

Transformation et génération de mots à partir de grammaires formelles

Année : 2024-2025

Nom : François

Prénom : Alexandre

Numéro étudiant : 22201695

Contents

1	Structure de données	3
1.1	Classe <code>Grammaire</code>	3
1.2	Organisation des règles de production	3
2	Analyseur lexical	4
2.1	Classe <code>AnalyseurLexical</code>	4
2.2	Fonctionnement	5
2.3	Exemple d'analyse lexicale	5
3	Explications détaillées du code et des algorithmes	5
3.1	Organisation générale	5
3.2	La classe <code>Grammaire</code>	6
3.3	Génération dynamique de nouveaux Non-Terminaux	6
3.4	Détails sur les transformations CNF et GNF	7
3.4.1	Forme Normale de Chomsky (CNF)	7
3.4.2	Forme Normale de Greibach (GNF)	7
3.5	Lecture et écriture de la grammaire : <code>lire</code> et <code>ecrire</code>	8
3.6	Script <code>grammaire.py</code> : transformations	9
3.7	Script <code>generer.py</code> : génération de mots	9
3.8	Fichiers de test (<code>test.general</code>)	9
3.9	Conclusion	10
4	Gestion du projet et automatisation	10
4.1	Le <code>Makefile</code>	10
4.2	Les dépendances : <code>requirements.txt</code>	11

1 Structure de données

Dans le cadre de ce projet, la structure de données principale est celle permettant de représenter une **grammaire formelle**. Cette structure est définie au sein de la classe **Grammaire**, qui sert de réceptacle pour l'ensemble des règles de production, des Non-Terminaux et de l'Axiome.

1.1 Classe Grammaire

Axiome (`self.axiome`) Il s'agit du symbole de départ de la grammaire (l'Axiome). Toutes les dérivations qui génèrent des mots (ou des chaînes de Terminaux) commencent par ce symbole. Dans l'implémentation, le premier Non-Terminal rencontré sera automatiquement considéré comme Axiome.

Non-Terminaux (`self.non_terminaux`) L'ensemble des symboles Non-Terminaux (par exemple `S0`, `A0`, `B1`, etc.) est conservé sous forme d'un `set`. Les Non-Terminaux sont les symboles qui peuvent encore être développés via des règles de production afin de générer des chaînes de Terminaux.

Règles de production (`self.regles_de_production`) La clé de ce dictionnaire est un Non-Terminal, tandis que la valeur associée est une liste de *productions*. Chaque production est une liste (ou sous-liste) de symboles ; chaque symbole peut être :

- ("NON_TERMINAL", "A1") : un Non-Terminal nommé A1.
- ("TERMINAL", "a") : un Terminal, par exemple la lettre a.
- ("EPSILON", "E") : la production vide (EPSILON).

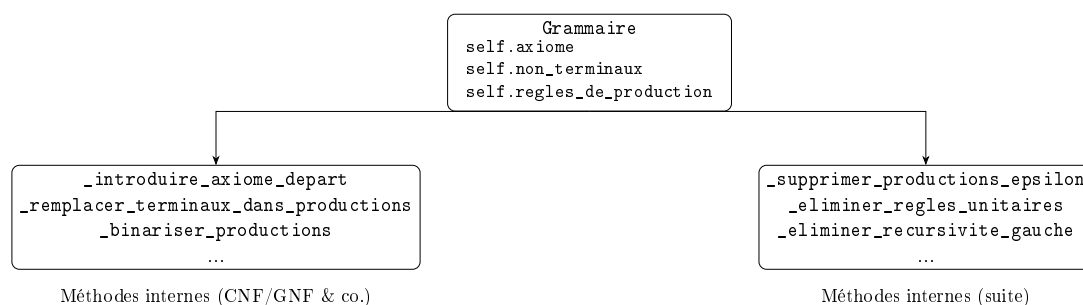


Figure 1: Structure simplifiée de la classe **Grammaire**, avec ses principaux attributs et méthodes internes.

1.2 Organisation des règles de production

Concrètement, chaque entrée de `self.regles_de_production` a pour clé un Non-Terminal (*exemple* : `"S0"`) et pour valeur une liste de listes. Par exemple :

Listing 1: Exemple de règles de production

```

1 {
2     "S0": [
3         [("NON_TERMINAL", "A0")],
4         [("TERMINAL", "a"), ("NON_TERMINAL", "B1")]
5     ],
6     "A0": [
7         [("EPSILON", "E")]
8     ],
9     ...
10 }

```

Ici, S0 se dérive soit en A0, soit en aB1. Quant à A0, il peut produire la chaîne vide (EPSILON).

