1.1 Expressions rationnelles

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 $L_1 \cup L_2$, $L_1 L_2$, 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) εet $\emptyset$ 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) εdénote 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}`	$(e^n + e^{n+1} \dots + e^m)$	à condition que $n \leq m$
Regroupements		
`[abc]`	$(a + b + c)$	a, b, c sont des caractères
`[a-z]`	$(a + b + c \dots z)$	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 $nom^\star$.

1.2 Automates finis

Introduction

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 symbole 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, reprise du cours de Master 1.

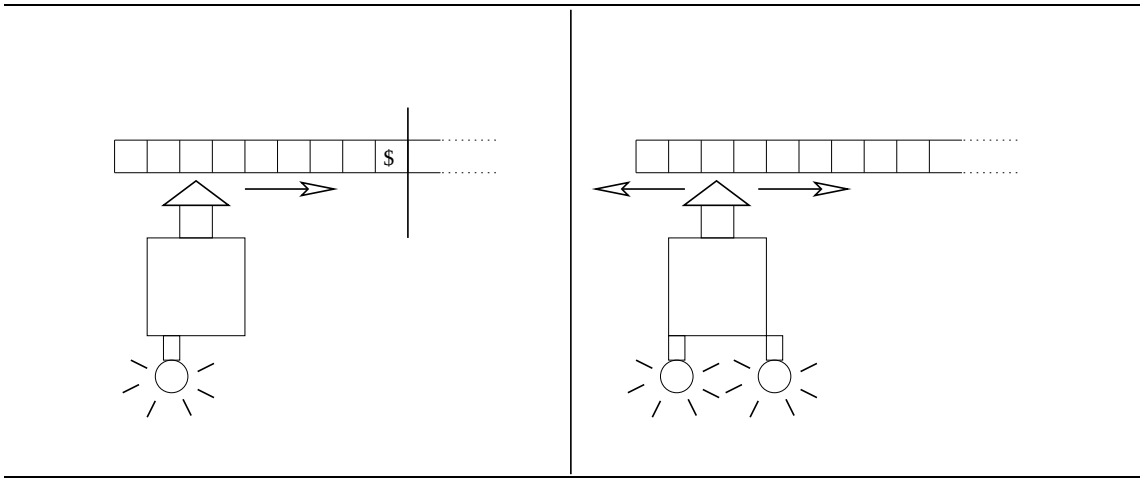


FIG. 1.1 – Comparaison graphique AFD/MT

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 un autre raison, et on décide que le mot est reconnu si l'état dans lequel se trouve la machine est un état dit final.

On peut distinguer plusieurs types d'automates, selon qu'ils sont complets ou non, déterministes ou non, etc. Ces différents types d'automates sont souvent équivalents, ce qui se démontre au moyen d'algorithmes, que nous verrons à la section 2.

Par ailleurs, les automates forment un domaine où peuvent être définies certaines opérations déjà connues sur les langages (union, etc). Ces opérations permettent de former de nouveaux automates, et là encore, c'est le point de vue algorithmique qui nous guidera, dans la section 3.

1.2.1 Définitions

Déf. 1 (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 $\delta(q, a) = r$.

Déf. 2 (Reconnaissance)

Un mot $a_1 a_2 ... a_n$ 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

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

Déf. 3 (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 :

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

Déf. 4 (Automate fini non déterministe - AFnD)

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

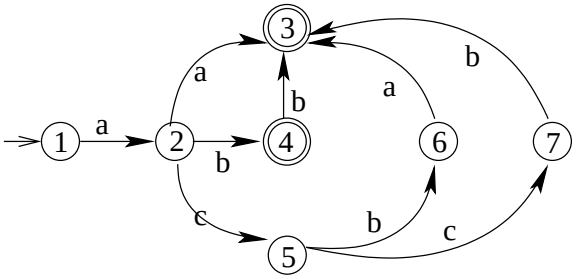
$$\mathcal{A} = \langle Q, \Sigma, q_0, F, \delta \rangle$$

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.

1.2.1.1 Commentaires et exemples

AFD Exemple

Langage $\{aa, ab, abb, acba, accb\}$



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

δ : (1,a) $\mapsto$ 2
(2,b) $\mapsto$ 4
(2,a) $\mapsto$ 3
...

Reconnaissance Différentes situations (avec un automate déterministe non (nécessairement) complet) :

- *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 ($acb|c$)).

