TP 5 : Implémentation des automates

Automates finis (AF3)

1 Découpage en objets

On se propose, pour implémenter les automates, d'utiliser plusieurs classes :

Etat qui décrit un état complètement, c'est-à-dire s'il est initial ou terminal ainsi que la liste des transitions dont il est la source ;

Transition qui décrit une transition (déterministe) c'est-à-dire ses états de départ et d'arrivée, et son étiquette (lettre) ;

Automate qui décrit l'automate proprement dit, c'est-à-dire le tableau de tous les états.

On va commencer par écrire ces classes avant de leur donner vraiment du corps. Ces classes ne contiendront donc pour le moment que leurs champs et leurs constructeurs.

Exercice 1 Écrivez la classe « `Transition` », avec un (ou plusieurs) constructeur(s) qui permet d'initialiser ses champs.

Pour décrire un état, on veut donner la « liste » des transitions qui partent de cet état. Pour faire une telle liste, on pourra utiliser une implémentation quelconque des listes. La version faite lors du précédent TP pourra faire l'affaire[1].

Exercice 2 Écrivez la classe « `ListeTransition` » qui implémente une liste de transitions.

Exercice 3 Écrivez la classe « `Etat` » avec ses constructeurs (dont un constructeur par défaut), ainsi que les fonctions suivantes :
- « `boolean existeTrans(char c)` » qui dit si une transition portant la lettre `c` part de l'état donné
- « `Etat cible(char c)` » qui retourne l'état où nous envoie la transition portant la lettre `c`

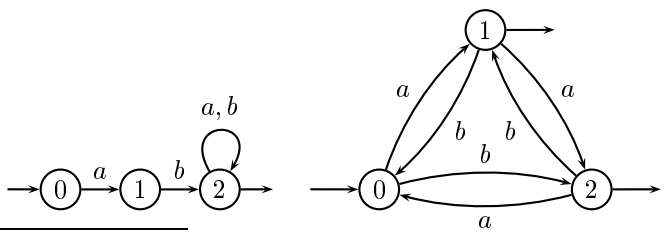
Exercice 4 Écrivez la classe « `Automate` ».

2 Fonctionnement de l'automate

Exercice 5 — État initial Ajoutez une méthode à votre classe « `Automate` » qui retourne l'état initial.

Exercice 6 — Reconnaissance Ajoutez une méthode à votre classe « `Automate` » qui renvoie si un mot, donné sous la forme d'une chaîne de caractères, est reconnu par l'automate.

Exercice 7 — Tests Testez vos méthodes sur les automates suivants :



[1] Vous pouvez aussi choisir d'utiliser une classe de la bibliothèque standard de Java, telle que « `java.util.Vector` ». Cette classe fournit toutes les fonctionnalités dont on aura besoin, telle que « `add` » pour ajouter un nouvel élément ou « `elementAt` » pour obtenir un élément d'indice donné. « `Vector` » fonctionne en effet comme un tableau, c'est-à-dire qu'il permet d'accéder directement à un élément donné, contrairement à une liste qui suppose un parcours des prédecesseurs pour trouver un élément.

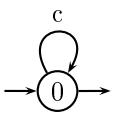
3 Lecture d'un automate depuis un fichier

Pour le moment, la classe « `Automate` » n'a pas de constructeur pertinent. On se propose de lui fournir un constructeur de signature « `Automate(String fichier)` » qui, à partir d'un nom de fichier, crée effectivement un automate.

On choisit le format de fichier suivant pour décrire un automate :

- la première ligne du fichier contiendra toujours le nombre d'états de l'automate (dans la suite du fichier on fera référence aux états par leur numéro compris entre 0 et le nombre d'états moins 1) ;
- les lignes de tout le reste du fichier auront une des formes suivantes :
  - « `initial 10` » pour dire que l'état 10 est initial ;
  - « `terminal 10` » pour dire que l'état 10 est terminal ;
  - « `12 a 10` » pour dire qu'il y a une transition de l'état 12 vers l'état 10 à la lecture de la lettre `a`.

