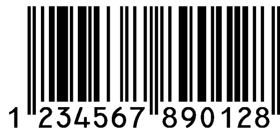
1 Langages

1 Introduction

Le but de ce chapitre est d'introduire la notion de *langage*. En informatique, un langage est un ensemble de mots, eux-mêmes constitués d'une suite de symboles, de même que la langue française est constituée d'un ensemble de mots constitués des lettres usuelles (y compris accentuées, « c cédille », « e dans l'o »...) et de quelques symboles (apostrophe, trait d'union).

Nous allons mettre en place des outils permettant de déterminer si une suite de symboles appartient bien à un langage donné ou non. Par exemple, le numéro de sécurité sociale [1] est une suite de quinze chiffres, mais toute série de quinze chiffres ne représente pas un numéro valable : seules certaines séquences peuvent exister (et parmi ces séquences possibles, seules une partie d'entre elles sont effectivement attribuées). $O(n)$

Il en est de même pour quantité d'autres codes : code IBAN, numéro SIREN, numéro SIRET, code ISBN, etc. On peut également considérer les codes-barres, qui sont des séquences [2] de zones blanches et noires, toutes les séquences n'étant pas permises (notamment, pour faciliter la lecture de ces codes, il n'est pas permis d'avoir des suites trop longues de blancs ou de noirs).



On pourra également se servir des langages pour vérifier des saisies : il est par exemple possible de définir des règles qui permettront à un ordinateur de déterminer automatiquement si ce qu'un utilisateur a entré comme adresse électronique en a bien la forme (présence d'un unique caractère `@`, d'au moins un point dans le domaine, etc.)

La théorie des langages permet aussi d'établir des codes, qui serviront à encoder (pour permettre leur transmission), compresser, chiffrer ou signer des données. Certains de ces codes sont tolérants à quelques erreurs (de transmission, de lecture, etc.) et permettent une correction automatique. C'est par exemple le cas des données sur un disque compact, de sorte qu'un erreur de lecture du laser peut être corrigée [3].

Nous mettrons par ailleurs en place des outils permettant de rechercher, dans une séquence de symboles, les mots d'un langage donné. Par exemple, pour extraire d'une page web les adresses électroniques qui y figurent, ou retrouver un passage dans un texte.

En biologie, l'ADN, l'ARN ou les protéines peuvent être associées à des séquences d'acides aminés, ou bases (au nombre de quatre pour l'ADN et pour l'ARN, vingt-deux pour les protéines). Être capable de retrouver des séquences de bases dans une séquence donnée, ou de comparer deux séquences, est un aspect majeur de cette branche de la biologie.

Enfin, la théorie des langages est au cœur des problèmes de compilation en informatique : il s'agit de pouvoir, dans un code source, identifier les mots-clés, les identifiants, etc. Puis, une fois ces éléments identifiés, vérifier que leur arrangement dans le code source respecte les règles du langage [4].

1. Ou officiellement « numéro d'inscription au répertoire des personnes physiques ».
2. Il existe des codes à deux dimensions, tels les QR codes et Datamatrix, mais on peut en tirer des séquences de 0 et de 1 en parcourant ces codes selon l'ordre prescrit.
3. C'est aussi partiellement le cas de la plupart des codes précités.
4. Ces deux étapes sont qualifiées d'analyse lexicale et d'analyse grammaticale du code source.

2 Alphabets et mots

2.1 Terminologie

Définition. On considère un ensemble fini Σ, appelé *alphabet*, de *symboles* [a].

On appelle *mot* de longueur $n \in \mathbb{N}$ toute application $[\![1\,..\,n]\!] \mapsto \Sigma$, ou de façon équivalente toute suite $s_1 s_2 \dots s_n$ de n symboles $s_i \in \Sigma$.

Le « *mot vide* », correspondant à $n = 0$, sera désigné [b] par ε.

L'ensemble (infini et dénombrable) des mots construits à partir de Σ est noté Σ^*.

On définit également $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$, l'ensemble des mots de Σ^* contenant au moins un symbole (ou ensemble des mots non-vides).

a. Égalements appelés *caractères*, *lettres* ou *lexèmes*.
b. On utilise parfois 1 ou bien encore 1_Σ.

Par exemple, à partir de l'alphabet $\Sigma = \{a, b, c\}$, on peut définir, entre autres, les mots suivants : ε, a, b, c, ba, abc, $abba$, $cabc$...

À partir de l'alphabet binaire $\Sigma = \{0, 1\}$, très utile en informatique, on peut construire les mots ε, 0, 1, 10, 0110, 10001, 100101, 0101010... Attention, 110 et 0110 désignent des mots distincts !

En biologie, les gênes sont des brins d'ADN que l'on peut modéliser comme une séquence de nucléotides, composés d'un phosphate, d'un sucre, et d'une baze azotée parmi quatre possibles : l'adénine (généralement notée A), la cytosine (C), la guanine (G) et la thymine (T). Les gènes sont donc assimilables à des mots que l'on peut construire à partir de l'alphabet $\Sigma = \{A, C, G, T\}$.

Enfin, à partir de l'alphabet[5] $\Sigma = \{\boxed{}, \boxed{.}, \boxed{\text{chat}}, \boxed{\text{est}}, \boxed{\text{le}}, \boxed{\text{mort}}, \boxed{\text{petit}}\}$, on peut construire par exemple le « mot »[6] « $\boxed{\text{mort}}\boxed{}\boxed{\text{le}}\boxed{}\boxed{\text{petit}}\boxed{}\boxed{\text{chat}}\boxed{}\boxed{\text{est}}\boxed{.}$ ».

Ce dernier exemple peut être utile lorsque l'on se penche sur les problèmes de compilation (les symboles dans un programme sont les mots-clés du langage, les identifiants, les opérateurs, les valeurs immédiates... le programme lui-même étant un mot construit à partir de ces symboles).

Pour un mot $u \in \Sigma^*$, on note $|u|$ la longueur du mot. On a naturellement $|\varepsilon| = 0$, et on peut construire $|\Sigma|^n$ mots distincts de longueur n. On note parfois Σ^n l'ensemble des mots de longueur n construits à partir de l'alphabet Σ. On a donc

$$\Sigma^* = \bigcup_{k\in\mathbb{N}} \Sigma^k \quad \text{et} \quad \Sigma^+ = \bigcup_{k\in\mathbb{N}^*} \Sigma^k$$

Les symboles peuvent être assimilés à des mots de longueur 1. Dans la suite, si `a` est un symbole de Σ, `a` sera également utilisé pour désigner le mot de Σ^* de longueur 1 constitué de ce seul symbole `a`.

Pour un mot $u \in \Sigma^*$ et un symbole $s \in \Sigma$, on note $|u|_s$ le nombre d'occurrences[7] de s dans le mot u. De façon évidente,

$$|u| = \sum_{s\in\Sigma} |u|_s$$

2.2 Concaténation

Définition. On munit l'ensemble Σ^* d'une loi de composition interne, notée[a] « $\cdot$ » et appelée *concaténation*. Pour deux mots $u, v \in \Sigma^* \times \Sigma^*$, avec $u = s_1 s_2 \dots s_n$ et $v = s'_1 s'_2 \dots s'_p$, leur concaténation $u \cdot v$ sera $u \cdot v = s_1 s_2 \dots s_n s'_1 s'_2 \dots s'_p$.

a. pour des raisons de lisibilité, comme pour la multiplication, le symbole est parfois omis lorsqu'il n'y a pas ambiguité. Un mot $s_1 s_2 \dots s_n$ peut lui-même être vu comme une concaténation de mots de longueur 1.

On a évidemment $|u \cdot v| = |u| + |v|$.

La concaténation étant une opération associative sur les mots ($(u \cdot v) \cdot w = u \cdot (v \cdot w)$), et possédant un élément neutre[8], ε ($u \cdot \varepsilon = \varepsilon \cdot u = u$), elle confère à $(\Sigma^*, \cdot)$ une structure de monoïde.

La concaténation n'est en revanche pas commutative (si par exemple $u =$ `aaba` et $v =$ `bba`, les mots $u \cdot v =$ `aababba` et $v \cdot u =$ `bbaaaba` sont distincts).

Pour tout mot $u \in \Sigma^*$, on peut définir par récurrence le mot u^n où $n \in \mathbb{N}$ par

$$u^0 = \varepsilon \quad \text{et} \quad \forall n \in \mathbb{N}^*,\ u^n = u^{n-1} \cdot u \quad \left(= u \cdot u^{n-1}\right)$$

Si $s \in \Sigma$ est un symbole, puisque l'on peut également le considérer comme un mot de longueur 1, on utilisera fréquemment la notation s^n pour désigner le mot de longueur n contenant n fois le symbole s.

2.3 Préfixes, suffixes, facteurs, sous-mots

Définition. Soient u et v deux mots de Σ^*.

v est un *préfixe* de u si et seulement si il existe un mot $w \in \Sigma^*$ tel que $u = v \cdot w$.

v est un *suffixe* de u si et seulement si il existe un mot $w \in \Sigma^*$ tel que $u = w \cdot v$.

Un préfixe ou suffixe est dit propre si $w \neq \varepsilon$.

v est un *facteur* de u si et seulement si il existe deux mots $w, w' \in \Sigma^* \times \Sigma^*$ tels que $u = w \cdot v \cdot w'$.

Un facteur est dit propre si $w \neq \varepsilon$ ou $w' \neq \varepsilon$.

Par exemple, « *rhombo* » est un préfixe, « *èdre* » un suffixe et « *dodéca* » un facteur du mot *rhombododécaèdre*[9] (pour un alphabet Σ incluant au moins minuscules et minuscules accentuées).

Précisons que les termes de préfixe et suffixe ne sont pas liés à un quelconque découpage sémantique du mot, « *rhom* », « *bodo* » et « *dre* » en sont ainsi un autre préfixe, facteur et suffixe.

De même, 0101, 1101 et 1010 sont par exemple un préfixe, facteur et suffixe de 01011010, mot construit sur l'alphabet binaire $\Sigma = \{0, 1\}$.

Si on choisit de représenter des mots en OCaml par des chaînes de caractères[10], on peut aisément vérifier si un mot `v` est un préfixe ou un suffixe de `u` en $O(|v|)$.

5. Le terme *symbole* pour désigner les éléments de Σ sera possiblement un peu moins perturbant dans une telle situation que les termes de *caractère* ou *lettre*!

6. Dans ce genre de situation, on parlera parfois de *phrase*, mais en théorie des langages, pour l'alphabet proposé, cela reste un « mot ».

7. On parle parfois de « *longueur en s* de u ».

8. C'est la raison pour laquelle ε est parfois noté 1 ou 1_Σ.

9. Également appelé dodécaèdre rhombique, ce polyèdre semi-régulier, constitué de douze faces en losange, fait partie des solides de Platon.

10. En fait, un mot n'étant qu'un ensemble ordonné de symboles, on dispose de quantité de représentations possibles. Un type `'a list` ou `'a vect` où `'a` représente le type des éléments de Σ conviendrait également, par exemple.

On écrira par exemple :

```
# let estPrefixe u v =
    String.length v <= String.length u
    && v = String.sub u 0 (String.length v);;

# let estSuffixe u v =
    String.length v <= String.length u
    && v = String.sub u (String.length u - String.length v)
                      (String.length v);;

val estPrefixe : string -> string -> bool = <fun>
val estSuffixe : string -> string -> bool = <fun>
```

Déterminer efficacement si un mot v est un facteur de u est un problème beaucoup plus difficile. On dispose d'une solution naïve évidente en $O((|u|-|v|+1)\times|v|)$, consistant à tester chacune des positions possibles pour v dans u, mais nous verrons ultérieurement qu'il existe des approches plus efficaces.

Définition. Soient $u = s_1 s_2 \dots s_n$ et $v = s'_1 s'_2 \dots s'_p$ deux mots de Σ^*.

v est un sous-mot de u si et seulement si il existe une application ϕ strictement croissante de $[\![1\,..\,p]\!]$ vers $[\![1\,..\,n]\!]$ telle que $\forall i \in [\![1\,..\,p]\!],\ s_{\phi(i)} = s'_i$.

Par exemple, « *ocre* », « *cadre* » ou « *rhbdc* » sont des sous-mots de *rhombododécaèdre*, mais « *ordre* » n'en est pas un. 0000, 1111, 011100 sont des sous-mots de 01011010 mais 0110110 n'en est pas un.

Pour des mots u et v représentés en Caml par des chaînes de caractères, il est possible de vérifier en $O(|u|)$ si le mot v est un sous-mot du mot u, grâce à une stratégie gloutonne, en écrivant par exemple [11] :

```
# let estSousMot u v =
    let phi_i = ref 0 and i = ref 0 in
      while !phi_i < String.length u && !i < String.length v do
        if u.[!phi_i] = v.[!i]
          then i := !i + 1;
        phi_i := !phi_i + 1
      done;
      !i = String.length v;;

val estSousMot : string -> string -> bool = <fun>
```

11. Attention à l'indexation qui commence à 0 !

Pour deux mots u et v de Σ^*, on peut définir le *plus long préfixe commun* $\text{plpc}(u,v)$ comme le plus long mot [12] $w \in \Sigma^*$ qui soit à la fois un préfixe de u et un préfixe de v.

De la même façon, on peut également définir un *plus long suffixe commun* $\text{plsc}(u,v)$.

On peut également s'intéresser à l'ensemble des sous-mots communs de u et v. Cet ensemble est fini (un mot de longueur n a au plus 2^n sous-mots distincts). La longueur des sous-mots communs à u et v admet donc un maximum (inférieure, de façon évidente, à $|u|$ et $|v|$).

Il n'y a cependant pas, en général, un unique sous-mot commun de taille maximale. Par exemple, si $u =$ `abca` et $v =$ `acba`, `aba` et `aca` sont deux sous-mots communs de u et v, et il n'existe, de façon évidente, aucun sous-mot commun plus long.

Il est possible de déterminer, avec une complexité temporelle en $O(|u|\,|v|)$, la longueur du plus long sous-mot commun à $u = u_1 u_2 \dots u_n$ et $v = v_1 v_2 \dots v_p$. Pour ce faire, on définit $c(i,j)$ sur $[\![0\,..\,n]\!] \times [\![0\,..\,p]\!]$ comme la longueur du plus long sous-mot commun entre le préfixe de longueur i de u et le préfixe de longueur j de v.

Ces $c(i,j)$ peut être déterminés par récurrence grâce aux relations :

- $c(i,0) = 0$;
- $c(0,j) = 0$;
- pour $i > 0$ et $j > 0$, $c(i,j) = \begin{cases} 1 + c(i-1,j-1) & \text{si } u_i = v_j\,; \\ c(i,j) = \max(c(i-1,j), c(i,j-1)) & \text{sinon.} \end{cases}$

La longueur du plus long sous-mot commun est alors $c(n,p)$.

Par exemple, pour les mots `abacacb` et `bacbcab`, les $c(i,j)$ correspondraient au tableau suivant :

		b	a	c	b	c	a	b
	0	0	0	0	0	0	0	0
a	**0**	0	1	1	1	1	1	1
b	0	**1**	1	1	2	2	2	2
a	0	1	**2**	2	2	2	3	3
c	0	1	2	**3**	3	3	3	3
a	0	1	2	3	**3**	**3**	4	4
c	0	1	2	3	3	4	**4**	4
b	0	1	2	3	4	4	4	**5**

Ici, $c(n,p) = 5$, soit la longueur des plus long sous-mots communs, `bacab` et `baccb`.

Il est possible de déterminer $c(n,p)$ avec un besoin en terme de mémoire en $O(\min(|u|,|v|))$ (en construisant ligne par ligne le tableau précédent, et en ne conservant qu'une unique ligne).

12. Il est nécessairement unique.

Cette stratégie algorithmique pour le calcul efficace des $c(n,p)$ porte le nom de *programmation dynamique*. Elle sera étudiée dans le cours de tronc commun, mais nous aurons l'occasion de croiser de nombreux exemples dans ce cours. On peut par exemple écrire :

```
# let rec lplsmc u v =
    let n = String.length u
    and p = String.length v in

    if n < p then lplsmc v u  (* si |u| < |v|, on permute u et v pour
                                         économiser un peu de mémoire *)

    else let tab = Array.make (p+1) 0 in (* tab = une ligne du tableau,
                                             initialement la première *)

          for i=1 to n do         (* On détermine les autres  lignes *)
            let m = ref 0 in
              for j=1 to p do     (* On calcule chacun des c(i,j)    *)

                (* invariant de boucle : m désigne c(i-1,j-1)
                                          tab.(k) désigne c(i,k) si k<j,
                                                  et c(i-1,k) si k>=j *)
                let n = if u.[i-1] = v.[j-1]
                          then 1 + !m
                          else max tab.(j-1) tab.(j)
                in m := tab.(j);
                   tab.(j) <- n

              done;
          done;
          tab.(p);;            (* On retourne le dernier coefficient *)

lplsmc : string -> string -> int = <fun>
```

Pour déterminer un sous-mot de longueur maximale, il suffit de reconstruire [13] le « chemin » ayant déterminé la valeur $c(n,p)$. Pour notre exemple, la série de règles ayant amené au calcul de $c(n,p)$ a été représentée en gras sur le tableau ci-dessus. Elle fournit le mot *baccb* comme exemple de sous-mot commun de longueur maximale. Il peut exister plusieurs parcours menant à $c(n,p)$, en fonction des règles que l'on privilégie [14], conduisant à des sous-mots communs différents, mais de même longueur.

13. Il est possible de conserver un coût linéaire en temps, mais le coût en espace sera plus important, en $O(|u|\times|v|)$.

14. Ici, si par exemple $u_i \neq v_j$ et $c(i-1,j) = c(i,j-1)$, on a considéré que la valeur $c(i,j)$ a été obtenue à partir de $c(i,j-1)$.

2.4 Quelques résultats combinatoires

Lemme 1 (de Levi). *Soient u, u', v et v' quatres mots de Σ^* tels que $u\cdot u' = v\cdot v'$.*

Il existe un unique mot w tel que une (et une seule si $u\neq v$) des deux propositions suivantes soit vérifiée :

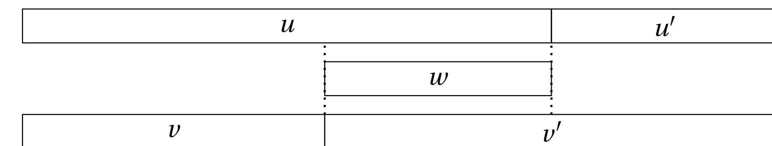
- $u = v\cdot w$ et $v' = w\cdot u'$
- $v = u\cdot w$ et $u' = w\cdot v'$

Démonstration. v est un préfixe de $v\cdot v'$, donc de $u\cdot u'$.

Supposons $|u| \geqslant |v|$. v est donc un préfixe de u, et il existe un mot w tel que $u = v\cdot w$.

Par conséquent, $u\cdot u' = (v\cdot w)\cdot u' = v\cdot(w\cdot u')$. On a donc $v' = w\cdot u'$.

Cela se résume bien sur un schéma :



Le cas $|u| \leqslant |v|$ se traite de façon similaire et conduit à la réalisation de la seconde proposition énoncée dans le lemme de Levi. □

Théorème 1. *Soient x, y et z trois mots de Σ^* vérifiant $x\cdot y = y\cdot z$ et $x \neq \varepsilon$.*

Il existe deux mots u et v de Σ^* et un entier $k\in\mathbb{N}$ tels que

$$x = u\cdot v \qquad y = (u\cdot v)^k\cdot u = u\cdot(v\cdot u)^k \qquad z = v\cdot u$$

Démonstration. Si $|x| > |y|$, alors d'après le lemme de Levi, il existe un mot $w\in\Sigma^*$ tel que $x = y\cdot w$ et $z = w\cdot y$.

On a donc $x\cdot y = y\cdot z = y\cdot w\cdot y$. il suffit donc de prendre $u = y$, $v = w$ et $k = 0$.

Si, en revanche, $|x| \leqslant |y|$, on raisonne par récurrence, en fonction de la longueur $|y|$ (avec $|y| \geqslant 1$, puisque $|y| \geqslant |x|$ et $x\neq\varepsilon$) :

- si $|y| = 1$, alors $|x| = 1$, et donc $x = y$ et $y = z$. Il suffit de choisir $u = x = y = z$, $v = \varepsilon$ et $k = 0$;
- supposons le résultat vrai pour tout $|y| < n$ et considérons le cas $|y| = n$: d'après le lemme de Levi, il existe $w\in\Sigma^*$ tel que $y = x\cdot w = w\cdot z$; on a alors $x\cdot w = w\cdot z$, avec $|w| < |y|$ (puisque $|x| > 0$), ce qui permet d'affirmer qu'il existe u, v et k tels que $x = u\cdot v$, $w = (u\cdot v)^k\cdot u$ et $z = v\cdot u$; et enfin, $y = x\cdot w = u\cdot v\cdot(u\cdot v)^k\cdot u = (u\cdot v)^{k+1}\cdot u$.

La décomposition proposée existe donc bien quelle que soit $|y|$. □

Théorème 2. *Soient x et y, deux mots de Σ^+ tels que $x \cdot y = y \cdot x$.*

Il existe $w \in \Sigma^+$ et $i, j \in \mathbb{N}^{*2}$ tels que $x = w^i$ et $y = w^j$.

Démonstration. On raisonne par récurrence sur $n = |x \cdot y|$ (avec $|x \cdot y| \geqslant 2$ puisque, d'après les hypothèses, $x \neq \varepsilon$ et $y \neq \varepsilon$) :

- si $|x \cdot y| = 2$, $|x| = |y| = 1$ donc $x = y$; il suffit de choisir $w = x = y$ et $i = j = 1$;
- pour $n > 2$, supposons la propriété vérifiée pour tout x et y vérifiant les hypothèses et satisfaisant à la condition $|x \cdot y| < n$, et considérons deux mots x et y vérifiant les hypothèses, et tels que $|x \cdot y| = n$; d'après le théorème précédent, il existe u et v dans Σ^* et un entier $k \in \mathbb{N}$ tels que $x = u \cdot v = v \cdot u$ et $y = (u \cdot v)^k \cdot u$
  - si $u = \varepsilon$, $x = v$ et $y = v^k$, donc $w = v$, $i = 1$ et $j = k$ conviennent ;
  - si $v = \varepsilon$, $x = u$ et $y = u^{k+1}$, donc $w = u$, $i = 1$ et $j = k + 1$ conviennent ;
  - sinon, on a $u \cdot v = v \cdot u$ avec $|u \cdot v| = |x| < |x \cdot y| = n$, l'hypothèse de récurrence permet alors d'affirmer qu'il existe w, i et j tels que $u = w^i$ et $v = w^j$, et donc $x = w^{i+j}$ et $y = w^{k(i+j)+i}$. □

2.5 Un peu d'ordre

Il peut être utile de munir l'ensemble Σ^* d'une relation d'ordre (par exemple pour prouver la terminaison d'un algorithme travaillant sur des mots, l'ordre devant pour ce faire être bien fondé). Il est en fait possible de construire de nombreuses relations d'ordres intéressantes sur un ensemble de mots Σ^*.

Tout d'abord, on peut considérer l'ordre « sous-mot » :

Définition. Soient deux mots u et v de Σ^*.

On a $v \preceq u$ pour l'ordre sous-mot $\preceq$ si et seulement si v est un sous-mot de u.

Par exemple, on a ocre $\preceq$ rhombododécaèdre.

C'est bien une relation d'ordre, car elle est réflexive ($\forall u \in \Sigma^*,\ u \preceq u$), antisymétrique (si $u \preceq v$ et $v \preceq u$ alors $u = v$) et transitive (si $u \preceq v$ et $v \preceq w$ alors $u \preceq w$). Par contre, ce n'est généralement pas [15] un ordre total sur Σ^*.

Théorème 3. *L'ordre sous-mot $\preceq$ est un ordre bien fondé sur Σ^*.*

Démonstration. Pour justifier que l'ordre sous-mot $\preceq$ est bien fondé, on peut remarquer que si $v \prec u$, alors $|v| < |u|$.

Supposons en effet que $v \prec u$. La fonction ϕ (définie précédemment pour caractériser un sous-mot) est strictement croissante de $[\![1\,..\,p]\!]$ dans $[\![1\,..\,n]\!]$ (avec $p = |v|$ et $n = |u|$), donc $\forall i \in [\![1\,..\,p]\!]$, $\phi(i) \geqslant i$ par une récursion immédiate. Par conséquent, $\phi(p) \geqslant p$.

Or, $\phi(p) \leqslant n$, donc $p \leqslant n$, d'où $|v| \leqslant |u|$.

Maintenant, si $|v| = |u|$, alors la fonction ϕ ne peut être que la fonction identité, et par conséquent $u = v$, ce qui est incompatible avec l'hypothèse $v \prec u$. On a donc bien $|v| < |u|$.

Toute suite de mots (u_i) de Σ^* strictement décroissante pour l'ordre sous-mot $\preceq$ contient donc des mots de longueur strictement décroissante. Cette suite est donc nécessairement finie, puisque $\mathbb{N}$ est bien fondé. □

15. Excepté le cas particulier où $|\Sigma| \leqslant 1$.

L'ordre sous-mot a en fait de nombreuses autres propriétés intéressantes, son principal inconvénient, pour certaines applications, est de ne pas être un ordre total.

Si l'alphabet Σ dispose d'un ordre total $\leqslant$, un autre ordre couramment utilisé est l'ordre *lexicographique*.

Définition. Soient un alphabet Σ muni d'un ordre total $\leqslant$, et deux mots u et v de Σ^*.

On a $v \prec_L u$ pour l'ordre lexicographique $\preceq_L$ si et seulement si v est un préfixe propre de u ou s'il existe des mots $w \in \Sigma^*$, $w' \in \Sigma^*$ et $w'' \in \Sigma^*$ et des symboles $s, s' \in \Sigma^2$ avec $s' < s$ tels que $u = w \cdot s \cdot w'$ et $u = w \cdot s' \cdot w''$.

L'ordre lexicographique $\preceq_L$ est un ordre total sur Σ^*. Il n'est cependant pas toujours le plus pratique à utiliser en théorie des langages, car ce n'est pas un ordre bien fondé dès que l'alphabet Σ contient au moins deux symboles. Par exemple, si $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ avec $\mathtt{a} < \mathtt{b}$, la suite de mots $(u_n)_{n \in \mathbb{N}} = \mathtt{a}^n\mathtt{b}$ est infinie tout en étant strictement décroissante. On peut lui préférer l'ordre *hiérarchique* :

Définition. Soient Σ^* muni d'un ordre lexicographique $\preceq_L$, et deux mots u et v de Σ^*.

On a $v \preceq_H u$ pour l'ordre hiérarchique $\preceq_H$ si et seulement si $|v| < |u|$ ou $|v| = |u|$ et $v \preceq_L u$.

Théorème 4. *L'ordre hiérarchique $\preceq_H$ est un ordre total bien fondé sur Σ^*.*

Démonstration. Le caractère bien fondé de l'ordre hiérarchique est évident si l'on considère que les mots de Σ^* correspondent exactement aux entiers naturels dont l'écriture en base $|\Sigma| + 1$ ne comportent pas le chiffre zéro : il suffit d'associer aux symboles de Σ, pris par ordre croissant, les chiffres 1, 2, ..., $|\Sigma|$.

Si à deux mots u et v vérifiant $u \prec_H v$ on associe de la sorte deux entiers p et q, on aura $p < q$. Une suite strictement décroissante dans Σ^* pour l'ordre hiérarchique correspond donc à une suite strictement décroissante sur $\mathbb{N}$ pour l'ordre usuel $\leqslant$. Elle ne saurait donc être infinie ! □

2.6 Distances entre mots

De la même façon qu'il existe de nombreux ordres sur les mots, on peut construire quantité de distances entre mots.

La distance la plus simple que l'on puisse définir entre deux mots de même longueur est la *distance de Hamming*, déterminant le nombre de symboles qui diffèrent entre deux mots :

Définition. Soient Σ un alphabet et $u = u_1 u_2 \dots u_n$, $v = v_1 v_2 \dots v_n$ deux mots de Σ^* tels que $|u| = |v|$.

La *distance de Hamming* $d_\mathrm{H}(u, v)$ entre u et v est définie par

$$d_\mathrm{H}(u,v) = \sum_{i=1}^{n} \mathbb{1}_{u_i \neq v_i}$$

où $\mathbb{1}_{u_i \neq v_i}$ est la fonction indicatrice de $u_i \neq v_i$ (elle vaut 1 si $u_i \neq v_i$ et 0 sinon).

C'est bien une distance au sens des mathématiques :

Démonstration. On vérifie les trois propriétés des distances.

- $d_\mathrm{H}(u, v) \geqslant 0$ puisque l'on somme des entiers positifs (0 ou 1).
- $d_\mathrm{H}(u, v) = 0 \iff u = v$ car une distance nulle signifie que $\forall i \in [\![1\,..\,n]\!]$, $u_i = v_i$.
- $d_\mathrm{H}(u, w) \leqslant d_\mathrm{H}(u, v) + d_\mathrm{P}(v, w)$ car $\mathbb{1}_{u_i \neq w_i} \leqslant \mathbb{1}_{u_i \neq v_i} + \mathbb{1}_{v_i \neq w_i}$. □

Nous verrons que cette distance peut être utile dans le cadre de codages autocorrecteurs. Elle n'est pourtant pas toujours pertinente, et présente l'inconvénient de ne pouvoir prendre en compte des mots de longueur différente.

On peut également définir des distances à partir des notions de préfixe, suffixe, facteur et sous-mot. Par exemple, on peut définir la distance préfixe entre deux mots :

Définition. Soient Σ un alphabet et u, v deux mots de Σ^*.

La *distance préfixe* $d_\mathrm{P}(u, v)$ entre u et v est définie par

$$d_\mathrm{P}(u, v) = |u| + |v| - 2\,|w| \quad \text{où } w \text{ est le plus long préfixe commun à } u \text{ et } v.$$

Là encore, la distance préfixe est bien une distance au sens mathématique :

Démonstration. On vérifie à nouveau les trois propriétés requises.

- $d_\mathrm{P}(u, v) \geqslant 0$ puisque $|w| \leqslant \min(|u|, |v|)$.
- $d_\mathrm{P}(u, v) = 0 \iff u = v$.
  En effet, $|w| \leqslant \min(|u|, |v|)$, donc si $d_\mathrm{P}(u, v) = 0$, on a $|w| = |u| = |v|$, ce qui ne peut arriver que si $u = v$.
  Réciproquement, si $u = v$, on a $w = u = v$, et donc $|u| + |v| - 2\,|w| = 0$.
- $d_\mathrm{P}(u, w) \leqslant d_\mathrm{P}(u, v) + d_\mathrm{P}(v, w)$.
  Pour cette inégalité, posons x le plus long préfixe commun de u et v, y celui de v et w et z celui de u et w. Il faut donc montrer que $|u| + |w| - 2\,|z| \leqslant |u| + |v| - 2\,|x| + |v| + |w| - 2\,|y|$, soit $|z| + |v| \geqslant |x| + |y|$;
  Supposons $|x| \geqslant |y|$. x est un préfixe de v, donc $|v| \geqslant |x|$. Par ailleurs, y est un préfixe de x, donc de u. Puisque y est également un préfixe de w, c'est un préfixe de z, donc $|z| \geqslant |y|$. Il reste simplement à sommer les inégalités.
  Si $|x| \leqslant |y|$, le raisonnement est similaire. □

Plus intéressante, la distance sous-mot est définie par :

Définition. Soient Σ un alphabet et u, v deux mots de Σ^*.

La *distance sous-mot* $d_\mathrm{S}(u, v)$ entre u et v est définie par

$$d_\mathrm{S}(u, v) = |u| + |v| - 2\pi(u, v)$$

où $\pi(u, v)$ est la [a] longueur du plus long sous-mot commun à u et v.

a. s'il peut exister plusieurs sous-mots communs de longueur maximale, la longueur est bien définie.

Démonstration. Cette fois encore, c'est l'inégalité triangulaire qui est le point délicat.

En développant cette inégalité triangulaire, on montre qu'il nous faut cette fois vérifier que $\pi(u, w) + |v| \geqslant \pi(u, v) + \pi(v, w)$.

Considérons un sous-mot commun à u et v de longueur $\pi(u, v)$, et notons I l'ensemble des positions des symboles dans v correspondant à ce sous-mot.

De même, notons J l'ensemble des positions des symboles dans v correspondant à un sous-mot commun à v et w de longueur $\pi(v, w)$.

On a $\pi(u, v) + \pi(v, w) = |\mathrm{I}| + |\mathrm{J}| = |\mathrm{I} \cup \mathrm{J}| + |\mathrm{I} \cap \mathrm{J}|$.

On a $|\mathrm{I} \cup \mathrm{J}| \leqslant |v|$, et le sous-mot de v construit à partir des indices de $\mathrm{I} \cap \mathrm{J}$ est également un sous-mot de u et w, donc $|\mathrm{I} \cap \mathrm{J}| \leqslant \pi(u, w)$. La somme des deux inégalités est l'inégalité recherchée. □

Par exemple, entre les mots $u =$ `abacacb` et $v =$ `bacbcab`, on a une distance sous-mot de $|u| + |v| - 2\pi(u, v) = 7 + 7 - 2 \times 5 = 4$ puisque les plus longs sous-mots communs ont une longueur égale à 5.

La distance de Hamming determine, en quelque sorte, le nombre de symboles qu'il faut changer pour aller d'un mot u à un mot v de même longueur. La distance sous-mot représente quant à elle le nombre de caractères qu'il faut retirer et ajouter pour aller d'un mot u à un mot v.

Par exemple, `abacacb` ▷ `bacacb` ▷ `baccb` ▷ `bacbcb` ▷ `bacbcab`.

En permettant à la fois les substitutions et les ajouts/suppressions de symboles, on obtient la *distance d'édition* ou *distance de Levenshtein*.

Définition. Soient Σ un alphabet et u, v deux mots de Σ^*

La *distance de Levenshtein* $d(u,v)$ entre u et v correspond au nombre minimal d'opérations à effectuer pour transformer u en v, ces opérations étant :
- la suppression d'un symbole ;
- l'ajout d'un symbole ;
- la substitution d'un symbole en un autre.

La distance de Levenshtien entre les mots `abacacb` et `bacbcab` est 3, car on peut passer de l'un à l'autre avec la suppression du premier `a`, une substitution d'un `a` en `b`, et un ajout d'un `a` : `abacacb` ▷ `bacacb` ▷ `bacbcb` ▷ `bacbcab`, et il n'existe pas de solution utilisant moins d'opérations.

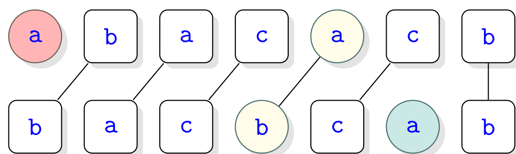
Cette fois encore, on a bien défini une distance au sens des mathématiques :

Démonstration. Seule l'inégalité triangulaire n'est pas totalement immédiate. Cependant, il est aisé de voir que si l'on peut passer de u à v en $d(u,v)$ opérations et de v à w en $d(v,w)$ opérations, il est possible de passer de u à w en $d(u,v)+d(v,w)$ opérations (et possiblement moins).

