# TP2 - MISE SOUS FORME NORMALE DE CHOMSKY

## 1. Langages formels - TP 2

1.1. **Objectif du TP.** L'objectif de ce deuxième TP est d'implémenter un programme prenant comme argument un fichier `.txt` contenant la description d'une grammaire algébrique $\mathcal{G}$. Votre programme doit :

[1] lire/charger la grammaire $\mathcal{A}$ à partir du fichier `.txt`,

[2] mettre $G$ sous forme normale de Chomsky

**Dépôt de fichiers :** Si vous n'arrivez pas à finir le TP pendant le créneau de TP, il faudra finir en dehors, avant le prochain TP, pour éviter de prendre du retard.
Vous pouvez travailler seul ou en binôme, dans ce cas pensez à bien préciser le nom des deux étudiants. Les rendus des 3 TP sont à déposer sur Amétice. Seul le rendu du TP3, qui contiendra toutes les fonctions implémentées au cours des 3 séances, sera noté.

1.2. **Format de fichier `.txt`.** Une grammaire décrite dans un fichier .txt (UTF-8) consiste en une suite de lignes terminées par `\n` (CR), où chaque ligne est une paire. Les deux éléments sont séparés par un espace. La paire $S$ $\alpha$ correspond à une règle qui transforme le non-terminal (appelé variable) $S$ en $\alpha$ qui consiste en une suite de chaines de caractères séparés par des points (`.`). Par convention, il est recommandé de représenter les lettres avec des minuscules non-accentuées et les variables par des lettres majuscules. Le symbole spécial % est utilisé pour les règles produisant $\varepsilon$. La première ligne désigne la variable initiale.

Voici un exemple de grammaire décrite dans un fichier. Cette grammaire génère $\{a^n b^n | \ n \geq 1\}$.

```
S a.S.b
S %
```

1.3. **Bibliothèque `automaton.py` fournie.** Une nouvelle version de la bibliothèque python automaton.py est disponible sur le github du cours. Elle contient une nouvelle classe 'Grammar' et contient les fonctionnalités suivantes :

- création et manipulation d'un objet Grammar qui représente une grammaire algébrique,
- lecture et écriture à partir d'un fichier textuel,
- transformer un automate à pile en grammaire équivalente

Vous devez bien comprendre le fonctionnement de cette classe 'Grammar'. Les prochaines sections vous permettent de vous familiariser avec cette classe. Si vous êtes à l'aise en python, vous pouvez ouvrir le code de la bibliothèque pour le lire et même l'augmenter si vous le souhaitez. Vous pouvez aussi n'utiliser qu'une partie des fonctionnalités fournies. Pour commencer, vous devez télécharger la nouvelle version de la bibliothèque automaton.py, qui doit être placée dans le même dossier où se trouve votre script/programme source. Ensuite, vous pouvez créer une nouvelle grammaire et lui donner un nom. Cette grammaire, pour le moment, sera vide (vérifiez en exécutant le code ci-dessous) :

```
import automaton
g_test = Grammar("g_test")
```

Vous pouvez ensuite ajouter des variables et des règles. La source de la première règle sera considérée comme la variable initiale.

```
g_test.reset()
g_test.add_rule("S",["a","S","b"])
```

Notez que cet grammaire ne génère aucun mot, car toute règle produit au moins une variable. Pour y remédier, vous pouvez ajouter une règle supplémentaire :

```
g_test.add_rule("S",[])
```

Vous pouvez afficher la grammaire sous la forme textuelle à n'importe quel moment dans votre code, par exemple, pour déboguer. Notez que les règles de productions sont listée par variable et les règles correspondant à une même variable sont séparées par un symbole '|' :

```
print(g_test)
```

```
g_test = <A={a,b}, N={S}, P, S0=S>
P =
S -> a.S.b | ε
```

Dans cet exemple, `A` contient les lettres `a`,`b`, `N` contient seulement la variable `S` qui est donc la variable initiale, et la liste `P` des règles de productions est donnée dessous. Ces valeurs sont accessibles aussi directement, sous la forme de listes ou de chaînes de caractères :

```
print(g_test.alphabet)
print(g_test.variables)
print(g_test.initial)
```

Vous pouvez aussi afficher votre grammaire sous la forme de fichier .txt :

```
print(g_test.to_txtfile("test.txt"))
```

Une grammaire peut être réinitialisée à tout moment :

```
g_test.reset()
```

Vous pouvez aussi construire unz grammaire à partir d'une chaîne de caractères directement à l'aide de la fonction `from_txt` :

```
source = """S a.S.B
S %
"""
anbn = Grammar("anbn")
anbn.from_txt(source)
```

Les symboles du membre droit de la règle sont séparés par des points (`.`). Les productions $\varepsilon$ sont représentées par le symbole spécial `%` (constante `automaton.Automaton.EPSILON`). Essayez de modifier `source` et observez le résultat dans la grammaire. Ajouter ou supprimez des règles, des nouvelles variables.

