PETITE EXPLORATION DES BITSTRINGS EN ELIXIR

ELIXIR

Voyons, en quelques slides un rapide apperçu du langage qui va nous occuper ce soir.

ERLANG : LE GRAND FRÈRE

- Fin des années 80 chez *Ericson* (Ericson language + homage à A.K. Erlang)
- Typage dynamique
- fonctionnel
- pattern matching
- Une API native pour démarrer et faire communiquer des process extrêmement ergonomique

OTP

(aka. **Open Telecom Platform**)

Bibliothèque standard fortement dogmatique car fournissant des modules de gestion des process comme :

- `GenServer`
- `Supervisor`
- `GenEvent`…

1.3 ELIXIR 

José Valim

- la puissance d'erlang
- la productivité de ruby
- l'extensibilité de ruby
- le tooling (mix <3, hex, hexdoc/exdoc)

KOOP

Mon side project (pas l'ambition d'aller plus loin que ça)

Objectif ludique et pédagogique (pour moi même)

OBJECTIFS :

- parcourir une bibliothèque musicale moyennement vaste et permettre de la streamer depuis le web
- utiliser des technos que j'ai envie d'approfondire pour y parvenir
- extraire au maximum des petites libs qui pourront me servir à faire autre chose

BESOINS :

- parcourir le file system (easy)
- pour chaque fichier : extraire les métadonnées qui m'intéressent (nom du morceau, artiste, album, numéro de piste etc...)

PRÉ-REQUIS :

- inventorier les formats de fichier qui composent ma bibliothèque
- écrire des modules capable de parser ces fichiers vers un format commun
- pour chaque fichier réussir à déterminer quel parser utiliser

UN PREMIER PARSER

Commençons par le commencement : ID3V1 (en un tout petit peu simplifié)

- Un bloc de 128 octets préfixé par **"TAG"**
- Une suite de blocs de 30 octets décrivant les métadonnées du morceau

ÉQUIPONS NOUS, UN PEU DE SYNTAXE !

```
intro = "je m'appelle Julien"
#> "je m'appelle Julien"

"je m'appelle " <> firstname = intro
#> "je m'appelle Julien"

greetings = "Bonjour #{firstname} !"
#> "Bonjour Julien !"
```

```
a_string = "a string"
#> "a string"

String.length a_string
#> 8
```

Le langage permet aussi de manipuler des chaines de bits :

```
<<1, 3>>
#> <<1,3>>
```

Par défaut les éléments sont des octets

```
<< 72, 101, 121, 32, 33>>
#> "Hey !"
```

Les strings sont strictement équivalents à des bitstring dont les élèments sont des octets

On peut surcharger la taille de chaque élèment d'un bitstring à l'aide de l'expression size()

```
<< a::size(1), b::size(7) >> = "T"
#> "T"

a
#> 0

b
#> 7
```

On a utilisé le pattern matching pour assigner *a* et b dans la première ligne

```
<< a_binary :: binary >> = a_string
#> "a string"

a_binary == a_string
#> true
```

Voyons d'abord si notre fichier a bien le bon préfixe

```
defmodule ID3V1 do

  def parse(mp3) do
    case mp3 do
      "TAG" <> actual_tag -> do_parse(actual_tag)
      _ -> IO.puts "This file does not respect the ID3V1 standard
    end
  end

end
```

```elixir
def do_parse(mp3) do
  <<  title          :: binary-size(30),
      artist         :: binary-size(30),
      album          :: binary-size(30),
      year           :: binary-size(4),
      comment        :: binary-size(30),
      _track_present? :: binary-size(1),
      track          :: binary-size(1),
      genre          :: binary-size(1)
      _              :: binary >> = mp3
```

```elixir
  %{
    title:   title,
    artist:  artist,
    album:   album,
    year:    year,
    comment: comment
    track:   (if track_present? == 0, do: track, else: nil )
  }
end
```

YOUPI !