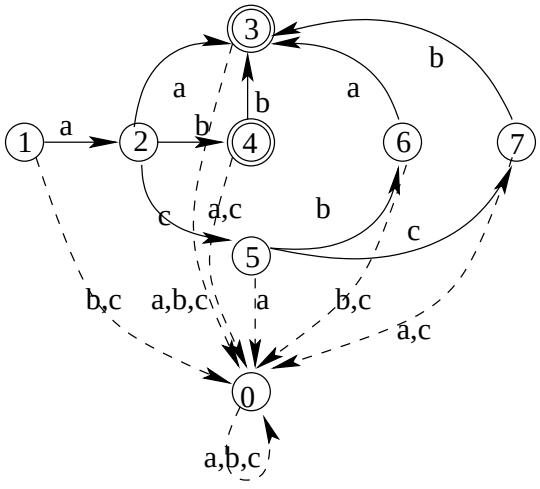
AFD complet
- C'est quelquefois ce qu'on appelle un AFD
- Il y a des automates finis déterministes complets sans puits

1.2.2 Transformations

1.2.2.1 Complétion

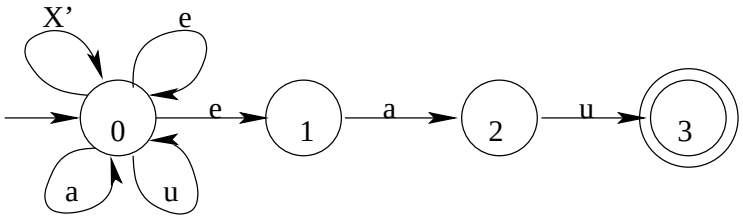
Comment rendre un automate complet ? Facile : on rajoute un « puits », ou état mort. Il suffit de boucher les trous de la table, et de rajouter des boucles sur le puits.

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



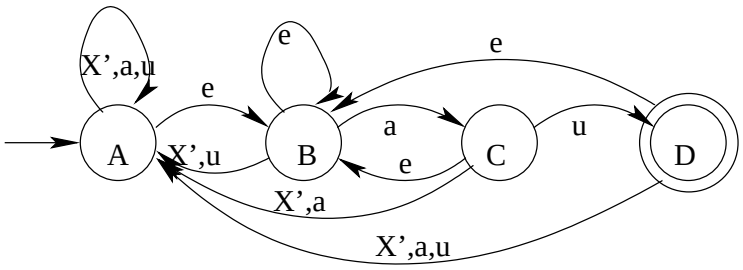
1.2.2.2 Déterminisation

Voici une illustration de l'algorithme sur un exemple : $.^*eau$



	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

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



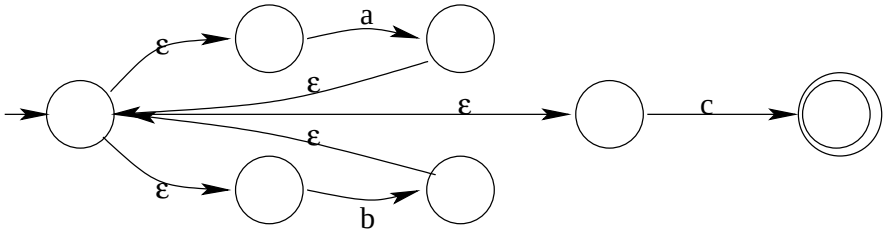
En exercice, faire la même chose à partir de l'automate complété. Conclusion : pour cet algo, ce n'est pas une bonne idée de compléter l'automate.

1.2.2.3 Elimination des ε-transitions

Parmi les automates non déterministes, on peut inclure (quelque fois, ils ne sont pas distingués) des *automates à ε-transitions*.

Même définition : $\langle X, Q, q_0, F, \delta\rangle$, mais δ change : $\delta : Q \times X \cup \{\varepsilon\} \to Q$

Exemple : $(a|b)^*c$



Reconnaissance : il peut y avoir des transitions par ε dans le chemin.

Elimination Étant donné un ε-automate $\langle X, Q, I, F, \delta\rangle$, on peut construire un automate non déterministe qui reconnaît le même langage.

