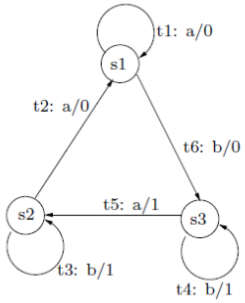
CS312 – Programmation Objet

TP4 : Automates à entrées et sorties (FSMIO)

Ce TP a pour objectif principal de vous initier à la programmation de classes génériques et à l'utilisation des exceptions.

Nous nous intéressons à la programmation *d'automates d'états finis déterministes à entrées et sorties (FSMIO)* définis par:

- un ensemble Q d'états
- dont l'état initial q_0
- un ensemble E d'entrées
- un ensemble S de sorties
- une fonction de transition t : Q x E → Q x S : pour chaque état et chaque entrée il existe un et un seul état successeur est une et une seule sortie.



Cette définition s'illustre sur l'exemple suivant de l'automate A = (Q, q_0, E, S, t):
Q = {s1, s2, s3}, q0 = s1
E = {a, b}, S = {0, 1},
t définie comme suit :

- t(s1, a) = (s1, 0) t(s1, b) = (s3, 0)
- t(s2, a) = (s1, 0) t(s2, b) = (s2, 1)
- t(s3, a) = (s2, 1) t(s3, b) = (s3, 1)

Notre objectif est de réaliser une classe Java permettant de construire de tels automates puis d'effectuer diverses opérations sur eux en commençant par simuler leur fonctionnement en exécutant des transitions.

Partie I : SimpleFSMIO

Afin de réaliser cette classe Java (et les éventuelles autres classes utiles), il est nécessaire de déterminer le type des différents constituants de l'automate que la définition formelle ci-dessus ne précise pas. Dans cette partie, nous allons supposer que:

- les états de l'automate sont identifiés par leur nom (une chaîne de caractères);
- les entrées sont des lettres minuscules (de 'a' à 'z');
- les sorties sont des entiers (int).

Nous supposons également disposer d'une classe `SimpleFonctionTransition` fournissant, outre son constructeur, trois méthodes:

- `SimpleFonctionTransition(String nomsEtats[], char entrées[])` : constructeur dont les paramètres sont les ensembles d'états et d'entrées de l'automate.
- `ajouterTransition(String nomEtatOrig, char entrée, String nomEtatDest, int sortie)` : permet de définir la fonction de transition en spécifiant que pour `(nomEtatOrig, entrée)` elle retourne le résultat `(nomEtatDest, sortie)`.
- `getEtatSuivant(String nomEtatOrig, char entrée)`: retourne l'état destination de la transition ayant pour état d'origine `nomEtatOrig` et pour entrée `entrée`.
- `getSortie(String nomEtatOrig, char entrée)` : retourne la sortie correspondant à la transition ayant comme état d'origine `nomEtatOrig` et pour entrée `entrée`.

L'annexe I propose une réalisation de la classe `SimpleFSMIO`. Pour construire l'automate de l'exemple ci-dessus, nous devons écrire le code suivant:

```
public static void main(String args[]){
        String nomEtats[] = new String[3];
        char entrées[] = new char[2];
        nomEtats[0] = "s1";
        nomEtats[1] = "s2";
        nomEtats[2] = "s3";
```

```
        entrées[0] = 'a';
        entrées[1] = 'b';

        SimpleFSMIO auto = new SimpleFSMIO(nomEtats, "s1", entrées);
        auto.ajouterTransition("s1", 'a', "s1", 0);
        auto.ajouterTransition ("s1", 'b', "s3", 0);
        auto.ajouterTransition ("s2", 'a', "s1", 0);
        auto.ajouterTransition ("s2", 'b', "s2", 1);
        auto.ajouterTransition ("s3", 'a', "s2", 1);
        auto.ajouterTransition ("s3", 'b', "s3", 1);
        System.out.println(auto.getEtatSuivant("s1", 'a'));
        System.out.println(auto.getSortie("s3", 'b'));
}
```

L'appel `auto.getEtatSuivant("s1", 'a')` retournera la valeur `"s1"` tandis que l'appel `auto.getSortie("s3", 'b')` retournera `1`.

Question 1
Modifier les méthodes `faireTransition`, `getEtatSuivant` et `getSortie` de la classe `SimpleFSMIO` afin qu'elles vérifient que leurs paramètres sont valides (i.e. que les états et les entrées existent). En cas d'invalidité, ces méthodes doivent déclencher une exception que l'on définira.

Question 2
Proposer une réalisation de la classe `SimpleFonctionTransition`.

