UVSQ **2024-2025**

Licence 3 Informatique

Projet – Transformation de grammaires (LSIN520)

Le projet a été réalisé en binôme :

- **MACHTER Massinissa : 22304222 (TD1)**
- **KHAZEM Lynda : 22304263 (TD1)**

1. Présentation :

Le projet consiste à lire une grammaire algébrique à partir d'un fichier, puis à la transformer en deux formes normales : la **forme normale de Greibach** (FNG) et la **forme normale de Chomsky** (FNC). Ensuite, pour chacune de ces formes transformées, le programme doit générer tous les mots de la grammaire dont la longueur est inférieure ou égale à une longueur spécifiée.

- Le but de ce projet est de programmer différentes transformations entre des grammaires algébriques puis d'utiliser ces grammaires en formes normales pour générer des mots de leur langage.
- Le projet a été réalisé en Python.

2. Structure de données :

La classe **Grammaire** est conçue pour modéliser une grammaire formelle, en particulier une grammaire libre de contexte, dans le but de gérer des règles de production, l'axiome, d'analyser des symboles terminaux et non-terminaux, et de permettre diverses opérations sur ces règles. Voici une explication détaillée de la structure de cette classe :

- **Attributs de classe (constantes):**

  - **tous_terminaux** : Il s'agit d'une liste contenant tous les symboles terminaux possibles, ici représentés par les lettres minuscules de l'alphabet (a à z).

- **tous_les_non_terminaux** : Un ensemble contenant tous les non-terminaux possibles (250), qui sont constitués d'une lettre majuscule suivie d'un chiffre, excepté "E" pour epsilon. Cela génère des symboles comme "A1", "B2", etc.

- **Méthodes principales :**

  - **__init__** : Le constructeur prend en entrée un axiome (le symbole de départ de la grammaire) et un dictionnaire de règles de production. Il initialise également les ensembles des symboles terminaux et non-terminaux en appelant la méthode 'extraire_symboles'.

  - **copy** : Cette méthode crée une copie indépendante de l'instance actuelle de la grammaire, en dupliquant l'axiome et les règles de production. Cela permet de travailler sur une version séparée de la grammaire sans altérer l'originale.
  - **extraire_symboles** : Elle parcourt les règles de production pour extraire les symboles terminaux et non-terminaux utilisés dans la grammaire. Les terminaux sont représentés par des lettres minuscules, tandis que les non-terminaux sont des symboles constitués d'une lettre majuscule suivie d'un chiffre.
  - **extraire_symboles_de_production** : Cette méthode découpe une production donnée en une liste de symboles terminaux et non-terminaux. Par exemple, la production "aA1B2" sera découpée en ["a", "A1", "B2"].
  - **ajout_regles** : Cette méthode permet d'ajouter des règles de production pour un non-terminal donné. Si le non-terminal n'existe pas encore dans la grammaire, il est ajouté avec les nouvelles règles associées.
  - **identifier_annulables** : Elle identifie les non-terminaux annulables, c'est-à-dire ceux qui peuvent dériver en epsilon. Cela se fait en examinant les productions et en propagent l'annulabilité des symboles au fil des dérivations.
  - **variable_dispo** : Cette méthode retourne la prochaine variable non-terminale disponible, c'est-à-dire celle qui n'a pas encore été utilisée dans les règles de production.
  - **ecrire** : Elle permet d'écrire la grammaire dans un fichier de sortie spécifié, en respectant la structure des règles de production et en commençant par l'axiome.
  - **__str__** : Cette méthode permet de retourner une représentation lisible de la grammaire sous forme de chaîne de caractères. Elle affiche les terminaux, les non-terminaux, l'axiome, et les règles de production de manière formatée.

3. Différents algorithmes de transformation d'une grammaire :

Les algorithmes de transformation d'une grammaire sont tous implémentés dans le module **funct.py .**Soit une grammaire algébrique générale donnée, où, dans tous ses algorithmes de transformation, les lettres **minuscules** représentent les **terminaux**, tandis que les lettres **majuscules** désignent les **non-terminaux** (ou variables) :

- START : Suppression de l'axiome dans les membres droits de règles :

Pour cela on introduit un nouveau symbole S0 qui devient le nouvel axiome et on ajoute la règle :

$$S0 \rightarrow S$$

Ou S est l'ancien axiome. Ceci ne modifie pas le langage engendré, et la variable S0 donc ne figure jamais dans la partie droite des règles de productions.

- TERM : Suppression des lettres terminales dans les membres droits de règles de longueur au moins 2

Si une lettre terminale « a » figure dans un membre droit de règle de **longueur au moins 2**

(X→Y1…a…Yn), on la remplace par une nouvelle variable (non terminale) N avec:

  - Ajout de la règle : **N→a**
  - Remplacer a dans toutes les règles de longueur au moins 2 par « **N** » .
  - Par exemple ici : On remplace la règle **X→Y1…a…Yn** par **X→Y1…N…Yn**

Nb : Cette opération augment le nombre des règles de productions d'au plus le nombre de lettre terminales dans la grammaire.

