Notes de cours sur les automates (NFP108)

F. Barthélemy

10 octobre 2018

Avertissement : ce document est en cours d'élaboration et susceptible d'évoluer. Il est dans un état provisoire.

1 Introduction

Les automates finis à états (automates finis en abrégé) offrent un formalisme de description peu puissant mais avec beaucoup d'algorithmes efficaces. Ils sont très utilisés notamment dans deux domaines : le traitement de chaînes de caractères et la description de comportement dynamique de systèmes.

2 Un premier exemple

Un automate fini est un graphe orienté dont les arcs sont étiquetés par des symboles. Voyons un exemple.

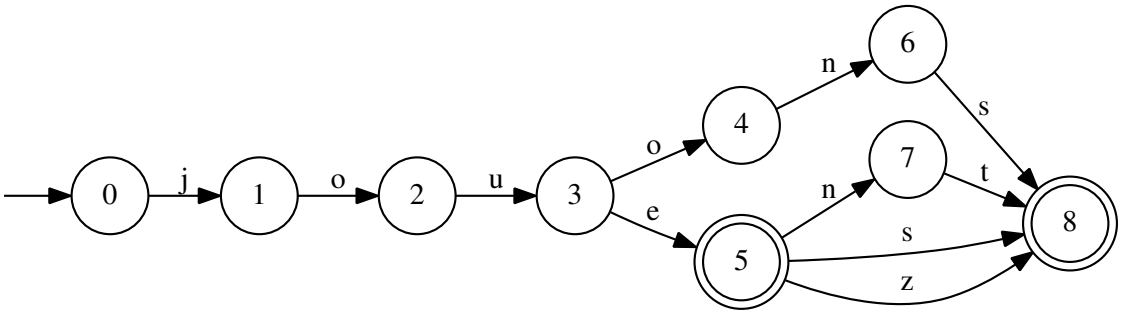


FIGURE 1 – exemple d'automate fini

La notation du graphe met en évidence deux types de noeuds particulier. D'une part l'état initial, qui est caractérisé par une flèche entrante venant de nulle part et sans étiquette (ici l'état 0). D'autre part les états finals, identifiés par un double cercle (ici les états 5 et 8). Il y a toujours exactement un état initial, alors qu'il peut y avoir 0, 1 ou plusieurs états finals. Les noeuds du graphe sont appelés états et les arcs sont appelés des transitions. Dans un tel graphe, on s'intéresse

aux chemins qui vont de l'état initial à un état final. A chaque chemin est associée la chaîne des étiquettes des arcs parcourus.

Par exemple, il y a un chemin qui part de l'état 0, passe par les états 1, 2, 3 et arrive en 5, qui est un état final. La chaîne correspondante est *joue*. Il existe de même des chemins pour les chaînes *joues, jouent, jouez et jouons*. L'ensemble de ces chaînes forme ce que l'on appelle le langage de l'automate. Cet exemple est donc un automate qui représente le présent de l'indicatif du verbe *jouer*. Notez que des chemins différents partagent des parties communes. Par exemple, ici, le préfixe *jou* est représenté une fois alors qu'il appartient à toutes les chaînes.

3 Automates finis

3.1 Définitions de base

Définition 1 *Alphabet, chaîne*
On appelle alphabet un ensemble fini de symbole. Une chaîne sur un alphabet Σ est une séquence éventuellement vide de symboles de Σ. La séquence vide est notée ϵ (epsilon). Les autres séquences sont notées par la juxtaposition des symboles qui les composent.

Définition 2 *Langage*
Un langage est un ensemble de chaînes sur un alphabet Σ.

Définition 3 *Automate fini*
Un automate fini est un quintuplet $A = (\Sigma, Q, \delta, i, F)$ où :
- *Σ est un ensemble fini de symboles appelé alphabet.*
- *Q est un ensemble fini dont les éléments sont appelés* états.
- *δ est une relation de $Q \times \Sigma \times Q$ appelée* transition *ou* ensemble des transitions *de A.*
- *i est un état de Q appelé* état initial.
- *F est un sous-ensemble de Q appelé* ensemble des états finals *de A.*

L'ensemble des transitions δ est une relation, c'est-à-dire un ensemble de triplets. Cet ensemble est nécessairement fini puisque Q et Σ sont finis. Un automate fini est fait de composantes qui sont toutes finies (Σ, Q, δ, F), d'où le qualificatif de *fini*.

Définition 4 *Représentation graphique*
Un automate fini peut être représenté graphiquement comme un graphe orienté dont les sommets sont les états et les arcs sont les transitions. Une transition (q_1, x, q_2) est représentée par un arc reliant les sommets q_1 et q_2, étiqueté par x.

Nous avons déjà vu un exemple de représentation graphique (figure 1). Reprenons pour cet exemple les différentes définitions. L'alphabet de l'automate est l'ensemble $\{e, j, n, o, s, t, u, z\}$. Le quintuplet décrivant l'automate est le suivant : $(\Sigma, \{0, 1, 2, 3, 4, 5, 6, 7, 8\}, \delta, 0, \{4, 8\})$ avec
- $\Sigma = \{e, j, n, o, s, t, u, z\}$

— $\delta = \{(0, j, 1), (1, o, 2), (2, u, 3), (3, e, 4), (3, o, 5), (4, n, 6), (4, s, 8), (4, z, 8), (5, n, 7), (6, t, 8), (7, s, 8)\}$

Définition 5 *Chemin*
Soit $A = (\Sigma, Q, \delta, i, F)$ un automate. Un chemin de cet automate est une séquence $(q_0, x_0, q_1)(q_1, x_1, q_2) \dots (q_{n-1}, x_{n-1}, q_n)$ de transitions de δ telle que $n \geq 0$. La chaîne associée à ce chemin est $x_0 x_1 \dots x_{n-1}$. On notera un chemin de la façon suivante : $q_0 \overset{x_0}{\to} q_1 \overset{x_1}{\to} q_2 \dots q_{n-1} \overset{x_{n-1}}{\to} q_n$

Cette notion de chemin correspond à celle de chemin en théorie des graphes.

Définition 6 *Chemin succès*
Soit $A = (\Sigma, Q, \delta, i, F)$ un automate. Un chemin $q_0 \overset{x_0}{\to} q_1 \overset{x_1}{\to} q_2 \dots q_n$ de cet automate est appelé un **succès** *si $q_0 = i$ et $q_n \in F$.*

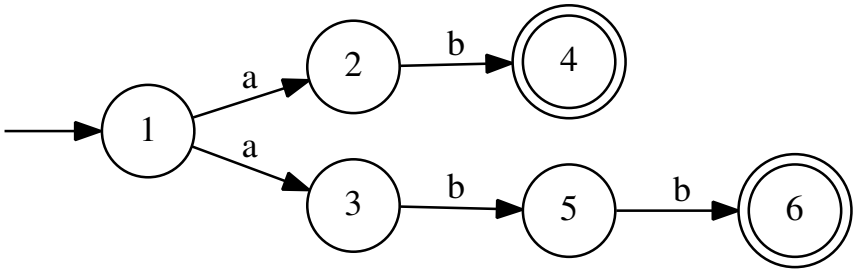
En d'autres termes, un chemin succès est un chemin qui part de l'état initial et qui arrive dans un état final.

