Automates

C. Charignon

Table des matières

I Langages et automates **3**

1 Exemple introductif **3**

2 Programmation d'un automate (fini déterministe complet) **4**

3 Vocabulaire sur les langages **5**

4 Définition formelle d'un automate (fini déterministe complet) **6**

5 Opérations sur les langages **7**
- 5.1 Concaténation . . . 7
- 5.2 Union . . . 7
- 5.3 Étoile de Kleene . . . 7
- 5.4 Langage régulier . . . 8

6 Automate incomplet **8**
- 6.1 Définition . . . 8
- 6.2 Complétion . . . 9

7 Automate émondé **9**
- 7.1 Définitions . . . 10
- 7.2 Émondage . . . 10
- 7.3 Programmation de l'émondage . . . 10

8 Automate non déterministe **10**
- 8.1 Définition . . . 10
- 8.2 Programmation . . . 11
- 8.3 Déterminisation . . . 11
- 8.4 Programmation de la déterminisation . . . 12
- 8.5 Application : automate reconnaissant les chaînes contenant un certain mot . . . 15

9 Complexité **15**
- 9.1 Automate déterministe . . . 15
- 9.2 Automate non déterministe . . . 15

10 Autres manipulations d'automates **15**
- 10.1 Reconnaître le complémentaire . . . 16
- 10.2 Reconnaître une union, intersection, ou différence . . . 16

1 Mots **1**

2 Langages **1**

3 Exemples d'automates déterministes et de langages rationnels **1**

4 Automates déterministes : exercices théoriques **2**

5 **Opérations sur les automates déterministes** **2**

6 **Automates non déterministes** **3**

7 **Utilisation d'un automate** **4**

II Algorithme de Berry Sethi **8**

1 **Expression régulière** **8**

2 **Langages locaux** **9**
2.1 Définition, exemples . . . 9
2.2 Automate associé à un langage local . . . 11
2.2.1 Définition . . . 11
2.2.2 Propriétés . . . 11
2.2.3 Programmation . . . 12
2.3 Opération sur les langages locaux . . . 13

3 **Expression rationnelle linéaire** **14**

4 **Algorithme de Berry-Sethi / automate de Glushkov** **14**
4.1 Linéarisation . . . 14
4.2 Retour à l'expression rationnelle de départ . . . 15
4.3 Programmation . . . 16

5 **Conséquences, et compléments** **17**

6 **Résumé sur les différents types de langages** **18**

1 **Langages locaux** **1**

2 **Algorithme de Glushkov** **1**

3 **Langages reconnaissables** **2**

Première partie
Langages et automates

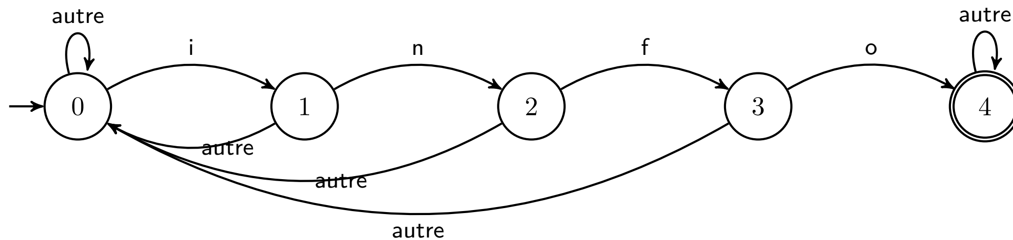
1 Exemple introductif

Écrivons une fonction pour déterminer si une chaîne de caractères contient le facteur `info`.

De manière générale, pour rechercher un facteur de longueur n dans une chaîne de longueur m, l'algorithme naïf vu en première année du tronc commun a une complexité en $O(nm)$. Ici, $n = 4$ ce qui n'est certes pas énorme, mais nous pouvons faire mieux : nous allons décrire un algorithme qui ne lit qu'une seule fois chacun des m caractères de la chaîne dans laquelle on recherche.

Voici l'idée : nous allons garder en mémoire où nous en sommes du mot `info`. Initialement, nous n'avons lu aucune des lettres de ce mot. Si à un moment nous lisons un `i`, nous garderons cette information en mémoire. Ensuite, si nous lisons juste après un `n`, nous garderons en mémoire que nous avons trouvé `in`, par contre si nous lisons autre chose nous revenons à l'état initial, à savoir que nous n'avons rien du mot `info`. En effet, un `i` suivi d'autre chose qu'un `n` ne peut pas servir à obtenir le mot voulu. On parcourt de la sorte tout le texte entré.
Remarque : Le point important qui autorise cette méthode simple sur cet exemple est que toutes les lettres du mot cherché sont distinctes. Ainsi, si j'ai lu `inf` en partant de la k-ième lettre de la chaîne de caractère cherchée, il est inutile de recommencer une recherche à partir de la $(k+1)$-ième : nous savons que c'est un `n`, elle ne peut donc être le début de `info`. Nous pouvons poursuivre la recherche en position $(k+3)$.

Maintenant, une méthode pratique pour retenir où nous en sommes de la lecture du mot cherché est d'utiliser le graphe orienté suivant :



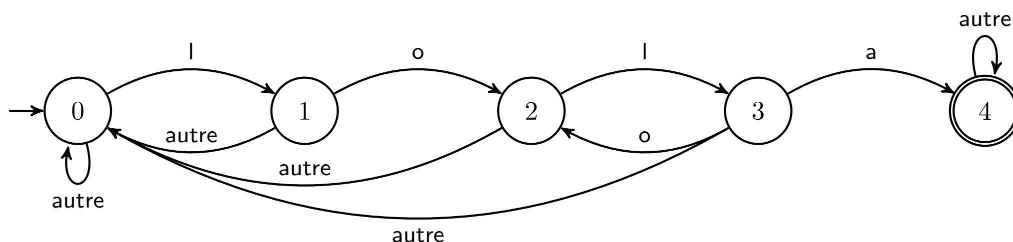
On utilise ce graphe ainsi : on part du sommet 0, et à chaque lettre lue, on suit l'arête correspondante. Si à un moment nous arrivons dans le sommet 4, c'est que le mot cherché a été trouvé.

Remarque : Pour être précis, il faut rajouter des transitions de 4 vers 4 : pour y rester jusqu'à la fin de la lecture.

Un tel graphe s'appelle un « automate », et ses sommets des « états ». La fonction qui à un état et un caractère associe l'état suivant s'appelle la « fonction de transition ». Il est très facile de programmer un tel automate, et nous obtenons bien un algorithme permettant de détecter si une chaîne de caractère contient le mot `info` en lisant une seule fois chaque caractère de la chaîne.

Exercice : Dessiner un automate reconnaissant l'ensemble des mots dont « info » est un sous-mot, c'est-à-dire contenant les lettres i, n, f et o dans le bon ordre mais pas forcément à la suite.

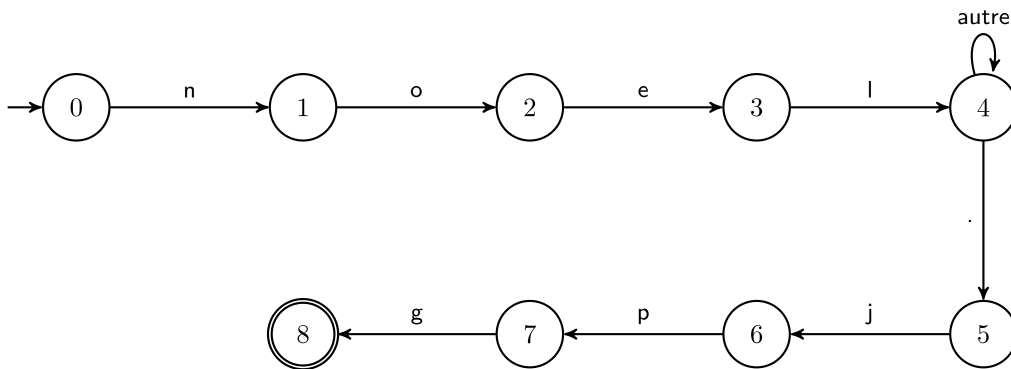
Maintenant, voyons comment adapter la méthode pour détecter le mot `lola`. Il y a une subtilité, due à la présence du deuxième *l*. Après le deuxième `l`, nous espérons un `a`, cependant si nous trouvons un `o`, nous ne devons pas revenir au départ mais au deuxième état. L'automate obtenu est :



Remarque : Nous allons assez rapidement nous débarrasser de toutes les flèches « autres » pour simplifier les graphes.

N.B. Nous verrons d'un point de vue théorique à la fin du chapitre que la présence de lettres qui se répètent est source de difficulté dans la création de l'automate. On dit que l'expression à rechercher n'est pas *linéaire* lorsqu'elle contient plusieurs fois la même lettre.

L'utilité des automates ne se limite pas à rechercher un simple mot dans un texte. Par exemple, imaginons que nous cherchions sur le disque dur une image au format `.jpg` dont le nom contient `noël`. Nous pouvons alors utiliser l'automate :



où il faut encore rajouter des transitions étiquetées par « autre » des états 0, 1, 2, 3 vers 0 et 5, 6, 7 vers 4.

2 Programmation d'un automate (fini déterministe complet)

Un automate est essentiellement un graphe dont les arêtes sont étiquetées par des lettres. Les sommets du graphe s'appellent des « états », et les arêtes des « transitions ».

Un état particulier est distingué comme état « initial », et un ou plusieurs sommets ont été marqués comme sommets « finals » ou « acceptants ».

À chaque lettre lue, nous allons parcourir toutes les arêtes issues du sommet actuel pour voir si l'une est étiquetée par cette lettre. La représentation la plus pratique sera donc par tableau de listes d'adjacence.

En outre, nous devons retenir l'état initial et les états finals [1]. Finalement, nous allons créer un type enregistrement à trois champs :

```
type automate={initial:int ; acceptant:int list; transition: (char*int) list vect};;
```

Remarque : Il s'agit donc d'enregistrer pour chaque sommet un dictionnaire associant à chaque lettre une arête. Ici nous avons choisi une liste d'association car il y a peu de lettres et car c'est l'option que nous avions utilisée pour les graphes. Les options ABR et tables de hachage sont toujours possibles. On pourrait même utiliser des fonctions : `transition : char -> int -> int`.

```
(* Exemple : automate reconnaissant les mots contenant "lola" *)

let list_of_string m=
    let l=String.length m in

    let rec litAPartirDe i=
        if i=l then []
        else m.[i]::litAPartirDe (i+1)

    in litAPartirDe 0
;;

let alphabet=list_of_string "abcdefghijklmnopqrstuvwxyz";;

let toutVers i= map (fun x-> (x,i)) alphabet;;
```

1. et pas « finaux »

```
let exLola=
{
  initial=0;
  acceptant=[4];
  transitions= [|  ('l',1)::toutVers 0;
    ('o',2)::toutVers 0;
    ('l',3)::toutVers 0;
    ('a',4)::('o',2)::toutVers 0;
    toutVers 4
  |]
}
;;
```

On programme maintenant la reconnaissance d'un motif par un automate. La fonction qui à un état et une lettre associe l'état suivant est traditionnellement notée δ. Celle qui à un état et un mot associe l'état atteint après lecture de toutes les lettres du mot est notée δ^*. Les définitions formelles sont dans la partie 3.

```
(* La fonction de transition *)
let delta etat lettre transition =
    (* Renvoie l'état dans lequel arrive l'automate en partant de etat et en lisant lettre.*)
    List.assoc lettre transition.(etat);;


(* La fonction de transition étendue *)
let deltaEtoile etat mot transition =
    (* Renvoie l'état dans lequel arrive l'automate en partant de etat et en lisant le mot
       mot*)
    let etatActuel = ref etat in
    for i=0 to string_length mot -1 do
        etatActuel := delta !etatActuel mot.[i] transition
    done;
    !etatActuel
;;

let accepte mot a=
    List.mem (deltaEtoile a.initial mot a.transitions)  a.acceptant
;;

accepte "sdbvlolage" exLola;;
accepte "sdbvloloage" exLola;;
```

3 Vocabulaire sur les langages

À présent on passe à la théorie.

On fixe dans toute la suite un ensemble fini Σ qu'on appellera l'alphabet.

- Chaque élément de Σ sera appelé une lettre.
- Une suite finie de lettres s'appelle un mot.
- L'ensemble des mots se note Σ^*. Le mot vide est noté ϵ.
  Remarque : On note parfois Σ^+ l'ensemble des mots non vides.
- La concaténation sera notée $\cdot$, c'est une loi de composition interne sur Σ^*.
- Pour tout $u \in \Sigma^*$, on note $|u|$ le nombre de lettres de u.
- Pour tout $n \in \mathbb{N}$, Σ^n est l'ensemble des mots de n lettres. Donc $\Sigma^0 = \{\epsilon\}$, et $\Sigma^* = \bigcup_{n\in\mathbb{N}} \Sigma^n$.
- Nous identifions les lettres avec les mots d'une seule lettre, c'est-à-dire que nous convenons que $\Sigma = \Sigma^1$.
- Un ensemble de mots est appelé un « langage ».

Proposition 3.1.
- $(\Sigma^*, \cdot)$ *est un monoïde. (On dit parfois « semi-groupe ».)*

- *Son élément neutre est ϵ.*
- *Pour tout $(u,v) \in (\Sigma^*)^2$, $|uv| = |u| + |v|$.*

Remarque : $(\Sigma^*, \cdot)$ n'est pas commutatif dès que $\text{Card}(\Sigma) \geqslant 2$.

Le but de ce chapitre est de mettre en place des méthodes efficaces pour déterminer si un mot est dans un certain langage. On retrouve ce genre d'algorithmes dans la plupart des langages de programmation (bibliothèque `Str`de Ocaml, `re` de Python, programme `grep` de linux...).

Dans la pratique, le plus souvent, l'alphabet considéré est l'ensemble des caractères disponibles dans l'encodage utilisé, cependant dans les exercices nous utiliserons souvent pour simplifier des alphabets réduits à quelques lettres.

Exemples :

1. Reconnaître un numéro de téléphone. Le langage des numéros de téléphone est l'ensemble des mots formés de 10 chiffres. On peut le noter ainsi : $\{0,1,2,3,4,5,6,7,8,9\}^{10}$.
2. Reconnaître une adresse électronique française. Le langage des adresses électroniques françaises est l'ensemble des mots contenant un `@`, et se terminant par `.fr` ou `.com`. Nous verrons plus loin que ce langage peut être noté $\Sigma^*@\Sigma^*(.com|.fr)$
3. Rechercher un document des impôts : on pourra rechercher sur son disque dur l'ensemble des fichiers dont le nom contient `impot` ou `impôt` et dont l'extension soit `.pdf`. On notera ce langage $\Sigma^*\cdot$ `imp(o|ô)t` $\cdot\Sigma^*$ `.pdf`
4. L'ensemble des expressions bien parenthésées.
5. L'ensemble des mots utilisant uniquement des caractères ASCII (en particulier sans accent).

Définition 3.2. *Soient u et v deux mots.*

- *On dit que u est préfixe de v lorsqu'il existe $w \in \Sigma^*$ tel que $v = uw$.*
- *On dit que u est suffixe de v lorsqu'il existe $w \in \Sigma^*$ tel que $v = wu$.*
- *On dit que u est facteur de v lorsqu'il existe $w, z \in \Sigma^*$ tel que $v = wuz$.*
- *Enfin, notons $n = |u|$ et $u_0, \dots u_{n-1}$ les lettres de u, on dit que u est un sous-mot de v lorsqu'il existe $(w_0, \dots, w_n) \in (\Sigma^*)^{n+1}$ tel que $u = w_0 u_0 w_1 u_1 \dots w_{n-1} u_n w_n$.*

cf exercice : **??**

4 Définition formelle d'un automate (fini déterministe complet)

Définition 4.1. *Un automate (complet fini déterministe) sur l'alphabet Σ est un quadruplet formé de :*

- *un ensemble fini Q dont les éléments sont appelés les états ;*
- *un état $q_0 \in Q$ appelé l'état initial ;*
- *un ensemble d'états $F \subset Q$ dont les éléments sont appelés les états finals, ou acceptants ;*
- *une fonction δ de type $Q \times \Sigma \to Q$ appelée la fonction de transition.*

Remarque : Dans le type Caml utilisé ci-dessus, on n'enregistre pas directement la fonction δ, mais son graphe, et on a reprogrammé la fonction δ.

cf exercice : 15

Définition 4.2. *(Fonction de transition étendue δ^*)*
Soit $\Sigma = (Q, q_0, F, \delta)$ un automate sur Σ. On définit récursivement sa fonction de transition étendue, de type $\mathbb{Q} \times \sigma^* \to Q$, qu'on note δ^* par :

- $\forall q \in Q$, $\delta^*(q, \epsilon) = q$
- $\forall (q,m,x) \in Q \times \Sigma^* \times \Sigma$, $\delta^*(q, mx) = \delta\left(\delta^*(q,m), x\right)$.

Remarque : Soit $(q,x) \in Q \times \Sigma$. Alors :

$$\delta^*(q,x) = \delta^*(q, \epsilon x) = \delta\left(\delta^*(q,\epsilon), x\right) = \delta(q,x).$$

Ainsi, la fonction δ^* prolonge la fonction δ. Elle est définie sur les mots alors que δ n'est définie que sur les lettres.
Voici la propriété élémentaire de δ^* :

Proposition 4.3. *Pour tout* $(m, n) \in (\Sigma^*)^2$*, pour tout* $q \in Q$*,*

$$\delta^*(q, nm) = \delta^*\left(\delta^*(q, n), m\right)$$

Démonstration :

□

Définition 4.4. *Soit* $\mathcal{A} = (Q, q_0, F, \delta)$ *un automate sur* Σ*.*

- *Pour tout* $m \in \Sigma^*$*, on dit que* m *est reconnu par* $\mathcal{A}$ *lorsque* $\delta^*(q_0, m) \in F$*.*
- *L'ensemble des mots reconnus par* $\mathcal{A}$ *s'appelle le langage reconnu par* $\mathcal{A}$*.*
- *Un langage est dit reconnaissable lorsqu'il existe un automate le reconnaissant.*

5 Opérations sur les langages

5.1 Concaténation

On définit de la manière naturelle la concaténation de deux langages :

Définition 5.1. *Soit* $(L, L') \in \mathcal{P}(\Sigma^*)^2$*, on pose alors :*

$$L \cdot L' = \{m \cdot m' \; ; \; m \in L, m' \in L'\}.$$

Cette opération est associative, et admet pour neutre $\{\varepsilon\}$.

