

# Projet - Transformation de grammaires

Ahmad HATOUM (22202060) & Bastien GUIBERT (22307051)

January 18, 2025

## 1 Introduction

Au cours du semestre, nous avons étudié divers concepts de la théorie des langages tels que les **automates**, les **expressions régulières** et les **grammaires**. Dans ce projet, on se concentre sur les concepts de transformation de grammaire en **formes normales de Chomsky et de Greibach**.

## 2 Rappels sur les Grammaires (issus du cours)

### 2.1 Définition d'une Grammaire

Une grammaire est une représentation formelle utilisée pour définir un langage. Elle est définie comme un quadruplet  $G = (\Sigma, V, S, P)$ , où :

- $\Sigma$  : un alphabet fini de symboles **terminaux** (par exemple :  $a, b, c$ ).
- $V$  : un alphabet fini de symboles **non-terminaux** (ou variables), distincts des terminaux.
- $S \in V$  : un **axiome**.
- $P$  : un ensemble de **règles de production** sous la forme  $\alpha \rightarrow \beta$ , où  $\alpha, \beta \in (V \cup \Sigma)^*$ .

### 2.2 Règles de production

Une règle de production d'une grammaire  $G = (\Sigma, V, S, P)$  s'écrit sous la forme :  $\alpha \rightarrow \beta \in (V \cup \Sigma)^*$

- $\alpha$  est appelé la **membre gauche** de la règle
- $\beta$  est appelé la **membre droit** de la règle

### 3 Forme des grammaires algébriques

Une grammaire est **algébrique** si toutes les règles sont de la forme :

- $X \rightarrow \alpha$

Le membre gauche d'une règle est toujours un unique non-terminal et le membre droit un mot quelconque.

Les formes normales ci-dessous donnent des contraintes sur le membre droit.

#### 3.1 Forme normale de Greibach

Une grammaire algébrique est sous **forme normale de Greibach** si toutes les règles sont de la forme :

- $X \rightarrow aA_1A_2...A_n$  avec  $n \geq 1$
- $X \rightarrow \epsilon$ , seulement si  $\epsilon$  appartient au langage.

Un algorithme pour transformer une grammaire algébrique en grammaire équivalente sous la **forme normale de Greibach** est :

1. retirer l'axiome des membres droits des règles ;
2. supprimer les règles  $X \rightarrow \epsilon$  sauf si  $X$  est l'axiome ;
3. supprimer les règles unité  $X \rightarrow Y$  ;
4. supprimer les non-terminaux en tête des règles ;
5. supprimer les symboles terminaux qui ne sont pas en tête des règles.

#### 3.2 Forme normale de Chomsky

Une grammaire algébrique est sous **forme normale de Chomsky** si toutes les règles sont de la forme :

- $X \rightarrow YZ$
- $X \rightarrow a$
- $S \rightarrow \epsilon$ , seulement si  $\epsilon$  appartient au langage.

Un algorithme pour transformer une grammaire algébrique en grammaire équivalente sous la **forme normale de Chomsky** est :

1. retirer l'axiome des membres droits des règles ;
2. supprimer les terminaux dans les membre droit des règles de longueur au moins deux ;
3. supprimer les règles avec plus de deux non-terminaux ;
4. supprimer les règles  $X \rightarrow \epsilon$  sauf si  $X$  est l'axiome ;
5. supprimer les règles unité  $X \rightarrow Y$

## 4 Implémentation

Dans cette partie, on présentera chaque simplification, puis ensuite les fonctions de transformation des deux formes normales et l'énumération des mots générés.

### 4.1 Mise en contexte

Dans notre projet, nous avons choisi de travailler avec **Python**.

**L'architecture du projet** est décrite ci-dessous, l'ensemble des fichiers sont expliqués dans les parties qui suivent.

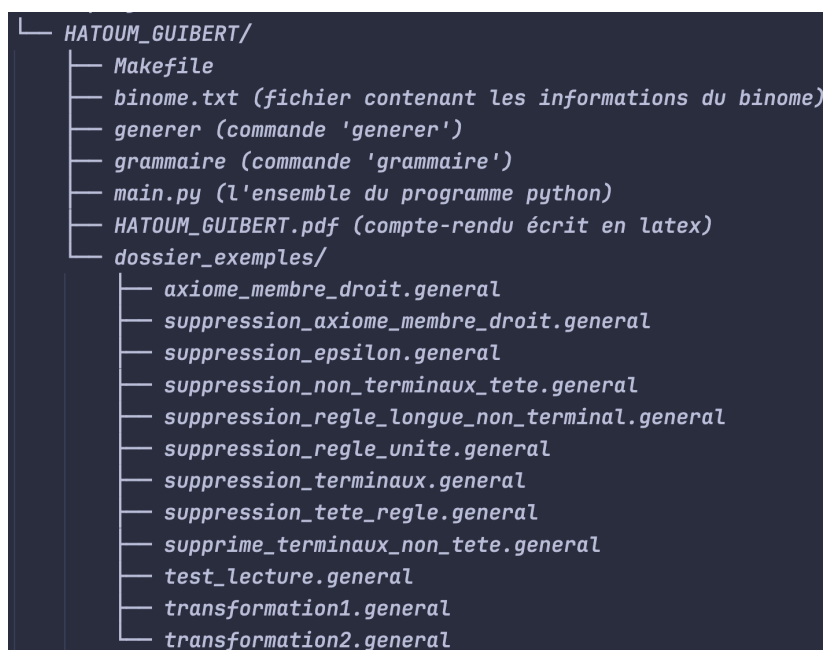


Figure 1: Architecture du projet

Tous les fichiers requis pour le projet sont présents dans le dossier **HATOUM\_GUIBERT**.

Un dossier contenant des **fichiers d'entrée de test** est également inclus.

L'ensemble des tests est présenté dans chaque **partie correspondante**.

Dans le fichier **main.py**, les tests mentionnés peuvent être exécutés **manuellement** dans la **section test** du fichier.

Pour la réalisation du projet, nous avons travaillé avec GitHub afin de **centraliser le code**, suivre les **modifications**, rester organisés et collaborer plus **facilement**. Le code source est **accessible** à cette adresse.