Question 3
Tester la classe `SimpleFSMIO` en construisant l'automate ci-dessus puis en introduisant la séquence d'entrées "abbaabbaaa" et affichant la séquence des sorties correspondantes et le nom des états atteints.

Partie II : FSMIO

On souhaite maintenant réaliser une classe plus générale, permettant de traiter des entrées autres que des caractères et des sorties autres que des entiers, tout en considérant que les états sont toujours identifiés par leur nom (chaîne de caractères). On définit pour cela les interfaces et classes génériques données en Annexe II.

Question 4
Après avoir bien lu les classes de l'Annexe II, écrire une méthode `main` construisant l'automate utilisé en exemple dans la partie précédente à l'aide de la classe FSMIO.

```
public static void main(String args[]){
        // Compléter
        FSMIO<Character, Integer> auto = ...
        // Compléter
}
```

Question 5
Compléter la méthode `doTransition` de la classe `FSMIO` permettant d'exécuter une transition et de mettre à jour l'état courant. Si la transition n'est pas définie, cette méthode doit déclencher une exception.

Question 6
Compléter la méthode `addTransition` de la classe `FSMIO` permettant d'ajouter à l'automate une nouvelle transition entre deux états *après avoir vérifié que ces états ont été préalablement ajoutés (si tel n'est pas le cas, une exception doit être déclenchée)*.

Question 7
Compléter la méthode `getTransition` de la classe `TransitionFunction`. Cette méthode retourne la transition correspondant à un état et une entrée donnés (une exception est déclenchée si la transition n'existe pas).

Question 8
Nous souhaitons ajouter à la classe `TransitionFunction` une méthode retournant la liste des transitions issues d'un état donné. Donner une implémentation de cette méthode.

Question 9
La classe FSMIOString permet de lire la description d'un automate de type FSMIO<String, String> dans un fichier et de construire l'objet associé. Utiliser cette classe pour tester vos réponses aux questions précédentes.

ANNEXE I : SimpleFSMIO

```
public class SimpleFSMIO {
    private SimpleFonctionTransition t;
    private String étatInit;
    private String étatCourant;
    private String états[];
    private char entrées[];


    public SimpleFSMIO(String états[], String étatInit, char entrées[]) {
        t = new SimpleFonctionTransition(états, entrées);
        this.états = états;
        this.entrées = entrées;
        this.étatCourant = this.étatInit = étatInit;
    }

    public void ajouterTransition(String étatOrig, char valEntrée,
                                  String étatDest, int sortie){
        boolean étatOrigValide = false;
        boolean étatDestValide = false;
        boolean entréeValide = false;

        for(int ie = 0; ie < this.états.length &&
                          (!étatOrigValide || !étatDestValide); ie++){
            if(étatOrig.equals(this.états[ie])) étatOrigValide = true;
            if(étatDest.equals(this.états[ie])) étatDestValide = true;
        }

        for(int i = 0; i < this.entrées.length && !entréeValide; i++){
                entréeValide = (valEntrée == this.entrées[i]);
        }

        if (étatOrigValide && étatDestValide && entréeValide){
            this.t.ajouterTransition(étatOrig, valEntrée, étatDest, sortie);
        }
    }

    public int faireTransition(char val){
        int sortie = this.t.getSortie(étatCourant, val);
        this.étatCourant = this.t.getEtatSuivant(étatCourant, val);
        return sortie;
    }

    public void reset(){
        this.étatCourant = this.étatInit;
    }

    public String getEtatCourant(){
        return this.étatCourant;
    }

    public String getEtatSuivant(String s, char e){
        String es = "";
        try{
            es = t.getEtatSuivant(s, e);
        }
        catch (Exception ex){
            System.out.println("Exception " + ex);
            System.exit(0);
        }
        return(es);
    }

    public int getSortie(String s, char e){
```

```
        int es = 0;
        try{
            es = t.getSortie(s, e);
        }
        catch (Exception ex){
            System.out.println("Exception " + ex);
            System.exit(0);       }
        return es;
    }
}

public class SimpleFonctionTransition {
    // Compléter
    public SimpleFonctionTransition(String nomsEtats[], char entrées[]) {
        // Compléter
    }

    public void ajouterTransition(String étatOrig, char entrée,
                                  String étatDest, int sortie){
        // Compléter
    }

    public String getEtatSuivant(String étatOrig, char entrée){
        // Compléter
    }

    public int getSortie(String étatOrig, char entrée){
        // Compléter
    }
}
```