Définition 7 *Chaîne associée à un chemin*
Soit $A = (\Sigma, Q, \delta, i, F)$ un automate et $q_0 \overset{x_0}{\to} q_1 \overset{x_1}{\to} q_2 \dots q_n$ un chemin de A. La chaîne associée à ce chemin est $x_0 \dots x_{n-1}$.

Définition 8 *Langage défini par un automate fini*
Soit A un automate. Le langage défini par cet automate est l'ensemble des chaînes associées aux chemins succès de cet automate.

Sur l'exemple de la figure 1, voici deux exemples de chemins.
— $0 \overset{j}{\to} 1 \overset{o}{\to} 2 \overset{u}{\to} 3 \overset{e}{\to} 4$
— $0 \overset{j}{\to} 1 \overset{o}{\to} 2 \overset{u}{\to} 3 \overset{o}{\to} 5 \overset{n}{\to} 7$
Les deux chemins commencent dans l'état initial 0. Le premier chemin se termine dans l'état 4 qui est un état final. Donc ce chemin est un succès. Le second chemin se termine dans l'état 7 qui n'est pas final. Ce n'est donc pas un succès. Les chaînes associées à ces deux chemins sont respectivement *joue* et *jouon*. Le language défini par l'automate est l'ensemble $\{joue, joues, jouons, jouez, jouent\}$.

Attention, une chaîne appartient à un langage s'il existe un chemin succès associé à cette chaîne. Cela ne signifie pas que tous les chemins partant de l'état initial et associés à cette chaîne sont des succès. Par exemple dans l'automate suivant, il y a deux chemins pour la chaîne ab.



Il y a le chemin $1 \xrightarrow{a} 2 \xrightarrow{b} 4$ qui est un succès parce que 4 est un état final et $1 \xrightarrow{a} 3 \xrightarrow{b} 5$ qui est un échec parce que 5 n'est pas un état final. La chaîne appartient au langage de l'automate puisqu'il existe un chemin succès. Donc l'existance d'un échec ne dit rien sur l'appartenance de la chaîne au langage.

3.2 Exécution d'un automate fini

3.2.1 Définitions des notions liées à l'exécution

Nous avons vu dans la section précédente qu'un automate fini définit un langage, c'est-à-dire un ensemble de chaîne. Mais un automate est aussi une machine, que l'on peut exécuter pour tester l'appartenance d'une chaîne à ce langage ou pour générer les chaînes de ce langage.

Nous allons d'abord voir l'exécution d'un automate en reconnaissance.

Définition 9 *Parcours partiel, parcours complet, parcours succès*
Un parcours partiel d'automate est une paire comprenant un chemin partant de l'état initial et une chaîne. Un parcours complet est un parcours comprenant un chemin partant de l'état initial et la chaîne vide. Un parcours succès est un parcours complet dont le chemin est un succès, c'est à dire un chemin se terminant dans un état final.

Un parcours partiel reflète un état intermédiaire d'un parcours d'un chemin. Le dernier état du chemin donne l'état dans lequel on est arrivé et la chaîne est ce qu'il reste à reconnaître sur la suite du chemin. C'est un suffixe de la chaîne que l'on teste.

Définition 10 *Pas de calcul, calcul, calcul succès*
Un pas de calcul permet de passer d'un parcours partiel $(i \xrightarrow{x} \dots q, aw)$ *où* $a \in \Sigma$ *et* $w \in \Sigma^*$ *à un parcours partiel* $(i \xrightarrow{x} \dots q \xrightarrow{a} q', w)$ *s'il existe une transition* (q, a, q') *dans* δ.
Un calcul est une succession de parcours partiels $p_1, \dots p_n$ *tels que pour tout* i*, on peut passer de* p_i *à* p_{i+1} *par un pas de calcul.*
Un calcul succès est un calcul $p_1, \dots p_n$ *tel que* p_n *est un parcours succès.*

Un calcul est un moyen de parcourir un chemin du graphe à la recherche d'un chemin succès (c'est à dire un chemin partant de l'état initial et arrivant dans l'état final, étiqueté par w).

Propriété 1 *Soit* $A = (\Sigma, Q, \delta, i, F)$ *un automate. Une chaîne* w *appartient à* $L(a)$ *si et seulement si il existe un calcul succès commençant du parcours partiel* (i, w).

Attention, il faut qu'un calcul succès existe, cela ne signifie pas que tous les calculs partant de cette configuration doivent être des succès. S'il y a plusieurs chemins dans le graphe partant de l'état initial qui sont étiquetés par w, il suffit qu'un de ces chemins arrive dans un état final pour que w appartienne au langage. Pour prouver que w appartient à $L(A)$, il faut trouver un chemin succès. Pour prouver que w **n'appartient pas** à $L(A)$, il faut prouver qu'aucun chemin n'est un succès. Il faut donc avoir regardé **tous** les chemins pour tirer cette conclusion.

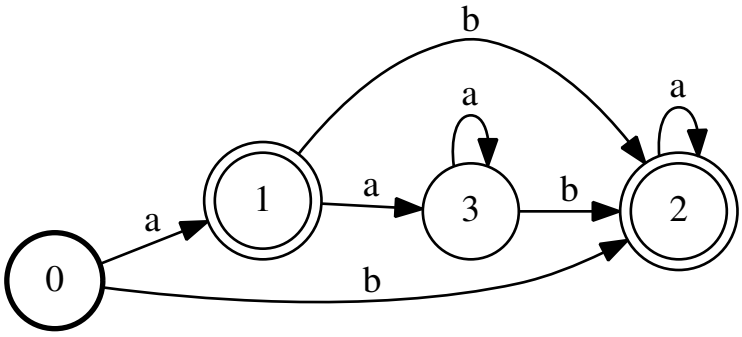


FIGURE 2 – automate à exécuter

3.2.2 Exemple d'exécution

Prenons comme exemple l'automate de la figure 2. Voyons pour cet automate un calcul qui permet de montrer que `aaba` appartient au langage. Le parcours partiel de départ est `(0,aaba)`. Le calul succès est le suivant :
$(0, aaba), (0 \xrightarrow{a} 1, aba), (0 \xrightarrow{a} 1 \xrightarrow{a} 3, ba), (0 \xrightarrow{a} 1 \xrightarrow{a} 3 \xrightarrow{b} 2, a), (0 \xrightarrow{a} 1 \xrightarrow{a} 3 \xrightarrow{b} 2 \xrightarrow{a}, \epsilon)$

Ce calcul est un succès parce que le dernier parcour a une chaîne vide et son chemin arrive dans l'état final 2.

On peut définir un notion de calcul et d'exécution d'automate en *génération*, pour énumérer les chaînes d'un automate. Nous ne détaillerons pas cela ici. Le principe général consiste à accumuler les symboles vus sur la partie du chemin déjà parcourue au lieu de stocker le reste de ce qui reste à trouver sur le chemin non encore parcouru.

3.3 Langage régulier, non-déterminisme

Définition 11 *Langage régulier*
Un langage L est dit régulier s'il existe un automate fini A tel que $L = L(A)$.

Définition 12 *Automate fini non-déterministe*
Un automate fini $A = (\Sigma, Q, \delta, i, F)$ est dit non déterministe s'il existe dans δ deux transitions (q_1, x, q_2) et (q_1, x, q_3) telles que $q_2 \neq q_3$.

Un automate est non-déterministe si dans certains cas, il existe plusieurs chemins étiquetés par la même chaîne. Un automate fini est déterministe s'il n'est pas non-déterministe.