5.2 Union

L'opération de réunion $\cup$ est déjà connue. Elle est associative, commutative et admet pour neutre l'ensemble vide $\emptyset$.
On emploie parfois la notation $+$ au lieu de $\cup$. Ainsi $a + b$ désigne le langage $\{a, b\}$.

La propriété ci-dessous, outre le fait que $\emptyset$ ressemble à 0, justifie cette notation :

Proposition 5.2. *La concaténation* $\cdot$ *est distributive par rapport à* $\cup$*.*

Démonstration :

⸾ La démonstration vous rappellera les démonstrations sur les ensembles de début de MPSI.

Soient L_1, L_2 et L_3 trois langages. Pouvons que $(L_1 \cup L_2) \cdot L_3 = L_1 \cdot L_3 \cup L_2 \cdot L_3$.

- « $\subset$ » : Soit $m \in (L_1 \cup L_2) \cdot L_3$. Il existe donc $u \in L_1 \cup L_2$ et $v \in L_3$ tels que $m = u \cdot v$.
  Si $u \in L_1$ alors $uv \in L_1 \cdot L_3$.
  Si $u \in L_2$, alors $uv \in L_2 \cdot L_3$.
  Dans tous les cas, $uv \in L_1 \cdot L_3 \cup L_2 \cdot L_3$.
- « $\supset$ » : Soit $m \in L_1 \cdot L_3 \cup L_2 \cdot L_3$. Traitons le cas où $m \in L_1 \cdot L_3$, l'autre cas est similaire. Il existe $u \in L_1$ et $v \in L_3$ tels que $m = uv$. Comme $u \in L_1 \cup L_2$, on peut bien dire que $uv \in (L_1 \cup L_2) \cdot L_3$.

5.3 Étoile de Kleene

Nous avons déjà vu la notation $*$ pour un ensemble de lettres. Pour une seule lettre, on s'autorise à noter a^* au lieu de $\{a\}^*$.
On étend la définition de l'étoile aux langage : pour tout langage L, on pose :

$$L^* = \bigcup_{n \in \mathbb{N}} L^n.$$

Ainsi, L^* est l'ensemble des mots pouvant être obtenus en concaténant un nombre quelconques d'éléments de L.

Proposition 5.3. *(Propriétés de l'étoile de Kleene)*

1. *(l'étoile est croissante) Pour tous langages* L *et* M *tels que* $L \subset M$*,* $L^* \subset M^*$*.*
2. *(l'étoile est idempotente) Pour tout langage* L*,* $(L^*)^* = L^*$*.*

Démonstration :

- **Croissance :** Soient L et M deux langages tels que $L \subset M$. Soit $m \in L^*$. Il existe $n \in \mathbb{N}$ et $(l_1, ..., l_n) \in L^n$ tels que $M = l_1 \cdot ... \cdot l_n$. Mais $\forall i \in [\![1, n]\!]$, $l_i \in M$. Donc $m \in M$.

- **Idempotence :**
  - L'inclusion $L^* \subset (L^*)^*$ est évidente : en fait pour tout langage K, $K \subset K^*$.
  - Soit $m \in (L^*)^*$. Le mot m est concaténation de mots de L^* qui sont eux-même concaténation de mots de L. Donc m est concaténation de mots de L.

□

À titre d'exercice, démontrons la propriété suivante :

Proposition 5.4. *Soient L_1 et L_2 deux langages sur Σ.*

1. $(L_1 + L_2)^* = (L_1^* L_2^*)^*$
2. $(L_1^* L_2^*)^* = (L_2^* L_1^*)^*$.

5.4 Langage régulier

On définit un ensemble de langages qui sera de grande importance dans la suite du cours.

Définition 5.5. *(Langage régulier)*
Un langage pouvant être défini uniquement à l'aide d'un nombre fini

- *de $\emptyset$;*
- *de lettres et de mots vides ;*
- *de réunions ;*
- *de concaténations ;*
- *d'étoiles,*

s'appelle un langage régulier.

Pour ce qui est des priorités, on conviendra que l'étoile est l'opération la plus prioritaire, suivie de la concaténation.

N.B. Dans cette définition, on n'autorise pas l'intersection, ni le complémentaire. La raison est que ces opérations ne sont pas gérées par l'algorithme de Berry-Sehti, que nous verrons dans la deuxième partie de ce cours.

Une formule permettant de décrire un langage à l'aide de réunions, concaténations, étoiles, et des lettres s'appelle une « expression régulière ». La définition précise et l'étude des expressions régulières sera menée plus loin. Le but final du chapitre est d'écrire un programme prenant une expression régulière et renvoyant un automate reconnaissant le langage associé.

Remarque : Test d'expression régulière, par exemple sur https://regex101.com/. Avec la syntaxe de linux, qui n'est pas tout à fait la syntaxe mathématique.

6 Automate incomplet

6.1 Définition

Un automate incomplet est un quadruplet (Q, q_0, F, δ) vérifiant les mêmes conditions qu'un automate sauf que δ n'est pas obligatoirement définie sur $Q \times \Sigma$ entier. Un couple $(q, x) \in Q \times \Sigma$ hors du domaine de définition de δ s'appelle un « blocage ».

On définit alors δ^* comme dans le cas d'un automate complet ; la fonction obtenue est définie sur une partie de $\mathbb{Q} \times \Sigma^*$. Et un couple $(q, m) \in Q \times \Sigma^*$ hors du domaine de définition de δ^* s'appelle encore un blocage.

Fixons $\mathcal{A} = (Q, q_0, F, \delta)$ un automate incomplet. Soit $m \in \Sigma^*$. Nous dirons que m est accepté par $\mathcal{A}$ lorsque (q_0, m) n'est pas un blocage et $\delta^*(q_0, m) \in F$.

Exemple :

- Reprenons les exemples du début du cours et supprimons les transitions étiquetées par « autre ». Que reconnaissent-ils ?
- Dessinons un automate pour reconnaître :
  - les mots commençant par « info » ;
  - les mots terminant par « info ».

Programmons l'utilisation d'un automate incomplet. Nous pouvons garder le même type, simplement il y aura éventuellement moins que Card(Σ) éléments dans chaque liste de voisins.

On choisit de renvoyer une erreur lors d'un blocage. Pour l'occasion, utilisons une erreur plus propre qu'un simple `failwith`. En fait, il existe un type pour les erreurs : c'est le type `Exception`. Et on peut créer soi-même des erreurs :

```
exception Blocage;;
```

Par la suite, pour renvoyer l'exception `Blocage` que nous venons de créer, il suffira de taper `raise blocage`.
Et les différentes exceptions peuvent être filtrées (« matchées ») dans le « `with` » de la construction `try ... with...`.
La fonction `delta` devient :

```
let delta etat lettre transition=
    try
        assoc lettre transition.(etat)
    with
        | Not_found -> raise Blocage;;
```

Et la fonction finale donnera :

```
let accepte mot a=
    try
        mem (deltaEtoile a.initial mot a.transitions)  a.acceptant
    with
        |Blocage   -> false
;;
```

Il est inutile de modifier la fonction `deltaEtoile`. En effet, en cas de blocage, la fonction `delta` lèvera l'exception `Blocage`. Cette exception sera fidèlement transmise par `deltaEtoile`, et sera ensuite rattrapée par `accepte`.

6.2 Complétion

Le théorème suivant prouve que les automates complets ou incomplets peuvent reconnaître les mêmes langages. Un automate incomplet présente l'avantage d'occuper moins de mémoire et d'être plus facile à dessiner et à concevoir.

Théorème 6.1. *Tout langage reconnu par un automate fini déterministe non complet l'est aussi par un automate fini déterministe complet.*

Démonstration : Soit $\mathcal{A}$ un automate incomplet, soit (Q, i, F, δ) ses constituants. Soit $\mathcal{A}'$ obtenu ainsi à partir de $\mathcal{A}$ de la manière suivante :

- On rajoute un état p (pour « puits »).
- Pour tout blocage (q, x) de $\mathcal{A}$, on rajoute une transition de q vers p étiquetée par x dans $\mathcal{A}'$.
- Pour tout $x \in \Sigma$, on ajoute la transition (p, p, x) dans $\mathcal{A}'$.
- On garde i et F.

Ainsi, p est un puits : une fois qu'on n'y arrive on ne peut plus en sortir. Comme ce n'est pas un état acceptant, un mot ne sera jamais accepté si l'automate arrive à un moment donné sur l'état p.

Vérifions que $\mathcal{A}'$ convient.

- $\mathcal{A}'$ est complet car pour tout $x \in \Sigma$, il y a une transition étiquetée par x depuis chaque $q \in Q$ (soit il y en avait une dans $\mathcal{A}$, soit c'était un blocage et alors il y a $q \underset{x}{\curvearrowright} p$ dans $\mathcal{A}'$), et il y a une transition étiquetée par x depuis p (qui mèn à p).
-
  - Soit m un mot reconnu par $\mathcal{A}$ et c le chemin étiqueté par m dans $\mathcal{A}$ qui part de i et arrive dans un état final. Ce même chemin existe aussi dans $\mathcal{A}'$, donc m est aussi seconnu dans $\mathcal{A}'$.
  - Soit m un mot reconnu par $\mathcal{A}'$ et c le chemin correspondant. Le chemin c ne peut passer par le puits p sans quoi il finirait à p qui n'est pas final. Donc il est formé uniquement d'états et de trasitions qui sont dans $\mathcal{A}$. Donc m est reconnu par $\mathcal{A}$.

Comme souvent, de la démonstration on peut déduire le programme. Ici, on déduit une fonction prenant un automate complet et renvoyant un automate complet qui reconnaît le même langage.
Cepemdant, cette fonction ne sera utile que dans des cas particuliers (**cf exercice :** 18 par exemple). En effet en général on préfère conserver un automate incomplet qui

- nécessite moins de mémoire ;
- est plus rapide car d'une part chaque liste d'association est plus courte, et d'autre part la lecture peut être dans certains cas interrompue avant la fin du mot.

7 Automate émondé

La démarche ici est inverse de celle du paragraphe précédent : nous allons supprimer autant d'état que possible, et obtenir un automate en général incomplet.

7.1 Définitions

Définition 7.1. *Soit $\mathcal{A} = (Q, q_0, F, \delta)$ un AFD.*

- *Un état $q \in Q$ est dit accessible lorsqu'il existe $m \in \Sigma^*$ tel que $\delta^*(m, q_0) = q$.*
- *Un état $q \in Q$ est dit co-accessible lorsqu'il existe $m \in \Sigma^*$ et $q_f \in F$ tel que $\delta^*(q, m) = q_f$.*
- *Un état accessible et co-accessible sera dit « utile ».*
- *L'automate $\mathcal{A}$ est dit émondé si tous ses états sont accessibles et co-accessibles.*

7.2 Émondage

Si un automate $\mathcal{A}$ n'est pas émondé, on peut supprimer tous ses états non accessibles ou non co-accessibles pour obtenir un automate émondé qui reconnaît le même langage, tout en étant plus léger et donc plus rapide à utiliser.

Proposition 7.2. *Soit (Q, q_0, F, δ) un AFD qu'on notera $\mathcal{A}$. On définit :*

- *Q l'ensemble des états utiles de $\mathcal{A}$;*
- *$F = Q \cap F$*
- *δ la restriction de δ à F.*

Soit $\mathcal{A} = \left(Q, q_0, F, \delta\right)$. Alors $\mathcal{A}$ est un automate émondé qui reconnaît le même langage que $\mathcal{A}$.

L'automate $\mathcal{A}$ sera appelé l'automate émondé de $\mathcal{A}$. *Démonstration :*

- Prouvons que $\mathcal{A}$ est émondé. Seule subtilité : ses états sont par définition accessibles et coaccessibles dans $\mathcal{A}$. Il faut vérifire qu'ils le sont encore dans $\mathcal{A}'$.

  Soit $q \in Q$. Il est accessible et coaccessible dans $\mathcal{A}$. Il existe donc un chemin γ de q_0 vers un état $q \in F$. L'existence de γ prouve alors que chacun de ses sommets est accessible et co-accessible dans $\mathcal{A}$, de sorte que γ est inclu dans Q. Et finalement, q est accessible et co-accessible dans $\mathcal{A}$.
- Prouvons que $\mathcal{A}$ reconnaît le même langage que $\mathcal{A}$.
  - Soit $m \in \mathcal{L}\left(\mathcal{A}\right)$. Soit γ un chemin étiqueté par m dans $\mathcal{A}$ qui va de q_0 vers un état $q \in F$. C'est en particulier un chemin dans $\mathcal{A}$ de q_0 vers un état $q \in F$, étiqueté par m, donc $m \in \mathcal{L}\left(\mathcal{A}\right)$.
  - Soit $m \in \mathcal{L}\left(\mathcal{A}\right)$. Soit γ un chemin dans $\mathcal{A}$ de q_0 vers un état $q \in F$, étiqueté par m. Comme dans la première partie de la preuve, l'existence de γ prouve que chacun de ses états est accessible et co-accessible et donc figure dans Q. Donc γ est aussi un chemin acceptant dans $\mathcal{A}$ et $m \in \mathcal{L}\left(\mathcal{A}\right)$.

□

7.3 Programmation de l'émondage

Pour commencer on récupère l'ensemble des états accessibles. Il suffit de se souvenir du cours sur les parcours de graphe... N'importe quel parcours fera l'affaire ici.

Ensuite pour les états co-accessibles, le plus simple est de renverser toutes les arêtes. Les états co-accessibles de $\mathcal{A}$ sont, dans l'automate miroir ainsi obtenu, les états accessibles à partir d'un élément de F.

Une fois obtenus les états accessibles et co-accessibles, il n'y a plus qu'à calculer l'intersection de ces deux ensembles, et à supprimer dans l'automate initial toutes les trasitions qui partent ou qui arrivent à un état inutile. Pour simplifier, on ne supprime pas les états eux-mêmes (cela nécessiterait de renuméroter les états restant...)

cf exercice : 19

8 Automate non déterministe

8.1 Définition

Dans un automate non déterministe, on peut être dans plusieurs états à la fois. Il peut y avoir plusieurs états initiaux, et depuis chaque état, une lettre peut mener vers plusieurs autres états.

Définition 8.1. *Un automate non déterministe est un quadruplet formé de :*

- *Un ensemble Q dont les éléments sont appelés les états ;*
- *Une partie I de Q dont les élément sont appelés les états initiaux ;*
- *Une partie F de Q dont les élément sont appelés les états finals, ou acceptant ;*
- *Une fonction $\delta : \Sigma \times Q \to \mathcal{P}(Q)$.*

Pour programmer la lecture d'un mot par un automate non déterministe, on va donc maintenir une liste d'états actuels.
La définition formelle de δ^* est alors la suivante :

Définition 8.2. *Soit (Q, I, F, δ) un automate non déterministe sur Σ. On définit sa fonction de transition étendue δ^* ainsi : c'est la fonction de $\mathcal{P}(Q) \times \Sigma^*$ dans $\mathcal{P}(Q)$ telle que*

- *Pour tout $X \in \mathcal{P}(Q)$, $\delta^*(X, \epsilon) = X$.*
- *Pour tout $X \in \mathcal{P}(Q)$, pour tout $m \in \sigma^*$ et tout $x \in \Sigma$, $\delta^*(X, mx) = \bigcup_{q \in \delta^*(X,m)} \delta(q, x)$.*

Définition 8.3. *Soit $a = (Q, I, F, \delta)$ un automate non déterministe sur Σ. Soit $m \in \Sigma^*$. On dit que m est reconnu par a lorsque il existe $q_0 \in I$ tel que $\delta^*(q_0, m) \cap F \neq \emptyset$.*

Proposition 8.4. *Un mot m est reconnu par a si et seulement si il existe un chemin depuis un des états initiaux vers un des états finals étiqueté par les lettres de m.*

Un tel chemin pourra être appelé un chemin « acceptant » pour m.

Exemple : Soit $\Sigma = \{a, b\}$, $n \in \mathbb{N}^*$, et $L = \Sigma^* a \Sigma^{n-1}$ le langage des mots ayant un a en position n à partir de la fin.

- Soit $\mathcal{A}$ un automate complet déterministe reconnaissant L. Soient u et v deux mots distincts de longueur $n-1$. Nous allons prouver que $\delta^*(q_0, u) \neq \delta^*(q_0, v)$.
  Notons w le plus grand suffixe commun de u et v. Quitte à échanger u et v, supposons que la première lettre en partant de la fin qui diffère entre u et v vaut a pour u et b pour v, donc il existe u', v' tels que $u = u'aw$ et $v = v'bw$. Soit $k = n - 1 - |w|$, on a alors $ua^k \in L$ et $va^k \notin L$. Ceci entraîne que $\delta^*(q_0, u) \neq \delta^*(q_0, v)$ (sans quoi on aurait $\delta^*(q_0, ua^k) \neq \delta^*(q_0, va^k)$).
  Ainsi le nombre d'états de $\mathcal{A}$ est supérieur au nombre de mots de longueur $n-1$ (car la fonction $u \mapsto \delta^*(q_0, u)$ est injective), donc à 2^{n-1}.
- Par contre, un automate non déterministe à $n+1$ états suffit à reconnaître L.

8.2 Programmation

Voici une bonne occasion d'utiliser `List.map` et `List.fold`.

- Le type Caml pour un AFND.
- Union sans doublon.
- La fonction δ.
- La fonction δ^*.
- Il sera pratique pour la suite de programmer la fonction suivante :

$$\delta_d^* : \begin{array}{rcl} \mathcal{P}(Q) & \to & \mathcal{P}(Q) \\ X & \mapsto & \bigcup_{q \in X} \delta^*(q, m) \end{array}$$

- Fonction finale pour tester si un mot est reconnu par un AFND.

