

# Expressions régulières

OPTION INFORMATIQUE - TP n° 3.8 - Olivier Reynet

## À la fin de ce chapitre, je sais :

- ✎ faire le lien entre un ensemble de mots et une expression régulière
- ✎ utiliser la syntaxe des expressions régulières
- ✎ utiliser la sémantique des expressions régulières pour simplifier une expression régulière
- ✎ utiliser le filtrage (pattern matching) sur un type algébrique
- ✎ définir et utiliser un type algébrique

## A Exprimer par des mots des expressions régulières

Tenter de décrire en français les langages dénotés par les expressions régulières suivantes :

A1.  $\Sigma\Sigma$

**Solution :** Le langage des mots de longueur deux.

A2.  $(\epsilon + \Sigma)(\epsilon + \Sigma)$

**Solution :** Le langage des mots dont la longueur est au plus deux.

A3.  $(\Sigma\Sigma)^*$

**Solution :** Le langage des mots de longueur paire.

A4.  $\Sigma^* a \Sigma^*$

**Solution :** Le langage des mots comportant au moins une occurrence de  $a$ .

A5.  $\Sigma^* ab \Sigma^*$

**Solution :** Le langage des mots comportant au moins une occurrence du facteur  $ab$

A6.  $\Sigma^* a \Sigma^* b \Sigma^*$

**Solution :** Le langage des mots comportant au moins une occurrence de  $a$  puis au moins une occurrence de  $b$ .

A7.  $(ab)^*$

**Solution :** Le langage des mots commençant par  $a$ , finissant par  $b$  et où les  $a$  et les  $b$  n'apparaissent jamais consécutivement.

## B Des mots aux expressions régulières

Soit l'alphabet  $\Sigma = \{a, b\}$ . Trouver une expression régulière qui dénote l'ensemble des mots :

B1. de longueur paire

**Solution :**  $(\Sigma\Sigma)^*$

B2. de longueur impaire

**Solution :**  $(\Sigma\Sigma)^*\Sigma$

B3. de longueur au moins un et au plus trois

**Solution :**  $\Sigma(\varepsilon|\Sigma|\Sigma\Sigma)$

B4. qui possèdent un nombre pair de  $b$

**Solution :**  $(a^*ba^*ba^*)^*$

B5. qui possèdent un nombre impair de  $a$

**Solution :**  $b^*a(\varepsilon|b^*ab^*a)^*b^*$

B6. qui possèdent un nombre de  $a$  multiple de 3

**Solution :**  $b^*(ab^*ab^*a)^*b^*$

## C Combien de mots dans le langage?

Soit l'alphabet  $\Sigma = \{a, b\}$ . Combien de mots de longueur 100 sont-ils dans  $\mathcal{L}_{ER}(e)$ ?

C1.  $e = a(a|b)^*b$

**Solution :** Les premières et dernières lettres étant fixées, il reste 98 lettres au milieu à choisir entre a et b. Cela fait donc  $2^{98}$  mots.

C2.  $e = a^* bab^*$

**Solution :** Les lettres du milieu étant fixées, on peut mettre :

- zéro a à gauche et 98 b à droite
- un a à gauche et 97 b à droite
- ....
- 97 a à gauche et un b à droite
- 98 a à gauche et 0 b à droite.

Cela fait donc 99 mots.

C3.  $e = (a|ba)^*$  (On peut utiliser  $(u_n)_{n \in \mathbb{N}}$  le nombre de mots de longueur  $n$  dans  $\mathcal{L}_{ER}(e)$ .)

**Solution :** Si on définit  $(u_n)_{n \in \mathbb{N}}$  comme le nombre de mots de longueur  $n$  dans  $\mathcal{L}_{ER}(e)$ , alors on peut dire que lorsqu'on choisit une lettre dans un mot de 100 lettres, il nous reste à choisir soit une lettre dans un mot de 99 lettres, soit deux lettres dans un mot de 98 lettres. Ce qui s'écrit :  $u_n = u_{n-1} + u_{n-2}$ . On a  $u_0 = 1$ , le mot vide et  $u_1 = 1$ ,  $a$ . On reconnaît la suite de Ficonnacci. On a donc  $u_{100} = \alpha\phi^{100} + \beta\phi'^{100}$  avec  $\alpha = \frac{1}{2} \left(1 + \frac{1}{2\sqrt{5}}\right)$ ,  $\beta = \frac{1}{2} \left(1 - \frac{1}{2\sqrt{5}}\right)$ ,  $\phi = \frac{1+\sqrt{5}}{2}$  et  $\phi' = -\frac{1}{\phi}$ .

