Combinaison d'analyseurs syntaxiques en Elixir avec NimbleParsec

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 11 novembre 2021

https://www.bortzmeyer.org/combi-analyseurs-elixir.html

On a souvent besoin en programmation de faire de l'analyse syntaxique, pour vérifier qu'un texte est correct et pour y trouver les éléments qui nous intéressent. (Je ne considère ici que les textes écrits dans un langage formel, l'analyse de la langue naturelle étant une autre histoire.) Une des techniques que je trouve les plus agréables est celle des combinaisons d'analyseurs. Voici un exemple en Elixir, avec le module NimbleParsec <https://github.com/dashbitco/nimble_parsec>.

L'idée de base est simple : on écrit l'analyseur en combinant des analyseurs simples, puis en combinant les combinaisons. Cela se fait par exemple en Haskell avec le module Parsec dont j'ai déjà parlé ici <https://www.bortzmeyer.org/combinatorial-parser.html>. En Elixir, cela peut se faire avec l'excellent module NimbleParsec <https://github.com/dashbitco/nimble_parsec> qui est plutôt bien documenté <https://hexdocs.pm/nimble_parsec/NimbleParsec.html>.

Commençons par un exemple trivial, des textes composés d'un seul mot, qui ne doit comporter que des lettres ASCII. L'analyseur consistera en un seul analyseur primitif, `ascii_string`, fourni par NimbleParsec, et qui prend deux arguments, un pour restreindre la liste des caractères autorisés, et l'autre pour indiquer des options comme, ici, le nombre minimal de caractères :

```
verb = ascii_string([], min: 1)
defparsec :test, verb
```

(Le code complet est en (en ligne sur `https://www.bortzmeyer.org/files/nimbleparsec-1.exs`).) On peut alors analyser des chaînes de caractères et voir le résultat. Si je mets dans le code :

```
IO.inspect Test1.test("foobar")
IO.inspect Test1.test("")
```

La première chaine sera acceptée (on n'a guère mis de restrictions à `ascii_string`), la seconde refusée (en raison du `min: 1`) :

```
{:ok, ["foobar"], "", %{}, {1, 0}, 6}
{:error, "expected ASCII character", "", %{}, {1, 0}, 0}
```

La valeur retournée par l'analyseur est un tuple commençant par un mot qui indique si tout s'est bien passé (`:ok` ou `:error`), une liste donnant les termes acceptés (ici, il n'y en a qu'un) si tout s'est bien passé.

On peut utiliser ces résultats dans du code Elixir classique, avec *"pattern matching"* :

```
def print_result(r) do
    case r do
      {:ok, list, _, _, _, _} -> IO.inspect("Tout va bien et j'ai récupéré #{list}")
      {:error, message, _, _, _, _} -> IO.inspect("C'est raté, à cause de #{message}")
    end
end
```

Ah, si vous ne connaissez pas bien Elixir, la méthode « normale » pour utiliser NimbleParsec dans un projet serait de le déclarer en dépendance et d'utiliser Mix <https://hexdocs.pm/mix/Mix.html> pour gérer ces dépendances mais, ici, on va simplifier, on installe NimbleParsec avec Hex <https://hex.pm/> et on lance mix avec les bonnes options pour exécuter le code :

```
% mix archive.install hex NimbleParsec
% mix run --no-mix-exs nimbleparsec-2.exs
"Tout va bien et j'ai récupéré foobar"
"C'est raté, à cause de expected ASCII character"
```

Contrairement à ce que son nom pourrait faire croire, `ascii_string` n'accepte pas que des caractères ASCII, mais tout caractère codé sur huit bits. Dans l'exemple ci-dessus, il accepterait des chaines comme « café au lait » (avec le caractère composé et les espaces). Restreignons un peu :

```
  verb = ascii_string([?a..?z], min: 1)
```

Cette fois, seuls les caractères entre le petit a et le petit z seront acceptés. Testons :

```
% mix run --no-mix-exs nimbleparsec-3.exs
{:ok, ["foobar"], "", %{}, {1, 0}, 6}
{:error, "expected ASCII character in the range 'a' to 'z'", "", %{}, {1, 0}, 0}
{:ok, ["caf"], "é au lait", %{}, {1, 0}, 3}
```