## 4.2 Structure de données choisie

On présente ci-dessous la classe 'Grammaire' sur laquelle on travaillera le long du projet.

```
1 class Grammaire:
2
3     def __init__(self):
4         self.__axiome = None
5         self.__terminaux = set()
6         self.__non_terminaux = set()
7         self.__regles = {}
8         alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
9         for letter in alphabet:
10             self.ajout_terminal(letter.lower())
11             for i in range(1, 11):
12                 self.ajout_non_terminal(f"{letter}{i}")
13             self.ajout_terminal("ε")
14
15     # Getters et Setters
16     def get_terminaux(self):
17         return self.__terminaux
18
19     def get_non_terminaux(self):
20         return self.__non_terminaux
21
22     def get_axiome(self):
23         return self.__axiome
24
25     def get_regles(self):
26         return self.__regles
27
28     def get_non_terminal_non_utilise(self):
29         for non_terminal in self.non_terminaux:
30             if non_terminal not in self.regles.keys() and all(non_terminal not in regle for regle in self.regles.values()):
31                 return non_terminal
32         return None
33
34     def set_axiome(self, axiome):
35         self.__axiome = axiome
36
37     terminaux = property(get_terminaux)
38     non_terminaux = property(get_non_terminaux)
39     axiome = property(get_axiome, set_axiome)
40     regles = property(get_regles)
```

Figure 2: Structure de données choisie pour représenter une grammaire

Dans cette structure de données, on ajoute tout les **terminaux** possible (les 26 lettres minuscules) et les **non-terminaux** (les 25 lettres majuscules (hors  $E$  que l'on considère comme  $\epsilon$  dans le code) tel que  $A1A2...A9A10$ ,  $\forall A \in$  non-terminaux ainsi que l'axiome) à leurs sets respectifs.

**L'axiome sera le premier membre gauche lu** (comme demander dans l'énoncé) dans le fichier d'entrée (voir Lecture du fichier d'entrée).

### 4.3 Lecture du fichier d'entrée

Comme demandé dans l'énoncé du projet, nous devons pouvoir **lire en entrée un fichier contenant la grammaire initiale** (les règles de transformation) puis l'intégrer dans notre structure de données.

```
1 def lire(self, file):
2     """ Lit une grammaire depuis un fichier texte avec extension .general uniquement. """
3
4     with open(file) as file:
5         data = file.readlines()
6
7     for line in data:
8         line = line.strip()
9         if not line or ":" not in line:
10            continue
11         membre_gauche, membre_droit = line.split(":")
12         membre_droit = [part.strip() for part in membre_droit.split("|")]
13         membre_gauche = membre_gauche.strip()
14         if membre_gauche in self.non_terminaux :
15             if self.regles == {}:
16                 self.set_axiome(membre_gauche)
17
18         membre_droit = [re.findall(fr'[A-Z](?:10|[1-9])[a-z]{{self.axiome}}E', symbol) for symbol in membre_droit]
19         if membre_gauche not in self.regles :
20             self.regles[membre_gauche] = membre_droit
21         else:
22             for regle in membre_droit :
23                 self.ajout_regle(membre_gauche, regle)
```

Figure 3: Fonction permettant la lecture d'un fichier

On '**découpe**' les lignes lues en entrée.

La fonction a été pensée afin de pouvoir lire le fichier d'entrée et de **l'intégrer proprement** dans la structure de données. Elle permet de gérer tout type d'entrée (respectant le format **membre gauche : membre droit**)

Comme signaler dans la partie précédente, le **premier membre gauche valide** (respectant la syntaxe) **devient l'axiome**.

**Remarque:** On identifie les symboles acceptés grâce au module '**regex**' (et donc à **une expression régulière**).

```
Fichier lu en entrée
Productions de la grammaire :
S1 -> a S1 b S2
S2 -> a | b | c

Règles de production de la grammaire stockées dans la structure de données
{'S1': [['a', 'S1', 'b', 'S2']], 'S2': [['a'], ['b'], ['c']]}
```

Figure 4: Exemple de stockage des règles d'une grammaire dans la structure de données

## 4.4 Les simplifications

Dans cette partie on liste **chaque algorithme de simplification** utilisé, et **un exemple** d'exécution.

### 4.4.1 Suppression de l'axiome des membres droits des règles

```
1 def suppression_axiome_membre_droit(self):
2     """ Supprime l'axiome des membres droits des règles. """
3
4     regles = list(self.regles.items())
5
6     for _, membre_droit in regles:
7         for valeur in membre_droit:
8             if self.axiome in valeur:
9                 new_axiome = self.get_non_terminal_non_utilise()
10                for regle in self.regles[self.axiome]:
11                    self.ajout_regle(new_axiome, regle)
12                self.set_axiome(new_axiome)
```

Figure 5: Fonction qui retire l'axiome des membres droits

Dans cette fonction, nous parcourons les règles de la grammaire et identifions celle où l'**axiome** est présent dans le **membre droit**. Une fois trouvé, **nous mettons à jour notre axiome** (en prenant un non-terminal disponible) **puis nous recopions les règles**.

```
---- TEST SUPPRESSION AXIOME MEMBRE DROIT ----
Productions de la grammaire :
S1 -> a S1 a | b S1 b | a | b | E
APRES SUPPRESSION AXIOME MEMBRE DROIT
Productions de la grammaire :
S1 -> a S1 a | b S1 b | a | b | E
K9 -> a S1 a | b S1 b | a | b | E
```

Figure 6: Test d'exécution

#### 4.4.2 Suppression des terminaux

```
1 def suppression_terminaux(self):
2     """ Supprime les terminaux. """
3
4     regles = list(self.regles.items())
5     association_terminal_non_terminal = {}
6
7     for membre_gauche, membre_droit in regles :
8         for i, regle in enumerate(membre_droit):
9             if len(regle) >= 2:
10                 for j, symbol in enumerate(regle):
11                     if symbol in self.terminaux:
12
13                         if symbol not in association_terminal_non_terminal:
14                             nouveau_non_terminal = self.get_non_terminal_non_utilise()
15
16                             association_terminal_non_terminal[symbol] = nouveau_non_terminal
17
18                             self.ajout_regle(nouveau_non_terminal, [symbol])
19
20                             self.regles[membre_gauche][i][j] = nouveau_non_terminal
```

Figure 7: Fonction qui supprime les terminaux

Dans cette fonction, nous parcourons les règles pour **supprimer les terminaux dans des membres droits où ils apparaissent avec d'autres symboles**. La fonction remplace chaque terminal présent par un non-terminal dédié (un non-terminal non utilisé). Si le terminal n'a pas encore de non-terminal associé, un nouveau est créé, une règle l'associant est ajoutée et **le terminal est remplacé dans la règle initiale par ce non-terminal**.

```
---- TEST SUPPRESSION TERMINAUX ----
Productions de la grammaire :
S1 -> a S1 b S1

APRES SUPPRESSION TERMINAUX

Productions de la grammaire :
S1 -> K6 S1 A9 S1
K6 -> a
A9 -> b
```