Pour cela, on définit l'application ε^+ 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 q_i ∈ Q faire
        pour tous les x ∈ X faire
            δ'(q_i, x) := δ(q_i, x)
    pour tous les q_i tels que ε+(q_i) ≠ ∅ faire
        pour tous les q_j ∈ ε+(q_i) faire
            pour tous les x et q_r tels que q_r ∈ δ(q_j, x) faire
                δ'(q_i, x) := δ'(q_i, x) ∪ {q_r}
            si q_j ∈ F alors F' := F' ∪ {q_i}
Fin
```

Idée de l'algorithme : on enlève toutes les transitions alphabétiques, et on fait un calcul d'atteignabilité. On peut représenter cela sous forme d'une matrice.

Exemple :

	1	2	3	4	5	6	7
1	1	1		1		1	
2		1					
3	1	1	1	1		1	
4				1			
5	1	1		1	1	1	
6						1	
7							1

Algo avec la matrice (p. 631 Aho) : 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.

Exemple Soit l'automate reconnaissant $(a|b)^*c$ (déjà vu).

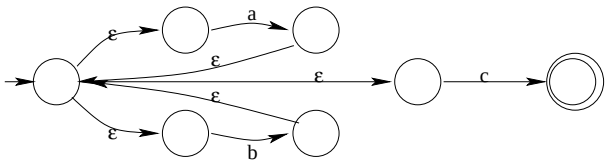


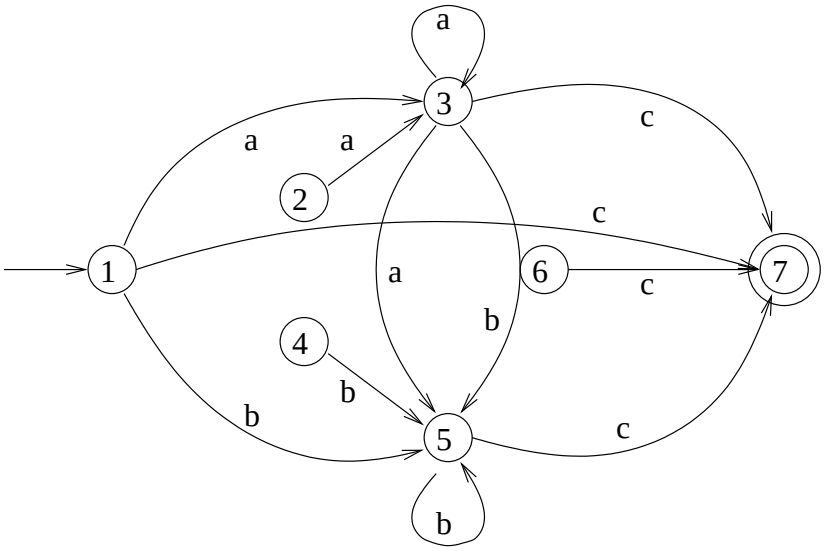
Table de transition initiale :

δ	ε	a	b	c
$\rightarrow$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
$\leftarrow$7	0	0	0	0

Table après calcul :

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

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



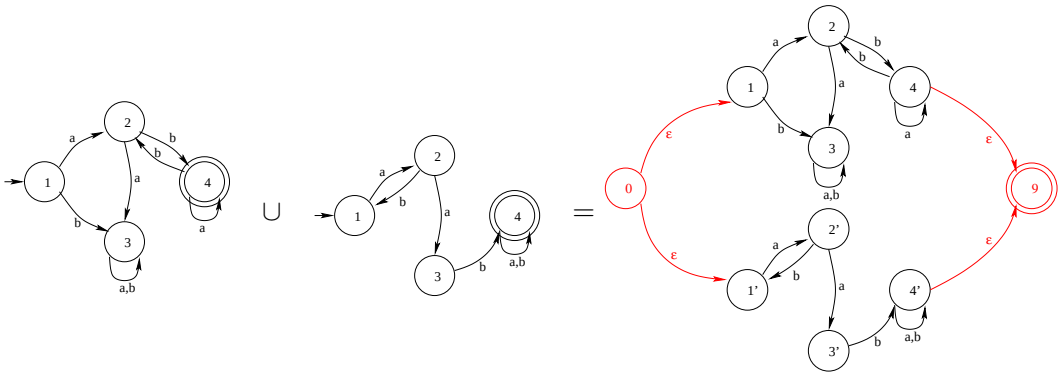
1.2.3 Propriétés de fermeture

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

1.2.3.1 Union

Pour réaliser une 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 un créer de nouveau.

Exemple :



Formulation mathématique : cf. la section "Théorème de Kleene".

1.2.3.2 Concaténation, étoile

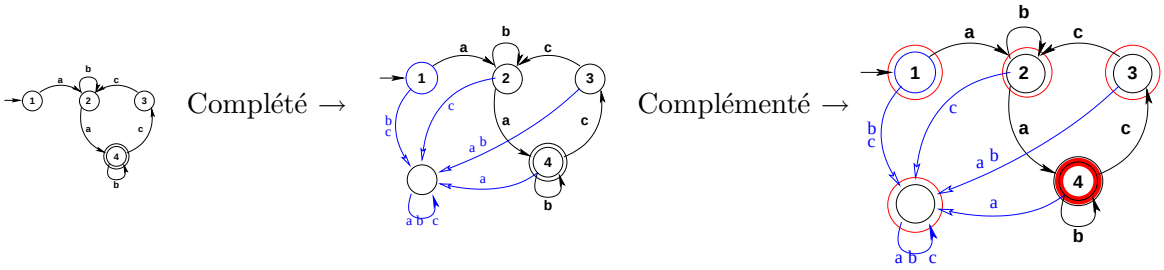
Il est facile d'imaginer, sur la même base que précédemment, comment créer un automate réalisant la concaténation de deux automates, ou l'étoile d'un automate. C'est en fait exactement ce que l'on fait dans l'algorithme de traduction d'une expression rationnelle en automate.

1.2.3.3 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 $Q \setminus F$).

1.2.3.4 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.

	a	b
→ 1	2	4
2	4	3
