1 Une implémentation des automates en Java

Ce TD machine est consacré à l'implémentation des automates en Java et à l'utilisation de cette implémentation. Une (très grande) partie du travail d'implémentation est déjà fournie. Dans un premier temps, vous vous familiariserez avec l'architecture du projet puis complèterez l'implémentation de quelques éléments manquants.

1.1 Cadre général

Les automates implémentés sont des automates a priori **non déterministes**. Un automate sera bien évidemment représenté par un objet Java. Les principales interfaces qui interviendront sont décrites ci-dessous. On a placé dans des interfaces distinctes les méthodes permettant de simplement connaître et utiliser l'automate de celles utiles pour définir ou modifier l' automate.

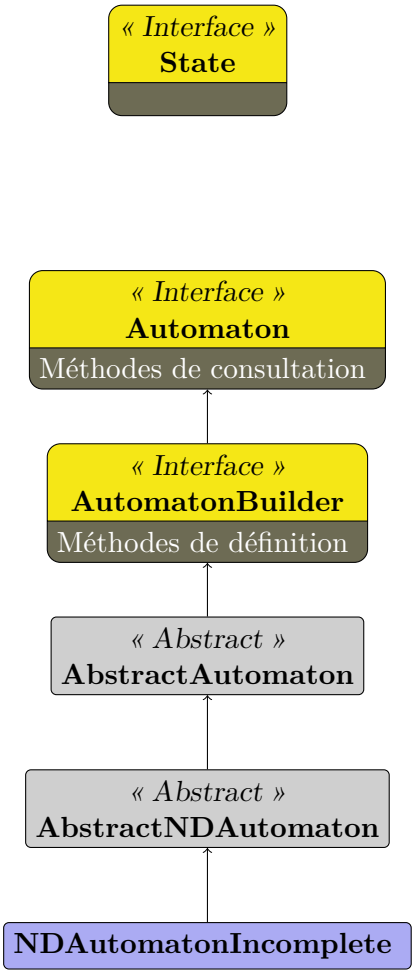
Copiez et ouvrez l'archive avec les sources java. Dans un terminal, placez-vous dans le répertoire `ael` puis générez la javadoc :

```
javadoc -sourcepath src -d doc -docencoding utf8 -encoding utf8 -charset utf8 automata
```

1.1.1 interfaces

(liste incomplète)

- interface `State` : modélise un **état** de l'automate. Chaque état possède un nom et un identifiant. Le nom est choisi à la création de l'état, alors que l'identifiant est attribué « automatiquement » .
- interface `Automaton` : regroupe les méthodes permettant de connaître les composants d'un automate. Par exemple `Set<State> getStates()` permet de récupérer l'ensemble des états de l'automate. La méthode `boolean isAccepting(State)` permet de savoir si un état est acceptant.
- interface `AutomatonBuilder` : regroupe les méthodes permettant de définir ou de modifier les composants d'un automate . Par exemple, `addNewState(String name)` permet d'ajouter un nouvel état.



1.1.2 classes abstraites

(liste incomplète)

- abstract class `AbstractAutomaton` : implémentation des méthodes communes à tous les automates.
- abstract class `AbstractNDAutomaton` : implémentation de la quasi totatlité des méthodes des automates non déterministes (à l'exception de la méthode `accept()`)

1.1.3 classe instanciable

- class `NDAutomatonIncomplete` : extension de la classe `AbstractNDAutomaton` dans laquelle la méthode `accept()` renvoie toujours `false`. De même la méthode `Set<State> getTransitionSet(Set<State> set, char letter)` renvoie `null`.

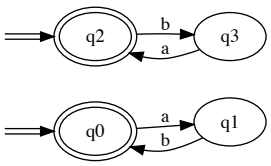
  Cette classe vous est donnée **pour un usage provisoire**, afin de tester immédiatement la création d'automates. Elle sera par la suite **à remplacer par une autre**, possédant une « véritable » implémentation de ces 2 méthodes.

2 Travail à fournir

> *Vous utiliserez les méthodes publiques des classes fournies (vous n'avez pas besoin d'examiner les attributs ou méthodes privées), c'est à dire les méthodes définies dans les interfaces `Automaton` et `AutomatonBuilder`.*

2.1 Premier pas

Il s'agit ici d'une simple prise en main. Éditez la classe `TestND`. Ce petit programme construit l'automate suivant puis affiche sa description sous forme de texte.



Consultez le source de la méthode `main` qui illustre l'utilisation des interfaces Automaton et AutomatonBuilder.
Le programme génère aussi un fichier DOT (représentation graphique de l'automate) nommé `automate-test.dot`. Vous pouvez le convertir en PDF par la commande

```
dot -T pdf -O automate-test.dot
```

2.2 La méthode accept() et la classe NDAutomaton

Vous allez maintenant créer une classe`NDAutomaton` par copie de la classe `NDAutomatonIncomplete`. Dans cette nouvelle classe, vous allez implémenter « réellement » les deux méthodes.
NB : la classe`NDAutomaton` remplacera, pour toute la suite, la classe `NDAutomatonIncomplete` qu'il ne faut plus utiliser

> *Il vous est fortement conseillé de consulter la javadoc de l'interface* `java.util.Set`*. Vous pouvez fabriquer des instances d'ensembles en utilisant la classe* `java.util.HashSet` *ou la classe fournie* `automata.Printset` *(qui étend HashSet en redéfinissant seulement la méthode* `toString()`*). Par exemple, en supposant définis* `Set<T> e,f; T x;` *où T est un type quelconque :*

