THL (Théorie des langages)
Notes de cours

David A. Madore

11 juin 2021

INF105

Git : `c36193c Fri Jun 11 10:46:53 2021 +0200`
(Recopier la ligne ci-dessus dans tout commentaire sur ce document)

Table des matières

0.0.1. Présentation générale et plan : On se propose de développer ici des techniques mathématiques et algorithmiques ayant à la fois un intérêt théorique comme théorie des langages formels, et une application informatique pratique à l'analyse de textes ou de chaînes de caractères (qu'on appellera par la suite « mots », cf. §1.1).

Après des définitions générales (sections 1.1 à 1.3), ces notes sont divisées en grandes parties (inégales) de la manière suivante :

— l'étude des expressions rationnelles et des automates finis, et des langages qu'ils définissent, dans les sections 1 à 3, qui sert notamment dans la recherche de texte et de motifs dans une chaîne de caractères ou un document ;
— l'étude de certaines grammaires formelles dans la section 4, qui sert notamment à définir et analyser la syntaxe de langages de programmation ;
— une introduction à la calculabilité dans la section 5, qui place des limites théoriques sur ce qu'un algorithme peut faire.

À ces parties seront associées la définition de différentes classes de « langages » (voir §1.3 pour la définition de ce terme) qu'il s'agit d'étudier :

— les langages « rationnels » (définis par des expressions rationnelles, §1.4) et « reconnaissables » (définis par des automates finis, §2), qui s'avèrent coïncider (3.4.11) ;
— les langages « algébriques » (définis par des grammaires hors contexte, §4.1) ;
— les langages « décidables » (définis par un algorithme quelconque, 5.1.5) et « semi-décidables » (définis par un algorithme qui pourrait ne pas terminer).

La progression logique vient de l'inclusion qui donne une hiérarchie entre ces classes : tout langage rationnel/reconnaissable est algébrique (4.2.1), tout langage algébrique est décidable (4.7.4), et tout langage décidable est semi-décidable (5.1.8). On procède donc du plus particulier au plus général.

0.0.2. Notations : On écrit $A := B$, ou parfois $B =: A$, pour dire que A est défini comme égal à B (cela signifie simplement $A = B$ mais en insistant sur le fait que c'est la définition du membre du côté duquel se situent les deux points). On note $\mathbb{N}$ l'ensemble $\{0, 1, 2, 3 \ldots\}$ des entiers naturels.

Le symbole ☺ marque la fin d'une démonstration.

Les passages en petits caractères (ainsi que les notes en bas de page) sont destinés à apporter des éclaircissements, des précisions ou des compléments mais peuvent être ignorés sans nuire à la compréhension globale du texte.

1 Alphabets, mots et langages ; langages rationnels

1.1 Introduction, alphabets, mots et longueur

1.1.1. L'objet de ces notes, au confluent des mathématiques et de l'informatique, est l'étude des **langages** : un langage étant un ensemble de **mots**, eux-mêmes suites finies de **lettres** choisies dans un **alphabet**, on va commencer par définir ces différents termes avant de décrire plus précisément l'objet de l'étude.

1.1.2. Le point de départ est donc ce que les mathématiciens appelleront un **alphabet**, et qui correspond pour les informaticiens à un **jeu de caractères**. Il s'agit d'un ensemble *fini*, sans structure particulière, dont les éléments s'appellent **lettres**, ou encore **caractères** dans une terminologie plus informatique, ou parfois aussi **symboles**.

Les exemples mathématiques seront souvent donnés sur un alphabet tel que l'alphabet à deux lettres $\{a, b\}$ ou à trois lettres $\{a, b, c\}$. On pourra aussi considérer l'alphabet $\{0, 1\}$ appelé **binaire** (puisque l'alphabet n'a pas de structure particulière, cela ne fait guère de différence par rapport à n'importe quel autre alphabet à deux lettres). Dans un contexte informatique, des jeux de caractères (=alphabets) souvent importants sont ASCII, Latin-1 ou Unicode : en plus de former un ensemble, ces jeux de caractère attribuent un numéro à chacun de leurs éléments (par exemple, la lettre A majuscule porte le numéro 65 dans ces trois jeux de caractères), mais cette structure supplémentaire ne nous intéressera pas ici. Dans tous les cas, il est important pour la théorie que l'alphabet soit *fini*.

L'alphabet sera généralement fixé une fois pour toutes dans la discussion, et désigné par la lettre Σ (sigma majuscule).

1.1.3. Un **mot** sur l'alphabet Σ est une suite finie de lettres (éléments de Σ) ; dans la terminologie informatique, on parle plutôt de **chaîne de caractères**, qui est une suite finie (=liste) de caractères. Le mot est désigné en écrivant les lettres les unes à la suite des autres : autrement dit, si $x_1, \ldots, x_n \in \Sigma$ sont des lettres, le mot formé par la suite finie $x_1, \ldots, x_n$ est simplement écrit $x_1 \cdots x_n$.

À titre d'exemple, $abbcab$ est un mot sur l'alphabet $\Sigma = \{a, b, c, d\}$, et `foobar` est un mot (= chaîne de caractères) sur l'alphabet ASCII. (Dans un contexte informatique, il est fréquent d'utiliser une sorte de guillemet pour délimiter les chaînes de caractères : on écrira donc `"foobar"` pour parler du mot en question. Dans ces notes, nous utiliserons peu cette convention.)

L'ensemble de tous les mots sur un alphabet Σ sera désigné Σ^* (on verra en 1.3.8 ci-dessous cette notation comme un cas particulier d'une construction « étoile » plus générale). Par exemple, si $\Sigma = \{0, 1\}$, alors Σ^* est l'ensemble (infini !) dont les éléments sont toutes les suites finies binaires (= suites finies de 0 et de 1). Ainsi, écrire « $w \in \Sigma^*$ » signifie « w est un mot sur l'alphabet Σ ».

1.1.4. Typographiquement, on essaiera autant que possible de désigner des mots par des variables mathématiques telles que u, v, w, tandis que x, y, z désigneront plutôt des lettres quelconques dans un alphabet (quant à a, b, c, ils

serviront de lettres dans les exemples). Il n'est malheureusement pas possible d'être complètement systématique (il arrivera que x désigne un mot) : on cherchera donc à toujours rappeler le type de toute variable en écrivant, par exemple, $t \in \Sigma$ ou $t \in \Sigma^*$.

1.1.5. Le nombre n de lettres dans un mot $w \in \Sigma^*$ est appelé la **longueur** du mot, et généralement notée $|w|$ ou bien $\ell(w)$: autrement dit, si $x_1, \ldots, x_n \in \Sigma$, alors la longueur $|x_1 \cdots x_n|$ du mot $x_1 \cdots x_n$, vaut n. Ceci coïncide bien avec la notion usuelle de longueur d'une chaîne de caractères en informatique. À titre d'exemple, sur l'alphabet $\Sigma = \{a, b, c, d\}$, la longueur du mot $abbcab$ vaut 6 (on écrira $|abbcab| = 6$ ou bien $\ell(abbcab) = 6$).

1.1.6. Quel que soit l'alphabet, il existe un unique mot de longueur 0, c'est-à-dire un unique mot n'ayant aucune lettre, appelé le **mot vide** (ou la **chaîne [de caractères] vide**). Étant donné qu'il n'est pas commode de désigner un objet par une absence de symbole, on introduit un symbole spécial, généralement ε, pour désigner ce mot vide : on a donc $|\varepsilon| = 0$. On souligne que le symbole ε ne fait pas partie de l'alphabet Σ, c'est un symbole *spécial* qui a été introduit pour désigner le mot vide. (Lorsque les mots sont délimités par des guillemets, comme il est usage pour les chaînes de caractères en informatique, le mot vide n'a pas besoin d'un symbole spécial : il s'écrit juste `""` — sans aucun caractère entre les guillemets.)

Lorsque l'alphabet Σ est *vide*, c'est-à-dire $\Sigma = \varnothing$, alors le mot vide est le seul mot qui existe : on a $\Sigma^* = \{\varepsilon\}$ dans ce cas. C'est la seule situation où l'ensemble Σ^* des mots est un ensemble fini. Dans la suite, nous négligerons parfois ce cas particulier, qu'on pourra oublier : c'est-à-dire que nous ferons parfois l'hypothèse tacite que $\Sigma \neq \varnothing$.

La notation Σ^+ est parfois utilisée pour désigner l'ensemble des mots *non vides* sur l'alphabet Σ (par opposition à Σ^* qui désigne l'ensemble de tous les mots, y compris le mot vide) ; on verra en 1.3.9 ci-dessous que c'est un cas particulier d'une construction plus générale.

1.1.7. Les mots d'une seule lettre sont naturellement en correspondance avec les lettres elles-mêmes : on identifiera souvent tacitement, quoique un peu abusivement, une lettre $x \in \Sigma$ et le mot de longueur 1 formé de la seule lettre x. (En informatique, cette identification entre *caractères* et *chaînes de caractères de longueur* 1 est faite par certains langages de programmation, mais pas par tous : *caveat programmator*.) Cette convention permet d'écrire par exemple $\Sigma \subseteq \Sigma^*$ ou bien $|x| = 1 \iff x \in \Sigma$.

1.1.8. Si le cardinal de l'alphabet Σ vaut $\#\Sigma = N$, alors, pour chaque n, le nombre de mots de longueur exactement n est égal à N^n (combinatoire classique). Le nombre de mots de longueur $\leq n$ vaut donc $1 + N + \cdots + N^n = \frac{N^{n+1}-1}{N-1}$ (somme d'une série géométrique).

1.2 Concaténation de mots, préfixes, suffixes, facteurs, sous-mots

1.2.1. Si $u := x_1 \cdots x_m$ et $v := y_1 \cdots y_n$ sont deux mots, de longueurs respectives m et n, sur un même alphabet Σ, alors on définit un mot $uv := x_1 \cdots x_m y_1 \cdots y_n$ de longueur $m + n$, dont les lettres sont obtenues en mettant bout à bout celles de u puis celles de v (dans cet ordre), et on l'appelle **concaténation** (ou, si cela ne prête pas à confusion, simplement **produit**) des mots u et v. (Dans un contexte informatique, on parle de concaténation de chaînes de caractères.)

1.2.2. Parmi les propriétés de la concaténation, signalons les faits suivants :

- le mot vide ε est « **neutre** » pour la concaténation, ce qui signifie par définition : $\varepsilon w = w\varepsilon = w$ quel que soit le mot $w \in \Sigma^*$;
- la concaténation est « **associative** », ce qui signifie par définition : $u(vw) = (uv)w$ quels que soient les mots $u, v, w \in \Sigma^*$ (on peut donc noter uvw sans parenthèse).

On peut traduire de façon savante ces deux propriétés en une phrase : l'ensemble Σ^* est un **monoïde**, d'élément neutre ε, pour la concaténation (cela signifie exactement ce qui vient d'être dit).

1.2.3. On a par ailleurs $|uv| = |u| + |v|$ (la longueur de la concaténation de deux mots est la somme des concaténations), et on rappelle par ailleurs que $|\varepsilon| = 0$; on peut traduire cela de manière savante : la longueur est un **morphisme de monoïdes** entre le monoïde Σ^* des mots (pour la concaténation) et le monoïde $\mathbb{N}$ des entiers naturels (pour l'addition) (cela signifie exactement ce qui vient d'être dit).

1.2.4. Complément : Le monoïde Σ^* possède la propriété suivante par rapport à l'ensemble Σ : si M est un monoïde quelconque (c'est-à-dire un ensemble muni d'une opération binaire associative $\cdot$ et d'un élément e neutre pour cette opération), et si $\psi\colon \Sigma \to M$ est une application quelconque, alors il existe un unique morphisme de monoïdes $\hat{\psi}\colon \Sigma^* \to M$ (c'est-à-dire une application préservant le neutre et l'opération binaire) tel que $\hat{\psi}(x) = \psi(x)$ si $x \in \Sigma$. (Démonstration : on a nécessairement $\hat{\psi}(x_1 \cdots x_n) = \psi(x_1) \cdots \psi(x_n)$, or ceci définit bien un morphisme comme annoncé.) On dit qu'il s'agit là d'une propriété « universelle », et plus précisément que Σ^* est le **monoïde libre** sur l'ensemble Σ. Par exemple, le morphisme « longueur » $\ell\colon \Sigma^* \to \mathbb{N}$ est le $\ell = \hat{\psi}$ obtenu en appliquant cette propriété à la fonction (constante) $\psi(x) = 1$ pour tout $x \in \Sigma$.

1.2.5. Lorsque $w \in \Sigma^*$ et $r \in \mathbb{N}$, on définit un mot w^r comme la concaténation de r mots tous égaux à w, autrement dit, comme la répétition r fois du mot w. Formellement, on définit par récurrence :

— $w^0 = \varepsilon$ (le mot vide),
— $w^{r+1} = w^r w$.

(Ces définitions valent, d'ailleurs, dans n'importe quel monoïde. On peut constater que $w^r w^s = w^{r+s}$ quels que soient $r, s \in \mathbb{N}$.) On a bien sûr $|w^r| = r|w|$.

Cette définition sert notamment à désigner de façon concise les mots comportant des répétitions d'une même lettre : par exemple, le mot $aaaaa$ peut s'écrire tout simplement a^5, et le mot $aaabb$ peut s'écrire a^3b^2. (De même que pour le mot vide, il faut souligner que ces exposants *ne font pas partie* de l'alphabet.)

1.2.6. Lorsque $u, v, w \in \Sigma^*$ vérifient $w = uv$, autrement dit lorsque le mot w est la concaténation des deux mots u et v, on dira également :

— que u est un **préfixe** de w, ou
— que v est un **suffixe** de w.

De façon équivalente, si $w = x_1 \cdots x_n$ (où $x_1, \ldots, x_n \in \Sigma$) est un mot de longueur n, et si $0 \leq k \leq n$ est un entier quelconque compris entre 0 et n, on dira que $u := x_1 \cdots x_k$ (c'est-à-dire, le mot formé des k premières lettres de w, dans le même ordre) est le **préfixe de longueur** k de w, et que $v := x_{k+1} \cdots x_n$ (mot formé des $n - k$ dernières lettres de w, dans le même ordre) est le **suffixe de longueur** $n - k$ de w. Il est clair qu'il s'agit bien là de l'unique façon d'écrire $w = uv$ avec $|u| = k$ et $|v| = n - k$, ce qui fait le lien avec la définition donnée au paragraphe précédent ; parfois on dira que v est le suffixe **correspondant** à u ou que u est le préfixe correspondant à v (dans le mot w).

Le mot vide est préfixe et suffixe de n'importe quel mot. Le mot w lui-même est aussi un préfixe et un suffixe de lui-même. Entre les deux, pour n'importe quelle longueur k donnée, il existe un unique préfixe et un unique suffixe de longueur k. (Il peut tout à fait se produire que le préfixe et le suffixe de longueur k soient égaux pour d'autres k que 0 et $|w|$, comme le montre l'exemple qui suit.)

À titre d'exemple, le mot $abbcab$ sur l'alphabet $\Sigma = \{a, b, c, d\}$ a les sept préfixes suivants, rangés par ordre croissant de longueur : ε (le mot vide), a, ab, abb, $abbc$, $abbca$ et $abbcab$ lui-

même ; il a les sept suffixes suivants, rangés par ordre croissant de longueur : ε (le mot vide), b, ab, cab, $bcab$, $bbcab$ et $abbcab$ lui-même. Le suffixe correspondant au préfixe abb est cab puisque $abbcab = (abb)(cab)$.

1.2.7. Comme généralisation à la fois de la notion de préfixe et de celle de suffixe, on a la notion de facteur : si $u_0, v, u_1 \in \Sigma^*$ sont trois mots quelconques sur un même alphabet Σ, et si $w = u_0 v u_1$ est leur concaténation, on dira que v est un **facteur** de w. Alors qu'un préfixe ou suffixe du mot w est déterminé simplement par sa longueur, un facteur est déterminé par sa longueur et l'emplacement à partir duquel il commence.

À titre d'exemple, les facteurs du mot $abbcab$ sont : ε (le mot vide), a, b, c, ab, bb, bc, ca, abb, bbc, bca, cab, $abbc$, $bbca$, $bcab$, $abbca$, $bbcab$ et $abbcab$ lui-même.

Dans un contexte informatique, ce que nous appelons ici « facteur » est souvent appelé « sous-chaîne [de caractères] ». Il ne faut cependant pas confondre ce concept avec celui de sous-mot défini ci-dessous.

1.2.8. Si $w = x_1 \cdots x_n$ est un mot de longueur n, on appelle **sous-mot** de w un mot de la forme $x_{i_1} \cdots x_{i_k}$ où $1 \leq i_1 < \cdots < i_k \leq n$. En plus clair, cela signifie que v est obtenu en ne gardant que certaines lettres du mot w, dans le même ordre, mais en en effaçant d'autres ; à la différence du concept de facteur, celui de sous-mot n'exige pas que les lettres gardées soient consécutives.

À titre d'exemple, le mot acb est un sous-mot du mot $abbcab$ (obtenu en gardant les lettres soulignées ici : $\underline{a}bb\underline{c}a\underline{b}$; pour se rattacher à la définition ci-dessus, on pourra prendre $(i_1, i_2, i_3) = (1, 4, 6)$).

1.2.9. Les remarques de dénombrement suivantes peuvent aider à mieux comprendre les notations de préfixe, suffixe, facteur et sous-mot : si w est un mot de longueur n, alors il a

- exactement $n + 1$ préfixes (car un préfixe est déterminé par sa longueur k entre 0 et n),
- exactement $n + 1$ suffixes (raison analogue),
- au plus $\sum_{k=1}^{n}(n+1-k)+1 = \frac{1}{2}(n^2+n+2)$ facteurs (car un facteur est déterminé par sa longueur k et son point de départ qui peut être choisi parmi $n + 1 - k$ possibilités, le $+1$ final étant mis pour le facteur vide),
- au plus 2^n sous-mots (car un sous-mot est déterminé en choisissant, pour chacune des n lettres, si on l'efface ou la conserve).

Le nombre exact peut être plus petit en cas de coïncidences entre certains choix (par exemple, aaa n'a que 4 facteurs, ε, a, aa, aaa alors que abc en a bien $\frac{1}{2}(3^2 + 3 + 2) = 7$) ; mais les bornes ci-dessus sont effectivement atteintes pour certains mots.

1.2.10. Si $w = x_1 \cdots x_n$, où $x_1, \ldots, x_n \in \Sigma$, est un mot de longueur n sur un alphabet Σ, alors on définit son mot **miroir** ou **transposé**, parfois noté w^{R} ou w^{T} (parfois les exposants sont écrits à gauche), comme le mot $x_n \cdots x_1$ dont les lettres sont les mêmes que celles de w mais dans l'ordre inverse. À titre d'exemple, $(ababb)^{\mathsf{R}} = bbaba$. On remarquera que $(w_1 w_2)^{\mathsf{R}} = w_2^{\mathsf{R}} w_1^{\mathsf{R}}$ si w_1, w_2 sont deux mots quelconques.

Un mot w est dit **palindrome** lorsque $w = w^{\mathsf{R}}$. Par exemple, $abba$ est un palindrome sur $\{a, b, c, d\}$ (ou bien le mot « ressasser » sur l'alphabet du français).

1.2.11. La notation suivante est souvent utile : si w est un mot sur un alphabet Σ et si z est une lettre (= élément de Σ), on note $|w|_z$ le nombre total d'occurrences de la lettre z dans w. À titre d'exemple, sur l'alphabet $\Sigma = \{a, b, c, d\}$, le nombre d'occurrences des différentes lettres dans le mot $abbcab$ sont : $|abbcab|_a = 2$, $|abbcab|_b = 3$, $|abbcab|_c = 1$ et $|abbcab|_d = 0$.

Formellement, on peut définir $|w|_z$ de la façon suivante : si $w = x_1 \cdots x_n$ où $x_1, \ldots, x_n \in \Sigma$, alors $|w|_z$ est le cardinal de l'ensemble $\{i : x_i = z\}$. On peut remarquer qu'on a : $|w| = \sum_{z \in \Sigma} |w|_z$ (i.e., la longueur de w est la somme des nombres d'occurrences dans celui-ci des différentes lettres de l'alphabet).

1.3 Langages et opérations sur les langages

1.3.1. Un **langage** L sur l'alphabet Σ est simplement un ensemble de mots sur Σ. Autrement dit, il s'agit d'un sous-ensemble (= une partie) de l'ensemble Σ^* (de tous les mots sur Σ) : en symboles, $L \subseteq \Sigma^*$.

On souligne qu'on ne demande pas que L soit fini (mais il peut l'être, auquel cas on parlera fort logiquement de « langage fini »).

1.3.2. À titre d'exemple, l'ensemble $\{d, dc, dcc, dccc, dcccc, \ldots\} = \{dc^r : r \in \mathbb{N}\}$ des mots formés d'un d suivi d'un nombre quelconque (éventuellement nul) de c est un langage sur l'alphabet $\Sigma = \{a, b, c, d\}$. On verra plus loin que ce langage est « rationnel » (et pourra être désigné par l'expression rationnelle $dc*$).

Voici quelques autres exemples de langages :

— Le langage (fini) $\{foo, bar, baz\}$ constitué des seuls trois mots foo, bar, baz sur l'alphabet $\Sigma = \{a, b, f, o, r, z\}$.

— Le langage (fini) constitué des mots de longueur exactement 42 sur l'alphabet $\Sigma = \{p, q, r\}$. Comme on l'a vu en 1.1.8, cet ensemble a pour cardinal exactement 3^{42}.

— Le langage constitué des mots de longueur exactement 1 sur un alphabet Σ (= mots de une seule lettre), qu'on peut identifier à Σ lui-même (en identifiant un mot de une lettre à la lettre en question, cf. 1.1.7).

— Le langage (fini) constitué du seul mot vide (= mot de longueur exactement 0) sur l'alphabet, disons, $\Sigma = \{p, q, r\}$. Ce langage $\{\varepsilon\}$ a pour cardinal 1 (ou 3^0 si on veut). Il ne faut pas le confondre avec le suivant :

— Le langage vide, qui ne contient aucun mot (sur un alphabet quelconque). Ce langage a pour cardinal 0.

— Le langage sur l'alphabet $\Sigma = \{a, b\}$ constitué des mots qui commencent par trois a consécutifs : ou, si on préfère, qui ont le mot aaa comme préfixe.

— Le langage sur l'alphabet $\Sigma = \{a, b\}$ constitué des mots qui contiennent trois a consécutifs ; ou, si on préfère, qui ont aaa comme facteur.

— Le langage sur l'alphabet $\Sigma = \{a, b\}$ constitué des mots qui contiennent au moins trois a, non nécessairement consécutifs ; ou, si on préfère, qui ont aaa comme sous-mot.

— Le langage sur l'alphabet $\Sigma = \{a\}$ constitué de tous les mots dont la longueur est un nombre premier ($L = \{aa, aaa, a^5, a^7, a^{11}, \ldots\}$). Ce langage est infini.

— Le langage sur l'alphabet $\Sigma = \{0, 1\}$ constitué de tous les mots commençant par un 1 et qui, interprétés comme un nombre écrit en binaire, désignent un nombre premier ($L = \{10, 11, 101, 111, 1011, \ldots\}$).

— Le langage sur l'alphabet Unicode constitué de tous les mots qui représentent un document XML bien-formé d'après la spécification XML 1.0.

1.3.3. On pourrait aussi considérer un langage (sur l'alphabet Σ) comme une *propriété* des mots (sur l'alphabet en question). Précisément, si P est une propriété qu'un mot $w \in \Sigma^*$ peut ou ne pas avoir, on considère le langage $L_P = \{w \in \Sigma^* : w \text{ a la propriété } P\}$, et inversement, si $L \subseteq \Sigma^*$ est un langage, on considère la propriété « appartenir à L » : en identifiant la propriété et le langage qu'on vient d'associer l'un à l'autre (par exemple, le langage des mots commençant par a et la propriété « commencer par a »), un langage pourrait être considéré comme une propriété des mots.

(Ce qui précède n'a rien de spécifique aux langages : une partie d'un ensemble E quelconque peut être identifiée à une propriété que les éléments de E peuvent ou ne pas avoir, à savoir, appartenir à la partie en question.)

On évitera de faire cette identification pour ne pas introduire de complication, mais il est utile de la garder à l'esprit : par exemple, dans un langage de programmation fonctionnel, un « langage » au sens de ces notes peut être considéré comme une fonction (pure, c'est-à-dire, déterministe et sans effet de bord) prenant en entrée une chaîne de caractères et renvoyant un booléen.

1.3.4. Si L_1 et L_2 sont deux langages sur un même alphabet Σ (autrement dit, $L_1, L_2 \subseteq \Sigma^*$), on peut former les langages **union** $L_1 \cup L_2$ et **intersection** $L_1 \cap L_2$ qui sont simplement les opérations ensemblistes usuelles (entre parties de Σ^*).

Les opérations correspondantes sur les propriétés de mots sont respectivement le « ou logique » (=disjonction) et le « et logique » (=conjonction) : à titre d'exemple, sur $\Sigma = \{a, b\}$ si L_1 est le langage des mots commençant par a et L_2 le langage des mots finissant par b, alors $L_1 \cup L_2$ est le langage des mots commençant par a *ou bien* finissant par b, tandis que $L_1 \cap L_2$ est le langage des mots commençant par a *et* finissant par b.

1.3.5. Si L est un langage sur l'alphabet Σ, autrement dit $L \subseteq \Sigma^*$, on peut former le langage $\Sigma^* \setminus L$, parfois noté simplement $\overline{L}$ si ce n'est pas ambigu, dit **complémentaire** de L, et qui est simplement l'ensemble des mots sur Σ *n'appartenant pas* à L. L'opération correspondante sur les propriétés de mots est la négation logique.

À titre d'exemple, sur $\Sigma = \{a, b\}$, si L est le langage des mots commençant par a, alors $\overline{L}$ est le langage des mots ne commençant pas par a, c'est-à-dire, la réunion de $\{\varepsilon\}$ et du langage des mots commençant par b (car sur $\Sigma = \{a, b\}$, un mot ne commençant pas par a est vide ou bien commence par b).

1.3.6. Si L_1 et L_2 sont deux langages sur un même alphabet Σ (autrement dit, $L_1, L_2 \subseteq \Sigma^*$), on peut former le langage **concaténation** L_1L_2 : il est défini comme l'ensemble des mots w qui peuvent s'écrire comme concaténation d'un mot w_1 de L_1 et d'un mot w_2 de L_2, soit

$$\begin{aligned} L_1L_2 &:= \{w_1w_2 : w_1 \in L_1,\, w_2 \in L_2\} \\ &= \{w \in \Sigma^* : \exists w_1 \in L_1\, \exists w_2 \in L_2\, (w = w_1w_2)\} \end{aligned}$$

À titre d'exemple, sur l'alphabet $\Sigma = \{a, b, c, d\}$, si on a $L_1 = \{a, bb\}$ et $L_2 = \{bc, cd\}$ alors $L_1L_2 = \{abc, acd, bbbc, bbcd\}$.

1.3.7. Si L est un langage sur l'alphabet Σ, autrement dit $L \subseteq \Sigma^*$, et si $r \in \mathbb{N}$, on peut définir un langage L^r, par analogie avec 1.2.5, comme le langage $L^r = \{w_1 \cdots w_r : w_1, \ldots, w_r \in L\}$ constitué des concaténation de r mots appartenant à L, ou si on préfère, par récurrence :

— $L^0 = \{\varepsilon\}$,

— $L^{r+1} = L^rL$.

À titre d'exemple, sur l'alphabet $\Sigma = \{a, b, c, d\}$, si on a $L = \{a, bb\}$, alors $L^2 = \{aa, abb, bba, bbbb\}$ et $L^3 = \{aaa, aabb, abba, abbbb, bbaa, bbabb, bbbba, bbbbbb\}$.

Attention, L^r n'est pas le langage $\{w^r : w \in L\}$ constitué des répétitions r fois (w^r) des mots w de L : c'est le langage des concaténations de r mots appartenant à L *mais ces mots peuvent être différents*. À titre d'exemple, si $L = \{a, b\}$ alors L^r est le langage constitué des 2^r mots de longueur exactement r sur $\{a, b\}$, ce n'est pas l'ensemble à deux éléments $\{a^r, b^r\}$ constitué des seuls deux mots $a^r = aaa \cdots a$ et $b^r = bbb \cdots b$.

1.3.8. Si L est un langage sur l'alphabet Σ, on définit enfin l'**étoile de Kleene** L^* de L comme le

langage constitué des concaténations d'un nombre *quelconque* de mots appartenant à L, c'est-à-dire la réunion de tous les langages L^r mentionnés ci-dessus :

$$L^* := \bigcup_{r=0}^{+\infty} L^r = \bigcup_{r\in\mathbb{N}} L^r$$
$$= \{w_1 \cdots w_r : r \in \mathbb{N},\, w_1, \ldots, w_r \in L\}$$

À titre d'exemple, sur l'alphabet $\Sigma = \{a, b, c, d\}$, si on a $L = \{a, bb\}$, alors on a $L^* = \{\varepsilon, a, bb, aa, abb, bba, bbbb, aaa, aabb, abba, abbbb, bbaa, bbabb, bbbba, bbbbbb, \ldots\}$.

Comme ci-dessus, il faut souligner que les mots $w_1, \ldots, w_r$ concaténés n'ont pas à être égaux : notamment, $\{a, b\}^*$ est le langage constitué de tous les mots sur l'alphabet $\{a, b\}$, pas le langage $\{a\}^* \cup \{b\}^*$ constitué des mots obtenus en répétant la lettre a ou en répétant la lettre b.

On remarquera que la définition de L^* ci-dessus redonne bien, lorsqu'on l'applique à l'alphabet Σ lui-même (considéré comme langage des mots de longueur 1), l'ensemble Σ^* de tous les mots : la notation Σ^* est donc justifiée *a posteriori*.

Le mot vide appartient toujours à L^* (quel que soit L) puisque $L^0 = \{\varepsilon\}$ et qu'on peut prendre $r = 0$ ci-dessus (autrement dit, le mot vide est la concaténation de zéro mots de L).

On peut par ailleurs montrer que L^* est le plus petit (pour l'inclusion) langage M tel que $M = \{\varepsilon\} \cup LM$. Cette observation sera implicite par exemple dans 4.2.3 ci-dessous, et motive aussi la construction L^+ qui suit.

1.3.9. On introduit parfois la notation $L^+ := \bigcup_{r=1}^{+\infty} L^r = \{w_1 \cdots w_r : r > 0,\, w_1, \ldots, w_r \in L\}$ pour l'ensemble des mots formés par concaténation d'un nombre *non nul* de mots de L. Lorsque le mot vide ε n'appartient pas déjà à L, ce langage L^+ diffère de L^* seulement en ce qu'il ne contient pas ε ; tandis que si ε appartient déjà à L, alors L^+ est égal à L^*. En toute généralité, on a $L^+ = LL^*$.

1.3.10. En rappelant la définition du mot miroir faite en 1.2.10, si L est un langage sur l'alphabet Σ, on définit le langage miroir L^{R} comme l'ensemble des mots miroirs des mots de L, c'est-à-dire $L^{\mathsf{R}} = \{w^{\mathsf{R}} : w \in L\}$.

1.3.11. De même que l'ensemble des mots sur un alphabet Σ admet une notation, à savoir Σ^*, on peut introduire une notation pour l'ensemble de tous les *langages* (= ensembles de mots) sur Σ : ce sera $\mathscr{P}(\Sigma^*)$. Il s'agit d'un cas particulier de la construction « ensemble des parties » (si E est un ensemble, $\mathscr{P}(E) = \{A : A \subseteq E\}$ est l'ensemble de tous les sous-ensembles de A ; c'est un axiome de la théorie des ensembles qu'un tel ensemble existe bien). On pourrait donc écrire « $L \in \mathscr{P}(\Sigma^*)$ » comme synonyme de « $L \subseteq \Sigma^*$ » ou de « L est un langage sur Σ » ; on évitera cependant de le faire, car cette notation est plus lourde qu'utile.

Il sera marginalement question dans ces notes de « classes de langages » : une classe de langages est un ensemble de langages (c'est-à-dire une partie de $\mathscr{P}(\Sigma^*)$, ou si on préfère un élément de $\mathscr{P}(\mathscr{P}(\Sigma^*))$). On ne développera pas de théorie générale des classes de langages et on n'en parlera pas de façon systématique, mais on parlera de certaines classes de langages importantes : la classe des langages rationnels ou reconnaissables (§1.4 à §3), la classe des langages algébriques (§4), la classe des langages décidables (§5) et la classe des langages semi-décidable (*ibid.*).

1.4 Langages rationnels et expressions rationnelles

1.4.1. Introduction : Les langages qui vont jouer le rôle le plus important dans ces notes sont les langages dits « rationnels » (qui seront définis de 1.4.2 à 1.4.4 ci-dessous, et qui s'avéreront être les mêmes que les langages « reconnaissables » par automates finis définis en 2.1.6).

Pour donner un premier aperçu informel de ce dont il s'agit avant de passer à une définition précise, commençons par donner quelques exemples, disons, sur l'alphabet $\Sigma = \{a, b, c, d\}$, de langages rationnels :

— le langage constitué des mots contenant au moins un d ;
— le langage constitué des mots ne contenant aucun d ;
— le langage constitué des mots commençant par un d ;
— le langage constitué des mots commençant par un d et finissant par un c ;
— le langage constitué des mots commençant par un d et finissant par un c, et dont toutes les lettres intermédiaires sont des a ou des b, c'est-à-dire, le langage constitué des mots de la forme suivante : un d, puis un nombre quelconque de a et de b, et enfin un c ;
— le langage constitué des mots de la forme suivante : un d, puis un nombre quelconque de mots qui peuvent être soit a soit bb, et enfin un c.

Ce dernier exemple est assez typique : on dira plus loin qu'il s'agit du langage « dénoté » par l'expression rationnelle $d(a|bb)*c$, à lire comme « un d, puis autant qu'on veut ("$*$") de mots qui peuvent être soit ("|") un a soit bb, et enfin un c ». Les constructions essentielles qui permettront de fabriquer les langages rationnels sont : la réunion (j'autorise ceci *ou* cela, par exemple « un a *ou bien* bb », cf. 1.3.4), la concaténation (je demande ceci *puis* cela, par exemple « un d, *puis* une suite quelconque de a ou de b », cf. 1.3.6), et l'étoile de Kleene (représentant une répétition quelconque d'un certain motif, cf. 1.3.8).

L'importance des langages rationnels, et des expressions rationnelles (=régulières) qui les décrivent, vient :

— du point de vue théorique : de ce qu'ils forment une classe à la fois suffisamment simple pour pouvoir être étudiée, suffisamment riche pour contenir toutes sortes de langages intéressants, suffisamment naturelle pour être définie de plusieurs manières différentes, et suffisamment flexible pour être stable un certain nombre d'opérations ;
— du point de vue pratique et informatique : de ce qu'ils sont algorithmiquement commodes à manier (grâce à la théorie) et suffisent à représenter beaucoup de « recherches » qu'on a naturellement envie de faire dans un texte, de « motifs » qu'on peut vouloir trouver, ou de « contraintes » qu'on peut vouloir imposer à une chaîne de caractères.

Passons maintenant à une définition plus précise.

1.4.2. Soit Σ un alphabet. On va considérer les langages de base triviaux suivants :

— le langage vide $\varnothing$,
— le langage constitué du seul mot vide, $\{\varepsilon\}$, et
— les langages constitués d'un seul mot lui-même formé d'une seule lettre, $\{x\}$ pour chaque $x \in \Sigma$,

et on va les combiner par les opérations dites « rationnelles » suivantes :

— la réunion $(L_1, L_2) \mapsto L_1 \cup L_2$ (discutée en 1.3.4),
— la concaténation $(L_1, L_2) \mapsto L_1 L_2$ (définie en 1.3.6), et
— l'étoile de Kleene $L \mapsto L^*$ (définie en 1.3.8).

On obtient ainsi une certaine famille de langages (cf. ci-dessous pour une définition plus précise), qu'on appelle **langages rationnels** : les langages rationnels sont par définition exactement ceux qui peuvent s'obtenir à partir des langages de base énumérés ci-dessus par application (un nombre *fini* de fois) des opérations qu'on vient de dire.

Autrement dit, la réunion de deux langages rationnels, la concaténation de deux langages rationnels, et l'étoile de Kleene d'un langage rationnel, sont rationnels ; et les langages rationnels sont exactement ceux qu'on obtient ainsi à partir des langages de base.

À titre d'exemple, sur l'alphabet $\{a, b, c, d\}$, comme le langage $\{c\}$ (constitué du seul mot c) est rationnel, son étoile de Kleene, c'est-à-dire $\{c\}^* = \{\varepsilon, c, cc, ccc, cccc, \ldots\}$, est rationnel, et comme $\{d\}$ l'est aussi, la concaténation $\{d\}(\{c\}^*) = \{d, dc, dcc, dccc, \ldots\}$ est encore un langage rationnel.

1.4.3. Formellement, la définition des langages rationnelles est la suivante : un ensemble $\mathscr{C} \subseteq \mathscr{P}(\Sigma^*)$ de langages (où $\mathscr{P}(\Sigma^*)$ est l'ensemble des parties de Σ^*, i.e., l'ensemble de tous les langages sur Σ, cf. 1.3.11) est dit *stable par opérations rationnelles* lorsqu'il est stable par les opérations de réunion, concaténation et étoile de Kleene, i.e., si $L_1, L_2 \in \mathscr{C}$ alors $L_1 \cup L_2 \in \mathscr{C}$ et $L_1 L_2 \in \mathscr{C}$, et si $L \in \mathscr{C}$ alors $L^* \in \mathscr{C}$; le *plus petit*[1] (pour l'inclusion) ensemble de langages stable par opérations rationnelles et contenant les langages $\varnothing$, $\{\varepsilon\}$ et $\{x\}$ pour $x \in \Sigma$ (i.e. $\varnothing \in \mathscr{C}$, $\{\varepsilon\} \in \mathscr{C}$ et si $x \in \Sigma$ alors $\{x\} \in \mathscr{C}$), ou plus exactement, l'intersection de tous les ensembles $\mathscr{C}$ vérifiant tous ces propriétés, est la classe $\mathscr{R}$ des langages rationnels (et un langage rationnel est simplement un élément de $\mathscr{R}$).

Attention !, le fait que la classe $\mathscr{R}$ des langages rationnels soit stable par concaténation signifie que si L_1 et L_2 sont rationnels alors le langage $L_1 L_2$ (constitué de tous les mots concaténés d'un mot de L_1 et d'un mot de L_2) est rationnel ; *cela ne signifie pas* qu'un langage rationnel donné soit stable par concaténation (un langage stable L par concaténation est un langage tel que si $w_1, w_2 \in L$ alors $w_1 w_2 \in L$).

1.4.4. Pour décrire la manière dont un langage rationnel est fabriqué (à partir des langages de base par les opérations rationnelles), comme il est malcommode d'écrire quelque chose comme $\{d\}(\{c\}^*)$, on introduit un nouvel objet, les **expressions rationnelles** (certains préfèrent le terme d'**expressions régulières**), qui sont des expressions servant à dénoter un langage rationnel. Par exemple, plutôt que d'écrire « $\{d\}(\{c\}^*)$ », on parlera du langage « dénoté par l'expression rationnelle $dc*$ ». Ceci fournit du même coup une nouvelle définition des langages rationnels : ce sont les langages dénotés par une expression rationnelle.

Plus exactement, une expression rationnelle (sur un alphabet Σ) est un mot sur l'alphabet $\Sigma \cup \{\bot, \underline{\varepsilon}, (,), |, *\}$, où $\bot, \underline{\varepsilon}, (,), |, *$ sont de nouveaux caractères *n'appartenant pas* à l'alphabet Σ, appelés **métacaractères**, et qui servent à marquer la manière dont est formée l'expression rationnelle. On définit simultanément la notion d'expression rationnelle r et de **langage dénoté** (ou **désigné** ou simplement **défini**) par l'expression r, noté $L(r)$ (ou L_r), de la manière suivante[2] :

— $\bot$ est une expression rationnelle et son langage dénoté est $L(\bot) := \varnothing$,
— $\underline{\varepsilon}$ est une expression rationnelle et son langage dénoté est $L(\underline{\varepsilon}) := \{\varepsilon\}$,
— si $x \in \Sigma$ est une lettre de l'alphabet Σ, alors le mot x est une expression rationnelle et son langage dénoté est $L(x) := \{x\}$,

1. La classe des langages rationnelles (qu'on cherche à définir) n'est pas le seul ensemble de langages stable par opérations rationnelles : l'ensemble $\mathscr{P}(\Sigma^*)$ de tous les langages est aussi évidemment stable par opérations rationnelles ; on s'intéresse au plus petit $\mathscr{C}$ possible pour n'avoir que ce qu'on peut construire à partir des langages de base triviaux par un nombre fini d'opérations rationnelles.

2. Si on veut être tout à fait rigoureux, il faudrait démontrer, pour que cette définition ait un sens, qu'une expression rationnelle r ne désigne qu'un seul langage $L(r)$, c'est-à-dire, qu'il n'y a qu'une seule façon de la lire : pour cela, il faudrait faire appel aux techniques qui seront introduites en §4 et dire que la grammaire que nous définissons ici pour les expressions rationnelles est *inambiguë* (cf. 4.5.1).

— si r_1, r_2 sont deux expressions rationnelles et $L_1 = L(r_1)$ et $L_2 = L(r_2)$ les langages dénotés correspondants, alors $r_1 r_2$ est une expression rationnelle et son langage dénoté est $L(r_1 r_2) := L_1 L_2$,
— si r_1, r_2 sont deux expressions rationnelles et $L_1 = L(r_1)$ et $L_2 = L(r_2)$ les langages dénotés correspondants, alors $(r_1|r_2)$ est une expression rationnelle [3] et son langage dénoté est $L((r_1|r_2)) := L_1 \cup L_2$,
— si r est une expression rationnelle et $L = L(r)$ les langage dénoté correspondant, alors $(r)*$ est une expression rationnelle [4] et son langage dénoté est $L((r)*) := L^*$.

Un langage rationnel est par construction la même chose qu'un langage pour lequel il existe une expression rationnelle qui le dénote.

1.4.5. À titre d'exemple, sur l'alphabet $\Sigma = \{a, b, c, d\}$, c est une expression rationnelle qui dénote le langage $\{c\}$, donc $(c)*$ en est une qui dénote le langage $\{c\}^* = \{\varepsilon, c, cc, ccc, \ldots\}$, et enfin $d(c)*$ en est une qui dénote le langage $\{d, dc, dcc, \ldots\}$ des mots formés d'un d et d'une succession quelconques de c. Voici quelques autres exemples, toujours sur $\Sigma = \{a, b, c, d\}$:
— l'expression rationnelle $(a|b)$ dénote le langage $\{a\} \cup \{b\} = \{a, b\}$ constitué des deux mots d'une seule lettre a et b ;
— l'expression rationnelle $(a|b)c$ dénote le langage $\{a, b\}\{c\} = \{ac, bc\}$, de même que $(ac|bc)$;
— l'expression rationnelle $(bc)*$ dénote le langage $\{bc\}^* = \{\varepsilon, bc, bcbc, bcbcbc, \ldots\}$;
— l'expression rationnelle $(a|(bc)*)$ dénote le langage $\{a\} \cup \{bc\}^* = \{a, \varepsilon, bc, bcbc, bcbcbc, \ldots\}$;
— l'expression rationnelle $(a|(bc)*)d$ dénote le langage $\{ad, d, bcd, bcbcd, bcbcbcd, \ldots\}$;
— l'expression rationnelle $\bot d$ dénote le langage vide $\varnothing$ (car il n'y a pas de mot dans le langage vide, donc pas non plus de mot dans sa concaténation avec le langage $\{d\}$) ;
— l'expression rationnelle $\underline{\varepsilon} d$ dénote le langage $\{d\}$;
— l'expression rationnelle $(\bot|c)$ dénote le langage $\{c\}$;
— l'expression rationnelle $(\underline{\varepsilon}|c)$ dénote le langage $\{\varepsilon, c\}$.

1.4.6. On dira qu'un mot w **vérifie** une expression rationnelle r lorsque ce mot appartient au langage qu'elle dénote (i.e., $w \in L(r)$). Par exemple, $dccc$ vérifie l'expression rationnelle $d(c)*$.

1.4.7. Deux expressions rationnelles r_1, r_2 sont dites **équivalentes** lorsqu'elles dénotent le même langage. À titre d'exemple, sur l'alphabet $\{a, b\}$, les deux expressions rationnelles $(ab)*a$ et $a(ba)*$ sont équivalentes (toutes les deux dénotent le langage $\{a, aba, ababa, \ldots\}$ constitué des mots commençant et finissant par un a et dans lesquels chaque paire de a est séparée par un unique b).

On verra plus loin (en 3.6.8) qu'on dispose d'un algorithme permettant de décider si deux expressions rationnelles sont équivalentes.

1.4.8. Voici quelques exemples d'équivalences d'expressions rationnelles (valables en remplaçant les différents r qui interviennent dedans par des expressions rationnelles quelconques) :

3. Certains préfèrent la notation $r_1 + r_2$ ici, cf. 1.4.10 pour une discussion.

4. Notons que certains auteurs préfèrent mettre le $*$ en exposant ici, c'est-à-dire noter $(r)^*$ (comme nous l'avons fait pour l'étoile de Kleene L^* d'un langage L) ; comme nous avons choisi de définir une expression rationnelle comme un mot sur l'alphabet $\Sigma \cup \{\bot, \underline{\varepsilon}, (,), |, *\}$, il semble plus raisonnable de ne pas rentrer dans la considération byzantine de savoir si un mot peut contenir des symboles en exposant. En tout état de cause, il n'apparaîtra jamais aucune circonstance où les deux notations X^* et $X*$ auraient tous les deux un sens et que ces sens diffèrent ! On peut donc ignorer purement et simplement la question de savoir si oui ou non l'étoile est en exposant.

— identités « triviales » : $(r|\bot) \equiv r$, $(\bot|r) \equiv r$, $r\bot \equiv \bot$, $\bot r \equiv \bot$, $r\underline{\varepsilon} \equiv r$, $\underline{\varepsilon}r \equiv r$, $(\bot)* \equiv \underline{\varepsilon}$;
— identité d'associativité : $((r_1|r_2)|r_3) \equiv (r_1|(r_2|r_3))$;
— identités de distributivité : $r(r_1|r_2) \equiv (rr_1|rr_2)$, $(r_1|r_2)r \equiv (r_1r|r_2r)$;
— identité de commutativité : $(r_1|r_2) \equiv (r_2|r_1)$;
— identités « apériodiques » : $((r_1|r_2))* \equiv (r_1)*(r_2(r_1)*)*$, $((r_1|r_2))* \equiv ((r_1)*r_2)*(r_1)*$ et $(r_1r_2)* \equiv (\underline{\varepsilon}|r_1(r_2r_1)*r_2)$;
— identités « cycliques » : $(r)* \equiv (\underline{\varepsilon}|r)(rr)* \equiv (\underline{\varepsilon}|r|rr)(rrr)* \equiv (\underline{\varepsilon}|r|rr|rrr)(rrrr)* \equiv \cdots$;
— identités d'« idempotence » : $(r|r) \equiv r$, $((r)*)* \equiv (r)*$;

Ces différentes identités présentent un intérêt théorique dont l'explication dépasserait le cadre de ces notes ; cependant, il peut être intéressant de réfléchir à ce que chacune signifie.

Signalons au passage que toutes ces identités *ne suffisent pas* à produire toutes les équivalences entre expressions rationnelles. Par exemple, elles ne permettent pas de démontrer l'équivalence suivante : $(a|b)* \equiv ((a|b)(b|ab*ab*ab*a))*(\underline{\varepsilon}|(a|b)(|ab*|ab*ab*|ab*ab*ab*))$ (écrite avec les conventions de 1.4.9) ; la question d'arriver à trouver un système d'axiomes qui permet de déduire toutes les équivalences entre expressions rationnelles est un problème délicat (il n'y en a notamment pas qui soit fini).

1.4.9. La convention de parenthésage introduite ci-dessus est inambiguë mais parfois inutilement lourde : on se permettra parfois de l'alléger, par exemple d'écrire $(r_1|r_2|r_3)$ pour $((r_1|r_2)|r_3)$ (ou pour $(r_1|(r_2|r_3))$, ce qui n'a guère d'importance vu qu'elles dénotent le même langage), ou encore $x*$ pour $(x)*$ lorsque x est formé d'un seul caractère.

Pour retirer des parenthèses, la convention sur la priorité des opérations est la suivante : l'opération d'étoile $*$ est la plus prioritaire (c'est-à-dire que $ab*$ se lit comme $a(b)*$ et non pas comme $(ab)*$), la concaténation est de priorité intermédiaire, et la barre de disjonction $|$ est la moins prioritaire (c'est-à-dire que $ab|cd$ se lit comme $(ab|cd)$ et pas comme $a(b|c)d$).

1.4.10. La disjonction que nous avons notée « $|$ » est plus souvent notée « $+$ » par les mathématiciens [5]. Cette notation est plus élégante pour faire le lien avec l'algèbre, et plus logique dans des cadres plus généraux (expressions rationnelles et automates « avec multiplicités ») ; nous avons cependant préféré l'éviter parce qu'elle prête à confusion : en effet, on a introduit en 1.3.9 une opération $L \mapsto L^+$ qui n'a rien à voir avec la disjonction, et de nombreux moteurs d'expressions régulières en informatique (par exemple, `egrep`, cf. §1.5) utilisent effectivement le symbole + dans ce sens, c'est-à-dire pour désigner « au moins une répétition » (cf. 1.5.3). Même si nous avons cherché à ne pas utiliser le symbole $+$ pour éviter l'ambiguïté, il faut savoir que ce double sens existe (soit comme disjonction, soit comme l'opération de 1.3.9).

Les métacaractères $\bot$ et $\underline{\varepsilon}$ sont introduits ici par souci de complétude mais sont rarement utilisés dans les expressions rationnelles (le métacaractère $\underline{\varepsilon}$ a été souligné parce qu'il s'agit d'une vraie lettre et non pas du mot vide ; on peut ignorer cette subtilité qui n'a que très peu d'importance).

1.5 Remarques sur les moteurs d'expressions régulières en informatique

1.5.1. Dans le monde informatique, il existe de nombreux *moteurs d'expressions régulières*, c'est-à-dire outils (qu'il s'agisse de primitives d'un langage, de bibliothèques externes, de programmes en ligne de commande, ou autres) permettant de savoir si un mot est reconnu par une expression régulière (=rationnelle), autrement dit, s'il appartient au langage dénoté par

5. Dans le même contexte mathématique, il est alors fréquent de noter « 0 » pour ce que nous avons noté « $\bot$ » (c'est un élément neutre pour la disjonction), et on en profite souvent pour noter « 1 » pour « ε » et/ou « $\underline{\varepsilon}$ » (c'est un élément neutre pour la concaténation). Par exemple, beaucoup des identités énoncées en 1.4.8 sembleront plus naturelles si on les écrit avec ces notations.

elle. L'un de ces moteurs est le programme `egrep` standard sous Unix/POSIX ; un autre est le moteur `java.util.regex` de Java.

1.5.2. Les expressions régulières au sens de ces différents moteurs sont généralement plus puissantes que les expressions rationnelles au sens mathématique défini en 1.4.4 ci-dessus : différentes extensions permettent de désigner des langages qui ne sont pas rationnels au sens mathématique.

L'extension la plus fréquente est celle des *références arrière* (ou *backreferences* en anglais) qui permettent de demander qu'un facteur du mot se retrouve aussi à un autre emplacement. Par exemple, pour beaucoup de moteurs (notamment `egrep`), l'expression régulière « `(a*)b\1` » désigne le langage $\{a^n b a^n \,:\, n \in \mathbb{N}\} = \{b, aba, aabaa, \ldots\}$ des mots formés d'un nombre quelconque de a puis d'un b puis de la *même suite de* a (le « `\1` » désigne « une copie de la chaîne de caractères qui a été capturée par le premier jeu de parenthèses ») ; or ce langage *n'est pas rationnel* au sens mathématique (ce sera une conséquence du « lemme de pompage » 3.5.2), c'est donc bien le signe qu'on sort du cadre théorique décrit plus haut.

1.5.3. Il existe aussi un certain nombre de constructions qui, sans dépasser la puissance des expressions rationnelles au sens mathématique, apportent des commodités d'écriture dans un contexte informatique.