## D Simplification d'expressions régulières

Simplifier les expressions régulières suivantes :

D1.  $\varepsilon|ab|abab(ab)^*$

**Solution :** On passe par la sémantique des expressions régulières.

$$\mathcal{L}_{ER}(e) = \mathcal{L}_{ER}(\varepsilon) \cup \mathcal{L}_{ER}(ab) \cup \mathcal{L}_{ER}(abab(ab)^*) \quad (1)$$

$$= \mathcal{L}_{ER}(\varepsilon) \cup \mathcal{L}_{ER}(ab) \cup \mathcal{L}_{ER}(abab(ab)^*) \quad (2)$$

$$= \{\varepsilon\} \cup \{ab\} \cup \{abab \bigcup_{n \geq 0} (ab)^n\} \quad (3)$$

$$= \{\varepsilon\} \cup \{ab\} \cup \{\bigcup_{n \geq 2} (ab)^n\} \quad (4)$$

$$= \{\varepsilon\} \cup \{\bigcup_{n \geq 1} (ab)^n\} \quad (5)$$

$$= \{\bigcup_{n \geq 0} (ab)^n\} \quad (6)$$

$$= \mathcal{L}_{ER}((ab)^*) \quad (7)$$

$$(8)$$

On a donc  $e = (ab)^*$ .

D2.  $aa(b^*|a)|a(ab^*|aa)$

**Solution :** De la même manière, on trouve :  $e = aa(b^*|a)$

D3.  $a(a|b)^*|aa(ab^*)|aaa(a|b)^*$

**Solution :** On trouve :  $e = a(a|b)^*$ . On remarquera que certains langages sont inclus dans les autres. Par exemple  $\mathcal{L}_{ER}(aa) \subset \mathcal{L}_{ER}(aab^*)$

## E Miroirs et induction

■ **Définition 1 — Mot miroir.** Le mot miroir d'un mot  $w = a_1 a_2 \dots a_n$  est  $w^R = a_n a_{n-1} \dots a_1$ .

■ **Définition 2 — Langage miroir.** Soit  $\mathcal{L}$  un langage sur  $\Sigma$ . Le langage miroir de  $\mathcal{L}$  est :

$$\mathcal{L}^R = \{w^R, w \in \mathcal{L}\} \quad (9)$$

E1. Montrer que pour deux mots  $v$  et  $w$  d'un langage  $\mathcal{L}$  on a  $(vw)^R = w^R v^R$ .

**Solution :** Il suffit de revenir à la définition : soit  $v = a_1 a_2 \dots a_n$  et  $w = b_1 b_2 \dots b_n$ .  
On a  $(vw)^R = (a_1 a_2 \dots a_n b_1 b_2 \dots b_n)^R = b_n \dots b_1 a_n \dots a_1 = w^R v^R$ .

E2. Montrer que si  $\mathcal{L}_1$  et  $\mathcal{L}_2$  sont deux langages, on a  $\mathcal{L}_1^R \cup \mathcal{L}_2^R = (\mathcal{L}_1 \cup \mathcal{L}_2)^R$ .

**Solution :**  $\mathcal{L}_1^R \cup \mathcal{L}_2^R = \{w^R, w \in \mathcal{L}_1\} \cup \{w^R, w \in \mathcal{L}_2\} = \{w^R, w \in \mathcal{L}_1 \cup \mathcal{L}_2\} = (\mathcal{L}_1 \cup \mathcal{L}_2)^R$

E3. Montrer que si  $\mathcal{L}_1$  et  $\mathcal{L}_2$  sont deux langages, on a  $\mathcal{L}_1^R \mathcal{L}_2^R = (\mathcal{L}_2 \mathcal{L}_1)^R$ .