2 Analyseur lexical

Pour **interpréter** les lignes de texte représentant des règles de grammaire (dans un fichier `.general`), nous avons besoin de distinguer différents types de symboles : Non-Terminaux, Terminaux, séparateurs de règles, symboles EPSILON, etc. C'est la tâche de l'analyseur lexical `AnalyseurLexical`.

2.1 Classe `AnalyseurLexical`

Le fichier `analyseur_lexicale.py` définit la classe `AnalyseurLexical` qui utilise le module `ply.lex` (Python Lex-Yacc) pour mettre en place un lexer. Voici ses principales composantes :

- Liste des tokens :

Listing 2: Extrait du code pour la liste des tokens

```

1 tokens = ('REGLE', 'NON_TERMINAL', 'TERMINAL', 'PIPE', 'EPSILON')

```

- `t_REGLE = r':'` : permet de détecter le symbole ":", qui sépare le Non-Terminal à gauche des productions.
- `t_PIPE = r'\|'` : pour reconnaître le séparateur "|" entre productions.
- `t_TERMINAL = r'[a-z]'` : ici, on considère que tout symbole en minuscule sera traité comme un Terminal (ex. a, b, c).
- `t_EPSILON = r'E'` : ce qui permet de reconnaître le symbole "E" comme EPSILON.
- `t_ignore = ' \t'` : pour ignorer les espaces et tabulations.
- `def t_NON_TERMINAL(self, t):`
`r'[A-DF-Z]\s*[0-9]'` : cette méthode spéciale détecte un Non-Terminal, par exemple A0 ou B1. L'expression rationnelle autorise une lettre majuscule (sauf E pour éviter la confusion avec EPSILON) suivie d'un chiffre. Au moment de la capture, les espaces intermédiaires sont retirés via `t.value.replace(' ', '')`.

2.2 Fonctionnement

Lorsque l'on appelle la méthode `analyser_texte(texte: str)`, le lexer va découper la chaîne `texte` en une suite de **tokens**. Chaque token contient deux informations essentielles : **type** (ex. `NON_TERMINAL`) et **value** (ex. `"A0"`). Si un caractère non reconnu est rencontré, une exception `ValueError` est levée pour signaler l'erreur (fonction `t_error`).

2.3 Exemple d'analyse lexicale

Supposons qu'une ligne du fichier contienne :

S0 : aB1 | A1

Le lexer va identifier les tokens successifs :

Listing 3: Exemple d'analyse lexicale

```
1 NON_TERMINAL("S0") # Le Non-Terminal S0
2 REGLE(":") # Le symbole de separation :
3 TERMINAL("a") # Un Terminal 'a'
4 NON_TERMINAL("B1") # Le Non-Terminal B1
5 PIPE("|") # Le separateur de production
6 NON_TERMINAL("A1") # Le Non-Terminal A1
```

Ensuite, chaque production sera traitée pour être stockée dans la **Grammaire**, comme une liste de symboles (voir [1.2](#)).

3 Explications détaillées du code et des algorithmes

Dans cette section, nous étudions plus en profondeur la structure et le fonctionnement des diverses parties du code, depuis la classe **Grammaire** (section [1.1](#)) jusqu'aux scripts d'exécution.

3.1 Organisation générale

Le projet est organisé en plusieurs fichiers :

- `analyseur_lexical.py` : contient la classe `AnalyseurLexical`, chargée de découper chaque ligne de la grammaire en tokens (`NON_TERMINAL`, `TERMINAL`, `EPSILON`, etc.).
- `outil_grammaire.py` (ou équivalent) : déclare la classe `Grammaire` et les fonctions associées `lire` et `ecrire`, permettant de lire ou d'écrire des grammaires depuis/vers un fichier.
- `grammaire.py` : script principal qui transforme une grammaire (CNF, GNF) et exporte les résultats.
- `generer.py` : script qui génère des mots à partir de la grammaire, jusqu'à une longueur spécifiée.

Cette séparation rend le code modulaire et facilite l'utilisation en ligne de commande.

