Chapitre 1

Langages rationnels

Master de Linguistique Informatique
notes de cours

Pascal Amsili

1er février 2016
v. 1.11

Table des matières

1 Langages rationnels **0**
1.1 Introduction : définition 1
1.2 Expressions rationnelles 2
1.2.1 Définition 2
1.2.2 Equivalence et réductions . . . 2
1.2.3 Extensions notationnelles . . . 3
1.3 Automates 3
1.3.1 Définition 3
1.3.2 Transformations 6
1.3.3 Propriétés de fermeture 14
1.4 Théorèmes d'équivalence 16
1.4.1 Le théorème triangulaire 16
1.4.2 Grammaires et automates . . . 17
1.4.3 Automates et Exp. Rat. 18
1.5 Propriétés des langages rationnels . . . 22
1.5.1 Le lemme de pompage 22
1.5.2 Conséquences 23

Ce chapitre présente des algorithmes, avec le choix pédagogique suivant : les algorithmes seront illustrés sur des exemples (et mis en œuvre pendant les TD), mais ils ne seront pas, en général, donnés sous forme de code (que ce soit en pseudo-langage plus ou moins abstrait ou dans un langage spécifique). Ces exercice est laissé aux étudiants, qui doivent non seulement être capable de « dérouler » les algorithmes sur des exemples spécifiques, mais qui sont fortement invités à formuler ces algorithmes en pseudo-langage, et à les implémenter pour eux-même, ce qui est le seul moyen d'aboutir à une réelle compréhension de ces algorithmes.

1.1 Introduction : définition

Rappelons qu'on travaille en théorie des langages formels sous l'hypothèse qu'on dispose d'un *alphabet*, c'est-à-dire un ensemble fini de symboles (noté X ou Σ), sur lequel on définit une opération, la *concaténation* (notée .), associative, non symétrique, et admettant un élément neutre ε (le mot-vide). On définit un *mot* comme une concaténation finie de symboles de X, et on étend l'opération de concaténation de façon naturelle à tous les mots possibles sur X, ce qui permet de définir le *monoïde libre* $(X^*, .)$.
Par définition, un *langage formel* sur un alphabet X est un ensemble de *mots*, c'est-à-dire un sous-ensemble de X^*.
Un langage rationnel est un langage obtenu en prenant comme *base* les langages singletons formés sur un alphabet, et comme *opérations* les opérations d'union, de produit et d'étoile (les « opérations rationnelles »).
Commençons par rappeler la définition de ces opérations rationnelles :

Déf. 1 (Opérations rationnelles)

On peut définir deux opérations binaires et une opération unaire sur les langages :
- L'*union* des langages est définie comme d'habitude (union ensembliste)
- Le *produit* des langages est défini de la manière suivante (on suppose l'opération de concaténation définie) : $L_1.L_2 = \{uv \,/\, u \in L_1 \text{ et } v \in L_2\}$
  En généralisant, on peut proposer la notation

$$\begin{array}{lcl} A^0 & = & \{\varepsilon\} \\ A^1 & = & A \\ A^{i+1} & = & A.A^i \\ \vdots & & \\ A^n & = & \{a_1...a_n \,/\, a_i \in A\} \end{array}$$

- L'*étoile* (ou fermeture) d'un langage est définie ainsi : $A^* = \bigcup_{n \geqslant 0} A^n$

Déf. 2 (Langage rationnel)

Un langage rationnel sur un alphabet X est un sous-ensemble de X^* défini inductivement de la façon suivante :
- $\emptyset$ et $\{\varepsilon\}$ sont des langages rationnels ;
- pour tout $a \in X$, le singleton $\{a\}$ est un langage rationnel ;
- pour tous g et h rationnels, les ensembles $g \cup h$, $g.h$ et g^* sont des langages rationnels.

Remarques Tous les langages finis sont rationnels ; X^* est rationnel.

Caractérisation Quels sont les moyens de caractériser les langages rationnels ?
- les expressions rationnelles (section 1.2) ;
- les automates (section 1.3) ;
- les grammaires régulières (section 1.4.2).

Le programme dans la suite de ce chapitre est l'étude des différents moyens de caractériser, et surtout de reconnaître, les (classes de) langages rationnels, avec un accent mis sur les automates, de loin le meilleur outil. On étudiera aussi les relations entre ces différents formalismes, et les algorithmes de transformations qui portent sur eux.

1.2 Expressions rationnelles

Les expressions rationnelles (ou « régulières ») ont largement été évoquées dans le cadre du cours d'introduction au TAL, on donne simplement dans cette section quelques éléments de rappel, qui sont complétés par un extrait de [Yvon et Demaille, 2005] : voir annexes pp. 25 à 30.

1.2.1 Définition

On commence par donner une définition syntaxique des expressions rationnelles.

Déf. 3 (Expression rationnelle – syntaxe)

Soit X un alphabet. On définit les expressions rationnelles récursivement de la façon suivante :
- Pour tout $x \in X$, x est une expression rationnelle
- ε est une expression rationnelle
- Si φ et ψ sont des expressions rationnelles, alors
  - $(\varphi|\psi)$,
  - $(\varphi.\psi)$,
  - et φ^* sont des expressions rationnelles.

Une fois donnée la syntaxe, il faut donner la sémantique, qui est assez transparente : on interprète le symbole | comme l'union, le symbole . comme le produit des langages, et le symbole * comme l'étoile de Kleene.

Déf. 4 (Expression rationnelle – sémantique)

- Pour tout $x \in X$, l'e.r. x dénote le langage $\{x\}$,
- L'e.r. ε dénote le langage $\{\varepsilon\}$,
- Si φ et ψ sont des expressions rationnelles, alors
  - l'e.r. $(\varphi|\psi)$ dénote l'union des langages dénotés par φ et ψ ;
  - l'e.r. $(\varphi.\psi)$ dénote le produit des langages dénotés par φ et ψ ;
  - et l'e.r. φ^* dénote l'étoile du langage dénoté par φ.

1.2.2 Equivalence et réductions

Voir la section 2.1.3 « équivalence et réductions » de l'extrait de [Yvon et Demaille, 2005], à la page 27.

1.2.3 Extensions notationnelles

Les expressions rationnelles sont directement utilisées dans certains programmes, soit de bas niveau (`grep` sous Unix, ou `emacs`), soit dans les fonctions de recherche d'applications (d'un concordancier, par exemple...).
En interne, ces expressions sont le plus souvent traduites en automates, car c'est un bon moyen de disposer d'un algorithme de recherche efficace.
Pour des raisons évidentes de facilité d'utilisation, les langages fournis sont plus riches que ce qu'on a vu (+ en plus de l'étoile, listes alphabétiques, complémentaires...), mais il est important de noter que ces facilités notationnelles n'augmentent pas le pouvoir expressif des expressions, qui continuent à reconnaître des langages rationnels.
Voir la section 2.2 « extensions notationnelles » de l'extrait de [Yvon et Demaille, 2005], à la page 28.

1.3 Automates

1.3.1 Définition

Les automates finis sont un cas particulier des machines à nombre finis d'états, dont la machine de Turing représente la version la plus sophistiquée. On peut les caractériser de façon purement mathématique, comme dans la définition donnée plus loin, mais on peut aussi donner la métaphore courante : on dispose d'une bande de lecture, composée de cases, et d'un alphabet de symboles qui peuvent chacun occuper exactement une case. On suppose que la bande est infinie à droite, et qu'une "tête de lecture" peut lire les symboles sur la bande, en se déplaçant à chaque lecture d'une case vers la droite. On suppose de plus un "organe de contrôle", qui est susceptible d'être dans un nombre fini d'états (on peut imaginer des lampes différentes selon l'état, par exemple). Voir la figure 1.1.

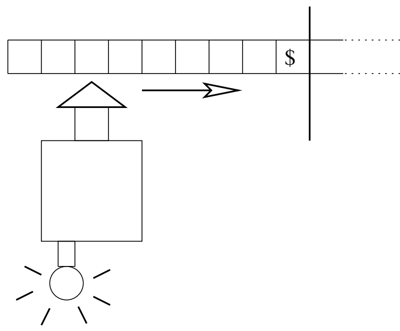


FIGURE 1.1 – Représentation graphique AFD

Alors le mécanisme de reconnaissance d'un mot peut être décrit de la façon suivante : au départ, la tête se trouve sur le symbole le plus à gauche, dans l'état initial. À chaque top d'horloge, la machine avance d'une case, et lit le symbole correspondant, ce qui (peut) provoque(r) un changement d'état. La machine s'arrête soit à la fin du mot, soit pour une autre raison, et on décide que le mot est reconnu si la machine l'a lu en entier, et qu'elle se trouve dans un état dit final.
Un automate est défini par les différents états qu'il peut prendre, et surtout par son comportement en fonction des symboles sur la bande de lecture : les différentes situations possibles

se distinguent d'une part par le symbole sous la tête de lecture, et d'autre part par l'état courant. Le comportement d'un automate donné est défini par la liste des changements d'état provoqués par chaque sitation.
Un automate déterministe est un automate pour lequel la présence d'un symbole sous la tête de lecture, dans un état particulier de l'organe de contrôle, conduit à un seul autre état possible.

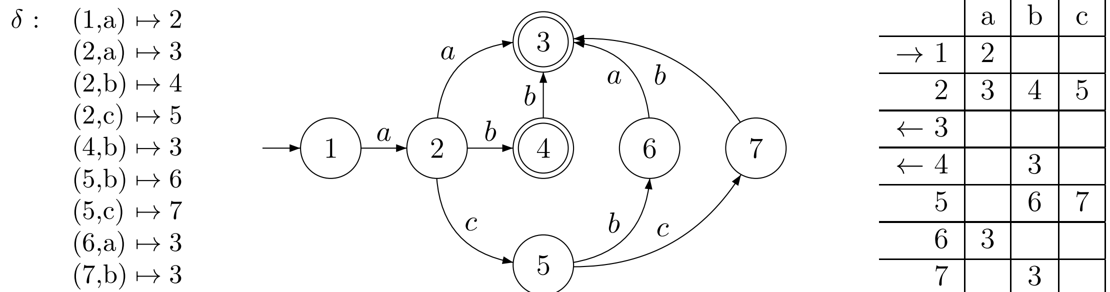
Déf. 5 (Automate fini déterministe - AFD)

Un automate à nombre fini d'états (automate fini) déterministe $\mathcal{A}$ est défini par :

$$\mathcal{A} = \langle Q, \Sigma, q_0, F, \delta \rangle$$

Q est un ensemble fini d'états
Σ est un vocabulaire (ou alphabet)
q_0 est un élément de Q, appelé état initial
F est un sous-ensemble de Q, dont les éléments sont appelés états terminaux
δ est une **fonction** de $Q \times \Sigma$ dans Q. On écrit $\delta(q, a) = r$.

