TP1 - EXÉCUTION D'UN AUTOMATE À PILE

1. Langages formels - TP 1

1.1. **Objectif du TP.** L'objectif de ce premier TP est d'implémenter un programme avec deux arguments : un fichier `.txt` contenant la description d'un automate à pile $\mathcal{A}$ et un mot w à reconnaître. Votre programme doit :

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

[2] afficher ERROR si l'automate est non déterministe, ou

[3] si, au contraire, l'automate est déterministe, afficher YES si le mot est accepté par l'automate, ou NO si le mot n'est pas accepté par l'automate.

1.2. **Environnement de travail.** Les besoins de votre environnement de travail sont les mêmes que lors des TP de l'UE Automates Finis du premier semestre.

Programmation python. Une fois que vous aurez lu l'énoncé, vous devrez écrire un programme pour répondre à l'objectif. La programmation doit se faire en python car nous vous fournissons une bibliothèque de base (cf. ci-dessous). Pour programmer, utilisez l'environnement que vous préférez. Vous pouvez utiliser une IDE installée, telle que intelliJ, eclipse, pycharm, Spyder ou autre. Vous pouvez utiliser un simple éditeur de texte comme atom, SublimeText, geany ou Notepad++. Vous pouvez utiliser l'environnement de programmation 100% en ligne repl.it mais celui-ci ne permet pas l'affichage graphique avec graphviz (pas indispensable, mais pratique pour déboguer).

Dépôt de fichiers : Prenez un moment maintenant pour préparer votre environnement de travail. Nous vous conseillons de créer un dépôt de contrôle de version git pour ce mini-projet, par exemple sur etulab. Alternativement, créez une dossier Dropbox, Google Drive ou AMUBox synchronisé pour sauvegarder. 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.

Installation des pré-requis : Pour l'affichage graphique d'automates, vous devez installer l'outil graphviz et la bibliothèque python correspondante. Sur Linux, une fois python 3 installé, vous pouvez exécuter :

```
sudo apt install graphviz
pip3 install graphviz
```

Si vous êtes sur Windows, le plus simple est d'installer Anaconda. Une fois que vous l'avez téléchargé et installé, allez sur Anaconda Navigator > Environement > Update index. Affichez les packages 'Not installed' au lieu de 'Installed' puis cherchez graphviz dans la barre de recherche. Installez les packages graphviz et python-graphviz (les deux sont nécessaires). Ensuite, vous pouvez revenir à la page d'accueil (Home) et lancer l'IDE Spyder.

1.3. **Format de fichier `.txt`.** Un automate à pile décrit dans un fichier .txt (UTF-8) consiste en une suite de lignes terminées par `\n` (CR), où chaque ligne est un 5-uplet, sauf la dernière. Les éléments du 5-uplet sont séparés par des espaces. Le 5-uplet p a X Y q correspond à une transition entre un état source p et un état cible q étiquetée par le symbole a, avec X comme symbole de tête de pile et Y est la chaine de symboles de pile empilés, séparés par des points (`.`). Par convention, il est recommandé de représenter les états p et q avec des nombres, les symboles a avec des minuscules non-accentuées et les symboles de pile par des lettres majuscules. Le symbole spécial % est utilisé pour les transition-ε et également pour ne rien empiler sur la pile. La transition initiale détermine à la fois l'état initial (l'état source) et le symbole de pile initial (le symbole de tête de pile). La dernière ligne est toujours précédée de la lettre majuscule A indiquant les états d'acceptation. Ensuite, les états d'acceptation sont listés, séparés par des espaces.

Voici un exemple d'automate à pile déterministe décrit dans un fichier. Cet automate reconnait $\{a^n b^n |\ n \geq 1\}$. Pour vous en convaincre, dessinez-le sur une feuille :

```
0 a Z A.Z 0
0 a A A.A 0
0 b A % 1
1 b A % 1
1 % Z % 2
A 2
```

1.4. **Bibliothèque `automaton.py` fournie.** Nous vous fournissons une bibliothèque python automaton.py, disponible sur le github du cours, avec les fonctionnalités suivantes :

- création et manipulation d'un objet StackAutomaton qui représente un automate à pile,
- lecture et écriture à partir d'un fichier textuel,
- affichage graphique à l'aide de graphviz.

