INF424 - TP :
Automates en Java (ou en Python)

1 Objectifs pédagogiques :

Le but de ce TP est de comprendre comment passer du formalisme des automates à une implémentation logicielle concrète. Ceci permet de bien assimiler le principe de *reconnaissance des mots par un automate*, et de mettre en évidence les implications pratiques du *déterminisme*. Nous mettrons avant tout l'accent sur les aspects algorithmiques.

NB : Le travail demandé est ici basé sur une programmation en langage Java, mais on pourra tout aussi bien le réaliser dans un autre langage orienté objet, par exemple Python.

2 Introduction

Nous devons trouver un moyen de simuler le travail d'un automate : lorsqu'on proposera une chaine de caractères en entrée, ce simulateur devra répondre `true` si l'automate en question accepte/reconnait cette chaine, et `false` sinon. Rappelons qu'un automate accepte une chaine si et seulement si il a un chemin correspondant au mot et menant d'un état initial à un état terminal.

Par exemple, l'automate suivant accepte la chaine `ababa`, mais pas `abab`.

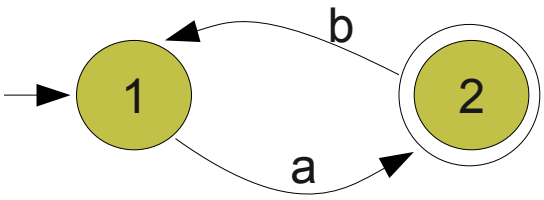

Illustration 1: Aef1, un automate simple

3 Consignes préalables

Chaque automate sera simulé par une classe dérivée de la classe abstraite `Aef` :

```
public abstract class Aef {
       public abstract boolean accepte(String input);
}
```

Pour tester interactivement les différents automates que nous simulerons, nous pourrons utiliser un programme principal dont l'objet sera de soumettre interactivement des chaines de caractères à l'automate désiré, celui-ci étant une instance d'une classe fille de `Aef` :

```
import java.util.*;
public class Simu {
  public static void main(String[] args) {
    Aef aef = new Aef1(); // l'automate a simuler
    Scanner scan = new Scanner(System.in);
    while (true) {
    System.out.println("Veuillez entrer votre chaine de test");
    String input = scan.next();
    if (aef.accepte(input)) System.out.println ("Chaine " + input + " acceptee");
    else System.out.println ("Chaine " + input + " refusee");
}  } }
```

Voyons maintenant les principaux éléments de notre simulateur d'automate.

4 Gestion de la chaine d'entrée

Pour s'exécuter, l'automate va lire la chaine d'entrée caractère par caractère. Une approche impérative classique consiste à travailler directement sur la chaîne avec un index pour connaître la position du caractère courant.

Quelles méthodes de la classe `String` permettront de savoir s'il reste des caractères à traiter, connaître le caractère courant, et connaître le reste de la chaine à traiter ?

5 Simulation d'un automate déterministe

La simulation d'un automate déterministe est simple. En partant de son *état de départ* et en suivant à chaque itération la *transition* concernée par le *caractère lu*, trois situations peuvent être rencontrées :

- On a lu tout le mot et on se retrouve dans un *état terminal* ; le mot est alors accepté.
- On a lu tout le mot et on se retrouve dans un *état non terminal* ; le mot est refusé.
- On ne peut pas lire tout le mot car le caractère lu n'est pas prévu pour l'état courant (c-à-d il n'y a pas de transition correspondant à la configuration rencontrée) ; le mot est alors également refusé.

Voici un exemple de mise en œuvre de `Aef1` correspondant à l'automate donné par l'Illustration 1. Pour simplifier, les états de l'automate sont représentés par des entiers.

```
public class Aef1 extends Aef{
  /**
   * Teste si une chaine est acceptee par l'automate simule
   * Version imperative
   * @param entree - la chaine de caracteres a tester
   * @return - true si la chaine est acceptee, false sinon
   */
  public boolean accepte (String entree){
    int etat = 1; // etat initial de l'automate
    int index = 0; // rang du premier caractere a traiter
    char carlu; // caractere courant

    while (index != entree.length()) {
    // tant qu'il reste des caracteres a traiter
      carlu = entree.charAt(index++); // lecture caractere courant et passage
au suivant
      if ((etat == 1)&&(carlu == 'a')) etat = 2;
      else if ((etat == 2)&&(carlu=='b')) etat=1;
      else return false ; // si aucune transition, entree n'est pas acceptee
    }
    // il n'y a plus rien a lire : est-on dans un etat terminal ?
    if (etat==2) return true; // entree est acceptee
    else return false; // entree n'est pas acceptee
}}
```

Avec le programme principal, testez interactivement cette réalisation sur des chaînes appartenant au langage, et sur d'autres n'appartenant pas au langage.