3.2 La classe Grammaire

La classe `Grammaire`, déjà décrite à la section 1.1, propose des méthodes de transformation (CNF/GNF) et un mécanisme de nettoyage. Nous rappelons seulement qu'elle manipule l'Axiome (`self.axiome`), l'ensemble de Non-Terminaux (`self.non_terminaux`) et les règles de production (`self.regles_de_production`).

Transformations principales :

- `convertir_en_forme_normale_de_Chomsky()` (CNF) : introduit un nouvel Axiome si besoin, remplace les Terminaux isolés, binarise les productions, supprime les EPSILON et élimine les règles unitaires.
- `convertir_en_forme_normale_de_Greibach()` (GNF) : supprime les EPSILON, élimine les règles unitaires, gère la récursivité gauche, re-supprime EPSILON si nécessaire, introduit des Terminaux en tête de règle et segmente les productions longues.

Méthodes internes (privées) :

- `_introduire_axiome_depart()` : crée un nouveau Non-Terminal pour pointer vers l'ancien Axiome.
- `_remplacer_terminaux_dans_productions(...)` : déplace les TERMINAL vers des règles dédiées (`T_a -> a`) pour la CNF ou GNF.
- `_binariser_productions()` : découpe toute production de plus de deux symboles en productions binaires (ex. `S -> A X`, `X -> B C`).
- `_supprimer_productions_epsilon()` : identifie les Non-Terminaux *annulables* (EPSILON) et supprime les règles vides, sauf si l'Axiome est concerné.
- `_eliminer_regles_unitaires()` : remplace `X -> Y` (un seul Non-Terminal) par les productions de Y.
- `_eliminer_rekursivite_gauche()` : gère la récursivité gauche directe (`A -> A ...`) et indirecte (`A -> B`, `B -> A`).

Nettoyage (`nettoyer_grammaire()`) : La classe `Grammaire` inclut aussi des fonctions pour supprimer les Non-Terminaux inaccessibles, vides ou non-productifs :

- `supprimer_non_terminaux_vides()`,
- `supprimer_non_productifs()`,
- `supprimer_non_terminaux_inaccessibles()`.

3.3 Génération dynamique de nouveaux Non-Terminaux

Pour les transformations CNF ou GNF, il peut être nécessaire de créer des symboles Non-Terminaux *temporaires*. La fonction `generer_non_terminal(non_terminaux)` génère un nom du type "A0", "B0", etc., en évitant les collisions avec l'ensemble existant.

3.4 Détails sur les transformations CNF et GNF

Au-delà du simple appel aux méthodes `convertir_en_forme_normale_de_Chomsky()` et `convertir_en_forme_normale_de_Greibach()`, nous détaillons ici les différentes étapes suivies pour chaque forme normale.

3.4.1 Forme Normale de Chomsky (CNF)

Une grammaire est dite en *Forme Normale de Chomsky* si toutes ses productions vérifient l'un des trois types :

1. $A \rightarrow BC$, où A, B, C sont des non-terminaux (pas d'autres symboles),
2. $A \rightarrow a$, où a est un terminal,
3. (Optionnel) $S \rightarrow \epsilon$ (production vide) seulement si l'axiome S est nullable.

Notre méthode `convertir_en_forme_normale_de_Chomsky()` applique successivement :

1. **START** : on introduit un nouvel axiome S' pour éviter les cas où l'axiome initial est présent à droite de certaines règles. Concrètement, si l'axiome initial est $S0$, on crée $S' \rightarrow S0$.
2. **TERM** : on remplace les terminaux dans les productions de longueur > 1 par des non-terminaux dédiés. Par exemple, si une production contient $\dots a \dots$, on crée un nouveau non-terminal A_a tel que $A_a \rightarrow a$, et on remplace a par A_a .
3. **BIN** : on binarise toutes les productions qui ont plus de deux symboles. Par exemple, $S \rightarrow ABC$ devient $S \rightarrow AX$ et $X \rightarrow BC$. Le code parcourt récursivement les productions afin de créer de nouveaux non-terminaux si nécessaire.
4. **DEL** : on supprime les ϵ -productions (c'est-à-dire les règles produisant la chaîne vide). On identifie d'abord tous les non-terminaux *nullable*, puis on génère toutes les variantes des productions sans ces non-terminaux, sauf si l'axiome lui-même est nullable (dans ce cas, on autorise ϵ uniquement pour l'axiome).
5. **UNIT** : on élimine les règles unitaires (de la forme $A \rightarrow B$). Pour cela, on remplace $A \rightarrow B$ par l'ensemble des productions de B .