Donc on a bien $d(u,w) \leqslant d(u,v)+d(v,w)$. □

Pour déterminer cette distance, on peut remarquer les propriétés suivantes :
- si les derniers symboles des mots u et v sont identiques, la distance entre u et v ne peut pas être plus grande que celle entre les mots privés de leur dernier symbole ;
- si les derniers symboles des mots u et v sont différents, la distance entre u et v ne peut pas être plus grande que celle entre les mots privés de leur dernier symbole augmentée de 1 (substitution) ;
- la distance entre u et v ne peut excéder la distance entre u privé de son dernier symbole et v augmentée de 1 (ajout) ;
- la distance entre u et v ne peut excéder la distance entre u et v privé de son dernier symbole augmentée de 1 (suppression).

Par ailleurs, la distance entre u et v correspond nécessairement à l'un des cas précédents. En effet, on peut remarquer que quelle que soit la suite de transformations amenant le mot u au mot v, il est possible d'ordonner les opérations de la gauche vers la droite. Pour ce faire, on identifie les symboles qui se « correspondent » (ceux qui seront conservés ou substitués) comme sur l'exemple ci-dessous :



Dans cet exemple de transformation de `abacacb` en `bacbcab`, la suite d'opérations peut donc être ordonnée de la sorte :
- suppression du premier `a` ;
- substitution d'un `b` en `a` ;
- ajout d'un `a`.

Une fois cet ordonnancement fait (qui ne change pas le nombre d'opérations dans la transformation), on peut aisément voir que la dernière opération est bien nécessairement l'une des quatre opérations décrites ci-dessus [16].

Pour déterminer la distance entre u et v de longueur respectives n et p, on va donc simplement définir une grandeur $c : [\![0\,..\,|u|]\!] \times [\![0\,..\,|v|]\!] \mapsto \mathbb{N}$, distance entre le préfixe de longueur i de u et celui de longueur j de v, définie par récurrence de la sorte :
- $c(i,0)=i$ et $c(0,j)=j$;
- pour $i>0$ et $j>0$, $c(i,j)=\min\Big(c(i-1,j)+1, c(i,j-1)+1, c(i-1,j-1)+\mathbb{1}_{u_i \neq v_j}\Big)$.

On a alors $d(u,v)=c(|u|,|v|)$.

Dans notre exemple, le tableau des $c(i,j)$ ressemblera à :

		a	b	a	c	a	c	b
	0	**1**	2	3	4	5	6	7
b	1	1	**1**	2	3	4	5	6
a	2	1	2	**1**	2	3	4	5
c	3	2	2	2	**1**	2	3	4
b	4	3	2	3	2	**2**	3	3
c	5	4	3	3	3	3	**2**	3
a	6	5	4	3	4	3	**3**	3
b	7	6	5	4	4	4	4	**3**

Comme la longueur du plus long sous-mot commun, la distance de Levenshtein entre deux mots peut donc être calculée en un temps $O(|u||v|)$ avec un coût spatial en $O(\min(|u|,|v|))$ grâce à la programmation dynamique, et il est possible de déterminer la suite des opérations à effectuer en reconstruisant un chemin menant de la case en haut à droite à celle en bas à gauche respectant les opérations effectuées par l'algorithme.

Il existe de nombreuses variantes de cette distance, car elle joue notamment un rôle important en terme de correction des fautes de frappe (si un mot entré par un utilisateur est incorrect, on le remplace par le mot correct le plus proche vis-à-vis de la distance de Levenshtein). La distance de Damerau-Levenshtein permet par exemple la transposition de deux symboles successifs (`abca` ↷ `acba`).

Par ailleurs, plutôt que de compter simplement le nombre de transformations, on peut leur attribuer des « coûts » différents, et sommer ces coûts. L'idée étant que, connaissant la disposition des touches sur un clavier, certaines substitutions sont plus probables que

16. Sous réserve que l'on évite les séquences d'opérations qui ne font qu'augmenter la distance, comme la suppression d'un caractère suivi d'un ajout, que l'on peut remplacer avantageusement par une substitution.

d'autres. Par exemple, *sac*, *sec*, *sic*, *soc* et *suc* sont cinq corrections possibles de *sqc*, mais sur un clavier azerty ou qwerty, la touche *a* étant plus proche de la touche *q* que les touches *i*, *o* et *u*, la première de ces corrections est probablement la bonne.

Il faut cependant prendre garde au fait que, selon les coûts que l'on choisit, on peut perdre le caractère de « distance » au sens des mathématiques, l'inégalité triangulaire pouvant ne plus être respectée. Cela étant dit, dans un cadre pratique, cela n'a souvent guère d'importance.

3 Langages

3.1 Définir un langage

Définition. Soit Σ un alphabet.

On appelle *langage* toute partie, finie ou infinie, de Σ^*. En d'autres termes, un langage correspond donc à un ensemble de mots.

Il existe de nombreuses façons de définir un langage. La première solution consiste à énumérer les mots du langage. Par exemple, le langage $L = \left\{ \mathtt{a}^p\mathtt{b}\mathtt{a}^q \;\middle|\; p,q \in \mathbb{N}^2 \right\}$ est le langage des mots contenant des `a` et un unique `b`. Le langage des palindrômes sur un alphabet Σ correspond à $\left\{ w = s_1 s_2 \ldots s_n\, s\, s_n \ldots s_2 s_1 \;\middle|\; (s_1, s_2, \ldots, s_n) \in \Sigma^n \text{ et } s \in \Sigma \cup \{\varepsilon\} \right\}$.

On peut également définir les mots d'un langage par une propriété. Le langage L précédent peut être défini par $L = \left\{ w \in \{\mathtt{a},\mathtt{b}\}^* \;\middle|\; |w|_{\mathtt{b}} = 1 \right\}$. De la même façon, le langage des palindrômes sur un alphabet Σ est $\left\{ w = s_1 s_2 \ldots s_n \in \Sigma^* \;\middle|\; \forall i \in [\![1\,..\,\lfloor n/2 \rfloor]\!],\; s_i = s_{n+1-i} \right\}$.

On peut enfin les définir à partir d'une *grammaire formelle*. Pour notre premier exemple, on peut écrire que $\mathtt{b} \in L$ et $w \in L \Longrightarrow \mathtt{a} \cdot w \in L$ et $w \cdot \mathtt{a} \in L$. Pour le langage des palindrômes sur Σ, on écrira $\varepsilon \in L$, $\Sigma \subset L$ et $w, s \in L \times \Sigma \Longrightarrow s \cdot w \cdot s \in L$.

3.2 Langages récursifs

Si l'ensemble Σ^* est dénombrable (les mots sont énumérables par ordre hiérarchique croissant), l'ensemble des langages n'est en revanche pas dénombrable. Ils ne peuvent pas tous être décrits ou manipulés aisément. On distingue plusieurs types de langages :

Définition. Un langage L est dit *récursivement énumérable* s'il existe un algorithme permettant d'énumérer tous les mots de L.

Un langage L est dit *récursif* s'il existe un algorithme qui, étant donné un mot w, retourne `true` si $w \in L$ et `false` sinon.

Notons que le terme « récursif » est à prendre ici au sens des mathématiques, c'est à dire de calculabilité à partir d'un ensemble de règles, et non dans le sens qu'on lui prête souvent en informatique.

Un langage récursif L est nécessairement récursivement énumérable, puisqu'il suffit d'énumerer les mots de Σ^* (dénombrable) et de vérifier leur appartenance à L. La réciproque n'est en revanche pas vraie (même si l'on peut énumérer les mots de L, pour un mot w qui n'appartiendrait pas à L, on ne peut retourner `false` avant d'avant d'avoir terminé cette énumération, ce qui n'arrivera pas pour des langages avec une infinité de mots).

Dans la suite, on s'intéressera aux langages récursivement énumérables, plus intéressants car ils nous permettent de distinguer les mots de L et de $\Sigma^* \setminus L$.

3.3 Opérations ensemblistes sur les langages

Les langages étant des parties de Σ^*, il est possible d'utiliser les opérations ensemblistes usuelles : union, intersection, différence, complémentation...

Par exemple, si L_1 et L_2 sont deux langages sur un même alphabet [17] Σ, on peut définir l'union $L_1 + L_2$ de ces langages [18] par

$$L_1 + L_2 = \left\{ w \in \Sigma^* \;\middle|\; w \in L_1 \text{ ou } w \in L_2 \right\}$$

De même, on peut définir l'intersection, la différence et la complémentation par

$$L_1 \cap L_2 = \left\{ w \in \Sigma^* \;\middle|\; w \in L_1 \text{ et } w \in L_2 \right\}$$

$$L_1 \setminus L_2 = \left\{ w \in \Sigma^* \;\middle|\; w \in L_1 \text{ et } w \notin L_2 \right\}$$

$$\complement_\Sigma L = \left\{ w \in \Sigma^* \;\middle|\; w \notin L \right\}$$

3.4 Opérations sur les langages dérivées de la concaténation

La concaténation offre une autre solution pour combiner des langages. Ainsi, pour deux langages L_1 et L_2 sur un même alphabet [19] Σ, on définit leur concaténation L_1L_2 par

$$L_1L_2 = \left\{ w \in \Sigma^* \;\middle|\; \exists u, v \in L_1 \times L_2,\; w = u \cdot v \right\}$$

La concaténation étant une opération associative, on peut de la même façon exprimer les « puissances » L^n d'un langage ($n \in \mathbb{N}$) par

$$L^0 = \{\varepsilon\} \quad \text{et} \quad \forall n \in \mathbb{N}^*,\, L^n = L^{n-1}L \quad \left(= LL^{n-1}\right)$$

17. Si ce sont des langages sur des alphabets différents Σ_1 et Σ_2, il reste possible d'en définir l'union, qui sera un langage sur l'alphabet $\Sigma_1 \cup \Sigma_2$.
18. L'usage privilégie généralement le symbole « + » pour désigner l'union de deux langages, de préférence au symbole « ∪ » rencontré dans la théorie des ensembles.
19. Même remarque que précédemment.

On prendra garde à éviter les confusions entre $L^2 = \{u \cdot v \mid u, v \in L \times L\}$ et le langage des « carrés » de L, plus restreint, $\{u \cdot u \mid u \in L\}$!

La *fermeture de Kleene* L^* d'un langage L est la cloture de ce langage pour la concaténation, c'est-à-dire que

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

On parle parfois simplement d'« étoile » du langage L. Il s'agit de l'ensemble des mots qui peuvent se construire par une concaténation d'un nombre fini de mots de L (y compris zéro, aussi ε est toujours un mot de L^*).

À nouveau, on retrouve une structure de monoïde dans $(L^*, \cdot)$. En effet, la concaténation est bien une loi de composition interne ($\forall u, v \in L^* \times L^*,\ u \cdot v \in L^*$) associative possèdant un élément neutre $\varepsilon \in L^*$.

On qualifie de *sous-monoide* de Σ^* tout langage qui contient le mot ε et qui est stable pour l'opération de concaténation. L^* est le plus petit sous-monoïde de Σ^* contenant L. Inversement, tout sous-monoïde M de Σ^* est de la forme P^* puisque[20] $M = M^*$.

Il est intéressant de noter que les notations pour les puissances et l'étoile d'un langage sont cohérentes avec celles déjà utilisées précédemment : par exemple, l'ensemble de tous les mots est bien l'étoile de son alphabet[21].

On peut également définir L^+ comme on l'a fait pour Σ^+ par

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

De même, on peut définir la « *racine carrée* » d'un langage L, regroupant les mots de Σ^* dont le « carré » est dans L :

$$\sqrt{L} = \{w \in \Sigma^* \mid w \cdot w \in L\}$$

Attention, si l'on a bien $L \subset \sqrt{L^2}$ ($\forall u \in L,\ u^2 \in L^2$ et donc $u \in \sqrt{L^2}$), $\sqrt{L^2}$ est un sous-ensemble de Σ^*, mais n'est pas nécessairement un sous-ensemble de L. Par exemple, si $L = \{\varepsilon, \mathtt{aa}, \mathtt{b}\}$, on a

$$L^2 = \{\varepsilon, \mathtt{aa}, \mathtt{b}, \mathtt{aaaa}, \mathtt{aab}, \mathtt{baa}, \mathtt{bb}\}$$

$$\sqrt{L^2} = \{\varepsilon, \mathtt{a}, \mathtt{aa}, \mathtt{b}\} \supset L$$

20. Il peut exister $P \subsetneq M$ tel que $M = P^*$.
21. Et inversement, on peut voir L^* comme l'ensemble des mots construit à partir de l'alphabet L

3.5 Langages quotients

Pour un langage L sur un alphabet Σ et un mot $u \in \Sigma^*$, on appelle *langage quotient à gauche*[22] de L par rapport à u, et on note $u^{-1}L$, l'ensemble des mots de $w \in \Sigma^*$ tels que uw soit un mot de L :

$$u^{-1}L = \{w \in \Sigma^* \mid u \cdot w \in L\}$$

Attention, la notation peut aisément prêter à confusion, u^{-1} n'a pas de sens !

Cette définition se généralise à un quotient par un langage M :

$$M^{-1}L = \{w \in \Sigma^* \mid \exists u \in M,\ u \cdot w \in L\} = \cup_{u \in M} u^{-1}L$$

Là encore, soulignons que la notation M^{-1} n'a pas de sens seule !

On a $L \subset M^{-1}(ML)$, mais aucune relation particulière entre $M(M^{-1}L)$ et L.

Par exemple, si $M = \{\mathtt{a}, \mathtt{aa}, \mathtt{b}\}$ et $L = \{\mathtt{ac}, \mathtt{add}, \mathtt{cd}\}$:

$$ML = \{\mathtt{aaac}, \mathtt{aaadd}, \mathtt{aac}, \mathtt{aacd}, \mathtt{aadd}, \mathtt{acd}, \mathtt{bac}, \mathtt{badd}, \mathtt{bcd}\}$$

$$M^{-1}L = \{\mathtt{c}, \mathtt{dd}\}$$

$$M^{-1}(ML) = \{\mathtt{aac}, \mathtt{aadd}, \mathtt{ac}, \mathtt{acd}, \mathtt{add}, \mathtt{c}, \mathtt{cd}, \mathtt{dd}\} \supset L$$

$$M(M^{-1}L) = \{\mathtt{ac}, \mathtt{add}, \mathtt{bc}, \mathtt{bdd}\}$$

3.6 Langages de Dyck

Définition

Nous allons à présent voir, en étudiant un ensemble de langages appelé *langages de Dyck*, que la théorie des langages permet de décrire un grand nombre d'objets différents (chemins, expressions, arbres, polygônes...) et de déduire des propriétés sur ceux-ci.

Considérons un ensemble A de symboles, et une copie $\overline{A}$ de A disjointe de A, c'est-à-dire l'ensemble $\overline{A} = \{\overline{s} \mid s \in A\}$ (on a $A \cap \overline{A} = \emptyset$). Et posons $\Sigma = A \cup \overline{A}$.

Par exemple, pour $A = \{\mathtt{a}, \mathtt{b}\}$, on a $\overline{A} = \{\bar{\mathtt{a}}, \bar{\mathtt{b}}\}$ et $\Sigma = \{\mathtt{a}, \mathtt{b}, \bar{\mathtt{a}}, \bar{\mathtt{b}}\}$.

On peut construire le langage de Dyck $\mathscr{D}_A$ sur $\Sigma = A \cup \overline{A}$ formellement par[23] :

$$\begin{cases} \varepsilon \in \mathscr{D}_A \\ \forall u, v, s \in \mathscr{D}_A \times \mathscr{D}_A \times A, \quad u \cdot s \cdot v \cdot \overline{s} \in \mathscr{D}_A \end{cases}$$

22. Il existe, de façon similaire, un langage quotient à droite.
23. On peut également utiliser $s \cdot u \cdot \overline{s} \cdot v$, le langage obtenu sera le même. La formulation que nous avons choisie ici permettra de faire quelques parallèles un peu plus loin avec d'autres structures.

Les langages de Dyck correspondent aux mots « bien parenthésés » sur un alphabet composé de symboles ouvrants et de leurs pendants fermants.

Par exemple, sur l'alphabet $\Sigma = A \cup \overline{A}$, avec $A = \{(, [\}$ et $\overline{A} = \{),]\}$, les mots ε, ( ), ([ ]) [ ], ou bien encore [ ] ([( )] ( )), bien parenthésés, sont des mots de Dyck, tandis que les mots ([ ], ([ )] ou encore) [ ] (n'en sont pas.

Une formulation équivalente [24] du langage est :

$$\begin{cases} \varepsilon \in \mathscr{D}_A & \\ \forall u, s \in \mathscr{D}_A \times A, \quad s \cdot u \cdot \overline{s} \in \mathscr{D}_A & \text{(parenthésage)} \\ \forall u, v \in \mathscr{D}_A \times \mathscr{D}_A, \quad u \cdot v \in \mathscr{D}_A & \text{(concaténation)} \end{cases}$$

Un tel langage trouve naturellement du sens en programmation, où il est important que les éléments ouvrant et fermant des « blocs » ou des expressions dans un programme soient bien équilibrés. Il s'agit non seulement des parenthèses, mais en Caml par exemple, également des mots-clés `do` et `done`, des `begin` et `end`, etc. De même, les balises en XML ou les environnements en LaTeX suivent des règles similaires. Bien évidemment, dans ces exemples, il n'y a pas *que* des délimiteurs.

Théorème 5. *Tout mot de Dyck w sur un alphabet $\Sigma = A \cup \overline{A}$ a les propriétés suivantes :*

- *sa longueur $|w|$ est paire ;*
- $\forall s \in A, |w|_s = |w|_{\overline{s}}$.

Démonstration. Ces propriétés découlent directement de la définition du langage de Dyck sur Σ, elles se retrouvent de façon immédiate par induction :

Pour la longueur :

- $|\varepsilon|$ est pair ;
- $\left|u \cdot s \cdot v \cdot \overline{s}\right| = |u| + 1 + |v| + 1 = 2 + |u| + |v|$ sera également pair.

De même, pour la seconde égalité, pour tout $s \in A$:

- $|\varepsilon|_s = 0$ et $|\varepsilon|_{\overline{s}} = 0$;
- pour tout $w \in \mathscr{D}_A$ non vide, il existe $u \in \mathscr{D}_A$, $v \in \mathscr{D}_A$ et $s' \in A$ tels que $w = u \cdot s' \cdot v \cdot \overline{s'}$.
  - si $s = s'$, $|w|_s - |w|_{\overline{s}} = \left|u \cdot s \cdot v \cdot \overline{s}\right|_s - \left|u \cdot s \cdot v \cdot \overline{s}\right|_{\overline{s}}$
    $= (|u|_s + 1 + |v|_s + 0) - \left(|u|_{\overline{s}} + 0 + |v|_{\overline{s}} + 1\right) = |u|_s - |u|_{\overline{s}} + |v|_s - |v|_{\overline{s}} = 0$;
  - si $s \neq s'$, $|w|_s - |w|_{\overline{s}} = \left|u \cdot s' \cdot v \cdot \overline{s'}\right|_s - \left|u \cdot s' \cdot v \cdot \overline{s'}\right|_{\overline{s}}$
    $= (|u|_s + 0 |v|_s + 0) - \left(|u|_{\overline{s}} + 0 + |v|_{\overline{s}} + 0\right) = |u|_s - |u|_{\overline{s}} + |v|_s - |v|_{\overline{s}} = 0$. □

24. L'équivalent n'a rien d'immédiat. On peut aisément voir que tous les mots que l'on peut construire à partir de la première définition peuvent aussi l'être avec la seconde, mais l'inverse est plus délicat à établir. Nous l'admettrons pour le moment.

Définitions alternatives

Il existe de très nombreuses façons de caractériser ou construire un langage de Dyck. Une autre approche, équivalente, consiste à utiliser une relation de réécriture.

Définition. Soit A un alphabet, et $\overline{A}$ une copie disjointe de A.

Pour deux mots w et w' de $\Sigma^* = \left(A \cup \overline{A}\right)^*$, on définit la *relation de réécriture* $\mapsto$ par

$$w \mapsto w' \iff \exists u, v, s \in \Sigma^* \times \Sigma^* \times A \text{ tels que } w = u \cdot s \cdot \overline{s} \cdot v \text{ et } w' = u \cdot v$$

On définit la *clôture réflexive*[a] *transitive* $\overset{\star}{\mapsto}$ de cette relation de réécriture par

$$w \overset{\star}{\mapsto} w' \iff \exists w_1, w_2, \dots w_n \in \left(\Sigma^*\right)^n \text{ tels que } w = w_1 \mapsto w_2 \mapsto \dots \mapsto w_n = w'$$

Le langage de Dyck sur Σ^* est $\left\{ w \in \Sigma^* \;\middle|\; w \overset{\star}{\mapsto} \varepsilon \right\}$

a. Le caractère réflexif signifie que $w \overset{\star}{\mapsto} w$ même si l'on n'a pas $w \mapsto w$, puisqu'il suffit de prendre $n = 1$.

Démonstration. Montrons successivement que tout mot de Dyck se réduit bien à ε via la réécriture $\mapsto$, et que tout mot qui peut être réécrit en ε est bien un mot de Dyck.

Tout d'abord, prouvons par récurrence sur $|w|$ que tout mot de Dyck w se réduit à ε.

- C'est vrai pour $|w| = 0$, puisque $\varepsilon \overset{\star}{\mapsto} \varepsilon$;
- Supposons que c'est vrai pour tout mot $w \in \mathscr{D}_A$ vérifiant $|w| \leqslant 2n$, et considérons $w \in \mathscr{D}_A$ tel que $|w| = 2n+2$ (rappelons que $|w|$ est pair). On peut écrire $w = u \cdot s \cdot v \cdot \overline{s}$ où u et v sont deux mots de Dyck et s un symbole de A.

  On a par ailleurs $|u| \leqslant 2n$ et $|v| \leqslant 2n$, donc d'après la récurrence, $u \overset{\star}{\mapsto} \varepsilon$ et $v \overset{\star}{\mapsto} \varepsilon$.

  Donc $w = u \cdot s \cdot v \cdot \overline{s} \overset{\star}{\mapsto} u \cdot s \cdot \overline{s} \mapsto u \overset{\star}{\mapsto} \varepsilon$.

Ce qui prouve que tout mot de Dyck w vérifie $w \overset{\star}{\mapsto} \varepsilon$.

Pour montrer, inversement, tout mot w vérifiant $w \overset{\star}{\mapsto} \varepsilon$ est un mot de Dyck, notons que $|w|$ est nécessairement pair (chaque application de la réduction supprime deux symboles, et on termine avec un mot de longueur nulle). Puis travaillons par récurrence sur $|w|$.

- C'est vrai pour $|w| = 0$, puisque ε est un mot de Dyck.
- Supposons que tout mot w vérifiant $w \overset{\star}{\mapsto} \varepsilon$ et $|w| \leqslant 2n$ est un mot de Dyck, et considérons à présent un mot w tel que $w \overset{\star}{\mapsto} \varepsilon$ avec $|w| = 2n+2$. Il existe une suite de mots w_i tels que $w = w_1 \mapsto w_2 \mapsto \dots \mapsto w_n = \varepsilon$, et une étape $w_i \mapsto w_{i+1}$ qui fait disparaître le dernier symbole de w_i.

Les mots w_i sont des sous-mots de w (on ne fait que retirer des symboles). Notons j et n les indices dans w des caractères supprimés lors de l'étape $w_i \mapsto w_{i+1}$ et posons $u = s_1 \dots s_{j-1}$ et $v = s_{j+1} \dots s_{n-1}$. On a $w = u \cdot s_j \cdot v \cdot s_n$ avec $s_n = \overline{s_j}$.

Les réécritures successives qui amènent w jusqu'à ε sont nécessairement appliquées soit à paires des symboles de u, soit à des paires de symboles de v. En effet, le symbole s_j sépare u et v tant que v n'est pas vide, et on ne peut éliminer que des symboles consécutifs. Par conséquent, on a $u \overset{\star}{\mapsto} \varepsilon$ et $v \overset{\star}{\mapsto} \varepsilon$.

Puisque par ailleurs $|u| \leqslant 2n$ et $|v| \leqslant 2n$, par récurrence, sont des mots de Dyck, aussi $w = u \cdot s_i \cdot v \cdot \overline{s_i}$ est également un mot de Dyck.

On a donc bien une équivalence entre les deux définitions. □

3.7 Langages de Dyck à deux symboles

Dorénavant, on se limite au cas où A est réduit à un seul symbole, soit $\Sigma = \{a, \bar{a}\}$. On peut proposer une nouvelle caractérisation pour les mots de Dyck sur un tel alphabet Σ :

Définition. Soit $\Sigma = \{a, \bar{a}\}$ un alphabet.

Considérons le morphisme additif $\sigma : \Sigma^* \mapsto \mathbb{Z}$ tel que $\sigma(a) = 1$ et $\sigma(\bar{a}) = -1$.

Le langage de Dyck $\mathscr{D}$ sur l'alphabet $\Sigma = \{a, \bar{a}\}$ est constitué des mots $w \in \Sigma^*$ tels que
- $\sigma(w) = 0$;
- pour tout préfixe u de w, $\sigma(u) \geqslant 0$.

Démonstration. Notons $\mathscr{D}'$ l'ensemble des mots w de Σ^* vérifiant $\sigma(w) = 0$ et, pour tout préfixe u de w, $\sigma(u) \geqslant 0$, et montrons que $\mathscr{D} = \mathscr{D}'$ en prouvant $\mathscr{D} \subset \mathscr{D}'$ et $\mathscr{D}' \subset \mathscr{D}$.

Tout d'abord, nous allons montrer que tout mot $w \in \mathscr{D}$ est également un mot de $\mathscr{D}'$ par récurrence sur $|w|$.

- Si $|w| = 0$, alors $w = \varepsilon$, donc $w \in \mathscr{D}$.
- Supposons que tout mot de Dyck $w \in \mathscr{D}$ vérifiant $|w| \leqslant 2n$ est également un mot de $\mathscr{D}'$, et considérons un mot de Dyck $w \in \mathscr{D}$ avec $|w| = 2n + 2$.

  Par définition, w s'écrit $u \cdot a \cdot v \cdot \bar{a}$, avec $u, v \in \mathscr{D}$.

  On a $|u| \leqslant 2n$ et $|v| \leqslant 2n$, donc d'après la récurrence, u et v sont des mots de $\mathscr{D}'$.

  On a $\sigma(w) = \sigma(a) + \sigma(u) + \sigma(\bar{a}) + \sigma(v) = 1 + 0 + (-1) + 0 = 0$.

  Les préfixes propres de w (le cas de w est déjà traité) sont :
  — u', où u' est un préfixe de u, avec $\sigma(\cdot u') \geqslant 0$;
  — $u \cdot a$, avec $\sigma(u \cdot a) = 0 + 1 = 1 \geqslant 0$;
  — $u \cdot a \cdot v'$, où v' est un préfixe de v, avec $\sigma(u \cdot a \cdot v') = 1 + \sigma(v') \geqslant 1 \geqslant 0$.

On a donc bien $w \in \mathscr{D} \Longrightarrow w \in \mathscr{D}'$.

Dans un second temps, nous allons montrer, inversement, que tout mot $w \in \mathscr{D}'$ est également un mot de $\mathscr{D}$. Remarquons tout d'abord que $|w|$ est nécessairement pair puisque l'on a $\sigma(w) = 0$. Puis procédons à nouveau par récurrence sur $|w|$

- Si $|w| = 0$, alors $w = \varepsilon$, donc $w \in \mathscr{D}$.
- Supposons que tout mot de $\mathscr{D}'$ vérifiant $|w| \leqslant 2n$ est également un mot de $\mathscr{D}$, et considérons un mot $w = s_1 s_2 \dots s_{2n+2} \in \mathscr{D}'$.

  On a $\sigma(w) = 0$ et $\sigma(s_1 s_2 \dots s_{2n+1}) \geqslant 0$, donc $\sigma(s_{2n+2}) = 1$, et par conséquent $s_{2n+2} = \bar{a}$.

  Considérons à présent $E = \{u_k \text{ préfixe propre de } w \mid \sigma(u_k) = 0\}$. Cet ensemble est non-vide puisque $\sigma(\varepsilon) = 0$, d'où $\varepsilon \in E$. Tous les mots de E étant de taille différente, il y en a un de longueur de longueur maximale, que l'on note u.

  On peut donc décomposer w en écrivant $w = u \cdot s_{|u|+1} \cdot v \cdot \bar{a}$.

  Puisque $\sigma(u) = 0$ et $\sigma(u \cdot s_{|u|+1}) \geqslant 0$, on a nécessairement $s_{|u|+1} = a$, et par conséquent, w s'écrit $w = u \cdot a \cdot v \cdot \bar{a}$.

  $\sigma(u) = 0$ par construction, et pour tout préfixe u' de u, u' est un préfixe de w, donc $\sigma(u') \geqslant 0$, et par conséquent $u \in \mathscr{D}$ d'après l'hypothèse de récurrence.

  On a par ailleurs $\sigma(w) = 0 = \sigma(u) + \sigma(a) + \sigma(v) + \sigma(\bar{a}) = 0 + 1 + \sigma(v) + \sigma(-1)$, d'où $\sigma(v) = 0$. Et pour tout préfixe v' de v, $\sigma(u \cdot a \cdot v') = \sigma(u) + \sigma(a) + \sigma(v') = 1 + (v')$. Or $\sigma(u \cdot a \cdot v') \geqslant 1$ (puisque u est le plus long mot de E) d'où $\sigma(v') \geqslant 0$. Ce qui permet d'affirmer que $v \in \mathscr{D}$.

  Pour conclure, puisque $w = u \cdot a \cdot v \cdot \bar{a}$ avec $u \in \mathscr{D}$ et $v \in \mathscr{D}$, d'après la définition des mots de Dyck, $w \in \mathscr{D}$.

On a donc bien $w \in \mathscr{D}' \Longrightarrow w \in \mathscr{D}$.

Et par conséquent, $\mathscr{D} = \mathscr{D}'$. □

Chemins de Dyck

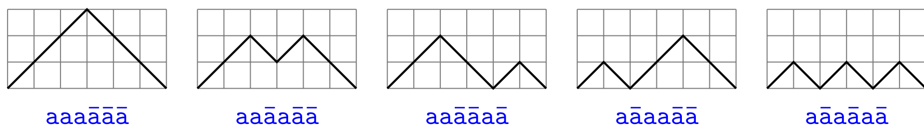
Il existe cinq mots de Dyck de longueur 6 sur $\Sigma = \{a, \bar{a}\}$: il s'agit des mots $aaa\bar{a}\bar{a}\bar{a}$, $aa\bar{a}a\bar{a}\bar{a}$, $aa\bar{a}\bar{a}a\bar{a}$, $a\bar{a}aa\bar{a}\bar{a}$ et $a\bar{a}a\bar{a}a\bar{a}$. On peut se demander combien il existe de mots de Dyck de longueur $2n$ parmi les 2^{2n} mots de Σ^{2n}. Pour répondre à cette question, nous allons interpréter géométriquement les mots de Dyck.

Définition. Un *chemin de Dyck* (ou *montagne de Dyck*) de longueur $2n$ est une suite de points du plan $(M_k)_{k \in [\![0..2n]\!]}$ telle que
- $M_0 = (0, 0)$ et $M_{2n} = (2n, 0)$;
- pour tout $k \in [\![0..2n-1]\!]$, on a $\overrightarrow{M_k M_{k+1}} = (1, 1)$ ou $\overrightarrow{M_k M_{k+1}} = (1, -1)$;
- pour tout $k \in [\![1..2n-1]\!]$, l'ordonnée de M_k est positive.

La « traduction » d'un mot de Dyck en un chemin de Dyck est immédiate : on peut, à partir d'un quelconque mot de Dyck, construire un chemin de Dyck en partant de l'origine, et en associant à chaque symbole un déplacement :

- un « a » correspond à un $\overrightarrow{M_kM_{k+1}} = (1,1)$,
- un « ā » correspond à un $\overrightarrow{M_kM_{k+1}} = (1,-1)$.

Par exemple, les cinq mots de Dyck de longueur 6 correspondent aux chemins suivants :

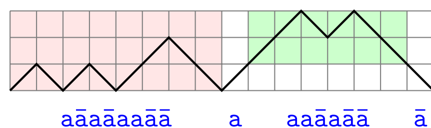


Le morphisme σ et ses propriétés sur les mots de Dyck permet de faire le lien entre les mots de Dyck et cette interprétation géométrique : pour un mot w et le chemin $M_0, M_1, \dots, M_k, \dots, M_{2n}$ associé, le morphisme σ sur le préfixe de longueur k de w correspond à l'ordonnée du point M_k dans le chemin.

Ainsi, le mot sur Σ^{2n} construit à partir d'un chemin de Dyck est bien un mot de Dyck, car $\sigma(w) = 0$ (c'est l'ordonnée du point M_{2n}) et pour tout $k \in [\![1 \,..\, 2n-1]\!]$, le préfixe u de longueur k de w vérifie $\sigma(u) \geqslant 0$ (il s'agit de l'abcisse de M_k).

Inversement, une suite de points M_k construit à partir d'un mot de Dyck constitue bien un chemin de Dyck pour les mêmes raisons.

Les chemins de Dyck permettent de voir aisément que la décomposition d'un mot de Dyck w en $w = u \cdot$ a $\cdot\, v \cdot$ ā où u et v sont des mots de Dyck est unique [25] : le symbole a doit correspondre à la dernière montée d'une ordonnée 0 vers une ordonnée 1, car v devant être un mot de Dyck, cette partie du chemin ne peut « redescendre » au niveau de l'axe des abscisses. Par exemple, pour le mot aāaāaaāāaaaāaāāā, on a nécessairement $u =$ aāaāaaāā et $v =$ aaāaāā :



Théorème 6. *Si l'on note C_n le nombre de mots de Dyck de longueur $2n$, la suite $(C_n)_{n\in\mathbb{N}}$ peut être définie récursivement par*

$$C_0 = 1 \quad et \quad \forall n \in \mathbb{N},\; C_{n+1} = \sum_{k=0}^{n} C_k \times C_{n-k}$$

Il s'agit de la suite des nombres de Catalan [a] (nommés d'après le mathématicien belge E. C. Catalan), que l'on rencontre dans beaucoup de problèmes combinatoires.

a. Suite A00018 dans l'encyclopédie des suites d'entiers, `oeis.org`.

25. Ce qui signifie qu'il n'y a pas d'ambiguité dans le parenthésage d'une expression.

Les premiers termes de cette suite sont $C_0 = 1$, $C_1 = 1$, $C_2 = 2$, $C_3 = 5$, $C_4 = 14$, $C_5 = 42$, $C_6 = 132$, $C_7 = 429$, $C_8 = 1430$, $C_9 = 4862$, $C_{10} = 16796$...

Démonstration. Il existe un unique mot de Dyck de longueur 0, ε, d'où $C_0 = 1$.

Tout mot de Dyck w de longueur $2n+2$ (avec $n \in \mathbb{N}$) se décompose de façon unique en $w = u \cdot$ a $\cdot\, v \cdot$ ā, avec u et v mots de Dyck, donc de longueur respectives $2p$ et $2q$ vérifiant $2p + 2q = 2n$.