8.3 Déterminisation

On voit que les automates non déterministes peuvent être plus simples, en terme de nombre d'états. En revanche leur utilisation est plus lente car nous manipulons sans arrêt des listes d'états là où pour un automate déterministe un seul état suffit.

Ainsi, un automate non déterministe sera souvent plus simple à concevoir, à décrire, et à étudier, tandis que pour l'implémentation sur ordinateur, un automate déterministe sera plus efficace. D'où l'intérêt du théorème suivant, qui permet de déterminiser un automate :

Théorème 8.5. *Tout langage reconnu par un automate déterministe peut aussi l'être par un automate non déterministe.*

Démonstration : Soit L un langage reconnu par un automate non déterministe $\mathcal{A} = (Q, I, F, \delta)$. On construit un automate déterministe $\mathcal{A}_d$ ainsi :

- On pose comme ensemble des états $Q_d = \mathcal{P}(Q)$. *Remarque :* $\mathcal{A}_d$ sera appelé l'automate « des parties » de $\mathcal{A}$.
- On définit la fonction de transition δ_d ainsi : $\forall x \in \Sigma$, $\forall E \in \mathcal{P}(Q)$, $\delta_d(x, E) = \bigcup_{q \in E} \delta(x, q)$.
- On prend comme état initial I.
- On prend comme états finals tous les états contenant un élément de F : on pose $F_d = \left\{ E \in \mathcal{P}(Q) \mid E \cap F \neq \emptyset \right\}$.

Il reste à vérifier que $\left(\mathcal{P}(Q), I, F_d, \delta_d\right)$ reconnaît le même langage que $\mathcal{A}$.
On prouve par récurrence que pour tout $X \in \mathcal{P}(Q)$, pour tout $m \in \Sigma^*$,

$$\delta_d^*(X, m) = \bigcup_{q \in X} \delta^*(q, m).$$

Remarque : En terme de chemins : notons $Y = \delta_d^*(X, m)$. Alors la propriété que nous voulons prouver formellement est que Y est l'ensemble des états q de a tels qu'il existe un chemin étiqueté par m partant d'un état de X et aboutissant à q, ce qu'on pourrait écrire ainsi :

$$\delta_d^*(X, m) = \left\{ q \in Q \mid \exists q_0 \in X \text{ tq } q_0 \underset{m}{\curvearrowright} q \right\}.$$

Fixons $X \in \mathcal{P}(Q)$ et notons pour tout $k \in \mathbb{N}$, $P(k)$: « pour tout $m \in \Sigma^k$, $\delta_d^*(X, m) = \bigcup_{q \in X} \delta^*(q, m)$ ».

- **Initialisation :** Soit $m \in \Sigma^0$, autrement dit $m = \epsilon$. On a
$$\delta_d^*(X, \epsilon) = X = \bigcup_{q \in X} \{q\} = \bigcup_{q \in X} \delta^*(q, \epsilon).$$
d'où $P(0)$.
- **Hérédité :** Soit $k \in \mathbb{N}$. Supposons $\forall i \in [\![0, k]\!]$, $P(i)$. Soit $m \in \Sigma^{k+1}$. Notons x sa dernière lettre et m' le mot formé de ses k premières lettres. Ainsi :
$$\begin{aligned}
\delta_d^*(X, m) &= \delta_d^*(X, m'x) && \text{(Propriété d'un automate déterministe)} \\
&= \delta_d\left(\delta_d^*(X, m'), x)\right) && \text{(Par P(k))} \\
&= \delta_d\left(\bigcup_{q \in X} \delta^*(q, m'), x\right) && \text{(Définition de } \delta_d\text{)} \\
&= \bigcup_{q \in X} \bigcup_{r \in \delta^*(q, m'} \delta(r, x) && \text{(Définition de } \delta^*\text{)} \\
&= \bigcup_{q \in X} \delta^*(q, m'x) \\
&= \bigcup_{q \in X} \delta^*(q, m)
\end{aligned}$$
D'où $P(k+1)$.

Par récurrence, pour tout $k \in \mathbb{N}$, $P(k)$, d'où la propriété.

On termine alors facilement la preuve : Soit $m \in \Sigma^*$, on a les équivalences :
$$\begin{aligned}
m \text{ reconnu par } a_d &\Leftrightarrow \delta_q(I, m) \in F_d \\
&\Leftrightarrow \delta_q(I, m) \cap F \neq \emptyset \\
&\Leftrightarrow \bigcup_{q \in I} \delta^*(q, m) \cap F \neq \emptyset \\
&\Leftrightarrow m \text{ est reconnu par } a.
\end{aligned}$$

8.4 Programmation de la déterminisation

Passons à la programmation de la déterminisation. Nous allons donc écrire une fonction prenant en entrée un automate non déterministe et renvoyant un automate déterministe reconnaissant le même langage.

La difficulté technique, au vu de l'implémentation que nous avons choisie pour les automate, est que les états doivent être des entiers. Or dans l'automate déterminisé, il s'agit d'ensembles.

Cette difficulté ne se serait pas présentée si nous avions utilisé un dictionnaire (`('a*char) 'a dictionnaire`) pour enregistrer les transitions.

Pour la résoudre, nous devons numéroter chaque état du nouvel automate. Le but est d'obtenir deux fonction `etats_of_int` et `int_of_etat`, réciproques l'une de l'autre, qui attribuent un entier à chaque état du nouvel automate.

Pour commencer, programmons la fonction δ_d :

```
1 let deltad qd x transitions=
2    (* La fonction deltad de Ad *)
3    (* Concrètement, qd est un ensemble d'états de l'automate initial (type int list), et ceci
       ↪ renvoie l'ensemble des états accessibles depuis un des éléments de qd en lisant la
       ↪ lettre x.*)
4   List.flatten (List.map (fun q-> deltaND q x transitions) qd);;
```

Notons que la réunion présente dans la définition de δ_d se traduit immédiatement par `List.flatten`.

Cependant, nous allons être confrontés à un autre problème : comment savoir si deux états de $\mathcal{A}_d$ sont égaux ? Mathématiquement, il s'agit d'ensembles, mais informatiquement nous utilisons des listes. Deux listes contenant les mêmes éléments, mais dans un ordre différent, devront être considérées comme égales. En outre, il faudra ne pas prendre en compte les éventuels doublons.

Pour résoudre ceci, convenons que nous n'utiliserons que des listes triées et sans doublons (on pourrait dire « triées dans l'ordre strictement croissant »). Ainsi deux listes seront égales si et seulement si les ensembles sous-jacents sont égaux.

Je vais reprendre le programme du tri-fusion en modifiant la fonction `fusion` pour qu'elle supprime les doublons. Ainsi, nous pourrons non seulement utiliser la fonction de tri pour trier nos listes, mais aussi la fonction `fusion` pour rassembler deux listes.

```ocaml
let rec fusion_stricte l m=
  (* fusion de deux listes triées et sans doublon, pour obtenir une liste triée sans
     doublon.*)
   match l,m with
   |[], _ -> m
   |_,[] -> l
   |a::q, b ::r when a<b -> a:: (fusion_stricte q m)
   |a::q, b ::r when a=b ->  (fusion_stricte q m)
   |a::q, b ::r  -> b:: (fusion_stricte l r)
;;

let rec partition = function
  |[] -> [],[]
  |[t]-> [t], []
  |s::t::q -> let l1, l2 = partition q in s::l1, t::l2
;;


let rec tri l=
   match l with
   |[]->[]
   |[t]->l
   |_ -> let l1, l2= partition l in fusion_stricte (tri l1) (tri l2);;
```

La fonction suivante permet alors de remplacer `List.flatten` pour maintenir des listes triées et sans doublons :

```ocaml
let union_tri l=
   (* entrée : l une liste de listes triées.
      sortie : la concaténation de ces listes, le résultat étant trié et sans doublon (càd
         strictement trié). *)
  List.fold_left  fusion_stricte [] l;;
```

Et voici enfin la fonction δ_d corrigée. Notez l'usage à la fois de `tri` pour trier les listes renvoyées par la fonction de transition initiale, et de `union_tri` pour maintenir le caractère strictement trié du résultat.

```ocaml
let delta_d a lq x =
    union_tri
    (List.map (fun q-> tri (deltaND a q x))
               lq
    )
;;
```

Tant qu'on y est, programmons également l'intersection pour les listes strictement croissantes :

```ocaml
let rec intersection l1 l2=
  match l1, l2 with
  |[],_ -> []
  |_, [] -> []
  | t::q, r::p when t< r -> intersection q l2
  | t::q, r::p when t> r -> intersection l1 p
  | t::q, r::p -> t:: intersection q p
;;
```

Voici alors la fonction `numerote`. Elle va renvoyer le résultat sous forme d'une table de hachage (état de $\mathcal{A}_d \to$ entier). En outre, nous créerons également le tableau (entier $\to$ état de $\mathcal{A}_d$) qui permettra de retrouver l'état à partir de son numéro.

Afin d'éviter de créer trop d'états inutiles, nous prenons soin de ne mettre dans cette liste que les états accessibles. En fait, on suit exactement la même méthode que lorsqu'on déterminise à la main un automate. Ainsi, l'automate créé ne contiendra que des états accessibles ; pour finir de l'émonder il restera à supprimer les états non co-accessibles.

Enfin, c'est cette même fonction qui est la mieux placée pour calculer les états finals : à chaque fois que nous visitons un état de $\mathcal{A}_d$, il suffit de tester si son intersection avec F est non vide.

N.B. Au final, la fonction ci-dessous est un brave parcours de graphe ! C'est une version en profondeur écrite à l'aide de deux fonctions mutuellement récursives.

```
let numerote_etats_accessibles a alphabet=
  (* Renvoie une table de hachage (état accessible de Qd) -> int, ainsi que le tableau
      réciproque int -> (état)
Renvoie tant qu'on y est la liste des numéros des états finals de Qd *)

  (* Oh ! Un parcours de graphe ! *)
  let res = Hashtbl.create 47 (* nous servira aussi de dejaVu *)
  and liste_utiles = ref []
  and finals = ref []
  and i = ref 0 in (*  prochain entier à utiliser. *)

  let rec parcours s =
    (* s : sommet à visiter, type int list.*)
    if not (Hashtbl.mem res s) then begin
        Hashtbl.add res s !i;
        liste_utiles := s::!liste_utiles;
        if intersection s a.finalsND <> [] then finals:= !i::!finals;
        incr i;

        parcours_voisins s alphabet
      end


  and parcours_voisins s = function
    (* Prend une liste de char et traite les voisins de s auquels on accède en lisant ces
        lettres. *)
    | [] -> ()
    | x::q -> parcours (delta_d a s x);
              parcours_voisins s q

  in
  parcours a.initiaux;

  (* Création du tableau numro -> état *)
  let tab = Array.make !i [] in
  List.iteri (fun i  s -> tab.(i) <- s) (List.rev !liste_utiles);

  res, tab, !finals
;;
```

Voici enfin la fonction de déterminisation. L'automate obtenu n'est pas complètement émondé : nous pouvons utiliser la procédure `emonde` sur celui-ci avant de le renvoyer.

```
  (* Entrée : un AFND a, et l'alphabet correspondant.
     Sortie : un AFD reconnaissant le même langage *)


  (* Numérotation des états de Ad *)
  let dico_etats, tab_etats, finals = numerote_etats_accessibles a alphabet in
  let int_of_etat q = Hashtbl.find dico_etats q
  and etat_of_int i = tab_etats.(i)
  and n=Array.length tab_etats in


  (* La fonction delta_d, version prenant et envoyant un numéro d'état et non l'état
      lui-même. *)
  let delta_d_int i lettre =
    int_of_etat (delta_d a (etat_of_int i) lettre)
  in


  (* Création des transitions *)
  let trans = Array.make n [] in

```

```
  for i=0 to n-1 do
    trans.(i) <- List.map
                   (fun lettre ->
                     lettre, delta_d_int i lettre
                   )
                   alphabet
  done;


  (* On renvoie enfin l'automate déterminisé *)
  let ad={initial = int_of_etat a.initiaux;
   finals = finals;
   transitions=trans} in
  emonde ad;
  ad
;;
```

8.5 Application : automate reconnaissant les chaînes contenant un certain mot

Soit $u \in \Sigma^*$. Il est très facile de créer un automate pour reconnaître $\Sigma^* u \Sigma^*$. En le déterminisant, on obtient alors de manière automatique les automates que nous avions donner en exemple au tout début du chapitre.

Programmons ceci.

9 Complexité

9.1 Automate déterministe

Soit m un mot et $\mathcal{A}$ un automate déterministe. Tester si m est reconnu par $\mathcal{A}$ nécessite de lire une fois chaque lettre de m. Et pour chaque lettre lue, il s'agit d'explorer au pire toutes les arêtes issues de l'état actuel. Si nous notons v le nombre maximal d'arêtes issues d'un état, la complexité est donc $O(|m|v)$.

Dans un automate déterministe complet, le nombre d'arêtes issues de chaque état est de $\text{Card}(\Sigma)$. La complexité est donc $O\left(|m|\,|\Sigma|\right)$.

Si nous considérons le nombre de lettres de l'alphabet comme une constante, ceci devient $O(|m|)$: linéaire est $|m|$.

On remarque qu'un automate incomplet sera plus efficace car le nombre de transitions dans chaque état est moindre, et de plus la lecture pourra s'arrêter avant la fin de m s'il y a un blocage. D'où l'intérêt de l'émondage.

On notera aussi que l'implémentation que nous avons effectuée n'est pas optimale. En effet, pour chaque sommet i nous avons enregistré une liste d'association pour associer à chaque lettre l'état suivant. L'utilisation d'un ABR ou d'une table de hachage serait plus performante : on remplacerait $|\Sigma|$ par $\log|\Sigma|$ avec un ABR, et par un « gros $O(1)$ amorti » avec une table de hachage.

Encore mieux : si on numérotait les lettres de Σ on pourrait utiliser une matrice `transitions` tel que pour tout $(i,j) \in [\![0, |Q|[\![\times [\![0, |\Sigma|[\![$, `transitions.(i).(j)` serait l'état où on arrive depuis l'état i en lisant la lettre j. On remplace alors le $|\Sigma|$ par $O(1)$. Cependant, cette méthode nécessite un gros espace mémoire.

9.2 Automate non déterministe

Bien que souvent pratique à écrire, un automate non déterministe sera plus complexe à utiliser (d'où l'intérêt de la détermination puis de l'émondage).

En effet, nous maintenons une liste d'états accessibles. Et pour chaque lettre lue, nous devons calculer les états accessibles depuis chaque élément de cette liste. Le coût sera donc de l'ordre de « nombre d'états accessibles $\times$ $|\Sigma|$ ».

En majorant le nombre d'états accessibles par $|Q|$, le coût de la lecture d'une lettre est en $O\left(|Q| \times |\Sigma|\right)$, et le coût de la lecture d'un mot m est en :

$$O\left(|m| \times |Q| \times |\sigma|\right).$$

10 Autres manipulations d'automates

Voic quelques constructions classiques à partir d'automates. Elles ne sont pas strictement au programme : considérons-les comme des exercices classiques.

Le but de cette partie est, étant donné un ou des automates reconnaissant certains langages, comment obtenir de nouveaux automates pour reconnaître d'autres langages ? (Au contraire dans les parties précédentes, le but était d'obtenir de nouveaux automates pour reconnaître le même langage, mais plus efficacement).

10.1 Reconnaître le complémentaire

Soit a un automate reconnaissant un langage L. Pour reconnaître $\Sigma^* \setminus L$, c'est très simple : on inverse les états acceptants et non acceptants. Mais ceci ne fonctionne qu'à la condition que a soit déterministe et complet. Conclusion, nous allons :

1. Déterminiser a ;
2. Le compléter ;
3. Inverser les états acceptants et non acceptants ;
4. Et tant qu'on y est, l'émonder.

cf exercice : 18

10.2 Reconnaître une union, intersection, ou différence

cf exercice : 20

Exercices : langages et automates

1 Mots

Exercice 1. * Une sorte d'intégrité
Soit $(u, v) \in (\Sigma^*)^2$ tel que $uv = \epsilon$. Que dire de u et v ?

Exercice 2. ** Équation $ax = xb$
Soit $(a, b) \in \Sigma^2$ et $x \in \Sigma^*$ tel que $ax = xb$. Montrer que $a = b$ et que $x \in a^*$.

Exercice 3. ** Conjugaison

1. Soit $(x, y) \in (\Sigma^*)^2$ qu'on suppose non vides. Montrer qu'il existe $(u, v) \in (\Sigma^*)^2$ tel que $x = uv$ et $y = vu$ si et seulement si il existe $w \in \Sigma^*$ tel que $xw = wy$.

   Dans ce cas, x et y se déduisent l'un de l'autre par permutation circulaire. On dit que x et y sont « conjugués ».

2. Montrer que la relation « être conjugués » est une relation d'équivalence.

Exercice 4. * Mots commutants**
Soit $(x, y) \in (\Sigma^*)^2$. Montrer que x et y commutent ssi ils sont puissances d'un même mot.

2 Langages

Exercice 5. ** Langage contenant ϵ
Soit L un langage non vide. Montrer que $\epsilon \in L$ ssi $L \subset L \cdot L$.

Exercice 6. ** Autre manipulation de langages
Soient L_1 et L_2 deux langages contenant ϵ. Démontrer que $(L_1 L_2)^* = (L_1 + L_2)^*$.

Exercice 7. ** Fonction génératrice
Pour tout langage L, sa fonction génératrice est la fonction $\chi_L : x \mapsto \sum_{n=0}^{\infty} |L \cap \Sigma^n| \, x^n$.

On fixe ci-dessous un langage L.