Exemple Soit le langage (fini) $\{aa, ab, abb, acba, accb\}$.
On peut définir l'automate suivant, qui reconnaît ce langage : $\langle Q, \Sigma, q_0, F, \delta \rangle$,
avec $Q = \{1, 2, 3, 4, 5, 6, 7\}$, $\Sigma = \{a, b, c\}$, $q_0 = 1$, $F = \{3, 4\}$, et δ est définie ainsi :



δ : (1,a) $\mapsto$ 2
(2,a) $\mapsto$ 3
(2,b) $\mapsto$ 4
(2,c) $\mapsto$ 5
(4,b) $\mapsto$ 3
(5,b) $\mapsto$ 6
(5,c) $\mapsto$ 7
(6,a) $\mapsto$ 3
(7,b) $\mapsto$ 3

	a	b	c
→ 1	2		
2	3	4	5
← 3			
← 4		3	
5		6	7
6	3		
7		3	

Les deux représentations données ci-dessus, la première sous forme de graphe sagittal, la seconde sous forme de « table de transition », permettent chacune, avec des conventions simples, de définir entièrement un automate (on peut lire sur ces représentations tous les éléments pertinents : l'ensemble d'états Q, l'alphabet Σ, l'état initial q_0, et surtout la fonction de transition *delta*).

La reconnaissance est définie comme l'**existence** d'une suite d'états définie de la manière suivante. On appelle une telle suite un **chemin** dans l'automate.

Déf. 6 (Reconnaissance)

Un mot $a_1a_2...a_n$ est **reconnu** par l'automate si et seulement si il existe une suite $k_0, k_1, ..., k_n$ d'éléments de Q (ensemble d'états) telle que

$$k_0 = q_0$$
$$k_n \in F$$
$$\forall i \in [1, n], \quad \delta(k_{i-1}, a_i) = k_i$$

Informellement, on peut distinguer les différentes situations pour la reconnaisance (ou la non reconnaissance) d'un mot (ici des exemples avec l'automate donné plus haut) :

— *input* consommé, sur état terminal : SUCCÈS ($acba$)
— *input* consommé, sur un état non terminal : ÉCHEC (acb)
— *input* non consommé, pas de transition : ÉCHEC ($ab|a$, $acb|c$) (à noter que dans ce dernier cas, on peut être sur un état terminal, mais on échoue parce qu'il reste de l'*input* ($ab|a$), ou bien on peut être sur un état non terminal ($acb|c$)).

La définition précédente d'un automate laissait ouverte la possibilité que pour un symbole donné et un état donné, la transition ne soit pas définie. On peut exiger que cette situation soit exclue, c'est-à-dire que le comportement de l'automate soit défini pour toute paire (symbole, état courant). Alors on dira que l'automate est *complet*. Un automate complet est un automate dont la table de transition ne comporte pas de cases vides.

Déf. 7 (AFD complet)

Un automate à nombre fini d'états déterministe complet $\mathcal{A}$ est défini par :

$$\mathcal{A} = \langle Q, \Sigma, q_0, F, \delta \rangle$$

Q est un ensemble fini d'états
Σ est un vocabulaire (ou alphabet)
q_0 est un élément de Q, appelé état initial
F est un sous-ensemble de Q, dont les éléments sont appelés états terminaux
δ est une **fonction** de $Q \times \Sigma$ dans Q, qui vérifie la propriété :

$$\forall (q, a) \in Q \times \Sigma,\ \exists q' \in Q \text{ t.q. } \delta(q, a) = q'$$

La définition de base ne prévoit pas qu'un automate soit déterministe, c'est-à-dire qu'il n'y ait qu'un seul état possible étant donnés un symbole et un état. En effet, la reconnaissance d'un mot u par un automate est définie par l'**existence** d'un chemin dans l'automate pour u : il est donc possible en théorie de devoir tenter plusieurs chemins avant de trouver celui qui reconnaît le mot. En pratique, on préfère utiliser des automates déterministes, qui se caractérisent par le fait que dans une configuration (état, symbole) donnée, il n'y a qu'une seule possibilité. Autrement dit, la table de transition contient **au plus** un état par case (il peut n'y en avoir aucun si l'automate n'est pas complet).

Déf. 8 (Automate fini non déterministe - AFnD)

Un automate à nombre fini d'états (automate fini) non déterministe $\mathcal{A}$ est défini par :

$$\mathcal{A} = \langle Q, \Sigma, q_0, F, \delta \rangle$$

Q est un ensemble fini d'états
Σ est un vocabulaire (ou alphabet)
q_0 est un élément de Q, appelé état initial
F est un sous-ensemble de Q, dont les éléments sont appelés états terminaux
δ est une **fonction** de $Q \times \Sigma \cup \{\varepsilon\}$ dans 2^Q.

Sources de non déterminisme Les automates non déterministes ainsi définis comportent deux sources distinctes de non déterminisme : d'une part le fait qu'une combinaison (q, x) puisse donner lieu à plusieurs changements d'état, et d'autre part le fait que l'on introduit des transitions d'un type nouveau, étiquetées par ε. On parle de *transitions spontanées*, ou d'ε-transition : c'est une transition qui permet de changer d'état sans déplacer la tête de lecture.

Ainsi, par exemple, l'automate représenté ci-après est non déterministe parce que dans l'état 3, à la lecture du symbole b, deux options sont possibles ; mais il est aussi non déterministe parce que dans l'état 1, à la lecture du symbole a, deux options sont possibles : passer à l'état 2, ou passer par une ε-transition à l'état 4 et passer ensuite à l'état 3.

	a	b	ε
$\rightarrow 1$	2	3	4
2		4	
3		3, 1	
$\leftarrow 4$	3		3

Algorithme de reconnaissance L'algorithme de reconnaissance esquissé plus haut pour un automate déterministe est considérablement compliqué dans le cas non déterministe : sans entrer dans les détails, il faut que l'algorithme soit capable de considérer tous les chemins possibles avant de prendre sa décision. Il faut donc que l'algorithme soit capable de mémoriser chaque point de choix, et de revenir par retour-arrière (*backtrack*) explorer les autres solutions quand une option a abouti à un échec.

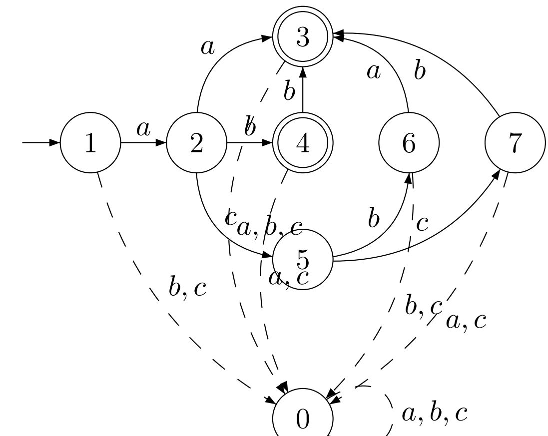
1.3.2 Transformations

On présente dans cette section différents algorithmes qui visent à transformer des automates, avec comme principe (souvent implicite) que la transformation doit conserver le langage reconnu.

1.3.2.1 Complétion

Comment rendre un automate complet ? Facile : on rajoute un « puits », ou état mort. Il suffit de boucher les trous de la table, et de rajouter des boucles sur le puits.

	a	b	c
$\rightarrow 1$	2	0	0
2	3	4	5
$\leftarrow 3$	0	0	0
$\leftarrow 4$	0	3	0
5	0	6	7
6	3	0	0
7	0	3	0
0	0	0	0



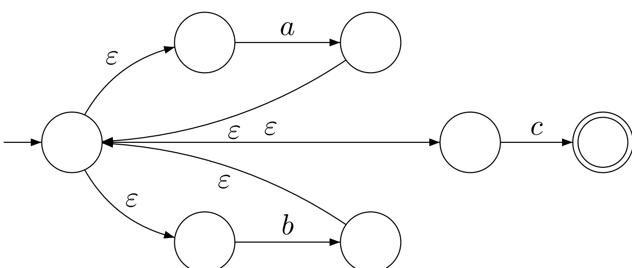
Synthèse: algorithme de complétion

Entrée	Automate quelconque reconnaissant $\mathcal{L}$
Sortie	Automate complet reconnaissant $\mathcal{L}$
Méthode	Si la table de transition contient des cases vides, créer un nouvel état « puits » qui boucle sur lui-même, et tel que toutes les transitions non définies pointent vers lui.
Remarque	Il n'est pas toujours nécessaire de créer un état puits.

1.3.2.2 Elimination des ε-transitions

On a vu plus haut la définition des automates comportant des ε-transitions. On a vu qu'ils étaient source de non déterminisme, mais on dispose du théorème selon lequel *Tout langage reconnu par un automate comportant des ε-transitions est reconnu par un automate sans ε-transition.* Ce théorème peut être démontré en exhibant un algorithme qui transforme, sans changer le langage reconnu, un automate avec ε-transitions en un automate sans ε-transitions.
Illustrons cet algorithme avec l'automate illustré à la figure 1.2, qui reconnaît le langage $(a|b)^*c$. Cet automate ressemble à ce qui serait produit en appliquant de façon stricte l'algorithme de conversion d'une expression rationnelle en automate (cf. section 1.4.3.1).

FIGURE 1.2 – ε-automate reconnaissant $(a|b)^*c$



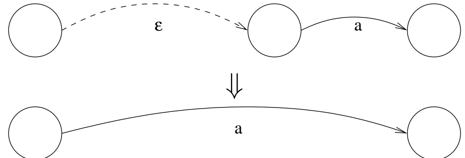
L'algorithme est basé sur l'application ε^+ définie de la manière suivante :

1. Si $q_j \in \delta(q_i, \varepsilon)$ alors $q_j \in \varepsilon^+(q_i)$
2. Si $q_j \in \varepsilon^+(q_i)$ et $q_k \in \delta(q_j, \varepsilon)$ alors $q_k \in \varepsilon^+(q_i)$

On construit l'automate non déterministe $\langle X, Q, I, F', \delta' \rangle$ comme suit :

```
Début
    F' := F
    pour tous les qi ∈ Q faire
        pour tous les x ∈ X faire
            δ'(qi, x) := δ(qi, x)
    pour tous les qi tels que ε+(qi) ≠ ∅ faire
        pour tous les qj ∈ ε+(qi) faire
            pour tous les x et qr tels que qr ∈ δ(qj, x) faire
                δ'(qi, x) := δ'(qi, x) ∪ {qr}
            si qj ∈ F alors F' := F' ∪ {qi}
Fin
```

Idée de l'algorithme : d'une part, toutes les transitions alphabétiques (*i.e.* celles qui ne sont pas des ε-transitions) sont conservées dans le nouvel automate ; d'autre part, pour chaque état, on détermine tous les états qui peuvent être atteints « spontanément », et on court-circuite les transitions alphabétiques auquelles ces transitions spontannées donnent accès. Schématiquement, on peut représenter le processus, pour chaque état atteint par une chaîne d'ε-transitions, pour chaque lettre, de la manière suivante :



Un autre façon de caractériser l'algorithme est la suivante : Dans le nouvel automate, il existe une transition de l'état i vers l'état j dont l'étiquette contient le symbole x s'il existe un certain état k tel que

1. L'état k est accessible à partir de l'état i en suivant un chemin de zéro ε-transitions ou plus. On notera que $k = i$ est toujours autorisé.
2. Il existe, dans l'ancien automate, une transition de l'état k vers l'état j, étiquetée par x.

Il reste ensuite à modifier éventuellement les états d'acceptation.

Illustration de l'algorithme appliquée à l'exemple de la figure 1.2 : la table de transition initiale, l'application ε^+, ainsi que la table résultant du calcul, sont données à la figure 1.3, l'automate résultant étant donné à la figure 1.4.