Vous pouvez aussi lire une grammaire en passant directement le nom du fichier à la fonction `from_txtfile` (n'oubliez pas de charger le fichier si vous êtes sur Colab) :

```
anbn.from_txtfile("test/anbn.gr")
```

Vous aurez besoin d'accéder à la liste de variables et de règless de la grammaire. Chaque varaible est dans un dictionnaire python indexé par son nom, et contient une liste de règle. Les règles sont des listes contenant des chaines de caractères. Prenez le temps de bien comprendre cette structure, cela vous aidera à mieux vous servir de la bibliothèque lors des TP.
Alternativement, utilisez l'attribut rules de l'automate, qui donne la liste des règles sous la forme d'une `list` python simple. Chaque règle est un paire : (source, production). Le terme source est de type `str` et production est une liste de `str`. Cela peut être pratique, si vous voulez manipuler des objets plus simples que les dictionnaires et objets Variable.

```
for (source, prod) in a.rules : # et non pas a.rules()
print( "{} --> {}".format(source,"".join(prod)))
```

Vous trouverez d'autres exemples d'usage de la bibliothèque dans `automaton.py` à la toute fin du code-source.

## 2. Travail à effectuer

Votre travail consiste à implémenter deux fonctionnalités :

(1) Écrire une fonction qui prend une grammaire en entrée et qui renvoie un booléen pour indiquer si, oui ou non, la grammaire est sous forme normale de Chomsky

(2) Écrire une fonction qui prend une grammaire et la met sous forme normale de Chomsky

Pour implémenter la mise sous forme normale de Chomsky, il est conseillé de décomposer votre fonction en 5 fonctions : `chomsky_step1, chomsky_step2, chomsky_step3, chomsky_step4, chomsky_step5` comme vu en cours. Chacune de ces fonctions doit commencer par vérifier que l'étape est nécéssaire puis l'exécuter si besoin.

— `chomsky_step1`

Si la variable initiale `S` apparait dans le terme droit d'une règle, ajouter un nouvel axiome `S0` (bien vérifier que ce nom n'est pas déjà pris, et en choisir un autre sinon) ainsi qu'une nouvelle règle `S0 -> S`.

— `chomsky_step2`

Pour chaque règle produisant au moins deux symboles, remplacer chaque occurence de lettre `a` par une nouvelle variable `Xa` (vérifier que ce nom n'est pas déjà pris). Ajouter également une règle `Xa -> a` pour chaque lettre.

— `chomsky_step3`

Chaque règle produisant au moins trois symboles doit être remplacée par plusieurs règles ne produisant que deux symboles. Une solution pour implémenter cette étape est d'écrire une fonction qui réduit d'au moins 1 la longeur des membres droits (de longeur au moins 3), et d'utiliser une boucle `while` pour répéter cette étape tant que nécéssaire.

— `chomsky_step4`

Pour implémenter cette étape, il faut premièrement identifier les variables annulables.

On peut initialiser une liste contenant toutes les variables produisant $\varepsilon$ en une étape. Puis, toute variable possédant une règle produisant seulement des variables annulables est ajoutée à la liste des variables annulables (si elle n'y est pas déjà). On répète cette étape tant que la liste des variables annulables augmente.

Maintenant que les variables annulables sont identifiées il faut ajouter des nouvelles règles en supprimant des variables annulables dans les membres droits des règles. Comme pour l'étape précédente, on peut considérer le nombre $n$ de variables annulables maximal apparaissant dans un membre droit et implémenter une fonction qui diminue de au moins 1 ce nombre. Enfin, à l'aide d'une boucle while on répète cette fonction jusqu'à ce que $n = 0$.

Finalement, on peut retirer toutes les règles $\varepsilon$ (sauf éventuellement celle de la variable initiale).

— `chomsky_step5`

Pour implémenter cette fonction vous pouvez définir un dictionnaire `unitary = OrderedDict()` qui à chaque nom de variable `X` associe la liste `unitary[X]` des variables atteignables en utilisant seulement des règles unitaires. On commence par définir `unitary[X]=[]` pour toutes les variables. On peut initialiser ce dictionnaire par `unitary[X].append(Y)` pour chaque règle unitaire `X -> Y`. Puis, pour chaque variables `X,Y` si `Y` est dans `unitary[X]` on ajoute tous les éléments de `unitary[Y]` à la liste `unitary[X]` (attention à éviter les répétitions !). On répète cette étape tant que le dictionnaire grandit.

Une fois le dictionnaire construit, pour chaque variables `Y` dans `unitary[X]` on ajoute toutes les règles de `Y` à la variable `X`.

Enfin, on peut retirer toutes les règles unitaires.

Deux fichiers de grammaires sont déjà fournis, `test/anbn.gr` ainsi que `test/astarbstar.gr`. Vous devez non seulement implémenter la fonction mais aussi la tester. Pour cela, écrivez 3 grammaires au format textuel pour étendre la base de tests au délà des fichiers fournis. Vous pouvez utiliser des grammaires vues en cours ou en TD. Ces grammaires doivent être placées dans le dossier test et formeront votre base de tests, qui grandira au fur et à mesure des TPs. Pour déboguer votre programme, n'oubliez pas que vous pouvez à tout moment appeler `print(a)` pour afficher la grammaire sur le terminal. En python, vous pouvez/devez aussi utiliser la bibliothèque pdb pour le débogage, et vous pouvez aussi typer votre programme et vérifier les types à l'aide de mypy.