Définition 13 *Automates équivalents*
Deux automates A_1 et A_2 sont dits équivalents si $L(A_1) = L(A_2)$.

4 Propriétés de clôture

Nous allons nous intéresser à deux langages réguliers particuliers et à des opérations ensemblistes qui conservent la régularité.

Propriété 2 *Le langage vide est régulier.*

L'automate qui ne comprend que l'état initial qui n'est pas final ne possède aucun chemin succès. Il n'y a aucun état final, donc il n'y a pas de chemin succès. Donc le langage reconnu ne comporte aucune chaîne, c'est le langage vide (l'ensemble vide).

Propriété 3 *Le langage $\{\epsilon\}$ est régulier.*

L'automate qui ne comprend que l'état initial qui est également final reconnaît ce langage, s'il n'a aucune transition. Il convient de bien saisir la différence entre le langage vide qui ne contient aucune chaîne et ce langage qui contient une chaîne, la chaîne vide.

Propriété 4 *L'union de deux langages réguliers est un langage régulier.*

La preuve de cette propriété est basée sur un algorithme qui construit un automate reconnaissant $L(A_1) \cup L(A_2)$ étant donnés les deux automates A_1 et A_2.

$A1 = (\Sigma, Q_1, \delta_1, i_1, F_1)$ et $A2 = (\Sigma, Q_2, \delta_2, i_2, F_2)$. On suppose que Q_1 et Q_2 sont disjoints. Si ce n'est pas le cas, on peut renommer les états d'un des deux automates. Cela ne change pas le langage reconnu, puisque le nom des états n'intervient pas dans les chaînes. $A_1 \cup A_2 = (\Sigma, Q_1 \cup Q_2 \cup \{i\}, \delta, i, F)$ où

— i est un nouvel état initial n'apprtenant ni à Q_1, ni à Q_2.

— $\delta = \delta_1 \cup \delta_2 \cup \{(i, x, q) | \exists (i_1, x, q) \in \delta_1\} \cup \{(i, x, q) | \exists (i_2, x, q) \in \delta_2\}$

— $F = F_1 \cup F_2 \cup \{i\}$ si $i_1 \in F_1$ ou $i_2 \in F_2$
$F = F_1 \cup F2$ sinon.

La figure 3 montre un exemple d'union. Le nouvel état initial créé est l'état 0.

Définition 14 *Concaténation de langages*
Soient L_1 et L_2 deux langages. La concaténation de L_1 et L_2, notée $L_1.L_2$ est définie par : $L_1.L_2 = \{w_1 w_2 | w_1 \in L_1$ et $w_2 \in L_2\}$.

Propriété 5 *La concaténation de deux langages réguliers est un langage régulier.*

$A1 = (\Sigma, Q_1, \delta_1, i_1, F_1)$ et $A2 = (\Sigma, Q_2, \delta_2, i_2, F_2)$. On suppose que Q_1 et Q_2 sont disjoints. $A_1.A_2 = (\Sigma, Q_1 \cup Q_2, \delta, i_1, F)$ où

— $\delta = \delta_1 \cup \delta_2 \cup \{(q_1, x, q_2) | q_1 \in F_1$ et $(i_2, x, q_2) \in \delta_2\}$

— $F = F_1 \cup F_2$ si $i_2 \in F_2$

— $F = F_2$ sinon

Définition 15 *Clôture sous concaténation*
Soit L un langage. On appelle clôture sous concaténation de L et on note L^* le langage définit par $L^* = \{w_1 \dots w_n | n \geq 0$ et $\forall i, 1 \leq i \leq n, w_i \in L\}$.

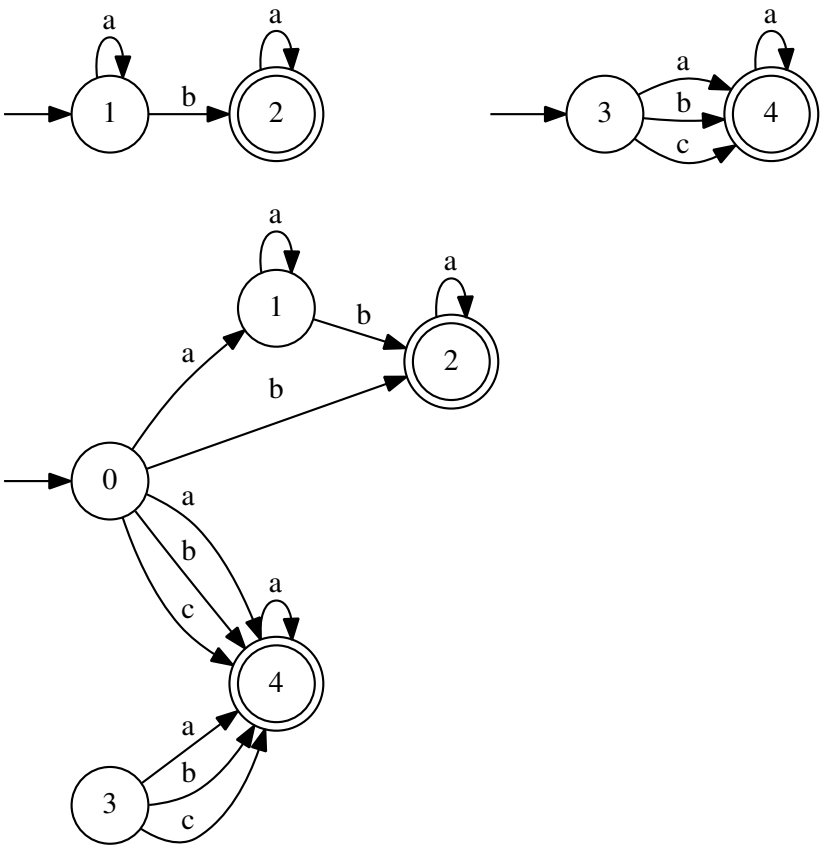


FIGURE 3 – exemple d'union d'automates finis

Propriété 6 *La clôture sous concaténation d'un langage régulier est un langage régulier.*

$A = (\Sigma, Q, \delta, i, F)$. $A^* = (\Sigma, Q \cup \{i'\}, \delta', i, F \cup \{i'\})$ où $\delta' = \delta \cup \{(q_1, x, q_2) | q_1 \in F \cup \{i'\}$ et $(i, x, q_2) \in \delta\}$

Propriété 7 *L'intersection de deux langages réguliers est un langage régulier.*

Soient $A_1 = (\Sigma, Q_1, \delta_1, i_1, F_1)$ et $A_2 = (\Sigma, Q_2, \delta_2, i_2, F_2)$. $A_1 \cap A_2 = (\Sigma, Q_1 \times Q_2, \delta, (i_1, i_2), F_1 \times F_2)$ avec $\delta = \{((q_1, q_2), x, (r_1, r_2)) | (q_1, x, r_1) \in \delta_1, (q_2, x, r_2) \in \delta_2\}$

Propriété 8 *Le complémentaire d'un langage régulier L, noté $\overline{L}$, est un langage régulier.*

Soit $A = (\Sigma, Q, \delta, i, F)$ un automate. Pour calculer l'automate définissant $\overline{L(A)}$, on procède en deux temps. D'abord on complète l'automate A au moyen d'un état dit *état poubelle* et de transitions allant des autres états vers cet état. Pour tout état de Q et tout symbole x de Σ, soit il existe une transition (q, x, r) dans δ, soit on ajoute une transition $(q, x, poub)$ à l'automate ($poub$ étant l'état poubelle). Ensuite, on inverse le statut des états : les états finals deviennent non finals et les états non finals deviennent finals.

