Analyse de Données Structurées - Cours 2

Ralf Treinen

Université Paris Diderot
UFR Informatique
Laboratoire Preuves, Programmes et Systèmes

treinen@pps.univ-paris-diderot.fr

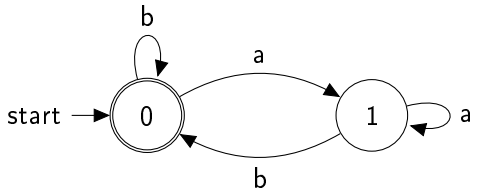
28 janvier 2015

Automates Déterministes

- Étant donné un alphabet fini Σ.
- Un automate est un tuple (Q, q_o, F, Δ) où
  - Q est un ensemble fini d'états
  - $q_0 \in Q$ est l'état initial
  - $F \subseteq Q$ est l'ensemble d'états acceptants
  - $\Delta\colon Q \times \Sigma \to Q$ la fonction de transition.

Exemple ($\Sigma = \{a, b\}$)

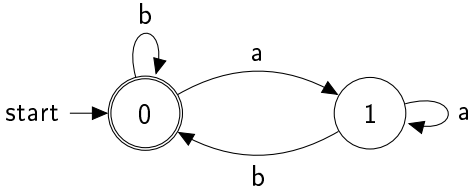
- (Q, q_0, F, Δ) où
  - $Q = \{0, 1\}$
  - $q_0 = 0$
  - $F = \{0\}$
  - $\Delta(0, a) = 1 \quad \Delta(0, b) = 0$
    $\Delta(1, a) = 1 \quad \Delta(1, b) = 0$
- Représentation graphique :



Exécution sur un mot

- Définition d'une fonction d'exécution $\Delta^*\colon \Sigma^* \to Q$ par récurrence :
  - $\Delta^*(q, \epsilon) = q$
  - $\Delta^*(q, xw) = \Delta^*(\Delta(q, x), w)$ où $x \in \Sigma$
- An automate $A = (Q, q_o, F, \Delta)$ *accepte* le mot $w \in \Sigma^*$ ssi $\Delta^*(w) \in F$.
- $\mathcal{L}(A) = \{w \in \Sigma^* \mid \Delta^*(w) \in F\}$
- Un langage L est *régulier* quand il existe un automate A tel que $L = \mathcal{L}(A)$.

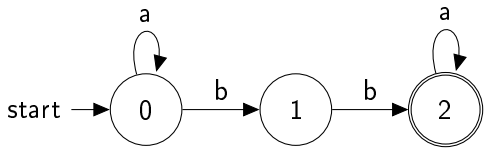
Sur l'exemple



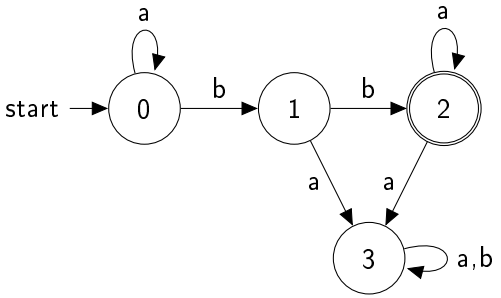
- $\Delta(bbab) = 0 \in F$: mot accepté
- $\Delta(bababa) = 1 \notin F$: mot pas accepté
- $\mathcal{L}(A) = \{\epsilon\} \cup \{wb \mid w \in \Sigma^*\}$

Raccourci : Automate avec une fonction Δ partielle

-

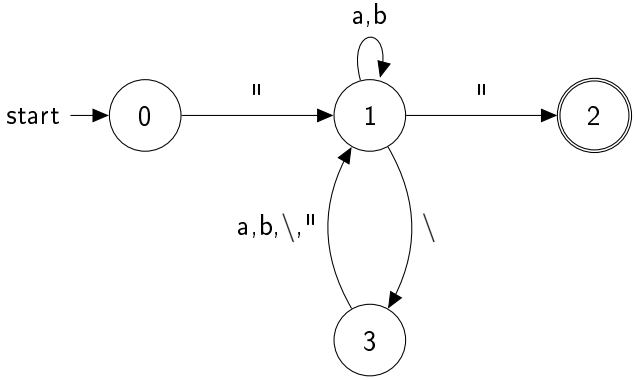


- Raccourci pour :



Exemple : un automate pour les strings de Java

- Simplification : $\Sigma = \{a, b, \backslash, "\}$
- Expression rationnelle vue au cours 1 : $"(a \mid b \mid \backslash.)*"$
- Automate :



Vu dans le cours *Langages et Automates* du S3

- Automates *non*-déterministes
- Automates avec ϵ-transitions
- Algorithme de *déterminisation*
- Algorithme de *minimisation*
- Traduction d'un automate en expression régulière
- Traduction d'une expression régulière en automate
- Un langage est rationnel si et seulement s'il est régulier.