← 3	3	3
4	4	4

	a	b
↔ 1	2	5
2	5	3
3	4	5
4	1	4
5	5	5

	a	b
→ (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)
← (3,1)	(3,2)	(3,4)
(3,2)	(3,4)	(3,3)
(3,5)	(3,5)	(3,5)

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

1.3 Théorèmes d'équivalence

1.3.1 Le théorème triangulaire

On a établi (Kleene y a contribué) une ensemble de résultats d'équivalence que l'on peut résumer de la façon suivante[1]

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

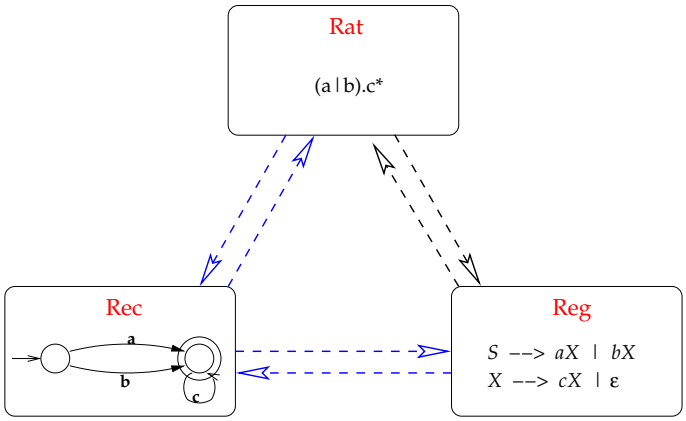
où

– $\mathcal{L}_{\text{Rec}}$ 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 ;

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

– $\mathcal{L}_{\text{Reg}}$ 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 1.2.

FIG. 1.2 – 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. (§ 1.3.2.2)

$\mathcal{L}_{\text{Reg}} \subset \mathcal{L}_{\text{Rec}}$ Algorithme très proche du précédent, toujours basé sur l'identité entre symbole non-terminal et état. (§ 1.3.2.3)

$\mathcal{L}_{\text{Rat}} \subset \mathcal{L}_{\text{Rec}}$ Algorithme basé sur la décomposition syntaxique de l'expression rationnelle, et la composition d'automates correspondants. (§ 1.3.3.1)

$\mathcal{L}_{\text{Rec}} \subset \mathcal{L}_{\text{Rat}}$ 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. (§ 1.3.3.2)

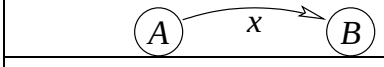
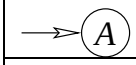
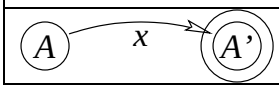
1.3.2 Grammaires et automates

1.3.2.1 Principe

Rappel une grammaire **régulière** (dite aussi linéaire) est une grammaire dont toutes les règles de production sont sous l'une des formes suivantes[2] : $A \to xB$
$A \to x$
$A \to \varepsilon$

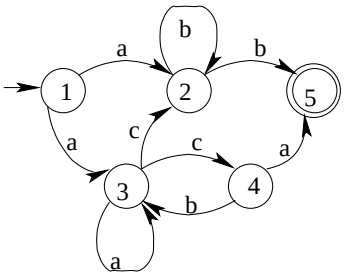
Note On peut toujours proposer une définition sans ε-production, ou plutôt sans autre ε-production qu'une règle $S \to \varepsilon$, où S est l'axiome, et S est inaccessible.

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 1.2 résume cette correspondance, qui donne les bases des algorithmes dans les deux sens.