6 Simulation d'un deuxième automate déterministe

On veut maintenant simuler un autre automate :

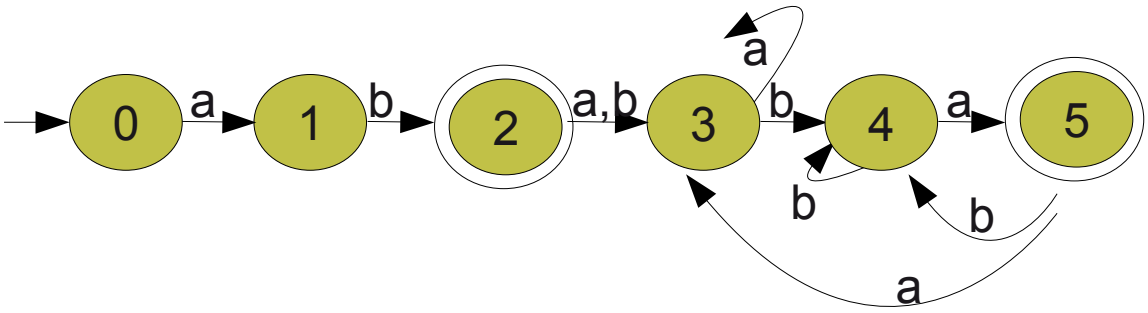

Illustration 2: Aef2, un autre automate déterministe

Décrivez en *langue naturelle* le langage reconnu par cet automate. Donnez quelques exemples et contre-exemples.

Créez une classe `Aef2` correspondant à cet automate, et testez-la.

Que pensez-vous du code (`Aef1` + `Aef2`) ? Quelles critiques pouvez-vous en faire ?

7 Table de transition

Plutôt que de diluer les spécificités de chaque automate dans des lignes de code, nous allons maintenant utiliser des tables de transition. Celles-ci récapitulent toutes les transitions prévues dans l'automate et permettent typiquement de savoir, à partir d'un état courant et d'un symbole lu, quel sera le prochain état. Voici ce que cela donne pour `Aef2` :

Aef2	a	b
0	1	.
1	.	2
2	3	3
3	3	4
4	5	4
5	3	4

Quelle(s) structure(s) de données utiliser en Java pour représenter cette table ? Comment représenter l'absence de transition ? Comment savoir quels sont les états terminaux ?

Créez une nouvelle version `Aef2tt` de l'automate `Aef2` utilisant une telle table de transition.

Que pensez-vous de la réutilisabilité de ce nouveau code ? Dans l'hypothèse où l'on souhaite développer plusieurs automates différents, peut-on factoriser plus de code entre leurs implémentations respectives ? Essayez de maximiser cette réutilisabilité en mettant tout le code possible dans la classe mère `Aef`, et testez en réécrivant une nouvelle version de l'automate `Aef1`.

En quoi le concept de table de transition est-il utile à des logiciels – tel que l'algorithme de déterminisation - qui produisent de nouveaux automates ?

8 Simulation d'un automate non déterministe

Soit `Aefnd2` une version non déterministe de `Aef2` :

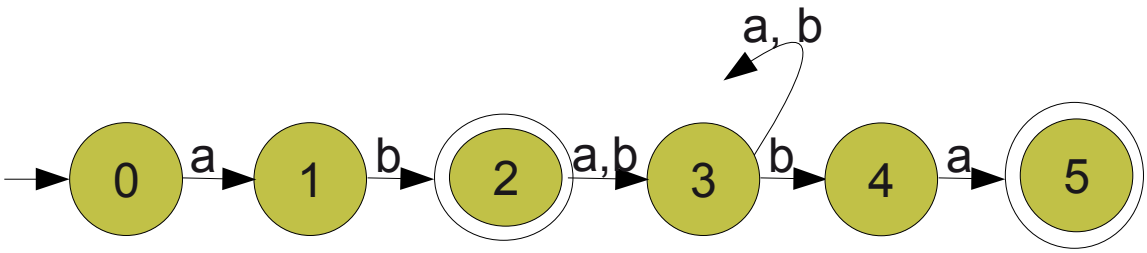

Illustration 3: Aefnd2, une version non déterministe de Aef2

Pourquoi `Aefnd2` n'est-il pas déterministe ?

Que faut-il modifier dans votre code pour pouvoir simuler un tel automate, et de façon générale pour pouvoir simuler un automate non déterministe ? Quel sera l'impact sur les performances de vos algorithmes ?

En vous inspirant des classes `Aef` et `Aef2tt` que vous avez déjà développées, écrivez une classe `Aefnd` et sa classe fille `Aefnd2` qui permettent de simuler `Aefnd2`. Testez-les avec des cas judicieux.

9 Traitements sémantiques

On peut associer des traitements sémantiques aux transitions de l'automate. Chaque traversée d'une transition provoque alors l'exécution du traitement associé, *qui peut être différent selon les transitions*.

Nous représenterons un traitement sémantique par un objet `Runnable`, dont l'exécution pourra être déclenchée au moment adéquat :

```
class Ts1 implements Runnable {
  public void run(){...un traitement sémantique...}
}
class Ts2 implements Runnable {
  public void run(){...un autre traitement sémantique...}
}
...
Runnable ts1 = new Ts1();
Runnable ts2 = new Ts2();
...
ts1.run();
...
ts2.run();
...
```

Modifiez votre classe `Aef2tt` pour associer le traitement « affichage d'un point » à chaque transition de votre automate, ainsi qu'un traitement « afficher 'Etat Terminal' » à la transition 4→5 avec `a`. Testez. Peut-on prévoir le nombre de points affichés ?

Que se passe-t-il si on associe des traitements sémantiques aux transitions d'un automate non déterministe ?