Parmi les caractères les plus courants de cette sorte, mentionnons le + qui signifie « au moins une répétition » (alors que $\star$ signifie « au moins zéro répétitions ») : ainsi, « r+ » est équivalent à $rr\star$; on trouve aussi `?` pour « au plus une répétition » : c'est-à-dire que « r`?` » est équivalent à $(\underline{\varepsilon}|r)$ (qui sera d'ailleurs simplement noté $(|r)$ car il n'y a jamais de symbole spécial pour $\underline{\varepsilon}$).

Parmi d'autres constructions qui ne sortent pas du cadre théorique des langages rationnels, mentionnons encore le point « `.` » qui désigne un caractère quelconque de l'alphabet (on peut le voir comme une abréviation pour $(x_1|x_2|\ldots|x_N)$ où $x_1, x_2, \ldots, x_N$ sont tous les éléments de Σ — ce qui serait très fastidieux à écrire si on devait énumérer tout Unicode). Les crochets tels que « `[`xyz`]` » désignent un caractère quelconque parmi ceux listés (c'est donc la même chose que $(x|y|z)$ mais cela ne fonctionne qu'avec des caractères individuels ; en contrepartie, on peut écrire des intervalles comme « `[a-z]` » qui désigne un caractère quelconque entre `a` et `z` dans l'ordre ASCII/Unicode [6], ou bien des négations, comme « `[^a-z]` » qui désigne un caractère qui *n'est pas* entre `a` et `z`) ou « `[^aeio-z]` » qui désigne un caractère qui *n'est ni* `a` ni `e` ni`i` ni entre `o` et `z`).

Toutes sortes d'autres raccourcis ou commodités de notation peuvent exister, par exemple `\<` et `\>` pour désigner un début et une fin de mot (la définition précise de « mot » pouvant varier), ou encore r{n_1`,`n_2`}` qui cherche entre n_1 et n_2 répétitions de r : ainsi, « r{`2,5``}` » est équivalent à $(rr|rrr|rrrr|rrrrr)$.

1.5.4. Une autre subtilité est que la plupart des moteurs d'expressions régulières en informatique vont, par défaut, *rechercher un facteur* (appelé « sous-chaîne » en informatique) vérifiant l'expression *à l'intérieur* de la chaîne donnée, plutôt que tester si la chaîne tout entière vérifie l'expression. Pour éviter ce comportement, on peut utiliser des *ancres*, typiquement commandées par les (méta)caractères « `^` » et « `$` » qui servent à ancrer l'expression au début et à la fin de la chaîne respectivement : c'est-à-dire que rechercher « `a` » recherche un `a` quelconque à l'intérieur de la chaîne donnée, rechercher « `^a` » demande que le `a` soit au début de la chaîne donnée rechercher « `a$` » demande que le `a` soit à la fin de la chaîne donnée, et rechercher

6. ...ou parfois l'ordre de tri défini par la « locale » en cours : c'est malheureusement une source supplémentaire de confusion et de bugs.

« `^a$` » demande que la chaîne donnée soit exactement `a` (cet exemple n'est donc pas très utile, mais de façon générale, trouver si une chaîne vérifie une expression rationnelle r, revient à y chercher « `^`r`$` »).

1.5.5. Comme les expressions régulières en informatique sont représentées par des chaînes de caractères qui appartiennent au même alphabet (ASCII ou Unicode) que les chaînes sur lesquelles on effectue la recherche, le problème se pose de distinguer les métacaractères (l'étoile de Kleene `*`, par exemple) des caractères eux-mêmes (comment rechercher les chaînes contenant le caractère « `*` » si `*` est utilisé par l'étoile de Kleene ?). La solution est d'introduire un mécanisme d'*échappement*, normalement par le caractère *backslash* : ainsi, « `x\*` » recherche un `x` suivi d'un astérisque `*`, tandis que « `x*` » recherche un nombre quelconque de répétitions de la lettre `x`.

1.5.6. Il existe malheureusement de nombreuses différences, parfois très subtiles, entre moteurs, ne serait-ce que dans les notations : un moteur pourra par exemple noter « `(?)` » ce qu'un autre note « `\(\?\)` » et vice versa. La seule solution est de consulter attentivement la documentation de chaque moteur d'expressions régulières pour connaître la syntaxe utilisée.

Signalons tout de même qu'il existe deux principales familles de syntaxes d'expressions régulières en informatique : les expressions régulières « POSIX étendues », utilisée notamment par le programme Unix `egrep`, et les expressions régulières de Perl, qui ont été réadaptées dans beaucoup de langages, notamment Java, JavaScript, Python et d'autres.

1.5.7. Signalons comme complication supplémentaire que dans de nombreux langages, les expressions régulières sont saisies comme des chaînes de caractères plutôt que d'avoir une syntaxe spéciale, et ceci a pour effet d'introduire un niveau supplémentaire d'échappement : par exemple, en Java, pour rechercher si une chaîne de caractères s contient un astérisque, on utilisera s`.matches("\\*")` puisque l'expression régulière à utiliser est `\*` et que cette chaîne de caractères s'écrit `"\\*"` en Java.

De même, dans le shell Unix, pour rechercher une ligne d'un fichier contenant un `x` suivi d'un nombre quelconque de `y` suivi d'un `z`, on utilisera `egrep 'xy*z' monfichier`, les guillemets (simples) ayant pour fonction d'éviter que le shell Unix interprète lui-même l'astérisque comme un caractère spécial (dont le rôle est différent de l'étoile de Kleene, quoique suffisamment proche pour pouvoir causer une confusion).

2 Automates finis

2.0.1. Les automates finis sont un modèle de calcul particulièrement simple et particulièrement appropriés à l'étude et à l'analyse des langages rationnels. Il faut imaginer un automate fini comme une machine disposant d'une quantité finie (et sous-entendu, très limitée) de mémoire : la configuration complète de cette mémoire est décrite par un *état*, qui appartient à un ensemble fini d'états possibles. On fournit un mot à l'automate en le lui faisant consommer lettre par lettre (de la première à la dernière), et à chaque lettre qu'il reçoit, l'automate prend une décision (à savoir, passer dans un nouvel état) en fonction de son état actuel et de la lettre qu'on lui donne à consommer ; l'automate commence par un état dit *initial* ; lorsque le mot a été entièrement consommé, l'état dans lequel l'automate se trouve a pour conséquence soit que l'automate *accepte* le mot (s'il se trouve dans un état *final*), soit qu'il le *rejette*. L'ensemble des mots

acceptés par l'automate est un langage dit *reconnaissable*. On va voir en 3.4.11 que ces langages reconnaissables sont en fait les mêmes que les langages rationnels définis en 1.4.2.

2.0.2. Une subtilité est qu'il existe plusieurs sortes d'automates finis. Nous allons en définir quatre ou cinq, de plus en plus généraux, mais nous allons voir qu'ils sont, finalement, tous équivalents, c'est-à-dire que leur puissance de calcul, ou les langages qu'ils sont capables de reconnaître, sont toujours les mêmes (en revanche, cette équivalence peut avoir un coût algorithmique, car la conversion d'une sorte d'automate en une autre n'est pas gratuite). Du plus particulier au plus général, les automates que nous allons définir sont informellement décrits ainsi :

- les automates finis déterministes (complets) (cf. 2.1.1) : ceux-ci partent d'un état initial bien défini, suivent à chaque lettre consommée une transition bien définie vers un nouvel état, et leur comportement est donc entièrement déterministe (d'où le nom) ;
- les automates finis déterministes à spécification incomplète (cf. 2.2.1) : la différence est qu'ici l'automate n'a pas forcément d'instruction sur quoi faire quand il est dans un certain état et reçoit un certain symbole, il peut manquer des transitions, auquel cas l'automate cesse de fonctionner et rejette le mot ;
- les automates non-déterministes (cf. 2.3.2) : ceux-ci peuvent admettre *plusieurs* possibilités de nouvel état dans lequel transitionner lorsqu'on leur donne une lettre donnée à consommer depuis un état donné, et l'automate accepte le mot s'il y a une manière de faire les transitions qui résulte en ce que le mot soit accepté ;
- les automates non-déterministes à transitions spontanées (cf. 2.4.1) : ceux-ci ont, en plus des précédents, le pouvoir d'évoluer spontanément, sans consommer de lettre ;
- on introduira aussi des automates encore plus généraux, à transitions étiquetées par des expressions rationnelles (cf. 3.4.2) pour montrer que tout langage reconnaissable est rationnel.

2.0.3. Dans tous les cas, les automates se représenteront comme des graphes orientés (admettant des boucles et arêtes multiples ; cf. 2.1.3 ci-dessous pour un exemple simple), dont les sommets sont les états de l'automate (on leur donne généralement des noms pour les reconnaître, mais ces noms sont arbitraires), dont les arêtes sont étiquetées par des symboles (des lettres de l'alphabet, ou éventuellement le mot vide ε ou une expression rationnelle sur l'alphabet), et dont certains sommets (=états) sont distingués comme « initiaux » et/ou « finaux » (par une flèche pointant depuis nulle part vers l'état en question ou depuis l'état en question vers nulle part).

Un mot sera accepté par l'automate toujours selon la même logique : s'il y a moyen de relier un état initial à un état final par un chemin orienté dans le graphe tel que le mot corresponde à la lecture des étiquettes des arêtes du chemin dans l'ordre.

2.1 Automates finis déterministes complets (=DFA)

2.1.1. Un **automate fini déterministe** (complet), ou en abrégé **DFA** (pour *deterministic finite automaton*), sur un alphabet Σ est la donnée

- d'un ensemble fini Q dont les éléments sont appelés **états**,
- d'un état $q_0 \in Q$ appelé **état initial** (un DFA a un *unique* état initial),
- d'un ensemble $F \subseteq Q$ d'états appelés états **finaux**[7] ou **acceptants**,
- d'une fonction $\delta \colon Q \times \Sigma \to Q$ appelée **fonction de transition**.

7. Le pluriel de « final » est indifféremment « finaux » ou « finals ».

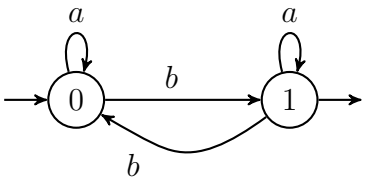
2.1.2. Graphiquement, on représente un DFA comme un graphe orienté aux arêtes étiquetées par des éléments de Σ : plus exactement, on trace un nœud pour chaque élément $q \in Q$, et lorsque $\delta(q, x) = q'$ on introduit une flèche $q \to q'$ étiquetée par la lettre x.

La condition cruciale (pour être un DFA) est que, pour chaque état $q \in Q$ et chaque lettre $x \in \Sigma$, *il existe une unique* arête partant de q et étiquetée par x (c'est essentiellement une reformulation du fait que δ est une fonction).

En outre, on introduit une flèche pointant de nulle part vers q_0 (pour marquer celui-ci comme l'état initial), et pour chaque $q \in F$ une flèche pointant de q vers nulle part [8] (pour marquer ceux-ci comme états finaux).

Pour abréger la représentation graphique, lorsque plusieurs arêtes étiquetées par des lettres x, y différentes relient les mêmes sommets q, q' (i.e., lorsqu'on a à la fois $\delta(q, x) = q'$ et $\delta(q, y) = q'$), on pourra écrire « x, y », voire « $x|y$ », sur l'arête en question et ne la tracer qu'une seule fois. (Voir 2.1.7 ci-dessous pour un exemple.)

2.1.3. Pour donner un exemple simple, l'automate sur $\Sigma = \{a, b\}$ représenté ci-dessous a $Q = \{0, 1\}$ et $q_0 = 0$ et $F = \{1\}$ et la fonction de transition δ donnée par $\delta(0, a) = 0$ et $\delta(0, b) = 1$ et $\delta(1, a) = 1$ et $\delta(1, b) = 0$. On pourra se convaincre (une fois lues les définitions plus loin) que cet automate accepte les mots dont le nombre de b est impair.



2.1.4. Il faut comprendre le fonctionnement d'un DFA de la manière suivante : initialement, l'automate est dans l'état initial q_0. On va lui présenter un mot $w \in \Sigma^*$, lettre par lettre, de la gauche vers la droite : i.e., si $w = x_1 \cdots x_n$ on va faire consommer à l'automate les lettres $x_1, x_2, \ldots, x_n$ dans cet ordre. Le fait de consommer une lettre x fait passer l'automate de l'état q à l'état $\delta(q, x)$; autrement dit, l'automate passe successivement dans les états q_0 puis $q_1 := \delta(q_0, x_1)$ puis $q_2 := \delta(q_1, x_2)$, et ainsi de suite jusqu'à $q_n := \delta(q_{n-1}, x_n)$: on dit que l'automate effectue les transitions $q_0 \to q_1$ (en consommant x_1) puis $q_1 \to q_2$ (en consommant x_2) et ainsi de suite. Si q_n est l'état dans lequel se trouve l'automate une fois qu'il a consommé le mot w, on dira que l'automate *accepte* ou *rejette* le mot selon que $q_n \in F$ ou que $q_n \notin F$.

Graphiquement, on peut présenter la procédure de la manière suivante : on part de l'état q_0 (sommet du graphe représentant l'automate) indiqué par la flèche entrante (pointant de nulle part), et pour chaque lettre du mot $w = x_1 \cdots x_n$ considéré, on suit l'arête portant cette lettre pour étiquette (et partant de l'état où on se trouve actuellement). Si à la fin l'état q_n est acceptant (représenté par une flèche pointant vers nulle part), le mot w est accepté, sinon il est rejeté.

Cela revient encore à dire que le mot w est accepté lorsqu'il existe un chemin orienté dans l'automate, reliant l'état q_0 initial à un état q_n final, et tel que le mot $w = x_1 \cdots x_n$ soit obtenu en lisant dans l'ordre les étiquettes x_i des différentes arêtes $q_{i-1} \to q_i$ de ce chemin. Cette définition pourra resservir presque à l'identique pour les autres sortes d'automates, la spécificité

8. Certains auteurs préfèrent d'autres conventions, par exemple celle consistant à entourer deux fois les états finaux.

des DFA est que l'état initial q_0 est unique et que connaissant l'étiquette x_i il n'y a qu'une seule transition $q_{i-1} \to q_i$ possible (il n'y a pas de choix à faire, pas de non-déterminisme).

2.1.5. Pour donner de façon plus formelle la définition du langage accepté par un automate, il sera utile d'introduire une fonction δ^* de transition multiple (une généralisation de δ), qui définit l'état $\delta^*(q, w)$ dans lequel aboutit un DFA si à partir de l'état q on lui fait consommer successivement les lettres d'un mot w.

Formellement : si $A = (Q, q_0, F, \delta)$ est un DFA sur l'alphabet Σ, on définit une fonction $\delta^* \colon Q \times \Sigma^* \to Q$ par $\delta^*(q, x_1 \cdots x_n) = \delta(\cdots \delta(\delta(q, x_1), x_2) \cdots, x_n)$ ou, ce qui revient au même (par récurrence sur la longueur du second argument) :

— $\delta^*(q, \varepsilon) = q$ quel que soit $q \in Q$ (où ε désigne le mot vide),

— $\delta^*(q, wx) = \delta(\delta^*(q, w), x)$ quels que soient $q \in Q$, $w \in \Sigma^*$ et $x \in \Sigma$,

(en particulier, $\delta^*(q, x) = \delta(q, x)$ si $x \in \Sigma$, donc avec la convention faite en 1.1.7, on peut dire que δ^* prolonge δ ; il sera par ailleurs utile de remarquer que $\delta^*(q, ww') = \delta^*(\delta^*(q, w), w')$, ce qui se démontre par récurrence).

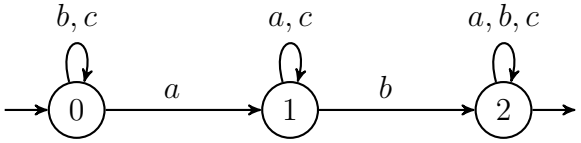
Cette fonction δ^* étant définie, on dira que l'automate A **accepte** ou **reconnaît** un mot w lorsque $\delta^*(q_0, w) \in F$; dans le cas contraire, on dira qu'il **rejette** le mot w.

2.1.6. L'ensemble $L(A)$ (ou L_A) des mots acceptés par l'automate A s'appelle **langage accepté**, ou **reconnu**, ou **défini**, par l'automate A.

Un langage $L \subseteq \Sigma^*$ qui peut s'écrire sous la forme du langage $L(A)$ accepté par un DFA A s'appelle **reconnaissable** (sous-entendu : par automate déterministe fini).

On dit que deux DFA A, A' sont **équivalents** lorsqu'ils reconnaissent le même langage, i.e., $L(A) = L(A')$.

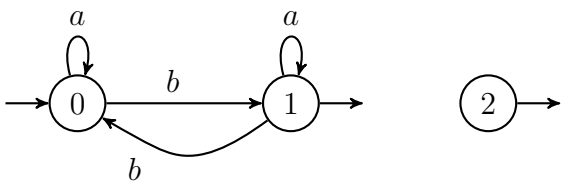
2.1.7. Donnons un exemple : l'automate fini ci-dessous sur $\Sigma := \{a, b, c\}$ a trois états, $Q = \{0, 1, 2\}$. On peut en faire la description informelle suivante : l'automate commence dans l'état 0, où il reste jusqu'à rencontrer un a qui le fait passer dans l'état 1, où il reste ensuite jusqu'à rencontrer un b qui le fait passer dans l'état 2, où il reste définitivement et qui constitue le seul état acceptant.



Cette description rend clair le fait que l'automate en question accepte exactement les mots contenant un a suivi, pas forcément immédiatement, d'un b ; autrement dit, les mots dont ab est un sous-mot (cf. 1.2.8). Ce langage est donc reconnaissable. (Il est aussi rationnel puisque dénoté par l'expression rationnelle $(b|c)*a(a|c)*b(a|b|c)*$.)

2.1.8. Un état q d'un DFA est dit **accessible** lorsqu'il existe un mot $w \in \Sigma^*$ tel que $q = \delta(q_0, w)$, autrement dit, graphiquement, lorsqu'il existe un chemin orienté $q_0, q_1, \ldots, q_n = q$ reliant l'état initial q_0 à l'état q considéré : bref, cela correspond à un état auquel il est possible que l'automate arrive (en partant de l'état initial et en consommant un certain mot). Dans le cas contraire, l'état est dit **inaccessible**. Il est évident qu'ajouter ou supprimer (ou modifier) les états inaccessibles dans un DFA ne change rien au langage reconnu (on obtient des automates équivalents).

Par exemple, dans le DFA qui suit, l'état 2 est inaccessible (l'automate est donc équivalent à celui représenté en 2.1.3). On remarquera qu'il ne change rien que cet état soit final ou non.



2.1.9. On va maintenant introduire différentes variations sur le thème des automates finis, c'est-à-dire différentes généralisations de la définition faite en 2.1.1 correspondant à des types d'automates finis plus puissants que les DFA mais dont on va montrer, à chaque fois, qu'ils peuvent se ramener à des DFA au sens où pour chacun de ces automates généralisés on pourra construire algorithmiquement un DFA qui reconnaît le même langage (si bien que la classe des langages reconnaissables par n'importe laquelle de ces sortes d'automates sera toujours la même). Les plus simples sont les automates déterministes finis incomplets et on va donc commencer par eux.

2.2 Automates finis déterministes à spécification incomplète (=DFAi)

2.2.1. Un **automate fini déterministe à spécification incomplète** ou **...partielle**, ou simplement **automate fini déterministe incomplet**[9], en abrégé **DFAi**, sur un alphabet Σ est la donnée

— d'un ensemble fini Q d'états,
— d'un état initial $q_0 \in Q$,
— d'un ensemble $F \subseteq Q$ d'états finaux,
— d'une fonction de transition *partielle*[10] $\delta\colon Q \times \Sigma \dashrightarrow Q$,

autrement dit, la seule différence avec la définition faite en 2.1.1 est que la fonction δ est partielle, ce qui signifie qu'elle n'est pas obligatoirement définie sur tout couple $(q, x) \in Q \times \Sigma$.

Un DFA est considéré comme un DFAi particulier où la fonction de transition δ se trouve être définie partout.

2.2.2. Graphiquement, on représente un DFAi comme un DFA, à la différence près que pour chaque $q \in Q$ et chaque $x \in \Sigma$, il y a maintenant *au plus une* (et non plus exactement une) arête partant de q et étiquetée par x.

L'intérêt informatique des DFAi est de ne pas s'obliger à stocker inutilement des transitions et des états inutiles au sens où ils ne permettront jamais d'accepter le mot (voir la notion d'automate « émondé » en 2.2.9 ci-dessous). C'est la raison pour laquelle, même si les DFA complets sont théoriquement plus satisfaisants à manier (pour la même raison qu'une fonction totale est plus satisfaisante qu'une fonction partielle), il est souvent algorithmiquement plus judicieux de travailler sur des DFAi.

2.2.3. Le fonctionnement d'un DFAi est le même que celui d'un DFA, à la modification suivante près : si on donne à consommer à l'automate une lettre pour laquelle la transition n'est pas définie, i.e., s'il rencontre un x pendant qu'il se trouve dans un état q pour lequel $\delta(q, x)$ n'est

9. Le mot « incomplet » signifie en fait « non nécessairement complet », i.e., l'automate a le droit de manquer certaines transitions, mais il peut très bien être complet (un DFA est un DFAi particulier, pas le contraire).

10. Une « fonction partielle » $f\colon X \dashrightarrow Y$, où X, Y sont deux ensembles est, par définition, la même chose qu'une fonction $f\colon D \to Y$ où $D \subseteq X$ est un sous-ensemble de X appelé **ensemble de définition** de f. (Lorsque en fait $D = X$, la fonction est dite « totale ».)

pas défini, alors l'automate cesse de fonctionner : l'automate n'a plus d'état, n'effectue plus de transition, et n'acceptera pas le mot quelles que soient les lettres ultérieures.

Cela revient une fois de plus à dire que le mot w est accepté lorsqu'il existe un chemin orienté dans l'automate, reliant l'état q_0 initial à un état q_n final, et tel que le mot $w = x_1 \cdots x_n$ soit obtenu en lisant dans l'ordre les étiquettes x_i des différentes arêtes $q_{i-1} \to q_i$ de ce chemin. La différence des DFAi par rapport aux DFA est que, cette fois, la tentative de chemin pourrait s'arrêter prématurément (on ne peut plus consommer un symbole et passer dans un nouvel état, et à plus forte raison, aboutir à un état final).

2.2.4. Formellement : si $A = (Q, q_0, F, \delta)$ est un DFAi sur l'alphabet Σ, on définit une fonction $\delta^* \colon Q \times \Sigma^* \dashrightarrow Q$ par $\delta^*(q, x_1 \cdots x_n) = \delta(\cdots \delta(\delta(q, x_1), x_2) \cdots, x_n)$ avec la convention que dès qu'une sous-expression n'est pas définie, toute l'expression n'est pas définie, ou, ce qui revient au même (par récurrence sur la longueur du second argument) :

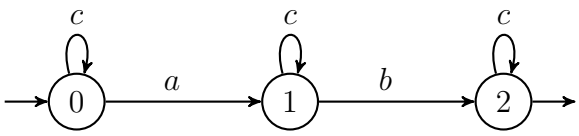
— $\delta^*(q, \varepsilon) = q$ quel que soit $q \in Q$ (où ε désigne le mot vide),

— $\delta^*(q, wx) = \delta(\delta^*(q, w), x)$ à condition que $q' := \delta^*(q, w)$ soit défini et que $\delta(q', x)$ le soit (et si ces deux conditions ne sont pas satisfaites, $\delta^*(q, wx)$ n'est pas défini).

Enfin, l'automate A accepte un mot w lorsque $\delta^*(q_0, w)$ ***est défini*** et appartient à F ; dans le cas contraire (que ce soit parce que $\delta^*(q_0, w)$ n'est pas défini ou parce qu'étant défini il n'appartient pas à F), l'automate rejette le mot.

Le langage accepté $L(A)$ et l'équivalence de deux automates sont définis de façon analogue aux DFA (cf. 2.1.6).

2.2.5. Voici un exemple de DFAi sur l'alphabet $\Sigma = \{a, b, c\}$. Cet automate reconnaît exactement les mots formés d'un nombre quelconque de c, suivis d'un a, suivis d'un nombre quelconque de c, suivis d'un b, suivis d'un nombre quelconque de c.



(Ce langage est aussi dénoté par l'expression rationnelle $c{*}ac{*}bc{*}$.)

Proposition 2.2.6. Soit $A = (Q, q_0, F, \delta)$ un DFAi sur un alphabet Σ. Alors il existe un DFA $A' = (Q', q'_0, F', \delta')$ (sur le même alphabet Σ) qui soit équivalent à A au sens où il reconnaît le même langage $L(A') = L(A)$. De plus, A' se déduit algorithmiquement de A en ajoutant au plus un état « puits » à A : on a $\#Q' \leq \#Q + 1$.

Démonstration. On définit $Q' = Q \cup \{q_\perp\}$ où $q_\perp$ est un nouvel état (n'appartenant pas à Q), qu'on appellera « puits ». On garde l'état initial $q'_0 = q_0$. On garde l'ensemble $F' = F$ d'états finaux, c'est-à-dire notamment que le puits n'est pas acceptant. Enfin, on définit $\delta'(q, x)$ pour $q \in Q'$ et $x \in \Sigma$ par

$$\delta'(q, x) = \delta(q, x) \text{ si } \delta(q, x) \text{ est défini}$$
$$\delta'(q, x) = q_\perp \text{ sinon}$$

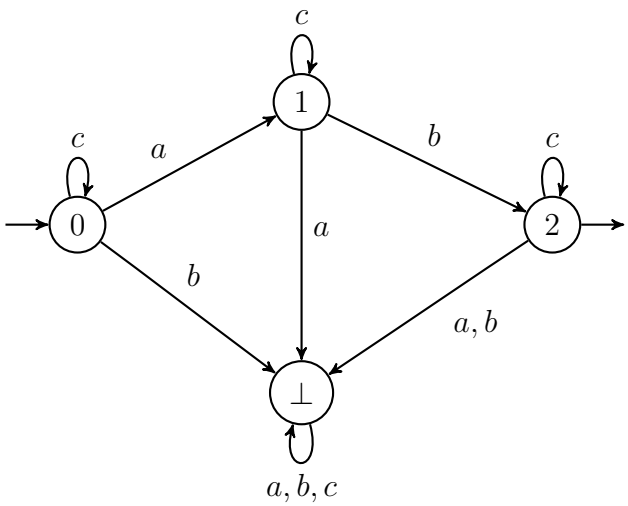
(notamment, $\delta'(q_\perp, x) = q_\perp$ quel que soit x).

Il est alors facile de voir que A' a le même comportement que A au sens où $\delta'^*(q, w) = \delta^*(q, w)$ lorsque le terme de droite est défini et $\delta'^*(q, w) = q_\perp$ sinon (le DFA A' « tombe dans le puits » lorsque le DFAi A cesse de fonctionner). En particulier, ils reconnaissent les mêmes langages. ☺

2.2.7. De façon générale, un état **puits** dans un DFA est un état $q_\perp$ tel que $\delta(q_\perp, x) = q_\perp$ pour toute lettre x.

On dit que le DFA A' est obtenu en **complétant** le DFAi A lorsqu'il est obtenu par la procédure décrite dans la démonstration de cette proposition, c'est-à-dire par l'addition d'un état puits vers lequel on fait pointer toutes les transitions manquantes (sauf si A est déjà complet, auquel cas on convient qu'il est son propre complété, i.e., on n'ajoute un puits que quand c'est réellement nécessaire).

2.2.8. À titre d'exemple, le DFA suivant représente la complétion du DFAi représenté en 2.2.5 :



2.2.9. On définit un état accessible d'un DFAi comme pour un DFA (cf. 2.1.8).

On dira en outre d'un état q d'un DFAi qu'il est **co-accessible** lorsqu'il existe un mot $w \in \Sigma^*$ tel que $\delta^*(q, w)$ soit défini et soit final, autrement dit, graphiquement, lorsqu'il existe un chemin orienté reliant l'état q considéré à un état final (remarquer que les états finaux eux-mêmes sont co-accessibles : prendre $w = \varepsilon$ dans ce qu'on vient de dire). Un état non co-accessible est donc un état à partir duquel il est impossible de faire accepter le mot. Cette définition pourrait également être faite pour les DFA, mais pour les DFAi elle présente l'intérêt qu'on peut supprimer les états non co-accessibles (ainsi, bien sûr, que toutes les transitions qui y conduisent) et obtenir de nouveau un DFAi.

Un DFAi dont tous les états sont à la fois accessibles et co-accessibles (on les dit aussi **utiles**) est parfois appelé **émondé**. On peut émonder un DFAi en ne conservant que ses états utiles [11] : ainsi, tout DFAi est équivalent à un DFAi émondé.

2.2.10. Il faut prendre garde au fait que certains auteurs définissent les « automates finis déterministes » (sans précision supplémentaire) comme étant complets par défaut, d'autres comme étant incomplets par défaut. Le plus prudent est de préciser systématiquement « complet » ou « incomplet » (en se rappelant qu'« incomplet » signifie « non nécessairement complet », cf. la note 9 sous 2.2.1) dès qu'il est important de ne pas confondre.

11. Si on aime pinailler, il y a un petit problème pour émonder un DFAi n'ayant aucun état final (donc aucun état co-accessible, donc aucun état utile) ; pour résoudre ce problème, on peut modifier légèrement la définition d'un DFAi et autoriser que l'état initial ne soit pas défini (auquel cas l'automate n'accepte évidemment aucun mot, i.e., reconnaît le langage $\varnothing$), si bien que l'automate émondé d'un DFAi sans état final est l'automate vide (sans aucun état).

2.3 Automates finis non-déterministes (=NFA)

2.3.1. Idée générale : Le principe directeur général du non-déterminisme en informatique est, grossièrement parlant, le suivant : on a un mécanisme de calcul (ici, un automate fini) qui doit accepter ou rejeter une certaine donnée (ici, un mot qui lui est présenté), mais plutôt que d'évoluer de façon déterministe d'une configuration à une autre (ici, d'un état à un autre), il se peut qu'il existe plusieurs possibilités différentes, autrement dit, une configuration peut évoluer de plusieurs manières différentes, ce qui donne lieu à plusieurs chemins de calcul différents. Selon le point de vue qu'on adopte, et éventuellement la stratégie algorithmique pour modéliser le non-déterminisme, on peut considérer que ces chemins sont empruntés tous simultanément (et qu'on est dans tous les états possibles à la fois [12], c'est d'ailleurs l'approche qui servira en 2.3.8), ou que tous les chemins possibles sont testés successivement, ou encore que l'appareil non-déterministe « devine » quel est le bon ; mais en tout cas la règle fondamentale du non-déterminisme est toujours la suivante : *la donnée est acceptée dès lors qu'il existe au moins un chemin de calcul possible qui conduit à l'accepter* (dans notre cas, cela signifiera au moins un chemin acceptant le mot).

2.3.2. Un **automate fini non-déterministe**, en abrégé **NFA**, sur un alphabet Σ est la donnée
- d'un ensemble fini Q d'états,
- d'un ensemble $I \subseteq Q$ d'états dits initiaux,
- d'un ensemble $F \subseteq Q$ d'états dits finaux,
- d'une *relation* de transition $\delta \subseteq Q \times \Sigma \times Q$ (c'est-à-dire une partie du produit cartésien $Q \times \Sigma \times Q$, i.e., un ensemble de triplets $(q, x, q') \in Q \times \Sigma \times Q$) ; lorsque $(q, x, q') \in \delta$, on dira qu'il existe une transition étiquetée par x de q vers q'.

Autrement dit, on autorise maintenant un ensemble quelconque d'états initiaux, et de même, au lieu qu'un état q et une lettre x déterminent un unique état $q' = \delta(q, x)$, on a maintenant affaire à un ensemble δ quelconque de triplets (q, x, q') pour les transitions possibles.

2.3.3. Un DFAi (ou *a fortiori* un DFA) est considéré comme un NFA particulier en définissant l'ensemble des états initiaux du NFA comme un singleton, $I_{\mathrm{NFA}} = \{q_{0,\mathrm{DFAi}}\}$, et en définissant la relation de transition du NFA comme le graphe de la fonction de transition du DFAi, c'est-à-dire $(q, x, q') \in \delta_{\mathrm{NFA}}$ lorsque $\delta_{\mathrm{DFAi}}(q, x)$ est défini et vaut q'.

2.3.4. Graphiquement, on représente un NFA comme un DFA : comme un graphe orienté dont les nœuds sont les éléments de Q, et où on place une arête étiquetée x de q vers q' pour chaque triplet $(q, x, q') \in \delta$; comme précédemment, on marque les états initiaux par une flèche entrante (i.e., pointant de nulle part) et les états finaux par une flèche sortante (i.e., pointant vers nulle part).

2.3.5. Il faut comprendre le fonctionnement d'un NFA de la manière suivante : un mot w est accepté par l'automate lorsqu'*il existe* un chemin orienté conduisant d'*un* état initial q_0 à un état final q_n et tel que le mot $w = x_1 \cdots x_n$ soit obtenu en lisant dans l'ordre les étiquettes x_i des différentes arêtes $q_{i-1} \to q_i$ de ce chemin ; autrement dit, w est accepté lorsqu'*il existe* $q_0, \ldots, q_n \in Q$ tels que $q_0 \in I$ et $q_n \in F$ et $(q_{i-1}, x_i, q_i) \in \delta$ pour chaque $1 \leq i \leq n$.

12. Certains sont parfois tentés de comparer avec une superposition quantique : il existe effectivement une analogie, mais le quantique implique une description bien précise des états (avec des amplitudes associées) qui n'est pas pertinente ici, et l'analogie mal appliquée conduirait à des erreurs en complexité algorithmique ; il vaut donc mieux ne pas y penser.

La différence cruciale avec les DFAi est donc que, maintenant, il pourrait exister plusieurs chemins possibles partant d'un état initial dont les transitions sont étiquetées par les lettres du même mot. Insistons bien sur le fait que le mot est accepté dès lors que *l'un* au moins de ces chemins relie un état initial à un état final.

2.3.6. De même que dans le cadre des DFA et DFAi on a étendu la fonction de transition δ à une fonction δ^* qui décrit le comportement de l'automate quand on consomme plusieurs caractères (cf. 2.1.5), on va étendre la relation de transition δ d'un NFA à la situation où on consomme plusieurs caractères. On a $(q, w, q') \in \delta^*$ lorsqu'il existe un chemin orienté conduisant de q à q' et tel que le mot w soit obtenu en lisant dans l'ordre les étiquettes des différentes arêtes de ce chemin.

Formellement : si $A = (Q, I, F, \delta)$ est un NFA sur l'alphabet Σ, on définit une relation $\delta^* \subseteq Q \times \Sigma^* \times Q$ par $(q, w, q') \in \delta^*$ lorsque $w = x_1 \cdots x_n$ et qu'il existe $(q_0, \ldots, q_n)$ tels que $q_0 = q$ et $q_n = q'$ et $(q_{i-1}, x_i, q_i) \in \delta$ pour chaque $1 \leq i \leq n$; ou, ce qui revient au même (par récurrence sur la longueur de w) :

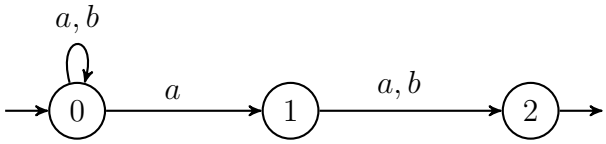
— $(q, \varepsilon, q') \in \delta^*$ si et seulement si $q' = q$,

— $(q, wx, q') \in \delta^*$ si et seulement si il existe $q^\sharp$ tel que $(q, w, q^\sharp) \in \delta^*$ et $(q^\sharp, x, q') \in \delta$.

Enfin, l'automate A accepte un mot w lorsqu'il existe $q_0 \in I$ et $q_\infty \in F$ tels que $(q_0, w, q_\infty) \in \delta^*$.

Le langage accepté $L(A)$ et l'équivalence de deux automates sont définis de façon analogue aux DFA (cf. 2.1.6).

2.3.7. Pour illustrer le fonctionnement des NFA, considérons l'automate à trois états sur $\Sigma = \{a, b\}$ représenté par la figure suivante : on a $Q = \{0, 1, 2\}$ avec $I = \{0\}$ et $F = \{2\}$ et $\delta = \{(0, a, 0), (0, b, 0), (0, a, 1), (1, a, 2), (1, b, 2)\}$.



Cet automate n'est pas déterministe car il existe deux transitions étiquetées a partant de l'état 0. En considérant les différents chemins possibles entre 0 et 2 dans ce graphe, on comprend que le langage qu'il reconnaît est le langage des mots sur $\{a, b\}$ dont l'avant-dernière lettre est un a (c'est aussi le langage dénoté par l'expression rationnelle $(a|b)*a(a|b)$). Une façon de présenter le non-déterminisme est que l'automate « devine », quand il est dans l'état 0 et qu'on lui fait consommer un a, si ce a sera l'avant-dernière lettre, et, dans ce cas, passe dans l'état 1 pour pouvoir accepter le mot.

Comme les DFAi avant eux, les NFA sont en fait équivalents aux DFA ; mais cette fois-ci, le coût algorithmique de la transformation peut être bien plus important :

Proposition 2.3.8. Soit $A = (Q, I, F, \delta)$ un NFA sur un alphabet Σ. Alors il existe un DFA $A' = (Q', \mathbf{q}'_0, F', \delta')$ (sur le même alphabet Σ) qui soit équivalent à A au sens où il reconnaît le même langage $L(A') = L(A)$. De plus, A' se déduit algorithmiquement de A avec une augmentation au plus exponentielle du nombre d'états : $\#Q' \leq 2^{\#Q}$.

Démonstration. On définit $Q' = \mathscr{P}(Q) = \{\mathbf{q} \subseteq Q\}$ l'*ensemble des parties* de Q : c'est ce qui servira d'ensemble d'états du DFA A' qu'on construit (i.e., un état de A' est un ensemble d'états

de A — intuitivement, c'est l'ensemble des états dans lesquels on se trouve simultanément). On pose $\mathbf{q}'_0 = I$ (l'état initial de A' sera l'ensemble de tous les états initiaux de A, vu comme un élément de Q') ; et on pose $F' = \{\mathbf{q} \subseteq Q : \mathbf{q} \cap F \neq \varnothing\}$: les états finaux de A' sont les états $\mathbf{q}$ qui, vus comme des ensembles d'états de A, contiennent *au moins un* état final. Enfin, pour $\mathbf{q} \subseteq Q$ et $x \in \Sigma$, on définit $\delta'(\mathbf{q}, x) = \{q_1 \in Q : \exists q_0 \in \mathbf{q}\,((q_0, x, q_1) \in \delta)\}$ comme l'ensemble de tous les états q_1 de A auxquels on peut accéder depuis un état q_0 dans $\mathbf{q}$ par une transition (q_0, x, q_1) (étiquetée par x) de A.

Il est alors facile de voir (par récurrence sur $|w|$) que $\delta'^*(\mathbf{q}, w)$ est l'ensemble de tous les les états $q_1 \in Q$ tels que $(q_0, w, q_1) \in \delta^*$, i.e., auxquels on peut accéder depuis un état q_0 dans $\mathbf{q}$ par une suite de transitions de A étiquetées par les lettres de w. En particulier, $\delta'^*(I, w)$ est l'ensemble de tous les états de A auxquels on peut accéder depuis un état initial de A par une suite de transitions de A étiquetées par les lettres de w : le mot w appartient à $L(A)$ si et seulement si cet ensemble contient un élément de F, ce qui par définition de F' signifie exactement $\delta'^*(I, w) \in F'$. On a bien prouvé $L(A') = L(A)$.

Enfin, $\#Q' = \#\mathscr{P}(Q) = 2^{\#Q}$ (car une partie de Q peut se voir à travers sa fonction indicatrice, qui est une fonction $Q \to \{0, 1\}$). ☺

2.3.9. On dit que le DFA A' est obtenu en **déterminisant** le NFA A lorsqu'il est obtenu par la procédure décrite dans la démonstration de cette proposition en ne gardant que les états accessibles.

Algorithmiquement, la déterminisation de A s'obtient par la procéduire suivante :

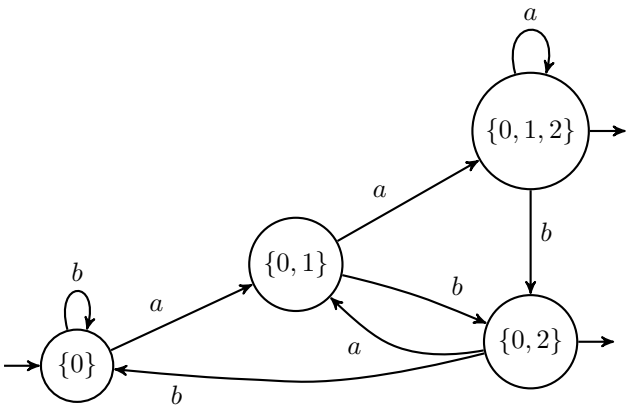
- créer une file (ou une pile) d'ensembles d'états de A ; initialiser cette file avec le seul élément I (vu comme un sous-ensemble de Q) ; et créer l'automate A' avec initialement l'unique état I, marqué comme état initial, et aussi comme final s'il contient un état final de A ;
- tant que la file/pile n'est pas vide : en extraire un élément $\mathbf{q}$, et, pour chaque lettre $x \in \Sigma$,
  - calculer l'ensemble $\mathbf{q}' = \{q_1 \in Q : \exists q_0 \in \mathbf{q}\,((q_0, x, q_1) \in \delta)\}$ (en listant tous les triplets $(q_0, x, q_1) \in \delta$ dont le premier élément est dans $\mathbf{q}$ et le second élément est x),
  - si $\mathbf{q}'$ n'existe pas déjà comme état de A', l'y ajouter, et dans ce cas, l'ajouter à la file/pile, et de plus, si $\mathbf{q}'$ contient un état final de A, marquer cet état comme final pour A',
  - et ajouter à A' la transition $\delta'(\mathbf{q}, x) = \mathbf{q}'$.

La file/pile sert à stocker les états de A' qui ont été créés mais pour lesquels les transitions sortantes n'ont pas encore été calculées. L'algorithme se termine quand la file/pile se vide, ce qui se produit toujours en au plus $2^{\#Q}$ étapes car chaque $\mathbf{q} \subseteq Q$ ne peut apparaître qu'une seule fois dans la file/pile.

Il se peut que l'état $\varnothing$ soit créé : cet état servira effectivement de puits, au sens où on aura $\delta'(\varnothing, x) = \varnothing$ quel que soit x (et l'état n'est pas acceptant).

Il arrive souvent que l'automate déterminisé soit plus petit que les $2^{\#Q}$ états qu'il a dans le pire cas.

2.3.10. À titre d'exemple, déterminisons le NFA A présenté en 2.3.7 : on commence par construire un état $\{0\}$ pour A' car le NFA a $\{0\}$ pour ensemble d'états initiaux ; on a $\delta'(\{0\}, a) = \{0, 1\}$ car 0 et 1 sont les deux états auxquels on peut arriver dans A par une transition partant de 0 et étiquetée par a, tandis que $\delta'(\{0\}, b) = \{0\}$; ensuite, $\delta'(\{0, 1\}, a) = \{0, 1, 2\}$ car $0, 1, 2$ sont les trois états auxquels on peut arriver dans A par une transition partant de 0 ou 1 et étiquetée par a ; et ainsi de suite. En procédant ainsi, on construit l'automate à 4 états qui suit :



On remarquera qu'on a construit moins que les $2^3 = 8$ états qu'on pouvait craindre.

Il est par ailleurs instructif de regarder comment fonctionne l'automate A' ci-dessus pour déterminer si l'avant-dernière lettre d'un mot est un a : essentiellement, il retient deux bits d'information, à savoir « est-ce que la dernière lettre consommée était un a ? » et « est-ce que l'avant-dernière lettre consommée était un a ? » ; chacune des quatre combinaisons de ces deux bits correspond à un état de A' : non/non est l'état $\{0\}$, oui/non est l'état $\{0, 1\}$, non/oui est l'état $\{0, 2\}$, et oui/oui est l'état $\{0, 1, 2\}$.

2.3.11. On vient donc de voir que les NFA sont équivalents aux DFA en ce sens qu'ils permettent de définir les mêmes langages. Il est néanmoins intéressant de les définir et de les étudier pour plusieurs raisons : d'une part, l'équivalence qu'on vient de voir, c'est-à-dire la déterminisation d'un NFA en DFA, a un coût algorithmique important (exponentiel en nombre d'états dans le pire cas), d'autre part, certaines opérations sur les automates se définissent plus naturellement sur les NFA que sur les DFA, et certaines constructions conduisent naturellement à un NFA. On verra en particulier en 3.2 que pour transformer une expression rationnelle en automate, on passera naturellement par un NFA, même si on peut souhaiter le déterminiser ensuite.

2.4 Automates finis non-déterministes à transitions spontanées (=εNFA)

2.4.1. Un **automate fini non-déterministe à transitions spontanées** ou **...à ε-transitions**, en abrégé **εNFA**, sur un alphabet Σ est la donnée

— d'un ensemble fini Q d'états,
— d'un ensemble $I \subseteq Q$ d'états dits initiaux,
— d'un ensemble $F \subseteq Q$ d'états dits finaux,
— d'une relation de transition $\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$.

Autrement dit, on autorise maintenant des transitions étiquetées par le mot vide ε plutôt que par une lettre $x \in \Sigma$: ces transitions sont dites **spontanées**, ou **ε-transitions**.

Soulignons qu'on ne définit les ε-transitions *que* pour les automates non-déterministes : ou, pour dire les choses autrement, *un automate qui possède des ε-transitions est par nature même non-déterministe*.

La représentation graphique des εNFA est évidente (on utilisera le symbole « ε » pour étiqueter les transitions spontanées). Un NFA est considéré comme un εNFA particulier pour lequel il n'y a pas de ε-transition.

2.4.2. Intuitivement, les transitions spontanées signifient que l'automate a la possibilité de passer spontanément, c'est-à-dire *sans consommer de lettre*, d'un état q à un état q', lorsque ces

états sont reliés par une ε-transition. (On comprend, du coup, pourquoi un automate à transition spontanée est forcément non-déterministe : ces transitions spontanées ne sont qu'une *possibilité*, ce qui sous-entend le non-déterminisme.)

De façon plus précise, un εNFA accepte un mot w lorsqu'*il existe* un chemin orienté conduisant d'un état initial q_0 à un état final q_n et tel que w coïncide avec le mot obtenu en lisant dans l'ordre les étiquettes t_i des différentes arêtes $q_{i-1} \to q_i$ de ce chemin, quitte à ignorer les ε.

Autrement dit, w est accepté lorsqu'*il existe* $q_0, \dots, q_n \in Q$ et $t_1, \dots, t_n \in (\Sigma \cup \{\varepsilon\})$ tels que $q_0 \in I$ et $q_n \in F$ et $(q_{i-1}, t_i, q_i) \in \delta$ pour chaque $1 \leq i \leq n$ et $w = t_1 \cdots t_n$ (*attention* : dans cette écriture, $t_1, \dots, t_n$ ne sont pas forcément les lettres de w, certains des t_i peuvent être le mot vide ε, les lettres de w sont obtenues en ignorant ces symboles).

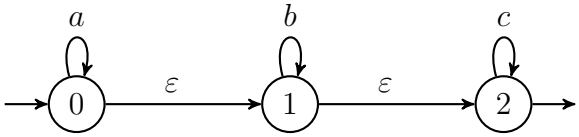
2.4.3. On veut de nouveau définir une relation δ^* telle que $(q, w, q') \in \delta^*$ lorsqu'il existe un chemin orienté conduisant de q à q' et tel que le mot w soit obtenu en lisant dans l'ordre les étiquettes des différentes arêtes de ce chemin.

Voici une formalisation possible : si $A = (Q, I, F, \delta)$ est un εNFA sur l'alphabet Σ, on définit une relation $\delta^* \subseteq Q \times \Sigma^* \times Q$ par $(q, w, q') \in \delta^*$ lorsqu'il existe $q_0, \dots, q_n \in Q$ et $t_1, \dots, t_n \in (\Sigma \cup \{\varepsilon\})$ tels que $q_0 = q$ et $q_n = q'$ et $(q_{i-1}, t_i, q_i) \in \delta$ pour chaque $1 \leq i \leq n$, et enfin $w = t_1 \cdots t_n$.

Enfin, l'automate A accepte un mot w lorsqu'il existe $q_0 \in I$ et $q_\infty \in F$ tels que $(q_0, w, q_\infty) \in \delta^*$.

Le langage accepté $L(A)$ et l'équivalence de deux automates sont définis de façon analogue aux DFA (cf. 2.1.6).

2.4.4. Voici un exemple de εNFA particulièrement simple sur $\Sigma = \{a, b, c\}$:



En considérant les différents chemins possibles entre 0 et 2 sur ce graphe, on comprend que le langage qu'il reconnaît est celui des mots sur $\{a, b, c\}$ formés d'un nombre quelconque de a suivi d'un nombre quelconque de b suivi d'un nombre quelconque de c, ou, si on préfère, le langage dénoté par l'expression rationnelle $a*b*c*$.

2.4.5. Si q est un état d'un εNFA, on appelle **ε-fermeture** de q l'ensemble des états q' (y compris q lui-même) accessibles depuis q par une succession quelconque de ε-transitions, c'est-à-dire, si on veut, $\{q' \in Q : (q, \varepsilon, q') \in \delta^*\}$. On notera temporairement $C(q)$ cet ensemble. (Par exemple, dans l'exemple 2.4.4 ci-dessus, on a $C(0) = \{0, 1, 2\}$ et $C(1) = \{1, 2\}$ et $C(2) = \{2\}$. Dans tout NFA sans ε-transitions, on a $C(q) = \{q\}$ pour tout état q.)

Il est clair qu'on peut calculer algorithmiquement $C(q)$ (par exemple par un algorithme de Dijkstra / parcours en largeur, sur le graphe orienté dont l'ensemble des sommets est Q et l'ensemble des arêtes est l'ensemble des ε-transitions de A : la ε-fermeture $C(q)$ est simplement l'ensemble des sommets accessibles depuis q dans ce graphe).

Proposition 2.4.6. Soit $A = (Q, I, F, \delta)$ un εNFA sur un alphabet Σ. Alors il existe un NFA $A^\S = (Q, I^\S, F^\S, \delta^\S)$ (sur le même alphabet Σ) ayant le même ensemble d'états Q que A et qui soit équivalent à A au sens où il reconnaît le même langage $L(A^\S) = L(A)$. De plus, $A^\S$ se déduit algorithmiquement de A.

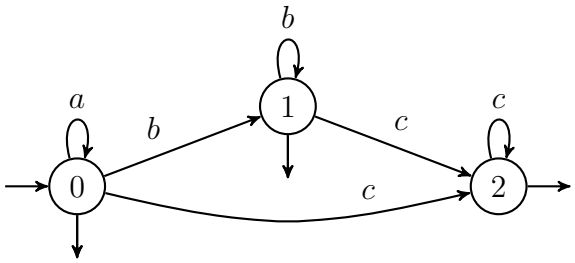
Démonstration. On pose $I^\S = I$ (mêmes états initiaux). L'idée est maintenant de faire une transition $(q, x, q') \in \delta^\S$ à chaque fois qu'on peut atteindre q' à partir de q dans A par une suite quelconque de ε-transitions suivie d'une unique transition étiquetée par x, autrement dit, $(q, \varepsilon, q^\sharp) \in \delta^*$ (c'est-à-dire $q^\sharp \in C(q)$) et $(q^\sharp, x, q') \in \delta$.

On définit donc $\delta^\S \subseteq Q \times \Sigma \times Q$ par $(q, x, q') \in \delta^\S$ lorsqu'il existe $q^\sharp \in C(q)$ tel que $(q^\sharp, x, q') \in \delta$: autrement dit, pour créer les transitions $q \to q'$ dans $A^\S$, on parcourt tous les $q^\sharp \in C(q)$, et on crée une transition $q \to q'$ étiquetée par $x \in \Sigma$ dans $A^\S$ lorsqu'il existe une transition $q^\sharp \to q'$ étiquetée par ce x dans A. De même, on définit $F^\S \subseteq Q$ comme l'ensemble des $q \in Q$ tels que $C(q) \cap F \neq \varnothing$, c'est-à-dire, depuis lesquels peut atteindre un état final par une succession de ε-transitions.

