# Recdescent

12 janvier 2026

Aujourd'hui on va faire un peu d'analyse syntaxique (cf les cours de théorie des langages). Le but du jeu est d'écrire un interpréteur de grammaire hors-contexte : étant donnée une *grammaire hors-contexte*, on crée un parseur à descente récursive qui permet de faire des requête sur le langage reconnu par une de ses équations. Afin de s'entraîner un peu sur les durées de vies chaque on va s'interdire d'utiliser l'élimination des durées de vies sur les différents prototypes de fonctions.

Formellement, notre interpréteur de grammaires supporte les grammaires respectant le format défini par la EBNF (= BNF étendue) suivante (si vous ne maîtrisez pas encore ces notations - on va détailler ci-dessous) :

```
Parseur ::= Equation ("\n" Equation)*
Equation ::= NonTerminal "->" Choix ("|" Choix)*
NonTerminal ::= ("a".."z")+
Choix ::= Symbole+
Symbole ::= NonTerminal | "'" Litteral "'"
Litteral ::= ("0".."9" | "+" | "$")*
```

Concrètement, on va tester notre interpréteur sur la grammaire suivante qui respecte donc la EBNF ci-dessus (en supposant que l'espace est un séparateur des méta-terminaux) :

```
axiome -> addition '$'
addition -> nombre '+' nombre
chiffre -> '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
nombre -> chiffre nombre | chiffre
```

Dans notre format de grammaire, chaque ligne définit une *équation*, qui correspond elle-même à la définition d'un langage. Le début de la ligne, à gauche de `->`, donne le nom de ce langage : c'est un *non-terminal*, qui est syntaxiquement juste une suite de lettre. La partie à droite de `->` donne la définition associée à ce nom : cette définition correspond à une union finie de *choix* (de langages) séparés par des `|`. Chaque *choix* correspond une concaténation finie de *symboles* (de langages). Un symbole est :

- soit un non-terminal qui, lorsque la grammaire est *correcte*, est associé à une **unique** définition, et dans ce cas, il représente le (plus petit) langage vérifiant cette définition dans la grammaire ;
- soit un *littéral* entre apostrophes `'`, et dans ce cas, le langage associé est réduit à un unique mot : ce littéral.

Dans notre exemple, les littéraux sont tous les chiffres de 0 à 9 et les caractères `+` et `$`. Et il y a 4 équations associées à 4 non-terminaux. Et le langage associé à `addition` contient le mot `24+325`, où `24` et `325` sont eux-mêmes dans le langage `nombre`.

Ainsi, dans notre programme, on va donner la grammaire ci-dessus comme une grosse chaîne de caractères `equations` qui permet de créer le parseur via `Parseur::new(equations)`. Et on peut alors savoir si pour un mot `chaine` donné, il existe un préfixe de cette chaîne dans le langage d'un non-terminal `equation` du parseur `p`, en vérifiant que `p.parse(equation, chaine)` retourne un vecteur non vide.

En Rust, on utilisera les classes `Symbole, Choix, Parseur` pour stocker notre grammaire. Chacune de ces classes représente le (méta)non-terminal correspondant dans la EBNF. `Symbole` est un type énuméré permettant de stocker un littéral ou un non-terminal. `Choix` contient simplement un vecteur de symboles (par exemple pour `addition` il n'y a qu'un seul choix mais il contient 3 symboles). Enfin `Parseur` est une table de hachage pour stocker les équations, dont chaque clé est un non-terminal et chaque valeur est le vecteur des choix possibles associés à ce non-terminal. Chacune de ces classes a un constructeur `new` à partir d'une chaîne respectant la syntaxe de la EBNF et une méthode `parse` pour effectuer les recherches avec ce parseur. Dans le cas de `Symbole` et `Choix`, le parsing dépend d'un parseur englobant. Dans le cas de `Parseur`, le parsing dépend d'un non-terminal (correspondant à une équation).

# 1 La construction du parseur

On vous demande dans un premier temps d'ajouter les durées de vie manquantes sur les différentes structures puis d'implémenter les constructeurs `new` de `Symbole`,`Choix` et `Parseur`. Vous pouvez ici implémenter ces trois fonctions incrémentalement en testant au fur et à mesure. Pour simplifier un peu les choses, on supposera que les grammaires utilisées sont correctes et respectent la syntaxe de la EBNF (on ne vous demande pas de rejeter les grammaires incorrectes).

La bibliothèque standard est votre amie. Il y a de nombreuses fonctions très puissantes de manipulation de chaînes de caractères (*lines, split, split_once, split_whitespace*) qui peuvent vous simplifier la vie.

# 2 L'implémentation du parsing

Pour le parsing, vous devez implémenter les méthodes `parse` : attention, elles sont mutuellement récursives (comme finalement les équations de la grammaire sous-jacente). Pour simplifier, on supposera que la grammaire en entrée *n'est pas récursive à gauche*, c'est-à-dire qu'il n'existe pas de non-terminal $A$ et de mot de symboles $\alpha$ tel que $A \Rightarrow^+ A\alpha$. Sous cette hypothèse, toute dérivation gauche "finit" par produire $\varepsilon$ ou un mot qui commence par un terminal.

Pour chacune des méthodes, étant donné une `chaine` en entrée, on doit retourner un `Vec<&str>` qui pour la `chaine` en entrée, contient tous ses suffixes $v$ tels qu'il existe un mot $u$ dans le langage considéré satisfaisant `chaine` $= uv$. Le vecteur est donc vide, s'il n'y a aucun préfixe de `chaine` dans le langage. Et, pour le langage associé à `addition`, sur la chaine `24+325`, `Parseur::parse` retourne le vecteur `["", "5", "25"]`. Pour le langage associé à `axiome`, sur la chaine `24+325$`, il retourne le vecteur `[""]`.

Vous pouvez commencer par vous poser un peu et réfléchir aux prototypes de ces fonctions et à leurs annotations de durée de vie. Le code est finalement assez court mais pas si facile.

## 3 Bonus (optionnel) “encore plus de théorie des langages !”

Avec votre implémentation, répondez aux questions suivantes :

1. Que donne le parsing des chaînes `0011$` et `00$` et `1$`
   sur l’équation `axiome` de la grammaire suivante
   qui reconnaît le langage $\{0^n 1^{2m}\$ \mid \ n \geq m\}$ ?

   ```
   axiome -> a '$'
   a -> '0' a | '0' a '11' | ''
   ```

2. Que donne le parsing des chaînes `1100$` et `00$` et `1$`
   sur l’équation `axiome` de la grammaire suivante
   qui reconnaît le langage $\{1^{2m} 0^n\$ \mid \ n \geq m\}$ ?

   ```
   axiome -> a '$'
   a -> a '0' | '11' a '0' | ''
   ```

3. Y a-t-il une grammaire équivalente à la précédente et telle que le parsing des chaînes précédentes fonctionne ?