TP Automates

En fin de ce document, vous trouverez un appendice contenant quelques rappels sur les classes `Set`*,* `HashSet`*,* `Map`*,* `Stack`*, etc.*

1 Langages

Fichiers à télécharger : `Langage.java`*.*

Le but de cette section est d'écrire une classe représentant des langages. Récupérez la classe à compléter sur DIDEL et écrivez le code manquant dans les fonctions `ajoute`, `union`, `concat`... Vous penserez bien à tester dans un `main` votre code au fur et à mesure.

2 Implémentation des automates

Fichiers à télécharger : `Automate.java`*,* `Etat.java`*,* `EnsEtat.java`*.*

Le but de ce cette section est de simuler le fonctionnement d'un automate fini (non nécessairement déterministe). On aura trois types d'objets correspondant aux trois fichiers à télécharger. La classe `EnsEtat` servira à représenter un ensemble d'états. C'est pourquoi on la définit comme une classe dérivée de `HashSet<Etat>`.

```
public class EnsEtat extends HashSet<Etat> {...}
```

Un état est identifié par un nombre entier. On précisera également si l'état est initial ou terminal. Enfin l'ensemble des transitions qui partent de cet état, sera représenté par une fonction (classe `Map` en Java) qui à une lettre associera un ensemble détats (on est dans le cas non déterministe). Pour que l'ensemble ne puisse contenir deux états ayant le même identifiant, on a redéfini (dans le code à compléter qui vous est fourni) les méthodes `equals()` et `hashCode()`. Deux états ayant le même identifiant auront le même hashcode et seront considérés comme égaux.

```
public class Etat {
    int id;
    boolean estInit;
    boolean estTerm;
    HashMap<Character,EnsEtat> transitions
...}
```

Un automate sera représenté par un ensemble d'états particuliers, on lui rajoute un champ `initiaux` représentant l'ensemble de ses état initiaux (un seul si il est déterministe).

```
public class Automate extends EnsEtat {
    Set<Etat> initiaux;
...}
```

Exercice 1 :

Ajouter à la classe `Etat` les méthodes `EnsEtat succ(char c)`, qui renvoie les successeurs de l'état courant pour une lettre donnée, et `EnsEtat succ()`, qui renvoie l'ensemble de tous les successeurs pour toutes les transitions partant de cet état.

Exercice 2 :

Écrire dans la classe `Etat` une méthode `void ajouteTransition(char c, Etat e)` permettant d'ajouter une transition vers l'état `e`. On récupérera l'`EnsEtat` associé à cette lettre `c` dans la

map `transitions`, pour lui rajouter `e`. Faire attention au cas où aucune transition n'existe deja avec cette lettre.

Exercice 3 :

Ajouter la méthode `boolean ajouteEtatSeul(Etat e)` à la classe `Automate`. Elle doit renvoyer `false` si l'état est déjà présent (`true` sinon), et aussi mettre à jour l'ensemble `initiaux` si besoin est.

Exercice 4 :

Ajouter la méthode `boolean ajouteEtatRecursif(Etat e)` à classe `Automate`. Cette méthode tente d'ajouter l'état `e` en utilisant la méthode de la fonction précédente, puis si l'état n'est pas déjà présent, ajoute récursivement à l'automate les successeurs de l'état `e`.

Exercice 5 :

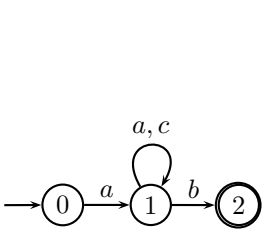
Ajouter à la classe `Automate` une méthode `boolean estDeterministe()`. Il faudra inspecter chaque état pour voir s'il n'a pas plus d'une transition associée à chaque lettre. Autrement dit, il faut voir si dans la map `transitions`, il existe certaines lettres pour lequelles l'`EnsEtat` associé est de taille au moins `2`.

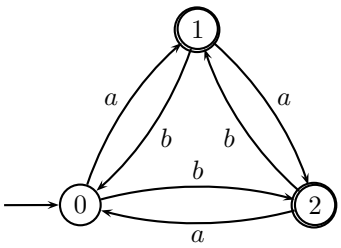
Exercice 6 :

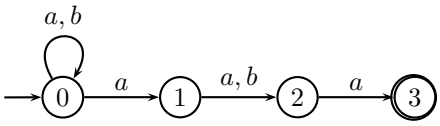
Écrire des méthodes `String toString()` pour les trois classes permettant d'avoir un affichage lisible des informations relatives à ces objets.