Ainsi, l'automate suivant :



peut être représenté par le fichier suivant :

```
1
initial 0
terminal 0
0 c 0
```

Pour écrire ce constructeur vous aurez besoin d'un certain nombre de fonctions fournies par Java. Référez-vous à la documentation de Java pour trouver leur fonctionnement particulier. Cette documentation est disponible à l'adresse `http://java.sun.com/j2se/1.4.2/docs/api/index.html`.

Vous pourrez en particulier utiliser la classe « `BufferedReader` » qui fournit une méthode « `readLine()` », la méthode « `split` » fournie par la classe « `String` » et la méthode statique « `Integer.parseInt` ». On rappelle aussi que l'égalité des chaînes de caractères « `s1` » et « `s2` » se teste par « `s1.equals(s2)` ».

Exercice 8 Écrivez un constructeur « `Automate(String fichier)` » pour la classe « `Automate` » qui prenne un nom de fichier et initialise l'automate suivant les données du fichier. On procèdera par exemple de la façon suivante :
- ouverture du fichier passé en argument ;
- lecture du nombre d'états de l'automate ;
- initialisation des états de l'automate avec des valeurs par défaut ;
- traitement du reste du fichier.

Testez votre constructeur sur des fichiers écrits par vos soins.

Exercice 9 Écrivez une méthode « `void ecrit(String fichier)` » qui écrit dans le fichier « `fichier` » l'automate, sous la forme décrite précédemment pour la lecture. On pourra utiliser pour cela la classe « `java.io.PrintStream` », à laquelle appartient « `System.out` ». Vérifiez que l'automate que vous produisez par cette méthode est équivalent à celui que vous fournissez en entrée.

4 Pour aller plus loin : quelques fonctions plus complexes

Dans ce qui suit, les fonctions à définir supposent que l'on génère un nouvel automate à partir de l'automate courant (celui sur lequel la méthode est appelée). Cela implique donc que l'on recrée entièrement la structure de l'automate (tous les états et toutes les transitions sont réalloués, afin de ne pas modifier l'automate originel).

Si ces opérations vous paraissent plus compliquées pour cette raison, vous pouvez, au moins dans un premier temps, écrire des méthodes modifiant leur automate (donc « `Automate complete()` » devient ainsi « `void complete()` »).

La seconde étape du travail est du coup d'écrire une méthode qui crée une copie de l'automate. Elle ressemblera, comme on peut s'en douter, à votre constructeur lisant dans un fichier, à ceci près qu'il suffit de lire les informations directement dans l'automate existant.

Exercice 10 — Copie Écrivez une méthode « `makecopy()` » pour votre classe « `Automate` ».

Exercice 11 — Complété Définissez une classe « `Alphabet` » qui contient un tableau des lettres correspondant à l'alphabet considéré.

Ajoutez une méthode « `boolean estComplet()` » à votre classe « `Automate` », puis une méthode « `Automate complete()` » qui calcule le complété.

Exercice 12 — Complémentaire Ajoutez à votre classe « `Automate` » une méthode « `static Automate complementaireLangage(Automate A)` » qui construit un automate reconnaissant le complémentaire du langage reconnu par l'automate « `A` ».

Exercice 13 Écrivez une méthode « `Automate standardise()` ».

Exercice 14 Écrivez des méthodes « `static Automate unionLangages(Automate A, Automate B)` » et « `static Automate intersectionLangages(Automate A, Automate B)` ».

Exercice 15 — automate non déterministe La structure de nos automates nous permet de coder directement des automates non déterministes. Écrire une fonction « `boolean reconnaissanceND(char[] m)` » qui teste la reconnaissance d'un mot par un automate non déterministe. Réfléchir à la manière dont va être codé l'algorithme de déterminisation.