Figure 8: Test d'exécution

#### 4.4.3 Suppression des règles $X \rightarrow \epsilon$ sauf si $X$ est l'axiome

On décompose cet algorithme en deux parties, une qui itère et une autre qui relance tant qu'il est nécessaire.

```
1 def iteration_suppression_epsilon(self):
2     """ Itère la suppression des règles epsilon. """
3
4     terminaux_annule = set()
5
6     for membre_gauche, membre_droit in self.regles.items():
7         for regle in membre_droit:
8             for symbol in regle:
9                 if symbol == "E":
10                     if membre_gauche != self.axiome:
11                         terminaux_annule.add(membre_gauche)
12                         self.regles[membre_gauche].remove(regle)
13                         break
14
15     for membre_gauche, membre_droit in self.regles.items():
16         for regle in membre_droit:
17             for i, symbol in enumerate(regle):
18                 if symbol in terminaux_annule:
19                     nouvelle_regle = regle[:i] + regle[i+1:]
20                     if len(nouvelle_regle) == 0:
21                         nouvelle_regle = ["E"]
22
23                     if nouvelle_regle not in self.regles[membre_gauche]:
24                         self.ajout_regle(membre_gauche, nouvelle_regle)
25
26 def suppression_epsilon(self):
27     """ Supprime les règles epsilon. """
28
29     while any(
30         regle == ["E"] and membre_gauche != self.axiome
31         for membre_gauche, membre_droit in self.regles.items()
32         for regle in membre_droit
33     ):
34         self.iteration_suppression_epsilon()
```

Figure 9: Fonction qui supprime des règles  $X \rightarrow \epsilon$  sauf si  $X$  est l'axiome

Dans la fonction d'itération, on identifie d'abord les non-terminaux (hors axiome) qui s'annulent, c'est-à-dire qui possèdent une règle  $X \rightarrow \epsilon$  et on supprime ces règles. Ensuite, re-parcourt les règles pour modifier celles contenant ces non-terminaux, on génère une nouvelle version de celles-ci sans ce symbole, de plus s'il était unique dans une règle, on ajoute la règle qui donne  $\epsilon$ .

La fonction récursive permet elle de relancer l'exécution tant que toutes les conditions ne sont pas acquises (elle vérifie la présence de règle  $X \rightarrow \epsilon$  (hors axiome)).



```
---- TEST SUPPRESSION EPSILON ----  
  
Productions de la grammaire :  
S1 -> A1 b B1 | C1  
B1 -> A1 A1 | A1 C1  
C1 -> b | c  
A1 -> a | E  
  
APRES SUPPRESSION EPSILON  
  
Productions de la grammaire :  
S1 -> A1 b B1 | C1 | b B1 | A1 b | b  
B1 -> A1 A1 | A1 C1 | A1 | C1  
C1 -> b | c  
A1 -> a
```

Figure 10: Test d'exécution

#### 4.4.4 Suppression des règles unité $X \rightarrow Y$

```
1 def suppression_regle_unite(self):
2     """ Supprime les règles unités. """
3
4     regles = list(self.regles.items())
5
6     for membre_gauche, membre_droit in regles:
7         for regle in membre_droit:
8             if len(regle) == 1 and regle[0] in self.non_terminaux and regle[0] != membre_gauche:
9                 symbol = regle[0]
10
11                 for nouvelle_regle in self.regles[symbol]:
12                     if nouvelle_regle not in self.regles[membre_gauche]:
13                         self.ajout_regle(membre_gauche, nouvelle_regle)
14
15                 self.regles[membre_gauche].remove(regle)
```

Figure 11: Fonction qui supprime les règles unité  $X \rightarrow Y$

Dans cette fonction on parcourt les règles pour **supprimer celles qui sont dites 'unité'**, c'est-à-dire où le **membre droit se compose uniquement d'un non-terminal**. Lorsqu'une telle règle est détectée, le **non-terminal est remplacé par ces règles associées dans la grammaire**. Les nouvelles règles sont ajoutées intelligemment (uniquement si elles n'existent pas déjà pour le membre gauche). **On supprime ensuite cette règle.**

```
---- TEST SUPPRESSION REGLE UNITE ----

Productions de la grammaire :
A1 -> a A1 | B1
B1 -> b B1 | C1
C1 -> c C1

APRES SUPPRESSION REGLE UNITE

Productions de la grammaire :
A1 -> a A1 | b B1 | c C1
B1 -> b B1 | c C1
C1 -> c C1
```

Figure 12: Test d'exécution

#### 4.4.5 Suppression des règles avec plus de deux non-terminaux dans le membre de droite

On décompose cet algorithme en deux parties, une qui itère et une autre qui relance tant qu'il est nécessaire.

```
1 def iteration_suppression_regle_plus_deux_non_terminaux_membre_droite(self):
2     """ Itère la suppression des règles contenant plus de deux non-terminaux dans le membre droit. """
3
4     regles = list(self.regles.items())
5
6     for membre_gauche, membre_droit in regles:
7         nouvelles_regles = []
8
9         for regle in membre_droit :
10             nb_non_terminaux = sum(1 for symbol in regle if symbol in self.non_terminaux)
11             if nb_non_terminaux > 2 :
12                 compteur = 0
13                 for i, symbol in enumerate(regle):
14                     if symbol in self.non_terminaux and compteur < 2:
15                         compteur += 1
16                         continue
17
18                 if compteur == 2:
19                     nouveau_non_terminal = self.get_non_terminal_non_utilise()
20                     regle_modif = regle[:i-1] + [nouveau_non_terminal]
21                     nouvelle_regle = regle[i-1:]
22
23                     self.ajout_regle(nouveau_non_terminal, nouvelle_regle)
24                     nouvelles_regles.append(regle_modif)
25                     break
26             else:
27                 nouvelles_regles.append(regle)
28
29         self.regles[membre_gauche] = nouvelles_regles
30
31 def suppression_regle_plus_deux_non_terminaux_membre_droite(self):
32     """ Supprime les règles contenant plus de deux non-terminaux dans le membre droit. """
33
34     while any(
35         sum(1 for symbol in regle if symbol in self.non_terminaux) > 2
36         for membre_gauche, membre_droit in self.regles.items()
37         for regle in membre_droit
38     ):
39         self.iteration_suppression_regle_plus_deux_non_terminaux_membre_droite()
```

Figure 13: Fonction qui supprime des règles avec plus de deux non-terminaux dans le membre de droite

Dans cette fonction, on parcourt les règles afin de **supprimer celles contenant plus de deux non-terminaux dans le membre droit**. Pour cela, on vérifie le nombre de non-terminaux pour chaque règle. **Si une règle en contient plus de deux, on la modifie**. On utilise un compteur pour ignorer le premier, puis on stocke la suite de la règle dans un nouveau non-terminal non utilisé.

