COMPILATION
Projet d'analyse lexicale

1 Introduction

1.1 Description du projet

Ce projet est organisé en trois séances de TEA.
Au cours de ces séances, vous allez mettre en oeuvre, en `Python`, une application relative aux AEF qui sera à même d'effectuer l'essentiel du travail d'un analyseur lexical.
Vous travaillerez en trinôme, répartissez vous donc les tâches de manière efficace.
Votre projet sera évalué via une démonstration faites aux enseignants lors de la dernière séance.
Le projet est organisé en deux parties, dépendantes l'une de l'autre :

- Le moteur d'automate
- La déterminisation

1.2 Le dépôt

Vous déposerez une archive compressée sur `Moodle` qui contiendra l'ensemble de votre projet.

2 Moteur d'AEF

2.1 Objectifs

Le but de cette première partie est de construire un moteur d'exécution d'automates à états finis, déterministes et sans λ-transitions.
Le moteur commence par consulter la description (fichier `.descr`, expliqué à la fin du document de cours ou sur Moodle) de l'automate $\mathcal{A}$ qu'il incarne et afficher cette description, à fins de vérification.
Vous implémenterez également une méthode permettant d'exporter vos automates au format `.dot` utilisé par `Graphviz`. En utilisant `Graphviz`, vous pourrez alors générer des images `.png` de vos automates. La grammaire des fichiers `.dot` est expliquée dans le mode d'emploi déposé sur `Moodle` (dotguide).
Le moteur d'automage est capable de traiter les entrées qu'on lui fournit. Les différents mots fournis seront séparés par des espaces ou des retours à la ligne. Pour signifier la fin de la saisie des entrées vous utiliserez ###. Il est également rappelé que la phrase vide est pertinente.

Pour chacune des entrées qu'il analyse, le moteur produit :

— la liste des configurations successives de l'automate,
— la sortie éventuellement produite,
— le diagnostic d'acceptation de l'entrée ou l'explication d'un éventuel blocage.

Lors de la démonstration à un enseignant, votre moteur doit indiquer clairement l'état dans lequel l'automate s'est arrêté : l'automate est bloqué au milieu d'un mot et le moteur justifie pourquoi, l'automate a tout lu et le moteur explique si le mot est accepté ou pas, etc.

Rappels :

— Un automate déterministe possède un seul état initial, mais peut avoir plusieurs états acceptants.
— Un automate qui a plusieurs états initiaux est nécessairement non déterministe.
— Un automate peut générer des sorties. Nous ne considèrerons pas d'automates non-déterministes générant des sorties, par contre vous devez dans cette première partie prendre en compte les éventuelles sorties.
— Un automate non-complet peut bloquer au milieu d'un mot, faute de trouver une transition.

2.2 Description de l'automate

Elle est contenue dans un fichier `.descr` et respecte la syntaxe définie en annexe.
Pour tester votre moteur d'automate, vous devez créer vous-même des automates de test.
Les enseignants valideront votre travail à l'aide de leurs propres jeux d'essais, conformes à cette syntaxe.
Dans un premier temps, vous ne travaillez qu'avec des automates déterministes mais votre moteur devra ensuite être capable de travailler avec tous les automates définis par un fichier `.descr`. En particulier, un automate ayant plusieurs état initiaux, ayant des λ-transitions, etc.

2.3 Remarque

Le sujet vous laisse une grande latitude dans vos choix d'implémentation. Prenez le temps de bien analyser le problème avant de vous lancer dans le codage. Les définitions données dans le document de cours peuvent être un guide utile ...
Prenez également soin de maintenir un code propre et commenté ; validez vous-même votre code à l'aide de tests unitaires.

3 Déterminisation d'un AEFND

3.1 Objectif

Le but de cette partie est d'étendre le travail précédent avec la déterminisation d'un automate $\mathcal{N}$ non-déterministe avec $\lambda-$transitions.
Votre application construira l'automate déterministe $\mathcal{D}$ équivalent à $\mathcal{N}$ et en produira une représentation compatible avec le moteur que vous avez mis au point dans la première partie.
Vous devez donc pouvoir faire fonctionner $\mathcal{N}$ en deux étapes :

1. Déterminisation de $\mathcal{N}$ avec cette seconde partie (donne $\mathcal{D}$)
2. Émulation de $\mathcal{D}$ par le moteur de la première partie.

3.2 Spécifications de l'application

3.2.1 Description de l'AEF non déterministe

C'est la même que celle de la partie précédente (fichier `.descr`). Vous aurez probablement à modifier les méthodes de lecture d'un automate à partir d'un fichier `.descr` pour tenir compte des états initiaux multiples, des $\lambda-$transitions et des configurations à états successeurs multiples.

3.2.2 Description de l'AEF déterministe produit

Elle sera contenue dans un fichier texte et respectera les contraintes de la première partie. Vous implémenterez donc une méthode permettant d'exporter au format `.descr`.
Elle devra pouvoir être fournie au moteur de la première partie, sans modification.

3.2.3 Algorithme mis en œuvre

C'est celui vu en cours. Vous n'avez pas, ici, à minimiser l'automate obtenu.
Pour illustrer votre compréhension de cet algorithme et favoriser la mise au point de votre application, il serait souhaitable que les méthodes calculant λ-fermeture(T) et transiter(T, a) soient explicitées dans votre code.
Ce n'est donc peut-être pas une perte de temps, que de formuler la structure de votre code, en utilisant au plus près le formalisme du document de cours ...

3.2.4 Bilan

Une fois cette deuxième étape terminée, votre projet doit être capable de :

- Lire un fichier `.descr` d'automate quelconque (déterministe ou non, avec lambda-transitions, plusieurs états initiaux, etc.).
- Produire un fichier `.descr` ou `.dot`.
- Une fois l'automate chargé en mémoire, lire des phrases et les analyser.
- Déterminiser un automate.