FIGURE 1.3 – Tables illustrant l'élimination des ε-transitions

Table de transition initiale :

δ	ε	a	b	c
→1	2,6,4	0	0	0
2	0	3	0	0
3	1	0	0	0
4	0	0	5	0
5	1	0	0	0
6	0	0	0	7
←7	0	0	0	0

$\varepsilon+$

1	$\longrightarrow$	2 4 6
2	$\longrightarrow$	$\emptyset$
3	$\longrightarrow$	1 2 4 6
4	$\longrightarrow$	$\emptyset$
5	$\longrightarrow$	1 2 4 6
6	$\longrightarrow$	$\emptyset$
7	$\longrightarrow$	$\emptyset$

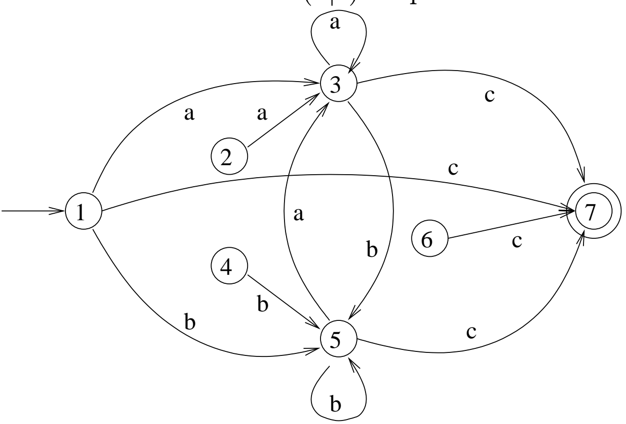
Table après calcul :

δ'	a	b	c
→1	3	5	7
2	3	0	0
3	3	5	7
4	0	5	0
5	3	5	7
6	0	0	7
←7	0	0	0

Synthèse: algorithme d'élimination des ε-transitions

Entrée	Automate avec ε-transitions reconnaissant $\mathcal{L}$
Sortie	Automate **sans ε-transitions** reconnaissant $\mathcal{L}$
Méthode	Créer un nouvel automate avec toutes les transitions alphabétiques initiales, et ajouter la transition $i \xrightarrow{x} j$ chaque fois que l'on a à la fois $i \xrightarrow{\varepsilon^*} k$ et $k \xrightarrow{x} j$.
Remarque	Cet algorithme augmente en général le non déterminisme lié à la multiplicité des transitions.

FIGURE 1.4 – Automate reconnaissant $(a|b)^*c$ après élimination des ε-transitions



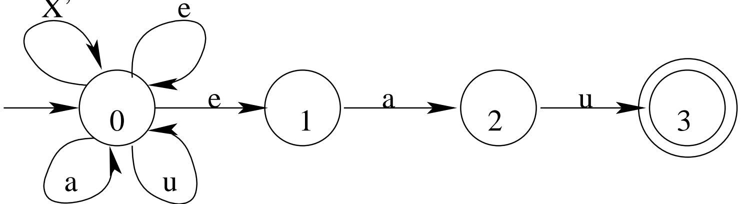
1.3.2.3 Déterminisation

Les automates déterministes sont bien plus faciles à manipuler au point de vue algorithmique, mais il est quelquefois nettement plus facile de concevoir un automate non déterministe que directement la version déterministe.

Heureusement, nous disposons d'un théorème : *pour tout langage reconnu par un automate non déterministe, il existe un automate déterministe qui reconnaît ce langage*, et la démonstration de ce théorème repose sur l'algorithme de déterminisation illustré ci-après.

Pour illustrer l'algorithme, partons d'un cas typique de ces situations où il est nettement plus facile de concevoir un automate non déterministe : soit le langage L_e de tous les mots sur l'alphabet $X = \{a, b, c, d, e, ...z\}$ qui se terminent par *eau*. L'automate illustré à la figure 1.5[1] reconnaît ce langage, mais il est non déterministe : dans l'état initial, il y a deux transitions étiquetées par la lettre e.

FIGURE 1.5 – Automate non déterministe pour le langage L_e



La déterminisation consiste à créer un nouvel automate, dont les états sont formés par des ensembles d'états de l'automate initial. La construction de ce nouvel automate se fait en partant de l'état initial (ou de l'ensemble des états initiaux s'il y en a plusieurs) et en créant les états nécessaires en fonction des transitions de l'automate initial. Dans l'exemple, cela donne le nouvel automate représenté à la figure 1.6.

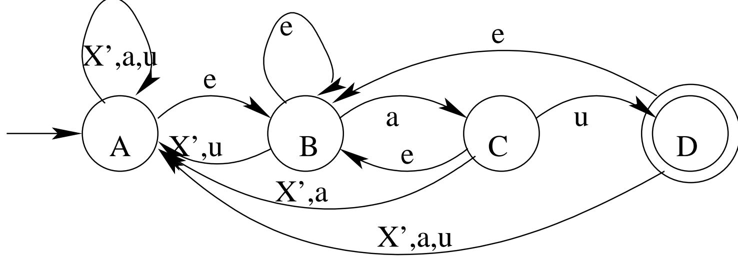
En exercice, faire la même chose à partir de l'automate complété. Conclusion : pour cet algorithme, ce n'est pas une bonne idée de compléter d'abord l'automate, il est plus simple de compéter le résultat.

1. Par convention, on note X' tous les symboles de l'alphabet X qui ne sont pas explicitement figurés dans l'automate : pour toutes les lettres de X', le comportement de l'automate est le même en toute circonstance.

FIGURE 1.6 – Automate déterministe pour le langage L_e

	X'	e	a	u	
$\{0\}$	$\{0\}$	$\{0,1\}$	$\{0\}$	$\{0\}$	A
$\{0,1\}$	$\{0\}$	$\{0,1\}$	$\{0,2\}$	$\{0\}$	B
$\{0,2\}$	$\{0\}$	$\{0,1\}$	$\{0\}$	$\{0,3\}$	C
$\{0,3\}$	$\{0\}$	$\{0,1\}$	$\{0\}$	$\{0\}$	D

$X' = X \setminus \{e, a, u\}$



Synthèse: algorithme de déterminisation

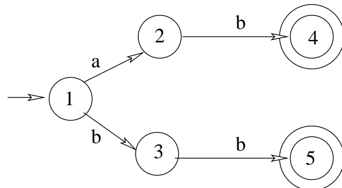
Entrée	Automate sans ε-transition reconnaissant $\mathcal{L}$
Sortie	Automate **déterministe** reconnaissant $\mathcal{L}$
Méthode	Créer un nouvel automate dont les états sont formés d'ensembles d'états, en partant de l'état initial, et en créant les états nouveaux selon les besoins.
Remarques	— Il n'est pas nécessaire que l'automate de départ soit complet. — Il n'est pas interdit de partir d'un automate qui comprend des ε-transitions, mais ce n'est pas une bonne idée, car la suppression des ε-transitions conduit en général au non déterministe qu'on veut éliminer ici. — Si l'automate initial comprend des états inaccessibles (v. plus loin), ces états sont supprimés dans l'algorithme.

Suppression du non déterminisme Pour garantir l'absence de non déterminisme d'un automate, on doit commencer par éliminer les ε-transitions, ce qui peut se faire à partir d'un automate quelconque (pas nécessairement complet) ; ensuite, on applique l'algorithme de déterminisation.

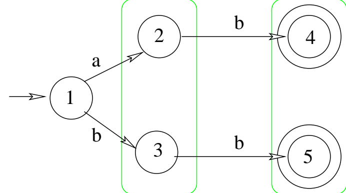
1.3.2.4 Minimisation

Pour les automates finis déterministes complets (et seulement ceux-là), il existe un algorithme de minimisation, permettant de construire à partir de n'importe quel AFDC un AFDC qui reconnaît le même langage et qui comprend un nombre inférieur ou égal d'états. (Théorème de Myhill & Nerode.)

Principe de l'algorithme La base de l'algorithme est la notion d'équivalence entre états. Si l'automate n'est pas minimal, c'est qu'il existe deux états à partir desquels on reconnaît le même langage (*i.e.* le même suffixe). On peut se faire un idée sur l'exemple (simplissime) suivant :



D'une part les deux états terminaux ne se distinguent pas puisque rien ne part d'eux (ou plutôt tout va vers le puits) ; d'autre part les deux états (2) et (3) ne se distinguent pas non plus : ils caractérisent une situation où il faut un b pour passer à un état final. Donc cet automate reconnaît le même langage qu'un automate qui n'aurait que 3 états, l'un correspondant à (2) ou (3), l'autre correspondant à (4) ou (5).



Pour minimiser un automate, il faut donc trouver toutes les classes d'états équivalents, et les fusionner en un seul état. Comment trouver toutes ces classes ?

Classes d'équivalence Pour la mise en œuvre d'un tel algorithme, il faut résoudre deux problèmes distincts : d'une part générer les partitions sur l'ensemble d'états, sachant que le nombre de partitions à considérer est très elevé (nombre de Bell) ; d'autre part trouver une méthode pour décider que deux états sont équivalents.

L'algorithme proposé ci après est un algorithme de point fixe, qui consiste non pas à tenter de regrouper les états équivalents, mais au contraire à séparer les états non équivalents.

On commence par définir la « non-équivalence » entre états (définition inductive) :

base Si s est un état d'acceptation et t n'en est pas un (ou réciproquement), alors s et t ne sont pas équivalents.

récurrence Si $\exists x \in X$ tel que $\delta(s, x)$ et $\delta(t, x)$ ne sont pas équivalents, alors s et t ne sont pas équivalents. [Aho et Ullman, 1993, p. 604]

La base conduit à démarrer (en général) avec deux classes : la classe des états d'acceptation, et la classe des états non finals.

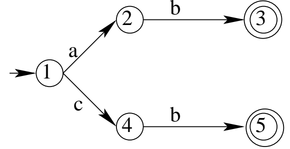
La suite de l'algorithme consiste à examiner, dans chaque classe ainsi définie, les états 2 à 2, pour « vérifier » qu'ils sont équivalents. En cas de non équivalence entre deux états, on les sépare, donnant naissance à de nouvelles classes, et à la nécessité de vérifier à nouveau pour toutes les classes existantes qu'elles ne contiennent que des états équivalents. L'algorithme aboutit lorsqu'après un examen 2 à 2 de tous les états de toutes les classes, aucune nouvelle séparation n'est déterminée.

Comment traiter (toutes) les paires d'une classe ? Au moyen d'une table de séparation, qui est une demi-matrice dans laquelle on représente la non équivalence des paires d'état.
Voici un exemple avec un ensemble $\{0, 1, 2, 3\}$. La table représentée à la figure 1.7 représente les 6 paires pertinentes, et indique que 1 doit être séparé de 3 ; et que 2 doit être séparé de 0.
On peut donc lire sur la table les (sous-)classes qui peuvent être définies sur la base de cette table : la partition $\{0, 3\}$ et $\{1, 2\}$, ou la partition $\{0, 1\}$ et $\{3, 2\}$. Il suffirait d'une croix de plus pour que le découpage soit non ambigü.

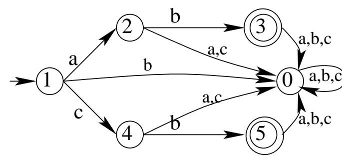
FIGURE 1.7 – Table de séparation, exemple pour la classe $\{0, 1, 2, 3\}$

	0	1	2	3
3		X		
2	X			
1				
0				

Le déroulement complet de l'algorithme peut être illustré, toujours avec le même exemple très simple :
Soit l'automate déterministe suivant.

