INF 304 : Langages formels et compilation

Étienne KOUOKAM

Année académique 2016-2017

Table des matières

Préambule

OBJECTIFS VISES PAR LE COURS :

Les microprocesseurs disposent généralement d'un jeu d'instructions très sommaire, peu convivial et qui varie d'un processeur à l'autre. Afin de fournir à l'utilisateur de systèmes informatiques des formalismes universels, expressifs et concis, de nombreux langages de programmation ont été créés. L'objet de la théorie des langages est de les définir.

La compilation, pour sa part, à pour objet de transformer les programmes écrits dans ces langages en code machine. L'objectif de ce cours est de fournir à l'étudiant les connaissances conceptuelles qui lui permettront de mieux comprendre les langages informatiques ainsi que leur compilation.

PREREQUIS : Eléments d'algèbre.

OUVRAGES DE REFERENCE

1. J. E. Hopcroft, R. Motwani, and J. D. Ullman ; Introduction to Automata Theory, Languages, and Computation, Second Edition, Addison-Wesley, New York, 2001.
2. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, Compilers : Principles, Techniques and Tools", Addison-Wesley, 1986
3. Jean-Michel Autebert. Théorie des langages et des automates. Masson, 1994.

But du cours (séances de 3 heures)

Ce cours permet à l'étudiant :

– de savoir comment définir formellement un modèle pour décrire

  - un langage (de programmation ou autre)
  - un système (informatique)
- de savoir comment déduire des propriétés sur ce modèle
- de savoir ce qu'est
  - un compilateur
  - un outil de traitement de données
- d'avoir une idée sur la notion d'outil permettant de construire d'autres outils
  Exemple : générateur de (partie de) compilateur
- de savoir comment construire un compilateur ou un outil de traitement de données
  - en programmant
  - en utilisant ces outils
- d'avoir quelques notions de décidabilité
- d'écrire un programme **complexe** et **élégant** dans un langage de très haut niveau (C ou Objective Caml par exemple) ;
- de comprendre le **fossé** entre l'intention humaine et le langage de bas niveau exécuté par le microprocesseur ;
- de découvrir des **techniques** et **algorithmes** d'usage général : analyse lexicale, analyse syntaxique, transformation d'arbres de syntaxe abstraite, analyse de flot de données, allocation de registres par coloriage de graphe, etc.

Démarche

Ce cours montre en outre la démarche scientifique et de l'ingénieur qui consiste à

1. Comprendre les outils (mathématiques / informatiques) disponibles pour résoudre le problème
2. Apprendre à manipuler ces outils
3. Concevoir à partir de ces outils un système (informatique)
4. Implémenter ce système

Les outils ici sont :
- les formalismes pour définir un langage ou modéliser un système
- les générateurs de parties de compilateurs

Chapitre 1

Généralités

1.1 Historique et définition

Historiquement, la machine de Turing (dès 1936) a été conçu pour modéliser une machine qui résout le plus de problèmes possibles. Par la suite, ce fut le tour des automates (années 40 et 50) pour modéliser certaines fonctionnalités du cerveau. Les grammaires formelles (fin des années 50, Chomsky) feront leur apparition pour modéliser les langues naturelles. Actuellement :pour la compilation.

Un langage formel se définit comme un ensemble de suites de symboles. Chaque suite de symbole appartenant au langage représente un "objet". Ainsi, la solution à un problème ne peut être déterminée que si les éléments du langages sont bien connus.

Exemple :
- Quelles sont les couleurs existant dans un jeu de cartes ?
- Est-ce que cette suite de chiffres représente un nombre pair ?
- Est-ce que cette suite de chiffres représente un nombre premier ?

Les langages formels représentent uniquement l'aspect syntaxique des langages et non la sémantique qu'on pourrait leur associer (comme on le ferait avec les langues naturelles).

Dans le sens le plus usuel du terme, la compilation est une transformation que l'on fait subir à un programme décrit dans un langage évolué pour le rendre exécutable.
Fondamentalement, **la compilation** est une opération consistant en la traduction d'un programme écrit en langage source en un programme équivalent écrit en langage cible ou objet, comme le montre la figure 1.1. Par exemple, un texte écrit en Pascal, C, Java, etc., exprime un algorithme et il s'agit de produire un

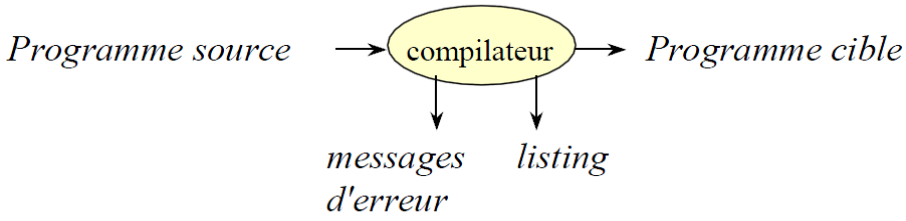


FIGURE 1.1 – Ce que fait un compilateur

autre texte, spécifiant le même algorithme dans le langage d'une machine que nous cherchons à programmer.
Le langage source n'est autre que celui qu'il faut analyser ; le langage objet, celui vers lequel il faut traduire ; ce dernier pouvant être le langage machine, l'assembleur ou un langage intermédiaire.

A ce stade, il importe de faire une distinction entre interprète, compilateur, décompilateur et traducteur .

Un interprète exécute directement un programme écrit en langage source comme le montre la figure 1.2.

Un compilateur traduit un langage source (généralement de haut niveau c'est-

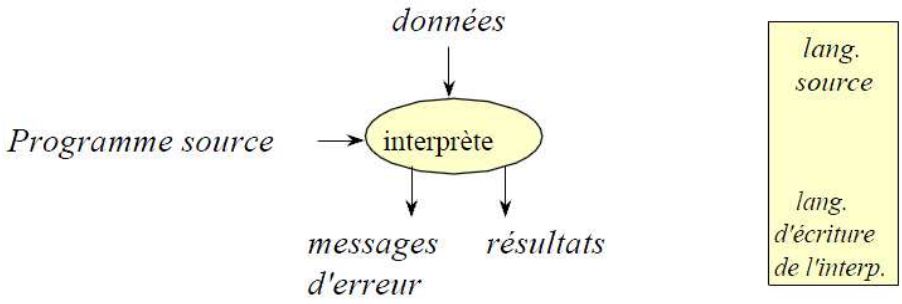


FIGURE 1.2 – Ce que fait un interprète

à-dire adapté à l'esprit humain) vers un langage de plus bas niveau (conçu pour être exécuté efficacement par une machine).
A l'opposée, **un décompilateur** est un programme qui essaye de reconstruire le programme source à partir du code objet.

Tout compilateur doit avoir les qualités suivantes :
- Correction : le programme cible est équivalent au programme source

- Rapidité (temps de compilation proportionnel à la taille du programme)
- Production de code de bonne qualité : les programmes produits doivent être efficaces en temps et en espace
- Bon diagnostic d'erreur
- Possibilité d'utiliser un debugger sur le code produit

Les traducteurs (compilateurs et interprètes) sont des programmes comme tous les autres, écrits dans un certain langage de programmation (de haut niveau si possible) à la différence (avec les autres programmes) qu'ils prennent des programmes comme données et produisent (cas des compilateurs) du code exécutable.
Au rang des traducteurs, on peut citer :

- le préprocesseur : d'un sur-langage de L vers L (C par exemple : # utilisé dans le include) ;
- l'assembleur : du langage d'assemblage vers le code machine binaire
- l'éditeur de liens : d'un ensemble de sous-programmes en binaire relogeable (les .o de linux par exemple) vers un programme exécutable

1.2 Types de compilateurs

On distingue les compilateurs monolithiques des compilateurs modulaires.

1.2.1 Les compilateurs monolithiques

Les tous premiers compilateurs étaient monolithiques : il pouvaient travailler en une seule passe, du fait que les langages d'alors étaient simples. Ainsi, on avait autant de compilateurs que de couples (langage source, machine cible), soit $m * n$ comme le montre la figure 1.3.

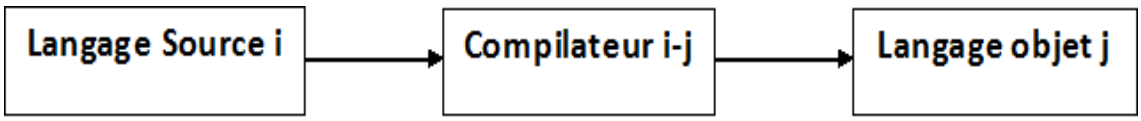


FIGURE 1.3 – Un compilateur monolithique

1.2.2 Les compilateurs modulaires

Les compilateurs modulaires, eux, ont général deux parties : une partie avant et une partie arrière.

Partie avant ou partie frontale

Cette partie permet d'analyser le code source. Elle et qui elle-même se charge de :

- l'analyse lexicale (A voir) : dans cette phase, les caractères isolés qui constituent le texte source sont regroupés pour former des unités lexicales, qui sont les mots du langage. L'analyse lexicale opère sous le contrôle de l'analyse syntaxique ; elle apparaît comme une sorte de fonction de lecture améliorée, qui fournit un mot lors de chaque appel.
- l'analyse syntaxique (A voir) : alors que l'analyse lexicale reconnaît les mots du langage, l'analyse syntaxique en reconnaît les phrases. Le rôle principal de cette phase est de dire si le texte source appartient au langage considéré, c'est-à-dire s'il est correct relativement à la grammaire de ce dernier.
- l'analyse sémantique (que nous ne verrons pas) : la structure du texte source étant correcte, il s'agit ici de vérifier certaines propriétés sémantiques, c'est-à-dire relatives à la signification de la phrase et de ses constituants :
  - les identificateurs apparaissant dans les expressions ont-ils été déclarés ?
  - les opérandes ont-ils les types requis par les opérateurs ?
  - les opérandes sont-ils compatibles ? n'y a-t-il pas des conversions à insérer ?
  - les arguments des appels de fonctions ont-ils le nombre et le type requis ?
  - etc.

Partie arrière ou partie finale

C'est cette partie qui fait la synthèse en générant le programme cible. Elle se charge de :

- Optimisation du code : il s'agit généralement ici de transformer le code afin que le programme résultant s'exécute plus rapidement. Par exemple
  - détecter l'inutilité de recalculer des expressions dont la valeur est déjà connue,
  - transporter à l'extérieur des boucles des expressions et sous-expressions dont les opérandes ont la même valeur à toutes les itérations
  - détecter, et supprimer, les expressions inutiles
- la Génération de code intermédiaire (que nous ne verrons pas non plus cours de M1) : Après les phases d'analyse, certains compilateurs ne produisent pas directement le code attendu en sortie, mais une représentation intermédiaire, une sorte de code pour une machine abstraite. Cela permet de concevoir indépendamment les premières phases du compilateur (constituant ce que l'on appelle sa face avant) qui ne dépendent que du langage source compilé et les dernières phases (formant sa face arrière) qui ne dépendent que du langage cible ; l'idéal serait d'avoir plusieurs faces avant et

plusieurs faces arrière qu'on pourrait assembler librement [1]

– Génération du code final Cette phase, la plus impressionnante pour le néophyte, n'est pas forcément la plus difficile à réaliser. Elle nécessite la connaissance de la machine cible (réelle, virtuelle ou abstraite), et notamment de ses possibilités en matière de registres, piles, etc.

Une telle présentation de la compilation est très schématique (en particulier, les analyses ne sont pas indépendantes l'une de l'autre), mais il faut savoir que les outils qui sont à notre disposition (les théories développées dans le cours ainsi que LEX et YACC, qui en appliquent une grande partie) permettent effectivement de construire des compilateurs pour des langages non triviaux.
La figure 1.4 présente les différentes parties d'un tel compilateur.

Cette décomposition présente des avantages : m parties avant + n parties ar-

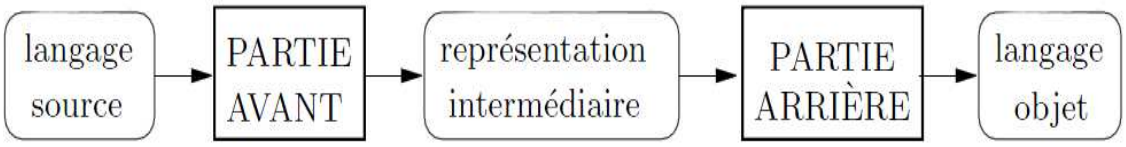


FIGURE 1.4 – Un compilateur modulaire

rières permettent d'avoir $m * n$ compilateurs conformément à la figure 1.5. Par ailleurs, le langage objet peut être celui d'une machine virtuelle (JVM . . .) : le 1.3. 1.1. programme résultant sera portable. Par ailleurs, on peut interpréter la

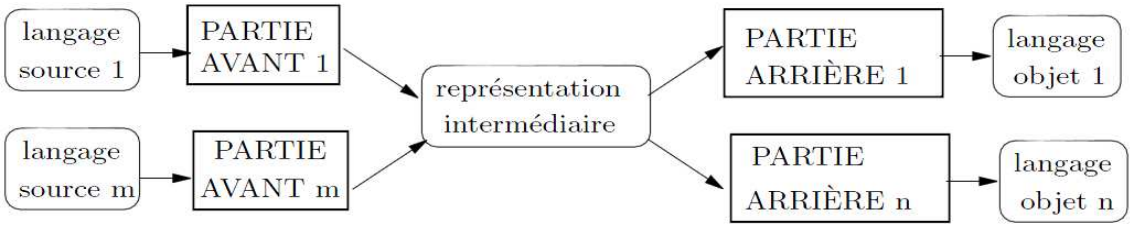


FIGURE 1.5 – Avantages d'un compilateur modulaire

représentation intermédiaire

1. De la sorte, avec n faces avant pour n langages source et m faces arrière correspondant à m machines cibles, on disposerait automatiquement de $n * m$ compilateurs distincts. Mais cela reste, pour le moment, un fantasme d'informaticien.

1.3 Architecture de la partie avant d'un compilateur

Nous présentons à la figure 1.6 le schéma synthétique de la partie avant d'un compilateur. 1.1. Cette architecture de la partie avant du compilateur se justifie :

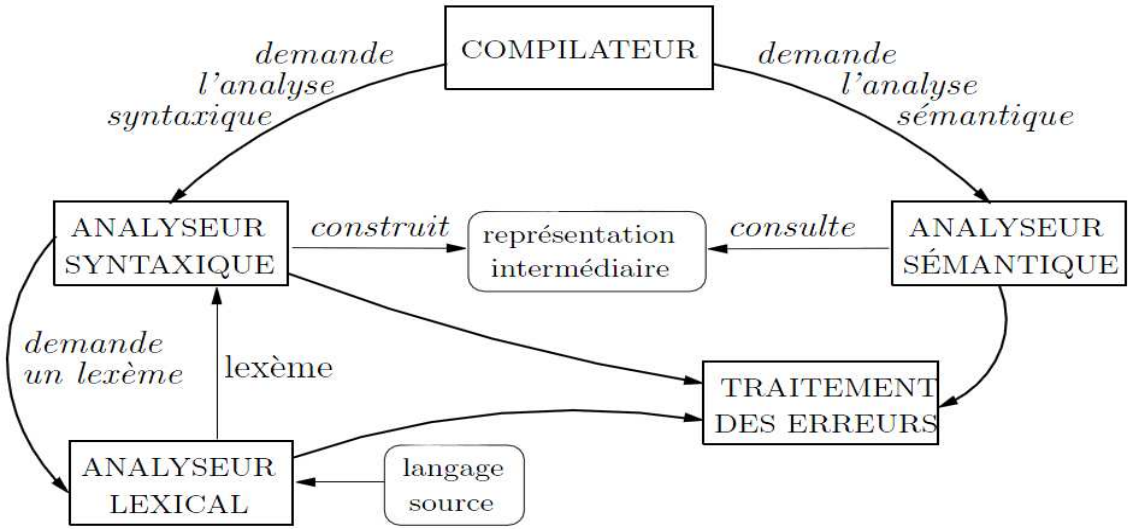


FIGURE 1.6 – Schéma de la partie avant d'un compilateur

- du point de vue lexico-syntaxique car
  - le programme source est parfois plein de "bruits" (espaces inutiles, commentaires . . .)
  - pour la grammaire, de nombreux symboles sont équivalents (identificateurs, nombres . . .)
  - ce qui justifie le pré-traitement (en général au vol) du texte source
- du point de vue sémantique :
  - existence de références en avant (utilisation d'un identificateur avant sa déclaration par exemple)
  - unification du traitement de constructions équivalentes (attr=0 et this.attr=0 par exemple) ou proches (boucles notamment)
  - ce qui justifie la mémorisation (sous forme intermédiaire) du texte à compiler

Dans ce cours, nous allons prioritairement nous intéresser à :

- l'analyse lexicale : langages réguliers, expressions régulières, automates finis pour l'essentiel
- l'analyse syntaxique : grammaires hors-contexte, automates à pile (analyseurs descendants ou ascendants)

Chapitre 2

Les langages

Un mot autrement appelé **une chaîne**, est une chaîne de caractères et un **langage** est un ensemble de mots.
Dans ce chapitre, on étudie des propriétés générales des langages.

Deux types de langages seront étudiés par la suite. Leur application à la compilation donne lieu à deux types d'analyse :
- Les langages réguliers : l'analyse lexicale,
- Les langages algébriques : l'analyse syntaxique.

Nous considérons dans ce cours des listes finies de symboles, et des ensembles homogènes de telles listes : d'une façon générale une liste finie de symboles s'appellera un mot et un ensemble de mots s'appellera un langage. Cependant, ce vocabulaire n'est pas adapté à toutes les situations :

2.1 Les mots

Un **alphabet** est un ensemble Σ, dont les éléments sont appelés lettres, caractères ou symboles.

Exemple 1 *{0,1}, {A,C,G,T}, l'ensemble de toutes les lettres, l'ensemble des chiffres, le code ASCII, etc.*

On notera que les caractères blancs (c'est-à-dire les espaces, les tabulations et les marques de fin de ligne) ne font généralement pas partie des alphabets [1].
On appelle ***mot*** toute suite finie (éventuellement vide) d'éléments de Σ. Ainsi, un mot u sur l'alphabet Σ est une application

$$u : \{1, \cdots, m\} \to \Sigma$$

1. Il en découle que les unités lexicales, sauf mesures particulières (apostrophes, guillemets, etc.), ne peuvent pas contenir des caractères blancs. D'autre part, la plupart des langages autorisent les caractères blancs entre les unités lexicales.

où :

m est un entier appelé la longueur de $|u|$

$\{1, \cdots, m\}$ est l'ensemble des entiers naturels i tels que $1 \leq i \leq m$.

- $u(i)$ est appelée la i$^{\text{ème}}$ lettre, le i$^{\text{ème}}$ caractère ou le i$^{\text{ème}}$ symbole de u.
- Si $u(i) = x$, on dira que u(i) est une occurrence de x dans u.
- On peut noter $u[i]$ au lieu de $u(i)$: on constate alors que la définition des mots nous est très familière.

L'ensemble de tous les mots formés à partir de l'alphabet Σ (resp. de tous les mots non-vides) est désigné par Σ^* (resp. Σ^+).

La k^{ème} puissance d'un alphabet Σ notée Σ^k se définit par $\Sigma^0 = \{\epsilon\}$ et $\forall k > 0, \Sigma^k = \Sigma^{k-1}\Sigma^1$, i-e l'ensemble de tous les mots de longueur k que l'on peut former à partir de Σ.

Exemple 2 *Si* $\Sigma = \{a, b\}$, *alors* $\Sigma^0 = \{\epsilon\}$, $\Sigma^1 = \{a, b\}$, $\Sigma^2 = \{aa, ab, ba, bb\}$, *etc.*

Σ^* est donc l'ensemble de tous les mots qu'on peut former à partir de Σ. On note Σ^+ le même ensemble mais privé du mot vide : $\Sigma^* = \Sigma^+ \cup \{\epsilon\}$

2.1.1 Définitions de base

Soit $u \in \Sigma^*$. La longueur d'un mot u, notée $|u|$, correspond au nombre total de symboles de u (chaque symbole étant compté autant de fois qu'il apparaît).

- Lorsque $|u| = 0, u : \emptyset \to \Sigma$ est le mot sans caractère (mot vide). Par convention, le mot vide est noté ϵ ; certains auteurs le notent 1, voire 1_Σ.
- Lorsque $|u| = 1, u : \{1\} \to \Sigma$ est défini par le seul caractère $u(1) \in \Sigma$

On convient d'identifier tout $x \in \Sigma$ au mot de longueur 1 qu'il définit. Par conséquent, $\Sigma \subset \Sigma^*$. Les identifications ne sont valables que parce que les éléments de A sont reconnaissables pour tels.
Autre notation utile, $|u|_a$ compte le nombre total d'occurrences du symbole a dans le mot u. On a naturellement : $|u| = \Sigma_{a \in \Sigma} |u|_a$.