Vous devez bien comprendre le fonctionnement de cette bibliothèque. Les prochaines sections vous permettent de l'essayer. Si vous êtes à l'aise en python, vous pouvez ouvrir le code de la bibliothèque pour le lire. Vous pouvez aussi n'utiliser qu'une partie des fonctionnalités fournies.

Pour commencer, vous devez importer la bibliothèque fournie automaton.py, qui doit être placée dans le même dossier où se trouve votre script/programme source. Ensuite, vous pouvez créer un nouvel automate et lui donner un nom. Cet automate, pour le moment, sera vide (vérifiez en exécutant le code ci-dessous) :

```
import automaton
a_test = StackAutomaton("a_test")
```

Vous pouvez ensuite ajouter des états et des transitions. La source de la première transition sera considérée comme l'état initial et le symbole de tête de pile sera le symbole de pile initial.

```
a_test.reset()
a_test.add_transition("0","a","Z",["A","Z"],"1")
```

Notez que cet automate ne reconnait aucun mot, car il n'a aucun état d'acceptation/final. Par défaut on considère que les automates à piles acceptent par état final. Vous pouvez marquer un ou plusieurs états comme finaux à l'aide de la fonction suivante :

```
a_test.make_accept("1")
```

Alternativement, vous pouvez marquer plusieurs états comme états d'acceptation :

```
a_test.make_accept(["0","1"])
```

Vous pouvez afficher l'automate sous la forme textuelle à n'importe quel moment dans votre code, par exemple, pour déboguer. Notez que la fonction delta est donnée sous la forme d'une liste de transitions :

```
print(a_test)
```

```
a_test = <Q={0,1}, S={a}, Z={Z,A}, D, Z0=Z, q0=0, F={0,1}>
D =
(0,a,Z,[A,Z],1)
```

Dans cet exemple, `Q` contient la liste d'états `0,1`, l'alphabet S est automatiquement construit à partir des transitions et contient uniquement le symbole `a`, l'alphabet de pile contient les symboles `Z,A`, Z le symbole de pile initial, l'état initial est `q0=0` et les états d'acceptation `F=0,1`. Ces valeurs sont accessibles aussi directement, sous la forme de listes ou de chaînes de caractères, via des variables de l'automate :

```
print(a_test.states)
print(a_test.alphabet)
print(a_test.stack_alphabet)
print(a_test.initial)
```

```
print(a_test.initial_stack)
print(a_test.acceptstates)
```

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

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

Un automate peut être réinitialisé à tout moment :

```
a_test.reset()
```

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

```
source = """0 a Z A.Z 0
0 a A A.A 0
0 b A % 1
1 b A % 1
1 % Z % 2
A 2
"""
anbn = StackAutomaton("anbn")
anbn.from_txt(source)
```

Les transitions-ε sont représentées par le symbole spécial `%` (constante `automaton.Automaton.EPSILON`). Essayez de modifier la variable source et observez le résultat dans l'automate. Ajouter ou supprimez des transitions, des états, des états d'acceptation.

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

```
[11]: anbn.from_txtfile("test/anbn.ap")
```

Les automates peuvent s'affichent graphiquement à l'aide de la fonction `to_graphviz()`. Pour cela, il suffit d'appeler la fonction avec un nom de fichier, et un fichier (avec l'extension .pdf automatiquement ajoutée) sera créé. La fonction renvoie l'objet graphviz au format DOT.