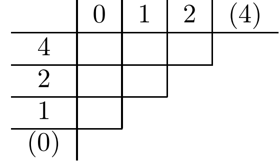


Il faut d'abord en faire un AFDC.



Étape 0 Classes initiales : F et $Q \backslash F$, ici : $C_1 = \{0, 1, 2, 4\}$ et $C_2 = \{3, 5\}$.
Etape 1 On a les deux classes C_1 et C_2.
Étape 1.1 On s'occupe d'abord de C_1.
Pour chaque paire d'états de C_1, existe-t-il une lettre qui ne mène pas de ces deux états dans la même classe ?

La liste des paires d'états à considérer pour C_1 est la suivante (ce sont des paires « non ordonnées ») : (0,1), (0,2), (0,4), (1,2), (1,4), (2,4), il s'agit donc de remplir les 6 cases du tableau suivant :



	0	1	2	(4)
4				
2				
1				
(0)				

⇒

<u>Pour 0 et 1</u> : a mène dans C_1 dans les 2 cas (0 pour 0 et 2 pour 1) ; b mène dans C_1 dans les 2 cas (0) ; *idem* pour c (0 pour 0 et 4 pour 1). 0 et 1 sont équivalents jusque là.
<u>Pour 0 et 2</u> : a mène dans C_1 dans les 2 cas ; b ne mène pas dans la même classe : $\delta(0, b) = 0 \in C_1$ alors que $\delta(2, b) = 3 \in C_2$. On peut donc noter que 0 et 2 doivent être séparés :

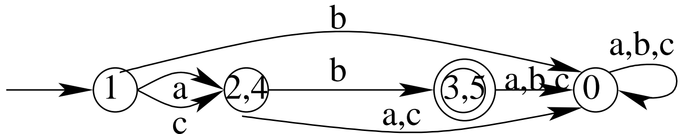
	0	1	2
4			
2	x		
1			

<u>Autres paires</u> : de façon analogue, on aboutit à la conclusion qu'il faut séparer 1 et 2, 1 et 4, et 0 et 2 :

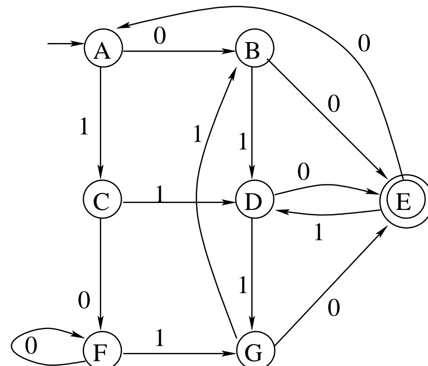
	0	1	2
4	x	x	
2	x	x	
1			

De ce tableau on peut déduire la nouvelle partition de C_1 :
$C_{1_1} = \{0, 1\}$ et $C_{1_2} = \{2, 4\}$.

Étape 1.2 On s'occupe de C_2 : la classe reste inchangée.
Étape 2. On a maintenant 3 classes : C_{1_1}, C_{1_2} et C_2.
Étape 2.1 Considérons d'abord $C_{1_1} = \{0, 1\}$. Il y a une seule paire à considérer : (0,1). La première lettre considérée, a, suffit à conclure à la nécessité de séparer 0 et 1 : par a de 1 on va en $2 \in C_{1_2}$, par a de 0 on va en $0 \in C_{1_1}$. Il faut donc créer les classes $C_{1_{1_1}} = \{0\}$ et $C_{1_{1_2}} = \{1\}$.
Étape 2.2 Considérons $C_{1_2} = \{2, 4\}$. Il y a aussi une seule paire à considérer, mais cette fois, on ne trouve pas de raison de les séparer : par b, on va dans les deux cas dans C_2, et par a et c dans les deux cas dans C_{1_1}. C_{1_2} est inchangée.
Étape 2.3 Considérons enfin C_2. Bien que cette classe n'aie pas été découpée à l'étape précédente, il faut refaire les calculs, puisque les autres classes ont changé. Dans le cas présent la classe reste inchangée.
Étape 3. On a maintenant 4 classes : $C_{1_{1_1}}$, $C_{1_{1_2}}$, C_{1_2} et C_2. Puisqu'il y a eu changement depuis l'étape précédente, il faut reconsidérer toutes les classes.
Étape 3.1. $C_{1_{1_1}}$ étant un singleton, il n'y a rien à faire.
Étape 3.2. $C_{1_{1_2}}$ étant un singleton, il n'y a rien à faire.
Étape 3.3. Le lecteur vérifiera facilement que C_{1_2} est inchangée.
Étape 3.4. Le lecteur vérifiera facilement que C_2 est inchangée.
L'étape 3 n'ayant induit aucune modification, on sait (on peut prouver) que l'on ne pourra pas découper plus les classes. L'algorithme peut s'arrêter.



Deuxième exemple

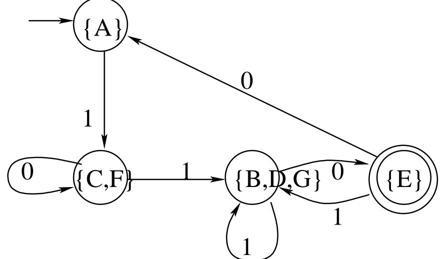


On peut créer un seul tableau pour représenter toutes les paires et le remplir à chaque étape.
À l'étape 0, on sépare simplement les finaux des autres (noté X_0).

	A	B	C	D	E	F
G					X_0	
F					X_0	
E	X_0	X_0	X_0	X_0		
D						
C						
B						

En parcourant le tableau (par exemple colonne par colonne), on peut construire les classes. Ici, à l'étape 0, il y a {A, B, C, D, F, G} et {E}.

	A	B	C	D	E	F
G	X_1		X_1		X_0	X_1
F	X_2	X_1		X_1	X_0	
E	X_0	X_0	X_0	X_0		
D	X_1		X_1			
C	X_2	X_1				
B	X_1					



Étape 1 (X_1) : {A, C, F}, {B, G, D} et {E}.
Étape 2 (X_2) : {A}, {C, F}, {B, G, D} et {E}.
Étape 3 : {A}, {C, F}, {B, G, D} et {E}. (point fixe)

Synthèse: algorithme de minimisation

Entrée	Automate déterministe complet reconnaissant $\mathcal{L}$
Sortie	Automate déterministe complet minimal reconnaissant $\mathcal{L}$
Méthode	Après avoir découpé l'ensemble d'états en classes d'équivalence avec un algorithme de point fixe, créer un automate où les états équivalents sont fusionnés.

1.3.3 Propriétés de fermeture

Comme on a défini des opérations (*internes*) sur les langages, on peut envisager des opérations internes sur les automates. Voici les plus courantes.

1.3.3.1 Union

Pour réaliser un automate qui reconnaît l'union de deux langages, il suffit de faire en sorte que le nouvel automate comprenne tous les chemins du premier automate et tous ceux du second. Un moyen simple de procéder est de créer un nouvel état initial, duquel partent des ε-transitions vers les états initiaux des deux automates à réunir, et de créer un nouvel état d'acceptation, qui sera la cible par une ε-transition de tous les (anciens) états d'acceptation des deux automates. Le résultat est bien sûr non déterministe. Noter qu'on peut aussi garder les états d'acceptation sans en créer de nouveau.
Exemple :

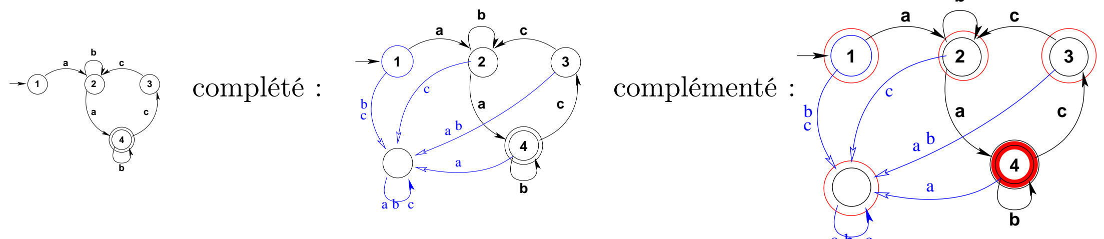
$\cup$ $=$

1.3.3.2 Concaténation, étoile

Il est facile d'imaginer, sur la même base que précédemment, comment créer un automate réalisant la concaténation de deux automates, ou l'étoile d'un automate. C'est en fait exactement ce que l'on fait dans l'algorithme de traduction d'une expression rationnelle en automate (cf. p. 18).

1.3.3.3 Complémentation

Le complément d'un langage $\mathcal{L}_1$ est l'ensemble de tous les mots du monoïde qui n'appartiennent pas à ce langage. En terme d'automate, il s'agit donc de tous les mots qui n'ont pas de chemin aboutissant à un état final dans l'automate reconnaissant $\mathcal{L}_1$.
L'algorithme pour construire le complément d'un automate est relativement intuitif : il suffit que tous les états d'échec de l'automate initial deviennent des états de réussite, et réciproquement. Pratiquement, il suffit de rendre terminaux les états non terminaux et réciproquement (on échange Q et $Q \setminus F$). Mais attention, il est nécessaire que tous les chemins possibles soient présent dans l'automate, et donc qu'il soit **complet** ; de même il est nécessaire que l'automate initial soit **déterministe**.
Exemple :



En exercice, le lecteur est invité à se figurer ce qui se produit lorsque ces contraintes ne sont pas vérifiées, et que l'on applique l'algorithme (échange de Q et $Q \setminus F$).

1.3.3.4 Intersection

Théorie : on sait que $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$. On pourrait donc utiliser les algorithmes précédents. Mais il y a une autre méthode, moins fastidieuse (ne pas oublier que la complémentation nécessite d'abord une déterminisation).
Intuitivement, l'idée est de parcourir "en parallèle" les deux automates, et de ne garder que les chemins qui existent dans les deux automates. Pour cela, les états du nouvel automate sont des couples (q_i, q_j), où q_i appartient au premier automate et q_j au second. Pour chaque lettre de transition, on crée le nouvel état-couple atteint, et on continue.

L_1	a	b
$\rightarrow$ 1	2	4
2	4	3
$\leftarrow$ 3	3	3
4	4	4

L_2	a	b
$\leftrightarrow$ 1	2	5
2	5	3
3	4	5
4	1	4
5	5	5

$L_1 \cap L_2$	a	b
$\rightarrow$ (1,1)	(2,2)	(4,5)
(2,2)	(4,5)	(3,3)
(4,5)	(4,5)	(4,5)
(3,3)	(3,4)	(3,5)
(3,4)	(3,1)	(3,4)
$\leftarrow$ (3,1)	(3,2)	(3,5)
(3,2)	(3,5)	(3,3)
(3,5)	(3,5)	(3,5)

Les mêmes contraintes que précédemment s'appliquent : on part de deux automates **déterministes complets**[2]. À noter aussi qu'un tel algorithme, comme l'algorithme de déterminisation, a le mérite de ne pas conserver les états non atteints depuis l'état initial.

2. En pratique, on peut simplifier légèrement les calculs car tous les états dont un des membres est un puits sont par définition des puits, il n'est pas utile de calculer les transitions partant de ces états.