**Solution :**  $\mathcal{L}_1^R \mathcal{L}_2^R = \{v^R w^R, v \in \mathcal{L}_1 \wedge w \in \mathcal{L}_2\} = \{(wv)^R, v \in \mathcal{L}_1 \wedge w \in \mathcal{L}_2\} = \{(wv)^R, wv \in \mathcal{L}_2 \mathcal{L}_1\} = \{u^R, u \in \mathcal{L}_2 \mathcal{L}_1\} = (\mathcal{L}_2 \mathcal{L}_1)^R$

E4. Montrer que si  $\mathcal{L}$  est un langage, on a  $(\mathcal{L}^*)^R = (\mathcal{L}^R)^*$ .

**Solution :**  $(\mathcal{L}^*)^R = \{w^R, w \in \mathcal{L}^*\} = \bigcup_{n \geq 0} \{w^R, w \in \mathcal{L}^n\} = \bigcup_{n \geq 0} \{w, w \in (\mathcal{L}^n)^R\}$ .

Or, on peut montrer par récurrence sur  $n$ , d'après la définition inductive des puissances d'un langage, que  $(\mathcal{L}^n)^R = (\mathcal{L}^R)^n$ .

**(Initialisation)** comme  $\varepsilon = \varepsilon^R$ , on a  $(\mathcal{L}^0)^R = (\mathcal{L}^R)^0$ .

**(Hérédité)** supposons que  $(\mathcal{L}^n)^R = (\mathcal{L}^R)^n$ . Alors on a :  $(\mathcal{L}^{n+1})^R = (\mathcal{L} \mathcal{L}^n)^R = (\mathcal{L})^R (\mathcal{L}^n)^R = (\mathcal{L})^R (\mathcal{L}^R)^n = (\mathcal{L}^R)^{n+1}$ .

**(Conclusion)**  $(\mathcal{L}^n)^R = (\mathcal{L}^R)^n$  est vrai pour tout  $n$ .

C'est pourquoi,  $(\mathcal{L}^*)^R = \bigcup_{n \geq 0} \{w, w \in (\mathcal{L}^R)^n\} = (\mathcal{L}^R)^*$ .

E5. Définir de manière inductive une fonction miroir dont le paramètre d'entrée est une expression régulière  $e$  et qui renvoie l'expression régulière miroir  $e^R$  qui dénote le langage  $\mathcal{L}_{ER}^R(e)$ .

**Solution :** On définit la fonction miroir  $m : ER \longrightarrow ER$  comme suit

**(Base (i))**  $\emptyset^R = \emptyset$ ,

**(Base (ii))**  $\varepsilon^R = \varepsilon$ ,

**(Base (iii))**  $\forall a \in \Sigma, a^R = a$ ,

**(Règle de construction (i))**  $\forall e_1, e_2 \in ER, (e_1 | e_2)^R = e_1^R | e_2^R$

**(Règle de construction (ii))**  $\forall e_1, e_2 \in ER, (e_1 e_2)^R = e_2^R e_1^R$

**(Règle de construction (iii))**  $\forall e \in ER, (e^*)^R = (e^R)^*$ .

E6. Démontrer que  $\forall e \in ER, \mathcal{L}_{ER}(e^R) = \mathcal{L}_{ER}^R(e)$ , c'est-à-dire démontrer que l'algorithme de construction inductive de l'expression régulière miroir est correct.

**Solution :** On démontre par induction la correction de  $m$  :

**(Cas de base (i))**  $\mathcal{L}_{ER}(\emptyset^R) = \mathcal{L}_{ER}(\emptyset) = \{\emptyset\} = \mathcal{L}_{ER}^R(\emptyset)$ . Le miroir du langage vide est le langage vide.

**(Cas de base (ii))**  $\mathcal{L}_{ER}(\varepsilon^R) = \mathcal{L}_{ER}(\varepsilon) = \{\varepsilon\} = \mathcal{L}_{ER}^R(\varepsilon)$ .

**(Cas de base (iii))**  $\forall a \in \Sigma, \mathcal{L}_{ER}(a^R) = \mathcal{L}_{ER}(a) = \{a\} = \mathcal{L}_{ER}^R(a)$ .