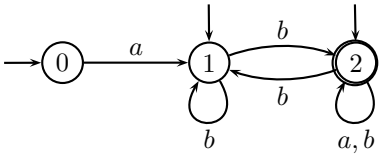
Exercice 7 :

Écrire une méthode `main` (dans la classe `Automate` ou dans un fichier dédié) pour tester vos méthodes avec les automates suivants. Tester la méthode `estDeterministe()`.









3 Reconnaissance

On va écrire la reconnaissance de façon récursive d'abord dans la classe `EnsEtat`, puis dans la classe `Automate`.

Exercice 8 :

Écrire dans la classe `EnsEtat` les méthodes `EnsEtat succ(c)` et `EnsEtat succ()` qui réalisent l'union des fonctions du même nom sur les `Etat` de l'ensemble.

Exercice 9 :

Écrire dans la classe `EnsEtat` une fonction `boolean contientTerminal()` qui renvoie `true` si l'ensemble d'états contient un état terminal, et `false` sinon.

Exercice 10 :

Écrire dans la classe `EnsEtat` une fonction `boolean accepte(String s, int i)` qui renvoie `true` si à partir de l'ensemble d'états courant, et en lisant le mot `s` à partir de sa i-ème lettre, on arrive dans un ensemble d'états contenant un état terminal. En déduire la méthode `boolean accepte(String s)` de la classe `Automate`.

4 Déterminisation

Exercice 11 :

Ajouter une méthode `Set<Character> alphabet()` aux deux classes `Etat` et `EnsEtat`, qui renvoie l'ensemble des lettres étiquetant les transitions partant de l'état (resp. l'ensemble d'états).

Exercice 12 :

Écrire dans la classe `Automate determinise()` la méthode implémentant l'algorithme de déterminisation vu en cours et TD. On en rappelle les grandes lignes :

- On construit un automate A_{det} dont les états correspondent à des ensembles d'états du premier automate A. On aura donc en permanence une `HashMap<EnsEtat,Etat>()` qui, à chaque ensemble d'états rencontré de A, associe un état de A_{det}.
- A_{det} ne contient au départ qu'un seul état initial, associé à l'ensemble des état initiaux de A, et on place cet ensemble d'état initiaux dans une pile (dont les éléments sont des `EnsEtat`).
- Tant que la pile n'est pas vide, on prend son premier élément et, pour chaque lettre de l'alphabet, on cherche l'ensemble d'états successeur. On doit d'abord vérifier qu'il n'a pas déjà été rencontré (il faudra donc avoir un ensemble d'`EnsEtat` qui garde trace de tous les ensembles d'états rencontrés jusqu'à maintenant). S'il est nouveau on rajoute un nouvel état à A_{det} et on met à jour la correspondance via la `Map`, et on rajoute cet état sur la pile. On rajoute ensuite à A_{det} la transition qu'on vient de découvrir.

Exercice 13 :

Tester vos méthodes dans le `main`. Déterminiser les automates de l'Exercice 7.

5 Constructions classiques

Exercice 14 :

Écrire dans la classe `Automate` une méthode `Automate complete()` qui renvoie un automate complet reconnaissant le même langage.

Exercice 15 :

Écrire dans la classe `Automate` une méthode `Automate complementaire()` qui renvoie un automate reconnaissant le complémentaire du langage reconnu par l'automate courant.

Exercice 16 :

Écrire dans la classe `Automate` une méthode `Automate miroir()` qui renvoie un automate reconnaissant le langage miroir du langage reconnu par l'automate courant.

Exercice 17 :

Écrire deux méthodes `Automate union(Automate a)` et `Automate intersection(Automate a)` qui renvoient respectivement un automate reconnaissant l'union et l'intersection des langages reconnus par `a` et par l'automate courant.

6 Expressions rationnelles et Arbres

Fichiers à télécharger : `Arbre.java`, `Feuille.java`, `Unaire.java`, `Binaire.java`.

Pour représenter les expressions rationnelles, on utilisera les classes suivantes. Le symbole sera une lettre de l'alphabet pour toutes les feuilles, `'*'` pour les `Unaire`, et `+` ou `.` pour les `Binaire`.

```
abstract class Arbre{
        char symbole;
}
```

```
class Feuille extends Arbre{}
```

```
class Unaire extends Arbre{
        Arbre fils;
}
```

```
class Binaire extends Arbre{
    Arbre gauche;
    Arbre droit;
}
```

Exercice 18 :

Écrire, pour les trois classes `Feuille`, `Unaire` et `Binaire` des constructeurs, une méthode `toString()`. On pourra opter pour la méthode `toString` pour une représentation infixe mais dans ce cas il faudra mettre des parenthèses nécessaires. Écrire une méthode `main()` qui construit l'arbre de l'expression `(a+b)*bc`.

