

# Théorie des Langages 1

## Cours 6 : Preuves, implémentation et applications

L. Rieg

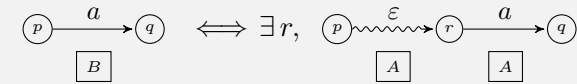
Grenoble INP - Ensimag, 1<sup>re</sup> année

Année 2023-2024

## Rappels sur l'élimination des $\varepsilon$ -transitions

1. Calculer  $\text{Acc}_\varepsilon(p)$ , les états accessibles par  $\varepsilon$ -transitions  
 $\rightsquigarrow$  par itération (cf. cours 1)

2. Construire un automate  $B$  équivalent sans  $\varepsilon$ -transition



### Remarques

- Même  $Q$ ,  $V$  et  $I$ , seuls  $\delta$  et  $F$  changent
- Par construction,  $B$  est sans  $\varepsilon$ -transition

## Correction de l'élimination des $\varepsilon$ -transitions

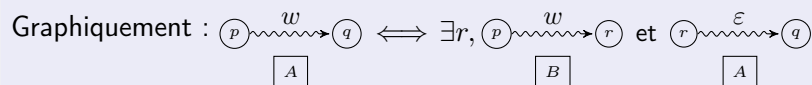
### Théorème

$\forall$  automate  $A$ , l'automate  $B$  défini précédemment est équivalent à  $A$ .

### Lemme intermédiaire : caractérisation des chemins

L'automate  $B$  vérifie la propriété suivante :

Il existe un chemin de  $p$  à  $q$  de trace  $w$  dans  $A$   
 si et seulement si  
 il existe  $r \in Q$  tel qu'il existe un chemin de  $p$  à  $r$  de trace  $w$  dans  $B$   
 et un chemin de  $r$  à  $q$  de trace  $\varepsilon$  dans  $A$ .

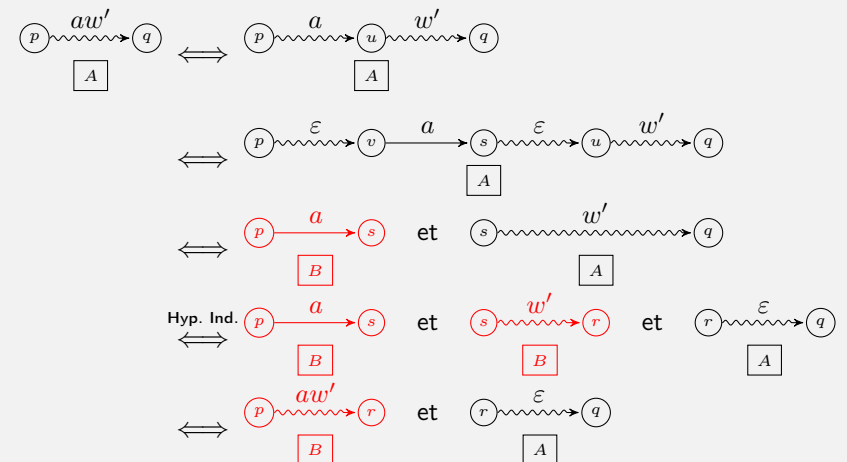


Preuve par induction sur  $w$ .

- Base :  $w = \varepsilon$ . Il suffit de prendre  $r \stackrel{\text{def}}{=} p$ .

## Preuve par induction, suite

- Induction :  $w = aw'$  ( $a \in V$ )



## Preuve du théorème

## Rappels sur la déterminisation

**Idée : suivre tous les chemins en parallèle**

- Entrée : un automate  $A = \langle Q, V, \delta_A, I, F_A \rangle$  **sans  $\varepsilon$ -transitions**
- Sortie : un automate  $B$  déterministe complet équivalent à  $A$

### Définition (Automate des parties)

Etant donné un automate  $A = \langle Q, V, \delta_A, I, F_A \rangle$  **sans  $\varepsilon$ -transition**, on construit l'automate  $B = \langle \mathcal{P}(Q), V, \delta_B, \{I\}, F_B \rangle$ , où :

- $\delta_B$  est défini par
$$\forall P \subseteq Q, \forall a \in V, \delta_B(P, a) = \{q \in Q \mid \exists p \in P : (p, a, q) \in \delta_A\}$$
- $F_B = \{P \subseteq Q \mid P \cap F_A \neq \emptyset\}$

## Propriété (caractérisation des chemins)

### Proposition

Pour tout  $w \in V^*$  et pour tout  $P \subseteq Q$ , on a
$$\delta_B^*(P, w) = \{q \in Q \mid \exists p \in P, \exists \text{ un chemin dans } A \text{ de } p \text{ à } q \text{ de trace } w\}.$$