1.4 Théorèmes d'équivalence

1.4.1 Le théorème triangulaire

On a établi un ensemble de résultats d'équivalence que l'on peut résumer de la façon suivante [3]

$$\mathcal{L}_{\text{Rec}} = \mathcal{L}_{\text{Rat}} = \mathcal{L}_{\text{Reg}}$$

où

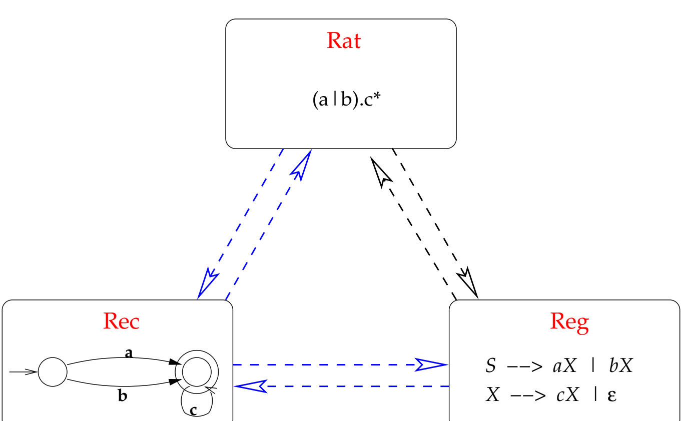
— $\mathcal{L}_{\text{Rec}}$ est la classe des langages **rec**onnaissables par un automate à nombre fini d'états ;

— $\mathcal{L}_{\text{Rat}}$ est la classe des langages que l'on peut décrire avec une expression **rat**ionnelle ;

— $\mathcal{L}_{\text{Reg}}$ est la classe des langages engendrés par une grammaire **rég**ulière.

On symbolise en général ce résultat sous la forme du triangle représenté à la figure 1.8.

FIGURE 1.8 – Théorème d'équivalence



La démonstration du thèorème peut se faire de manière *constructive* : par exemple, pour montrer que tout langage rationnel est reconnaissable, il suffit d'exhiber un algorithme qui prenant une expression rationnelle quelconque en entrée, produit en sortie un automate qui reconnaît le même langage. Outre la difficulté de définir l'algorithme, il faut pour que la démonstration soit valide, d'une part garantir que l'algorithme fournit une réponse pour toute entrée possible, et d'autre part démontrer que l'automate fourni reconnaît bien le même langage.

Nous ne verrons pas ici ces deux derniers aspects de la démonstration (qui sont assez techniques), nous nous contenterons de donner (sous forme d'exemples) les algorithmes pour certaines des flèches pointillées de la figure (en bleu). Les algorithmes manquants existent, mais ils sont théoriquement inutiles si les deux autres équivalences sont établies.

Plus précisément, nous définirons les algorithmes permettant de démontrer :

$\mathcal{L}_{\text{Rec}} \subset \mathcal{L}_{\text{Reg}}$ Algorithme construisant une grammaire régulière à partir d'un automate, en identifiant les non-terminaux de la grammaire et les états de l'automate. (§ 1.4.2.2)

$\mathcal{L}_{\text{Reg}} \subset \mathcal{L}_{\text{Rec}}$ Algorithme très proche du précédent, toujours basé sur l'identité entre symbole non-terminal et état. (§ 1.4.2.3)

3. Le théorème de Kleene correspond à l'équation : $\mathcal{L}_{\text{Rec}} = \mathcal{L}_{\text{Rat}}$.

$\mathcal{L}_{\text{Rat}} \subset \mathcal{L}_{\text{Rec}}$ Algorithme basé sur la décomposition syntaxique de l'expression rationnelle, et la composition d'automates correspondants. (§ 1.4.3.1)

$\mathcal{L}_{\text{Rec}} \subset \mathcal{L}_{\text{Rat}}$ Algorithme de Mac Naughton et Yamada, qui construit itérativement, en partant de l'automate initial, un *automate fini généralisé* qui finit par ne contenir qu'une transition étiquetée par une expression rationnelle équivalente. (§ 1.4.3.2)

1.4.2 Grammaires et automates

1.4.2.1 Principe

Rappel une grammaire **régulière** (dite aussi linéaire) est une grammaire dont toutes les règles de production sont sous l'une des formes suivantes [4] : $A \to xB$
$A \to x$
$A \to \varepsilon$

Note On peut toujours proposer une définition sans ε-production, ou plutôt sans autre ε-production qu'une règle $S \to \varepsilon$, où S est l'axiome, et S est inaccessible.

Le principe de correspondance entre automates et grammaires régulières est très intuitif : il correspond à l'observation que chaque transition dans un automate produit exactement un symbole, de même que chaque dérivation dans une grammaire régulière. Le tableau 1.1 résume cette correspondance, qui donne les bases des algorithmes dans les deux sens.

Automate	Grammaire
(A) —x→ (B)	$A \to xB$
→(A)	Axiome = A
((B)) (état final)	$B \to \varepsilon$
(A) —x→ ((A')) (A' nouvel état)	$A \to x$

Table 1.1 – Correspondances automate ↔ grammaire régulière

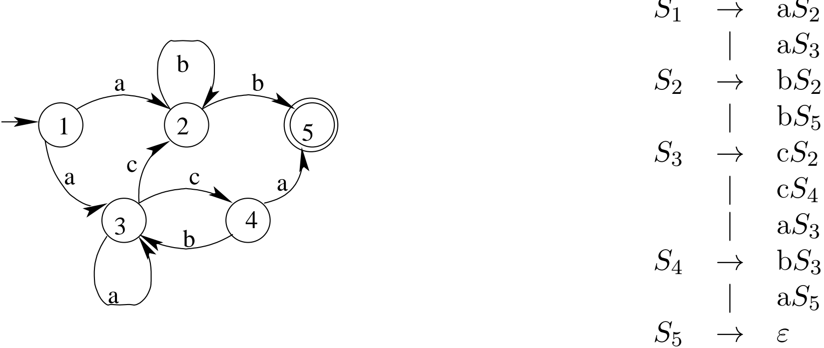
1.4.2.2 Automates → grammaires régulières

On peut partir d'un automate quelconque (non déterministe, non complet), il suffit de considérer une à une toutes les transitions et de produire les règles correspondantes d'après le tableau 1.1.

Si l'automate contient des transitions vides, on peut bien sûr s'en débarrasser (algorithme déjà vu), ou bien les traduire en productions singulières ((A) —ε→ (B) devient $A \to B$). Mais il faut ensuite supprimer les productions singulières (qui ne sont pas permises dans une grammaire régulière), avec un algorithme qui ressemble beaucoup à l'algorithme de suppression des ε-productions dans un automate (voir aussi les algorithmes de « nettoyage » de grammaires algébriques).

L'application de l'algorithme est illustrée à la figure 1.9 avec un automate non déterministe.

4. Où, conformément aux conventions habituelles, A et B sont des non-terminaux, et x est un symbole terminal. La définition donnée ici correspond à une grammaire linéaire/régulière **gauche**. Définition



$$
\begin{array}{lcl}
S_1 & \rightarrow & \mathrm{a}S_2 \\
 & | & \mathrm{a}S_3 \\
S_2 & \rightarrow & \mathrm{b}S_2 \\
 & | & \mathrm{b}S_5 \\
S_3 & \rightarrow & \mathrm{c}S_2 \\
 & | & \mathrm{c}S_4 \\
 & | & \mathrm{a}S_3 \\
S_4 & \rightarrow & \mathrm{b}S_3 \\
 & | & \mathrm{a}S_5 \\
S_5 & \rightarrow & \varepsilon
\end{array}
$$

Figure 1.9 – Exemple Rec → Reg (automate non déterministe)

1.4.2.3 Grammaires régulières → automates

Partant d'une grammaire régulière, il suffit de créer un état pour chaque non-terminal, et de "traduire" chaque règle de production en utilisant le même tableau de correspondance. Le seul cas un peu particulier concerne les règles de la forme $A \rightarrow x$, pour lesquelles il suffit de remarquer que ce sont des règles terminales (la dérivation s'arrête nécessairement dès qu'une production de cette forme est déclenchée). Il faut créer un nouvel état, terminal (A' dans le tableau). Pour comprendre cette correspondance, on peut observer que la dérivation $A \rightarrow x$ est équivalente à une dérivation avec les deux règles $A \rightarrow xA'$, et $A' \rightarrow \varepsilon$.

1.4.3 Automates et expressions rationnelles

1.4.3.1 Expression rationnelle → Automate

On peut montrer (voir section 1.3.3) que la *réunion*, la *concaténation*, et l'*étoile* peuvent être définis sur les automates ; il est donc possible, par exemple, de construire un automate qui reconnaît $L_1 \cup L_2$ par la réunion de l'automate qui reconnaît L_1 et de l'automate qui reconnaît L_2. Ces considérations permettent de définir facilement un algorithme de "traduction" d'une expression rationnelle quelconque en un automate reconnaissant le même langage.
Voici cet algorithme, spécifié d'abord sous forme mathématique, puis sous la forme d'un tableau de correspondance graphique, dans le même esprit que le tableau 1.1 donné plus haut (mais orienté cette fois-ci).

Traduction récursive d'une expression rationnelle en un automate

1. Au mot vide ε, on associe l'automate $\langle X, \{q_0\}, \{q_0\}, \{q_0\}, \emptyset \rangle$
2. À l'expression rationnelle x ($x \in X$), on associe l'automate
$$\langle X, \{q_0, q_1\}, \{q_0\}, \{q_1\}, \{(q_0, x, q_1)\} \rangle$$
3. Soit R une expression rationnelle, associée à l'automate $\langle X, Q_R, I_R, F_R, \delta_R \rangle$;
à R^*, on associe l'automate $\langle X, Q_R \cup \{Q_0\}, \{Q_0\}, \{Q_0\}, \delta'_R \rangle$ [5],
où $\delta'_R = \delta_R \cup \bigcup_{q \in I_R} (Q_0, \varepsilon, q) \cup \bigcup_{q \in F_R} (q, \varepsilon, Q_0)$
4. Soient R et S deux expressions rationnelles auxquelles ont été associés respectivement $\langle X, Q_R, I_R, F_R, \delta_R \rangle$ et $\langle X, Q_S, I_S, F_S, \delta_S \rangle$, dont on suppose que tous les états sont distincts ($Q_S \cap Q_R = \emptyset$).

analogue possible à droite.