Ce qui conduit à la relation $C_{n+1} = \sum\limits_{k=0}^{n} C_k \times C_{n-k}$. □

Théorème 7. *Le nombre de Catalan C_n vaut*

$$C_n = \binom{2n}{n} - \binom{2n}{n-1} = \frac{1}{n+1}\binom{2n}{n}$$

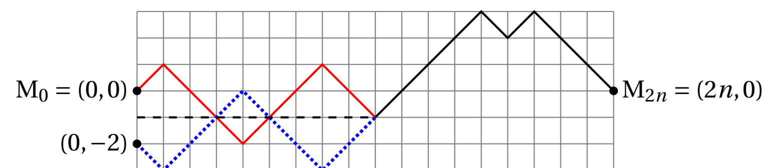
Démonstration. Il existe quantité de démonstrations de ce résultat, mais les chemins de Dyck nous en fournissent une intéressante, utilisant le principe de réflexion.

Considérons les chemins de longueur $2n$ menant de $(0,0)$ à $(2n,0)$, et dont chaque segment correspond à un vecteur de composantes $(1,1)$ ou $(1,-1)$.

Il y a $\binom{2n}{n}$ chemins de ce type (il doit y avoir autant de montées que de descentes).

Parmi ces chemins, C_n sont des chemins de Dyck.

Pour les chemins qui ne sont pas des chemins de Dyck, au moins un des points sur le chemin a une ordonnée égale à -1. Prenons le dernier de ces points, et effectuons une symétrie de la partie du chemin à gauche de ce point par rapport à la droite $y = -1$.



On peut donc associer à tout chemin qui n'est pas un chemin de Dyck un chemin menant de $(0,-2)$ à $(2n,0)$, dont chaque segment correspond à un vecteur de composantes $(1,1)$ ou $(1,-1)$.

De tels chemins montent $n+1$ fois, et descendent $n-1$ fois. Il en existe donc $\binom{2n}{n-1}$.

On a donc $\binom{2n}{n} = C_n + \binom{2n}{n-1}$, d'où le résultat. □

Mots de Dyck et arbres

Les mots de Dyck ont a la particularité remarquable d'être liés plus ou moins fortement à de très nombreux objets que l'on peut être amenés à manipuler en informatique, ce qui permettra de déduire quelques résultats intéressants, par exemple de dénombrement. Par exemple, il y a un lien très fort avec différent types d'arbres.

Tout d'abord, on peut construire une bijection liant les mots de Dyck de longueur $2n$ et les arbres binaires à n nœuds[26] en associant :

- au mot de Dyck ε, l'arbre vide (`Nil`) ;
- à tout autre mot de Dyck w, lequel se décompose sous la forme $u\cdot \mathtt{a}\cdot v\cdot \bar{\mathtt{a}}$ où u et v des mots de Dyck, l'arbre binaire construit par induction structurelle, dont la racine a pour fils gauche l'arbre associé au mot u et pour fils droit celui associé au mot v.

Démonstration. On travaille par récurrence sur n.

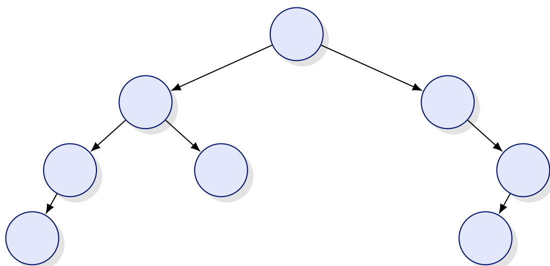
- le mot ε, de longueur nulle, est associé à un arbre sans nœud.
- pour tout entier $n > 0$, supposons que tout mot de Dyck de longueur $2p$ avec $p < n$ est bien en bijection avec un arbre contenant p nœuds, et considérons un mot de Dyck w de longueur $|w| = 2n$.

  Ce mot w se décompose en $w = u\cdot \mathtt{a}\cdot v\cdot \bar{\mathtt{a}}$, où u et v sont des mots de Dyck. On peut donc trouver des entiers positifs p et p' tels que $|u| = 2p$ et $|v| = 2p'$.

  Par ailleurs, $|u| + |v| = 2n - 2$, donc $p + p' = n - 1$, $p < n$ et $p' < n$.

  Le mot w est donc associé à un arbre contenant une racine plus p nœuds dans le sous-arbre gauche (par application de la récurrence) et p' nœuds dans le sous-arbre droit (même chose), soit $p + p' + 1$ nœuds au total, soit n nœuds. □

Par exemple, le mot $\mathtt{a\bar{a}a\bar{a}aa\bar{a}\bar{a}aaa\bar{a}a\bar{a}\bar{a}\bar{a}}$ (de longueur 16) correspond à l'arbre binaire à 8 nœuds ci-dessous :



Pour un type arbre défini en Caml par

```
# type 'a arbre = Nil | Noeud of 'a * 'a arbre * 'a arbre;;
```

26. On qualifie ici de nœuds à la fois les nœuds internes et les feuilles de l'arbre.

On peut définir la conversion d'un arbre en mot de Dyck par[27] [28] [29] :

```
# let rec versMot = function
    | Nil -> ""
    | Noeud (_, fg, fd) -> (versMot fg) ^ "a" ^ (versMot fd) ^ "b";;

val versMot : 'a arbre -> string = <fun>
```

Par exemple :

```
# versMot (Noeud(1,
             Noeud(2,
               Noeud(4,
                 Noeud(7, Nil, Nil),
                 Nil),
               Noeud(5, Nil, Nil)),
             Noeud(3,
               Nil,
               Noeud(6,
                 Noeud(8, Nil, Nil),
                 Nil))));;

- : string = "ababaabbaaababbb"
```

Pour effectuer la transformation inverse, on a besoin de déterminer la décomposition $u\cdot \mathtt{a}\cdot v\cdot \bar{\mathtt{a}}$ d'un mot w non vide. Pour ce faire, on peut chercher le plus grand préfixe u de w tel que $\sigma(u) = 0$, ce qui peut se faire en parcourant le mot à rebours :

```
# let decomp w =
    let sigma = ref 1 and i = ref (String.length w - 1) in
      while !sigma > 0 do   (* invariant : sigma = σ(u) où u est *)
        i := !i - 1;        (*    le préfixe de longueur !i de w *)
        sigma := !sigma - (if w.[!i] = 'a' then 1 else -1);
      done;
      (String.sub w 0 !i),
      (String.sub w (!i+1) (String.length w - !i - 2));;

val decomp : string -> string * string = <fun>
```

27. Les étiquettes sur les nœuds de l'arbre n'ont pas d'importance dans cette transformation.
28. On utilise dans le résultat le caractère b en lieu et place de $\bar{\mathtt{a}}$, qui ne figure pas dans le code ASCII.
29. L'utilisation de concaténations est maladroite en terme de complexité, une meilleure solution consisterait à déterminer la taille de l'arbre pour connaître à l'avance la taille de la chaîne de caractères à construire, puis de muter les éléments de la chaîne.

La fonction decomp permet ainsi d'obtenir les facteurs u et v de la décomposition [30] $u \cdot s \cdot v \cdot \overline{s}$ d'un mot de Dyck :

```
# decomp "ababaabbaaababbb";;

- : string * string = ("ababaabb", "aababb")
```

La construction de l'arbre associé à la chaîne est alors immédiate [31] :

```
# let rec versArbre = function
    | "" -> Nil
    | w -> let u, v = decomp w in Noeud((), versArbre u, versArbre v);;

val versArbre : string -> unit arbre = <fun>
```

Précisons que cette fonction, destinée à illustrer la décomposition, n'est guère efficace (quadratique en n car decomp est linéaire en la taille de son argument), et que l'on pourrait reconstruire l'arbre plus efficacement, linéairement en la taille du mot de Dyck, en écrivant par exemple :

```
# let versArbre ch =
    let rec construit = function
      | i when i >= 0 && ch.[i] = 'b'
            -> let fd, j = construit (i-1) in
               let fg, k = construit (j-1) in
                 Noeud ((), fg, fd), k
      | i -> Nil, i
    in fst (construit (String.length ch - 1));;

val versArbre : string -> unit arbre = <fun>
```

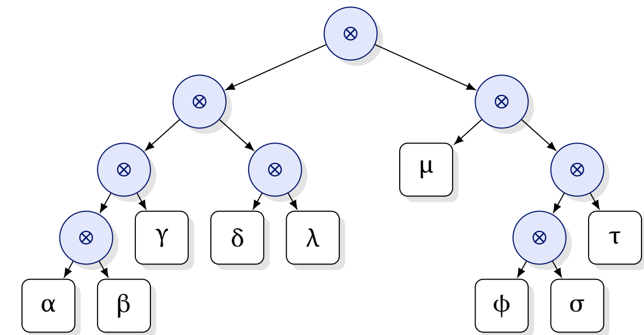
On peut de la même façon associer établir une bijection entre les mots de Dyck et les arbres binaires stricts non-vides (un mot de longueur $2n$ correspondant à un arbre binaire strict avec n nœuds internes et $n-1$ feuilles), puis qu'il existe une bijection naturelle entre les arbres binaires à n nœuds et les arbres binaires stricts à n nœuds internes (et donc $n-1$ feuilles), simplement en conservant les nœuds de l'arbre binaire comme nœuds internes, et leur ajoutant des feuilles.

L'arbre binaire strict résultat d'une telle transformation, pour notre exemple, est par exemple représenté ci-après.

30. Unique, rappelons-le.
31. On a choisi des objets () de type « unit » comme étiquettes pour les feuilles car le mot de Dyck ne décrit que la structure de l'arbre, pas son contenu.

Cette bijection entre mots de Dyck et arbres binaires stricts n'est évidemment pas surprenante. En effet, pour un opérateur binaire ⊗ non associatif, un arbre binaire strict représente un ordre d'évaluation, et le mot de Dyck peut être compris comme le parenthésage associé [32], en associant « ⊗(» à « a » et «) » à « ā », et en intercalant les arguments (au début de l'expression et après chaque parenthèse ouvrante) :

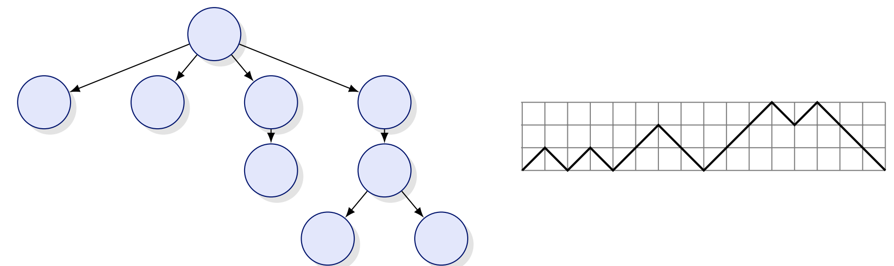
$$a\bar{a}a\bar{a}aa\bar{a}\bar{a}aaa\bar{a}a\bar{a}\bar{a}\bar{a} \quad \triangleright \quad \alpha \otimes (\beta) \otimes (\gamma) \otimes (\delta \otimes (\lambda)) \otimes (\mu \otimes (\phi \otimes (\sigma) \otimes (\tau)))$$



Puisqu'il existe C_n mots de Dyck de longueur $2n$, on peut en conclure qu'il existe donc C_n arbres binaires à n nœuds, et C_n arbres binaires stricts à n nœuds internes et $n+1$ feuilles, soit $2n+1$ nœuds.

Les mots de Dyck fournissent ici à la fois un moyen de décrire un quelconque arbre binaire ou arbre binaire strict, et des outils pour obtenir des résultats sur ceux-ci.

De la même façon, il existe une bijection entre les mots de Dyck de longueur $2n$ et les arbres contenant n arcs : il suffit de considérer le parcours en profondeur d'un tel arbre, a dénotant la descente d'un nœud vers son fils, et ā la remontée d'un nœud vers son père. Le chemin de Dyck associé peut être vu comme l'évolution de la profondeur lors du parcours. Par exemple, notre mot $a\bar{a}a\bar{a}aa\bar{a}\bar{a}aaa\bar{a}a\bar{a}\bar{a}\bar{a}$ correspond à l'arbre à 8 arcs ci-dessous :



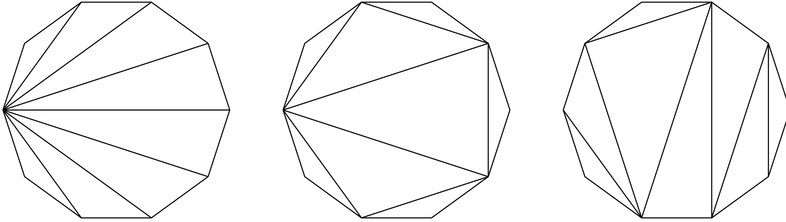
Cela nous permet d'en déduire qu'il existe C_n arbres à n arcs (et $n+1$ nœuds), et donc un lien entre les arbres binaires à n nœuds et les arbres à $n+1$ nœuds.

32. Il ne s'agit pas du parenthésage minimal, car les seules parenthèses ne permettraient alors pas de déterminer la taille de l'arbre ($a \otimes (b \otimes c)$ et $a \otimes (b \otimes c \otimes d)$ font apparaître les mêmes parenthèses, mais correspondent à deux arbres distincts), mais d'un parenthésage systématique.

Mots de Dyck et polygônes

On peut aussi trouver un lien entre les mots de Dyck et la triangulation d'un polygône convexe.

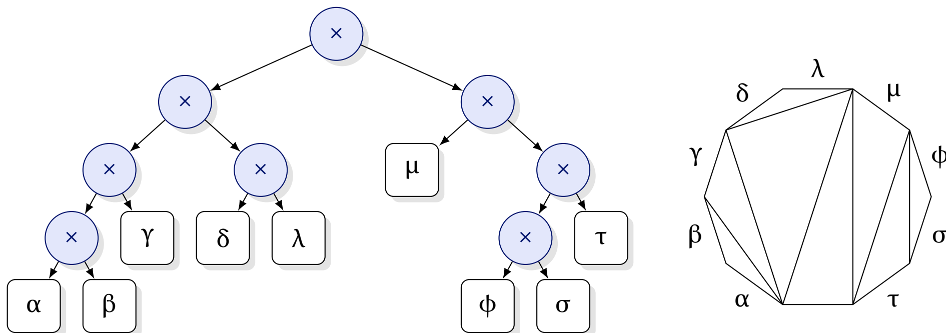
Considérons un polygône convexe à n côtés. Trianguler ce polygône consiste à le transformer en un assemblage de triangles, en choisissant $n-3$ cordes reliant deux sommets de ce polygône, ces cordes n'étant pas sécantes deux à deux. Ci-dessous, on retrouve trois triangulations possibles pour un décagone :



Il est possible de construire une bijection entre ces triangulations [33] et les mots de Dyck de taille $2(n-2)$. Pour le voir, le plus simple est d'interpréter le mot de Dyck comme un arbre binaire strict, contenant $n-1$ feuilles. Ces $n-1$ feuilles sont associées à $n-1$ des arêtes du polygône (prises dans l'ordre). Chaque nœud interne de l'arbre doit être interprété comme l'élimination d'un des sommets du polygône, celui entre les deux arêtes désignées par ses deux fils. L'élimination du sommet crée ainsi un polygône avec une arête de moins, jusqu'à ce qu'il ne reste qu'un triangle.

Toute triangulation du polygône peut être associée à un tel arbre, on a donc bien une bijection entre les arbres binaires à $2n-3$ nœuds et les triangulations des polygônes à n côtés.

Toujours pour notre mot $\mathtt{a\bar{a}a\bar{a}aa\bar{a}\bar{a}aaa\bar{a}a\bar{a}\bar{a}\bar{a}}$:



On déduit ainsi qu'il existe C_{n-2} triangulations possibles d'un polygône à n côtés.

33. On suppose chaque sommet du polygone identifié : s'il possède des symétries, on retrouve autant de triangulations « différentes » mais qui se déduisent par ces mêmes symétries.

4 Expressions et langages réguliers

4.1 Introduction

Nous allons à présent nous pencher plus avant sur une manière de décrire pratiquement et efficacement un langage, les *expression régulière* [34] (ou *expressions rationnelles*).

Définition. Soit un alphabet Σ.
L'ensemble $\mathcal{R}_\Sigma$ des expressions régulières sur un alphabet Σ est un langage sur l'alphabet $\Sigma \cup \{\emptyset, \epsilon, \ast, |, ., (,)\}$ construit par induction :

- $\emptyset \in \mathcal{R}_\Sigma$ et $\epsilon \in \mathcal{R}_\Sigma$;
- pour tout $s \in \Sigma$, $s \in \mathcal{R}_\Sigma$;
- pour tout $e \in \mathcal{R}_\Sigma$, on a $(e\ast) \in \mathcal{R}_\Sigma$;
- pour tout $e, f \in \mathcal{R}_\Sigma^2$, on a $(e\,.\,f) \in \mathcal{R}_\Sigma$;
- pour tout $e, f \in \mathcal{R}_\Sigma^2$, on a $(e\,|\,f) \in \mathcal{R}_\Sigma$.

Ainsi, le mot `(((a*).b)|(b.(a*)))` sur l'alphabet $\{a, b, \emptyset, \epsilon, \ast, |, ., (,)\}$ est un exemple d'expression régulière sur l'alphabet $\Sigma = \{a, b\}$.

Le symbole « `|` » est un constructeur binaire généralement appelé *choix*, le symbole « `.` », un constructeur binaire appelé *concaténation*, le symbole « `*` » un constructeur unaire appelé *étoile*.

Il est d'usage d'omettre le point du constructeur de concaténation dans les expressions régulières. Les parenthèses peuvent être également omises lorsqu'il n'y a pas ambiguïté [35]. On définit des règles de priorités permettent d'en limiter le nombre. Par ordre décroissant de priorité :

- l'étoile,
- la concaténation,
- le choix.

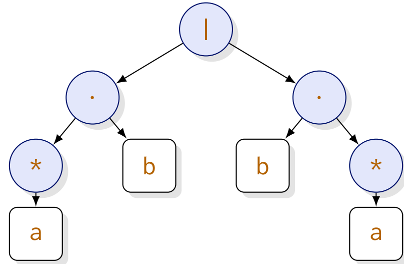
Ainsi, l'expression régulière `(((a*).b)|(b.(a*)))` sera généralement réécrite, plus simplement, `a*b|ba*`.

Lorsque l'on considère l'ensemble des expressions régulières sur Σ comme un langage sur l'alphabet $\Sigma \cup \{\emptyset, \epsilon, \ast, |, ., (,)\}$, il faut prendre garde à pouvoir différencier les éléments de Σ et les symboles tels que $\emptyset$, ϵ, `|` ou les parenthèses [36]. Il faut également faire attention à ne pas confondre l'expression régulière « ϵ » et le mot vide ε du langage $\mathcal{R}_\Sigma$!

34. Par influence de l'anglais, où l'on parle de *regular expressions*, parfois abrégé en « *regex* ».
35. L'omission du point et le fait de pouvoir omettre les parenthèses quand il n'y a pas ambiguïté conduit donc à un langage *différent* de celui décrit dans la définition.
36. Par exemple, il est fréquent de travailler avec pour Σ l'ensemble des caractères ASCII, qui incluent notamment les parenthèses. Dans la pratique, on utilisera par exemple `(` et `\(` pour différencier la parenthèse de Σ et celle utilisée pour l'expression régulière proprement dite (et `\\` pour désigner le caractère `\`). La syntaxe exacte des expressions régulières dépend beaucoup de l'application qui les utilise.

Une expression régulière sur un alphabet Σ peut être associée à un arbre binaire où les feuilles sont des éléments de $\emptyset \cup \epsilon \cup \Sigma$, et les nœuds internes aux constructeurs d'étoile `*`, de concaténation `.` et de choix `|`.

Par exemple, l'expression régulière `a*b|ba*` correspond à l'arbre suivant :



L'intérêt majeur de ces expressions régulières est qu'elles offrent un moyen pratique de décrire certains langages :

Définition. Toute expression régulière e sur un alphabet Σ peut être *interprétée* en un langage $\mathscr{L}(e)$ sur Σ :

- à l'expression régulière $\emptyset$ est associé le langage vide $\mathscr{L}(\emptyset) = \{\}$;
- à l'expression régulière ϵ est associé le langage $\mathscr{L}(\epsilon) = \{\varepsilon\}$ réduit au mot vide ;
- à l'expression régulière s où $s \in \Sigma$ est associé le langage $\mathscr{L}(s) = \{s\}$ réduit au mot contenant un unique caractère s ;
- à l'étoile $e*$ de l'expression régulière e est associée l'étoile du langage $\mathscr{L}(e)$, soit $\mathscr{L}(e*) = \mathscr{L}(e)^*$;
- à la concaténation $e\,.\,f$ des expressions régulières e et f est associée la concaténation des langages $\mathscr{L}(e)$ et $\mathscr{L}(f)$, soit le langage $\mathscr{L}(e\,.\,f) = \mathscr{L}(e)\mathscr{L}(f)$;
- et enfin, au choix $e\,|\,f$ des expressions régulières e et f est associé la union des langages $\mathscr{L}(e)$ et $\mathscr{L}(f)$, soit le langage $\mathscr{L}(e\,|\,f) = \mathscr{L}(e) + \mathscr{L}(f)$.

L'interprétation $\mathscr{L}$ est une application de $\mathcal{R}_\Sigma$ dans l'ensemble $\mathcal{P}(\Sigma^*)$ des langages sur Σ, c'est-à-dire des parties de Σ^*.

4.2 Exemples

On considère l'alphabet $\Sigma = \{$a, b, c$\}$.

L'expression régulière `aa*bb*` correspond au langage aa*bb* des mots contenant une succession d'un ou plusieurs a suivis d'une succession d'un ou plusieurs b.

L'expression régulière `(abc)*` correspond au langage (abc)* des mots soit vides, soit résultats d'une concaténation d'un nombre quelconque de facteurs abc.

L'expression régulière `(a|b)*` correspond au langage (a + b)* des mots ne contenant pas le symbole c.

L'expression régulière `(a|b)*(a|c)*` correspond au langage (a + b)*(a + c)* des mots soit ne contenant pas de b, soit ne contenant pas de c, soit dans lesquels les symboles c apparaissent après les symboles b (lorsque le mot contient à la fois des b et des c).

L'expression régulière `a(a|b|c)*a` correspond au langage a(a + b + c)*a des mots de longueur au moins 2 débutant et se terminant par un a.

L'expression régulière `(a|b)*ba(a|b)*` correspond au langage (a + b)*ba(a + b)* des mots sans c contenant au moins une fois le facteur ba.

4.3 Langages réguliers

Définition. Un langage L sur Σ est dit *régulier* si et seulement si il existe une expression régulière e telle que $\mathscr{L}(e) = \mathrm{L}$.

On aura remarqué, dans les exemples précédents, qu'il y a une forte ressemblance entre une expression régulière et le langage qu'elle représente. En fait, en choisissant les mêmes notations pour le constructeur de choix et l'union de langages, et pour les autres opérations, on se retrouve avec les mêmes représentations.

Il est donc fréquent que l'on confonde une expression régulière avec le langage qu'elle désigne, pour simplifier. Mais il faut garder à l'esprit que les deux objets sont différents (comme les polynômes, en mathématiques, ne sont pas des fonctions, même si là aussi la correspondance est immédiate), cela pourra avoir une importance par exemple en logique.

Théorème 8. *L'ensemble des langages réguliers sur Σ est la plus petite famille de langages (plus petite partie de $\mathcal{P}(\Sigma)$) qui contienne le langage vide, les langages $\{s\}$ réduit à un symbole $s \in \Sigma$, et qui soit stable par étoile, par concaténation et par union.*

En fait, on peut, dans le théorème précédent, remplacer les langages réduits à un symbole par les langages finis, la formulation obtenue sera équivalente (puisqu'il doit y avoir stabilité par la concaténation, tous les langages finis doivent être présents, et inversement, si tous les langages finis sont présents, tous les langages réduits à un symbole le sont aussi).

Notons au passage que cet ensemble des langages réguliers, muni de l'union (d'élément neutre le langage vide) et la concaténation (d'élément neutre le langage réduit au mot vide) a une structure de semi-anneau.

4.4 Équivalences d'expressions régulières

L'opération d'interprétation $\mathscr{L} : \mathcal{R}_\Sigma \mapsto \mathcal{P}(\Sigma^*)$ n'est pas injective [37]. Il existe des expressions régulières différentes qui s'interprètent en un même langage.

Définition. Deux expressions régulières e et f sont dites *(sémantiquement) équivalentes* si elles s'interprètent en un même langage, soit $\mathscr{L}(e) = \mathscr{L}(f)$.

Par exemple, les expressions régulières `ϵ|aa*|a*ba*` et `a*(b|ϵ)a*` sont *distinctes*, mais sémantiquement équivalentes car elles s'interprètent en un même langage L des mots sur $\Sigma = \{a, b\}$ contenant au plus un b.

Notons qu'il existe de très nombreuses façons de définir le langage L :

$$L = \{\varepsilon\} + aa^* + a^*ba^*$$

$$L = \left\{a^n \mid n \in \mathbb{N}\right\} + \left\{a^nba^p \mid n, p \in \mathbb{N}^2\right\}$$

$$L = \left\{ w \in \{a, b\}^* \mid |w|_b \leqslant 1 \right\}$$

Il s'agit cependant à chaque fois du *même* langage L, car un langage désigne une partie de Σ^*, pas la façon dont celle-ci est définie.

4.5 Élimination des symboles ∅ et ϵ

Nous allons à présent montrer que, pour un langage régulier L non-vide, on peut se passer du symbole ∅ pour écrire une expression régulière e telle que $\mathscr{L}(e) = L$, et que ϵ n'est indispensable que pour des langages contenant le mot vide. Il est donc fréquent que des outils utilisant les expressions régulières fassent l'impasse sur ces deux symboles.

Lemme 2. *Si un langage décrit par une expression régulière e est non-vide, alors il existe une expression régulière e' équivalente ne contenant pas le symbole ∅.*

Démonstration. Nous allons construire une telle expression régulière e' équivalente par induction.

- si $e = ϵ$, $e' = e$ convient;
- si $e = s$ où $s \in \Sigma$, $e' = e$ convient;
- si $e = f*$,
  - — si $\mathscr{L}(f)$ est le langage vide, alors $\mathscr{L}(f*) = \{\varepsilon\}$, donc ϵ convient;
  - — sinon, f s'interprète en un langage non-vide, donc par hypothèse d'induction, il existe une expression régulière f' sans le symbole ∅ telle que $\mathscr{L}(f) = \mathscr{L}(f')$, et $e' = f'*$ convient;
- si $e = f\,.\,g$, ni f ni g ne peuvent s'interpréter en un langage vide (sinon e s'interpréterait également en un langage vide), donc par hypothèse d'induction, il existe f' et g' dépourvues de ∅ telles que $\mathscr{L}(f') = \mathscr{L}(f)$ et $\mathscr{L}(g') = \mathscr{L}(g)$, et $e' = f'\,.\,g'$ convient;
- enfin, si $e = f \mid g$, l'un des langages $\mathscr{L}(f)$ et $\mathscr{L}(g)$ au moins est non-vide, aussi trois cas sont possibles :
  - — si $\mathscr{L}(f) = \emptyset$, par hypothèse d'induction, il existe une expression régulière g' dépourvue de ∅ telle que $\mathscr{L}(g') = \mathscr{L}(g)$ et $e' = g'$ convient;
  - — si $\mathscr{L}(g) = \emptyset$, par hypothèse d'induction, il existe une expression régulière f' dépourvue de ∅ telle que $\mathscr{L}(f') = \mathscr{L}(f)$ et $e' = f'$ convient;
  - — si $\mathscr{L}(f) \neq \emptyset$ et $\mathscr{L}(g) \neq \emptyset$, par hypothèse d'induction, il existe des expressions régulières f' et g' dépourvues de ∅ telles que $\mathscr{L}(f') = \mathscr{L}(f)$ et $\mathscr{L}(g') = \mathscr{L}(g)$, et $e' = f' \mid g'$ convient. □

Lemme 3. *Si un langage décrit par une expression régulière e est non-vide, alors il existe une expression régulière e' équivalente ne contenant ni le symbole ∅, ni le symbole ϵ tel que e soit sémantiquement équivalente soit à ϵ, soit à e' soit à ϵ | e'.*

Démonstration. On suppose avoir déjà éliminé le symbole ∅ de l'expression régulière, et on procède par induction pour trouver une expression régulière sémantiquement équivalente ayant l'une des trois formes proposées.

- pour $e = ϵ$ et $e = s$ (avec $s \in \Sigma$), $e' = e$ convient;
- pour $e = f*$, d'après l'hypothèse d'induction, il existe une expression régulière sémantiquement équivalente à f qui soit :
  - — soit ϵ, auquel cas $e' = ϵ$ convient;
  - — soit de la forme f' ou $ϵ \mid f'$ (où le symbole ϵ est absent de f'), auquel cas $e' = f'*$ convient;
- pour $e = f\,.\,g$, on utilise l'hypothèse d'induction pour trouver des expressions régulières sémantiquement équivalentes répondant aux critères, et on les combine de la sorte :

	$ϵ$	g'	$ϵ \mid g'$
$ϵ$	$ϵ$	g'	$ϵ \mid g'$
f'	f'	$(f'\,.\,g')$	$(f' \mid f'\,.\,g')$
$ϵ \mid f'$	$ϵ \mid f'$	$(g' \mid f'\,.\,g')$	$ϵ \mid (f' \mid g' \mid f'\,.\,g')$

- pour $e = f \mid g$, on procède de même avec ces combinaisons :

	$ϵ$	g'	$ϵ \mid g'$
$ϵ$	$ϵ$	$ϵ \mid g'$	$ϵ \mid g'$
f'	$ϵ \mid f'$	$(f' \mid g')$	$ϵ \mid (f' \mid g')$
$ϵ \mid f'$	$ϵ \mid f'$	$ϵ \mid (f' \mid g')$	$ϵ \mid (f' \mid g')$

□

37. Nous verrons plus tard qu'elle n'est pas surjective non plus, il existe des langages pour lesquels il n'existe aucune expression régulière dont l'interprétation est le langage en question.

On le devine, déterminer si deux expressions régulières différentes s'interprètent en un même langage est un problème difficile, de même que trouver l'expression régulière la « plus simple » décrivant un langage régulier[38]. La notion de « plus simple » n'étant pas très précise, on peut définir des critères pour comparer deux expressions régulières, comme leur *longueur* :

Définition. On définit la longueur $|e|$ d'une expression régulière par induction :
- $|\emptyset| = |\epsilon| = 1$;
- $\forall s \in \Sigma, |s| = 1$;
- si $e = f\star$, alors $|e| = |f| + 1$;
- si $e = f\,.\,g$, alors $|e| = |f| + |g| + 1$;
- si $e = f\,|\,g$, alors $|e| = |f| + |g| + 1$.

Il s'agit donc de la longueur de l'expression régulière en nombre de symboles, en ignorant les parenthèses, mais en comptant les « . » des concaténations. Ou bien, de façon équivalente, la taille (le nombre de nœuds, internes et feuilles) de l'arbre binaire représentant l'expression régulière.

On peut également définir la *profondeur* d'une expression régulière comme la hauteur de l'arbre qui lui est associé.

Définition. On définit la profondeur $p(e)$ d'une expression régulière par induction :
- $p(\emptyset) = p(\epsilon) = 1$;
- $\forall s \in \Sigma, p(s) = 1$;
- si $e = f\star$, alors $p(e) = p(f) + 1$;
- si $e = f\,.\,g$, alors $p(e) = \max(p(f) + p(g)) + 1$;
- si $e = f\,|\,g$, alors $p(e) = \max(p(f) + p(g)) + 1$.

4.6 Expressions régulières linéaires

Définition. Une expression régulière e est dite *linéaire* lorsque tout symbole $s \in \Sigma$ apparaît une fois au plus dans e.

4.7 Expressions régulières étendues

Bien que les expressions régulières puissent être construites à partir des seuls constructeurs d'étoile, de concaténation et de choix, il peut être pratique d'introduire d'autres constructeurs pour simplifier l'écriture d'expressions régulières et d'en simplifier l'usage.

38. Nous verrons quelques outils pour ce faire dans le prochain chapitre.

Les constructeurs unaires + et ? sont notamment fréquemment utilisés. Si e est une expression régulière, l'expression régulière $e+$ est sémantiquement équivalente à l'expression $ee\star$ (au moins une répétition du motif identifié par e). Quant à l'expression régulière $e?$, elle est sémantiquement équivalent à $\epsilon\,|\,e$.

Il est assez fréquent de souhaiter désigner « un symbole s quelconque de l'alphabet Σ ». On ajoute alors généralement simplement le symbole Σ dans l'alphabet des expressions régulières (par exemple, aΣ est une expression régulière dont l'interprétation est le langage des mots de Σ^* de longueur 2 et commençant par a).

Dans le langage des expressions régulières d'OCaml[39], fourni par le module `Str`, le point « . » n'étant pas utilisé pour la concaténation (où il est omis), c'est ce caractère qui désigne « un quelconque $s \in \Sigma$ ». Il offre quelques autres raccourcis. Par exemple, `[abc]` est sémantiquement équivalent à `(a|b|c)`. Il est même possible de spécifier un ensemble de caractères, par exemple l'ensemble des minuscules à l'exception de `a` et `z` en écrivant `[b-y]`.

Le langage OCaml fournit ainsi quelques fonctions (qui ne sont pas à connaître) permettant de manipuler des expressions régulières. On les construit à partir d'une chaîne de caractères grâce à la fonction `Str.regexp`, laquelle retourne un objet de type `regexp`. Par exemple, pour l'expression régulière `a*b|ba*`, on écrira[40] :

```
# let r = Str.regexp "ab*\\|b*a";;
val r : Str.regexp = <abstr>
```

On peut ensuite par exemple demander s'il existe une sous-chaîne d'une chaîne de caractère donnée, débutant sur le caractère d'index `i`, qui soit un mot du langage associé à l'expression régulière, et le cas échéant le plus long de tels mots :

```
# Str.string_match r "abbab" 0;; (* On cherche un préfixe *)
- : bool = true

# Str.matched_string "abbab";;
- : string = "abb"

# Str.string_match r "bbb" 0;;
- : bool = false
```

Signalons enfin que certains outils utilisant les expressions régulières proposent également des constructeurs indiquant l'intersection, la différence et ou le complémentaire d'expressions régulières, opérations qui peuvent parfois être utiles dans la pratique, ou des mécanismes plus complexes. On parle alors d'expressions régulière *étendues*.

39. Similaire au langage utilisé par exemple dans l'éditeur Emacs.
40. Le symbole « | » doit être précédé d'un \ dans la chaîne fournie, mais ce même caractère doit être doublé lorsqu'il est entré entre guillemets.

5 Langages locaux

5.1 Définition

Considérons un alphabet Σ et un langage L sur cet alphabet. Posons :

- $P(L) = \{ s \in \Sigma \mid \{s\} \cdot \Sigma^* \cap L \neq \emptyset \}$, l'ensemble des premiers symboles des mots de L;
- $S(L) = \{ s \in \Sigma \mid \Sigma^* \cdot \{s\} \cap L \neq \emptyset \}$, l'ensemble des derniers symboles des mots de L;
- $N(L) = \{ ss' \in \Sigma^2 \mid \Sigma^* \cdot \{ss'\} \cdot \Sigma^* \cap L = \emptyset \}$, l'ensemble des mots de longueur 2 qui ne soient pas des facteurs de mots de L.