La fonction qui relance l'itération **vérifie si les conditions ne sont pas encore respectées**. Elle est **essentielle**, car de nouvelles règles problématiques peuvent être ajoutées au fil des modifications. **Cette relance garantit un nettoyage complet des règles de la grammaire**.

```
---- TEST SUPPRESSION REGLE PLUS DE DEUX NON TERMINAUX MEMBRE DROITE ----
Productions de la grammaire :
S1 -> U1 V1 W1 X1

APRES SUPPRESSION REGLE PLUS DE DEUX NON TERMINAUX MEMBRE DROITE
Productions de la grammaire :
S1 -> U1 R0
R0 -> V1 R2
R2 -> W1 X1
```

Figure 14: Test d'exécution

#### 4.4.6 Suppression des non-terminaux en tête des règles

```
1 def suppression_non_terminaux_en_tete(self):
2     """ Supprime les non-terminaux en tête des règles. """
3
4     for non_terminal, regles in list(self.regles.items()):
5         nouvelles_regles = []
6
7         for regle in regles:
8             if regle[0] in self.non_terminaux:
9                 for nouvelle_regle in self.regles[regle[0]]:
10                     nouvelles_regles.append(nouvelle_regle + regle[1:])
11             else:
12                 nouvelles_regles.append(regle)
13
14         self.regles[non_terminal] = nouvelles_regles
```

Figure 15: Fonction qui supprime les non-terminaux en tête des règles

Dans cette fonction, on parcourt les règles pour **supprimer celles dont le premier symbole est un non-terminal**. Si c'est le cas, on les remplace en ajoutant à la règle actuelle toutes les règles associées à ce non-terminal, tout en conservant la suite de la règle originale.

```
---- TEST SUPPRESSION NON TERMINAUX EN TETE ----

Productions de la grammaire :
S1 -> A1 a b | c A1 | B1 c | d
A1 -> a | b
B1 -> e | f

APRES SUPPRESSION NON TERMINAUX EN TETE

Productions de la grammaire :
S1 -> a a b | b a b | c A1 | e c | f c | d
A1 -> a | b
B1 -> e | f
```

Figure 16: Test d'exécution

#### 4.4.7 Suppression des symboles terminaux qui ne sont pas en tête des règles

```
1 def suppression_terminaux_non_en_tete(self):
2     """ Supprime les terminaux non en tête des règles. """
3
4     regles = list(self.regles.items())
5
6     for membre_gauche, membre_droit in regles:
7         for regle in membre_droit :
8             for i, symbol in enumerate(regle) :
9                 if symbol in self.terminaux and i > 0:
10                     nouveau_non_terminal = self.get_non_terminal_non_utilise()
11                     self.ajout_regle(nouveau_non_terminal, [symbol])
12                     regle[i] = nouveau_non_terminal
```

Figure 17: Fonction qui supprime les symboles terminaux non en tête des règles

Dans cette fonction, on parcourt les règles pour **supprimer les symboles terminaux qui ne sont pas en tête des règles**. Lorsqu'un terminal est trouvé dans une position autre que la tête, **il est remplacé par un non-terminal (non utilisé)**. Ce non-terminal possède une règle contenant le terminal concerné.

```
---- TEST SUPPRESSION TERMINAUX NON EN TETE ----

Productions de la grammaire :
S1 -> a A1 b | B1 c | E f g | h
A1 -> x | y
B1 -> z
D1 -> m | n

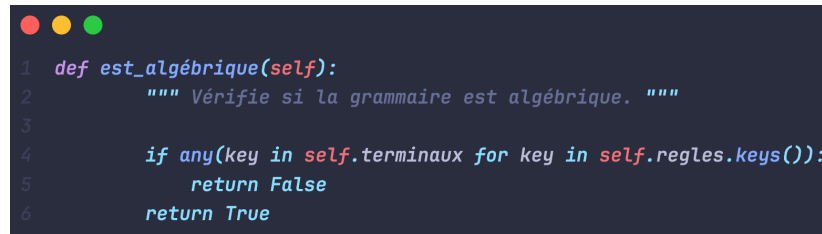
APRES SUPPRESSION TERMINAUX NON EN TETE

Productions de la grammaire :
S1 -> a A1 B6 | B1 O9 | E D6 N8 | h
A1 -> x | y
B1 -> z
D1 -> m | n
B6 -> b
O9 -> c
D6 -> f
N8 -> g
```

Figure 18: Test d'exécution

## 4.5 Les formes normales

Avant d'exécuter les simplifications dans l'ordre respectif des transformations, il faut nous assurer que la grammaire lue est algébrique (Forme des grammaires algébriques).



```
1 def est_algebrique(self):
2     """ Vérifie si la grammaire est algébrique. """
3
4     if any(key in self.terminaux for key in self.regles.keys()):
5         return False
6     return True
```

Figure 19: Fonction qui vérifie que la grammaire est algébrique

Ci-dessous, on présente les algorithmes implémentés ainsi que deux exemples d'exécution.

#### 4.5.1 Forme normale de Greibach

Simplement, on vérifie si la grammaire est bien algébrique puis on exécute les simplifications les unes après les autres (Étapes de transformation en forme normale de Greibach)

```
1 def transformation_greibach(self):
2     '''Forme normale de Greibach'''
3
4     if self.est_algebrique():
5         self.suppression_axiome_membre_droit()
6         self.suppression_epsilon()
7         self.suppression_regle_unite()
8         self.suppression_non_terminaux_en_tete()
9         self.suppression_terminaux_non_en_tete()
```

Figure 20: Algorithme de transformation en forme normale de Greibach

```
--- TEST GREIBACH n°1 ---

Productions de la grammaire :
S1 -> a S1 a | b S1 b | a | b | E

--- APRES GREIBACH ---

Productions de la grammaire :
F8 -> a S1 Y9 | b S1 Q5 | a | b | E | a A4 | b I6
S1 -> a S1 Y9 | b S1 Q5 | a | b | a T6 | b R4
Y9 -> a
Q5 -> b
T6 -> a
R4 -> b
A4 -> a
I6 -> b
```

Figure 21: Test n°1 de la transformation en forme normale de Greibach

```

--- TEST GREIBACH n°2 ---

Productions de la grammaire :
S1 -> a S1 b S1 | a | b | E

--- APRES GREIBACH ---

Productions de la grammaire :
C4 -> a S1 G3 S1 | a | b | E | a K1 S1 | a S1 T2 | a R3
S1 -> a S1 G3 S1 | a | b | a N6 S1 | a S1 M1 | a V3
G3 -> b
N6 -> b
M1 -> b
V3 -> b
K1 -> b
T2 -> b
R3 -> b

```

Figure 22: Test n°2 de la transformation en forme normale de Greibach



#### 4.5.2 Forme normale de Chomsky

De la même façon que pour Greibach, on exécute les simplifications les une après les autres (Étapes de transformation en forme normale de Chomsky)

```
1 def transformation_chomsky(self):
2     '''Forme normale de Chomsky'''
3
4     if self.est_algebrique() :
5         self.suppression_axiome_membre_droit()
6         self.suppression_terminaux()
7         self.suppression_regle_plus_deux_non_terminaux_membre_droite()
8         self.suppression_epsilon()
9         self.suppression_regle_unite()
```