1. Montrer que le rayon de convergence de χ_L est non nul.
2. Calculer χ_ϵ.
3. Soit $a \in \Sigma$, calculer χ_{aL} en fonction de χ_L.
4. Soit M un autre langage, disjoint de L. Calculer $\chi_{L \cup M}$ en fonction de χ_M et χ_L.
5. Soit M un autre langage. Exprimer $\chi_{L \cdot M}$ en fonction de χ_M et χ_L.

Exercice 8. ** Exemple d'égalité entre deux langages
Montrer que $\{a, b\}^* = (a^* b)^* a^*$.

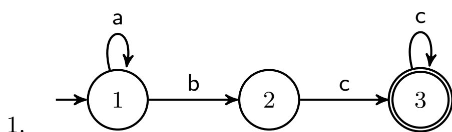
3 Exemples d'automates déterministes et de langages rationnels

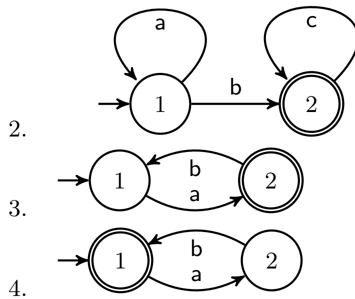
Exercice 9. * Le langage $\mathcal{A}^+$
On note Σ^+ l'ensemble des mots contenant au moins une lettre sur l'alphabet Σ^+. Démontrer que Σ^+ est rationnel et proposer un automate le reconnaissant.

Exercice 10. * Exemples simples et concrets
On note Σ l'ensemble des caractères disponibles dans l'encodage choisi. Décrire chacun des langages suivants par une expression régulière et donner un automate pour le reconnaître :

1. l'ensemble des chaînes de caractères sans espace ;
2. l'ensemble des numéros de téléphone, puis de téléphone portable ;
3. l'ensemble des adresses électroniques.

Exercice 11. * Exemples simples de lecture d'automates
On prend l'alphabet $\Sigma = \{a, b, c\}$. Que reconnaissent les automates suivants ?

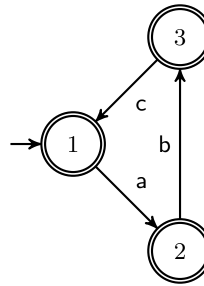
1. 



2.

3.

4.



5.

Exercice 12. * Mots utilisant un seul a

Déterminer un automate reconnaissant le langage des mots sur l'alphabet $\{a, b\}$ ne contenant qu'au plus un a. Puis exactement un a. Écrire une expression régulière décrivant ces langages.

Exercice 13. ** Suites croissantes

On prend ici $\Sigma^+ = [\![0, 9]\!]$ l'ensemble des chiffres, qu'on munit de sa relation d'ordre usuelle.

1. Montrer que l'ensemble des suites croissantes de chiffres est un langage rationnel. Donner une expression rationnelle le décrivant et un automate le reconnaissant.
2. Même question pour l'ensemble des suites strictement croissantes de chiffres.

Exercice 14. ** Avec une intersection

Pour cet exercice nous prenons $\Sigma = \{a, b\}$.

1. Donner une expression régulière et un automate pour l'ensemble des mots contenant un nombre pair de a. Puis pour l'ensemble des mots contenant un nombre de b multiple de 3.
2. (***) Donner un automate qui reconnaît l'ensemble des mots contenant un nombre pair de a et un nombre multiple de 3 de b.

4 Automates déterministes : exercices théoriques

Exercice 15. **! Utilité de plusieurs états acceptants

Montrer qu'il existe des langages reconnus par un automate déterministe à plusieurs états acceptants qui ne peuvent être reconnus par un automate déterministe à un seul état acceptant.

Par exemple considérer a^*b^*.

Exercice 16. ** Un lien avec le nombre d'états

Soit $\mathcal{A}$ un automate et n le nombre d'états de $\mathcal{A}$. Soit $L = \mathcal{L}(\mathcal{A})$. On suppose L non vide.

Montrer que L contient au moins un mot de longueur strictement inférieure à n.

Exercice 17. * Autre représentation

On représente un automate par la liste de ses transitions : `type automate = (int * char * int) list`. Programmer la reconnaissance d'un mot par un automate de ce type.

5 Opérations sur les automates déterministes

Exercice 18. ** Complémentaire

1. Soit $\mathcal{A}$ un automate complet déterministe et $L = \mathcal{L}(\mathcal{A})$. Décrire un automate reconnaissant le complémentaire de L dans Σ^*. Écrire une fonction Caml prenant en entrée l'automate $\mathcal{A}$ et renvoyant l'automate reconnaissant le complémentaire de L.
2. Même question pour un automate incomplet.
3. *(Avec le cours sur les automates non déterministes)* Même question pour un automate non déterministe.

Exercice 19. * Langage vide ?

Écrire une fonction prenant un automate et indiquant si le langage qu'il reconnaît est vide.

Exercice 20. **! Automate produit

Soient $\mathcal{A}_1 = (Q_1, i_1, F_1, \delta_1)$ et $\mathcal{A}_2 = (Q_2, i_2, F_2, \delta_2)$ deux automates déterministes et L_1 et L_2 les langages qu'ils reconnaissent.

On pose alors $Q = Q_1 \times Q_2$ et $\delta : \begin{array}{ccc} \Sigma \times Q & \to & Q \\ (x, (q_1, q_2)) & \mapsto & (\delta_1(x, q_1), \delta_2(x, q_2)) \end{array}$. Un automate basé sur cet ensemble d'états et cette fonction de transition s'appelle un automate produit.

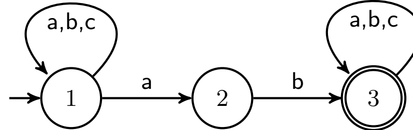
1. Démontrer que pour tout $(q_1, q_2) \in Q$ et tout $m \in \Sigma^*$, $\delta^*(m, (q_1, q_2)) = (\delta_1^*(m, q_1), \delta_2^*(m, q_2))$.
2. Définir un état initial et des états finals pour obtenir un automate qui reconnaît $L_1 \cap L_2$.

3. Même question pour reconnaître $L_1 \cup L_2$. On supposera ici $\mathcal{A}_1$ et $\mathcal{A}_2$ complets.
4. Même question pour $L_1 \setminus L_2$.
5. Écrire une fonction Caml prenant en entrée les deux automates $\mathcal{A}_1$ et $\mathcal{A}_2$ et renvoyant un automate qui reconnaît $L_1 \cap L_2$. On pourra, en notant $n_1 = |Q_1|$ et $n_2 = |Q_2|$, utiliser la fonction $\varphi : \begin{array}{ccc} [\![0, n_1[\![\times [\![0, n_2[\![& \to & [\![0, n_1 n_2[\![\\ (a, b) & \mapsto & a n_1 + b \end{array}$.

6 Automates non déterministes

Exercice 21. * Exemple d'automate non déterministe

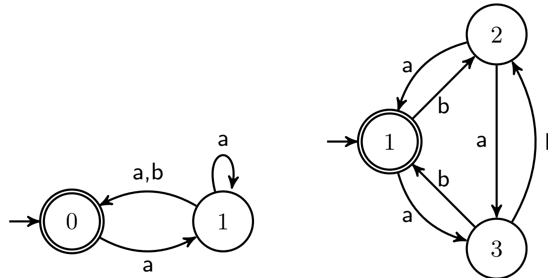
Soit $\Sigma = \{a, b, c\}$ et $\mathcal{A}$ l'automate suivant :



1. Quel langage reconnaît $\mathcal{A}$?
2. Déterminiser $\mathcal{A}$.
3. Proposer un automate déterministe qui reconnaît le langage complémentaire.

Exercice 22. ** Autres exemples de lecture d'automate

Déterminer le langage reconnu par les automates suivants :



Exercice 23. * Exemple simple d'écriture d'automate non déterministe

Trouver un automate non déterministe, puis déterministe, reconnaissant le langage $a\Sigma^* a$.

Exercice 24. * Reconnaître le langage des préfixes d'un mot

Soit u un mot.

1. Décrire un automate qui reconnaît l'ensemble des préfixes de u.
2. Mêmes questions pour l'ensemble des mots suffixes de u.
3. Mêmes questions pour l'ensemble des mots dont u est préfixe, et l'ensemble des mots dont u est suffixe.
4. Ces automates sont-ils déterministes ?
5. Écrire des fonctions Caml pour créer ces automates.

Exercice 25. ** Commentaires Caml

1. Écrire un automate pour reconnaître les commentaires Caml.
2. Déterminiser celui-ci.
3. Donner une expression régulière reconnaissant les commentaires Caml.

Exercice 26. ** ! En changeant les états initiaux et finals

Soit $a = (Q, q_0, F, \delta)$ un automate fini déterministe émondé, et $L = \mathcal{L}(a)$. Décrire le langage reconnu par :

1. (Q, q_0, Q, δ)
2. (Q, Q, F, δ)
3. (Q, Q, Q, δ)

Démontrer votre réponse.
Prendre soin d'utiliser le fait que a est émondé. La réponse attendue utilise les termes « préfixe », « suffixe », et « facteur ».

Exercice 27. ** ! Nombre d'états

Soit $\Sigma = \{a, b\}$ et $L = \Sigma^* a \Sigma^{n-1}$.

1. Trouver un automate non déterministe à $n+1$ états qui reconnaît L.
2. Soit $\mathcal{A}$ un automate déterministe qui reconnaît L, notons (Q, i, F, δ) ses composants.
   (a) Soient u et v deux mots distincts de longueur $n-1$. Prouver que $\delta^*(i, u) \neq \delta^*(i, v)$.

(b) Montrer que $\mathcal{A}$ a au moins 2^{n-1} états.

Exercice 28. ** Normalisation d'un automate

Un automate est dit *normalisé* s'il a un seul état initial, qui n'est la fin d'aucune transition, et un seul état final, dont aucune transition ne part.

1. Écrire un prédicat pour indiquer si un automate est normalisé.
2. Écrire une fonction qui prend un automate non déterministe $\mathcal{A}$ en argument et qui renvoie un automate normalisé reconnaissant le langage $\mathcal{L}(\mathcal{A}) \setminus \{\epsilon\}$.

Exercice 29. ** Calculer l'expression régulière associée à un automate

Voici un algorithme pour calculer une expression régulière qui définit le langage associé à un automate.

On va utiliser des automates dont les transitions seront étiquetées par des expressions régulières et non des lettres. On appellera ces automates des automates « généralisés ».

On commence par rajouter un nouvel état qui deviendra l'unique état initial et qu'on relie aux anciens états initiaux par des transitions étiquetées ε. On rajoute également un nouvel état qui devient l'unique état final et qu'on relie aux anciens états finals par des transitions étiquetées ε.

Ensuite on va supprimer l'un après l'autre tous les état, hormis l'état initial et l'état final, en appliquant les deux règles suivantes :

1. S'il existe deux transitions entre les mêmes états q_1 et q_2 étiquetées pas des expressions régulières e_1 et e_2, on les remplace par $q_1 \rightarrow^{e_1+e_2} q_2$.
2. Quand on supprime un état q : soit s l'étiquette de la boucle sur q (prendre $s = \varepsilon$ s'il n'y en a pas). Pour tout chemin $q_p \rightarrow^e q \rightarrow^f q_s$ passant par q, on le remplace par la transition $q_p \rightarrow es^*fq_s$.

Une fois éliminés tous les états sauf l'état initial et l'état final, l'expression régulière cherchée est l'étiquette de la seule transition restant.

1. Proposer un type Caml pour représenter un automate dont les transitions sont étiquetés par des expressions régulières. Pour simplifier les manipulations à suivre, les transitions seront enregistrées dans une matrice m telle que $m.(i).(j)$ est l'étiquette de la trasition depuis l'état i vers l'état j.
2. Écrire une fonction prenant un AFND classique et renvoyant l'automate généralisé décrit ci-dessus.
3. Appliquer cette méthode sur un exemple.
4. La programmer. Il n'est pas utile de supprimer réellement les états : se contenter de supprimer les transitions.

7 Utilisation d'un automate

Exercice 30. ** Utilisation concrète

Le but est de prendre un automate a et une chaîne de canactères m et de déterminer si une sous-chaîne de m est reconnue par a, et si oui de renvoyer des indices i et j tels que $m.[i : j] \in \mathcal{L}(a)$ (notations Python pour la sous-chaîne). On note Σ l'alphabet utilisé.

On utilisera des automates non déterministes. Récupérer :

- la définition du type ;
- les fonctions de manipulation le listes strictement croissantes ;
- la fonction `deltaND`.

1. Écrire si cela n'a pas été fait en cours une fonction `deltaListe` de type `afnd -> int list -> char ->int list` telle que si a est un automate de fonction de transition δ, si l est une liste d'états de a et x une lettre, `deltaListe a l x` renvoie $\bigcup_{q \in l} \delta(q, m)$. Le résultat sera sous forme d'une liste strictement croissante.
2. Écrire une fonction `indice_fin_motif` de type `afnd -> str -> int * int list` prenant un AFND a, une chaîne m, et renvoyant un couple (i, l) où i est un indice tel que $m.[0 : i] \in \mathcal{L}(a)$ et l est l'ensemble des états finals accessibles après avoir lu $m.[0 : i]$
3. Écrire une fonction `on_peut_rester` prenant un AFND a et renvoyant l'AFND obtenu à partir de a en rajoutant des transitions permettant de rester sur les états initiaux quelle que soit la lettre lue.
4. Écrire une fonction `retourned` prenant en entrée un AFND a et une liste d'états `nv_initiaux` et renvoyant l'automate obtenu en renversant chaque transition, et en mettant `nv_initiaux` comme ensemble d'états initiaux. Comme états finals on prendra les états initiaux de a.
5. Soit a un AFND et a' obtenu en renversant a comme à la question précédente. Notons (Q, I, F, δ) les composants de a et (Q, I', F', δ') ceux de a'. Pour tout mot m, on notera m^T l'image miroir de m.
   (a) Soit $q \in Q$ et $x \in \Sigma$. Que représente, dans a, $\delta'(q, x)$?
   (b) Soit $X \in \mathcal{P}(Q)$. Que représente, dans a, l'ensemble $(\delta')^*(X, m^T)$?
6. En déduire une fonction `place_motif` qui prend un automate a, l'alphabet Σ, et qui renvoie un couple (i, j) tel que $M.[i : j] \in \mathcal{L}(a)$ s'il en existe, et $(-1, -1)$ sinon.

Exercice 31. * L'énigme du barman aveugle**

Un barman joue au jeu suivant avec un client : il a devant lui quatre jetons d'Othello placés en carré sur un plateau. Pour mémoire, un jeton d'Othello a un côté noir et un côté blanc. À chaque tour de jeu, il peut retourner un ou deux jetons, puis le client peut faire un nombre arbitraire de quarts de tours au plateau. Son but est de mettre tous les jetons de la même couleur (le client lui dit quand il a réussi).

1. Montrer qu'il suffit de considérer quatre états possibles du jeu et trois manipulations de jetons.
2. Écrire l'automate décrivant le jeu. Il aura donc quatre états, et l'alphabet aura trois éléments.
3. En déduire une stratégie gagnante pour le barman.

Quelques indications

1 Utiliser la longueur.

2 On peut utiliser une récurrence sur $|x|$.

3 Pour le sens non évident, choisir w de longueur minimale.

4 Procéder par récurrence forte sur $|x| + |y|$.

5 Prendre le mot de L de longueur minimale.

7 1. Pour tout $n \in \mathbb{N}$, $|L \cap \Sigma^n| \leqslant |\Sigma|^n$.

8 Démontrer posément une inclusion puis l'autre.

14 Pour le deuxième : prendre comme états des couples $(i, j) \in [\![0, 1]\!] \times [\![0, 2]\!]$; et faire en sorte que l'automate arrive dans l'état (i, j) lorsque le nombre de a lus modulo 2 est i, et le nombre de b lus, modulo 3, est j.

15 Déjà, exhiber un automate avec plusieurs états acceptants qui reconnaît a^*b^*. Faire une preuve par l'absurde. Montrer qu'un automate avec un seul état acceptant qui reconnaît a^*b^* reconnaît aussi $(a + b)^*$.

16 Soit m un mot de L de longueur minimale et c le chemin acceptant correspondant. Vérifier que c n'a pas de boucle.

18 2. Le plus simple est de compléter $\mathcal{A}$, autrement dit de rajouter un puits.
3. Ici aussi, on constate que le plus simple est de déterminiser puis compléter $\mathcal{A}$...

19 Émonder l'automate.

20 1. Faire proprement une récurrence sur la longueur du mot lu.
5. La fonction φ servira à numéroter les états de l'automate produit. Sa réciproque est $\varphi^{-1} : i \mapsto (i/n_1, i \texttt{mod} n_1)$.

24 Pour les suffixes de u : utiliser un état initial qui permet d'aller à tous les autres états selon la lettre lue.

25 1. Un commentaire Caml doit commencer par `(*`, terminer par `*)` et ne doit pas contenir un autre `*)` au milieu.
2. Voir quel est le langage permettant de boucler sur l'état 2.

28 2) Créer un nouvel état initial et un nouvel état final.

30 Pour la fonction finale : commencer par obtenir l'indice j au moyen de `on_peut_rester` et de `indice_fin_motif`. Puis lire m à l'envers dans a' en partant de j jusqu'à retomber sur un élément de I.

31

Quelques solutions

1 On a $|u| + |v| = 0$ donc $|u| = 0$ et $|v| = 0$ (somme nulle de nombres positifs). Donc $u = \epsilon$ et $v = \epsilon$.

2 Pour tout $n \in \mathbb{N}$, posons $P(n)$: « Soit $x \in \Sigma^n$ tel que $ax = xb$. Alors $a = b$ et $x = a^n$. ».

- Soit $x \in \Sigma^0$. Alors $x = \varepsilon = a^0$. Donc $P(0)$.
- Soit $n \in \mathbb{N}$ tel que $P(n)$. Soit $x \in \Sigma^{n+1}$ tel que $ax = xb$. Soit u la première lettre de x et x' le mot formé de sus n dernières lettre. Ainsi $x = u \cdot x$' et l'hypothèse devient $aux' = ux'b$. Les premières lettres des mots de chaque côté du $=$ sont identiques, donc $a = u$. Puis $ux' = x'b$, ou encore $ax' = x'b$.
  On peut alors utiliser $P(n)$ à x' : on obtient $a = b$ et $x' = a^n$.
  Et enfin, $x = ax' = a^{n+1}$.