- BIN : Suppression des membres droits avec plus de deux symboles non terminaux :

Dans cette étape, On remplace une règle **X→Y1Y2…..Yn** par :

  - Les règles : **X→Y1X1, X1→Y2X2,…, Xn-2→Yn-1Yn**
    Ou les Xi sont les nouveau Non terminaux ajoutes.
  - Exemple : **A→BCDE** alors ca devient : **A→BX1, X1→CX2, X2→DE**

Nb : cette opération augment le nombre des règles de productions d'au plus de triple.

- DEL : Élimination des ε-règles :

On commence par déterminer les variables (sauf l'axiome) qui dérivent en ε ; ces variables, appelées *annulables.*

- Par récurrence : *X* est annulable s'il existe une règle **X→Y1Y2…Yn** telle que **Y1,Y2,….,Yn** sont annulables.
- **Alors l'algo consiste à faire** : Pour tout variable **X** annulable ou non, toute règle **X→Y1…Yn**, est remplacée par toutes les règles obtenues en supprimant une ou plusieurs, voire toutes les variables annulables dans le membre droit de règle, puis a la fin on supprime toutes les ε-règles (à l'exception de celle de l'axiome si elle est présente)

- UNIT : Suppression des règles unité

Une *règle unité* est une règle de la forme X→Y, Pour éliminer ce type de règles:

  - On la remplace par la règle par : X→ α, pour chaque règle Y→ α
  - α mot

Sauf si c'est une règle unité précédemment enlevée.

Nb : La transformation **UNIT** peut faire passer la taille de la grammaire de $|G|$ à $|G|^2$

- DelVarHead : Suppression des non-terminaux en tête des règles

Ici on suppose que la grammaire sans ε-règles et sans règles UNIT, alors on procède par 2 étapes pour transformer la grammaire à une grammaire équivalente :

- **[ETP01]** ***Suppression de la récursivité gauche*** *:* Pour toute règle **X→Xα|β**, on remplace cette règle par l'introduction d'une nouvelle variable **X'** :

  X→Xα|β **≡** **X→ βX' | β**
  X' →αX' |α

- **[ETP02]** ***Suppression des non terminaux en tête*** *:* les occurrences de variables qui figurent en tête dans les membres droits de règles sont remplacées par l'ensemble des règles de ces variables.

- TermHead : Suppression des terminaux qui ne sont pas en tête des règles

Ici pour chaque terminal apparaissant dans les membres droits des règles autre qu'en tête, est remplacé par une variable supplémentaire $\mathbf{X_a}$, une pour chaque terminale **a**, avec la règle $\mathbf{X_a \rightarrow a}$.

4. Formes normales de grammaire algébrique :

Conversion d'une grammaire algébrique générale en forme normale :

La plus petite augmentation de taille des règles de productions se produit pour les ordres d'application des transformations suivantes :

- **FN-Chomsky:** START→ TERM → BIN → DEL→ UNIT.
- **FN-Greibach:** START→ DEL → UNIT → DelVarHead → TermHead.

Les fonctions de conversion vers les formes normales de Chomsky (**fct chomsky**) et de Greibach (**fct greibach**) sont implémentées dans le module **grammaire.py**, en exploitant les algorithmes de transformation implémentés dans le module **funct.py.**

5. Génération de mots de longueur inférieure ou égale à n:

Le module **generer.py** implémente un algorithme permettant de générer tous les mots de longueur inférieure ou égale à un nombre donné, en utilisant les règles d'une grammaire. L'algorithme part de l'axiome, applique récursivement les règles de production et génère les mots possibles. Il utilise la programmation dynamique pour mémoriser les résultats déjà calculés et éviter les redondances. À la fin, les mots générés sont triés par ordre lexicographique**.**

6. Démo:

En CLI :

- Tout d'abord, la commande suivante charge la grammaire à partir du fichier **test.general** et génère les formes normales Chomsky (FNC) et Greibach (FNG) dans des fichiers dédiés (**test.chomsky**, **test.greibach**)

  - **python grammaire.py test.general**

- Ensuite, pour générer les mots de longueur inférieure ou égale à n (entier) à partir des grammaires en forme normales, on utilise la commande suivante, et les résultats sont sauvegardés dans le fichier test_n_NomForme.res , ici pour n=5 :

  - **python generer.py 5 test.chomsky > test_5_chomsky.res**
  - **python generer.py 5 test.greibach > test_5_greibach.res**

- Enfin, la commande **diff** permet de comparer les deux fichiers générés et d'afficher les différences entre eux. Dans ce cas, **diff** ne doit rien afficher:
  - **diff test_4_chomsky.res test_4_greibach.res**

En utilisant la commande make :

Avec le Makefile, voici comment simplifier le processus :

- Pour traiter un fichier spécifique, comme **test.general** :
  - **make test**
  - Cette commande : Exécute « **python grammaire.py test.general** » pour charger la grammaire et les FN. Puis « **python generer.py 4 test.chomsky > test_4_chomsky.res** » (avec LENGTH=4 « peut être changé »).Ensuite « **python generer.py 4 test.greibach > test_4_greibach.res** ». Et enfin Compare les fichiers générés avec diff.

- Pour traiter **tous** les fichiers **.general** présents dans le répertoire :
  - **make**
  - Cette commande : Effectue automatiquement les étapes ci-dessus pour **chaque fichier .general** détecté dans le répertoire.