Figure 23: Algorithme de transformation en forme normale de Chomsky

```
--- TEST CHOMSKY n°1 ---

Productions de la grammaire :
S1 -> a S1 a | b S1 b | a | b | E

--- APRES CHOMSKY ---

Productions de la grammaire :
T7 -> C5 C7 | J4 A3 | a | b | E
S1 -> C5 L8 | J4 G3 | a | b
C5 -> a
J4 -> b
L8 -> S1 C5 | a
G3 -> S1 J4 | b
C7 -> S1 C5 | a
A3 -> S1 J4 | b
```

Figure 24: Test n°1 de la transformation en forme normale de Chomsky

```
--- TEST CHOMSKY n°2 ---

Productions de la grammaire :
S1 -> a S1 b S1 | a | b | E

--- APRES CHOMSKY ---

Productions de la grammaire :
U7 -> F6 Q2 | a | b | E
S1 -> F6 J1 | a | b
F6 -> a
G4 -> b
J1 -> S1 T6 | G4 S1 | b
Q2 -> S1 F5 | G4 S1 | b
T6 -> G4 S1 | b
F5 -> G4 S1 | b
```

Figure 25: Test n°2 de la transformation en forme normale de Chomsky

## 4.6 Énumération des mots générés par les formes normales

Dans cette partie, on cherche à lister tout les mots que l'on peut obtenir depuis une grammaire, plus précisément tout les mots d'une longueur maximale (inférieure ou égale) donné (que l'on note  $n$ ).

Pour cela on implémente un algorithme qui effectue un parcours récursif, une approche de **parcours en profondeur** (DFS) sur les règles de la grammaire. Étant donné une grammaire et un mot généré par la grammaire  $w$ , met dans un ensemble 'langage' tous les mots générés par la grammaire à partir du mot

```
1 def contient_que_des_terminaux(self, w):
2     ''' Retourne un booléen indiquant si le mot w ne contient que des terminaux '''
3
4     return all(symbol in self.terminaux for symbol in w)
5
6 def enumere_mots(self, n, w, langage) :
7     ''' Génère les mots de longueur inférieure à n à partir de w '''
8
9     if len(w) > n :
10        return
11
12    if self.contient_que_des_terminaux(w) :
13        langage.add("".join(w))
14        return
15
16    for i in range(len(w)) :
17        if w[i] in self.non_terminaux :
18            for w3 in self.regles[w[i]] :
19                w2 = w[:i] + w3 + w[i+1:]
20
21                self.enumere_mots(n, w2, langage)
22
23 def enumere_mots_langage(self, n):
24     ''' Enumère les mots de longueur inférieure à n générés par la grammaire '''
25
26     langage = set()
27     self.enumere_mots(n, [self.axiome], langage)
28     if self.is_epsilon_generable():
29         langage.add("E")
30
31     return sorted(langage, key=lambda x: (len(x), x))
```

Figure 26: Algorithme d'énumération des mots issus d'une grammaire

**Remarque :** On vérifie si l'axiome peut produire  $\epsilon$  (le mot vide). Si c'est le cas, le mot vide est généré par la grammaire. Dans le cas contraire, la grammaire ne pourra pas générer le mot vide.

```

--- TEST ENUMERATION ---

--- GRAMMAIRE INITIALE ---
Productions de la grammaire :
S1 -> a S1 b S1 | a | b | E

--- APRES TRANSFORMATIONS ---
Les mots générés par la forme normale de Greibach : ['E', 'a', 'b', 'ab', 'aab', 'aba', 'abb', 'aaba', 'aabb', 'abab',
', 'abba', 'abbb', 'aaabb', 'aabab', 'aabbba', 'aabbba', 'abaab', 'ababa', 'ababb', 'abbab']

Les mots générés par la forme normale de Chomsky : ['E', 'a', 'b', 'ab', 'aab', 'aba', 'abb', 'aaba', 'aabb', 'abab',
', 'abba', 'abbb', 'aaabb', 'aabab', 'aabbba', 'aabbba', 'abaab', 'ababa', 'ababb', 'abbab']

Les deux formes génèrent les mêmes mots : True

```

Figure 27: Test d'exécution après deux transformations

#### 4.6.1 Limites rencontrées

Au cours de nos tests d'implémentation, nous avons observé un détail que l'on a trouvé **intéressant à explorer**.

Nous rencontrons une **borne de longueur maximale des mots** (notée  $n$ ) lors de l'énumération pour les deux formes normales :

- Après transformation en **forme normale de Greibach** pour un  $n \geq 12$ , l'algorithme d'énumération prend un temps **non déterminé**.
- Après transformation en **forme normale de Chomsky**, c'est pour un  $n \geq 10$ .

```
1 def contient_que_des_terminaux(self, w):
2     ''' Retourne un booléen indiquant si le mot w ne contient que des terminaux '''
3
4     return all(symbol in self.terminaux for symbol in w)
5
6 def enumere_mots(self, n, w, langage, fichier, niveau=0):
7     ''' Génère les mots de longueur inférieure à n à partir de w '''
8
9     if len(w) > n :
10         return
11
12     with open(fichier, "a") as f:
13         f.write(" " * niveau + "".join(w) + "\n")
14
15     if self.contient_que_des_terminaux(w) :
16         langage.add("".join(w))
17         return
18
19     for i in range(len(w)) :
20         if w[i] in self.non_terminaux :
21             for w3 in self.regles[w[i]] :
22                 w2 = w[:i] + w3 + w[i+1:]
23
24                 self.enumere_mots(n, w2, langage, fichier, niveau+1)
25
26 def enumere_mots_langage(self, n, fichier="mots_generees.txt"):
27     ''' Enumère les mots de longueur inférieure à n générés par la grammaire '''
28
29     langage = set()
30     if os.path.exists(fichier):
31         os.remove(fichier)
32     self.enumere_mots(n, [self.axiome], langage, fichier)
33     if self.is_epsilon_generable():
34         langage.add("E")
35
36     return sorted(langage, key=lambda x: (len(x), x))
```

Figure 28: Code revu générant l'arborescence

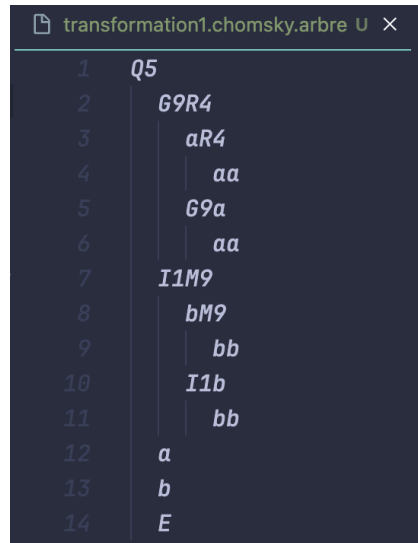
On **modifie légèrement** l'algorithme vu précédemment (exclusivement dans un but expérimental) afin de pouvoir stocker dans un fichier **.txt l'arborescence**

de la génération des mots.