3
1. - Supposons qu'il existe $(u, v) \in (\Sigma^*)^2$ tel que $x = uv$ et $y = vu$. Il suffit alors de poser $w = v$ pour obtenir l'autre condition.
   - Supposons qu'il existe $w \in \Sigma^*$ tel que $xw = wy$. Prenons w de longueur minimale. Notons que $|x| = |y|$.
     Supposons que $|x| \leqslant |w|$. Vu l'égalité, on déduit qu'il existe $w' \in \Sigma^*$ tel que $w = xw'$. On déduit $xxw' = xw'y$, puis $xw' = w'y$. Ceci contredit le caractère minimal de w.
     Ainsi, $|x| > |w|$. Il existe alors $x' \in \Sigma^*$ tel que $x = wx'$. L'égalité devient $wx'w = wy$ puis $x'w = y$. Alors on constate que $u = w$ et $v = x'$ conviennent.
2. - **Réflexivité :** prendre $w = \varepsilon$.
   - **Symétrie :** immédiat vu la première caractérisation.
   - **Transitivité :** Soit x, y, z trois mots. Supposons qu'il existe $(w, w') \in (\Sigma^*)^2$ tel que $xw = wy$ et $yw' = w'z$.
     Alors $xww' = wyw' = ww'z$. Ainsi, x et z sont conjugués.

4 Pour tout $l \in \mathbb{N}$, soit $P(l)$: « pour tout $(x, y) \in (\Sigma^*)^2$ tel que $|x| + |y| = l$ et $xy = yx$, il existe $m \in \Sigma^*$ tel que x et y sont des puissances de m.»

- Soit $(x, y) \in (\Sigma^*)^2$ tel que $|x| + |y| = 0$. Alors $x = y = \varepsilon$. Et $m = \varepsilon$ convient. D'où $P(0)$.
- Soit $l \in \mathbb{N}$. Supposons que pour tout $k \in [\![0, l]\!]$, $P(k)$.
  Soit $(x, y) \in (\Sigma^*)^2$ tel que $|x| + |y| = l + 1$ et $xy = yx$.
  Supposons sans perte de généralité que $|x| \leqslant |y|$. Si $|x| = 0$, c'est-à-dire $x = \varepsilon$, alors $m = y$ convient.
  Sinon de l'égalité $xy = yx$ on déduit qu'il existe $y' \in \Sigma$ tel que $y = xy'$. Puis en simplifiant par x il vient $xy' = y'x$. Comme $|x| > 0$, on a $|x| + |y'| < |x| + |y|$ donc on peut utiliser l'hypothèse de récurrence : il existe $m \in \Sigma^*$ et $(p, q) \in \mathbb{N}$ tel que $x = m^p$ et $y' = m^q$. Dès lors $y = m^{p+q}$.
  D'où $P(l + 1)$.

D'où la preuve, par le théorème de récurrence.

5
- Supposons $\epsilon \in L$. Soit $m \in L$. Alors $m = m \cdot \epsilon \in L \cdot L$.
  Ceci prouve que $L \subset L \cdot L$.
- Supposons $L \subset LL$. Soit m un mot de L de longueur minimale.
  Comme $m \in L \cdot L$, il existe $(m_1, m_2) \in L^2$ tel que $m = m_1 m_2$. On a alors $|m| = |m_1| + |m_2|$, donc $|m_1| \leqslant |m|$. Mais comme m est un mot de longueur minimale de L, on a $|m_1| = |m|$, d'où $|m_2| = 0$, c'est-à-dire $m_2 = \epsilon$. Or $m_2 \in L$. □

6
- Comme les langages contiennent ϵ, on a $L_1 + L_2 \subset L_1 L_2$. Et l'étoile est croissante, donc $(L_1 L_2)^* \supset (L_1 + L_2)^*$.
- Réciproquement, on a $L_1 L_2 \subset (L_1 + L_2)^2 \subset (L_1 + L_2)^*$. D'où par croissance de $*$, $(L_1 L_2)^* = \left((L_1 + L_2)^*\right)^* = (L_1 + L_2)^*$

Les deux inclusions sont prouvées, donc $(L_1 L_2)^* = (L_1 + L_2)^*$.

7
1. Notons $a = |\Sigma|$. Pour tout $n \in \mathbb{N}$, $|L \cap \Sigma^n| \leqslant |\Sigma|^n = a^n$.
   Or la série entière $\sum_n a^n$ a un rayon de convergence de $\frac{1}{a}$. Donc χ_L a un rayon de convergence d'au moins $\frac{1}{a}$.
2. On trouve $\chi_\epsilon = 1$
3. Pour tout $n \in \mathbb{N}^*$, $(aL) \cap \Sigma^n = a \cdot \left(L \cap \Sigma^{n-1}\right)$, d'où $|aL \cap \Sigma^n| = \left|L \cap \Sigma^{n-1}\right|$.
   On trouve que pour tout $x \in]-R, R[$, $\chi_{aL}(x) = x\chi_L(x)$.
4. On trouve $\chi_{L \cup M} = \chi_L + \chi_M$.

8

9 $\Sigma^+ = \Sigma \cdot \Sigma^*$, donc Σ^+ est rationnel.

10

11

1. a^*bcc^*
2. a^*bc^*
3. $a(ba)^*$
4. $(ab)^*$
5. $(abc)^* + a(bca)^* + ab(cba)^*$.

12

13

14

- $b^*(ab^*a)^*b^*$, et $a^*\left(ba^*ba^*b\right)^* a^*$.

15 Soit $\Sigma^+ = \{a, b\}$ et $\mathcal{L} = a^*b^*$. On dessine facilement au automate à deux états acceptants qui reconnaît $\mathcal{L}$.

Soit a un automate déterministe reconnaissant $\mathcal{L}$. Supposons de plus que a n'a qu'un seul état acceptant. Notons q_0 son état initial.

Comme $\epsilon \in \mathcal{L}$, q_0 est un état acceptant, c'est donc le seul état acceptant de a. Comme $a \in \mathcal{L}$, on a une trasition $q_0 \underset{a}{\longrightarrow} q_0$. De même, $b \in \mathcal{L}$ donc $q_0 \underset{b}{\longrightarrow} q_0$.

Mais alors $(a+b)^*$ est reconnu par a, contradiction car $aba \notin \mathcal{L}$.

16 Comme L est non vide, l'ensemble $\big\{|m|\ ;\ m \in L\big\}$ admet un minimum (ensemble d'entiers naturels non vide). Soit $m \in L$ tel que $|m|$ soit ce minimum. Soit c le chemin acceptant dans $\mathcal{A}$ correspondant.

Le chemin c n'a pas de cycle, sans quoi en supprimant un cycle on obtiendrait un nouveau chemin acceptant strictement plus court, qui donnerait un nouveau mot dans L strictement plus court, ce qui contredit la définition de m.

Il ne passe donc qu'au plus une fois par chaque état de $\mathcal{A}$. Donc sa longueur, qui est aussi celle de m, est au plus de n.

17

18

19 Soit $\mathcal{A}$ un automate, et $\mathcal{A}_e$ son émondé. Alors $\mathcal{L}(\mathcal{A}) = \emptyset$ ssi $\mathcal{A}_e$ ne contient aucun état final.

20

21

22

1. $\big(aa^*(a+b)\big)^*$
2.

23

24 Le premier est déterministe, les autres non.

25

1.
2. $(*\left(\Sigma \setminus \{*\} + *\Sigma \setminus \{)\}\right)^* *)$.

26 Le premier reconnaît l'ensemble des préfixes des mots de L, le deuxième l'ensemble des suffixes, et le troisième l'ensemble des facteurs.

27

1.
2. (a) Notons w le plus grand suffixe commun de u et v. Quitte à échanger u et v, supposons que la première lettre en partant de la fin qui diffère entre u et v vaut a pour u et b pour v, donc il existe u', v' tels que $u = u'aw$ et $v = v'bw$. Soit $k = n - 1 - |w|$, on a alors $ua^k \in L$ et $va^k \notin L$. Ceci entraîne que $\delta^*(q_0, u) \neq \delta^*(q_0, v)$ (sans quoi on aurait $\delta^*(q_0, ua^k) \neq \delta^*(q_0, va^k)$).

   (b) La fonction $u \mapsto \delta^*(q_0, u)$ est injective entre Σ^{n-1} et Q.

28

31

Deuxième partie

Algorithme de Berry Sethi

Le but est d'écrire un algorithme qui permet, en lisant une expression régulière, de créer un automate qui reconnaît le langage associé.

1 Expression régulière

Comme dans le chapitre sur la logique, nous allons définir formellement la notion d'expression régulière. Il s'agit d'une formule abstraite qui permet de définir un langage.

En Caml, nous coderons une expression régulière au moyen du type suivant :

```
8  type reg_exp=
9    Lettre of char
10 |Etoile of reg_exp
11 |Somme of reg_exp*reg_exp
12 |Concat of reg_exp*reg_exp
13 ;;
```

Programmation : Écrire une fonction pour tester si le langage associé à une expression rationnelle contient ϵ.

```
39 let rec contient_eps = function
40   (* Indique si le language décrit par l'expression régulière passée en argument contient ε *)
41   |Lettre x -> false
42   |Etoile e -> true
43   |Somme (e, f) -> contient_eps e || contient_eps f
44   |Concat (e, f) -> contient_eps e && contient_eps f
45 ;;
```

Passons à la définition formelle. On fixe, comme dans la première partie de ce cours, un alphabet Σ.

Définition 1.1. *(Expression régulière)*
On définit les expressions régulières sur Σ par récursivité de la manière suivante :

- *$\emptyset$, ϵ, et les lettres sont des expressions régulières.*
- *Pour toute expression régulière e, (e^*) est une expression régulière.*
- *Pour toutes expressions régulières e et f, $(e+f)$ et $(e \cdot f)$ sont des expressions régulières.*

Remarque : On dit aussi que l'ensemble des expressions régulières est la clôture pour les opérations étoile, somme, et concaténation de $\emptyset$, ϵ, et des lettres.

À ce stade, une expression régulière est un objet formel, qui n'a pas encore de sens. Nous lui en donnons un ci-dessous :

Définition 1.2. *(Langage associé à une expression régulière)*
On définit la fonction $\mathcal{L}$ qui à toute expression régulière associe un langage de la manière suivante :

- $\mathcal{L}(\emptyset) = \emptyset$
- $\mathcal{L}(\epsilon) = \{\epsilon\}$
- $\forall x \in \Sigma$, $\mathcal{L}(x) = \{x\}$
- $\forall e \in \mathcal{R}(\Sigma)$, $\mathcal{L}(e^*) = \mathcal{L}(e)^*$
- $\forall (e, f) \in \mathcal{R}(\Sigma)^2$, $\mathcal{L}(e+f) = \mathcal{L}(e) \cup \mathcal{L}(f)$ *et* $\mathcal{L}(e \cdot f) = \mathcal{L}(e) \cdot \mathcal{L}(f)$.

On avait déjà utilisé ces notations sans formaliser, et sans utiliser la notation $\mathcal{L}$ dans la première partie du cours. De fait, on omet souvent le $\mathcal{L}$.

On remarque aussi qu'il y a des conflits de notation : le symbole $\emptyset$ et utilisé à la fois pour désigner l'expression rationnelle et le langage. De même, le symbole d'une lettre peut désigner à la fois la lettre, élément de Σ, mais aussi l'expression rationnelle. Heureusement, dans la partie programmation c'est clair : les expressions rationnelles sont les objets du type Caml qu'on a défini ci-dessus.

Encore un peu de vocabulaire :

Définition 1.3.
- *Deux expressions régulières e et f sont dites équivalentes lorsque $\mathcal{L}(e) = \mathcal{L}(f)$.*
- *Un langage L est dit régulier lorsqu'il existe une expression régulière e telle que $L = \mathcal{L}(e)$.*

La proposition suivante justifie que dans le type Caml précédent on n'a pas pris la peine de rajouter des constructeurs pour $\emptyset$ ou pour ϵ.

Proposition 1.4. *Soit e une expression régulière, autre que ϵ et $\emptyset$. Alors il existe une expression régulière e' telle que e équivaut à e' ou à $e' + \epsilon$ et e' n'utilise ni ϵ ni $\emptyset$.*

Démonstration :

On procède par récurrence. Pour cadrer exactement avec le théorème de la récurrence tel que vu en MPSI, on peut faire une récurrence sur la huateur de la formule (vue comme un arbre). Je me permets ici une rédaction plus rapide qui suit la définition même d'uen expression régulière (appelée parfois «induction struturelle». Il s'agit de traiter les différents cas de la définition.

- **Lettre** Soit x une lettre. Alors c'est une expression régulièrene contenant ni ε ni $\emptyset$.
- **Étoile** Soit R telle qu'il existe S telle que $R = S^*$. Supposons qu'il existe S' sans ε ni $\emptyset$ telle que $\mathcal{L}(S) = \mathcal{L}(S')$ ou $\mathcal{L}(S') \cup \{\varepsilon\}$. Dans les deux cas, $\mathcal{L}(R) = \mathcal{L}(S)^* = \mathcal{L}(S')^* \cup \{\varepsilon\} = \mathcal{L}((S')^*) \cup \{\varepsilon\}$.
- **Somme** : Soit R une expression régulièretelle qu'il existe S_1, S_2 telles que $R = S_1 + S_2$. On suppose qu'il existe S_1', S_2' sans ε ni $\emptyset$ telles que $\mathcal{L}(S_1) = \mathcal{L}(S_1')$ ou $\mathcal{L}(S_1') \cup \{\varepsilon\}$ et analogue pour S_2.
  Notons $R' = S_1' + S_2'$: on bien que R' ne contient ni ε ni $\emptyset$ et que $\mathcal{L}(R) = \mathcal{L}(S_1) \cup \mathcal{L}(S_1) = \mathcal{L}(S_1') \cup \mathcal{L}(S_2')$ ou $\mathcal{L}(S_1') \cup \mathcal{L}(S_2') \cup \{\varepsilon\} = \mathcal{L}(R')$ ou $\mathcal{L}(R') + \{\varepsilon\}$.
- **Concaténation :** Soit R une expression régulièretelle qu'il existe S_1, S_2 telles que $R = S_1 \cdot S_2$. On suppose qu'il existe S_1', S_2' sans ε ni $\emptyset$ telles que $\mathcal{L}(S_1) = \mathcal{L}(S_1') \cup^? \{\varepsilon\}$ et analogue pour S_2.
  Ici il y a plusieurs cas. J'indique dans le tableau ci-dessous ce qu'on prend pour R' selon que ε appartient à $\mathcal{L}(S_1)$ ou

$\mathcal{L}(S_2)$:	$\varepsilon \in \mathcal{L}(S_1)$	$\varepsilon \notin \mathcal{L}(S_1)$
$\varepsilon \in \mathcal{L}(S_2)$	$S_1' \cdot S_2' + S_1' + S_2'$	$S_1' \cdot S_2' + S_1'$
$\varepsilon \notin \mathcal{L}(S_2)$	$S_1' \cdot S_2' + S_2'$	$S_1' \cdot S_2'$

□

Remarque : Cotte proposition est plus importante qu'il n'y paraît car dans les constructions à suivre, $\emptyset$ et ϵ devront être écartés.

2 Langages locaux

Les langages locaux sont un type de langages pour lesquels il est très facile de créer un automate. Ils seront utilisés comme étape intermédiaire dans l'algorithme de Berry Sethi.

2.1 Définition, exemples

Définition 2.1. *(Langage local)*
Soit L un langage. On dit qu'il est local lorsqu'il existe $(P, S, N) \in \mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma) \times \mathcal{P}\left(\Sigma^2\right)$ tel que :

$$L \setminus \{\epsilon\} = \left(P\Sigma^* \bigcap \Sigma^* S\right) \setminus \Sigma^* N \Sigma^*.$$

Ainsi, P et S sont des ensembles de lettres qui donnent respectivement les premières et les dernières lettres autorisées pour un mot de L, et N est un ensemble de mots de deux lettres qui donne l'ensemble des facteurs de longueur 2 interdits pour les mots de L.
Remarque : Choix des lettres : « P » pour « préfixe », « S » pour « Suffixe » et « N » pour « non facteur ».
En d'autres termes, un mot m est dans L si et seulement si les trois conditions suivantes sont réunies :

- La première lettre de m est dans P
- La dernière lettre de m est dans S
- Aucun facteur de longueur 2 n'est dans N.

Remarque : On pourrait aussi donner les facteurs de longueur 2 autorisés dans les mots de L, c'est-à-dire $\Sigma^2 \setminus N$. C'est souvent plus pratique pour déterminer L, mais pas pour écrire la formule ci-dessus. En général je noterai F cet ensemble.

Un langage local L est uniquement déterminé par P, S, N et un booléen indiquant si $\epsilon \in L$. En notant α ce booléen, on notera $\text{Loc}(P, S, N, \alpha)$ le langage local correspondant.
Par ailleurs, si L est local, on peut retrouver ces paramètres facilement : P est l'ensemble des premières lettres des éléments des mots de L, S est l'ensemble de leurs dernières lettres, etc.
De manière générale, si L est un langage quelconque, si nous posons P l'ensemble des premières lettres des éléments de L, S l'ensemble de leurs dernières lettres, N l'ensemble des facteurs de longueur 2 qui n'apparaissent dans aucun mot de L, alors on a toujours l'inclusion :

$$L \setminus \{\epsilon\} \subset \left(P\Sigma^* \bigcap \Sigma^*\right) \setminus \Sigma^* N \Sigma^*.$$

L'inclusion réciproque est vraie si et seulement si L est local.

Exemple de langage régulier non local : Soit $L = a + (ab)^*$. Supposons que L est local, soit alors I, F, P, α tel que $L = \text{Loc}(I, F, P, \alpha)$.