2.1.2 Récurrence sur les mots basés sur l'adjonction d'une occurrence à droite

L'adjonction d'une occurrence à droite d'un mot $\Sigma^* \times \Sigma \to \Sigma$ se définit de la façon suivante :
pour tout mot $u : \{1, \cdots, m\} \to \Sigma$ et tout symbole $x \in \Sigma$,
$ux : \{1, \cdots, m+1\} \to \Sigma$ est le mot défini par :

$ux(i) = u(i)$ pour tout $i \in \{1, \cdots, m\}$
$ux(m+1) = x)$

Remarque 1:
e

$\epsilon x = x$ si, comme il a été convenu ci-dessus, on identifie le symbole x avec le mot de longueur 1 qu'il définit.

$|ux| = |u| + 1$.

Ainsi, tout élément de Σ^* ou bien est ϵ ou bien s'obtient à partir d'un élément de Σ par "adjonction" d'un caractère à droite. Les deux clauses suivantes :

– $\epsilon \in \Sigma^*$ (le mot sans caractère)

– $\forall x \in \Sigma^*$: si $u \in \Sigma^*$ alors $ux \in \Sigma^*$ (adjonction de x à droite de u).

constituent donc une définition inductive de Σ^*.
Plus précisément, Σ^* est le plus petit ensemble vérifiant les deux propriétés ci-dessus.
De plus, chaque élément de Σ^* admet une construction unique à partir de ϵ par adjonctions successives de caractères à droite.

Exemple 3 *Voir le tableau 2.1*

Construction	**Résultat**	**En abrégé**
Mot initial	ϵ	ϵ
Adjonction de a	ϵa	a
Adjonction de a	ϵaa	aa
Adjonction de b	ϵaab	aab

TABLE 2.1 – Adjonction d'occurrence à droite

En tenant compte de la remarque ci-dessus, il est clair que la mention du mot sans caractère ϵ n'est indispensable que dans l'étape initiale. On convient tout naturellement de ne plus l'écrire dès qu'un mot comporte au moins une occurrence d'un symbole.

On est conduit à la représentation naturelle suivante :
$u : \{1, \cdots, m\} \to \Sigma$ est représenté par la juxtaposition de ses caractères successifs, c'est-à-dire par $u(1) \dots u(m)$.
Lorsqu'elle nécessite l'usage de pointillés (c'est-à-dire lorsque l'on parle d'un mot "en général") cette représentation est une illustration qui permet d'avoir une intuition qui est souvent utile, mais pas de faire des raisonnements concrets, traduisibles en algorithmes ! Elle est cependant parfaitement correcte et utile lorsqu'on a affaire à des mots connus explicitement

Exemple 4 *Le mot* $u : \{1,2,3,4\} \to \{a,b\}$ *tel que* $u(1) = u(2) = u(4) = a$ *et* $u(3) = b$ *sera noté* $u = aaba$.

Lorsque l'on écrit les mots par simple juxtaposition de caractères, il faut éviter que ceux-ci interagissent les uns avec les autres : chaque caractère doit être indécomposable en des éléments appartenant à l'alphabet en cause.
La condition que Σ doit satisfaire pour cela s'exprime sous la forme suivante :

Soit P[u] un énoncé au sujet de $u \in \Sigma^*$. Alors, si les deux propriétés suivantes sont vraies :
- $P[\epsilon]$
- $\forall u \in \Sigma^*$ et tout $x \in \Sigma$, $P[u] \Rightarrow P[ux]$

on peut conclure que $P[u]$ est vraie pour tout $u \in \Sigma^*$

Les notions de base relatives aux mots peuvent se définir par récurrence sur les mots : on doit comprendre une telle définition comme le schéma d'une procédure récursive. En fait, la plupart des définitions relatives aux mots seront basées sur ce principe de récurrence, ou l'une de ses variantes (cf. ci-dessous).

Exemple 5 *La longueur d'un mot peut se redéfinir utilement par récurrence de la façon suivante :*
$|\epsilon| = 0$
$|ux| = |u| + 1$ *pour tout* $u \in \Sigma^*$ *et tout* $x \in \Sigma$
L'énoncé P[u] qui est en cause ici est "u est définie".

Remarque 2:
Dans ce qui précède, nous avons supposé implicitement que les symboles (par exemple a et b) étaient des objets insécables et que l'on pouvait placer "côte à côte" sans qu'ils risquent de se mélanger : ceci est nécessaire en particulier lorsque l'on a besoin de connaître le symbole qui a été adjoint le dernier dans la construction d'un mot.
Dans la pratique, on utilise souvent des symboles qui se présentent déjà sous la forme de "chaînes de caractères" : pour pouvoir les reconnaître, il faut alors user d'artifices : ces artifices sont les séparateurs (parenthèses, caractère blanc, etc.).

Exemple 6 *Si l'on veut absolument utiliser l'ensemble aba, ab, bb, b comme un alphabet, on ne peut pas se contenter de juxtaposer des occurrences de ses éléments pour en faire des mots car, par exemple le "mot" ababb peut se lire de trois façons différentes ! On peut résoudre ce problème en intercallant des blancs (par exemple aba bb), ou bien, et c'est la solution que nous adopterons le plus souvent, en considérant l'alphabet [aba], [ab], [bb], [b] où chacun des symboles est soigneusement encapsulé : ainsi, les trois lectures précédentes correspondent-elles à trois mots bien identifiables : [aba][bb], [aba][b][b] et [ab][ab][b].*

2.1.3 La concaténation

La concaténation est l'opération "." : $\Sigma^* \times \Sigma^* \to \Sigma^*$ qui consiste à mettre deux mots "bout à bout". Ainsi, pour tout $u \in \Sigma^*$ et tout $v \in \Sigma^*, u.v$ est définie par récurrence sur v de la façon suivante :

1. $u.\epsilon = u$
2. $u.(vx) = (u.v)x$ pour tout $v \in \Sigma^*$ et tout $x \in \Sigma$

L'énoncé P[v] en cause ici est " pour tout $u \in \Sigma^* u.v$ est défini"

La concaténation est une opération interne de Σ^* ; elle est associative, mais pas commutative (sauf dans le cas dégénéré où Σ ne contient qu'un seul symbole). ϵ est l'élément neutre pour la concaténation : $u\epsilon = \epsilon u = u$; ceci justifie la notation 1 ou encore 1_Σ.

Un ***sous-mot*** v est une sous-suite de la suite des symboles d'un mot w.
Le facteur v d'un mot w est un mot tel qu'il existe des mots u et u' tels que w = u.v.u'.
Un facteur gauche (ou préfixe) est un facteur tel que $u = \epsilon$.
Un facteur droit (ou suffixe) est un facteur tel que $u' = \epsilon$.
Une occurence d'un facteur est l'apparition de ce facteur a une position précise dans le mot.
Un mot fini u est **périodique** si $u = x^n$ pour $n \geq 2$. Tout mot non périodique est dit **primitif**

Conventionnellement, on notera u^n la concaténation de n copies de u, avec bien sûr $u^0 = \epsilon$. Si u se factorise sous la forme u = xy, alors on écrira parfois $y = x^{-1}u$ et $x = uy^{-1}$.

Si w se factorise en $u_1 v_1 u_2 v_2 \dots u_n v_n u_{n+1}$ où tous les u_i et v_i sont des mots de Σ^*, alors $v = v_1 \dots v_n$ est un sous-mot [2] de w.

Contrairement aux facteurs, les sous-mots sont donc construits à partir de fragments non nécessairement contigus, mais dans lesquels l'ordre d'apparition des symboles est toutefois respecté. On appelle ***facteur (resp. préfixe, suffixe, sous-mot) propre de u*** tout facteur (resp. préfixe, suffixe, sous-mot) de u différent de u.

Nous écrirons très souvent uv au lieu de u.v. De même, l'associativité de la concaténation permet de faire abstraction des parenthèses et d'écrire uvw au lieu de (u.v).w ou u.(v.w). Nous ferons de même par la suite pour toutes les opérations

2. Attention : il y a désaccord entre les terminologies françaises et anglaises : subword signifie en fait facteur et c'est scattered subword qui est l'équivalent anglais de notre sous-mot.

associatives (après avoir vérifié qu'elles le sont !).
Remarque 3:
Σ^*, muni de l'opération de concaténation, possède donc une structure de ***monoïde*** (c'est-à-dire un ensemble muni d'une opération interne associative et d'un élément neutre ; lorsqu'il n'y a pas d'élément neutre on parle de semi-groupe). Ce monoïde est le monoïde libre engendré par Σ : tout mot u se décompose de manière unique comme concaténation de symboles de Σ.

Il existe un nombre dénombrable de mots dans Σ^*, mais le nombre de langages dans Σ^* est indénombrable. Parmi ceux-ci, tous ne sont pas à la portée des informaticiens : il existe, en effet, des langages qui "résistent" à tout calcul, c'est-à-dire, plus précisément, qui ne peuvent pas être énumérés par un algorithme.

2.2 Langages et opérations élémentaires sur les langages

Un langage L se définit à partir d'un alphabet Σ comme étant un sous-ensemble de Σ^*.
L'ensemble des langages sur Σ est désigné par $\mathcal{P}(\ \Sigma^*)$, l'ensemble des parties de Σ^* souvent noté 2^{Σ^*}.

Exemple 7 $\emptyset \subseteq \Sigma^*, \Sigma \subseteq \Sigma^*, \Sigma^* \subseteq \Sigma^*$ *sont des langages sur* Σ.
Pour tout $u \in \Sigma^*, \{u\} \subseteq \Sigma^*$ *est un langage sur* Σ

Il est intéressant d'identifier un mot au langage à un seul élément qu'il définit. Cette identification se traduit par une convention d'écriture (abus de notation) que l'on peut symboliser par $\Sigma^* \subseteq \mathcal{P}(\ \Sigma^*)$.
Le regroupement de nos deux conventions d'écriture donne $\Sigma \subseteq \Sigma^* \subseteq \mathcal{P}(\ \Sigma^*)$. C'est généralement la notation la plus simple qui sera utilisée, par exemple :

- Si $x \in \Sigma$, x désignera aussi le langage $\{x\}$ dont le seul mot ne comporte que le seul caractère x.
- ϵ désignera aussi le langage $\{\epsilon\}$ dont le seul mot est le mot sans caractère.

En général, on définira un langage en utilisant une notation ensembliste de la forme $w|\ propritsurw$, qui signifie "l'ensemble des mots w (de Σ^*) tels qu'ils respectent une certaine propriété".

Exemple 8 $\{w|w$ *se termine par a*$\}$ *ou encore* $\{w|2 \leq |w| \leq 5\}$.

L'ensemble des langages $P(\Sigma^*)$ étant l'ensemble des parties d'un ensemble, est naturellement muni d'opérations ensemblistes "classiques" qui leur sont applicables. bien connues : réunion, intersection, complémentaire. ***La réunion, aussi notée additivement,*** est souvent appelé la ***somme*** : il s'agit ici de la réutilisation d'un symbole connu, à des fins nouvelles.

2.2.1 Réunion ou somme de langages

Soient L_1 et L_2 des langages de Σ^*. La réunion de L_1 et L_2 est définie par :

$$L_1 \cup L_2 = L_1 + L_2 = \{u \in \Sigma^* | u \in L_1 \text{ ou } u \in L_2\}$$

La somme

$\forall L_1, L_2, L_3$ langages de Σ^* :

$$(L_1 + L_2 \subseteq L_3) \Rightarrow (L_1 \subseteq L_3 \text{ et } L_2 \subseteq L_3)$$

Exercice 1 : Démontrer ce résultat

Les autres propriétés les plus connues de la réunion se traduisent immédiatement dans la notation additive récapitulée dans le tableau 2.2

Propriété de la somme

Règle	**Réunion**
Neutralité de $\emptyset$	$\emptyset + L = L + \emptyset = L$
Associativité	$L_1 + (L_2 + L_3) = (L_1 + L_2) + L_3$
Commutativité	$L_1 + L_2 = L_2 + L_1$
Idempotence	L + L = L
Croissance	$(L_1 \subseteq L_2) \Rightarrow (L + L_1 \subseteq L + L_2)$

TABLE 2.2 – Propriétés de la réunion de 2 langages

Remarque 4:
toutes les propriétés de l'addition ordinaire ne sont pas vraies ici, en particulier, on ne peut pas "simplifier". Plus précisément, on peut avoir $L+L_1 = L+L_2$ alors que $L_1 \neq L_2$. En effet, si par exemple on a $L_1 \subseteq L$ et $L_2 \subseteq L$, on a $L + L_1 = L$ et $L + L_2 = L$ même si $L_1 \neq L_2$.

A partir de la somme, on peut définir une opération de différence ensembliste par

$$L_1 - L_2 = \{u | u \in L_1 \text{ et } u \notin L_2\}$$

Cependant, il faut ne jamais oublier que cette opération n'est pas la réciproque de la somme. En effet :

- $(L_1 - L_2) + L_2 = L_1 + L_2$, ce qui n'est égal à L_1 que si $L_2 \subseteq L_1$
- $(L_1 + L_2) - L_2 = L_1 - L_2$, ce qui n'est égal à L_1 que si $L_1 \cap L_2 = \emptyset$

2.2.2 Concaténation des langages

Soient L_1 et L_2 des langages de Σ^*. La concaténation de L_1 et L_2 est définie par :

Concaténation des langages

$$L_1.L_2 = \{u \in \Sigma^* | \exists v \in L_1 \text{ et } w \in L_2 \text{ tels que } u = v.w\}$$

Les propriétés les plus connues de la concaténation se traduisent immédiatement dans la notation multiplicative récapitulée dans le tableau 2.3

Propriétés de la concaténation

Règle	**Réunion**
Neutralité	$\epsilon.L = L.\epsilon = L$
Associativité	$L_1.(L_2.L_3) = (L_1.L_2).L_3$
Croissance	$(L_1 \subseteq L_2) \Rightarrow (L.L_1 \subseteq L.L_2 \text{ et } L_1.L \subseteq L_2.L)$
Nullité	$L.\emptyset = \emptyset.L = \emptyset$
Distributivité	$L_1.(L_2 + L_3) = L_1.L_2 + L_1.L_3$ et $(L_1 + L_2).L_3 = L_1.L_3 + L_2.L_3$

TABLE 2.3 – Propriétés de la concaténation de 2 langages

Par ailleurs, on a :

$$|L_1.L_2| \leq |L_1|.|L_2|$$

2.2.3 Itération et quotient de langages

Nous allons voir dans cette section que la construction de l'ensemble Σ^* des mots que l'on peut écrire à l'aide des éléments du langage particulier Σ peut s'étendre à tout langage $L \subseteq \Sigma^*$ pour construire le langage itéré de L (aussi appelé la fermeture de Kleene de L), que l'on notera L^*.
Dans la définition de Σ^*, par "adjonction de caractères à droite", il est facile de compter les étapes (le nombre de fois que l'on a adjoint un caractère) : l'ensemble Σ^i des mots que l'on obtient en i étapes se définit par une récurrence sur l'entier i

- $\Sigma^0 = \epsilon$ (un mot sans caractère)
- $\Sigma^{i+1} = \Sigma^i\Sigma, \forall i$ (adjonction d'un caractère à droite)

Σ^* est l'ensemble des mots ainsi construits : $\Sigma^* = \sum_{i\geq 0} \Sigma^i$.
Si dans la construction, on remplace "caractère", c'est-à-dire "élément de Σ par "élément de L", on obtient la définition de L^* qui suit :

Le langage itéré et quotient d'un langage

Soit $L \subseteq \Sigma^*$.
La puissance $L^i \subseteq \Sigma^*$ ***de L*** est définie pour tout entier naturel i par la récurrence :

- $L^0 = \epsilon$
- $L^{i+1} = L^i L, \forall i$

Le langage itéré de L est le langage $L^* \subseteq \Sigma^*$ défini par $L^* = \sum_{i\geq 0} L^i$.
Le quotient d'un langage L_1 ***par un langage*** L_2 est l'ensemble défini comme suit :

$$L_2^{-1}.L_1 = \{v \in \Sigma^* | \exists u \in L_2, u.v \in L_1\}$$

Les puissances de langages ont les propriétés suivantes :

Propriétés des puissances

1. $L^1 = L$
2. $L^i L^j = L^{i+j}$
3. $L^i L = L L^i = L^{i+1}$
4. $(\epsilon + L)^k = \sum_{0\leq i\leq k} L^i$

Soient L, L_1, L_2, L_3 des langages définis sur Σ.

Les itérations ont les propriétés suivantes :

Propriétés des itérations

1. Stabilité par concaténation : Si $L_1 \subseteq L^*$ et $L_2 \subseteq L^*$ alors $L_1 L_2 \subseteq L^*$
2. Stabilité par itération : Si $L_1 \subseteq L^*$ alors $L_1^* \subseteq L^*$
3. Croissance : Si $L_1 \subseteq L$ alors $L_1^* \subseteq L^*$
4. $\emptyset^* = \epsilon$ et $\epsilon^* = \epsilon$
5. $L^* L = L L^*$
6. $L^* L^* = (L^*)^* = L^*$
7. $L^* = \epsilon + L^* L = \epsilon + L(L^*)$
8. $(L_1 + L_2)^* = (L_1^* + L_2^*)^* = (L_1^*.L_2^*)^* = L_1^*(L_1.L_2^*)^* = (L_1^* L_2)^*.L_1^*$

2.3 Système d'équations linéaires en langages

La notation multiplicative pour la concaténation et additive pour la réunion nous permet d'écrire très naturellement des équations de type algébrique dans l'ensemble des langages sur un alphabet Σ et de tenter leur résolution. Dans un premier temps, nous étudierons le cas d'une "équation linéaire à une inconnue", puis nous verrons rapidement une méthode simple applicable aux "systèmes de n équations linéaires à n inconnues".

2.3.1 Equations linéaires à une inconnue (lemme d'Arden)

Lemme d'Arden

Soient $A \subseteq \Sigma^*$ et $B \subseteq \Sigma^*$ deux langages formels. Nous appellerons solution de l'équation :

$$(E) \quad X = AX + B \tag{2.1}$$

tout langage L contenu dans Σ^* vérifiant la relation $L = AL + B$.

Résolution de l'équation (2.1)

1. $L = A^*B$ est une solution de (2.1)
2. L est la plus petite solution de (2.1), c'est-à-dire que si $M \subseteq \Sigma^*$ vérifie $M = AM + B$, alors $A^*B \subseteq M$.
3. Si $\epsilon \notin A$, alors (2.1) admet une solution unique

---

Démonstration

On peut voir l'équation $X = A \cdot X \cup B$ comme la définition récursive d'un langage : un mot de ce langage est soit dans B, soit formé d'un mot dans A suivi d'un autre mot du langage, et on peut interpréter la solution comme une définition itérative : un mot est formé d'une suite de mots dans A, puis d'un mot final dans B.

- La vérification de 1. est aisée. En effet, $AL+B = AA^*B+B = (AA^*+\epsilon)B = A^*B = L$
- Deux possibilités :

  Méthode 1 Soit M une solution de (2.1) : ceci signifie que $M = AM + B$ et donc en particulier, que $AM + B \subseteq M$ i-e $B \subseteq M$ et $AM \subseteq M$. Pour montrer 2., i-e $A^*B \subseteq M$, il suffit de vérifier que $A^iB \subseteq M$ pour tout i. C'est une récurrence facile :
  - Pour i=0, on a $A^0B = B \subseteq M$
  - Supposons que $A^iB \subseteq M$. Alors, $A^{i+1}B = A(A^iB) \subseteq AM \subseteq M$

Méthode 2 Le langage $A^* \cdot B$ est la plus petite solution. Soit en effet L une autre solution. Alors on a : $L = AL \cup B = A(AL \cup B) \cup B = A^2L \cup AB \cup B$. En continuant à remplacer L par $AL \cup B$, on obtient pour tout $n > 0$ l'équation $L = A^{n+1}L \cup A^nB \cup A^{n-1}B \cup \cdots \cup AB \cup B$, ce qui montre que L contient tous les A^nB, donc A^*B.
Supposons maintenant que $\epsilon \notin \Sigma$. Alors, pour tout $u \in \Sigma^*$, on a $u \notin A^{|u|+1}M$, puisque les éléments de A sont au moins de longueur 1 ; si donc $u \in M$, $(E_{|u|})$ implique $u \in (\epsilon + A)^{|u|}B \subseteq A^*B$: ceci signifie que $M \subseteq L$, i-e $M = L$, puisque L est la plus petite solution de (E).

– Pour la preuve de 3., deux possibilités aussi.

Méthode 1 Montrons d'abord, par récurrence sur k, que si M est une solution de (2.1) alors (cf. les propriétés des puissances) pour tout entier naturel k :

$$(E_k) \quad M = A^{k+1}M + (\epsilon + A)^kB \tag{2.2}$$

– (E_0) signifie simplement que M vérifie (2.1).
– Si (E_k) est vraie, alors, en utilisant encore le fait que $M = AM + B$, on peut écrire :

$$M = A^{k+1}(AM+B)+(\epsilon+A)^kB = A^{k+2}M+((\epsilon+A)^k+A^{k+1})B = A^{k+2}M+(\epsilon+A)^{k+1}B$$

Ce qui implique (E_{k+1}).

