene variables.
  - Si no está representado de la forma  $\{:\text{ground}, R\}$ , entonces ha de contener alguna variable.

# Traducción de Core a función de Elixir

## Estructura de datos

Se manejarán algunas estructuras de datos para facilitar la traducción:

- $\Delta$ : Relaciona los términos del conjunto Var con los términos del conjunto EVar.
- $\theta$ : Relaciona los términos del conjunto Var con los términos del conjunto RTerm.

$$\Delta = [X_1 \rightarrow x_1, X_2 \rightarrow x_2, \dots, X_n \rightarrow x_n]$$
$$\theta = [X_1 \rightarrow R_1, X_2 \rightarrow R_2, \dots, X_n \rightarrow R_n]$$

# Traducción de Core a función de Elixir

## Funciones auxiliares

Se definen funciones auxiliares para facilitar la traducción y la reducción de términos:

- `build_list/2`
- `build_tuple/1`
- `groundify/2`

# Traducción de Core a función de Elixir

## Términos

Los términos Core serán traducidos por la función `tr_term/3`:

$$\begin{aligned} tr\_term(\Delta, x, lit) &= \{ :ground, lit \} \\ tr\_term(\Delta, x, X) &= \{ :var, \Delta(X) \} \\ tr\_term(\Delta, x, [T_1 \mid T_2]) &= build\_list(tr\_term(\Delta, x, T_1), tr\_term(\Delta, x, T_2)) \\ tr\_term(\Delta, x, \{T_1, \dots, T_n\}) &= build\_tuple([tr\_term(\Delta, x, T_1), \dots, tr\_term(\Delta, x, T_n)]) \\ tr\_term(\Delta, x, f(T_1, \dots, T_n)) &= \left[ \begin{array}{l} x_1 = groundify(x, tr\_term(\Delta, x, T_1)) \\ \vdots \\ x_n = groundify(x, tr\_term(\Delta, x, T_n)) \\ \{ :ground, f(x_1, \dots, x_n) \} \end{array} \right] \end{aligned}$$

# Traducción de Core a función de Elixir

## Objetivos y secuencias de objetivos (I)

$$tr\_goals(\Delta, \epsilon) = \text{fn } th \rightarrow [th] \text{ end}$$
$$tr\_goals(\Delta, (g, G)) = \left[ \begin{array}{l} \text{fn } th_1 \rightarrow \\ \quad (tr\_goal(\Delta, g)).(th_1) \\ \quad |> \text{Stream.flat\_map}(\text{fn } th_2 \rightarrow \\ \quad \quad (tr\_goals(\Delta, G)).(th_2) \\ \quad \quad \text{end}) \\ \text{end} \end{array} \right]$$



# Traducción de Core a función de Elixir

## Objetivos y secuencias de objetivos (II)

$$\begin{aligned} tr\_goal(\Delta, p(T_1, \dots, T_n)) &= \left[ \begin{array}{l} \text{fn } th \rightarrow \\ \quad p(tr\_term(\Delta, th, T_1), \dots, tr\_term(\Delta, th, T_n)).(th) \\ \text{end} \end{array} \right] \\ tr\_goal(\Delta, T_1 = T_2) &= \left[ \begin{array}{l} \text{fn } th \rightarrow \\ \quad \text{unify\_gen}(th, tr\_term(\Delta, th, T_1), tr\_term(\Delta, th, T_2)) \\ \text{end} \end{array} \right] \\ tr\_goal(\Delta, G_1; \dots; G_n) &= \left[ \begin{array}{l} \text{fn } th \rightarrow [tr\_goals(\Delta, G_1), \dots, tr\_goals(\Delta, G_n)] \\ \quad |> \text{Stream.flat\_map}(\text{fn } f \rightarrow f.(th) \text{ end}) \\ \text{end} \end{array} \right] \\ tr\_goal(\Delta, @(T)) &= \left[ \begin{array}{l} \text{fn } th \rightarrow \\ \quad \text{check\_b}(th, \text{groundify}(th, tr\_term(\Delta, th, T))) \\ \text{end} \end{array} \right] \end{aligned}$$

# 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
   $x_1 = \text{VarBuilder.gen\_var}()$ 
  :
  :
   $x_n = \text{VarBuilder.gen\_var}()$ 
   $y_1 = \text{VarBuilder.gen\_var}()$ 
  :
  :
   $y_m = \text{VarBuilder.gen\_var}()$ 
  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.

# Consulta de predicados

## Traducción

$tr\_findall(T, G, E \setminus \setminus nil) =$

```
x1 = VarBuilder.gen_var()
...
xn = VarBuilder.gen_var()
solutions = (tr_goals(Δ, G)).(%{})
  |> Stream.map(fn sol →
    t = tr_term(Δ, sol, T)
    groundify(sol, t)
  end)
case E do
  nil → solutions
  Z   → solutions |> Enum.into(Z)
end
```

# Dificultades encontradas

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

# Conclusiones

- La metaprogramación es una area bonita (y MUY compleja)
- Seguramente nunca tengas que usar esta técnica
- Nunca está de más conocer qué hace la “magia”
- Los tests son tus amigos

# Bibliografía

- El libro de Metaprogramming Elixir
- La web de Sasa Juric
- La docu de SpecialForms, Macro y Module
- The art of prolog