5. Q_0 est un nouvel état t.q. $Q_0 \notin Q$.

(a) À RS on associe l'automate

$$\left\langle X, Q_R \cup Q_S, I_R, F_S, \delta_R \cup \delta_S \cup \bigcup_{q \in F_R} \bigcup_{q' \in I_S} (q, \varepsilon, q') \right\rangle$$

(b) À $R|S$ on associe l'automate

$$\left\langle X, Q_R \cup Q_S, Q_0, F_R \cup F_S, \delta_R \cup \delta_S \cup \bigcup_{q \in I_R \cup I_S} (Q_0, \varepsilon, q) \right\rangle$$

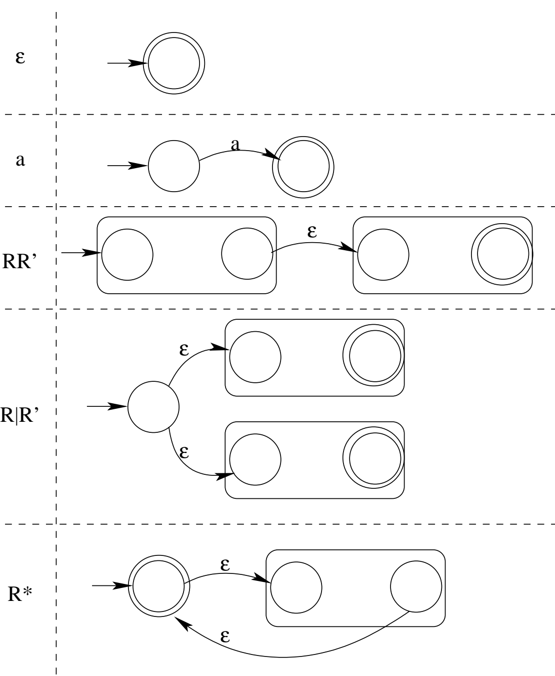


Table 1.2 – D'une expression rationnelle vers un automate

1.4.3.2 Automate → expression rationnelle

Il s'agit d'un des algorithmes les plus sophistiqués de la série présentée dans ce chapitre, c'est l'algorithme de McNaughton et Yamada.

L'algorithme est divisé en deux étapes. Lors de la première étape, l'automate est transformé en un automate d'un autre type, appelé *automate (fini) généralisé*. Cet automate est ensuite transformé (itérativement) en expression régulière lors d'une seconde étape.

Un automate généralisé est un automate dont les transitions sont étiquetées par des expressions rationnelles (plus le symbole $\emptyset$, voir plus loin) et non pas simplement par des symboles ou ε. L'automate généralisé lit le mot à reconnaître par blocs de symboles.

La première étape de l'algorithme, qui consiste à **transformer l'automate initial en automate généralisé**, revient à ajouter un état initial et un état final reliés par des ε-transitions aux états initial et finaux de l'automate initial ; puis à faire en sorte que tous les états soient reliés à tous les états, soit par une transition marquée $\emptyset$ (lorsqu'il n'existe pas de chemin entre les deux états), soit par une transition unique portant l'étiquette de

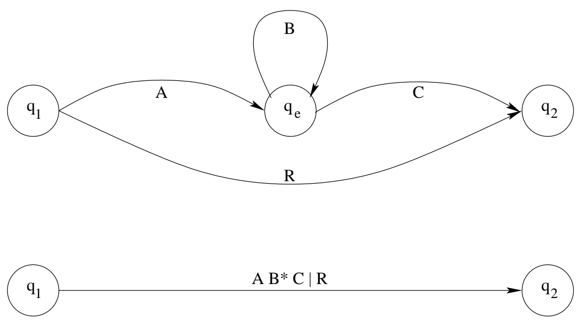
l'automate initial (ou l'union des étiquettes s'il y a plusieurs transitions entre deux états). Il est facile de vérifier que l'automate généralisé reconnaît le même langage (les transitions $\emptyset$ sont "non passantes").

Déf. 9 (Automate généralisé)

- L'état initial possède une transition vers tous les autres états (éventuellement une transition "non passsante", étiquetée par $\emptyset$) ;
- Aucun état n'a de transition vers l'état initial
- Il existe un et un seul état d'acceptation,
  - distinct de l'état initial,
  - qui n'a aucune transition vers les autres états
  - qui est atteint par tous les autres états
- Tous les états (sauf initial et acceptation) possèdent une et une seule transition vers tous les autres états.

La seconde étape est une **réduction itérative du nombre d'états** de l'automate généralisé, jusqu'à obtenir un automate n'ayant que deux états, et une transition qui sera étiquetée par l'expression rationnelle correspondant au langage reconnu. Il est clair que cette réduction doit conserver à chaque étape le langage reconnu.
Chaque étape de cette réduction consiste en la **suppression d'un état** en réarrangeant l'automate de façon à reconnaître le même langage.
Soit q_e l'état à supprimer, il faut considérer **tous** les couples d'états (q_1, q_2), et faire en sorte que les transitions allant de q_1 à q_2 en passant ou non par q_e soient "synthétisées" sur une seule transition représentant tous les chemins possibles. Cette élimination est représentée par la figure 1.10.

Figure 1.10 – Elimination d'un état, Algorithme de McNaughton & Yamada



Il est important de noter que l'élimination d'un état conduit à considérer **tous** les couples d'états de l'automate, y compris les couples (q_i, q_i) (mais en tenant compte de la définition d'un automate généralisé, le couple (q_0, q_i) est considéré, mais pas le couple (q_i, q_0) (où q_0 est l'état initial)).

Un exemple détaillé Le point important est de bien déterminer l'ensemble des combinaisons à considérer. Soit l'automate représenté à la figure 1.11. À l'issue de la première étape, on passe à l'automate généralisé représenté à la figure 1.12 (pour rendre la figure

plus lisible, on a représenté les arcs non passants par des traits pointillés).

Figure 1.11 – McNaughton & Yamada (McN&Y), exemple : automate initial

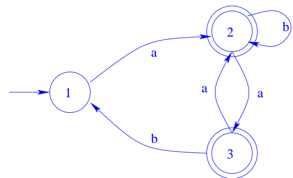
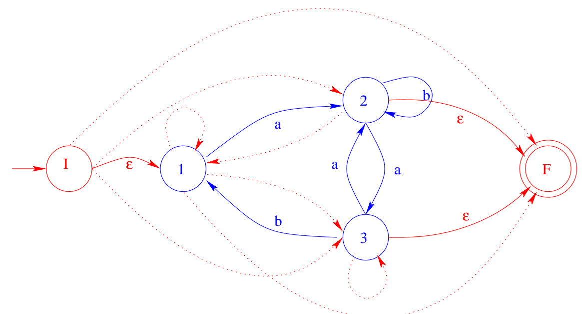


Figure 1.12 – McN&Y, exemple : automate généralisé



La réduction consiste donc à éliminer successivement les états de l'automate, en ne laissant que I et F. Commençons par la suppression de l'état 1. Le tableau de gauche de la table 1.3 liste la totalité des triplets de la forme $(q_i, 1, q_j)$; pour chaque triplet, on construit l'étiquette de la transition (q_i, q_j) qui reconnaît le même langage. La troisième colonne donne la forme simplifiée de l'expression rationnelle [6].

I 1 I	*pas à considérer, I n'a pas d'arcs entrants*		
I 1 2	$\varepsilon\ \emptyset^*$	$a \mid \emptyset$	a
I 1 3	$\varepsilon\ \emptyset^*$	$\emptyset \mid \emptyset$	$\emptyset$
I 1 F	$\varepsilon\ \emptyset^*$	$\emptyset \mid \emptyset$	$\emptyset$
2 1 2	$\emptyset\ \emptyset^*$	$a \mid b$	b
2 1 3	$\emptyset\ \emptyset^*$	$\emptyset \mid a$	a
2 1 F	$\emptyset\ \emptyset^*$	$\emptyset \mid \varepsilon$	ε
3 1 2	$b\ \emptyset^*$	$a \mid a$	$ba\|a$
3 1 3	$b\ \emptyset^*$	$\emptyset \mid \emptyset$	$\emptyset$
3 1 F	$b\ \emptyset^*$	$\emptyset \mid \varepsilon$	ε

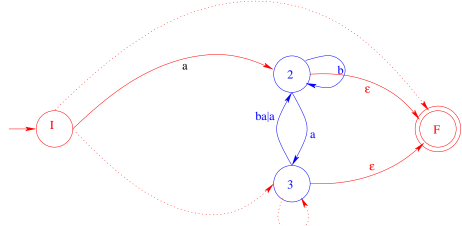
I 2 3	$a\ b^*$	$a \mid \emptyset$	ab^*a
I 2 F	$a\ b^*$	$\varepsilon \mid \emptyset$	ab^*
3 2 3	$ba\|a\ b^*$	$a \mid \emptyset$	$(ba\|a)b^*a$
3 2 F	$ba\|a\ b^*$	$\varepsilon \mid \varepsilon$	$(ba\|a)b^*\|\varepsilon$

Table 1.3 – Les triplets impliquant l'état 1, puis l'état 2

Cette table nous donne directement le nouvel automate généralisé, débarrassé de l'état 1, mais reconnaissant le même langage. Il est représenté à la figure 1.13.

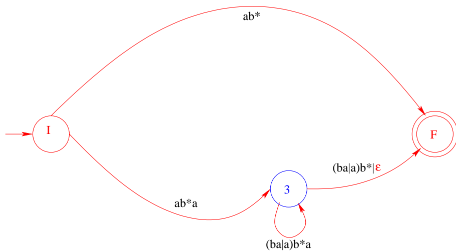
6. Les définitions courantes de la sémantique du langage des expressions rationnelles donnent aisément, pour r une expression rationnelle quelconque : $\emptyset r = \emptyset$, $r\emptyset = \emptyset$, $\emptyset^* = \varepsilon$, et enfin $r\varepsilon = \varepsilon r = r$. Voir la section 2.1.3 « équivalence et réductions » de l'extrait de [Yvon et Demaille, 2005], à la page 27.

FIGURE 1.13 – McN&Y, exemple : Automate généralisé après suppression de 1



On recommence alors pour l'état 2, le nombre de combinaisons à considérer est bien sûr beaucoup plus réduit. Voir tableau de droite de la table 1.3 L'automate résultant est représenté à la figure 1.14.

FIGURE 1.14 – McN&Y, exemple : Automate généralisé après suppression de 2



Il ne reste plus qu'à supprimer l'état 3, ce qui se fait en appliquant directement la règle illustrée plus haut. L'expression résultante est :

$$\left(ab^*a\left((ba|a)b^*a\right)^*\left((ba|a)b^*|\varepsilon\right)\Big|ab^*\right)$$

1.5 Propriétés des langages rationnels

1.5.1 Le lemme de pompage

Déf. 10 (Lemme de pompage)

Soit L un langage rationnel infini. Il existe un entier k tel que :

$$\forall x \in L,\ |x| > k,\ \exists u, v, w \text{ tels que } x = uvw, \text{ avec } \begin{array}{rl} (i) & |v| \geq 1 \\ (ii) & |uv| \leq k \\ (iii) & \forall i \geq 0,\ uv^iw \in L \end{array}$$

Intuitivement : **tout mot** assez long de L comprend un facteur, éventuellement répété, qui apparaît aussi dans les mots plus courts de L. Dit d'une autre manière, ce lemme dit que dans les mots assez longs, il existe un facteur qui peut être itéré en "restant" dans L.

Un premier exemple Illustrons avec un langage qui vérifie trivialement le lemme de pompage : a^*bc. Soit $k = 3$, le mot abc est assez long, on peut le décomposer : $\underbrace{\varepsilon}_{u}\ \underbrace{a}_{v}\ \underbrace{b\ c}_{w}$ et on peut vérifier chacune des 3 propriétés du lemme :

- $|v| \geq 1$ $(v = a)$
- $|uv| \leq k$ $(uv = a)$
- $\forall i \in \mathbb{N}$, uv^*w qui est égal à a^*bc, appartient au langage, par définition.