De façon évidente, on a [41] $L \setminus \{\varepsilon\} \subset (P(L)\Sigma^* \cap \Sigma^* S(L)) \setminus (\Sigma^* N(L) \Sigma^*)$. L'inclusion précédente indique que les mots de L sont les mots qui commencent par un symbole de P(L) *et* se terminent par un symbole de S(L) *et* ne contiennent *pas* de facteur présents dans N(L).

Un langage est dit *local* lorsque l'inclusion est une égalité. Plus précisément :

Définition. Un langage L sur un alphabet Σ est dit *local* s'il existe des parties P et S de Σ et une partie N de Σ^2 telles que

$$L \setminus \{\varepsilon\} = \left(P\Sigma^* \cap \Sigma^* S\right) \setminus \left(\Sigma^* N \Sigma^*\right)$$

Pour définir un langage local, il suffit donc de préciser Σ, P, S, N et l'appartenance ou non de ε au langage. Si un langage est local, alors de façon évidente P(L), S(L) et N(L) sont toujours un choix possible pour P, S et N (il peut en exister d'autres). Inversement, si P(L), S(L) et N(L) ne permettent pas d'obtenir l'égalité ci-dessus, alors le langage n'est pas local. Pour montrer qu'un langage L n'est pas local, il suffit donc d'exhiber un mot $w \in (P(L)\Sigma^* \cup \Sigma^* S(L)) \setminus (\Sigma^* N(L) \Sigma^*)$ qui n'appartienne pas à L.

Ainsi, sur l'alphabet $\Sigma = \{a, b\}$, le langage a^* des mots ne contenant que des a, est un langage local. On peut par exemple choisir $P = \{a\}$, $S = \{a\}$ et $N = \{ab, ba, bb\}$.

On vérifie ici que choix de P, S et N n'est pas unique : on aurait pu prendre $P = \{a, b\}$ et obtenir le même langage local. Ou bien $S = \{a, b\}$, ou bien encore $N = \{ab\}$, entre autres possibilités.

Sur ce même alphabet $\Sigma = \{a, b\}$, le langage $(ab)^*$ est également un langage local, puisque l'on peut satisfaire l'égalité en prenant $P = \{a\}$, $S = \{b\}$ et $N = \{aa, bb\}$.

En revanche, le langage $L = a^* ba^*$ n'est pas local. Par exemple, le mot ababa ne fait pas partie du langage, et il n'est pas possible de trouver des ensembles P, S et N qui l'excluent sans également exclure aba qui, lui, est un mot du langage. On remarquera enparticulier que $P(L) = S(L) = \{a, b\}$ et $N(L) = \{bb\}$ ne conviennent pas.

De tels langages nous seront utiles au chapitre suivant pour élaborer des outils capables de reconnaître un langage rationnel.

41. P(L), S(L) et N(L) sont naturellement des langages sur Σ^*.

Plutôt qu'un ensemble N des facteurs de longueur 2 que l'on ne doit *pas* trouver dans les mots de L, on préfère parfois [42] travailler avec un ensemble F regroupant les seuls facteurs de longueur 2 qu'il soit permis de trouver dans un mot de L.

On peut définir en particulier l'ensemble des mots de Σ^* de longueur 2 que l'on trouve comme facteurs parmi les mots de L :

$$F(L) = \{ ss' \in \Sigma^2 \mid \Sigma^* \cdot \{ss'\} \cdot \Sigma^* \cap L \neq \emptyset \}$$

Définition. Un langage L sur un alphabet Σ local s'il existe des parties P et S de Σ et une partie F de Σ^2 telles que

$$L \setminus \{\varepsilon\} = \left(P\Sigma^* \cap \Sigma^* S\right) \setminus \left(\Sigma^* \left(\Sigma^2 \setminus F\right) \Sigma^*\right)$$

Les deux définitions sont évidemment équivalentes, puisqu'il suffit de poser $F = \Sigma^2 \setminus N$ ou $N = \Sigma^2 \setminus F$ pour passer d'une définition à l'autre.

Si L est un langage local, alors P(L), S(L) et F(L) sont un choix possible (éventuellement parmi d'autres) pour P, S et F. Inversement, si $L \setminus \{\varepsilon\} \neq (P(L)\Sigma^* \cap \Sigma^* S(L)) \setminus \left(\Sigma^* \left(\Sigma^2 \setminus F(L)\right) \Sigma^*\right)$, alors L n'est pas local.

5.2 Reconnaissance

L'avantage des langages locaux (par rapport aux langages en général) est qu'il est très facile d'identifier si un mot donné en fait partie : il suffit en effet de vérifier ses premiers et derniers symboles, ainsi que tous ses facteurs de longueur 2 pour pouvoir conclure.

On pourrait par exemple écrire en Caml, pour un langage Σ correspondant aux caractères ASCII (et donc des mots sous la forme de chaînes de caractères), une fonction indiquant si un mot w non-vide est reconnu par un langage local défini [43] par P, S et N :

```
# let estUnMotDuLangage w p s n =
    let len = String.length w in
    let rec verifieFacteurs i =
      if i = len-1 then true else
        not (List.mem (String.sub w i 2) n) && verifieFacteurs (i+1)
    in len >= 1 && List.mem w.[0] p && List.mem w.[len-1] s
         && verifieFacteurs 0;;

val estUnMotDuLangage :
  string -> char list -> char list -> string list -> bool = <fun>
```

42. Les notations varient de façon importante d'un ouvrage à un autre, il conviendra de prendre garde à regarder si l'on a défini le langage local à partir des facteurs de deux lettres autorisés ou interdits !

43. Ici des listes de char/string, même si des dictionnaires donneraient une meilleure complexité.

La fonction précédente retourne `false` pour le mot vide. Celui-ci peut pourtant faire partie du langage. Pour l'inclure, on peut simplement remplacer les deux dernières lignes par les suivantes :

```
    in len = 0 || (List.mem w.[0] p && List.mem w.[len-1] s
                        && verifieFacteurs 0 );;
```

5.3 Concaténation, union, intersection et étoile de langages locaux

L'ensemble des langages locaux n'est pas stable pour la concaténation, ni pour l'union.

En effet, si les langages $L_1 = a^*b$ et $L_2 = ab^*$ sont des langages locaux, la concaténation de ces langages $L = L_1 L_2 = a^*bab^*$ n'est en revanche pas un langage local, puisque le langage local défini par $P(L) = S(L) = \{a, b\}$ et $N(L) = \emptyset$ correspond à Σ^*, plus grand que L (qui ne contient pas par exemple les mots a, b ou ab).

Il en est de même pour leur union $L' = a^*b + ab^*$. En effet, le langage local défini par $P(L') = S(L') = \{a, b\}$ et $N(L') = \{ba\}$ contient par exemple le mot aabb qui n'est pas un mot de L'.

Les choses se passent mieux avec l'intersection et la fermeture de Kleene.

Théorème 9. *Si L_1 et L_2 sont des langages locaux sur un alphabet Σ, alors leur intersection $L = L_1 \cap L_2$, est également un langage local.*

Démonstration. Posons $P = P(L_1) \cap P(L_2)$, $S = S(L_1) \cap S(L_2)$ et $N = N(L_1) \cup N(L_2)$.

On peut alors écrire

$$\begin{aligned}
L \setminus \{\varepsilon\} &= (L_1 \cap L_2) \setminus \{\varepsilon\} \\
&= (L_1 \setminus \{\varepsilon\}) \cap (L_2 \setminus \{\varepsilon\}) \\
&= \Big(\big(P(L_1)\Sigma^* \cap \Sigma^* S(L_1)\big) \setminus \big(\Sigma^* N(L_1)\Sigma^*\big)\Big) \cap \Big(\big(P(L_2)\Sigma^* \cap \Sigma^* S(L_2)\big) \setminus \big(\Sigma^* N(L_2)\Sigma^*\big)\Big) \\
&= \Big(\big(P(L_1)\Sigma^* \cap \Sigma^* S(L_1)\big) \cap \big(P(L_2)\Sigma^* \cap \Sigma^* S(L_2)\big)\Big) \setminus \Big(\big(\Sigma^* N(L_1)\Sigma^*\big) \cup \big(\Sigma^* N(L_2)\Sigma^*\big)\Big) \\
&= \big(P\Sigma^* \cap \Sigma^* S\big) \setminus \big(\Sigma^* N \Sigma^*\big)
\end{aligned}$$

Le langage $L_1 \cap L_2$ est donc bien un langage local. □

Théorème 10. *Si L est un langage local sur un alphabet Σ, alors sa fermeture de Kleene L^* est également un langage local.*

Démonstration. Il est évident que $P(L^*) = P(L)$ et $S(L^*) = S(L)$ (tout mot non-vide de L^* a pour préfixe et pour suffixe un mot de L, donc commence par un symbole de P(L) et se termine par un symbole de S(L)).

On a $F(L^*) = F(L) \cup S(L)P(L)$. En effet, aux facteurs présents dans les mots de L, il faut ajouter les « collages » lorsqu'un mot $w \in L^*$ est la concaténation $w = u_1 \cdot u_2 \cdot \ldots \cdot u_n$ de plusieurs mots $u_i \in L$. On trouve en effet les facteurs constitués du dernier symbole d'un mot u_i suivi du premier symbole du mot u_{i+1}.

On a donc l'inclusion

$$L^* \setminus \{\varepsilon\} \subset \big(P(L^*)\Sigma^* \cap \Sigma^* S(L^*)\big) \setminus \big(\Sigma^* N(L^*)\Sigma^*\big) \quad \text{où} \quad N(L^*) = \Sigma^2 \setminus F(L^*) = N(L) \setminus S(L)P(L)$$

Il nous faut à présent montrer l'inclusion inverse, soit montrer que tout mot non-vide $w \in (P(L^*)\Sigma^* \cap \Sigma^* S(L^*)) \setminus (\Sigma^* N(L^*)\Sigma^*)$ est un mot de L^*.

Pour ce faire, identifions tous les facteurs de longueur 2 de w qui sont dans S(L)P(L). Ils permettent de décomposer w en une concaténation de mots $w = u_1 \cdot u_2 \cdot \ldots \cdot u_n$, avec $u_i \in (P(L)\Sigma^* \cap \Sigma^* S(L)) \setminus (\Sigma^* N(L^*)\Sigma^*) \setminus (\Sigma^* S(L)P(L)\Sigma^*) \subset (P(L)\Sigma^* \cap \Sigma^* S(L)) \setminus (\Sigma^* N(L)\Sigma^*)$.

Mais puisque L est local, tous les mots u_i sont des mots de L, et par conséquent w est un mot de L^*. □

Si ni l'union, ni la concaténation de deux langages locaux ne sont nécessairement des langages locaux, on peut cependant montrer que c'est le cas lorsque l'on travaille avec des alphabets distincts :

Théorème 11. *Si L_1 et L_2 sont des langages locaux sur des alphabets Σ_1 et Σ_2 disjoints, alors*

- *leur concaténation $L = L_1 L_2$, définie sur l'alphabet $\Sigma = \Sigma_1 \cup \Sigma_2$, est un langage local ;*
- *leur union $L = L_1 + L_2$, définie sur l'alphabet $\Sigma = \Sigma_1 \cup \Sigma_2$, est un langage local.*

Démonstration. Pour le cas de la concaténation $L = L_1 L_2$, on a :

$$\begin{cases}
P(L_1 L_2) = P(L_1) \quad \text{si} \quad \varepsilon \notin L_1 \quad \text{sinon} \quad P(L_1) \cup P(L_2) \\
S(L_1 L_2) = S(L_2) \quad \text{si} \quad \varepsilon \notin L_2 \quad \text{sinon} \quad S(L_1) \cup S(L_2) \\
F(L_1 L_2) = F(L_1) \cup F(L_2) \cup S(L_1)P(L_2)
\end{cases}$$

On a $L_1 L_2 \subset (P(L_1 L_2)\Sigma^* \cap \Sigma^* S(L_1 L_2)) \setminus \big(\Sigma^*\big(\Sigma^2 \setminus F(L_1 L_2)\big)\Sigma^*\big)$.

Pour montrer que $L_1 L_2$ est local, il nous faut montrer l'égalité, et donc l'inclusion du second membre dans le premier.

Considérons donc un mot non-vide $w = s_1 s_2 s_3 \ldots s_n$ (où les s_i sont des symboles de Σ) quelconque de l'ensemble $(P(L_1 L_2)\Sigma^* \cap \Sigma^* S(L_1 L_2)) \setminus \big(\Sigma^*\big(\Sigma^2 \setminus F(L_1 L_2)\big)\Sigma^*\big)$.

- Si le mot w ne contient que des symboles de Σ_1, alors son premier caractère est nécessairement dans l'ensemble $P(L_1 L_2) \cap \Sigma_1$, son dernier caractère dans l'ensemble $S(L_1 L_2) \cap \Sigma_1$ et ses facteurs de longueur 2 tous dans l'ensemble $F(L_1 L_2) \cap \Sigma_1^2$.

  L'ensemble $S(L_1 L_2) \cap \Sigma_1$ n'est donc pas vide, ce qui implique, puisque les alphabets sont disjoints, que $\varepsilon \in L_2$ et $S(L_1 L_2) \cap \Sigma_1 = S(L_1)$.

  On a également $P(L_1 L_2) \cap \Sigma_1 = P(L_1)$ et $F(L_1 L_2) \cap \Sigma_1^2 = F(L_1)$. Puisque L_1 est local, w est nécessairement un mot de L_1. Et puisque $\varepsilon \in L_2$, on a bien $w \in L_1 L_2$.
- Le même raisonnement peut être tenu si w ne contient que des symboles de Σ_2.
- Sinon, il existe nécessairement deux mots $u \in \Sigma_1^*$ et $v \in \Sigma_2^*$ tels que $w = u \cdot v$.

  En effet, les facteurs de longueur 2 de $s_i s_{i+1}$ sont soit dans $F(L_1)$ (donc les deux symboles sont dans Σ_1), soit dans $F(L_2)$ (donc les deux symboles sont dans Σ_2) soit dans $S(L_1)P(L_2)$ (le premier symbole dans Σ_1, le second dans Σ_2).

  Il suffit donc de poser $k = \min\{ i \in [\![1 \,..\, n]\!] \mid s_i \in \Sigma_1 \quad \text{et} \quad s_{i+1} \in \Sigma_2 \}$ (ensemble non-vide par hypothèse), et on montre aisément que $s_i \in \Sigma_1$ si $i \leqslant k$ et $s_i \in \Sigma_2$ si $i > k$. On peut donc prendre $u = s_1 \ldots s_k$ et $v = s_{k+1} \ldots s_n$.

  On a $s_1 \in P(L_1 L_2) \cap \Sigma_1 = P(L_1)$, $s_k \in S(L_1)$ (puisque $s_k s_{k+1}$ appartient à l'ensemble $F(L_1 L_2) \cap \Sigma_1 \Sigma_2 = S(L_1)P(L_2)$), et pour $i < k$, $s_i s_{i+1} \in F(L_1 L_2) \cap \Sigma_1^2 = F(L_1)$.

  Puisque L_1 est local, on a $u \in L_1$. De même, $v \in L_2$, et donc $w = u \cdot v \in L_1 L_2$.

Dans tous les cas, $w \in L_1 L_2$, donc $L_1 L_2$ est bien un langage local. □

Démonstration. Pour le cas de l'union $L = L_1 + L_2$, on a :

$$\begin{cases} P(L_1 + L_2) = P(L_1) \cup P(L_2) \\ S(L_1 + L_2) = S(L_1) \cup S(L_2) \\ F(L_1 + L_2) = F(L_1) \cup F(L_2) \end{cases}$$

On a $L_1 + L_2 \subset (P(L_1 + L_2)\Sigma^* \cup \Sigma^* S(L_1 + L_2)) \setminus \left(\Sigma^* \left(\Sigma^2 \setminus F(L_1 + L_2)\right)\Sigma^*\right)$.

Pour montrer que $L_1 + L_2$ est local, il nous faut montrer l'égalité, et donc l'inclusion du second membre dans le premier.

Considérons un mot non-vide $w = s_1 s_2 s_3 \ldots s_n$ (où les s_i sont des symboles de Σ) quelconque de l'ensemble $(P(L_1 + L_2)\Sigma^* \cap \Sigma^* S(L_1 + L_2)) \setminus \left(\Sigma^* \left(\Sigma^2 \setminus F(L_1 + L_2)\right)\Sigma^*\right)$.

- Si $s_1 \in \Sigma_1$, il est aisé de montrer de proche en proche que tous les s_i sont dans Σ_1 puisque les $s_i s_{i+1}$ ne peuvent être que dans $F(L_1)$ (tous les éléments de $F(L_2)$ commencent par un symbole de Σ_2 puisque les alphabets sont disjoints).

  Donc $s_1 \in P(L_1)$, $s_n \in S(L_1)$ et pour tout i, $s_i s_{i+1} \in F(L_1)$. Puisque L_1 est un langage local, on a $w \in L_1$.
- Sinon, $s_1 \in L_2$, et par un raisonnement similaire, on a $w \in L_2$.

On a donc $w \in L_1 + L_2$. □

5.4 Langages locaux et expressions régulières linéaires

Théorème 12. *L'interprétation de toute expression régulière linéaire est un langage local.*

Démonstration. Raisonnons une fois encore par induction structurelle.

- l'expression régulière ∅ s'interprète en un langage vide, qui est un langage local ;
- l'expression régulière ϵ s'interprète en un langage réduit au mot vide, qui est un langage local ;
- l'expression régulière s où $s \in \Sigma$ s'interprète en un langage $\{s\}$ réduit au seul mot s, qui est un langage local (il suffit de prendre $P = S = \{s\}$ et $N = \varnothing$, ce qui interdit tout mot de longueur supérieure ou égale à deux) ;
- si e* est une expression régulière linéaire, alors e l'est également, et s'interprète donc, par induction, en un langage local, dont l'étoile est également un langage local ;
- si e.f est une expression régulière linéaire, e et f le sont également, et donc s'interprètent en des langages locaux, et la concaténation de deux langages locaux sur des alphabets distincts (un symbole présent dans e ne peut pas être présent dans f et inversement) est également un langage local ;
- si e|f est une expression régulière linéaire, e et f le sont également, et donc s'interprètent en des langages locaux, et l'union de deux langages locaux sur des alphabets distincts est également un langage local. □

Notons que la réciproque est fausse : si le langage $L =$ `aa`* est local, il n'existe pas d'expression régulière linéaire dont l'interprétation est L.

5.5 Implémentation en Caml

Nous allons, enfin, brièvement présenter une façon de créer un type Caml correspondant à une expression régulière [44] ne contenant pas le symbole ∅ et quelques fonctions permettant de manipuler des expressions régulières ainsi représentées.

On choisira pour alphabet Σ l'ensemble des caractères que Caml peut manipuler (des éléments de type `char`). Σ^* correspond donc aux chaînes de caractères (des éléments de type `string`).

44. Notre but n'est pas ici de développer un outil pour gérer les expressions régulières, lequel existe déjà dans le module `Str`, mais simplement d'illustrer une façon de les manipuler formellement en Caml.

On peut donc construire notre type représentant les expressions régulières par induction structurelle :

```
type regexp =
  | Epsilon
  | Symbole of char
  | Etoile of regexp
  | Concatenation of regexp * regexp
  | Choix of regexp * regexp;;
```

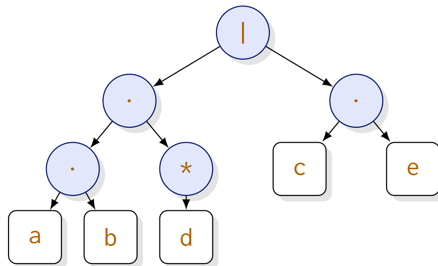
Pour faciliter les choses, nous allons toutefois utiliser un type légèrement différent, s'éloignant un peu de la définition originale des expressions régulières :

```
type regexp =
  | Constante of string
  | Etoile of regexp
  | Concatenation of regexp * regexp
  | Choix of regexp * regexp;;
```

Cela nous permettra en effet d'utiliser « `Constante "abc"` » plutôt que d'avoir à écrire « `Concatenation (Symbole 'a') (Concatenation (Symbole 'b') (Symbole 'c')` ».

On remarquera la disparition de `Epsilon` dans la définition. En effet, on pourra se servir de « `Constante ""` » pour désigner ε.

Avec un type ainsi défini, l'expression régulière `abd*|ce`, correspondant à l'arbre ci-dessous :



Peut donc être représentée en Caml par :

```
# let r = Choix(Concatenation(Constante "ab", Etoile(Constante "d")),
                Constante "ce");;

val r : regexp =
  Choix (Concatenation (Constante "ab", Etoile (Constante "d")),
   Constante "ce")
```

Une fois une expression régulière représentée par un arbre, il devient plus aisé de travailler dessus, et d'en extraire des informations. Comme souvent, les fonctions travailleront par induction structurelle. Par exemple, pour savoir si l'expression régulière s'interprète en un langage contenant le mot-vide ε, on peut écrire :

```
# let rec contientMotVide = function
    | Constante "" -> true                         (* ε ∈ ℒ(ε) *)
    | Constante _ -> false
    | Etoile e -> true                             (* ε ∈ ℒ(e*) *)
    | Concatenation (e,f)
        -> contientMotVide e        (* ε ∈ ℒ(ef) ⟺ ε ∈ ℒ(e) et ε ∈ ℒ(f) *)
         && contientMotVide f
    | Choix (e,f)
        -> contientMotVide e        (* ε ∈ ℒ(e|f) ⟺ ε ∈ ℒ(e) ou ε ∈ ℒ(f) *)
         || contientMotVide f;;

val contientMotVide : regexp -> bool = <fun>
```

Plaçons-nous à présent dans le cadre des expressions régulières *linéaires*, c'est-à-dire, rappelons-le, les expressions régulières dans lesquelles chaque symbole de Σ apparaît au plus une unique fois.

Pour une expression régulière linéaire [45] e, on peut écrire des fonctions qui permettent de déterminer $\mathrm{P}(\mathscr{L}(e))$:

```
# let rec prefixes = function
    | Constante "" -> []
    | Constante ch -> [ ch.[0] ]
    | Etoile e -> prefixes e
    | Concatenation (e,f) when contientMotVide e
        -> prefixes e @ prefixes f
    | Concatenation (e,f) -> prefixes e
    | Choix (e,f) -> prefixes e @ prefixes f;;

val prefixes : regexp -> char list = <fun>
```

45. Une fonction similaire peut déterminer $\mathrm{P}(\mathscr{L}(e))$ pour une expression régulière non nécessairement linéaire, mais il faudra éliminer les doublons qui apparaissent dans la liste lors de sa construction. Il en est de même pour les fonctions suivantes.

Il est aisé de faire de même avec $S(\mathcal{L}(e))$:

```
# let rec suffixes = function
    | Constante "" -> []
    | Constante ch -> [ ch.[String.length ch - 1] ]
    | Etoile e -> suffixes e
    | Concatenation (e,f) when contientMotVide f
      -> suffixes e @ suffixes f
    | Concatenation (e,f) -> suffixes f
    | Choix (e,f) -> suffixes e @ suffixes f;;

val suffixes : regexp -> char list = <fun>
```

Essayons ces fonctions sur notre expression régulière `abd*|ce` pour déterminer $P(\mathcal{L}(e))$ et $S(\mathcal{L}(e))$:

```
# prefixes r;;
- : char list = ['a'; 'c']

# suffixes r;;
- : char list = ['b'; 'd'; 'e']
```

Pour déterminer l'ensemble $F(\mathcal{L}(e))$ des facteurs de deux symboles que l'on peut trouver dans $\mathcal{L}(e)$, il nous faut d'abord écrire une fonction construisant le produit cartésien de deux listes de symboles (`char list`) :

```
# let rec produit = function
    | ([], _) -> []
    | (_, []) -> []
    | (t::q, lst) -> let t = String.make 1 t in
                     List.map (fun c -> (t ^ String.make 1 c)) lst
                       @ (produit (q, lst));;

val produit : char list * char list -> string list = <fun>
```

On peut alors construire $F(\mathcal{L}(e))$, une fois encore, par induction structurelle :

- pour les « constantes », on détermine tous les facteurs qu'elles contiennent (ici avec une fonction récursive) ;
- pour $e*$, on détermine les facteurs de e et les éléments de $S(\mathcal{L}(e))P(\mathcal{L}(e))$;
- pour $e\,.\,f$, on détermine les facteurs de e, de f, et les éléments de $S(\mathcal{L}(e))P\big(\mathcal{L}(f)\big)$;
- pour $e\,|\,f$, on détermine les facteurs de e et de f.

Cela donne par exemple :

```
# let rec facteurs2 = function
    | Constante ch when String.length ch < 2
      -> []
    | Constante ch
      -> List.init (String.length ch - 1)
           (fun i -> String.sub ch i 2)
    | Etoile e
      -> (facteurs2 e) @ (produit (suffixes e, prefixes e))
    | Concatenation (e,f)
      -> (facteurs2 e) @ ((facteurs2 f)
           @ (produit (suffixes e, prefixes f)))
    | Choix (e,f)
      -> (facteurs2 e) @ (facteurs2 f);;

val facteurs2 : regexp -> string list = <fun>
```

Ce qui donne, toujours avec `abd*|ce` :

```
# facteurs2 r;;

- : string list = ["ab"; "dd"; "bd"; "ce"]
```

Ayant déterminé la présence ou non de ε dans le langage et les ensembles P, S et F, on dispose donc de tous les éléments pour transformer automatiquement une expression régulière linéaire (sous la forme d'un arbre) en une fonction capable de reconnaître le langage associé.

Dans le prochain chapitre, nous porterons cette idée plus loin, afin d'obtenir aisément et systématiquement, pour une expression régulière quelconque, une fonction reconnaissant le langage correspondant à son interprétation.

2 Automates finis

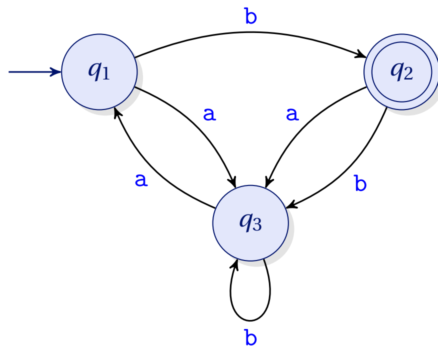
1 Introduction

Dans ce second chapitre, nous nous intéresserons aux *automates finis*, un outil permettant, entre autres usages, de « reconnaître » des séquences de symboles, autrement dit d'identifier si mot fait partie ou non d'un langage donné.

Nous verrons notamment comment il est possible de traduire une expression régulière en un tel automate, qui à son tour pourra être traduit algorithmiquement. On disposera ainsi d'une méthode systématique pour transformer un langage régulier en algorithme permettant d'en identifier les mots.

Un automate fini sur un alphabet Σ est une « machine à états [1] » qui peut être représentée par un graphe orienté, dont les arcs sont étiquetés par les symboles $s \in \Sigma$.

En voici un exemple, sur $\Sigma = \{\texttt{a}, \texttt{b}\}$:

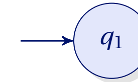


Il s'agit en fait d'un *multigraphe*, dans lequel
- plusieurs arcs peuvent exister entre deux états (par exemple de q_2 à q_3 ci-dessus) ;
- un arc peut mener d'un état à lui-même (de l'état q_3 à ce même état q_3 dans notre exemple).

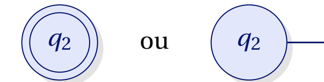
On remarquera par ailleurs une autre particularité des automates : certains sommets jouent un rôle particulier dans le graphe.

1. Nous reviendrons progressivement sur cette dénomination, mais il s'agit d'un dispositif qui sera, à un instant donné, dans un (ou plusieurs) « états », et peut évoluer au cours du temps.

On distingue :
- des sommets dit *initiaux*, signalés par une flèche pointant vers le sommet en question ;



- des sommets dit *terminaux* (ou *acceptants*), signalés par un double trait bordant le sommet (ou, dans certains ouvrages, par une flèche sortant du sommet en question et ne pointant pas vers un autre sommet).



Un même sommet peut être à la foit initial et terminal. Nous nous pencherons sur la signification de ces attributs par la suite.

2 Automates finis déterministes

2.1 Définitions

Définition. Un *automate fini déterministe* sur un alphabet Σ est la donnée d'un quadruplet (Q, q_0, F, δ) composé de
- un ensemble fini Q, correspondant aux *états* de l'automate ;
- un état *initial* $q_0 \in Q$;
- un ensemble d'états *terminaux* (ou *finaux*) $F \subset Q$;
- une *fonction de transition* δ, une application d'une partie de $Q \times \Sigma$ vers Q.

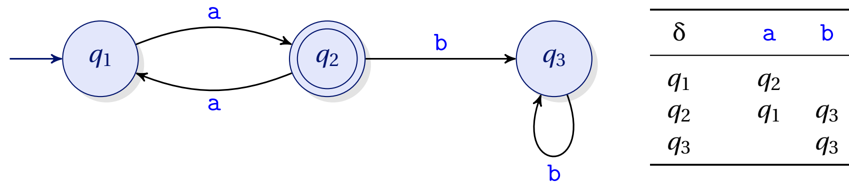
Le graphe présenté précédemment est la représentation de l'automate fini déterministe sur $\Sigma = \{a, b\}$ associé au quadruplet $(\{q_1, q_2, q_3\}, q_1, \{q_2\}, \delta)$ où la fonction de transition δ est définie par la table d'association suivante :

δ	a	b
q_1	q_3	q_2
q_2	q_3	q_2
q_3	q_1	q_2

On notera qu'il y a, dans le cas d'un automate fini déterministe (nous reviendrons sur ce qualificatif ultérieurement), un *unique* sommet initial, et qu'il n'y a jamais plusieurs arcs étiquetés avec un même symbole $s \in \Sigma$ partant d'un quelconque sommet. En revanche, la fonction de transition δ peut très bien ne pas être définie pour tout couple $(q, s) \in Q \times \Sigma$.

Définition. Un *blocage* d'un automate (Q, q_0, F, δ) sur un alphabet Σ est un couple $(q, s) \in Q \times \Sigma$ pour lequel la fonction de transition δ n'est pas définie. Un automate sans blocage est dit *complet*

L'automate précédent est un automate complet. Il n'en est pas de même pour l'automate $(\{q_1, q_2, q_3\}, q_1, \{q_2\}, \delta)$ ci-dessous, où $(q_1, \texttt{b})$ et $(q_3, \texttt{a})$ sont des blocages :



δ	a	b
q_1	q_2	
q_2	q_1	q_3
q_3		q_3

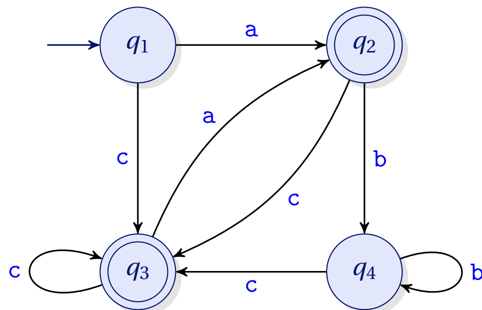
Définition. Un automate fini déterministe (Q, q_0, F, δ) est dit *standard* s'il n'existe pas de couple $(q, s) \in Q \times \Sigma$ tel que $\delta(q, s) = q_0$

En d'autres termes, un automate fini déterministe est dit standard s'il n'existe pas, dans le graphe représentant l'automate, d'arc pointant vers le sommet correspondant à l'état initial q_0 (en d'autres termes, que le degré entrant du sommet q_0 est nul [2]). Aucun des deux automates précédemment présenté n'est standard.

Définition. Un automate fini déterministe (Q, q_0, F, δ) est dit *local* si, pour tout $s \in \Sigma$, il existe un état $q \in Q$ tel que, $\forall q' \in Q$, (q', s) est un blocage ou $\delta(q', s) = q$.

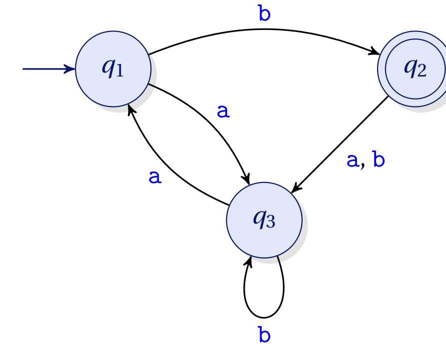
Dans le graphe représentant un automate fini déterministe local, tous les arcs étiquetés par le même symbole s doivent donc pointer vers un même sommet q. Là encore, aucun des deux automates précédents n'est local.

Voici un exemple d'automate fini déterministe sur $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}\}$ à la fois standard et local [3] :



2. La flèche pointant vers q_0 ne compte pas : elle marque le caractère initial de q_0, il ne s'agit pas d'un arc.
3. Il n'est en revanche pas complet, on peut dénombrer quatre cas de blocage, un pour chacun des états.

Si l'alphabet Σ contient de nombreux symboles, le graphe associé à un automate sur Σ peut rapidement comporter un très grand nombre d'arcs. Pour éviter qu'il soit trop chargé, on étiquète parfois un arc par un ensemble de symboles plutôt que par un unique symbole, s'il existe plusieurs arcs menant d'un sommet associé à un état q à un sommet associé à un état q'. Par exemple, le graphe associé au premier automate présenté peut être simplifié de la sorte :



2.2 Langage reconnu

Définition. Soit un automate fini déterministe (Q, q_0, F, δ) sur un alphabet Σ.

La fonction de transition étendue aux mots $\delta^* : Q \times \Sigma^* \mapsto Q$ est définie récursivement par :

$$\begin{cases} \text{pour tout } q \in Q,\ \delta^*(q, \varepsilon) = q; \\ \text{pour tous } q, s, w \in Q \times \Sigma \times \Sigma^*,\ \delta^*(q, s \cdot w) = \delta^*(\delta(q, s), w) \end{cases}$$

Dans un automate incomplet, $\delta^*(q, w)$ peut ne pas être défini si l'on rencontre un blocage. Dans le cas contraire, l'état $\delta^*(q, w)$ est donc l'unique état auquel on parvient en partant de l'état q et en égrennant les symboles de w.

Par exemple, dans le cas du premier automate cité, $\delta^*(q_1, \texttt{ababa}) = q_3$. En effet, on effectue les transitions suivantes :

$$q_1 \xrightarrow{\texttt{a}} q_3 \xrightarrow{\texttt{b}} q_3 \xrightarrow{\texttt{a}} q_1 \xrightarrow{\texttt{b}} q_2 \xrightarrow{\texttt{a}} q_3$$

Définition. La suite de transitions définie par $\delta^*(q, w)$ est appelée *chemin*.

Le mot $w \in \Sigma^*$ est l'*étiquette* de ce chemin.

Définition. Soit un automate fini déterministe (Q, q_0, F, δ) sur un alphabet Σ.

Un chemin partant de l'état initial q_0 est dit *acceptant* si l'état d'arrivée est un état terminal.

Un mot $w \in \Sigma^*$ est *reconnu* par l'automate si et seulement si ce mot est l'étiquette d'un chemin acceptant, c'est-à-dire si $\delta^*(q_0, w) \in F$.