Si on a un chemin $q_0 \to q_1 \to \cdots \to q_n$ dans A menant d'un état initial $q_0 \in I$ à un état final $q_n \in F$ et étiquetées par $t_1, \ldots, t_n \in (\Sigma \cup \{\varepsilon\})$ (c'est-à-dire $(q_{i-1}, t_i, q_i) \in \delta$), appelons $j_1 < \ldots < j_m$ les indices tels que $t_j \in \Sigma$, autrement dit, tels que la transition $q_{j-1} \to q_j$ ne soit pas spontanée, et posons $j_0 = 0$. Alors on passe de $q_{j_{(i-1)}}$ à q_{j_i} par une succession de ε-transitions (de $q_{j_{(i-1)}}$ à $q_{(j_i)-1}$) suivie par une unique transition non spontanée : on a $q_{(j_i)-1} \in C(q_{j_{(i-1)}})$ et $(q_{(j_i)-1}, t_{j_i}, q_{j_i}) \in \delta$, autrement dit $(q_{j_{(i-1)}}, t_{j_i}, q_{j_i}) \in \delta^\S$; et comme le mot $w = t_1 \cdots t_n$ s'écrit aussi $t_{j_1} \cdots t_{j_m}$, on a un chemin reliant $q_{j_0} = q_0 \in I$ à $q_{j_m} \in F^\S$ (puisque $q_n \in C(q_{j_m}) \cap F$). Le mot w supposé accepté par A est donc accepté par $A^\S$. La réciproque est analogue. ☺

2.4.7. On dit que le NFA $A^\S$ est obtenu en **éliminant les ε-transitions** dans le εNFA A lorsqu'il est obtenu par la procédure décrite dans la démonstration de cette proposition, et en supprimant tous les états non-initiaux de $A^\S$ auxquels n'aboutissent dans A que des ε-transitions (ces états sont devenus inaccessibles dans $A^\S$). Algorithmiquement, il s'agit donc, pour chaque état $q \in Q$ et chaque $q^\sharp$ dans la ε-fermeture $C(q)$ de q, de créer une transition $q \to q'$ étiquetée par x dans $A^\S$ pour chaque transition $q^\sharp \to q'$ étiquetée par x dans A.

2.4.8. À titre d'exemple, éliminons les ε-transitions du εNFA A présenté en 2.4.4 : comme $C(0) = \{0, 1, 2\}$, on fait partir de 0 toutes les transitions partant d'un des états $0, 1, 2$ et étiquetées par une lettre, et de même, comme $C(1) = \{1, 2\}$, on fait partir de 1 toutes les transitions partant d'un des états $1, 2$ et étiquetées par une lettre. On obtient finalement l'automate suivant :



(Sur cet exemple précis, on obtient un automate déterministe incomplet, mais ce n'est pas un phénomène général : en général il faut s'attendre à obtenir un NFA.)

2.4.9. Remarque : La manière dont on a éliminé les ε-transitions ci-dessus consiste à remplacer *une succession de ε-transitions suivie d'une unique transition étiquetée x* par une transition étiquetée x (et de même, on modifie les états finaux, mais pas les états initiaux). Il existait un moyen « dual » d'éliminer les ε-transitions, à savoir remplacer *une unique transition étiquetée x suivie d'une succession de ε-transitions* par une transition étiquetée x

(et de même, on modifie les états initiaux, mais pas les états finaux). Autrement dit, la construction $A \mapsto A^{\S}$ décrite en 2.4.6 définit $(q, x, q') \in \delta^{\S}$ lorsqu'il existe $q^\sharp \in C(q)$ tel que $(q^\sharp, x, q') \in \delta$, et $I^{\S} = I$ et $F^{\S}$ comme l'ensemble des états q tels que $C(q) \cap F \neq \varnothing$; la construction « duale » $A \mapsto A^{\P}$ consiste à poser $(q, x, q') \in \delta^{\P}$ lorsqu'il existe $q^{\flat\prime}$ tel que $(q, x, q^{\flat\prime}) \in \delta$ et $q' \in C(q^{\flat\prime})$, et $F^{\P} = F$ et $I^{\P}$ comme la réunion des $C(q)$ pour tout $q \in I$.

Ces deux manières d'éliminer les ε-transitions donnent des NFA équivalents. En fait, on peut définir l'une en termes de l'autre en utilisant la définition de l'automate transposé A^{R} présentée en 3.1.6 plus bas (et qui consiste simplement à inverser le sens de toutes les flèches de l'automate) : si on inverse les flèches, qu'on élimine les ε-transitions à la manière décrite en 2.4.6, et qu'on inverse de nouveau les flèches, on obtient l'élimination « duale », autrement dit, $A^{\P} = ((A^{\mathsf{R}})^{\S})^{\mathsf{R}}$.

L'une ou l'autre manière d'éliminer les ε-transitions était possible, mais il vaut mieux ne pas les mélanger. C'est pour cette raison qu'on a fait un choix en 2.4.6 ; la présente remarque a principalement pour objectif d'expliquer la raison d'une perte de symétrie (notamment entre états initiaux et finaux) dans ce choix.

2.4.10. L'intérêt des εNFA, même s'ils sont finalement équivalents aux NFA, est que certaines constructions sur les automates sont plus simples ou plus transparentes lorsqu'on s'autorise à fabriquer des εNFA (à titre d'exemple, l'automate présenté en 2.4.4 est beaucoup plus transparent à lire que le NFA équivalent donné en 2.4.8). On verra notamment en 3.3 qu'il est facile (quoique inefficace) de construire un εNFA reconnaissant le langage dénoté par une expression rationnelle quelconque ; et en 3.2 que, même si on fabrique directement un NFA, il peut être utile d'introduire temporairement des transitions spontanées dans la construction de ce NFA, quitte à les éliminer immédiatement (cela simplifie la description).

3 Langages reconnaissables et langages rationnels

3.1 Stabilité des langages reconnaissables par opérations booléennes et miroir

3.1.1. On rappelle qu'on a défini un langage reconnaissable comme un langage L pour lequel il existe un DFA A tel que $L = L(A)$. D'après 2.2.6, 2.3.8 et 2.4.6, on peut remplacer « DFA » dans cette définition par « DFAi », « NFA » ou « εNFA » sans changer la définition.

Nous allons maintenant montrer que les langages reconnaissables sont stables par différentes opérations. Dans cette section, nous traitons le cas des opérations booléennes (complémentaire, union, intersection) et l'opération « miroir » ; la section 3.2 traite des opérations rationnelles.

Proposition 3.1.2. Si L est un langage reconnaissable sur un alphabet Σ, alors le complémentaire $\Sigma^* \setminus L$ de L est reconnaissable ; de plus, un DFA reconnaissant l'un se déduit algorithmiquement d'un DFA reconnaissant l'autre.

Démonstration. Par hypothèse, il existe un DFA (complet !) $A = (Q, q_0, F, \delta)$ tel que $L = L(A)$. Considérons le DFA A' défini par l'ensemble d'états $Q' = Q$, l'état initial $q'_0 = q_0$, la fonction de transition $\delta' = \delta$ et pour seul changement l'ensemble d'états finaux $F' = Q \setminus F$ complémentaire de F.

Pour $w \in \Sigma^*$, on a $w \in L(A')$ si et seulement si $\delta'^*(q_0, w) \in F'$, c'est-à-dire $\delta^*(q_0, w) \in F'$ (puisque $\delta' = \delta$), c'est-à-dire $\delta^*(q_0, w) \notin F$ (par définition du complémentaire), c'est-à-dire $w \notin L(A)$. Ceci montre bien que $L(A')$ est le complémentaire de $L(A)$. ☺

3.1.3. Cette démonstration a utilisé la caractérisation des langages reconnaissables par les DFA : il était crucial de le faire, et les autres sortes d'automates définis plus haut n'auraient

pas permis d'arriver (simplement) à la même conclusion. Il est intéressant de réfléchir à pourquoi. (Essentiellement, dans un NFA, un mot est accepté dès qu'*il existe* un chemin qui l'accepte, or l'existence d'un chemin aboutissant à un état non-final n'est pas la même chose que l'inexistence d'un chemin aboutissant à un état final.)

Proposition 3.1.4. Si L_1, L_2 sont des langages reconnaissables (sur un même alphabet Σ), alors la réunion $L_1 \cup L_2$ et l'intersection $L_1 \cap L_2$ sont reconnaissables ; de plus, un DFA reconnaissant l'un comme l'autre se déduit algorithmiquement de DFA reconnaissant L_1 et L_2.

Démonstration. Traitons le cas de l'intersection. Par hypothèse, il existe des DFA (complets !) $A_1 = (Q_1, q_{0,1}, F_1, \delta_1)$ et $A_2 = (Q_2, q_{0,2}, F_2, \delta_2)$ tels que $L_1 = L(A_1)$ et $L_2 = L(A_2)$. Considérons le DFA A' défini par l'ensemble d'états $Q' = Q_1 \times Q_2$ (c'est-à-dire l'ensemble des couples formés d'un état de A_1 et d'un état de A_2), l'état initial $q'_0 = (q_{0,1}, q_{0,2})$, la fonction de transition $\delta' \colon ((p_1, p_2), x) \mapsto (\delta_1(p_1, x), \delta_2(p_2, x))$ et pour ensemble d'états finaux $F' = F_1 \times F_2$. Remarquons que $(p_1, p_2) \in Q'$ appartient à $F' = F_1 \times F_2$ si et seulement si $p_1 \in F_1$ *et* $p_2 \in F_2$ (i.e., $F' \subseteq Q'$ est l'ensemble des couples dont les deux composantes sont finales). Par ailleurs, si $w \in \Sigma^*$, on a $\delta'^*(q'_0, w) = (\delta_1^*(q_{0,1}, w), \delta_2^*(q_{0,2}, w))$, et par ce qui vient d'être dit, ceci appartient à F' si et seulement $\delta_1^*(q_{0,1}, w) \in F_1$ et $\delta_2^*(q_{0,2}, w) \in F_2$. On voit donc qu'un mot w appartient à $L(A')$ si et seulement il appartient à la fois à L_1 et à L_2, ce qu'il fallait démontrer.

Pour la réunion, on peut invoquer le fait que la réunion est le complémentaire de l'intersection des complémentaires, et utiliser 3.1.2 ; si on déroule cette démonstration, on voit qu'on construit un DFA A'' exactement égal à A' construit ci-dessus, à la seule différence près que son ensemble d'états finaux est $F'' = (F_1 \times Q_2) \cup (Q_1 \times F_2)$, qui est le sous-ensemble de $Q'' = Q_1 \times Q_2$ formé des couples dont *l'une au moins* des deux composantes est finale. ☺

3.1.5. La construction A' ci-dessus est parfois appelée automate *produit* des DFA A_1 et A_2. Intuitivement, il faut comprendre que faire fonctionner l'automate A' revient à faire fonctionner les automates A_1 et A_2 simultanément (en parallèle), en leur donnant à consommer les mêmes symboles.

La construction de l'automate produit pour fabriquer le langage union ou intersection utilise la caractérisation des langages reconnaissables par les DFA ; une construction du même type pourrait être définie pour les NFA, mais uniquement pour le langage intersection. (Pour la réunion effectuée sur les NFA, on renvoie à la construction présentée en 3.2.3.)

Proposition 3.1.6. Si L est un langage reconnaissable sur un alphabet Σ, alors le langage miroir (cf. 1.3.10) L^{R} de L est reconnaissable ; de plus, un NFA ou εNFA reconnaissant l'un se déduit algorithmiquement d'un NFA ou εNFA reconnaissant l'autre : il s'agit simplement d'inverser le sens de toutes les flèches (y compris celles qui marquent les états initiaux et finaux).

L'automate ainsi construit en inversant toutes les flèches d'un automate A (la définition précise est donnée dans la démonstration qui suit) et qui reconnaît le langage miroir de celui reconnu par A peut s'appeler automate **transposé** A^{R} de A.

Démonstration. Par hypothèse, il existe un NFA ou un εNFA $A = (Q, I, F, \delta)$ tel que $L = L(A)$. Considérons l'automate A^{R} de même type défini par l'ensemble d'états $Q^{\mathsf{R}} = Q$ et inversant toutes les flèches de A, c'est-à-dire $I^{\mathsf{R}} = F$ et $F^{\mathsf{R}} = I$ et $(q, t, q') \in \delta^{\mathsf{R}}$ si et seulement si $(q', t, q) \in \delta$. Un chemin existe dans A^{R} si et seulement si le même chemin inversé existe dans A, ce qui montre qu'un mot appartient à $L(A^{\mathsf{R}})$ si et seulement si son miroir appartient à $L(A)$. On a donc bien $L(A^{\mathsf{R}}) = L^{\mathsf{R}}$, langage miroir de L. ☺

3.1.7. Alors que les constructions du complémentaire et de l'intersection s'effectuaient naturellement sur les DFA, celle du langage miroir s'effectue naturellement sur les NFA. (On peut, bien sûr, considérer un DFA comme un NFA particulier, et effectuer dessus l'opération d'inversion des flèches qu'on vient de décrire, mais en général on n'obtiendra pas un DFA, seulement un NFA ; les NFA dont l'automate transposé est déterministe — c'est-à-dire tels que, pour chaque état q et chaque lettre x, il existe une unique arête aboutissant à q et étiquetée par x — sont parfois dits « co-déterministes ».)

3.2 Stabilité des langages reconnaissables par opérations rationnelles, automates standards, construction de Glushkov

3.2.1. Nous allons maintenant montrer que la classe des langages reconnaissables est stable par les opérations rationnelles (union, concaténation et étoile de Kleene, cf. 1.4.3 ; on l'a déjà vu sur les DFA en 3.1.4 pour la réunion, mais on va en donner une nouvelle démonstration, cette fois basée sur les NFA, qui a un contenu algorithmique utile dans des circonstances différentes). Cela permettra de conclure en 3.2.7 que les langages rationnels sont reconnaissables (la réciproque faisant l'objet de la section 3.4).

Pour établir ces stabilités, on va travailler sur les NFA et utiliser la construction parfois appelée « de Glushkov » ou « automate standard » ; ceci fournira un « automate de Glushkov » pour chaque expression rationnelle r. Cette construction travaille, en fait, sur des NFA vérifiant une propriété supplémentaire facile à assurer, et on va commencer par un lemme dans ce sens :

Lemme 3.2.2. Soit A un NFA. Alors il existe un NFA A' (sur le même alphabet Σ) qui soit équivalent à A et qui possède la propriété supplémentaire d'avoir un *unique* état initial q_0 et qu'aucune transition n'aboutit à q_0. De plus, A' se déduit algorithmiquement de A et a au plus un état de plus que A.

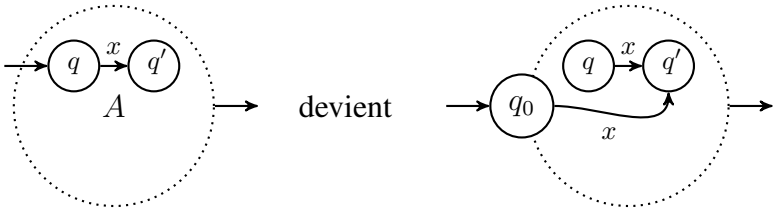
(On pourra appeler **standard** un NFA vérifiant cette propriété d'avoir un unique état initial qui n'est la cible d'aucune transition. L'affirmation est donc que tout NFA est équivalent à un NFA *standard* qui s'en déduit algorithmiquement par l'ajout d'au plus un état.)

Démonstration. On fabrique A' en reprenant le même ensemble d'états Q que dans A auquel on ajoute un unique nouvel état q_0 qui sera le seul état initial de A' ; pour chaque transition partant d'un état initial de A, on ajoute dans A' une transition identiquement étiquetée, et de même cible, partant de q_0.

Formellement : soit $A = (Q, I, F, \delta)$. On définit alors $A' = (Q', \{q_0\}, F', \delta')$ de la manière suivante : $Q' = Q \uplus \{q_0\}$ (où $\uplus$ désigne une réunion disjointe [13]), δ' est la réunion de l'ensemble des transitions (q, x, q') qui étaient déjà dans δ et de l'ensemble des (q_0, x, q') telles qu'il existe une transition $(q, x, q') \in \delta$ avec $q \in I$, et enfin $F' = F$ ou $F' = F \cup \{q_0\}$ selon que $I \cap F = \varnothing$ ou $I \cap F \neq \varnothing$ respectivement (i.e., q_0 est final lorsqu'il existait déjà un état à la fois initial et final).

La figure suivante illustre la transformation en question :

13. C'est-à-dire qu'on exige que q_0 n'appartienne pas à Q (c'est un nouvel état).



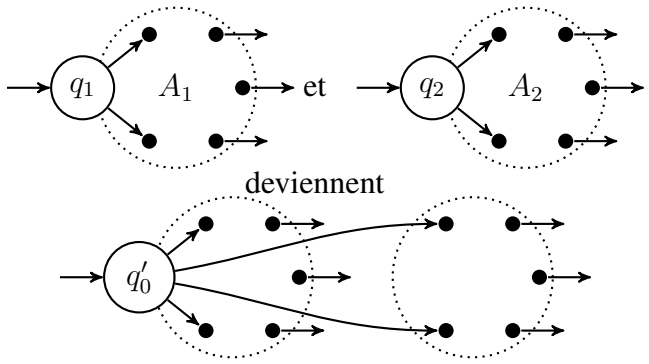
(*Remarque :* De façon équivalente, on peut fabriquer A' en ajoutant d'abord un unique état initial q_0 et des ε-transitions de q_0 vers chacun des états qui étaient initiaux dans A, puis en éliminant les ε-transitions qu'on vient d'ajouter (cf. 2.4.6). Cela donne le même résultat que ce qui vient d'être dit.) ☺

On a vu en 3.1.4 une preuve, à base de DFA, que $L_1 \cup L_2$ est reconnaissable lorsque L_1 et L_2 le sont. Donnons maintenant une autre preuve de ce fait, à base de NFA :

Proposition 3.2.3. Si L_1, L_2 sont des langages reconnaissables (sur un même alphabet Σ), alors la réunion $L_1 \cup L_2$ est reconnaissable ; de plus, un NFA la reconnaissant se déduit algorithmiquement de NFA reconnaissant L_1 et L_2.

Démonstration. Par hypothèse, il existe des NFA reconnaissant L_1 et L_2 : d'après 3.2.2, on peut supposer qu'ils sont *standards* en ce sens qu'ils ont un unique état initial qui n'est la cible d'aucune transition. Disons que $A_1 = (Q_1, \{q_1\}, F_1, \delta_1)$ et $A_2 = (Q_2, \{q_2\}, F_2, \delta_2)$ sont des NFA standards tels que $L_1 = L(A_1)$ et $L_2 = L(A_2)$.

L'automate A' s'obtient réunissant A_1 et A_2 mais en « fusionnant » les états initiaux q_1 et q_2 de A_1 et A_2 en un unique état initial q_0', d'où partent les mêmes transitions (avec les mêmes étiquettes) que depuis l'un ou l'autre de q_1 ou q_2. Graphiquement :



De façon plus formelle, considérons un nouvel ensemble d'états $Q' = (Q_1 \uplus Q_2) \setminus \{q_1, q_2\} \uplus \{q_0'\}$ où $\uplus$ désigne la réunion disjointe (autrement dit, on prend la réunion disjointe des états non-initiaux de A_1 et A_2 et on ajoute un nouvel état q_0'), et la fonction $\varphi_1 \colon Q_1 \to Q'$ qui envoie q_1 sur q_0' et tout autre état de Q_1 sur lui-même, et $\varphi_2 \colon Q_2 \to Q'$ de façon analogue. On définit alors l'automate A' dont l'ensemble d'états est Q', l'état initial est q_0', les états finaux $F' = \varphi_1(F_1) \cup \varphi_2(F_2)$, et la relation de transition δ' est formée des triplets $(\varphi_1(q), x, q')$ où $(q, x, q') \in \delta_1$ et des $(\varphi_2(q), x, q')$ où $(q, x, q') \in \delta_2$.

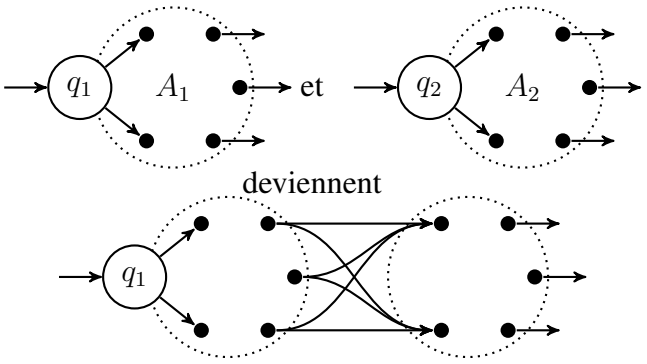
(*Remarque :* De façon équivalente, on peut fabriquer A' en ajoutant d'abord un unique état initial q_0' à la réunion disjointe de A_1 et A_2 et des ε-transitions de q_0' vers q_1 et q_2 (qui cessent d'être initiaux), puis en éliminant les ε-transitions qu'on vient d'ajouter (cf. 2.4.6) ainsi que les états q_1 et q_2 devenus inutiles. Cela donne le même résultat que ce qui vient d'être dit.)

Il est alors clair qu'un chemin de l'état initial à un état final dans cet automate A' consiste soit en un chemin d'un état initial à un état final dans A_1 soit en un tel chemin dans A_2. On a donc bien $L(A') = L_1 \cup L_2$. ☺

Proposition 3.2.4. Si L_1, L_2 sont des langages reconnaissables (sur un même alphabet Σ), alors la concaténation L_1L_2 est reconnaissable ; de plus, un NFA la reconnaissant se déduit algorithmiquement de NFA reconnaissant L_1 et L_2.

Démonstration. Par hypothèse, il existe des NFA reconnaissant L_1 et L_2 : d'après 3.2.2, on peut supposer qu'ils sont *standards* en ce sens qu'ils ont un unique état initial qui n'est la cible d'aucune transition. Disons que $A_1 = (Q_1, \{q_1\}, F_1, \delta_1)$ et $A_2 = (Q_2, \{q_2\}, F_2, \delta_2)$ sont des NFA standards tels que $L_1 = L(A_1)$ et $L_2 = L(A_2)$.

L'automate A' s'obtient en réunissant A_1 et A_2, en ne gardant que les états finaux de A_2, en supprimant q_2 et en remplaçant chaque transition sortant de q_2 par une transition identiquement étiquetée, et de même cible, partant de *chaque* état final de A_1 (ces derniers seront marqués finaux si q_2 était final dans A_2). Graphiquement :



De façon plus formelle, considérons un nouvel ensemble d'états $Q' = (Q_1 \uplus Q_2) \setminus \{q_2\}$ où $\uplus$ désigne la réunion disjointe. On définit alors l'automate A' dont l'ensemble d'états est Q', l'état initial est q_1, l'ensemble F' des états finaux est F_2 si q_2 n'était pas final dans A_2 et $F_1 \cup F_2$ si q_2 était final dans A_2, la relation de transition δ' est la réunion de δ_1, de l'ensemble des triplets $(q, x, q') \in \delta_2$ tels que $q \neq q_2$, et enfin de l'ensemble des triplets (q, x, q') tels que $(q_2, x, q') \in \delta_2$ et que $q \in F_1$.

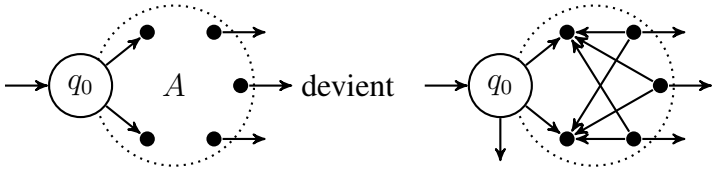
(*Remarque :* De façon équivalente, on peut fabriquer A' en ajoutant d'abord à la réunion disjointe de A_1 et A_2 une ε-transition de chaque état final de A_1 vers q_2 (qui cesse d'être initial), puis en éliminant les ε-transitions qu'on vient d'ajouter (cf. 2.4.6) ainsi que l'état q_2 devenu inutile. Cela donne le même résultat que ce qui vient d'être dit.)

Il est alors clair qu'un chemin de l'état initial q_1 à un état final dans cet automate A' consiste en un chemin de q_1 à un état final dans A_1 suivi d'un chemin de q_2 à un état final dans A_2 (moins q_2 lui-même). On a donc bien $L(A') = L_1L_2$. ☺

Proposition 3.2.5. Si L est un langage reconnaissable (sur un alphabet Σ), alors l'étoile de Kleene L^* est reconnaissable ; de plus, un NFA la reconnaissant se déduit algorithmiquement de NFA reconnaissant L.

Démonstration. Par hypothèse, il existe un NFA reconnaissant L : d'après 3.2.2, on peut supposer qu'ils est *standard* en ce sens qu'il a un unique état initial qui n'est la cible d'aucune transition. Disons que $A = (Q, \{q_0\}, F, \delta)$ est un NFA standard tel que $L = L(A)$.

L'automate A' s'obtient en ajoutant à A, pour chaque transition sortant de q_0, une transition identiquement étiquetée, et de même cible, partant de chaque état final de A, et en rendant q_0 final s'il ne l'était pas déjà :

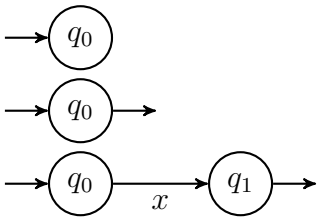


De façon plus formelle, on considère l'automate A' dont l'ensemble d'états est $Q' := Q$, l'état initial est q_0, les états finaux $F' := F \cup \{q_0\}$, et la relation de transition δ' est la réunion de δ et de l'ensemble des triplets (q, x, q') tels que $(q_0, x, q') \in \delta$ et que $q \in F$.

(*Remarque :* De façon équivalente, on peut fabriquer A' en ajoutant d'abord à A une ε-transition de chaque état final de A vers q_0, puis en éliminant les ε-transitions qu'on vient d'ajouter (cf. 2.4.6), et enfin en marquant q_0 comme final. Cela donne le même résultat que ce qui vient d'être dit.)

Il est alors clair qu'un chemin de l'état initial q_0 à un état final dans cet automate A' consiste en un nombre quelconque (éventuellement nul) de chemins de l'état initial à un état final dans A. On a donc bien $L(A') = L^*$. ☺

3.2.6. Il sera utile de fixer également des NFA (« standards » au sens de 3.2.2) reconnaissant les langages de base triviaux $\varnothing$, $\{\varepsilon\}$ et $\{x\}$ (pour chaque $x \in \Sigma$), c'est-à-dire ceux dénotés par les expressions rationnelles $\bot$, $\underline{\varepsilon}$ et x respectivement. On prendra les suivants :



pour le langage $\varnothing$ (i.e., pour l'expression rationnelle $\bot$),

pour le langage $\{\varepsilon\}$ (i.e., pour l'expression rationnelle $\underline{\varepsilon}$), et

pour le langage $\{x\}$ (i.e., pour l'expression rationnelle x).

Récapitulons le contenu essentiel de ce que nous avons montré :

Corollaire 3.2.7. Tout langage rationnel est reconnaissable ; de plus, un NFA le reconnaissant se déduit algorithmiquement d'une expression rationnelle le dénotant.

En particulier, il existe un algorithme qui, donnée une expression rationnelle r (sur un alphabet Σ) et un mot $w \in \Sigma^*$, décide si $w \in L(r)$, c'est-à-dire si w vérifie r. (Autrement dit, les langages rationnels sont *décidables* au sens de 5.1.5.)

Démonstration. L'affirmation du premier paragraphe résulte de façon évidente de la définition des langages rationnels (cf. §1.4), du fait que les langages $\varnothing$, $\{\varepsilon\}$ et $\{x\}$ (pour chaque $x \in \Sigma$) sont reconnaissables par automates finis (cf. 3.2.6), et grâce aux propositions 3.2.3, 3.2.4 et 3.2.5. On renvoie à 3.2.8 ci-dessous (ou à §3.3) pour une description algorithmique plus précise.

Pour décider si un mot w vérifie une expression rationnelle r, on commence par transformer cette expression rationnelle en NFA comme on vient de l'expliquer (c'est-à-dire construire un NFA $A(r)$ reconnaissant le langage $L(r)$ dénoté par r), puis on déterminise cet automate (cf. 2.3.8), après quoi il est facile de tester si le DFA résultant de la déterminisation accepte le mot w considéré (il suffit de suivre l'unique chemin dans l'automate partant de l'état initial et étiqueté par w, et de voir si l'état auquel il aboutit est final). ☺

3.2.8. Les constructions que nous avons décrites dans cette section associent naturellement un NFA standard à chaque expression rationnelle : il s'obtient en partant des automates de base décrits en 3.2.6 et en appliquant les constructions décrites dans les démonstrations de 3.2.3, 3.2.4 et 3.2.5.

Plus exactement, on associe à chaque expression rationnelle r (sur un alphabet Σ fixé) un automate $A(r)$ (ou $A_{\text{Glushkov}}(r)$) standard, appelé **automate de Glushkov**, qui reconnaît le langage $L(r)$ dénoté par r, de la manière suivante (par induction sur la complexité de l'expression rationnelle telle que définie en 1.4.4) :

— les automates de Glushkov de $\bot$, $\varnothing$ et x (pour $x \in \Sigma$) sont définis comme ceux décrits et illustrés en 3.2.6,
— connaissant les automates de Glushkov A_1 et A_2 de r_1 et r_2 respectivement, celui de $(r_1|r_2)$ est celui décrit et illustré dans la démonstration de 3.2.3,
— connaissant les automates de Glushkov A_1 et A_2 de r_1 et r_2 respectivement, celui de $r_1 r_2$ est celui décrit et illustré dans la démonstration de 3.2.4,
— connaissant l'automate de Glushkov A de r, celui de $(r)*$ est celui décrit et illustré dans la démonstration de 3.2.5.

Cet automate de Glushkov $A_{\text{Glushkov}}(r)$ possède les propriétés suivantes :

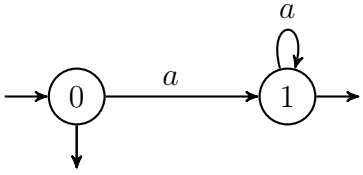
— c'est un NFA reconnaissant le langage $L(r)$ dénoté par l'expression rationnelle r dont on est parti,
— il est standard au sens de 3.2.2, c'est-à-dire qu'il possède un unique état initial auquel n'aboutit aucune transition,
— les transitions aboutissant à n'importe quel état donné sont toutes étiquetées par la même lettre,
— son nombre d'états est égal à 1 plus le nombre de lettres (à l'exclusion des métacaractères) contenues dans l'expression r.

Les deux dernières propriétés se vérifient inductivement, c'est-à-dire qu'on observe qu'elles sont satisfaites sur les automates de base décrits en 3.2.6 qu'elles sont préservées par les constructions de 3.2.3, 3.2.4 et 3.2.5. En fait, on peut être un peu plus précis : chaque état, autre que l'état initial, de l'automate de Glushkov associé à l'expression rationnelle r correspond à une lettre x (à l'exclusion des métacaractères) de r, l'état en question provient de l'état q_1 de l'automate décrit en 3.2.6 pour le langage $\{x\}$, et toutes les transitions menant à cet état sont étiquetées par x.

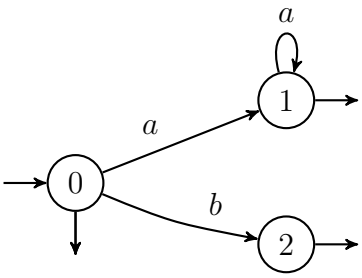
Ces observations sont utiles pour détecter des erreurs lors de la construction de l'automate.

3.2.9. À titre d'exemple pour illustrer la construction de Glushkov, construisons l'automate qu'elle associe à l'expression rationnelle $((a*|b)c)*$ sur l'alphabet $\Sigma = \{a, b, c\}$. On doit obtenir un automate ayant 4 états.

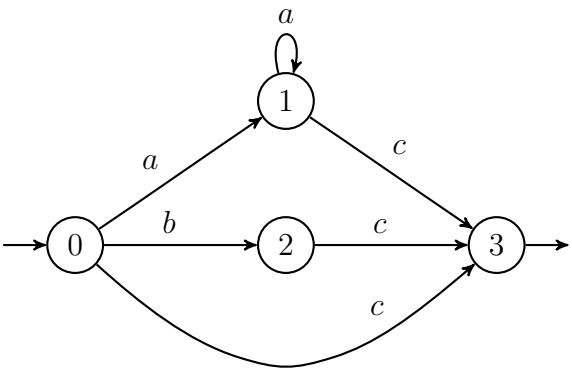
L'automate de Glushkov de $a*$ est le suivant, obtenu en appliquant la construction de 3.2.5 à l'automate trivial pour le langage $\{a\}$ (donné en 3.2.6) :



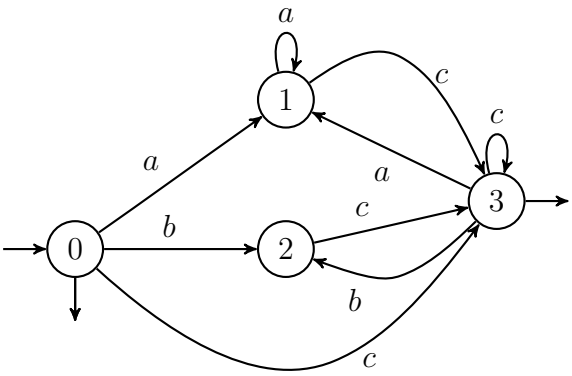
On en déduit au moyen de 3.2.3 l'automate suivant pour $a*|b$:



On en déduit au moyen de 3.2.4 l'automate suivant pour $(a*|b)c$:



Et enfin, de nouveau par 3.2.5 l'automate suivant pour $((a*|b)c)*$:



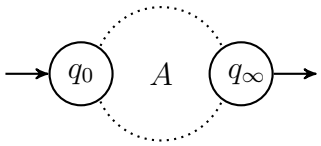
Toutes les transitions aboutissant à l'état 1 sont étiquetées a, toutes celles aboutissant à l'état 2 sont étiquetées b, et toutes celles aboutissant à 3 sont étiquetées c.

3.3 L'automate de Thompson (alternative à l'automate de Glushkov)

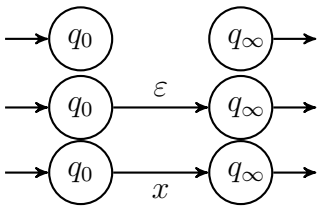
3.3.1. La construction de Glushkov (exposée en 3.2.8) d'un automate reconnaissant le langage dénoté par expression rationnelle r fabrique un NFA. Cette construction produit un automate raisonnablement compact (en nombre d'états), mais il peut être intéressant de disposer d'une autre construction, plus transparente mais moins efficace : la **construction de Thompson** fournit un autre moyen d'associer à une expression rationnelle r un automate $A_{\text{Thompson}}(r)$ reconnaissant le langage qu'elle dénote. Elle possède pour sa part les propriétés suivantes :

— c'est un εNFA reconnaissant le langage $L(r)$ dénoté par l'expression rationnelle r dont on est parti,
— il possède un unique état initial auquel n'aboutit aucune transition, et un unique état final duquel ne part aucune transition,
— son nombre d'états est égal au double du nombre de symboles autres que les parenthèses constituant l'expression r (en comptant aussi bien les lettres de Σ que les métacaractères $\bot$, $\underline{\varepsilon}$, $|$ et $*$; mais sans compter la concaténation implicite).

Dans les dessins qui suivent, on symbolisera de la manière suivante un automate de Thompson A quelconque :



3.3.2. Les automates de Thompson des expressions rationnelles $\bot$, $\underline{\varepsilon}$ et x (pour $x \in \Sigma$) seront les suivants :

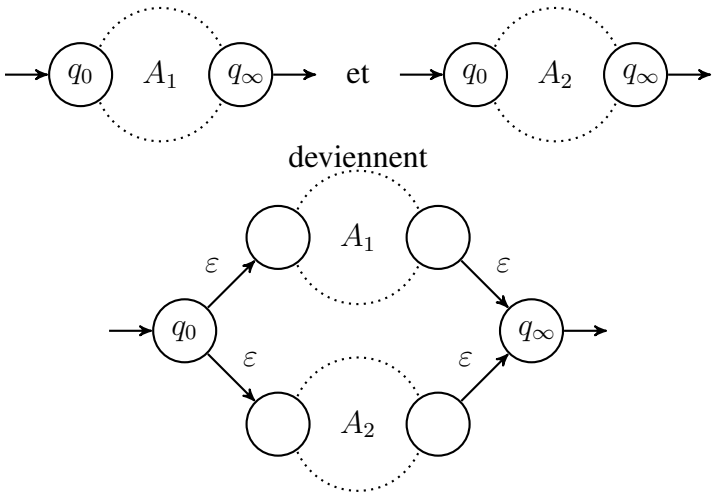


pour le langage $\varnothing$ (i.e., pour l'expression rationnelle $\bot$),

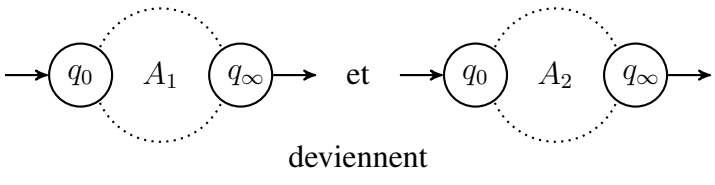
pour le langage $\{\varepsilon\}$ (i.e., pour l'expression rationnelle $\underline{\varepsilon}$), et

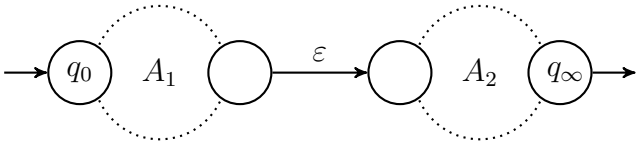
pour le langage $\{x\}$ (i.e., pour l'expression rationnelle x).

3.3.3. Si A_1 et A_2 sont les automates de Thompson pour les expressions rationnelles r_1 et r_2, celui de $(r_1|r_2)$ sera construit de la manière suivante :

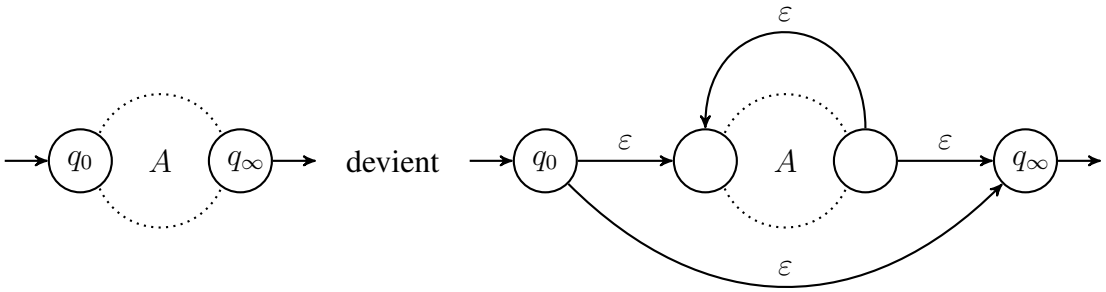


3.3.4. Si A_1 et A_2 sont les automates de Thompson pour les expressions rationnelles r_1 et r_2, celui de $r_1 r_2$ sera construit de la manière suivante :



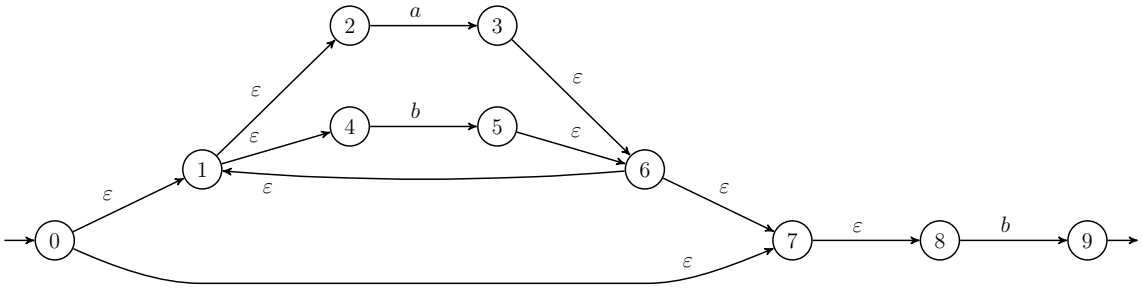


3.3.5. Si A est l'automate de Thompson pour l'expression rationnelle r, celui de $(r)*$ sera construit de la manière suivante :

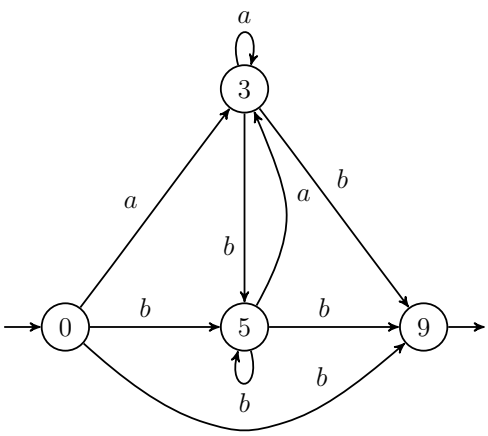


3.3.6. Comme on le voit ci-dessus, la construction de Thompson est très simple à appliquer ; mais elle conduit à des automates rapidement énormes, comportant un nombre considérable d'états et de transitions spontanées « stupides ».

À titre d'exemple, voici l'automate de Thompson, déjà gros, de l'expression rationnelle $(a|b)*b$:



(Il a 10 états puisqu'il y a 5 symboles autres que les parenthèses dans $(a|b)*b$.)
Pour comparaison, voici son automate de Glushkov :



Il a 4 états puisqu'il y a 3 lettres dans $(a|b)*b$. Ces états ont été étiquetés de manière à illustrer la proposition suivante, qui fait le lien entre les deux constructions :

Proposition 3.3.7. L'élimination des transitions spontanées (au sens de 2.4.6, suivie de la suppression des états devenus inutiles) dans l'automate de Thompson d'une expression rationnelle conduit à l'automate de Glushkov de cette même expression.

Sans que cela constitue une démonstration, on peut comprendre l'idée en considérant les « remarques » qui ont été faites dans les démonstrations de 3.2.3, 3.2.4 et 3.2.5.

3.4 Automates à transitions étiquetées par des expressions rationnelles (=RNFA), algorithme d'élimination des états

3.4.1. On cherche dans cette section à montrer la réciproque de 3.2.7, c'est-à-dire, que les langages rationnels sont reconnaissables. On va pour cela donner un algorithme (très coûteux !) qui transforme un automate en expression rationnelle (dénotant le langage qu'il reconnaît). Cette algorithme « d'élimination des états » fonctionne naturellement sur une sorte d'automate encore plus générale que tous ceux que nous avons définis jusqu'à présent : on va donc commencer par définir ces automates, même si leur intérêt réside presque uniquement en la preuve de 3.4.7 ci-dessous.

3.4.2. Un **automate fini (non-déterministe) à transitions étiquetées par des expressions rationnelles**, en abrégé **RNFA**, sur un alphabet Σ est la donnée

— d'un ensemble fini Q d'états,
— d'un ensemble $I \subseteq Q$ d'états dits initiaux,
— d'un ensemble $F \subseteq Q$ d'états dits finaux,
— d'un ensemble *fini* de transitions $\delta \subseteq Q \times (\mathrm{regexp}(\Sigma)) \times Q$ où $(\mathrm{regexp}(\Sigma))$ désigne l'ensemble des expressions rationnelles sur Σ.

Autrement dit, on autorise maintenant des transitions étiquetées par des expressions rationnelles quelconques sur Σ. Remarquons qu'on doit maintenant demander *explicitement* que l'ensemble δ des transitions permises soit fini car l'ensemble $Q \times (\mathrm{regexp}(\Sigma)) \times Q$, lui, ne l'est pas.

3.4.3. Pour un tel automate, on définit une relation $\delta^* \subseteq Q \times \Sigma^* \times Q$ par $(q, w, q') \in \delta^*$ lorsqu'il existe $q_0, \ldots, q_n \in Q$ et $r_1, \ldots, r_n \in \mathrm{regexp}(\Sigma)$ tels que $q_0 = q$ et $q_n = q'$ et $(q_{i-1}, r_i, q_i) \in \delta$ pour chaque $1 \leq i \leq n$, et enfin $w \in L(r_1 \cdots r_n)$.

Concrètement, $(q, w, q') \in \delta^*$ signifie que le RNFA peut passer de l'état q à l'état q' en effectuant des transitions ($q_0 \to q_1 \to \cdots \to q_n$ étiquetées par $r_1, \ldots, r_n \in \mathrm{regexp}(\Sigma)$) et en consommant le mot w au sens où ce dernier se décompose comme concaténation d'autant de facteurs que de transitions ($w = v_1 \cdots v_n$), chacun vérifiant l'expression rationnelle qui étiquette la transition (soit $v_i \in L(r_i)$).

Enfin, l'automate A accepte un mot w lorsqu'il existe $q_0 \in I$ et $q_\infty \in F$ tels que $(q_0, w, q_\infty) \in \delta^*$.

Le langage accepté $L(A)$ et l'équivalence de deux automates sont définis de façon analogue aux DFA (cf. 2.1.6).

3.4.4. Un εNFA (ou *a fortiori* un NFA, DFAi ou DFA) est considéré comme un RNFA particulier dont les transitions sont étiquetées soit par une unique lettre (considérée comme

expression rationnelle) soit par le symbole $\underline{\varepsilon}$ (dénotant le langage $\{\varepsilon\}$) dans le cas des transitions spontanées.

Une expression rationnelle r peut aussi être considérée comme un RNFA particulier comportant un unique état initial, un unique état final, et une unique transition de l'un vers l'autre, étiquetée par l'expression r elle-même. Il est évident que ce RNFA reconnaît exactement le langage dénoté par r.

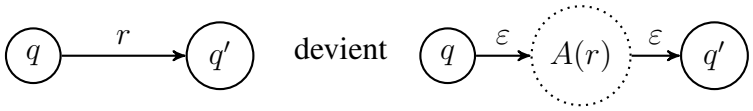
La représentation graphique des RNFA ne pose pas de problème particulier (voir en 3.4.9 pour différents exemples).

3.4.5. On peut toujours modifier un RNFA de manière à ce qu'il y ait au plus une, ou même si on le souhaite, exactement une, transition entre deux états q et q' donnés. En effet, s'il existe plusieurs transitions $(q, r_1, q'), \ldots, (q, r_k, q') \in \delta$ possibles entre q et q', on peut les remplacer par une unique transition $(q, (r_1| \cdots |r_k), q')$, cela ne change visiblement rien au fonctionnement de l'automate (et notamment pas le langage reconnu). S'il n'y a *aucune* transition de q vers q', on peut toujours choisir d'en ajouter une $(q, \bot, q')$ (qui ne peut pas être empruntée !) si c'est commode.

Comme les εNFA, les NFA et les DFAi avant eux, les RNFA peuvent se ramener aux automates précédemment définis :

Proposition 3.4.6. Soit $A = (Q, I, F, \delta)$ un RNFA sur un alphabet Σ. Alors il existe un εNFA $A' = (Q', I', F', \delta')$ (sur le même alphabet Σ) et qui soit équivalent à A au sens où il reconnaît le même langage $L(A') = L(A)$. De plus, A' se déduit algorithmiquement de A.

Démonstration. On a vu en 3.2.7 que pour chaque expression rationnelle r on peut trouver (algorithmiquement) un εNFA $A(r)$ (par exemple l'automate de Glushkov ou l'automate de Thompson) qui reconnaît le langage dénoté par r. On peut donc construire A' en remplaçant chaque transition (q, r, q') de A par une copie de l'automate $A(r)$ placée entre les états q et q'. Symboliquement :



Plus précisément, si $\{(q_j, r_j, q'_j) : 1 \leq j \leq M\}$ est une énumération de δ, on construit A' en lui donnant pour ensemble d'états $Q \uplus \biguplus_{j=1}^{M} Q_{r_j}$ où Q_{r_j} est l'ensemble d'états de l'automate $A(r_j)$ construit pour reconnaître r_j, les ensembles d'états initiaux et finaux sont $I' = I$ et $F' = F$ comme dans A, et la relation de transition δ' est la réunion de chacune δ_{r_j} de celle des εNFA $A(r_j)$ à quoi on ajoute encore des transitions spontanées $(q_j, \varepsilon, q^\sharp)$ pour tout état initial $q^\sharp \in I_{r_j}$ de $A(r_j)$ et des transitions spontanées $(q^\flat, \varepsilon, q'_j)$ pour tout état final $q^\flat \in F_{r_j}$ de $A(r_j)$. Il est clair que faire un chemin dans A' revient à un faire un chemin dans A où, à chaque fois qu'on fait la transition $q_j \to q'_j$ étiquetée par r_j, on la remplace par un chemin $q_j \to q^\sharp \to \cdots \to q^\flat \to q'_j$ formé d'une transition spontanée vers un état initial de $A(r_j)$ suivi d'un chemin dans ce dernier, suivi d'une transition spontanée depuis un état final de $A(r_j)$. ☺

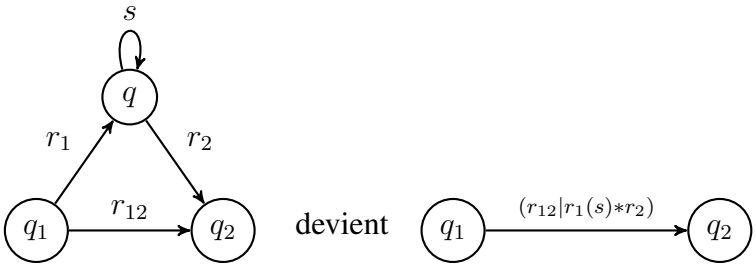
Mais la surprise des RNFA est qu'ils peuvent aussi se ramener à des expressions rationnelles !

Proposition 3.4.7. Soit $A = (Q, I, F, \delta)$ un RNFA (ou, en particulier, un NFA ou DFA(i)) sur un alphabet Σ. Alors il existe une expression rationnelle r sur Σ qui dénote le langage reconnu par A, soit $L(r) = L(A)$. De plus, r se déduit algorithmiquement de A.

Démonstration. On a expliqué qu'on pouvait considérer une expression rationnelle comme un RNFA ayant un unique état initial, un unique état final, et une unique transition de l'un vers l'autre (étiquetée par l'expression rationnelle en question). On va montrer que A est équivalent à un RNFA de cette nature, ce qui montrera bien qu'il est équivalent à une expression rationnelle.

Remarquons tout d'abord qu'on peut supposer que A a un unique état initial q_0, qui ne soit pas final, et qui n'ait aucune transition qui y aboutisse (si ce n'est pas le cas, il suffit de créer un nouvel état q_0, d'en faire le seul état initial, et de le munir de transitions spontanées — c'est-à-dire étiquetées par ε — vers tous les états précédemment initiaux). De même (symétriquement), on peut supposer que A a un unique état final q_∞, qui ne soit pas initial, et sans aucune transition qui en part. On fera l'hypothèse que A a ces propriétés, et on s'arrangera pour les préserver dans ce qui suit.

Soient maintenant q un état de A qui n'est ni l'état initial q_0 ni l'état final q_∞. On va montrer qu'on peut *éliminer* q, c'est-à-dire, quitte à ajouter des transitions, remplacer A par un automate équivalent A' qui n'a pas cet état. Pour cela, soient q_1, q_2 deux états quelconques de A, autres que q mais possiblement égaux entre eux, où q_1 peut être l'état initial (mais pas l'état final) et q_2 peut être l'état final (mais pas l'état initial). On a vu en 3.4.5 qu'on pouvait supposer qu'il existait une unique transition (q_1, r_{12}, q_2) et de même (q_1, r_1, q) et (q, r_2, q_2) et (q, s, q) (transition de q vers lui-même). En même temps qu'on élimine q, on met dans A' la transition $(r_{12}|r_1(s)*r_2)$ entre q_1 et q_2. Symboliquement :



Cette transformation doit être effectuée *simultanément pour toute paire* [14] (q_1, q_2) d'états de A pour laquelle $q_1 \notin \{q, q_\infty\}$ et $q_2 \notin \{q, q_0\}$: pour chaque telle paire, on remplace l'étiquette de la transition r_{12} entre eux par $(r_{12}|r_1(s)*r_2)$. Ceci ne change pas le fonctionnement de l'automate, car tout chemin dans A peut être remplacé par un chemin dans A' en effaçant simplement les q (si on considère q_1 et q_2 les états avant un bloc de q dans le chemin, on voit que le chemin $q_1 \to q \to q \to \cdots \to q \to q_2$ peut se transformer en $q_1 \to q_2$ en consommant un mot qui vérifie l'expression rationnelle $(r_{12}|r_1(s)*r_2)$).