- Comme $a \in L$, on a $a \in I \cap F$.
- Comme $abab \in L$, on a $\{ab, ba\} \in P$.

Mais alors, $aba \in L$: contradiction.

Programmation : Écrire des programmes prenant en entrée une expression régulière e et renvoyant les ensembles P, S et F comme ci-dessus.
N.B. On répète ce qui est dit ci-dessus : on peut calculer ces trois ensembles pour n'importe quel langage L, mais ce n'est que s'il est local que la connaissance de ces trois ensembles permet de retrouver L.
Remarque : Pour écrire le programme `calcul_F`, on aura besoin des deux premiers.

```
ε
let rec premieres_lettres = function
    (* Renvoie la listes des premières lettres des mots du langage décrit par la régexp passée en argument *)
  | Lettre x -> [x]
  | Etoile e -> premieres_lettres e
  | Somme(f1,f2) -> (* L(f1) U L(f2) *) fusion_stricte (premieres_lettres f1) (premieres_lettres f2)
  | Concat (f1,f2) -> (* L(f1).L(f2) *)
     if contient_eps f1 then fusion_stricte (premieres_lettres f1)  (premieres_lettres f2)
     else premieres_lettres f1
;;
```

```
ε
let rec dernieres_lettres = function
  (* Renvoie la listes des dernières lettres des mots du langage décrit par la régexp passée en argument *)
  | Lettre x -> [x]
  | Etoile e -> dernieres_lettres e
  | Somme(f1,f2) -> (* L(f1) U L(f2) *) fusion_stricte (dernieres_lettres f1) (dernieres_lettres f2)
  | Concat (f1,f2) -> (* L(f1).L(f2) *)
     if contient_eps f2 then fusion_stricte (dernieres_lettres f1)  (dernieres_lettres f2)
     else dernieres_lettres f2
;;
```

```
ε
let rec produit_cart l1 l2=
  match l1 with
        | [] -> []
        | t::q -> (List.map (fun x -> (t,x)) l2) @ (produit_cart q l2);;



produit_cart [1;2;3] [5;6;7];;

let rec facteurs_long_2 = function
  | Lettre x -> []
  | Etoile e -> (* e^* *)
     fusion_stricte  (facteurs_long_2 e)
       ( produit_cart (dernieres_lettres e) (premieres_lettres e))

  | Somme (f1, f2) -> (* L(f1) U L(f2) *)  fusion_stricte (facteurs_long_2 f1) (facteurs_long_2 f2)

  | Concat (f1, f2) -> (* L(f1).L(f2) *)
     union_tri
        [ (facteurs_long_2 f1);
          (facteurs_long_2 f2);
          ( produit_cart (dernieres_lettres f1) (premieres_lettres f2))
```

```
104            ]
105 ;;
```

2.2 Automate associé à un langage local

2.2.1 Définition

Définition 2.2. *(Automate associé à un langage local)*
Soit L un langage local et (P, S, N, α) les paramètres correspondants. On définit l'automate déterministe associé à (P, S, N, α) ainsi :

- *On utilise un état pour chaque lettre de Σ ainsi qu'un autre état. Pour tout $x \in \Sigma$, notons q_x l'état associé à x. L'état supplémentaire sera noté q_0 (en supposant que $0 \notin \Sigma$).*
- *On prend q_0 comme état initial.*
- *Pour tout $x \in P$, on ajoute une transition de q_0 à q_x étiquetée par x.*
- *Pour tout facteur de longueur 2 autorisé $xy \in \Sigma^2 \setminus N$, on rajoute une transition de q_x vers q_y étiquetée par y.*
- *Comme ensemble d'états acceptant, on prend $\{q_x \ ; \ x \in S\}$, auxquels on adjoint q_0 si α.*

2.2.2 Propriétés

Lemme 2.3. *On reprend les notations de la définition, et on note a l'automate associé à (P, S, N, α). L'automate a vérifie la propriété suivante : pour toute lettre $x \in \Sigma$, toutes les transitions étiquetées par x aboutissent au même état (c'est q_x).*

Un automate vérifiant cette propriété s'appelle un automate « local » :

Définition 2.4. *On dit qu'un automate $\mathcal{A}$ est local lorsque pour toute lettre x, toutes les transitions étiquetées par x aboutissent au même état.*

Voici la propriété qui nous intéresse :

Proposition 2.5. *On reprend les notations précédentes. L'automate a reconnaît L.*

Ainsi, tout langage local est reconnaissable par un automate, et il est en outre facile de construire cet automate.

Démonstration :

- Soit $m \in \mathcal{L}(a)$. Notons $n = |m|$ et $x_1 \dots x_n$ ses lettres.
  - En lisant la première lettre de m, il n'y a pas eu de blocage. Donc x_1 est l'étiquette d'une transition issue de q_0, donc $x_1 \in P$.
  - La dernière lettre a mené à un état final, donc q_{x_n} est un état final, donc $x_n \in S$.
  - Soit $i \in [\![1, n-1]\!]$. Après la lecture de la lettre x_i, on était dans l'état q_i. Puis en lisant x_j il n'y a pas eu de blocage, donc $q_{x_i} \underset{x_j}{\longrightarrow} q_{x_j}$ est une transition dans a, donc $x_i x_j \in F$.

  Les trois conditions sont vérifiées, donc m est bien dans L.
- Réciproquement : soit $m \in L$, notons encore $n = |m|$ et $x_1 \dots x_n$ ses lettres.
  - Comme $x_1 \in P$, la transition $q_0 \underset{x_1}{\longrightarrow} q_1$ est présente dans a. Ainsi, après la lecture de x_1 nous sommes dans l'état q_{x_1}.
  - On prouve par récurrence que pour tout $i \in [\![1, n]\!]$, après lecture de $x_1 \dots x_i$, on arrive à l'état x_i. Pour l'hérédité : le fait que $x_i x_{i+1} \in F$ entraîne la présence de la transition $q_{x_i} \underset{x_{i+1}}{\longrightarrow} q_{x_{i+1}}$.
  - Ainsi, après lecture de m, nous sommes dans l'état q_{x_n}. Mais $x_n \in S$, donc q_{x_n} est un état acceptant.

  Ainsi, m est reconnu par a, donc $m \in \mathcal{L}(a)$.

□

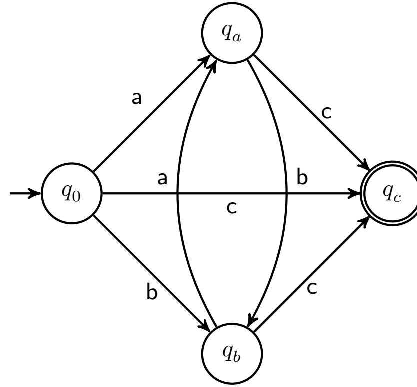
Une deuxième propriété de a est qu'aucune transition ne permet de revenir à l'état initial. On dit que a est « standard » :

Définition 2.6. *Un automate déterministe dont aucune transition n'aboutit à l'état initial est appelé « standard ».*

Remarques :

- L'automate ainsi construit est déterministe (une trasition étiquetée par une lettre x ne peut arriver quqà l'état q_x)
- Il n'est pas complet car pour tout $xy \in \Sigma^2 \setminus F$, il n'a pas de transition issue de q_x étiquettée par y.

Exemple : Soit $e = (a + b)^* c$. Alors $L(e)$ est local, décrit par les paramètres $P = \{a, b, c\}$, $S = \{c\}$, $F = \{ab, ba, aa, bb, ac, bc\}$ et $\alpha = \perp$. Ainsi l'automate local associé est :



À titre d'exercice (ex 4), on peut démontrer la propriété suivante, qui est la réciproque du lemme ci-dessus :

Proposition 2.7. *Le langage reconnu par un automate local est un langage local.*

Démonstration : Quitte à rajouter un nouvel état initial, on peut supposer a standard. Quitte à émonder a, on peut le supposer émondé.

Soit (Q, q_0, F, δ) les éléments constitutifs de a.

Pour tout $x \in \Sigma$, notons q_x l'état où mènent les transitions étiquetées par x (s'il existe des transitions étiquetées par x). Comme a est émondé, il n'y a pas d'autre état : $Q = \left\{q_x \;;\; x \in \Sigma\right\} \cup \{q_0\}$.

On pose $S = \left\{x \in \Sigma \mid\; q_x \in F\right\}$, $P = \left\{\delta(q_0, x) \;\middle|\; x \in \Sigma\right\}$, $\alpha = (q_0 \in F)$ et $N = \left\{xy \in \Sigma^2 \;\middle|\; \delta(q_x, y) \text{ est un blocage }\right\}$. On vérifie alors que $\mathcal{L}(a) = \mathrm{Loc}(P, S, N, \alpha)$. □

2.2.3 Programmation

Pour commencer il faudra récupérer les lettres intervenant dans la expression régulière. On crée la fonction idoine :

```
1   ε
120 let rec lettres_dans_regexp = function
121   (* Renvoie la liste des lettres apparaissant dans la regexp (sans doublon) *)
122   |Lettre x -> [x]
123   |Etoile e -> lettres_dans_regexp e
124   |Somme (r1,r2) -> fusion_stricte (lettres_dans_regexp r1) (lettres_dans_regexp r2)
125   |Concat(r1,r2) -> fusion_stricte (lettres_dans_regexp r1) (lettres_dans_regexp r2)
126 ;;
```

Ensuite on crée un état pour chaque lettre. Dans notre type Caml, les états sont des entiers. Il faut donc associer à chaque lettre un entier. Ci-dessous, on a utilisé une simple liste d'association :

```
1   ε
132 let rec numerote i = function
133   (* i est le prochain numéro à utiliser *)
134   |[] -> []
135   |t::q -> (t,i) :: numerote (i+1) q
136 ;;
```

Voici alors le programme final :

```
1   ε
142 let auto_local e=
143   (* Entrée : une regexp e
144      Précondition : L(e) est local
145      Sortie : un automate qui reconnaît L(e)
146    *)
147
148   let alphabet = lettres_dans_regexp e in
149   let dico = numerote 1 alphabet in (* liste d'asso lettre->numéro *)
150   let int_of_lettre x = List.assoc x dico in
151   let n = List.length alphabet + 1 in (* nb d'états *)
152   let trans = Array.make n [] in
153
```

```
154   (* On récupère les élément car du langage local L(e) *)
155   let p = premieres_lettres e and s=dernieres_lettres e and f= facteurs_long_2 e and alpha =
          ↪ contient_eps e in
156
157
158   (* Lecture de p pour créer les transitions issues de q0 *)
159   trans.(0) <- List.map
160                  (fun x -> (x, int_of_lettre x    ) (* q0 ->_x q_x *)    )
161                  p ;
162
163
164   (* Lecture de f pour créer les autres transitions *)
165   List.iter
166     (fun (x,y) -> let qx = int_of_lettre x and qy = int_of_lettre y in
167                 ( trans.(qx) <- (y, qy) :: trans.(qx)           )
168                 (* Rajoute la trans qx ->_y qy *)
169     )
170     f;
171
172   {initial = 0 ;
173    finals =  (if alpha then [0] else []  ) @  List.map int_of_lettre s ;
174    transitions = trans
175   }
176 ;;
```

2.3 Opération sur les langages locaux

Proposition 2.8. *(Opérations sur les langages locaux)*
L'intersection de langages locaux est un langage local. L'étoile d'un langage local est local.

Démonstration :

- **Intersection :** Soient L_1 et L_2 deux langages locaux, soient $(P_1, S_1, N_1, \alpha_1)$ et $(P_2, S_2, N_2, \alpha_2)$ décrivant ces langages en tant que langages locaux.
  Soit L le langage local décrit par $\left(P_1 \cap P_2, S_1 \cap S_2, N_1 \cup N_2, \alpha_1 \wedge \alpha_2\right)$. Prouvons que $L = L_1 \cap L_2$.
  - Soit $m \in L$. Si $m = \epsilon$, c'est que $\alpha_1 \wedge \alpha_2 = \top$ et donc α_1 et α_2 sont vrais, et donc $\epsilon \in L_1 \cap L_2$.
    Si $m \neq \epsilon$, sa première lettre est dans $P_1 \cap P_2$ donc en particulier dans P_1. De même, sa dernière lettre est dans S_1. Il n'a aucun facteur de longueur 2 dans $N_1 \cup N_2$, donc en particulier aucun facteur dans N_1. Ainsi, il est dans L_1.
    On vérifie de même qu'il est dans L_2.
    On a donc $L \subset L_1 \cap L_2$.
  - Réciproquement, soit $m \in L_1 \cap L_2$. Si $m = \varepsilon$, c'est que α_1 et α_2. Donc $\alpha_1 \wedge \alpha_2$ et $m \in L$.
    Sinon, sa première lettre est dans P_1 car $m \in L_1$, et dans P_2 car $m \in L_2$, donc dans $P_1 \cap P_2$. De même, sa dernière lettre est dans $S_1 \cap S_2$, et ses facteurs de deux lettres ne sont ni dans N_1 ni dans N_2.
- **Étoile** Soit L un langage local décrit par le quadruplet (P, S, N, α). Notons $F = \Sigma^2 \setminus N$ l'ensemble des facteurs de longueur 2 autorisés dans L.
  Posons $F' = F \cup (S \cdot P)$. Vérifions que L^* est le langage local décrit par $(P, S, \Sigma^2 \setminus F', \top)$.
  - Soit $m \in L^*$. Soit $n \in \mathbb{N}$ et $(m_1, \dots, m_n) \in L^n$ tel que $m = m_1 \dots m_n$.
    - La première lettre de m est la première lettre de m_1, elle est dans P car $m_1 \in L$.
    - Idem pour la dernière lettre.
    - Soit f un facteur de longueur 2. S'il est dans un m_i, alors c'est un élément de F. Sinon, c'est qu'il existe $i \in [\![1, n-1]\!]$ tel que f est formé de la dernière lettre de m_i et de la première de m_{i+1}. Dans ce cas, $f \in S \cdot P$.
  - Réciproquement, soit m dans le langage décrit par $(P, S, \Sigma^2 \setminus F', \top)$. Si $m = \epsilon$, il est bien dans L^*. Sinon, notons k le nombre de facteurs de longueur 2 de m qui sont dans SP, et notons $i_1, \dots, i_k$ l'emplacement de ces facteurs. Notons encore $n = |m|$ et $x_1, \dots, x_n$ les lettres de m. Alors m s'écrit ainsi :

    $$m = x_1 \dots x_{i_1} x_{i_1+1} \dots x_{i_2} x_{i_2+1} \dots x_{i_k} x_{i_k+1} \dots x_n.$$

    et chaque facteur $x_{i_q+1} \dots x_{i_{q+1}}$ (en notant $i_q = 0$ et $i_{k+1} = n$) est un élément de L. Donc $m \in L^*$.

En revanche, cela ne fonctionne pas pour la concaténation, l'étoile, ou la réunion.

- *Union* : une union de langages locaux n'est pas forcément un langage local. Par exemple, soit $L_1 = \{ab\}$. C'est le langage local défini par $(\{a\}, \{b\}, \{ab\}, \bot)$. Soit $L_2 = a^*$, le langage local défini par $(\{a\}, \{a\}, \emptyset, \top)$.
  Supposons que $L_1 \cup L_2$ soit local, soit alors (I, F, P, α) un quadruplet décrivant $L_1 \cup L_2$. Comme $L_1 \cup L_2$ contient ab, on a déjà $a \in I$, $b \in F$ et $ab \in P$. Ensuite, comme $aa \in L_1 \cup L_2$, on a aussi $a \in F$ et $aa \in P$. Mais alors $aab \in L_1 \cup L_2$: contradiction.
- *concaténation :* une concaténation de langages locaux n'est pas forcément un langage local. Reprenons L_1 et L_2 comme avant. Supposons $L_1 \cdot L_2$ local, et soit (I, F, P, α) un quadruplet décrivant $L_1 \cdot L_2$. Notons que $L_1 \cdot L_2 = aba^*$.
  Le fait que $aba \in L_1 \cdot L_2$ entraîne que $a \in I \cap F$ et $\{ab, ba\} \subset P$. Mais alors $ababa \in L_1 \cdot L_2$: contradiction.

3 Expression rationnelle linéaire

Définition 3.1. *Une expression régulière e est dite linéaire lorsque chaque lettre de Σ apparaît au plus une fois dans e.*

Proposition 3.2. *(Union et concaténation de langages locaux sur des alphabets disjoints)*
Soient Σ_1 et Σ_2 deux alphabets disjoints*. Soient L_1 et L_2 deux langages locaux sur Σ_1 et Σ_2, respectivement.*
Alors $L_1 \cdot L_2$ et $L_1 \cup L_2$ sont locaux.

Démonstration : Soit $(P_1, F_1, S_1, \alpha_1)$ et $(P_2, F_2, S_2, \alpha_2)$ des paramètres décrivant L_1 et L_2 comme langages locaux.