Méthode 2 Si A ne contient pas ε , alors cette solution est la seule. Soit en effet L une autre solution. On sait déjà que L contient A^*B. Par ailleurs, on a pour tout $n > 0$ l'équation

$$L = A^{n+1}L \cup A^nB \cup A^{n-1}B \cup \cdots \cup AB \cup B$$

Soit w un mot de L de longueur n. Il appartient alors au membre droit de l'équation, mais il n'est pas dans $A^{n+1}L$ parce que tout mot de ce langage a longueur au moins $n + 1$ (puisque tout mot de A contient au moins une lettre). Donc le mot w appartient à un autre langage de l'union, donc à A^*B. Ceci prouve que L est contenu dans A^*B, donc l'égalité $L = A^*B$.

Application

Le lemme d'Arden permet, par la résolution d'un système d'équation par substitution, de déterminer le langage reconnu par un automate fini. On procède comme dans la méthode d'élimination de Gauss : on exprime une variable en fonction des autres, on la remplace par cette expression, on résout le système à une variable de moins, et on explicite la valeur de la variable éliminée. Soit $\mathcal{A} = (Q, \mathcal{F}, I, T)$ un automate fini sur un alphabet Z. Pour chaque état $q \in Q$,

soit L_q le langage reconnu à partir de l'état q, c'est-à-dire le langage reconnu en prenant q pour état initial. On pose enfin $Z_{q,r} = \{a \in Z \mid (q, a, r) \in T\}$. Ce sont les étiquettes de transition de q à r. On a alors :

$L_q = \bigcup_{r \in Q} Z_{q,r} \cdot L_r \cup F_q$ où $F_q = \begin{cases} \{\varepsilon\} & q \in F \\ \emptyset & q \notin F \end{cases}$.

L'application du lemme d'Arden permet alors d'éliminer une à une les inconnues L_q des n équations de la forme précédente, et d'obtenir une expression explicite des L_q et notamment des $L_i, i \in I$ qui permettent de déterminer le langage reconnu par l'automate $\mathcal{A}$.

Exemple

Un automate à deux états L'automate ci-contre donne le système d'équations :
$\begin{array}{rcl} L_1 & = & 1L_1 \cup 0L_2 \cup \varepsilon \\ L_2 & = & 1L_2 \cup 0L_1 \end{array}$ Le lemme d'Arden donne $L_2 = 1^*0 \cdot L_1$. En injectant cette expression de L_2 dans l'expression précédente de L_1 et en factorisant, on obtient $L_1 = (1 \cup 01^*0) \cdot L_1 \cup \{\varepsilon\}$, et par application du lemme d'Arden, $L_1 = (1 \cup 01^*0)^*$.

2.3.2 Inéquations linéaires à une inconnue

Dans la démonstration 2. ci-dessus, nous n'avons utilisé que la condition $AM + B \subseteq M$: ceci veut dire que nous avons démontré un résultat supplémentaire, que nous allons énoncer maintenant.
Nous appellerons solution de l'inéquation

$$(I) \quad AX + B \subseteq X \tag{2.3}$$

tout langage $L \subseteq \Sigma^*$ vérifiant la relation $AL + B \subseteq L$.

Résolution de l'inéquation (2.3)

L'inéquation (I) a une plus petite solution et celle-ci est égale à la plus petite solution $L = A^*B$ ***de l'équation (E)(2.1).***

Il est facile par ailleurs de montrer que $M = A^*(B + C)$ est une solution de (I) pour tout $C \subseteq \Sigma^*$.
Dans la pratique, on parle souvent de la plus petite solution de (I) comme étant le plus petit ensemble contenant B et qui est stable par concaténation à gauche par A. **Remarque 5:**
Dans le cas où, dans (E), le coefficient de X contient ϵ, on peut écrire ce système sous la forme $X = (\epsilon + A)X + B$, i-e $X = X + AX + B$: or, ceci est équivallent à l'inéquation (I) ci-dessus.

2.3.3 Système d'équations linéaires

Soit m un entier naturel et soient $A_{i,j} \subseteq \Sigma^* et B_i \subseteq \Sigma^*$ pour $1 \leq i, j \leq m$. Nous appellerons solution du système

$$(E) \quad \sum_{j=1}^{m} A_{i,j} X_j + B_i \text{ pour } 1 \leq i \leq m \tag{2.4}$$

tout m-uplet $L = \begin{pmatrix} L_1 \\ L_2 \\ \vdots \\ L_m \end{pmatrix}$ de langages sur Σ satisfaisant

$$L_i = \sum_{j=1}^{m} A_{i,j} L_j + B_i \text{ pour tout } 1 \leq i \leq m$$

Les m-uplets de langages se comparent terme à terme, i-e $\begin{pmatrix} L_1 \\ L_2 \\ \vdots \\ L_m \end{pmatrix} \subseteq \begin{pmatrix} M_1 \\ M_2 \\ \vdots \\ M_m \end{pmatrix}$

ce qui revient à dire que $L_i \subseteq M_i$ pour chaque i.

Exercice 2 : Démontrer toutes ces affirmations.

La définition des matrices dont les éléments sont des langages sur un alphabet Σ est évidente et les notations que nous utilisons (additive pour la réunion et multiplicative pour la concaténation) permettent alors d'écrire la seule définition raisonnable de la somme et du produit de matrices. Si l'on écrit (2.4) sous la forme matricielle, $X = AX + B$ où

$$X = \begin{pmatrix} X_1 \\ \vdots \\ X_m \end{pmatrix} \quad A = \begin{pmatrix} A_{1,1} & \dots & A_{1,m} \\ \vdots & \ddots & \vdots \\ A_{m,1} & \dots & A_{m,m} \end{pmatrix} \quad B = \begin{pmatrix} B_1 \\ \vdots \\ B_m \end{pmatrix}$$

Sa plus petite solution est tout simplement A^*B : la preuve de ce fait est une transposition du cas d'une seule équation à une seule inconnue. Ce résultat est intéressant mais le calcul de A^* n'est pas très effectif : la résolution pratique de (2.4) se fait par un algorithme facile à implanter, dont nous allons parler dans un instant.

Résolution du système d'équations (2.4)

1. (2.4) admet une plus petite solution.
2. Si $\epsilon \notin A_{i,j}$ pour chaque i et chaque j, alors (2.4) admet une solution unique.

1. ne fait que reprendre le résultat admis plus haut (1. et 2.). On peut encore observer que le système d'inéquations correspondant à (2.4) admet la même plus petite solution que lui.

Méthode de GAUSS

Un usage itéré des deux opérations suivantes permet de calculer effectivement la plus petite solution de (2.4) : il suffit de transposer aux langages la méthode de Gauss, bien connue pour la résolution des équations à coefficients réels. L'équation de X_i peut se mettre sous la forme $X_i = A_{i,i}X_i + C_i$ où C_i ne dépend pas de X_i : sous cette forme, il est possible de la "résoudre" en appliquant le résultat relatif à une équation unique ; on obtient ainsi la "résolvante partielle de (l'équation de) X_i" : $X_i = A^*_{i,i}C_i$, dont le second membre ne dépend plus de X_i.
On vérifie que pour tout entier i compris entre 1 et m :

Résolution partielle

Soit (F) le système obtenu à partir de (2.4) en remplaçant l'équation de X_i par sa résolvante partielle, alors :
la plus petite solution de (F) est égale à la plus petite solution de (2.4).

Substitution

De même, on vérifie que, pour tout couple d'entiers $i \neq k$ compris entre 1 et m :

Substitution

Soit (F) le système obtenu à partir de (2.4) en remplaçant X_i par le second membre $\sum_{j=1}^{m} A_{i,j} + B_i$ de son équation dans celle de X_k, alors :
la plus petite solution de (F) est égale à la plus petite solution de (2.4).

Exemple

Considérons le système d'équations (2.5) suivant :

$$\begin{cases} X_0 &= bX_0 + aX_1 \\ X_1 &= aX_2 + bX_3 \\ X_2 &= aX_1 + bX_3 + \epsilon \\ X_3 &= bX_1 + aX_3 \end{cases} \tag{2.5}$$

Les opérations successives :
– résolution de X_0

- résolution de X_3
- substitution de X_3 dans les équations de X_2 puis X_1
- substitution de X_2 dans l'équation de X_1

le transforment en :
$X_0 = b^*aX_1$
$X_3 = a^*bX_1$
$X_2 = aX_1 + ba^*bX_1 + \epsilon = (a + ba^*b)X_1 + \epsilon$
$X_1 = aX_2 + bX_3 = a(a + ba^*b)X_1 + a + ba^*bX_1 = (aa + ba^*b + aba^*b)X_1 + a$
Enfin, la résolution de X_1 et des substitutions évidentes, conduisent à la solution cherchée :
$X_0 = b^*aX_1 = b^*a(aa + ba^*b + aba^*b)^*a$
$X_1 = (aa + ba^*b + aba^*b)^*a$
$X_3 = a^*bX_1 = a^*b$
$X_2 = aX_1 + ba^*bX_1 = (a + ba^*b)X_1 + \epsilon$
$X_1 = aX_2 + bX_3 = a(a + ba^*b)X_1 + ba^*bX_1 = (aa + ba^*b + aba^*b)X_1$

$$\begin{cases} X_0 &= b^*a(aa + ba^*b + aba^*b)^*a \\ X_1 &= (aa + ba^*b + aba^*b)^*a \\ X_2 &= (a + ba^*b)(aa + ba^*b + aba^*b)^*a + \epsilon \\ X_3 &= a^*b \end{cases} \tag{2.6}$$

Remarques

1. Même lorsque la condition d'unicité est vérifiée, l'expression de la solution peut varier très sensiblement en fonction de l'ordre suivant lequel on effectue les opérations, comme il est facile de le constater en résolvant le système précédent d'une autre façon.
2. L'opération de substitution peut, dans certaines circonstances, être utilisée "à l'envers". Par exemple, dans le système (2.7) suivant :

$$\begin{cases} X_0 &= bX_0 + aX_1 \\ X_1 &= aX_1 + bX_3 \\ X_2 &= aX_1 + bX_3 + \epsilon \\ X_3 &= bX_1 + aX_3 \end{cases} \tag{2.7}$$

il serait dommage de ne pas constater le fait que $X_2 = X_1 + \epsilon$ et de l'utiliser en considérant le système (2.8) :

$$\begin{cases} X_0 &= bX_0 + aX_1 \\ X_1 &= aX_1 + bX_3 \\ X_2 &= X_1 + \epsilon \\ X_3 &= bX_1 + aX_3 \end{cases} \tag{2.8}$$

Ce dernier est "plus simple" que le système initial mais il admet la même plus petite solution que lui, car une substitution du second membre de l'équation de X_1 dans l'équation de X_2 redonne le système initial !

2.4 Monoïdes et morphismes de monoïdes

Pour simplifier l'écriture et la lecture, on a introduit des notations négligées, que l'on appelle des conventions d'écriture (ou abus de notation) ; ceci n'est possible que dans la mesure où l'on peut restituer la notation complète sans aucun risque d'erreur !
Ces conventions ont été résumées comme suit : $\Sigma \subseteq \Sigma^* \subseteq \mathcal{P}(\Sigma^*)$.

Plus explicitement, Σ désigne un alphabet et
- le mot de longueur 1 défini par $x \in \Sigma$ est identifié à la lettre x ;
- le langage $\{u\}$ comportant $u \in \Sigma$ comme unique élément est identifié au mot u.

En fait, ces identifications ne sont valables que parce que les éléments de Σ sont reconnaissables pour tels, comme le montre la propriété principale ci-dessous.

Dans cette section, nous donnons quelques définitions générales qui recouvrent des situations très courantes par la suite ; nous ferons encore des conventions d'écriture qui sont simplement inspirées de celles qui précèdent : elles simplifient sensiblement l'écriture et la lecture de mainte propriété mais, pour s'en convaincre et pour garder conscience de ce qu'elles signifient, on peut revenir à la notation complète !

2.4.1 Définition des monoïdes

Le triplet $(\Sigma^*, ., \epsilon)$ est le modèle le plus simple d'une structure algébrique que nous avons déjà rencontrée.

Les monoïdes

Un monoïde est un triplet $(D, \times, e)$ où :
- D est un ensemble, (analogue de Σ^*)
- $\times : D \times D \to D$ est une application, (analogue de .)
- $e \in D$ est un élément particulier de D ; (analogue de ϵ)

qui vérifie les propriétés suivantes :
- Neutralité : $x \times e = x$ et $e \times x = x$ quel que soit $x \in D$,
- Associativité : $(x \times y) \times z = x \times (y \times z)$ quels que soient $x \in D$, $y \in D$ et $z \in D$.

Exemple 9 .
- *L'ensemble N des entiers naturels est muni de deux structures de monoïdes :* $(N, +, 0)$ *et* $(N, \times, 1)$.
- *Les triplets qui suivent, où* $L \subseteq \Sigma^*$ *est un langage sur un alphabet* Σ*, sont des monoïdes, en vertu de propriétés vues dans les sections précédentes :*

  - $(L^*, ., \epsilon)$, *en particulier* $(\Sigma^*, ., \epsilon)$,
  - $(\mathcal{P}(L^*), +, \emptyset)$, *en particulier* $(\mathcal{P}(\Sigma^*), +, \emptyset)$,
  - $(\mathcal{P}(L^*), ., \epsilon)$, *en particulier* $(\mathcal{P}(\Sigma^*), ., \epsilon)$.
- *Ceci n'épuise pas les exemples de monoïdes que l'on peut construire facilement : si* $\mathcal{A}$ *et* $\mathcal{B}$ *sont deux alphabets, on peut munir l'ensemble* $\mathcal{A}^* \times \mathcal{B}^*$ *des couples de mots d'une opération de "concaténation de couples"*

$$(u, v)(u', v') = (uu'vv')$$

qui, avec l'élément neutre (ϵ, ϵ), *en fait un très beau monoïde.*

Soient deux mots $u = u_1 \dots u_n$ et $v = v1 \dots v_m$. L'égalité $u = v$ équivaut à l'égalité de toutes les lettres de u et v une à une, c'est-à-dire $u_i = v_i$ pour tout $i \leq= m$. Pour cette raison, on dit que Σ^* est le monoïde libre sur Σ.

Exemple 10 *Cette notion de « liberté » par rapport à un ensemble générateur (ici l'alphabet* Σ*) est similaire à celle des espaces vectoriels lorsqu'on dit qu'une famille de vecteurs est libre si l'égalité de deux combinaisons linéaires de ces vecteurs implique l'égalité de chacun de leurs coefficients.*

Un sous-monoïde de Σ^* *est-il toujours "libre" par rapport à l'ensemble qui l'engendre ?*
Non.

Exemple 11 *le sous-monoïde engendré par* $C_1 = \{a, ab, c, bc\}$ *n'est pas libre par rapport à* C_1 *: (a)(bc) = (ab)(c)*

On dit qu'un sous-monoïde est libre s'il existe un ensemble générateur par rapport auquel il est libre.

Exemple 12 *le sous-monoïde engendré par* $C2 = \{ab, cd, abcd\}$ *n'est pas libre par rapport à C2 car (ab)(cd) = abcd, mais il l'est pas rapport à* $C2' = \{ab, cd\}$. *En revanche le sous-monoïde précédent engendré par* C_1 *n'est intrinsèquement pas libre quelle que soit sa partie génératrice.*

Remarque 6:
Les sous-monoïdes du monoïde libre ne sont pas nécessairement libres (c'est-à-dire qu'ils n'ont pas nécessairement de partie génératrice par rapport à laquelle ils sont libres). La situation est différente dans les espaces vectoriels où tout espace vectoriel admet une base (même en dimension infinie, grâce au théorème de Zorn), en particulier tout sous-espace d'un espace vectoriel admet une base (donc une partie libre).
Un code est une partie C de Σ^* *qui engendre un sous-monoïde de* Σ^* *libre par rapport à C. Tout élément du sous-monoïde s'écrit d'une manière unique comme suite d'éléments de C. Cela signifie qu'on peut "décoder" les éléments de C*.*

- *L'ensemble Σ^n des mots de longueur n est un code, en particulier l'alphabet Σ est un code (longueur 1).*
- *Attention : Le code morse n'est pas un code dans le sens ci-dessus.*
  En effet, On a E = ".", T =" — ", et N =" —.", donc on a N = TE.
  Le décodage en morse se fait grâce à la présence de séparateurs entre les lettres.

2.4.2 Morphisme de monoïdes et propriété principale de Σ^*

Lorsque l'on passe des ensembles aux applications, on est conduit naturellement à la définition suivante :

Morphismes de monoïdes

Soient $(D, \times, e)$ et $(D', \times', e')$ deux monoïdes.
Un morphisme $(D, \times, e) \to (D', \times', e')$ est une application $h : D \to D'$ qui vérifie les propriétés suivantes :

- $h(e) = e'$
- $h(x \times y) = h(x) \times' h(y)$ quels que soient $x \in D$ et $y \in D$

En bref, un morphisme de monoïde est une application ϕ telle que $\phi(xy) = \phi(x)\phi(y)$ et $\phi(e) = e$.
Toute application d'un alphabet Σ dans un monoïde quelconque se prolonge dans Σ^* en un unique morphisme de monoïdes. Il en résulte que pour définir un morphisme sur Σ^*, il suffit de définir l'image de toutes ses lettres (de la même manière que dans un espace vectoriel, on définit un morphisme en donnant l'image de tous les vecteurs de la base).

Exemple 13 .
L'application "longueur" $u \mapsto |u|$ définit un morphisme $(\Sigma^*, ., \epsilon) \to (N, +, 0)$

La plupart des morphismes de monoïdes provient de la construction suivante.

Propriétés principales de Σ^*

Soit $(D, \times, e)$ un monoïde. Alors, toute application $f : \Sigma \to D$ s'étend de façon unique en un morphisme de monoïdes $(\Sigma^*, ., \epsilon) \to (D, \times, e)$

En effet, supposons qu'un tel morphisme $\overline{f}$ existe.

- "$\overline{f}$ étend f" signifie que $\overline{f}(x) = f(x)$ pour tout $x \in \Sigma$
- "$\overline{f}$ est un morphisme de monoïdes" implique $\overline{f} = e$ et pour tout $u \in \Sigma^*$ et tout $x \in \Sigma$, $(ux) = \overline{f}(u) \times \overline{f}(x) = \overline{f}(u) \times f(x)$.

Or, les conditions

1. $\overline{f}(\epsilon) = e$

2. $\overline{f}(ux) = \overline{f}(u) \times f(x)$ pour tout $u \in \Sigma^*$ et tout $x \in \Sigma$

définissent entièrement $\overline{f}$ par récurrence sur les mots et on peut vérifier que l'application ainsi obtenue est un morphisme de monoïdes : on sait déjà que $\overline{f}(\epsilon) = ee$, il reste donc à vérifier que l'on a $\overline{f}(u.v) = \overline{f}(u) \times \overline{f}(v)$ pour tout $u \in \Sigma^*$ et tout $v \in \Sigma^*$. Ceci se fait par récurrence sur v :

$$\begin{array}{rcll} \overline{f}(u.\epsilon) & = & \overline{f}(u) & (u.\epsilon = u) \\ & = & \overline{f}(u) \times e & (e\ est\ neutre\ pour \times) \\ & = & \overline{f}(u) \times \overline{f}(\epsilon) & (par\ 1.\ ci - dessus) \end{array}$$

Supposons que $\overline{f}(u.v) = \overline{f}(u) \times \overline{f}(v)$ et soit $x \in \Sigma$:

$$\begin{array}{rcll} \overline{f}(u.(vx)) & = & \overline{f}((u.v)x) & (u.vx = (u.v)x) \\ & = & \overline{f}(u.v) \times f(x) & (par\ 2.\ ci - dessus) \\ & = & (\overline{f}(u) \times \overline{f}(v)) \times f(x) & (par\ Hypothse\ de\ Rcurrence) \\ & = & \overline{f}(u) \times (\overline{f}(v) \times f(x)) & (par\ associativit\ de \times) \\ & = & \overline{f}(u) \times \overline{f}(vx) & (par\ 2.\ ci - dessus) \end{array}$$

L'intérêt de la propriété principale devient évident lorsque l'on réalise que, dès que Σ n'est pas vide, Σ^* est infini, même lorsque Σ est fini (ce qui sera le cas dans toutes nos applications)

Représentation des rythmes

L'ensemble $C = 10*$ des mots commençant par 1 suivi d'une suite de 0 est un code. Le sous-monoïde libre engendré par C dans Σ^* avec l'alphabet $\Sigma = 0, 1$ est l'ensemble des mots soit vide, soit commençant par 1.

Exemple 14 *décodage de 1000010011101 ? = (10000)(100)(1)(1)(10)(1)*

Remarque 7:
Cette représentation ne prend pas en compte les syncopes 000001.
L'application qui a tout élément du code $C = 10^*$ associe sa longueur se prolonge en un isomorphisme de C^* dans le monoïde libre sur l'alphabet N privé de 0 (où chaque entier, est une "lettre", donc l'alphabet est infini) :

$$\phi(1000010011101) = 531121$$

Vocabulaire et convention

Nous dirons que l'application $\overline{f}$ ci-dessus est l'extension de f aux mots et, en nous inspirant de l'identification Σ à une partie de Σ^*, nous la noterons simplement f : la propriété précédente nous le permet !