ANNEXE II : FSMIO

```
public interface Input<T> {
    T getInput();
}

public interface Output<T> {
    T getOutput();
}

public class Tag<T1, T2> implements Input<T1>, Output<T2>{

    T1 input;
    T2 output;

    public Tag(T1 i, T2 o) {
        this.input = i;
        this.output = o;
    }

    public T1 getInput(){
        return this.input;
    }

    public T2 getOutput(){
        return this.output;
    }

    public String toString(){
        return this.input + "/" + this.output;
    }
}
```

```
public class State {
    private String name;

    public State(String name){
        this.name = name;
    }

    public String getName(){
        return this.name;
    }

    public boolean equals(State e){
        return this.name.equals(e.name);
    }

    public String toString(){
        return this.name;
    }
}

public class Transition<T1, T2> {
    private State orig, dest;
    private Tag<T1, T2>  tag;

    public Transition(State eorig, Tag<T1, T2> t, State edest){
        this.orig = eorig;
        this.tag = t;
        this.dest = edest;
    }

    public State getOrig(){
        return this.orig;
    }

    public Tag<T1, T2> getTag(){
        return this.tag;
    }

    public State getDest(){
        return this.dest;
    }

    public void setOrig(State o){
        this.orig = o;
    }

    public void setDest(State d){
        this.dest = d;
    }
}

public class TransitionFunction <T1, T2>{
    private List<Transition<T1, T2>> transitions;

    public TransitionFunction( ) {
        this.transitions = new ArrayList<Transition<T1, T2>>();
    }

    public void addTransition(State orig, T1 input, T2 output, State dest){
        this.transitions.add(new Transition<T1, T2>(orig,
                        new Tag<T1, T2>(input, output), dest));
    }

    // Retourne la transition correspondant à l'état orig et à l'entrée input
    public Transition<T1, T2> getTransition(State orig, T1 input){
```

```
        Iterator<Transition<T1, T2>> iter = this.transitions.iterator();
        // Compléter
    }
}


public class FSMIO<T1, T2> {
    private List<State> states;
    private TransitionFunction<T1, T2> tf ;
    private State currentState;
    private State initialState;

    // Constructors
    public FSMIO(List<State> states, State init) {
        this.states = states;
        this.tf = new TransitionFunction<T1, T2>();
        this.currentState = this.initialState = init;
    }

    public FSMIO(State init){
        this.currentState = this.initialState = init;
        this.states = new ArrayList<State>();
        this.states.add(init);
        this.tf = new TransitionFunction<T1, T2>();
    }

    public void addTransition(State orig, T1 input, T2 output, State dest){
        if(/* Compléter */){
            tf.addTransition(orig, input, output, dest);
        }
    }
    public boolean addState(State s){
        boolean done = false;
        if(!this.states.contains(s)){
            this.states.add(s);
            done = true;
        }
        return done;
    }

    public void reset(){
            currentState = initialState;
        }

    public T2 doTransition(T1 input){
        Transition<T1, T2> nt = tf.getTransition(currentState, input);
        // Compléter
    }

    public String toString(){
        return tf.toString();
    }
}

// A FSMIO with Strings in the Tag reading the automaton description from a file
public class FSMIOString{

    FSMIO<String, String> fsms;

    // Constructor : Reading a FSMIO description from a file
    // The file contains one transition per line
    // Every line has the following syntax :
    // statenameorig input output statenamedest
    // The first transition is supposed to start from the initial state
    public  FSMIOString (String filename) {
        BufferedReader fileR;
```

```
        try {
            fileR = new BufferedReader(new FileReader(filename));
            String line = fileR.readLine();
            String [] t;
            State orig, dest;
            t = readTransition(line);
            orig = new State(t[0]);
            this.fsms = new FSMIO<String, String>(orig);
            while(line != null && !line.equals("")){
                orig = new State(t[0]);
                dest = new State(t[3]);
                this.fsms.addState(orig);
                this.fsms.addState(dest);
                this.fsms.addTransition(orig, t[1], t[2], dest);
                line = fileR.readLine();
                t = readTransition(line);
            }
        } catch (FileNotFoundException e) {
            e.printStackTrace();
            System.exit(1);
        } catch (IOException e) {
            e.printStackTrace();
            System.exit(1);
        }
    }

    // returns an array of the four words describing a transition
    private String[] readTransition(String line) {
        String word[] = new String[4];
        int cword = 0;
        int pos = 0;
        while (pos < line.length() && cword <=3){
            word[cword] = new String();
            while (pos < line.length()
                   && (line.charAt(pos) == ' '
                   || line.charAt(pos) == '\t')){ pos++;}
            while (pos < line.length()
                   && line.charAt(pos) != ' '
                   && line.charAt(pos) != '\t'){
                word[cword] += line.charAt(pos++);
            }
            cword++;
        }
        return word;
        }


        public FSMIO<String, String> getFSM(){
            return this.fsms;
        }

        public String toString(){
            return this.fsms.toString();
        }

}
```