En éliminant (dans n'importe quel ordre) tous les états autres que q_0 et q_∞, on aboutit ainsi à un automate ayant une unique transition (q_0, r, q_∞), qui est donc essentiellement l'expression rationnelle r. ☺

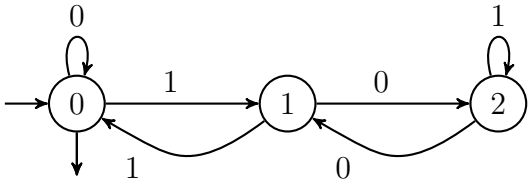
14. Plutôt qu'examiner toutes les paires, on peut se contenter d'examiner les cas où *soit* il existe une transition de q_1 vers q_2 *soit* il existe à la fois une transition de q_1 vers q et une transition de q vers q_2. En revanche, insistons bien sur le fait que le cas où q_1 et q_2 coïncide doit être traité comme les autres.

3.4.8. La procédure qu'on a décrite dans la démonstration de cette proposition s'appelle l'algorithme d'**élimination des états** ou **algorithme de Kleene** [15].

Il va de soi qu'on peut la simplifier un petit peu : s'il n'y a pas de transition de q_1 vers q ou qu'il n'y en a pas de q vers q_2 (c'est-à-dire que soit r_1 soit r_2 doit être considéré comme valant $\perp$), on ne touche simplement pas à r_{12} (et si la transition de q_1 vers q_2 n'existait pas non plus, il n'y a pas besoin de la créer) ; de même, s'il n'y a pas de transition de q vers lui-même, on ignore la partie $s*$. En revanche, il faut bien penser à créer une transition de q_1 vers q_2, même si elle n'existait pas au départ, lorsqu'on peut arriver de l'un vers l'autre en passant par q. Et il faut se souvenir que le cas $q_2 = q_1$ est à traiter aussi.

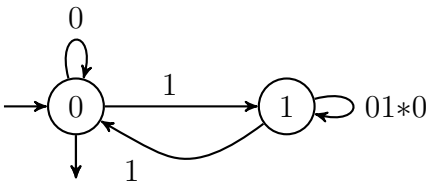
En général, l'élimination des états conduit à une expression extrêmement compliquée (exponentielle dans le nombre d'états de l'automate, au moins dans le pire cas, mais aussi dans beaucoup de cas « typiques »).

3.4.9. À titre d'exemple, considérons le DFA suivant sur l'alphabet $\{0, 1\}$, qui reconnaît les suites binaires qui représentent un nombre multiple de 3 écrit en binaire [16] (en convenant que le mot vide est une représentation binaire du nombre 0, ce qui est logique) :



On commence par ajouter un état initial q_0 et un état final q_∞, avec des ε-transitions $q_0 \to 0$ et $0 \to q_\infty$. Pour gagner de la place, nous ne figurerons pas ces deux états sur les dessins qui suivent, mais il faut s'imaginer qu'ils sont toujours là.

L'élimination de l'état 2 conduit à l'automate suivant :



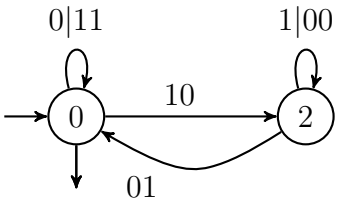
(Répétons qu'on n'a pas figuré l'état initial q_0 ni l'état final q_∞ : les flèches vers et depuis l'état 0 doivent se comprendre comme des ε-transitions $q_0 \to 0$ et $0 \to q_\infty$.)

L'élimination de l'état 1 conduit alors à l'automate ayant un unique état 0, avec une transition vers lui-même étiquetée $0|1(01*0)*1$. Enfin, en éliminant l'état 0, il ne reste qu'une transition de l'état initial vers l'état final, étiquetée par $(0|1(01*0)*1)*$: le langage reconnu par l'automate de départ est donc celui dénoté par l'expression rationnelle $(0|1(01*0)*1)*$.

On pouvait aussi choisir d'éliminer l'état 1 en premier, ce qui conduit à l'automate suivant :

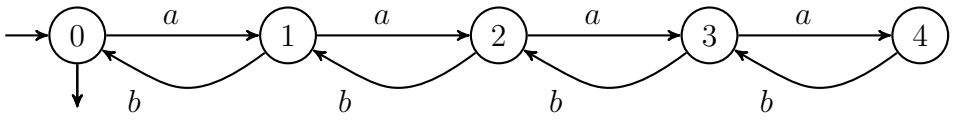
15. Peut-être abusivement (le théorème 3.4.11 est indubitablement dû à Kleene, mais le contenu algorithmique ne l'est peut-être pas) ; il est peut-être plus correct d'attribuer l'algorithme à Brzozowski et McCluskey.

16. Les états $0, 1, 2$ représentent respectivement les états dans lesquels le nombre binaire lu jusqu'à présent par l'automate est congru à $0, 1, 2$ modulo 3.



et finalement à l'expression rationnelle $(0|11|10(1|00)*01)*$, qui est équivalente à la précédente.

3.4.10. Donnons encore l'exemple du DFAi suivant :



Il s'agit d'un automate « compteur limité », qui ne sait compter que de 0 à 4, incrémentant son compteur quand il reçoit un a et le décrémentant quand il reçoit un b (et cessant de fonctionner si le compteur passe au-dessus du maximum ou en-dessous du minimum), et qui accepte finalement les mots dont le nombre de b égale le nombre de a sans qu'il y ait jamais eu plus de b que de a ni plus de quatre a de plus que de b. (On peut dire aussi qu'il s'agit d'une *approximation* du langage des expressions bien-parenthésées définies en 4.3.4 plus loin, où a joue le rôle de parenthèse ouvrante et b de parenthèse fermante ; l'approximation est due au fait qu'on n'accepte que quatre niveaux d'imbrication des « parenthèses ».)

Si on élimine les états dans l'ordre $4, 3, 2, 1, 0$, on montre que le langage reconnu par cet automate est décrit par l'expression rationnelle $(a(a(a(ab)*b)*b)*b)*$. Si on les élimine dans l'ordre $0, 1, 2, 3, 4$, en revanche, on obtient une expression de taille considérable [17] et beaucoup moins transparente.

Récapitulons le contenu essentiel à retenir comme conséquence immédiate de 3.2.7 et 3.4.7 :

Théorème 3.4.11 (Kleene)**.** La classe des langages rationnels et celle des langages reconnaissables coïncident. (On pourra donc considérer ces termes comme synonymes.)

Il faut cependant retenir que s'il y a, mathématiquement, équivalence entre ces deux classes de langages, cette équivalence *a un coût algorithmique*, c'est-à-dire que la conversion d'une expression rationnelle en automate (surtout si on souhaite un DFA), ou à plus forte raison d'un automate en expression rationnelle, a une complexité exponentielle dans le pire cas. Il est donc pertinent, en informatique, de ne pas considérer les descriptions d'un langage par une expression rationnelle, un DFA, et un NFA, comme interchangeables.

3.4.12. Une conséquence de 3.4.11 est, par exemple, que le complémentaire d'un langage rationnel est rationnel (cf. 3.1.2) ou que l'intersection de deux langages rationnels est rationnelle (cf. 3.1.4), et que ces opérations se calculent algorithmiquement. Il est donc possible, en principe, à partir d'une expression rationnelle r (sur un alphabet Σ), de fabriquer

17. À savoir : $\varepsilon \mid a(ba)*b \mid a(ba)*a(b(ba)*a)*b(ba)*b \mid a(ba)*a(b(ba)*a)*a(b(b(ba)*a)*a)*b(b(ba)*a)*b(ba)*b \mid a(ba)*a(b(ba)*a)*a(b(b(ba)*a)*a)*a(b(b(b(ba)*a)*a)*a)*b(b(b(ba)*a)*a)*b(b(ba)*a)*b(ba)*b$ (mais si on la regarde d'assez près, on peut comprendre comment elle fonctionne).

algorithmiquement une expression rationnelle r' (« négation » de r) qui dénote le langage complémentaire de celui dénoté par r ; et de même, à partir d'expressions rationnelles r_1, r_2, de fabriquer algorithmiquement une expression rationnelle r'' (« conjonction » de r_1 et r_2) qui dénote le langage intersection de ceux dénotés par r_1 et r_2.

Le coût de ces opérations, cependant, est astronomique : doublement exponentiel, puisqu'il s'agit de convertir l'expression en NFA, de déterminiser le NFA en DFA (à un premier coût exponentiel), d'effectuer l'opération sur un DFA, et de reconvertir le DFA en expression rationnelle (à un deuxième coût exponentiel). Ce n'est pas faute d'avoir les bons algorithmes : on peut montrer qu'il existe effectivement des familles d'expressions rationnelles pour lesquelles la négation ou la conjonction produisent une augmentation de taille doublement exponentielle.

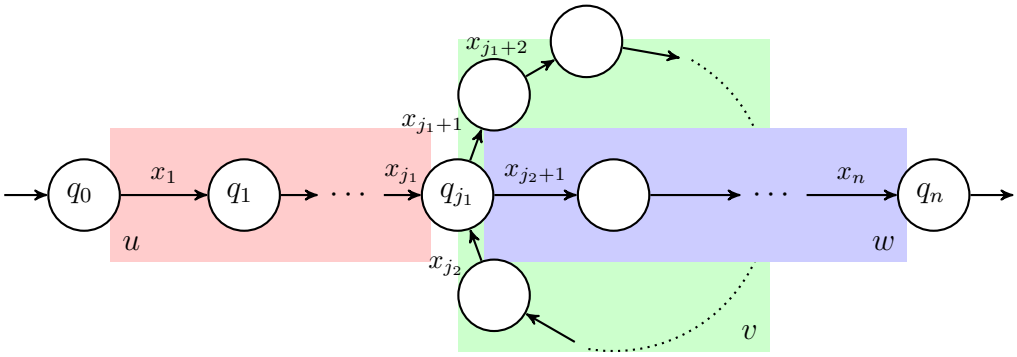
3.5 Le lemme de pompage

3.5.1. On ne dispose à ce stade-là d'aucun moyen pour montrer qu'un langage *n'est pas* rationnel. La proposition 3.5.2 qui va suivre, et qui s'appelle couramment « lemme de pompage » (une traduction abusive de l'anglais « pumping lemma ») constitue le moyen le plus fréquent permettant d'y arriver : il énonce une condition *nécessaire* pour qu'un langage soit rationnel, si bien qu'on peut arriver à montrer qu'un langage n'est pas rationnel en invalidant cette condition (généralement en procédant par l'absurde).

Proposition 3.5.2 (lemme de pompage pour les langages rationnels)**.** Soit L un langage rationnel. Il existe alors un entier k tel que tout mot de $t \in L$ de longueur $|t| \geq k$ admette une factorisation $t = uvw$ en trois facteurs $u, v, w \in \Sigma^*$ où :

(i) $|v| \geq 1$ (c'est-à-dire $v \neq \varepsilon$),
(ii) $|uv| \leq k$,
(iii) pour tout $i \geq 0$ on a $uv^i w \in L$.

Démonstration. Soit A un DFA (complet) qui reconnaît L, et soit k son nombre d'états : on va montrer que k vérifie les propriétés énoncées. Pour cela, soit $t = x_1 \cdots x_n$ un mot de L de longueur $n \geq k$, et soient $q_0, \ldots, q_n$ les états traversés par A pendant la consommation de t, autrement dit, q_0 est l'état initial, et $q_j = \delta(q_{j-1}, x_j)$ pour chaque $1 \leq j \leq n$; l'état $q_n = \delta^*(q_0, t)$ est final puisque $t \in L$. Comme $n + 1 > k$ et comme l'automate A a k états, par le principe des tiroirs, il existe $j_1 \neq j_2$ tels que $q_{j_1} = q_{j_2}$: pour être plus précis, soit j_2 le plus petit possible tel que les états $q_0, \ldots, q_{j_2}$ ne soient pas tous distincts, autrement dit, le premier état répété, et soit j_1 la précédente occurrence (forcément unique) de cet état, c'est-à-dire l'indice tel que $j_1 < j_2$ et $q_{j_1} = q_{j_2}$.



Posons $u = x_1 \cdots x_{j_1}$ (le préfixe de t de longueur j_1, qui est le mot vide si $j_1 = 0$) et $v = x_{j_1+1} \cdots x_{j_2}$ (de longueur $j_2 - j_1$), et enfin $w = x_{j_2+1} \cdots x_n$ (le suffixe de t de longueur $n - j_2$, avec la convention $w = \varepsilon$ si $j_2 = n$). Ceci définit bien une factorisation $t = uvw$.

On a bien (i) $|v| \geq 1$ puisque $j_2 > j_1$. On a par ailleurs (ii) $|uv| \leq k$ puisque $|uv| = j_2$ et que les j_2 états $q_0, \ldots, q_{j_2-1}$ sont distincts (c'est la minimalité de j_2) de sorte que $j_2 \leq k$ (toujours par le principe des tiroirs).

Montrons enfin (iii). On rappelle tout d'abord que $\delta^*(q, x_1 \cdots x_j) = \delta(\cdots \delta(\delta(q, x_1), x_2) \cdots, x_j)$. Remarquons que $\delta^*(q_0, u) = \delta^*(q_0, x_1 \cdots x_{j_1}) = q_{j_1}$, et que $\delta^*(q_{j_1}, v) = \delta^*(q_{j_1}, x_{j_1+1} \cdots x_{j_2}) = q_{j_2} = q_{j_1}$. De cette dernière égalité, on tire $\delta^*(q_{j_1}, v^i) = q_{j_1}$ pour tout $i \geq 0$ (par récurrence sur i). Enfin, $\delta^*(q_{j_1}, w) = \delta^*(q_{j_2}, w) = \delta^*(q_{j_2}, x_{j_2+1} \cdots x_n) = q_n$ (qui est un état final). En mettant ces faits ensemble, on a $\delta^*(q_0, uv^iw) = \delta^*(q_{j_1}, v^iw) = \delta^*(q_{j_1}, w) = q_n$, et puisque q_n est final, ceci montre que le mot uv^iw est accepté par A, i.e., $uv^iw \in L$. ☺

3.5.3. On attire l'attention sur l'alternation des quantificateurs. Le lemme de pompage énonce le fait que :

— *pour tout* langage rationnel L,
— *il existe* un entier $k \geq 0$ tel que
— *pour tout* mot $t \in L$ de longueur $|t| \geq k$,
— *il existe* une factorisation $t = uvw$ vérifiant les propriétés (i) $|v| \geq 1$, (ii) $|uv| \leq k$ et (iii) qui suit :
— *pour tout* $i \geq 0$ on a $uv^iw \in L$.

La complexité logique d'un énoncé étant justement mesurée par le nombre d'alernations de quantificateurs (passages entre « pour tout » et « il existe » ou vice versa), celui-ci mérite une attention particulière. Rappelons donc, du point de vue logique, que, quand on veut *appliquer* un résultat de ce genre, on *choisit librement* les objets introduits par un quantificateur universel (« pour tout »), mais *on ne choisit pas* ceux qui sont introduits par un quantificateur existentiel (« il existe ») (ces derniers sont, si on veut, choisis par l'énoncé qu'on applique : on ne fait que recevoir leur existence) ; les choses sont inversées quand on doit démontrer un tel énoncé, mais la démonstration a été faite ci-dessus et il est donc plus fréquent de devoir appliquer le lemme de pompage.

3.5.4. Le modèle d'une démonstration par l'absurde pour montrer qu'un langage L n'est pas rationnel est donc quelque chose comme ceci :

— on entame un raisonnement par l'absurde en supposant que L est rationnel, et on choisit L pour appliquer le lemme de pompage (parfois on l'applique à autre chose, comme l'intersection de L avec un langage connu pour être rationnel, mais en général ce sera L),
— le lemme de pompage fournit un k (on *ne choisit donc pas* ce k, il est donné par le lemme),
— on choisit alors un mot $t \in L$ de longueur $\geq k$, et c'est là que réside la difficulté principale de la démonstration,
— le lemme de pompage fournit une factorisation $t = uvw$, qu'on *ne choisit pas* non plus mais qu'on peut analyser, souvent en utilisant (i) et (ii),
— et on cherche à appliquer la propriété (iii), ce qui implique de choisir un i, pour arriver à une contradiction (typiquement : le mot uv^iw n'est pas dans le langage alors qu'il est censé y être).

Donnons maintenant un exemple d'utilisation du lemme :

Proposition 3.5.5. Soit $\Sigma = \{a, b\}$. Le langage $L = \{a^n b^n : n \in \mathbb{N}\} = \{\varepsilon, ab, aabb, aaabbb, \ldots\}$ constitué des mots formés d'un certain nombre (n) de a suivis du même nombre de b n'est pas rationnel.

Démonstration. Appliquons la proposition 3.5.2 au langage L considéré : appelons k l'entier dont le lemme de pompage garantit l'existence. Considérons le mot $t := a^k b^k$: il doit alors exister une factorisation $t = uvw$ pour laquelle on a (i) $|v| \geq 1$, (ii) $|uv| \leq k$ et (iii) $uv^i w \in L$ pour tout $i \geq 0$. La propriété (ii) assure que uv est formé d'un certain nombre de répétitions de la lettre a (car tout préfixe de longueur $\leq k$ de $a^k b^k$ est de cette forme) ; disons $u = a^\ell$ et $v = a^m$, si bien que $w = a^{k-\ell-m} b^k$. La propriété (i) donne $m \geq 1$. Enfin, la propriété (iii) affirme que le mot $uv^i w = a^{k+(i-1)m} b^k$ appartient à L ; mais dès que $i \neq 1$, ceci est faux : il suffit donc de prendre $i = 0$ pour avoir une contradiction. ☺

3.5.6. L'idée intuitive derrière la démonstration qu'on vient de faire est la suivante : un automate fini ne dispose que d'une quantité finie (bornée) de mémoire, donc ne peut « compter » que jusqu'à un nombre borné (au-delà, il va retomber sur un état déjà atteint par un plus petit nombre de a, et sera incapable de vérifier si le nombre de b est égal). C'est ce type de raisonnement que le lemme de pompage permet de formaliser. Généralement parlant, on doit garder à l'esprit le fait que toutes sortes de langages ne sont pas rationnels pour la raison informelle que les identifier demande une quantité de mémoire qui pourrait être arbitrairement grande, et que pour le montrer rigoureusement, le lemme de pompage sera souvent utile.

3.6 L'automate minimal, et la minimisation

3.6.1. On sait maintenant convertir une expression rationnelle en un automate équivalent, et réciproquement convertir un automate en expression rationnelle équivalente. Il reste un problème auquel nous n'avons pas donné de réponse : comment savoir si deux automates *donnés* ou deux expressions rationnelles données (ou un de chaque) sont équivalents ?

Pour cela, on va introduire un DFA particulier, *canonique*, associé à un langage rationnel, qu'on pourra calculer algorithmiquement, et qui sera véritablement associé au langage, c'est-à-dire que deux descriptions équivalentes du même langage donneront exactement le *même* automate canonique ; par conséquent, pour tester l'égalité de deux langages, quelle que soit leur description, il suffira de calculer leurs automates canoniques et de les comparer. En fait, cet automate canonique est simplement le DFA ayant le plus petit nombre d'états reconnaissant le langage en question ; ce qui est remarquable, et qui n'est pas du tout évident *a priori*, c'est qu'il est effectivement canonique (c'est-à-dire qu'il n'existe qu'un seul DFA ayant un nombre minimal d'états reconnaissant le langage) et qu'on peut le calculer algorithmiquement.

Théorème 3.6.2 (Myhill-Nerode)**.** Soit L un langage. Pour $w \in \Sigma^*$, notons $w^{-1}L := \{t \in \Sigma^* : wt \in L\}$ (autrement dit, l'ensemble des mots qu'on peut concaténer à w pour obtenir un mot de L). Considérons la relation d'équivalence $\equiv$ sur Σ^* définie par $u \equiv v$ si et seulement si $u^{-1}L = v^{-1}L$; ce qui signifie, si on préfère, que $\forall t \in \Sigma^* \, ((ut \in L) \iff (vt \in L))$.

Alors :

— le langage L est rationnel si et seulement si la relation d'équivalence $\equiv$ possède un nombre *fini* k de classes d'équivalence,

— lorsque c'est le cas, il existe un DFA (complet) ayant k états qui reconnaît L, il est unique à renommage des états [18] près, et il n'existe pas de DFA (complet) ayant $< k$ états qui reconnaisse L.

Démonstration. Supposons d'abord que l'ensemble $\Sigma^*/\equiv$ des classes d'équivalence pour $\equiv$ soit fini : appelons-le Q, et expliquons comment on peut construire un DFA reconnaissant L et dont l'ensemble des états soit Q. Notons $[u] := \{v \in \Sigma^* : u \equiv v\}$ pour la classe d'équivalence de u pour $\equiv$. Posons $q_0 := [\varepsilon]$ la classe du mot vide. Remarquons que si $u \equiv v$ (pour $u, v \in \Sigma^*$), alors on a $u \in L$ si et seulement si $v \in L$ (en effet, la définition de $\equiv$ est que $(ut \in L) \iff (vt \in L)$ pour tout t, et on applique ça à $t = \varepsilon$) : autrement dit, une classe $[u] \in Q$ est soit entièrement incluse dans L soit disjointe de L ; appelons F l'ensemble $\{[u] : u \in L\}$ des classes incluses dans L. Enfin remarquons que si $u \equiv v$ (pour $u, v \in \Sigma^*$) et $x \in \Sigma$, on a encore $ux \equiv vx$ (en effet, $uxt \in L \iff vxt \in L$ pour tout $t \in L$) : il y a donc un sens à définir $\delta([u], x) = [ux]$. On a ainsi fabriqué un DFA $A = (Q, q_0, F, \delta)$. Vues les définitions de q_0 et δ, il est clair que $\delta^*(q_0, w) = [w]$ pour cet automate, et vue la définition de F, on a $\delta^*(q_0, w) \in F$ si et seulement si $w \in L$. Ceci montre bien que A reconnaît le langage L. On a donc prouvé que si $\Sigma^*/\equiv$ est fini, le langage L est rationnel et même il existe un DFA ayant $k := \#(\Sigma^*/\equiv)$ états qui le reconnaît.

Supposons maintenant que $B = (Q_B, q_{0,B}, F_B, \delta_B)$ soit un DFA reconnaissant L. Définissons une nouvelle relation d'équivalence $\equiv_B$ sur Σ^* par : $u \equiv_B v$ si et seulement si $\delta_B^*(q_{0,B}, u) = \delta_B^*(q_{0,B}, v)$ (autrement dit, les mots u et v mettent l'automate B dans le même état). Si on a $u \equiv_B v$, on a aussi $u \equiv v$: en effet, pour tout $t \in L$ on a $\delta_B^*(q_{0,B}, ut) = \delta_B^*(\delta_B^*(q_{0,B}, u), t) = \delta_B^*(\delta_B^*(q_{0,B}, v), t) = \delta_B^*(q_{0,B}, vt)$, et notamment le membre de gauche appartient à F_B si et seulement si le membre de droite y appartient, c'est-à-dire que $ut \in L \iff vt \in L$. On vient donc de montrer que $u \equiv_B v$ implique $u \equiv v$ (la relation $\equiv_B$ est *plus fine* que $\equiv$) : si on préfère, chaque classe d'équivalence pour $\equiv$ est donc une réunion de classes d'équivalences pour $\equiv_B$. Notamment, comme $\equiv_B$ a un nombre fini de classes d'équivalence (puisque $\delta_B^*(q_{0,B}, u)$ ne peut prendre qu'un nombre fini de valeurs, celles dans Q_B), il en va de même de $\equiv$, et plus précisément, comme $\equiv_B$ a au plus $\#Q_B$ classes d'équivalence, il en va de même de $\equiv$, c'est-à-dire $k \leq \#Q_B$. Il n'existe donc pas de DFA ayant $< k$ états reconnaissant L.

Enfin, si B a k états et reconnaît L, cela signifie que les deux relations $\equiv$ et $\equiv_B$ coïncident, et on définit une bijection ψ entre le $Q = Q_A$ du paragraphe précédent et Q_B en associant à une classe $[w] \in Q$ l'état $\psi([w]) := \delta_B^*(q_{0,B}, w)$ (qui ne dépend que de la classe de w pour $\equiv_B$ et on vient de voir que c'est la classe de w modulo $\equiv$, c'est-à-dire $[w]$: ceci est donc bien défini). Cette bijection ψ vérifie $\psi(q_0) = \psi([\varepsilon]) = \delta_B^*(q_{0,B}, \varepsilon) = q_{0,B}$; on a $[w] \in F$ si et seulement si $w \in L$, c'est-à-dire si et seulement si $\delta_B^*(q_{0,B}, w) \in F_B$ autrement dit $\psi([w]) \in F_B$. Et enfin, pour $w \in \Sigma^*$ et $x \in \Sigma$, on a $\psi(\delta([w], x)) = \psi([wx]) = \delta_B^*(q_{0,B}, wx) = \delta_B(\delta_B^*(q_0, w), x) = \delta_B(\psi([w]), x)$. Bref, ψ préserve l'état initial, les états finaux, et la relation de transition : c'est donc bien un isomorphisme d'automates (un renommage des états). ☺

3.6.3. Ce théorème affirme donc qu'il existe (à renommage des états près) un unique DFA (complet) ayant un nombre minimal d'états parmi ceux qui reconnaissent le langage rationnel L : on l'appelle **automate minimal** ou **automate canonique** du langage L. La démonstration ci-dessus en donne une construction à partir d'une relation d'équivalence, mais

18. C'est-à-dire, si on préfère ce terme, isomorphisme d'automates.

cette démonstration n'est pas algorithmique : on va voir comment on peut le construire de façon algorithmique à partir d'un DFA quelconque qui reconnaît L.

Proposition 3.6.4. Soit $B = (Q_B, q_{0,B}, F_B, \delta_B)$ un DFA (complet !) reconnaissant un langage L, et dont tous les états sont accessibles (cf. 2.1.8). Alors l'automate minimal A de L peut s'obtenir en fusionnant dans B chaque classe d'équivalence pour la relation d'équivalence $\equiv$ définie sur Q_B par

$$q \equiv q' \iff \forall t \in \Sigma^* \, (\delta_B^*(q,t) \in F_B \iff \delta_B^*(q',t) \in F_B)$$

(Fusionner chaque classe d'équivalence signifie qu'on construit l'automate A dont l'ensemble d'états est l'ensemble $Q_A := Q_B/\equiv$ des classes d'équivalence, l'état initial $q_{0,A}$ est la classe $[q_{0,B}]$ de $q_{0,B}$ pour $\equiv$, les états finaux sont les classes contenant au moins un état final et qui d'ailleurs sont entièrement constituées d'états finaux, et la relation de transition est donnée par $\delta_A([q], x) = [\delta_B(q, x)]$ en notant $[q]$ la classe de q pour $\equiv$.)

De plus, cet automate A (ou de façon équivalente, la relation $\equiv$) peut se déduire algorithmiquement de B.

Démonstration. On a vu dans le cours de la démonstration de 3.6.2 que l'automate minimal a pour ensemble d'états $\Sigma^*/\equiv_L$ où $\equiv_L$ désigne la relation d'équivalence (alors notée $\equiv$) définie par $u \equiv_L v$ lorsque $\forall t \in \Sigma^*((ut \in L) \iff (vt \in L))$. Si $q, q' \in Q_B$, comme q, q' sont accessibles, il existe $w, w' \in \Sigma^*$ tels que $\delta_B^*(q_{0,B}, w) = q$ et $\delta_B^*(q_{0,B}, w') = q'$; et on a $q \equiv q'$ si et seulement si $\delta_B^*(q,t) \in F_B \iff \delta_B^*(q',t) \in F_B$ pour tout $t \in \Sigma^*$, c'est-à-dire $\delta_B^*(q_{0,B}, wt) \in F_B \iff \delta_B^*(q_{0,B}, w't) \in F_B$, c'est-à-dire $wt \in L \iff w't \in L$, autrement dit, $w \equiv_L w'$. L'application φ qui à un état $[w]_L$ de l'automate minimal associe la classe de $\delta_B^*(q_{0,B}, w)$ pour $\equiv$ est donc une bijection, et comme dans la démonstration de 3.6.2 on vérifie qu'elle préserve l'état initial, les états finaux, et la relation de transition. On obtient donc bien l'automate minimal en fusionnant chaque classe d'équivalence pour $\equiv$.

Montrons maintenant comment on peut construire $\equiv$ algorithmiquement. Pour cela, on va définir des relations d'équivalence $\equiv_i$ pour $i \in \mathbb{N}$ par

$$q \equiv_i q' \iff \forall t \, (|t| \leq i \implies (\delta_B^*(q,t) \in F_B \iff \delta_B^*(q',t) \in F_B))$$

C'est-à-dire par récurrence

$$\begin{aligned} q \equiv_0 q' &\iff (q \in F_B \iff q' \in F_B) \\ q \equiv_{i+1} q' &\iff (q \equiv_i q' \text{ et } \forall x \in \Sigma \, (\delta_B(q,x) \equiv_i \delta_B(q',x))) \end{aligned}$$

(la première équivalence vient de ce que $\delta_B^*(q, \varepsilon) = q$, et la seconde de ce que $\delta_B^*(q, xt) = \delta_B^*(\delta_B(q,x), t)$).

Il est trivial que $q \equiv_{i+1} q'$ implique $q \equiv_i q'$, c'est-à-dire que $\equiv_{i+1}$ est plus fine que $\equiv_i$, mais $\equiv$ est plus fine que toutes, et comme elle n'a qu'un nombre fini de classes d'équivalence, le nombre de classes ne peut croître strictement qu'un nombre fini de fois, et il doit donc stationner : il existe i tel que $(\equiv_{i+1}) = (\equiv_i)$, et à ce moment-là, la seconde équivalence de la récurrence montre que $(\equiv_j) = (\equiv_i)$ pour tout $j \geq i$ et donc $(\equiv) = (\equiv_i)$.

On peut donc calculer $\equiv$ selon l'algorithme suivant : calculer $\equiv_0$, et par récurrence calculer les $\equiv_i$ jusqu'à ce que la relation ne change plus, $(\equiv_{i+1}) = (\equiv_i)$, auquel cas la dernière relation calculée est la relation recherchée $(\equiv) = (\equiv_i)$, et l'automate minimal A s'obtient en fusionnant chaque classe d'équivalence pour $\equiv$. ☺

3.6.5. L'algorithme décrit par la proposition 3.6.4 porte le nom d'algorithme **de Moore** ou **de minimisation** ou **de réduction**.

Voici comment on peut le mettre en œuvre de façon plus concrète :

— s'assurer qu'on a affaire à un DFA *complet sans état inaccessible* (si nécessaire, déterminiser l'automate s'il n'est pas déterministe, le compléter s'il est incomplet, et supprimer les états inaccessibles s'il y en a) ;

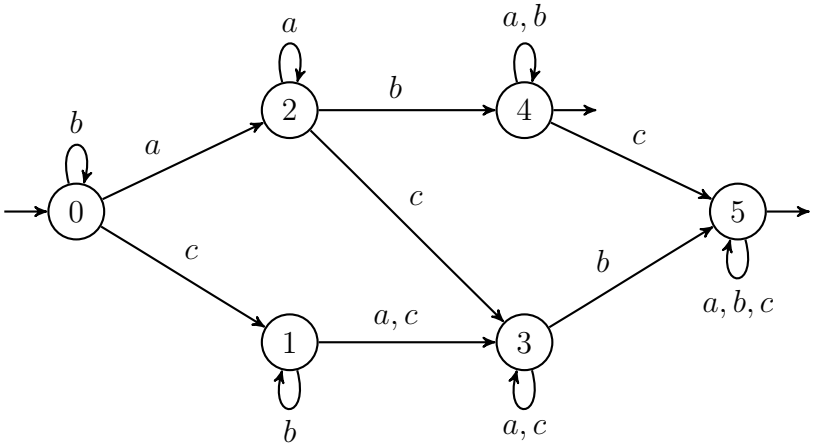
— appeler Π la partition des états en deux classes : les états finaux d'un côté, et les non-finaux de l'autre ;

— répéter l'opération suivante tant que la partition Π change : pour chaque classe C de Π et chaque lettre $x \in \Sigma$, s'il existe deux états $q, q' \in C$ tels que $\delta(q, x)$ et $\delta(q', x)$ ne tombent pas dans la même classe de Π, séparer cette classe C selon la classe de $\delta(—, x)$ (autrement dit, q, q' restent dans la même classe pour la nouvelle partition lorsqu'ils sont dans la même classe pour l'ancienne et que pour chaque lettre x leurs images $\delta(q, x)$ et $\delta(q', x)$ sont aussi dans la même classe) ;

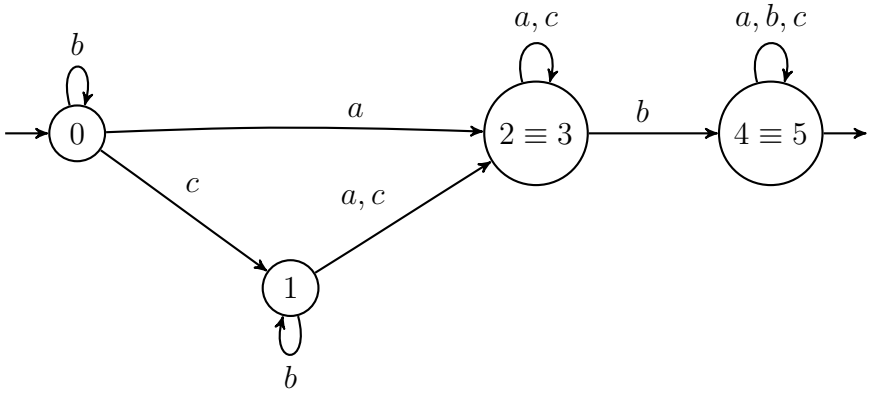
— si Π est la (dernière) partition ainsi obtenue, l'ensemble des états de l'automate construit est l'ensemble des classes de Π, l'état initial est la classe de l'état initial, les états finaux sont les classes qui contiennent un état final (ils sont alors forcément tous finaux), et la fonction de transition est obtenue en prenant la fonction de transition sur un représentant quelconque de la classe (la classe ne doit pas dépendre du représentant).

La dernière étape (construction de l'automate) permet de vérifier qu'on a correctement terminé l'étape précédente (raffinement de la partition) : si deux états dans la même classe ont une transition sortante d'étiquette x et qui mènent vers des classes différentes, c'est que ces états auraient dû être séparés. Il est donc utile de refaire un contrôle à ce niveau.

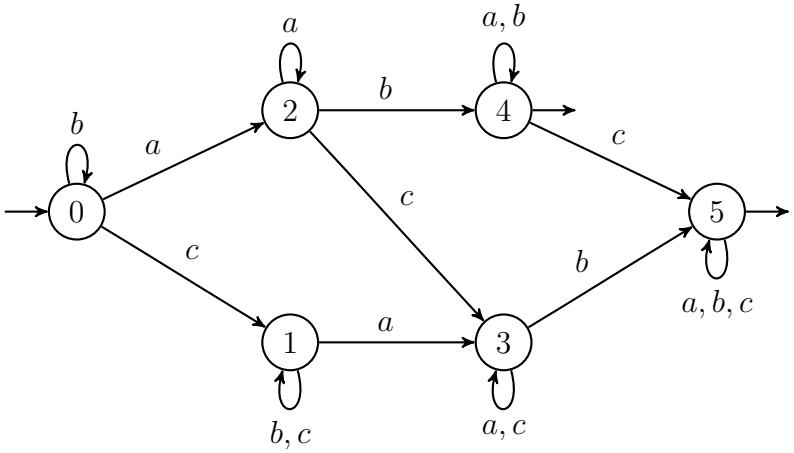
3.6.6. À titre d'exemple d'exécution de l'algorithme de minimisation, considérons l'automate suivant :



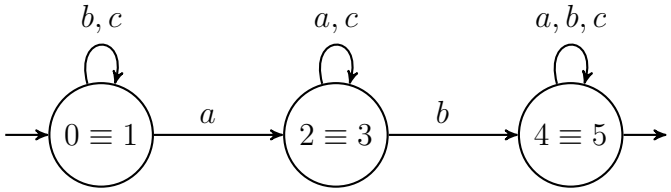
Cet automate est bien déterministe, complet et sans état inaccessible. Dans un premier temps, on partitionne les états en états non-finaux et finaux, soit $\{0, 1, 2, 3\}$ d'un côté et $\{4, 5\}$ de l'autre. Ensuite, on doit séparer la classe $\{0, 1, 2, 3\}$ en deux selon que la transition étiquetée par b tombe dans la classe $\{0, 1, 2, 3\}$ elle-même ou bien dans la classe $\{4, 5\}$: on arrive donc à trois classes, $\{0, 1\}$, $\{2, 3\}$ et $\{4, 5\}$. Enfin, on doit séparer la classe $\{0, 1\}$ en deux selon que la transition étiquetée par c tombe dans la classe $\{0, 1\}$ elle-même ou bien dans la classe $\{2, 3\}$: on arrive alors à quatre classes, $\{0\}$, $\{1\}$, $\{2, 3\}$ et $\{4, 5\}$, et à l'automate minimal suivant :



Il est intéressant de voir comment des petits changements sur l'automate initial modifient la minimisation. Si on fait pointer la transition étiquetée par c de l'état 1 vers lui-même (au lieu d'aller vers l'état 3), c'est-à-dire :



alors la minimisation ne sépare pas les états 0 et 1, et on obtient (le même automate qu'en 2.1.7, à savoir) :



En revanche, si on change l'automate pour rendre l'état 4 non-final (avec ou sans la modification précédemment évoquée), la minimisation aboutit sur la partition triviale en six états, c'est-à-dire que l'automate est, en fait, déjà minimal.

3.6.7. Cas dégénérés : Si aucun état d'un DFA n'est final (c'est-à-dire s'il reconnaît le langage vide $\varnothing$), alors l'algorithme de minimisation termine immédiatement avec une seule classe d'équivalence, et donne donc un automate minimal ayant un unique état, initial mais non final, et des transitions étiquetées par toutes les lettres de cet état vers lui-même. Si *tous* les états d'un DFA sont finaux (il reconnaît le langage Σ^* de tous les mots), de même, l'algorithme de minimisation termine immédiatement avec une seule classe d'équivalence, et donne donc un automate minimal ayant un unique état, à la fois initial et final (et toujours des transitions étiquetées par toutes les lettres de cet état vers lui-même).

Énonçons ici le fait évoqué plus haut comme application de l'algorithme de minimisation :

Corollaire 3.6.8. On peut décider algorithmiquement si deux automates finis (de n'importe quelle sorte), ou deux expressions rationnelles, ou un automate et une expression rationnelle, sont équivalents (au sens de reconnaître/dénoter le même langage).

Démonstration. D'après ce qu'on a déjà vu, et quitte à transformer une expression rationnelle en εNFA (cf. 3.2.7), et quitte à éliminer les ε-transitions (cf. 2.4.6) et à déterminiser (cf. 2.3.8), on peut construire algorithmiquement un DFA reconnaissant le même langage que chacune des deux données. La question devient donc de savoir si deux DFA reconnaissent le même langage. Or d'après 3.6.4, on sait transformer un DFA en DFA minimal reconnaissant le même langage, et d'après 3.6.2 on sait que ce DFA est unique à renumérotation près des états. On est donc ramené au problème suivant : donnés deux DFA A et A' (complets et sans états inaccessibles), trouver s'ils sont le même à renumérotation près (i.e., s'ils sont isomorphes).

La correspondance entre états de A et de A' peut se construire état par état : on fait correspondre l'état initial de A à celui de A', puis pour chaque état q de A mis en correspondance avec un état q' de A', on fait correspondre chacun des états $\delta(q, x)$ avec chacun des états $\delta'(q', x)$ où x parcourt les lettres de l'alphabet. Si on aboutit ainsi à une contradiction (deux états de l'un des automates veulent être mis en correspondance avec le même état de l'autre), on renvoie un échec ; sinon, tous les états de A et ceux de A' seront mis en correspondance bijective, et les langages reconnus sont les mêmes. ☺

4 Grammaires hors contexte

4.0.1. Alors que les langages et expressions rationnelles servent, dans le monde informatique, principalement à définir des outils de recherche de « motifs » (pour la recherche et le remplacement dans un texte, la validation d'entrées, la syntaxe de bas niveau, etc.), les grammaires formelles, dont les plus importantes sont les grammaires *hors contexte* que nous allons maintenant considérer, ont pour principal intérêt de définir des *syntaxes structurées*, par exemple la syntaxe d'un langage informatique (typiquement un langage de programmation). En fait, l'étude des grammaires formelles en général, et des grammaires hors contexte en particulier, a été démarrée [19] en 1956 par le linguiste (et activiste politique) Noam Chomsky pour servir dans l'analyse des langues naturelles ; mais c'est plus en informatique qu'en linguistique qu'elles ont trouvé leur utilité, à commencer surtout par la définition de la syntaxe du langage ALGOL 60.

Cette fois-ci, on ne s'intéressera pas simplement au langage défini (par la grammaire hors contexte, dit langage « algébrique »), mais aussi à la manière dont ces mots s'obtiennent par la grammaire, et donc, à la manière d'*analyser* (en anglais, *to parse*) un mot / programme en une structure de données qui le représente : pour cela, on va définir la notion d'*arbre d'analyse*.

4.0.2. Un exemple typique d'usage de grammaire hors contexte pour définir la syntaxe d'un

19. Certains font remonter leur origine bien plus loin : on peut trouver une forme de grammaire hors contexte dans la manière dont l'Indien Pāṇini (probablement au IV^{e} siècle av. notre ère) décrit la grammaire du sanskrit.

langage de programmation hypothétique pourrait ressembler à ceci :

$$\begin{aligned}
\mathit{Statement} &\to \texttt{begin}\ \mathit{Block}\ \texttt{end}\\
\mathit{Statement} &\to \texttt{if}\ \mathit{Expression}\ \texttt{then}\ \mathit{Block}\ \texttt{fi}\\
\mathit{Statement} &\to \texttt{if}\ \mathit{Expression}\ \texttt{then}\ \mathit{Block}\ \texttt{else}\ \mathit{Block}\ \texttt{fi}\\
\mathit{Block} &\to \varepsilon\\
\mathit{Block} &\to \mathit{Statement}\ \mathit{Block}
\end{aligned}$$

Il faut comprendre ces règles de la manière suivante : « pour définir une instruction ($\mathit{Statement}$), on peut mettre un `begin` suivi d'un bloc (Block) suivi d'un `end`, ou bien un `if` suivi d'une expression ($\mathit{Expression}$) suivi d'un `then` suivi d'un bloc, éventuellement suivie encore d'un `else` et d'un nouveau bloc, et enfin un `fi` ; pour définir un bloc (Block), on peut soit ne rien mettre du tout, soit mettre une instruction suivi d'un nouveau bloc ».

Notre but va être d'expliquer quel genre de règles de ce genre on peut autoriser, comment elles se comportent, quels types de langages elles définissent, et comment on peut analyser (essentiellement, retrouver la manière dont on a construit) un texte produit par une application de telles règles.

4.0.3. Dans un contexte informatique, l'usage des grammaires formelles se fait à un niveau différent de celui des chaînes de caractères où nous nous sommes placés dans les sections précédentes : cette fois, les lettres de notre alphabet ne seront généralement pas des caractères mais des **tokens** du langage dont on définit la grammaire, un « token » pouvant être une unité variable selon le langage, mais généralement soit un caractère spécial, soit un mot-clé du langage (`begin`, `end`, `for`, etc., constituent donc généralement un unique token dans l'analyse, et ne sont pas divisés en leurs lettres individuelles), soit encore une unité (« nombre », « identificateur », éventuellement « commentaire ») qui aura été découpée dans une première phase d'analyse, appelée « analyse lexicale ».

Nous ne rentrerons pas ici dans ces considérations, et nous nous en tiendrons à l'approche mathématique où on continue à avoir affaire à des lettres d'un alphabet Σ abstrait.

4.1 Définition, notations et premier exemple

4.1.1. Une **grammaire hors contexte** (on dit parfois aussi « sans contexte » ; en anglais, « context-free grammar » ou « CFG » ; ou encore grammaire de **type 2**) sur un alphabet Σ est la donnée

— d'un second alphabet N, disjoint de Σ, appelé ensemble des **nonterminaux**,
— d'un élément $S \in N$ appelé **axiome** ou **symbole initial** de la grammaire,
— d'un ensemble *fini* $R \subseteq N \times (\Sigma \cup N)^*$ de couples (T, α) où T est un nonterminal et α un mot sur l'alphabet $\Sigma \cup N$, les éléments (T, α) de R étant appelés les **règles** ou **productions** de la grammaire.

4.1.2. Une grammaire hors contexte fait donc intervenir deux alphabets : l'alphabet Σ sur lequel sera le langage (encore à définir), et l'alphabet N (disjoint de Σ) qui intervient dans la définition de la grammaire. Pour fixer la terminologie, on appellera **symbole** un élément de $\Sigma \cup N$, ces symboles étant dits **terminaux** lorsqu'ils appartiennent à Σ (ou simplement « lettres »), et **nonterminaux** lorsqu'ils appartiennent à N. Un mot sur $\Sigma \cup N$ (c'est-à-dire, une suite finie de symboles) sera appelé **pseudo-mot** (ou parfois une « forme »), tandis que le terme « mot »

sans précision supplémentaire sera utilisé pour désigner un mot sur Σ (autrement dit, un mot est un pseudo-mot ne comportant que des symboles terminaux).

Pour redire les choses autrement, les symboles terminaux sont les lettres des mots du langage qu'on cherche à définir ; les symboles nonterminaux sont des symboles qui servent uniquement à titre transitoire dans l'utilisation de la grammaire, et qui sont destinés à disparaître finalement. Un « pseudo-mot » est un mot pouvant contenir des nonterminaux, tandis qu'un « mot » sans précision du contraire ne contient que des terminaux.

Typographiquement, on aura tendance à utiliser des lettres minuscules pour désigner les symboles terminaux, et majuscules pour désigner les symboles nonterminaux ; et on aura tendance à utiliser des lettres grecques minuscules pour désigner les pseudo-mots. Il n'est malheureusement pas possible d'être complètement systématique.

4.1.3. Intuitivement, il faut comprendre la grammaire de la manière suivante. Les règles (T, α), où on rappelle que T est un symbole nonterminal et α un pseudo-mot, doivent se comprendre comme « le symbole T peut être remplacé par α » (et on notera $T \to \alpha$, cf. ci-dessous). On part de l'axiome S, et on peut effectuer librement des substitutions $T \to \alpha$ (consistant à remplacer un nonterminal T par un pseudo-mot α) autorisées par les règles : le langage défini par la grammaire est l'ensemble de tous les mots (ne comportant plus aucun nonterminal) qu'on peut obtenir de la sorte.

Rendons maintenant cette définition précise :

4.1.4. Lorsque G est une grammaire hors contexte comme en 4.1.1, on note $T \to_G \alpha$, ou simplement $T \to \alpha$ (lorsque G est clair) pour signifier que (T, α) est une règle de G.

On définit une relation $\Rightarrow$ en posant $\gamma T \gamma' \Rightarrow \gamma\alpha\gamma'$ pour toute règle (T, α) et tous pseudo-mots γ, γ' : autrement dit, formellement, on définit $\lambda \Rightarrow \xi$ lorsqu'il existe $\gamma, \gamma' \in (\Sigma \cup N)^*$ et $(T, \alpha) \in R$ (ensemble des règles de G) tels que $\lambda = \gamma T \gamma'$ et $\xi = \gamma\alpha\gamma'$. Concrètement, $\lambda \Rightarrow \xi$ signifie donc que ξ est obtenu en remplaçant un nonterminal T du pseudo-mot λ par la partie droite d'une production $T \to \alpha$ de la grammaire G : on dit encore que $\lambda \Rightarrow \xi$ est une **dérivation immédiate** de G. On pourra dire que $T \to \alpha$ est la règle **appliquée** dans cette dérivation immédiate, que T est le **symbole réécrit** (souvent souligné dans l'écriture de la dérivation immédiate), et que γ et γ' sont respectivement le **contexte gauche** et le **contexte droit** de l'application de la règle [20]. Bien sûr, si nécessaire, on précisera la grammaire appliquée en écrivant $\lambda \Rightarrow_G \xi$ au lieu de simplement $\lambda \Rightarrow \xi$.

Une suite de pseudo-mots $(\lambda_0, \ldots, \lambda_n)$ telle que

$$\lambda_0 \Rightarrow \lambda_1 \Rightarrow \cdots \Rightarrow \lambda_n$$

autrement dit $\lambda_{i-1} \Rightarrow \lambda_i$ pour chaque $1 \leq i \leq n$, est appelée **dérivation** de λ_n à partir de λ_0 dans la grammaire G, et on note $\lambda_0 \Rightarrow^* \lambda_n$ pour indiquer son existence (c'est-à-dire, si on préfère, que la relation $\Rightarrow^*$ est la clôture réflexive-transitive de $\Rightarrow$, i.e., la plus petite relation binaire réflexive et transitive contenant $\Rightarrow$). Remarquons que $n = 0$ est permis, autrement dit $\lambda \Rightarrow^* \lambda$ pour tout pseudo-mot λ. Bien sûr, si nécessaire, on précisera la grammaire appliquée en écrivant $\lambda \Rightarrow^*_G \xi$ au lieu de simplement $\lambda \Rightarrow^* \xi$.

20. Pour être extrêmement rigoureux, une dérivation immédiate doit comporter la donnée du symbole réécrit (ou de façon équivalente, du contexte gauche et/ou droit), car elle ne peut pas se déduire de la seule donnée de λ et ξ (par exemple, dans $XX \Rightarrow XXX$, même si on sait que la règle appliquée était $X \to XX$, on ne peut pas deviner si c'est le X de gauche ou de droite qui a été réécrit : il faut donc écrire $\underline{X}X \Rightarrow XXX$ ou bien $X\underline{X} \Rightarrow XXX$, en soulignant le symbole réécrit, pour distinguer les deux). De même, dans une dérivation, on devrait inclure la donnée du symbole réécrit à chaque étape.

Concrètement, $\lambda \Rightarrow^* \xi$ signifie donc qu'on peut passer de λ à ξ en effectuant une suite (finie !) de dérivations immédiates, c'est-à-dire en remplaçant à chaque étape un nonterminal T par la partie droite α d'une règle $T \to \alpha$ de la grammaire G.

Il va de soi qu'un pseudo-mot qui ne comporte que des terminaux, i.e., qui est en fait un mot (sur Σ), ne peut pas être dérivé plus loin. Ceci justifie au moins en partie la terminologie de « terminal ».

4.1.5. Le **langage engendré** $L(G)$ par une grammaire hors contexte G est l'ensemble des mots w (ne comportant plus de nonterminaux !) qui peuvent s'obtenir par dérivation à partir de l'axiome S de G, autrement dit :

$$L(G) = \{w \in \Sigma^* : S \Rightarrow^* w\}$$

Un langage qui peut s'écrire sous la forme $L(G)$ où G est une grammaire hors contexte est appelé **langage hors contexte** ou **algébrique**.

Deux grammaires hors contexte G et G' sont dites **faiblement équivalentes** ou **langage-équivalentes** lorsqu'elles engendrent le même langage ($L(G) = L(G')$).

4.1.6. À titre d'exemple, considérons la grammaire sur l'alphabet $\Sigma = \{a, b\}$ donnée par

$$S \to aSb$$
$$S \to \varepsilon$$

où implicitement S est l'axiome et le seul nonterminal : autrement dit, G est donnée par $N = \{S\}$, d'axiome S, et de règles de production (S, aSb) et (S, ε) (où ε, bien entendu, désigne le mot vide). Un pseudo-mot pour cette grammaire est un mot sur l'alphabet $\Sigma \cup N = \{a, b, S\}$, et une dérivation immédiate consiste *soit* à ajouter un a et un b à gauche et à droite d'un S dans un pseudo-mot, *soit* à retirer un S.