Une fois ces étapes réalisées dans l'ordre indiqué, la grammaire obtenue est en CNF : toute production aura soit deux non-terminaux (type $A \rightarrow BC$), soit un terminal isolé (type $A \rightarrow a$), soit éventuellement ϵ si l'axiome est nullable.

3.4.2 Forme Normale de Greibach (GNF)

Une grammaire est dite en *Forme Normale de Greibach* si toutes ses productions commencent par un **terminal**, éventuellement suivi d'une suite (éventuellement vide) de non-terminaux. Formellement, toute règle est du type :

$$A \rightarrow a X_1 X_2 \dots X_k$$

où a est un terminal, et chaque X_i est un non-terminal. En plus, on autorise $S \rightarrow \epsilon$ si l'axiome S peut produire la chaîne vide.

Notre méthode `convertir_en_forme_normale_de_Greibach()` procède ainsi :

1. **DEL (préliminaire)** : on supprime les ϵ -productions (comme pour la CNF), sauf éventuellement pour l'axiome.
2. **UNIT (préliminaire)** : on élimine les règles unitaires $A \rightarrow B$.
3. **SUPPRIMER_RECURSIVITE_GAUCHE** : on élimine la récursivité gauche directe et indirecte, c'est-à-dire on transforme les règles de la forme $A \rightarrow A\alpha$ ou $A \rightarrow B$, $B \rightarrow A$, etc., afin d'éviter qu'un non-terminal ne se réécrive par lui-même en tête.
4. **DEL (encore une fois)** : il se peut que la phase précédente ait introduit de nouvelles ϵ -productions, on les élimine à nouveau si nécessaire.
5. **TERM** : on veille à ce que chaque production commence par un terminal. Si ce n'est pas le cas (ex. $\dots \rightarrow B\dots$), on remplace B par un terminal dédié ou on réinjecte les règles qui conduisent à un terminal.
6. **UNIT (encore une fois)** : on vérifie à nouveau l'absence de règles unitaires, car l'étape TERM peut en créer.
7. **DÉCOMPOSER_LONGUES_PRODUCTIONS** : enfin, si une production contient plus de deux symboles après le terminal (ex. $A \rightarrow aBC$), on segmente en plusieurs étapes ($A \rightarrow aX$, $X \rightarrow BC$) pour s'approcher du schéma « un terminal suivi de quelques non-terminaux ».

Ainsi, chaque règle débute par un unique terminal. Toute éventuelle récursivité gauche a été éliminée. La GNF a l'avantage de faciliter certaines preuves (ex. temps de reconnaissance linéaire dans les automates à pile, etc.) et de simplifier l'algorithme d'analyse descendante.

3.5 Lecture et écriture de la grammaire : lire et écrire

`lire(fichier: str) -> Grammaire`

1. Crée un `AnalyseurLexical()` pour analyser chaque ligne en tokens.
2. Instancie une `Grammaire`.
3. Parcourt le fichier `.general`, construit les productions et détecte le Non-Terminal de gauche.
4. Nettoie la grammaire via `nettoyer_grammaire()` et renvoie l'objet `Grammaire`.

`ecrire(grammar, fichier_base, extension)`

1. Construit un fichier de sortie (`.chomsky` ou `.greibach`).
2. Écrit d'abord l'Axiome et ses productions, puis les autres Non-Terminaux triés.
3. Pour chaque production, on concatène les symboles (ex. `X1`, `a`, `E`) en séparant par `|` si nécessaire.

3.6 Script `grammaire.py` : transformations

Ce script :

1. Lit le fichier `.general` donné en argument.
2. Copie la Grammaire pour appliquer deux transformations : `convertir_en_forme_normale_de_Chomsky()` et `convertir_en_forme_normale_de_Greibach()`.
3. Écrit les grammaires transformées dans `.chomsky` et `.greibach`.

3.7 Script `generer.py` : génération de mots

Le script `generer.py` sert à tester la grammaire en générant les mots reconnus jusqu'à une longueur donnée.

`generer_mots(longueur: int)`

- Démarre depuis l'Axiome, explore toutes les productions (méthode récursive).
- Ajoute les `TERMINAL`, déplie les `NON_TERMINAL` et ignore `EPSILON`.
- Stocke chaque mot final (entièrement dérivé) dans un `set` pour éviter les doublons.
- Renvoie la liste triée (ou signale qu'aucun mot n'est généré).