Chapitre 3

Langages réguliers et automates finis

Ce chapitre est consacré à l'étude des langages réguliers (aussi qualifiés de rationnels) qui jouent un grand rôle dans la théorie des langages, et à celle des automates finis qui permettent de les reconnaître.
En compilation, la définition des langages réguliers comme interprétation des expressions régulières constitue la "spécification lexicale", les automates finis, quant à eux, sont essentiels pour l'analyse lexicale mais ils jouent aussi un rôle dans les méthodes d'analyse syntaxique que nous étudierons plus tard.

Dans toute la suite, **les alphabets sont supposés finis** et cette hypothèse joue un rôle essentiel.

3.1 Les langages réguliers

On appelle opérations régulières, les trois opérations suivantes sur les langages :
- réunion finie,
- concaténation,
- itération.

On peut donner la définition des langages réguliers de la façon suivante :

Les langages réguliers

Un langage est dit régulier ssi on peut le construire, à partir de langages finis, par un nombre fini d'applications d'opérations régulières.

Mais nous allons reformuler cette définition de façon plus formelle avant de donner des exemples.

Techniquement, il est intéressant de considérer un langage régulier comme l'interprétation d'une expression régulière ; pour écrire ces expressions, nous utilisons les symboles suivants :$\emptyset$, les $x \in \Sigma$, $+, ., *$ et les parenthèses (et).

Les expressions régulières

$EReg(\Sigma)$ est l'ensemble des expressions définies par application des clauses inductives suivantes :

a $x \in EReg(\Sigma)$ pour tout $x \in \Sigma$

b $\emptyset \in EReg(\Sigma)$

c Si $\alpha \in EReg(\Sigma)$ et $\beta \in EReg(\Sigma)$, alors $(\alpha + \beta) \in EReg(\Sigma)$

d Si $\alpha \in EReg(\Sigma)$ et $\beta \in EReg(\Sigma)$, alors $(\alpha.\beta) \in EReg(\Sigma)$

e Si $\alpha \in EReg(\Sigma)$ alors $(\alpha^*) \in EReg(\Sigma)$

Les parenthèses déterminent entièrement la façon dont une expression régulière a été construite à partir du symbole $\emptyset$ et des éléments de Σ). Ceci signifie qu'une expression régulière admet une lecture unique, ce qui permet de raisonner par **induction** sur l'ensemble $EReg(\Sigma)$. La première application de ce principe d'induction est la définition d'une interprétation $I : EReg(\Sigma) \to \mathcal{P}(\Sigma^*)$ des expressions régulières par des langages.
I est définie par les clauses inductives suivantes :

a $I(x) = x$ pour tout $x \in \Sigma$ (langage dont le seul élément est le mot à une seule lettre x)

b $I(\emptyset) = \emptyset$ (langage vide)

c $I(\alpha + \beta) = I(\alpha) + I(\beta)$ (réunion de langages)

d $I(\alpha.\beta) = I(\alpha).I(\beta)$ (concaténation de langages)

e $I(\alpha^*) = I(\alpha)^*$ (itération d'un langage)

Les langages réguliers

$L \subseteq \Sigma^*$ est un langage régulier sur Σ ssi il existe une expression régulière $\alpha \in EReg(\Sigma)$ telle que $L = I(\alpha)$. On dira alors que α est une expression régulière de L.

Les propriétés de la somme, de la concaténation et de l'itération (cf. chapitre 1) montrent qu'en fait, chaque langage régulier admet une infinité s'expressions régulières ! La définition des langages réguliers permet de caractériser l'ensemble qu'ils forment d'une façon plus directe :

Propriétés des langages réguliers

$EReg(\Sigma)$ est l'ensemble des expressions définies par application des clauses inductives suivantes :

a $x \in EReg(\Sigma)$ pour tout $x \in \Sigma$

b $\emptyset \in EReg(\Sigma)$

c Si $L \in EReg(\Sigma)$ et $M \in EReg(\Sigma)$, alors $(L + M) \in EReg(\Sigma)$

d Si $L \in EReg(\Sigma)$ et $M \in EReg(\Sigma)$, alors $(L.M) \in EReg(\Sigma)$

e Si $L \in EReg(\Sigma)$ alors $(L^*) \in EReg(\Sigma)$

Une expression régulière est, littéralement, une présentation formelle d'un langage régulier : elle définit la forme (ou le motif) qui sert de modèle aux mots du langage en question.

Il y a peu de différence entre une expression régulière et son interprétation ; il est courant de faire des inductions, soi-disant sur les langages réguliers : en fait, il s'agit bien d'inductions sur les expressions régulières qui sont représentées par leur interprétation ...
De même, il est courant de négliger certaines parenthèses dans l'écriture d'une expression régulière : celles qui, compte tenu des conventions habituelles pour en réinstaller, ne peuvent pas en modifier l'interprétation.

Exemple 15 .

- *Le langage vide $\emptyset \subseteq \Sigma^*$ est régulier.*
- *Le langage $\epsilon \subseteq \Sigma^*$, interprétation de l'expression régulière $\emptyset^*$ est régulier. On utilise souvent $\emptyset^*$ pour désigner l'expression régulière.*
- *Un langage réduit à un seul mot est régulier. En particulier tout $x \in \Sigma$ est un langage régulier.*
- *Tout langage fini est la réunion d'une famille finie de langages à un seul mot : c'est pourquoi un langage fini est régulier.*
- *Rappelons que nous ne considérons que des alphabets Σ finis : ceci sert à assurer que $\Sigma \subseteq \Sigma^*$ lui-même est un langage régulier.*
- *En appliquant la clôture de $EReg(\Sigma)$ par itération, on déduit du point précédent que $\Sigma^* \subseteq \Sigma^*$ est régulier.*
- *Pour toute expression régulière α et tout entier n, on peut définir α^n par récurrence sur n : le résultat est une façon d'écrire une expression régulière.*
- *On peut évidemment donner une expression régulière de langages plus particuliers que ceux qui précèdent. Par exemple, pour l'alphabet $\Sigma = \{a, b\}$ comportant deux symboles distincts :*
  - *$\emptyset, \epsilon, a, b, \Sigma = a + b$ et $\Sigma^* = (a + b)^*$ définissent des langages réguliers déjà décrits,*
  - *$\Sigma^n = (a + b)^n$ est une expression régulière de l'ensemble des mots de longueur n*

- $(\epsilon+\Sigma)^n = (\epsilon+a+b)^*$ *est une expression régulière de l'ensemble des mots de longueur* $\leq n$
- $\Sigma^m(\epsilon+\Sigma)^n = \Sigma^m(\epsilon+a+b)^*$ *est une expression régulière de l'ensemble des mots dont la longueur est comprise entre* m *et* $m+n$*,*
- *enfin, voici un exemple plus spécial :*$((\epsilon+a)b)^*(\epsilon+a)$ *est une expression régulière du langage formé des mots qui ne comportent pas le facteur* aa

Ce dernier exemple est un peu plus délicat à justifier que ceux qui le précèdent : les automates que nous étudierons par la suite permettent de calculer des expressions régulières de façon plus systématique **Remarque 8:**
.

1. Il existe des langages sur Σ qui ne sont pas réguliers, dès que $\Sigma \neq \emptyset$
2. On peut construire des langages qui sont réguliers, mais pour lesquels on ne peut pas trouver effectivement d'expression régulière

3.1.1 Equations linéaires à coefficients réguliers

Nous avons vu, au chapitre 2, que la plus petite solution d'un système d'équations linéaires se calcule à partir de ses coefficients, par application d'opérations régulières : or, ces opérations transforment des langages réguliers en des langages réguliers. On a donc :

Propriété

La plus petite solution d'un système d'équations linéaires à coefficients réguliers est constituée de langages réguliers.

De plus, on peut remarquer que le calcul de cette solution (par la "méthode de Gauss") est effectif, lorsque le système est fini, comme dans la section 2.3.3. Un cas important est celui o'u les coefficients sont des langages finis : c'est lui qui nous sera utile dans la démonstration du Théorème de Kleene (Plus tard).

3.2 Automates déterministes et complets (ADC)

Nous allons d'abord donner une définition relative à un monoïde quelconque, puis, nous la spécialiserons au monoïde $(\Sigma^*, ., \epsilon)$ des mots sur un alphabet fini Σ

Actions d'un monoïde

Soient Q un ensemble et $(D, \times, e)$ un monoïde.
Une action de $(D, \times, e)$ sur Q est une application $\bullet : Q \times D \to Q$ qui vérifie :

$q \bullet e = q$

$q \bullet (x \times y) = (q \bullet x) \bullet y$

quels que soient $q \in Q,\ x \in D$ *et* $y \in D$

Remarque 9:
Les éléments de l'ensemble Q s'appellent souvent des états. Par ailleurs, il est intéressant de considérer Q comme un alphabet : les états sont alors des symboles. Dans ce qui suit, ces symboles sont souvent des entiers naturels avec lesquels il faut prendre les précautions d'usage (cf section 2.1). Nous appliquerons àQtoutes les conventions faites à propos des alphabets, par exemple, $Q = 0 + 1 + 2 + 3 + 4$ représente $\{0\} + \{1\} + \{2\} + \{3\} + \{4\} = \{0, 1, 2, 3, 4\}$ et non pas la somme, au sens arithmétique du terme !

Exemple 16 *Lorsque l'on prend $Q = D$, on a une action naturelle $q \bullet x = q \times x$. En effet, il suffit d'appliquer la définition d'un monoïde pour écrire : $q \bullet e = q \times e = q$ et $q \bullet (x \times y) = q \times (x \times y) = (q \times x) \times y = (q \bullet x) \bullet y$.*

Nous n'utilisons que les actions du monoïde $(\Sigma^*, ., \epsilon)$ des mots sur un alphabet fini Σ.

Propriété principale

Toute application $\bullet : Q \times \Sigma \to Q$ s'étend de façon unique en une action $\bar{\bullet} : Q \times \Sigma^* \to Q$

Cette propriété est une proche parente de la propriété principale de Σ^* énoncée dans la section 2.4.2.
Contentons-nous de dire que l'action $\bar{\bullet}$ peut se définir de la façon suivante, par récurrence :

1. $q \bar{\bullet} \epsilon = q$
2. $q \bar{\bullet} (ux) = (q \bar{\bullet} u) \bar{\bullet} x$

Vocabulaire et convention

Nous dirons que l'application $\bar{\bullet}$ est l'action définie par $\bullet$ et nous la noterons simplement $\bullet$: la propriété principale nous y autorise.
Avec cette convention, la seconde condition que doit vérifier $\bullet$ pour être une action est

$$q \bar{\bullet} (uv) = (q \bar{\bullet} u) \bar{\bullet} v \text{ pour tout } u, v \in \Sigma^*$$

La vérification de cette propriété est un peu longue (voyez la démonstration analogue qui a été faite pour la propriété principale de Σ^*).
La récurrence définissant l'action correspond à une procédure récursive.

Exemple 17

$$\begin{array}{rcll} q \bullet abba & = & (q \bullet abb) \bullet a & (par2) \\ & = & ((q \bullet ab) \bullet b) \bullet a & (par2) \\ & = & (((q \bullet a) \bullet b) \bullet b) \bullet a & (par2) \\ & = & ((((q \bullet \epsilon) \bullet a) \bullet b) \bullet b) \bullet a & (par2) \\ & = & (((q \bullet a) \bullet b) \bullet b) \bullet a & (par1) \end{array}$$

Les égalités justifiées par 2) sont des appels récursifs. Dans la dernière expression, tous les $\bullet$ ont enfin leur sens d'origine : il ne reste plus qu'à effectuer les calculs. On éviterait les appels récursifs en remplaçant 2) par $q \bullet xu = (q \bullet x) \bullet u$ utilisable lorsque l'on construit les mots par adjonction d'occurrences à gauche : ceci est évidemment correct et peut quelquefois se montrer intéressant.

Exemple 18 *Lorsque l'on fait agir le monoïde* $(\Sigma^*, ., \epsilon)$ *sur l'ensemble* Σ^* *lui-même (cf. l'exemple précédent), l'action naturelle est la concaténation* $\Sigma^* \times \Sigma$

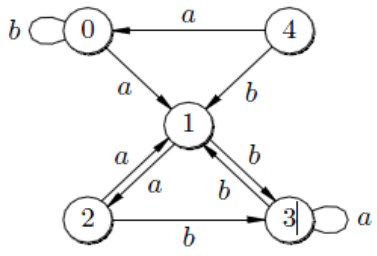
La propriété principale est importante car elle permet de donner de bonnes représentations d'une action $\bullet$ de $(\Sigma^*, ., \epsilon)$ sur un ensemble Q :

- L'action $\bullet$ peut se définir par une table sur laquelle on reporte la valeur de $q \bullet x$ à l'intersection de la ligne $q \in Q$ et de la colonne $x \in \Sigma$: cette table est finie lorsque Q l'est (ce que nous supposerons définitivement un peu plus bas).
- Le graphe de transition est une représentation géométrique dont la lecture est plus immédiate (du moins, lorsque Q et Σ n'ont pas trop d'éléments). Ce graphe est constitué de nœuds et d'arêtes étiquetés :
  - un nœud q pour chaque $q \in Q$
  - une arête de q ver r pour chaque $q \in Q$, chaque $x \in \Sigma$ et $r = q \bullet x$.

Exemple 19 *On pose* $\Sigma = a + b$ *et* $Q = 0 + 1 + 2 + 3 + 4$

Remarquez qu'il n'est pas nécessaire de préciser l'orientation d'une arête dont l'origine et l'extrémité sont confondues. Une autre convention consiste à coller plusieurs étiquettes sur une arête pour en représenter plusieurs de même origine et même extrémité

q	$q \bullet a$	$a \bullet b$
0	1	0
1	2	3
2	1	3
3	3	1
4	0	1



3.2.1 Bases

Dans cette section, nous introduisons le modèle le plus simple d'automate fini : l'automate déterministe complet. Ce modèle nous permet de définir les notions de calcul et de langage associé à un automate. Nous terminons cette section en définissant la notion d'équivalence entre automates, ainsi que la notion d'utilité d'un état.

Définition 1 ***(Automate fini déterministe (complet)).***
Un automate fini déterministe (complet) (DFA) est défini par un quintuplet $A = (\Sigma; Q; q_0; F; \delta)$*, où :*

- Σ *est un ensemble fini de symboles (l'alphabet)*
- Q *est un ensemble fini d'états*
- $q_0 \in Q$ *est l'état initial*
- $F \subset Q$ *est l'ensemble des états finaux*
- δ *est une fonction totale de* $Q \times \Sigma$ *dans* Q*, appelée fonction de transition.*

La terminologie anglaise correspondante parle de deterministic finite-state automaton. Le qualicatif de déterministe sera justifié lors de l'introduction des automates non-déterministes.

Un automate fini correspond à un graphe orienté, dans lequel certains des noeuds (états) sont distingués et marqués comme initial ou finaux et dans lequel les arcs (transitions) sont étiquetés par des symboles de Σ. Une transition est donc un triplet de $Q \times \Sigma \times Q$. Si $\delta(q, a) = r$ on dit que a est l'étiquette de la transition $(q; a; r)$; q en est l'origine, et r la destination.
Les automates admettent une représentation graphique, comme celle de la figure 3.1. Dans cette représentation, l'état initial 0 est marqué par un arc entrant sans origine et les états finaux (ici l'unique état final est 2) par un arc sortant sans destination. La fonction de transition correspondant à ce graphe s'exprime matriciellement par le graphe de la figure 3.2 : Un calcul dans l'automate A est une

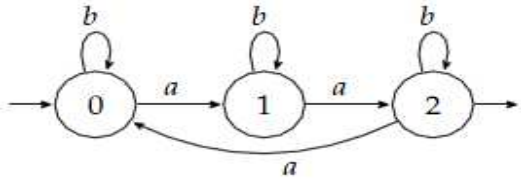


FIGURE 3.1 – Exemple d'automate

δ	a	b
0	1	0
1	2	1
2	0	2

FIGURE 3.2 – Fonction de transition

séquence de transitions $e_1 \dots e_n$ de A, telle que pour tout couple de transitions sucessives e_i, e_{i+1}, l'état destination de e_i est l'état origine de e_{i+1}. **L'étiquette** d'un calcul est le mot construit par concaténation des étiquettes de chacune des transitions.

Un calcul dans A est réussi si la première transition a pour état d'origine l'état initial, et si la dernière transition a pour destination un des états finaux. **Le langage reconnu** par l'automate A, noté L(A), est l'ensemble des étiquettes des calculs réussis. Dans l'exemple précédent, le mot baab appartient au langage reconnu, puisqu'il étiquette le calcul (réussi) : $(0, b, 0), (0, a, 1), (1, a, 2), (2, b, 2)$.

La relation $\vdash_A$ permet de formaliser la notion d'étape élémentaire de calcul. Ainsi on écrira, pour a dans Σ et v dans Σ^* $(q, av) \vdash_A (\delta(q, a), v)$ pour noter une étape de calcul utilisant la transition $(q, a, \delta(q, a))$, ce qui correspond à $(q \bullet av) = (q \bullet a) \bullet v$ vu plus haut. La clôture réflexive et transitive de $\vdash_A$ se note $\vdash_A^*$; ainsi, $(q, uv) \vdash_A^* (p, v)$ s'il existe une suite d'états $q = q_1 \dots q_n = p$ tels que $(q1, u1 \dots u_n v) \vdash_A (q2; u2 \dots u_n v) \cdots \vdash_A (q_n; v)$. La réflexivité de cette relation signifie que pour tout $(q, u), (q, u) \vdash_A^* (q, u)$. Avec ces notations, on a :

$$L(A) = \{u \in \Sigma^* \mid (q_0, u) \vdash_A^* (q, \epsilon), \text{ avec } q \in F\}$$

Cette notation met en évidence l'automate comme une machine permettant de reconnaître des mots : tout parcours partant de q_0 permet de « consommer » un à un les symboles du mot à reconnaître ; ce processus stoppe lorsque le mot est entièrement consommé : si l'état ainsi atteint est final, alors le mot appartient au langage reconnu par l'automate.

On dérive un algorithme permettant de tester si un mot appartient au langage reconnu par un automate fini déterministe.

La complexité de cet algorithme découle de l'observation que chaque étape de calcul correspond à une application de la fonction $\delta()$, qui elle-même se réduit à la lecture d'une case d'un tableau et une affectation, deux opérations qui s'effectuent en temps constant. La reconnaissance d'un mot u se calcule en exactement $|u|$ étapes.

On peut donc écrire un algorithme permettant de reconnaître un DFA :

Algorithme 1 *//$u = u_1 \dots u_n$ est le mot à reconnaître*
//$A = (Q, q0, \delta, F)$ est le DFA
$q := q_0$
$i := 1$
while$(i \leq n)$ ***do***
$q := \delta(q, u_i)$
$i := i + 1$
od
if$(q \in F)$ ***then*** $return(true)$ ***else*** $return(false)$

Définition 2 ***(Langage reconnaissable).***
Un langage est reconnaissable s'il existe un automate fini qui le reconnaît.

Un second exemple d'automate, très similaire au premier, est représenté à la figure 3.3. Dans cet exemple, 0 est à la fois initial et final. A reconnaît le langage correspondant aux mots u tels que $|u|_a$ est divisible par 3 : chaque état correspond à une valeur du reste dans la division par 3 : un calcul réussi correspond nécessairement à un reste égal à 0 et réciproquement.

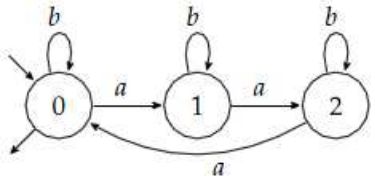


FIGURE 3.3 – AFD comptant les a (modulo 3)

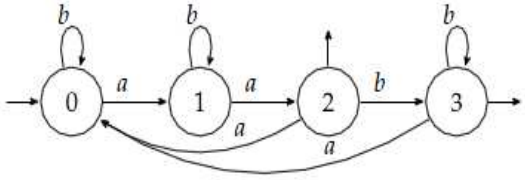


FIGURE 3.4 – AFD équivalent à 3.1

Par un raisonnement similaire à celui utilisé pour définir un calcul de longueur quelconque, il est possible d'étendre récursivement la fonction de transition δ par une fonction $\delta^* : Q \times \Sigma^* \to Q$ par :

- $\delta^*(q, u) = \delta(q, u)$ si $|u| = 1$
- $\delta^*(q, au) = \delta^*(\delta(q, a), u)$

On notera que, puisque δ est une fonction totale, δ^* est également une fonction totale : l'image d'un mot quelconque de Σ^* par δ^* est toujours bien définie, i.e. existe et est unique. Cette nouvelle notation permet de donner une notation alternative pour le langage reconnu par un automate A :

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

Nous avons pour l'instant plutôt vu l'automate comme une machine permettant de reconnaître des mots. Il est également possible de le voir comme un système de production : partant de l'état initial, tout parcours conduisant à un état final construit itérativement une séquence d'étiquettes par concaténation des étiquettes rencontrées le long des arcs.

