# TME programmation d'automates finis

L'objectif de ce TME est de programmer en python quelques uns des algorithmes pour les automates finis vus en cours et en TD, en utilisant des structures de données fournies dans le code mis à votre disposition.

## CONSIGNES

Copiez dans votre répertoire de travail les fichiers présents dans le Dossier `Fichiers Python fournis` de la page Moodle de l'UE.

Ils contiennent les définitions de structures de données décrites ci-dessous, ainsi que des aide-mémoire sur l'utilisation de python.

**Le seul fichier que vous êtes autorisés à modifier est** `automate.py`, partiellement prérempli. Les instructions `return` sont à supprimer lorsque vous remplirez le contenu des différentes fonctions. Les autres fichiers n'ont pas besoin d'être lus.

## 1 Présentation

Le projet utilise le langage python avec une syntaxe légèrement différente de celle vue en LU1I001, parce qu'il exploite la notion de classes d'objets. Les particularités sont brièvement expliquées en annexe de ce document. Par ailleurs, vous trouverez dans le dossier `Aide-mémoire Python` un mémo sur la syntaxe python, ainsi que la carte de référence du langage utilisée en LU1I001. On rappelle qu'une ligne commençant par `#` est un commentaire, ignoré par l'interpréteur.

Toutes les structures de données nécessaires à la construction des automates sont fournies sous la forme de classes python, pour les transitions d'un automate, ses états, et les automates eux-mêmes. Cette section indique comment les utiliser.

### 1.1 La classe `State`

Un état est représenté par
- un entier `id` (type `int`) qui définit son identifiant,
- un booléen `init` (type `bool`) indiquant si c'est un état initial,
- un booléen `fin` (type `bool`) indiquant si c'est un état final,
- une chaîne de caractères `label` (type `str`) qui définit son étiquette, permettant de le "décorer". Par défaut, cette variable est la version chaîne de caractères de l'identifiant de l'état.

On définit l'alias de type `State` pour représenter les variables de ce type.

Ainsi, l'instruction ci-dessous crée une variable `s` représentant un état d'identifiant `1`, qui est initial mais pas final, dont l'identifiant et l'étiquette sont `1` :

```
# s : State
s = State(1, True, False)
```

Si l'on souhaite avoir une étiquette différente de l'identifiant, on utilise un quatrième argument : `s = State(1, True, False, ''etat 1'')`.

On accède ensuite aux différents champs de `s` par la notation pointée : la séquence d'instructions suivante provoque l'affichage indiqué en regard

```
print(s.id)              1
print(s.init)            True
print(s.fin)             False
print(s.label)           1
print(s)                 1(init)
```

Ainsi, une variable de type `State` est affichée par son étiquette et, entre parenthèses, si c'est un état initial et/ou final.

### 1.2 La classe `Transition`

Une transition est représentée par
— un état `stateSrc` (type `State`) correspondant à son état de départ
— un caractère `etiquette` (type `str`) donnant son étiquette
— un état `stateDest` (type `State`) correspondant à son état de destination
On définit l'alias de type `Transition` pour représenter les variables de ce type.

La séquence d'instructions suivante crée la transition d'étiquette `a` de l'état `s` (défini ci-dessus) vers lui-même et affiche les différents champs de la transition :

```
# t : Transition
t = Transition(s, "a", s)
print(t.etiquette)              a
print(t.stateSrc)               1(init)
print(t.stateDest)              1(init)
print(t)                        [1(init)-a->1(init)]
```

On remarque que la variable `t.stateSrc` est de type `State`, on obtient donc un état, et non uniquement un identifiant d'état.

### 1.3 La classe `Automate`

Un automate est représenté par
— la liste de ses transitions `listTransitions` (de type `list[Transition]`)
— la liste de ses états `listStates` (de type `list[State]`)
— une étiquette `label` (de type `str`) qui est éventuellement vide.
On définit l'alias de type `Automate` pour représenter les variables de ce type.

Ainsi, de même que pour les classes précédentes, l'accès aux différents champs se fait par la notation pointée. Par exemple, on obtient la liste des états d'un automate `monAutomate` par l'instruction `monAutomate.listStates`.

Pour créer un automate, il existe trois possibilités.

**Création à partir d'une liste de transitions.** On peut d'abord utiliser le constructeur de signature `Automate : list[Transition] -> Automate`. Il déduit alors la liste des états à partir de la liste des transitions et définit par défaut l'étiquette de l'automate comme la chaîne de caractères vide.