$\overline{A} = (\Sigma, Q_1 \cup \{poub\}, \delta', i, Q_1 - F)$ avec $\delta' = \delta \cup \{(q, x, poub)|$ il n'existe pas d'état r tel que $(q, x, r) \in \delta\}$.

Propriété 9 *La différence ensembliste de deux langages réguliers est un langage régulier.*

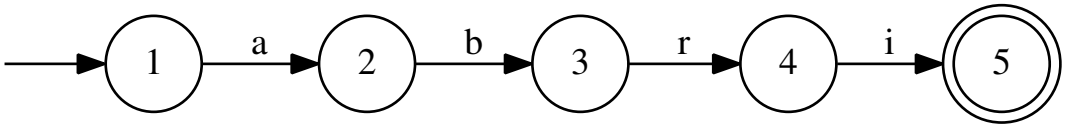
Cette propriété est la conséquence des deux propriétés précédentes. En effet, $L_1 - L_2 = L_1 \cap \overline{L_2}$.

4.1 Exemple d'utilisation des opérations

Les différentes propriétés étudiées dans cette section ont un intérêt direct pour la spécification. Par exemple, l'union est la base de la modularité. La différence ensembliste permet un traitement facile des exceptions. Illustrons cela sur un exemple.

Supposons que l'on veuille représenter avec un automate fini tous les mots du français pour une application de correction orthographique. On va faire une liste de tous les noms au singulier.

Chaque nom peut facilement être représenté avec un automate fini qui n'a qu'un chemin succès correspondant à ce mot. Par exemple `abri` est représenté par :



Le lexique de tous les noms au singulier peut être obtenu en faisant l'union de tous les automates ainsi construits. Appelons `noms_singulier` cet automate.

Pour avoir maintenant les noms au pluriel, il faut ajouter un s à la fin des mots. Cela peut être obtenu au moyen de l'opération de concaténation.

Mais il y a certaines exceptions qui ont un pluriel en `x`. On peut faire la liste de ces exceptions (pou, hibou, chou, etc). Appelons `exceptions` l'automate construit à partir de cette liste. Appelons `s_final` et `x_final` les deux automates qui contiennent respectivement les chaînes `s` et `x`.

La liste des mots au pluriel peut s'obtenir par :
`((noms_singulier-exceptions).s_final)∪(exceptions.x_final)`

Dans une application réelle, il faut gérer aussi d'autres exceptions (pluriel des mots en *al*, en *ail*, en *eu* et pluriels irréguliers). Cela peut se faire avec le même genre d'utilisation des opérations ensemblistes.

5 Expression régulières

Les expressions régulières sont une façon de noter un langage régulier. Nous allons d'abord les définir, puis ensuite montrer qu'elles recouvrent exactement la notion de langage régulier.

Définition 16 *Expression régulière*
Soit Σ un alphabet.

- *$\emptyset$ est une expression régulière dénotant le langage vide.*
- *ϵ est une expression régulière dénotant le langage $\{\epsilon\}$.*
- *soit $x \in \Sigma$. x est une expression régulière dénotant le langage $\{x\}$.*
- *Supposons que p et q sont deux expressions régulières dénotant les langages L_1 et L_2. Alors,*
  - *$p|q$ est une expression régulière dénotant le langage $L_1 \cup L_2$.*
  - *pq est une expression régulière dénotant le langage $L_1.L_2$.*
  - *p^* est une expression régulière dénotant le langage L_1^*.*

On peut étend la notation des expressions régulières avec des parenthèses facultatives et avec la notation p^+ pour abréger l'expression régulière p^*p.

Deux exemples d'expressions régulières :

- `(a|b)(a|c)` dénote le langage `{aa,ac,ba,bc}`.
- ab^*a dénote l'ensemble des chaînes commençant par un a, suivies d'un nombre quelconque de b et terminée par un a.

Deux expressions régulières sont égales si elles dénotent le même ensemble régulier. Voici quelques égalités d'expressions régulières :

- $p|q = q|p$ (par commutativité de l'union d'ensemble)
- $\emptyset^* = \epsilon$
- $(p|q)|r = p|(q|r)$ (par associativité de l'union ensembliste)
- $p\epsilon = \epsilon p = p$
- $p|p = p$
- $(p^*)^* = p^*$
- $\emptyset p = p\emptyset = \emptyset$

Les langages définis par les expressions régulières et les automates finis sont les mêmes : ce sont les langages réguliers. Cela peut s'énoncer au moyen de deux propriétés (double inclusion).

Propriété 10 *Pour toute expression régulière p, il existe un automate fini qui reconnaît le langage défini par p.*

Cette propriété est une conséquence évidente des propriété de clôtures énoncées à la section précédente.

Propriété 11 *Pour tout automate fini A, il existe une expression régulière dénotant le langage $L(A)$.*

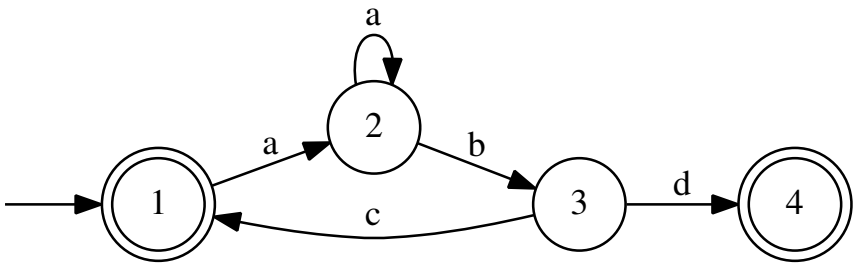
Nous n'allons pas prouver cette propriété. Nous allons donner un moyen de déterminer l'expression régulière associée à un automate fini. Pour chaque état q de l'automate, on note X_q le langage comprenant toutes les chaînes étiquetant un chemin partant de q et arrivant dans un état final. Si q est final, alors le chemin vide conduit à un état final, donc ϵ appartient à X_q. Tout chemin non vide commence par une première transition conduisant à un état q' et fini par une chemin allant de q' à un état final. Un tel chemin est étiqueté par une chaîne de $X_{q'}$. En suivant cette idée, on peut établir pour chaque X une équation composée de la disjonction de tous les moyens de faire un chemin allant dans un état final.

Le langage X_q serait le langage défini par l'automate si l'état q était l'état initial. En résolvant les équations, on trouve les valeurs des X_q sous forme d'expressions régulières et notamment le langage X_i où i est l'état initial. Ce langage est celui défini par l'automate.

Nous avons un système d'équations avec autant d'équations que d'états dans l'automate. On résoud ce système en utilisant le lemme d'Arden.

Lemme 1 *d'Arden*
Une équation du type $X = \alpha X|\beta$ où ϵ n'appartient pas à α a une solution unique $X = \alpha^*\beta$.

Prenons un exemple.



Pour chaque état, on pose une équation en regardant deux choses :
— si l'état q est final, alors ϵ appartient à X_q.
— Chaque transition qui sort de l'état est potentiellement un moyen de commencer un chemin succès. Ce chemin se poursuit par un chemin succès partant de l'état d'arrivée de la transition.