**(Pas d'induction (i))** On suppose maintenant qu'on dispose de deux expressions régulières  $e_1, e_2 \in ER$  telles que  $\mathcal{L}_{ER}(e_1^R) = \mathcal{L}_{ER}^R(e_1)$  et  $\mathcal{L}_{ER}(e_2^R) = \mathcal{L}_{ER}^R(e_2)$ . On cherche à construire le langage miroir de l'union de ces deux expressions en utilisant la sémantique des expressions régulières, la définition inductive des expressions miroirs et l'hypothèse d'induction :

$$\mathcal{L}_{ER}((e_1 | e_2)^R) = \mathcal{L}_{ER}(e_1^R | e_2^R) \quad \text{définition du miroir (10)}$$

$$= \mathcal{L}_{ER}(e_1^R) \cup \mathcal{L}_{ER}(e_2^R) \quad \text{sémantique ER (11)}$$

$$= \mathcal{L}_{ER}^R(e_1) \cup \mathcal{L}_{ER}^R(e_2) \quad \text{hypothèse d'induction (12)}$$

$$= (\mathcal{L}_{ER}(e_1) \cup \mathcal{L}_{ER}(e_2))^R \quad \text{résultat précédent (13)}$$

$$= \mathcal{L}_{ER}^R(e_1 | e_2) \quad \text{sémantique ER (14)}$$

**(Pas d'induction (ii))** Avec la même hypothèse sur  $e_1$  et  $e_2$ , on cherche maintenant à construire le langage miroir de la concaténation de ces deux expressions :

$$\mathcal{L}_{ER}((e_1 e_2)^R) = \mathcal{L}_{ER}(e_2^R e_1^R) \quad \text{définition du miroir (15)}$$

$$= \mathcal{L}_{ER}(e_2^R) \mathcal{L}_{ER}(e_1^R) \quad \text{sémantique ER (16)}$$

$$= \mathcal{L}_{ER}^R(e_2) \mathcal{L}_{ER}^R(e_1) \quad \text{hypothèse d'induction (17)}$$

$$= (\mathcal{L}_{ER}(e_1) \mathcal{L}_{ER}(e_2))^R \quad \text{résultat précédent (18)}$$

$$= \mathcal{L}_{ER}^R(e_1 e_2) \quad \text{sémantique ER (19)}$$

**(Pas d'induction (iii))** On suppose maintenant qu'on dispose d'une expression régulière  $e \in ER$  telle que  $\mathcal{L}_{ER}(e^R) = \mathcal{L}_{ER}^R(e)$ . On cherche maintenant à construire le langage miroir de la

fermeture de Kleene de l'expression  $e$  :

$$\mathcal{L}_{ER}((e^*)^R) = \mathcal{L}_{ER}((e^R)^*)$$

définition du miroir (20)

$$= (\mathcal{L}_{ER}(e^R))^*$$

sémantique ER (21)

$$= (\mathcal{L}_{ER}^R(e))^*$$

hypothèse d'induction (22)

## F Implémentation d'un type expression régulière

■ **Définition 3 — Syntaxe des expressions régulières.** L'ensemble des expressions régulières  $\mathcal{E}_R$  sur un alphabet  $\Sigma$  est défini inductivement par :

**(Base)**  $\{\emptyset, \varepsilon, \} \cup \Sigma \in \mathcal{E}_R$ ,

**(Règle de construction (union))**  $\forall e_1, e_2 \in \mathcal{E}_R, e_1 \mid e_2 \in \mathcal{E}_R$

**(Règle de construction (concaténation))**  $\forall e_1, e_2 \in \mathcal{E}_R, e_1 e_2 \in \mathcal{E}_R$ ,

**(Règle de construction (fermeture de Kleene))**  $\forall e \in \mathcal{E}_R, e^* \in \mathcal{E}_R$ .

F1. Créer un type algébrique regexp OCaml qui représente une expression régulière selon la définition 3.

**Solution :**

```
type regexp =      EmptySet
  | Epsilon
  | Letter of char
  | Sum of  regexp * regexp
  | Concat of regexp * regexp
  | Kleene of regexp ;;
```

F2. Créer en OCaml une variable e représentant l'expression régulière  $(a^* \mid b)c$  sur l'alphabet  $\Sigma = \{a, b, c\}$ .