Le *langage reconnu* par l'automate est l'ensemble des mots de Σ^* reconnus par l'automate, soit le langage

$$L = \{ w \in \Sigma^* \mid \delta^*(q_0, w) \in F \}$$

Dans notre exemple, `ababa` n'est pas un mot reconnu, mais les mots `b`, `abab` ou `abababab` le sont.

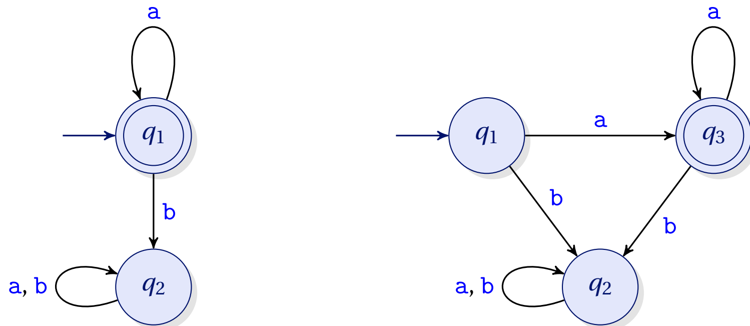
Notons que dans le cas d'un automate incomplet, s'il n'existe pas de chemin dont l'étiquette est $w \in \Sigma^*$ car on passe par un blocage de l'automate, alors le mot w n'est pas reconnu par l'automate. Il en sera naturellement de même pour tout mot $w' \in \Sigma^*$ dont w est un préfixe.

Définition. Un langage est dit *reconnaissable* s'il existe un automate dont il est le langage reconnu.

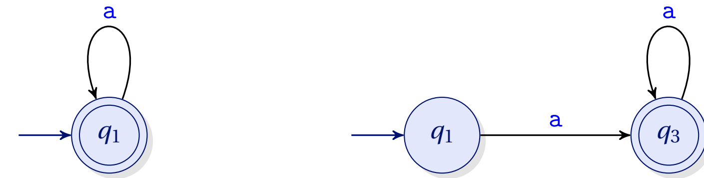
Puisque l'ensemble des automates est dénombrable mais pas l'ensemble des langages, tous les langages ne sont pas reconnaissables.

2.3 Quelques exemples

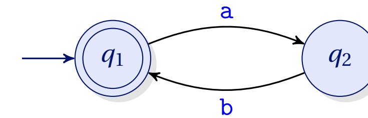
On se place sur le langage $\Sigma = \{$`a`, `b`$\}$. Les deux automates finis déterministes ci-dessous reconnaissent respectivement les langages décrits par les expressions régulières `a*` (langage des mots ne contenant que des `a`) et `aa*` (langage des mots non-vides ne contenant que des `a`).



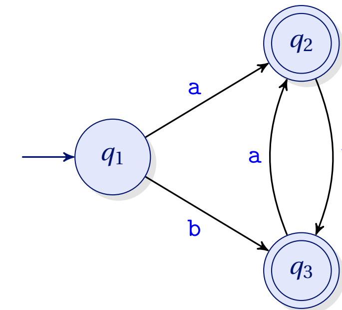
Les deux automates précédents sont complets (le second est également standard). On peut reconnaître les mêmes langages avec des automates incomplets :



L'automate ci-dessous reconnaît le langage décrit par l'expression régulière `ab*` des mots constitués d'un nombre quelconque de répétition de `ab` :



Enfin, l'automate ci-dessous reconnaît les mots non vides[4] de Σ^* qui ne contiennent pas de facteurs `aa` ou `bb`[5] :



2.4 Des automates aux expressions régulières

On pourra être amené à vouloir déterminer une expression régulière dont l'interprétation correspond au langage reconnu par un automate fini déterministe[6]. Il exite pour cela différentes méthodes. Une approche possible utilise un résultat sur les langages appelé lemme d'Arden[7].

4. Pour que cet automate reconnaisse également les mots vides, il suffit de rendre l'état q_0 terminal.
5. On pourra remarquer que ce langage est local, et que l'automate est lui aussi un automate local. Ce n'est pas une coïncidence, comme nous le verrons un peu plus loin.
6. Il est possible de montrer qu'il existe toujours une telle expression régulière, nous y reviendrons.
7. Ce lemme ne figure pas au programme, aussi n'est-il pas possible de l'utiliser dans un concours sans le redémontrer au préalable.

Lemme 4 (lemme d'Arden)**.** *Soient* A *et* B *deux langages sur un même alphabet* Σ. *L'équation* $L = AL + B$ *d'inconnue le langage* L *admet* $L = A^* B$ *comme plus petite solution (pour l'inclusion).*
Si $\varepsilon \notin A$, *alors la solution* $L = A^* B$ *est unique.*

Démonstration. Puisque $L = AL + B$, on peut écrire $L = A^2 L + AB + B$. Par une récurrence immédiate, on a, pour tout entier strictement positif n,

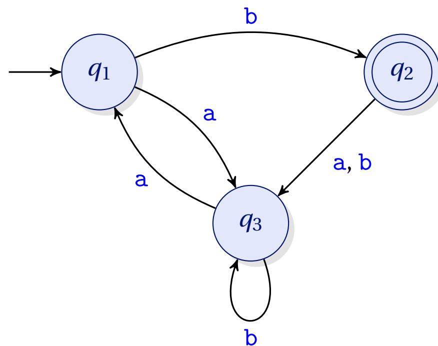
$$L = A^n L + \sum_{i=0}^{n-1} A^i B$$

Pour tout entier i positif, tout mot de $A^i B$ est donc un mot de L. On a donc $A^* B \subset L$.
Si $\varepsilon \notin A$, pour montrer que l'inclusion devient une égalité, considérons un mot $w \in L$.
On a $L = A^{|w|+1} L + \sum_{i=0}^{|w|} A^i B$.
Or les mots de $A^{|w|+1} L$ ont nécessairement au moins une longueur $|w| + 1$, donc

$$w \in \sum_{i=0}^{|w|} A^i B \subset A^* B \qquad \text{et donc} \qquad L \subset A^* B \qquad \text{d'où} \quad L = A^* B$$ □

Pour déterminer le langage reconnu par un automate fini déterministe, on peut noter L_i le langage des étiquettes des chemins **débutant sur l'état** q_i et se terminant sur un état acceptant. On peut alors obtenir un système d'équations [8] sur les langages L_i.

Par exemple, pour notre premier automate :



$$\begin{cases} L_1 = \{a\}L_3 + \{b\}L_2 \\ L_2 = \{\varepsilon\} + \{a, b\}L_3 \\ L_3 = \{a\}L_1 + \{b\}L_3 \end{cases}$$

D'après le lemme d'Arden, on a donc $L_3 = \{b\}^*\{a\}L_1$.

Par substitution, $L_2 = \{\varepsilon\} + \{a, b\}\{b\}^*\{a\}L_1$.

8. Dans ces équations, on rappelle que, par exemple, $\{a\}L_1$ doit se comprendre comme la concaténation du langage ne contenant que le mot a et du langage L_1, donc le langage des mots de L_1 précédés d'un a supplémentaire. Cette écriture étant un peu lourde, il n'est pas rare de mélanger dans de tels raisonnements les langages et les expressions régulières les représentant.

Ce qui conduit à $L_1 = \{a\}\{b\}^*\{a\}L_1 + \{b\} + \{b\}\{a, b\}\{b\}^*\{a\}L_1$.

Et finalement, $L_1 = (\{a\}\{b\}^*\{a\} + \{b\}\{a, b\}\{b\}^*\{a\})^*\{b\}$.

Puisque l'état initial est q_1, le langage reconnu par l'automate est L_1, et est décrit par l'expression régulière `(ab*a|b(a|b)b*a)*b`.

2.5 Des expressions régulières vers les automates

Plus fréquemment encore [9], on peut s'intéresser au problème inverse, à savoir, pour une expression régulière e quelconque, comment on peut construire un automate reconnaissant le langage $L = \mathcal{L}(e)$ correspondant à son interprétation.

Pour l'instant, on se contentera d'étudier le cas particulier des expressions régulières *linéaires*, dont nous avons établi au chapitre précédent qu'elles s'interprétaient en un langage local. Et il est aisé de construire un automate fini déterministe reconnaissant un tel langage :

Théorème 13. *Tout langage local est reconnaissable par un automate local.*

Démonstration. Considérons un langage local L sur un alphabet Σ caractérisé par des ensembles $P \subset \Sigma$, $S \subset \Sigma$ et $N \subset \Sigma^2$.

Construisons l'automate fini déterministe (Q, q_0, F, δ) sur Σ de la façon suivante :

- $Q = \{q_0\} \cup \{q_s \mid s \in \Sigma\}$;
- $F = \{q_s \mid s \in S\}$ si $\varepsilon \notin L$, et $F = \{q_0\} \cup \{q_s \mid s \in S\}$ sinon ;
- pour tout $s \in P$, $\delta(q_0, s) = q_s$;
- pour tout $ss' \in \Sigma^* \setminus N$, $\delta(q_s, s') = q_{s'}$.

Par construction, on a bien un automate local, qui est également standard.

Pour montrer qu'il reconnaît le langage L, considérons un mot non-vide $w = s_1 s_2 \ldots s_n$. Ce mot est reconnu par l'automate si et seulement si :

- $s_1 \in P$ (pas de blocage, dans l'état q_0, pour le symbole s_1) ;
- pour tout $k \in [\![1 \mathinner{..} n-1]\!]$, $s_k s_{k+1} \notin N$ (pas de blocage dans l'état q_{s_k} pour s_{k+1}) ;
- $s_n \in F$.

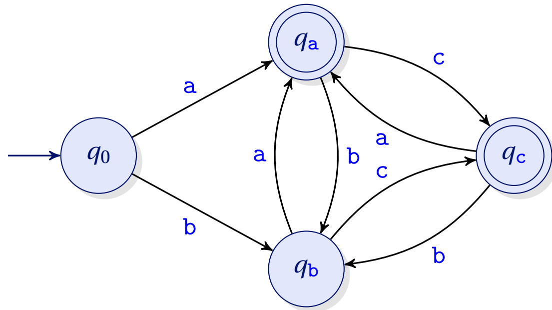
Autrement dit, le mot est reconnu si et seulement si $w \in L$.

Le mot vide ε correspond à l'unique chemin qui se termine sur l'état q_0 (puisque l'automate est standard), et donc $\varepsilon \in L$ si et seulement si $q_0 \in F$. □

Par conséquent, toute expression régulière linéaire s'interprète en un langage reconnaissable par un automate local, qu'il est aisé de construire. Par exemple, le langage local L des mots non vides sur $\Sigma = \{a, b, c\}$ qui commencent par a ou bien b, se terminent par

9. Par exemple, dans un logiciel, l'utilisateur peut entrer une expression régulière pour effectuer une recherche, et on souhaite construire l'automate pour mettre cette recherche en œuvre.

a ou bien c, et n'ont pas deux symboles successifs identiques (soit P = {a, b}, F = {a, c}, N = {aa, bb, cc}) peut être reconnu par l'automate fini déterministe local suivant [10] :



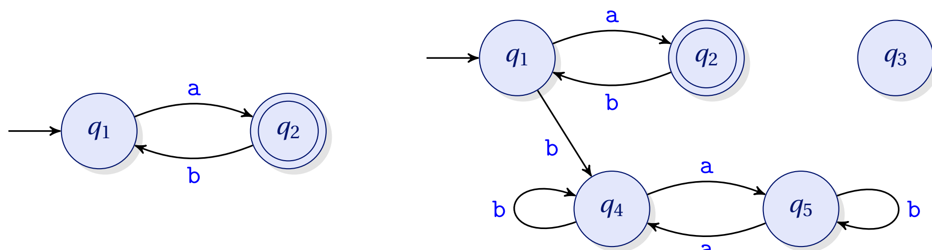
Nous verrons plus loin qu'il y a, en fait, une équivalence entre les expressions régulières et les automates finis déterministes, les deux notions permettant de reconnaître exactement les mêmes langages.

2.6 Transformations des automates

Émondage

On l'a vu dans les exemples présentés, de nombreux automates fini déterministes reconnaissent le même langage. On peut en effet aisément ajouter des états supplémentaires, ou bien, à partir d'un cas de blocage, ajouter une transition vers un état à partir duquel il n'existe pas de chemin acceptant.

Les automates ci-dessous reconnaissent ainsi le même langage correspondant à l'interprétation de l'expression régulière a(ba)* :



Définition. Deux automates finis reconnaissant un même langage sont dits *équivalents*.

Il y a de nombreuses raisons de vouloir un automate le plus simple possible. Par exemple, sa représentation, dans la mémoire d'un ordinateur, occupera moins de place. En outre, il est intéressant, lorsque l'on essaie de vérifier qu'un mot n'appartient pas au langage reconnu par l'automate, de rencontrer un blocage de l'automate le plus tôt possible, puis qu'il permet de conclure directement.

Il est donc souhaitable de supprimer autant d'états et de transitions que possible, tant que ces suppressions n'ont pas de conséquence le langage reconnu. Une telle opération est appelée *émondage* de l'automate.

Définition. Soit un automate fini déterministe (Q, q_0, F, δ) sur un alphabet Σ.

Un état $q \in Q$ est dit *accessible* s'il existe un chemin menant de q_0 à q.

Un état $q \in Q$ est dit *co-accessible* s'il existe un chemin menant de q à un état $q' \in F$.

Un état $q \in Q$ est dit *utile* s'il est à la fois accessible et co-accessible.

Un automate ne contenant que des états utiles est dit *émondé*

Précisons dès à présent que les notions d'automates équivalents, d'états accessibles, co-accesibles et utiles, et d'automates émondés restent valables dans le cadre des automates fini non-déterministes que nous étudierons un peu plus loin.

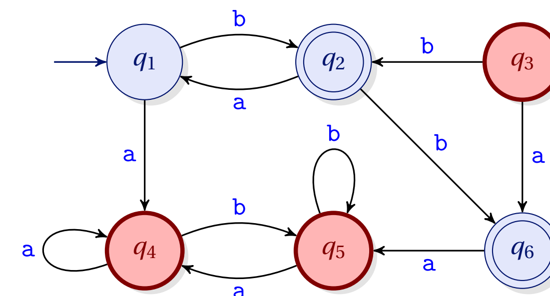
Théorème 14. *Soit un automate fini déterministe* (Q, q_0, F, δ) *sur un alphabet* Σ.

On en change pas le langage reconnu par l'automate en supprimant tous les états qui ne sont pas utiles, et les transitions les concernant.

Démonstration. Tout mot reconnu par un automate est en effet l'étiquette d'un chemin menant de l'état initial q_0 à un état final $q \in F$. Ce chemin ne passe donc que par des états q' utiles, puisqu'ils sont nécessairement accessibles et co-accessibles.

L'automate obtenu par la suppression des états qui ne sont pas utiles reconnaîtra donc toujours les mêmes mots, et aucun mot supplémentaire. Il est donc équivalent à l'automate initial. □

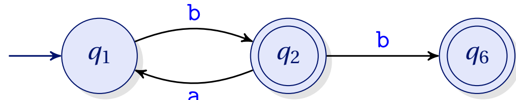
Considérons par exemple l'automate fini déterministe ci-dessous :



10. Si l'on souhaite également reconnaître le mot vide, il suffit de rendre q_0 terminal.

L'état q_3 n'est pas accessible, les états q_4 et q_5 ne sont pas co-accessibles. Seuls les états q_1, q_2 et q_6 sont donc des états utiles. On peut émonder l'automate en ne conservant que ces trois états.

Après simplification, on obtient l'automate émondé ci-dessous :



On peut aisément montrer que ces automates reconnaissent le langage décrit par l'expression régulière `b(ab)*(ϵ|b)`.

Notons qu'il est aisé d'automatiser l'émondage d'un automate : puisque l'on peut considérer un automate comme un graphe orienté, les états accessibles sont les sommets du graphe que l'on peut atteindre à partir d'une exploration depuis l'état initial q_0. De même, les états co-accessibles sont ceux que l'on peut atteindre grâce à une exploration depuis les états $q \in F$, *après avoir retourné les arcs du graphe*[11].

Complétion

Si en général on cherche à construire les automates les plus simples possibles, il peut être utile, dans certaines situations, de disposer d'un automate avec certaines propriétés particulières : automate complet, automate standard, etc.

Théorème 15. *Pour tout automate fini déterministe, on peut trouver un automate fini déterministe complet équivalent.*

Tout langage reconnaissable peut donc être reconnu par un automate fini déterministe complet.

Démonstration. Soit $A = (Q, q_0, F, \delta)$ un automate fini déterministe incomplet sur un langage Σ. On construit un automate fini déterministe $A' = (Q', q_0, F, \delta')$ en posant :

- $Q' = Q \cup \{q_\omega\}$;
- pour tout $(q, s) \in Q \times \Sigma$ qui n'est pas un blocage, $\delta'(q, s) = \delta(q, s)$;
- pour tout blocage $(q, s) \in Q \times \Sigma$, $\delta'(q, s) = q_\omega$;
- pour tout $s \in \Sigma$, $\delta'(q_\omega, s) = q_\omega$.

On ajoute en fait un état « puits » q_ω qui récupère tous les cas de blocage.

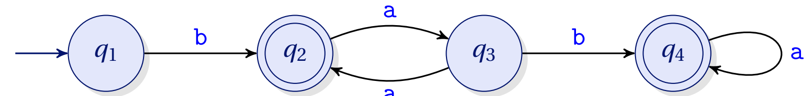
L'automate A' ainsi construit est bien complet.

Soit w un mot de Σ^* reconnu par l'automate A. Il existe un chemin acceptant étiqueté par w dans A. Ce même chemin est également un chemin acceptant dans A'. Donc le langage reconnu par A est inclus dans celui reconnu par A'.

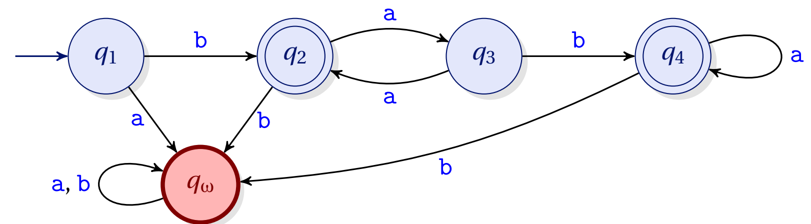
Inversement, soit w un mot de Σ^* reconnu par l'automate A'. Il existe un chemin acceptant étiqueté par w dans A'. Ce chemin ne peut pas passer par l'état q_ω, puisque tout chemin passant par q_ω se termine sur $q_\omega \notin F$. Donc le chemin est également un chemin acceptant dans A. Le langage reconnu par A' est inclus dans celui reconnu par A.

Les deux automates A et A' reconnaissent donc les mêmes langages. □

Par exemple, l'automate suivant :



reconnaît le même langage que l'automate complet ci-dessous :



Standardisation

Théorème 16. *Pour tout automate fini déterministe, on peut trouver un automate fini déterministe standard équivalent.*

Tout langage reconnaissable peut donc être reconnu par un automate fini déterministe standard.

Démonstration. Soit $A = (Q, q_0, F, \delta)$ un automate fini déterministe non-standard sur un langage Σ. On construit un automate fini déterministe $A' = (Q', q'_0, F', \delta')$ en posant :

- $Q' = Q \cup \{q_\alpha\}$;
- $F' = F \cup \{q_\alpha\}$ si $q_0 \in F$, sinon $F' = F$;
- pour tout $(q, s) \in Q \times \Sigma$ qui n'est pas un blocage, $\delta'(q, s) = \delta(q, s)$;
- pour tout $s \in \Sigma$ tel que (q_0, s) n'est pas un blocage, $\delta'(q_\alpha, s) = \delta'(q_0, s)$.
- les autres couples (q, s) sont des blocages de l'automate A'.

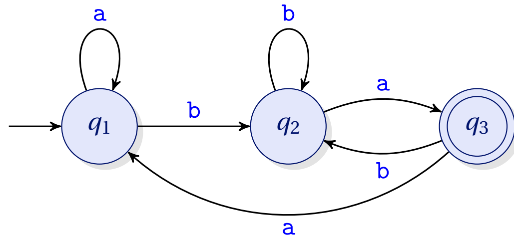
En d'autres termes, on « clone » l'état initial et les transitions qui en partent, de façon à ce qu'aucune transition ne permette de revenir au premier état de l'automate.

L'automate A' ainsi construit est bien standard.

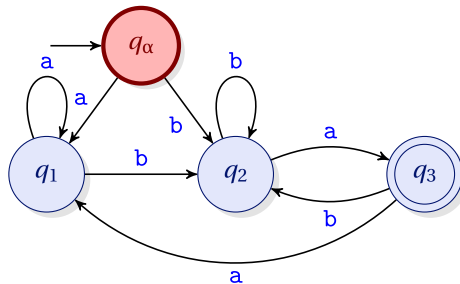
Avec un raisonnement similaire à celui utilisé pour la construction d'un automate complet, il est aisé de montrer que A et A' reconnaissent les mêmes langages (un chemin acceptant dans A devient un chemin acceptant dans A' simplement en remplaçant le premier état q_0 du chemin dans A par q_α et inversement). □

11. Attention, si l'on retourne les arcs d'un automate fini déterministe, on n'obtient pas nécessairement un automate fini déterministe ! On ne raisonne ici qu'en terme de graphes orientés.

Par exemple, l'automate suivant :



peut être transformé en un automate standard :



Bien évidemment, pour tout langage reconnaissable, il est possible de construire un automate équivalent qui soit à la fois complet *et* standard !

2.7 Implémentation Caml

Pour implémenter un automate $A = (Q, q_0, F, \delta)$ sur un alphabet Σ, on peut par exemple considérer que les éléments de Σ correspondent à des caractères [12] (type `char`) et les éléments de Q peuvent être d'un type `'a` quelconque (qui identifiera chacun des états de l'automate).

L'état initial q_0 serait alors désigné par un élément de type `'a`, et l'ensemble F est représenté par une liste de tels éléments (`'a list`).

En ce qui concerne la fonction de transition $\delta : Q \times \Sigma \mapsto Q$, son type sera donc naturellement `'a * char -> 'a`.

On définit donc un automate fini déterministe avec le type suivant :

```
# type 'a afd = { q_initial : 'a;
                  q_terminaux : 'a list;
                  delta : 'a * char -> 'a };;
```

12. Ainsi, les mots seront naturellement des chaînes de caractères.

Notons qu'il n'est pas indispensable de mémoriser les ensembles Q ou Σ (ce sont simplement des sous-ensembles des objets représentés par les types `'a` et `char`).

La fonction de transition ne sera probablement pas définie pour tous les couples (q, s), aussi est-il utile de définir une exception qui sera utilisée pour les blocages de l'automate :

```
# exception Blocage;;
```

L'exemple d'automate étudié depuis le début de ce chapitre est ainsi défini, en utilisant des entiers pour identifier les états, par :

```
# let automate_exemple = { q_initial = 1;
                           q_terminaux = [ 2 ];
                           delta = function
                                 | (1, 'a') -> 3
                                 | (1, 'b') -> 2
                                 | (2, 'a') -> 3
                                 | (2, 'b') -> 3
                                 | (3, 'a') -> 1
                                 | (3, 'b') -> 3
                                 | _ -> raise Blocage };;
```

La fonction de transition étendue aux mots s'écrit simplement récursivement :

```
# let rec deltaStar automate = function
    | (q, "") -> q
    | (q, w)  -> deltaStar automate ((automate.delta (q, w.[0])),
                    (String.sub w 1 (String.length w - 1)));;

val deltaStar : afd -> 'a * string -> 'a = <fun>
```

Il est alors aisé d'écrire une fonction prenant un automate et un mot, et retournant un booléen indiquant si le mot w est reconnu par l'automate :

```
# let reconnu automate w =
    try
      List.mem (deltaStar automate (automate.q_initial, w))
               automate.q_terminaux
    with
      Blocage -> false;;

val reconnu : afd -> string -> bool = <fun>
```

On peut alors vérifier simplement si un mot fait partie du langage décrit par l'expression

régulière `(ab*a|b(a|b)b*a)*b` (reconnu par le présent automate) :

```
# reconnu automate_exemple "ababa";;
- : bool = false

# reconnu automate_exemple "babbaabbab";;
- : bool = true
```

Si l'on écrit un programme Caml visant à *construire* des automates, il peut être utile de pouvoir définir les transitions au fur et à mesure du programme. Une solution simple consiste à utiliser une table d'association (un dictionnaire) pour ranger les associations $(q,s) \mapsto \delta(q,s)$. La fonction `delta` consiste alors simplement à effectuer une recherche dans le dictionnaire :

```
# let dict = Hashtbl.create 97;;

# let automate_exemple = { q_initial = 1;
                           q_terminaux = [ 2 ];
                           delta = Hashtbl.find dict };;
```

La définition des transitions s'effectue alors de la sorte :

```
# Hashtbl.add dict (1, 'a') 3;;
# Hashtbl.add dict (1, 'b') 2;;
# Hashtbl.add dict (2, 'a') 3;;
# Hashtbl.add dict (2, 'b') 3;;
# Hashtbl.add dict (3, 'a') 1;;
# Hashtbl.add dict (3, 'b') 3;;
```

Précisons toutefois que dans ce cas, un blocage déclenche l'exception `Not_found`, il faut donc modifier `reconnu` en conséquence.

```
# let reconnu automate w =
    try
      List.mem (deltaStar automate (automate.q_initial, w))
               automate.q_terminaux
    with
      Not_found -> false;;

val reconnu : afd -> string -> bool = <fun>
```

Ainsi, avec les outils du chapitre précédent permettant d'identifier les ensembles P, F et S associés à une expression régulière linéaire, on peut écrire une fonction prenant en argument une expression régulière **linéaire** et retournant un automate reconnaissant son interprétation. On identifie les états par des chaînes de caractères : la chaîne vide pour l'état initial, et la chaîne contenant le caractère c pour tous les arcs dans l'automate local étiquetés par ce symbole c. Cela donne par exemple :

```
# let construit r =
    let string_of_char = String.make 1 in
    let dict = Hashtbl.create 97 in
      List.iter
        (fun c -> Hashtbl.add dict ("", c) (string_of_char c))
        (prefixes r);
      List.iter
        (fun f -> Hashtbl.add dict (string_of_char f.[0], f.[1])
                                   (string_of_char f.[1]))
        (facteurs2 r);
      { q_initial = "";
        delta = Hashtbl.find dict;
        q_terminaux = let f = List.map (String.make 1) (suffixes r)
                      in if ContientMotVide r then ""::f else f };;

val construit : regexp -> string afd = <fun>
```

Cela donne bien, sur l'expression régulière linéaire `abd*|ce` du chapitre précédent, le fonctionnement attendu :

```
# let r = Choix(Concatenation(Constante "ab", Etoile(Constante "d")),
                Constante "ce")
val r : regexp =
  Choix (Concatenation (Constante "ab", Etoile (Constante "d")),
   Constante "ce")

# let automate = construit r;;
val automate : string afd =
  {q_initial = ""; q_terminaux = ["b"; "d"; "e"]; delta = <fun>}

# reconnu automate "abddd";;
- : bool = true

# List.filter (reconnu automate)
    [""; "ab"; "abc"; "abd"; "abdddd"; "ad"; "bd"; "c"; "cde"; "ce"];;
- : string list = ["ab"; "abd"; "abdddd"; "ce"]
```

Nous verrons, dans la suite, comment ces fonctions agissant sur des expressions régulières **linéaies**, peuvent être adaptées pour des expressions régulières quelconques.

3 Automates fini non-déterministes

3.1 Définition

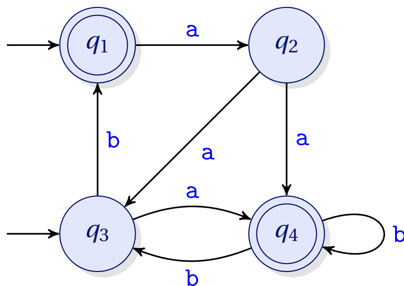
Définition. Un *automate fini non-déterministe* sur un alphabet Σ est la donnée d'un quadruplet (Q, I, F, δ) composé de
- un ensemble fini Q, correspondant aux *états* de l'automate;
- un ensemble d'états *initiaux* $I \subset Q$;
- un ensemble d'états *terminaux* (ou *finaux*) $F \subset Q$;
- une fonction de transition δ, une application d'une partie de $Q \times \Sigma$ vers l'ensemble des parties de $\mathcal{P}(Q)$.

On peut aisément prolonger la fonction de transition δ sur l'ensemble de $Q \times \Sigma$ en associant simplement l'ensemble vide $\emptyset$ pour tout couple de $Q \times \Sigma$ pour lequel δ n'était pas définie. Il n'y a donc plus vraiment de « blocage » à strictement parler de l'automate dans le cadre des automates non-déterministes.

Si l'on compare les automates non-déterministes aux automates déterministes, on peut constater plusieurs différences, qui justifient leur qualification de « non-déterministe ». Tout d'abord, il y a possiblement plusieurs états initiaux, ou aucun [13]. Cela implique que, lors de l'utilisation d'un automate non-déterministe, plusieurs états peuvent être « actifs » simultanément. Ou bien on peut se retrouver avec une situation où aucun état de l'automate n'est « actif ».

Cela se retrouve avec la fonction de transition : pour un même état $q \in Q$ et un même symbole $s \in \Sigma$, $\delta(q, s)$ peut désigner un ensemble de plusieurs états. Autrement dit, l'évolution de l'automate peut également conduire à plusieurs états actifs, ou aucun.

Les automates non-déterministes sont représentés de façon similaire aux automates déterministes, à l'aide d'un graphe. Cependant, plusieurs arcs étiquetés avec le même symbole $s \in \Sigma$ peuvent partir d'un même sommet. L'automate ci-dessous est par exemple un automate fini non-déterministe sur $\Sigma = \{\texttt{a}, \texttt{b}\}$, avec $Q = \{q_1, q_2, q_3, q_4\}$, $I = \{q_1, q_3\}$, $F = \{q_1, q_4\}$ et la fonction de transition δ définie par le tableau ci-dessous :



δ	a	b
q_1	$\{q_2\}$	$\emptyset$
q_2	$\{q_3, q_4\}$	$\emptyset$
q_3	$\{q_4\}$	$\{q_1\}$
q_4	$\emptyset$	$\{q_3, q_4\}$

13. Même si l'intérêt pratique d'un automate sans état initial est des plus limités.

3.2 Langage reconnu

Définition. Soit un automate fini non-déterministe (Q, I, F, δ) sur un alphabet Σ.

La fonction de transition étendue aux mots $\delta^* : Q \times \Sigma^* \mapsto \mathcal{P}(Q)$ est définie récursivement par :

$$\begin{cases} \text{pour tout } q \in Q,\ \delta^*(q, \varepsilon) = \{q\}; \\ \text{pour tous } q, s, w \in Q \times \Sigma \times \Sigma^*,\ \delta^*(q, s \cdot w) = \bigcup_{q' \in \delta(q,s)} \delta^*(q', w) \end{cases}$$

Un automate fini non-déterministe permet aussi de reconnaître des mots de Σ^*.

Définition. Soit un automate fini non-déterministe (Q, I, F, δ) sur un alphabet Σ.

Un mot $w \in \Sigma^*$ est *reconnu* par l'automate si et seulement si il existe $q \in I$ tel que $\delta^*(q, w) \wedge F \neq \emptyset$.

Le *langage reconnu* par l'automate est l'ensemble des mots de Σ^* reconnus par l'automate, soit le langage

$$L = \left\{ w \in \Sigma^* \;\middle|\; \bigcup_{q \in I} \delta^*(q, w) \wedge F \neq \emptyset \right\}$$

Autrement dit, $w \in \Sigma^*$ est reconnu par l'automate s'il est l'étiquette d'un chemin menant d'un état initial à un état acceptant.

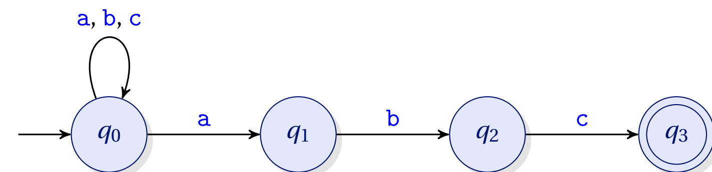
Généralement, on ne considère pas les états de I un par un, mais on travaille directement sur les parties de Q. Le mot `aabab` est reconnu par l'automate précédent car il correspond au déroulement ci-dessous :

$$I = \{q_1, q_3\} \xrightarrow{\texttt{a}} \{q_2, q_4\} \xrightarrow{\texttt{a}} \{q_3, q_4\} \xrightarrow{\texttt{b}} \{q_1, q_3, q_4\} \xrightarrow{\texttt{a}} \{q_2, q_4\} \xrightarrow{\texttt{b}} \{q_3, q_4\}$$

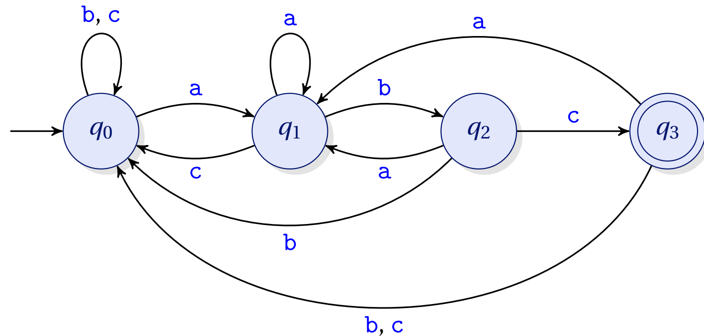
avec $\{q_3, q_4\} \wedge F \neq \emptyset$.

3.3 Quelques exemples

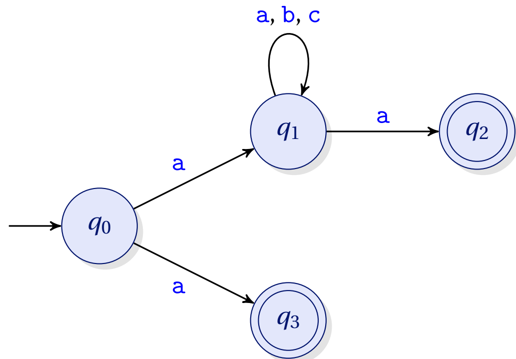
Il est souvent plus facile de créer un automate non-déterministe reconnaissant un langage qu'un automate déterministe. Par exemple, sur l'alphabet $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}\}$, l'automate suivant reconnaît le langage des mots se terminant par `abc` :



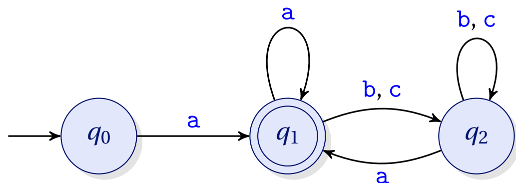
Son équivalent déterministe est sensiblement plus complexe :



De même, on peut simplement créer [14] un automate non-déterministe reconnaissant le langage des mots de Σ^* commençant et se terminant par un a :



Si la conception d'un automate non-déterministes est souvent plus facile, cela ne signifie pas que les automates ainsi obtenus soient toujours les plus simples. Par exemple, ce même langage est reconnu par l'automate déterministe plus simple suivant :



La conception de ce second automate à partir du langage à reconnaître peut en revanche sembler, à première vue tout du moins [15], un peu moins évidente.

