Programmation Python
TP n° 6 : Expressions régulières et automates

L'objectif de ce TP est d'implementer en Python deux algorithmes de conversion automate → expression régulière, vu dans le cours Langages formels. Pour tous les exercices, utilisez un style orienté objet et n'oubliez pas de tester toutes les fonctions.
Référence :
[1] O. Carton, Langages formels, Calculabilité et Complexité, Ch. 1: Langages rationnels, https://www.irif.fr/~carton/Lfcc/chap1.pdf

I) Expressions régulières

On implémentera des classes et des méthodes qui permettent de représenter des expressions régulières et de manipuler ces expressions à l'aide d'opérations comme l'union, la concaténation, et l'étoile de Kleene.

Question 1. Définir une énumération pour représenter les différents types d'opérations dans les expressions régulières : `EPSILON`, `EMPTY`, `LETTER`, `PLUS`, `TIMES`, et `STAR`.

Question 2. Implémenter une classe `Expression` qui stocke le type d'opération (via l'énumération) et une liste d'arguments. La classe aura un constructeur qui prend comme paramètres un type d'opération (issu de l'énumération) et un nombre variable d'arguments.

Question 3. Écrire une méthode `__str__()` pour représenter une expression sous forme de chaîne de caractères. Pour cette méthode, utiliser un **`match`** pour distinguer les différents types d'opérations (`EPSILON`, `EMPTY`, etc.) et formater la sortie en conséquence.

Question 4. Écrire des fonctions "constructrices" pour créer des instances spécifiques d'expressions régulières, par exemple,

- `Letter(l: str) -> Expression` crée une expression contenant une lettre,
- `Plus(*args: Expression) -> Expression` crée une expression représentant une union,
- de la même manière pour `Times`, `Star`, `Empty`, et `Epsilon`.

Question 5. Écrire une fonction `expand(e: Expression) -> Expression` qui simplifie et développe une expression régulière en suivant (au minimum) les règles suivantes :

1. Distributivité de la concaténation (`Times`) sur l'union (`Plus`).
2. Flattening : Fusionnez des unions et concaténations imbriquées, e.g., `Plus(a,Plus(b,c))` est réécrit en `Plus(a,b,c)`.
3. Idempotence de l'étoile de Kleene : `Star(Star(a))` est réécrit en `Star(a)`.
4. Des règles de calcul pour l'ensemble vide et le mot vide : `Star(Empty)` est réécrit en `Epsilon`, `Plus(Empty, a)` → `a`, `Times(Empty, a)` → `Empty`, et `Times(Epsilon, a)` → `a`.

II) Automates et conversion

On implementera des méthodes pour convertir un automate en une expression régulière. On utilisera deux méthodes : l'algorithme de McNaughton-Yamada et la méthode d'élimination d'états après normalisation.

Question 6. Implémenter une classe `Automaton` qui prend le nombre d'états, l'alphabet, les transi-

tions, les états initiaux et les états finaux. Le constructeur de cette classe doit vérifier que les états sont bien numérotés de `1` à `n` et lever une erreur sinon.

Question 7. Écrire une méthode `K(p,q,r)` qui calcule l'expression régulière représentant les transitions possibles de l'état `p` à l'état `q` en traversant uniquement des états de numéro inférieur ou égal à `r`. [1, p. 38].

Question 8. Écrire une méthode `expressionK()` qui calcule l'expression régulière pour l'automate en utilisant la méthode de McNaughton-Yamada.

Question 9. Écrire une méthode `normalize()` qui modifie l'automate pour qu'il ait exactement un état initial et un état final. [1, Prop. 1.62].

Question 10. Implémenter une fonction qui utilise l'élimination d'états pour convertir un automate normalisé en une expression régulière [1, p. 39].