Si chaque automate fini reconnaît un seul langage, la réciproque n'est pas vraie : plusieurs automates peuvent reconnaître le même langage. Comme pour les expressions rationnelles, on dira dans ce cas que les automates sont équivalents.

Définition 3 ***(Automates équivalents).***
Deux automates finis A1 et A2 sont équivalents si et seulement s'ils reconnaissent le même langage.

Ainsi, par exemple, l'automate de la figure 3.4 est-il équivalent à celui de la figure 3.1 : tous deux reconnaissent le langage de tous les mots qui contiennent un nombre de a congru à 2 modulo 3.

Nous l'avons noté plus haut, δ^* est définie pour tout mot de Σ^*. Ceci implique que l'algorithme de reconnaissance (par exple l'algorithme 1) demande exactement $|u|$ étapes de calcul, correspondant à une exécution complète de la boucle. Ceci peut s'avérer particulièrement inefficace, comme dans l'exemple de l'automate de la figure 3.5, qui reconnaît le langage $ab\{a, b\}^*$. Dans ce cas en effet, il est en fait possible d'accepter ou de rejeter des mots en ne considérant que les deux premiers symboles.

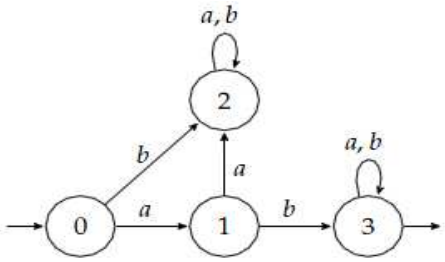


FIGURE 3.5 – Un AFD pour $ab(a + b)^*$

3.2.2 Spécification partielle

Pour contourner ce problème et pouvoir arrêter le calcul aussi tôt que possible, nous introduisons dans cette section des définitions alternatives, mais qui s'avèrent en fait équivalentes, des notions d'automate et de calcul.

Définition 4 ***(Automate fini déterministe).***
Un automate fini déterministe est défini par un quintuplet $A = (\Sigma, Q, q_0, \delta, F)$ *où :*

- Σ *est un ensemble fini de symboles (l'alphabet)*
- Q *est un ensemble fini d'états*
- $q_0 \in Q$ *est l'état initial*
- $F \subset Q$ *sont les états finaux*
- δ *est une fonction partielle de* $Q \times \Sigma$ *dans* Q

La différence avec la définition 1 est que δ est ici définie comme une fonction partielle. Son domaine de définition est un sous-ensemble de $Q \times \Sigma$. Selon cette nouvelle définition, il est possible de se trouver dans une situation où un calcul s'arrête avant d'avoir atteint la fin de l'entrée. Ceci se produit dès que l'automate

atteint une configuration (q, au) pour laquelle il n'existe pas de transition d'origine q étiquetée par a.

La définition 4 est en fait strictement équivalente à la précédente, dans la mesure où les automates partiellement spécifiés peuvent être complétés par ajout d'un état puits absorbant les transitions absentes de l'automate original, sans pour autant changer le langage reconnu.
Formellement, soit $A = (\Sigma, Q, q_0, \delta, F)$ un automate partiellement spécifié, on définit $A' = (\Sigma', Q', q'_0, \delta', F')$ avec :

- $Q' = Q \cup \{q_p\}$
- $q'_0 = q_0$
- $F' = F$
- $\forall q \in Q, a \in \Sigma, \delta'(q, a) = \delta(q, a)$ si $\delta(q, a)$ existe ; $\delta'(q, a) = q_p$ sinon.
- $\forall a \in \Sigma, \delta'(q_p, a) = q_p$

L'état puits, q_p, est donc celui dans lequel on aboutit dans A' en cas d'échec dans A ; une fois dans q_p, il est impossible d'atteindre les autres états de A et donc de rejoindre un état final. Cette transformation est illustrée pour l'automate de la figure 3.6, dont le transformé est précisément l'automate de la figure 3.5, l'état 2 jouant le rôle de puits. A' reconnaît le même langage que A puisque :

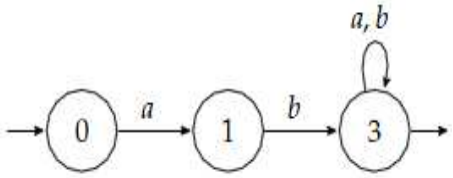


FIGURE 3.6 – Un AFD partiellement spécifié de 3.5

- Si $u \in L(A), \delta^*(q_0, u)$ existe et appartient à F. Dans ce cas, le même calcul existe dans A' et aboutit également dans un état final
- Si $u \notin L(A)$, deux cas sont possibles : soit $\delta^*(q_0, u)$ existe mais n'est pas final, et la même chose se produit dans A' ; soit le calcul s'arrête dans A après le préfixe v : on a alors $u = va$ et $\delta(\delta^*(q_0, v), a)$ n'existe pas. Or, le calcul correspondant dans A' conduit au même état, à partir duquel une transition existe vers q_p. Dès lors, l'automate est voué à rester dans cet état jusqu'à la fin du calcul ; cet état n'étant pas final, le calcul échoue et la chaîne est rejetée.

Pour tout automate (au sens de la définition 4), il existe donc un automate complètement spécifié (ou automate complet) équivalent.

3.2.3 Etats utiles

Un second résultat concernant l'équivalence entre automates demande l'introduction des quelques définitions complémentaires suivantes.

Définition 5 ***(Accessible, co-accessible, utile, émondé).***

- *Un état q de A est dit* ***accessible*** *s'il existe u dans Σ^* tel que $\delta^*(q_0, u) = q$. q_0 est trivialement accessible (par $u = \epsilon$). Un automate accessible est un automate dont tous les états sont accessibles.*
- *Un état q de A est dit co-accessible s'il existe u dans Σ^* tel que $\delta^*(q, u) \in F$. Tout état final est trivialement co-accessible (par $u = \epsilon$). Un automate co-accessible est un automate dont tous les états sont co-accessibles.*
- *Un état q de A est dit utile s'il est à la fois accessible et co-accessible.* ***Un automate émondé*** *(en anglais trim automaton) est un automate dont tous les états sont utiles.*

Les états utiles sont donc les états qui servent dans au moins un calcul réussi : on peut les atteindre depuis l'état initial, et les quitter pour un état final. Les autres états, les états inutiles, ne servent pas à grand'chose, en tout cas pas au peuplement de $L(A)$. C'est précisément ce que montre le théorème suivant.

ThéorËme 1 ***(Émondage).***
Si $L(A) \neq \emptyset$ est un langage reconnaissable, alors il est également reconnu par un automate émondé.

Démonstration : Soit A un automate reconnaissant L, et $Q_u \subset Q$ l'ensemble de ses états utiles ; Q_u n'estr pas vide dès lors que $L(A) \neq \emptyset$. La restriction δ' de δ à Q_u permet de définir un automate $A' = (\Sigma, Q_u, q_0, F, \delta')$. A' est équivalent à A. Q_u étant un sous-ensemble de Q, on a en effet immédiatement $L(A') \subset L(A)$. Soit u dans $L(A)$, tous les états du calcul qui le reconnaît étant par définition utiles, ce calcul existe aussi dans A' et aboutit dans un état final : u est donc aussi reconnu par A'.

3.2.4 Automates non-déterministes

Dans cette section, nous augmentons le modèle d'automate de la définition 4 en autorisant plusieurs transitions sortantes d'un état q à porter le même symbole : les automates ainsi spécifiés sont dits non-déterministes. Nous montrons que cette généralisation n'augmente toutefois pas l'expressivité du modèle : les langages reconnus par les automates non-déterministes sont les mêmes que ceux reconnus par les automates déterministes.

Non-déterminisme

Définition 6 ***(Automate fini non-déterministe).***
Un automate fini non-déterministe (NFA) est défini par un quintuplet $A = (\Sigma, Q, I, F, \delta)$ *où :*

- Σ *est un ensemble fini de symboles (l'alphabet)*
- Q *est un ensemble fini d'états*
- $I \subset\in Q$ *est l'état initial*
- $F \subset Q$ *sont les états finaux*
- $\delta \subset Q \times \Sigma \times Q$ *est une relation, chaque triplet* $(q, a, q') \in \delta$ *est une transition.*

On peut également considérer δ comme une fonction (partielle) de $Q \times \Sigma$ dans 2^Q : l'image par δ d'un couple (q, a) est un sous-ensemble de Q. On aurait aussi pu, sans perte de généralité, n'autoriser qu'un seul état initial.

La nouveauté introduite par cette définition est l'indétermination qui porte sur les transitions : pour une paire (q, a), il peut exister dans A plusieurs transitions possibles. On parle, dans ce cas, de **non-déterminisme**, signifiant qu'il existe des états dans lesquels la lecture d'un symbole a dans l'entrée provoque un choix (ou une indétermination) et que plusieurs transitions alternatives sont possibles.

Notez que cette définition généralise proprement la notion d'automate fini : la définition 4 est un cas particulier de la définition 7, avec pour tout (q ; a), l'ensemble $\delta(q, a)$ ne contient qu'un seul élément. En d'autres termes, tout automate déterministe est non-déterministe.
Remarquez que le vocabulaire est très trompeur : être non-déterministe ne signifie pas ne pas être déterministe ! C'est pour celà que nous notons "non-déterministe" avec un tiret, et non pas "non déterministe".

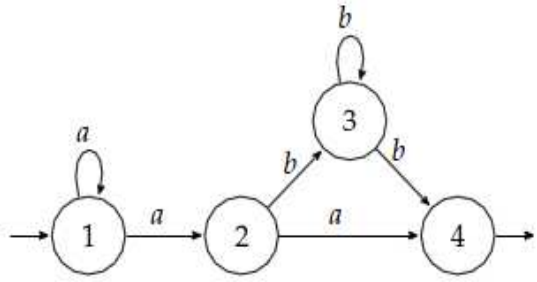
Les notions de calcul et de calcul réussi se définissent exactement comme dans le cas déterministe. On définit également la fonction de transition étendue δ^* de $Q \times \sigma^*$ dans 2^Q par :

- $\delta^*(q, u) = \{q' \in Q \mid (q, u, q') \in \delta \ si \ |u| = 1$
- $\delta^*(q, au) = \bigcup_{r \in \delta(q,a)} \delta^*(r, u)$

Ces notions sont illustrées sur l'automate non-déterministe de la figure 3.7.
Le langage reconnu par un automate non-déterministe est défini par :

$$L(A) = \{u \in \Sigma^* \mid \exists q_0 \in I : \ \delta^*(q_0, u) \cap F \neq \emptyset\}$$

Pour qu'un mot appartienne au langage reconnu par l'automate, il suffit qu'il existe, parmi tous les calculs possibles, un calcul réussi, c'est-à-dire un calcul qui consomme tous les symboles de u entre un état initial et un état final ; la reconnaissance n'échoue donc que si tous les calculs aboutissent à une des situations d'échec. Ceci implique que pour calculer l'appartenance d'un mot à un langage, il



Deux transitions sortantes de 1 sont étiquetées par a : $\delta(1, a) = \{1, 2\}$. aa donne lieu à un calcul réussi passant successivement par 1, 2 et 4, qui est final ; aa donne aussi lieu à un calcul $(1a, 1)(1, a, 1)(1, a, 1)$, qui n'est pas un calcul réussi.

FIGURE 3.7 – Un automate non-déterministe

faut examiner successivement tous les chemins possibles, et donc éventuellement revenir en arrière dans l'exploration des parcours de l'automate lorsque l'on rencontre une impasse. Dans ce nouveau modèle, le temps de reconnaissance d'un mot n'est plus linéaire, mais proportionnel au nombre de chemins dont ce mot est l'étiquette, qui peut être exponentiel en fonction de la taille de l'entrée.

Le non-déterminisme ne paye pas

La généralisation du modèle d'automate fini liée à l'introduction de transitions non-déterministes est, du point de vue des langages reconnus, sans effet : tout langage reconnu par un automate fini non-déterministe est aussi reconnu par un automate déterministe.

ThéorËme 2 *(Déterminisation). Pour tout AFN A défini sur Σ, il existe un AFD A' équivalent à A. Si A a n états, alors A' a au plus 2^n états.*

Preuve. on pose $A = (\Sigma, Q, I, F, \delta)$ et on considère A' défini par : $A' = (\Sigma, 2^Q, I, F', \delta')$ avec

- $F' = \{G \subset Q \mid F \cap G \neq \emptyset\}$
- $\forall G \subset Q, \forall a \in \Sigma, \delta'(G, a) = \bigcup_{q \in G} \delta(q, a)$

Les états de A' sont donc associés de manière biunivoque à des sous-ensembles de Q (il y en a un nombre fini) : l'état initial est l'ensemble $\{I\}$; chaque partie contenant un état final de A donne lieu à un état final de A' ; la transition sortante d'un sous-ensemble E, étiquetée par a, atteint l'ensemble de tous les états de Q atteignables depuis un état de E par une transition étiquetée par a. A' est le **déterminisé** de A. Illustrons cette construction sur l'automate de la figure 3.8.

L'automate de la figure 3.8 ayant 4 états, son déterminisé en aura donc 16, correspondant au nombre de sous-ensembles de $\{1, 2, 3, 4\}$. Son état initial est

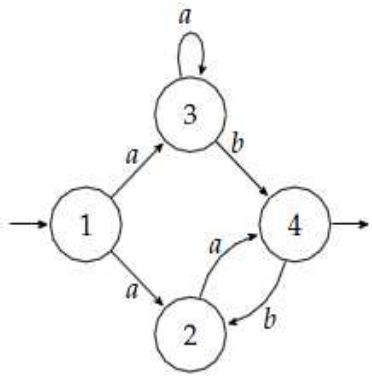


FIGURE 3.8 – Automate à déterminiser

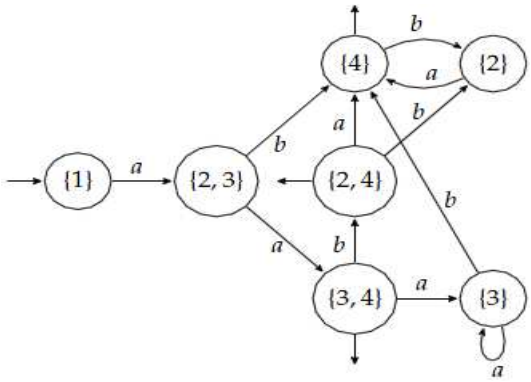


FIGURE 3.9 – Résultat de la déterminisation

le singleton $\{1\}$, et ses états finaux tous les sous-ensembles contenant 4 : il y en a exactement 8, qui sont : $\{4\}$, $\{1,4\}$, $\{2,4\}$, $\{3,4\}$, $\{1,2,4\}$, $\{1,3,4\}$, $\{2,3,4\}$, $\{1,2,3,4\}$. Considérons par exemple les transitions sortantes de l'état initial : 1 ayant deux transitions sortantes sur le symbole a, $\{1\}$, aura une transition depuis a vers l'état correspondant au doubleton $\{2,3\}$, . Le déterminisé est représenté à la figure 3.9. On notera que cette figure ne représente que les états utiles du déterminisé : ainsi $\{1,2\}$ n'est pas représenté, puisqu'il n'existe aucun moyen d'atteindre cet état.

Exercice 3 : Que se passerait-il si l'on ajoutait à l'automate de la figure 3.8 une transition supplémentaire bouclant dans l'état 1 sur le symbole a ? Construire le déterminisé de ce nouvel automate.

Démontrons maintenant le théorème 2 ; et pour saisir le sens de la démonstration, reportons nous à la figure 3.8, et considérons les calculs des mots préfixés par aaa : le premier a conduit à une indétermination entre 2 et 3 ; suivant les cas, le second a conduit donc en 4 (si on a choisi d'aller initialement en 2) ou en 3 (si on a choisi d'aller initialement en 3). La lecture du troisième a lève l'ambiguïté, puisqu'il n'y a pas de transition sortante pour 4 : le seul état possible après aaa est 3. C'est ce qui se lit sur la figure 3.9 : les ambiguïtés initiales correspondent aux états $\{2,3\}$(atteint après le premier a) et $\{3,4\}$ (atteint après le second a) ; après le troisième le doute n'est plus permis et l'état atteint correspond au singleton $\{3\}$.
Formalisons maintenant ces idées pour démontrer le résultat qui nous intéresse.

Première remarque : A' est un automate fini déterministe, puisque l'image par δ' d'un couple (H ; a) est uniquement définie.

Nous allons ensuite montrer que tout calcul dans A correspond à exactement un calcul dans A', soit formellement que :

$Si \quad \exists q_0 \in I : (q_0, u) \vdash_A^* (p, \epsilon) \quad Alors \quad \exists G \subset Q : p \in G, (\{I\}, u) \vdash_A^* (G, \epsilon)$
$Si \quad (\{I\}, u) \vdash_A^* (G, \epsilon) \quad Alors \quad \exists q_0 \in I : \forall p \in G : (q_0, u) \vdash_A^* (p, \epsilon)$

Opérons une récurrence sur la longueur de u : si u est égal à ϵ le résultat est vrai par définition de l'état initial dans A'. Supposons que le résultat est également vrai pour tout mot de longueur strictement inférieure à $|u|$, et considérons $u = va$. Soit $(q_0, va) \vdash_A^* (p, a) \vdash_A (q, \epsilon)$, un calcul dans A : par l'hypothèse de récurrence, il existe un calcul dans A' tel que : $(\{I\}, v) \vdash_{A'}^* (G, \epsilon)$ avec $p \in G$. Ceci implique, en vertu de la définition même de δ' que q appartient à $H = \delta'(G, a)$, et donc que $(\{I\}, u = va) \vdash_A^* (H, \epsilon)$, avec $q \in H$. Inversement, soit $(\{I\}, u = va) \vdash_{A'}^* (G, a) \vdash_{A'} (H, \epsilon)$: pour tout p dans G il existe un calcul dans A tel que $(q_0, v) \vdash_A^* (p, \epsilon)$. G ayant une transition étiquetée par a, il existe également dans G un état p tel que $\delta(p, a) = q$ avec $q \in H$ puisque $(q_0, u = va) \vdash_A^* (q, \epsilon)$ avec $q \in H$. On déduit alors que l'on a $(q_0, u = va) \vdash_A^* (q, \epsilon)$ avec $q \in F$ si et seulement si $(\{I\}, u) \vdash_{A'}^* (G, \epsilon)$ avec $q \in G$, donc avec $F \cap G \neq \emptyset$, soit encore $G \in F'$. Il s'en suit directement que $L(A) = L(A')$.

La construction utilisée pour construire un AFD équivalent à un AFN s'appelle la **construction des sous-ensembles** : elle se traduit directement dans un algorithme permettant de construire le déterminisé d'un automate quelconque. On notera que cette construction peut s'organiser de façon à ne considérer que les états réellement utiles du déterminisé. Il suffit, pour cela, de construire de proche en proche depuis $\{I\}$, les états accessibles, résultant en général à des automates (complets) ayant moins de 2^n états. On en déduit que l'idée générale sous-tendant la transformation d'un AFN en un AFD est que chaque état de l'AFD correspond à un ensemble d'états de l'AFN. L'AFD utilise un état pour garder trace de tous les états possibles que l'AFN peut atteindre après avoir lu chaque symbole d'entrée. Ceci revient à dire qu'après avoir lu le texte d'entrée $a_1 a_2 \ldots a_n$, l'AFD est dans un état qui représente le sous-ensemble T des états de l'AFN accessibles à partir de l'état de départ de l'AFN en suivant le chemin $a_1 a_2 \ldots a_n$.

Il existe toutefois des automates pour lesquels l'explosion combinatoire annoncée a lieu, comme celui qui est représenté à la figure 3.10. Sauriez-vous expliquer d'où vient cette difficulté ? Quel est le langage reconnu par cet automate ?

Dans la mesure où ils n'apportent aucun gain en expressivité, il est permis

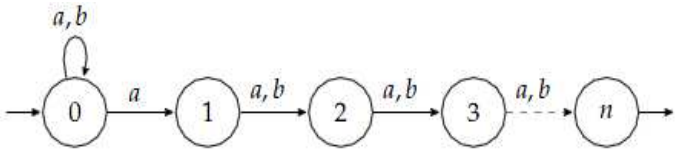


FIGURE 3.10 – Un automate difficile à déterminiser

de se demander à quoi servent les automates non-déterministes. Au moins à deux choses : ils sont plus faciles à construire à partir d'autres représentations des langages et sont donc utiles pour certaines preuves (voir plus loin) ou algorithmes. Ils fournissent également des machines bien plus (dans certains cas exponentiellement plus) "compactes" que les AFD, ce qui n'est pas une propriété négligeable.