Dans ces conditions, il est clair qu'une dérivation partant de l'axiome S est constituée d'un certain nombre d'applications de la première règle suivi d'au plus une application de la seconde (puisque le nombre d'occurrences de S va alors tomber de 1 à 0 et on ne pourra plus dériver). Autrement dit, une telle dérivation prend la forme

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow \cdots \Rightarrow a^nSb^n$$

ou éventuellement

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow \cdots \Rightarrow a^nSb^n \Rightarrow a^nb^n$$

Le langage $L(G)$ défini par la grammaire est donc $\{a^nb^n : n \in \mathbb{N}\}$.

On a vu en 3.5.5 que ce langage n'est pas rationnel : il existe donc des langages algébriques qui ne sont pas rationnels. En revanche, pour ce qui est de la réciproque, on verra dans la section suivante que tout langage rationnel est algébrique.

4.1.7. Mentionnons brièvement qu'il existe des types de grammaires plus généraux que les grammaires hors contexte. Les **grammaires contextuelles** (ou grammaires de **type 1**) sont définies par des règles du type $\gamma T\gamma' \to \gamma\alpha\gamma'$ où T est un nonterminal, et α, γ, γ' des pseudo-mots (= mots sur l'alphabet de tous les symboles), c'est-à-dire des règles qui autorisent la réécriture d'un symbole T en α mais uniquement s'il est entouré d'un certain contexte (γ à gauche, γ' à droite). Les **grammaires syntagmatiques générales** (ou grammaires de **type 0**)

sont définies par des règles de réécriture $\lambda \to \mu$ où λ, μ sont des pseudo-mots quelconques, c'est-à-dire qu'elles permettent la réécriture de multiples symboles à la fois. Dans tous les cas, le langage défini par la grammaire est l'ensemble de tous les mots (sans nonterminaux) qui peuvent s'obtenir par application des règles de remplacement à partir de l'axiome.

Les grammaires de types 0 et 1, avec celles de type 2 c'est-à-dire hors contexte, et celles de type 3 (= régulières) qui seront définies en 4.2.2 ci-dessous, forment une hiérarchie appelée **hiérarchie de Chomsky** : plus le numéro du type est élevé plus la grammaire est contrainte et plus la classe de langages définie est petite.

4.2 Langages rationnels et algébriques, stabilité

Proposition 4.2.1. Tout langage rationnel est algébrique. Mieux, on peut déduire algorithmiquement une grammaire hors contexte G d'un εNFA A de façon à avoir $L(G) = L(A)$.

Démonstration. Soit A un εNFA : on sait qu'on peut supposer sans perte de généralité qu'il a un unique état initial q_0 (quitte à en créer un et à ajouter pour tout état q anciennement initial une ε-transition $q_0 \to q$). Soit Q son ensemble des états et $\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$ sa fonction de transition. On construit une grammaire hors contexte G dont l'ensemble des nonterminaux est Q, dont l'axiome est $S := q_0$, et dont les règles sont les suivantes : (A) pour chaque transition $(q, t, q') \in \delta$ une règle $q \to tq'$ (on rappelle que $t \in \Sigma \cup \{\varepsilon\}$), et (B) pour chaque état final $q \in F$ une règle $q \to \varepsilon$.

De même que dans l'exemple 4.1.6, une dérivation partant de l'axiome va comporter un certain nombre d'applications de règles de type (A) suivies éventuellement d'une unique application d'une règle de type (B) (ce qui est nécessaire pour faire passer le nombre de nonterminaux de 1 à 0). Autrement dit, elle prend la forme $q_0 \Rightarrow t_1 q_1 \Rightarrow t_1 t_2 q_2 \Rightarrow \cdots \Rightarrow t_1 t_2 \cdots t_n q_n$ où $(q_{i-1}, t_i, q_i) \in \delta$ pour $1 \leq i \leq n$, suivie éventuellement par $t_1 \cdots t_n q_n \Rightarrow t_1 \cdots t_n$ lorsque $q_n \in F$. Manifestement, les dérivations de la sorte (terminant sur un mot sur Σ) sont en bijection avec les chemins $q_0 \to \cdots \to q_n$ dans A où $q_n \in F$, et le mot $t_1 \cdots t_n$ dérivé est précisément le mot formé en concaténant les étiquettes du chemin. On a donc bien $L(G) = L(A)$. ☺

4.2.2. Une grammaire hors contexte telle que construite dans la preuve de 4.2.1 est dite **régulière** : autrement dit, il s'agit d'une grammaire ayant uniquement des règles de deux formes : (A) $Q \to tQ'$ où $t \in \Sigma \cup \{\varepsilon\}$, et (B) $Q \to \varepsilon$. Les langages définis par des grammaires régulières sont donc exactement les langages rationnels.

La définition des grammaires régulières varie un peu d'auteur en auteur, ces différences n'étant pas très importantes. Certains auteurs imposent $t \in \Sigma$ dans les règles de type (A), ce qui revient à imposer qu'on part d'un NFA sans ε-transition dans la construction ; et certains autorisent (ou autorisent uniquement) dans le type (B) les règles de la forme $Q \to x$ avec $x \in \Sigma$, ce qui impose des petits changements dans la construction ci-dessus. Dans tous les cas, il reste vrai que les langages définis par des grammaires régulières sont exactement les langages rationnels.

Les grammaires régulières sont également appelées grammaires de **type 3**.

Proposition 4.2.3. Si L_1, L_2 sont des langages algébriques (sur un même alphabet Σ), alors la réunion $L_1 \cup L_2$ est algébrique ; de plus, une grammaire hors contexte l'engendrant se déduit algorithmiquement de grammaires hors contexte engendrant L_1 et L_2.

Si L_1, L_2 sont des langages algébriques (sur un même alphabet Σ), alors la concaténation $L_1 L_2$ est algébrique ; de plus, une grammaire hors contexte l'engendrant se déduit algorithmiquement de grammaires hors contexte engendrant L_1 et L_2.

Si L est un langage algébrique, alors l'étoile de Kleene L^* est algébrique ; de plus, une grammaire hors contexte l'engendrant se déduit algorithmiquement d'une grammaire hors contexte engendrant L.

Démonstration. Pour la réunion : supposons que G_1 et G_2 engendrent L_1 et L_2 respectivement, et ont des ensembles de nonterminaux N_1 et N_2 disjoints, avec axiomes S_1 et S_2 respectivement. On construit une grammaire G dont l'ensemble des nonterminaux est $N_1 \cup N_2 \cup \{S\}$ où S est un nouveau nonterminal, choisi comme axiome, et les règles de production de G sont celles de G_1, celles de G_2, et les deux règles supplémentaires $S \to S_1$ et $S \to S_2$. Il est évident qu'une dérivation de G partant de S est soit de la forme $S \Rightarrow S_1$ suivie d'une dérivation de G_1, soit de la forme $S \Rightarrow S_2$ suivie d'une dérivation de G_2 : ainsi, $L(G) = L_1 \cup L_2$.

Pour la concaténation : la construction est presque exactement la même, mais on prend pour règle supplémentaire $S \to S_1 S_2$.

Pour l'étoile : si G engendre L et a pour ensemble de nonterminaux N et pour axiome S, on défit une nouvelle grammaire G' dont l'ensemble des nonterminaux est $N' = N \cup \{S'\}$ où S' est un nouveau nonterminal, choisi comme axiome, et les règles de production de G' sont celles de G et les deux règles supplémentaires $S' \to SS'$ et $S' \to \varepsilon$. Il est facile de se convaincre qu'une dérivation de G' partant de S' et utilisant n fois la règle $S' \to SS'$ se ramène à n dérivations de G partant de S, et seule la règle $S' \to \varepsilon$ permet de faire disparaître le symbole S'. (Les détails sont omis et peuvent être rendus plus clairs en utilisant la notion d'arbre de dérivation.) ☺

4.2.4. Ceci fournit une nouvelle démonstration (partant d'une expression rationnelle plutôt que d'un automate) de 4.2.1, tant il est évident que les langages $\varnothing$, $\{\varepsilon\}$ et $\{x\}$ (pour $x \in \Sigma$) sont algébriques.

À cause de cette construction, on se permettra parfois, dans l'énoncé des règles d'une grammaire, d'écrire $T \to \alpha_1 | \cdots | \alpha_n$ pour réunir en une seule ligne plusieurs règles $T \to \alpha_1$ jusqu'à $T \to \alpha_n$. Par exemple, la grammaire présentée en 4.1.6 peut s'écrire $S \to aSb|\varepsilon$. Il s'agit d'un simple raccourci d'écriture (comparer avec une convention semblable faite pour les automates en 2.1.2).

On pourrait même se permettre l'abus de notation consistant à écrire $T \to U^*$ pour $T \to UT|\varepsilon$ (c'est-à-dire $T \to UT$ et $T \to \varepsilon$), mais il vaut mieux éviter car cela pourrait aussi bien signifier $T \to TU|\varepsilon$, qui engendre le même langage (i.e., elle est faiblement équivalente à la précédente) mais dont l'analyse peut être plus problématique. De façon encore plus générale, on pourrait imaginer d'autoriser des règles de la forme $T \to r$ où T est un nonterminal et r une expression rationnelle quelconque sur l'alphabet $\Sigma \cup N$: de telles règles pourraient se ramener aux règles qu'on a autorisées, quitte à introduire de nouveaux nonterminaux pour les expressions intermédiaires (par exemple, $T \to UV^*$ se transformerait par l'introduction d'un nouveau nonterminal V' en $T \to UV'$ et $V' \to VV'|\varepsilon$). Nous éviterons ces extensions aux grammaires hors contexte pour ne pas introduire de confusion.

4.2.5. Il peut être utile, pour raisonner sur les langages algébriques, d'introduire, lorsque G est une grammaire hors contexte et T un nonterminal quelconque, le langage $L(G,T) := \{w \in \Sigma^* : T \Rightarrow_G^* w\}$ des mots qui dérivent de T, c'est-à-dire le langage engendré par la grammaire G' identique à G à ceci près que son axiome est T.

On a alors $L(G) = L(G,S)$ où S est l'axiome de G, et chaque règle de production $T \to \alpha_1 | \cdots | \alpha_n$ se traduit par une équation portant sur $L(G,T)$, par exemple $T \to aUbV|cW$ implique $L(G,T) = a\,L(G,U)\,b\,L(G,V) \cup cL(G,W)$. De cette manière, une grammaire hors contexte donne lieu à un système d'équations portant sur des langages (par exemple, la grammaire de 4.1.6 donne lieu à l'équation $L(G) = (a\,L(G)\,b) \cup \{\varepsilon\}$: on peut montrer que la définition des grammaires hors contexte revient à considérer la plus petite (au sens de l'inclusion) solution de ce système d'équations.

4.2.6. À la différence des langages rationnels, il *n'est pas vrai* en général que l'intersection de deux langages algébriques soit algébrique : un contre-exemple est fourni par les deux langages $\{a^i b^i c^j \,:\, i,j \in \mathbb{N}\}$ et $\{a^i b^j c^j \,:\, i,j \in \mathbb{N}\}$ (chacun des deux est algébrique : par exemple, le premier est la concaténation du langage $\{a^i b^i \,:\, i \in \mathbb{N}\}$ dont on a vu en 4.1.6 qu'il était algébrique, et du langage $\{c\}^*$, qui est algébrique car rationnel) ; leur intersection $\{a^i b^i c^i : i \in \mathbb{N}\}$ n'est pas algébrique, comme on le démontrera en 4.6.2.

En conséquence, il n'est pas non plus vrai en général que le complémentaire d'un langage algébrique soit algébrique.

En revanche, le fait suivant, que nous admettons sans démonstration, peut s'avérer utile :

Proposition 4.2.7. L'intersection d'un langage *algébrique* et d'un langage *rationnel* est algébrique.

4.3 Autres exemples de grammaires hors contexte

4.3.1. Sur l'alphabet $\Sigma = \{a, b\}$, considérons la grammaire (d'axiome S)

$$\begin{aligned} S &\to AT \\ A &\to aA \mid \varepsilon \\ T &\to aTb \mid \varepsilon \end{aligned}$$

Il est clair que le langage $L(G, A)$ des mots qui dérivent de A est simplement $\{a\}^* = \{a^i \,:\, i \in \mathbb{N}\}$; et le langage $L(G, T)$ est $\{a^j b^j \,:\, j \in \mathbb{N}\}$ comme en 4.1.6. Le langage $L(G) = L(G, A)\, L(G, T)$ est donc $\{a^i b^j \,:\, i \geq j\}$, l'ensemble des mots de la forme $a^i b^j$ pour lesquels $i \geq j$.

On peut aussi considérer la grammaire G' définie par

$$\begin{aligned} S &\to T \mid aS \\ T &\to aTb \mid \varepsilon \end{aligned}$$

Il n'est pas difficile de se convaincre qu'elle engendre le même langage $L(G)$ que la précédente (elle est donc faiblement équivalente à G).

(Ce langage est intéressant sur le plan théorique, car bien qu'il soit algébrique, et ici défini par une grammaire G' inambiguë (cf. 4.5.1), il ne peut pas être analysé par un analyseur LL (cf. 4.7.6).)

4.3.2. Sur l'alphabet $\Sigma = \{a, b\}$, considérons la grammaire (d'axiome S)

$$\begin{aligned} S &\to U \mid V \\ U &\to aUb \mid ab \\ V &\to aVbb \mid abb \end{aligned}$$

Il est clair que le langage $L(G, U)$ des mots qui dérivent de U est $\{a^i b^i : i \geq 1\}$ comme en 4.1.6, et de façon analogue, le langage $L(G, V)$ des mots qui dérivent de V est $\{a^i b^{2i} \,:\, i \geq 1\}$. Le langage $L(G) = L(G, U) \cup L(G, V)$ est donc l'ensemble des mots de la forme $a^i b^j$ où $i \geq 1$ et $j = i$ *ou bien* $j = 2i$.

(Ce langage est intéressant sur le plan théorique, car bien qu'il soit algébrique et défini par une grammaire inambiguë (cf. 4.5.1), il n'est pas analysable par un automate à pile déterministe (cf. 4.7.2).)

4.3.3. Sur l'alphabet $\Sigma = \{a, b, c\}$, considérons la grammaire (d'axiome S)

$$\begin{aligned} S &\to UC \mid AV \\ A &\to aA \mid \varepsilon \\ C &\to cC \mid \varepsilon \\ U &\to aUb \mid \varepsilon \\ V &\to bVc \mid \varepsilon \end{aligned}$$

Il est clair que le langage $L(G, A)$ des mots qui dérivent de A est simplement $\{a\}^* = \{a^i : i \in \mathbb{N}\}$ et que le langage $L(G, C)$ des mots qui dérivent de C est simplement $\{c\}^* = \{c^j : j \in \mathbb{N}\}$; le langage $L(G, U)$ est $\{a^i b^i : i \in \mathbb{N}\}$ comme en 4.1.6, et de même $L(G, V) = \{b^j c^j : j \in \mathbb{N}\}$. Finalement, $L(G) = L(G, U)\, L(G, C) \ \cup \ L(G, A)\, L(G, V)$ montre que $L(G) = \{a^i b^i c^j : i, j \in \mathbb{N}\} \cup \{a^i b^j c^j : i, j \in \mathbb{N}\}$ est le langage des mots de la forme $a^i b^r c^j$ où $r = i$ ***ou*** $r = j$.

(Ce langage est intéressant sur le plan théorique, car il est algébrique mais « intrinsèquement ambigu » (cf. 4.5.4).)

4.3.4. Sur l'alphabet $\Sigma = \{a, b\}$, considérons la grammaire (d'axiome S)

$$S \ \to \ aSbS \mid \varepsilon$$

ou, ce qui revient au même

$$\begin{aligned} S &\to TS \mid \varepsilon \\ T &\to aSb \end{aligned}$$

(la première ligne revient par ailleurs à $S \to T^*$). On a alors $L(G) = L(G, T)^*$ où $L(G, T) = a\, L(G)\, b$ désigne le langage des mots qui peuvent être dérivés à partir de T.

Ce langage $L(G)$ s'appelle langage des **expressions bien-parenthésées** où a représente une parenthèse ouvrante et b une parenthèse fermante (et $L(G, T)$ est le langage des blocs parenthésés : autrement dit, une expression bien-parenthésée est une suite de blocs parenthésés, et un bloc parenthésé est une expression bien-parenthésée entourée par une parenthèse ouvrante et une parenthèse fermante — c'est ce que traduit la grammaire).

Soit dit en passant, ce langage algébrique des expressions bien-parenthésées n'est pas rationnel : la façon la plus simple de s'en convaincre est de remarquer que s'il l'était, son intersection avec le langage $\{a^i b^j : i, j \in \mathbb{N}\}$ dénoté par l'expression rationnelle $a*b*$ le serait aussi (par 3.1.4), or cette intersection est précisément le langage $\{a^n b^n : n \in \mathbb{N}\}$ dont on a montré en 3.5.5 qu'il n'était pas algébrique.

4.3.5. Sur l'alphabet $\Sigma = \{a, b\}$, montrons que la grammaire G (d'axiome S) donnée par

$$S \ \to \ aSbS \mid bSaS \mid \varepsilon$$

(c'est-à-dire $N = \{S\}$ où S est l'axiome, et pour règles $S \to aSbS$ et $S \to bSaS$ et $S \to \varepsilon$) engendre le langage L formé des mots ayant un nombre total de a égal au nombre total de b, i.e., $L := \{w \in \Sigma^* : |w|_a = |w|_b\}$ où $|w|_x$ désigne le nombre total d'occurrences de la lettre x dans le mot w (cf. 1.2.11).

Démonstration. Dans un sens il est évident que tout pseudo-mot (et en particulier, tout mot sur Σ) obtenu en dérivant S a un nombre de a égal au nombre de b, puisque cette propriété est conservée par toute dérivation immédiate. Autrement dit, $L(G) \subseteq L$.

Pour ce qui est de la réciproque, on va montrer par récurrence sur $|w|$ que tout mot $w \in \Sigma^*$ ayant autant de a que de b est dans $L(G)$. Pour $|w| = 0$ c'est trivial. Sinon, on peut supposer sans perte de généralité que le mot w considéré commence par a, soit $w = aw'$ où w' vérifie $|w'|_b - |w'|_a = 1$. Considérons maintenant la fonction h qui à un entier k entre 0 et $|w'|$ associe le nombre de b moins le nombre de a dans le préfixe de longueur k de w' (i.e., parmi les k premières lettres de w') : cette fonction prend les valeurs $h(0) = 0$ et $h(|w'|) = 1$, et elle varie de ± 1 à chaque fois (i.e., $h(i) = h(i-1) \pm 1$ selon que la i-ième lettre de w' est b ou a). Soit k_0 le plus grand entier tel que $h(k_0) = 0$: cet entier est bien défini car $h(0) = 0$. On a forcément $h(k_0 + 1) = 1$ car l'autre possibilité, $h(k_0 + 1) = -1$ impliquerait que h devrait repasser par la valeur 0 pour atteindre la valeur finale $h(|w'|) = 1$, ce qui contredirait la maximalité de k_0. Bref, si u est le préfixe de w' de longueur k_0, on a $|u|_b - |u|_a = h(k_0) = 0$ et la lettre suivante de w' est un b, si bien que $w' = ubv$ et finalement $w = aubv$ où u et donc v ont autant que a que de b, i.e., appartiennent à L, et par récurrence (ils sont de longueur strictement plus courte que w) appartiennent à $L(G)$. On peut donc dériver u et v de S, donc w de $aSbS$, et comme $aSbS$ se dérive immédiatement de S, on a bien $w \in L(G)$, ce qui conclut la récurrence. ☺

Un raisonnement analogue montre que la grammaire G' donnée par

$$\begin{aligned} S &\to aUbS \mid bVaS \mid \varepsilon \\ U &\to aUbU \mid \varepsilon \\ V &\to bVaV \mid \varepsilon \end{aligned}$$

engendre le même langage $L = \{w \in \Sigma^* : |w|_a = |w|_b\}$ que ci-dessus (elle est donc faiblement équivalente à G) : l'idée est que le langage $L(G, U)$ des mots qui dérivent de U est l'ensemble des expressions bien-parenthésées où a est la parenthèse ouvrante et b la fermante (avec les notations de la démonstration ci-dessus, ce sont les mots pour lesquels la fonction h, partant de 0, revient à 0, et reste toujours positive ou nulle), tandis que $L(G, V)$ est l'ensemble des expressions bien-parenthésées où b est la parenthèse ouvrante et a la fermante (la fonction h reste toujours négative ou nulle), et tout mot de L peut s'écrire comme concaténation de tels mots (en divisant selon le signe de h). Une différence entre G et G' est que la grammaire G est ambiguë (terme qui sera défini en 4.5.1) tandis que G' ne l'est pas.

4.4 Arbres d'analyse, dérivations gauche et droite

4.4.1. Soit G une grammaire hors contexte sur un alphabet Σ et N l'ensemble de ses nonterminaux. Un **arbre d'analyse** (ou **arbre de dérivation** ; en anglais **parse tree**) **incomplet** pour G est un arbre (fini, enraciné et ordonné [21]) dont les nœuds sont étiquetés par des éléments de $\Sigma \cup N \cup \{\varepsilon\}$, vérifiant les propriétés suivantes :

— la racine de l'arbre est étiquetée par l'axiome S de G ;

— si un nœud de l'arbre n'est pas une feuille (i.e., s'il a des fils) et si on appelle T son étiquette, alors

21. C'est-à-dire que l'ensemble des fils d'un nœud quelconque de l'arbre est muni d'un ordre total, ou, ce qui revient au même, qu'ils sont numérotés (disons de la gauche vers la droite).

— soit ce nœud a un unique fils étiqueté ε et il existe une règle $T \to \varepsilon$ dans G,
— soit ce nœud a des fils étiquetés par des éléments $x_1 \cdots x_n$ (où $n \geq 1$) de $\Sigma \cup N$ et il existe une règle $T \to x_1 \cdots x_n$ dans G,

(dans ces deux sous-cas, il résulte de l'existence d'une règle $T \to \cdots$ que T est un nonterminal : seules les feuilles peuvent être étiquetées par un terminal ou ε).

On dit de plus que l'arbre est **complet**, ou simplement qu'il s'agit d'un arbre d'analyse, lorsqu'il vérifie la propriété supplémentaire suivante :

— toute feuille de l'arbre est étiquetée soit par un terminal (= élément de Σ) soit par ε.

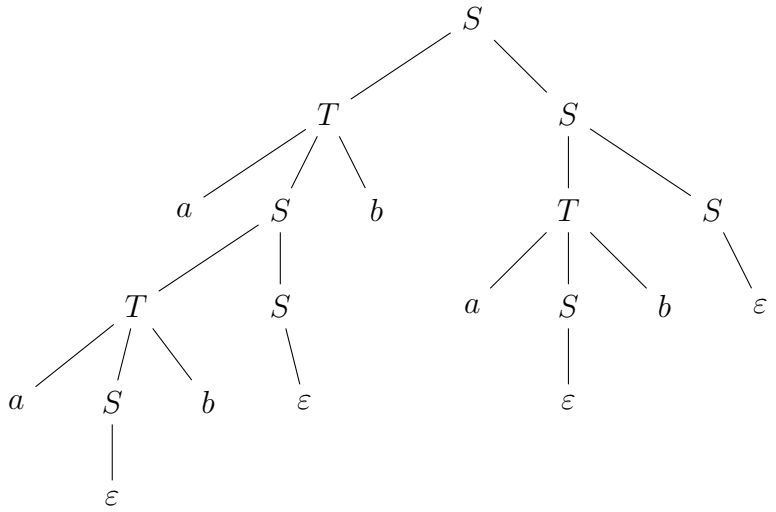
(Les feuilles étiquetées ε servent uniquement à marquer la complétude de l'arbre : certains auteurs procèdent différemment ou les omettent.)

4.4.2. Si T est un arbre d'analyse incomplet, et α le pseudo-mot obtenu en lisant les étiquettes des *feuilles* de T en profondeur ordonnée (c'est-à-dire en commençant par toutes les feuilles qui descendent du premier fils de la racine elles-mêmes dans le même ordre, puis toutes celles qui descendent du second fils, et ainsi de suite) et en ignorant tous les ε, alors on dit que T est un arbre d'analyse *de* α, ou que α est le pseudo-mot analysé par T. La définition d'un arbre d'analyse complet signifie exactement qu'il est l'arbre d'analyse d'un *mot* (c'est-à-dire, d'un mot sur Σ).

4.4.3. À titre d'exemple, considérons la grammaire (considérée en 4.3.4)

$$S \to TS \mid \varepsilon$$
$$T \to aSb$$

L'arbre d'analyse suivant est un arbre d'analyse du mot $aabbab$ (et on verra en 4.5.3 que c'est le seul possible) :



On obtient des exemples d'arbres d'analyse incomplet en effaçant arbitrairement tous les descendants d'un ou plusieurs nœuds de cet arbre.

4.4.4. À toute dérivation à partir de l'axiome S dans une grammaire hors contexte G on peut associer un arbre d'analyse incomplet : ce dernier est construit en partant de la racine et, à chaque fois qu'un symbole est réécrit (dans la dérivation), en faisant pousser des fils à la feuille correspondante de l'arbre (qui cesse donc d'être une feuille) pour indiquer la règle appliquée.

De façon plus formelle : si $S =: \lambda_0 \Rightarrow \lambda_1 \Rightarrow \cdots \Rightarrow \lambda_n$ est une dérivation à partir de l'axiome S dans une grammaire hors contexte G, on va lui associer, par récurrence sur le nombre d'étapes n de la dérivation, un arbre d'analyse incomplet du pseudo-mot λ_n. Si $n = 0$, l'arbre $\mathscr{T}_0$ en question est simplement l'arbre trivial (ayant pour seul nœud la racine S). Sinon, on part de l'arbre $\mathscr{T}_{n-1}$ (qu'on a défini par récurrence) pour la dérivation $\lambda_0 \Rightarrow \cdots \Rightarrow \lambda_{n-1}$: supposons que la dernière dérivation immédiate $\lambda_{n-1} \Rightarrow \lambda_n$ réécrive le symbole nonterminal T par application de la règle $T \to \alpha$ avec pour contextes gauche et droit γ, γ' (cf. 4.1.4), si bien que $\lambda_{n-1} = \gamma T \gamma'$ et $\lambda_n = \gamma\alpha\gamma'$; le symbole T réécrit correspond à un certain nœud de l'arbre $\mathscr{T}_{n-1}$, à savoir la feuille, étiquetée T, telle qu'on lise γ à sa gauche et γ' à sa droite en suivant les feuilles dans l'ordre de profondeur ; on ajoute alors au nœud en question des fils correspondant à la règle $T \to \alpha$, c'est-à-dire soit un unique fils étiqueté ε si $\alpha = \varepsilon$, soit k fils étiquetés $x_1, \ldots, x_k$ si $\alpha = x_1 \cdots x_k$ avec $k \geq 1$.

Notons que l'arbre obtenu est complet exactement lorsque la dérivation aboutit à un mot (sur Σ). Notons aussi que le nombre n d'étapes dans la dérivation est égal au nombre de nœuds de l'arbre qui *ne sont pas* des feuilles.

Deux dérivations (forcément du même mot ou pseudo-mot) auxquelles sont associées le même arbre d'analyse sont dites **équivalentes**.

4.4.5. À titre d'exemple, l'arbre illustré en 4.4.3 est associé à la dérivation

$$\begin{aligned}\underline{S} \Rightarrow \underline{T}S \Rightarrow a\underline{S}bS \Rightarrow a\underline{T}SbS \Rightarrow aa\underline{S}bSbS \Rightarrow aab\underline{S}bS \Rightarrow aabb\underline{S} \\ \Rightarrow aabb\underline{T}S \Rightarrow aabba\underline{S}bS \Rightarrow aabbab\underline{S} \Rightarrow aabbab\end{aligned}$$

(on a souligné à chaque fois le symbole réécrit), mais aussi à la dérivation

$$\begin{aligned}\underline{S} \Rightarrow T\underline{S} \Rightarrow TT\underline{S} \Rightarrow T\underline{T} \Rightarrow Ta\underline{S}b \Rightarrow \underline{T}ab \\ \Rightarrow a\underline{S}bab \Rightarrow aT\underline{S}bab \Rightarrow a\underline{T}bab \Rightarrow aa\underline{S}bbab \Rightarrow aabbab\end{aligned}$$

Ces deux dérivations sont donc équivalentes. Elles ont toutes les deux 10 étapes puisque l'arbre considéré a 10 nœuds qui ne sont pas des feuilles.

4.4.6. On appelle **dérivation gauche** une dérivation (pour une grammaire hors contexte donnée) dans laquelle le symbole réécrit est toujours *le nonterminal le plus à gauche* du pseudo-mot courant : autrement dit, une dérivation gauche est une dérivation telle que le contexte gauche de chaque dérivation immédiate la constituant ne comporte que des symboles terminaux (i.e., appartient à Σ^*). Symétriquement, on appelle **dérivation droite** une dérivation dans laquelle le symbole réécrit est toujours *le nonterminal le plus à droite*, c'est-à-dire que le contexte droit de chaque dérivation immédiate la constituant ne comporte que des symboles terminaux.

À titre d'exemple, les deux dérivations données en 4.4.5 sont respectivement une dérivation gauche et une dérivation droite.

À chaque arbre d'analyse est associée une et une seule dérivation gauche : on l'obtient de façon évidente en réécrivant à chaque étape le symbole nonterminal le plus à gauche en suivant la règle indiquée par l'arbre d'analyse sous le nœud correspondant ; cela revient à parcourir l'arbre d'analyse en profondeur de gauche à droite, et à réécrire le symbole correspondant à chaque nœud que l'on rencontre qui n'est pas une feuille. De même, à chaque arbre d'analyse est associée une et une seule dérivation droite.

4.5 Ambiguïté

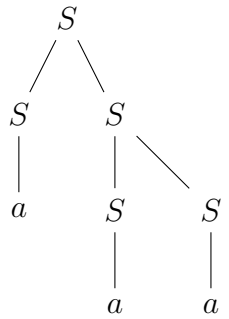
4.5.1. On dit qu'une grammaire hors contexte est **ambiguë** lorsqu'il existe un mot (i.e., un mot sur l'alphabet Σ des terminaux) qui admet deux arbres d'analyse *différents*. Dans le cas contraire, elle est dite **inambiguë** : autrement dit, une grammaire inambiguë est une grammaire G pour laquelle tout $w \in L(G)$ a un unique arbre d'analyse ; il revient au même de dire que tout mot de $L(G)$ a une unique dérivation droite, ou encore que tout mot de $L(G)$ a une unique dérivation gauche.

Les grammaires des langages informatiques réels sont évidemment (presque ?) toujours inambiguës : on souhaite qu'un programme (c'est-à-dire, dans la terminologie mathématique, un mot du langage) admette une unique interprétation, autrement dit, un unique arbre d'analyse.

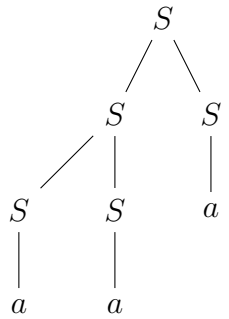
4.5.2. À titre d'exemple, la grammaire

$$S \ \to \ SS \mid a$$

sur l'alphabet $\{a\}$ est ambiguë. En effet, le mot aaa admet l'arbre d'analyse



mais aussi



Cette grammaire est donc ambiguë. Remarquons que le langage qu'elle engendre est $\{a\}^+ = \{a^n : n \geq 1\}$ (car il est évident que tout mot engendré par la grammaire est dans $\{a\}^+$, et réciproquement il est facile de fabriquer une dérivation de a^n pour tout $n \geq 1$, par exemple $\underline{S} \Rightarrow S\underline{S} \Rightarrow \cdots \Rightarrow S^{n-1}S \Rightarrow \cdots a^n$).

Le *même* langage $\{a\}^+ = \{a^n : n \geq 1\}$ peut aussi être engendré par la grammaire

$$S \ \to \ aS \mid a$$

(faiblement équivalente à la précédente, donc) qui, elle, *n'est pas* ambiguë : en effet, la seule manière de dériver a^n consiste à appliquer $n-1$ fois la règle $S \to aS$ et finalement une fois la règle $S \to a$ (il y a donc une unique dérivation du mot, et *a fortiori* un unique arbre d'analyse).

4.5.3. La grammaire G des expressions bien-parenthésées

$$S \to TS \mid \varepsilon$$
$$T \to aSb$$

considérée en 4.3.4 est inambiguë. Le point crucial pour s'en convaincre est que dans l'application de la règle $S \to TS$ dans l'analyse d'un mot $w \neq \varepsilon$ engendré par la grammaire, il n'y a qu'une seule possibilité sur la limite du préfixe u qui dérivera de T (et donc du suffixe qui dérivera de S) : en s'inspirant de 4.3.5 on peut se convaincre que u est le préfixe de w de la plus petite longueur > 0 possible comportant autant de b que de a (par exemple si $w = aabbab$ alors $u = aabb$) ; ce point étant acquis, tout mot $w \neq \varepsilon$ de la grammaire $L(G) = L(G,S)$ s'analyse de façon unique comme concaténation d'un mot $u \in L(G,T)$ (c'est-à-dire dérivant de T) et d'un mot $v \in L(G,S)$, et chacun de ces morceaux s'analyse à son tour de façon unique (la règle $T \to aSb$ ne permet manifestement qu'une seule analyse d'un mot de $L(G,T)$: une fois qu'on enlève le a initial et le b final, il reste un mot de $L(G,S)$).

Notamment, l'arbre représenté en 4.4.3 est l'*unique* arbre d'analyse de $aabbab$ pour la grammaire présentée ci-dessus.

4.5.4. Il arrive que le *même* langage puisse être engendré par une grammaire ambiguë et par une grammaire inambiguë (on a vu un exemple en 4.5.2). L'ambiguïté est donc une caractéristique de la *grammaire* hors contexte et non du *langage* algébrique qu'elle engendre.

Cependant, certains langages algébriques ne sont définis *que* par des grammaires hors contexte ambiguës. De tels langages sont dits **intrinsèquement ambigus**. C'est le cas du langage $\{a^i b^i c^j : i,j \in \mathbb{N}\} \cup \{a^i b^j c^j : i,j \in \mathbb{N}\}$ dont on a vu en 4.3.3 qu'il était algébrique : il n'est pas évident (cela dépasse le cadre de ce cours) de démontrer qu'il est intrinsèquement ambigu, mais on peut au moins en donner une explication intuitive : quelle que soit la manière dont on construit une grammaire engendrant ce langage, elle devra forcément distinguer le cas de $a^i b^i c^j$ et celui de $a^i b^j c^j$, or ces cas ne sont pas disjoints, il existe des mots $a^i b^i c^i$ qui sont à l'intersection des deux, et ce sont ces mots qui forcent ce langage à être intrinsèquement ambigu.

4.6 Le lemme de pompage pour les langages algébriques

On a vu en §3.5 une condition nécessaire que doivent vérifier les langages rationnels, et qui est souvent utile pour montrer qu'un langage *n'est pas* rationnel. Un lemme tout à fait analogue existe pour les langages algébriques, est qui s'avère utile dans des circonstances semblables, même si son emploi est plus difficile ; on prendra garde au fait que, dans l'énoncé suivant, x et y désignent des mots et non des lettres comme d'habitude :

Proposition 4.6.1 (lemme de pompage pour les langages algébriques)**.** Soit L un langage algébrique. Il existe alors un entier k tel que tout mot de $t \in L$ de longueur $|t| \geq k$ admette une factorisation $t = uvwxy$ en cinq facteurs $u, v, w, x, y \in \Sigma^*$ où :

(i) $|vx| \geq 1$ (c'est-à-dire que l'un au moins de v et x est $\neq \varepsilon$),

(ii) $|vwx| \leq k$,

(iii) pour tout $i \geq 0$ on a $uv^iwx^iy \in L$.

Donnons maintenant un exemple d'utilisation du lemme :

Proposition 4.6.2. Soit $\Sigma = \{a, b, c\}$. Le langage $L = \{a^nb^nc^n : n \in \mathbb{N}\} = \{\varepsilon, abc, aabbcc, aaabbbcc, \ldots\}$ n'est pas algébrique.

Démonstration. Appliquons la proposition 4.6.1 au langage L considéré : appelons k l'entier dont le lemme de pompage garantit l'existence. Considérons le mot $t := a^kb^kc^k$: il doit alors exister une factorisation $t = uvwxy$ pour laquelle on a (i) $|vx| \geq 1$, (ii) $|vwx| \leq k$ et (iii) $uv^iwx^iy \in L$ pour tout $i \geq 0$. La propriété (ii) assure que le facteur vwx ne peut pas contenir simultanément les lettres a et c : en effet, tout facteur de t comportant un a et un c doit avoir aussi le facteur b^k, et donc être de longueur $\geq k + 2$. Supposons que vwx ne contienne pas la lettre c (l'autre cas étant complètement analogue) : en particulier, ni v ni x ne la contient, donc le mot uv^iwx^iy, qui est dans L d'après (iii), a le même nombre de c que le mot t initial ; mais comme son nombre de a ou bien de b est différent (d'après (i)), on a une contradiction. ☺

4.6.3. La proposition 4.2.7 peut s'avérer utile pour montrer qu'un langage n'est pas algébrique, en permettant de simplifier le langage auquel on va appliquer le lemme de pompage.

À titre d'exemple, montrons que le langage L formé des mots sur $\{a, b, c\}$ ayant le même nombre total de a, de b et de c (autrement dit $\{w \in \{a, b, c\}^* : |w|_a = |w|_b = |w|_c\}$ où $|w|_x$ désigne le nombre d'occurrences de la lettre x dans le mot w, cf. 1.2.11) n'est pas algébrique ; le plus simple pour le voir est de l'intersecter avec le langage rationnel $M := \{a^ib^jc^k : i, j, k \in \mathbb{N}\}$ (dénoté par l'expression rationnelle $a*b*c*$) : si L était algébrique alors d'après 4.2.7, le langage $L \cap M$ le serait aussi ; mais $L \cap M = \{a^ib^ic^i : i \in \mathbb{N}\}$, et on vient de voir en 4.6.2 qu'il n'est pas algébrique ; c'est donc que L n'est pas non plus algébrique.

4.7 Notions sur l'analyse algorithmique des langages hors contexte

4.7.1. Il est naturel de se poser la question suivante : existe-t-il un algorithme qui, donnée une grammaire hors contexte G, permet de déterminer si un mot donné appartient au langage $L(G)$ engendré par G, et, si oui, d'en trouver un arbre d'analyse ? La réponse à cette question est positive, mais plus délicate que dans le cadre des langages rationnels où la notion d'automate fini a permis de donner un point de vue clair (cf. 3.2.7).

4.7.2. Il existe bien un modèle de calcul qu'on peut imaginer comme l'analogue pour les langages algébriques (et grammaires hors contexte) de ce que les automates finis sont pour les langages rationnels (et expressions régulières) : il s'agit des *automates à pile*. Informellement, un automate à pile non déterministe fonctionne comme un automate fini non déterministe à transitions spontanées, mais il a, en plus de son état courant, accès à une « pile », qui contient des éléments d'un autre alphabet (l'alphabet de pile), et pour choisir la transition à effectuer, il peut consulter, en plus du symbole proposé, les symboles au sommet de la pile (jusqu'à une profondeur bornée), et une fois cette transition effectuée, décider de rajouter, retirer ou remplacer des symboles au sommet de la pile.

De façon plus formelle, un automate à pile non déterministe est la donnée d'un ensemble fini Q d'états, d'un ensemble $I \subseteq Q$ d'états dits initiaux, d'un ensemble $F \subseteq Q$ d'états dits finaux, d'un ensemble fini Γ appelé alphabet de pile, et d'une relation de transition $\delta \subseteq (Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma^*) \times (Q \times \Gamma^*)$. Dire que $((q, t, \lambda), (q', \lambda')) \in \delta$

signifie que l'automate peut transitionner de l'état q avec λ au sommet de la pile vers l'état q' avec λ' (à la place de λ) au sommet de la pile en consommant la lettre t (ou spontanément si $t = \varepsilon$), et l'automate *accepte* un mot w lorsqu'il existe une suite de transitions d'un état initial avec pile vide vers un état final avec pile vide qui consomme les lettres de w.

De façon plus précise, l'automate accepte w lorsqu'il existe $q_0, \ldots, q_n \in Q$ (les états traversés) et $t_1, \ldots, t_n \in (\Sigma \cup \{\varepsilon\})$ (les symboles consommés) et $\gamma_1, \ldots, \gamma_n \in \Gamma^*$ (les états intermédiaires de la pile) et $\lambda_1, \ldots, \lambda_n \in \Gamma^*$ (les mots dépilés) et $\lambda'_1, \ldots, \lambda'_n \in \Gamma^*$ (les mots empilés) tels que : $q_0 \in I$ et $q_n \in F$ et et $\lambda_1\gamma_1 = \varepsilon$ et $\lambda'_n\gamma_n = \varepsilon$ et $((q_{i-1}, t_i, \lambda_i), (q_i, \lambda'_i)) \in \delta$ pour chaque $1 \leq i \leq n$ et $w = t_1 \cdots t_n$ et enfin $\lambda_i\gamma_i = \lambda'_{i-1}\gamma_{i-1}$ pour chaque $1 \leq i \leq n$.

(Il existe différentes variations autour de cette notion, certaines sans importance : notamment, on peut relaxer l'exigence que l'automate termine le calcul avec la pile vide, cela ne change rien à la classe des langages acceptés.)

On peut montrer qu'il y a équivalence entre grammaires hors contexte et automates à pile non déterministes au sens où tout langage engendré par une grammaire hors contexte est le langage accepté par un automate à pile non déterministe et réciproquement. Il n'est d'ailleurs pas très difficile de construire algorithmiquement un automate à pile non déterministe qui accepte le langage engendré par une grammaire hors contexte donnée. Mais une différence essentielle avec les automates finis est que cette fois *le non déterministe est essentiel* : les automates à pile déterministes (qu'il faut définir soigneusement) acceptent strictement moins de langages que les automates à pile non déterministes.

4.7.3. Une approche simple, quoique terriblement inefficace, pour résoudre algorithmiquement, *en théorie*, le problème de décider si un mot w appartient au langage $L(G)$ engendré par une grammaire hors contexte G (absolument quelconque) est la suivante :

— réécrire la grammaire (i.e., la remplacer par une grammaire équivalente) de manière à ce que le membre de droite de chaque production soit *non vide*, quitte à traiter spécialement la question de savoir si $\varepsilon \in L(G)$,

— on a ensuite affaire à une grammaire *monotone*, c'est-à-dire que l'application d'une règle ne peut qu'augmenter (au sens large) la longueur du pseudo-mot en cours de dérivation, ce qui permet d'explorer exhaustivement toutes les possibilités et de s'arrêter dès qu'on dépasse la longueur $|w|$ à atteindre.

Énonçons précisément le résultat en question :

Théorème 4.7.4. Il existe un algorithme qui, donnée une grammaire hors contexte G (sur un alphabet Σ) et un mot $w \in \Sigma^*$, décide si $w \in L(G)$. Autrement dit, les langages algébriques sont *décidables* au sens de 5.1.5.

Plus exactement, on montre :

— Il existe un algorithme qui, donnée une grammaire hors contexte G (sur un alphabet Σ), calcule une grammaire G' (sur le même alphabet Σ et ayant le même ensemble N de nonterminaux que G) dont toutes les productions sont soit de la forme $T \to \alpha$ avec $|\alpha| \geq 1$, et une partie E de $\{\varepsilon\}$ (c'est-à-dire soit $\varnothing$ soit $\{\varepsilon\}$), telles que $L(G) = L(G') \cup E$.

— Il existe un algorithme qui, donnée une grammaire G' comme on vient de le dire, et un mot $w \in \Sigma^*$, décide si $w \in L(G')$.

Démonstration. Montrons d'abord l'affirmation du premier point.

Pour cela, on va d'abord calculer l'ensemble N_0 des nonterminaux « évanescents » de G, un nonterminal T étant dit « évanescent » lorsque $T \Rightarrow^* \varepsilon$. Mais il est évident que toute dérivation de ε dans G ne peut faire intervenir que des nonterminaux évanescents. On peut donc calculer l'ensemble des nonterminaux évanescents de la manière suivante : on commence avec $N_0 = \varnothing$,

et tant qu'il existe dans G une règle $T \to \alpha$, pour laquelle T n'est pas encore dans N_0, avec $\alpha \in N_0^*$ (c'est-à-dire, ne faisant intervenir que des nonterminaux connus pour être évanescents ; y compris $\alpha = \varepsilon$), on ajoute T à N_0 et on recommence. Cette boucle va évidemment terminer en temps fini (il n'y a qu'un nombre fini de nonterminaux) et lorsque c'est le cas N_0 sera l'ensemble des nonterminaux évanescents.

Une fois calculé l'ensemble N_0 des nonterminaux évanescents, on peut définir G' de la manière suivante : pour chaque production $T \to \alpha$ de G, on met dans G' toutes les productions $T \to \alpha'$ où α' est un sous-(pseudo-)mot $\neq \varepsilon$ de α obtenu en effaçant un sous-ensemble quelconque de ses nonterminaux évanescents. La grammaire G' est alors « presque » faiblement équivalente à G : tout arbre de dérivation dans G' donne un arbre de dérivation du même mot dans G, quitte à ajouter, à chaque fois qu'une règle $T \to \alpha'$ est utilisée dans G', les nonterminaux évanescents manquants, qui portent eux-mêmes un sous-arbre de dérivation du mot vide (puisqu'ils sont, justement, évanescents) ; et réciproquement, tout arbre de dérivation dans G en donne un dans G' quitte à effacer tout sous-arbre qui ne porte que des feuilles ε (ce qui assure que sa racine est évanescente). La seule subtilité est qu'on a éventuellement perdu le mot vide dans $L(G)$ (la procédure qu'on vient de décrire conduisant à effacer la totalité de l'arbre), mais il suffit de poser $E = \{\varepsilon\}$ exactement lorsque l'axiome S de G est évanescent pour corriger ce problème.

Montrons maintenant l'affirmation du second point. Pour cela, on considère l'ensemble (fini !) $(\Sigma \cup N)^{\leq |w|}$ de tous les pseudo-mots de longueur $\leq |w|$. On construit et on explore progressivement (par exemple par un algorithme de Dijkstra / parcours en largeur), à partir de S, le graphe sur cet ensemble de sommets dont les arêtes sont les dérivations immédiates de G' (dont le membre de droite soit de longueur $\leq |w|$) : comme la grammaire G' est monotone, toute dérivation de w sera un chemin dans le graphe qu'on vient de dire (elle ne peut pas passer par des pseudo-mots de longueur $> |w|$), donc on peut détecter sur ce graphe fini si une telle dérivation existe. ☺

4.7.5. Expliquons comment on peut approcher de façon algorithmiquement plus efficace le problème de reconnaître si $w \in L(G)$, et fournir une autre démonstration du théorème 4.7.4, tout en continuant à ne faire aucune hypothèse sur la grammaire hors contexte G.

Il s'agit de travailler en deux étapes :

— D'abord, trouver algorithmiquement une grammaire G' et un $E \subseteq \{\varepsilon\}$ tels que $L(G) = L(G') \cup E$, et que G' soit en **forme normale de Chomsky**, c'est-à-dire que toute production de G' est de la forme $T \to UV$ avec U, V (exactement) deux nonterminaux, ou bien $T \to x$ avec x un terminal.

— Ensuite, utiliser un algorithme de programmation dynamique pour calculer, pour chaque facteur u de w (par ordre croissant de taille), l'ensemble de tous les terminaux T tels que $T \Rightarrow^* u$ (ce qui répond notamment à la question de savoir si $S \Rightarrow^* w$).

Détaillons un peu plus chacune de ces étapes.

Pour transformer la grammaire G en une grammaire G' sous forme normale de Chomsky, on effectue les transformations suivantes :

— Introduire pour chaque terminal x un nouveau nonterminal X et une règle $X \to x$, et remplacer chaque occurrence de x par X dans le membre de droite de toute règle autre que $X \to x$. Ceci permet de faire en sorte qu'à part les règles $X \to x$, le membre de droite de toute règle soit uniquement constitué de nonterminaux.

— Pour chaque règle $T \to U_1 \cdots U_n$ dont le membre de droite est de longueur $n \geq 3$, introduire $n-2$ nouveaux nonterminaux $Z_1, \ldots, Z_{n-2}$ et remplacer la règle $T \to U_1 \cdots U_n$ par les $n-1$ règles $T \to U_1 Z_1$ et $Z_i \to U_{i+1} Z_{i+1}$ pour $1 \leq i \leq n-3$ et $Z_{n-2} \to U_{n-1} U_n$. Ceci permet de faire en sorte que le membre de droite de chaque règle soit de longueur ≤ 2.

— Éliminer les règles produisant ε (et calculer l'ensemble E) exactement comme dans la démonstration du premier point de 4.7.4. À ce stade-là, le membre de droite de chaque règle est de longueur exactement

1 ou 2 (et dans le second cas, constitué de deux nonterminaux).

— Pour chaque règle $V \to \alpha$ où α n'est pas un unique nonterminal, et chaque terminal tel qu'il existe une suite $T \to \cdots \to V$ de règles produisant à chaque fois un unique nonterminal, et réécrivant T en V, introduire une règle $T \to \alpha$, puis supprimer toutes les règles dont le membre de droite est un unique nonterminal. (Cette transformation fonctionne exactement comme l'élimination des ε-transitions d'un εNFA, cf. 2.4.6, si on imagine que les règles de la forme $T \to U$ sont des sortes de transitions spontanées d'un nonterminal vers un autre.)

L'ordre de ces transformations peut être légèrement varié, mais celui proposé ci-dessus est sans doute le meilleur.

Une fois la grammaire G' en forme normale de Chomsky connue, lorsqu'on a un mot w dont on cherche à tester s'il appartient à $L(G')$, on calcule, pour chaque facteur u de w (identifié par son point de départ et sa longueur), dans l'ordre croissant de longueur, l'ensemble $\Lambda(u)$ des nonterminaux T tels que $u \in L(G', T)$ (on rappelle, cf. 4.2.5, que $L(G', T) := \{v \in \Sigma^* : T \Rightarrow^* v\}$) :

— Si $|u| = 1$, c'est-à-dire $u \in \Sigma$, il s'agit simplement de l'ensemble des T tels que la règle $T \to u$ soit dans G'.

— Si $|u| \geq 2$, considérer chaque factorisation $u = v_1 v_2$ en deux facteurs de taille ≥ 1 (il y en a $|u| - 1$ possibles), pour chacune d'entre elles, pour chaque $X_1 \in \Lambda(v_1)$ (i.e., tel que $v_1 \in L(G', X_1)$) et chaque $X_2 \in \Lambda(v_2)$ (i.e., tel que $v_2 \in L(G', X_2)$) (ces deux ensembles étant connus par récurrence), et chaque Y tel que la règle $Y \to X_1 X_2$ soit dans G', mettre l'élément Y dans $\Lambda(u)$.

Il n'est pas difficile de se convaincre que ceci construit bien les ensembles $\Lambda(u)$ annoncés : la deuxième partie revient à rechercher toutes les dérivations $Y \Rightarrow X_1 X_2 \Rightarrow^* v_1 v_2 = u$ possibles dans lesquelles X_1 a donné v_1 et X_2 a donné v_2. Une fois les $\Lambda(u)$ connus, tester si $w \in L(G')$ revient à tester si $S \in \Lambda(w)$.

L'algorithme qu'on vient de décrire (pour tester si $w \in L(G')$ une fois G' sous forme normale de Chomsky) porte le nom d'**algorithme de Cocke–Younger–Kasami** ou algorithme **CYK**. Sa complexité est cubique en la longueur de w.

4.7.6. Du point de vue pratique, on ne cherche pas simplement à savoir si un mot appartient au langage engendré par une grammaire, mais aussi à en construire un arbre d'analyse ; par ailleurs, la complexité algorithmique des approches décrites en 4.7.4 et même en 4.7.5 est inacceptable. En contrepartie de ces exigences, on est prêt à accepter de mettre des contraintes sur la grammaire qui la rendent plus facile à analyser : *au minimum*, on supposera que la grammaire est inambiguë (cf. 4.5.1), et en fait, on imposera des contraintes beaucoup plus fortes (par exemple, la grammaire présentée en 4.3.2, bien qu'inambiguë, ne sera pas acceptable car il n'y a pas de manière déterministe de l'analyser, ni même d'analyser le langage qu'elle engendre) ; ces contraintes sont assez techniques et difficiles à décrire : dans la pratique, elles consistent essentiellement à essayer de fabriquer l'analyseur et à constater si l'algorithme échoue.