On a maintenant une map avec les métadonnées de notre morceau

Maintenant un autre format : un ID3 légèrement amélioré qui consiste en

- un tag : "TAGV2"
- un champ de 4 octets déterminants la taille du tag
- une liste de champs divisés en
  - 4 octets déterminant la taille du champ
  - un champ constitué de son identifiant `ARTIST`, `ALBUM`, `COMMENT`... et de sa valeur séparé par un "="

Comme tout à l'heure une petite fonction pour controller que le fichier est correct

```
defmodule TagV2 do
  def parse(tagV2) do
    case mp3V2 do
      "TAGV2" <> actual_tag -> do_parse(actual_tag)
      _ -> IO.puts "This file does not respect the TAGV2 standard
  end
end
```

Avec notre nouvelle norme on peut maintenant avoir plusieurs valeurs pour le même champ.

Définissons donc une nouvelle manière de les stocker :

```elixir
def new_track_map() do
  %{
    title: [],
    artist: [],
    album: [],
    track: [],
    comment: [],
    year: []
  }
end
```

De façon assez naïve on utilise juste des listes vides pour chaque champ

On peut maintenant écrire une fonction qui pour un champ donné va enrichir une map qui a cette forme

```
def add_field(map, field_string) do
  case field_string do
    "ARTIST=" <> artist  ->
      %{ map | artist: [artist | map[:artist]]}
    "TITLE=" <> title    ->
      %{ map | title: [title | map[:title]]}
    "ALBUM=" <> album    ->
      %{ map | album: [album | map[:album]]}
    "TRACK" <> track     ->
      %{ map | track: [track | map[:track]]}
    "YEAR" <> year       ->
      %{ map | year: [year | map[:year]]}
    "COMMENT" <> comment ->
      %{ map | comment: [comment | map[:comment]]}
  end
end
```

L'écriture de la fonction do_parse ne requiert plus qu'un peu de pattern matching

```
def do_parse(tag_v2), do: do_parse(new_track_map(), tag_v2)

def do_parse( map, <<>>), do: map
def do_parse( map, tag_v2) do
  <<
    field_length   :: size(32),
    field          :: binary-size(field_length),
    followup_bytes :: binary
  >> = tag_v2

  map
  |> add_field(field)
  |> do_parse(followup_bytes)
end
```

Si on récapitule un peu, on a :

- deux modules qui contiennent des fonctions Parse pour 2 formats de métadonnées différents

Le problème que l'on a : **nos map ne sont pas de la même forme**

PROPOSITION DE SOLUTION :

Elixir propose un moyen de rendre générique une interface de module :

Les Behaviours

Un **behaviour** est un module dans lequel certaines fonction ne sont pas implémentées, elles sont appelées **callbacks**

On va donc définir un module **Parser** qui exposera l'interface voulue

Tout d'abord une structure commune pour les résultats de parsing

```
defmodule Parser do

  defmodule Track do
    defstruct [
      title: [],
      artist: [],
      album: [],
      track: [],
      comment: [],
      year: []
    ]
  end

end
```

```elixir
defmodule Parser do
  # ...

  @callback valid?(parser, bit_string) :: boolean

  @callback parse(bit_string) :: %__MODULE__.Track{}

end
```

```
known_parsers = [ID3, TagV2]

def get_metadata(bytes) do
  parser = List.find(known_parsers, fn parser -> parser.valid?(by

  if is_nil(parser) do
    {:error, "No known parser can parse this bitstring}
  else
    {:ok, parser.parse(bytes)}
  end
```

CONCLUSION

Jusqu'à présent je pense qu'Elixir tient largement les promesses dont tout le monde entend parler.

Mais en plus, au détour de vos experiences de programmation vous pouvez tomber sur des features originales, j'avais déja essayé d'implémenter ce genre de parsers, mais jamais avec un tel confort et j'avais envie de vous partager cette heureuse trouvaille.

MERCI !