**Solution :**

```
let e = Concat (Sum (Kleene (Letter 'a'), Letter 'b'), Letter 'c');
```

F3. Créer une variable esigma de type regexp dont le langage dénote l'alphabet  $\Sigma = \{A, B, C\}$ .

**Solution :**

```
let a = Letter 'A';;
let b = Letter 'B';;
let c = Letter 'C';;
let esigma = Sum (Sum (a, b), c);;
```

F4. Créer une variable `esigmastar` de type `regexp` dont le langage dénote l'alphabet  $\Sigma^*$ .

**Solution :**

```
let esigmastar = Kleene esigma;;
```

F5. Créer une fonction récursive et utilisant le pattern matching de signature `regexp_to_string : regexp -> string` qui permet d'afficher lisiblement un type `regexp` sur la console. Par exemple, pour l'expression `esigma`, celle-ci renvoie la chaîne de caractère `((A|B)|C)`, pour `e` elle renvoie `((a)*|b)c`. On rappelle que la concaténation de chaîne de caractères se fait via l'opérateur `^` en OCaml.

**Solution :**

```
let rec regexp_to_string e =
  match e with
  | EmptySet -> "{}"
  | Epsilon -> "Epsilon"
  | Letter a -> String.make 1 a
  | Sum (e1, e2) -> "(" ^ (regexp_to_string e1) ^ "|" ^ (
    regexp_to_string e2) ^ ")"
  | Concat (e1, e2) -> "(" ^ (regexp_to_string e1) ^ "" ^ (
    regexp_to_string e2) ^ ")"
  | Kleene e -> "(" ^ (regexp_to_string e) ^ ")*"
;;
regexp_to_string e;;
regexp_to_string esigma;;
regexp_to_string esigmastar;;
```

## G Langages vides, réduits au mot vide ou finis

G1. Créer une fonction de signature `is_empty_language : regexp -> bool` qui teste si une expression régulière dénote le langage vide.

**Solution :**

```
let rec is_empty_language e = match e with
  | EmptySet -> true
  | Epsilon | Letter _ | Kleene _ -> false
  | Sum (e1,e2) -> is_empty_language e1 && is_empty_language e2
  | Concat(e1,e2) -> is_empty_language e1 || is_empty_language e2
is_empty_language e
is_empty_language (Kleene EmptySet)
is_empty_language EmptySet
is_empty_language a
is_empty_language (Sum (a, EmptySet))
is_empty_language (Concat (a, EmptySet))
```

- G2. Créer une fonction de signature `is_reduced_to_epsilon : regexp -> bool` qui teste si une expression régulière dénote le langage réduit au mot vide.

**Solution :**

```
let rec is_reduced_to_epsilon e = match e with
| Epsilon -> true
| EmptySet | Letter _ -> false
| Kleene e -> if e = EmptySet then true
               else is_reduced_to_epsilon e
| Sum (e1,e2) | Concat(e1,e2) -> is_reduced_to_epsilon e1
                                && is_reduced_to_epsilon e2
```

- G3. Créer une fonction de signature `is_finite_language : regexp -> bool` qui teste si une expression régulière dénote un langage fini, c'est-à-dire qui comporte un nombre fini de mots.

**Solution :**

```
let rec is_finite_language e = match e with
| EmptySet | Epsilon | Letter _ -> true
| Sum (e1, e2) | Concat (e1, e2) -> is_finite_language e1
                                     && is_finite_language e2
| Kleene e -> is_reduced_to_epsilon e || is_empty_language e;;

is_finite_language e;;
is_finite_language EmptySet;;
is_finite_language Epsilon;;
is_finite_language (Kleene EmptySet);;
is_finite_language (Kleene Epsilon);;
is_finite_language (Kleene a);;
```

## H Tester l'appartenance d'un mot à un langage rationnel

- H1. Écrire une fonction de signature `matches_regex : regexp -> string -> bool` qui statue sur le fait qu'un mot appartient à un langage dénoté par une expression rationnelle. On pourra s'appuyer sur les fonctions `String.sub` et `String.length`.