Une classe Java très simple pour un automate donné

- Représenter les états par des entiers $0, 1, \ldots, nbe - 1$
- Représenter l'ensemble des états acceptants par un vecteur de booléens
- Représenter les symboles de l'alphabet par des entiers $0, \ldots, nbs - 1$
- Représenter la fonction de transition par un tableau à deux dimensions.
- On utilise la valeur -1 pour l'état poubelle implicite.

Implémentation naïve I

```
public class Abba {

    static int nbe=3;   /* nombre d'etats */
    static int nbs=2;   /* nombre de symboles */
    static int q0=0;    /* etat initial */
    static boolean[] accepting = {false, false, true};
    static int[][] delta = {{0,1}, {-1,2}, {2,-1}};

    public static int int_of_char(char c) {
        if (c=='a') {
            return 0;
        } else {
            return 1;
```


Implémentation naïve II

```
        }
    }

    public static void main(String[] args) {
        int q=q0;
        for(int i=0; i<args[0].length(); i++) {
            q=delta[q][int_of_char(args[0].charAt(i))];
            if (q==-1) {break;}
        }
        if ( q > -1 && accepting[q] ) {
            System.out.println("accepted");
        } else {
            System.out.println("not␣accepted");
```


Implémentation naïve III

```
        }
    }
}
```

Un problème en pratique

- La taille du tableau des transitions est le nombre d'états $\times$ le nombre de symboles.
- Gourmand en mémoire quand l'alphabet est très grand (Unicode : $>$ 1.000.000 caractères).
- En pratique il y a très peu de flèches qui partent d'un état, toutes étiquetées par une classe de symboles.
- Il convient de trouver une implémentation plus efficace en mémoire, à l'aide des classes de caractères.

Parenthèse : recherche d'un mot dans un texte

- On veut savoir si un mot de longueur r (le motif) apparaît dans un texte de longueur n.
- Cas d'application typique : $n >> r$.
- L'algorithme naïf qu'on a vu en IF1/IP1 a une complexité $n * r$.
- Les automates permettent un algorithme plus efficace !

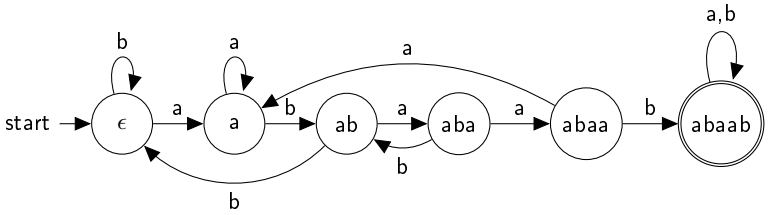
Recherche d'un mot dans un texte : l'idée

- Idée : on construit l'automate qui correspond à l'expression régulière $\Sigma * r \Sigma *$.
- Cet automate s'exécute sur le texte avec une complexité de n : Chaque caractère du texte est traité une seule fois.
- Construction de l'automate : on peut utiliser la traduction générique des expressions régulières vers des automates ...
- ... ou faire une construction directe.

La construction directe de l'automate

- Les états sont les *préfixes* du motif r.
- Exemple : $r = abaab$.
  Préfixes : ϵ, a, ab, aba, $abaa$, $abaab$.
- Signification : l'état signifie le *préfixe le plus long* de r qui est un *suffixe* de la partie du texte vue.
- État initial : ϵ.
  États acceptants : $\{r\}$
- Construction de Δ ? Voir l'exemple.

L'automate pour $\Sigma * abaab\Sigma *$



Construction de l'automate

- Un algorithme efficace pour la construction de l'automate, pour un motif r quelconque ?
- C'est le célèbre algorithme de *Knuth, Morris, et Pratt*.
- Complexité linéaire dans la taille du motif.
- Conséquence : Complexité $n + r$ pour la recherche d'un motif dans un texte (au lieu de $n * r$ avec l'algorithme naïf).
- Voir un cours d'Algorithmique.

L'objectif de l'analyse lexicale

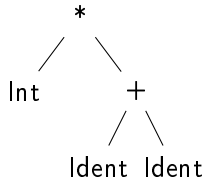
- Découper un texte d'entrée en une séquence de *lexèmes*, et les représenter par des *jetons* (*tokens* en anglais)
- À la base : Classification des lexèmes qui peuvent paraître dans un texte d'entrée, à l'aide des expressions régulières.
- La phase suivante de l'analyse (l'analyse syntaxique, voir plus tard) va travailler sur le résultat de ce découpage : il s'agit d'une *abstraction* du texte d'entrée.