$e = \emptyset$?	`e.isEmpty()`
$x \in e$?	`e.contains(x)`
réaliser une copie de e	`new XXXSet(e)`
transformer e *en* $e \cup f$	`e.addAll(f);`
transformer e *en* $e \cap f$	`e.retainAll(f);`
transformer e *en* $e - f$	`e.removeAll(f);`
ajouter x *à* e	`e.add(x);`
ôter x *de* e	`e.remove(x);`

1. écrivez d'abord la méthode (spécifiée dans l'interface `Automaton`) :

```
/**
 * Calcul de l'ensemble des états pouvant être obtenus depuis un ensemble d'états
 * @param fromSet : ensemble d'états
 * @param letter : lettre de l'alphabet
 * @return ensemble d'états pouvant être obtenus en lisant letter,
 * en partant de n'importe lequel des états de l'ensemble fromSet
 */
  public Set<State> getTransitionSet(Set<State> set, char letter);
```

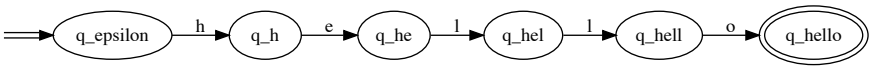
   En reprenant l'exemple précédent,
   — si `fromSet` vaut $\{q_1, q_3\}$ et `letter` vaut a, le résultat vaut $\{q_2\}$
   — si `fromSet` vaut $\{q_1, q_2\}$ et `letter` vaut b, le résultat vaut $\{q_0, q_3\}$
   — si `fromSet` vaut $\{q_1, q_2\}$ et `letter` vaut a, le résultat vaut $\emptyset$
2. écrivez la méthode `accept()`, en utilisant la précédente.
3. modifiez la classe `TestND` en conséquence et décommentez la partie du `main` qui vous permettra de tester la méthode `accept()`.

2.3 Constructions et transformations d'automates

Renommez la classe `AutomataUtilsSkeleton` en `AutomataUtils`.
La classe `AutomataUtils` est destinée à recevoir diverses méthodes statiques de création ou de transformation d'automates. Ces méthodes ne sont pas écrites, ou pas totalement. Il vous revient de les écrire.

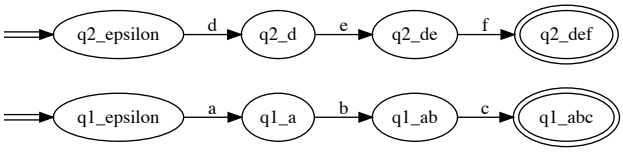
2.3.1 Automate pour singleton ; automate pour langage fini

L'automate reconnaissant un singleton (langage d'un seul mot) est très simple à concevoir : il est linéaire ; pour un mot de n lettres il comporte $n+1$ états , dont un seul état acceptant.



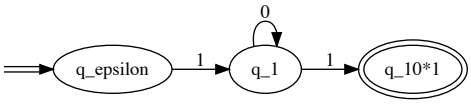
La méthode que vous allez écrire travaille à partir d'un automate non déterministe et lui ajoute les états et transitions nécessaire de façon à assurer que l'automate résultant accepte le mot fourni (en plus de ce qu'il reconnaissait avant).

Une fois cette méthode conçue, il est très simple de construire un automate non déterministe pour un ensemble fini de mots (NB : nous ne cherchons pas ici à « optimiser » l'automate). Exemple pour le langage $\{abc, def\}$:



2.3.2 Automate pour expression régulière « plate »

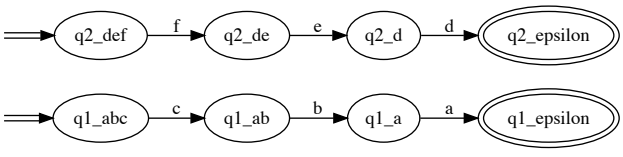
On appellera expression régulière plate une forme particulière d'expression régulière sans parenthèses et où n'intervient pas l'union. En fait seuls apparaissent les lettres et l'opérateur `*` (qui suit nécessairement une lettre). Par exemple `a*bc*` Voici l'automate pour `10*1`.



Comme cela a été fait pour les singletons, la méthode `addFlatExp()` permet d'ajouter à un automate une branche assurant la reconnaissance du langage dénoté par l'expression.

2.3.3 Automate transposé

L'automate transposé, noté $tr(A)$, d'un automate non déterministe est obtenu en échangeant les états initiaux et acceptants et en inversant le sens des flèches. Plus précisément, si $A = (\Sigma, \mathcal{Q}, Ini, \mathcal{F}, \delta)$ alors $tr(A) = (\Sigma, \mathcal{Q}_t, Ini_t, \mathcal{F}_t, \delta_t)$ où $Q_t = Q,\ Ini_t = \mathcal{F},\ \mathcal{F}_t = Ini$ et $\delta_t(q,x) = \{p \mid q \in \delta(p,x)\}, \forall q \in Q, \forall x \in \Sigma,$ On remarque que $q \in \delta(p,x) \iff p \in \delta_t(q,x)$



2.3.4 Automate déterminisé

L'automate sera déterminisé selon l'algorithme vu en cours, algorithme qui produit un automate déterministe, complet et accessible. La méthode est déjà partiellement implémentée, il vous reste à la compléter.