Il existe deux principales approches pour construire un analyseur pour une grammaire hors contexte (sujette à diverses contraintes supplémentaires) ; dans les deux cas, on construit une sorte d'automate à pile (cf. 4.7.2) déterministe, mais l'utilisation de la pile est très différente dans les deux cas. De façon très simplifiée :

— Les analyseurs **LL** procèdent de façon *descendante* (en anglais « top-down »), parcourent le mot depuis la gauche (« L ») et génèrent la dérivation gauche (« L ») de l'arbre d'analyse fabriqué, en partant de la racine et en descendant jusqu'aux feuilles [22]. La pile de l'analyseur LL sert, intuitivement, à mémoriser les règles qu'on a commencé à reconnaître, en partant de l'axiome de la grammaire, et qui restent encore à compléter.

— Les analyseurs **LR** procèdent de façon *ascendante* (en anglais « bottom-up »), parcourent le mot depuis la gauche (« L ») et génèrent la dérivation droite (« R ») de

22. En botanique, un arbre a la racine en bas et les feuilles en haut ; en informatique, on les représente plutôt racine en haut et feuilles en bas.

l'arbre d'analyse fabriqué, en partant des feuilles et en remontant jusqu'aux racines. La pile de l'analyseur LR sert, intuitivement, à mémoriser les fragments d'arbre déjà construits, en partant des feuilles, et qui restent encore à regrouper.

On peut écrire un analyseur LL ou (plus difficilement) LR à la main dans un cas simple, mais en général ces analyseurs sont fabriqués par des algorithmes systématiques, implémentés dans des programmes tels que YACC ou Bison (qui produit des analyseurs LR, même si Bison peut dépasser ce cadre) ou JavaCC (qui produit des analyseurs LL).

L'idée générale à retenir est que les analyseurs LR sont strictement plus puissants que les analyseurs LL (ils sont capables d'analyser strictement plus de grammaires, cf. 4.3.1), mais leur génération est plus difficile et les messages d'erreur qu'ils retournent en cas de problème de syntaxe sont plus difficiles à comprendre pour l'utilisateur.

4.7.7. Pour illustrer le fonctionnement et les différences des analyseurs LL et LR, considérons une grammaire très simple à analyser comme

$$S \to TS \mid c$$
$$T \to aSb$$

(il s'agit d'une variante de celle considérée en 4.3.4, où on a ajouté un c pour rendre l'analyse plus simple en évitant les productions ε ; on peut imaginer, si on veut, qu'il s'agit d'une forme extrêmement primitive de XML où a représente une balise ouvrante, b une balise fermante, et c une balise vide).

L'approche la plus évidente, si on doit écrire une fonction « analyser un mot comme dérivant de S dans cette grammaire » consiste à coder deux fonctions mutuellement récursives, « chercher un préfixe qui dérive de S » et « chercher un préfixe qui dérive de T ». En observant que tout mot qui dérive de T doit commencer par la lettre a, ce qui permet de distinguer les mots dérivant des règles $S \to TS$ et $S \to c$, on va écrire :

— La fonction « rechercher un préfixe qui dérive de S » (prenant en entrée un mot $w \in \{a, b\}^*$, et renvoyant un préfixe de w et un arbre de dérivation de w à partir de S) est définie comme suit :
  — si la première lettre de w est c, renvoyer le préfixe c et l'arbre trivial $S \to c$, sinon :
  — appeler la fonction « rechercher un préfixe qui dérive de T » sur w, qui retourne un préfixe u de w et un arbre $\mathscr{U}$,
  — appeler la fonction « rechercher un préfixe qui dérive de S » sur le suffixe correspondant t de w (c'est-à-dire le t tel que $w = ut$), qui retourne un préfixe v de t et un arbre $\mathscr{V}$,
  — renvoyer le préfixe uv de w ainsi que l'arbre d'analyse dont la racine est donnée par la règle $S \to TS$ et les sous-arbres $\mathscr{U}$ et $\mathscr{V}$ (i.e., une racine étiquetée S et deux fils étiquetés T et S qui sont chacun racines de sous-arbres donnés par $\mathscr{U}$ et $\mathscr{V}$ respectivement).
— La fonction « rechercher un préfixe qui dérive de T » (prenant en entrée un mot $u \in \{a, b\}^*$, et renvoyant un préfixe de u et un arbre de dérivation de u à partir de T) est définie comme suit :
  — vérifier que la première lettre est un a (sinon, soulever une exception indiquant une erreur d'analyse),
  — appeler la fonction « rechercher un préfixe qui dérive de S » sur le suffixe correspondant x de u (c'est-à-dire le x tel que $u = ax$), qui retourne un préfixe w de x et un arbre $\mathscr{W}$,

— vérifier que la lettre qui suit w dans x est bien b, c'est-à-dire que u commence par awb (sinon, soulever une exception indiquant une erreur d'analyse),
— renvoyer le préfixe awb de u ainsi que l'arbre d'analyse dont la racine est donnée par la règle $T \to aSb$ et les sous-arbres a, $\mathscr{W}$ et b (i.e., une racine étiquetée T et trois fils étiquetés a, S et b, celui du milieu étant racine d'un sous-arbre donné par $\mathscr{W}$).

Cette approche réussit sur cette grammaire très simple (où on peut notamment se convaincre que l'éventuel préfixe dérivant de S ou de T est toujours défini de façon unique). L'analyseur qu'on vient de décrire s'appelle un « analyseur par descente récursive ». Or, plutôt qu'utiliser la récursivité du langage de programmation (c'est-à-dire la pile système), on peut aussi utiliser une pile comme structure de données, et transformer les fonctions récursives en fonctions itératives [23]. On obtient ainsi essentiellement un automate à pile, qui utilise sa pile pour retenir les règles qu'il a commencé à analyser (à partir de la racine de l'arbre d'analyse en cours de construction). On a essentiellement construit un analyseur LL, ou plus exactement LL(1) (le « 1 » indiquant qu'on se contente de lire une unique lettre du mot pour décider quelle règle chercher à analyser), pour ce langage. C'est ici l'approche « descendante » : l'arbre se construit à partir de la racine et la pile sert à retenir les règles qu'on a commencé à reconnaître.

L'approche « ascendante » de la même grammaire serait plutôt la suivante : on parcourt le mot de gauche à droite en gardant de côté une pile (initialement vide) qui pourra contenir les symboles a, S, T, les deux derniers étant alors associés à des arbres d'analyse ;
— si on lit un a, on se contente de l'empiler,
— si on lit un c, on crée un arbre d'analyse $S \to c$ et on empile le S, puis, tant que la pile contient T et S en son sommet, on dépile ces deux symboles, on rassemble les deux arbres d'analyse associés en les mettant sous un $S \to TS$ et on empile le S correspondant,
— si on lit un b, on vérifie que les deux symboles au sommet de la pile sont a et S (sinon on soulève une erreur d'analyse), on les dépile et on rassemble l'arbre d'analyse associé au S en le mettant sous un $T \to aSb$, et enfin on empile un T associé à cet arbre,
— enfin, si on arrive à la fin du mot, la pile ne doit contenir qu'un unique symbole S (sinon on soulève une erreur d'analyse), et l'arbre d'analyse final est celui qui lui est associé.

Il est un peu difficile d'expliquer en général comment construire un tel analyseur, mais sur cet exemple précis il est facile de se convaincre qu'il fonctionne et de comprendre pourquoi : il s'agit essentiellement là d'un analyseur LR (en fait, LR(0), le « 0 » indiquant qu'on n'a jamais eu besoin de regarder au-delà du symbole courant pour décider quoi faire). C'est ici l'approche « ascendante » : l'arbre se construit à partir des feuilles et la pile sert à retenir les nonterminaux au sommet des morceaux d'arbre déjà construits (et éventuellement les arbres eux-mêmes).

5 Introduction à la calculabilité

5.1 Présentation générale

5.1.1. Discussion préalable. On s'intéresse ici à la question de savoir ce qu'un **algorithme** peut ou ne peut pas faire. Pour procéder de façon rigoureuse, il faudrait formaliser la notion d'algorithme (par exemple à travers le concept de machine de Turing) : on a préféré rester

23. Ceci est un fait général : tout ensemble de fonctions récursives peut se réécrire pour utiliser une pile comme structure de données explicite à la place de la pile système.

informel sur cette définition — par exemple « un algorithme est une série d'instruction précises indiquant des calculs à effectuer étape par étape et qui ne manipulent, à tout moment, que des données finies » ou « un algorithme est quelque chose qu'on pourrait, en principe, implémenter sur un ordinateur » — étant entendu que cette notion est déjà bien connue et comprise, au moins dans la pratique. Les démonstrations du fait que tel ou tel problème est décidable par un algorithme ou que telle ou telle fonction est calculable par un algorithme deviennent beaucoup moins lisibles quand on les formalise avec une définition rigoureuse d'algorithme (notamment, programmer une machine de Turing est encore plus fastidieux que programmer un ordinateur en assembleur, donc s'il s'agit d'exhiber un algorithme, c'est probablement une mauvaise idée de l'écrire sous forme de machine de Turing).

Néanmoins, il est essentiel de savoir que ces formalisations existent : on peut par exemple évoquer le paradigme du λ-calcul de Church (la première formalisation rigoureuse de la calculabilité), les fonctions générales récursives (=μ-récursives) à la Herbrand-Gödel-Kleene, les machines de Turing (des machines à états finis capables de lire, d'écrire et de se déplacer sur un ruban infini contenant des symboles d'un alphabet fini dont à chaque instant tous sauf un nombre fini sont des blancs), les machines à registres, le langage « FlooP » de Hofstadter, etc. Toutes ces formalisations sont équivalentes (au sens où, par exemple, elles conduisent à la même notion de fonction calculable ou calculable partielle, définie ci-dessous). La **thèse de Church-Turing** affirme, au moins informellement, que tout ce qui est effectivement calculable par un algorithme [24] est calculable par n'importe laquelle de ces notions formelles d'algorithmes, qu'on peut rassembler sous le nom commun de **calculabilité au sens de Church-Turing**, ou « calculabilité » tout court.

Notamment, quasiment tous les langages de programmation informatique [25], au moins si on ignore les limites des implémentations et qu'on les suppose capables de manipuler des entiers, chaînes de caractère, tableaux, etc., de taille arbitraire (mais toujours finie) [26], sont « Turing-complets », c'est-à-dire équivalents dans leur pouvoir de calcul à la calculabilité de Church-Turing. Pour imaginer intuitivement la calculabilité, on peut donc choisir le langage qu'on préfère et imaginer qu'on programme dedans. Dans la pratique, pour qu'un langage soit Turing-complet, il lui suffit d'être capable de manipuler des entiers de taille arbitraire, de les comparer et de calculer les opérations arithmétiques dessus, et d'effectuer des tests et des boucles.

5.1.2. Il faut souligner qu'on s'intéresse uniquement à la question de savoir ce qu'un algorithme peut ou ne peut pas faire (calculabilité), pas au temps ou aux autres ressources qu'il peut prendre pour le faire (complexité), et on ne cherche donc pas à rendre les algorithmes efficaces en quelque sens que ce soit. Par exemple, pour arguër qu'il existe un algorithme qui décide si un entier naturel n est premier ou non, il suffit de dire qu'on peut calculer tous les produits pq avec $2 \leq p, q \leq n-1$ et tester si l'un d'eux est égal à n, peu importe que cet algorithme soit absurdement inefficace.

De même, nos algorithmes sont capables de manipuler des entiers arbitrairement grands : ceci permet de dire, par exemple, que toute chaîne binaire peut être considérée comme un entier, peu importe le fait que cet entier ait peut-être des milliards de chiffres ; dans les langages

24. Voire, dans certaines variantes de la thèse, tout ce qui est physiquement calculable dans notre Univers (y compris par des processus quantiques).

25. C, C++, Java, Python, JavaScript, Lisp, OCaml, Haskell, Prolog, etc. Certains langages se sont même révélés Turing-complets alors que ce n'était peut-être pas voulu : par exemple, HTML+CSS.

26. Autre condition : ne pas utiliser de générateur aléatoire matériel.

informatiques réels, on a rarement envie de considérer toute donnée comme un entier, mais en calculabilité on peut se permettre de le faire.

5.1.3. Soulignons et développons le point esquissé au paragraphe précédent : plutôt que de travailler avec des « mots » (éléments de Σ^* avec Σ un alphabet fini), en calculabilité, il est possible, dès que cela est commode, de remplacer ceux-ci par des entiers naturels.

Il suffit pour cela de choisir un « codage », parfois appelé **codage de Gödel** permettant de représenter un élément de Σ^* par un entier naturel : la chose importante est que la conversion d'un mot en entier ou d'un entier en mot soit calculable par un algorithme (même très inefficace), si bien que toute manipulation algorithmique sur des mots puisse être convertie en manipulation sur des entiers. Il est commode pour simplifier les raisonnements (quoique pas indispensable) de supposer que le codage est bijectif, c'est-à-dire qu'on dispose d'une bijection $\mathbb{N} \to \Sigma^*$ algorithmiquement calculable et dont la réciproque l'est aussi.

Il existe toutes sortes de manières d'obtenir un tel codage. La plus simple est sans doute la suivante : une fois choisi un ordre total arbitraire sur l'alphabet (fini !) Σ, on peut énumérer les mots sur Σ par ordre de taille et, pour une taille donnée, par ordre lexicographique (par exemple : $\varepsilon, a, b, aa, ab, ba, bb, aaa, aab...$ sur l'alphabet $\{a, b\}$) ; cette énumération est visiblement faisable algorithmiquement, et quitte à numéroter au fur et à mesure qu'on énumère, on obtient un codage comme souhaité (par exemple $0 \leftrightarrow \varepsilon$, $1 \leftrightarrow a$, $2 \leftrightarrow b$, $3 \leftrightarrow ab$, etc.). Insistons sur le fait que ce n'est là qu'une possibilité parmi d'autres et que les détails du codage sont sans importance tant qu'il est calculable ; l'important est que ce soit faisable en théorie et donc qu'on peut considérer qu'au lieu de mots on a affaire à des entiers naturels.

Il va de soi que la concaténation de deux mots, la longueur d'un mot, le miroir d'un mot, sont tous calculables algorithmiquement.

Grâce au codage de Gödel, on peut considérer que, dans cette partie, le terme de « langage » désigne non plus une partie de Σ^* mais une partie de $\mathbb{N}$.

(Le remplacement des mots par des entiers naturels en utilisant un codage comme on vient de le dire est assez standard en calculabilité. Il ne doit pas dérouter : on peut imaginer qu'on a affaire à Σ^* à chaque fois qu'il est question de $\mathbb{N}$ ci-dessous, et on a toujours affaire à la théorie des langages. Le point central est justement que cela n'a pas d'importance ; comme $\mathbb{N}$ est un objet mathématiquement plus simple, c'est surtout pour cela qu'il est utilisé à la place.)

5.1.4. Terminaison des algorithmes. Un algorithme qui effectue un calcul utile doit certainement terminer en temps fini. Néanmoins, même si on voudrait ne s'intéresser qu'à ceux-ci, il n'est pas possible d'ignorer le « problème » des algorithmes qui ne terminent jamais (et ne fournissent donc aucun résultat). C'est le point central de la calculabilité (et du théorème de Turing 5.2.4 ci-dessous) qu'on ne peut pas se débarrasser des algorithmes qui ne terminent pas : on ne peut pas, par exemple, formaliser une notion suffisante [27] de calculabilité dans laquelle tout algorithme termine toujours ; ni développer un langage de programmation suffisamment général dans lequel il est impossible qu'un programme « plante » indéfiniment ou parte en boucle infinie.

(Cette subtilité est d'ailleurs sans doute en partie responsable de la difficulté historique à dégager la bonne notion d'« algorithme » : on a commencé par développer des notions d'algorithmes terminant forcément, comme

27. Tout dépend, évidemment, de ce qu'on appelle « suffisant » : il existe bien des notions de calculabilité, plus faibles que celle de Church-Turing, où tout calcul termine, voir par exemple la notion de fonction « primitive récursive » ou le langage « BlooP » de Hofstadter ; mais de telles notions ne peuvent pas disposer d'une machine universelle comme expliqué plus loin (en raison d'un argument diagonal), donc elles sont nécessairement incomplètes en un certain sens.

les fonctions primitives récursives, et on se rendait bien compte que ces notions étaient forcément toujours incomplètes.)

Définition 5.1.5. On dit qu'une fonction $f\colon \mathbb{N} \to \mathbb{N}$ est **calculable** (ou « récursive ») lorsqu'il existe un algorithme qui prend en entrée $n \in \mathbb{N}$, termine toujours en temps fini, et calcule (renvoie) $f(n)$.

On dit qu'un ensemble $A \subseteq \mathbb{N}$ (un « langage », cf. 5.1.3) est **décidable** (ou « calculable » ou « récursif ») lorsque sa fonction indicatrice $\mathbf{1}_A\colon \mathbb{N} \to \mathbb{N}$ (valant 1 sur A et 0 sur son complémentaire) est calculable. Autrement dit : lorsqu'il existe un algorithme qui prend en entrée $n \in \mathbb{N}$, termine toujours en temps fini, et renvoie « oui » (1) si $n \in A$, et « non » (0) si $n \notin A$ (on dira que l'algorithme « décide » A).

On dit qu'une fonction partielle $f\colon \mathbb{N} \dashrightarrow \mathbb{N}$ (c'est-à-dire une fonction définie sur une partie de $\mathbb{N}$, appelé ensemble de définition de f) est **calculable partielle** (ou « récursive partielle ») lorsqu'il existe un algorithme qui prend en entrée $n \in \mathbb{N}$, termine en temps fini si et seulement si $f(n)$ est définie, et dans ce cas calcule (renvoie) $f(n)$. (Une fonction calculable est donc simplement une fonction calculable partielle qui est toujours définie : on dira parfois « calculable totale » pour souligner ce fait.)

On utilisera la notation $f(n)\downarrow$ pour signifier le fait que la fonction calculable partielle f est définie en n, c'est-à-dire, que l'algorithme en question termine.

On dit qu'un ensemble $A \subseteq \mathbb{N}$ est **semi-décidable** (ou « semi-calculable » ou « semi-récursif ») lorsque la fonction partielle $\mathbb{N} \dashrightarrow \mathbb{N}$ définie exactement sur A et y valant 1, est calculable partielle. Autrement dit : lorsqu'il existe un algorithme qui prend en entrée $n \in \mathbb{N}$, termine en temps fini si et seulement si $n \in A$, et renvoie « oui » (1) dans ce cas [28] (on dira que l'algorithme « semi-décide » A).

On s'est limité ici à des fonctions d'une seule variable (entière), mais il n'y a pas de difficulté à étendre ces notions à plusieurs variables, et de parler de fonction calculable $\mathbb{N}^k \to \mathbb{N}$ (voire $\mathbb{N}^* \to \mathbb{N}$ avec $\mathbb{N}^*$ l'ensemble des suites finies d'entiers naturels) ou de fonction calculable partielle de même type : de toute manière, on peut « coder » un couple d'entiers naturels comme un seul entier naturel (par exemple par $(m,n) \mapsto 2^m(2n+1)-1$, qui définit une bijection calculable $\mathbb{N}^2 \to \mathbb{N}$), ou bien sûr un nombre fini quelconque (même variable), ce qui permet de faire « comme si » on avait toujours affaire à un seul paramètre entier.

5.1.6. Complément : Comme on n'a pas défini formellement la notion d'algorithme, il peut être utile de signaler explicitement les faits suivants (qui devraient être évidents sur toute notion raisonnable d'algorithme) : les fonctions constantes sont calculables ; les opérations arithmétiques usuelles sont calculables ; les projections $(n_1,\dots,n_k) \mapsto n_i$ sont calculables, ainsi que la fonction qui à (m,n,p,q) associe p si $m = n$ et q sinon ; toute composée de fonctions calculables (partielle ou totale) est calculable idem ; si $\underline{m} \mapsto g(\underline{m})$ est calculable (partielle ou totale, où $\underline{m}$ désigne une liste d'entiers) et que $(\underline{m},n,v) \mapsto h(\underline{m},n,v)$ l'est, alors la fonction f définie par récurrence par $f(\underline{m},0) = g(\underline{m})$ et $f(\underline{m},n+1) = h(\underline{m},n,f(\underline{m},n))$ est encore calculable idem (algorithmiquement, il s'agit juste de boucler n fois) ; et enfin, si $(\underline{m},n) \mapsto g(\underline{m},n)$ est calculable partielle, alors la fonction f (aussi notée $\mu_n g$) définie par $f(\underline{m}) = \min\{n : g(\underline{m},n) = 0 \wedge \forall n' < n (g(\underline{m},n')\downarrow)\}$ (et non définie si ce $\min$ n'existe pas) est calculable partielle (algorithmiquement, on teste $g(\underline{m},0), g(\underline{m},1), g(\underline{m},2)\dots$ jusqu'à tomber sur 0). Ces propriétés peuvent d'ailleurs servir à *définir* rigoureusement la notion de fonction calculable, c'est le modèle des fonctions « générales récursives ». (Dans ce qui précède, la notation $\underline{m}$ signifie $m_1,\dots,m_k$.)

5.1.7. Exemples : L'ensemble des nombres pairs, des carrés parfaits, des nombres premiers, sont décidables, c'est-à-dire qu'il est algorithmique de savoir si un nombre est pair, parfait, ou

28. En fait, la valeur renvoyée n'a pas d'importance ; on peut aussi définir un ensemble semi-décidable comme l'ensemble de définition d'une fonction calculable partielle.

premier. Quitte éventuellement à coder les mots d'un alphabet fini comme des entiers naturels (cf. 5.1.3), tout langage rationnel, et même tout langage défini par une grammaire hors contexte, est décidable (cf. 3.2.7 et 4.7.4).

On verra plus bas des exemples d'ensembles qui ne le sont pas, et qui sont ou ne sont pas semi-décidables.

Les deux propositions suivantes, outre leur intérêt intrinsèque, servent à donner des exemples du genre de manipulation qu'on peut faire avec la notion de calculabilité et d'algorithme :

Proposition 5.1.8. Un ensemble $A \subseteq \mathbb{N}$ est décidable si et seulement si A et $\mathbb{N} \setminus A$ sont tous les deux semi-décidables.

Démonstration. Il est évident qu'un ensemble décidable est semi-décidable (si un algorithme décide A, on peut l'exécuter puis effectuer une boucle infinie si la réponse est « non » pour obtenir un algorithme qui semi-décide A) ; il est également évident que le complémentaire d'un ensemble décidable est décidable (quitte à échanger les réponses « oui » et « non » dans un algorithme qui le décide). Ceci montre qu'un ensemble décidable est semi-décidable de complémentaire semi-décidable, i.e., la partie « seulement si ». Montrons maintenant le « si » : si on dispose d'algorithmes T_1 et T_2 qui semi-décident respectivement A et son complémentaire, on peut lancer leur exécution en parallèle sur $n \in \mathbb{N}$ (c'est-à-dire exécuter une étape de T_1 puis une étape de T_2, puis de T_1, et ainsi de suite jusqu'à ce que l'un des deux termine) : comme il y en a toujours (exactement) un qui termine, selon lequel c'est, ceci permet de décider algorithmiquement si $n \in A$ ou $n \notin A$. ☺

Proposition 5.1.9. Un ensemble $A \subseteq \mathbb{N}$ non vide est semi-décidable si et seulement si il existe une fonction calculable $f\colon \mathbb{N} \to \mathbb{N}$ dont l'image ($f(\mathbb{N})$) vaut A (on dit aussi que A est « **calculablement énumérable** » ou « récursivement énumérable »).

Démonstration. Montrons qu'un ensemble semi-décidable non vide est calculablement énumérable. Fixons $n_0 \in A$ une fois pour toutes. Soit T un algorithme qui semi-décide A. On définit une fonction $f\colon \mathbb{N}^2 \to \mathbb{N}$ de la façon suivante : $f(m,n) = n$ lorsque l'algorithme T, exécuté sur l'entrée n, termine au plus m étapes ; sinon, $f(m,n) = n_0$. On a bien sûr $f(m,n) \in A$ dans tous les cas ; par ailleurs, si $n \in A$, comme l'algorithme T appliqué à n doit terminer, on voit que pour m assez grand on a $f(m,n) = n$, donc n est bien dans l'image de f. Ceci montre que $f(\mathbb{N}^2) = A$. Passer à $f\colon \mathbb{N} \to \mathbb{N}$ est alors facile en composant par une bijection calculable $\mathbb{N} \to \mathbb{N}^2$ (par exemple la réciproque de $(m,n) \mapsto 2^m(2n+1) - 1$).

Réciproquement, si A est calculablement énumérable, disons $A = f(\mathbb{N})$ avec f calculable, on obtient un algorithme qui semi-décide A en calculant successivement $f(0)$, $f(1)$, $f(2)$, etc., jusqu'à trouver un k tel que $f(k) = n$ (où n est l'entrée proposée), auquel cas l'algorithme renvoie « oui » (et sinon, il ne termine jamais puisqu'il effectue une boucle infinie à la recherche d'un tel k). ☺

5.1.10. Éclaircissement : Les deux démonstrations ci-dessus font appel à la notion intuitive d'« étape » de l'exécution d'un algorithme. Un peu plus précisément, pour chaque entier m et chaque algorithme T, il est possible d'« exécuter au plus m étapes » de l'algorithme T, c'est-à-dire commencer l'exécution de celui-ci, et si elle n'est pas finie au bout de m étapes, s'arrêter (on n'aura pas le résultat de l'exécution de T, juste l'information « ce n'est pas encore fini » et d'éventuels résultats intermédiaires, mais on peut décider de faire autre chose, y compris reprendre l'exécution plus tard). La longueur d'une « étape » n'est pas spécifiée et n'a pas d'importance, les choses qui importent sont que (A) le fait d'exécuter les m premières étapes de T termine toujours (c'est bien l'intérêt), et

(B) si l'algorithme T termine effectivement, alors pour m suffisamment grand, exécuter au plus m étapes donne bien le résultat final de T comme résultat.

5.1.11. Complément/exercice : Un ensemble $A \subseteq \mathbb{N}$ infini est décidable si et seulement si il existe une fonction calculable $f : \mathbb{N} \to \mathbb{N}$ strictement croissante dont l'image vaut A. (Esquisse : si A est décidable, on peut trouver son n-ième élément par ordre croissant en testant l'appartenance à A de tous les entiers naturels dans l'ordre jusqu'à trouver le n-ième qui appartienne ; réciproquement, si on a une telle fonction, on peut tester l'appartenance à A en calculant les valeurs de la fonction jusqu'à tomber sur l'entier à tester ou le dépasser.) En mettant ensemble ce fait et la proposition, on peut en déduire le fait suivant : tout ensemble semi-décidable infini a un sous-ensemble décidable infini (indication : prendre une fonction qui énumère l'ensemble et jeter toute valeur qui n'est pas strictement plus grande que toutes les précédentes). Cf. exercice A.3.6.

5.2 Machine universelle, et problème de l'arrêt

5.2.1. Codage et machine universelle. Les algorithmes sont eux-mêmes représentables par des mots sur un alphabet fini donc, si on préfère (cf. 5.1.3), par des entiers naturels : on parle aussi de **codage de Gödel** des algorithmes/programmes par des entiers. On obtient donc une énumération $\varphi_0, \varphi_1, \varphi_2, \varphi_3 \ldots$ de toutes les fonctions calculables partielles (la fonction φ_e étant la fonction que calcule l'algorithme [codé par l'entier] e, avec la convention que si cet algorithme est syntaxiquement invalide ou erroné pour une raison quelconque, la fonction φ_e est simplement non-définie partout). Les détails de cette énumération dépendent de la formalisation utilisée pour la calculabilité et ne sont pas importants [29].

On aura tendance, dans la suite, à écrire « le e-ième algorithme », voire « l'algorithme e » (ou « le programme e »), pour désigner l'algorithme codé par l'entier naturel e (c'est-à-dire qu'on identifie librement un programme et son codage de Gödel) : on peut donc écrire que $\varphi_e(n)$ est le résultat de l'exécution du programme e quand on lui fournit n en entrée (ou non défini si cette exécution ne termine pas).

Un point crucial dans cette numérotation des algorithmes par des entiers naturels e est l'existence d'une **machine universelle**, c'est-à-dire d'un algorithme U qui prend en entrée un entier e (codant un algorithme T) et un entier n, et effectue la même chose que T sur l'entrée n (i.e., U termine sur les entrées e et n si et seulement si T termine sur l'entrée n, et, dans ce cas, renvoie la même valeur). Autrement dit, le calcul de $\varphi_e(n)$ en fonction de e et n est lui-même algorithmique.

Informatiquement, ceci représente le fait que les programmes informatiques sont eux-mêmes représentables informatiquement : dans un langage de programmation Turing-complet, on peut écrire un *interpréteur* U pour le langage lui-même (ou pour un autre langage Turing-complet), c'est-à-dire un programme qui prend en entrée la représentation e d'un autre programme et qui exécute ce programme (sur une entrée n).

Mathématiquement, on peut le formuler comme le fait que la fonction (partielle) $(e, n) \mapsto \varphi_e(n)$ (= résultat du e-ième algorithme appliqué sur l'entrée n) est elle-même calculable partielle.

Philosophiquement, cela signifie que la notion d'exécution d'un algorithme est elle-même algorithmique : on peut écrire un algorithme qui, donnée une description (formelle !) d'un algorithme et une entrée à laquelle l'appliquer, effectue l'exécution de l'algorithme fourni sur l'entrée fournie.

29. Par exemple, ceux qui aiment le langage Java peuvent considérer que φ_e désigne le résultat de l'exécution du programme Java représenté par le mot codé par e, si ce programme est valable (remplacer Java par tout autre langage préféré).

On ne peut pas démontrer ce résultat ici faute d'une description rigoureuse d'un modèle de calcul précis, mais il n'a rien de conceptuellement difficile (même s'il peut être fastidieux à écrire dans les détails : écrire un interpréteur d'un langage de programmation demande un minimum d'efforts). La machine universelle dépend, bien sûr, du codage des algorithmes par des entiers naturels.

5.2.2. Compléments : Les deux résultats classiques suivants sont pertinents en lien avec la numérotation des fonctions calculables partielles. • Le *théorème de la forme normale de Kleene* assure qu'il existe un ensemble décidable $\mathscr{T} \subseteq \mathbb{N}^4$ tel que $\varphi_e(n)$ soit défini si et seulement si il existe m, v tels que $(e, n, m, v) \in \mathscr{T}$, et dans ce cas $\varphi_e(n) = v$ (pour s'en convaincre, il suffit de définir $\mathscr{T}$ comme l'ensemble des (e, n, m, v) tels que le e-ième algorithme exécuté sur l'entrée n termine en au plus m étapes et renvoie le résultat v : le fait qu'on dispose d'une machine universelle et qu'on puisse exécuter m étapes d'un algorithme assure que cet ensemble est bien décidable — il est même « primitif récursif »). • Le *théorème s-m-n* assure qu'il existe une fonction calculable s telle que $\varphi_{s(e,\underline{m})}(\underline{n}) = \varphi_e(\underline{m}, \underline{n})$ (intuitivement, donné un algorithme qui prend plusieurs entrées et des valeurs $\underline{m}$ de certaines de ces entrées, on peut fabriquer un nouvel algorithme dans lequel ces valeurs ont été fixées — c'est à peu près trivial — mais de plus, cette transformation est *elle-même algorithmique*, i.e., on peut algorithmiquement substituer des valeurs $\underline{m}$ dans un programme [codé par l'entier] e : c'est intuitivement clair, mais cela ne peut pas se démontrer avec les seules explications données ci-dessus sur l'énumération des fonctions calculables partielles, il faut regarder précisément comment le codage standard est fait pour une formalisation de la calculabilité).

La machine universelle n'a rien de « magique » : elle se contente de suivre les instructions de l'algorithme T qu'on lui fournit, et termine si et seulement si T termine. Peut-on savoir à l'avance si T terminera ? C'est le fameux « problème de l'arrêt ».

Intuitivement, le « problème de l'arrêt » est la question « l'algorithme suivant termine-t-il sur l'entrée suivante » ?

Définition 5.2.3. On appelle **problème de l'arrêt** (ou « langage de l'arrêt ») l'ensemble des couples (e, n) tels que le e-ième algorithme termine sur l'entrée n, i.e., $\{(e, n) \in \mathbb{N}^2 : \varphi_e(n) \downarrow\}$ (où la notation « $\varphi_e(n) \downarrow$ » signifie que $\varphi_e(n)$ est défini, i.e., l'algorithme termine). Quitte à coder les couples d'entiers naturels par des entiers naturels (par exemple par $(e, n) \mapsto 2^e(2n + 1) - 1$), on peut voir le problème de l'arrêt comme une partie de $\mathbb{N}$. On peut aussi préférer [30] définir le problème de l'arrêt comme $\{e \in \mathbb{N} : \varphi_e(e) \downarrow\}$, on va voir dans la démonstration ci-dessous que c'est cet ensemble-là qui la fait fonctionner.

(On pourrait aussi définir le problème de l'arrêt comme $\{e \in \mathbb{N} : \varphi_e(0) \downarrow\}$ si on voulait, ce serait moins pratique pour la démonstration, mais cela ne changerait rien au résultat comme on peut le voir en appliquant le théorème s-m-n.)

Théorème 5.2.4 (Turing)**.** Le problème de l'arrêt est semi-décidable mais non décidable.

Démonstration. Le problème de l'arrêt est semi-décidable en vertu de l'existence d'une machine universelle : donnés e et n, on exécute le e-ième algorithme sur l'entrée n (c'est ce que fait la machine universelle), et s'il termine on renvoie « oui » (et s'il ne termine pas, bien sûr, la seule possibilité est de ne pas terminer).

Montrons par l'absurde que le problème de l'arrêt n'est pas décidable. S'il l'était, on pourrait définir un algorithme qui, donné un entier e, effectue les calculs suivants : (1°) utiliser le problème de l'arrêt (supposé décidable !) pour savoir, algorithmiquement en temps fini, si le e-ième algorithme termine quand on lui passe son propre numéro e en entrée, i.e., si $\varphi_e(e) \downarrow$, et ensuite (2°) si oui, effectuer une boucle infinie, et si non, terminer, en renvoyant, disons, 42.

30. Même si au final c'est équivalent, c'est *a priori* plus fort de dire que $\{e \in \mathbb{N} : \varphi_e(e) \downarrow\}$ n'est pas décidable que de dire que $\{(e, n) \in \mathbb{N}^2 : \varphi_e(n) \downarrow\}$ ne l'est pas.

L'algorithme qui vient d'être décrit aurait un certain numéro, disons, p, et la description de l'algorithme fait que, quel que soit e, la valeur $\varphi_p(e)$ est indéfinie si $\varphi_e(e)$ est définie tandis que $\varphi_p(e)$ est définie (de valeur 42) si $\varphi_e(e)$ est indéfinie. En particulier, en prenant $e = p$, on voit que $\varphi_p(p)$ devrait être défini si et seulement si $\varphi_p(p)$ n'est pas défini, ce qui est une contradiction. ☺

La démonstration ci-dessus est une instance de l'« argument diagonal » de Cantor, qui apparaît souvent en mathématiques. (La « diagonale » en question étant le fait qu'on considère $\varphi_e(e)$, i.e., on passe le numéro e d'un algorithme en argument à cet algorithme lui-même, donc on regarde la diagonale de la fonction de deux variables $(e, n) \mapsto \varphi_e(n)$; en modifiant les valeurs sur cette diagonale, on produit une fonction qui ne peut pas se trouver dans une ligne φ_p.) Une variante facile du même argument permet de fabriquer des ensembles non semi-décidables (voir 5.2.7 ci-dessous), ou bien on peut appliquer ce qui précède :

Corollaire 5.2.5. Le complémentaire du problème de l'arrêt n'est pas semi-décidable.

Démonstration. On a vu que le problème de l'arrêt n'est pas décidable, et qu'un ensemble est décidable si et seulement si il est semi-décidable et que son complémentaire l'est aussi : comme le problème de l'arrêt est bien semi-décidable, son complémentaire ne l'est pas. ☺

5.2.6. Complément : L'argument diagonal est aussi au cœur du (voire, équivalent au) *théorème de récursion de Kleene*, qui affirme que pour toute fonction calculable partielle $h\colon \mathbb{N}^2 \dashrightarrow \mathbb{N}$, il existe p tel que $\varphi_p(n) = h(p, n)$ pour tout n (la signification intuitive de ce résultat est qu'on peut supposer qu'un programme a accès à son propre code source p, i.e., on peut programmer comme s'il recevait en entrée un entier p codant ce code source ; ceci permet par exemple — de façon anecdotique mais amusante — d'écrire des programmes, parfois appelés « quines », qui affichent leur propre code source sans aller le chercher sur disque ou autre tricherie). *Démonstration :* donné $e \in \mathbb{N}$, on considère $s(e, m)$ tel que $\varphi_{s(e,m)}(n) = \varphi_e(m, n)$: le théorème s-m-n (cf. ci-dessus) assure qu'une telle fonction calculable $(e, m) \mapsto s(e, m)$ existe, et $(e, n) \mapsto h(s(e, e), n)$ est alors aussi calculable partielle ; il existe donc q tel que $\varphi_q(e, n) = h(s(e, e), n)$: on pose $p = s(q, q)$, et on a $\varphi_p(n) = \varphi_q(q, n) = h(s(q, q), n) = h(p, n)$, comme annoncé. ☺ La non-décidabilité du problème de l'arrêt s'obtient en appliquant (de nouveau par l'absurde) ce résultat à $h(e, n)$ la fonction qui n'est pas définie si $\varphi_e(n)$ l'est et qui vaut 42 si $\varphi_e(n)$ n'est pas définie.

La non-décidabilité du problème de l'arrêt est un résultat fondamental, car très souvent les résultats de non-décidabilité soit sont démontrés sur un modèle semblable, soit s'y ramènent directement : pour montrer qu'un certain ensemble A (un « problème ») n'est pas décidable, on cherche souvent à montrer que si un algorithme décidant A existait, on pourrait s'en servir pour construire un algorithme résolvant le problème de l'arrêt.

5.2.7. Bonus / exemple(s) : L'ensemble des $e \in \mathbb{N}$ tels que la fonction calculable partielle φ_e soit totale (i.e., définie sur tout $\mathbb{N}$) n'est pas semi-décidable. En effet, s'il l'était, d'après ce qu'on a vu, il serait « calculablement énumérable », c'est-à-dire qu'il existerait une fonction calculable $f\colon \mathbb{N} \to \mathbb{N}$ dont l'image soit exactement l'ensemble des e pour lesquels φ_e est totale, i.e., toute fonction calculable totale s'écrirait sous la forme $\varphi_{f(k)}$ pour un certain k. Mais la fonction $n \mapsto \varphi_{f(n)}(n) + 1$ est calculable totale, donc il devrait exister un m tel que cette fonction s'écrive $\varphi_{f(m)}$, c'est-à-dire $\varphi_{f(m)}(n) = \varphi_{f(n)}(n) + 1$, et on aurait alors en particulier $\varphi_{f(m)}(m) = \varphi_{f(m)}(m) + 1$, une contradiction. • Son complémentaire, c'est-à-dire l'ensemble des $e \in \mathbb{N}$ tels que la fonction calculable partielle φ_e ne soit pas totale, n'est pas non plus semi-décidable. En effet, supposons qu'il existe un algorithme qui, donné e, termine si et seulement si φ_e n'est pas totale. Donnés e et m, considérons l'algorithme qui prend une entrée n, *ignore* celle-ci, et effectue le calcul $\varphi_e(m)$: ceci définit une fonction calculable partielle (soit totale et constante, soit définie nulle part !) $\varphi_{s(e,m)}$ où s est calculable (on applique ici le théorème

s-m-n) — en appliquant à $s(e, m)$ l'algorithme supposé semi-décider si une fonction récursive partielle est non-totale, on voit qu'ici il semi-décide si $\varphi_e(m)$ est non-défini, autrement dit on semi-décide le complémentaire du problème de l'arrêt, et on a vu que ce n'était pas possible !

5.2.8. Exercice : Considérons une fonction h qui à e associe un nombre au moins égal au nombre d'étapes (cf. ci-dessus) du calcul de $\varphi_e(e)$, si celui-ci termine, et une valeur quelconque si $\varphi_e(e)$ n'est pas défini. Alors h n'est pas calculable. (Indication : si elle l'était, on pourrait décider si $\varphi_e(e)$ est défini en exécutant son calcul pendant $h(e)$ étapes.) On peut même montrer que $H(n) := \max\{h(i) : i \leq n\}$ domine asymptotiquement n'importe quelle fonction calculable mais c'est un peu plus difficile.

5.2.9. Application à la logique : Sans rentrer dans les détails de ce que signifie un « système formel », on peut esquisser, au moins informellement, les arguments suivants. Imaginons qu'on ait formalisé la notion de démonstration mathématique (c'est-à-dire qu'on les écrit comme des mots dans un alphabet indiquant quels axiomes et quelles règles logiques sont utilisées) : même sans savoir quelle est exactement la logique formelle, le fait de *vérifier* qu'une démonstration est correcte doit certainement être algorithmique (il s'agit simplement de vérifier que chaque règle a été correctement appliquée), autrement dit, l'ensemble des démonstrations est décidable. L'ensemble des théorèmes, lui, est semi-décidable (on a un algorithme qui semi-décide si un certain énoncé est un théorème en énumérant toutes les chaînes de caractères possibles et en cherchant s'il s'agit d'une démonstration valable dont la conclusion est l'énoncé recherché). Or l'ensemble des théorèmes n'est pas décidable : en effet, si on avait un algorithme qui permet de décider si un énoncé mathématique est un théorème, on pourrait appliquer cet algorithme à l'énoncé formel (*)« le e-ième algorithme termine sur l'entrée e », en observant qu'un tel énoncé, s'il est vrai, est forcément démontrable (i.e., si l'algorithme termine, on peut *démontrer* ce fait en écrivant étape par étape l'exécution de l'algorithme pour constituer une démonstration qu'il a bien été appliqué jusqu'au bout et a terminé), et en espérant que s'il est démontrable alors il est vrai : on aurait alors une façon de décider le problème de l'arrêt, une contradiction. Mais du coup, l'ensemble des non-théorèmes ne peut pas être semi-décidable ; or comme l'ensemble des énoncés P tels que $\neg P$ (« non-P », la négation logique de P) soit un théorème est semi-décidable (puisque l'ensemble des théorèmes l'est), ils ne peuvent pas coïncider. Ceci montre qu'il existe un énoncé tel que ni P ni $\neg P$ ne sont des théorèmes : c'est une forme du *théorème de Gödel* que Turing cherchait à démontrer ; mieux : en appliquant aux énoncés du type (*), on montre ainsi qu'il existe un algorithme qui *ne termine pas* mais dont la non-terminaison *n'est pas démontrable*. (Modulo quelques hypothèses qui n'ont pas été explicitées sur le système formel dans lequel on travaille.)

A Exercices

A.1 Langages rationnels et automates

Exercice A.1.1.

Soit $\Sigma = \{0, 1\}$. On appelle *mot binaire* un mot sur l'alphabet Σ, et mot binaire *normalisé* un mot binaire qui *soit* commence par 1, *soit* est exactement égal à 0.

(1) Montrer que le langage $L_n = \{0, 1, 10, 11, 100, 101, \ldots\}$ des mots binaires normalisés est rationnel en exhibant directement une expression rationnelle qui le dénote, et montrer qu'il est reconnaissable en exhibant directement un automate fini qui le reconnaît.

(2) On définit la *valeur numérique* d'un mot binaire $x_{n-1} \cdots x_0$ comme $\sum_{i=0}^{n-1} x_i 2^i$ (où x_i vaut 0 ou 1 et est numéroté de 0 pour le chiffre le plus à droite à $n-1$ pour le plus à gauche) ; la valeur numérique du mot vide ε est 0.

Parmi les langages suivants, certains sont rationnels. Dire lesquels et justifier brièvement pourquoi ils le sont (on ne demande pas de justifier pourquoi ceux qui ne sont pas rationnels ne le sont pas) :

(a) le langage L_a des mots binaires dont la valeur numérique est paire,

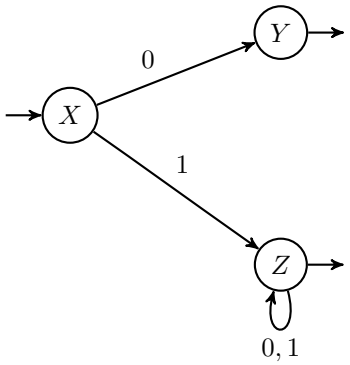
(b) le langage L_b des mots binaires *normalisés* dont la valeur numérique est paire,

(c) le langage L_c des mots binaires dont la valeur numérique est multiple de 3 (indication : selon que n est congru à 0, 1 ou 2 modulo 3, et selon que x vaut 0 ou 1, à quoi est congru $2n+x$ modulo 3 ?),

(d) le langage L_d des mots binaires dont la valeur numérique est un nombre premier,

(e) le langage L_e des mots binaires dont la valeur numérique est une puissance de 2, i.e., de la forme 2^i pour $i \in \mathbb{N}$,

Corrigé. (1) On peut écrire $L_n = L_{0|1(0|1)*}$, langage dénoté par l'expression rationnelle $0|1(0|1)*$. Ce langage est reconnu, par exemple, par le DFAI suivant :

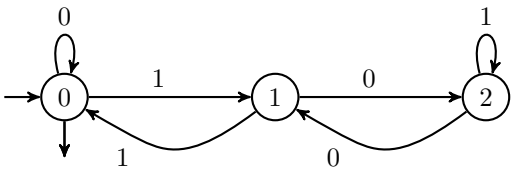


(2) (a) Le langage L_a est rationnel car il s'agit du langage des mots binaires qui soit sont le mot vide soit finissent par 0 : il est dénoté par l'expression rationnelle $\varepsilon|(0|1)*0$. (b) On a $L_b = L_a \cap L_n$ et on a vu que L_a et L_n sont rationnels, donc L_b l'est aussi (on peut aussi exhiber une expression rationnelle qui le dénote : $0|1(0|1)*0$).

(c) Ajouter un 0 ou un 1 à la fin d'un mot binaire de valeur numérique n le transforme en un mot de valeur numérique $2n + x$ où x est le chiffre affixé. Considérons les six combinaisons entre les trois cas possibles de la valeur numérique n modulo 3 et les deux cas possibles de la valeur de x :

$n \equiv ?\pmod 3$	$x = ?$	$2n + x \equiv ?\pmod 3$
0	0	0
0	1	1
1	0	2
1	1	0
2	0	1
2	1	2

Ceci définit un DFA dont les trois états correspondent aux trois valeurs possibles de n modulo 3, la transition $n \to n'$ étiquetée par x correspond au passage de n à $2n + x$ modulo 3, c'est-à-dire :



(On a marqué l'état 0 comme initial car le mot vide a une valeur numérique congrue à 0 modulo 3, et seul 0 comme final car on veut reconnaître les multiples de 3.)

(d) Le langage L_d n'est pas rationnel (on pourrait le démontrer à l'aide du lemme de pompage, mais ce n'est pas très facile).

(e) Le langage L_e est rationnel car il s'agit du langage dénoté par l'expression rationnelle $0*10*$. ✓

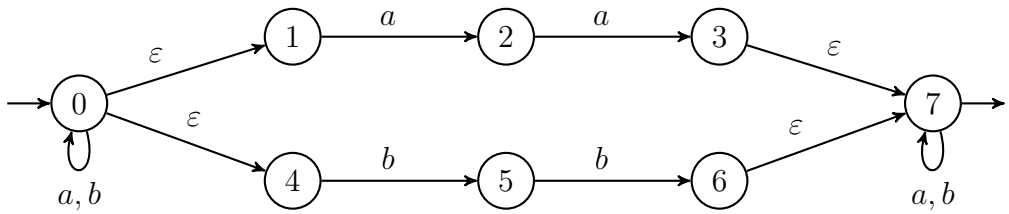
Exercice A.1.2.

Soit $\Sigma = \{a\}$. Montrer que le langage $L = \{a^2, a^3, a^5, a^7, a^{11}, a^{13} \ldots\}$ constitué des mots ayant un nombre *premier* de a, n'est pas rationnel.

Corrigé. Supposons par l'absurde que L soit rationnel. D'après le lemme de pompage, il existe un certain k tel que tout mot de L de longueur $\geq k$ se factorise sous la forme uvw avec (i) $|v| \geq 1$, (ii) $|uv| \leq k$ et (iii) $uv^i w \in L$ pour tout $i \geq 0$. Soit p un nombre premier supérieur ou égal à k (qui existe car l'ensemble des nombres premiers est infini) : le mot $a^p \in L$ admet une factorisation comme on vient de dire. Posons $|u| =: m$ et $|v| =: n$, si bien que $|w| = p - m - n$. On a alors $n \geq 1$ d'après (i), et $|uv^i w| = m + in + (p - m - n) = p + (i-1)n$ est premier pour tout $i \geq 0$ d'après (iii). En particulier pour $i = p + 1$ on voit que $p + pn = p(n + 1)$ est premier, ce qui contredit le fait qu'il s'agit d'un multiple non-trivial ($n + 1 \geq 2$) de p. ✓

Exercice A.1.3.

On considère l'automate suivant :



(0) Décrire brièvement le langage accepté par l'automate en question.

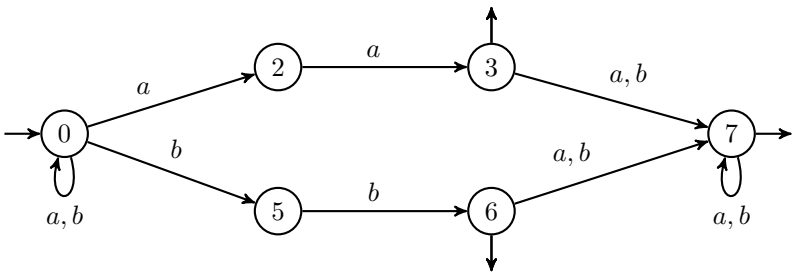
(1) Cet automate est-il déterministe ? Si non, le déterminiser.

(2) Minimiser l'automate déterminisé (on doit trouver un DFA ayant quatre états). Décrire brièvement la signification de ces quatre états, de façon à vérifier qu'il accepte le même langage que décrit en (0).

(3) Éliminer les états de l'automate d'origine de façon à obtenir une expression rationnelle dénotant le langage reconnu par le langage décrit en (0).

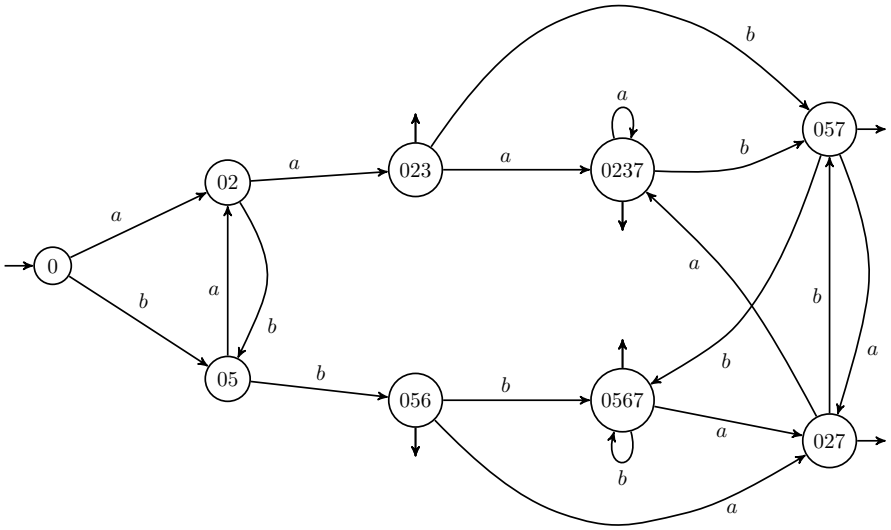
Corrigé. (0) L'automate proposé accepte les mots ayant soit deux a consécutifs (en passant par le chemin $0 \to 1 \to 2 \to 3 \to 7$) soit deux b consécutifs (en passant par le chemin $0 \to 4 \to 5 \to 6 \to 7$).