Pour l'état 3, il y a deux transitions qui sortent, donc deux façons de commencer un chemin. On pose la disjonction des deux, de la façon suivante :

$X_3 = dX_4|cX_1$

L'état 1 est final et il a une transition sortante :

$X_1 = aX_2|\epsilon$

Si l'on collecte toutes les équations, cela donne :

$X_1 = aX_2|\epsilon$

$X_2 = aX_2|bX_3$

$X_3 = dX_4|cX_1$

$X_4 = \epsilon$

On peut remplacer X_4 par sa valeur. Cela donne

$X_3 = d\epsilon|cX_1 = d|cX_1$

On peut maintenant remplacer X_3 ce qui donne l'équation :

$X_2 = aX_2|b(d|cX_1) = aX_2|bd|bcX_1$

Ensuite, on remplace X_1 dans cette équation.

$X_2 = aX_2|bd|bc(aX_2|\epsilon) = aX_2|bd|bcaX_2|bc\epsilon = (a|bca)X_2|(bd|bc)$

Cette équation a la forme requise pour appliquer le lemme d'arden. Pour appliquer ce lemme, on peut poser $\alpha = (a|bca)$ et $\beta = (bd|bc)$ Le résultat de l'application est la valeur de X_2.

$X_2 = (a|bca)^*(bd|bc)$

Ensuite, on remplace X_2 par sa valeur dans l'équation de X_1.

$X_1 = a((a|bca)^*(bd|bc))|\epsilon = a(a|bca)^*(bd|bc)|\epsilon$

C'est là le langage défini par l'automate.

5.1 Syntaxe détaillée pour les expressions régulières

Il n'existe pas de convention générale pour écrire les expressions régulières. On trouve différentes notations aussi bien dans les ouvrages de référence que dans les logiciels proposant des expressions régulières. Nous allons proposer ici une notation à utiliser dans les exercices qui seront proposés.

Nous allons reprendre en grande partie la syntaxe utilisée par OpenGrm, un des outils que nous allons utiliser en TP. Une chaîne de caractère est notée entre doubles guillemets droits. Par exemple `"abc"`. Dans une chaîne de caractère, un symbole de l'alphabet comportant plusieurs caractères dans son nom sera noté entre crochets. Par exemple `"[timeout]"`. La chaîne vide est noté sans surprise `""`. La barre oblique inversée est un caractère d'échappement permettant de noter dans une chaîne un caractère qui a un autre usage dans la syntaxe, notamment les caractères `",` `[` et `]`. Les fins de lignes et tabulations sont notées respectivement `\n` et `\t`.

Les opérateurs ensemblistes seront utilisés pour bâtir des expressions régulières. Ceux-ci porteront soit sur des chaînes de caractères, soit sur d'autres expressions régulières. L'union ou disjonction sera notée avec une barre verticale |, la concaténation sans rien ou avec un point, la différence avec un tiret, l'étoile avec une étoile, l'intersection avec une esperluette & (cette opération n'existe pas dans OpenGrm). Les parenthèses serviront comme d'habitude à indiquer des priorités dans les expressions.

Exemples d'usage d'opérateurs :

— `"a"|"b"`

— `"ab" "cd"`

— `"ab"* - "abab"`

— `("ab"* - "abab")&"bcde"`

On pourra donner un nom à une expression régulière puis utiliser ce nom dans les expressions régulières qui suivront. Par exemple :

```
voyelles = "a"|"e"|"i"|"o"|"u" ;
syllabe = "c" voyelle | "c" voyelle "c";
```

Deux opérateurs supplémentaires sont définis comme des raccourcis :
— `w+` dénote `w*w` (une séquence non vide d'occurrences de w)
— `w?` dénote `w|""` (une occurrence optionnelle de w)
On pourra utiliser une notation d'intervalle pour désigner la disjonction de tous les caractères compris dans cet intervalle. La notation sera la suivante : `"a".."e"` comme abréviation pour l'expression `"a"|"b"|"c"|"d"|"e"`. Notez que cette notation n'existe pas dans OpenGrm.

6 Déterminisation, minimisation, epsilon transition

Propriété 12 *Pour tout langage régulier L, il existe un automate fini déterministe A tel que $L = L(A)$.*

Par définition, un langage est régulier s'il existe un automate fini qui le reconnaît. La propriété énoncée spécifie de plus que s'il existe un automate qui reconnaît un langage, alors il existe nécessairement un automate fini déterministe qui le reconnaît. Cette propriété est intéressante opérationnellement, car, si l'on peut utiliser cet automate déterministe, on pourra savoir si une chaîne appartient ou non au langage en effectuant un seul calcul de l'automate.

Propriété 13 *Il existe un algorithme dit* algorithme de déterminisation *qui pour tout automate fini non déterministe A, calcule un automate fini déterministe A' tel que $L(A') = L(A)$.*

Cet algorithme utilise la notion d'ensemble de successeurs d'un état pour un symbole donné.

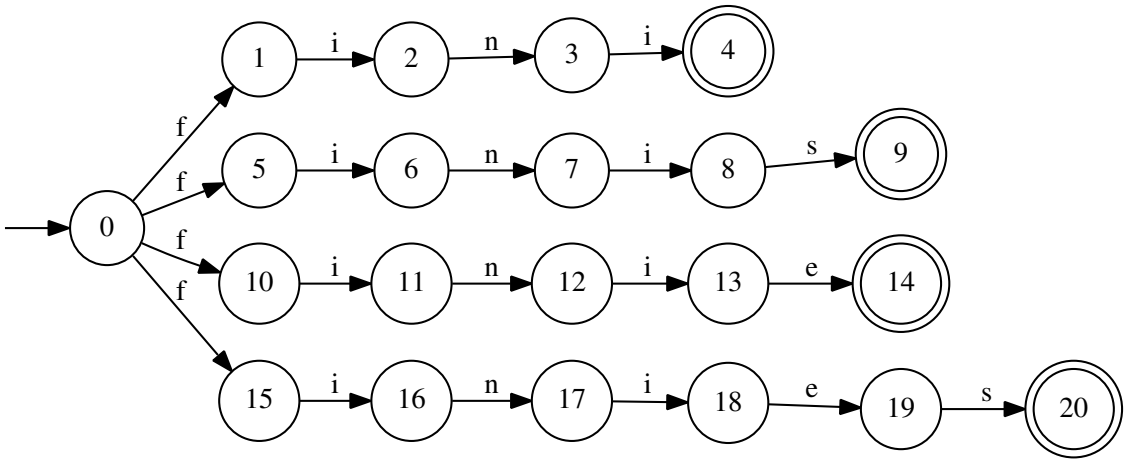
Définition 17 *Ensemble de successeurs*
Soit $A = (\Sigma, Q, \delta, i, F)$ un automate. On appelle ensemble des successeurs de l'état q pour un symbole x et on note succ(q, x) l'ensemble des états r tels qu'il existe une transition (q, x, r) dans δ.

Soit $A = (\Sigma, Q, \delta, i, F)$ un automate fini non-déterministe. On définit $det(A) = (\Sigma, 2^Q, \delta', \{i\}, F')$ avec
— $\delta' = \{(M, x, N) | N = \bigcup_{q \in M} \text{succ}(q, x)\}$
— $F' = \{M \in 2^Q | M \cap F \neq \emptyset\}$
Rappel : 2^Q est une notation pour désigner l'ensemble des parties de l'ensemble Q. On peut montrer que $det(A)$ est déterministe et que $L(det(A)) = L(A)$ (ce que nous ne ferons pas ici).