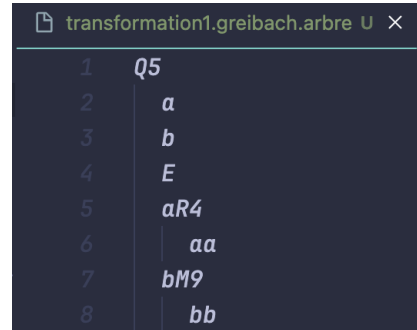
Nous pouvons donc **comparer** les arbres afin de voir quelle forme est **la plus optimisée** pour une grammaire donnée, nous travaillons sur l'exemple ci-dessous.

$S1 : aS1a \mid bS1b \mid a \mid b \mid E$

Figure 29: Grammaire de test



(a) Arbre obtenu après transformation en forme normale de Chomsky



(b) Arbre obtenu après transformation en forme normale de Greibach

Figure 30: Arborescences obtenues pour les mots de longueur 2

On choisi un petit  $n$  afin de faciliter l'affichage.

Nous pouvons remarquer très rapidement que la **forme normale de Greibach** est bien plus '**efficace**'. La **taille de l'arborescence** est presque divisée par 2 (quand on augmente le  $n$  on arrive à une complexité environ divisée par 10). Une telle différence nous semble logique au vu du **nombre de récursivité** présent après transformation en forme normale de **Chomsky** par rapport à la grammaire obtenue après transformation en forme normale de **Greibach**.

**Remarque :** Le code modifié permettant la **génération des fichiers des arborescences** est présent dans la branch 'arbre' sur GitHub.

## 4.7 Les commandes

Dans ce projet, on cherche à mettre à disposition **des commandes pour faciliter l'exécution des différents algorithmes**. Les utiliser sous forme de commandes est pertinent car cela permet d'avoir un moyen standardisé d'utiliser le code, sans devoir à chaque fois re-plonger dedans pour changer des variables. Nous utilisons le module **sys** de python qui permet de passer des arguments. On a également besoin d'une fonction qui va nous permettre **d'écrire dans des fichiers**, tout en respectant la syntaxe.



```
def ecrire(self, file):
    """ Écrit la grammaire dans un fichier texte. """
    with open(file, "w") as file:
        file.write(f'{self.axiome} : {" | ".join([" ".join(part) for part in self.regles[self.axiome]])}\n')
        for membre_gauche, membre_droit in self.regles.items():
            if membre_gauche != self.axiome:
                membre_droit = " | ".join([" ".join(part) for part in membre_droit])
                file.write(f'{membre_gauche} : {membre_droit}\n')
```

Figure 31: Fonction permettant d'écrire une grammaire depuis la structure de données vers un fichier

**Rappel : Le premier membre gauche du fichier est l'axiome**

#### 4.7.1 Commande 'grammaire'

```
1 import sys
2 from main import Grammaire
3 import os
4 def main():
5     if len(sys.argv) != 2:
6         print("Usage: python script.py <fichier_grammaire>")
7         sys.exit(1)
8
9     fichier_grammaire = sys.argv[1]
10
11     try:
12         # On choisi d'utiliser deux objets Grammaire différents (n'est pas obligatoire)
13         grammaire_chomsky = Grammaire()
14         grammaire_greibach = Grammaire()
15         grammaire_chomsky.lire(fichier_grammaire)
16         grammaire_greibach.lire(fichier_grammaire)
17
18         grammaire_chomsky.transformation_chomsky()
19         grammaire_chomsky.ecrire(os.path.splitext(os.path.basename(fichier_grammaire))[0] + ".chomsky")
20
21         grammaire_greibach.transformation_greibach()
22         grammaire_greibach.ecrire(os.path.splitext(os.path.basename(fichier_grammaire))[0] + ".greibach")
23
24     except FileNotFoundError:
25         print(f"Erreur : le fichier '{fichier_grammaire}' est introuvable.")
26         sys.exit(1)
27     except Exception as e:
28         print(f"Une erreur est survenue : {e}")
29         sys.exit(1)
30
31 if __name__ == "__main__":
32     main()
```

Figure 32: Exécutable permettant de générer un fichier pour chaque forme normale depuis une grammaire

Le script commence par vérifier le nombre d'arguments passés en ligne de commande, on attend **1 argument** (hormis le nom de l'exécutable) : un **fichier contenant la grammaire à traiter**.

Après avoir lu cette grammaire, le script **génère deux fichiers** (*test.chomsky* et *test.greibach*), chacun contenant la grammaire convertie respectivement **en forme normale de Chomsky** et **en forme normale de Greibach**.

Ces fichiers sont sauvegardés grâce au **chemin parsé**, permettant de s'assurer qu'ils se retrouvent à l'endroit où la commande a été exécutée.

Ci-dessous un exemple de ce que fait l'exécutable.

```
● ahmad@MacBook-Pro-de-Ahmad HATOUM_GUIBERT % ls
Makefile      binome.txt      generer          main.py
__pycache__   dossier_exemples  grammaire
● ahmad@MacBook-Pro-de-Ahmad HATOUM_GUIBERT % python3 grammaire dossier_exemples/transformation1.general
● ahmad@MacBook-Pro-de-Ahmad HATOUM_GUIBERT % ls
Makefile      dossier_exemples  main.py
__pycache__   generer           transformation1.chomsky
binome.txt    grammaire         transformation1.greibach
```

Figure 33: Test d'utilisation de l'exécutable **grammaire**



```

1 K5 : M8 D6 / R1 N3 / a / b / E
2 S1 : M8 Z1 / R1 K4 / a / b
3 M8 : a
4 R1 : b
5 Z1 : S1 M8 / a
6 K4 : S1 R1 / b
7 D6 : S1 M8 / a
8 N3 : S1 R1 / b

```

(a) Contenu du fichier  
*transformation1.chomsky*

```

1 K5 : a S1 M8 / b S1 R1 / a / b / E / a D6 / b N3
2 S1 : a S1 M8 / b S1 R1 / a / b / a Z1 / b K4
3 M8 : a
4 R1 : b
5 Z1 : a
6 K4 : b
7 D6 : a
8 N3 : b