Par exemple, en commençant par créer les états et les transitions nécessaires

```
## création d'états
# s1 : State
s1 = State(1, True, False)
# s2 : State
s2 = State(2, False, True)
## création de transitions
# t1 : Transition
```

```
t1 = Transition(s1,"a",s1)
# t2 : Transition
t2 = Transition(s1,"a",s2)
# t3 : Transition
t3 = Transition(s1,"b",s2)
# t4 : Transition
t4 = Transition(s2,"a",s2)
# t5 : Transition
t5 = Transition(s2,"b",s2)
# liste : list[Transition]
liste = [t1,t2,t3,t4,t5]

## création de l'automate
# aut : Automate
aut = Automate(liste)
```

L'affichage de cet automate, par la commande `print(aut)` produit alors le résultat suivant :

```
Etats :1(init)
2(fin)
Transitions :[1(init)-a->1(init)]
[1(init)-a->2(fin)]
[1(init)-b->2(fin)]
[2(fin)-a->2(fin)]
[2(fin)-b->2(fin)]
```

Les états de l'automate sont déduits de la liste de transitions.

Optionnellement, on peut donner un nom à l'automate, en utilisant la variable `label`, par exemple

```
# aut2 : Automate
aut2 = Automate(liste_trans, label="A")
```

**Création à partir d'une liste de transitions et d'une liste d'états.** Dans le second cas, on crée un automate à partir d'une liste de transitions mais aussi d'une liste d'états, par exemple pour représenter des automates contenant des états isolés. Pour cela, on utilise le constructeur `Automate : list[Transition] x liste[State] -> Automate`.

On peut également, optionnellement, donner un nom à l'automate :

```
# aut3 : Automate
aut3 = Automate(liste_trans, liste_etats, "B")
```

L'ordre des paramètres peut ne pas être respecté *à la condition* que l'on donne leur nom explicitement. Ainsi, la ligne suivante est correcte

```
aut = Automate(listStates = liste_etats, label = "A", listTransitions = liste_trans)
```

**Création à partir d'un fichier contenant sa description.**

La fonction `Automate.creationAutomate : str -> Automate` prend en argument un nom de fichier qui décrit un automate et construit l'automate correspondant (voir exemple ci-dessous).

La description textuelle de l'automate doit suivre le format suivant (voir exemple ci-dessous) :

— `#E:` suivi de la liste des noms des états, séparés par des espaces ou des passages à la ligne. Les noms d'états peuvent être n'importe quelle chaîne alphanumérique pouvant également contenir le symbole _. Par contre, si le nom d'état contient des symboles *non numériques* il ne doit pas commencer par un chiffre, sous peine de provoquer une erreur à l'affichage. Ainsi, `10` et `A1` sont des noms d'états possibles, mais `1A` ne l'est pas.

— `#I:` suivi de la liste des états initiaux séparés par des espaces ou des passages à la ligne,
— `#F:` suivi de la liste des états finals séparés par des espaces ou des passages à la ligne,
— `#T:` suivi de la liste des transitions séparées par des espaces ou des passages à la ligne. Chaque transition est donnée sous le format `(etat1, lettre, etat2)`.

Par exemple le fichier `exempleAutomate.txt` contenant

```
#E: 0 1 2 3
#I: 0
#F: 3
#T:  (0 a 0)
(0 b 0)
(0 a 1)
(1 a 2)
(2 a 3)
(3 a 3)
(3 b 3)
```

est formaté correctement. L'appel suivant

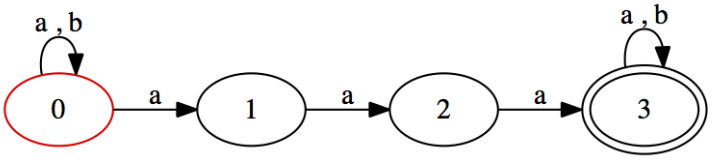
```
# automate : Automate
automate = Automate.creationAutomate("exempleAutomate.txt")
print(automate)
```

produit l'affichage suivant

```
Etats :0(init)
3(fin)
1
2
Transitions :[0(init)-a->0(init)]
[0(init)-b->0(init)]
[0(init)-a->1]
[1-a->2]
[2-a->3(fin)]
[3(fin)-a->3(fin)]
[3(fin)-b->3(fin)]
```

**Fonctions de manipulation des automates.** La classe automate contient également de nombreuses fonctions utiles. Elles s'appliquent à un objet de type `Automate` et s'utilisent donc sous la forme `aut.<`*fonction*`>(<`*parametres*`>)` où `aut` est une variable de type `Automate`.

