

Métaprogrammation en Elixir

Une introduction naïve

Xavier Van de Woestyne

`xaviervdw@gmail.com` - *margo.com*

Paris.ex 12 - Mai 2019

Bonjour/Bonsoir !

Xavier Van de Woestyne

- Bruxelles, Lille, Paris
- *Data Engineer* chez Margo Bank

Réseaux

- <https://xvw.github.io>
- *vdwxv* sur Twitter
- *xvw* sur Github/Gitlab
- *xvw@merveilles.town* sur Mastodon

OCaml, F#, Erlang/Elixir, Kotlin, Io, Ruby, Elm, Racket

LilleFP

- Meetup régulier
- Langages applicatifs
- Programmation fonctionnelle
- Systèmes de types
- Fusion de LilleFP, Lille Elixir, Lille Rust et LilleSUG
- **On recrute des speakers !**

Après presque 10 ans de Erlang

J'appartiens aux gens qui n'ont pas été particulièrement emballé par Elixir

- Je préférais la syntaxe de **Erlang** (amoureux de **Prolog**) ;
- j'étais inquiet de la transmissions d'idiomes (Ruby) ;
- donc je trouvais que Elixir ne servait à rien (à tort...).

Ce qui m'a fait changé d'avis sur Elixir

- **Phoenix** (plus spécifiquement, **Ecto** et **Plug**) ;
- l'unification des types pour les chaînes de caractères ;
- les mécanismes de méta-programmation ;
- **mix** ;
- et plus tard : `gen_stage` et `Flow`.

Sommaire et objectifs

- Comprendre ce qu'est *rapidement* la méta-programmation ;
- survoler les perspectives qu'Elixir offre pour *méta-programmer* ;
- observer quelques cas d'usages aux **macros** ;
- conclure sur les usages et avantages.

Ceci est introduction naïve ... pour les non-initiés qui, comme le temps est mon ennemi, n'évoquera malheureusement pas tout ce qu'il est possible de dire sur la méta-programmation en Elixir :'(

- Ceci n'est pas du tout une présentation technique ! (Désolé...)

Meta-programmation

*L'écriture de programme qui manipulent des données **décrivant** elles-mêmes des programmes. (d'où le "meta")*

Meta-programmation

On peut méta-programmer de plusieurs manières

- Via la réflexion (implicite ou explicite) ;
- via des raccourcis du langage ;
- via des générateurs de code (par exemple les affreux templates de C++) ;
- via des étapes de compilation (multi-staged metaprogramming) ;
- via des macros ;
- en utilisant un Lisp (aha !)

A quoi ça sert ?

- Limiter le **boilerplate** ;
- limiter le **boilerplate** ;
- limiter le **boilerplate** ;
- étendre le langage (eDSL) ;
- déléguer à une machine des tâches rémanantes.

Elixir offre plusieurs outils de méta-programmation !

Introspection via `Module.__info__(subject)`

```
iex> Map.__info__(:functions)
[delete: 2, drop: 2, ... values: 1]
```

Dispatch dynamique

```
def call(module) do
  apply(module, function, [args])
end
```

Génération de code et dérivation d'interfaces

- **Génération:** Behaviour
- **Dérivation:** Protocol

Ce sont les manières rapides (et un peu cheap) de faire de la meta-programmation.

Pré-processeur et macros

On entre enfin dans le vif du sujet !

Une manière de **transformer** une unité exécutable **avant** son exécution

Pré-processeur et macros

Par exemple

- `#include <stdlib.h>` et `#define AGE 29`
- Les préprocesseurs CSS (Less, Sass, etc.)
- Les postprocesseurs CSS (aha)
- `cat myProgram.ex | sed ... | awk ... > myNewProgram.ex`

Cependant, tout le monde n'agit pas comme des sauvages

- **Lisp** introduit la capacité de manipuler des **quasi-quotations**
- elles permettent de manipuler des termes du langage ... comme des termes du langage
- c'est en partie possible parce que la grammaire du langage est très simple
- le langage peut être étendu (parfois ... de manière suréaliste, **Racket**).