	$A \to xB$
	Axiome $= A$
	$B \to \varepsilon$
 (A' nouvel état)	$A \to x$

Tab. 1.1 – Correspondances automate ↔ grammaire régulière

L'application de l'algorithme est illustrée à la figure 1.3 avec un automate non déterministe.



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

Fig. 1.3 – Exemple Rec → Reg (automate non déterministe)

[2]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 analogue possible à droite.

1.3.2.2 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 (Ⓐ —ε→ Ⓑ 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.

1.3.2.3 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 $A \to x$, 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 $A \to x$ est équivalente à une dérivation avec les deux règles $A \to xA'$, et $A' \to \varepsilon$.

1.3.3 Automates et expressions rationnelles

1.3.3.1 Expression rationnelle → Automate

On peut montrer (voir section "Propriétés de fermeture") 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 $\langle X, Q_R \cup \{Q_0\}, \{Q_0\}, \{Q_0\}, \delta'_R\rangle$[3],
   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 ($Q_S \cap Q_R = \emptyset$).

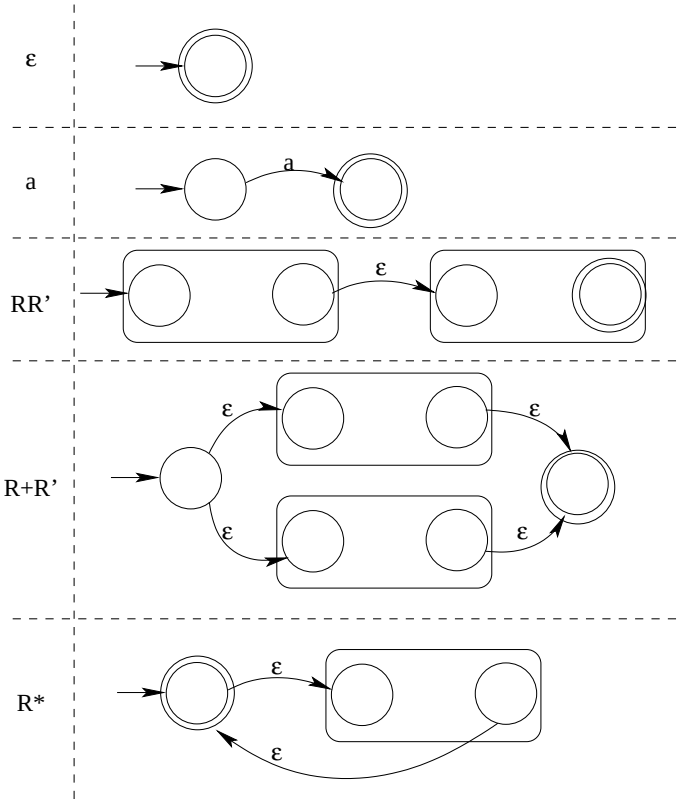
[3] Q_0 est un nouvel état t.q. $Q_0 \notin Q$.

(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

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



TAB. 1.2 – D'une expression rationnelle vers un automate

1.3.3.2 Automate → expression rationnelle

Il s'agit de l'algorithme le plus sophistiqué de la série présentée ici, 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.

Les automates généralisés que nous allons manipuler vérifient les contraintes suivantes :

- 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.

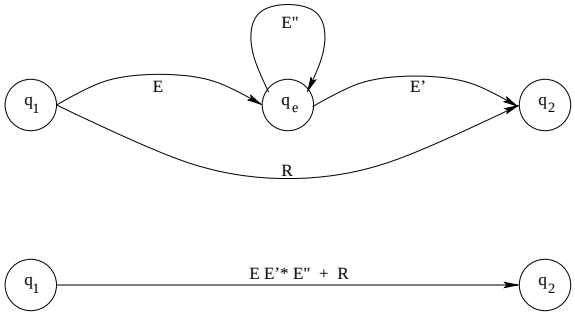
La première étape de l'algorithme, qui consiste à **transformer l'automate initial en automate généralisé**, revient à ajouter un état initial et un état final vérifiant les contraintes précédentes (reliés par des ε-transitions aux états initial et finaux de l'automate initial) ; puis à faire en sorte 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").