14. On notera la facilité avec laquelle on crée un automate non-déterministe reconnaissant l'union de langages.

15. En fait, si l'on s'aperçoit que le langage est local (avec $P = F = \{a\}$ et $N = \emptyset$, on peut simplement utiliser la méthode de construction d'un automate déterministe locale évoquée tantôt pour obtenir ce résultat.

3.4 Déterminisation d'un automate

Si la création d'un automate reconnaissant un langage donné est souvent rendue plus aisée par l'emploi d'automates non-déterministes, l'utilisation en revanche d'un automate non-déterministe est sensiblement plus coûteuse en calculs, en raison de la possibilité que plusieurs états soient « actifs » en même temps. On peut donc leur préférer des versions déterminisites, ce qui est fort heureusement toujours possible.

Théorème 17. *Pour tout automate fini non-déterministe, il existe un automate fini déterministe reconnaissant le même langage.*

Démonstration. Soit un automate fini non-déterministe $A = (Q, I, F, \delta)$ sur un langage Σ^*. Considérons l'automate fini déterministe sur Σ^* défini par $A' = \left(\mathcal{P}(Q), \{I\}, F', \delta'\right)$ dont les états sont les parties de Q et tel que

- $F' = \left\{P \in \mathcal{P}(Q) \mid P \wedge F \neq \emptyset\right\} \in \mathcal{P}\left(\mathcal{P}(Q)\right)$;
- $\delta' : \begin{cases} \mathcal{P}(Q) \times \Sigma \longmapsto \mathcal{P}(Q) \\ P, s \longmapsto \bigcup\limits_{q \in P} \delta(q, s) \end{cases}$.

Pour montrer que A' est équivalent à A, il suffit de montrer que tout mot $w \in \Sigma^*$ reconnu par A l'est aussi par A' et inversement.

Pour ce faire, nous allons travailler par récurrence sur la taille de w, et montrer qu'il existe dans A un chemin étiqueté par w menant à un état q si et seulement si il existe dans A' un chemin étiqueté par w menant à un état q' « contenant » q.

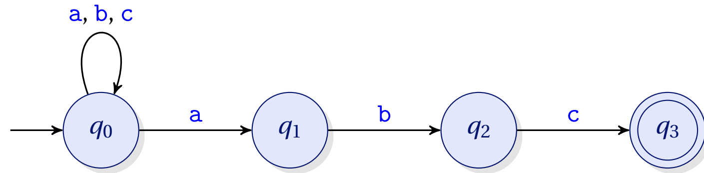
- Si $|w| = 0$, alors $w = \varepsilon$. Les chemins dans A étiquetés par ε mènent à un élément de I. Le chemin (unique) dans A' étiqueté par ε mène à I.
- Si $|w| = n > 0$, on note $w = s_1 s_2 \dots s_n$. Supposons le résultat acquis pour tout mot de longueur strictement inférieure à n.
  - S'il existe, dans A, un chemin $q_{i_0} \xrightarrow{s_1} q_{i_1} \xrightarrow{s_2} q_{i_2} \xrightarrow{s_3} \dots \xrightarrow{s_{n-1}} q_{i_{n-1}} \xrightarrow{s_n} q_{i_n}$, d'après la récurrence, il existe dans A' un chemin $I \xrightarrow{s_1} q'_{i_1} \xrightarrow{s_2} q'_{i_2} \xrightarrow{s_3} \dots \xrightarrow{s_{n-1}} q'_{i_{n-1}}$ avec $q_{i_{n-1}} \in q'_{i_{n-1}}$.

    Or $q_{i_n} \in \delta\left(q_{i_{n-1}}, s_n\right)$, donc $q_{i_n} \in \delta'\left(q'_{i_{n-1}}, s_n\right)$. En posant $q'_{i_n} = \delta'\left(q'_{i_{n-1}}, s_n\right)$, on a donc dans A' un chemin $I \xrightarrow{s_1} q'_{i_1} \xrightarrow{s_2} q'_{i_2} \xrightarrow{s_3} \dots \xrightarrow{s_{n-1}} q'_{i_{n-1}} \xrightarrow{s_n} q'_{i_n}$ avec $q_{i_n} \in q'_{i_n}$.
  - S'il existe, dans A', un chemin $I \xrightarrow{s_1} q'_{i_1} \xrightarrow{s_2} q'_{i_2} \xrightarrow{s_3} \dots \xrightarrow{s_{n-1}} q'_{i_{n-1}} \xrightarrow{s_n} q'_{i_n}$ où q'_{i_n} contient un état q_{i_n}, alors il existe $q_{i_{n-1}} \in q'_{i_{n-1}}$ tel que $q_{i_n} \in \delta\left(q_{i_{n-1}}, s_n\right)$.

    Par récurrence, il existe un chemin $q_{i_0} \xrightarrow{s_1} q_{i_1} \xrightarrow{s_2} q_{i_2} \xrightarrow{s_3} \dots \xrightarrow{s_{n-1}} q_{i_{n-1}}$ dans A. Chemin que l'on peut compléter avec q_{i_n}.

Puisque $F' = \left\{P \in \mathcal{P}(Q) \mid P \wedge F \neq \emptyset\right\}$, un mot est donc reconnu par A si et seulement si il est reconnu par A'. □

Illustrons ce processus de déterminisation sur l'automate non-déterministe suivant :



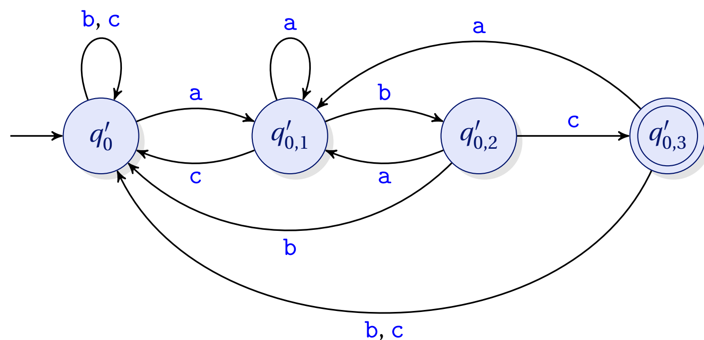
On détermine les transitions entre les parties de $\{q_0, q_1, q_2, q_3\}$:

δ	a	b	c
∅	∅	∅	∅
$\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$	$\{q_0\}$
$\{q_1\}$	∅	$\{q_2\}$	∅
$\{q_2\}$	∅	∅	$\{q_3\}$
$\{q_3\}$	∅	∅	∅
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$	$\{q_0\}$
$\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$	$\{q_0, q_3\}$
$\{q_0, q_3\}$	$\{q_0, q_1\}$	$\{q_0\}$	$\{q_0\}$
$\{q_1, q_2\}$	∅	$\{q_2\}$	$\{q_3\}$
$\{q_1, q_3\}$	∅	$\{q_2\}$	∅
$\{q_2, q_3\}$	∅	∅	$\{q_3\}$
$\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$	$\{q_0, q_3\}$
$\{q_0, q_1, q_3\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$	$\{q_0\}$
$\{q_0, q_2, q_3\}$	$\{q_0, q_1\}$	$\{q_0\}$	$\{q_0, q_3\}$
$\{q_1, q_2, q_3\}$	∅	$\{q_2\}$	$\{q_3\}$
$\{q_0, q_1, q_2, q_3\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$	$\{q_0, q_3\}$

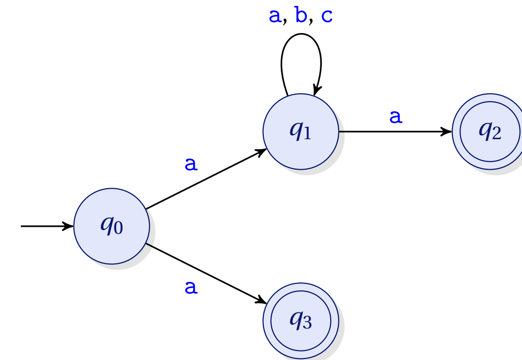
Les seuls états accessibles (et également co-accessibles, donc utiles) sont $\{q_0\}$, $\{q_0, q_1\}$, $\{q_0, q_2\}$ et $\{q_0, q_3\}$. En notant ces états respectivement q'_0, $q'_{0,1}$, $q'_{0,2}$ et $q'_{0,3}$, q'_0 est le nouvel état initial, $q'_{0,3}$ l'unique état terminal, car la seule partie de $\{q_0, q_1, q_2, q_3\}$ conservée qui contienne l'état terminal q_3 de l'automate non-déterministe. Quant à la fonction de transition δ', elle correspond à la table suivante :

δ	a	b	c
q'_0	$q'_{0,1}$	q'_0	q'_0
$q'_{0,1}$	$q'_{0,1}$	$q'_{0,2}$	q'_0
$q'_{0,2}$	$q'_{0,1}$	q'_0	$q'_{0,3}$
$q'_{0,3}$	$q'_{0,1}$	q'_0	q'_0

L'automate déterministe ainsi obtenu est celui que l'on a déjà présenté :



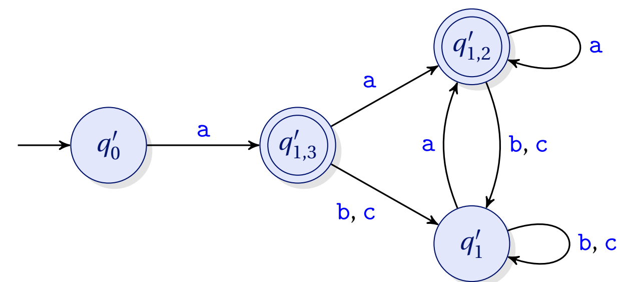
Notons que le processus de déterminisation ne conduit pas nécessairement à l'automate le plus simple. Dans le second exemple, on avait



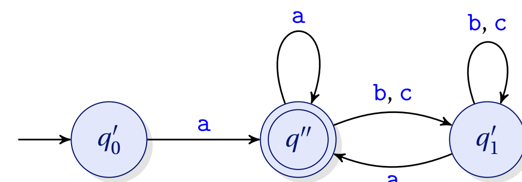
En éliminant directement les parties inutiles, on construit la table associée à δ' :

δ	a	b	c
$\{q_0\}$	$\{q_1, q_3\}$	∅	∅
$\{q_1, q_3\}$	$\{q_1, q_2\}$	$\{q_1\}$	$\{q_1\}$
$\{q_1, q_2\}$	$\{q_1, q_2\}$	$\{q_1\}$	$\{q_1\}$
$\{q_1\}$	$\{q_1, q_2\}$	$\{q_1\}$	$\{q_1\}$

On obtient donc l'automate fini déterministe suivant, qui compte un état de plus que celui présenté tantôt :



En fait, un examen plus approfondi permet de voir que les états $q'_{1,2}$ et $q'_{1,3}$ peuvent être rassemblés en un unique état q'', ce qui redonne l'automate à trois états :



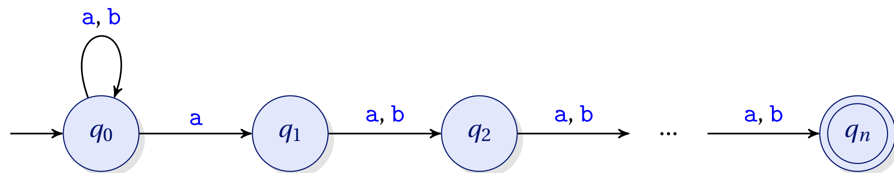
Si l'on considère la table associée à la fonction de transition δ', cela peut se comprendre : ces deux états évoluent, dans les mêmes conditions, vers les mêmes états.

C'est aussi le cas de q'_1, mais à la différence des deux autres, celui-ci n'est pas un état final. Comme il n'a pas les mêmes attributs que les deux autres, on est contraint de les garder distincts. C'est la même raison qui empêche de fusionner q'_0 et $q'_{0,3}$ sur le premier exemple.

Les automates finis non-déterministes ne sont donc pas plus « riches » que les automates finis déterministes. Mais comme ils est souvent plus facile d'obtenir un automate fini non-déterministe reconnaissant un langage donné, il est fréquent que l'on crée dans un premier temps un automate fini non-déterministe, puis qu'on le déterminise et qu'on l'émonde pour limiter son nombre d'états.

Ce nombre d'états peut cependant être particulièrement grand (au point que l'automate déterministe devient délicat à manipuler par un ordinateur, pour des raisons de mémoire). En effet, la déterminisation telle que présentée ci-dessous fait appel aux parties de Q, dont le cardinal vérifie $|\mathcal{P}(Q)| = 2^{|Q|}$. Et il n'est pas toujours possible de supprimer une partie substantielle de ces états lors de l'émondage.

Par exemple, si l'on souhaite reconnaître le langage des mots de $\Sigma^* = \{a, b\}^*$ ayant un a à la n^e position en partant de la fin, il est aisé de concevoir un automate fini non-déterministe à $n+1$ états dans ce but :



Un automate déterministe reconnaissant le même langage aura nécessairement au moins 2^n états.

Démonstration. Soit $A = (Q, q_0, F, \delta)$ un automate fini déterministe sur $\Sigma = \{a, b\}$ reconnaissant le langage des mots de Σ^* ayant un a à la n^e position en partant de la fin.

Raisonnons par l'absurde, et faisons l'hypothèse qu'il existe deux mots distincts de Σ^n, $w = s_1 s_2 \dots s_n$ et $w' = s'_1 s'_2 \dots s'_n$, tels que $\delta^*(q_0, w) = \delta^*(q_0, w')$.

w et w' étant distincts, il existe $i \in [\![1 \,..\, n]\!]$ tel que $s_i \neq s'_i$. Supposons sans perte de généralité que $s_i = a$ et $s'_i = b$.

Soit v un mot quelconque de Σ^{i-1}. Le n^e symbole de $w \cdot v$ en partant de la fin est a, le n^e symbole en partant de la fin de $w' \cdot v$ est b. Par conséquent, l'automate reconnaît $w \cdot v$ mais ne reconnaît pas $w' \cdot v$.

Seulement, on a $\delta^*(q_0, w \cdot v) = \delta^*(\delta^*(q_0, w), v) = \delta^*(\delta^*(q_0, w'), v) = \delta^*(q_0, w' \cdot v)$.

En d'autres termes, les chemins dans l'automate A partant de q_0 et étiquetés par $w \cdot v$ et $w' \cdot v$ conduisent au même état. Cet état ne peut être à la fois acceptant et non acceptant.

Par conséquent, il n'existe pas deux mots w et w' de Σ^n tels que $\delta^*(q_0, w) = \delta^*(q_0, w')$.

Les chemins partant de q_0 et étiquetés par les 2^n mots de Σ^n conduisent donc à des états deux à deux distincts. L'automate a donc nécessairement au moins 2^n états. □

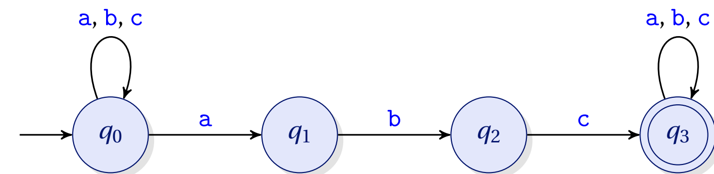
3.5 Recherche de facteur

Une tâche fréquente consiste à vérifier la présence [16] un facteur u dans un mot de Σ^* (ce qui, en langage courant, correspond par exemple à la recherche d'un mot dans un texte). C'est une tâche suffisamment courante pour qu'elle justifie à elle seule l'étude des automates. Il s'agit simplement de concevoir un automate reconnaissant le langage $\Sigma^* u \Sigma^*$.

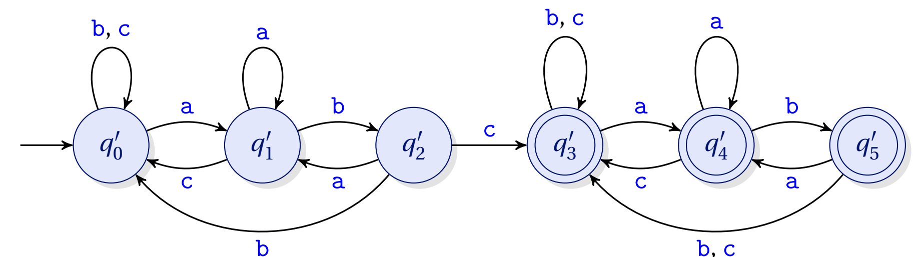
Par déterminisation d'un automate

Il est aisé de construire un automate non-déterministe dans ce but, que l'on peut déterminiser et émonder ensuite afin d'obtenir un automate déterministe.

Par exemple, rechercher la présence d'un facteur abc dans un mot de Σ^* sur l'alphabet $\Sigma = \{a, b, c\}$ revient à reconnaître le langage $(a+b+c)^* abc (a+b+c)^*$. Ce qui peut être fait très simplement par un automate non-déterministe :



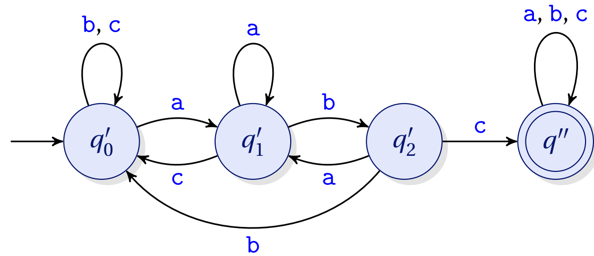
La déterminisation de cet automate (dont on laisse les détails aux soins du lecteur) conduit, après émondage, à l'automate déterministe ci-dessous :



On peut simplifier davantage l'automate obtenu en remarquant que l'on peut fusionner les états q'_3 et q'_4, puis l'état obtenu avec q'_5, avec un raisonnement sur δ' similaire à celui présenté tantôt, mais sur un automate comme celui du dessus, il est de fait assez évident que les trois états peuvent être rassemblés en un seul état q''.

16. Et, à terme, localiser, même si ce n'est pas notre propos ici.

Cela donne donc, en définitive, l'automate suivant :



Algorithme KMP

L'algorithme KMP, nommé d'après leurs inventeurs, D. Knuth, J. Morris et V. Pratt, vise à gagner du temps et construire directement un automate déterministe simplifié reconnaissant le langage $\Sigma^* u \Sigma^*$ des mots contenant un facteur u.

Dans un premier temps, on construit un automate déterministe reconnaissant les mots ayant pour suffixe le facteur recherché, $u = \texttt{abc}$ dans notre cas.

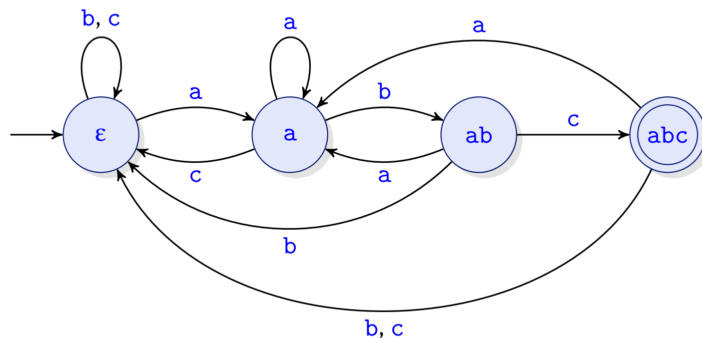
Pour ce faire, on construit un automate déterministe $A = (P(u), \varepsilon, u, \delta)$ où $P(u)$ désigne l'ensemble des préfixes du facteur u recherché : ε, a, ab et abc.

La fonction de transition δ est définie par

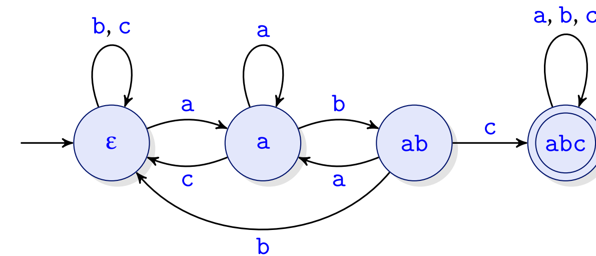
$$\delta : \begin{cases} P(u) \times \Sigma \longmapsto P(u) \\ p, x \longmapsto s(p \cdot x) \end{cases}$$

où $s(v)$ désigne le plus grand suffixe de v qui soit dans $P(u)$.

On obtient alors l'automate fini déterministe représenté ci-dessous, qui reconnait le langage $(\texttt{a}+\texttt{b}+\texttt{c})^*\texttt{abc}$ des mots se terminant par abc.



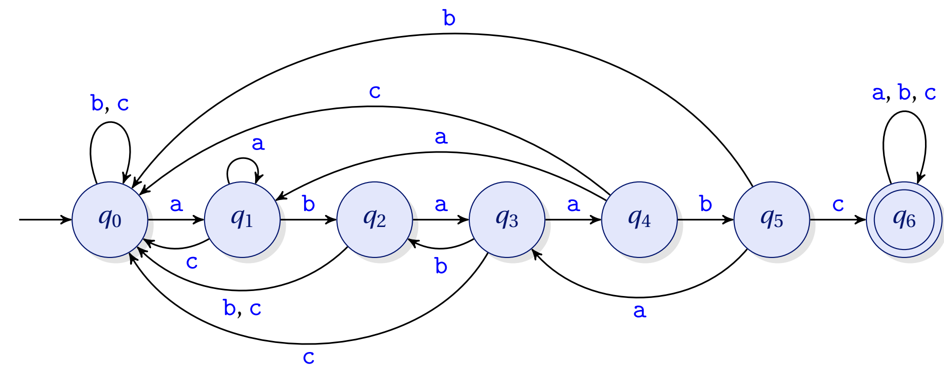
Puisque, lors de l'utilisation d'un automate avec un mot w on visite tous les états atteints pour un préfixe de w, le mot w contient le facteur u si et seulement si le dernier état est visité. Pour transformer l'automate précédent en un automate fini déterministe reconnaissant le langage $(\texttt{a}+\texttt{b}+\texttt{c})^*\texttt{abc}(\texttt{a}+\texttt{b}+\texttt{c})^*$ des mots contenant le facteur abc, il ne reste donc qu'à transformer le dernier état en un puits :



L'algorithme KMP a donc permis d'obtenir directement un automate fini déterministe simple identifiant les mots contenant le facteur u, sans avoir à construire, déterminiser et émonder un automate fini non-déterministe, ce qui permet, lorsque l'on effectue cette tâche procéduralement avec un ordinateur, de gagner du temps (et de la mémoire).

Hors construction de l'automate, et avec une implémentation similaire à celle présentée tantôt, un tel automate peut déterminer si un mot $u \in \Sigma^*$ est un facteur d'un mot $w \in \Sigma^*$ en un temps $O(|w|)$, indépendamment de la longueur du mot u recherché, plus efficacement que l'algorithme naïf dont la complexité est $O(|w| \times |u|)$. Cela permet donc de vérifier efficacement la présence [17] d'une séquence de symboles dans un texte suffisamment long [18]

En fait, il n'est pas indispensable de construire complètement l'automate. Pour le comprendre, supposons que l'on souhaite déterminer si le mot $u = \texttt{abaabc}$ est un facteur de w. Pour ce faire, l'algorithme KMP considère l'automate fini déterministe ci-dessous reconnaissant le langage $\Sigma^* u \Sigma^*$:

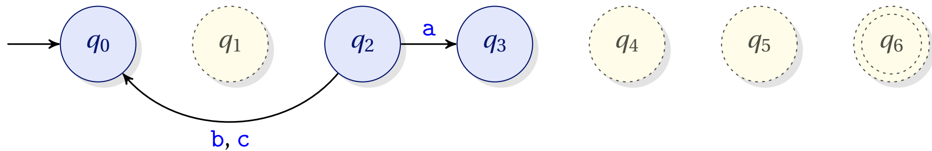


17. Certes sans en préciser la position, et l'analyse se poursuit même une fois u trouvé. Mais on peut conclure dès l'état puits atteint, et cela fournit par ailleurs la position du facteur u dans w.

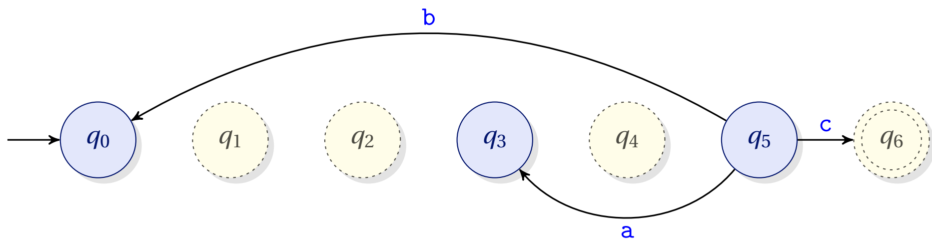
18. Le temps de construction de l'automate devenant négligeable si la taille du texte est suffisamment grande.

Toutes les transitions n'ont pas à être déterminées. Nous allons analyser plus précisément comment les choses se passent.

Supposons par exemple que, lors de l'analyse du mot $w = s_1 s_2 \dots s_n$, après i étapes, on se trouve dans l'état q_2 (le préfixe de longueur i de w se termine donc par `ab` mais pas par `abaab`) et que le symbole suivant s_{i+1} n'est pas un `a`. On revient nécessairement à l'état q_0, comme on le voit ci-dessous, où l'on n'a pas représenté toutes les transitions de l'automate :



C'est un peu plus compliqué si l'on se trouve, à un instant donné, dans l'état q_5 et que le symbole suivant n'est pas un `c` : on doit revenir à l'état q_0 si c'est un `b`, mais on ne revient qu'à l'état q_3 si c'est un `a`.



Plus l'alphabet contient de symboles, plus le nombre de transitions à considérer sera important. Il serait possible de précalculer toutes les transitions, mais ce n'est pas indispensable.

L'idée est que si l'on se trouve à l'état q_5 et que le symbole suivant n'est pas un `b`, on retournera soit à q_3, soit à q_0. Pour savoir si l'on peut revenir à q_3, comme seul un `a` peut nous y conduire par un déplacement vers la droite (depuis q_2), on peut revenir en q_2 et **fournir une nouvelle fois le symbole à traiter** : si c'est un `a`, on arrive en q_3, et sinon, on recule davantage dans l'automate.

Ainsi, après certains « échecs », on revient à un état plus à gauche de l'automate, et on essaie à nouveau le *même* caractère. Il suffit donc de précalculer les retours vers la gauche les plus favorables (s'il y en a plusieurs, et que l'on échoue au premier, on tentera naturellement le second, et ainsi de suite, jusqu'à éventuellement retourner au début).

Dans le cas de notre mot u = `abaabc`, cela se résume ainsi :

échec depuis l'état	q_0	q_1	q_2	q_3	q_4	q_5
nouvelle tentative depuis l'état		q_0		q_1	q_0	q_2

Si l'on note u_i le préfixe de longueur i de u, un échec à l'état q_i doit ramener à l'état q_j tel que u_j corresponde au plus long suffixe propre de u_i. Par exemple, lorsque l'ont arrive à l'état q_5 lors de l'analyse d'un mot, les cinq derniers symboles considérés sont ceux que l'on trouve dans u_5 = `abaab`. Seul u_2 = `ab` est un suffixe propre de u_5, donc lorsque l'on échoue le passage de q_5 à q_6, on revient à l'état q_2 (pour essayer la transition de q_2 à q_3).

Pour construire le tableau, on peut utiliser la fonction suivante :

```
# let calcKMP u =
    let t = Array.make (String.length u) (-1)
    and ecart = ref 1 in
      for i = 1 to String.length u - 1 do
        if u.[i] = u.[i - !ecart] then
          t.(i) <- t.(i - !ecart)
        else
          begin
            t.(i) <- i - !ecart;
            ecart := i - t.(i - !ecart);
            while i >= !ecart && u.[i] != u.[i - !ecart] do
              ecart := i - t.(i - !ecart)
            done
          end
      done;
      t;;

val calcKMP : string -> int array = <fun>
```

Dans cette fonction, on a utilisé le fait que si u_j [19] était le plus long suffixe propre de u_i, et que le dernier symbole de u_{i+1} est le même que celui de u_{j+1}, alors u_{j+1} est le plus long suffixe propre de u_{i+1}.

Si les derniers symboles diffèrent, on utilise le tableau pour trouver u_k plus long suffixe propre de u_j (et donc également un suffixe propre de u_i, mais second en terme de longueur), et on regarde si le dernier caractère de u_{k+1} correspond au dernier caractère de u_{i+1}. Si c'est le cas, u_{k+1} est le plus long suffixe propre de u_{i+1}, sinon on réitère la démarche (jusqu'à trouver un préfixe de u qui convienne ou épuiser tous les préfixes possibles).

Sur notre exemple, pour le facteur `abaabc`, cela donne le tableau suivant :

```
# calcKMP "abaabc";;

- : int array = [|-1; 0; -1; 1; 0; 2|]
```

19. Dans la fonction, la quantité `!ecart` correspond à $i - j$, aussi j vaut-il `i - !ecart`.

La valeur `-1` dans le tableau signifie donc simplement que l'on a épuisé les possibilités, et que l'on recommencera au niveau de l'état initial de l'automate à l'étape suivante.

La construction de ce tableau est linéaire vis-à-vis de la taille de u. En effet, à chaque itération de la boucle `for`, `i` augmente, et à chaque itération de la boucle `while`, `!ecart` augmente (en effet, `!ecart` est modifié de sorte que `i - !ecart` prenne la valeur de `t.(i - !ecart)`, or il est aisé de montrer que pour tout k, on a en permanence `t.(k)<k`). Or `i` est un entier positif majoré par $|u|$ et `!ecart` est également un entier positif majoré par $|u|$.

Il ne reste ensuite qu'à écrire l'algorithme de recherche proprement dit :

```
# let kmp u w =                   (* recherche le facteur u dans w   *)
    let t = calcKMP u
    and n = String.length w       (* n=|w|, p=|u|                    *)
    and p = String.length u in    (* i est un index dans w           *)
      let rec cherche i = function (* q_j est l'état courant dans A  *)
        | j when j=p -> i-p       (* état q_p, u est un facteur de w *)
        | j when i=n -> -1        (* fin de w, u n'est pas un facteur *)
        | j when w.[i] = u.[j]    (* transition q_j -> q_j+1 possible *)
          -> cherche (i+1) (j+1)  (* on passe à l'état suivant dans A *)
        | j when t.(j) = -1
          -> cherche (i+1) 0      (* retour à l'état q_0 dans A      *)
        | j
          -> cherche i t.(j)      (* nouvelle tentative depuis q_j   *)
      in cherche 0 0;;

val kmp : string -> string -> int = <fun>
```

Dans cet algorithme, `cherche` est une fonction prenant en argument deux entiers `i` et `j`. `j` permet d'identifier l'état courant q_j dans l'automate, tandis que i désigne le caractère s_i de w actuellement considéré. Pour le reste, il s'agit des règles précédemment décrites.

Dès que l'on atteint l'état puits q_n (le facteur u a été trouvé dans w), la fonction s'arrête et retourne la position de la première occurrence du facteur u dans w. Si la recherche échoue, la fonction précédente retourne -1.

On peut vérifier que la quantité `2i-j` est un entier, initialement nul, qui augmente strictement à chaque appel à la fonction `Cherche`. Puisque `s` reste positif et que `i` est majoré par $|w|$, la fonction Cherche est exécutée au plus $2|w|+1$ fois, ce qui implique que la recherche est en $O(|w|)$.

Finalement, en prenant en compte le précalcul, la recherche d'un facteur v dans un mot w a donc une complexité $O(|v|+|w|)$.

3.6 Implémentation en OCaml

Pour implémenter un automate fini non-déterministe en OCaml, on peut s'inspirer de ce que nous avons fait pour les automates déterministes. Toutefois, il y a quelques différences, qui notamment altèrent un peut le type utilisé pour décrire l'automate :

- pour les états initiaux, on peut simplement utiliser une liste (`'a list`) comme pour les états acceptants ;
- pour la fonction de transition δ, qui est dorénavant à valeurs dans $\mathcal{P}(Q)$ et non plus dans Q, on pourra utiliser une fonction qui retourne une liste d'états (soit un type `'a * char -> 'a list`).

Cela donne donc, par exemple :

```
# type 'a afnd = { q_initiaux : 'a list;
                   q_terminaux : 'a list;
                   delta : 'a * char -> 'a list };;
```

Le tout premier exemple d'automate fini non-déterministe que nous avons présenté se déclarerait donc ainsi (on remarquera que l'on n'a plus besoin de la notion de blocage) :

```
# let automate_exemple = { q_initiaux = [ 1; 3 ];
                           q_terminaux = [ 1; 4 ];
                           delta = function
                                     | (1, 'a') -> [ 2 ]
                                     | (2, 'a') -> [ 3; 4 ]
                                     | (3, 'a') -> [ 4 ]
                                     | (3, 'b') -> [ 1 ]
                                     | (4, 'b') -> [ 3; 4 ]
                                     | _ -> [] };;
```

Pour utiliser l'automate afin de tester si un mot est reconnu ou non, c'est en revanche un peu plus difficile. À tout instant, on a un *ensemble* d'états « actifs », ensemble qu'il faut mettre à jour à chaque caractère. On pourrait envisager d'utiliser une liste d'identifiants d'état pour ce faire. Toutefois, on voudra sans doute que cette liste ne contienne pas de doublon, aussi ajouter un élément à la liste aura souvent un coût important.

En fait, on dispose généralement en informatique de structures de données appelées « *sets* » qui servent à cela. Il s'agit de conteneurs qui permettent de gérer des ensembles, offrant des outils pour ajouter un élément ou tester la présence d'un élément efficacement (en temps constant $O(1)$ si possible, ou éventuellement en temps logarithmique en la taille de l'ensemble).

On pourrait pour ce faire utiliser des arbres binaires de recherche, étudiés en première année (l'ajout et le test d'appartenance sont de coût logarithmique en la taille de l'ensemble,

si les arbres sont auto-équilibrés), mais cela nécessite d'écrire pas mal de choses [20].

Mais on dispose d'une autre solution : utiliser un dictionnaire pour représenter notre liste d'états actifs. On utilisera les identifiants des états comme clés, et on ignorera les valeurs associées (on utilisera par exemple `()` comme valeurs), car on n'en a pas besoin : les dictionnaires nous fournissent des fonctions `Hashtbl.mem` pour tester efficacement la présence d'une clé, et `hashtbl.add` pour ajouter un couple clé-valeur [21].

Il nous manque toutefois un détail : on aimerait pouvoir itérer sur les clés du dictionnaire (comme « `for cle, valeur in D.items()` » en Python lorsque `D` est un dictionnaire). C'est possible avec la fonction [22] `Hashtbl.iter`, de signature `('a -> 'b -> unit) -> ('a, 'b) Hashtbl.t -> unit`, qui fonctionne comme `List.iter` et appelle la fonction (fournie en premier argument) successivement pour tous les couples clés-valeurs.