3.2.5 Transitions spontanées

Il est commode, dans la pratique, de disposer d'une définition encore plus plastique de la notion d'automate fini, en autorisant des transitions étiquetées par le mot vide, qui sont appelées les transitions spontanées.

Définition 7 ***(Automate fini à transitions spontanées).***
Un automate fini (non-déterministe) à transitions spontanées (ϵ-AFND) est défini par un quintuplet $A = (\Sigma, Q, I, F, \delta)$ où :

- *Σ est un ensemble fini de symboles (l'alphabet)*
- *Q est un ensemble fini d'états*
- *$I \subset\in Q$ est l'état initial*
- *$F \subset Q$ sont les états finaux*
- *$\delta \subset Q \times (\Sigma \cup \{\epsilon\}) \times Q$ est une relation, chaque triplet $(q, a, q') \in \delta$ est une transition.*

Encore une fois, on aurait pu ne conserver qu'un seul état initial, et définir δ comme une fonction (partielle) de $Q \times (\Sigma \cup \{\epsilon\})$ dans 2^Q.

Formellement, un automate non-déterministe avec transitions spontanées se définit comme un AFND à la différence près que δ permet d'étiqueter les transitions par ϵ. Les transitions spontanées permettent d'étendre la notion de calcul : $(q, u) \vdash_A (p, v)$ *si* $(i) u = avet(q, a, p) \in \delta$ ou bien $(ii) u = v$ *et* $(q, \epsilon, p) \in \delta$.
En d'autres termes, dans un ϵ-AFND, il est possible de changer d'état sans consommer de symbole, en empruntant une transition étiquetée par le mot vide.
Le langage reconnu par un ϵ-NFA A est, comme précédemment, défini par :

$$L(A) = \{u \in \Sigma^* \mid \exists (q_0, u) \vdash_A^* (q, \epsilon) \text{ avec } q_0 \in I, q \in F\}$$

La figure 3.11 représente un ϵ-AFND. Cette nouvelle extension n'ajoute rien de plus à l'expressivité du modèle, puisqu'il est possible de transformer chaque ϵ-AFND A en un AFND équivalent.
On utilise les opérations définies dans le tableau 3.1 pour garder trace des ensembles d'états de l'AFN (e représente un état de l'AFN et T un ensemble d'états de l'AFN)

ThéorËme 3 *Pour tout ϵ-AFND A, il existe un AFND A' tel que L(A)=L(A').*

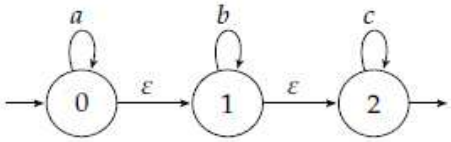


FIGURE 3.11 – Un automate avec transitions spontanées correspondant à $a^*b^*c^*$

OPERATION	DESCRIPTION
ϵ-fermeture(e)	Ensemble des états de l'AFN accessibles depuis l'état e de l'AFN par des ϵ-transitions uniquement i-e $\epsilon - fermeture(e) = \{p \in Q \mid (e,\epsilon) \vdash_A^* (p,\epsilon)\}$
ϵ-fermeture(T)	Ensemble des états de l'AFN accessibles depuis un état e appartenant à T par des ϵ-transitions uniquement i-e $\epsilon - fermeture(T) = \{p \in Q \mid \forall e \in T,\ (e,\epsilon) \vdash_A^* (p,\epsilon)\}$
Transiter(T,a)	Ensemble des états de l'AFN vers lesquels il existe une transition sur le symbole d'entrée a à partir d'un état e appartenant à T

TABLE 3.1 – Opérations sur les états d'un AFN

Démonstration
En posant $A = (\Sigma, Q, q_0, F, \delta)$, on construit $A' = (\Sigma, Q, q_0, F', \delta')$ avec
- $F' = \{q \in Q |\ \epsilon - fermeture(q) \cap F \neq \emptyset\}$
- $\delta'(q,a) = \bigcup_{p \in \epsilon - fermeture(q)} \delta(p,a)$

On déduit directement un algorithme constructif pour supprimer, à nombre d'états constant, les transitions spontanées. Appliqué à l'automate de la figure 3.11, cet algorithme construit l'automate représenté à la figure 3.12.

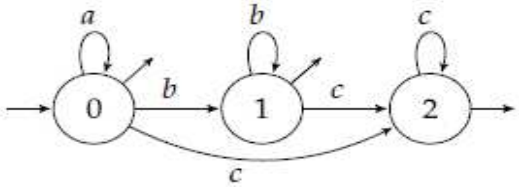


FIGURE 3.12 – L'automate de la figure 3.11 sans les transitions spontanées

Voici du reste un algorithme de construction d'un AFD à partir d'un AFN.

Algorithme de construction des sous-ensembles.
Données : Un AFN N
Résultat : Un AFD D qui accepte le même langage.
Méthode : Construction d'une table de transition Dtran pour D et Dtran est

construit de telle sorte que D simulera "en parallèle" tous les déplacements possibles que N peut effectuer sur une chaîne d'entrée donnée.

On construit Détats, l'ensemble des états de D et Dtran, la table de transition de D, de la manière suivante. Chaque état de D correspond à un ensemble d'états de l'AFN dans lesquels N pourrait se trouver après avoir lu une suite de symboles d'entrée, en incluant toutes les ϵ-transitions possibles avant ou après la lecture des symboles. L'état de départ de D est ϵ-fermeture(e_0). On ajoute des états et des transitions à D en utilisant l'algorithme 2 suivant :

Algorithme 2 ***Construction des sous-ensembles.***

Au départ, ϵ-fermeture(e_0) est l'unique état de Détats et il est non marqué
Tant que *il existe un état non marqué T dans Détats* ***faire Début***
Marquer T ;
Pour *chaque symbole d'entrée a* ***faire Début***
U := ϵ-fermeture(Transiter(T,a)) ;
Si *U $\notin$ Détats* ***Alors***
Ajouter U comme nœud non marqué à Détats ;
fsi
Dtran[T, a] :=U
fpour
ftantque

Un algorithme simple pour calculer ϵ-fermeture(T) utilise une pile pour conserver les états dont les transitions sur ϵ n'ont pas encore été examinées. L'algorithme 3 suivant présente une telle procédure.

Algorithme 3 ***Calcul de la ϵ-fermeture.***

Empiler tous les états de T dans Pile ;
Initialier ϵ-fermeture(T) à T ;
Tant que *Pile est non vide* ***faire Début***
Dépiler T, le sommet de Pile ;
Pour *chaque état u avec un arc de t à u étiqueté ϵ* ***faire***
Si *u n'est pas dans ϵ-fermeture(T)* ***Alors***
Ajouter U à ϵ-fermeture(T)
Empiler U dans Pile
fsi
finTantQue

3.3 Langages reconnaissables

Nous avons défini à la section 3.2 les langages reconnaissables comme étant les langages reconnus par un automate fini déterministe. Les sections précédentes nous ont montré que nous aurions tout aussi bien pu les définir comme les langages reconnus par un automate fini non-déterministe ou encore par un automate fini non-déterministe avec transitions spontanées.
Dans cette section, nous montrons dans un premier temps que l'ensemble des langages reconnaissables est clos pour toutes les opérations " classiques ", en produisant des constructions portant directement sur les automates. Nous montrons ensuite que les reconnaissables sont exactement les langages rationnels (présentés à la section 3.1), puis nous présentons un ensemble de résultats classiques permettant de caractériser les langages reconnaissables.

3.3.1 Opérations sur les reconnaissables

ThéorËme 4 ***(Clôture par complémentation).***
Les langages reconnaissables sont clos par complémentation.

Corrigé: Soit L un reconnaissable et A un DFA complet reconnaissant L. On construit alors un automate A' pour L en prenant $A' = (\Sigma, Q, q_0, F', \delta)$, avec $F' = Q \cap F$. Tout calcul réussi de A se termine dans un état de F, entraînant son échec dans A'. Inversement, tout calcul échouant dans A aboutit dans un état non-final de A, ce qui implique qu'il réussit dans A'. □

ThéorËme 5 ***(Clôture par union ensembliste).***
Les langages reconnaissables sont clos par union ensembliste.

Corrigé: Soit L_1 et L_2 deux langages reconnaissables, reconnus respectivement par A_1 et A_2, deux AFD complets. On construit un automate $A' = (\Sigma, Q = Q_1 \times Q_2, q_0, F, \delta)$ pour $L_1 \cup L_2$ de la manière suivante :

- $q_0 = (q_{01}, q_{02})$
- $F = (F_1 \times Q_2) \cup (Q_1 \times F_2)$
- $\delta((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a))$

La construction de A est destinée à faire fonctionner A_1 et A_2 "en parallèle" : pour tout symbole d'entrée a, on transite par δ dans la paire d'états résultant d'une transition dans A_1 et d'une transition dans A_2. Un calcul réussi dans A est un calcul réussi dans A_1 (arrêt dans un état de $F_1 \times Q_2$) ou dans A_2 (arrêt dans un état de $Q_1 \times F_2$). Nous verrons un peu plus loin une autre construction, plus simple, pour cette opération, mais qui à l'inverse de la précédente, ne préserve pas le déterminisme de la machine réalisant l'union. □

ThéorËme 6 ***(Clôture par intersection ensembliste).***
Les langages reconnaissables sont clos par intersection ensembliste.

Corrigé: Nous présentons une preuve constructive, mais notez que le résultat découle directement des deux résultats précédents et de la loi de de Morgan $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$
Soit L_1 et L_2 deux langages reconnaissables, reconnus respectivement par A_1 et A_2, deux AFD complets. On construit un automate $A' = (\Sigma, Q = Q_1 \times Q_2, q_0, F, \delta)$ pour $L_1 \cap L_2$ de la manière suivante :

- $q_0 = (q_{01}, q_{02})$
- $F = (F_1 \times Q_2) \cup (Q_1 \times F_2)$
- $\delta((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a))$

La construction de l'automate intersection est identique à celle de l'automate réalisant l'union, à la différence près qu'un calcul réussi dans A doit ici réussir simultanément dans les deux automates A_1 et A_2. Ceci s'exprime dans la nouvelle définition de l'ensemble F des états finaux comme : $F = (F_1, F_2)$. □

THéorËme 7 ***(Clôture par miroir).***
Les langages reconnaissables sont clos par miroir.

Corrigé: Soit L un langage reconnaissable, reconnu par $A = (\Sigma, Q, q_0, F, \delta)$, et n'ayant qu'un unique état final, noté q_F. A', défini par $A' = (\Sigma, Q, q_F, \{q_0\}, \delta')$, où $\delta'(q, a) = p$ si et seulement si $\delta(p, a) = q$ reconnaît exactement le langage miroir de L. A' est en fait obtenu en inversant l'orientation des arcs de A : tout calcul de A énumérant les symboles de u entre q_0 et q_F correspond à un calcul de A' énumérant u^R entre q_F et q'. On notera que même si A est déterministe, la construction ici proposée n'aboutit pas nécessairement à un automate déterministe pour le miroir. □

THéorËme 8 ***(Clôture par concaténation).***
Les langages reconnaissables sont clos par concaténation.

Corrigé: Soient L_1 et L_2 deux langages reconnus respectivement par A_1 et A_2, où l'on suppose que les ensembles d'états Q_1 et Q_2 sont disjoints et que A_1 a un unique état final de degré extérieur nul (aucune transition sortante). On construit l'automate A pour $L_1 L_2$ en identifiant l'état final de A_1 avec l'état initial de A_2. Formellement on a $A = (\Sigma, Q = Q_1 \cup Q_2 \ \{q_{02}\}, q_{01}, F_2, \delta)$ où δ est défini par :

- $\forall q \in Q_1, q \neq q_F, a \in \Sigma, \delta(q, a) = \delta_1(q, a)$
- $\delta(q, a) = \delta_2(q, a)$ *si* $q \in Q_2$
- $\delta(q_{F1}, a) = (\delta_1(q_F, a) \cup \delta_2(q_{02}, a))$

Tout calcul réussi dans A doit nécessairement atteindre un état final de A_2 et pour cela préalablement atteindre l'état final de A_1, seul point de passage vers les états de A_2. De surcroît, le calcul n'emprunte, après le premier passage dans q_{1F}, que des états de A_2 : il se décompose donc en un calcul réussi dans chacun des automates.
Réciproquement, un mot de $L_1 L_2$ se factorise sous la forme $u = vw, v \in L_1$ *et* $w \in L_2$. Chaque facteur correspond à un calcul réussi respectivement dans A_1 et dans A_2, desquels se déduit immédiatement un calcul réussi dans A. □

ThéorËme 9 ***(Clôture par étoile).***
Les langages reconnaissables sont clos par étoile.

Corrigé: La construction de A' reconnaissant L^* à partir de A reconnaissant L est immédiate : il suffit de rajouter une transition spontanée depuis tout état final de A vers l'état initial q_0. Cette nouvelle transition permet l'itération dans A' de mots de L. Pour compléter la construction, on vérifie si ϵ appartient à L(A) : si ce n'est pas le cas, alors il faudra marquer l'état initial de A comme état final de A'. □

En application de cette section, vous pourrez montrer (en construisant les automates correspondants) que les langages reconnaissables sont aussi clos pour les opérations de préfixation, suffixation, pour les facteurs, les sous-mots...

3.3.2 Langages reconnaissables et langages rationnels

Les propriétés de clôture démontrées pour les reconnaissables (pour l'union, la concaténation et l'étoile) à la section précédente, complétées par la remarque que tous les langages finis sont reconnaissables, nous permettent d'affirmer que tout langage rationnel est reconnaissable.
L'ensemble des langages rationnels étant en effet le plus petit ensemble contenant tous les ensembles finis et clos pour les opérations rationnelles, il est nécessairement inclus dans l'ensemble des reconnaissables. Nous montrons dans un premier temps comment exploiter les constructions précédentes pour construire simplement un automate correspondant à une expression rationnelle donnée. Nous montrons ensuite la réciproque, à savoir que tout reconnaissable est également rationnel : les langages reconnus par les automates finis sont précisément ceux qui sont décrits par des expressions rationnelles.

Des expressions rationnelles vers les automates

Les constructions de la section précédente ont montré comment construire les automates réalisant des opérations élémentaires sur les langages. Nous allons nous inspirer de ces constructions pour dériver un algorithme permettant de convertir une expression rationnelle en un automate fini reconnaissant le même langage.
Il existe de nombreuses façons de construire un tel automate, celle que nous donnons ci-dessous est la construction de Thompson "pure" qui présente quelques propriétés simples :

- un unique état initial q_0 sans transition entrante
- un unique état final q_F sans transition sortante
- exactement deux fois plus d'états que de symboles dans l'expression rationnelle (sans compter les parenthèses ni la concaténation, implicite).

Cette dernière propriété donne un moyen simple de contrôler le résultat en contrepartie d'automates parfois plus lourds que nécessaire.
Puisque les expressions rationnelles sont formellement définies de manière inductive (récursive) nous commençons par présenter les automates finis pour les "briques" de base que sont $\emptyset$ (figure 3.13), ϵ (figure 3.14) et les symboles de Σ (figure 3.15). À partir de ces automates élémentaires, nous allons construire de

FIGURE 3.13 – AFD pour $\emptyset$

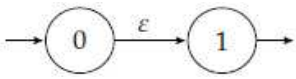


FIGURE 3.14 – Automate pour ϵ

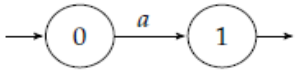


FIGURE 3.15 – AFD pour {a}

manière itérative des automates pour des expressions rationnelles plus complexes.
Si A_1 et A_2 sont les automates de Thompson de e_1 et e_2, alors l'automate de la figure 3.16 reconnaît le langage dénoté par l'expression $e_1 + e_2$.

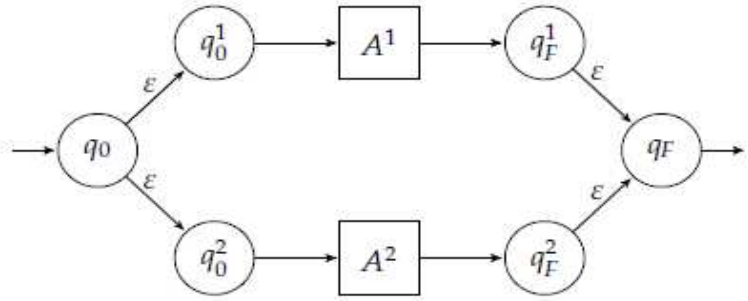


FIGURE 3.16 – Automate de Thompson pour $e_1 + e_2$

L'union correspond donc à une mise en parallèle de A_1 et de A_2 : un calcul dans cette machine est réussi si et seulement s'il est réussi dans l'une des deux machines A_1 ou A_2.

La machine reconnaissant le langage dénoté par concaténation de deux expressions e_1 et e_2 correspond à une mise en série des deux machines A_1 et A_2, où l'état final de A_1 est connecté à l'état initial de A_2 par une transition spontanée comme sur la figure 3.17.

La machine réalisant l'étoile est, comme précédemment, construite en rajoutant une possibilité de reboucler depuis l'état final vers l'état initial de la machine, ainsi qu'un arc permettant de reconnaître ϵ, comme représenté sur la figure 3.18.
À partir de ces constructions simples, il est possible de dériver un algorithme permettant de construire un automate reconnaissant le langage dénoté par une

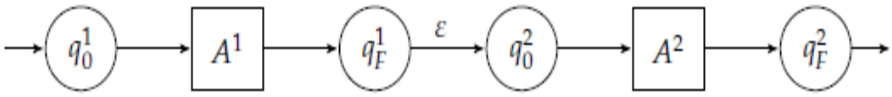


FIGURE 3.17 – Automate de Thompson pour $e_1 e_2$

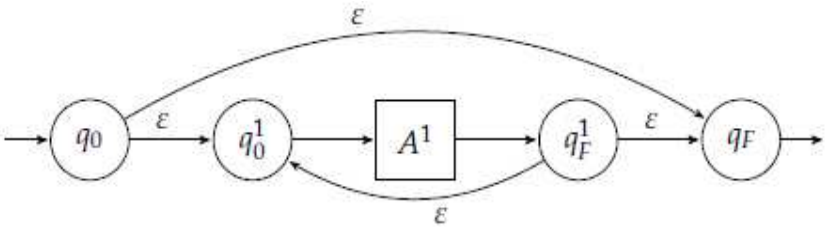


FIGURE 3.18 – Automate de Thompson pour e_1^*

expression régulière quelconque : il suffit de décomposer l'expression en ses composants élémentaires, puis d'appliquer les constructions précédentes pour construire l'automate correspondant. Cet algorithme est connu sous le nom d'algorithme de Thompson.

La figure 3.19 illustre cette construction.

Cette construction simple produit un automate qui a 2n états pour une expres-

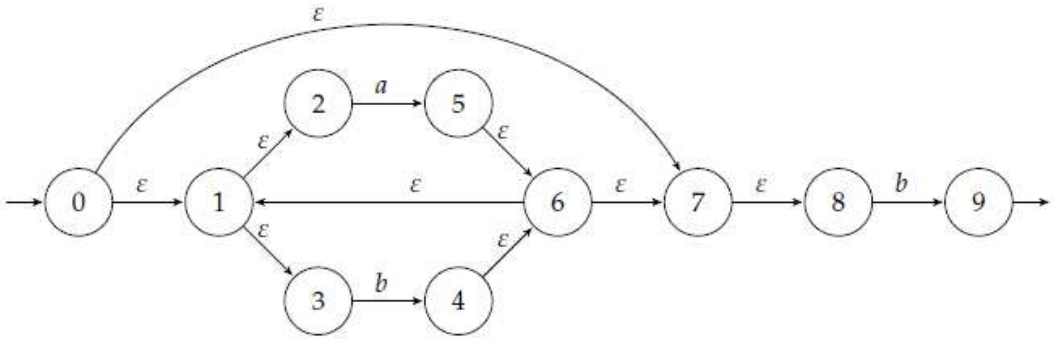


FIGURE 3.19 – Automate de Thompson pour $(a+b)^*b$

sion formée par n opérations rationnelles autres que la concaténtation (car toutes les autres opérations rajoutent exactement deux états) ; chaque état de l'automate possède au plus deux transitions sortantes. Cependant, cette construction a le désavantage de produire un automate non-déterministe, contenant de multiples transitions spontanées. Il existe d'autres procédures permettant de traiter de ma-

nière plus efficace les expressions ne contenant pas le symbole ϵ (par exemple la construction de Gloushkow) ou pour construire directement un automate.

Des automates vers les expressions rationnelles

La construction d'une expression rationnelle dénotant le langage reconnu par un automate A demande l'introduction d'une nouvelle variété d'automates, que nous appellerons généralisés. Les automates généralisés diffèrent des automates finis en ceci que leurs transitions sont étiquetées par des sous-ensembles rationnels de Σ^*. Dans un automate généralisé, l'étiquette d'un calcul se construit par concaténation des étiquettes rencontrées le long des transitions ; le langage reconnu par un automate généralisé est l'union des langages correspondant aux calculs réussis. Les automates généralisés reconnaissent exactement les mêmes langages que les automates finis "standard".

