Programmation et Langages de Programmation

Licence Informatique, mai 2000

Tout document autorisé, durée 4 heures

Justifier clairement et succinctement vos choix, et décrivez brièvement vos algorithmes lorsque vous le jugez nécessaire.

Toute réponse difficile à comprendre sera considérée comme fausse.

Exercice 1

On considère la classe `Compteur` présentée ci-dessous.

```
import javax.swing.*;
import java.awt.event.*;

public class Compteur extends JFrame {
    private JButton button;
    private int cpt = -1;

    public Compteur() {
        super("Compteur");
        button = new JButton();
        incremente();
        addWindowListener(new WindowAdapter() {
                public void windowClosing(WindowEvent e) {
                    System.exit(0);
                }
            });
        getContentPane().add(button);
        setVisible(true);
        pack();
    }

    public void incremente() {
        button.setText(Integer.toString(++cpt));
    }
}
```

Question 1 Modifier le code de `Compteur` pour garantir l'écoulement d'1 seconde entre deux demandes d'incrémentation.

Question 2 Modifier le code de `Compteur` pour garantir qu'il n'y ait pas de "saut" dans l'incrémentation si deux threads différents utilisent une même instance de `Compteur`.

Question 3 Modifier la classe `Compteur` et rajouter une méthode `main(String [] args)` pour créer n compteurs qui défilent en parallèle les entiers à raison d'un par seconde, n étant donné sur la ligne de commande.

Exercice 2

On souhaite écrire une classe `CircularList` représentant des listes circulaires et qui de plus implémente l'interface `java.util.Collection`.

Une instance de `CircularList` sera non modifiable, ses éléments ayant été spécifiés à la construction à l'aide d'un tableau d'`Object`. La liste circulaire ainsi créée sera par la suite totalement indépendante du tableau qui a servi à sa création, c'est à dire que toute modification de l'un n'aura aucun effet sur l'autre. L'ordre dans la liste circulaire devra respecter celui du tableau d'origine.

On se servira pour implémenter `CircularList` de la classe `java.util.AbstractCollection` qui ne laisse à la charge du programmeur, dans le cas d'une collection non modifiable, que les méthodes `int size()` et `Iterator iterator()` à implémenter (la classe `Iterator` est rappelée en annexe). Les éléments de la liste seront stockés dans un simple tableau.

Question 1 L'itération fournie par `iterator()` démarrera par défaut au premier élément de la liste circulaire. `CircularList` implémentera en plus des méthodes exigées par `Collection` la méthode `void forward()` qui aura pour effet de décaler vers l'élément suivant le début de l'itération fournie par `iterator()`. Sauf dans le cas où la liste est vide, une itération, une fois créée, fournira toujours un élément "suivant", en revenant au début de la liste quand ce sera nécessaire. La méthode `void remove()` de l'itération renverra l'exception `UnsupportedOperationException`.

Question 2 Proposer une deuxième version où cette fois, `CircularList` implémente directement `Iterator` (c'est à dire que désormais, une instance de `CircularList` est également une instance d'`Iterator`). La méthode `forward()` n'étant plus utile, elle sera supprimée.

Question 3 Dans la deuxième version de `CircularList`, rendre possible la suppression d'un élément en implémentant la méthode `remove()` d'`Iterator`.

Problème

On souhaite réaliser une classe `AutomateDeterministe` pour représenter des automates d'états finis déterministes. Un *automate d'états fini déterministe* est un quintuplet (V, Q, i, T, F) où V est un ensemble de symboles appelé *alphabet*, Q est un ensemble fini appelé ensemble des *états*, i est un élément de Q appelé *état initial*, T est un sous-ensemble de Q appelé ensemble des *états terminaux* et F est un sous-ensemble de $Q \times V \times Q$ appelé ensemble des *transitions*. Le premier état d'une transition est appelé son *origine*, le second sa *destination* et le symbole associé son *étiquette*. Dans un automate déterministe, pour toute paire d'état e et de symbole s, il existe au plus une transition ayant pour état origine e et pour étiquette s.

Plus simplement, on représente généralement un automate par un graphe, les états étant représentés par les sommets du graphe, les transitions par les arcs étiquetés par le symbole correspondant. L'état initial sera désigné par une flèche entrante et les états terminaux par une flèche sortante.

Ainsi, l'automate $A = (V = \{a, b\}, Q = \{0, 1, 2, 3\}, i = 0, T = \{0, 2\}, F = \{0a1, 0b2, 1a0, 1b1, 2a2, 2b3, 3b0\})$ est représenté par la figure **??**.

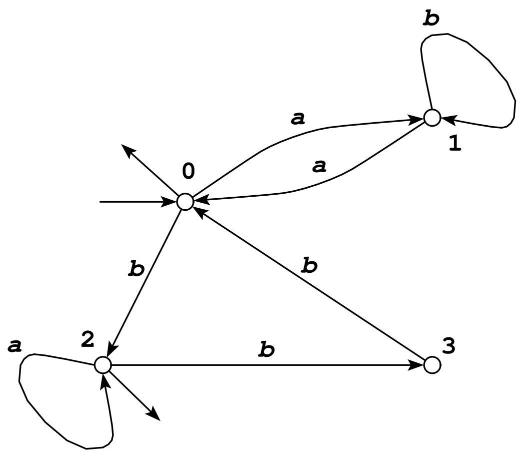

FIG. 1 – *Un automate déterministe*

Un mot sur l'alphabet V est une suite finie de symboles de V. Le mot $m = m_1 m_2 \dots m_k$ sera dit reconnu par l'automate A s'il est possible d'aller de l'état initial à un état terminal par une suite de transitions étiquetées successivement $m_1, \dots, m_k$. Si l'état initial i est également terminal, le mot vide sera reconnu par l'automate. Par exemple, l'automate de la figure **??** reconnaît les mots vide, `aa`, `aba`, `ba`, mais pas les mots `bb` ou `abb`.