Vous aurez besoin d'accéder à la liste d'états et de transitions de l'automate. Chaque état est dans un dictionnaire python indexé par son nom, et contient une liste de transitions. Les transitions sont, des listes contenant des tuples (lettre, tête de pile, liste de symboles à empiler, état cible). 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 transitions de l'automate, qui donne la liste de transitions sous la forme d'une `list` python simple. Chaque transition est une tuple de 5 éléments : source, lettre, tête, empile, destination. Tous sont de type `str` sauf empile qui est une `list` de `str`. Cela peut être pratique, si vous voulez manipuler des objets plus simples que les dictionnaires et objets StackState.

```
for (source, letter, head, push, dest) in a.transitions : # et non pas a.transitions()
print( "{} --{},{}/{}--> {}".format(source,letter,head,push,dest))
```

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 un automate fini en entrée et qui renvoie un booléen pour indiquer si, oui ou non, l'automate est déterministe

(2) Écrire une fonction qui prend un automate fini déterministe et un mot en entrée, et qui renvoie un booléen pour indiquer si, oui ou non, le mot est reconnu par l'automate

De plus, vous devez écrire un script/programme qui peut être exécuté sur un terminal et qui prend en entrée `(sys.argv)` deux arguments : un fichier texte contenant un automate et un mot à reconnaître, dans cet ordre. Le script doit :

(1) Charger l'automate depuis le fichier texte (à l'aide de la bibliothèque fournie)

(2) Vérifier si l'automate est déterministe (fonctionnalité 1 ci-dessus) et, sinon, afficher ERROR

(3) Si l'automate est déterministe, vérifier s'il reconnaît le mot (fonctionnalité 2 ci-dessus) et afficher YES si le mot est reconnu, NO sinon

2.1. **Fichiers tests.** Quelques fichiers de test vous sont fournis dans le dossier `test`. Voici quelques exemples d'exécution pour votre programme (supposons qu'il s'appelle `tp1langages.py` - squelette fourni) :

```
$ ./tp1langages.py test/anbn.ap ab
YES
$ ./tp1langages.py test/anbn.ap aaabbb
YES
$ ./tp1langages.py test/anbn.ap %
NO
$ ./tp1langages.py test/anbn.ap aaaaaaa
NO
```

```
$ ./tp1langages.py test/anbn.ap bbaa
NO
$ ./tp1langages.py test/anbn.ap aba
NO
$ ./tp1langages.py test/anbn.ap abc
NO
$ ./tp1langages.py test/aplusbstar.ap ab
YES
$ ./tp1langages.py test/aplusbstar.ap aabaab
YES
$ ./tp1langages.py test/aplusbstar.ap abb
NO
$ ./tp1langages.py test/anan.ap abb
ERROR
$ ./tp1langages.py test/anan.ap abb
ERROR
```

Vous devez non seulement implémenter la fonction mais aussi la tester. Pour cela, **écrivez 3 automates à pile au format .ap** pour étendre la base de tests au délà des fichiers fournis. Vous pouvez utiliser des automates vus en cours ou en TD. Ces automates doivent être placés dans le dossier test et formeront votre base de tests, qui grandira au fur et à mesure des TPs.

2.2. **Tester le déterminisme.** On rappelle qu'un automate à pile est déterministe si pour chaque configuration, au plus une transition est utilisable. Formellement, un automate à pile est déterministe si pour chaque état p, chaque symbole de tête de pile X et chaque lettre a il y a au plus une transition possible, c'est à dire $\delta(p, a, X) \cup \delta(p, \varepsilon, X)$ contient au plus un élément.

2.3. **Exécution d'un automate à piles déterministe 1- transitions epsilon.** Un automate à pile déterminite peut avoir des transitions epsilon. Définir une fonction `execute_epsilon` qui prend en entrée un automate à pile a, un état p et un contenu de pile w et exécute, tant que possible (donc avec une boucle while) des transitions ε.

2.4. **Exécution d'un automate à piles déterministe 2.** Finalement écrivez la fonction `recognizes` qui exécute un automate à pile sur un mot. Pour cela vous utiliserez la fonction `execute_epsilon` définie précédemment.