L'idée générale de la transformation que nous allons étudier consiste à partir d'un automate fini standard et de supprimer un par un les états, tout en s'assurant que cette suppression ne modifie pas le langage reconnu par l'automate. Ceci revient à construire de proche en proche une série d'automates généralisés qui sont tous équivalents à l'automate de départ. La procédure se termine lorsqu'il ne reste plus que l'unique état initial et l'unique état final : en lisant l'étiquette des transitions correspondantes, on déduit une expression rationnelle dénotant le langage reconnu par l'automate originel.

Pour se simplifier la tâche commençons par introduire deux nouveaux états, q_I et q_F, qui joueront le rôle d'uniques états respectivement initial et final. Ces nouveaux états sont connectés aux états initiaux et finaux par des transitions spontanées. On s'assure ainsi qu'à la fin de l'algorithme, il ne reste plus qu'une seule et unique transition, celle qui relie q_I à q_F.

L'opération cruciale de cette méthode est celle qui consiste à supprimer l'état q_i, où q_i n'est ni initial, ni final. On suppose qu'il y a au plus une transition entre deux états : si ça n'est pas le cas, il est possible de se ramener à cette configuration en prenant l'union des transitions existantes. On note eii l'étiquette de la transition de qi vers qi si celle-ci existe ; si elle n'existe pas on a simplement $e_{ii} = \epsilon$.

La procédure de suppression de qi comprend alors les étapes suivantes :

- pour chaque paire d'états (q_j, q_k) avec $j \neq i, k \neq i$, telle qu'il existe une transition $q_j \rightarrow q_i$ étiquetée e_{ji} et une transition $q_i \rightarrow qk$ étiquetée e_{ik}, ajouter la transition $q_j \rightarrow q_k$, portant l'étiquette $e_{ji}e_{ii}^*e_{ik}$. Si la transition $q_j \rightarrow q_k$ existe déjà avec l'étiquette e_{jk}, alors il faut aussi faire : $e_{jk} = (e_{jk} + e_{ji}e_{ii}e_{ik})$. Cette transformation doit être opérée pour chaque paire d'états (y compris quand $q_j = q_k$) avant que q_i puisse être supprimé. Ce

mécanisme est illustré graphiquement sur la figure 3.20.
– supprimer q_i, ainsi que tous les arcs incidents.

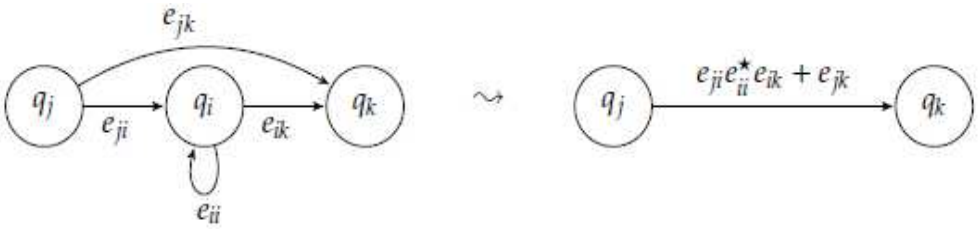


FIGURE 3.20 – Illustration de BMC : élimination de l'état i

La preuve de la correction de cet algorithme réside en la vérification qu'à chaque itération le langage reconnu ne change pas. Cet invariant se vérifie très simplement : tout calcul réussi passant par q_i avant que q_i soit supprimé contient une séquence $q_j q_i q_k$. L'étiquette de ce sous-calcul étant copiée lors de la suppression de q_i sur l'arc $q_j \to q_k$, un calcul équivalent existe dans l'automate réduit.

Cette méthode de construction de l'expression équivalente à un automate est appelée méthode d'élimination des états, connue également sous le nom d'algorithme de Brzozowski et McCluskey.

Vous noterez que le résultat obtenu dépend de l'ordre selon lequel les états sont examinés.

En guise d'application, il est recommandé de déterminer quelle est l'expression rationnelle correspondant aux automates des figures 3.8 et 3.9.

Un corollaire fondamental de ce résultat est que pour chaque langage reconnaissable, il existe (au moins) une expression rationnelle qui le dénote : tout langage reconnaissable est rationnel.

Théorème de Kleene

Nous avons montré comment construire un automate correspondant à une expression rationnelle et comment dériver une expression rationnelle dénotant un langage équivalent à celui reconnu par un automate fini quelconque. Ces deux résultats permettent d'énoncer un des résultats majeurs de ce chapitre.

THéorËme 10 ***(Théorème de Kleene).***
Un langage est reconnaissable (reconnu par un automate fini) si et seulement si il est rationnel (dénoté par une expression rationnelle).

Ce résultat permet de tirer quelques conséquences non encore envisagées : par exemple que les langages rationnels sont clos par complémentation et par intersection finie : ce résultat, loin d'être évident si on s'en tient aux notions d'opérations rationnelles, tombe simplement lorsque l'on utilise l'équivalence entre rationnels et reconnaissables. De manière similaire, les méthodes de transformation d'une représentation à l'autre sont bien pratiques : s'il est immédiat de dériver de l'automate de la figure 3.3 une expression rationnelle pour le langage contenant un nombre de a multiple de 3, il est beaucoup moins facile d'écrire directement l'expression rationnelle correspondante. En particulier, le passage par la représentation sous forme d'automates permet de déduire une méthode pour vérifier si deux expressions sont équivalentes : construire les deux automates correspondants, et vérifier qu'ils sont équivalents. Nous savons déjà comment réaliser la première partie de ce programme ; une procédure pour tester l'équivalence de deux automates est présentée à la section 3.4.2.

3.4 Quelques propriétés des langages reconnaissables

L'équivalence entre langages rationnels et langages reconnaissables a enrichi singulièrement notre palette d'outils concernant les langages rationnels : pour montrer qu'un langage est rationnel, on peut au choix exhiber un automate fini pour ce langage ou bien encore une expression rationnelle. Pour montrer qu'un langage n'est pas rationnel, il est utile de disposer de la propriété présentée dans la section 3.4.1, qui est connue sous le nom de lemme de pompage (en anglais pumping lemma). Intuitivement, cette propriété pose des limitations intrinsèques concernant la diversité des mots appartenant à un langage rationnel infini : au delà d'une certaine longueur, les mots d'un langage rationnel sont en fait construits par itération de motifs apparaissant dans des mots plus courts.

3.4.1 Lemme de pompage

ThéorËme 11 ***(Lemme de pompage).***
Soit L un langage rationnel infini. Il existe un entier p (appelé longueur du pompage) tel que pour tout mot s de L de taille au moins égale à p, s se factorise en $s = xyz$*, avec*

(i) pour tout $i \geq 0$, $xy^iz \in L$.

(ii) $|y| > 0$

(iii) $|xy| \leq k$ *et*

Si un langage est infini, alors il contient des mots de longueur arbitrairement grande. Ce que dit ce lemme, c'est essentiellement que dès qu'un mot de L est

assez long, il contient un facteur différent de ϵ qui peut être itéré à volonté tout en restant dans L. En d'autres termes, les mots "longs" de L sont construits par répétition d'un facteur s'insérant à l'intérieur de mots plus courts.

Corrigé: Soit A un AFD de p états reconnaissant L et soit s un mot de L, de longueur supérieure ou égale à p. La reconnaissance de s dans A correspond à un calcul $q_0 \dots q_n$ impliquant $|s| + 1$ états. A n'ayant que p états, le préfixe de longueur p + 1 de cette séquence contient nécessairement deux fois le même état q, aux indices i et j, avec $0 \leq i < j \leq p$. Si l'on note u le préfixe de s tel que $\delta^*(q_0, x) = q$ et y le facteur tel que $\delta^*(q, y) = q$, alors on a bien (ii) car au moins un symbole est consommé le long du cycle $q \dots q$; (iii) car $j \leq p$; (i) en court-circuitant ou en itérant les parcours le long du circuit $q \dots q$. □

Remarque 10:
: le lemme est aussi vérifié pour de nombreux langages non-rationnels : il n'exprime donc qu'une condition nécessaire (mais pas suffisante) de rationalité.
Ce lemme permet, par exemple, de prouver que le langage des carrés parfaits défini par $L = \{u \in \Sigma^*,\ \exists v \textit{ tel que } u = v^2\}$ n'est pas un langage rationnel. En effet, soit p l'entier spécifié par le lemme de pompage et s un mot plus grand que 2p : s = tt avec $|t| \geq k$. Il est alors possible d'écrire s = xyz, avec $|xy| \geq p$. Ceci implique que xy est un préfixe de t, et t un suffixe de w. Pourtant, xy^iz doit également être dans L, alors qu'un seul des t est affecté par l'itération : ceci est manifestement impossible.

Une manière simple d'exprimer cette limitation intrinsèque des langages rationnels se fonde sur l'observation suivante : dans un automate, le choix de l'état successeur pour un état q ne dépend que de q, et pas de la manière dont le calcul s'est déroulé avant q. En conséquence, un automate fini ne peut gérer qu'un nombre fini de configurations différentes, ou, dit autrement, possède une mémoire bornée. C'est insuffisant pour un langage tel que le langage des carrés parfaits pour lequel l'action à conduire (le langage à reconnaître) après un préfixe u dépend de u tout entier : reconnaître un tel langage demanderait en fait un nombre infini d'états.

3.4.2 Quelques conséquences

Dans cette section, nous établissons quelques résultats complémentaires portant sur la décidabilité, c'est-à-dire sur l'existence d'algorithmes permettant de résoudre quelques problèmes classiques portant sur les langages rationnels. Nous connaissons déjà un algorithme pour décider si un mot appartient à un langage rationnel ; cette section montre en fait que la plupart des problèmes classiques pour les langages rationnels ont des solutions algorithmiques.

ThéorËme 12 *Si A est un automate fini contenant k états :*

(i) L(A) est non vide si et seulement si A reconnaît un mot de longueur strictement inférieure à k.

(ii) L(A) est infini si et seulement si A reconnaît un mot u tel que $k \leq |u| < 2k$

Corrigé:

(i) : un sens de l'implication est trivial : si A reconnaît un mot de longueur inférieure à k, L(A) est non-vide. Supposons L(A) non vide et soit u le plus petit mot de L(A) ; supposons que la longueur de u soit strictement supérieure à k. Le calcul $(q0, u) \vdash_A^* (q, \epsilon)$ contient au moins k étapes, impliquant qu'un état au moins est visité deux fois et est donc impliqué dans un circuit C. En court-circuitant C, on déduit un mot de L(A) de longueur strictement inférieure à la longueur de u, ce qui contredit l'hypothèse de départ. On a donc bien $|u| < k$.

(ii) : un raisonnement analogue à celui utilisé pour montrer le lemme de pompage nous assure qu'un sens de l'implication est vrai. Si maintenant L(A) est infini, il doit au moins contenir un mot plus long que k. Soit u le plus petit mot de longueur au moins k : soit il est de longueur strictement inférieure à 2k, et le résultat est prouvé. Soit il est au moins égal à 2k, mais par le lemme de pompage on peut court-circuiter un facteur de taille au plus k et donc exhiber un mot strictement plus court et de longueur au moins k, ce qui est impossible. C'est donc que le plus petit mot de longueur au moins k a une longueur inférieure à 2k.

□

ThéorËme 13 *Soit A un automate fini, il existe un algorithme permettant de décider si :*

- *L(A) est vide*
- *L(A) est fini / infini*

Ce résultat découle directement des précédents : il existe, en effet, un algorithme pour déterminer si un mot u est reconnu par A. Le résultat précédent nous assure qu'il suffit de tester $|\Sigma|^k$ mots pour décider si le langage d'un automate A est vide. De même, $|\Sigma|^{2k} - |\Sigma|^k$ vérifications suffisent pour prouver qu'un automate reconnaît un langage infini.

On en déduit un résultat concernant l'équivalence :

ThéorËme 14 *Soient A1 et A2 deux automates finis. Il existe une procédure permettant de décider si A1 et A2 sont équivalents.*

Corrigé: Il suffit en effet pour cela de former l'automate reconnaissant $(L(A1)\ \overline{L(A2)}) \cup (\overline{L(A1)}(A2))$ (par exemple en utilisant les procédures décrites à la section 3.3.1) et de tester si le langage reconnu par cet automate est vide. Si c'est le cas, alors les deux automates sont effectivement équivalents. □

3.5 L'automate canonique

Dans cette section, nous donnons une nouvelle caractérisation des langages reconnaissables, à partir de laquelle nous introduisons la notion d'automate canonique d'un langage. Nous présentons ensuite un algorithme pour construire l'automate canonique d'un langage reconnaissable représenté par un DFA quelconque.

3.5.1 Une nouvelle caractérisation des reconnaissables

Commençons par une nouvelle définition : celle d'indistinguabilité.
Soit L un langage de Σ^*. Deux mots u et v sont dits indistinguables dans L si pour tout $w \in \Sigma^*$, soit uw et vw sont tous deux dans L, soit uw et vw sont tous deux dans L. On notera $\equiv_L$ la relation d'indistinguabilité dans L. En d'autres termes, deux mots u et v sont distinguables dans L s'il existe un mot $w \in \Sigma^*$ tel que $uw \in L$ et $vw \notin L$, ou bien le contraire. La relation d'indistinguabilité dans L est une relation réflexive, symétrique et transitive : c'est une relation d'équivalence.

Considérons, à titre d'illustration, le langage $L = a(a + b)(bb)^*$. Pour ce langage, $u = aab$ et $v = abb$ sont indistinguables : pour tout mot $x = uw$ de L ayant pour préfixe u, $y = vw$ est en effet un autre mot de L. $u = a$ et $v = aa$ sont par contre distinguables : en concaténant $w = abb$ à u, on obtient $aabb$ qui est dans L ; en revanche, $aaabb$ n'est pas un mot de L.

De manière similaire, on définit la notion d'indistinguabilité dans un automate déterministe.
Soit $A = (\Sigma, Q, q_0, F, \delta)$ un automate fini déterministe. Deux mots de u et v sont dits indistinguables dans A si et seulement si $\delta^*(q_0, u) = \delta^*(q_0; v)$. On notera $\equiv_A$ la relation d'indistinguabilité dans A. Autrement dit, deux mots u et v sont indistinguables pour A si le calcul par A de u depuis q_0 aboutit dans le même état q que le calcul de v. Cette notion rejoint bien la précédente, puisque tout mot w tel que $\delta^*(q, w)$ aboutisse dans un état final est une continuation valide à la fois de u et de v dans L ; inversement tout mot conduisant à un échec depuis q est une continuation invalide à la fois de u et de v.

Pour continuer, rappelons la notion de congruence droite.
Une relation d'équivalence $\Re$ sur Σ^* est dite invariante à droite si et seulement si : $(u\Re v) \Rightarrow \forall w, uw\Re vw$.
Une relation invariante à droite est appelée une congruence droite.
Par définition, les deux relations d'indistinguabilité définies ci-dessus sont invariantes à droite.
Nous sommes maintenant en mesure d'exposer le résultat principal de cette section.

ThéorËme 15 ***(De Myhill-Nerode).***
Soit L un langage sur Σ^*, les trois assertions suivantes sont équivalentes :

(i) L est un langage rationnel

(ii) il existe une relation d'équivalence $\equiv$ sur Σ^*, invariante à droite, ayant un nombre fini de classes d'équivalence et telle que L est égal à l'union de classes d'équivalence de $\equiv$

(iii) $\equiv_L$ possède un nombre fini de classes d'équivalence

Corrigé:
i $\Rightarrow$ **ii** : A étant rationnel, il existe un AFD A qui le reconnaît. La relation d'équivalence $\equiv_A$ ayant autant de classes d'équivalence qu'il y a d'états, ce nombre est nécessairement fini. Cette relation est bien invariante à droite, et L, défini comme $\{u \in \Sigma^* \delta(q_0; u) \in F$, est simplement l'union des classes d'équivalence associées aux états finaux de A.
ii $\Rightarrow$ **iii** : Soit $\equiv$ la relation satisfaisant la propriété ii, et u et v tels que $u \equiv v$. Par la propriété d'invariance droite, on a pour tout mot w dans $\Sigma^*, uw \equiv vw$. Ceci entraîne que soit uw et vw sont simultanément dans L (si leur classe d'équivalence commune est un sous-ensemble de L), soit tout deux hors de L (dans le cas contraire). Il s'en suit que $u \equiv_L v$: toute classe d'équivalence pour $\equiv$ est incluse dans une classe d'équivalence de $\equiv_L$, il y a donc moins de classes d'équivalence pour $\equiv_L$ que pour $\equiv$, ce qui entraîne que le nombre de classes d'équivalence de $\equiv_L$ est fini.
iii $\Rightarrow$ **i** : Construisons l'automate $A = (\Sigma, Q, q_0, F, \delta)$ suivant :

- chaque état de Q correspond à une classe d'équivalence [u]L de $\equiv_L$, d'où il s'en suit que Q est fini.
- $q_0 = [\epsilon]_L$, classe d'équivalence de ϵ
- $F = \{[u]_L, u \in L\}$
- $\delta([u]_L, a) = [ua]_L$. Cette définition de δ est indépendante du choix d'un représentant de $[u]_L$: si u et v sont dans la même classe pour $\equiv_L$, par invariance droite de $\equiv_L$, il en ira de même pour ua et va.

A ainsi défini est un automate fini déterministe et complet. Montrons maintenant que A reconnaît L et pour cela, montrons par induction que $(q_0, u) \vdash_A [u]_L$. Cette propriété est vraie pour $u = \epsilon$, supposons la vraie pour tout mot de taille inférieure à k et soit $u = va$ de taille k + 1. On a $(q_0, ua) \vdash_A (p, a) \vdash_A (q, \epsilon)$. Or, par l'hypothèse de récurrence on sait que $p = [u]_L$; et comme $q = \delta([u]_L, a)$, alors $q = [ua]_L$, ce qui est bien le résultat recherché. On déduit que si u est dans L(A), un calcul sur l'entrée u aboutit dans un état final de A, et donc que u est dans L. Réciproquement, si u est dans L, un calcul sur l'entrée u aboutit dans un état $[u]_L$, qui est, par définition de A, final. □

Ce résultat fournit une nouvelle caractérisation des langages reconnaissables et peut donc être utilisé pour montrer qu'un langage est, ou n'est pas, reconnaissable. Ainsi, par exemple, $L = \{u \in \Sigma^* | \ \exists a \in \Sigma, i \in N \ tq. \ u = a^{2^i}\}$ n'est

pas reconnaissable. En effet, pour tout i, j, a^{2^i} et a^{2^j} sont distingués par a^{2^i}. Il n'y a donc pas un nombre fini de classes d'équivalence pour $\equiv_L$, et L ne peut en conséquence être reconnu par un automate fini. L n'est donc pas un langage rationnel.

3.5.2 Automate canonique

La principale conséquence du résultat précédent concerne l'existence d'un représentant unique (à une renumérotation des états près) et minimal (en nombre d'états) pour les classes de la relation d'équivalence sur les automates finis.

ThéorËme 16 ***(Automate canonique).***
L'automate A_L, fondé sur la relation d'équivalence $\equiv_L$, est le plus petit automate déterministe complet reconnaissant L. Cet automate est unique (à une renumérotation des états près) et appelé automate canonique de L.

Corrigé:
Soit A un automate fini déterministe reconnaissant L. $\equiv_A$ définit une relation d'équivalence satisfaisant les conditions de l'alinéa ii de la preuve du théorème 15 présenté ci-dessus. Nous avons montré que chaque état de A correspond à une classe d'équivalence (pour $\equiv_A$) incluse dans une classe d'équivalence pour $\equiv_L$. Le nombre d'états de A est donc nécessairement plus grand que celui de AL. Le cas où A et A_L ont le même nombre d'états correspond au cas où les classes d'équivalence sont toutes semblables, permettant de définir une correspondance biunivoque entre les états des deux machines. □

L'existence de A_L étant garantie, reste à savoir comment le construire : la construction directe des classes d'équivalence de $\equiv_L$ n'est en effet pas nécessairement immédiate. Nous allons présenter un algorithme permettant de construire A_L à partir d'un automate déterministe quelconque reconnaissant L. Comme préalable, nous définissons une troisième relation d'indistinguabilité, portant cette fois sur les états :
Deux états q et p d'un automate fini déterministe A sont distinguables s'il existe un mot w tel que le calcul $(q, ; w)$ termine dans un état final alors que le calcul (p, w) échoue. Si deux états ne sont pas distinguables, ils sont indistinguables. Comme les relations d'indistinguabilité précédentes, cette relation est une relation d'équivalence, notée $\equiv_v$ sur les états de Q. L'ensemble des classes d'équivalence $[q]_v$ est notée Q_v. Pour un automate fini déterministe $A = (\Sigma, Q, q_0, F, \delta)$, on définit l'automate fini A_v par : $A_v = (\Sigma, Q_v, [q_0]_v, F_v, \delta)_v$, avec : $\delta_v([q]_v, a) = [\delta(q, a)]_v$, et $F_v = [q]_v$, avec q dans F. δ_v est correctement défini en ce sens que si p et q sont indistinguables, alors nécessairement $\delta(q, a) \equiv_v \delta(p, a)$.
Montrons, dans un premier temps, que ce nouvel automate est bien identique à

l'automate canonique A_L. On définit pour cela une application Φ qui associe un état de A_v à un état de A_L de la manière suivante :

$$\Phi([q]_v) = [u]_L \text{ s'il existe u tel que } \delta^*(q_0, u) = q$$

Notons d'abord que Φ est une application : si u et v de Σ^* aboutissent tous deux dans des états indistinguables de A, alors il est clair que u et v sont également indistinguables, et sont donc dans la même classe d'équivalence pour $\equiv_L$: le résultat de Φ ne dépend pas d'un choix particulier de u.