**Solution :**

```
let rec matches_regex regex word =
  match word, regex with
  | "", Epsilon -> true
  | s, Letter ch when String.length s = 1 && s.[0] = ch -> true
  | _, Concat (r1, r2) ->
    let rec split_and_match i =
      matches_regex r1 (String.sub word 0 i) &&
      matches_regex r2 (String.sub word i (String.length word - i)) in
    List.exists split_and_match (List.init (String.length word + 1) (fun x
      -> x))
```



```

| _, Sum (r1, r2) -> matches_regex r1 word || matches_regex r2 word
| "", Kleene _ -> true
| s, Kleene e when String.length s = 1 -> matches_regex e s
| _, Kleene r ->
    let rec split_and_match i =
        matches_regex r (String.sub word 0 i) && matches_regex regex (String
            .sub word i (String.length word - i)) in
        List.exists split_and_match (List.init (String.length word + 1) (fun x
            -> x))
| _, _ -> false;;

```

H2. Quelle est la complexité de cette fonction dans le pire des cas ?

**Solution :** La complexité est exponentielle à cause des appels multiples récursifs. Les automates procurent une solution de complexité linéaire en fonction de la longueur du mot.

## I Constructeurs intelligents d'expressions régulières

Afin d'accélérer la constructions d'expressions régulières, on considère les équivalences suivantes

$$\emptyset | e \equiv e | \emptyset \equiv e \quad (23)$$

$$e \cdot \varepsilon \equiv \varepsilon \cdot e \equiv e \quad (24)$$

$$e \cdot \emptyset \equiv \emptyset \cdot e \equiv \emptyset \quad (25)$$

$$\emptyset^* \equiv \varepsilon \quad (26)$$

$$\varepsilon^* \equiv \varepsilon \quad (27)$$

$$(e^*)^* \equiv e^* \quad (28)$$

La fonction suivante réalise une simplification à la racine sur une expression du type Union en suivant la règle donnée.

I1. Écrire une fonction de signature `su : regexp -> regexp` qui simplifie la construction d'une expression régulière pour l'union en utilisant les équivalences précédentes.

**Solution :**

```

let su expr =
    match expr with
    | Sum (e, EmptySet) | Sum (EmptySet, e) -> e
    | _ -> expr

```

I2. Écrire une fonction de signature `sc : regexp -> regexp` qui simplifie la construction d'une expression régulière pour la concaténation en utilisant les équivalences précédentes.

**Solution :**

```

let sc expr =
  match expr with
  | Concat (e, Epsilon) | Concat (Epsilon, e) -> e
  | Concat (e, EmptySet) | Concat (EmptySet, e) -> EmptySet
  | _ -> expr

```

- I3. Écrire une fonction de signature `se : regexp -> regexp` qui simplifie la construction d'une expression régulière pour la fermeture de Kleene en utilisant les équivalences précédentes.

**Solution :**

```

let se expr =
  match expr with
  | Kleene (EmptySet) | Kleene (Epsilon) -> Epsilon
  | Kleene (Kleene e) -> Kleene e
  | _ -> expr

```

- I4. Écrire une fonction `simplifie : regexp -> regexp` qui simplifie une expression régulière en utilisant les fonctions précédentes.

**Solution :**

```

let rec simplifie expr =
  match expr with
  | Sum(a, b) -> su (Sum (simplifie a, simplifie b))
  | Concat(a, b) -> sc (Concat (simplifie a, simplifie b))
  | Kleene(a) -> se (Kleene (simplifie a))
  | _ -> expr;;

let tosimpl = Kleene(Sum(Concat(Letter 'b', Epsilon), Concat(EmptySet, Letter 'a')));;
simplifie tosimpl;;

```

## J Jouer avec les expressions régulières --> HORS PROGRAMME

Lors d'une campagne de tests, on a collecté l'évolution de la position GPS d'un véhicule. Le fichier contient toutes les positions du test.

- J1. À l'aide d'une ligne de commande et en utilisant `grep`, isoler la latitude et la longitude dans un fichier. Chaque ligne contiendra une information comme suit :

```
5920.7009,N,01803.2938,E
```

**Solution :** `grep -oE "[[:digit:]]+[[:digit:]]+(S|N),[[:digit:]]+[[:digit:]]+(E|W)" gps.dat`