Ainsi, on peut écrire une fonction prenant un automate fini non-déterministe, un dictionnaire représentant un ensemble d'états et un caractère, et retournant un dictionnaire contenant l'ensemble des états actifs après avoir traité le caractère. Cela s'écrira par exemple :

```
# let suivants automate etats c =
    (* n_etats contiendra les états actifs à l'étape suivante *)
    let n_etats = Hashtbl.create 42 in
      (* Pour chaque état s dans le dictionnaire etats,          *)
      (* on détermine la liste des états atteints depuis l'état s   *)
      (* avec le caractère c avec un appel à automate.delta (s, c)  *)
      (* et on ajoute ces états à n_etats s'ils n'y sont pas encore *)
      Hashtbl.iter
        (fun s () -> List.iter
                       (fun d -> if not (Hashtbl.mem n_etats d)
                                 then Hashtbl.add n_etats d ())
                       (automate.delta (s, c)))
        etats;
      (* On termine en retournant le dictionnaire n_etats *)
      n_etats;;

val suivants : 'a afnd -> ('a, unit) Hashtbl.t -> char
                                  -> ('a, unit) Hashtbl.t = <fun>
```

20. En fait, il existe un module `Set` dans la bibliothèque standard OCaml qui fournit précisément cela, mais ce module n'est pas au programme, et il fait appel à la notion de *foncteur*, elle aussi hors programme, donc nous laisserons cette option de côté.

21. On n'ajoutera que des clés qui ne sont pas encore présentes, car l'implémentation de `Hashtbl.add` mémorise les associations précédentes, ce qui a un coût spatial. Ce détail de fonctionnement des dictionnaires OCaml n'est pas à connaître.

22. Qui ne figure pas au programme.

Une fois cette fonction `suivante` écrite, on peut écrire une fonction prenant un ensemble d'état et une chaîne de caractère, et va appeler la fonction précédente sur chacun des caractères de la chaîne :

```
# let rec deltaStar automate = function
    | (etats, "") -> etats
    | (etats, w)
        -> deltaStar automate (suivants automate etats w.[0],
                                String.sub w 1 (String.length w - 1));;

val deltaStar : 'a afnd -> ('a, unit) Hashtbl.t * string
                                  -> ('a, unit) Hashtbl.t = <fun>
```

Ceci fait, la fonction prenant un automate et une chaîne de caractères et retournantun booléen indiquant si cette chaîne est acceptée par l'automate n'aura qu'à initialiser un dictionnaire avec les états initiaux de l'automate, faire appel à la fonction précédente, et enfin tester si l'intersection entre les états obtenus et les états acceptants de l'automate est non vide. Ce qui s'écrit par exemple :

```
# let reconnu automate w =
    (* On remplit un dictionnaire etats_i avec les états initiaux *)
    let etats_i = Hashtbl.create 42 in
      List.iter (fun s -> Hashtbl.add etats_i s ()) automate.q_initiaux;
      (* On égrenne les caractères *)
      let etats_f = deltaStar automate (etats_i, w) in
        (* On teste si un de ces états est acceptant *)
        List.exists (fun s -> Hashtbl.mem etats_f s) automate.q_terminaux;;

val reconnu : 'a afnd -> string -> bool = <fun>
```

On peut aisément vérifier le bon fonctionnement de ces fonctions sur l'exemple pris au début de cette partie du chapitre :

```
# reconnu automate_exemple "aabab";;
- : bool = true

# reconnu automate_exemple "aababaa";;
- : bool = false
```

Si la complexité de `reconnu` dans le cas d'un automate fini déterministe était naturellement en $O(|w|)$, les choses sont un peu plus complexes ici. Toutefois, on peut établir une borne supérieure : à tout instant, on a au plus $|Q|$ états actifs, qui peuvent chacun avoir une transition vers $|Q|$ états. On a donc une complexité dans le pire des cas en $O\big(|w|\,|Q|^2\big)$ (même si, dans la pratique, on sera généralement très en-dessous de cette limite).

4 Langages reconnaissables

4.1 Propriétés de clôture

Théorème 18. *L'ensemble des langages reconnaissables est stable pour les opérations*
- *de complémentation dans Σ^*,*
- *de concaténation,*
- *d'intersection,*
- *d'union,*
- *d'étoile de Kleene.*

Pour démontrer ces propriétés, nous allons, dans la suite, voir comment construire un automate reconnaissant le langage résultat de chacune de ces opérations.

Complémentation

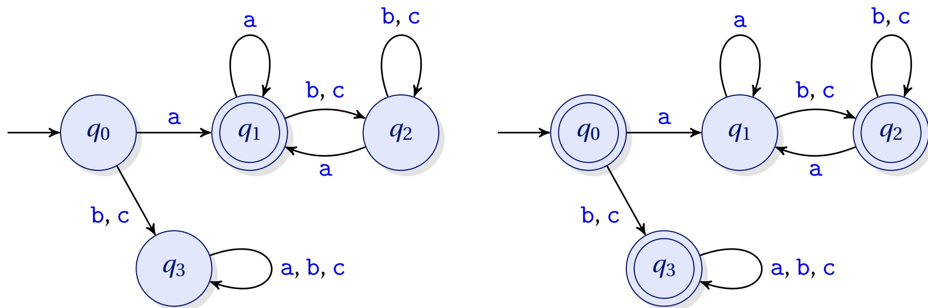
Soit un langage reconnaissable L. Considérons un automate fini déterministe complet [23] $A = (Q, q_0, F, \delta)$ reconnaissant ce langage.

L'automate fini déterministe $A_c = (Q, q_0, Q \setminus F, \delta)$ reconnaît le langage $\Sigma^* \setminus L$, donc $\Sigma^* \setminus L$ est reconnaissable.

Démonstration. Soit $w \in \Sigma^*$. Si w est reconnu par A_c, alors $\delta^*(q_0, w) \in F$, donc $\delta^*(q_0, w) \notin \Sigma^* \setminus F$, donc w n'est pas reconnu par A_c. Si w n'est pas reconnu par A_c, alors $\delta^*(q_0, w) \notin F$, donc $\delta^*(q_0, w) \in \Sigma^* \setminus F$, donc w est reconnu par A_c.

A_c reconnait donc bien $\Sigma^* \setminus L$. □

Par exemple, l'automate fini complet de gauche reconnait le langage $a + a(a+b+c)^*a$ des mots commençant et se terminant par un a [24], et celui de droite son complémentaire $\epsilon + (b+c)\Sigma^* + \Sigma^*(b+c)$.



23. Puisque L est reconnaissable, il existe un automate qui le reconnaît, et nous avons vu que l'on peut déterminiser l'automate, et trouver un automate complet équivalent.
24. Il s'agit de celui que l'on a déjà rencontré, modifié pour qu'il soit complet.

Concaténation

Soient deux langages L et L′ reconnaissables. Considérons deux automates finis déterministes $A = (Q, q_0, F, \delta)$ et $A' = (Q', q'_0, F', \delta')$ reconnaissant ces langages. On suppose par ailleurs Q et Q′ disjoints (il suffit de renommer les états communs) et A′ standard (nous avons vu qu'il était possible de trouver un automate fini déterministe standard équivalent).

L'automate fini non-déterministe $A_\bullet = (Q \cup Q' \setminus \{q'_0\}, \{q_0\}, F_\bullet, \delta_\bullet)$ avec pour ensemble des états terminaux $F_\bullet = F \cup F'$ si $q'_0 \in F'$ et $F_\bullet = F'$ sinon, et

$$\delta_\bullet(q,s) = \begin{cases} \{\delta(q,s)\} & \text{si } q \in Q \setminus F \\ \{\delta'(q,s)\} & \text{si } q \in Q' \setminus q'_0 \\ \{\delta(q,s), \delta'(q'_0,s)\} & \text{si } q \in F \end{cases}$$

reconnaît le langage L L′.

Démonstration. Pour justifier la construction de $A_\bullet$, montrons que les mots de L L′ sont reconnus par $A_\bullet$, et inversement que les mots reconnus par $A_\bullet$ sont des mots de L L′.

- Soit un mot $w \in L L'$. Par définition il existe $u, v \in L \times L'$ tels que $w = u \cdot v$.

  Puisque A reconnaît L, il existe un chemin dans A étiqueté par u menant de q_0 à un état $q \in F$. Ce chemin existe toujours dans $A_\bullet$. Deux cas se présentent ensuite :
  — Si $v = \varepsilon$, alors $q'_0 \in F'$, et alors $q \in F \subset F_\bullet$, donc $A_\bullet$ reconnaît w.
  — Sinon, v s'écrit sv' où s est le premier symbole de v. Puisque $v \in L'$, v est reconnu par A′, donc il existe un chemin étiqueté par v dans A′ menant de q'_0 à un état de F′. Notons $q' = \delta'(q'_0, s)$. Il existe un chemin étiqueté par v' de q' vers un état de F′. Comme l'automate A′ est standard, tous les états sont dans $Q' \setminus \{q_0\}$. Ce chemin existe toujours dans $A_\bullet$. Enfin, $q' \in \delta_\bullet(q,s)$ par construction de $\delta_\bullet$. On peut donc « recoller » les deux morceaux de chemin dans A′ et obtenir un chemin. $q_0 \xrightarrow{u} q \xrightarrow{s} q' \xrightarrow{v'} q'' \in F'$. Ce chemin part de l'état initial de $A_\bullet$ et termine dans F′, donc $A_\bullet$ reconnaît w.

- Inversement, considérons un mot $w \in \Sigma^*$ reconnu par $A_\bullet$. Il existe un chemin dans A_+ étiqueté par w menant de q_0 à un état $q'' \in F_+$. Par construction, les transitions dans A′ vont
  — d'un état de Q vers un état de Q ;
  — d'un état de Q′ vers un état de Q′ ;
  — d'un état de $F \subset Q$ vers un état de Q′.

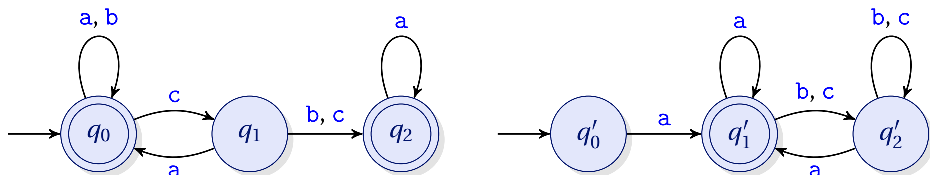
  Si $q'' \in Q$, alors $q'' \in F$, et le chemin étiqueté par w ne passe que par des états de Q. Ce chemin existe aussi dans A. Donc $w \in L$. Mais si $F_\bullet \cup F \neq \emptyset$, alors $\varepsilon \in L'$. Donc $w = w \cdot \varepsilon \in L L'$.

  Sinon, $q_0 \in Q$ et $q'' \in Q'$, donc le chemin passe par des états de Q puis des états

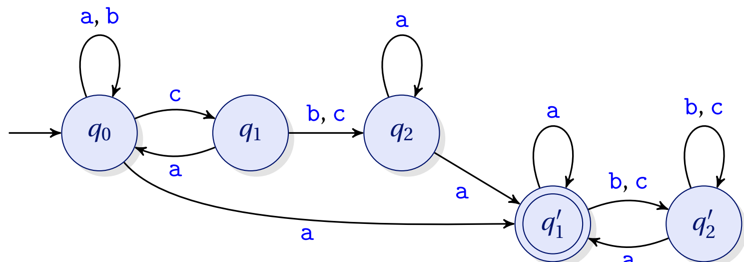
de Q'. Notons q le dernier état du chemin dans Q et q' le premier dans Q'. On a nécessairement $q \in F$, et le chemin de q_0 à q existe dans A, donc l'étiquette u de la partie de chemin de q_0 à q existe dans A. u est donc reconnu par A, soit $u \in L$.

Le chemin de q' à q'' existe dans A', et $q'' \in F'$. Notons v' l'étiquette de ce chemin, et s le symbole étiquetant la transition de q à q'. Il existe dans A' une transition de q'_0 à q' étiquetée par s. Par conséquent, il existe dans A' un chemin de q'_0 à $q'' \in F'$ étiqueté par $s \cdot v'$. Et donc $s \cdot v' \in L'$. w est donc la concaténation de $u \in L$ et $s \cdot v' \in L'$, d'où $w \in L L'$. □

Par exemple, considérons les automates ci-dessous, reconnaissant des langages L et L' :



L'automate non déterministe (mais déterminisable) suivant reconnaît le langage $L L'$:



Étoile de Kleene

Soit un langage reconnaissable L. Considérons un automate fini déterministe standard $A = (Q, q_0, F, \delta)$, reconnaissant celui-ci.

L'automate fini (non-déterministe) $A_* = (Q, \{q_0\}, F_*, \delta_*)$ où $F_* = F \cup \{q_0\}$ et

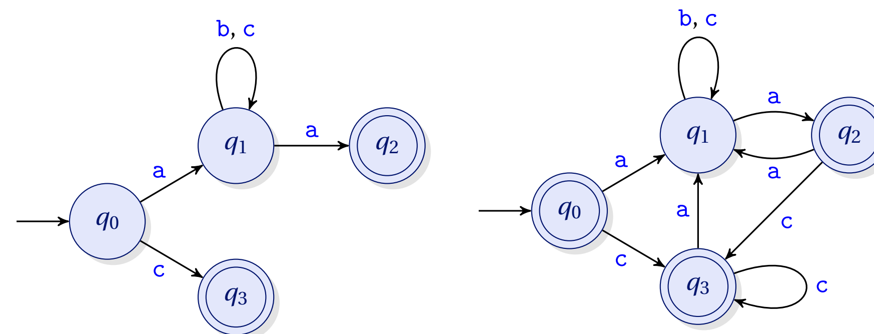
$$\delta_*(q, s) = \begin{cases} \{\delta(q, s)\} & \text{si } q \in Q \setminus F \\ \{\delta(q, s), \delta(q_0, s)\} & \text{si } q \in F \end{cases}$$

reconnaît le langage L^*.

Démonstration. L'idée est la même que pour l'automate reconnaissant $L L'$:

- si $w \in L^*$, soit $w = \varepsilon$ (et w est reconnu par A_* puisque $q_0 \in F_*$), soit on peut décomposer $w = u_1 \cdot u_2 \cdot \ldots \cdot u_n$ en un ensemble de mots u_i reconnus par A, et on peut vérifier que les chemins étiquetés par les u_i dans A permettent de reconstruire un chemin dans A_* ;
- si w est reconnu par A_*, soit $w = \varepsilon$ (et donc $w \in L^*$), soit on le décompose le chemin étiqueté par w allant de q_0 à un état de F en une suite de chemins obtenus en le découpant au niveau des états dans F, puis on montre que l'on peut retrouver dans A des chemins avec les mêmes étiquettes menant de q_0 à un état de F, et donc que w est une concaténation de mots de L, soit $w \in L^*$. □

Par exemple, l'automate ci-dessous à droite reconnaît l'étoile du langage reconnu par l'automate de gauche :



Intersection

Soient deux langages L et L' reconnaissables. Considérons deux automates $A = (Q, q_0, F, \delta)$ et $A' = (Q', q'_0, F', \delta')$ reconnaissant ceux-ci. On construit un automate « produit » $A_\cap = (Q \times Q', (q_0, q'_0), F \cap F', \delta_\cap)$ avec [25]

$$\delta_\cap : \begin{cases} (Q \times Q') \times \Sigma \longmapsto Q \times Q' \\ (q, q'), s \longmapsto (\delta(q, s), \delta'(q, s)) \end{cases}$$

Cet automate $A_\cap$ reconnait le langage $L \cap L'$.

Démonstration. On définit simplement la fonction de transition étendue aux mots $\delta^*_\cap$ pour $A_\cap$ par

$$\delta^*_\cap : \begin{cases} (Q \times Q') \times \Sigma \longmapsto Q \times Q' \\ (q, q'), w \longmapsto (\delta^*(q, w), \delta'^*(q', w)) \end{cases}$$

Si l'on rencontre un cas de blocage dans $A_\cap$, c'est qu'il y a un blocage dans A ou dans A', et inversement, en cas de blocage dans A ou A', il y a un blocage dans $A_\cap$.

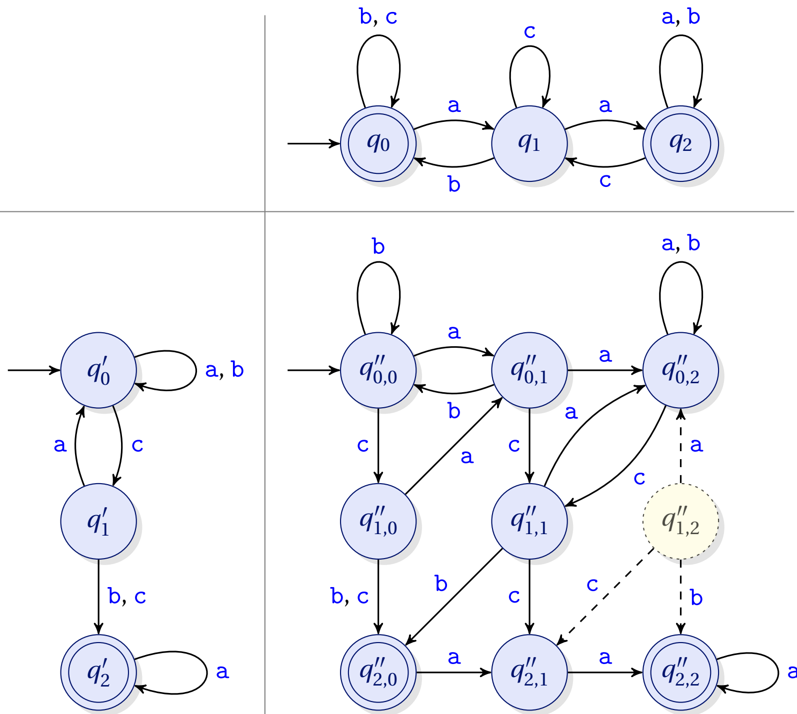
Supposons $w \in L \cap L'$. w est reconnu par A et A', donc $\delta^*(q_0, w) \in F$ et $\delta'^*(q'_0, w) \in F'$. Par conséquent, $\delta^*_\cap((q_0, q'_0), w) = (\delta^*(q_0, w), \delta'^*(q'_0, w)) \in F \cap F'$. w est donc reconnu par $A_\cap$.

25. Si (q, s) est un blocage pour l'automate A, ou (q', s) est un blocage pour l'automate A', alors $((q, q'), s)$ est un blocage pour $A_\cap$.

Inversement, si $w \in \Sigma^*$ est reconnu par $A_\cap$, alors $\delta^*_\cap((q_0, q'_0), w) = (\delta^*(q_0, w), \delta'^*(q'_0, w)) \in F \cap F'$, d'où $\delta^*(q_0, w) \in F$ et $\delta'^*(q'_0, w) \in F'$. w est donc reconnu par A et A′, donc $w \in L \cap L'$.

$A_\cap$ reconnaît donc bien l'intersection $L \cap L'$ des deux langages L et L′ reconnus par A et A′. □

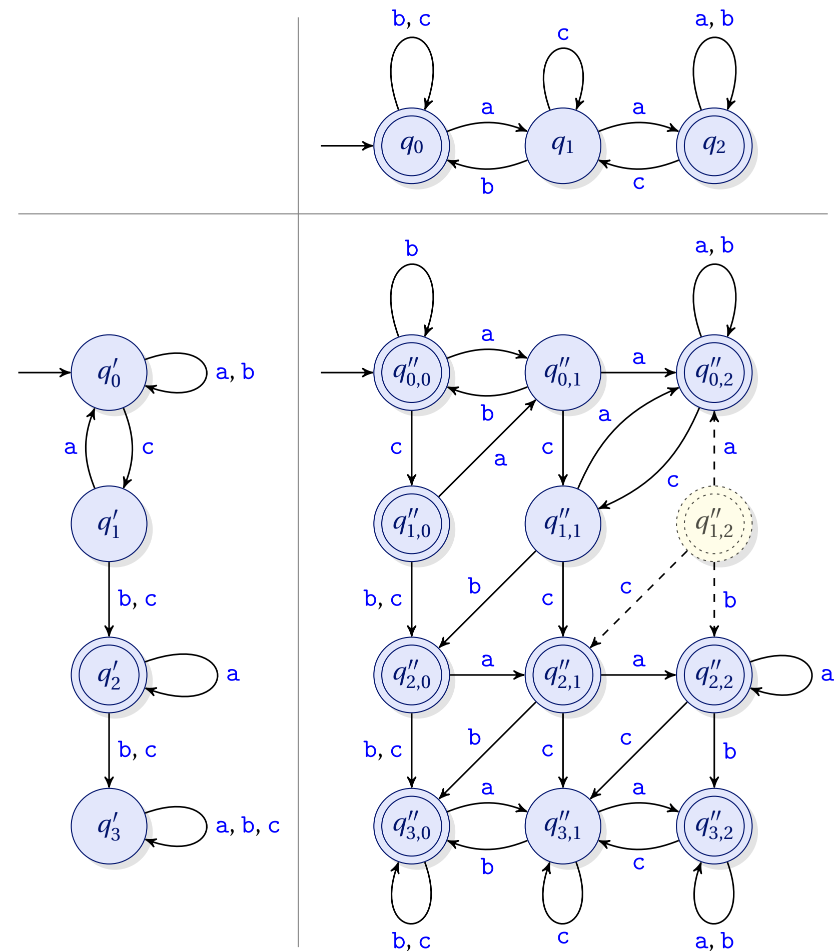
Par exemple, l'automate ci-dessous, en bas à droite, est l'automate produit tel que défini précédemment [26] de l'automate du dessus et de l'automate de gauche. Il reconnait donc l'intersection de leurs langages :



Union

Il est des plus aisés d'obtenir un automate reconnaissant l'union de deux langages reconnus par des automates $A = (Q, q_0, F, \delta)$ et $A' = (Q', q'_0, F', \delta')$ sur un même langage Σ : il suffit de rassembler ces automates, sans altération, pour obtienir un automate non-déterministe qui remplit cette fonction. Reste ensuite, éventuellement, à le déterminiser.

En fait, si A et A′ sont déterministes et complets [27], la déterminisation de ces deux automates réunis conduit naturellement à un automate A_+ de type « produit », où l'ensemble F_+ des états terminaux est $F_+ = F \cup F'$. Pour les mêmes automates que dans la section précédente (après avoir complété celui de gauche), on peut ainsi obtenir un automate A_+ déterministe reconnaissant $L + L'$:



Une façon alternative, équivalente, de voir les choses est d'écrire que, en suivant les règles de la théorie des ensembles, on a $L + L' = \Sigma^* \setminus ((\Sigma^* \setminus L) \cap (\Sigma^* \setminus L'))$, et d'utiliser les règles précédentes pour déterminer les automates reconnaissant les complémentaires et intersections des langages.

26. Et que l'on peut émonder en supprimant l'état $q''_{1,2} = (q_1, q'_2)$, qui n'est pas utile car non-accessible.

27. S'ils ne le sont pas, il suffira de les rendre complets en ajoutant un état-puits comme vu précédemment.

4.2 Théorème de Kleene

Théorème 19 (de Kleene)**.** *Un langage* L *sur un alphabet* Σ *est régulier*[a] *si et seulement si il est reconnaissable par un automate fini.*

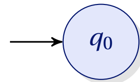
a. C'est-à-dire qu'il existe une expression régulière e telle que $\mathcal{L}(e) = \mathrm{L}$.

« Langage régulier » et « langage reconnaissable » sont donc synonymes.

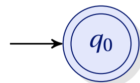
Démonstration. Soit une expression régulière e quelconque sur un alphabet Σ. Pour montrer que le langage $\mathcal{L}(e)$ est reconnaissable, nous allons montrer qu'il est possible de construire un automate reconnaissant $\mathcal{L}(e)$.

Rappelons que toute expression régulière est construite à partir de ∅, ε et des symboles $s \in \Sigma$, et en utilisant les opérateurs d'étoile, de choix, et de concaténation.

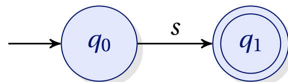
Le langage {}, interprétation de l'expression régulière Ø, est reconnu par l'automate suivant :



Le langage {ε}, interprétation de l'expression régulière ϵ, est reconnu par l'automate suivant :



Pour tout $s \in \Sigma$, le langage $\{s\}$, interprétation de l'expresion régulière s, est reconnu par l'automate suivant :



Pour tout autre expression régulière e, on construit l'automate reconnaissant le langage $\mathcal{L}(e)$ en travaillant par induction :

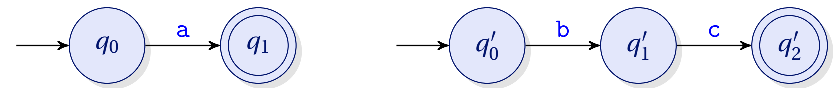
- si $e = f\star$, on construit un automate A reconnaissant le langage $\mathcal{L}(f)$, et on en déduit un automate A_* reconnaissant le langage $\mathcal{L}(f\star)$;
- si $e = fg$, on construit deux automates A et A′ reconnaissent les langages $\mathcal{L}(f)$ et $\mathcal{L}(g)$, puis on en déduit un automate $\mathrm{A}_\bullet$ reconnaissant leur concaténation, le langage $\mathcal{L}(fg) = \mathcal{L}(f)\,\mathcal{L}(g)$;
- si $e = f \mid g$, on construit deux automates A et A′ reconnaissent les langages $\mathcal{L}(f)$ et $\mathcal{L}(g)$, puis on en déduit un automate A_+ reconnaissant leur union, le langage $\mathcal{L}(f \mid g) = \mathcal{L}(f) + \mathcal{L}(g)$.

Il est donc possible de construire un automate reconnaissant une expression régulière e quelconque, et donc tout langage régulier est reconnaissable.

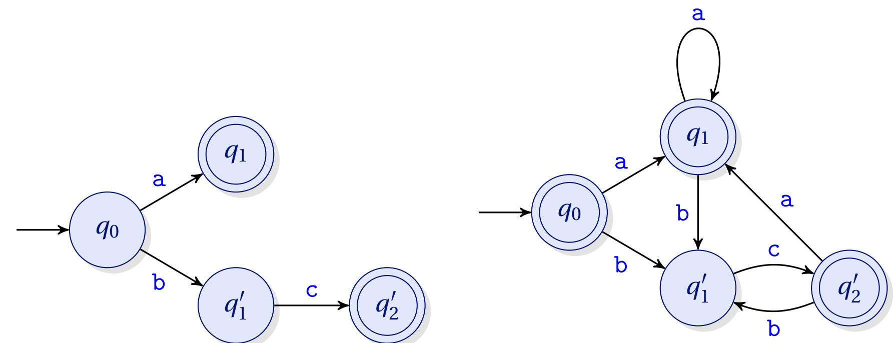
Nous aborderons la question de la réciproque ultérieurement. □

Pour illustrer la construction d'un automate par induction, considérons par exemple l'expression régulière `(a|bc)*b*a` dont l'interprétation est le langage $(\mathtt{a} + \mathtt{bc})^*\mathtt{b}^*\mathtt{a}$.

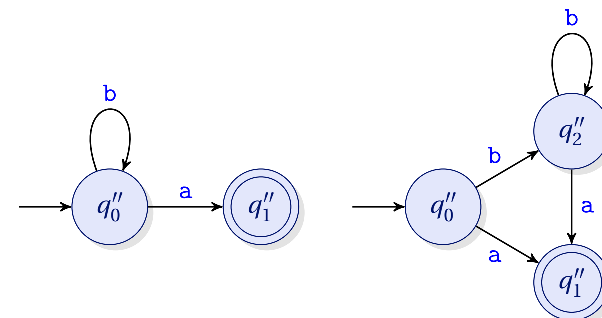
On commence par construire deux automates, représentés ci-dessous, reconnaissant respectivement les langages a et bc :



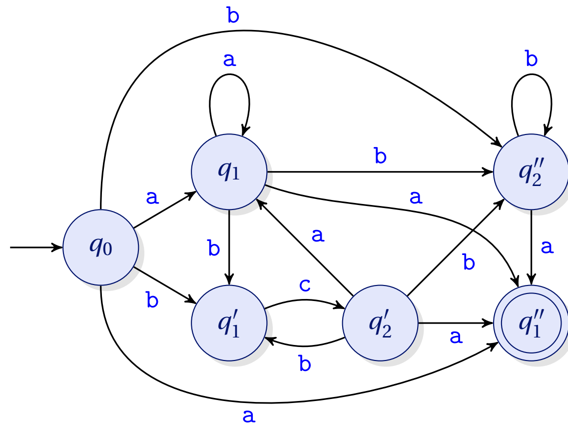
On en déduit un automate reconnaissant le langage $\mathtt{a} + \mathtt{bc}$ (ci-dessous, à gauche), puis un automate reconnaissant le langage $(\mathtt{a} + \mathtt{bc})^*$ (ci-dessous, à droite) :



Puis on crée un automate reconnaissant $\mathtt{b}^*\mathtt{a}$ (ci-dessous, à gauche), que l'on standardise (ci-dessous, à droite) :



Il ne reste plus qu'à utiliser les règles de concaténation pour obtenir un automate reconnaissant le langage $(\mathtt{a}+\mathtt{bc})^*\mathtt{b}^*\mathtt{a}$:

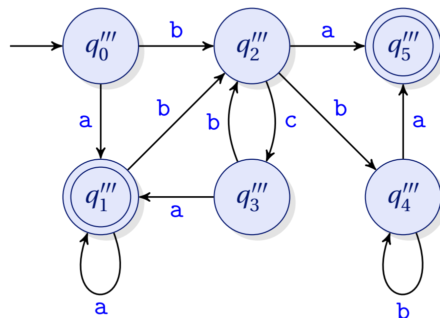


Cet automate n'est pas déterministe (il y a par exemple deux transitions étiquetés par a et deux transitions étiquetés par b partant de l'état initial q_0), mais on peut le déterminiser.

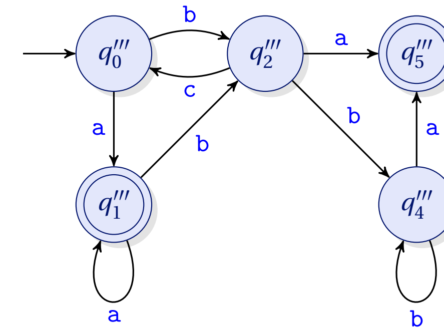
Pour ce faire, on calcule les états et transitions de l'automate déterministe équivalent :

δ	a	b	c
$q'''_0 = \{q_0\}$	$\{q_1, q''_1\}$	$\{q'_1, q''_2\}$	$\emptyset$
$q'''_1 = \{q_1, q''_1\}$	$\{q_1, q''_1\}$	$\{q'_1, q''_2\}$	$\emptyset$
$q'''_2 = \{q'_1, q''_2\}$	$\{q''_1\}$	$\{q''_2\}$	$\{q'_2\}$
$q'''_3 = \{q'_2\}$	$\{q_1, q''_1\}$	$\{q'_1, q''_2\}$	$\emptyset$
$q'''_4 = \{q''_2\}$	$\{q''_1\}$	$\{q''_2\}$	$\emptyset$
$q'''_5 = \{q''_1\}$	$\emptyset$	$\emptyset$	$\emptyset$

Et on obtient finalement un automate reconnaissant le langage correspondant à l'interprétation de l'expression régulière `(a|bc)*b*a` :



Notons que l'état q'''_3 n'est pas indispensable, et peut être fusionné avec l'état q_0, ce qui donne :



4.3 Algorithme de Berry-Sethi et automate de Glushkov

Principe

L'algorithme de Berry-Sethi est une autre méthode permettant d'obtenir un automate fini (non-déterministe) reconnaissant une expression régulière donné, qui se prête bien à l'automatisation.

Nous avons vu qu'il est aisé de construire un automate fini déterministe reconnaissant un langage local. Or, dans le chapitre précédent, nous avons montré que toute expression régulière linéaire s'interprétait en un langage local. Construire un automate associé à une expression régulière linéaire est donc simple.

Pour une expression régulière qui n'est pas linéaire mais dans laquelle n'apparaissent pas les symboles ϵ et ∅, on procède de la sorte de la sorte :

Algorithme : Algorithme de Berry-Sethi

① Linéariser l'expression régulière e en étiquetant les symboles

② Calculer les ensembles P, S et F du langage local qu'est l'interprétation de l'expression régulière linéaire

③ Construire l'automate fini déterministe associé au langage local

④ Retirer les étiquettes des transitions de l'automate précédent pour obtenir un automate non-déterministe reconnaissant $\mathcal{L}(e)$

On peut, si on le souhaite, déterminiser ensuite l'automate obtenu.

Pour les expressions régulières $e = \emptyset$ ou $e = \epsilon$, nous avons vu qu'il existait un automate reconnaissant $\mathcal{L}(e)$.

Enfin, pour une expression régulière e dans laquelles apparaissent $\emptyset$ et/ou ϵ autre que les deux cas précédents, nous avons montré dans le dernier chapitre qu'il existe une expression régulière e' ne contenant ni $\emptyset$, ni ϵ, telle que $\mathcal{L}(e) = \mathcal{L}(e')$ ou $\mathcal{L}(e) = \mathcal{L}(\epsilon \mid e')$.

On utilise donc la méthode de Berry-Sethi pour construire un automate reconnaissant $\mathcal{L}(e')$, et, si l'on se trouve dans le cas $\mathcal{L}(e) = \mathcal{L}(\epsilon \mid e')$, on ajoute l'état initial de l'automate ainsi construit à l'ensemble des états acceptants.

En effet, l'automate local obtenu dans l'algorithme de Berry-Sethi est standard (et le reste si on le déterminise), aussi ajouter l'état initial parmi les états acceptant permet de reconnaître le mot vide ε et uniquement celui-ci !

L'automate fini non-déterministe obtenu avec l'algorithme de Berry-Sethi est appelé *automate de Glushkov*.

Exemple

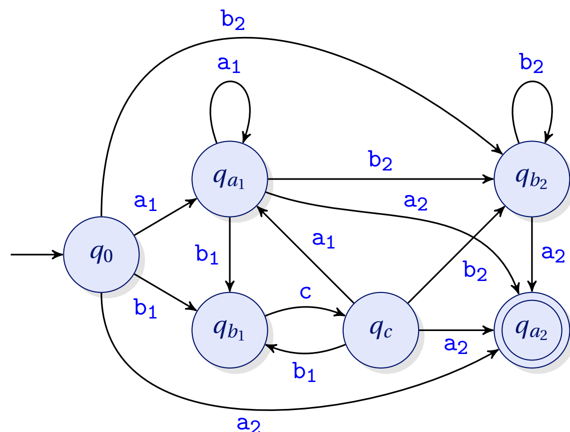
Pour illustrer l'algorithme de Berry-Sethi, reprenons l'expression régulière `(a|bc)*b*a` qui nous a précédemment servi d'exemple.

La première étape consiste à linéariser cette expression régulière, en étiquetant les symboles pour qu'ils n'apparaissent qu'une seule fois.

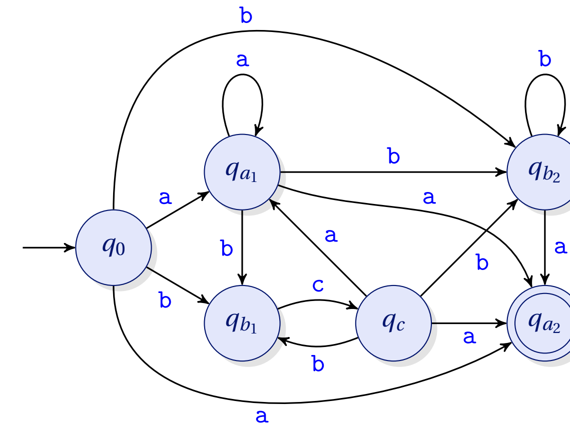
On construit ainsi une expression régulière linéaire : $(a_1 \mid b_1 c)^* \; b_2^* \; a_2$. Son interprétation est un langage local, dont on peut calculer les ensembles P, S et F :

- $P = \{a_1, b_1, a_2, b_2\}$;
- $S = \{a_2\}$;
- $F = \{a_1a_1, a_1a_2, a_1b_1, a_1b_2, b_1c, ca_1, cb_1, ca_2, cb_2, b_2a_2, b_2b_2\}$.