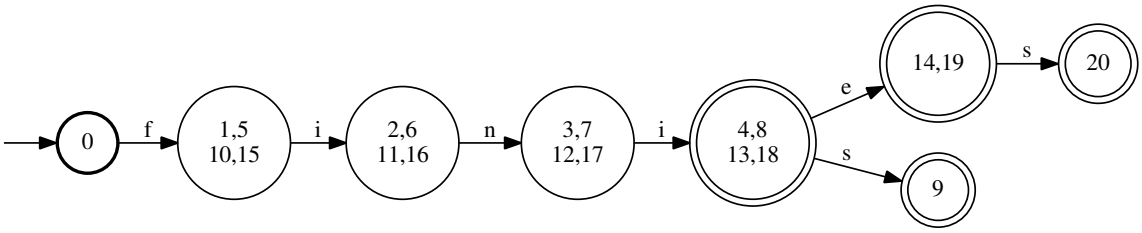
Cet algorithme est potentiellement coûteux parce que le nombre d'état croît de façon exponentielle par rapport à l'automate non déterministe. En pratique, ce coût n'est pas toujours effectif parce qu'un grand nombre d'états de 2^Q sont inaccessibles depuis l'état initial $\{i\}$ et ne

sont pas calculés si l'on s'y prend bien. Néanmoins, dans les mauvais cas, la taille du résultat est une fonction exponentielle de la taille de l'automate non-déterministe.

Prenons un exemple d'automate non déterministe qui décrit les différentes formes de l'adjectif *fini*, à savoir : *fini, finis, finie* et *finies*. Une construction naïve de l'automate consiste à écrire un chemin distinct pour chaque mot du langage.



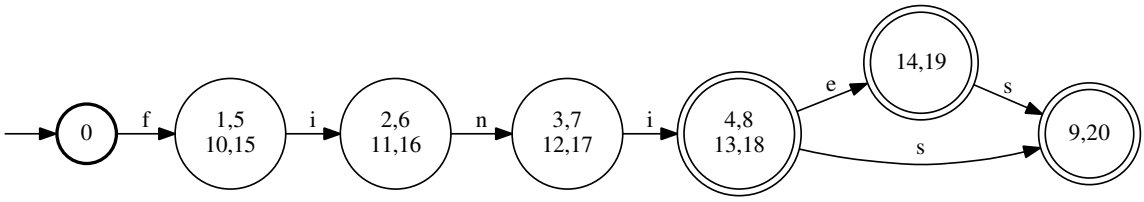
La construction de l'automate déterministe part de l'état initial, 0. Par `f`, on peut aller dans les états 1, 5, 10, et 15. On crée un nouvel état de nom `1,5,10,15`. De 1 par `i` on va en 2, de 5 en 6, de 10 en 11, de 15 en 16. On crée donc un nouvel état appelé `2,6,11,16`, avec une flèche de `1,5,10,15` vers `2,6,11,16`, étiquetée par `i`. Et ainsi de suite, ce qui donne le résultat suivant.



Propriété 14 *Il existe un algorithme appelé* algorithme de minimisation *qui pour tout automate fini* **déterministe** A *calcule un automate fini déterministe* A' *tel que* $L(A) = L(A')$ *et* A' *a un nombre d'état inférieur ou égal à tout autre automate définissant le langage* $L(A)$.

Nous ne détaillerons pas cet algorithme un peu complexe. Il est basé sur la recherche de redondance entre états, c'est à dire des états différents qui calculent le même langage. Si on en trouve, ces états redondants peuvent être fusionnés en un seul. Cet algorithme est également assez coûteux.

La minimisation de l'automate déterministe précédent permet de fusionner les états 9 et 20.



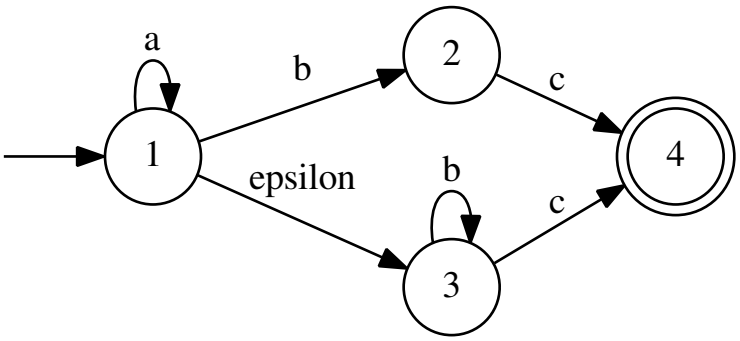
Nous allons maintenant présenter une variante des automates fini qui autorise l'utilisation de transitions sans étiquettes, qui ne lisent pas de symbole de la chaîne. On appelle ces transitions *epsilon-transitions* et on les note avec un epsilon en guise d'étiquette. Cette extension de la syntaxe des automates finis ne change rien à la puissance du formalisme. C'est à dire que les automates avec epsilon-transitions décrivent la même classe de langages que les automates sans epsilon-transition, c'est-à-dire les langages réguliers.

Définition 18 *Automate fini avec ϵ-transition*
Un automate fini est un quintuplet $A = (\Sigma, Q, \delta, i, F)$ où :

- Σ *est un ensemble fini de symboles appelé alphabet.*
- Q *est un ensemble fini dont les éléments sont appelés* états.
- δ *est une relation de $Q \times (\Sigma \cup \{\epsilon\}) \times Q$ appelée* transition *ou* ensemble des transitions *de A.*
- i *est un état de Q appelé* état initial.
- F *est un sous-ensemble de Q appelé* ensemble des états finals *de A.*

Propriété 15 *Il existe un algorithme dit* algorithme d'élimination des ϵ *qui pour tout automate A avec des ϵ-transitions calcule un automate A' sans ϵ-transition et tel que $L(A) = L(A')$.*

Le principe de l'algorithme consiste à remplacer chaque chemin de longueur 1 commençant par une epsilon-transition par une nouvelle transition qui décrit ce chemin. Prenons un exemple.



automate avec epsilon-transition

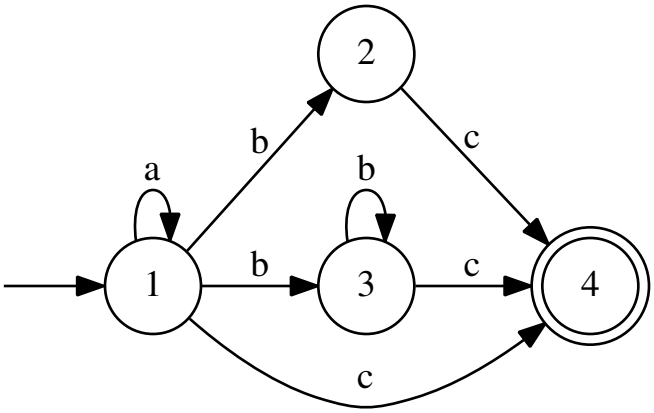
Il y a deux chemin de longueur 1 qui commencent par la transition sur epsilon :

- $(1, \epsilon, 3)(3, b, 3)$
- $(1, \epsilon, 3)(3, c, 4)$

On ajoute à l'automate deux nouvelles transitions qui résument ces deux chemins :

- $(1, b, 3)$
- $(1, c, 4)$

Et on supprime la transition $(1, \epsilon, 3)$. Il est facile de voir que pour tout succès dans l'ancien automate, il existe un succès dans le nouveau et réciproquement.



automate sans epsilon-transition

L'algorithme est un peu plus complexe dans les cas où plusieurs epsilon-transitions se suivent et si elles vont dans un état final, mais le principe général présenté ici reste valable.