Montrons maintenant que Φ est une bijection. Ceci se déduit de la suite d'équivalences suivante :

$$\begin{aligned} \Phi([q]_v) = \Phi([p]_v) &\Leftrightarrow \exists u, v \in \Sigma^*, \delta^*(q_0, u) = q, \delta^*(q_0, u) = p \text{ et } u \equiv_L v \\ &\Leftrightarrow \delta^*(q_0, u) \equiv_v \delta^*(q_0, u) \\ &\Leftrightarrow [q]_v = [p]_v \end{aligned}$$

Montrons enfin que les calculs dans A_v sont en bijection par Φ avec les calculs de A_L. On a en effet :

- $Phi([q_0]_v) = [\epsilon]_L$ car $\delta^*(q_0, \epsilon) = q_0 \in [q_0]_v$
- $Phi(\delta([q]_v, a)) = \delta_L(Phi([q]_v, a)$ car soit u tel que $\delta^*(q_0, u) \in [q]_v$, alors (i)$\delta(\delta^*(q_0, u), a) \in \delta_v(q_v, a)$ (cf. la définition de δ_v) et $[ua]_L = Phi(\delta([q]_v, a)) = \delta_L([u], a)$ (cf. la définition de δ_L), ce qu'il fallait prouver.
- si $[q]_v$ est final dans A_v, alors il existe u tel que $\delta^*(q_0, u)$ soit un état final de A, impliquant que u est un mot de L, et donc que $[u]_L$ est un état final de l'automate canonique.

Il s'en suit que chaque calcul dans A_v est isomorphe (par Φ) à un calcul dans A_L, puis, que ces deux automates ayant les mêmes états initiaux et finaux, ils reconnaissent le même langage.

3.5.3 Minimisation

L'idée de l'algorithme de minimisation de l'automate déterministe $A = (\Sigma, Q, q_0, F, \delta)$ consiste alors à chercher à identifier les classes d'équivalence pour $\equiv_v$, de manière à dériver l'automate A_v (alias A_L). La finitude de Q nous garantit l'existence d'un algorithme pour calculer ces classes d'équivalence. La procédure itérative décrite ci-dessous esquisse une implantation naïve de cet algorithme, qui construit la partition correspondant aux classes d'équivalence par raffinement d'une partition initiale Π_0 qui distingue simplement états finaux et non-finaux.
Cet algorithme se glose comme suit :

- Initialiser avec deux classes d'équivalence : F et Q F
- Itérer jusqu'à stabilisation :
  - pour toute paire d'état q et p dans la même classe de la partition Π_k, s'il existe $a \in \Sigma$ tel que $\delta(q, a)$ et $\delta(p, a)$ ne sont pas dans la même classe pour Π_k, alors ils sont dans deux classes différentes de Π_{k+1}.

On vérifie que lorsque cette procédure s'arrête (après un nombre fini d'étapes), deux états sont dans la même classe si et seulement si ils sont indistinguables.

Cette procédure est connue sous le nom d'algorithme de Moore. Implantée de manière brutale, elle aboutit à une complexité quadratique (à cause de l'étape de comparaison de toutes les paires d'états). En utilisant des structures auxiliaires, il est toutefois possible de se ramener à une complexité en n log(n), avec n le nombre d'états.

Considérons pour illustrer cette procédure l'automate reproduit à la figure 3.21 : Les itérations successives de l'algorithme de construction des classes d'équivalence pour $\equiv_v$ se déroulent alors comme suit :

- $\Pi_0 = \{\{q_0, q_1, q_2, q_3, q_4\}, \{q_5\}\}$ (car q_5 est le seul état final)
- $\Pi_1 = \{\{q_0, q_1, q_3\}, \{q_2, q_4\}, \{q_5\}\}$ (car q_2 et q_4, sur le symbole b, atteignent q_5).
- $\Pi_2 = \{\{q_0\}, \{q_1, q_3\}, \{q_2, q_4\}, \{q_5\}\}$ (car q_1 et q_3, sur le symbole b, atteignent respectivement q_2 et q_4).
- $\Pi_3 = \Pi_2$ Fin de la procédure

L'automate minimal résultant de ce calcul est reproduit à la figure 3.22, qui permet de voir que l'expression régulière $(a + b)a^*ba^*b(a + b)^*$ dénote un tel automate.

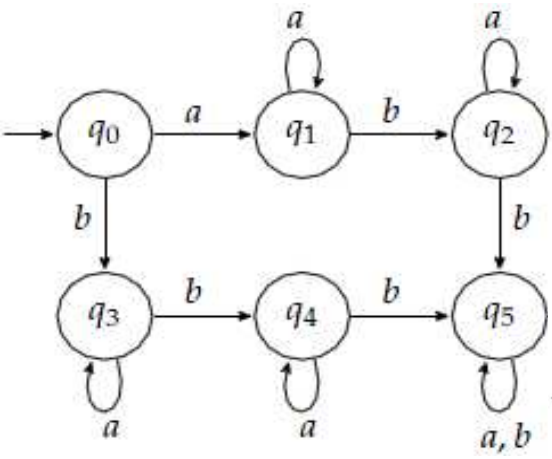


FIGURE 3.21 – Un AFD à minimiser

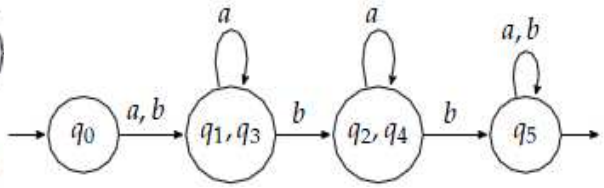


FIGURE 3.22 – L'automate minimal correspondant

Chapitre 4

Analyse lexicale

Introduction

L'analyse lexicale est la première phase de la compilation. Dans le texte source, qui se présente comme un flot de caractères, l'analyse lexicale reconnaît des unités lexicales, qui sont les " mots " avec lesquels les phrases sont formées, et les présente à la phase suivante, l'analyse syntaxique.

Les principales sortes d'unités lexicales qu'on trouve dans les langages de programmation courants sont :

- les caractères spéciaux simples : +, =, etc.
- les caractères spéciaux doubles : <=, ++, etc.
- les mots-clés : if, while, etc.
- les constantes littérales : 123, -5, etc.
- les identificateurs : i, $vitesse_d u_v ent$, etc.

A propos d'une unité lexicale reconnue dans le texte source on doit distinguer quatre notions importantes :

- l'unité lexicale, représentée généralement par un code conventionnel ; pour nos dix exemples +, =, <=, ++, if, while, 123, -5, i et $vitesse_d u_v ent$, ce pourrait être, respectivement [1] : PLUS, EGAL, INFEGAL, PLUSPLUS, SI, TANTQUE, NOMBRE, NOMBRE, IDENTIF, IDENTIF.
- le lexème, qui est la chaîne de caractères correspondante. Pour les dix exemples précédents, les lexèmes correspondants sont : "+", "=", "<=", "++", "if", "while", "123", "-5", "i" et $vitesse_d u_v ent$
- éventuellement, un attribut, qui dépend de l'unité lexicale en question, et qui la complète. Seules les dernières des dix unités précédentes ont un attribut ; pour un nombre, il s'agit de sa valeur (123,-5) ; pour un identificateur, il s'agit d'un renvoi à une table dans laquelle sont placés tous les identifi-

1. Dans un analyseur lexical écrit en C, ces codes sont des pseudo-constantes introduites par des directives #define

cateurs rencontrés (on verra cela plus loin).
- le modèle qui sert à spécifier l'unité lexicale. Nous verrons ci-après des moyens formels pour définir rigoureusement les modèles ; pour le moment nous nous contenterons de descriptions informelles comme :
  - pour les caractères spéciaux simples et doubles et les mots réservés, le lexème et le modèle coïncident,
  - le modèle d'un nombre est "une suite de chiffres, éventuellement précédée d'un signe",
  - le modèle d'un identificateur est "une suite de lettres, de chiffres et du caractère '_', commençant par une lettre".

Outre la reconnaissance des unités lexicales, les analyseurs lexicaux assurent certaines tâches mineures comme la suppression des caractères de décoration (blancs, tabulations, fins de ligne, etc.) et celle des commentaires (généralement considérés comme ayant la même valeur qu'un blanc), l'interface avec les fonctions de lecture de caractères, à travers lesquelles le texte source est acquis, la gestion des fichiers et l'affichage des erreurs, etc.

Remarque 11:

La frontière entre l'analyse lexicale et l'analyse syntaxique n'est pas fixe. D'ailleurs, l'analyse lexicale n'est pas une obligation, on peut concevoir des compilateurs dans lesquels la syntaxe est définie à partir des caractères individuels. Mais les analyseurs syntaxiques qu'il faut alors écrire sont bien plus complexes que ceux qu'on obtient en utilisant des analyseurs lexicaux pour reconnaître les mots du langage.

Simplicité et efficacité sont les raisons d'être des analyseurs lexicaux. Comme nous allons le voir, les techniques pour reconnaître les unités lexicales sont bien plus simples et efficaces que les techniques pour vérifier la syntaxe.

4.1 Expressions régulières

4.1.1 Quelques compléments

Les expressions régulières sont une importante notation pour spécifier formellement des modèles.

Remarque 12:

Les opérateurs *, concaténation et | sont associatifs à gauche, et vérifient

$$priorit(*) > priorit(concatnation) > priorit(|)$$

Ainsi, on peut écrire l'expression régulière **oui** au lieu de **(o)(u)(i)** et **oui|non** au lieu de **(oui)|(non)**, mais on ne doit pas écrire **oui*** au lieu de (**oui**)*.

Les expressions régulières se construisent à partir d'autres expressions régulières ; cela amène à des expressions passablement touffues. On les allège en introduisant des définitions régulières qui permettent de donner des noms à certaines expressions en vue de leur réutilisation. On écrit donc :

$$\begin{array}{lcl} d_1 & \rightarrow & r_1 \\ d_2 & \rightarrow & r_2 \\ & \vdots & \\ d_n & \rightarrow & r_n \end{array}$$

où chaque d_i est une chaîne sur un alphabet disjoint de Σ [2], distincte de $d_1, d_2, \cdots d_{i-1}$, et chaque r_i une expression régulière sur $\Sigma \cup \{d_1, d_2, \cdots d_{i-1}\}$

Exemple 20 *Voici quelques définitions régulières, et notamment celles de identificateur et nombre, qui définissent les identificateurs et les nombres du langage Pascal :*

lettre	$\rightarrow$	$A\|B\|\cdots\|Z\|a\|b\|\cdots\|z$
chiffre	$\rightarrow$	$0\|1\|\cdots\|9$
identificateur	$\rightarrow$	*lettre*(*lettre* $\mid$ *chiffre*)*
chiffres	$\rightarrow$	*chiffre chiffre**
fraction − opt	$\rightarrow$	$.$*chiffres*$\|\epsilon$
exposant − opt	$\rightarrow$	$(E(+\|-\|\epsilon)$*chiffres*$)\|\epsilon$
nombre	$\rightarrow$	*chiffres fraction-opt exposant-opt*

Notations abrégées

Pour alléger certaines écritures, on complète la définition des expressions régulières en ajoutant les notations suivantes :

- soit x une expression régulière, définissant le langage L(x) ; alors $(x)^+$ est une expression régulière, qui définit le langage $(L(x))^+$,
- soit x une expression régulière, définissant le langage L(x) ; alors (x)? est une expression régulière, qui définit le langage $L(x) \cup \epsilon$,
- si $c_1, c_2, \cdots c_k$ sont des caractères, l'expressions régulière $c_1|c2|\cdots c_k$ peut se noter $[c_1 c_2 \cdots c_k]$,
- à l'intérieur d'une paire de crochets comme ci-dessus, l'expression $c_1 - c_2$ désigne la séquence de tous les caractères c tels que $c1 \leq c \leq c2$.

Les définitions de lettre et chiffre données ci-dessus peuvent donc se réécrire :

lettre	$\rightarrow$	$[A - Za - z]$
chiffre	$\rightarrow$	$[0 - 9]$

2. On assure souvent la séparation entre Σ^* et les noms des définitions régulières par des conventions typographiques.

4.1.2 Ce que les expressions régulières ne savent pas faire

Les expressions régulières sont un outil puissant et pratique pour définir les unités lexicales, c'est-à-dire les constituants élémentaires des programmes. Mais elles se prêtent beaucoup moins bien à la spécification de constructions de niveau plus élevé, car elles deviennent rapidement d'une trop grande complexité.

De plus, on démontre (Cf. lemme de pompage, section 3.4.1) qu'il y a des chaînes qu'on ne peut pas décrire par des expressions régulières. Par exemple, le langage suivant (supposé infini) $\{a, (a), ((a)), (((a))), \cdots\}$ ne peut pas être défini par des expressions régulières, car ces dernières ne permettent pas d'assurer qu'il y a dans une expression de la forme

$$((\cdots((a))\cdots))$$

autant de parenthèses ouvrantes que de parenthèses fermantes. On dit que les expressions régulières "ne savent pas compter".

Pour spécifier ces structures équilibrées, si importantes dans les langages de programmation (penser aux parenthèses dans les expressions arithmétiques, les crochets dans les tableaux, $begin \cdots end$, $\{\cdots\}$, $if \cdots then \cdots$, etc.).
Nous ferons appel aux grammaires non contextuelles, expliquées plus loin.

4.2 Reconnaissance des unités lexicales

Nous avons vu comment spécifier les unités lexicales ; notre problème maintenant est d'écrire un programme qui les reconnaît dans le texte source. Un tel programme s'appelle un analyseur lexical.

Dans un compilateur, le principal client de l'analyseur lexical est l'analyseur syntaxique. L'interface entre ces deux analyseurs est une fonction **int uniteSuivante(void)** [3], qui renvoie à chaque appel l'unité lexicale suivante trouvée dans le texte source. Cela suppose que l'analyseur lexical et l'analyseur syntaxique partagent les définitions des constantes conventionnelles définissant les unités lexicales. Si on programme en C, cela veut dire que dans les fichiers sources des deux analyseurs on a inclus un fichier d'entête (fichier ".h") comportant une

3. Si on a employé l'outil lex pour fabriquer l'analyser lexical, cette fonction s'appelle plutôt yylex ; le lexème est alors pointé par la variable globale yytext et sa longueur est donnée par la variable globale yylen. Tout cela est expliqué à la section 2.3.

série de définitions comme [4] :

$$\begin{array}{lcl} \neq define & IDENTIF & 1 \\ \neq define & NOMBRE & 2 \\ \neq define & SI & 3 \\ \neq define & Alors & 4 \\ \neq define & SINON & 5 \\ & etc. & \end{array}$$

Cela suppose aussi que l'analyseur lexical et l'analyseur syntaxique partagent également une variable globale contenant le lexème correspondant à la dernière unité lexicale reconnue, ainsi qu'une variable globale contenant le (ou les) attribut(s) de l'unité lexicale courante, lorsque cela est pertinent, et notamment lorsque l'unité lexicale est NOMBRE ou IDENTIF.

On se donnera, du moins dans le cadre de ce cours, quelques "règles du jeu" supplémentaires :

- l'analyseur lexical est "glouton" : chaque lexème est le plus long possible [5] ;
- seul l'analyseur lexical accède au texte source. L'analyseur syntaxique n'acquiert ses données d'entrée autrement qu'à travers la fonction uniteSuivante ;
- l'analyseur lexical acquiert le texte source un caractère à la fois. Cela est un choix que nous faisons ici ; d'autres choix auraient été possibles, mais nous verrons que les langages qui nous intéressent permettent de travailler de cette manière.

4.2.1 Diagrammes de transition

Pour illustrer cette section nous allons nous donner comme exemple le problème de la reconnaissance des unités lexicales INFEG, DIFF, INF, EGAL, SUPEG, SUP, IDENTIF, respectivement définies par les expressions régulières $<=$, $<>$, $<$, $=$, $>=$, $>$ et lettre(lettre|chiffre)*, lettre et chiffre ayant leurs définitions déjà vues.

Les diagrammes de transition sont une étape préparatoire pour la réalisation d'un analyseur lexical. Au fur et à mesure qu'il reconnaît une unité lexicale, l'analyseur lexical passe par divers états. Ces états sont numérotés et représentés dans le diagramme par des cercles.

De chaque état e sont issues une ou plusieurs flèches étiquetées par des caractères. Une flèche étiquetée par c relie e à l'état e_1 dans lequel l'analyseur passera si,

4. Peu importent les valeurs numériques utilisées, ce qui compte est qu'elles soient distinctes.

5. Cette règle est peu mentionnée dans la littérature, pourtant elle est fondamentale. C'est grâce à elle que 123 est reconnu comme un nombre, et non plusieurs, vitesseV ent comme un seul identificateur, et force comme un identificateur, et non pas comme un mot réservé suivi d'un identificateur.

alors qu'il se trouve dans l'état e, le caractère c est lu dans le texte source.

Un état particulier représente l'état initial de l'analyseur ; on le signale en en faisant l'extrémité d'une flèche étiquetée debut.

Des doubles cercles identifient les états finaux, correspondant à la reconnaissance complète d'une unité lexicale. Certains états finaux sont marqués d'une étoile : cela signifie que la reconnaissance s'est faite au prix de la lecture d'un caractère au-delà de la fin du lexème[6]. Par exemple, la figure **??** montre les diagrammes traduisant la reconnaissance des unités lexicales INFEG, DIFF, INF, EGAL, SUPEG, SUP et IDENTIF.

Il est clair que le diagramme de la figure **??** est déterministe.

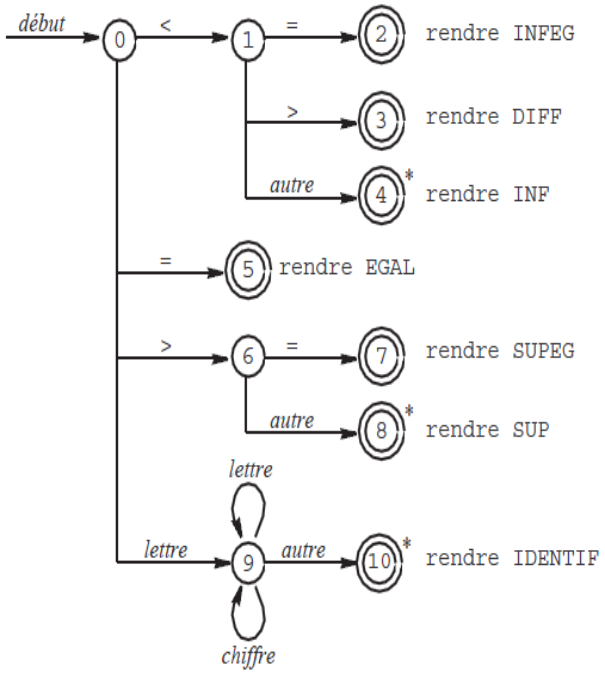


FIGURE 4.1 – Automate pour opérateurs relationnels et identificateurs

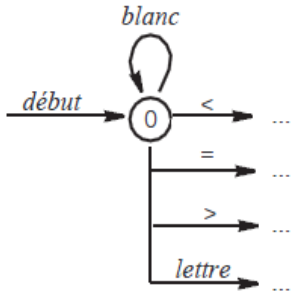


FIGURE 4.2 – Ignorer les blancs devant une unité lexicale

4.2.2 Analyseurs lexicaux programmés "en dur"

Les diagrammes de transition sont une aide importante pour l'écriture d'analyseurs lexicaux. Par exemple, à partir du diagramme de la figure 4.1 on peut

6. Il faut être attentif à ce caractère, car il est susceptible de faire partie de l'unité lexicale suivante, surtout s'il n'est pas blanc.

obtenir rapidement un analyseur lexical reconnaissant les unités INFEG, DIFF, INF, EGAL, SUPEG, SUP et IDENTIF.

Auparavant, nous apportons une légère modification à nos diagrammes de transition, afin de permettre que les unités lexicales soient séparées par un ou plusieurs blancs [7]. La figure 4.2 montre le (début du) diagramme modifié [8].

Et voici le programme obtenu :

7. Nous appelons "blanc" une espace, un caractère de tabulation ou une marque de fin de ligne

8. Notez que cela revient à modifier toutes les expressions régulières, en remplaçant "<=" par "(*blanc*)*<=", "<" par "(*blanc*)* <",etc.