```

(b) Contenu du fichier  
*transformation1.greibach*

Figure 34: Exemple après exécution

**Remarque :** Ici **K5** est l'axiome après simplification (**initialement S1** dans le fichier d'entrée).

#### 4.7.2 Commande 'generer'

```
1 import sys
2 from main import Grammaire
3
4 def main():
5     if len(sys.argv) != 3:
6         print("Usage: python3 generer.py <nombre_max_mots> <fichier_grammaire>")
7         sys.exit(1)
8
9     try:
10         nombre_max_mots = int(sys.argv[1])
11         fichier_grammaire = sys.argv[2]
12
13         grammaire = Grammaire()
14         grammaire.lire(fichier_grammaire)
15
16         print("\033c")
17
18         print(f"\nGénération des mots (longueur maximale {nombre_max_mots}):")
19         mots = grammaire.enumere_mots_langage(nombre_max_mots)
20         print("\n".join(mots))
21
22     except FileNotFoundError:
23         print(f"Erreur : le fichier '{fichier_grammaire}' est introuvable.")
24         sys.exit(1)
25     except ValueError:
26         print("Erreur : le premier argument doit être un nombre entier valide pour la longueur maximale des mots.")
27         sys.exit(1)
28     except Exception as e:
29         print(f"Une erreur est survenue : {e}")
30         sys.exit(1)
```

Figure 35: Exécutable permettant de générer un fichier contenant tout les mots d'une longueur donnée d'une grammaire

Ce script attend **2 arguments en ligne de commande** (hormis le nom de l'exécutable) : **un fichier contenant la grammaire** à traiter et **un entier** (que l'on note ici  $n$ ) **représentant le nombre maximum de lettre dans un mots**.

De la même manière, le fichier en entrée est lu, et l'algorithme d'énumération de mots est lancé.

Le fichier en sortie **contient tous les mots de longueur inférieure ou égale à  $n$**  générés par la grammaire contenue dans le fichier, ils sont triés dans l'ordre lexicographique, un par ligne et sans espace.

**Remarque** : Par défaut, la sortie est standard et aucun fichier n'est créé.

Pour rediriger la sortie vers un fichier, il suffit d'utiliser **>** (**sous Linux**)

On reprend dans la ligne de commande où l'on c'était arrêté pour tester.

```

--- TEST ENUMERATION ---

--- GRAMMAIRE INITIALE ---
Productions de la grammaire :
S1 -> a S1 b S1 | a | b | E

--- APRES TRANSFORMATIONS ---
Les mots générés par la forme normale de Greibach : ['E', 'a', 'b', 'ab', 'aab', 'aba', 'abb', 'aaba', 'aabb', 'abab',
', 'abba', 'abbb', 'aaabb', 'aabab', 'aabba', 'aabb', 'abaab', 'ababa', 'ababb', 'abbab']

Les mots générés par la forme normale de Chomsky : ['E', 'a', 'b', 'ab', 'aab', 'aba', 'abb', 'aaba', 'aabb', 'abab',
', 'abba', 'abbb', 'aaabb', 'aabab', 'aabba', 'aabb', 'abaab', 'ababa', 'ababb', 'abbab']

Les deux formes génèrent les mêmes mots : True

```

Figure 36: Test d'exécution de l'exécutable **generer**

On observe bien la création de nouveaux fichiers  
(*test\_5\_chomsky.res* et *test\_5\_greibach.res*)

```

1 %c
2
3 Génération des mots (longueur maximale 5):
4 E
5 a
6 b
7 aa
8 bb
9 aaa
10 aba
11 bab
12 bbb
13 aaaa
14 abba
15 baab
16 bbbb
17 aaaaa
18 aabaa
19 ababa
20 abbba
21 baaab
22 babab
23 bbabb
24 bbbbb

```

(a) Contenu du fichier  
*test\_5\_chomsky.res*

```

1 %c
2
3 Génération des mots (longueur maximale 5):
4 E
5 a
6 b
7 aa
8 bb
9 aaa
10 aba
11 bab
12 bbb
13 aaaa
14 abba
15 baab
16 bbbb
17 aaaaa
18 aabaa
19 ababa
20 abbba
21 baaab
22 babab
23 bbabb
24 bbbbb

```

(b) Contenu du fichier  
*test\_5\_greibach.res*

Figure 37: Exemple après exécution

Nous pouvons désormais vérifier si l'ensemble du programme **fonctionne correctement**, c'est-à-dire **comparer les résultats des algorithmes de transformation** en forme normale de Chomsky et de Greibach pour s'assurer qu'ils produisent le même ensemble de mots.

Cette vérification peut être effectuée à l'aide de la **commande Linux diff**, qui affiche les **différences** si les **fichiers sont différents**, ou **rien** si les **fichiers sont identiques**.

```

● ahmad@MacBook-Pro-de-Ahmad HATOUM_GUIBERT % diff test_5_chomsky.res test_5_greibach.res
○ ahmad@MacBook-Pro-de-Ahmad HATOUM_GUIBERT %

```

Figure 38: Exemple de comparaison

Nous pouvons en conclure que le programme marche parfaitement, les mots générés sont les mêmes après les deux transformations. On peut simuler le cas contraire en comparant deux fichiers indépendants.

```

● ahmad@MacBook-Pro-de-Ahmad HATOUM_GUIBERT % ls
Makefile      binome.txt      generer          main.py
__pycache__   dossier_exemples  grammaire
● ahmad@MacBook-Pro-de-Ahmad HATOUM_GUIBERT % python3 grammaire dossier_exemples/transformation1.general
● ahmad@MacBook-Pro-de-Ahmad HATOUM_GUIBERT % python3 grammaire dossier_exemples/transformation2.general
● ahmad@MacBook-Pro-de-Ahmad HATOUM_GUIBERT % ls
Makefile      dossier_exemples  main.py          transformation2.chomsky
__pycache__   generer           transformation1.chomsky  transformation2.greibach
binome.txt    grammaire        transformation1.greibach
● ahmad@MacBook-Pro-de-Ahmad HATOUM_GUIBERT % python3 generer 5 transformation1.chomsky > test_5_chomsky.res
● ahmad@MacBook-Pro-de-Ahmad HATOUM_GUIBERT % python3 generer 5 transformation2.greibach > test_5_greibach.res