« café au lait » a quand même été accepté car NimbleParsec s'arrête avec succès dès qu'il est satisfait. Deux solutions : tester le troisième membre du tuple (les caractères restants) pour vérifier qu'il est vide ou bien **combiner** `ascii_string` avec l'analyseur `eos` qui indique la fin de la chaine :

```
verb = ascii_string([?a..?z], min: 1)
defparsec :test, verb |> eos
```

La combinaison se fait avec l'opérateur classique de séquencement en Elixir, `|>`. Cette fois, ça marche :

```
% mix run --no-mix-exs nimbleparsec-3.exs
{:ok, ["foobar"], "", %{}, {1, 0}, 6}
{:error, "expected ASCII character in the range 'a' to 'z'", "", %{}, {1, 0}, 0}
{:error, "expected end of string", "é au lait", %{}, {1, 0}, 3}
```

Maintenant qu'on sait faire des combinaisons, allons plus loin. On voudrait analyser des chaînes du type « foo{bar} » avec un verbe suivi d'une valeur entre accolades :

```
verb = ascii_string([not: ?\{], min: 1)
value = ascii_string([not: ?\}], min: 1)
body = ignore(string("{")) |> concat(value) |> ignore(string("}"))
defparsec :test, verb |> concat(body) |> eos
```

Décortiquons un peu : `verb` est défini comme une chaine ne comportant **pas** l'accolade ouvrante (sinon, l'analyseur, qui est gourmand, ira jusqu'au bout et avalera tout, y compris l'accolade ouvrante). De même, `value` ne comporte pas l'accolade fermante. `body` est la composition des deux accolades et de la valeur. Les deux accolades ne servent que de délimiteurs, on ne veut pas récupérer leur valeur, donc on ignore leur résultat (alors que celui de `value` nous sera retourné, c'est le rôle de `concat`). Essayons avec ce code (en ligne sur `https://www.bortzmeyer.org/files/nimbleparsec-4.exs`) :

```
# Correct
IO.inspect Test4.test("foobar{ga}")
# Tous les autres sont incorrects
IO.inspect Test4.test("foobar")
IO.inspect Test4.test("foobar{ga")
IO.inspect Test4.test("foobar{}")
IO.inspect Test4.test("{ga}")
IO.inspect Test4.test("foobar{ga}extra")
```

Et ça donne :

```
% mix run --no-mix-exs nimbleparsec-4.exs
{:ok, ["foobar", "ga"], "", %{}, {1, 0}, 10}
{:error, "expected string \"{\"", "", %{}, {1, 0}, 6}
{:error, "expected string \"}\", followed by end of string", "", %{}, {1, 0}, 9}
{:error, "expected ASCII character, and not equal to '}'", "}", %{}, {1, 0}, 7}
{:error, "expected ASCII character, and not equal to '{'", "{ga}", %{}, {1, 0}, 0}
{:error, "expected string \"}\", followed by end of string", "}extra", %{}, {1, 0}, 9}
```

C'est parfait, « foobar{ga} » a été accepté, les autres sont refusés.

Maintenant, il est temps d'introduire un outil très utile de NimbleParsec, la fonction `generate`. Elle permet de générer des chaines de caractères conformes à la grammaire qu'on a décrite en combinant les analyseurs (lisez quand même la documentation, il y a des pièges). Voici un exemple (en (en ligne sur `https://www.bortzmeyer.org/files/nimbleparsec-5.exs`)) :

```
% mix run --no-mix-exs nimbleparsec-5.exs
<<125, 40, 252, 204, 123, 151, 153, 125>>
```

Que signifie cette réponse incompréhensible ? C'est parce que `ascii_string`, en dépit de son nom, n'accepte pas que des caractères ASCII mais aussi tous les caractères sur huit bits, qu'Elixir refuse ensuite prudemment d'imprimer. On va donc restreindre les caractères autorisés (avec l'intervalle `?a..?z`, déjà vu) et cette fois, ça marche :

```
% mix run --no-mix-exs nimbleparsec-6.exs
"gavi{xt}"
% mix run --no-mix-exs nimbleparsec-6.exs
"y{ltww}"
% mix run --no-mix-exs nimbleparsec-6.exs
"ha{yxsy}"
% mix run --no-mix-exs nimbleparsec-6.exs
"q{yx}"
```

Nous générons bien désormais des chaines conformes à la grammaire. `generate` est vraiment un outil très pratique (j'avais travaillé sur un projet ayant des points communs, Eustathius <`https://www.bortzmeyer.org/eustathius-test-grammars.html`>, mais qui n'est plus maintenu).