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 1.4.

FIG. 1.4 – 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, donc 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 1.5. À l'issue de la première étape, on passe à l'automate généralisé représenté à la figure 1.6 (pour rendre la figure plus lisible, on a représenté les arcs non passants par des traits pointillés).

FIG. 1.5 – McNaughton & Yamada, exemple. Automate initial

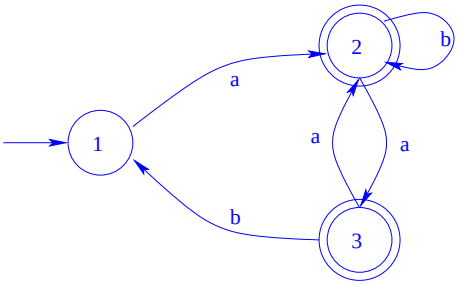
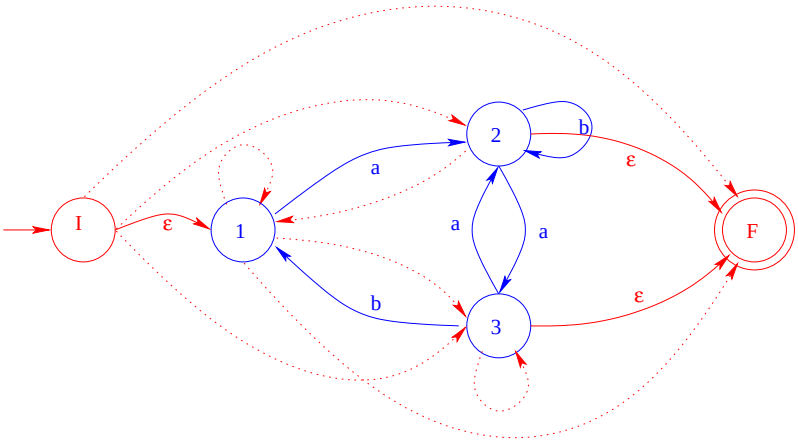


FIG. 1.6 – Exemple d'application de McNaughton & Yamada. Automate généralisé



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 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[4].

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 1.7.

On recommence alors pour l'état 2, le nombre de combinaisons à considérer est bien sûr beaucoup plus réduit.

L'automate résultant est représenté à la figure 1.8.

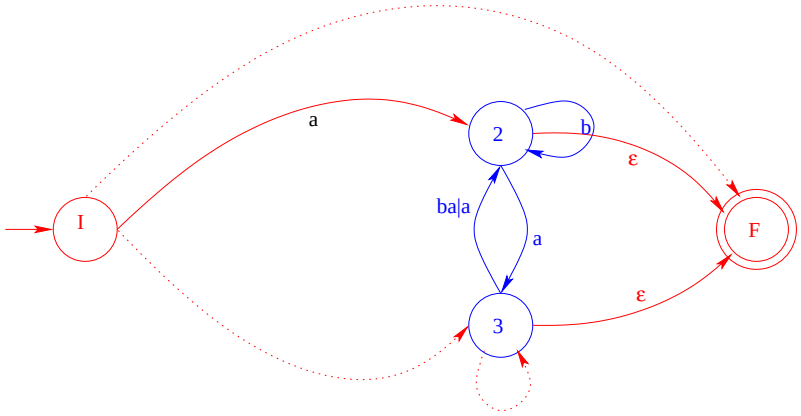
Il ne reste plus qu'à supprimer l'état 3, ce qui se fait en appliquant directement la règle

[4]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 $r\varepsilon = \varepsilon r = r$.

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 I	*pas à considérer, I n'a pas d'arcs entrants*		
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 I	*pas à considérer, I n'a pas d'arcs entrants*		
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

TAB. 1.3 – Les triplets impliquant l'état 1

FIG. 1.7 – McNaughton & Yamada, exemple. Automate généralisé après suppression de 1



illustrée plus haut. L'expression résultante est :

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

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	$ba\|a\ b^*$	$\varepsilon \mid \varepsilon$	$(ba\|a)b^*\|\varepsilon$

TAB. 1.4 – Les triplets à considérer impliquant l'état 2

FIG. 1.8 – McNaughton & Yamada, exemple. Automate généralisé après suppression de 2