Processus de compilation d'un programme Elixir

La grammaire d'Elixir étant plus complexe, on manipule son **AST**.



Figure 1: Processus de compilation

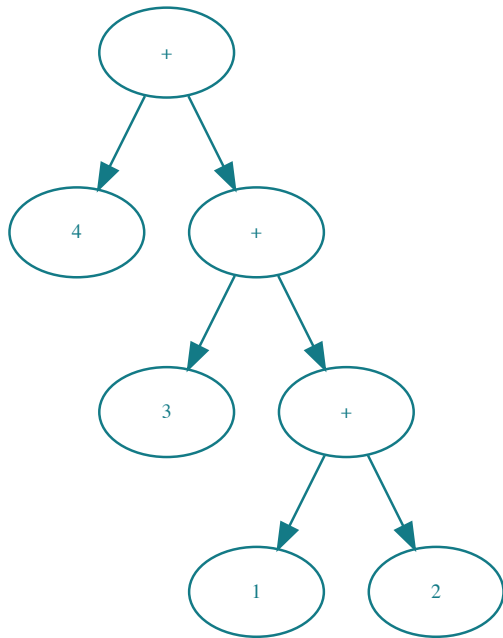
- Il sera possible d'injecter des fragments d'AST avant la phase **d'expansion**
- ces fragments sont des **macros**

Les macros (enfin !)

- On presque la même sémantique qu'une fonction
- concrètement, c'est une fonction qui prend des arguments et qui renvoie un fragment d'AST
- résolue avant la phase **d'expansion**

A quoi ressemble un fragment d'AST ?

1 + 2 + 3 + 4



```
{:+, [context: Elixir, import: Kernel],  
  [  
    {:+, [context: Elixir, import: Kernel],  
      [{:+, [context: Elixir, import: Kernel], [1, 2]}, 3]},  
    4  
  ]}  
}}
```

C'est un peu ennuyant à écrire à la main...

Quotations

Heureusement, Elixir permet, comme Lisp, de manipuler des quotations via la macro `quote/1`

```
iex> quote do: if(true, do: "foo", else: "bar")
```

```
{:if, [
  context: Elixir, import: Kernel], [
  true, [do: "foo", else: "bar"]]}
```

- Les macros permettent d'injecter, à la compilation, un fragment d'AST **syntactiquement valide**
- Les quotations permettent de simplifier l'écriture de fragments d'AST
- On peut **dé-quoter** (via `unquote/1`) des identifiants (qui seront injectés en fonction du contexte)
- Utilisées par beaucoup de bibliothèques majeures (Ecto, Plug, ExUnit etc.) pour étendre le langage (via des DSL's)
- Utilisées pour décrire 80% des expressions du langage

Un premier exemple : implémentons `unless`

Partiellement volé à la documentation d'Elixir

```
if(x) == unless(x)
```

Une première approche au moyen de fonctions

```
def my_unless(predicate, do: expression) do
  if(!predicate, do: expression)
end
```

```
iex(2)> my_unless false, do: IO.puts "foo"
foo # Ça semble fonctionner
:ok
```

```
iex(1)> my_unless true, do: IO.puts "foo"
foo # Damn, le "foo" s'affiche tout de même !
nil
```

Utilisation de la paresse pour corriger l'approche fonctionnelle

```
def my_unless(predicate, do: expression) do
  if(!predicate, do: expression.())
end
```

```
iex(1)> my_unless false, do: fn() -> IO.puts "foo" end
foo
:ok
```

```
iex(2)> my_unless true, do: fn () ->IO.puts "foo" end
nil # Victoire !
```

Mais c'est ... un peu laid... on est loin d'un DSL.

Utilisation de macro

```
defmacro my_unless(predicate, do: expression) do
  quote do
    if(! unquote(predicate), do: unquote(expression))
  end
end
```

```
iex(9)> require Sample
iex(10)> Sample.my_unless false, do: IO.puts "foo"
foo
:ok
```

```
iex(11)> Sample.my_unless true, do: IO.puts "foo"
nil # Victoire Réelle !
```