Voyons maintenant quelques autres combinateurs possibles (je vous rappelle que NimbleParsec a une bonne documentation <`https://hexdocs.pm/nimble_parsec/NimbleParsec.html`>). `repeat` permet de répéter un analyseur :

```
# Permettra "foo{bar}{baz}"
lang = verb |> repeat(body)
```

Tandis qu'`optional` permettra de rendre un analyseur facultatif :

```
# Permettra "foo{bar}{baz}" mais aussi "foo"
lang = verb |> optional(repeat(body))
```

Autres trucs utiles, le premier argument d'`ascii_string` permet de donner une liste d'intervalles de caractères acceptés. Le programme (en ligne sur `https://www.bortzmeyer.org/files/nimbleparsec-6.exs`) impose ainsi une lettre majuscule au début, puis permet des tirets :

```
verb = ascii_char([?A..?Z]) |> concat(ascii_string([?a..?z, ?-], min: 1))
value = ascii_string([?a..?z, ?-], min: 1)
body = ignore(string("{")) |> concat(value) |> ignore(string("}"))
parser = verb |> optional(repeat((body))) |> eos
```

Voici le résultat :

```
% mix run --no-mix-exs nimbleparsec-6.exs
"Yga{-c--}{yt}{--}"

% mix run --no-mix-exs nimbleparsec-6.exs
"H--n{-r-v}{-}{j--h}"

% mix run --no-mix-exs nimbleparsec-6.exs
"L-x{-bj}{-}"

% mix run --no-mix-exs nimbleparsec-6.exs
"Upi"
```

Comme exercice, je vous laisse déterminer comment interdire deux tirets consécutifs.

`ascii_string` va chercher des caractères ou plus exactement des octets. Si le texte est de l'Unicode, il sera probablement encodé en UTF-8 et on aura peut-être plutôt envie d'utiliser `utf8_string` :

```
verb = utf8_string([], min: 1)
...
IO.inspect Test7.test("café-au-lait")
```

Qui donnera :

```
{:ok, ["café-au-lait"], "", %{}, {1, 0}, 13}
```

Mais une sérieuse limite apparait : tous les caractères Unicode seront acceptés. On peut en éliminer certains (ici, l'espace - que vous ne voyez pas - et le point) avec `not` :

```
verb = utf8_string([not: ? , not: ?.], min: 1)
```

Mais les intervalles (comme `?a..?z` que vous avez vu plus haut) n'ont guère d'intérêt en Unicode, où les caractères qui vous intéressent ne sont probablement pas consécutifs. Il faudrait pouvoir utiliser les catégories Unicode <https://unicode.org/reports/tr44/#General_Category_Values>, mais je ne trouve pas de moyen de le faire avec NimbleParsec.

Des limites ou des défauts de cette solution ? Les deux principaux me semblent :

— Les messages d'erreur retournés par défaut ne me semblent pas idéaux et il est difficile d'insérer son propre traitement d'erreur.

— Il ne semble pas y avoir de moyen de faire de l'analyse incrémentale (« "*streaming*" ») ce qui est notamment génant pour l'analyse de protocoles réseau. De même, on ne peut pas facilement modifier la grammaire en fonction des données lues.

Bref, cette solution a l'avantage d'être très simple à mettre en œuvre pour un besoin ponctuel ou personnel, mais n'est pas forcément bien adaptée pour un analyseur « de production ».

Si vous voulez approfondir, je répète une dernière fois que la doc <https://hexdocs.pm/nimble_parsec/NimbleParsec.html> est très complète, mais il y a aussi l'article de Gints Dreimanis <https://serokell.io/blog/parser-combinators-in-elixir> et celui de Drew Olson <https://blog.drewolson.org/parser-combinators-in-elixir>. Et, sur Elixir lui-même, je compte beaucoup sur le livre de Dave Thomas <https://www.bortzmeyer.org/programming-elixir.html>.