Feuille de TP 2 : Implémentation des automates

Fichiers à télécharger: `Automate.java, AutomateD.java, Etat.java, EtatD.java`.

Le but de ce TP est de simuler le fonctionnement d'un automate fini. On représentera un automate par un ensemble d'états:

```
public abstract class Automate {
    Set<Etat> etats
}
```

Un état est identifié par un nombre entier. On précisera également si l'état est initial ou terminal.

```
public abstract class Etat {
    int id
    boolean estInit
    boolean estTerm
}
```

Pour que l'ensemble ne puisse contenir deux états ayant le même identifiant, on a redéfini (dans le code qui vous est fourni) les méthodes `equals()` et `hashCode()`. Deux états ayant le même identifiant auront le même hashcode et seront considérés comme égaux.

Exercice 1 :

Compléter la méthode `void ajouteEtatSeul(Etat e)` de la classe `Automate`.

1 Automates déterministes

Dans un premier temps, on suppose que chaque état a au plus une transition sortant par lettre et que chaque automate a au plus un état initial. Ainsi, on travaillera avec les nouvelles classes `EtatD` et `AutomateD`:

```
public class EtatD extends Etat{
    HashMap<Character, EtatD> transitions
}

public class AutomateD extends Automate{
}
```

Dans le cas déterministe, chaque transition s'identifie avec une paire d'étiquette et état de cible. Ainsi, on utilisera un champ de type `HashMap<Character, EtatD>` pour stocker les transitions sortant de chaque état.

Exercice 2 :

Dans la classe `EtatD`:

1. Compléter la méthode `void ajouteTransition(char c, EtatD e)`.
2. Compléter les méthodes `EtatD succ(char c)` et `Set<Etat> succ()` qui renvoient respectivement la cible d'une transition et l'ensemble des états successeurs de l'état courant.
3. Compléter la méthode `boolean accepte(String mot)` qui renvoie `true` si et seulement si à partir de l'état courant, en suivant le chemin étiquetté par `mot`, on arrive à un état terminal.

Exercice 3 :

Dans la classe `AutomateD`:

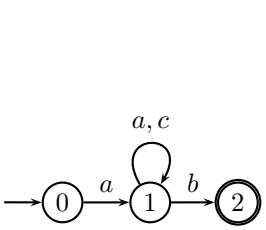
1. compléter la méthode `void ajouteEtatSeul(Etat e)` pour qu'elle rejète tous les états initiaux s'il y en a déjà un ;
2. ajouter une méthode `boolean accepte(String mot)` pour tester la reconnaissance par l'automate.

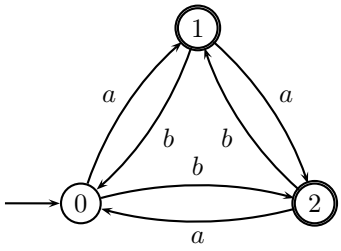
Exercice 4 :

Compléter la méthode `void ajouteEtatRecursif(Etat e)` des classes `Automate` et `AutomateD`.

Exercice 5 :

Écrire une méthode `main` de la classe `AutomateD` pour tester avec les automates suivants:





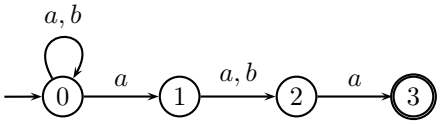
2 Automates non déterministes

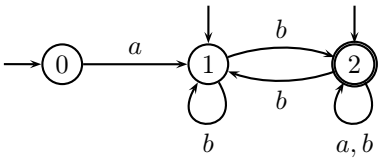
Exercice 6 :

Écrire les classes `public class EtatND extends Etat` et `public class AutomateND extends Automate` pour implémenter des automates non déterministes. Pour implémenter les transitions, on pourra utiliser `HashMap< Character, Set<EtatND>>`.

Exercice 7 :

Écrire une méthode `main` de la classe `AutomateND` pour tester avec les automates suivants et vérifier que les langages reconnus sont bien les bons:





3 Pour aller plus loin

Exercice 8 :

On veut simplifier l'automate en enlevant les états qui ne sont pas accessibles à partir des états initiaux. On ajoutera une méthode `void emonde()` qui supprime de tels états de l'automate. Pour ce faire, on pourra écrire une méthode `Set<Etat> accessibles()` qui renvoie l'ensemble des états accessibles.