L'utilisation des epsilon-transitions permet parfois de décrire plus lisiblement un langage. Elle permet aussi de simplifier la description de certaines opérations (par exemple l'union ou la concaténation). D'un autre côté, l'existence d'epsilon-transitions rend généralement l'automate non-déterministe puisqu'on a toujours la possibilité d'emprunter une telle transition même quand il existe une transition ordinaire que l'on pourrait emprunter. Par exemple, sur notre exemple, depuis la configuration initiale $(1, abc)$, on peut par un pas de calcul aller aussi bien dans la configuration $(1, bc)$ en suivant la transition sur a que dans la configuration $(3, abc)$ en suivant la transition sur epsilon.

7 Utilisation des automates finis

Les automates finis et les expressions régulières sont utilisées dans différents contextes.

— Pour décrire des traitements textuels (par exemple, recherche de sous-chaîne). Dans les traitements de textes, les langages de script (awk, perl), etc.

— Pour décrire les léxèmes d'un langage dans un compilateur ou un interpréteur (langage de programmation, de script, de description de document, d'interrogation de base de donnée). De même pour les fichiers de configuration d'applications.

— Pour décrire l'étage morpho-lexical des langues naturelles (dictionnaires, lexiques).

— Pour décrire les propriétés dynamiques d'un système. Par exemple dans des langages de description comme UML (diagrammes états-transitions) ou SA-RT.

— Pour décrire le flot de contrôle d'un programme séquentiel (par exemple, pour faire des tests d'accessibilité d'instructions).

Les propriétés des automates finis sont très intéressantes. Grâce à la déterminisation et à la minimisation, toute description sous forme d'automate fini peut être exécutée efficacement. Les opérations ensemblistes ainsi que la concaténation et l'étoile ouvrent la voie à la modularité et au traitement d'exceptions. Des boîtes à outil efficace existent qui permettent de décrire et d'exécuter de très gros automates (plusieurs millions d'états et de transitions).

D'un autre côté, les automates finis ont une puissance limitée. Ils n'ont pas la puissance des machines de Turing. La seule mémoire des automates finis est les états qui sont en nombre fini.

Parmi les langages qui ne peuvent pas être décrits par automates finis, voici quelques exemples classiques :

— les chaînes qui ont le même nombre de a que de b.

— les chaînes bien parenthésées (avec une parenthèse fermante pour chaque ouvrante et une bonne imbrication des parenthèses).

Il existe plusieurs formalismes qui augmentent le pouvoir descriptif des automates finis. Cela permet de représenter des langages non réguliers. Il y a donc surcroît de puissance. Mais la contrepartie est la perte de certaines des bonnes propriétés des automates finis. Citons quelques exemples :

— les transducteurs finis sont utilisés pour décrire des relations régulières ou des fonctions de traduction. Sur chaque transition, il y a deux symbole : un en lecture, un en écriture. Reconnaître une chaîne permet d'obtenir son image par la fonction.

— les automates à pile : une pile de taille illimitée est ajouté à un automate fini. Chaque transition lit un symbole de chaîne et réalise une opération de pile (empilement, dépilement, lecture du sommet). Ce dispositif décrit les langages non contextuels qui sont utilisés pour décrire la syntaxe des langages informatique et parfois aussi la syntaxe des langues naturelles.

— les réseaux de Petri : sur la base d'un graphe orienté, on ajoute la notion de jetons qui peuvent se déplacer dans le graphe, être créés ou éliminés. Des contraintes spécifiques permettent de limiter le nombre de jetons dans certains nœuds du graphe.

Compte tenu de leur faible pouvoir expressif, les spécifications sous forme d'automates finis sont souvent des *spécifications partielles* qui ne décrivent qu'imparfaitement, approximativement un système. Ils sont néanmoins les bienvenus dans ce rôle du fait de leur simplicité.

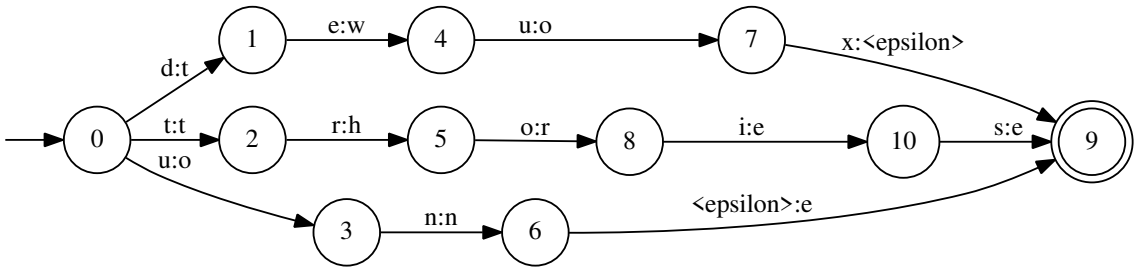
8 Transducteurs finis à états

8.1 Notion et définition de transducteur fini à états

Selon le dictionnaire Trésor de la Langue Française informatisé, un transducteur est un dispositif ou élément d'une chaîne de communication (mécanique, électrique, etc.) recevant un message sous une certaine forme et le transformant en une autre. Selon le Larousse, transducteur est un synonyme de traducteur.

Un transducteur fini à état est un automate fini qui non seulement reconnaît un ensemble régulier de chaînes, mais encore qui traduit chaque chaîne de cet ensemble dans une autre chaîne appartenant à un autre langage régulier. Concrètement, comme un automate fini, c'est un graphe orienté étiqueté, mais chaque transition est étiquetée par un (ou zéro) symbole à reconnaître et un (ou zéro) symbole utilisé pour construire la traduction, séparés par un point-virgule.

Voyons un exemple qui concerne la traduction de quelques chiffres du français à l'anglais (un→one, deux → two et trois→three).



exemple de transducteur

Définition 19 *Transducteur fini à état*
Un automate fini est un sextuplet $A = (\Sigma_i, \Sigma_o, Q, \delta, i, F)$ *où :*

- Σ_i *et* Σ_o *sont deux ensembles finis de symboles appelés respectivement alphabet d'entrée et alphabet de sortie.*
- Q *est un ensemble fini dont les éléments sont appelés* états.
- δ *est une relation de* $Q \times (\Sigma_i \cup \{\epsilon\}) \times (\Sigma_o \cup \{\epsilon\}) \times Q$ *appelée* transition *ou* ensemble des transitions *de* A.
- i *est un état de* Q *appelé* état initial.
- F *est un sous-ensemble de* Q *appelé* ensemble des états finals *de A.*

La notion de chemin succès est inchangée par rapport aux automates : c'est un chemin qui part de l'état initial et arrive dans un état final. Mais un tel chemin n'est plus associé à une chaîne mais à une paire de chaînes : une chaîne de symboles de Σ_i (ou entrée) et une de symboles de Σ_o.

Un transducteur ne défini pas un langage (ensemble de chaînes), mais une relation binaire (ensembles de paires de chaînes). Une relation définissable par un transducteur fini est appelée *relation régulière*.

Bien que la terminologie rappelle un peu celle des fonctions, il s'agit bien de relations, ce qui signifie qu'une chaîne peut avoir plusieurs traductions et deux chaînes différentes peuvent avoir la même traduction. Par exemple, dans une traduction français-anglais, on peut avoir la relation suivante : $\{$`(un,a)`, `(un,one)`, `(gratuit,free)`, `(libre,free)`$\}$.