(1) L'automate ayant des ε-transitions, il ne peut pas être déterministe : on a affaire à un εNFA. Avant de le déterminiser, on élimine ses ε-transitions : la ε-fermeture de 0 est $\{0, 1, 4\}$, celle de 3 est $\{3, 7\}$ et celle de 6 est $\{6, 7\}$; on est amené au NFA suivant :



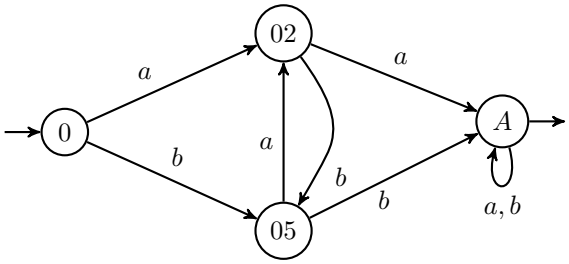
(Les états 1 et 4 étant inaccessibles, ils ont été retirés. Il ne faut pas oublier que 3 et 6 sont finaux puisqu'ils ont l'état final 7 dans leur ε-fermeture.)

On peut maintenant procéder à la déterminisation. Pour abréger les noms des états, on note, par exemple, 023 pour $\{0, 2, 3\}$. En construisant de proche en proche, on obtient le DFA suivant :



(À titre d'exemple, la transition étiquetée b partant de l'état 0237 conduit à l'état 057 car les transitions étiquetées b dans le NFA précédent et partant des états parmi $\{0, 2, 3, 7\}$ sont $0 \to 0$, $0 \to 5$, $3 \to 7$ et $7 \to 7$. Tous les états contenant l'un des symboles $3, 6, 7$ sont finaux.)

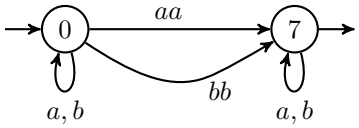
(2) On a affaire à un DFA (sous-entendu : *complet*) dont tous les états sont accessibles, on peut donc appliquer directement l'algorithme de Moore. Une première partition sépare les états finaux, soit $023, 0237, 027, 056, 0567, 057$ des non-finaux, soit $0, 02, 05$. L'étape suivante distingue 02 parce que sa a-transition conduit à une classe différente que celles de 0 et 05, et 05 parce que sa b-transition conduit à une classe différente de 0 et 02. Les étapes suivantes ne changent rien. Finalement, on arrive à un automate à quatre états :



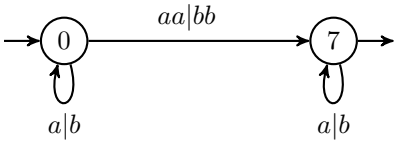
où A représente la classe de tous les états finaux de l'automate précédent.

La signification des quatre états est la suivante : l'état 0 signifie que l'automate n'a encore rien lu, l'état 02 signifie que l'automate vient de lire un a, le 05 signifie qu'il vient de lire un b, le A signifie qu'il a lu deux a consécutifs ou bien deux b consécutifs. Sur cette description, il est clair que l'automate accepte les mots contenant deux a consécutifs ou bien deux b consécutifs.

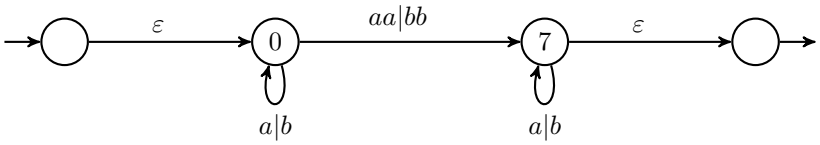
(3) L'élimination des états 1 à 6 peut se faire dans un ordre quelconque et conduit à l'automate (à transitions étiquetées par des expressions rationnelles) suivant :



Il y a plusieurs flèches entre les mêmes états : quitte à les remplacer par des disjonctions, on obtient :



Enfin, on doit éliminer les états 0 et 7 eux-mêmes : pour cela, on ajoute un nouvel état initial qui ne soit la cible d'aucune flèche et un nouvel état final d'où ne part aucune flèche,



Et l'élimination des états 0 et 7 dans un ordre quelconque conduit finalement à l'expression rationnelle $(a|b)*(aa|bb)(a|b)*$. ✓

Exercice A.1.4.

Soit $\Sigma = \{a, b\}$.

(1) Tracer l'automate de Thompson de l'expression rationnelle $a*(baa*)*$. Cet automate est-il déterministe ?

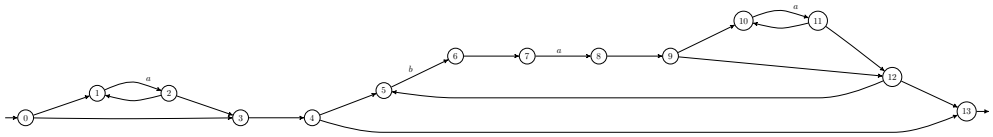
(2) En éliminer les transitions spontanées.

(3) Déterminiser l'automate obtenu (on demande un automate complet).

(4) Minimiser l'automate obtenu (on demande un automate complet).

(5) Vérifier le résultat en décrivant en français le langage dénoté par l'expression rationnelle initiale et reconnu par l'automate finalement obtenu.

Corrigé. (1) L'automate de Thompson de $a*(baa*)*$ doit comporter 14 états puisque cette expression rationnelle contient 7 symboles parenthèses non comptées. Il est le suivant (où on a omis les ε sur les transitions spontanées) :

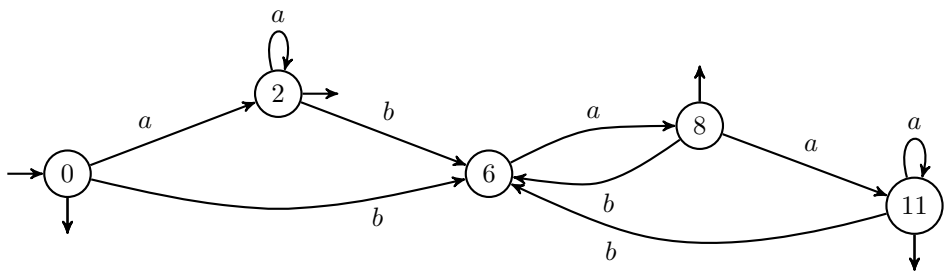


Cet automate n'est pas déterministe : un automate comportant des ε-transitions est forcément non-déterministe.

(2) Tous les états autres que 0 (car il est initial) et $2, 6, 8, 11$ (car des transitions non spontanées y aboutissent) vont disparaître ; les ε-fermetures $C(q)$ de ces états sont les suivantes :

q	ε-fermeture $C(q)$
0	$\{0, 1, 3, 4, 5, 13\}$
2	$\{1, 2, 3, 4, 5, 13\}$
6	$\{6, 7\}$
8	$\{5, 8, 9, 10, 12, 13\}$
11	$\{5, 10, 11, 12, 13\}$

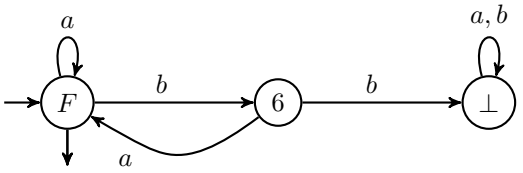
En remplaçant chaque transition $q^\sharp \to q'$ étiquetée d'un $x \in \Sigma$ dans l'automate par une transition $q \to q'$ pour chaque état q tel que $q^\sharp \in C(q)$, on obtient le NFA suivant :



Les états $0, 2, 8, 11$ sont finaux car ce sont eux qui ont 13 dans leur ε-fermeture.

(3) L'automate ainsi obtenu est déjà déterministe incomplet ; pour le déterminiser-compléter, il n'y a qu'à ajouter un puits $\perp$ avec la seule transition qui manque, c'est-à-dire une transition étiquetée par b depuis l'état 6 (et des transitions de $\perp$ vers lui-même étiquetées a et b). Nous ne représentons pas l'automate à 6 états ainsi fabriqué.

(4) On part de l'algorithme déterministe complet obtenu à la question (3), et on lui applique l'algorithme de minimisation. On sépare d'abord ses états en deux classes, les finaux $\{0, 2, 8, 11\}$ et les non-finaux $\{6, \perp\}$. La transition étiquetée a sépare les états 6 et $\perp$ car le premier aboutit dans la classe $\{0, 2, 8, 11\}$ tandis que le second aboutit dans la classe $\{6, \perp\}$. On vérifie ensuite qu'aucune transition ne sépare des états. L'automate minimal est donc le suivant :

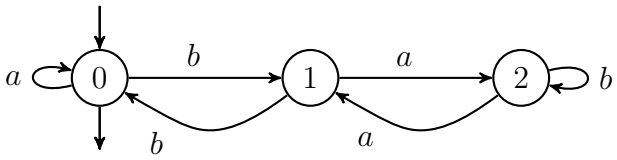


où l'état F représente la classe $0 \equiv 2 \equiv 8 \equiv 11$.

(5) Il s'agit du langage constitué des mots n'ayant jamais deux b consécutifs ni de b final, c'est-à-dire des mots dans lesquels chaque b est suivi d'au moins un a : l'expression rationnelle initiale le présente comme le langage constitués des mots formés d'un nombre quelconque de a puis d'un nombre quelconque de répétitions d'un b suivi d'au moins un a. L'automate final interdit les suites de deux b consécutifs comme ceci : l'état F correspond à la situation où on ne vient pas de rencontrer un b (=la lettre précédente était un a ou bien on vient de commencer le mot) et on n'en a jamais rencontré deux, l'état 6 à la situation où on vient de rencontrer un b et on n'en a jamais rencontré deux, et l'état $\perp$ à la situation où on a rencontré au moins une fois deux b consécutifs. Avec cette description, il est clair que l'automate fait ce qui était demandé. ✓

Exercice A.1.5.

Donner plusieurs (au moins trois) expressions rationnelles équivalentes dénotant le langage reconnu par l'automate suivant sur l'alphabet $\Sigma = \{a, b\}$:



(On pourra considérer les ordres suivants d'élimination des états : (A) $2, 1, 0$, ensuite (B) $1, 2, 0$ et enfin (C) $0, 2, 1$.)

Corrigé. (A) Si on commence par éliminer l'état 2 (en considérant l'automate comme un automate à transitions étiquetées par des expressions rationnelles), l'état 1 reçoit une transition vers lui-même étiquetée $ab{*}a$. Si on élimine l'état 1, l'état 0 reçoit à la place une transition vers lui-même étiquetée par $b(ab{*}a){*}b$, qu'on peut fusionner avec la transition vers lui-même déjà existante étiquetée par a pour une seule étiquetée par $a|b(ab{*}a){*}b$. Quitte éventuellement à ajouter un nouvel état initial (conduisant à 0 par une transition spontanée) et un nouvel état final (vers lequel 0 conduit par une transition spontanée) et à éliminer n'état 0, on obtient finalement l'expression rationnelle

$$(a|b(ab{*}a){*}b){*}$$

(B) Si on commence par éliminer l'état 1, il apparaît une transition $0 \to 2$ étiquetée ba et une $2 \to 0$ étiquetée ab (si on veut appliquer l'algorithme de façon puremnet mécanique, l'état 1 n'a pas de transition vers lui-même, c'est-à-dire qu'on pourrait l'étiqueter par $\perp$, symbole d'expression rationnelle qui dénote le lagange vide, et l'expression rationnelle $\perp{*}$ est équivalente à ε) ; mais il ne faut pas oublier que l'état 2 reçoit lui aussi une transition vers lui-même (en passant par 1) étiquetée aa, qu'on peut fusionner avec la transition vers lui-même déjà existante étiquetée par b pour obtenir une transition étiquetée $b|aa$; de même, l'état 0 reçoit une transition étiquetée bb, qu'on peut fusionner avec celle existante pour obtenir $a|bb$. L'élimination de l'état 2 fait alors apparaître une

transition de 0 vers lui-même étiquetée $ba(b|aa)*ab$, qu'on peut fusionner avec la transition vers lui-même déjà étiquetée par $a|bb$ pour une seule étiquetée par $a|bb|ba(b|aa)*ab$. On obtient finalement

$$(a|bb|ba(b|aa)*ab)*$$

(en particulier, cette expression est équivalente à celle obtenue précédemment).

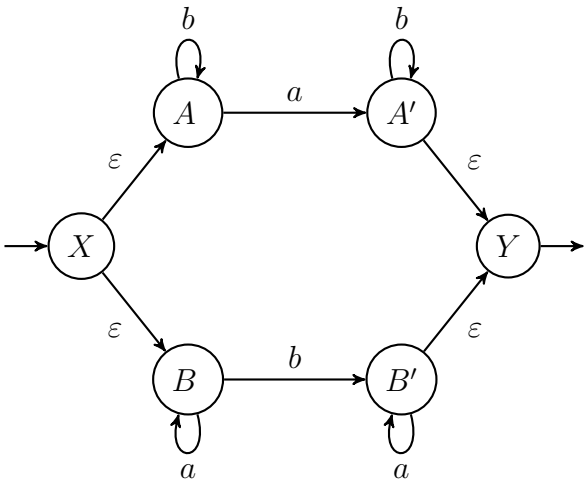
(C) Si on préfère commencer par éliminer l'état 0, il faut au préalable ajouter un nouvel état initial I (conduisant à 0 par une transition spontanée) et un nouvel état final F (vers lequel 0 conduit par une transition spontanée). L'élimination de l'état 0 fait apparaître une transition de I vers 1 étiquetée $a*b$, une transition 1 vers F étiquetée $ba*$, et une transition de l'état 1 vers lui-même étiquetée $ba*b$, et enfin une transition de I vers F étiquetée $a*$. L'élimination de l'état 2 fait apparaître une transition de 1 vers lui-même étiquetée $ab*a$, qu'on peut fusionner avec celle déjà existante étiquetée $ba*b$ pour obtenir une transition $(ab*a|ba*b)$. Finalement, l'élimination de l'état 1 done l'expression rationnelle

$$a*|a*b(ab*a|ba*b)*ba*$$

(toujours équivalente aux précédentes). ✓

Exercice A.1.6.

On considère l'automate fini M sur l'alphabet $\Sigma = \{a, b\}$ représenté par la figure suivante :



(0) De quelle sorte d'automate s'agit-il ? (Autrement dit : est-il déterministe ou non ? avec transitions spontanées ou non ?)

(1a) Décrire brièvement, en français, le langage L reconnu (=accepté) par l'automate M, puis donner une expression rationnelle qui le dénote. (On pourra préférer traiter la question (1b) d'abord.)

(1b) Pour chacun des mots suivants, dire s'ils sont dans L ou non : ε, a, b, ab, aa, aab, $aabb$, $abab$, $ababa$. (Note : il est recommandé de réutiliser ces mots pour vérifier rapidement les réponses aux questions suivantes et ainsi détecter d'éventuelles erreurs lors des transformations des automates.)

(2) Éliminer les transitions spontanées de l'automate M. (On supprimera les états devenus inutiles.) On appellera M_2 l'automate obtenu.

(3) Déterminiser l'automate M_2 obtenu en (2), si nécessaire. (On demande un automate déterministe complet.) On appellera M_3 l'automate déterminisé.

Pour simplifier le travail du correcteur, on demande de représenter M_3 de sorte que les transitions étiquetées par a soient, dans la mesure du possible, horizontales de la gauche vers la droite, et celles étiquetées par b, verticales du haut vers le bas.

(4) Minimiser l'automate M_3 obtenu en (3), si nécessaire (justifier).

(5) Donner un automate (de n'importe quelle sorte) qui reconnaît le langage $\overline{L} = \Sigma^* \setminus L$ complémentaire de L.

(6) Décrire brièvement, en français, ce langage complémentaire $\overline{L}$.

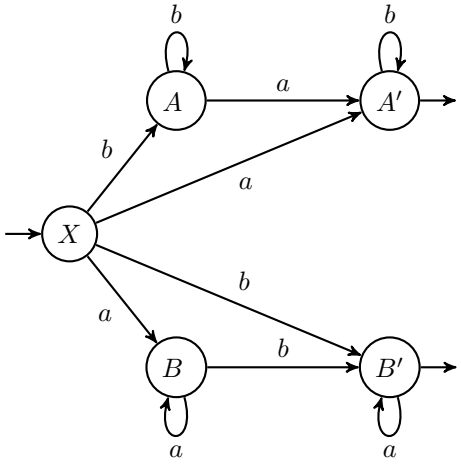
(7) (Question bonus, plus longue, à ne traiter qu'en dernier.) Calculer une expression rationnelle qui dénote ce langage complémentaire $\overline{L}$. (Ne pas hésiter à introduire des notations intermédiaires.)

Corrigé. (0) L'automate M est un automate fini non-déterministe à transitions spontanées, ou ε-NFA (le concept d'« automate déterministe à transitions spontanées » n'aurait tout simplement pas de sens).

(1a) Le chemin par les états X, A, A', Y accepte les mots exactement un a, c'est-à-dire le langage dénoté par $b*ab*$. Le chemin par les états X, B, B', Y accepte les mots comportant exactement un b, c'est-à-dire le langage dénoté par $a*ba*$. L'automate M dans son ensemble accepte les mots comportant exactement un a ou (inclusif) exactement un b (i.e. $L = \{w \in \Sigma^* : |w|_a = 1$ ou $|w|_b = 1\}$ si $|w|_x$ désigne le nombre total d'occurrences de la lettre x dans le mot w). C'est le langage dénoté par l'expression rationnelle $b*ab*|a*ba*$ (nous notons ici et ailleurs $|$ pour la disjonction, qu'on peut aussi noter $+$).

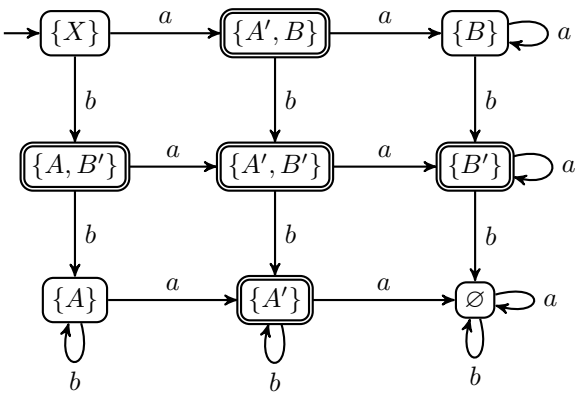
(1b) Parmi les mots proposés, a, b, ab et aab appartiennent à L, tandis que ε, aa, $aabb$, $abab$ et $ababa$ n'y appartiennent pas.

(2) La ε-fermeture (arrière) de l'état X est $\{X, A, B\}$; la ε-fermeture de l'état A' est $\{A', Y\}$ et celle de l'état B' est $\{B', Y\}$; les autres états sont leur propre ε-fermeture (i.e., celle-ci est un singleton). L'élimination des transitions spontanées conduit donc à l'automate M_2 suivant :

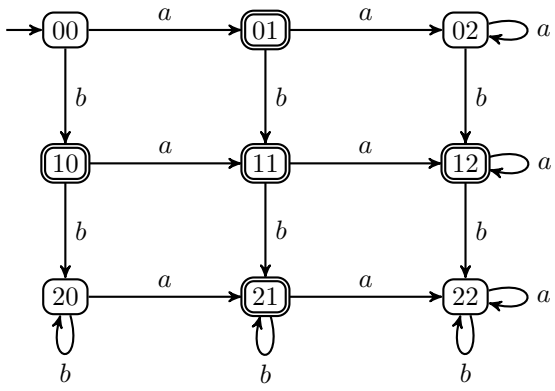


(On a supprimé l'état Y qui est devenu inutile car aucune transition non-spontanée n'y conduit.)

(3) L'algorithme de déterminisation conduit à l'automate M_3 suivant où, pour plus de lisibilité, les états finaux ont été marqués en les entourant deux fois plutôt que par une flèche sortante :



Pour la commodité de la suite de la correction, on renomme les états de cet automate M_3 de la façon suivante :



Ici, l'état 0• signifie que l'automate n'a pas rencontré de a, l'état 1• qu'il en a rencontré exactement un, et l'état 2• qu'il en a rencontré au moins deux ; les états •0, •1 et •2 ont la même signification pour la lettre b.

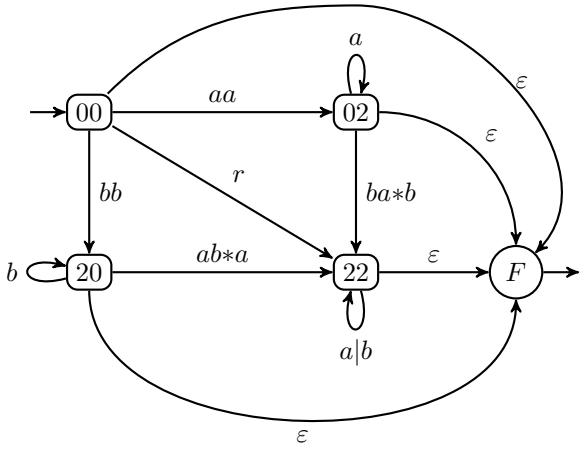
(4) L'automate M_3 est déjà minimal. En effet, l'algorithme de minimisation commence par séparer les classes $\{01, 10, 11, 12, 21\}$ (états finaux) et $\{00, 02, 20, 22\}$ (états non-finaux) ; ensuite, la transition étiquetée par a sépare la classe $\{01, 10, 11, 12, 21\}$ en $\{01, 21\}$ (qui vont vers un état non-final) et $\{10, 11, 12\}$ (qui vont vers un état final), et la classe $\{00, 02, 20, 22\}$ en $\{00, 20\}$ (qui vont vers un état final) et $\{02, 22\}$ (qui vont vers un non-final). La transition étiquetée par b sépare ensuite en deux chacune des trois classes $\{00, 20\}$, $\{01, 21\}$ et $\{02, 22\}$ (car le premier élément va dans la classe $\{10, 11, 12\}$ tandis que le second reste dans la même classe) et sépare en trois la classe $\{10, 11, 12\}$. On a donc séparé chacun des états.

(5) Pour reconnaître le complémentaire du langage reconnu par un automate fini déterministe complet, il suffit d'échanger états finaux et non-finaux : on peut donc prendre l'automate dessiné en (3) avec, cette fois, la convention que les états simplement entourés sont finaux (et les doublement entourés sont non-finaux). Appelons-le M_5.

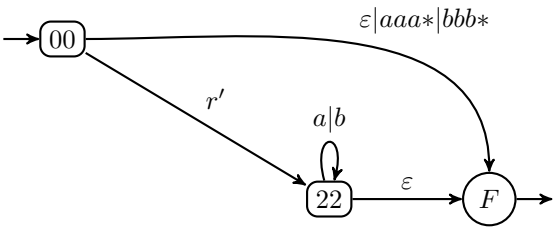
Attention : échanger états finaux et non-finaux ne marche pas pour reconnaître le complémentaire du langage reconnu par un automate non-déterministe ou incomplet (car la négation de « il existe un chemin qui va vers un état final » est « aucun chemin ne va vers un état final » et pas « il existe un chemin qui va vers un état non-final »).

(6) Puisque L est le langage formé des mots comportant exactement un a ou (inclusif) exactement un b, son complémentaire $\overline{L}$ est formé des mots ayant un nombre différent de 1 de a *et* un nombre différent de 1 de b ; si on préfère, il s'agit du langage comportant (0 ou au moins 2 fois la lettre a) *et* (0 ou au moins 2 fois la lettre b).

(7) L'élimination des états n'est pas trop complexe car l'automate M_5 a très peu de boucles. Éliminons simultanément tous les états non-finaux (01, 10, 11, 12 et 21), et profitons-en pour créer un nouvel (et unique) état final F :



où $r := abaa{*}b \,|\, abbb{*}a \,|\, baaa{*}b \,|\, babb{*}a$ (correspondant aux quatre façons de passer de 00 à 22 dans le graphe ci-dessus). Éliminons l'état 02 et l'état 20 :



où $r' := r \,|\, aaa*ba*b \,|\, bbb*ab*a = abaa*b \,|\, abbb*a \,|\, baaa*b \,|\, babb*a \,|\, aaa*ba*b \,|\, bbb*ab*a$. On obtient finalement l'expression rationnelle suivante pour $\overline{L}$:

$$\varepsilon \,|\, aaa* \,|\, bbb* \,|\, (abaa*b \,|\, abbb*a \,|\, baaa*b \,|\, babb*a \,|\, aaa*ba*b \,|\, bbb*ab*a)(a|b)*$$

Pour comprendre cette expression rationnelle, la disjonction de plus haut niveau correspond aux quatre possibilités : (i) 0 fois la lettre a et 0 fois la lettre b, (ii) au moins 2 fois la lettre a et 0 fois la lettre b, (iii) 0 fois la lettre a et au moins 2 fois la lettre b, et (iv) au moins 2 fois la lettre a et au moins 2 fois la lettre b. Pour mieux comprendre l'expression du cas (iv), on peut remarquer que $abaa*b \,|\, baaa*b \,|\, aaa*ba*b$ dénote le langage formé des mots comportant au moins deux a et exactement deux b et qui finissent par un b, et symétriquement $abbb*a \,|\, babb*a \,|\, bbb*ab*a$ dénote le langage formé des mots comportant au moins deux b et exactement deux a et qui finissent par un a : l'expression du cas (iv) correspond donc à écrire un mot ayant au moins deux a et au moins deux b comme le premier préfixe qui vérifie cette propriété suivi d'un suffixe quelconque. (On pouvait utiliser directement ce raisonnement pour produire l'expression.) ✓

Exercice A.1.7.

Dans cet exercice, on pose $\Sigma = \{a, b\}$.

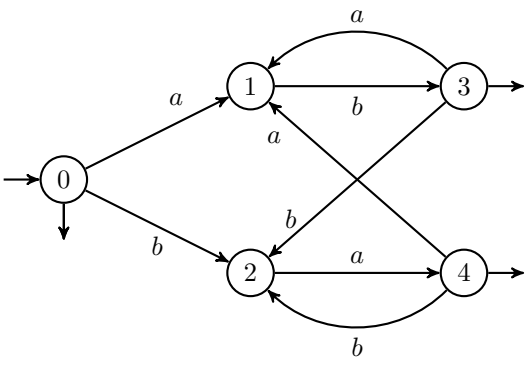
On considère l'expression rationnelle r suivante : $(ab|ba)*$ (sur l'alphabet Σ). On appelle $L := L(r)$ le langage qu'elle dénote.

(0) Donner quatre exemples de mots de L et quatre exemples de mots n'appartenant pas à L.

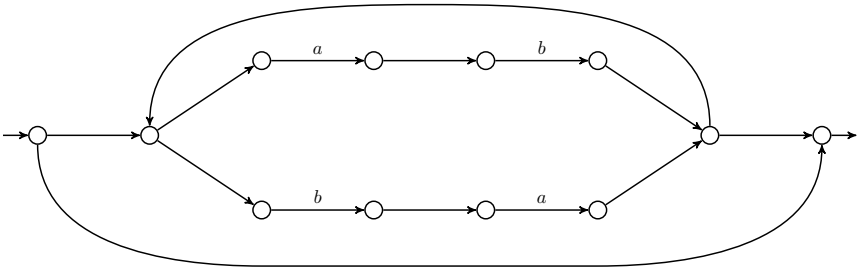
Corrigé. Par exemple, les mots ε, ab, ba et $abba$ appartiennent à L. Les mots a, b, aa et aba n'appartiennent pas à L. ✓

(1) Traiter l'une *ou* l'autre des questions suivantes : (i) construire l'automate de Glushkov $\mathscr{A}_1$ de r ; (ii) construire l'automate de Thompson de r, puis éliminer les transitions spontanées (= ε-transitions) de ce dernier (on retirera les états devenus inutiles) : on appellera $\mathscr{A}_1$ l'automate ainsi obtenu.

Corrigé. L'automate $\mathscr{A}_1$ obtenu est le suivant :



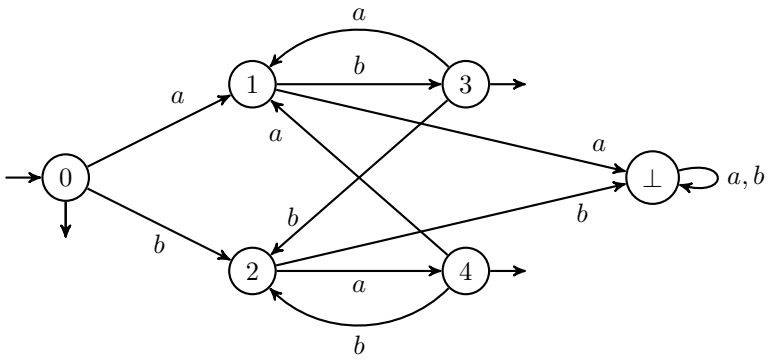
C'est l'automate de Glushkov. Si on commence par construire l'automate de Thompson, on obtient le suivant (à 12 états) :



(où toutes les flèches non étiquetées sont des transitions spontanées, c'est-à-dire qu'il faut les imaginer étiquetées par ε) qui par élimination des transitions spontanées donne celui $\mathscr{A}_1$ qu'on a tracé au-dessus (les seuls états qui subsistent après élimination des transitions spontanées sont l'état initial et les quatre états auxquels aboutissent une transition non spontanée). ✓

(2) À partir de $\mathscr{A}_1$, construire un automate fini déterministe complet reconnaissant L. On appellera $\mathscr{A}_2$ l'automate en question.

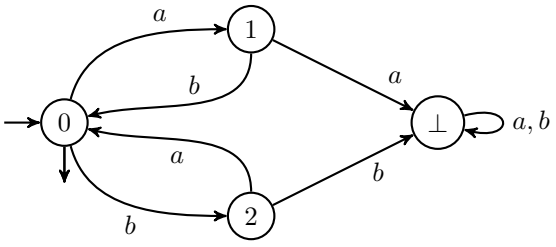
Corrigé. L'automate $\mathscr{A}_1$ est déterministe incomplet : il s'agit donc simplement de lui ajouter un état « puits » pour le rendre complet, ce qui donne l'automate $\mathscr{A}_2$ suivant :



✓

(3) Minimiser l'automate $\mathscr{A}_2$. On appellera $\mathscr{A}_3$ l'automate canonique ainsi obtenu.
(On obtient un automate ayant 4 états.)

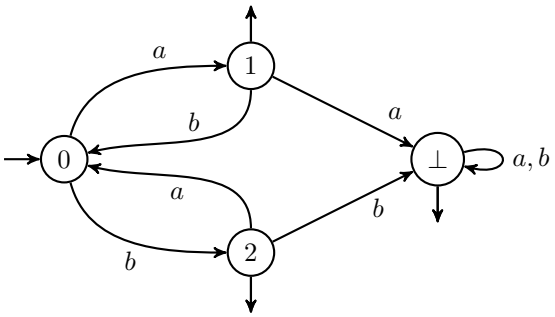
Corrigé. L'algorithme de minimisation identifie les trois états finaux $(0, 3, 4)$ de l'automate $\mathscr{A}_2$, ce qui donne l'automate $\mathscr{A}_3$ suivant (on note 0 pour l'état résultant de l'identification de $0, 3, 4$) :



✓

(4) Construire un automate $\mathscr{A}_4$ reconnaissant le langage $M := \Sigma^* \setminus L$ complémentaire de L.

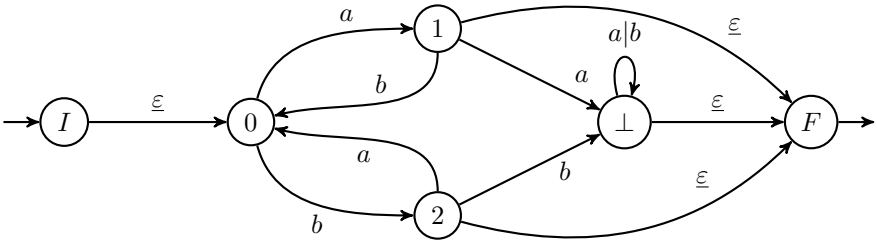
Corrigé. L'automate $\mathscr{A}_3$ étant un automate fini déterministe complet reconnaissant L, il suffit de marquer finaux les états non-finaux et vice versa pour obtenir l'automate $\mathscr{A}_4$ suivant :



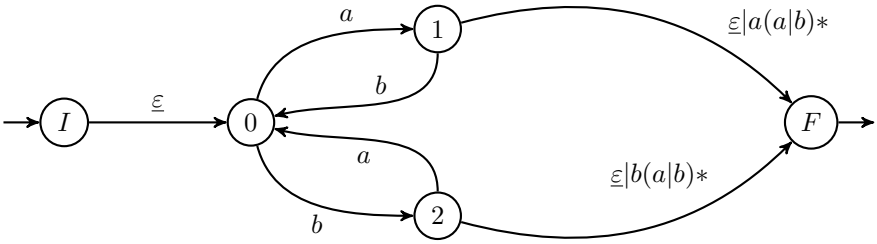
✓

(5) En appliquant à $\mathscr{A}_4$ la méthode d'élimination des états, déterminer une expression rationnelle dénotant le langage M.

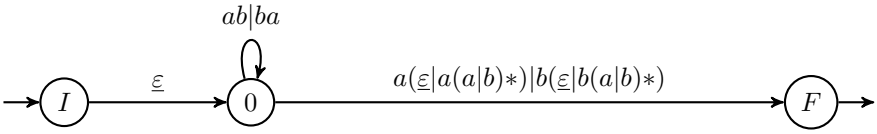
Corrigé. Pour appliquer la méthode d'élimination des états, on commence par modifier l'automate pour avoir un unique état initial I, qui ne soit pas final, et qui n'ait aucune transition qui y aboutisse, et un unique état final F, qui ne soit pas initial, et sans aucune transition qui en part :



On élimine ensuite l'état $\perp$:



Puis les états 1 et 2 peuvent être éliminés simultanément :



Et l'expression rationnelle finalement obtenue est la suivante :

$$(ab|ba)*(a(\underline{\varepsilon}|a(a|b)*)|b(\underline{\varepsilon}|b(a|b)*))$$

On pouvait aussi donner, entre autres choses :

$$(ab|ba)*(a|aa(a|b)*|b|bb(a|b)*)$$

(obtenue en éliminant les états 1 et 2 avant $\perp$). ✓

A.2 Langages algébriques et grammaires hors contexte

Exercice A.2.1.

Considérons le fragment simplifié suivant de la grammaire d'un langage de programmation hypothétique :

$$\begin{aligned}
\textit{Instruction} &\to \texttt{foo} \mid \texttt{bar} \mid \texttt{qux} \mid \textit{Conditional} \\
&\mid \texttt{begin}\ \textit{InstrList}\ \texttt{end} \\
\textit{Conditional} &\to \texttt{if}\ \textit{Expression}\ \texttt{then}\ \textit{Instruction}\ \texttt{else}\ \textit{Instruction} \\
&\mid \texttt{if}\ \textit{Expression}\ \texttt{then}\ \textit{Instruction} \\
\textit{InstrList} &\to \textit{Instruction} \mid \textit{Instruction}\ \textit{InstrList} \\
\textit{Expression} &\to \texttt{true} \mid \texttt{false} \mid \texttt{happy} \mid \texttt{trippy}
\end{aligned}$$

(Ici, les « lettres » ou tokens ont été écrits comme des mots, par exemple `foo` est une « lettre » : les terminaux sont écrits en police à espacement fixe tandis que les nonterminaux sont en italique et commencent par une majuscule. On prendra *Instruction* pour axiome.)

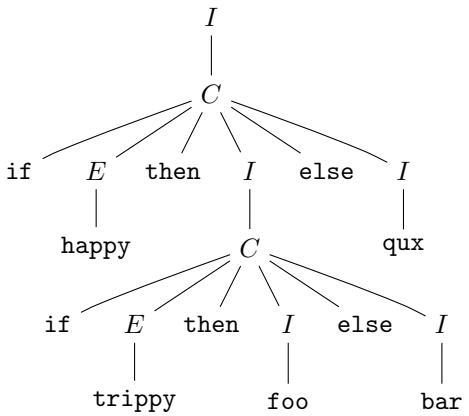
(1) Donner l'arbre d'analyse de : `if happy then if trippy then foo else bar else qux` ; expliquer brièvement pourquoi il n'y en a qu'un.

(2) Donner deux arbres d'analyse distincts de : `if happy then if trippy then foo else bar`. Que peut-on dire de la grammaire présentée ?

(3) En supposant que, dans ce langage, `begin` I `end` (où I est une instruction) a le même effet que I seul, comment un programmeur peut-il réécrire l'instruction considérée en (2) pour obtenir un comportant équivalent à l'une ou l'autre des deux interprétations ?

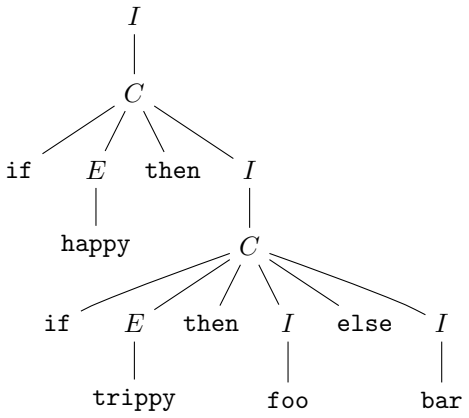
(4) Modifier légèrement la grammaire proposée de manière à obtenir une grammaire faiblement équivalente dans laquelle seul l'un des arbres d'analyse obtenus en (2) est possible (i.e., une grammaire qui force cette interprétation-là par défaut) ; on pourra être amené à introduire des nouveaux nonterminaux pour des variantes de *Instruction* et *Conditional* qui interdisent récursivement les conditionnelles sans `else`.

Corrigé. (1) L'arbre d'analyse de `if happy then if trippy then foo else bar else qux` est le suivant (en notant I, C et E pour *Instruction*, *Condition* et *Expression* respectivement) :

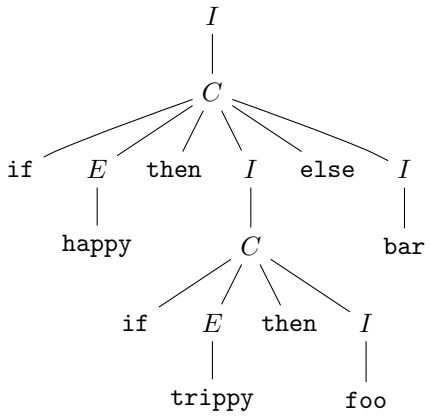


Il est le seul possible car une fois acquis que les deux `if` comportent chacun un `else`, il se construit ensuite en descendant de façon unique (l'instruction est forcément une condition, qui s'analyse en `if` E `then` I `else` I de façon unique, et chacun des morceaux s'analyse de nouveau de façon unique).

(2) Un arbre d'analyse possible consiste à associer le `else bar` avec `if trippy then foo` :



un autre consiste à associer le `else bar` avec `if happy then` ... :



La grammaire présentée est donc ambiguë.

(3) Pour forcer la première interprétation (le `else bar` se rapporte au `if trippy`), on peut écrire : `if happy then begin if trippy then foo else bar end`.

Pour forcer la seconde interprétation (le `else bar` se rapporte au `if happy`), on peut écrire : `if happy then begin if trippy then foo end else bar`.

(4) Pour forcer la première interprétation (le `else` se rapporte au `if` le plus proche possible), on peut modifier la grammaire comme suit :

$$
\begin{aligned}
\mathit{Instruction} &\rightarrow \texttt{foo} \mid \texttt{bar} \mid \texttt{qux} \mid \mathit{Conditional} \\
&\mid \ \texttt{begin}\ \mathit{InstrList}\ \texttt{end} \\
\mathit{InstrNoSC} &\rightarrow \texttt{foo} \mid \texttt{bar} \mid \texttt{qux} \mid \mathit{CondNoSC} \\
&\mid \ \texttt{begin}\ \mathit{InstrList}\ \texttt{end} \\
\mathit{Conditional} &\rightarrow \texttt{if}\ \mathit{Expression}\ \texttt{then}\ \mathit{InstrNoSC}\ \texttt{else}\ \mathit{Instruction} \\
&\mid \ \texttt{if}\ \mathit{Expression}\ \texttt{then}\ \mathit{Instruction} \\
\mathit{CondNoSC} &\rightarrow \texttt{if}\ \mathit{Expression}\ \texttt{then}\ \mathit{InstrNoSC}\ \texttt{else}\ \mathit{InstrNoSC} \\
\mathit{InstrList} &\rightarrow \mathit{Instruction} \mid \mathit{Instruction}\ \mathit{InstrList} \\
\mathit{Expression} &\rightarrow \texttt{true} \mid \texttt{false} \mid \texttt{happy} \mid \texttt{trippy}
\end{aligned}
$$

L'idée est d'obliger une instruction conditionnelle qui apparaîtrait après le `then` d'une conditionnelle complète à être elle-même complète (elle ne peut pas être courte, car alors le `else` devrait se rattacher à elle), et ce, récursivement. On peut montrer que la grammaire ci-dessus est inambiguë et faiblement équivalente à celle de départ.

On peut aussi fabriquer une grammaire inambiguë, faiblement équivalente à celle de départ, qui force l'autre interprétation (le `else` se rapporte au `if` le plus lointain possible), mais c'est nettement plus complexe (l'idée

générale pour apparier un `else` avec un `if`...`else` dans cette logique est de demander que *soit* le `else` n'est suivi d'aucun autre `else`, *soit* toute instruction conditionnelle entre le `then` et le `else` est elle-même complète). Contrairement à la grammaire précédente, cette grammaire, bien qu'inambiguë, est probablement impossible à analyser avec un analyseur LR (ou même, déterministe). ✓

Exercice A.2.2.

Soit $\Sigma = \{a, b\}$. On considère le langage M des mots qui *ne s'écrivent pas* sous la forme ww avec $w \in \Sigma^*$ (c'est-à-dire sous la forme d'un carré ; autrement dit, le langage M est le *complémentaire* du langage Q des carrés considéré dans l'exercice A.2.3) : par exemple, M contient les mots a, b, ab, aab et $aabb$ mais pas ε, aa, $abab$ ni $abaaba$.

(0) Expliquer pourquoi tout mot sur Σ de longueur impaire est dans M, et pourquoi un mot $x_1 \cdots x_{2n}$ de longueur paire $2n$ est dans M si et seulement si il existe i tel que $x_i \neq x_{n+i}$.

On considère par ailleurs la grammaire hors contexte G (d'axiome S)

$$\begin{aligned} S &\to A \mid B \mid AB \mid BA \\ A &\to a \mid aAa \mid aAb \mid bAa \mid bAb \\ B &\to b \mid aBa \mid aBb \mid bBa \mid bBb \end{aligned}$$

(1) Décrire le langage $L(G, A)$ des mots dérivant de A dans la grammaire G (autrement dit, le langage engendré par la grammaire identique à G mais ayant pour axiome A). Décrire de même $L(G, B)$.

(2) Montrer que tout mot de longueur impaire est dans le langage $L(G)$ engendré par G.

(3) Montrer que tout mot $t \in M$ de longueur paire est dans $L(G)$. (Indication : si $t = x_1 \cdots x_{2n}$ est de longueur paire $2n$ et que $x_i \neq x_{n+i}$, on peut considérer la factorisation de t en $x_1 \cdots x_{2i-1}$ et $x_{2i} \cdots x_{2n}$.)

(4) Montrer que tout mot de $L(G)$ de longueur paire est dans M.

(5) En déduire que M est algébrique.

Corrigé. (0) En remarquant que si $n = |w|$ alors $|ww| = 2n$, on constate que tout mot de la forme ww est de longueur paire, et de plus, que pour un mot de longueur $2n$, être de la forme ww signifie que son préfixe de longueur n soit égal à son suffixe de longueur n ; c'est-à-dire, si $t = x_1 \cdots x_{2n}$, que $x_1 \cdots x_n = x_{n+1} \cdots x_{2n}$, ce qui signifie exactement $x_i = x_{n+i}$ pour tout $1 \leq i \leq n$.

(1) La règle $A \to a \mid aAa \mid aAb \mid bAa \mid bAb$ permet de faire à partir de A une dérivation qui ajoute un nombre quelconque de fois une lettre (a ou b) de chaque part de A, et finalement remplace ce A par a. On obtient donc ainsi exactement les mots de longueur impaire ayant un a comme lettre centrale : $L(G, A) = \{w_1 a w_2 : |w_1| = |w_2|\}$. De même, $L(G, B) = \{w_1 b w_2 : |w_1| = |w_2|\}$.

(2) Tout mot de longueur impaire est soit dans $L(G, A)$ soit dans $L(G, B)$ selon que sa lettre centrale est un a ou un b. Il est donc dans $L(G)$ en vertu des règles $S \to A$ et $S \to B$.

(3) Soit $t = x_1 \cdots x_{2n}$ un mot de M de longueur paire $2n$: d'après (0), il existe i tel que $x_i \neq x_{n+i}$. Posons alors $u = x_1 \cdots x_{2i-1}$ et $v = x_{2i} \cdots x_{2n}$. Chacun de u et de v est de longueur impaire. De plus, leurs lettres centrales sont respectivement x_i et x_{n+i}, et elles sont différentes : l'une est donc un a et l'autre un b ; mettons sans perte de généralité que $x_i = a$ et $x_{n+i} = b$. Alors $u \in L(G, A)$ d'après (1) et $v \in L(G, B)$: le mot $t = uv$ s'obtient donc par la règle $S \to AB$ (suivie de dérivations de u à partir de A et de v a partir de B) : ceci montre bien $t \in L(G)$.

(4) On a vu en (1) que tout mot dérivant de A ou de B est de longueur impaire. Un mot t de $L(G)$ de longueur paire $2n$ dérive donc forcément de AB ou de BA. Sans perte de généralité, supposons qu'il dérive de AB, et on veut montrer qu'il appartient à M. Appelons u le facteur de t qui dérive de A et v le facteur de t qui dérive de B : on sait alors (toujours d'après (1)) que u s'écrit sous la forme $x_1 \cdots x_{2i-1}$ où la lettre centrale x_i vaut a, et que v s'écrit sous la forme (quitte à continuer la numérotation des indices) $x_{2i} \cdots x_{2n}$ où la lettre centrale x_{n+i} vaut b. Alors $x_{n+i} \neq x_i$ donc le mot t est dans M d'après (0).

(5) On a $M = L(G)$ car d'après les questions précédentes, tout mot de longueur impaire est dans les deux et qu'un mot de longueur paire est dans l'un si et seulement si il est dans l'autre. On a donc montré que M est algébrique. ✓

Exercice A.2.3.

Soit $\Sigma = \{a, b\}$. Montrer que le langage $Q := \{ww : w \in \Sigma^*\}$ constitué des mots de la forme ww (autrement dit, des carrés ; par exemple, ε, aa, $abab$, $abaaba$ ou encore $aabbaabb$ sont dans Q) n'est pas algébrique. On pourra pour cela considérer son intersection avec le langage L_0 dénoté par l'expression rationnelle $a*b*a*b*$ et appliquer le lemme de pompage.

Corrigé. Supposons par l'absurde que Q soit algébrique : alors son intersection avec le langage rationnel $L_0 = \{a^m b^n a^{m'} b^{n'} : m, n, m', n' \in \mathbb{N}\}$ est encore algébrique. Or $Q \cap L_0 = \{a^m b^n a^m b^n : m, n \in \mathbb{N}\}$. On va maintenant utiliser le lemme de pompage pour arriver à une contradiction.

Appliquons le lemme de pompage pour les langages algébriques au langage $Q \cap L_0 = \{a^m b^n a^m b^n : m, n \in \mathbb{N}\}$ considéré : appelons k l'entier dont le lemme de pompage garantit l'existence. Considérons le mot $t := a^k b^k a^k b^k$: dans la suite de cette démonstration, on appellera « bloc » de t un des quatre facteurs a^k, b^k, a^k et b^k. D'après la propriété de k garantie par le lemme de pompage, il doit exister une factorisation $t = uvwxy$ pour laquelle on a (i) $|vx| \geq 1$, (ii) $|vwx| \leq k$ et (iii) $uv^i wx^i y \in Q \cap L_0$ pour tout $i \geq 0$.

Chacun de v et de x doit être contenu dans un seul bloc, i.e., doit être de la forme a^ℓ ou b^ℓ, sinon sa répétition (v^i ou x^i pour $i \geq 2$, qui appartient à L_0 d'après (iii)) ferait apparaître plus d'alternations entre a et b que le langage L_0 ne le permet. Par ailleurs, la propriété (ii) assure que le facteur vwx ne peut rencontrer qu'un ou deux blocs de t (pas plus). Autrement dit, v et x sont contenus dans deux blocs de t qui sont identiques ou bien consécutifs [31].

D'après la propriété (i), au moins l'un de v et de x n'est pas le mot vide. Si ce facteur non trivial est dans le premier bloc a^k, l'autre ne peut pas être dans l'autre bloc a^k d'après ce qui vient d'être dit : donc $uv^i wx^i y$ est de la forme $a^{k'} b^n a^k b^k$ avec $k' > k$ si $i > 1$, qui n'appartient pas à $Q \cap L_0$, une contradiction. De même, le facteur non trivial est dans le premier bloc b^k, l'autre ne peut pas être dans l'autre bloc b^k : donc $uv^i wx^i y$ est de la forme $a^m b^{k'} a^{m'} b^k$ avec $k' > k$ si $i > 1$, qui n'appartient pas à $Q \cap L_0$, de nouveau une contradiction. Les deux autres cas sont analogues. ✓

Remarque : Les exercices A.2.2 et A.2.3 mis ensemble donnent un exemple explicite d'un langage M algébrique dont le complémentaire Q n'est pas algébrique.

Exercice A.2.4.

On considère la grammaire hors contexte G suivante (d'axiome S) sur l'alphabet $\Sigma = \{a, b\}$:

$$S \to aSS \mid b$$

Soit $L = L(G)$ le langage algébrique qu'elle engendre.

(0) Donner quelques exemples de mots dans L.

(1) Expliquer pourquoi un $w \in \Sigma^*$ appartient à L si et seulement si : soit $w = b$, soit il existe $u, v \in L$ tels que $w = auv$.

(2) En déduire par récurrence sur la longueur $|w|$ de w que si $w \in L$ alors on a $wz \notin L$ pour tout $z \in \Sigma^+$ (c'est-à-dire $z \in \Sigma^*$ et $z \neq \varepsilon$). Autrement dit : en ajoutant des lettres à la fin d'un mot de L on obtient un mot qui n'appartient jamais à L. (Indication : si on a $auvz = au'v'$ on pourra considérer les cas (i) $|u'| > |u|$, (ii) $|u'| < |u|$ et (iii) $|u'| = |u|$.)

(3) En déduire que si $auv = au'v'$ avec $u, v, u', v' \in L$ alors $u = u'$ et $v = v'$. (Indication analogue.)

(4) En déduire que G est inambiguë, c'est-à-dire que chaque mot $w \in L$ a un unique arbre d'analyse pour G (on pourra reprendre l'analyse de la question (1) et procéder de nouveau par récurrence sur $|w|$).

31. La formulation est choisie pour avoir un sens même si v ou x est le mot vide (ce qui est possible *a priori*).

(5) En s'inspirant des questions précédentes, décrire un algorithme simple (en une seule fonction récursive) qui, donné un mot $w \in \Sigma^*$ renvoie la longueur du préfixe de w appartenant à L s'il existe (il est alors unique d'après la question (2)) ou bien « échec » s'il n'existe pas ; expliquer comment s'en servir pour décider si $w \in L$ (i.e., écrire une fonction qui répond vrai ou faux selon que $w \in L$ ou $w \notin L$).

Corrigé. (0) Quelques exemples de mots dans L sont b, abb, $aabbb$, $ababb$ ou encore $aabbabb$.

(1) Si $w \in L$, considérons un arbre d'analyse $\mathscr{W}$ de w pour G : sa racine, étiquetée S, doit avoir des fils étiquetés selon l'une des deux règles de la grammaire, soit $S \to b$ ou bien $S \to aSS$, autrement dit, soit elle a un unique fils étiqueté b, soit elle a trois fils étiquetés respectivement a, S, S. Dans le premier cas, le mot w est simplement b ; dans le second, les sous-arbres $\mathscr{U}, \mathscr{V}$ ayant pour racine les deux fils étiquetés S sont encore des arbres d'analyse pour G, et si on appelle u et v les mots dont ils sont des arbres d'analyse (c'est-à-dire, ceux obtenus en lisant les feuilles de $\mathscr{U}$ et $\mathscr{V}$ respectivement par ordre de profondeur), alors on a $w = auv$ et $u, v \in L$ (puisqu'ils ont des arbres d'analyse pour G).