```

```

● ahmad@MacBook-Pro-de-Ahmad HATOUM_GUIBERT % diff test_5_chomsky.res test_5_greibach.res
7,9c7,8
< aa
< bb
< aaa
---
> ab
> aab
11,13c10,13
< bab
< bbb
< aaaa
---
> abb
> aaba
> aabb
> abab
15,18c15,20
< baab
< bbbb
< aaaaa
< aabaa
---
> abbbb
> aaabb
> aabab
> aabbba
> aabbbb
> abaab
20,24c22,23
< abbbba
< baabab
< bbaabb
< bbbbbb
---
> ababb
> abbab

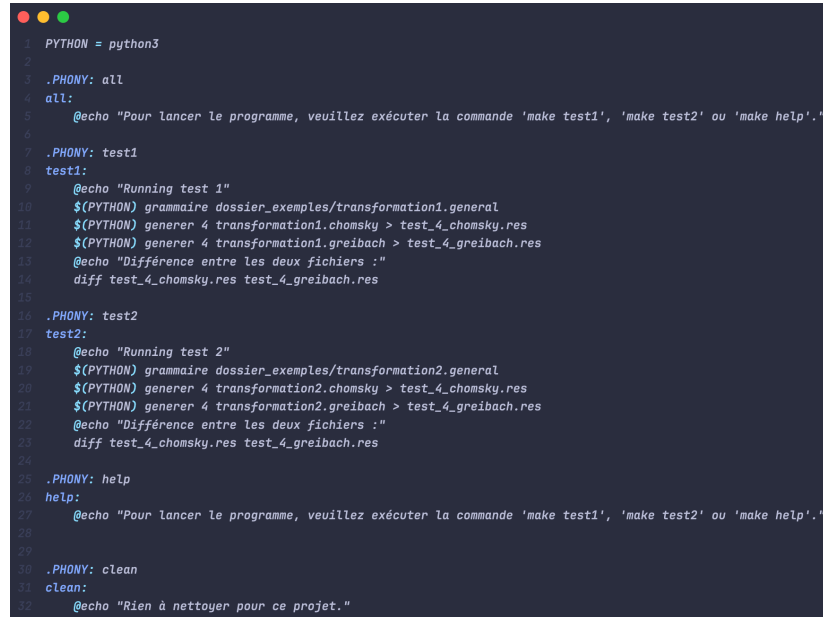
```

Figure 39: Exemple de cas où les deux transformations ne génèrent pas les mêmes mots

### 4.7.3 Makefile

Le fichier **Makefile** simplifie l'exécution des exécutables (grammaire et generer) via la commande **make**.

Dans notre fichier, nous avons deux tests pour **vérifier** le bon fonctionnement des algorithmes implémentés.

A screenshot of a terminal window with a dark background and light-colored text. The terminal shows the content of a Makefile. The first line is 'PYTHON = python3'. The next line is '.PHONY: all'. The 'all' target has a message: '@echo "Pour lancer le programme, veuillez exécuter la commande \'make test1\', \'make test2\' ou \'make help\'.\"'. The 'test1' target has a message: '@echo "Running test 1\"'. It then runs '\$(PYTHON) grammaire dossier\_exemples/transformation1.general', '\$(PYTHON) generer 4 transformation1.chomsky > test\_4\_chomsky.res', '\$(PYTHON) generer 4 transformation1.greibach > test\_4\_greibach.res', and a diff command: '@echo "Différence entre les deux fichiers :\"' followed by 'diff test\_4\_chomsky.res test\_4\_greibach.res'. The 'test2' target has a message: '@echo "Running test 2\"'. It runs '\$(PYTHON) grammaire dossier\_exemples/transformation2.general', '\$(PYTHON) generer 4 transformation2.chomsky > test\_4\_chomsky.res', '\$(PYTHON) generer 4 transformation2.greibach > test\_4\_greibach.res', and a diff command: '@echo "Différence entre les deux fichiers :\"' followed by 'diff test\_4\_chomsky.res test\_4\_greibach.res'. The 'help' target has a message: '@echo "Pour lancer le programme, veuillez exécuter la commande \'make test1\', \'make test2\' ou \'make help\'.\"'. The 'clean' target has a message: '@echo "Rien à nettoyer pour ce projet.\"'.

```
1 PYTHON = python3
2
3 .PHONY: all
4 all:
5     @echo "Pour lancer le programme, veuillez exécuter la commande 'make test1', 'make test2' ou 'make help'."
6
7 .PHONY: test1
8 test1:
9     @echo "Running test 1"
10    $(PYTHON) grammaire dossier_exemples/transformation1.general
11    $(PYTHON) generer 4 transformation1.chomsky > test_4_chomsky.res
12    $(PYTHON) generer 4 transformation1.greibach > test_4_greibach.res
13    @echo "Différence entre les deux fichiers :\"
14    diff test_4_chomsky.res test_4_greibach.res
15
16 .PHONY: test2
17 test2:
18     @echo "Running test 2"
19    $(PYTHON) grammaire dossier_exemples/transformation2.general
20    $(PYTHON) generer 4 transformation2.chomsky > test_4_chomsky.res
21    $(PYTHON) generer 4 transformation2.greibach > test_4_greibach.res
22    @echo "Différence entre les deux fichiers :\"
23    diff test_4_chomsky.res test_4_greibach.res
24
25 .PHONY: help
26 help:
27     @echo "Pour lancer le programme, veuillez exécuter la commande 'make test1', 'make test2' ou 'make help'."
28
29
30 .PHONY: clean
31 clean:
32     @echo "Rien à nettoyer pour ce projet."
```

Figure 40: Contenu du fichier Makefile

Nous définissons une **variable globale** (qui sert à définir l'interpréteur python).

Le fichier contient deux commandes de base pour **guider l'utilisation** **all** et **help**, qui affichent un message d'aide. Les commandes **test1** et **test2** permettent **de lancer les différents algorithmes** sur les deux grammaires que nous avons choisies de tester.

**Pour utiliser une commande**, il faut simplement faire **make** suivi du nom de la commande souhaitée.

```

● ahmad@MacBook-Pro-de-Ahmad HATOUM_GUIBERT % ls
Makefile          binome.txt          generer             main.py
__pycache__       dossier_exemples   grammaire
● ahmad@MacBook-Pro-de-Ahmad HATOUM_GUIBERT % make help
Pour lancer le programme, veuillez exécuter la commande 'make test1', 'make test2' ou 'make help'.
● ahmad@MacBook-Pro-de-Ahmad HATOUM_GUIBERT % make test1
Running test 1
python3           grammaire dossier_exemples/transformation1.general
python3           generer 4 transformation1.chomsky > test_4_chomsky.res
python3           generer 4 transformation1.greibach > test_4_greibach.res
Différence entre les deux fichiers :
diff test_4_chomsky.res test_4_greibach.res
○ ahmad@MacBook-Pro-de-Ahmad HATOUM_GUIBERT %
● ahmad@MacBook-Pro-de-Ahmad HATOUM_GUIBERT % make test2
Running test 2
python3           grammaire dossier_exemples/transformation2.general
python3           generer 4 transformation2.chomsky > test_4_chomsky.res
python3           generer 4 transformation2.greibach > test_4_greibach.res
Différence entre les deux fichiers :
diff test_4_chomsky.res test_4_greibach.res
○ ahmad@MacBook-Pro-de-Ahmad HATOUM_GUIBERT %
● ahmad@MacBook-Pro-de-Ahmad HATOUM_GUIBERT % ls
Makefile          grammaire           transformation1.greibach
__pycache__       main.py             transformation2.chomsky
binome.txt        test_4_chomsky.res  transformation2.greibach
dossier_exemples test_4_greibach.res
generer           transformation1.chomsky

```

Figure 41: Test d'exécution du Makefile