Exemple de commande :

Listing 4: Exemple d'utilisation de `generer.py`

```
1 $ python generer.py 3 exemple.general
```

Cette commande affiche tous les mots de longueur ≤ 3 produits par la grammaire contenue dans `exemple.general`.

3.8 Fichiers de test (`test.general`)

Deux fichiers de test ont été préparés, nommés `1.general` et `2.general`, correspondant à différentes grammaires. Les grammaires sont les suivantes :

1.general	S0 : A1S0B1 C1 A1 : a B1 : b C1 : c E	Mots de la forme $a^n cb^n$ ou un simple c.
2.general	S0 : A1S0A1 B1S0B1 E A1 B1 A1 : a B1 : b	Palindromes en a et b.

Table 1: Récapitulatif des fichiers de test `.general` et des langages correspondants.

Ces différents fichiers permettent de tester le **pipeline complet** (lecture, transformation, génération de mots) sur des grammaires variées, incluant des langages vides, des combinaisons de lettres, ou des palindromes.

3.9 Conclusion

Avec ces scripts, on dispose d'un **pipeline complet** pour :

1. Lire une grammaire (**.general**).
2. **Analyser** lexicalement et la convertir en une structure **Grammaire**.
3. Appliquer des **transformations** en formes normales (CNF, GNF).
4. **Générer** des mots pour valider la grammaire.

L'ensemble permet d'explorer, de tester et de manipuler différentes grammaires contextuelles de manière modulaire et extensible.

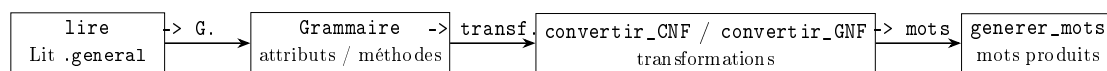


Figure 2: Pipeline général : lecture (**lire**) → création d'une **Grammaire** → transformations CNF/GNF → génération de mots.

4 Gestion du projet et automatisation

4.1 Le Makefile

Pour faciliter la génération et les tests autour de la grammaire, un fichier **Makefile** est fourni. Il définit plusieurs cibles (**all**, **test**, **clean**, ...) et automatise l'exécution des scripts **grammaire.py** (section 3.6) et **generer.py** (section 3.7). Voici ses principales fonctionnalités :

- **make** : Génère automatiquement les fichiers **.chomsky** et **.greibach** à partir de tous les fichiers **.general** détectés dans le répertoire. Concrètement, il invoque :

Listing 5: Extrait de Makefile pour la cible 'all'

```
1 python grammaire.py <fichier.general>
```

- **make N=<longueur> test** : Teste la génération de mots à partir de chaque fichier **.chomsky** et **.greibach**, pour une longueur de mots **obligatoirement spécifiée** par **N=<longueur>**. Par exemple, **make N=5 test** indiquera de générer des mots de longueur 5. Il appelle en interne :

Listing 6: Extrait de Makefile pour la cible ‘test’

```
1 python generer.py <longueur> <fichier.chomsky> > base_<longueur>_chomsky.  
   res  
2 python generer.py <longueur> <fichier.greibach> > base_<longueur>_greibach.  
   res  
3 diff -u base_<longueur>_chomsky.res base_<longueur>_greibach.res
```

afin de comparer les résultats `_chomsky.res` et `_greibach.res`.

- `make N=<longueur> clean` : Supprime tous les fichiers générés (`.chomsky`, `.greibach`, `_chomsky.res`, `_greibach.res`) pour la longueur `<longueur>`.
- `make help` : Affiche un résumé des différentes commandes proposées et rappelle qu’il faut spécifier `N=<longueur>` pour `test` et `clean`.

4.2 Les dépendances : `requirements.txt`

Le fichier `requirements.txt` répertorie les bibliothèques Python nécessaires au bon fonctionnement du projet. Dans notre cas, il se limite principalement à :

Listing 7: `requirements.txt`

```
1 ply~=3.11
```

Cela indique que nous utilisons la bibliothèque `ply` (version 3.11 ou approchante). Pour installer automatiquement cette dépendance, vous pouvez utiliser la commande :

Listing 8: Installation des dépendances

```
1 pip install -r requirements.txt
```

Version de Python Ce projet a été développé et testé avec **Python 3.11**.