— `show : str ->`
prend en argument une chaîne de caractères `nomFichier` et produit une représentation graphique de l'automate dans le fichier `nomFichier.pdf`.
Ainsi, en utilisant l'automate défini dans le fichier d'exemple précédent, l'instruction `automate.show("exemple")` produit un fichier exemple.pdf contenant l'image reproduite ci-dessous :




— `addTransition : Transition -> Bool`
prend en argument une transition `t`, fait la mise à jour de l'automate en lui ajoutant `t` et ajoute les états impliqués dans l'automate s'ils en sont absents. Elle rend `True` si l'ajout a eu lieu, `False` sinon (si `t` était déjà présente dans l'automate).

— `removeTransition : Transition -> Bool`
prend en argument une transition `t` et fait la mise à jour de l'automate en lui enlevant la transition, sans modifier les états. Elle rend `True` si la suppression a eu lieu, `False` sinon (si `t` était absente de l'automate).
— `addState : State -> Bool`
prend en argument un état `s` et fait la mise à jour de l'automate en lui ajoutant `s`. Elle rend `True` si l'ajout a eu lieu, `False` sinon (si `s` était déjà présent dans l'automate).
— `removeState : State -> Bool`
prend en argument un état `s` et fait la mise à jour de l'automate en supprimant `s` ainsi que toutes ses transitions entrantes et sortantes. Elle rend `True` si l'ajout a eu lieu, `False` sinon (si `s` était absent de l'automate)
— `getListInitialStates : -> list[State]`
rend la liste des états initiaux
— `getListFinalStates : -> list[State]`
rend la liste des états finals
— `getListTransitionsFrom : State -> liste[Transition]`
rend la liste des transitions sortant de l'état passé en argument
— `prefixStates : int ->`
modifie les identifiants et les étiquettes de tous les états de l'automate en les préfixant par l'entier passé en argument.
— `succElem : State x str -> list[State]`
étant donné un état $s$ et un caractère $a$, elle rend la liste des états successeurs de $s$ par le caractère $a$. Formellement,

$$succElem(s, a) = \{s' \in S \mid s \xrightarrow{a} s'\}.$$

Avec l'exemple précédent, `liste = aut.succElem(s1, 'a')` donne `liste = [1(init), 2(fin)]`.

# 2 Prise en main

## 2.1 Création d'automates

Soit l'automate $\mathcal{A}$ défini sur l'alphabet $\{a, b\}$, d'états $0, 1, 2$, d'état initial 0, d'état final 2 et de transitions $(0, a, 0)$, $(0, b, 1)$, $(1, a, 2)$, $(1, b, 2)$, $(2, a, 0)$ et $(2, b, 1)$.

1. Créer l'automate $\mathcal{A}$ à l'aide de sa liste de transitions. Pour cela, créer un état `s0` d'identifiant 0 qui soit initial, un état `s1` d'identifiant 1 et un état `s2` d'identifiant 2 qui soit final. Puis créer `t1`, `t2`, `t3`, `t4`, `t5` et `t6` les 6 transitions de l'automate. Créer enfin l'automate `auto` à partir de ses transitions, par exemple avec l'appel

   ```
   auto = Automate([t1,t2,t3,t4,t5,t6])
   ```

   Vérifier que l'automate correspond bien à $\mathcal{A}$ en l'affichant à l'aide des appels

   ```
   print(auto)
   auto.show("A_ListeTrans")
   ```

2. Créer l'automate $\mathcal{A}$ à l'aide de sa liste de transitions et d'états, par exemple à l'aide de l'appel `auto1 = Automate([t1,t2,t3,t4,t5,t6], [s0,s1,s2])`

   puis afficher l'automate obtenu à l'aide de `print` puis dans un fichier pdf. Vérifier que l'automate `auto1` est bien identique à l'automate `auto`.

3. Créer l'automate $\mathcal{A}$ à partir d'un fichier. Pour cela, créer un fichier `auto.txt`, dans lequel seront indiqués les listes des états et des transitions, ainsi que l'état initial et l'état final, en respectant la syntaxe donnée dans la section précédente. Par exemple la liste d'états sera décrite par la ligne `#E: 0 1 2`. Utiliser ensuite par exemple l'appel `auto2=Automate.creationAutomate("auto.txt")`, puis afficher l'automate `auto2` à l'aide de `print` ainsi que dans un fichier pdf.