**Preuve :** par induction sur  $w$

- $w = \varepsilon$  :

## Propriété (caractérisation des chemins)

### Proposition

Pour tout  $w \in V^*$  et pour tout  $P \subseteq Q$ , on a
$$\delta_B^*(P, w) = \{q \in Q \mid \exists p \in P, \exists \text{ un chemin dans } A \text{ de } p \text{ à } q \text{ de trace } w\}.$$

**Preuve :** par induction sur  $w$

- $w = aw'$  :

## Correction de la détermination

### Théorème

*L'automate B est équivalent à A.*

Preuve :

## Implémentation des automates

## Pour les AFD complets

AFD = cas facile

- jamais de choix à faire (déterminisme)
- toujours défini (complétude)

↪ existence + unicité du chemin

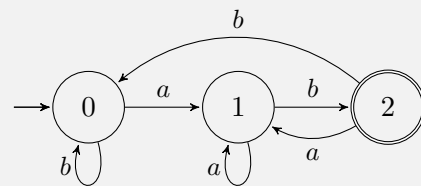
↪ important pour définir l'état d'un système (cf cours d'architecture)

3 méthodes :

- 
- 
- 

Exemple filé :

$L = V^* \cdot \{ab\}$



## Interface

Comment est représenté le mot ?

- par une (tout est connu d'un coup)
- par une (arrivée au compte-goutte)

```
# initialisation de la source  
init_input(...)
```

```
# acces au caractere suivant  
next_char()
```

### Exemples

```
import sys  
in_stream = sys.stderr  
  
def init_input():  
    global in_stream = sys.stdin  
  
def next_char():  
    global in_stream  
    return in_stream.read(1)
```

```
word = ""  
index = 0  
  
def init_input(w):  
    global word = w  
    global index = -1  
  
def next_char():  
    global index+=1  
    return global word[index]
```

## Automates et fin de mot

```
def exec():
    state = i
    init_input()
    ch = next_char()
    while ???:
        state = step(state, ch)
        ch = next_char()
    return state ∈ accepting
```

avec :

- i l'état initial
- step la fonction de transition
- accepting les états acceptants

### Comment reconnaître la fin du mot ?

- Connaître la taille du mot (exemple : chaînes Python, OCaml)  
~> possible uniquement si toute l'entrée est disponible à la fois
- Ajouter un caractère spécial  $\notin V$  de fin de mot ('\\0', EOL/EOF)  
~> marche dans tous les cas (transmission sur réseau)

Ici, on choisit d'ajouter un caractère spécial \$ (ou END)

## Implémentation par table ou par tests

### Par table

Idée :

```
def A = {0: {'a': 1, 'b': 0},
         1: {'a': 1, 'b': 2},
         2: {'a': 1, 'b': 0}}

def step(state, ch):
    return A.transitions[state][ch]
```

~> l'automate est une donnée

Coût = lecture mémoire

### Par tests

Idée :

```
def step(state, ch):
    if state == 0:
        if ch == 'a':
            return 1
        elif ch == 'b':
            return 0
    elif state == 1:
        ...
```

~> l'automate est un programme

Coût = tests + sauts

## Implémentation par fonctions

Idée :

~> pas de boucle while ni de fonction step

```
def state0():
    ch = next_char()
    if ch == 'a':
        return state1()
    elif ch == 'b':
        return state0()
    elif ch == '$':
        return False

def state1():
    ch = next_char()
    if ch == 'a':
        return state1()
    elif ch == 'b':
        return state2()
    elif ch == '$':
        return False

def state2():
    ch = next_char()
    if ch == 'a':
        return state1()
    elif ch == 'b':
        return state0()
    elif ch == '$':
        return True
```

- plus modulaire que la boucle while
- permet de faire du calcul
- Coût = appel de fonction

automate = ensemble de fonctions  
lecture = appeler l'état initial  
état courant = la fonction qui s'exécute

```
def exec_v2():
    init_input()
    return state0()
```

## Quelle méthode pour les AFND ?

- Par fonction  
une fonction est spécifique à un état...  
**Exponentiel avec plusieurs états !** (nb de chemins)
- Par matrice ou tests  
variable state = ensemble d'états  
+ itération sur state pour calculer l'état suivant  
~> deux variables : state, new\_state  
~> boucles sur state paralléliser les tests / lectures mémoire

```
def step(A, state, ch):
    new_state = set.empty()
    for q in state:
        new_state.add(
            A.transitions[
                state, ch])
    return new_state

def exec(A):
    state = A.init
    init_input()
    ch = next_char()
    while (ch != '$'):
        state = step(A, state, ch)
        ch = next_char()
    return A.accepting.inter(state)
```

## Comparaison AFD/AFND

### AFD

- exécution très rapide  $O(|w|)$  (jamais de choix à faire)
- plus gros que AFND (exponentiellement !)