Exercice 19 :

On veut écrire une méthode `static Arbre lirePostfixe(String expr)` pour construire

l'arbre directement à partir de l'expression rationnelle. On suppose que cette dernière est entrée en notation postfixe, l'avantage étant qu'on n'a pas de parenthèses, et qu'on n'a pas à gérer les priorités d'opérateurs, ce qui simplifie grandement la tâche. On rappelle que la notation postfixe s'obtient en lisant Fils Gauche puis Fils Droit puis le symbole. Par exemple, l'expression `(a+b)*bc` correspond à la notation `ab+*b.c.` postfixe.

Pour implémenter cette méthode, il faut employer une pile d'`Arbre`.

- Chaque fois qu'on lit une lettre, on rajoute sur la pile l'`Arbre` correspondant à une nouvelle feuille étiquetée par cette lettre.
- Si on rencontre un opérateur binaire (`.` ou `+`), on prend les deux premiers éléments sur le dessus de la pile, on leur applique l'opérateur pour créer un nouvel `Arbre`, qu'on repose ensuite sur le dessus de la pile.
- Si on rencontre le symbole `*`, on prend le premier élément de la pile et on lui applique `*` en créant un nouvel `Arbre`, qu'on repose ensuite sur le dessus.

7 Algorithme de Glushkov

Exercice 20 :

Ajouter à la classe `Arbre` un champ `boolean contientMotVide` qui sera mis à jour de façon récursive via le constructeur de chaque sous-classe. Il doit être égal à `true` si et seulement si l'expression associée à l'arbre contient le mot vide.

Exercice 21 :

Ajouter également à la classe `Arbre` une champ `Set<Feuille> premiers` qui contient l'ensemble des feuilles de l'arbre correspondant à des lettres pouvant être le début d'un mot décrit par cette expression. De nouveau, ce champ sera a mis à jour récursivement dans le constructeur de chaque sous classe.

Exercice 22 :

Ajouter de façon similaire un champ `Set<Feuille> derniers`.

Exercice 23 :

Ajouter à la classe `Arbre` une méthode `abstract Map<Feuille,Set<Feuille>> succ()` qu'il faudra redéfinir dans chaque sous-classe et qui calcule la table des successeurs de chaque feuille. Il faudra bien sur se servir de `premiers` et `derniers`.

Exercice 24 :

Conclure en écrivant une méthode qui construit l'automate à partir de l'arbre correspondant à une expression rationnelle. En dehors de l'état initial, il y a exactement un état par feuille de l'arbre.

8 Lecture (resp. Ecriture) dans (resp. depuis) un Fichier

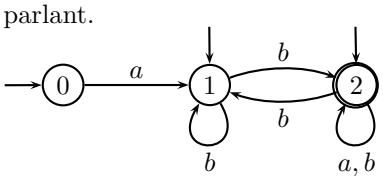
Pour cette partie, vous aurez besoin d'utiliser la classe `Scanner`, qui permet (entre autres) de lire dans un fichier, et de la classe `PrintStream`, qui permet d'écrire du texte dans un fichier. Lire l'appendice pour une description rapide de ces classes. La méthode `String[] split()` de la classe `String` pourra aussi vous être utile.

Pour le moment, la classe `Automate` n'a pas de constructeur pertinent et facile à utiliser. 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. Par la même occasion, on va aussi écrire une méthode qui prend un `Automate` et écrit dans un fichier texte sa représentation. Ce sera l'occasion de découvrir/réviser les méthodes basiques de lecture et d'écriture dans un fichier texte.

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) ;
- ensuite, séparés par des lignes vides, des blocs de lignes décrivant les état un à un.

Chaque bloc état sera formaté de la façon suivante
- Une première ligne décrivant l'`Etat` avec son numéro, suivi de initial et/ou terminal selon si l'état en question est ou non terminal.
- Ensuite, autant de lignes que de lettres différentes pour lesquelles l'état admet une transition. Chacune de ces lignes commencera par le caractère associé à la transition, suivi d'une séquence de numéros correspondant aux états fléchés par une transition portant ce caractère.

Un exemple est surement plus parlant.



peut être représenté par le fichier suivant :

```
3 Etats

0 initial
a 1

1 initial
b 1 2

2 initial terminal
a 2
b 1 2
```

Exercice 25 :

On va commencer par la fonction d'écriture `toFile(String nomFichier)` d'un automate dans un fichier. Commencer par modifier les méthodes `toString()` des classes `Etat`, `EnsEtat` et `Automate`, pour que le format de représentation décrit plus haut soit exactement le résultat de la fonction `toString()` appliqué à l'automate (qui elle-même appelle successivement les fonctions `toString()` de chaque `Etat` pour former chacun des blocs de lignes).