1.5.2 Conséquences

Il s'agit ici de conséquences « théoriques » sur la décidabilité.
Rappelons tout d'abord qu'avec l'automate (dont on sait qu'il peut être minimal et déterministe), on dispose d'un algorithme qui peut répondre à la question $u \in X^* \in L(A)$? en un temps fini, dans tous les cas : cet algorithme **décide**. On dira donc que le problème de l'appartenance (d'un mot à un langage rationnel) est **décidable**.
Mais il y a des questions plus complexes (même si elles semblent simples), dans le sens où la réponse ne sera pas toujours positive dans toutes les classes de langage.
Par exemple, pour un automate donné, la question de savoir si le langage reconnu est vide, ou infini, reçoit une réponse en un temps fini grâce au théorème suivant :

Déf. 11 (Conséquence du lemme de pompage)

Soit $\mathcal{A}$ un automate à k états :

1. $L(\mathcal{A}) \neq \emptyset$ ***ssi*** $\mathcal{A}$ reconnaît (au moins) un mot u t.q. $|u| < k$.
2. $L(\mathcal{A})$ est infini ***ssi*** $\mathcal{A}$ reconnaît (au moins) un mot u t.q. $k \leq |u| < 2k$.

Démonstrations

1. Première conséquence
   (a) Dans un sens, l'implication est triviale : si $\mathcal{A}$ reconnaît un mot, alors $L(\mathcal{A})$ est non vide. ■
   (b) Dans l'autre sens : soit $L(\mathcal{A})$ non vide, et soit u le plus petit mot de $L(\mathcal{A})$.
   Supposons que $|u| = l \geq k$. Le chemin de reconnaissance de u, $q_0, q_1, \ldots, q_l$ contient au moins $k+1$ états, qui ne peuvent pas être tous différents. Il y a donc un état q qui est visité deux fois dans la reconnaissance de u, il y a donc un circuit C dans le chemin. Alors le mot u' correspondant au même chemin de reconnaissance, mais dans lequel le circuit C est court-circuité, est un mot de longueur strictement inférieure à k. Par définition, $u' \in L(\mathcal{A})$, et il est plus court que u, ce qui est contraire à l'hypothèse.
   Le plus petit mot de $L(\mathcal{A})$ est donc de longueur strictement inférieure à k. ■
2. Deuxième conséquence
   (a) Dans un sens, on peut revenir directement au lemme de pompage : s'il existe un mot de longueur supérieure à k, le lemme de pompage s'applique, et il existe une infinité de mots dans $L(\mathcal{A})$. ■
   (b) Dans l'autre sens : Soit $L(\mathcal{A})$ infini, alors il existe un mot de longueur supérieure à k. Soit u le plus petit mot de longueur supérieure à k. Soit il est de longueur strictement inférieure à $2k$, et le résultat est prouvé, soit il est au moins de longueur égale à $2k$, mais par le lemme de pompage on peut court-circuiter dans ce mot un facteur de longueur inférieure ou égale à k (car si le circuit est plus long c'est qu'il contient lui-même un circuit), ce qui exhibe un mot de longueur supérieure à k et plus petit que u, ce qui est contraire à l'hypothèse que u est le plus petit. C'est donc que le plus petit mot de longueur supérieure à k est de longueur inférieure à $2k$. ■

Conséquence Pour décider si le langage reconnu par un automate est non vide, il suffit de tester tous les mots de longueur inférieure à k. On est sûr de disposer d'un algorithme décidable pour répondre à la question $L(\mathcal{A}) \stackrel{?}{=} \emptyset$.

Conséquence De même, pour décider si le langage reconnu par un automate est infini, on sait qu'il suffit de considérer tous les mots de longueur comprise entre $2k$ et k.

Remarque Déterminer si les langages engendrés par deux automates sont le même langage revient à déterminer si le langage $[L(A_1) \cap \overline{L(A_2)}] \cup [L(A_2) \cap \overline{L(A_1)}]$ est vide. Par conséquent, on dispose aussi d'un algorithme décidable pour décider si deux automates reconnaissent le même langage.

Synthèse

— On dispose avec le lemme de pompage d'un moyen de démontrer qu'un langage n'est pas rationnel ;

— et on se souvient que pour démontrer qu'un langage est rationnel, le lemme n'est pas utilisable — il existe des langages non rationnels qui vérifient le lemme —, mais il suffit de trouver un automate ou une expression rationnelle.

— Tous les problèmes suivants sont décidables dans $\mathcal{L}_{\text{Rat}}$: $u \in X^* \stackrel{?}{\in} L(A)$
$L(A) \stackrel{?}{=} \emptyset$
$L(A) \stackrel{?}{\text{est}}$ infini
$L(A_1) \stackrel{?}{=} L(A_2)$

Remerciements

Merci à Alexis Nasr, François Yvon, Marcel Cori, Robert Cori, Giuliana Bianchi, Marie Candito, Corentin Ribeyre, Timothée Bernard, et les étudiants de Toulouse, Lille, São Carlos, et bien sûr Paris Diderot, qui ont subi certains de ces enseignements.

Bibliographie

[Aho et Ullman, 1993] Alfred Aho et Jeffrey Ullman. *Concepts fondamentaux de l'informatique*. Dunod, 1993. Traduction de *Foundations of Computer Science*, 1992, W.H. Freeman and Company, New York.

[Yvon et Demaille, 2005] François Yvon et Akim Demaille. Théorie des langages. notes de cours. Polycopié pour le cours de 1e année BCI, Telecom ParisTech, Novembre 2005.

Annexe : Expressions rationnelles

Les pages qui suivent sont extraites de [Yvon et Demaille, 2005] (reproduites avec l'autorisation des auteurs), qui est un excellent polycopié, extrêmement pertinent pour les cours « bases formelles du TAL », « langages formels » du master de Linguistique Informatique Paris Diderot. En particulier, on y trouve les algorithmes de parsing tabulaire exposés en français, ce qui est précieux.

La version dont nous avons extrait les pages qui suivent est une version de 2005, il existe deux versions plus récentes (2008), accessibles depuis la page de François Yvon : `https://perso.limsi.fr/yvon/classes/thl/thl-1.pdf` pour la version courte, et `https://perso.limsi.fr/yvon/classes/thl/thl-2.pdf` pour la version avec les extensions.

Chapitre 2

Langages et expressions rationnels

Une première famille de langage est introduite, la famille des langages *rationnels*. Cette famille contient en particulier tous les langages finis, mais également de nombreux langages infinis. La caractéristique de tous ces langage est la possibilité de les *décrire* par des formules (on dit aussi *motifs*, en anglais *patterns*) très simples. L'utilisation de ces formules, connues sous le nom d'expressions rationnelles[1]s'est imposé sous de multiples formes comme la «bonne» manière de décrire des motifs représentant des ensembles de mots.

Après avoir introduit les principaux concepts formels (à la section 2.1), nous étudions quelques systèmes informatiques classiques mettant ces concepts en application.

2.1 Rationalité

2.1.1 Langages rationnels

Parmi les opérations définies dans $\mathcal{P}(\Sigma^\star)$ à la section 1.4, trois sont distinguées et sont qualifiées de *rationnelles* : il s'agit de l'union, de la concaténation et de l'étoile. *A contrario*, notez que la complémentation et l'intersection ne sont pas des opérations rationnelles. Cette distinction permet de définir une famille importante de langages : les langages *rationnels*.

Définition 2.1 (Langages rationnels). *Soit Σ un alphabet fini. Les* langages rationnels *sur Σ sont définis inductivement par :*
(i) $\{\varepsilon\}$ et $\emptyset$ sont des langages rationnels
(ii) $\forall a \in \Sigma, \{a\}$ est un langage rationnel
(iii) si L, L_1 et L_2 sont des langages rationnels, alors $L_1 \cup L_2$, L_1L_2, et $L^\star$ sont également des langages rationnels.
Est alors rationnel tout langage construit par un nombre fini d'application de la récurrence (iii).

Par définition, tous les langages finis sont rationnels, puisqu'ils se déduisent des singletons par un nombre fini d'application des opérations d'union et de concaténation. Par définition également, l'ensemble des langages rationnels est clos pour les trois opérations rationnelles (on dit aussi qu'il est rationnellement clos).

[1]On trouve également le terme d'expression *régulière*, mais cette terminologie, quoique bien installée, est trompeuse et nous ne l'utiliserons pas dans ce cours.

La famille des langages rationnels correspond précisément au plus petit ensemble de langages qui (i) contient tous les langages finis, (ii) est rationnellement clos.

Un langage rationnel peut se décomposer sous la forme d'une formule finie, correspondant aux opérations (rationnelles) qui permettent de le construire. Prenons l'exemple du langage sur $\{0, 1\}$ contenant tous les mots dans lesquels apparaît au moins une fois le facteur 111. Ce langage peut s'écrire : $\{0, 1\}^*\{111\}\{0, 1\}^*$, exprimant que les mots de ce langage sont construits en prenant deux mots quelconques de $\Sigma^\star$ et en insérant entre eux le mot 111 : on peut en déduire que ce langage est bien rationnel. Les *expressions rationnelles* définissent un système de formules qui simplifient et étendent ce type de notation des langages rationnels.

2.1.2 Expressions rationnelles

Définition 2.2 (Expressions rationnelles). *Soit Σ un alphabet fini. Les* expressions rationnelles *(RE) sur Σ sont définies inductivement par :*
(i) ε et $\emptyset$ sont des expressions rationnelles
(ii) $\forall a \in \Sigma$, a est une expression rationnelle
(iii) si e_1 et e_2 sont deux expressions rationnelles, alors $(e_1 + e_2)$, (e_1e_2), $(e_1^\star)$ et $(e_2^\star)$ sont également des expressions rationnelles.
On appelle alors expression rationnelle toute formule construite par un nombre fini d'application de la récurrence (iii).

Illustrons ce nouveau concept, en prenant maintenant l'ensemble des caractères alphabétiques comme ensemble de symboles :
– $r, e, d, é$, sont des RE (par (ii))
– (re) et $(dé)$ sont des RE (par (iii))
– $((((fa)i)r)e)$ est une RE (par (ii), puis (iii)
– $((re) + (dé))$ est une RE (par (iii))
– $((((re) + (dé)))^\star)$ est une RE (par (iii))
– $(((re + dé))^\star((((fa)i)r)e))$ est une RE (par (iii))
– ...
À quoi servent ces formules ? Comme annoncé, elles servent à dénoter des langages rationnels. L'interprétation (la sémantique) d'une expression est définie par les règles inductives suivantes :
(i) ε dénote le langage $\{\varepsilon\}$ et $\emptyset$ dénote le langage vide.
(ii) $\forall a \in \Sigma$, a dénote le langage $\{a\}$
(iii.1) $(e_1 + e_2)$ dénote l'union des langages dénotés par e_1 et par e_2
(iii.2) (e_1e_2) dénote la concaténation des langages dénotés par e_1 et par e_2
(iii.3) $(e^\star)$ dénote l'étoile du langage dénoté par e
Revenons à la formule précédente : $(((re + dé))^\star faire)$ dénote l'ensemble des mots formés en itérant à volonté un des deux préfixes *re* ou *dé*, concaténé au suffixe *faire* : cet ensemble décrit en fait un ensemble de mots existants ou potentiels de la langue française qui sont dérivés par application d'un procédé tout à fait régulier de préfixation verbale.