### 2.2 Premières manipulations

1. Appeler la fonction `removeTransition` sur l'automate `auto` en lui donnant en argument la transition $(0, a, 1)$. Il s'agit donc de créer une variable `t` de type `Transition` représentant $(0, a, 1)$ et d'effectuer l'appel `auto.removeTransition(t)`. Observer le résultat sur un affichage. Appeler ensuite cette fonction sur `auto` en lui donnant en argument la transition `t1`. Observer le résultat sur un affichage. Appeler la fonction `addTransition` sur l'automate `auto` en lui donnant en argument la transition `t1`. Vérifier que l'automate obtenu est bien le même qu'initialement.
2. Appeler la fonction `removeState` sur l'automate `auto` en lui donnant en argument l'état `s1`. Observer le résultat. Appeler la fonction `addState` sur l'automate `auto` en lui donnant en argument l'état `s1`. Créer un état `s2` d'identifiant 0 et initial. Appeler la fonction `addState` sur `auto` avec `s2` comme argument. Observer le résultat.
3. Appeler la fonction `getListTransitionsFrom` sur l'automate `auto1` avec `s1` comme argument. Afficher le résultat.

## 3 Exercices de base : tests et complétion

1. Donner une définition de la fonction `succ` qui, étant donné une liste d'états $L$ et une chaîne de caractères $a$, renvoie la liste des états successeurs de tous les états de $L$ par le caractère $a$. Cette fonction doit généraliser la fonction `succElem` pour qu'elle prenne en paramètre une liste d'états au lieu d'un seul état. Formellement, si $L$ est une liste d'états et $a$ une lettre,
$$succ(L, a) = \bigcup_{s \in L} succ(s, a) = \{s' \in S \mid \text{il existe } s \in L \text{ tel que } s \xrightarrow{a} s'\}.$$
2. Donner une définition de la fonction `accepte` qui, étant donné une chaîne de caractères `mot`, renvoie un booléen qui vaut vrai si et seulement si `mot` est accepté par l'automate. Attention, noter que l'automate peut ne pas être déterministe.
3. Donner une définition de la fonction `estComplet` qui, étant donné un automate `auto` et une liste de caractères `alphabet`, renvoie un booléen qui vaut vrai si et seulement si `auto` est complet par rapport à l'alphabet. Attention, cette fonction prend l'automate en argument, c'est une fonction statique (cette précision n'est pas répétée pour les autres fonctions statiques demandées).
4. Donner une définition de la fonction `estDeterministe` qui, étant donné un automate `auto`, renvoie un booléen qui vaut vrai si et seulement si `auto` est déterministe.
5. Donner une définition de la fonction `completeAutomate` qui, étant donné un automate `auto` et l'alphabet d'entrée `alpha`, renvoie l'automate complété d'`auto`. Attention, il ne faut pas modifier `auto`, mais construire un nouvel automate. Il pourra être intéressant d'utiliser l'appel de fonction `copy.deepcopy(auto)` qui renvoie un nouvel automate identique à `auto`.

## 4 Déterminisation

Donner une définition de la fonction `determinisation` qui, étant donné un automate `auto`, renvoie l'automate déterminisé d'`auto`. Comme pour la fonction `completeAutomate`, il ne faut pas modifier `auto`, mais construire un nouvel automate.

Les ensembles python, de type `set`, peuvent être utiles, quelques rappels de leur utilisation sont donnés en annexe.

## 5 Constructions sur les automates réalisant des opérations sur les langages acceptés

### 5.1 Opérations ensemblistes sur les langages

1. Donner une définition de la fonction `complementaire` qui, étant donné un automate `auto` et une liste de caractères `alphabet`, renvoie l'automate acceptant comme langage le complémentaire du

langage accepté par `auto`.

2. Donner une définition de la fonction `intersection` qui, étant donné deux automates `auto1` et `auto2`, renvoie l'automate acceptant comme langage l'intersection des langages acceptés par `auto1` et `auto2`. L'automate construit ne doit pas avoir d'état non accessible depuis l'état initial.