Les instances de la classe `AutomateDeterministe` seront non mutables, c'est à dire qu'il sera impossible de modifier un automate déjà créé. Les étiquettes seront des instances de `Object`, un mot étant représenté par un tableau. Un automate sera construit à partir d'un tableau de transitions, les transitions étant des instances de la classe `Transition`. Les états seront représentés par des entiers de 0 à $n-1$, où n désigne le nombre d'états. Le code proposé pourra être modifié si vous le jugez nécessaire pour répondre à une question.

```
public class Transition {
    private final int origine;
    private final int destination;
    private final Object etiquette;

    public Transition(int origine, int destination, Object etiquette) {
        this.origine = origine;
        this.destination = destination;
        this.etiquette = etiquette;
    }
    public int etatOrigine() {
        return origine;
    }
    public int etatDestination() {
        return destination;
    }
    public Object etiquette() {
        return etiquette;
    }
}

public class AutomateDeterministe {
    private int nEtats;
    private Transition [] transitions;
    private int etatInitial;
    private int [] etatsTerminaux;
```

```
    public AutomateDeterministe(int nombreEtats,
                                Transition [] transitions,
                                int etatInitial,
                                int [] etatsTerminaux) {
        this.nEtats = nombreEtats;
        this.transitions = transitions;
        this.etatInitial = etatInitial;
        this.etatsTerminaux = etatsTerminaux;
    }
    ...
}
```

Question 1 Implémenter les méthodes `int nombreEtats()` qui renvoie le nombre d'états de l'automate, `int nombreTransitions()` qui renvoie le nombre de transitions, `int etatInitial()` qui renvoie l'état initial, `int [] etatsTerminaux()` qui renvoie les états terminaux, `boolean etatTerminal(int e)` qui renvoie `true` si et seulement si `e` est un état terminal.

Question 2 Implémenter de façon simple les méthodes `boolean existeTransition(int etatOrigine, Object etiquette)` qui renvoie `true` s'il existe une transition d'origine `etatOrigine` et d'étiquette `etiquette` et `int destination(int etatOrigine, Object etiquette)` qui renvoie l'état destination correspondant. Si la transition n'existe pas, `destination(int etatOrigine, Object etiquette)` lèvera une exception `TransitionInvalideException` (la classe `TransitionInvalideException` n'est pas à fournir).

Modifier le constructeur pour obtenir une version plus efficace de ces méthodes à l'aide de tables de hachage.

Question 3 Modifier le constructeur proposé pour qu'une exception `AutomateNonDeterministeException` soit levée s'il existe deux transitions ayant les mêmes état origine et étiquette. On proposera une implémentation de la classe `AutomateNonDeterministeException`.

Question 4 Implémenter la méthode `boolean reconnait(Object [] m)` qui teste si le mot `m` est reconnu par l'automate.

On s'intéresse maintenant aux automates non déterministes. C'est à dire qu'il peut y avoir plusieurs états initiaux et plusieurs transitions ayant le même couple d'état origine et étiquette.

Question 5 Comment faciliter au client l'utilisation indifférenciée d'instances des deux classes `AutomateNonDeterministe` et `AutomateDeterministe`?

Comment éviter la duplication de code entre ces deux classes?

Donner le code correspondant aux solutions proposées (sans fournir pour l'instant d'implémentation de `AutomateNonDeterministe`).

Question 6 Ecrire une classe `AutomateNonDeterministe` en vous inspirant de la classe `AutomateDeterministe` déjà réalisée. Indication: la méthode `boolean reconnait(Object [] m)` pourra être implémentée à l'aide de la méthode auxiliaire `boolean reconnait(Object [] m, int i, int [] E)` qui renvoie `true` si et seulement si il est possible d'aller d'un état de `E` à un état terminal par une suite de transitions étiquetées par les éléments successifs de `m` d'indice supérieur ou égal à i. *Cette question est assez longue et délicate, prenez le temps de bien justifier tous vos choix. Il est préférable de ne la traiter qu'après avoir répondu à toutes les autres.*

Annexe

```
// Fournit une itération sur une collection.
public interface Iterator {

    // true ssi l'itération possède encore un élément.
    boolean hasNext();

    // renvoie l'élément suivant.
    // Lève l'exception NoSuchElementException s'il ne reste plus
    // d'élément.
    Object next();

    // Supprime de la collection le dernier élément renvoyé par
    // next().
    // Lève l'exception UnsupportedOperationException si la méthode
    // n'est pas utilisable.
    // Lève l'exception IllegalStateException si next() n'a pas été
    // encore invoquée ou si remove() a déja été utilisée depuis la
    // dernière invocation de next().
    void remove();
}
```