- **union** On va prouver que $L_1 \cup L_2$ est le langage local décrit par $(P_1 \cup P_2, F_1 \cup F_2, S_1 \cup S_2, \alpha_1 \vee \alpha_2)$. En attendant, notons K ce langage.
  L'inclusion non évidente est $K \subset L_1 \cup L_2$. Soit donc $m \in K$. Supposons par exemple que sa première lettre est dans P_1, donc a fortiori, elle est dans Σ_1. Or aucun facteur de longueur 2 de m ne combine une lettre de Σ_1 avec une lettre de Σ_2. Dès lors, toutes les lettres de m sont dans Σ_1. On prouve alors sans mal que $m \in L_1$.
- **Concaténation :** Traitons le cas où $\neg\alpha_1$ (c'est-à-dire $\epsilon \notin L_1$). Notons K le langage local défini par $(P_1, F_1 \cup F_2 \cup S_1 P_2, S_2, \bot)$. Et montrons que $K = L_1 \cdot L_2$.
  - Soit $m \in K$. Notons $n = |m|$ et $m_1 \dots m_n$ les lettres de m. Il n'y a aucun facteur de longueur 2 possible dont la première lettre est dans Σ_2 et la deuxième dans Σ_1. En gros, une fois qu'on passe dans Σ_2, on ne peut plus repasser dans Σ_1. Comme de plus, $m_1 \in \Sigma_1$ et $m_n \in \Sigma_2$, il existe $k \in [\![1, n-1]\!]$ tel que $m_1 \dots m_k \in \Sigma_1^k$ et $m_{k+1} \dots m_n \in \Sigma_2^{n-k}$.
    Le facteur $m_k m_{k+1}$ est dans $F_1 \cup F_2 \cup S_1 P_2$ et il est formé d'une lettre de Σ_1 et d'une lettre de Σ_2. Il est donc dans $S_1 P_2$. Ainsi, $m_k \in S_1$ et $m_{k+1} \in P_2$.
    On termine de prouver sans mal que $m_1 \dots m_k \in L_1$ et $m_{k+1} \dots m_n \in L_2$.
  - Réciproquement, soit $m \in L_1 \cdot L_2$. Soit $m_1 \in L_1$ et $m_2 \in L_2$ tel que $m = m_1 \cdot m_2$.

On déduit alors immédiatement :

Théorème 3.3. *Soit e une expression régulière linéaire. Alors $\mathcal{L}(e)$ est local.*

Démonstration : Par récursivité sur e, en utilisant les propositions 3.2 et 2.8.

Remarque : La réciproque est fausse. Par exemple aa^* est un langage local qui n'est pas représentable par une expression régulière linéaire.

4 Algorithme de Berry-Sethi / automate de Glushkov

On fixe pour toute cette partie une expression régulière e.
L'algorithme de Glushkov consiste à :

1. transformer e en une expression régulière linéaire e' (étape de « linéarisation ») ;
2. créer l'automate local associé à e' ;
3. le transformer en un automate reconnaissant $\mathcal{L}(e)$.

4.1 Linéarisation

Soit n le nombre de lettre intervenant dans e. Soit Σ' un alphabet quelconque de n lettres, que nous notons $x_1, \dots x_n$.
Soit enfin e' l'expression régulière obtenue en remplaçant pour tout $i \in [\![0, n[\![$ la i-ème lettre de e par x_i. Ainsi, e' est une expression régulière locale. Soit a' l'automate local associé à e', cette automate reconnaît $\mathcal{L}(e')$ (en formule, $\mathcal{L}(a') = \mathcal{L}(e')$).

4.2 Retour à l'expression rationnelle de départ

Soit $\varphi : \Sigma' \to \sigma$ la fonction telle que pour tout $i \in [\![0, n[\![$, $\varphi(x_i)$ est la i-ème lettre apparaissant dans e.

La fonction φ s'étend naturellement en une fonction, encore notée φ sur les mots : pour tout $k \in \mathbb{N}$ et tout mot $x_0 \dots x_{k-1} \in (\Sigma')^k$, on pose $\varphi(x_0 \dots x_{k-1}) = \varphi(x_0) \dots \varphi(x_{k-1})$. On obtient de la sorte une fonction $\varphi : (\Sigma')^* \to \Sigma^*$, et de plus cette fonction est un morphisme pour la concaténation (morphisme de monoïde pour donner le vocabulaire précis : elle préserve le neutre et l'opération $\cdot$).

On peut même étendre la fonction φ aux langages : pour tout $L \in \mathcal{P}\left((\Sigma')^*\right)$, on pose $\varphi(L) = \left\{\varphi(m)\ ;\ m \in L\right\}$. C'est juste la définition habituelle de l'image directe d'un ensemble par une fonction.

On vérifie sans peine que pour tout $(L_1, L_2) \in P\left((\Sigma')^*\right)^2$,

- $\varphi(L_1 \cup L_2) = \varphi(L_1) \cup \varphi(L_2)$ (propriété de l'image directe d'un ensemble par une fonction) ;
- $\varphi(\emptyset) = \emptyset$ (idem) ;
- $\varphi(L_1 \cdot L_2) = \varphi(L_1) \cdot \varphi(L_2)$ (car φ est compatible à $\cdot$) ;
- $\varphi(L_1^*) = \varphi(L_1)^*$.

Enfin, on étend aussi φ aux expressions rationnelles : pour toute expression rationnelle f sur Σ on pose $\varphi(f)$ l'expression rationnelle sur Σ' obtenue en remplaçant toute lettre x dans f par la lettre $\varphi(x)$ et en maintenant tout le reste de la formule.

Les propriétés ci-dessus permettent de prouver que $\mathcal{L}\left(\varphi(f)\right) = \varphi\left(\mathcal{L}(f)\right)$.

En gros, φ permet de transformer tout objet associé à e' en son objet analogue concernant e, et φ est compatible à toutes les opérations liées à la structure de langage. Par conséquent, φ va également nous permettre de transformer l'automate a' qui reconnaît $\mathcal{L}(e')$ en un automate a qui reconnaît $\varphi(e)$.

Définition 4.1. *Soit (Q', q_0', F', δ') les éléments constitutifs de a'.*
Soit a l'automate non déterministe définit par :

- *Son ensemble d'état, et et son ensemble d'états finaux sont le même que pour a, ainsi on pose $Q = Q'$ et $F' = F$.*
- *Son ensemble d'états initiaux est $\{q_0\}$, on pose donc $I = \{q_0\}$.*
- *Son ensemble de transitions est obtenu en remplaçant toute étiquette x d'une transition dans a' par $\varphi(x)$.*
  Formellement, on définit la fonction δ par $\forall q \in Q$, $\forall x \in \Sigma$, $\delta(q, x) = \left\{\delta'(q, x')\ ;\ x' \in \varphi^{-1}(\{x\})\right\}$.

On note $a = (Q, I, F, \delta)$.

Remarque : À cette étape, nous sommes passés d'un automate déterministe à un automate non déterministe. En effet, partant d'un état q, il pouvait y avoir des transitions étiquetées par différentes lettres dans a' qui se sont retrouvées étiquetées par la même lettre dans a.

On pourra donc dans un second temps déterminiser a.

Théorème 4.2. *On a $\mathcal{L}(a) = \mathcal{L}(e)$.*

Ainsi, a est l'automate recherché : il reconnaît le langage $\mathcal{L}(e)$.
Démonstration : On rappelle que $\varphi(e') = e$, et donc $\varphi\left(\mathcal{L}(e')\right) = \mathcal{L}(e)$.

- Soit $m \in \mathcal{L}(e)$. Il existe $m' \in \mathcal{L}(e')$ tel que $m = \varphi(m')$. Comme $m' \in \mathcal{L}(e')$, m' est reconnu par a'. Notons $n = |m'|$. Il existe un chemin $r_0 \underset{m_1'}{\longrightarrow} r_1 \underset{m_2'}{\longrightarrow} r_2 \dots \underset{m_n'}{\longrightarrow} r_n$ dans a' tel que $r_0 = q_0$ et $r_n \in F$.
  Mais alors, par définition de a, le chemin $r_0 \underset{\varphi(m_1')}{\longrightarrow} r_1 \underset{\varphi(m_2')}{\longrightarrow} r_2 \dots \underset{\varphi(m_n')}{\longrightarrow} r_n$, $r_0 \underset{m_1}{\longrightarrow} r_1 \underset{m_2}{\longrightarrow} r_2 \dots \underset{m_n}{\longrightarrow} r_n$c'est-à-dire est un chemin valide dans a. Ceci prouve que $m_1 \dots m_n$ est reconnu par a. Mais $m_1 \dots m_n = \varphi(m_1') \dots \varphi(m_n') = \varphi(m_1' \dots m_n') = \varphi(m') = m$. Donc $m \in \mathcal{L}(a)$.
- Réciproquement, soit $m \in \mathcal{L}(a)$ et n sa longueur. Il existe donc un chemin $r_0 \underset{m_1}{\longrightarrow} r_1 \underset{m_2}{\longrightarrow} r_2 \dots \underset{m_n}{\longrightarrow} r_n$ dans a tel que $r_0 = q_0$ et $r_n \in F$.
  Pour tout $i \in [\![1, n]\!]$, soit $m_i' \in \Sigma'$ la lettre correspondant à l'état r_i (on rappelle que les états de a sont les mêmes que ceux de a', et qu'il y en a un associé à chaque lettre de Σ').
  La transition $r_{i-1} \to r_i$ venait d'une transition de a', et cette transition est étiquetée par m_i' car dans un automate local, toutes les transitions menant à un état q_x sont étiquetées par x. Ainsi, on a dans a' la transition $r_{i-1} \underset{m_i'}{\longrightarrow} r_i$. De plus, on a $\varphi(m_i') = m_i$ vu la définition des transitions de a.
  Et finalement, le chemin suivant existe dans a' : $r_0 \underset{m_1'}{\longrightarrow} r_1 \underset{m_2'}{\longrightarrow} r_2 \dots \underset{m_n'}{\longrightarrow} r_n$, donc le mot $m_1' \dots m_n'$ est reconnu par a. Notons m' ce mot. On a $m' \in \mathcal{L}(e')$ et $m = \varphi(m') \in \varphi\left(\mathcal{L}(e')\right) = \mathcal{L}(e)$.

Les deux inclusions sont prouvées : on a bien $\mathcal{L}(a) = \mathcal{L}(e)$.

4.3 Programmation

On commence par la linéarisation. Il convient de retenir quelle lettre de Σ' a été associée à quelle lettre de Σ, autrement dit de renvoyer la fonction φ en même temps.

```
1   ε
201 let linearised_et_phi e=
202   (* Entrée une regexp e
203      Sortie :
204               (regexp e' obtenue en changeant des lettres dans e pour obtenir une regexp
                      ↪ linéaire,
205               phi)
206               phi sera renvoyée sous forme d'une liste d'asso (lettre de e', lettre de e
                      ↪ correspondante)
207    *)
208
209
210   let rec aux i = function
211       (* argument supp : i, le code ASCII du prochain caractère à utiliser. càd la prochaine
              ↪ lettre à utiliser sera char_of_int i
212        sortie : couple (la regexp obtenue, le prochain i à utiliser, phi)
213        *)
214     | Lettre x -> Lettre (char_of_int i), [(char_of_int i, x )]    ,  i+1
215
216     | Etoile f -> let (f', phi, prochain_i) = aux i f in  (Etoile f', phi, prochain_i)
217
218     | Concat (f1, f2) -> let (f1', phi1, prochain_i) = aux i f1 in
219                          let (f2', phi2, prochain_prochain_i) =  aux prochain_i f2 in
220                          Concat (f1',f2'), phi1@phi2    , prochain_prochain_i
221
222     | Somme(f1, f2) ->  let (f1', phi1, prochain_i) = aux i f1 in
223                         let (f2', phi2,  prochain_prochain_i) =  aux prochain_i f2 in
224                         Somme (f1',f2'),  phi1@phi2    , prochain_prochain_i
225   in
226   let (e',phi,_)= aux 97 e
227   in (e',phi)
228 ;;
```

L'algorithme de Berry-Sethi est maintenant simple à coder :

```
1   ε
238 let berry_sethi e =
239   (* Renvoie un automate qui reconnaît L(e) *)
240
241   (* 1) on linéarise *)
242   let e', phi = linearised_et_phi e in
243   (* phi est une liste d'asso *)
244   let fonction_phi x= List.assoc x phi in
245
246   (* 2) auto local de e'. Possible car L(e') est local *)
247   let a' = auto_local e' in
248
249   (* 3) obtention de a qui reconnaît L(e) à partir de a' *)
250   (* Il s'agit d'appliquer phi à chaque étiquette de transition dans a' *)
251   (* Attention : a' est déterministe, mais pas a *)
252
253   let n = Array.length (a'.transitions) in
254   let transitions = Array.make n [] in
255   for i=0 to n-1 do
256     transitions.(i) <- List.map
257                         ( fun (x, q)->( fonction_phi x, q)    )
258                          a'.transitions.(i)
259       (* a'.transitions(i) où chaque lettre x a été remplacée par phi(x) *)
260   done;
261
```

```
262
263  {
264    initiaux = [a'.initial]  ;
265    finalsND = a'.finals  ;
266    transitionsND = transitions
267  }
268 ;;
```

5 Conséquences, et compléments

Le fait que l'algorithme de Glushkov existe et fonctionne pour toute expression régulière prouve le

Théorème 5.1. *Tout langage régulier est reconnaissable par un automate.*

Mais on peut démontrer la réciproque (non exigible selon le programme officiel). C'est le

Théorème 5.2. *(Kleene)*
Un langage est régulier si et seulement si il est reconnaissable par un automate.

Démonstration :
Le sens direct (« seulement si ») a déjà été vu.
Réciproquement, soit a un automate. Il s'agit de trouver une expression régulière e telle que $\mathcal{L}(e) = \mathcal{L}(a)$. Pour ce, l'idée est de supprimer un à un les sommets de a en utilisant des arêtes étiquetées par des expression rationnelles et non plus par des lettres.

Corollaire 5.3. *L'ensemble des langages réguliers est stable par intersection et par passage au complémentaire.*

N.B. L'ensemble des langages réguliers est évidemment stable par union, concaténation et étoile.
Démonstration : Nous avons déjà vu que ceci est vrai pour les langages reconnaissables. Et maintenant, nous savons que les langages réguliers sont les langages reconnaissables.
Avec plus détails :

- *Intersection :* Soient L_1 et L_2 deux langages réguliers. Soient a_1 et a_2 deux langages déterministes reconnaissant L_1 et L_2, obtenus en déterminisant les automates de Glushkov. La méthode de xsl'automate produit permet de créer un automate a reconnaissant $L_1 \cap L_2$ (exercice 20). Donc a est reconnaissable, et par la réciproque du théorème de Kleene, $L_1 \cap L_2$ est régulier.

Corollaire 5.4. *L'ensemble des langages reconnaissables est stable par intersection, union, concaténation, et étoile.*

Démonstration : Même principe que ci-dessus : on le sait déjà pour les langages réguliers, et maintenant on sait que les langages reconnaissables sont les langages réguliers.

Mais les détails sont intéressants :

- *Intersection et union :* déjà vu : utiliser l'automate produit
- *Passage au complémentaire :* déjà vu : passer à un automate complet et déterministe, et inverser les états acceptants et non acceptants.
- *Concaténation :* soient L_1 et L_2 deux langages reconnaissables. Soient e_1 et e_2 deux expressions rationnelles telles que $L_1 = \mathcal{L}(e_1)$ et $L_2 = \mathcal{L}(e_2)$. Alors $e_1 \cdot e_2 = \mathcal{L}(e_1 \cdot e_2)$, et il est reconnu par l'automate de Glushkov de $e_1 \cdot e_2$.
  Cependant, soit a_1 et a_2 des automates déterministes reconnaissant L_1 et L_2, on peut décrire une construction permettant d'obtenir directement un automate reconnaissant $L_1 \cdot L_2$ à partir de a_1 et a_2. Le principe en gros est de brancher les états initiaux de a_2 sur les états finaux de a_1.
  Notons $(Q_1, i_1, F_1, \delta_1)$ et $(Q_1, i_2, F_2, \delta_2)$ les éléments constitutifs de a_1 et a_2. On définit alors :
  - ◇ $Q = Q_1 \cup Q_2$
  - ◇ $I = \{i_1\}$
  - ◇ $F = F_2$
  - ◇ la fonction de transition δ est obtenue en gardant les transitions de δ_1 et de δ_2, et en rajoutant de plus pour tout $f \in F_1$ et toute lettre x tel qu'il existe une transition de i_2 étiquetée par x, la transition $f \underset{x}{\longrightarrow} \delta(i_2, x)$.
  - ◇ On pose alors $a = (Q, I, F, \delta)$. C'est un automate non déterministe.

  Soit $(m_1, m_2) \in L_1 \times L_2$, lisons $m_1 m2$ dans a :
  - ◇ La lecture de m_1 aboutit à un état final f de a_1
  - ◇ Ensuite la lecture de m_2 permet d'aboutir à $\delta_1(i_2, m_2)$ (comme si on était parti de i_2 dans a_2), donc à un état final de a_2.

Ainsi, $m_1 m_2$ est reconnu.

Réciproquement, soit m reconnu par a. La lecture de m commence à i_1 qui est dans Q_1 et finit dans un état acceptant de a, donc un élément de F_2, en particulier un élément de Q_2.

- *Étoile :* pareil.

Définition 5.5. *(Expression régulière étendue)*
Les expressions régulières étendues sont définies de la même manière que les expressions régulières standard en rajoutant en plus les opérations d'intersection et de différence.

D'après ce qu'on a vu, tout langage pouvant être décrit par une expression régulière étendue peut l'être aussi par une expression régulière standard.

Par contre, l'intersection de langages locaux n'étant pas forcément un langage local (et le concept d'alphabet disjoint n'est clairement pas applicable ici, à moins de vouloir une intersection vide !) on ne peut pas appliquer l'algorithme de Glushkov à une expression régulière étendue.

Si on veut programmer la reconnaissance par automate d'une expression régulière étendue, on utilisera donc le produit d'automates déjà étudié dans la partie précédente.

6 Résumé sur les différents types de langages

- langages reconnaissables par un automate : permet une reconnaissance en une seule lecture du texte.
- langage réguliers : facile à décrire par l'utilisateur. En pratique dans un logiciel de recherche, l'utilisateur tape une expression régulière.

  Théorème de Kleene : un langage est régulier ssi il est reconnaissable par un automate. Le « seulement si » vient de l'algorithme de Glushkov, le « si » n'est pas exigible selon le programme.
- langage locaux : il est facile d'obtenir un automate reconnaissant un langage local
- langage venant d'une expression régulière linéaire : il est alors local. Il est en outre immédiat de voir si une expression régulière est linéaire.