   Remarque : python fournit, dans le package `itertools`, la fonction `product : list[alpha] x list[beta] -> iterable` qui, étant donné deux listes `L1` et `L2`, rend le produit cartésien de `L1` et `L2`. Attention, le résultat n'est pas une liste, mais peut être converti en lui appliquant la fonction `list`, on obtient alors un élément de type `list[tuple[alpha, beta]]`. Tester par exemple la séquence d'instructions

   ```
   import itertools
   L1 = [1, 2, 3]
   L2 = ["a", "b", "c"]
   L = list(itertools.product(L1, L2))
   ```

3. *Question facultative*

   Donner une définition de la fonction `union` qui, étant donné deux automates `auto1` et `auto2`, renvoie l'automate acceptant comme langage l'union des langages acceptés par `auto1` et `auto2`.

### 5.2 Opérations rationnelles sur les langages

Programmer *une des deux* méthodes suivantes :

1. Donner une définition de la fonction `concatenation` qui, étant donné deux automates `auto1` et `auto2`, renvoie l'automate acceptant comme langage la concaténation des langages acceptés par `auto1` et `auto2`.
2. Donner une définition de la fonction `etoile` qui, étant donné un automate `auto`, renvoie l'automate acceptant comme langage l'étoile du langage accepté par `auto`.

## A Quelques rappels de python

Des éléments de python sont fournis dans le fichier `memo-pythonv2.pdf`, ainsi que, de façon plus concise mais plus complète, dans le fichier `carteref1I001.pdf`. Ici sont seulement donnés des éléments concernant les signatures de fonction et la notion d'objet en python.

### A.1 Fonction et signature

La syntaxe de définition d'une fonction est

```
def nom_fonction(nom_arguments):
    """ type_arguments -> type_resultat
        description brève de ce que fait la fonction
    """
```

La première ligne entre les guillemets est appelée *signature* de la fonction, elle est donnée dans le sujet pour les fonctions de manipulation des automates ci-dessus, elle est aussi donnée dans les fichiers python fournis pour le projet.

Les types des arguments, séparés par des `x`, ainsi que le type du résultat, peuvent être

— des types de base : `int` (entier), `float` (réel à l'exclusion des entiers), `Number` (entier ou réel), `Bool` (booléen)
— `str` : chaîne de caractères
— `list[`⟨type⟩`]` : liste dont les éléments sont du type indiqué entre crochet
— `set[`⟨type⟩`]` : ensemble dont les éléments sont du type indiqué entre crochet
— `State`, `Transition` et `Automate` dans le cadre du projet

## A.2 Python objet

Python permet de construire des *objets*, structures regroupant des informations de type différents, comme les états, les transitions ou les automates dans ce projet.

A titre d'exemple, on considère ici les automates, de type `Automate` et une variable `a1` de ce type.

**Accès aux informations de l'objet.** On accède aux différentes informations associées à un objet par la notation pointée illustrée précédemment dans le sujet : `a1.listStates` est une variable de type `list[State]` qui contient la liste de tous les états de l'automate `a1`, de même `a1.label` est une variable de type `str` qui contient l'étiquette de l'automate `a1`.

**Utilisation de fonctions de manipulation.** On peut appliquer des fonctions particulières, appelées *méthodes*, à des objets en utilisant la notation pointée. C'est par exemple le cas de la fonction de concaténation de listes `L1.append(L2)` qui ajoute, à la suite d'une liste `L1`, la liste `L2`, en modifiant la variable `L1`.

Dans le cas des automates, de même, `a1.show(''exemple'')` applique la fonction `show` à l'objet `a1`.

**Définition de méthodes.** Pour qu'une fonction s'applique à un objet, comme cela est demandé dans le sujet pour les fonctions `succ` ou `accepte` par exemple, deux conditions doivent être remplies :

- la fonction est définie dans le bloc `class` qui définit l'objet considéré. Ici, les fonctions doivent être dans le fichier `automate.py` et avec l'indentation qui les place dans le bloc `class Automate`
- la fonction a pour premier argument `self` : cette variable représente l'objet auquel la fonction est appliquée et elle peut être utilisée comme toute autre variable. La seule différence, mineure, est qu'elle ne figure pas dans la signature de la fonction.

Ces deux principes sont par exemple illustrés par la fonction `succElem` fournie dans le fichier `automate.py`.

**Définition de fonctions statiques dans une classe.** Les fonctions qui ne sont pas des méthodes sont appelées *fonctions statiques*. Elles sont signalées par le mot-clé `@staticmethod` qui précède le `def`.