La réciproque est analogue : le mot b appartient trivialement à L, et si $u, v \in L$, ils ont des arbres d'analyse $\mathscr{U}, \mathscr{V}$ pour G, et on peut fabriquer un arbre d'analyse pour $w := auv$ qui a une racine étiquetée S ayant trois fils étiquetés a, S, S, ces deux derniers ayant pour descendants des sous-arbres donnés par $\mathscr{U}$ et $\mathscr{V}$.

(2) Montrons par récurrence sur $|w|$ que si $w \in L$ alors on a $wz \notin L$ pour tout $z \in \Sigma^+$. La récurrence permet de supposer la conclusion déjà connue pour tout mot de longueur $< |w|$. D'après la question (1), le mot w est soit b soit de la forme auv avec $u, v \in L$ et trivialement $|u| < |w|$ et $|v| < |w|$. Si $w = b$, il est évident qu'aucun mot de la forme bz ne peut appartenir à L (la question (1) montre que les seuls mots de L sont le mot b et des mots commençant par a). Il reste le cas $w = auv$: on veut montrer que wz, c'est-à-dire $auvz$, n'appartient pas à L. Mais s'il y a appartenait, toujours d'après la question (1), il serait de la forme $au'v'$ (le cas b étant trivialement exclu), où $u', v' \in L$; notamment, $uvz = u'v'$. Distinguons trois cas : (i) soit $|u'| > |u|$, mais alors u est un préfixe strict de u', c'est-à-dire que u' peut s'écrire sous la forme $u' = ut$ pour un $t \in \Sigma^+$, et par l'hypothèse de récurrence, on a $u' \notin L$, une contradiction ; (ii) soit $|u'| < |u|$, mais alors u' est un préfixe strict de u, c'est-à-dire que u peut s'écrire sous la forme $u = u't$ pour un $t \in \Sigma^+$, et par l'hypothèse de récurrence (puisque $|u'| < |u| < |w|$), on a $u \notin L$, de nouveau une contradiction ; (iii) soit $|u'| = |u|$, donc $u' = u$ (puisqu'ils sont préfixes de même longueur du même mot $uvz = u'v'$), et on a alors $v' = vz$, mais comme $v \in L$, l'hypothèse de récurrence entraîne $v' \notin L$, encore une contradiction.

(3) Montrons que si $auv = au'v'$ avec $u, v, u', v' \in L$ alors $u = u'$ et $v = v'$. On a notamment $uv = u'v'$. Distinguons trois cas : (i) soit $|u'| > |u|$, mais alors u est un préfixe strict de u', c'est-à-dire que u' peut s'écrire sous la forme $u' = ut$ pour un $t \in \Sigma^+$, et par la question (2), on a $u' \notin L$, une contradiction ; (ii) soit $|u'| < |u|$, mais alors u' est un préfixe strict de u, c'est-à-dire que u peut s'écrire sous la forme $u = u't$ pour un $t \in \Sigma^+$, et par la question (2), on a $u \notin L$, de nouveau une contradiction ; (iii) soit $|u'| = |u|$, donc $u' = u$ (puisqu'ils sont préfixes de même longueur du même mot $uv = u'v'$), et on a alors $v' = v$, la conclusion annoncée.

(4) Soit $w \in L$: on veut montrer qu'il a un unique arbre d'analyse pour G. On procède par récurrence sur $|w|$, ce qui permet de supposer la conclusion connue pour tout mot de longueur $< |w|$. Comme on l'a expliqué en (1), il y a deux possibilités pour un arbre d'analyse de w : soit la racine a un unique fils étiqueté b et le mot analysé est $w = b$, soit la racine a trois fils étiquetés a, S, S, et des deux derniers fils partent des arbres d'analyse $\mathscr{U}, \mathscr{V}$ de mots $u, v \in L$ tels que $w = auv$. Ces deux cas sont évidemment incompatibles : il reste donc simplement à expliquer que dans le dernier, $\mathscr{U}$ et $\mathscr{V}$ sont uniquement déterminés. Or la question (3) assure que u, v (tels que $w = auv$) sont uniquement déterminés, et l'hypothèse de récurrence permet de conclure (comme $|u| < |w|$ et $|v| < |w|$) que les arbres d'analyse $\mathscr{U}$ et $\mathscr{V}$ de u et v sont uniquement déterminés, comme on le voulait.

(5) Donné un mot $w \in \Sigma^*$, la fonction « rechercher préfixe dans L » suivante renvoie la longueur du préfixe de w appartenant à L, ou bien « échec » si ce préfixe n'existe pas :

- si $w = \varepsilon$, renvoyer échec,
- si la première lettre de w est b, renvoyer 1,
- sinon (la première lettre de w est a), soit x le suffixe de w correspondant (c'est-à-dire $w = ax$, ou si on préfère, x enlève la première lettre de w),
- appeler la fonction elle-même (rechercher préfixe dans L) sur x :
- si elle échoue, renvoyer échec,
- si elle retourne k, soit u le préfixe de x de longueur k, et y le suffixe correspondant (c'est-à-dire $x = uy$, ou si on préfère, y enlève les k premières lettres de x),
- appeler la fonction elle-même (rechercher préfixe dans L) sur y :

— si elle échoue, renvoyer échec,

— si elle retourne ℓ, retourner $1+k+\ell$ (en effet, on a $w = auvz$ où u est de longueur k et v est de longueur ℓ).

Pour savoir si un mot appartient à w, il s'agit simplement de vérifier que la valeur retournée (=la longueur du préfixe appartenant à L) n'est pas un échec et est égale à la longueur $|w|$.

La terminaison de cet algorithme est claire par récurrence sur la longueur (chaque appel récursif est fait sur un mot de longueur strictement plus courte), et sa correction est garantie par les questions précédentes : les cas b et auv sont disjoints et dans le dernier cas, u et v sont uniquement déterminés (c'est ce qu'affirme la non-ambiguïté de la grammaire).

(Il s'agit ici du cas le plus simple d'un analyseur LL, et l'algorithme présenté ci-dessus est essentiellement un analyseur LL(1) camouflé sous forme d'analyseur par descente récursive.) ✓

Exercice A.2.5.

On considère la grammaire hors-contexte G d'axiome S et de nonterminaux $N = \{S, T, U, V\}$ sur l'alphabet $\Sigma = \{\#, @, (,), x, y, z\}$ (soit 7 lettres) donnée par

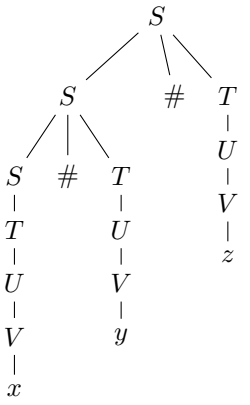
$$\begin{aligned} S &\to T \mid S \,\#\, T \\ T &\to U \mid U \,@\, T \\ U &\to V \mid (\, S \,) \\ V &\to x \mid y \mid z \end{aligned}$$

(On pourra imaginer que $\#$ et $@$ sont deux opérations binaires, les parenthèses servent comme on s'y attend, et x, y, z sont trois opérandes auxquelles on peut vouloir appliquer ces opérations.)

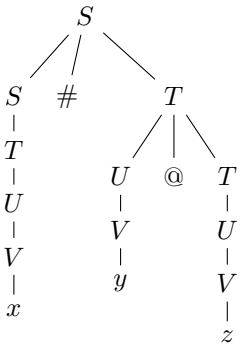
La grammaire en question est inambiguë (on ne demande pas de le démontrer).

(1) Donner les arbres d'analyse (=de dérivation) des mots suivants : (a) $x\#y\#z$, (b) $x\#y@z$, (c) $x \,@\, y \,\#\, z$, (d) $x \,@\, y \,@\, z$, (e) $(x \,\#\, y) \,@\, z$.

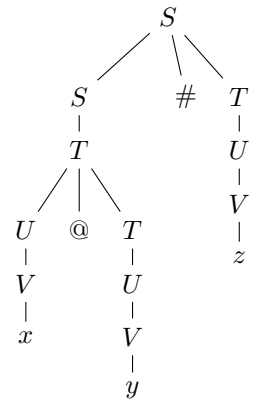
Corrigé. On obtient les arbres d'analyse suivants : (a)



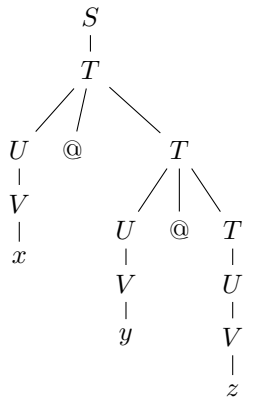
(b)



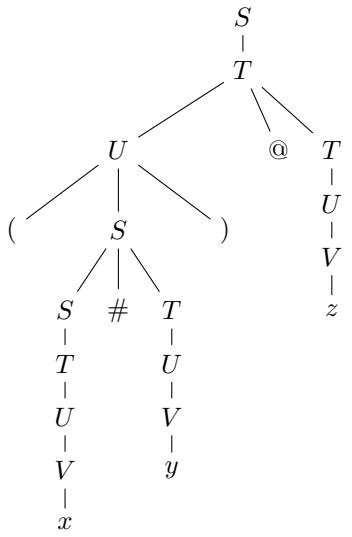
(c)



(d)



(e) ✓



(2) En considérant que (w) a le même effet ou la même valeur que w : (a) L'une des deux opérations $\#$ et $@$ est-elle prioritaire [32] sur l'autre ? Laquelle ? (b) L'opération $\#$ s'associe-t-elle [33] à gauche ou à droite ? (c) L'opération $@$ s'associe-t-elle à gauche ou à droite ?

Corrigé. (a) Les arbres d'analyse (b) et (c) de la question (1) montrent que l'opération $@$ est prioritaire sur $\#$ (elle est toujours plus profonde dans l'arbre, c'est-à-dire appliquée en premier aux opérandes). (b) L'arbre d'analyse (a) de la question (1) montre que $\#$ s'associe à gauche (le $\#$ entre les deux opérandes gauche est plus profond, c'est-à-dire appliqué en premier). (c) Symétriquement, l'arbre d'analyse (d) de la question (1) montre que $@$ s'associe à droite. ✓

(La question (3) est indépendante de (1) et (2). Par ailleurs, on pourra, si on souhaite s'épargner des confusions, choisir d'appeler a la lettre « parenthèse ouvrante » de Σ et b la lettre « parenthèse fermante » afin de les distinguer des parenthèses mathématiques.)

(3) Soit $L = L(G)$ le langage engendré par G. Soit M le langage des mots sur Σ formés d'un certain nombre de parenthèses ouvrantes, puis de la lettre x, puis d'un certain nombre (possiblement différent) de parenthèses fermantes : autrement dit, des mots de la forme « ${(}^{i}\, x\, {)}^{j}$ », où on a noté « ${(}^{i}$ » une succession de i parenthèses ouvrantes et « ${)}^{j}$ » une succession de j parenthèses fermantes (on pourra noter ce mot $a^i x b^j$ si on préfère). (a) Décrire le langage $L \cap M$ (des mots de L qui appartiennent aussi à M). (b) Ce langage $L \cap M$ est-il rationnel ? (c) Le langage L est-il rationnel ?

Corrigé. (a) Cherchons pour quelles valeurs $(i, j) \in \mathbb{N}^2$ le mot « ${(}^{i}\, x\, {)}^{j}$ » de M appartient à L. La dérivation $S \Rightarrow T \Rightarrow U \Rightarrow (S) \Rightarrow (T) \Rightarrow (U) \Rightarrow ((S)) \Rightarrow \cdots \Rightarrow {(}^{i}\, S\, {)}^{i} \Rightarrow {(}^{i}\, T\, {)}^{i} \Rightarrow {(}^{i}\, U\, {)}^{i} \Rightarrow {(}^{i}\, V\, {)}^{i} \Rightarrow {(}^{i}\, x\, {)}^{i}$ montre que c'est le cas si $j = i$, autrement dit s'il y a autant de parenthèses fermantes qu'ouvrantes. Mais réciproquement, la propriété d'avoir autant de parenthèses fermantes qu'ouvrantes est préservée par toutes les productions de G (et est satisfaite par l'axiome), donc est vérifiée par tout mot de $L = L(G)$. On a donc prouvé que le $L \cap M$ est l'ensemble des mots de la forme « ${(}^{i}\, x\, {)}^{i}$ » où $i \in \mathbb{N}$.

(b) Le langage $L \cap M$ constitué des mots de la forme « ${(}^{n}\, x\, {)}^{n}$ » où $i \in \mathbb{N}$ n'est pas rationnel. Cela résulte du lemme de pompage (presque exactement comme l'exemple du langage $\{a^n b^n : i \in \mathbb{N}\}$ donné en cours) : donnons l'argument en remplaçant la parenthèse ouvrante par a et la parenthèse fermante par b pour plus de clarté notationnelle. Appliquons le lemme de pompage pour les langages rationnels au langage $L \cap M = \{a^n x b^n : i \in \mathbb{N}\}$ supposé par l'absurde être rationnel : appelons k l'entier dont le lemme de pompage garantit l'existence. Considérons le mot $t := a^k x b^k$: il doit alors exister une factorisation $t = uvw$ pour laquelle on a (i) $|v| \geq 1$, (ii) $|uv| \leq k$ et (iii) $uv^i w \in L \cap M$ pour tout $i \geq 0$. La propriété (ii) assure que uv est formé d'un

32. On dit qu'une opération binaire $\boxtimes$ est *prioritaire* sur $\boxplus$ lorsque « $u \boxtimes v \boxplus w$ » se comprend comme « $(u \boxtimes v) \boxplus w$ » et « $u \boxplus v \boxtimes w$ » comme « $u \boxplus (v \boxtimes w)$ ».

33. On dit qu'une opération binaire $\boxdot$ *s'associe à gauche* lorsque « $u \boxdot v \boxdot w$ » se comprend comme « $(u \boxdot v) \boxdot w$ », et *s'associe à droite* lorsque « $u \boxdot v \boxdot w$ » se comprend comme « $u \boxdot (v \boxdot w)$ ».

certain nombre de répétitions de la lettre a (car tout préfixe de longueur $\leq k$ de $a^k x b^k$ est de cette forme) ; disons $u = a^\ell$ et $v = a^m$, si bien que $w = a^{k-\ell-m} x b^k$. La propriété (i) donne $m \geq 1$. Enfin, la propriété (iii) affirme que le mot $uv^i w = a^{k+(i-1)m} x b^k$ appartient à $L \cap M$; mais dès que $i \neq 1$, ceci est faux : il suffit donc de prendre $i = 0$ pour avoir une contradiction.

(c) Le langage M est rationnel puisqu'il est dénoté par l'expression rationnelle $a*xb*$ (toujours en notant a la parenthèse ouvrante et b la parenthèse fermante). Si le langage L était rationnel, le langage $L \cap M$ le serait aussi (on sait que l'intersection de deux langages rationnels est rationnelle), ce qui n'est pas le cas d'après la question (b). Le langage L n'est donc pas rationnel. ✓

Exercice A.2.6.

On considère la grammaire hors contexte G suivante (d'axiome S) sur l'alphabet $\Sigma = \{a, b\}$:

$$S \to SaS \mid b$$

Soit $L = L(G)$ le langage algébrique qu'elle engendre.

(0) Donner quelques exemples de mots dans L.

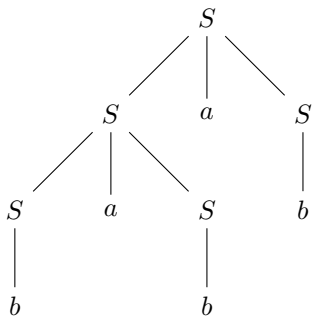
(1) Cette grammaire G est-elle ambiguë ?

(2) Décrire simplement le langage L.

(3) Donner une grammaire inambiguë engendrant le même langage L que G.

Corrigé. (0) Quelques exemples de mots dans L sont b, bab, $babab$, et ainsi de suite (cf. (2)).

(1) La grammaire G est ambiguë : le mot $babab$ a deux arbres d'analyse différents, à savoir



et son symétrique par rapport à l'axe vertical.

(2) Montrons que L est égal au langage $L_{(ba)*b}$ dénoté par l'expression rationnelle $(ba)*b$ comme la question (0) le laisse entrevoir. Pour cela, il suffit de voir que l'ensemble des pseudo-mots (:= mots sur l'ensemble des terminaux et des nonterminaux) qu'on peut dériver de S dans G est le langage $((S|b)a)*(S|b)$ constitué des pseudo-mots de la forme $xaxax \cdots ax$ où chaque x est soit un S soit un b, indépendamment les uns des autres. Or il est clair que remplacer un x (et notamment, remplacer un S) par SaS (qui est de la forme xax) dans un pseudo-mot de cette forme donne encore un pseudo-mot de cette forme : ceci montre qu'une dérivation de G, quelle que soit l'application faite de la règle $S \to SaS$, donne des pseudo-mots de cette forme, donc tout pseudo-mot obtenu par dérivation de S dans la grammaire G est de la forme qu'on vient de dire ; et réciproquement, il est facile de produire n'importe quel nombre n de répéritions du motif aS en appliquant n fois la règle $S \to SaS$ à partir de S, et on peut ensuite librement transformer certains S en b.

(3) La grammaire G' donnée par $S \to baS \mid b$ engendre le même langage que G d'après la question précédente ; et il est évident que cette grammaire est inambiguë : toutes les dérivations sont de la forme $S \Rightarrow baS \Rightarrow babaS \Rightarrow (ba)^n S$ suivie éventuellement de $\Rightarrow (ba)^n b$. ✓

A.3 Introduction à la calculabilité

Exercice A.3.1.

On rappelle la définition du problème de l'arrêt : c'est l'ensemble H des couples[34] (e, x) formés d'un programme (=algorithme) e et d'une entrée x, tels que l'exécution du programme e sur l'entrée x termine en temps fini. On a vu en cours que H était semi-décidable mais non décidable[35].

On considère l'ensemble H' des triplets (e_1, e_2, x) tels que $(e_1, x) \in H$ *ou bien* $(e_2, x) \in H$ (ou les deux). Autrement dit, au moins l'un des programmes e_i termine en temps fini quand on l'exécute sur l'entrée x.

(1) L'ensemble H' est-il semi-décidable ?

Corrigé. Nous allons montrer que H' est semi-décidable.

Considérons l'algorithme suivant : donné en entrée un triplet (e_1, e_2, x), on exécute en parallèle, en fournissant à chacun l'entrée x, les programmes (codés par) e_1 et e_2 (au moyen d'une machine universelle), et en s'arrêtant immédiatement si l'un des deux s'arrête, et on renvoie alors « vrai ». Manifestement, cet algorithme termine (en renvoyant « vrai ») si et seulement si $(e_1, e_2, x) \in H'$, ce qui montre que H' est semi-décidable. ✓

(2) L'ensemble H' est-il décidable ?

Corrigé. Nous allons montrer que H' n'est pas décidable.

Supposons par l'absurde qu'il existe un algorithme T qui décide H'. Soit e' (le code d')un programme quelconque qui effectue une boucle infinie (quelle que soit son entrée) : comme e' ne termine jamais, un triplet (e, e', x) appartient à H' si et seulement si (e, x) appartient à H. Considérons maintenant l'algorithme suivant : donné en entrée un couple (e, x), on applique l'algorithme T supposé exister pour décider si $(e, e', x) \in H'$; comme on vient de l'expliquer, ceci équivaut à $(e, x) \in H$. On a donc fabriqué un algorithme qui décide H, ce qui est impossible, d'où la contradiction annoncée. ✓

Exercice A.3.2.

Soit Σ un alphabet (fini, non vide) fixé. Les questions suivantes sont indépendantes (mais on remarquera leur parallélisme). Ne pas hésiter à décrire les algorithmes de façon succincte et informelle.

(1) Expliquer, au moyen des résultats vus en cours, pourquoi il existe un algorithme A_1 qui, étant donnée une expression rationnelle r sur Σ, décide si le langage L_r dénoté par r est différent du langage Σ^* de tous les mots sur Σ. (Autrement dit, l'algorithme A_1 doit prendre en entrée une expression rationnelle r, terminer en temps fini, et répondre « vrai » s'il existe un mot $w \in \Sigma^*$ tel que $w \notin L_r$ et « faux » si $L_r = \Sigma^*$. On ne demande pas que l'algorithme soit efficace.)

(2) Expliquer pourquoi il existe un algorithme A_2 qui, étant donnée une grammaire hors contexte G sur Σ, « semi-décide » si le langage L_G engendré par G est différent du langage Σ^* de tous les mots. (« Semi-décider » signifie que l'algorithme A_2 doit prendre en entrée une grammaire hors contexte G, terminer en temps fini en répondant « vrai » s'il existe un mot $w \in \Sigma^*$ tel que $w \notin L_G$, et ne pas terminer[36] si $L_G = \Sigma^*$.) Indication : on peut tester tous les mots possibles.

(3) Expliquer pourquoi il existe un algorithme A_3 comme suit : on lui fournit en entrée un algorithme T qui décide un langage $L_T \subseteq \Sigma^*$ (c'est-à-dire que T termine toujours en temps fini quand on lui présente un mot sur Σ, et répond « vrai » ou « faux », et L_T est le langage des mots sur lesquels il répond « vrai »), et l'algorithme A_3 doit semi-décider si L_T est différent

34. Pour être rigoureux, on a fixé un codage permettant de représenter les programmes e, les entrées x à ces programmes, et les couples (e, x), comme des éléments de $\mathbb{N}$ (ou bien de Σ^* sur un alphabet $\Sigma \neq \varnothing$ arbitraire). Il n'est pas nécessaire de faire apparaître ce codage dans la description des algorithmes proposés, qui peut rester informelle.

35. Une variante consiste à considérer l'ensemble des e tels que e termine sur l'entrée e : l'ensemble H qu'on vient de définir s'y ramène facilement.

36. On peut admettre qu'il termine parfois en répondant « faux », mais ce ne sera pas utile.

de Σ^*. (C'est-à-dire que A_3 doit terminer en répondant « vrai » s'il existe un mot $w \in \Sigma^*$ tel que $w \notin L_T$, et ne pas terminer si $L_T = \Sigma^*$.) Indication : la même approche permet de traiter les questions (2) et (3).

(4) Expliquer pourquoi il *n'existe pas* d'algorithme A_4 qui, dans les mêmes conditions que A_3, décide (au lieu de seulement semi-décider) si L_T est différent de Σ^*. (C'est-à-dire que A_4 est censé terminer toujours, et répondre « vrai » s'il existe un mot $w \in \Sigma^*$ tel que $w \notin L_T$, et « faux » si $L_T = \Sigma^*$.) Indication : expliquer comment on pourrait utiliser un tel A_4 pour résoudre le problème de l'arrêt, en cherchant à fabriquer un T qui rejette un mot précisément si un programme donné s'arrête.

Corrigé. (1) Donnée une expression rationnelle r, on sait qu'on peut algorithmiquement fabriquer un automate fini non-déterministe à transitions spontanées qui reconnaît exactement le langage L_r dénoté par r, et ensuite éliminer les transitions spontanées et déterminiser l'automate pour obtenir un automate fini déterministe complet reconnaissant L_r. Sur un tel automate, savoir si $L_r \neq \Sigma^*$ est trivial : dès lors qu'il existe un état q non-final accessible, il existe un mot rejeté par l'automate (i.e., n'appartenant pas à L_r), à savoir le mot lu en suivant les étiquettes d'un chemin quelconque de l'état initial q_0 jusqu'à q, et inversement, si tous les états sont finaux, il est trivial que l'automate accepte tous les mots. (On pouvait aussi minimiser l'automate et le comparer à l'automate minimal trivial qui reconnaît le langage Σ^*.)

(2) On sait qu'il existe un algorithme qui, donnée une grammaire hors-contexte G et un mot w, décide si $w \in L_G$. Pour semi-décider s'il existe un w tel que $w \notin L_G$, il suffit de tester tous les mots possibles : plus exactement, on construit un algorithme A_2 qui effectue une boucle infinie sur tous les $w \in \Sigma^*$ (il est évidemment algorithmiquement faisable d'énumérer tous les mots sur Σ) et, pour chacun, teste si $w \in L_G$, et si ce n'est pas le cas, termine immédiatement en répondant « vrai » (on a trouvé un w n'appartenant pas à L_G), tandis que si c'est le cas, l'algorithme A_2 ne terminera jamais.

(3) On procède exactement comme en (2) : par hypothèse on dispose d'un algorithme T qui, donné un mot w, décide si $w \in L_T$. Pour semi-décider s'il existe un w tel que $w \notin L_T$, il suffit de tester tous les mots possibles : plus exactement, on construit un algorithme A_3 qui effectue une boucle infinie sur tous les $w \in \Sigma^*$ et, pour chacun, teste si $w \in L_T$ (en lançant l'algorithme T qui, par hypothèse, termine toujours), et si ce n'est pas le cas, termine immédiatement en répondant « vrai » (on a trouvé un w n'appartenant pas à L_T), tandis que si c'est le cas, l'algorithme A_3 ne terminera jamais.

(4) Supposons par l'absurde qu'on dispose d'un algorithme A_4 comme on vient de dire, et montrons pour arriver à une contradiction qu'on peut s'en servir pour résoudre le problème de l'arrêt. On se donne donc un algorithme S et une entrée x de S et on cherche à savoir (en utilisant A_4) si S termine sur l'entrée x. Pour cela, on va construire un T auquel appliquer A_4.

Voici une solution possible : donné un mot $w \in \Sigma^*$, le programme T ne considère que la longueur $|w|$ de w, et lance (=simule) l'exécution de S sur l'entrée x pour au plus $|w|$ étapes : si l'exécution termine dans le temps imparti, alors T rejette le mot w, sinon, il l'accepte (dans tous les cas, T termine et répond « vrai » ou « faux », donc il est une entrée légitime à A_4). Cette construction fait que L_T rejette au moins un mot précisément lorsque S termine sur x : si au contraire S ne termine pas sur x, alors $L_T = \Sigma^*$. L'utilisation de A_4 sur T permet donc de savoir algorithmiquement si S termine sur x, ce qui contredit l'indécidabilité du problème de l'arrêt.

Variante de la même idée : on appelle « trace d'exécution » de S sur x un mot w qui code le calcul complet de l'exécution de S sur x (par exemple, si on voit S comme une machine de Turing, l'état courant et le contenu du ruban à chaque étape), du début à l'arrêt. Une telle trace d'exécution existe donc précisément si S termine sur x. Or il est visiblement décidable de savoir si un mot w donné est une trace d'exécution (il suffit de vérifier qu'à chaque étape la machine a bien fait ce qu'elle devait faire). On peut donc écrire un algorithme T qui termine toujours et accepte précisément les mots qui *ne sont pas* une trace d'exécution de S sur x. Le fait que L_T soit différent de Σ^* signifie alors exactement qu'une trace d'exécution existe, donc que S termine sur x. Ainsi l'utilisation de A_4 permet de savoir algorithmiquement si S termine sur x, ce qui contredit l'indécidabilité du problème de l'arrêt. ✓

Exercice A.3.3.

(1) Soit Σ un alphabet (i.e., un ensemble fini). L'ensemble $L = \{u^k : u \in \Sigma^*, k \geq 2\}$ des mots qui sont une puissance k-ième pour un $k \geq 2$ est-il décidable ? Semi-décidable ?

(2) L'ensemble des $e \in \mathbb{N}$ tels que l'exécution du e-ième programme (ou, si on préfère, de

la e-ième machine de Turing), exécuté sur l'entrée 42, termine en au plus 10^{1000+e} étapes est-il décidable ? Semi-décidable ?

(3) L'ensemble des $e \in \mathbb{N}$ tels que l'exécution du e-ième programme (ou, si on préfère, de la e-ième machine de Turing), exécuté sur l'entrée 42, termine en temps fini est-il décidable ? Semi-décidable ? (On pourra montrer qu'on peut y ramener le problème de l'arrêt.)

Corrigé. (1) Si $w = u^k$ pour un certain $u \neq \varepsilon$, alors nécessairement $k \leq |w|$ puisque $|w| = k \cdot |u|$. On dispose donc de l'algorithme suivant pour décider si $w \in L$: si $w = \varepsilon$, retourner vrai immédiatement ; sinon, pour k allant de 2 à $|w|$ et qui divise $|w|$, considérer les k facteurs successifs de w de longueur $|w|/k$ (c'est-à-dire, pour $0 \leq i < k$, le facteur de w de longueur $\ell := |w|/k$ commençant à la position $i\ell$) : s'ils sont tous égaux, renvoyer vrai ; si la boucle termine sans avoir trouvé de k qui convienne, renvoyer faux. Le langage proposé est donc décidable (et *a fortiori* semi-décidable).

(2) Donné un $e \in \mathbb{N}$, la fonction 10^{1000+e} est évidemment calculable. On peut ensuite lancer l'exécution du e-ième programme, sur l'entrée 42, pour au plus ce nombre d'étapes (en utilisant la machine universelle, c'est-à-dire, par exemple, en simulant la e-ième machine de Turing sur une machine de Turing). Si l'exécution termine en le temps imparti, on renvoie vrai, sinon, on renvoie faux : ceci montre que l'ensemble proposé est bien décidable (et *a fortiori* semi-décidable).

(3) L'ensemble A proposé est « presque » le problème de l'arrêt. La différence est que le problème de l'arrêt est l'ensemble des couples (e, n) tels que le e-ième programme termine sur l'entrée n alors qu'ici on a fixé l'entrée à 42. Il s'agit donc de montrer que cette limitation ne rend pas pour autant calculable l'ensemble considéré. Or donnés deux entiers (e, n), on peut fabriquer un programme e' qui prend en entrée une valeur, *ignore* cette valeur, et exécute le e-ième programme sur l'entrée n ; de plus un tel e' se calcule algorithmiquement [37] à partir de e et n. L'exécution du programme e' sur l'entrée 42 (ou n'importe quelle autre entrée) se comporte donc comme l'exécution du programme e sur l'entrée n, et notamment, termine si et seulement si elle termine. Autrement dit, e' appartient à l'ensemble A considéré dans cette question si et seulement si (e, n) appartient au problème de l'arrêt. Comme on vient de dire qu'on peut calculer e' algorithmiquement à partir de (e, n), si l'ensemble A était décidable, le problème de l'arrêt le serait, ce qui n'est pas le cas. Donc A n'est pas décidable. En revanche, A est semi-décidable : il suffit de lancer l'exécution du programme e sur l'entrée 42 et renvoyer vrai si elle termine (si elle ne termine pas, on ne termine pas). ✓

Exercice A.3.4.

Soit Σ un alphabet (i.e., un ensemble fini). On s'intéresse à des langages sur Σ.

(A) Montrer que si deux langages L_1 et L_2 sont décidables, alors $L_1 \cup L_2$ et $L_1 \cap L_2$ et $L_1 L_2$ sont décidables ; montrer que si un langage L est décidable alors L^* est décidable (pour ce dernier, on pourra commencer par chercher, si $w \in \Sigma^*$ est un mot de longueur n, comment énumérer toutes les façons de le factoriser en mots de longueur non nulle).

(B) Montrer que si deux langages L_1 et L_2 sont semi-décidables, alors $L_1 \cup L_2$ et $L_1 \cap L_2$ et $L_1 L_2$ sont semi-décidables ; montrer que si un langage L est semi-décidable alors L^* est semi-décidable.

Corrigé. (A) Supposons qu'on dispose d'algorithmes T_1 et T_2 qui décident L_1 et L_2 respectivement (i.e., donné $w \in \Sigma^*$, l'algorithme T_i termine toujours en temps fini, en répondant oui si $w \in L_i$ et non si $w \notin L_i$).

Pour faire un algorithme qui décide $L_1 \cup L_2$, donné un mot $w \in \Sigma^*$, il suffit de lancer successivement T_1 et T_2 : si l'un des deux répond oui, on répond oui, sinon on répond non (autrement dit, on calcule les valeurs de vérité de $w \in L_1$ et $w \in L_2$ au moyen de T_1 et T_2, et on calcule ensuite leur « ou » logique). De même pour décider $L_1 \cap L_2$, il suffit de lancer successivement T_1 et T_2, si les deux répondent oui on répond oui, sinon on répond non (i.e., on calcule les valeurs de vérité de $w \in L_1$ et $w \in L_2$ au moyen de T_1 et T_2, et on calcule ensuite leur « et » logique).

Pour décider $L_1 L_2$, on effectue une boucle sur toutes les factorisation $w = uv$ de w, c'est-à-dire, une boucle sur toutes les longueurs $0 \leq i \leq |w|$ en appelant à chaque fois u le préfixe de w de longueur i et v le suffixe de w de longueur $|w| - i$, et pour chaque paire (u, v) ainsi trouvée, on utilise T_1 et T_2 pour tester si $u \in L_1$ et $v \in L_2$:

37. Techniquement, on invoque ici le théorème s-m-n ; mais dans les faits, il s'agit essentiellement d'ajouter une ligne « `let n=`(représentation décimale de n) » au début d'un programme, ce qui est certainement faisable.

si c'est le cas, on termine l'algorithme en répondant oui (on a $w = uv \in L_1 L_2$) ; si aucune paire ne convient, on répond non.

L'algorithme pour décider L^* est semblable : il s'agit de tester toutes les manières de factoriser un mot $w \in \Sigma^*$ en facteurs de longueur non nulle. (On peut d'ores et déjà exclure $w = \varepsilon$ car le mot vide appartient de toute façon à L^*.) Si $n = |w| > 0$, on peut effectuer une boucle pour un nombre de facteurs k allant de 1 à n, et, pour chaque k, effectuer k boucles emboîtées pour déterminer les limites des facteurs $u_1, \dots, u_k \in \Sigma^+$ tels que $w = u_1 \cdots u_k$ (il suffit par exemple de faire boucler $i_1, \dots, i_k$ chacun de 1 à n, et lorsque $i_1 + \cdots + i_k = n$, appeler u_j le facteur de w de longueur i_j commençant à la position $i_1 + \cdots + i_{j-1}$). Pour chaque factorisation comme on vient de le dire, on teste si tous les u_i appartiennent à L, et si c'est le cas on renvoie vrai (le mot w appartient à L^*) ; si aucune factorisation ne convient, on renvoie faux.

(Dans l'algorithme qui précède, on a écarté les factorisations faisant intervenir le mot vide, car si w est factorisable en mots de L en faisant intervenir le mot vide, quitte à retirer celui-ci, il est encore factorisable en mots non vides de L.)

(B) Les algorithmes sont très semblables à ceux de la partie (A) si ce n'est qu'il faut tenir compte de la possibilité qu'ils puissent ne pas terminer. À part pour l'intersection, on doit donc les lancer « en parallèle » et pas « en série » : lorsqu'on dira qu'on lance deux algorithmes T et T' « en parallèle », cela signifie qu'on exécute une étape du calcul de T, puis une étape de T', puis de nouveau une de T, et ainsi de suite en alternant entre les deux, jusqu'à ce que l'un termine et renvoie vrai.

Si L_1 et L_2 sont semi-dédicables et si T_1 et T_2 sont des algorithmes qui les « semi-décident » (i.e., T_i termine en temps fini et répond oui si $w \in L_i$, et ne termine pas sinon), pour semi-décider $L_1 \cup L_2$, on lance les deux algorithmes T_1 et T_2 en parallèle sur le même mot w : si l'un d'eux termine, on termine en renvoyant vrai (sinon, bien sûr, on ne termine pas).

Pour semi-décider $L_1 \cap L_2$, en revanche, il n'y a pas de raison de procéder en parallèle : on lance d'abord T_1 sur le mot w à tester : si T_1 termine, on lance ensuite T_2 sur le même mot : si T_2 termine et renvoie vrai, on renvoie vrai ; si l'un des deux algorithmes T_i lancés séquentiellement ne termine pas, bien sûr, le calcul dans son ensemble ne terminera pas.

Pour semi-décider $L_1 L_2$ ou L^*, on procède comme dans le cas (A) en lançant en parallèle les algorithmes pour tester toutes les différentes factorisations possibles $w = uv$ ou bien $w = u_1 \cdots u_k$ (en mots non vides) du mot w. ✓

Exercice A.3.5.

On rappelle qu'une fonction $f : \mathbb{N} \to \mathbb{N}$ est dite *calculable* lorsqu'il existe un algorithme (par exemple, un programme pour une machine de Turing) prenant en entrée un $n \in \mathbb{N}$ qui termine toujours en temps fini et renvoie la valeur $f(n)$. On rappelle qu'une partie E de $\mathbb{N}$ ou de $\mathbb{N}^2$ est dite *décidable* lorsque sa fonction indicatrice est calculable, ou, ce qui revient au même, lorsqu'il existe un algorithme prenant en entrée un élément de $\mathbb{N}$ ou de $\mathbb{N}^2$ qui termine toujours en temps fini et renvoie vrai (1) ou faux (0) selon que l'élément fourni appartient ou non à E. On rappelle enfin qu'une partie E de $\mathbb{N}$ ou de $\mathbb{N}^2$ est dite *semi-décidable* lorsqu'il existe un algorithme prenant en entrée un élément de $\mathbb{N}$ ou de $\mathbb{N}^2$ qui termine toujours en temps fini et renvoie vrai (1) si l'élément fourni appartient à E, et sinon ne termine pas (on peut aussi accepter qu'il termine en renvoyant faux, cela ne change rien).

Soit $f : \mathbb{N} \to \mathbb{N}$: montrer qu'il y a équivalence entre les affirmations suivantes :

1. la fonction f est calculable,
2. le graphe $\Gamma_f := \{(n, f(n)) : n \in \mathbb{N}^2\} = \{(n, p) \in \mathbb{N}^2 : p = f(n)\}$ de f est décidable,
3. le graphe Γ_f de f est semi-décidable.

(Montrer que (3) implique (1) est le plus difficile : on pourra commencer par s'entraîner en montrant que (2) implique (1). Pour montrer que (3) implique (2), on pourra chercher une façon de tester en parallèle un nombre croissant de valeurs de p de manière à s'arrêter si l'une quelconque convient.)

Corrigé. Montrons que (1) implique (2) : si on dispose d'un algorithme capable de calculer $f(n)$ en fonction de n, alors il est facile d'écrire un algorithme capable de décider si $p = f(n)$ (il suffit de calculer $f(n)$ avec l'algorithme supposé exister, de comparer avec la valeur de p fournie, et de renvoyer vrai/1 si elles sont égales, et faux/0 sinon).

Le fait que (2) implique (3) est évident car tout ensemble décidable est semi-décidable.

Montrons que (2) implique (1) même si ce ne sera au final pas utile : supposons qu'on ait un algorithme T qui décide Γ_f (i.e., donnés (n, p), termine toujours en temps fini, en répondant oui si $p = f(n)$ et non si $p \neq f(n)$), et on cherche à écrire un algorithme qui calcule $f(n)$. Pour cela, donné un n, il suffit de lancer l'algorithme T successivement sur les valeurs $(n, 0)$ puis $(n, 1)$ puis $(n, 2)$ et ainsi de suite (c'est-à-dire faire une boucle infinie sur p et lancer T sur chaque couple (n, p)) jusqu'à trouver un p pour lequel T réponde vrai : on termine alors en renvoyant la valeur p qu'on a trouvée, qui vérifie $p = f(n)$ par définition de T.

Reste à montrer que (3) implique (1) : supposons qu'on ait un algorithme T qui « semi-décide » Γ_f (i.e., donnés (n, p), termine en temps fini et répond oui si $p = f(n)$, et ne termine pas sinon), et on cherche à écrire un algorithme qui calcule $f(n)$. Pour cela, on va tester les valeurs $0 \leq p \leq M$ chacune pour M étapes et faire tendre M vers l'infini : plus exactement, on utilise l'algorithme U suivant :

— pour M allant de 0 à l'infini,
  — pour p allant de 0 à M,
    — exécuter l'algorithme T sur l'entrée (n, p) pendant au plus M étapes,
    — s'il termine en renvoyant vrai (1), terminer et renvoyer p (sinon, continuer les boucles).

(Intuitivement, U essaie de lancer l'algorithme T sur un nombre de valeurs de p de plus en plus grand et en attendant de plus en plus longtemps pour voir si l'une d'elles termine.)

Si l'algorithme U défini ci-dessus termine, il renvoie forcément $f(n)$ (puisque l'algorithme T a répondu vrai, c'est que $p = f(n)$, et on renvoie la valeur en question) ; il reste à expliquer pourquoi U termine toujours. Mais la valeur $f(n)$ existe (même si on ne la connaît pas) car la fonction f était supposée définie partout, et lorsque l'algorithme T est lancé sur $(n, f(n))$ il est donc censé terminer en un certain nombre (fini !) d'étapes : si M est supérieur à la fois à $f(n)$ et à ce nombre d'étapes, la valeur $f(n)$ va être prise par p dans la boucle intérieure, et pour cette valeur, l'algorithme T va terminer sur l'entrée (n, p) en au plus M étapes, ce qui assure que U termine effectivement.

L'algorithme U calcule donc bien la fonction f demandée, ce qui prouve (1). ✓

Exercice A.3.6.

Soit $A \subseteq \mathbb{N}$ un ensemble infini. Montrer qu'il y a équivalence entre :

— l'ensemble A est décidable,

— il existe une fonction calculable *strictement croissante* $f : \mathbb{N} \to \mathbb{N}$ telle que $f(\mathbb{N}) = A$.

Corrigé. Supposons A décidable : on va construire f comme indiqué. Plus exactement, on va appeler $f(n)$ le n-ième élément de A par ordre croissant (c'est-à-dire que $f(0)$ est le plus petit élément de A, et $f(1)$ le suivant par ordre de taille, et ainsi de suite ; noter que A est infini donc cette fonction est bien définie). Montrons que f est calculable : donné un entier n, on teste successivement si $0 \in A$ puis $1 \in A$ puis $2 \in A$ et ainsi de suite, à chaque fois en utilisant un algorithme décidant A (qui est censé exister par hypothèse) jusqu'à obtenir n fois la réponse « oui » ; plus exactement :

— initialiser $m \leftarrow 0$,
— pour k allant de 0 à l'infini,
  — interroger l'algorithme qui décide si $k \in A$,
  — s'il répond « oui » :
    — si $m = n$, terminer et renvoyer k,
    — sinon, incrémenter m (c'est-à-dire faire $m \leftarrow m + 1$).

La boucle termine car A est infini.

Réciproquement, supposons f strictement croissante calculable et posons $A = f(\mathbb{N})$: on veut montrer que A est décidable. Or pour décider si $k \in A$, il suffit de calculer successivement $f(0)$, $f(1)$, $f(2)$ et ainsi de suite, et de terminer si $f(n)$ atteint ou dépasse le k fixé : s'il l'atteint, on renvoie vrai (on a trouvé n tel que $f(n) = k$), sinon, on renvoie faux (la valeur k a été sautée par la fonction f et ne sera donc jamais atteinte). L'algorithme est donc explicitement :

— pour n allant de 0 à l'infini,
  — calculer $f(n)$,
  — si $f(n) = k$, renvoyer vrai,

— si $f(n) > k$, renvoyer faux.

La boucle termine car toute fonction strictement croissante $\mathbb{N} \to \mathbb{N}$ est de limite $+\infty$ en l'infini (donc $f(n)$ finit forcément par atteindre ou dépasser k). ✓

Exercice A.3.7.

Soit $S(e, n)$ le nombre d'étapes de l'exécution du e-ième programme (ou, si on préfère, de la e-ième machine de Turing) quand on lui fournit le nombre n en entrée, à supposer que cette exécution termine ; sinon, $S(e, n)$ n'est pas défini.

Soit par ailleurs $M(k)$ le maximum des $S(e, n)$ pour $0 \leq e \leq k$ et $0 \leq n \leq k$ qui soient définis (et 0 si aucun d'eux n'est défini). Autrement dit, il s'agit du plus petit entier supérieur ou égal au nombre d'étapes de l'exécution de l'un des programmes $0 \leq e \leq k$ sur l'un des entiers $0 \leq n \leq k$ en entrée, lorsqu'ils terminent.

Montrer que la fonction M n'est pas calculable (i.e., n'est pas calculable par un algorithme) : on pourra pour cela montrer que la connaissance de M permet de résoudre le problème de l'arrêt. Montrer même qu'*aucune* fonction M' telle que $M'(k) \geq M(k)$ pour tout k n'est calculable. Montrer que même si M' vérifie simplement $M'(k) \geq M(k)$ pour $k \geq k_0$, alors M' n'est pas calculable.

Remarque : La fonction M, ou différentes variantes de celle-ci, s'appelle fonction du « castor affairé ». On peut montrer encore plus fort : si F est une fonction calculable quelconque, alors il existe k_0 tel que $M(k) \geq F(k)$ pour $k \geq k_0$ (autrement dit, la fonction M finit par dépasser n'importe quelle fonction calculable : Radó, 1962, *On Non-Computable Functions*).

Corrigé. Supposons que M soit calculable. On peut alors résoudre le problème de l'arrêt de la manière suivante : donné un algorithme T, de numéro e, et une entrée n à fournir à cet algorithme, pour savoir si T s'arrête, on calcule $M(k)$ où $k = \max(e, n)$, on exécute ensuite l'algorithme T pendant au plus $M(k)$ étapes : s'il termine dans le temps imparti, on répond vrai (il a terminé), sinon, on répond faux (il ne terminera jamais). Cette résolution du problème de l'arrêt est correcte, car si T termine sur l'entrée n, il prendra par définition $S(e, n)$ étapes, avec $0 \leq e \leq k$ et $0 \leq n \leq k$ par définition de k, donc $S(e, n) \leq M(k)$ par définition de $M(k)$: ceci signifie précisément que si T n'a pas terminé en $M(k)$ étapes, il ne terminera jamais.

Exactement le même argument montre que M' n'est pas calculable sous l'hypothèse que $M'(k) \geq M(k)$ pour tout k : s'il l'était, on pourrait exécuter l'algorithme T pendant au plus $M'(k)$ étapes, et comme on a $S(e, n) \leq M(k) \leq M'(k)$, la même démonstration convient.

Enfin, si on suppose seulement $M'(k) \geq M(k)$ pour $k \geq k_0$, la fonction M' n'est toujours pas calculable : en effet, si on suppose par l'absurde qu'elle l'est, la fonction M'' qui à k associe $M'(k)$ si $k \geq k_0$ et $M(k)$ sinon, serait encore calculable puisqu'elle ne diffère de M' qu'en un nombre fini de valeurs, or changer la valeur en un point d'une fonction calculable donne toujours une fonction calculable (même si on « ne connaît pas » la valeur à changer, elle existe, donc l'algorithme modifié existe). Mais d'après le paragraphe précédent, M'' n'est pas calculable puisqu'elle est partout supérieure ou égale à M. ✓

Exercice A.3.8.

Dans cet exercice, on s'intéresse au langage L formé des programmes e (codés, dans une formalisation quelconque de la calculabilité [38], comme des entiers naturels ou comme des mots sur un alphabet fixé sans importance) qui, quel que soit le paramètre n qu'on leur fournit en entrée, terminent en temps fini et retournent la valeur 0 : soit $L = \{e : (\forall n)\varphi_e(n) \downarrow = 0\}$. Si l'on préfère, L est l'ensemble de toutes les façons de coder la fonction constante égale à 0. On appellera aussi M le complémentaire de L. On se demande si L ou M sont semi-décidables.

(1) Thésée et Hippolyte se disputent pour savoir si L est semi-décidable. Thésée pense qu'il l'est, et il tient le raisonnement suivant pour l'expliquer :

38. Par exemple, un langage de programmation (Turing-complet) quelconque.

Pour savoir si un programme e est dans l'ensemble L, il suffit de l'examiner pour vérifier que toutes les instructions qui mettent fin au programme renvoient la valeur 0 : si c'est le cas, on termine en répondait « oui » (c'est-à-dire $e \in L$) ; sinon, on rentre dans une boucle infinie. Ceci fournit un algorithme qui semi-décide L.

Hippolyte, elle, pense que L n'est pas semi-décidable. Son argument est le suivant :

Si L était semi-décidable, je pourrais m'en servir pour résoudre le problème de l'arrêt. En effet, donné un programme e' et une entrée m sur laquelle je cherche à tester l'arrêt de e', je peux fabriquer le programme e qui prend une entrée n, lance l'exécution de e' sur l'entrée m pendant au plus n étapes et à la fin renvoie 1 si ces n étapes ont suffi à terminer l'exécution de e', et 0 sinon. Dire que $e \in L$ signifie que e' ne termine jamais sur m, donc pouvoir semi-décider L permet de résoudre algorithmiquement le problème de l'arrêt, ce qui est impossible.

Qui a raison ? Expliquer précisément quelle est l'erreur (ou les erreurs) commise(s) par le raisonnement incorrect, et détailler les éventuels passages incomplets dans le raisonnement correct.

Corrigé. C'est Hippolyte qui a raison (L n'est pas semi-décidable).

L'argument de Thésée est stupide : une instruction mettant fin au programme avec une valeur autre que 0 pourrait ne jamais être atteinte, et réciproquement, même si toutes les instructions mettant fin au programme renvoyaient 0, il pourrait aussi ne jamais terminer. (Au passage, l'argument de Thésée semblerait même montrer que L est décidable, pas juste semi-décidable.)

L'argument d'Hippolyte, lui, est correct : il montre que si L était semi-décidable, le complémentaire du problème de l'arrêt (l'ensemble des e' qui ne terminent jamais) serait semi-décidable, ce qui n'est pas le cas. (Le complémentaire du problème de l'arrêt n'est pas semi-décidable, car s'il l'était, le problème de l'arrêt serait décidable, vu qu'il est déjà semi-décidable.) Parmi les points qui méritent éventuellement d'être précisés, on peut mentionner : le fait que e se déduise algorithmiquement de e' et m ; et le fait qu'il est algorithmiquement possible de lancer l'exécution d'un programme sur n étapes (peu importe la définition exacte de « étape » tant que chacune termine à coup sûr en temps fini) et tester si elle est bien finie au bout de ce temps. ✓

(2) Achille et Patrocle se disputent pour savoir si M (le complémentaire de L) est semi-décidable. Achille pense qu'il l'est, et il tient le raisonnement suivant pour l'expliquer :

Pour savoir si un programme e est dans l'ensemble M, il suffit de tester successivement les valeurs $\varphi_e(n)$ pour tous les n possibles : si l'on rencontre un n tel que $\varphi_e(n)$ n'est pas 0, alors on termine en répondant « oui » (c'est-à-dire $e \in M$) ; sinon, on ne va jamais terminer, et cela signifie que $e \notin M$. Ceci fournit un algorithme qui semi-décide M.

Patrocle, lui, pense que M n'est pas semi-décidable. Son argument est le suivant :

Si M était semi-décidable, je pourrais m'en servir pour résoudre le problème de l'arrêt. En effet, donné un programme e' et une entrée m sur laquelle je cherche à tester l'arrêt de e', je peux fabriquer le programme e qui prend une entrée n, l'ignore purement et simplement, exécute e' sur l'entrée m et à la fin renvoie 0. Dire que $e \in M$ signifie que e' ne termine pas sur m, donc pouvoir semi-décider M permet de résoudre algorithmiquement le problème de l'arrêt, ce qui est impossible.

Qui a raison ? Expliquer précisément quelle est l'erreur (ou les erreurs) commise(s) par le raisonnement incorrect, et détailler les éventuels passages incomplets dans le raisonnement correct.

Corrigé. C'est Patrocle qui a raison (M n'est pas semi-décidable).

Le raisonnement d'Achille se fonde sur l'idée que le complémentaire de « l'ensemble des programmes qui renvoient toujours la valeur 0 » est « l'ensemble des programmes qui renvoient parfois une valeur autre que 0 »,

ce qui oublie la possibilité que le programme ne termine pas. C'est là la principale erreur (le reste est globalement correct).

L'argument de Patrocle, lui, est correct : il montre que si M était semi-décidable, le complémentaire du problème de l'arrêt (l'ensemble des e' qui ne terminent jamais) serait semi-décidable, ce qui n'est pas le cas. (Le complémentaire du problème de l'arrêt n'est pas semi-décidable, car s'il l'était, le problème de l'arrêt serait décidable, vu qu'il est déjà semi-décidable.) On fait appel à des fonctions constantes, et qui ne peuvent renvoyer que 0 ou ne pas terminer, mais ce n'est pas spécialement problématique. Parmi les points qui méritent éventuellement d'être précisés, on peut mentionner le fait que e se déduise algorithmiquement de e' et m. ✓

Index