Par construction, les expressions rationnelles permettent de dénoter précisément tous les langages rationnels, et rien de plus. Si, en effet, un langage est rationnel, alors il existe une expression rationnelle qui le dénote. Ceci se montre par une simple récurrence sur le nombre d'opérations rationnelles utilisées pour construire le langage. Réciproquement, si un langage est dénoté par une expression rationnelle, alors il est lui-même rationnel [de nouveau par induction sur le nombre d'étapes dans la définition de l'expression]. Ce dernier point est important, car il fournit une première méthode pour *prouver* qu'un langage est rationnel : il suffit pour cela d'exhiber une

expression qui le dénote.

Pour alléger les notations (et limiter le nombre de parenthèses), on imposera les règles de priorité suivantes : l'étoile ($\star$) est l'opérateur le plus liant, puis la concaténation, puis l'union (+). Ainsi, $aa^\star + b^\star$ s'interprète-t-il comme $((a(a^\star)) + (b^\star))$.

2.1.3 Équivalence et réductions

La correspondance entre expression et langage n'est pas biunivoque : chaque expression dénote un unique langage, mais à un langage donné peuvent correspondre plusieurs expressions différentes. Ainsi, les deux expressions suivantes : $a^\star(a^\star ba^\star ba^\star)^\star$ et $a^\star(ba^\star ba^\star)^\star$ sont-elles en réalité deux variantes notationnelles du même langage sur $\Sigma = \{a, b\}$.

Définition 2.3 (Expressions rationnelles équivalentes). *Deux expressions rationnelles sont* équivalentes *si elles dénotent le même langage.*

Comment déterminer automatiquement que deux expressions sont équivalentes ? Existe-t-il une expression canonique, correspondant à la manière la plus courte de dénoter un langage ? Cette question n'est pas anodine : pour calculer efficacement le langage associé à une expression, il semble préférable de partir de la version la plus simple, afin de minimiser le nombre d'opérations à accomplir.

Un élément de réponse est fourni avec les formules de la Table 2.1, qui expriment, (par le signe =), un certain nombre d'équivalences élémentaires :

$\emptyset e = e\emptyset = \emptyset$	$\varepsilon e = e\varepsilon = e$
$\emptyset^\star = \varepsilon$	$\varepsilon^\star = \varepsilon$
$e + f = f + e$	$e + \emptyset = e$
$e + e = e$	$e^\star = (e^\star)^\star$
$e(f + g) = ef + eg$	$(e + f)g = eg + fg$
$(ef)^\star e = e(fe)^\star$	
$(e + f)^\star = e^\star(e + f)^\star$	$(e + f)^\star = (e^\star + f)^\star$
$(e + f)^\star = (e^\star f^\star)^\star$	$(e + f)^\star = (e^\star f)^\star e^\star$

Tab. 2.1 – Identités rationnelles

En utilisant ces identités, il devient possible d'opérer des transformations purement syntaxiques (c'est-à-dire qui ne changent pas le langage dénoté) d'expressions rationnelles, en particulier pour les simplifier. Un exemple de réduction obtenue par application de ces expressions est le suivant :

$$\begin{aligned} bb^\star(a^\star b^\star + \varepsilon)b &= b(b^\star a^\star b^\star + b^\star)b \\ &= b(b^\star a^\star + \varepsilon)b^\star b \\ &= b(b^\star a^\star + \varepsilon)bb^\star \end{aligned}$$

La conceptualisation algorithmique d'une stratégie efficace permettant de réduire les expressions rationnelles sur la base des identités de la Table 2.1 étant un projet difficile, l'approche la plus utilisée pour tester l'équivalence de deux expressions rationnelles n'utilise pas directement ces identités, mais fait plutôt appel à leur transformation en des automates finis, qui sera présentée dans le chapitre suivant (à la section 3.2.2).

2.2 Extensions notationnelles

Les expressions rationnelles constituent un outil puissant pour décrire des langages simples (rationnels). La nécessité de décrire de tels langages étant récurrente en informatique, ces formules sont donc utilisées, avec de multiples extensions, dans de nombreux outils d'usage courant.

Par exemple, `grep` est un utilitaire disponible sous UNIX pour rechercher les occurrences d'un mot(if) dans un fichier texte. Son utilisation est simplissime :

```
> grep 'chaine' mon.texte
```

imprime sur la sortie standard toutes les *lignes* du fichier `mon.texte` contenant au moins une occurrence du mot 'chaine'.

En fait `grep` permet un peu plus : à la place d'un mot unique, il est possible d'imprimer les occurrences de tous les mots d'un langage rationnel quelconque, ce langage étant défini sous la forme d'une expression rationnelle. Ainsi, par exemple :

```
> grep 'cha*ine' mon.texte
```

recherche (et imprime) toute occurrence d'un mot du langage $cha^{\star}ine$ dans le fichier `mon.texte`. Étant donné un motif exprimé sous la forme d'une expression rationnelle e, `grep` analyse le texte ligne par ligne, testant pour chaque ligne si elle appartient (ou non) au langage $\Sigma^{\star}(e)\Sigma^{\star}$; l'alphabet (implicitement) sous-jacent étant l'alphabet ASCII ou le jeu de caractère étendu IS0 Latin 1.

La syntaxe des expressions rationnelles permises par `grep` fait appel aux caractères '`*`' et '`|`' pour noter respectivement les opérateurs $\star$ et $+$. Ceci implique que, pour décrire un motif contenant le symbole '`*`', il faudra prendre la précaution d'éviter qu'il soit interprété comme un opérateur, en le faisant précéder du caractère d'échappement '\'. Il en va de même pour les autres opérateurs (`|`, `(`, `)`)... et donc aussi pour `\`. La syntaxe complète de `grep` inclut de nombreuses extensions notationnelles, permettant de simplifier grandement l'écriture des expressions rationnelles, au prix de la définition de nouveaux *caractères spéciaux*. Les plus importantes de ces extensions sont présentées dans la Table 2.2.

Supposons, à titre illustratif, que nous cherchions à mesurer l'utilisation de l'imparfait du subjonctif dans les romans de Balzac, supposément disponibles dans le (volumineux) fichier `Balzac.txt`. Pour commencer, un peu de conjugaison : quelles sont les terminaisons possibles ? Au premier groupe : `asse`, `asses`, `ât`, `âmes`, `assions`, `assiez`, `assent`. On trouvera donc toutes les formes du premier groupe avec un simple[2] :

```
> grep -E '(ât|âmes|ass(e|es|ions|iez|ent))' Balzac.txt
```

Guère plus difficile, le deuxième groupe : `isse`, `isses`, `î`, `îmes`, `issions`, `issiez`, `issent`. D'où le nouveau motif :

```
> grep -E '([îâ]t|[âî]mes|[ia]ss(e|es|ions|iez|ent))' Balzac.txt
```

Le troisième groupe est autrement complexe : disons simplement qu'il implique de considérer également les formes en `usse` (pour "boire" ou encore "valoir"); les formes en `insse` (pour "venir", "tenir" et leurs dérivés...). On parvient alors à quelque chose comme :

[2] L'option `-E` donne accès à toutes les extensions notationnelles

L'expression	dénote	remarque
.	Σ	. vaut pour n'importe quel symbole
Répétitions		
e*	$e^\star$	
e+	$ee^\star$	
e?	$e + \varepsilon$	
e{n}	(e^n)	
e{n,m}	$(e^n + e^{n+1} \ldots + e^m)$	à condition que $n \leq m$
Regroupements		
[abc]	$(a + b + c)$	a, b, c sont des caractères
[a-z]	$(a + b + c \ldots z)$	utilise l'ordre des caractères ASCII
[^a-z]	$\Sigma \backslash \{a, b, c\}$	n'inclut pas le symbole de fin de ligne \n
Ancres		
\<e	e	e doit apparaître en début de mot, ie. précédé d'un séparateur (espace, virgule...)
e\>	e	e doit apparaître en fin de mot, ie. suivi d'un séparateur (espace, virgule...)
^e	e	e doit apparaître en début de ligne
e$	e	e doit apparaître en fin de ligne
Caractères spéciaux		
\.	.	
*	*	
\+	+	
\n		dénote une fin de ligne
...	+	

Tab. 2.2 – Définition des motifs pour grep

```
> grep -E '([îâû]n?t|[îâû]mes|[iau]n?ss(e|es|ions|iez|ent))' Balzac.txt
```

Cette expression est un peu trop générale, puisqu'elle inclut des séquences comme unssiez; pour l'instant on s'en contentera. Pour continuer, revenons à notre ambition initiale : chercher des verbes. Il importe donc que les terminaisons que nous avons définies apparaissent bien comme des suffixes. Comment faire pour cela ? Imposer, par exemple, que ces séquences soient suivies par un caractère de ponctuation parmi : [,;.! :?]. On pourrait alors écrire :

```
> grep -E '([îâû]n?t|[îâû]mes|[iau]n?ss(e|es|ions|iez|ent))[,;.! :? ]' \
  Balzac.txt
```

indiquant que la terminaison verbale doit être suivie d'un des séparateurs. grep connaît même une notation un peu plus générale, utilisant : [:punct :], qui comprend toutes les ponctuations et [:space :], qui inclut tous les caractères d'espacement (blanc, tabulation...). Ce n'est pourtant pas cette notation que nous allons utiliser, mais la notation \>, qui est une notation pour εlorsque celui-ci est trouvé à la fin d'un mot. La condition que la terminaison est bien en fin de mot s'écrit alors :

```
> grep -E '([îâû]n?t|[îâû]mes|[iau]n?ss(e|es|ions|iez|ent))\>' Balzac.txt
```

Dernier problème : réduire le bruit. Notre formulation est en effet toujours excessivement laxiste, puisqu'elle reconnaît des mots comme `masse` ou `passions`, qui ne sont pas des formes de l'imparfait du subjonctif. Une solution exacte est ici hors de question : il faudrait rechercher dans un dictionnaire tous les mots susceptibles d'être improprement décrits par cette expression : c'est possible (un dictionnaire est après tout fini), mais trop fastidieux. Une approximation raisonnable est d'imposer que la terminaison apparaissent sur un radical comprenant au moins trois lettres, soit finalement (en ajoutant également \< qui spécifie un début de mot) :

```
> grep "\<[a-zéèêîôûç]{3,}([îâû]n?t|[îâû]mes|[iau]n?ss(e|es|ions|iez|ent))\>" \
  Balzac.txt
```

D'autres programmes disponibles sur les machines UNIX utilisent ce même type d'extensions notationnelles, avec toutefois des variantes mineures suivant les programmes : c'est le cas en particulier de `(f)lex`, un générateur d'analyseurs lexicaux ; de `sed`, un éditeur en batch ; de `perl`, un langage de script pour la manipulation de fichiers textes ; de `(x)emacs`... On se reportera aux pages de documentation de ces programmes pour une description précise des notations autorisées. Il existe également des bibliothèques permettant de manipuler des expressions rationnelles. Ainsi, pour ce qui concerne C, la bibliothèque `regexp` permet de «compiler» des expressions rationnelles et de les rechercher dans un fichier. Une bibliothèque équivalente existe en C++, en java...

Attention Une confusion fréquente à éviter : les shells UNIX utilisent des notations complètement différentes pour exprimer des ensembles de noms de fichiers. Ainsi, par exemple, la commande `ls nom*` liste tous les fichiers dont le nom est préfixé par `nom`; et pas du tout l'ensemble de tous les fichiers dont le nom appartient au langage $nom^\star$.