### Cas d'utilisation

- Vitesse exigée
- Beaucoup d'utilisation
- Construction de l'automate à l'avance

→ ex : compilateur

### AFND

- exécution plus lente  $O(|w| \cdot |Q|)$  (ensembles d'états)
- plus compacts que AFD

### Cas d'utilisation

- Contraintes d'espace
- Utilisation unique (ou faible)
- Construction de l'automate à l'utilisation
- Facile dans les circuits

→ ex : expressions régulières

Choix AFD/AFND = compromis espace/temps

## Exemple d'utilisation : analyse lexicale

Première partie d'un compilateur : reconnaître les programmes corrects

### Étapes :

1. Décrire les programmes corrects      expression régulière / grammaire
2. Construire un AFND
3. Le déterminer
4. En faire une implémentation par goto (fonction)

### En plus

- faire du calcul vs. OUI/NON      voir TP/projet
- gestion des erreurs (cas else des if)

En TP : reconnaissance des constantes flottantes en Python

## Application des automates

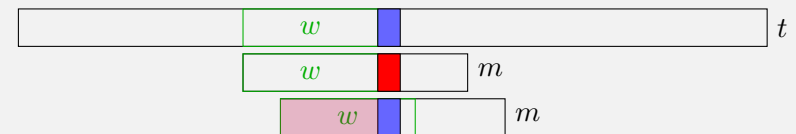
## Application 1 : algo KMP

Contexte : recherche d'un motif  $m$  dans un texte  $t$

### Contraintes :

- construction de l'automate linéaire (en  $|m|$ )
- parcours du texte **en ligne**      = caractère par caractère  
pas de retour en arrière possible  
Complexité linéaire en  $|t|$

Idée : décaler le motif de « **juste ce qu'il faut** » en cas d'erreur  
**sans rater d'occurrence de  $m$**



Que peut-on dire de  par rapport à  $w$  ?

## Calcul des bords d'un mot

### Définition (Bord)

Le **bord** d'un mot  $w \neq \varepsilon$  est son plus grand préfixe strict qui en est également un suffixe. On le note  $\varphi(w)$ .

Pourquoi strict ?

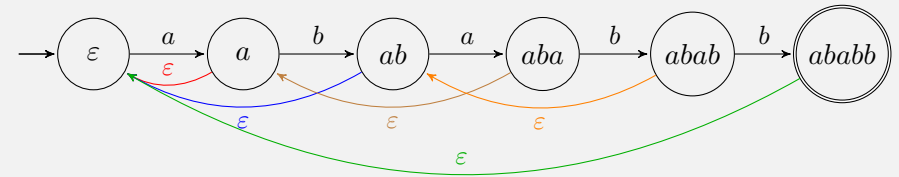
### Calcul de $\varphi(w)$

- pour  $a \in V$ ,  $\varphi(a) = ?$
- pour  $w \in V^+$ ,  $a \in V$ ,  $\varphi(wa) = ?$

**Idée : si problème après  $w$ , réessayer avec  $\varphi(w)$  !**

## Exemple : recherche de $ababb$

Recherche de  $m = ababb$  dans  $t = abaababb$ .



Calcul de  $\varphi(w)$  pour  $w$  préfixe de  $m$  :

$\varphi(a) =$	$\varphi(ab) =$
$\varphi(aba) =$	$\varphi(abab) =$
$\varphi(ababb) =$	

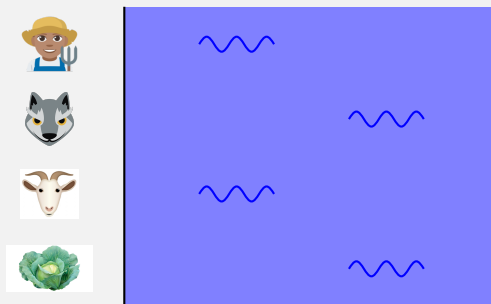
Complexité de la construction :

Taille de l'AFD complet :

Complexité de la lecture :

## Application 2 : stratégie dans un jeu

Jeu du fermier (cf. exercice 14 du recueil de TD)



### Exercice

1. Représenter le problème par un automate, en précisant le vocabulaire choisi.
2. Comment déterminer une stratégie à partir de l'automate ?  
Quelles sont les stratégies optimales ?

## Solution