Exemple

- Texte d'entrée :

(	7	5	6	e	2		*		(	e	5	e	7		+		v	a	l	e	u	r	2	)	)

- Jetons :
  PARG INT MULT PARG IDENT PLUS IDENT PARD PARD
- Résultat de l'analyse syntaxique (voir plus tard) :



Définition des catégories lexicales

Sur l'exemple des expressions arithmétiques (on utilise \ comme symbole d'échappement) :

- INT : [0..9]+
- IDENT : [a..zA..Z][a..zA..Z0..9]*
- PARG : \(
- PARD : \)
- MULT : *
- PLUS : \+

Jetons avec arguments

- En réalité, on veut aussi garder certaines informations avec les jetons, comme la valeur d'une constante entière, ou le nom d'un identificateur.
- Certains jetons doivent donc avoir un argument :
  - IDENT(string)
  - INT(int) (c'est bien `int` et pas `string` !)
- Séquence des jetons obtenue sur l'exemple :
  PARG INT(75600) MULT PARG IDENT("e5e7") PLUS IDENT("valeur2") PARD PARD

Ignorer des informations pas pertinentes

L'analyse lexicale sert aussi à faire abstraction de certaines informations dans le texte d'entrée qui ne sont pas pertinentes pour l'analyse du texte. Souvent il s'agit de :

- Les espaces : sont utiles pour indiquer la fin d'un mot. Les espaces sont utiles *pour* l'analyse lexicale, mais une fois le découpage fait on peut les oublier.
- Les commentaires : souvent l'analyse lexicale vérifie l'écriture correcte des commentaires, mais ne les représente pas dans sa sortie.

Exemple

Différents textes d'entrée qui peuvent donner la même séquence de jetons :

- `34 * (x + y)`
- `34*(x+y)`
- `34     *(     x+   y)`
- `34 * (x+y) (* Ceci est un commentaire *)`

Quelle information retenir dans les jetons

- On retient dans les jetons seulement l'information qui est utile pour la suite.
- La distinction entre information utile/inutile dépend de l'application.
- Par exemple : Les commentaires peuvent être utiles à retenir pour certaines applications.
- Il peut être utile de conserver avec les jetons aussi des informations de *localisation* : nom du fichier source, numéro de ligne, numéro de colonne.

Résoudre les ambiguïtés

- L'analyse lexicale va, pour produire le jetons suivant, chercher un *préfixe* du reste du texte qui correspond à une des catégories lexicales, et construire le jeton correspondant.
- Il y a deux sources d'ambiguïtés :
  - Des préfixes de longueurs différents peuvent être reconnues
  - Les expressions régulières peuvent avoir une intersection non vide

Préfixes de longueur différentes reconnues

Exemple :

- Catégorie lexicale :
  - IDENT : `[a..z]+`
- Début du texte d'entrée :
  `xyz`
- Plusieurs possibilités de découpage :
  1. `IDENT("x") IDENT("y") IDENT("z")`
  2. `IDENT("xy") IDENT("z")`
  3. `IDENT("x") IDENT("yz")`
  4. `IDENT("xyz")`
- La règle normale est : on cherche le préfixe *maximal*.

Plusieurs expressions régulières s'appliquent

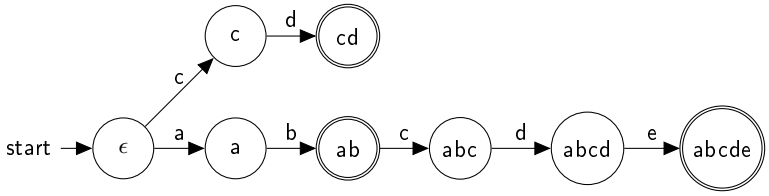
Exemple :

- Catégories lexicales :
  - PUBLIC : `public`
  - IDENT : `[a..z]+`
- Début du texte d'entrée :
  `public publication`
- Plusieurs possibilités de découpage :
  1. `IDENT("public") IDENT("publication")`
  2. `PUBLIC IDENT("publication")`
- La règle normale est : à longueur égale du mot reconnu, c'est la première expression régulière qui gagne.

Exécution de l'automate pour le recherche d'un préfixe maximal

Exemple (artificiel) :

- Catégorie lexicales :
  - AB : `ab`
  - CD : `cd`
  - ABCDE : `abcde`
- Automate :



Analyse de Données Structurées - Cours 2
└Analyse Lexicale

Exécution de l'automate pour le recherche d'un préfixe maximal

- Si on cherche le préfixe le plus *court* reconnu on doit s'arrêter dès qu'on arrive dans un état acceptant.
- Si on cherche le préfixe le plus *long* reconnu on doit continuer à lire tant que possible, et quand on passe par un état acceptant :
  - mémoriser l'état acceptant ;
  - mémoriser la position dans le mot d'entrée

  Si l'automate ne peut plus continuer : remettre le pointeur de lecture dans l'entrée à la position où on a vu le dernier état acceptant.