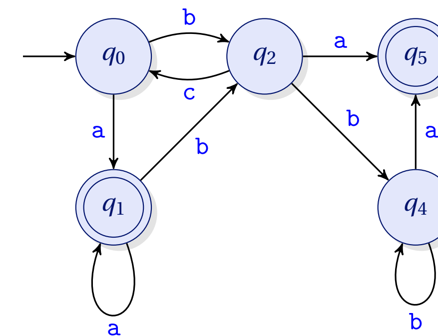
On peut alors construire un automate fini déterministe reconnaissant ce langage :



La dernière étape consiste à retirer les étiquettes sur les transitions. On retrouve alors très exactement l'automate obtenu précédemment :



On peut éventuellement déterminiser et simplifier (comme nous l'avons fait précédemment) cet automate de Glushkov pour obtenir :



Complexité de la construction

La première étape d'étiquetage des symboles est, de façon évidente, linéaire ($\Theta(n)$) en la longueur (en nombre de caractères/symboles) de l'expression régulière [28], puisqu'elle consiste en une substitution de symboles, et que l'on peut utiliser un tableau ou un dictionnaire pour savoir quels symboles ont déjà été utilisés [29].

28. Et, *généralement*, en sa taille, définie rappelons-le comme la taille de l'arbre qui la représente, même si, en théorie, on peut rendre la longueur de l'expression régulière arbitrairement grande par rapport à sa taille, en ajoutant des parenthèses inutiles.
29. Si n est grand, on peut s'arranger pour que leur initialisation ne soit pas le coût dominant

La détermination des paramètres du langage local et la construction de l'automate fini non-déterministe correspondant (qui a moins de $n+2$ états) ont des complexités quadratiques en le nombre de symboles utilisés, donc en $O(n^2)$. Retirer les étiquettes, sur $O(n^2)$ transitions, a également une complexité en $O(n^2)$. On peut donc construire l'automate fini non-déterministe à partir d'une expression régulière en temps quadratique en sa longueur.

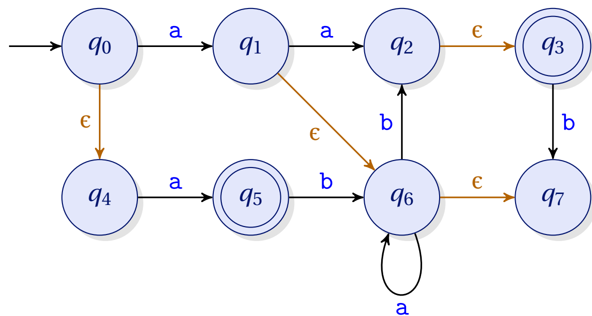
La détermination (et l'éventuel émondage) peuvent en revanche être plus coûteux, comme nous l'avons vu précédemment.

5 Généralisation des automates

5.1 Automates asynchrones

Principe

Pour simplifier certaines preuves ou certains raisonnements, il peut être utile de généraliser encore la notion d'automate fini. Un *automate fini asynchrone*[30] est une généralisation des automates finis non-déterministe dans laquelle les transitions sont étiquetées avec des éléments de $\Sigma \cup \{\epsilon\}$. En d'autres termes, on ajoute la possibilité de créer des transitions entre deux états de l'automate étiquetés par ϵ. Ces transitions particulières sont appelées ϵ-transitions, ou encore *transitions spontanées*. L'automate ci-dessous, contenant quatre transitions spontanées, est un exemple d'un tel automate :



Langages reconnus

Dans un tel automate, on dit qu'il existe un *ϵ-chemin* entre deux états q et q' s'il existe une séquence d'états $q_0, q_1, \ldots, q_n$ avec $q_0 = q$ et $q_n = q'$ telle que, pour tout $i \in [\![0\,..\,n-1]\!]$, il existe une transition spontanée entre q_i et q_{i+1}.

L'utilisation d'un tel automate, dans le cadre de la reconnaissance d'un mot w, est similaire à un automate fini non déterministe, avec une subtilité : si, à un instant donné, un état q est actif, alors tout état q' tel qu'il existe un ϵ-chemin entre q et q', alors q' est « actif » également. Entre deux évolutions dûes à la lecture de symboles de w, on va donc glisser une propagation le long des transitions spontanées[31]. On le fera également avant le premier symbole de w, et après le dernier symbole de W.

Par exemple, le mot `aba` est reconnu par l'automate asynchrone précédent, car :

$$\mathrm{I} = \{q_0\} \xrightarrow{\epsilon} \{q_0, q_4\} \xrightarrow{\mathtt{a}} \{q_1, q_5\} \xrightarrow{\epsilon} \{q_1, q_5, q_6, q_7\} \xrightarrow{\mathtt{b}} \{q_2, q_6\} \xrightarrow{\epsilon} \{q_2, q_3, q_6, q_7\}$$

$$\xrightarrow{\mathtt{a}} \{q_2\} \xrightarrow{\epsilon} \{q_2, q_3\}$$

avec $\{q_2, q_3\} \wedge \mathrm{F} \neq \emptyset$.

On remarquera en particulier qu'après la lecture du premier `a`, les transtions spontanées activent non seulement l'état q_6 mais également l'état q_7, puisqu'il existe un ϵ-chemin menant de q_1 à q_7. On peut donc suivre autant de transitions spontanées que nécessaire entre deux lectures de symboles.

Élimination des transitions spontanées

Si, nous le verrons, les automates asynchrones peuvent présenter des avantages en terme de construction ou de raisonnement, ils ne sont pas plus expressifs que les automates finis (déterministes ou non) que nous avons décrits précédemment. En effet, on peut aisément construire un automate fini déterministe équivalent à un automate asynchrone, reconnaissant le même langage, en éliminant chacune des transitions spontanées.

Pour y parvenir, on peut, par exemple, appliquer une méthode que l'on qualifie généralement de *fermeture avant* de l'automate asynchrone :

- pour chaque transition d'un état q à un état q' étiquetée par un symbole s et chaque ϵ-chemin de q' à un état q'', on ajoute une transition de q à q'' étiquetée par s;
- pour chaque ϵ-chemin menant d'un état initial q à un état q', on ajoute q' dans l'ensemble des états initiaux;
- on supprime toutes les transitions spontanées.

Comme le nom le laisse penser, il existe une seconde méthode pour parvenir au même résultat, très similaire, dite de *fermeture arrière* :

- pour chaque ϵ-chemin d'un état q à un état q' et chaque transition de q' à un état q'' étiquetée par un symbole s, on ajoute une transition de q à q'' étiquetée par s;
- pour chaque ϵ-chemin menant d'un état q à un terminal q', on ajoute q dans l'ensemble des états terminaux;
- on supprime toutes les transitions spontanées.

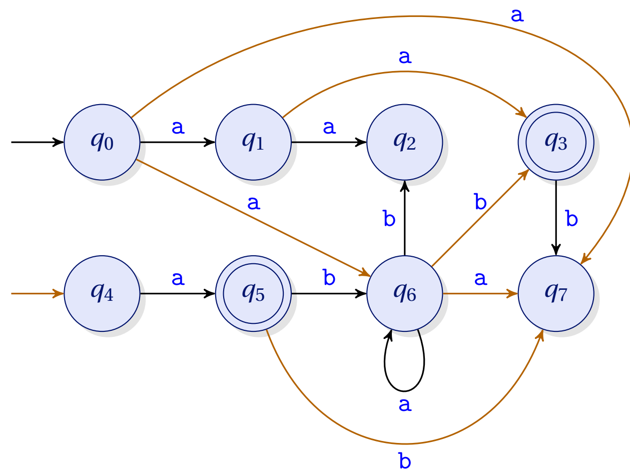
30. On parle également d'automate à ϵ-transitions, ou simplement d'automate fini non-déterministe avec ϵ-transitions.

31. D'où la qualification d'« asynchrone » pour l'automate, puisque l'on ne passe pas d'un état à un autre uniquement lors de la lecture d'un symbole.

Considérons par exemple la première de ces deux méthodes, et appliquons-la à l'automate asynchrone qui nous a servi d'exemple. Avant de pouvoir supprimer les transitions spontanées, il nous faut effectuer les transformations suivantes :

- la transition de q_0 à q_1 étiquetée par a conduit à la création de transitions de q_0 à q_6 et de q_0 à q_7 étiquetées par a ;
- la transition de q_1 à q_2 étiquetée par a conduit à la création d'une transition de q_1 à q_3 étiquetée par a ;
- la transition de q_6 à q_2 étiquetée par b conduit à la création d'une transition de q_6 à q_3 étiquetée par b ;
- la transition de q_5 à q_6 étiquetée par b conduit à la création d'une transition de q_5 à q_7 étiquetée par b ;
- la boucle sur q_6 étiquetée par a conduit à la création d'une transition de q_6 à q_7 étiquetée par a ;
- l'état q_0 étant un état initial, l'état q_4 le devient également.

Cela conduit donc à l'automate fini (non-déterministe) ci-dessous :



5.2 Retour sur la construction des automates

Les transitions spontanées permettent de simplifier certaines opérations sur les automates. En particulier, elles nous fournissent un moyen très simple, à partir d'automates asynchrones A et A′ reconnaissant deux langages L et L′, de construire un automate asynchrone reconnaissant le langage LL′ : il suffit en effet d'ajouter une transition spontanée menant de chaque état terminal de A à chaque état initial de A′.

Il est à peine plus compliqué, à partir d'un automate A reconnaissant un langage L, de construire un automate asynchrone reconnaissant le langage L* : il suffit d'ajouter des transitions spontanées menant de chaque état terminal de A vers chaque état initial de A, et de rendre terminaux tous les états initiaux de A.

Puisque, pour reconnaitre l'union de deux langages L et L′ il suffit d'unir les automates capable de les reconnaître, et qu'il est immédiat de construire un automate reconnaissant le langage vide, le langage réduit au mot vide, et les langages réduits à un unique mot s pour tout symbole $s \in \Sigma$ (comme nous l'avons fait précédemment), on dispose donc d'une autre méthode pour transformer une expression régulière e en un automate asynchrone reconnaissant le langage $\mathscr{L} e$. On parle de *méthode de Thomson*. Une fois l'automate asynchrone ainsi construit, on pourra le transformer en automate fini non-déterministe par élimination des transitions spontanées, puis éventuellement en automate fini déterministe par déterminisation.

Précisons que les méthodes précédemment proposées pour la construction des automates reconnaissant les langages LL′ et L* reviennent très précisément à utiliser la présente approche d'ajout de transitions spontanées entre état terminaux et initiaux, suivie de l'élimination de ces transitions spontanées par une fermeture arrière[32].

5.3 Des automates aux expressions régulières

Automates étiquetés par des expressions régulières

Dans la démonstration du théorème de Kleene, il nous restait à justifier qu'il est toujours possible, à partir d'un automate fini (déterministe ou non), de trouver une expression régulière dont l'interprétation est le langage reconnu par l'automate.

Pour ce faire, une façon de procéder consiste à généraliser encore davantage la notion d'automate, et de permettre l'étiquetage des transitions, non plus par des symboles de Σ ou $\Sigma \cup \{\epsilon\}$, mais par des expressions régulières.

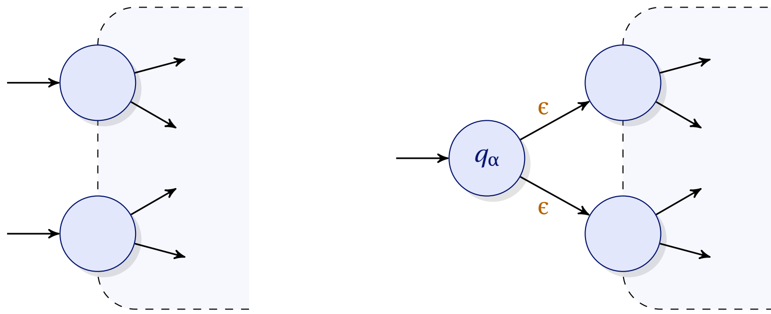
Le fonctionnement de tels automates est similaire à ceux déjà décrits, même si nous ne tenterons pas de le formaliser complètement car cela présente des difficultés qui nous importent présentement peu.

Principe de la transformation

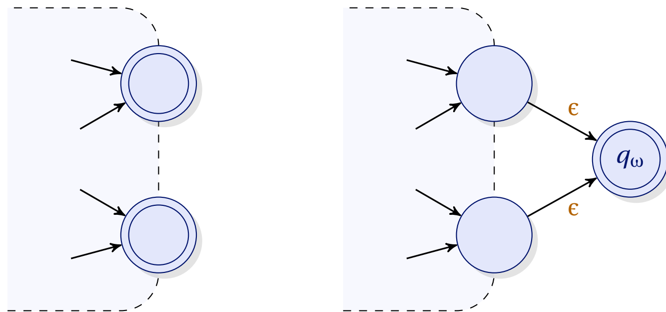
Tout automate fini, déterministe ou non, est un cas particulier d'un tel automate, en considérant les symboles de σ comme les expressions régulières qui leurs sont associées. Pour retrouver l'expression régulière dont l'interprétation est le langate reconnu par l'automate, nous allons, par des opérations simples, transformer notre automate en des automates équivalents successifs, jusqu'à obtenir un automate trivial à deux états, l'un initial, l'autre terminal, et une seule transition, dont l'étiquette sera naturellement l'expression régulière recherchée.

32. Une fermeture avant conviendrait également, mais elle conduirait à des automates légèrement différents.

Il est plus simple d'opérer des transformations sur un automate standard (ayant donc un seul état initial sur lequel n'arrive aucune transition), aussi commence-t-on en créant un état initial supplémentaire q_α, en plaçant des transitions spontanées entre cet état et tous les états initiaux de l'automate, et en retirant le caractère initial de ces derniers.

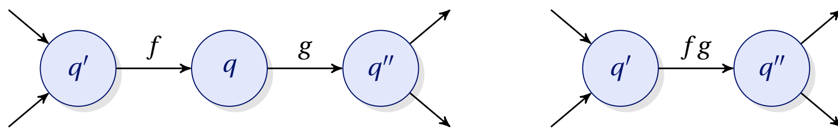


De même, on crée un état terminal q_ω, des transitions spontanées de chaque état terminal de l'automate vers cet état q_ω, et on retire le caractère terminal de ces états pour n'avoir qu'un seul état terminal d'où ne part aucune transition.

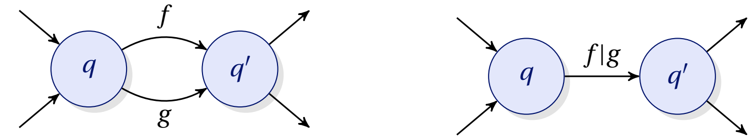


Ceci fait, on applique, dans un ordre quelconque et jusqu'à parvenir au résultat souhaité, quatre règles :

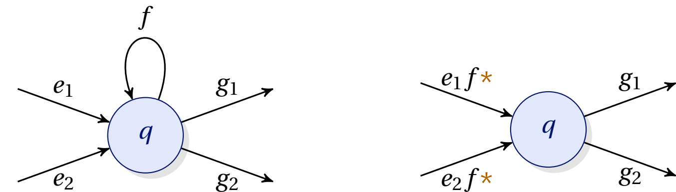
- si un état q est tel qu'une unique transition, étiquetée par f, y mène (depuis un état q') et qu'une seule transition, étiquetée par g, le quitte (vers un état[33] q''), alors on peut supprimer l'état q et ces deux transitions, et ajouter une transition directe de q à q' étiquetée par fg ;



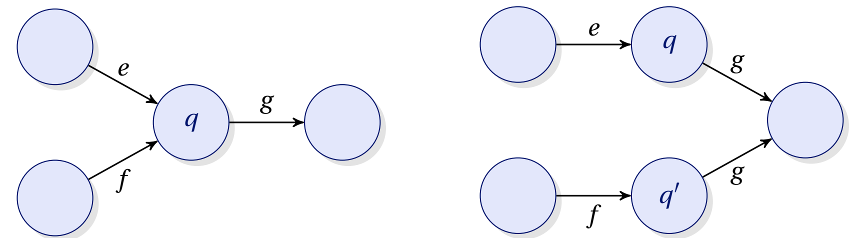
- si deux transitions, étiquetées respectivement par des expressions régulières f et g, mènent d'un même état q à un même état[34] q', on peut les remplacer par une unique transition étiquetée par $f\,|\,g$;



- si l'on a une unique boucle[35] sur un état q, étiquetée par une expression régulière f, on peut supprimer cette boucle en concaténant f à chacune des expressions régulières étiquetant les transitions menant à l'état q ;



- si un état q a plusieurs transitions qui y mène et/ou plusieurs transitions qui en parte, on pourra le remplacer par autant d'états qu'il y a de couples transition entrante/transition sortante, chacun n'ayant qu'une seule transition entrante et une seule transition sortante. Par exemple, un état avec deux transitions entrantes et une transition sortante sera dédoublé de la sorte :



On peut montrer que ces quatre règles n'altèrent pas le langage reconnu, et suffisent pour parvenir[36] à un automate à deux états q_α et q_ω, liés par une unique transition. En effet, la dernière règle permet d'obtenir des états avec une seule transition entrante et une seule transition sortante, état qui peut ensuite être éliminé avec la première règle. Les deux autres règles permettent d'éliminer les boucles et les transitions multiples qui peuvent être présentes au début, ou apparaître lors des transformations[37].

33. Cette règle peut être appliquée lorsque $q'' = q'$.

34. Cette règle peut être appliquée lorsque $q' = q$.

35. S'il y en a plusieurs, on appliquera d'abord la règle précédente.

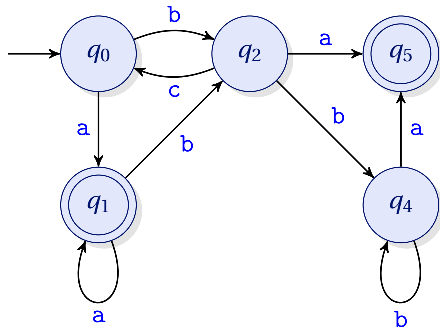
36. Après un émondage préalable si nécessaire.

37. Pour en faire une vraie preuve, il faudrait toutefois justifier la terminaison de cette méthode, qui n'est pas triviale.

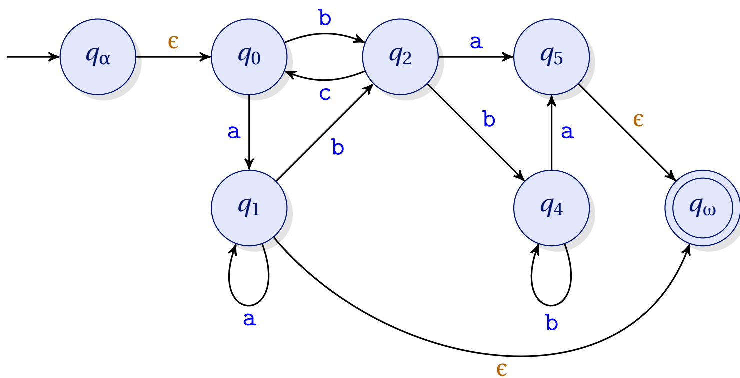
Cette approche systématique, connue sous le nom d'algorithme de Brzozowski et McCluskey, permet de justifier la réciproque du théorème de Kleene présenté tantôt, à savoir qu'à tout automate fini reconnaissant un langage L on peut associer une expression régulière dont l'interprétation est ce langage L, et donc que tout langage reconnaissable est régulier.

Exemple

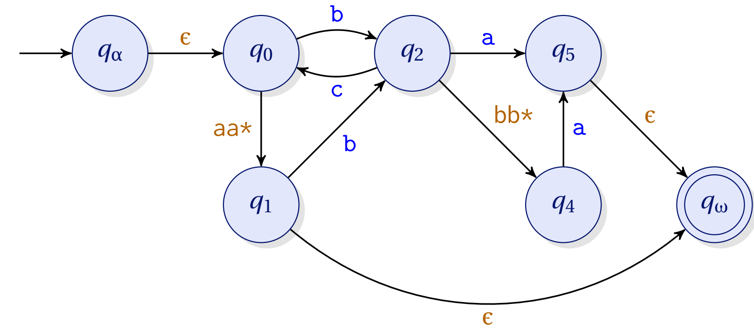
Pour illustrer cette démarche, reprenons le cas de l'automate fini étudié précédemment, et appliquons ces transformations afin de retrouver l'expression régulière dont l'interprétation est le langage que cet automate reconnaît :



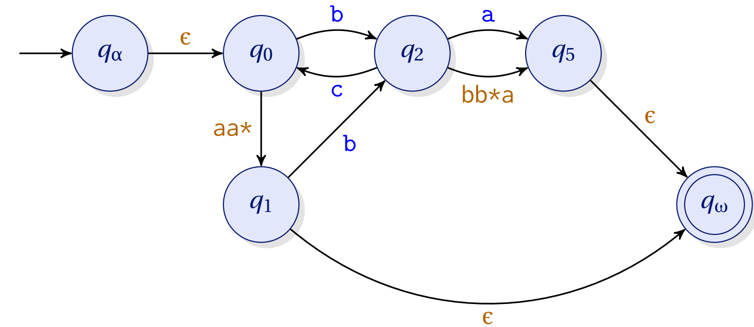
On commence par créer un nouvel état initial q_α, relié à q_0 par une transition spontanée (et on retire le caractère initial de q_0), ainsi qu'un nouvel état terminal q_ω, et on lie q_1 et q_5 à cet état q_ω par des transitions spontanées, avant de retirer le caractère terminal de q_1 et q_5. Cela donne :



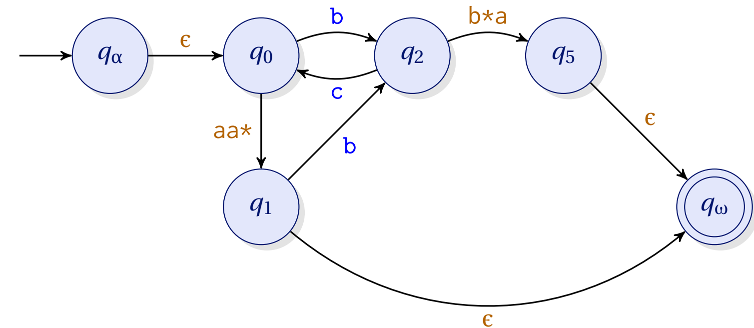
On peut à présent appliquer les règles pour simplifier notre automate. On commence par l'utilisation de la troisième règle pour procéder à l'élimination des deux boucles :



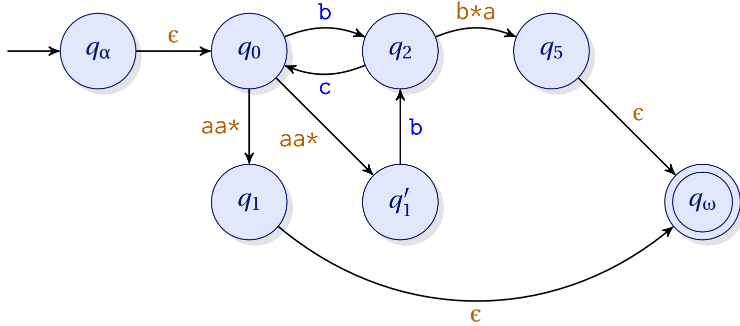
On poursuit par l'élimination de l'état q_4, qui n'a qu'une seule transition entrante et seule une transition sortante :



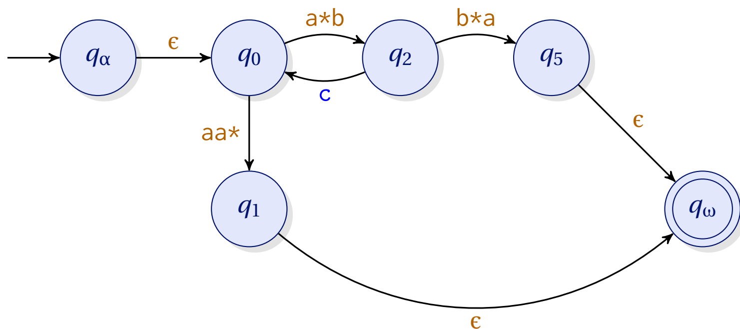
Les deux transitions étiquetées a et bb*a entre les états q_2 et q_5 peuvent être rassemblées en une unique transition étiquetée (ε|bb*)a. En remarquant que (ε|bb*) est équivalent à b*, on peut étiqueter cette transition plus simplement b*a :



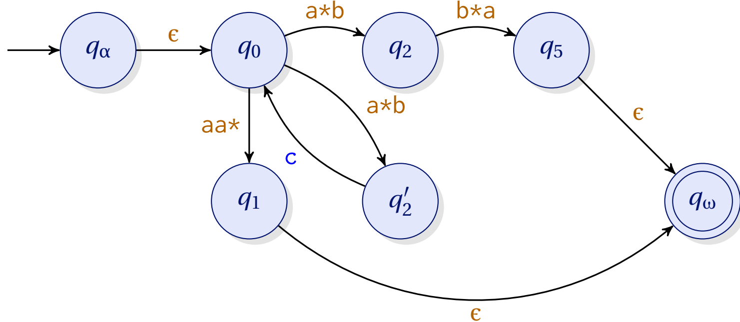
Portons à présent notre attention sur l'état q_1. Comme deux transitions en partent, nous allons le dédoubler en un état q_1 et un état q_1', un pour chacune des transitions sortantes :



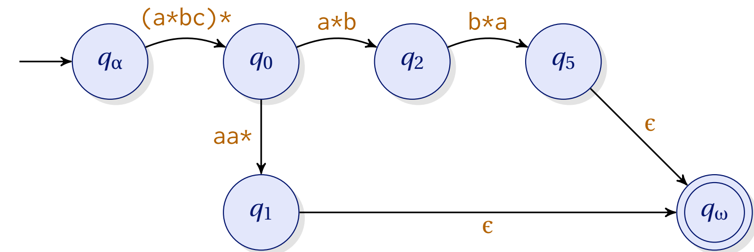
On peut ensuite éliminer cet état q_1' au profit d'une transition étiquetée `aa*b` menant de q_0 à q_2, qui se combine à l'autre transition de q_0 à q_2 en une unique transition étiquetée `a*b` :



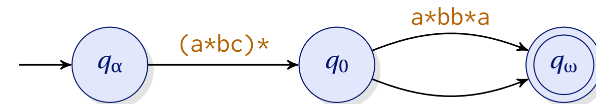
L'état q_2 ayant deux transitions sortantes, on le sépare à son tour en deux états q_2 et q_2' :



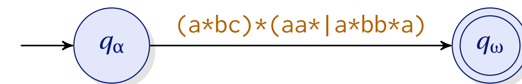
On élimine ensuite l'état q_2', ce qui produit une boucle étiquetée par `a*bc` sur l'état q_0, boucle que l'on élimine en concaténant `(a*bc)*` à droite des étiquettes des transitions menant à q_0 :



On peut ensuite éliminer les états q_2, q_5 et q_1 :



Puis la double transition de q_0 à q_ω, avant d'éliminer l'état q_0, et de parvenir à un automate réduit à deux états q_α et q_ω liés par une unique transition :



Cette transition est étiquetée par une expression régulière qui s'interprête en le langage reconnu par notre automate.

On peut ensuite remarquer que `aa*` est équivalent à `a*a`, donc `aa*|a*bb*a` est équivalent à `a*(ϵ|bb*)a`, et donc à `a*b*a`. Par conséquent, `(a*bc)*(aa*|a*bb*a)` est équivalent à `(a*bc)*a*b*a`. Mais `(a*bc)*a*` est équivalent à `(a|bc)*`. L'automate reconnaît donc l'interprétation de `(a|bc)*b*a`. On retrouve bien l'expression régulière à partir de laquelle on avait construit l'automate.

6 Lemme de l'étoile

6.1 Principe

Nous avons vu qu'il existe de nombreux moyens de définir un langage. Certains d'entre eux ne sont pas réguliers (pas reconnaissables). C'est par exemple le cas de $L = \{ a^n b^n \mid n \in \mathbb{N} \}$ ou de $L = \{ a^n \mid n \text{ premier} \}$.

Déterminer si un langage donné est régulier ou non n'est pas une tâche facile. Pour montrer qu'un langage est régulier, on peut exhiber un automate le reconnaissant. Pour

montrer qu'un langage n'est pas régulier, une solution est d'utiliser le *lemme de l'étoile*, également appelé *lemme de pompage* :

Lemme 5 (de l'étoile). *Si L est un langage régulier sur un alphabet Σ, alors il existe $k \in \mathbb{N}$ tel que tout mot $w \in \Sigma^*$ de longueur $|w|$ supérieure ou égal à k se factorise sous la forme $w = u \cdot v \cdot u'$ avec :*

- $|v| \geqslant 1$;
- $|u \cdot v| \leqslant k$;
- $\forall n \in \mathbb{N},\ u \cdot v^n \cdot u' \in \mathrm{L}$.

Démonstration. Soit L un langage régulier, donc reconnaissable. Par conséquent, il existe un automate fini $\mathrm{A} = (\mathrm{Q}, q_0, \mathrm{F}, \delta)$ reconnaissant L.

Posons $k = |\mathrm{Q}|$ le nombre d'états de l'automate A.

Si L ne contient aucun mot de longueur supérieure ou égale à w, on en a terminé.

Sinon, soit $w = s_1 s_2 \ldots s_n$ un mot quelconque de L tel que $|w| \geqslant k$. Il existe un chemin acceptant dans A étiqueté par w :

$$q_0 = q_{i_0} \xrightarrow{s_1} q_{i_1} \xrightarrow{s_2} q_{i_2} \xrightarrow{s_3} \ldots \xrightarrow{s_n} q_{i_n} \in \mathrm{F}$$

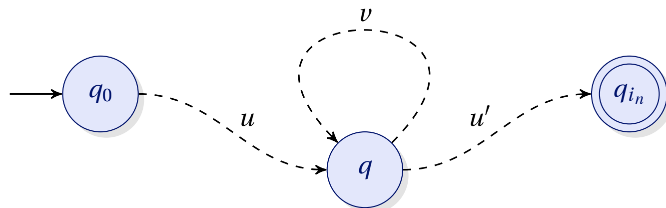
Parmi les $k+1$ premiers états traversés par le chemin acceptant, puisqu'il n'existe que k états distincts, au moins un d'entre eux est visité deux fois.

Autrement dit, il existe $0 \leqslant j < j' \leqslant k$ tels que $q_{i_j} = q_{i_{j'}}$. Notons q un tel état.

Séparons le chemin menant de q_0 à q_{i_n} en trois morceaux, et notons u, v et u' les étiquettes de ces trois chemins. On a donc

$$q_0 = q_{i_0} \xrightarrow{u} q \xrightarrow{v} q \xrightarrow{u'} q_{i_n} \in \mathrm{F}$$

Il y a donc un chemin menant de q à q étiqueté par v, avec $|v| \geqslant 1$. On peut voir ceci comme une boucle dans l'automate :



On peut emprunter cette boucle un nombre quelconque de fois, le chemin restera acceptant. Par exemple, le chemin étiqueté par $u \cdot v \cdot v \cdot u'$ est acceptant :

$$q_0 = q_{i_0} \xrightarrow{u} q \xrightarrow{v} q \xrightarrow{v} q \xrightarrow{u'} q_{i_n} \in \mathrm{F}$$

Par conséquent, $\forall n \in \mathbb{N},\ u \cdot v^n \cdot u' \in \mathrm{L}$, avec $|v| \geqslant 1$ et $|u \cdot v| \leqslant k$. □

Attention, le lemme de l'étoile n'exprime pas une équivalence : il existe des langages non-réguliers qui vérifient les hypothèses du lemme de pompage.

Comme nous l'avons laissé entendre, en général, ce lemme sert à prouver qu'un langage n'est pas régulier, en montrant que l'on peut construire des mots arbitrairement grands qu'il n'est pas possible de factoriser avec les conditions proposées.

6.2 Exemples d'utilisation

Lemme 6. *Le langage $\mathrm{L} = \{\mathtt{a}^n\mathtt{b}^n \mid n \in \mathbb{N}\}$ n'est pas régulier.*

Démonstration. Supposons que w se factorise sous la forme $w = u \cdot v \cdot u'$, avec $|v| \geqslant 1$ et $\forall n \in \mathbb{N},\ u \cdot v^n \cdot u' \in \mathrm{L}$. On a en particulier $u \cdot v^2 \cdot u' \in \mathrm{L}$.

Tout mot de L contient autant de $\mathtt{a}$ que de $\mathtt{b}$.

On a donc notamment $\left|u \cdot v \cdot u'\right|_a = \left|u \cdot v \cdot u'\right|_b$, et $\left|u \cdot v^2 \cdot u'\right|_a = \left|u \cdot v^2 \cdot u'\right|_b$,

Par conséquent, $|v|_a = |v|_b$, et donc $|v|$ est pair. En outre, puisque le langage ne peut contenir de facteur $\mathtt{ba}$, on a nécessairement $v = \mathtt{a}^{|v|/2}\mathtt{b}^{|v|/2}$.

D'où $u \cdot v^2 \cdot u' = u \cdot \mathtt{a}^{|v|/2}\mathtt{b}^{|v|/2}\mathtt{a}^{|v|/2}\mathtt{b}^{|v|/2} \cdot u'$.

Ce mot ne peut pas être de la forme $\mathtt{a}^n\mathtt{b}^n$, et donc n'appartient pas à L, ce qui est contraire avec nos hypothèses. □

Lemme 7. *Le langage $\mathrm{L} = \{\mathtt{a}^n \mid n \text{ premier}\}$ n'est pas régulier.*

Démonstration. Supposons que w se factorise sous la forme $w = u \cdot v \cdot u'$, avec $|v| \geqslant 1$ et $\forall n \in \mathbb{N},\ u \cdot v^n \cdot u' \in \mathrm{L}$.

On a donc $\left|u \cdot v^n \cdot u'\right| = |u| + n \times |v| + \left|u'\right|$ premier pour tout $n \in \mathbb{N}$.

Posons $p = |u| + \left|u'\right|$ et $q = |v| \geqslant 1$. On a donc $p + nq$ premier pour tout n.

Si $p = 0$, alors nq est premier pour tout n. Si $q = 1$, tous les entiers sont premiers, ce qui est absurde. Si $q > 1$, pour tout $n \geqslant 2$, on aurait une décomposition d'un nombre premier en un produit de deux entiers strictement supérieurs à 1, c'est absurde.

Si $p = 1$, alors $1 + nq$ est premier pour tout n, et donc $1 + (2+q)q = 1 + 2q + q^2 = (q+1)^2$ est premier, ce qui est absurde.

Si $p \geqslant 2$, puisque $p + nq$ est premier pour tout n, $p + pq$ est premier, donc $p(q+1)$ est premier. Là encore, on a une décomposition en un produit d'entiers strictement supérieurs à 1, ce qui est absurde. □