La fonction `toFile` fera donc juste les opérations suivantes :
- création d'un objet `PrintStream` (voir appendice) pour écrire dans le fichier.
- utilisation de cet objet pour écrire dans le fichier le résultat de la méthode `toString()`.

Exercice 26 :

Écrire le constructeur `Automate(String fichier)`. Il doit ouvrir le fichier (et supposer qu'il est bien au format voulu), puis parcourir les lignes une à une pour pouvoir construire l'automate correspondant.

Vérifier en combinant les deux méthodes qu'elles donnent le bon résultat.

9 Appendice : rappels

9.1 `Set,Hashset`

`Set<E>` est une classe abstraite représentant un ensemble d'objet de type `E` (`E` pourra être n'importe quelle classe). Une implémentation possible est `HashSet<E>`. On pourra donc écrire `Set<Integer> s = new HashSet<Integer>();`

Les méthodes les plus courantes pour cette classe sont :

- `boolean contains(E e);`
- `boolean isEmpty();`
- `int size();`
- `boolean add(E e)`, qui tente d'ajouter l'élément à l'ensemble, renvoie `false` s'il est déjà présent, et `true` sinon ;
- `boolean remove(Object o);`
- `boolean addAll(Set<E> s)`, qui rajoute à l'ensemble courant tous les éléments d'un ensemble `s`.

À noter, il est possible de parcourir les éléments d'un `Set<E> s` à l'aide d'une boucle comme celle ci

```
for(E e: s){...}
```

qui peut être vue comme un équivalent pour un tableau `E[] t` de

```
for(int i =0; i<t.length ; i++){E e = t[i]; ....}
```

9.2 `Stack<E>`

Cette classe représente une pile d'éléments de `E`. Les trois méthodes les plus utiles sont :

- `boolean isEmpty();`
- `E pop()`, qui enlève l'élément en haut de la pile et le renvoie ;
- `E peek()`, qui renvoie l'élément en haut sans l'enlever de la pile ;
- `E push(E e)`, qui rajoute l'élément `e` en haut de la pile (et le renvoie, même si ça ne sert pas à grand chose)

9.3 `Map` et `HashMap`

La Classe `Map<E,F>` permet de représenter des fonctions sur un ensemble `E` dans un ensemble `F`. Comme pour `Set`, c'est une classe abstraite, une implémentation pratique est `HashMap<E,F>`. L'ensemble de départ `E` est appelé ensemble de *clés* et l'ensemble `F` ensemble de *valeurs*. Les associations clés-valeurs definissant la fonction sont stockées sous formes de paires. On récupère la valeur associée à une clé par la méthode `F get(E e)` (qui renvoie `null` si aucune valeur n'est associée à cette clé) et on rajoute un nouvelle association clé-valeur à l'aide de la méthode `m.put(E e, F f)`. Il existe plein d'autres méthodes pratiques, voir la doc.

9.4 `Scanner` et `PrintStream`

`Scanner`

Cette classe permet de lire à partir d'une source quelconque spécifiée dans le constructeur. Ainsi, l'instruction suivante permet de créer un `Scanner` pouvant lire ce qui est écrit au clavier dans le terminal

```
Scanner c = new Scanner(System.in)
```

et celle qui suit permet de créer un `Scanner` pouvant lire dans un fichier de nom `monfichier.txt`

```
File f = new File("monfichier.txt");
Scanner c = new Scanner(f);
```

Il faut penser ici à l'objet comme une tete de lecture qui avance dans le fichier. Voici des méthodes basiques (voir la javadoc pour plus de fonctions) dont vous pourrez vous servir :

- `boolean hasNext()`, qui renvoie `false` si on est à la fin du fichier, et `true` sinon ;
- `String next()`, qui lit (et donc avance dans le fichier) et renvoie le prochain mot délimité par des espaces ;
- `int nextInt()`, qui fait de même mais s'attend à trouver une chaine de caractères repréntant un entier, et le renvoie sous forme d'un `int`. S'il trouve autre chose qu'un entier, il lance une exception ;
- `String nextLine()`, qui va jusqu'à la fin de la ligne courante, renvoie tout ce qui a été lu à cette occasion et place la tête de lecture en début de ligne suivante.

`PrintStream`

Cette classe permet de faire de l'écriture dans un fichier. Un constructeur est :

```
PrintStream p = new PrintStream("monfichier.txt");
```

Les opérations d'écriture se font ensuite avec la commande `print` ou `println` que vous connaissez car c'est celle du `System.out.println` (`System.out` est en fait un objet de type `PrintStream`).