Feuille de TP 3: Déterminisation

Fichier à télécharger: `Tables.java`.

1 Alphabet

Il faut d'abord récupérer l'alphabet du langage. On parcourira l'ensemble des états de l'automate et ajoutera à l'alphabet toutes les lettres présentes dans les tables de transitions des états.

Comme les éléments de `etats` sont de type `Etat` et non pas `EtatD` ou `EtatND`, on ne pourra pas utiliser directement la méthode `keySet()` de `HashMap` pour récupérer les étiquettes des transitions sortant de chaque état. On contournera ce problème en ajoutant dans la classe `Etat` une méthode `abstract Set<Character> lettres()`; ce seront les implémentations de cette méthode dans les classes `EtatD` et `EtatND` qui appellent `keySet()`.

Exercice 1 :

Ajouter une méthode `abstract Set<Character> lettres()` à la classe `Etat` et compléter ses implémentations dans les classes `EtatD` et `EtatND`. Ces méthodes renvoient l'ensemble des étiquettes des transitions sortant de l'état.

Exercice 2 :

Ajouter une méthode `abstract Set<Character> alphabet()` à la classe `Automate` et compléter ses implémentations dans les classes `AutomateD` et `AutomateND`. Ces méthodes renvoient l'ensemble de toutes les étiquettes des transitions de l'automate.

2 Gestion de la correspondance état - ensemble d'états

Lors de la déterminisation, on créera "à la volée" des états déterministes correspondants à des ensembles d'états de l'automate de départ. On aura donc besoin d'un `Map< Set<EtatND>, EtatD >`, qui à chaque ensemble d'états rencontré associe un état déterministe. Lors de l'ajout des transitions dans l'automate déterminisé, on aura aussi besoin de retrouver l'ensemble d'états correspondant à un état déterministe, donc d'un `Map< EtatD, Set<EtatND> >`.

On créera ainsi une classe `Tables` qui contient ces deux `Map`:

```
public class Tables {
    Map<Set<EtatND>, EtatD> table;
    Map<EtatD, Set<EtatND>> inverse;
    private int nextId;
    ...
}
```

À chaque fois on rencontre un nouvel ensemble d'états, on crée un nouvel état puis on ajoute cette paire à la fois à `table` et à `inverse`. La variable `nextId` contient l'identifiant du prochain état créé: elle sera initialisée à 0 et incrémentée à chaque ajout d'état.

Les méthodes des exercices 3 et 4 se trouveront dans la classe `Tables`.

Exercice 3 :

Écrire les méthodes suivantes:

1. Un constructeur `Tables()`,

2. Une méthode `EtatD getD(Set<EtatND> s)` (resp. `Set<EtatND> getND(EtatD e)`) qui renvoie l'état déterministe (resp. l'ensemble d'états) correspondant à l'ensemble d'états (resp. l'état) en argument si il existe et `null` sinon.

Exercice 4 : Ajout des éléments dans les tables

Écrire une méthode `boolean add(Set<EtatND> s)` qui cherche l'ensemble `s` dans ses tables. Elle renvoie `false` si `s` s'y trouve déjà. Sinon elle

- créé un nouvel état `ed` avec l'identifiant `nextId`,
- ajoute la paire (`s`, `ed`) aux deux tables,
- incrémente `nextId`,
- renvoie `true`.

Ne pas oublier de marquer le nouvel état comme terminal si l'ensemble qui lui correspond contient un état terminal.

3 Déterminisation

Exercice 5 :

Écrire dans la classe `AutomateND` une méthode `AutomateD determinise()` qui renvoie le déterminisé de l'automate.

Pour initialiser, on ajoute l'ensemble des états initiaux aux tables de correspondances. On crée aussi une liste d'attente des états non traités. On marque l'état qu'on vient de créer comme initial et l'ajoute à la liste d'attente.

En suite, tant qu'il y a des états en attente, on en sort un de la liste et lui ajoute des transitions vers d'autres états déterministes. Ci la cible d'une transition n'existe pas encore, on la crée et la met dans la liste d'attente.

À la fin, on renvoie l'automate déterministe qui contient les états déterministes des tables.

On pourra gérer la liste d'attente à l'aide d'une `LinkedList<EtatD>`. Pour l'usage, référer à la page `Queue` de l'API de Java.

Exercice 6 :

Tester vos méthodes dans `main`.