Un transducteur peut donner lieu à différent types d'exécution :

- étant donné une chaîne entrée, calculer sa ou ses traductions en sortie (si cette chaîne appartient à une ou plusieurs paires de la relation).
- étant donné une chaîne en sortie, calculer la ou les entrées dont cette sortie est la traduction.
- étant donné une paire de chaînes, déterminer si cette paire appartient à la relation.
- exhiber une ou plusieurs paires de la relation.

Nous allons décrire le calcul pour le premier type d'exécution, que nous appellerons *exécution canonique* du transducteur.

Une configuration est un triplet avec un chemin du transducteur, une chaîne de Σ_i^* et une chaîne de Σ_o^*. Soit $T = (\Sigma_i, \Sigma_o, Q, \delta, i, F)$ un transducteur et w_i la chaîne de Σ_i^* à traduire. La configuration initiale est (i, w_i, ϵ). Un pas de calcul permet de passer d'une configuration à une

autre en utilisant une transition $t = (q_i, x_i, x_o, q_f)$. $(q_1 \dots q_i, x_i v_i, v_o) \vdash^t (q_1 \dots q_i \rightarrow, v_i, v_o x_o)$
Une configuration $(q_0 \rightarrow \dots q_n, w_i, w_o)$ est finale si et seulement si q_n est final et $w_i = \epsilon$.

8.2 Opérations rationnelles et ensemblistes

Certaines opérations sur les langages réguliers et les automates finis se définissent de façon analogue sur les relations régulières et transducteurs finis. Pour ces opérations, les algorithmes sont les mêmes pour automates et transducteurs.

Il s'agit des trois opérations rationnelles : concaténation, union et clôture sous concaténation.

En revanche, les relations régulières ne sont pas closes sous certaines opérations ensemblistes et opérations d'optimisation. Par exemple, l'intersection de deux relations régulières n'est pas toujours une relation régulière. En termes de transducteurs, l'intersection de deux transducteurs finis n'est pas toujours calculable sous la forme d'un transducteur fini. De même pour le complémentation et la différence ensembliste.

Les opérations de déterminisation et de minimisation ne s'appliquent pas non plus aux transducteurs finis en général. Elles peuvent s'appliquer à certaines conditions. Il existe également des algorithmes d'optimisation approchée qui ne garantissent pas un résultat optimal.

L'élimination des ϵ transitions permet d'éliminer les transitions qui sont étiquetées par un ϵ en entrée et un ϵ en sortie. Il n'y a pas d'algorithme général pour éliminer les transitions avec un ϵ d'un seul des deux côtés (entrée ou sortie).

8.3 Opérations spécifiques des transducteurs

La première opération consiste à restreindre une relation régulière aux entrées appartenant à un certain langage régulier. Soit R une relation régulière et L un langage régulier. Cette restriction notée $\sigma_L(R)$ est définie par $\sigma_L(R) = \{(w_i, w_o) \in R | w_i \in L\}$. La restriction d'une relation régulière à un langage régulier est une relation régulière.

Etant donné un transducteur fini T et un automate fini A, il existe un algorithme permettant de construire un transducteur T' qui définit $\sigma_{L(A)}(R(T))$. Cet algorithme est une variante de l'algorithme d'intersection d'automates finis.

La seconde opération est la composition qui consiste à enchaîner deux relations, la sortie de la première étant utilisée comme entrée de la seconde. $R_1 \circ R_2 = \{(w_1, w3) | \exists w_2, (w_1, w_2) \in R_1, (w_2, w_3) \in R_3\}$ La composition de deux relations régulières est une relation régulière. Il existe un algorithme qui construit la composition de deux transducteurs finis sous la forme d'un transducteur fini. C'est une variante de l'algorithme d'intersection d'automates finis.

La troisième opération est la projection qui consiste à extraire un langage d'une relation en ne conservant que le premier ou que le second composant de chaque paire de la relation. C'est donc en fait deux opérations différentes notées π_1 et π_2, définies par $\pi_1(R) = \{w_i \in \Sigma_i^* | \exists w_o \in \Sigma_o, (w_i, w_o) \in R\}$ et $\pi_2(R) = \{w_o \in \Sigma_o^* | \exists w_i \in \Sigma_i, (w_i, w_o) \in R\}$. La projection d'une relation régulière est un langage régulier. L'algorithme sur les transducteurs est trivial : il suffit de remplacer sur chaque transition la paire par la composante conservée.

L'exécution canonique d'un transducteur fini à toutes les chaînes d'un langage régulier peut se calculer avec une restriction suivie d'une projection sur la sortie. Le résultat est un langage

régulier.

Une autre opération consiste à inverser l'entrée et la sortie du transducteur.

8.4 Extension des expressions régulières

Si l'on veut avoir pour les relations régulières une notation équivalente à celle des expressions régulières pour les langages réguliers, il faut ajouter un nouvel opérateur pour définir les relations et les opérations spécifiques aux transducteurs.

L'opérateur permettant de définir des relations régulières est le produit cartésien de langage régulier. Ce produit cartésien se définit comme suit : $L_1 \times L_2 = \{(w_i, w_o) | w_i \in L_1, w_o \in L_o\}$. Le produit cartésien est noté : (deux points) dans les expressions régulières.

Prenons quelques exemples : `"ab":"x"` est l'expression régulière qui note la relation { `(ab,x)` }. `("ab"|"ac"):("x"|"y")` est l'expression régulière qui note la relation { `(ab,x),(ab,y),(ac,x),(ac,y)` }. `"a"*:"x"` note la relation $\{(\epsilon, x), (a, x), (aa, x), (aaa, x), \ldots\}$.

Avec ce nouvel opérateur, il faut faire attention lorsqu'on écrit une expression : toutes les expressions n'ont pas de sens. Par exemple, le produit cartésien doit porter sur deux expressions qui dénotent des langages et non des relations. Par exemple, il n'est pas correct d'écrire : `("a":"x"):"t"`. Les opérations comme la disjonction ou la concaténation peuvent s'appliquer sur deux langages ou sur deux relations, mais pas sur un langage et une relation. Il n'est pas correct d'écrire `"ab"("c":"d")`.

Pour les opérations spécifiques aux transducteurs, on a la sélection que l'on notera #, la composition que l'on notera @ et les deux projections que l'on notera respectivement /1 et /2.

8.5 Conclusion sur les transducteurs

Les trasducteurs offrent un modèle plus puissant que les automates finis, mais ce surcroît de puissance se fait au prix de la perte de certaines propriétés des automates finis : la clôture sous intersection et différence, les propriétés d'optimisation automatique (déterminisation, minimisation).

Les transducteurs sont souvent utilisés pour définir un calcul comme une succession d'étapes élémentaires, chaque étape étant réalisée par l'application d'un transducteur fini. On appelle cela une *cascade de transducteurs*. Avec une telle cascade, il est équivalent de faire l'exécution canonique des transducteurs successifs ou de calculer la composition des transducteurs et de faire l'exécution canonique du résultat.

Supposons qu'on ait une cascade de 3 transducteurs et une entrée w à transformer. L'équivalence des deux calculs s'écrit :
$(((w \# t_1)/2 \# t_2)/2 \# t_3) \equiv (w \# (t_1 @ t_2 @ t_3))/2$