Exercices sur les automates 2
Algorithme de Glushkov, expressions régulières, langages locaux

1 Langages locaux

Exercice 1. * Exemples de langages locaux
Soit $\Sigma = \{a, b, c\}$. Vérifier que les langages suivants sont locaux, et préciser les paramètres (P, S, F, α) correspondants :

1. ab
2. $(ab)^+$
3. a^+
4. $(abc)^*$

Exercice 2. * Retrouver le langage local à partir de ses paramètres
Déterminer le langage local défini par :

1. $P = \{a\}$, $S = \{b\}$, $F = \{ab, ba\}$, $\alpha = \top$
2. $P = \{a\}$, $S = \{b\}$, $F = \{ab, aa\}$, $\alpha = \bot$

Exercice 3. * Exemple de langage non local
Soit $\Sigma = \{a, b\}$ et $L = a^*ba$. Prouver que L n'est pas local.

Exercice 4. ** Automate local
On dit qu'un automate est local lorsque toutes les transitions étiquetées par une même lettre arrivent dans un même état.
Soit $\mathcal{A}$ un automate local. Démontrer que le langage qu'il reconnaît est local.

Exercice 5. ** Langage des facteurs d'un langage local
Soit L un langage local et L' le langage des facteurs des mots de L. Montrer que L' est local.

Exercice 6. * Une caractérisation des langages locaux**
Soit L un langage sur Σ. Montrer que L est local si et seulement si :

$$\forall (u, v, u', v') \in (\Sigma^*)^4, \forall a \in \Sigma, \ (uav \in L \wedge u'av' \in L) \Rightarrow uav' \in L.$$

Exercice 7. * Contre exemple pour l'union ou la concaténation de langages locaux
Trouver des exemples de langages locaux L_1 et L_2 tels que $L_1 \cup L_2$ n'est pas local, puis $L_1 \cdot L_2$ n'est pas local.

2 Algorithme de Glushkov

Exercice 8. * Un exemple
Déterminer l'automate de Glushkov associé à l'expression régulière $a(a + b)^*a$. Le déterminiser puis l'émonder.

Exercice 9. * CCP 2019 On veut construire l'automate de Glushkov de L décrit par $a(a + ba^*b)^*ba^*$.

1. Décrire L' , le linéarisé de L.
2. Déterminer les préfixes de L' de longueur 1, les suffixes de L' de longueur 1 et les facteurs de L' de longueur 2.
3. En déduire l'automate de Glushkov G de L.
4. Déterminiser l'automate G.

Exercice 10. * Minimisation**
Une fois obtenu l'automate de Glushkov par l'algorithme de Berry-Sethi, on voudra en général le simplifier au maximum pour que sa lecture soit la plus rapide possible.
L'algorithme présenté ici, dû à Brzozowski est un des trois algorithmes principaux pour minimiser un automate.
Notations :

- Pour tout automate $\mathcal{A}$, on notera $\mathcal{A}^T$ ($\mathcal{A}$ « transposé ») l'automate obtenu en retournant les transitions et échangeant les états initiaux et finals.
  Remarque : Même si $\mathcal{A}$ est déterministe, $\mathcal{A}^T$ ne l'est en général pas.
- Pour tout $m \in \Sigma^*$, on note m^T le mot obtenu en renversant l'ordre des lettres de m. Pour tout langage L, on note $L^T = \left\{m^T \ ; \ m \in L\right\}$. On appelle L^T le langage miroir de L.
- On dit qu'un automate est accessible lorsque tous ses états sont accessibles.
- On notera également $\mathcal{D}(\mathcal{A})$ l'automate déterminisé obtenu à partir de $\mathcal{A}$ par la méthode vu en cours (automate des parties)dans lequel on a supprimé les états non accessibles. En revanche, on ne supprime pas les états non co-accessibles, de sorte que $\mathcal{D}(\mathcal{A})$ est complet. Au final, $\mathcal{D}(\mathcal{A})$ est donc complet et accessibles.

1. Soit $\mathcal{A}$ un automate. Quel lien y-a-t-il entre $\mathcal{L}(\mathcal{A})$ et $\mathcal{L}(\mathcal{A}^T)$?
2. (résidus) Pour tout langage L et tout $u \in \Sigma^*$, on définit $u^{-1}L = \left\{ m \in L \mid \ u \cdot m \in L\right\}$.

(a) Quel lien y-a-t-il entre $u(u^{-1}L)$ et L ? Puis entre $u^{-1}(uL)$ et L ?

(b) Soit $\mathcal{A}$ un automate déterministe qui reconnaît L. Démontrer que pour tout $(u,v) \in (\Sigma^*)^2$, $\delta^*(i,u) = \delta^*(i,v) \Rightarrow u^{-1}L = v^{-1}L$.

Contraposément, on peut donc dire que lorsque deux mots u et v ont des résidus différents, ils nécessiteront deux états différents dans l'automate.

(c) En déduire que le nombre d'états de $\mathcal{A}$ est au moins égal au nombre de résidus de L.

3. Soit $\mathcal{A}$ un automate non déterministe et (Q, I, F, δ) ses composants. Alors l'automate des parties de $\mathcal{A}$ contient un puits évident. Voyez-vous lequel ?
4. (théorème de Brzozowski) Soit $\mathcal{A}$ un automate fini déterministe accessible (c'est-à-dire dont tous les états sont accessibles). On note (Q, i, δ, F) ses composants.

(a) Démontrer que $\mathcal{D}(\mathcal{A}^T)$ reconnaît le langage miroir de $\mathcal{L}(\mathcal{A})$.
On note dans la suite $\tilde{\mathcal{A}}$ cet automate. On note $L = \mathcal{L}(\mathcal{A})$, donc $\mathcal{L}\left(\tilde{\mathcal{A}}\right) = L^T$.

(b) Soient $(u,v) \in (\Sigma^*)^2$. On suppose que $u^{-1}L^T = v^{-1}L^T$. Montrer que $\delta^*_{\tilde{\mathcal{A}}}(F,u) = \delta^*_{\tilde{\mathcal{A}}}(F,v)$.

(c) Soit $\mathcal{B}$ un autre automate fini déterministe accessible reconnaissant L^T, dont on note $(Q_B, i_B, \delta_B, F_B)$ les composants. On définit alors

$$\phi : \begin{array}{ccc} Q_B & \to & Q_{\tilde{\mathcal{A}}} \\ \delta^*_B(m,F) & \mapsto & \delta^*_{\tilde{\mathcal{A}}}(m,F) \end{array}.$$

Démontrer que ϕ est bien définie et surjective.

(d) Qu'en déduire concernant le cardinal de Q_B ?

(e) Et si $\mathcal{B}$ n'est pas accessible ?

5. Déduire du théorème une construction pour obtenir un automate minimal qui reconnaît $\mathcal{L}(A)$.

3 Langages reconnaissables

Exercice 11. *!! Petit bilan

(Il s'agit essentiellement d'une question de cours.)

Soit Σ un alphabet. Démontrer que l'ensemble des langages reconnaissables est stable par :

1. union ;
2. intersection ;
3. complémentaire ;
4. étoile ;
5. concaténation.

Exercice 12. ** Le langage des mots bloquant

Soit a un automate déterministe. Montrer que le langage des mots bloquants de a est reconnaissable.

Exercice 13. **! Racine carrée

Soit L un langage reconnu par un automate fini déterministe $\mathcal{A}$. On note $\sqrt{L} = \left\{ m \in \Sigma^* \mid m^2 \in L \right\}$. Soient (Q, i, F, δ) les constituants de $\mathcal{A}$. On note enfin pour tout $(a,b) \in Q^2$, $L_{a,b}$ l'ensemble des mots m tels que $\delta^*(a,m) = b$.

1. Montrer que pour tout $(a,b) \in Q^2$, $L_{a,b}$ est un langage régulier.
2. Soit $(a,b,c) \in Q^3$ et $m \in L_{a,b} \cap L_{b,c}$. Que dire de m^2 ?
3. Montrer que $\sqrt{L}$ est régulier.

Exercice 14. ** Mots doubles

Soit Σ un alphabet. On appelle mot double tout mot dont chaque lettre apparaît au moins deux fois.

1. Montrer qu'un mot de longueur $2^{|\Sigma|}$ contient toujours un facteur double.
2. Montrer que ce nombre est optimal c'est-à-dire qu'il existe un mot de longueur $2^{|\Sigma|} - 1$ ne contenant aucun facteur double.
3. Montrer que le langage des mots doubles est reconnaissable.

Quelques indications

3 Supposons L local. On détermine alors facilement ses paramètres P, S et F.

4 Quitte à émonder $\mathcal{A}$, on peut le supposer émondé. Quitte à rajouter un état en plus et à en faire l'état initial, on peut supposer qu'aucune transition n'arrive à l'état initial (on dit alors que $\mathcal{A}$ est standard).

On peut alors reconnaître l'automate local associé à un langage local tel que défini dans le cours.

5 Il est facile de deviner les paramètres P', S', F' et α caractérisant L'.

6

11 Selon les cas, il sera plus pratique d'utiliser les automates, ou les expressions rationnelles.

12 On peut utiliser l'automate complété.

13 1. Trouver un automate qui reconnaît $L_{a,b}$.

14 1) et 2) : récurrence sur $|\Sigma|$
3) Utiliser la stabilité des langages reconnaissables par complémentaire.

Quelques solutions

1

2 1. $(ab)*$

2. a^*b

3

4

5 Soit (P, S, F, α) permettant de décrire L comme langage local. Soit $\mathcal{A}$ l'ensemble de toutes les lettres utilisées dans les mots de L.

Soit M le langage local décrit par les paramètres $(\mathcal{A}, \mathcal{A}, F, \top)$. Montrons que $M = L'$.

- L'inclusion $L' \subset M$ est claire : si $m \in L'$, soit $n \in L$ tel que m est facteur de n. Alors tous les facteurs de longueur 2 de m sont aussi des facteurs de longueur 2 de n et donc des éléments de F.
- Soit $m \in M$. Si $m = \varepsilon$ c'est un facteur d'un mot de L. Si m est une seule lettre, alors $m \in \mathcal{A}$ donc par définition de $\mathcal{A}$, m est un facteur d'un mot de L.
  Traitons le cas où m a au moins deux lettres. Notons $n = |m|$ et $x_1 \dots x_n$ les lettres de m.
  Comme $x_1 \in \mathcal{A}$, il existe un mot $u \in L$ contenant x_1. Soit u_1, u_2 tel qu"e $u = u_1 x_1 u_2$. De même, on trouve v_1, v_2 tel que $v_1 x_n v_2 \in L$.
  À présent, on constate que le mot $u_1 m v_2$ est dans L. Ce qui prouve que $m \in L'$.

6

7
- **Union :** $(ab)^* + (bc)^* + (ca)^*$ n'est pas local (sinon il contiendrait abc) alors que $(ab)^*$, $(bc)^*$, $(ca)^*$ le sont.
- **Concaténation :** aa n'est pas local (car le langage local décrit par $P = \{a\}$, $S = \{a\}$, $F = \{aa\}$, $\alpha = \bot$ est a^+) alors que a l'est.

8

10 1. On a $\mathcal{L}(\mathcal{A}^T) = \mathcal{L}(\mathcal{A})^T$. En effet, soit m un mot et $n = |m|$.
   - Supposons $m \in \mathcal{L}(\mathcal{A}^T)$. Soit c un chemin acceptant dans $\mathcal{A}^T$ étiqueté par m. Alors dans $\mathcal{A}$ on trouve le chemin c «retourné», notons-le c^T. Ce chemin est acceptant et étiqueté par m^T. Donc $m^T \in \mathcal{L}(\mathcal{A})$, puis $m \in \mathcal{L}(\mathcal{A})^T$.
   - Autre inclusion similaire.

2.
   -
   - Supposons $\delta^*(i, u) = \delta^*(i, v)$. Soit $m \in u^{-1}L$, ce qui signifie que um est reconnu par $\mathcal{A}$. Donc la lecture depuis l'état initial i de u puis de m mène à un état final. En formule, $\delta^*(\delta^*(i, u), m) \in F$.
     Maintenant, la lecture de v mène au mène état que celle de u, donc la lecture de vm mène au même état que celle de um. En formule : $\delta^*(\delta^*(i, v), m) = \delta^*(\delta^*(i, u), m) \in F$. Donc vm est reconnu.
     L'inclusion réciproque est similaire.
   - Soit ϕ la fonction de l'ensemble des états accessibles vers l'ensemble des résidus de L telle que pour tout $u \in \Sigma^*$, $\phi\left(\delta^*(i, u)\right) = u^{-1}L$. La fonction ϕ est bien définie car :
     - pour tout état accessible q, il existe $u \in \Sigma^*$ tel que $q = \delta^*(i, u)$ (par définition)
     - Si deux mots u et v mènent au même état (c'est-à-dire $\delta^*(i, u) = \delta^*(i, v)$) alors ils ont le même résidu. Ainsi, l'image par ϕ est bien unique.

     Enfin, ϕ est surjective car pour tout $u \in \Sigma^*$, le résidu $u^{-1}L$ est image par ϕ de $\delta^*(i, u)$.
     Il ne reste plus qu'à citer la propriété selon laquelle l'ensemble d'arrivée d'une fonction surjective est de cardinal inférieur à celui de son ensemble de départ.

3. $\emptyset$ est un puits dans l'automate des parties.

4.
   - Nous savons déjà que $\mathcal{D}(\mathcal{A}^T)$ reconnaît le même langage que $\mathcal{A}^T$ (théorème prouvé dans le paragraphe du cours sur la déterminisation).
     Or on vérifie de manière immédiate que ce dernier reconnaît $\mathcal{L}(\mathcal{A})^T$.
   - Soit $q \in \delta^*_{\tilde{\mathcal{A}}}(F, u)$. Cela signifie qu'il existe $q_f \in F$ tel que dans $\mathcal{A}^T$, $q_j \underset{u}{\longrightarrow} q$. Et donc dans $\mathcal{A}$, $q \underset{m^T}{\longrightarrow} q_f$.
     Comme $\mathcal{A}$ est accessible, il existe un mot $m \in \Sigma^*$ tel que, dans $\mathcal{A}$, $i \underset{m}{\longrightarrow} q \underset{u^T}{\longrightarrow} q_f$. Et dans $\mathcal{A}^T$:

     $$q_f \underset{u}{\longrightarrow} q \underset{m^T}{\longrightarrow} i.$$

     N'oublions pas que i est l'état final de $\mathcal{A}^T$. Donc $um^T \in L^T$, et donc $m^T \in u^{-1}L^T$. On utilise alors notre hypothèse : $m^T \in v^{-1}L^T$.

On remonte alors le raisonnement précédent avec v au lieu de u : $vm^T \in L^T$, puis il existe $q'_f \in F$ tel que dans $\mathcal{A}^T$, $q'_f \underset{v}{\longrightarrow}\underset{m^T}{\longrightarrow} i$. L'état intermédiaire est alors q car c'est $\delta^*(i,m)$ ($\mathcal{A}$ est déterministe).
Au final, $q'_f \underset{v}{\longrightarrow} q$ donc $q \in \delta^*_{\tilde{\mathcal{A}}}(F,v)$.

Autre inclusion similaire.

- ◇ **Bien définie :** En gros il s'agit de prouver que pour tout $q \in Q$, l'image de q existe et est unique. Ici, il faut donc montrer d'une part que tout état q peut s'écrire sous la forme $\delta^*_B(m,F)$ pour un certain $m \in \Sigma^*$, et d'autre part que si q état peut être écrit sous cette forme de deux manière différente, alors l'image $\delta^*_A(m,F)$ sera toujours la même.
  Le premier point découle du fait que B est accessible.
  Précisément : soit $q \in Q$, supposons qu'il existe $(m1,m2) \in (\Sigma^*)^2$ tel que $\delta^*_B(m1,F) = \delta^*_B(m2,F)$. Par la question 1.(a) de l'exercice, on a $m1^{-1}L = m2^{-1}L$. Et alors par la question précédente, $\delta^*_A(m1,F) = \delta^*_A(m2,F)$.
  ◇ **Surjective :** c'est le fait que $\mathcal{A}$ est accessible.
- Comme déjà employé ci-dessus, pour une fonction surjective l'ensemble d'arrivée d'une fonction surjective est de cardinal inférieur à celui de son ensemble de départ. Ici, $\mathcal{B}$ contient plus d'états que $\tilde{\mathcal{A}}$.
  Ainsi, $\tilde{\mathcal{A}}$ est, parmi les automates reconnaissant L^T, un automate ayant le plus petit nombre d'états possible.
- Si $\mathcal{B}$ n'est pas accessible, il a encore plus d'états.

5. Il suffit d'utiliser l'automate $\mathcal{D}(\mathcal{D}(\mathcal{A}^T)^T)$.
   Notons qu'on pourrait aussi utiliser $\mathcal{D}((\mathcal{A}^T)^T)$, mais dans la formule précédente, l'automate intermédiaire est minimal (parmi ceux qui reconnaissent L^T), cela va donc raccourcir les calculs.

11

12 Soit a' l'automate déterministe complet obtenu en rajoutant un puits à a. Donc tous les mots bloquant de a aboutissent dans a' au puits.

Soit alors a'' obtenu à partir de a en prenant comme unique état final le puits. L'automate a'' reconnaît les mots bloquant de a.

13

1. Soit $(a,b) \in Q^2$. Le langage $L_{a,b}$ est reconnu par l'automate obtenu à partir de $\mathcal{A}$ en prenant a comme état initial et b comme état final.
2. On constate que $m^2 \in L_{a,b}$.
3. On a, en notant i l'état initial de $\mathcal{A}$, $\sqrt{L} = \sum_{q \in Q} \sum_{f \in F} L_{i,q} \cap L_{q,f}$.

14

1.
2.
3. Soit $L = \bigcup_{x\in\Sigma} \left(\Sigma \ \{x\}\right) x \left(\Sigma \ \{x\}\right)$. C'est l'ensemble des mots m tel qu'il existe une lettre x n'apparaissant qu'une fois dans m. Autrement dit, c'est l'ensemble des mots non doubles.
   Le langage L est reconnaissable car décrit par une expression régulière.
   Alors, comme l'ensemble des langages reconnaissables est stable par complémentaire, $(\Sigma)^* \ L$ est reconnaissable aussi.